CS-200
Computer Architecture

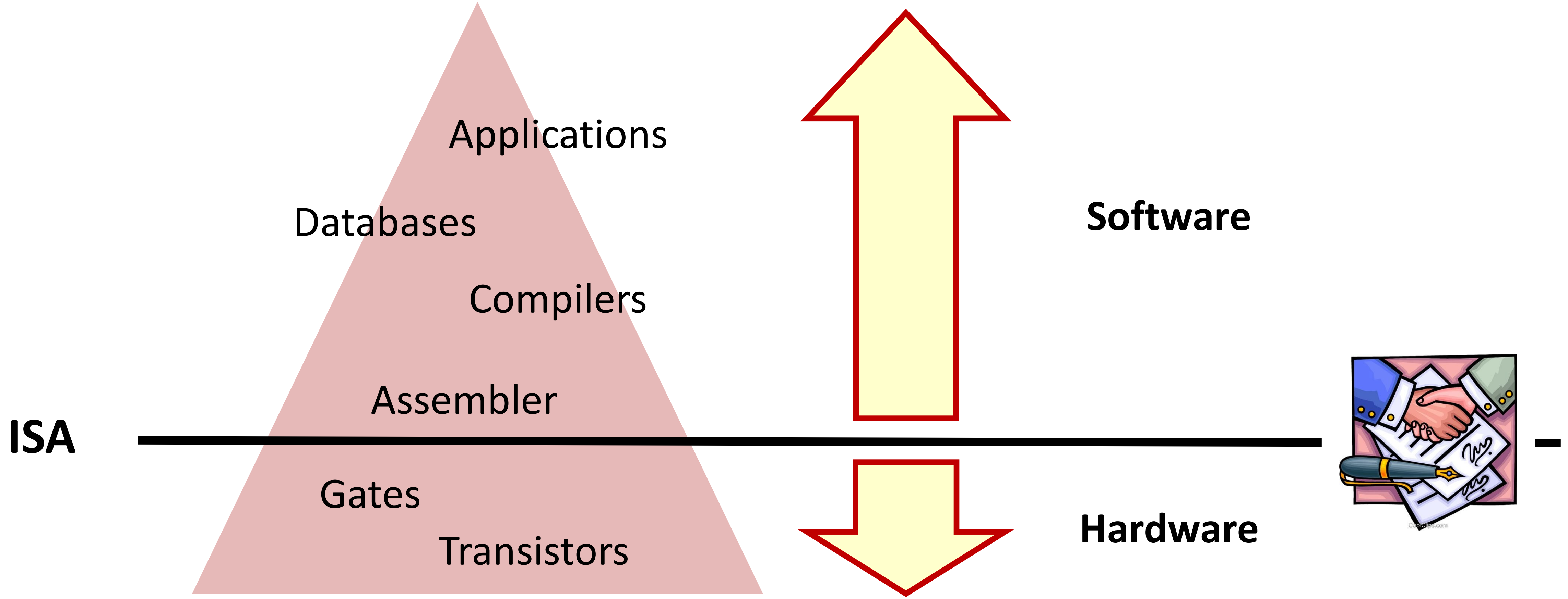
—

Part 1b. Instruction Set Architecture
Branches, Functions, and Stack

Paolo Ienne

<paolo.ienne@epfl.ch>

The Contract between HW and SW



Arithmetic and Logic Instructions Are Easy

True
False

- Two operands

`sll`	`x5, x5, x9`	Shift `x5` left of `x9` positions → `x5`
`add`	`x6, x5, x7`	Add `x5` and `x7` → `x6`
`xor`	`x6, x6, x8`	Logic XOR bitwise `x6` and `x8` → `x6`
`slt`	`x8, x6, x7`	Set `x8` to **1** if `x6` is lower than `x7`, to **0** otherwise

- One operand and a constant (**12-bit immediate**)

`slli`	`x5, x5, 3`	Shift `x5` left of 3 positions → `x5`
`addi`	`x6, x5, 72`	Add 72 to `x5` → `x6`
`xori`	`x6, x6, -1`	Logic XOR bitwise `x6` and `0xFFFFFFFF` → `x6`
`slti`	`x8, x6, 321`	Set `x8` to **1** if `x6` is lower than `321`, to **0** otherwise

`xori rd,rs1,imm` $\quad rd \leftarrow rs1 \,\hat{}\, \boxed{\mathrm{sext}(imm)}$

0xFFF sign extended
to 0xFFFFFFFF

Constants Must Take ≤ 12 Bits

- The constant (**immediate**) is part of the instruction!

	31 25	24 20	19 15	14 12	11 7	6 0	
R	funct7	rs2	rs1	funct3	rd	opcode	Register-Register
I	imm[11:0]		rs1	funct3	rd	opcode	Register-Immediate
I	funct7	imm[4:0]	rs1	funct3	rd	opcode	Register-Immediate Shift
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Store
B	imm[12—10:5]	rs2	rs1	funct3	imm[4:1—11]	opcode	Branch
U	imm[31:12]				rd	opcode	Upper Immediate
J	imm[20—10:1—11—19:12]				rd	opcode	Jump

- To use larger constants, one needs to go through a register

~~`xori  x6, x6, 0x12345678`~~

```
lui    x5, 0x12345
addiu  x5, x5, 0x678
xor    x6, x6, x5
```

Assembler Directives

- The assembler can help to have more readable code

```
lui     x5, 0x12345
addiu   x5, x5, 0x678
xor     x6, x6, x5
```

```
.equ    something, 0x12345678

lui     x5, %hi(something)
addiu   x5, x5, %lo(something)
xor     x6, x6, x5
```

Directive	Effect
`.text`	Store subsequent instructions at next available address in *text* segment
`.data`	Store subsequent items at next available address in *data* segment
`.asciiz`	Store string followed by null-terminator in `.data` segment
`.byte`	Store listed values as 8-bit bytes
`.word`	Store listed values as 32-bit words
`.equ`	Define constants

A Strange Register: `x0`

- The register `x0` does not behave like all others:
  - Register `x0` is always zero
  - One can write into `x0` but this has no effect—because it is always zero

- Both aspects are handy in many situations
  - Zero is a very common, useful value—e.g., set `x5` ← `-x5`

```
sub x5, x0, x5
```

  - If one does not care of the result of an instruction—e.g., no operation

```
add x0, x0, x0
```

Pseudoinstructions

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero

`li` should probably be called `mvi` but is not for historical reasons

```
li     x5, 0x123
addiu  x5, x0, 0x123
```

```
li     x5, 0x12345678
lui    x5, 0x12345
addiu  x5, x5, 0x678
```

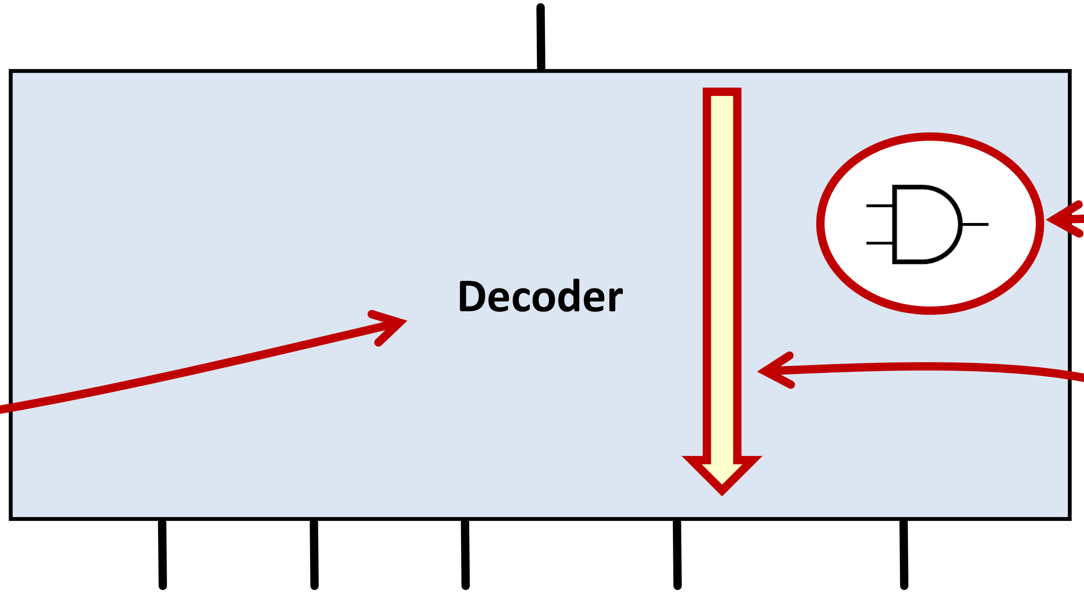
Why Pseudoinstructions?

“and x5, x1, x3”

0000 0000 0010 0000 1000 0011 0010 0011

We want to minimize hardware resources…

Make this simple!



…and we want to spend minimal time decoding!

01001	1	00001	00011	0110011
“x5”	1	“x1”	“x3”	“and”

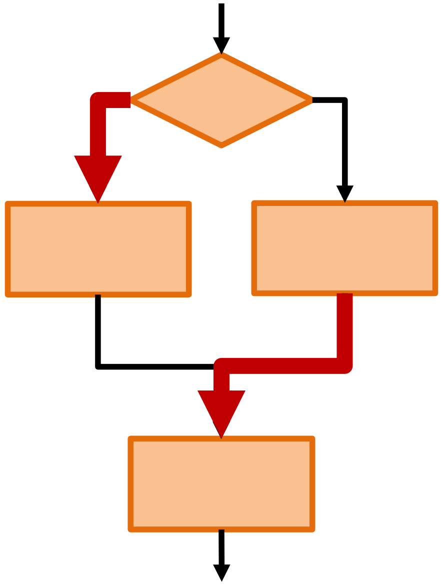
Control Flow (or Control Transfer)

	value
x0	0
x1	0x00123456
x2	0
x3	…
…	
x30	…
x31	…

```
       li   x1, 0x00123456
       li   x2, 0
       li   x3, 1
       li   x4, 0
       li   x5, 0
       li   x6, 32
loop:  and  x5, x1, x3
       add  x2, x2, x5
       srli x1, x1, 1
       addi x4, x4, 1
       bne  x4, x6, loop
```

Only one form of control flow:
branch/jump to a particular instruction

An IF-THEN-ELSE



```
if (x5 == 72) {
   x6 = x6 + 1;
} else {
   x6 = x6 - 1;
}
…
```

Branch if equal

```
.text

    li x7, 72
    beq x5, x7, then_clause

else_clause:
    addi x6, x6, -1
    j end_if

then_clause:
    addi x6, x6, 1

end_if:
    …
```

Jump back to the rest of the code

B	imm[12—10:5]	rs2	rs1	funct3	imm[4:1—11]	opcode	Branch

beqi does not exist

(no space in the encoding for an immediate)

Jumps and Branches

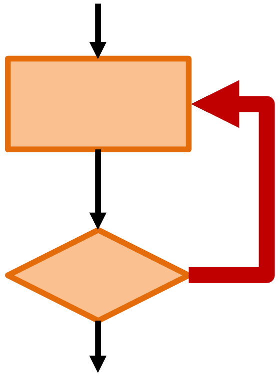
- A common but far from universal distinction
  - **Jumps** → **unconditional** control transfer instructions
  - **Branches** → **conditional** control transfer instructions
- This is the RISC-V convention (inherited from MIPS and used by SPARC, Alpha, etc.)
- Other processors do not. In **x86**, for instance, everything is a jump: JMP, JZ, JC, JNO

Pseudoinstructions

Pseudoinstruction	Base Instruction(s)	Meaning
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned

The processor implements only < and ≥
and the assembler "creates" ≤ and >

An DO-WHILE Loop



```
do {
  x5 = x5 >> 1;
  x6 = x6 + 1;
} while (x5 != 0);
…
```

```
.text

loop:
    srli x5, x5, 1
    addi x6, x6, 1
    bnez x5, loop
…
```

Functions

- Our high-level code is conveniently organized in **functions** (a.k.a. **routines**, **subroutines**, **procedures**, **methods**, etc.)
- Reuse and modularity!

```
int main() {
  …
  b = sqrt(a);
  …
  d = sqrt(c);
  …
}
```

```
int sqrt(int n) {
  int r;
  …
  return r;
}
```

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

A Too Simple (i.e., Not Working) Approach

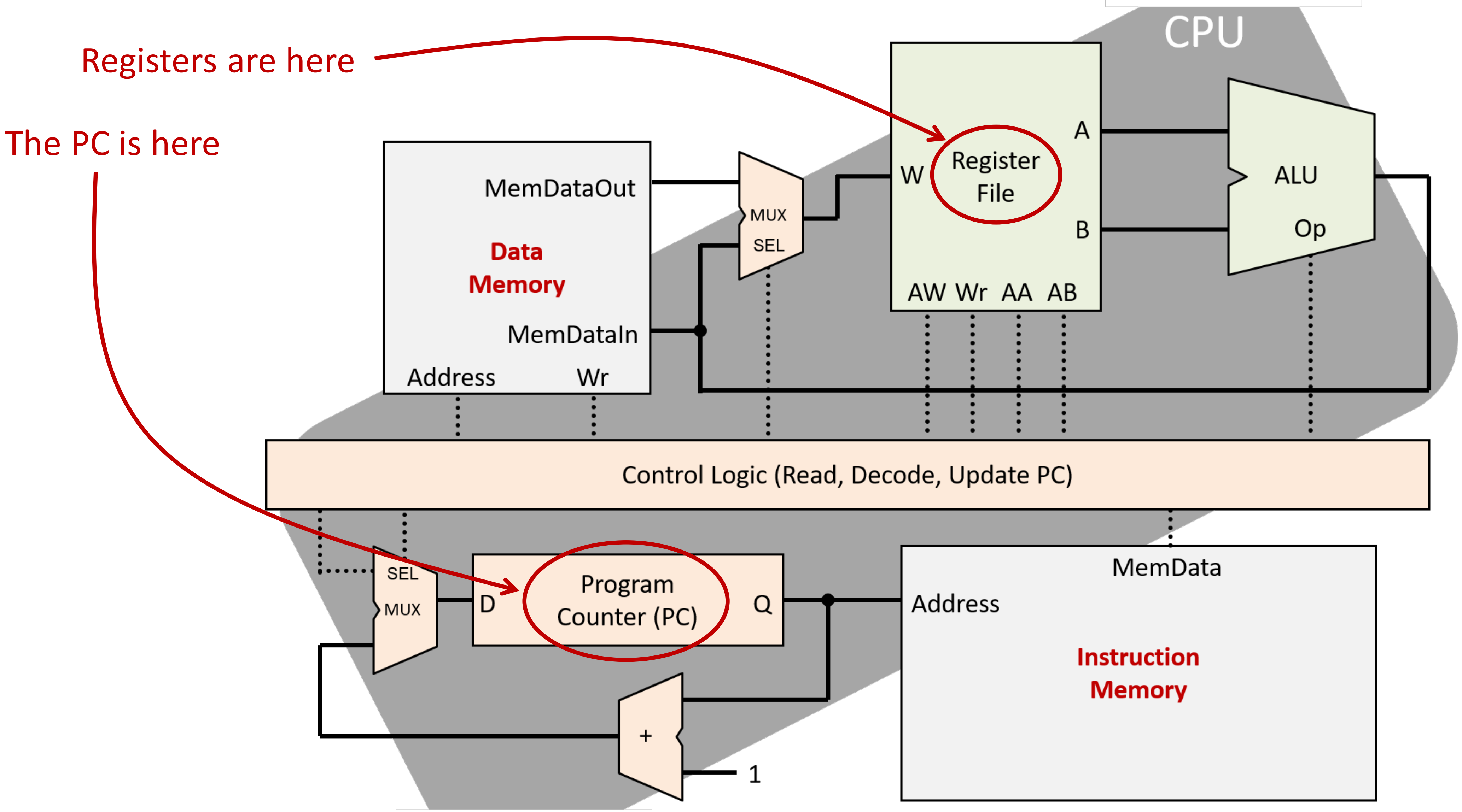
```
main:
   …
   j    sqrt
back:
   …
   j    sqrt
back2:
   …
```

```
sqrt:
      …
      j    back
```

We need to know the value of the
Program Counter (PC) when the call takes place

We need to use **that information** when jumping back

The PC Is Not a Normal Register



The Good Approach

Jump and **link**,
that is, leave PC + 4 into x1
as a "link" to the caller

```
main:
1200    …
1204    jal  x1, sqrt
1208    …

2416    …
2420    jal  x1, sqrt
2424    …
```

```
sqrt:
3648        …

4896        …
4900    jr   x1
```

Jump to the address
specified in a register

x1 ← 1208

The Good Approach

Jump and **link**,
that is, leave PC + 4 into x1
as a “link” to the caller

```
        main:
1200        …
1204        jal   x1, sqrt
1208        …

2416        …
2420        jal   x1, sqrt
2424        …
```

```
        sqrt:
3648                …

4896                …
4900            jr      x1
```

Jump to the address
specified in a register

x1 ← 2424

Only Two Real Jump Instructions!

Jump and link		
`jal`	`rd,imm`	$\texttt{rd} \leftarrow \texttt{pc} + 4$
		$\texttt{pc} \leftarrow \texttt{pc} + \mathrm{sext}(\texttt{imm} \ll 1)$
`jalr`	`rd,rs1,imm`	$\texttt{rd} \leftarrow \texttt{pc} + 4$
		$\texttt{pc} \leftarrow (\texttt{rs1} + \mathrm{sext}(\texttt{imm}))\ \&\ (\sim 1)$

In RISC-V, `imm(reg)` simply means `reg+imm`

Pseudoinstr.	Base Instruction(s)	Meaning
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine

By **convention** RISC-V uses always register `x1` to store the return address

Register Conventions

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No

We will understand
this column later…

The Good Approach

call

in many other ISAs

```
        main:
1200      …
1204      jal  sqrt
1208      …

2416      …
2420      jal  sqrt
2424      …
```

```
        sqrt:
3648          …

4896          …
4900          ret
```

ret

almost universally

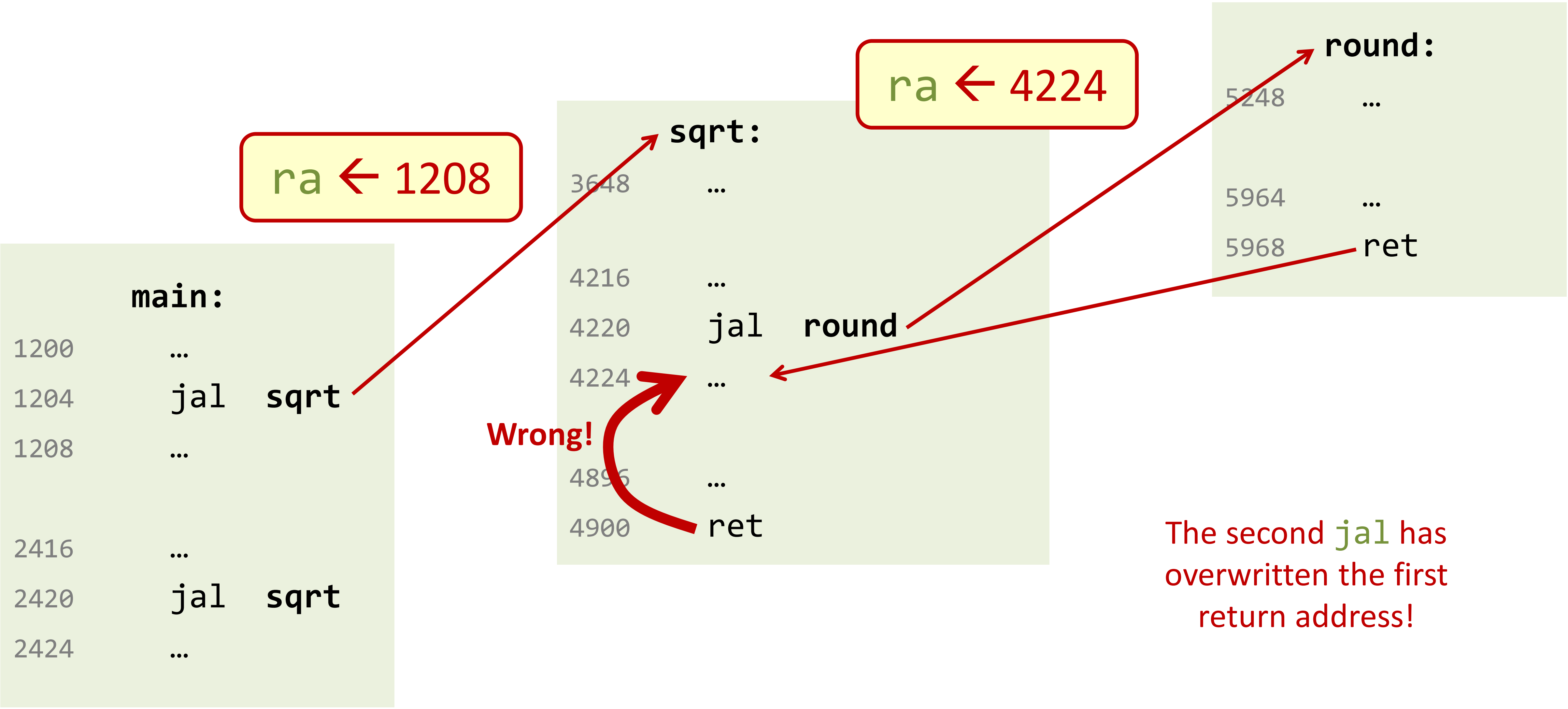
What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program ✓

Calling a Function from a Function



The second `jal` has overwritten the first return address!

Calling a Function from a Function

```
        main:
1200       …
1204       jal   sqrt
1208       …

2416       …
2420       jal   sqrt
2424       …
```

```
        sqrt:
3648       …

4216       add   x5, x7, x8
4220       jal   round
4224       sub   x6, x6, x5

4896       …
4900       ret
```

```
        round:
5248       …
5252       addi  x5, x11, 3

5964       …
5968       ret
```

Who can use x5?!

A Very Very Simple Approach

```
        main:
1200        …
1204        jal   sqrt
1208        …

2416        …
2420        jal   sqrt
2424        …
```

```
        sqrt:
3648        …

4216        add   x5, x7, x8
4220        jal   round
4224        sub   x6, x6, x5

4896        …
4900        ret
```

```
        round:
5248        …
5252        addi x10, x11, 3

5964        …
5968        ret
```

sqrt can
only use x2 to x9

round can
only use x10 to x15

Clearly not scalable…

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

A Simple Approach

```
.data

sqrt_save_ra:    .word 0
sqrt_save_x5:    .word 0
```

Obtain some memory space

```
.text

  sqrt:
    …
    add  x5, x7, x8
    sw   ra, (sqrt_save_ra)
    sw   x5, (sqrt_save_x5)
    jal  round
    lw   ra, (sqrt_save_ra)
    lw   x5, (sqrt_save_x5)
    sub  x6, x6, x5
    …
    ret
```

These load/store instructions with immediate addresses do not exist!

Preserve what we care before calling

Restore after returning

Problem: A Recursive Function!

```
.data

  find_child_save_ra:   .word 0


.text

  find_child:
    …
    sw   ra, (find_child_save_ra)
    jal  find_child
    lw   ra, (find_child_save_ra)
    …
    ret
```

Back to square one!

Every invocation of **find_child** saves its return address in **find_child_save_ra**, overwriting the value of the caller…

What Is the Problem?

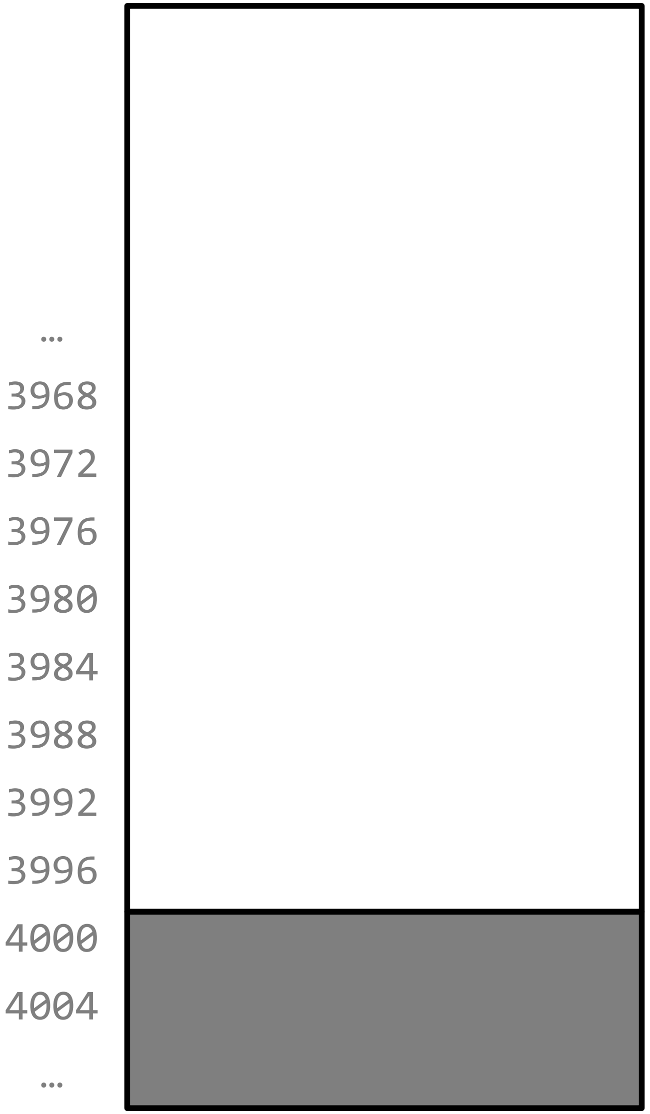
- Static memory allocation vs. dynamic
  - **Static** → fixed at assembly / compile / program writing time
  - **Dynamic** → fixed during execution
- This is static!

```
.data

  find_child_save_ra:    .word 0
```

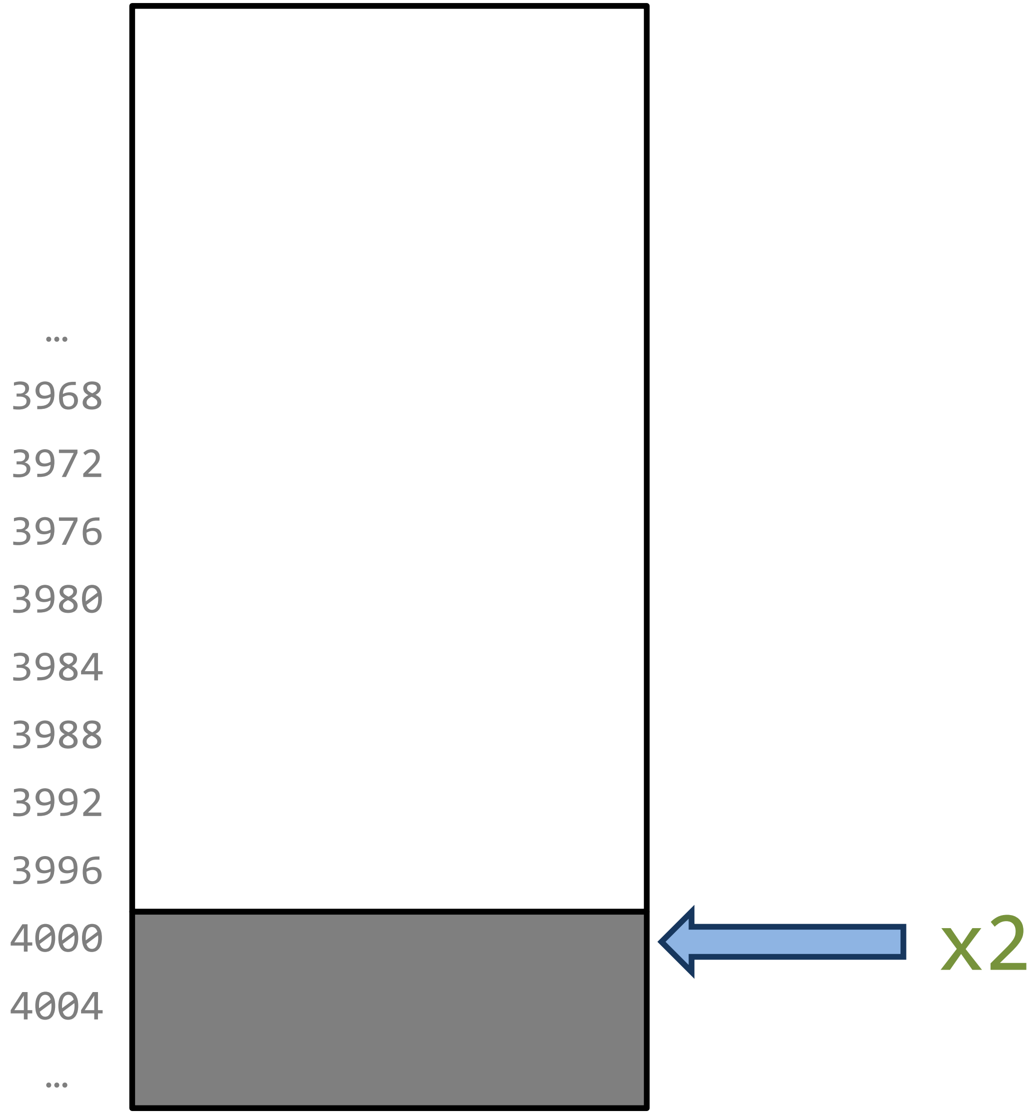
- **Every invocation** of the function needs a **new place** to store the return address and their data

The Idea of a Stack



Reserve statically an **empty** region of memory

The Idea of a Stack

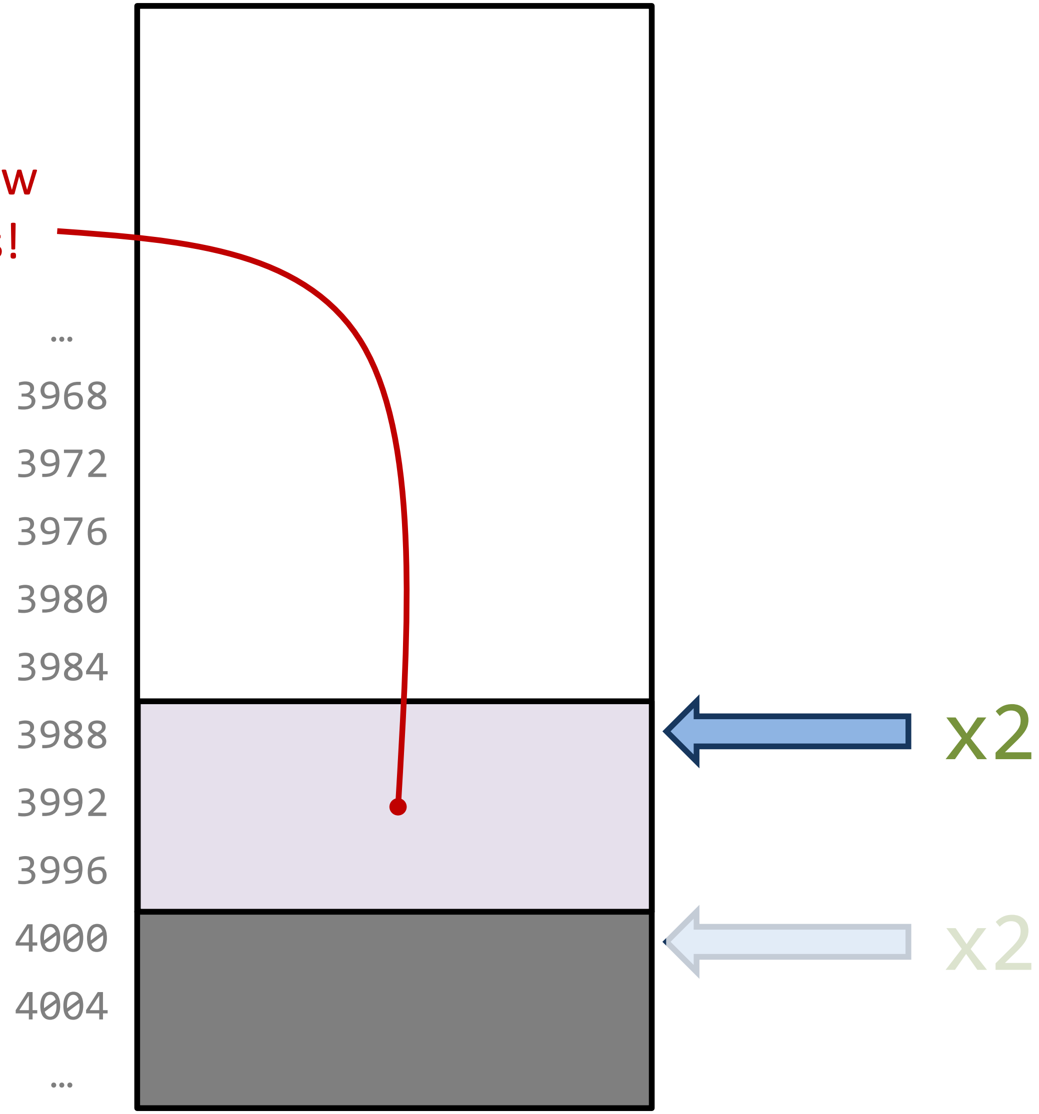


x2 = 4000

Use a register (e.g. x2) to point to the **first used word** after the end of the region

Dynamically Allocating Space (e.g., 3 Words)

This space is now reserved for us!

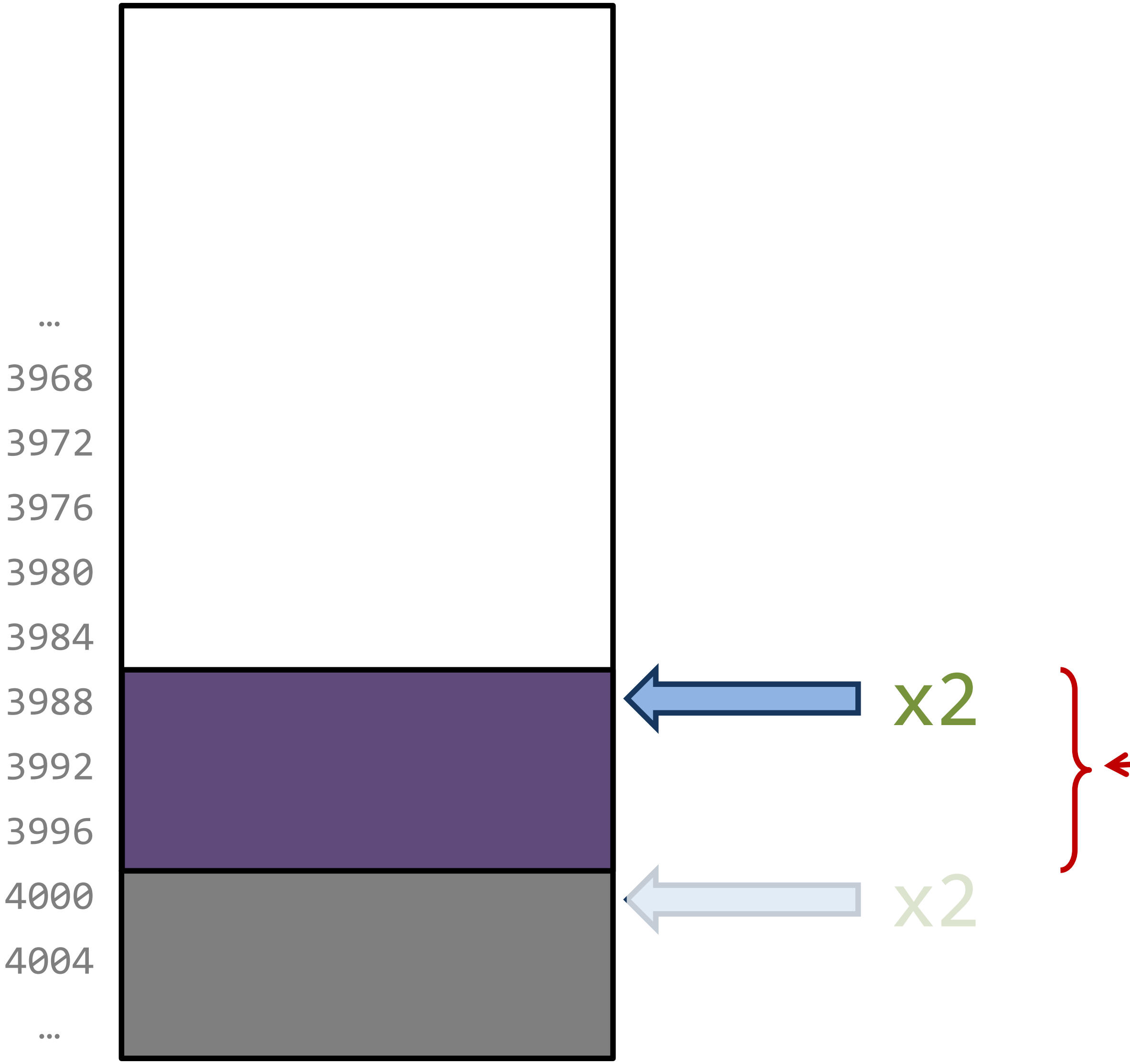


x2 = 3988

3 words of 4 bytes

```
addi x2, x2, -12
```

Using the Space

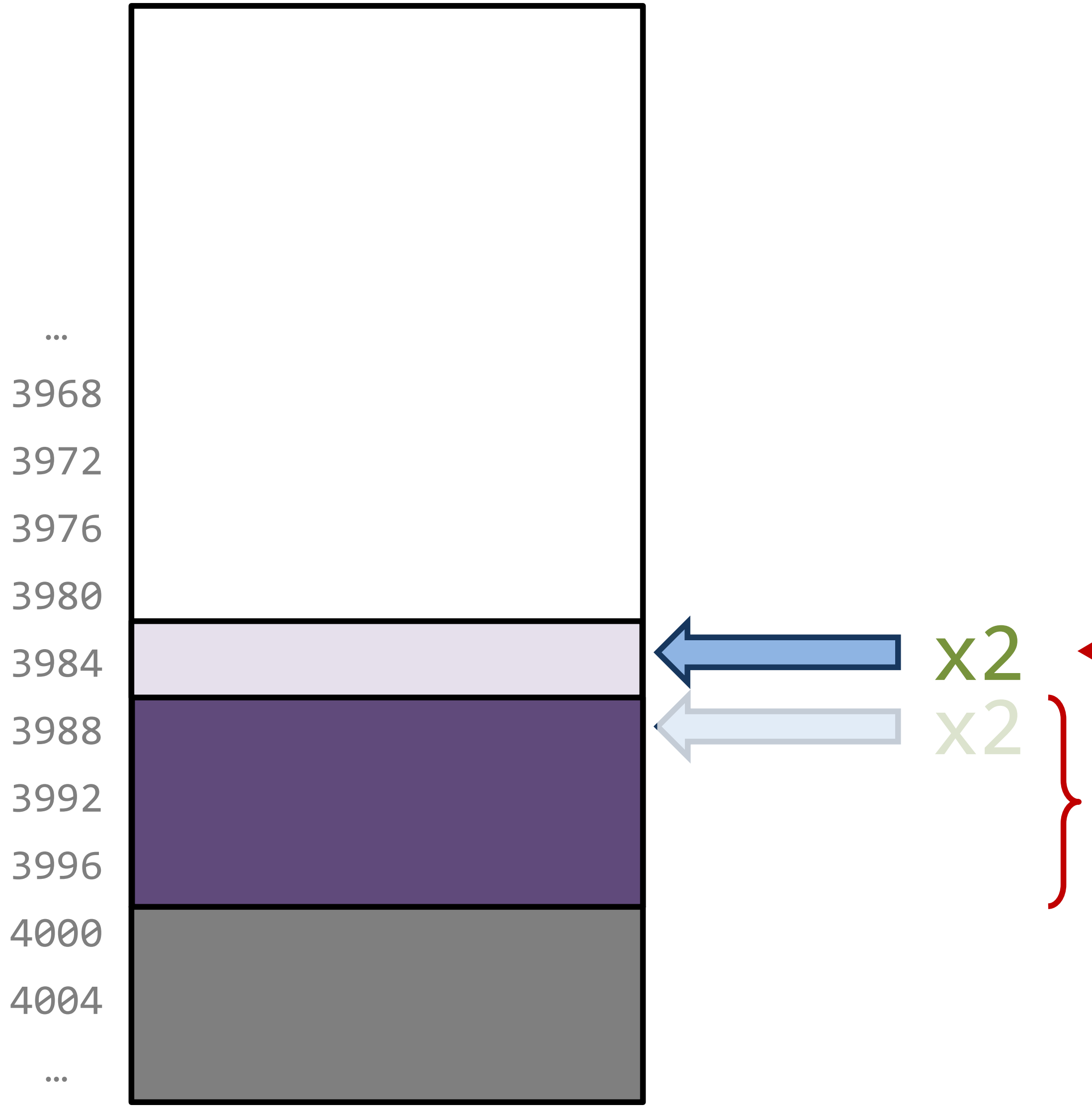


x2 = 3988

```
sw    ra, 0(x2)
sw    x5, 4(x2)
sw    x6, 8(x2)
```

This means "store at the address pointed by the value in x2 plus 8"

Dynamically Allocating More Space

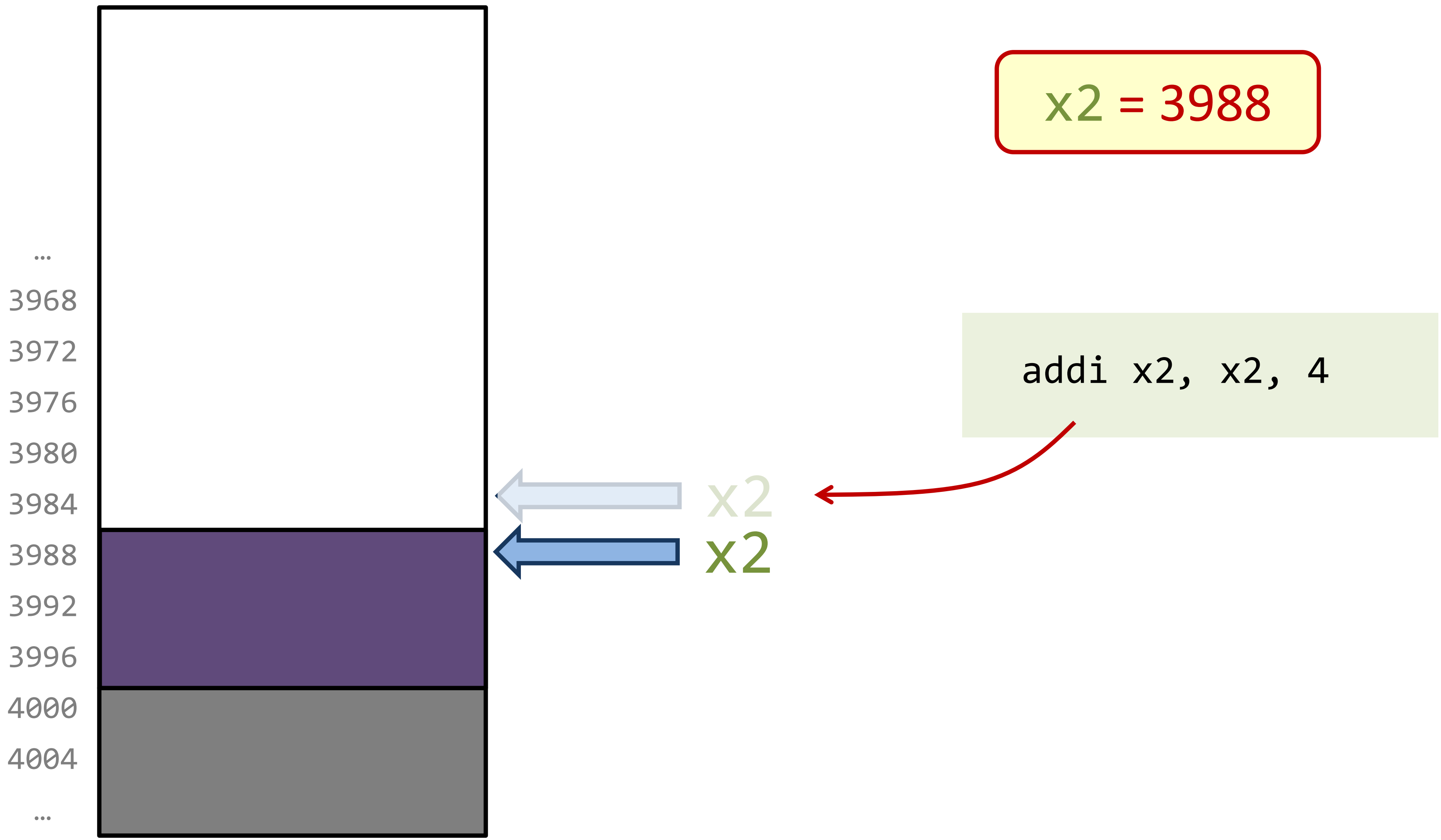


x2 = 3984

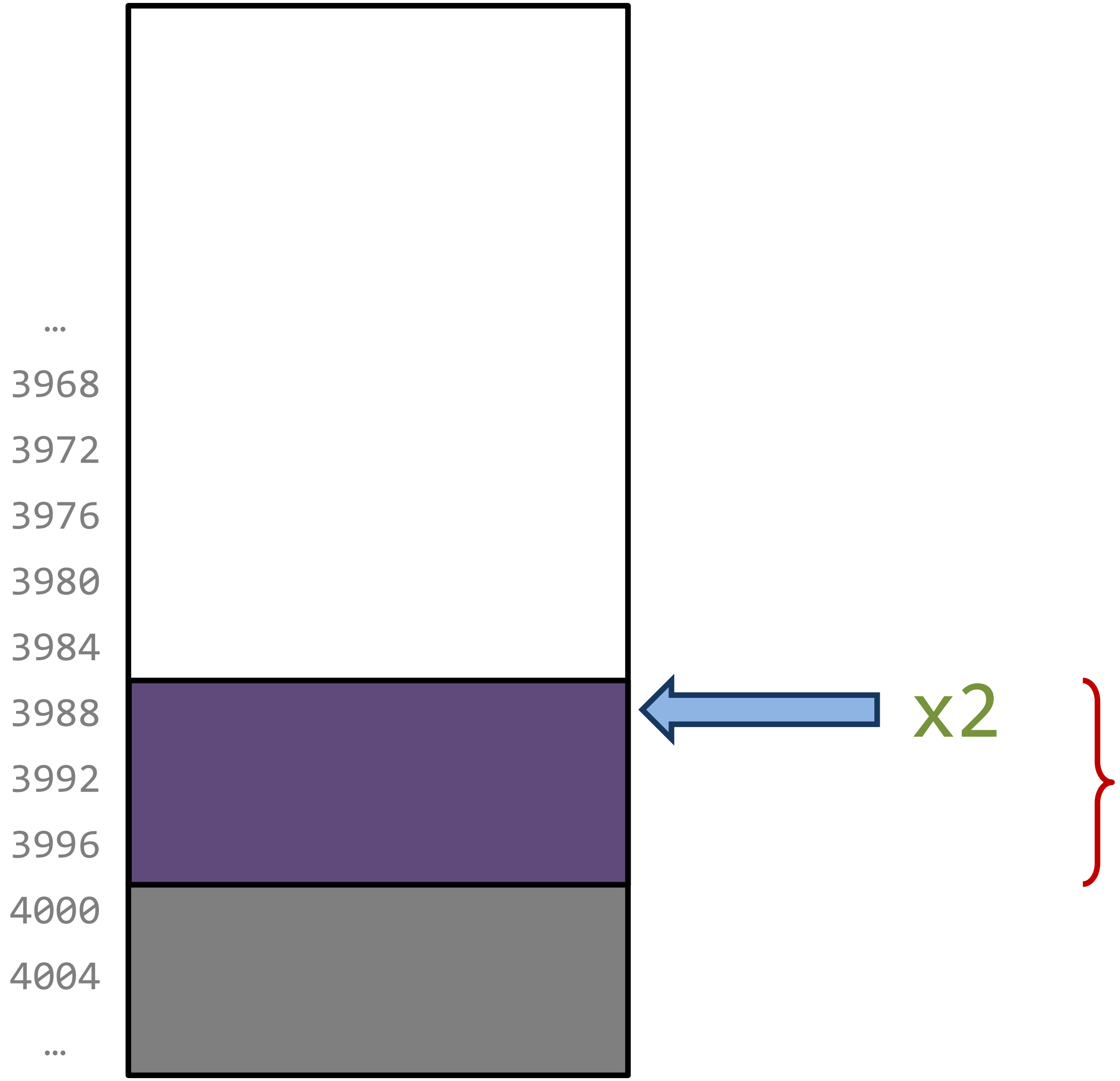
```
addi x2, x2, -4
```

The values we saved are **safe**!

Deallocating Space Is Simple



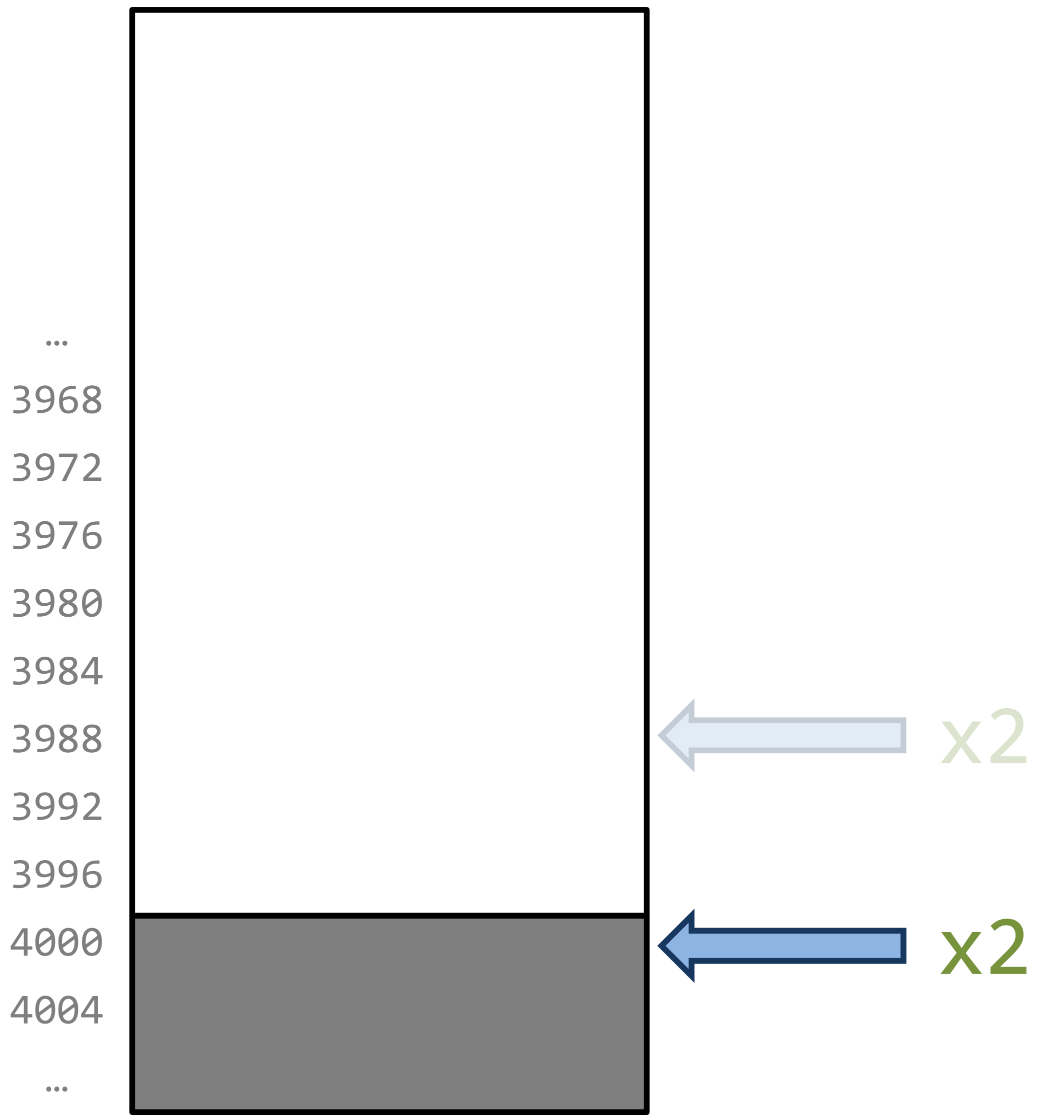
Retrieve Saved Values



x2 = 3988

```
lw    ra, 0(x2)
lw    x5, 4(x2)
lw    x6, 8(x2)
```

More Deallocation



x2 = 4000

```
addi x2, x2, 12
```

Stack: Limited but Effective

- The simplest form of **dynamic memory allocation**
- Simplicity comes with a big limitation
  - **Last-In, First-Out (LIFO)**
  - Deallocation order must be the inverse of the allocation order!
- Yet, **perfectly matched** to our application (**function calls**)
- Other needs will require much more complex approaches that generally need more “management” and need to deal with **garbage collection**

Stack Pointer

- This is so important that we are going to devote a register to this purpose and **everybody** will comply with our **conventions**

Register	ABI Name	Description	Preserved across call?
x2	sp	Stack pointer	Yes

- Other architectures have special instructions to place stuff on the stack (**push**) and to retrieve it (**pop**)

```
PUSH AX
```

```
addi    sp, sp, -4
sw      x5, 0(sp)
```

Spilling Registers to Memory

```
…
addi  sp, sp -8
sw    x8, 0(sp)
sw    x9, 4(sp)
…

      # freely use
      # x8 and x9

…
lw    x9, 4(sp)
lw    x8, 0(sp)
addi  sp, sp, 8
…
```

Whoever needs to free registers, can obtain some space from the stack

```
sqrt:
  addi  sp, sp -4
  sw    ra, 0(sp)
  …
        # freely call
        # other functions
  …
  lw    ra, 0(sp)
  addi  sp, sp, 4
  ret
```

In particular, functions can save their return address (if they call other functions)

Registers across Functions

Functions **change registers** and callers save their stuff

```
…
addi  sp, sp -4
sw    x20, 0(sp)
          # sqrt changes x20
jal   sqrt
lw    x20, 0(sp)
addi  sp, sp, 4
…
```

?

Functions **preserve registers**

```
sqrt:
  addi  sp, sp -4
  sw    x20, 0(sp)
  …
            # uses freely x20
  …
  lw    x20, 0(sp)
  addi  sp, sp, 4
  ret       # x20 is preserved
```

It does not matter (much) provided that **everyone agrees**…

Preserving Registers: The RISC-V Way

- A bit of both

Of course!

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

Saved registers are **callee** saved

Temporary registers are **caller** saved

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

Passing Arguments: Option 1

- Use some particular registers, both for the **arguments** and for the returned **result**
- We can do it **ad-hoc**…
  - `sqrt` gets the argument in `x5` and returns the result in `x6`
- …or we can have some **convention**
  - All functions pass arguments in registers `x10` to `x17` and return the result in `x10`
- Can this be insufficient? **More arguments** than allocated registers? What if we have 10 arguments for `x10` to `x17`?

Passing Arguments: Option 2

```
    …
    addi sp, sp, -4
    sw   s3, 0(sp)        # s3 = arg
    call sqrt
    …

sqrt:

    addi sp, sp, -4
    sw   ra, 0(sp)
    …
    lw   t4, 4(sp)        # t4 = arg
    …
    ret
```

Offset from **sp** changes because **sp** changes

- We can put them on the **stack**
- **Universal** solution (the stack is “unlimited”)
- **More work** than simply using registers, though…
- Many commercial processors do that

Passing Arguments: Option 2 (+ fp)

```
    …
    addi sp, sp, -4
    sw   s3, 0(sp)        # s3 = arg
    call sqrt
    …

sqrt:
    mv   fp, sp
    addi sp, sp, -4
    sw   ra, 0(sp)
    …
    lw   t4, 0(fp)        # t4 = arg
    …
    ret
```

- In addition, one can have another register (**Frame Pointer**; `fp` or `x8` in RISC-V) point to the same location as `sp` on entry
- Code may be more **readable**:
  - `sp` changes inside the function and so do relative offsets
  - Offsets with respect to the `fp` are **fixed**
- Use of `fp` is **optional** and even varies among users and compilers

Passing Arguments: The RISC-V Way

- A bit of both
  - **Some registers reserved** for the arguments and return value(s)

Register	ABI Name	Description	Preserved across call?
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No

  - Rest goes on the **stack**

Summary of RISC-V Register Conventions

Not covered in CS-200

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/~~alternate link register~~	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

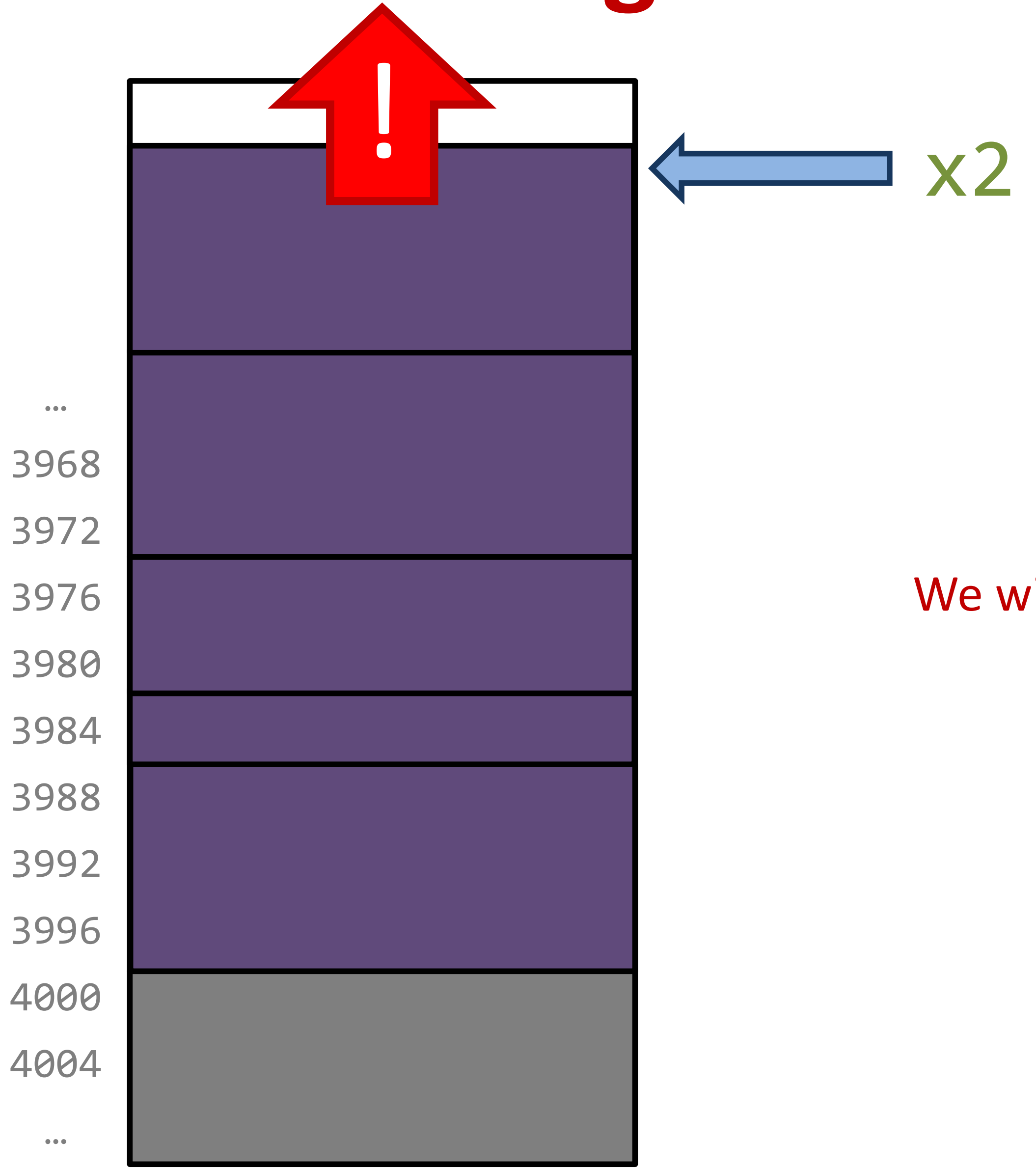
What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

What Calling a Function Involves

1. Place arguments where the called function can access them ✓
2. Jump to the function ✓
3. Acquire storage resources the function needs ✓
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program ✓
6. Release any local storage resources ✓
7. Return control to the calling program ✓

Something to Worry About…



We will definitely need to find a solution!

(In a few weeks' time…)

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 2** and, in particular, **Sections 2.6**, **2.7**, and **2.8**