Module 8:

Exceptions

Introduction to computers II

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Outline

These slides are based on:

- S.L. Harris and D. Harris. *Digital Design and Computer Architecture. RISC-V Edition.*
- D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. RISC-V Edition.*

Introduction

- What happens if the reduced ISA RISC-V executes a program that, by mistake, contains...
  - ... an instruction of the full ISA?
  - ... a branch instruction to an address of the data section?
  - ... a branch instruction to an address that is not a multiple of 4?
  - ... a `lw`/`sw` instruction of a data in an address that is not a multiple of 4?

- The result will be unpredictable because, depending on the case, it reads from memory...
  - ... an unforeseen instruction code, which is executed.
  - ... a mix of two data, which is processed.

- For this reason, all processors implement mechanisms to:
  - Detect these cases by HW.
  - Allow the SW to perform corrective actions in these cases.

Introduction

Exceptions vs. interrupts (i)

- Trap: unexpected event that alters the execution of a program.
  - Exception: the cause of the event is internal to the processor.
    - It is synchronous with the program: it happens when the instructions are executed.
    - The processor detects the trap.
    - Illegal instruction, alignment error, etc...
  - Interrupt: the cause of the event is external to the processor.
    - It is asynchronous with the program: it happens regardless of the program.
    - The processor has an input which indicates when the trap happens.
    - I/O request, overheating warning, RTC tick, etc...
- Traps are handled as unscheduled function calls:
  - When the processor detects a trap, it automatically branches to a predetermined address where the programmer has placed a SW function.
  - The function performs a certain action and returns to the program (or aborts it).
  - These functions are usually part of the operating system:
    - ESR: Exception service-routine handler
    - ISR: Interrupt service-routine handler

Introduction

Exceptions vs. interrupts (ii)

- In order for the SW service routine to perform the appropriate action, it must know the specific trap generated by the HW.

- There are two basic methods (service modes) for this:
  - Direct: the processor stores a different value for each trap in a “special” register and branches to a unique address.
    - In that address, a unique common routine for all the traps is located.
    - The routine reads the special register to determine what to do: the stored value works as a parameter to the routine.
  - Vectored: the processor branches to different addresses for each kind of generated trap.
    - There are as many service routines as different traps.
    - Each routine performs the appropriate action depending on the generated trap.

Control and status registers

- A RISC-V may have up to 4096 32-bit CSR (Control and Status Registers) to control the system.
  - They are different from the conventional **`x0`**-**`x31`** registers.
  - To operate with them, they must be copied in conventional registers using privileged instructions.
- Some CSR examples are:

# Reg.	Alias	Description
`0xf11`	**`mvendorid`**	manufacturer id*
`0xf12`	**`marchid`**	microarchitecture id*
`0xf13`	**`mimpid`**	implementation id*
`0x301`	**`misa`**	implemented base and extended ISA*
`0xb00`	**`mcycle`**	number of cycles since a given instant
`0xb02`	**`minstret`**	number of instructions executed since a given instant

(*) read-only registers

Control and status registers

For exception handling (i)

- The main CSR to handle exceptions are:

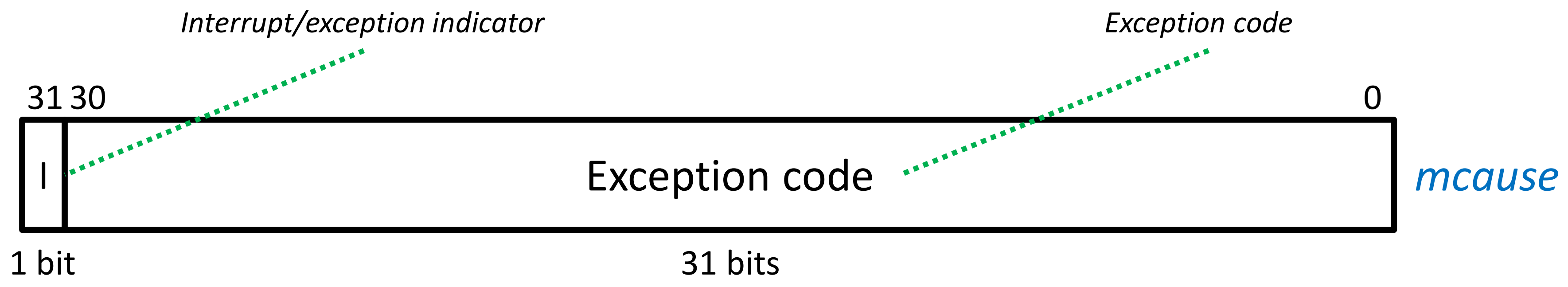
# Reg.	Alias	Description
`0x305`	`mtvec`	machine trap-vector – service routine base address
`0x342`	`mcause`	machine trap cause – exception cause id
`0x341`	`mepc`	machine exception program counter – address of the instruction that has produced the exception
`0x340`	`mscratch`	machine scratch – pointer to a service routine auxiliary stack
`0x343`	`mtval`	machine trap value – memory address that has produced the exception

- If an exception happens in a RISC-V, the processor:
  - Cancels the execution of the current instruction (which causes the exception).
  - Saves the address of that instruction in **`mepc`**.
  - If the exception happens when accessing a data, it saves its address in **`mtval`**.
  - Saves a code that identifies the exception in **`mcause`**.
  - Branches to the address stored in **`mtvec`**.

Control and status registers

For exception handling (ii)

- The **mcause** register is structured in two fields:



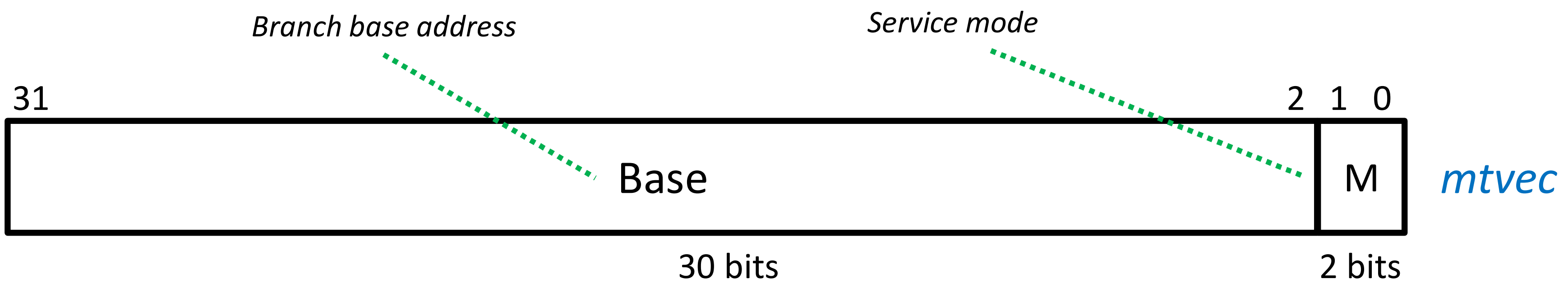
- It indicates the kind of exception/interrupt generated, e.g.:

I	Exception Code	Kind of trap
0	0x0000 (0)	Misaligned instruction address
0	0x0002 (2)	Illegal instruction
0	0x0004 (4)	Misaligned load address
0	0x0006 (6)	Misaligned store address
0	0x000b (11)	Execution of the `ecall` instruction
1	0x0007 (7)	Timer interrupt
1	0x000b (11)	External interrupt

Control and status registers

For exception handling (iii)

- The **mtvec** register is also structured in two fields:



- It defines the branch address and the service mode

M	Description	Performed branch
00	Direct mode	*if* (exception | interrupt) *then* (PC ← PC + Base)
01	Vectored mode	*if* (exception) *then* (PC ← PC + Base) *elsif* (interrupt) *then* (PC ← PC + Base + (mcause$_{30:0}$ << 2))

Privileged instructions

To access a CSR

- They allow copying a CSR into a conventional register, modifying at the same time the value stored in the read CSR.

Instruction	Operation	Description

`csrrw rd, csr, rs1`	rd ← csr
	csr ← rs1
		control and status register read/write
		swaps csr and register

`csrrs rd, csr, rs1`	rd ← csr
	csr ← csr | rs1
		control and status register read/set
		copies csr and sets bits (=1) using a mask

`csrrc rd, csr, rs1`	rd ← csr
	csr ← csr & ~rs1
		control and status register read/clear
		copies csr and resets bits (=0) using a mask

`csrrwi rd, csr,` imm_{5b}	rd ← csr
	csr ← zExt(imm)
		control and status register read/write immediate
		copies csr and loads a constant

`csrrsi rd, csr,` imm_{5b}	rd ← csr
	csr ← csr | zExt(imm)
		control and status register read/set immediate
		copies csr and sets bits (=1) using a constant mask

`csrrci rd, csr,` imm_{5b}	rd ← csr
	csr ← csr & ~zExt(imm)
		control and status register read/clear immediate
		copies csr and resets bits (=0) using a constant mask

Privileged instructions

Others

Instruction	Operation	Description
`mret`	PC ← mepc	**m**achine **r**eturn returns from an exception
`ecall`	generates exception 11	**e**nvironment **call** operating system call

- RISC-V can operate with different privilege levels called modes.
  - They determine if the processor can execute privileged instructions and on what CSR subset.
  - A processor can only be in a mode at each instant.
- The maximum privilege mode in RISC-V is the M-mode: machine mode.
  - All RISC-V have this mode, which is the mode after a reset.
  - Depending on the implementation, there may be others:
    - S-mode: Supervisor mode, mode in which the operating system typically runs.
    - U-mode: User mode, mode in which the applications typically run.
  - A change of mode happens when a trap is generated or a privileged instruction that modifies a certain CSR (**`mstatus`**) is executed.

Privileged instructions

Pseudo-instructions

- It is very common to change/read/write a CSR without the need to swap it with a conventional register.

Instruction	Operation	Translation	Description
`csrr rd, csr`	rd ← csr	`csrrs rd, csr, x0`	csr read copies csr in a register
`csrw csr, rs1`	csr ←rs1	`csrrw x0, csr, rs1`	csr write copies a register in csr
csrwi *csr*, imm_{5b}	csr ← zExt(imm)	`csrrwi x0, csr, imm`	csr write immediate copies constant in csr
csrsi *csr*, imm_{5b}	csr ← csr | zExt(imm)	`csrrwi x0, csr, imm`	csr set immediate sets (=1) csr bits
csrci *csr*, imm_{5b}	csr ← csr & ~zExt(imm)	`csrrwi x0, csr, imm`	csr clear immediate resets (=0) csr bits

Privileged instructions

Instruction formats (i)

- All privileged instruction are I-type and have the same operation code:

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
$imm_{11:0}$	rs1	funct3	rd	op	*I-type*
000000000000	00000	000	00000	1110011	**`ecall`**
001100000010	00000	000	00000	1110011	**`mret`**
csr	rs1	fucnt3	rd	1110011	**`csrrX`**
csr	imm	funct3	rd	1110011	**`csrrXi`**

Privileged instructions

Instruction formats (ii)

- In order to differentiate them, they have different function codes:

Privileged instructions

op	funct3	Instruction	Type
1110011	001	`csrrw`	I
	010	`csrrs`	I
	011	`csrrc`	I
	101	`csrrwi`	I
	110	`csrrsi`	I
	111	`csrrci`	I

Exception service routines

- The processor, when an exception is detected, branches to the service routine automatically.
  - The executing program is interrupted without its knowledge.

- To do so, the exception service routine must:
  - Push all the registers in use in the memory region pointed by the **`mscratch`** CSR.
    - Since it is an unexpected situation, it must push both the preserved and the temporary registers.
    - If it calls other functions, it must also push the **`ra`** register.
    - Using a different memory region so that the program stack is not altered.
  - Read **`mcause`** to know the exception that has been produced.
  - Act accordingly.
- If the exception allows continuing the program execution, it must also:
  - Pop the pushed registers.
  - Return to the interrupted program by executing **`mret`**.

Exception service routines

Example

ASM

```
esr:
 csrrw t0, mscratch, t0
 add   t0, t0, -8
 sw    t1, 4(t0)
 sw    t2, 0(t0)
 csrr  t1, mcause
 li    t2, 2
 bne   t1, t2, other
illegalop:
 csrr  t2, mepc
 addi  t2, t2, 4
 csrw  mepc, t2
 j     done
other:
 j     .
done:
 lw    t1, 4(t0)
 lw    t2, 0(t0)
 add   t0, t0, 8
 csrrw t0, mscratch, t0
 mret
```

Swaps `t0` *and* `mscratch`

Pushes the context

Checks if the exception cause was illegal instruction (code 2)

If that was the cause: skips the instruction by modifying `mepc`

If it was another cause: aborts the program

Pops the context

Swaps `t0` *and* `mscratch`

Returns to the program

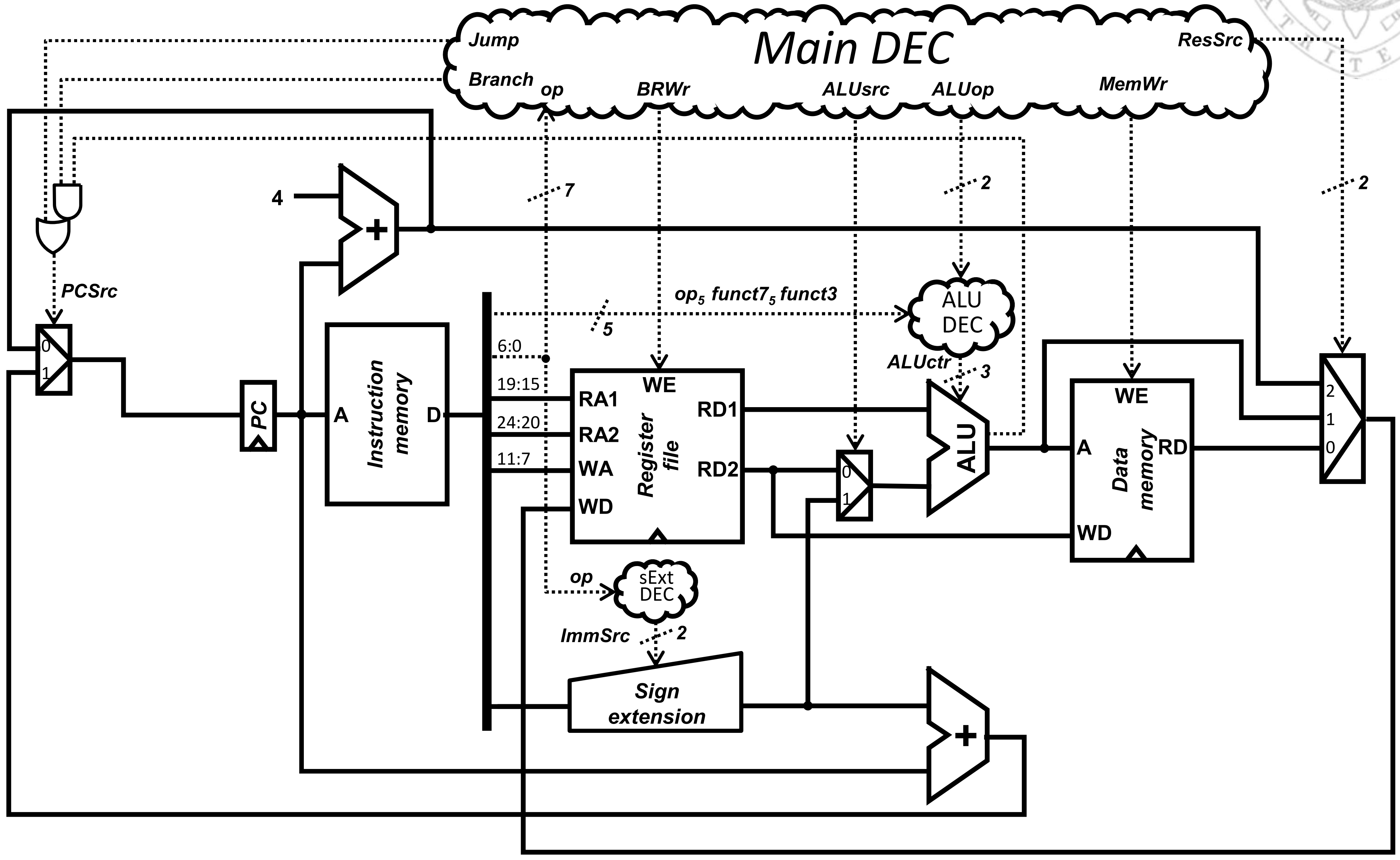
Reduced architecture RISC-V

With basic exception handling

- The reduced ISA RISC-V microarchitecture will be extended in order to handle the following exceptions:
  - Misaligned access to the instruction memory (cause 0).
  - Illegal instruction due to an unknown operation code (cause 2).
    - E.g., if any of the following is executed: **`lui`**, **`auipc`**, **`jalr`**, **`mul`**...
  - Illegal instruction due to a non-implemented arithmetic-logic instruction (cause 2).
    - E.g., if any of the following is executed: **`xor`**, **`sll`**, **`sra`**, **`xori`**...
  - Misaligned access to the data memory (read: cause 4, write: cause 6).
- 3 CSR with reduced functionality will be added:
  - **`mepc`**
  - **`mcause`** (read-only).
  - **`mtvec`** with a fixed value of **`0x1c000000`** (read-only).
- 2 privileged instructions with reduced functionality will be added:
  - **`csrrw`** limited to the **`mepc`** and **`mcause`** CSR
  - **`mret`**

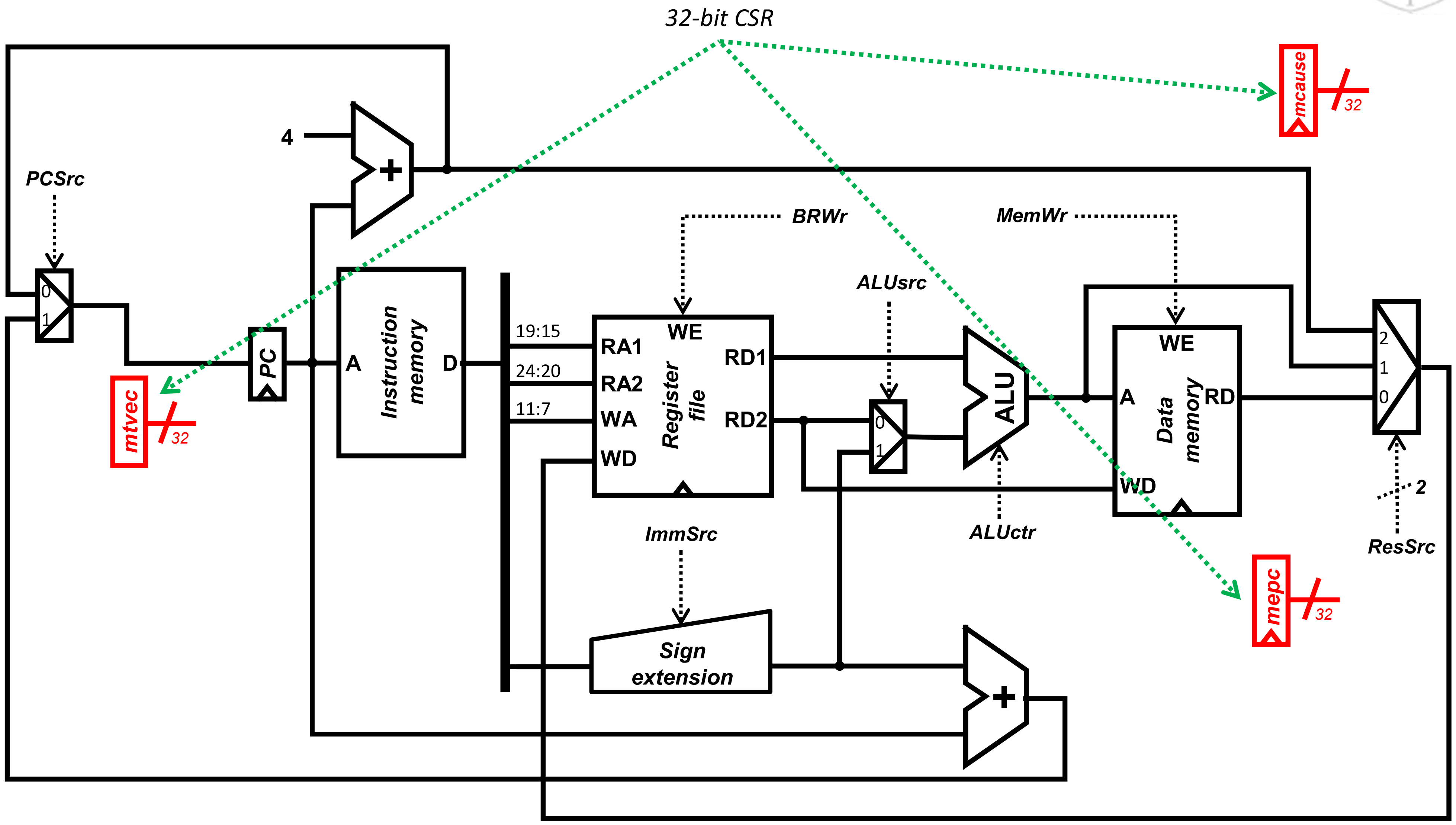
Single-cycle processor

Original data path + controller



Single-cycle processor

Data path + exception handling (i)

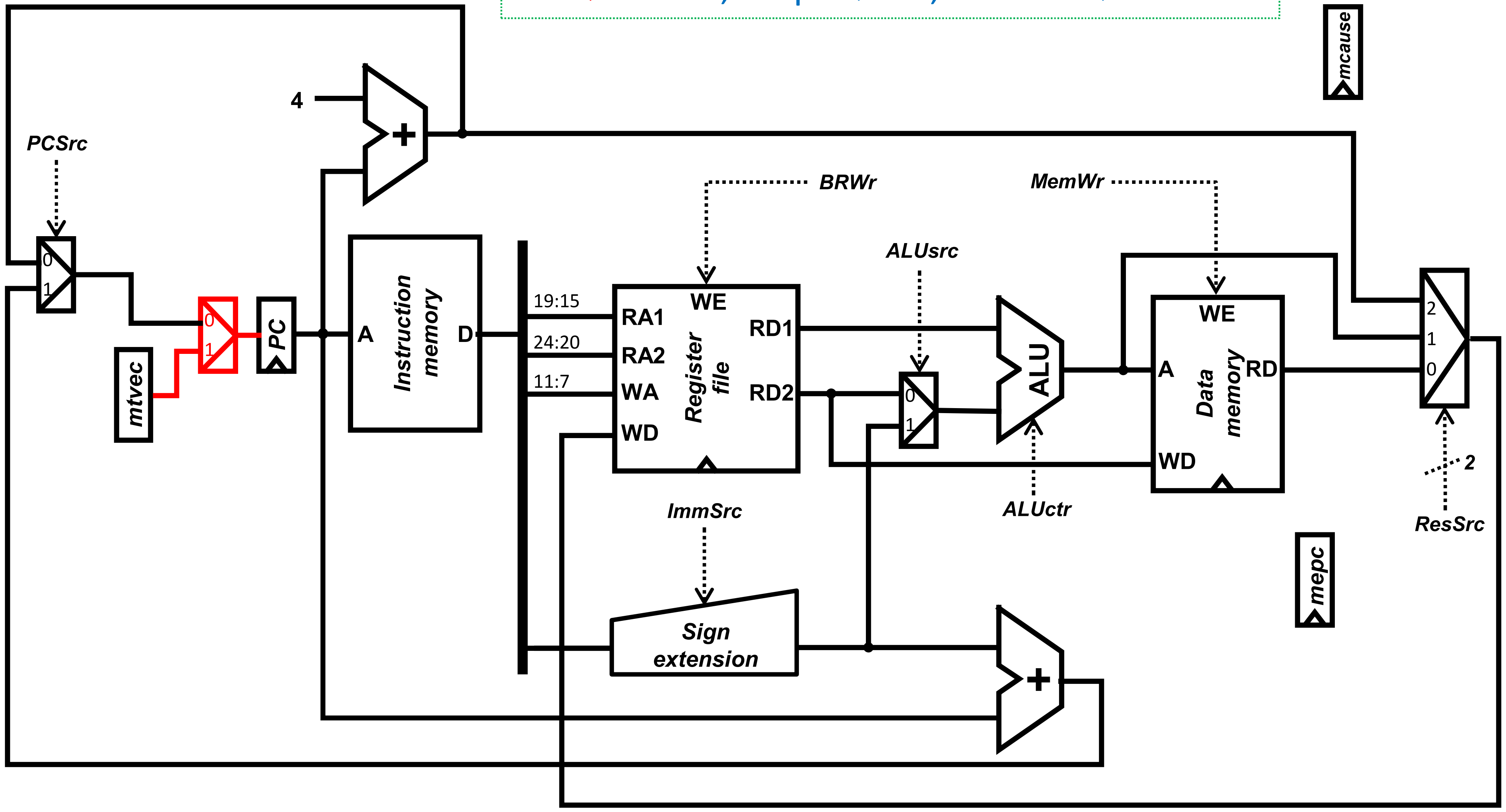


Single-cycle processor

Data path + exception handling (ii)

if exception:

PC ← mtvec, mepc ← PC, mcause ← *"cause"*

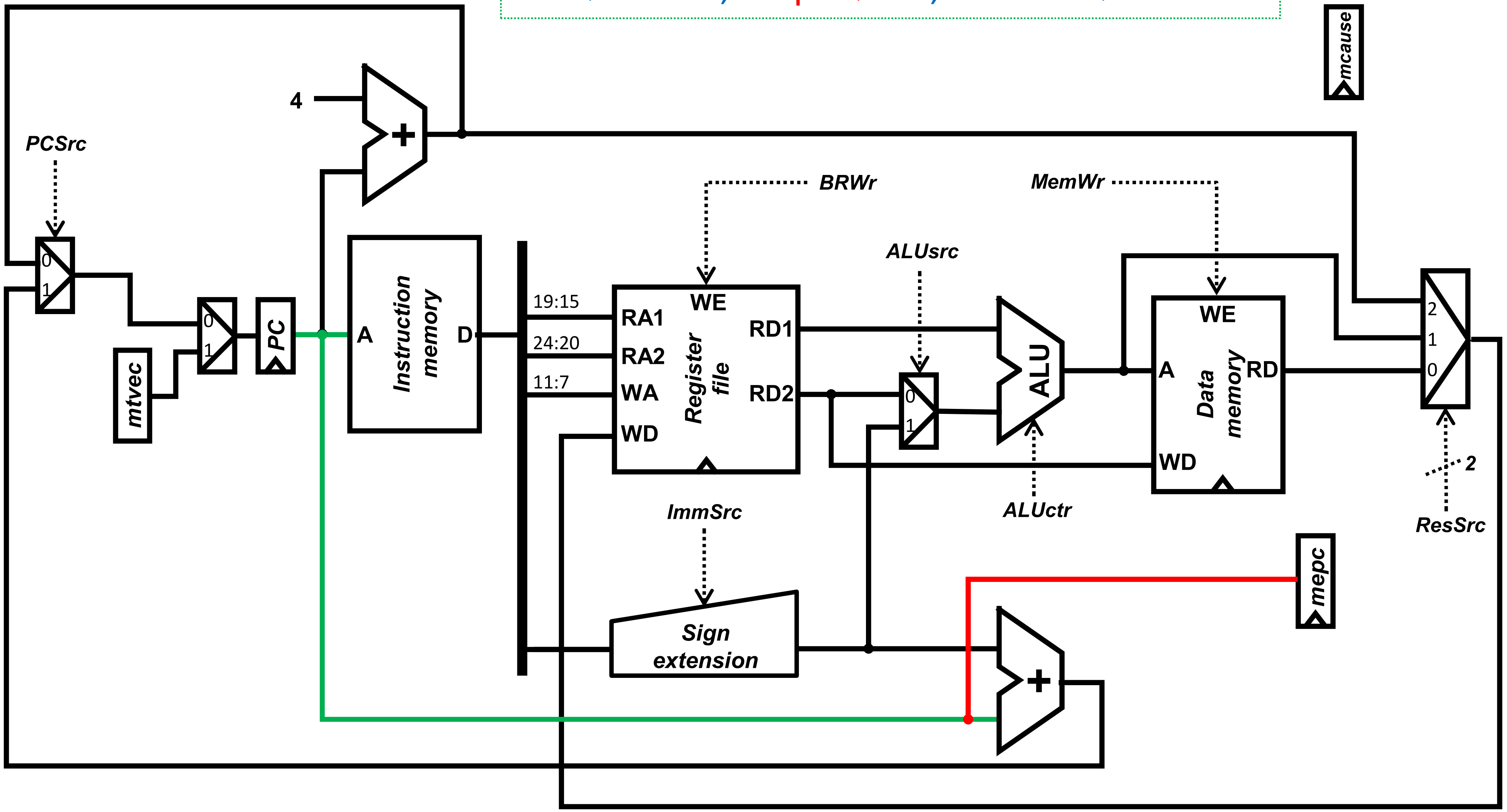


Single-cycle processor

Data path + exception handling (ii)

if exception:

PC ← mtvec, mepc ← PC, mcause ← *"cause"*

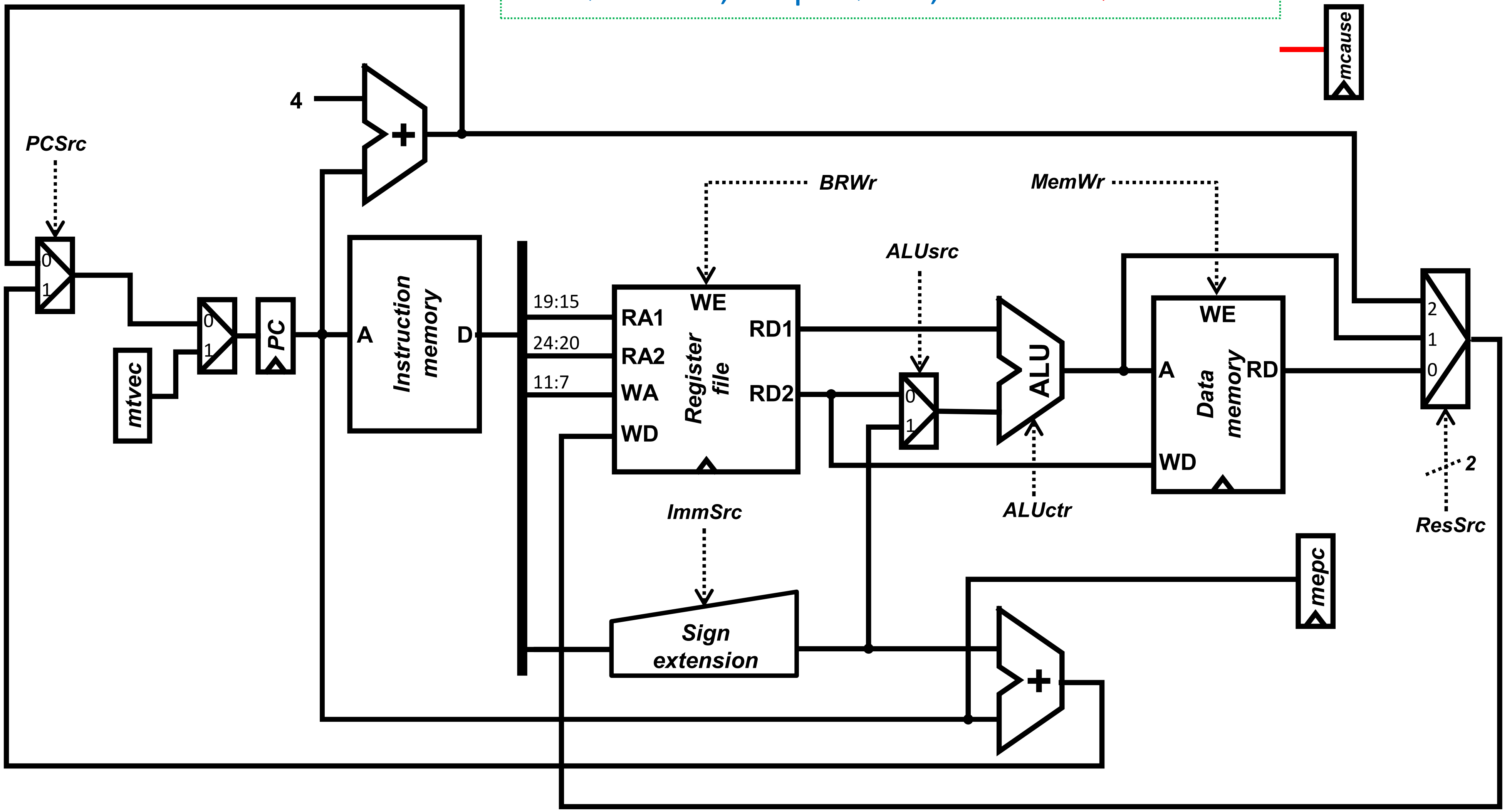


Single-cycle processor

Data path + exception handling (ii)

if exception:

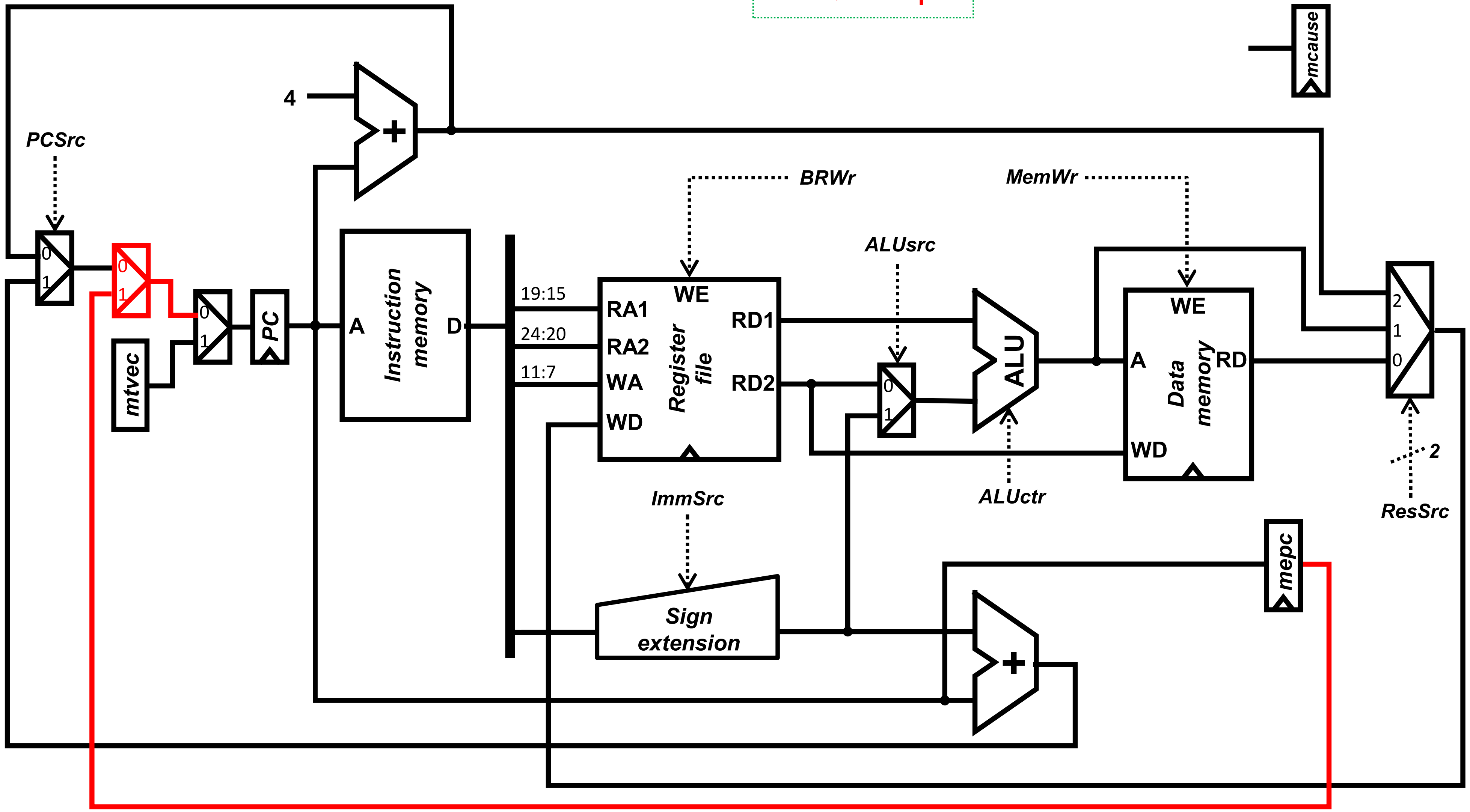
PC ← mtvec, mepc ← PC, mcause ← *“cause”*



Single-cycle processor

Data path + exception handling (iii)

```
mret
```
PC ← mepc

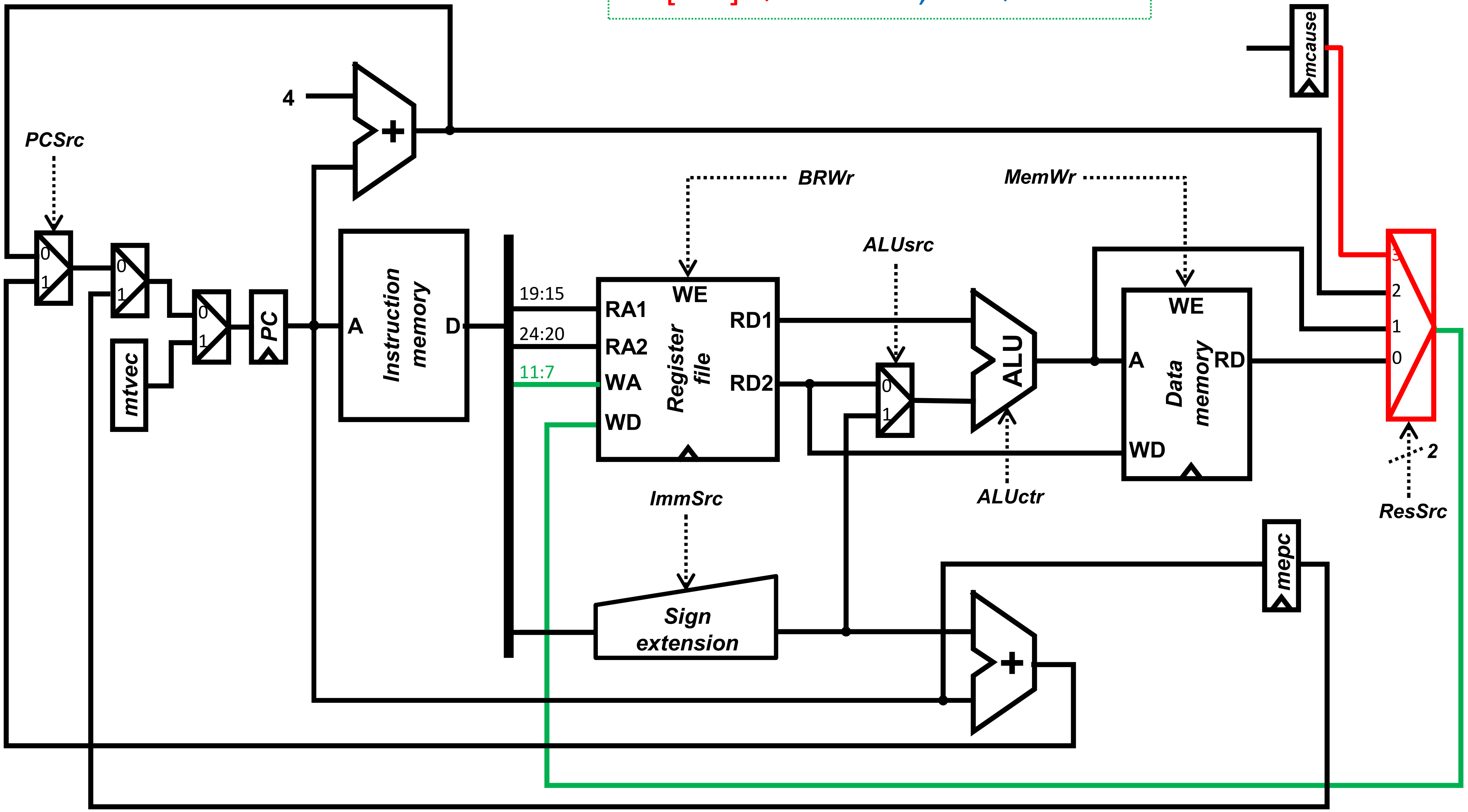


Single-cycle processor

Data path + exception handling (iv)

```
csrrw rd, mcause, rs1
```

RF[rd] ← mcause, PC ← PC+4

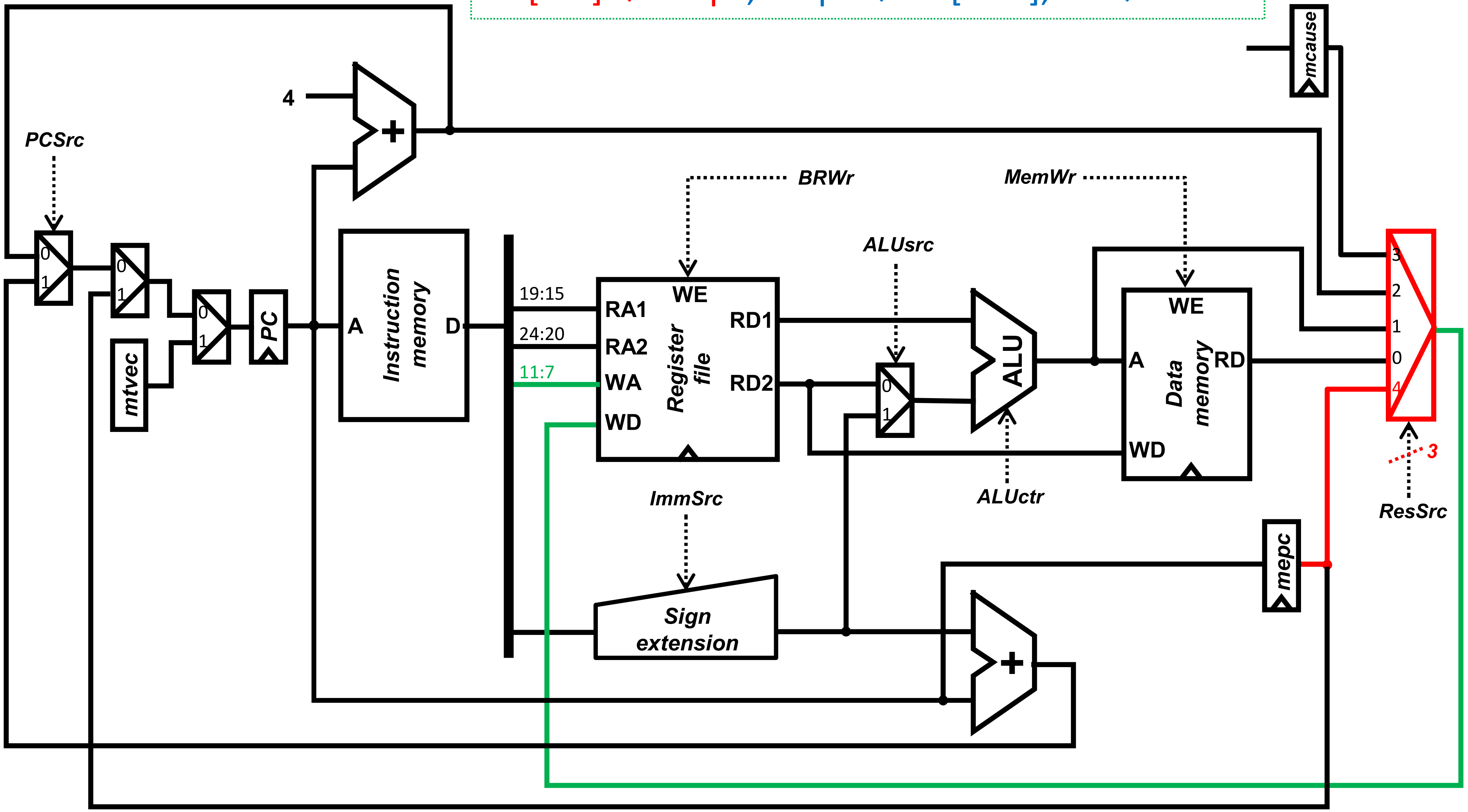


Single-cycle processor

Data path + exception handling (v)

```
csrrw rd, mepc, rs1
```

RF[rd] ← mepc, mepc ← RF[rs1], PC ← PC+4

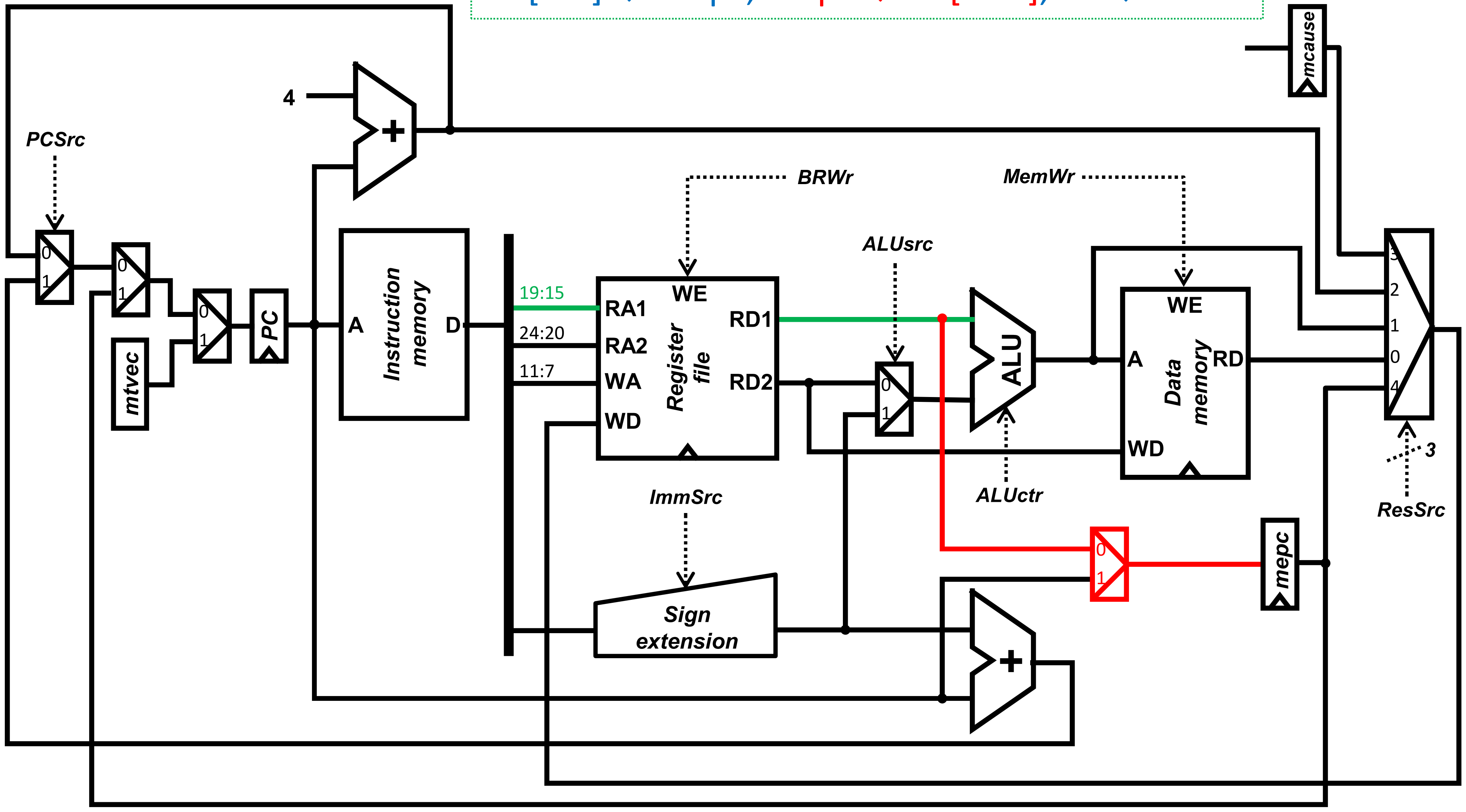


Single-cycle processor

Data path + exception handling (v)

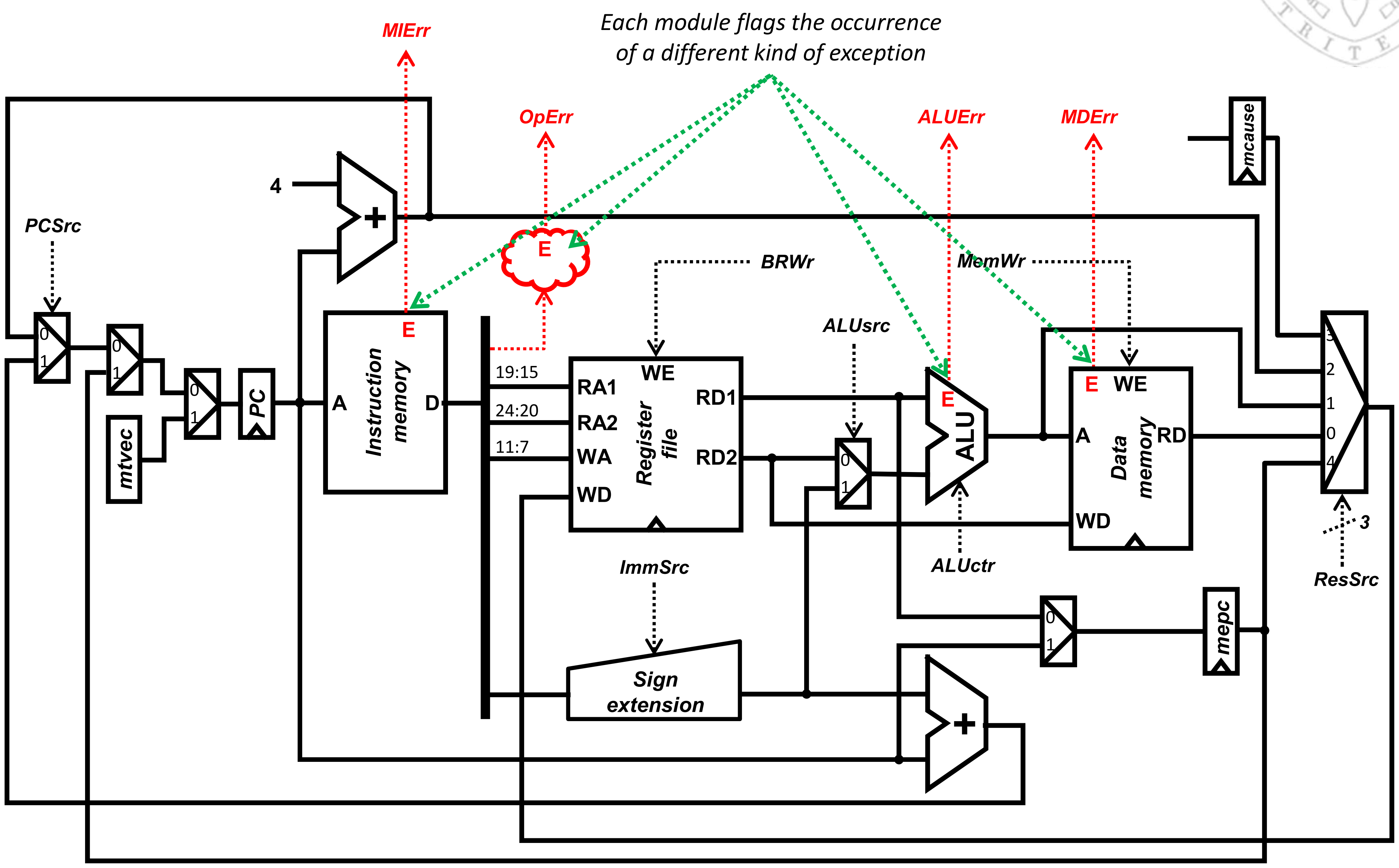
```
csrrw rd, mepc, rs1
```

RF[rd] ← mepc, mepc ← RF[rs1], PC ← PC+4



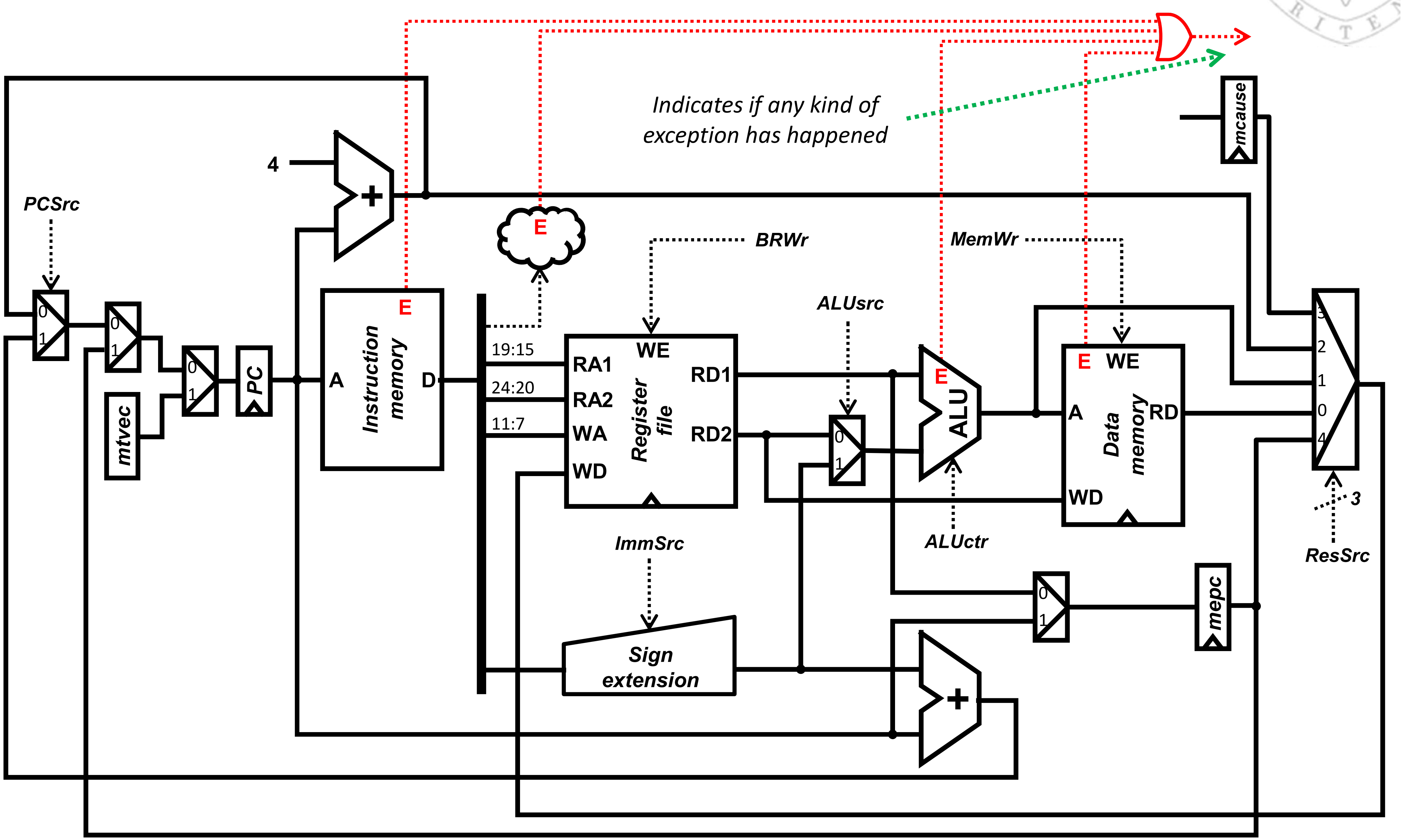
Single-cycle processor

Exception controller (i)



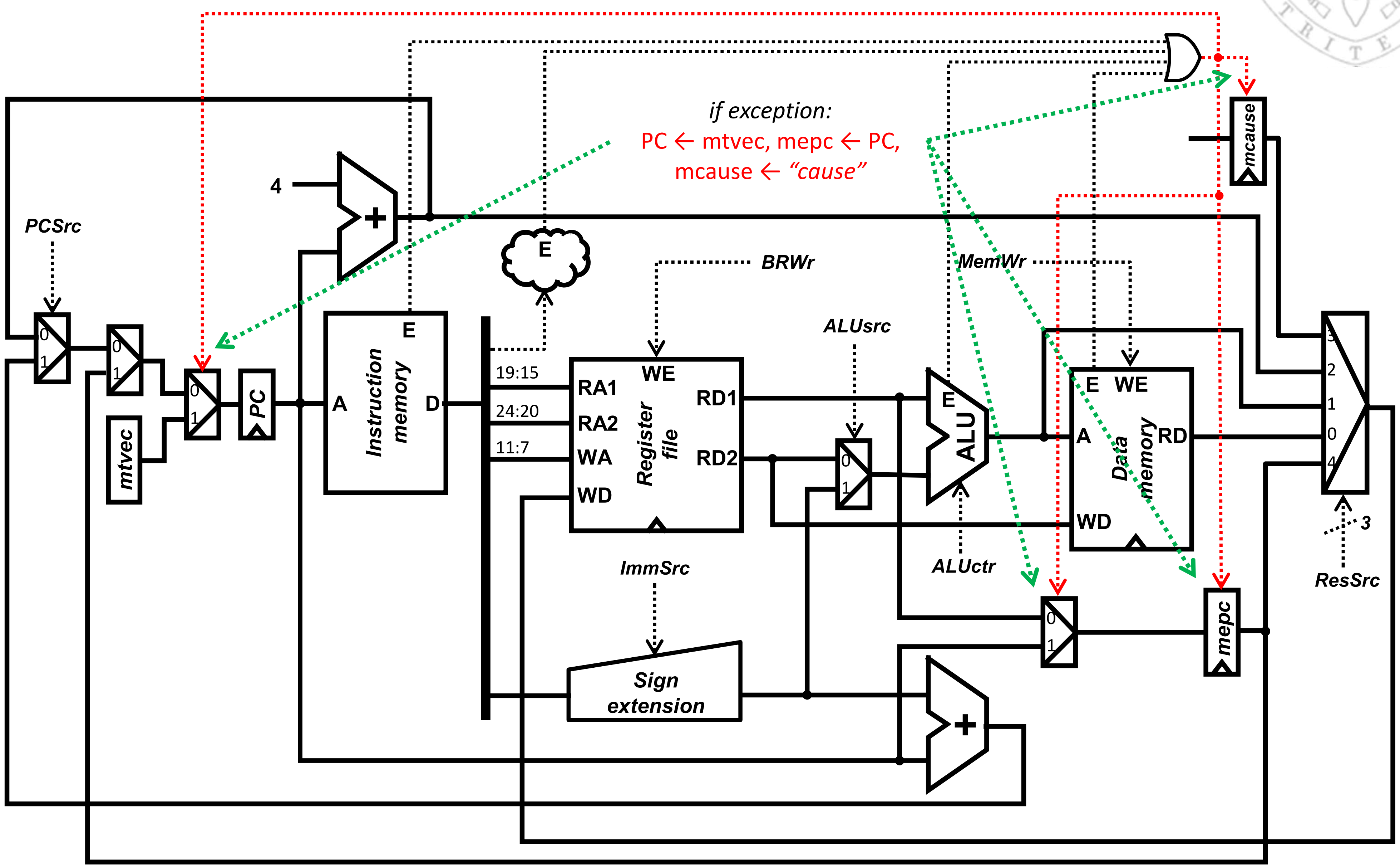
Single-cycle processor

Exception controller (ii)



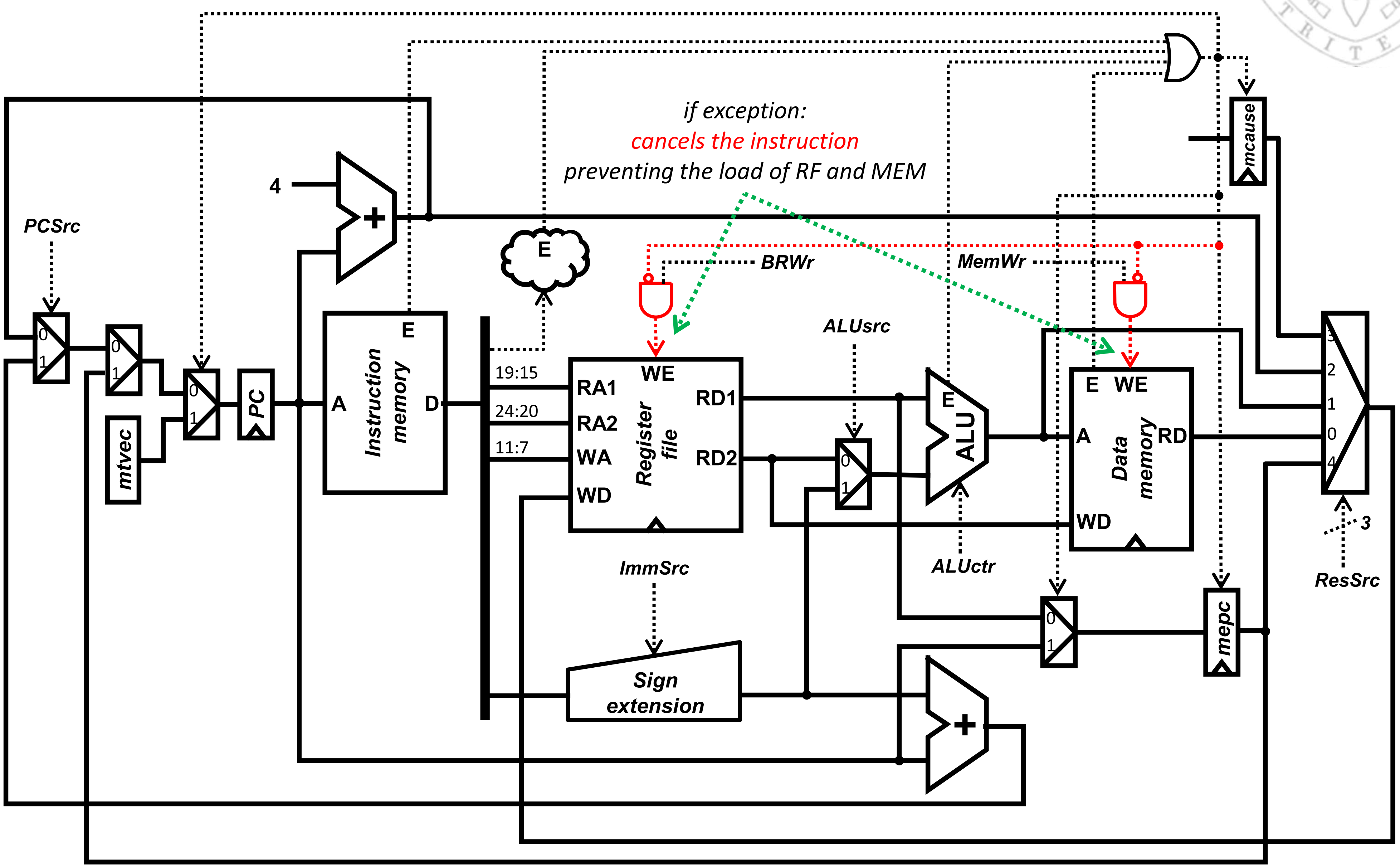
Single-cycle processor

Exception controller (iii)



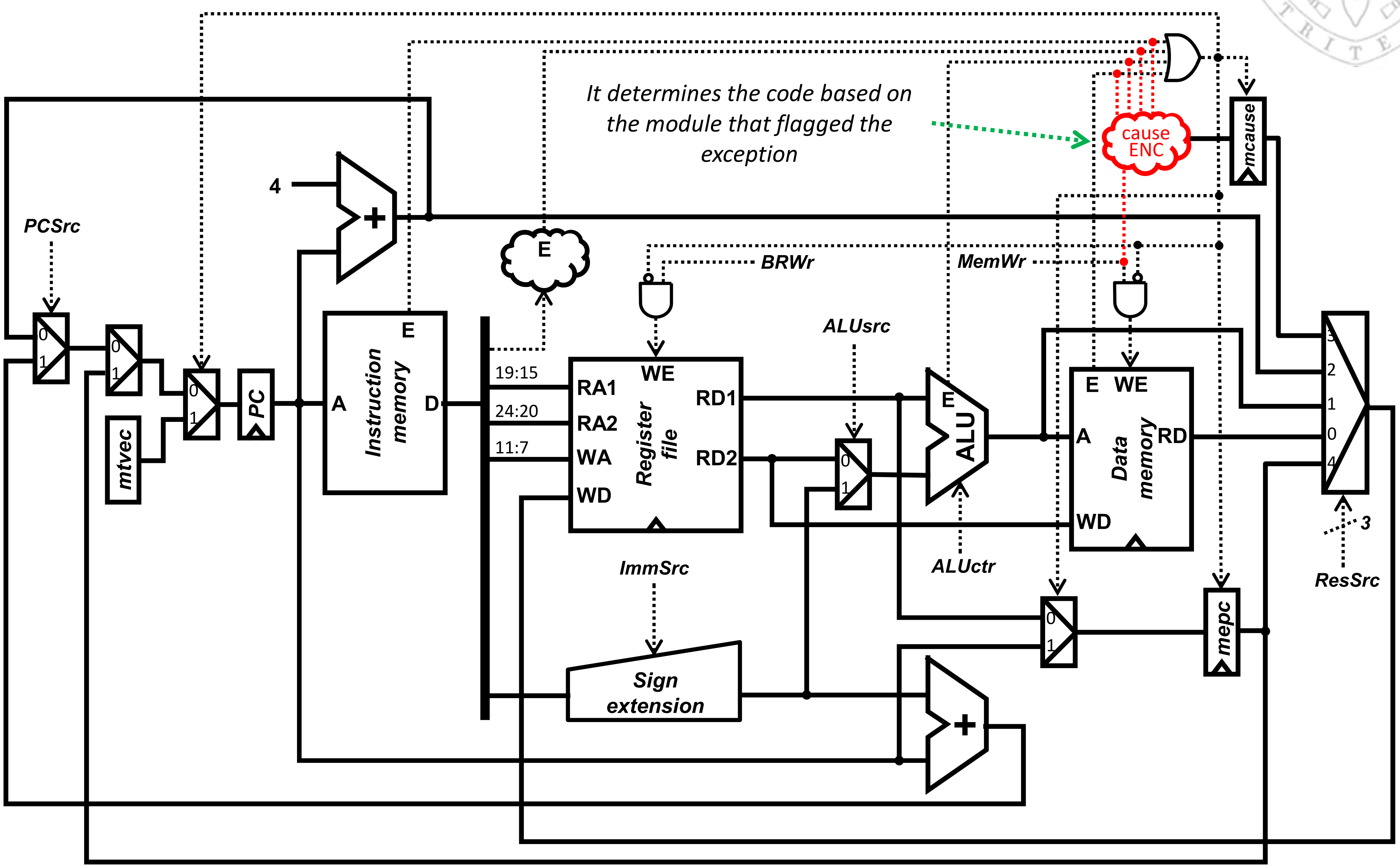
Single-cycle processor

Exception controller (iv)



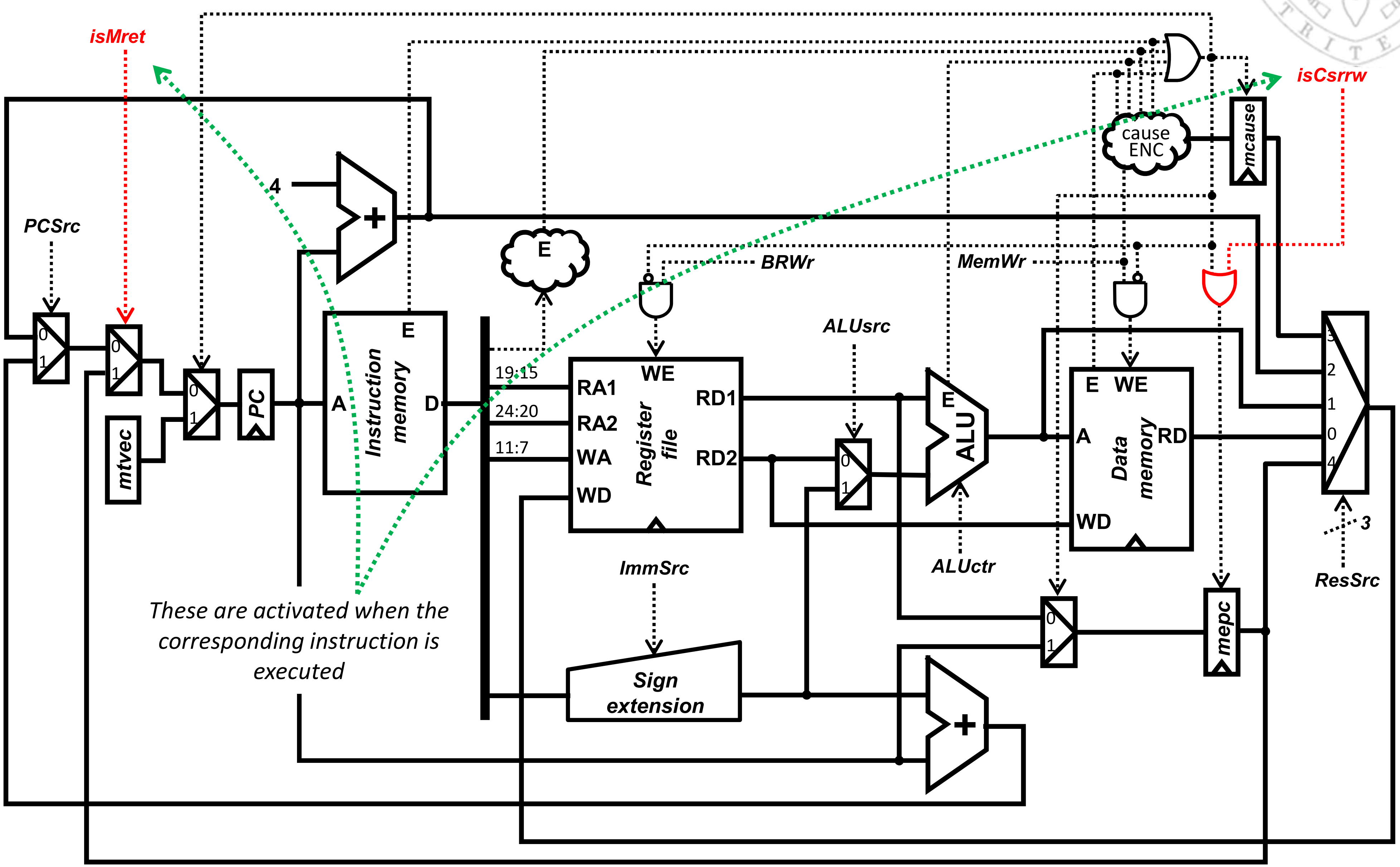
Single-cycle processor

Exception controller (v)



Single-cycle processor

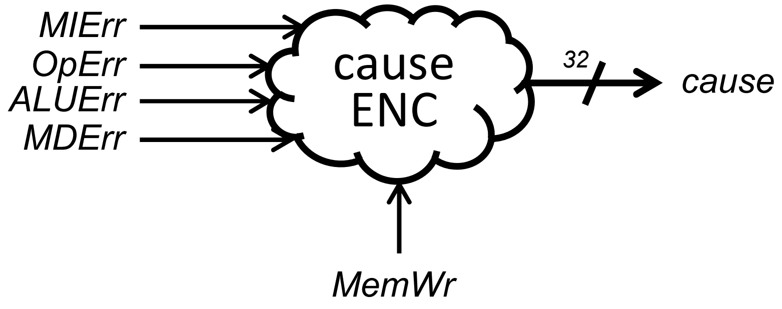
Exception controller (v)



Single-cycle processor

Exception controller: cause ENC

- This subcircuit encodes the exception cause.
  - Besides, it determines a priority among them in case a single instruction produces several exceptions.

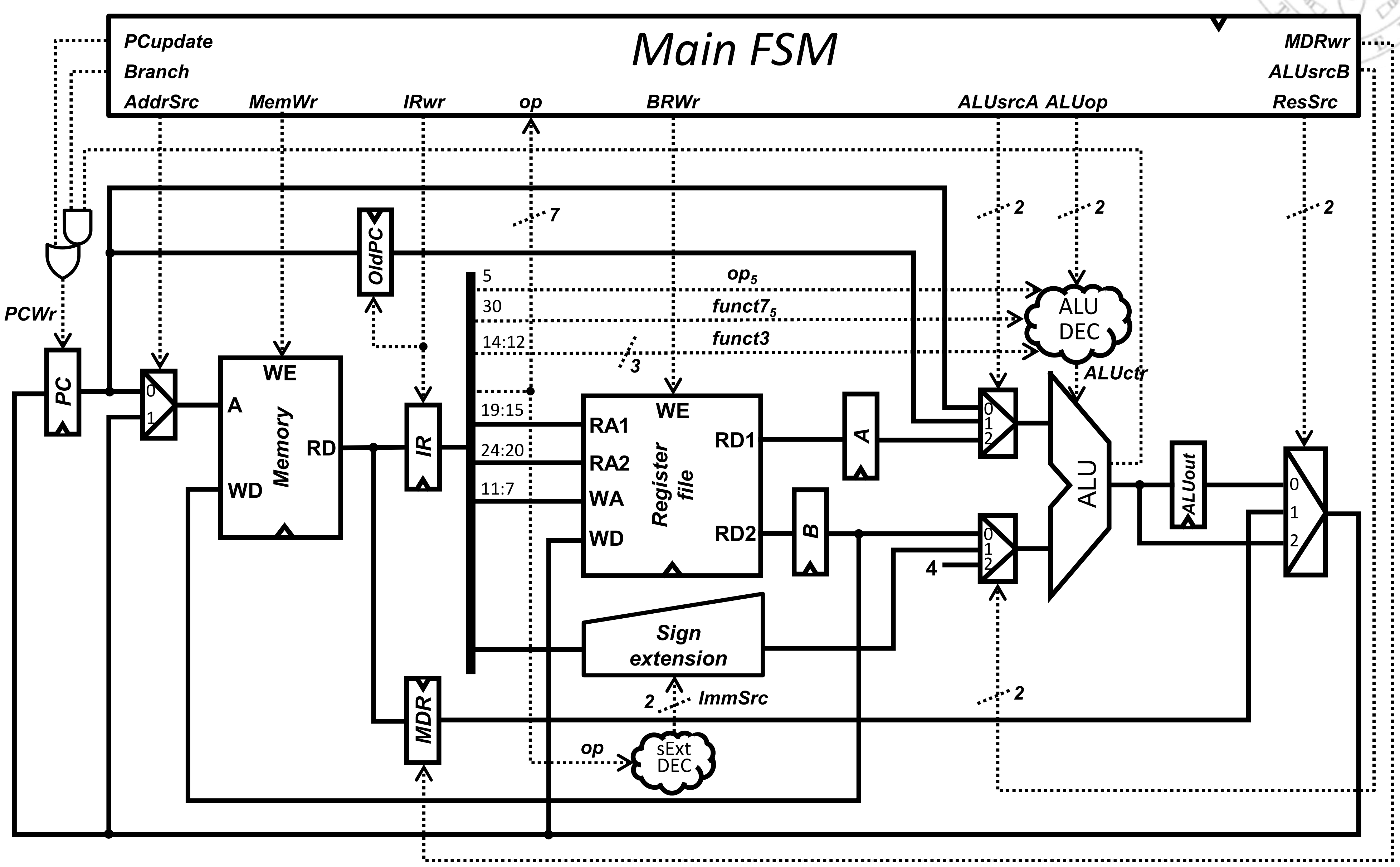


Truth table

MIErr	OpErr	ALUErr	MDErr	MemWr	cause
1	X	X	X	X	0x0000
0	1	X	X	X	0x0002
0	0	1	X	X	0x0002
0	0	0	1	0	0x0004
0	0	0	1	1	0x0006
0	0	0	0	0	–

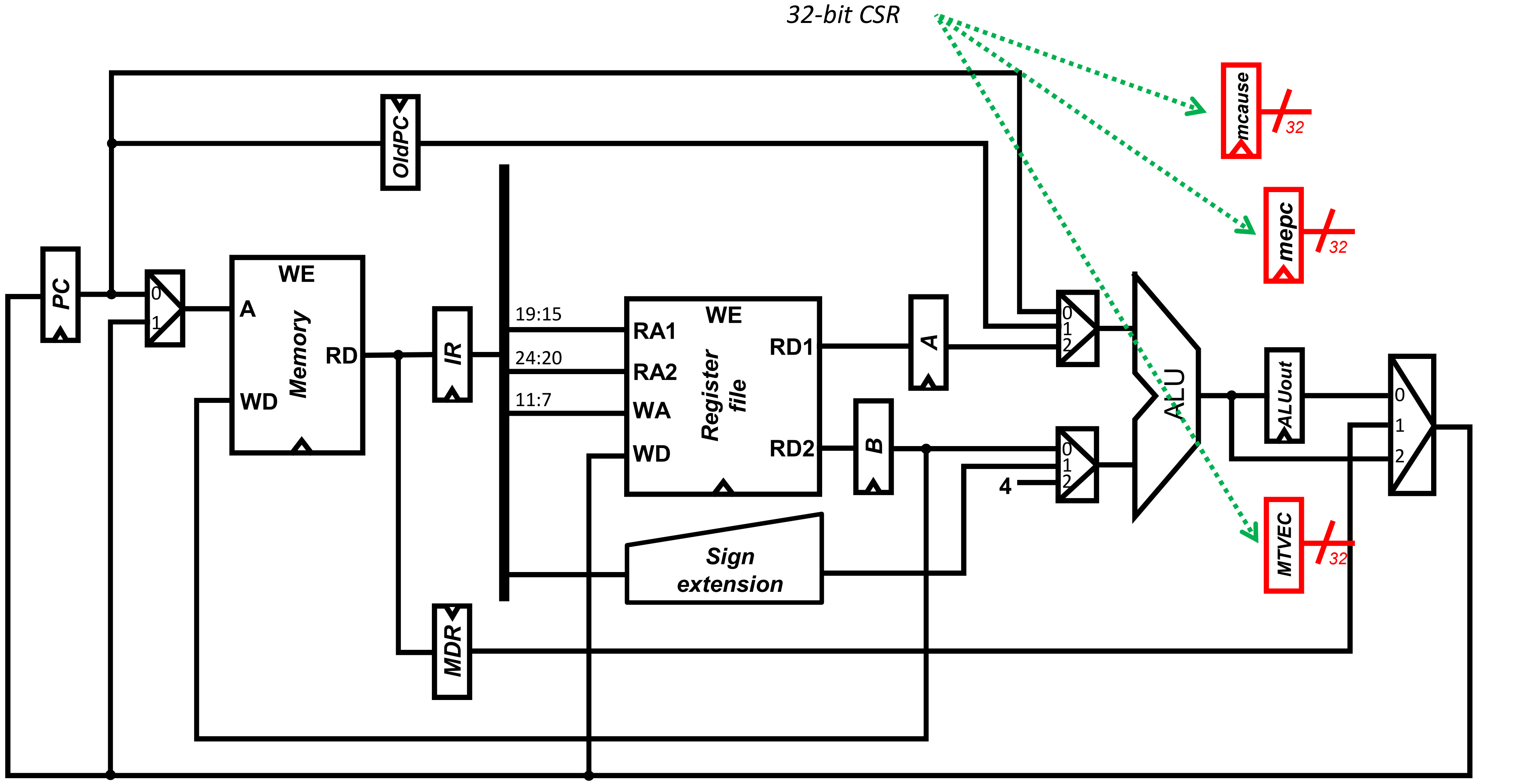
Multicycle processor

Original data path + optimized controller



Multicycle processor

Data path + exception handling (i)

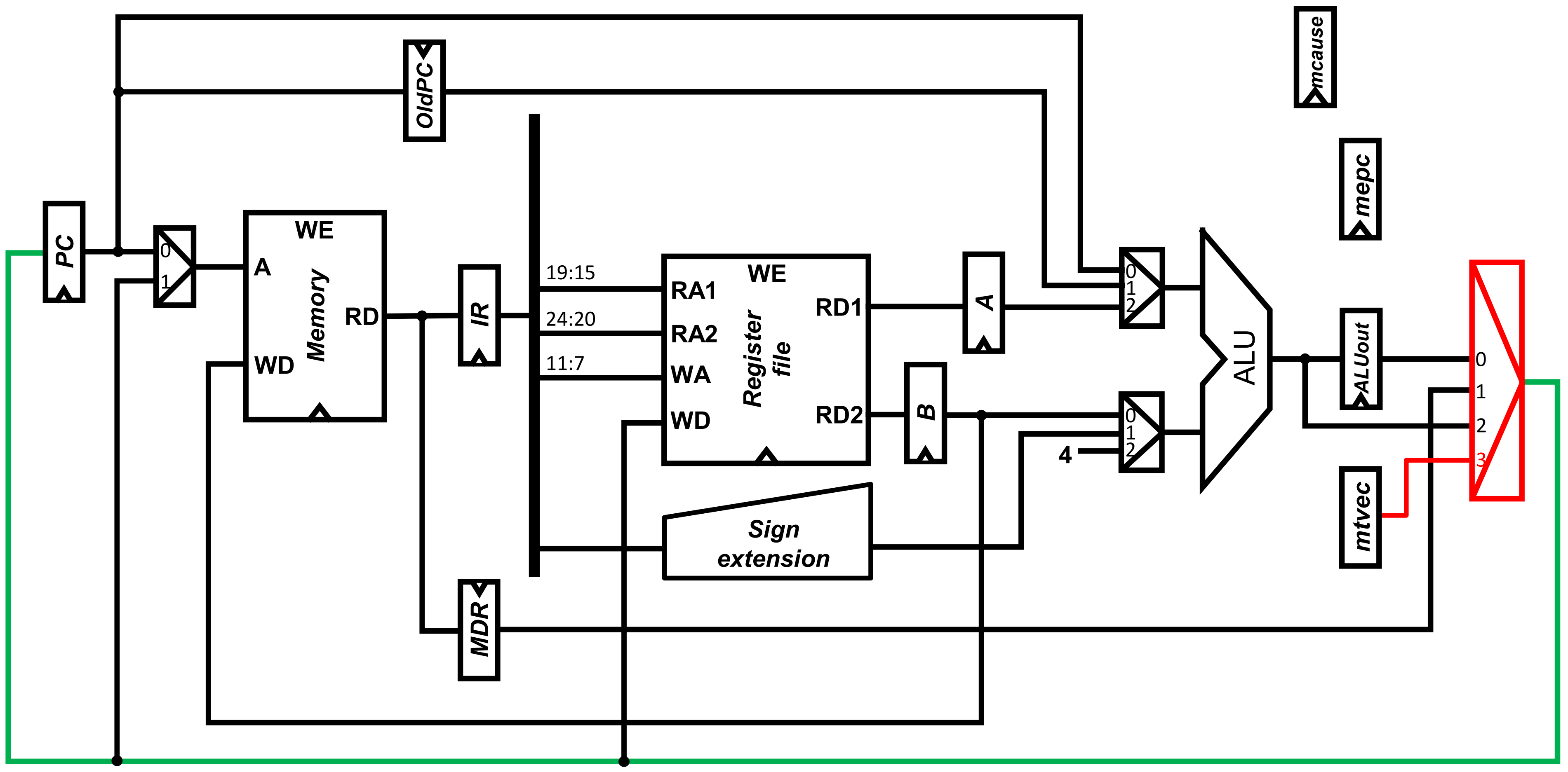


Multicycle processor

Data path + exception handling (ii)

if exception:

PC ← mtvec, mepc ← PC, mcause ← "*cause*"

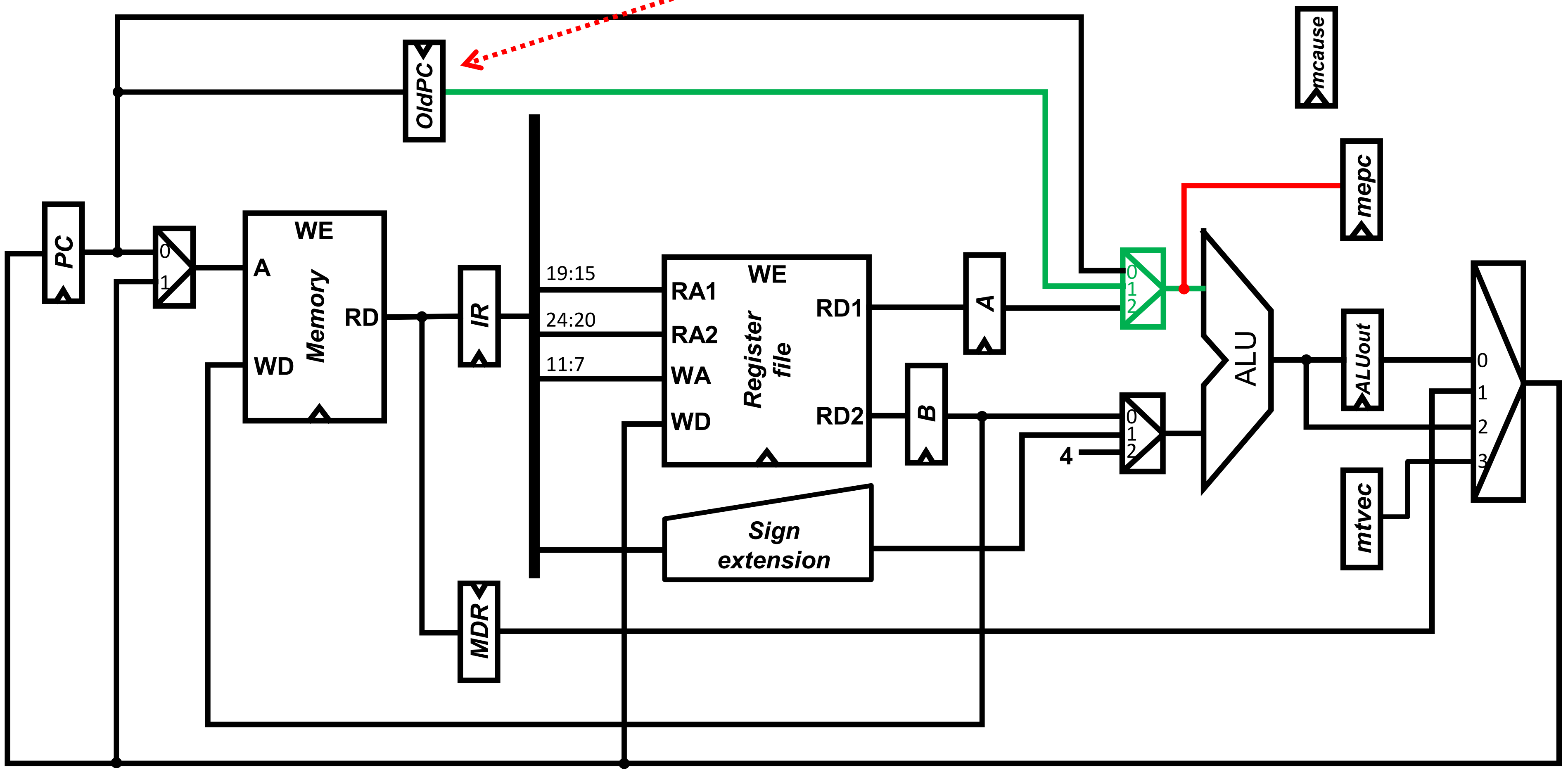


Multicycle processor

Data path + exception handling (iii)

if exception:
PC ← mtvec, mepc ← PC, mcause ← "*cause*"

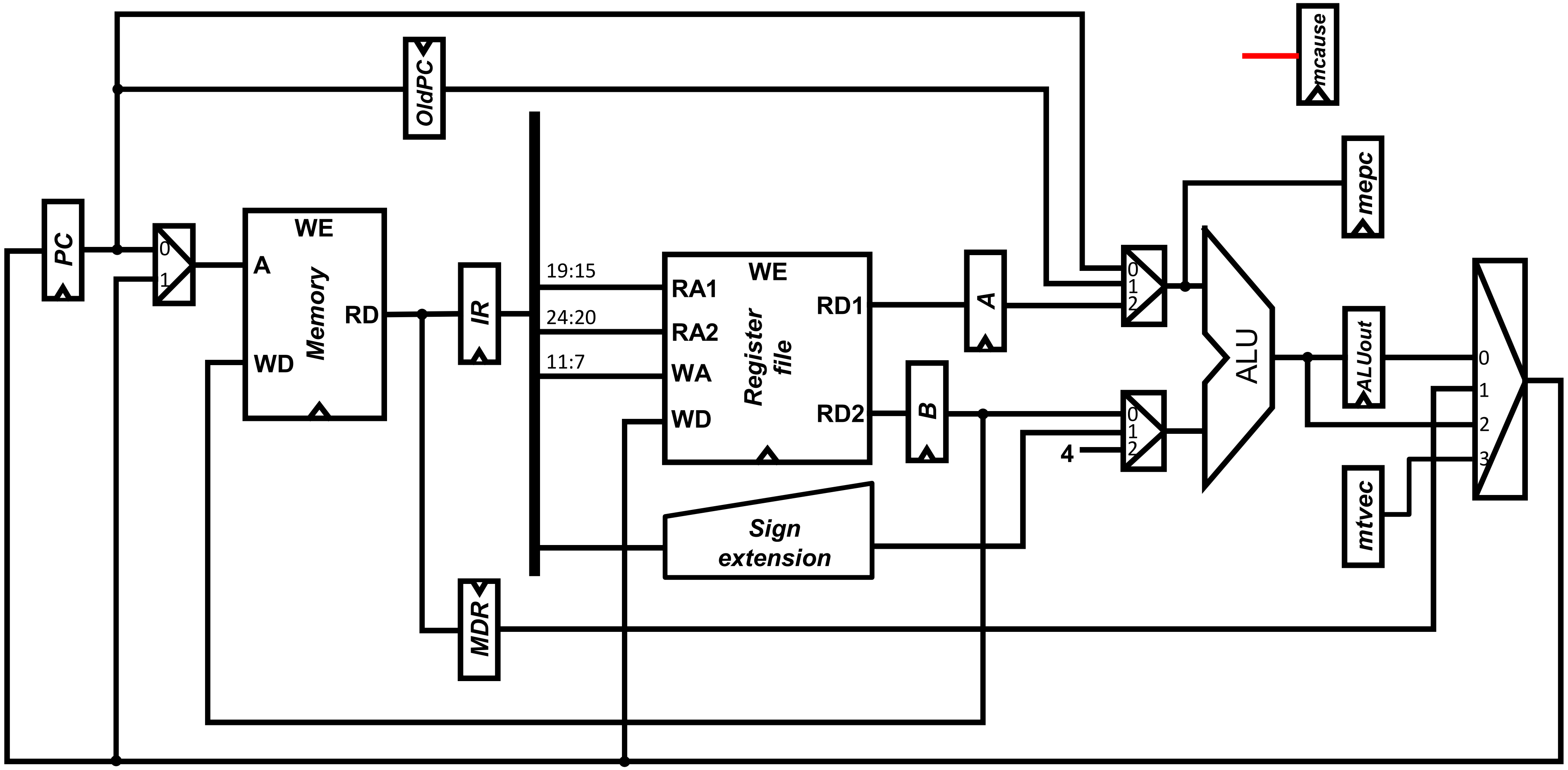
The address of the instruction in execution is stored in OldPC



Multicycle processor

Data path + exception handling (iv)

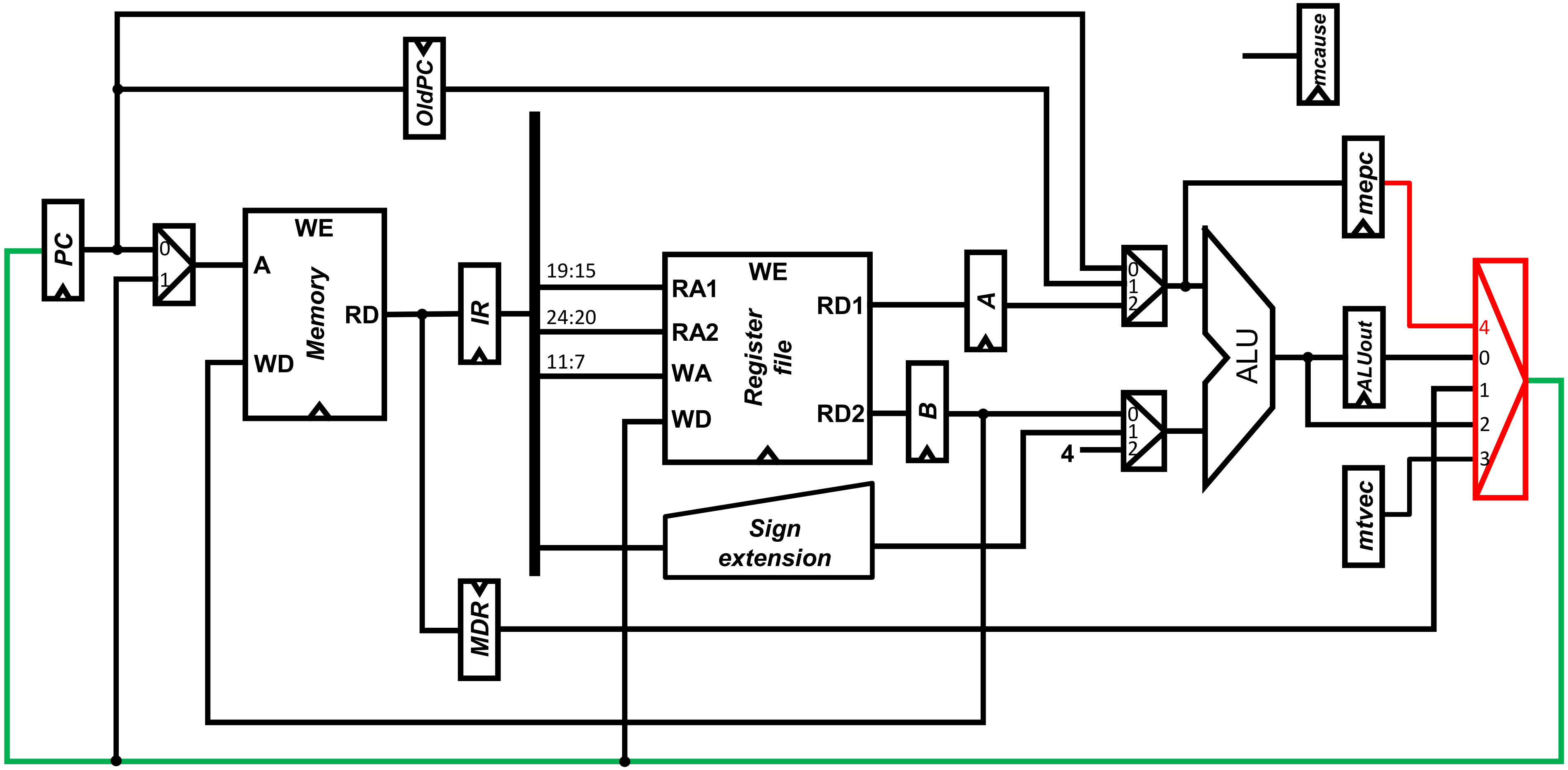
if exception:
PC ← mtvec, mepc ← PC, mcause ← "*cause*"



Multicycle processor

Data path + exception handling (v)

```
mret
```
PC ← mepc

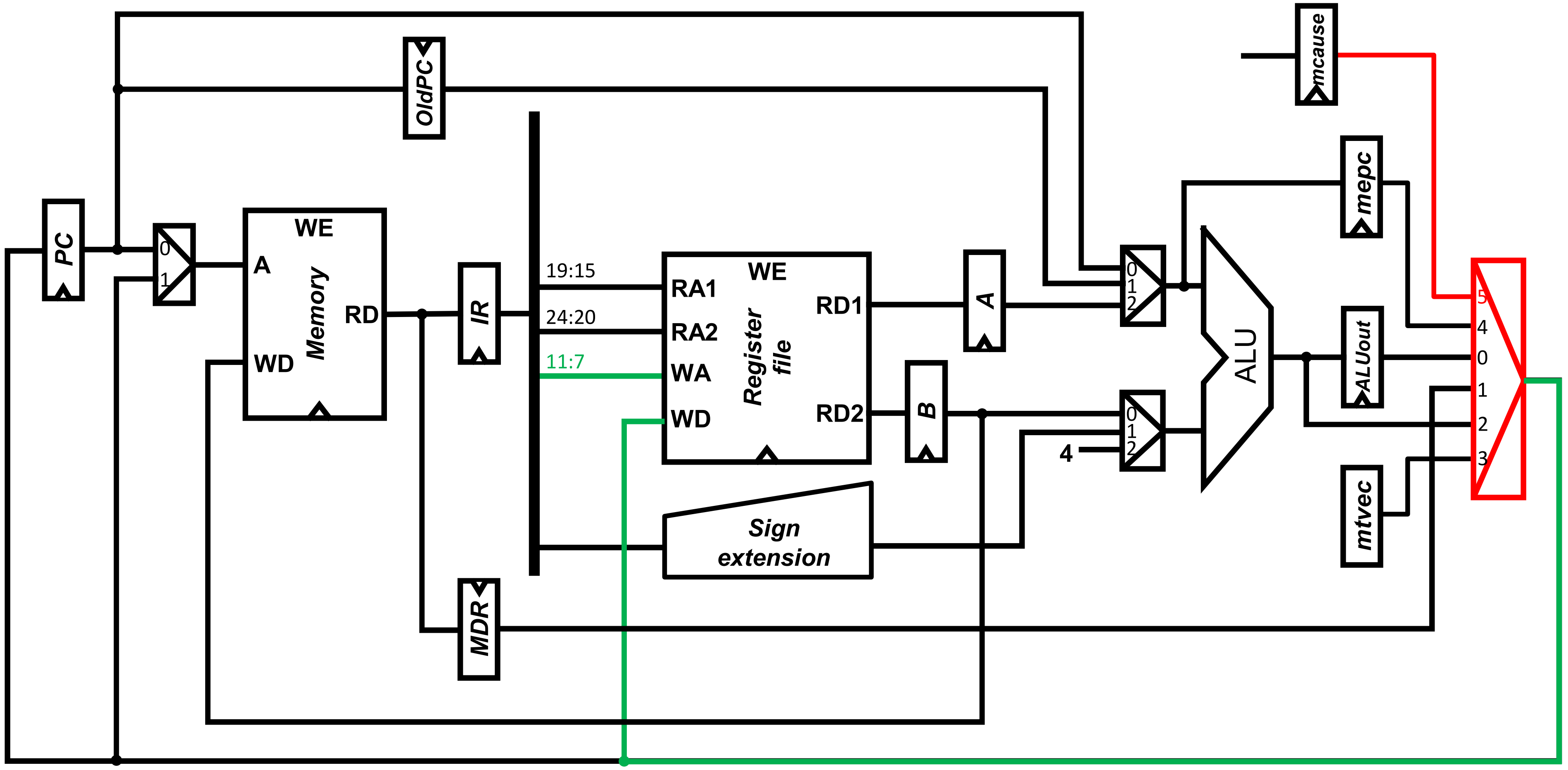


Multicycle processor

Data path + exception handling (vi)

```
csrrw rd, mcause, rs1
```

RF[rd] ← mcause, PC ← PC+4

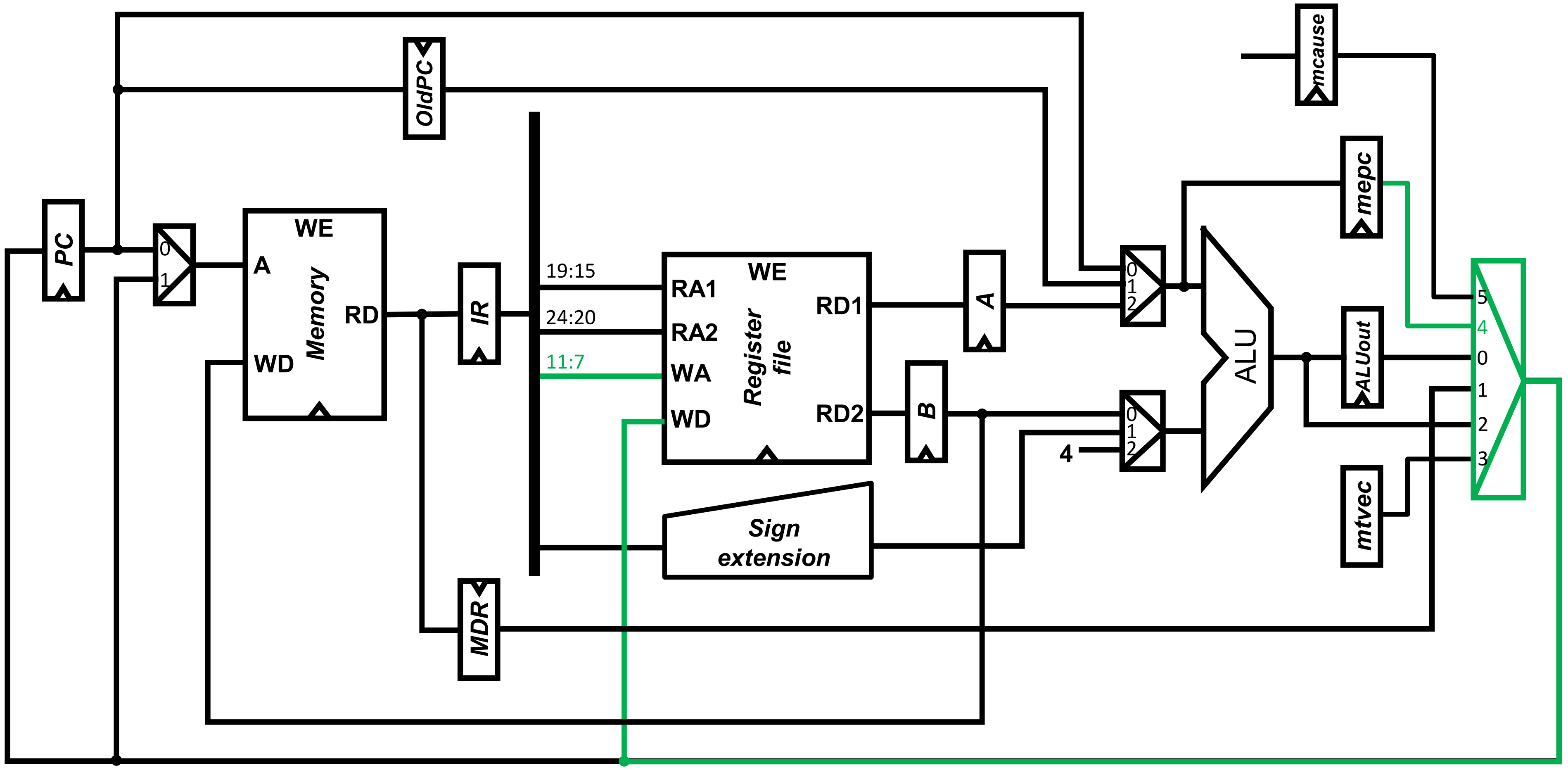


Multicycle processor

Data path + exception handling (vii)

```
csrrw rd, mepc, rs1
```

RF[rd] ← mepc, mepc ← RF[rs1], PC ← PC+4



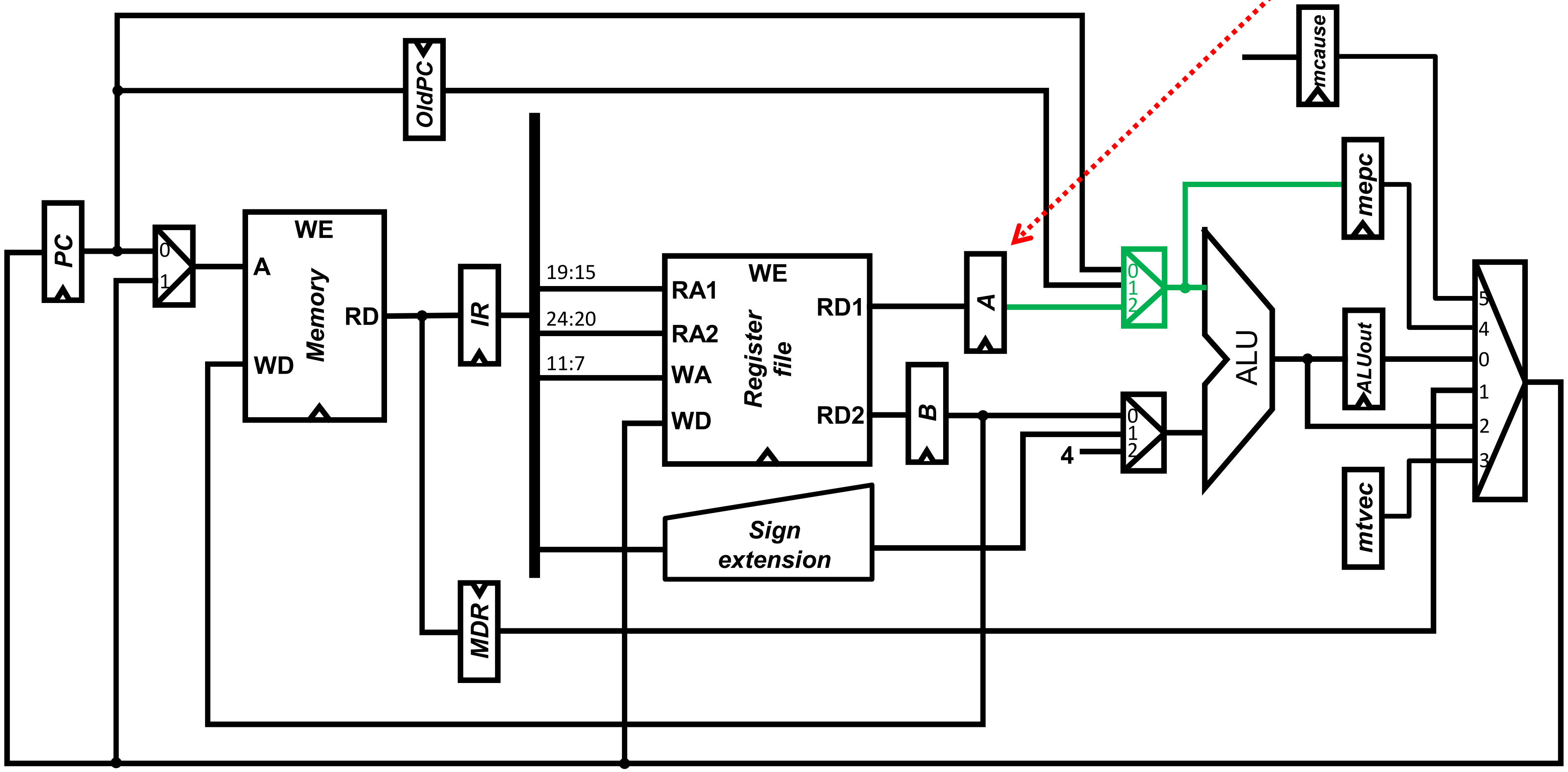
Multicycle processor

Data path + exception handling (viii)

```
csrrw rd, mepc, rs1
```

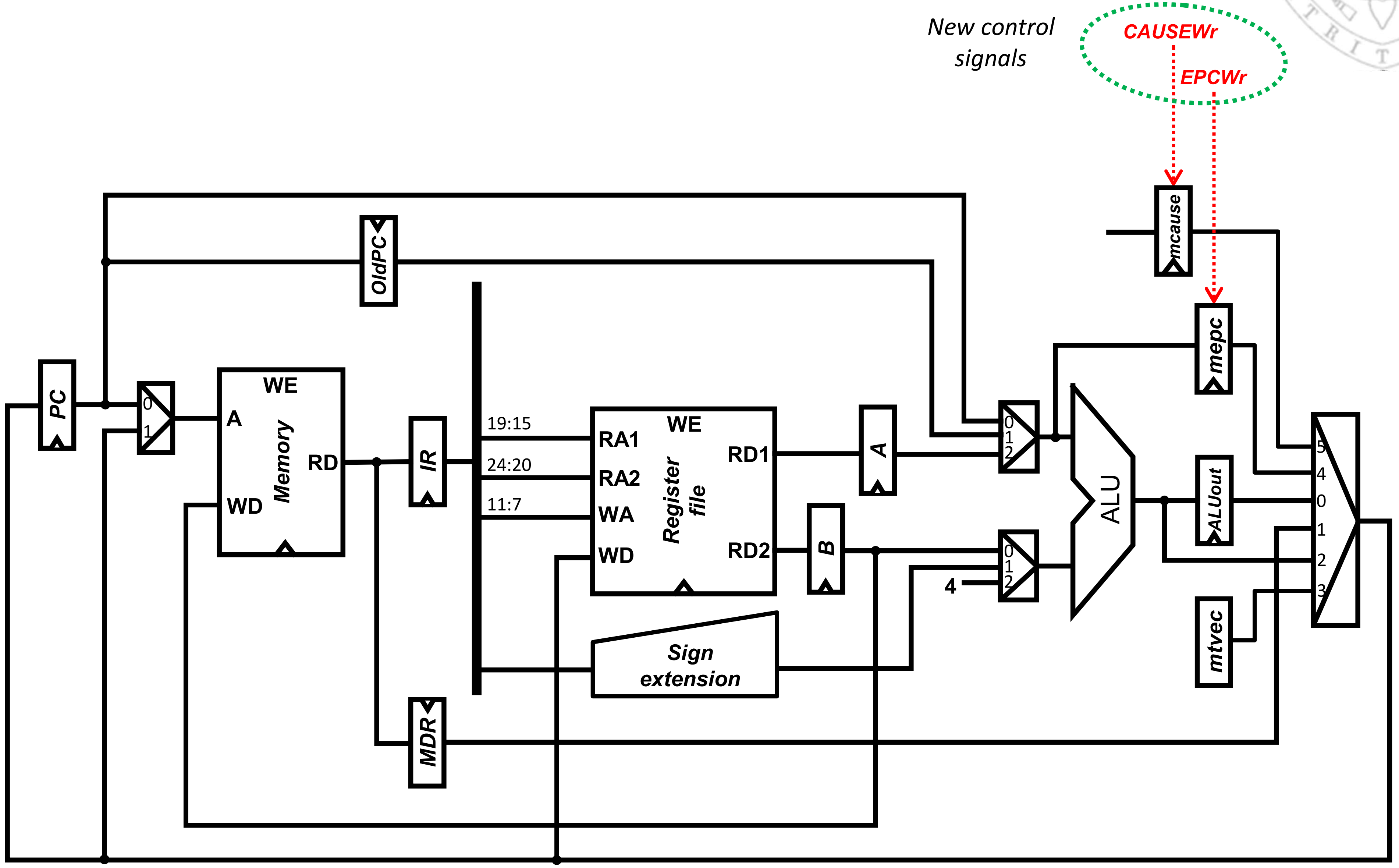
RF[rd] ← mepc, mepc ← RF[rs1], PC ← PC+4

the source operand is stored in A



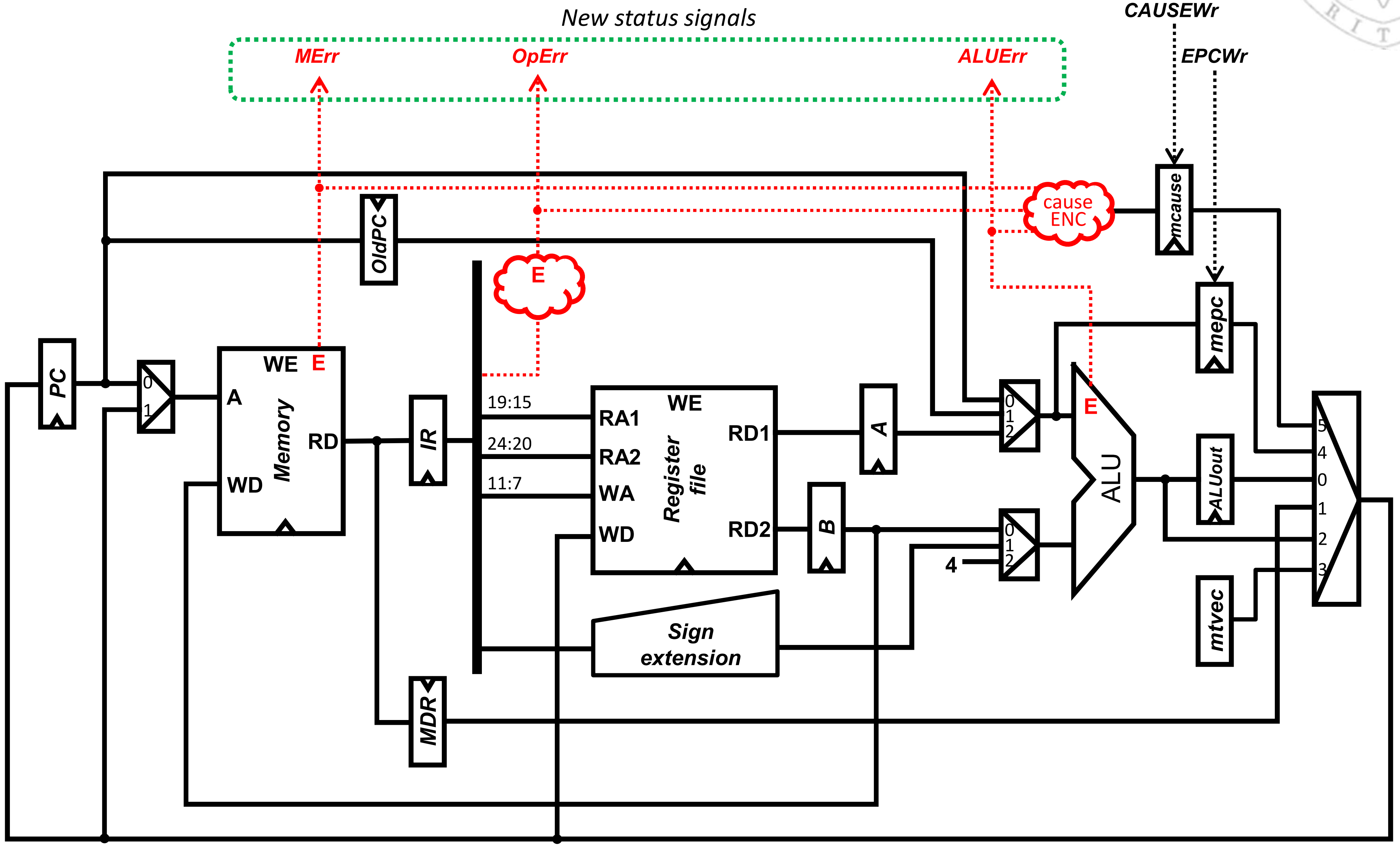
Multicycle processor

Data path + exception handling (ix)



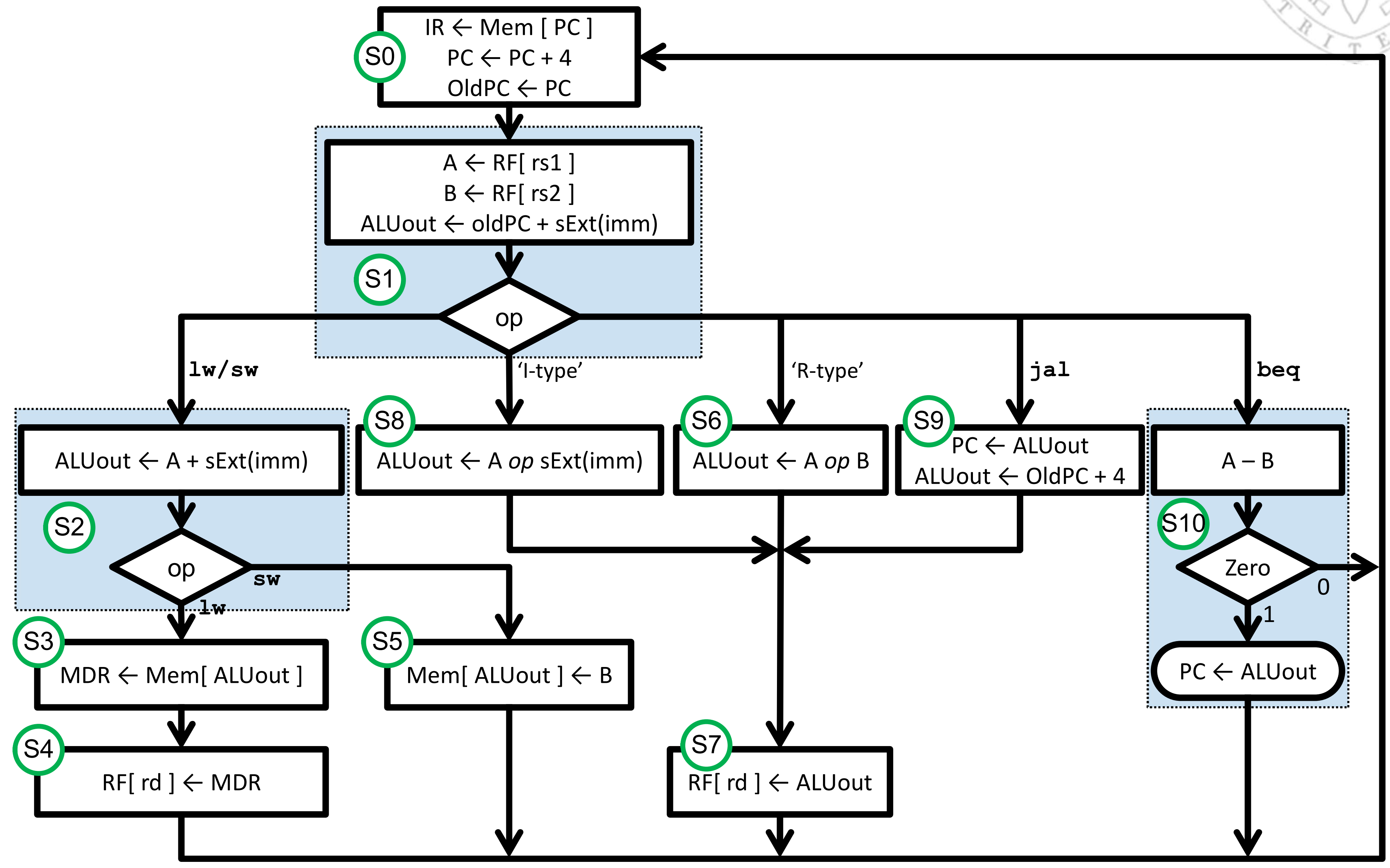
Multicycle processor

Data path + exception handling (x)



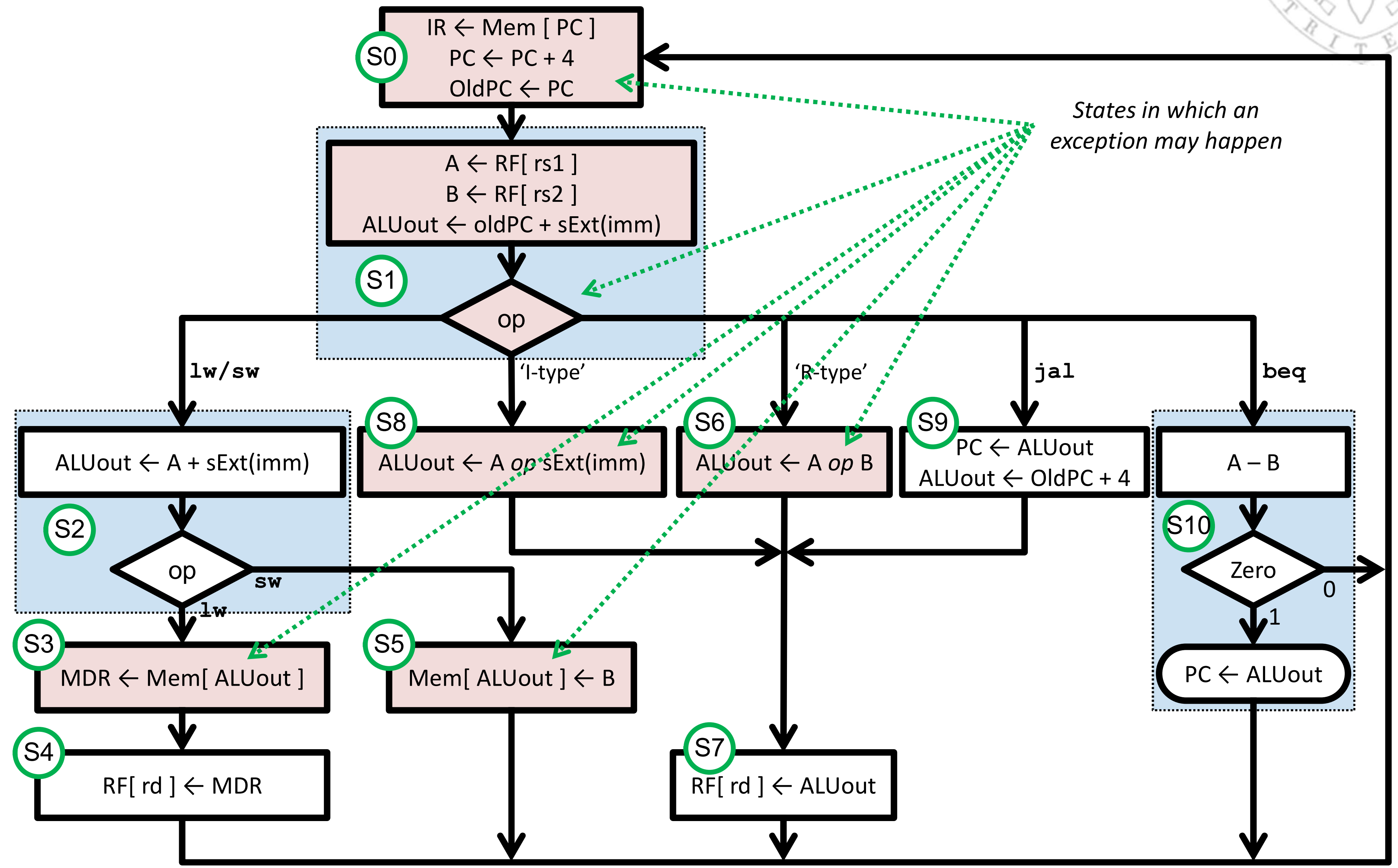
Multicycle processor

Original ASM diagram of the main FSM (i)



Multicycle processor

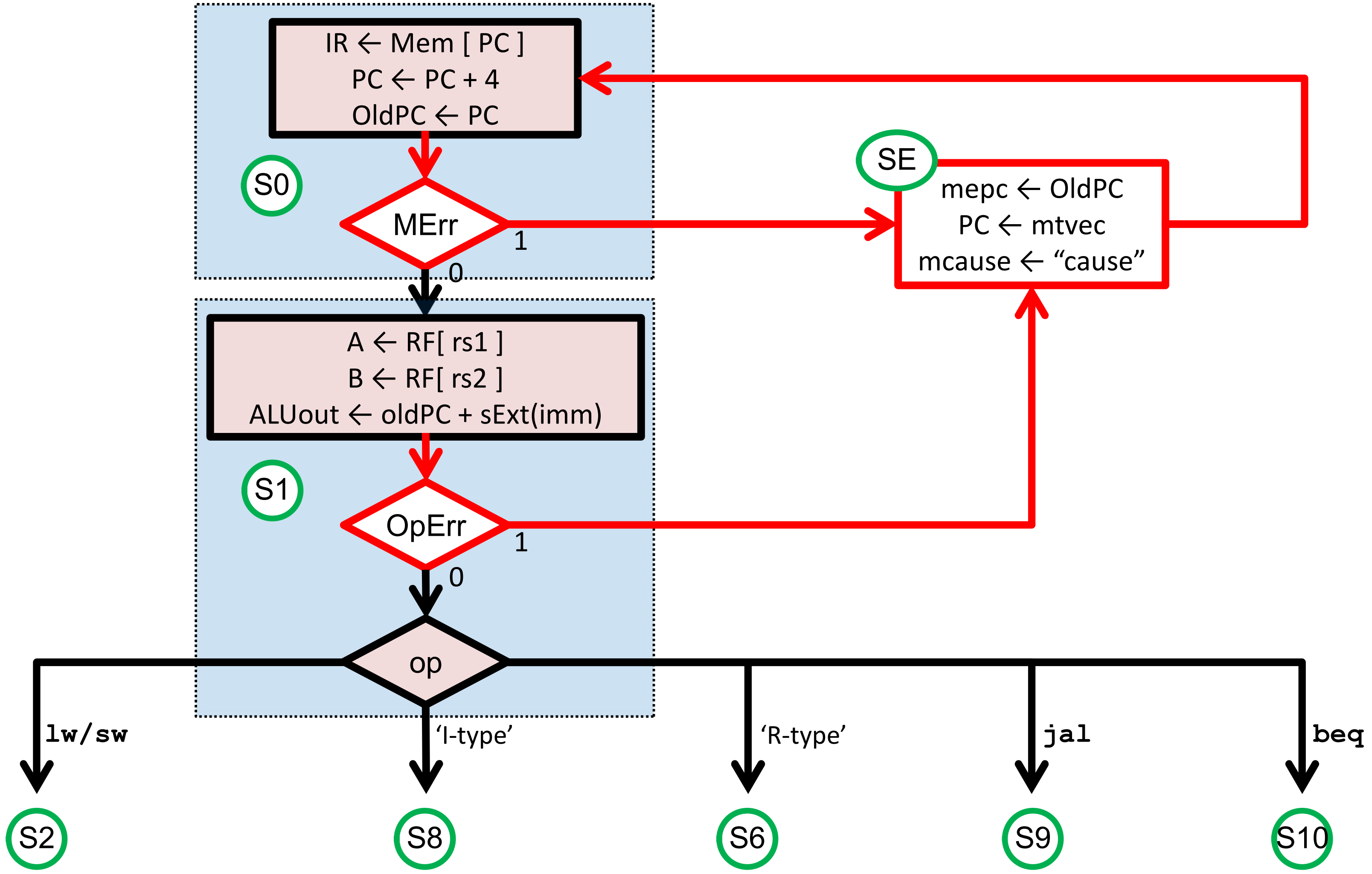
Original ASM diagram of the main FSM (i)



Multicycle processor

ASM of the main FSM + exception handling (i)

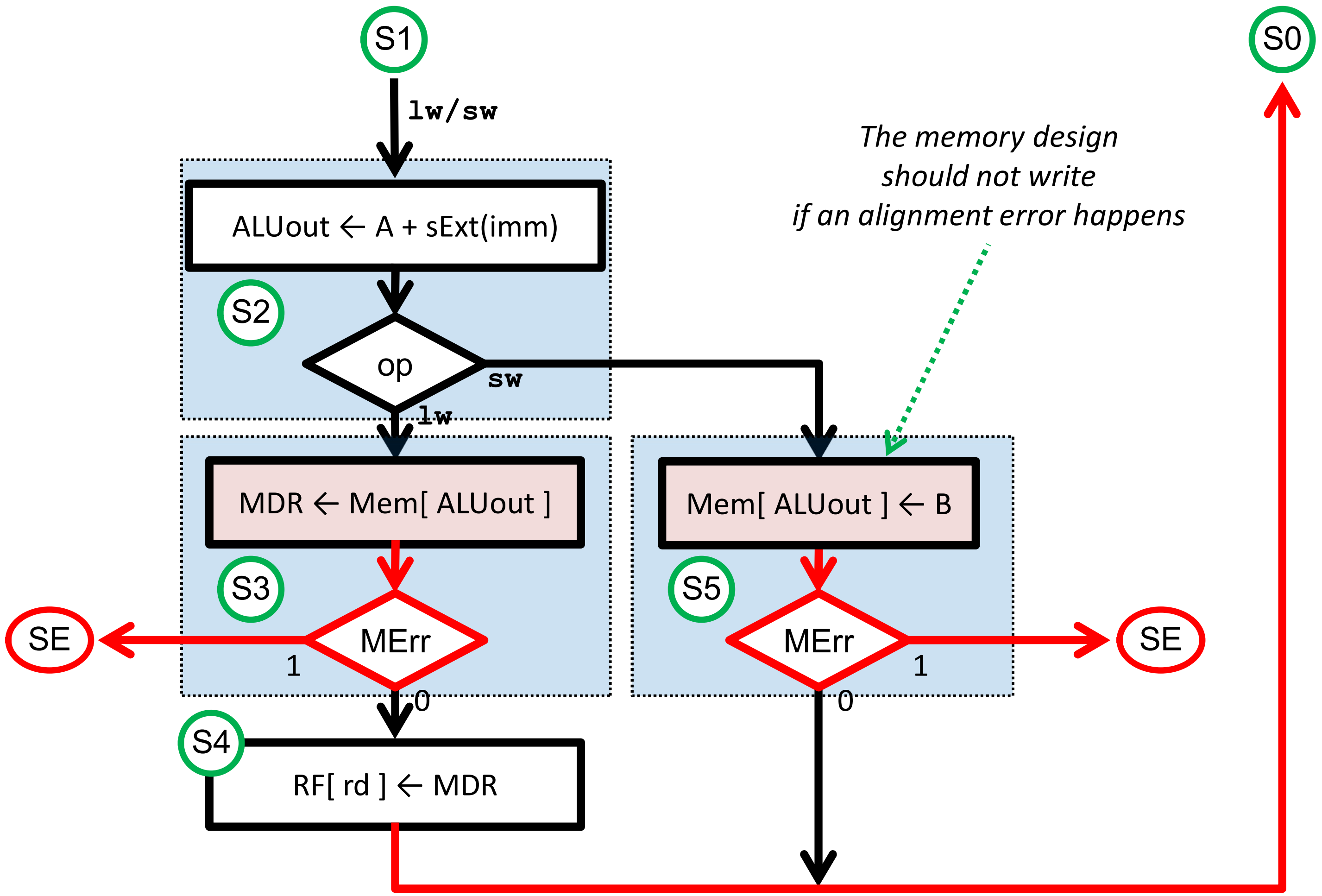
- The SE state is added to handle exceptions.
  - SE updates CSR and cancels the current instruction branching to S0.



Multicycle processor

ASM of the main FSM + exception handling (ii)

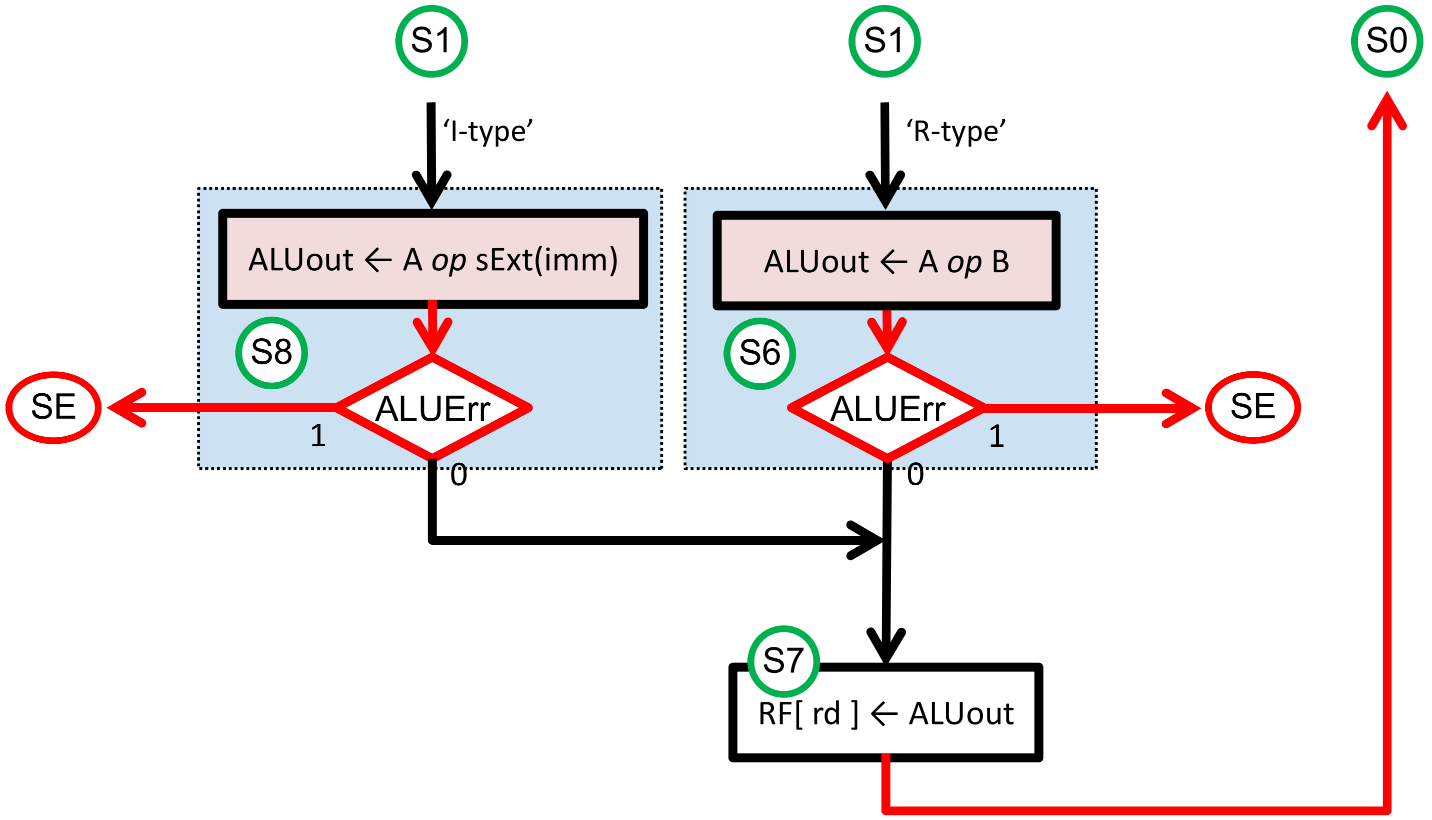
- All the states in which an exception may happen have to check the corresponding status signal and decide whether to branch to SE or not.



Multicycle processor

ASM of the main FSM + exception handling (iii)

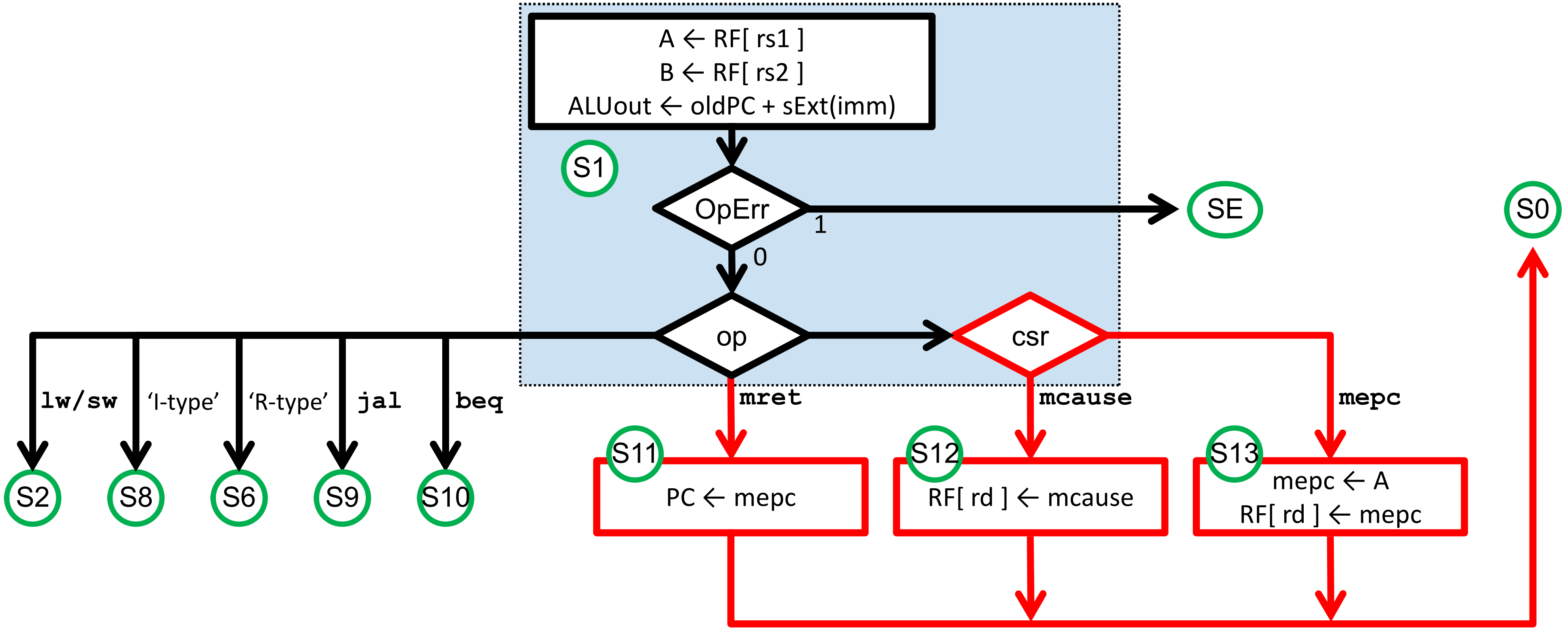
- All the states in which an exception may happen have to check the corresponding status signal and decide whether to branch to SE or not.



Multicycle processor

ASM of the main FSM + exception handling (iv)

- Some states are added to support the new instructions:
  - None of them may trigger an exception.



Pipelined processor

Previous considerations

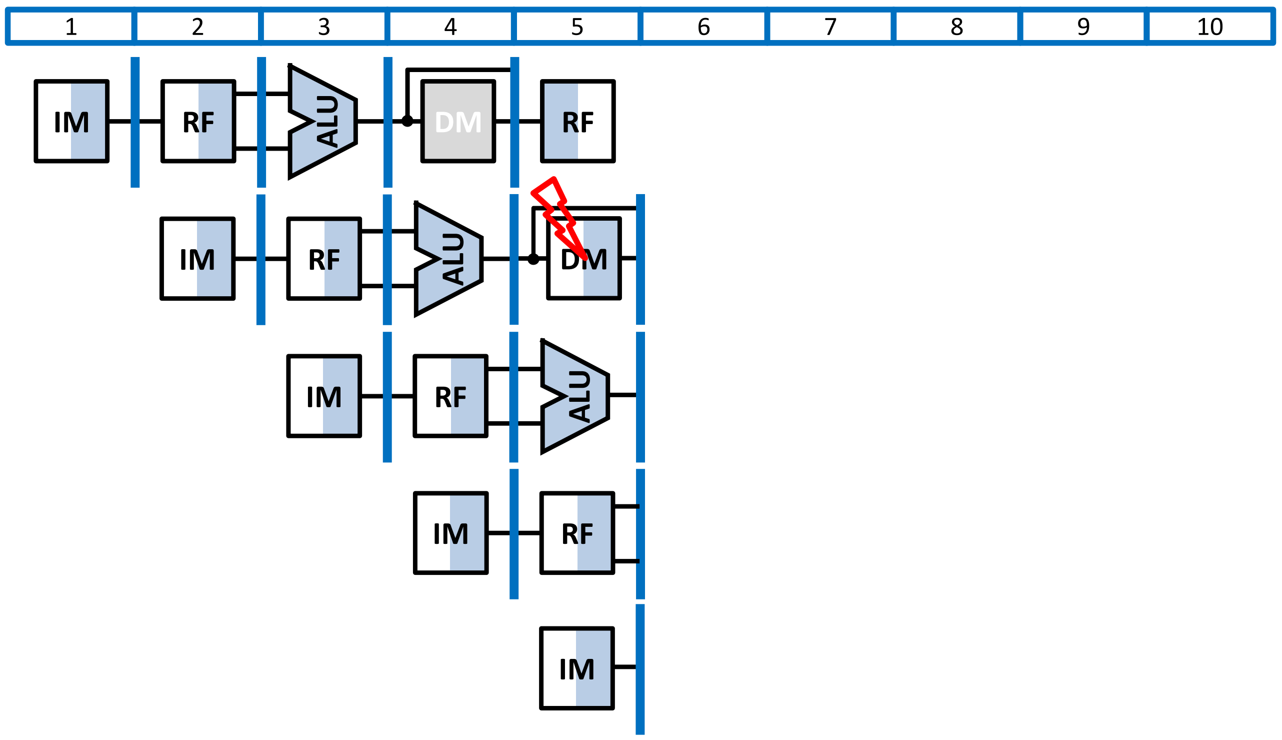
- Exceptions are taken branches to an implicit address. Therefore, RISC-V handles them in a similar way to the control hazards:
  - The instructions previous to the one that produced the exception are completed as usual.
  - The instruction that produces the exception, and the ones fetched after it, are flushed.
  - The instruction located in the implicit address is fetched.
  - But, contrary to the branches that happen in the EX stage, exceptions are handled when the instruction causing it is in the MEM stage.

Pipelined processor

Simple exception (i)

- The misaligned data access exception will be handled as followed:
  - Cycle 5: **`lw`** (in MEM) generates an exception.

```
0000d400        sub x2, x1, x3
0000d404        lw  x2, 20(x1)
0000d408        or  x7, x5, x2
0000d40c        and x6, x2, x4
0000d410        add x8, x1, x3
```



Pipelined processor

Simple exception (i)

- The misaligned data access exception will be handled as followed:
  - Cycle 5: **`lw`** (in MEM) generates an exception. **`lw`** (and the following) are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 4

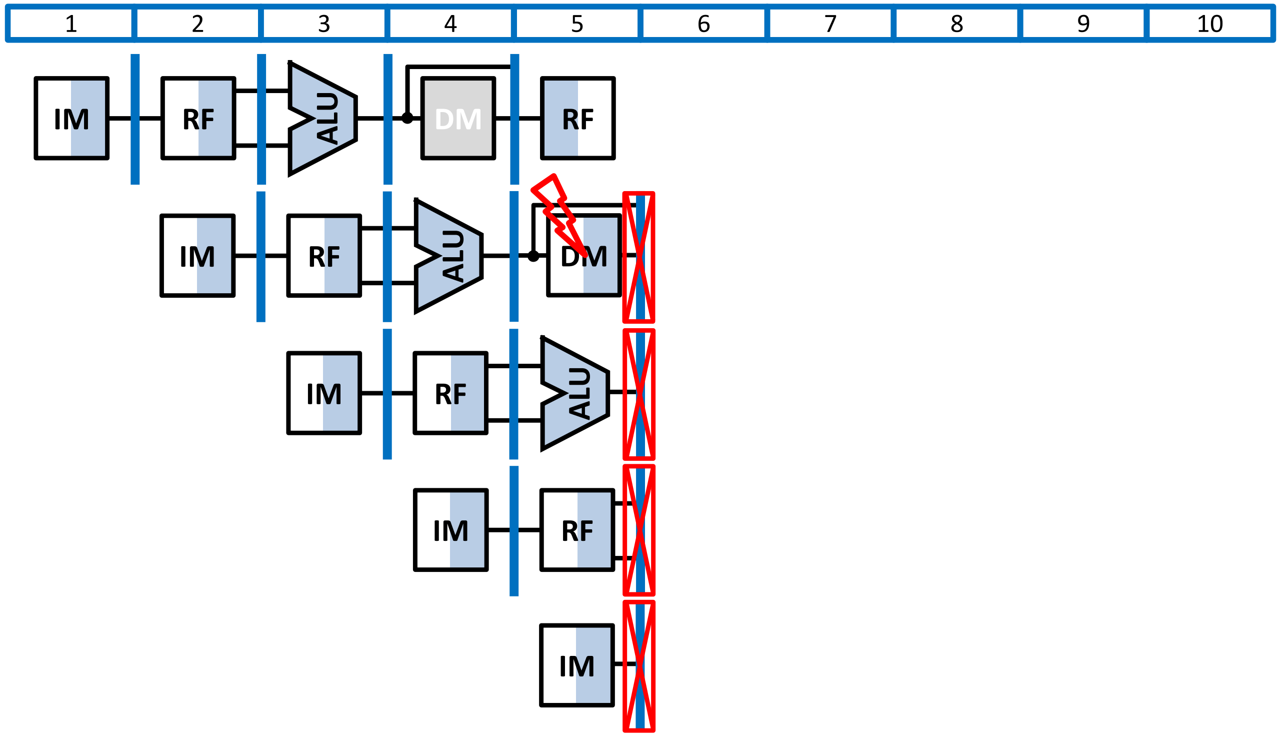
```
0000d400    sub x2, x1, x3
```

0000d404 ~~`lw x2, 20(x1)`~~

0000d408 ~~`or x7, x5, x2`~~

0000d40c ~~`and x6, x2, x4`~~

0000d410 ~~`add x8, x1, x3`~~

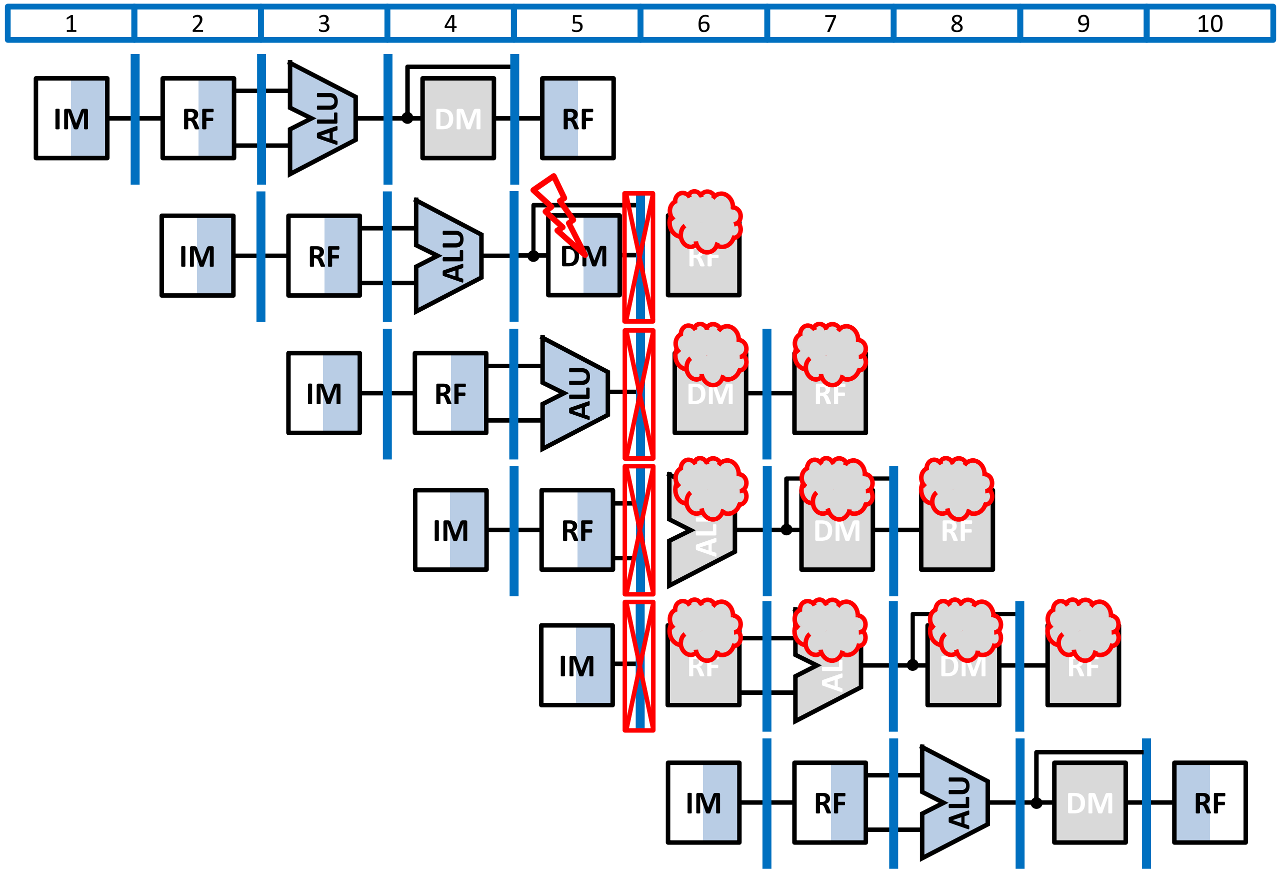


Pipelined processor

Simple exception (i)

- The misaligned data access exception will be handled as followed:
  - Cycle 5: **lw** (in MEM) generates an exception. **lw** (and the following) are flushed.
    - **mepc** = 0x0000d404 and **cause** = 4
  - Cycle 6: the instruction whose address is in **mtvec** is fetched.

```
0000d400        sub x2, x1, x3
0000d404        ~~lw  x3, 20(x1)~~
0000d408        ~~or  x7, x5, x2~~
0000d40c        ~~and x6, x2, x4~~
0000d410        ~~add x8, x1, x3~~
1c000000  ESR:  add sp, sp, -8
```

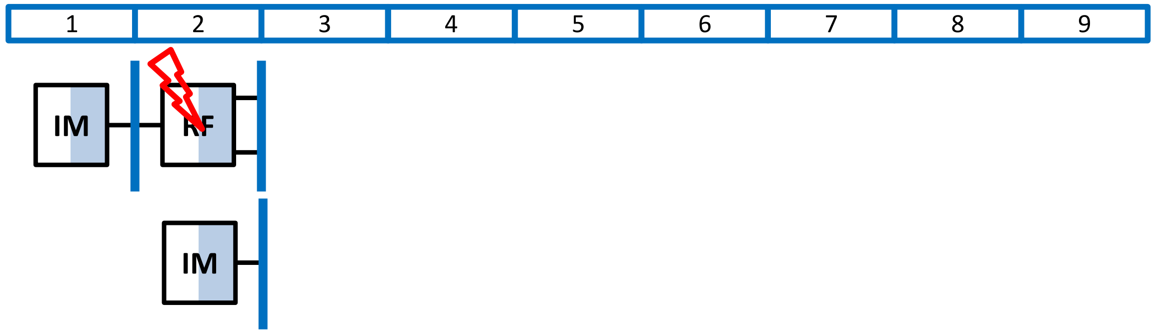


Pipelined processor

Simple exception (ii)

- Regardless of when the exception happens, it will always be handled when the instruction producing it arrives at MEM:
  - Cycle 2: **`mul`** (in ID) generates an exception.

```
0000d404        mul  x6, x2, x4
0000d408        or   x7, x5, x2
```

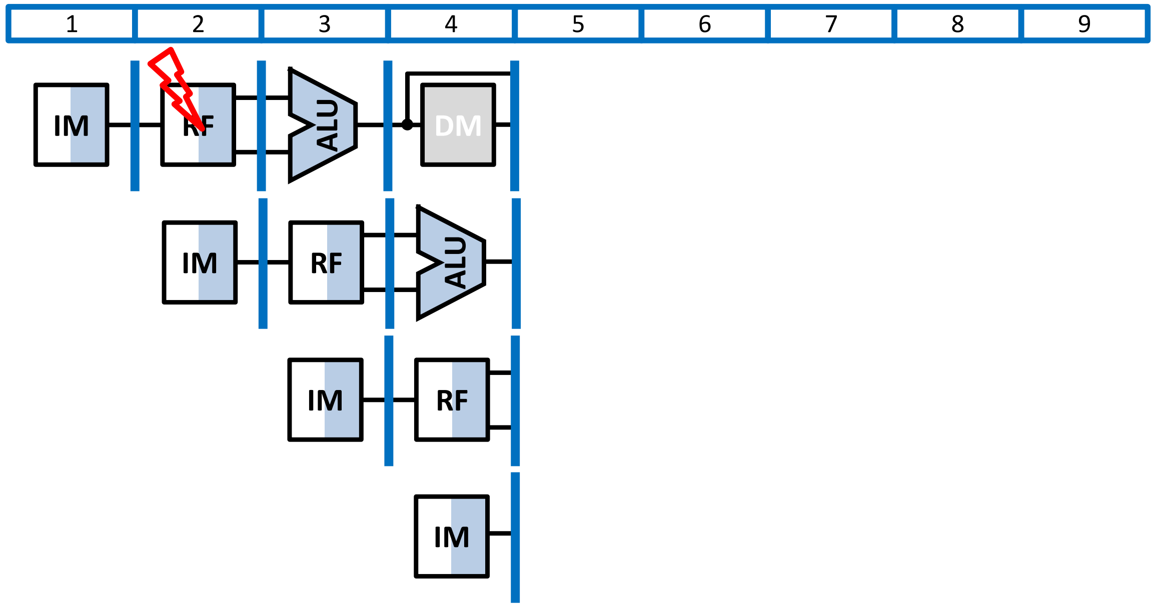


Pipelined processor

Simple exception (ii)

- Regardless of when the exception happens, it will always be handled when the instruction producing it arrives at MEM:
  - Cycle 2: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`mul`** (in MEM).

```
0000d404     mul x6, x2, x4
0000d408     or  x7, x5, x2
0000d40c     and x4, x1, x3
0000d410     sub x2, x1, x3
```

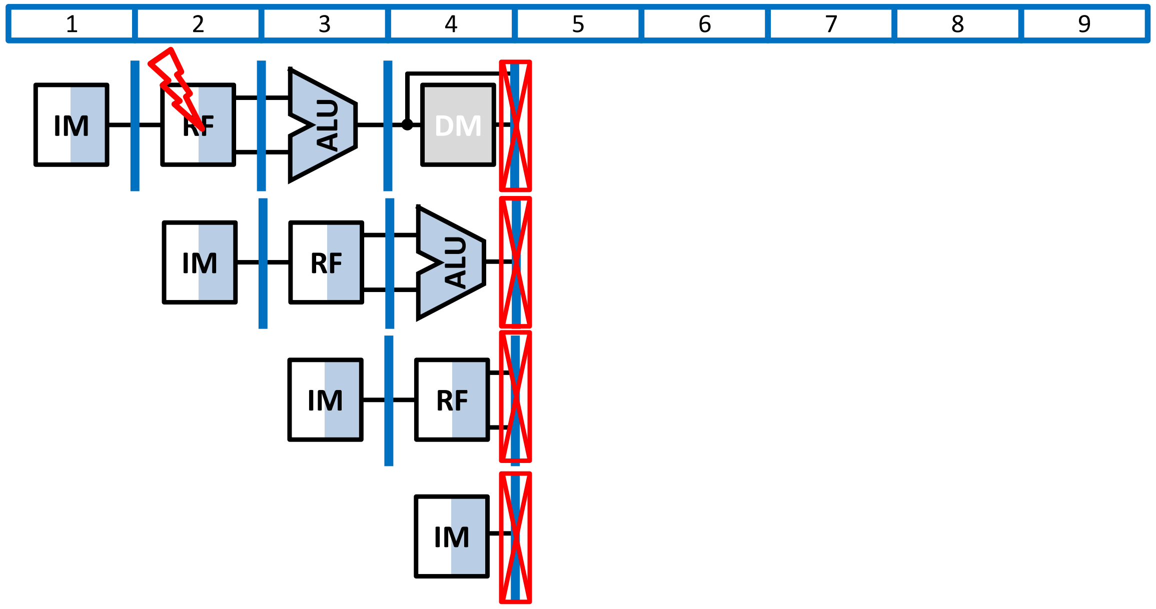


Pipelined processor

Simple exception (ii)

- Regardless of when the exception happens, it will always be handled when the instruction producing it arrives at MEM:
  - Cycle 2: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`mul`** (in MEM). **`mul`** (and the following) are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 2

```
0000d404    ~~mul x6, x2, x4~~
0000d408    ~~or  x7, x5, x2~~
0000d40c    ~~and x4, x1, x3~~
0000d410    ~~sub x2, x1, x3~~
```

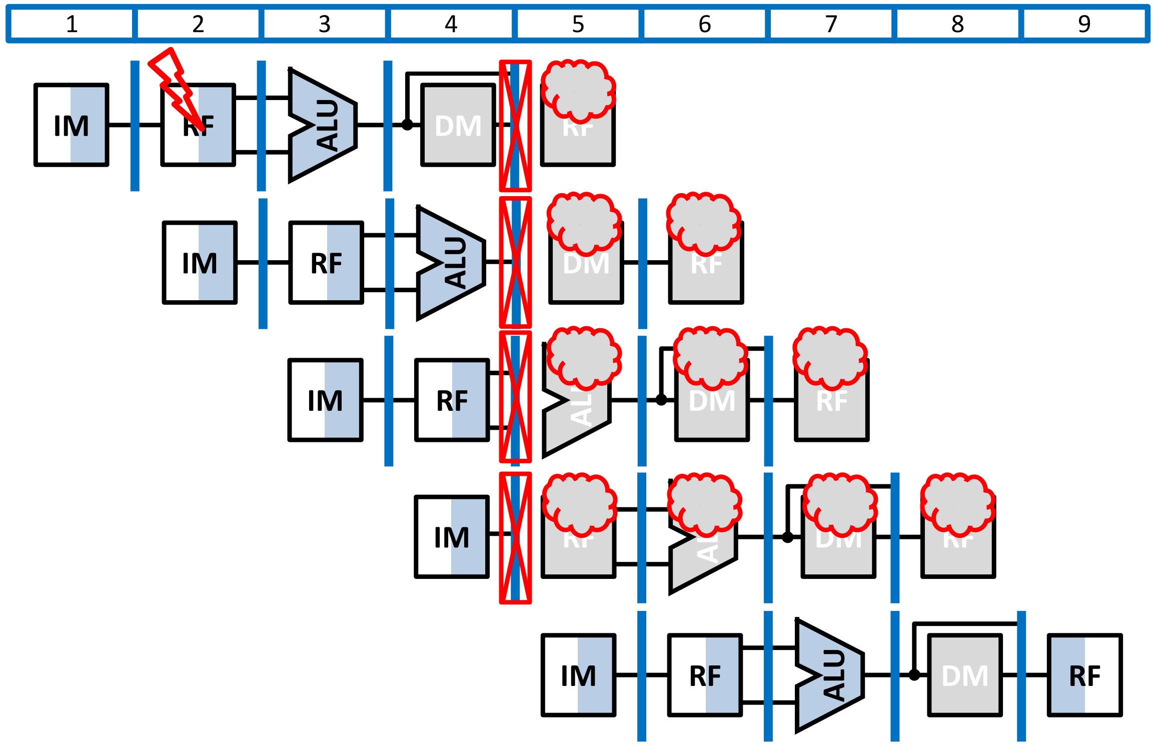


Pipelined processor

Simple exception (ii)

- Regardless of when the exception happens, it will always be handled when the instruction producing it arrives at MEM:
  - Cycle 2: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`mul`** (in MEM). **`mul`** (and the following) are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 2
  - Cycle 6: the instruction whose address is in **`mtvec`** is fetched.

```
0000d404      ~~mul  x6, x2, x4~~
0000d408      ~~or   x7, x5, x2~~
0000d40c      ~~and  x4, x1, x3~~
0000d410      ~~sub  x2, x1, x3~~
1c000000 ESR: add  sp, sp, -8
```

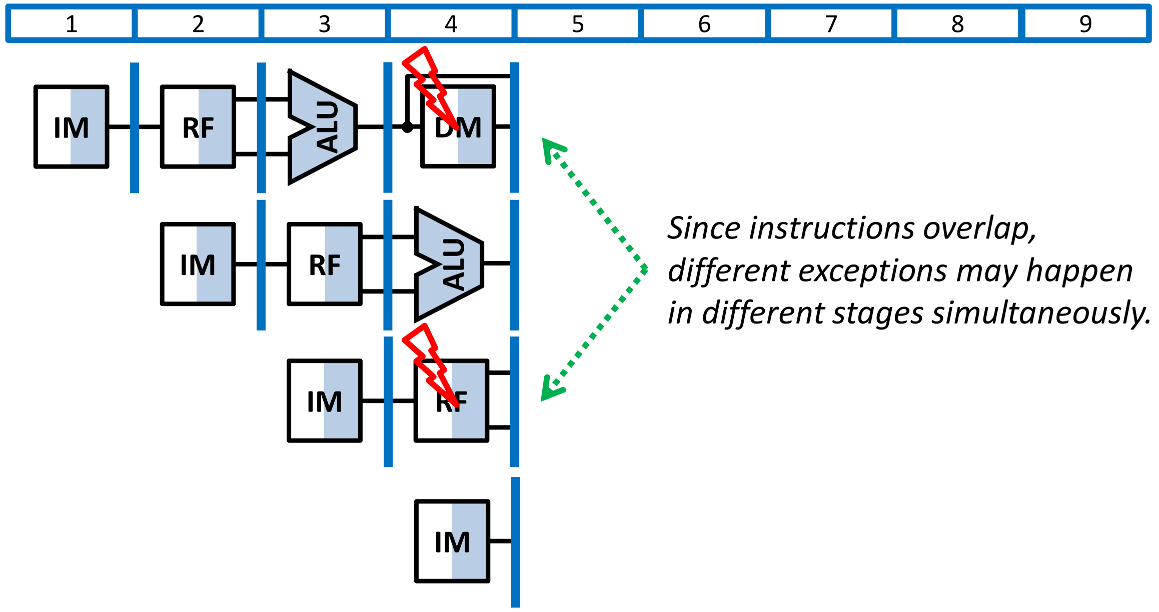


Pipelined processor

Multiple simultaneous exceptions

- If there are simultaneous exceptions, this allows handling the instruction that was fetched earlier, since this is the one that arrives at MEM first:
  - Cycle 4: Instructions **`lw`** (in MEM) and **`mul`** (in ID) generate exceptions.

```
0000d404        lw   x2, 20(x1)
0000d408        or   x7, x5, x2
0000d40c        mul  x6, x2, x4
0000d410        add  x8, x2, x2
```



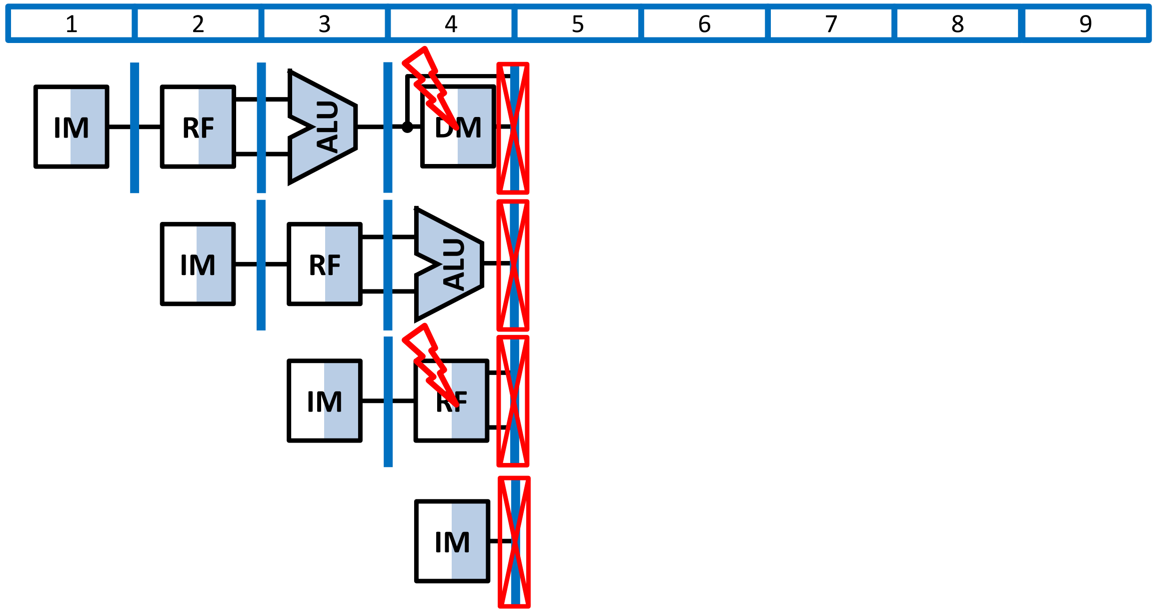
Since instructions overlap, different exceptions may happen in different stages simultaneously.

Pipelined processor

Multiple simultaneous exceptions

- If there are simultaneous exceptions, this allows handling the instruction that was fetched earlier, since this is the one that arrives at MEM first:
  - Cycle 4: Instructions `lw` (in MEM) and `mul` (in ID) generate exceptions. `lw` (and the following) are flushed.
    - `mepc` = 0x0000d404 and `cause` = 4

```
0000d404    ~~lw  x2, 20(x1)~~
0000d408    ~~or  x7, x5, x2~~
0000d40c    ~~mul x6, x2, x4~~
0000d410    ~~add x8, x2, x2~~
```

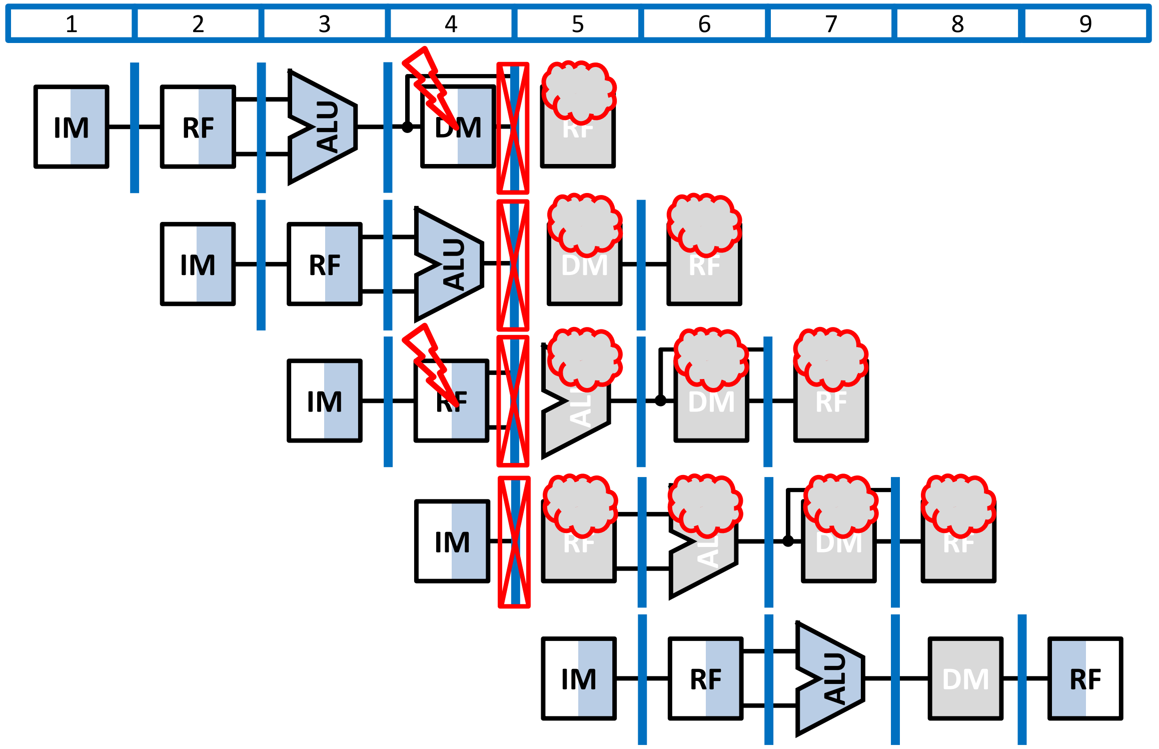


Pipelined processor

Multiple simultaneous exceptions

- If there are simultaneous exceptions, this allows handling the instruction that was fetched earlier, since this is the one that arrives at MEM first:
  - Cycle 4: Instructions `lw` (in MEM) and `mul` (in ID) generate exceptions. `lw` (and the following) are flushed.
    - `mepc` = 0x0000d404 and `cause` = 4
  - Cycle 5: the instruction whose address is in `mtvec` is fetched.

```
0000d404       ~~lw  x2, 20(x1)~~
0000d408       ~~or  x7, x5, x2~~
0000d40c       ~~mul x6, x2, x4~~
0000d410       ~~add x8, x2, x2~~
1c000000  ESR: add sp, sp, -8
```

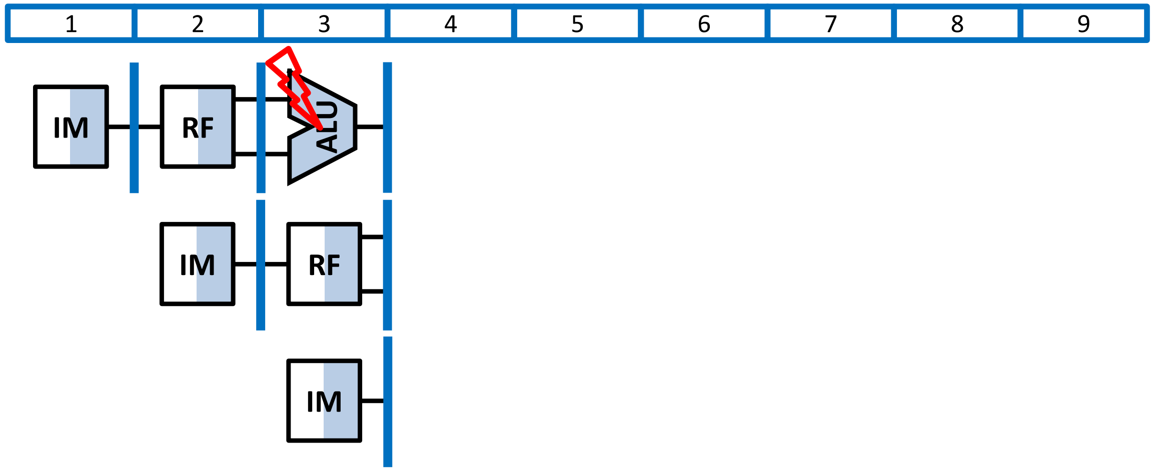


Pipelined processor

In-order multiple non-simultaneous exceptions

- Same if the exceptions are not simultaneous.
  - Cycle 3: **`xor`** (in EX) generates an exception.

```
0000d404        xor x5, x1, x3
0000d408        sra x6, x2, x4
0000d40c        or  x7, x5, x2
```

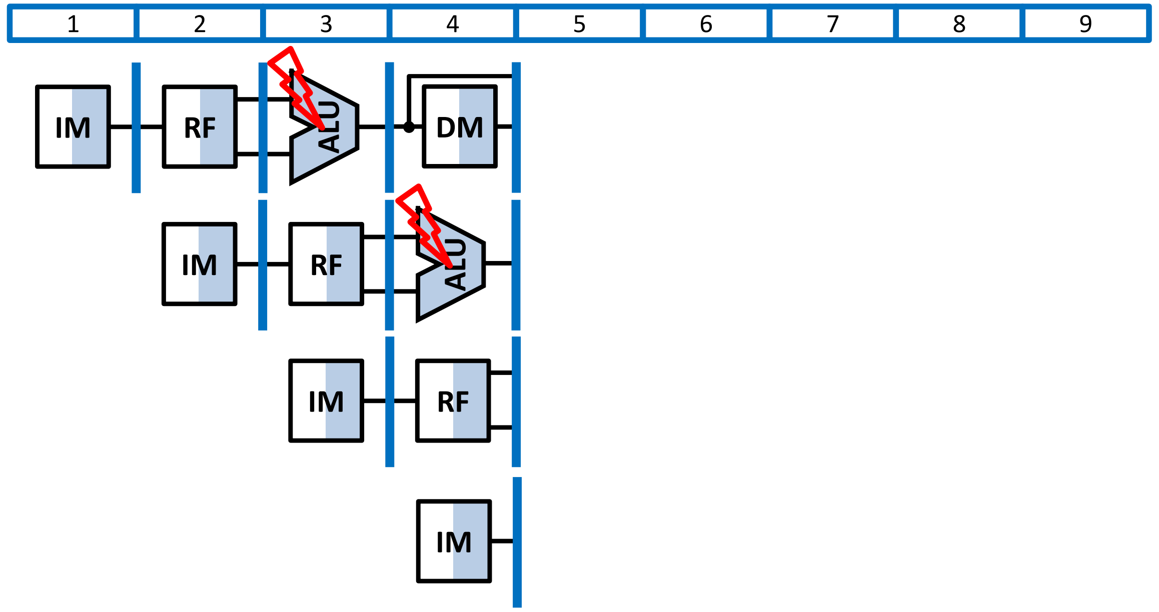


Pipelined processor

In-order multiple non-simultaneous exceptions

- Same if the exceptions are not simultaneous.
  - Cycle 3: **`xor`** (in EX) generates an exception.
  - Cycle 4: **`sra`** (in EX) also generates an exception

```
0000d404        xor x5, x1, x3
0000d408        sra x6, x2, x4
0000d40c        or  x7, x5, x2
0000d410        add x8, x2, x2
```

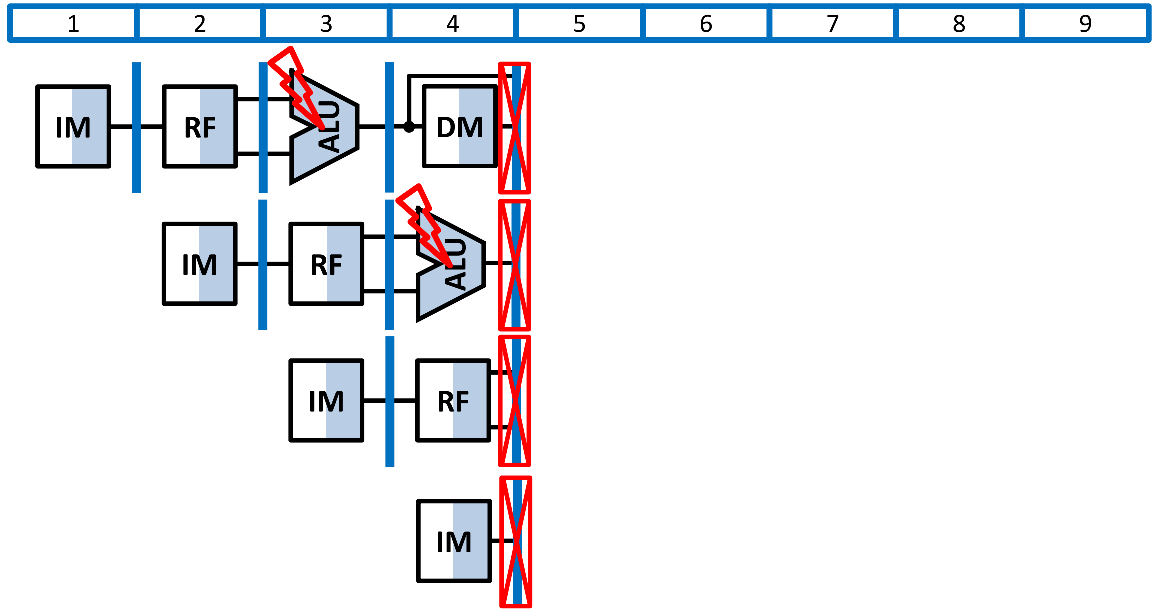


Pipelined processor

In-order multiple non-simultaneous exceptions

- Same if the exceptions are not simultaneous.
  - Cycle 3: **xor** (in EX) generates an exception.
  - Cycle 4: **sra** (in EX) also generates an exception, but **xor** (in MEM) and the following are flushed.
    - **mepc** = 0x0000d404 and **cause** = 2

```
0000d404    ~~xor  x5, x1, x3~~
0000d408    ~~sra  x6, x2, x4~~
0000d40c    ~~or   x7, x5, x2~~
0000d410    ~~add  x8, x2, x2~~
```

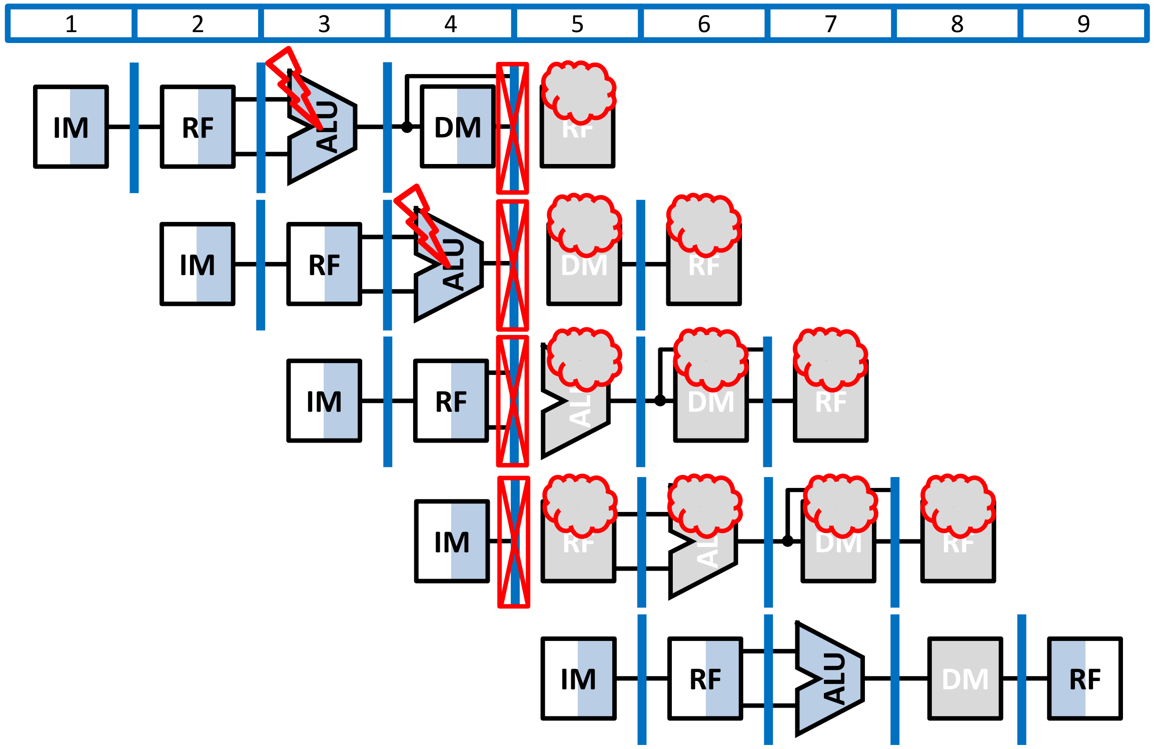


Pipelined processor

In-order multiple non-simultaneous exceptions

- Same if the exceptions are not simultaneous.
  - Cycle 3: **`xor`** (in EX) generates an exception.
  - Cycle 4: **`sra`** (in EX) also generates an exception, but **`xor`** (in MEM) and the following are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 2
  - Cycle 5: the instruction whose address is in **`mtvec`** is fetched.

```
0000d404          ~~xor  x5, x1, x3~~
0000d408          ~~sra  x6, x2, x4~~
0000d40c          ~~or   x7, x5, x2~~
0000d410          ~~add  x8, x2, x2~~
1c000000   ESR:   add  sp, sp, -8
```

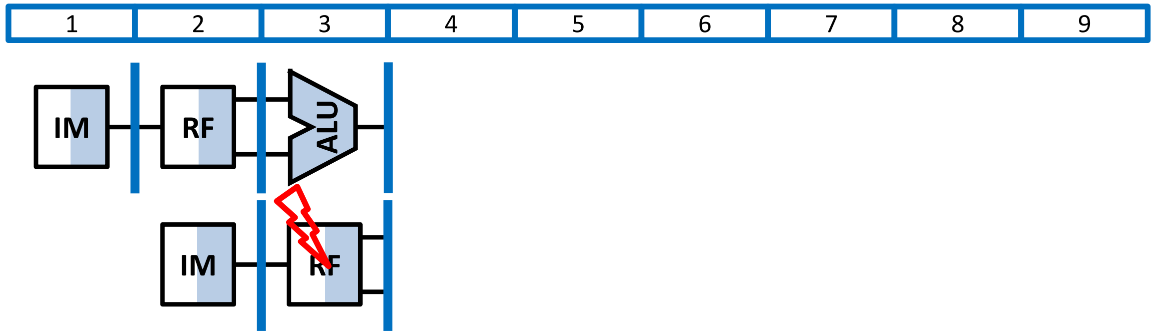


Pipelined processor

Out-of-order multiple non-simultaneous exceptions

- Same for out-of-order exceptions:
  - Cycle 3: **`mul`** (in ID) generates an exception.

```
0000d404        lw   x2, 20(x1)
0000d408        mul  x6, x2, x4
```

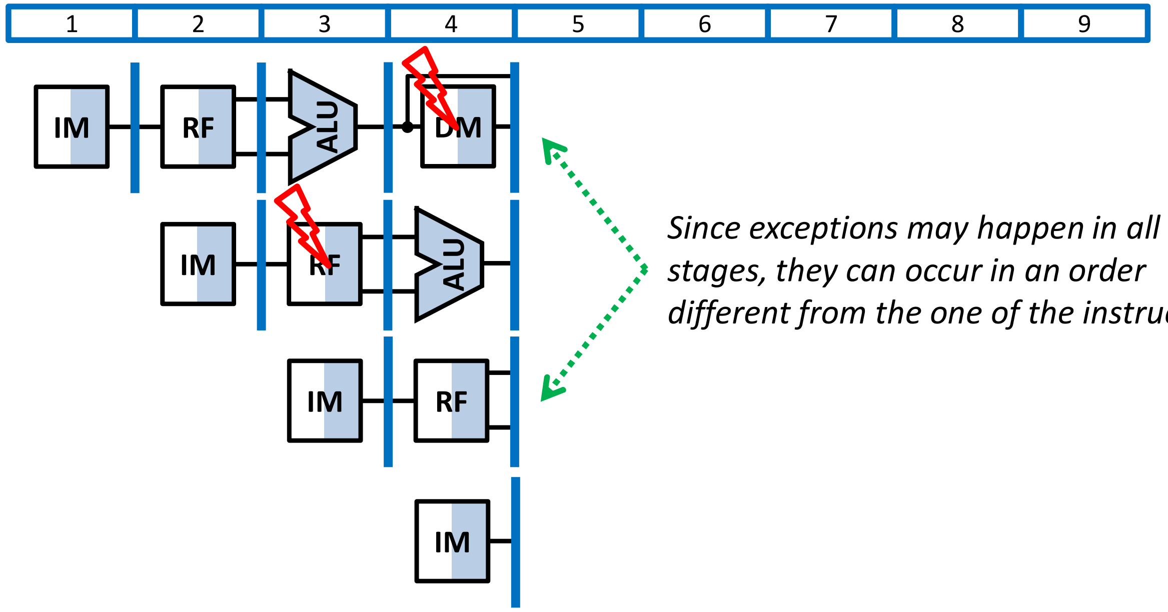


Pipelined processor

Out-of-order multiple non-simultaneous exceptions

- Same for out-of-order exceptions:
  - Cycle 3: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`lw`** (in MEM) generates an exception after **`mul`** , although it is a previous instruction.

```
0000d404        lw   x2, 20(x1)
0000d408        mul  x6, x2, x4
0000d40c        or   x7, x5, x2
0000d410        add  x8, x2, x2
```



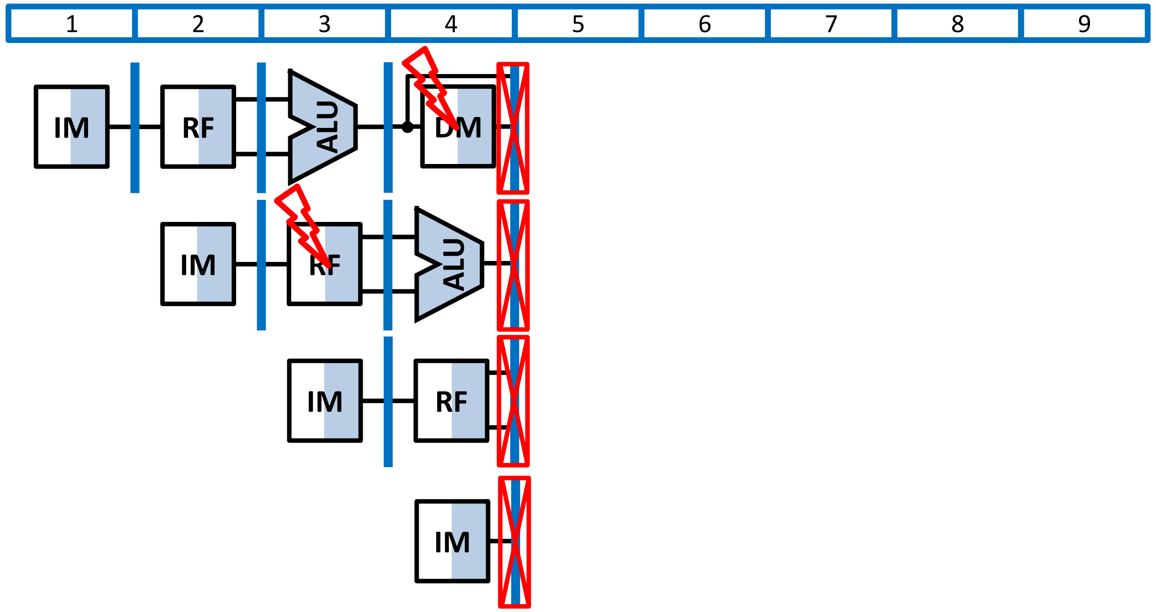
Since exceptions may happen in all the stages, they can occur in an order different from the one of the instructions

Pipelined processor

Out-of-order multiple non-simultaneous exceptions

- Same for out-of-order exceptions:
  - Cycle 3: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`lw`** (in MEM) generates an exception after **`mul`**, although it is a previous instruction. **`lw`** (and the following) are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 4

```
0000d404      ~~lw  x2, 20(x1)~~
0000d408      ~~mul x6, x2, x4~~
0000d40c      ~~or  x7, x5, x2~~
0000d410      ~~add x8, x2, x2~~
```

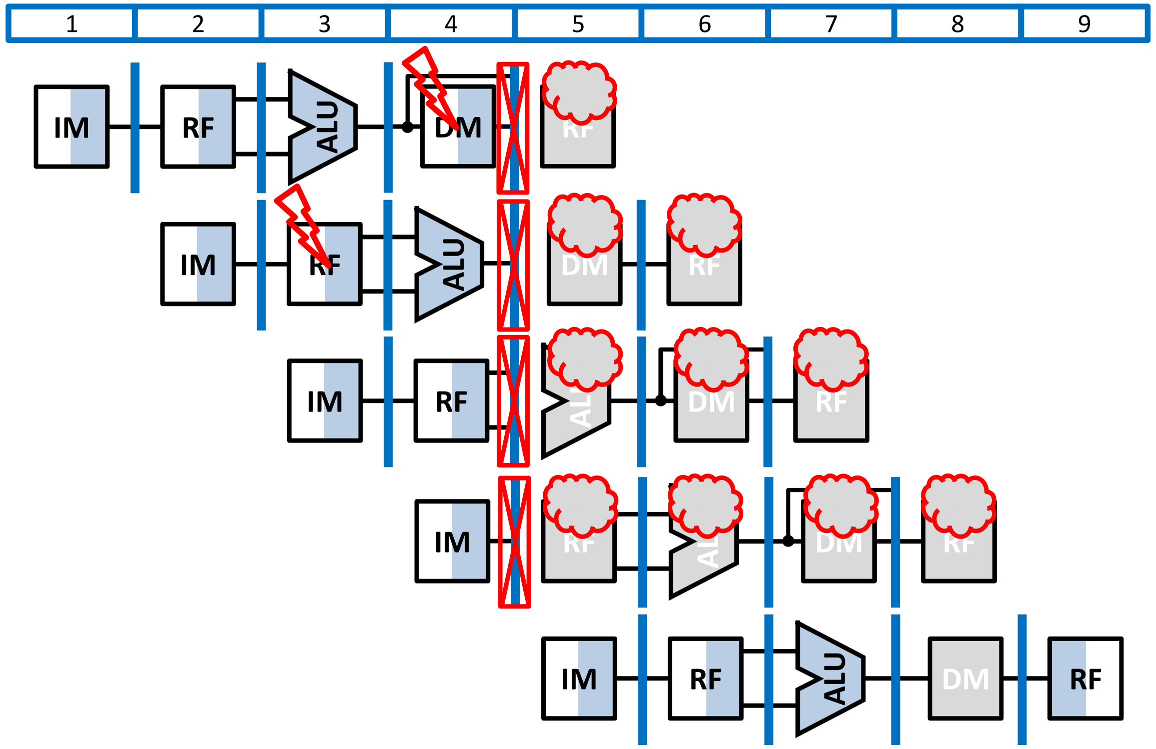


Pipelined processor

Out-of-order multiple non-simultaneous exceptions

- Same for out-of-order exceptions:
  - Cycle 3: **`mul`** (in ID) generates an exception.
  - Cycle 4: **`lw`** (in MEM) generates an exception after **`mul`**, although it is a previous instruction. **`lw`** (and the following) are flushed.
    - **`mepc`** = 0x0000d404 and **`cause`** = 4
  - Cycle 5: the instruction whose address is in **`mtvec`** is fetched.

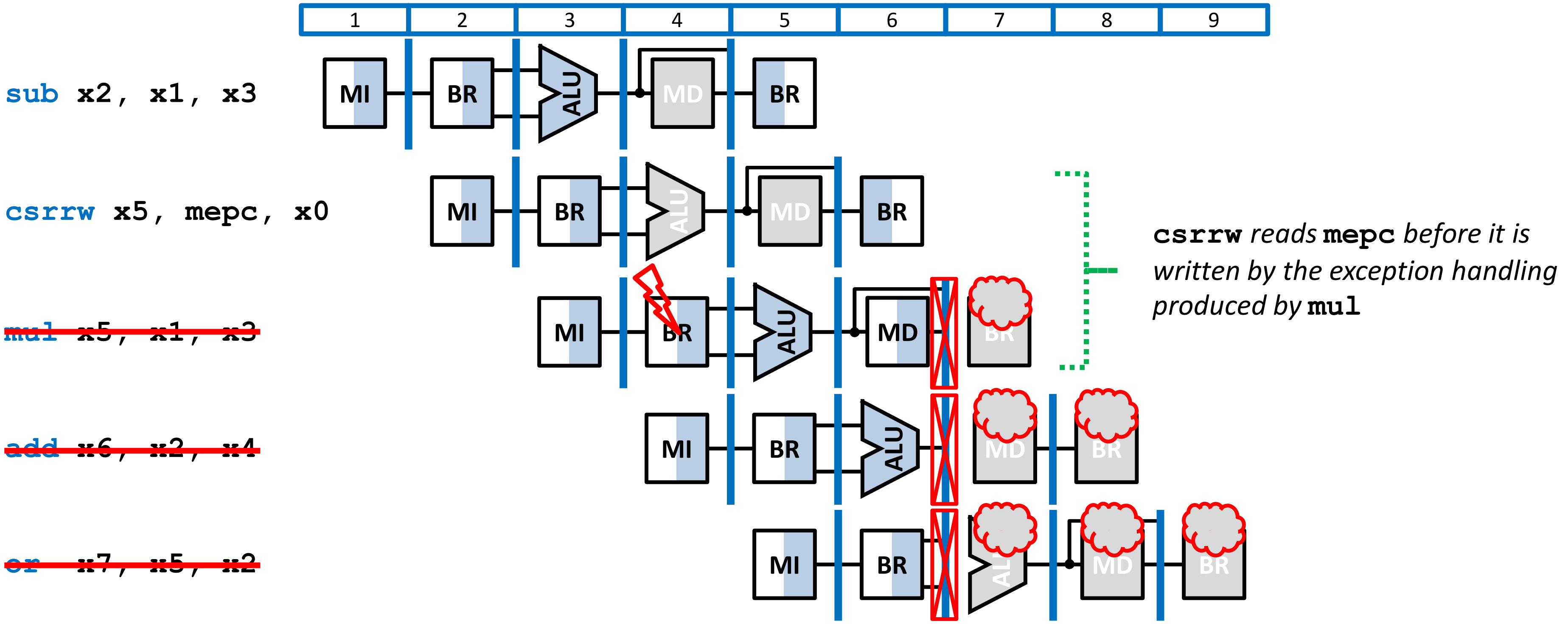
```
0000d404        ~~lw  x2, 20(x1)~~
0000d408        ~~mul x6, x2, x4~~
0000d40c        ~~or  x7, x5, x2~~
0000d410        ~~add x8, x2, x2~~
1c000000  ESR:  add sp, sp, -8
```



Pipelined processor

CSR consistency

- Besides, since the exception handling is performed in the MEM stage of the instruction that produced it:
  - Neither this instruction nor the following ones modify the RF or the memory.
  - All the previous instructions finish as usual.
    - If they had read the **`mepc`** or **`cause`** registers, they would have gotten the previous values.



Pipelined processor

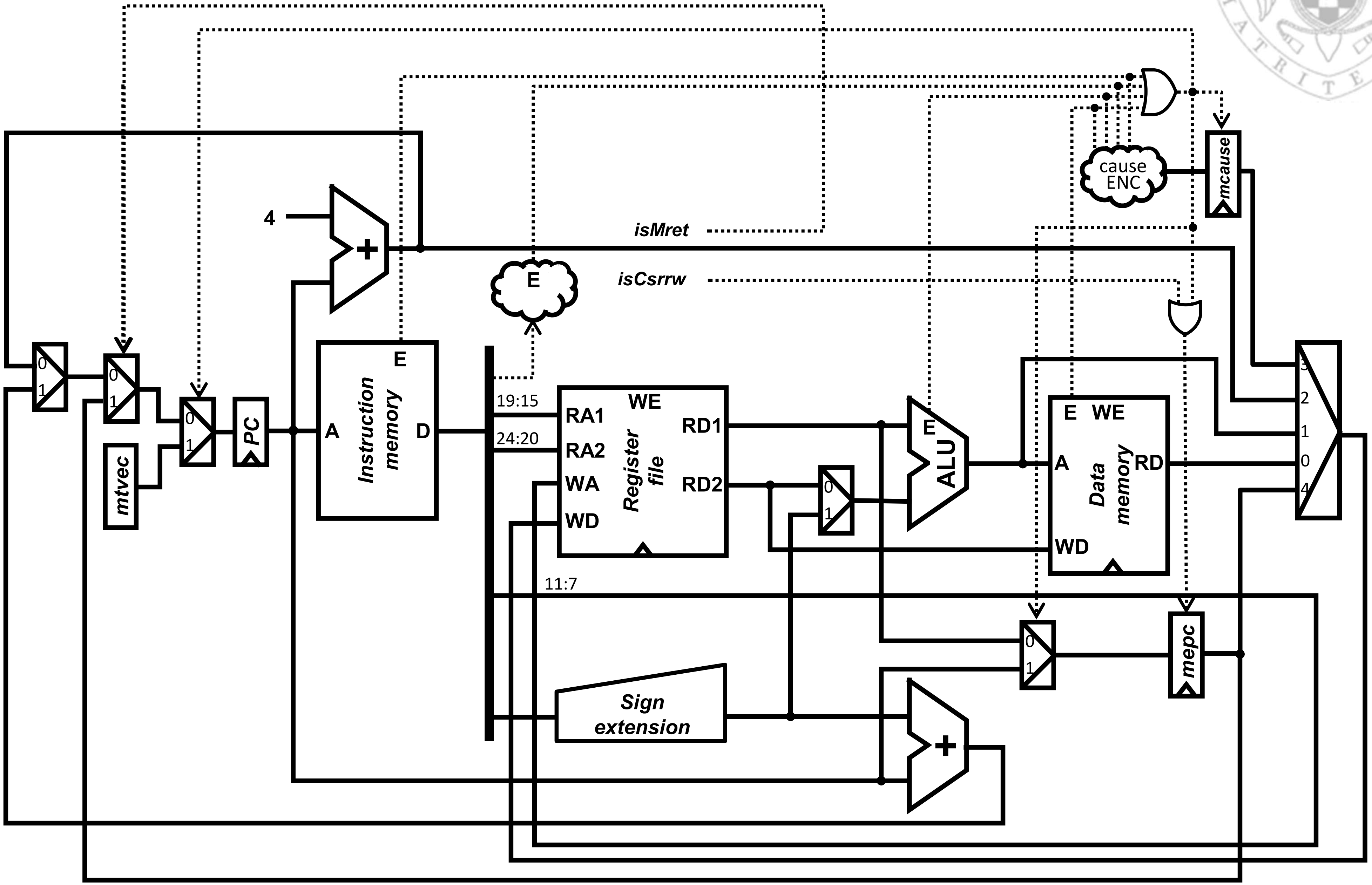
Precise vs. imprecise exceptions

- This handling mechanism is known as precise exceptions, and replicates what happens in a non-pipelined processor, because:
  - The exception generated by the instruction that comes before in the code is always handled first.
  - The instructions previous to the one that produces the exception are completed.
  - The instruction that produces the exception and all the following ones are flushed.
  - The address of the instruction that produced the exception and its cause are stored precisely.

- There are pipelined processors that do not meet all the previous conditions, and therefore they have imprecise exceptions.

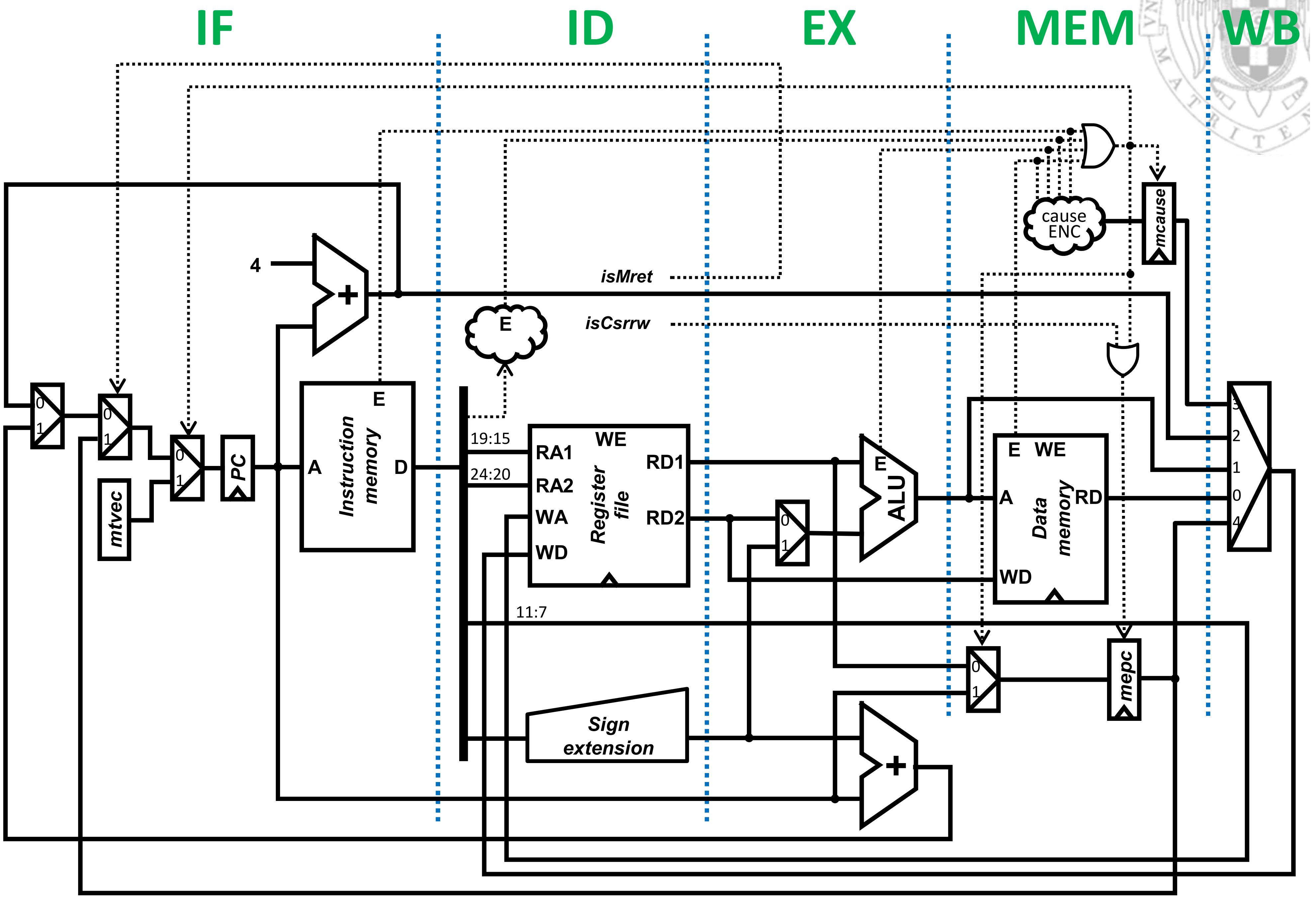
Pipelined processor

Single-cycle data path + exception handling (i)



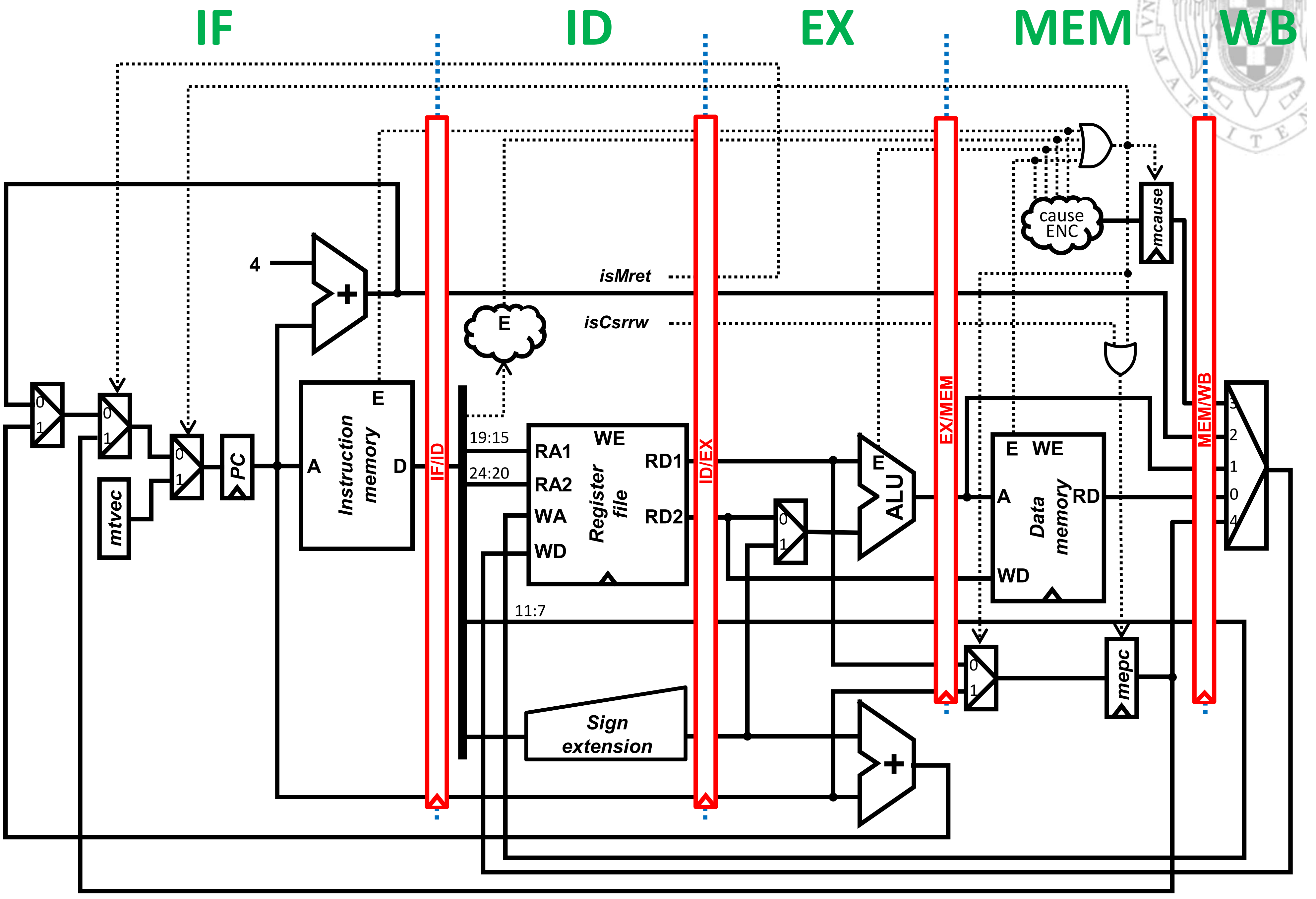
Pipelined processor

Single-cycle data path + exception handling (ii)



Pipelined processor

Pipelined data path + exception handling (i)

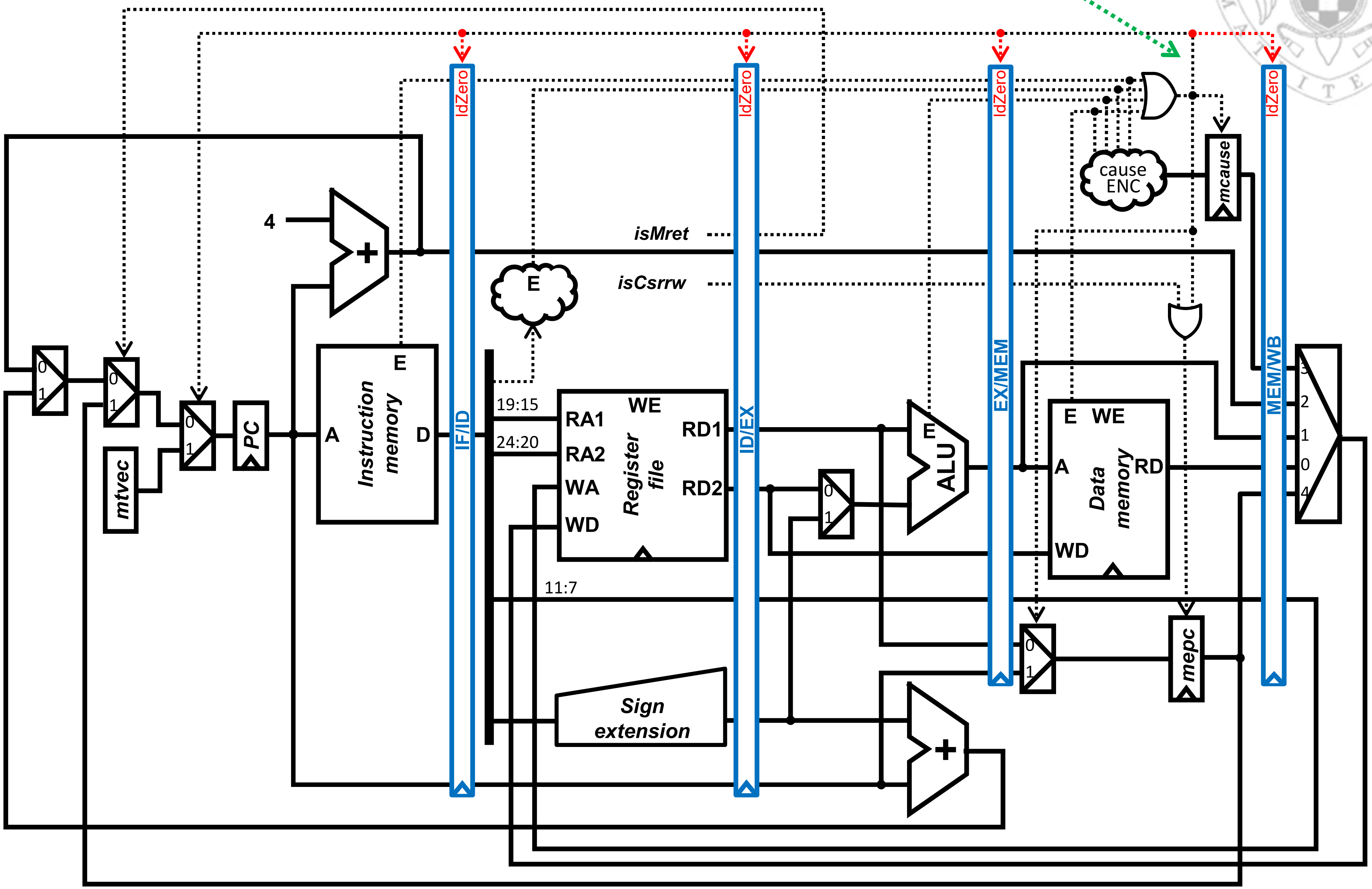


Pipelined processor

Pipelined data path + exception handling (ii)

if exception:

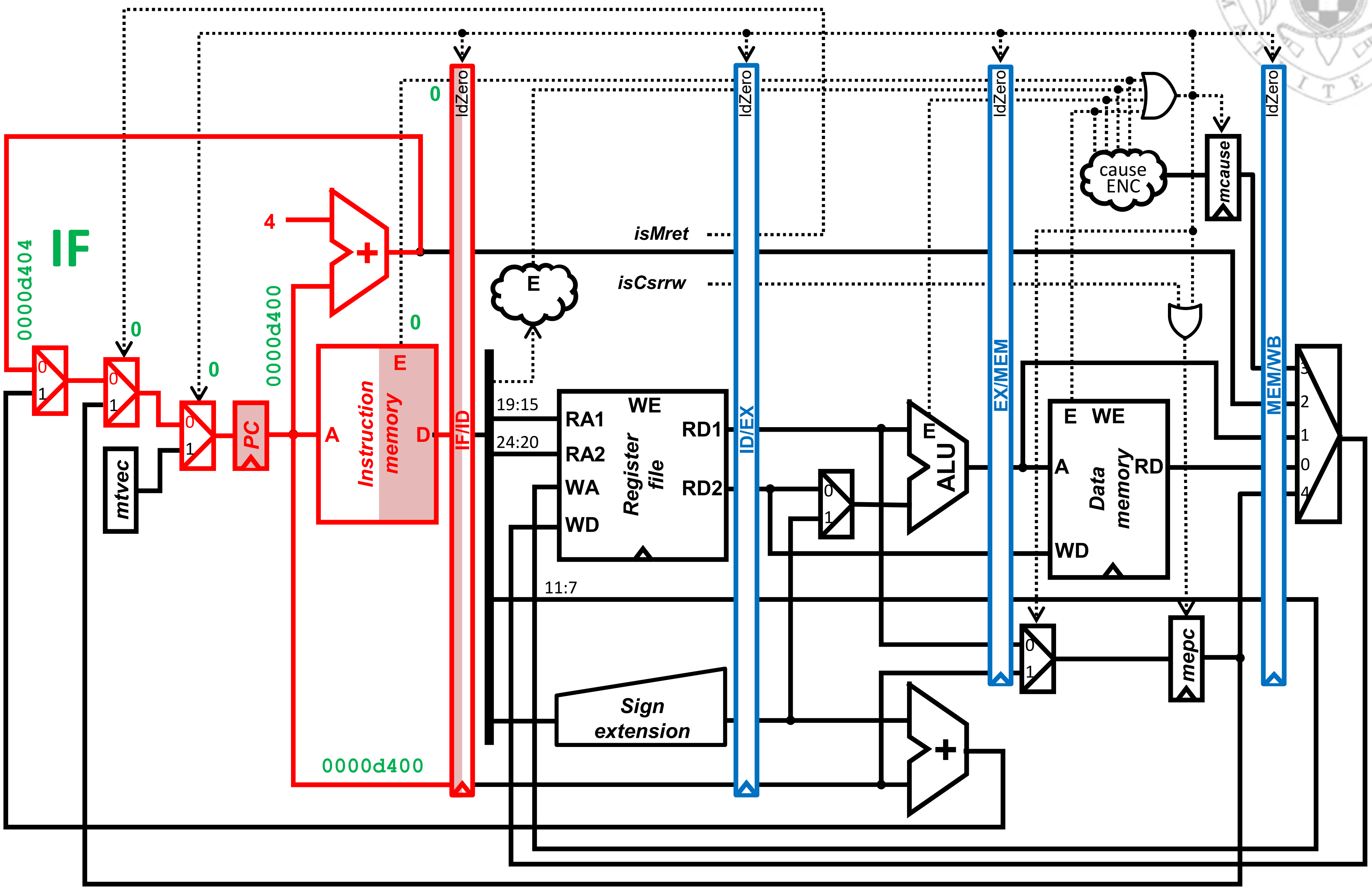
the instruction in MEM and the following are flushed



Pipelined processor

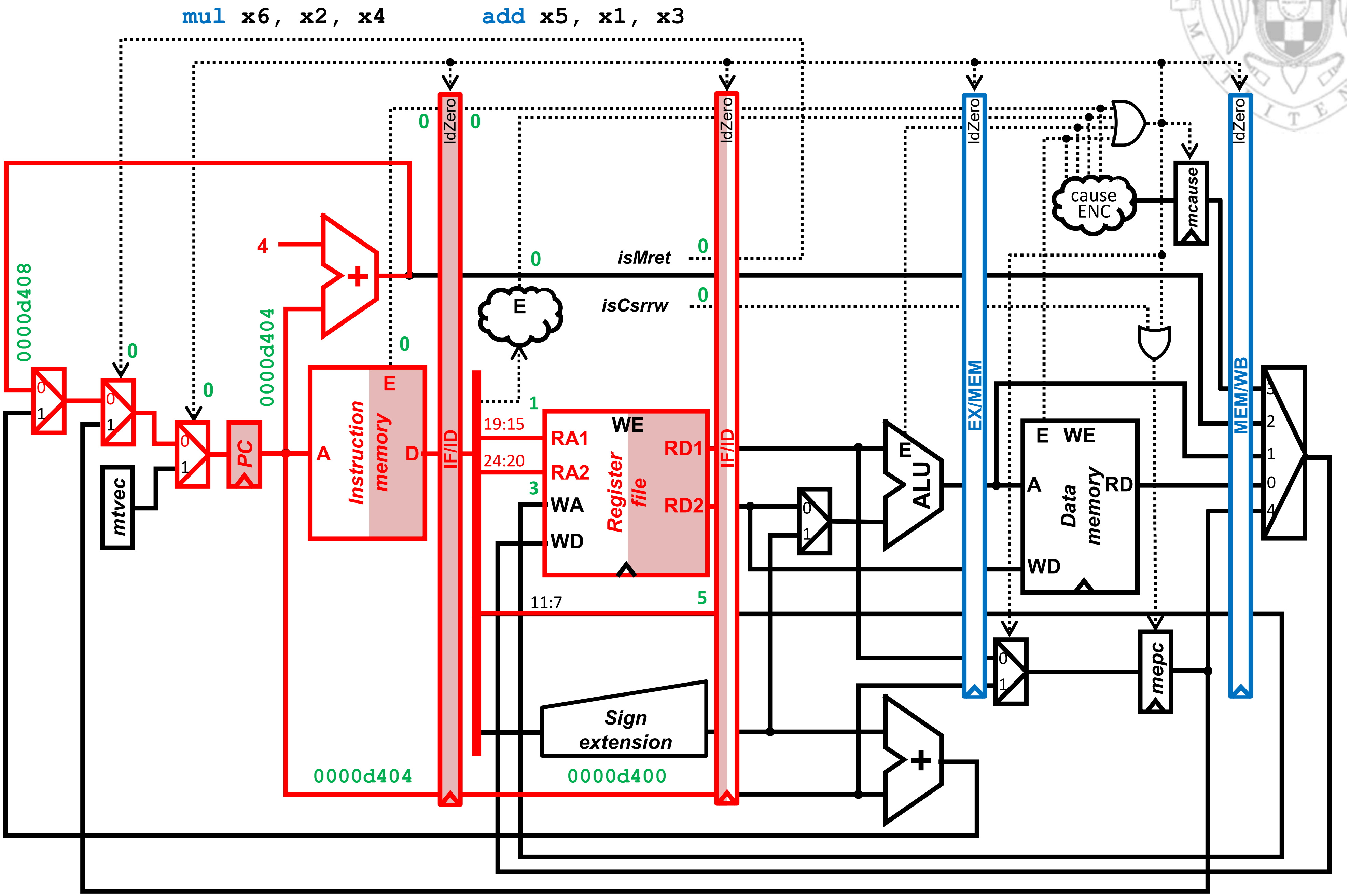
Illegal instruction exception: 1st. cycle

`add x5, x1, x3`



Pipelined processor

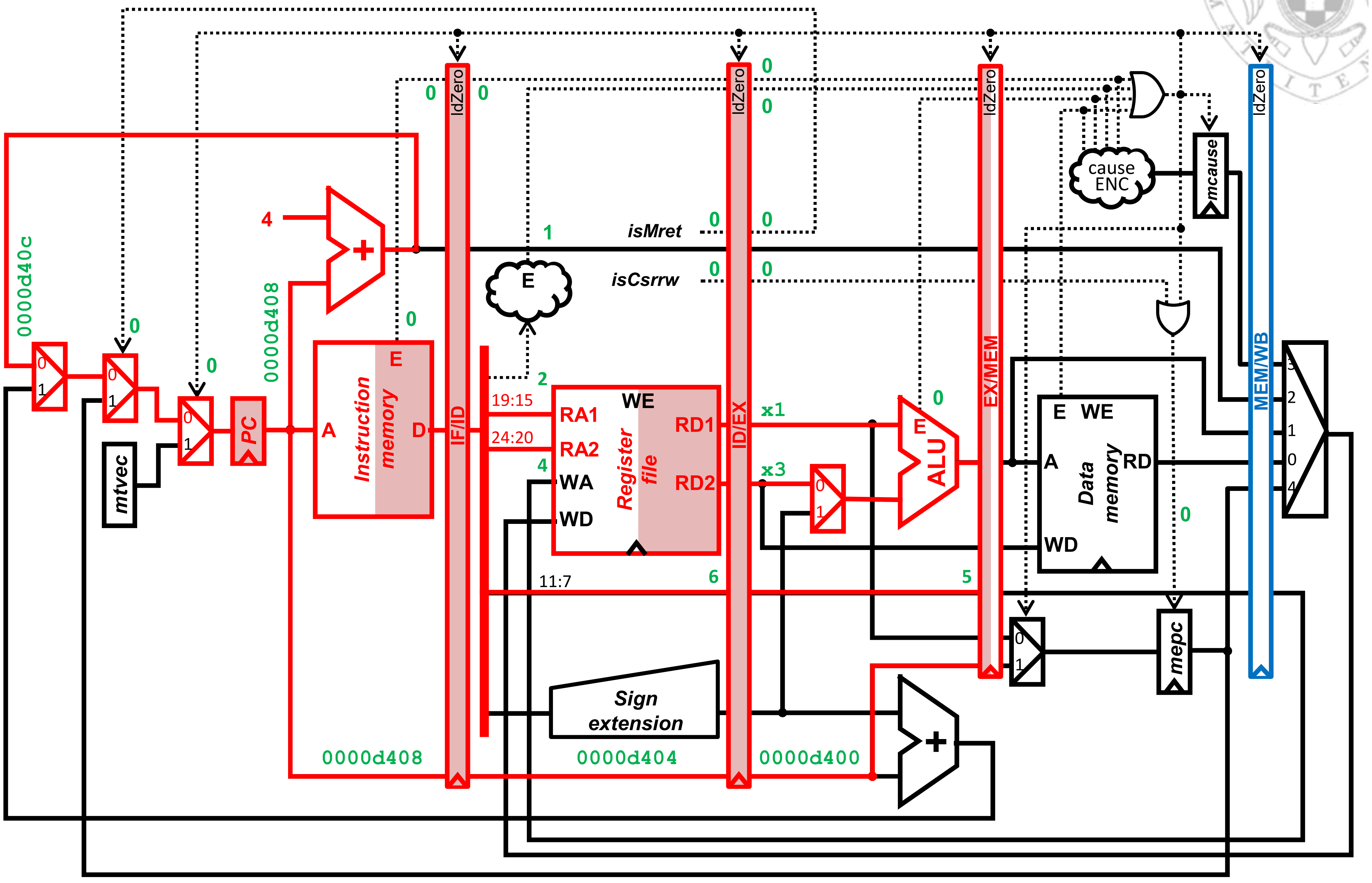
Illegal instruction exception: 2nd. cycle



Pipelined processor

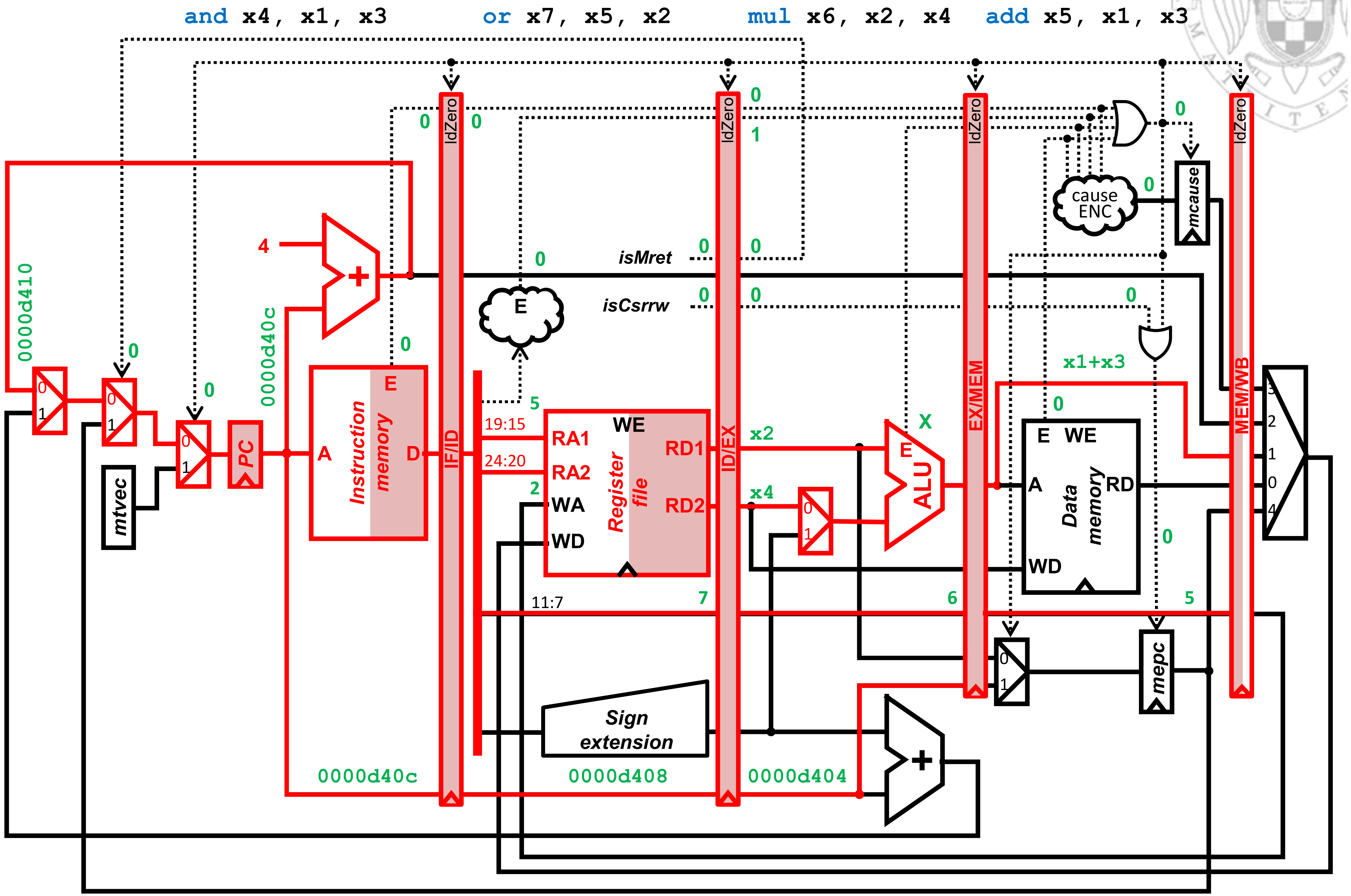
Illegal instruction exception: 3rd. cycle

`or x7, x5, x2` `mul x6, x2, x4` `add x5, x1, x3`



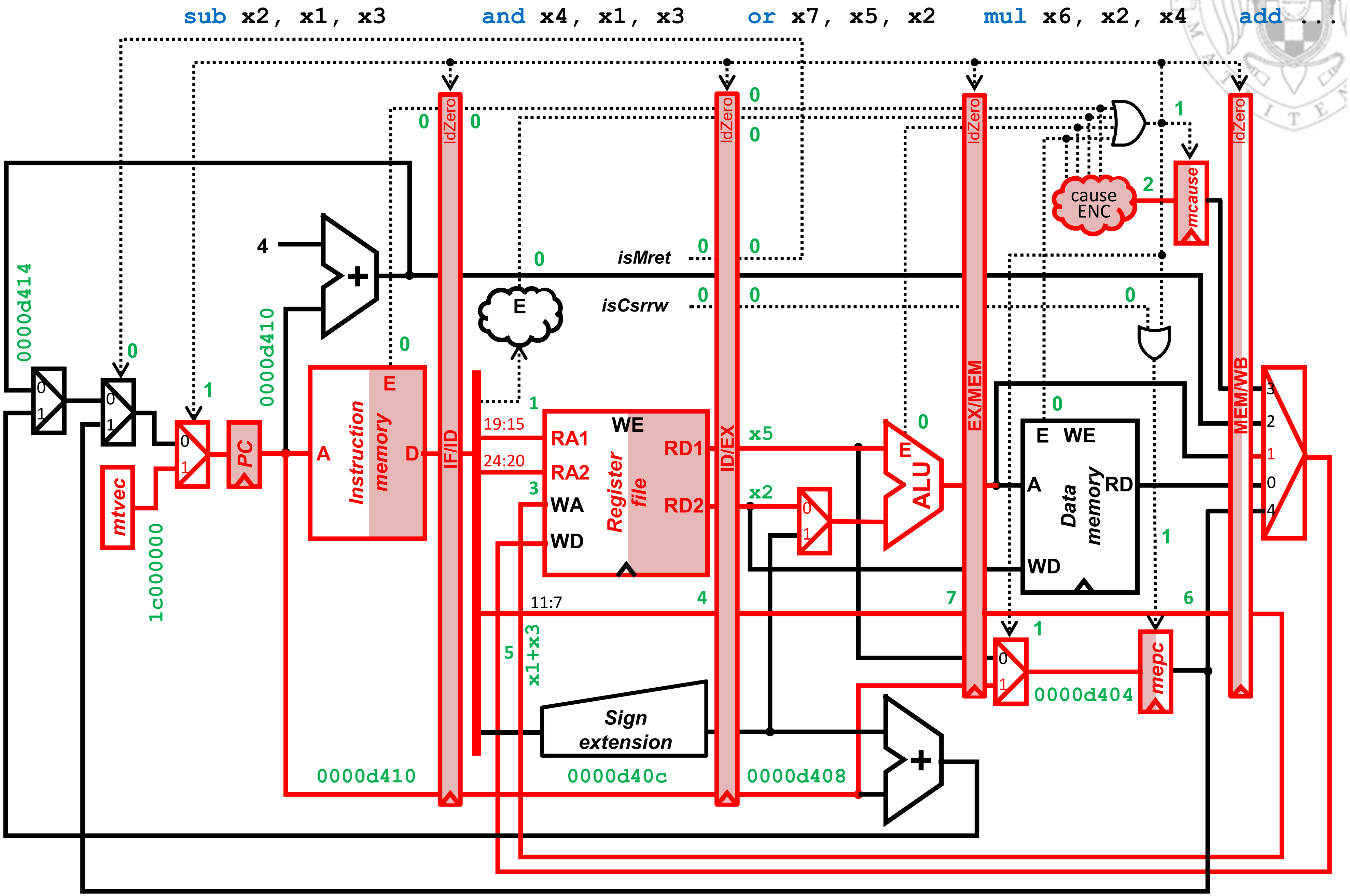
Pipelined processor

Illegal instruction exception: 4th. cycle



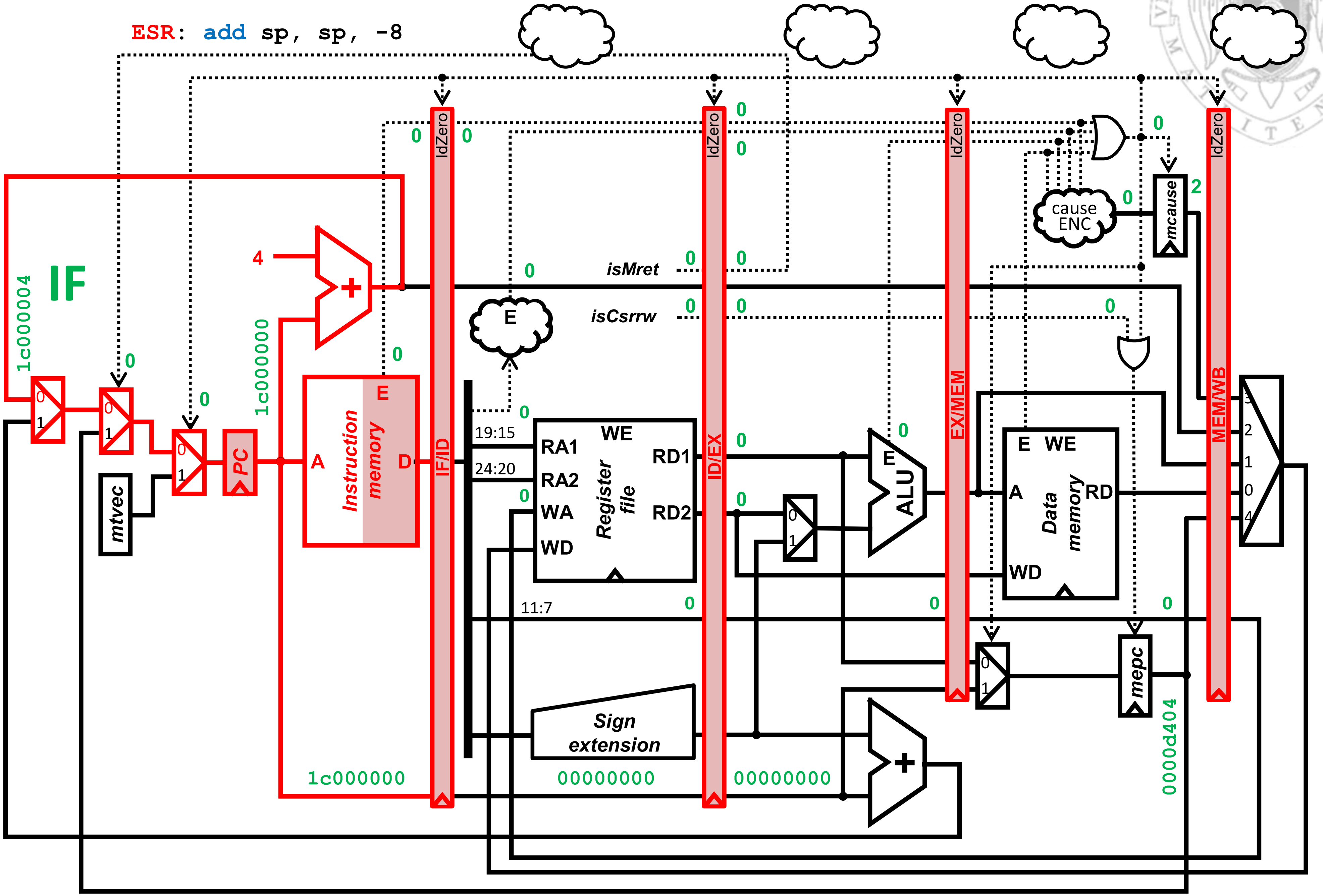
Pipelined processor

Illegal instruction exception: 5th. cycle



Pipelined processor

Illegal instruction exception: 6th. cycle

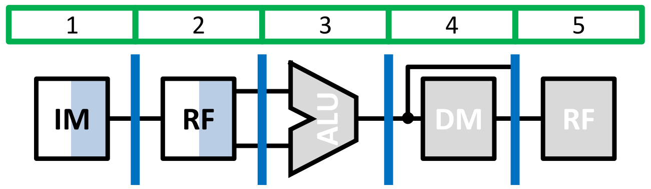


Pipelined processor

New instructions

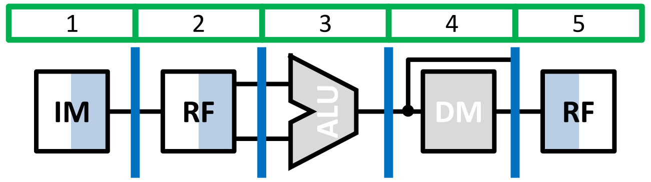
- The `mret` instruction takes <u>5 cycles</u> and as any branch instruction:
  - It does not use the memory in the MEM stage or the RF in the WB stage.
  - It takes the branch in the EX stage (but it does not use the ALU).

`mret`



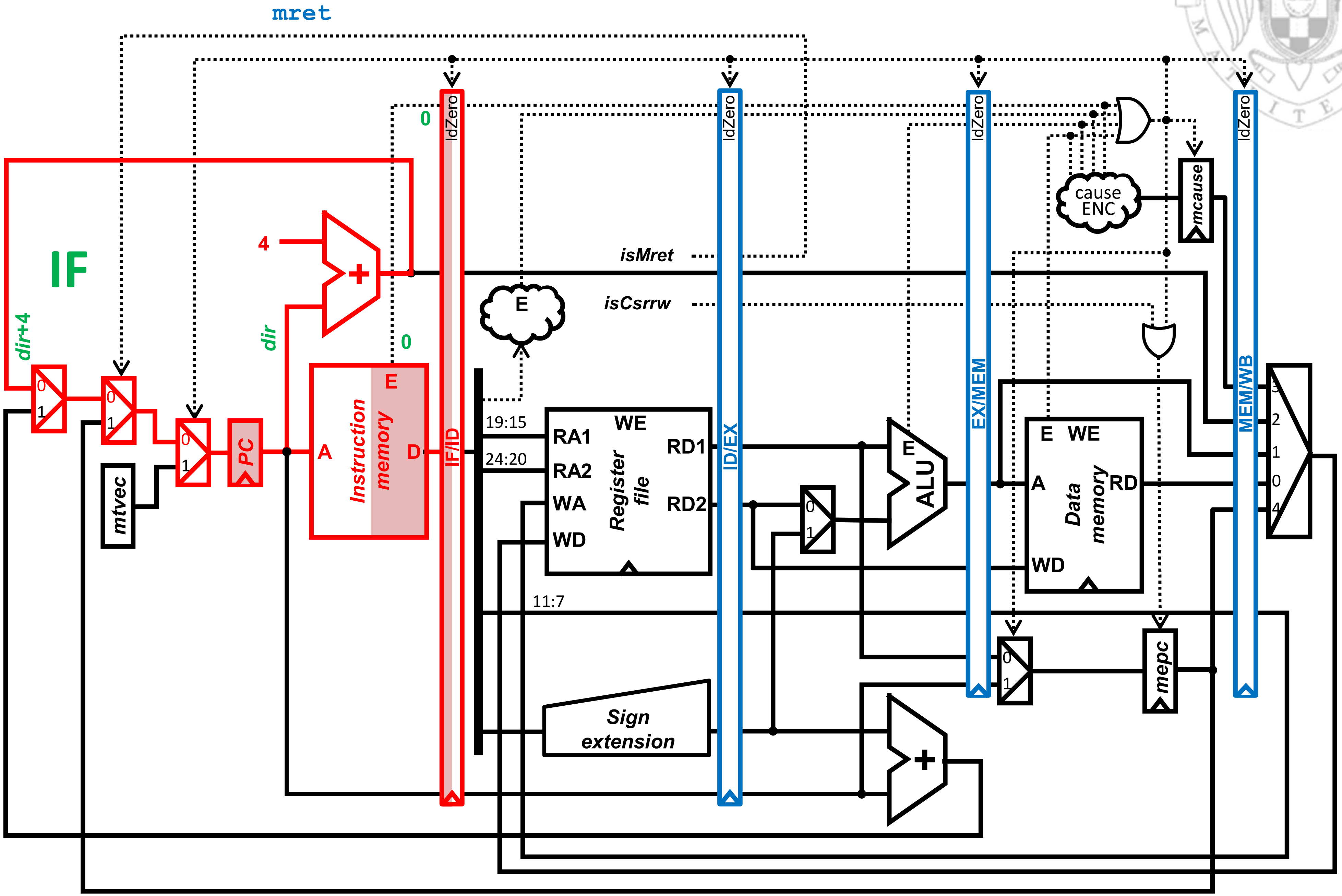
- The `csrrw` instruction takes <u>5 cycles</u>:
  - It does not use the ALU in the EX stage or the memory in the MEM stage.
  - As any other instruction, it reads the RF in ID and writes it in WB.
  - CSR is updated in the MEM stage.

`csrrw x1, mepc, x2`



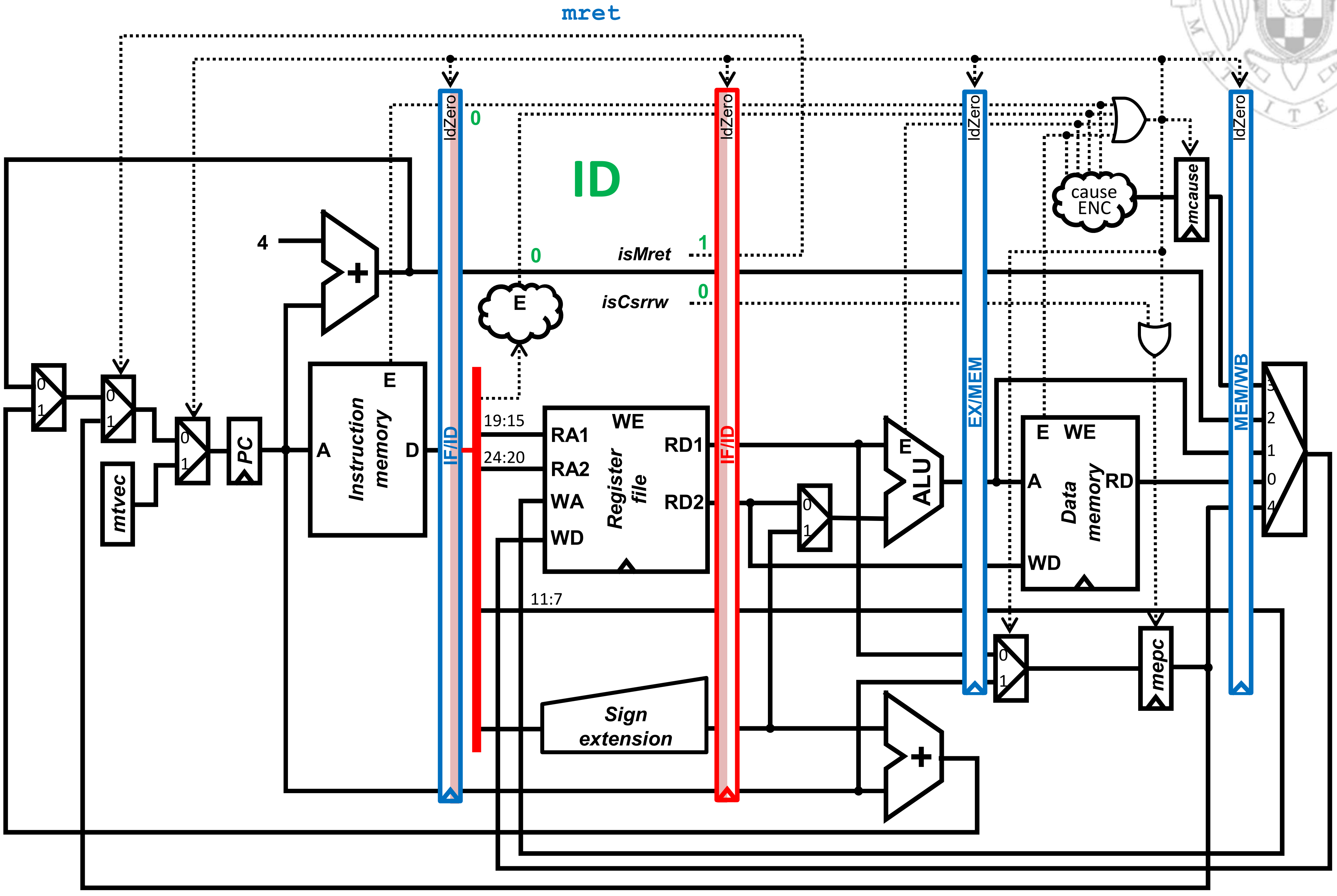
Pipelined processor

mret instruction: IF stage



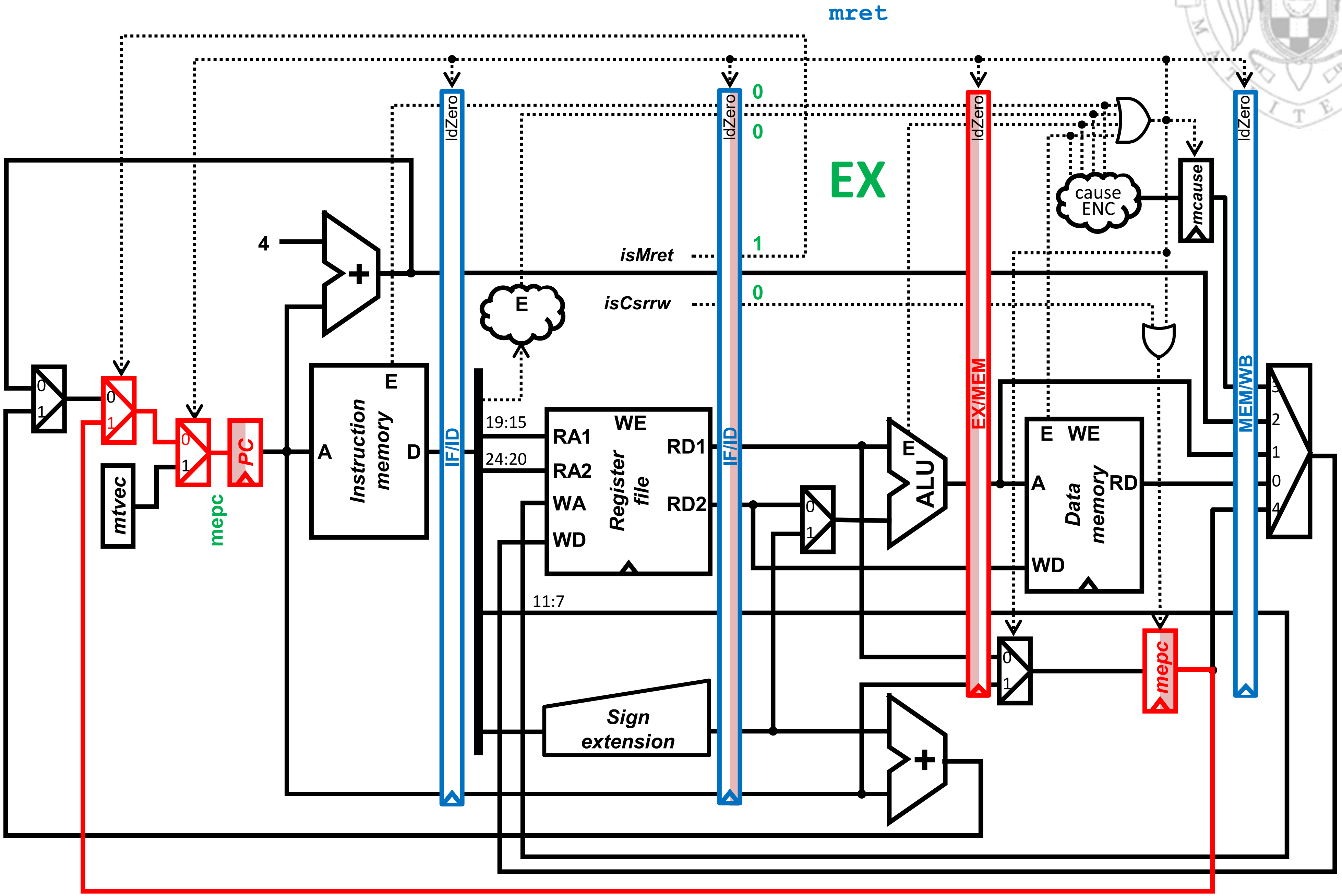
Pipelined processor

mret instruction: ID stage



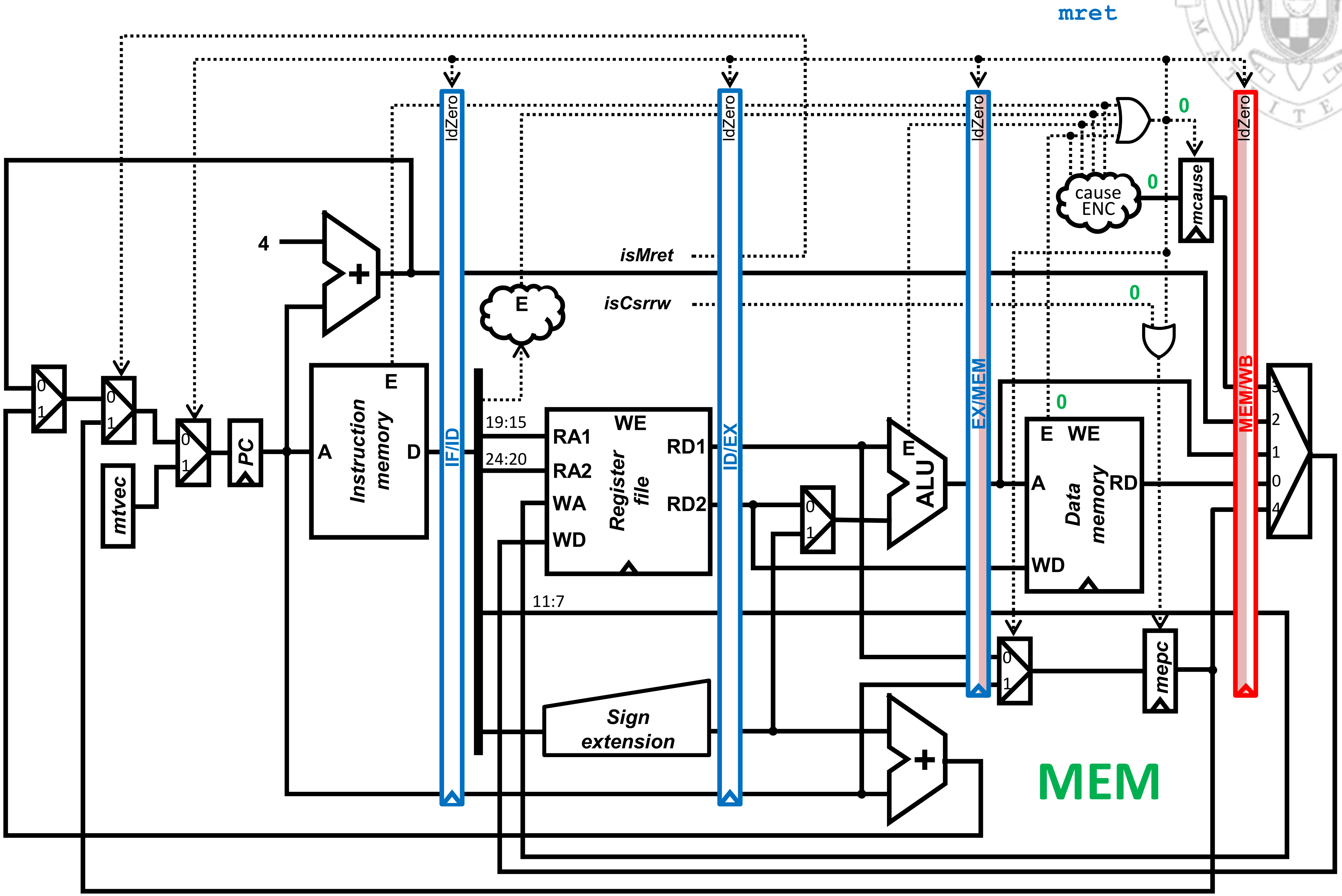
Pipelined processor

mret instruction: EX stage



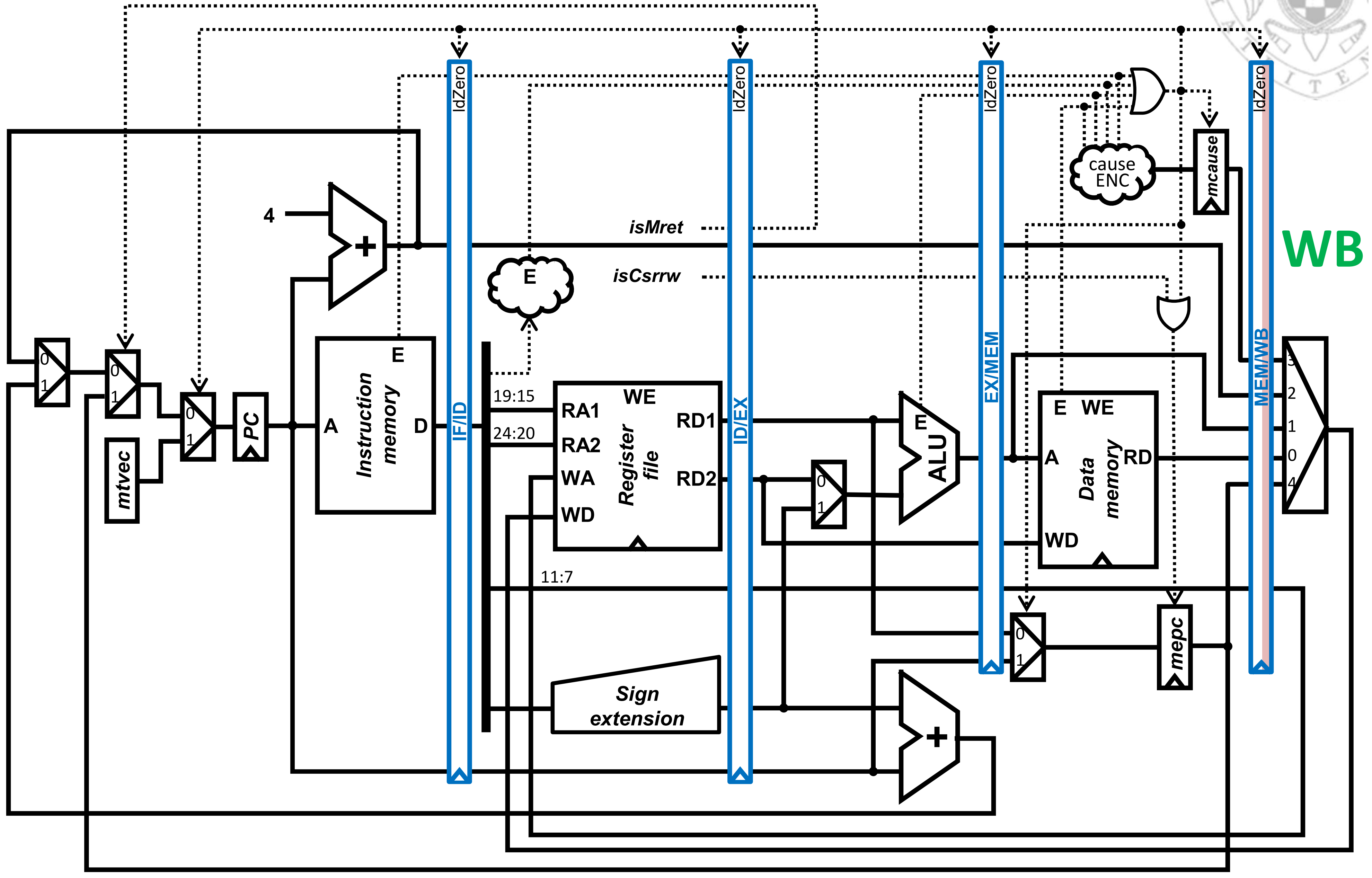
Pipelined processor

mret instruction: MEM stage



Pipelined processor

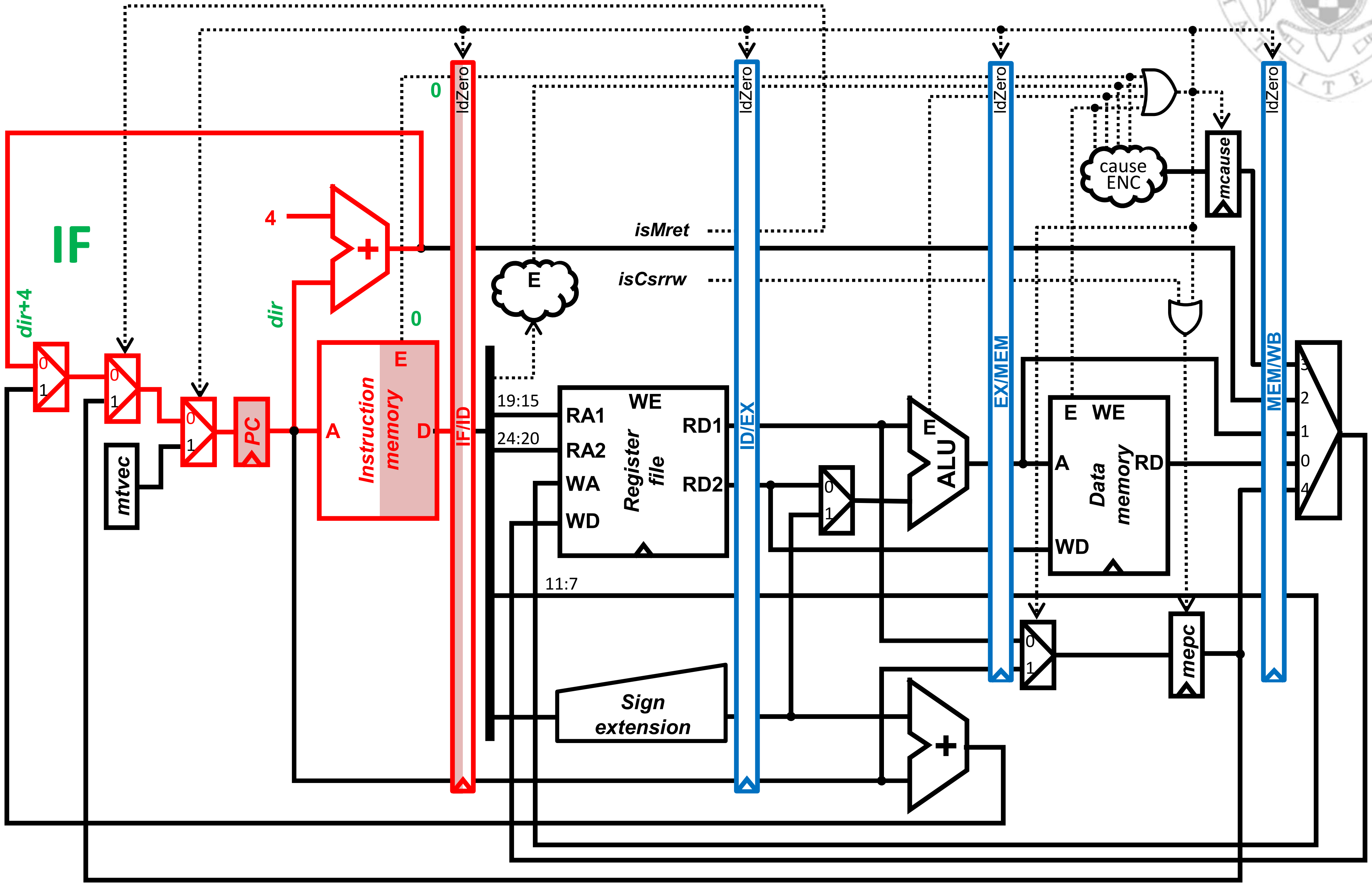
`mret` instruction: WB stage



Pipelined processor

csrrw instruction: IF stage

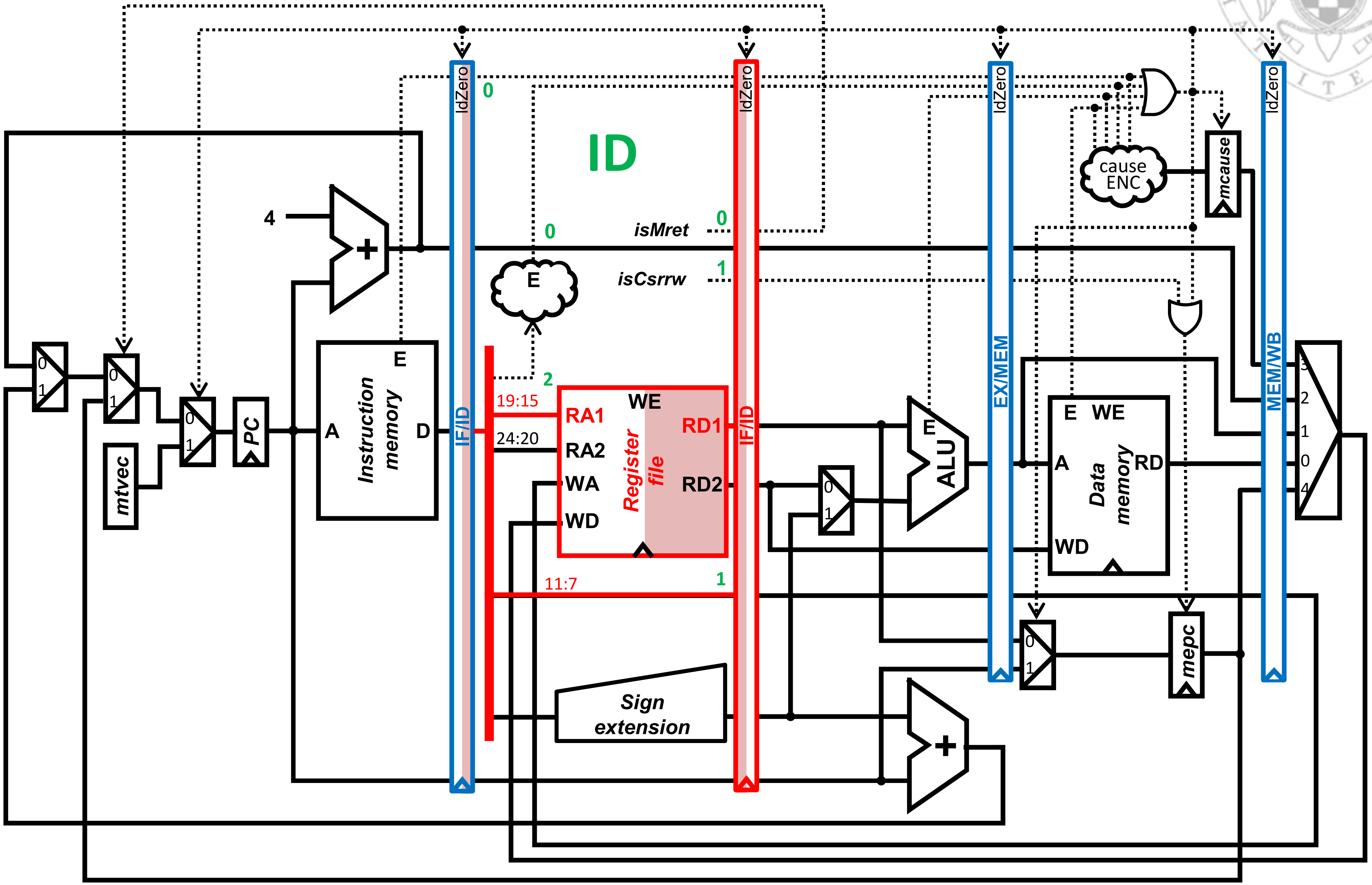
```
csrrw x1, mepc, x2
```



Pipelined processor

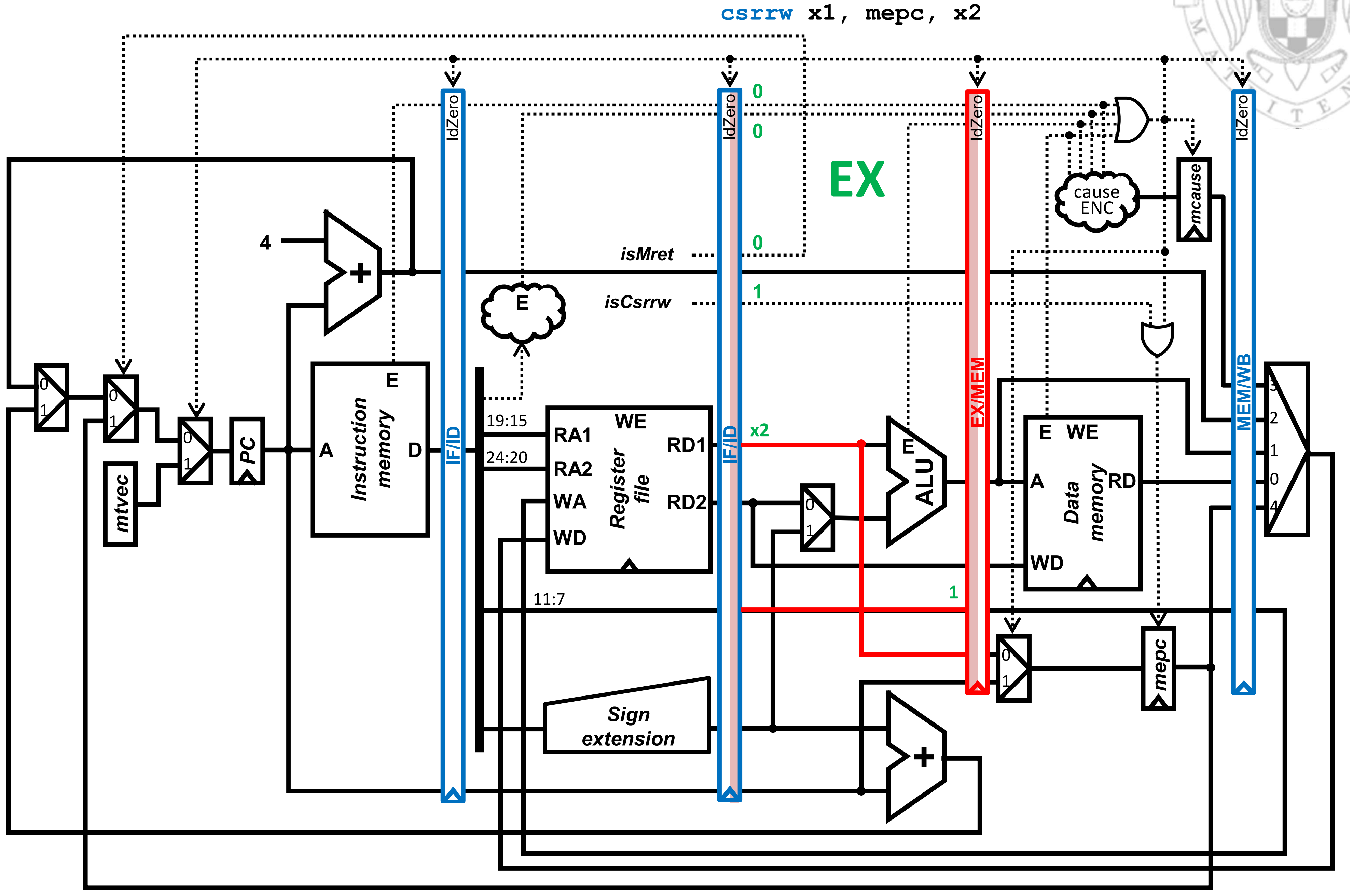
csrrw instruction: ID stage

`csrrw x1, mepc, x2`



Pipelined processor

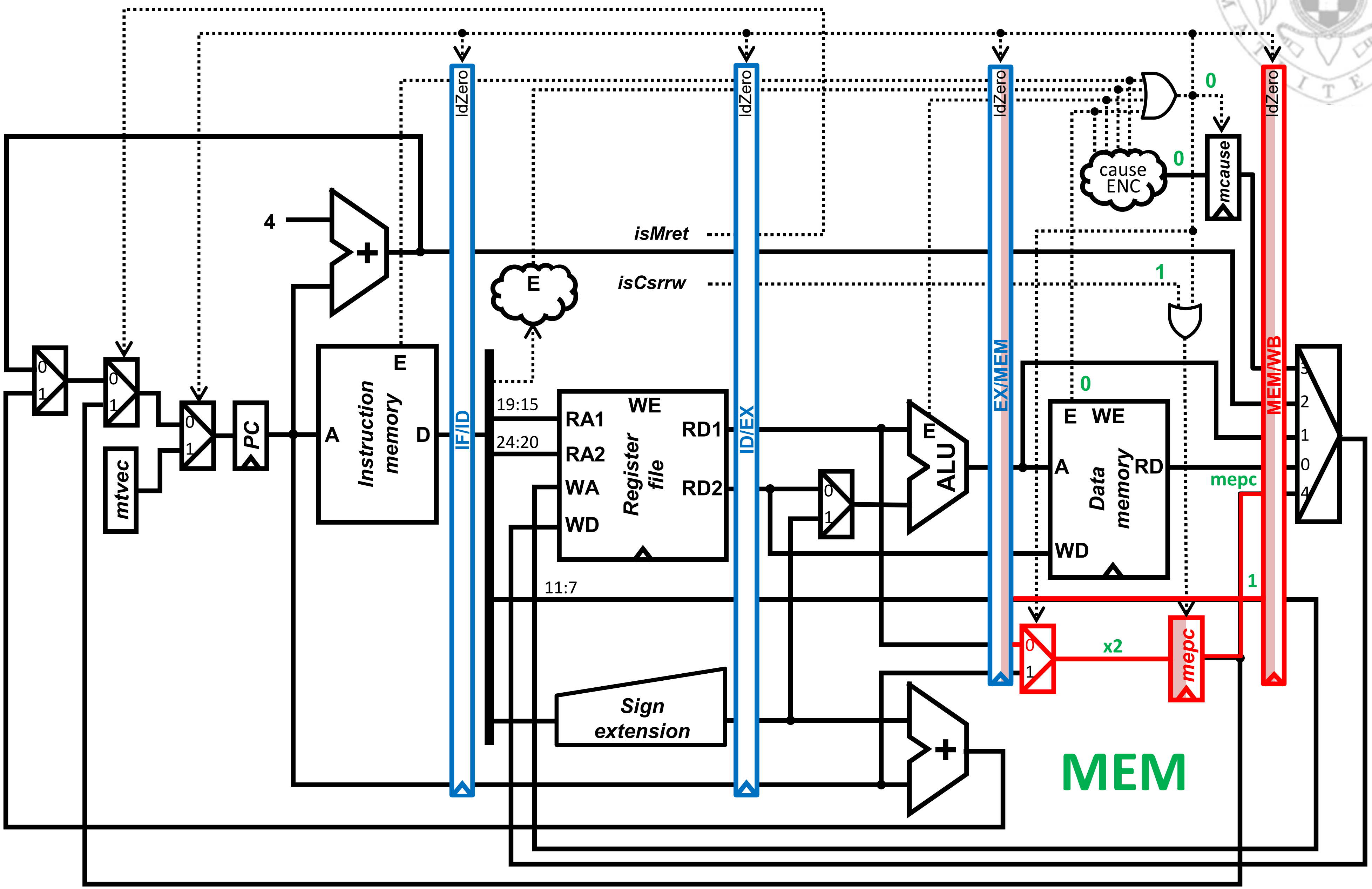
csrrw instruction: EX stage



Pipelined processor

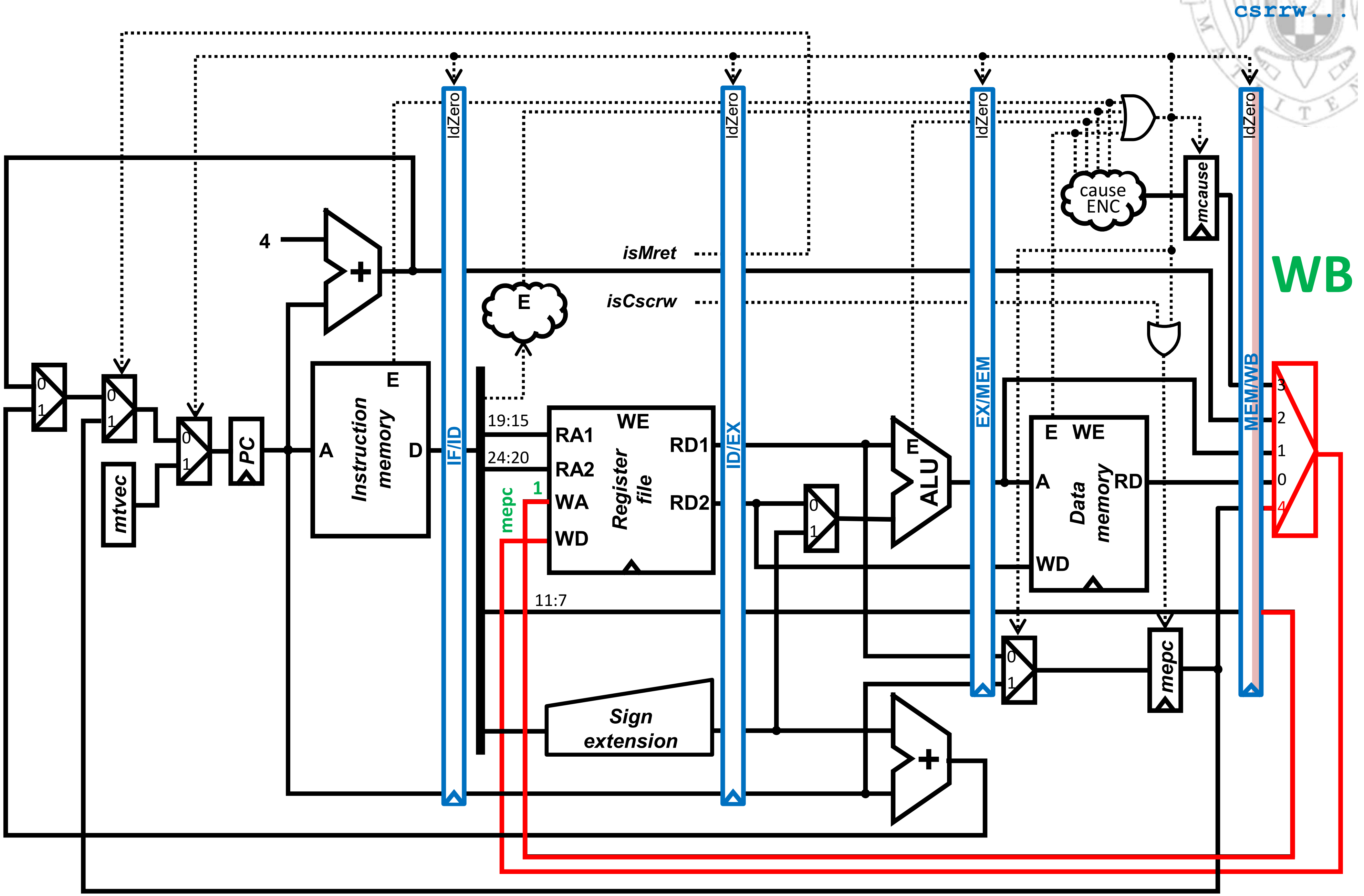
csrrw instruction: MEM stage

`csrrw x1, mepc, x2`



Pipelined processor

csrrw instruction: WB stage



Pipelined processor

Hazards (i)

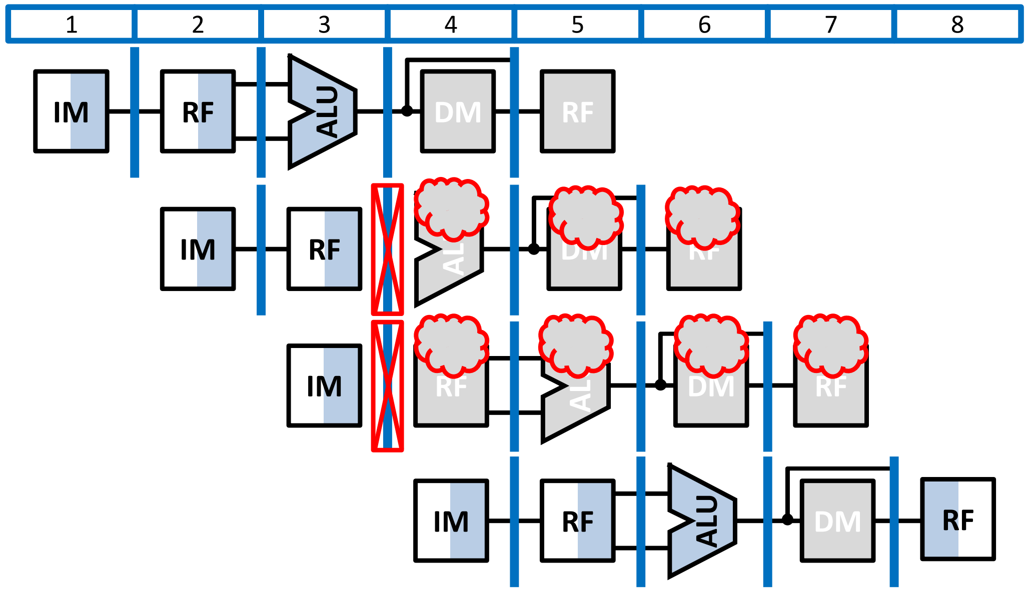
- For the sake of simplicity, the exception handling of the pipelined processor has been designed without hazard management.
- Hazards between already existing instructions are solved by adding the same logic as the one designed in the previous module.
- However, the new instructions, **`mret`** and **`csrrw`**, generate new hazards that must be analyzed.

Pipelined processor

Hazards (i)

- There is a control hazard when executing the **`mret`** instruction:
  - The branch is taken in EX, but the 2 following instructions have been already fetched.
- It is solved as with any branch instruction:
  - With not-taken branch prediction and 2 penalty cycles.
  - The hazard unit must be extended to delete the IF/ID and ID/EX pipeline registers, if a **`mret`** instruction is detected.

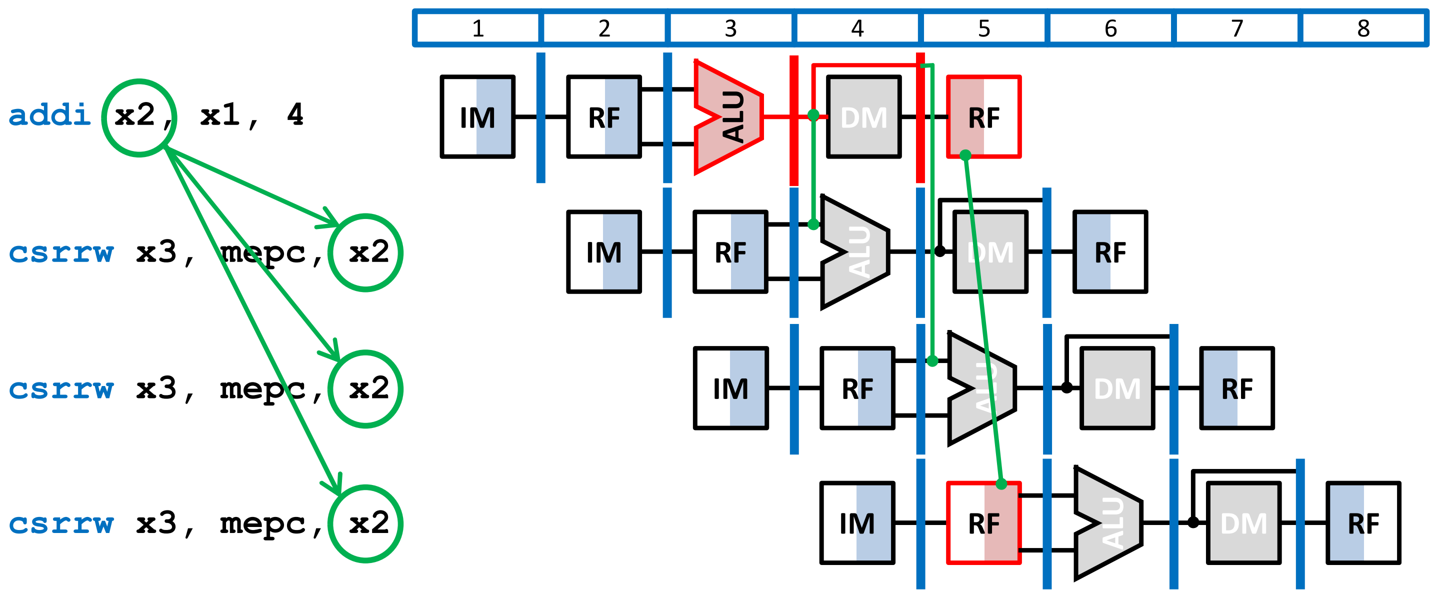
```
mret
~~sub x6, x2, x4~~
~~or  x7, x5, x2~~
add x7, x3, x4
```



Pipelined processor

Hazards (ii)

- There is a data hazard when executing a **`csrrw`** instruction that reads the same register that a previous instruction writes.

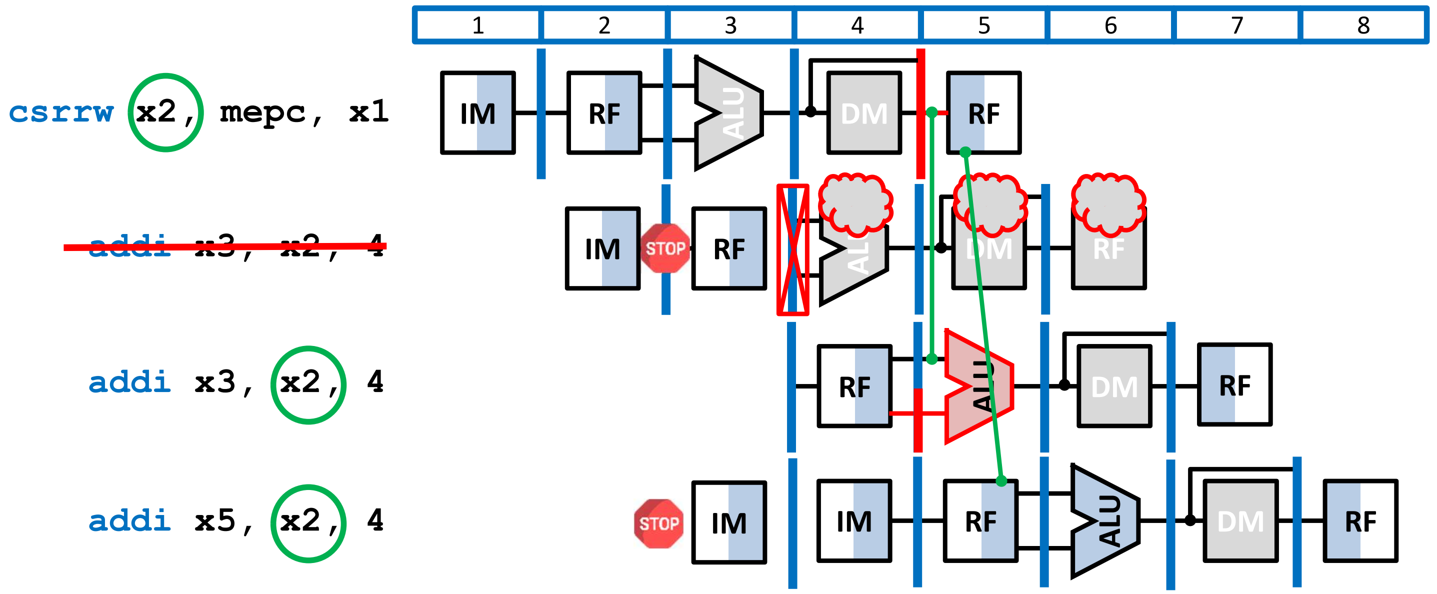


- It is solved as with any data hazard:
  - With forwarding or writing the RF in the first half of the cycle.
  - Also, there is a one-cycle stall if the previous instruction is a **`lw`**.
  - No changes are required in the forwarding/hazard unit.

Pipelined processor

Hazards (iii)

- There is a data hazard when executing a **`csrrw`** instruction that writes the same register that a following instruction reads.

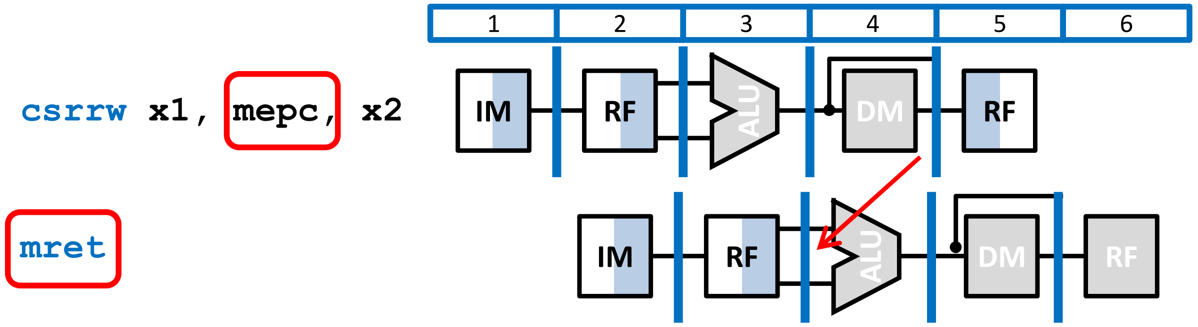


- It is solved in a similar way to the **`lw`** case:
  - Stalling the instruction after **`csrrw`** one cycle and forwarding the data.
  - The hazard unit must be extended to stall the pipeline if a **`csrrw`** instruction in EX with hazard is detected.

Pipelined processor

Hazards (iv)

- There is an implicit data hazard due to **`mepc`**, when a **`csrrw`** instruction that updates **`mepc`** is followed by a **`mret`** instruction:
  - **`csrrw`** writes **`mepc`** in the MEM stage.
  - **`mret`** reads **`mepc`** in the EX stage to determine the branch target address.



- It is solved as with any data hazard:
  - Writing **`mepc`** at the end of the first half of the clock cycle.
  - It requires changes in the data path.

Pipelined processor

Hazards (v)

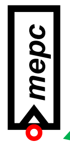
- In summary, to make the pipelined processor with exception handling able to solve hazards, the following is needed:
  - Add the forwarding multiplexers to the data path.
  - Add the forwarding unit.
  - Add the hazard unit with the following modifications:

```
Stall   ← if ( (((ResSrcE = 0) & BRwrE) | isCsrrwE )) & ((Rs1D = RdE) | (Rs2D = RdE)) ) then   ( 1 )
          else                                                                                 ( 0 )
StallF  ← Stall
StallD  ← Stall
FlushE ← Stall | PCsrcE | isMretE
FlushD ← PCsrcE | isMretE
```

The pipeline is also stalled if there is a **`csrrw`** *instruction with hazard in the EX stage*

The IF/ID and ID/EX registers are also deleted if there is a **`mret`** *instruction in the EX stage*

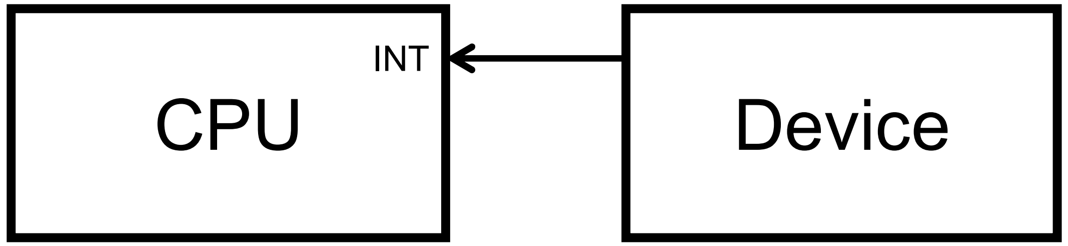
- Invert the clock input of the **`mepc`** register:



When inverting the clock input, **`mepc`** *loads in the falling edge (rising edge half cycle)*

Interrupts

- Interrupts are traps triggered by external devices.
  - This is achieved by activating an input signal to the processor.



- The interrupt handling, from the point of view of the processor, is similar to the exception handling, with a difference:
  - In the exceptions, the current instruction is flushed (the one that produces the exception) before branching to the service routine.
  - In the interrupts, the current instruction is finished (during whose execution the interrupt was triggered) and the processor branches afterwards.

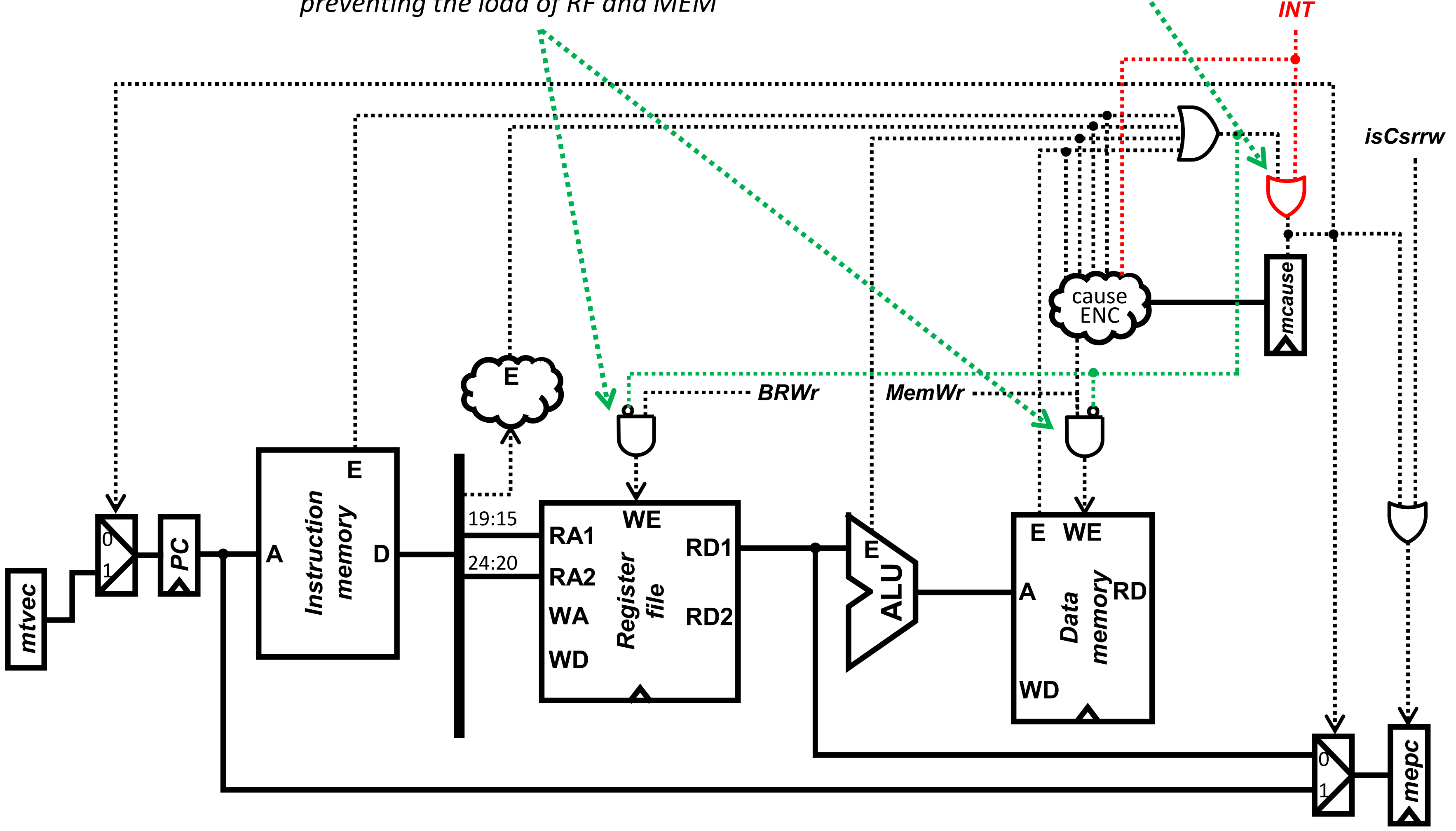
- The interrupt handling, as a whole, is more complex and will be studied in later courses.

Single-cycle processor

Interrupt handling

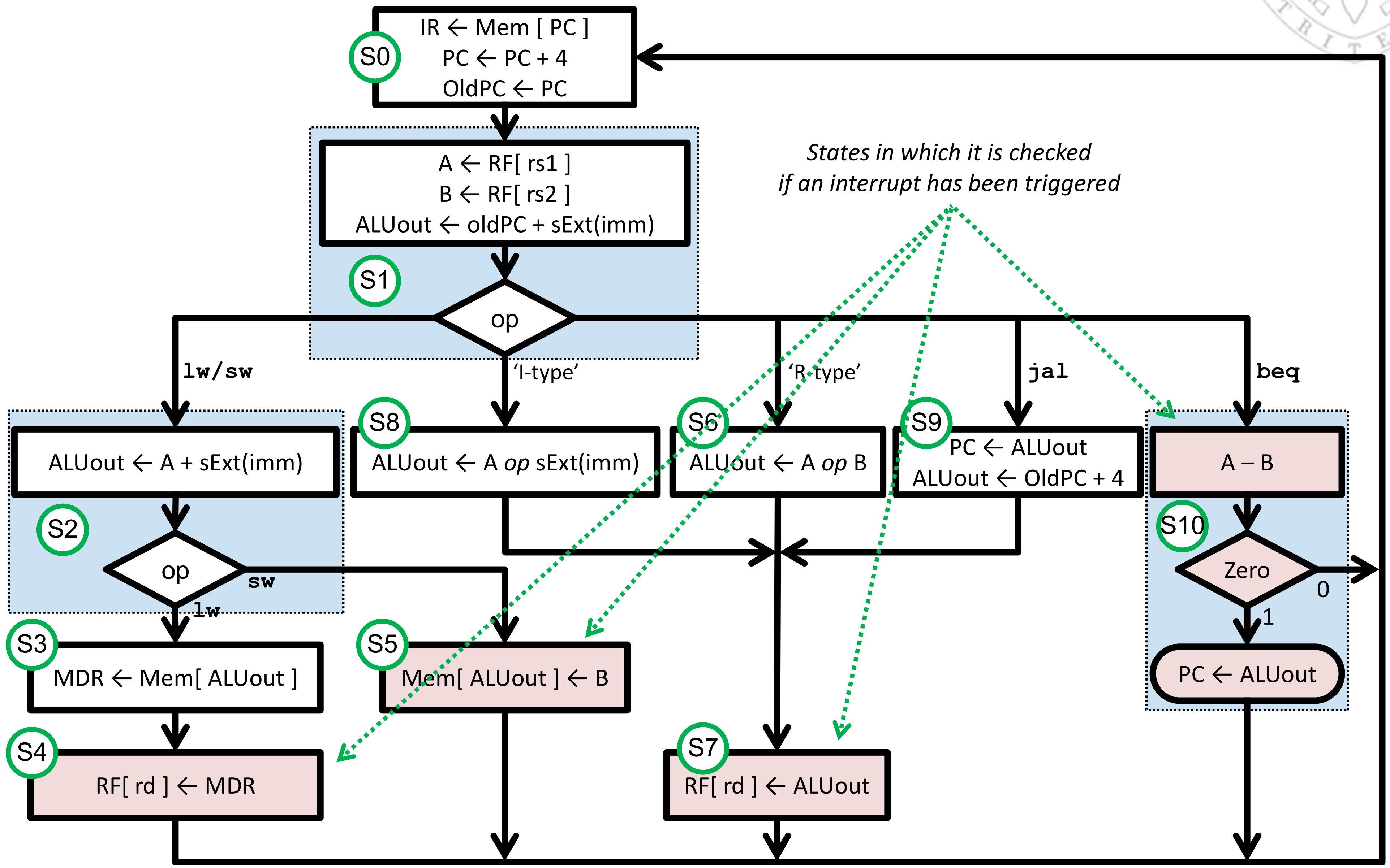
if exception or interrupt:
PC ← mtvec, mepc ← PC, mcause ← *"cause"*

only if exception: *cancels the instruction preventing the load of RF and MEM*



Multicycle processor

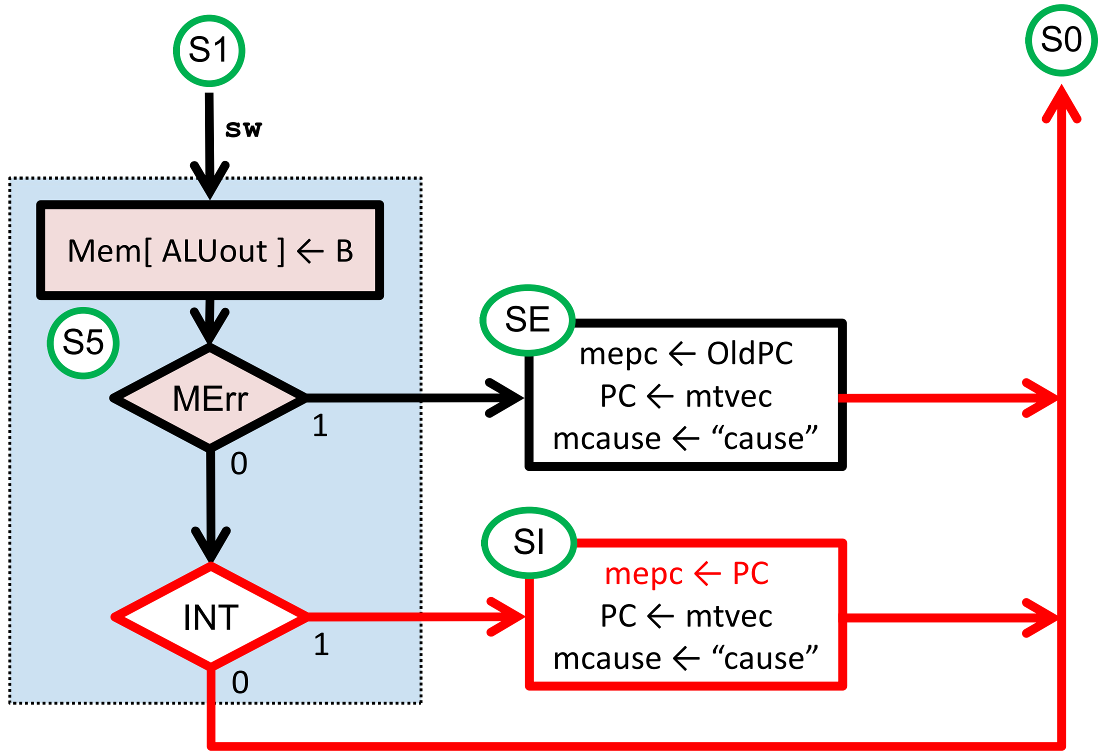
Interrupt handling (i)



Multicycle processor

Interrupt handling (ii)

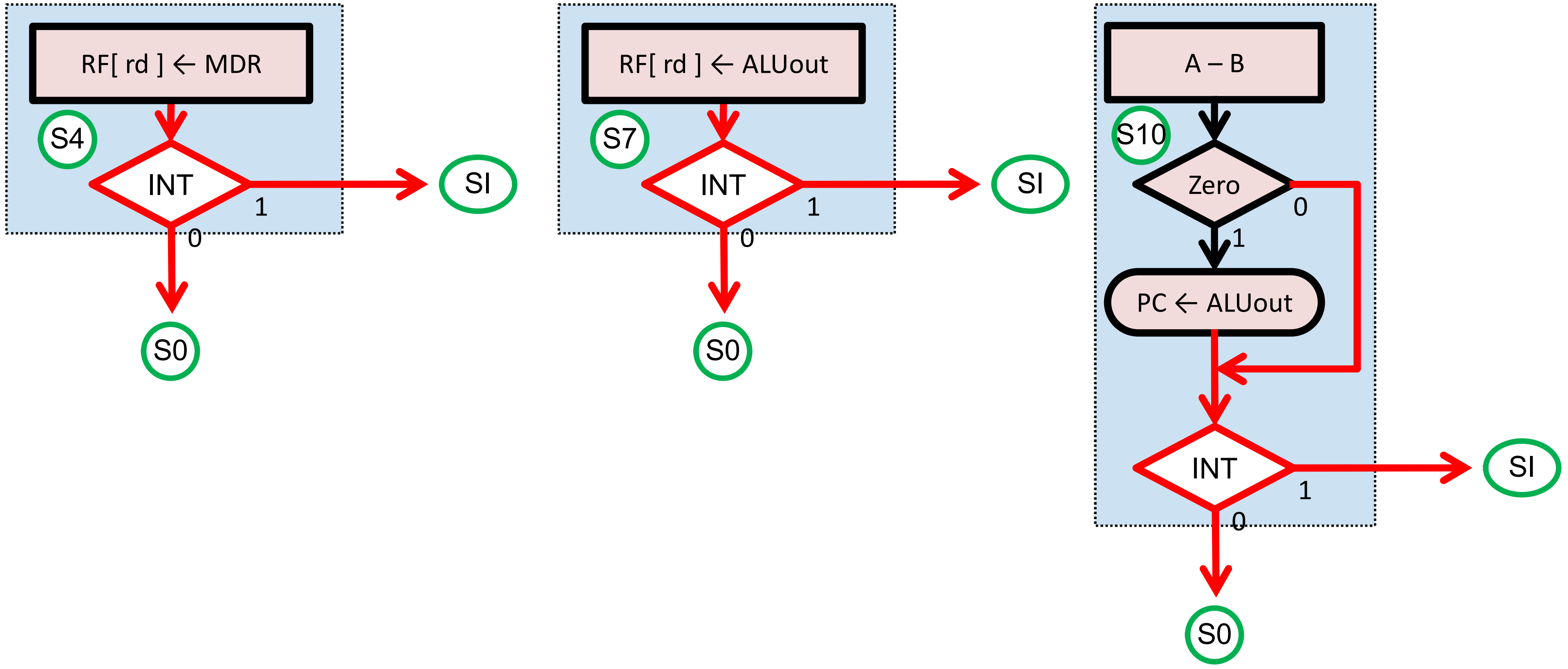
- The SI state is added to handle interrupts.
  - SI updates CSR and since the instruction has finished, it saves the address of the following instruction (stored in PC) in **`mepc`**.
  - SI branches to S0 to initiate the execution of the service routine.



Multicycle processor

Interrupt handling (iii)

- All the final states of every instruction (when it has finished) check the interrupt signal to decide whether to branch to SI or not.

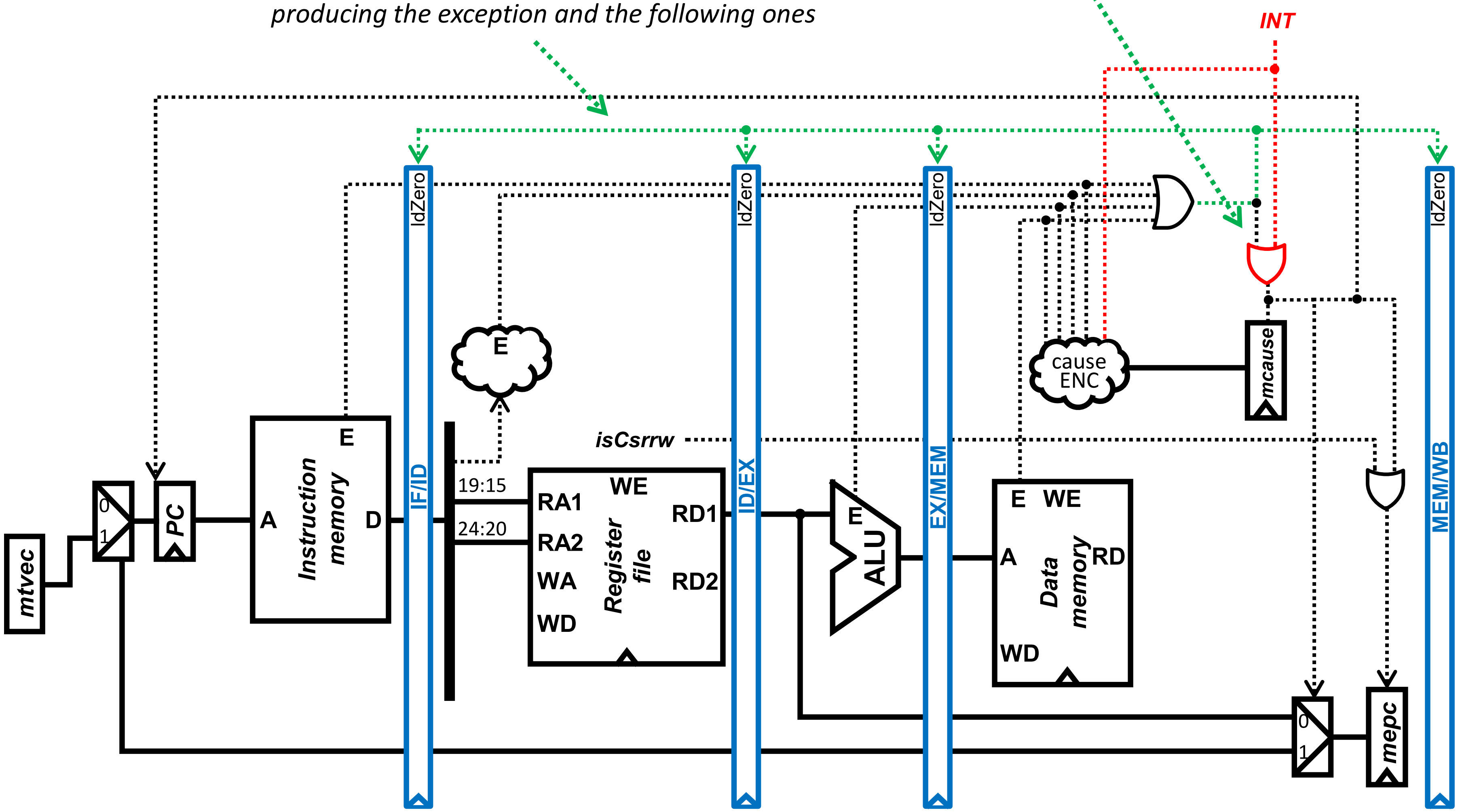


Pipelined processor

Interrupt handling

if exception or interrupt:
PC ← mtvec, mepc ← PC, mcause ← *"cause"*

only if exception: *flushes the instruction producing the exception and the following ones*

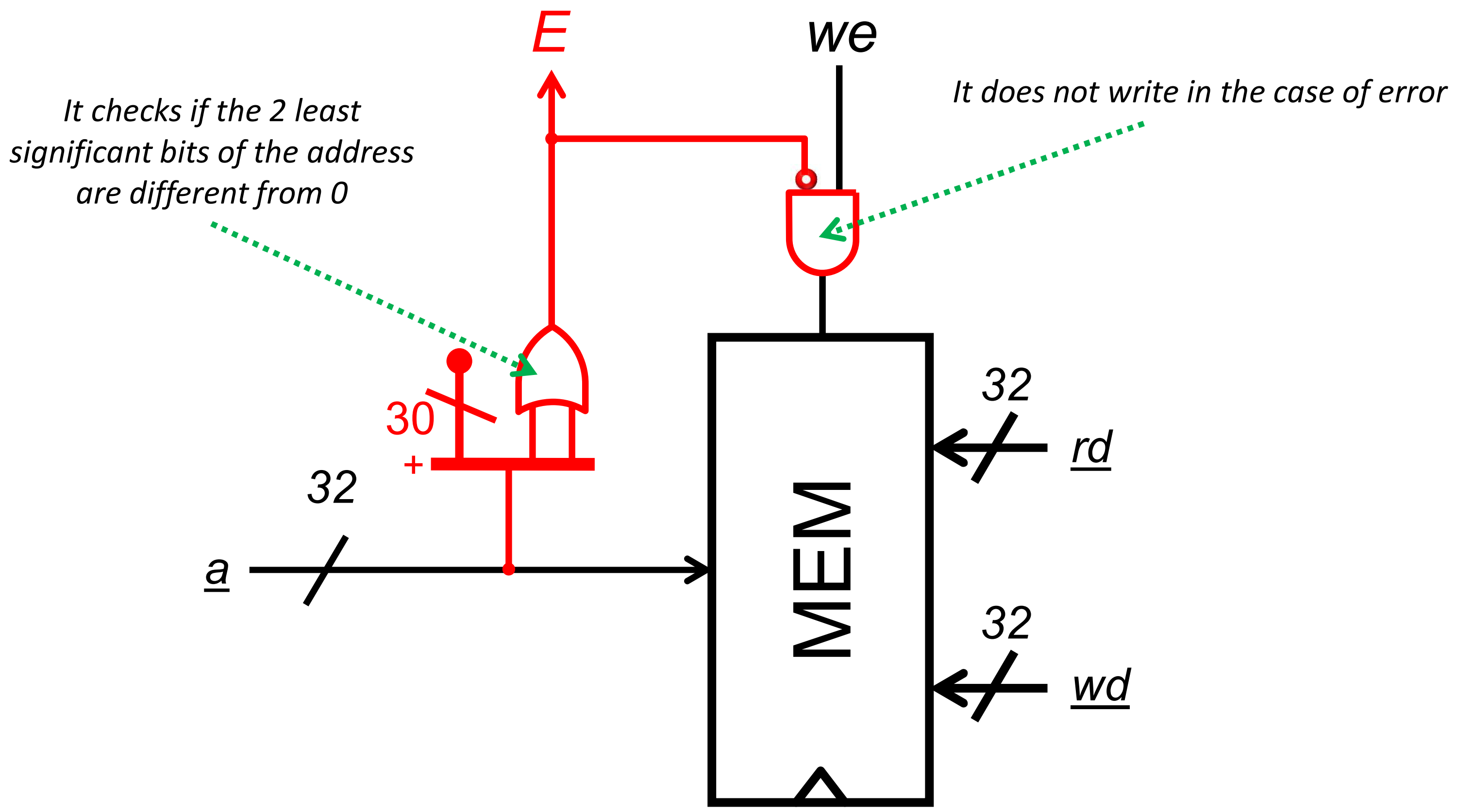


- Memory redesign.
- ALU redesign.
- Design of the illegal instruction detector.
- Single-cycle controller redesign.
- Multicycle controller redesign.

Technology

Memory redesign

- Some logic is added to flag an error in case of misaligned word access (addresses not multiple of 4):
  - Apart from indicating the error, writing is avoided in this situation.

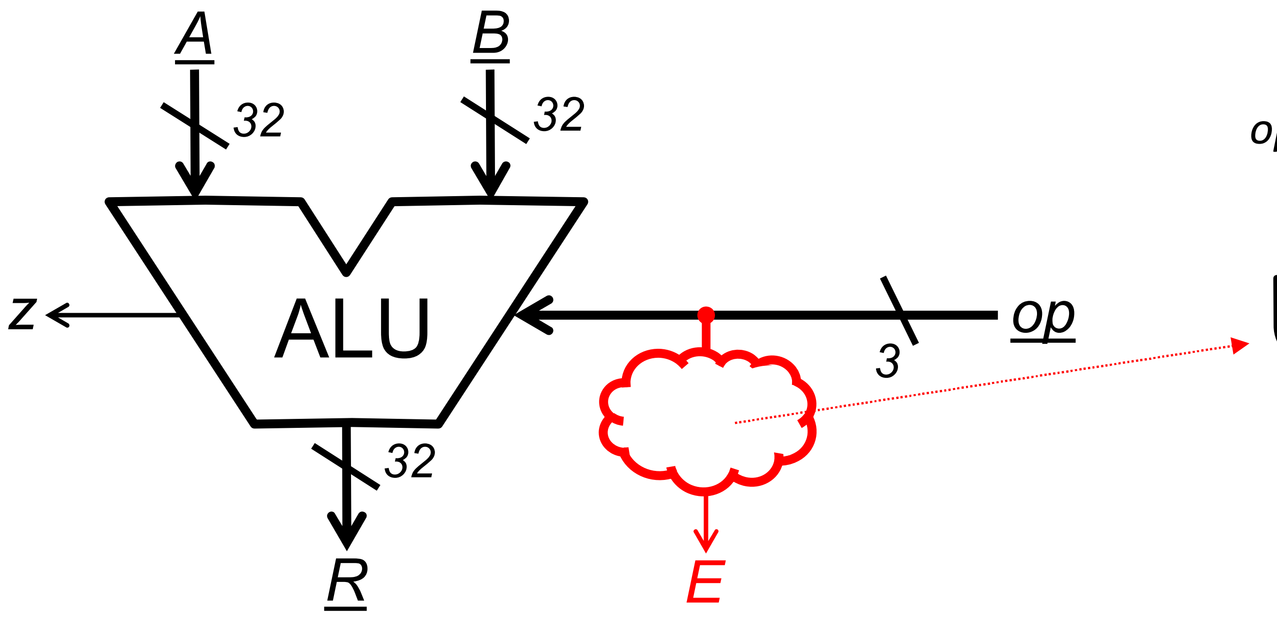

E
we
It checks if the 2 least significant bits of the address are different from 0
It does not write in the case of error
30
+
32
a
MEM
32
rd
32
wd

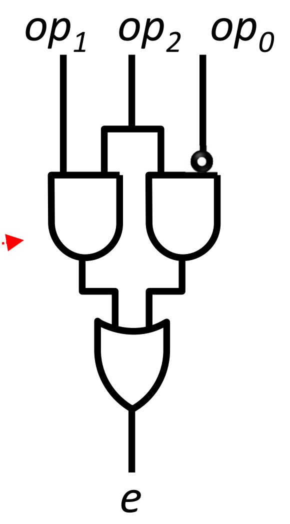
ALU redesign

- Some logic is added to flag an error in case of non-implemented arithmetic-logic operation:

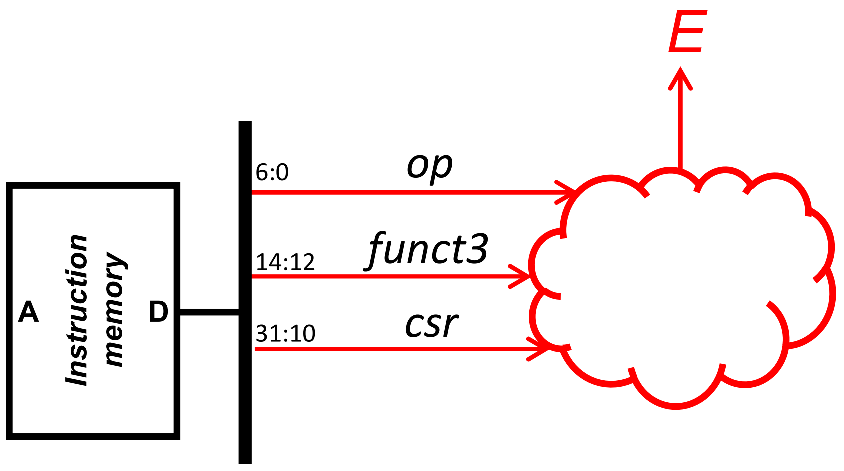
op_2	op_1	op_0	R	E
0	0	0	A + B	0
0	0	1	A – B	0
1	0	0	–	1
1	0	1	*if* (A<B) *then* 1 *else* 0	0

op_2	op_1	op_0	R	E
0	1	0	A & B	0
0	1	1	A | B	0
1	1	0	–	1
1	1	1	–	1





Design of the illegal instruction detector



Truth table

op	funct3	csr	E
0000011 (`lw`)	X	X	0
0100011 (`sw`)	X	X	0
0010011 (I-type)	X	X	0
0110011 (R-type)	X	X	0
1100011 (`beq`)	X	X	0
1101111 (`jal`)	X	X	0
1110011 (mret)	000	X	0
1110011 (csrrw)	001	0x342 (mcause)	0
1110011 (csrrw)	001	0x341 (mepc)	0
others			1

Single-cycle processor

Main DEC redesign

Truth table

op	funct3	csr	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc	isMret	isCsrrw
0000011 (lw)	X	X	0	0	1	1	00(add)	0	000	0	0
0100011 (sw)	X	X	0	0	0	1	00(add)	1	–	0	0
0010011 (I-type)	X	X	0	0	1	1	10(operate)	0	001	0	0
0110011 (R-type)	X	X	0	0	1	0	10(operate)	0	001	0	0
1100011 (beq)	X	X	1	0	0	0	01(subtract)	0	–	0	0
1101111 (jal)	X	X	0	1	1	–	–	0	010	0	0
1110011 (mret)	000	0x302	0	0	0	–	–	0	–	1	0
1110011 (csrrw)	001	0x342 (mcause)	0	0	1	–	–	0	011	0	1
1110011 (csrrw)	001	0x341 (mepc)	0	0	1	–	–	0	100	0	1

Multicycle processor

Main FSM redesign: transition function (i)

Truth table

state	MErr	OpErr	ALUErr	INT	op	funct3	csr	state’
S0	0	X	X	X	X	X	X	S1
S0	1	X	X	X	X	X	X	SE
S1	X	0	X	X	0X00011 (`lw/sw`)	X	X	S2
S1	X	0	X	X	0010011 (I-type)	X	X	S8
S1	X	0	X	X	0110011 (R-type)	X	X	S6
S1	X	0	X	X	1101111 (`jal`)	X	X	S9
S1	X	0	X	X	1100011 (`beq`)	X	X	S10
S1	X	0	X	X	1110011 (`mret`)	000	0x302	S11
S1	X	0	X	X	1110011 (`csrrw`)	001	0x342 (`mcause`)	S12
S1	X	0	X	X	1110011 (`csrrw`)	001	0x341 (`mepc`)	S13
S1	X	1	X	X	X	X	X	SE
S2	X	0	X	X	0000011 (`lw`)	X	X	S3
S2	X	0	X	X	0100011 (`sw`)	X	X	S5
S3	0	X	X	X	X	X	X	S4
S3	1	X	X	X	X	X	X	SE
S4	X	X	X	X	X	X	X	S0
S4	X	X	X	1	X	X	X	SI

Multicycle processor

Main FSM redesign: transition function (ii)

Truth table (cont.)

state	MErr	OpErr	ALUErr	INT	op	funct3	csr	state'
S5	0	X	X	X	X	X	X	S0
S5	1	X	X	X	X	X	X	SE
S5	0	X	X	1	X	X	X	SI
S6	X	X	0	X	X	X	X	S7
S6	X	X	1	X	X	X	X	SE
S7	X	X	X	X	X	X	X	S0
S7	0	X	X	1	X	X	X	SI
S8	X	X	0	X	X	X	X	S7
S8	X	X	1	X	X	X	X	SE
S9	X	X	X	X	X	X	X	S7
S10	X	X	X	X	X	X	X	S0
S10	0	X	X	1	X	X	X	SI
S11	X	X	X	X	X	X	X	S0
S12	X	X	X	X	X	X	X	S0
S13	X	X	X	X	X	X	X	S0
SE	X	X	X	X	X	X	X	S0
SI	X	X	X	X	X	X	X	S0

Multicycle processor

Main FSM redesign: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	IRwr	BRwr	ALUsrcA	ALUsrcB	ALUop	ResSrc	CauseWr	EPCWr
S0	0	1	0	0	1	0	00	10	00	010	0	0
S1	0	0	–	0	0	0	01	01	00	–	0	0
S2	0	0	–	0	0	0	10	01	00	–	0	0
S3	0	0	1	0	0	0	–	–	–	000	0	0
S4	0	0	–	0	0	1	–	–	–	001	0	0
S5	0	0	1	1	0	0	–	–	–	000	0	0
S6	0	0	–	0	0	0	10	00	10	–	0	0
S7	0	0	–	0	0	1	–	–	–	000	0	0
S8	0	0	–	0	0	0	10	01	10	–	0	0
S9	0	1	–	0	0	0	01	10	00	000	0	0
S10	1	0	–	0	0	0	10	00	01	000	0	0
S11	0	1	–	0	0	0	–	–	–	100	0	0
S12	0	0	–	0	0	1	–	–	–	101	0	0
S13	0	0	–	0	0	1	10	–	–	100	0	1
SE	0	1	–	0	0	0	01	–	–	011	1	1
SI	0	1	–	0	0	0	00	–	–	011	1	1

About *Creative Commons*

- CC license (Creative Commons)
  - This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms:

Attribution:
Credit must be given to the creator.

Non commercial:
Only noncommercial uses of the work are permitted.

Share alike:
Adaptations must be shared under the same terms.

More information: https://creativecommons.org/licenses/by-nc-sa/4.0/