CS-200
Computer Architecture
—
Part 2d. Processor, I/Os, and Exceptions
Exceptions

Paolo Ienne
<paolo.ienne@epfl.ch>

Exceptions, Interrupts, Faults, Traps, Checks, etc.

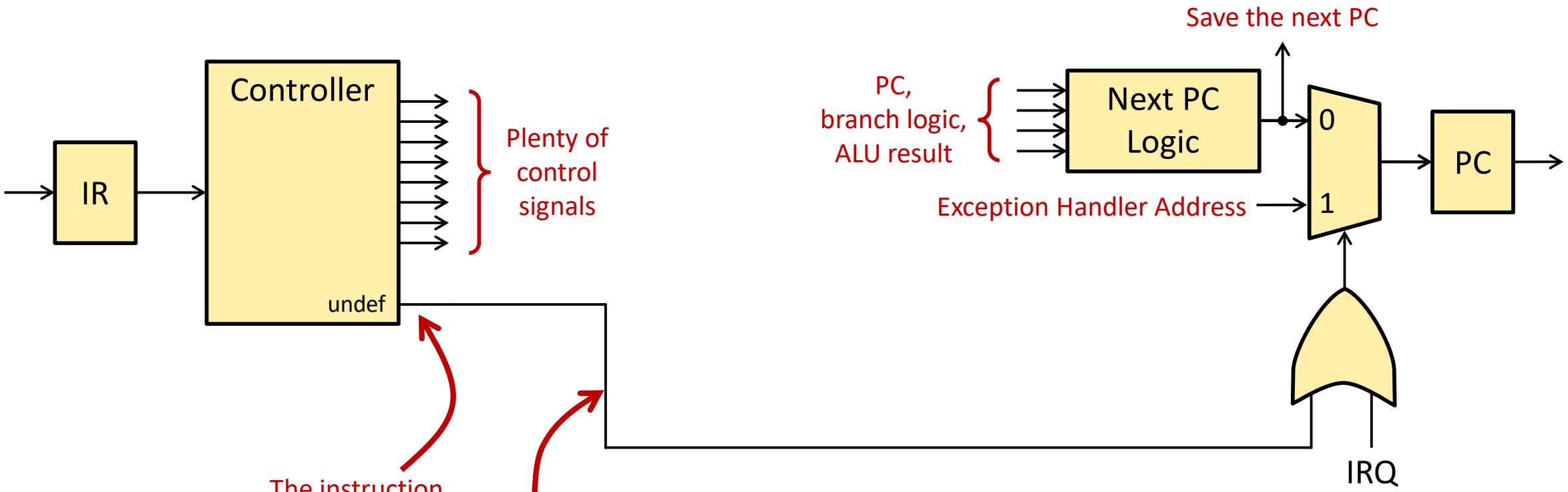
- Normally **control flow** (= sequence of the instructions to be executed) is completely under the control of the **programmer** who specifies jumps, branches, procedure calls, etc.
- "Exceptions" are… the exception!
  - Control flow changes that are not explicit in the program and triggered by **special conditions**
  - **Exception handlers** are special functions that take appropriate actions when an exception arises
  - Example already seen: **I/O Interrupts**

Exceptions, Interrupts, Faults, Traps, Checks, etc.

- Naming varies **widely** across systems!
- Our convention (RISC-V, COD)
  - **Exceptions**: general name for all of them
  - **Interrupts**: special exceptions generated outside of the processor

- Thus, interrupts are the only form of exception that we have met so far

Undefined Instruction



An **exception**!

Undefined Instruction

- But now this new exception is **synchronous**
  - It does not happen at some random point in our code (as an I/O interrupt), but **at a very specific point** (where the undefined instruction is!...)
  - **Synchronous** means that if I run the program again from the very same starting state, the **exception occurs every time**

- We could **serve it** (that is, jump to the handler) at any point in the future (see I/O interrupts) but if we could guarantee to **serve it before the next instruction**, we could do cool stuff…

An Optional fadd.s Instruction

- Suppose we want to have a **FP add instruction**:

```
fadd.s rd, rs1, rs2
```

- Some processors could have a special ALU supporting this instruction; **cheaper processors do not support it**
- What of those that do not? They will trigger an undefined instruction exception leading to the **execution of a handler**
- How can we “handle” things? **Emulate**…

This is **not** the real `fadd.s` instruction of RISC-V but the idea is the same…

Outline of Undefined Instruction Handler

1. **Save** on the stack **all registers** we and our callees will modify
   - Remember: calling conventions do not apply!
2. **Get the instruction** where the problem happened
   - If the PC has been saved somewhere, we can load from that address
3. **Decode in software** the instruction and determine that it is an fadd.s
4. **Read the source registers** (operands) and **call a library** (or write the code) to perform the FP addition
5. **Store the result** in the destination register
6. **Increment the PC** of the failed fadd.s to point to the next instruction
7. **Jump** there to continue execution

Exceptions, Interrupts, Faults, Traps, Checks, etc.

- Why one needs these “exceptions”?
  - **Input/output request**
    - Data are requested or new inputs should be processed
  - **Timer** interrupts
  - Unsupported or **undefined instructions**
    - E.g., missing floating-point support in an implementation
  - **Arithmetic faults**
    - The operation is not defined on specific operands (e.g., division by zero)
  - **Memory** protection **violation**
    - A user tries to read/write data belonging to another user (we will talk at length about this later!)
  - Debugging, **breakpoints**, etc.
  - Hardware **malfunctions**, power failures, etc.

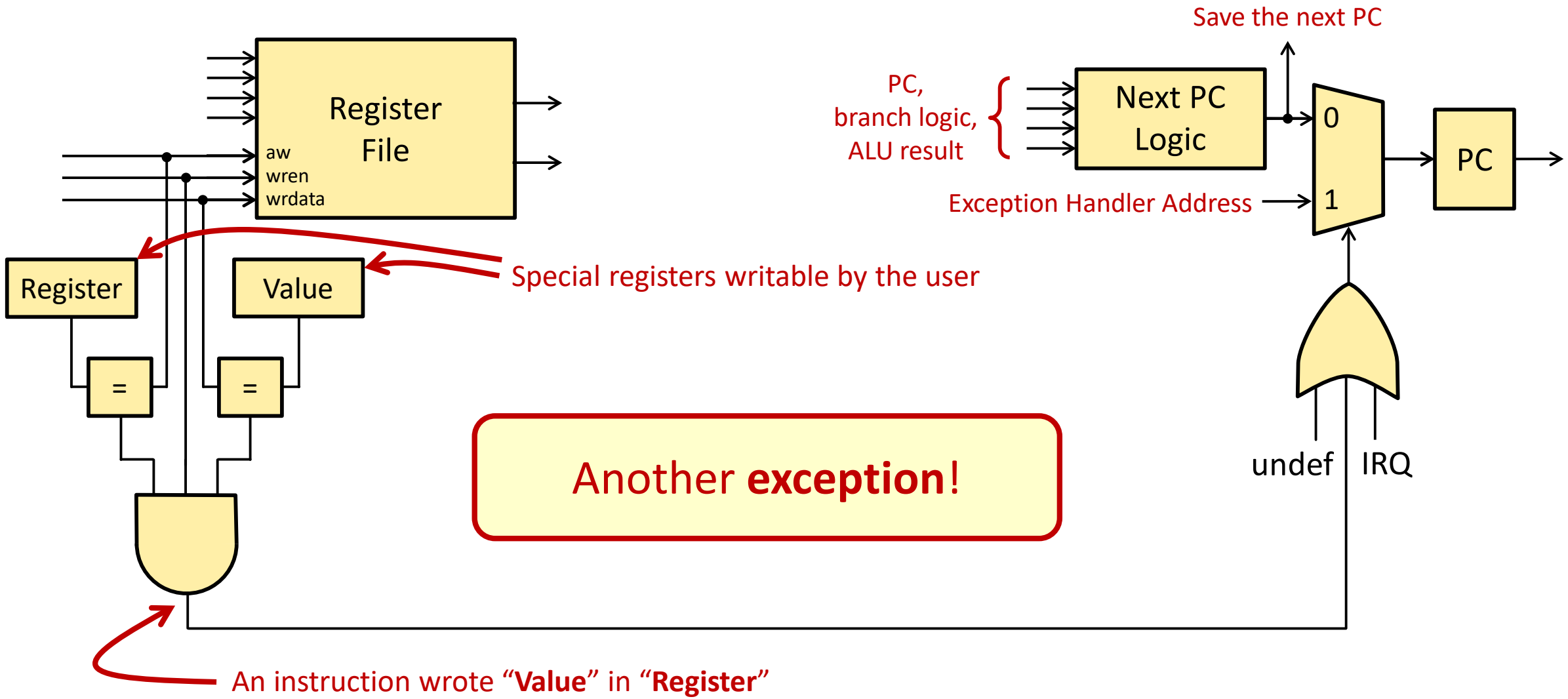
Exceptions, Interrupts, Faults, Traps, Checks, etc.

- Three possible dimensions:
  - **Synchronous** or asynchronous
    - Related to a given instruction with some specific data or external?
  - User requested or **coerced**
    - Software interrupts?
  - **Resume** or terminate
    - Possibility of resuming the execution after exception?
- Some choices are a matter of policy (e.g., OS):
  - Switch off system if power failure
    - Disk state is clean but current work lost
  - “Hibernate” system if power failure
    - Disk state is clean and session is preserved, but is there time?!

A Possible Classification of Exceptions

Type	Synchronous?	Coerced?	Resume?
I/O request	Asynchronous	Coerced	Resume
Invoke OS	Synchronous	User requested	Resume
Trace instruction	Synchronous	User requested	Resume
Breakpoint	Synchronous	User requested	Resume
Page fault	Synchronous	Coerced	Resume
Misaligned access	Synchronous	Coerced	Resume
Memory protection violation	Synchronous	Coerced	Terminate
Bus error	Synchronous	Coerced	Terminate
Arithmetic fault	Synchronous	Coerced	Terminate
Undefined instruction	Synchronous	Coerced	Terminate
Hardware malfunction	Asynchronous	Coerced	Terminate
Power failure	Asynchronous	Coerced	Terminate

Watchpoint


Register File
aw
wren
wrdata
Register
Value
=
=
Special registers writable by the user
Another **exception**!
An instruction wrote "**Value**" in "**Register**"
Save the next PC
PC,
branch logic,
ALU result
Next PC Logic
Exception Handler Address
0
1
PC
undef
IRQ

Raising Exceptions

- The **address of the current or next instruction** must be saved somewhere, but it cannot be in `ra` (or where the ordinary return is saved) → Where?
  - A "(next) PC at the time of the exception" or **Exception PC (EPC) register**
  - We need a **special instruction** or procedure **to return** using the EPC
- There are **many causes** possible now → What happened?
  - One handler address and **cause register**
  - A **vector of handlers**, one per cause
- A **nested interrupt/exception** may appear while handling one → What happens?
  - **Disable** the ones that can be disabled (e.g., interrupts)
  - Avoid those that cannot (e.g., divide by zero or undefined instruction)

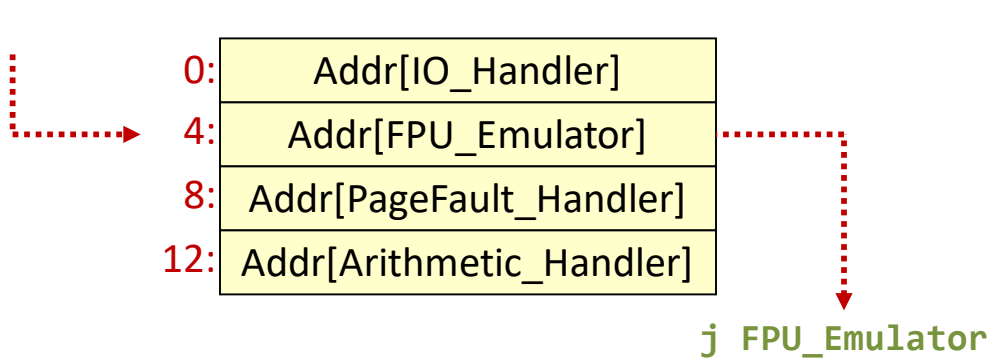
Assessing the Position of Exception

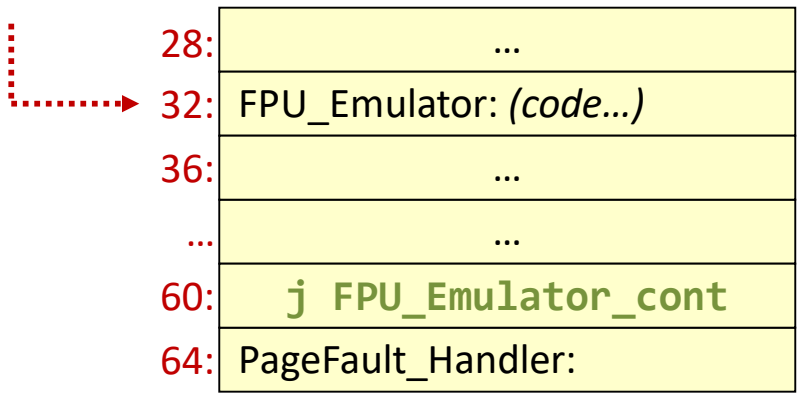
- For asynchronous exceptions
  - What is the **next instruction** to execute?
- For synchronous exceptions
  - What is the **instruction that faulted**?
    - We will restart from that one if we want to correct something and retry (e.g., invoking the OS, see later)
    - We will restart from the next one if we implement the functionality otherwise (e.g., undefined instruction emulated in software)

- Various solutions
  - Reserve a **dedicated ordinary register** for EPC (e.g., Nios II)
  - Place EPC on the **stack** (e.g., x86)
  - Have a set of **special registers dedicated to exception handling** and special instructions to access them (e.g., MIPS, RISC-V)

Assessing the Cause of Exception

- How is the control passed to the handler?
  - Single handler with **cause register** (e.g., RISC-V, MIPS)
    - Processor executes a jump to a fixed address
    - Handler does the dispatching in software by reading a special **cause register**
  - **Vector of handler addresses** (e.g., RISC-V, 68k)
    - Processor executes a jump to `mem[`*Exception Vector Address* + (4 × *Exception Number*)`]`
  - **Vector of handlers** (e.g., PA-RISC 2.0, SPARC)
    - Processor executes a jump to *Exception Vector Address* + (32 × *Exception Number*)





RISC-V Machine-Mode Exception Handling

Control and Status Registers (CSRs)

Name	31	30 12	11	10 8	7	6 4	3	2	1 0
mstatus	*reserved*				MPIE	*reserved*	MIE	*reserved*	
mie	*reserved*		MEIE	*reserved*	MTIE	*reserved*			
mip	*reserved*		MEIP	*reserved*	MTIP	*reserved*			
mtvec	BASE								MODE
mepc	MEPC								
mcause	Interrupt	Exception Code							

Instructions to access the special registers

A special **ret**

Instruction	Pseudocode	Meaning
csrrw rd, csr, rs1	`rd` ← `csr`; `csr` ← `rs1`	Read/Write CSR
csrrs rd, csr, rs1	`rd` ← `csr`; `csr` ← `csr` | `rs1`	Read/Set Bits CSR
csrrc rd, csr, rs1	`rd` ← `csr`; `csr` ← `csr` & (∼`rs1`)	Read/Clear Bits CSR
csrrwi rd, csr, imm	`rd` ← `csr`; `csr` ← zext(`uimm`)	Read/Write CSR immediate
csrrsi rd, csr, imm	`rd` ← `csr`; `csr` ← `csr` | zext(`uimm`)	Read/Set Bits CSR immediate
csrrci rd, csr, imm	`rd` ← `csr`; `csr` ← `csr` & (∼ zext(`uimm`))	Read/Clear Bits CSR immediate

mret	`mstatus.MIE` ← `mstatus.MPIE`
	`mstatus.MPIE` ← 1
	`pc` ← `mepc`
		Return from M-Mode

RISC-V Machine-Mode Exception Handling

Pseudoinstructions:

`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate

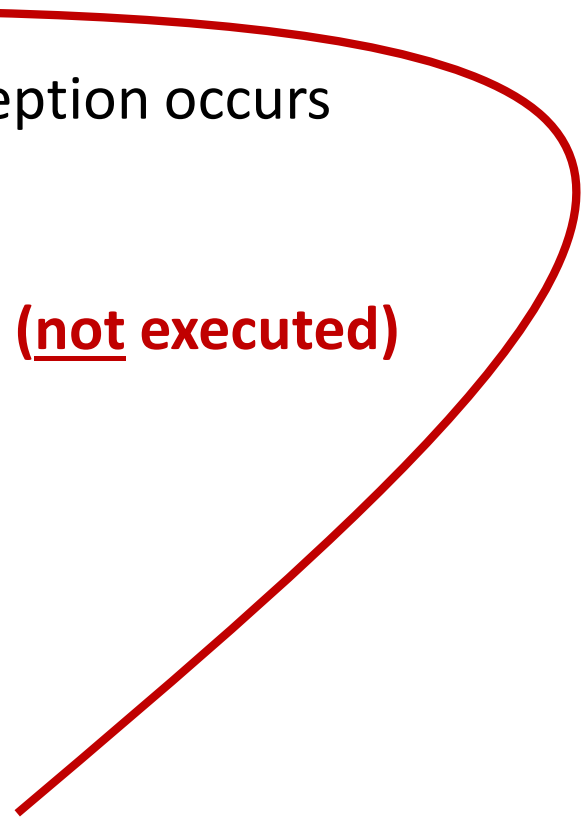
Simply read and write **`csr`**

RISC-V Machine-Mode Exception Handling

- `mtvec` (Machine Trap Vector)
  → holds the **address the processor jumps to** when an exception occurs

- `mepc` (Machine Exception PC)
  → points to the **instruction where the exception occurred (not executed)**

- `mcause` (Machine Exception Cause)
  → indicates **which exception occurred**



Often just a **constant value** to jump to,
but it could implement vectored interrupts

RISC-V Interrupt and Exception Codes

I/O Interrupts

Memory management
(we will come back to these)

Interrupt / Exception `mcause[31]`	Exception Code `mcause[30..0]`	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

Possible Undefined Instruction Handler

```
handler:  addi  sp, sp, -128        # Save all registers but sp (leaving space in memory so that it is a regular array)
          sw    x0, 0(sp)
          sw    x1, 4(sp)
          sw    x3, 12(sp)
             … etc. …
          sw    x31, 124(sp)

          csrr  a1, mcause          # Read exception cause
          bltz  a1, interrupt       # Branch if not an exception (MSB = 1, looks like a negative number…)
          li    a2, 2               # a2 = illegal instruction cause
          bne   a1, a2, otherExcp   # Branch if not an illegal instruction exception

          csrr  t0, mepc            # Load the current faulting instruction address
          lw    a0, 0(t0)           # Read the faulty instruction
          jal   decodeInst          # Gets a0 = instruction; returns a0 = 0 if it cannot be emulated or a0 = operation, a1, a2, and a3 = index of rs1, rs2, and rd
          beqz  a0, undefinedInst   # Branch if really undefined and not possible to emulate
          mv    s0, a3              # Save index of rd
             … etc. …               # Read from the stack the content of the original registers pointed by a1 and a2 into a1 and a2 (“read the source registers”)
          jal   emulateInst         # Gets a0 = operation, a1, a2 = operands; returns a0 = FP result
             … etc. …               # Write a0 on the stack into original register pointed by s0 (“write the destination register”)

          lw    x1, 4(sp)           # Restore all registers but zero and sp
             … etc. …
          lw    x31, 124(sp)
          addi  sp, sp, 128

          csrr  t0, mepc            # Load the current faulting instruction address
          addi  t0, t0, 4           # Increment to point to the next instruction (4 bytes for RV32/RV64)
          csrw  mepc, t0            # Write the updated value back to me
          mret
```

Read the exception cause in **mcause**

Read the EPC in **mepc**

Adjust the **mepc** to execute the instruction following the faulty one next

Use **mret** to return

RISC-V Machine-Mode Interrupt Handling

`mie` (Machine Interrupt Enable)

→ lists which **interrupts the processor can take** and which it must ignore

- **MEIE**: Interrupt-enable bit for machine-level external interrupts
- **MTIE**: Interrupt-enable bit for machine timer interrupts

`mip` (Machine Interrupt Pending)

→ lists the **interrupts currently pending**

- **MEIP**: Interrupt-pending bit for machine-level external interrupts
- **MTIP**: Interrupt-pending bit for machine timer interrupts

`mstatus` (Machine Status)

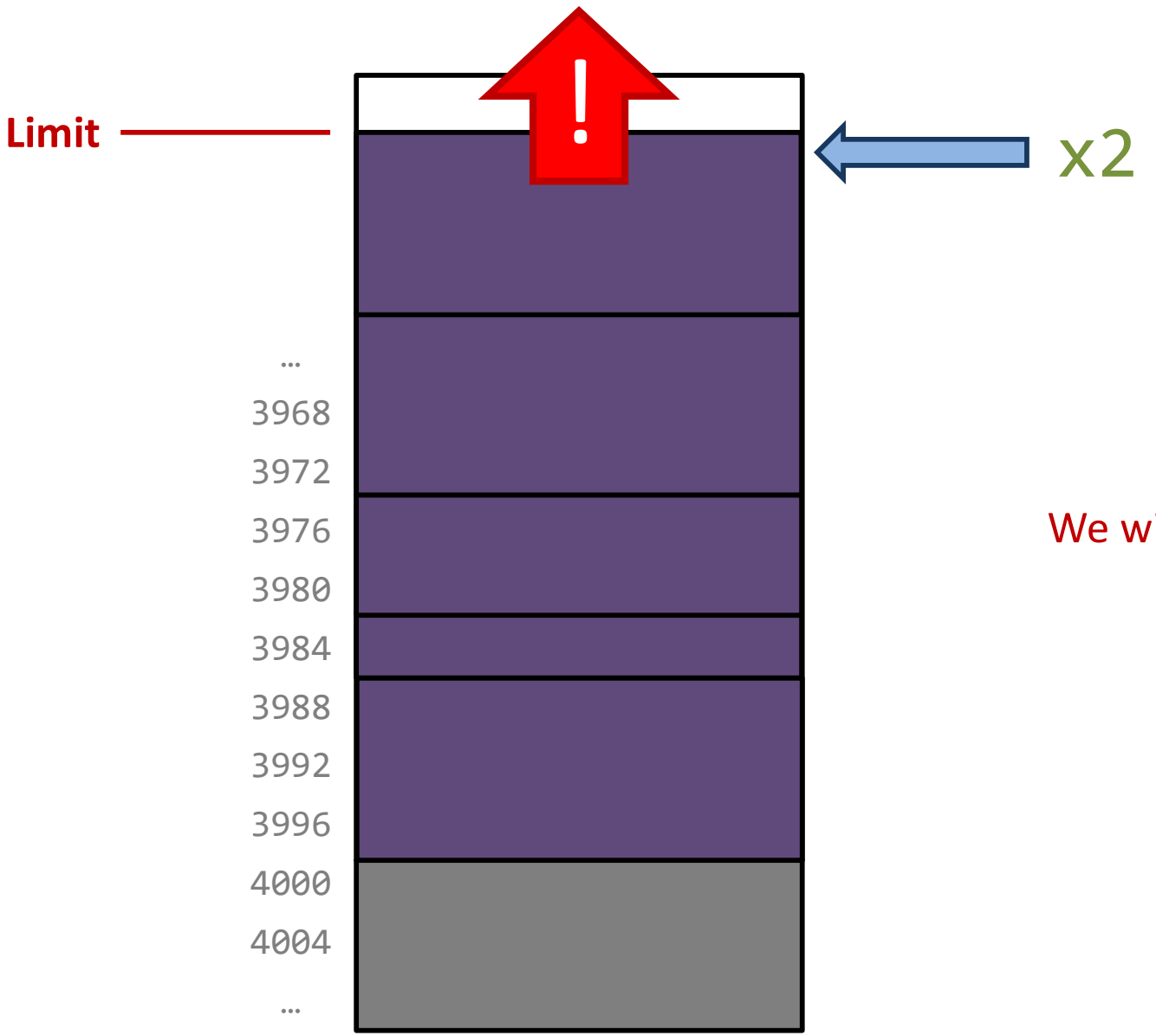
→ holds the **global interrupt enable**, along with other state

- **MIE**: Interrupts are globally enabled when MIE = 1 and globally disabled when MIE = 0.
- **MPIE**: MPIE holds the value of MIE prior to the trap.

The subset we use in **CS-200**

mstatus	*reserved*			MPIE	*reserved*	MIE	*reserved*
mie	*reserved*	MEIE	*reserved*	MTIE	*reserved*		
mip	*reserved*	MEIP	*reserved*	MTIP	*reserved*		

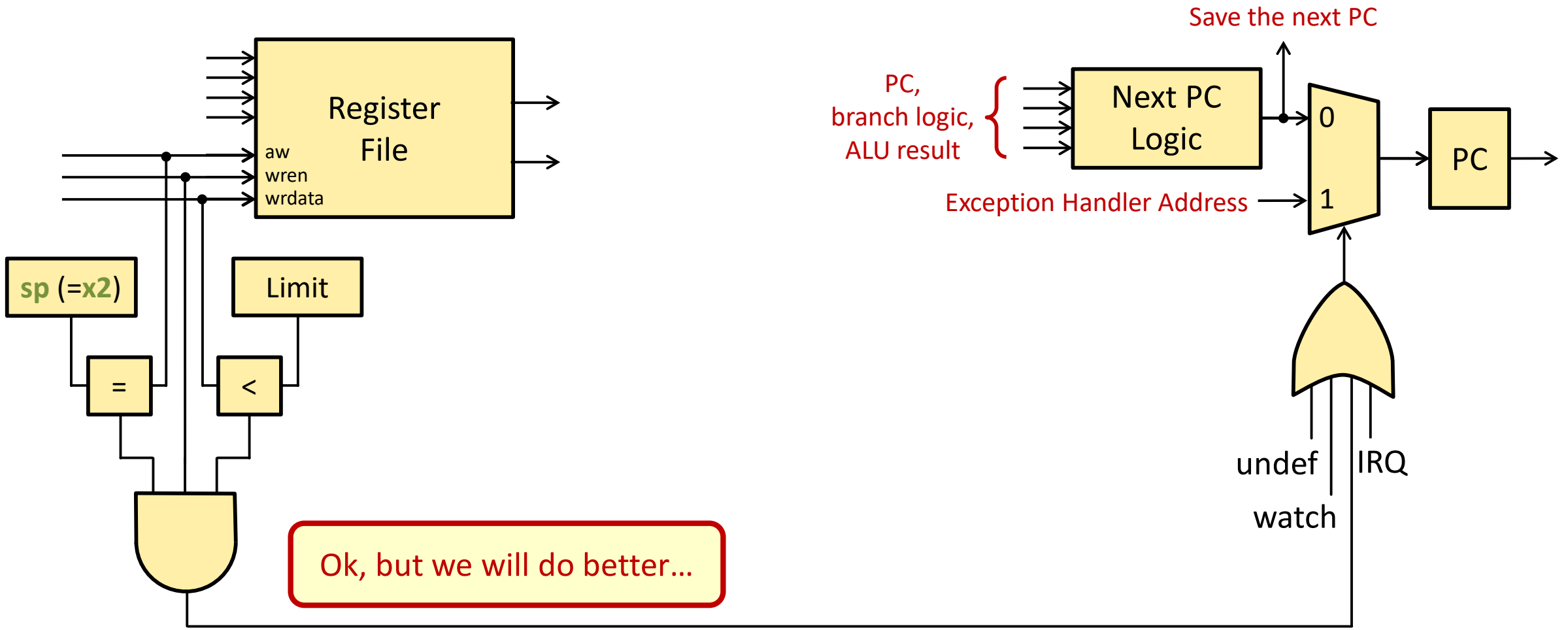
Remember?



We will definitely need to find a solution!

(In a few weeks' time…)

Stack-Full Detection?



Writing Handlers Is Very Very Tricky!

- Notice the problem of writing the handler for the above: we **cannot use the stack!**

- In general, one **cannot touch any register** and, maybe, one **cannot touch the stack**…
- Various solutions:
  - **Reserved ordinary registers** as in MIPS (`$k0` and `$k1`)
  - **Shadow registers** in old processors (early x86 and before)
  - Automatic **switch to a known safe stack** in x86

RISC-V solution is one dedicated CSR:

- `mscratch` (Machine Scratch) [not represented in the slide above]

  → holds one word of data for **temporary storage**
- Can be used to store a pointer to an empty region of memory or to a known safe stack

Speaking of the Stack…

```
funct:      addi   sp, sp, -12
            sw     ra, 0(sp)
            sw     s0, 4(sp)
            sw     s1, 8(sp)

               … etc. …

            lw     ra, 0(sp)
            lw     s0, 4(sp)
            lw     s1, 8(sp)
            addi   sp, sp, 12
            ret
```

=

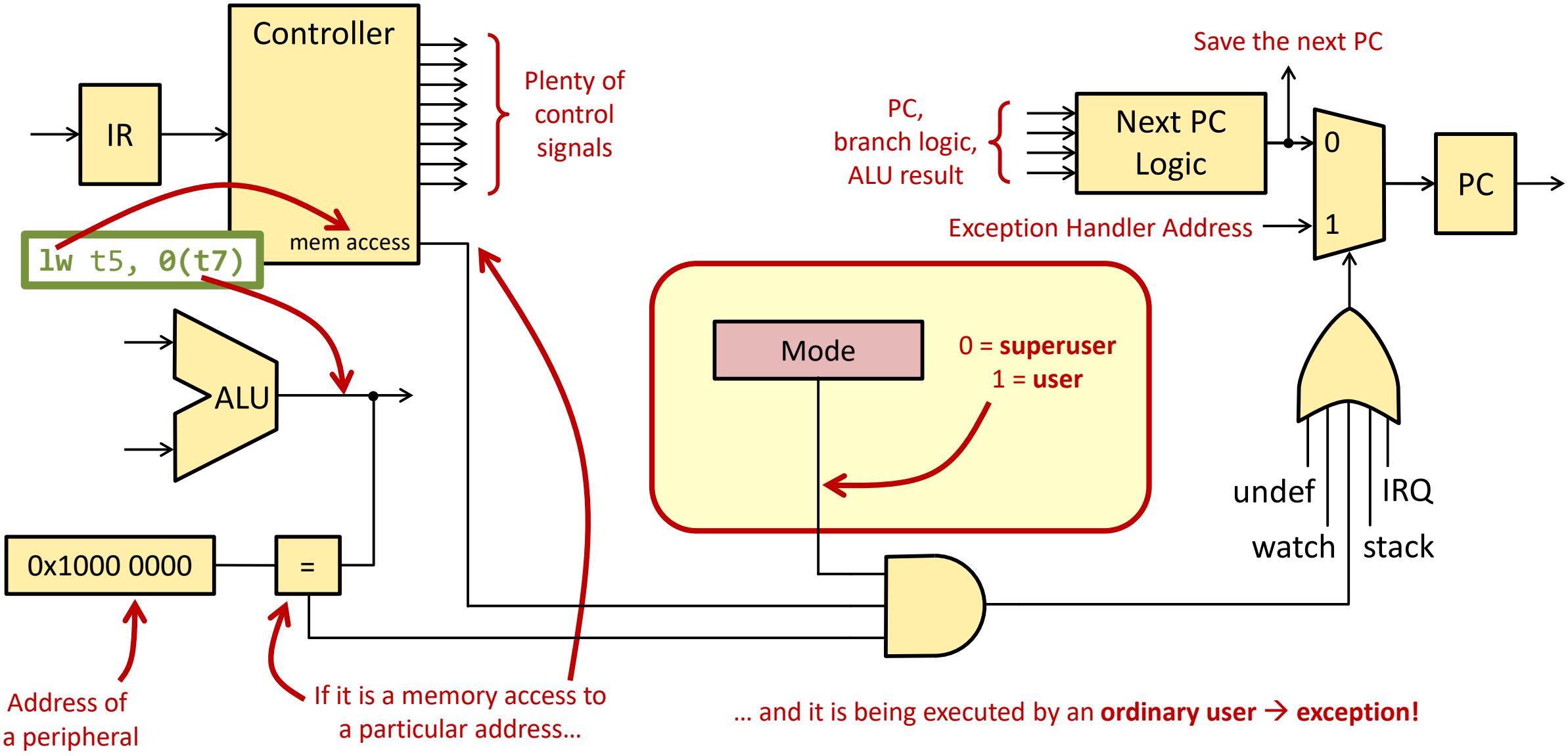
?

```
funct:      sw     ra, -12(sp)
            sw     s0, -8(sp)
            sw     s1, -4(sp)
            addi   sp, sp, -12

               … etc. …

            addi   sp, sp, 12
            lw     ra, -12(sp)
            lw     s0, -8(sp)
            lw     s1, -4(sp)
            ret
```

Protection: I/Os Are Not for Everyone!



Levels of Privilege = Processor Modes

1. Distinguish **at least** two processor **modes***:*
   - **User mode** for the user's programs
   - **Kernel, Supervisor, Executive,** etc. for the OS (kernel)
   - RISC-V has up to three: Machine, Supervisor, User

2. Have a part of the **processor state readable by all**, but only **writable with highest levels of privilege**; at least a
   - Current **mode register**
   - Other configuration registers (we will see some when discussing the Memory Hierarchy…)

3. Methods to **switch mode** back and forth
   - (i) A **dedicated instruction** to trigger a **software exception** and (ii) an instruction to **reset**
   - RISC-V has `ecall` (system call) and `mret`/`sret` (return from exception)

Processor Tasks on Exception

- What the processor should or could do when an exception is raised (depending on processors and types of exceptions):
  1. Mask further interrupts
  2. Save EPC
  3. Save information on the reason for the exception
  4. Modify privilege level (exception handlers run in some privileged mode)
  5. Free up some registers (e.g., copying them to shadow registers, where supported)
  6. Jump to the handler
- Most or all of these tasks are **reverted implicitly** with special instructions on exit
  - `mret` in RISC-V reverts the privilege level and the interrupt enable
- Some have to be **reverted explicitly by the handler**
  - Programmers may want to unmask further interrupts **as soon as it is safe**

Priorities

- We have seen that hardware **interrupt controllers** can help managing priorities (which interrupt is more urgent to serve?)
- Yet, this only affects the order IRQs are presented to the processor. But we may also want to **serve a high-priority interrupt while serving a lower-priority one**
- Alas, there is only one `mepc` and `mcause` register, and this is why, as soon as the processor takes an interrupt, it **must disable further interrupts**
- What can we do?
  - Save critical information about the interrupt (`mepc`, `mcause`, `mstatus`) on some **safe stack**, so that CSRs can be overwritten by further interrupts
  - Manually **reenable interrupts** (`mstatus`) without returning from the handler

Writing Handlers Is Very Very Tricky!

- Writing exception handlers is a **difficult task**!
  - Maybe the **stack cannot be used** (e.g., the exception results from a stack overflow)
  - Maybe the **exception handler cannot be interrupted** (e.g., the handler uses static locations to save data including `mscratch` and is therefore a nonreentrant procedure)
  - Maybe the **system cannot withstand not serving interrupts** for a long time (e.g., I/Os buffers fill up)

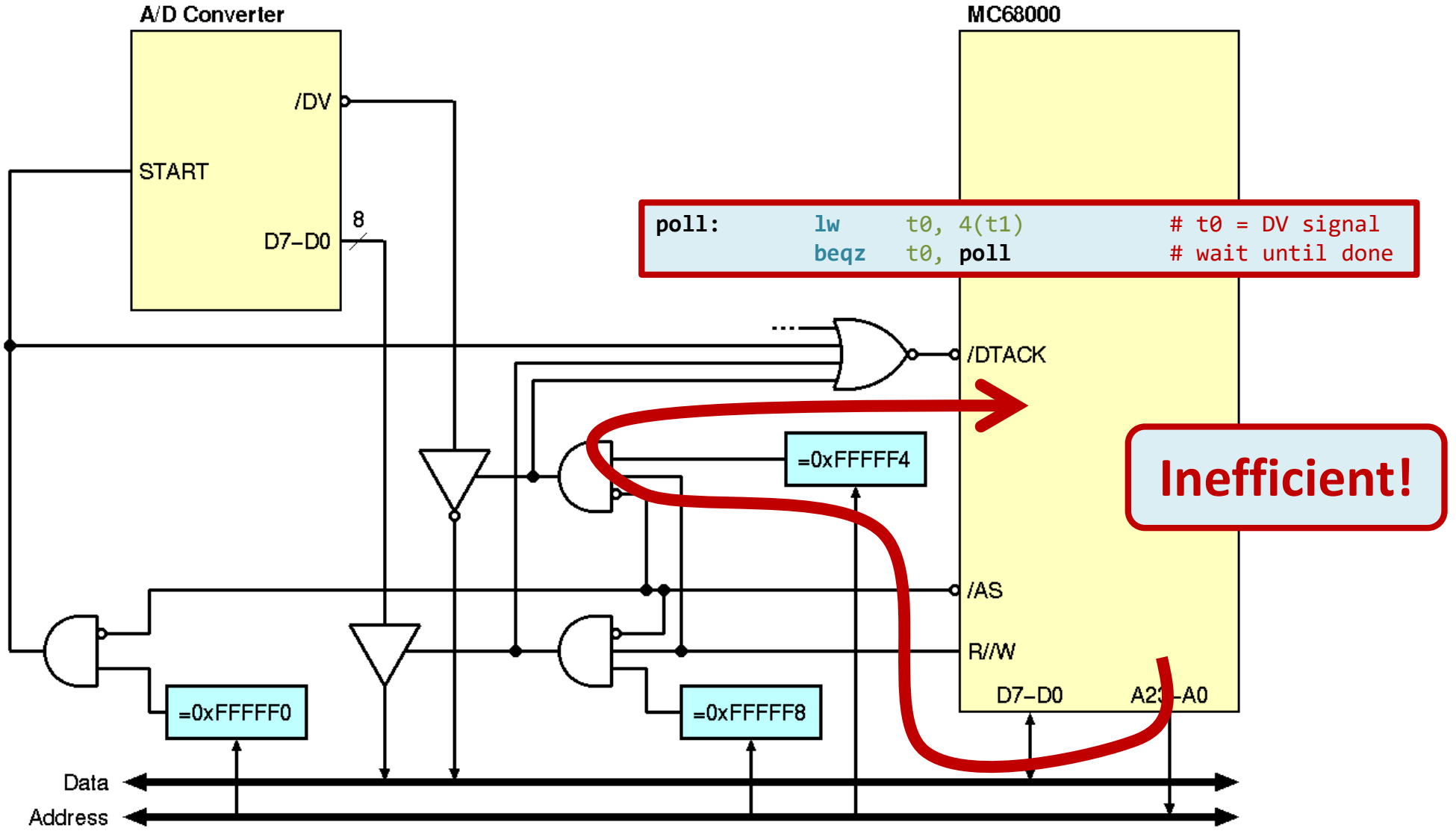
- Buggy **device drivers** from vendors of peripherals (invoked by the interrupt handler of the operating system and running in some privileged mode) are often responsible for operating system instability

  → This is why Microsoft **formally verifies** and **certifies** device drivers

Processor Design Issues with Exceptions

- Handling exceptions is **one of the biggest challenges** of high-performance processor design
- Great difficulties in determining the exact state of execution and supporting a **precise restart mechanism**
- Older processors did not support at all precise exceptions—**every exception was a terminating one**, and thus things were easy
- **More later in CS-200** and in elective courses…

Back to Our A/D Converter

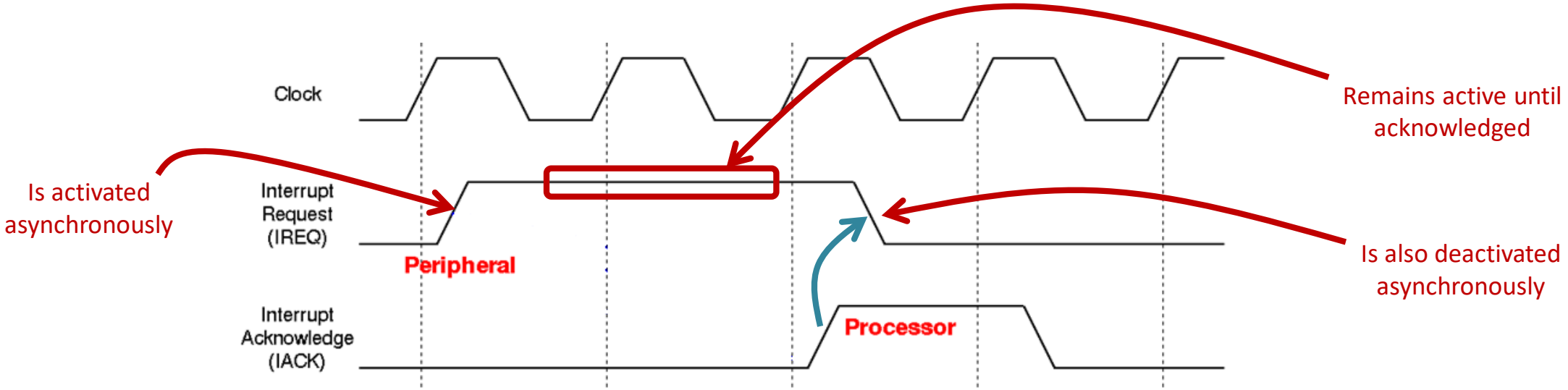


Example:
Better A/D Converter Interface

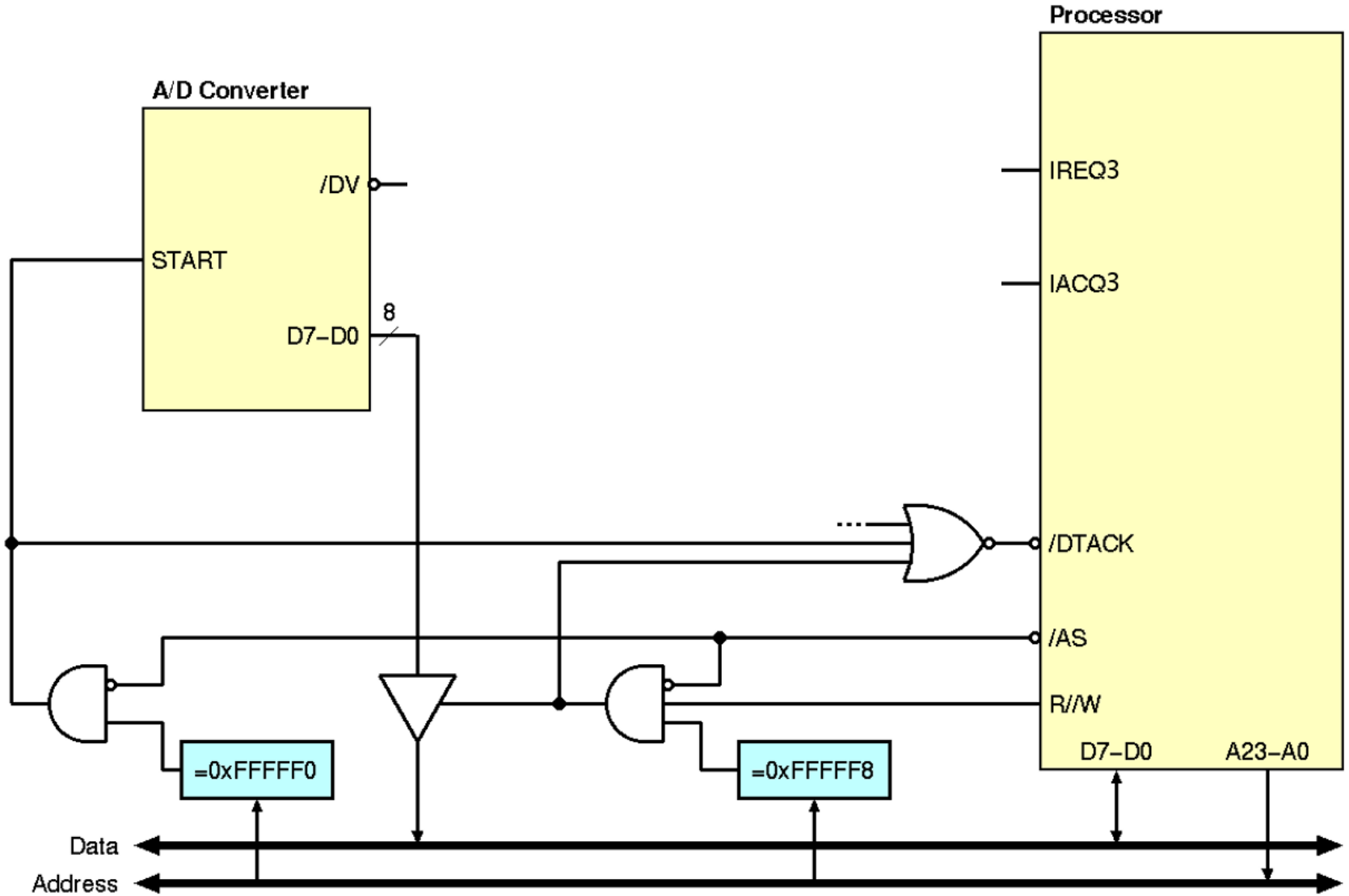
- Improve the interface to the A/D converter so that:
  - Any access (R or W) to address `0xFFFFF0` starts a new conversion
  - **Upon completion, the A/D converter raises an interrupt through the appropriate interrupt request signal of the processor**
  - The result of the conversion can be read by the processor at address `0xFFFFF8`

Example:
Simple IREQ and IACK Mechanism

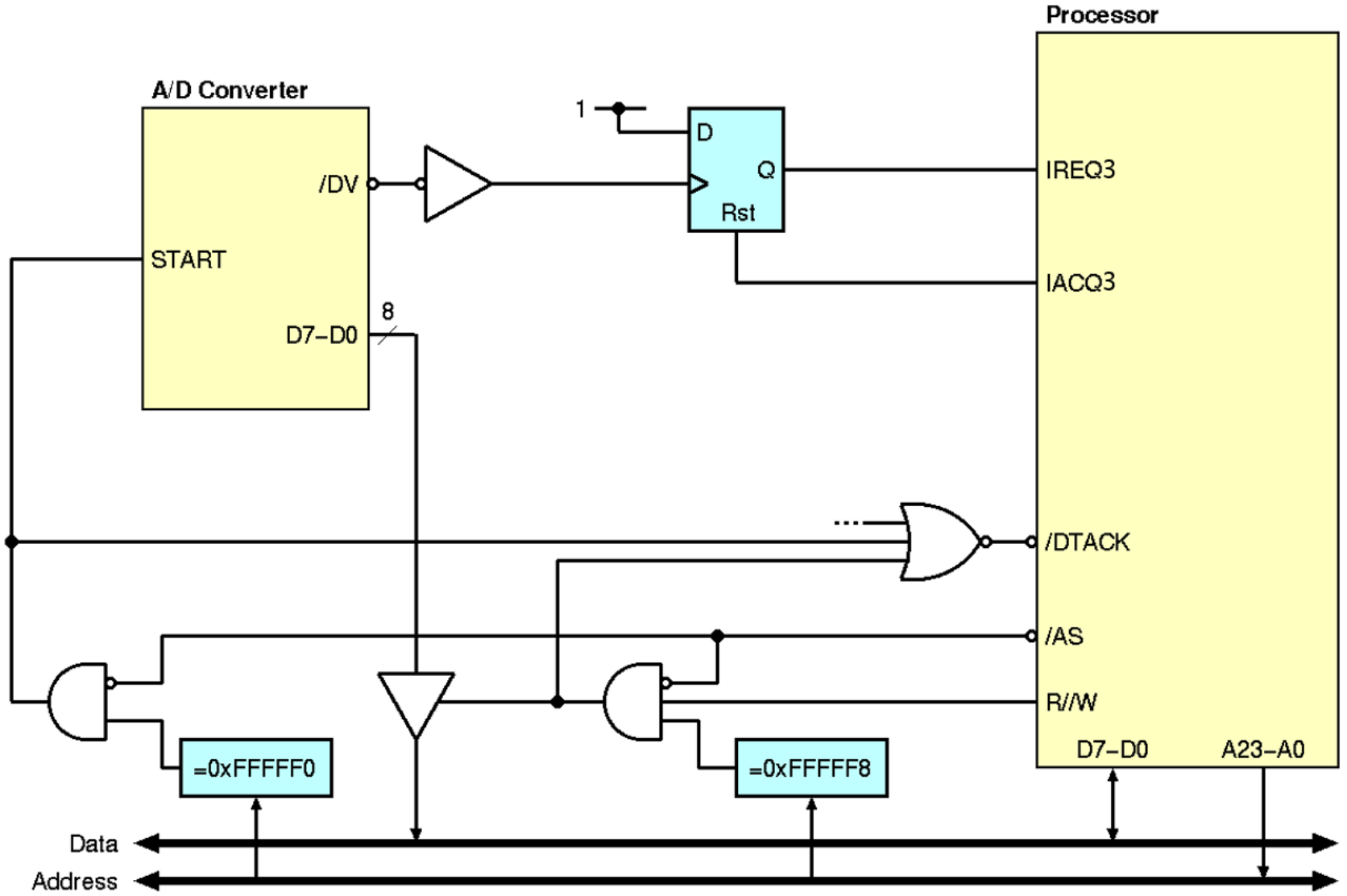
- Suppose that our 8-bit processor has an internal interrupt controller with various **IREQ**/**IACK** signal pairs for I/O interrupt requests
- We have been assigned for our ADC these:
  - **IREQ3**: input; dedicated to our peripheral to request attention
  - **IACK3**: output; used by the processor to signal to our peripheral that the request is acknowledged and is being served



Circuit with IREQ/IACK



A Better A/D Converter Interface



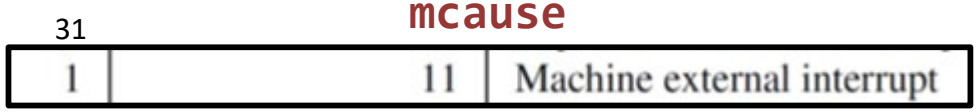
A/D Converter: `startADC`

```
startADC:   lui     t0, 0xfff
            addi    t0, t0, 0xff0       # t0 = 0xfffff0
            sw      zero, 0(t0)         # start conversion

            ret
```

Software: handler

mcause



A/D Converter: handler

```
handler:   addi  sp, sp, -120            # Save all registers but zero and sp
           sw    x1, 0(sp)
           sw    x3, 4(sp)
              … etc. …
           sw    x31, 116(sp)

           csrr  s0, mcause              # Read exception cause
           bgez  s0, handleExceptions    # Branch if not an interrupt (MSB = 0, looks like zero or a positive number…)
           slli  s0, s0, 1               # Get rid of the MSB of s0, so that what is left is the cause
           srli  s0, s0, 1
           li    s1, 11                  # s1 = external interrupt cause
           bne   s0, s1, handleOtherInts # Branch if not an external interrupt

           jal   readADC                 # Returns a0 = ADC result
           jal   insertIntoBuffer        # Gets a0 = value to add to a circular buffer

restore:   lw    x1, 0(sp)               # Restore all registers but zero and sp
           lw    x3, 4(sp)
              … etc. …
           lw    x31, 116(sp)
           addi  sp, sp, 120

           mret
```

A/D Converter: readADC

```
readADC:    li      t0, 0xfffff0        # t0 = 0xfffff0
            lw      a0, 8(t0)           # get ADC data output

            ret
```

A/D Converter: insertIntoBuffer

```
.section .data
                .equ    bufferSize, 1024            # Define buffer size (must be a power of two)
                .equ    bufferBytes, bufferSize * 4 # Compute the total size in bytes for the buffer
bufferPointer:  .word   0                           # Initialize the pointer to index 0
buffer:         .space  bufferBytes                 # Allocate space for bufferSize * wordSize bytes

.section .text
insertIntoBuffer:   la      t0, bufferPointer       # Load address of bufferPointer into t0
                    lw      t1, 0(t0)               # Load current buffer pointer into t1
                    la      t2, buffer              # Load base address of the buffer into t2
                    slli    t3, t1, 2               # Multiply buffer pointer (t1) by 4 to get byte offset
                    add     t4, t2, t3              # Add offset to buffer base address (= next word)
                    sw      a0, 0(t4)               # Store a0 into buffer at calculated position
                    addi    t1, t1, 1               # Increment buffer pointer by 1
                    li      t5, bufferSize - 1      # Load bufferSize - 1 into t5 (mask for power of 2)
                    and     t1, t1, t5              # Apply bitwise AND to wrap around
                    sw      t1, 0(t0)               # Store updated buffer pointer

                    ret
```

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 4**; **Section 4.10** only until p. 335.
  - **Chapter 5**; **Section 5.10** only bottom of p. 458 and bottom of p. 460, and **Section 5.14** (not the `fence` stuff).