Exercise Book

Computer Architecture

1st October 2024

Part I: Instruction Set Architecture

[Exercise 1] Array Comparator

a) Write a function in RISC-V that compares two vectors of 32-bit signed numbers. The function returns `'0'` if at least a pair of elements with the same index in both vectors differ in absolute value by more than 1000 (decimal). Otherwise, the function returns `'1'`.

When the function is called, register `a0` contains the address of the first vector in memory, while register `a1` contains the address of the second vector and register `a2` contains the number of elements in each vector.

Note: to find the absolute value of the difference between two numbers, subtract the smaller from the greater.

b) Briefly discuss when can an overflow occur in your function.

[Solution 1] Array Comparator

a) The code of the solution is given below:

```
prog:
    li t5, 1
    li t6, 1000

loop:
    beq a2, zero, finish
    lw  t2, 0(a0)
    lw  t3, 0(a1)
    slt t4, t3, t2      # if(t3<t2) t4=1
    bne t4, zero, next  # if(t4=1) goto next
    sub t3, t3, t2
    j   next2

next:
    sub t3, t2, t3

next2:
    slt  t4, t6, t3     # if(t3>1000) t4=1
    bne  t4, zero, err  # if(t4!=0) goto err
    addi a0, a0, 4
    addi a1, a1, 4
    addi a2, a2, -1
    j    loop

err:
    add t5, zero, zero

finish:
    mv a0, t5
    ret
```

b) An overflow can occur when a negative value is subtracted from a positive one and the difference is greater than the maximal positive value.

Example: `0000 0000 - 8000 0000; 7FFF FFFF - FFFF FFFF;`

[Exercise 2] Understanding RISC-V

Consider the following RISC-V program:

```
    add  t0, a0, zero
    add  t1, a1, zero
    add  t2, a2, a2
    add  t2, t2, t2
    add  t3, t0, t2
loop:
    lw   t4, 0(t0)
    sw   t4, 0(t1)
    addi t0, t0, 4
    addi t1, t1, 4
    sltu t5, t0, t3
    bne  t5, zero, loop
```

Assume that initially registers `a0` and `a1` store addresses in memory and register `a2` stores an integer N. Registers `t0` to `t5` are used to store temporary values and `zero` is a register that always has the value zero.

a) Briefly comment each line of the code.

b) Describe in one sentence what this program does (its purpose).

c) Why did the instruction **`addi`** add 4 to registers `t0` and `t1`?

d) Why is the instruction **`sltu`** (set less than unsigned) used instead of the instruction **`slt`**?

[Solution 2] Understanding RISC-V

a) Commented code:

```
    add t0, a0, zero     # t0 <- a0
    add t1, a1, zer      # t1 <- a1
    add t2, a2, a2       # t2 <- 2*a2
    add t2, t2, t2       # t2 <- 4*a2
    add t3, t0, t2       # t3 <- a0 + *a2
loop:
    lw   t4, 0(t0)       #
    sw   t4, 0(t1)       #
    addi t0, t0, 4       # t0 <- t0+4
                           # (points on the next word)
    addi t1, t1, 4       # t1 <- t1+4
                           # (points on the next word)
    sltu t5, t0, t3      # if (t0 < t3)
                           # t5 <- 1 else t5 <- 0

    bne  t5, zero, loop # if (t5 != 0) go to loop
```

b) This RISC-V program copies the contents of the memory area that ranges from address `a0` to address `a0+4N`, to the memory area ranging from `a1` to `a1+4N`.

c) `t0` and `t1` are pointers on the source and destination memory areas, respectively. The memory in a RISC-V system is Byte addressable and hence words of 32 bits (4 bytes) are aligned on an address that is a multiple of 4. The **`lw`** and **`sw`** instructions access 32-bit words. Thus to reach the following word, the current address must be incremented by 4. This is done in instructions (**`addi t0, t0, 4`**) and (**`addi t1, t1, 4`**), for each vector.

d) The code line **`sltu t5, t0, t3`** compares registers `t0` and `t3` which are pointers. The address space of the system resides between `0x0000'0000` and `0xFFFF'FFFF`. A pointer is thus always unsigned and comparisons are made between two unsigned values.

[Exercise 3] Understanding RISC-V

Consider the following RISC-V program:

```
begin:
    add  t0, a0, zero
    add  t1, a1, zero
    add  t2, a2, zero
    add  t3, zero, zero
    add  t4, zero, zero

outer:
    lbu  t5, 0(t0)
    bne  t3, a3, cont

inner:
    lbu  t6, 0(t1)
    sb   t6, 0(t2)
    addi t1, t1, 1
    addi t2, t2, 1
    addi t4, t4, 1
    bne  t6, zero, inner
    addi t2, t2, -1
    addi t4, t4, -1

cont:
    sb   t5, 0(t2)
    addi t0, t0, 1
    addi t2, t2, 1
    addi t3, t3, 1
    bne  t5, zero, outer
    addi t3, t3, -1
    add  a0, t3, t4

fin:
    ret
```

a) Describe in a sentence what this program does, knowing that it takes four arguments stored in registers `a0`, `a1`, `a2` and `a3`. Arguments `a0` and `a1` contain each the memory address of a string that ends with the NULL character, i.e. a zero byte (`0000'0000`).

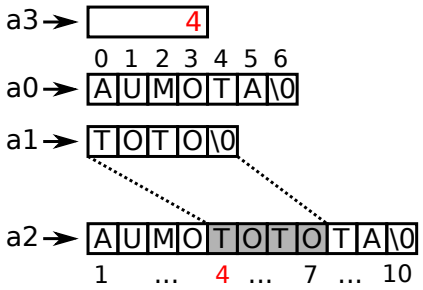
b) Modify the program such that the **`lbu`** instruction is replaced by **`lw`** without altering the functionality (behaviour). String addresses (i.e. beginning of the string) are always

multiples of four and the processor is little-endian. The `sb` instruction is available.

c) Should the program in the preceding point be modified if the processor were big-endian? If so, briefly describe the necessary modifications.

[Solution 3] Understanding RISC-V

a) The program returns (in `a0`) a pointer on a string that contains the string pointed by `a0` with the string pointed by `a1` inserted at the position indexed by `a3` as shown in the figure below:



b) The following subroutine does not use the **`lbu`** instruction. Instead, it uses the **`lw`** instruction keeping in mind that the processor is little-endian.

```
begin:
    add  t0, zero, a0
    add  t1, zero, a1
    add  t2, zero, a2
    add  t3, zero, zero
    add  t4, zero, zero
outer:
    andi s0, t3, 0x3
    bne  s0, zero, no_ld1
    lw   t5, 0(t0)
    addi t0, t0, 4
no_ld1:
    andi s0, t5, 0xff    # ***
    srli t5, t5, 8       # ***
    bne  t3, a3, cont
inner:
    andi s1, t4, 0x3
    bne  s1, zero, no_ld2
    lw   t6, 0(t1)
    addi t1, t1, 4
no_ld2:
    andi s1, t6, 0xff    # +++
    srli t6, t6, 8       # +++
    sb   s1, 0(t2)
    addi t2, t2, 1
```

```
    addi t4, t4, 1
    bne  s1, zero, inner
    addi t2, t2, -1
    addi t4, t4, -1
cont:
    sb   s0, 0(t2)
    addi t2, t2, 1
    addi t3, t3, 1
    bne  s0, zero, outer
    addi t3, t3, -1
    add  a0, t3, t4

fin:
    ret
```

c) Yes. Instead of the sequence of instructions **andi** and **srli** (marked by *** and +++) the following instruction sequence might be used instead (alternate solutions are possible):

```
    srl  s0, t5, 24     # prepare byte
    andi s0, s0, 0xff   # get byte
    sll  t5, t5, 8      # prepare next
                ...
    srl  s1, t6, 24     # prepare byte
    andi s1, s1, 0xff   # get byte
    sll  t6, t6, 8      # prepare for next
```

[Exercise 4] Vector Difference

Write a RISC-V function that produces a vector **C** from two vectors **A** and **B** containing 32-bit signed numbers in Two's complement representation, according to the following relationship:

$$\vec{C} = \left| \vec{A} - \vec{B} \right|$$

Thus vector **C** contains the absolute value of the difference between vectors **A** and **B**. These values are also 32-bit signed numbers.

Registers `a0`, `a1` and `a2` point on vectors **A**, **B** and **C**, respectively. Register `a3` indicates the number of elements in each vector.

a) Write the function described above, ignoring any possible overflows. Also, consider a RISC-V version where subtraction and arithmetic negation are not available. However all logical operations are available (**`and`**, **`or`**, **`xor`**, **`not`** and so on).

b) Discuss the possible overflow cases. Which instructions in your program could potentially generate an overflow? Briefly describe (without necessarily modifying the program) how to detect such overflows.

[Solution 4] Vector Difference

a) Since neither arithmetic negation nor subtraction are available, bitwise negation and addition must be used instead to implement the requested function.

```
begin:
    add  t0, zero, a0    # t0 <- a0
    add  t1, zero, a1    # t1 <- a1
    add  t2, zero, a2    # t2 <- a2
    add  t3, zero, a3    # t3 <- a3
loop:
    beq  t3, zero, fin   # if t3 = 0 then goto finish
    lw   t4, 0(t0)       # t4 <- mem[t0]
    lw   t5, 0(t1)       # t5 <- mem[t1]
    not  t5, t5          # t5 <- not t5
    addi t5, t5, 1       # t5 <- t5 + 1 (***)
    add  t5, t4, t5      # t5 <- t4 + t5 (+++)
    slt  t4, t5, zero    # check the sign
    beq  t4, zero, skip #if positive goto skip
    not  t5, t5          # t5 <- t5 (not not)
    addi t5, t5, 1       # t5 <- t5 + 1 (***)

skip:
    sw   t5, 0(t2)       # mem[t2] <- t5
    addi t0, t0, 4       # t0 <- t0 + 4
    addi t1, t1, 4       # t1 <- t1 + 4
    addi t2, t2, 4       # t2 <- t2 + 4
    addi t3, t3, -1      # t3 <- t3 - 1
    j    loop
fin:
    ret
```

b) The instructions marked by `***` and `+++` can cause an overflow.

In the `***` case, an overflow will occur if register `t5` contains the smallest negative number (`'1'` at the MSB followed by `'0'`s) before the **`not`** instruction is performed. Thus we must detect this case in order to detect the overflow.

In the `+++` case, there is a standard overflow treatment.

[Exercise 5] Understanding RISC-V

Study the following RISC-V program:

```
begin :
    mv t0, a0
    mv t1, a1
    mv t2, zero
    addi a0, zero, -1

cont :
    lbu  t4, 0(t1)
outer :
    lbu  t3, 0(t0)
    beq  t3, zero, fin
    bne  t3, t4, skip
    mv   a0, t2
    mv   t5, t0
inner :
    addi t0, t0, 1
    addi t1, t1, 1
    lbu  t3, 0(t0)
    lbu  t4, 0(t1)
    beq  t4, zero, fin
    beq  t3, zero, fail
    beq  t3, t4, inner
    addi t0, t5, 1
    mv   t1, a1
    addi a0, zero, -1
    j    cont

skip :
    addi t0, t0, 1
    addi t2, t2, 1
    j    outer

fail :
    addi a0, zero, -1
fin :
    ret
```

a) Describe in a sentence what this program does, knowing that it takes two arguments in registers `a0` and `a1`. Each of the two registers contains the memory address of a

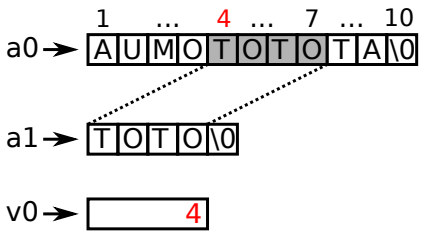
string ending with a NULL character, i.e. a zero byte.

b) Explain in a few words what the content of `t5` represents and the situation in which this value is needed.

c) Suppose that the **`lbu`** instruction is not available, but instead **`lw`** is used to read the memory. Write a function that provides the same functionality as the instruction **`lbu`** `v0, 0(a0)`, knowing that `a0` contains the memory address of the byte to be loaded, and `a0` is the register where this byte must be stored at the end of the function call. Consider a big-endian processor. Recall that the **`lbu`** instruction does not perform a sign extension. Make sure that the addresses that are passed to **`lw`** are always aligned, i.e. multiples of 4.

[Solution 5] Understanding RISC-V

a) The subroutine returns the position (index) of the first appearance of a substring in the input string. `a1` points on the substring and `a0` on the input string. If the substring is not found, the negative position is returned (-1).




```
begin:
    mv   t0, a0          # t0 <- a0
    mv   t1, a1          # t1 <- a1
    mv   t2, zero        # t2 <- 0
    addi a0, zero, -1    # a0 <- -1

cont:
    lbu  t4, 0(t1)       # t4 <- mem[t1]

outer:
    lbu  t3, 0(t0)       # t3 <- mem[t0]
    beq  t3, zero, fin   # if t3 = 0 goto fin
    bne  t3, t4, skip    # if t3 <> t4 goto skip
    mv   a0, t2          # remember index
    mv   t5, t0          # wrong guess backup

inner:
    addi t0, t0, 1       # t0 <- t0 + 1
    addi t1, t1, 1       # t1 <- t1 + 1
    lbu  t3, 0(t0)       # t3 <- mem[t0]
    lbu  t4, 0(t1)       # t4 <- mem[t1]
    beq  t4, zero, fin   # return        found
    beq  t3, zero, fail  # if t3 = 0 fail
    beq  t3, t4, inner   # continue inner loop
    addi t0, t5, 1       # recover wrong guess
    mv   t1, a1          # recover
    j    cont            # goto continue

skip:
```

```
    addi t0, t0, 1       # t0 <- t0 + 1
    addi t2, t2, 1       # t2 <- t2 + 1
    j    outer           # goto outer

fail:
    addi a0, zero, -1    # a0 <- -1

fin:
    ret
```

b) Register `t5` enables recovery from a "wrong guess". It holds the address where the matching started. If it does not succeed (the substring is not found), the matching is restarted from the next potential match at the address `t5` + 1.

c) The following subroutine provides the functionality of the **`lbu`** instruction. Two possible solutions are presented. Both of them assume a big-endian processor.

```
xlbu:
    li   t1, 0xfffffffc # t1 <- 0xfffffffc
    and  t0, a0, t1     # t1 <- a0 & t1 (align)
    li   t1, 0xff000000 # t1 <- 0xff000000
    andi t2, a0, 0x3    # t2 <- a0 & 0x3 (get offset)
    lw   t3, 0(t0)      # t3 <- mem[t0]

loop:
    beq  t2, zero, done # if t2 = 0 done
    slli t3, t3, 8      # t3 <- t3 << 8 (next byte)
    addi t2, t2, -1     # t2 <- t2 - 1
    j    loop           # goto loop

done:
    and  t3, t3, t1     # t3 <- t3 & 0xff000000
    srli a0, t3, 24     # a0 <- t3 >> 24
    ret                 # return
```

Another possible solution that uses **`srli`**:

```
xlbu:
    li   t1, 0xfffffffc  # t1 <- 0xfffffffc
    and  t0, a0, t1      # t1 <- a0 & t1 (align)
    andi t2, a0, 0x3     # t2 <- a0 & 0x3 (get offset)
    lw   t3, 0(t0)       # t3 <- mem[t0]

loop:
```

```
    sltiu t1, t2, 3       # check t2 < 3
    beq   t1, zero, done  # if t2 = 3 done
    srli  t3, t3, 8       # t3 <- t3 >> 8 (next byte)
    addi  t2, t2, 1       # t2 <- t2 + 1
    j     loop            # goto loop

done:
    andi a0, t3, 0xff     # v0 <- t3 & 0xff (get byte)
    ret                   # return
```

[Exercise 6] Binary Coded Decimal

Write a RISC-V program that converts from the BCD (Binary Coded Decimal) representation to the ordinary binary representation. In BCD representation, numbers are encoded directly from their decimal representation and each decimal digit is represented in binary using 4 bits. For example, the value 1992 in decimal is encoded in BCD using 16 bits as `0001'1001'1001'0010`, while its ordinary binary representation is `0000'0111'1100'1000`. Certain binary values such as `1111'1010'1100'1110` cannot represent BCD values; this happens whenever a group of 4 bits represents a value greater than 9.

The unsigned 32-bit value to be converted is located in register `a0` and the binary result at the end of the conversion must be saved in register `a0`. If the value contained in register `a0` cannot represent a BCD value, the `a0` register must contain -1 at the end of the execution.

a) Write the conversion function ignoring any possible overflows. You can assume the availability of multiplication instructions on 32-bit operands:

```
mul  rd, rs, rt
muli rd, rs, imm
```

b) The **`mul`** and **`muli`** instructions do not exist in RISC-V. Modify the program such that it no longer makes use of these multiplication instructions.

c) Discuss the possible overflow cases. Modify the program, if necessary, to remedy the situation.

[Solution 6] Binary Coded Decimal

a) In order to simplify the computation, the conversion is done using the following decomposition (as an example, a four digit BCD number is used - $abcd$):

$$abcd_{10} = a * 10^3 + b * 10^2 + c * 10^1 + d * 10^0 = (((a * 10 + b) * 10 + c) * 10 + d)$$

An eight digit BCD number is converted using the following procedure:

```
begin:
    add  t0, a0, zero      # init t0
    add  t6, zero, zero    # init t6 to 0 (final result)
    beq  t0, zero, fin     # finish if zero (optional)
    addi t1, zero, 8       # number of digits
loop:
    srli  t2, t0, 28       # get the MS digit d
    sltiu t3, t2, 10       # check d < 10
    bne   t3, zero, skip   # skip if ok
    addi  t6, zero, -1     # if error,
    j     fin              # finish
skip:
    muli  t6, t6, 10       # t6 <- t6*10
    add   t6, t6, t2       # t6 <- t6 + d
    slli  t0, t0, 4        # prepare next digit
    addi  t1, t1, -1       # decrement counter
    bne   t1, zero, loop   # continue until 0
fin:
    addi a0, t6, 0         #final value
    ret                    # return
```

b) Multiplication by 10 can be replaced by shifting and adding (the idea is X * 10 = X*8 + X*2) as shown in the following RISC-V implementation:

```
    ...
skip:
    slli  t3, t6, 3        # mul t6 by 8
    slli  t4, t6, 1        # mul t6 by 2
    add   t6, t3, t4       # t6 <- 10 * t6
    ...
```

c) It is not possible to have an overflow since the largest number in eight digit BCD representation (32 bits) is smaller than the largest 32-bit unsigned number (or even the largest number in two's complement representation).

[Exercise 7] ASCII Characters Transfer

We use a RISC-V processor to control the transmission of ASCII characters (8 bits per character: 7 bits of information and 1 parity bit) on a serial line. The latter is particularly exposed to "burst" errors (burst errors on the serial line affect several consecutive bits). In order to improve error detection, we use a technique called "interleaving" before transmission:

1. We group 4 characters to be transmitted in a 32-bit block (4×8 bits). We can thus assign an absolute position (i.e. 0..31) to each bit in the 32-bit block. For example, bit 5 of byte 2 will have position 21 ($5 + 2 \times 8$), while bit 0 of byte 3 will have the position 24 ($0 + 3 \times 8$).
2. We change the order of the bits and form a new 32-bit block for transmission in the following way:

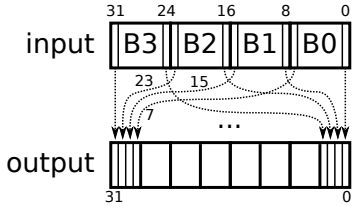


Figure 1: Bit interleaving

We can define from Figure 1 the `interleaving` function that determines the position of a bit after interleaving (where $\mathrm{i} = 0..31$ represents the original position):

$$\mathrm{newpos(i)} = \mathrm{i} \,\%\, 8 + (\mathrm{i}\ mod\ 8) \cdot 4$$

The `%` operation represents the integer division and `mod` is the modulo (rest of an integer division). For example, after interleaving is done, bit 23 (original position) will be relocated to position 30 ($23 \,\%\, 8 + (23\ mod\ 8) \times 4 = 2 + 28$), while bit 24 will be relocated to position 3 ($24 \,\%\, 8 + (24\ mod\ 8) \times 4 = 3 + 0$).

a) Implement the `newpos(i)` function in a RISC-V program. Parameter `i` is passed to the function using register `a0` and the function returns the new position in register `v0`.

b) Using the `newpos` function, write a RISC-V program that carries out the interleaving of a 4 byte block (32 bits). The source block is located in register `a0`. The result of the program must be placed in register `a0`.

c) Discuss the RISC-V convention regarding register saving between function calls. Discuss their application to the functions written in the preceding points.

d) Suppose that we send information between a little endian source computer and a big endian destination computer. Should the program that performs the interleaving be modified ? If so, briefly discuss the necessary modifications. If not, explain why.

[Solution 7] ASCII Characters Transfer

a)

```
newpos:
    add  t5, a0, zero  # t5 <- a0
    srli t6, t5, 3     # t6 <- t5 % 8
    andi t5, t5, 0x7   # t5 <- t5 mod 8
    slli t5, t5, 2     # t5 <- t5 * 4
    add  a0, t5, t6    # v0 <- t5 + t6
    ret
```

b)

```
prog:
    add  t0, a0, zero   # t0 <- a0
    add  t1, zero, zero # t1 <- 0, result
    add  t2, zero, zero # t2 <- 0, index i
    addi t3, zero, 32   # t3 <- 32

loop:
    andi t4, t0, 0x1    # test bit0
    beq  t4, zero, skip # skip if bit0 = 0
    add  a0, t2, zero   # a0 <- i
    jal  ra, newpos     # call newpos(i)
    addi t4, zero, 0x1  # t4 <- 1
    sll  t4, t4, a0     # shift to newpos(i)
    or   t1, t1, t4     # set bit at newpos(i)

skip:
    srli t0, t0, 1       # shift to next bit
    addi t2, t2, 1       # t2 <- t2 + 1
    bne  t2, t3, loop    # if t2 != t3 goto loop

fin :
    add  a0, t1, zero   # a0 <- t1
```

c) One of the conventions stipulates that the called routine (function) must save the callee-saved registers (registers `s0` to `s7`) on the stack at the beginning of its execution. The function then restores the saved registers from the stack at the end of its execution. The convention also stipulates that the caller of the routine (`prog`) must save caller-saved registers (every other register) whose content is necessary before calling the routine, as it might modify them.

Applied to the functions in the preceding two points, we can see that despite a correct execution, the original contents of register `ra` in the main program is lost after calling `newpos`. This is not a problem in the case of a stand-alone program, however it becomes one if the program is called from elsewhere! Given a stack that grows towards descending addresses and a pointer on the top of the stack, we could have the following save and restore code:

```
prog:
    addi sp, sp, -4      # sp <- sp - 4
    sw   ra, 0(sp)       # mem[sp] <- ra

    ...

    ...
fin:
    ...


    lw   ra, 0(sp)       # ra <- mem[sp]
    addi sp, sp, 4       # sp <- sp + 4
    ret                  # return
```

d) The correction operation of the program depends on the endianness. If a big endian destination computer receives the 4-byte packets form a little endian source computer (order before interleaving is $B_3B_2B_1B_0$) it must correctly interpret them. That is, it must first deinterleave the packet, then restore the correct order $B_3B_2B_1B_0$ before storing the data in the memory. Having an elegant deinterleaving function at our disposal allows us to avoid restoring the correct order. Write this function.

[Exercise 8] Understanding RISC-V

Consider the following RISC-V program.

```
start:
    addi t1, a1, -1
outer:
    beq  t1, zero, fin
    add  t0, a0, zero
    add  t2, t1, zero
inner:
    beq  t2, zero, cont
    lw   t3, 0(t0)
    lw   t4, 4(t0)
    sltu t5, t4, t3
    beq  t5, zero, skip
    sw   t3, 4(t0)
    sw   t4, 0(t0)
skip:
    addi t0, t0, 4
    addi t2, t2, -1
    j    inner

cont:
    addi t1, t1, -1
    j    outer
fin:
```

a) Describe in a sentence what the program does, knowing that the two arguments in registers `a0` and `a1` contain the address of an array in memory and the number of its elements, respectively.

b) Explain in a few words the purpose of the two **`lw`** and the two **`sw`** instructions in this context.

c) What is the type of the processed values ? Justify your answer.

d) Explain in a few words what the contents of the three registers `t0`, `t1` and `t2` represent. Modify the program in order to correctly handle the case where `a1 = 0`.

e) The original program does not return any result. We want the function to return the following values in register `a0`:

- `0` in the case where no data in the memory has been changed.
- `-1` in the case where some data has been changed in the memory.

Modify the program such that it is possible to call it as a standard RISC-V function.

[Solution 8] Understanding RISC-V

a) This program sorts the elements of a vector in an ascending order.

b) The two **lw** and two **sw** instructions are used to swap the locations of two elements in memory.

c) The program processes unsigned integers as the comparison is done using the **sltu** (set less than unsigned) instruction.

d) Register `t0` is used as a pointer on the elements of the vector. Registers `t1` and `t2` are used as counters to control the two loops. `t1` contains the remaining iterations of the `inner` loop, in other words, the number of iterations of the `outer` loop. Register `t2` contains the total number of iterations of the `inner` loop.

When `a1 = 1` the program will not behave correctly (there is a risk of having the program generate wrong addresses). The following case should thus be avoided:

```
start:
    beq  a1, zero, fin
    addi t1, a1, -1
    ...
```

e) We present here possible changes.

```
start:
    addi t6, zero, 0     # t6 <- 0
    addi t1, a1, -1

outer:
    beq   t1, zero, fin

    ...                          # no changes

    sw    t3, 4(t0)
    sw    t4, 0(t0)
    addi  t6, zero, -1   # t6 <- -1

skip:
    addi  t0, t0, 4
    addi  t2, t2, -1
    j     inner

cont:
```

```
20      addi t1, t1, -1
21      j    outer
22
23  fin:
24      mv a0, t6           # a0 <- t6
25      ret
```

[Exercise 9] Understanding RISC-V

Analyze the following RISC-V function:

```
func:
    slli t0, a1, 2
    add  t0, a0, t0
    addi t0, t0, -4
loop:
    slt  t1, a0, t0
    beq  t1, zero, fin
    lw   t2, 0(a0)
    lw   t3, 0(t0)
    sw   t2, 0(t0)
    sw   t3, 0(a0)
    addi t0, t0, -4
    addi a0, a0, 4
    j    loop

fin:
    ret
```

When the function is called, `a0` contains the memory address of a vector of 32-bit numbers and `a1` contains an integer.

a) Describe in a sentence what the program does.

b) Must the numbers contained in the vector be either signed or unsigned? Or is it possible to have both signed and unsigned numbers in the vector ? Briefly explain your answer.

c) We would like to change this program so that it can process (handle) bytes. To this effect we need a function that receives four bytes in `a0` and returns the same four bytes in the reverse order in `a0`: byte B_3 (bits 32-24) is swapped with byte B_0 and B_2 with B_1. Write such a function respecting ordinary RISC-V conventions.

[Solution 9] Understanding RISC-V

a) This program inverts the order of elements in a vector in memory.

`a0` points (starting from the beginning) on the vector's element that must be swapped next.

`t0` points (starting from the end) on the vector's (other) element that must be swapped with the first.

```
func:
    slli  t0, a1, 2      # t0 = Nb elements * sizeof(elem)
    add   t0, a0, t0     # t0 points to the end of the vector
    addi  t0, t0, -4     # t0 points to the last element
loop:
    slt   t1, a0, t0
    beq   t1, zero, end # if( a0 >= t0) go to end
    lw    t2, 0(a0)      # Store in t2 mem(a0)
    lw    t3, 0(t0)      # Store in t3 mem(t0)
    sw    t2, 0(t0)      # Copy t2 to mem(t0)
    sw    t3, 0(a0)      # Copy t3 to mem(a0)
    addi  t0, t0, -4     # Update the end pointer
    addi  a0, a0, 4      # Update the start pointer
    j     loop           # Continue the loop

end:
    ret                  # Return of the function
```

b) The RISC-V function does not modify any of the vector's elements, there is thus no restriction on their type, they can be signed, unsigned or in any other representation.

c) The function that inverts the order of the bytes of a 32-bit input is given below.

```
Inv_byte:
    li   t2, 0           # t2 will store the result
    addi t0, zero, 0xFF # Mask byte 7-0 in t0
    and  t1, a0, t0      # Byte 7-0 in t1
    slli t1, t1, 24      # Shift byte 7-0 to position 31-24
    add  t2, zero, t1    # Byte 31-24 of the result is ready

    slli t0, t0, 8       # Mask byte 15-8 in t0
    and  t1, a0, t0      # Byte 15-8 in t1
```

```
10      slli t1, t1, 8       # Shift byte 15-8 to position 23-16
11      or   t2, t2, t1      # Byte 23-16 of the result is ready
12
13      slli t0, t0, 8       # Mask byte 23-16 in t0
14      and  t1, a0, t0      # Byte 23-16 in t1
15      srli t1, t1, 8       # Shift byte 23-16 to position 15-8
16      or   t2, t2, t1      # Byte 15-8 of the result is ready
17
18      slli t0, t0, 8       # Mask byte 31-24 in t0
19      and  t1, a0, t0      # Byte 31-24 in t1
20      srli t1, t1, 24      # Shift byte 31-24 to position 7-0
21      or   t2, t2, t1      # Byte 7-0 of the result is ready
22
23  fin:
24      mv a0, t2
25      ret                  # return of the function
```

[Exercise 10] Understanding RISC-V

Analyze the following RISC-V function:

```
1  func:
2      add  t0, zero, zero
3      add  t1, zero, a0
4      lw   t2, 0(a0)
5      lw   t3, 0(a0)
6  label:
7      lw   t5, 0(t1)
8      slt  t4, t2, t5
9      bne  t4, zero, cont1
10     add  t2, zero, t5
11 cont1:
12     slt  t4, t5, t3
13     bne  t4, zero, cont2
14     add  t3, zero, t5
15 cont2:
16     addi t0, t0, 1
17     addi t1, t1, 4
18     bne  a1, t0, label
19     add  t4, t2, t3
20     sra  a0, t4, 1
21 fin:
22     ret
```

When the function is called, `a0` contains the memory address of a vector containing 32-bit numbers, `a1` contains an integer and `a0` contains the value returned by the function.

a) Describe in a sentence what the program does. What value is returned by the program? Is this value always accurate (exact)?

b) Must the vector's elements be either signed or unsigned? Or is it possible to have both signed and unsigned values in the vector? Briefly explain your answer, but be precise. If only a single type of numbers can be processed (handled) by the function in its current form, indicate the necessary modifications so that it can process the other type.

c) Modify the function such that it can process a vector of 16-bit numbers instead of 32-bit. The 16-bit half-words are located in memory as shown in Figure 2. Minimize the number of memory accesses (use **`lw`** to access the memory). Suppose that the processor is **big-endian** and that `a0` always contains a multiple of 4 when the function is called.

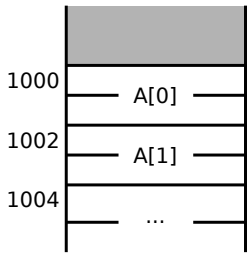


Figure 2: Location of vector's 16-bit elements in memory

[Solution 10] Understanding RISC-V

a) The purpose of the program is to find the arithmetic mean of the minimum and maximum value of the vector's elements (`a0` points on the vector). The mean is the value returned by the program. Since an integer division by two is performed in the last instruction before returning, the resulting mean is not always accurate (exact). A commented version of the code is given below:

```
func:
    add  t0, zero, zero   # Initialize element counter (t0)
    add  t1, zero, a0     # t1 points to the first element
    lw   t2, 0(a0)        # Initialize the min (t2)
    lw   t3, 0(a0)        # Initialize the max (t3)
label:
    lw   t5, 0(t1)        # The next elemenent in t5
    slt  t4, t2, t5       # (t2 >= t5 ) => t4 = 0
    bne  t4, zero, cont1  #
    add  t2, zero, t5     # Update the min in t2
cont1:
    slt  t4, t5, t3       # (t5 >= t3 ) => t4 = 0
    bne  t4, zero, cont2  #
    add  t3, zero, t5     # Update the max in t3
cont2:
    addi t0, t0, 1        #
    addi t1, t1, 4        # Go to next element
    bne  a1, t0, label    # Check if we reached the end
    add  t4, t2, t3       # Add the max and the min
    srai a0, t4, 1        # Divide the sum by 2
fin:
    ret                   # Return of the function
```

b) The elements of the vector pointed on by `a0` must be signed because the comparisons used to find the max and min are done on signed numbers using the **`slt`** instruction. The mean calculations are also done using signed numbers. Thus the program must be modified as follows to handle unsigned numbers:

```
    ...
    slt  t4, t2, t5       =>   sltu  t4, t2, t5
    ...
    slt  t4, t5, t3       =>   sltu  t4, t5, t3
    ...
    add  t4, t2, t3       =>   add   t4, t2, t3
    srai a0, t4, 1        =>   srli  a0, t4, 1
```

c) In order to minimize memory accesses, each time the memory is accessed using **lw**, the 32-bit read value is stored in a register to avoid having to access the memory again to read the second element of the vector, given that the values are encoded using 16 bits, register t6 is used for this purpose.

```
func:
    lw   t2, 0(a0)        # First 2 elements in t2
    add  t6, zero, t2     # First 2 elements in t6
    srai t2, t2, 16       # Initialize the min
    add  t3, zero, t2     # Initialize the max
    addi t0, zero, 0x1    # Initialize counter of elements t0
    add  t1, zero, a0     # t1 points on the 1st element
loop:
    andi t4, t0, 1        # If even element counter fetch new word
    bne  t4, zero, LSW    #
    lw   t5, 0(t1)        # Read the 2 following elements
    add  t6, zero, t5     # Copy t5 in t6
    j    MSW              # Next element on MSW
LSW:
    slli t5, t6, 16       # Shift on most significant word
MSW:
    srai t5, t5, 16       # Shift on LSW with sign
    slt  t4, t2, t5       # (t2 >= t5 ) => t4 = 0
    bne  t4, zero, cont1
    add  t2, zero, t5     # Update the max in t2
cont1:
    slt  t4, t5, t3       # (t5 >= t3 ) => t4 = 0
    bne  t4, zero, cont2  #
    add  t3, zero, t5     # Update the max in t3
cont2:
    addi t0, t0, 0x1      # Go the next element
    andi t4, t0, 1        # If even element counter,
    bne  t4, zero, cont3  #
    addi t1, t1, 0x4      # Update the vector pointer
cont3:
    bne  a1, t0, loop     # Check if we reached the end
    add  t4, t2, t3       # Add the max and the min
    srai a0, t4, 0x1      # Divide the sum by 2
fin:
    ret                   # Return
```

[Exercise 11] CORDIC

The CORDIC algorithm (COordinate Rotation DIgital Computer) allows calculating the length of a vector V(X,Y) using the following iterative relationships:

$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i} \tag{1}$$
$$y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i} \tag{2}$$
$$z_{i+1} = z_i - d_i \cdot tan^{-1}(2^{-i}) \tag{3}$$

Where $d_i = \begin{cases} -1 & \text{if } y_i > 0 \\ 1 & otherwise \end{cases}$

x, y, and z are initialized as follows: $x_0 = X$, $y_0 = Y$, and $z_0 = 0$.

After n iterations (n large enough), x_n, y_n, and z_n can be expressed as follows:

$$x_n \approx A_n \cdot \sqrt{X^2 + Y^2} \tag{4}$$
$$y_n \approx 0 \tag{5}$$
$$z_n \approx tan^{-1}\left(\frac{Y}{X}\right) \tag{6}$$

If we neglect constant A_n, these iterative formulas allow calculating the length of a vector as well as its angle with the x axis.

a) Suppose that initially $x_0 = -24$, $y_0 = 32$, and $z_0 = 0$. Calculate by hand the value of x_2 and y_2. Do not calculate z_2.

b) Using the aforementioned iterative formulas, write a RISC-V function that computes in 31 iterations the length and angle (with the x axis) of a vector whose x and y coordinates are available in registers `a0` and `a1`. These values are 32-bit signed integer values encoded in two's complement. Register `a2` points on an array of 32-bit words in the main memory which contains the already computed $tan^{-1}(2^{-i})$ coefficients (you do not need to compute them). Element 0 in the array is the coefficient of iteration 0, element 1 in the array is the coefficient of iteration 1 and so on. Ignore possible overflows. The length and angle should be returned in `a0` and `a1` respectively.

[Solution 11] CORDIC

a) We can find x_2 and y_2 using the iterative formulas. Initially, we have:

$$x_0 = -24$$
$$y_0 = 32$$
$$z_0 = 0$$

Iteration 1 will be:

$$y_0 > 0 \Rightarrow d_0 = -1$$
$$x_1 = -24 - (-1) \cdot 32 \cdot 20 = -24 + 32 = 8$$
$$y_1 = 32 + (-1) \cdot (-24) \cdot 20 = 32 + 24 = 56$$

Then, iteration 2:

$$y_1 > 0 \Rightarrow d_1 = -1$$
$$x_2 = 8 - (-1) \cdot 56 \cdot 2^{-1} = 8 + 28 = 36$$
$$y_2 = 56 + (-1) \cdot 8 \cdot 2^{-1} = 56 - 4 = 52$$

b) The RISC-V function that calculates the length and angle of a vector using the CORDIC formulas is given below:

```
1  cordic:
2      add   t0, zero, a0      # t0 <- x0
3      add   t1, zero, a1      # t1 <- y0
4      add   t2, zero, zero    # t2 <- z0 (init to 0)
5      add   t3, zero, a2      # t3 <- a2 (coef table)
6      addi  t4, zero, 1       # t4 <- 1 (index)
7  loop:
8      sltiu t5, t4, 32        # if t4 = 32
9      beq   t5, zero, end     #  then goto end
10     slt   t5, zero, t1      # if  0 < yi , t5=1 else t5=0
11     beq   t5, zero, dpos    #  then go to dpos, else
12 dneg:
13     sra   t5, t1, t4        # t5 <- yi*2**(-i)
14     sra   t6, t0, t4        # t6 <- xi*2**(-i)
15     add   t0, t0, t5        # compute next xi
16     sub   t1, t1, t6        # compute next yi
17     lw    t5, 0(t3)         # t5 <- coeff[i]
18     add   t2, t2, t5        # compute next zi
```

```
19      j       cont                # goto cont
20  dpos:
21      sra     t5, t1, t4          # t5 <- yi*2**(-i)
22      sra     t6, t0, t4          # t6 <- xi*2**(-i)
23      sub     t0, t0, t5          # compute next xi
24      add     t1, t1, t6          # compute next yi
25      lw      t5, 0(t3)           # t5 <- coeff[i]
26      sub     t2, t2, t5          # compute next zi
27  cont:
28      addi    t3, t3, 4            # t3 <- next coeff addr
29      addi    t4, t4, 1            # t4 <- next index
30      j       loop                 # goto loop
31  end:
32      add     a0, zero, t0        # a0 <- xn
33      add     a1, zero, t2        # a1 <- zn
34      ret                         # return
```

[Exercise 12] Division Rest

We want to write a function that calculates the rest of an integer division of a 32-bit unsigned integer by 15 without performing the division per se. We can use the following property: the division of a number by 15 is the recursive sum of the digits of its hexadecimal representation. We thus start by summing all the digits (in hexadecimal representation) of the given number. Suppose the hexadecimal representation of this sum has more than one digit. In that case, we apply the same procedure recursively, i.e., we sum the digits of the sum until we obtain a value that is represented by a single hexadecimal digit. This value is the rest of the integer division by 15, except if it is equal to 15, in which case the rest is 0 (and not 15).

For example, let N = `0x3204'1EF2` = 839'130'866. We calculate the rest of its division by 15 as follows:

- We first sum the 8 digits of the hexadecimal representation of N:
  `§3 + 2 + 0 + 4 + 1 + E + F + 2 = 0x29 (41)§`
- Since the obtained sum `0x29` has two digits, we compute their sum:
  `§2 + 9 = 0xB (11)§` which has only one digit.

This value `0xB` is the result of the integer division of N by 15, as it has only one digit and is different from 15 (no need to replace it with 0). It can easily be verified that $839'130'866 = 55'942'057 \cdot 15 + 11$.

a) Write a function that calculates and returns the rest of the division of a 32-bit number N by 15. N is the only parameter supplied to the function through the `a0` register. The value of this register does not necessarily need to be preserved. Make use of the calculation's recursive structure by writing a recursive function. Your code must respect RISC-V conventions.

[Solution 12] Division Rest

a) We make the following choices:

- To isolate the 4 bits necessary to compute the partial sums we successively use the **sll** and **srl** instructions. Alternately we could also use a mask.
- To recursively compute the partial sums we can use the "jump" instruction (**j**) instead of "jump and link" (**jal**), as only the ra register must me preserved.

A possible solution is given below:

```
start:
    addi t0, zero, 28
    addi t1, zero, 28
    add  t2, zero, zero  # intermediary sum
    addi t3, zero, 8     # 32-bit hex digits
sum :
    beq  t3, zero, rec   # end partial sums test
    sll  t4, a0, t0
    srl  t4, t4, t1
    add  t2, t2, t4      # partial sum
    addi t0, t0, -4      # shift left value
    addi t3, t3, -1      # loop counter update
    j    sum
rec :
    add  t5, t2, zero
    li   t6, 0xFFF0
    and  t5, t5, t6
    beq  t5, zero, fin   # recursion test
    add  a0, t2, zero    # update a0
    j    start           # recursion
fin :
    addi t0, zero, 15    # Edge case
    bne  t2, t0, skip
    addi t2, zero, 0
skip :
    add  a0, zero, t2
    ret
```

Note that we can also use the **jal** instruction for recursion, but in this case the ra register must be saved on the stack as each new (recursive) call of the function overwrites the previous value (of ra).

[Exercise 13] Run-Length Encoding

Write a RISC-V function that performs Run-Length Encoding. This encoding is efficient for representing a string containing characters that are often repeated consecutively. Each character, repeated or not, is represented using two bytes, the first being the character itself, and the second the number of times this character is repeated consecutively. For example, the string `'aaaabccc'` will be encoded into the RLE sequence `'a',4,'b',1,'c',3`.

Some specifications of the RISC-V function that performs the RLE encoding are given below:

- Characters are in ASCII format (an unsigned byte whose value ranges from 0 to 127).
- The function receives in register `a0` a pointer on a string that ends with a null character, i.e., the last byte of the string is zero.
- The result of the function is a list of bytes representing the encoded string. The memory address where the function must write the encoded string is given in register `a1`. The encoded list of bytes also ends with a null byte.

Figure 3 below shows the input string and the output encoded list of bytes:

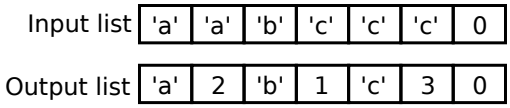


Figure 3: Example of RLE encoding

a) Write a RISC-V function that performs RLE encoding. Assume that a character is never repeated consecutively more than 255 times. Your code should conform to RISC-V conventions.

b) Give a simple way of modifying the encoding in the event of a character being repeated more than 255 times. Modify the code of the function accordingly.

c) Modify the function to implement a more efficient encoding as follows:

- If a character has a single occurrence and is not repeated, the `'1'` is omitted.

- If a character is repeated (consecutive occurrences), the encoding is essentially the same as before. However, in order to determine whether an element (byte) represents a character or a number of repetitions, the most significant bit (MSB) is set to `'1'` in the latter case (i.e., 128 is added to the value representing the number of repetitions).

The example string `'aaaabccc'` is now encoded into the sequence `'a',132` (4 + 128), `'b','c',131`. Note that since the MSB is now used to distinguish between characters and repetitions, the maximum number of repetitions that can be handled is now 127 instead of 255 (we lose the MSB).

[Solution 13] Run-Length Encoding

a) The code of the RISC-V function that performs RLE encoding is given below with appropriate comments. `t0` contains the last read character, `t1` contains the repeated character, and `t2` the number of repetitions.

```
rle:  lbu  t0, 0(a0)      # Initialisations
      beq  t0, zero, fin
      add  t1, zero, t0
      add  t2, zero, zero

loop: bne  t0, t1, diff   # Check if char is different
      addi t2, t2, 1      # If similar, increment counter
      j    meme

# Write the repeated char and the number of repetitions
# on the output list. Update the pointer of the output list
diff: sb   t1, 0(a1)
      sb   t2, 1(a1)
      addi a1, a1, 2

# If the input list is completed, terminate the function
      beq  t0, zero, fin

# Update the repeated char and the number of repetitions
      add  t1, t0, zero
      addi t2, zero, 1

meme: addi a0, a0, 1      # Go to next char
      lbu  t0, 0(a0)      # Read the next char
      j    loop

fin:  sb   zero, 0(a1)    # Ends the output list with 0
      ret                 # return
```

b) The simplest way of modifying the code is to interpret a character that is repeated more than 255 times as a new character. For example, if a character repeats 256 times, it will be encoded as `'a', 255, 'a', 1`. This only requires adding a simple control sequence after the `loop` label to implement the desired new functionality. The modification is thus:

```
loop: bne  t0, t1, diff
      addi t3, zero, 255   # Verify the 255 limit
      beq  t2, t3, diff    # If the limit is reached
```

```
                            # consider the next repeated
                            # char as different

        addi t2,t2,1
        j    meme
```

Note that in order to increase the performance of the code, we can move instruction `addi t3, zero, 255` within the initialization part of the function.

c) We only need to slightly modify the part that writes the encoded sequence: i.e., the first two lines after the `diff` label. If there are no repetitions, we simply increment the pointer on the output (encoded) sequence of bytes and continue, otherwise we add 128 to the number of repetitions and write it to the encoded string.

The modified code is given below.

```
diff:
    sb   t1, 0(a1)
    addi t4, zero, 1
    beq  t2, t4, ignore  # Repeated char ?
    addi t2, t2, 128     # yes, add 128
    sb   t2, 1(a1)       # write on the output encoded string
    addi a1, a1, 1       # increment pointer

ignore:
    addi a1, a1, 1       # increment pointer
    beq  t0, zero, fin
        ...
```

[Exercise 14] Understanding RISC-V

Consider the following RISC-V function:

```
1  func: slli t0, a0, 16
2        srli t0, t0, 16
3        slli t1, a1, 16
4        srli t1, t1, 16
5        add  a0, zero, zero
6  loop: beq  t1, zero, end
7        andi t2, t1, 1
8        beq  t2, zero, cont
9        add  a0, a0, t0
10 cont: slli t0, t0, 1
11       srli t1, t1, 1
12       j    loop
13 end:  ret
```

Two input parameters are passed to the function via registers `a0` and `a1`, and the result is returned through register `a0`.

a) Describe in a single sentence what the function does.

b) What is the maximum effective size (in bits) of the input and output parameters? Is there an overflow risk? Explain.

c) Are the input parameters signed or unsigned numbers? Explain.

d) If we replaced (only!) the first instruction **`srli`** by **`srai`**, would this function still perform a useful but eventually different action? If so, describe the new action and explain the result. If not, precisely explain why modifying the function in this manner makes no sense.

e) If we replaced the first and second **`srli`** instructions by **`srai`**, would this function still perform a useful but eventually different action? If so, describe the new action and explain the result. If not, precisely explain why modifying the function in this manner makes no sense.

f) If we replaced all three **`srl`** instructions by **`sra`**, would this function still perform a useful but eventually different action? If so, describe the new action and explain the result. If not, precisely explain why modifying the function in this manner makes no sense.

[Solution 14] Understanding RISC-V

a) The function performs the multiplication of two 16-bit unsigned numbers.

The product of the two numbers is calculated using the shift and add algorithm. The solution with comments is given below:

```
# Initialize the inputs and outputs.
# Set the 16 most significant bits of a0 and a1 to zero.
func:
    slli t0, a0, 16
    srli t0, t0, 16
    slli t1, a1, 16
    srli t1, t1, 16

# Initialize the product
    add  a0, zero, zero

#If the operand in t1 is zero, we end the loop
loop:
    beq t1, zero, end

#The least significant bit of t1 determines if
#the value must be added in t0. If this bit is
#zero, we continue without making the addition,
#otherwise we add to a0.
    andi t2, t1, 1       # We make the addition if the
    beq  t2, zero, cont  # least significant bit of t1 is 1.
    add  a0, a0, t0      # do the addition

# Prepare the two operands by shifting them by 1 bit
# The operand added to the result (t0) is shifted
# to the left. The operand that controls if the addition
# for the actual weight (t1) must be made, is shifted
# to the right.
cont:
    slli t0, t0, 1     # Shift t0 to the left
    srli t1, t1, 1     # Shift t1 to the right
    j    loop          # return to loop

end:
    ret                # return
```

b) As both 16-bit operands are initialized to zero at the beginning of the program, their maximum effective size is 16 bits. The result is 32-bit wide, as a result, there can be no overflow since multiplying two 16-bit numbers yields a number that can be represented using 32 bits.

$$(2^{16}-1)(2^{16}-1) = 2^{32} - 2^{17} + 1 < 2^{32} - 1$$

c) The 16 most significant bits of the operands are set to zero by the **srli** instructions and no sign extension is performed. These operands are thus interpreted as unsigned.

d) If we replace the first **srli** instruction by **srai** this means that we interpret this operand as being signed, while the other remains unsigned. The question is whether this has any effect on the proper execution of the function. There are two cases to consider:

- The first operand is positive. This case is not different than the original program. Thus using **srai** has no effect and the function returns a valid result.
- The first operand is negative. In this case a value `-0xC` is represented as 2^{32} - `0xC`, and thus multiplying this value with a positive number e.g., `0xB` yields:

  $$(2^{32} - C) \cdot B = 2^{32} \cdot B - C \cdot B = 0 - C \cdot B = (-C) \cdot B$$

  Which is correct if we interpret the result as being signed. Note that $2^{32} \cdot B$ cannot be represented on 32 bits and becomes zero.

Thus the function still returns a valid result if we replace **srli** with **srai** as long as it is interpreted as being signed.

e) If we replace the first two **srli** instructions by **srai**, then the function still returns a valid result. There are four cases to consider:

- Both operands are positive. This case is the same as the original program. No problems.
- The first operand is negative. This case is similar to the one discussed in the previous point.
- The second operand is negative. This case is similar to the preceding one. If the first operand has a value of `0xA` and the second operand a value of `-0xC` represented as 2^{32} - `0xC`), then:

$$A \cdot (2^{32} - C) = 2^{32} \cdot A - A \cdot C = A \cdot (-C)$$

Which yields a correct result if it is interpreted as being signed.

- Both operands are negative. Suppose the first operand has a value of `-0xC` (2^{32} - `0xC`) and the second a value of `-0xD` (2^{32} - `0xD`), then:

$$\begin{aligned}(2^{32} - C)(2^{32} - D) &= (2^{64} - 2^{32} \cdot C - 2^{32} \cdot D + C \cdot D) \\ &= 2^{32}(2^{32} - C - D) + C \cdot D \\ &= 0 + C \cdot D \\ &= C \cdot D\end{aligned}$$

This is the expected and correct value.

Hence, replacing the first two instructions does not alter the behaviour of the function as long as the result is interpreted as being signed.

f) The function does not yield correct results if all three **`srli`** are replaced by **`srai`**. Indeed, if register `t1` contains a negative value then shifting it will extend the sign, the value in `t1` will never reach zero and the program will not terminate.

[Exercise 15] Floating Point Larger-Than

Floating point formats in computer science are similar to the common scientific notation, with a mantissa and an exponent. For example $-9.062 \cdot 10^2$ or $3.87 \cdot 10^{-1}$. Usually we want to represent numbers with a mantissa bound between 10^0 and 10^1 in which case it is considered as normalized. If a given number is not normalized it is multiplied by an appropriate power of 10 to adapt the exponent so that the mantissa is within normalized bounds. For example :

$$\begin{aligned} -0.0041 \cdot 10^4 &= -4.1 \cdot 10^1 \\ 9760.1 \cdot 10^{-8} &= 9.7601 \cdot 10^{-5} \end{aligned}$$

Now consider the following 32-bit binary floating point format:

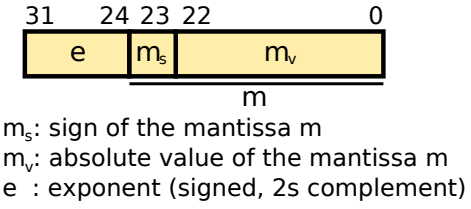


Figure 4: 32-bit floating point format

Bits 31 to 24 represent the exponent `e` in Two's Complement, while bits 23 to 0 represent the mantissa `m` in sign and magnitude, i.e. bit 23 represents the sign and bits 22 to 0 the absolute value. A value in floating point format can thus be represented as:

$$(-1)^{m_s} \cdot m_v \cdot 2^e$$

The point in the mantissa is implied after the most significant bit MSB, and thus bit 22 has a weight of 1, bit 21 a weight of $\frac{1}{2}$, bit 20 a weight of $\frac{1}{4}$ and so on. Moreover, numbers are normalized, i.e. $2^0 \leq m_v \leq 2^1$, meaning that the mantissa is aligned in a way such that bit 22 is always `'1'` and the exponent is adjusted accordingly. Of course in a real application this would be implied because it is useless, except to represent the value zero whose existence we will ignore in what follows.

As an example we represent some values in normalized floating point notation:

$$\begin{aligned} 14 &= 1.75 \cdot 2^3 &&= \langle 0000'0011'0111'0000'0000'0000'0000'0000 \rangle \\ -0.312 &= -1.25 \cdot 2^{-2} &&= \langle 1111'1110'1101'0000'0000'0000'0000'0000 \rangle \end{aligned}$$

a) Write a RISC-V function `larger-than` that receives two numbers in the aforementioned floating point format and yields an unsigned integer number. This function compares the two numbers and returns 1 if the first number is strictly greater than the second one, if not the function returns 0. Usual RISC-V conventions must be respected.

b) Write a RISC-V function `normalize` that converts a non-normalized number into a normalized one, according to the floating point format presented earlier. The zero number must be ignored as well as potential overflows of the floating point format. This function must also respect RISC-V conventions.

[Solution 15] Floating Point Larger-Than

a) To perform the required comparison, we can use the algorithm illustrated below:

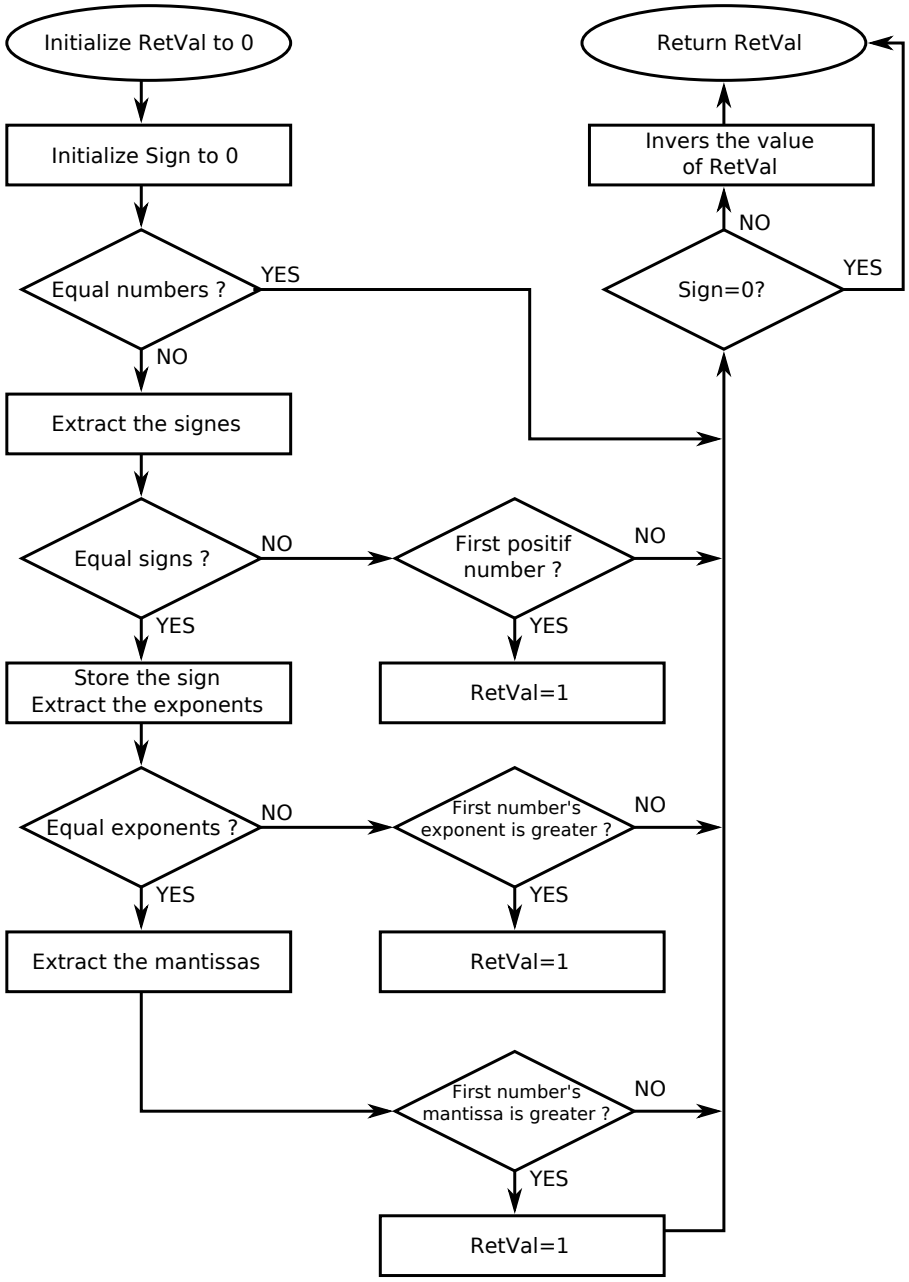


Figure 5: Algorithm for the comparison

The corresponding RISC-V program is given below with comments:

```
compare:
    add  t4, zero, zero          # Extract RetVal
    add  t0, zero, zero          # Initialize sign
    beq  a0, a1,   smaller_eq    # Equal numbers?
sign_check:
    slli t1, a0,   8             # Extract the signs
    srli t1, t1,   31
    slli t2, a1,   8
    srli t2, t2,   31
    beq  t1, t2,   set_sign      # Same signs ?
    beq  t1, zero, greater       # 1st pos number?
    j    smaller_eq
set_sign:
    add  t0, zero, t1            # Store the sign
exp_check:
    srai t1, a0,   24            # Extract the exponent
    srai t2, a1,   24
    beq  t1, t2,   mantissa_chk # same exponent?
    slt  t3, t1,   t2
    beq  t3, zero, greater
    j    smaller_eq
mantissa_chk:
    slli t1, a0,   9             # Extract the mantissa
    slli t2, a1,   9
    sltu t3, t2,   t1
    beq  t3, zero, smaller_eq
greater:
    addi t4, zero, 1             # RetVal=1
smaller_eq:
    xor  t4, t4, t0              # Adjust RetVal
    mv   a0, t4
    ret                          # Return
```

b) The idea is to shift the mantissa to the left until the MSB is `'1'` and adjust the exponent accordingly. The RISC-V code is given below:

```
normalize:
    slli t0, a0,   9          # Extract the mantissa
    srai t1, a0,   24         # Extract the exponent
test_nrm:
    slt  t2, zero,t0          # If MSB is 1
    beq  t2, zero,reformat    # end loop
    slli t0, t0,   1          # Left shift mantissa
```

```
8       addi t1, t1,  -1        # Decrement exponent
9       j    test_nrm           # Stay in the loop
10  reformat:
11      srli t2, a0,  23
12      slli t2, t2,  31        # Sign of MSB
13      srli t0, t0,  1         # Shift mantissa by 1 bit
14      add  t2, t2,  t0        # Mantissa and sign
15      srli t2, t2,  8         # Shift mantissa and sign
16      slli t1, t1,  24        # Shift exponent
17      add  a0, t2,  t1        # The format is correct
18      ret                     # return
```

[Exercise 16] Understanding RISC-V

Consider the following RISC-V function:

```
1  func:  add  t0,   zero, a0
2         add  t1,   zero, a1
3         add  t2,   zero, a2
4         add  a0,   zero, zero
5  loop:  beq  t2,   zero, fin
6         lw   t3,   0(t0)
7         lw   t4,   0(t1)
8         addi t5,   zero, 32
9         slt  t6,   t2,   t5
10        beq  t6,   zero, cont1
11        add  t5,   zero, t2
12 cont1: xor  t6,   t3,   t4
13 cont2: andi t3,   t6,   1
14        add  a0,   a0,   t3
15        srli t6,   t6,   1
16        addi t2,   t2,   -1
17        addi t5,   t5,   -1
18        bne  zero, t5,   cont2
19        addi t0,   t0,   4
20        addi t1,   t1,   4
21        j    loop
22 fin:   ret
```

Input parameters are passed to the function via registers `a0`, `a1`, and `a2`. Registers `a0` and `a1` contain each the start address of a list of bytes and `a2` contains an unsigned number.

a) Explain in a sentence what the above RISC-V code does.

b) Is the given code written for a little-endian or big-endian processor? Clearly state the reason.

c) The above code makes use of the **`xor`** instruction. Suppose this instruction does not exist. Write a sequence of instructions that replace the **`xor`** instruction.

[Solution 16] Understanding RISC-V

a) The supplied RISC-V code counts the number of bits that are different in the two input lists. The beginning of each list is given in registers `a0` and `a1`, while `a2` contains the number of bits in each list. A commented version of the RISC-V code is given below:

```
# Initializations
func:  add  t0, zero, a0   # Addresses of the lists
       add  t1, zero, a1   # in t0 et t1
       add  t2, zero, a2   # The number of bits in t2
       add  a0, zero, zero # Initialize the result

# The loop takes 32 bits of each list and counts the number of
# different bits for each pair of 32 bits. It finds the total
# number of different bits in the lists
loop:
    beq  t2, zero, fin    # All the bits have been checked?
    lw   t3, 0(t0)        # Put the 32 following bits of
    lw   t4, 0(t1)        # each list in t3 and 4
    addi t5, zero, 32     # Init. counter at 32 (bits)
    slt  t6, t2,   t5     # If less than 32 bits remain
    beq  t6, zero, cont1  # set counter value to the number
    add  t5, zero, t2     # of remaining bits

cont1:
    xor  t6, t3, t4       # The different bits of the 32 bit
                          # pair are stored in t6

# The loop cont2 counts the number of bits set to 1 in the
# register t6 (the number of different bits in the current
# comparison)
cont2:
    andi  t3,  t6, 1      # Extract the last bit
    add   a0,  a0, t3     # Add this bit to a0
    srli  t6,  t6, 1      # Right shift t6 by 1
    addi  t2,  t2, -1     # Decrement the counter of total bits
    addi  t5,  t5, -1     # Decrement the counter of  the loop
    bne   zero, t5, cont2 # Check the end of the loop
    addi  t0,  t0, 4      # Increment pointer
    addi  t1,  t1, 4      # to read the next 32 bits
    j     loop            # Jumps to the top of the loop
fin:
```

```
37      ret                     # Function returns
```

b) The RISC-V code is written for a little-endian processor. This can be deduced from the fact that counting the bits that are different starts from the least significant bit (instruction `andi t3, t6, 1`. That is, it is assumed that the least significant byte is stored at the smallest address in memory, which characterizes little-endian processors.

Figure 6 shows how the bytes are organized in the memory and the difference between little and big-endian processors.

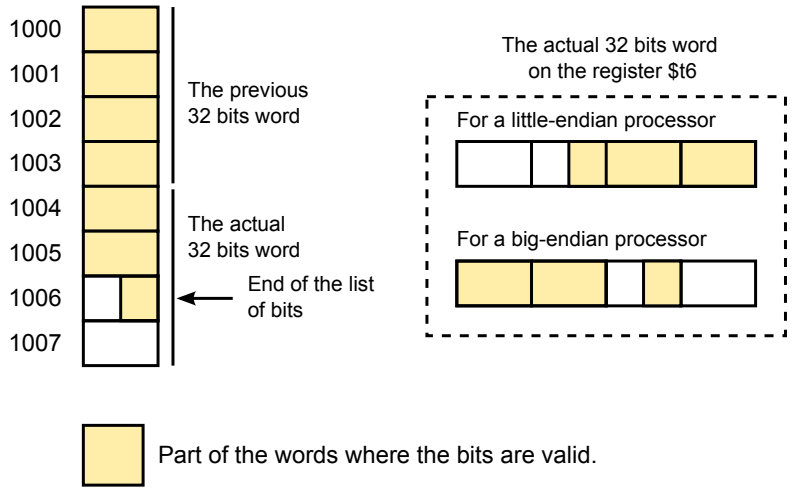


Figure 6: Words in memory for little/big5-endian processors

c) The `xor` operation can be expressed logically as follows:

$$\mathrm{A\ xor\ B} = \mathrm{A} \cdot \bar{\mathrm{B}} + \bar{\mathrm{A}} \cdot \mathrm{B} \tag{1}$$

We can thus replace instruction `xor t6, t3, t4` with the following sequence of instructions, the result is stored in `t6` as in the original instruction.

```
1       not t6, t3              # not(A) in t6
2       not s0, t4              # not(B) in s0
3       and s1, s0, t3          # A.not(B) in s1
4       and s0, t6, t4          # B.not(A) in s0
5       or  t6, s1, s0          # A.not(B) + B.not(A) in t6
```

[Exercise 17] Multiplication in Finite Fields

We would like to create a RISC-V routine that implements elementary operations for finite fields modular arithmetic, in binary representation. A finite field F(2^n) is the set of integers that can be represented on n bits.

When performing an addition in a finite field, each pair of bits is added independently, i.e., the carry is not propagated to the adjacent higher weight pair as done in an ordinary addition.

Therefore, an addition corresponds to a simple bit-wise **`xor`** operation. The following example compares an ordinary addition to one carried out in a finite field F(2^4):

Ordinary Addition	Addition in Finite Field F(2^n)
0 1 0 1	0 1 0 1
+ 0 0 1 1	+ 0 0 1 1
1 0 0 0	0 1 1 0

Multiplication in a finite field F(2^n) is carried out in two phases. The first phase consists in multiplying the two numbers similarly to an ordinary multiplication, i.e., by generating partial products then summing them. The addition of these partial products will yield a different result because this operation is defined differently in finite fields. This is illustrated by the following example:

Ordinary Multiplication	Multiplication in Finite Field F(2^n)
1 0 1 1	1 0 1 1
× 1 1 1 0	× 1 1 1 0
0 0 0 0	0 0 0 0
1 0 1 1	1 0 1 1
1 0 1 1	1 0 1 1
+ 1 0 1 1	+ 1 0 1 1
1 0 0 1 1 0 1 0	1 1 0 0 0 1 0

a) Write a RISC-V procedure that implements the first phase of the multiplication in a finite field. The procedure's arguments are two 16-bit operands in registers `a0` and `a1`. The 31-bit result is returned in register `v0`. Conventions regarding register use must be respected.

It can be observed that the result of the multiplication's first phase cannot be correct as it generally does not belong to the finite field F(2^4). This is why the second phase is

necessary. The second phase of a multiplication in a finite field F(2^4) consists in finding a result on n bits. To this effect we use a property of finite fields: in each finite field, a value $m \in$ F(2^n) is associated to 2^n. For example, in the finite field F(2^4), we can have $2^4 = m = 3$. Thanks to this property, all bits whose weight is greater or equal to n can be rewritten as a function of m and added to the result.

In the example we have F(2^4) and $2^4 = 3$:

$$\begin{aligned} 2^4 &= \quad 3 \quad &&= 0011 \\ 2^5 &= 2^4 << 1 &&= 0110 \\ 2^6 &= 2^4 << 2 &&= 1100 \end{aligned}$$

Starting from a certain power that depends on m, many iterations are necessary before having a result on n bits:

$$\begin{aligned} 2^7 = 2^4 << 3 = \quad 11000 &= 2^4 + 8 &&= 0011 + 1000 = 1011 \\ 2^8 = 2^4 << 4 = \quad 110000 &= 2^5 + 2^4 &&= 0110 + 0011 = 0101 \\ 2^9 = 2^4 << 5 = 1100000 &= 2^6 + 2^5 &&= 1100 + 0110 = 1010 \\ \ldots & & & \end{aligned}$$

Note that all sums are done according to the definition of the addition in the finite field. Given this property, we can replace each bit of the first phase's result whose weight lies between n and $2n-1$ with values obtained from m, so as to obtain a value that belongs to the finite field F(2^n). In the preceding multiplication example, if $2^4 = 3$, we obtain:

$$1100010 = 2^6 + 2^5 + 2 = 1100 + 0110 + 0010 = 1000$$

Thus, in the finite field F(2^4), with $2^4 = 3$, the result of the multiplication of 11 by 14 is 8.

b) In the finite field F(2^4), with $2^4 = 3$, multiply 5 by 13 giving all steps of the calculation.

c) In order to reduce the result to n bits, we can go through all bits whose weight lies between n and $2n-1$ and if their value is `'1'` we set it to `'0'` and add a value depending on m. Does it seem wiser to go through the bits starting from the one with the most significant weight or from the one with the least significant weight ? Why ? How should be generated the value that you will add ?

d) Create a RISC-V procedure that reduces a 32-bit number in a given finite field. The number is supplied as a parameter in register `a0` and could be the result of the multiplication's first phase returned by the routine created in the preceding question. The size n of the finite field is given as a parameter in register `a1`. The value m to be associated to 2^n is given as a parameter in `a2`. Register `a0` is used to return the result. Care must be taken to respect all conventions regarding register use.

[Solution 17] Multiplication in Finite Fields

a) Multiplication's first phase:

```
phase1: add  t3, zero, zero # partial sum
        add  t0, zero, a0
        add  t1, zero, a1
loop:   beq  t1, zero, return
        andi t2, t1, 1
        beq  t2, zero, next
        xor  t3, t3, t0
next:   slli t0, t0, 1
        srli t1, t1, 1
        j    loop
return: addi a0, t3, 0
        ret
```

b) The result of 5 times 13 in the finite field is given below:

$$
\begin{array}{rr}
 & 1\,1\,0\,1 \\
\times & 0\,1\,0\,1 \\
\hline
 & 1\,1\,0\,1 \\
 & 0\,0\,0\,0 \\
 & 1\,1\,0\,1 \\
+ & 0\,0\,0\,0 \\
\hline
 & 0\,1\,1\,1\,0\,0\,1
\end{array}
$$

$111001 = 2^5 + 2^4 + 9 = 0110 + 0011 + 1001 = 1100$

c) By going through the bits from right to left we run the risk of reintegrating bits with an index greater than $n-1$, and thus having to test the same bits several times. By going through them them from left to right this problem no longer exists, as there is no way of re-injecting new bits into already visited slots. The value to be added is initialized as follows: $r = (2^n + m) << (31 - n)$. Thus for the first iteration, if the most significant bit of the value to convert is `'1'`, performing a **`xor`** between this value and r will allow inverting the most significant bit and adding m (shifted to the right position). Then, r must be shifted by 1 bit to the right for the next iteration.

d) The code for the second phase is given below:

```
phase2: add  t3, zero, a0
        addi t0, zero, 1
        slli t1, t0, 31
```

```
4           sll  t0, t0, a1
5           add  t0, t0, a2      # t0 = 2^n + m
6           addi t2, zero, 31    # t2 = 31
7           sub  t2, t2, a1      # t2 = t2 - a1
8           sll  t0, t0, t2
9   loop:   srl  t2, t3, a1
10          beq  t2, zero, return
11          and  t2, t3, t1
12          beq  t2, zero, next
13          xor  t3, t3, t0
14  next:   srli t1, t1, 1
15          srli t0, t0, 1
16          j    loop
17  return: addi a0, t3, 0
18          ret
```

`t0` is the replacement value for the most significant bits. `t1` is the mask used to go through the bits whose index (weight) is greater than $n - 1$.

[Exercise 18] Square Root

We want to write a function in assembly that performs an integer square root operation. The algorithm to be implemented is represented with the control flow diagram shown in Figure 7. In the figure, the diamonds represent conditions and the rectangles represent operations. All operations are on integers and produce integer results.

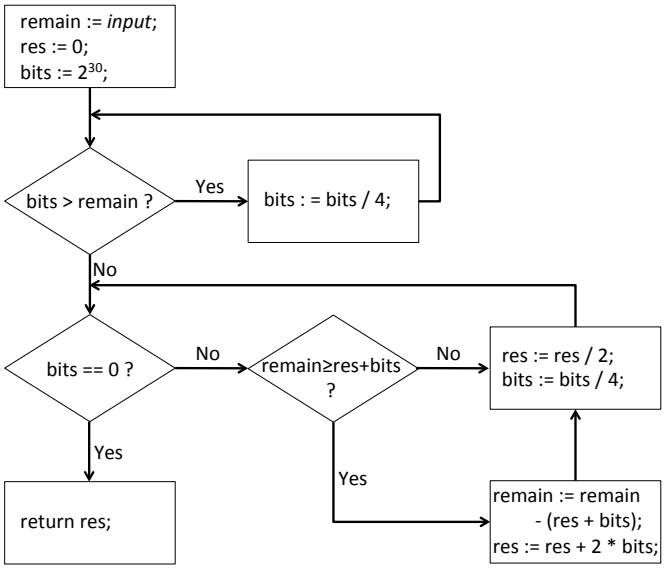


Figure 7: Control flow diagram of the algorithm

a) Perform the given algorithm with the *input* 100 by showing the values of *remain*, *res* and *bits* at each step (square boxes) of the algorithm. Show the results as a n x 3 table, where the rows represent the steps and columns represent *remain*, *res*, and *bits*, respectively.

b) Write an assembly function `sqrt` which implements the given algorithm. *input* is an unsigned integer and is passed to the function in the register `a0`. You should return the result of the function in the register `a0`. The usual conventions of the assembly should be respected.

c) Let us use this `sqrt` function to calculate the distance of a point from the origin in 2-dimensional space. Each coordinate of the points is represented by 16-bit half-word in *sign-and-magnitude* format. The x and y coordinates of the points are stored sequentially at the memory address pointed by `a0`. The result should be stored to the memory address pointed by `a1` in the same format. Assume a **big-endian** machine. **Reminder:** Distance of a point (x, y) from the origin is $\sqrt{x^2 + y^2}$.

Write a function `vectorlen` which works as described above. Suppose that we don't have access to *load-word* and *load-half-word* instructions and can use signed and unsigned versions of *load-byte* instruction (**`lb`** and **`lbu`**). Similarly, you have access to *store-byte* instruction (**`sb`**) to write into the memory. To simplify the access to the stack, you can use the macros **`push`** `rA` and **`pop`** `rA`, which adds an element to the stack and removes the last element from the stack, respectively. You can use **`mul`** `rC, rA, rB` instruction which stores the result of the multiplication of `rA` and `rB` into `rC`.

d) Is it possible to have *overflow* by the `vectorlen` function? If yes, explain which instructions can cause the overflow and why using an example.

[Solution 18] Square Root

a)

remain	res	bits
100	0	2^{30}
100	0	2^{28}
...		
100	0	2^{6}
36	128	2^{6}
36	64	2^{4}
36	32	2^{2}
0	40	2^{2}
0	20	2^{0}
0	10	0

b) The procedure is written in RISC-V. Note the unsigned comparisons.

```
1  sqrt:
2      add   t3, zero, zero # result
3      add   t0, a0, zero   # remain = input
4      addi  t1, zero, 1
5      slli  t1, t1, 30     # bits
6  align:
7      bgeu  t0, t1, loop
8      srli  t1, t1, 2      # bits /= 4
9      j     align
10 loop:
11     beq   t1, zero, end  # bits==0 -> end
12     add   t2, t3, t1     # t2 = results+bits
13     bltu  t0, t2, skip   # remain < t2 -> skip
14     sub   t0, t0, t2     # remain -= t2
15     slli  t2, t1, 1      # t2 = 2*bits
16     add   t3, t3, t2     # res+=t2
17 skip:
18     srli  t3, t3, 1      # res/=2
19     srli  t1, t1, 2      # bits/=4
20     j     loop
21 end:
22     addi  a0, t3, 0
23     ret
```

c) Before using the **mul** instruction to get the square of the coordinates, we would need

to convert them in 2's complement format. Yet, because we know that the square of x or $-x$ is the same, we can simply square the absolute value of the coordinates, which is extremely simple to find in sign and magnitude: we can simply mask out the sign and retain the mantissa before squaring.

```
vectorlen:
    addi  sp, sp, -4      # sp=sp-4
    sw    ra, 0(sp)       # save ra on the stack

    lbu   t0, 0 (a0)      # t0=xMSB
    slli  t0, t0, 8       # t0=t0<<=8
    lbu   t1, 1 (a0)      # t1=xLSB
    or    t0, t0, t1      # t0=x
    li    t3, 0x7FFF
    and   t0, t0, t3      # t0=magnitude(x)
    lbu   t1, 2 (a0)      # we do the same for
    slli  t1, t1, 8       # the 2nd coordinate
    lbu   t2, 3 (a0)
    or    t1, t1, t2
    li    t3, 0x7FFF
    and   t1, t1, t3      # t1=magnitude(y)
    mul   t0, t0, t0      # t0=x^2
    mul   t1, t1, t1      # t1=y^2
    add   a0, t0, t1      # a0=t0+t1
    jal   ra, sqrt            # a0=sqrt(a0)
    sb    a0, 1 (a1)      # we store the result
    srli  a0, a0, 8       # in memory
    sb    a0, 0 (a1)

    lw    ra, 0(sp)       # load ra from the stack
    addi  sp, sp, 4       # sp=sp+4
    ret
```

d) The only place where an overflow can occur is when we convert the result back into the sign-magnitude format on 16 bits. If we consider the maximum values possible of the coordinates $(2^{15}-1)$ we will get a vector length of $\sqrt{2\cdot(2^{15}-1)^2} = \sqrt{2}\cdot(2^{15}-1)$ whose magnitude cannot fit on 15 bits.

There is no other possible overflow. The sum of the maximum squares $(2\cdot(2^{15}-1)^2 < 2^{31})$ fits on 32 bits.

[Exercise 19] Matrix multiplication

In this question, you will implement the standard matrix multiplication algorithm in assembly and analyse the implementation from different perspectives. Some useful information about the matrices and matrix multiplication is given as Appendix at the end of question. Please refer to the section for relevant definitions.

The multiplication of two matrices is only defined when the number of columns of the first matrix equals to the number of rows of the second matrix. We calculate the multiplication of two matrices A and B as given in Algorithm 1. Basically, to get the element $C_{i,j}$ we get the inner product of i^{th} row of A and j^{th} column of B.

Algorithm 1 Calculate the multiplication of two matrices: $C = AB$

Require: A: $n \times m$ matrix
Require: B: $m \times s$ matrix
Ensure: C: $n \times s$ matrix

```
for i = 0 to n do
    for j = 0 to s do
        sum = 0;
        for k = 0 to m do
            sum += A_{i,k} · B_{k,j}
        end for
        C_{i,j} = sum;
    end for
end for
```

Using the given algorithm you can verify that

$$\text{if } A = \begin{bmatrix} -2 & 3 & 0 \\ 1 & -5 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix} \text{ then } C = AB = \begin{bmatrix} 13 \\ -7 \end{bmatrix}.$$

You will implement a RISC-V assembly procedure to multiply two square matrices using the given algorithm. Assume that we have two $N \times N$ integer (32-bit signed) matrices, A and B, stored consecutively in the memory. You can see how two 3×3 matrices are stored in memory starting from the address `0x1040` in Figure 8. Also assume that the size of the matrices, i.e., N, is loaded into the register `a0` and the start address of the first matrix, i.e., A, is loaded into the register `a1` before calling your procedure.

a) First, implement a procedure called `innerProduct` which will calculate the inner

Address	+0	+4	+8	+C
0x1040	1	7	-9	8
0x1050	3	2	-6	5
0x1060	4	1	6	0
0x1070	-5	4	-3	2
0x1080	-7	8		

Figure 8: Organization of the matrices $A = \begin{bmatrix} 1 & 7 & -9 \\ 8 & 3 & 2 \\ -6 & 5 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 6 & 0 \\ -5 & 4 & -3 \\ 2 & -7 & 8 \end{bmatrix}$ in the memory.

product of a given row of the first matrix with a given column of the second matrix. Assume that the address of the first element of the row (of the first matrix) is given in the register `a2` and the address of the first element of the column (of the second matrix) is given in the register `a3`. For example, if the caller wants to calculate the inner product of the second row of A with the third column of B in Figure 8, then it will load `0x104c` into `a2` and `0x106c` into `a3` before calling your procedure. You should return the result of the inner product in the register `a0`.

Note that you should push and pop the necessary registers to the stack to avoid overwriting the existing data. To simplify the access to the stack, you can use the macros **`push`** `rA` and **`pop`** `rA`, which adds an element to the stack and removes the last element from the stack, respectively.

You can use **`mul`** `rC, rA, rB` instruction which stores the least significant 32-bit of the result of the multiplication of `rA` and `rB` into `rC`.

b) Using the procedure you implemented in part **a)**, write a procedure called `matrixMultiplication` which stores the result of the multiplication, i.e., matrix C, into the memory area immediately following the second matrix. Before calling the procedure `innerProduct` do not forget to assign the values correctly to the registers `a2` and `a3`.

c) Discuss whether it is possible to have overflow in your code. If yes, indicate which instructions of your code could cause the overflow and why.

d) Assume that you are given another implementation of matrix multiplication which multiplies the first matrix A with the second matrix B which is now already stored in transposed form, i.e., B^T, in the memory. In order to multiply these two matrices, as opposed to the Algorithm 1, instead of calculating the inner products of rows of A with columns of B, we calculate the inner products of rows of A with rows of B^T, i.e., the line 5 of Algorithm 1 changes to "sum += $A_{i,k} \cdot B^T_{j,k}$".

Consider a system with 2-way set-associative cache with an LRU (Least Recently Used)

replacement policy. The cache is initially empty, has a capacity of 1024 bytes (256 words) and a four word block size. You have two 128×128 integer matrices stored consecutively starting from the address `0x1000` in the memory. Will the new implementation will perform better than the original implementation of Algorithm 1? Explain your answer by roughly comparing the number of cache hits/misses for each implementation.

Appendix:

An $n \times m$ integer matrix A is a rectangular array of integers, i.e., elements, represented in the form $A_{i,j}$, where $0 \leq i < n$ shows the row and $0 \leq j < m$ shows the column of the element. Here is an example of a 2×3 matrix:

$$A = \begin{bmatrix} -2 & 3 & 0 \\ 1 & -5 & 4 \end{bmatrix}.$$

From the definition, $A_{0,0} = -2$, $A_{1,2} = 4$, etc.

If $n = m$, then we call A a square matrix.

The inner product of two vectors v and w of length l is given by

$$p = \sum_{i=0}^{l-1} v_i \cdot w_i. \tag{1}$$

For example, if $v = \begin{bmatrix} 1 & 3 & 5 \end{bmatrix}$ and $w = \begin{bmatrix} -1 \\ 4 \\ -2 \end{bmatrix}$, then the inner product of v and w, i.e., $p = v \cdot w$, is calculated as $(1 \cdot -1) + (3 \cdot 4) + (5 \cdot -2) = 1$.

The transpose of the matrix A, represented by A^T, is obtained by writing the rows of A as columns of A^T and vice versa, i.e., $A^T_{j,i} = A_{i,j}$. For example, the transpose of the matrix A is given by

$$A^T = \begin{bmatrix} -2 & 1 \\ 3 & -5 \\ 0 & 4 \end{bmatrix}.$$

[Solution 19] Matrix multiplication

a)

```
innerProduct:
        add t6, zero, zero      # t6 will hold the result
        add t0, zero, a2        # iterator for the row
        add t1, zero, a3        # iterator for the column
        add t2, zero, zero      # counter (counts until t2 = n)
        slli t5, a0, 2          # t5 = 4*n

loopInner:
        lw  t3, 0(t0)           # loads the row element
        lw  t4, 0(t1)           # loads the column element
        mul t4, t3, t4          # multiplies the two elements
        add t6, t6, t4          # adds to the result
        add t1, t1, t5          # column iterator incremented by 4*n
        addi t0, t0, 4          # row iterator incremented by 4
        addi t2, t2, 1          # increment the counter
        bne t2, a0, loopInner   # loops until t2=n

        addi a0, t6, 0
        ret
```

b)

```
matrixMultiplication:
        # PUSH TO THE STACK
        addi sp, sp, -24
        sw   ra, 0(sp)
        sw   s0, 4(sp)
        sw   s1, 8(sp)
        sw   s2, 12(sp)
        sw   s3, 16(sp)
        sw   s4, 20(sp)

        slli s0, a0, 2      # s0 = 4*n
        mul  s1, s0, a0     # s1 = 4*n*n
        add  s2, a1, s1
        add  s4, s2, zero   # s4 <- end of row iterator
        add  s3, s2, s0     # s3 <- end of column iterator
        add  s2, s2, s1     # s2 <- result iterator (a1 + 8*n*n)
        add  a2, a1, zero   # a2 <- row iterator

```

```
19 matrixOuterLoop:
20         add  a3, a1, s1     # a3 <- column iterator
21
22 matrixInnerLoop:
23         addi sp, sp, -4     # Push a0 as it will be modified
24         sw   a0, 0(sp)      #  by innerProduct
25         jal  ra, innerProduct
26         sw   a0, 0(s2)      # update the memory with result
27         lw   a0, 0(sp)
28         addi sp, sp, 4
29         addi s2, s2, 4      # increment result iterator by 4
30         addi a3, a3, 4      # increment column iterator by 4
31         bne  a3, s3, matrixInnerLoop # inner loop
32
33         add  a2, a2, s0      # increment row iterator by 4*n
34         bne  a2, s4, matrixOuterLoop # outer loop
35
36         # POP FROM THE STACK
37         lw   ra, 0(sp)
38         lw   s0, 4(sp)
39         lw   s1, 8(sp)
40         lw   s2, 12(sp)
41         lw   s3, 16(sp)
42         lw   s4, 20(sp)
43         addi sp, sp, 24
44         ret
```

c) The possible instructions that can cause an overflow are the instructions given in 11^{th} and 12^{th} lines of `innerProduct` procedure. On line 11, if the result of the multiplication of two elements is greater than $2^{31} - 1$ or lower than -2^{31}, we will have an overflow. On line 12, if the accumulated sum of the multiplications exceeds $2^{31} - 1$, or is under -2^{31}, we will have an overflow. Note that, since the matrices could fit into the memory, the instructions used for address calculation can not cause an overflow.

d) We can approximately count the number of misses for each implementation. Let's start with the first implementation. When we are accessing the first matrix, i.e., A, $\frac{1}{4}$ of the first accesses to a row are miss, since each cache line is 4 words and only the access to the first word of a line is a miss. Then, the whole row will remain in the cache, thus all the other accesses to the same row will be hit. Hence, in total, $\frac{1}{4} \cdot 128 \cdot 128$ accesses to A are miss. Note that, there are $128 \cdot 128 \cdot 128$ accesses to A in total.

For the second matrix, i.e., B, since each element of a column of B will go to the same line on the cache (note that the cache is 2-way and one is already occupied by the

elements of first matrix), all the accesses will be miss, resulting in $128 \cdot 128 \cdot 128$ misses.

For the second implementation, the hits/misses for the first matrix, A, will have similar behaviour. However, the number of misses will be less for the second matrix, B. When we are multiplying the i^{th} row of A with j^{th} row of B^T, $\frac{1}{4}$ of the accesses to B^T will be miss because the cache is 2-way and the first one occupied by the i^{th} row of A while the second one is occupied by j^{th} row of B^T. As a result, $\frac{1}{4} \cdot 128 \cdot 128 \cdot 128$ misses will occur for accessing B^T.

After a simple calculation, we could conclude that the overall performance has increased, since the number of misses are reduced by approximately 4 times. Note that we ignored the accesses to the result matrix, i.e., C, in calculations, since it has minimal impact on the result (in total, we have $128 \cdot 128$ accesses to the matrix C, which is negligible compared to the number of accesses to the matrix B).

[Exercise 20] Polynomial Multiplication

In this question, you will implement a univariate polynomial (a polynomial in one variable) multiplication algorithm in assembly. A univariate polynomial is expressed as a sum of non-negative integral powers of a variable multiplied by coefficients. For example, $P(x) = c_n x^n + c_{n-1}x^{n-1} + ... + c_2 x^2 + c_1 x + c_0$ is a univariate polynomial of degree n. The product of two polynomials is obtained by multiplying the monomials term by term and combining the results of the same degree—e.g.,

$$\begin{aligned}(5x^3 - 7x - 2)(8x + 7) &= (5 \cdot 8)x^4 + (5 \cdot 7)x^3 + (-7 \cdot 8)x^2 + ((-7 \cdot 7) + (-2 \cdot 8))x + (-2 \cdot 7) \\ &= 40x^4 + 35x^3 - 56x^2 - 65x - 14.\end{aligned}$$

A simple polynomial multiplication algorithm is given below.

Algorithm 2 Calculate the multiplication of two polynomials: $R = PQ$

Require: P: a polynomial of degree n, i.e., $P = p_n x^n + ... + p_1 x + p_0$.
Require: Q: a polynomial of degree m, i.e., $Q = q_m x^m + ... + q_1 x + q_0$.
Ensure: R: a polynomial of degree $n + m$, i.e., $R = P \cdot Q = r_{n+m} x^{n+m} + ... + r_1 x + r_0$.

```
for i = 0 to n + m do {initialization of coefficients of R}
    r_i = 0
end for
for i = 0 to n do {calculate all coefficients of R}
    for j = 0 to m do
        r_{i+j} += p_i · q_j
    end for
end for
```

Implementation specifications:

- A polynomial is represented as an array of 32-bit signed integers corresponding to the coefficients, preceded by a 32-bit unsigned integer corresponding to the degree of the polynomial—i.e., $P(x) = c_n x^n + c_{n-1}x^{n-1} + ... + c_2 x^2 + c_1 x + c_0$ is represented as $[n, c_0, c_1, c_2, ..., c_{n-1}, c_n]$. For example, $5x^3 - 7x - 2$ is represented with words $[3, -2, -7, 0, 5]$, stored consecutively in memory.
- The starting addresses of the input polynomials in memory are given in registers `a0` and `a1`.
- The starting address of the output polynomial in memory is given in register `a2`. The result of the multiplication should be written in the $n + m + 2$ words starting from the address `a2`.

- You can use the instruction `mul rC, rA, rB`; it places in `rC` the least significant 32 bits of the result of the multiplication of `rA` and `rB`.
- Your procedures should not modify the value of any memory locations except the output array.

An example is given in Figure 9.

Address	+0	+4	+8	+C
0x1040	3	-2	-7	0
0x1050	5			
0x1060		1	7	8
0x1070		4	-14	-65
0x1080	-56	35	40	

Figure 9: Multiplication of the two polynomials $P = 5x^3 - 7x - 2$ and $Q = 8x + 7$ (both shown in white with a dark border). The inputs to the procedure are `a0` = 0x1040, `a1` = 0x1064 and `a2` = 0x1074. Your procedure should write the result in the memory as shown in the grey area.

a) First, implement a RISC-V assembly procedure called `initialize`, which will initialize all elements of the output array to zero (see lines 1–3 of Algorithm 2), considering the implementation specifications given above.

b) Now, implement another RISC-V assembly procedure called `mulPoly` which will perform the polynomial multiplication, again considering the implementation specifications given above. This procedure should be self-contained, so it should make a proper call to the `initialize` procedure written in the previous question.

c) Is it possible to have an arithmetic overflow in your code? If so, explain when (referencing precise instructions in your code) and why.

[Solution 20] Polynomial Multiplication

a)

```
initialize:
    lw      t0, 0(a0)                   # t0 = degree P
    lw      t1, 0(a1)                   # t1 = degree Q
    add     a0, t0, t1                  # a0 = degree R
    sw      a0, 0(a2)                   # store a0 to beginning of R

    addi    a0, a0, 2                   # calculate end address of R
                                        # (a0 = a2 + (a0+2) * 4)
    slli    a0, a0, 2
    add     a0, a0, a2

    addi    t2, a2, 4                   # iteration address of R
initialize_loop:
    sw      zero, 0(t2)                 # store 0
    addi    t2, t2, 4                   # increment
    bne     t2, a0, initialize_loop
    ret
```

b)

```
mulPoly:
    addi    sp, sp, -12             # push to the stack
    sw      ra, 0(sp)
    sw      s0, 4(sp)
    sw      s1, 8(sp)

    lw      s0, 0(a0)               # s0 = degree P
    lw      s1, 0(a1)               # s1 = degree Q
    jal     ra, initialize          # call the initialization

    addi    s0, s0, 2               # calculate end address of P
    slli    s0, s0, 2
    add     s0, s0, a0

    addi    s1, s1, 2               # calculate end address of Q
    slli    s1, s1, 2
    add     s1, s1, a1

    addi    t0, a0, 4               # iteration address of P
    addi    t1, a1, 4               # iteration address of Q
```

```
21      addi    t2, a2, 4               # iteration address of R
22      addi    t3, a2, 4               # iteration address of R for
23                                      # outer loop
24  mulPoly_outer:
25      lw      t4, 0(t0)               # load coefficient from P
26  mulPoly_inner:
27      lw      t5, 0(t1)               # load coefficient from Q
28      mul     t5, t5, t4              # multiply coefficients
29      lw      t6, 0(t2)               # load existing coefficients
30                                      # from R
31
32      add     t5, t5, t6              # add them
33      sw      t5, 0(t2)               # store back the calculated
34                                      # coefficient
35
36      addi    t1, t1, 4               # increment t1
37      addi    t2, t2, 4               # increment t2
38      bne     t1, s1, mulPoly_inner
39      addi    t0, t0, 4               # increment t0
40      addi    t1, a1, 4               # t1 = a1 + 4 (t1 goes to the
41                                      # beginning of Q)
42
43      addi    t3, t3, 4               # increment t3
44      add     t2, t3, zero            # t2 = t3 (t2 goes to the iterated
45                                      # beginning of R)
46
47      bne     t0, s0, mulPoly_outer
48
49      lw      s1, 8(sp)               # pop from the stack
50      lw      s0, 4(sp)
51      lw      ra, 0(sp)
52      addi    sp, sp, 12
53
54      ret
```

c) Yes, lines 28 and 32 of the `mulPoly` procedure can cause an arithmetic overflow. These instructions are the multiplication of two coefficients and the addition of its result to the previous multiplications, respectively. If the coefficients are sufficiently large, an overflow is possible. The other arithmetic operations in the code do not cause an overflow since they are used for address calculations and we can assume that there is no memory overflow (i.e., all the arrays can fit into the memory).

[Exercise 21] Least Common Multiple

The *Least Common Multiple* (LCM) of two positive integers a and b is the smallest positive integer that is divisible by both a and b. Similarly, the LCM of a given array x_1, ..., x_n is the smallest positive integer that is divisible by each x_i, where $1 \leq i \leq n$.

Consider the following algorithm for computing the LCM of an array. Start from the smallest prime number $p = 2$, and try to divide all numbers of the array by p in the first iteration. For each element of the array, if the remainder of the division is 0, then you substitute that element with the integer quotient (the result of the division). Otherwise, if the remainder is not 0, then you keep the original value. If at least one number was divisible by p, then you try to divide by p again. Otherwise, you take the next prime number and you iterate with it. The iterations end when all numbers in the array are equal to 1, meaning that they were all factorised using their divisors. The LCM is computed as a product of the prime numbers used in the iterations in which at least one successful division was performed.

For example, consider the following array.

2	84	12	18	14

The different execution steps of the algorithm are shown with the following table:

Iteration	Divided by	Array				
		2	84	12	18	14
1	2	1	42	6	9	7
2	2	1	21	3	9	7
3	2	1	21	3	9	7
4	3	1	7	1	3	7
5	3	1	7	1	1	7
6	3	1	7	1	1	7
7	5	1	7	1	1	7
8	7	1	1	1	1	1

Since the LCM is computed as the product of the numbers with which we performed at least one division in each iteration, the LCM of the given array is $2 \cdot 2 \cdot 3 \cdot 3 \cdot 7 = 252$. For instance, the prime number 5 is not included in the product, because none of the numbers was divisible by 5.

In this exercise, you will incrementally implement a RISC-V assembly procedure that computes the LCM of a given array A with n elements. Assume that A contains un-

signed 32-bit numbers stored on consecutive locations in the memory. Although it is atypical for a RISC-V processor, assume that in this implementation the memory is word-addressed. Make sure that your code respects all conventions regarding register use.

Assume that the quotient and the remainder of the division can be computed using the **div** instruction. This instruction saves the integer quotient and the remainder in two dedicated registers, `Lo` and `Hi`, respectively. To obtain the values saved in `Lo` and `Hi` use the instructions **mfhi** and **mflo**, respectively, as shown in this example:

```
1    div  t1, t2    #  Lo = t1 / t2   (integer quotient)
2                   #  Hi = t1 mod t2 (remainder)
3    mfhi t3        #  t3 = Hi
4    mflo t4        #  t4 = Lo
```

Similarly, for the multiplication, you can use the pseudo instruction **mul**, which operates on two 32-bit numbers and stores the result in a given 32-bit register:

```
1    mul  t5, t1, t2  #  t5 = t1 * t2  (multiplication)
```

a) First, implement a procedure called `divide_all`, which receives, as input, the following parameters:

- The starting address of an array A using the register `a0`;
- The number of elements, n, using the register `a1`;
- A positive integer p using the register `a2`.

This procedure divides each element of A by p and it replaces the element with its quotient only if the remainder of the division is 0. For example, assume that $p = 3$ and the array A before calling the procedure `divide_all` is as follows:

1	25	9	11	63	385	3	93

Then, at the end of the procedure, A has the following elements:

1	25	**3**	11	**21**	385	**1**	**31**

This procedure returns two values:

- In the register `a0`, it returns 1 if at least one element in the array was changed. Otherwise, it returns 0.
- In the register `a1`, it returns 1 if, at the end of the procedure, all elements of A are equal to 1. Otherwise, it returns 0.

b) Now, implement the main procedure, called `lcm`, which computes the LCM of a given array of numbers A, using the described algorithm. The computed LCM is returned using the register `a0`. The main procedure receives as input the following parameters:

- The starting address of an array A, given in the register `a0`.
- The number of elements n, of A, given in the register `a1`.
- The starting address of an array P, which contains the prime numbers saved in ascending order, given in the register `a2`.
- The number of elements of the array P, given in the register `a3`. If P does not have a sufficient number of elements to compute the LCM of A, then the `lcm` procedure returns -1 using the register `a0`.

For implementing the procedure `lcm` you should use the procedure `divide_all` from part 1 and remember that both procedures must obey all conventions regarding register use and register saving between function calls.

c) Is it possible to have an arithmetic overflow in your code? If yes, indicate which instruction(s) could cause the overflow and explain why.

[Solution 21] Least Common Multiple

a) Following is the RISC-V code of the procedure `divide_all`.

```
divide_all:
    addi    t5, zero, 0     # nothing is changed
    addi    t6, zero, 1     # all elements are 1
    addi    t0, zero, 0     # counter = 0
    addi    t1, a0, 0       # get the start address
    addi    t4, zero, 1     # t4 = 1

da_loop:
    beq     t0, a1, da_end  # end of array?
    lw      t2, 0(t1)       # t2 = mem[t1]
    beq     t2, t4, da_inc_counters
    div     t2, a2
    mfhi    t3              # get the remainder
    bne     t3, zero, da_check_one
    mflo    t2              # get the quotient
    sw      t2, 0(t1)       # mem[t1] = t2
    addi    t5, zero, 1     # save that a change is made
da_check_one:
    beq     t4, t2, da_inc_counters
    addi    t6, zero, 0     # found an element different than 1
da_inc_counters:
    addi    t0, t0, 1       # increment counter
    addi    t1, t1, 1       # increment address (word-addressed)
    j       da_loop

da_end:
    addi a0, t5, 0
    addi a1, t6, 0
    ret
```

b) Following is the RISC-V code of the procedure `lcm`, which computes the least common multiple.

```
lcm:
    addi sp, sp, -6
    sw   ra, 0(sp)
    sw   s0, 1(sp)
    sw   s1, 2(sp)
    sw   s2, 3(sp)
    sw   s3, 4(sp)
```

```
8       sw   s4, 5(sp)
9
10      addi s0, zero, 1          # for computing the LCM
11      addi s1, a2, 0            # get the start address of P
12      addi s2, zero, 0          # counter = 0
13      addi s3, a0, 0            # Remember a0 and a1 as it will
14      addi s4, a1, 0            #  be modified by function calls
15
16  lcm_loop:
17      beq  s2, a3, lcm_error    # not enough prime numbers?
18      addi a0, s3, 0            # Prepare function arguments
19      addi a1, s4, 0
20      lw   a2, 0(s1)            # get the prime number
21      jal  ra, divide_all
22      beq  a0, zero, lcm_no_change
23      mul  s0, s0, a2           # s0 = s0 * a2
24      j    lcm_check_one
25  lcm_no_change:
26      addi s1, s1, 1            # increment address
27      addi s2, s2, 1            # increment counter
28  lcm_check_one:
29      beq  a1, zero, lcm_loop   # check if all elements are one
30      j    lcm_end
31
32  lcm_error:
33      addi s0, zero, -1         # error: return -1
34
35  lcm_end:
36      addi a0, s0, 0            # get the return value
37      lw   ra, 0(sp)
38      lw   s0, 1(sp)
39      lw   s1, 2(sp)
40      lw   s2, 3(sp)
41      lw   s3, 4(sp)
42      lw   s4, 5(sp)
43      addi sp, sp, 6
44
45      ret
```

c) The instruction **`mul`** is the only instruction that can cause overflow. An overflow can occur when the LCM of the numbers in the given array is greater than $2^{32} - 1$.

[Exercise 22] Palindrome binary numbers

A palindrome binary number is a binary number that reads the same backwards as forwards. For example, the following two 8-bit numbers are palindromes: `01100110` and `10111101`.

In this exercise, you are to write an assembly program that reads an array of 8-bit numbers (**array of bytes**), detects the bytes that are palindromes, and creates a bit-vector in which a bit in the position `i` is set only if the byte `i` of the array is a palindrome (Figure 10). For an array of **N** bytes, the output bit-vector is **N**-bit long.

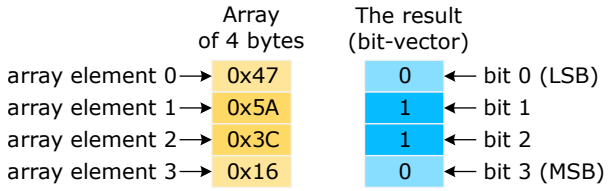


Figure 10: An example of an array of four bytes and its corresponding bit-vector. Array elements 1 and 2 are palindromes. Consequently, bits 1 and 2 in the output bit-vector are set to one.

The array starts at address **ARRAY**. The number of bytes in the array is always a multiple of 32 and it is stored in a 32-bit word located at address **ARRAY_SIZE**. The resulting bit-vector should be stored in memory, starting from address **RESULT**. The values of these three symbolic constants are **smaller than 0xFFFF**.

Instructions:

- Assume a **little-endian** machine.
- To access the memory you are allowed to use **only load-word and store-word** instructions.
  These access 32-bit words.
- Your code should conform to the assembly coding conventions.

a) Consider the following array of 32 bytes (**ARRAY_SIZE** = 32), with palindrome numbers appearing in **bold**:

0x12, **0xFF**, **0x3C**, 0x11, 0x54, **0x42**, 0xA0, 0xAA,
0x00, 0xDE, 0xAD, 0xBE, 0xEF, **0xA5**, **0x5A**, 0x13,
0x00, **0x00**, ..., **0x00**

Draw the two tables shown in Figure 11, **strictly respecting the given format**.

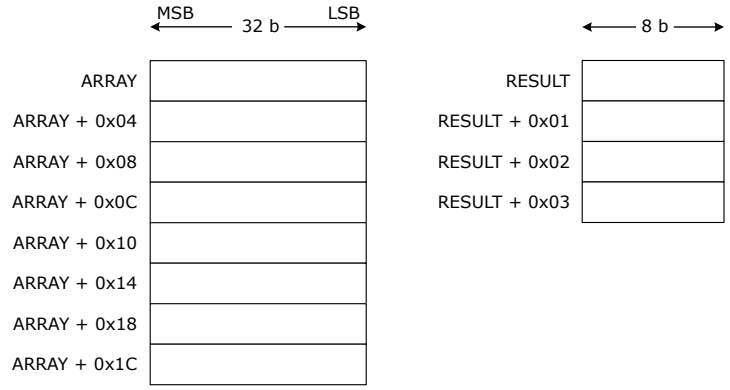


Figure 11: A view on the memory.

In the table on the left, show the content of memory from address **ARRAY** to **ARRAY + 0x1C**, assuming that the above sequence is stored in memory in the exact same order as above and starting from address **ARRAY**.

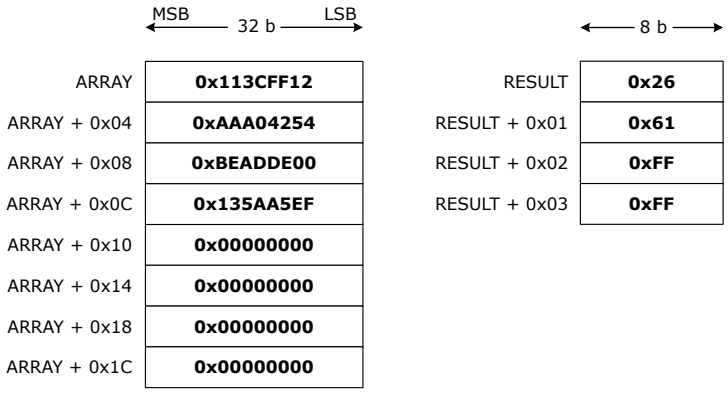
In the table on the right, show the content of memory from address **RESULT** to **RESULT + 0x03** after the execution of the program.

b) An easy way to detect whether a binary number is a palindrome is to first reverse it and then compare it with the original binary number. If the original and the reversed binary numbers are the same, then the original binary number is a palindrome. Write a function `invert_byte`, which takes as its argument an 8-bit binary number and returns this number **reversed**. For example, if the function argument is `11010000`, the function should return `00001011`.

c) Write the program's `main` function, which traverses the array of bytes, identifies if a byte is palindrome using the `invert_byte` function, and creates a corresponding bit-vector as described above.

[Solution 22] Palindrome binary numbers

a)



b)

```
# symbolic constants
.equ LSB_MASK    0xFF     # mask to extract the LSB

# storing data in memory
.data
ARRAY_SIZE:
.word 96
ARRAY:
.byte 0x12   # 0
.byte 0xFF   # 1, palindrome
.byte 0x3C   # 2, palindrome
.byte 0x11   # 3
.byte 0x54   # 4
.byte 0x42   # 5, palindrome
.byte 0xA0   # 6
.byte 0xAA   # 7
.byte 0x00   # 8, palindrome
.byte 0xDE   # 9
.byte 0xAD   # A
.byte 0xBE   # B
.byte 0xEF   # C
.byte 0xA5   # D, palindrome
.byte 0x5A   # E, palindrome
.byte 0x13   # F
.byte 0x54   # 10
.byte 0x42   # 11, palindrome
.byte 0xA0   # 12
.byte 0xAA   # 13
.byte 0x12   # 14
.byte 0xFF   # 15, palindrome
.byte 0x3C   # 16, palindrome
.byte 0x11   # 17
.byte 0xEF   # 18
.byte 0xA5   # 19, palindrome
.byte 0x5A   # 1A, palindrome
```

```
.byte 0x13  # 1B
.byte 0x00  # 1C, palindrome
.byte 0xDE  # 1D
.byte 0xAD  # 1E
.byte 0xBE  # 1F
RESULT:
.word 0x0000
.word 0x0000
.word 0x0000

# storing program
.text

#-----------------------------------------------------------------------------
main:
#-----------------------------------------------------------------------------

    lw      s0, ARRAY_SIZE(zero)    # s0: array size
    add     s1, zero, zero          # s1: current array index, initialized to zero

loop:
    bge     s1, s0, end             # if current array index == array size,
                                    # we're done traversing the array

load_word:
    lw      s2, ARRAY(s1)           # s2: word currently processed
    add     s3, zero, zero          # s3: byte counter, initialized to 0.
                                    # The correct range is [0-3].

loop_over_bytes_within_word:
    slli    t0, s3, 3               # the number of bits to shift right to move
                                    # this byte to the LSB position
                                    # t0 = s3 * 8 = s3 << 3
    srl     s4, s2, t0              # shift right for t0 bits
    andi    s4, s4, LSB_MASK        # keep only the LSB the word currently processed
    addi    a0, s4, 0               # Prepare function arguments

    jal     invert_byte             # call the function that will invert this byte.
                                    # Function takes argument in a0 and stores the
                                    # result in v0

    bne     s4, a0, next_byte       # once the byte is inverted, check if v0 and
                                    # a0 are the same.
                                    # If yes, we found a palindrome and the result
                                    # needs to be updated.
                                    # Else, the next byte should be looked into.
palindrome_found:
    srli    t1, s1, 5               # t1: s1 / 32 = index of the word in RESULT
                                    # array
    slli    t1, t1, 2               # align the word address offset on word boundary
    lw      t2, RESULT(t1)          # t2 = load current word from RESULT array
```

```
    andi    t3, s1, 31              # location of bit within the word = array
                                    # index modulo 32
    addi    t4, zero, 1             # t4: one bit mask, used to set one bit in
                                    # the RESULT array
    sll     t4, t4, t3              # t4: all zeros but one '1' at the position t3
    or      t2, t2, t4              # set one bit in result array
    sw      t2, RESULT(t1)          # store the new value of the word

next_byte:
    addi    s1, s1, 1               # increment array index
    addi    s3, s3, 1               # increment byte counter

    li      t0, 4
    bne     s3, t0, loop_over_bytes_within_word

next_word:
    j       loop

end:
    li      a0, 10                  # END
    ecall

#-----------------------------------------------------------------------------
invert_byte:
#-----------------------------------------------------------------------------
    add     t0, a0, zero            # BEGIN invert_byte. t0: initialized to a0
    add     a0, zero, zero          # a0: output
    addi    t1, zero, 8             # t1: bit counter, initialized to 8

loop_over_bits_within_byte:
    beq     t1, zero, return        # count down

    andi    t2, t0, 1               # t2: lowest-order bit from the input
    slli    a0, a0, 1               # shift output left by 1 bit
    or      a0, a0, t2              # append the extracted bit (t2) to the result

    srli    t0, t0, 1               # consume lowest-order bit of input
    addi    t1, t1, -1              # decrement bit counter

next_bit:
    j       loop_over_bits_within_byte

return:
    ret                             # END invert_byte
```

[Exercise 23] The Game of Life

The **Game of Life** is a cellular automaton devised by British mathematician John Conway in 1970. The game requires no players: its evolution is determined by its initial state (also called the seed of the game). The playing field of the game is an infinite two-dimensional grid of cells, where each cell is either alive or dead. At each time step, the game evolves following a set of rules:

- **Underpopulation**: any living cell dies if it has (strictly) fewer than two live neighbours.
- **Overpopulation**: any living cell dies if it has (strictly) more than three live neighbours.
- **Reproduction**: any dead cell becomes alive if it has exactly three live neighbours.
- Any live cell remains alive if it has two or three live neighbours.

Every cell has eight neighbours (cells that are horizontally, vertically, or diagonally adjacent).

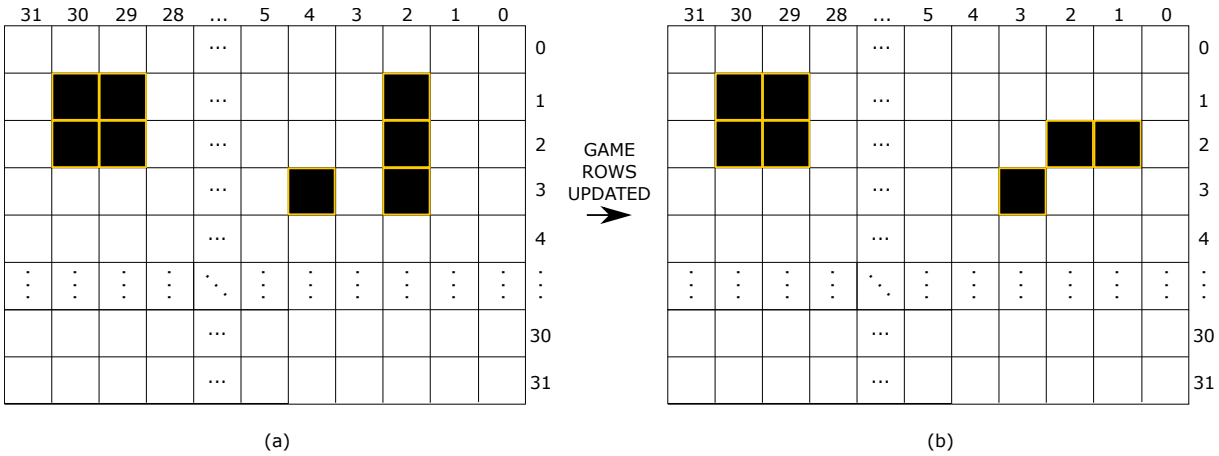


Figure 12: Game evolution between two successive time steps: (a) current time step and (b) next time step. Live cells are black. Dead cells are white.

Figure 12 illustrates the game in two subsequent time steps: (a) current state and (b) next state.

Current state: there are eight alive cells, marked in black, at the following (row, column) coordinate pairs: (1, 2), (1, 29), (1, 30), (2, 2), (2, 29), (2, 30), (3, 2), and (3, 4). All the other cells are dead (in white).

Next state: Cells (1, 29), (1, 30), (2, 29), and (2, 30) remain living as they all have three living neighbors. Cell (2, 2) has two living neighbors, which is sufficient for it to remain alive. Cell (3, 4) has no living neighbor, so it dies. Cells (1, 2) and (3, 2) have only one living neighbor, which is insufficient to continue living and so they die. Dead cells (3, 3) and (2, 1) have exactly three living neighbors, so they become alive. As a result, the game state in the next time step (also called the next state) contains seven living cells: (1, 29), (1, 30), (2, 29), (2, 30), (2, 1), (2, 2), and (3, 3).

Note that the next game state is determined solely based on the current game state. The transition from the current to the next game state is completed once the decision whether a cell should live or die is made for all the game cells.

Your task is to write a program in assembly that plays this game on a 32×32 game field. To imitate an infinite size of game field, you should assume that all border cells (leftmost column, rightmost column, top row, and bottom row) are dead and can never become alive. As a consequence, the active cells (those that can live or die) are to be limited to a 30×30 field.

The state of the game is kept in memory as an array of words (game array), each word being a 32-bit value representing one row of the game field. If a bit i of the word is set (1), the cell at the corresponding row and the column i is considered alive. Otherwise, that cell is considered dead. The game array is stored starting from the address 0x1000. The row indices grow towards higher addresses. See Figure 13.

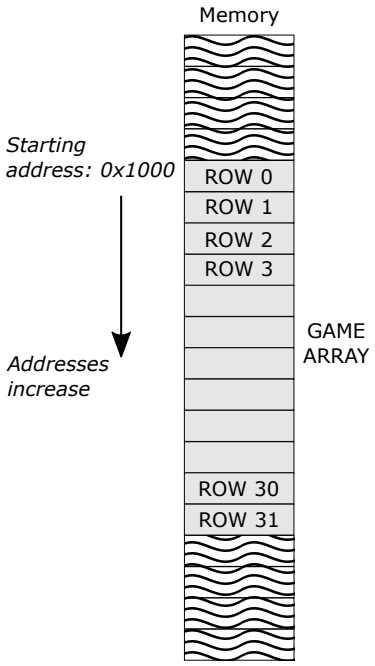


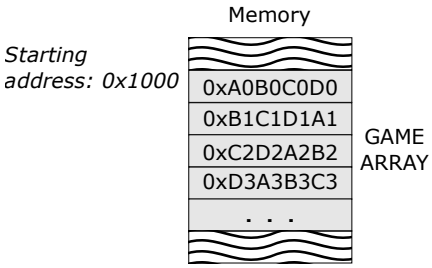
Figure 13: Game array in memory.

Instructions:

- Memory is byte addressed.
- To access memory, you are allowed to use **only** load word and store word instructions.
- Assume a **little-endian** machine.
- You are **not** allowed to use any multiplication instruction.
- Your code should conform to the assembly calling conventions.

a)

Assuming that the first four words in the game state array are 0xA0B0C0D0, 0xB1C1D1A1, 0xC2D2A2B2, and 0xD3A3B3C3, write the values of **bytes** (in hexadecimal) stored at the following memory addresses: 0x1003, 0x1004, 0x1007, 0x1008, 0x100B, and 0x100C. Your answer must be of the format `mem[0x1003] = value, mem[0x1004] = value`, etc.



b) Write the function `cell_fate` which tells if a cell should live or die, following the game rules. The function takes the number of living neighboring cells and the state of the cell itself (`1` if alive, `0` if dead), to return `1` if the cell should live in the next game iteration or `0`, if the cell should die.

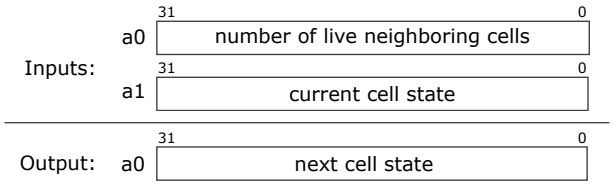


Figure 14: Inputs and outputs of `cell_fate` function.

c) Write the function `update_row`, which takes as arguments three subsequent rows of the game array (upper, middle, lower), to compute the next value of the middle row and return it. To update the value of individual cells in one row, you should call `cell_fate`.

- Describe in detail your algorithm and implementation of `update_row` function. Note that you do not necessarily need to provide a highly optimized implementation.
- Write the assembly code of `update_row` function.

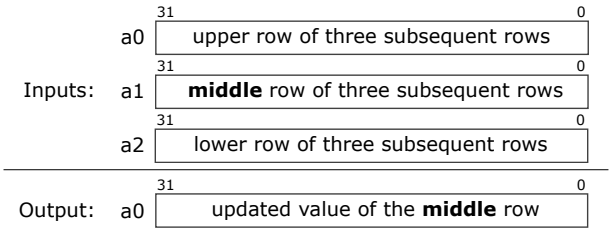


Figure 15: Inputs and outputs of `update_row` function.

d) Write the `main` loop of you program, which executes one game iteration. This loop iterates over all game rows and updates the game array in memory. The `main` loop should call the function `update_row`.

Note that the next state should always depend on the current one: be careful not to use an already updated line as input argument to the function `update_row`.

e) Write all the missing lines to complete your program.

Stack memory starts at the address `0x1200` and grows towards lower addresses. Once the program is launched, the game should play indefinitely. Between every two game iterations, a delay (a pause) of $\approx 2^{20}$ instructions should be made. The delay should not exceed the following range $(2^{20} - 0.001\%, 2^{20} + 0.001\%)$. You can assume that the game array is already initialized in memory when the program is launched.

[Solution 23] The Game of Life

a)

```
mem[0x1003] = A0
mem[0x1005] = A1
mem[0x1007] = B1
mem[0x1008] = B2
mem[0x100B] = C2
mem[0x100C] = C3
```

b)

```
# cell_fate
# Decides if a cell shall live, depending on its and its
# neighbours current states
#
# Arguments:
# - a0 = number of living neighbours
# - a1 = current state of the cell
# Returns:
# - a0 = next state of the cell
#
cell_fate:
    add  t3, zero, zero  # Default answer is 0 (dead)
    addi t0, zero, 1     # Creating registers to use in the next
                         # comparisons
    addi t1, zero, 2
    addi t2, zero, 3

    # If the cell has 3 living neighbours, it will live no
    # matter its current state.
    # CASE III (see comments at the end of this file for the
    # rules)
    beq a0, t2, cell_will_live

    # If the cell is already dead, it will not live (3
    # neighbours case excluded thanks to the previous
    # condition).
    # CASES I.1, II.1, IV.1
    bne a1, t0, cell_will_die

    # If the cell hasn't got 2 neighbours, it will die no
```

```
    # matter its current state.
    # CASES I.2, IV.2
    bne a0, t1, cell_will_die

    # CASE II.2 is implicit

    cell_will_live:
        addi t3, zero, 1 # Indicate that the cell will live
    cell_will_die:
        addi a0, t3, 0
        ret

# Rules of the game of life:
# 0-1 neighb.: cell dies (0->0, 1->0)    CASE I.1  / CASE I.2
# 2   neighb.: status quo (0->0, 1->1)   CASE II.1 / CASE II.2
# 3   neighb.: cell lives (*->1)         CASE III
# 4-8 neighb.: cell dies (0->0, 1->0)    CASE IV.1 / CASE IV.2
```

c)

```
# update_row
# Updates a row using its 3 relevant rows (upper, current and
# lower)
#
# Arguments:
# - a0: upper line
# - a1: current line
# - a2: lower line
# Returns:
# - a0: updated current (a1) line
#
update_row:
    addi sp, sp, -8 # Save registers
    sw   s0, 0(sp)
    sw   s1, 4(sp)

    addi s0, zero, TO_HANDLE # Set bit counter to 30, first bit
                             # to handle
    add  s1, zero, zero       # The updated line will be saved
                             # here

    # The main loop considers each of the 30 cells individually.
    bit_loop:
        add  t1, a0,   zero      # Copy rows from arguments
```

```
25          add  t2, a1,   zero
26          add  t3, a2,   zero
27          add  s2, zero, zero      # Set neighbour counter
28          addi t6, zero, 3         # Set row counter
29
30          # The next loops will iterate over all 9 bits of the
31          # current "zone".
32
33          # This is the loop that iterates over the 3 lines
34          # neighbouring the focused bit.
35          row_loop:
36              add  t0, zero, t1    # Advance in row list
37              add  t1, zero, t2
38              add  t2, zero, t3
39              addi t3, s0, -1      # The first neighbour is
40                                   # before the focused bit.
41              srl  t0, t0, t3      # Get to first neighbour
42              addi t5, zero, 3     # Set column counter
43
44              # This is the loop that iterates over the columns
45              # of the line.
46              col_loop:
47                  andi t4, t0, 1   # Isolate neighbour
48                  add  s2, s2, t4  # Add to counter
49                  srli t0, t0, 1   # Select next neighbour
50
51                  addi t5, t5, -1  # Update column counter
52                  bne  t5, zero, col_loop
53
54              addi t6, t6, -1      # Update row counter
55              bne  t6, zero, row_loop
56
57          # The next lines are here to subtract the value of the
58          # focused bit from the current neighbour count.
59          add  t0, a1, zero
60          srl  t0, t0, s0
61          andi t0, t0, 1
62          sub  s2, s2, t0
63
64          # Cell fate call + saving/restoring important registers
65          addi sp, sp, -12
66          sw   a0, 0(sp)
67          sw   a1, 4(sp)
68          sw   ra, 8(sp)
```

```
        add  a0, s2, zero # Current neghbour count
        add  a1, t0, zero # Current cell state
        jal  ra, cell_fate
        lw   a0, 0(sp)
        lw   a1, 4(sp)
        lw   ra, 8(sp)
        addi sp, sp, 12

        add  s1, s1, a0  # Add the new cell to the
                         # updated line
        slli s1, s1, 1   # Shift it to make room for the
                         # next bit

        addi s0, s0, -1          # Update bit counter
        bne  s0, zero, bit_loop  # Bit 0 does not interest us
                                 # (wall).

    # Return updated line and restore saved registers.
    add  a0, s1, zero
    lw   s0, 0(sp)
    lw   s1, 4(sp)
    addi sp, sp, 8
    ret
```

d) and **e)**

```
.equ ROWS, 0x1000
.equ STACK, 0x1200

.equ TO_HANDLE, 30


la   sp, STACK   # Set value of stack pointer

# main
# Executes the update code in an infinite loop.
#
main:
    add  s0, zero, zero          # Set address counter to 0
    addi s1, zero, TO_HANDLE     # Only 30 lines interest us
    slli s1, s1, 2               # Counter counts by 4, so
                                 # adapt upper bound

    la   t0, ROWS              # Start at ROWS
```

```
19      add  s0, s0, t0
20      add  s1, s1, t0
21
22      lw   a1, 0(s0)                  # Load rows 1 and 2
23      lw   a2, 4(s0)
24
25      main_loop:
26          add  a0, a1, zero         # Go to next row
27          add  a1, a2, zero
28          lw   a2, 8(s0)        # Only need to load the newest
29                                    # row
30
31          jal  ra, update_row    # Update current (a1) row
32
33          addi s0, s0, 4            # Increment counter by 4 (for
34                                    # next address)
35          sw   a0, 0(s0)        # Store the last-handled row
36                                    # in memory
37          beq  s0, s1, main_wait    # Wait for 2^20 instructions
38                                    # when finished
39          j main_loop
40
41  # main_wait
42  # Waits for ca. 2^20 instructions.
43  #
44  main_wait:
45      # Shift by 19, not 20, because there are 2 instructions
46      # checked by decrementation!
47      addi t0, zero, 1
48      slli t0, t0, 19
49
50      # Waiting loop
51      main_wait_loop:
52          addi t0, t0, -1
53          bne t0, zero, main_wait_loop
54
55      j  main      # Restart row update
```

Part II: Processor, I/Os, and Exceptions

[Exercise 1]

A room must be warmed up to keep its temperature within 17°C and 21°C. For that you have a processor (assume a little-endian machine) that has access to 2 peripherals, a sensor and a switch mapped at addresses 0x8800 and 0x8900, respectively. The sensor provides the room's current temperature and can be configured to raise an interrupt request when this temperature crosses a user-defined threshold. The switch is connected to an electric radiator and has only 2 states: on and off. When it is on, the radiator warms up the room, when it is off the heat dissipates slowly.

The following table lists the sensor registers.

Register	RW	31 . . . 9	8	7 . . . 2	1	0
`status`	RW	*Unused*	IR	Temp		
`control`	RW	*Unused*			IH	IL
`tmax`	RW	Unused		Temp max		
`tmin`	RW	Unused		Temp min		

The status register has a Temp field reporting the current temperature in °C on 8 bits and is signed. The IR bit reads at 1 if an interrupt request is pending. Writing 0 to the status register clears the IR bit (but does not modify the Temp field)

The control register is used to activate or deactivate interrupts coming from the sensor. When IL/IH is set to 1, the sensor raises an interrupt if the current temperature is lower/higher than the temperature defined by the tmin/tmax register, respectively.

The switch has a very simple interface, depicted in the following table. Writing 0/1 to the ON bit will turn off/on the radiator, respectively.

Register	RW	31 . . . 1	0
`status/control`	RW	*Unused*	ON

a) In RISC-V Assembly, write the `init_sensor` function, which should initialize the threshold temperatures of the sensor interface and, depending on the current temperature, activate interrupts coming from the sensor and initialize the state of the switch.

b) In RISC-V Assembly, write the `sensor_isr` interrupt service routine which is called upon an interrupt from the sensor. This routine is used to make sure that the room temperature stays within 17Â°C and 21Â°C; it activates the radiator when the temperature is too low, deactivates it when it is too high and configures the sensors for the next interrupt.

[Solution 1]

For this solution to work, we assume that `init_sensor` and `sensor_isr` are the only functions accessing the switch and the sensor interfaces.

Note that we use the instruction **`lb`** to load the current temperature from the `status` register. This instruction extends the 8-bit signed value on 32 bits. We assume a *Little Endian* model.

If the `lb` instruction is not available, we could instead use the following:

```
    la  t6, SENSOR
    lw  t1, 0(t6)        # load word
    slli t1, t1, 24      # only keep the LSByte
    srai t1, t1, 24      # extend sign
```

a) The `init_sensor` function below shows a straightforward solution.

```
.equ SENSOR, 0x8800
.equ RADIATOR, 0x8900

init_sensor:

    # set temperature thresholds
    la   t6, SENSOR
    addi t0, zero, 21          # t0 = tmax = 21
    sw   t0, 8(t6)
    addi t1, zero, 17          # t1 = tmin = 17
    sw   t1, 12(t6)

    # set sensor interrupts & radiator
    lb   t2, 0(t6)      # t2 = temp (load byte to extend
                        # the sign)
init_sensor_check_temp_too_low:
    # if (temp >= tmin) goto init_sensor_check_temp_too_high
    bge  t2, t1, init_sensor_check_temp_too_high
    addi t3, zero, 2           # IH = 1, IL = 0
    addi t4, zero, 1           # ON = 1
    j    init_sensor_ret       # goto init_sensor_ret

init_sensor_check_temp_too_high:
    # else if (temp <= tmax) goto init_sensor_temp_ok
    bge  t0, t2, init_sensor_temp_ok
    addi t3, zero, 1           # IH = 0, IL = 1
```

```
27      addi t4, zero, 0            # ON = 0
28      j    init_sensor_ret        # goto init_sensor_ret
29
30  init_sensor_temp_ok:
31      # else if (tmin <= temp <= tmax)
32      addi t3, zero, 1            # IH = 0, IL = 1
33      addi t4, zero, 0            # ON = 0
34      j    init_sensor_ret        # goto init_sensor_ret
35
36  init_sensor_ret:
37      la   t6, SENSOR
38      sw   zero, 0(t6)      # clear IR
39      sw   t3, 4(t6)        # store IH, IL
40      la   t6, RADIATOR
41      sw   t4, 0(t6)        # store ON
42
43      jalr zero, ra, 0
```

However, we can see that the code in `init_sensor_check_temp_too_high` and `init_sensor_temp_ok` is identical. Therefore, we can factor out this code and place it before the `if` statements, as a default value, and then only check `init_sensor_check_temp_too_low`. This shorter version of the solution is shown below.

```
.equ SENSOR, 0x8800
.equ RADIATOR, 0x8900

init_sensor:

    # set temperature thresholds
    la   t6, SENSOR
    addi t0, zero, 21          # t0 = tmax = 21
    sw   t0, 8(t6)
    addi t1, zero, 17          # t1 = tmin = 17
    sw   t1, 12(t6)

    # set sensor interrupts & radiator
    lb   t2, 0(t6)      # t2 = temp (load byte to extend
                        # the sign)

    addi t3, zero, 1           # IH = 0, IL = 1
    addi t4, zero, 0           # ON = 0

init_sensor_check_temp_too_low:
    # if (temp >= tmin) goto init_sensor_ret
    bge  t2, t1, init_sensor_ret
    addi t3, zero, 2           # IH = 1, IL = 0
    addi t4, zero, 1           # ON = 1

init_sensor_ret:
    la   t6, SENSOR
    sw   zero, 0(t6)     # clear IR
    sw   t3, 4(t6)       # store IH, IL
    la   t6, RADIATOR
    sw   t4, 0(t6)       # store ON

    ret
```

b) The job of the service routine is now simply to invert the switch status and the interrupt enable bits. It is important to disable the interrupt coming from the threshold that caused it. The temperature may still be over or under the corresponding threshold and we want to avoid interrupt loops, which will cause our system to fail.

```
sensor_isr:
    addi sp, sp, -12
    sw   t0, 0(sp)
    sw   t1, 4(sp)
    sw   t2, 8(sp)

    la   t1, SENSOR
    la   t2, RADIATOR

    lw   t0, 4(t1)      # load IH and IL
    xori t0, t0, 3      # invert them
    sw   t0, 4(t1)      # store IH and IL
    lw   t0, 0(t2)      # load ON bit
    xori t0, t0, 1      # invert it
    sw   t0, 0(t2)      # store it
    sw   zero, 0(t1)    # acknowledge interrupt

    lw   t0, 0(sp)
    lw   t1, 4(sp)
    lw   t2, 8(sp)
    addi sp, sp, 12
    ret
```

[Exercise 2]

Consider a system using a RISC-V processor, with 32-bit data and address bus. The memory is byte-addressed (cf. Figure 16).

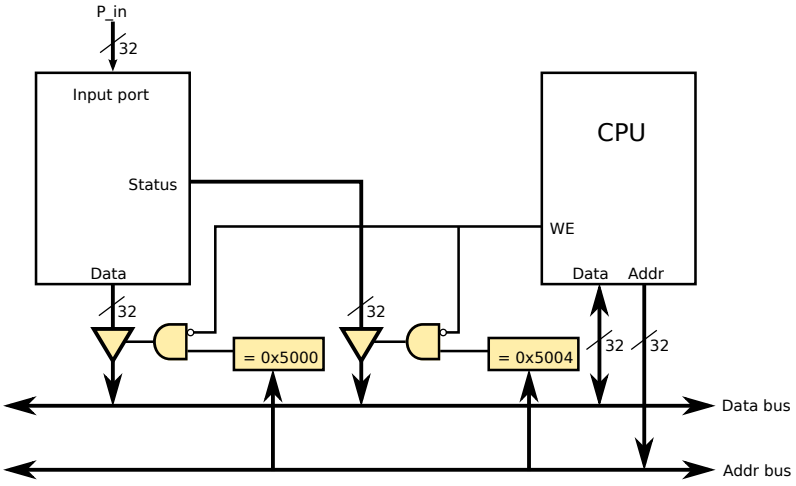


Figure 16: RISC-V system

a) You are asked to write a polling procedure `Poll()` which does the following :

- Polls bit 0 of the I/O component's status word until it is active
- Reads the `P_In` port
- Calls a `calc(a)` procedure with `P_In`'s value as argument

b) Then, write the `calc(a)` procedure which calculates $a + 100$ on signed integers and returns the result. This routine must leave all registers at their original value when the routine was called.

c) Explicit a situation where an exception could happen during the execution of this program, specifying in which circumstances it could happen.

d) Draw the state of the stack:

1. Before `Poll()` was called.
2. During `calc(a)`.
3. After `Poll()`.

[Solution 2]

a)

```
.equ IO 0x5000

poll:
    addi sp, sp, -12
    sw   t0, 0(sp)
    sw   t1, 4(sp)
    sw   ra, 8(sp)

    la   t1, IO
poll_loop:
    lw   t0, 4(t1)            # load status word
    andi t0, t0, 0x0001       # keep bit 0
    beq  t0, zero, poll_loop # if (status(0) = 0) goto poll_loop
    lw   a0, 0(t1)            # else a0 = load data word
    jal  ra, calc

    lw   t0, 0(sp)
    lw   t1, 4(sp)
    lw   ra, 8(sp)
    addi sp, sp, 12
    ret
```

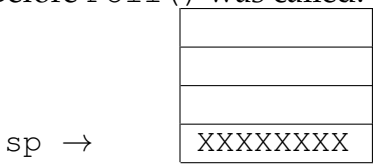
b)

```
calc:
    addi a0, a0, 100
    ret
```

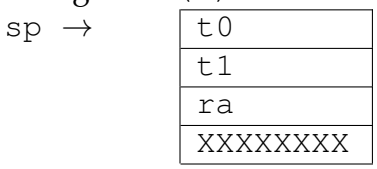
c) An arithmetic exception might occur at the **addi** instruction in `calc`, if the value is larger than `0x7FFF'FFFF - 100`.

d) We move the stack pointer in 4-byte offsets as we push and pop 32-bit words. The state of the stack is shown below:

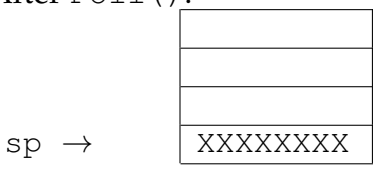
1. Before `Poll()` was called.



2. During `call(a)`.



3. After `Poll()`.



[Exercise 3]

Consider a system using a RISC-V processor, with 32-bit data and address bus. The memory is byte-addressed (cf. Figure 17).

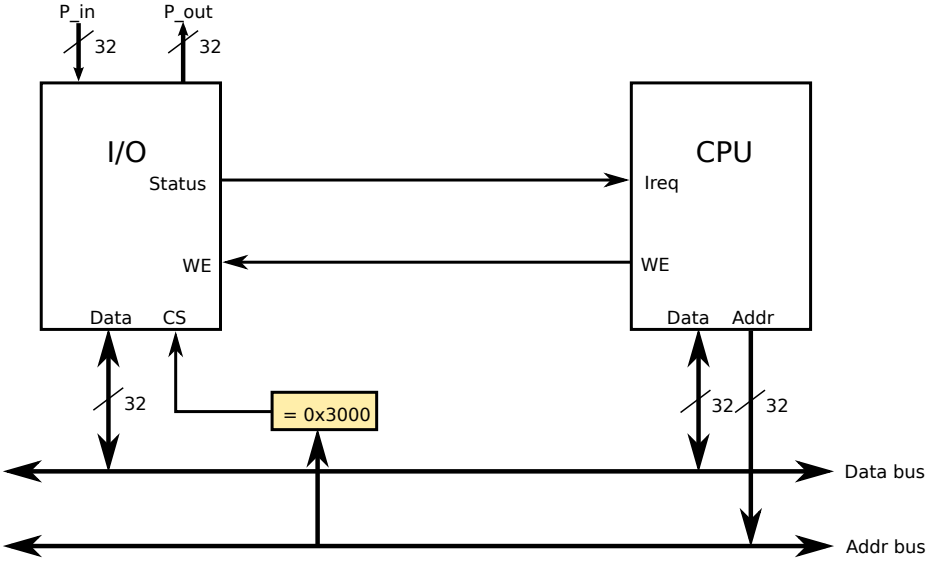


Figure 17: RISC-V system

Whenever a new value is available at the `P_in` input, an interrupt *pulse* is generated through the `Status` port of the I/O component. In this system, the I/O component is the only peripheral that generates interrupts.

a) Write a `WritePort(a)` procedure that receives a value through the `a0` register and writes it to the `P_out` port.

b) Write an interrupt service routine (ISR) for the I/O component. This routine must read `P_In` and write its value to `P_Out`. Use the `WritePort(a)` procedure you previously defined.

For this exercise, we are using a nonstandard RISC-V processor. Take the following into account:

- the `k1` register contains the return address of the ISR;
- the stack may be used in the ISR;
- all registers must have their original value when leaving the ISR;

- the processor disables all interrupts before calling the ISR. It is therefore required to reactivate them using the `rfe` instruction.

c) Draw the state of the stack:

1. At the beginning of the ISR;
2. When `WritePort()` is called;
3. At the end of the ISR.

[Solution 3]

a)

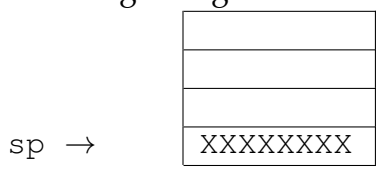
```
.equ IO 0x3000

WritePort:
    la   t0, IO
    sw   a0, 0(t0)
    ret
```

b)

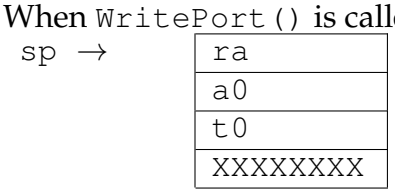
```
    addi sp, sp, -12
    sw   ra, 0(sp)
    sw   a0, 4(sp)
    sw   t0, 8(sp)

    la   t0, IO
    lw   a0, 0(t0)
    jal  ra, WritePort

    lw   ra, 0(sp)
    lw   a0, 4(sp)
    lw   t0, 8(sp)
    addi sp, sp, 12
    rfe
    jalr zero, k1, 0
```

c) The state of the stack:

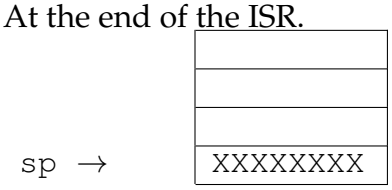
1. At the beginning of the ISR.



2. When `WritePort()` is called.



3. At the end of the ISR.



[Exercise 4]

Figure 18 describes a system using a RISC-V processor. An interrupt controller is used to handle interrupts coming from several peripherals. The controller receives the interrupts on the **irq0** to **irq2** lines.

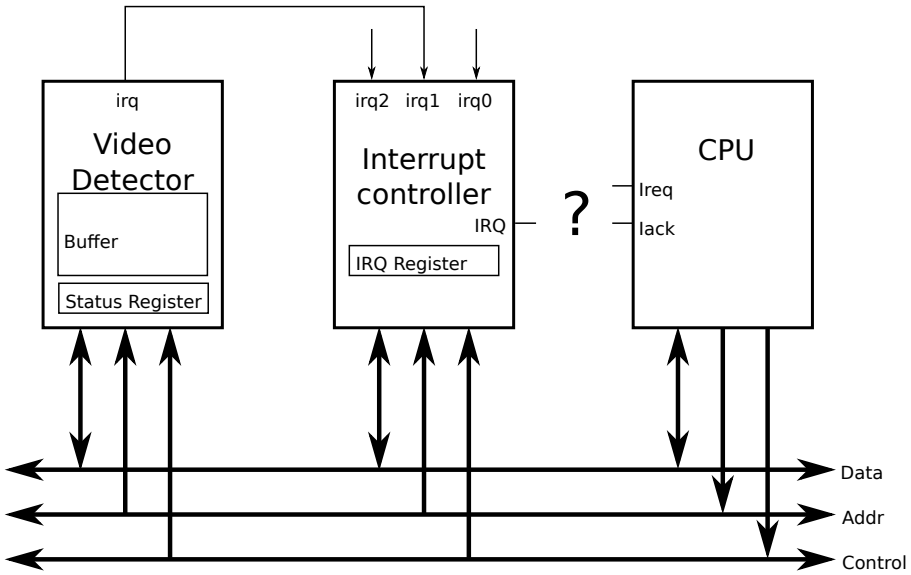


Figure 18: RISC-V system

When a peripheral generates an exception on one of the **irq0** to **irq2** wires, the controller sets the corresponding bit in the **IRQ Register** to 1. Figure 19 shows the various registers available in the system.

Figure 19: Control registers

Figure 20 illustrates an interrupt being generated and the way it is serviced.

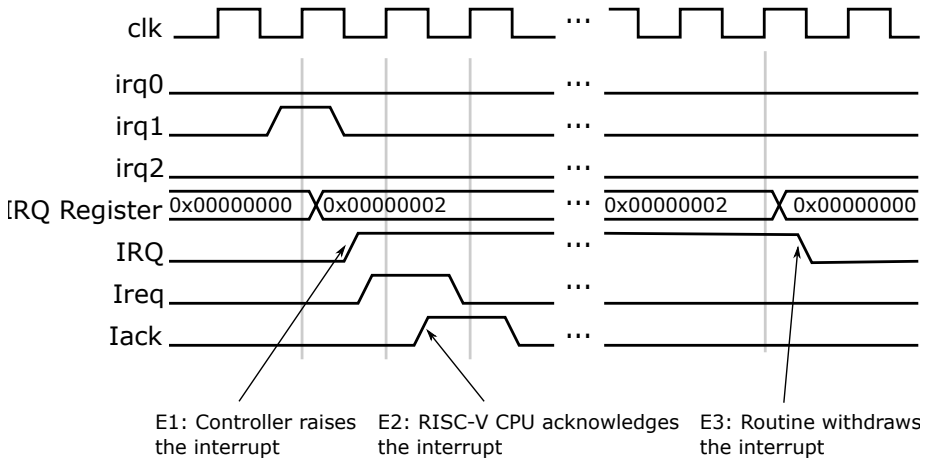


Figure 20: Interrupt timing diagram

The **IRQ** signal is combinatorial and is set if at least one of the bits from the **IRQ Register** is set. "E1" represents this in the diagram. The processor uses another protocol: **Ireq** gets asserted *immediately* after **IRQ**, and gets de-asserted *immediately* when **Iack** (coming from the processor) is asserted.

a) Complete the figure describing the system, and add the logic component(s) needed to generate **Ireq** from **IRQ** and **Iack**.

After having received the interrupt through **Ireq**, the processor generates a confirmation ("E2" represents this on the diagram) and automatically invokes the routine handling external interrupts. This routine must first decide which irq line caused the interrupt by reading the **IRQ Register**, and then call the routine specific to the peripheral. As soon as the peripheral's request is handled by its routine, the main routine confirms the interrupt by erasing the corresponding bit in the **IRQ Register** ("E3" in the diagram). The main routine enables future interrupts and returns to normal execution (code outside of ISR).

b) Write the main routine handling external interrupts, which calls the specific routines. Take the following into account:

- When more than one interrupt is waiting to be serviced (more than one bit asserted in the **IRQ Register**), the main routine must handle them all according to their relative priorities (**irq2** has the highest priority and **irq0** the lowest priority).
- The `k1` register contains the return address of the ISR.
- The processor disables all interrupts before calling the ISR. It is therefore required to reactivate them using the `rfe` instruction.

- The routine must follow the semanstic of RISC-V Assembly. Assume that the stack is available.
- For **irq2**, the routine must call `disk_int`; for **irq1**, it must call `det_int`; and for **irq0**, it must call `key_int`.
- Table 21 shows the memory layout of the system.

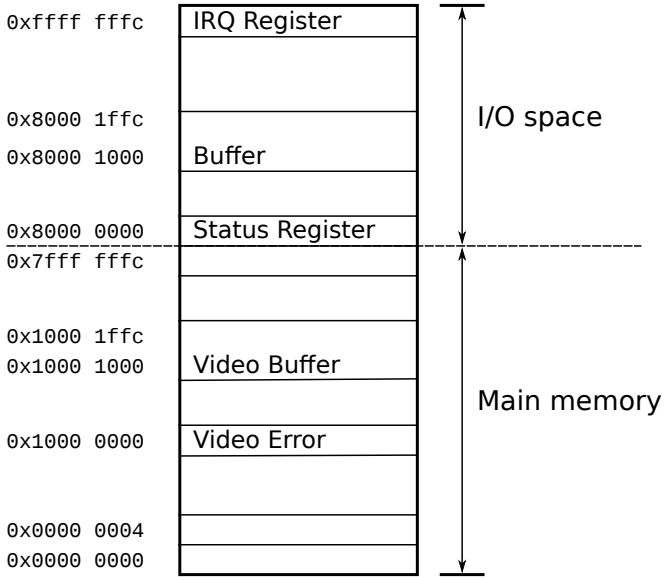


Figure 21: Memory layout

One of the peripherals is a video detector whose purpose is to detect emergencies and save some relevant data. Its controller has a **Status Register** and a **Buffer** that the processor can access. An interrupt is asserted when an emergency occurs. The **Buffer** memory contains the relevant data. To handle the interrupt, the routine must:

1. Check that the emergency detection did not suffer from any errors. An error is indicated by the most significant bit of the **Status Register**. If there is no error, proceed. Otherwise write -1 to the main memory at address **Video Error**, and return from the routine;
2. Read the *12-bit* **Count** value from the **Status Register** to determine the number of words that are ready in the **Buffer**;
3. Transmit all data received in **Buffer** to the **Video Buffer** in main memory;
4. Set the **Count** bits to 0.

c) Write the `det_int` routine which handles the video interrupts described above.

d) The copy process from the **Buffer** to the **Video Buffer** is not very efficient on this RISC-V system, explain why. Suggest a way to improve the copy, adding additional components if needed. Make a drawing of that new system and comment it briefly.

[Solution 4]

a) The following figure describes the circuit generating **IRQ** and **lack**. Pay attention to the following details:

- The statement says that "**Ireq** gets asserted *immediately* after **IRQ**", so we use **IRQ** as an edge detector to assert the register to 1. Note that we do not use **IRQ** as the SET port of the register because that would keep the register asserted as long as **IRQ** is asserted, which is not the behaviour depicted in the timing diagram.
- The statement says that "(**Ireq**) gets deasserted *immediately* when **lack** (coming from the processor) is asserted", so we use **lack** as an *asynchronous* reset for the register. Note that we don't use a *synchronous* reset because the timing diagram shows that **Ireq** gets de-asserted before **lack** is seen on the next rising edge of the clock.

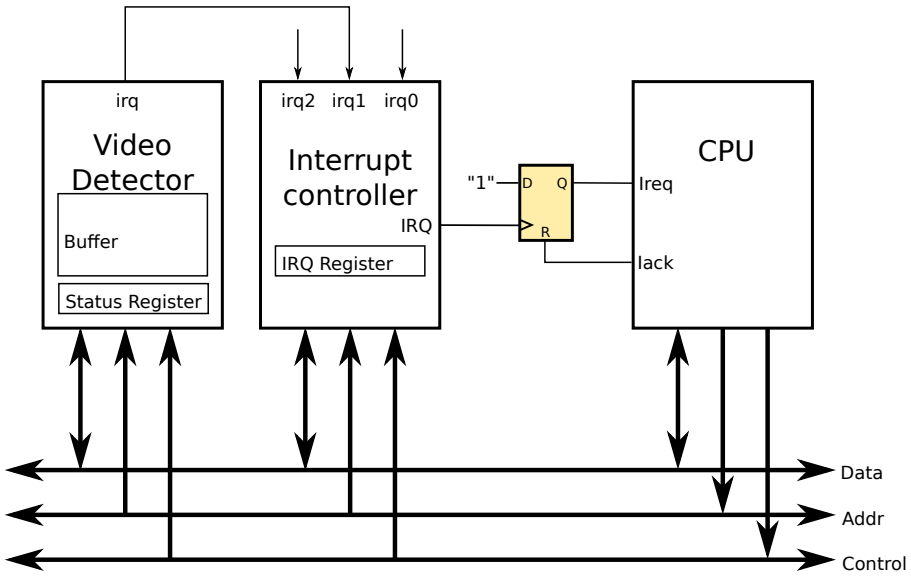


Figure 22: Solution for IRQ generating circuit

b) The following code is a possible solution for the main routine:

```
ext_ir:
    addi    sp, sp, -16      # update stack pointer
    sw      ra, 0(sp)        # save return address
    sw      s0, 4(sp)        # save s0
    sw      s1, 8(sp)        # save s1
    sw      t0, 12(sp)       # save t0
```

```
7
8       li      s0, 0xfffffffc   # address of IRQ Register in s0
9       lw      s1, 0(s0)        # s1 <- IRQ Register
10      andi    t0, s1, 0x4      # get the bit no 2 (irq2)
11      beq     t0, zero, skip2  # if t0 = 0 then goto skip2
12      jal     ra, disk_int     # call disk_int handler
13      andi    s1, s1, 0x3      # clear the bit no 2
14      sw      s1, 0(s0)        # update IRQ Register
15  skip2:
16      andi    t0, s1, 0x2      # get the bit no 1 (irq1)
17      beq     t0, zero, skip1  # if t0 = 0 then goto skip1
18      jal     ra, det_int      # call det_int handler
19      andi    s1, s1, 0x5      # clear the bit no 1
20      sw      s1, 0(s0)        # update IRQ Register
21  skip1:
22      andi    t0, s1, 0x1      # get the bit no 0 (irq0)
23      beq     t0, zero, skip0  # if t0 = 0 then goto skip0
24      jal     ra, key_int      # call key_int handler
25      andi    s1, s1, 0x6      # clear the bit no 0
26      sw      s1, 0(s0)        # update IRQ Register
27  skip0:
28      lw      ra, 0(sp)        # restore return address
29      lw      s0, 4(sp)        # restore s0
30      lw      s1, 8(sp)        # restore s1
31      lw      t0, 12(sp)       # restore t0
32      addi    sp, sp, 16       # update stack pointer
33
34      rfe                      # restore interrupt mask and
35                               # processor mode
36      jalr    zero, k1, 0      # return from handler
```

Since the routine calls specific routines, we must save the registers needed after the call. We use RISC-V convention which defines that `s0` and `s1` are kept. Return address and the registers are saved on the stack before the stack and are reloaded at the end of the routine.

This code defines the priorities for the interrupts. The way the bits are treated guarantees that **irq2** has the highest priority, followed by **irq1**, and then **irq0**.

c) The RISC-V code for the `det_int` routine is:

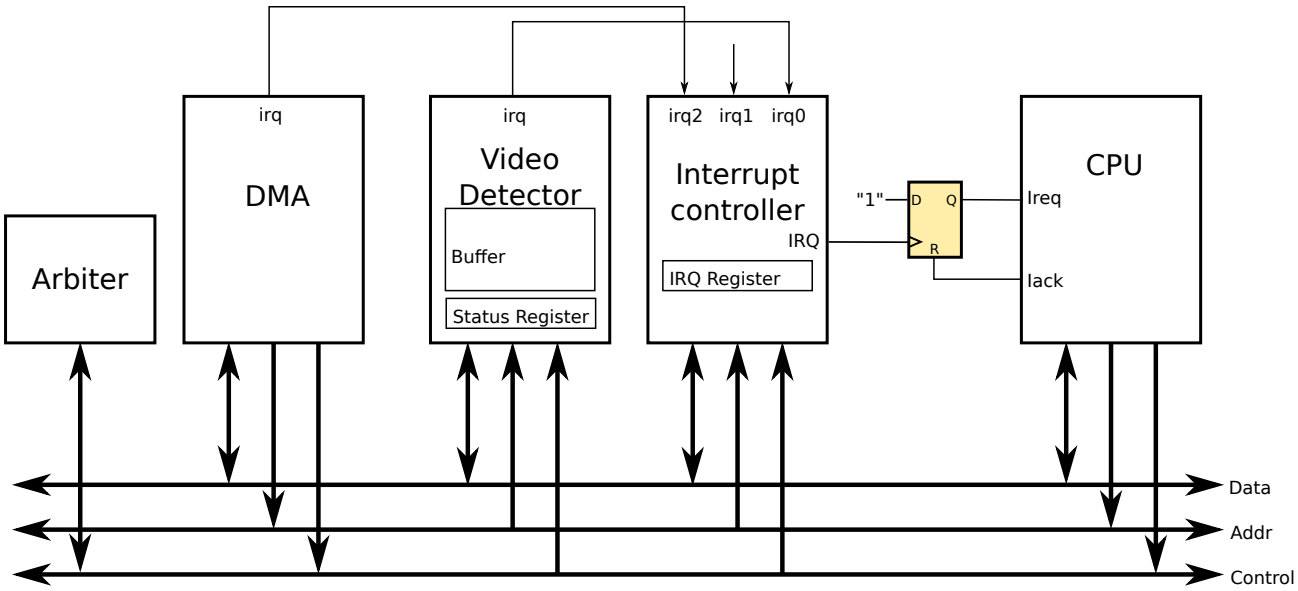
```
1   det_int:
2       addi    sp, sp -28
3       sw      t0, 0(sp)
```

```
4       sw      t1, 4(sp)
5       sw      t2, 8(sp)
6       sw      t3, 12(sp)
7       sw      t4, 16(sp)
8       sw      t5, 20(sp)
9       sw      t6, 24(sp)
10
11      li      t0, 0x80001000      # t0 <- addr(Buffer)
12      li      t1, 0x80000000      # t1 <- addr(Status Register)
13      li      t2, 0x10000000      # t2 <- addr(Video Error)
14      li      t3, 0x10001000      # t3 <- addr(Video Buffer)
15      lw      t4, 0(t1)           # t4 <- Status Register
16      srli    t5, t4, 31          # t5 <- t4 >> 31
17      beq     t5, zero, no_error  # if err = 0, goto no_error
18  error:
19      addi    t5, zero, -1        # t5 <- -1
20      sw      t5, 0(t2)           # Video Error <- t5
21      j       end                 # goto end
22  no_error:
23      li      t6, 0xfff
24      and     t4, t4, t6          # t4 <- count
25  loop:
26      beq     t4, zero, loop_end  # if t4 = 0 then goto loop_end
27      lw      t5, 0(t0)           # read from Buffer
28      sw      t5, 0(t3)           # store to Video Buffer
29      addi    t0, t0, 4           # t0 <- t0 + 4
30      addi    t3, t3, 4           # t3 <- t3 + 4
31      addi    t4, t4, -1          # t4 <- t4 - 1
32      j       loop                # goto loop
33  loop_end:
34      sw      zero, 0(t1)         # set count bits to 0 (the err bit
35                                  # is already 0 and the other bits
36                                  # in the status register are
37                                  # undefined)
38  end:
39      lw      t0, 0(sp)
40      lw      t1, 4(sp)
41      lw      t2, 8(sp)
42      lw      t3, 12(sp)
43      lw      t4, 16(sp)
44      lw      t5, 20(sp)
45      lw      t6, 24(sp)
46      addi    sp, sp 28
47
```

```
48      ret                             # return
```

d) This procedure is not efficient because the processor is used to copy large amounts of data. Furthermore, this copy is done in an interrupt service routine, in which the processor should not stay too long. The program running before the interrupt is blocked during that copy, which takes a lot of time. One way to improve the system is to add a Direct Memory Access (DMA) controller, which takes care of the memory copy operation without any processor intervention. The processor has to start the copy by telling the DMA which addresses it shall copy, but can move on to other tasks during the copy.

In this scenario, two devices may issue requests on the bus. They are called bus masters. An arbitrator, which decides which busmaster may take the bus if needed because it is necessary to avoid collisions and to guarantee a fair sharing of the bus. The following figure shows the system with the DMA and the arbiter.



[Exercise 5]

Consider a RISC-V based system that has two types of general exceptions: *hardware interrupts* and *instruction-related exceptions*; the former having a higher priority. *Hardware interrupts* are requested by external sources such as peripheral devices. *Instruction-related exceptions* could be caused either by a *software trap* or by an *unimplemented instruction*. A *software trap* is generated when a special instruction, **`trap`**, is issued. An *unimplemented instruction* exception is generated when a valid instruction (e.g., **`mul`**, **`muli`**, **`div`**, etc.) is issued but the instruction is not implemented in the hardware. In this question, you are requested to write an exception handler which determines the cause of the exception and performs the necessary operations to handle it.

The cause of the exception is determined as follows (see also the reminder at the end). If the `MPIE` bit of the `mstatus` register is 1 and the value of the `mip` register is non-zero, an external hardware interrupt caused the exception. Otherwise (i.e., if the `MPIE` bit of the `mstatus` register is 0 and/or the value of the `mip` register is 0), an instruction-related exception occurred. To distinguish between *software traps* and *unimplemented instructions*, the instruction at the address stored in `mepc` is read. If the instruction is **`trap`**, the exception is a *software trap*; else, the exception was caused by an *unimplemented instruction*.

a) In this question, you are asked to write a RISC-V assembly code for the exception handler which determines the cause of the exception and calls the corresponding routines. Assume that the following is available.

The *hardware interrupts* are properly managed by an already defined routine called `int_routine`. This routine internally loops through all requested hardware interrupts and calls the necessary routines to handle them. You should call this routine in your exception handler *only if* there is a hardware interrupt. The interrupts should be disabled before calling this routine.

There are already defined software routines for the *unimplemented instructions*, behaving exactly like the hardware counterparts; these routines have the same names as the corresponding instructions followed by an underscore (e.g., a routine called `mul_`). These routines *internally* read the instruction that caused the exception, find out which registers are used as parameters, perform the necessary operations for the arithmetic functionality and write back the results to the correct register. You should store the instruction code (32-bit value) that caused the exception, to a fixed memory location (0x4000), before calling these routines. For simplicity, assume that there are only two

unimplemented instructions that need to be handled: **mul** and **div**. Also, before calling the instruction routines (i.e., `mul_` and `div_`), you should restore `MIE` to enable the nesting of exceptions: *hardware interrupts* can interrupt *instruction-related exceptions*.

In all the given routines (`int_routine`, `mul_`, `div_`), the content of all registers (except the registers used for returning results) is preserved and the exceptions are never enabled/disabled. The *trap exception* or an unknown exception should be ignored by properly returning from the exception handler (without creating any side-effects). The instruction encoding of **mul**, **div** and **trap** are given in Figure 23.

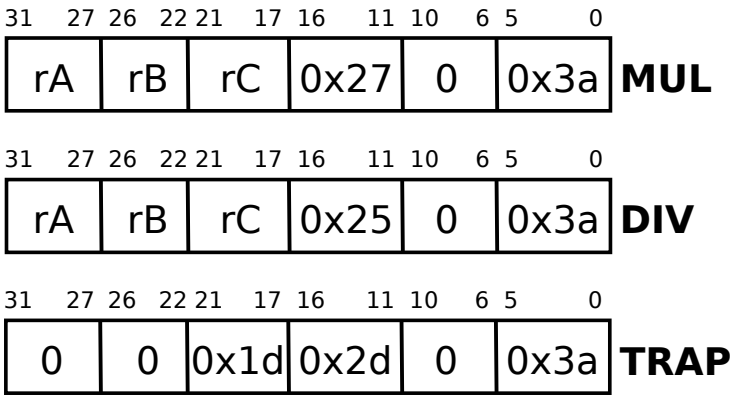


Figure 23: Instruction codes of **mul**, **div** and **trap**.

b) Why do you think the designers of this RISC-V system implemented an exception-based mechanism for multiplication and division operations?

c) Can you give an example application for the use of **trap** instruction?

Reminder

The instructions to manipulate the control registers are the following:

- **csrrw** `rd, csr, rs1`: reads the value of the control register `csr` and stores it in the register `rd`. It also writes the value of `rs1` in `csr`
- **mret**: restores MIE from MPIE and jumps to the address in `mepc`.
- **trap**: saves the address of the instruction in register `mepc`, saves the contents of the `MIE` bit in `MPIE`, disables interrupts, and transfers execution to the exception handler.

The following control registers are related to interrupts:

- `mstatus`: contains MIE, which is the processor interrupt-enable bit, in its fourth LSB. Write 1/0 to the MIE bit to enable/disable interrupts. When MIE is 0, interrupts are ignored. MIE is cleared at reset. It also contains MPIE in its eigth LSB, which holds a saved copy of MIE during exception processing.
- `mie`: this register controls the handling of external hardware interrupts. It contains MEIE and MTIE, controlling machine-level external and timer interrupts respectively. A value of 1 in bit MEIE/MTIE means that the corresponding interrupt is enabled; a value of 0 means that the corresponding interrupt is disabled.
- `mip`: the value of this register indicates which interrupts are currently pending. A value of 1 in bit MEIP/MTIP means that the corresponding interrupt is enabled. Writing a value to this register has no effect (read-only).

At power on, RISC-V executes the instruction at address 0. On an exception, register `mepc` is loaded with the address of the instruction at the time of the exception. The current instruction is never completed. Stack is assumed to be initialized at the beginning of the program and you are allowed to use it freely. **All exceptions trigger the execution of a global handler at address 4.**

[Solution 5]

a)

```
interrupt_handler:
    addi   sp, sp, -16         # set the stack
    sw     t0, 0(sp)
    sw     t1, 4(sp)
    csrrw  t0, mepc, zero      # Read and store the exception address
    sw     t0, 8(sp)
    csrrw  t1, mstatus, zero   # read mstatus
    sw     t1, 12(sp)          # and store it to stack

    beq    t1, zero, chk_inst # if zero, not a hardware interrupt
    csrrw  t1, mip, zero       # read mip
    beq    t1, zero, chk_inst # if zero, not a hardware interrupt
    jal    ra, int_routine     # call the hardware interrupt routine
    j      end_hardware        # jump to the end

chk_inst:
    lw     t1, 0(t0)           # get the content of [t0] (instruction
                               # at which the interrupt was generated)
                               # t1 = instruction
    li     t0, 0x4000
    sw     t1, 0(t0)    # store the instr at 0x4000 as required
                               # by mul_/div_ emulation procedures
    li     t0, 0xffff
    and    t1, t1, t0          # get the last 16 bits
    li     t0, 0x383a          # check whether a mul instruction
    beq    t1, t0, is_mul      # if yes, jump
    li     t0, 0x283a          # check whether a div instruction
    beq    t1, t0, is_div      # if yes, jump
    li     t0, 0x683a          # check whether a trap instruction
    beq    t1, t0, is_trap     # if yes, jump
    j      is_undefined        # else undefined exception

is_mul:
    csrrw  t1, mstatus, zero   # t1[7] = MPIE
    srli   t1, t1, 4
    andi   t1, t1, 0b1000      #
    csrrs  zero, mstatus, t1   # recover MPIE

    lw     t0, 0(sp)           # pop t0/t1, as it might be used by mul_
```

```
40      lw      t1, 4(sp)
41      call    mul_                # call the multiply routine
42
43      csrrci  zero, mstatus, 0b1000 # disable interrupts
44      j       end_mul_div         # jump to div/mul ending
45
46  is_div:
47      csrrw   t1, mstatus, zero   # t1[7] = MPIE
48      srli    t1, t1, 4
49      andi    t1, t1, 0b1000
50      csrrs   zero, mstatus, t1   # recover MPIE
51
52      lw      t0, 0(sp)           # pop t0/t1, as it might be used by mul_
53      lw      t1, 4(sp)
54      call    div_                # call the division routine
55
56      csrrci  zero, mstatus, 0b1000 # disable interrupts
57      j       end_mul_div         # jump to div/mul ending
58
59  is_trap:
60      lw      t1, 8(sp)           # increment t1 (mepc) register, because
61                                  # we don't want to re-execute the trap
62                                  # instruction.
63      addi    t1, t1, 4
64      j       end_handler         # ignore -> jump to the end
65
66  is_undefined:
67      lw      t1, 8(sp)
68      j       end_handler         # ignore -> jump to the end
69
70  end_hardware:
71      lw      t1, 8(sp)
72      j       end_handler
73
74  end_mul_div:
75      lw      t1, 8(sp)           # increment t1 (mepc) register, because
76                                  # we handled an unimplemented instruction
77                                  # exception with a software routine, so
78                                  # we want to skip the instruction to
79                                  # avoid retriggering the same exception
80      addi    t1, t1, 4
81      j       end_handler
82
83  end_handler:
```

```
84      csrrw   zero, mepc, t1      # Store updated mepc
85      lw      t1, 12(sp)          # pop mstatus from stack
86      csrrw   zero, mstatus, t1
87      lw      t0, 0(sp)
88      lw      t1, 4(sp)
89      addi    sp, sp, 16
90      mret                        # return from the exception handler
```

b) Because of resource limitations, sometimes designers do not want to include a multiply or a divide unit in the cores. Invoking an *unimplemented instruction* exception makes it possible to use the same binary executable on any RISC-V system, irrespective of the availability of the required units and provided that an appropriate exception handler is installed.

c) A **trap** instruction is typically used when the program requires servicing by the other resources, such as the operating system or a peripheral (e.g., debugger). For example, if the program needs an OS service, it raises an exception using **trap** instruction; the general exception handler of the operating system determines the reason for the trap and responds appropriately, possibly raising the level of privilege for the code being executed. A similar behaviour could be hard to achieve with an ordinary function call.

[Exercise 6]

The Programmable Interrupt Controller (PIC) is designed to prioritize and handle the hardware interrupt requests of multiple peripheral devices. It controls and enables the interaction with the processor through a single interrupt line. The Intel 8259A is the most known programmable interrupt controller, introduced with the 8086 microprocessors and still used on most x86 systems.

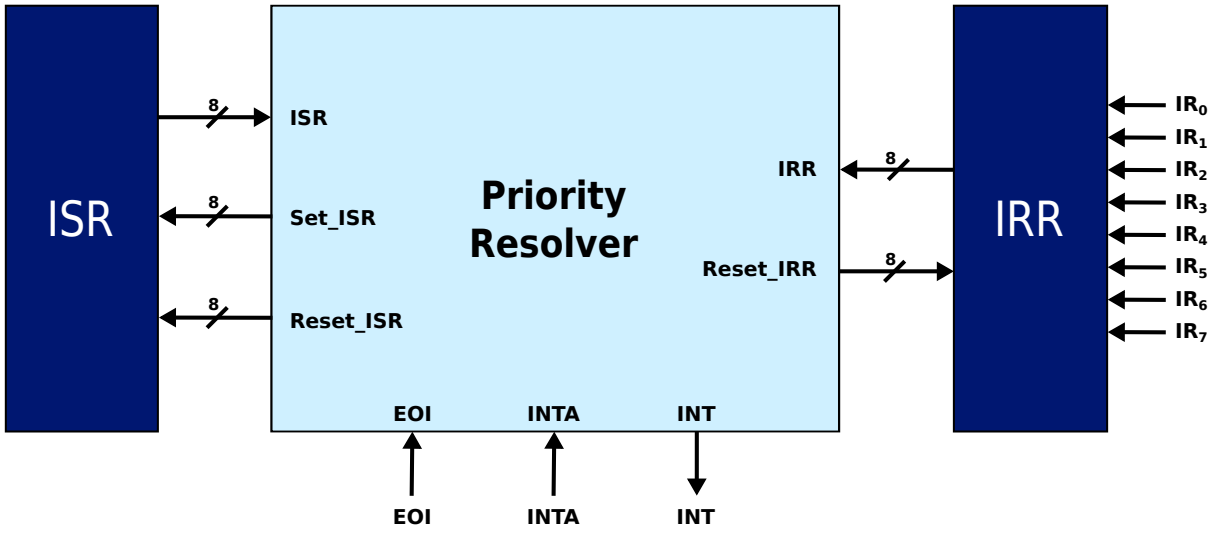


Figure 24: The interrupt controller diagram.

Consider the simplified version of the 8259A PIC shown in Figure 24, where all interrupts are enabled (all devices are allowed to issue a request). The PIC is pre-programmed and operating in a *Fully Nested mode* so that the lower indexed **IR** lines have higher priority than the higher indexed lines, i.e. $\mathbf{IR}_0$ has the highest priority while $\mathbf{IR}_7$ has the lower priority. The controller is connected to eight peripheral devices and consists of three main units: the two registers *IRR* and *ISR*, and a *Priority Resolver* unit.

The *Interrupt Request Register (IRR)* stores the interrupt requests received from the devices on the lines $\mathbf{IR}_0$ to $\mathbf{IR}_7$. The *In Service Register (ISR)* is used to keep track of the interrupts that are being serviced by the processor. The *Priority Resolver* is a fully combinatorial unit needed to decide which interrupt has the highest priority among the requesting devices. If the processor is already servicing another device, the *Priority Resolver* decides to interrupt it again only if the new request has a higher priority than the one being serviced. The *Priority Resolver* is also used to issue an interrupt signal

to the processor and interpret the interrupt acknowledgment and end-of-interrupt signals received back.

The following is a list of all the steps that occur from the moment a device sends an interrupt request to the controller until the processor finishes servicing the interrupt.

- One or more interrupt requests are received on the $\mathbf{IR}_0$ to $\mathbf{IR}_7$ lines from the eight devices connected to the controller. The *IRR* bits corresponding to the requesting devices are then set to 1.
- The priority of the requests is evaluated by the *Priority Resolver* and an interrupt request is issued to the CPU using the **INT** signal.
- The CPU acknowledges the **INT** and responds with an **INTA** pulse.
- When receiving the **INTA** pulse, the *Priority Resolver* understands that the interrupt with the highest priority is serviced and (a) sets the corresponding bit in *ISR*, (b) resets its related bit in *IRR* and (c) puts the 8-bit pointer of the interrupt subroutine on the data bus. For simplicity, all bus operations are not considered in this exercise. If an interrupt with a higher priority arrives after **INT** is sent and before **INTA** is received, the Priority Resolver will service the new interrupt (even if the CPU was not originally interrupted for that device).
- When an interrupt is being serviced, requests from lower or equal priority devices do not trigger new interrupts.
- The *ISR* bit of the currently-served interrupt remains set until the CPU issues an **End of Interrupt (EOI)** request, at the end of the interrupt subroutine execution.

a) Design the *Interrupt Request Register (IRR)* and show the details of your implementation using gate-level schematics or clear Verilog statements. Remember that the $\mathbf{IR}_0$ to $\mathbf{IR}_7$ signals are interrupt lines connected to peripheral devices.

b) Design the *Priority Resolver* unit and show the details of your implementation using gate-level schematics or clear Verilog statements.

c) Assuming a different version of the controller that generates also individual acknowledge and end-of-interrupt signals back to the peripheral devices, as shown in Figure 25, multiple PICs can be cascaded so that more than eight peripheral devices can be serviced.

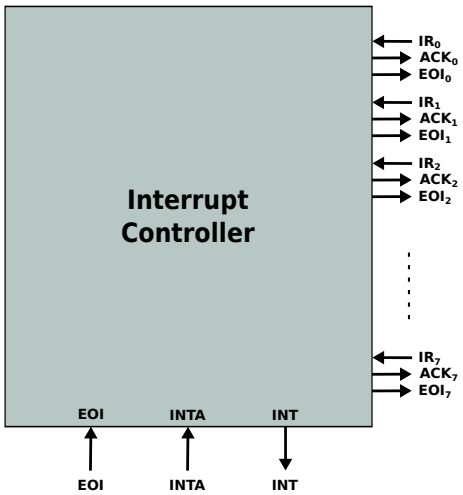


Figure 25: Modified interrupt controller interface with acknowledge signals to peripheral devices.

i) Design the peripherals' acknowledge **ACK**$_n$ and end-of-interrupt **EOI**$_n$ signals. Show the details of your implementation using gate-level schematics or clear Verilog statements.

ii) Propose ways of connecting multiple PICs to service 21 peripheral devices. Draw your design.

[Solution 6]

a) Figure 26 shows the design of the Interrupt Request Register (IRR).

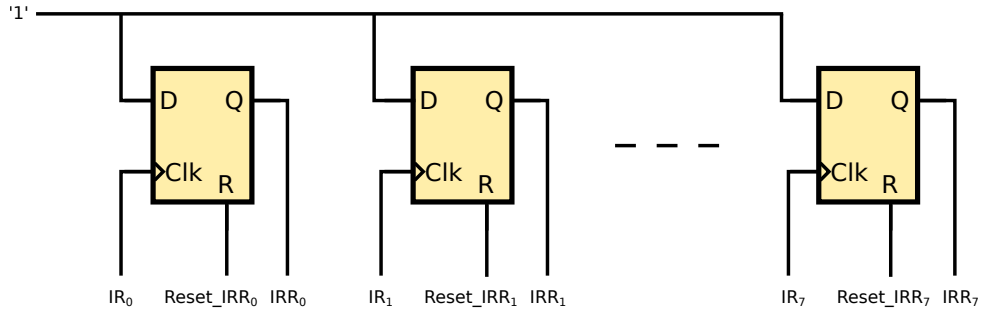


Figure 26: Design of the IRR.

b) To design the priority resolver, we first start by deciding which device has the highest priority using the priority encoder of Figure 27. Each of the bits P_0 to P_7 correspond to the one of the connected devices. However, only one of these bits is set to one at a point in time meaning that its device has the highest priority.

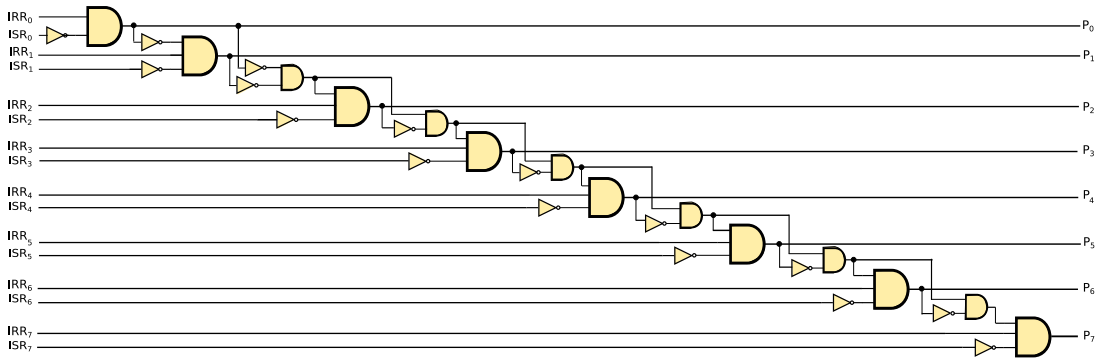


Figure 27: The priority encoder decides which device has the highest priority.

If any of the P_0 to P_7 bits is one, the INT signal is set as shown in Figure 28.

When the EOI signal is set, the ISR bit of the highest priority interrupt (the lowest indexed of the set bits) should be reset. Figure 29 shows the logic used to generate the ISR reset signals.

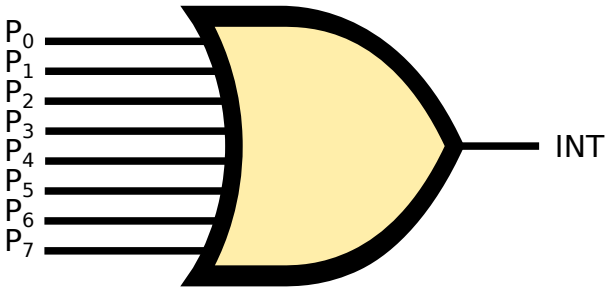


Figure 28: Generation of the INT signal.

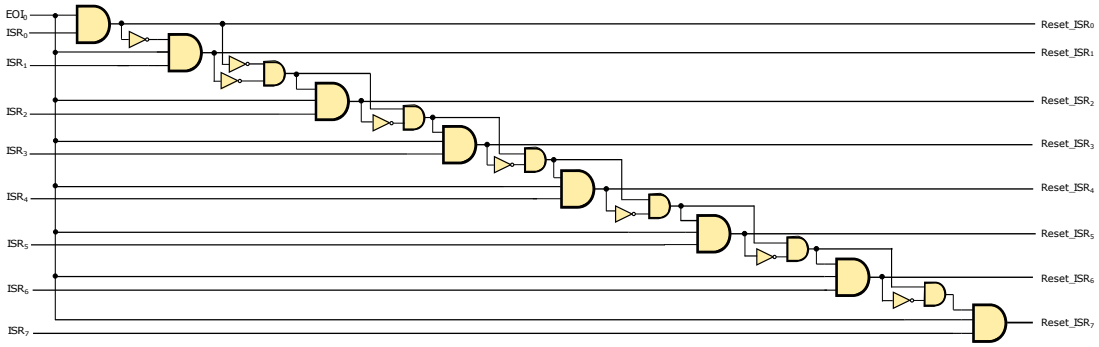


Figure 29: The logic used to generate the Reset_ISR$_n$ bits.

When an ISR bit is set, its corresponding bit must be reset in IRR; thus both signals are the same. When the INTA pulse is received, the interrupt with the highest priority must be served: its bit is ISR is set and reset in IRR.

c) In this question, the new version of the controller is used.

i) The individual End-Of-Interrupt signals (EOI_n) are the same as the Reset ISR signals ($Reset\_ISR_n$). The individual acknowledge signals (ACK_n) are the same as the Reset IRR signals ($Reset\_IRR_n$).

ii) Three Interrupt Controllers are needed to serve 21 devices. Figure 31 shows how the three PICs can be connected.

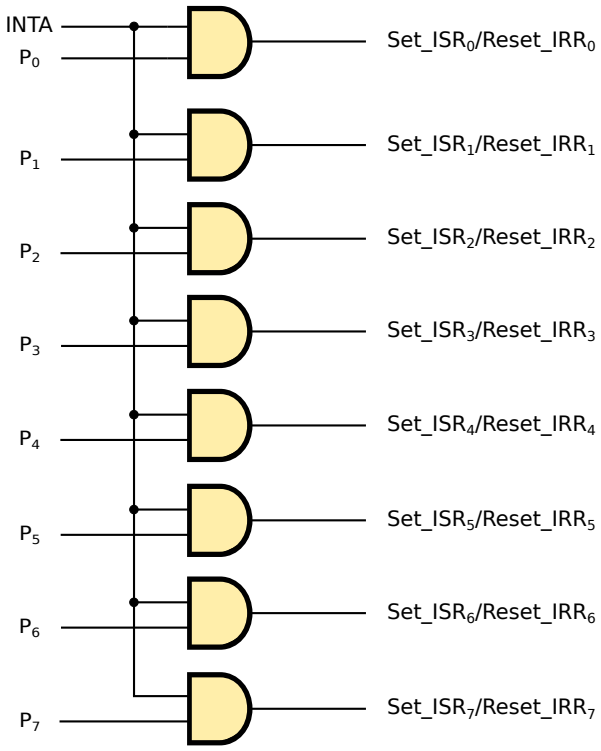


Figure 30: The logic used to generate the Reset_IRR$_n$ and Set_ISR$_n$bits.

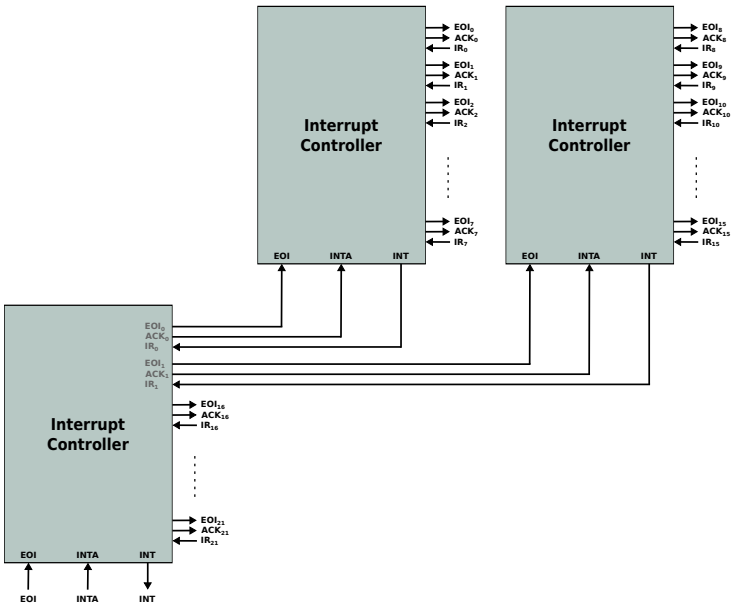


Figure 31: Three interrupt controllers can be connected to serve up to 22 devices.

[Exercise 7]

You need to design a digital stereo microphone that transmits through radio the audio signals. For simplicity and versatility, this system is based on a simple 32-bit processor with a byte-addressed memory and some peripherals: an analog-to-digital converter (ADC), a timer, a button and a transmitter.

The physical interface of the processor is shown in Figure 193, where **AS** is an address strobe signal indicating that the address on the **A** bus is valid and **WR** is a write signal indicating that the processor is writing to the bus.

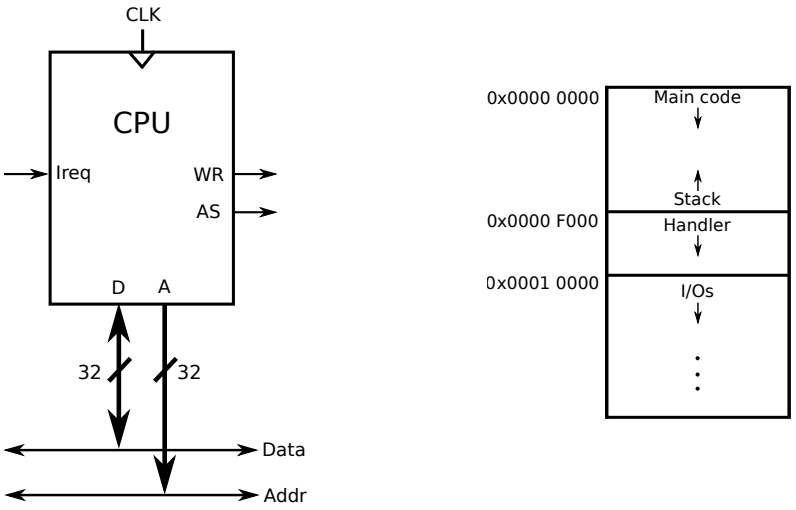


Figure 32: The physical interface of the processor and its memory.

The bus accesses are performed as shown in the timing diagram of Figure 33.

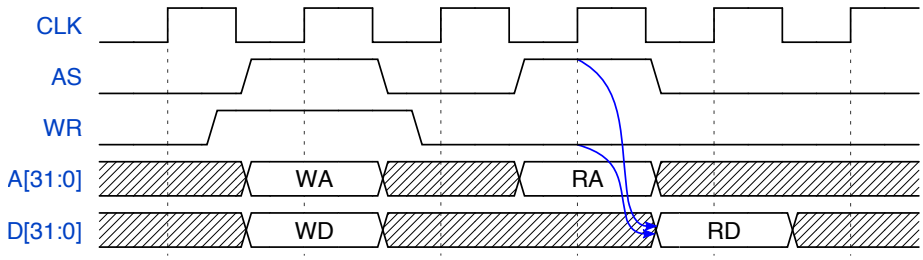


Figure 33: Timing diagram of the bus.

Main memory is composed of 2^{16} bytes (or 2^{14} words) at ranges between address 0 and address 0xFFFF. When the processor is powered up, it starts executing the code at address 0. The clock of the system has a frequency of 1 GHz.

The processor is not a RISC-V, but it executes the regular instruction set of RISC-V. It doesn't have any cache or virtual memory. In terms of interrupts, it has the following characteristics that are slightly different from those of the standard RISC-V:

- The processor has an instruction **`eint`** to enable the interrupts. The interrupts are not enabled immediately after the execution of the instruction **`eint`** but after a dozen of clock cycles, which generally allows the execution of at least one or two more instructions. When the processor is powered up, the interrupts are not enabled.
- It has a single **IREQ** signal for the interrupts. When this signal goes to 1, the processor, after a certain delay, will interrupt the current program and execute the program at address 0xF000.
- During the execution of the code at address 0xF000, the interrupts are automatically disabled and the register `mepc` contains the address of the second instruction after the last one that has been completely executed. Note that register `mepc` can be accessed with classical RISC-V instructions. For example, **`addi`** `mepc, mepc, -4` is a valid instruction (although in a regular case, one would need to use **`csrrw`** or similar instructions to access `mepc`).
- There is no hardware system to indicate that an interrupt is being served (in other words, there is no **IACK** signal). If needed, this functionality must be implemented by the user.

Note that more than one element resembles the mechanisms of the RISC-V but this remains a different interrupt protocol. There are no **`mret`** instructions and no RISC-V control registers.

The analog-to-digital converter (ADC) with its two microphones is shown in Figure 34 while its typical timing diagram is shown in Figure 35.

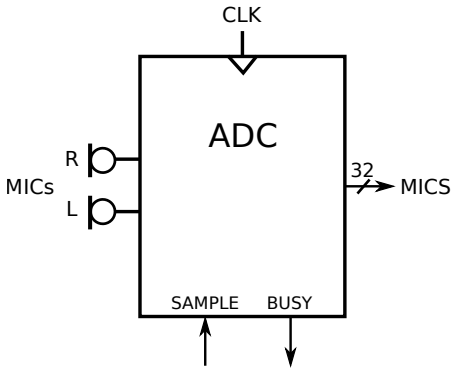


Figure 34: The analog-to-digital converter (ADC).

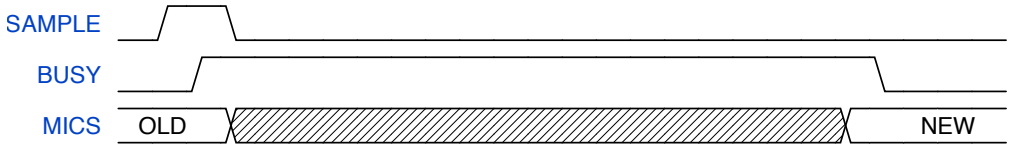


Figure 35: The timing diagram of the ADC.

To perform the conversion, a pulse (transition from 0 to 1 and then from 1 to 0) needs to be generated on the signal **SAMPLE**. Then we need to wait until the signal **BUSY** returns to 0 before reading the bus **MICS** with its two 16-bit words that represent the values of the analog signals on the *right* and *left* channels. You can assume that the ADC will never receive a pulse on **SAMPLE** when it is busy (**BUSY** equals to 1).

a) Connect the ADC to the processor so that the bit 0 of the word at address 0x10000 is the value assigned to the signal **SAMPLE**, that the bit 0 of the word at address 0x10004 is the value of the signal **BUSY**, and that the word at address 0x10008 is the value of the bus **MICS**. You may use logic gates, registers, tri-state buffers and comparators, as needed.

b) Write a function `start_conv` that starts a new conversion (no value is returned). Write a function `get_conv` that waits until a conversion is done and returns the value of the bus **MICS** at the end of that conversion.

We should sample the microphones at a 44 100 Hz and, as such, start a conversion every 22.675 μs (since the ADC has been chosen for this purpose, its conversion time is much smaller than this period).

For that, we want to use the timer of Figure 36 with its timing diagram of Figure 37.

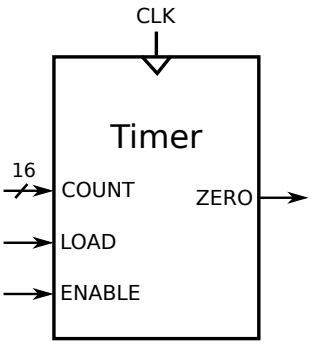


Figure 36: The timer interface.

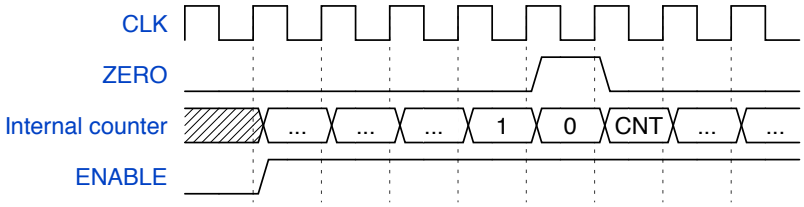


Figure 37: The timing diagram of the timer.

It is a simple counter that decrements its internal value at every clock cycle. When a signal **LOAD** is received, the value of the bus **COUNT** is saved in an internal register of the timer. When the value of the counter reaches 0, the output **ZERO** is set to 1 for one clock cycle and the value of the internal register is copied into the counter (so the output **ZERO** goes back to 0 after one clock cycle). If the signal **ENABLE** is 0, the counter is disabled.

c) Connect the timer to the processor so that the value of **COUNT** is set by writing to the address 0x1000C and the value of the signal **ENABLE** is set by writing to the bit 0 of address 0x10010. We also want to be able to know the last value that we assigned to **ENABLE** through a read of the bit 0 at address 0x10010. You may use logic gates, registers, tri-state buffers and comparators, as needed.

d) Write a function `config_timer` to configure the timer so that a pulse is generated on the output **ZERO** every 22.675 μs. Write also a function `toggle_state` to disable the timer if it is enabled and to enable it if it is disabled.

A button is also connected to the processor to start and stop the sampling. Every time

the button is pressed, a signal **BUTTON** goes from 0 to 1. Once the button is released, the signal **BUTTON** goes back to 0. Receiving a 1 on **BUTTON** stops all sampling if it has already started, and starts it if it has been stopped. When the processor is powered up, the sampling will not start until the button is pressed for the first time.

You need now to manage the interrupts received by the timer peripheral (every time the signal **ZERO** goes to 1), by the ADC (every time the signal **BUSY** goes to 0) and by the button (every time the signal **BUTTON** goes to 1).

e) Connect **ZERO**, **BUSY**, and **BUTTON** to the input **IREQ** to detect the end of a period, the end of a conversion, and to start/stop the sampling, respectively. You may use logic gates, registers, tri-state buffers and comparators, as needed. Make sure that no interrupt can be ignored by the processor (otherwise, samples can be missed and the signal can be corrupted). Implement, if needed, some logic to allow the processor to determine the source of the interrupt (out of the three). Implement also, if needed, the logic that allows the processor to signal that a particular interrupt has been received (acknowledge). Use, as needed, writes and reads to the addresses successive to 0x10010 (e.g. 0x10014 or 0x10018) while clearly specifying how these I/O ports must be used.

f) Write a function `handler` at the address 0xF000 that receives the interrupts. It should determine the cause of the interrupt (**ZERO**, **BUSY** or **BUTTON**). If the cause is **ZERO**, it should initiate a conversion. If the cause is that **BUSY** dropped to 0, it should read **MICS** and pass on the value to the transmitter by calling a function `send_radio` with the new value of **MICS** as the only parameter (you don't have to write the function `send_radio`). If the cause is **BUTTON**, it should restart or stop the sampling. Use, wherever applicable, the functions defined in the previous questions. If needed, the function `handler` must also indicate to the peripherals that their interrupts are being served. And, of course, in the end, it should re-enable all interrupts.

g) Write a new `main` program for the complete system with interrupts. Remember that the interrupts are not enabled when the system is powered up. You should use also the instruction **`halt`** to stop the execution when there is nothing to execute (it would put the processor in a low power consumption state, among others). After the execution of the instruction **`halt`**, the processor can only wake up by an interrupt; in this case, we assume that the execution of **`halt`** has completely finished.

```
1 main:
2     li sp, 0xF000
3     ...
```

[Solution 7]

a)

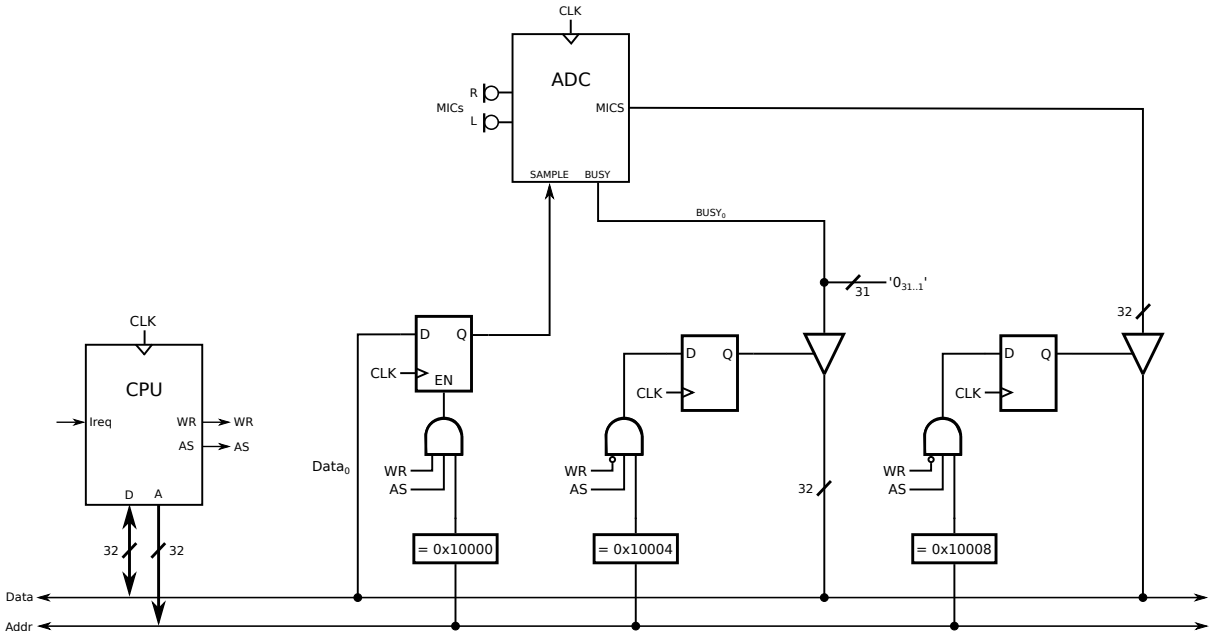


Figure 38: Connecting the ADC to the processor.

b)

```
1  start_conv:    addi   t0, zero, 1     # t0 = 1
2                 slli   t1, t0, 16      # t1 = 0x10000
3                 sw     t0, 0(t1)       # set SAMPLE to 1
4                 sw     zero, 0(t1)     # set SAMPLE to 0
5                 ret
6
7  get_conv:      addi   t0, zero, 1
8                 slli   t0, t0, 16      # t0 = 0x10000
9  loop:          lw     t1, 4(t0)       # read BUSY
10                bne    t1, zero, loop
11                lw     a0, 8(t0)       # read MICS
12                ret
```

c)

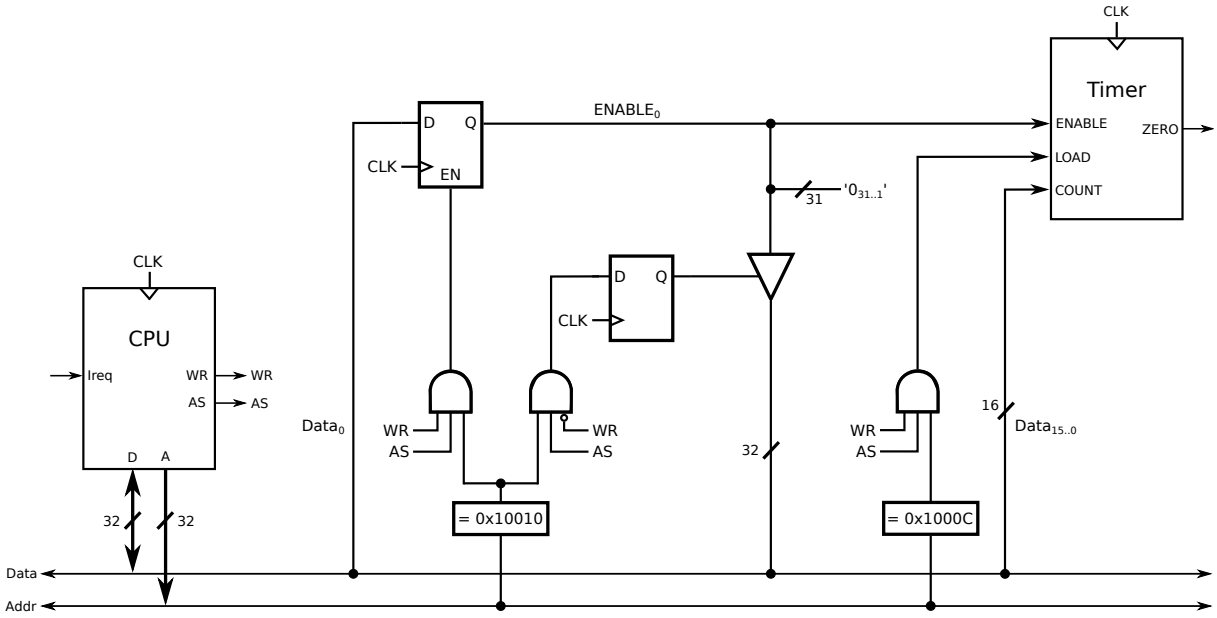


Figure 39: Connecting the timer to the processor.

d)

```
config_timer: addi   t0, zero, 1
              slli   t0, t0, 16       # t0 = 0x10000
              addi   t1, zero, 22674  # t1 = 22674
              sw     t1, 12(t0)       # write to LOAD
              ret

toggle_state: addi   t0, zero, 1
              slli   t0, t0, 16       # t0 = 0x10000
              lw     t1, 16(t0)       # read ENABLE
              xori   t1, t1, 1        # flip the last bit
              sw     t1, 16(t0)       # write to ENABLE
              ret
```

e)

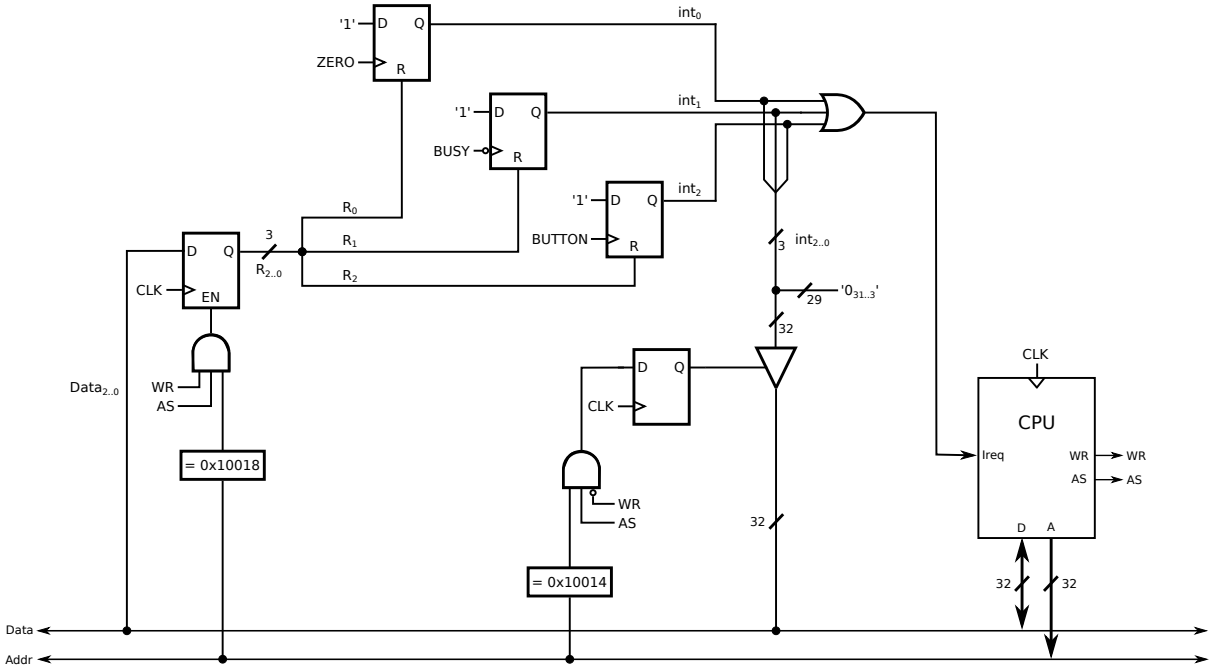


Figure 40: Logic for interrupt request and acknowledgement.

f)

```
handler:        addi    sp, sp, -20         # save registers on stack
                sw      s0,  0(sp)
                sw      t2,  4(sp)
                sw      t3,  8(sp)
                sw      t4, 12(sp)
                sw      a0, 16(sp)

                addi    s0, zero, 1
                slli    s0, s0, 16          # s0 = 0x100000
                lw      t2, 0x14(s0)        # read int[2..0]
                andi    t3, t2, 1
                bne     t3, zero, int_zero # interrupt from the timer
                andi    t3, t2, 2
                bne     t3, zero, int_busy # interrupt from the ADC
                j       int_button          # interrupt from the button

end_handler:
                lw      s0,  0(sp)
                lw      t2,  4(sp)
```

```
                lw      t3,  8(sp)
                lw      t4, 12(sp)
                lw      a0, 16(sp)
                addi    sp, sp, 20

                addi mepc, mepc, -4
                eint
                jalr zero, mepc, 0

int_zero:       addi    t4, zero, 1         # acknowledge the interrupt
                sw      t4, 0x18(s0)
                sw      zero, 0x18(s0)
                jal     ra, start_conv      # serve the interrupt
                j       end_handler

int_busy:       addi    t4, zero, 2         # acknowledge the interrupt
                sw      t4, 0x18(s0)
                sw      zero, 0x18(s0)
                lw      a0, 8(s0)           # serve the interrupt
                jal     ra, send_radio
                j       end_handler

int_button:     addi    t4, zero, 4         # acknowledge the interrupt
                sw      t4, 0x18(s0)
                sw      zero, 0x18(s0)
                jal     ra, toggle_state    # serve the interrupt
                j       end_handler
```

g)

```
main:           li      sp, 0xF000
                jal     ra, config_timer
                eint
forever:        halt
                br      forever
```

[Exercise 8]

We want to build a system based on a RISC-V processor capable of measuring the distance of an object. To this end, we decide to use an ultrasonic sensor (a device whose operating principle is based on the speed of sound). However, the speed of sound depends on the temperature, so we also need a temperature sensor in order to use the ultrasonic sensor reliably. Finally, for more precision in our measurements, we decide to include a hardware counter in the system. Figure 46 shows the processor and its 3 peripherals.

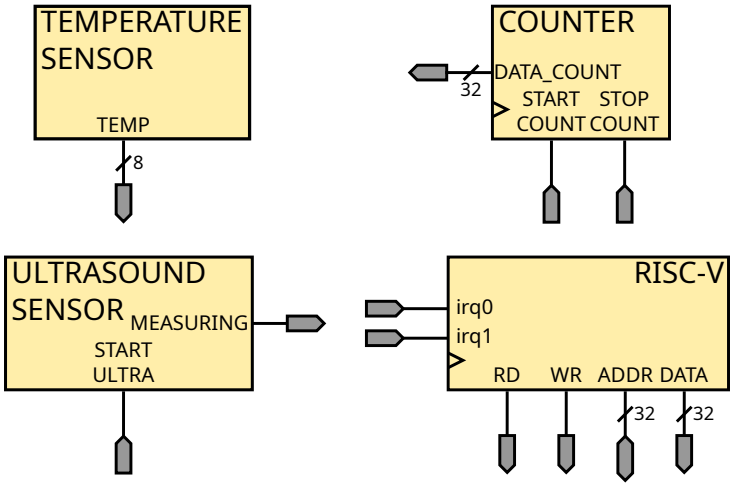


Figure 41: RISC-V processor and peripherals.

The system's operating principle is as follows:

1. An ultrasonic sensor is used to emit an ultrasonic wave. To do so, the device must receive a **pulse of a duration of at least 5 µs** on its `START_ULTRA` input pin.
2. As soon as the ultrasonic wave is emitted, the sensor's `MEASURING` output signal is raised.
3. The wave propagates at the speed of sound and bounces off the object whose distance is to be measured.
4. When the ultrasonic wave returns to the sensor, it is detected and the `MEASURING` output signal is deactivated.
5. The duration t of the pulse emitted on the `MEASURING` signal is between $0.5\,\mathrm{ms}$ and $20\,\mathrm{ms}$. It is proportional to the distance between the object and the ultrasonic sensor, and it depends on the ambient temperature T measured by the temperature sensor.

6. By measuring the duration of the MEASURING pulse and by knowing the temperature, we can then compute the real distance between the ultrasonic sensor and the object.

The following timing diagram summarizes the behaviour of the ultrasonic distance sensor:

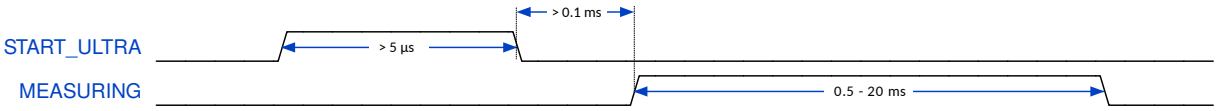


Figure 42: Ultrasonic sensor behaviour.

The hardware counter is used for **all** timing measurements in the system. The counter is clocked at 10 MHz and has an internal 32-bit register whose value is always provided on its DATA_COUNT output port. Activating the START_COUNT signal on the rising edge of the clock resets the internal register to zero and enables the counter, which means that the value of the internal register is incremented at every clock cycle. Activating the STOP_COUNT signal on the rising edge of the clock stops the counting.

a) Complete the system schematic provided at the end of the exercise in such a way to obtain the behavior described below. **It is essential that you read question 2 before drawing your solution in order to know the responsibilities of the software.**

- Writing to bit 0 of address 0x1000 controls the START_ULTRA signal.
- A **rising edge** of the MEASURING signal generates an interrupt on the irq0 pin. Writing 1 to bit 1 of address 0x1000 acknowledges the irq0 interrupt.
- A **falling edge** of the MEASURING signal generates an interrupt on the irq1 pin. Writing 1 to bit 2 of address 0x1000 acknowledges the irq1 interrupt.
- Reading from address 0x1004 returns the temperature value measured by the temperature sensor.
- Writing to bit 0 of address 0x1008 controls the START_COUNT signal.
- Writing to bit 1 of address 0x1008 controls the STOP_COUNT signal.
- Reading from address 0x100C returns the value of the internal register of the counter.

The system does not contain any other peripherals and no other interrupt signals are used.

For the following questions, use **all RISC-V conventions**.

b) Write the `ultrasound_start` **function** that starts the ultrasonic sensor by sending a pulse lasting at least **5 µs** to `START_ULTRA`. Clearly explain how you measure the time to ensure that the pulse width satisfies the duration constraint.

c) Write the `get_distance_parameters` **function**, the **interrupt handler**, and the **interrupt service routines** of `irq0` and `irq1` to obtain all the parameters needed to compute the distance between the ultrasonic sensor and the object. This exercise uses a nonstandard RISC-V core where `irq0` and `irq1` connect to bits 0 (`irq0`) and 1 (`irq1`) of `mie` and `mip`. Concretely, the standard behaviour of RISC-V is modified in the following way: (i) if bits 0 or 1 of `mie` are 0, the corresponding irq is ignored, even if MEIE is set; (ii) reading bits 0 or 1 of `mip` indicates if the corresponding irq is active (and if either is active, MEIP is set).

The general algorithm must perform the following steps:

1. Enable the interrupts from `irq0` and `irq1`.
2. Start the ultrasonic sensor by calling the `ultrasound_start` function written earlier.
3. The particular RISC-V processor that we use here has an additional instruction called **`halt`** that is not part of the standard RISC-V ISA. Use the **`halt`** instruction to put the processor in low-power mode until an interrupt is detected. Refer to the paragraph below for more details on the behaviour of the **`halt`** instruction.
4. When an interrupt from `irq0` is received, start the time measurement.
5. Use the **`halt`** instruction to put the processor in low-power mode until an interrupt is detected.
6. When an interrupt from `irq1` is received, stop the time measurement.
7. Disable the interrupts from `irq0` and `irq1`.
8. Read the time measured by the counter and convert it to ns, if necessary.
9. Read the temperature from the temperature sensor.

10. Return the measured time in register `a0` (in `ns`) and the temperature in register `a1`.

The **`halt`** instruction is functionally equivalent to the following code:

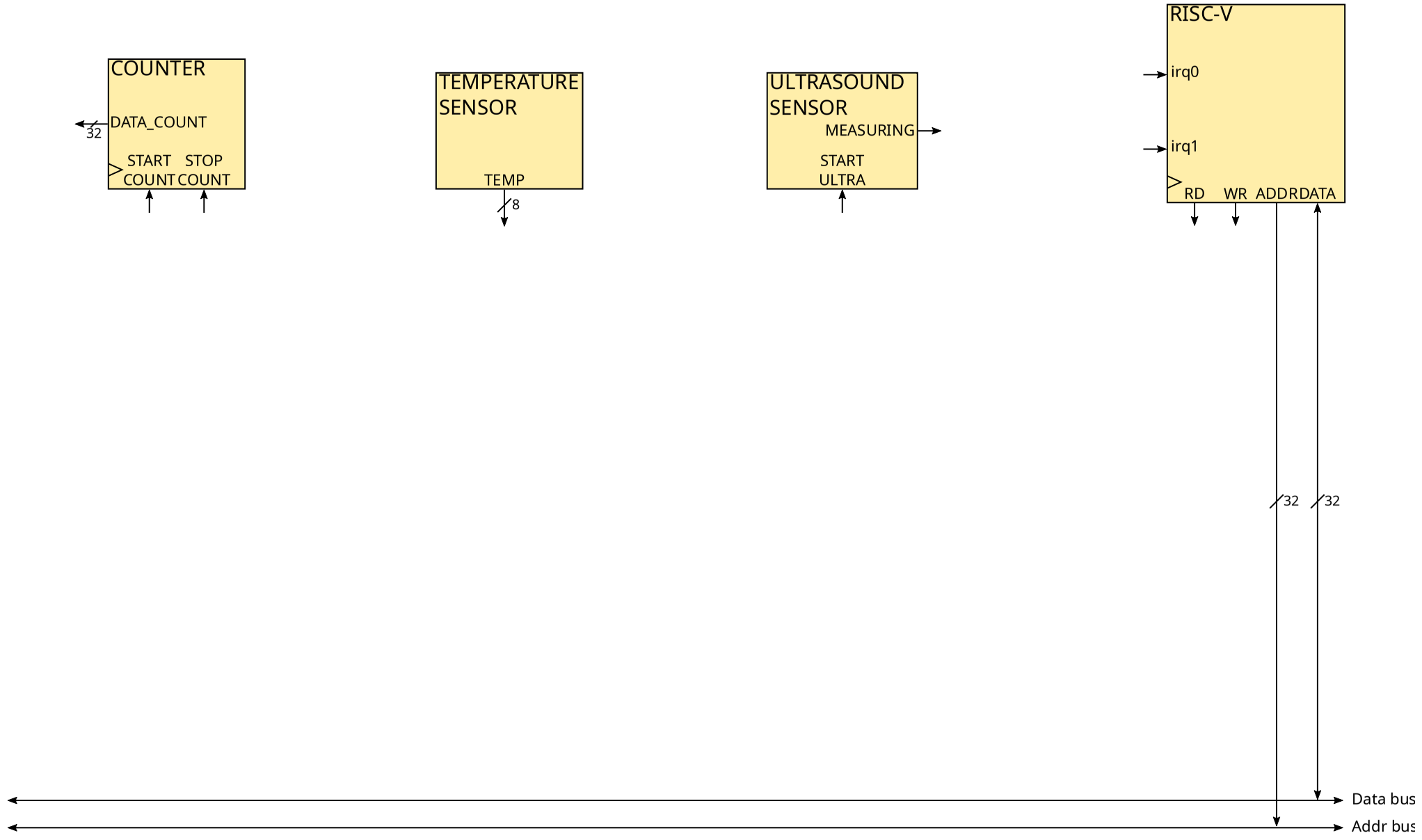
```
wait:
j wait
```

The only difference is that the code above will be executed in full-power mode, whereas the instruction **`halt`** puts the processor into low-power mode. Upon the detection of an **interrupt**, the processor leaves low-power mode and handles the interrupt as usual.

d) Explain in a clear and succinct manner (1-2 lines for each answer) the following choices you made regarding the code in point 3):

a. Did you use the value contained in the `mepc` register and, if so, why?

b. Taking into account the algorithm used in this exercise, what can you say about the accuracy of the measure of the duration of the pulse on `MEASURING`?


COUNTER
DATA_COUNT
32
START COUNT
STOP COUNT
TEMPERATURE SENSOR
TEMP
8
ULTRASOUND SENSOR
MEASURING
START ULTRA
RISC-V
irq0
irq1
RD
WR
ADDR
DATA
32
32
Data bus
Addr bus

[Solution 8]

a) The system schematic is shown in Figure 43.

b) $T_{counter}$ = 1/10 MHz = 100 ns → the counter must count until N = 5 µs/100 ns = 50.

```
ultrasound_start:
# store on stack
addi  sp, sp, -12
sw    t0, 0(sp)
sw    t1, 4(sp)
sw    t2, 8(sp)

# set START_ULTRA signal
addi  t0, zero, 1
la    t2, ULTRASOUND_SENSOR
sw    t0, 0(t2)

# 5 us = 50 clock cycles @ 10 MHz
addi  t0, zero, 50 # t0 = 5 us in clock cycles

# start counter
addi  t1, zero, 1               # t1 = counter start mask
la    t2, COUNTER_CONTROL
sw    t1, 0(t2)

ultrasound_start_poll_counter:
# poll counter until value >= 5 us
la    t2, COUNTER_DATA
lw    t1, 0(t2)                 # t1 = counter value
blt   t1, t0, ultrasound_start_poll_counter # if counter value < 5 us
# --> goto ultrasound_start_poll_counter
# stop counter
addi  t1, zero, 2               # t1 = counter stop mask
la    t2, COUNTER_CONTROL
sw    t1, 0(t2)

ultrasound_start_end:
# clear START_ULTRA signal
la    t2, ULTRASOUND_SENSOR
sw    zero, 0(t2)

# restore from stack
```

```
lw    t0, 0(sp)
lw    t1, 4(sp)
lw    t2, 8(sp)
addi  sp, sp, 12
ret
```

c)

```
get_distance_parameters:
# store on stack
addi  sp, sp, -12
sw    t0, 0(sp)
sw    t1, 4(sp)
sw    t2, 8(sp)

# enable irq0 + irq1 + MIE
li t0, 0x803
csrrs zero, mie, t0

# enable interrupts
csrrsi zero, mstatus, 8

# start the ultrasound sensor
addi  sp, sp, -4
sw    ra, 0(sp)
jal   ra, ultrasound_start
lw    ra, 0(sp)
addi  sp, sp, 4

# Halt the CPU and wait for the MEASURING signal to go high,
# generating an interrupt. The ISR of irq0 will run and when we return,
# we will return after this halt by
# construction of the interrupt handler.
halt

# Halt the CPU and wait for the MEASURING signal to go low, generating an
# interrupt. The ISR of irq1 will run and when we return, we will return
# AFTER this halt by construction of the interrupt handler.
halt

# Return value a0 = length in ns of of MEASURING pulse
# read from counter.
# Must convert from clock cycles to ns:
#
```

```
#   <=> 1 000 000 000 ns --> 10 000 000 clock cycles
#   <=>             x ns -->         a0 clock cycles
#
#   ==> x ns = a0 * 1000000000 / 10000000 = a0 * 100
# duration of MEASURING pulse in clk cycles.
lw    a0, COUNTER_DATA(zero)
slli  t0, a0, 6               # t0 = counter * 64
slli  t1, a0, 5               # t1 = counter * 32
slli  t2, a0, 2               # t2 = counter * 4
add   a0, t0, t1              # a0 = t0 + t1 = counter * 96
add   a0, a0, t2              # a0 = a0 + t2 = counter * 100

# Return value a1 = temperature read from temperature sensor.
lw    a1, TEMPERATURE_SENSOR(zero)

# disable irq0 + irq1
csrrci zero, mie, 3

# disable interrupts
csrrci zero, mstatus, 8

# restore from stack
lw    t0, 0(sp)
lw    t1, 4(sp)
lw    t2, 8(sp)
addi  sp, sp, 12
ret

ihandler:
addi  sp, sp, -8 # store on stack
sw    t0, 0(sp)
sw    t1, 4(sp)

csrrw t0, mip, zero                       # t0 = mip
andi  t0, t0, 2                           # t0 = mip AND irq1_mask
bne   t0, zero, ihandler_stop_measurement # if mip AND irq1_mask
# --> goto ihandler_stop_measurement

ihandler_start_measurement:
# acknowledge irq0 interrupt by resetting irq0 register.
addi  t0, zero, 2
la    t1, ULTRASOUND_SENSOR
sw    t0, 0(t1)
# stop resetting irq0 register.
```

```
81  sw    zero, 0(t1)
82
83  # reset and start counter.
84  addi  t0, zero , 1
85  la    t1, COUNTER_CONTROL
86  sw    t0, 0(t1)
87
88  # goto end of interrupt handler.
89  j     ihandler_return
90
91  ihandler_stop_measurement:
92  # acknowledge irq1 interrupt by resetting irq1 register.
93  addi  t0, zero, 4
94  la    t1, ULTRASOUND_SENSOR
95  sw    t0, 0(t1)
96  # stop resetting irq1 register.
97  sw    zero, 0(t1)
98
99  # stop counter
100 addi  t0, zero, 2
101 la    t1, COUNTER_CONTROL
102 sw    t0, 0(t1)
103
104 ihandler_return:
105 csrrw t0, mepc, zero
106 addi  t0, t0, 4
107 csrrw zero, mepc, t0
108 lw    t0, 0(sp) # restore from stack
109 lw    t1, 4(sp)
110 addi  sp, sp, 8
111
112 # note that we do increment mepc by 4, because we want to avoid re-
113 # executing the halt instructions we were sleeping on.
114 # If we had not incremented
115 # mepc here, we would re-execute the halt instructions
116 # and the CPU would never
117 # advance in the main program.
118 mret
```

d)

1. Normally, before the `mret` which terminates an ISR, one doesn't modify `mepc` in order to re-execute the instruction that was interrupted by the interrupt. How-

ever, in this case, interrupts can only happen while the CPU is in sleep mode after executing `halt` and, after the interrupt, we want the processor to continue with the execution of the program and not to return to sleep mode by re-executing `halt`. Therefore, `mepc` is incremented by 4 before returning from the interrupt.

2. Two factors limit the accuracy of the time measurement performed by this system:

   (a) The period of the counter, 100 ns.

   (b) The interrupt latency, i.e. the time between the falling edge of the `MEASURING` signal and the activation of the `STOP_COUNT` signal. If accuracy is the most important requirement, and there are no constraints in terms of power consumption nor the processor has to perform other tasks during the measurement, then polling the `MEASURING` signal would be a better solution.

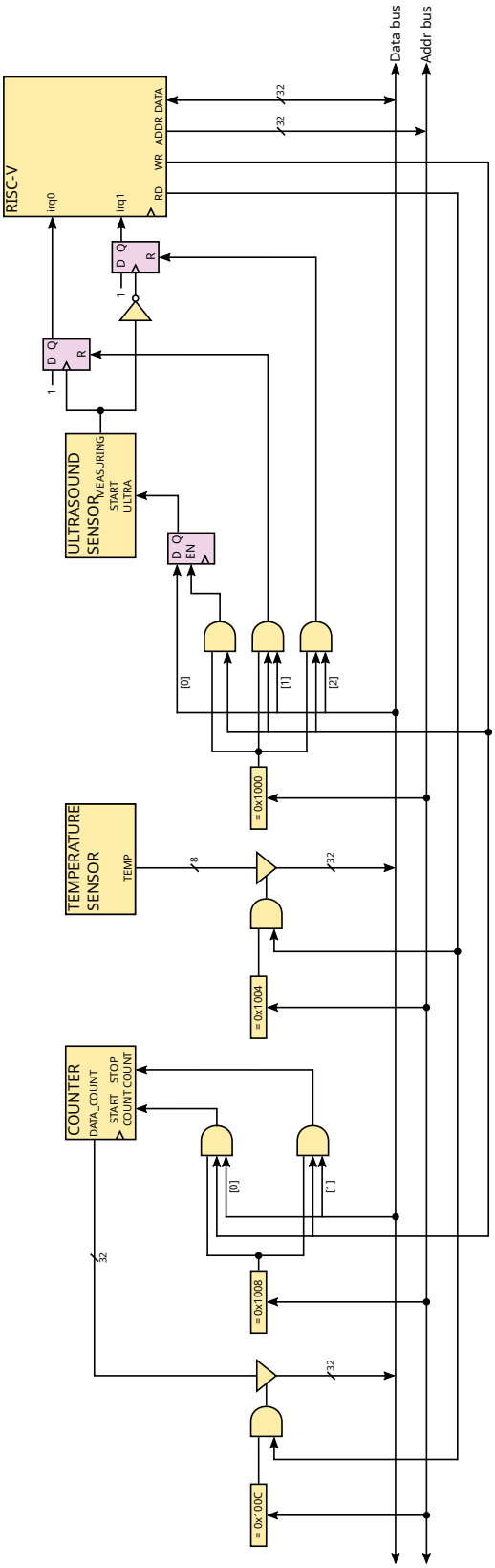


Figure 43: System schematic.

[Exercise 9] Interrupt Management

This exercise uses a nonstandard RISC-V core that has 32 distinct interrupt sources labeled `irq0` to `irq31`, each mapped to bits 0 through 31 of the `mie` (Machine Interrupt Enable) and `mip` (Machine Interrupt Pending) registers. Unlike the standard RISC-V architecture, this core does not use the MEIE and MTIE bits of `mie` for external and timer interrupts, nor the MEIP and MTIP bits of `mip` for their pending states. Instead, if bit 0 of the `mie` register is set to 0, interrupts from irq0 are ignored. The same applies to each irq source: if the corresponding bit in the `mie` register is 0, interrupts from that source are ignored, and the `mip` register's corresponding bit indicates whether an interrupt from that source is active. Thus, interrupt management is handled directly through these specific bits in the `mie` and `mip` registers.

Concretely, the standard behaviour of RISC-V is modified in the following way: (i) bits MEIE and MTIE of `mie`, along with bits MEIP and MTIP no longer exist, (ii) if bit 0 of `mie` is 0, interrupts triggered by `irq0` are ignored, (iii) reading bit 0 of `mip` indicates if `irq0` is active. The same applies to `irq1` to `irq31` that are connected to bits 1 to 31.

To increase the control we have on this non-conventional RISC-V system, we would like to define a new way for managing interrupt priorities.
To generalize interrupt handling, assume that an interrupt vector table is stored at address `0x100`. The interrupt vector table contains the address of the 32 event handlers in consecutive order—for example, the address of `irq7` is located at address `0x100+7×4`. Event handlers are standard RISC-V functions which follow the usual calling conventions.

In all the following questions, use RISC-V assembly, registers, and calling conventions.

a) Write the interrupt handler `ihandler` which looks for the source of the interrupt and calls the corresponding event handler in the interrupt vector table. Interrupts with lower index should have higher priority than those with higher index (for example, `irq2` has priority over `irq6`, which in turn has priority over `irq8`). For simplicity, there is no need to support nested interrupts.

Consider now the system shown in Figure 215 which is based on a nonstandard RISC-V processor and contains the following peripherals: a LED display, a button, and a timer. The system operates at a frequency of **100 MHz**. Any memory address that is not already used for something can be assumed to be available as RAM.

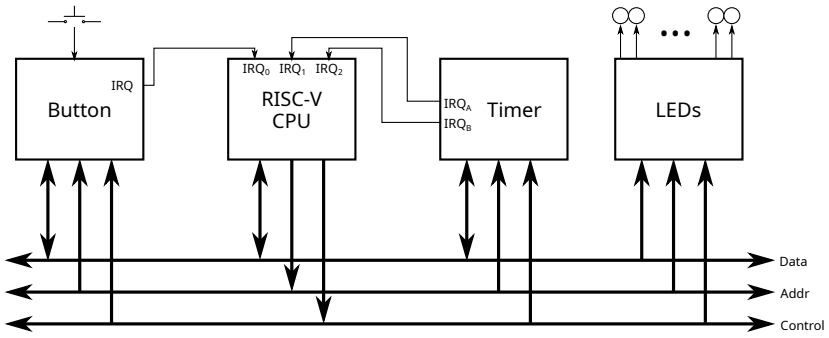


Figure 44: Block diagram of the system.

The LED display consists of a single row of 32 LEDs. The LEDs can be turned on or off by writing to the control register located at address `0x3000`. Bit 31 of the control register corresponds to the leftmost LED, bit 0 to the rightmost; writing 1 (0) turns on (off) the respective LED. The control register is **write-only**: reading from `0x3000` always returns zero.

When the button is pressed, it generates an interrupt on `irq0`. Writing any value to `0x3010` acknowledges the interrupt.

The dual timer contains two independent and identical counters, A and B. Its register map is shown below.

Name	Address	31 . . . 8	7	6	5	4	3	2	1	0
csr	0x3020	*Reserved*	contB	intEnB	zeroB	runB	contA	intEnA	zeroA	runA
prdA	0x3024	Counter A timeout period								
cntA	0x3028	Counter A current value								
prdB	0x302C	Counter B timeout period								
cntB	0x3030	Counter B current value								

In the following description, X refers to either A or B.

1. All registers are read/write.
2. Writing to the *Reserved* bits has no effect; reading from them always return 0.
3. Writing 1 to `runX` starts counter X, writing 0 stops it. Reading `runX` returns whether counter X is running (1) or not (0).
4. `cntX` contains the current value of counter X; it is decremented by one **every 10,000 cycles** while the counter is running.

5. `cntX` is initialized to `prdX` on (1) a write operation to `prdX`, or (2) when `cntX` reaches zero.
6. When `cntX` reaches zero, `zeroX` is set to 1 and, if `intEnX` is set to 1, an interrupt is generated; timer A is connected to `irq1` and timer B to `irq2`.
7. Writing 0 to `zeroX` clears it and acknowledges the interrupt.
8. After reaching zero, `cntX` immediately restarts decrementing from `prdX` if `contX` is set to 1; otherwise, it reloads `prdX` and then stops until writing 1 to `runX`.

The system's operating principle is as follows:

1. Timer A, connected to `irq1`, is used to generate an interrupt every 1 s.
2. At any given moment, there is always exactly one LED that is turned on. We will call this the *active* LED.
3. The leftmost LED is the first one to be active.
4. On an interrupt from the 1 s timer, the system changes state: the LED currently on is turned off, while one of its neighbors turns on. Initially, it is the right neighbor that turns on, which means that the active LED will move left-to-right.
5. When the active LED reaches one of the edges, the LED on the opposite edge becomes active. For example, if the current update direction is left-to-right, after the rightmost LED has been active for 1 s, the next active LED will be the leftmost LED.
6. Pushing the button changes the update direction. Note that this does not affect the timing of the next update, which will still happen 1 s after the previous one.

b) Write a `main` procedure that:

1. Initializes any internal variables. Assume the stack pointer is already initialized.
2. Initializes the interrupt vector table with two event handlers, `button_func` for `irq0` and `led_timer_func` for `irq1`. Function labels such as `button_func` are treated as constants by the RISC-V assembler and can be used wherever an immediate value is expected.
3. Configures timer A to send an IRQ every 1 s.
4. Enables the necessary interrupts.

5. Enters an infinite loop that does nothing.

In all the following questions, remember that event handlers must be written as standard RISC-V functions.

c) Write the event handler for the timer `led_timer_func` to modify the state of the LEDs whenever the timer generates an interrupt.

d) Write the event handler for the button `button_func` which changes the direction in which the LEDs are lit whenever the button generates an interrupt.

Whenever a button is pressed, its electrical contacts may experience some mechanical bouncing, as shown in Figure 45. These spurious transitions, very fast compared to human reaction times but slow compared to the cycles of a processor, may be read as multiple button pushes in a very short time, causing incorrect behavior—in the case of our system, it can cause the direction in which the LEDs are lit to change multiple times due to a single push of a button. A possible solution to this problem is **software debouncing**—ignoring the button for a certain time after detecting the first transition until it can be safely assumed that any possible bouncing is over.

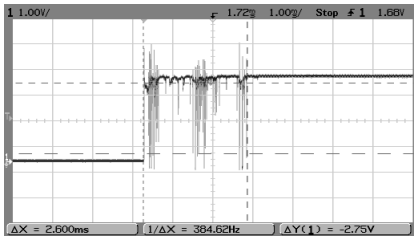


Figure 45: Snapshot of switch bounce on an oscilloscope. The switch bounces between on and off several times before settling (reproduced from `https://en.wikipedia.org/wiki/Switch`).

e) Write a new `button_func` which implements a purely software debouncing. Whenever the button causes an IRQ, wait for 10 ms before changing direction and returning from the event handler (you can safely delay servicing the timer interrupts while waiting). Measure the necessary time in **software**. Assume that each instruction takes **4 cycles** to execute.

f) Rewrite `button_func`, `debounce_timer_func`, and `main` to implement the debounce time measurement in **hardware**. Use interrupts from **timer B** to measure

the time interval of 10 ms. You should not need to modify `led_timer_func` and `ihandler`.

g) Compare in a clear and succinct manner (2-3 sentences) the software and hardware solution for time measurement from the previous two questions. Is there any benefit of using one approach over the other? If yes, provide an example situation in which one of the solutions would be beneficial. If no, justify your answer concisely.

[Solution 9]

```
# ihandler

ihandler:
addi sp, sp, -44
sw  ra, 0(sp)
sw  s0, 4(sp)
sw  s1, 8(sp)
sw  s2, 12(sp)
sw  t0, 16(sp)
sw  t1, 20(sp)
sw  t2, 24(sp)
sw  t3, 28(sp)
sw  t4, 32(sp)
sw  t5, 36(sp)
sw  t6, 40(sp)
csrrw s0, mip, zero
addi s2, zero, 0x100

loop:
andi s1, s0, 1
beq s1, zero, skip
lw  s1, 0(s2)
jalr ra, s1, 0
skip:
srli s0, s0, 1
addi s2, s2, 4
bne s0, zero, loop

sw  ra, 0(sp)
sw  s0, 4(sp)
sw  s1, 8(sp)
sw  s2, 12(sp)
sw  t0, 16(sp)
sw  t1, 20(sp)
sw  t2, 24(sp)
sw  t3, 28(sp)
sw  t4, 32(sp)
sw  t5, 36(sp)
sw  t6, 40(sp)
addi sp, sp, 44

```

```
42 mret
43
44 # main
45
46 .equ LEDS_COPY 0x2000
47 .equ DIR 0x2004
48 .equ LEDS 0x3000
49 .equ BUTTON 0x3010
50 .equ csr 0x3020
51 .equ prdA 0x3024
52 .equ prdB 0x302C
53 .equ LEFT 0
54 .equ RIGHT 1
55
56 main:
57 addi t0, zero, 1
58 srli t0, t0, 31
59
60 la  t1, LEDS_COPY
61 sw  t0, 0(t1)
62 la  t1, LEDS
63 sw  t0, 0(t1)
64
65 addi t0, zero, RIGHT
66 la  t1, DIR
67 sw  t0, 0(t1)
68
69 addi t0, zero, button_func
70 sw  t0, 0x100(zero)
71 addi t0, zero, led_timer_func
72 sw  t0, 0x104(zero)
73
74 li  t0, 10000
75 la  t1, prdA
76 sw  t0, 0(t1)
77
78 li  t0, 0b1101
79 la  t1, csr
80 sw  t0, 0(t1)
81
82 csrrwi zero, mie, 0x03
83 csrrwi zero, mstatus, 8
84
85 nothing:
```

```
86  j  nothing
87
88  # led_timer_func
89  led_timer_func:
90  la  t0, LEDS_COPY
91  lw  t0, 0(t0)
92  la  t1, DIR
93  lw  t1, 0(t1)
94
95  beq t1, zero, left
96  right:
97  srli t0, t0, 1
98  j store
99
100 left:
101 slli t0, t0, 1
102
103 store:
104 la  t1, LEDS
105 sw  t0, 0(t1)
106 la  t1, LEDS_COPY
107 sw  t0, 0(t1)
108
109 la  t1, csr
110 lw  t0, 0(t1)
111 andi t0, t0, 0xFD
112 sw  t0, 0(t1)
113
114 ret
115
116 # button func
117 button_func:
118 la  t1, DIR
119 lw  t0, 0(t1)
120 xori t0, t0, 1
121 sw  t0, 0(t1)
122
123 la  t1, BUTTON
124 sw  zero, 0(t1)
125
126 ret
127
128
129 # button func waiting
```

```
130
131 button_func:
132 addi t0, zero, 25000
133 loop:
134 add zero, zero, zero
135 add zero, zero, zero
136 add zero, zero, zero
137 addi t0, t0, -1
138 bne t0, zero, loop
139
140 la  t1, DIR
141 lw  t0, 0(t1)
142 xori t0, t0, 1
143 sw  t0, 0(t1)
144
145 la  t1, BUTTON
146 sw  zero, 0(t1)
147
148 ret
149
150 # button func timer
151
152 main:
153 .equ LEDS_COPY 0x2000
154 .equ DIR 0x2004
155 .equ LEDS 0x3000
156 .equ BUTTON 0x3010
157 .equ csr 0x3020
158 .equ prdA 0x3024
159 .equ prdB 0x302C
160 .equ LEFT 0
161 .equ RIGHT 1
162
163 addi t0, zero, 1
164 srli t0, t0, 31
165
166
167 la  t1, LEDS_COPY
168 sw  t0, 0(t1)
169 la  t1, LEDS
170 sw  t0, 0(t1)
171
172 addi t0, zero, RIGHT
173 la  t1, DIR
```

```
174 sw  t0, 0(t1)
175
176 addi t0, zero, button_func
177 sw  t0, 0x100(zero)
178 addi t0, zero, led_timer_func
179 sw  t0, 0x104(zero)
180 addi t0, zero, debounce_timer_func
181 sw  t0, 0x108(zero)
182
183 addi t0, zero, 10000
184 la  t1, prdA
185 sw  t0, 0(t1)
186
187 addi t0, zero, 100
188 la  t1, prdB
189 sw  t0, 0(t1)
190
191 addi t0, zero, 0x4D
192 la  t1, csr
193 sw  t0, 0(t1)
194
195 csrrwi zero, mie, 0x3
196 csrrwi zero, mstatus, 8
197
198 nothing:
199 j  nothing
200
201 button_func:
202 addi t0, zero, 0x10
203 la  t2, csr
204 lw  t1, 0(t2)
205 ori t1, t1, t0
206 sw  t1, 0(t2)
207
208 csrrci zero, mie, 1
209 csrrsi zero, mie, 4
210
211 ret
212
213 debouce_timer_func:
214 la  t1, DIR
215 lw  t0, 0(t1)
216 xori t0, t0, 1
217 sw  t0, 0(t1)
```

```
218
219 addi t0, zero, 0x20
220 addi t2, zero, -1
221 xor t0, t0, t2
222 la  t2, csr
223 lw  t1, 0(t2)
224 and t1, t0, t1
225 sw  t1, 0(t2)
226
227 csrrsi zero, mie, 1
228 csrrci zero, mie, 4
229
230 ret
```

g) Here both solutions are identical in terms of performance because the processor doesn't have anything else to do. The hardware solution is preferred if we need to take into account energy, or if the CPU has other things to process during the 10ms.

[Exercise 10] Interrupt Management

We want to build a system based on a RISC-V processor that controls room lighting in a university library. The last student leaving the library often forgets to turn off the light, so to reduce energy consumption we want to design a system that will turn lights off automatically if it detects no presence in the library for some time. To this end, we decide to use a presence sensor, which will work in conjunction with the physical light button present at the library's entrance to control lighting. In addition, the system includes a countdown timer. Figure 46 shows the processor and its four peripherals.

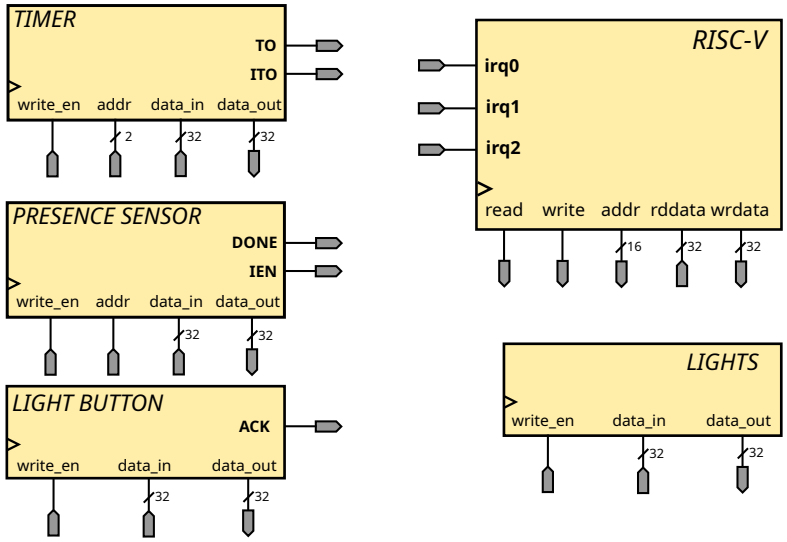


Figure 46: RISC-V processor and peripherals.

System's specifications

The system should operate as follows:

1. Whenever the button is pressed, the lights should be **flipped immediately** (i.e., turned from off to on or from on to off), regardless of the system's state.
2. Whenever the library becomes empty, the lights **shouldn't stay on for more than approximately 10 minutes**.

3. The light button is inside the library. Hence at the instant someone presses it the library is necessarily non-empty.

The presence sensor works by performing scans of the entire library on-demand (more details are provided below), using various physical metrics like noise, movement, and heat to check for presence. **The scan itself is very short-lived, taking only a few seconds, but is very energy-intensive.** We have to make sure we scan the library as rarely as possible—all while meeting the system's specification—so that the presence sensor doesn't end up consuming more energy than what we would have wasted if lights had just stayed on when no one needed them.

For simplicity, we assume the presence sensor is perfect: after a scan, it never reports a presence when the library is empty and always reports a presence when someone is in the library. We can then equate "the presence sensor reports a presence" with "someone was in the library during the scan".

In all subsequent peripherals' interface, *Reserved* bits are unused. Writing to them has no effect and reading them returns 0.

Countdown timer

The following table lists the registers of the countdown timer (chosen through the port `addr`).

Register	Address (RISC-V)	Name	31...3	1	0
0	`0x2000`	`period`	Timeout period		
1	`0x2004`	`control`	*Reserved*	**START**	**STOP/RST**
2	`0x2008`	`status`	*Reserved*		**ZERO**

`period` register

The `period` register maintains the timeout period value in number of clock cycles (see below). The internal counter is loaded with the `period` register value whenever one of the following event occurs:

- A write operation to the `period` register.
- The internal counter reaches 0.
- By writing 1 to the **STOP/RST** bit.

Writing to the `period` register stops the internal counter. The counter starts from the period's value, so **if the period is N the timer will tick after $N + 1$ clock cycles**.

`control` register

- By setting the **START** bit, the counter starts counting. The **START** bit is a *write-only* bit (i.e., when reading the `counter` register, this bit will always be read as 0). If the timer is stopped, writing 1 to the **START** bit causes the timer to start counting. When the internal counter reaches 0, it is loaded with the `period` register value and stopped. If the timer is already running, writing 1 in **START** has no effect. Writing 0 in the **START** bit has no effect.
- By setting the **STOP/RST** bit, the counter stops counting and resets its internal counter value to the `period`. The **STOP/RST** bit is a *write-only* bit. If the timer is already stopped, writing 1 to **STOP/RST** has no effect. Writing 0 to the **STOP/RST** bit has no effect.

`status` register

The **ZERO** bit is set to 1 when the counter reaches zero. The **ZERO** bit stays 1 until it is explicitly cleared by software. Write 0 to the `status` register to clear the **ZERO** bit. Writing 1 to the **ZERO** bit has no effect. The **ZERO** bit is both available at the output of the component and through the `status` register.

Presence sensor

The following table lists the registers of the presence sensor (chosen through the port `addr`).

Register	Address (RISC-V)	Name	31...2	1	0
0	0x200C	control	*Reserved*		**LAUNCH**
1	0x2010	status	*Reserved*	**PRES**	**SCANNED**

`control` register

By setting the **LAUNCH** bit, the presence sensor launches a scan of the library. The **LAUNCH** bit is a *write-only* bit (i.e., when reading the `control` register this bit will always be read as 0). If a scan is still ongoing, writing 1 to **LAUNCH** aborts it and restarts a new scan. Writing 0 to the **LAUNCH** bit has no effect.

`status` register

1. The presence sensor returns the previous scan's result in the **PRES** bit. The sensor will set the **PRES** bit to 1 if it detected someone in the library during the scan, and to 0 otherwise. The **PRES** bit is *read-only*, writing to it has no effect.
2. The **SCANNED** bit is set to 1 when the sensor completes a scan. The **SCANNED** bit stays 1 until it is explictly cleared by software. Write 0 to the `status` register to clear the **SCANNED** bit. Writing 1 to the **SCANNED** bit has no effect. The **SCANNED** bit is both available at the output of the component and through the `status` register.

Light button

The following table lists the registers of the light button.

Register	Address (RISC-V)	Name	31...1	0
0	0x2014	button	*Reserved*	**PRESSED**

`button` register

The **PRESSED** bit is set to 1 when the button is pressed. The **PRESSED** bit stays 1 until it is explictly cleared by software. Write 0 to the `button` register to clear the **PRESSED**

bit. Writing 1 to the **PRESSED** bit has no effect. The **PRESSED** bit is both available at the output of the component and through the `button` register.

Lights

The following table lists the registers of the lights.

Register	Address (RISC-V)	Name	31...1	0
0	0x2018	lights	*Reserved*	**ON/OFF**

`lights` register

The **ON/OFF** bit can be set to 1 or 0 to, respectively, turn lights on or off. Overwriting this bit with the same value has no effect. Reading the **ON/OFF** bit allows to determine whether lights are on or off in the library.

Hardware System

a) Complete the provided system diagram in such a way to obtain the I/O memory map described above. Note the following:

- When the timer reaches 0, the `irq0` pin should be enabled on the RISC-V processor. When the presence sensor completes a scan, pin `irq1` should be enabled. Finally, when the light button is pressed, pin `irq2` should be enabled.
- For each peripheral, the `write_en` input signal should be 1 when the data coming through `data_in` should be written to one of the peripheral's internal registers (indicated by `addr` when relevant). `data_out` always outputs the internal register's value pointed to by `addr`, hence the outgoing signal must be appropriately controlled to avoid conflicts on the bus. Figure 46 indicates the width and direction of all ports.
- For each peripheral, the write process is *synchronous*. However, the read process has a perfectly combinatorial (i.e., *asynchronous*) interface: given an address (when relevant), the value of the corresponding register is outputted a combinatorial delay later.

In your diagram, you may use any of the logic components from Figure 47. Recall that, in RISC-V convention, the *read* process of the processor is *synchronous* and has a latency of one cycle. The *write* process is *synchronous* and has no latency cycles. Timing diagrams for typical *read* and *write* processes are shown in Figure 48.

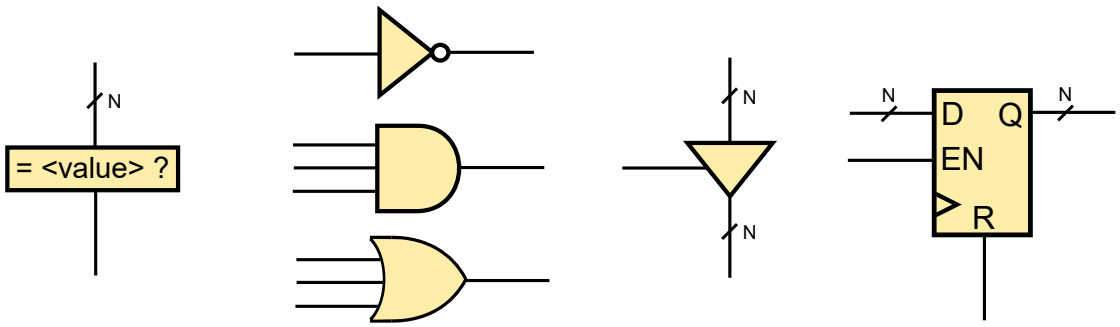


Figure 47: Logic components you are allowed to use in your diagram. From left to right: (1) equality test, (2) `NOT`, `AND`, and `OR` (arbitrary number of inputs allowed for `AND` and `OR`), (3) tri-state buffer, (4) flip-flop (`EN` and `R` signals are optional).

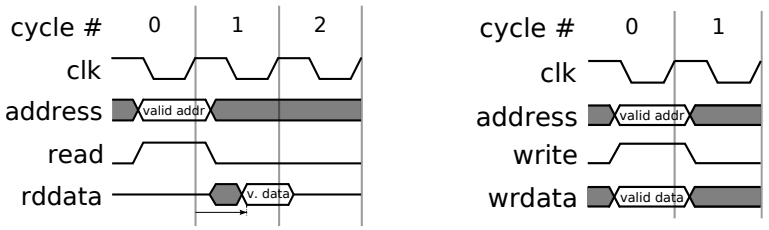


Figure 48: Timing diagrams of RISC-V for a typical *read* transaction (on the left) and *write* transaction (on the right).

Software Support

In the following questions, please use RISC-V assembly, registers, and calling conventions. This exercise uses a nonstandard RISC-V core where `irq0` connects to bit 0 of `mie` and `mip`. Concretely, the standard behaviour of RISC-V is modified in the following way: (i) if bit 0 of `mie` is 0, interrupts triggered when enabling `irq0` are ignored, even if MEIE is set; (ii) reading bit 0 of `mip` indicates if `irq0` is active. `irq1` and `irq2` are connected similarly to bits 1 and 2 respectively.

b) Implement the `init` procedure which **initializes the system** and then puts the processor into **low-power mode**. You can assume that lights are off when the processor starts and that the light button isn't pressed before the processor is put in low-power mode. ROM is mapped in memory from address `0x0000` (included) to address `0x1000` (excluded) and is followed by RAM mapped from address `0x1000` (included) to address `0x2000` (excluded). The `init` procedure should:

1. Initialize the stack pointer to address `0x2000`. Following RISC-V conventions, the stack should grow **toward lower memory addresses**.
2. Set the timer's period so that the timer reaches zero 10 minutes after being started. The timer's clock runs at a fixed 1 KHz.
3. Make the CPU able to receive interrupts from all interrupt-generating peripherals (timer, light button, and presence sensor) and **enable interrupts globally**.
4. Use the **`halt`** instruction to put the CPU in low-power mode (more details about this instruction are given below).

The particular RISC-V processor we use here has an additional instruction called **`halt`** that is not part of the standard RISC-V ISA. Use the **`halt`** instruction to put the processor in low-power mode until an interrupt is detected.
The `halt` instruction is functionally equivalent to the following code:

```
01   wait:
02     j wait
```

The only difference is that the code above will be executed in full-power mode, whereas the instruction **`halt`** puts the processor into low-power mode. Upon the detection of an interrupt, the processor leaves low-power mode and handles interrupts as usual.

c) Write the interrupt handler `int_handler` and the three interrupt service routines `button_isr`, `timer_isr`, and `sensor_isr` corresponding to, respectively, the light button, countdown timer, and presence sensor peripherals. ISRs should be **functions** which the interrupt handler can **`call`** and which end with **`ret`**. You are free to define and use as many helper functions as you need, as well as **`.equ`** statements. The system should implement the specification defined in subsection 10.

The main idea is to start the timer every time lights are turned on using the light button, and to stop and reset it whenever lights are turned off. When the timer reaches 0 (i.e., when 10 minutes have passed with lights remaining on), the system launches a scan

using the presence sensor. The scan's result is available after a few seconds and is signalled by the sensor raising an interrupt on `irq1`:

- If the sensor detected a presence (1 in **PRES** bit) **and the lights are still on** then they should remain on and the timer should be restarted.
- Otherwise (0 in **PRES** bit) the lights may be turned off to save energy.

When designing your interrupt handler, make sure you handle the case where there are **multiple interrupts pending**. Each ISR is **responsible for acknowledging** its corresponding interrupt. Upon returning from the interrupt handler, the previously aborted **`halt`** instruction **should be executed again**, which puts back the processor into low-power mode.

Access control using a card reader

The library's front door is controlled using a magnetic card reader, which—concurrently with the lighting system—communicates with the RISC-V processor through memory as well as a **4th interrupt lane `irq3`** (connected to `mie` and `mip` as described before). To enter the library students must swipe their card in the reader which reads out a 32-bits **U**nique **S**tudent **ID**entifier (USID) from the card's magnetic band. This USID is then sent to the university's server over the network—which can take some time—to check whether the student is authorized to access that room. The following table lists the registers of the card reader.

Register	Address (RISC-V)	Name	31...0
0	`0x201C`	`data`	USID

When a student swipes their card:

- Their unique identifier is loaded into the `data` register.
- The card reader generates an IRQ on `irq3`.

Reading the USID from the `data` register **acknowledges the interrupt**.

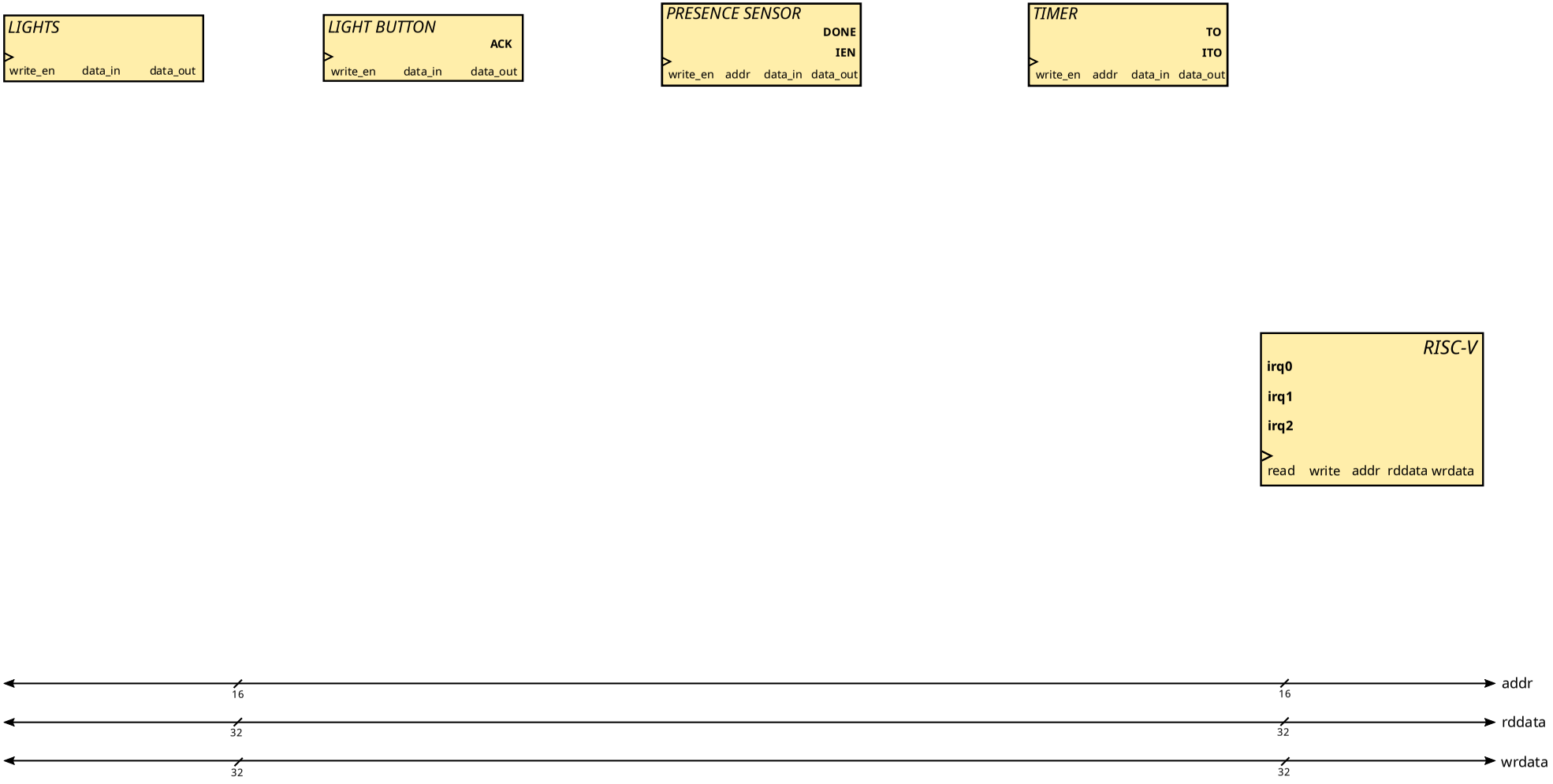
Software support

d) Quickly explain what you would **add or change in `init` and `int_handler`** to be able to receive the card reader's interrupts and call the card reader's ISR `reader_isr` whenever an interrupt from the card reader is pending. You do **not** have to rewrite the implementations of `init` and `int_handler` completely, simply point out what you would modify to support the new peripheral.

e) Write the card reader's ISR `reader_isr` which is called whenever a student swipes their card at the library's front door. Similarly to the other ISRs, `reader_isr` must behave like a function. You can assume that **no student swipes their card** while the previous student is waiting for the system to authorize or deny access. The card reader's ISR should:

1. Read the USID from the card reader's internal register (which acknowledges the interrupt).
2. Call the existing `check_authorization` function (you do **not** have to implement it), which expects to be passed the **USID as an argument in the `a0` register**. The function internally contacts the university's server to determine whether the student should be able to enter the room; based on the server's response, it automatically unlocks the door to let the student enter the library. The function itself doesn't return any value.

   `check_authorization` can take a few seconds to return (sometime the university's server and network can be slow to answer), during which the library's lighting system should continue to operate as smoothly as before. In particular, the light button should flip the lights immediately when it's pressed. Hence, make sure the card reader's ISR can itself be **interrupted by the other three interrupt sources**.
3. Return to the interrupt handler.


LIGHTS
write_en
data_in
data_out
LIGHT BUTTON
ACK
write_en
data_in
data_out
PRESENCE SENSOR
DONE
IEN
write_en
addr
data_in
data_out
TIMER
TO
ITO
write_en
addr
data_in
data_out
RISC-V
irq0
irq1
irq2
read
write
addr
rddata
wrdata
16
16
addr
32
32
rddata
32
32
wrdata

[Solution 10]

a) See diagram at the end.

b) Throughout the exercise, we'll use the following constants:

```
.equ TIMER_PERIOD  , 0x2000
.equ TIMER_CONTROL , 0x2004
.equ TIMER_STATUS  , 0x2008
.equ SENSOR_CONTROL, 0x200C
.equ SENSOR_STATUS , 0x2010
.equ LIGHT_BUTTON  , 0x2014
.equ LIGHTS        , 0x2018
.equ READER        , 0x201C
```

The initialization code is:

```
init:
    # Initialize stack pointer
    li sp, 0x2000
    # Set the timer's period to 1000 * 10 * 60 * 6000 - 1
        # = 599 999
    # 599 999 = 512 000 + 64 000 + 23 999
        # = (2^9 + 2^6) * 1000 + 23 999
    li   t0, 0x1000
    slli t1, t0, 9
    slli t2, t0, 6
    add t0, t1, t2
    li  t1, 23999
    add t0, t0, t1

    la  t1, TIMER_PERIOD
    sw  t0, 0(t1)
    # Enable all interrupt sources
    addi t0, zero, 0x807
    csrrw zero, mie, t0
    addi t0, zero, 8
    csrrw zero, mstatus, t0
    # Put the CPU in low-power mode
    halt
```

c) We first implement the interrupt handler:

```
int_handler:
```

```
    addi sp, sp, -4
    sw  ra, 0(sp)
    csrrw s0, mip, zero
    # Simply handle every interrupt source independently
button_check:
    andi s1, s0, 4
    beq s1, zero, sensor_check
    call button_isr
sensor_check:
    andi s1, s0, 2
    beq s1, zero, timer_check
    call sensor_isr
timer_check:
    andi s1, s0, 1
    beq s1, zero, int_handler_ret
    call timer_isr
int_handler_ret:
    lw   ra, 0(sp)
    addi sp, sp, 4
    mret # Re-execute the aborted halt instruction
```

The light button's ISR:

```
button_isr:
    # Flip the lights
    la t1, LIGHTS
    lw  t0, 0(t1)
    xori t0, t0, 1 # flip LSB of t0
    sw  t0, 0(t1)
    beq t0, zero, lights_off
lights_on:
    # Lights have been flipped on, start the timer
    addi t0, zero, 2
    j  button_isr_ret
lights_off:
    # Lights have been flipped off, stop the timer
    addi t0, zero, 1
button_isr_ret:
    la t1, TIMER_CONTROL
    sw  t0, 0(t1) # Start or stop the timer
    la t1, LIGHT_BUTTON
    sw  zero, 0(t1) # Ack the button's interrupt
    ret
```

The presence sensor's ISR:

```
sensor_isr:
    # Check if the sensor detected a presence
    la t2, SENSOR_STATUS
    lw  t0, 0(t2)
    andi t0, t0, 2
    beq t0, zero, no_presence
    # There is someone in the library
    la t2, LIGHTS
    lw  t1, 0(t2)
    beq t1, zero, sensor_isr_ret # If lights are already off,
        # do nothing
    # Lights are still on, restart the timer
    addi t1, zero, 2
    la t2, TIMER_CONTROL
    sw  t1, 0(t2)
    j  sensor_isr_ret
no_presence:
    la  t2, LIGHTS
    sw  zero, 0(t2) # No one in the library
                            # turn lights off
sensor_isr_ret:
    la  t2, SENSOR_STATUS
    sw  zero, 0(t2) # Ack the sensor's interrupt
    ret
```

The timer's ISR:

```
timer_isr:
    la  t1, LIGHTS
    lw  t0, 0(t1)
    beq t0, zero, timer_isr_ret # If lights are off,
                                    # don't start a scan
    la  t1, SENSOR_CONTROL
    sw  t0, 0(t1) # Start a presence scan
timer_isr_ret:
    la  t1, TIMER_STATUS
    sw  zero, 0(t1) # Ack the timer's interrupt
    ret
```

d) In `init`; we only need to change the set of interrupt sources we enable in `mie`:

```
li t0, 0x80F
csrrw zero, mie, t0
```

In `int_handler`, we need to handle the card reader's interrupts. We add the following snippet before the `button_check` label:

```
    ...
reader_check:
    addi s1, zero, 8
    and s1, s0, s1
    beq s1, zero, button_check
    call reader_isr
button_check:
    ...
```

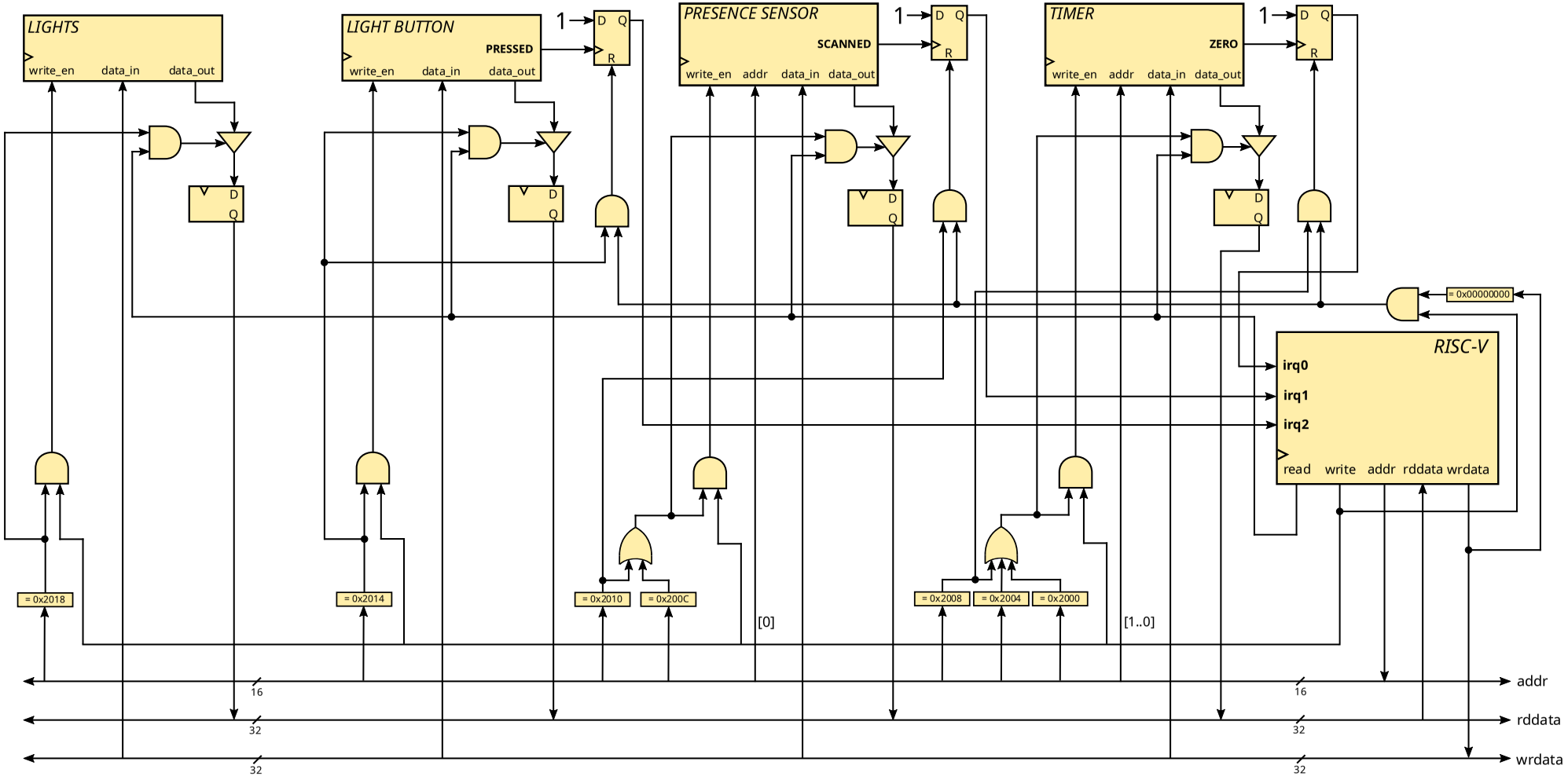
We'll also be calling and potentially interrupting the `check_authorization` function, which might use any register. Hence we need to save in the stack all registers we use within the interrupt handler and ISRs at the beginning. We restore them at the end.

```
int_handler:
    addi sp, sp, -28
    sw  ra, 0(sp)
    sw  a0, 4(sp)
    sw  t0, 8(sp)
    sw  t1, 12(sp)
    sw  t2, 16(sp)
    sw  s0, 20(sp)
    sw  s1, 24(sp)
    csrrw s0, mip, zero
    ...
int_handler_ret:
    lw  s1, 24(sp)
    lw  s0, 20(sp)
    lw  t2, 16(sp)
    lw  t1, 12(sp)
    lw  t0, 8(sp)
    lw  a0, 4(sp)
    lw  ra, 0(sp)
    addi sp, sp, 28
```

e) The card reader's ISR:

```
reader_isr:
    addi sp, sp, -8
    sw  ra, 0(sp)
    csrrw t0, mepc, zero
    sw  t0, 4(sp) # Save mepc because the ISR might get interrupted

```

```
7     la  t0, READER
8     lw  a0, 0(t0)
9     # Enable three other interrupt sources
10    addi t0, zero, 7
11    csrrs zero, mie, t0
12    # Re-enable interrupts globally
13    addi t0, zero, 8
14    csrrs zero, mstatus, t0
15    call check_authorization
16    # Disable interrupts globally
17    addi t0, zero, 8
18    csrrc zero, mstatus, zero
19    # Re-enable all interrupt sources
20    addi t0, zero, 15
21    csrrs zero, mie, t0
22
23    # Restore registers and return
24    lw  t0, 4(sp)
25    csrrw zero, mepc, t0
26    lw  ra, 0(sp)
27    addi sp, sp, 8
28    ret
```


LIGHTS
write_en
data_in
data_out
LIGHT BUTTON
PRESSED
write_en
data_in
data_out
PRESENCE SENSOR
SCANNED
write_en
addr
data_in
data_out
TIMER
ZERO
write_en
addr
data_in
data_out
1
D
Q
R
= 0x00000000
RISC-V
irq0
irq1
irq2
read
write
addr
rddata
wrdata
= 0x2018
= 0x2014
= 0x2010
= 0x200C
= 0x2008
= 0x2004
= 0x2000
[0]
[1..0]
16
32
addr
rddata
wrdata

[Exercise 11] System Design

Consider a security system composed of a RISC-V processor connected to a main memory storage and four peripherals through a bus as shown in Figure 49. The first peripheral is a video detector that records continuously for surveillance. The second peripheral is a video analyzer which does video analysis detecting temporal and spatial events. The third and fourth peripherals are a power failure sensor and an electric generator, respectively, together providing a backup power supply, necessary in the event of unexpected power outages (more details are provided later).

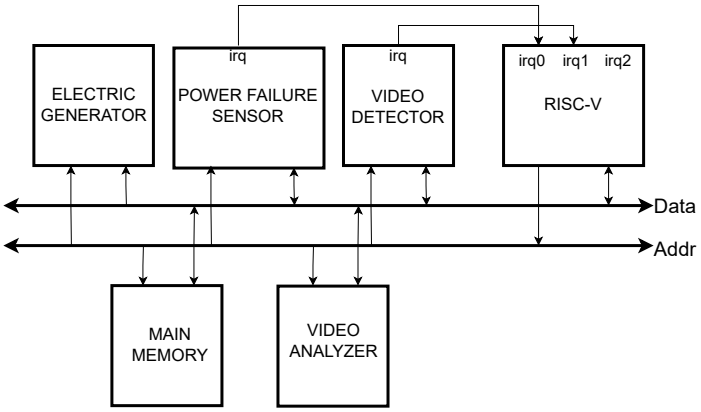


Figure 49: RISC-V Processor and System Peripherals.

Details of the Peripherals

Video Detector

The video detector is composed of a camera and a small internal storage buffer. It is managed by a single register (`video_status`) that holds control information. The MEMFULL bit in the `video_status` register is the only bit of interest. When the internal video frame buffer of the video detector peripheral is full, it sets the value of MEMFULL to 1. It remains 1 until it is explicitly cleared by writing 0 to it. On the contrary, when the internal storage is not yet full, the value of MEMFULL is 0. Writing 1 to the MEMFULL bit has no effect.

Register	Address	Name	31...1	0
0	`0x201C`	`video_status`	*Reserved*	MEMFULL

Video Analyzer

The video analyzer is a complete and independent subsystem that receives video frames through its input data port memory mapped at address `0x2018` (see details later in the memory map), processes them, and transmits the results to a remote server—the details of its functions are irrelevant to the scope of this exercise. It has no control registers of interest.

Power Failure Sensor

The power failure sensor detects when the mains power supply has an outage. When this happens, the whole system is not powered anymore and relies on a supercapacitor (i.e., a sort of very small battery) to operate for one or two seconds more. It is essential that as soon as a power outage is detected, an electric power generator is switched on very quickly (see below). The power failure sensor has a single status register `power_status` with a single bit (OUTAGE) of interest. In the event of a power outage, the OUTAGE bit is set to 1, and it remains 1 until explicitly cleared by the software. On the contrary, while mains power keeps coming, the OUTAGE bit is 0 and writing any value to it has no effect.

Register	Address	Name	31...1	0
0	`0x2010`	`power_status`	*Reserved*	OUTAGE

Electric Generator

The electric generator is managed by a single write only control register `generator_ctrl` with a single bit of interest (GENERATE). When the GENERATE bit is set to 1, it starts generating electricity. The generator automatically switches off by itself when it senses that the mains power is restored.

Register	Address	Name	31...1	0
0	`0x2014`	`generator_ctrl`	*Reserved*	GENERATE

System's Specifications

The system should operate as follows:

1. The video detector processes the video frames and stores the processed data in its small internal storage which is mapped in main memory.
2. When the video detector's internal storage is full, the stored data should be sent to the video analyzer through the memory mapped port at address `0x2018`; each word from the video detector's internal storage must be written sequentially to address `0x2018`. Then, the MEMFULL bit of the `video_status` register should be cleared.
3. In the event of the power failure sensor detecting an outage, the electric generator must be switched on as soon as possible and the OUTAGE bit of the `power_status` register should be cleared.

Memory Map

Figure 50 shows the memory layout of the system.

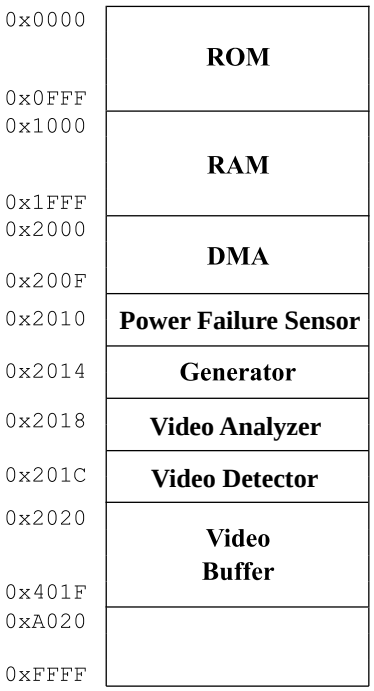


Figure 50: System's memory layout. The DMA controller is not relevant and can be ignored until Part 5.

Problem Statement

The goal of this exercise is to identify the best design that interfaces the RISC-V processor with the peripherals and implements the desired system behaviour such that (1) the time between the moment a power outage occurs and the electric generator is started is minimized in the worst-case scenario and (2) the processor's idle time is maximized.

In the following questions, you should abide by the following conventions:

1. Use RISC-V assembly, registers, and calling conventions.
2. Each instruction on the given RISC-V processor takes 3 cycles to execute.
3. Whenever the processor is idle, use the **`halt`** instruction that puts the processor in low-power mode. The **`halt`** instruction takes 3 cycles to execute before switching the processor to low-power mode (more details about this instruction are given below).

4. At power-up, the processor starts executing instructions at label `main`, where it should initialize the stack pointer with value `0x2000`.

5. When the processor is interrupted, it starts executing instructions at label `interrupt_handler`. It takes the processor 0 cycles to jump to the interrupt handler when an interrupt is raised by a peripheral.

The particular RISC-V processor we use here has an additional instruction called **`halt`** that is not part of the standard RISC-V ISA. Use the **`halt`** instruction to put the processor in low-power mode until an interrupt is detected.
The `halt` instruction is functionally equivalent to the following code:

```
01   wait:
02     j wait
```

The only difference is that the code above will be executed in full-power mode, whereas the instruction **`halt`** puts the processor into low-power mode. Upon the detection of an interrupt, the processor leaves low-power mode and handles interrupts as usual.

In the following questions, **you are free to reuse or refer to code you defined in previous parts of the exercise** in later questions provided that you state clearly which part of your implementation you take advantage of.

Part 1: Transferring Data

A function called `transfer_data` is needed to implement the logic of copying data from a memory area into a single address of the input port of a peripheral. It should take as arguments the number of words to transfer (which is a number that can be represented in 16 bits), the address of the first word in the memory area to start copying from (source address) and the address of the input port of the peripheral to copy to (destination address). The arguments are passed in registers `a0`, `a1`, and `a2`, respectively. The data transfer is word aligned and consists in copying complete 4-byte words one after the other, sequentially, from the first word in the source memory area to the fixed input port address.

For the specified system, the `transfer_data` function is needed to transfer the content of the video detector's internal buffer, which has its start address mapped at

`0x2020` and is 8 KiB long (see Figure 50), to the video analyzer's input port mapped at address `0x2018`.

a) Implement the described `transfer_data` function.

b) Use your implementation of the `transfer_data` function to count the number of cycles needed for executing it in terms of the number of words transferred.

Part 2: System Design Using Polling

The simplest method that allows the processor to communicate with peripherals is polling.

c) Write the complete code implementing the specification above and including everything needed from the moment the processor is switched on; in this first case, have the processor communicate with the peripherals by polling.

d) Explain, in words, the worst case scenario that will result in the maximum wait time between the occurrence of a power outage and starting the operation of the electric generator.

e) Considering the worst case scenario, compute the maximum wait time (in cycles) between the occurrence of a power outage and starting the operation of the electric generator.

Part 3: System Design Using Nonnested Interrupts

An alternative to polling is to use interrupts to interface the processor with the video detector and the power failure sensor. Given that the occurrence of a power outage requires a response within a minimum number of cycles, interrupts from the power failure sensor should be served first if interrupts from the two peripherals are issued at the same time. Interrupts from the power failure sensor arrive to the processor at `irq0`, whenever the OUTAGE bit of its `power_status` register is 1, and interrupts from the video detector arrive to the processor at `irq1`, whenever the MEMFULL bit

of its `video_status` register is 1. In this nonnested approach, further interrupts are ignored while the processor is already serving any other interrupt.

This exercise uses a nonstandard RISC-V core where `irq0` connects to bit 0 of `mie` and `mip`. Concretely, the standard behaviour of RISC-V is modified in the following way: (i) if bit 0 of `mie` is 0, interrupts triggered when enabling `irq0` are ignored, even if MEIE is set; (ii) reading bit 0 of `mip` indicates if `irq0` is active. `irq1` and `irq2` are connected similarly to bits 1 and 2 respectively.

f) Write the complete code implementing the specification above and including everything needed from the moment the processor is switched on; in this second case, write code for enabling interrupts, the interrupt handler and interrupt service routines (ISRs) to handle interrupts from the two peripherals.

g) Explain, in words, the worst case scenario that will result in the maximum wait time between the occurrence of a power outage and starting the operation of the electric generator.

h) Considering the worst case scenario, calculate the maximum wait time (in cycles) between the occurrence of a power outage and starting the operation of the electric generator.

Part 4: System Design Using Nested Interrupts

A more sophisticated implementation of interrupts allows for nesting one interrupt into another. We want to allow an interrupt from the power failure sensor to interrupt the interrupt handling of the video detector. However, we do not want interrupts from the video detector to interrupt those from the power failure sensor. As in the previous part, interrupts from the power failure sensor arrive to the processor at `irq0`, and interrupts from the video detector arrive to the processor at `irq1`.

i) Write the complete code implementing the specification above and including everything needed from the moment the processor is switched on; similarly to the nonnested case, write code for enabling interrupts, the interrupt handler and interrupt service routines (ISRs) to handle interrupts from the two peripherals. Additionally,

write code to allow the video detector to nest interrupts from the power failure sensor.

j) Explain, in words, the worst case scenario that will result in the maximum wait time between the occurrence of a power outage and starting the operation of the electric generator.

k) Considering the worst case scenario, calculate the maximum wait time (in cycles) between occurrence of a power outage and starting the operation of the electric generator.

Part 5: System Design Using DMA with Interrupts

Consider extending the system by adding a Direct Memory Access (DMA) controller which takes care of the memory copy operations without any processor intervention. The processor is only responsible for configuring the DMA controller's internal registers (more details are provided below), then can move on to executing other instructions or go into an idle state. The new system schematic is shown in Figure 51. Not shown on the figure is a memory arbiter that prevents memory conflicts on the bus between the RISC-V processor and the DMA controller. You *do not* need to take into account the latency of this arbiter in your calculations, i.e., you can assume that the memory bus is always immediately available to any peripheral needing to use it.

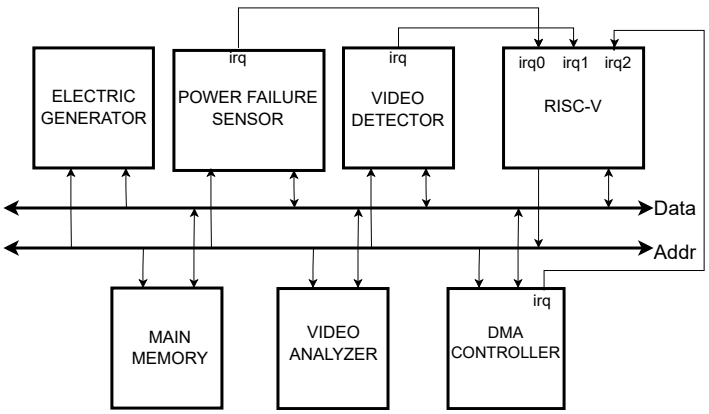


Figure 51: RISC-V Processor and System Peripherals including the DMA controller.

DMA Controller's Specification

For the DMA controller to start the data transfer, it requires some information to be provided by the processor in its registers. The memory-mapped source and destination addresses in main memory to use for the transfer should be written into the `dma_addr` register. The `dma_cfg` register is used to specify additional information about the memory transfer: The word count indicates how many 4-byte words will be copied during the transfer. The source and destination strides represent, respectively, the offset that is added to the source and destination addresses between two consecutive words (e.g., a stride of 4 indicates that words will be read or written contiguously and a stride of 0 indicates that all words will be read or written at the same address). The START bit in the `dma_ctrl` register should be set to 1 to start the data transfer. Writing 0 to the START bit has no effect. Writing 1 to the START bit while a transfer is ongoing has also no effect. The DMA controller sets the INTR bit to 1 upon completion of the data transfer and issues an interrupt to the processor on `irq2`. The INTR bit remains high until it is explicitly cleared by writing 0 to it.

Register	**Address**	**Name**	**31 ... 16**	**15 ... 8**	**7 ... 1**	**0**
0	`0x2000`	`dma_addr`	Source address	Destination address		
1	`0x2004`	`dma_cfg`	Word count	Source stride	Destination stride	
2	`0x2008`	`dma_ctrl`	*Reserved*			START
3	`0x200C`	`dma_status`	*Reserved*			INTR

New System Specification

The goal of this part is to modify the system's design to add support for DMA. The system should behave as follows:

1. When the video detector raises an interrupt, the DMA controller should be instructed to start a data transfer by loading appropriately its registers.
2. During the data transfer, the processor should not be sensitive to another interrupt from the video detector.
3. The DMA controller interrupts the processor when it completes the data transfer. After this, the processor should become sensitive to interrupts from the video detector again.
4. No interrupt nesting should be supported in this part.

5. You cannot use the `transfer_data` function in this part.

l) Write the complete code implementing the specification above and including everything needed from the moment the processor is switched on; write code for enabling interrupts, the interrupt handler and interrupt service routines (ISRs) to handle interrupts from the three peripherals (now including the DMA controller). Your code should instruct the DMA controller to do the data transfer when handling the interrupt from the video detector.

m) Explain, in words, the worst case scenario that will result in the maximum wait time between the occurrence of a power outage and starting the operation of the electric generator.

n) Considering the worst case scenario, calculate the maximum wait time (in cycles) between the occurrence of a power outage and starting the operation of the electric generator.

Part 6: Processor Idle Time

In this part we want to see how the different design choices explored before influence the processor utilization. Consider the following assumptions: (1) there are no outages during normal operation; (2) the video detector's internal storage buffer is full, on average, every 10ms; and (3) the processor runs at 10 MHz.

o) Compute, for each of the four system options developed in Part 2 to Part 5, the approximate fraction of time when the processor is idle (time executing the `halt` instruction). The result does not need to be exact but can be within, say, $\pm 5\%$; yet, you need to explain **very clearly** how you compute such idle fraction of the time and what components, if any, you disregard in your approximation.

p) Studying the four design choices, what do you think is the best design for the required system with respect to (1) the latency to start the electricity generator after the power outage occurs and (2) the amount of time the processor is left idle? Are there any qualitatively equivalent options? Briefly explain why the best design(s) perform(s) better than the others with respect to our objectives.

[Solution 11]

Throughout the exercise, we'll use the following constants:

```
.equ POWER_FAILURE_SENSOR, 0x2010
.equ GENERATOR           , 0x2014
.equ VIDEO_ANALYZER      , 0x2018
.equ VIDEO_DETECTOR      , 0x201C
.equ VIDEO_BUFFER        , 0x2020
```

Part 1: Transferring Data

a)

```
transfer_data:
    add t0, zero, a0 # retrieve the number of words to transfer
    add t1, zero, a1 # retrieve the source address of the first word
copy:
    beq t0, zero, end
    lw  t2, 0(t1)
    sw  t2, 0(a2) # write to the fixed destination port address
    addi t1, t1, 4
    addi t0, t0, -1
    j  copy
end:
    ret
```

b) Our implementation has 3 instructions outside the loop and 6 instructions inside the loop. Each instruction takes 3 cycles to execute. Therefore the number of cycles needed to execute the `transfer_data` function is

$$\text{Number of words} = \frac{8KiB}{4B} = \frac{(8 \times 1024)}{4} = 2048$$

$$\begin{aligned}\text{Number of cycles} &= (3 + 6 \times \text{Number of words}) \times 3 = 9 + 18 \times \text{Number of words} \\ &= 36,873\end{aligned}$$

Part 2: System Design Using Polling

c) We implement the polling design below.

```
main:
    li sp, 0x2000 # initialize the stack pointer

check_power_sensor:
    la  t1, POWER_FAILURE_SENSOR
    lw  t0, 0(t1) # check if a power outage is detected
    beq t0, zero, check_video_detector

handle_power_outage:
    addi t0, zero, 1
    la  t1, GENERATOR
    sw  t0, 0(t1) # start the generator
    la  t1, POWER_FAILURE_SENSOR
    sw  zero, 0(t1) # clear the power_status register

check_video_detector:
    la  t1, VIDEO_DETECTOR
    lw  t0, 0(t1) # check if the internal buffer is full
    beq t0, zero, check_power_sensor

handle_video_detector:
    li a0, 0x2000 # the video buffer size from Figure 2
    la a1, VIDEO_BUFFER
    la a2, VIDEO_ANALYZER
    call transfer_data
    la  t1, VIDEO_DETECTOR
    sw  zero, 0(t1) # clear the video_status register
    j   check_power_sensor # restart the poll loop
```

d) The worst case scenario happens when a power outage occurs one cycle after loading the value of the `power_status` register, and the internal buffer of the video detector becomes full before the `video_status` register is loaded.

e) In the worst case scenario, we'll execute 14 instructions and `transfer_data` before starting the electric generator. Considering the latency of instructions (3 cycles per instruction) and `transfer_data` (computed before), this creates a worst case latency of $14 \times 3 + 36,873 = 36,915$ cycles.

Part 3: System Design Using Nonnested Interrupts

f) We implement the nonnested interrupt design below.

```
interrupt_handler:
    csrrw t0, mip, zero

check_power_sensor:
    andi t1, t0, 1 # is there a power outage?
    beq t1, zero, check_video_detector

isr_power_sensor:
    addi t1, zero, 1
    la  t0, GENERATOR
    sw  t1, 0(t0) # start the generator
    la  t0, POWER_FAILURE_SENSOR
    sw  zero, 0(t0) # clear the power_status register

check_video_detector:
    andi t1, t0, 2# is the video detector's memory full?
    beq t1, zero, ih_return

isr_video_detector:
    li a0, 0x2000 # the video buffer size from Figure 2
    la a1, VIDEO_BUFFER
    la a2, VIDEO_ANALYZER
    call transfer_data
    la  t0, VIDEO_DETECTOR
    sw  zero, 0(t0) # clear the video_status register

ih_return:
    mret

main:
    li   sp, 0x2000 # initialize the stack pointer
    addi t0, zero, 8
    csrrs zero, mstatus, t0 # set the MIE bit to 1
    addi t0, zero, 0x803
    csrrs zero, mie, t0 # enable interrupts from the two peripherals
    halt # put the CPU in low-power mode
```

g) The worst case scenario happens when the video detector raises an interrupt and—when in the interrupt handler—a power outage occurs one cycle after the `mip` register is loaded.

h) In the worst case scenario, we'll execute 16 instructions and `transfer_data` before starting the electric generator. Considering the latency of instructions (3 cycles per

instruction) and `transfer_data` (computed before), this creates a worst case latency of $16 \times 3 + 36,873 = 36,921$ cycles.

Part 4: System Design Using Nested Interrupts

i) We reuse `main`, `check_power_sensor`, `isr_power_sensor`, and `check_video_detector` from the nonnested interrupt implementation. To add support for nested interrupts, we need to save the registers used by both `transfer_data` and the IH at the beginning of `interrupt_handler`. Otherwise the IH would override their values in case a power outage interrupts a video transfer.

```
interrupt_handler:
    addi sp, sp, -8 # save to stack
    sw  t0, 0(sp)
    sw  t1, 4(sp)
    csrrw t0, mip, zero
```

Conversely, we need to pop them from the stack at the end.

```
ih_return:
    lw  t0, 0(sp) # restore from stack
    lw  t1, 4(sp) # restore from stack
    addi sp, sp, 8
    mret
```

Finally, we rewrite the video detector's ISR so that it re-enables interrupts before calling `transfer_data`, and disables them after. We need to save `mepc` otherwise it would be lost when a nested interrupt happens.

```
isr_video_detector:
    li a0, 0x2000 # the video buffer size from Figure 2
    la a1, VIDEO_BUFFER
    la a2, VIDEO_ANALYZER

    addi sp, sp, -4 # save mepc in stack
    csrrw t0, mepc, zero # save mepc in stack
    sw  t0, 0(sp)

    # re-enable interrupts in order to keep getting interrupts from the p
    li t0, 0x801
    csrrw zero, mie, t0
    csrrsi zero, mstatus, 8
```

```
14
15      call transfer_data
16
17      # clear PIE and re-enable interrupts from the video_detector
18      csrrci zero, mstatus, 8
19      csrrsi  zero, mie, 3
20      lw  t0, 0(sp) # restore mepc from stack
21      csrrw zero, mepc, t0
22      addi sp, sp, 4
23
24      la t0, VIDEO_DETECTOR
25      sw  zero, 0(t0) # clear the video_status register
```

j) The worst case scenario happens when the video detector raises an interrupt and—when in the interrupt handler—a power outage occurs one cycle after the `mip` register is loaded (same as in the nonnested case).

k) In the worst case scenario, we'll execute 20 instructions before starting the electric generator. This time `transfer_data` may be interrupted and so we do not need to account for its latency in the worst case scenario. Considering the latency of instructions (3 cycles per instruction), this creates a worst case latency of $20 \times 3 = 60$ cycles.

Part 5: System Design Using DMA with Interrupts

l) We reuse everything but `check_video_detector` and `isr_video_detector` from the nonnested interrupt implementation. We modify the video detector's ISR once again to launch the DMA instead of calling `transfer_data`.

```
1  check_video_detector:
2      andi t1, t0, 2# is the video detector's memory full?
3      beq t1, zero, check_dma_controller
4
5  isr_video_detector:
6      # set up the source and destination addresses
7      la t2, VIDEO_BUFFER
8      la t3, VIDEO_ANALYZER
9      slli t2, t2, 16
10     or t2, t2, t3
11
12     la  t3, DMA_CONTROLLER
13     sw  t2, 0(t3)
```

```
    # set up the word count (0x2000) and strides
    # source stride is 4, destination stride is 0
    li  t2, 0x2000
    slli t2, t2, 16
    ori t2, t2, 0x0400
    sw  t2, 4(t3)
    # disable any future interrupts from the video_detector
    addi t2, zero, 0x805
    csrrw zero, mie, t2
    # start the DMA transfer
    addi t2, zero, 1
    sw  t2, 8(t3)
    j  ih_return
```

We then add support for interrupts coming from the DMA controller by adding the following code after the video detector's ISR.

```
check_dma_controller:
    andi t1, t0, 4 # is the DMA transfer done?
    beq t1, zero, ih_return

isr_dma_controller:
    # acknowledge the video detector and DMA controller's interrupt
    la  t1, VIDEO_DETECTOR
    sw  zero, 0(t1)
    la  t1, DMA_CONTROLLER
    sw  zero, 0xC(t1)
    addi t2, zero, 0x807
    csrrw zero, mie, t2 # re-enable interrupts from the video detector
```

m) The worst case scenario happens when the video detector raises an interrupt and—when in the interrupt handler—a power outage occurs one cycle after the `mip` register is loaded (same as in the nonnested and nested interrupts cases).

n) In the worst case scenario, we'll execute 24 instructions before starting the electric generator. Considering the latency of instructions (3 cycles per instruction), this creates a worst case latency of $24 \times 3 = 72$ cycles.

Part 6: Processor Idle Time

o) We make the following approximations:

1. Round up the number of cycles taken to transfer data that is calculated in part 1 to 36,000 cycles, which is 3.6 ms at 10 MHz.
2. The number of cycles needed to enable interrupts and identify the source of interrupt is negligible relative to the number of cycles needed to transfer data, therefore we do not account for it.

For each of the four given design choices, the proportion of time the CPU spends idling under these assumptions is:

1. Polling: 0%
2. Nonnested interrupts: 64%
3. Nested interrupts: 64%
4. DMA controller: 100%

p) With respect to the worst case wait time between the occurrence of a power outage and starting the generator, the design employing nested interrupts and the design using DMA are qualitatively equivalent in the sense that both allow the system to be sensitive to the occurrence of a power outage irrespective of the transfer of data. There is a slight difference in the calculated number of cycles from above, though, because of storing to the multiple registers of the DMA controller, but the difference is negligible.

With respect to the CPU idle time percentage, the DMA design choice is the best since it leaves the CPU fully idle during the transfer of data.

Part III: Memory Hierarchy

[Exercise 1] Cache Organization

Consider the memory access sequence at the following addresses:

$$1, 5, 20, 17, 5, 4, 2, 18, 43, 11, 43, 9, 17, 5, 6, 56, 19, 43$$

Or equivalently in hexadecimal:

$$1, 5, 14, 11, 5, 4, 2, 12, 2\text{B}, \text{B}, 2\text{B}, 9, 11, 5, 6, 38, 13, 2\text{B}$$

The memory is word addressable (1^{st} word = address 0, 2^{nd} = address 1, and so on).

a) Consider an initially empty, 2-way set-associative cache memory with an LRU (Least Recently Used) replacement policy, a capacity (size) of 16 words and single word block size. Indicate which accesses result in a hit and which result in a miss. Show the state of the cache after the access sequence.

b) Consider an initially empty, fully associative cache memory with an LRU (Least Recently Used) replacement policy, a capacity of 16 words and single word block size. Indicate which accesses result in a hit and which result in a miss. Show the state of the cache after the access sequence.

c) Consider an initially empty, 2-way set-associative cache memory with an LRU (Least Recently Used) replacement policy, a capacity of 16 words and a four-word block size. Indicate which accesses result in a hit and which result in a miss. Show the state of the cache after the access sequence.

d) Draw the structure of a cache memory similar to the one in the preceding question but with 32-bit addresses, 32-bit words and byte addressing. Clearly indicate the use of each bit of the address by describing the address format in detail.

[Solution 1] Cache Organization

a) Hits and Misses:

Address:	0x01	0x05	0x14	0x11	0x05	0x04	0x02	0x12	0x2B
Hit/Miss:	M	M	M	M	H	M	M	M	M

Address:	0x0B	0x2B	0x09	0x11	0x05	0x06	0x38	0x13	0x2B
Hit/Miss:	M	H	M	H	H	M	M	M	H

The final contents of the cache after the sequence of accesses are given in the table below:

Tag	**Data**
0x38	M[0x38]
(0x01→)0x09	M[0x09]
0x02	M[0x02]
0x2B	M[0x2B]
0x14	M[0x14]
0x05	M[0x05]
0x06	M[0x06]
None	Empty

Tag	**Data**
None	Empty
0x11	M[0x11]
0x12	M[0x12]
(0x0B→)0x13	M[0x13]
0x04	M[0x04]
None	Empty
None	Empty
None	Empty

Remarks:

- The order of each pair of corresponding elements in the two ways is irrelevant
- The tag does not actually contain the whole address (the three LSBs can be omitted); the complete address is given here only for readability
- Tag = `None` represents the Valid Bit = `'0'`.
- Bits 0 to 2 of the address select the block
- The replacement policy has been used twice

b) Hits and Misses:

Address:	0x01	0x05	0x14	0x11	0x05	0x04	0x02	0x12	0x2B
Hit/Miss:	M	M	M	M	H	M	M	M	M

Address:	0x0B	0x2B	0x09	0x11	0x05	0x06	0x38	0x13	0x2B
Hit/Miss:	M	H	M	H	H	M	M	M	H

The final contents of the cache after the sequence of accesses are given in the table below:

Tag	**Data**
0x01	M[0x01]
0x05	M[0x05]
0x14	M[0x14]
0x11	M[0x11]
0x04	M[0x04]
0x02	M[0x02]
0x12	M[0x12]
0x2B	M[0x2B]
0x0B	M[0x0B]
0x09	M[0x09]
0x06	M[0x06]
0x38	M[0x38]
0x13	M[0x13]
None	Empty
None	Empty
None	Empty

Remarks:

- The order of the table rows is irrelevant

- The tag contains the full address
- Tag = `None` represents the Valid Bit = `'0'`
- The replacement policy was irrelevant for this small number of accesses (no replacement performed)

c) Hits and Misses:

Address:	0x01	0x05	0x14	0x11	0x05	0x04	0x02	0x12	0x2B
Hit/Miss:	M	M	M	M	H	H	H	H	M

Address:	0x0B	0x2B	0x09	0x11	0x05	0x06	0x38	0x13	0x2B
Hit/Miss:	M	H	H	M	H	H	M	H	M

The final contents of the cache after the sequence of accesses are given in the table below:

	Tag	**Data**			
0	(0x00→)(0x28→)0x10	M[0x10]	M[0x11]	M[0x12]	M[0x13]
1	0x04	M[0x04]	M[0x05]	M[0x06]	M[0x07]

	Tag	**Data**			
0	(0x10→)(0x08→)(0x38→)0x28	M[0x28]	M[0x29]	M[0x2A]	M[0x2B]
1	0x14	M[0x14]	M[0x15]	M[0x16]	M[0x17]

Remarks:

- The order of each pair of corresponding elements in the two ways is irrelevant
- The tag does not actually contain the whole address (the three LSBs can be omitted); the complete address is given here only for readability
- All Valid Bits = `'1'`
- Bits 0 and 1 of the address are used to select one of the four words in the block

- Bit 2 of the address selects the block
- The replacement policy has been used five times

d) This case differs from the previous case because the last two bits of the address are actually used to identify one of the four bytes inside a word and are irrelevant for the cache.

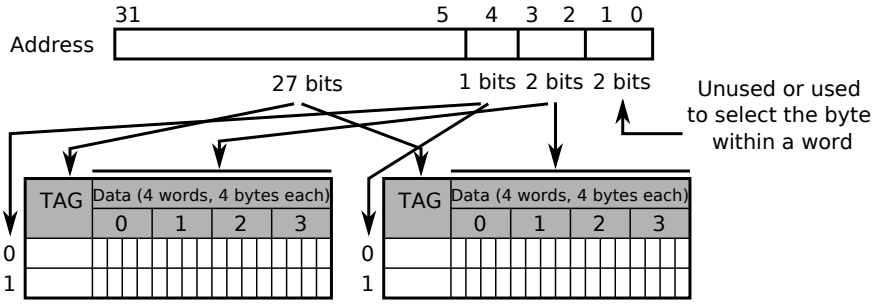


Figure 52: Structure of the cache for byte addressing

[Exercise 2] A Direct-Mapped Cache

Consider the following code for a RISC-V Processor (word addressable, 16-bit addresses, 32-bit data and `x0` is always '0').

```
0     lw   x4, 34(x0)    # x4=M[34]
1     lw   x5, 35(x0)    # x5=M[35]
2     mv   x6, x4        # x6=x4
3  loop:
4     lw   x2, 1(x4)     # x2=M[x4 + 1]
5     mul  x2, x2, x2    # x2=x2 * x2
6     beq  x5, x0, end   # if x5 = x0 then goto end
7     addi x2, x0, -1    # x2=-1
8     lw   x1, 0(x4)     # x1=M[x4]
9     beq  x1, x0, end   # if x1==x0 then goto end
10    mv   x4, x1        # x4=x1
11    add  x4, x4, x6    # x4=x4 + x6
12    addi x5, x5, -1    # x5=x5 - 1
13    j    loop          # goto loop
14 end:
```

Suppose that the state of the memory is as follows:

Address	Data
34	36
35	4
36	4
37	41
38	10
39	42
40	2
41	18
42	8
43	21
44	0
45	0
46	6
47	38

The program is stored in memory beginning at address '0'.

a) Given an initially empty, direct-mapped cache memory with a size of 32 words and four words per block, indicate which accesses result in a hit and the ones that result in a miss. Show the state of the cache after the sequence of accesses (after the Fetch phase of 13^{th}instruction). For each entry (location) in the cache indicate the address of the stored information. Assume the cache is unified, i.e. shared for Instruction and Data accesses.

b) Draw the structure of the cache memory. Clearly indicate the use of each bit of the address, i.e. give a detailed description of the address format.

[Solution 2] A Direct-Mapped Cache

a) The following is the access sequence that needs to be considered for the cache:

0, 34, 1, 35, 2, 3, 37, 4, 5, 6, 7, 36, 8, 9, 10, 11, 12, 3, 41, 4, 5, 6, 7, 40, 8, 9, 10, 11, 12, 3, 39, 4, 5, 6, 7, 38, 8, 9, 10, 11, 12, 3, 47, 4, 5, 6, 7, 46, 8, 9, 10, 11, 12, 3, 43, 4, 5, 13

Hits and Misses:

Address:	0	34	1	35	2	3	37	4	5
Hit/Miss:	M	M	M	M	M	H	M	M	H

Address:	6	7	36	8	9	10	11	12	3
Hit/Miss:	H	H	M	M	H	H	H	M	H

Address:	41	4	5	6	7	40	8	9	10
Hit/Miss:	M	M	H	H	H	H	M	H	H

Address:	11	12	3	39	4	5	6	7	38
Hit/Miss:	H	H	H	M	M	H	H	H	M

Address:	8	9	10	11	12	3	47	4	5
Hit/Miss:	H	H	H	H	H	H	M	M	H

Address:	6	7	46	8	9	10	11	12	3
Hit/Miss:	H	H	H	H	H	H	H	M	H

Address:	43	4	5	13
Hit/Miss:	M	H	H	H

Final State of the Cache:

	Tag	**Data**			
000	(0→32→0→32→) 0	M[0]	M[1]	M[2]	M[3]
001	(36→4→36→4→36→4→36→) 4	M[4]	M[5]	M[6]	M[7]
010	(8→40→8→) 40	M[40]	M[41]	M[42]	M[43]
011	(12→44→) 12	M[12]	M[13]	M[14]	M[15]
100	None	Empty	Empty	Empty	Empty
101	None	Empty	Empty	Empty	Empty
110	None	Empty	Empty	Empty	Empty
111	None	Empty	Empty	Empty	Empty

Remarks:

- The tag does not actually contain the whole address (the five LSB's should be omitted); the complete address is given here only for readability.
- The first four Valid Bits = '1' and the last four Valid Bits = '0'.
- Bits 0 and 1 of the address are used to select one of the four words in the block.
- Bits 2, 3 and 4 of the address select the cache line.
- It is a direct-mapped cache, hence no replacement policy is needed.

b) The structure of the cache is given in the schematic below:

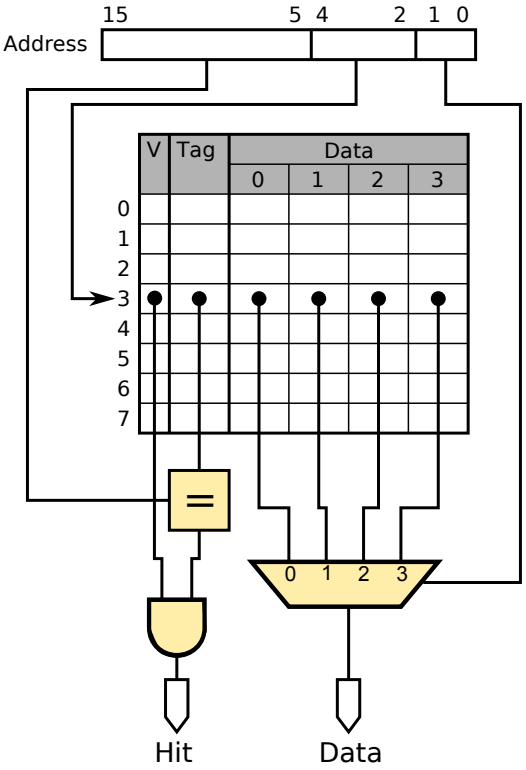


Figure 53: Structure of the cache

[Exercise 3] A 2-Way Set Associative Cache

Consider the following code for a RISC-V processor (word addressable, 16-bit addresses, 32-bit data and `x0` is always '0'):

```
0       lw   x4, A(x0)     # x4=M[A]
1       lw   x5, B(x0)     # x5=M[B]
2       lw   x6, C(x0)     # x6=M[C]
3  LOOP:beq  x5, x0, end   # if x5=x0 goto end
4       lw   x1, 0(x4)     # x1=M[x4]
5       addi x4, x4, 1     # x4=x4 + 1
6       lw   x2, 0(x4)     # x2=M[x4]
7       mul  x2, x1, x2    # x2=x1 * x2
8       sw   x2, 0(x6)     # M[x6]=x2
9       addi x6, x6, 1     # x6=x6 + 1
10      addi x5, x5, -1    # x5=x5 - 1
11      j    LOOP          # go to LOOP
12 end:
```

Suppose that the initial values are:

A=16	M[16]=20
B=17	M[17]= 4
C=18	M[18]=50

The program is stored in the memory starting at address '0'.

a) Given an initially empty, 2-way set-associative cache with an LRU (Least Recently Used) replacement policy, four-word blocks and a total size of 32 words, indicate the memory accesses that result in a hit and those that lead to a miss. Show the state of the cache after the sequence of accesses. For each cache entry, indicate the address of the stored information.

b) Draw the structure of the cache. Indicate the use of each bit of the address by describing its format in detail.

[Solution 3] A 2-way Set Associative Cache

a) The step-by-step execution of the program is given below:

```
0   lw x4, 16(x0) # x4=20. Read 0 (instr.) and 16 (M[16]): Misses
```

	Data			
	0	1	2	3
0	0	1	2	3
1				
2				
3				

	Data			
	0	1	2	3
0	16	17	18	19
1				
2				
3				

```
1   lw  x5, 17(x0)  # x5=4.  Rd 1 (instr.) and 17 (M[17]): Hits
2   lw  x6, 18(x0)  # x6=50. Rd 2 (instr.) and 18 (M[18]): Hits
3 LOOP:
4   beq x5, x0, end # x5=0.  Rd 3 (instr.): Hit
5   lw  x1, 0(x4)   # Rd 4 (instr.) and 20 (M[20]): Misses
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2				
3				

	Data			
	0	1	2	3
0	16	17	18	19
1	20	21	22	23
2				
3				

```
5       addi x4, x4, 1  # x4=21. Rd 5 (instr.): Hit
6       lw   x2, 0(x4)  # Rd 6 (instr.) and 21 (M[21]): Hits
7       mul  x2, x1, x2 # Rd 7 (instr.): Hit
8       sw   x2, 0(x6)  # Rd 8 (instr.); Wr 50 (M[50]): Misses
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				

	Data			
	0	1	2	3
0	48	49	50	51
1	20	21	22	23
2				
3				

```
9       addi x6, x6, 1  # x6=51. Rd 9 (instr.): Hit
10      addi x5, x5, -1 # x5=3. Rd 10 (instr.): Hit
11      j    LOOP       # Rd 11 (instr.): Hit
```

```
3     beq  x5, x0, end# Rd 3 (instr.): Hit
4     lw   x1, 0(x4)  # Rd 4 (instr.) and 21 (M[21]): Hits
5     addi x4, x4, 1  # x4=22. Rd 5 (instr.): Hit
6     lw   x2, 0(x4)  # Rd 6 (instr.) and 22 (M[22]): Hits
7     mul  x2, x1, x2 # Rd 7 (instr.): Hit
8     sw   x2, 0(x6)  # Rd 8 (instr.) and Wr 51 (M[51]): Hits
9     addi x6, x6, 1  # x6=52. Rd 9 (instr.): Hit
10    addi x5, x5, -1 # x5=2. Rd 10 (instr.): Hit
11    j    LOOP       # Rd 11 (instr.): Hit
```

```
3     beq  x5, x0, end# Rd 3 (instr.): Hit
4     lw   x1, 0[x4]  # Rd 4 (instr.) and 22 (M[22]): Hits
5     addi x4, x4, 1  # x4=23. Rd 5 (instr.): Hit
6     lw   x2, 0[x4]  # Rd 6 (instr.) and 23 (M[23]): Hits
7     mul  x2, x1, x2 # Rd 7 (instr.): Hit
8     sw   x2, 0[R6]  # Rd 8 (instr.): Hit
9                     # Wr 52 (M[52]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				

	Data			
	0	1	2	3
0	48	49	50	51
1	52	53	54	55
2				
3				

```
9     addi x6, x6, 1  # x6=53. Rd 9 (instr.): Hit
10    addi x5, x5, -1 # x5=1. Rd 10 (instr.): Hit
11    j    LOOP       # Rd 11 (instr.): Hit
```

```
3     beq  x5, x0, end# Rd 3 (instr.): Hit
4     lw   x1, 0(x4)  # Rd 4 (instr.): Hit
5                     # Rd 23 (M[23]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				

	Data			
	0	1	2	3
0	48	49	50	51
1	20	21	22	23
2				
3				

```
5     addi x4, x4, 1  # x4=24. Read word 5 (instr.): Hit
6     lw   x2, 0(x4)  # Read word 6 (instr.): Hit
7                     # Read word 24 (M[24]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				

	Data			
	0	1	2	3
0	48	49	50	51
1	20	21	22	23
2	24	25	26	27
3				

```
7       mul  x2, x1, x2 # Read word 7 (instr.): Hit
8       sw   x2, 0(x6)  # Read word 8 (instr.): Hit
9                       # Write word 53 (M[53]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				

	Data			
	0	1	2	3
0	48	49	50	51
1	52	53	54	55
2	24	25	26	27
3				

```
9       addi x6, x6, 1  # x6=53. Read word 9 (instr.): Hit
10      addi x5, x5, -1 # x5=1. Read word 10 (instr.): Hit
11      j    LOOP       # Read word 11 (instr.): Hit

3       beq  x5, x0, end# Read word 3 (instr.): Hit
```

b) The structure of the cache is given in the figure below:

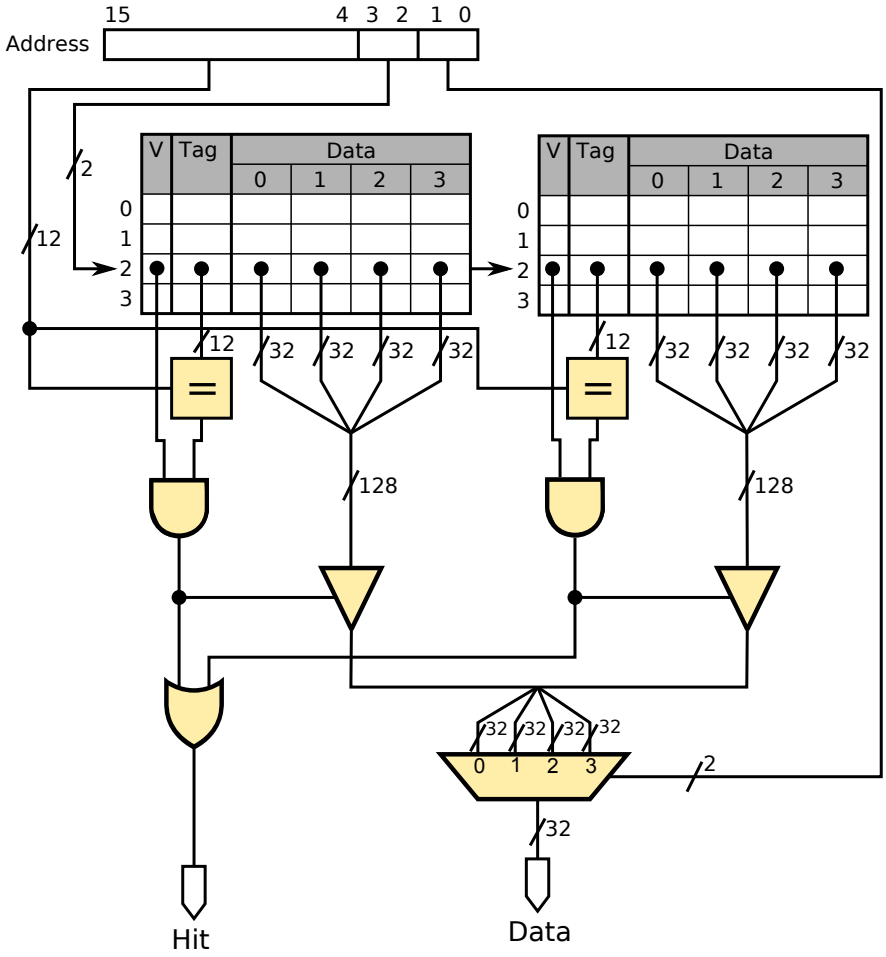


Figure 54: Structure of the cache

[Exercise 4] A Direct-Mapped Cache

Consider the following code for a RISC-V processor (word addressable, 16-bit addresses, 32-bit data, `x0` is always '0'):

```
0       lw   x4, A(x0)     # x4=M[A]
1       lw   x5, B(x0)     # x5=M[B]
2       lw   x6, C(x0)     # x6=M[C]
3  LOOP:beq  x5, x0, end   # if x5=x0 goto end
4       lw   x1, 0(x4)     # x1=M[x4]
5       addi x4, x4, 1     # x4=x4 + 1
6       lw   x2, 0(x4)     # x2=M[x4]
7       mul  x2, x1, x2    # x2=x1 * x2
8       sw   x2, 0(x6)     # M[x6]=x2
9       addi x6, x6, 1     # x6=x6 + 1
10      addi x5, x5, -1    # x5=x5 - 1
11      j    LOOP          # go to LOOP
12 end:
```

Suppose the initials values are:

A=16	M[16]=20
B=17	M[17]= 4
C=18	M[18]=50

The program is stored in memory starting at address 0.

a) Given an initially empty, direct-mapped cache memory with a size of 32 words and four words per block, indicate which accesses result in a hit and the ones that result in a miss. Show the state of the cache after the sequence of accesses (after the Fetch phase of instruction 12). For each entry (position) in the cache indicate the address of the stored information.

b) Draw the structure of the cache memory. Clearly indicate the use of each bit of the address by describing its format in detail.

[Solution 4] A Direct-Mapped Cache

a) The step-by-step execution of the program is given below:

```
0   lw x4, 16(x0) # R4=20. Read 0 (instr.) and 16 (M[16]): Misses
```

	Data			
	0	1	2	3
0	0	1	2	3
1				
2				
3				
4	16	17	18	19
5				
6				
7				

```
1   lw  x5, 17(x0)  # x5=4.  Rd 1 (instr.) and 17 (M[17]): Hits
2   lw  x6, 18(x0)  # x6=50. Rd 2 (instr.) and 18 (M[18]): Hits
3 LOOP:
4   beq x5, x0, end # x5=0.  Rd 3 (instr.): Hit
5   lw  x1, 0(x4)   # Rd 4 (instr.) and 20 (M[20]): Misses
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2				
3				
4	16	17	18	19
5	20	21	22	23
6				
7				

```
5       addi x4, x4, 1  # x4=21. Rd 5 (instr.): Hit
6       lw   x2, 0(x4)  # Rd 6 (instr.) and 21 (M[21]): Hits
7       mul  x2, x1, x2 # Rd 7 (instr.): Hit
8       sw   x2, 0(x6)  # Rd 8 (instr.); Wr 50 (M[50]): Misses

9       addi x6, x6, 1  # x6=51. Rd 9 (instr.): Hit
10      addi x5, x5, -1 # x5=3. Rd 10 (instr.): Hit
11      j    LOOP       # Rd 11 (instr.): Hit

3       beq  x5, x0, end# Rd 3 (instr.): Hit
4       lw   x1, 0(x4)  # Rd 4 (instr.) and 21 (M[21]): Hits
5       addi x4, x4, 1  # x4=22. Rd 5 (instr.): Hit
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				
4	48	49	50	51
5	20	21	22	23
6				
7				

```
6       lw   x2, 0(x4)   # Rd 6 (instr.) and 22 (M[22]): Hits
7       mul  x2, x1, x2  # Rd 7 (instr.): Hit
8       sw   x2, 0(x6)   # Rd 8 (instr.) and Wr 51 (M[51]): Hits
9       addi x6, x6, 1   # x6=52. Rd 9 (instr.): Hit
10      addi x5, x5, -1  # x5=2. Rd 10 (instr.): Hit
11      j    LOOP        # Rd 11 (instr.): Hit
```

```
3       beq  x5, x0, end# Rd 3 (instr.): Hit
4       lw   x1, 0[x4]   # Rd 4 (instr.) and 22 (M[22]): Hits
5       addi x4, x4, 1   # x4=23. Rd 5 (instr.): Hit
6       lw   x2, 0[x4]   # Rd 6 (instr.) and 23 (M[23]): Hits
7       mul  x2, x1, x2  # Rd 7 (instr.): Hit
8       sw   x2, 0[R6]   # Rd 8 (instr.): Hit
9                        # Wr 52 (M[52]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				
4	48	49	50	51
5	52	53	54	55
6				
7				

```
9       addi x6, x6, 1   # x6=53. Rd 9 (instr.): Hit
10      addi x5, x5, -1  # x5=1. Rd 10 (instr.): Hit
11      j    LOOP        # Rd 11 (instr.): Hit
```

```
3       beq  x5, x0, end# Rd 3 (instr.): Hit
4       lw   x1, 0(x4)   # Rd 4 (instr.): Hit
5                        # Rd 23 (M[23]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				
4	48	49	50	51
5	20	21	22	23
6				
7				

```
5       addi x4, x4, 1   # x4=24. Read word 5 (instr.): Hit
6       lw   x2, 0(x4)   # Read word 6 (instr.): Hit
7                        # Read word 24 (M[24]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				
4	48	49	50	51
5	20	21	22	23
6	24	25	26	27
7				

```
7       mul  x2, x1, x2 # Read word 7 (instr.): Hit
8       sw   x2, 0(x6)   # Read word 8 (instr.): Hit
9                        # Write word 53 (M[53]): Miss
```

	Data			
	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3				
4	48	49	50	51
5	52	53	54	55
6	24	25	26	27
7				

```
9       addi x6, x6, 1   # x6=53. Read word 9 (instr.): Hit
10      addi x5, x5, -1  # x5=1. Read word 10 (instr.): Hit
11      j    LOOP        # Read word 11 (instr.): Hit

3       beq  x5, x0, end# Read word 3 (instr.): Hit
```

b) The structure of the cache is given in the figure below:

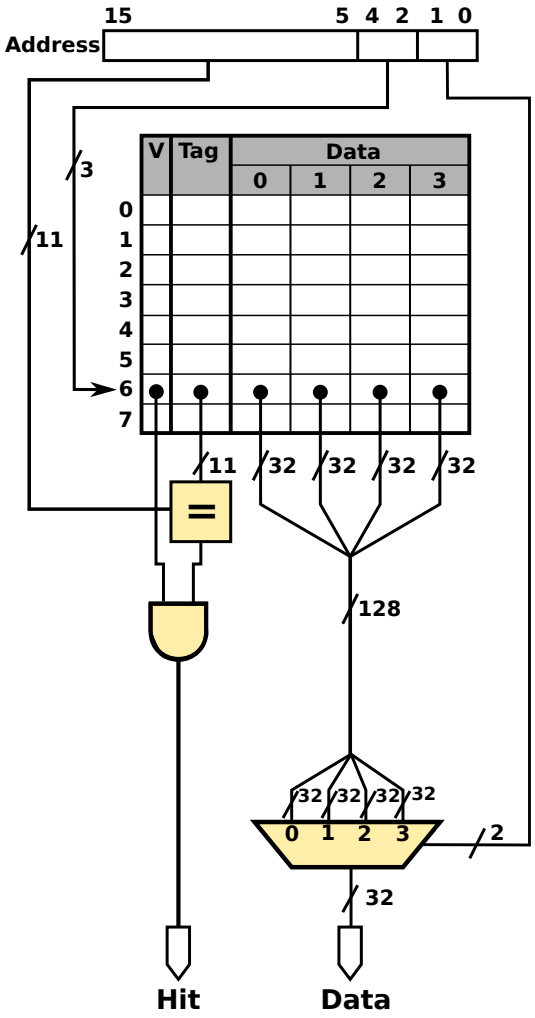


Figure 55: Structure of the cache

[Exercise 5] Cache Organizations

Consider a cache memory having 128 blocks of 4 words each, with 32-bit words, 32-bit addresses and word addressing.

a) Draw the structure of the following three caches, each time giving the use of the different bits of a word's address.

1. Direct-mapped cache
2. 4-way set-associative
3. Fully-associative

b) How many memory bits are necessary to implement the direct-mapped cache ?

c) If the direct-mapped cache is initially empty, indicate the hits and misses in the following memory access sequence.

What is the hit ratio for this access sequence ?

d) Give the contents of the cache at the end of the previous access sequence. Given that the memory is very large, only indicate words whose valid bit is set.

Address (decimal)	Address (hexadecimal)
1	0x0001
0	0x0000
127	0x007F
128	0x0080
129	0x0081
514	0x0202
512	0x0200
5125	0x1405
516	0x0204
514	0x0202
4	0x0004
3	0x0003
129	0x0081
9223	0x2407
1024	0x0400
5126	0x1406

[Solution 5] Cache Organizations

a) The structures of the data and instruction caches are given below:

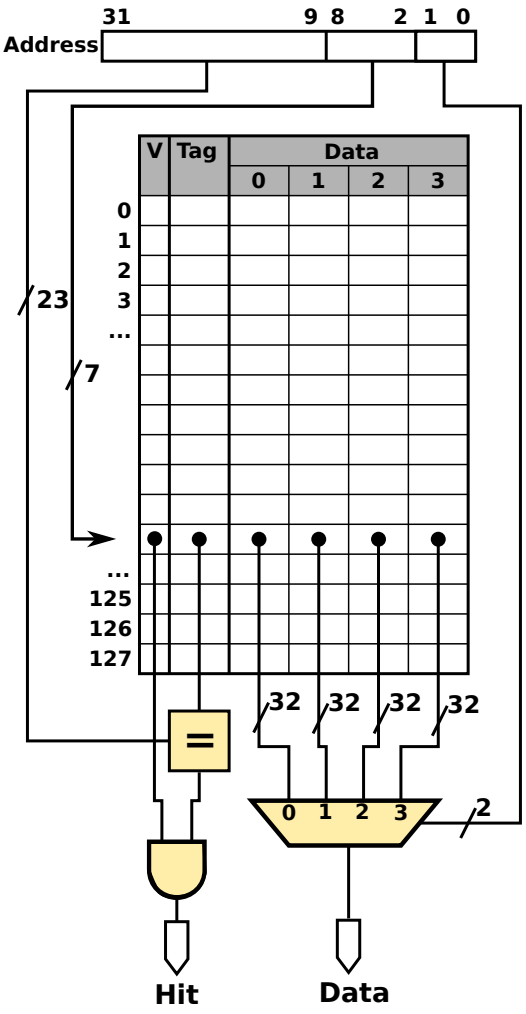


Figure 56: Structure of the data cache Direct Mapped

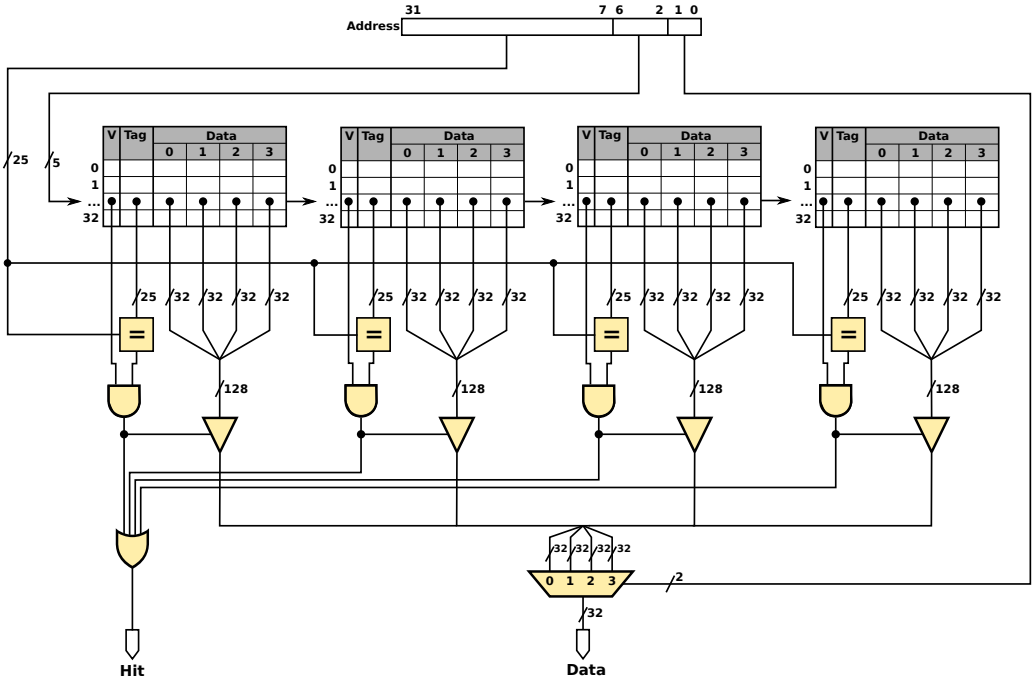


Figure 57: Structure of the data cache 4 Way Set associative

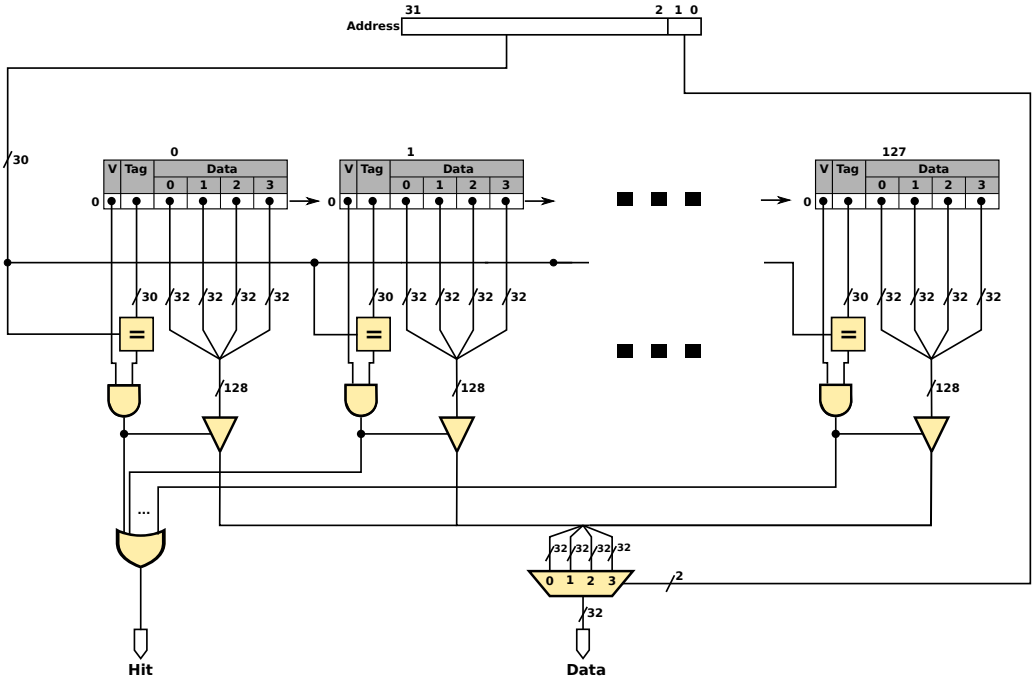


Figure 58: Structure of the data cache Fully Set associative

b) For the direct mapped cache, each line needs 1 valid bit, 23 tag bits and $4*32$ data bits (4 words of 32 bits). There are 128 lines in the cache, so the total is $128*(1+23+4*32) = 19456$ bits

c)

A	1	0	127	128	129	514	512	5125	516	514	4	3	129	9223	1024	5126
L	0	0	31	32	32	0	0	1	1	0	1	0	32	1	0	1
	M	H	M	M	H	M	H	M	M	H	M	M	H	M	M	M

Table 1: Miss (M) or Hit (H) depending on the accesses of address (A) on line (L) in the Direct Mapped Cache

The hit ratio is computed as $\frac{\#Hit}{\#Hit+\#Miss}$. In this case, the hit ratio is $\frac{5}{16} = 0.3125$.

d)

	Data			
	0	1	2	3
0	1024	1025	1026	1027
1	5124	5125	5126	5127
...				
31	124	125	126	127
32	128	129	130	131
...				

Figure 59: Cache after the memory accesses. Only cache lines 0, 1 and 31, 32 have the valid bit set

[Exercise 6] Direct-Mapped Caches

Consider a system with a data cache of 256 bytes and an instruction cache of 1 Kbytes. Both caches are direct mapped and have 16-byte lines. Instructions, data and addresses are all 32-bit wide and byte addressing is used in the system. Moreover, every write to the cache causes a write to main memory (write-through cache).

Now consider the following program given that `sp = 1200` and `a0 = 3` when the program is executed:

```
1000: main: addi sp, sp, -32
1004:       sw   ra, 20(sp)
1008:       sw   zero, 24(sp)
1012:       sw   zero, 28(sp)

1016: loop: lw   t1, 28(sp)
1020:       mul  t2, t1, t1
1024:       lw   t3, 24(sp)
1028:       add  t4, t3, t2
1032:       sw   t4, 24(sp)
1036:       addi t0, t1, 1
1040:       sw   t0, 28(sp)
1044:       blt  t0, a0, loop
1048:       lw   a1, 24(sp)
1052:       li   a2, 0
1056:       lw   ra, 20(sp)
1060:       addi sp, sp, 32
1064:       ret
```

a) Draw the structure of the cache. Clearly indicate the use of each bit of the address.

b) What is the hit ratio for each cache (data and instruction).

c) Indicate the state of each cache at the end of the program's execution.

d) Suppose the type of write policy is changed to copy-back (when a write results in a hit, only the cache is modified, but if it results in a miss the corresponding line is loaded from the main memory and modified in the cache). How would this change the answers to the preceding two questions **b)** and **c)**? Explain your answer.

e) Suppose the instruction cache and the data cache are replaced by a unified cache with a capacity of 2 Kbytes and write-through policy. How would this change the answers to the questions in points **b)** and **c)**? Explain your answer.

[Solution 6] Direct-Mapped Caches

a) The structures of the data and instruction caches are given below:

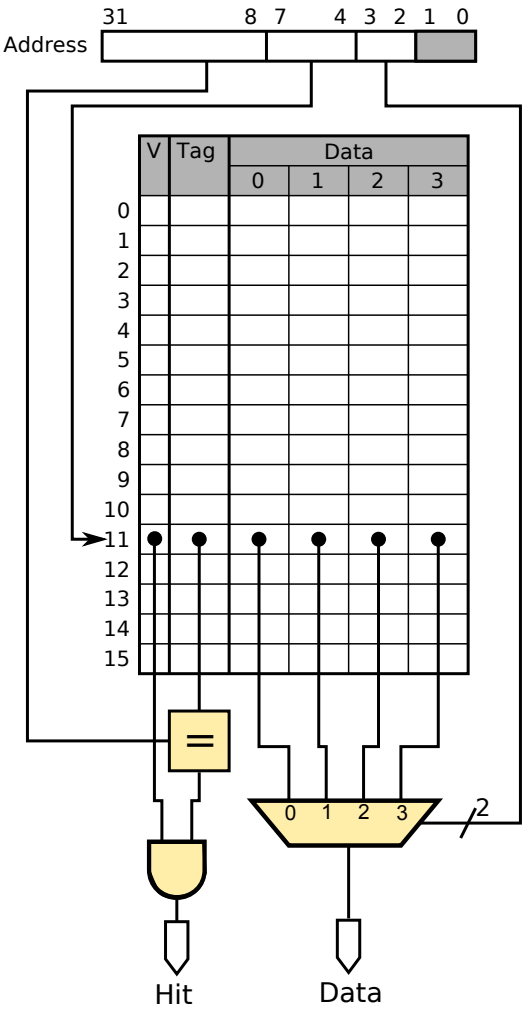


Figure 60: Structure of the data cache

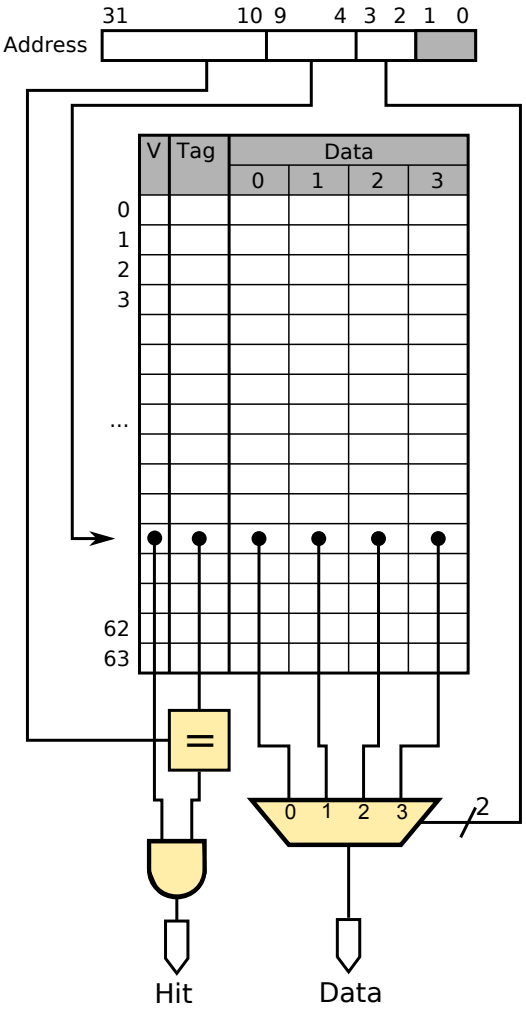


Figure 61: Structure of the instruction cache

b) The hit ratio can be deduced from the following table:

	Instruction	Instr. address decimal (hexa)	Instruction Cache access	Data address decimal (hexa)	Data cache access
main:	`addi sp, sp, -32`	1000 (0x3E8)	M		
	`sw ra, 20(sp)`	1004 (0x3EC)	H	1188 (0x4A4)	M
	`sw zero, 24(sp)`	1008 (0x3F0)	M	1192 (0x4A8)	H
	`sw zero, 28(sp)`	1012 (0x3F4)	H	1196 (0x4AC)	H
loop:	`lw t1, 28(sp)`	1016 (0x3F8)	H	1196 (0x4AC)	H
	`mul t2, t1, t1`	1020 (0x3FC)	H		
	`lw t3, 24(sp)`	1024 (0x400)	M	1192 (0x4A8)	H
	`add t4, t3, t2`	1028 (0x404)	H		
	`sw t4, 24(sp)`	1032 (0x408)	H	1192 (0x4A8)	H
	`addi t0, t1, 1`	1036 (0x40C)	H		
	`sw t0, 28(sp)`	1040 (0x410)	M	1196 (0x4AC)	H
	`blt t0, a0, loop`	1044 (0x414)	H		
loop:	`lw t1, 28(sp)`	1016 (0x3F8)	H	1196 (0x4AC)	H
	`mul t2, t1, t1`	1020 (0x3FC)	H		
	`lw t3, 24(sp)`	1024 (0x400)	H	1192 (0x4A8)	H
	`add t4, t3, t2`	1028 (0x404)	H		
	`sw t4, 24(sp)`	1032 (0x408)	H	1192 (0x4A8)	H
	`addi t0, t1, 1`	1036 (0x40C)	H		
	`sw t0, 28(sp)`	1040 (0x410)	H	1196 (0x4AC)	H
	`blt t0, a0, loop`	1044 (0x414)	H		
loop:	`lw t1, 28(sp)`	1016 (0x3F8)	H	1196 (0x4AC)	H
	`mul t2, t1, t1`	1020 (0x3FC)	H		
	`lw t3, 24(sp)`	1024 (0x400)	H	1192 (0x4A8)	H
	`add t4, t3, t2`	1028 (0x404)	H		
	`sw t4, 24(sp)`	1032 (0x408)	H	1192 (0x4A8)	H
	`addi t0, t1, 1`	1036 (0x40C)	H		
	`sw t0, 28(sp)`	1040 (0x410)	H	1196 (0x4AC)	H
	`blt t0, a0, loop`	1044 (0x414)	H		
	`lw a1, 24(sp)`	1048 (0x418)	H	1192 (0x4A8)	H
	`li a2, 0`	1052 (0x41C)	H		
	`lw ra, 20(sp)`	1056 (0x420)	M	1188 (0x4A4)	H
	`addi sp, sp, 32`	1060 (0x424)	H		
	`ret`	1064 (0x428)	H		

The instruction cache hit ratio is $\frac{28}{33} = 0.85$. The data cache hit ratio is $\frac{16}{17} = 0.94$.

c) The state of each cache is given in a separate table.

Cache line	**Data**			
0x00	0x400	0x404	0x408	0x40C
0x01	0x410	0x414	0x418	0x41C
0x02	0x420	0x424	0x428	0x42C
...				
0x3E	0x3E0	0x3E4	0x3E8	0x3EC
0x3F	0x3F0	0x3F4	0x3F8	0x3FC

Table 2: Instruction cache

Cache line	**Data**			
...				
0x0A	0x4A0	0x4A4	0x4A8	0x4AC
...				

Table 3: Data cache

d) If the writing policy changes from write-through to copy-back, the hit rate and the final state of the cache remain unchanged.

When there is a miss, both policies are the same. When there is a hit, the hit rate does not depend on the update method of the main memory. For the copy-back, the update of the main memory is delayed in a transparent way.

e) The hit rate will remain unchanged because accesses to the instructions and data occur on different lines of the cache. The unification of the cache can affect the hit rate if there is an overlap of instruction area with the data area.

Cache line	**Data**			
...				
0x3E	0x3E0	0x3E4	0x3E8	0x3EC
0x3F	0x3F0	0x3F4	0x3F8	0x3FC
0x40	0x400	0x404	0x408	0x40C
0x41	0x410	0x414	0x418	0x41C
0x42	0x420	0x424	0x428	0x42C
...				
0x4A	4A0	0x4A4	0x4A8	0x4AC
...				
0x7F				

Table 4: Unified cache

[Exercise 7] Code and Cache

Consider a system with a data cache of 32 Kbytes and an instruction cache of 16 Kbytes. Both caches are direct-mapped and have 128-byte (cache) lines. The system uses byte addressing using 32-bit addresses and instructions are also 32-bit. Integers are encoded on 32 bits.

Now consider the following program:

```
int a[128], b[128], c[128], i;
main () {
  for (i=0; i<128; i++) {
    a[i] = b[i] + c[i];
  }
}
```

a) Draw the structure of the two caches. Clearly indicate the use of each bit of the address by describing its format in detail.

b) Could the hit ratio of either cache memory be influenced by the address chosen by the system to store the program's instructions in memory? Explain your answer supposing that the program is the only one running on the system.

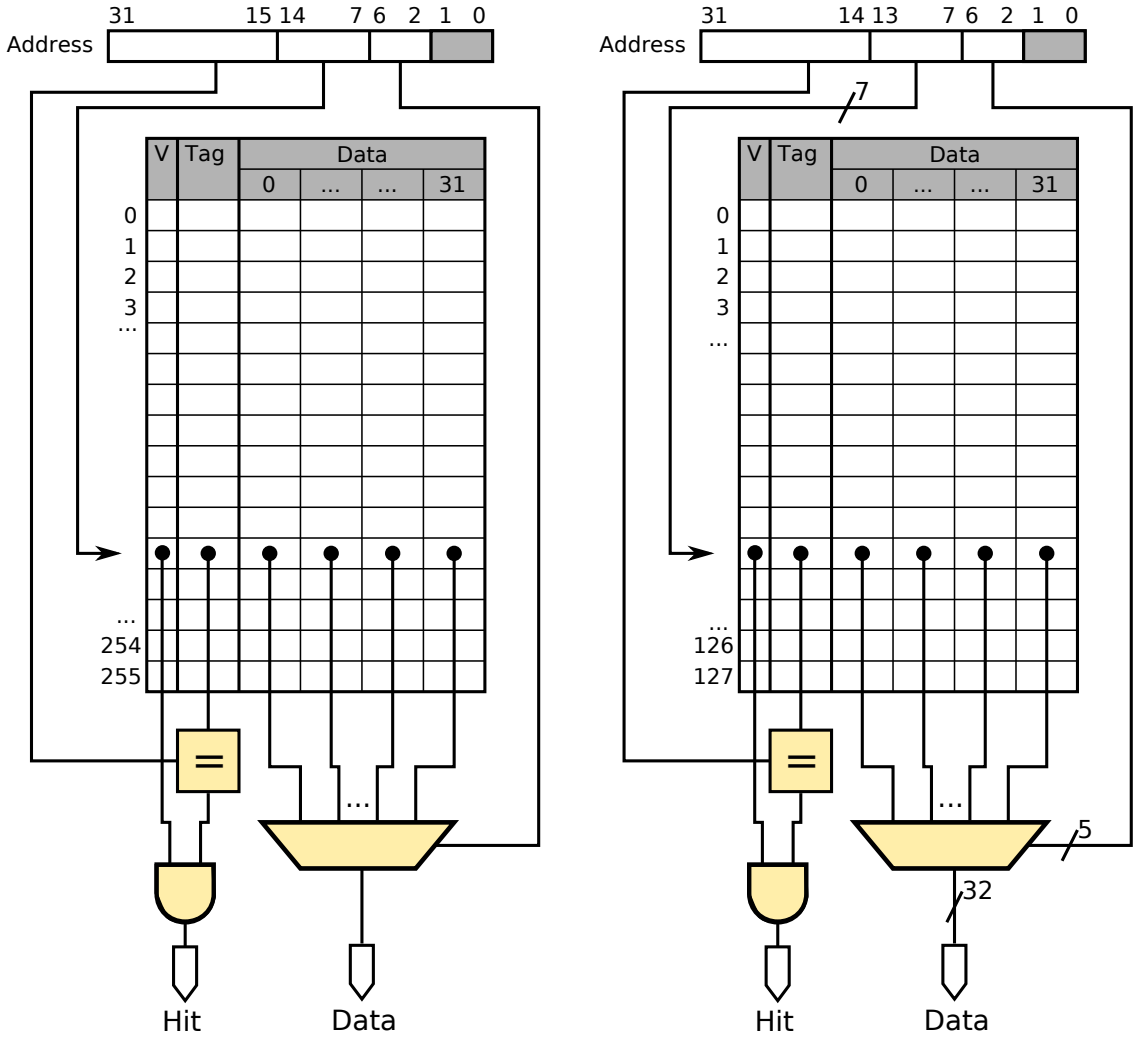
c) What restrictions regarding the placement in memory of arrays `a`, `b` and `c` would allow minimizing misses in the data cache ? Ignore integer `i` in this question and the following ones. Give two placement examples to illustrate two extreme situations (minimum and maximum number of misses).

d) Suppose that the cache's level of associativity is increased, i.e. the number of ways in the cache is increased. What is the data cache's highest level of associativity in which misses still occur?

e) Suppose that the separate instruction and data caches are replaced by a unified direct mapped 32 Kbyte cache. How would this modify the answers to questions **b)** and **c)**? Give two placing examples.

[Solution 7] Code and Cache

a) The structures of the data and instruction caches are given below:



(a) Structure of the data cache.

(b) Structure of the instruction cache.

Figure 62: Structure of the data cache (left) and the instruction cache (right).

b) The location of the program in memory does not affect the hit rate of the data cache, as both caches are separate.

The location of the program in memory does not affect the hit rate of the instruction cache either, since no other program is running, once the program is loaded in the

cache there will only be hits (We discard the potential miss resulting from the program overlapping two cache blocks, given the long loop).

c) To minimise the miss rate in the data cache, the three arrays must be mapped on three different blocks of the cache.

The addresses of the arrays must be chosen so as to avoid any overlapping in the cache.

Figure 63 shows the address format (structure) for the data cache.

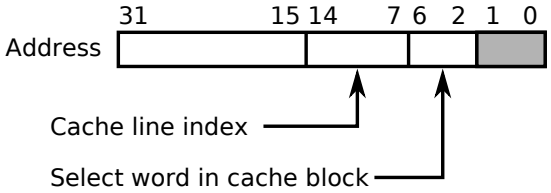


Figure 63: Structure of the address for the data cache

Address bits 7 to 14 select a line in the cache. To minimise the miss rate, these 8 bits must be different for each element of the array. Thus for each physical address of arrays `a`, `b` and `c` we must have address$_a$[7;14]≠address$_b$[7;14]≠address$_c$[7;14].

Figure 64 shows two different placings in memory for arrays `a`, `b` and `c`. The data cache has a size of 32 Kbytes, so addresses that are distant by 32K are placed in the same location in the cache, thus if the arrays are placed with such a difference in their addresses as in figure 64 (on the right), the miss rate will be maximal. In order to have a minimal miss rate, the tables must be placed as shown on the left side of figure 64.

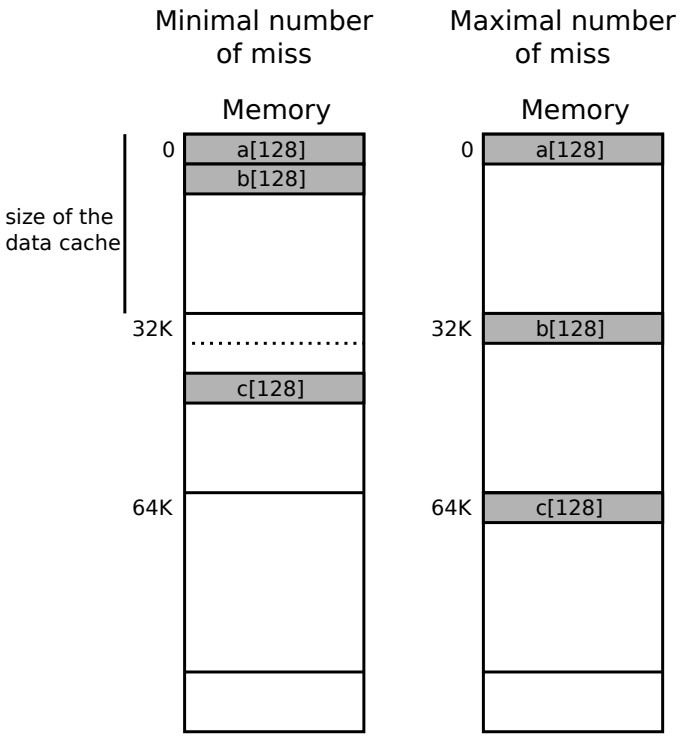


Figure 64: Placing arrays a, b and c

d) With an associativity level of 3 there can be no more misses because even in the worst case (all arrays mapped on the same cache line) the associativity allows avoiding misses.

With an associativity level of 2, it is still possible to have some misses, thus the highest associativity level where misses can still occur is 2.

e) The answer to **b)** is modified as follows: If the cache in unified, the hit rate can be affected if the program and one of the arrays overlap in the cache. In order to avoid this, the following conditions on the addresses must be fulfilled:

$$address_{instr}[7;14] \neq address_{a}[7;14]$$
$$address_{instr}[7;14] \neq address_{b}[7;14]$$
$$address_{instr}[7;14] \neq address_{c}[7;14]$$

For all possible values of the physical addresses of the instructions and arrays `a`, `b` and

c.

The answer to **c)** is modified as follows: The answer remains the same, but in addition the program must be mapped onto a different block in the cache than the ones used for the arrays. Thus the condition becomes:

$$address_{instr}[7;14] \neq address_a[7;14] \neq address_b[7;14] \neq address_c[7;14]$$

for every possible value of the physical addresses of the instructions and the arrays a, b and c.

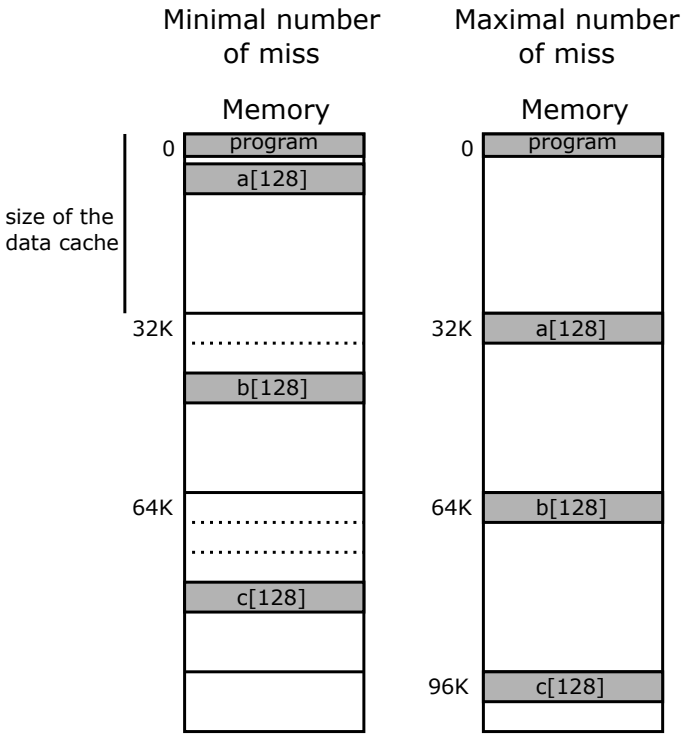


Figure 65: Placing arrays a, b and c

[Exercise 8] Cache Hierarchies

We consider a memory access system as shown in figure 66.

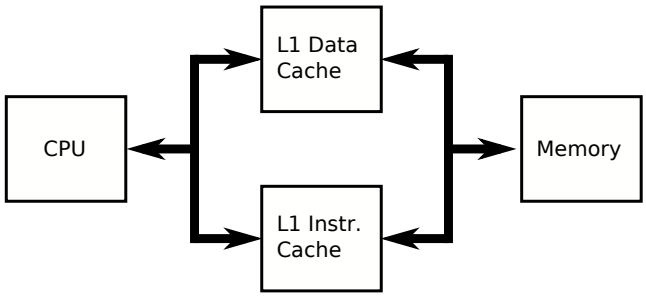


Figure 66: Organization of the memory

The L1 caches are organized as follows:

- Physical address on 32 bits. Byte addressing.
- Instructions and data are separated.
- 4 Kbytes for the data and 4 Kbytes for the instructions.
- Blocks of 32 bytes.
- Each cache has 2 ways. The replacement policy is LRU (Least Recently Used).
- Write-back is used.

a) Draw the diagram of each cache specifying the use of each bit of the physical address.

b) The average access time of the memory is:

$$T_{mem} = T_{h1} + P_{m1} \cdot T_{m1}$$

Where T_{h1} is the access time to the cache when there is a hit (hit time), P_{m1} is the miss rate and T_{m1} is the miss penalty. Referring to the diagram drawn in the previous question indicate what is T_{h1}? What is the time T_{m1} when a read miss occurs? Explain why T_{h1} should ideally be as small as possible.

c) Given that the cache is physically addressed, virtual addresses must be translated into physical ones before accessing the cache. Draw a diagram showing how the 32 bits of the virtual address are used when the latter is translated into a physical address in the case of 1 Kbyte pages as well as 4 Kbyte pages. Do you expect a better performance in one of the cases? Explain your answer.

d) What change do you expect on T_{h1}, P_{m1} T_{m1} if the associativity of the cache or the size of its blocks are increased without changing the total size (of the cache). Indicate whether it will result in an increase, a decrease or in no change, or whether the change is unpredictable as it depends on other parameters. Explain your answers.

e) In order to decrease the miss penalty without affecting the hit time, we propose adding an additional cache between the L1 caches (instruction and data) and the memory:

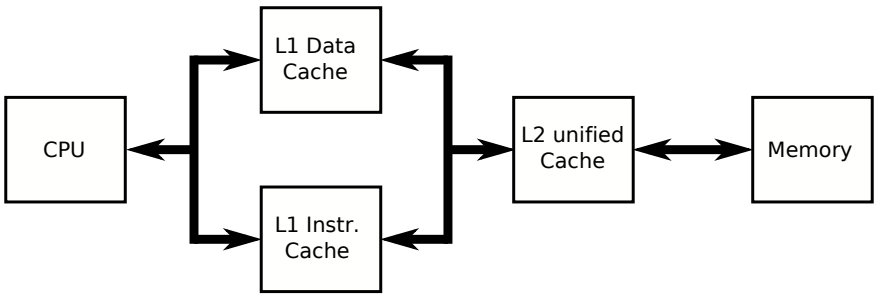


Figure 67: Organization of the memory with unified L2

The L2 cache has the following characteristics:

- Physical byte addressing.
- Size of 512 Kbyte, unified for instruction and data.
- 64-byte blocks.
- Direct-mapped.

Draw the diagram of the cache indicating the use of the bits of the physical address.

f) Indicate if the following accesses result in hits or misses in each of the caches L1 and L2 (the caches are initially empty). Calculate the hit-and-miss rates of each cache for the following access sequence (`I` for instructions and `D` for data):

```
I[0xffff 0000], I[0xffff 0004], I[0xffff 0008], I[0xffff 000c],
D[0x0000 0000], I[0xffff 0010], I[0xffff 0014], D[0x1000 0000],
I[0xffff 0018], D[0x2000 0000], I[0xffff 0020], I[0xffff 0024],
I[0xffff 0010], I[0xffff 0014], D[0x1000 0020], I[0xffff 0018],
D[0x2000 0004], I[0xffff 0020], I[0xffff 0024], I[0xffff 0010],
I[0xffff 0014], D[0x1000 0000], I[0xffff 0018], D[0x2000 0008]
```

g) Express the mean memory access time as a function of parameters T_{h1}, P_{m1}, T_{h2}, P_{m2} and T_{m2}. Parameters T_{h2}, P_{m2} and T_{m2} are defined the same way as T_{h1}, P_{m1}, T_{m1} but for the L2 cache. Compute it (mean memory access time) for the access sequence in the preceding question with $T_{h1} = 1$ cycle, $T_{h2} = 5$ cycles and $T_{m2} = 100$ cycles.

[Solution 8] Cache Hierarchies

a) The structure of the cache is shown in the diagram below:

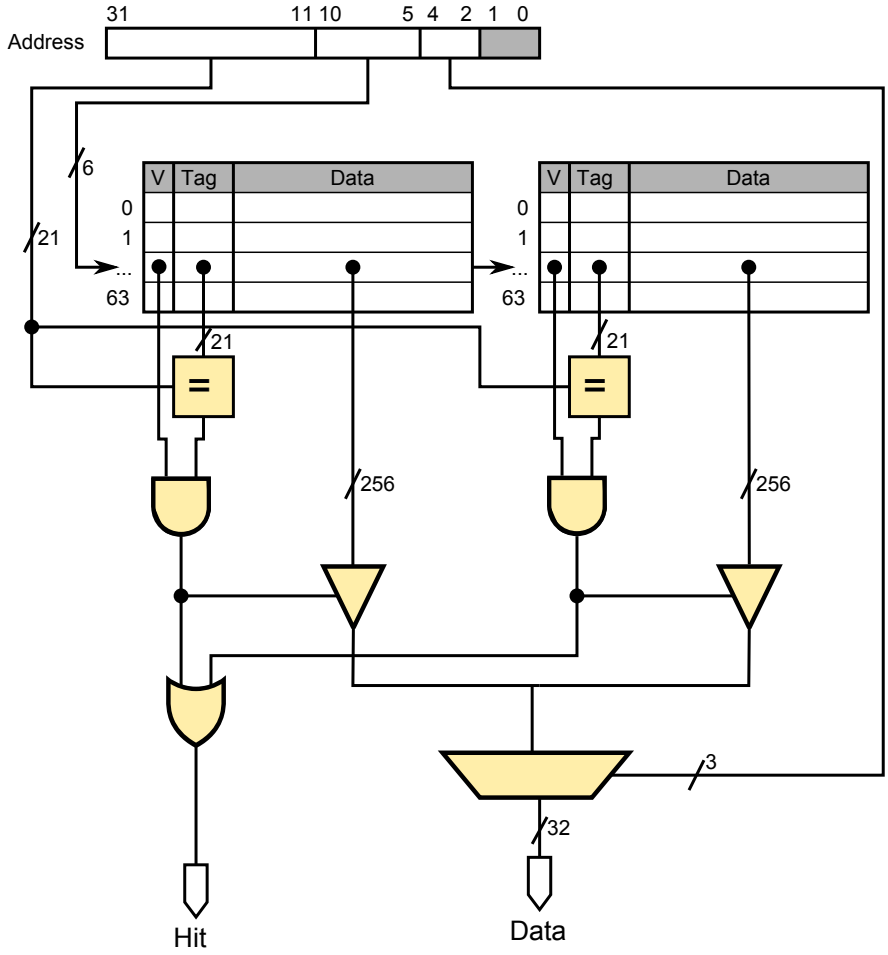


Figure 68: Structure of each cache

b) i) T_{h1} is the sum of the critical path of the combinatorial circuit that verifies the equality of the address and cache data tags and the time required to select (multiplex) the output word out of the 8 words of a cache line.

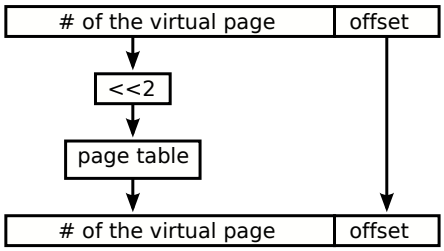
ii) T_{m1} is the sum of:

1. the time required to access the memory in order to fetch the block of the missing word and write this block in the cache.
2. the time required to write in memory (or in a write buffer) the block that will be taken out of the cache to resolve a miss.

Timing 2 above is not taken into consideration is the block taken out of the cache has not been modified (e.g. instructions).

iii) A processor's clock cycle is necessarily larger than T_{h1}. Consequently, we want T_{h1} to be as small as possible for performance reasons (i.e. to reach higher processor clock frequencies).

c) i) Pages of 1 Kbyte have an offset field of 10 bits, while 4 Kbyte pages need 12 bits for the offset field.



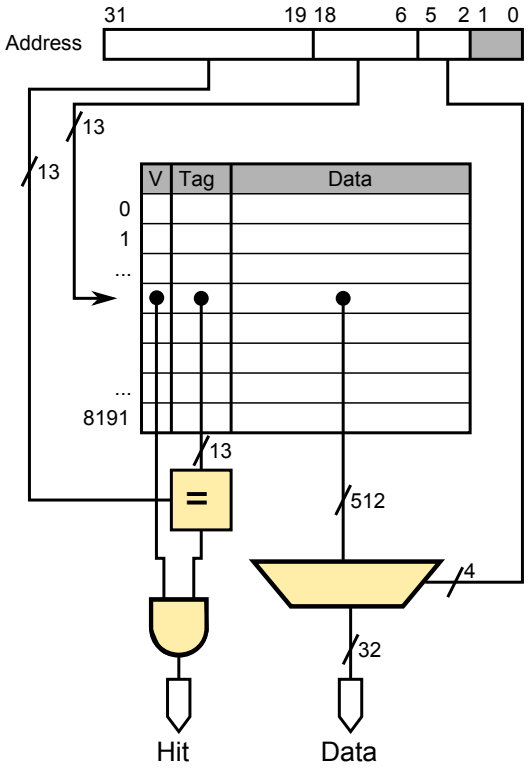
ii) In order to access the cache in parallel to the translation of the virtual address, the offset field of the virtual address must contain the 11 least significant bits of the physical address that are required to access the cache as shown in the diagram drawn in point **a)**. For 1 Kbyte pages, this is not possible as the offset field in the virtual address has only 10 bits. In the case of 4 Kbyte caches the virtual address has 12 bits, and can thus contain the 11 bits of the physical address that are required to access the cache while the translation is performed.

d) If the associativity of the cache is increased, T_{h1} remains unchanged. More tri-state buffers, comparators, and-gates are needed horizontally. The size of the and-gate is increased as it needs more entries. The size of the final multiplexer remains unchanged as it is only affected by the block size. Those changes do not significantly affect the hit time as everything is done in parallel. Only the size of the and-gate affects the critical path of the hit. Increasing the associativity decreases the number of misses due to conflicts. There are more slots available if multiple addresses are mapped on the same cache line. Misses due to the size limitation of the caches and compulsory (unavoidable) misses remain unchanged. Thus, P_{m1} decreases, while T_{m1} is not influenced by the associativity of the cache.

If we increase the block size, T_{h1} increases as well as the critical path of the combinatorial circuit that provides the desired block increases. The multiplexer which selects the block needs to be larger. Misses due to conflicts increase (if the size of the cache is unchanged). There are less slots if two addresses are mapped to the same cache line. Misses due to the size limitation of the cache and compulsory misses remain

unchanged. P_{m1} decreases during sequential (consecutive) accesses. For other access types, it is not possible to predict the effect on P_{m1}. T_{m1} increases because the time required to access and transfer the blocks from the main memory increases with the size of the blocks.

e) The structure of the cache is given in the diagram below.



f) Hits and misses are indicated for each access of the sequence.

Access	L1 Index I/D	L1 H/M	L2 Index	L2 H/M
I[0xFFFF 0000]	0x00	M	0x1C00	M
I[0xFFFF 0004]	0x00	H	0x1C00	
I[0xFFFF 0008]	0x00	H	0x1C00	
I[0xFFFF 000C]	0x00	H	0x1C00	
D[0x0000 0000]	0x00	M	0x0000	M
I[0xFFFF 0010]	0x00	H	0x1C00	
I[0xFFFF 0014]	0x00	H	0x1C00	
D[0x1000 0000]	0x00	M	0x0000	M
I[0xFFFF 0018]	0x00	H	0x1C00	
D[0x2000 0000]	0x00	M	0x0000	M
I[0xFFFF 0020]	0x01	M	0x1C00	H
I[0xFFFF 0024]	0x01	H	0x1C00	
I[0xFFFF 0010]	0x00	H	0x1C00	
I[0xFFFF 0014]	0x00	H	0x1C00	
D[0x1000 0020]	0x01	M	0x0000	M
I[0xFFFF 0018]	0x00	H	0x1C00	
D[0x2000 0004]	0x00	H	0x0000	
I[0xFFFF 0020]	0x01	H	0x1C00	
I[0xFFFF 0024]	0x01	H	0x1C00	
I[0xFFFF 0010]	0x00	H	0x1C00	
I[0xFFFF 0014]	0x00	H	0x1C00	
D[0x1000 0000]	0x00	H	0x0000	
I[0xFFFF 0018]	0x00	H	0x1C00	
D[0x2000 0008]	0x01	H	0x0000	

For the L1 instruction cache, there are 15 hits and 2 misses. The hit ratio is $\frac{15}{17}$ and the miss rate is $\frac{2}{17}$. For the L1 data cache, there are 3 hits and 4 misses. The hit ratio is $\frac{3}{7}$ and the miss rate is $\frac{4}{7}$. For the L2 cache, there is 1 hit and 5 misses. The hit ratio is $\frac{1}{6}$ and the miss rate is $\frac{5}{6}$.

g) The mean memory access time can be obtained by replacing T_{m1} by the mean time to access the memory from the L2 cache, i.e. $T_{h2} + P_{m2} \cdot T_{m2}$. Thus, the mean access time can be expressed as follows:

$$T_{mem} = T_{h1} + P_{m1} \cdot (T_{h2} + P_{m2} \cdot T_{m2})$$

[Exercise 9] Set Associative Caches

Consider a byte-addressable 2-way set associative cache memory of 1 Kbyte, with an LRU replacement policy, 32-bit words and two words per block.

a) Draw a diagram of the cache indicating the use of each bit of the address. Explain precisely what determines the position of a word in the cache.

b) Given the following access sequence, compute the hit and miss rates of the cache. Explain the obtained results.

```
0x18DE 0000, 0x18DE 0004, 0x2000 01FC, 0x2000 01F8,
0x18DE 0008, 0x18DE 000C, 0x2000 01F4, 0x2000 01F0,
0x18DE 0010, 0x18DE 0014, 0x2000 01EC, 0x2000 01E8,
     ...
0x18DE 01F8, 0x18DE 01FC, 0x2000 0004, 0x2000 0000,
0x18DE 0000, 0x18DE 0004, 0x2000 0000, 0x2000 0004,
     ...
0x18DE 01F8, 0x18DE 01FC, 0x2000 01F8, 0x2000 01FC
```

c) Is it possible to improve (reduce) the miss rate for the aforementioned access sequence by increasing the number of ways from 2 to 4 while keeping the same cache size ? Explain.

d) If we increase the block size to 4 words instead of 2 while keeping the same cache size, what will be the new miss rate for the (same) access sequence ? Explain and show the contents of the last line of both cache ways.

[Solution 9] Set Associative Caches

a) The 2 least significant bits are not used (byte addressing). Only one bit is necessary to address a word in a block. The cache contains 256 words. The number of lines for each way of the cache is $\frac{256}{(2 \cdot 2)} = 64$ and thus 6 bits are needed for the index as shown in Figure 69.

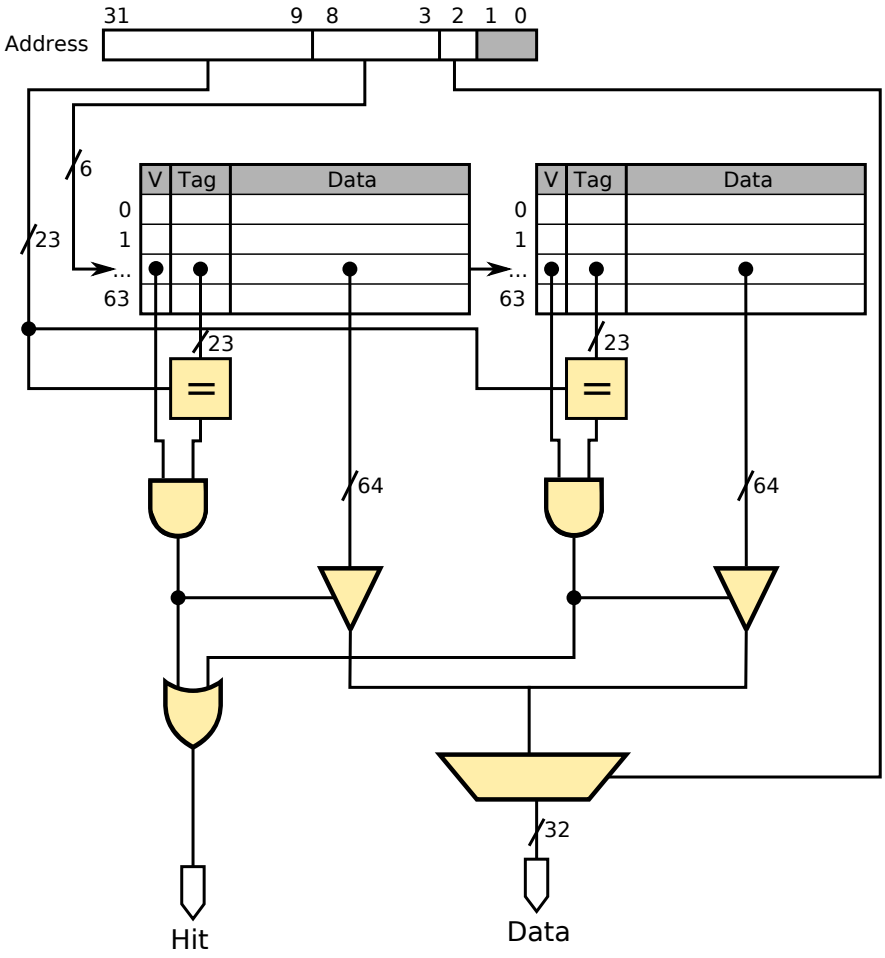


Figure 69: Structure of the cache

b) Figure 70 shows how the cache is filled after the first four accesses according to the address format introduced in figure 69. Accesses `0x18DE 0000` and `0x18DE 0004` are placed in the first block of one of the cache's two ways. The first access results in a miss while the second results in a hit. Accesses `0x2000 01FC` and `0x2000 01F8` are placed in the last block of the same way and the first one results in a miss while the second in a hit.

If we carefully observe the access sequence we can see that we can distinguish two

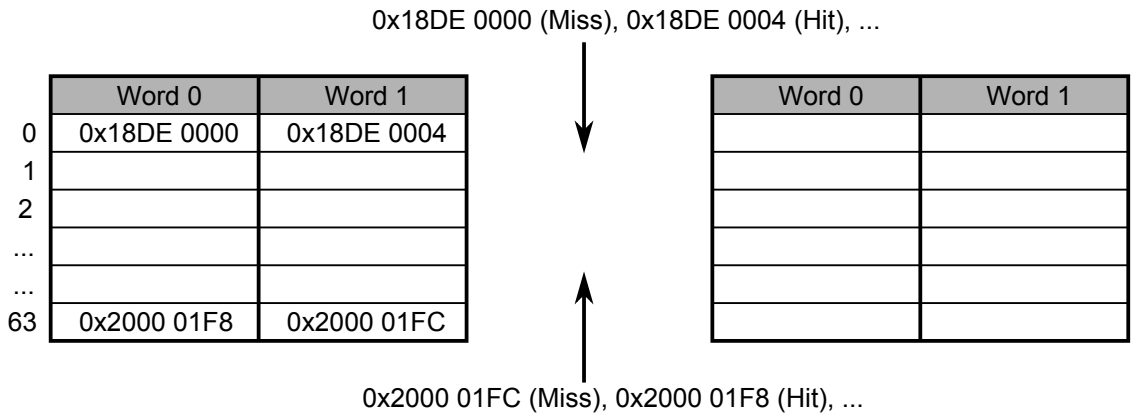


Figure 70: Cache state after first accesses

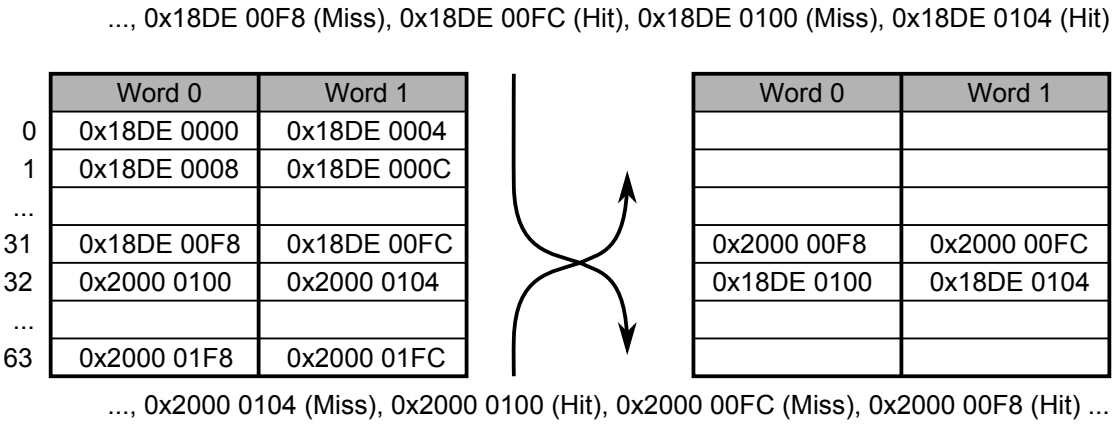


Figure 71: Cache state after first way is full

sequences. The first one is an ascending then descending access sequence starting at `0x18DE 0000`, culminating at `0x18DE 01FC` then back to `0x18DE 0000`. The other sequence has an opposite pattern, starting at `0x2000 01FC` reaching its lowest point at `0x2000 0000` then rising back to `0x2000 01FC`.

As a result, the first cache way will be filled at accesses `0x18de 00f8`, `0x18de 00fc` and `0x2000 0100`, `0x2000 0104`, therefore the following accesses will start filling the second way as shown in figure 71.

Both cache ways will be full halfway through the sequence at accesses `0x18DE 01FC` and `0x2000 0000`, which are the highest and lowest points of the first and second inner sequences, respectively. We can observe that up to this point the first access of

	Word 0	Word 1
0	0x18DE 0000	0x18DE 0004
1	0x18DE 0008	0x18DE 000C
...		
31	0x18DE 00F8	0x18DE 00FC
32	0x2000 0100	0x2000 0104
...		
63	0x2000 01F8	0x2000 01FC

Word 0	Word 1
0x2000 0000	0x2000 0004
0x2000 00F8	0x2000 00FC
0x18DE 0100	0x18DE 0104
0x18DE 01F8	0x18DE 01FC

Figure 72: Cache state halfway through the accesses

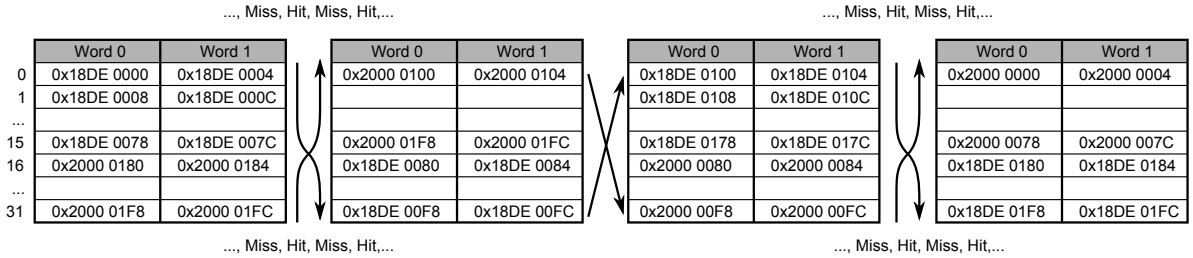


Figure 73: Cache state halfway with 4 word blocks

each inner sequence results in a miss while the second results in a hit, therefore the miss rate for the first half of the sequence is 50%.

The second half of the accesses is a mere repetition of the previous ones only in the reverse order, therefore all accesses result in a hit and the miss rate is 0%.

The overall miss rate for the whole sequence is thus 25%.

c) Figure 72 shows the state of the 4-way cache after the first half of the accesses.

The miss rate is still 50% because blocks contain two words. For the second half of the accesses, the miss rate is also 0%. Thus the overall miss rate for both caches is the same: 25%.

d) By increasing the block size to 4 words while keeping the same total cache size 9 bits are still needed to determine the position of a word in the cache. Consequently, the first half of the access sequence will fill the cache as shown in figure 73.

In this case however, the miss rate for the first half of the access sequence is only 25% as only one access out of four generates a miss. The second half, as before, generates no miss. Therefore the overall miss rate of this cache for this access sequence is 12.5%.

[Exercise 10] Cache Organizations

Consider the following data caches with a size of 128 words:

1. A 2-way set associative cache with an LRU (Least Recently Used) replacement policy and a block size of 4 words.
2. A fully associative cache with an LRU (Least Recently Used) replacement policy and a block size of 2 words.
3. A direct-mapped cache with a block size of 4 words.

Caches are byte addressed using 32-bit addresses and data is 32-bit wide.

a) Draw a diagram of the three caches indicating the use of each bit of the address.

b) Given the following access sequence, which cache will have the smallest miss rate? Explain.

```
0x8000 0000, 0x8000 0200, 0x8000 0004, 0x8000 0204,
    ...,     0x8000 03FC, 0x8000 05FC
```

c) Can the performance of the direct-mapped cache be enhanced by increasing the block size from 4 to 8 words while keeping the same cache size ? Qualitatively compare the performance of these two direct-mapped caches ?

[Solution 10] Cache Organizations

a) For all caches the two least significant bits are not used for addressing the cache as it is byte addressed.

i) The structure of the first 2-way set associative cache is given in figure 74. Blocks contain 4 words, hence two bits (2 to 3) are required to address a word inside a block. Each cache way contains 16 lines ($128 \div (2 \times 4)$). The index thus needs 4 bits (4 to 7). The rest (bits 8 to 31) determine the tag. Since the cache is 2-way set associative, two comparisons are needed to determine a hit.

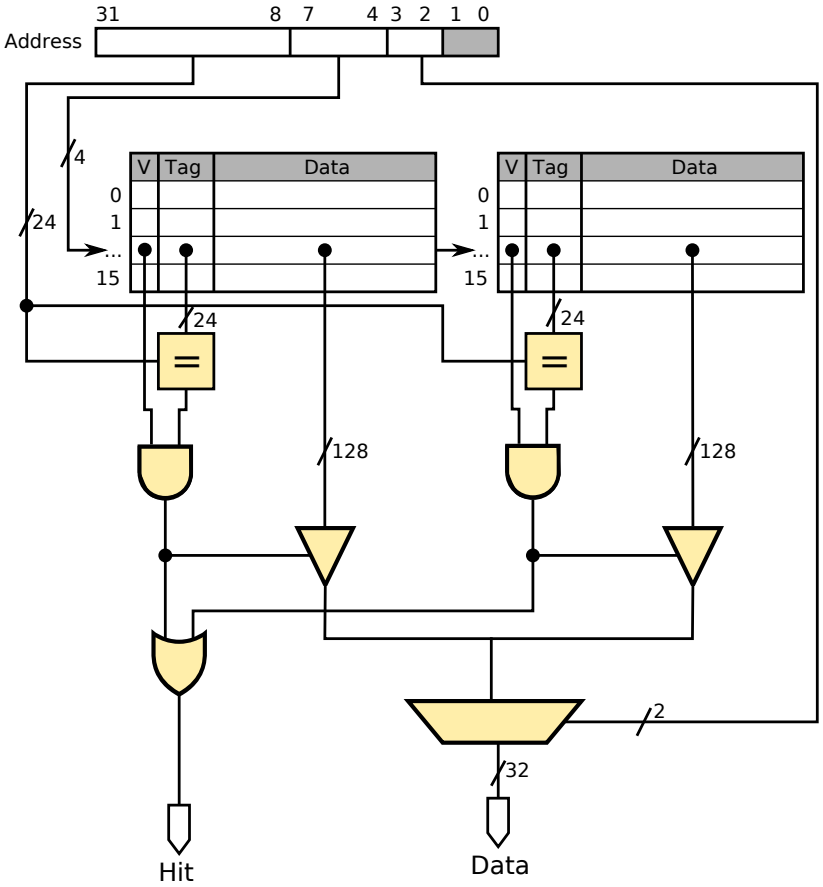


Figure 74: Structure of the 2-way set associative cache

ii) In this fully associative cache, a single bit is required to address a word in a line, bit 2 is used to this end. The structure of the cache along with the use of the address bits is given in figure 75. The cache has 64 lines ($128 \div 2$). Since it is fully associative, the tag consists of all the remaining bits (3 to 31). To determine if a hit has occurred requires 64 comparisons.

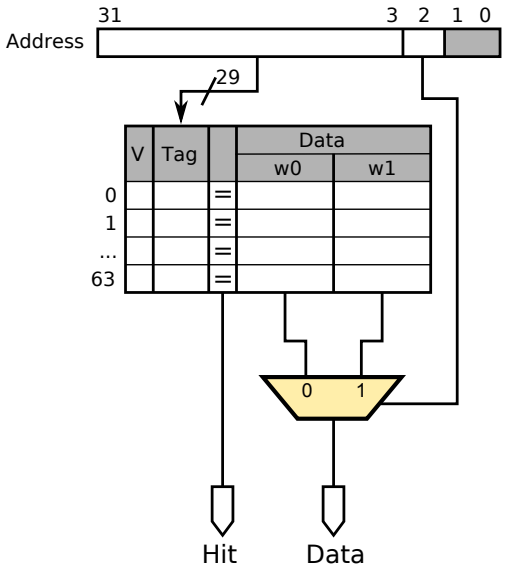


Figure 75: Structure of the fully associative cache

iii) The structure of the direct-mapped cache is given in figure 76 along with the use of the address bits. As we can see, given the block size of 4 words, 2 bits are required to address a word within a block, bits 2 and 3 are used to this end. The cache has 32 lines ($128 \div 4$). Five bits (4 to 8) are used for the index, and the tag consists of the remaining bits (9 to 31). A single comparison is needed to determine the occurrence of a hit.

b) Looking more closely at the access sequence we can see that it actually consists of 2 interlaced sequences of 256 consecutive accesses. The first sequence consists of accesses `0x8000 0000`, `0x8000 0004`, ..., `0x8000 03FC`. Accesses of the second sequence are distant from the first sequence accesses by `0x0000 0200` and the second sequence is thus `0x8000 0200`, `0x8000 0204`, ..., `0x8000 05FC`.

i) In the 2-way set associative cache, accesses of the second sequence will placed on the same line as the ones of the first sequence. However, given the 2 ways and the associativity of the cache, the accesses of both sequences do not interfere with one another. The block size of 4 words also contributes to fewer misses, only addresses ending with `0` result in a miss. Thus the miss rate of the 2-way set associative cache is 25%.

ii) In the fully associative cache accesses of both sequences do not interfere with one another due thanks to the associativity. Given that blocks are two words in size, accesses with addresses ending with `0` and `8` result in a miss, while those ending with `4` and `C`. Thus the miss rate of the fully associative cache is 50%.

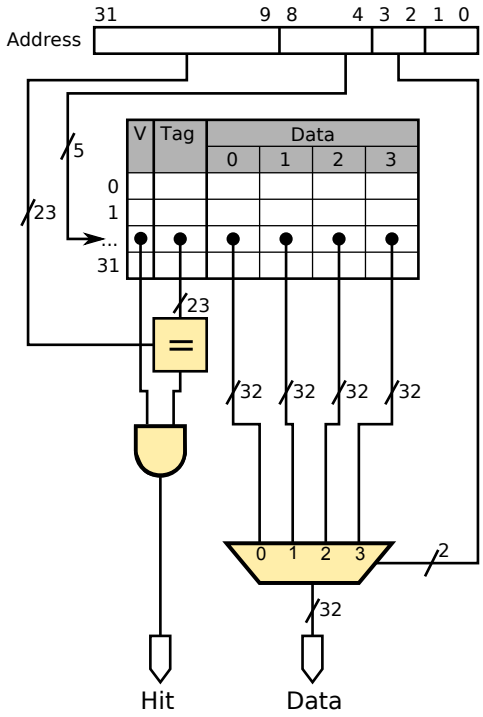


Figure 76: Structure of the direct mapped cache

iii) The lack of associativity of the direct-mapped cache results in a maximal penalty of a 100% miss rate. Given the fact that the 9 first (least significant) bits of each pair of consecutive accesses are identical, each access of the sequence will result in a miss.

c) The use of each bit of the address in the direct-mapped cache with 8 words per block is shown in figure 77.

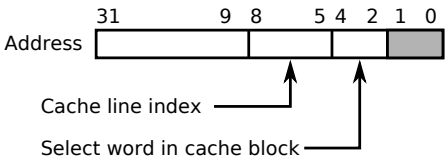


Figure 77: Structure of the address for the direct mapped cache

As we saw earlier, each pair of accesses is placed on the same line of the cache because the 9 least significant bits of each pair are identical. Increasing the block size to 8 words does not remedy the lack of associativity problem. In both caches the miss rate for the given sequence is 100%.

[Exercise 11] Code and Caches

The C program excerpt given below computes the multiplication of two square matrices **A** and **B** containing each 16 elements and stores the product in a matrix **C**. Integers `int` are encoded on 32 bits and memory is byte-addressed.

```
int A[4][4];
int B[4][4];
int C[4][4];
int i, j, k, acc;

...

for (i = 0; i < 4; i++)
{
    for (j = 0; j < 4; j++)
    {
        acc = 0;
        for (k = 0, k < 4; k++)
        {
            acc = acc + A[i][k] * B[k][j];
        }
        C[i][j] = acc;
    }
}
```

According to the C language standards, the three matrices **A**, **B** and **C** have virtual addresses as shown in figure 78.

Consider an architecture where data and instruction caches are separate. Caches are physically addressed, i.e. virtual addresses are translated before accessing the cache. The virtual memory uses 4 Kbyte pages and physical addresses are 32-bit wide.

a) If the aforementioned C code were given to a compiler, which C expressions would certainly be translated into assembly instructions that access the data cache? Suppose for the rest of the exercise that the order of these (assembly) instructions and their corresponding accesses is the same as in the C code and not modified (optimised) by the compiler.

b) Consider a direct mapped cache with 4-word blocks and a total size of 256 bytes. Draw the structure of this cache and precisely indicate the use of each bit of the physical address used to address the cache.

A	A[0][0]	B	B[0][0]	C	C[0][0]
A+4	A[0][1]	B+4	B[0][1]	C+4	C[0][1]
A+8	A[0][2]	B+8	B[0][2]	C+8	C[0][2]
A+12	A[0][3]	B+12	B[0][3]	C+12	C[0][3]
A+16	A[1][0]	B+16	B[1][0]	C+16	C[1][0]
A+20	A[1][1]	B+20	B[1][1]	C+20	C[1][1]
A+24	A[1][2]	B+24	B[1][2]	C+24	C[1][2]
A+28	A[1][3]	B+28	B[1][3]	C+28	C[1][3]
A+32	A[2][0]	B+32	B[2][0]	C+32	C[2][0]
A+36	A[2][1]	B+36	B[2][1]	C+36	C[2][1]
A+40	A[2][2]	B+40	B[2][2]	C+40	C[2][2]
A+44	A[2][3]	B+44	B[2][3]	C+44	C[2][3]
A+48	A[3][0]	B+48	B[3][0]	C+48	C[3][0]
A+52	A[3][1]	B+52	B[3][1]	C+52	C[3][1]
A+46	A[3][2]	B+46	B[3][2]	C+46	C[3][2]
A+60	A[3][3]	B+60	B[3][3]	C+60	C[3][3]

Figure 78: Virtual addresses of arrays A, B and C

c) Explain why it is possible to deduce the access sequence (physical addresses) to the data cache without knowing the contents of the page table.

d) Suppose `A = 0x400`, `B = 0x440` and `C = 0x480`. What is the miss rate of this cache when computing matrix C ? Consider the cache empty before the first `for` of the program. (Advice: do not simulate all accesses, but rather think of the position of the three matrices in the cache.)

e) Now suppose `A = 0x400`, `B = 0x800` and `C = 0x480`. What is the miss rate when computing the first three elements of matrix **C**? The cache is empty before the first access in memory.

f) Give the simplest structure of a 256-byte cache that always guarantees the same miss rate as in **d)** when computing the matrix **C** for all possible addresses of matrices **A**, **B** and **C**.

[Solution 11] Code and Caches

a) The C expression `acc = acc + A[i][k] * B[k][j];` accesses the data cache twice to load the coefficient of matrix **A** (`A[i][k]`) and the coefficient of matrix **B** (`B[k][j]`) in the registers.

Also, the expression `C[i][j] = acc;` generates a write access to the cache in order to store in memory the coefficient `c[i][j]` of matrix **C**.

b) A cache block contains 16 bytes. The cache thus contains 16 lines. The structure of the cache is given in figure 79.

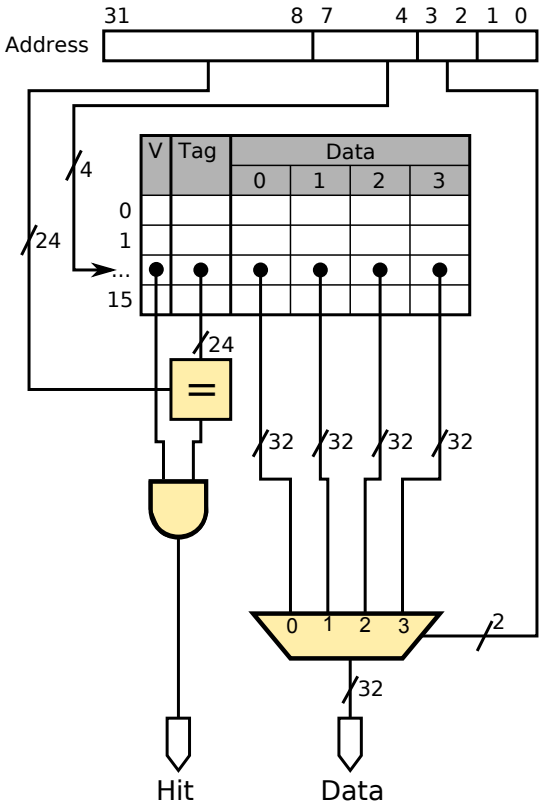


Figure 79: Structure of the direct-mapped cache

c) As indicated in **b)**, cache addressing is determined by the 8 least significant bits of the physical address. Moreover, pages have a size of 4 Kbytes, thus the 12 least significant bits of the virtual address determine the offset in a page. Since these 12 bits also constitute the 12 least significant bits of the physical address, we can determine the access sequence from the virtual address.

d) It can be noticed that accesses to matrices **A**, **B** and **C** do not generate any misses due to conflicts because they reside on different lines in the cache. Indeed, matrix **A** occupies lines 0, 1, 2 and 3, while matrix **B** occupies lines 4, 5, 6 and 7 and finally matrix **C** lines 8, 9, 10 and 11.

Computing matrix **C** generates 16 write accesses. For each one of them, 4 coefficients of matrix **A** and 4 coefficients of matrix **B** are read. The total number of accesses is thus $16 + 4 \times 16 + 4 \times 16 = 144$.

Within the accesses on matrices **A**, **B** and **C**, each access to a new line generates a miss. There are thus 4 misses for accesses to **A**, 4 misses for those to **B** and 4 as well for **C**.

Hence, there are 12 misses in total and thus the miss rate is $\frac{12}{144} = \frac{1}{12}$.

e) Contrary to point **d)**, matrices **A** and **B** reside in the same cache lines 0, 1, 2 and 3, which causes misses due to conflict. On the other hand, matrix **C** still resides in cache lines 8 to 11 and thus does not interfere with accesses to matrices **A** and **B**.

The access sequence generated by the the computation of the first three coefficients of **C** is given below along with the access type. (H: hit, CPM: Compulsory Miss, CFM: Conflict Miss).

Access	**H/CPM/CFM**	**Access**	**H/CPM/CFM**	**Access**	**H/CPM/CFM**
A[0][0]	CPM	B[0][0]	CPM		
A[0][1]	CFM	B[1][0]	CPM		
A[0][2]	H	B[2][0]	CPM		
A[0][3]	H	B[3][0]	CPM	C[0][0]	CPM
A[0][0]	H	B[0][1]	CFM		
A[0][1]	CFM	B[1][1]	H		
A[0][2]	H	B[2][1]	H		
A[0][3]	H	B[3][1]	H	C[0][1]	H
A[0][0]	H	B[0][2]	CFM		
A[0][1]	CFM	B[1][2]	H		
A[0][2]	H	B[2][2]	H		
A[0][3]	H	B[3][2]	H	C[0][2]	H

The sequence contains 27 accesses, of which 11 are misses. The miss rate for this sequence is thus $\frac{11}{27}$.

f) In order to avoid conflict misses as in **e)**, we can use an associative cache. Since there are three matrices, a 3-way associative cache would be enough to eliminate conflict misses. It is however not possible to build a 3-way cache with a size of 256 bytes (each way must have an integer number of bytes).

Thus, the simplest cache that eliminates conflict misses is associative, has 4 ways, 4 word blocks and 4 lines per way.

[Exercise 12] Victim Caches

Consider a system comprised of a 16-bit RISC processor with an 8-bit data bus. The processor is connected to two memory units, each having a 16-bit address bus and an 8-bit data bus. One memory is exclusively used for storing the program (*instruction memory*) and the other is used entirely for data (*data memory*). Between the processor and data memory, there is a 4kB, 2-way (*2-way set associative*) data-cache. Each line of the cache holds 32 bytes and the *Least Recently Used* (LRU) cache replacement policy and *write-back* policy is used.

a) Draw a diagram of the mentioned data cache indicating clearly how the address bits are used. Indicate the cache lines on which the following address ranges are mapped: [0x1000 - 0x10C7], [0x2000 - 0x20C7] and [0x3000 - 0x30C7].

b) Assume this system was used to perform the addition of two vectors A and B to compute a vector C, where A, B and C are stored as arrays. The following code snippet illustrates this:

```
1 /* A, B and C are the arrays representing the vectors
2 and have already been defined earlier */
3 for(int  i = 0; i<100; i++)
4 {
5         C[i] = A[i] + B[i];
6 }
```

The elements of A, B and C are 16-bit integers and are stored in the data memory in the address range given below. Note that the processor needs two accesses to the memory to read or to write a 16-bit value.

Variable	Adresse
A	[0x1000-0x10C7]
B	[0x2000-0x20C7]
C	[0x3000-0x30C7]

Assuming the cache was initially empty and the variable `i` is stored in a register, how many cache *hits* and cache *misses* occur while executing the above code snippet? Explain the impact of the cache on the system's performance in this case.

c) Propose another cache structure which has a significantly better performance when the code in question **b)** is being executed. The proposed cache should have same size with the original and should have minimal cost. Explain your choice briefly and

provide the new number of cache *hits* and *misses* for the same code snippet. In case of multiple solutions, discuss the advantages and disadvantages for each.

d) A *victim cache* is a small data cache used to hold blocks that are *evicted* from the primary cache upon replacement. The *victim cache* lies between the primary cache and the memory and is usually *fully-associative*. When a *victim cache* is present, following a regular cache *miss* the *victim cache* is first queried and the query propagates to the main memory only if the data item is not present in the *victim cache*. Note that it is not an L2 cache, since it does not fill its data from the memory, but only stores the cache-lines *evicted* from the primary cache. When a *hit* occurs in the *victim cache*, the retrieved data migrates to the primary cache.

We add a *fully-associative* victim cache that can hold 16 lines of 32-byte data between the data-cache and the data-memory to our system. Still using the same code given in **a)**:

1. How many data-cache *misses* would propagate to the main memory?
2. How many *hits* would occur in the *victim cache*?

e) Find the minimum size of the *victim cache* which allows the same performance as in question **d)** for the code snippet given in **a)**. If the primary cache is *direct-mapped*, is this minimal size would be affected? Give your responses in number of blocks with a brief explanation.

f) With the system given in **d)** and for the code snippet given in **a)**, a random replacement policy is used in the primary cache.

1. What is the minimum size of the *victim cache* that will maintain the same performance?
2. Can the performance of the system improve with this change? Explain.
3. Can the performance of the system degrade with this change? Explain.

[Solution 12] Victim Cache

a) The Figure 80 illustrate the 2-way set associative cache. The three ranges of addresses will all be mapped to the cache lines 0 to 6.

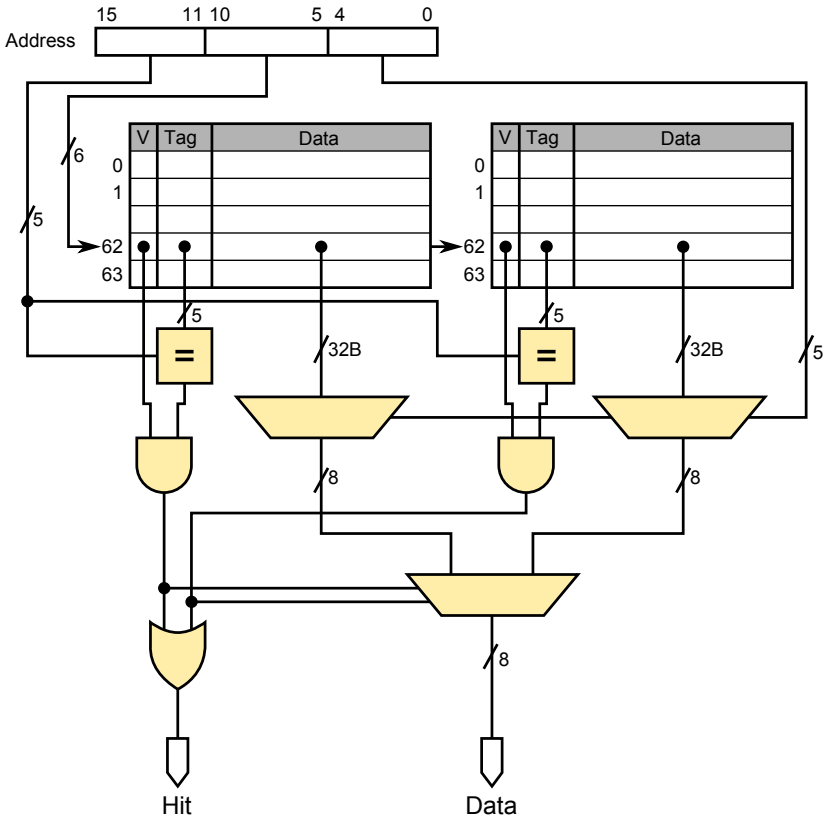


Figure 80: The 2-way set associative cache

b) In each iteration, we have two pairs of loads and one pair of stores. Each pair of accesses will first generate a miss (*compulsory miss*) and then a hit. The three arrays are aligned on the same cache line and have available only two ways: this will necessarily generate conflict misses. In total, we will get 300 misses and 300 hits. The cache can improve the performance of the reference system for only 50% of the accesses but could potentially do better.

c) An associativity of three (that is, adding a third way to the set-associative cache) will remove all the conflicts and the execution of the program will now generate only 21 compulsory misses for 579 hits. Note that there is no need for the associativity to be a power of two, but if one wants to design a cache with exactly the same total data memory, it would make sense to halve the number of lines in each way (which *must* be a power of two) and thus use four ways.

Another solution would be to change the write-back policy to a write-through with unallocated store misses. Here only two arrays would have to share the two ways of the cache, and therefore this change would remove all the conflicts as well. However, a write-through policy would be more expensive in terms of power and, potentially, performance because it would use the memory bus more frequently.

d) With a victim cache we still have the 21 compulsory misses and the 300 hits to the data cache. The remaining 279 accesses are still misses to the main cache but hits to the victim cache, as the corresponding cache lines would have been evicted very recently.

e) One could roughly see the victim cache as individual cache lines used to increase as needed the associativity of individual sets. Since in our particular case the program works only on a single set at a time and we would need an associativity of three to remove all conflict misses, a single entry in the victim cache is sufficient. Similarly, in the case of a direct-mapped data cache, we would need two entries in the victim cache to hide the conflicts.

f)

1. We have to consider the worst decisions possible for the random replacement. In the worst case, the cache could almost behave as a direct-mapped cache, always evicting the most useful piece of data. Therefore, we would need two entries in the victim cache, as discussed above.
2. Yes, we can potentially get better performance. For example, if the array A would load in one way of the cache and if the second way would be shared between B and C, we would get fewer hits to the victim cache but more to the data cache, which has a smaller hit latency.
3. No we cannot have lower performance, because no matter what is the random replacement we won't get less than 300 hits in the data cache (due to the read/write pairs) or more than 21 misses (compulsory) going to the memory.

[Exercise 13] Caches

Consider the following assembly code (for a 32-bit processor whose assembly language resembles RISC-V) and the equivalent C code. This program performs a peculiar addition of two arrays of `int` (32 bits) `a` and `b`, and stores the result in array `c`.

```
    addi  t0, zero, 0 # Initializing i to zero
loop:
    andi  t1, t0, 1
# Check if i is even/odd
    beq   t1, 1, odd
even:
    lw    t2, 0(a0)   # Load a[i] into t2
    lw    t3, 2(a1)
# Load b[i+2] into t3
    j     addition    # Go to addition
odd:
    lw    t2, 0(a0)   # Load a[i] into t2
    lw    t3, 0(a1)   # Load b[i] into t3
addition:
    add   t2, t2, t3  # Add t2 to t3
    sw    t2, 0(a2)
# Store result in c[i]
    addi  a0, a0, 1
# Increment array pointers
    addi  a1, a1, 1
    addi  a2, a2, 1
    addi  t0, t0, 1   # Increment i
    bne   t0, a3, loop# Iterate N times
end:
```

```
for(int i = 0; i < N; i++) {
    if (i % 2 == 0){
        // i is even
        c[i] = a[i] + b[i+2];
    }
    else {
        c[i] = a[i] + b[i];
    }
}
```

For the assembly code above, the following initializations are assumed:

- Register `a0` holds the address of the first element of array `a`.
- Register `a1` holds the address of the first element of array `b`.
- Register `a2` holds the address of the first element of array `c`.
- Register `a3` holds the total number of iterations `N`.

a) Is the memory addressed by byte or word in this particular processor? Justify your answer.

b) Suppose now that for a specific run of the above assembly code, the arrays are located in memory such that the first element of array `a` is stored at memory address

0x000 while the first element of array `b` is stored at address 0x3FE and the first element of array `c` is stored at memory address 0x820. Given an initially empty, direct-mapped data cache with a total size of 64 words and having two words per line/block:

1. Draw the structure of the cache memory, assuming that the address is 16-bit wide. Clearly indicate the use of each bit of the address by describing its format in detail.
2. Compute the hit rate of the cache while running the code for 1000 iterations (i.e., `N` is 1000).

c) How can you rewrite the above assembly code (with minimal changes) to improve the computed hit rate while maintaining the exact same functionality of the code? You do not need to rewrite the whole code, just clearly specify the changes and justify your answer. What is the improved hit rate?

d) Consider now that the direct-mapped cache is replaced by a 2-way set-associative cache with the Least Recently Used (LRU) replacement policy, having the same total cache size and still with two words per line/block. Assuming that the initial code provided above is used (and not the one modified by you), compute the new hit rate.

[Solution 13] Caches

a) The memory is word-addressable.
Reason 1: Examining the assembly code, each element of the arrays `a` and `b` is read as a word using the instruction **`lw`**, and then the array pointers are incremented by 1 to point to the next array element respectively.
Reason 2: The instruction `lw t3, 2(a1) # Load b[i+2] into t3` reads the element `b[i+2]` by loading the second word two locations away from the head of the pointer of the array.

b)

1. The structure of the cache is shown in the diagram on Figure 81.
2. Table 5 lists the cache misses and hits after running the code for a few iteration.

Table 5: Cache Miss/Hit while running the code for few iterations.

	Accessing a		Accessing b		Accessing c	
Operation	Cache line	Hit/Miss	Cache line	Hit/Miss	Cache line	Hit/Miss
$c[0] = a[0] + b[2]$	0	M	0	M	16	M
$c[1] = a[1] + b[1]$	0	M	31	M	16	H
$c[2] = a[2] + b[4]$	1	M	1	M	17	M
$c[3] = a[3] + b[3]$	1	M	0	M	17	H
$c[4] = a[4] + b[6]$	2	M	2	M	18	M

The hit rate (according to Table 5) is:

$$\text{Hit rate} = 1/6$$

c) We can notice that loading `b[i+2]` before `a[i]` would lead to more hits. For example, loading `a[0]` means that `a[1]` is also loaded to the cache. And since `b[2]` has already been read, then during the next iteration `a[1]` would be available in the cache (Hit).
The only modified section of the assembly code would be:

```
even:
    lw      t3, 2(a1)       # Load b[i+2] into t3
    lw      t2, 0(a0)       # Load a[i] into t2
    j       addition        # Go to addition
```

The new hit rate (according to Table 6) becomes:

$$\text{Hit rate} = 1/3$$

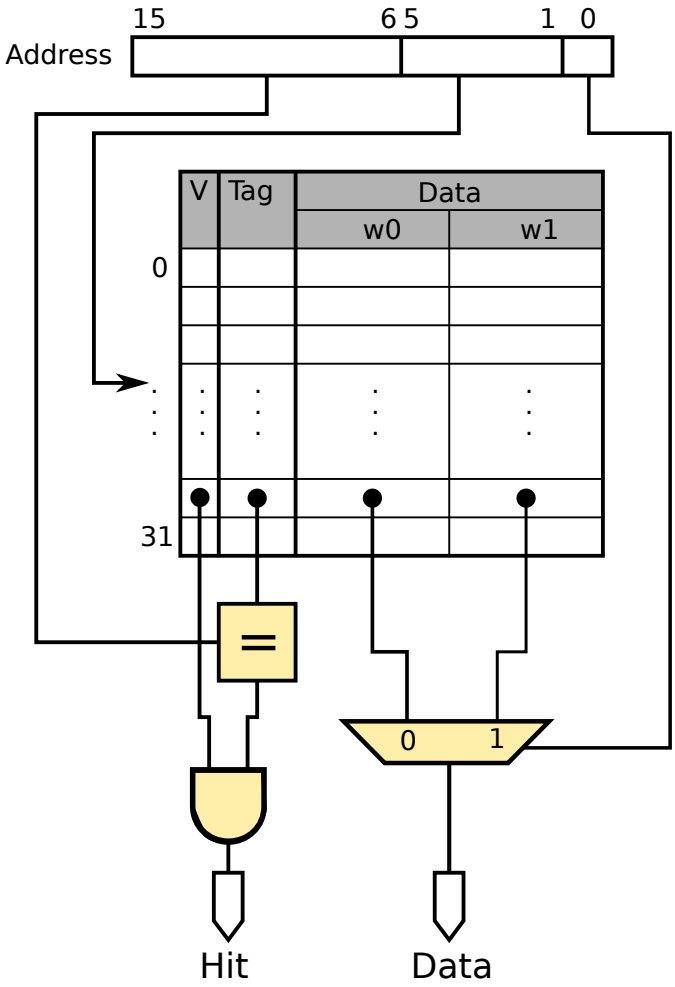


Figure 81: Structure of the cache.

d) Replacing the direct-mapped cache with a 2-way set associative cache, results in having all three arrays (a, b and c) aligned. The only hits will now be in c due to the LRU replacement protocol.

The hit rate (according to Table 7) would be:

$$\text{Hit rate} = 1/6$$

Table 6: Cache Miss/Hit while running the newly changed code for few iterations.

	Accessing a		Accessing b		Accessing c	
Operation	Cache line	Hit/Miss	Cache line	Hit/Miss	Cache line	Hit/Miss
$c[0] = b[2] + a[0]$	0	M	0	M	16	M
$c[1] = a[1] + b[1]$	0	H	31	M	16	H
$c[2] = b[4] + a[2]$	1	M	1	M	17	M
$c[3] = a[3] + b[3]$	1	H	0	M	17	H
$c[4] = b[6] + a[4]$	2	M	2	M	18	M

Table 7: Miss/Hit of a 2-way set associative cache, while running the code for few iterations.

	Accessing a		Accessing b		Accessing c	
Operation	Cache line	Hit/Miss	Cache line	Hit/Miss	Cache line	Hit/Miss
$c[0] = a[0] + b[2]$	0	M	0	M	0	M
$c[1] = a[1] + b[1]$	0	M	15	M	0	H
$c[2] = a[2] + b[4]$	1	M	1	M	1	M
$c[3] = a[3] + b[3]$	1	M	0	M	1	H
$c[4] = a[4] + b[6]$	2	M	2	M	2	M
$c[5] = a[5] + b[5]$	2	M	2	M	2	H

[Exercise 14] Memory Hierarchy

This problem consists of several short and independent questions covering the topics of caches and virtual memory.

a) Consider a 32-bit microprocessor that has an on-chip 16 Kbyte four-way set-associative cache. Assume that the cache has a line size of four 32-bit words and that main memory is word addressed.

1. Draw the structure of this cache showing its organization. Clearly indicate the use of each bit of the address by describing its format in detail.
2. Give any two main memory addresses with different tags that map to the same cache line. Justify your choice.

b) Consider a direct mapped cache that consists of 4 lines, where each line consists of 16 Bytes. The main memory is byte addressed and one word is on 8 bits.

Now, consider a program that accesses memory in the following sequence of addresses:

- Reading once from memory locations 62 to 70;
- Reading from memory locations 15 to 32, followed by 80 to 95, repeatedly for five times.

1. List the cache misses and hits that would occur at the execution of the above program.
2. Compute the hit rate.

c) Assume the same memory structure and program execution as in question 2. However, in this question, the cache is organized as a two-way set-associative cache, with two sets of two 16 Byte lines each.

1. Compute the hit rate of the two-way set-associative cache using the Least Recently Used (LRU) replacement policy.
2. Justify the hit rate changes, if any, between the direct mapped cache of question 2 and this two-way set-associative cache.

d) Consider a byte-addressed memory with a 32-bit physical address. This system has a virtual memory with the following characteristics:

- A 4 Gbyte page table size, organized as a regular array;
- A page table entry of 16 Bytes;
- A page size of 4 Kbytes.

1. What is the size of the offset field? How many virtual pages can there be in this system? How many bits are used for the virtual address?
2. Draw a diagram of the classical translation process properly showing all the steps needed to translate a virtual page number into a physical address. Clearly indicate the precision (width) of all variables.
3. What changes would you have in this system and in the diagram of question 2 if the virtual memory had a page table entry of 4 Bytes (instead of 16 Bytes)? Justify your answer.

[Solution 14] Memory Hierarchy

a)

1. The cache has a size of 16 KB, divided into 4-way sets. Each way contains a block consisting of 4 words. Therefore, we have the following cache size relations:

   - Cache size = 16 KB
   - Line size = 4×32 bits = 16 B
   - Size per way = $\frac{16\text{KB}}{4}$ = 4 KB
   - Number of lines per way = $\frac{4\text{KB}}{16\text{B}}$ = 256

   We can now compute the number of address bits used for the tag, index, and word selector:

   - Number of lines per way = 256 = $2^8 \rightarrow$ 8-bit index
   - Number of words per line = $4 \times$ 32-bit word $\rightarrow$ 2-bit word selector
   - Number of bits for tag = $32 - (8 + 2)$ = 22-bit tag

   Figure 82 shows the structure of the cache.

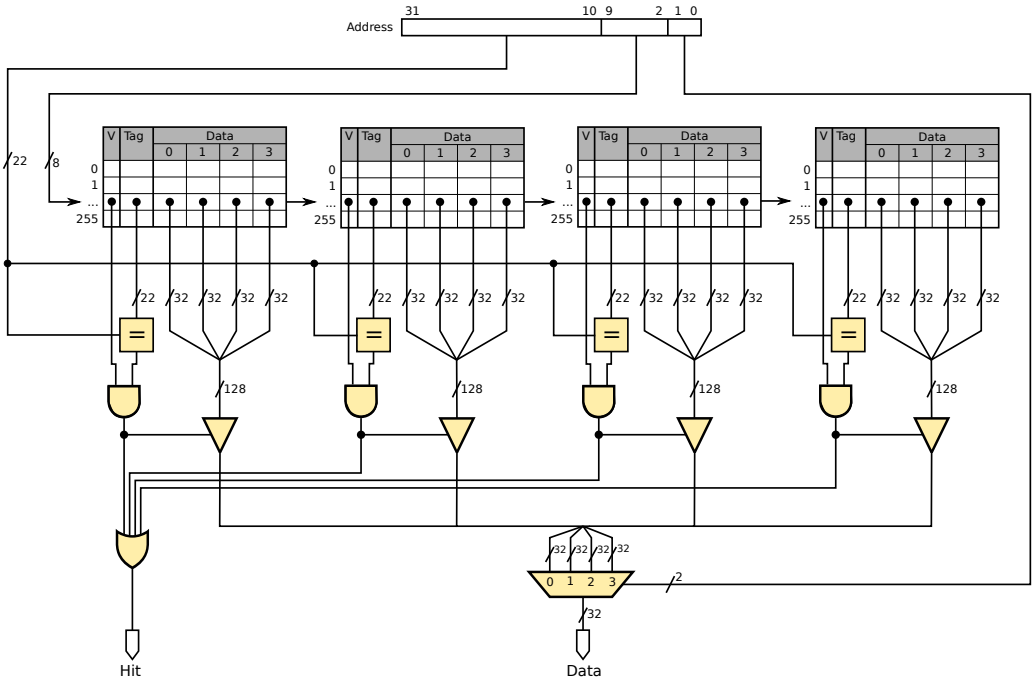


Figure 82: Structure of the 4-way set-associative cache

2. In order for two distinct addresses to map to the same cache line, they need to have the same index. For example, Addresses `0x0013` and `0x1013` each map to cache line 4.

   However, note that the word selection bits (2 LSBs) can be different!

b)

1. Figure 83 shows the structure of the cache (assuming a 32-bit address).

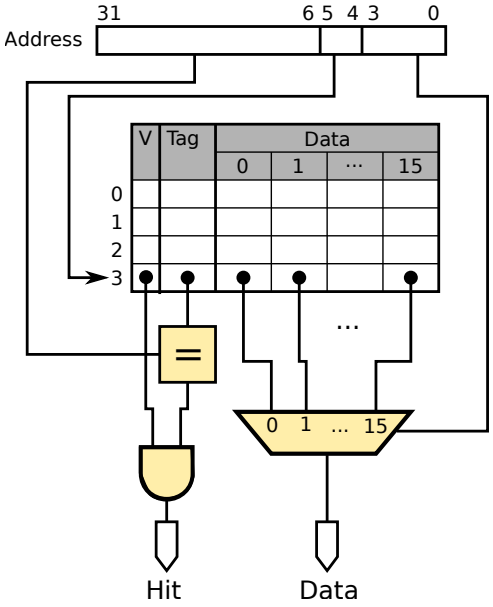


Figure 83: Structure of the direct mapped cache

Tables 8, 9, and 10 show the cache accesses for the address sequence provided.

Address:	62	63	64	65	66	67	68	69	70
Cache line:	3	3	0	0	0	0	0	0	0
Hit/Miss:	M	H	M	H	H	H	H	H	H

Table 8: Cache accesses for addresses 62 to 70

Address:	15	16	17	...	31	32	80	81	...	95
Cache line:	0	1	1	...	1	2	1	1	...	1
Hit/Miss:	M	M	H	...	H	M	M	H	...	H

Table 9: Cache accesses for iteration 1 of the loop for addresses 15 to 32 and 80 to 95

Address:	15	16	17	...	31	32	80	81	...	95
Cache line:	0	1	1	...	1	2	1	1	...	1
Hit/Miss:	H	M	H	...	H	H	M	H	...	H

Table 10: Cache accesses for iterations 2, 3, 4, and 5 of the loop for addresses 15 to 32 and 80 to 95

2. From the previous access tables, we can compute the following values:
   - Number of accesses = $9 + 5 \times (18 + 16) = 179$
   - Number of misses = $2 + 4 + (4 \times 2) = 14$
   - Hit rate = $\frac{179-14}{179} = 92.18\%$

c)

1. Figure 84 shows the structure of the cache.

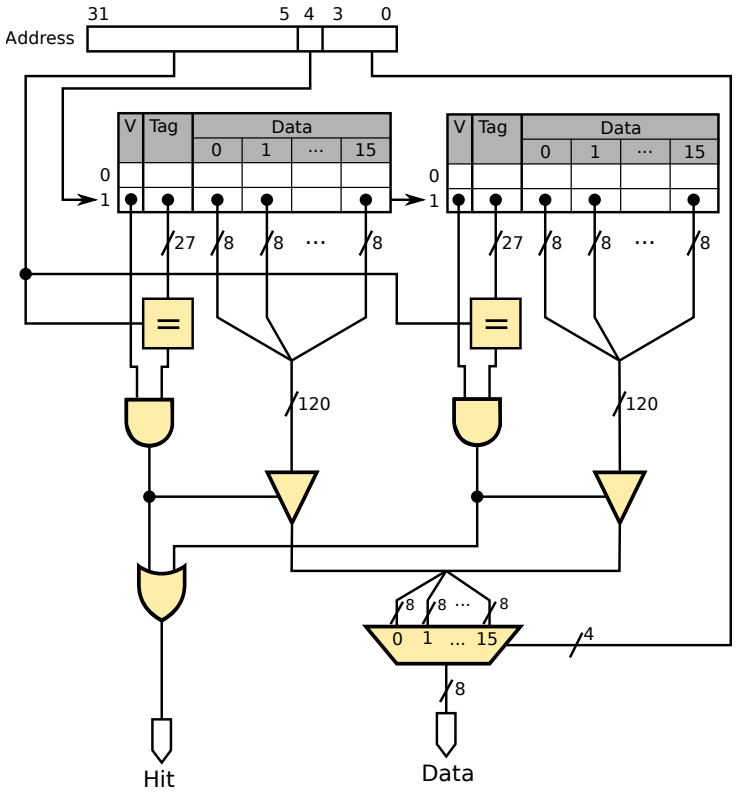


Figure 84: Structure of the 2-way set-associative cache

Tables 11, 12, and 13 show the cache accesses for the address sequence provided.

Address:	62	63	64	65	66	67	68	69	70
Cache line:	1	1	0	0	0	0	0	0	0
Hit/Miss:	M	H	M	H	H	H	H	H	H

Table 11: Cache accesses for addresses 62 to 70

2. From the previous access tables, we can compute the following values:
   - Number of accesses = 179
   - Number of misses = $2 + 4 + (4 \times 0) = 6$
   - Hit rate = $\frac{179-6}{179} = 96.65\%$

Address:	15	16	17	...	31	32	80	81	...	95
Cache line:	0	1	1	...	1	0	1	1	...	1
Hit/Miss:	M	M	H	...	H	M	M	H	...	H

Table 12: Cache accesses for iteration 1 of the loop for addresses 15 to 32 and 80 to 95

Address:	15	16	17	...	31	32	80	81	...	95
Cache line:	0	1	1	...	1	0	1	1	...	1
Hit/Miss:	H	H	H	...	H	H	H	H	...	H

Table 13: Cache accesses for iterations 2, 3, 4, and 5 of the loop for addresses 15 to 32 and 80 to 95

d)

1.
   - Page size = 4 KB = 2^{12} B $\rightarrow$ 12-bit page offset.
   - # of virtual pages = $\frac{\text{page table size}}{\text{page table entry size}} = \frac{4\text{GB}}{16\text{B}} = \frac{2^{32}}{2^4} = 2^{28} \rightarrow$ 28-bit virtual page number.
   - Virtual address size = 28 + 12 = 40.
2. Figure 85 shows the virtual memory addressing scheme.

   Each page table entry has a size of 16 B = 2^4 B. Therefore, we need to shift the virtual page number by 4 before adding the page table base address.
3. We would need to perform the following changes:
   - Shift the virtual page number by 2 instead of 4.
   - # of virtual pages = $\frac{4\text{GB}}{4\text{B}} = \frac{2^{32}}{2^2} = 2^{30} \rightarrow$ 30-bit virtual page #.
   - Virtual address size = 30 + 12 = 42 bits.

   Note that the page offset does not change.

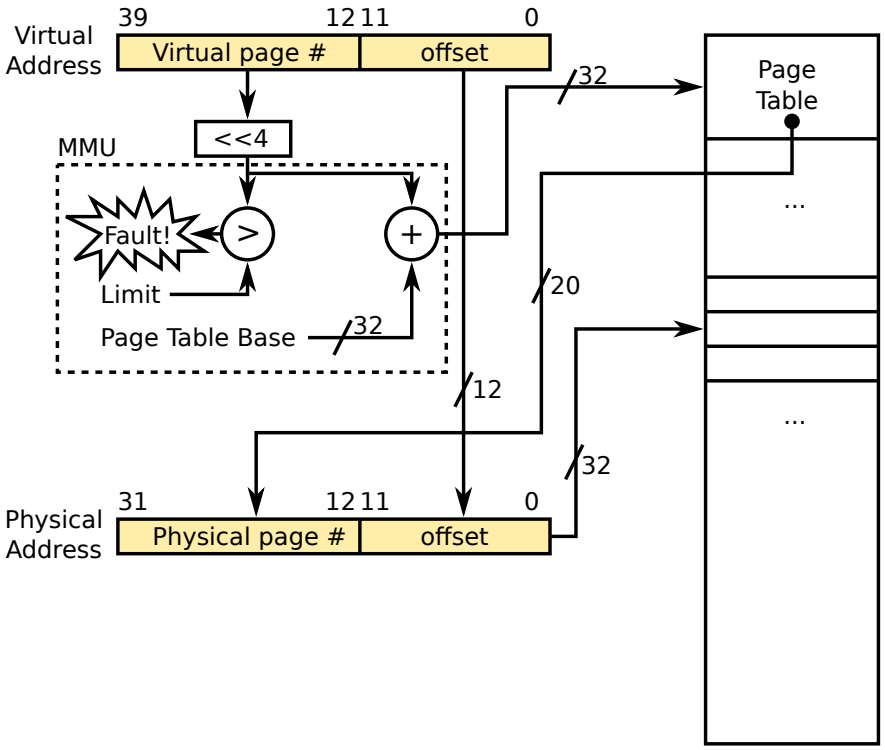


Figure 85: Virtual memory addressing scheme

[Exercise 15] Direct-Mapped Cache

Consider a **unified cache** (shared by data and instructions) with a total capacity of 32 bytes. Each cache block has 2 words; the size of one word is 4 bytes. The addresses are 16-bit wide. The main memory is **byte addressed**. These parameters are valid for the complete exercise.

Take a look at the following piece of **RISC-V** assembly code and its equivalent in **C** on the right:

```
loop: addi    t1, t0, -4
      lb      t2, x(t1)
      addi    t3, t0, 4
      lb      t4, x(t3)
      add     t2, t2, t4
      sb      t2, y(t0)
      addi    t0, t0, 1
      bne     t0, t5, loop
```

```
for (i = 4; i < 7; i++)
{
    y[i] = x[i - 4] + x[i + 4];
}
```

The initial value of register `t0` is 4 and the initial value of register `t5` is 7. Symbols `x` and `y` are equal to 0x4000 and 0x2010, respectively. These symbols represent the starting addresses of two **arrays of bytes**. The first instruction of this assembly code is stored in memory at the address 0x1000. Instructions are 32-bit wide.

a) Assume that the cache is **direct-mapped**.

a.1) Indicate the use of each bit of the address by describing the address format in detail. Draw the structure of the cache.

a.2) Draw the table below and fill it in with the **complete sequence of memory accesses** generated by the assembly code. Given an initially empty cache, for every memory access indicate precisely the index of the corresponding cache line, whether the memory access results in a cache hit (**H**) or in a miss (**M**), and what is the new content of the cache line.

Address (hexa)	Index of the cache line	Hit / Miss	Content of the cache line
...	...	...	...

a.3) How many cache hits occurred? Compute the hit rate.

b) Consider now a direct-mapped cache with a **dynamic exclusion (replacement) policy**. This replacement policy uses one additional control bit per cache block, called **sticky bit**, to decide if an entry should be evicted from or kept in the cache.

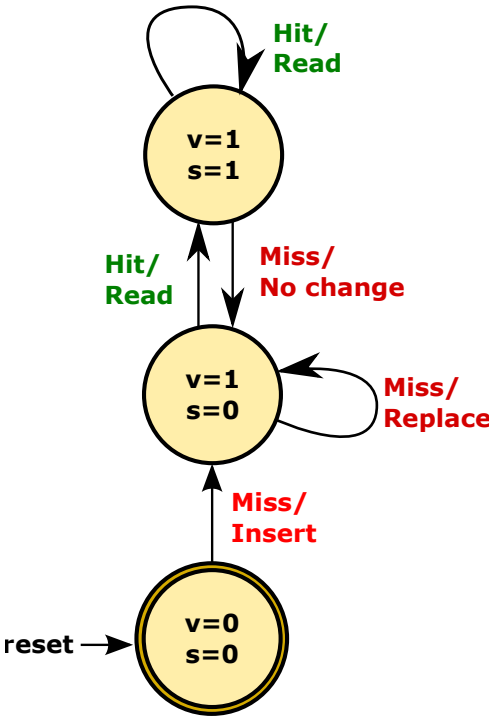


Figure 86: Finite state machine describing dynamic exclusion policy.

The decision mechanism is best described using a finite-state machine (FSM) shown in Figure 86. Each state of the FSM indicates the values of valid bit and sticky bit of the cache block currently in use. The arc labels give the conditions for the state changes and appropriate actions taken.

The first time an entry is saved (inserted) in the cache, its sticky bit is reset. After a cache hit for that cache block, the sticky bit is set and remains set as long as only hits happen, i.e., as long as there are no conflicting accesses (those that would result in a miss and eviction in regular direct-mapped cache). While **s = 1**, the cache entry will not be replaced, because it is considered very important. It takes two consecutive cache misses to replace an entry whose sticky bit is set. After the first miss, the sticky bit is reset, the conflicting access is executed, but the entry in the cache block is not changed. When **s = 0**, a single miss is sufficient to evict the entry and replace it by a new one.

Assuming a direct-mapped cache with the same parameters as in question (a), but which uses the described dynamic exclusion policy, answer the following questions.

b.1) Draw the table below and fill it in with the **complete sequence of memory ac-**

cesses generated by the assembly code. Given an initially empty cache, for every memory access indicate precisely the index of the corresponding cache line, whether the memory access results in a cache hit (**H**) or in a miss (**M**), the value of the sticky bit, and what is the new content of the cache line.

Address (hexa)	Index of the cache line	Hit / Miss	Sticky bit	Action taken (Insert/Read/Replace/No change)	Content of the cache line
...	...	...	...	...	...

b.2) How many cache hits occurred? Compute the hit rate. Which of the two caches performs better for the given sequence of accesses? Why? Justify your answer.

[Solution 15] Direct-Mapped Cache

a.1) The structure of the direct-mapped cache is given in the figure below:

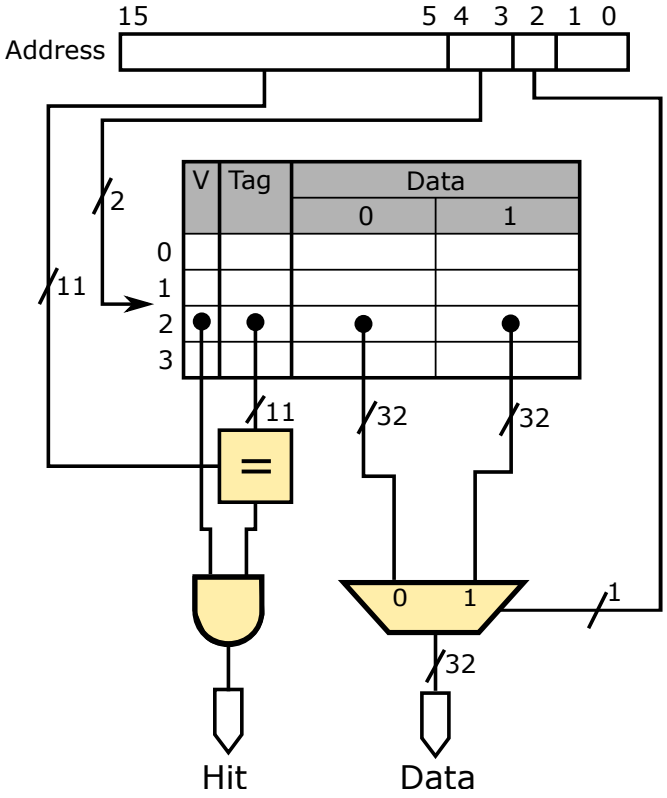


Figure 87: Structure of the direct-mapped cache.

a.2) The complete sequence of memory accesses is given in the table below.

a.3) The total number of accesses is 33. Hence, the hit rate of the cache is 14/33.

b.1) The complete sequence of memory accesses is given in the table further below.

b.2) The total number of accesses is 33. Hence, the hit rate of the cache is 20/33. The cache with the dynamic exclusion policy performs better because it keeps in the cache the addresses that need to be accessed multiple times (in this case, the addresses of the instructions).

Address (hexa)	Index of the cache line	Hit / Miss	Content of the cache line (Direct-mapped cache)
0x1000	0	miss	0x1000..3 0x1004..7
0x1004	0	hit	0x1000..3 0x1004..7
0x4000	0	miss	0x4000..3 0x4004..7
0x1008	1	miss	0x1008..B 0x100C..F
0x100C	1	hit	0x1008..B 0x100C..F
0x4008	1	miss	0x4008..B 0x400C..F
0x1010	2	miss	0x1010..3 0x1014..7
0x1014	2	hit	0x1010..3 0x1014..7
0x2014	2	miss	0x2010..3 0x2014..7
0x1018	3	miss	0x1018..B 0x101C..F
0x101C	3	hit	0x1018..B 0x101C..F
0x1000	0	miss	0x1000..3 0x1004..7
0x1004	0	hit	0x1000..3 0x1004..7
0x4001	0	miss	0x4000..3 0x4004..7
0x1008	1	miss	0x1008..B 0x100C..F
0x100C	1	hit	0x1008..B 0x100C..F
0x4009	1	miss	0x4008..B 0x400C..F
0x1010	2	miss	0x1010..3 0x1014..7
0x1014	2	hit	0x1010..3 0x1014..7
0x2015	2	miss	0x2010..3 0x2014..7
0x1018	3	hit	0x1018..B 0x101C..F
0x101C	3	hit	0x1018..B 0x101C..F
0x1000	0	miss	0x1000..3 0x1004..7
0x1004	0	hit	0x1000..3 0x1004..7
0x4002	0	miss	0x4000..3 0x4004..7
0x1008	1	miss	0x1008..B 0x100C..F
0x100C	1	hit	0x1008..B 0x100C..F
0x400A	1	miss	0x4008..B 0x400C..F
0x1010	2	miss	0x1010..3 0x1014..7
0x1014	2	hit	0x1010..3 0x1014..7
0x2016	2	miss	0x2010..3 0x2014..7
0x1018	3	hit	0x1018..B 0x101C..F
0x101C	3	hit	0x1018..B 0x101C..F

Address (hexa)	Index of the cache line	Hit / Miss	Sticky bit	Action taken (Insert/ Read/Replace/No change)	Content of the cache line (with dynamic exclusion)
0x1000	0	miss	0	insert	0x1000..3 0x1004..7
0x1004	0	hit	1	read	0x1000..3 0x1004..7
0x4000	0	miss	0	no change	0x1000..3 0x1004..7
0x1008	1	miss	0	insert	0x1008..B 0x100C..F
0x100C	1	hit	1	read	0x1008..B 0x100C..F
0x4008	1	miss	0	no change	0x1008..B 0x100C..F
0x1010	2	miss	0	insert	0x1010..3 0x1014..7
0x1014	2	hit	1	read	0x1010..3 0x1014..7
0x2014	2	miss	0	no change	0x1010..3 0x1014..7
0x1018	3	miss	0	insert	0x1018..B 0x101C..F
0x101C	3	hit	1	read	0x1018..B 0x101C..F
0x1000	0	hit	1	read	0x1000..3 0x1004..7
0x1004	0	hit	1	read	0x1000..3 0x1004..7
0x4001	0	miss	0	no change	0x1000..3 0x1004..7
0x1008	1	hit	1	read	0x1008..B 0x100C..F
0x100C	1	hit	1	read	0x1008..B 0x100C..F
0x4009	1	miss	0	no change	0x1008..B 0x100C..F
0x1010	2	hit	1	read	0x1010..3 0x1014..7
0x1014	2	hit	1	read	0x1010..3 0x1014..7
0x2015	2	miss	0	no change	0x1010..3 0x1014..7
0x1018	3	hit	1	read	0x1018..B 0x101C..F
0x101C	3	hit	1	read	0x1018..B 0x101C..F
0x1000	0	hit	1	read	0x1000..3 0x1004..7
0x1004	0	hit	1	read	0x1000..3 0x1004..7
0x4002	0	miss	0	no change	0x1000..3 0x1004..7
0x1008	1	hit	1	read	0x1008..B 0x100C..F
0x100C	1	hit	1	read	0x1008..B 0x100C..F
0x400A	1	miss	0	no change	0x1008..B 0x100C..F
0x1010	2	hit	1	read	0x1010..3 0x1014..7
0x1014	2	hit	1	read	0x1010..3 0x1014..7
0x2016	2	miss	0	no change	0x1010..3 0x1014..7
0x1018	3	hit	1	read	0x1018..B 0x101C..F
0x101C	3	hit	1	read	0x1018..B 0x101C..F

[Exercise 16] Direct-Mapped and Annex Caches

Consider an initially empty direct-mapped **data cache** with a total capacity of 2 KB and 64-byte lines. Write-through and write-allocate strategies are employed. The word size is 4 bytes. Word-addressing is used and memory addresses are 32-bit long.

a) [5 pts] Draw the structure of the cache. Clearly indicate the use of each bit of the address.

b) [4 pts] **Conflict within loops:** Consider a case where two memory references A and B in the body of the same loop (`while`, `for`, etc.) map to the same location in the cache. If the loop is executed 10 times, the memory access pattern may be represented as $(AB)^{10}$, where the superscript (l'exposant, in French) denotes the number of times that one or a set of instructions is executed. Since each access (A or B) will evict the data corresponding to the previous access out of the cache, neither will hit, and the behaviour of the cache can be described as

$$(A_{\mathrm{M}}B_{\mathrm{M}})^{10}, \tag{1}$$

where subscript (l'indice, in French) **M** denotes that reference A is a miss (a subscript **H** would denote a hit). Therefore, for the sequence of accesses $(AB)^{10}$, the cache miss rate is

$$\text{miss rate} = 100\%. \tag{2}$$

Conflict between inner and outer loops: Now consider a case of a nested loop (one inner, one outer loop) in which there is a conflict between a reference inside the inner loop (A) and another reference outside the inner loop (B). If the outer loop is executed 10 times and the inner loop is executed 3 times **per every iteration** of the outer loop, the memory access pattern may be represented as $(A^3B)^{10}$. For this access pattern and assuming the cache is initially empty:

b.1) Describe the cache behavior using the same approach as in Equation (1).

b.2) Calculate the miss rate.

b.3) A cache access (hit or miss) takes t_{C} = 1 cycle. In case of a cache miss, **additional** t_{MEM} = 100 cycles are spent to access main memory. Calculate the average memory access latency t_{AVG}.

b.4) ◇ How can we change the cache structure, while maintaining the same cache capacity, to achieve the best possible miss rate?
◇ Calculate the miss rate in that case.

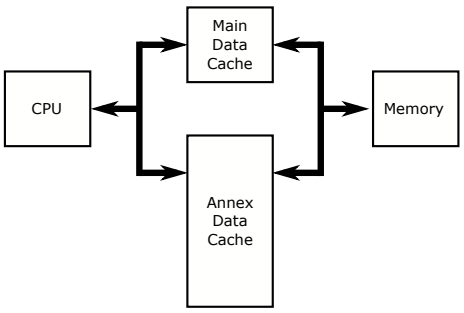


Figure 88: Memory hierarchy.

c) [6 pts] Let us now add another cache structure, annex cache, to our memory hierarchy (Figure 88).

In this system, upon a memory access request, both caches are probed simultaneously:

- If an address A misses in both caches and the corresponding cache line in the main cache has no valid data yet, the missing data is fetched from the memory and placed in the **main cache**.
- If an address A misses in both caches and the corresponding cache line in the main cache does hold valid data, the missing data is fetched from memory and placed in the **annex cache**.
- If an address A misses in the main cache but hits in the annex cache, then the history of misses and hits in the main cache line where A would be inserted is analyzed:
  - If this is the first miss for address A, the main cache remains **unchanged**. First miss for A means that before this miss the cache line was referenced using a different address B.
  - If this is the second consecutive miss for address A, the data corresponding to the address A is moved from annex cache to the main cache, while the data evicted from the main cache is moved to the annex cache (**swap**). Second consecutive miss for A means that there were no references to the same main cache line using a different address B between the first and the second miss for A.

  Whenever an address hits in the annex cache, it is the annex cache that replies to the CPU.
- If an address A hits in the main cache, the system stops looking for data inside the annex cache, to save time.

To illustrate how this system works on an example, let's consider a sequence of accesses $(AB^{10})^1$, where both references A and B map to the same line in the main cache, and both caches initially empty. Reference A will miss in both caches, after which it will enter the main cache from the memory. Reference B will miss in both caches, after which it will enter the annex cache from the memory. The next reference is again B, causing a second consecutive miss in the main cache. As a consequence, the data corresponding to B will be moved from the annex to the main cache, while the data corresponding to A will be moved from the main to the annex cache (swap). Next eight references to B will all result in a hit in the main cache. Therefore, the system behavior for the sequence $(AB^{10})^1$ can be described as:

$$A_{\mathrm{M}} B_{\mathrm{M}} B_{\text{H-A-SWAP}} B_{\mathrm{H}}^{8}, \tag{3}$$

where subscript M stands for miss in both caches, H for hit in the main cache, H-A-SWAP for hit in the annex cache followed by a swap.

Assume now that both the main cache and the annex cache are empty and that the annex cache has the exact same structure but a much larger capacity than the main cache. If needed, use subscript H-A to represent a hit in the annex cache.

c.1) ◇ Describe the system behavior for the sequence of accesses $(AB)^{10}$.
◇ How many times the requested data was found in the main cache?
◇ How many times the data which was **not** in the main cache was found in the annex cache?

c.2) ◇ Describe the system behavior for the sequence of accesses $(A^3B^2)^{10}$.
◇ How many times the requested data was found in the main cache?
◇ How many times the data which was **not** in the main cache was found in the annex cache?

c.3) Write the expression for the average memory access latency t_{AVG}, given the following information:

- N_{H} = number of times there was a hit in the main cache
- $N_{\text{H-A}}$ = number of times there was a miss in the main but hit in the annex cache
- N_{M} = number of times there was a miss in both caches, but the data was placed in the main
- N_{A} = number of times there was a miss in both caches, but the data was placed in the annex
- Accessing the main cache takes t_{C} = 1 cycle. Accessing the annex cache takes t_{A} = 10 cycles. In case of a cache miss, either in the main or the annex cache, **additional** t_{MEM} = 100 cycles are spent to access main memory and update the cache. The time to swap data between the main and the annex cache is negligible.

[Solution 16] Direct-Mapped and Annex Caches

a) Cache diagram

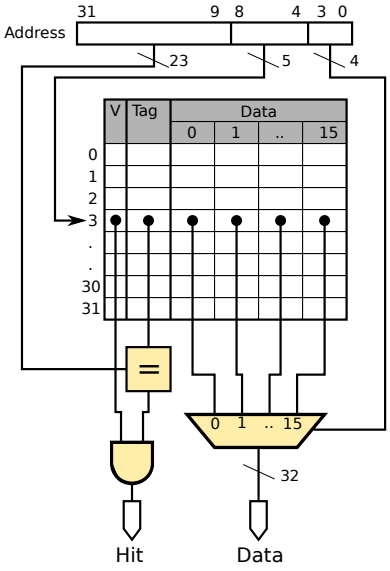


Figure 89: Direct Mapped Cache

b) Conflict between nested loops

1.1) $(A_{\text{M}} A_{\text{H}}^2 B_{\text{M}})^{10}$

2.2) miss rate $= \frac{20}{40} = 50\%$

3.3) $t_{\text{AVG}} = t_{\text{C}} +$ miss rate $\times\, t_{\text{MEM}} = 1 + 0.5 \times 100 = 51$ cycles

4.4) If the cache associativity is increased (higher than one), then both A and B can reside simultaneously in the cache. The miss rate will then be $\frac{2}{40} = 5\%$.

c) Annex cache

c.1)
- $(A_{\text{M}} B_{\text{M}})(A_{\text{H}} B_{\text{H-A}})^9$
- $N_{\text{H}} = 9$
- $N_{\text{H-A}} = 9$

c.2)
- $(A_{\text{M}} A_{\text{H}}^2 B_{\text{M}} B_{\text{H-A-SWAP}})(A_{\text{H-A}} A_{\text{H-A-SWAP}} A_{\text{H}} B_{\text{H-A}} B_{\text{H-A-SWAP}})^9$
- $N_{\text{H}} = 11$
- $N_{\text{H-A}} = 37$

c.3) $\frac{1}{N_{\text{H}}+N_{\text{H-A}}+N_{\text{M}}+N_{\text{A}}}[N_{\text{H}} \cdot t_{\text{C}} + N_{\text{H-A}} \cdot t_{\text{A}} + N_{\text{M}} \cdot (t_{\text{A}} + t_{\text{MEM}}) + N_{\text{A}} \cdot (t_{\text{A}} + t_{\text{MEM}})] =$
$= \frac{1}{N_{\text{H}}+N_{\text{H-A}}+N_{\text{M}}+N_{\text{A}}}[N_{\text{H}} + 10\ N_{\text{H-A}} + 110\ N_{\text{M}} + 110\ N_{\text{A}}]$

[Exercise 17] Cache Memory

Consider a system with two levels of cache memories. In the first level (L1 cache), there are two cache memories, one for instructions only (L1-I cache) and one for data only (L1-D cache). In the second level (L2 cache) there is only one cache memory, common for both instructions and data.

The caches are physically addressed. Addresses are 32 bits wide. The most significant bit (MSb) of the address determines whether that address corresponds to an instruction word (when MSb = 0) or to a data word (when MSb = 1). A word contains four bytes. The system uses byte addressing. On eviction, the least recently used replacement policy is applied.

L1 caches are both two-way set-associative, having 64 lines per way and four words per line. L2 cache is direct-mapped, having 512 lines and four words per line.

The cache hierarchy is exclusive, meaning that a piece of information, if present in the cache, can be found in either the L1 or the L2 cache, but not in both. When an instruction (respectively, data) is found in the L1-I (respectively, L1-D) cache, we have an L1 cache hit. If an instruction (respectively, data) is not found in the L1-I cache (respectively, L1-D cache), it is searched for in the L2 cache:

- In case of an L2 miss, the instruction (resp. data) is brought from the main memory to the L1-I (resp. L1-D) cache. If the target L1 cache line is already occupied (valid), its content is moved to L2 cache, possibly evicting a valid cache line from the L2 cache.
- In case of an L2 hit, the instruction (resp. data) is brought to L1 cache from the L2 cache. Since the cache is exclusive, the corresponding line in L2 cache is invalidated (valid bit is reset). If the target L1 cache line was already occupied (valid), its content is moved to L2 cache, possibly evicting an existing valid cache line from the L2 cache.

a) Draw a detailed diagram of the L1-I or the L1-D cache (one is sufficient), labelling clearly the width of all fields and signals.

b) Draw a detailed diagram of the L2 cache, labelling clearly the width of all fields and signals.

c) Assuming the sequence of 20 memory accesses listed in Table 14 and that the caches are initially empty, what is the content of the cache memories in the following moments:

c1) After the first eight memory accesses?

c2) After the first 16 memory accesses?

c3) After all 20 memory accesses?

Access number	Address
1	0x00000040
2	0x800064A0
3	0x07771040
4	0x977764A0
5	0x01112880
6	0x81114CB0
7	0x08883880
8	0x98884CB0
9	0x00000040
10	0x977764A0
11	0x01112880
12	0x98884CB0
13	0x02220040
14	0x822264A0
15	0x09992880
16	0x99994CB0
17	0x07771040
18	0x800064A0
19	0x08883880
20	0x81114CB0

Table 14: Sequence of memory accesses.

To answer these questions, please fill in Tables 15, 16, 17, 18 and 19 at the end of the assignment.

Address	Instruction (I) or Data (D)?	Line index in L1	Line index in L2
0x00000040			
0x800064A0			
0x07771040			
0x977764A0			
0x01112880			
0x81114CB0			
0x08883880			
0x98884CB0			
0x02220040			
0x822264A0			
0x09992880			
0x99994CB0			

Table 15: For each unique address, indicate whether it corresponds to an instruction or a data word by writing I or D respectively, and to which line index it maps in the respective L1 cache or in the L2 cache.

Access number	Address	L1-D hit?	L1-I hit?	L2 hit?	Miss?
1	0x00000040				
2	0x800064A0				
3	0x07771040				
4	0x977764A0				
5	0x01112880				
6	0x81114CB0				
7	0x08883880				
8	0x98884CB0				
9	0x00000040				
10	0x977764A0				
11	0x01112880				
12	0x98884CB0				
13	0x02220040				
14	0x822264A0				
15	0x09992880				
16	0x99994CB0				
17	0x07771040				
18	0x800064A0				
19	0x08883880				
20	0x81114CB0				

Table 16: For each memory access request, indicate whether it is a hit in one of the caches or a miss in all of them by ticking the respective column.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040		
0x800064A0		
0x07771040		
0x977764A0		
0x01112880		
0x81114CB0		
0x08883880		
0x98884CB0		

Table 17: For each unique address in the first eight memory access requests, indicate in which cache and line index the respective word is located **after the first eight requests**, or None and N/A if it is no longer in any of the caches.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040		
0x800064A0		
0x07771040		
0x977764A0		
0x01112880		
0x81114CB0		
0x08883880		
0x98884CB0		
0x02220040		
0x822264A0		
0x09992880		
0x99994CB0		

Table 18: For each unique address in the first 16 memory access requests, indicate in which cache and line index the respective word is located **after the first 16 requests**, or None and N/A if it is no longer in any of the caches.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040		
0x800064A0		
0x07771040		
0x977764A0		
0x01112880		
0x81114CB0		
0x08883880		
0x98884CB0		
0x02220040		
0x822264A0		
0x09992880		
0x99994CB0		

Table 19: For each unique address, indicate in which cache and line index the respective word is located **after all 20 memory access requests**, or None and N/A if it is no longer in any of the caches.

[Solution 17] Cache Memory

a) The structure of one of the L1 caches is shown below in Figure 90. Inside each L1 cache, the MSb of the address has always the same value, hence it does not need to be stored inside the tag. The MSb of the input address is used downstream to select whether the hit/data signals should be taken from the L1-D or L1-I cache (not shown in the figure).

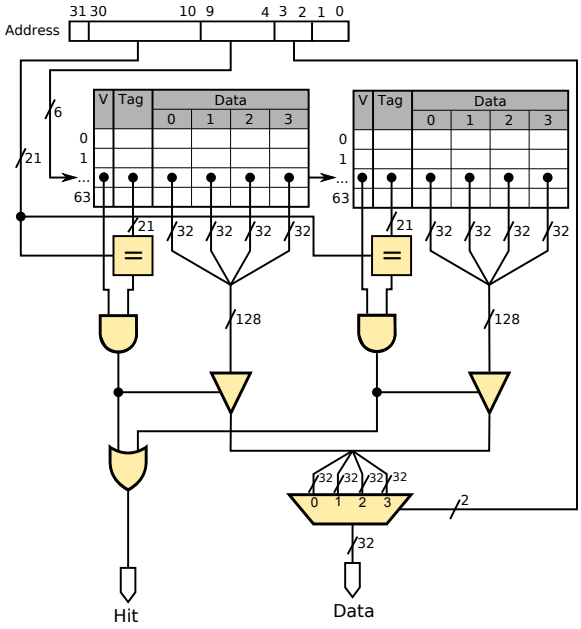


Figure 90: Structure of one of the L1 caches.

b) The structure of the L2 cache is shown below in Figure 91. Because the L2 cache is shared by instructions and data, the MSb of the address has to be included in the tag.

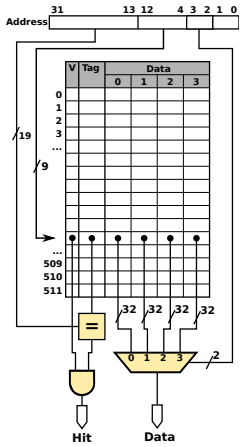


Figure 91: Structure of the L2 cache.

c)

Address	Instruction (I) / Data (D)	Line index in L1	Line index in L2
0x00000040	I	0x4	0x4
0x800064A0	D	0xA	0x4A
0x07771040	I	0x4	0x104
0x977764A0	D	0xA	0x4A
0x01112880	I	0x8	0x88
0x81114CB0	D	0xB	0xCB
0x08883880	I	0x8	0x188
0x98884CB0	D	0xB	0xCB
0x02220040	I	0x4	0x4
0x822264A0	D	0xA	0x4A
0x09992880	I	0x8	0x88
0x99994CB0	D	0xB	0xCB

Table 20: For each unique address, indicate whether it corresponds to an instruction or a data word by writing I or D respectively, and what would be its line index in the respective L1 cache or in the L2 cache.

Access number	Address	L1-D hit?	L1-I hit?	L2 hit?	Miss?
1	0x00000040				x
2	0x800064A0				x
3	0x07771040				x
4	0x977764A0				x
5	0x01112880				x
6	0x81114CB0				x
7	0x08883880				x
8	0x98884CB0				x
9	0x00000040		x		
10	0x977764A0	x			
11	0x01112880		x		
12	0x98884CB0	x			
13	0x02220040				x
14	0x822264A0				x
15	0x09992880				x
16	0x99994CB0				x
17	0x07771040			x	
18	0x800064A0			x	
19	0x08883880			x	
20	0x81114CB0			x	

Table 21: For each request, indicate whether it is a hit in one of the caches or a miss in all of them by ticking the respective column.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040	L1-I	0x4
0x800064A0	L1-D	0xA
0x07771040	L1-I	0x4
0x977764A0	L1-D	0xA
0x01112880	L1-I	0x8
0x81114CB0	L1-D	0xB
0x08883880	L1-I	0x8
0x98884CB0	L1-D	0xB

Table 22: For each unique address in the first eight requests, indicate in which cache and line index the respective word is located **after the first eight requests**, or None and N/A if it is not anymore in any of the caches.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040	L1-I	0x4
0x800064A0	L2	0x4A
0x07771040	L2	0x104
0x977764A0	L1-D	0xA
0x01112880	L1-I	0x8
0x81114CB0	L2	0xCB
0x08883880	L2	0x188
0x98884CB0	L1-D	0xB
0x02220040	L1-I	0x4
0x822264A0	L1-D	0xA
0x09992880	L1-I	0x8
0x99994CB0	L1-D	0xB

Table 23: For each unique address in the first 16 requests, indicate in which cache and line index the respective word is located **after the first 16 requests**, or None and N/A if it is not anymore in any of the caches.

Address	Current location L1-I/L1-D/L2/None	Line index (N/A if not in cache)
0x00000040	L2	0x4
0x800064A0	L1-D	0xA
0x07771040	L1-I	0x4
0x977764A0	L2	0x4A
0x01112880	L2	0x88
0x81114CB0	L1-D	0xB
0x08883880	L1-I	0x8
0x98884CB0	L2	0xCB
0x02220040	L1-I	0x4
0x822264A0	L1-D	0xA
0x09992880	L1-I	0x8
0x99994CB0	L1-D	0xB

Table 24: For each unique address, indicate in which cache and line index the respective word is located **after all 20 requests**, or None and N/A if it is not anymore in any of the caches.

[Exercise 18] Caches and Virtual Memory

Consider a system whose memory hierarchy includes a virtual memory with the following characteristics:

- 32-bit virtual addresses
- 32-bit physical addresses
- 32-bit data
- `0x800` byte pages
- 32-bit (4 bytes) page table entries
- No TLB
- Byte addressing

The system has a cache (that is not used to access the page table). The physical address is used to access the cache. Consider two cache types:

1. Direct-mapped with 16 lines and 4 word blocks (4 $\times$ 32-bit)
2. 4-way set associative with 4 lines and 4-word blocks (4 $\times$ 32-bit)

a) Draw a diagram of the memory hierarchy for both cache types.

b) Determine the access sequence (virtual addresses) generated by executing the first iterations of the following code (consider both instructions and data). You can find the page table on Table 26.

```
0x1000  main: lw   t2, 0x3000(zero)
0x1004        lw   t4, 0x3004(zero)
0x1008        lw   t3, 0x3008(zero)

0x100C  loop: beq  t2, zero, next
0x1010        lw   a0, 0 (t4)
0x1014        jal  ra, square
0x1018        sw   a0, 0(t3)
0x101C        addi t3, t3, 4
0x1020        addi t4, t4, 4
0x1024        addi t2, t2, -1
```

```
0x1028          j     loop
          next:
...
0x2000   square:mul a0, a0, a0
0x2004          ret
```

Address	Value
0x3000	0x0000 0002
0x3004	0x0000 3110
0x3008	0x0000 3210
...	...
0x3110	...
...	...
0x3210	...
...	...

Table 25: Data in memory

c) Determine the corresponding sequence of physical addresses obtained by simulating the address translation in the virtual memory. The page table of this program resides in the main memory starting at address 0 (first physical page), the operating system has thus set the value of the `Page Table Base` register to `0x0000`. The state of the page table is given below.

Address	V	Value
0x0000	1	0x0E
0x0004	0	0x11
0x0008	1	0x06
0x000C	1	0x0C
0x0010	1	0x07
0x0014	0	0x0D
0x0018	1	0x08

Table 26: Page table

d) Simulate both caches using the access sequence found at the preceding question. Determine the hit rate in both cases and show the state of each cache at the end of the simulation. Which cache has a better performance ? Why ?

[Solution 18] Caches and Virtual Memory

a) The memory hierarchy is shown in the following diagram:

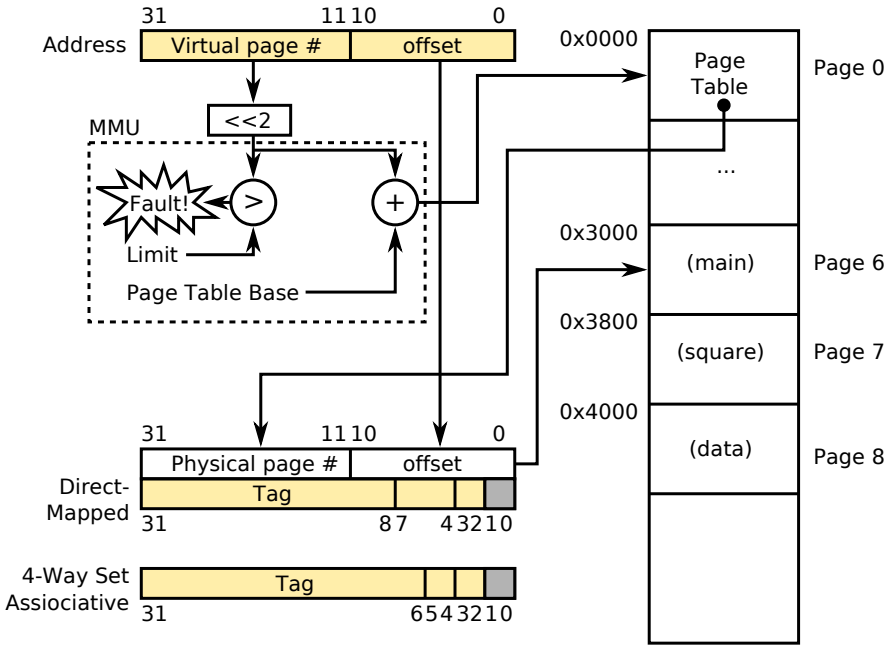


Figure 92: Schematic of the virtual memory addressing

b) and **c)**

Memory access sequence:

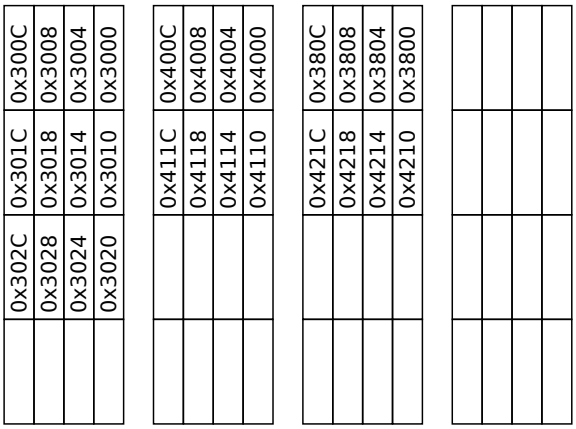
	Virtual address	Physical address	Direct-mapped	4-way set assoc.
1	0x1000	0x3000	Miss	Miss
2	0x3000	0x4000	Miss	Miss
3	0x1004	0x3004	Miss	Hit
4	0x3004	0x4004	Miss	Hit
5	0x1008	0x3008	Miss	Hit
6	0x3008	0x4008	Miss	Hit
7	0x100C	0x300C	Miss	Hit
8	0x1010	0x3010	Miss	Miss
9	0x3110	0x4110	Miss	Miss
10	0x1014	0x3014	Miss	Hit
11	0x2000	0x3800	Miss	Miss
12	0x2004	0x3804	Hit	Hit
13	0x1018	0x3018	Hit	Hit
14	0x3210	0x4210	Miss	Miss
15	0x101C	0x301C	Miss	Hit
16	0x1020	0x3020	Miss	Miss
17	0x1024	0x3024	Hit	Hit
18	0x1028	0x3028	Hit	Hit
19	0x100C	0x300C	Miss	Hit
20	0x1010	0x3010	Hit	Hit
21	0x3114	0x4114	Miss	Hit
22	0x1014	0x3014	Miss	Hit
23	0x2000	0x3800	Miss	Hit
24	0x2004	0x3804	Hit	Hit
25	0x1018	0x3018	Hit	Hit
26	0x3214	0x4214	Miss	Hit
27	0x101C	0x301C	Miss	Hit
28	0x1020	0x3020	Hit	Hit
29	0x1024	0x3024	Hit	Hit
30	0x1028	0x3028	Hit	Hit
31	0x100C	0x300C	Miss	Hit

d) The state of the direct-mapped cache with the consecutive values of each element is given in the table below:

~~0x300C~~	~~0x3008~~	~~0x3004~~	~~0x3000~~
~~0x400C~~	~~0x4008~~	~~0x4004~~	~~0x4000~~
~~0x300C~~	~~0x3008~~	~~0x3004~~	~~0x3000~~
~~0x400C~~	~~0x4008~~	~~0x4004~~	~~0x4000~~
~~0x300C~~	~~0x3008~~	~~0x3004~~	~~0x3000~~
~~0x400C~~	~~0x4008~~	~~0x4004~~	~~0x4000~~
~~0x300C~~	~~0x3008~~	~~0x3004~~	~~0x3000~~
~~0x380C~~	~~0x3808~~	~~0x3804~~	~~0x3800~~
~~0x300C~~	~~0x3008~~	~~0x3004~~	~~0x3000~~
~~0x380C~~	~~0x3808~~	~~0x3804~~	~~0x3800~~
0x300C	0x3008	0x3004	0x3000
~~0x301C~~	~~0x3018~~	~~0x3014~~	~~0x3010~~
~~0x411C~~	~~0x4118~~	~~0x4114~~	~~0x4110~~
~~0x301C~~	~~0x3018~~	~~0x3014~~	~~0x3010~~
~~0x421C~~	~~0x4218~~	~~0x4214~~	~~0x4210~~
~~0x301C~~	~~0x3018~~	~~0x3014~~	~~0x3010~~
~~0x411C~~	~~0x4118~~	~~0x4114~~	~~0x4110~~
~~0x301C~~	~~0x3018~~	~~0x3014~~	~~0x3010~~
~~0x421C~~	~~0x4218~~	~~0x4214~~	~~0x4210~~
0x301C	0x3018	0x3014	0x3010
0x302C	0x3028	0x3024	0x3020

The hit rate is $\frac{10}{31}$.

The state of the 4-way set associative cache is given below:



The hit rate is $\frac{24}{31}$.

[Exercise 19] Understanding Virtual Memories

Consider a virtual memory system with the following properties:

- 48-bit virtual byte-addresses
- 64-Kbyte pages
- 40-bit physical byte-addresses

a) Draw the structure of the virtual-to-physical address translation scheme, labelling clearly the width of all fields and signals. How many pages are there in the physical memory?

b) If all pages are in use, how large will be the page table of each process in bytes? Assume that the valid bit and all other control and protection bits take in total 8 bits/page.

c) Add to the previous system a fully associative 4-entry TLB and redraw the structure of the translation circuit.

d) Consider the following sequence of virtual address accesses. If the TLB is empty at the beginning, which accesses are hits and which are misses? What are the tag values at the end? Assume Least Recently Used replacement policy.

0xFFFF FFFF FFFF
0x0000 1000 FFFC
0x0000 1000 FFFE
0x0000 1000 FFFF
0x0000 1001 0000
0x0000 1001 0001
0xAAAA 3200 1234
0x0000 1001 0002
0xAAAA 3200 2345
0xFFFF EF00 0000
0xFFFF EF00 0000
0xFFFF FFFF 0000
0x0000 1000 FFFC
0x0000 1001 0000

[Solution 19] Understanding Virtual Memories

a)

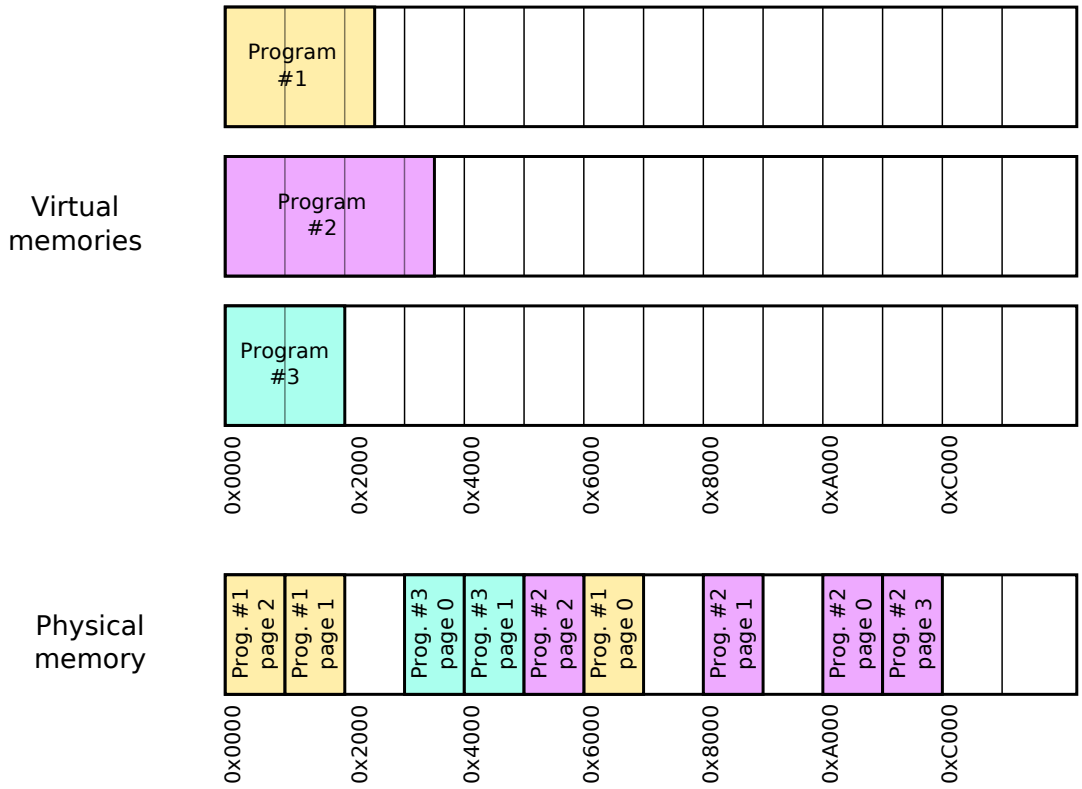


Figure 93: Virtual memory and physical paged memory

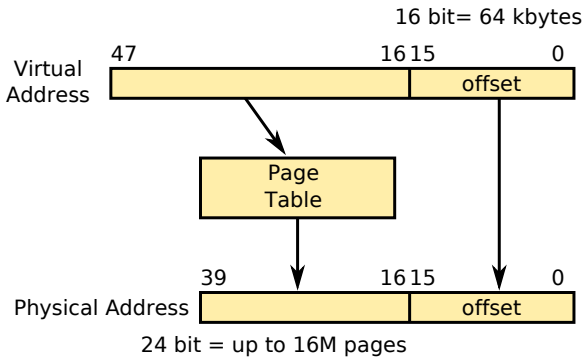


Figure 94: Translation Scheme

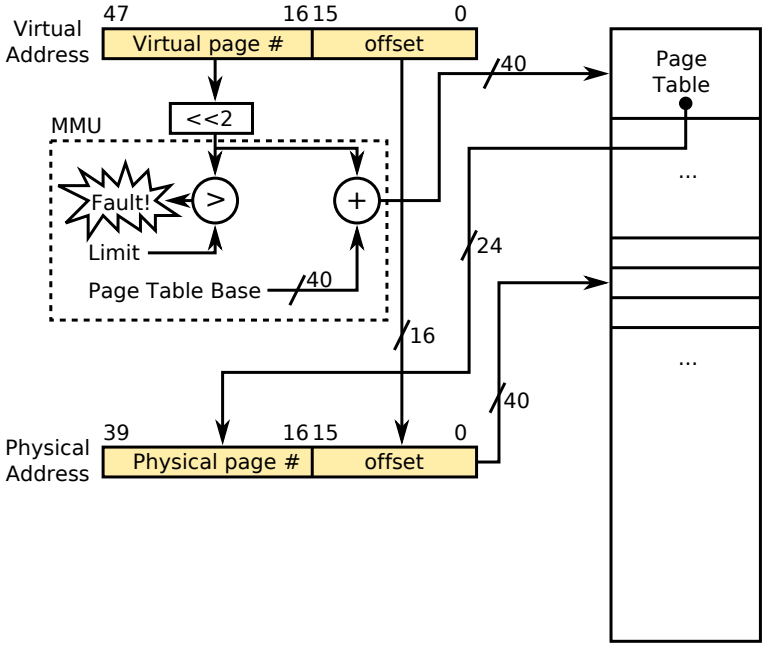


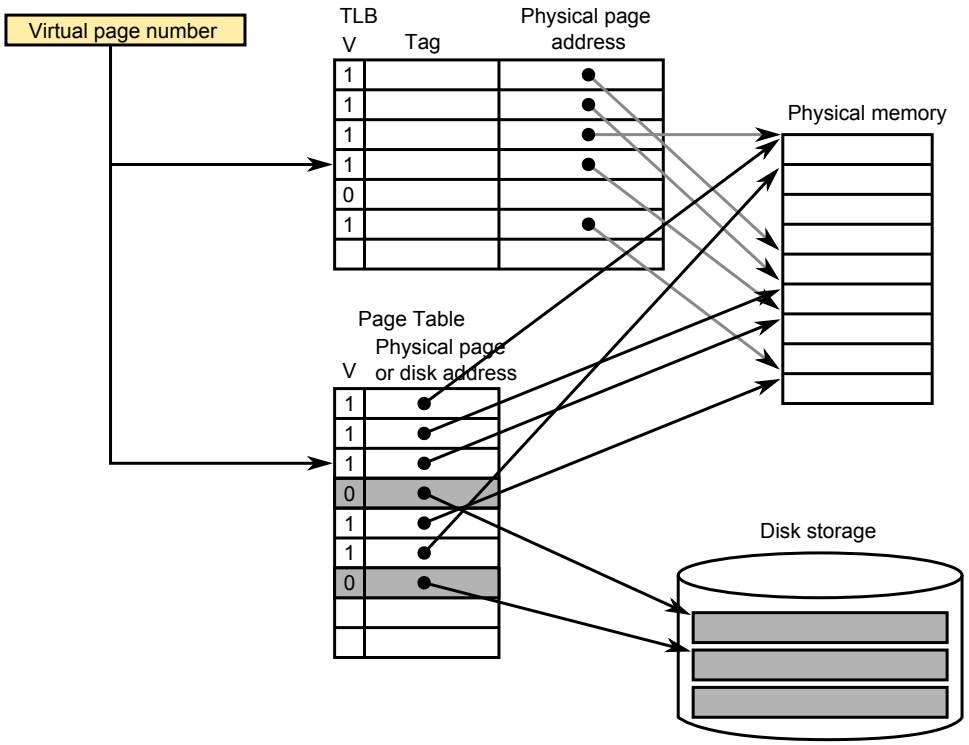
Figure 95: Virtual Address translation in paged MMU

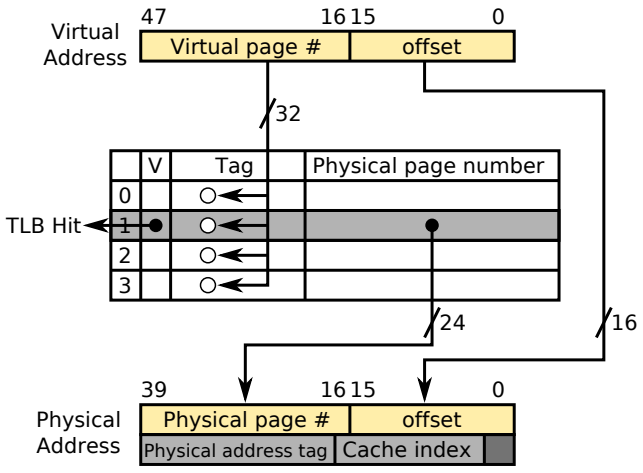
b) Each entry of the page table has 24 bits of physical page number and 8 bits of control bits. There are 4 bytes/entry.

There are 2^{32} possible virtual pages $\Rightarrow$ 4 Giga entries. 4 Bytes/entry $\cdot$ 4 Giga entries = 16 Gbytes!

Clearly unrealistic! The Page Table will need to be stored cleverly. See COD p.587 for some ideas.

c) TLB is a small cache in the CPU to avoid reading the page table on every access. The following representation is a generic structure:





d) TLB Hits and Misses

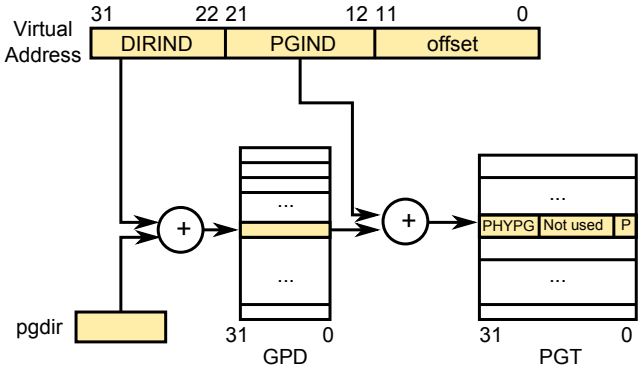
Address	Hit/Miss
0xFFFF FFFF FFFF	Miss
0x0000 1000 FFFC	Miss
0x0000 1000 FFFE	Hit
0x0000 1000 FFFF	Hit
0x0000 1001 0000	Miss
0x0000 1001 0001	Hit
0xAAAA 3200 1234	Miss
0x0000 1001 0002	Hit
0xAAAA 3200 2345	Hit
0xFFFF EF00 0000	Miss
0xFFFF EF00 0000	Hit
0xFFFF FFFF 0000	Miss
0x0000 1000 FFFC	Miss
0x0000 1001 0000	Miss

In the TLB:

0	0x0000 1001
1	0x0000 1000
2	0xFFFF FFFF
3	0xFFFF EF00

[Exercise 20] Hierarchical Virtual Memory

A RISC-V processor translates 32-bit virtual addresses (memory is byte-addressed) into 32-bit physical addresses according to the diagram below:



A special 32-bit register `pgdir` contains the base address of the GPD table (General Page Directory). A GPD entry is determined by `pgdir` and `DIRIND` (Directory Index, bits 31 to 22 of the virtual address) as shown in the diagram above, and contains the base address of a PGT table (Page Table). The PGT's address and PGIND (Page Index, bits 21 to 12 of the virtual address) determine the entry in the PGT that contains the most significant part PHYPG (Physical Page), of the physical page address. In addition to PHYPG, each entry in the PGT contains a P (Present) bit indicating the presence of the page in memory.

a) What is the size of the PHYPG field in the PGT tables? What is the size of the virtual pages in this system? What is the maximal size occupied in memory by a user program's GPD and PGT structures? **Bonus:** Discuss the advantages and drawbacks of this hierarchical translation structure by comparing it to the non-hierarchical structure of page tables.

b) To speed up address translations, the processor uses a fully associative TLB (Translation Lookaside Buffer) with 16 entries. Draw the diagram of the TLB and indicate the use and size of every field (component) of the structure.

c) The inability to translate a virtual address using the TLB generates a page fault exception. Supposing that the RISC-V processor is responsible for managing the TLB, write the code of the exception handler that finds the address of the physical page and updates the TLB. Saved registers (`s0-s7`) should not be modified. To write to the TLB, the processor uses a privileged instruction: `tlbwr r, s` (TLB write random), unavailable to normal programs. This instruction writes the contents of two ordinary RISC-V registers `r` and `s` to a randomly selected entry of the TLB. Indicate the use

of the 64 bits of `r` and `s` to load a line of the TLB according to your description in part **b)**. Register `mar` (Memory Address Register) contains the address whose access triggered a TLB miss. Registers `pgdir` and `mar` are accessible through a special instruction **`movsp`** `r, spec` that loads the value of `spec` into `r`, where `spec` is either `pgdir` or `mar` and `r` is an ordinary RISC-V register. The Operating System routine `sleep_and_swap()` loads the missing page into memory and updates the PGT. It takes as a parameter in `a0`, the address of the PGT's corresponding entry.

d) Can the replacement policy influence the performance of the system? Explain. What hardware and architecture extensions are required to implement a Least Recently Used (LRU) replacement policy?

[Solution 20] Hierarchical Virtual Memory

a) The virtual address is 32-bit wide and the offset is 12-bit wide. Given that the physical address is also 32-bit wide, the resulting size of the PHYPG field is 20 bits (32 - 12). The size of a virtual memory page in this system is $2^{12} = 4$ KB since the offset is 12-bit wide. If a program were to use the whole virtual address space, all the GPD and PGT structures would be used. The resulting occupied memory is:

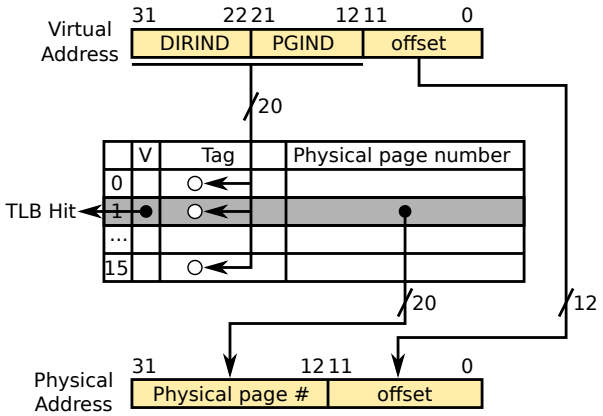
$$\text{GPD: } 2^{\text{Size(DIRIND)}} \text{ entries} \times 32 \ \tfrac{\text{bits}}{\text{entry}} = 2^{10} \times 4\text{B} = 4\text{KB}$$

$$\text{PGT: } 2^{\text{Size(DIRIND)}} \text{ tables} \times 2^{\text{Size(PGIND)}} \ \tfrac{\text{entries}}{\text{table}} \times 32 \ \tfrac{\text{bits}}{\text{entry}} = 2^{10} \times 2^{10} \times 4\text{B} = 4\text{MB}$$

Thus the occupied memory is 4KB + 4MB.

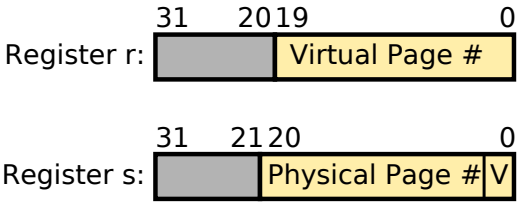
Bonus: A hierarchical translation structure is advantageous when the addressing space is used sparsely. For example, if a program only uses two consecutive pages of the virtual memory, a GPD of 4KB and a PGT of 4KB as well are enough to facilitate translations. The total size of 8KB is considerably smaller than the monolithical table of a non-hierarchical translation structure ($2^{20} \times 4\text{B} = 4\text{MB}$) for the same program. The main drawback is the additional step in the translation process (having to traverse two tables instead of one). However, with a TLB in the system, this longer translation time is visible only in case of a page fault. Therefore, this drawback does not significantly impact performance.

b) The diagram of the TLB is given in the figure below:



c) The use of the 64 bits of registers `r` and `s` is illustrated below:

The code of the interrupt handler is given below:




```
1  pg_fault:
2            addi  sp, sp, -4       #sp <- sp - 4
3            sw    ra, 0(sp)        #mem[sp] <- ra
4            movsp t0, pgdir        #t0 <- pgdir
5            movsp t1, mar          #t1 <- mar
6            srli  t2, t1, 22       #t2 <- DIRIND
7            slli  t2, t2, 2        #t2 <- t2 * 4
8            add   t2, t0, t2       #t2 <- t0 + t2
9            lw    t0, 0(t2)        #t0 <- GPD[DIRIND]
10           srli  t4, t1, 12       #t4 <- VIRTPG
11           andi  t2, t4, 0x3ff    #t2 <- PGIND
12           slli  t2, t2, 2        #t2 <- t2 * 4
13           add   t2, t0, t2       #t2 <- t0 + t2
14           lw    t0, 0(t2)        #t0 <- PGT[PGIND]
15           andi  t3, t0, 0x1      #test P bit
16           bne   t3, zero, skip   #skip if not 0
17           add   a0, t2, zero     #a0 <- t2
18           # We need to store t0 and t4 on the stack since we
19           # don't know if the following function will change
20           # them or not. By convention t0 and t4 are
21           # temporary and could be changed
22           addi  sp, sp, -8               #sp <- sp - 8
23           sw    t0, 0(sp)        #mem[sp] <- t0
24           sw    t4, 4(sp)        #mem[sp + 4] <- t4
25           jal   ra, sleep_and_swap #call function
26           lw    t0, 0(sp)        #t0 <- mem[sp]
27           lw    t4, 4(sp)        #t4 <- mem[sp + 4]
28           addi  sp, sp, 8        #sp <- sp + 8
29 skip:
30           srli  t0, t0, 12       #t0 <- t0 >> 12
31           slli  t0, t0, 1        #t0 <- t0 << 1
32           ori   t0, t0, 1        #t0 <- PHYPG | V
33           tlbwr t4, t0           #TLB update
34           lw    ra, 0(sp)        #ra <- mem[sp]
35           addi  sp, sp, 4        #sp <- sp + 4
```

```
36          ret
```

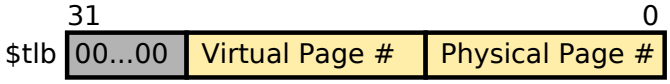
d) The performance of the system is influenced by the replacement policy in use. For example, in a random replacement policy, an entry that is about to be used can be evicted from the TLB right before it is accessed.

A Least Recently Used policy can be implemented by adding a counter for each line of the TLB. The counters are incremented at each clock cycle. The counter of an entry (line) is reset in case of a successful translation (TLB hit). If a miss occurs, the line with the highest counter value is evicted from the TLB. Comparing the counter values can be done either in hardware with a specialized circuit, or in software by allowing the programmer to access the registers of the counters.

[Exercise 21] TLB Miss Procedure

Consider a RISC-V system with virtual memory. Virtual addresses are 32-bit wide, byte addressing is used, the physical memory has a size of 64 MB and pages are 64 KB in size. Translation of virtual addresses for each access is done with the help of a TLB (Translation Look-aside Buffer) that has 8 entries. When the address cannot be translated by the TLB, an exception is generated and control is relinquished to the operating system.

This RISC-V processor has an additional 3 special registers, a memory access register `mar` that contains the address of the last memory access, a `pt` register that points on the current program's page table and a `tlb` register to access the TLB. These special registers are 32-bit wide. The `tlb` register is write-only, i.e., its contents cannot be read, it can only be used to update the TLB, e.g., in case of a TLB miss. The replacement policy is implemented in hardware. Writing to the `tlb` register writes its value to the TLB at a location chosen by the replacement policy. The bits of register `tlb` are used as follows:



The size of an entry in the TLB is smaller than 32 bits, thus some of the most significant bits must be set to '0' by the programmer. The three special registers can be accessed with an instruction called **spmov**, as shown in the following examples:

```
1    spmov  tlb, t0
2    spmov  t0, mar
```

a) Draw the structure of the TLB and the page table. Indicate the size of all fields, supposing that 6 bits are used for control in the page table (including the validity bit and the dirty bit).

b) Supposing that all pages of the program that generated the exception are in the physical memory, write an operating system RISC-V procedure that handles the TLB miss, i.e., accesses the page table and updates the TLB.

The following elements must be taken into consideration:

- The operating system has already determined what caused the exception, the procedure to be written only handles the TLB miss

- All registers must be preserved
- Register `ra` contains the return address
- The stack can be used
- Ignore the page table control bits

c) Give a precise list of all that needs to be done when the desired page is not in memory. What is the use of the dirty bit?

d) Suppose two identical programs with different data are executed in the system. To optimize the use of physical memory, the operating system could map certain pages on the same physical area. What are these pages? Explain.

[Solution 21] TLB Miss Procedure

a) Figure 96 shows the structure of the TLB. In case of a TLB miss, figure 97 shows how the page table is accessed and its structure.

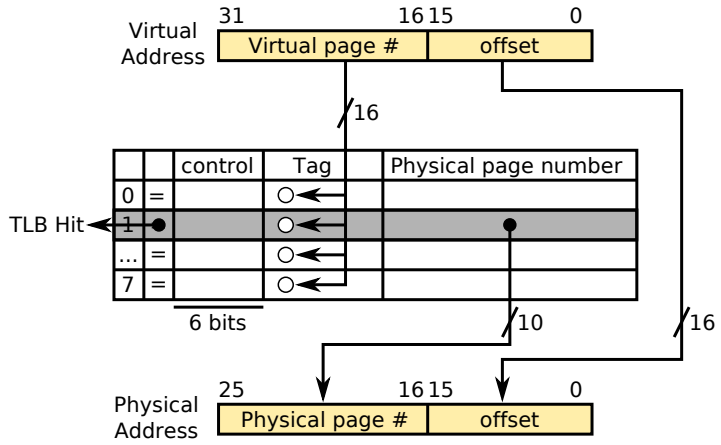


Figure 96: Structure of the TLB

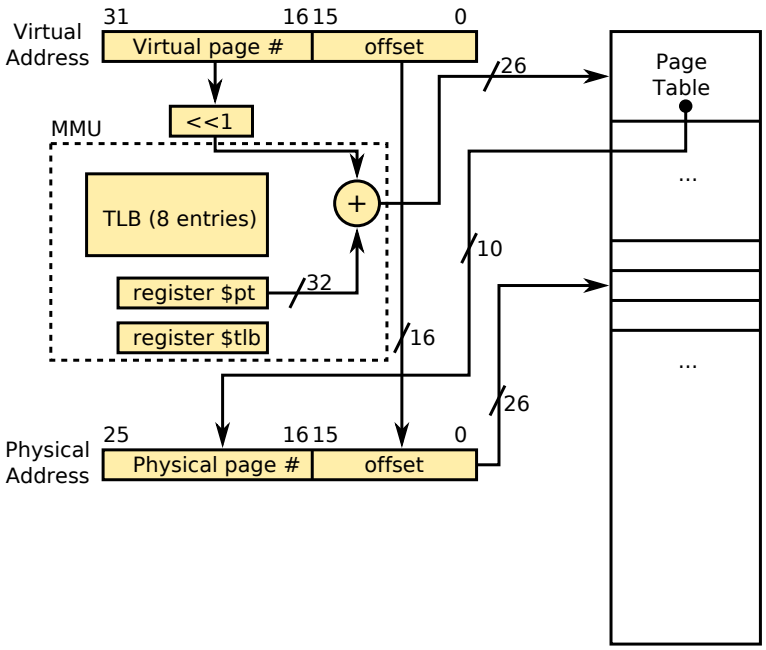


Figure 97: Structure of the page table

b) At the beginning of the procedure, the registers that will be used are saved on the stack. Register `mar` is accessed to obtain the address of the missing page which is

used to compute the address of the entry in the page table residing in main memory. This entry contains the corresponding physical page number. The virtual and physical page numbers are combined as indicated in the exercise and written to the TLB using the special register `tlb`. Before leaving the procedure, the saved registers are restored.

```
tlb_proc:
    addi  sp, sp, -16        # decrease stack pointer
    sw    t0, 12(sp)         # push t0
    sw    t1, 8(sp)          # push t1
    sw    t2, 4(sp)          # push t2
    sw    t3, 0(sp)          # push t3
tlb_manage:
    spmov t0, mar            # move fault address to t0
    spmov t1, pt             # get page table pointer
    srli  t0, t0, 16         # get the virtual page number
    slli  t2, t0, 1          # align to the entry size
    addi  t3, zero, 3        # t3 = 3
    xori  t3, t3, -1         # t3 = not 3
    and   t2, t2, t3         # align to the memory word
    add   t2, t1, t2         # compute the entry address
    lw    t2, 0(t2)          # t2 <- mem[t2], read table
    andi  t1, t0, 0x1        # test the last bit of index
    beq   t1, zero, aligned  # if zero, the entry is even
    srli  t2, t2, 16         # align the odd entry
aligned:
    andi  t2, t2, 0x3ff      # get the physic. page number
    slli  t0, t0, 10         # virt. page number << 10
    or    t0, t0, t2         # prepare the tlb entry
    spmov tlb, t0            # set new tlb entry
recover:
    lw    t3, 0(sp)          # pop t3
    lw    t2, 4(sp)          # pop t2
    lw    t1, 8(sp)          # pop t1
    lw    t0, 12(sp)         # pop t0
    addi  sp, sp, 16         # increase stack pointer
ret:
    ret                         # return from tlb_proc
```

c) When the desired page is not present in memory it must be loaded from the hard drive and its entry in the page table must be updated. Before doing that a page in memory must be evicted, i.e. written to the hard disk. If the selected page has been modified, i.e. its dirty bit is set to '1' in the page table, then the page is written to the hard disk so as not to lose the modifications and its state in the page table is updated, i.e. the dirty bit is set to '0' and it is indicated that it is not present in memory.

d) The codes of both programs are not modified and are identical. Thus, the virtual pages containing the code of the program could be mapped on the same physical page, reducing the amount of occupied physical memory.

[Exercise 22] Hierarchical Virtual Memory

In real virtual memory systems, it is common to use a hierarchical page table to translate virtual addresses. Hierarchical page tables have the advantage of generally occupying less memory space. The following figure shows the structure of a 3-level hierarchical translation system.

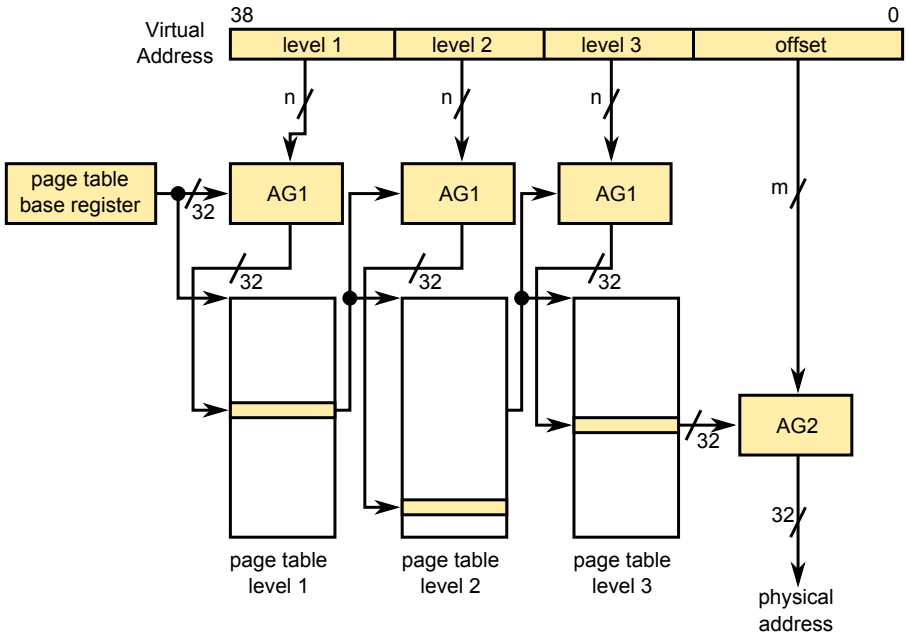


Figure 98: Structure of the virtual address

The Page Table Base Register (PTBR) contains the physical page number of the single/unique/only first level (level 1) page table. The L1 field of the virtual address serves as an index in the first-level page table to obtain the number of another physical page table that contains one of the second-level page tables. Similarly, the L2 and L3 fields serve as indexes in the second and third-level tables. Finally, the third-level page tables contain the numbers of the physical page tables that correspond to desired physical addresses and that are to be combined with the offset specified in the virtual address. Thus, to translate a virtual address, we successively access the first-level page table, then access one of the second-level page tables and finally a third-level one. There is a single first-level page table, but potentially several second and third-level page tables. The PTBR and the elements of all page tables (of any level) are 32-bit words that are organized as follows:

Consider a virtual memory system with the following characteristics:

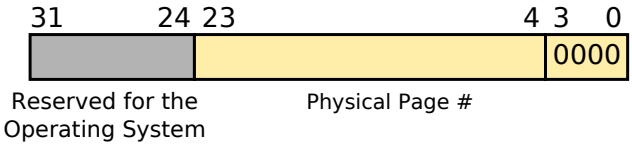


Figure 99: Organization of the 32 bits for the PTBR and page table entries

- 39-bit virtual addresses
- 32-bit physical addresses
- 4-Kbyte pages
- Byte addressing is used

a) What is the size of the `offset` field in the virtual address? What is the size n of fields L1, L2 and L3? What is the size of a page table at each level? Is it greater, smaller or equal to the size of a physical page?

b) Clearly explain the functionality of AG1 blocks and the AG2 block. For example, give their logic diagram if they were to be implemented in hardware.

The translation described in the logic diagram below is normally performed by the operating system software after a miss in the TLB.

c) Write an `addrgen1` RISC-V function that implements the functionality of the three AG1 blocks. Four registers, `a0` to `a3` are used store the function's arguments. Registers `a0` and `a1` contain the virtual address in the format described below. Register `a2` contains a physical page number in the format previously discussed and shown in figure 99. Finally, register `a3` contains the level number of the page table for which the address must be generated. Register `a0` is used by the function to return the physical address of the corresponding page table's element.

d) Write an `addrgen2` RISC-V function that implements the functionality of the AG2 block. Three registers, `a0`, `a1` and `a2` are used to store the function's arguments. Registers `a0` and `a1` contain the virtual address in the format described in the preceding question. Register `a2` contains a physical page number in the format shown in figure 99. Register `a0` is used to return the physical address corresponding to the received virtual address.

The most significant bit of the page table's elements is reserved for the operating system. When its value is '`1`', the physical page number is valid and present in main

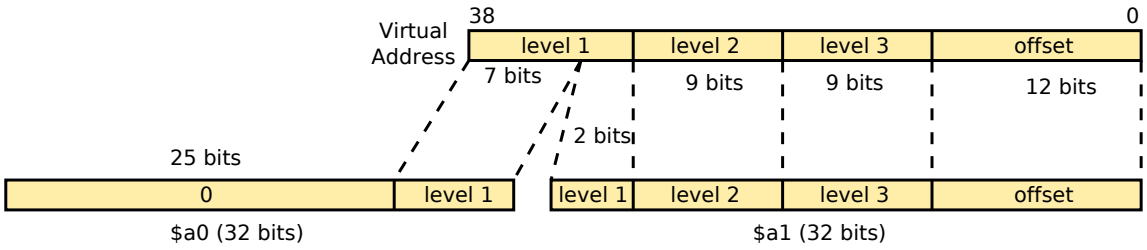


Figure 100: Distribution of the 39 bits of the virtual address in $a0 and $1

memory. If its value is '0', the translation cannot be performed.

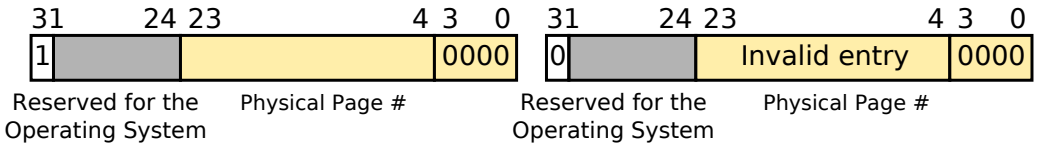


Figure 101: Usage of the first bit for the entries information

e) Write a `translate` RISC-V function that performs a complete translation of virtual addresses into physical ones, using functions `addrgen1` and `addrgen2`. Arguments are passed to the function through registers `a0`, `a1` and `a2`. The first two contain the virtual address in the format described in part **c)**, while the third register contains the value of the PTBR. The function uses register `a0` to return the physical address corresponding to the received virtual address if the translation is possible. The function returns the value '1' in register `a1` in the case of a successful translation, and the value '0' in case of a page fault.

[Solution 22] Hierarchical Virtual Memory

a) i) The size of the offset field is determined by the size of a physical page. Since the offset field must be capable of addressing each byte in every physical page (byte addressing), it must have enough bits to cover an entire physical page. The size of a physical page is 4 KB, hence the offset must have 12 bits to be able to address each byte in a page.

ii) The size n of fields L1, L2 and L3 can be computed from the size of the virtual address and the size of the offset field. Given that the sizes of all levels are equal, we can compute the value as follows:

$$\begin{aligned} n &= \frac{size(virtual\ address) - size(offset)}{3} \\ &= \frac{(39-12)}{3} \\ &= 9 \end{aligned}$$

iii) The size of a page table is the space occupied by all its entries. Thus, we need to know the size of each entry and the number of entries per page table.

- The entries of all page tables (of all levels) are 32-bit wide (4 bytes)
- 9 bits of the virtual address (fields L1, L2 and L3) are used to point on an entry in the corresponding page table. 9 bits allow pointing on 2^9 entries

Thus, the size of a page table is the total size of its entries, i.e. $2^9 \times 4 = 2^{11}\ \text{bytes} = 2\ \text{Kbytes}$. Given that the size of a physical page is 4 Kbytes, the size of a page table is half the size of a physical page.

Note: Given that the beginning of all page tables is pointed on by a physical page number, half of the pages where the page tables reside are not used.

b) The functionality of the AG1 and AG2 blocks simply consists of concatenating some fields of their inputs in order to generate a physical address. Their functionality is illustrated by the following two figures.

An AG1 block, shown in figure 102 generates the physical address of a page table entry. The physical page number is provided by the 20-bit field of the first block input. The

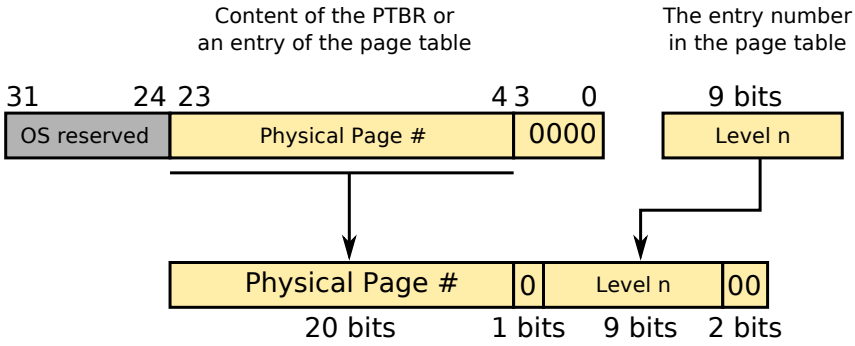


Figure 102: Structure of an AG1 block

second input provides the page table entry number. In order to find the entry offset, the latter input must be multiplied by the size, in bytes, of a page table entry, i.e. by 4. That's why the 9 bits of the second input are shifted 2 bits to the left when constructing the physical address.

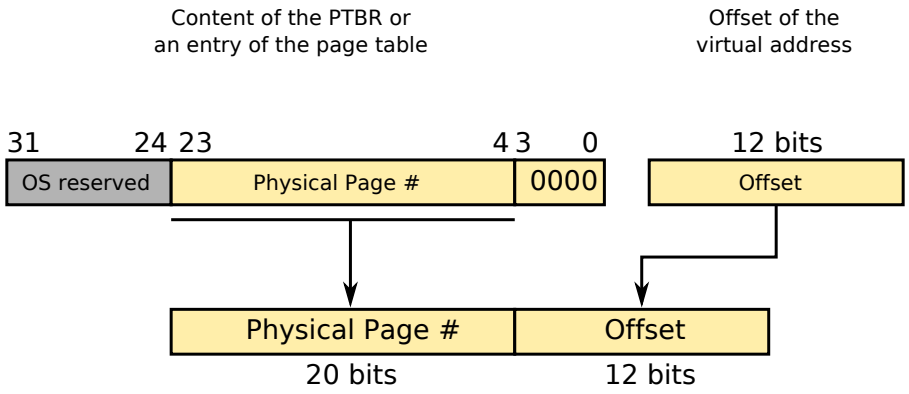


Figure 103: Structure of the AG2 block

The AG2 block, shown in figure 103 concatenates the physical page number corresponding to the virtual address with the offset in the same page (the `offset` field of the virtual address)

c) Figure 104 shows the steps taken to find the physical address. The corresponding RISC-V code is given next.

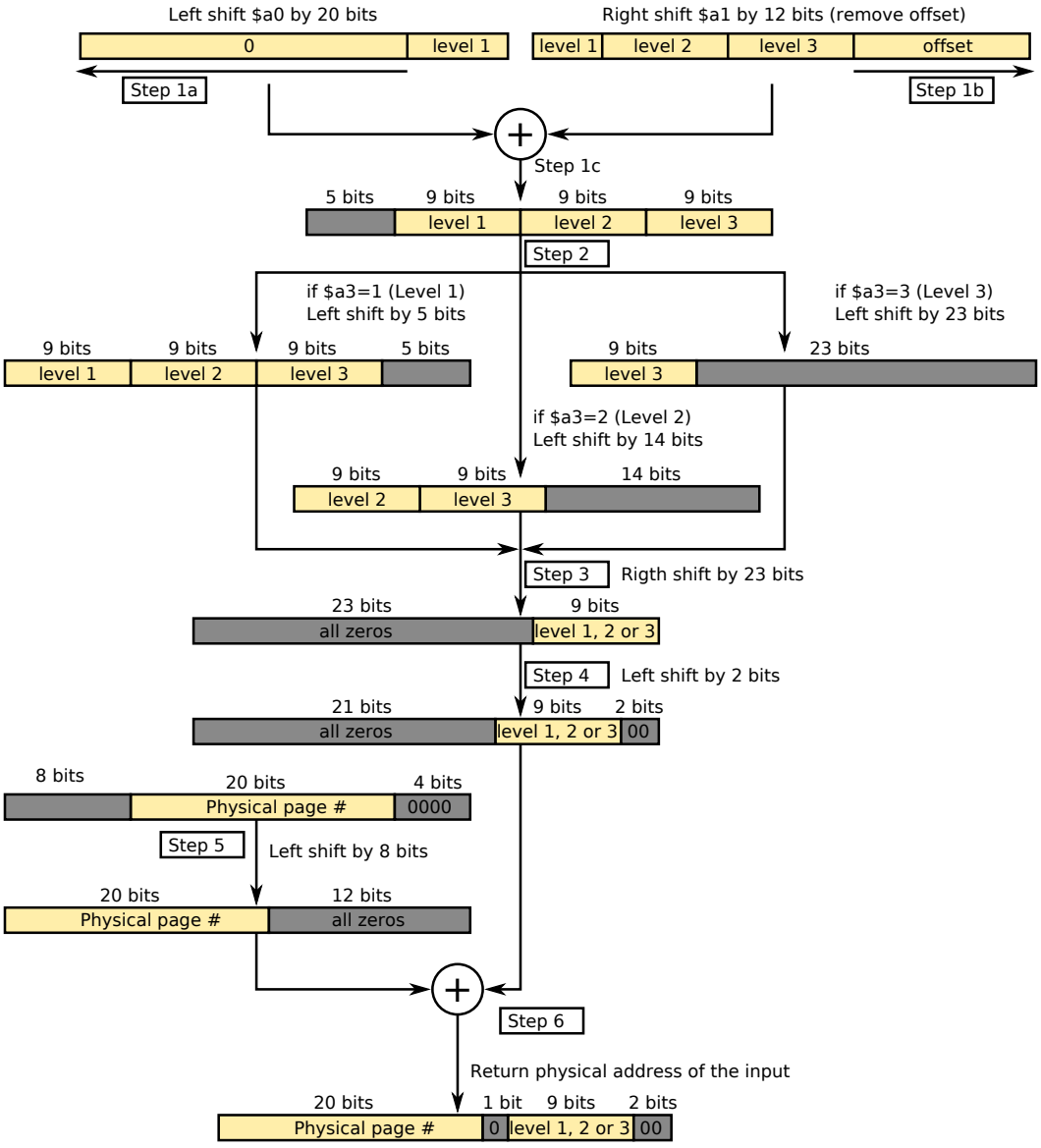


Figure 104: Algorithm steps of the AG1 block

```
addrgen1: slli t0, a0, 20                # Step 1a
          srli t1, a1, 12        # Step 1b
          or   t0, t1, t0        # Step 1c
try_lvl1: addi t1, a3, -1
          bne  t1, zero, try_lvl2
level1:   slli t2, t0, 5         # Step 2, if level 1
          j   entry_no
try_lvl2: addi t1, t1, -1
          bne  t1, zero, level3
level2:   slli t2, t0, 14        # Step 2, if level 2
          j   entry_no
level3:   slli t2, t0, 23        # Step 2, if level 3
entry_no: srli t2, t2, 23        # Step 3
          slli t1, t2, 2         # Step 4
          slli t0, a2, 8         # Step 5
          or   a0, t0, t1        # Step 6
          ret                              # function returns
```

Remarks:

1. Either of the **add** and **or** operations can be used to concatenate the offset and the beginning address of the page table.
2. Given that `addrgen1` does not call any functions, there is no need to save any registers. It only uses input, output and temporary registers.
3. Steps 3 and 4 could actually be merged by performing a 21-bit shift right instead of 23. This would allow reducing the number of instructions.

d) The `addrgen2` function simply concatenates the physical page number and the offset field of the virtual address. This concatenation can be performed by either of instructions **add** and **or**. Similarly to `addrgen1`, it does not call any functions and no registers need to be saved. Its code is given below:

```
addrgen2: slli t0, a2, 8      # physical page number on msb
          li   t1, 0xFFF
          and  t1, a1, t1        # extract the offset
          or   a0, t0, t1        # concatenation
          ret                    # function returns
```

e) The `translate` function simply calls `addrgen1` three times and `addrgen2` once. Before each call, it prepares the inputs and saves the appropriate registers.

```
# Save the return address to avoid erasing
# it when the functions addrgen1 and
# addrgen2 are called
          addi sp, sp, -4
          sw   ra, 0(sp)


# t0: the actual level (counter of the level)
# t1: the number of levels+1 (so the limit value of the level
# counter. Determines the number of calls of addrgen1)
# t2: a physical page number in the format PTBR

          addi t0, zero, 1
          addi t1, zero, 4
          add  t2, zero, a2

# Verify the validity of the given physical page number in the
# format PTBR. If the MSB is 1 (if the value in the register is
# negative) the physical page number is valid. Otherwise, the
# function returns indicating a page fault.
is_valid: slt  t3, t2, zero
          beq  t3, zero, page_flt

# Prepare the input registers before calling the function
# addgen1. The registers a0 and a1 already contain the
# virtual address, nothing is needed to update them.
          add  a2, zero, t2
          add  a3, zero, t0

# Save the content of the registers t0 and t1 (they are
# used in the translate function). t2 is not saved because
# we don't need the content anymore.
          addi sp, sp, -12
                  sw  a0, 8(sp)
          sw  t0, 4(sp)
          sw  t1, 0(sp)


# Call the function addrgen1
          jal ra, addrgen1

# Stock the entry's content of the actual level in t2.
# The contents of this register will be in PTBR format.
          lw t2, 0(a0)
```

```
45
46  # Restore the value of the saved registers
47            lw  t1, 0(sp)
48            lw  t0, 4(sp)
49                  sw  a0, 8(sp)
50            addi sp, sp, 8
51
52  # Increment the level counter by 1 and if the 3rd level
53  # is not reached, stay in the loop.
54            addi t0, t0, 1
55            bne t0, t1, is_valid
56
57  # ------------------------------------------------------------
58  # --- The last step; the call of the addrgen2 function     ---
59  # ------------------------------------------------------------
60  # Verify the validity of the physical page number given in
61  # the PTBR format. IF the msb is 1 (if the value in the
62  # register is negative) the physical page number is valid.
63  # Otherwise, we return the function indicating a page fault.
64            slt  t3, t2, zero
65            beq  t3, zero, page_flt
66
67  # Prepare the input register before calling the function
68  # addrgen2. The registers a0 and a1 already contain the
69  # virtual address, there is no update to do for these
70  # registers.
71            add  a2, zero, t2
72
73  # Save the content of the registers t0 and t1 (that are used
74  # by the program). t2 will not be used anymore by the
75  # function.
76            addi sp, sp, -8
77            sw  t0, 4(sp)
78            sw  t1, 0(sp)
79
80  # Call the addgen2 function.
81            jal ra, addrgen2
82
83  # Restore the state of the saved registers.
84            lw  t1, 0(sp)
85            lw  t0, 4(sp)
86            addi sp, sp, 8
87
88
```

```
89  # -------------------------------------------------------------
90  # ------------ Return of the translate function -------------
91  # -------------------------------------------------------------
92  # The return value is already in a0 (returned by addgen2).
93  # It is not modified before the return of the function. The
94  # value 1 is put on the register v1 to indicate the success
95  # of the function.
96            addi  a1, zero, 1
97
98  end:
99  # Restore the value of the saved register at the beginning
100 # of the function and return to the function that called
101 # translate.
102           lw    ra, 0(sp)
103           addi sp, sp, 4
104           ret
105
106 # If a entry was invalid during the research, a page fault
107 # is indicated setting the v1 value to zero. Once the value
108 # is set the function returns.
109 page_flt: add  a1, zero, zero
110           j    end
```

A more compact version of the `translate` function calls `addrgen1` and `addrgen2` in the same loop. In this case, `addrgen1` is called for levels 1, 2 or 3 and `addrgen2` is called in the last iteration of the loop. The code of this version is given below. Most of the comments were removed as they are identical to the ones in the previous version.

```
1  translate: addi sp, sp, -4
2             sw  ra, 0(sp)
3
4             addi t0, zero, 1
5
6  # The limit to end the loop is now 5, because we also call
7  # the addgren2 function in the loop.
8             addi t1, zero, 5
9             add  t2, zero, a2
10 is_valid:  slt  t3, t2, zero
11            beq  t3, zero, page_flt
12
13 # We prepare the parameters and save the registers as addrgen2
14 # was called.
15            add  a2, zero, t2
16
```

```
            addi sp, sp, -12
                    sw  a0, 8(sp)
            sw  t0, 4(sp)
            sw  t1, 0(sp)

# We determine which function to call here. If the level
# number is 4, we call addrgen2, otherwise we call addrgen1.
# To check it, we decrement the limit of the loop by 1 (t1).
            addi  t1, t1, -1
            beq  t1, t0, last_step

# If we call addrgen1, all the parameters to this function must
# be added.
            add  a3, zero, t0
            jal  ra, addrgen1
            j    continue

last_step: jal  ra, addrgen2

continue:
# The following instruction is useless after a call of
# addrgen2, but it is mandatory after the function addrgen1.
            lw   t2, 0(a0)

                    lw   t1, 0(sp)
            lw   t0, 4(sp)
                    lw   a0, 8(sp)
            addi sp, sp, 12

            addi t0, t0, 1
            bne  t0, t1, is_valid

            addi a1, zero, 1
fin:        lw   ra, 0(sp)
            addi sp, sp, 4
            ret
page_flt:   add  a1, zero, zero
            j    fin
```

[Exercise 23] Virtual Memory

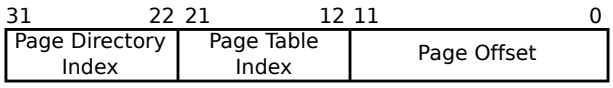
Consider a slightly simplified version of the addressing mode of the Intel IA-32 in Protected Mode. The virtual and physical addresses are 32 bits and the memory is byte-addressed. All pages are made up of 4K bytes and the translation is done in two steps. First, a special register of the processor (`CR3`) points to the Page Directory, a special page for each process which, in turn, points to 1024 pages (maximum), which compose the Page Table itself. Then, each of these pages contains 1024 pointers to the physical pages where the data is stored.

`CR3` register indicates the number of the page of the memory which contains the Page Directory in the bits 31-12 and the rest of the bits are ignored.

Similarly, each 32-bit element in the Page Directory indicates the number of the memory page that contains the corresponding Page Table in bits 31-12. A '1' in bit 0 indicates that the corresponding Page Table exists, while a '0' in this bit means all other bits have no meaning and no addresses corresponding to the address range in question has been allocated by the operating system. The other bits are not used for the translation.

In turn, each 32-bit element in the Page Table indicates the number of the memory page that contains the required data in bits 31-12. Here again, a '1' in bit 0 indicates that the corresponding page exists, while a '0' in this bit means all other bits have no meaning and no addresses corresponding to the address range in question has been allocated by the operating system. The other bits are not used for the translation.

The following figure shows the utilisation of bits of virtual address:



a) Design the translation scheme described above and clearly indicate the bits that come into play in different phases of translation.

b) Assume that `CR3 = 0x37215aef` and the physical memory contains these:

Physical address	Result of a 32bit load
0x12015470	0x7a126372
0x12315470	0x37215067
0x37215048	0x12015aef
0x37215120	0xaef12067
0x37215124	0xaef11066
0x37215480	0x12315aef
0x7a125470	0x7a315abd
0x7a125474	0x37215473
0x7a125e14	0x37215067
0x7a125384	0x7a126123
0x7a125388	0x7a125e14
0x7a126470	0xaef12345
0x7a126474	0xaef12237
0xaef120d0	0xaef12344
0xaef120d4	0x37215067
0xaef12344	0x12315067
0xaef12348	0x12015067
0xaef12d14	0x7a125001
0xaef12d18	0x37215066
0xaef12e14	0x7a126067

Find the bytes read by the following virtual addresses and explain the translation process:

Virtual address
0x12345473
0x12445473
0x12385474

Note that Intel processors are little-endian. Indicate whether any of these virtual addresses are not translatable. What would the processor or the operating system do if a program accesses these addresses while being executed?

c) A '1' in bit 4 of one element in the Page Table indicates that data from the memory page in question should never be cached: they are always read from the memory without affecting the contents of the cache. What applications do you think that such a bit could have?

d) Why do you think it is a good idea to make the reading of the table by two successive levels of indirection, as described above, rather than one? Specify at least one advantage and one disadvantage of the two-level scheme.

[Solution 23] Virtual Memory

a) The solution is provided in Figure 105.

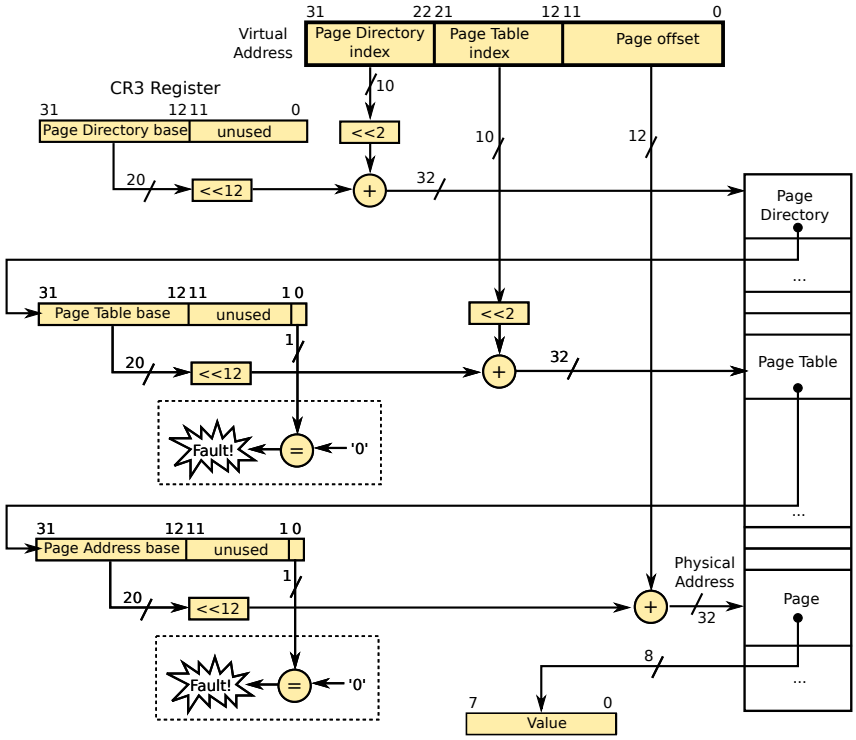


Figure 105: The virtual memory translation process.

b) The solution is provided in Figure 106.

c) This scheme could be used to prevent some data from being stored in the cache. For example, when a process which does not reuse its data uses the cache, it will cause unnecessary cache pollution.

Another case is when the data is volatile and its value can be changed by some external units. Then, this bit ensures that the most recent data is fetched from the memory whenever needed.

d) The two-level scheme enables the page table size to be kept small. With this scheme, the page directory is 4K (2^{10} entries) and each page table is 4K (2^{10} entries). Otherwise, the page table size would have been 4M (2^{20} entries). Alternatively, if the page table size had to be kept small, the page sizes would have to be increased.

However, the two-level scheme is slower as the address translation process needs more operations. This leads to performance, energy and/or silicon-area overheads.

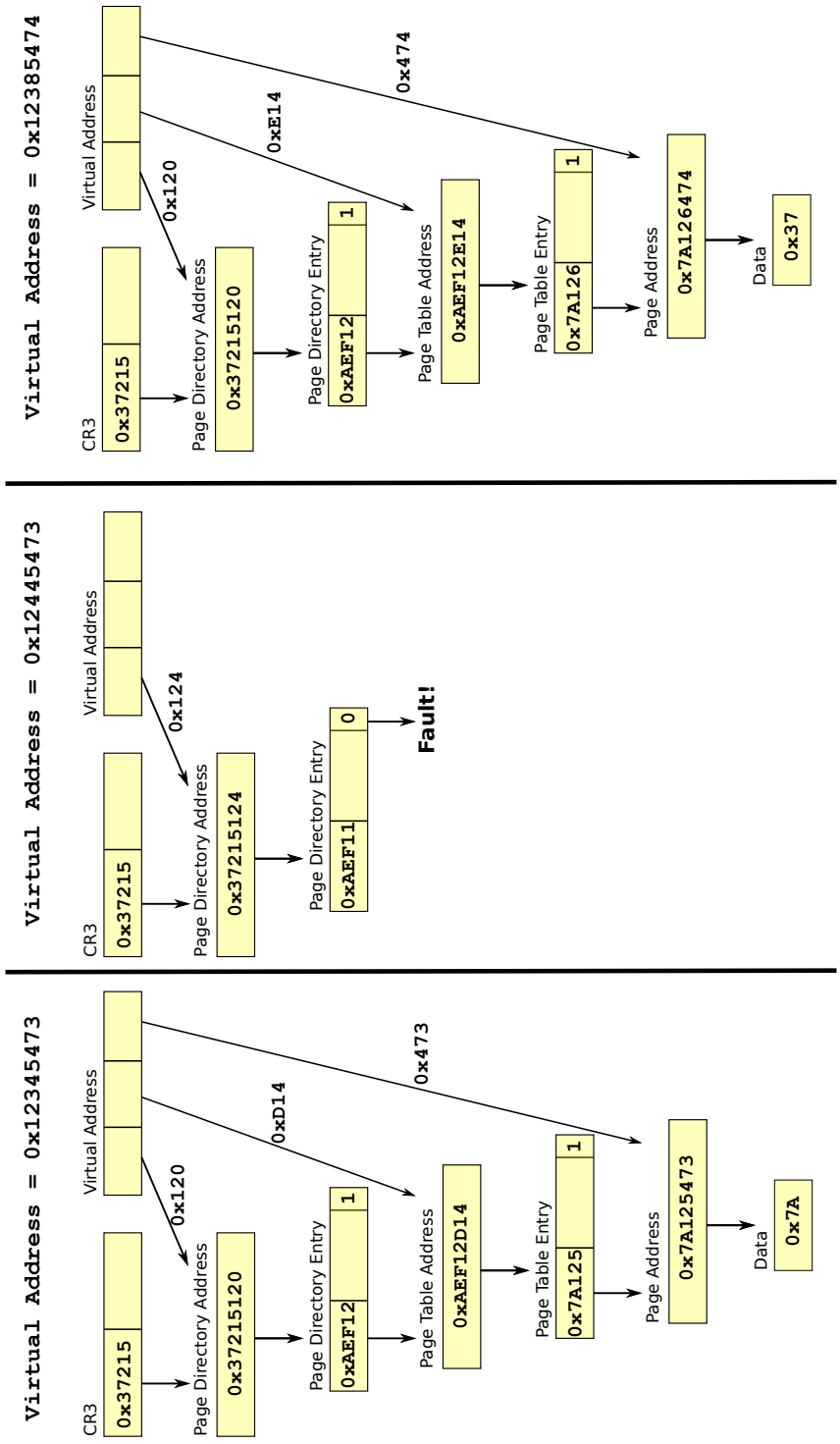


Figure 106: Address Translation

[Exercise 24] Virtual Memory

Consider a virtual memory system having the following properties: 48-bit virtual addresses, 32-bit physical addresses, and 4-Mbyte pages. The system uses **word addressing**; one word contains 4 bytes.

a) Assuming that a virtual address is interpreted as a virtual page number concatenated with an offset, as in Figure 107, answer the following questions:

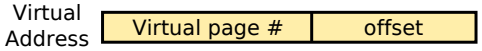


Figure 107: Structure of the virtual address

a.1) What is the size of the offset field?

a.2) How many virtual pages can there be in this system?

a.3) How many physical pages can there be in this system?

a.4) What is the size in bytes of a linear page table, supposing that one page-table entry takes one word?

a.5) Draw a diagram of the classical translation process, clearly indicating the width of all signals.

a.6) How many physical pages are needed to store one page table?

b) A different, hierarchical implementation of virtual address translation is shown in Figure 108.

To obtain a physical page number, it is required to go through two levels of indirection. The page-table-base register contains the address of the first-level page table. The **level 1** field of the virtual address serves as an index in this table. An entry in the first-level page table contains the starting address of a second-level page table. The **level 2** field indexes a second-level page table. An entry in the second-level page table contains the desired virtual-to-physical address translation. The width of the **level 2** field of the virtual address is 20 bits.

For the virtual address translation shown in Figure 108 and the system parameters listed at the beginning of the exercise, answer the following questions:

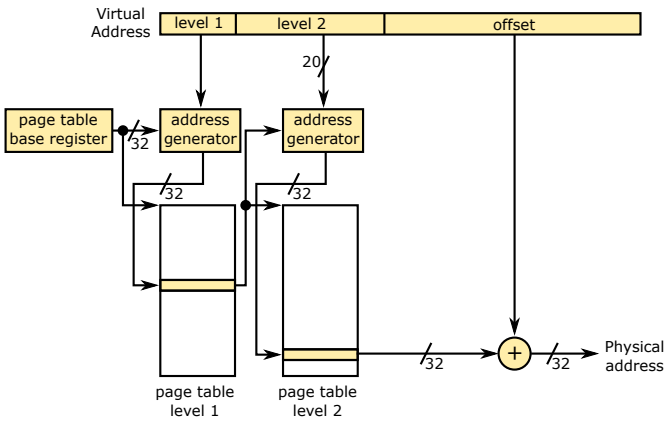


Figure 108: Hierarchical virtual address translation

b.1) Supposing that a program occupies the whole virtual address space, how many physical pages are needed to store all required first-level and second-level page tables?

b.2) Supposing that a program occupies the following range of virtual address space

- first address: 0x0800 0000 0000
- last address: 0x0BFF FFFF FFFF

how many physical pages are needed to store the required first-level and second-level page tables?

[Solution 24] Virtual Memory

Pages have a size of 4 Mbytes, i.e. 2^{22} bytes or 2^{20} words .

a) a.1) Since the memory is word-addressed, the offset must be able to index every word in a page and thus must be 20 bits wide.

a.2) The virtual address has 48 bits. Bits 47 down to 20 are used for the virtual page number. The system thus contains 2^{28} = 256 M virtual pages.

a.3) The physical address has 32 bits. Bits 31 to 20 are used for the physical page number. The system thus contains 2^{12} = 4 K physical pages.

a.4) Since a page table entry occupies one word, the size of a page table is $2^{28} \times$ 4 bytes = 1 Gbyte.

a.5) The translation process is shown below:

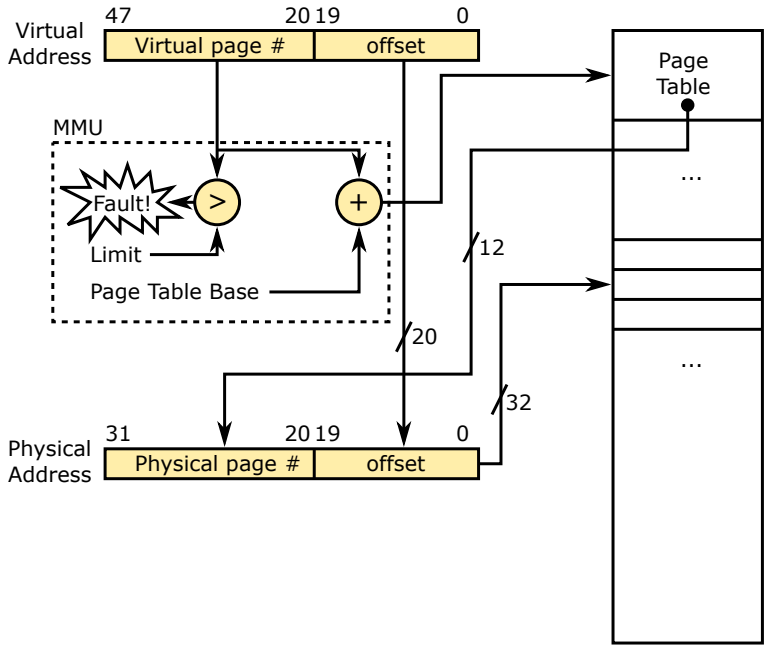


Figure 109: Virtual address translation

a.6) Since one page table takes 1 Gbyte and the page size is 4 Mbytes, a page table spans $2^{30}/2^{22} = 2^8$ = 256 pages.

b) Given that the level 2 field is 20 bits wide, the level 1 field must be 8 bits wide. Therefore, the first-level page table divides the virtual address space into 256 parts

of 2^{20} words each (the size of one page). Each part has a corresponding entry in the first-level page table.

b.1) If a program occupies the whole virtual space, one first-level page and 256 two-level pages are required. Given that each of these pages can fit in a single physical page, 257 physical pages are needed to store all required translations.

b.2) The specified virtual-address space-range spans from entry `0x8` to entry `0xB` in the first-level page. Therefore, 5 physical pages are needed to store all required translations (one first-level page plus 4 second-level pages). This amounts to 5×4 Mbytes = 20 Mbytes of physical memory space.

[Exercise 25] Virtual Memory

Consider a virtual memory system having the following properties: 56-bit virtual addresses, 32-bit physical addresses, 1-MiB pages, and byte addressing. Words contain 4 bytes.

a) Assuming that a virtual address is interpreted as a virtual page number concatenated with an offset, answer the following questions:

a.1) What determines the size of the offset field? Calculate it.

a.2) How many virtual pages can there be in this system?

a.3) What is the maximum number of physical pages this system can support?

a.4) Assuming that the control and protection bits take in a total of 20 bits per page table entry and that a program uses the entire virtual address space:

- How large would the linear page table have to be (in bytes)?
- How many physical pages would be needed to store the linear page table?
- Would the linear page table fit in the memory? Why?

b) The memory-management unit contains a translation lookaside buffer (TLB) to speed up the address translation. The system has a cache, which is accessed using physical addresses. Linear page table is used. The CPU issues a load word instruction.

b.1 List all the steps taken to load a word from the main memory to the CPU, taking into account all the events that can occur. No need to show the updates in any structure (cache, TLB).

b.2 Assuming that the complete page table is present in the main memory:

- What is the minimal number of main memory accesses required for one load word instruction?
- What is the maximal number of main memory accesses required for one load word instruction?

c) A different, hierarchical implementation of virtual address translation is shown in Figure 110. To obtain a physical address, it is required to go through two levels of indirection:

◇ The page-table-base register contains the base address of a first-level page table.

◇ The **level 1** field of the virtual address serves as an index in the first-level page table.

◇ An entry in the first-level page table contains the base address of a second-level page table.

◇ The **level 2** field (20-bit wide) of the virtual address serves as an index in the second-level page table.

◇ An entry in the second-level page table contains the base address of the desired physical page.

◇ The final physical address is obtained by adding the base address of the page with the offset.

A snippet of the memory content is shown in Table 27. For simplicity, addresses are shown as word-aligned and data is shown as words, even though the system uses byte addressing. An entry in the page table contains **only** the 32-bit base address of another page, there are **no** control nor protection bits. The first-level page table starts at the memory address 0x0000 0000.

Assume the following sequence of virtual addresses issued by the CPU: 0x00 0100 0100 0004,
0x00 0200 0020 00A4, and 0x00 0300 0010 00A0.

c.1) Find the corresponding physical addresses.

c.2) If the TLB in this system is initially empty, fully associative, having four lines and one address translation per line:

- Draw the diagram of the TLB, indicating the use of each bit.
- Show the TLB content after the given sequence of virtual addresses.

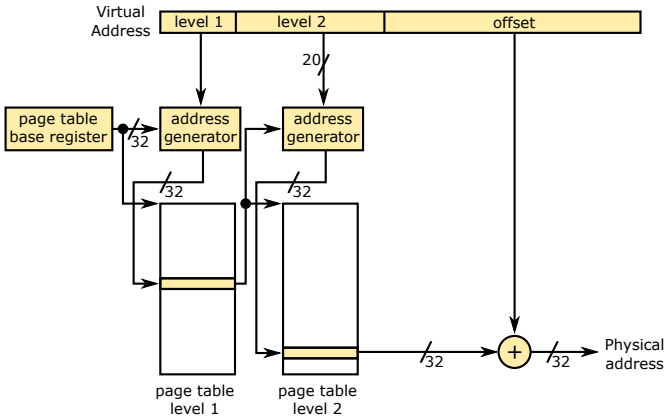


Figure 110: Hierarchical virtual address translation

0x0000 0000	0x0020 0000
0x0000 0004	0x0040 0000
0x0000 0008	0x0060 0000
0x0000 000C	0x0080 0000
...	...
0x0010 0014	0x001F FFFF
0x0010 0018	0x01FF FFFF
...	...
0x0020 00A0	0x0010 0000
0x0020 00A4	0x0090 0000
...	...
0x0030 0000	0x003F FFFF
0x0030 0004	0x03FF FFFF
...	...
0x0040 0040	0x0030 0000
0x0040 0044	0x00A0 0000
...	...
0x0050 00A0	0x005F FFFF
0x0050 00A4	0x05FF FFFF
...	...
0x0060 0008	0x0050 0000
0x0060 000C	0x00B0 0000
...	...
0x0070 000F	0x007F FFFF
0x0070 00A0	0x07FF FFFF
...	...
0x0080 0000	0x00C0 0000
0x0080 0004	0x0070 0000
...	...

Table 27: Memory content. Byte addressing is used. Word-aligned addresses are shown on the left.

[Solution 25] Virtual Memory

a) a.1) The virtual page offset is determined by the size of a memory page. Pages have a size of 1 Mbytes, i.e. 2^{20} bytes. Since the memory is byte-addressed, the offset must be able to index every byte in a page and thus must be 20 bits wide.

a.2) The virtual address has 56 bits. Bits 55 down to 20 are used for the virtual page number. The system thus contains 2^{36} = 64 G virtual pages.

a.3) The physical address has 32 bits. Bits 31 to 20 are used for the physical page number. The system thus contains 2^{12} = 4 K physical pages.

a.4) Each page table entry contains a physical page number (12 bits) and metadata (20 bits), so each entry is 32 bits or 4 bytes. The size of a page table is $2^{36} \times$ 4 bytes = 256 Gbytes.

The number of pages needed to store the page table is $2^{38}/2^{20} = 2^{18}$.

The number of pages required for a linear page table is much larger than the maximum number of physical pages the system can support. The linear page table does not fit in memory.

b) b.1)
- First we search for translation of the load's virtual address in the TLB
- If it is a TLB hit, we search the cache for the data using the translated physical address
  - If it is a cache hit, we load the data
  - If it is a cache miss, we load the data from memory
- If it is a TLB miss, we find the translation in memory, update the TLB, and then go back to the first step in this list

b.2)
- If we have both a TLB hit and a cache hit then we will require no accesses to main memory.
- If the TLB misses then we will require one memory access to read the translation. If the cache misses, then we will require one memory access to read the data. The worst case is when both TLB and cache miss, when we will require two memory accesses.

c) Given that the level 2 field is 20 bits wide, the level 1 field must be 16 bits wide.

c.1) 0x0001 00010 00004:
- Level 1 field = 0x0001
- Address of the 1^{st} level page table entry =
  = (Base address 1^{st} level page table) + (Level 1 field)$\times$(Entry size) =
  = 0x0000 0000 + 0x0001$\times$4 = 0x0000 0004

- Base address 2^{nd} level page table = 0x0040 0000
- Level 2 field = 0x00010
- Address of the 2^{nd} level page table entry =
  = (Base address 2^{nd} level page table) + (Level 2 field)$\times$(Entry Size) =
  = 0x0040 0000 + 0x00010$\times$4 = 0x0040 0040
- Base address physical page = 0x0030 0000
- Offset = 0x00004
- Physical Address = (Base address physical page) + Offset =
  = 0x0030 0000 + 0x00004 = 0x0030 0004

0x0002 00002 000A4:

- Level 1 field = 0x0002
- Address of the 1^{st} level page table entry = 0x0000 0000 + 0x0002$\times$4 = 0x0000 0008
- Base address 2^{nd} level page table = 0x0060 0000
- Level 2 field = 0x00002
- Address of the 2^{nd} level page table entry = 0x0060 0000 + 0x00002$\times$4 = 0x0060 0008
- Base address physical page = 0x0050 0000
- Offset = 0x000A4
- Physical Address = 0x0050 0000 + 0x000A4 = 0x0050 00A4

0x0003 00001 000A0:

- Level 1 field = 0x0003
- Address of the 1^{st} level page table entry = 0x0000 0000 + 0x0003$\times$4 = 0x0000 000C
- Base address 2^{nd} level page table = 0x00800000
- Level 2 field = 0x00001
- Address of the 2^{nd} level page table entry = 0x0080 0000 + 0x00001$\times$4 = 0x0080 0004
- Base address physical page = 0x0070 0000
- Offset = 0x000A0
- Physical Address = 0x0070 0000 + 0x000A0 = 0x0070 00A0

c.2) The TLB diagram and its contents are shown in Figure 111.

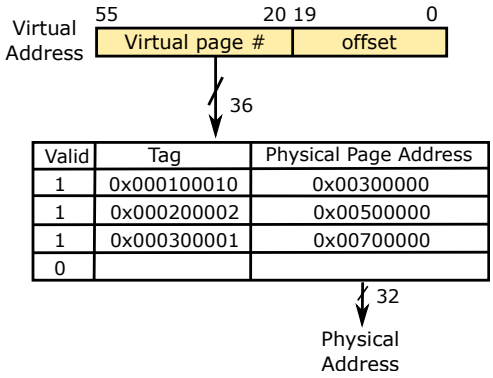


Figure 111: TLB

[Exercise 26] Virtual Memory

Consider a virtual memory system having the following properties: 64-bit virtual addresses, 32-bit physical addresses, 64 kiB pages, and word addressing. Words contain 4 bytes. The system uses a hierarchical virtual-to-physical address translation scheme. Virtual addresses have a total of five fields, as illustrated in Fig. 112:

- reserved (not in use),
- page global directory,
- page middle directory,
- page directory, and
- offset

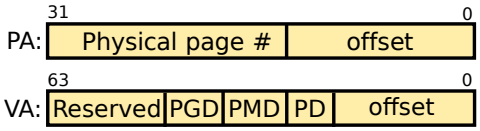


Figure 112: Structure of the virtual address.

There is only one page global directory (PGD) in the system; it occupies the entire first physical page (starting at address `0x0`). An entry in the PGD contains (in the least significant part of the entry) the physical page number of the page in memory where the corresponding page middle directory (PMD) is. An entry in the PMD contains (in the least significant part of the entry) the physical page number of the physical page in memory where the corresponding page directory (PD) is. An entry in the PD contains the physical page control bits (in the most significant part of the entry) and the physical page number (in the least significant part of the entry) of the physical page in memory where the corresponding page, containing data or instructions, is. An entry in PGD, PMD or PD occupies exactly one word. Each directory occupies an entire physical page.

a) What is the size of the offset field (in bits)?

b) How many physical pages are there in this system?

c) What is the total capacity of the physical memory (in bytes)?

d) What is the size of every of the PGD, PMD, and PD fields (in bits)?

e) Let us for a moment imagine that this system uses a linear translation scheme instead of the above-described hierarchical one. Assuming that the same number of the most significant bits of the virtual address are reserved (ignored) in both systems, how many pages would be needed for storing a linear page table?

f) Coming back to the hierarchical translation scheme and assuming that a program uses the entire virtual address space (except, of course, the reserved bits), how many physical pages would be needed for storing all the required directories?

g) Draw a detailed hierarchical virtual-to-physical address translation scheme. You can start by repeating the sketch in Fig. 113 and then continue by adding all that is missing.

h) How many pages are occupied by all the directories required by a program that uses the following virtual address space range:

- FROM `0x0000 0C00 3000 C003` (included)
- TO `0x0000 1400 3000 C003` (excluded)

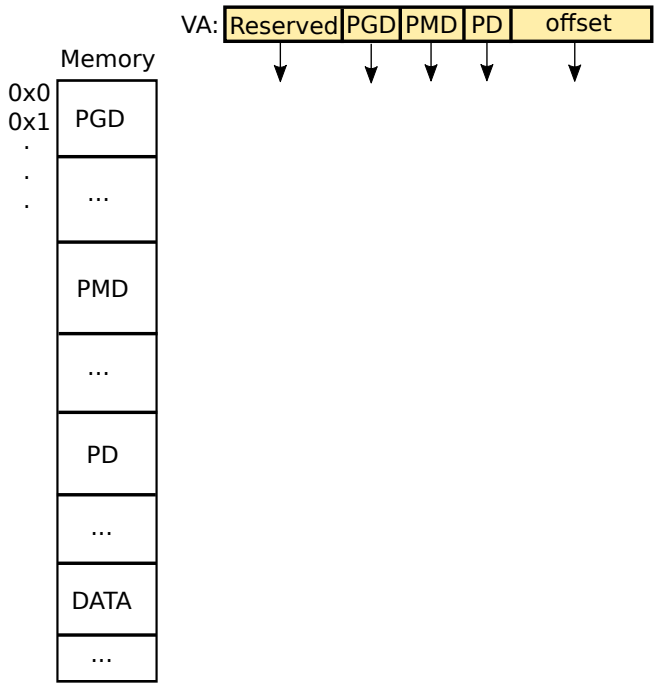


Figure 113: Virtual-to-physical address translation.

[Solution 26] Virtual Memory

a) A page is 64 KiB which is 2^{16} bytes or 2^{14} words. So the width of the offset field is 14 bits.

b) The size of the physical address is 32 bits. Since the offset is on 14 bits, the size of the physical pages number field is $32 - 14 = 18$ bits. Thus, there are 2^{18} pages.

c) The physical address is 32-bit wide and word-addressed. So there are 2^{32} words of 4 bytes, which is 2^{34} bytes or 16 GiB.

d) Each directory occupies an entire physical page. So the number of bits of each field is 14 bits.

e) The size of the virtual page number field would be $3 \times 14 = 42$ bits. So we would need 2^{42} entries (words), which amount to $2^{42}/2^{14} = 2^{28}$ pages.

f) We need 1 page to store the PGD, 2^{14} for the PMD and $2^{14} * 2^{14} = 2^{28}$ pages for the PD.

g) The translation scheme is shown in Fig. 114.

h) We need 1 page to store the PGD, 3 pages for the PMD, and $2 \times 2^{14} + 1$ for the PD, as shown in Fig. 115. Thus, $5 + 2^{15}$ pages in total.

	PGD	**PMD**	**PD**	Offset
From	0x0003	0x0003	0x0003	0x0003
To	0x0005	0x0003	0x0003	0x0003

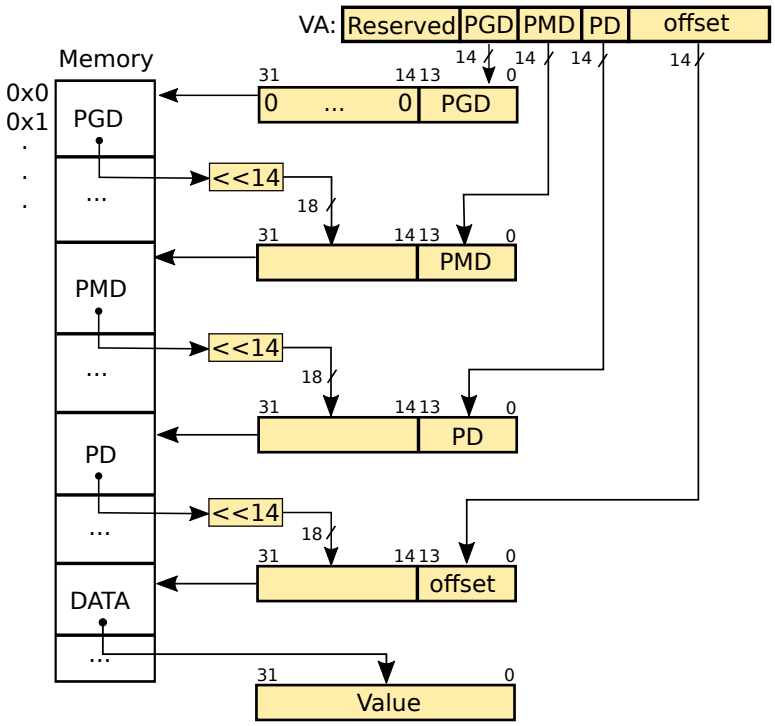


Figure 114: Virtual-to-physical address translation.

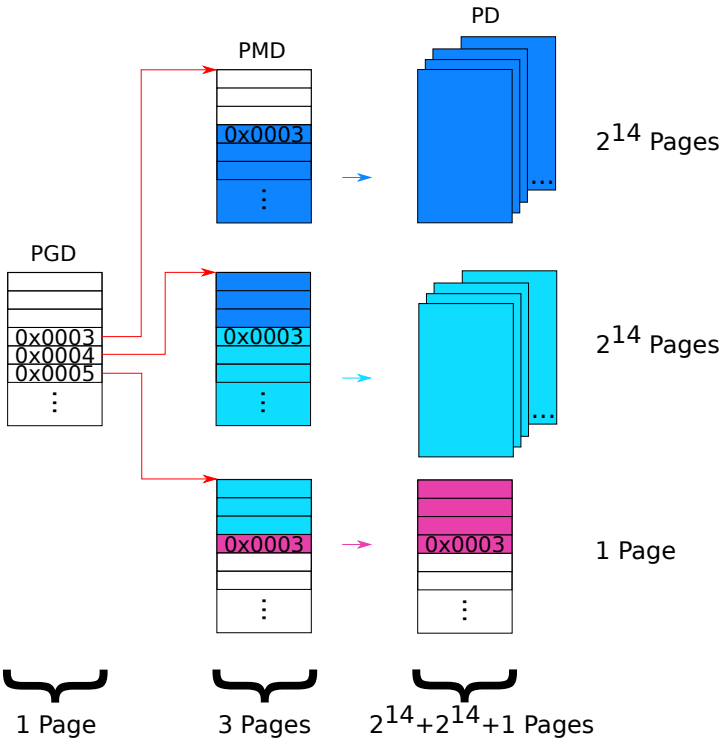


Figure 115: Pages occupied by all directories for this range.

Part IV: Instruction-Level Parallelism

[Exercise 1]

Consider the following RISC-V code:

```
1   lw   x1, 0(x0)
2   lw   x2, 0(x1)
3   add  x6, x5, x4
4   add  x3, x1, x2
5   lw   x4, 0(x6)
6   sub  x2, x0, x4
7   addi x7, x1, 4
8   add  x4, x1, x3
9   sub  x6, x7, x4
```

a) Show all RAW, WAR, and WAW dependencies in that code.

b) Draw an execution diagram representing each processor cycle executing this code. Assume there is a five-stage pipeline ("Fetch", "Decode", "Execute", "Memory", "Writeback"), without any forwarding paths (i.e., "Decode" cannot read values that are being written by "Writeback" in the same cycle, it has to wait for one cycle). What will be the IPC of that code? What would be the ideal IPC?

c) Assume the code is executed on a processor possessing all forwarding paths. Draw the execution diagram. What is the IPC on this processor?

d) Sometimes, running a program on a pipelined processor is not perfect: There are bubbles appearing in the pipeline. Show how a better compiler could optimize the program to remove the bubbles on the processor without forwarding paths. Hint: Do not try to reorder all instructions, but look for the bubbles and make a local reordering. Can you get rid of all the bubbles?

[Solution 1]

a)

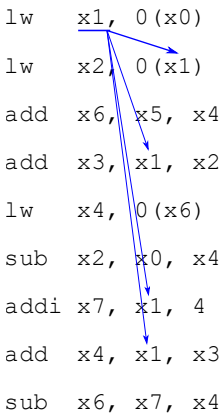


Figure 116: RAW dependencies (Check if a register is being read from after a write)

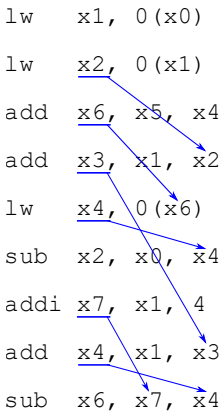


Figure 117: RAW dependencies (cont.)

b) Figure 120 shows the execution diagram. IPC = (9 instructions)/(24 cycles) = 0.375

c) Figure 121 shows the execution diagram. IPC = (9 instructions)/(15 cycles) = 0.6

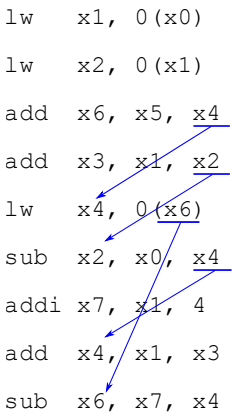


Figure 118: WAR dependencies (Check if a register is being written to after a read)

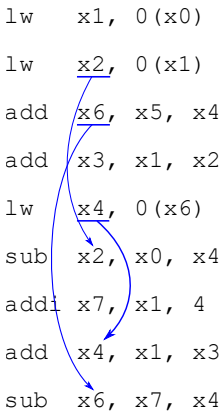


Figure 119: WAW dependencies (Check if a register is being written to after a write)

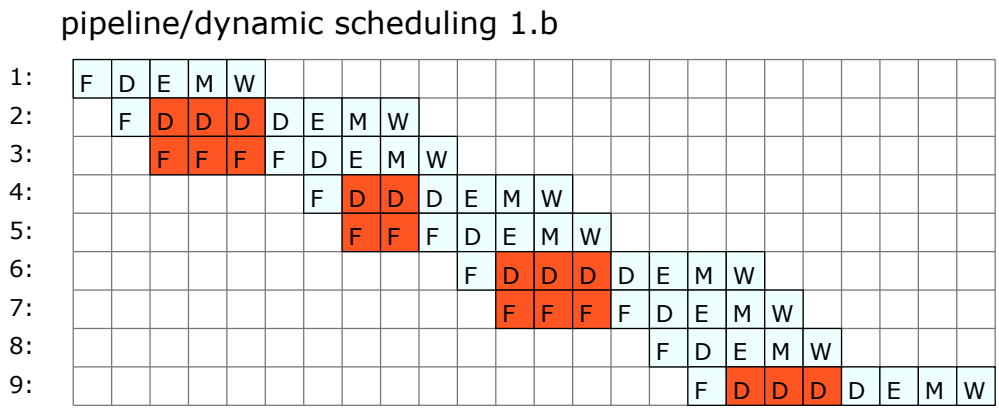


Figure 120: Execution diagram

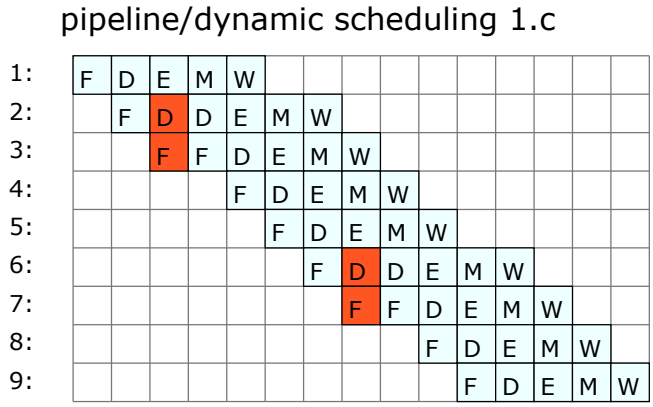


Figure 121: Execution diagram with forwarding paths

d)

```
1    lw   x1, 0(x0)
3    add  x6, x5, x4
     nop
     nop
2    lw   x2, 0(x1)
5    lw   x4, 0(x6)
7    addi x7, x1, 4
     nop
4    add  x3, x1, x2
6    sub  x2, x0, x4
     nop
     nop
8    add  x4, x1, x3
     nop
     nop
     nop
9    sub  x6, x7, x4
```

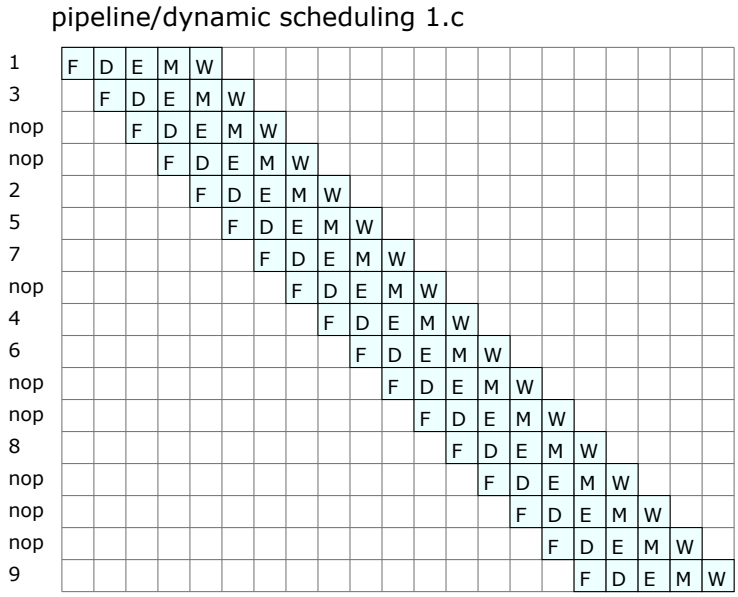


Figure 122: Execution diagram with modified code

[Exercise 2]

Consider the following RISC-V code :

```
1   lw   x1, 0(x5)
2   addi x5, x1, 1
3   lw   x1, 0(x6)
4   add  x3, x0, x5
5   addi x2, x6, -1
6   addi x4, x3, -5
7   add  x3, x2, x4
8   lw   x2, 0(x7)
9   or   x4, x2, x1
10  addi x7, x3, -9
```

a) Show all RAW, WAR and WAW dependencies in that code.

b) Draw an execution table which represents each cycle of a processor executing this code. Assume there is a four-stage pipeline ("Fetch", "Decode", "Execute", "Writeback"), with all forwarding paths. The writeback stage writes to registers in the first half of the cycle, and decode stage reads the register file in the second half of the cycle. Memory accesses are done at the "Execute" stage. How many cycles are needed to execute the code?

c) Assume now a pipeline with seven stages ("Fetch", "Decode", "Execute", three "Memory" cycles and "Writeback"), with all forwarding paths. Arithmetic and logic operations are done at the Execute stage and can be forwarded from any of the 3 Memory stages. The results from Memory operations are only available at the end of the last Memory stage. The writeback stage writes to registers in the first half of the cycle, and decode stage reads the register file in the second half of the cycle. Draw the execution diagram. How many cycles are needed to execute the code?

d) How many cycles are needed to execute instructions 1 to 3 on the processor from the preceding question? It would be better to execute them in the order 1, 3, 2, but something makes that impossible. Make some modifications to the code (but leave the semantics as is) so that it becomes possible to swap instructions 2 and 3. How many cycles are needed to execute 1, 3, 2 now?

[Solution 2]

a) The dependencies are as follows:

- **RAW:**
  - 1→2 for `x1`
  - 3→9 for `x1`
  - 5→7 for `x2`
  - 8→9 for `x2`
  - 4→6 for `x3`
  - 7→10 for `x3`
  - 6→7 for `x4`
  - 2→4 for `x5`
- **WAR:**
  - 2→3 for `x1`
  - 7→8 for `x2`
  - 6→7 for `x3`
  - 7→9 for `x4`
  - 1→2 for `x5`
  - 8→10 for `x7`
- **WAW:**
  - 1→3 for `x1`
  - 5→8 for `x2`
  - 4→7 for `x3`
  - 6→9 for `x4`

b) 13 cycles are necessary to execute the code with the 4-stage pipeline. The execution is represented in Figure 123.

c) 22 cycles are necessary to execute the code with the 7-stage pipeline. The execution is represented in Figure 124.

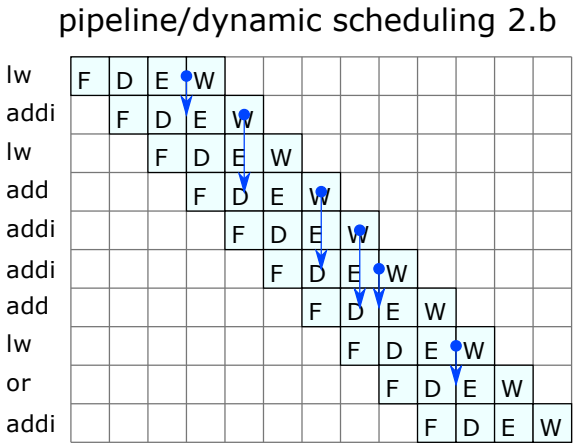


Figure 123: Execution diagram with 4-stage pipeline

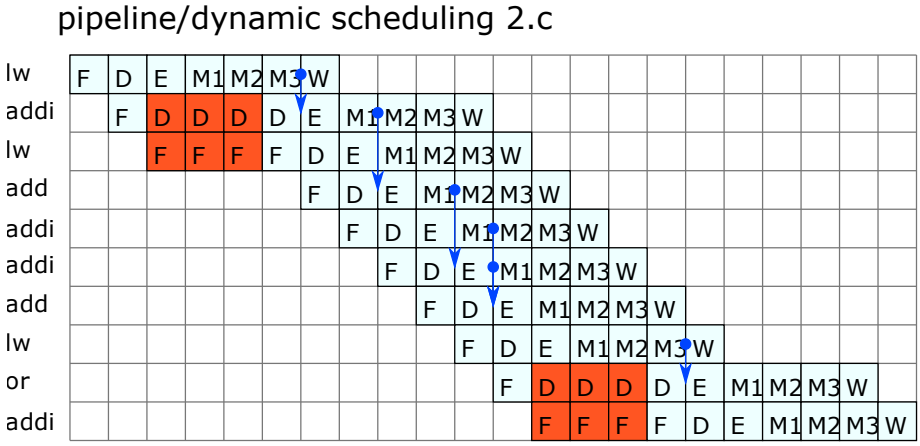


Figure 124: Execution diagram with 7-stage pipeline

d) 12 cycles are needed to execute instructions 1 to 3 on the processor with the 7-stage pipeline. We cannot execute the given code in the order 1, 3, 2, because instruction 3 has a WAR dependency with instruction 2.

A WAR dependency is purely a register naming issue. Provided we have enough registers, we can change the assignments to rename `x1` to another register and avoid the naming dependency.

By replacing `x1` by `x2` in instructions 1 and 2, we get the following code:

```
1   lw   x2, 0(x5)
2   addi x5, x2, 1
3   lw   x1, 0(x6)
4   add  x3, x0, x5
5   addi x2, x6, -1
6   addi x4, x3, -5
7   add  x3, x2, x4
8   lw   x2, 0(x7)
9   or   x4, x2, x1
10  addi x7, x3, -9
```

After this change, we can easily swap instructions 2 and 3 to obtain code that is semantically identical to the original code, and which avoids a few stall cycles.

```
1   lw   x2, 0(x5)
3   lw   x1, 0(x6)
2   addi x5, x2, 1
4   add  x3, x0, x5
5   addi x2, x6, -1
6   addi x4, x3, -5
7   add  x3, x2, x4
8   lw   x2, 0(x7)
9   or   x4, x2, x1
10  addi x7, x3, -9
```

The execution with the modified code is shown in Figure 125.

It now takes 11 cycles to execute the first 3 instructions in the order 1, 3, 2.

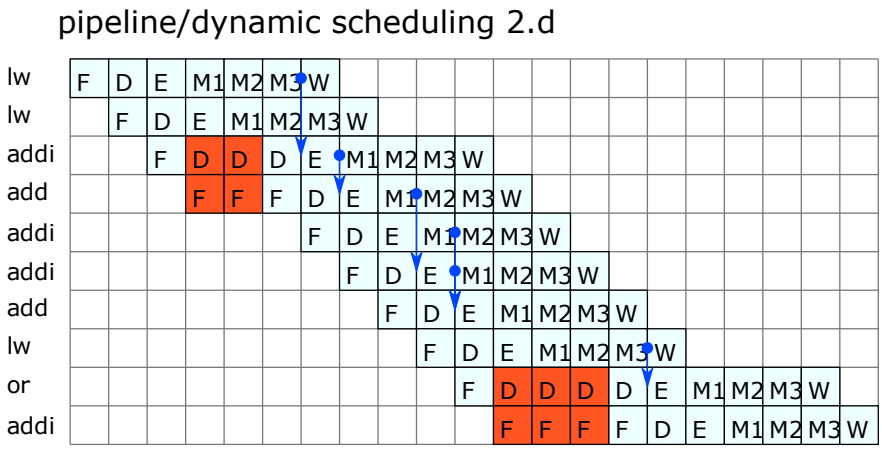


Figure 125: Execution diagram of 7-stage pipeline with modified code

[Exercise 3]

Consider the following RISC-V code for a word-addressed, 16bit address, 32bit data processor. (The `x0` register is always zero).

```
0     lw   x4, A(x0)    # x4 = M[A]
1     lw   x5, B(x0)    # x5 = M[B]
2     lw   x6, C(x0)    # x6 = M[C]
loop:
3     beq  x5, x0, end  # if x5=x0 goto end
4     lw   x1, 0(x4)    # x1 = M[x4]
5     addi x4, x4, 1    # x4 = x4 + 1
6     lw   x2, 0(x4)    # x2 = M[x4]
7     mul  x2, x1, x2   # x2 = x1 * x2
8     sw   x2, 0(x6)    # M[x6] = x2
9     addi x6, x6, 1    # x6 = x6 + 1
10    addi x5, x5, -1   # x5 = x5 - 1
11    j    loop         # goto loop
end:
```

a) Identify all RAW, WAR and WAW dependencies (ignore the pipelined structure of the processor). Consider the first two iterations of the loop.

b) Draw an execution diagram of the code. Assume a five-stage pipeline ("Fetch", "Decode", "Execute", "Memory" and "Writeback"), with all forwarding paths (From **E** to **E**, from **M** to **E** and from **W** to **D**). How many cycles are needed to execute the program? Specify on the diagram which forwarding paths are used. Also point out the cases where forwarding does not solve dependency issues.

Remarks

- For the cycle count of the whole program, stop counting cycles at the "Writeback" phase of the last execution of instruction 3.
- When executing a jump or a branch instruction, the PC gets modified at the end of "Memory".

[Solution 3]

a) Dependencies

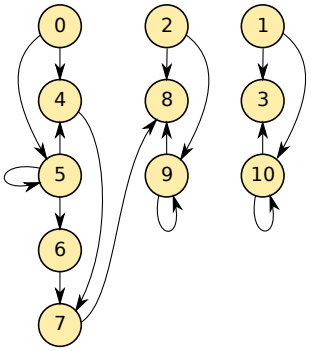


Figure 126: RAW dependencies

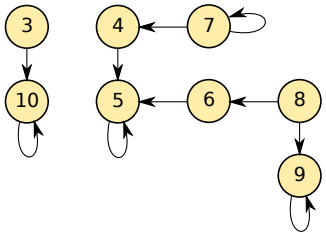


Figure 127: WAR dependencies

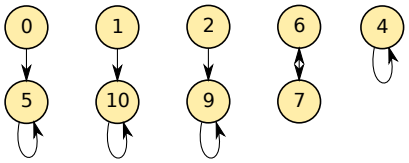


Figure 128: WAW dependencies

b) Execution diagram

Total schedule length (until **W** of last execution of instruction #3): 40 cycles

The RAW dependencies are handled in the following way:

- 0-4 inexistant

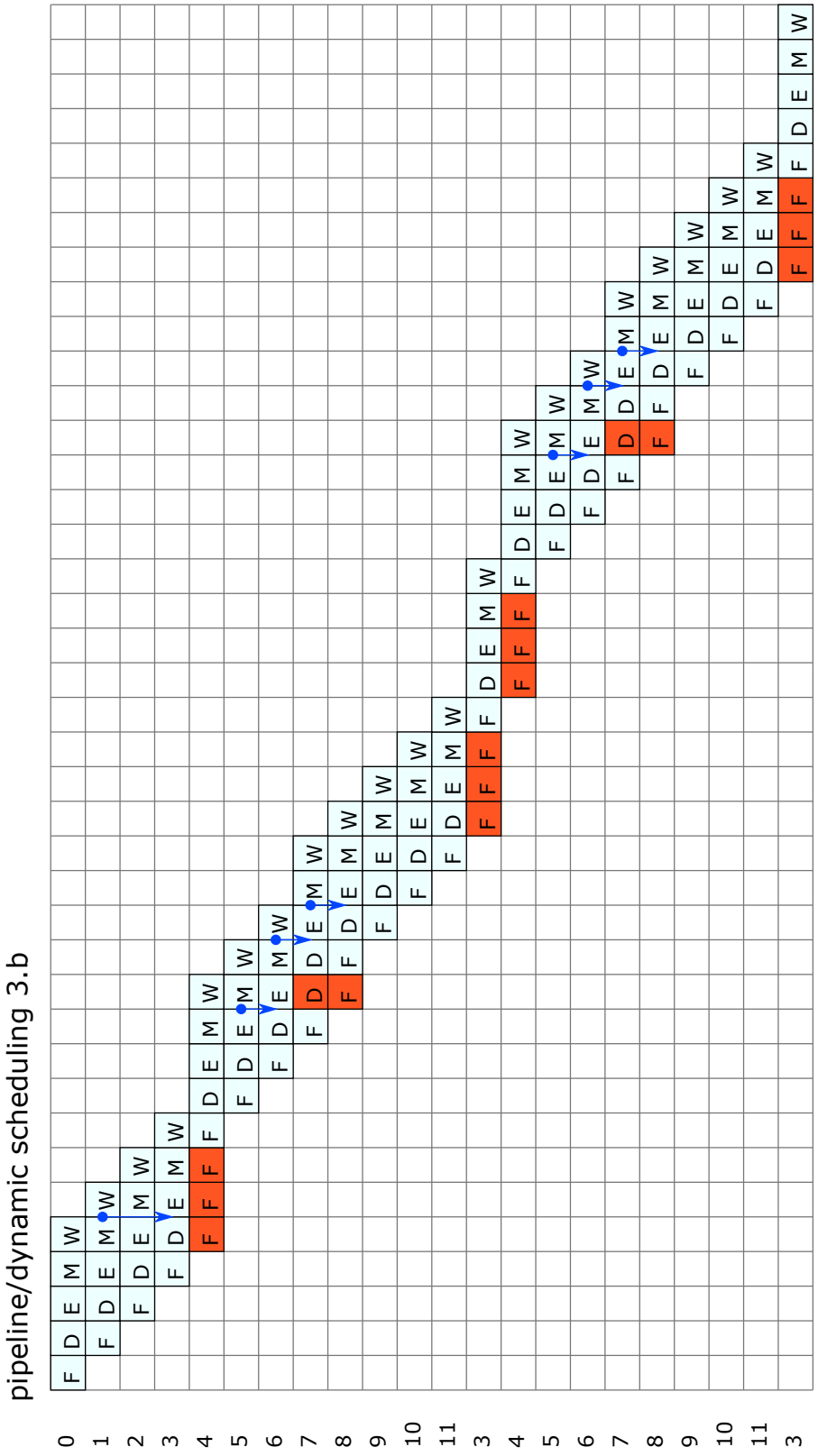


Figure 129: Execution diagram

- 0-5 inexistant
- 1-3 the value `x5` bypasses **M**
- 1-10 inexistant
- 2-8 inexistant
- 2-9 inexistant
- 4-7 inexistant
- 4-7 the value `x1` bypasses **M** during the first decode; inexistant during subsequent decodes
- 5-5 inexistant
- 5-6 the value `x4` bypasses **E**
- 6-7 the first read of `x1` is impossible, a bubble is inserted, afterwards, a **M** bypass is needed
- 7-7 inexistant
- 7-8 the value `x2` bypasses **E**
- 9-9 inexistant
- 9-8 inexistant
- 10-3 inexistant

[Exercise 4]

Consider the following RISC-V code for a hypothetical word-addressed, 16-bit address, 32-bit data processor.

For this exercise, you can assume that a new instruction **mult** has been added to the RISC-V ISA. This instruction is defined as: **mult** rA, rB, rC, corresponding to the pseudocode: rA = rB * rC.

```
0         lw    x4, 34(zero)    # x4 = M(34)
1         lw    x5, 35(zero)    # x5 = M(35)
2         addi  x6, x4, 0       # x6 = x4
3  loop:  lw    x2, 1(x4)       # x2 = M(x4 + 1)
4         mult  x2, x2, x2      # x2 = x2 * x2
5         beq   x5, zero, fin   # if x5 = zero then go to fin
6         li    x2, -1          # x2 = -1
7         lw    x1, 0(x4)       # x1 = M(x4)
8         beq   x1, zero, fin   # if x1 = zero then go to fin
9         addi  x4, x1, 0       # x4 = x1
10        add   x4, x4, x6      # x4 = x4 + x6
11        addi  x5, x5, -1      # x5 = x5 - 1
12        j     loop            # go to loop
13 fin:
```

a) Identify all RAW, WAR, and WAW dependencies (ignore the pipelined structure of the processor). Consider the first two iterations of the loop.

b) Draw an execution diagram of the code. Assume a six-stage pipeline ("Fetch", "Decode", "Execute", "Memory1", "Memory2", and "Writeback"), with all forwarding paths (From **E** to **E**, from **M1** to **E**, from **M2** to **E**, and from **W** to **D**). How many cycles are needed to execute the program? Specify on the diagram which forwarding paths are used. Also, point out the cases where forwarding does not solve dependency issues.

Remarks

- For the cycle count of the whole program, stop counting cycles at the "Writeback" phase of the second execution of instruction 5.
- When executing a jump or a branch instruction, the PC gets modified at the end of "Memory2".

[Solution 4]

a) Dependencies

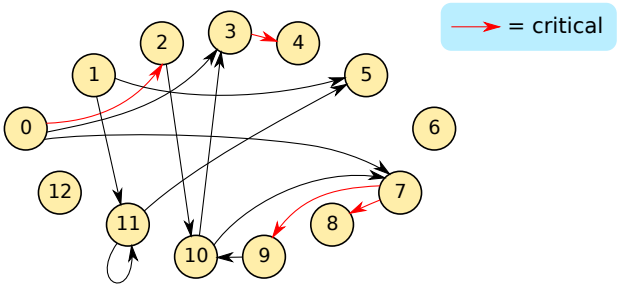


Figure 130: RAW dependencies

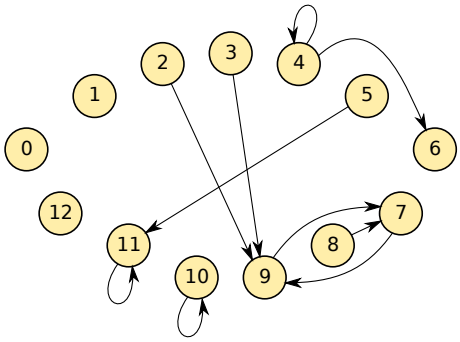


Figure 131: WAR dependencies

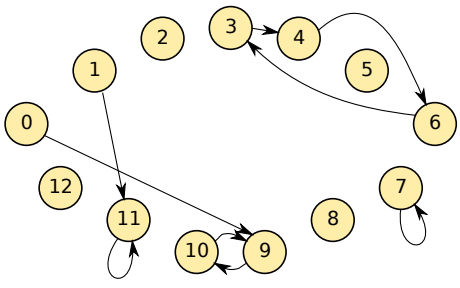


Figure 132: WAW dependencies

Zoom-in on M1/M2

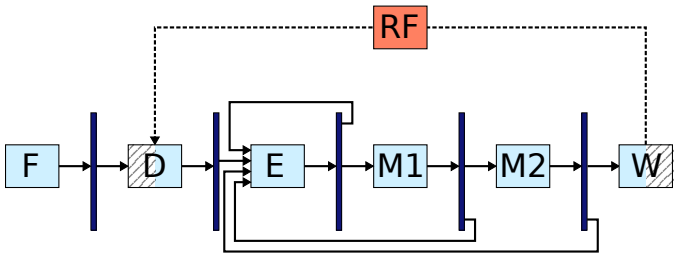


Figure 133: Pipeline structure

- Load values only available at end of **M2**!
- Only ALU results can be bypassed at the end of **M1**!

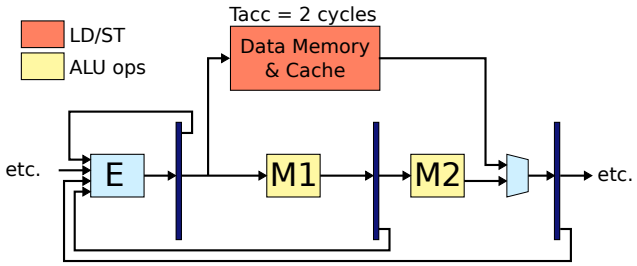


Figure 134: Zoom-in on M1/M2

b) Total schedule length (until **W** of second execution of instruction #5): 40 cycles

Situations where forwarding is not enough

- Bubbles are still added for:
  - Any instruction with a dependency from a Load 1 cycle before generates 2 bubbles
  - Any instruction with a dependency from a Load 2 cycle before generates 1 bubbles
- Bubbles are avoided for any dependency from ALU operations

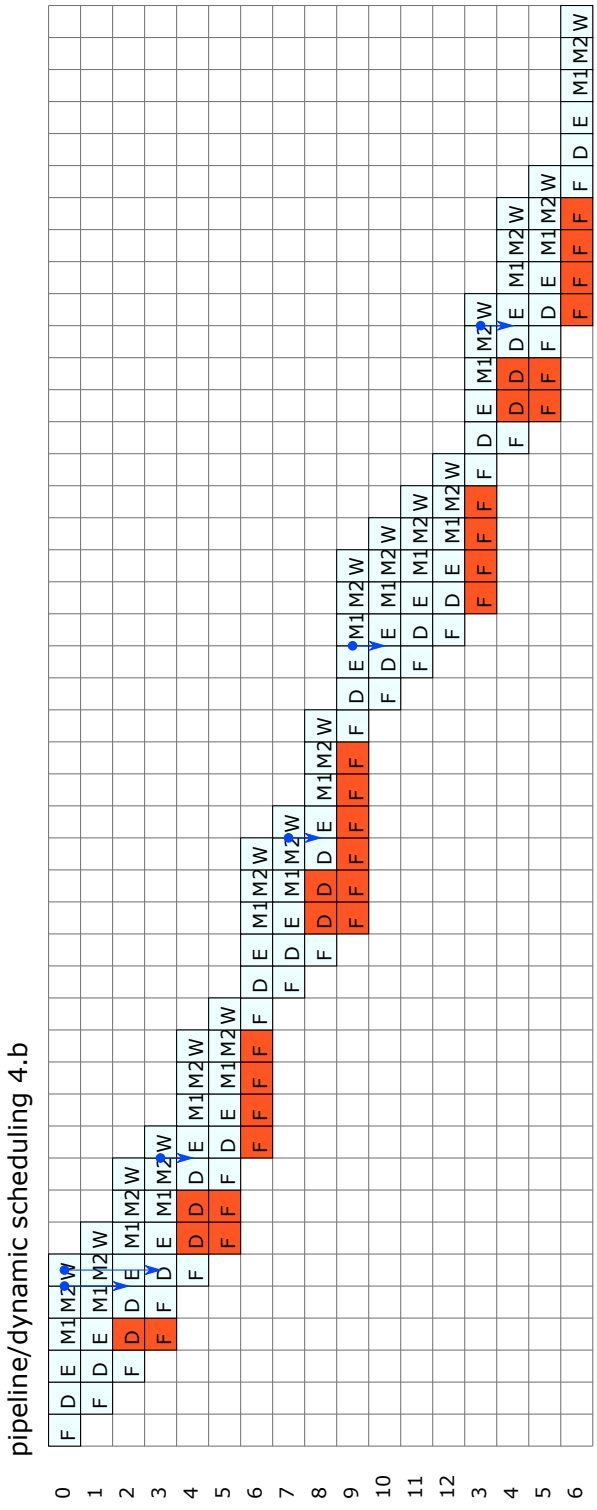


Figure 135: Execution diagram

[Exercise 5]

Consider a processor with a simple seven-stage pipeline: "Fetch", "Decode", two "Execute" stages, two "Memory" stages, and one "Writeback". All logic and arithmetic operations are done in the first **E1** stage, except multiplication which takes until **E2**. The processor designer asks if it would be better to add detection logic for control hazards, or additional forwarding paths. He considers the two following alternatives:

1. The processor has no forwarding paths. Reading a value in the RF can only be done one cycle after a write. If the processor encounters a jump (both conditional and unconditional), it continues to read and decode subsequent instructions. At the end of **E2**, the processor knows if the branch should be taken or not. If it is taken, the pipeline gets flushed, the instructions in the preceding stages are discarded and the processor jumps to the correct instruction. If not, execution continues.
2. The processor has the following three forwarding paths: From **E1** to **E1**, from **E2** to **E1**, and it can read a value from the RF during the same cycle it is written (Register file forwarding). However, there is no control hazard logic: all instructions entering the pipeline are executed until the destination of a jump is known (which is, as above, at the end of **E2**). Therefore, the code has to contain delay slots.

Consider the following benchmark program. You can assume that a new instruction **mult** has been added to the RISC-V ISA. This instruction is defined as: **mult** rA, rB, rC, corresponding to the pseudocode: rA = rB * rC.

```
1000: loop: lw   t1, 0(a1)
1004:       addi a1, a1, 4
1008:       beq  a1, a3, end
1012:       add  t1, t1, a2
1016:       lw   a0, 0(t1)
1020:       jal  ra, quad
1024:       add  t2, t2, a0
1028:       j    loop
1032: end:
2000: quad: mult a0, a0, a0
2004:       mult a0, a0, a0
2008:       jalr zero, ra, 0
```

At the beginning, a1 points to a vector of several elements: $[13, 12, 11, 7, 14, \cdots]$ and a3 points to the third element of the vector.

a) Simulate the code using the first processor. How many cycles are needed to execute the code? How many cycles are wasted because of data hazards? How many because of control hazards? (The program terminates when fetching the instruction at `end:`).

b) Insert **`nop`** operations where needed such that the execution is correct on the second processor. Simulate the resulting code. How many cycles are needed? How many cycles are wasted because of data hazards?

c) Are all those **`nop`**'s necessary? Reorder the instructions to replace the **`nop`**'s. How many are still remaining?

[Solution 5]

a)

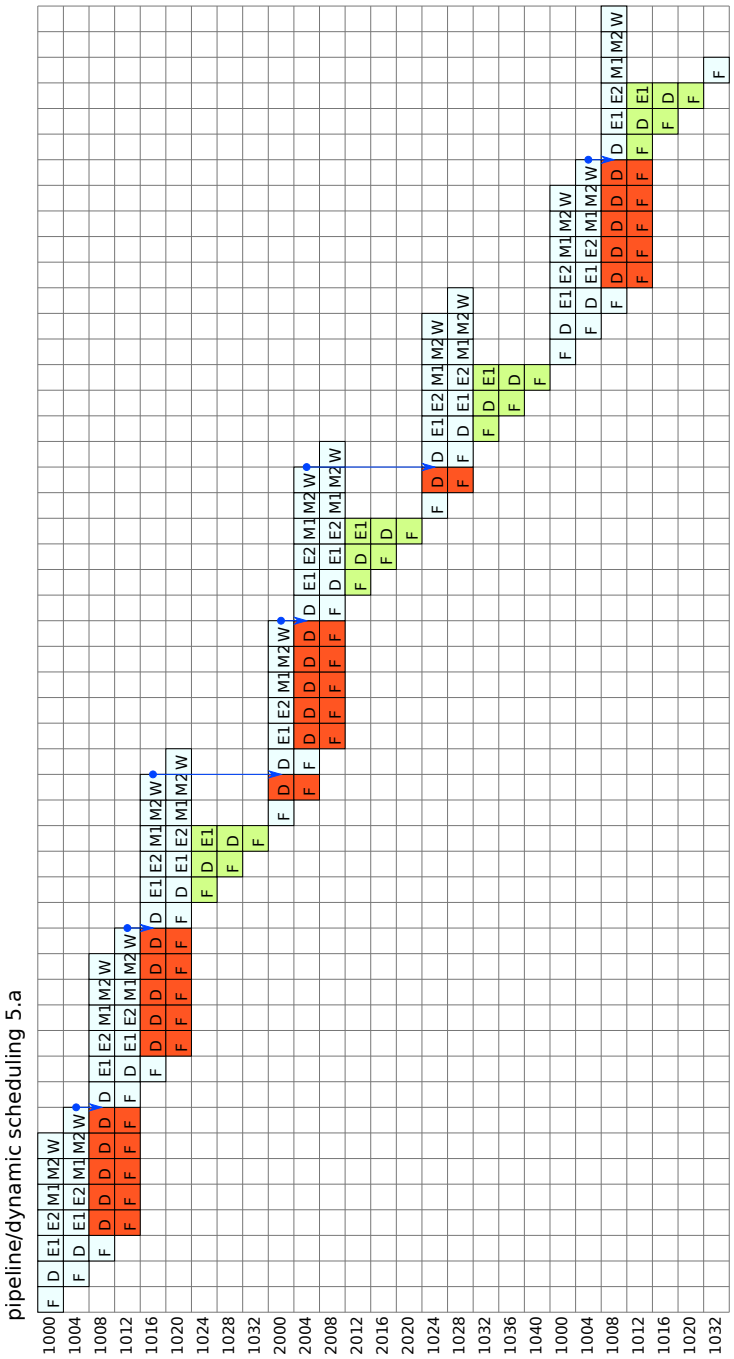


Figure 136: Execution diagram for processor 1

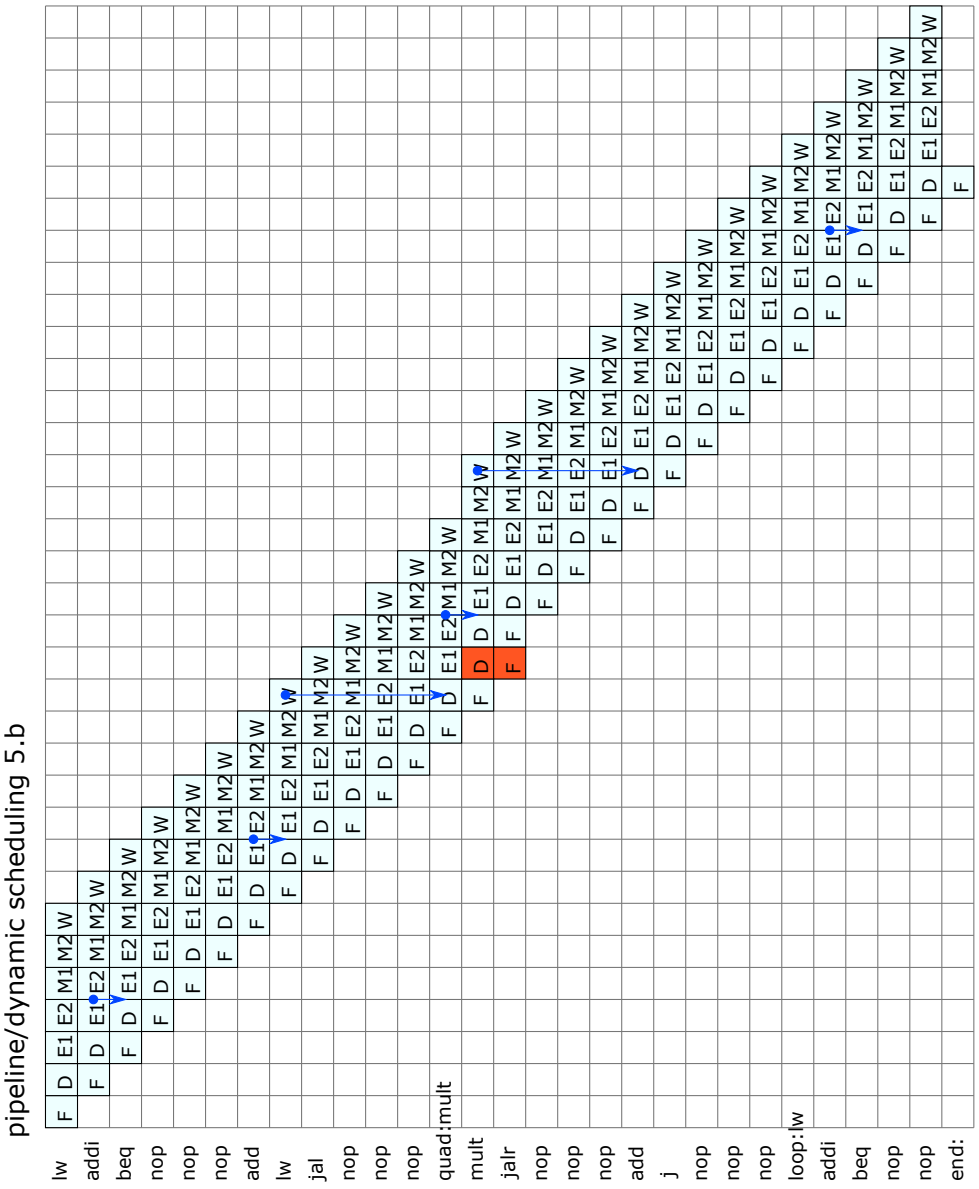


Figure 137: Execution diagram for processor 2

Total number of cycles 49

Lost cycles due to data hazards 22

Lost cycles due to control hazards 12

b)

Total number of cycles 31

Lost cycles due to data hazards 1

c)

```
loop: lw   t1, 0(a1)
      addi a1, a1, 4
      beq  a1, a3, end
      nop
      nop
      nop
      add  t1, t1, a2
      lw   a0, 0(t1)
      jal  ra, quad
      nop
      nop
      nop
      j    loop
      add  t2, t2, a0
      nop
      nop
end:
quad: jalr zero, ra, 0
      mult a0, a0, a0
      mult a0, a0, a0
      nop
```

Remarks

- There cannot be any jumps or branches in the delay slot;
- There cannot be any instructions in the delay slots of **jal** as they would be executed again after the **jalr**;
- We keep the first three **nop**'s to handle unconditional jumps more easily.

[Exercise 6]

Consider a processor with a simple seven-stage pipeline: "Fetch", "Decode", "Execute", three "Memory" stages and one "Writeback". The processor designer asks if it'd be better to add detection logic for control hazards or additional forwarding paths. He considers the two following alternatives:

1. The processor has no forwarding paths. Reading a value in the RF can only be done one cycle after a write. If the processor encounters a jump (both conditional and unconditional), it continues to read and decode subsequent instructions. At the end of **E**, the processor knows if the branch should be taken or not. If it is taken, the pipeline gets flushed, the instructions in the preceding stages are discarded and the processor jumps to the correct instruction. If not, execution continues.
2. The processor has the following two forwarding paths: From **E** to **E** and it can read a value from the RF during the same cycle it is written (Register file forwarding). However, there is no control hazard logic: all instructions entering the pipeline are executed until the destination of a jump is known (which is, as above, at the end of **E**). Therefore, the code has to contain delay slots.

Consider the following benchmark program. You can assume that a new instruction **mult** has been added to the RISC-V ISA. This instruction is defined as: **mult** rA, rB, rC, corresponding to the pseudocode: rA = rB * rC.

```
1000: loop:   lw   t1, 0(a1)
1004:         beq  t1, zero, end
1008:         add  t1, t1, a2
1012:         lw   a0, 0(t1)
1016:         jal  ra, square
1020:         add  t2, t2, a0
1024:         addi a1, a1, 4
1028:         j    loop
      end:
2000: square: mult a0, a0, a0
2004:         jr   ra
```

At the beginning, a1 points to a vector of two elements: $[151, 0]$.
a) Simulate the code using the first processor. How many cycles are needed to execute the code? How many cycles are wasted because of data hazards? How many because of control hazards? (The program terminates when fetching the instruction at end:).

b) Insert **nop** operations where needed such that the execution is correct on the second processor. Simulate the resulting code. How many cycles are needed? How many cycles are wasted because of data hazards?
c) Are all those **nop**'s necessary ? Reorder the instructions to replace the **nop**'s. How many are still remaining?

[Solution 6]

a)

Total number of cycles 42

Lost cycles due to data hazards 21

Lost cycles due to control hazards 8 (note that, even with a perfect branch predictor, 6 out of these 8 cycles would still be lost due to data hazards)

b)

Total number of cycles 34

Lost cycles due to data hazards 11

c)

```
loop:   lw   t1, 0(a1)
        beq  t1, zero, end
        nop
        nop
        jal  square
        add  t1, t1, a2
        lw   a0, 0(t1)
        j    loop
        add  t2, t2, a0 # can be swapped with next instruction
        addi a1, a1, 4
end:
square: jr   ra
        mult a0, a0, a0
        nop
```

Remarks

- There cannot be any jumps or branches in the delay slot;
- We keep the first two **nop**'s to handle unconditional jumps more easily.

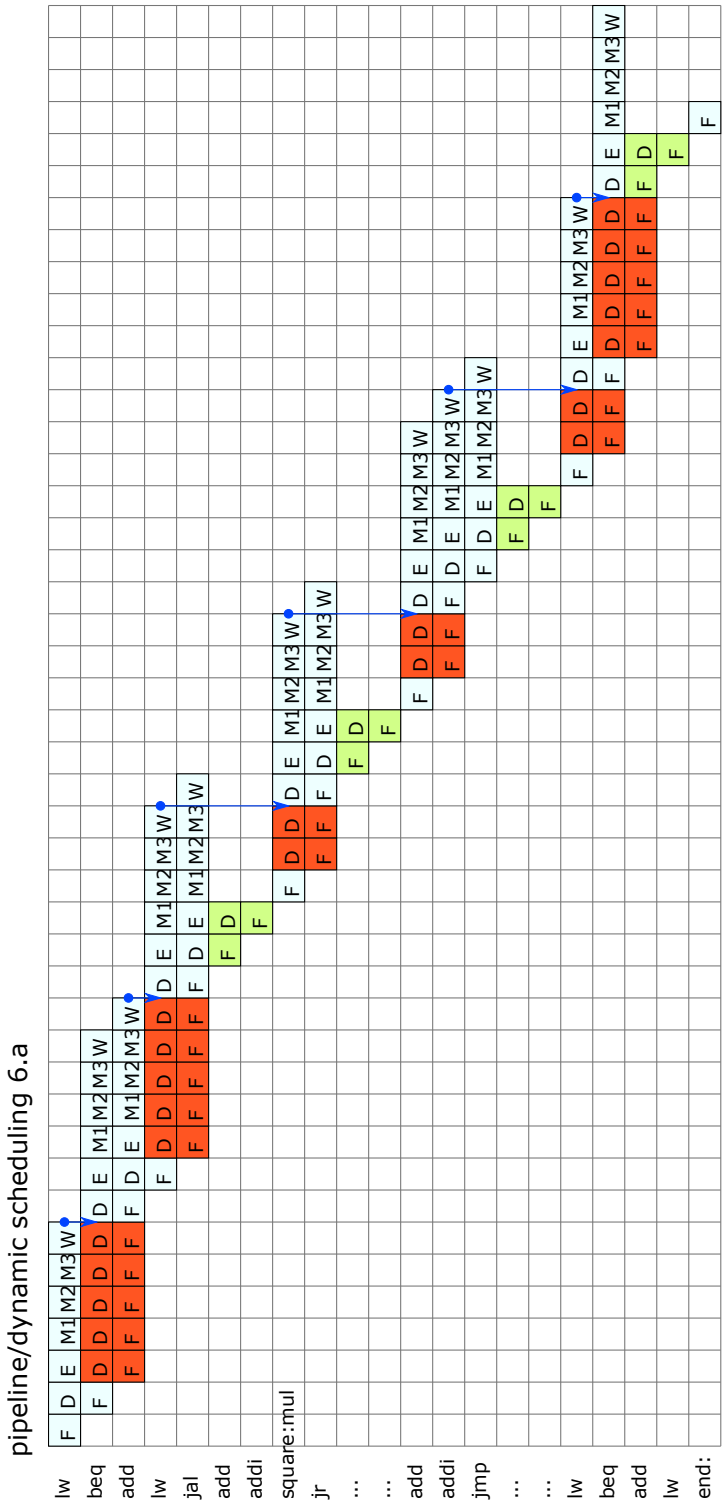


Figure 138: Execution diagram on processor 1

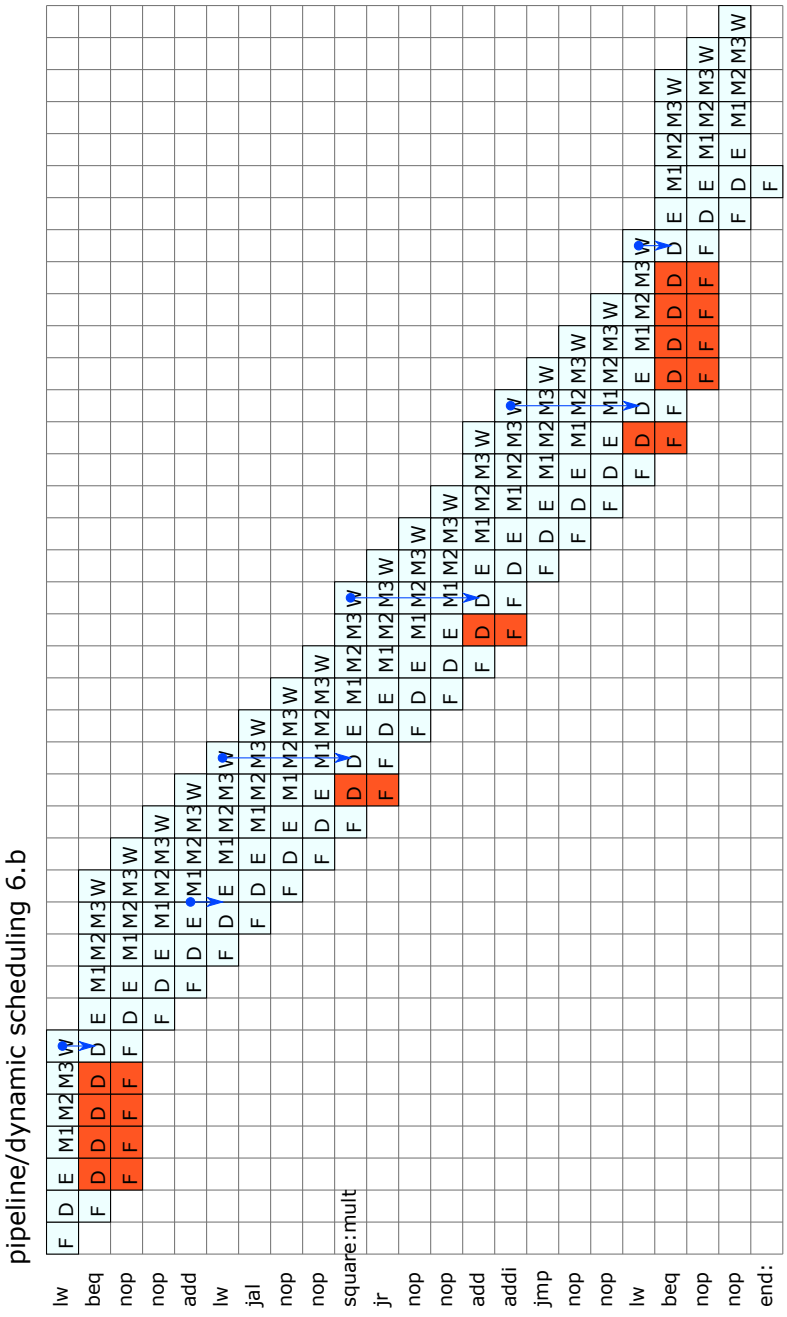


Figure 139: Execution diagram on processor 2

[Exercise 7]

```
1   lw   x1, 0(x5)
2   addi x5, x1, 1
3   lw   x1, 0(x6)
4   add  x3, x0, x5
5   addi x2, x6, -1
6   addi x4, x3, -5
7   add  x3, x2, x4
8   lw   x2, 0(x7)
9   or   x4, x2, x1
10  addi x7, x3, -9
```

a) Draw the execution diagram for this code for five different architectures and compare the number of cycles required.

Architecture #1 Multicycle, unpipelined processor.

Execution latencies:

ALU Operations 4 cycles

Memory Operations 6 cycles

Architecture #2 6-stage pipelined, no forwarding paths

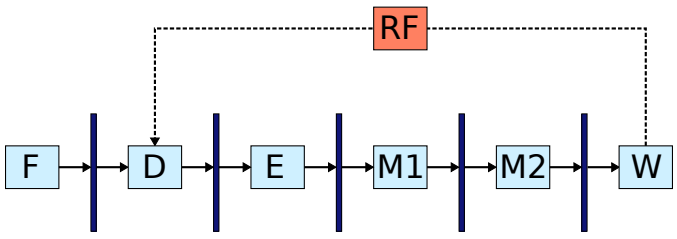


Figure 140: Architecture #2

Architecture #3 6-stage pipelined, some forwarding paths

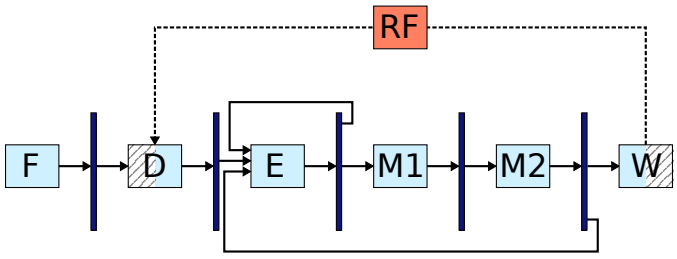


Figure 141: Architecture #3

Architecture #4 6-stage pipelined, all forwarding paths

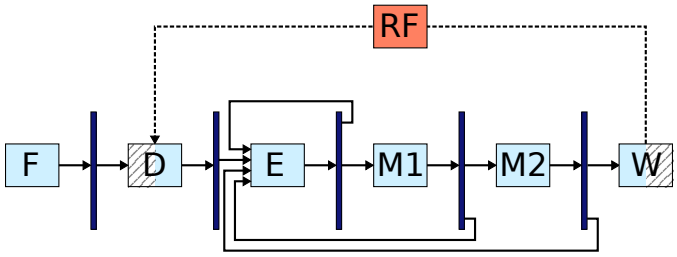


Figure 142: Architecture #4

Architecture #5 Dynamically scheduled, OOO, unlimited RS and ROB size, 1 ALU (latency 1), 1 Memory Unit (latency 3)

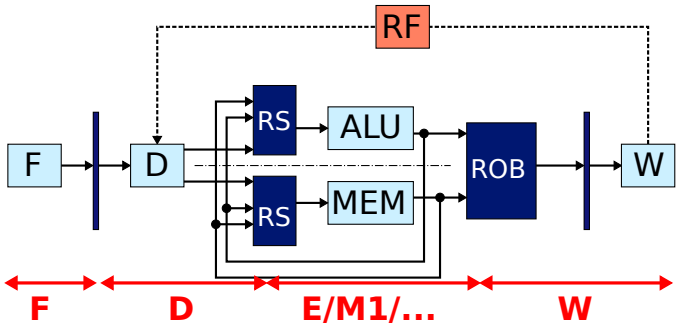


Figure 143: Architecture #5

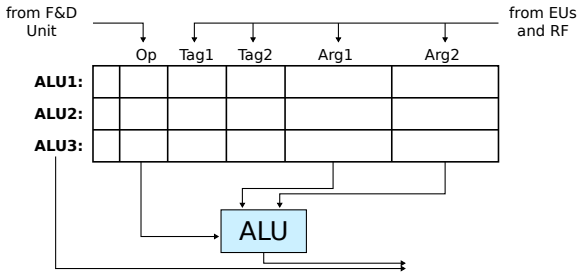


Figure 144: Architecture #5 - ALU reservation station

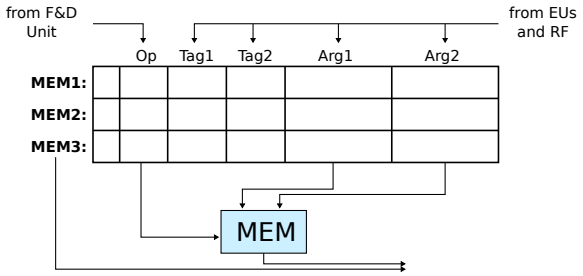


Figure 145: Architecture #5 - MEM reservation station

At the beginning of each cycle:

E phase Issue ready instructions - from all RSs, issue as many ready instructions as there are FUs available.

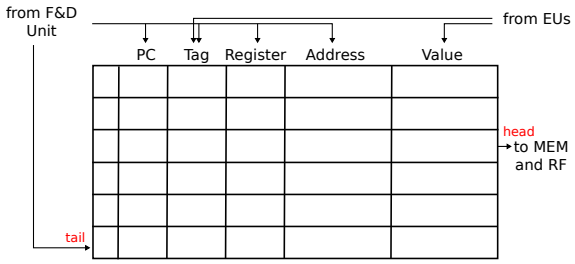


Figure 146: Architecture #5 - Reordering buffer (ROB)

W phase Writeback results in order - remove top entry from ROB if completed.

At the end of each cycle:

D phase Load result of decoding stage - To the relevant RS, including ready register values; To the ROB, to prepare the placeholder for the result.

E phase Broadcast results from all FUs - To all RSs (incl. deallocation of the entry); To the ROB.

[Solution 7]

a) Architecture 1

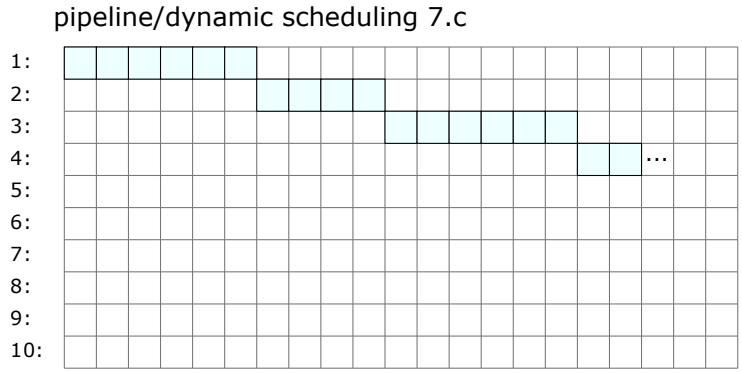


Figure 147: Execution diagram for architecture 1

```
1   lw   x1, 0(x5)   # 6 cycles
2   addi x5, x1, 1   # 4 cycles
3   lw   x1, 0(x6)   # 6 cycles
4   add  x3, x0, x5  # 4 cycles
5   addi x2, x6, -1  # 4 cycles
6   addi x4, x3, -5  # 4 cycles
7   add  x3, x2, x4  # 4 cycles
8   lw   x2, 0(x7)   # 6 cycles
9   or   x4, x2, x1  # 4 cycles
10  addi x7, x3, -9  # 4 cycles
```

46 cycles, CPI = 4.6

b) Architecture 2

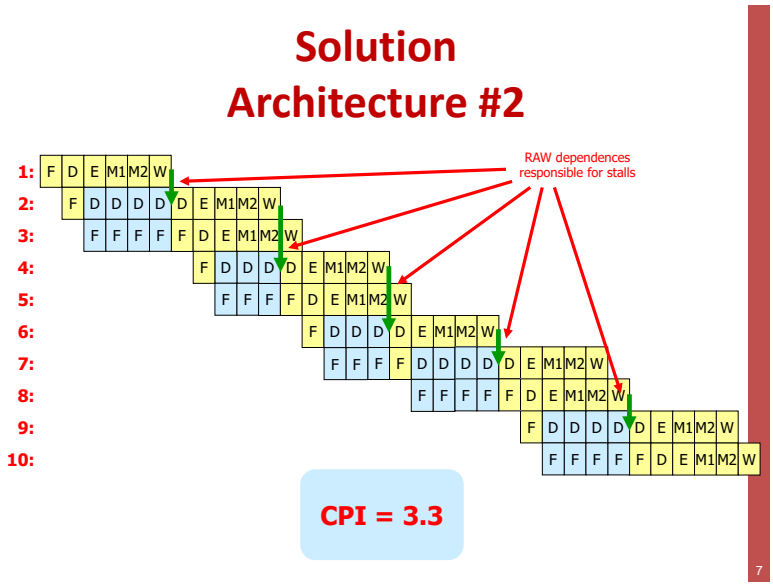


Figure 148: Execution diagram for architecture 2

c) Architecture 3

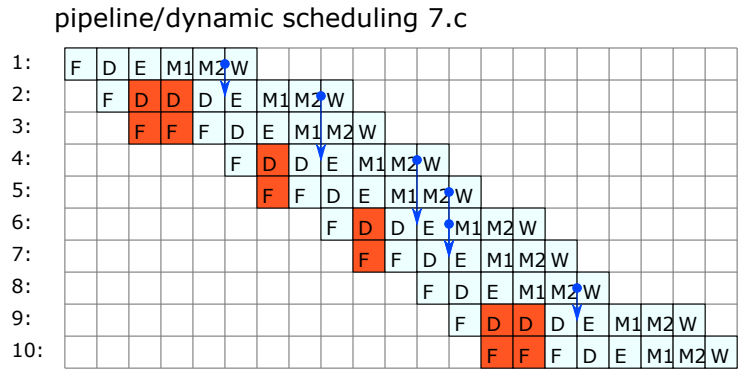


Figure 149: Execution diagram for architecture 3 (no M1 to E path)

d) Architecture 4

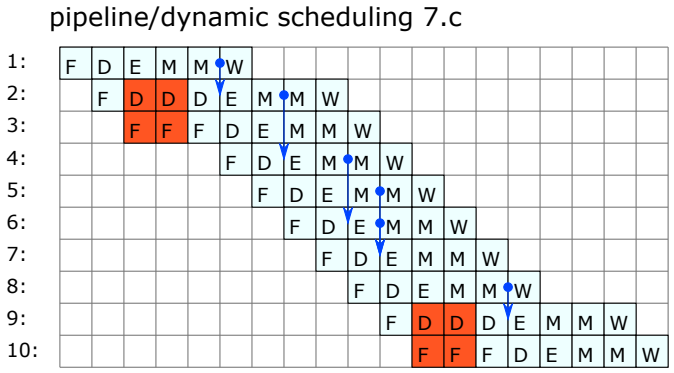


Figure 150: Execution diagram for architecture 4

e) Architecture 5

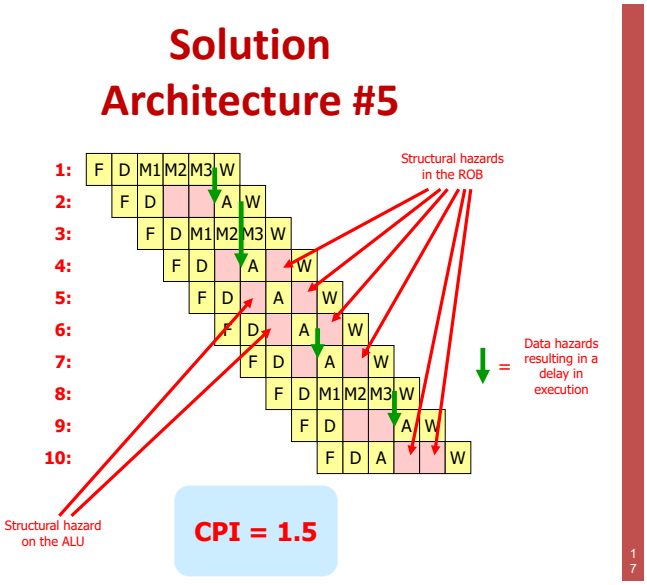


Figure 151: Execution diagram for architecture 5

[Exercise 8]

Consider a processor with a pipeline corresponding to the following figure:

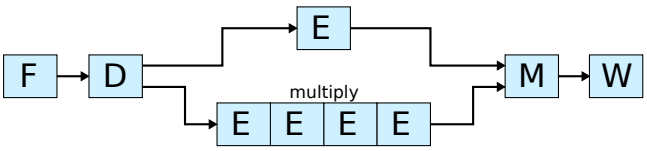


Figure 152: Pipeline structure

The `mul` has another latency compared to the other instructions.

All forwarding paths are implemented. The writeback stage writes to registers in the first half of the cycle, and decode reads the register file in the second half of the cycle. When an instruction stalls, all preceding stages stall too. If two instructions arrive at the Memory stage at the same time, the older one gets priority over the more recent one. The Writeback stage contains a reordering buffer and the writebacks are only made if necessary, and are commited in-order.

Assume that the processor always does a sequential execution, with static branch prediction "not taken", writes the branch destination address to the PC during the Execute stage, and flushes the pipeline in case the branch is taken.

```
0  loop: lw   t0, 0(t2)
1        lw   t4, 0(t3)
2        mul  t0, t0, t4
3        add  t1, t1, t0
4        addi t2, t2, 4
5        addi t3, t3, 4
6        bne  t5, t2, loop
```

Where the initial value for `t5` equals `t2`+8.

a) Draw RAW, WAR and WAW dependencies.

b) Draw an execution diagram of the code, and specify the forwarding paths used.

c) What is the branch penalty when a branch is taken ?

d) Calculate the IPC.

Assume now that the pipeline does not get flushed when a branch is taken. (The correct branch address is still available at the end of the E stage). This means that the compiler has to take care of the branch penalty (and has to insert delay slots at appropriate positions).

e) Do the minimal amount of modifications such that the above code gets executed correctly.

f) A compiler will try to maximise performance by eliminating stalls and make the best use possible of the delay slots. Apply these optimisations to the code and give the best possible code.

g) Draw the execution diagram of the code and specify all forwarding paths used.

h) Calculate the IPC.

[Solution 8]

a)

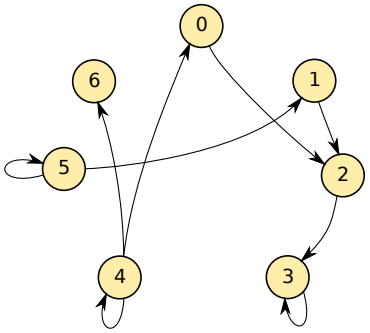


Figure 153: RAW dependencies

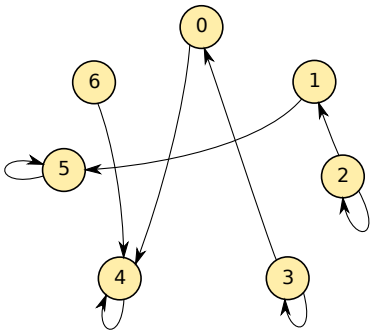


Figure 154: WAR dependencies

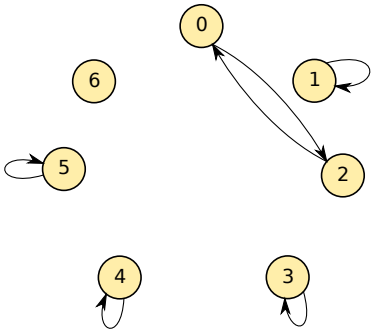


Figure 155: WAW dependencies

b)

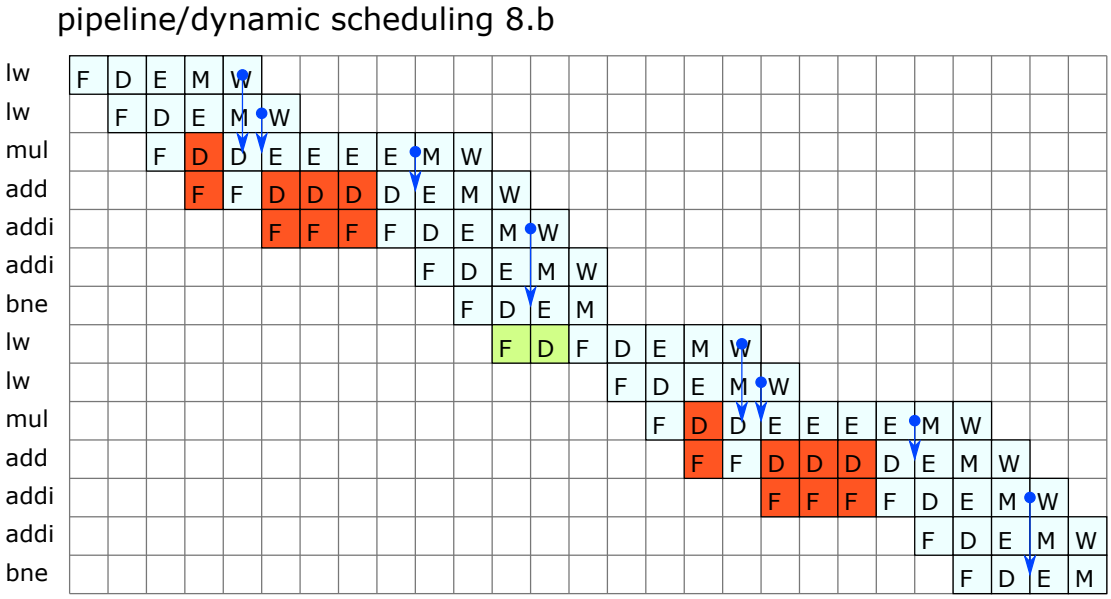


Figure 156: Execution diagram before modifications

c) Branch penalty = 2

d) IPC $= 14/27 = 0.518$

e) Minimal modifications

```
0  loop: lw   $t0, 0($t2)
1        lw   $t4, 0($t3)
2        mul  $t0, $t0, $t4
3        add  $t1, $t1, $t0
4        addi $t2, $t2, 4
5        addi $t3, $t3, 4
6        bne  $t5, $t2, loop
7        nop
8        nop
```

f) Optimised

```
0  loop: lw   $t0, 0($t2)
1        lw   $t4, 0($t3)
4        addi $t2, $t2, 4
2        mul  $t0, $t0, $t4
5        addi $t3, $t3, 4
6        bne  $t5, $t2, loop
```

```
7          nop
3          add  $t1, $t1, $t0
```

g)

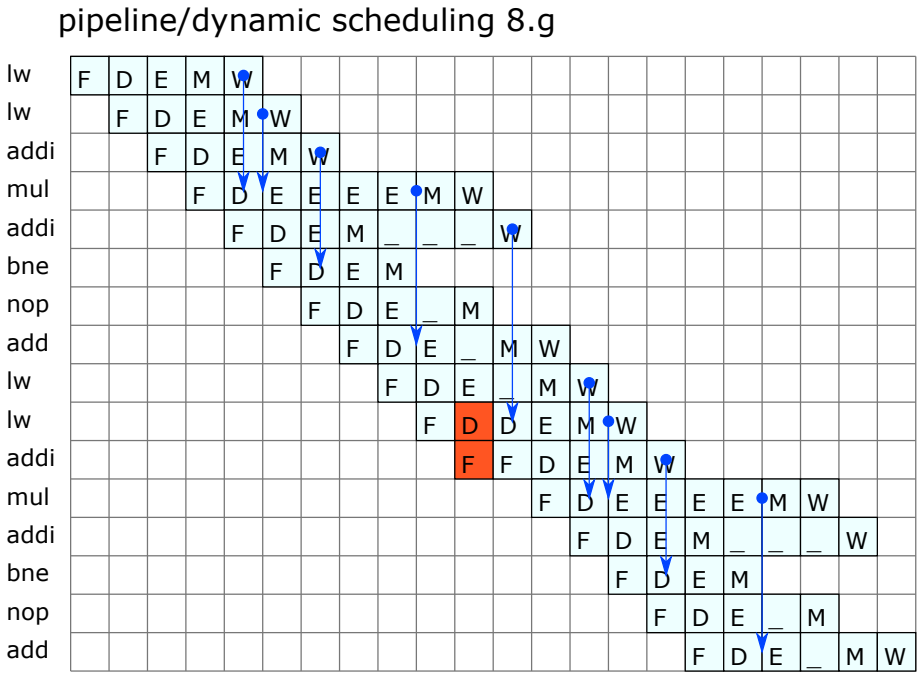
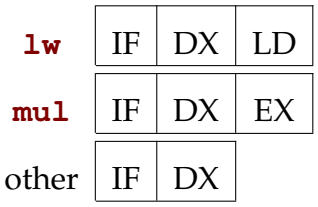


Figure 157: Execution diagram of optimised code

h) IPC $= 14/22 = 0.64$ (without counting the **nop**'s)

[Exercise 9]

Consider a RISC-V processor with a pipeline corresponding to the following figure:



Where the stages are:

IF Instruction Fetch

DX Decode and Execute

LD Memory read

EX Second half of execute for the multiplication

Register read is done at the beginning of **DX**, writing at the end of **LD**, **EX** and **DX**. Branch decision is done at the end of DX, and the destination address gets written into the PC.

The processor neither has forwarding paths nor can stall the pipeline: All hazards and conflicts have to be solved by the compiler or the programmer, who must insert instructions which do not cause hazards, or **nop**'s.

The program uses the **neg** pseudoinstruction of the RISC-V ISA. This pseudoinstruction is translated in the following way: **neg** `rA, rB` maps to the instruction: **sub** `rA, x0, rB`.
You can also assume that a new instruction **mult** has been added to the RISC-V ISA. This instruction is defined as: **mult** `rA, rB, rC`, corresponding to the pseudocode: `rA = rB * rC`.

Consider the following program:

```
0  loop:    lw   t1, 0(t3)
1           neg  t1, t1
2           lw   t2, 0(t4)
3           neg  t2, t2
4           mult t1, t1, t2
5           add  t3, t3, t1
6           addi t3, t3, 4
7           addi t4, t4, 4
8           addi t5, t5, -1
9           bne  t5, zero, loop
10 reinit:  addi t5, zero, 10
```

a) Give all RAW dependencies.

b) Draw the execution diagram representing the execution of this code in the pipeline (stop at **mult** in the second iteration of the loop).

c) Does this program work correctly? Explain the different problems, and correct the program inserting only **nop**'s.

d) Draw the execution diagram of the modified code.

e) Calculate the CPI.

f) Optimise the program, this time by reordering some instructions, and try to remove as many **nop**'s as possible.

g) Draw the execution diagram of the optimised code.

h) Calculate the CPI.

[Solution 9]

a)

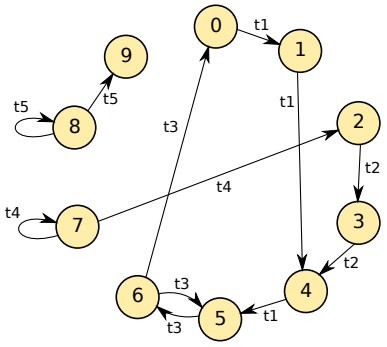


Figure 158: RAW dependencies

b)

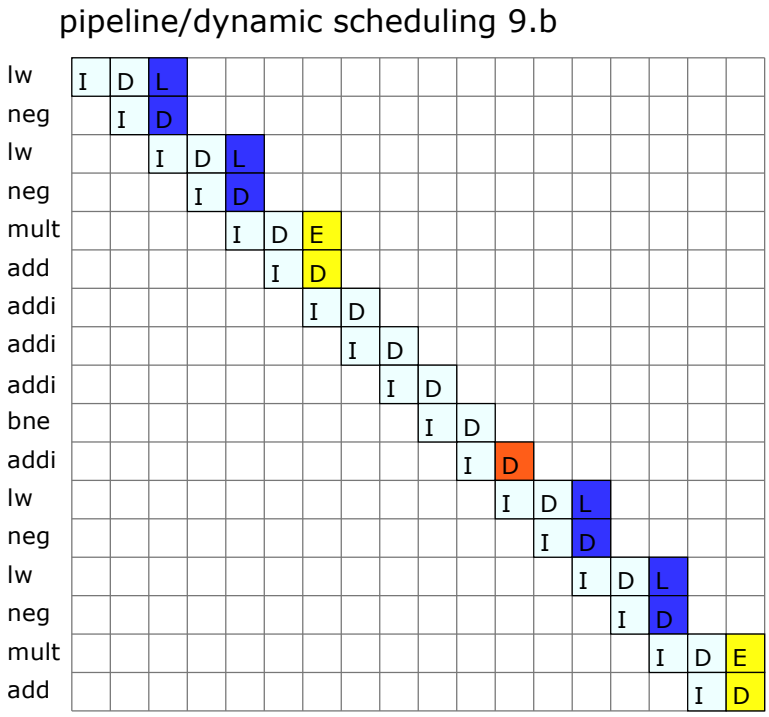


Figure 159: Execution diagram

c) Problems with the execution of the code: (Only dependencies of instructions that

come one after the other are critical in the given pipeline, thus only those dependencies are shown on the execution diagram)

- blue stages: Old value of register t1 (or t2) is negated instead of the value loaded from the memory. Also there is a race (thus a conflict) between the **D** and **L** stages to store the value of negated or memory read result to the register t1 (or t2).
- yellow stages: The value added to the content of t3 is not the multiplied value but one of the values read from memory.
- red stages: The value of t5 is reset to 10 even if the execution continues from the instruction labelled as loop. Thus, this loop will be executed infinitely!

All these problems are a consequence of the fact that the DX stage cannot be delayed since the pipeline does not have the capability to stall. Therefore, for each of the problem situations described above we have to insert a `nop`. The corrected code would be as follows:

```
loop:   lw   t1, 0(t3)
        nop
        neg  t1, t1
        lw   t2, 0(t4)
        nop
        neg  t2, t2
        mult t1, t1, t2
        nop
        add  t3, t3, t1
        addi t3, t3, 4
        addi t4, t4, 4
        addi t5, t5, -1
        bne  t5, zero, loop
        nop
reinit: addi t5, zero, 10
```

d) `addi t5, zero, 10` is no more executed. The branch followed by a `nop` now correctly branches to the label `loop:`.

e) CPI = 23 cycles / 15 instruction = 1.53 (The `nop` instructions do not count in the calculation)

f) What we need to do is to replace some instructions such that the instructions we move do not have the dependencies that cause the hazards and conflicts and do not

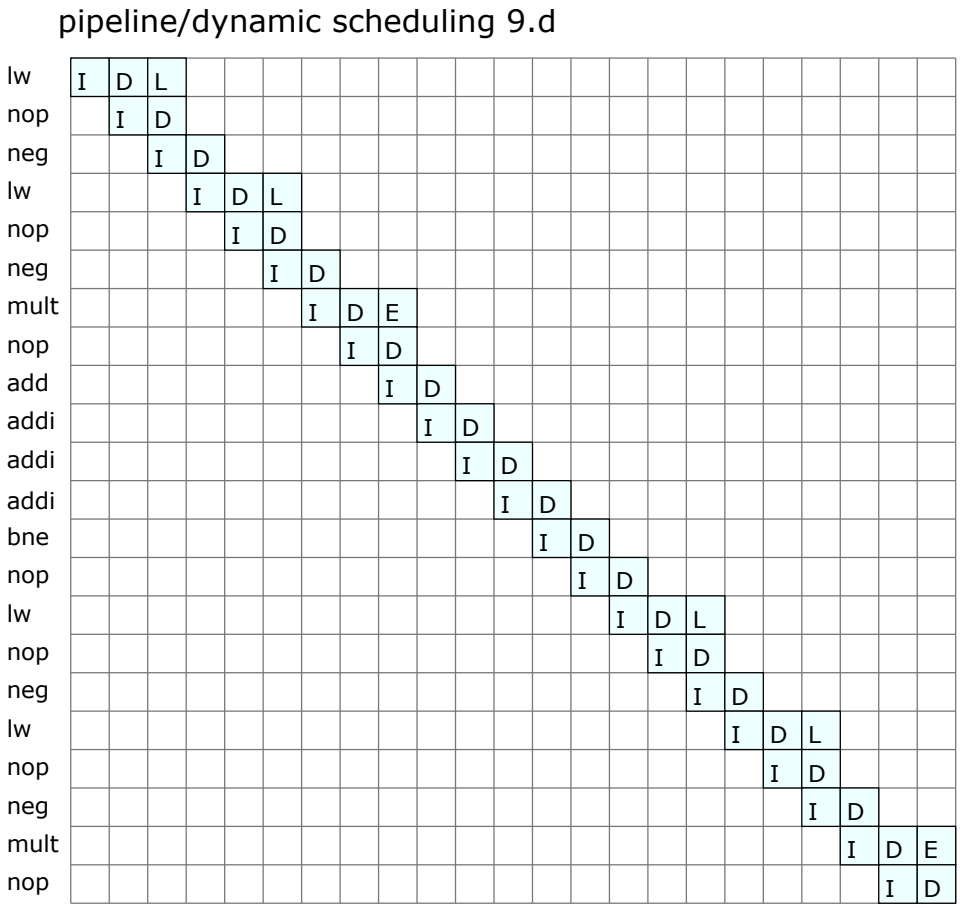


Figure 160: Execution diagram of modified code

change the functionality of the code, thus respect their own dependencies. We can move instruction 2 in the original code under instruction 0 since instruction 2 does not have any dependency on instructions 0 and 1. This move actually removes two **nop**'s from the code. One **nop** is filled by instruction 2 by being placed under instruction 0. The **nop** under instruction 2 becomes replaced by instruction 1.

Other than this move, instructions 6, 7 and 8 of the original code can be replaced. These instructions operate on a single register (read and write to the same register). Each of these instructions can be moved inside the region where the register they operate on is neither read nor modified. Thus, instructions 7 and 8 can be moved into the place of the **nop** that is under the **mult** instruction.

Similarly, instructions 6 and 7 can be moved into the place of the **nop** under the **bne** instruction. For the code presented below, one of the above replacement possibilities is selected and its execution diagram is depicted.

```
loop:   lw   t1, 0(t3)
        lw   t2, 0(t4)
        neg  t1, t1
        neg  t2, t2
        mult t1, t1, t2
        addi t5, t5, -1
        add  t3, t3, t1
        addi t3, t3, 4
        bne  t5, zero, loop
        addi t4, t4, 4
reinit: addi t5, zero, 10
```

g) In figure 161, the instructions which have been reordered are highlighted in green.

h) The CPI of this optimised code is: $CPI = 18\ cycles/15\ instructions = 1.2$

$Speedup = (1.53 - 1.2)/1.53 \approx 0.22$ (22% faster execution time)

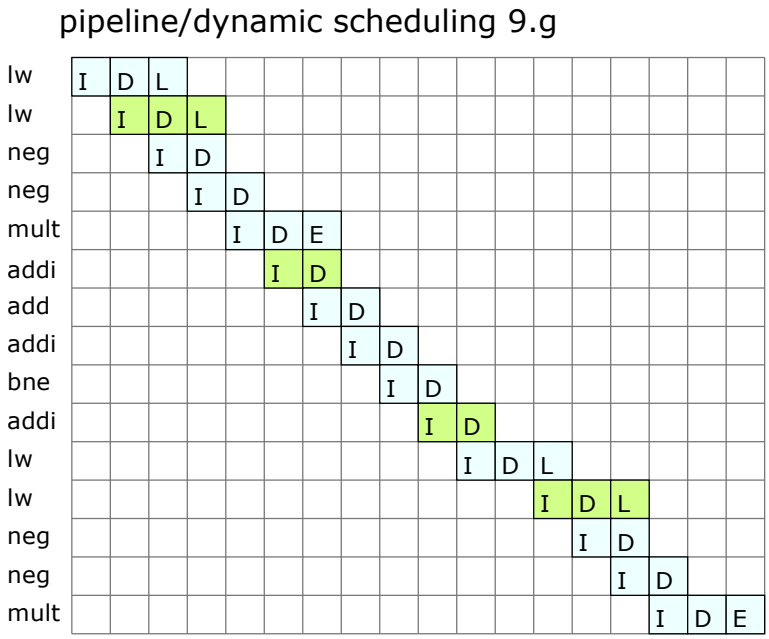


Figure 161: Execution diagram of optimised code

[Exercise 10]

Consider a processor which executes RISC-V instructions, and which has a six-stage pipeline: Fetch, Decode, Execute1, Execute2, Memory, Writeback. Logic operations (**and**, **or**, **sll**, ...) are done in **E1**, but the results from arithmetic operations are available at the end of **E2**. Jump destination gets loaded into the PC at the end of **E2**, and the processor has no control hazard logic (it cannot stall the pipeline).

Consider the following program (all constants are in decimal)

```
0         addi t0, zero, 100
1         addi t2, zero, 2
2         sw   t2, 900(t0)
3         addi t2, zero, 0
4         lw   t1, 1000(zero)
5   loop: add  t2, t2, t1
6         addi t1, t1, -1
7         bne  t1, zero, loop
8         add  t3, t2, zero
9         slli t2, t2, 2
10        add  t2, t2, t3
11        slli t2, t2, 2
```

a) What is the value of `t2` at the end of the execution ? Explain why.

b) Explain what characterises a RAW dependency. Does the instruction

```
          addi t1, t1, -1
```

have such dependencies? If so, specify which instruction it depends on.

c) Assume that another version of the processor has no forwarding paths, and cannot do register file bypass (Decode and Writeback cannot access the same register at the same time). The processor inserts stall cycles in case of data hazards. Draw the execution diagram showing the execution of the program. What is the IPC on this processor?

d) Consider a second version of our processor, for which we add a forwarding path from **E2** to **E1** (There is still no register file bypass available.) Modify the execution diagram, show where the forwarding path is used, and calculate the IPC.

e) Is it possible to make some changes to the second processor (without changing the pipeline structure, or the instruction latency) to make that code execute faster? Briefly present different solutions. Could jump prediction be useful?

f) Alternatively, assume that we do not change the second version of the processor: Could we optimise performance by reordering some instructions? If yes, show the optimised code and calculate the IPC. If not, explain why.

[Solution 10]

a) The execution and the results of each instruction for the given program is given below

```
        addi t0, zero, 100  # t0 <- 100
        addi t2, zero, 2    # t2 <- 2
        sw   t2, 900(t0)    # M[1000] <- 2
        addi t2, zero, 0    # t2 <- 0
        lw   t1, 1000(zero) # t1 <- M[1000]=2
loop:   add  t2, t2, t1     # t2 <- 2
        addi t1, t1, -1     # t1 <- 1
        bne  t1, zero, loop # go to loop
        add  t3, t2, zero   # t3 <- 2 (delay slot)
        slli t2, t2, 2      # t2 <- 8 (delay slot)
        add  t2, t2, t3     # t2 <- 10 (delay slot)
loop:   add  t2, t2, t1     # t2 <- 11
        addi t1, t1, -1     # t1 <- 0
        bne  t1, zero, loop # continue
        add  t3, t2, zero   # t3 <- 11 (delay slot)
        slli t2, t2, 2      # t2 <- 44 (delay slot)
        add  t2, t2, t3     # t2 <- 55 (delay slot)
        slli t2, t2, 2      # t2 <- 220
```

At the end of the execution, the value 220 is stored in register `t2`. Note that delay slots are always executed (whatever the result of the branch is) affecting the final value of register `t2`.

b) RAW (Read After Write) dependencies are characterised by an instruction reading a register which is modified (written) by a previous instruction.

```
        addi  t1, t1, -1
```

has RAW dependences on the following instructions:

```
        lw    t1, 1000(zero)
```

and itself (due to the loop). Note that the instruction **`bne`** `t1, zero, loop` has RAW dependency on this **`addi`** instruction.

c) Figure 162 shows the complete execution of the above code assuming the processor has no forwarding paths (including the register forwarding path).

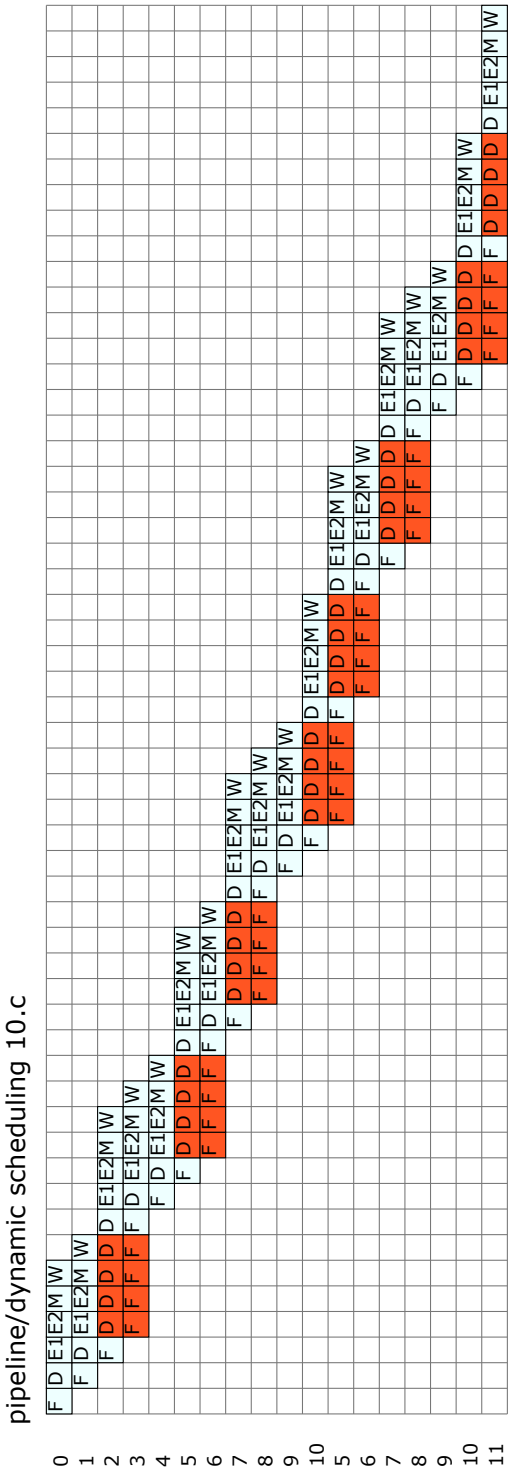


Figure 162: Execution diagram, no forwarding path

Notes on the organization of Figure 162:

- On the left of the diagram the code to be run can be found.
- The arrows shown on top of the diagram point to the earliest cycle end where a register which causes a RAW dependency is written to the register file. The red of these arrows depicts the end of cycles which frees the pipeline from stalling (while the grey dotted arrows do not have direct significance on the execution, and are just given for informative purposes).
- The cycles where the pipeline is stalled is shown with dark background
- The delay slots instructions are always executed after the branch.
- The fact that logic instructions are completed at the end of **E1** stage does not help avoiding the pipeline stalls. Since the pipeline is a rigid structure even if the result of the execution is obtained early it can only be written to the register file as if it is executed in two execution stages.

The IPC of this execution is (18 instructions)/(55 cycles) $\approx$ 0.33

d) Figure 163 on page 396 depicts the complete execution of the above code assuming the processor has a forwarding path between the end of the **E2** stage and the beginning of the **E1** stage.

The IPC of this execution is (17 instructions)/(40 cycles) $\approx$ 0.43

e) Since there is a single forwarding path these **E2** stage output values do not persist and are lost if the shown value can not be used in that cycle. There are such cases in the execution where the opportunity to use the forwarding path is missed because the forwarded value is only available at the beginning of one cycle (e.g., the forwarding path cannot be used for the `sw` instruction because it has a RAW dependency on the first 2 instructions). If the **E2** stage output cannot be used this is shown with a grey arrow while red arrows are used when the forwarding path can help avoiding stalls. The IPC of this processor pipeline version is approximately 0.43 as shown under the diagram. The improvement with respect to the first execution is (0.43 - 0.33)/0.33 = 0.3 = 30%.

f) Possible improvements:

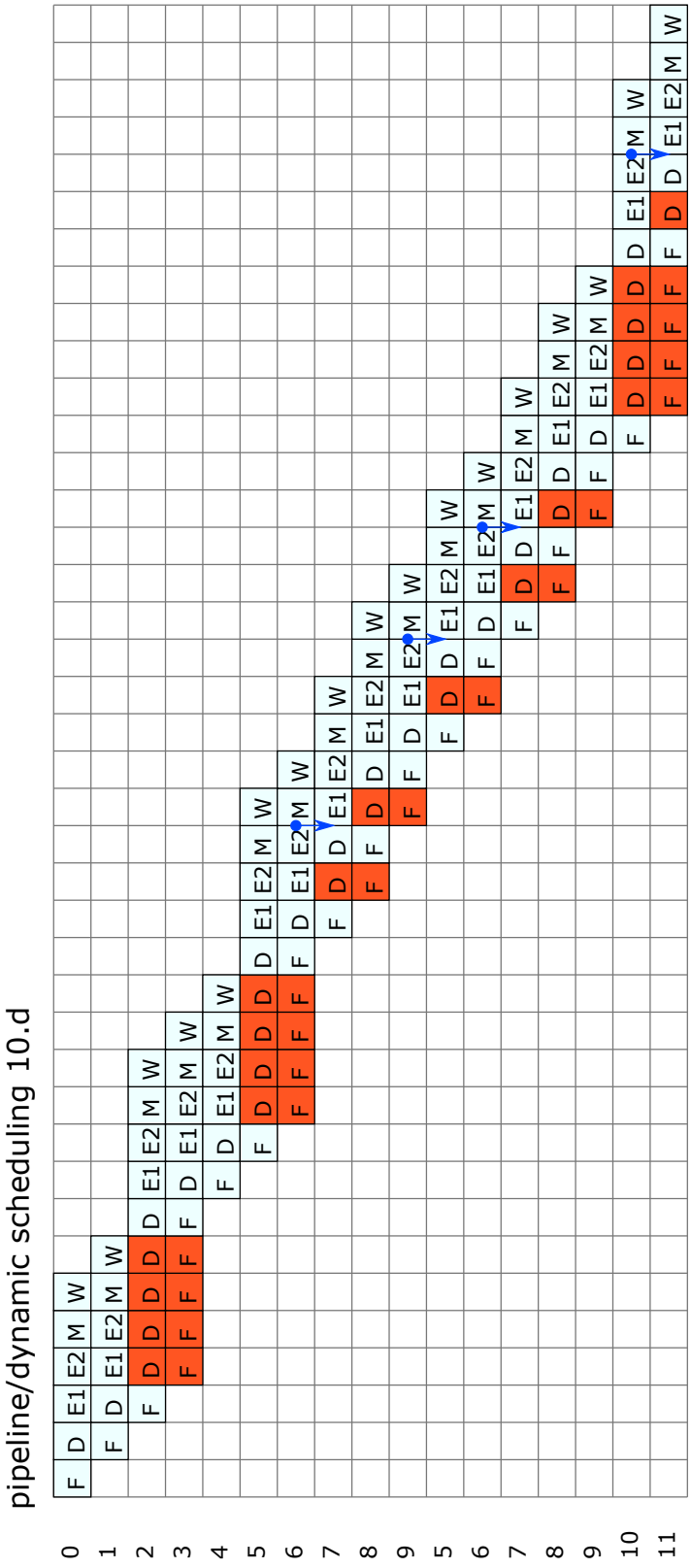


Figure 163: Execution diagram, with forwarding path E1 to E2

- Putting an extra forwarding path between the end of **M** stage and the beginning of **E1** stage. If the second execution is compared with the first we see that the single forwarding path is only useful if an instruction has a RAW dependence on a single previous instruction. If there are multiple instructions on which an instruction has RAW dependencies, then the forwarding path becomes useless. An example is the stall during the `sw` instruction. Having this extra forwarding path will avoid the stalls due to multiple RAW dependencies;
- Introducing an extra forwarding path between the end and the beginning of **E1** stage: This will provide the results of logical instructions to be transferred to the next instruction through the forwarding path thus avoid some of the stalls;
- Register file forwarding can of course be introduced to decrease the stall one cycle for stalls depending on the register file;
- Branch prediction will eliminate the use of delay slots. This simplifies the work of the compiler and the code becomes more readable. As for the performance, `nop`s will not be needed (when delay slots can not be filled with other instructions), which will lead to a better performance;

g) We cannot improve the code by reordering the instructions because the data dependencies of instructions are such that instructions can not be reordered without changing the semantics of the code (except for `addi t0, zero, 100` and `addi t2, zero, 2`, but still a change in the order does not bring any improvement).

[Exercise 11]

Consider a processor which executes RISC-V instructions, and has a five-stage pipeline: "Fetch", "Decode", "Execute", "Memory", "Writeback". RF reading is done during Decode, and writing during Writeback. There is a single forwarding path from **M** to **E**. Control hazards never stall the pipeline and the processor therefore needs two delay slots (the PC gets modified in case of jumps or branches at the end of Execute).

In addition to the usual RISC-V instructions, this processor also implements **`mult`** and **`multi`** which multiply two registers or a register with a constant respectively. These instructions take three cycles in the Execute stage instead of only one. During the first and second cycles, the pipeline gets stalled, and the other states (**F**, **D**, **M**, and **W**) are blocked.

Consider the following code:

```
2000: addi  t1, zero, 2
2004: addi  t0, zero, 1000
2008: lw    t2, 0(t0)
2012: multi t2, t2, 7
2016: addi  t1, t1, -1
2020: bne   t1, zero, 2008
2024: sw    t2, 100(t0)
2028: addi  t0, t0, 4
2032: addi  t0, t0, -4
```

a) Draw the execution diagram for the execution of that code (until the instruction at address 2032). Specify when the forwarding path gets used.

b) What is the CPI?

c) Why does this architecture have two delay slots, and not another amount?

You will now implement part of the control logic for the pipeline. More precisely, you have to write the logic which controls the pipeline progress according to the decoded instruction. When **D** decodes a new instruction, the **mcycle** signal gets asserted (**mcycle** $= 1$) if the instruction is a three-cycle instruction (in this case either **`mult`** or **`multi`**); or **mcycle** $= 0$ if it is a normal instruction. Using this signal, you must generate **advance**, which specifies if the pipeline must advance or not. Also, the two bit signal **cycle** must be generated for the Execute stage. For normal instructions, it has to be zero, whereas for three-cycle instructions it must count up to 2.

d) Complete the timing diagram with the **advance** and **cycle** signals:

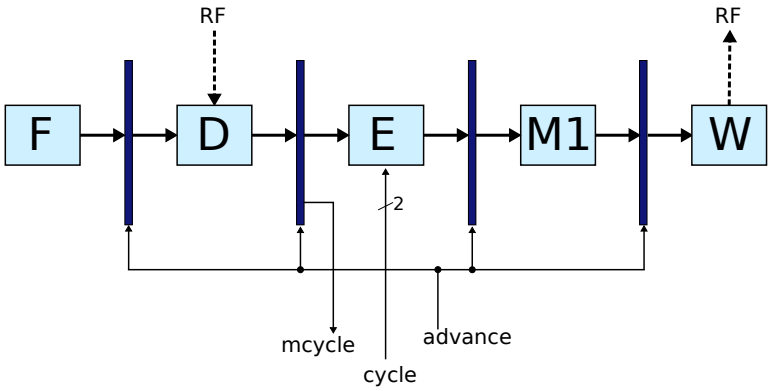


Figure 164: Pipeline and control signals

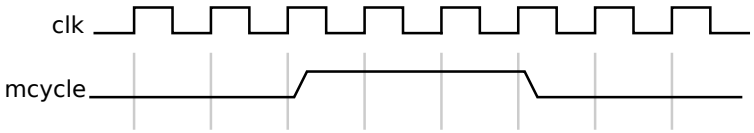


Figure 165: Timing diagram for the control signals

e) Draw the finite state machine which implements this behaviour, and generates the signals.

[Solution 11]

a)

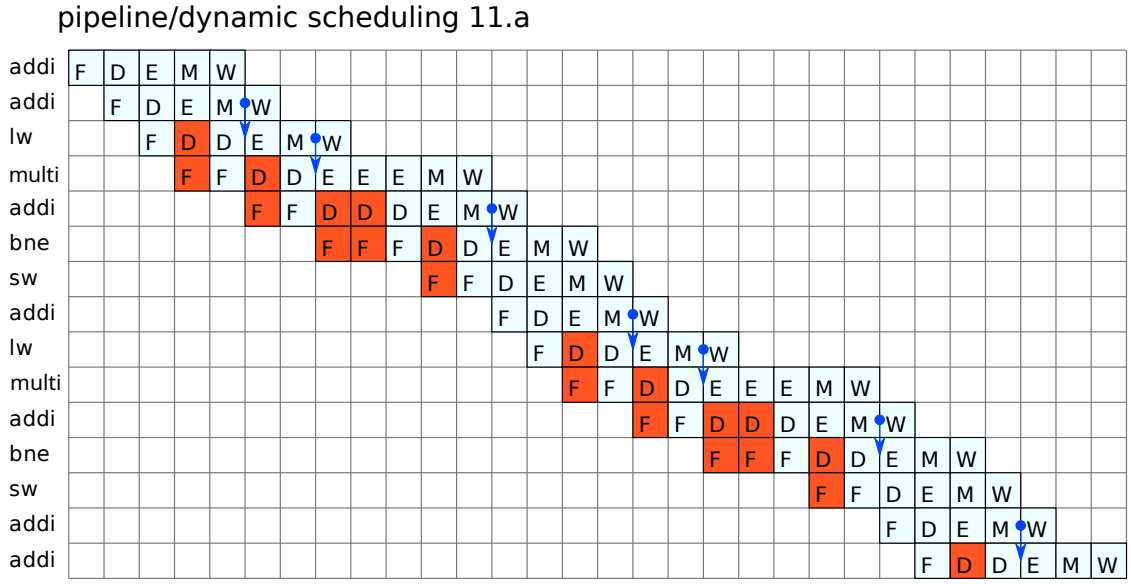


Figure 166: Execution diagram

b) CPI $= 30/15 = 2$

c) The **`sw`** and **`addi`** instructions at addresses 2024 and 2028 are instructions executed during delay slots.

By the time a branch instruction reaches the end of **E** stage, two more instructions have entered (e.g., in the above code **`sw`** and **`addi`** instructions) the pipeline. The slots occupied by these two instructions are called "delay slots".

d) The time diagram of (part of) the pipeline stall control is given below.

In Figure 167 we see the execution pipeline stage performing different operations for different instructions. The multiplication operation is performed during **Mul Instr Exec** cycles. As required the multiplication execution is performed in three cycles while operations corresponding to other instructions (such as **instr 1**, **instr 2** etc.) are performed in a single cycle. We can summarise the signal waveform for **Mul Instr Exec** as follows: In the first cycle the **mcycle** bit is set right after the clock since **mcycle** is a register bit. The **advance** signal becomes 0 together with the **mcycle** becoming 1 (at

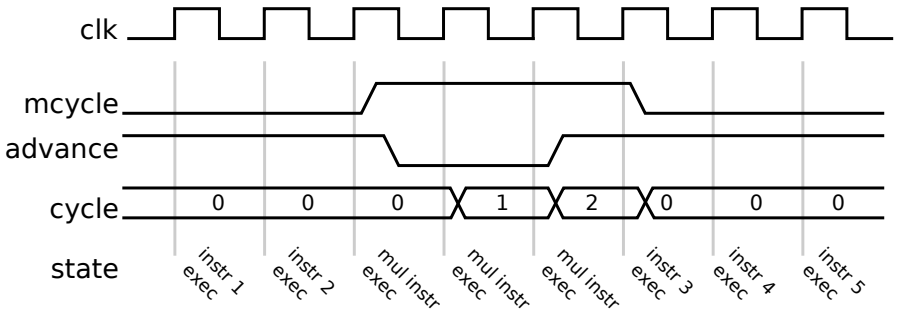


Figure 167: Solution of time diagram for pipeline control signals

this step advance is controlled by **mcycle**). This is required to stall the pipeline for the next cycle. At the beginning of the next cycle, the state machine detects that a multicycle operation is being performed (since **mcycle** $= 1$), keeps **advance** at level 0 and updates **cycle** as 1.

In the beginning of the 3rd cycle, the state machine sets the **advance** signal to 1 so that at the end of the 3rd cycle (at the beginning of the 4th cycle), information for the next instruction can be loaded to the execution stage input register. At the same time **cycle** is set to value 2. In the beginning of the 4th cycle, the state machine returns to the state where a single cycle operation is performed, setting **cycle** to 0 and allowing the **advance** signal being controlled by the **mcycle** signal (this is needed to restart the 3 cycle stall if there is again a multicycle operation). When we analyse the above behaviour we see that the state machine requires only three states and each state corresponds to a different value of the **cycle** signal. The state transitions of this state machine, as well as the outputs generated during each state are shown in the figure below. We see that the only input to the state machine is **mcycle**.

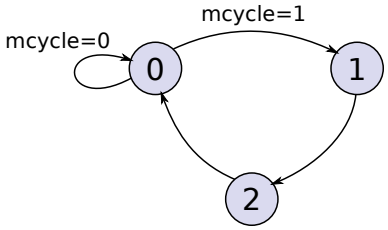


Figure 168: Finite state machine transitions

state	advance	cycle
0	$\overline{\text{mcycle}}$	0
1	0	1
2	1	2

Table 28: Output signals in each state

[Exercise 12]

Consider the following code (all constants are in decimal):

```
0            addi t0, zero, 100
1            addi t1, zero, 200
2            lw   t0, 0(t0)
3            lw   t1, 0(t1)
4  loop:     lw   t2, 0(t1)
5            add  t2, t2, t1
6            sw   t2, 0(t1)
7            addi t1, t1, 4
8            addi t0, t0, -1
9            bne  t0, zero, loop
10 end:      addi t3, zero, 400
```

This code will be executed on a RISC-V processor with a five-stage pipeline ("Fetch", "Decode", "Execute", "Memory", "Writeback"), which possesses the forwarding paths from **E** to **E**, and from **M** to **E**. It also has register file forwarding (Writes happen in the first half of **W**, reads happen in the second half of **D**). Jump decisions are taken at the end of **E** when the destination address is written into the PC. The **F** state is blocked until the correct jump address is known every time a jump or a branch is decoded. The processor has separate data and instruction caches, which allows **F** and **M** to be independent (like in usual RISC-V pipelines).

a) Give all RAW dependencies in this code.

b) Draw the execution diagram of the code, running on the specified processor. Stop your simulation at the instruction at label `end`. Assume that $M[100] = 2$. Specify which forwarding paths are used. What is the CPI of this program?

Assume now that the processor has a unified data and instruction cache, which is single-ported. **F** and **M** cannot access the cache at the same time. This can lead to structural hazards, which have to be solved. These situations happen when a **`lw`** or **`sw`** is in **M**. The pipeline controller must insert a stall either at **F** or **M**.

c) In what stage must the stall be inserted? Why? Draw the execution diagram for the code using the modified pipeline. Specify all forwarding paths that have been used. What is the CPI?

To lower the branch penalty, a branch predictor is added to the processor. It is implemented using a single bit. When a jump instruction is in **F** and the prediction bit is 1, the controller writes the destination address into the PC at the end of **F**. If the bit is 0,

the PC is incremented as usual.

If at the end of **E** we notice that the prediction was wrong, the correct address is written into the PC and the end of **E** and the instructions in the pipeline are flushed.

The prediction bit gets inverted when the prediction was wrong, and stays at its value if the prediction was correct.

d) Suppose that the initial value of the prediction bit is 1 and that $M[100] = 2$. Calculate the CPI for the processor with branch predictor (running the same code as above). Hint: Use the execution diagram from the preceding question and show the differences, instead of drawing a new one.

[Solution 12]

a) Dependencies are given in the following graph. The labels on the arrows represent the register name creating the dependency.

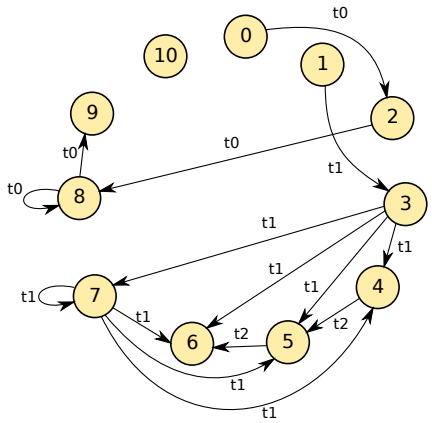


Figure 169: RAW dependencies

b) On Figure 170, blocked stages are represented by red backgrounds. It can be seen that there are blocked stages because of jump and memory instructions even though all forwarding paths are available. Jump instructions stall the pipeline until they get to the end of **E**.

c) If **M** is chosen to stall allowing **F** to access the cache, the whole pipeline will be blocked. Therefore, the controller must block **F**. Figure 171 shows execution with a unified cache.

The cycles where **F** is blocked to let **M** access the cache are shown against a green background. All other blocked states (because of RAW dependencies or jump hazards) are highlighted in red. The CPI is now $CPI = 32$ cycles$/17$ instructions ≈ 1.88.

d) Since the prediction bit is 1 at the beginning, the prediction is correct on the first time, but wrong for the second time. We lose two cycles in case of a prediction error, because the PC gets updated two cycles after **F**, at the end of **E**. In case of a correct prediction, we do not lose any cycles. Therefore, compared to the situation without branch predictor, the execution takes two cycles less.

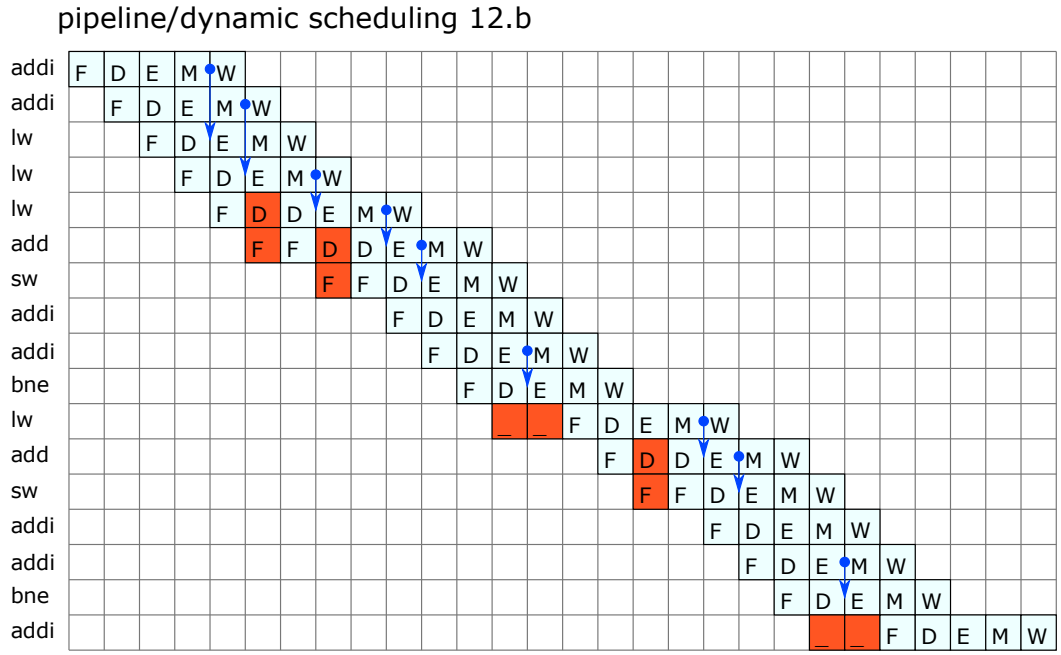


Figure 170: Execution diagram

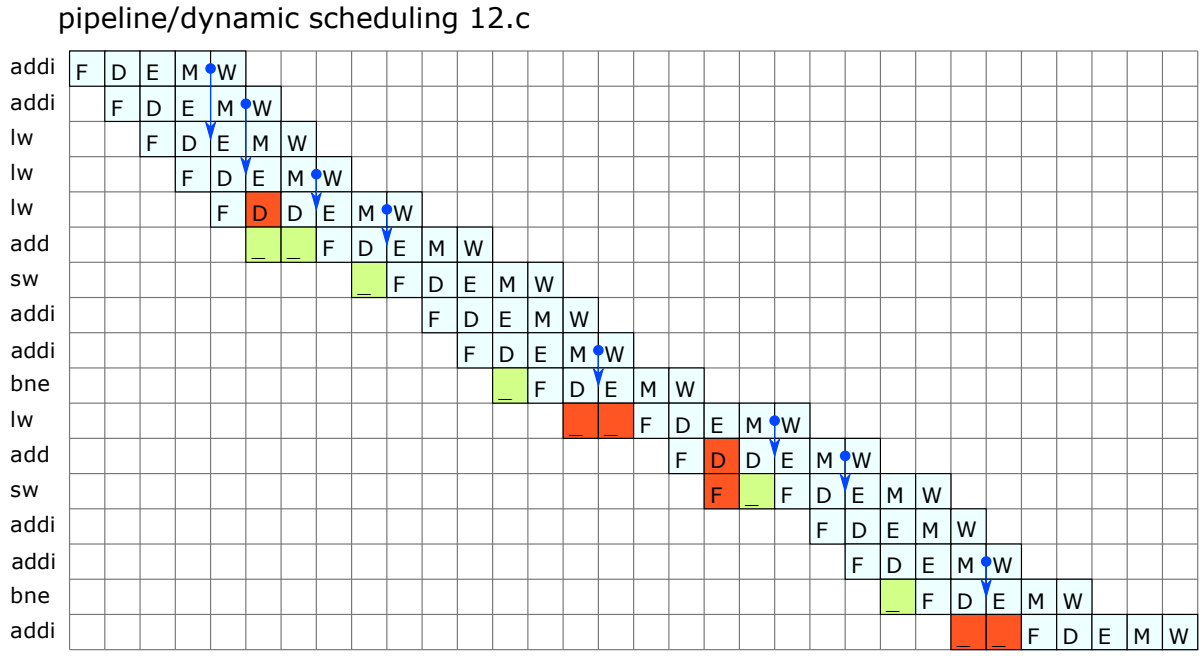


Figure 171: Execution diagram with modified pipeline

[Exercise 13]

Consider a pipeline consisting of seven stages: "Fetch", "Decode", "Execute1", "Execute2", "Memory1", "Memory2", "Writeback". All arithmetic operations except multiplication are done in **E1**, multiplication results are ready at the end of **E2**. Memory operations always need both **M1** and **M2**.

The following processor implements the pipeline described above and executes two programs:

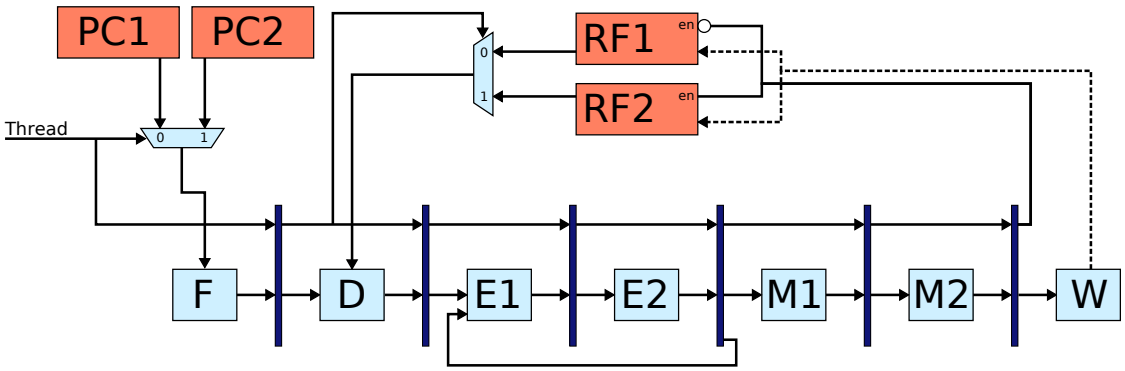


Figure 172: 2-way multithreaded pipeline

This structure is called "2-way multithreaded". The processor has two register files (**RF1** and **RF2**) and two program counters (**PC1** and **PC2**), which are used for programs 1 and 2, respectively. The **Thread** signal selects from which program the next instruction is fetched, and changes its value at every cycle. This signal also is inserted in the pipeline and moves forward along the instruction, because we have to remember to which program the instruction belongs: It is necessary to read and to write to the correct **RF** during **D** and **W**. In case of a data hazard, the pipeline is stalled as usual, and **Thread** stays at its value.

For this exercise, you can assume that a new instruction **`mul`** has been added to the RISC-V ISA. This instruction is defined the following way: **`mul`** `rA, rB, rC`, which corresponds to the pseudocode: `rA = rB * rC`. The **`subi`** instruction was also added to perform subtraction with an immediate value. **`subi`** `rA, rB, imm` corresponds to `rA = rB - imm`

Consider the following programs:

Program A

```
1    addi    t0, t0, 4
2    add     t1, t1, t0
3    add     t2, zero, t0
4    lw      t0, 0(t1)
5    mul     t3, t2, t0
6    add     t0, zero, t2
7    add     t3, t3, t0
```

Program B

```
1    lw      t1, 0(t0)
2    lw      t2, 4(t0)
3    addi    t1, t1, 4
4    add     t3, t1, t2
5    subi    t3, t3, 4
6    sw      t3, 0(t2)
```

a) Assume that the only forwarding path is **E2** $\rightarrow$ **E1**, as mentioned on the figure. (There is no register file forwarding). Draw the execution diagram showing the execution of the two programs on that processor. The processor starts by fetching the first instruction of program A and fetches the first instruction of program B during the second cycle. Specify when the forwarding path is used.

b) If we added the forwarding path from **E1** to **E1**, would we get a better performance? If yes, present a simulation showing such a situation. If no, explain why.

c) If we added the forwarding path from **M2** to **E1**, would we get a better performance? If yes, show where this forwarding path would be useful in the simulation or give a simulation in which it would be useful. If no, explain why.

d) If we added register file forwarding, would we get a better performance? If yes, show where this forwarding path would be useful in the simulation or give a simulation in which it would be useful. If no, explain why.

e) Assume that we change the arithmetic unit with a smaller one, but for which all operations take two cycles. All results are ready at the end of **E2**. What consequences would that change have on performance? Explain your answer. Explain in your own words what is the usefulness of a multithreaded processor.

[Solution 13]

a) Figure 173 shows the execution of both programs, where program A is in blue, and program B in yellow. From the diagram, we can conclude that execution lasts 29 cycles.

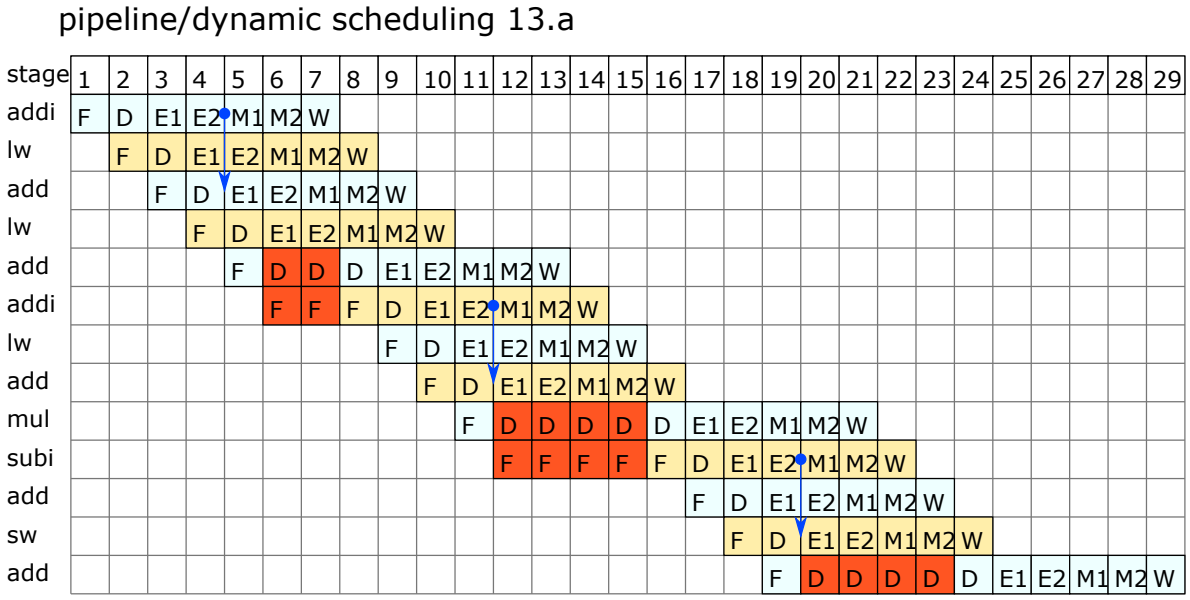


Figure 173: Execution diagram

b) The forwarding path from **E1** to **E1** is useless because fetch alternates between the two programs. Therefore we cannot forward any data back to the same stage, because it will contain an instruction from the other program during the next cycle. The data has to be given to an instruction from the same program.

c) The forwarding path from **M2** to **E1** can be useful to improve the performance. In case of a stall, waiting time can be reduced, and it can also avoid stalls. It is useful because the data we have at **M2** belongs to the same program as those in **D**, which might have a dependency on it. During the next cycle, the data will be available on the forwarding path, and the instruction having the dependency will be in **E1** and will be able to use the data from the forwarding path. Therefore, it is possible to give data not to the next instruction, but to the one after.

It is also possible to find a situation in the execution diagram for the two programs for which the existence of the forwarding path would lower the execution time. These situations are represented by red arrows on the following diagram. The red crosses show which stages would disappear with the changes.

Figure 174 shows the forwarding path possibilities brought by the **M2** to **E1** forwarding path, and Figure 175 shows the execution once the paths are implemented.

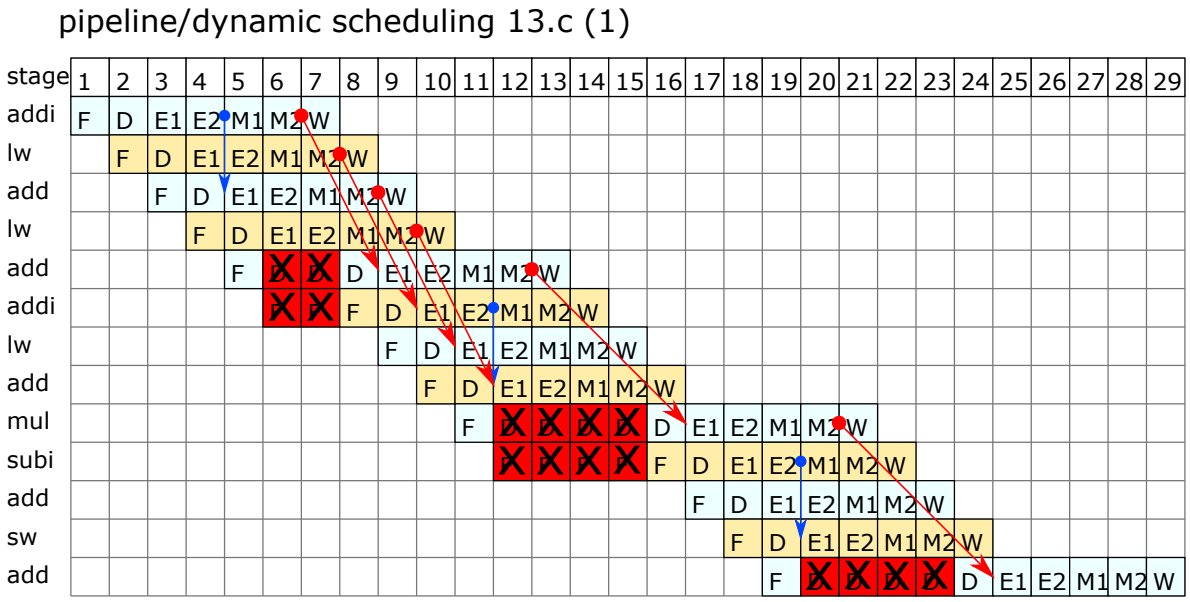


Figure 174: **M2** to **E1** forwarding path possibilities

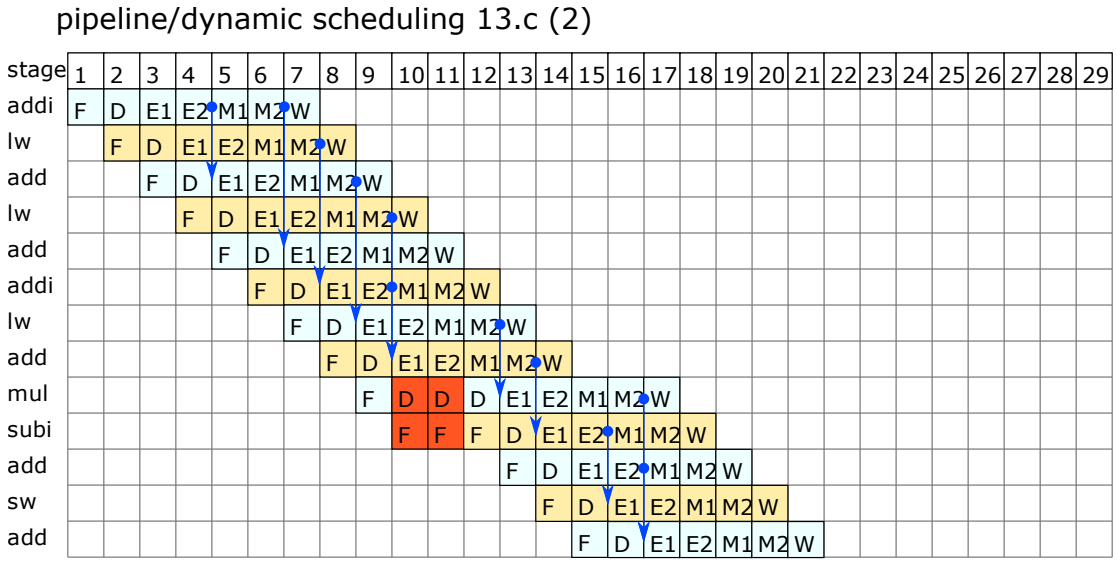


Figure 175: Execution diagram with **M2** to **E1** forwarding paths implemented. Note that the forwarding path from `add` to `mul` can't be used as there is still a dependency with `lw`.

d) This forwarding path can improve performance because it can lower the wait time in case of a stall. Figure 176 shows situations where register file forwarding is useful and Figure 177 shows the execution once the register file forwarding is implemented.

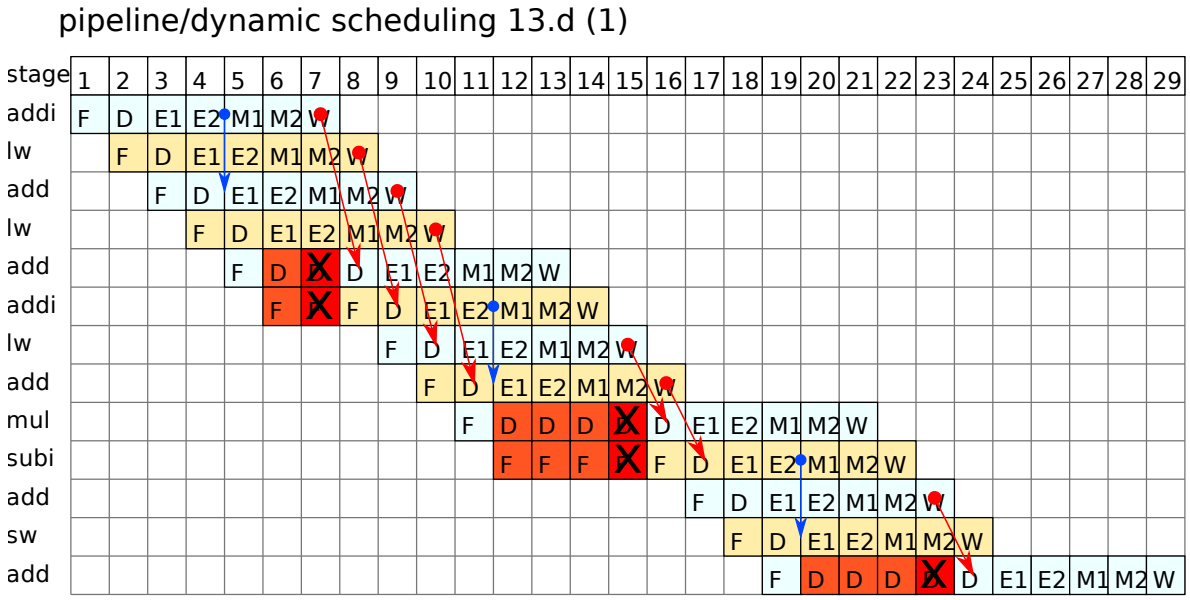


Figure 176: Register file forwarding possibilities

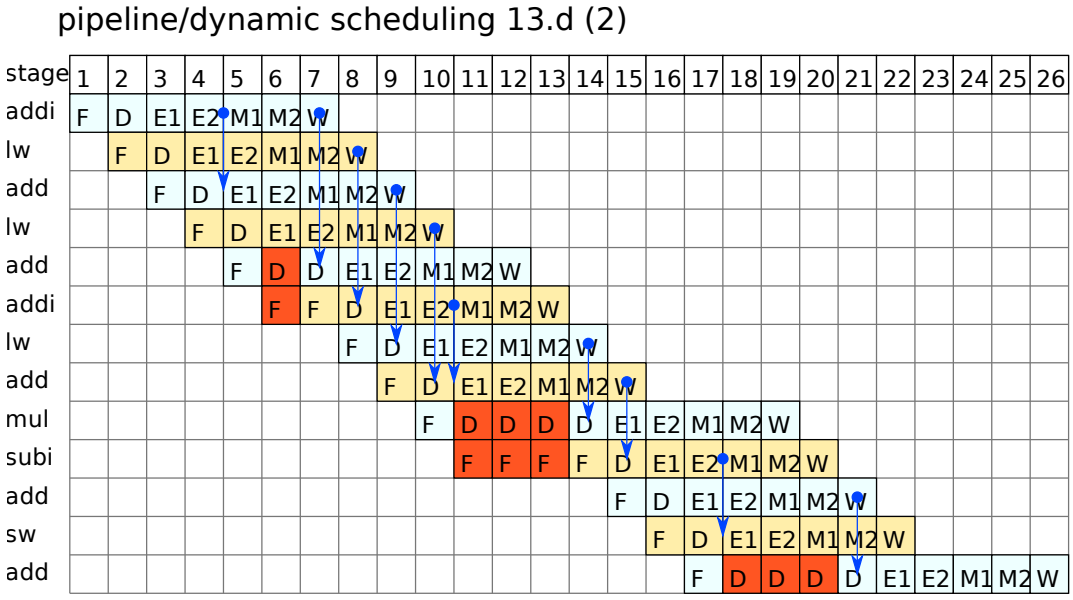


Figure 177: Execution diagram with Register File forwarding paths implemented

e) For the first processor, even though most operations complete in **E1**, they still have to traverse **E2**. It does not matter if calculations are performed in **E2** or not. Changing the execution unit only changes how the calculations are distributed, but does not change their latency. Therefore, total cycle count and performance will not change.

The idea behind multithreaded processors is to insert independent instructions into the pipeline to reduce the number of data hazards. Since the register files are distinct, the instructions coming from different programs become totally independent. If several mutually independent instruction streams are interleaved, the distance between the dependencies increases, which makes them easier to solve, or even makes them disappear. At the limit, if we have a n-way multithreaded processor, where n is larger or equals the number of stages in the pipeline, it will be impossible to have data hazards. In that case, there can only be one instruction from some program at a time in the pipeline. Unfortunately, this only works if there are n threads scheduled for execution.

[Exercise 14]

In this question, you will use a VLIW processor to evaluate the values of the first few Fibonacci numbers. The Fibonacci numbers are defined by the recurrence relation

$$F_i = F_{i-1} + F_{i-2},$$

where $F_0 = F_1 = 1$. To evaluate the first n Fibonacci numbers iteratively, we can write a C code like this:

```
1 F[0] = 1; F[1] = 1;
2 for (i=2 ; i<n ; i++)
3     F[i] = F[i-1] + F[i-2];
```

Assuming that the `F[]` will be stored to the memory starting from the address 0x1000, for $n = 6$ the corresponding assembly code for RISC-V would be like this:

```
1  addi  t0, zero, 1
2  sw    t0, 0x1000(zero)
3  addi  t1, zero, 1
4  sw    t1, 0x1004(zero)
5  addi  t5, zero, 16
6  add   t6, zero, zero
7  loop:
8  lw    t0, 0x1000(t6)
9  lw    t1, 0x1004(t6)
10 add   t2, t0, t1
11 sw    t2, 0x1008(t6)
12 addi  t6, t6, 4
13 bltu  t6, t5, loop
```

Our VLIW processor uses the same assembly except that we can specify which instructions could be run in parallel. To do this, we write each block of instructions that could run in parallel on a single line and separated by `&`. For example, assuming that the instructions on lines 1 and 3 could be run in parallel and so do the ones on 2 and 4, we can replace the first 4 lines of the code with:

```
1 addi  t0, zero, 1       & addi  t1, zero, 1
2 sw    t0, 0x1000(zero)  & sw    t1, 0x1004(zero)
```

Missing instructions in a block will be automatically filled with **nop**. A line with a single `&` character means all **nop**.

For all the questions, at the end of the execution, the values of the F array should be in the memory starting from the address 0x1000. You can use registers freely, i.e., you

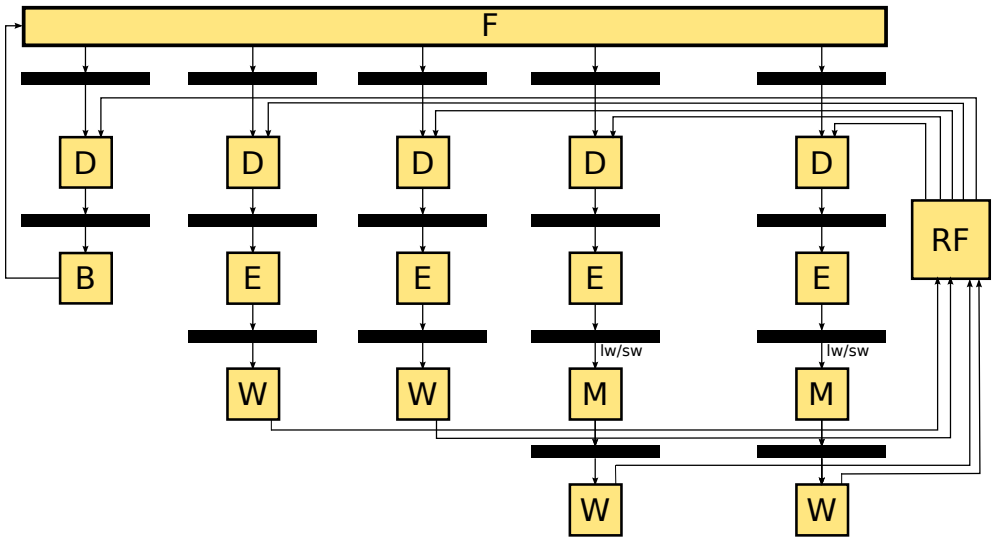


Figure 178: VLIW Processor Pipeline.

can assume that all the temporary registers `t0`-`t6` are free and need not to be restored at the end.

a) Assume that our processor has 2 *load/store* units, 1 *branch/jump* unit, and 2 *ALU* as shown in the Figure 178. It has RISC-V like 5 stage pipeline: *Fetch*, *Decode*, *Execute*, *Memory*, and *WriteBack* where all *forwarding paths* (of a classical RISC-V processor) are enabled (E→E, M→E, W→D) (they're not shown on the figure). For the *branch/jump* unit, we have E→B and M→B forwarding paths enabled.

You can also assume that you have a branch predictor that always evaluates to "true" and in the case of a branch instruction the next instruction is fetched without any delay slots. The *Fetch* and *Decode* stages will be flushed if the prediction is wrong. For example, in the given code, the instruction on line 8 would come after the branch on line 13 without any delay slots and the mentioned flush will occur at the end of the loop.

In the given assembly code, considering only one iteration of the loop (lines 8 to 13), show all the (intra-loop) dependencies and write that part of the code (without modifying the instructions) in our VLIW assembly to get the maximum IPC using parallelism. You can reorder the instructions to construct the blocks properly. Draw the execution diagram of one iteration, showing the pipeline stages and *forwarding paths* used.

b) Now, modify the code for the loop in any possible ways (lines 7 to 13) to finish execution in the least possible number of clock cycles for the case of $n = 6$. This time, you are allowed to remove or modify the instructions. Write your resulting code in our VLIW assembly and draw the execution diagram of the whole loop.

c) Assume that the value of n is determined dynamically and it is loaded to `t5` prior to the start of the code snippet; i.e., line 5 of the code is removed and `t5` is assumed to be determined during run time. Also assume that we now have one *load/store* less, i.e., a processor with 1 *load/store*, 1 *branch*, and 2 *ALU*. Modify the initial code for the loop to make use of the parallelism as much as possible. Write your resulting code in our VLIW assembly. Compare the number of clock cycles required to complete executing this code with the code you have written for question 1 for $n = 1000$ (you can ignore overflow problems).

[Solution 14]

a) The dependencies are as follows (the numbers represent the line numbers of the instructions):

- **RAW:**
  - 8→10 for `t0`
  - 9→10 for `t1`
  - 10→11 for `t2`
  - 12→13 for `t6`
- **WAR:**
  - 8→12 for `t6`
  - 9→12 for `t6`
  - 11→12 for `t6`
- **WAW:** Nothing.

The possible solution for the maximum IPC is given below.

```
1 lw   t0, 0x1000(t6) & lw  t1, 0x1004(t5)
2 &
3 add  t2, t0, t1
4 sw   t2, 0x1008(t6) & addi t6, t6, 4
5 bltu t6, t5, loop
```

The execution diagram for this code is given in Figure 179. There are two M→E, one E→E, and one E→B forwardings used.

b) Since we are given in the question that only the values in the memory at the end of the execution is important but not the values in the registers, we can change our code as following. Note that, we used loop unrolling and in addition, we avoided the unnecessary load instructions.

```
1 add t2, t0, t1
2 sw  t2, 0x1008(zero) & add t3, t1, t2
3 sw  t3, 0x100c(zero) & add t4, t2, t3
4 sw  t4, 0x1010(zero) & add t5, t3, t4
5 sw  t5, 0x1014(zero)
```

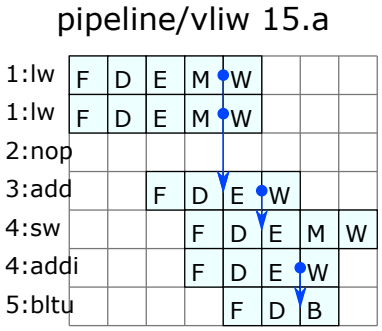


Figure 179: Execution Diagram of **a)**.

The execution diagram is given in Figure 180. There are seven E→E and two W→D forwardings used.

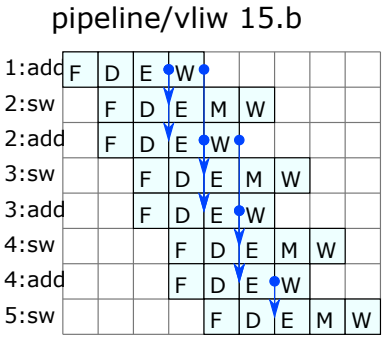


Figure 180: Execution Diagram of **b)**.

c) We can avoid unnecessary load instructions since the numbers are calculated in the loop and the results can be used directly from the registers. In addition, to eliminate the dependency caused by the register `t6`, which is used for address calculation, we can calculate the value of it at the beginning of the loop and adjust the memory offset accordingly. The new code is:

```
1 loop:
2 addi t6, t6, 4       & add  t2, t0, t1
3 add  t0, t1, zero    & add  t1, t2, zero
4 sw   t2, 0x1004(t6)  & bltu t6, t5, loop
```

The first implementation of the loop was taking $5 \cdot 1000 + 3 = 5003$ clock cycles in the first processor. The new implementation takes $3 \cdot 1000 + 4 = 3004$ clock cycles in the new processor. So, in spite of having one *load/store* unit less, we still have approximately 40% performance gain.

[Exercise 15]

Consider the superscalar processor of Figure 181 with the following properties:

- The Fetch and Decode Unit outputs up to 3 instructions per cycle. If one of the fetched instructions is a branch, then the successive instructions are not fetched, nor decoded. A branch predictor is available. The predicted branch destination is fetched on the cycle following the fetch of the branch instruction.
- The following functional units are available:
  - Arithmetic Logic Unit (ALU) that executes arithmetic and logic instructions, as well as branching instructions.
  - Multiplication unit that is used for multiplication.
  - Load/Store unit that is used for memory access.

  Each functional unit has a reservation station that can store up to 20 instructions.
- The processor has a reorder buffer for preserving in-order instruction commit. The reorder buffer can store 25 instructions and can commit up to 3 instructions per cycle.
- The result buses carry results generated by the functional units to the reservation stations. If the functional unit requires more than one cycle, then the results are available at the end of the last cycle.
- The implemented pipeline starts with a stage for fetching (F) followed by a stage for decoding (D) the instruction. Then, one or more stages for execution follow. The instructions executed by the ALU require a single execution stage (EA), the multiplication needs two execution stages (E1, E2) and the memory accesses require one address computation (EM) and one memory stage (M). Finally, the writeback stage (W) commits the instruction and deletes it from the reorder buffer.
- The multiplication unit can execute just one multiplication instruction at a time. On the other hand, the load/store unit can execute two instructions at a time—one in the address computation stage and one in the memory stage.
- The processor executes RISC-V instructions. You can assume that a new instruction `mult` has been added to the RISC-V ISA. This instruction is defined as: `mult rA, rB, rC`, corresponding to the pseudocode: `rA = rB * rC`.

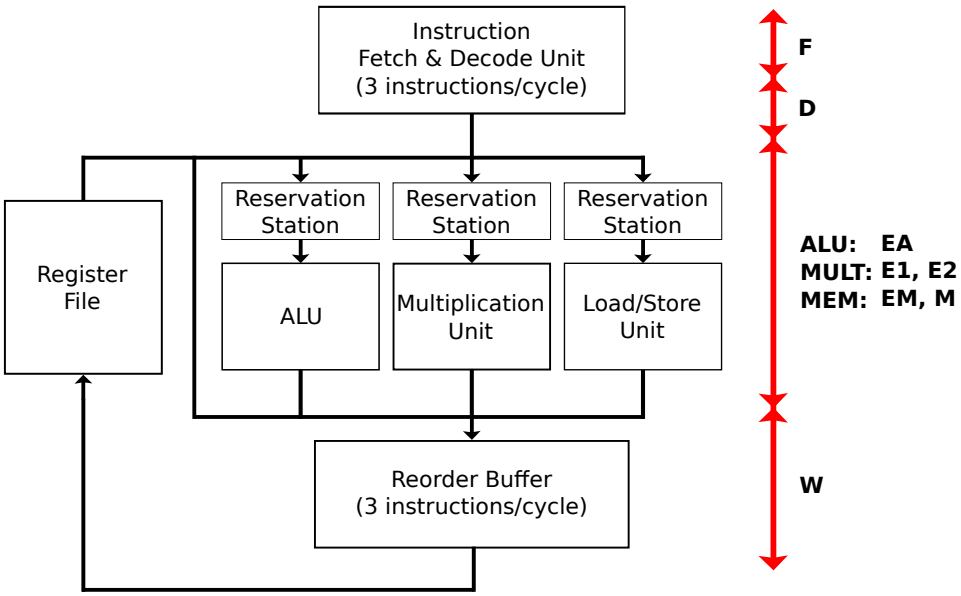


Figure 181: The superscalar processor.

Consider the following section of code.

```
01  loop:  lw   t1, 0(s1)
02         lw   t2, -4(s1)
03         mult t3, t2, t2
04         add  t3, t3, t1
05         sw   t3, 0(s2)
06         addi s2, s2, -4
07         addi s1, s1, -8
08         bne  s1, zero, loop
```

a) Simulate three executions of the loop using the superscalar processor. Assume that there is a perfect cache that always hits and that a perfect branch predictor decided correctly that the branch is taken twice.

i) Draw an execution table which represents each cycle of the processor. Mark with x the stalls resulting from data hazards and with ○ the stalls resulting from structural hazards including the case of instructions not ready for a commit. To answer this question, use the provided template.

ii) What is the achieved CPI?

iii) What would be the ideal CPI of this processor?

b) Find in which cycle the writeback (W) stage of the instruction from line 5

(`sw t3, 0(s2)`) from the first execution of the loop is executed. Consider the situation of the reorder buffer at the beginning of this cycle. For referencing the instructions when answering this question, use the instruction numbers given in the answer template for question 1.

i) List all instructions for which there is a valid entry in the reorder buffer.

ii) Which instructions have their results available in the reorder buffer?

iii) Which instructions are ready to commit assuming that the reorder buffer has enough bandwidth?

iv) Consider now two cycles later, as before, focus on the beginning of the cycle. List the instructions whose entries in the reorder buffer were modified. For each instruction specify shortly the reason for the modification.

c) In the lectures it has been shown that loop unrolling can increase the performance of VLIW processors. We want to check whether it can also increase the parallelism of instruction execution for superscalar processors, by simulating what a compiler could do. Assume that `s1` is a multiple of 24 at the beginning.

i) Thus, unroll the loop twice to have three executions of the original code. Rename registers, remove redundant instructions, if any, and reorder the code to increase the performance. Write the resultant code in the provided template. It is not necessary for the code to be optimal, but try to improve the performance.

ii) Draw an execution table which represents each cycle of the superscalar processor executing the optimised code from (i). Mark with `x` the stalls resulting from data hazards and with `o` the stalls resulting from structural hazards. For drawing the execution table, use the provided template.

iii) What is the new CPI?

iv) What is the speedup gained with the loop unrolling and the performed optimisations? What is the main reason for this speedup?

Execution Diagram

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	lw t1, 0(s1)																														
2	lw t2, -4(s1)																														
3	mult t3, t2, t2																														
4	add t3, t3, t1																														
5	sw t3, 0(s2)																														
6	addi s2, s2, -4																														
7	addi s1, s1, -8																														
8	bne s1, zero, loop																														
9	lw t1, 0(s1)																														
10	lw t2, -4(s1)																														
11	mult t3, t2, t2																														
12	add t3, t3, t1																														
13	sw t3, 0(s2)																														
14	addi s2, s2, -4																														
15	addi s1, s1, -8																														
16	bne s1, zero, loop																														
17	lw t1, 0(s1)																														
18	lw t2, -4(s1)																														
19	mult t3, t2, t2																														
20	add t3, t3, t1																														
21	sw t3, 0(s2)																														
22	addi s2, s2, -4																														
23	addi s1, s1, -8																														
24	bne s1, zero, loop																														

Execution Diagram

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

[Solution 15]

a)

i) Figure 182 shows the execution diagram.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	lw t1, 0(s1)	F	D	EM	M	W														
2	lw t2, -4(s1)	F	D	o	EM	M	W													
3	mult t3, t2, t2	F	D	x	x	x	E1	E2	W											
4	add t3, t3, t1		F	D	x	x	x	x	EA	W										
5	sw t3, 0(s2)		F	D	x	x	x	x	x	EM	M	W								
6	addi s2, s2, -4		F	D	EA	o	o	o	o	o	o	W								
7	addi s1, s1, -8			F	D	EA	o	o	o	o	o	W								
8	bne s1, zero, loop			F	D	x	EA	o	o	o	o	o	W							
9	lw t1, 0(s1)				F	D	EM	M	o	o	o	o	W							
10	lw t2, -4(s1)				F	D	o	EM	M	o	o	o	W							
11	mult t3, t2, t2				F	D	x	x	x	E1	E2	o	o	W						
12	add t3, t3, t1					F	D	x	x	x	x	EA	o	W						
13	sw t3, 0(s2)					F	D	x	x	x	x	x	EM	M	W					
14	addi s2, s2, -4					F	D	EA	o	o	o	o	o	o	W					
15	addi s1, s1, -8						F	D	o	EA	o	o	o	o	W					
16	bne s1, zero, loop						F	D	x	x	EA	o	o	o	o	W				
17	lw t1, 0(s1)							F	D	x	EM	M	o	o	o	W				
18	lw t2, -4(s1)							F	D	x	o	EM	M	o	o	W				
19	mult t3, t2, t2							F	D	x	x	x	x	E1	E2	o	W			
20	add t3, t3, t1								F	D	x	x	x	x	x	EA	W			
21	sw t3, 0(s2)								F	D	x	x	x	x	x	x	EM	M	W	
22	addi s2, s2, -4								F	D	o	o	EA	o	o	o	o	o	W	
23	addi s1, s1, -8									F	D	o	o	EA	o	o	o	o	W	
24	bne s1, zero, loop									F	D	x	x	x	EA	o	o	o	o	W

Figure 182: The execution diagram when the loop is executed three times.

ii) The CPI for the three executions of the loop shown on Figure 182 is

CPI = 19 cycles / 24 instructions = 0.792 cycles/instruction

iii) The given superscalar processor has 3 functional units. Thus, in an ideal situation, it would execute 3 instructions per cycle, from what follows that the ideal CPI would be

CPI_{ideal} = 1 cycles / 3 instructions = 0.333 cycles/instruction

b) The writeback stage (W) of the instruction from line 5 from the first execution of the loop is executed in cycle 11.

i) The reorder buffer has a valid entry for each instruction that is fetched, but that is still not committed. At the beginning of cycle 11, the reorder buffer keeps entries for instructions 5-24.

ii) In the reorder buffer, the result for an instruction is available after its last execution stage ends. At the beginning of cycle 11, instructions 5-11 and 14-16 have their results available in the reorder buffer.

iii) Ready to commit are the instructions that will be committed in the following cycle. Assuming that the reorder buffer has enough bandwidth, at the beginning of cycle 11, the instructions 5-11 are ready to commit.

iv) At the beginning of cycle 13, the following instructions have their entries modified.

- The instructions 5-10 committed, thus their entries are invalidated.
- The execution stages of instructions 12, 17, 18, and 22 finished and they have their results available.

c)

i)
```
01  loop: lw   t2, -4(s1)
02        lw   t1, 0(s1)
03        addi s1, s1, -24
04        addi s2, s2, -12
05        mult t3, t2, t2
06        lw   t5, 12(s1)
07        lw   t4, 16(s1)
08        add  t3, t3, t1
09        mult t6, t5, t5
10        lw   s5, 4(s1)
11        lw   s4, 8(s1)
12        add  t6, t6, t4
13        mult s6, s5, s5
14        sw   t3, 12(s2)
15        sw   t6, 8(s2)
16        add  s6, s6, s4
17        sw   s6, 4(s2)
18        bne  s1, zero, loop
```

ii) Figure 183 shows the execution diagram of the optimised code.

iii) The new CPI for the optimised code shown in Figure 183 is

CPI = 14 cycles / 18 instructions = 0.778 cycles/instruction

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	lw t2, -4(s1)	F	D	EM	M	W									
2	lw t1, 0(s1)	F	D	o	EM	M	W								
3	addi s1, s1, -24	F	D	EA	o	o	W								
4	addi s2, s2, -12		F	D	EA	o	W								
5	mult t3, t2, t2		F	D	x	E1	E2	W							
6	lw t5, 12(s1)		F	D	o	EM	M	W							
7	lw t4, 16(s1)			F	D	o	EM	M	W						
8	add t3, t3, t1			F	D	x	x	EA	W						
9	mult t6, t5, t5			F	D	x	x	E1	E2	W					
10	lw s5, 4(s1)				F	D	o	EM	M	W					
11	lw s4, 8(s1)				F	D	o	o	EM	M	W				
12	add t6, t6, t4				F	D	x	x	x	EA	W				
13	mult s6, s5, s5					F	D	x	x	E1	E2	W			
14	sw t3, 12(s2)					F	D	x	o	EM	M	W			
15	sw t6, 8(s2)					F	D	x	x	x	EM	M	W		
16	add s6, s6, s4						F	D	x	x	x	EA	W		
17	sw s6, 4(s2)						F	D	x	x	x	x	EM	M	W
18	bne s1, zero, loop						F	D	EA	o	o	o	o	o	W

Figure 183: The execution diagram when the loop is unrolled twice.

iv) The speedup gained with the loop unrolling and the performed optimisations is $\textit{Speedup} = (19 - 14)/19 \approx 0.263$ (26.3% faster execution time)

The main reason for this speedup is that by unrolling the loop we are decreasing the data hazards and we are able to use more functional units in parallel.

[Exercise 16]

Consider a RISC-V processor implemented as a synchronous 4-stage pipeline as in Figure 184, using a single clock signal common to all synchronous elements. The four stages are *Fetch*, *Decode* (which also reads needed operands from the register file), *Execute* (which executes ALU instructions or computes the address for memory instructions), and *Memory* (which accesses memory when needed). The writeback to the register file happens at the end of the *Memory* stage. The processor has no forwarding paths but needs to have implemented appropriate stall signals to respect dependencies in the code.

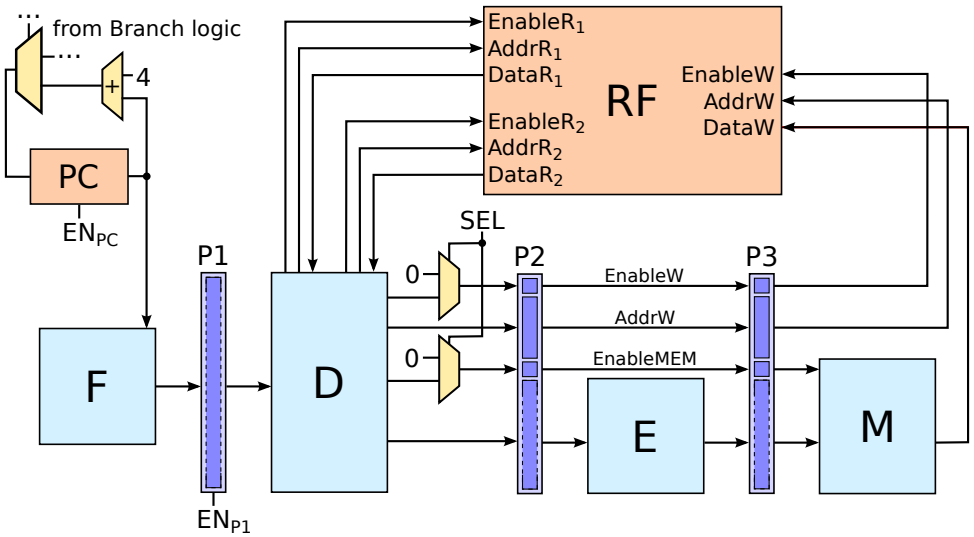


Figure 184: The 4-stage pipeline.

Note that some parts of the pipeline are shown in greater detail. For instance, the *Decode* logic communicates with the Register File through two sets of signals: $\texttt{AddrR}_x$ to indicate the address to read, $\texttt{DataR}_x$ to receive the read data, and $\texttt{EnableR}_x$ which is active to indicate that the corresponding address should indeed be read (that is, that $\texttt{AddrR}_x$ is a required operand for the instruction). Also, the *Decode* logic produces the address of the destination register `AddrW` and the corresponding enable signal `EnableW` (to indicate whether the instruction writes back into the register file), and these are introduced in the pipeline for use in the writeback phase. Similarly, it produces an enable signal `EnableMEM` to indicate if the instruction is a *store* instruction, also inserted in the pipeline for the *Memory* stage. Finally, note some detail for the Program Counter; ignore completely whatever signal comes from the branch logic.

a) Show the execution of the following piece of code in the pipeline, indicating as usual which stages of each instruction are executed in each cycle. Indicate how many cycles

are needed for the whole execution (from the first *Fetch* to the last *Memory* stage used).

```
1  add   x1, x2, x3
2  lw    x2, 0(x1)
3  sub   x3, x1, x4
4  lw    x3, 0(x5)
5  addi  x6, x3, 1
6  addi  x7, x1, -8
```

b) Detail all the logic needed to detect data hazards, stall the pipeline, and to insert bubbles. Specifically, indicate how to connect the enable inputs $\texttt{EN}_{\texttt{PC}}$ and $\texttt{EN}_{\texttt{P1}}$, and the select input `SEL`. Note that the pipeline registers `P2` and `P3` cannot be disabled and that you are not allowed to use signals other than those indicated in the figure. Consider exclusively arithmetic and logic instructions as well as loads and stores (that is, focus on data hazards and disregard completely control hazards). Use comparators, arithmetic operators, and logic gates to implement the requested circuitry.

Consider now two of the above pipelines (A and B) in parallel to implement an in-order superscalar processor. Of course, it will need a register file with four read ports and two write ports.

Since it executes instructions in order, it does not need any Reservation Station nor Reordering Buffer: simply, the new processor fetches two instructions at a time and the instructions advance from *Decode* to *Execute* under the following conditions:

- (a) both of them advance if there is no dependency
- (b) only the first one advances if the second one has a data hazard
- (c) none advance if they both have a data hazard

Note that if the second instruction is stalled in the *Decode* phase due to hazards, both *Fetch* units (or the single, double-width *Fetch* unit) are stalled until the critical instruction can advance (this is because the processor executes instructions in order and the first pipeline cannot ever process in the same cycle and the same stage instructions following the correspondent ones in the second pipeline).

c) Show the execution of the same piece of code above in the new processor. Assume for now that if two instructions need to commit two results to the same register at the

same time, this can be accomplished without delaying the second write. How many cycles are needed now?

d) Show in detail the logic between the registers for the signals $\texttt{AddrW}_\texttt{A}$, $\texttt{EnableW}_\texttt{A}$, $\texttt{AddrW}_\texttt{B}$, and $\texttt{EnableW}_\texttt{B}$ of pipeline registers P3A/P3B and the ports $\texttt{AddrW}_1$, $\texttt{EnableW}_1$ $\texttt{AddrW}_2$, and $\texttt{EnableW}_2$ of the register file. Note that the SRAM of the Register File implements a perfectly synchronous write (writes on the rising edge $\texttt{DataW}_\texttt{x}$ at address $\texttt{AddrW}_\texttt{x}$ if $\texttt{EnableW}_\texttt{x}$ is active) but cannot accept simultaneous writes on the two ports to the same address. The behaviour described in the previous question (without any stall due to a write conflict) must be implemented.

e) Draw the pipelines of the superscalar processor with the same level of detail as that of Figure 184. Describe precisely the modifications to the stall logic (without necessarily drawing it) to generate the signals $\texttt{EN}_\texttt{PC}$, $\texttt{EN}_\texttt{P1A}$, $\texttt{SEL}_\texttt{A}$, $\texttt{EN}_\texttt{P1B}$, and $\texttt{SEL}_\texttt{B}$. Still ignore the existence of control instructions (jumps, branches, etc.) and ignore exceptions.

[Solution 16]

a) Figure 185 shows the code execution in the pipeline. The whole execution requires 13 cycles.

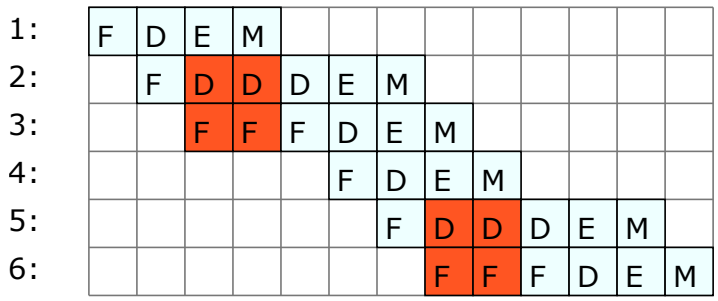


Figure 185: Simplescalar pipeline diagram.

b) See Figure 186.

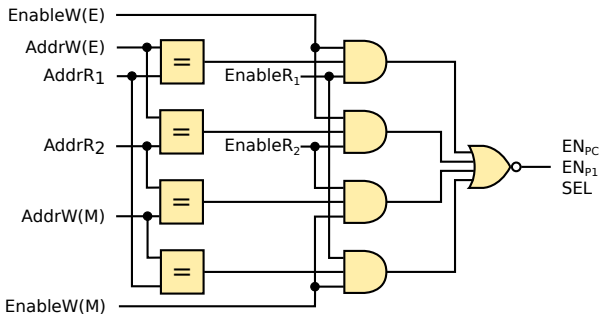


Figure 186: Hazard detection and stalling logic.

c) Figure 187 shows the execution of the code on the superscalar. It takes 11 cycles to finish.

d) See Figure 188.

e) See Figure 189.

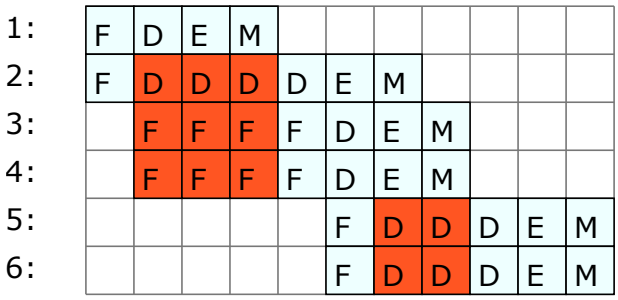


Figure 187: Superscalar pipeline diagram.

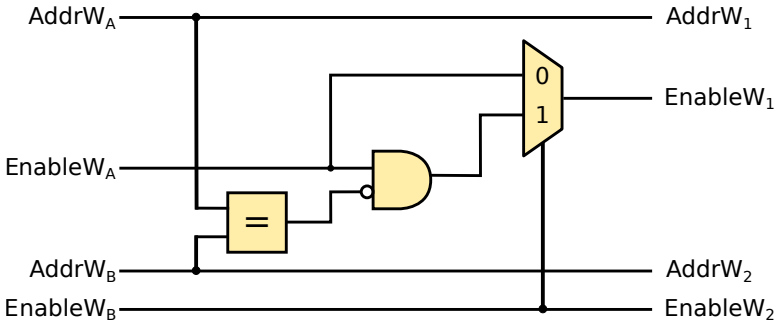


Figure 188: Logic between the registers.

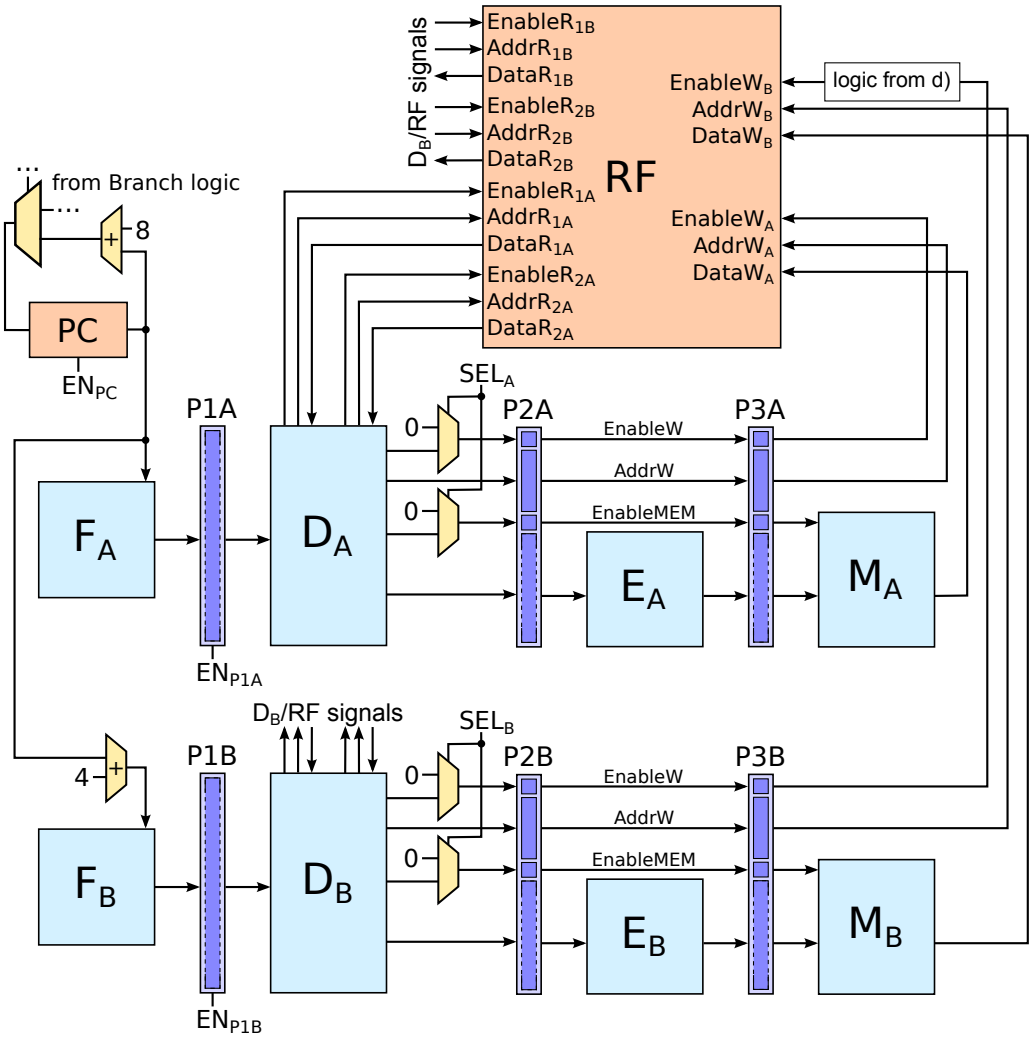


Figure 189: Pipelines of the superscalar processor.

[Exercise 17]

a) Consider a processor which executes integer instructions and is implemented as given in Figure 190.

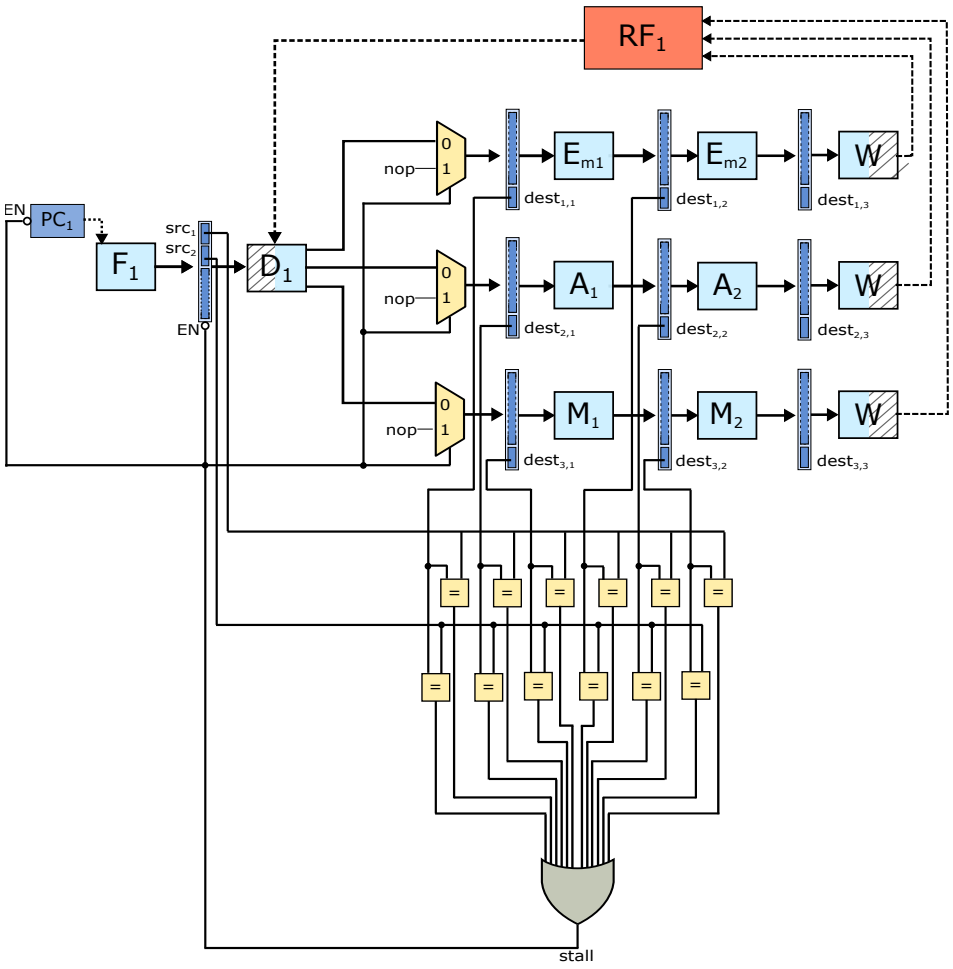


Figure 190: A processor with three pipelines for executing integer instructions.

The processor can fetch and decode **one instruction per cycle**. Once the instruction is decoded, it is dispatched to one of the three pipelines, depending on the instruction type:

- **Integer multiplication** is executed in pipeline 1, in two Execute stages (E_{m1}, E_{m2}).
- Other **integer arithmetic operations** are executed in pipeline 2, in stages A_1 an A_2.
- **Memory access operations** are executed in pipeline 3, in stages (M_1) and (M_2).

- For all pipelines, the corresponding Writeback stage writes to the registers in the first half of the cycle, and the Decode stage reads the register file in the second half of the cycle (i.e., the processor can forward data from all Writeback stages to the Decode stage).
- The processor has **no other forwarding paths**.

The processor has appropriate stall signals to respect data dependencies in the code, as detailed in Figure 190.

In all subsequent questions, assume that there is a perfect cache that always hits. You can also assume that a new instruction **`mul`** has been added to the RISC-V ISA. This instruction is defined as: **`mul`** `rA, rB, rC`, corresponding to the pseudocode: `rA = rB * rC`.

Consider the following program:

```
01    addi    x1, x1, 8
02    add     x2, x1, x1
03    slli    x1, x1, 1
04    sw      x2, 0(x1)
05    lw      x4, 0(x6)
06    mul     x2, x3, x4
07    mul     x5, x5, x5
08    srli    x4, x2, 4
09    sw      x5, 0(x1)
```

A Show the **execution of this program** on the processor detailed in Figure 190. Mark all stalls with `X` and mark when any of the Writeback to Decode forwarding paths is used. Use the provided template on the last page of the exercise.

B Calculate the **achieved IPC** and compare it to the **ideal IPC** for this processor.

C Are the Writeback stage of each instruction always performed **in order**? If not, why is this not a problem? Justify your answer concisely (1-2 sentences).

D Can **structural hazards** occur at Writeback, i.e., is it possible that multiple pipelines need to write back to the register file at the same time? If yes, what is the minimal number of ports the register file needs to have to resolve the hazard and guarantee correct functionality? Justify your answer concisely (1-2 sentences).

b) Consider now a **2-thread processor**, given in Figure 191. Each thread has its own Fetch and Decode stage (the threads are **completely independent**). The subsequent

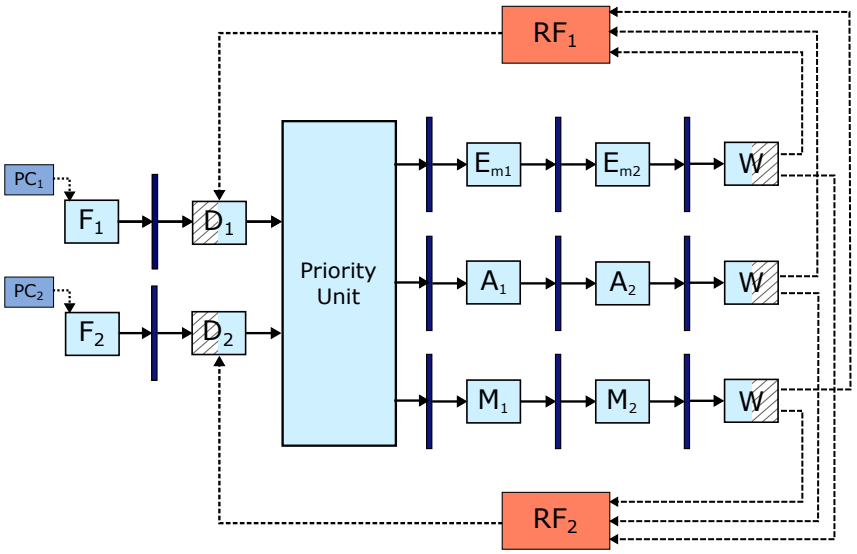


Figure 191: 2-thread processor with three pipelines.

pipelines and their stages are identical to the ones described in Part 1, with the forwarding paths only from the Writeback to the Decode stages.

Instructions can be issued from the threads simultaneously: Each fetched instruction is given to the corresponding Decode stage if it is free. **If the instructions of the two threads require different pipelines**, both instructions are decoded and dispatched to the corresponding Execute stages in parallel.

However, if, on a given decode cycle, **both instructions need to be dispatched to the same pipeline**, only one of them can be decoded and can proceed, and the other one needs to be stalled. This conflict is resolved by the **priority unit**, based on the following principle:

- On **odd clock cycles, Thread 1 has priority** over Thread 2; on **even clock cycles, Thread 2 has priority** over Thread 1.
- Unless the **thread with the higher priority needs to be stalled** due to a data hazard, its instruction is decoded and sent to the appropriate Execute stage.
- If the thread with the higher priority needs to be stalled due to a data hazard, **the thread with the lower priority is allowed to proceed instead** (unless it is also stalled by the hazard detection logic).

The execution of the processor is demonstrated in Figure 192.

In all subsequent questions, assume that there is a perfect cache that always hits.

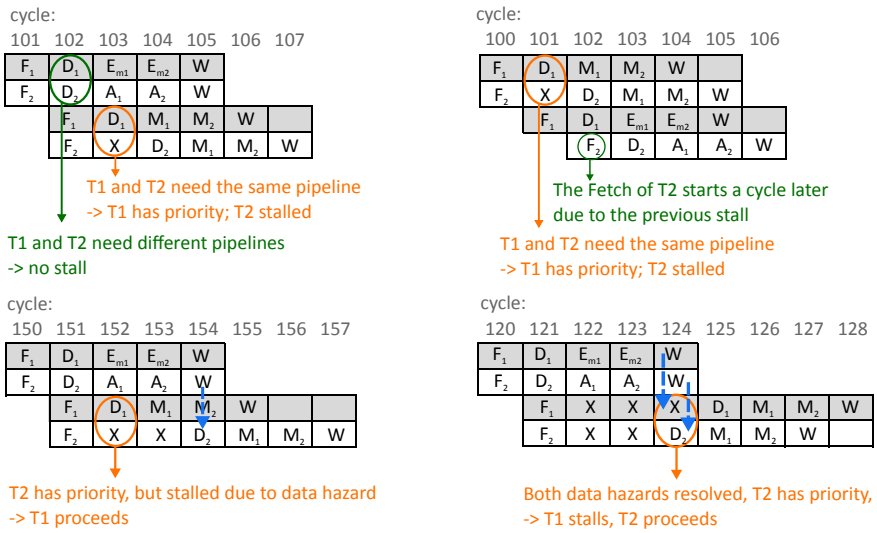


Figure 192: Execution examples for the 2-thread processor.

Consider the following programs:

- Program 1:

```
01    addi    x1, x1, 8
02    add     x2, x1, x1
03    slli    x1, x1, 1
04    sw      x2, 0(x1)
05    lw      x4, 0(x6)
06    mul     x2, x3, x4
07    mul     x5, x5, x5
08    srli    x4, x2, 4
09    sw      x5, 0(x1)
```

- Program 2:

```
01    lw      x5, 0(x7)
02    addi    x4, x5, 8
03    mul     x5, x5, x5
04    sw      x5, 0(x7)
05    addi    x1, x1, 4
06    mul     x3, x3, x1
07    add     x3, x5, x5
08    slli    x3, x3, 2
09    sw      x3, 0(x7)
```

The instructions of program 1 are fetched by Thread 1, and the instructions of program

2 are fetched by Thread 2 of the processor. The processor starts by fetching the first instruction of program 1 and the first instruction of program 2 in cycle 1.

A Show the **execution of this program** on the processor detailed in Figure 191. Mark all stalls with `X` and mark the used forwarding paths. Additionally, mark the cycles where the Decode stages of the two threads are conflicting (encircle the D and corresponding X stages as shown in the examples in Figure 192). Use the provided template in the last page of the exercise.

B Calculate the **achieved IPC** and compare it to the **ideal IPC** for this processor.

C Are the Writeback stage of the instructions of **each** thread always performed **in order**? If not, why is this not a problem?

D What is the relationship of the writebacks of **one thread with respect to the other**? Justify your answer concisely (1-2 sentences).

E Can the implemented priority policy lead to **starvation** of one of the threads, i.e., is it possible that one of the threads is continuously denied access to the pipelines? Justify your answer concisely (1-2 sentences).

F Can **structural hazards** occur at Writeback, i.e., is it possible that multiple pipelines need to write back to a register file at the same time? If yes, what is the minimal number of ports each register file needs to have to resolve the hazard and to guarantee correct functionality? Justify your answer concisely (1-2 sentences).

c) Consider the implementation of the **stall logic** of the 2-thread processor in Figure 191.

A Does the hazard detection logic of the 2-thread processor differ from the logic of the processor in Figure 190? If yes, briefly describe how the stall signals in the 2-thread processor needs to be generated.

B In Figure 190, the stall signal created by the hazard detection logic (marked **stall** in the figure) is used as the enable signal (**EN**) for the Decode pipeline register and PC. Does the same hold for the 2-thread processor in Figure 191? If no, briefly describe the difference between the two implementations.

Exercise 1

				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	t1:	addi	x1, x1, 8																											
2	t1:	add	x2, x1, x1																											
3	t1:	slli	x1, x1, 1																											
4	t1:	sw	x2, 0(x1)																											
5	t1:	lw	x4, 0(x6)																											
6	t1:	mul	x2, x3, x4																											
7	t1:	mul	x5, x5, x5																											
8	t1:	srli	x4, x2, 4																											
9	t1:	sw	x5, 0(x1)																											

Exercise 2

				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	t1:	addi	x1, x1, 8																											
2	t2:	lw	x5, 0(x7)																											
3	t1:	add	x2, x1, x1																											
4	t2:	addi	x4, x5, 8																											
5	t1:	slli	x1, x1, 1																											
6	t2:	mul	x5, x5, x5																											
7	t1:	sw	x2, 0(x1)																											
8	t2:	sw	x5, 0(x7)																											
9	t1:	lw	x4, 0(x6)																											
10	t2:	addi	x1, x1, 4																											
11	t1:	mul	x2, x3, x4																											
12	t2:	mul	x3, x3, x1																											
13	t1:	mul	x5, x5, x5																											
14	t2:	add	x3, x5, x5																											
15	t1:	srli	x4, x2, 4																											
16	t2:	slli	x3, x3, 2																											
17	t1:	sw	x5, 0(x1)																											
18	t2:	sw	x3, 0(x7)																											

[Solution 17]

a)

A The execution of the program is given in the filled template on the next page.

B The achieved IPC is $9/20 = 0.45$. The ideal IPC is 1.

C As all three pipelines have the same length, the Writeback stages are always performed in order.

D As only one instruction per cycle is issued, and the pipelines are of equal length, only one instruction will be in the Writeback stage in a given cycle and structural hazards cannot occur.

b)

A The execution of the program is given in the filled template on the next page.

B The achieved IPC is $18/24 = 0.75$. The ideal IPC is 2.

C As each thread decodes at most one instruction per cycle, and all pipelines are of equal length, the Writeback stages of each thread are always performed in order.

D The Writebacks of the two threads are completely independent since the threads use different register files (data hazards between the threads cannot occur).

E Starvation cannot occur, since the priority of the threads changes in each cycle—if a thread is stalled in a given cycle, it will certainly have priority and be able to execute in the next.

F In a given cycle, there can be at most two instructions from two different threads in the Writeback stage—since they will be written into different register files, structural hazards cannot occur.

Exercise 1

				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	t1:	addi	x1, x1, 8	F	D	A1	A2	W																						
2	t1:	add	x2, x1, x1		F	X	X	D	A1	A2	W																			
3	t1:	slli	x1, x1, 1					F	D	A1	A2	W																		
4	t1:	sw	x2, 0(x1)						F	X	X	D	M1	M2	W															
5	t1:	lw	x4, 0(x6)									F	D	M1	M2	W														
6	t1:	mul	x2, x3, x4										F	X	X	D	Em1	Em1	W											
7	t1:	mul	x5, x5, x5													F	D	Em1	Em2	W										
8	t1:	srli	x4, x2, 4														F	X	D	A1	A2	W								
9	t1:	sw	x5, 0(x1)																F	D	M1	M2	W							

Exercise 2

				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	t1:	addi	x1, x1, 8	F1	D1	A1	A2	W																						
2	t2:	lw	x5, 0(x7)	F2	D2	M1	M2	W																						
3	t1:	add	x2, x1, x1		F1	X	X	D1	A1	A2	W																			
4	t2:	addi	x4, x5, 8		F2	X	X	X	D2	A1	A2	W																		
5	t1:	slli	x1, x1, 1					F1	X	D1	A1	A2	W																	
6	t2:	mul	x5, x5, x5						F2	D2	Em1	Em2	W																	
7	t1:	sw	x2, 0(x1)							F1	X	X	X	D1	M1	M2	W													
8	t2:	sw	x5, 0(x7)							F2	X	X	D2	M1	M2	W														
9	t1:	lw	x4, 0(x6)											F1	D1	M1	M2	W												
10	t2:	addi	x1, x1, 4										F2	D2	A1	A2	W													
11	t1:	mul	x2, x3, x4												F1	X	X	D1	Em1	Em2	W									
12	t2:	mul	x3, x3, x1											F2	X	X	D2	Em1	Em2	W										
13	t1:	mul	x5, x5, x5															F1	D1	Em1	Em2	W								
14	t2:	add	x3, x5, x5														F2	D2	A1	A2	W									
15	t1:	srli	x4, x2, 4																F1	X	X	D1	A1	A2	W					
16	t2:	slli	x3, x3, 2															F2	X	X	D2	A1	A2	W						
17	t1:	sw	x5, 0(x1)																			F1	D1	M1	M2	W				
18	t2:	sw	x3, 0(x7)																		F2	X	X	D2	M1	M2	W			

c)

A A hazard can occur only between instructions belonging to the same thread, so the processor requires separate hazard detection logic for thread 1 and thread 2. Each instruction needs to be tagged with the ID of its corresponding thread, so the hazard detection logic can determine the origin of the instruction and generate each stall signal accordingly.

B In this processor, the priority unit determines which instruction(s) can be decoded based on the data hazards and the cycle priority. Hence, the stall signals from the hazard detection logic are not connected directly to the registers of the decode stages—the priority unit produces the enable signals.

[Exercise 18] Dynamically Scheduled Processor

Consider a dynamically scheduled processor, implemented as given in Figure 193. The processor can fetch and decode **one instruction per cycle**. Once the instruction is decoded, it is dispatched to one of the functional units, depending on the instruction type:

- **Floating point addition and multiplication** (`fmul`, `fadd`) are executed in the floating point unit, in three execution stages (X_1, X_2, X_3).
- All other **arithmetic and logic instructions** are executed in the Arithmetic Logic Unit (ALU), in stages A_1 and A_2.
- **Memory access operations** are executed in the Load/Store unit, in stages M_1, M_2, and M_3.

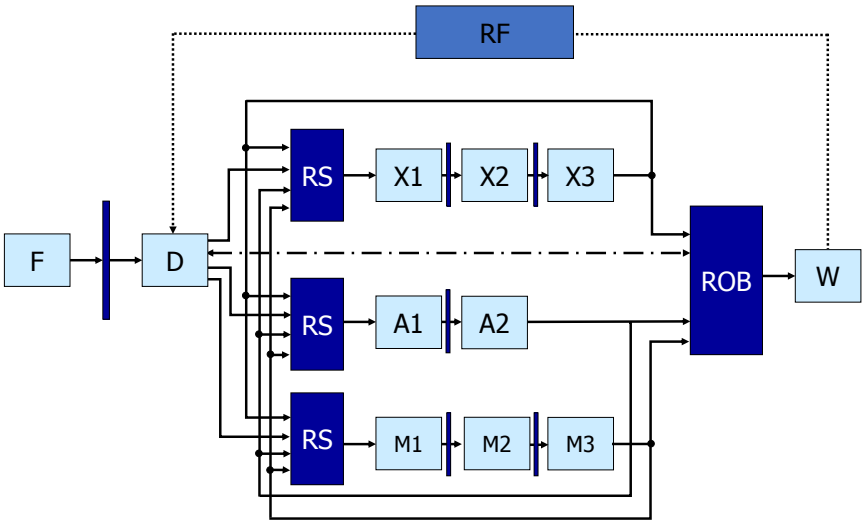


Figure 193: The dynamically scheduled, out-of-order processor.

Each functional unit has a reservation station (RS), which operates as follows:

- Results of the Decode stage are loaded into the relevant RS at the end of the decoding cycle, together with the ready operands.
- In case some value is not available, the tag of the corresponding operand indicates the instruction that produces the value. Once the operand becomes available and the value is written into the RS, the tag is removed.

- The state indicates whether RS entry is (i) invalid (i.e., empty), or the instruction is either (ii) executing or (iii) waiting to execute.
- The result of each functional unit is available at the end of the last cycle of execution. At the end of the cycle, all RS entries are updated (so that the result can be used on the following cycle) and the entry corresponding to the executed instruction is deallocated from the RS (i.e., its state is set as invalid).

The processor has a reorder buffer (ROB) for preserving in-order instruction commits and can commit a single instruction per cycle. The ROB operates as follows:

- Results of the Decode stage are loaded into the ROB at the end of the decoding cycle, to prepare the placeholder for the result (by updating the fields corresponding to the PC, the tag, and either the target register or address).
- The results from all functional units are written into the ROB at the end of the last cycle of execution. The instruction tag is removed at the same time.
- The entry is removed from the ROB at the end of the Writeback of the corresponding instruction.

In all subsequent questions, assume that there is a perfect cache that always hits. All reservation stations and the reorder buffer are of unlimited size. The processor executes RISC-V instructions.

Consider the following program:

```
01    lw      x2, 0(x1)
02    fmul    x3, x7, x2
03    fadd    x4, x4, x4
04    addi    x1, x1, -1
05    lw      x2, 0(x1)
06    lw      x5, 0(x6)
07    fmul    x7, x7, x2
08    addi    x5, x5, 1
09    lw      x2, 0(x5)
10    sub     x1, x2, x1
11    lw      x3, 0(x5)
12    lw      x4, 0(x6)
13    fmul    x3, x3, x4
14    addi    x6, x6, 4
```

a) Explain concisely (1-2 sentences) the mechanism that the processor may use to determine whether an instruction depends on any of the preceding instructions. What is the role of the ROB in this process? What are the possible situations?

b) In the provided template, show the complete **execution of this program** on the processor from Figure 193. Mark with x the stalls resulting from data hazards and with o the stalls resulting from structural hazards including the case of instructions not ready for a commit.

c) In the provided template, indicate the **states of each reservation station and the reorder buffer** during the following points of program execution:

i) The beginning of the cycle following the one when instruction 2 is committed (STATE1).

ii) The beginning of the cycle following the one when instruction 7 is starting to execute (STATE2).

iii) The beginning of the cycle following the one when instruction 8 is finishing execution (STATE3).

Use the following notation to fill out the template:

i) The tag corresponds to the name of the RS entry where the corresponding instruction is placed; it is composed by the name of the RS (ALU, MEM, FLOAT) and the ordinal number of this entry in the RS. RS entry names are indicated on the left of the RS entries in the provided template.

ii) In the State field, denote the state of the instruction as W (waiting) or E (executing). Invalid entries do not need to be shown.

iii) In the ROB, the top (first) line of the table always represents the Head (i.e., always fill the ROB starting from the top line).

iv) For the PC values, use the instruction numbers given in the code above.

v) In the Arg and Value fields, simply indicate whether the corresponding value is available (A) or missing (M) at the observed points in time. You are not required to calculate any actual values.

d) What is the **achieved CPI** for this execution?

e) Would replicating any of the three functional units result in a lower CPI in the case of this program? If so, which unit(s) would this program benefit from replicating? Elaborate your answer concisely (1-2 sentences) and justify your answer with an example from the execution.

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	lw	x2, 0(x1)																														
2	fmul	x3, x7, x2																														
3	fadd	x4, x4, x4																														
4	addi	x1, x1, -1																														
5	lw	x2, 0(x1)																														
6	lw	x5, 0(x6)																														
7	fmul	x7, x7, x2																														
8	addi	x5, x5, 1																														
9	lw	x2, 0(x5)																														
10	sub	x1, x2, x1																														
11	lw	x3, 0(x5)																														
12	lw	x4, 0(x6)																														
13	fmul	x3, x3, x4																														
14	addi	x6, x6, 4																														

STATE 1:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2						
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1						
MEM2						
MEM3						
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1						
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value

STATE 2:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2						
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1						
MEM2						
MEM3						
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1						
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value

STATE 3:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2						
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1						
MEM2						
MEM3						
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1						
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value

[Solution 18]

a) The ROB plays a very important role in the decoding process. It allows the decoder to know if there are any operations scheduled to modify the location it wants to access. To check the decode simply looks to see if there are any writes to the memory it wants to access, starting from the tail of the ROB. If it finds that there is a pending operation that will modify the value D will get the information that it needs from the ROB.

- **Case 1**: ROB has value but has not committed it to memory or RF. Decode takes value from the ROB.
- **Case 2**: ROB doesn't have the value but has the register or address of interest. Copy of the tag that is in the ROB.

Instructions are added to the tail of the ROB so it will always get the most updated value. If nothing is found in the ROB then there are no dependencies.

(g)

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	lw	x2, 0(x1)	F	D	M1	M2	M3	W																								
2	fmul	x3, x7, x2		F	D	X	X	X1	X2	X3	W																					
3	fadd	x4, x4, x4			F	D	X1	X2	X3	O	O	W																				
4	addi	x1, x1, -1				F	D	A1	A2	O	O	O	W																			
5	lw	x2, 0(x1)					F	D	X	M1	M2	M3	O	W																		
6	lw	x5, 0(x6)						F	D	O	M1	M2	M3	O	W																	
7	fmul	x7, x7, x2							F	D	X	X	X1	X2	X3	W																
8	addi	x5, x5, 1								F	D	X	X	A1	A2	O	W															
9	lw	x2, 0(x5)									F	D	X	X	X	M1	M2	M3	W													
10	sub	x1, x2, x1										F	D	X	X	X	X	X	A1	A2	W											
11	lw	x3, 0(x5)											F	D	X	O	M1	M2	M3	O	O	W										
12	lw	x4, 0(x6)												F	D	O	O	M1	M2	M3	O	O	W									
13	fmul	x3, x3, x4													F	D	X	X	X	X	X1	X2	X3	W								
14	addi	x6, x6, 4														F	D	A1	A2	O	O	O	O	O	W							

STATE 1:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1	w	ADDI	MEM2	-	M	A
ALU2						
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1	E	LW	-	-	A	-
MEM2	E	LW	-	-	A	-
MEM3						
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1	W	FMUL	-	MEM1	A	M
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value
3	-	X4	-	A
4	-	X1	-	A
5	MEM1	X2	-	M
6	MEM2	X5	-	M
7	FLOAT1	X7	-	M
8	ALU1	X5	-	M

STATE 2:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1	W	ADDI	-	-	A	A
ALU2	W	SUB	MEM1	-	M	A
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1	W	LW	ALU1	-	M	-
MEM2						
MEM3						
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1	E	FMUL	-	-	A	A
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value
5	-	X2	-	A
6	-	X5	-	A
7	FLOAT1	X7	-	M
8	ALU1	X5	-	M
9	MEM1	X2	-	M
10	ALU2	X1	-	M

STATE 3:

ALU RS

	State	Op	Tag1	Tag2	Arg1	Arg2
ALU1						
ALU2	W	SUB	MEM1	-	M	A
ALU3						
ALU4						
ALU5						
ALU6						
ALU7						
ALU8						

MEM RS

	State	Op	Tag1	Tag2	Arg1	Arg2
MEM1	W	LW	-	-	A	-
MEM2	W	LW	-	-	A	-
MEM3	W	LW	-	-	A	-
MEM4						
MEM5						
MEM6						
MEM7						
MEM8						

FLOAT RS

	State	Op	Tag1	Tag2	Arg1	Arg2
FLOAT1						
FLOAT2						
FLOAT3						
FLOAT4						
FLOAT5						
FLOAT6						
FLOAT7						
FLOAT8						

ROB

PC	Tag	Register	Address	Value
7	-	X7	-	A
8	-	X5	-	A
9	MEM1	X2	-	M
10	ALU2	X1	-	M
11	MEM2	X3	-	M
12	MEM3	X4	-	M

d)

$$\text{CPI} = \frac{23}{14}$$

e) Given the execution diagram of part B, adding a Load/Store unit would make the execution faster. For example, starting and finishing instruction 6 one cycle faster would allow instructions 8 and 9 to finish one cycle earlier, gaining one cycle overall.

[Exercise 19] Comparing Processor Pipelines

Consider two processor pipelines, shown in Figures 194 and 195.

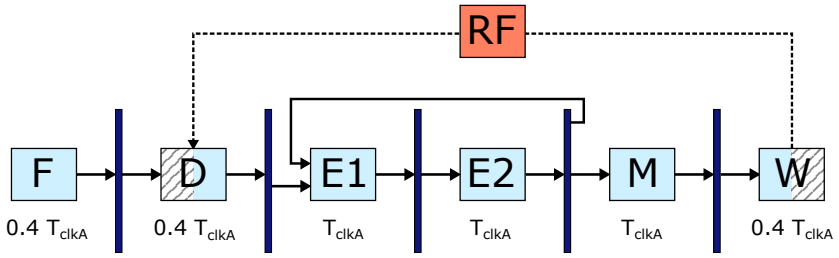


Figure 194: Pipeline A.

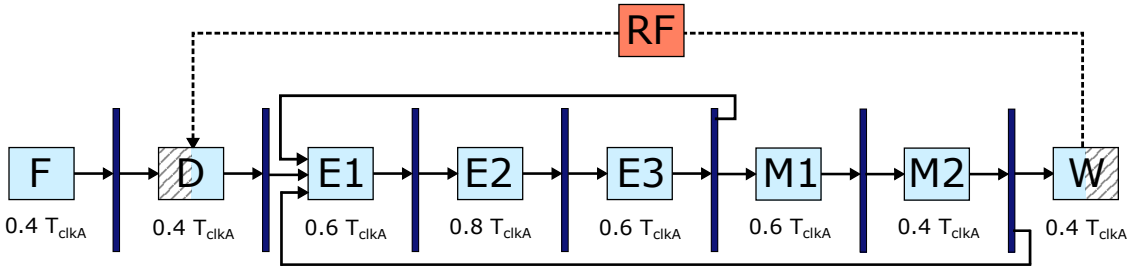


Figure 195: Pipeline B.

```
01    addi    x0, x0, 4
02    mul     x1, x1, x0
03    addi    x2, x2, -1
04    add     x1, x1, x1
05    addi    x6, x6, 1
06    lw      x5, 0(x3)
07    mul     x5, x5, x3
08    div     x4, x4, x7
09    add     x4, x0, x4
10    sub     x6, x0, x5
```

Listing 4.1

The pipelines have the following properties:

- Arithmetic and logic operations are completed in the final Execute stage (i.e., stage E2 in Pipeline A and stage E3 in Pipeline B) and can be forwarded using any given forwarding path.
- The results of the memory operations are only available at the end of the last Memory stage of each pipeline.
- In both pipelines, the Writeback stage writes to registers in the first half of the cycle and the Decode stage can read the register file in the second half of the cycle.
- Pipeline A operates at a frequency of 100 MHz. Pipeline B is obtained by inserting additional pipeline stages into Pipeline A in order to increase operation frequency. The latency of each stage of both pipelines is given in the figures as a function of the clock period of Pipeline A, T_{clkA}. The clock frequency of each pipeline is the maximum frequency which guarantees correct operation.

a) Consider Program 4.1.

A **In the provided templates**, show the complete **execution of this program** on Pipeline A and on Pipeline B. Mark with $\times$ the stalls resulting from data hazards and indicate which forwarding paths are used.

B For **both executions** from the previous question, compute the achieved CPI and the **total execution time** in ns. Keep in mind that the pipelines operate at **different frequencies**.

After a first generation of the code, compilers often try to **reorder instructions** to make the best possible use of the pipeline, i.e., to eliminate as many stalls as possible. Here is a possible way to compute a good instruction reordering.

Firstly, **a directed graph** of all RAW, WAW, and WAR dependencies is constructed as follows:

- Nodes represent instructions and directed edges represent dependencies (pointing to the instruction which needs to be executed last among the two). All **self-dependencies are ignored**.
- Edges corresponding to **WAW and WAR dependencies** are annotated with weight 0.
- Edges corresponding to **RAW dependencies** are annotated with a weight corresponding to the minimum number of cycles necessary in the target microarchitecture to use the result of the instruction at the source of the edge. For instance,

any edge out of the node of an `add` instruction of Pipeline B is annotated with a weight equal to 3 because the earliest that the result of an addition can be used for another operation is three cycles after its decode stage (through the forwarding path E3 $\rightarrow$ E1).

- If two instructions have multiple dependencies among them, they are connected with a **single edge**. The edge is annotated with the largest of the related weights. For instance, if there is a RAW and a WAW dependency between two instructions, the corresponding edge will have the weight of the RAW dependency (as the weight of the WAW dependency is always 0).

Figure 196 shows the dependency graph of Program 4.2 for Pipeline B.

```
01    addi    x3, x2, x1
02    mul     x4, x3, x1
03    sub     x2, x5, x5
04    div     x6, x4, x7
```

Listing 4.2

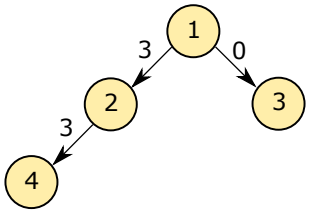


Figure 196: Dependency graph of Program 4.2 for Pipeline B.

Secondly, one creates the **optimized program** by building progressively the execution schedule and deciding which is the next instruction to add to the program. Initially, the schedule and the program are empty. One proceeds as follows:

- All instructions without predecessors (i.e., with no incoming edges in the dependency graph) are considered.
- One tries to place each of these instructions as the next one in the program and checks whether it would fit the schedule without stalls. Only the instructions that would not incur a stall are still considered.
- For each of those instructions, one looks at the **longest weighted path** in the directed graph to arrive to a node without successors (i.e., a node with no outgoing edges). For instance, instruction 1 in Figure 3 has a longest weighted path of 6 to instruction 4.
- The instruction with the longest weighted path among those considered is added to the program, the corresponding node is deleted from the dependency graph, and its execution is simulated in the schedule. If there is a tie, the instruction that comes **earlier in program order** (in the original program) is chosen. If there is no instruction that would not incur stalls, the instruction that comes earliest in the original program among those without predecessors is chosen.
- The process is repeated until all instructions are in the optimized program.

If one applies this to Program 4.2 with the dependency graph of Figure 196, one starts the program with instruction 1 (the only instruction without predecessors), then one selects instruction 3 (because both 2 and 3 have no predecessor but instruction 2 would cause two stalls), then instruction 2 (the only instruction without predecessors), and then instruction 4 (because it is the only remaining instruction). In this extremely simple case, the lengths of the paths never play a role.

b) Apply the above procedure to Program 4.1, targeting Pipeline A and Pipeline B:

A For each pipeline, show the annotated dependency graph before starting the reordering process. Reorder the instructions and show the resulting program for each pipeline.

B **In the provided templates**, show the complete **execution of the restructured program** for Pipeline A and for Pipeline B. Mark with $\times$ the stalls resulting from data hazards and indicate which forwarding paths are used.

C For **both executions**, compute the achieved CPI and the **total execution time** (in ns).

c) Concisely answer the following questions (1–2 sentences per answer).

A Does the compiler optimization from the previous question need to be aware of the underlying processor microarchitecture? Justify your answer.

B The above algorithm does not guarantee an optimal result. Are the results obtained in the previous question optimal? Answer clearly with yes or no and elaborate your answer.

C What is the effect of inserting additional pipeline stages on overall pipeline performance? Discuss the tradeoff between pipeline latency (i.e., number of pipeline stages) and achievable frequency.

a) Pipeline A

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	addi x0, x0, 4																														
2	mul x1, x1, x0																														
3	addi x2, x2, -1																														
4	add x1, x1, x1																														
5	addi x6, x6, 1																														
6	lw x5, 0(x3)																														
7	mul x5, x5, x3																														
8	div x4, x4, x7																														
9	add x4, x0, x4																														
10	sub x6, x0, x5																														

a) Pipeline B

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	addi x0, x0, 4																														
2	mul x1, x1, x0																														
3	addi x2, x2, -1																														
4	add x1, x1, x1																														
5	addi x6, x6, 1																														
6	lw x5, 0(x3)																														
7	mul x5, x5, x3																														
8	div x4, x4, x7																														
9	add x4, x0, x4																														
10	sub x6, x0, x5																														

b) Pipeline A, restructured code

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

b) Pipeline B, restructured code

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

[Solution 19]

a)

A The executions of the program are given in the filled templates below.

B The achieved CPI for the executions from the previous question are $CPI = 20/10 = 2$ for Pipeline A and $CPI = 26/10 = 2.6$ for Pipeline B.

With $T_{clkA} = 10$ ns, the execution time of Pipeline A is $20 \cdot T_{clkA} = 200$ ns and the execution time of Pipeline B is $26 \cdot 0.8 \cdot T_{clkA} = 208$ ns.

b)

A The dependency graphs are given in the figure below.

B The executions of the program are given in the filled templates below.

C The achieved CPI for the executions from the previous question are $CPI = 15/10 = 1.5$ for Pipeline A and $CPI = 18/10 = 1.8$ for Pipeline B.

With $T_{clkA} = 10$ ns, the execution time of Pipeline A is $15 \cdot T_{clkA} = 150$ ns and the execution time of Pipeline B is $18 \cdot 0.8 \cdot T_{clkA} = 144$ ns.

c)

A The compiler must be aware of the architecture to appropriately compute the weights of the edges between instructions.

B The result is optimal for Pipeline A. Pipeline B incurs a stall and the result is not optimal.

C Inserting pipeline stages increases frequency but also increases pipeline latency. In our exercise, Pipeline B has more stages and a higher frequency (125 MHz) than Pipeline A (100 MHz), but also higher latency—in the first question, the latency difference is dominant and Pipeline A outperforms Pipeline B; in the second case, due to higher frequency, Pipeline B outperforms Pipeline A.

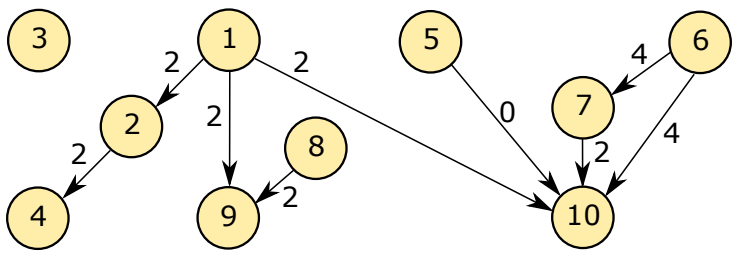


Figure 197: Dependency graph for Pipeline A.

a) Pipeline A

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	addi x0, x0, 4	F	D	E1	E2	M	W																								
2	mul x1, x1, x0		F	X	D	E1	E2	M	W																						
3	addi x2, x2, -1				F	D	E1	E2	M	W																					
4	add x1, x1, x1					F	D	E1	E2	M	W																				
5	addi x6, x6, 1						F	D	E1	E2	M	W																			
6	lw x5, 0(x3)							F	D	E1	E2	M	W																		
7	mul x5, x5, x3								F	X	X	X	D	E1	E2	M	W														
8	div x4, x4, x7												F	D	E1	E2	M	W													
9	add x4, x0, x4													F	X	D	E1	E2	M	W											
10	sub x6, x0, x5															F	D	E1	E2	M	W										

a) Pipeline B

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	addi x0, x0, 4	F	D	E1	E2	E3	M1	M2	W																						
2	mul x1, x1, x0		F	X	X	D	E1	E2	E3	M1	M2	W																			
3	addi x2, x2, -1					F	D	E1	E2	E3	M1	M2	W																		
4	add x1, x1, x1						F	X	D	E1	E2	E3	M1	M2	W																
5	addi x6, x6, 1								F	D	E1	E2	E3	M1	M2	W															
6	lw x5, 0(x3)									F	D	E1	E2	E3	M1	M2	W														
7	mul x5, x5, x3										F	X	X	X	X	D	E1	E2	E3	M1	M2	W									
8	div x4, x4, x7															F	D	E1	E2	E3	M1	M2	W								
9	add x4, x0, x4																F	X	X	D	E1	E2	E3	M1	M2	W					
10	sub x6, x0, x5																			F	D	E1	E2	E3	M1	M2	W				

b) Pipeline A, restructured code

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
6	lw x5, 0(x3)	F	D	E1	E2	M	W																								
1	addi x0, x0, 4		F	D	E1	E2	M	W																							
8	div x4, x4, x7			F	D	E1	E2	M	W																						
2	mul x1, x1, x0				F	D	E1	E2	M	W																					
7	mul x5, x5, x3					F	D	E1	E2	M	W																				
3	addi x2, x2, -1						F	D	E1	E2	M	W																			
5	addi x6, x6, 1							F	D	E1	E2	M	W																		
4	add x1, x1, x1								F	D	E1	E2	M	W																	
9	add x4, x0, x4									F	D	E1	E2	M	W																
10	sub x6, x0, x5										F	D	E1	E2	M	W															

b) Pipeline B, restructured code

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
6	lw x5, 0(x3)	F	D	E1	E2	E3	M1	M2	W																						
1	addi x0, x0, 4		F	D	E1	E2	E3	M1	M2	W																					
8	div x4, x4, x7			F	D	E1	E2	E3	M1	M2	W																				
3	addi x2, x2, -1				F	D	E1	E2	E3	M1	M2	W																			
2	mul x1, x1, x0					F	D	E1	E2	E3	M1	M2	W																		
7	mul x5, x5, x3						F	D	E1	E2	E3	M1	M2	W																	
5	addi x6, x6, 1							F	D	E1	E2	E3	M1	M2	W																
4	add x1, x1, x1								F	D	E1	E2	E3	M1	M2	W															
9	add x4, x0, x4									F	D	E1	E2	E3	M1	M2	W														
10	sub x6, x0, x5										F	X	D	E1	E2	E3	M1	M2	W												

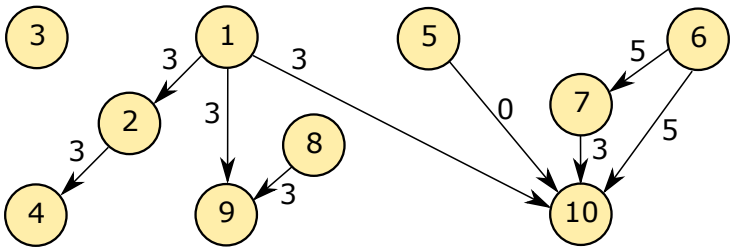


Figure 198: Dependency graph for Pipeline B.

[Exercise 20]

Consider a simple **pipelined** processor whose microarchitecture is shown in Figure 199. The pipeline is made of a register file (**RF**) and 5 stages: *Fetch* (**F**), *Decode* (**D**), *Execute* (**E**), *Memory* (**M**), and *Writeback* (**W**). The pipeline has no forwarding paths. The result of all arithmetic and branch operations is available after the **E** stage. The result of all memory operations is available after the **M** stage. The pipeline detects data hazards automatically and stalls instructions whose operands are not ready in the **D** stage.

The processor handles control hazards by initially assuming that every branch is not taken; after fetching a branch, the processor continues fetching and decoding instructions in program order until the outcome of the branch is known at the end of the branch instruction's **E** stage. At this point, if the real branch outcome is not taken, then the processor fetched the correct instructions and can continue pushing them through the pipeline. However, if the real branch outcome is taken, then the processor squashes the incorrectly fetched instructions by discarding them from the pipeline and starts fetching from the branch target address. You do not need to know how the squashing process is implemented in hardware, as such it is not represented in Figure 199.

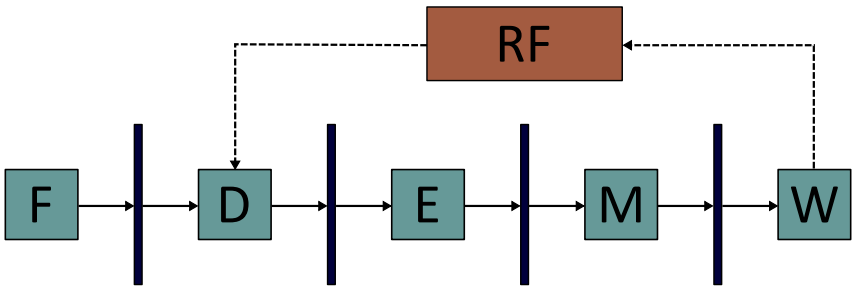


Figure 199: Simple pipelined processor architecture.

Consider the following RISC-V program. The `sum` function adds up all words located in a memory region specified by the two function arguments `a0` and `a1`, and updates the value stored at the memory location specified in `a2` by adding the computed sum to it. `a0` is the starting address of the memory region, and `a1` is its size (in number of 4-byte words).

```
sum:
  add t0, zero, zero     # initialize the accumulator
  loop:
    lw  t1, 0(a0)        # load value from memory
    add  t0, t0, t1      # accumulate the value
    addi a1, a1, -1      # decrement word count
    addi a0, a0, 4       # increment memory address
    bne  a1, zero, loop # end of the region?
  end:
```

```
    lw  t1, a2(zero)     # load value from destination address
    add  t1, t1, t0      # add computed sum to value
    sw  t1, a2(zero)     # store back on destination address
    addi v0, zero, t0    # save t0 in v0
```

a) On the first provided template, show the complete execution of the `sum` function on the processor shown in Figure 199. **Assume that `a1` (i.e., the number of words) is 4.** Clearly mark squashed instructions by putting a cross (×) in the dedicated column (named `Squashed`) of the template. You should still show the execution of squashed instructions through the pipeline stages up until the moment they are discarded.
b) What is the IPC achieved by this execution? You should only account for instructions that went through the **W** stage (i.e., ignore instructions that ended up squashed from the pipeline).
c) How many times will the static branch assumption not taken be correct during one execution of the `sum` function? Give your answer in terms of `a1` and briefly justify it. You can assume that `a1` is strictly greater than 0.

Consider now a **dynamically scheduled out-of-order speculative** processor whose microarchitecture is shown in Figure 200. The processor fetches and decodes one RISC-V instruction per cycle. Once an instruction is decoded, it is dispatched to one of the functional units, depending on the instruction's type, via Reservation Stations (RS). Decoded instructions, including branches, are inserted at the bottom of the Reorder Buffer (ROB), which can commit one instruction per cycle. The processor has the following functional units with the corresponding latencies:

- Branch (**B**) — 1 cycle;
- Arithmetic (**A**) — 1 cycle; and
- Memory (**M1** and **M2**) — 2 cycles.

The processor is speculative in the sense that it executes speculatively the instructions after a branch before the branch decision is actually known. For this, it uses branch prediction, which is a strategy for reducing the negative impact of control hazards by guessing the outcome of a branch instruction, even before decoding it. The PREDICTOR component in Figure 200 implements the logic of branch prediction. If the actual outcome of a branch instruction proves to be opposite to the prediction, the speculatively executed instructions should be squashed—that is, the processor must act as if these instructions had never been executed.

The PREDICTOR uses the PC of the currently fetched instruction (`currentAddr`) to decide if the next instruction to fetch is not the next instruction in the program order. It has two outputs: (1) `predTaken` which is 1 if the PREDICTOR suggests that the branch is taken, and 0 if the PREDICTOR suggests that the branch is not taken or if the fetched instruction is not a branch; (2) `predAddr` which represents the address of the next instruction to fetch depending on the PREDICTOR's decision. If `predTaken` is 1,

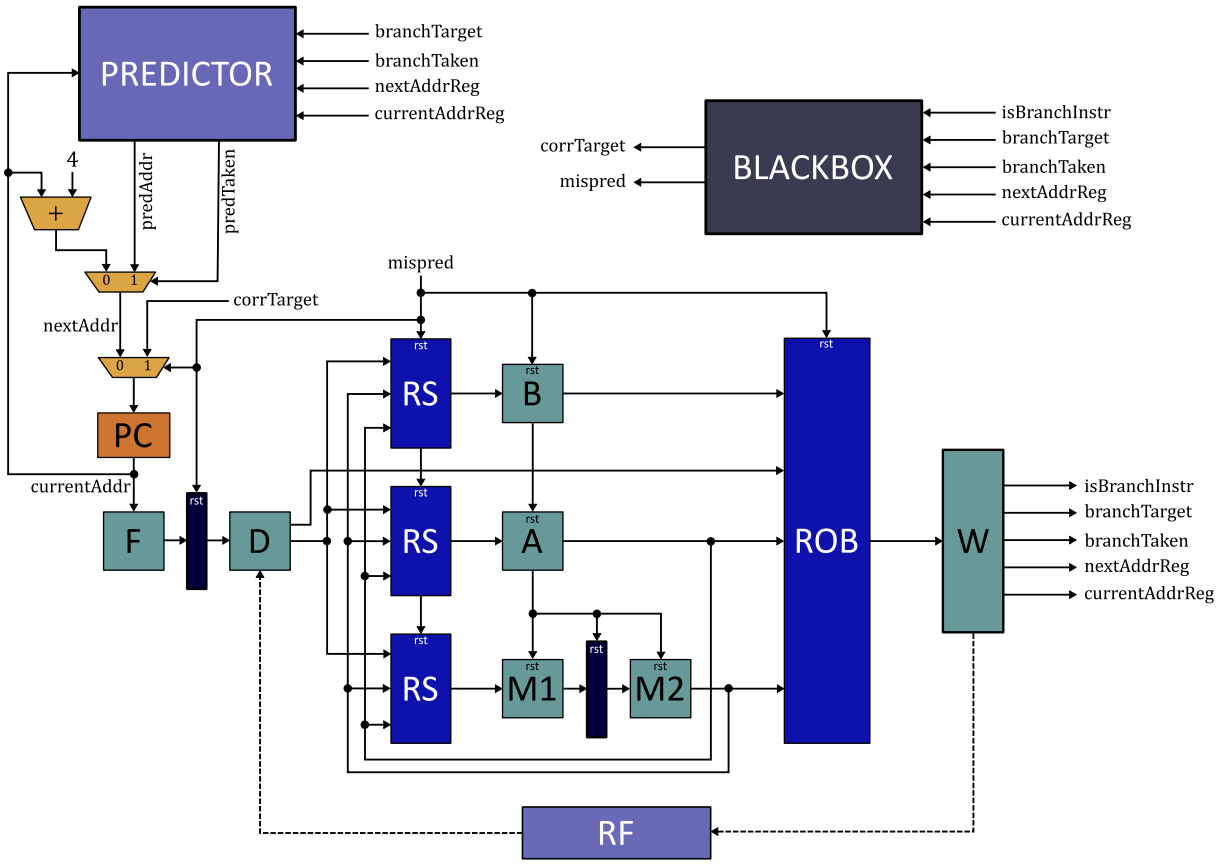


Figure 200: Dynamically scheduled out-of-order speculative processor architecture.

`predAddr` is the address of the branch destination, and if `predTaken` is 0, the value of `predAddr` is undefined. These two signals are used to decide what instruction to fetch next. The resulting instruction address `nextAddr` signal is transferred through the pipeline to the ROB, along with the `currAddr` signal. Figure 200 omits the transmission of the `currentAddr` and the `nextAddr` signals along the pipeline and only shows that they are propagated and output from the **W** stage (i.e., currentAddrReg and nextAddrReg).

The PREDICTOR "learns" to make better predictions with time thanks to the information provided to it by the **W** stage, which got the information, in the first place, from the ROB. This information informs the PREDICTOR of all committed branches. Whenever an instruction commits, the **W** stage provides the following signals: (1) `isBranchInstr` is 1 if the instruction corresponding to the `currAddrReg` address is a branch instruction, and is 0 otherwise; (2) `branchTarget` is the target address of the branch; (3) `branchTaken` is 1 if the actual decision of the branch (from the **B** stage) suggests that it should have been taken, and is 0 otherwise; (4) `nextAddrReg` is the `nextAddr` signal when the instruction was fetched; (5) `currAddrReg` is the value

of the PC when the instruction was fetched. Whenever the PREDICTOR encounters a new branch, it starts by assuming that the outcome is not taken, and then predicts the next branch outcome based on the actual outcome of the last execution of that same branch; that is, if the last execution of the branch was actually taken, the next execution is predicted to be taken too and vice versa.

The BLACKBOX is a circuit that uses the signals output from the **W** stage to provide information about how to fix a branch misprediction. To do so, it computes two signals: (1) `mispred` is 1 if the prediction of the branch ended up being wrong and is 0 if the prediction of the branch ended up being correct or if the instruction is not a branch; (2) `corrTarget` is the address of the correct instruction to execute after the branch in case of a misprediction. The signal `mispred` has essentially four functions: (i) loading the PC with `corrTarget` so that execution restarts from the place where the wrong prediction happened; (ii) cancelling the last incorrectly fetched and decoded instructions; (iii) resetting all execution units to cancel any ongoing instruction being executed; and (iv) removing any pending instruction from the RSs and from the ROB. The idea is that when a branch is committed and it turns out that we mispredicted its outcome, any subsequent instruction is removed from the processor and execution restarts from the correct instruction.

For arithmetic and memory instructions the ROB sets the `isBranchInstr` signal to 0 and the BLACKBOX should set the `mispred` to 0. The `branchTarget` and `branchTaken` signals have undefined values with arithmetic and memory instructions.
d) Draw the circuit of the BLACKBOX (see Figure 200 for the inputs and outputs) which produces the signals described above. You may only use basic logic gates (AND, OR, XOR, NOT), adders, multiplexers, and equality tests in your design.

Here are some functional details on the RSs and the ROB. RSs operate as follows:

- Results of the Decode stage are loaded into the relevant RS at the end of the decoding cycle, together with the ready operands.
- The result of each functional unit is available at the end of the last cycle of execution. At the end of that cycle, all RS entries are updated (so that the result can be used on the following cycle).
- If multiple entries are waiting to execute, the oldest instruction, in program order, has priority over newer ones. An instruction can be sent to the functional unit, at the earliest, the cycle after it gets loaded to the RS (i.e., one cycle after the decoding cycle).

The ROB operates as follows:

- Results of the Decode stage are loaded into the ROB at the end of the decoding cycle, to prepare the placeholder for the result. The ROB gets the available information needed at commit time, including the PC of the instruction

(the `currentAddr` signal) and the address of the next instruction fetched (the `nextAddr` signal).

- The results from all functional units are written into the ROB at the end of the last cycle of execution.
- An instruction can be committed, at the earliest, the cycle after it has completed execution.

Assume all RSs and the ROB are large enough so that they never fill up in practice, and that the PREDICTOR is large enough to predict all branches encountered in the program.

e) On the second provided template, show the complete execution of the `sum` function on the processor shown in Figure 200. **Again, assume that `a1` (i.e., the number of words) is 4.** Clearly mark squashed instructions by putting a cross (×) in the dedicated column (named `Squashed`) of the template. You should still show the execution of squashed instructions through the architecture up until the moment they are discarded. Mark with `x` the stalls resulting from data hazards, with `i` the stalls resulting from structural hazards in the functional units and with `o` the stalls resulting from structural hazards in the ROB and writeback stage.

f) What is the IPC achieved by this execution? You should only account for instructions that went through the **W** stage (i.e., ignore instructions that ended up squashed from the pipeline).

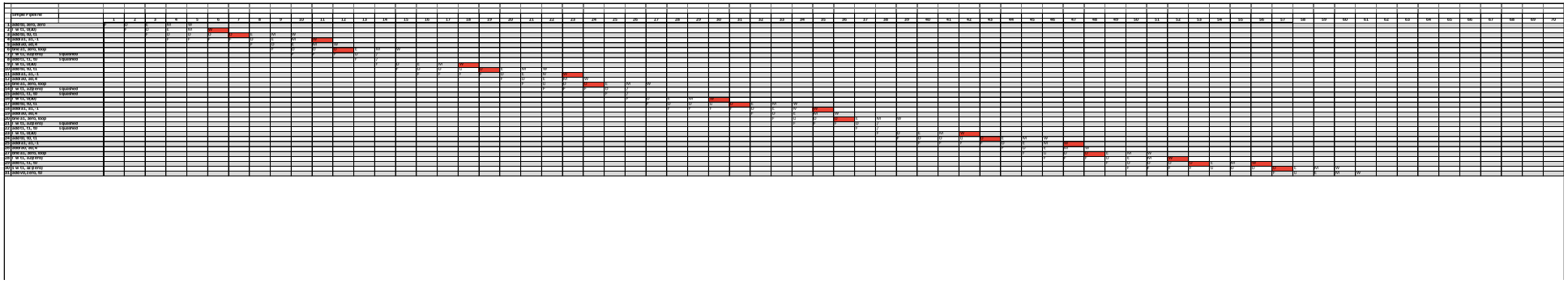
g) What is the exact accuracy of the PREDICTOR on this particular `sum` function? Give your answer in terms of `a1` and briefly justify it. As before, you should assume that the branch is predicted not taken the first time it is encountered. You can also assume that `a1` is strictly greater than 0.

h) In the context of the `sum` function, do you see any benefit in the dynamic branch prediction scheme implemented by the PREDICTOR over the static scheme (i.e., always assume not taken) exhibited by the first architecture? Briefly but clearly explain your reasoning.

Line #	Instruction	Squashed	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70

[Solution 20]

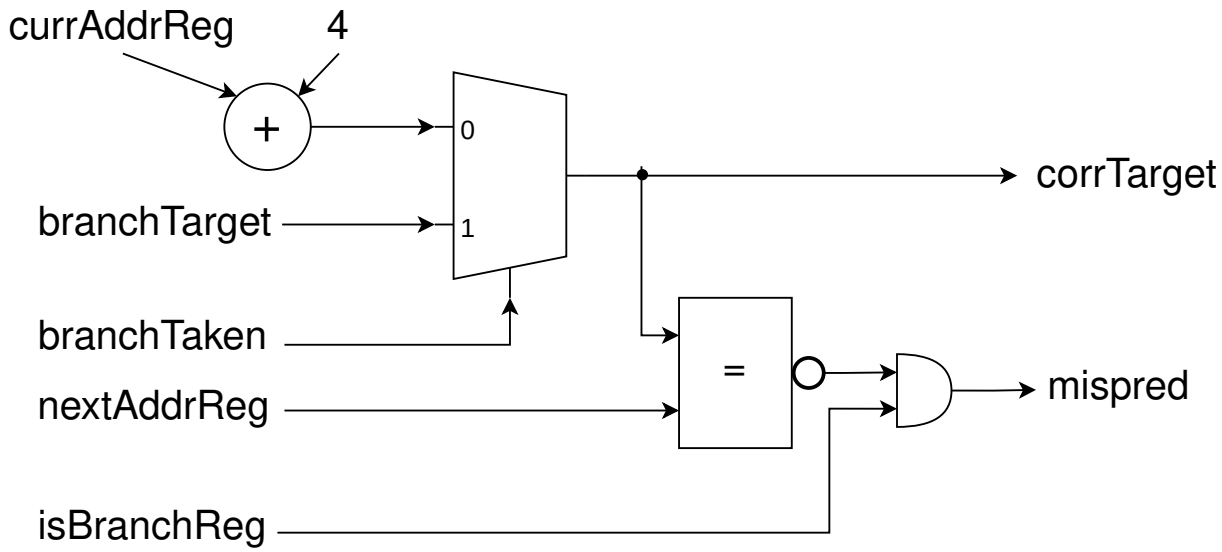
a)



b)

$$IPC = 25/61 = 0.41$$

c) Once in the end.
d)


currAddrReg
4
+
0
1
branchTarget
branchTaken
nextAddrReg
isBranchReg
=
corrTarget
mispred

e)

Big Architecture

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
1	add t0, zero, zero		F	D	A	W																																					
2	l w t1, 0(a0)			F	D	M1	M2	W																																			
3	add t0, t0, t1				F	D	x	A	W																																	
4	addi a1, a1, -1					F	D	I	A	W																																
5	addi a0, a0, 4						F	D	I	A	W																															
6	bne a1, zero, loop							F	D	B	o	W																														
7	l w t1, a2(zero)	squashed							F	D	M1	M2	/																													
8	add t1, t1, t0	squashed								F	D	x	/																												
9	s w t1, a2(zero)	squashed									F	D	/																												
10	add v0, zero, t0	squashed										F	/																												
11	l w t1, 0(a0)												F	D	M1	M2	W																								
12	add t0, t0, t1													F	D	x	A	W																							
13	addi a1, a1, -1														F	D	I	A	W																						
14	addi a0, a0, 4															F	D	I	A	W																					
15	bne a1, zero, loop																F	D	B	o	W																				
16	l w t1, 0(a0)																	F	D	M1	M2	W																			
17	add t0, t0, t1																		F	D	x	A	W																		
18	addi a1, a1, -1																			F	D	I	A	W																	
19	addi a0, a0, 4																				F	D	I	A	W																
20	bne a1, zero, loop																					F	D	B	o	W															
21	l w t1, 0(a0)																						F	D	M1	M2	W														
22	add t0, t0, t1																							F	D	x	A	W													
23	addi a1, a1, -1																								F	D	I	A	W												
24	addi a0, a0, 4																									F	D	I	A	W											
25	bne a1, zero, loop																										F	D	B	o	W										
26	l w t1, 0(a0)	squashed																										F	D	M1	M2	/									
27	add t0, t0, t1	squashed																											F	D	x	/									
28	addi a1, a1, -1	squashed																												F	D	/									
29	l w t1, 0(a0)	squashed																													F	/									
30	l w t1, a2(zero)																															F	D	M1	M2	W					
31	add t1, t1, t0																																F	D	x	A	W				
32	s w t1, a2(zero)																																	F	D	x	M1	M2	W			
33	add v0, zero, t0																																		F	D	A	o	o	W			

f)

$$IPC = 25/38 = 0.66$$

g) Accuracy of the predictor = correct predictions / total number of predictions
Total number of predictions = `a1`
Total number of correct predictions = `a1 - 2`
Accuracy of the predictor = (`a1` - 2) / `a1`
h) The `sum` function has a single branch that is responsible for managing the iterations of a loop. Since loops are repetitive by nature, the decision of the branch is going to be repetitive as well, therefore, the function greatly benefits from the dynamic branch prediction scheme that adjusts the predicted decision of the branch at runtime and thus gets plenty of correct predictions. On the contrary, the static scheme has a fixed prediction that assumes that a branch is always not taken which is opposite to the execution pattern of loops.

Part V: Multiprocessors

[Exercise 1]

Consider the multiprocessor system shown in Figure 201. This system has two 32-bit RISC-V processors, P1 and P2, with a 5-stage pipeline and all *forwarding paths* enabled (E→E, M→E, and W→D). The two processors have separate instruction memories from which a new instruction can be fetched every cycle, but they share the same data memory over a 256-bit shared data bus. When a processor needs to access the shared memory, it needs exclusive access to the shared bus. Hence, if the bus is busy, the processor stalls until it gets exclusive access to perform the bus transaction. Since accessing the shared memory can be a bottleneck, each processor has a private 128KB, *direct-mapped*, *write-back* data cache. The data caches can be accessed in a single cycle and each cache line holds 32 bytes of data.

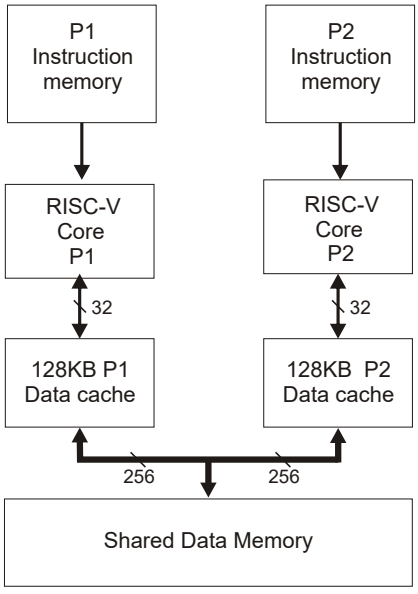


Figure 201: System Architecture

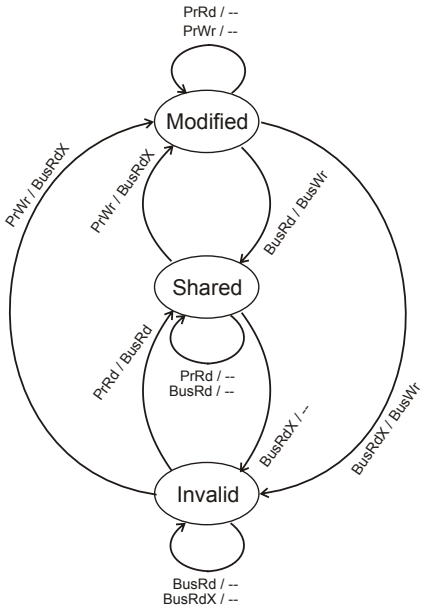
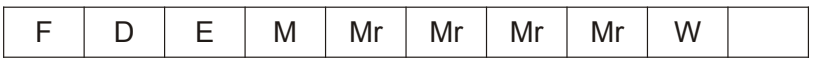


Figure 202: State transition diagram of MSI cache-coherency protocol

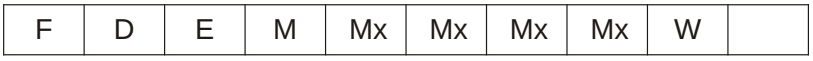
To keep the two caches coherent, we use the MSI cache-coherency protocol, shown in Figure 202. The code execution on the processors is similar to a normal RISC-V processor with a 5-stage pipeline (*Fetch*, *Decode*, *Execute*, *Memory*, and *WriteBack*). During code execution, a load/store instruction might cause a change in the state of a cache line. If this happens, the cache line's state is updated only when that load/store instruction enters the Write-back state.
In this system, the *Memory* stage accesses the data cache. So, in the best case, where the required data is held exclusively in the processor's data cache and no data exchanges

between the caches are necessary, the Memory stage needs only one cycle. However, if some signals and data need to be exchanged between the two caches or between the cache and the shared memory over the shared bus, there are several possibilities. The following diagrams illustrate these possibilities and how they execute on the processor:

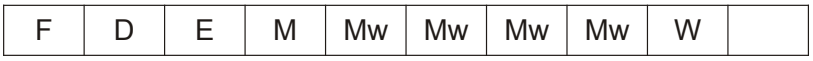
A **Reading data using the shared bus**: 'Mr' is used to indicate reading from the bus which takes 4 clock cycles after access to the shared bus is granted.



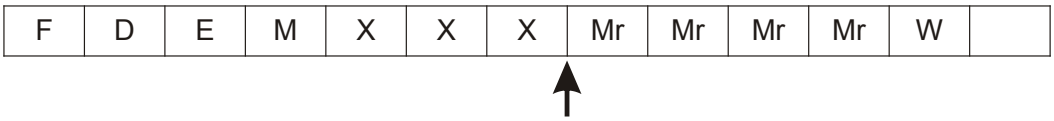
B **Reading data with intent to modify using the shared bus when the bus is free**: 'Mx' is used to indicate reading with intent to modify from the bus which takes 4 clock cycles after access to the shared bus is granted.



C **Writing data using the shared bus when the bus is free**: 'Mw' is used to indicate writing into the bus which takes 4 clock cycles after access to the shared bus is granted.



D **Accessing data (read) using the shared bus when the bus is currently busy**: Assume that the shared bus was busy until the point indicated by the arrow. The 'X' indicates pipeline stalls due to the non-availability of the shared bus when the processor wants to access it. The cases of read with intent to modify and write are similar with 'Mx' and 'Mw' replacing the 'Mr'.



During a data read, standard, or with intent to modify, if the data requested by a processor is available in the cache of another processor, the cache controller of the cache containing the data will provide the data on the bus without affecting the execution of the processor connected to it. For the processor that reads the data, this case is exactly the same as reading data from the shared bus. If both the processors are waiting to gain access (for read/write/read with intent to modify) to the shared bus,

the one that initiates the requests first is always given priority. In the event that both processors attempt to gain access at exactly the same execution cycle, the processor P1 is given priority.

a) According to the MSI protocol, what are the possible states a given cache line can have in the two processor caches at any given point in time?

b) Assume that the code in the table below is executed on the two processors, with each processor starting the execution of their respective first instructions on the first clock cycle. Also, assume that the two data caches were flushed just before starting the execution.

Code executed on Processor P1

```
1 lw   x1, 0x3000(zero)
2 lw   x2, 0x3004(zero)
3 add  x1, x1, x2
4 sw   x1, 0x3000(zero)
5 sw   x2, 0x3188(zero)
6 sw   x2, 0x3004(zero)
```

Code executed on Processor P2

```
1 add  x1, x3, x4
2 sw   x1, 0x3180(zero)
3 add  x1, x1, x8
4 slli x1, x1, 4
5 lw   x2, 0x3008(zero)
6 sw   x1, 0x3180(zero)
7 sw   x5, 0x3184(zero)
```

Draw an execution diagram showing the execution in each processor pipeline and the states of the relevant cache lines. You can indicate the cache line states using the letters **M**, **S**, and **I** for Modified, Shared, and Invalid.

c) Assume that we replaced the two data caches with *direct mapped*, *write-through* caches. Modify the MSI cache-coherence protocol's state diagram by only altering its state transitions to adapt it for the new system.

d) In the modified state diagram for question **c)** are all the three states really essential? If not, draw a simplified state diagram and explain your changes clearly.

[Solution 1]

a) For two corresponding lines in two caches, say Cache1 and Cache2, these are the simultaneous states that can have:

Line in Cache1		Line in Cache2
Invalid	-	Invalid
Invalid	-	Shared
Shared	-	Invalid
Shared	-	Shared
Modified	-	Invalid
Invalid	-	Modified

b) Solution is shown in Figure 203.
c) Solution is shown in Figure 204.
d) Solution is shown in Figure 205.

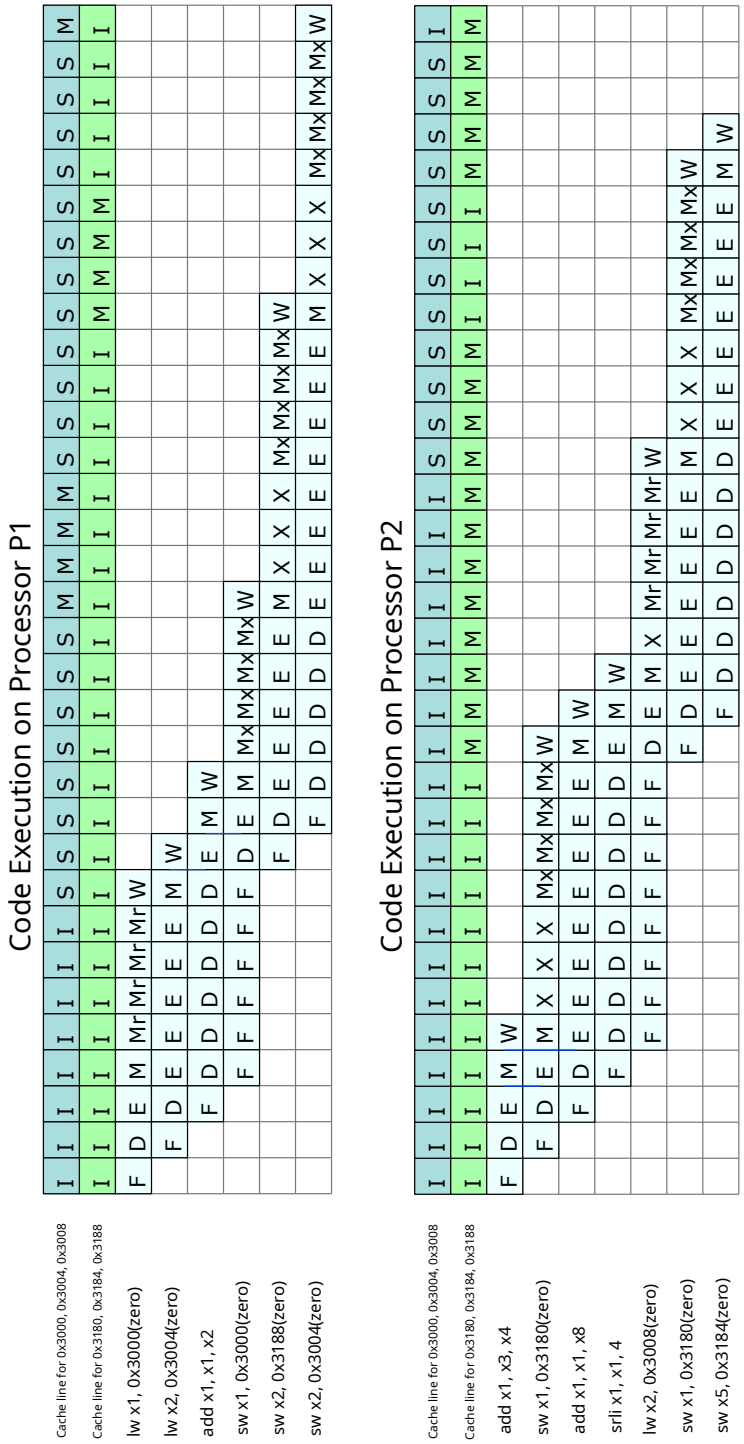


Figure 203: Cache coherency based on the MSI protocol

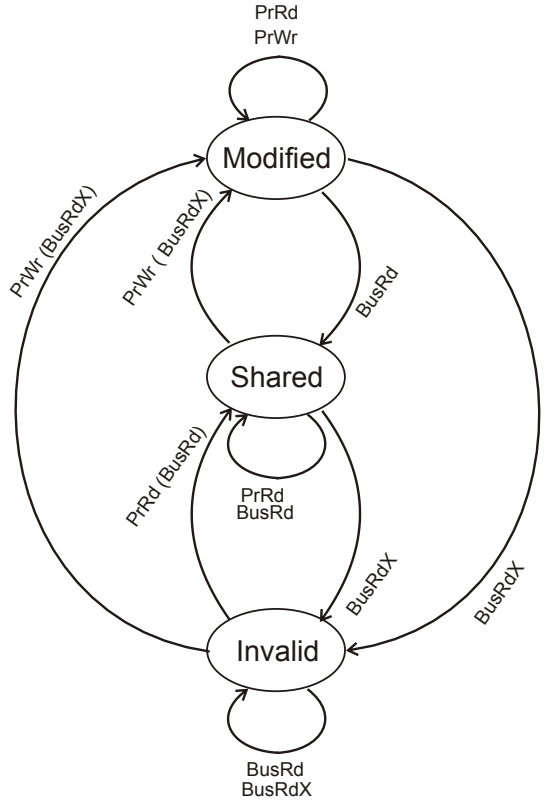


Figure 204: Simplified MSI protocol for *writethrough* cache

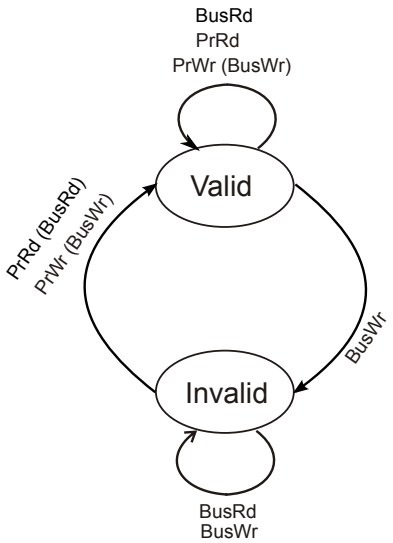


Figure 205: Simplified coherency state diagram for *writethrough* cache

[Exercise 2]

Consider the multiprocessor system shown in Figure 206 comprised of two superscalar processors, P1 and P2. Each processor has its own instruction memory and a 32 KB *private*, *direct-mapped*, *write-back* data cache with 32-byte cache lines. If the data access can not be serviced by the private cache, the data must be fetched from the shared data memory over the 256-bit shared bus. If the bus is busy, the processor stalls until it attains exclusive access. While using the shared bus, data can be either read directly from the memory or snooped from the bus while another cache performs a write-back operation.

To keep the data caches coherent, the system uses the MSI protocol whose state diagram is shown in Figure 207. During any data access operation, first, the cache is accessed, which takes one clock cycle. The operation completes if the access can be handled from the local cache without a bus transaction. Any bus transaction (`BusRd`, `BusWr`, `BusRdX`) adds exactly 3 clock cycles to the memory operation. If the bus is busy, then the operation will wait until it gains exclusive access to the bus. If there is contention between the processors for bus access or an operation requires writing-back a cache line from another processor's local cache, the processor that requests the operation first gets higher priority. If the requests arrive simultaneously, P1 has a higher priority over P2.

In this implementation, the cache line state is always updated at the beginning of the data access operation. Changes to a cache-line state caused by another processor's data access are performed when that processor accesses the shared bus. During a write-back operation, the cache line that is being written back is locked until the operation completes; any access to that line will cause the processor to stall.

a) Table 29 shows the current status of the load/store queues in the two processors. The order of the operations in the individual queues must be preserved during the execution. Using the provided template, indicate the cache line involved in every memory access, the initial and final states of the line before and after the access, and illustrate in the timing diagram the system cycles when each operation would be performed. In the execution diagram, indicate if the cache access results in a *hit* or *miss* when the cache access occurs, the cycles lost waiting for exclusive access to the shared bus, and the bus transaction performed. You must use `M` to indicate a cache miss, `H` to indicate a cache hit; `X` to indicate the cycles lost waiting for bus access, and `BusRd`, `BusRdX`, and `BusWr` to indicate the specific bus transaction performed. The first two data access operations have already been completed in the template for your reference. Show the execution of the rest of the operations by following the same format. Assume that the caches are empty before execution.

b) Assume that we change the cache coherence protocol in our system to MESI. The state diagram of the MESI protocol is shown in Figure 208. In the MESI protocol,

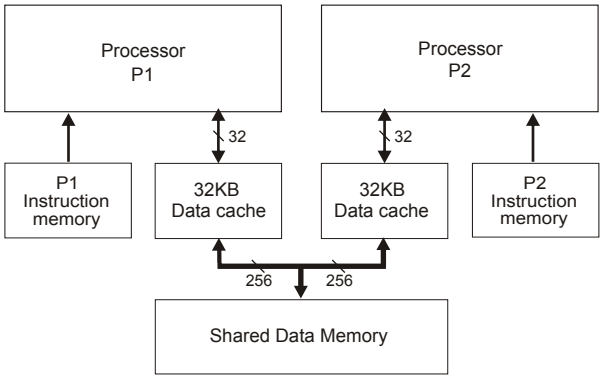


Figure 206: System Architecture

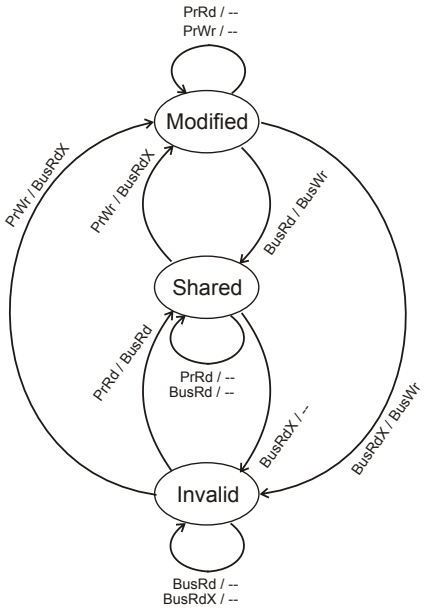


Figure 207: State transition diagram of MSI cache coherence protocol

instead of a single *Shared* state of the MSI protocol, we have *Exclusive* and *Shared* states. In the *Exclusive* state, only one cache holds a copy of the data and it can change the state of the data to *Modified* state without needing an extra bus transaction. In the *Shared* state, multiple caches may own copies of the data. Hence, a bus transaction is needed before the data can be modified by the local processor.

In the MESI protocol, when a processor reads a data item that was previously not in the cache, the status of the cache line holding this data is set to the *Shared* state if the data is already stored in another processor cache. If that data item is not being held in any other cache, its state is set to *Exclusive*. To know if the data is already in another cache or not, when one processor reads a data item with a `BusRd` transaction, the other caches must indicate if they hold a copy of that data item by setting `BusShr` to '1'. If this signal is '0', we know that the particular data item is not held in any other cache.

Now, considering the same initial load/store queue state shown in Table 29, show the execution of the memory operations by filling out the provided template. Use the same representation scheme as used for question 1. Assume again that the caches are empty prior to the execution.

c) According to the MESI protocol, what are the possible states a given cache line can have in two different processor caches at any given time? Consider the cache line to be "Invalid" if it does not exist in the cache, either because it has never been fetched or because it was evicted.

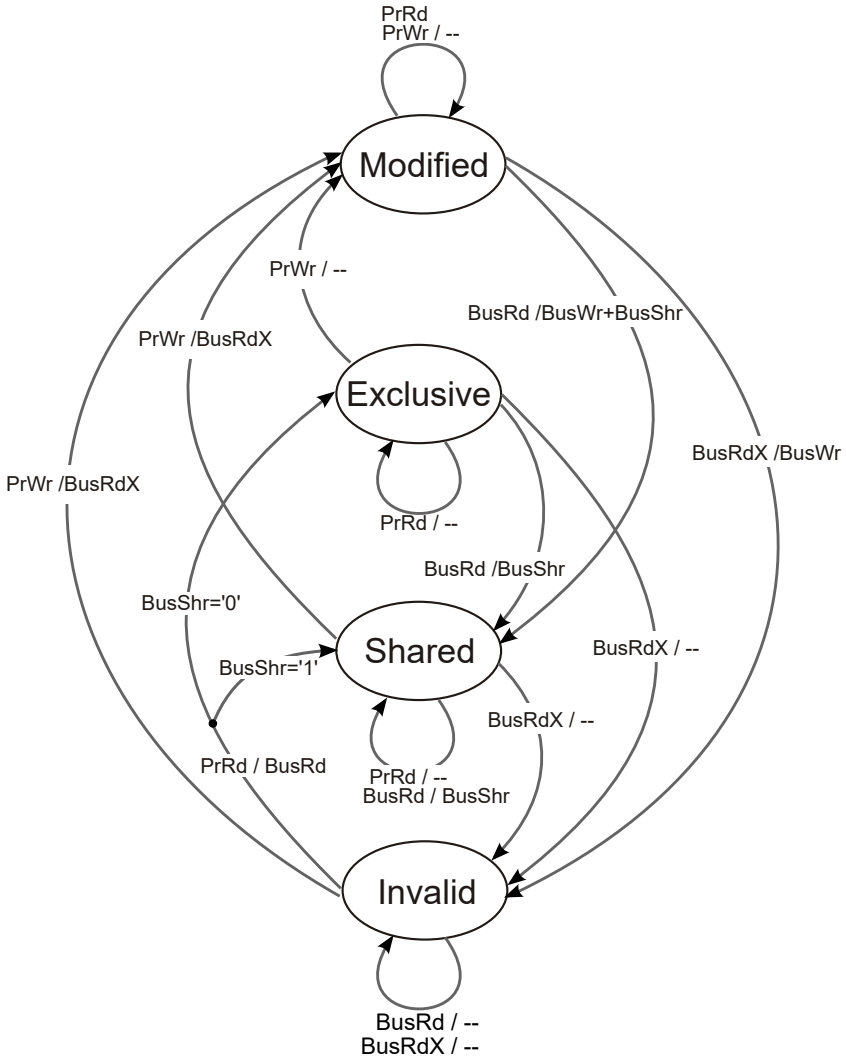


Figure 208: State transition diagram of the MESI protocol.

d) In the state diagram of the MESI protocol, does it make sense to have an edge from the *Shared* state to the *Exclusive* state? If so, what will be the benefit of this transition and why is it not already present in the general MESI protocol?

e) In the system described in question **b)**, a read access to a memory location which does not exist in the processor local cache, while already existing in *Shared* state in one or more caches, is supplied from the memory. An alternative would be to supply the data directly between the caches. Do you see any potential benefits in doing this? Do you see any problems with implementing this scheme when considering larger systems with many processors, each having their own private cache?

a)

Processor	Operation	State of cache line 256 in P1		State of cache line 272 in P1		State of cache line 256 in P2		State of cache line 272 in P2		Clock Cycles																																								
		Before	After	Before	After	Before	After	Before	After	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
P1	LOAD 0x2000	I	S	I	I	I	I	I	I	M	BusRd																																						
P2	LOAD 0x2204	S	S	I	I	I	I	I	S	M	X	X	X	BusRd																																			

b)

Processor	Operation	State of cache line 256 in P1		State of cache line 272 in P1		State of cache line 256 in P2		State of cache line 272 in P2		Clock Cycles																																								
		Before	After	Before	After	Before	After	Before	After	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40

Table 29: Initial state of the load/store queues on the two processors.

Load/Store Queue on P1	
1	`LOAD 0x2000`
2	`STORE 0x2008`
3	`LOAD 0x2204`
4	`STORE 0x200C`
5	`STORE 0x220C`

Load/Store Queue on P2	
1	`LOAD 0x2204`
2	`LOAD 0x220C`
3	`STORE 0x2200`
4	`LOAD 0x2000`
5	`LOAD 0x2004`
6	`STORE 0x2204`

[Solution 2]

The given set of load/store operations only uses the cache lines 256 and 272. Hence, we only need to track these two cache lines during the execution.
a) If the MSI cache coherence protocol was used, Figure 209 shows the execution of operation as the load/store operations are performed.
b) If the MESI cache coherence protocol was used, Figure 210 shows the execution of operation as the load/store operations are performed.
c) MESI protocol permits the following combination of states:

A Shared - Shared

B Invalid - Modified

C Invalid - Exclusive

D Invalid - Shared

E Invalid - Invalid

d) Yes, it is sensible to have this edge. Having this edge implies that under certain circumstances the cache line status can change from *Shared* state to the *Exclusive* state. This means that if the cache controller identifies that a previously shared cache line is no longer shared, it can promote it to an *Exclusive*. Hence, if the processor accessing that cache needs to write to that cache line, the operation can be carried out without any delays. However, to implement this feature, the cache controller would need to keep the information of the data held by other caches in the system. This significantly increases the controller's complexity and the amount of state information it needs to hold for each cache line.
e) One argument in favor of direct cache-to-cache sharing is that cache memory is faster than DRAM and this mode of transfer could potentially be faster. However, disturbing another cache to obtain the data might be more expensive (at a system level) than obtaining the data from the memory. Additionally, since there are multiple candidates who could supply the data, the coherency protocol also needs a selection algorithm to decide which cache should supply the data.

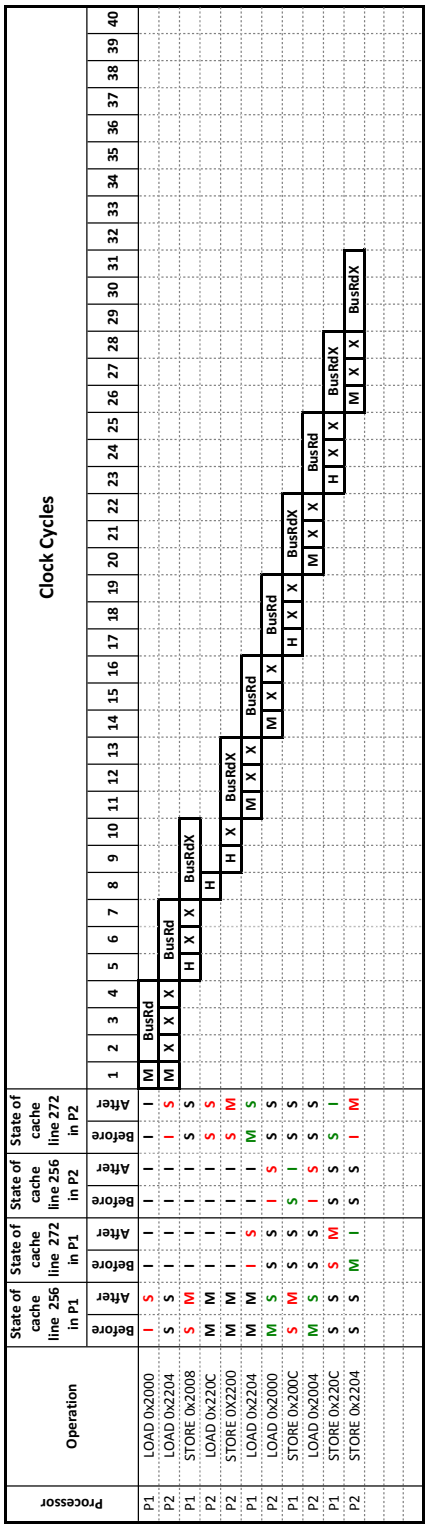


Figure 209: Execution of load/store operations when using MSI cache coherence protocol

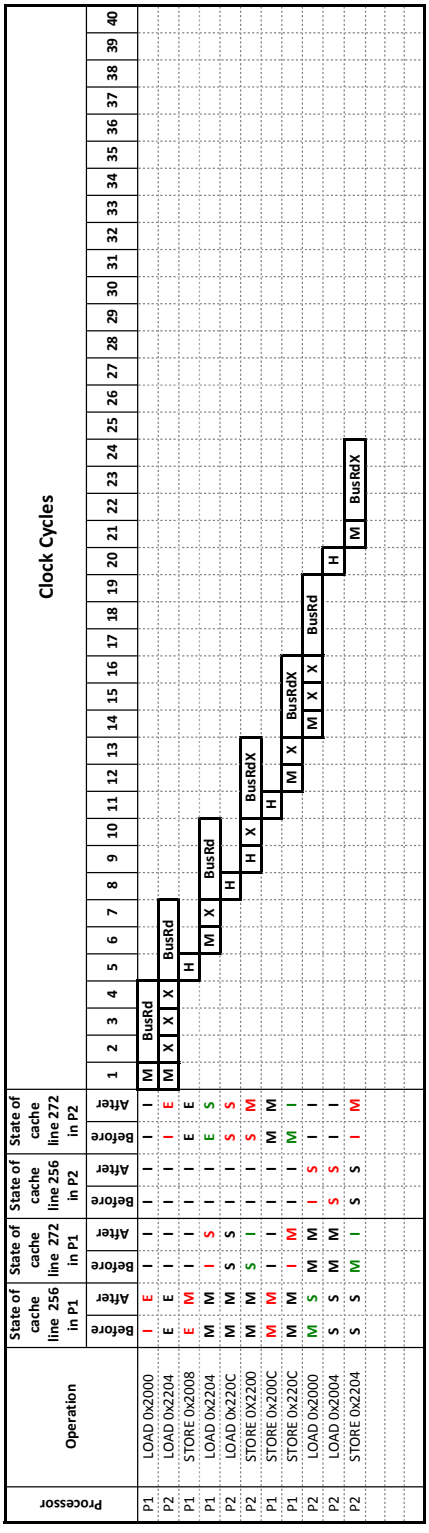


Figure 210: Execution of load/store operations when using MESI cache coherence protocol

[Exercise 3]

Consider the multiprocessor system of Figure 211 consisting of three 8-bit processors P0, P1, and P2 connected through a bus to the main memory. Each of the three processors has a single private 32-byte **direct-mapped** cache with 4 bytes per cache line. The memory is byte-addressed and starts at address 0x00.

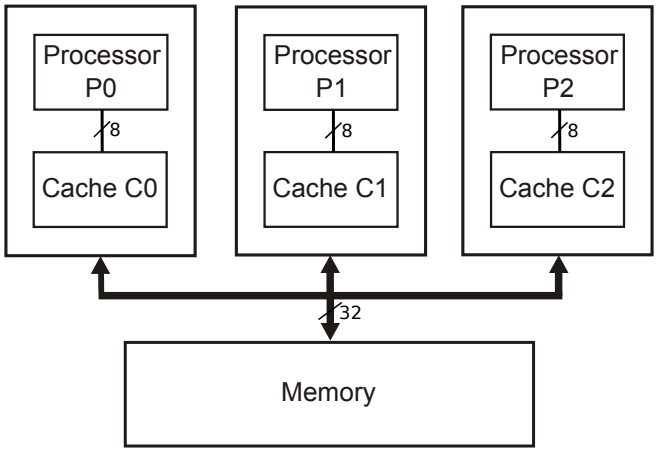


Figure 211: A multiprocessor architecture.

The caches are initialized with some data and each cache line has the initial state shown in Figure 212.

Cache C0		
Line	State	Address
0	S	0x00
1	M	0x04
2	--	--
3	S	0x2C
4	I	0x70
5	--	--
6	S	0x18
7	--	--

Cache C1		
Line	State	Address
0	S	0x00
1	S	0x24
2	M	0x28
3	I	0x0C
4	M	0x70
5	--	--
6	--	--
7	--	--

Cache C2		
Line	State	Address
0	S	0x20
1	I	0x04
2	M	0x08
3	I	0x0C
4	I	0x70
5	S	0x74
6	--	--
7	--	--

Figure 212: The initialization of the three caches.

Two different cache coherence protocols will be tested on this system, using the following sequences of load/store instructions. For simplicity, assume that the instructions are executed in the order shown, even if issued on different processors.

```
Set1
P2: load  0x04
P1: load  0x08
P0: load  0x08

Set2
P1: load  0x30
P0: load  0x30
P1: store 0x19

Set3
P0: store 0x00
P1: load  0x06
P2: load  0x1B
```

Executing each instruction can trigger several operations such as:

- R_{MEM}: Read a cache line from memory.
- R_{CACHE}: Read a cache line from a remote cache.
- W_{MEM}: Write a cache line back to memory.
- W_{CACHE}: Write a cache line to a remote cache.
- INV: Invalidate a cache line locally.

a) Assume that the three caches use the MSI snooping coherence protocol whose diagram is provided in Figure 213. Execute the sets of load/store instructions provided above. The first set is executed assuming that the cache is initialized as in Figure 212 while the remaining sets are executed *without* any cache re-initialization.

For each set (i.e., at the end of the third instruction of each set):

i) Show the contents and states of all three caches in the provided template. You may leave the unchanged entries empty.

ii) For each processor, list all the operations incurred by each instruction, in the provided template.

b) The MOSI protocol is an enhancement over the MSI protocol. It introduces an Owned state (denoted by O) with the following features:

- One cache at most may have a line of data in the Owned state; if another cache has the data as well, it would be in the Shared state.

- When a read miss occurs on processor A and processor B has the data in the Owned or Modified states, then B provides the requested line and transitions its state to (or stays in) Owned.
- When a write miss occurs on processor A and processor B has the data in the Owned or Modified states, then B provides the requested line and transitions its state to Invalid.
- Main memory is updated when a line in the Modified state or Owned state is replaced.
- Apart from the above differences, the MOSI protocol is identical to MSI.

Draw the MOSI protocol diagram.

c) Assuming that the caches use the MOSI protocol, re-execute the previous sets of instructions. The first set is executed assuming that the cache is initialized as in Figure 212 while the remaining sets are executed *without* any cache re-initialization.

For each set of instructions:

i) Show the contents and states of all three caches in the provided template. You may leave the unchanged entries empty.

ii) For each processor, list all the operations incurred by each instruction, in the provided template.

d) Compare the two protocols: What improvement does MOSI provide over MSI? Justify your answer.

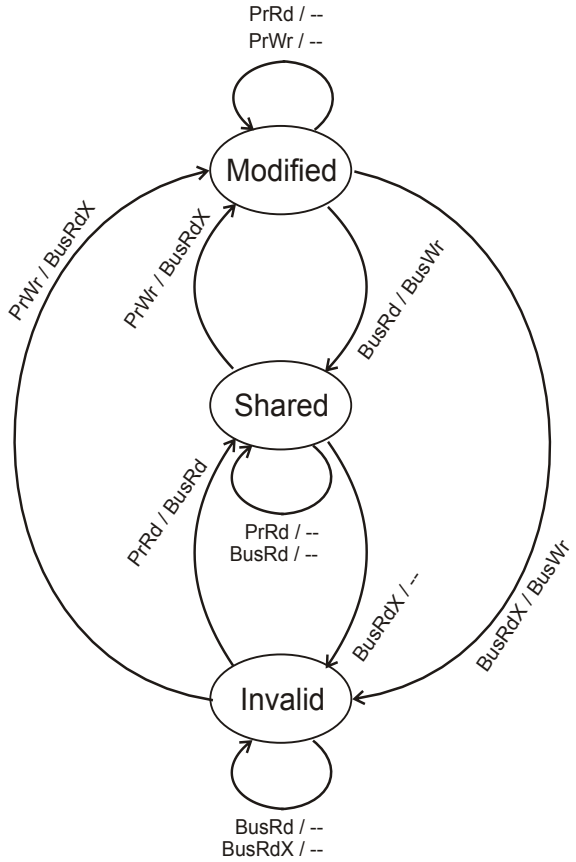


Figure 213: The MSI protocol diagram.

MSI protocol

Set1								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

Set2								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

Set3								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

	Set1		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load			
load			
load			

	Set2		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load			
load			
store			

	Set3		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
store			
load			
load			

MOSI protocol

Set1								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

Set2								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

Set3								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0			0			0		
1			1			1		
2			2			2		
3			3			3		
4			4			4		
5			5			5		
6			6			6		
7			7			7		

	Set1		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load			
load			
load			

	Set2		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load			
load			
store			

	Set3		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
store			
load			
load			

[Solution 3]

a) Please refer to the MSI protocol template below.
b)

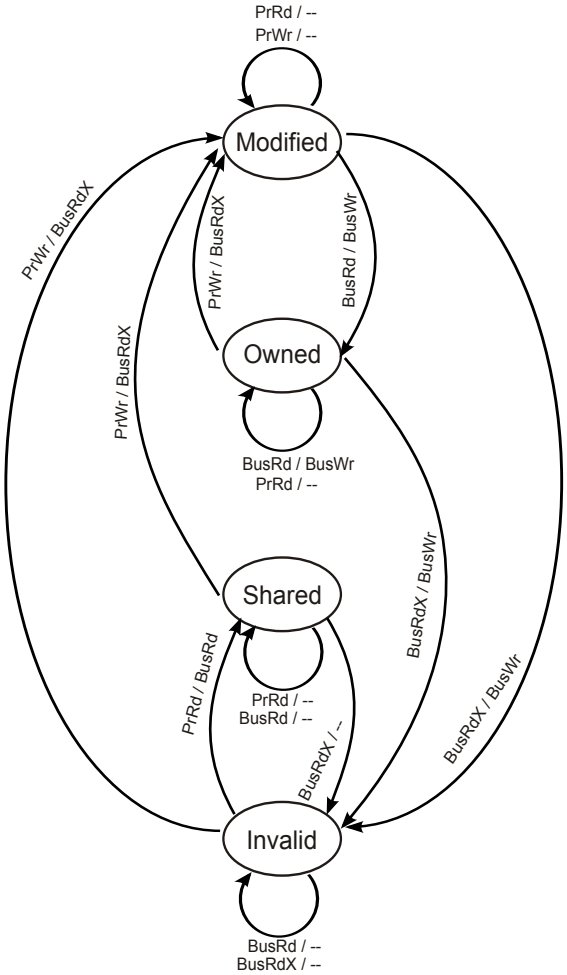


Figure 214: The MOSI protocol diagram.

c) Please refer to the MOSI protocol template below.
d) In the MOSI protocol, if cache X has a line in the Owned state, the other caches can read that line directly from X instead of main memory. So having an Owned state allows cache-to-cache transfer and reduces memory accesses.

MSI protocol

Set1								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	S	0x00	0	S	0x00	0	S	0x20
1	S	0x04	1	S	0x24	1	S	0x04
2	S	0x08	2	S	0x08	2	S	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	I	0x70	4	M	0x70	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	S	0x18	6	--	--	6	--	--
7	--	--	7	--	--	7	--	--

Set2								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	S	0x00	0	S	0x00	0	S	0x20
1	S	0x04	1	S	0x24	1	S	0x04
2	S	0x08	2	S	0x08	2	S	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	S	0x30	4	S	0x30	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	I	0x18	6	M	0x18	6	--	--
7	--	--	7	--	--	7	--	--

Set3								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	M	0x00	0	I	0x00	0	S	0x20
1	S	0x04	1	S	0x04	1	S	0x04
2	S	0x08	2	S	0x08	2	S	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	S	0x30	4	S	0x30	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	I	0x18	6	S	0x18	6	S	0x18
7	--	--	7	--	--	7	--	--

	Set1		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load	W_MEM W_CACHE	-	R_CACHE
load	-	W_MEM R_CACHE	W_MEM W_CACHE
load	R_MEM	-	-

	Set2		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load	-	W_MEM R_MEM	-
load	R_MEM	-	-
store	INV	R_MEM	-

	Set3		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
store	-	INV	-
load	-	R_MEM	-
load	-	W_MEM W_CACHE	R_CACHE

MOSI protocol

Set1								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	S	0x00	0	S	0x00	0	S	0x20
1	O	0x04	1	S	0x24	1	S	0x04
2	S	0x08	2	S	0x08	2	O	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	I	0x70	4	M	0x70	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	S	0x18	6	--	--	6	--	--
7	--	--	7	--	--	7	--	--

Set2								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	S	0x00	0	S	0x00	0	S	0x20
1	S	0x04	1	S	0x24	1	S	0x04
2	S	0x08	2	S	0x08	2	S	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	S	0x30	4	S	0x30	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	I	0x18	6	M	0x18	6	--	--
7	--	--	7	--	--	7	--	--

Set3								
Cache C0			Cache C1			Cache C2		
Line	State	Address	Line	State	Address	Line	State	Address
0	M	0x00	0	I	0x00	0	S	0x20
1	O	0x04	1	S	0x04	1	S	0x04
2	S	0x08	2	S	0x08	2	S	0x08
3	S	0x2C	3	I	0x0C	3	I	0x0C
4	S	0x30	4	S	0x30	4	I	0x70
5	--	--	5	--	--	5	S	0x74
6	I	0x18	6	O	0x18	6	S	0x18
7	--	--	7	--	--	7	--	--

	Set1		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load	W_MEM W_CACHE	-	R_CACHE
load	-	W_MEM R_CACHE	W_MEM W_CACHE
load	R_CACHE	-	W_CACHE

	Set2		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
load	-	W_MEM R_MEM	-
load	R_MEM	-	-
store	INV	R_MEM	-

	Set3		
	Cache C0	Cache C1	Cache C2
Instruction	Operations	Operations	Operations
store	-	INV	-
load	W_CACHE	R_CACHE	-
load	-	W_MEM W_CACHE	R_CACHE

[Exercise 4]

Consider the multiprocessor system of Figure 215, which consists of four 8-bit processors P0 to P3 along with their caches C0 to C3. Each cache is a 16-byte direct-mapped cache with one byte per cache line. Instead of having a centralized memory and a bus to enable memory-to-cache and cache-to-cache communications, the memory is distributed uniformly among the memories M0 to M3 and an interconnection network is used. A directory is added to keep track of the state of each memory location. In such a directory-based system, there is no bus snooping and all communications are done between individual nodes. For simplicity we assume that the memory is byte-addressed and that M0 starts at address 0x0000, M1 at 0x1000, M2 at 0x2000 and M3 at 0x3000.

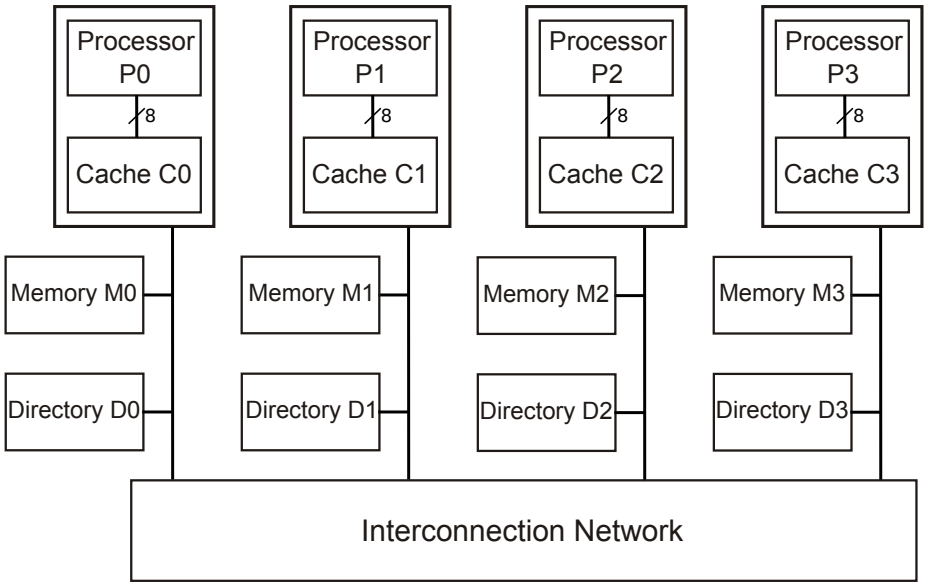


Figure 215: A multiprocessor architecture with distributed memory system.

A simple protocol, similar to the MSI protocol, is used to maintain the coherence of the different caches. As shown in Figure 216, the same states are used as in the case of a snoopy protocol; however, all bus operations (read, write, readX) are now network operations issued from/to the directory of the memory holding the data line in question. A new network operation *NetInv* is introduced to invalidate the data. It works similarly to *NetRdX*, except that the data is only invalidated without being read.

A second protocol, shown in Figure 217, is needed to update the states of the directories. Three states are used: *Uncached* meaning that the data exists only in memory and has not been read by any cache yet, *Shared* when the data exists in different caches, and *Exclusive* when the data has been modified in a certain cache and not written back to memory (that is, it is *dirty* as sometimes it is called).

Each directory keeps track, of every memory location, of the state of the data and maintains a list (known as *sharers*) of the processors that have it in their caches. The term *home* is used to indicate the processor responsible for the data (which happens when the address of the data is in the range covered by its directory/memory), while *owner* indicates the processor that has the most updated version (*exclusive*) of the data in its cache. For example, in the transition from the *Exclusive* to the *Shared* state, triggered by a read request from processor P, the directory sends a read request to the owner and then waits until it receives the data from the owner before sending it back to P and updating its list of *sharers*. This wait is represented using square brackets in Figure 217 (in this case, *[NetWr(from owner)]*).

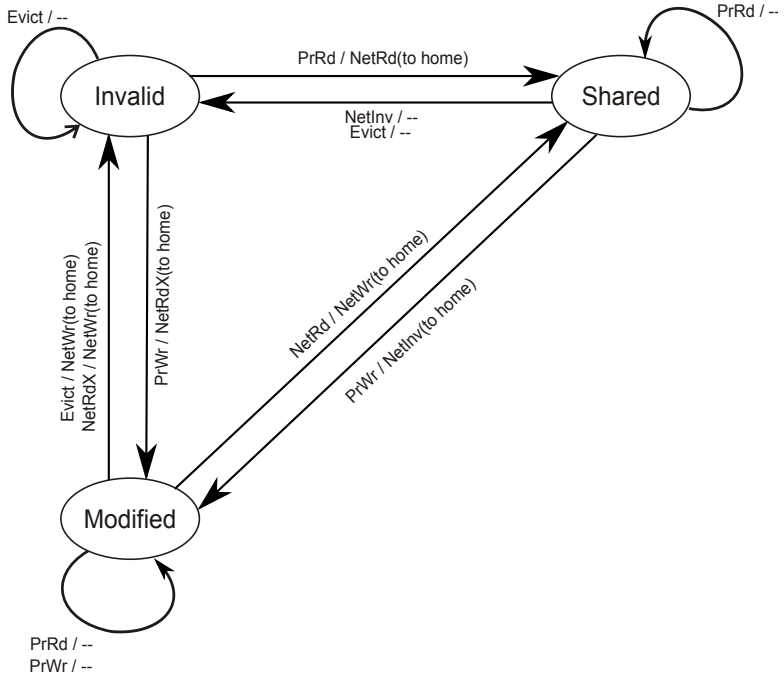


Figure 216: The protocol used to modify the states of the caches.

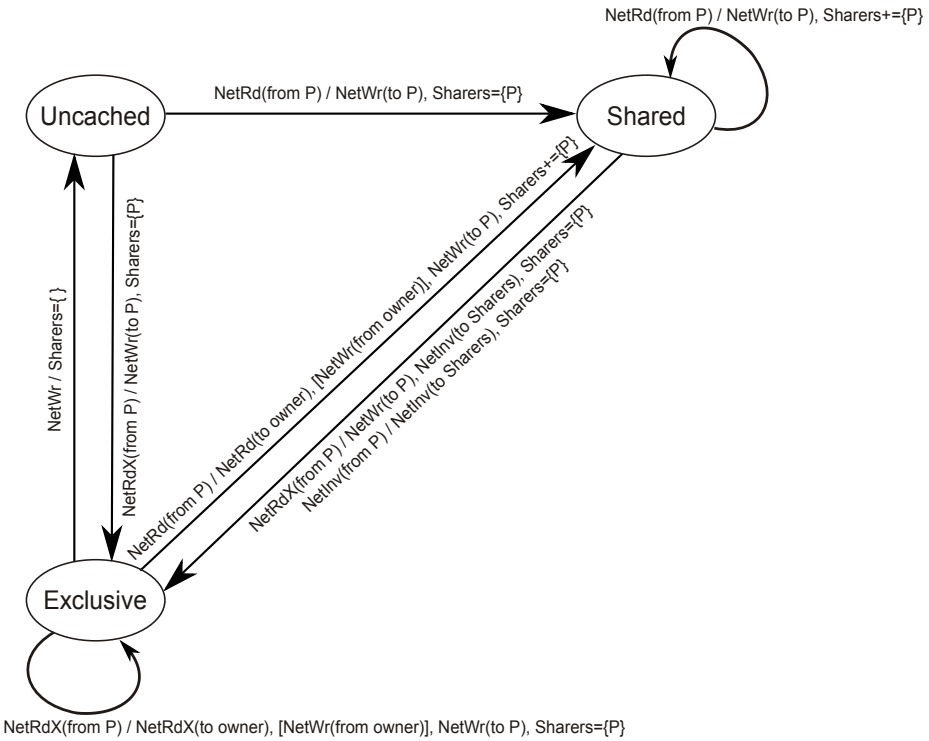


Figure 217: The protocol used to update the states of the directories.

Consider the following accesses executed in the given order and on the specified processors.

```
1 P1 Write 0x0006
2 P0 Write 0x0010
3 P2 Read  0x300F
4 P3 Read  0x0006
5 P1 Write 0x300F
6 P1 Write 0x321F
7 P0 Write 0x321F
8 P2 Write 0x0006
```

a) Using the cache protocol of Figure 216 and the directory protocol of Figure 217, and assuming that the caches are initially empty, trace each access and fill in the provided template:

i) The state transitions of the affected cache lines.

ii) The state transitions of the affected directory addresses.

iii) The network operations performed.

b) Assuming that each network operation takes a cycle to execute:

i) Which directory state transitions take the most cycles to execute (i.e., to deliver the required data)?

ii) Is there a simple way to speed up the execution while ensuring correct functionality?

Answer template

Access	Interconnection Network activity			Changes in Directories				Changes in Caches				
	From Processor	To Processor	Operation	Processor	Address	Old State	New State	Sharers	Processor	Line #	Address	Old State	New State

Figure 218: Template.

[Solution 4]

a) Please refer to the solution diagram of Figure 219.
b) From the transition diagrams and the solution of question 1:

i) When the directory transitions from Exclusive to Shared or from Exclusive to Exclusive, 4 cycles are needed for the data to reach its destination. The processor requests the data from the directory, but since the only valid value exists in the cache of another processor, the directory has to request the line and wait for it to be written back before sending it to the requesting processor.

ii) One way of speeding up the execution would be to enable processor-to-processor communication. Instead of waiting for the value to be written back to the directory before sending it to the requesting processor, the processor that has the exclusive value can send the data directly to the requesting one. This way, the data is available after 3 cycles (and an additional cycle can still used to update the directory but at least the processor has the data it needs and can proceed with its operation).

Solution

Access	Interconnection Network activity			Changes in Directories				Changes in Caches				
	From Processor	To Processor	Operation	Processor	Address	Old State	New State	Sharers	Processor	Line #	Address	Old State	New State
1: P1 Write 0x0006	P1	P0	NetRdX										
	P0	P1	NetWr	P0	0x0006	U	E	P1	P1	6	0x0006	I	M
2: P0 Write 0x0010	-	-	-	P0	0x0010	U	E	P0	P0	0	0x0010	I	M
3: P2 Read 0x300F	P2	P3	NetRd										
	P3	P2	NetWr	P3	0x300F	U	S	P2	P2	15	0x300F	I	S
4: P3 Read 0x0006	P3	P0	NetRd										
	P0	P1	NetRd										
	P1	P0	NetWr						P1	6	0x0006	M	S
	P0	P3	NetWr	P0	0x0006	E	S	P1, P3	P3	6	0x0006	I	S
5: P1 Write 0x300F	P1	P3	NetRdX										
	P3	P1	NetWr						P1	15	0x300F	I	M
	P3	P2	NetInv	P3	0x300F	S	E	P1	P2	15	0x300F	S	I
6: P1 Write 0x321F	P1	P3	NetWr	P3	0x300F	E	U	-	P1	15	0x300F	M	I
	P1	P3	NetRdX										
	P3	P1	NetWr	P3	0x321F	U	E	P1	P1	15	0x321F	I	M
7: P0 Write 0x321F	P0	P3	NetRdX										
	P3	P1	NetRdX										
	P1	P3	NetWr						P1	15	0x321F	M	I
	P3	P0	NetWr	P3	0x321F	E	E	P0	P0	15	0x321F	I	M
8: P2 Write 0x0006	P2	P0	NetRdX										
	P0	P2	NetWr						P2	6	0x0006	I	M
	P0	P1	NetInv						P1	6	0x0006	S	I
	P0	P3	NetInv	P0	0x0006	S	E	P2	P3	6	0x0006	S	I

Figure 219: Cache accesses.

[Exercise 5]

Consider **a system of four processors** P_0 to P_3, each having its own cache C_0 to C_3 respectively. The **MESI protocol** is used to maintain coherence among the different caches. With this protocol, a cache line can be in one of four states: Modified (M), Exclusive (E), Shared (S), and Invalid (I). The diagram of Figure 220 shows the states of the MESI protocol and the transitions between the different states.

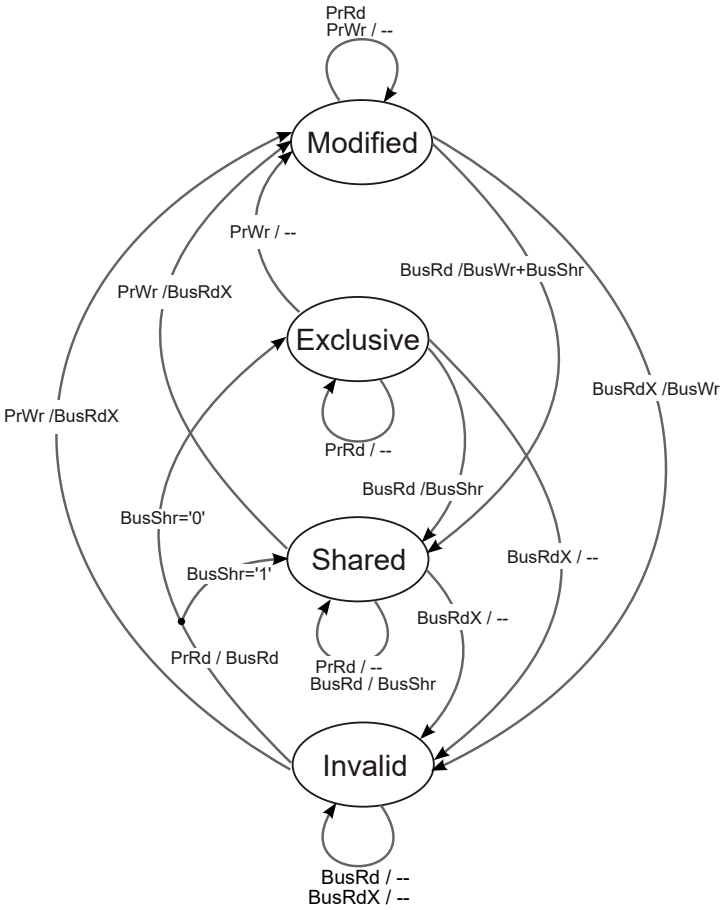


Figure 220: The MESI protocol diagram.

In the MESI protocol, the following needs to be considered when a processor reads a data item that was previously not in its own cache:

- If the data is already stored in another processor cache, then the status of the cache line holding this data is set to the *Shared* state.
- If the data is not already stored in any other cache, then the status of the cache line which will hold this data is set to the *Exclusive* state.

To know if the data is already in another cache or not, when one processor reads a data item with a `BusRd` transaction, the other caches must indicate if they hold a copy

of that data item by setting a special bus signal called `BusShr` to '1'. If this signal is '0', we know that the particular data item is not held in any other cache.

a) In the provided template, you are given the current state (labelled as **Old**) of a specific cache line in the four caches C_0 to C_3. For each cache, the address of the data saved in that cache line, as well as its state, is specified. When one of the processors (P_0 to P_3) executes an instruction that requires a memory access, then a state change is triggered in one or more of the caches (C_0 to C_3). This change is shown, for one cache, under the label **New**.

You are required to fill the rest of the template as follows:

A Specify the new state/address of the remaining caches.

B Specify the memory access that caused these changes. Your answer should be in the form:

$$P_x : \text{Load/Store Address}$$

where P_x is the processor that executed the memory operation, Load/Store is the type of memory access and Address is the address of the data to be loaded/stored.

If the **New** state cannot happen given the MESI protocol, or if no **single** instruction can change the state from Old to New, then you should write **"Impossible"**.

The solution of the first case in the template is provided as an example. Note that there might be up to 3 possible answers for each case (only 1 is possible in the example) and that **you are required to list all possible answers** in the provided space.

b) Under the MESI protocol, all data transfers always happen via shared memory (there is no direct cache-to-cache communication). Would enabling cache-to-cache communication be beneficial? If your answer is yes, provide an example of a situation where this would help; if your answer is no, justify your answer.

	Cache C0		Cache C1		Cache C2		Cache C3		Px: Load/Store Address
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	–
New	S	0x1000	**S**	**0x1000**	**S**	**0x2000**	**S**	**0x1000**	**P0: Load 0x1000**
	S	0x1000	–	–	–	–	–	–	–
	S	0x1000	–	–	–	–	–	–	–

Case 1:

Old	E	0x1000	I	0x1000	S	0x2000	I	0x1000	–
New	S	0x1000							
	S	0x1000							
	S	0x1000							

Case 2:

Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	–
New							I	0x1200	
							I	0x1200	
							I	0x1200	

Case 3:

Old	M	0x1300	I	0x1200	I	0x1100	E	0x1100	–
New							M	0x1100	
							M	0x1100	
							M	0x1100	

Case 4:

Old	I	0x1200	M	0x1200	S	0x1000	S	0x1000	–
New					M	0x1000			
					M	0x1000			
					M	0x1000			

Case 5:

Old	S	0x2000	I	0x2000	M	0x1600	E	0x1200	–
New	I	0x2000							
	I	0x2000							
	I	0x2000							

Case 6:

Old	E	0x1400	S	0x1000	S	0x1000	S	0x1000	–
New			S	0x1400					
			S	0x1400					
			S	0x1400					

Case 7:

Old	S	0x2200	M	0x2000	I	0x1600	S	0x2200	–
New					M	0x2200			
					M	0x2200			
					M	0x2200			

Case 8:

Old	S	0x1200	I	0x1200	S	0x1200	I	0x2000	–
New			E	0x2000					
			E	0x2000					
			E	0x2000					

Case 9:

Old	S	0x1400	S	0x1400	I	0x1400	I	0x1000	–
New			E	0x1400					
			E	0x1400					
			E	0x1400					

Case 10:

Old	I	0x1000	S	0x1000	S	0x1000	M	0x1200	–
New							S	0x1200	
							S	0x1200	
							S	0x1200	

Case 11:

Old	I	0x1000	S	0x2200	S	0x2200	M	0x1400	–
New							E	0x1400	
							E	0x1400	
							E	0x1400	

[Solution 5]

a) The solution is given in the filled template on the next page.
b) In the MESI protocol, modified data needs to be written into the main memory before it is shared, which can be time-consuming (for instance, it causes a delay when the modified data line needs to be fetched by another processor). Enabling cache-to-cache communication would help circumvent this issue.

	Cache C0		Cache C1		Cache C2		Cache C3		Px: Load/Store Address
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	–
New	S	0x1000	**S**	**0x1000**	**S**	**0x2000**	**S**	**0x1000**	**P0: Load 0x1000**
	S	0x1000	–	–	–	–	–	–	–
	S	0x1000	–	–	–	–	–	–	–

Case 1:

Old	E	0x1000	I	0x1000	S	0x2000	I	0x1000	–
New	S	0x1000	S	0x1000	S	0x2000	I	0x1000	P1: Load 0x1000
	S	0x1000	I	0x1000	S	0x1000	I	0x1000	P2: Load 0x1000
	S	0x1000	I	0x1000	S	0x2000	S	0x1000	P3: Load 0x1000

Case 2:

Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	–
New	M	0x1200	I	0x1200	I	0x1200	I	0x1200	P0: Store 0x1200
	I	0x1400	M	0x1200	I	0x1200	I	0x1200	P1: Store 0x1200
	I	0x1400	I	0x1200	M	0x1200	I	0x1200	P2: Store 0x1200

Case 3:

Old	M	0x1300	I	0x1200	I	0x1100	E	0x1100	–
New	M	0x1300	I	0x1200	I	0x1100	M	0x1100	P3: Store 0x1100
	–	–	–	–	–	–	M	0x1100	-
	–	–	–	–	–	–	M	0x1100	-

Case 4:

Old	I	0x1200	M	0x1200	S	0x1000	S	0x1000	–
New	I	0x1200	M	0x1200	M	0x1000	I	0x1000	P2: Store 0x1000
	–	–	–	–	M	0x1000	–	–	-
	–	–	–	–	M	0x1000	–	–	-

Case 5:

Old	S	0x2000	I	0x2000	M	0x1600	E	0x1200	–
New	I	0x2000	M	0x2000	M	0x1600	E	0x1200	P1: Store 0x2000
	I	0x2000	I	0x2000	M	0x2000	E	0x1200	P2: Store 0x2000
	I	0x2000	I	0x2000	M	0x1600	M	0x2000	P3: Store 0x2000

Case 6:

Old	E	0x1400	S	0x1000	S	0x1000	S	0x1000	–
New	S	0x1400	S	0x1400	S	0x1000	S	0x1000	P1: Load 0x1400
	–	–	S	0x1400	–	–	–	–	-
	–	–	S	0x1400	–	–	–	–	-

Case 7:

Old	S	0x2200	M	0x2000	I	0x1600	S	0x2200	–
New	I	0x2200	M	0x2000	M	0x2200	I	0x2200	P2: Store 0x2200
	–	–	–	–	M	0x2200	–	–	-
	–	–	–	–	M	0x2200	–	–	-

Case 8:

Old	S	0x1200	I	0x1200	S	0x1200	I	0x2000	–
New	S	0x1200	E	0x2000	S	0x1200	I	0x2000	P1: Load 0x2000
	–	–	E	0x2000	–	–	–	–	-
	–	–	E	0x2000	–	–	–	–	-

Case 9:

Old	S	0x1400	S	0x1400	I	0x1400	I	0x1000	–
New	–	–	E	0x1400	–	–	–	–	Impossible
	–	–	E	0x1400	–	–	–	–	-
	–	–	E	0x1400	–	–	–	–	-

Case 10:

Old	I	0x1000	S	0x1000	S	0x1000	M	0x1200	–
New	S	0x1200	S	0x1000	S	0x1000	S	0x1200	P0: Load 0x1200
	I	0x1000	S	0x1200	S	0x1000	S	0x1200	P1: Load 0x1200
	I	0x1000	S	0x1000	S	0x1200	S	0x1200	P2: Load 0x1200

Case 11:

Old	I	0x1000	O	0x2200	S	0x2200	M	0x1400	–
New	–	–	–	–	–	–	E	0x1400	Impossible
	–	–	–	–	–	–	E	0x1400	-
	–	–	–	–	–	–	E	0x1400	-

[Exercise 6] Cache Coherence

Consider a system of four processors P_0 to P_3, each having its own cache C_0 to C_3 respectively. The MOSI protocol is used to maintain coherence among the different caches. With this protocol, a cache line can be in one of four states: Modified (M), Owned (O), Shared (S), and Invalid (I). The diagram of Figure 221 shows the states of the MOSI protocol and the transitions between the different states.

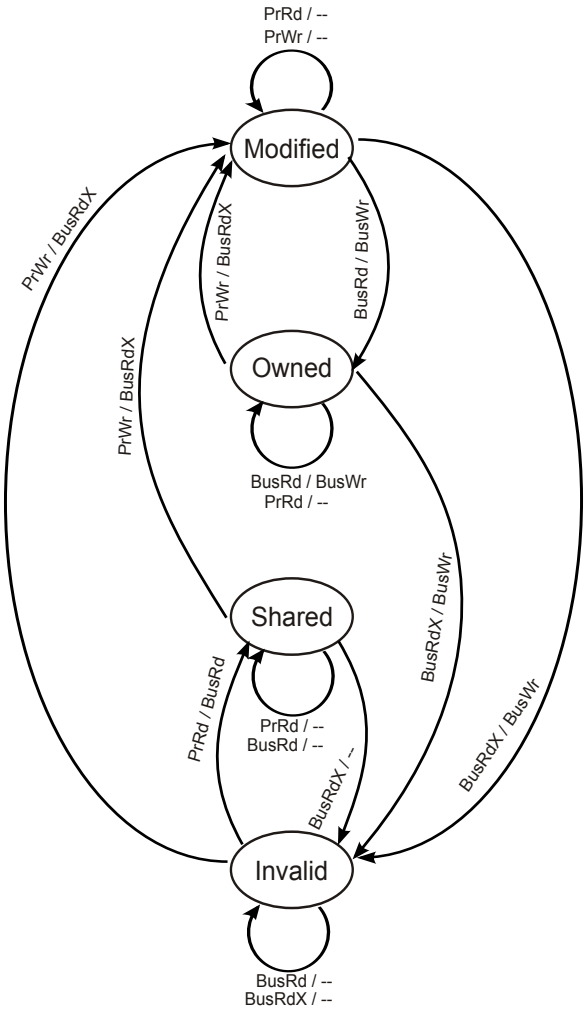


Figure 221: The MOSI protocol diagram.

a) In the provided template, you are given the current state, labeled as **Old**, of a specific cache line in the four caches C_0 to C_3. For each cache, the address of the data saved in that cache line, as well as its state, is specified. One of the processors P_0 to P_3 executes an instruction that requires a memory access and causes a change in one or more of these caches. This change is shown, for one cache, under the label **New**. You are required to fill the rest of the template as follows:

i) Specify the new state/address of the remaining caches.

ii) Specify the memory access that caused these changes. Your answer should be in the form:

$$P_x : \text{Load/Store } \textit{Address} \tag{1}$$

where P_x is the processor that executed the memory operation, Load/Store is the type of memory access and *Address* is the address of the data to be loaded/stored.

The solution of the first case in the template is provided as an example. Note that there might be more than one possible answer for each case and that you are required to list **all possible answers** in the provided space (each answer should be listed in a separate row of the table; if there are fewer answers than provided rows, the extra rows should be left empty).

b) According to the state diagram, are the caches:

i) *write-through* or *write-back*?

ii) *write-allocate* or *not write-allocate* (i.e., in case of write miss, is the data placed in the cache or only in memory)?

Justify your answers.

c) What is the maximum number of caches which can have the same line of data in the Owned state? Justify your answer with an example.

d) The diagram in Figure 222 shows the MSI protocol. What is the benefit of the MOSI protocol over MSI? Justify your answer with an example.

Exercise 2 - Template

	Cache C0		Cache C1		Cache C2		Cache C3		Px: Load/Store Address
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	–
New	S	0x1000	**S**	**0x1000**	**S**	**0x2000**	**S**	**0x1000**	**P0: Load 0x1000**
	S	0x1000	–	–	–	–	–	–	–
	S	0x1000	–	–	–	–	–	–	–

Old	I	0x1400	S	0x1000	S	0x1000	M	0x1400	–
New					M	0x1400			
					M	0x1400			
					M	0x1400			

Old	M	0x1000	O	0x2000	S	0x2000	I	0x1300	–
New			I	0x2000					
			I	0x2000					
			I	0x2000					

Old	I	0x1200	O	0x1300	I	0x1100	M	0x1100	–
New	M	0x1200							
	M	0x1200							
	M	0x1200							

Old	S	0x1000	I	0x1400	O	0x1000	I	0x1000	–
New			M	0x1000					
			M	0x1000					
			M	0x1000					

Old	S	0x1000	I	0x1000	O	0x1400	M	0x1600	–
New							M	0x1000	
							M	0x1000	
							M	0x1000	

Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	–
New			S	0x1200					
			S	0x1200					
			S	0x1200					

Old	S	0x3000	M	0x2000	S	0x3000	O	0x3000	–
New					I	0x3000			
					I	0x3000			
					I	0x3000			

Old	S	0x2000	O	0x4000	S	0x2000	O	0x2000	–
New	S	0x4000							
	S	0x4000							
	S	0x4000							

Old	S	0x1000	I	0x1000	O	0x2000	S	0x4000	–
New			M	0x1000					
			M	0x1000					
			M	0x1000					

Old	I	0x1000	O	0x1000	S	0x1000	M	0x1200	–
New			M	0x1200					
			M	0x1200					
			M	0x1200					

Old	S	0x1400	S	0x1400	I	0x1400	O	0x1400	–
New							M	0x1400	
							M	0x1400	
							M	0x1400	

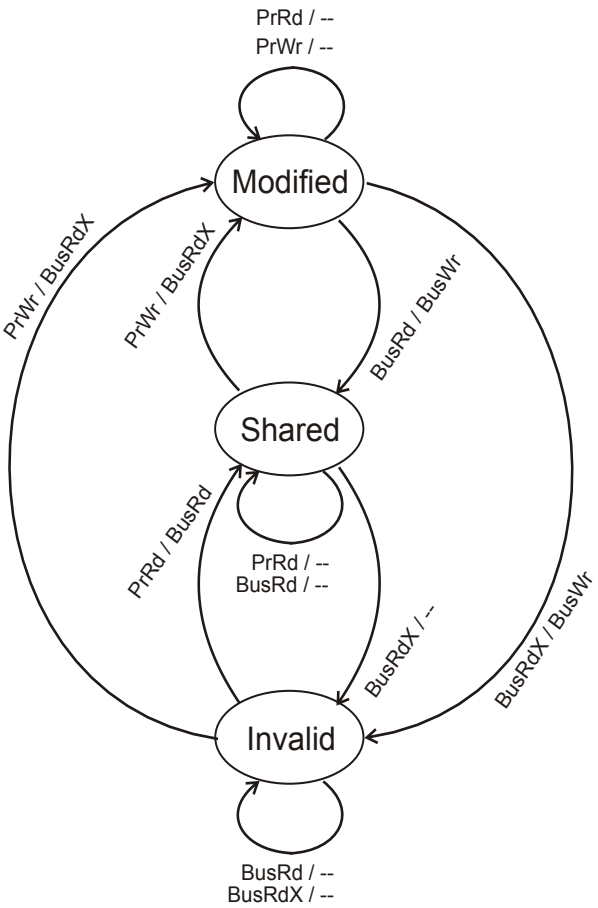


Figure 222: The MSI protocol diagram.

[Solution 6]

a) The solution is shown in the filled template below.
b) The caches are write-back—a write to the cache does not always imply a write to memory (e.g., I to M does not have a BusWr). The caches are write-allocate, as the data placed in the cache in case of a write miss.
c) One cache at most may have a line of data in the Owned state; if another cache has the data as well, it would be in the Shared state.
d) In the MOSI protocol, if cache X has a line in the Owned state, the other caches can read that line directly from X instead of main memory. Therefore, having an Owned state allows cache-to-cache transfer and reduces the number of memory accesses.

	Cache C0		Cache C1		Cache C2		Cache C3		Px: Load/Store Address
	State	Address	State	Address	State	Address	State	Address	
Old	I	0x1000	S	0x1000	S	0x2000	S	0x1000	–
New	S	0x1000	**S**	**0x1000**	**S**	**0x2000**	**S**	**0x1000**	**P0: Load 0x1000**
	S	0x1000	–	–	–	–	–	–	–
	S	0x1000	–	–	–	–	–	–	–

Case 1:

Old	I	0x1400	S	0x1000	S	0x1000	M	0x1400	–
New	**I**	**0x1400**	**S**	**0x1000**	M	0x1400	**I**	**0x1400**	**P2: Store 0x1400**
	–	**–**	**–**	**–**	M	0x1400	**–**	**–**	**–**
	–	**–**	**–**	**–**	M	0x1400	**–**	**–**	**–**

Case 2:

Old	M	0x1000	O	0x2000	S	0x2000	I	0x1300	–
New	**M**	**0x2000**	I	0x2000	**I**	**0x2000**	**I**	**0x1300**	**P0: Store 0x2000**
	M	**0x1000**	I	0x2000	**M**	**0x2000**	**I**	**0x1300**	**P2: Store 0x2000**
	M	**0x1000**	I	0x2000	**I**	**0x2000**	**M**	**0x2000**	**P3: Store 0x2000**

Case 3:

Old	I	0x1200	O	0x1300	I	0x1100	M	0x1100	–
New	M	0x1200	**O**	**0x1300**	**I**	**0x1100**	**M**	**0x1100**	**P0: Store 0x1200**
	M	0x1200	**–**	**–**	**–**	**–**	**–**	**–**	**–**
	M	0x1200	**–**	**–**	**–**	**–**	**–**	**–**	**–**

Case 4:

Old	S	0x1100	I	0x1400	O	0x1000	I	0x1000	–
New	**I**	**0x1000**	M	0x1000	**I**	**0x1000**	**I**	**0x1000**	**P1: Store 0x1000**
	–	**–**	M	0x1000	**–**	**–**	**–**	**–**	**–**
	–	**–**	M	0x1000	**–**	**–**	**–**	**–**	**–**

Case 5:

Old	S	0x1000	I	0x1000	O	0x1400	M	0x1600	–
New	**I**	**0x1000**	**I**	**0x1000**	**O**	**0x1400**	M	0x1000	**P3: Store 0x1000**
	–	**–**	**–**	**–**	**–**	**–**	M	0x1000	**–**
	–	**–**	**–**	**–**	**–**	**–**	M	0x1000	**–**

Case 6:

Old	I	0x1400	I	0x1200	I	0x1200	M	0x1200	–
New	**I**	**0x1400**	S	0x1200	**I**	**0x1200**	**O**	**0x1200**	**P1: Load 0x1200**
	–	**–**	S	0x1200	**–**	**–**	**–**	**–**	**–**
	–	**–**	S	0x1200	**–**	**–**	**–**	**–**	**–**

Case 7:

Old	S	0x3000	M	0x2000	S	0x3000	O	0x3000	–
New	**M**	**0x3000**	**M**	**0x2000**	I	0x3000	**I**	**0x3000**	**P0: Store 0x3000**
	I	**0x3000**	**M**	**0x3000**	I	0x3000	**I**	**0x3000**	**P1: Store 0x3000**
	I	**0x3000**	**M**	**0x2000**	I	0x3000	**M**	**0x3000**	**P3: Store 0x3000**

Case 8:

Old	S	0x2000	O	0x4000	S	0x2000	O	0x2000	–
New	S	0x4000	**O**	**0x4000**	**S**	**0x2000**	**O**	**0x2000**	**P0: Load 0x4000**
	S	0x4000	**–**	**–**	**–**	**–**	**–**	**–**	**–**
	S	0x4000	**–**	**–**	**–**	**–**	**–**	**–**	**–**

Case 9:

Old	S	0x1000	I	0x1000	O	0x2000	S	0x4000	–
New	**I**	**0x1000**	M	0x1000	**O**	**0x2000**	**S**	**0x4000**	**P1: Store 0x1000**
	–	**–**	M	0x1000	**–**	**–**	**–**	**–**	**–**
	–	**–**	M	0x1000	**–**	**–**	**–**	**–**	**–**

Case 10:

Old	I	0x1000	O	0x1000	S	0x1000	M	0x1200	–
New	**I**	**0x1000**	M	0x1200	**S**	**0x1000**	**I**	**0x1200**	**P1: Store 0x1200**
	–	**–**	M	0x1200	**–**	**–**	**–**	**–**	**–**
	–	**–**	M	0x1200	**–**	**–**	**–**	**–**	**–**

Case 11:

Old	S	0x1400	S	0x1400	I	0x1400	O	0x1400	–
New	**I**	**0x1400**	**I**	**0x1400**	**I**	**0x1400**	M	0x1400	**P3: Store 0x1400**
	–	**–**	**–**	**–**	**–**	**–**	M	0x1400	**–**
	–	**–**	**–**	**–**	**–**	**–**	M	0x1400	**–**

[Exercise 7] Cache Coherence

MSI is a simple invalidation cache-coherence protocol. With this protocol, a cache line can be in one of three states: **Modified** (dirty) where the data in the cache line has been modified and thus is inconsistent with the data in the main memory, **Shared** where the cache line is not modified and is present in at least one cache, and **Invalid** where the cache line does not hold a valid copy of the data, and valid copies of data can be either in main memory or in another cache. Figure 223 shows the states of the MSI protocol along with the transitions between the states, and the transactions on the bus.

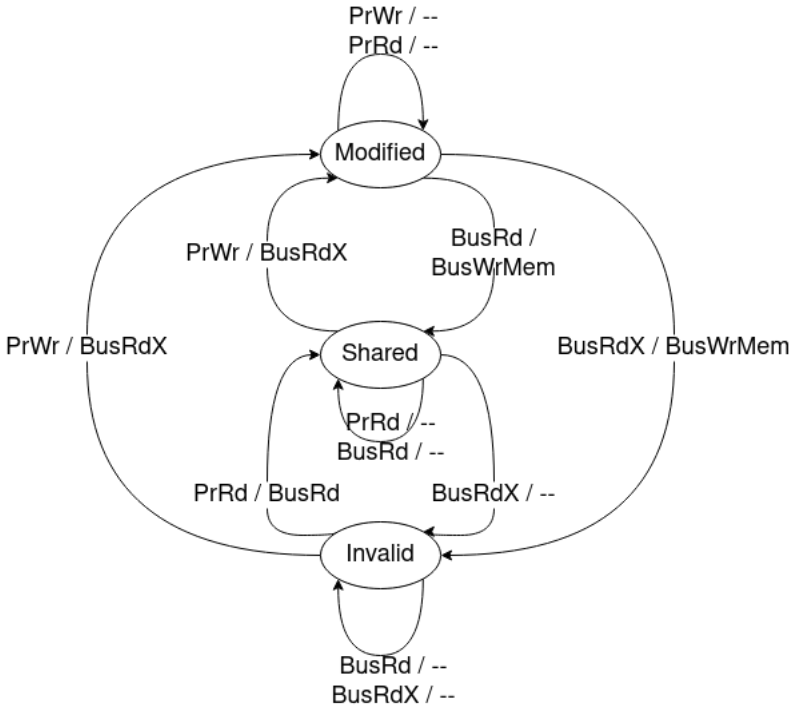


Figure 223: MSI protocol state transition diagram.

a) The typical MSI protocol as shown above follows a write-back and write-allocate policy (i.e., in case of a miss on write, data is written both to the cache and main memory). Perform the necessary modifications to make it follow a write-back and no-write-allocate policy (i.e., in case of a miss on write, data is written only to main memory). You can either redraw the protocol from scratch or apply your changes on the provided template (add missing edges and clearly mark removed edges).

MSI has a couple of shortcomings resulting in redundant traffic on the bus and creating extra unnecessary memory requests:

i If a processor requests to write a cache line, the corresponding cache creates a broadcasting transaction over the bus to signal all other caches to invalidate this cache line, if present. Yet, if the cache line that is about to be modified is not present in other caches, there is no point in such a broadcasting transaction.

ii Whenever a cache encounters a miss on read for a cache line that is dirty in another cache (i.e., in the Modified state), the cache line with the dirty data always writes the data to memory; instead, it could simply pass the data to the requesting cache.

The MOESI protocol addresses the two shortcomings by introducing two additional states **Owned** and **Exclusive**. MOESI replaces, in some cases, the transaction BusWrMem (which writes back a value to memory and whose write-back value can be snooped by other caches) with BusWrCache where the value being written can be snooped by other caches but is not actually written back to memory. MOESI also introduces a BusShr signal which is used by all caches to indicate whether they have a specific piece of data or not; it acts as a wired-OR signal, that is, if more than one cache asserts BusShr at once, then BusShr is active. Figure 224 (given in the next page) illustrates the MOESI protocol.

b) One could classify states in coherence protocols as dirty or clean depending on whether the value of the cache line they correspond to is different (i.e., more recent) from the value in the main memory or is it identical to it. One could also classify them as exclusive or nonexclusive depending on whether the state guarantees that the cache line is definitely not present in any other cache or if it does not give such a guarantee. Specify for the four non-invalid states of MOESI whether they are dirty or clean and exclusive or nonexclusive. Explain clearly each of your eight choices. If convenient, use the provided template.

c) For the states that you have classified as nonexclusive in the previous question, give the simplest sequence of memory accesses that explains your choice. Specify the memory accesses in the following format.

Pn: LD addr or **Pn: ST addr**
where n is the number of the processor (e.g., 1 or 2) and addr is replaced by a particular address. Assume that all caches are empty in the beginning.

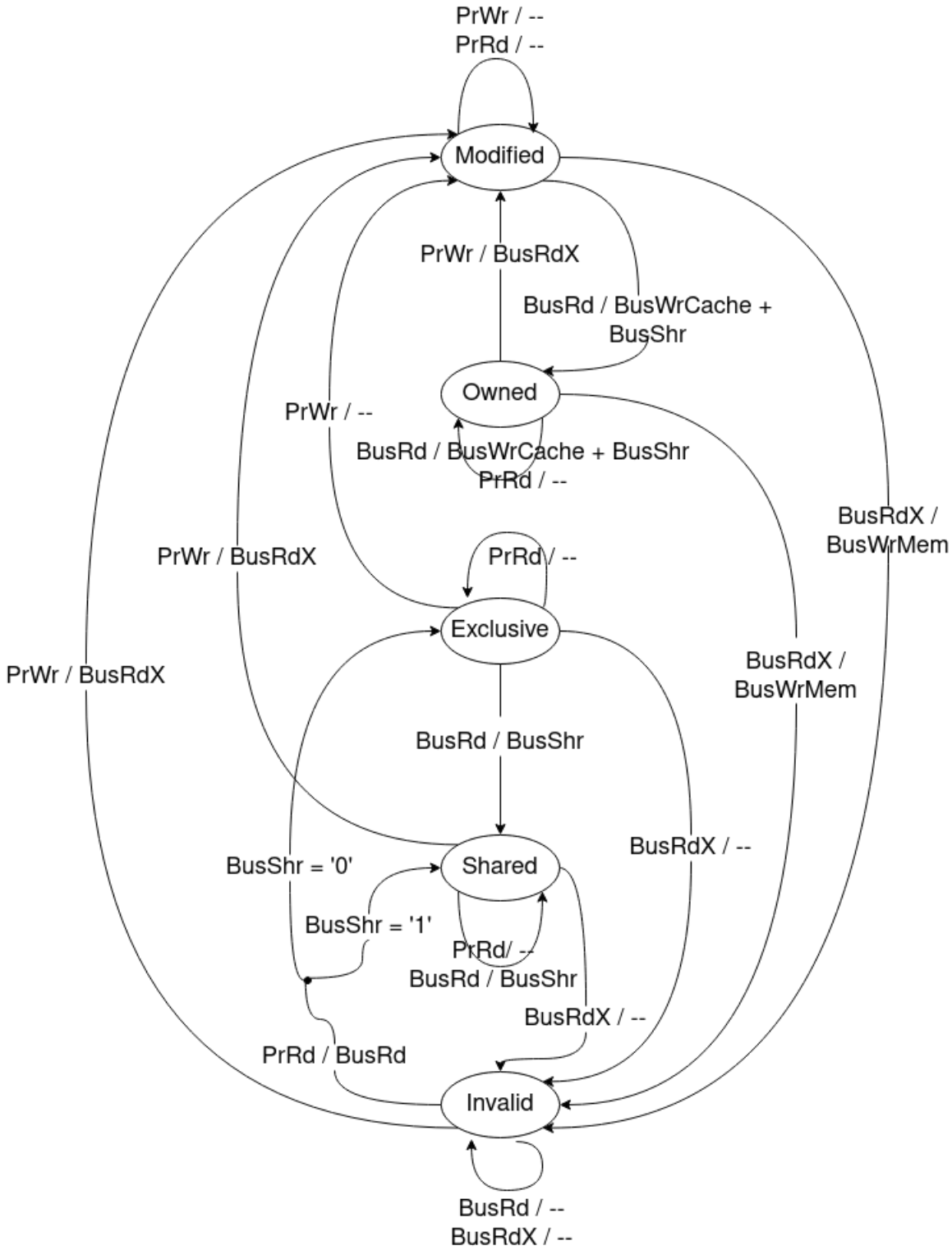


Figure 224: MOESI protocol state transition diagram.

d) Illustrate how MOESI avoids the first MSI shortcoming that is described above in **(i)** with a simple sequence of accesses where the cost in terms of transactions is smaller for MOESI compared to MSI. For simplicity, assume that only transactions involving memory have cost (i.e., BusRd, BusRdX and BusWrMem) and that all other transactions cost nothing. Use the provided template to write all the transactions demonstrating the difference.

e) Illustrate how MOESI avoids the second MSI shortcoming that is described above in **(ii)** with a simple sequence of accesses where the cost in terms of transactions is smaller for MOESI compared to MSI. Use the same template and same assumptions as in question **4)**.

			Reason for Choice
M	Dirty ☐	Clean ☐	
	Exclusive ☐	Non-Exclusive ☐	
O	Dirty ☐	Clean ☐	
	Exclusive ☐	Non-Exclusive ☐	
E	Dirty ☐	Clean ☐	
	Exclusive ☐	Non-Exclusive ☐	
S	Dirty ☐	Clean ☐	
	Exclusive ☐	Non-Exclusive ☐	

			MSI			MOESI		
	Processor #	**Memory Instruction**	**Cache 1 Transition**	**Cache 2 Transition**	**Transaction(s)**	**Cache 1 Transition**	**Cache 2 Transition**	**Transaction(s)**
Example	P1	LD 0x2000	I -> S	--	BusRd	I -> E	--	BusRd
	P2	ST 0x1004	--	I -> M	BusRdX	--	I -> M	BusRdX
4)								
5)								

[Solution 7]

a) No-write-allocate means that upon a miss on write, the data needs to be written directly to memory without writing it to the cache. This means that the state of the cache line will remain Invalid and that there will be two output transactions: BusRdX to invalidate other caches and BusWrMem to do the write to memory. In the following diagram, given in blue transition is the new transactions that need to be present whenever a PrWr comes at an Invalid state. Following this, the transition denoted in red needs to be removed.

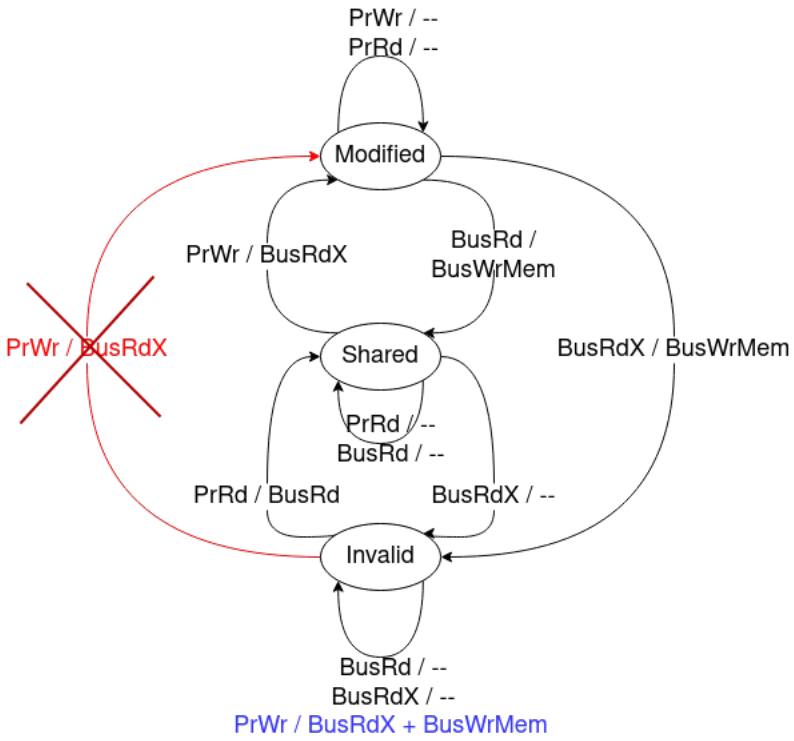


Figure 225: MSI with no-write-allocate state transition diagram.

b) Classification is as follows:

			Reason for Choice
M	**Dirty** ☑	**Clean** ☐	A cache line moves to this state when the processor writes new data to the cache and it doesn't get written to memory.
	Exclusive ☑	**Non-Exclusive** ☐	A cache line is in this state after the processor writes data to this cache and no other cache contains this data. And it exits the state if another cache issues a BusRd or BusRdX on this cache line.
O	**Dirty** ☑	**Clean** ☐	A cache line moves to this state when a processor requests to read a cache line that is in the modified state in another cache (i.e. data is not in memory).
	Exclusive ☐	**Non-Exclusive** ☑	A cache line moves to this state when a processor requests to read a cache line that is in the modified state in another cache, which means that the data will be present in two caches.
E	**Dirty** ☐	**Clean** ☑	A cache line moves to this state at a miss on read; when the processor requests to read data that is not yet present in any cache and is read from memory.
	Exclusive ☑	**Non-Exclusive** ☐	A cache line is in this state only if BusShr is 0 which means that no other cache contains this cache line. And it exits the state if another cache issues a BusRd or BusRdX on this cache line.
S	**Dirty** ☐	**Clean** ☑	A cache line is in this state if more than one cache requested to read this data from memory, so it's clean data.
	Exclusive ☐	**Non-Exclusive** ☑	A cache line is in this state when BusShr is 1 which means that other caches contain this cache line.

Figure 226: Classification of MSI states.

c) S: Two different processors requesting to read from the same memory address.
P1: LD 0x1000
P2: LD 0x1000
The first read by P1 will move C1′s cache line from Invalid to Exclusive and the read by P2 will move C2′s cache line from Invalid to Shared and C1′s cache line from Exclusive to Shared. Both caches now contain the same cache line.

O: One processor writes to a memory address and the second processor reads from the same memory address.
P1: ST 0x1000
P2: LD 0x1000
The write will move C1′s cache line from Invalid to Modified. The read will move C2′s cache line from Invalid to Shared and C1′s cache line from Modified to Owned. Both caches now contain the same cache line.

d) Solution must involve a transition from Exclusive to Modified. See the template at the end.
e) Solution must involve a transition from Modified to Owned. See the template at the end.

			MSI			MOESI		
	Processor #	**Memory Instruction**	**Cache 1 Transition**	**Cache 2 Transition**	**Transaction(s)**	**Cache 1 Transition**	**Cache 2 Transition**	**Transaction(s)**
Example	**P1** **P2**	**LD 0x2000** **ST 0x1004**	**I -> S** **--**	**--** **I -> M**	**BusRd** **BusRdX**	**I -> E** **--**	**--** **I -> M**	**BusRd** **BusRdX**
4)	P1 P1	LD 0x1000 ST 0x1000	I -> S S -> M	-- --	BusRd BusRdX	I -> E E -> M	-- --	BusRd --
5)	P1 P2	ST 0x1000 LD 0x1000	I -> M M -> S	-- I -> S	BusRdX BusWrMem	I -> M M -> O	-- I -> S	BusRdX BusShr + BusWrCache

[Exercise 8] Cache Coherence

Consider a four-processor (P_0 to P_3) system where each processor has its own cache (C_0 to C_3, respectively). The Dragon coherence protocol is used, such that each cache line can be in one of the four states: *Exclusive* (**E**), *Modified* (**M**), *Shared Modified* (**SM**), and *Shared Clean* (**SC**). The transition diagram in Figure 227 shows the transitions from each state depending on the operations performed by the processor (`Pr*`) or induced by the bus (`Bus*`).

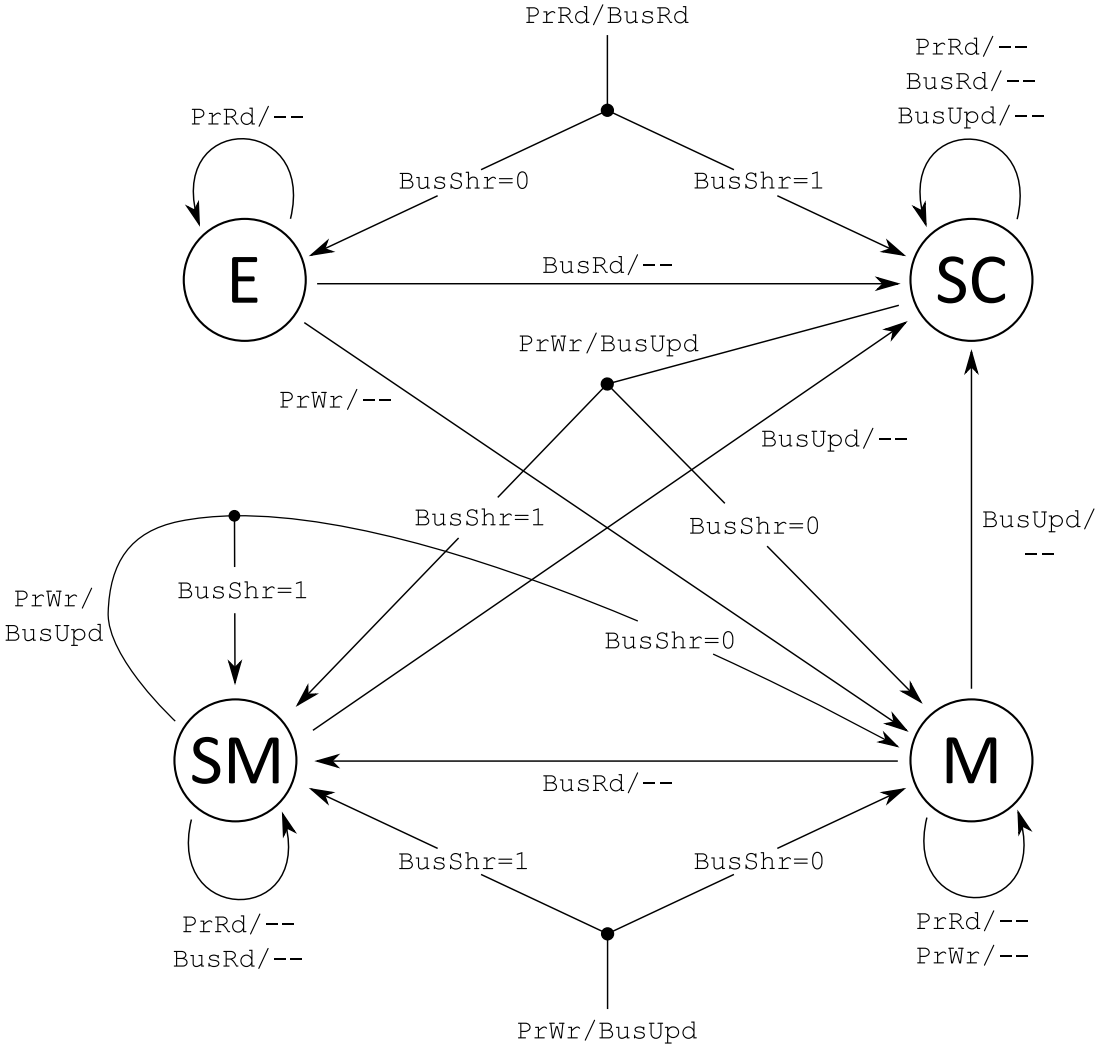


Figure 227: Dragon Cache Coherence Protocol

A cache line in the *Exclusive* state means that this cache has the only copy of the line and that the data present in main memory is correct. A line in the *Modified* state means that the cache has the only copy of the data and that the value stored in main memory is incorrect. The two shared states (*Shared Modified* and *Shared Clean*) indicate that the cache has the correct copy of the line and that the line also exists in other caches. The

last processor to modify the cache line is in the *Shared Modified* state, whereas all the other processors are in the *Shared Clean* state.

Below is a short description of each operation appearing on the transition diagram:

- `PrRd` indicates that the processor reads a cache line.
- `PrWr` indicates that the processor writes to a cache line.
- `BusRd` happens when a processor requests the bus to fetch the latest value of the cache line, whether it's from main memory or another processor's cache.
- `BusUpd` happens when a processor intends to modify a cache line. This allows other processors that also have the line in their cache to update it with the new value. This operation does not update main memory, only other caches. As such it does not generate any memory traffic and is faster than updating main memory.
- `--` indicates that an incoming operation does not produce a bus response in that state.

Additionally, a special bus line (referred to as `BusShr` in the transition diagram) is introduced to detect if a cache line is shared among several caches. For instance, when a cache line is in the *Shared Clean* state, a processor-write (`PrWr`) operation results in a bus update (`BusUpd`). Once the value is written on the bus, the other caches respond by asserting `BusShr` if they have a copy of the same cache line. Then, depending on the value of `BusShr`, the state transition is decided (*Shared Modified* or *Modified*).
The state of a line that is being read/written for the first time into a cache (or the first time after eviction) is defined according to the diagram of Figure 227 by the transitions that have no source state.

a) According to the state transition diagram, are the caches:

- write-through or write-back?
- write-allocate (i.e., in case of write miss, the data is placed in the cache) or write-no-allocate (i.e., in case of write miss, the data is written only to memory)?

Briefly justify both of your answers.

b) Consider the following cache accesses executed in the given order and on the specified processors. Each access is expressed in terms of the cache line such that, for example, `P0: Read 01` means that processor P_0 reads the line 1 of its cache C_0.

```
1  P1: Write 02
2  P0: Read  01
3  P0: Write 01
4  P3: Read  03
5  P1: Read  02
6  P2: Write 02
7  P3: Write 02
8  P3: Write 03
9  P1: Write 03
10 P1: Read  01
11 P2: Read  01
12 P2: Read  03
13 P0: Write 02
14 P3: Read  01
15 P2: Write 03
```

In the provided template, write the state transitions of each cache line in each cache. Assume that all the caches are direct-mapped and initially empty. Make sure that

- Each state is indexed by the instruction that caused the transition. For example, if instruction i causes a transition to the *Modified* state and the cache line was previously not present in the cache, it should be represented as $\rightarrow M_i$.
- A transition from *Exclusive* to *Modified* to *Shared Modified* on instructions i, j, and k, respectively, is represented as $E_i \rightarrow M_j \rightarrow SM_k$.
- If a cache line is in the *Exclusive* state due to instruction i, and then instruction j causes a transition to the same state, the transition is represented as $E_i \rightarrow E_j$.

c) When a cache line is present in more than one cache, the respective coherence state (among **E**, **M**, **SM**, and **SC**) in which it can be in each processor's cache can only take some specific combinations. In the provided template, mark such possible combinations with a checkmark (✓), and all others with a cross ($\times$).

In this protocol, the processor with the cache line in the *Shared Modified* state (or *Modified* state, in the situation where only a single cache has the cache line) is responsible for updating main memory when the cache line is evicted. Conversely, when a cache line in the state *Shared Clean* is evicted, the corresponding processor does not update main memory.

It's possible to simplify the *Dragon* protocol by merging the *Shared Modified* and *Shared Clean* states into a single *Shared* state that becomes responsible for updating main memory when a cache line shared among multiple caches is evicted. The resulting coherence protocol is called Firefly. In this new 3-state protocol, a cache line present in multiple caches at the same time is necessarily in the *Shared* state in all caches that contain it.

d) What is the main benefit of the *Dragon* protocol over *Firefly* with respect to CPU performance? Support your answer by providing a minimal sequence of cache accesses (in the same format as in question 2) that showcases this performance advantage and explains why it leads to higher performance using the *Dragon* protocol.

Dragon is said to be a write-update protocol. In such protocols, when a cache line is modified, the other values of the same line in other caches are updated directly. On the other hand, write-invalidate protocols such as MSI exhibit a significantly different behaviour. In those protocols, when a cache line is modified, the other values of the same line in other caches are not updated, but invalidated. These invalidated lines may cause coherence cache misses down the road if processors whose cache line got invalidated need to access the line again later.

e) The two protocol classes (write-update and write-invalidate) described above have different performance trade-offs. In particular, some memory access patterns are more efficient—in terms of bus and memory traffic—on one protocol than the other. For each of the two classes, describe a memory access pattern (i.e., a sequence of reads and writes to memory performed by a set of processors) that leads to a significantly higher efficiency than on the other class. Explain qualitatively what makes each pattern more efficient on its corresponding protocol class.

Cache Line	Cache C0	Cache C1	Cache C2	Cache C3
01				
02				
03				

Figure 228: Cache transitions

	E	M	SM	SC
E				
M				
SM				
SC				

Figure 229: Cache combinations

[Solution 8]

a)

- On a `PrWr`, the protocol either generates no operation or a `BusUpd` operation that does not update main memory, therefore the caches are write-back. In this protocol, main memory only gets updated on cache line evictions.
- When a `PrWr` occurs on a cache line that is not present in the cache (i.e., on a write miss, represented by the `PrWr` transition with no source state in the diagram), the cache stores the incoming cache line and transitions the line's state to *Modified* or *Shared Modified* without updating main memory, therefore the caches are write-allocate.

b) See Figure 230.

Cache Line	Cache C0	Cache C1	Cache C2	Cache C3
01	$\rightarrow E_2 \rightarrow M_3$ $\rightarrow SM_{10} \rightarrow SM_{11}$ $\rightarrow SM_{14}$	$\rightarrow SC_{10} \rightarrow SC_{11}$ $\rightarrow SC_{14}$	$\rightarrow SC_{11} \rightarrow SC_{14}$	$\rightarrow SC_{14}$
02	$\rightarrow SM_{13}$	$\rightarrow M_1 \rightarrow M_5$ $\rightarrow SC_6 \rightarrow SC_7$ $\rightarrow SC_{13}$	$\rightarrow SM_6 \rightarrow SC_7$ $\rightarrow SC_{13}$	$\rightarrow SM_7 \rightarrow SC_{13}$
03		$\rightarrow SM_9 \rightarrow SM_{12}$ $\rightarrow SC_{15}$	$\rightarrow SC_{12} \rightarrow SM_{15}$	$\rightarrow E_4 \rightarrow M_8$ $\rightarrow SC_9 \rightarrow SC_{12}$ $\rightarrow SC_{15}$

Figure 230: Cache transitions

c) See Figure 231.

	E	M	SM	SC
E	✗	✗	✗	✗
M	✗	✗	✗	✗
SM	✗	✗	✗	✓
SC	✗	✗	✓	✓

Figure 231: Cache combinations

d) Having two separate shared states allows Dragon to designate and keep track of a cache line's unique "owner". Dragon does this by ensuring that, in case a cache line has been modified and is shared across multiple caches, the latest processor to modify is necessarily in the *Shared Modified* state (assuming the line was not evicted yet). Furthermore, the protocol ensures that for any cache line, and at any time, at most one cache is in the *Shared Modified* state. Therefore, the responsibility of updating main memory with a modified line is given solely to the cache in the *Shared Modified* state (i.e., the line's "owner"), which reduces the number of expensive memory updates the system has to perform. On the other hand, Firefly, with its single *Shared* state, cannot keep track of a cache line's "owner", and is forced to have every cache in the *Shared* state update main memory when a line gets evicted, generating more memory updates overall.

Consider a two-processors system (P_0 and P_1) along with their respective caches (C_0 and C_1). Consider the following sequence of cache accesses.

```
1 P0: Write 01
2 P1: Write 01
```

With Firefly, by the end of this sequence, both caches will be in the `Shared` state for cache line `01`. On eviction, both caches will update main memory. With Dragon, however, P_0 will be in state *Shared Clean* and P_1 will be in state *Shared Modified*. On eviction, only P_1 will update main memory. Therefore, even in this minimal example, Dragon performs better than Firefly.

e) Write-update protocols are generally more efficient when a `PrWr` to a cache line shared across multiple caches is followed by multiple `PrRd` to the same line from different processors. In such a situation, write-update protocols update all caches that have the line in response to the `PrWr`. When corresponding processors attempt to read that line (`PrRd`) later, they do not experience a coherence miss or initiate any bus transaction, as the line is already available in their cache. On the other hand, a write-invalidate protocol invalidates the line in all other caches in response to the `PrWr`. Processors attempting to subsequently read the line then need to initiate a bus transaction to get the new value from another cache or main memory, causing more bus/memory traffic than with a write-update protocol.

Write-invalidate protocols are generally more efficient when multiple subsequent `PrWr` to the same cache line by the same processor happen on a line initially shared across multiple processors. In those cases, write-invalidate protocols perform the line invalidation (including the bus transaction it incurs) only on the first `PrWr` and further bus transactions by other processors are avoided. On the contrary, a write-update protocol updates all caches possessing the line on every `PrWr`, even if those caches never read the line again, generating a lot more bus traffic than a write-invalidate protocol would.