Module 7:

Pipelined processor design

Introduction to computers II

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

Outline

- ✓ Introduction.
- ✓ Data path design.
- ✓ Controller design.
- ✓ Structural hazards.
- ✓ Data hazards.
- ✓ Control hazards.
- ✓ Comparison: single-cycle vs. multicycle vs. pipelined.
- ✓ Advanced microarchitectures.
- ✓ Technology.

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.*

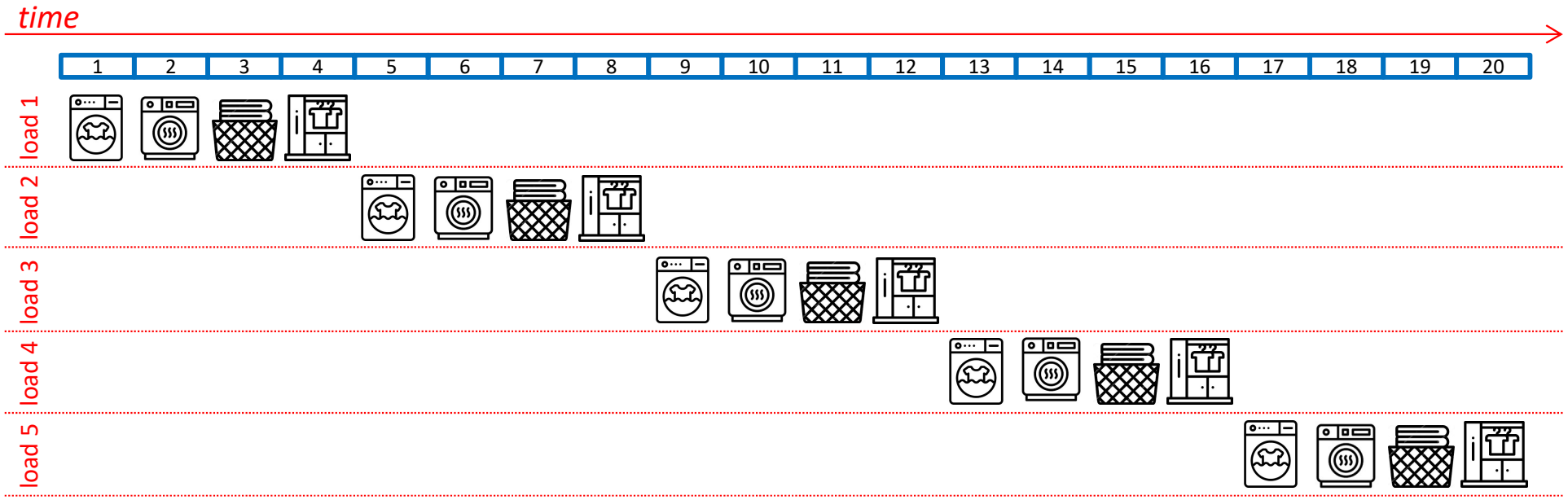
Introduction

- No modern processor is single-cycle.
  - This microarchitecture was only used in the first computers.
- No current processor is multicycle.
  - This microarchitecture was used until the late 80s:
    - Mainframes: IBM/360, DEC VAX
    - Microprocessors: 8088/86 (IBM PC), 68000 (Apple Macintosh), Z80 (Spectrum)
  - Nowadays, it is only used in low-performance microcontrollers:
    - 8051, 68HC11, PIC-16
- Since the 90s, all processors are pipelined.
  - Current processors use even more advanced microarchitectures but based on the pipelining concept.

Introduction

Pipelining (i)

- At home, it is usual to have a sequential laundry:
  - 4 stages with similar duration: wash, dry, iron and store.

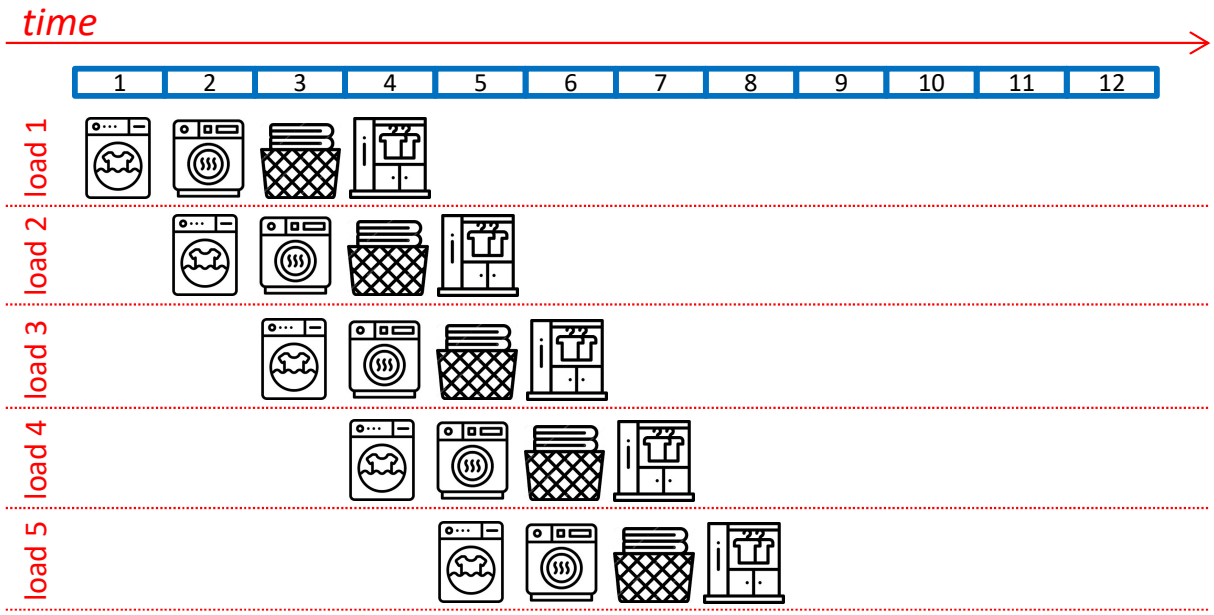


- Each appliance is inactive during 75% of the time.
- 1 load takes 4 units of time.
- 5 loads take 4×5 = 20 units of time.
- n loads take 4·n units of time.

Introduction

Pipelining (ii)

- In an industrial laundry, the process is more efficient:
  - A new load is started even if the previous one has not finished



$$Speedup = \frac{4 \cdot n}{4 + (n - 1)}$$

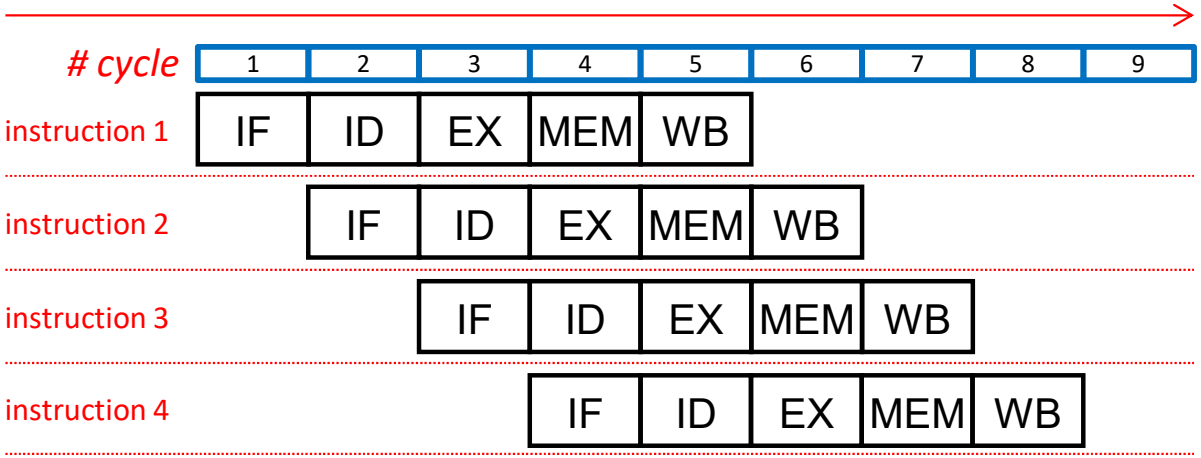
$$\lim_{n \to \infty} \frac{4 \cdot n}{4 + (n - 1)} = 4$$

- Now, appliances are used 100% of the time.
- 1 load still takes 4 units of time.
- 5 loads now take 4 + (5-1) = 8 units of time.
- n loads take 4 + (n-1) units of time.

Introduction

Pipelining (iii)

- A pipelined processor behaves as in the industrial laundry example, overlapping the execution of several instructions.

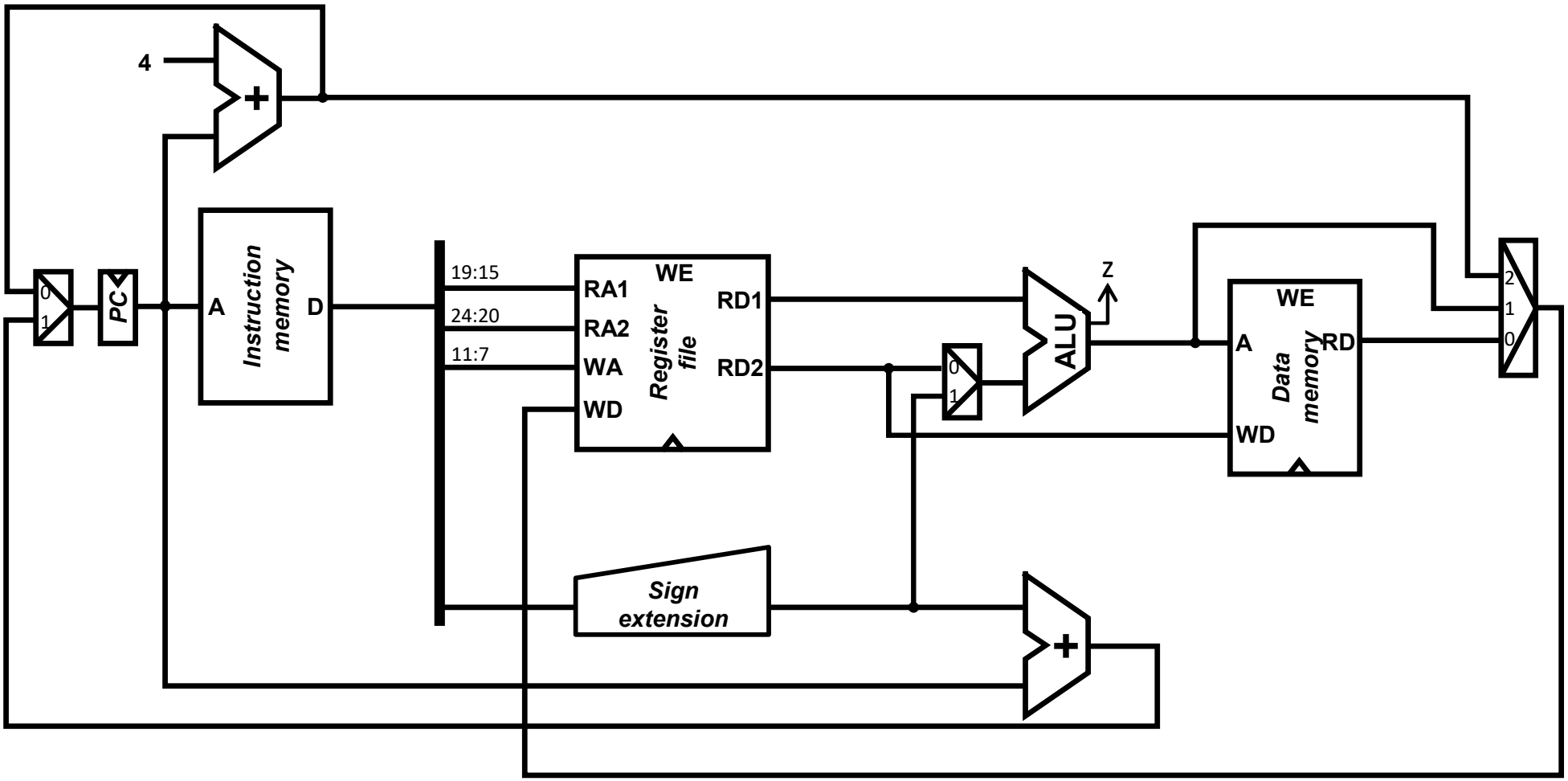


  - Each cycle, a new instruction is fetched before the previous one has finished.
  - Each instruction goes through 5 stages, taking 5 cycles to execute:
    - The latency of this processor is 5 cycles.
- The execution time of a program will be much lower because:
  - Several instructions are executed simultaneously.
  - The cycle time can be shorter (as in the multicycle processor).
  - Ideally, CPI = 1 (as in the single-cycle processor).

Data path design

Reduced RISC-V data path (i)

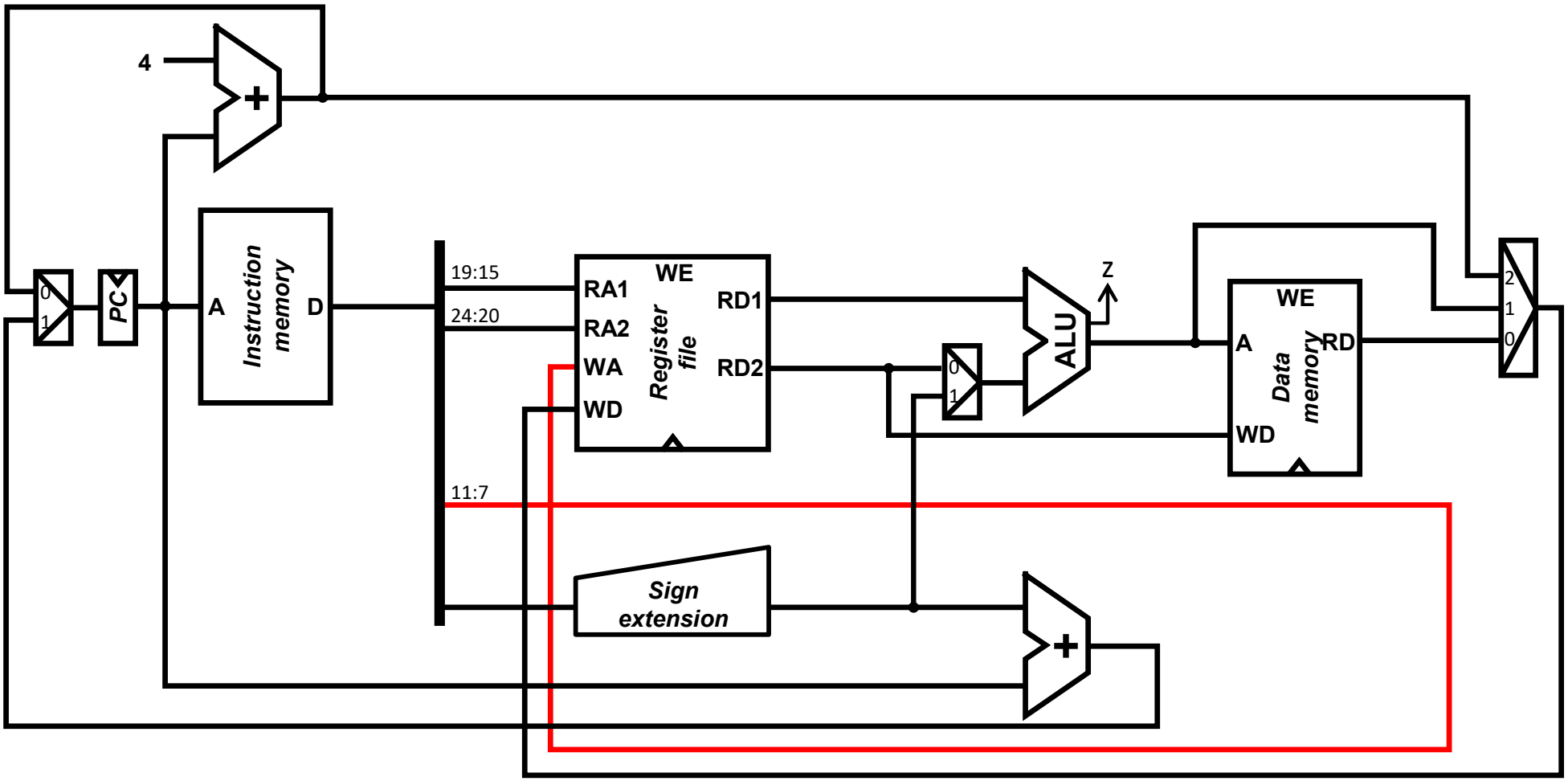
- The data path of the pipelined processor is:
  - single-cycle processor data path + pipeline registers



Data path design

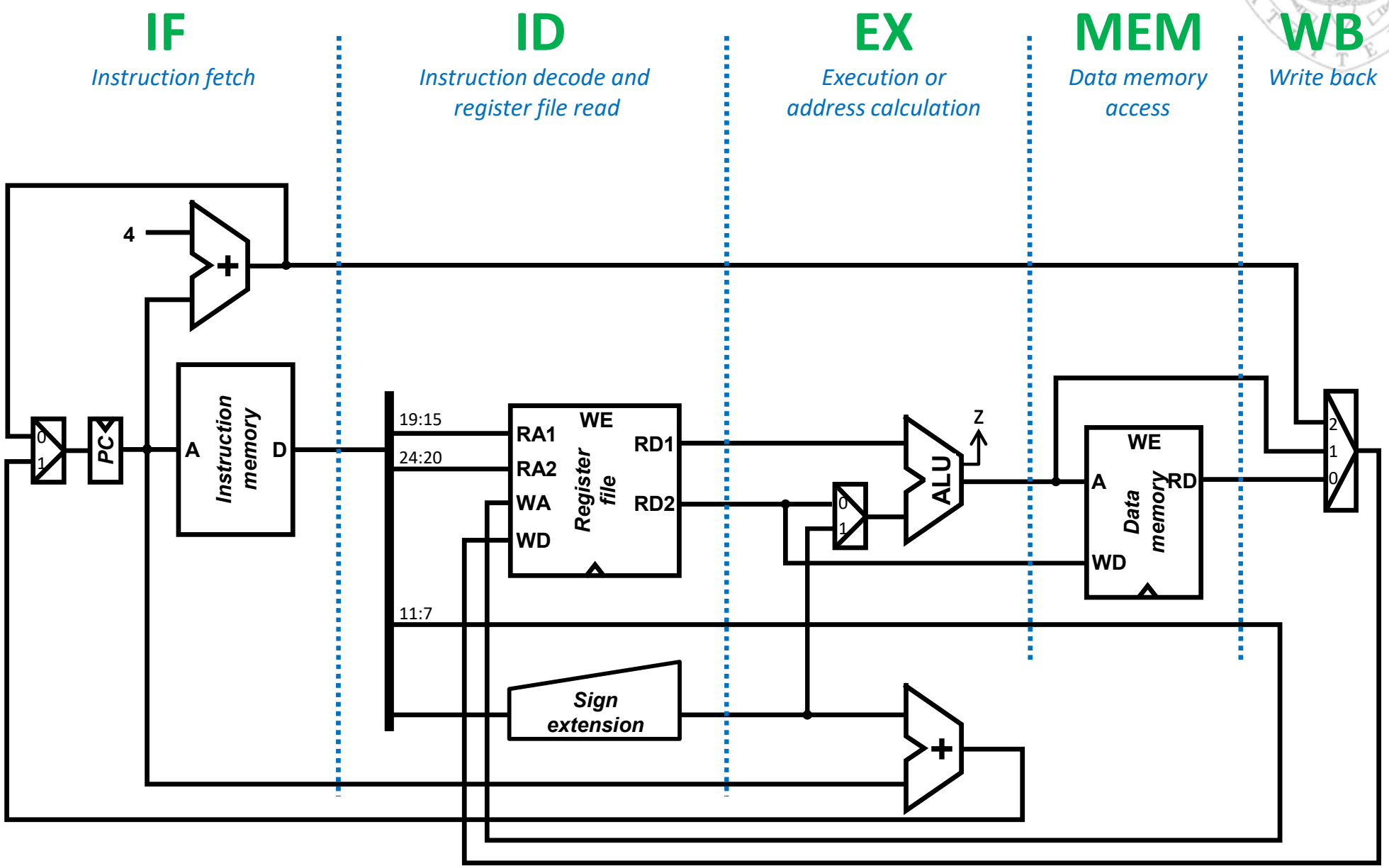
Reduced RISC-V data path (ii)

- The data path of the pipelined processor is:
  - single-cycle processor data path + pipeline registers



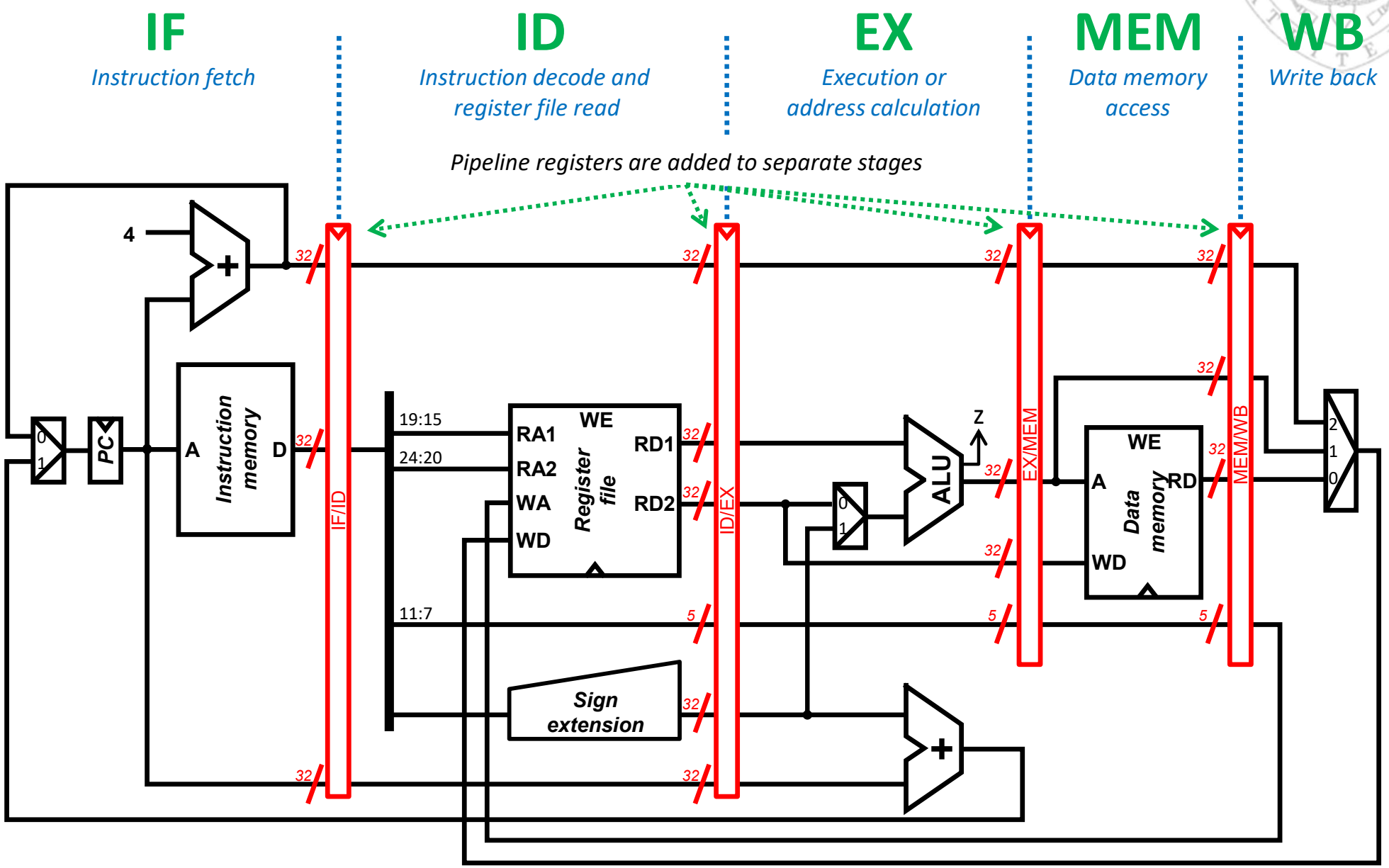
Data path design

Reduced RISC-V data path (iii)



Data path design

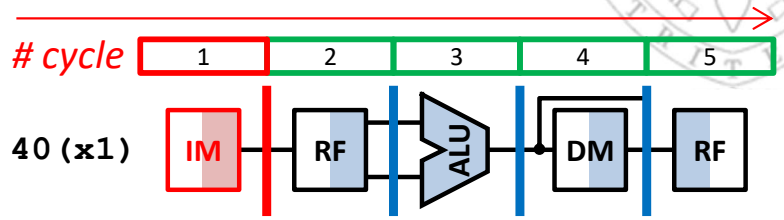
Reduced RISC-V data path (iv)

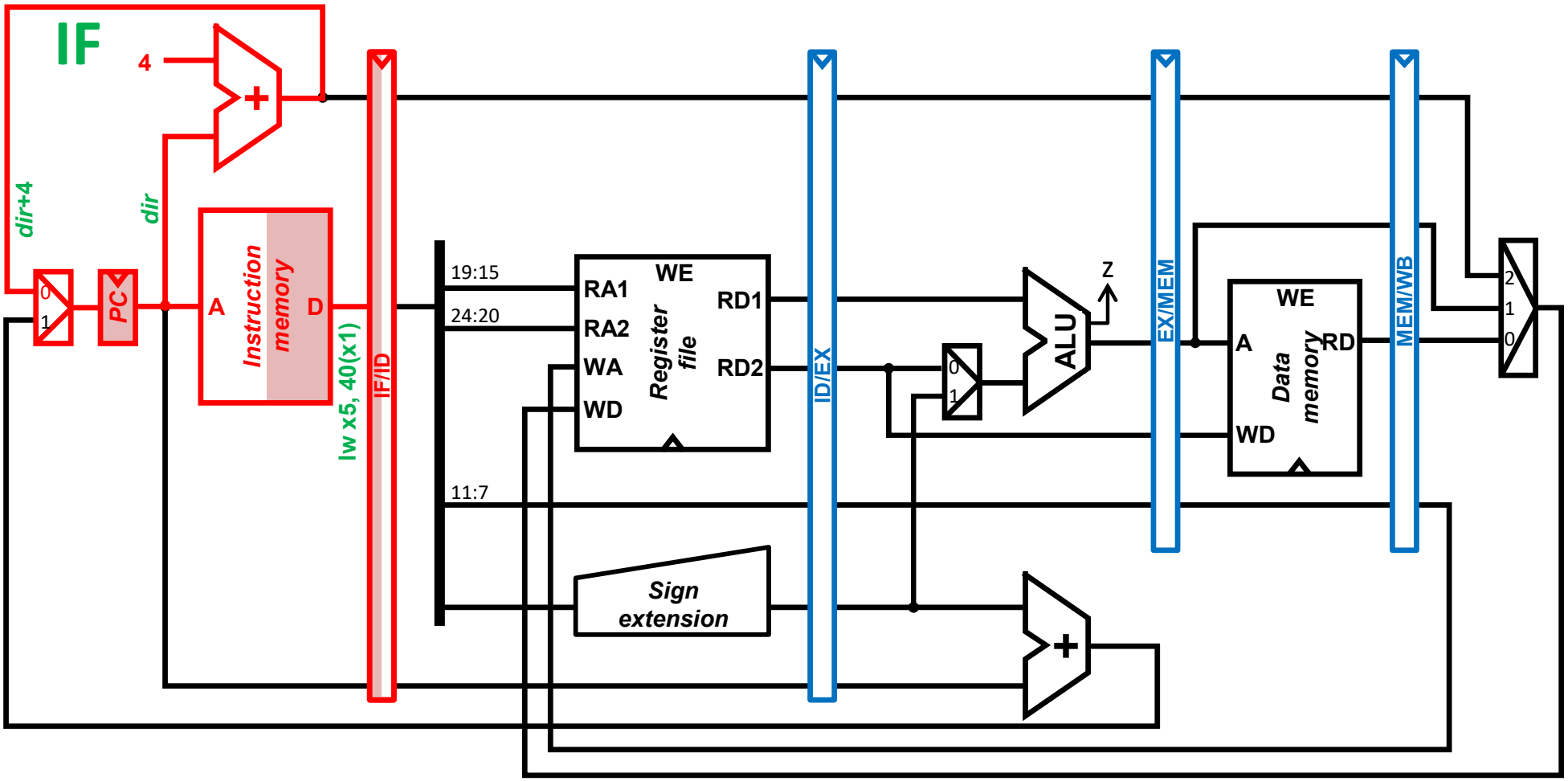


Data path design

`lw` instruction: IF stage

The `lw` load instruction takes 5 cycles using resources in all the stages

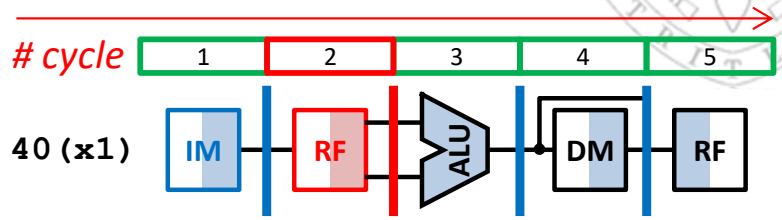


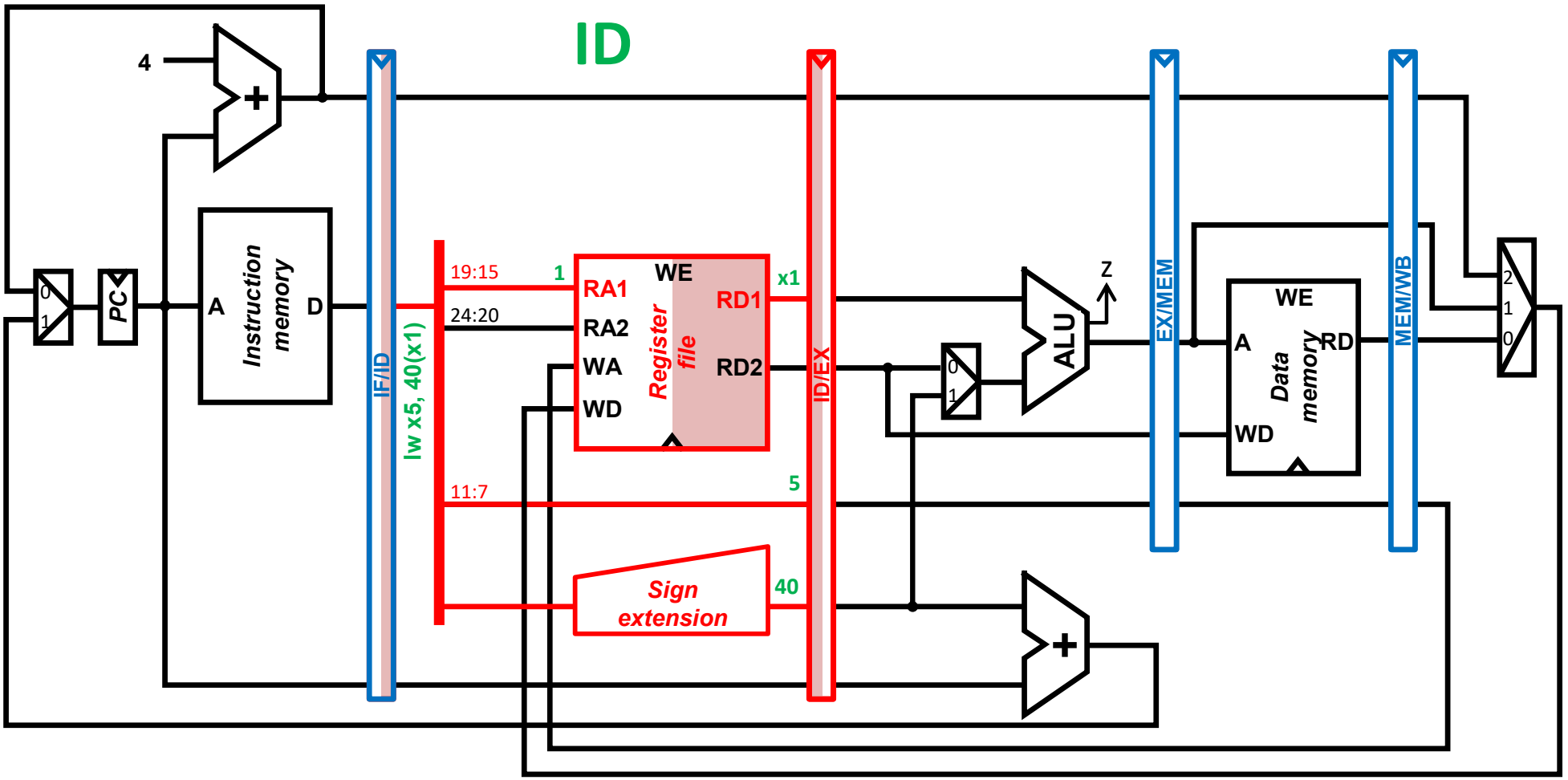


Data path design

`lw` instruction: ID stage

The `lw` load instruction takes 5 cycles using resources in all the stages

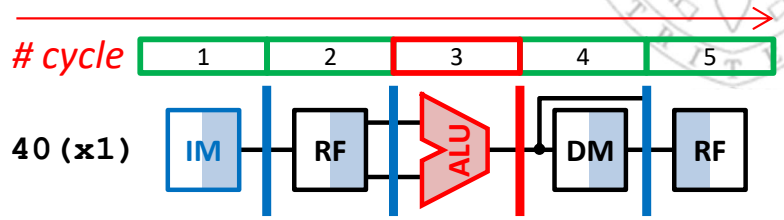


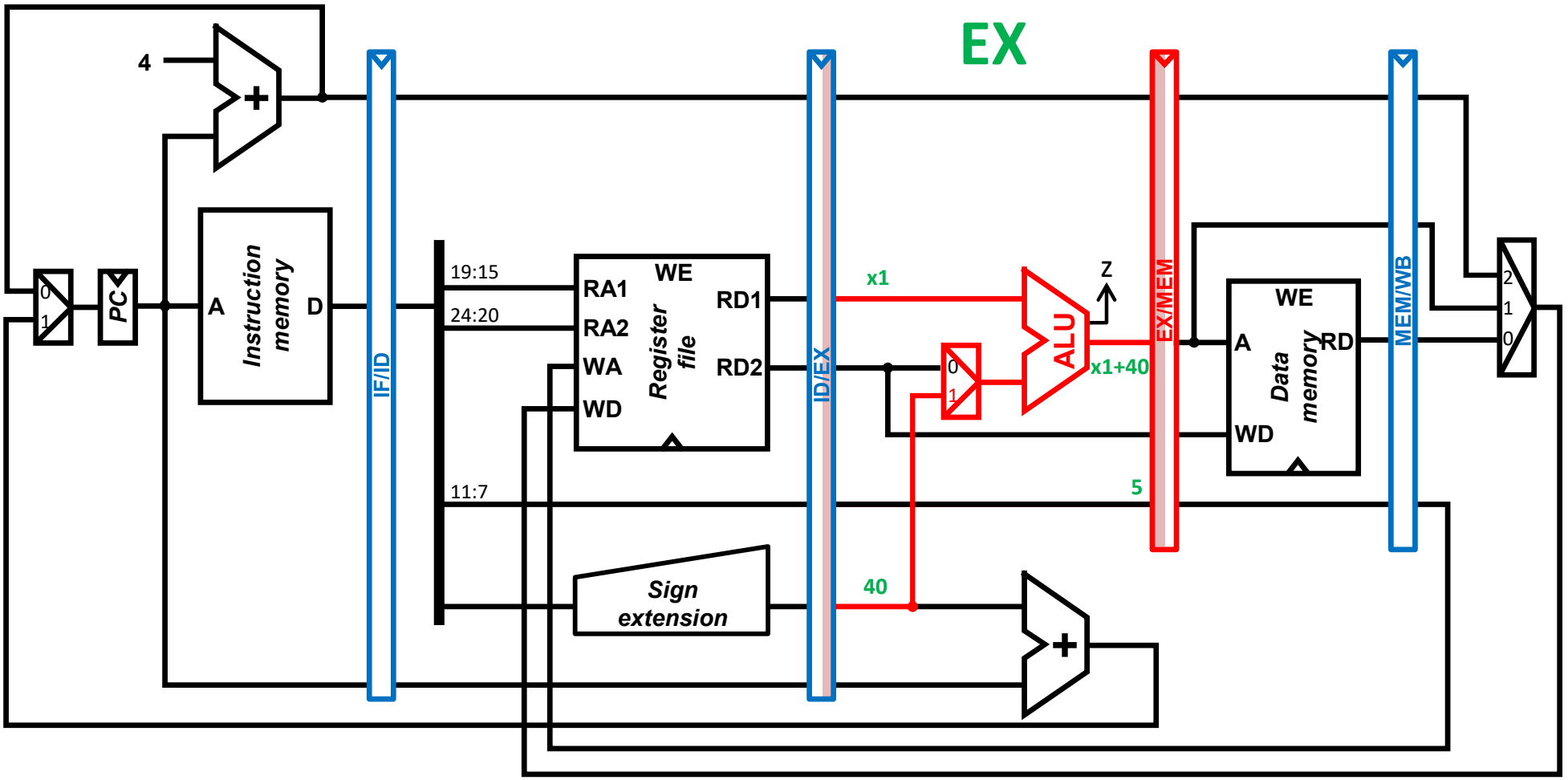


Data path design

`lw` instruction: EX stage

The `lw` load instruction takes 5 cycles using resources in all the stages

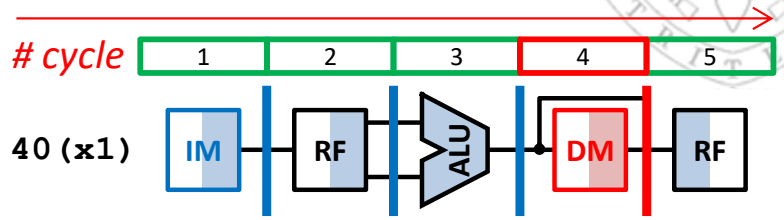


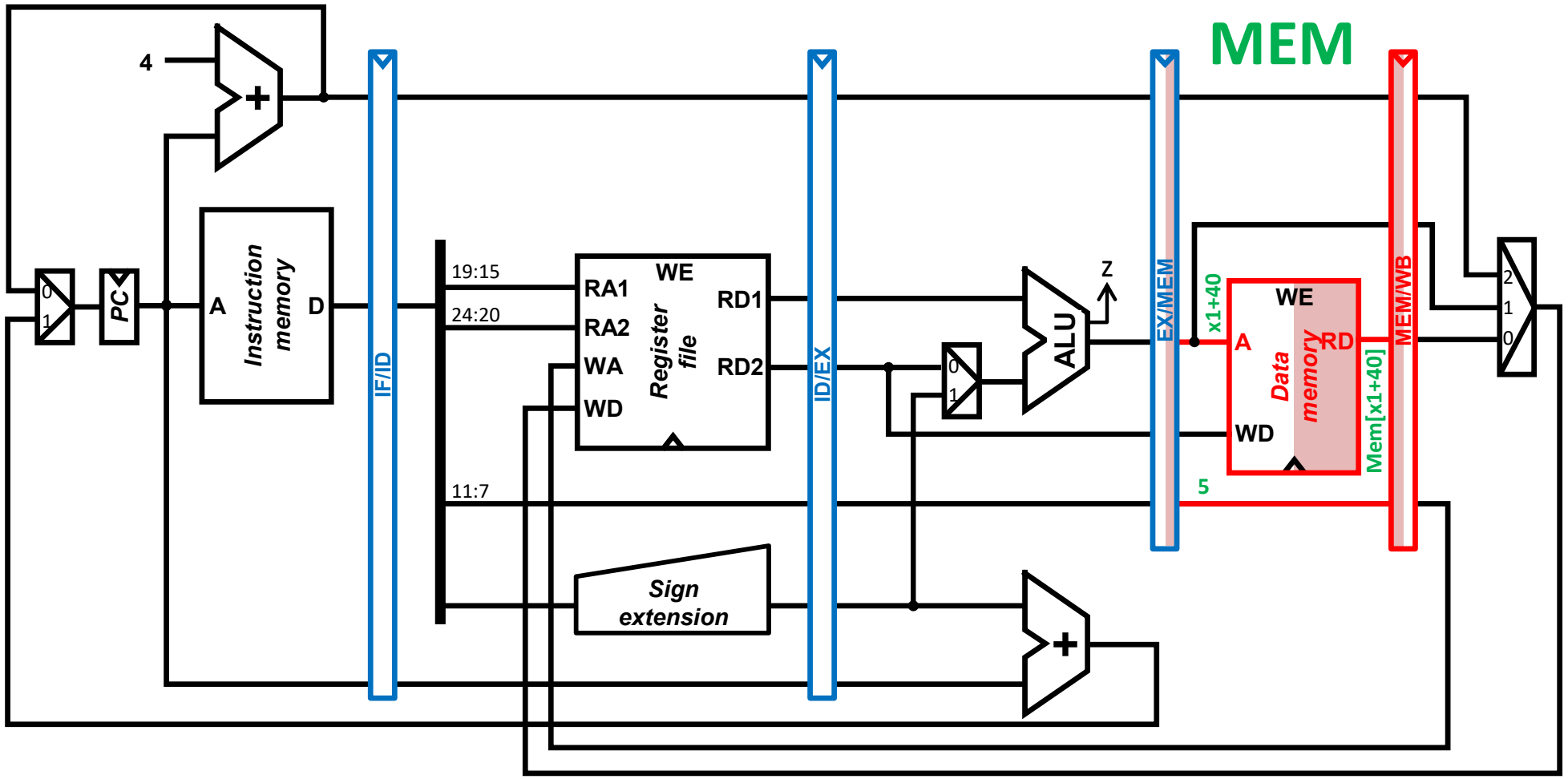


Data path design

`lw` instruction: MEM stage

The `lw` load instruction takes 5 cycles using resources in all the stages

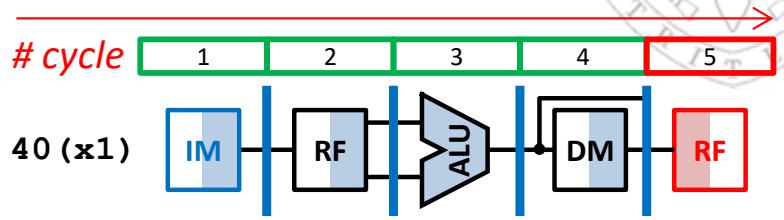


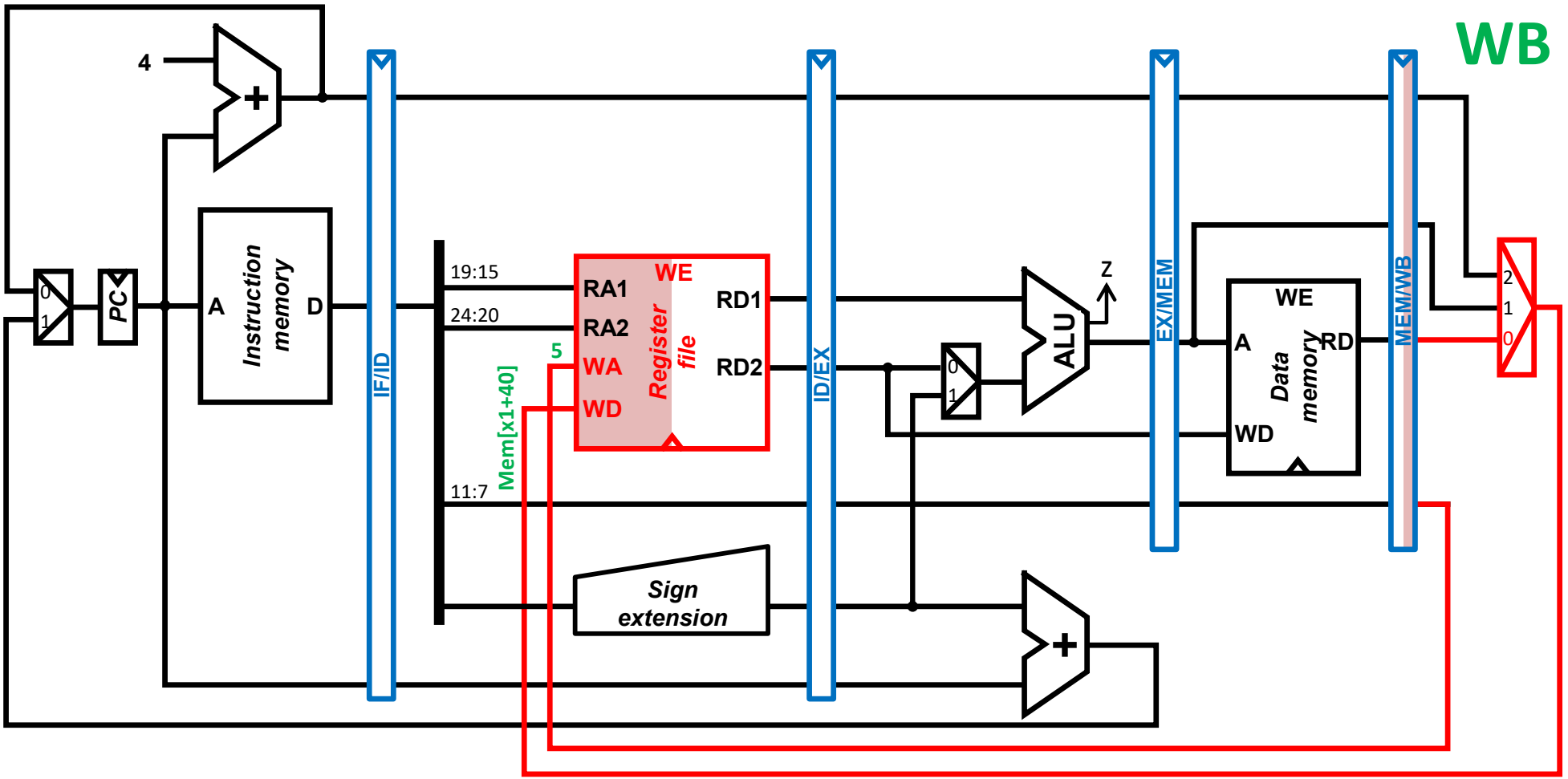


Data path design

`lw` instruction: WB stage

The `lw` load instruction takes 5 cycles using resources in all the stages

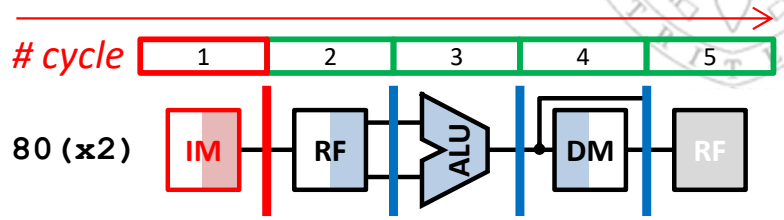


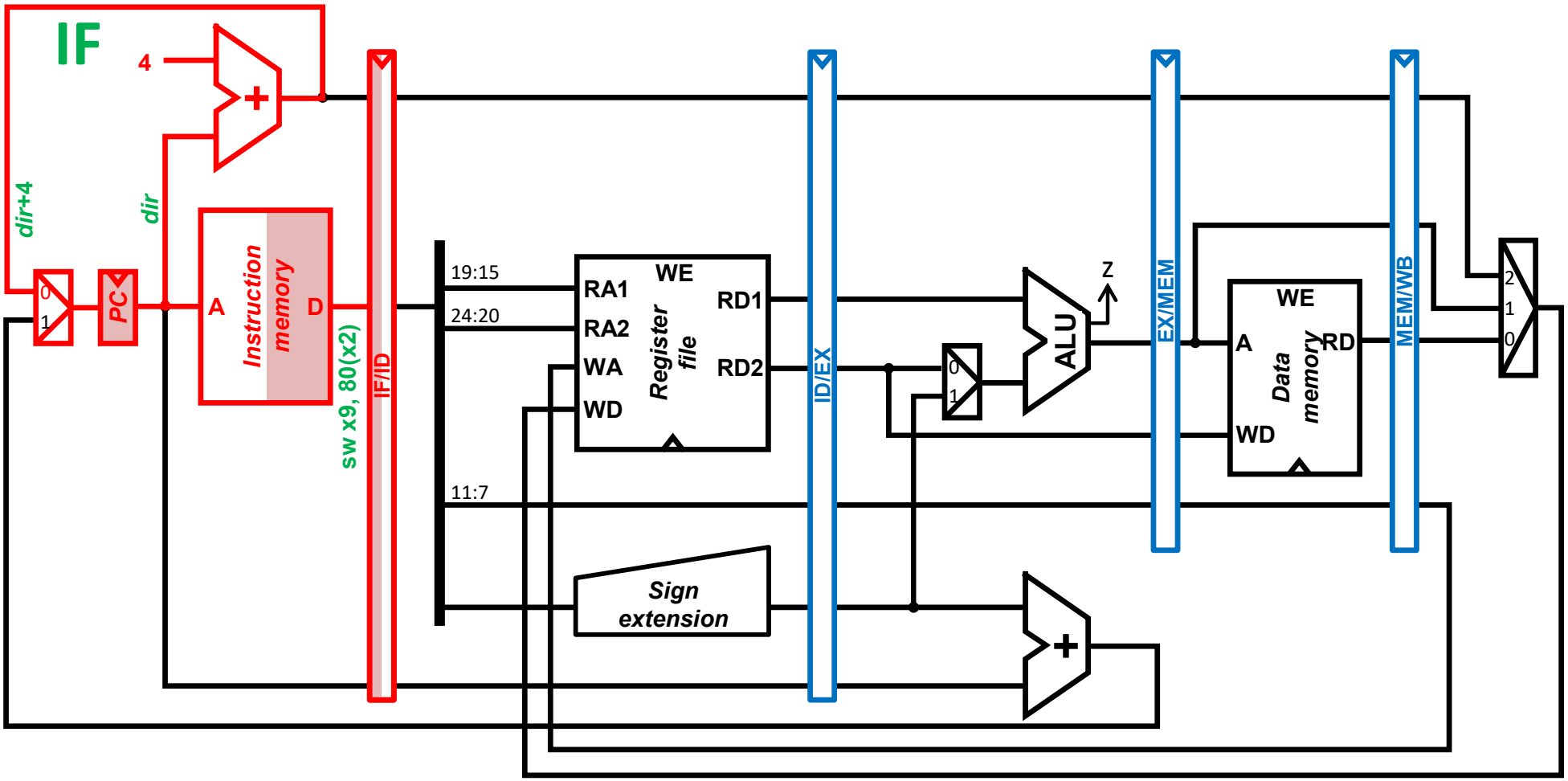


Data path design

`sw` instruction: IF stage

The `sw` *instruction takes 5 cycles without using the RF in the WB stage*

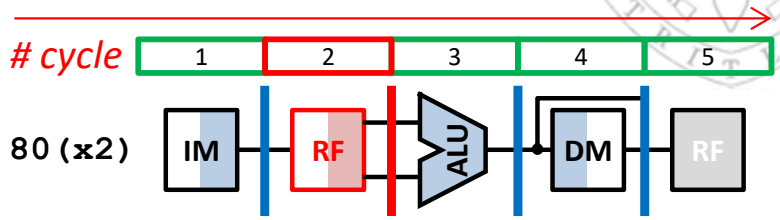


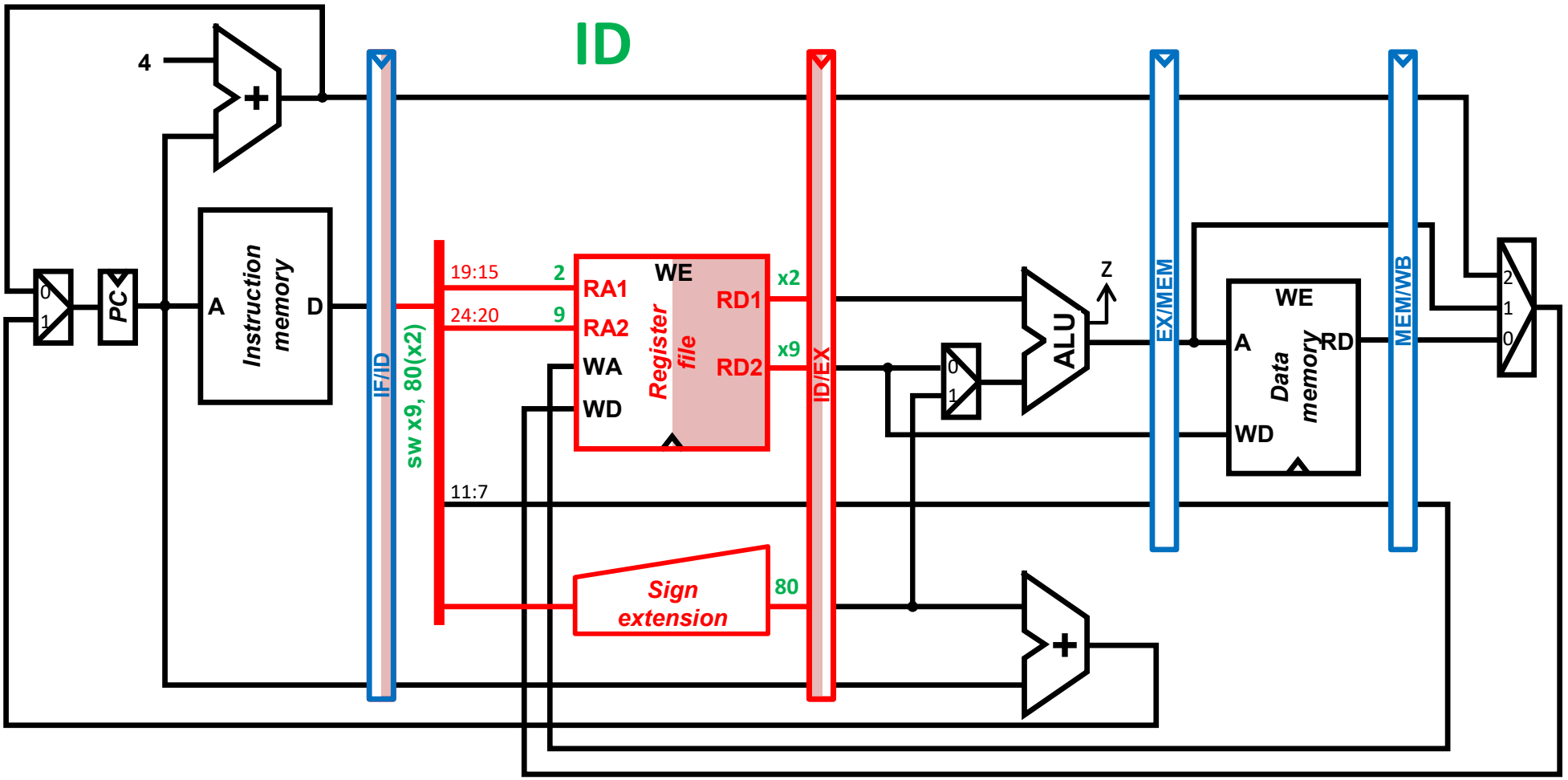


Data path design

`sw` instruction: ID stage

The `sw` *instruction takes 5 cycles without using the RF in the WB stage*

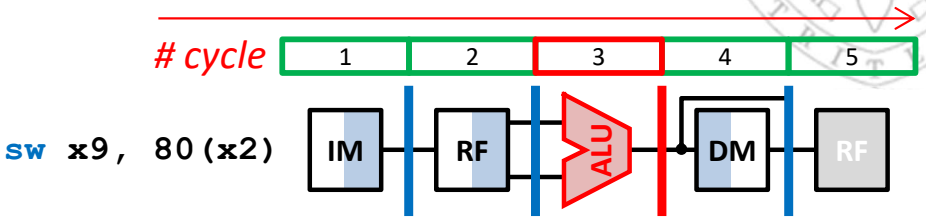


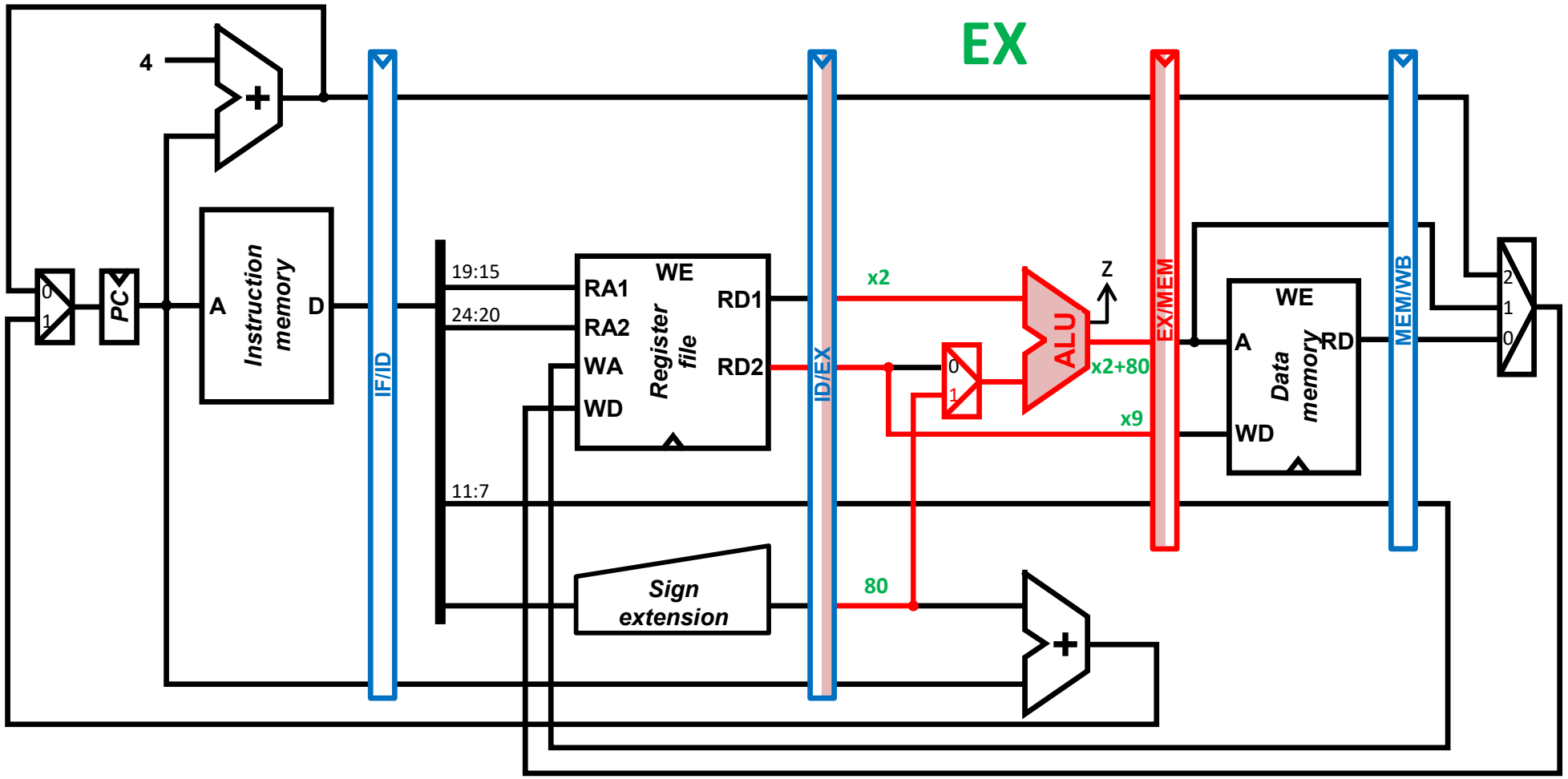


Data path design

`sw` instruction: EX stage

The `sw` instruction takes 5 cycles without using the RF in the WB stage

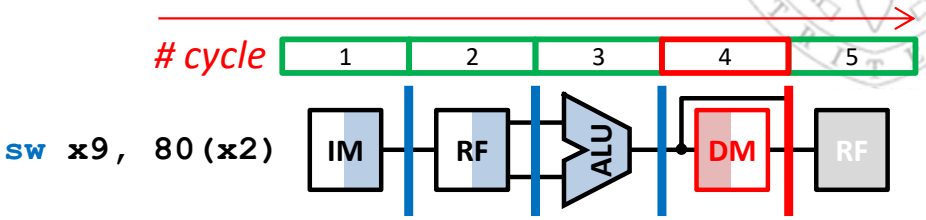


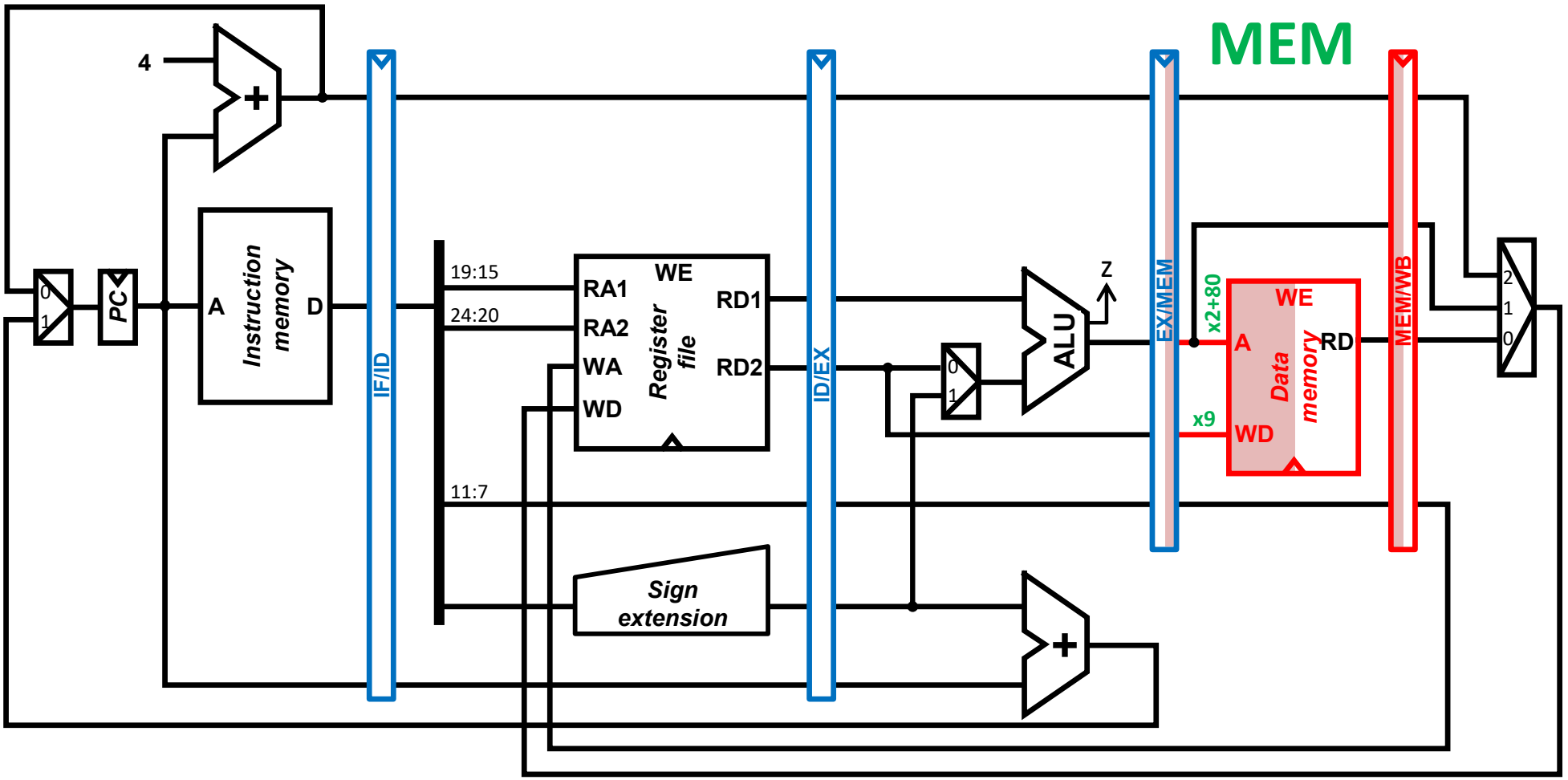


Data path design

`sw` instruction: MEM stage

The `sw` *instruction takes 5 cycles without using the RF in the WB stage*

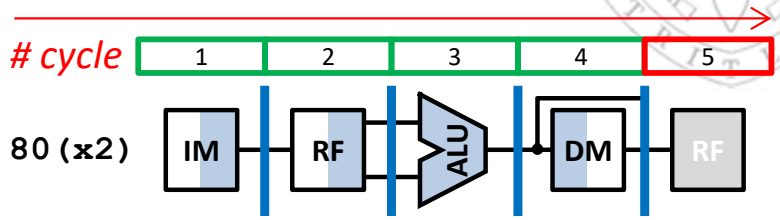


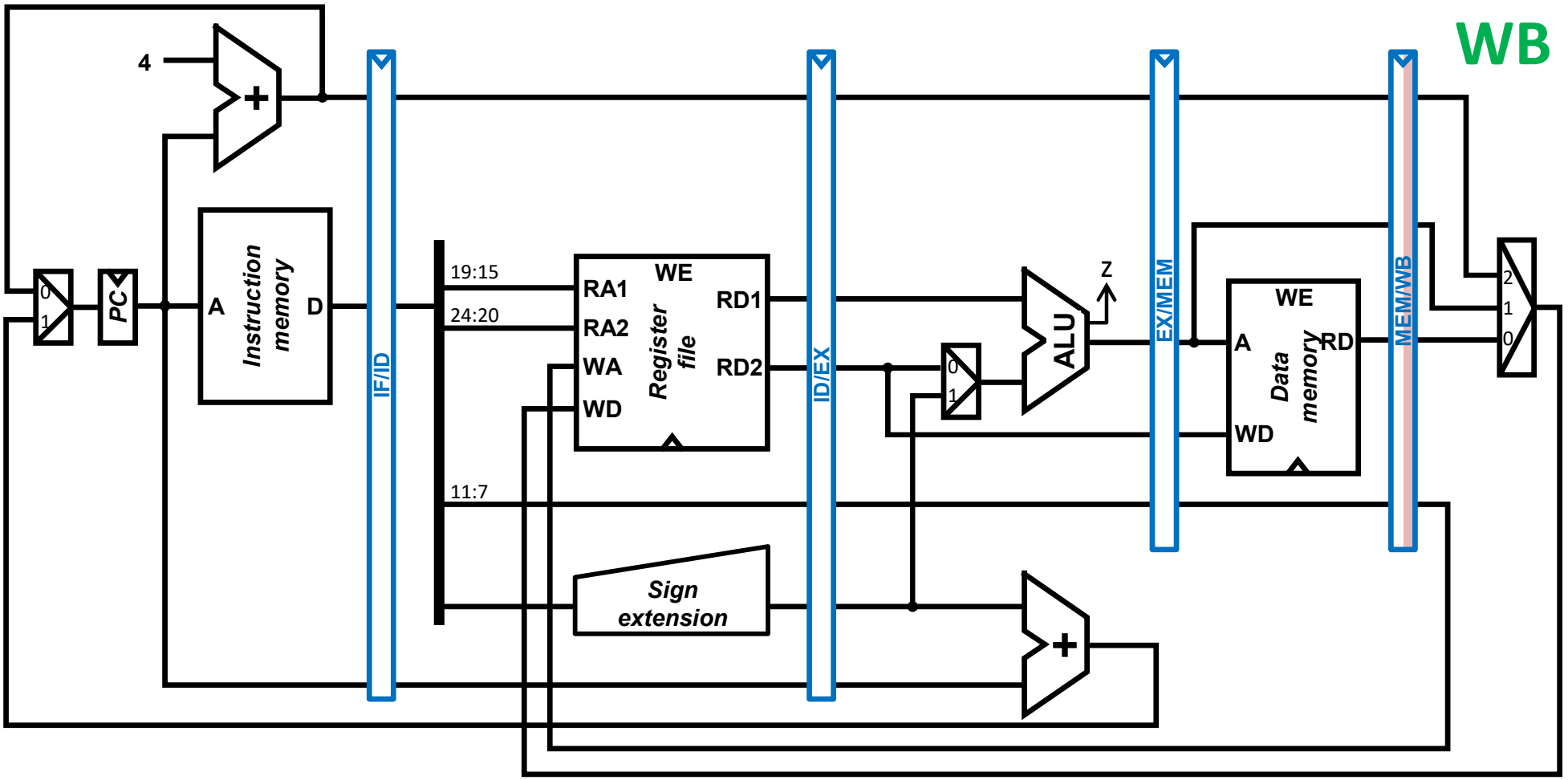


Data path design

`sw` instruction: WB stage

The **`sw`** *instruction takes 5 cycles without using the RF in the WB stage*

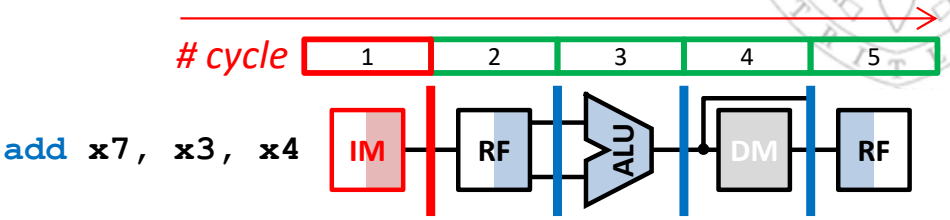


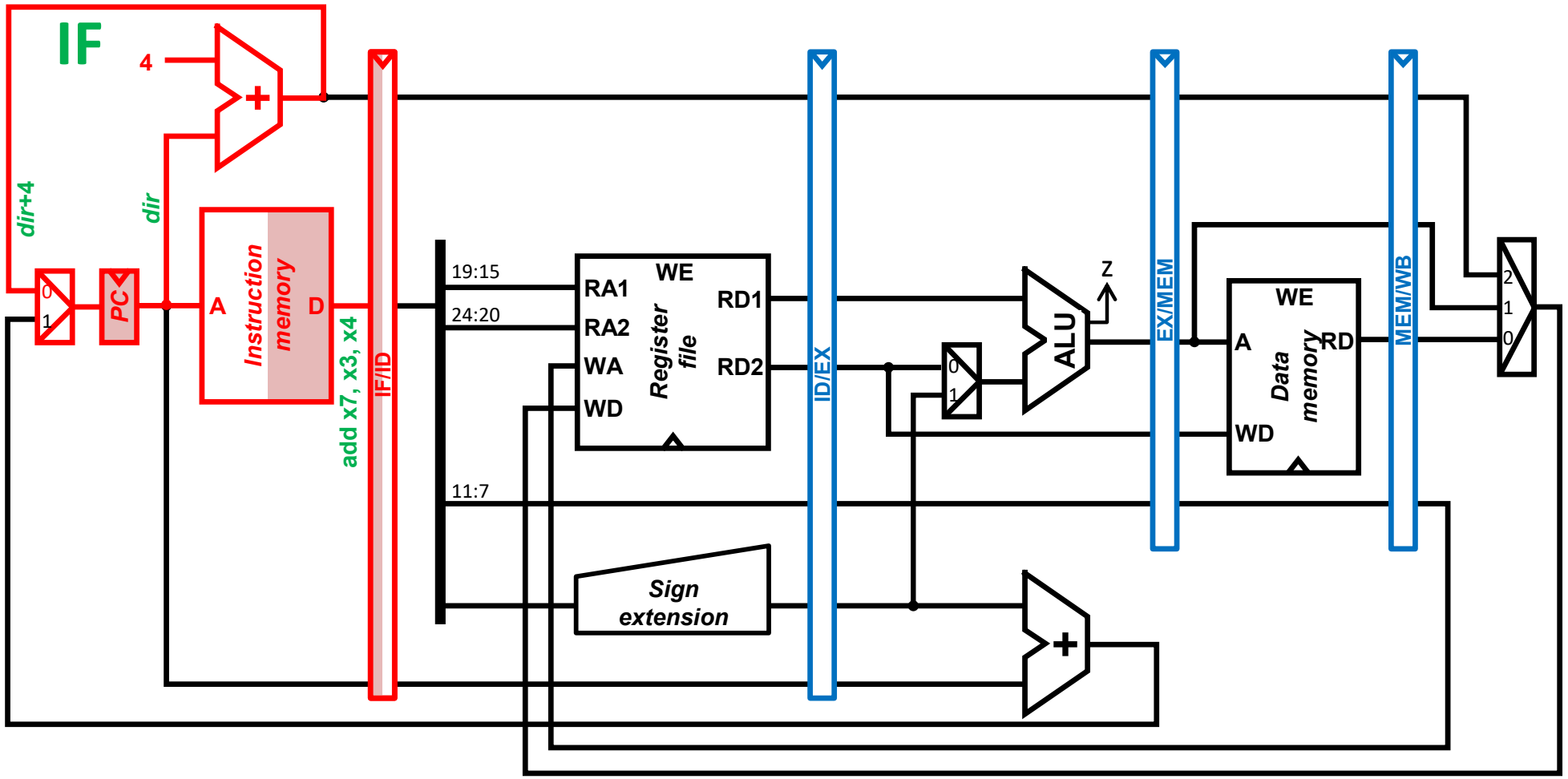


Data path design

add-like instruction: IF stage

The **add** *instruction takes 5 cycles without using the memory in the MEM stage*

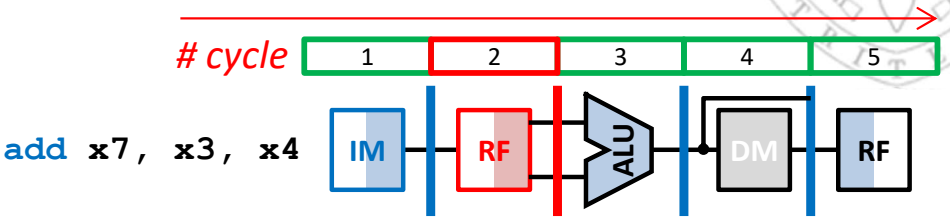


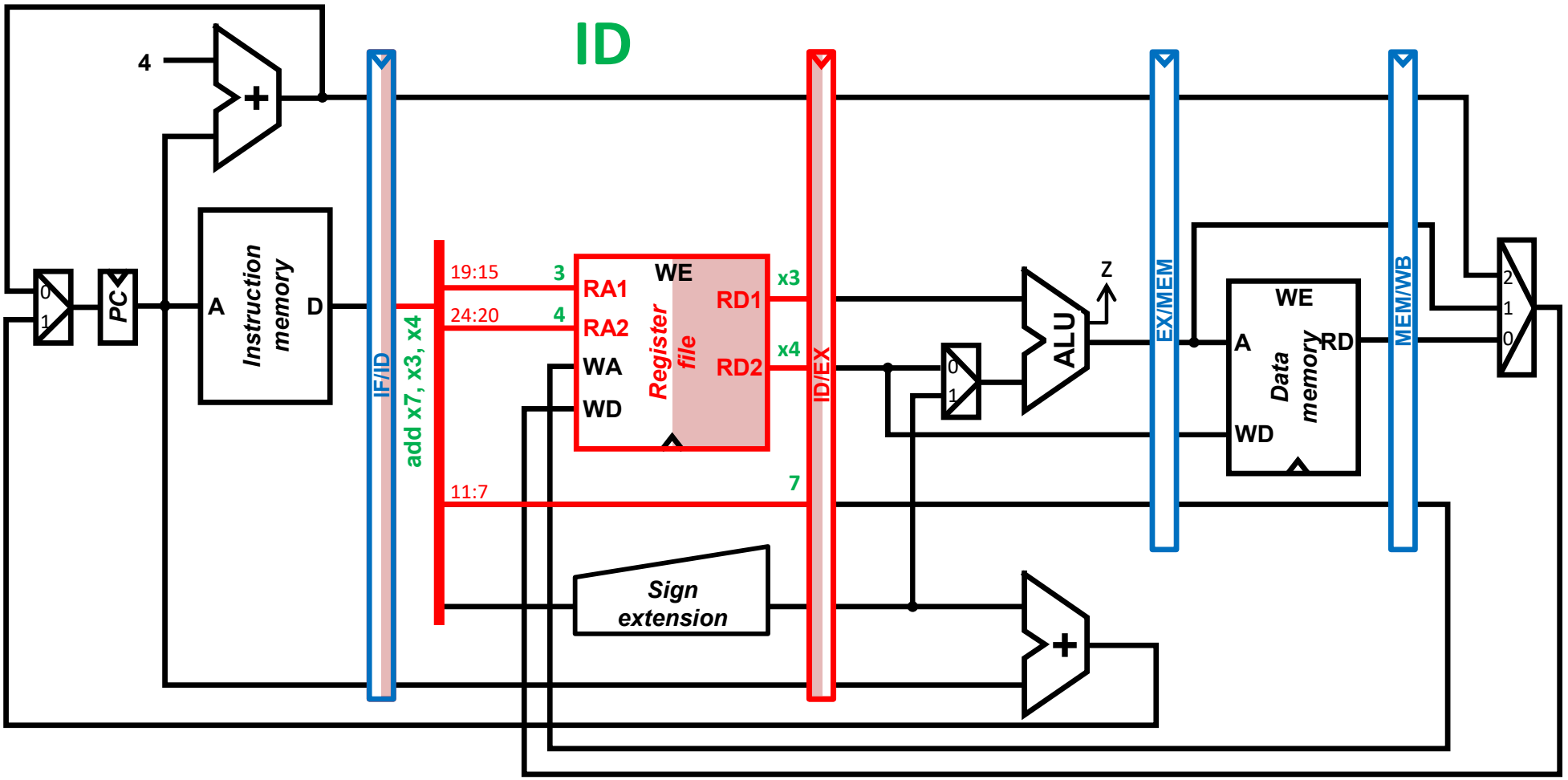


Data path design

add-like instruction: ID stage

The **add** *instruction takes 5 cycles without using the memory in the MEM stage*

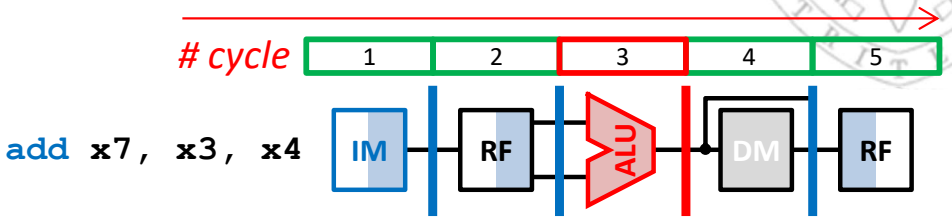


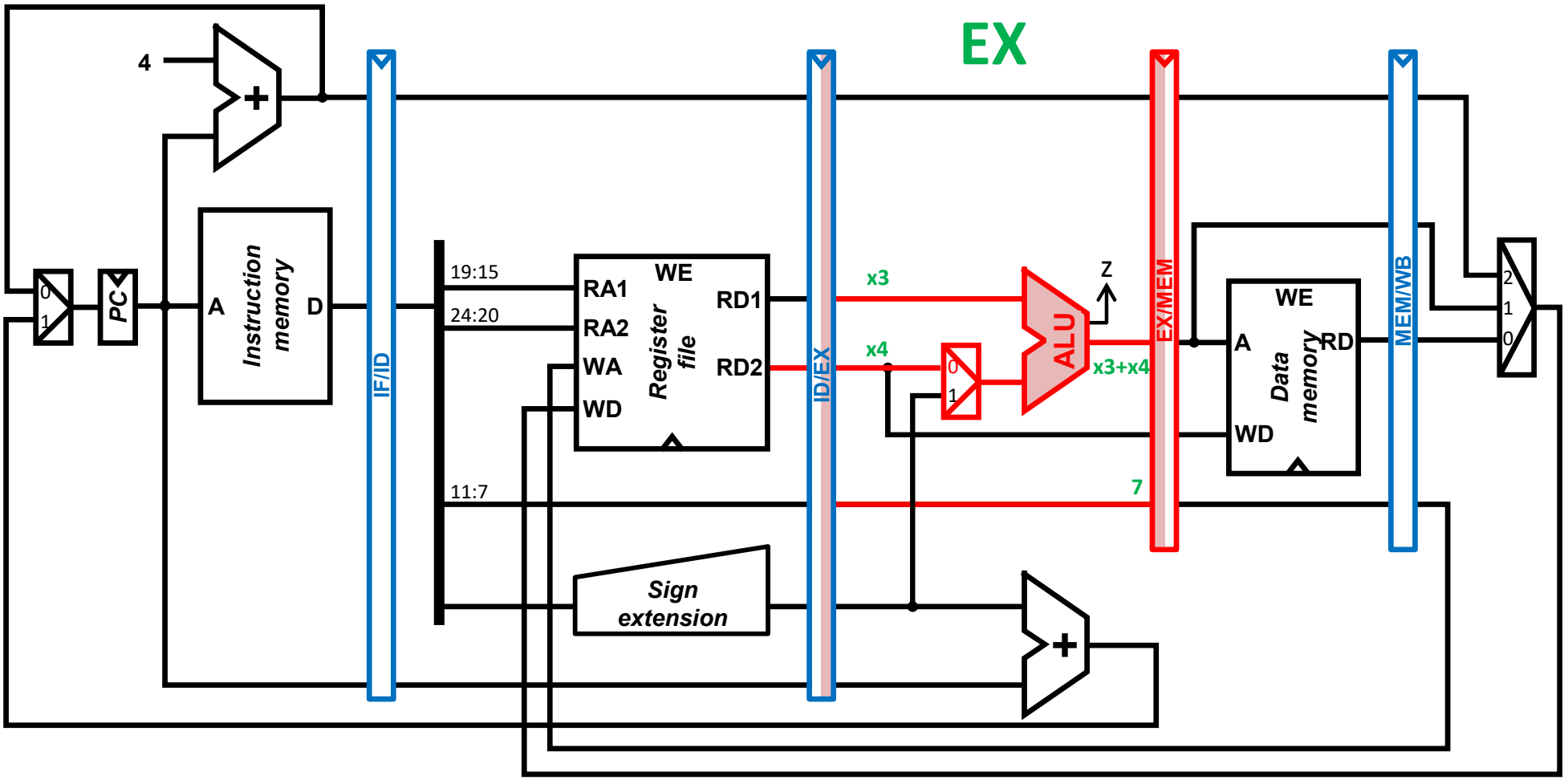


Data path design

add-like instruction: EX stage

The **add** *instruction takes 5 cycles without using the memory in the MEM stage*

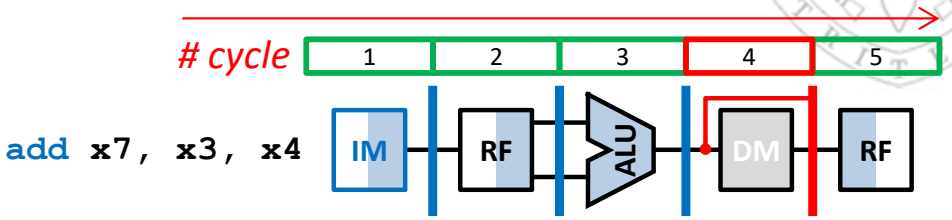


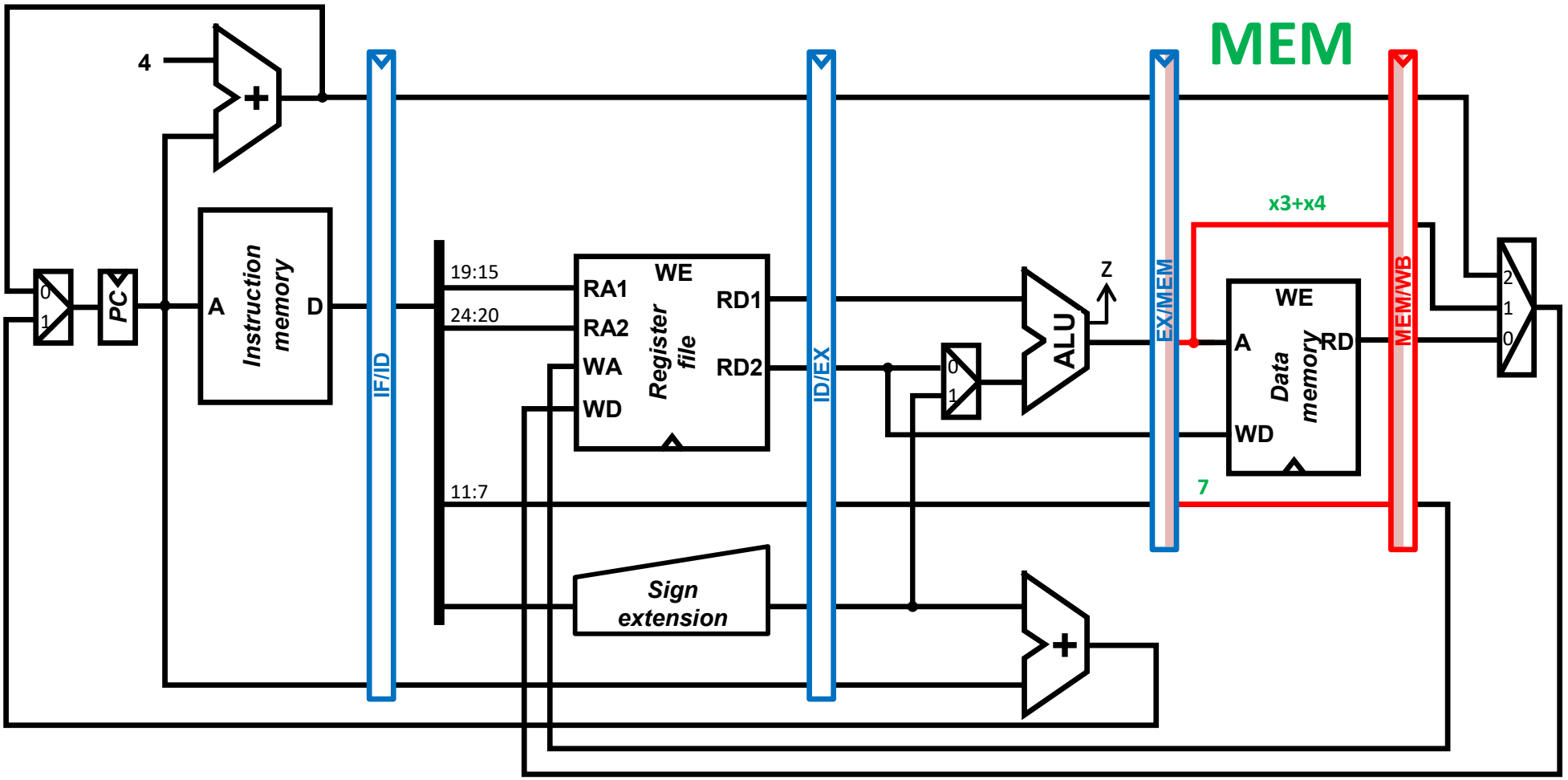


Data path design

add-like instruction: MEM stage

The **add** *instruction takes 5 cycles without using the memory in the MEM stage*

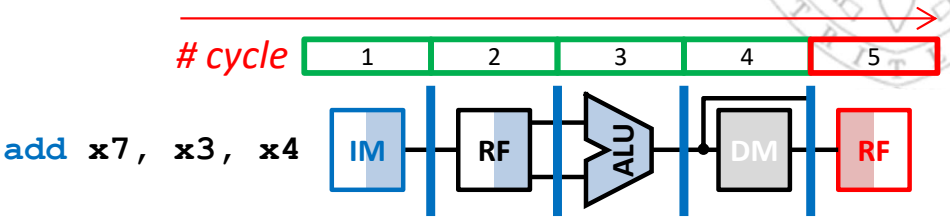


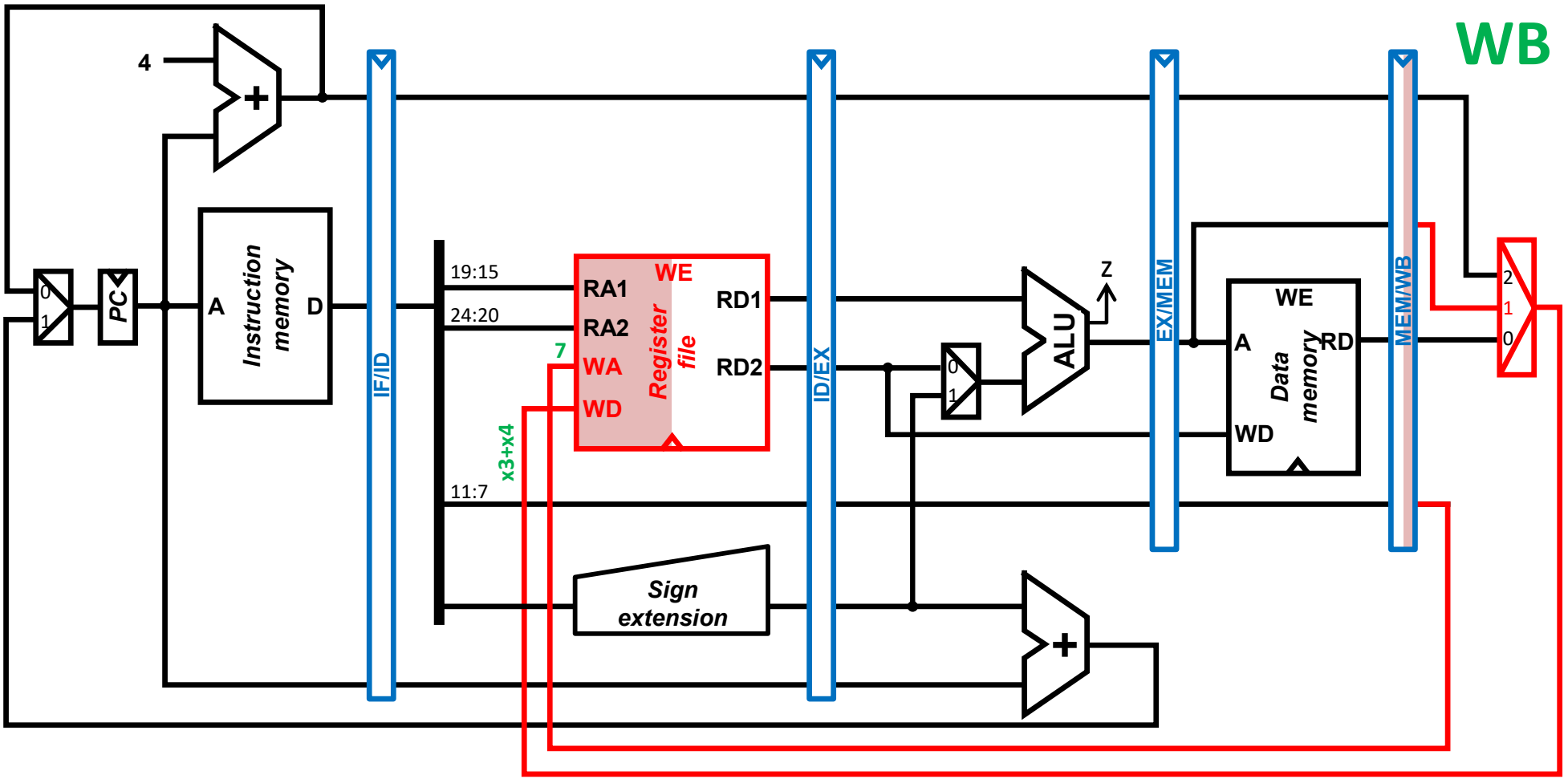


Data path design

add-like instruction: WB stage

The **add** *instruction takes 5 cycles without using the memory in the MEM stage*

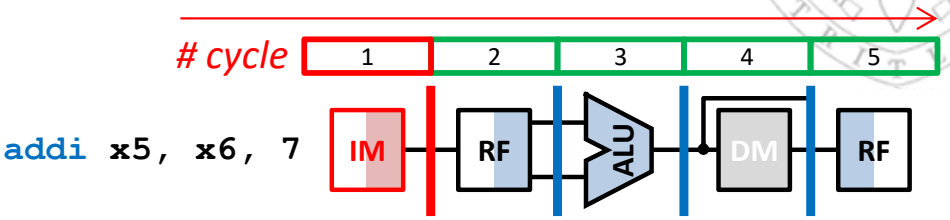


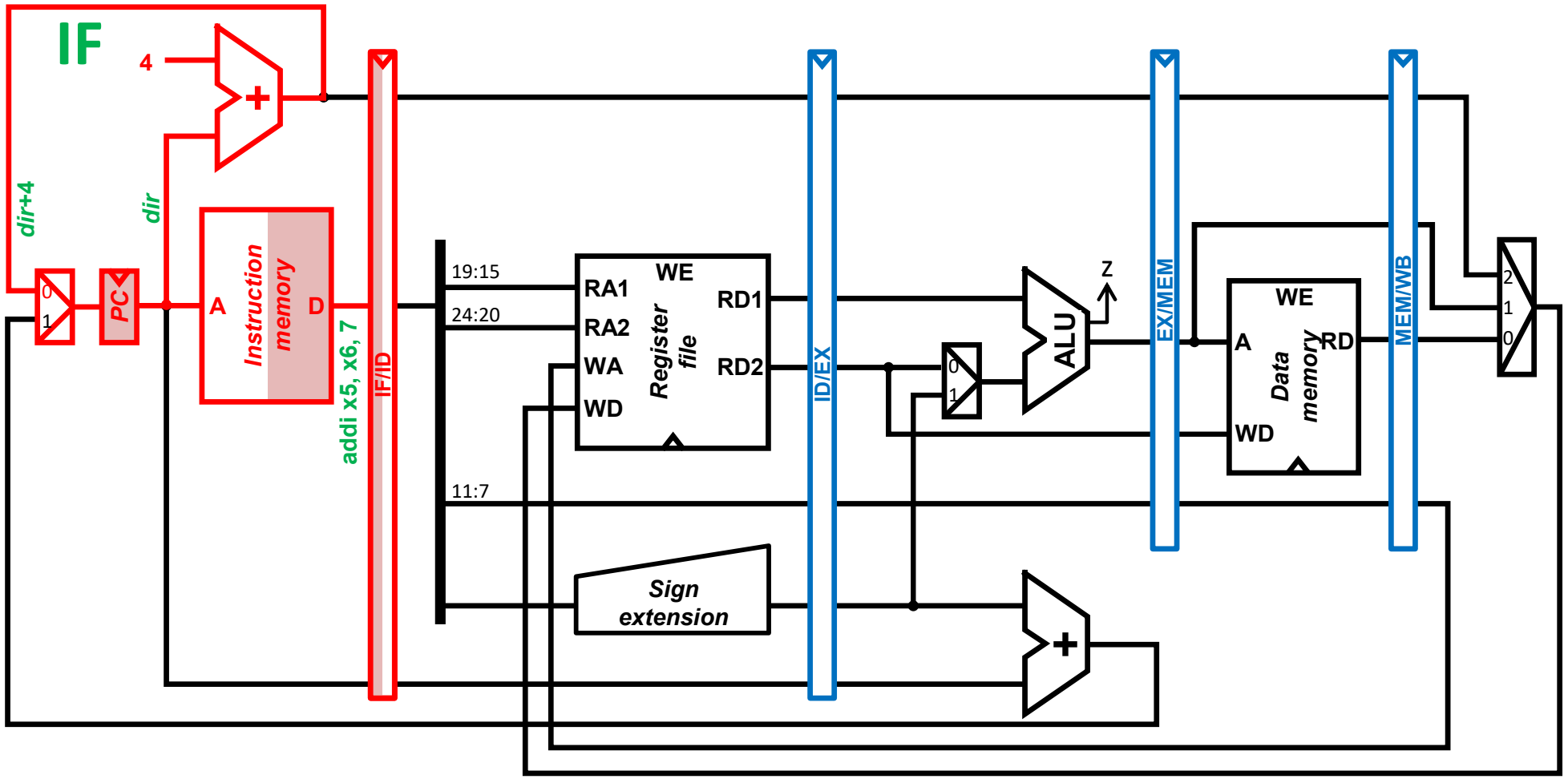


Data path design

`addi`-like instruction: IF stage

The **`addi`** *instruction takes 5 cycles without using the memory in the MEM stage*

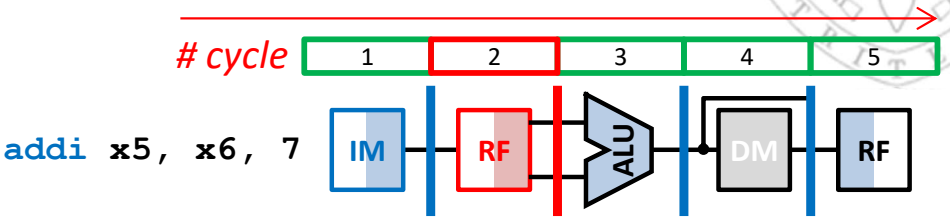


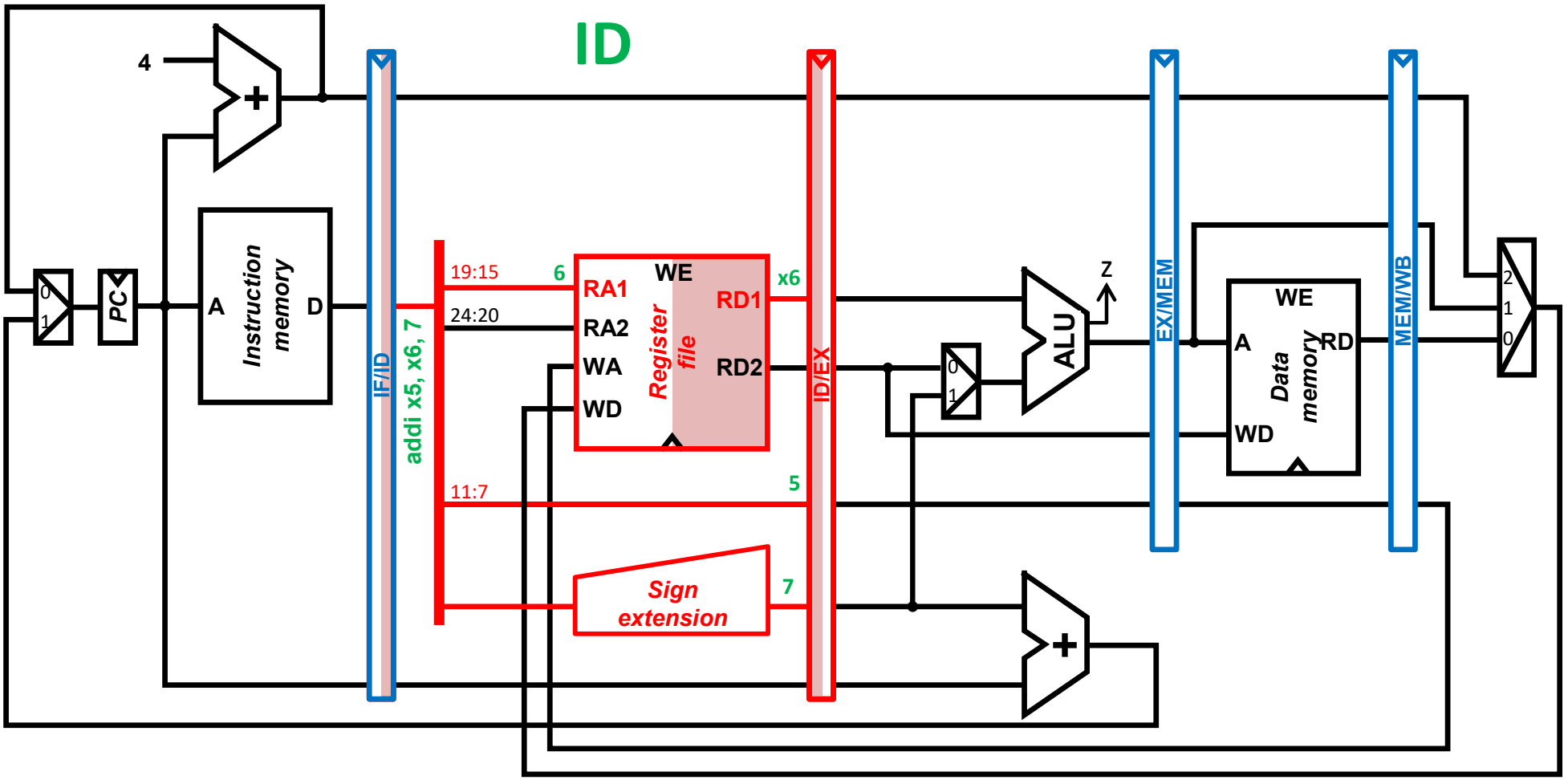


Data path design

`addi`-like instruction: ID stage

The **`addi`** *instruction takes 5 cycles without using the memory in the MEM stage*

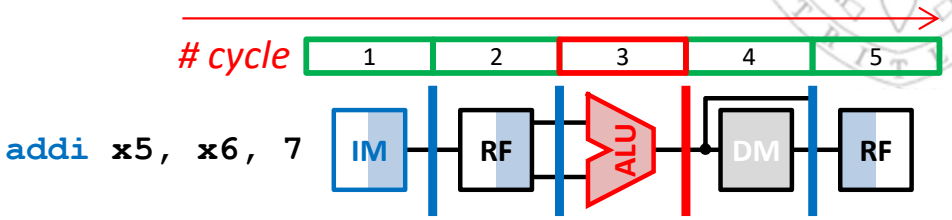


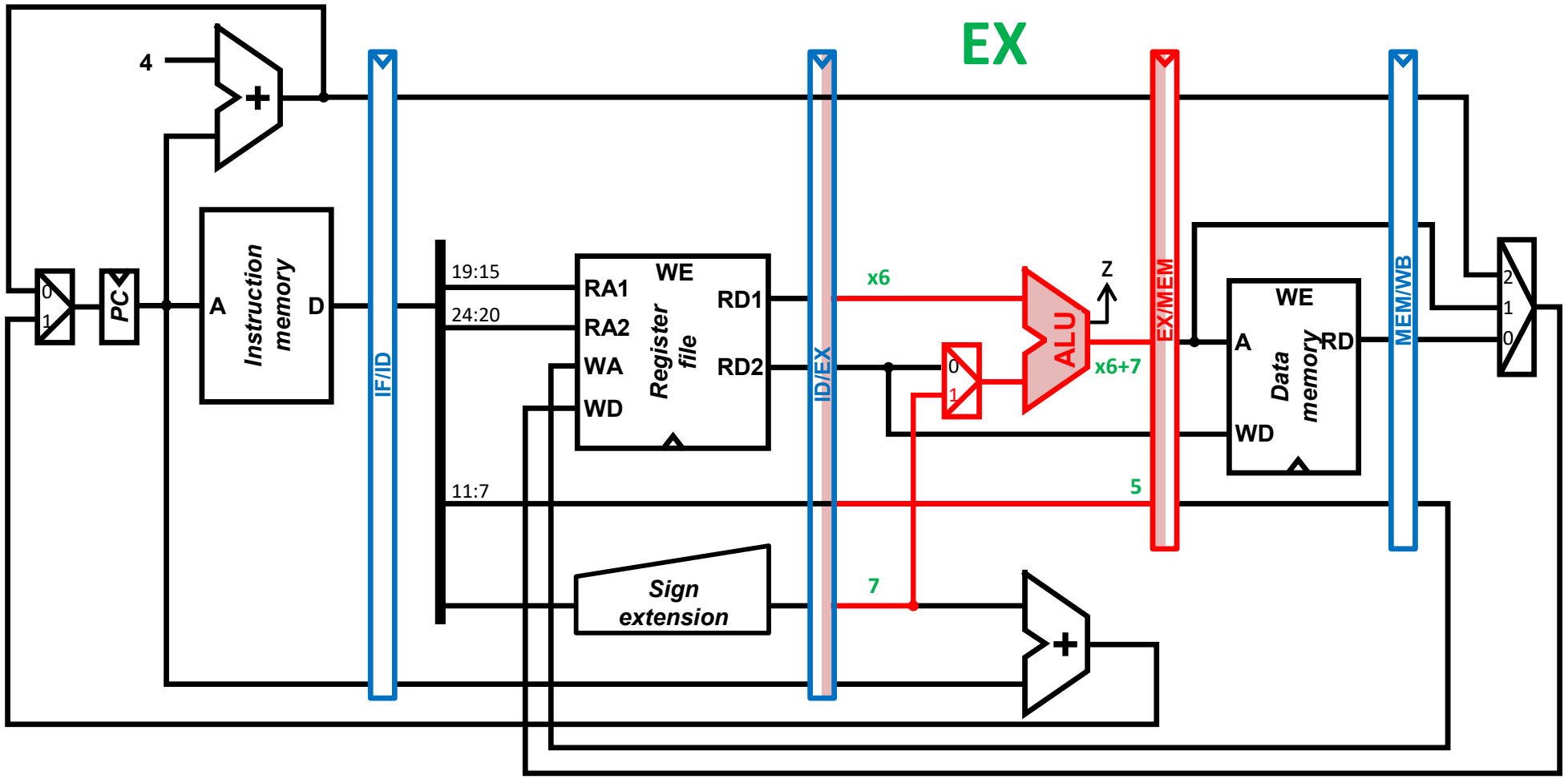


Data path design

`addi`-like instruction: EX stage

The **`addi`** *instruction takes 5 cycles without using the memory in the MEM stage*

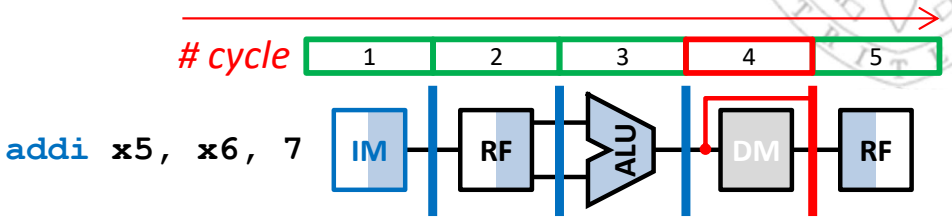


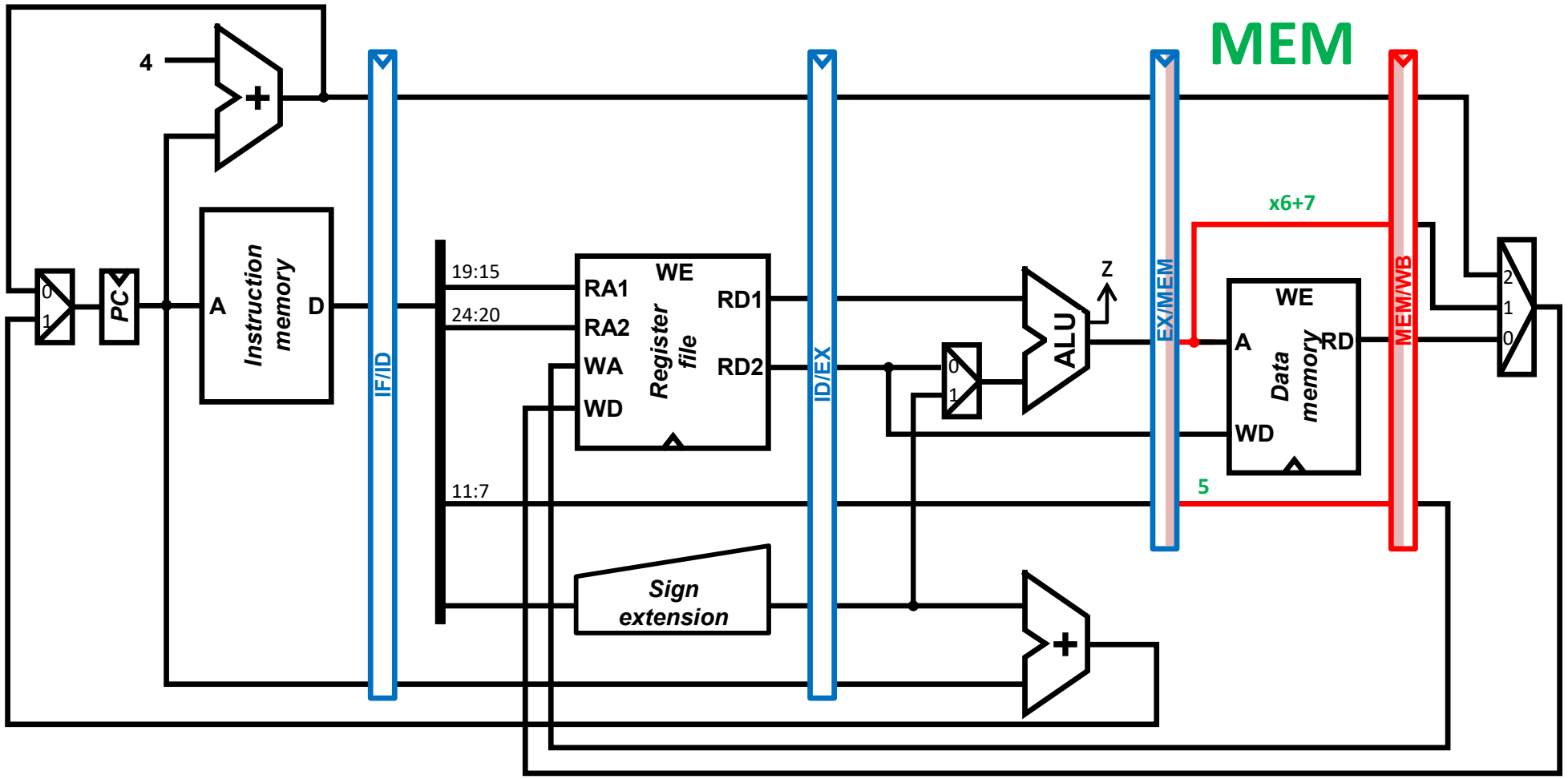


Data path design

addi-like instruction: MEM stage

The **addi** *instruction takes 5 cycles without using the memory in the MEM stage*

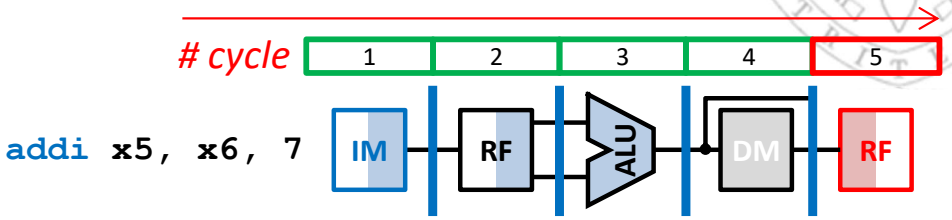


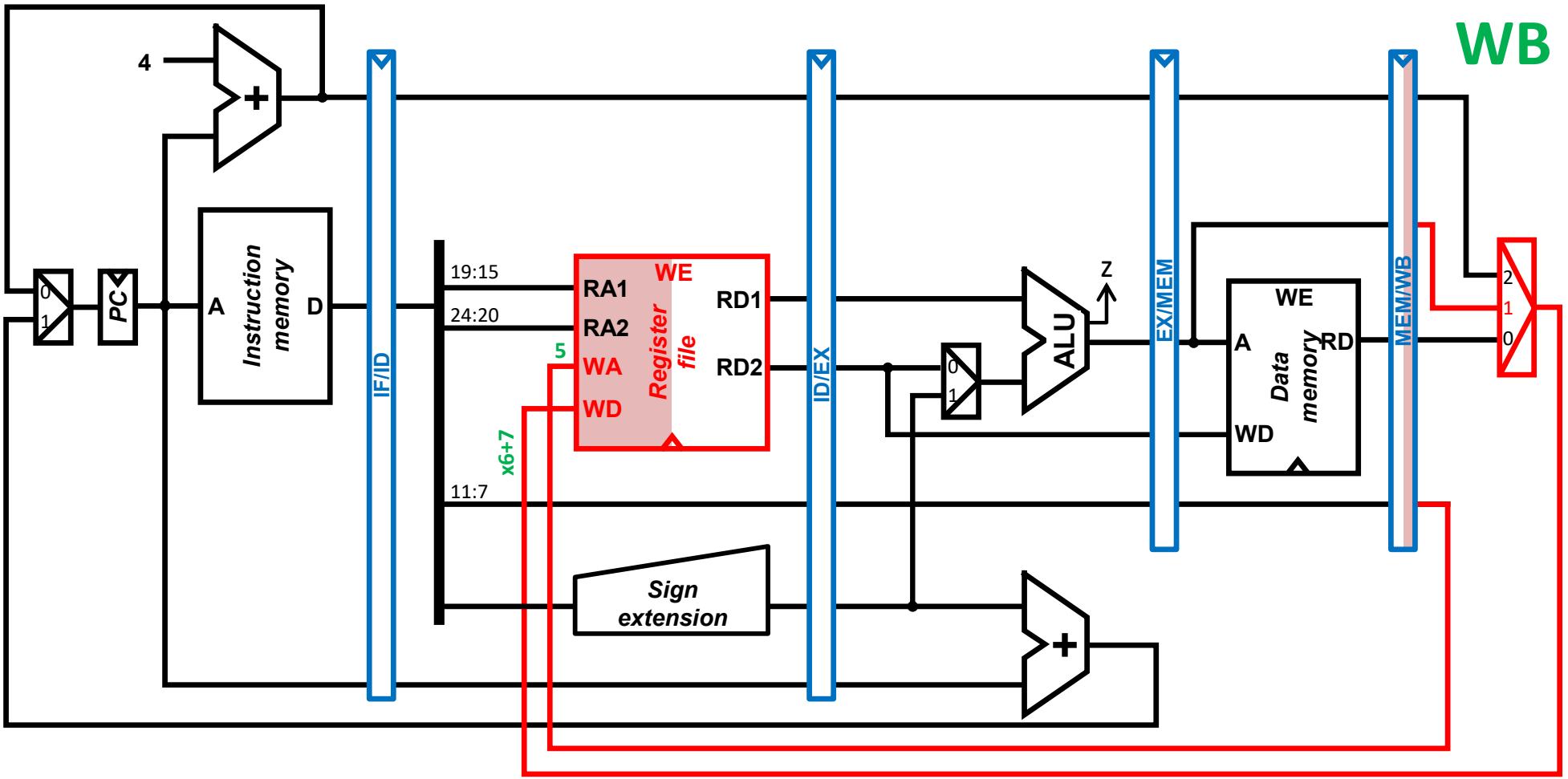


Data path design

`addi`-like instruction: WB stage

The **`addi`** *instruction takes 5 cycles without using the memory in the MEM stage*

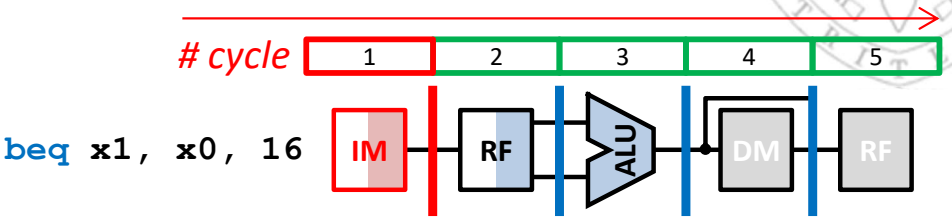


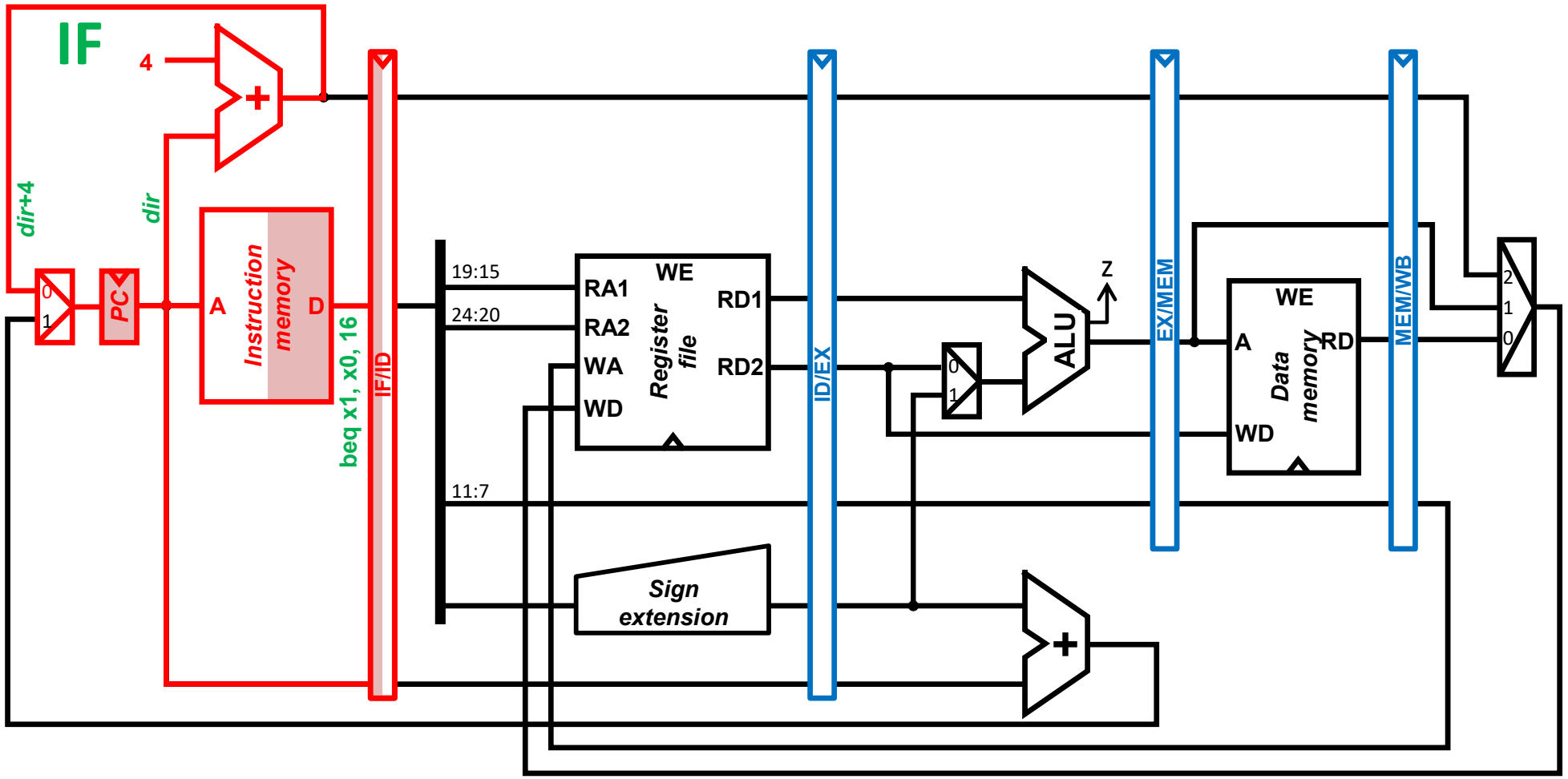


Data path design

`beq` instruction: IF stage

The **`beq`** *instruction takes 5 cycles without using the memory in the MEM stage or the RF in the WB stage*

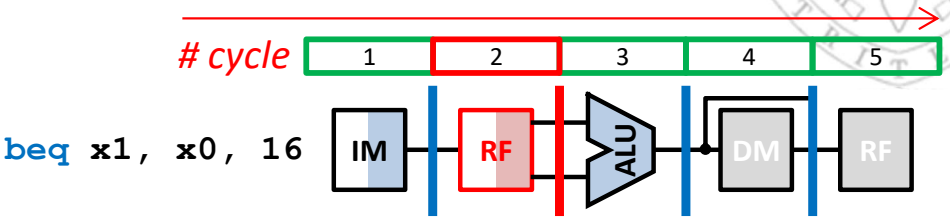


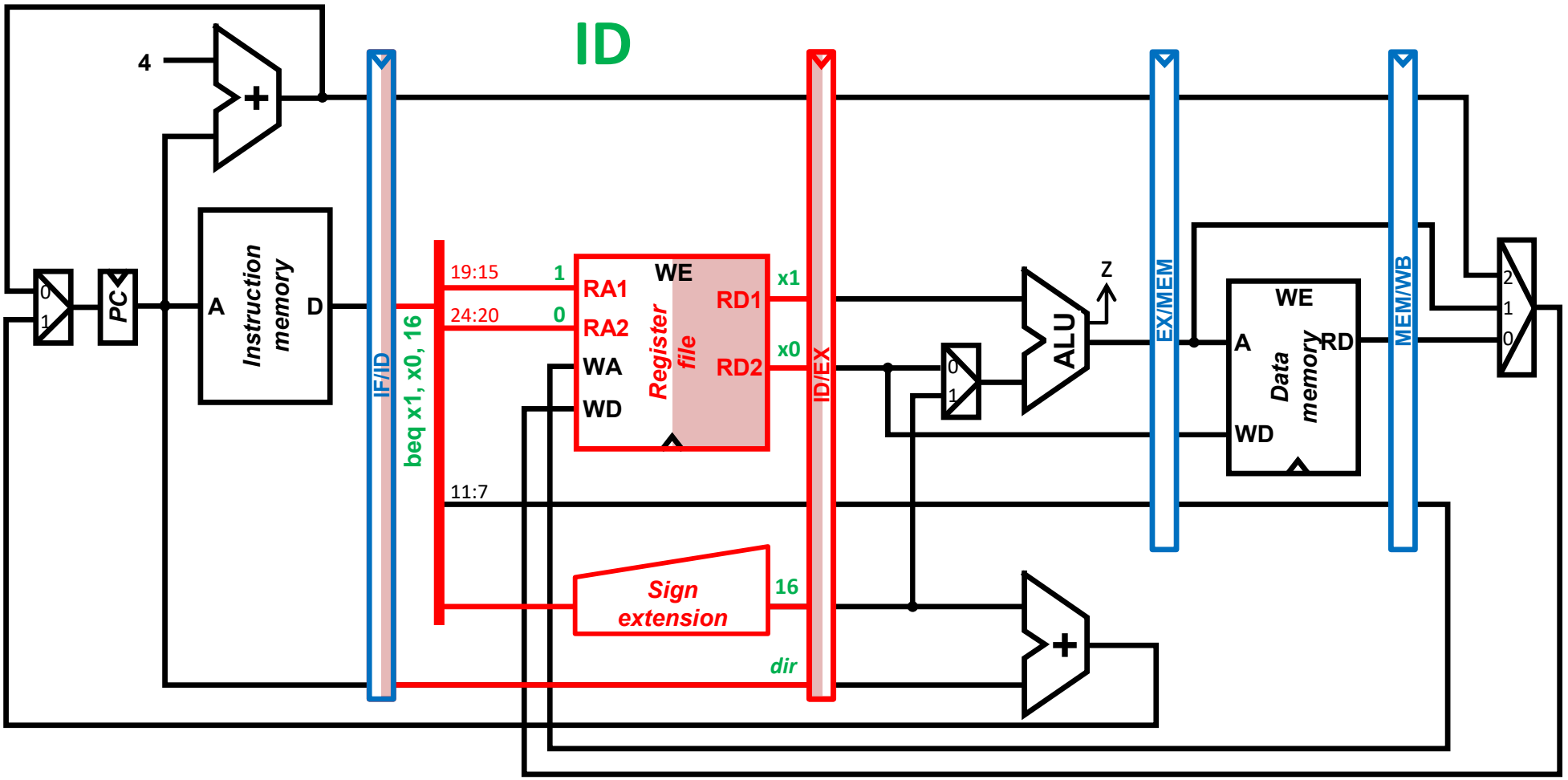


Data path design

`beq` instruction: ID stage

The `beq` *instruction takes 5 cycles without using the memory in the MEM stage or the RF in the WB stage*

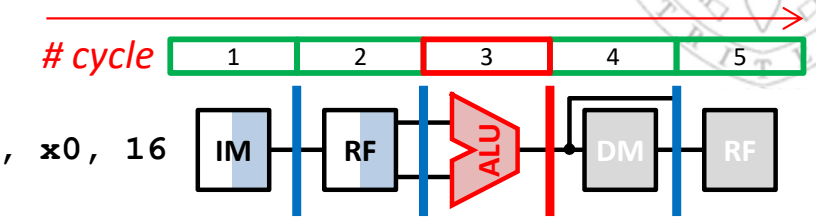


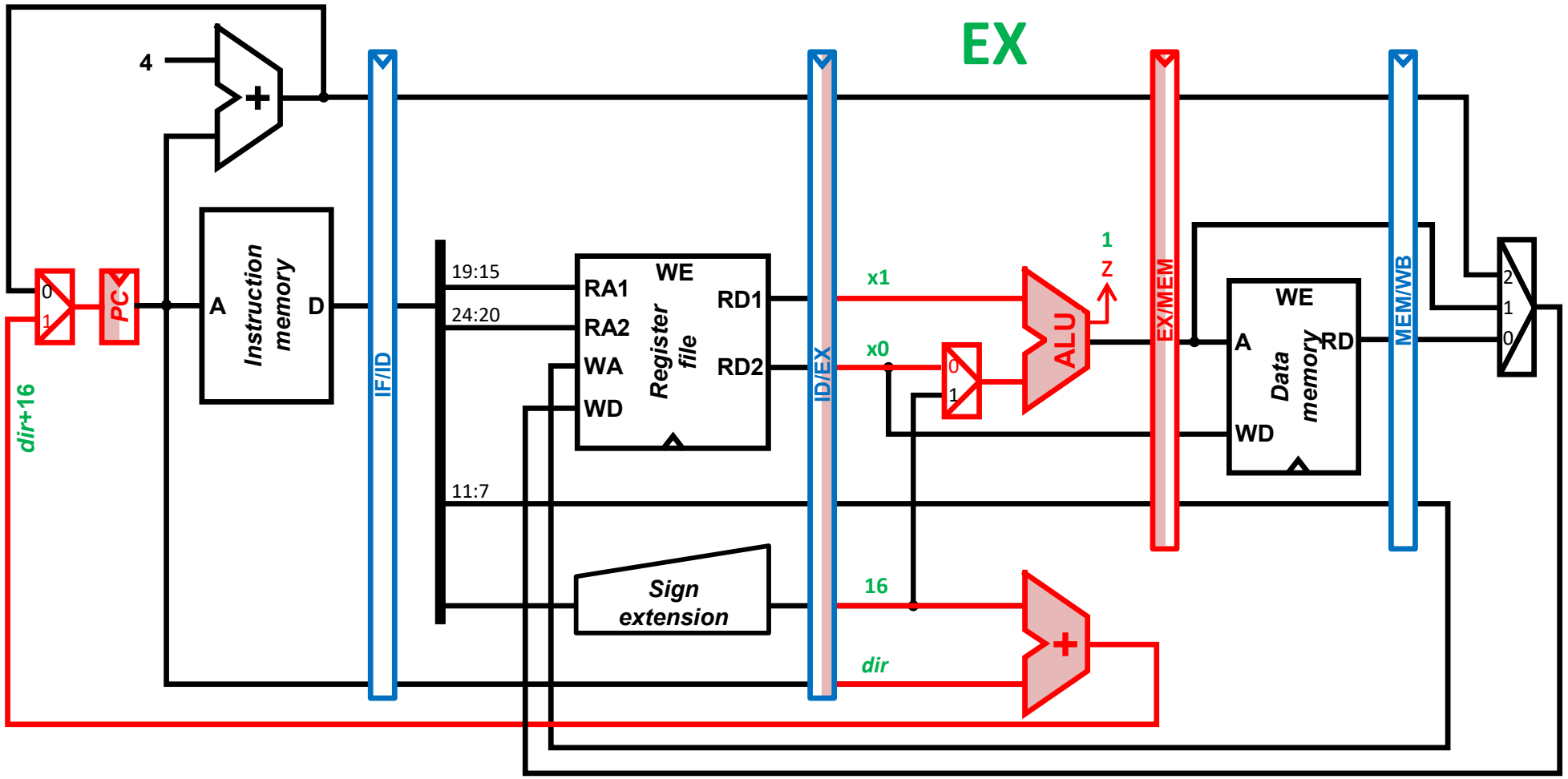


Data path design

beq instruction: EX stage

The **beq** *instruction takes 5 cycles without using the memory in the MEM stage or the RF in the WB stage*

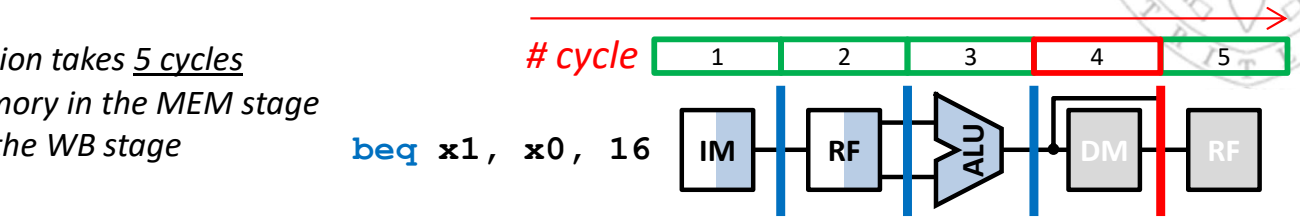


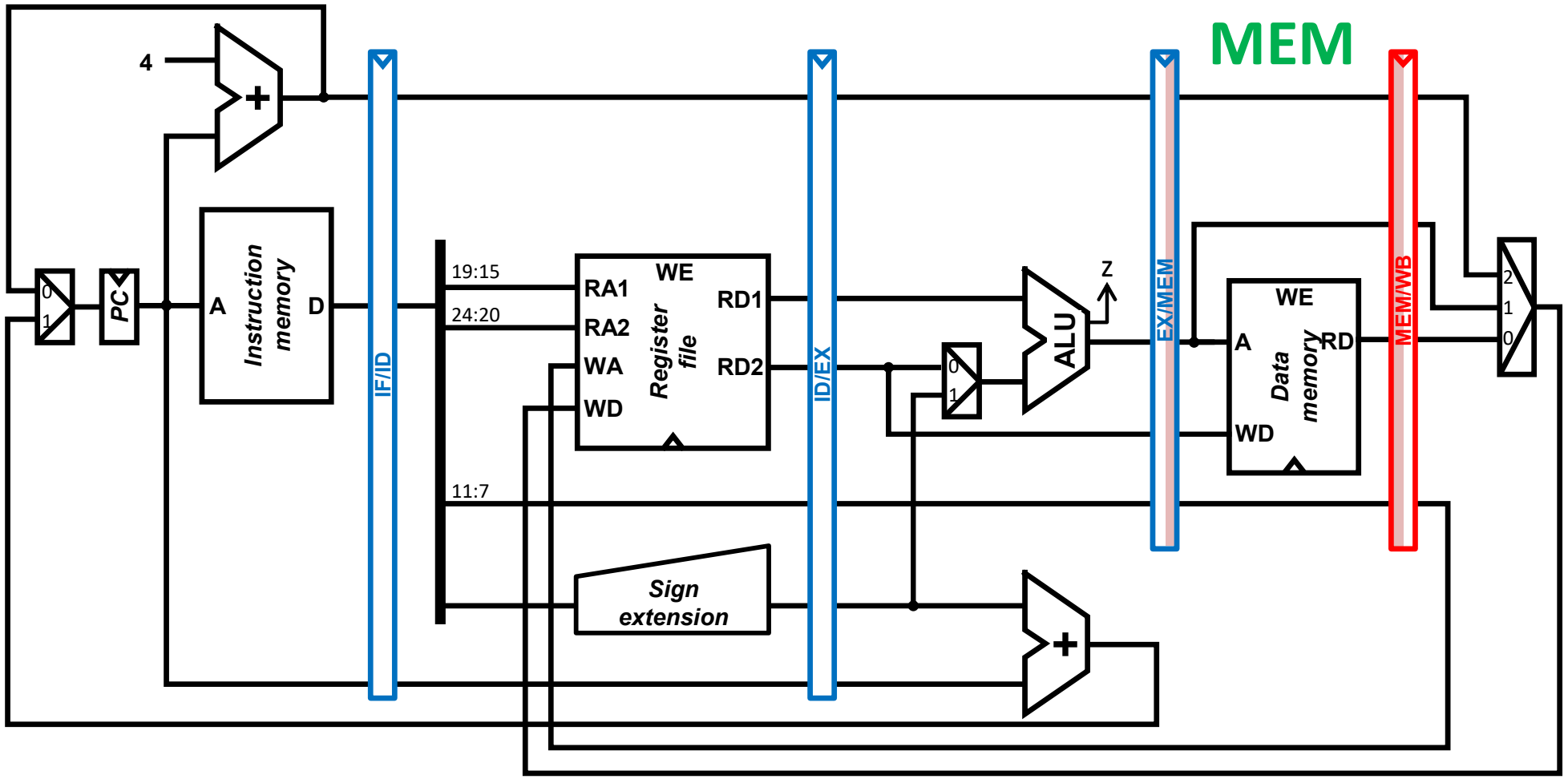


Data path design

`beq` instruction: MEM stage

The **`beq`** *instruction takes 5 cycles without using the memory in the MEM stage or the RF in the WB stage*

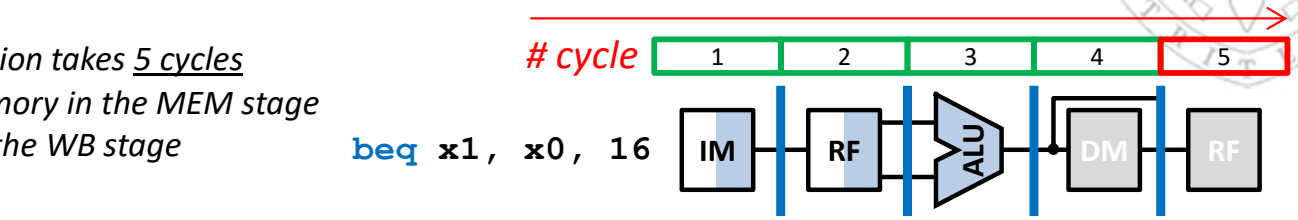


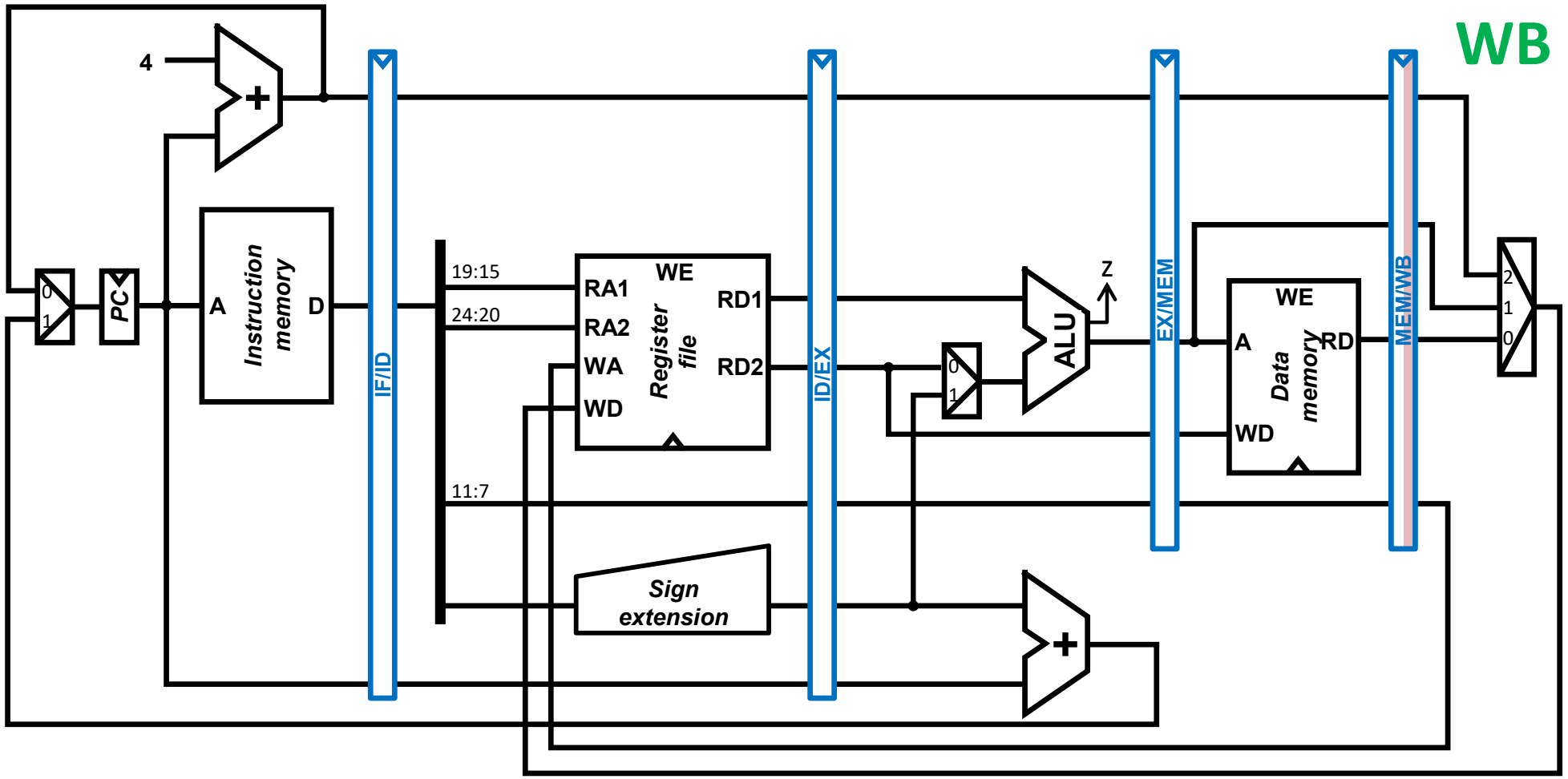


Data path design

`beq` instruction: WB stage

The **`beq`** *instruction takes 5 cycles without using the memory in the MEM stage or the RF in the WB stage*

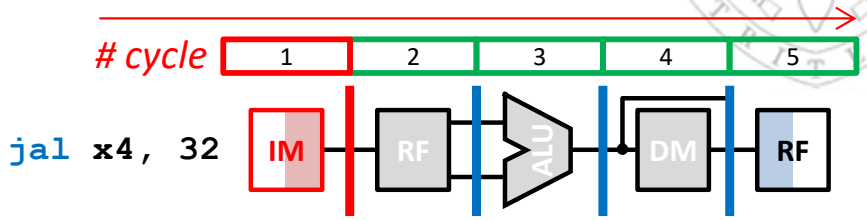


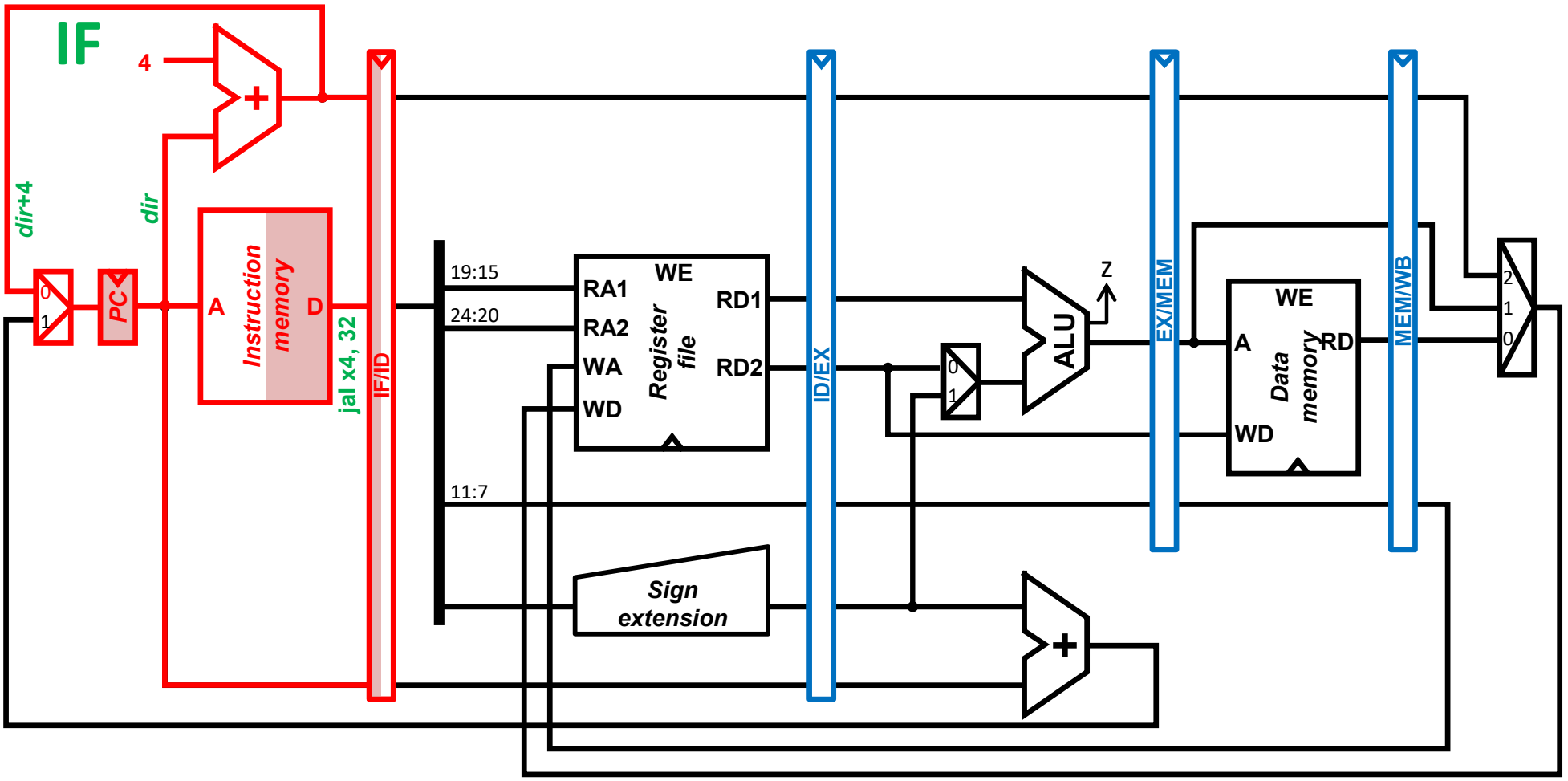


Data path design

`jal` instruction: IF stage

The `jal` *instruction takes 5 cycles without using the RF in the ID stage, the ALU in the EX stage or the memory in the MEM stage*

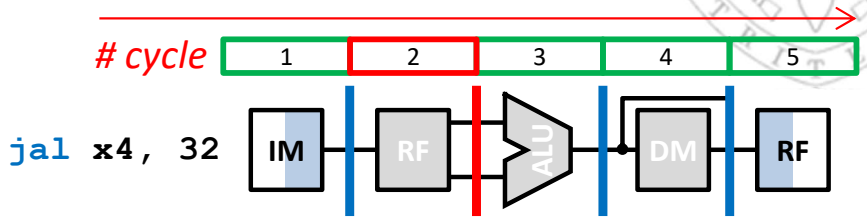


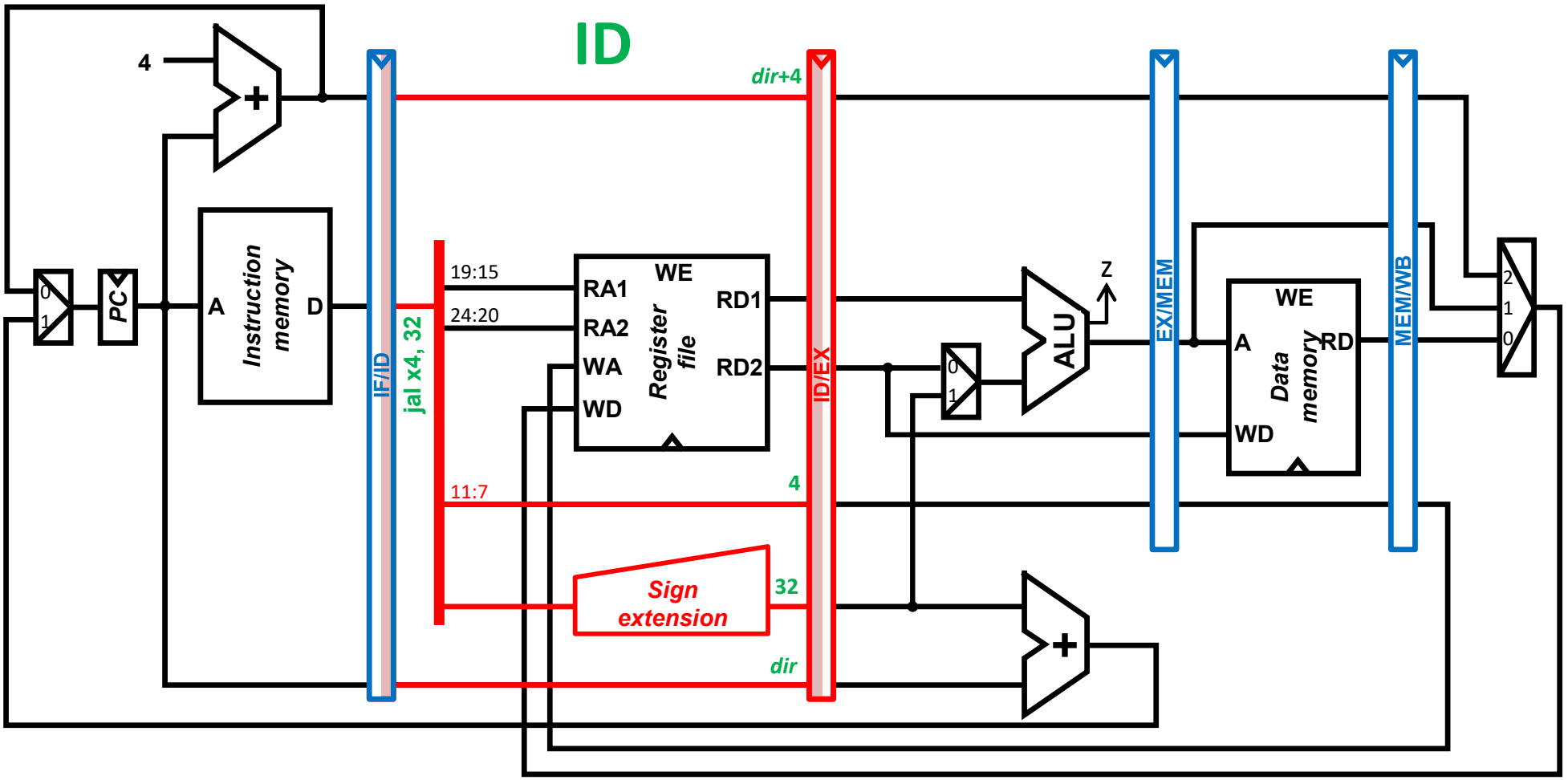


Data path design

`jal` instruction: ID stage

The `jal` *instruction takes 5 cycles without using the RF in the ID stage, the ALU in the EX stage or the memory in the MEM stage*

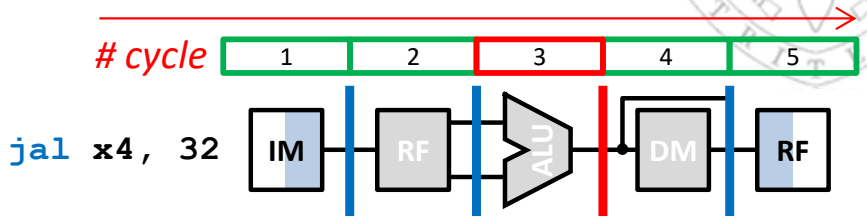


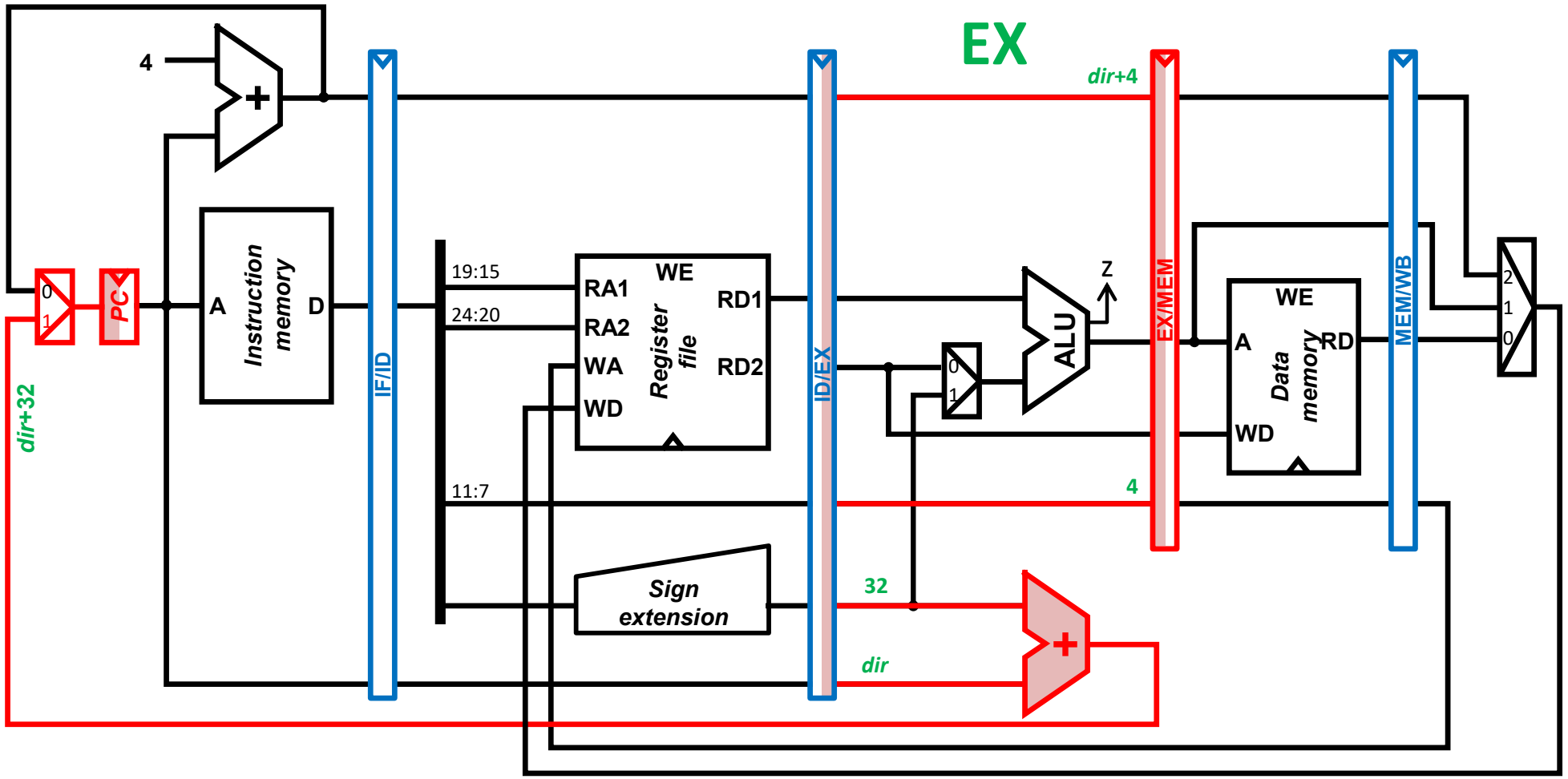


Data path design

`jal` instruction: EX stage

The `jal` *instruction takes 5 cycles without using the RF in the ID stage, the ALU in the EX stage or the memory in the MEM stage*

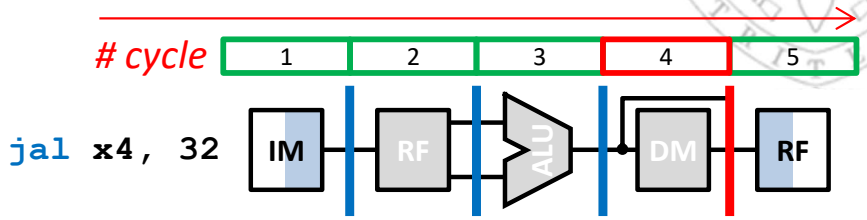


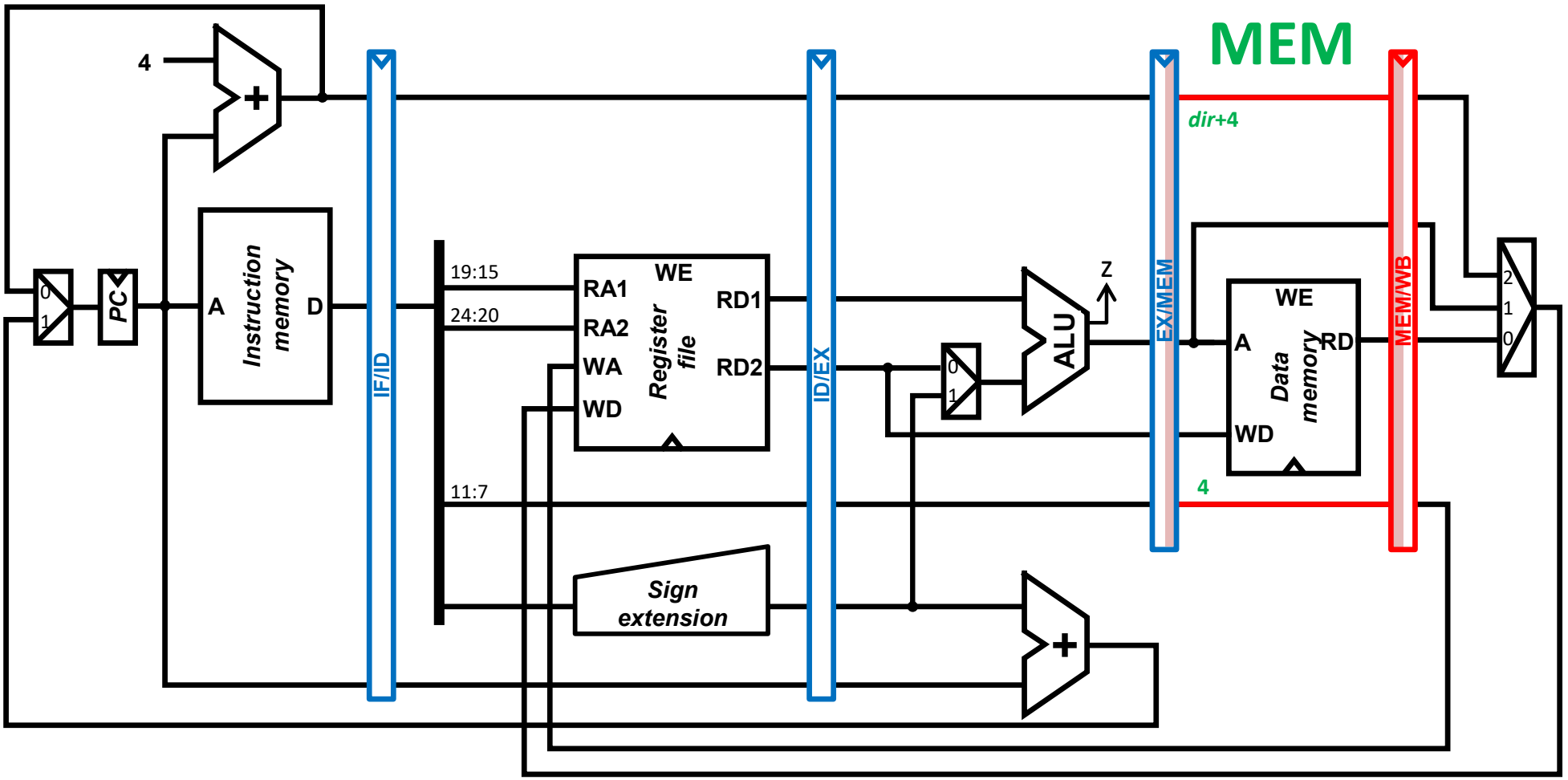


Data path design

`jal` instruction: MEM stage

The `jal` *instruction takes 5 cycles without using the RF in the ID stage, the ALU in the EX stage or the memory in the MEM stage*

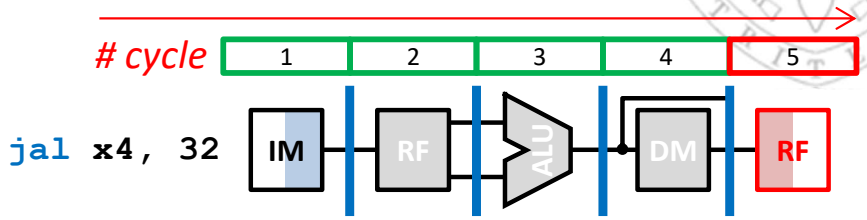


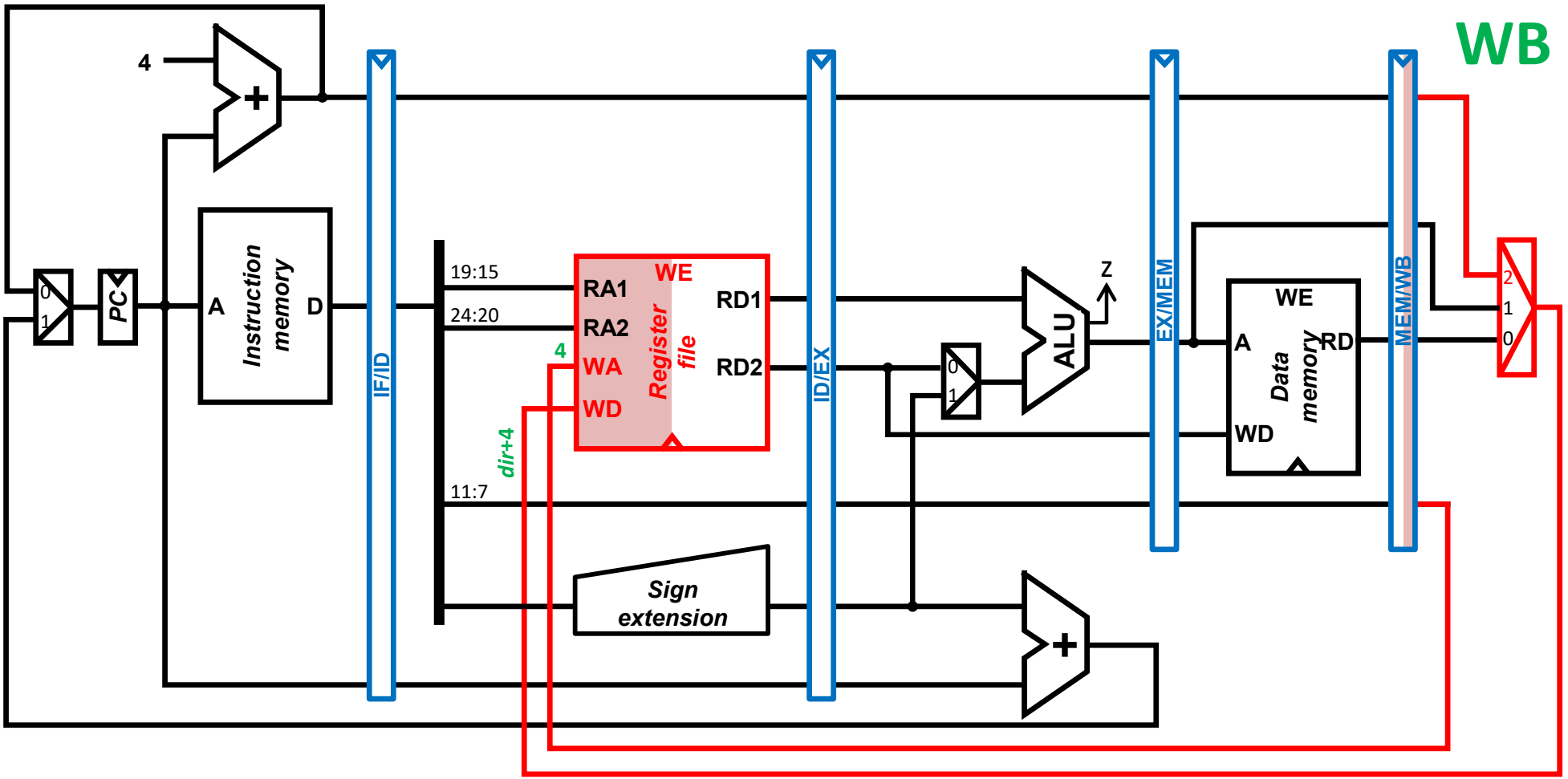


Data path design

`jal` instruction: WB stage

The `jal` *instruction takes 5 cycles without using the RF in the ID stage, the ALU in the EX stage or the memory in the MEM stage*

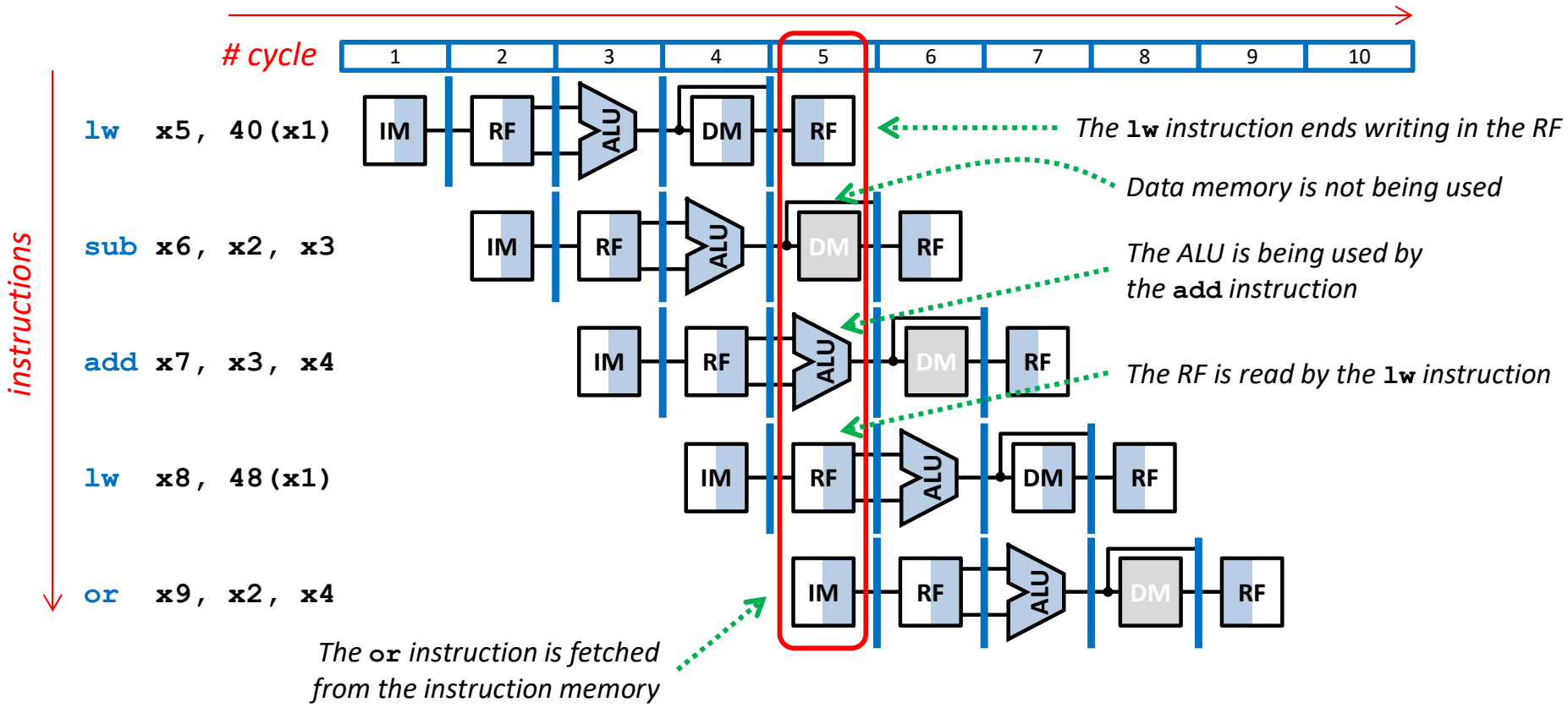




Data path design

Execution diagrams (i)

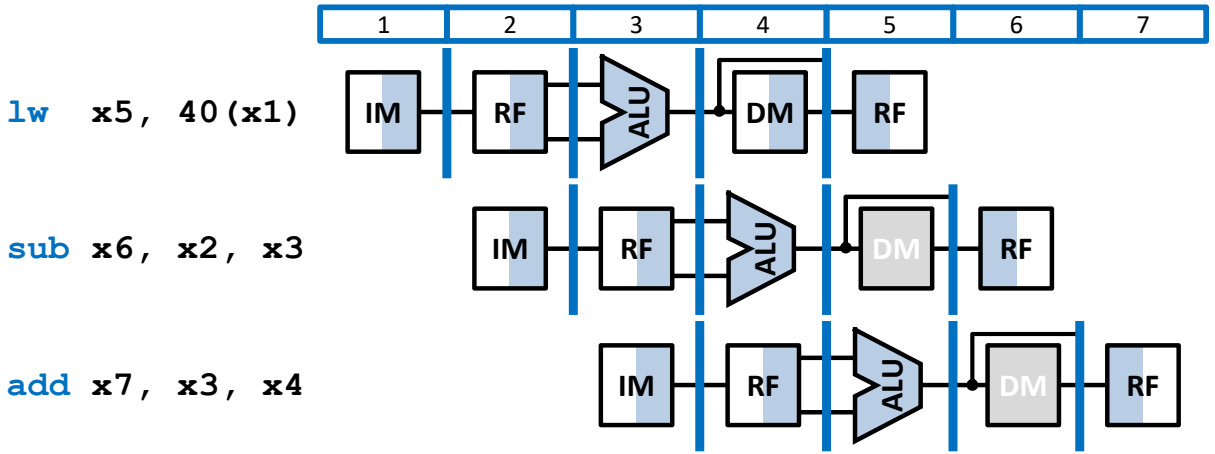
- An execution diagram allows visualizing the execution of a program in the pipeline:
  - For a given cycle, it visualizes the instructions in execution, in which pipeline stage each of them is and the resources that are used.



Data path design

Execution diagrams (ii)

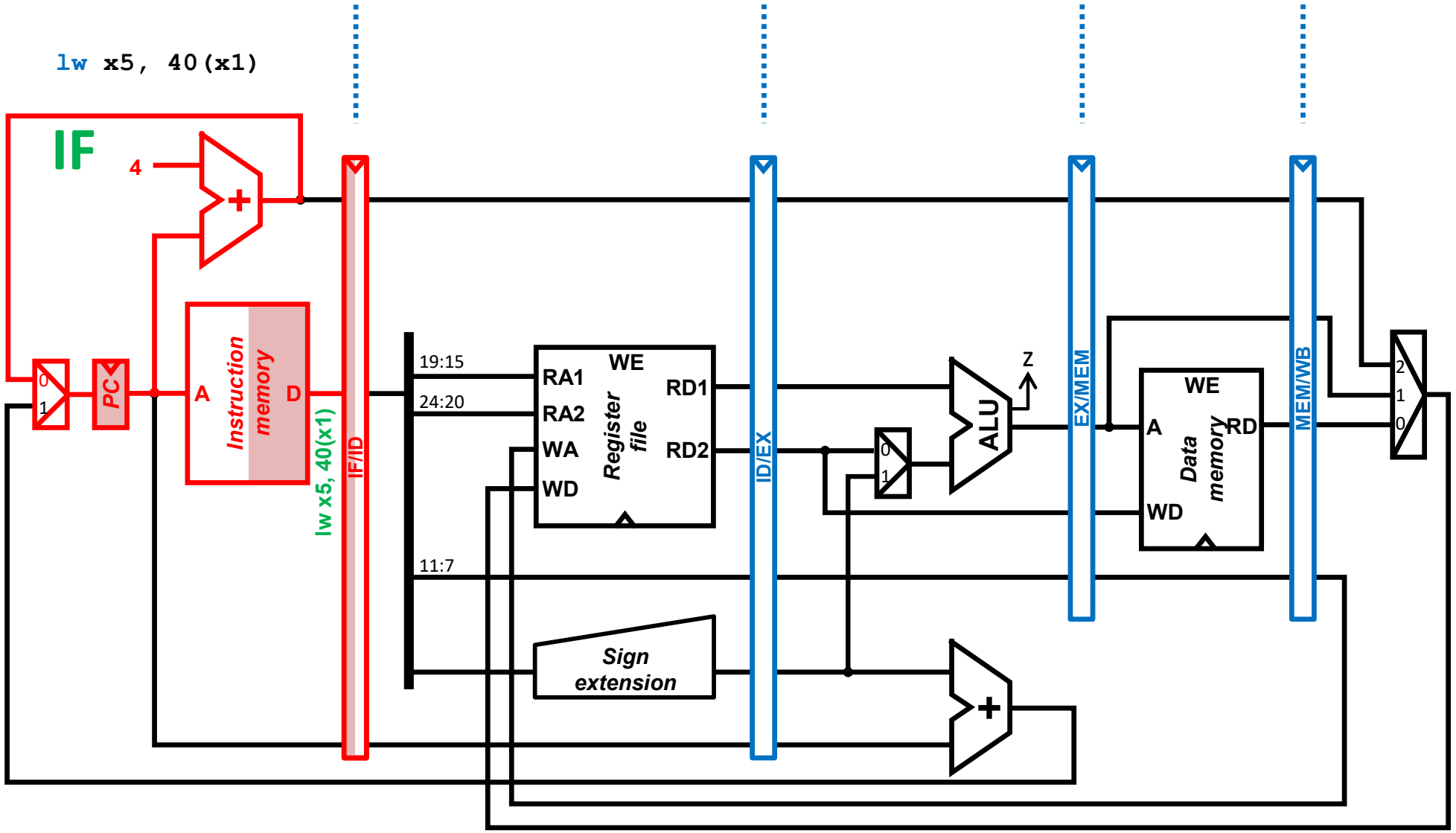
- Alternatively, it is common to use simplified execution diagrams that show, in each cycle, the stage in which each instruction is.



	1	2	3	4	5	6	7
`lw x5, 40(x1)`	IF	ID	EX	M	WB		
`sub x6, x2, x3`		IF	ID	EX	M	WB	
`add x7, x3, x4`			IF	ID	EX	M	WB

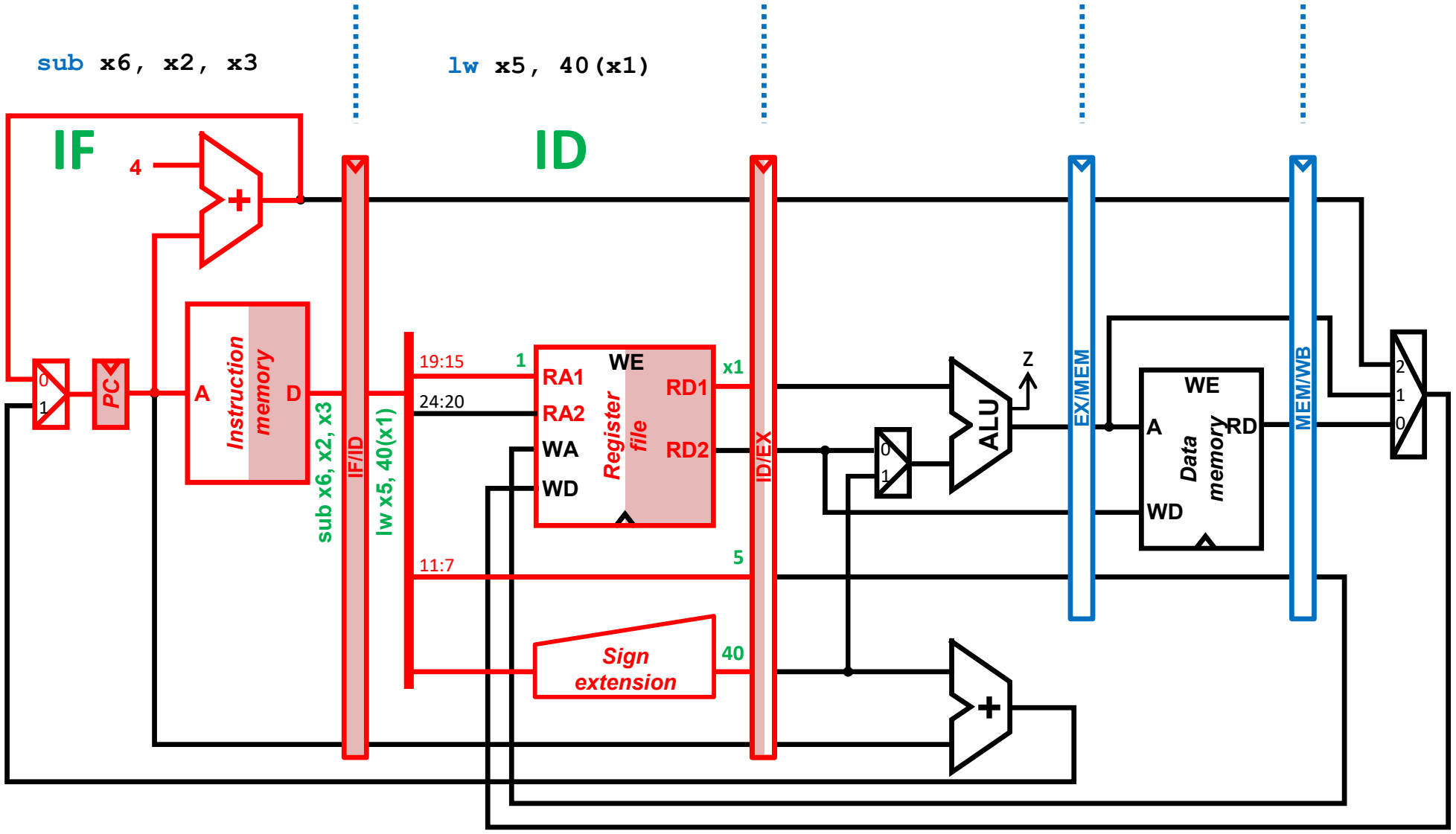
Data path design

Simulation: 1st. cycle



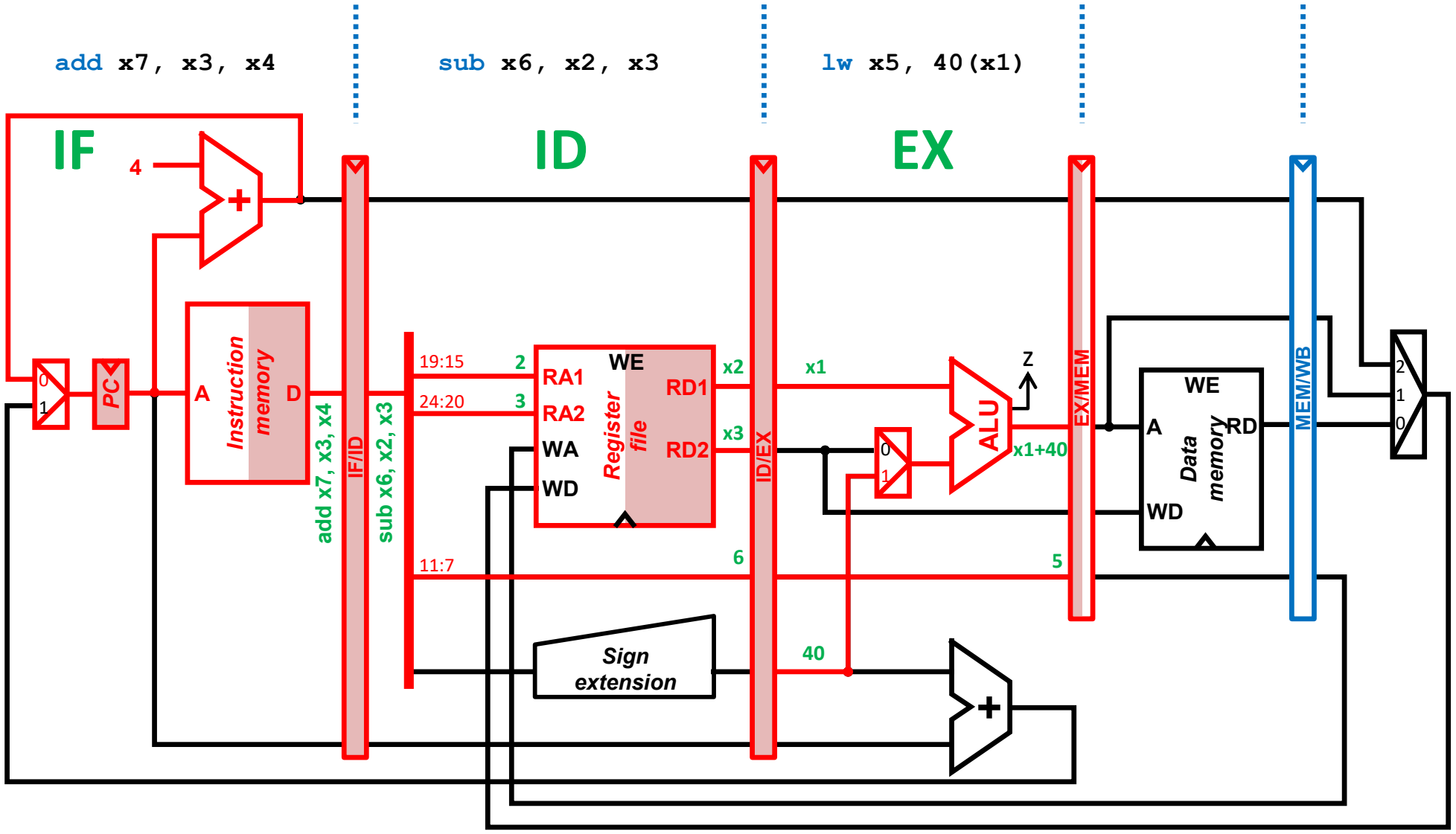
Data path design

Simulation: 2nd. cycle



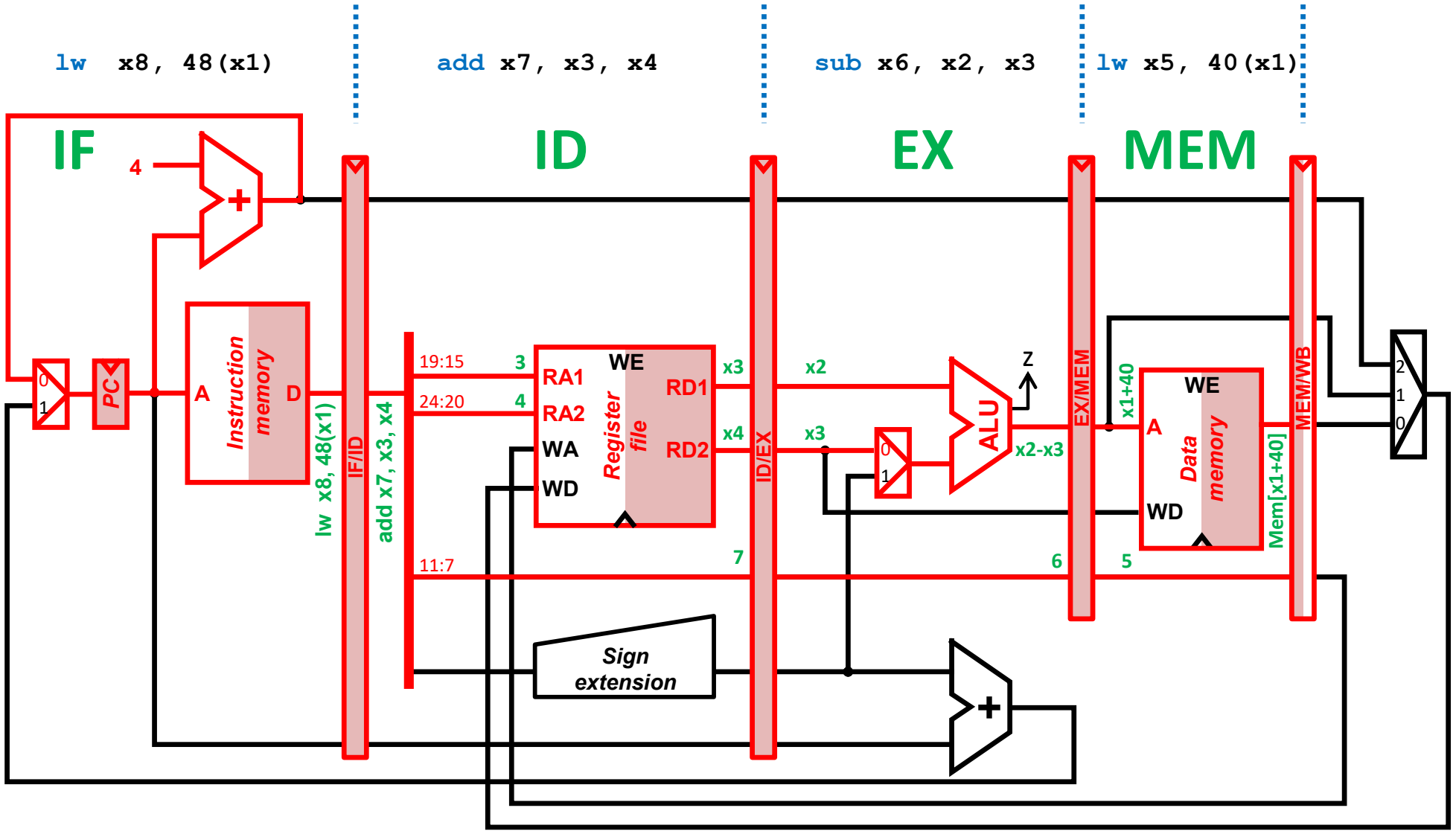
Data path design

Simulation: 3rd. cycle



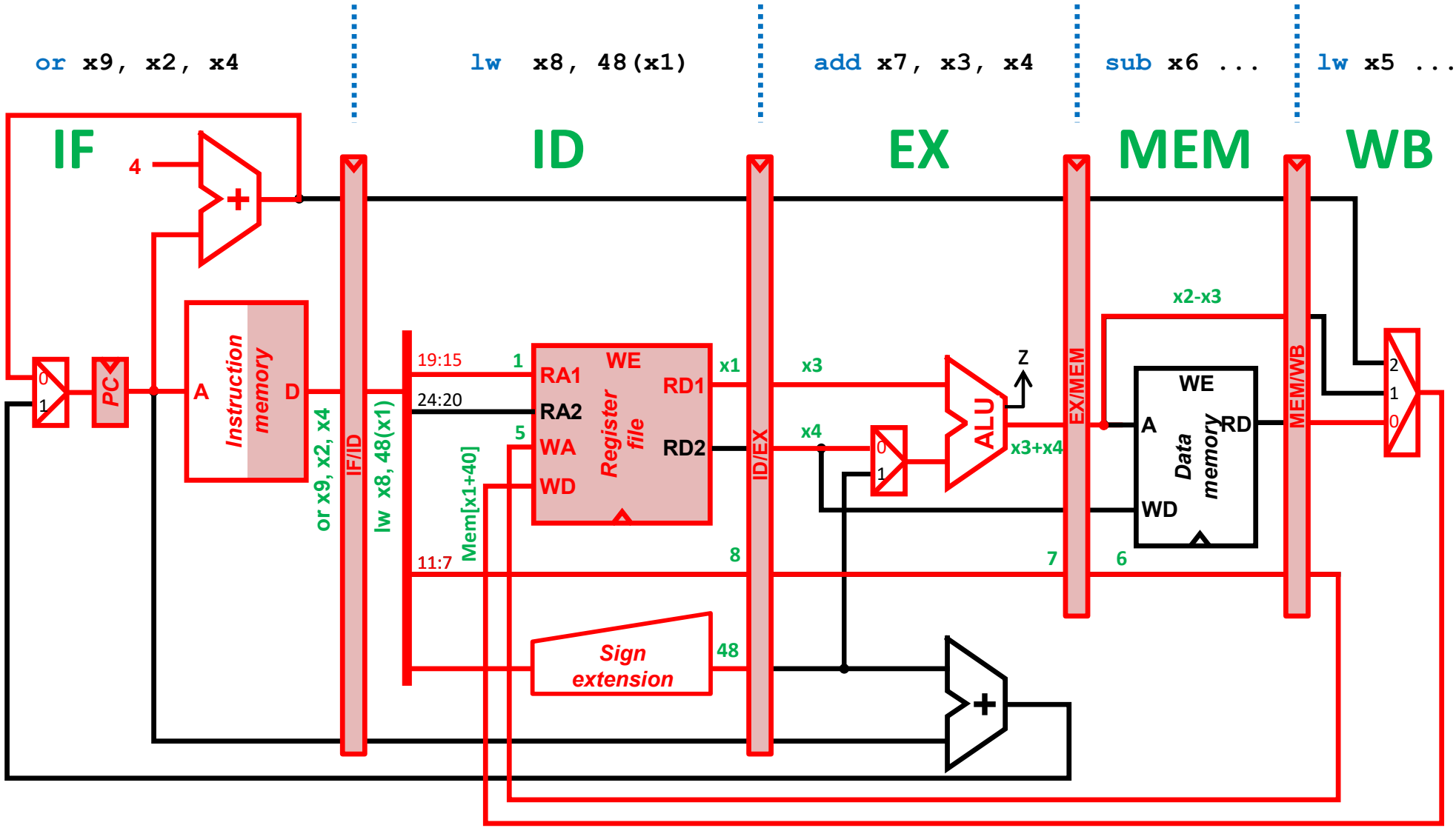
Data path design

Simulation: 4th. cycle



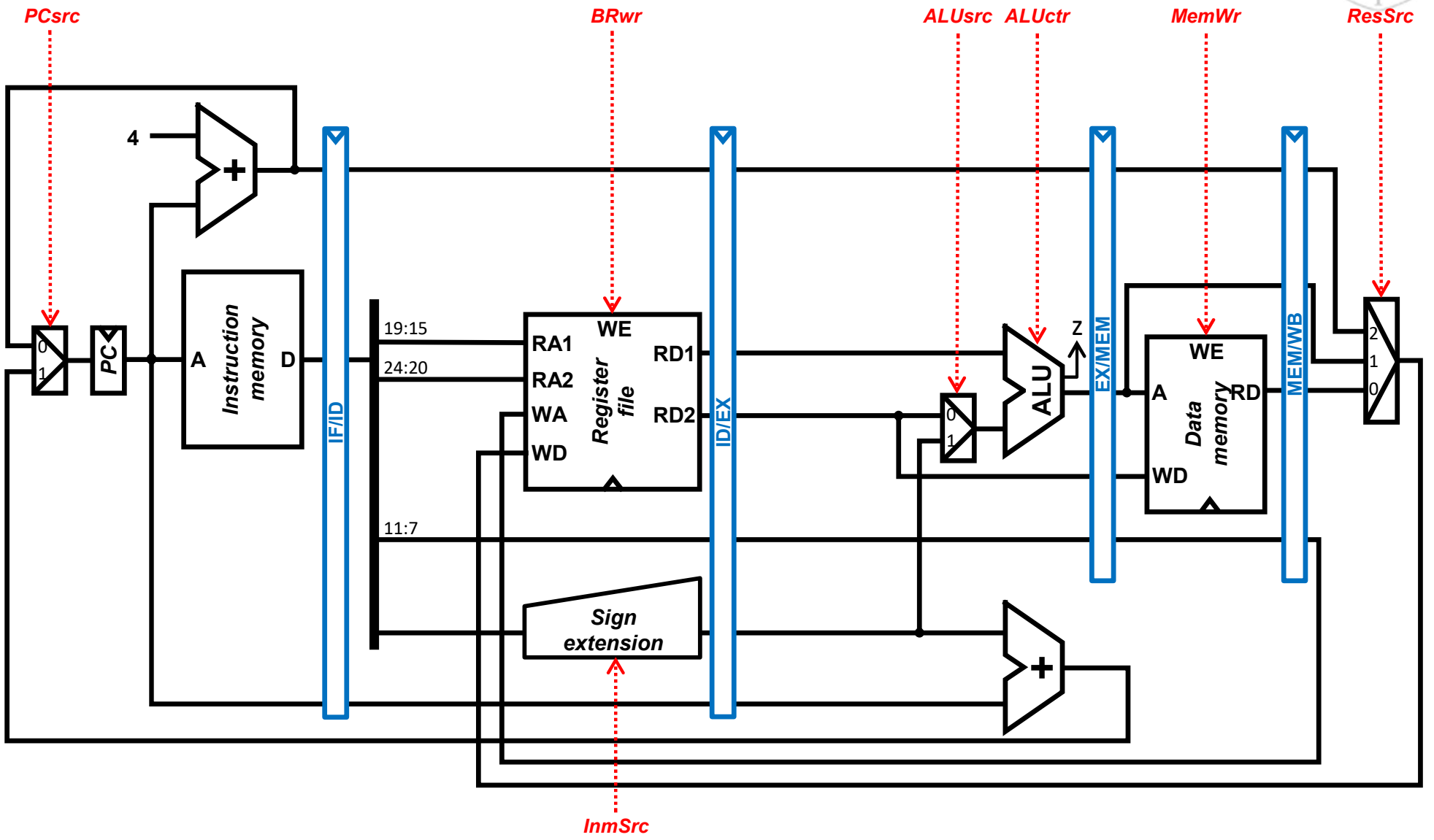
Data path design

Simulation: 5th. cycle



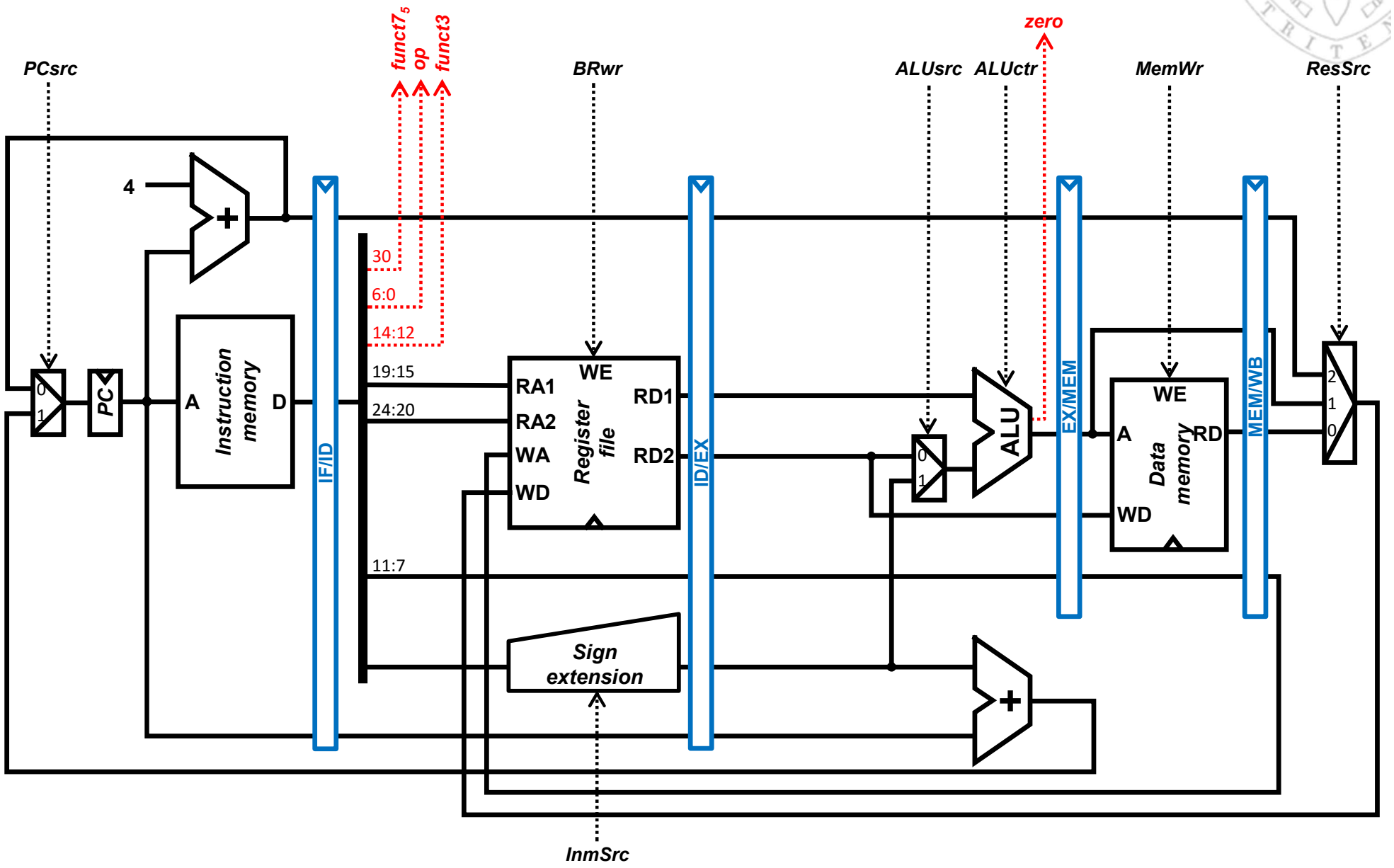
Data path design

Control signals



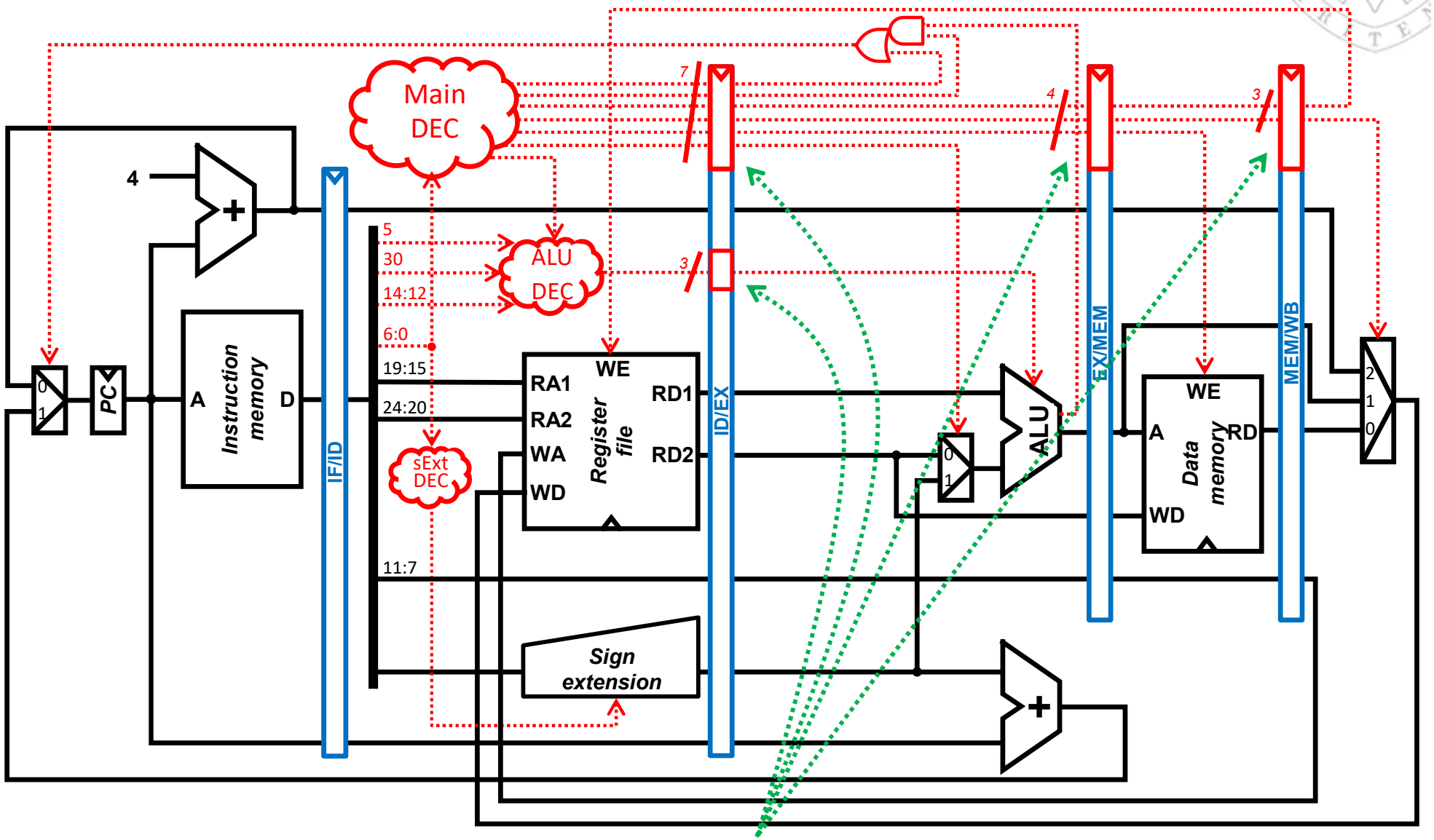
Data path design

Control signals



Controller design

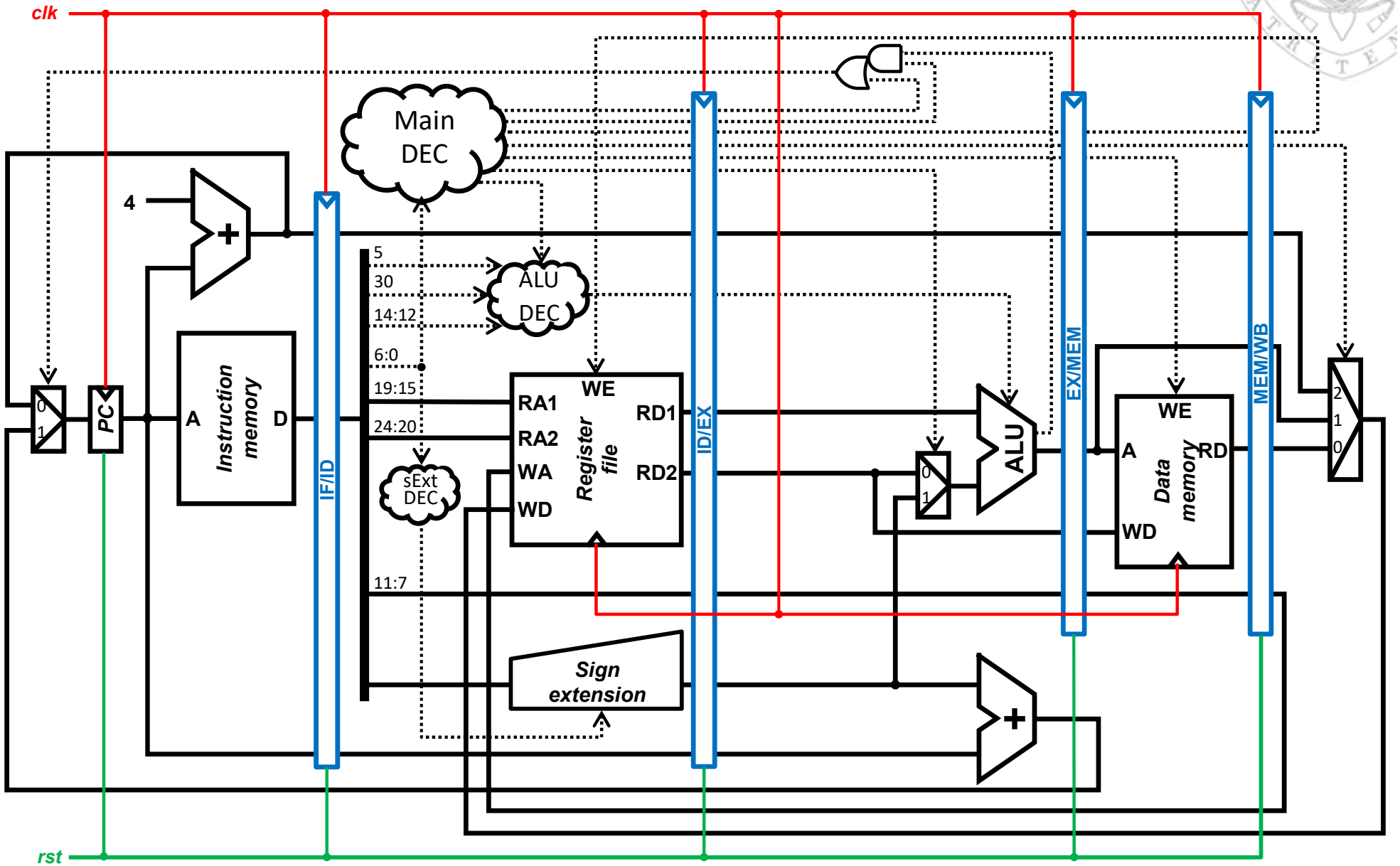
Connection with the controller



Pipeline registers are extended to transmit the control signals that are needed through the stages

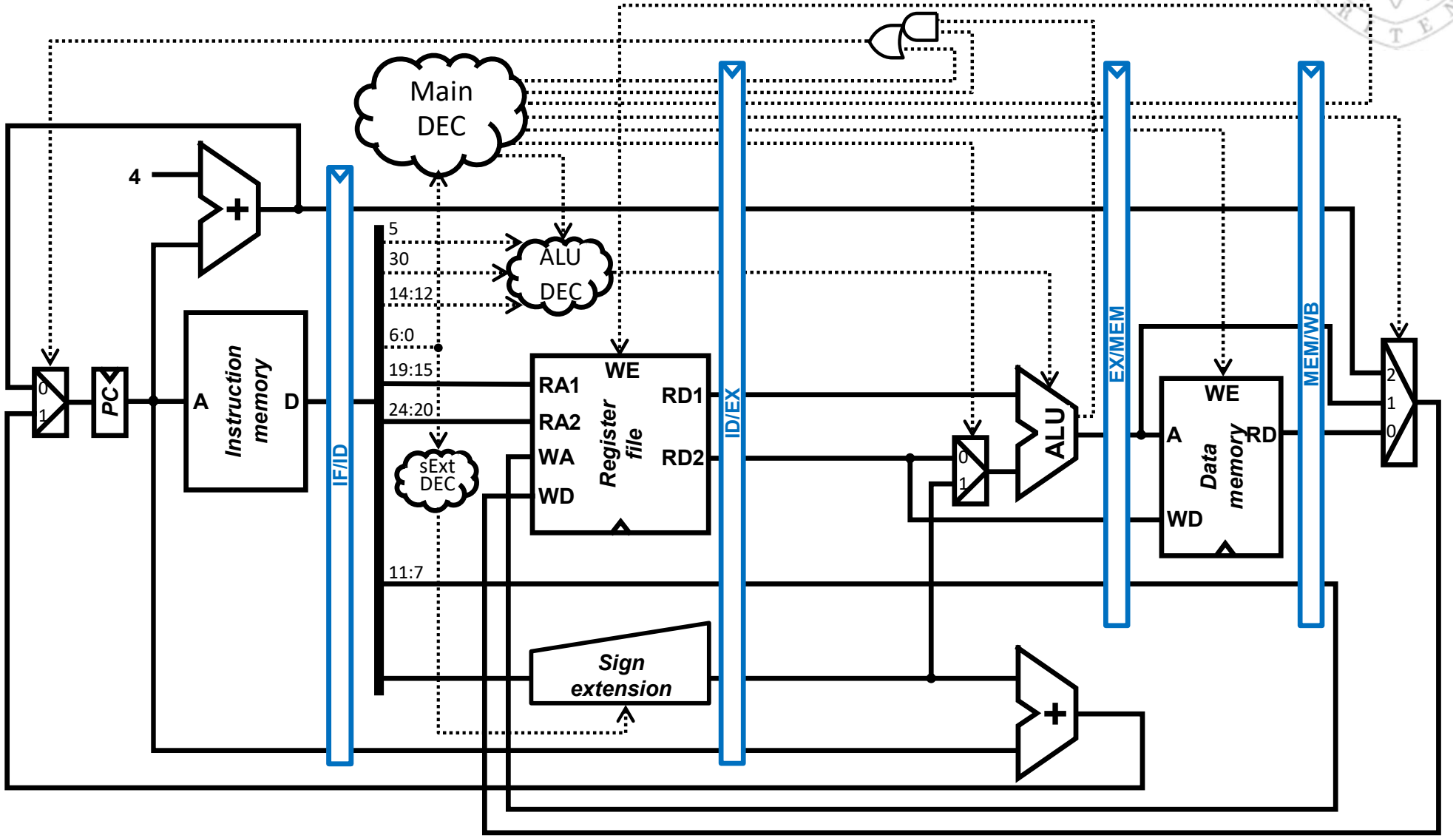
Pipelined processor

Connection of the clock and the reset



Pipelined processor

Full system structure



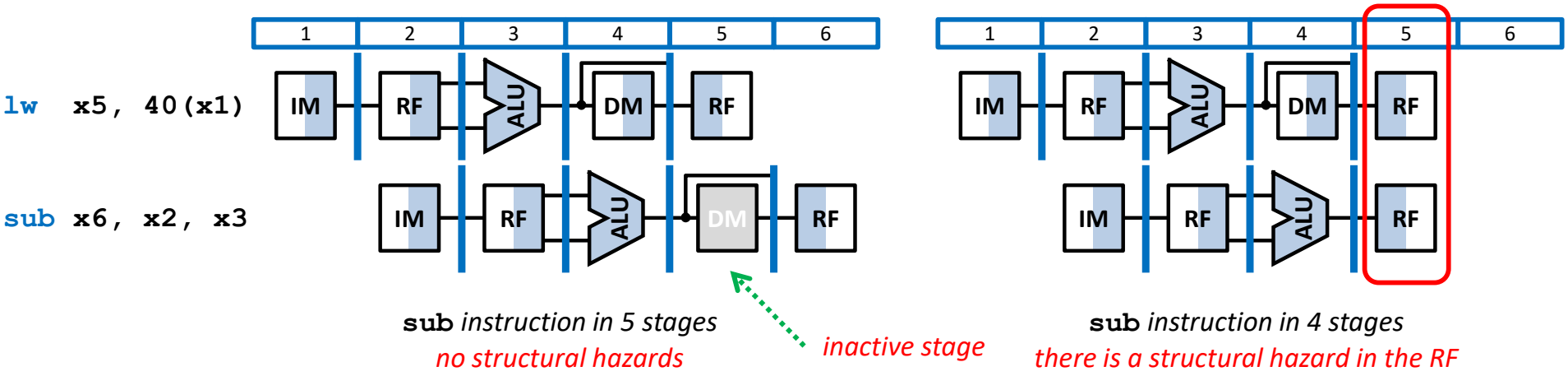
Hazards

- In a pipelined processor, hazards may appear between the instructions that are executed simultaneously.
  - These never happen in the single-cycle and multicycle processors because in those cases each instruction is executed after the previous one has ended.

- Types of hazards:
  - Structural: an instruction needs a hardware resource that is being used by a previous instruction.
  - Data: an instruction must read a data form a register that has not been written yet by a previous instruction.
  - Control: the next instruction must be fetched from memory, but its address has not been decided/calculated yet by a previous branch instruction.

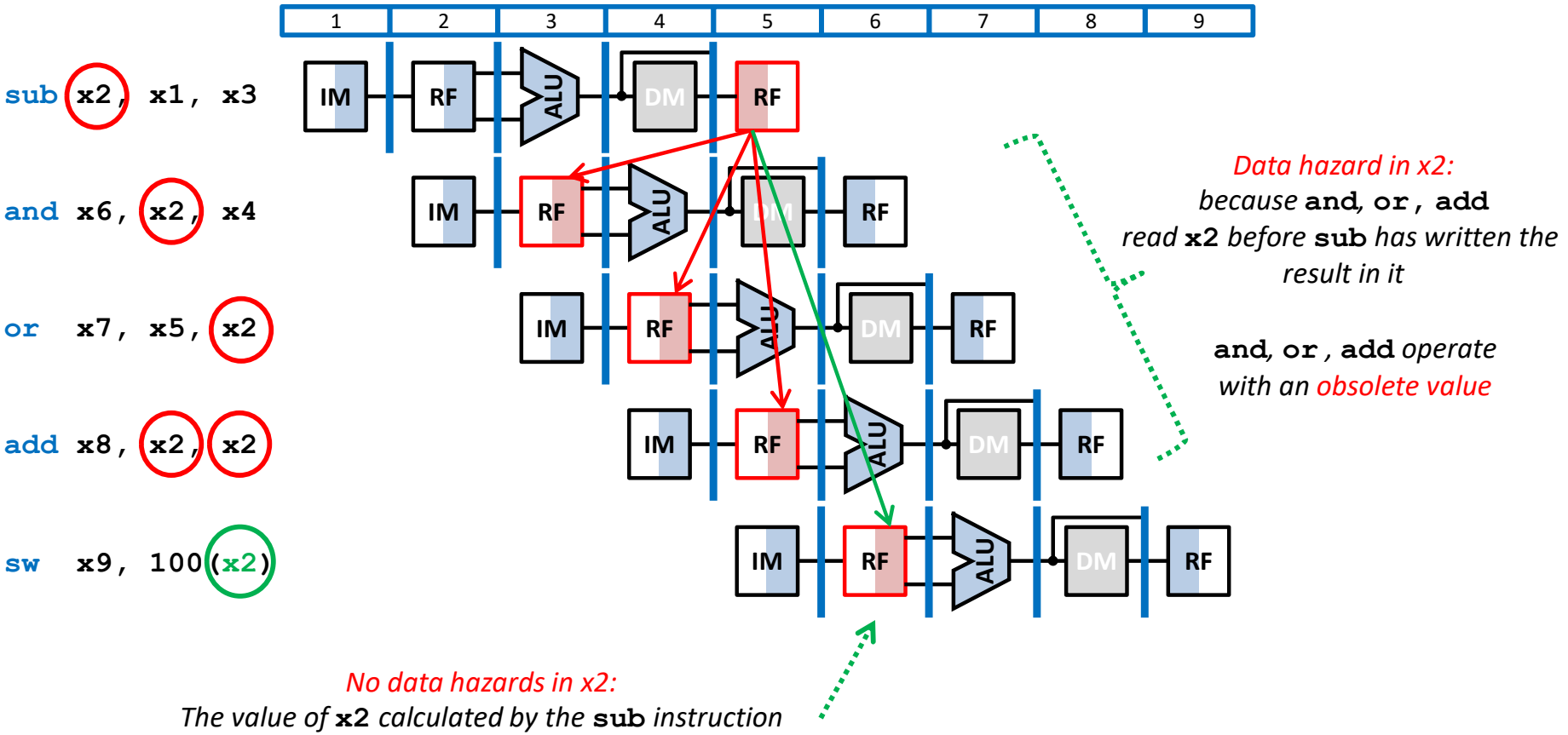
Structural hazards

- This pipelined processor does not have structural hazards because:
  - There are no shared resources.
    - The PC can be incremented (IF stage), the effective address calculated (EX stage) and the branch condition checked (EX stage) simultaneously.
  - Memory is split in two.
    - Instructions (IF stage) and data (MEM stage) can be read simultaneously.
  - The register file has a triple port.
    - 2 registers can be read (ID stage) and 1 written (WB stage) simultaneously.
  - All instructions go through the 5 stages.
    - Adding inactive stages when needed to avoid hazards.



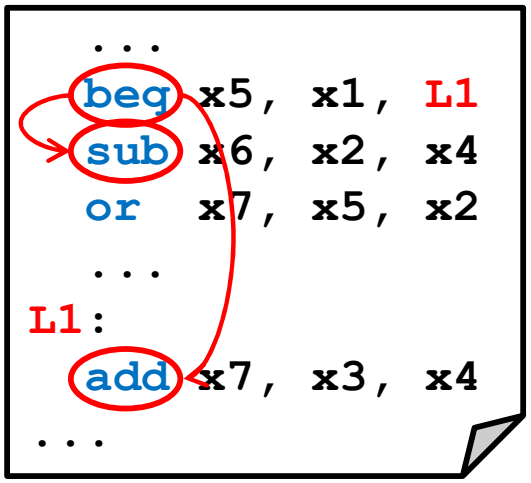
Data hazards

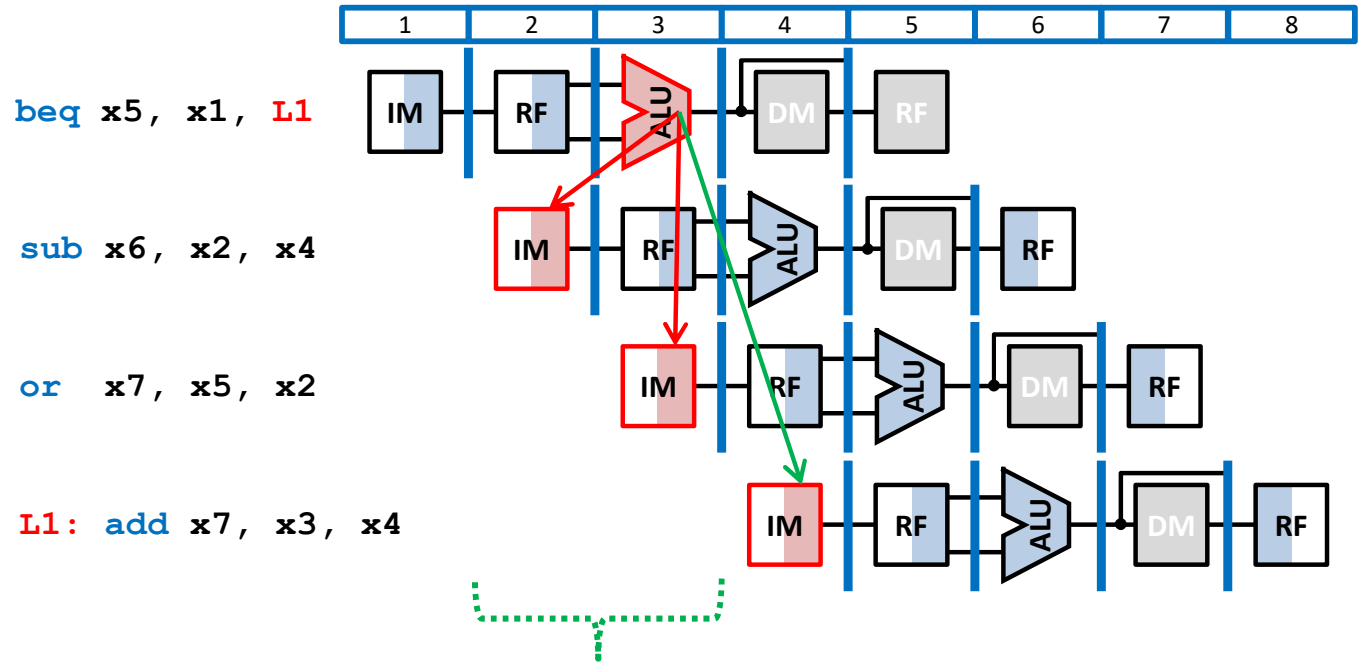
- This pipelined processor has data hazards when executing an instruction that needs to read a register written by any of the 3 previous instructions.



Control hazards

- This pipelined processor has control hazards when executing branch instructions, because the next instruction must be fetched:
  - Before deciding if the branch is taken or not (**beq** instruction)
  - Before having calculated the destination address, in the case the branch is taken (**beq** and **jal** instructions)



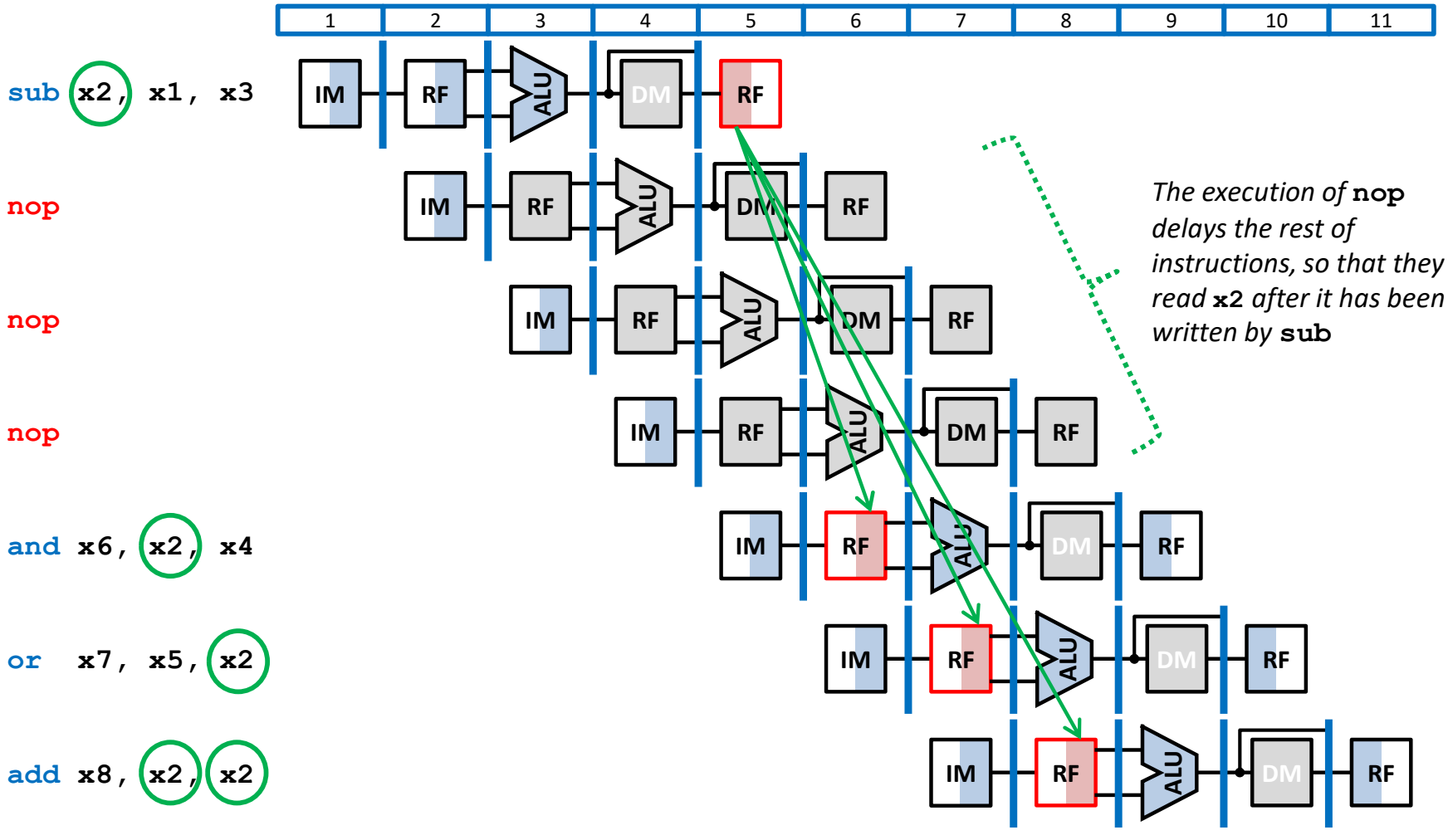


There is a control hazard because instructions are fetched before **beq** *decides if the branch is taken or not*

Data hazards

SW solution: inserting nop (i)

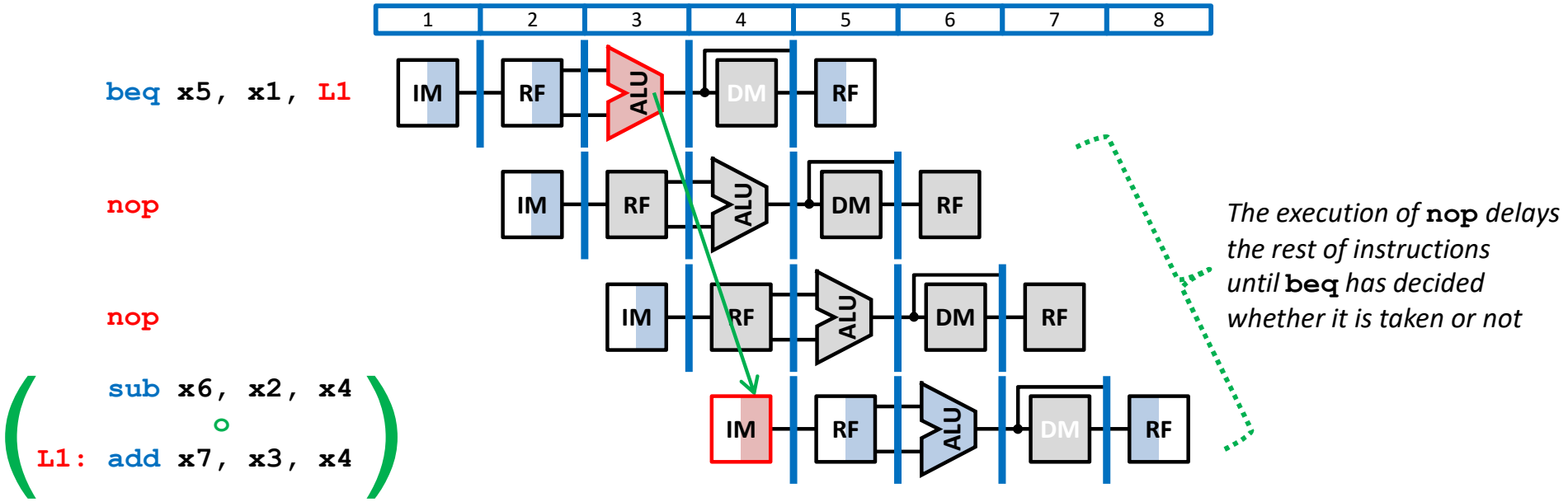
- They can be solved by software, inserting 1, 2 or 3 **nop** instructions between the instruction that writes the register and the one that reads it.



Control hazards

SW solution: inserting nop (ii)

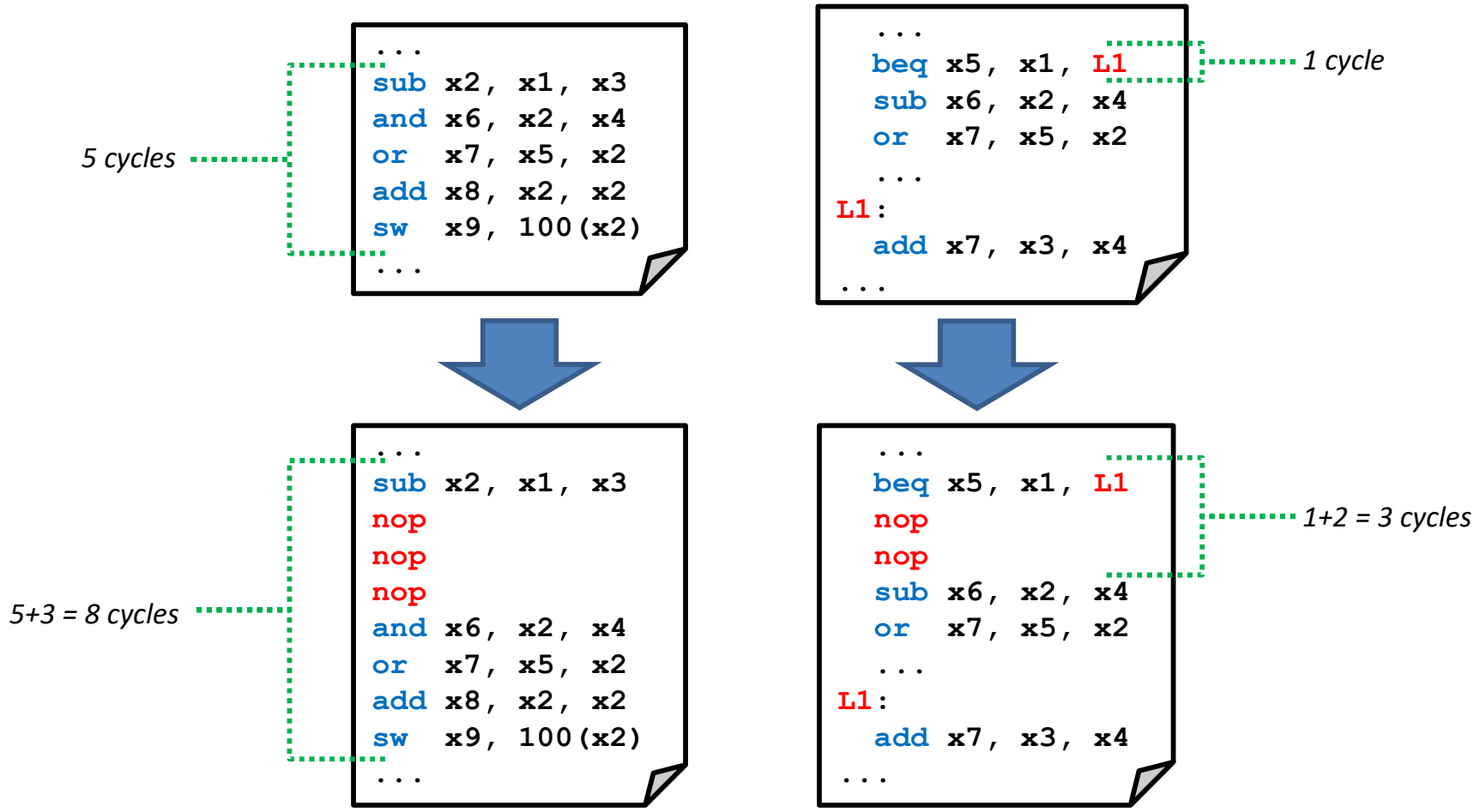
- They can be solved by software, inserting 2 **nop** instructions after each branch instruction.



Data and control hazards

SW solution: inserting nop (iii)

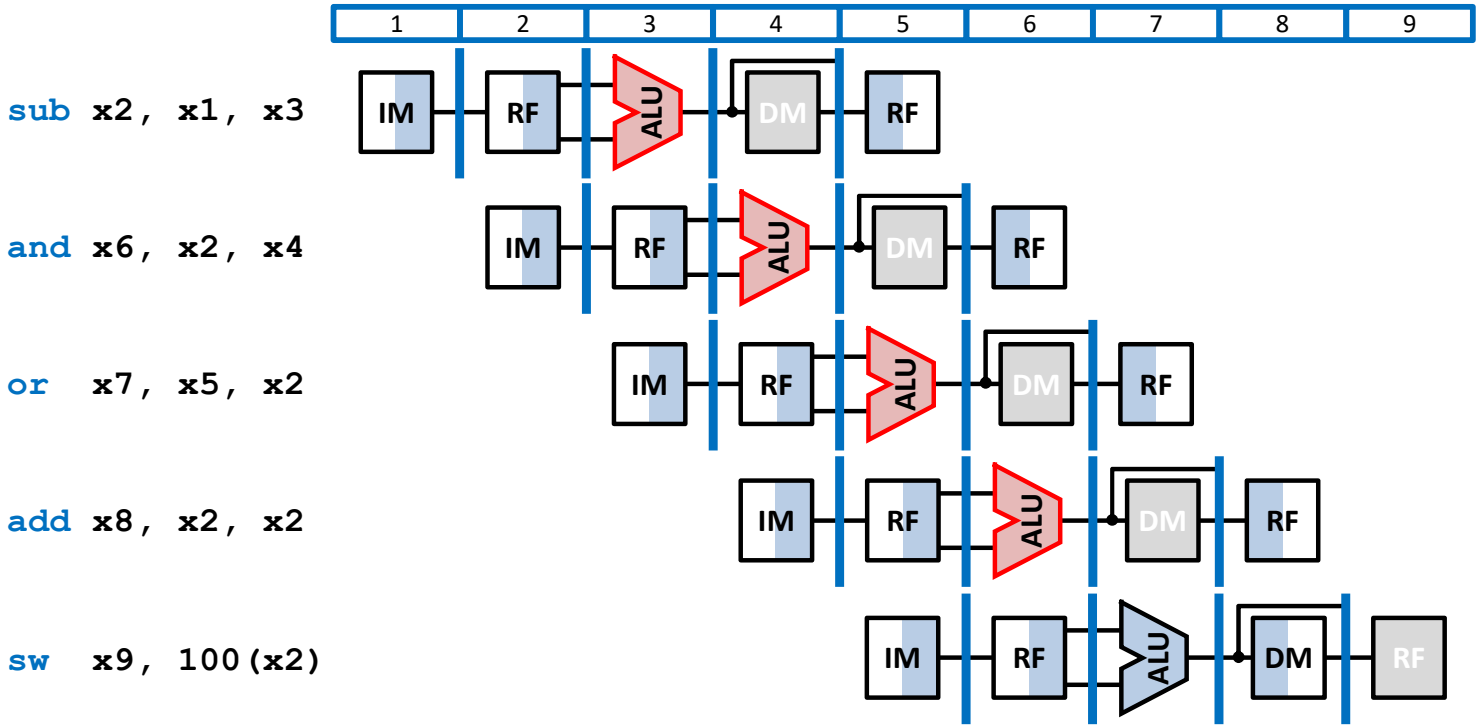
- The software solution has important disadvantages:
  - It makes programming harder because it requires inserting **nop** instructions.
  - The execution of each **nop** increases the execution time in 1 cycle.



Data hazards

HW solution: forwarding (i)

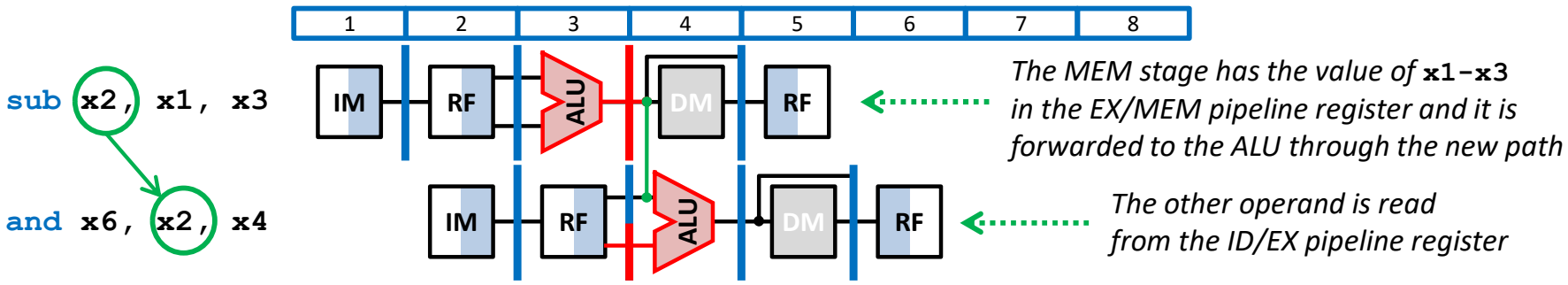
- There is a hardware solution that avoids this overhead given that:
  - The **sub** instruction uses the ALU in cycle 3 to calculate the subtraction.
  - The following instructions need the data in cycles 4, 5 and 6.
  - The data is available since cycle 4, and therefore it can be forwarded without waiting to read it from the RF.



Data hazards

HW solution: forwarding (ii)

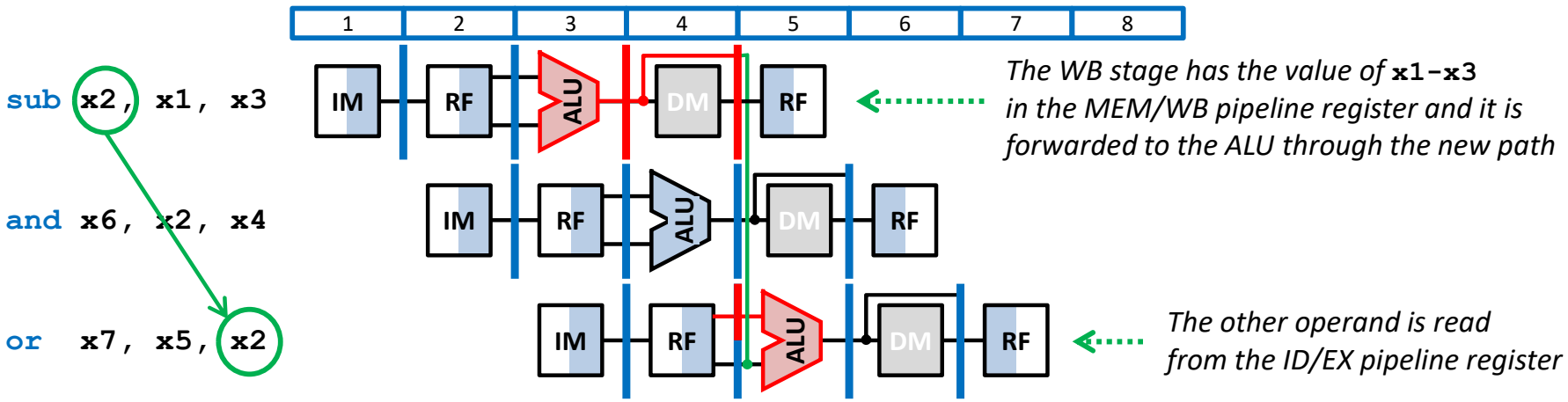
- Each hazard is solved in a different way:
  - **`sub - and`**: signal paths are added to forward the data from the MEM stage to each of the ALU inputs (EX stage)



Data hazards

HW solution: forwarding (ii)

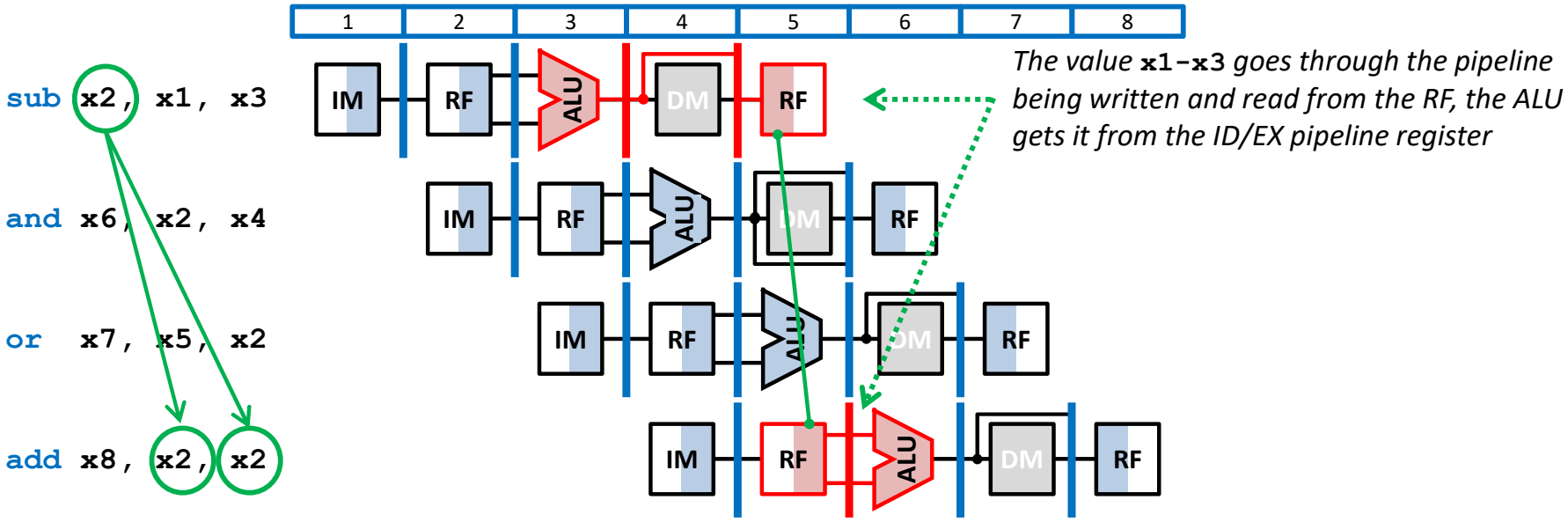
- Each hazard is solved in a different way:
  - **`sub - and`**: signal paths are added to forward the data from the MEM stage to each of the ALU inputs (EX stage)
  - **`sub - or`**: signal paths are added to forward the data from the WB stage to each of the ALU inputs (EX stage)



Data hazards

HW solution: forwarding (ii)

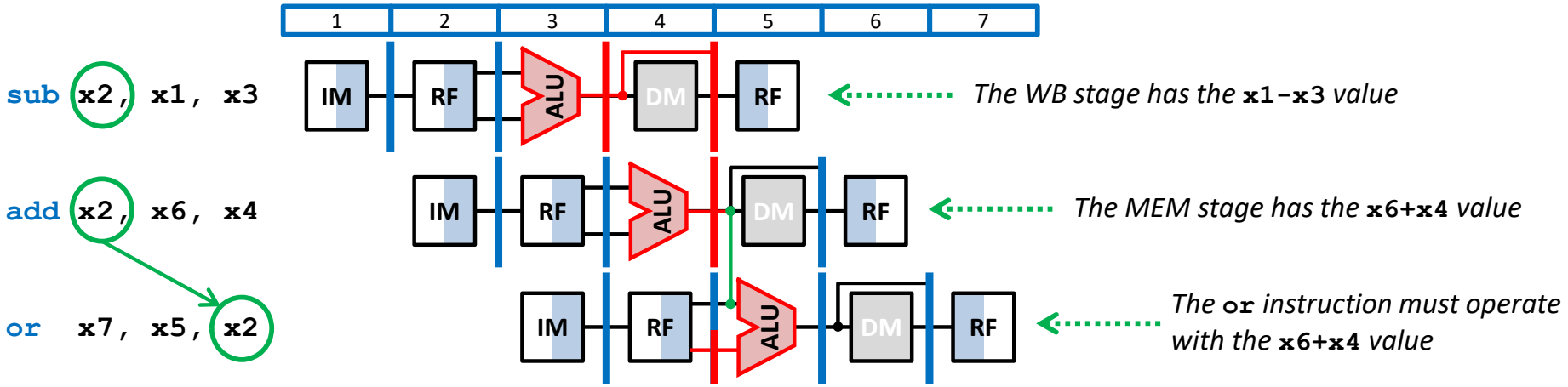
- Each hazard is solved in a different way:
  - **sub - and**: signal paths are added to forward the data from the MEM stage to each of the ALU inputs (EX stage)
  - **sub - or**: signal paths are added to forward the data from the WB stage to each of the ALU inputs (EX stage)
  - **sub - add**: it is solved by writing the RF at the end of the clock cycle first half, so that it can be read in the second half.



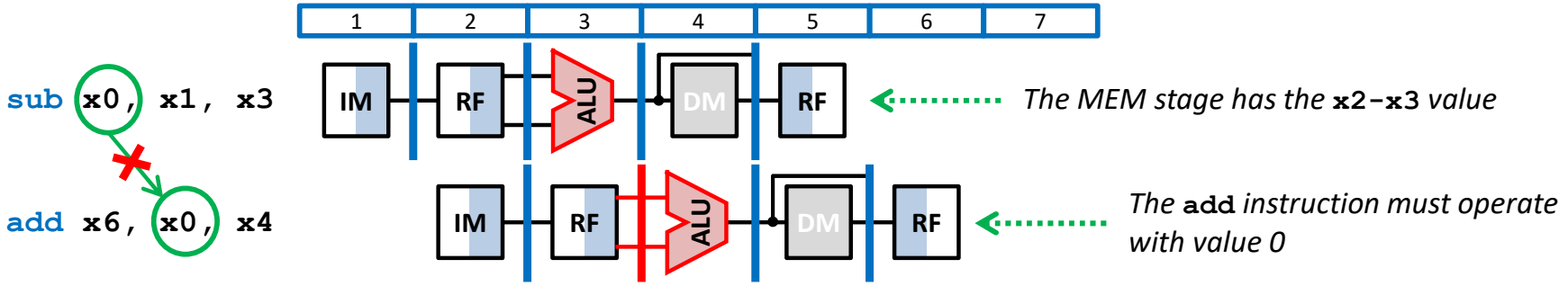
Data hazards

HW solution: forwarding (iii)

- In the case a register can be forwarded from MEM and WB:
  - It has to be done from the MEM stage, since this has the most recent value of the register causing the hazard.



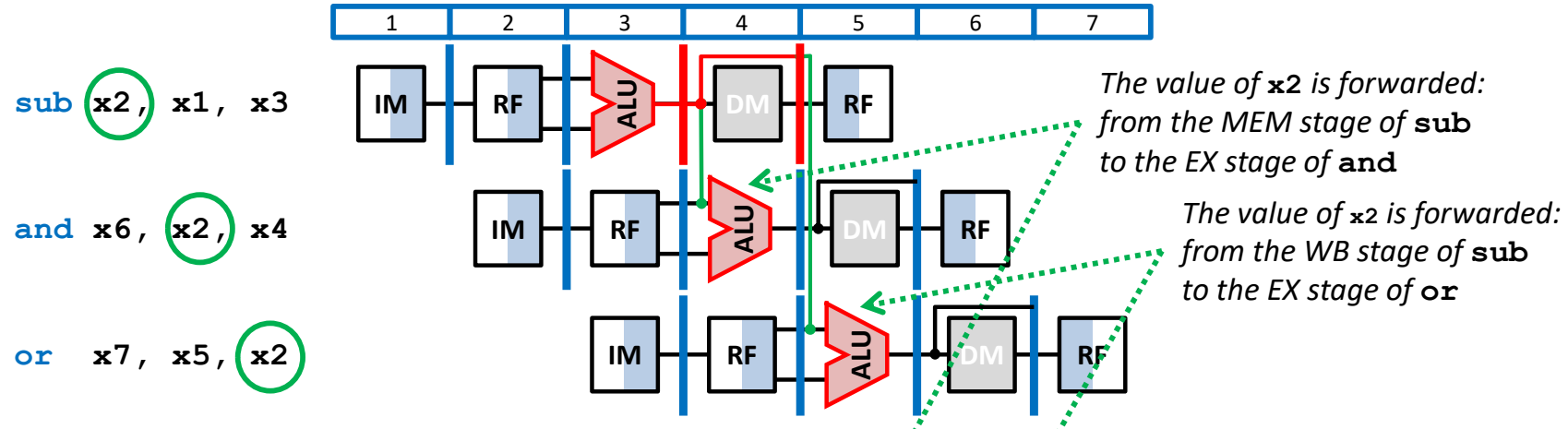
- The **x0** register is never forwarded:



Data hazards

HW solution: forwarding (iv)

- In the simplified execution diagrams, forwarding is indicated as dependencies between stages:



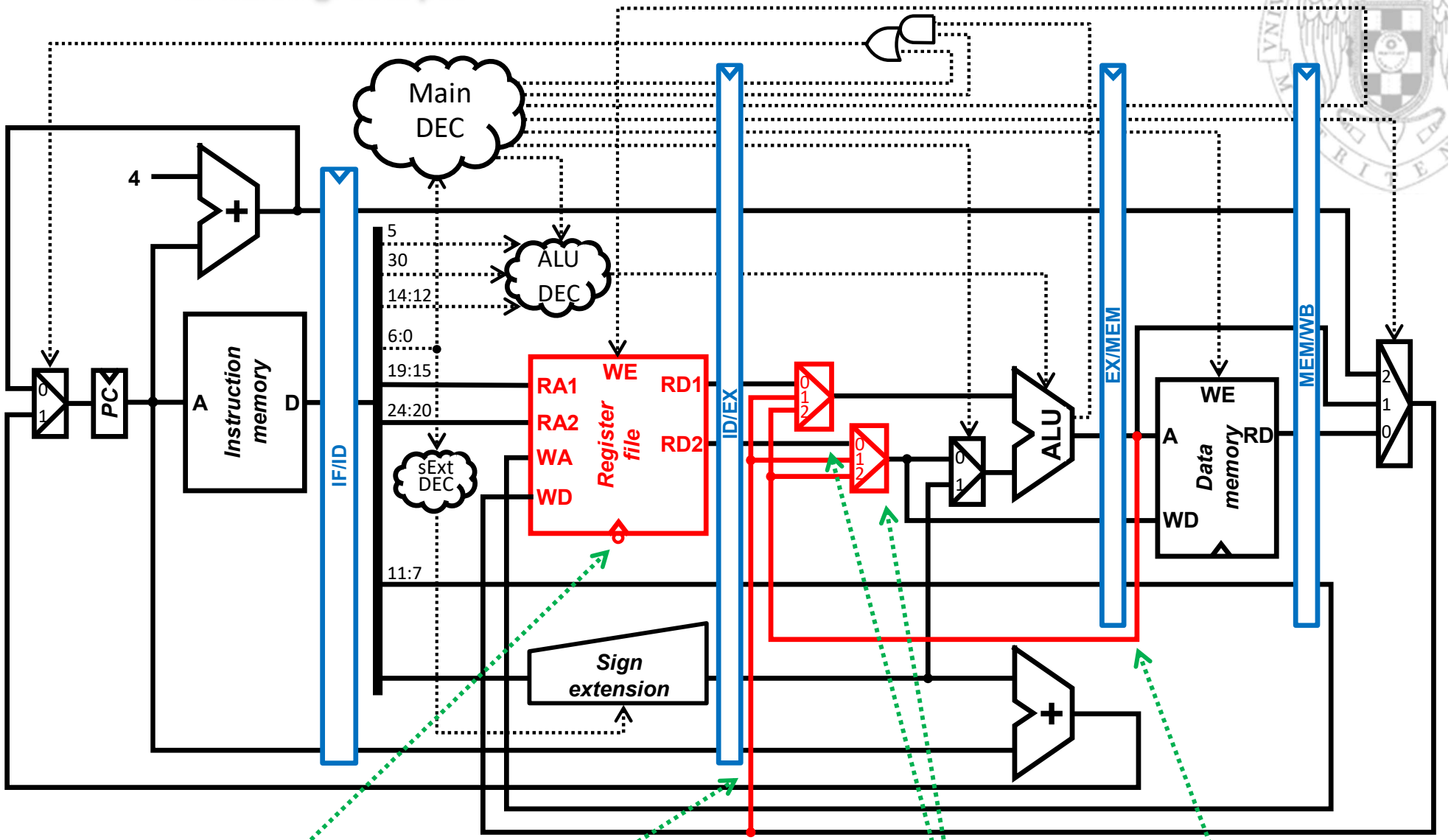
	1	2	3	4	5	6	7
sub x2, x1, x3	IF	ID	EX	M	WB		
and x6, x2, x4		IF	ID	EX	M	WB	
or x7, x5, x2			IF	ID	EX	M	WB

x2 (sub M → and EX)

x2 (sub WB → or EX)

Pipelined processor

+ forwarding: data path



By inverting the clock input, The RF loads at the falling edge (half cycle of the rising edge)

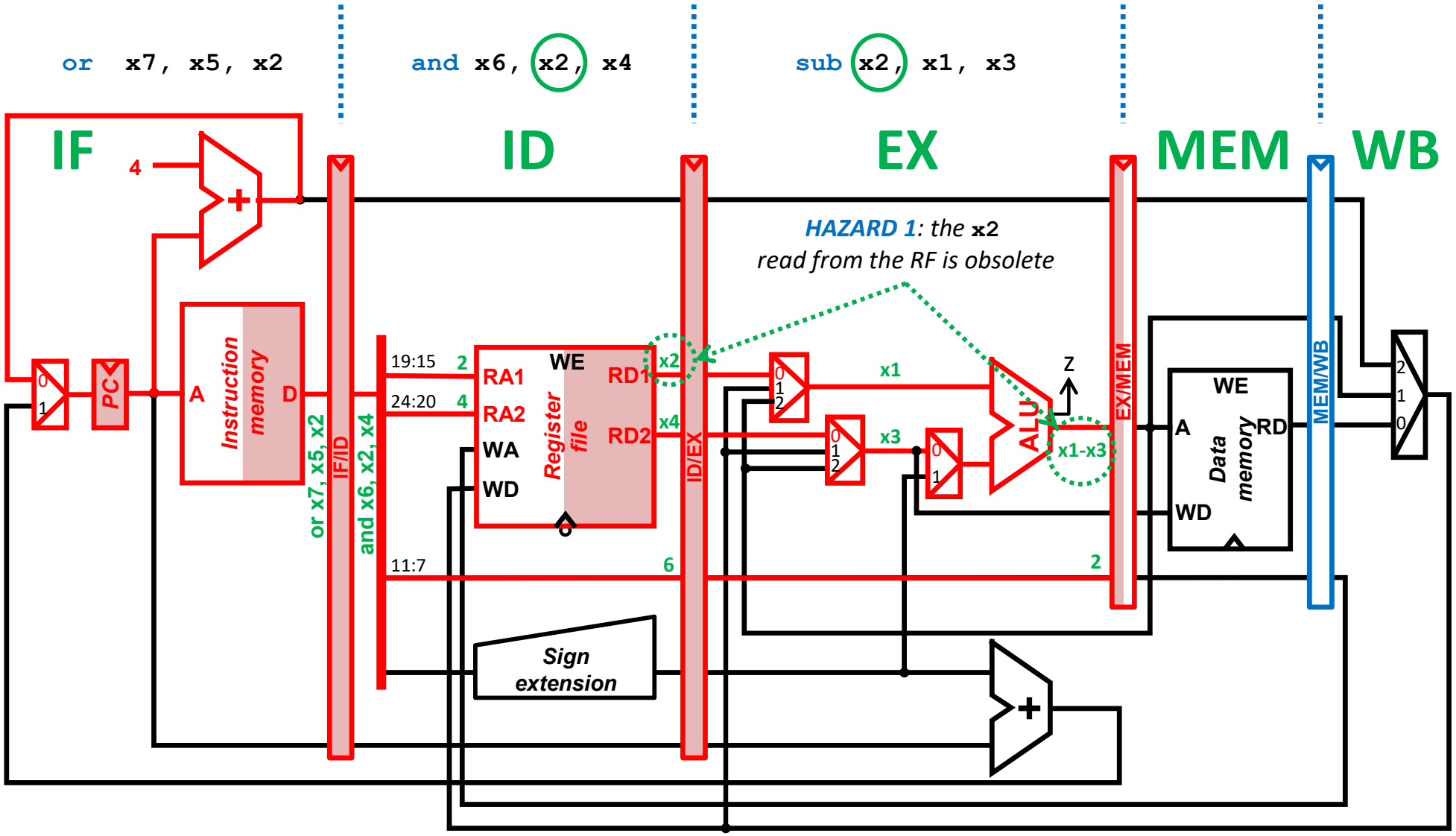
A forwarding path is added from the WB stage to the EX stage

A forwarding path is added from the MEM stage to the EX stage

MUX are added to the ALU inputs

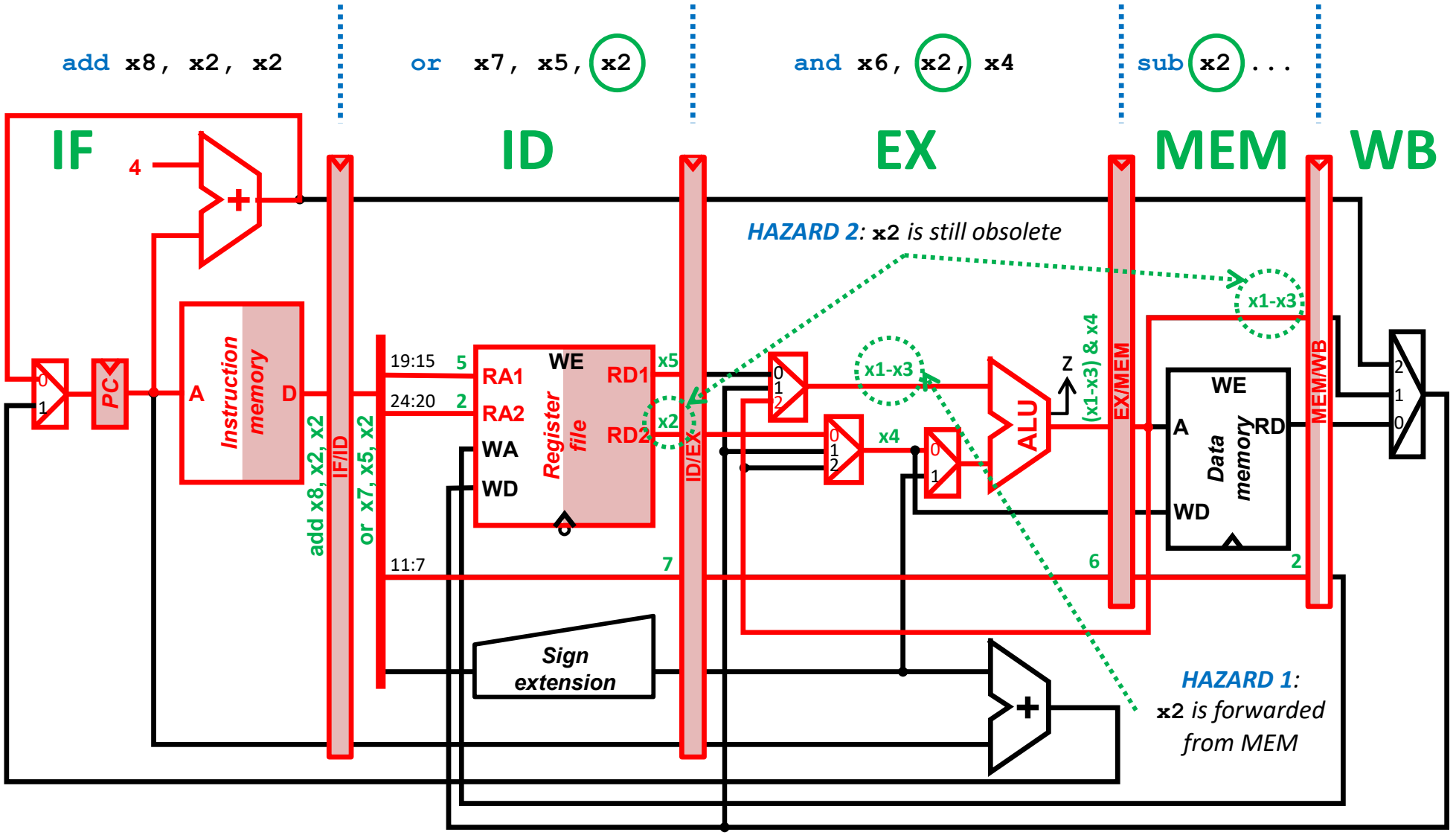
Pipelined processor

Forwarding simulation: 3rd. cycle



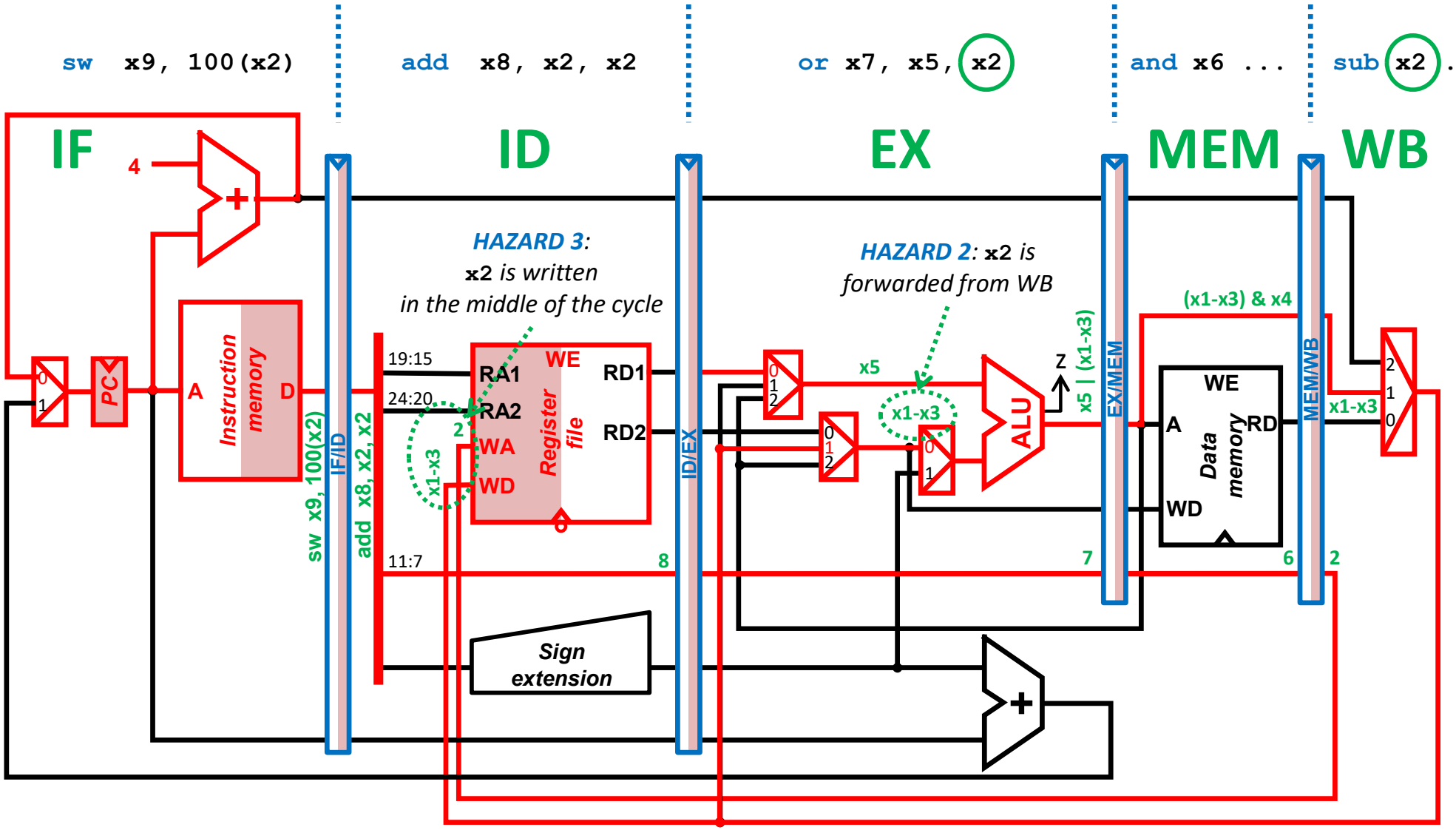
Pipelined processor

Forwarding simulation: 4th. cycle



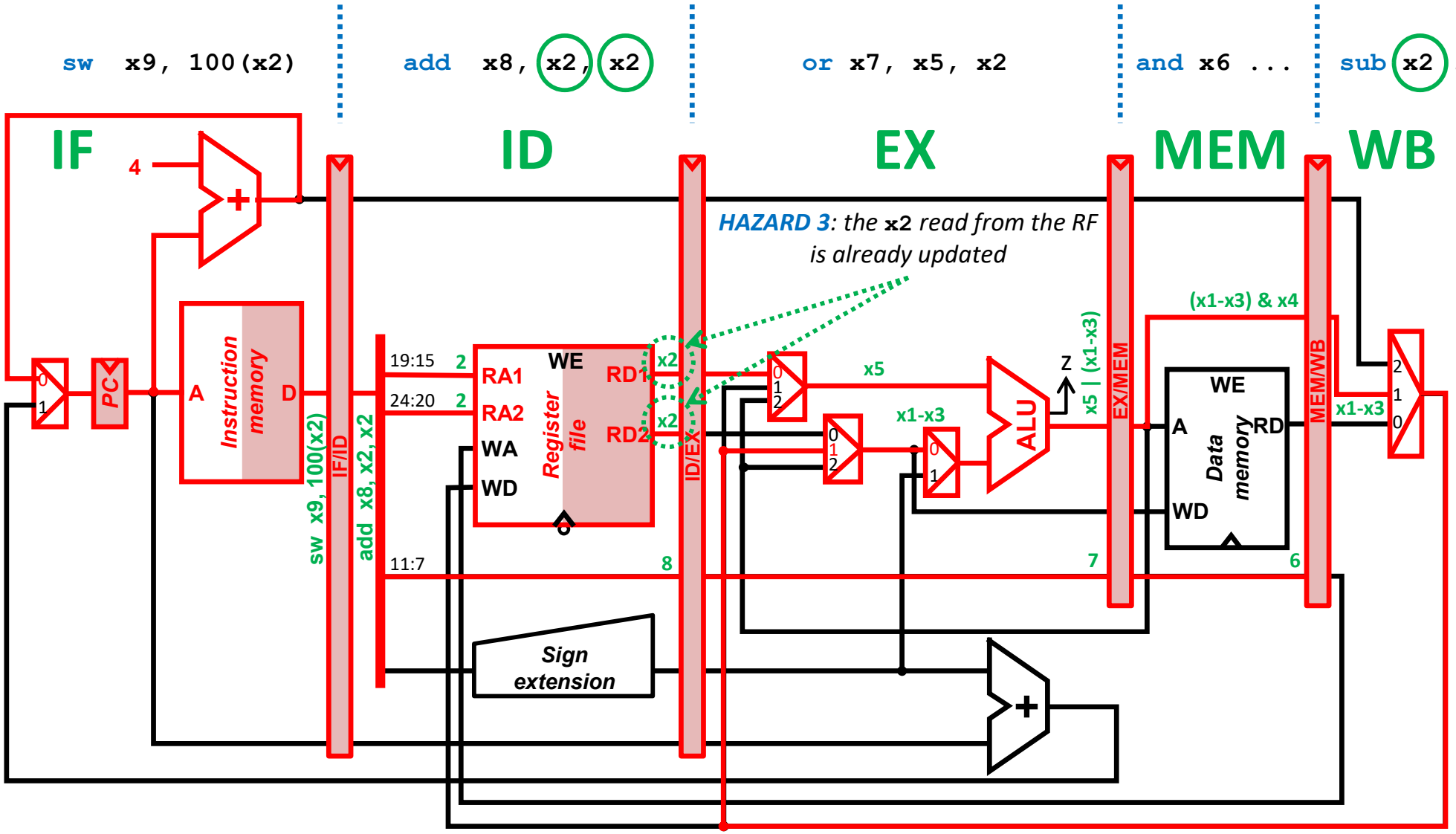
Pipelined processor

Forwarding simulation: 5th. cycle (1st. half)



Pipelined processor

Forwarding simulation: 5th. cycle (2nd. half)



Pipelined processor

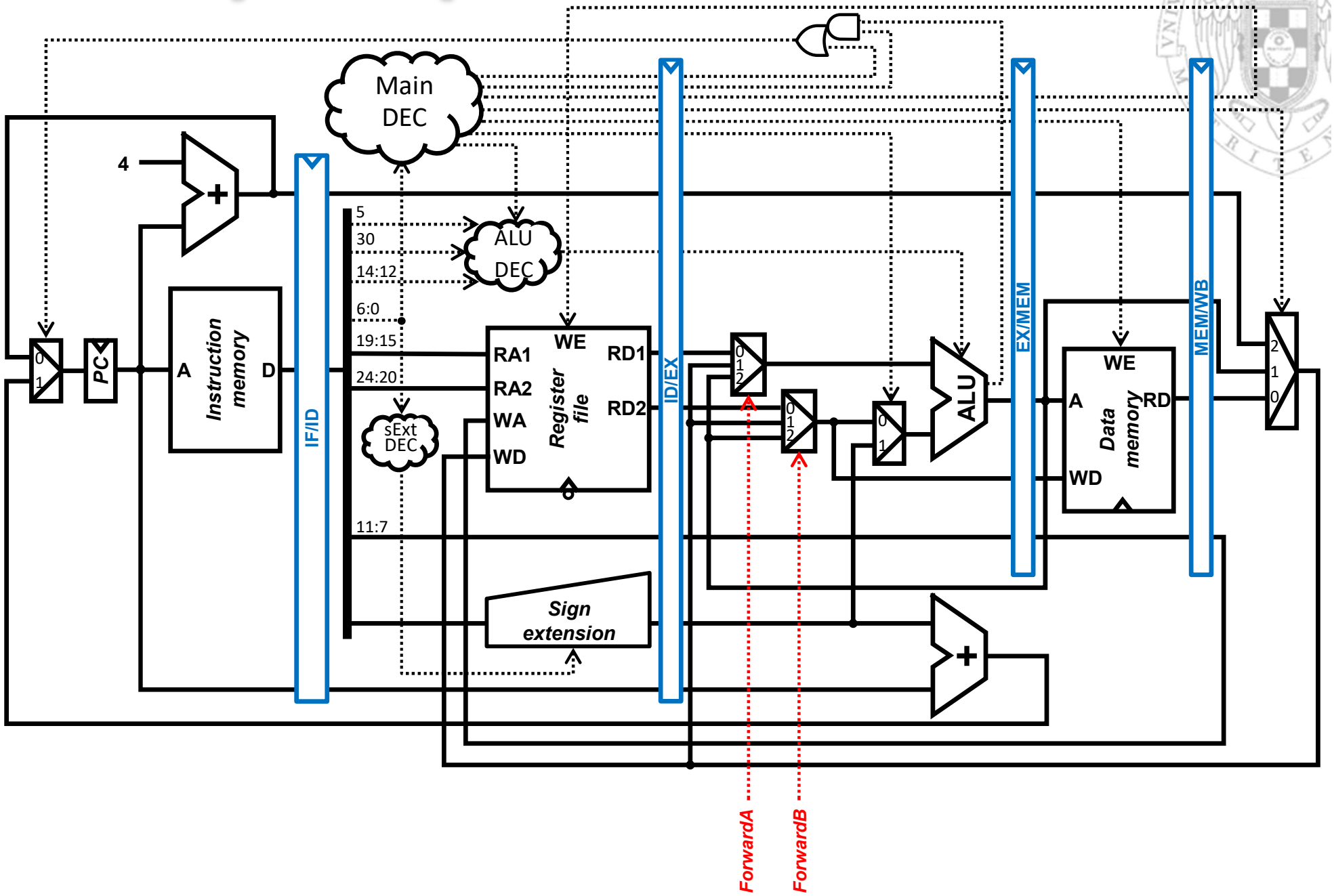
Forwarding unit

- The forwarding unit is a combinational circuit that controls the forwarding MUX so that the ALU can operate with data:
  - Read from the RF and available in the ID/EX forwarding register.
  - Available in pipeline registers of the following stages (EX/MEM or MEM/WB)

- In order to behave correctly, it must know:
  - Rs1E: number of source register 1 of the instruction in the EX stage.
  - Rs2E: number of source register 2 of the instruction in the EX stage.
  - RdM: number of the destination register of the instruction in the MEM stage.
  - BRwrM: whether the instruction in the MEM stage writes in the RF or not.
  - RdW: number of the destination register of the instruction in the WB stage.
  - BRweW: whether the instruction in the WB stage writes in the RF or not.

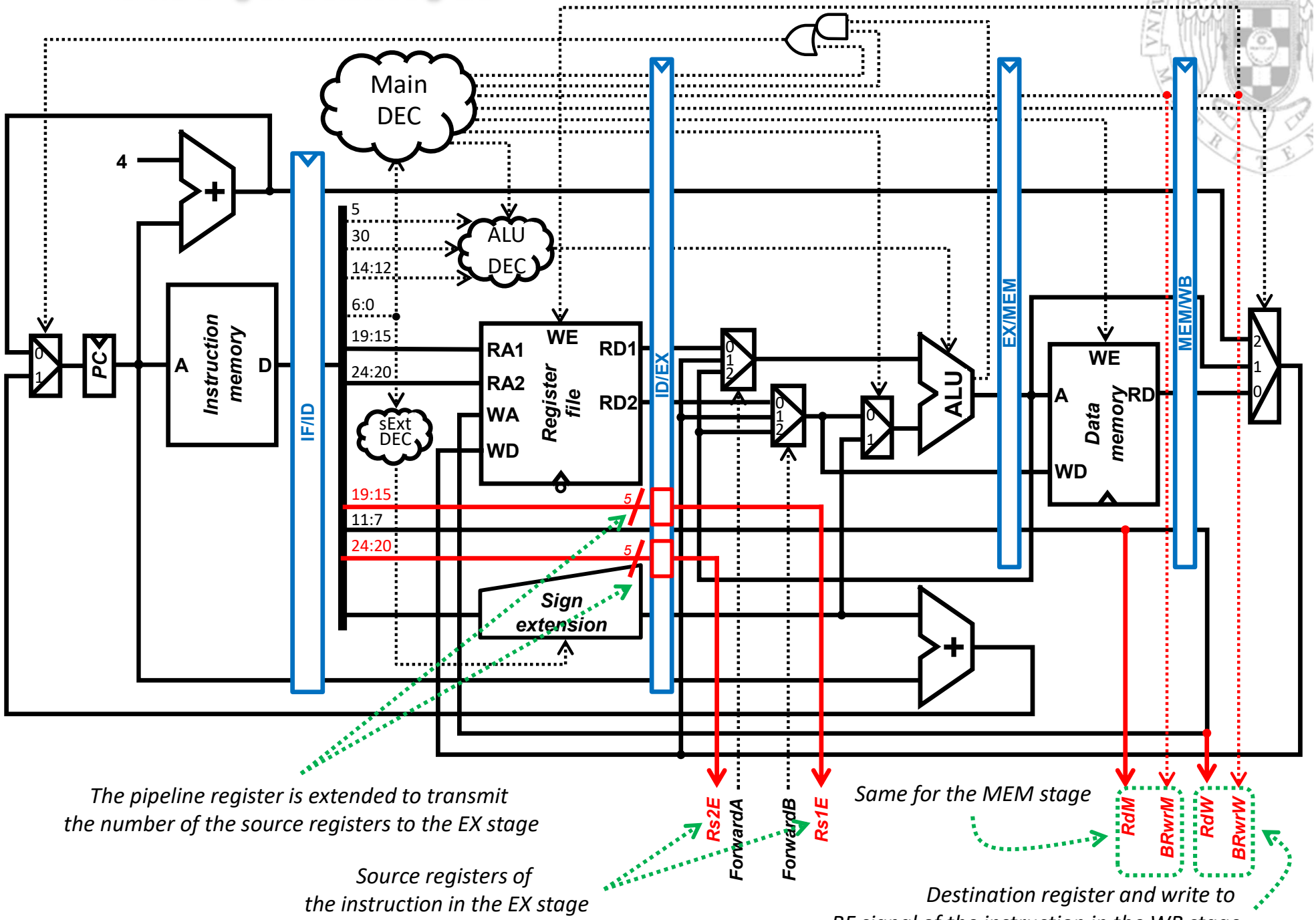
Pipelined processor

+ Forwarding unit: control signals



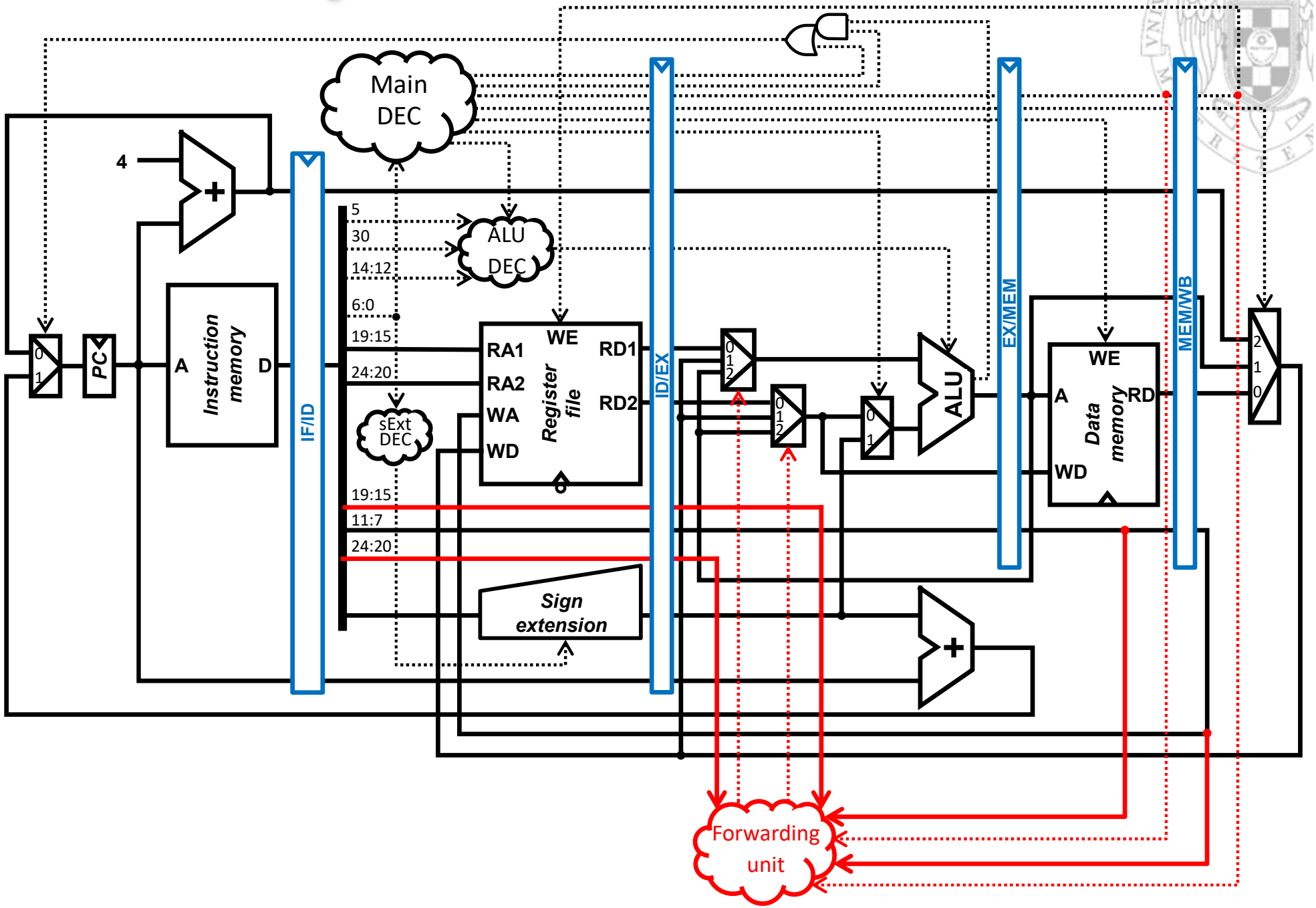
Pipelined processor

+ Forwarding unit: status signals



Pipelined processor

+ Forwarding unit



Pipelined processor

Forwarding unit design (i)

- A data must be forwarded to the input A of the ALU:
  - From the MEM stage, if the destination register of the MEM stage (RdM) will be written (BRwrM) and coincides with the source register of the EX stage (Rs1E).

ForwardA ← *if* (BRwrM & (Rs1E = RdM)) *then* (10) ⟵ *Forwarding from MEM*

Pipelined processor

Forwarding unit design (i)

- A data must be forwarded to the input A of the ALU:
  - From the MEM stage, if the destination register of the MEM stage (RdM) will be written (BRwrM) and coincides with the source register of the EX stage (Rs1E).
  - From the WB stage, if the destination register of the WB stage (RdW) will be written (BRwrW) and coincides with the source register of the EX stage (Rs1E).
    - This condition is only checked if the previous one is not met, because when the data can be forwarded from both stages, it has to be taken from the MEM stage.

```
ForwardA ← if (      BRwrM & (Rs1E = RdM) )   then  ( 10 )   <..... Forwarding from MEM
           elsif(    BRwrW & (Rs1E = RdW) )   then  ( 01 )   <..... Forwarding from WB
```

Pipelined processor

Forwarding unit design (i)

- A data must be forwarded to the input A of the ALU:
  - From the MEM stage, if the destination register of the MEM stage (RdM) will be written (BRwrM) and coincides with the source register of the EX stage (Rs1E).
  - From the WB stage, if the destination register of the WB stage (RdW) will be written (BRwrW) and coincides with the source register of the EX stage (Rs1E).
    - This condition is only checked if the previous one is not met, because when the data can be forwarded from both stages, it has to be taken from the MEM stage.
  - Register x0 is never forwarded because it has a constant value of 0.

ForwardA ← *if* ((Rs1E ≠ 0) & BRwrM & (Rs1E = RdM)) *then* (10) ⟵ *Forwarding from MEM*
elsif((Rs1E ≠ 0) & BRwrW & (Rs1E = RdW)) *then* (01) ⟵ *Forwarding from WB*

Pipelined processor

Forwarding unit design (i)

- A data must be forwarded to the input A of the ALU:
  - From the MEM stage, if the destination register of the MEM stage (RdM) will be written (BRwrM) and coincides with the source register of the EX stage (Rs1E).
  - From the WB stage, if the destination register of the WB stage (RdW) will be written (BRwrW) and coincides with the source register of the EX stage (Rs1E).
    - This condition is only checked if the previous one is not met, because when the data can be forwarded from both stages, it has to be taken from the MEM stage.
  - Register x0 is never forwarded because it has a constant value of 0.
- Otherwise, do not forward.

ForwardA ← *if* ((Rs1E ≠ 0) & BRwrM & (Rs1E = RdM)) *then* (10) ⟵ *Forwarding from MEM*
elsif((Rs1E ≠ 0) & BRwrW & (Rs1E = RdW)) *then* (01) ⟵ *Forwarding from WB*
else (00) ⟵ *Do not forward*

- Same for the forwarding unit to the input B of the ALU
  - Replacing RS1E with RS2E.

Pipelined processor

Forwarding unit design (ii)

ForwardA ← *if* ((Rs1E ≠ 0) & BRwrM & (Rs1E = RdM)) *then* (10)
elsif((Rs1E ≠ 0) & BRwrW & (Rs1E = RdW)) *then* (01)
else (00)

Truth table

Rs1E ≠ 0	BRwrM	BRwrW	Rs1E = RdM	Rs1E = RdW	ForwardA
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

Pipelined processor

Forwarding unit design (ii)

ForwardA ← *if* ((Rs1E ≠ 0) & BRwrM & (Rs1E = RdM)) *then* (10)
elsif((Rs1E ≠ 0) & BRwrW & (Rs1E = RdW)) *then* (01)
else (00)

ForwardB ← *if* ((Rs2E ≠ 0) & BRwrM & (Rs2E = RdM)) *then* (10)
elsif((Rs2E ≠ 0) & BRwrW & (Rs2E = RdW)) *then* (01)
else (00)

Truth table

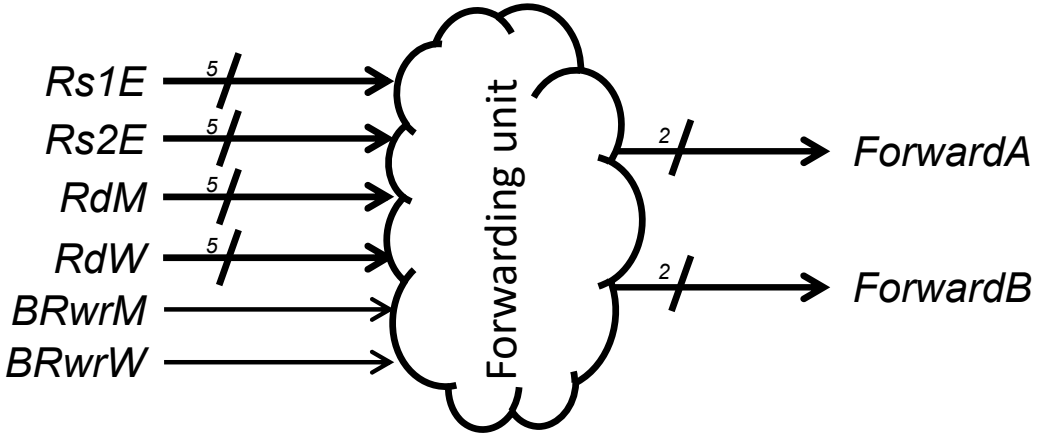
Rs1E ≠ 0	BRwrM	BRwrW	Rs1E = RdM	Rs1E = RdW	ForwardA
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

Truth table

Rs2E ≠ 0	BRwrM	BRwrW	Rs2E = RdM	Rs2E = RdW	ForwardB
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

Pipelined processor

Forwarding unit design (iii)



Truth table

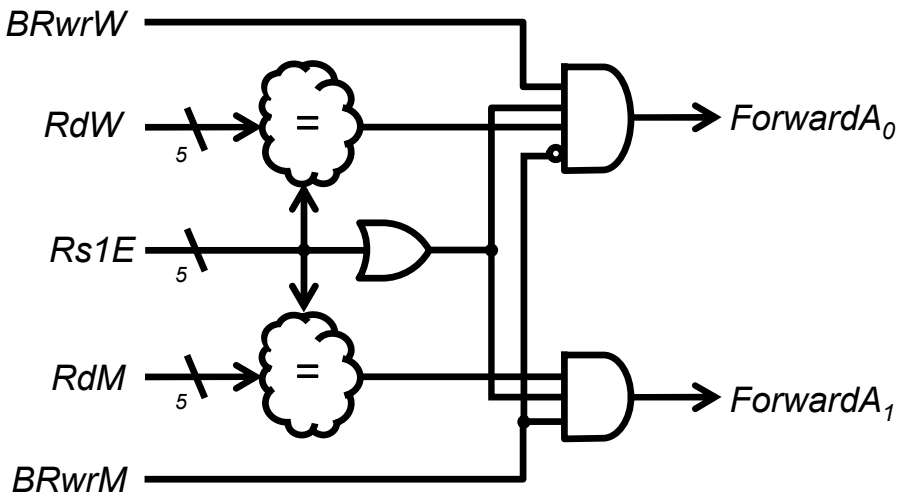
Rs1E ≠ 0	BRwrM	BRwrW	Rs1E = RdM	Rs1E = RdW	ForwardA
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

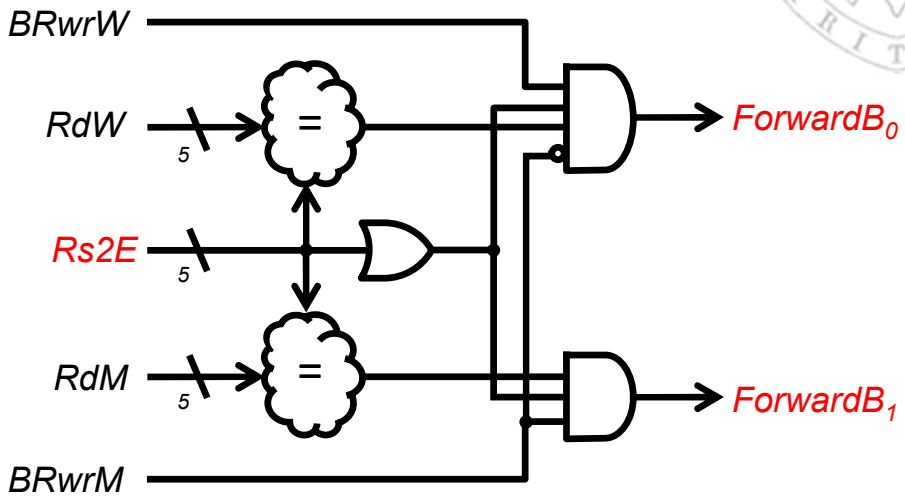
Truth table

Rs2E ≠ 0	BRwrM	BRwrW	Rs2E = RdM	Rs2E = RdW	ForwardB
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

Pipelined processor

Forwarding unit design (iv)





Truth table

Rs1E ≠ 0	BRwrM	BRwrW	Rs1E = RdM	Rs1E = RdW	ForwardA
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

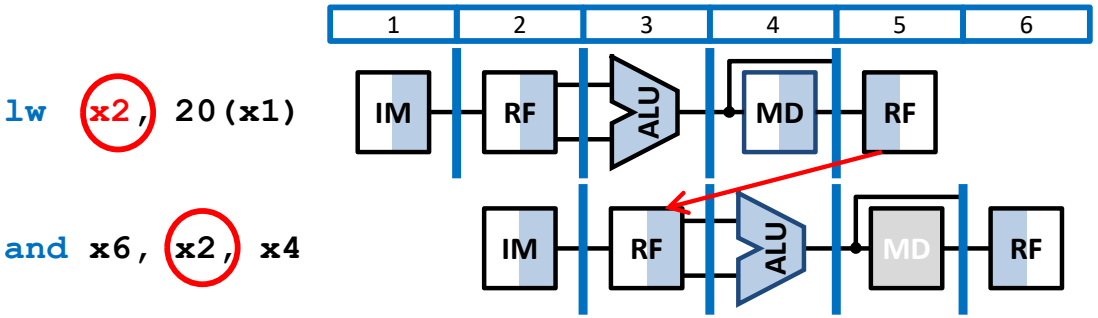
Truth table

Rs2E ≠ 0	BRwrM	BRwrW	Rs2E = RdM	Rs2E = RdW	ForwardB
0	X	X	X	X	00 (no forwarding)
1	0	1	X	0	00 (no forwarding)
1	0	1	X	1	01 (WB forwarding)
1	1	X	0	X	00 (no forwarding)
1	1	X	1	X	10 (MEM forwarding)

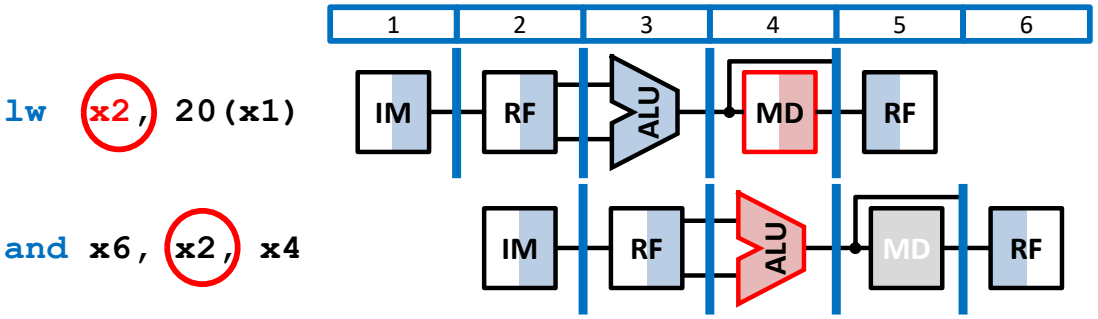
Data hazards

HW solution: `lw` hazard

- There is a kind of data hazard that requires a special treatment:
  - When a **`lw`** instruction loads a register that is read by the following instruction.



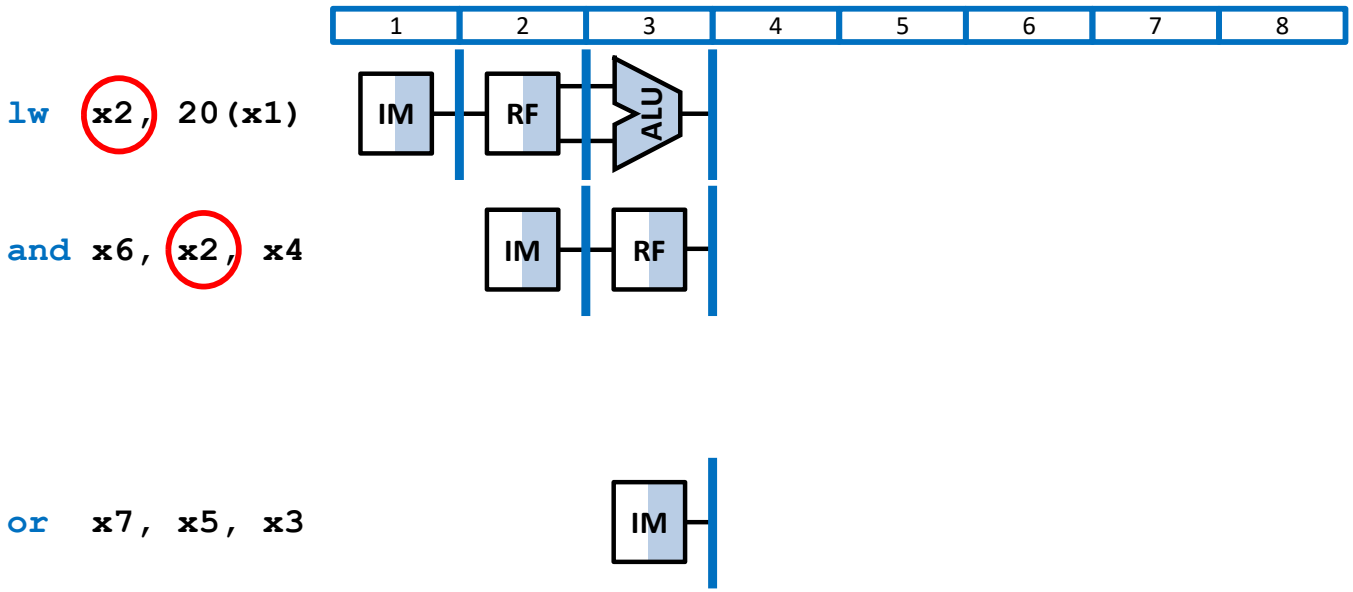
- The required data cannot be forwarded because:
  - The **`lw`** instruction reads the data from memory in cycle 4.
  - The following instruction needs that data in the same cycle.



Data hazards

HW solution: `lw` hazard, stalling (i)

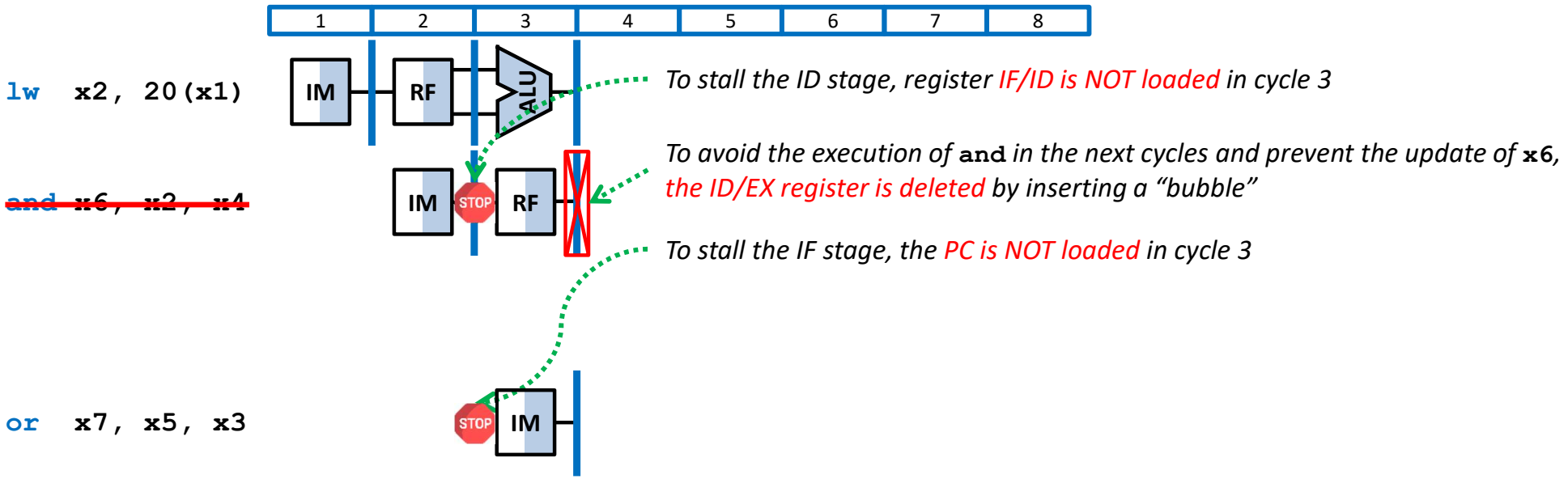
- The solution implies stalling the pipeline during one cycle in order to delay the instruction that requires the data, so that it can be forwarded.
  - Cycle 3: **`and`** (in ID), the hazard is detected.



Data hazards

HW solution: `lw` hazard, stalling (i)

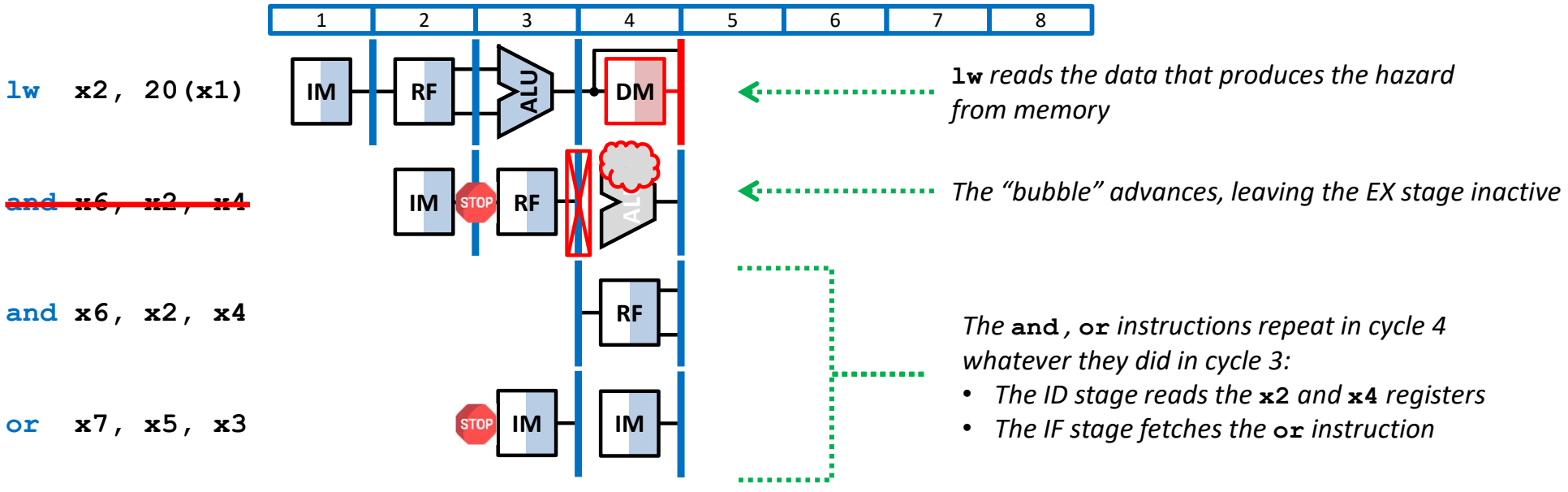
- The solution implies stalling the pipeline during one cycle in order to delay the instruction that requires the data, so that it can be forwarded.
  - Cycle 3: **`and`** (in ID), the hazard is detected. Instructions **`and`** , **`or`** are stalled and a nop "bubble" is inserted in the EX stage.



Data hazards

HW solution: `lw` hazard, stalling (i)

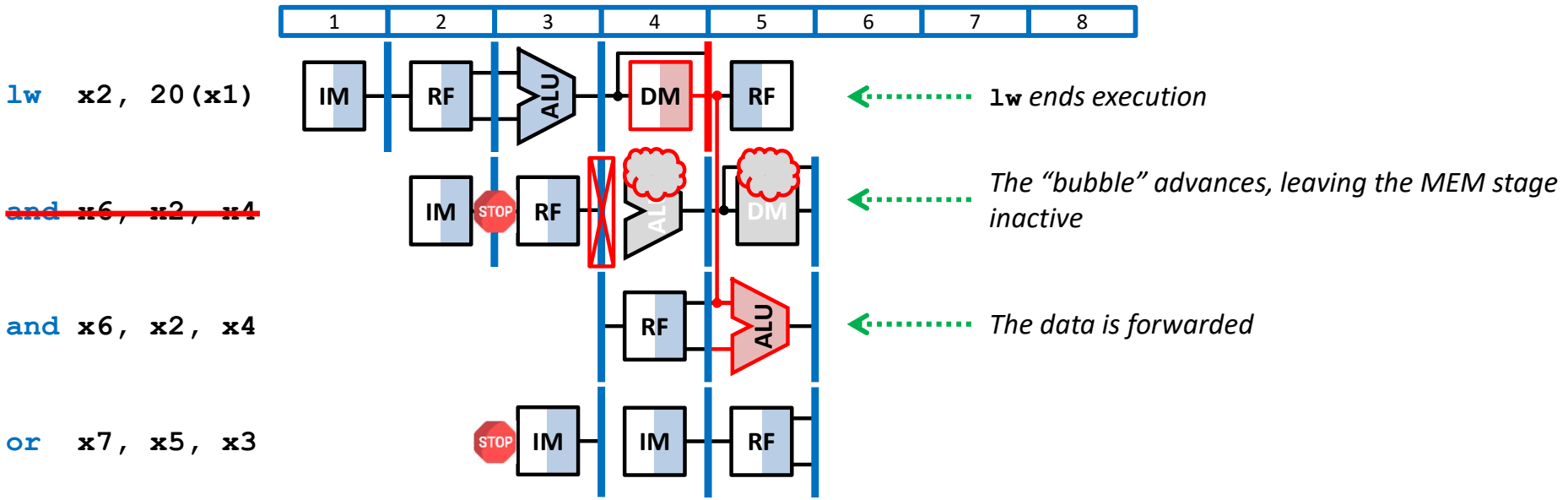
- The solution implies stalling the pipeline during one cycle in order to delay the instruction that requires the data, so that it can be forwarded.
  - Cycle 3: **`and`** (in ID), the hazard is detected. Instructions **`and`** , **`or`** are stalled and a nop "bubble" is inserted in the EX stage.
  - Cycle 4: **`lw`** reads the data from memory; **`and`** , **`or`** resume.



Data hazards

HW solution: `lw` hazard, stalling (i)

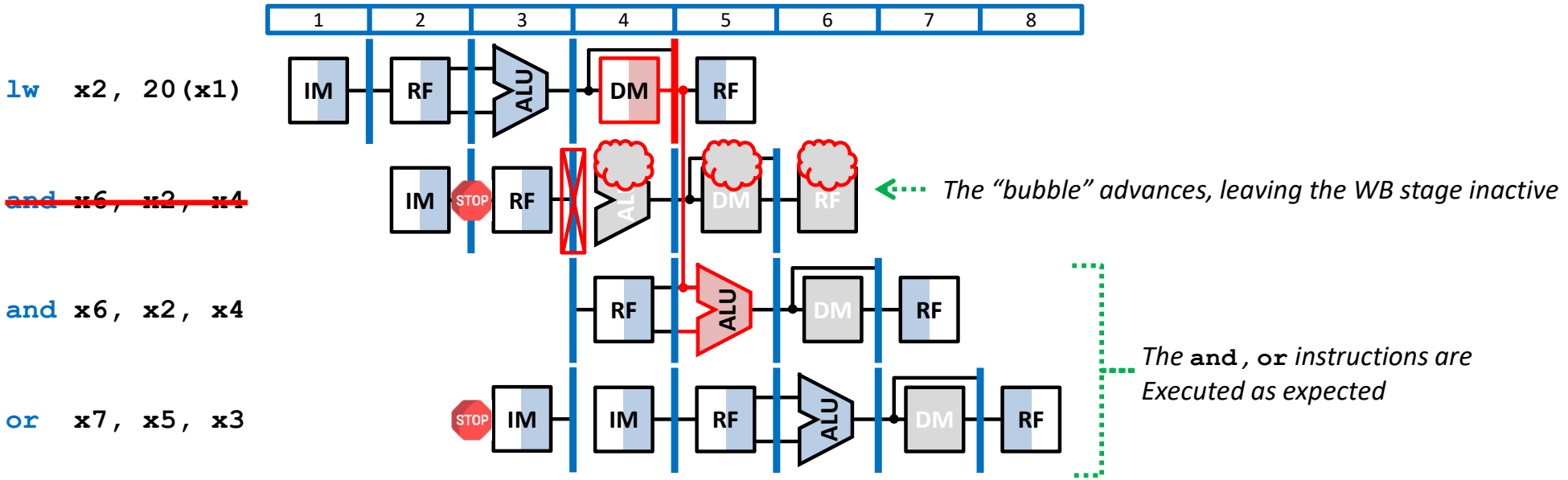
- The solution implies stalling the pipeline during one cycle in order to delay the instruction that requires the data, so that it can be forwarded.
  - Cycle 3: **`and`** (in ID), the hazard is detected. Instructions **`and`** , **`or`** are stalled and a nop "bubble" is inserted in the EX stage.
  - Cycle 4: **`lw`** reads the data from memory; **`and`** , **`or`** resume.
  - Cycle 5: the data is forwarded from the WB stage of **`lw`** to the EX stage of **`and`**.



Data hazards

HW solution: `lw` hazard, stalling (i)

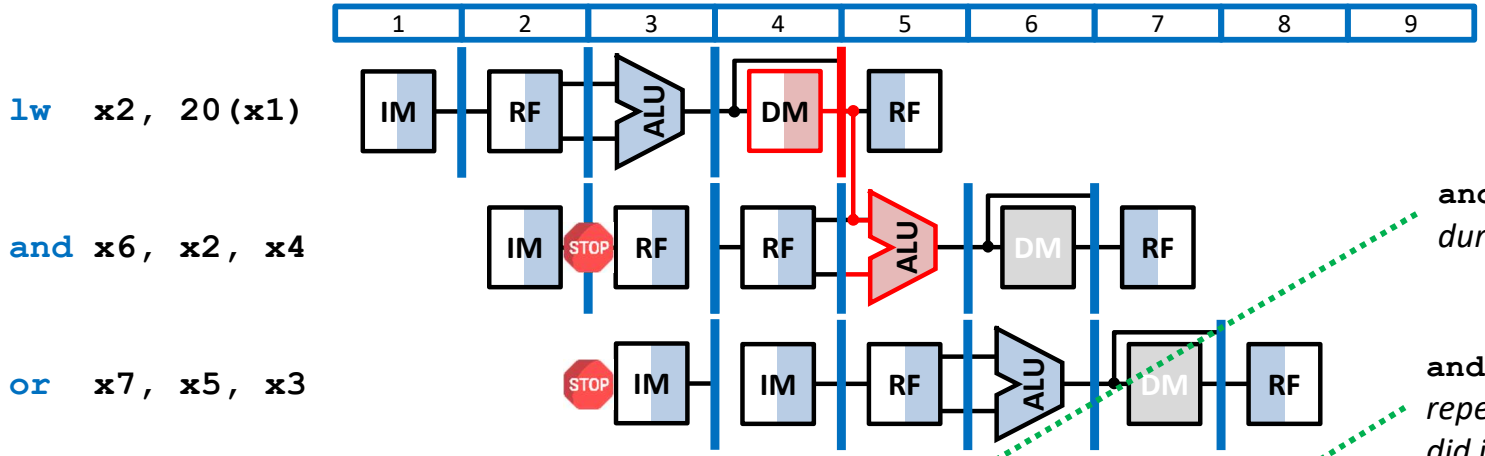
- The solution implies stalling the pipeline during one cycle in order to delay the instruction that requires the data, so that it can be forwarded.
  - Cycle 3: **`and`** (in ID), the hazard is detected. Instructions **`and`** , **`or`** are stalled and a nop "bubble" is inserted in the EX stage.
  - Cycle 4: **`lw`** reads the data from memory; **`and`** , **`or`** resume.
  - Cycle 5: the data is forwarded from the WB stage of **`lw`** to the EX stage of **`and`**.
  - Next cycles: the pipeline behaves as expected.
  - There is a one cycle penalty due to the **`lw`** hazard.



Data hazards

HW solution: `lw` hazard, stalling (ii)

- In the simplified execution diagrams, stalls are indicated marking the stages that are stalled ("bubbles" are implicit):



`and`, `or` are stalled during one cycle

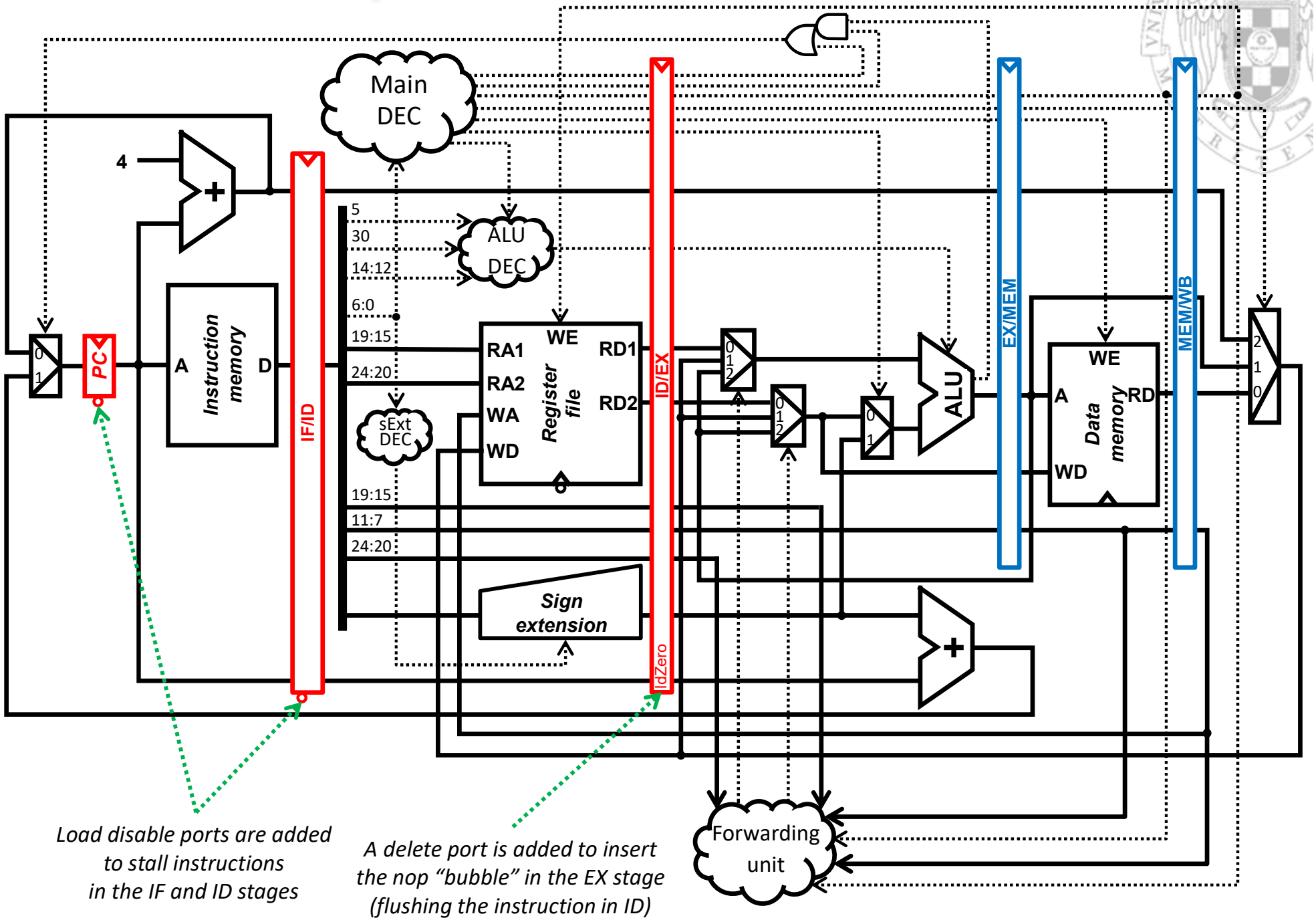
`and`, `or` resume repeating whatever they did in the previous cycle

	1	2	3	4	5	6	7	8	9
`lw x2, 20(x1)`	IF	ID	EX	M	WB				
`and x6, x2, x4`		IF	IDs	ID	EX x2	M	WB		
`or x7, x5, x3`			IFs	IF	ID	EX	M	WB	
`add x8, x2, x2`					IF	ID	EX	M	WB

The next instruction is fetched after resuming `and`, `or`

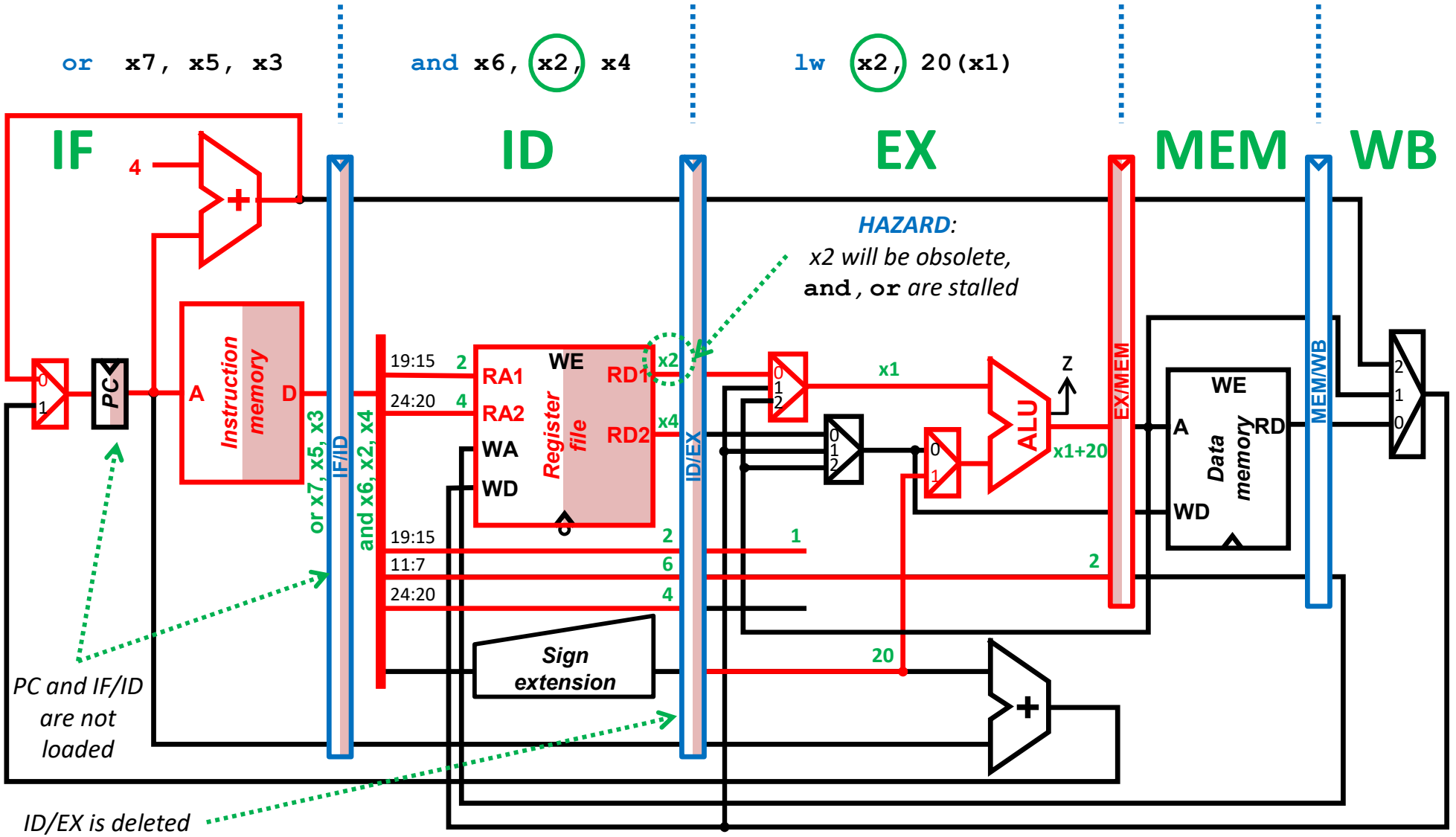
Pipelined processor

+ `lw` hazard: data path



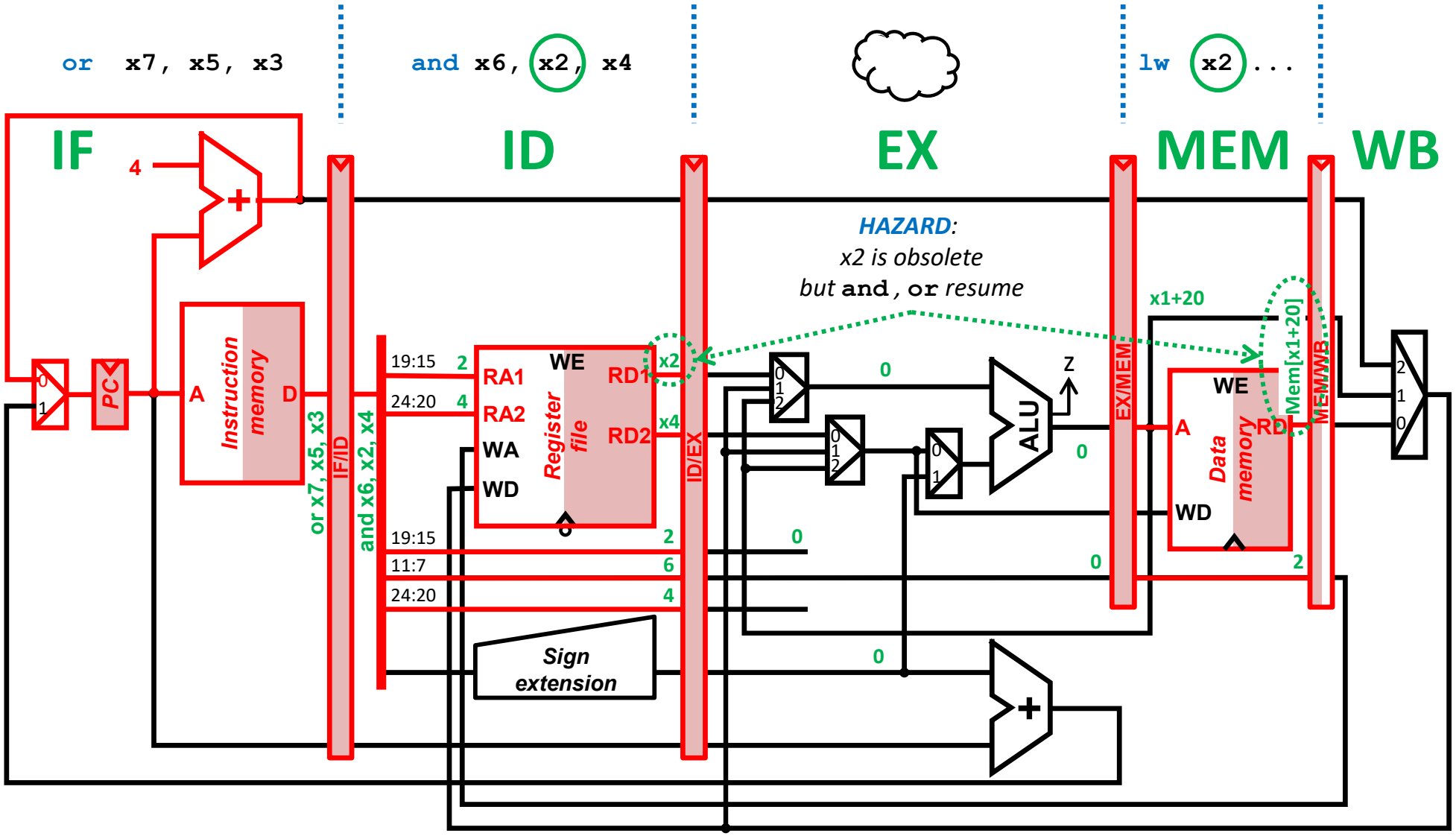
Pipelined processor

Stall simulation: 3rd. cycle



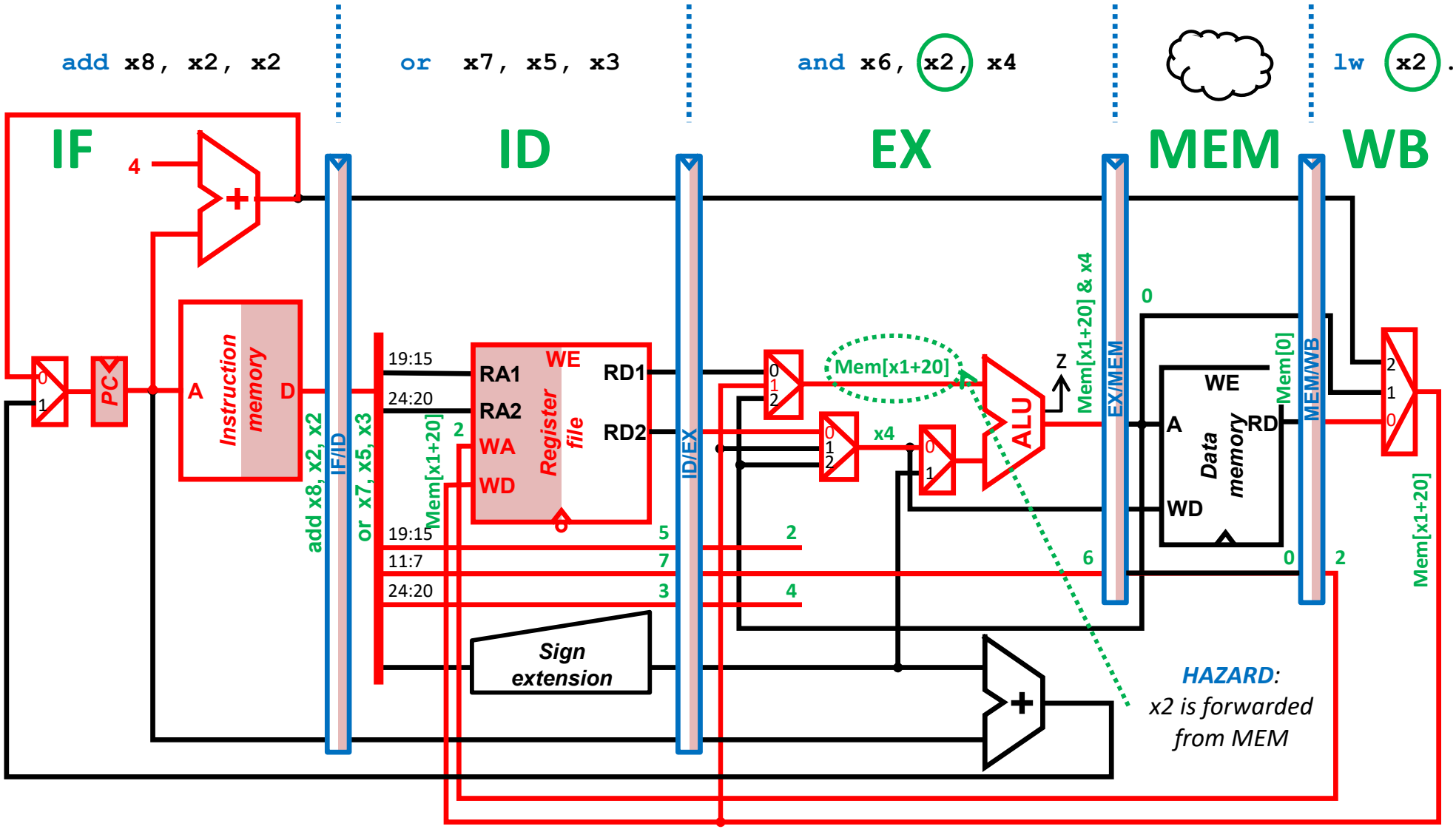
Pipelined processor

Stall simulation : 4th. cycle



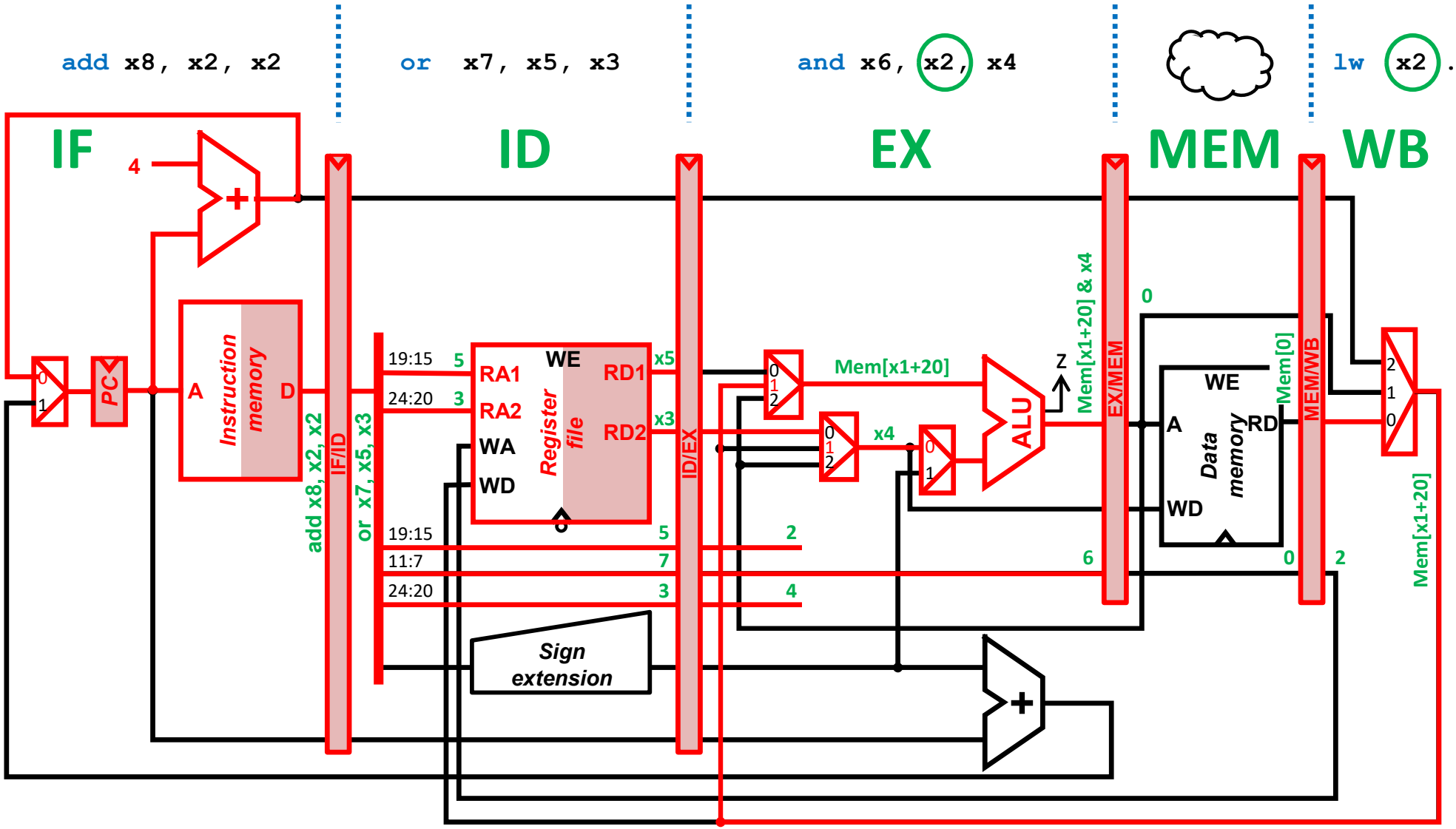
Pipelined processor

Stall simulation : 5th. cycle (1st. half)



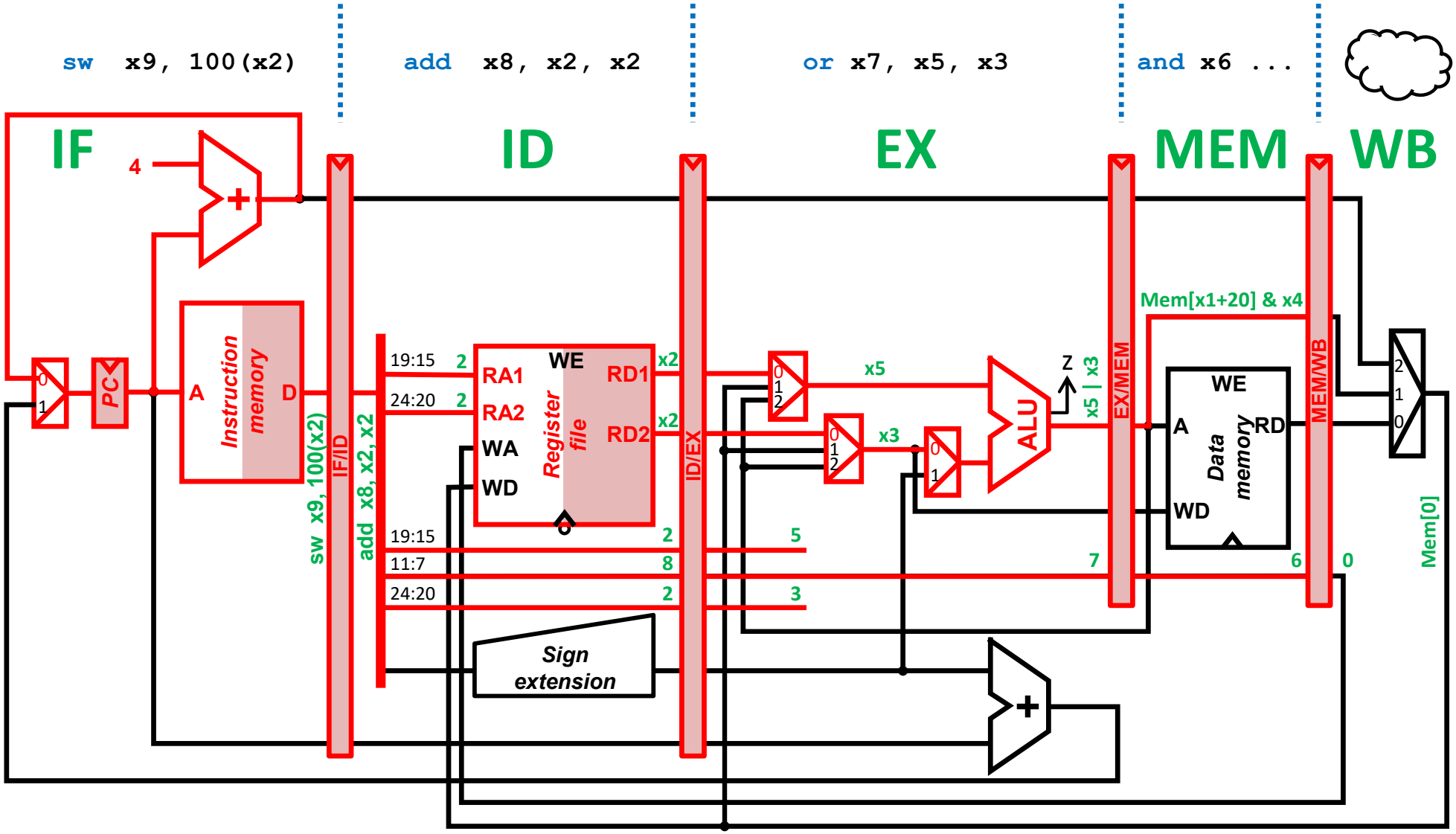
Pipelined processor

Stall simulation : 5th. cycle (2nd. half)



Pipelined processor

Stall simulation : 6th. cycle



Pipelined processor

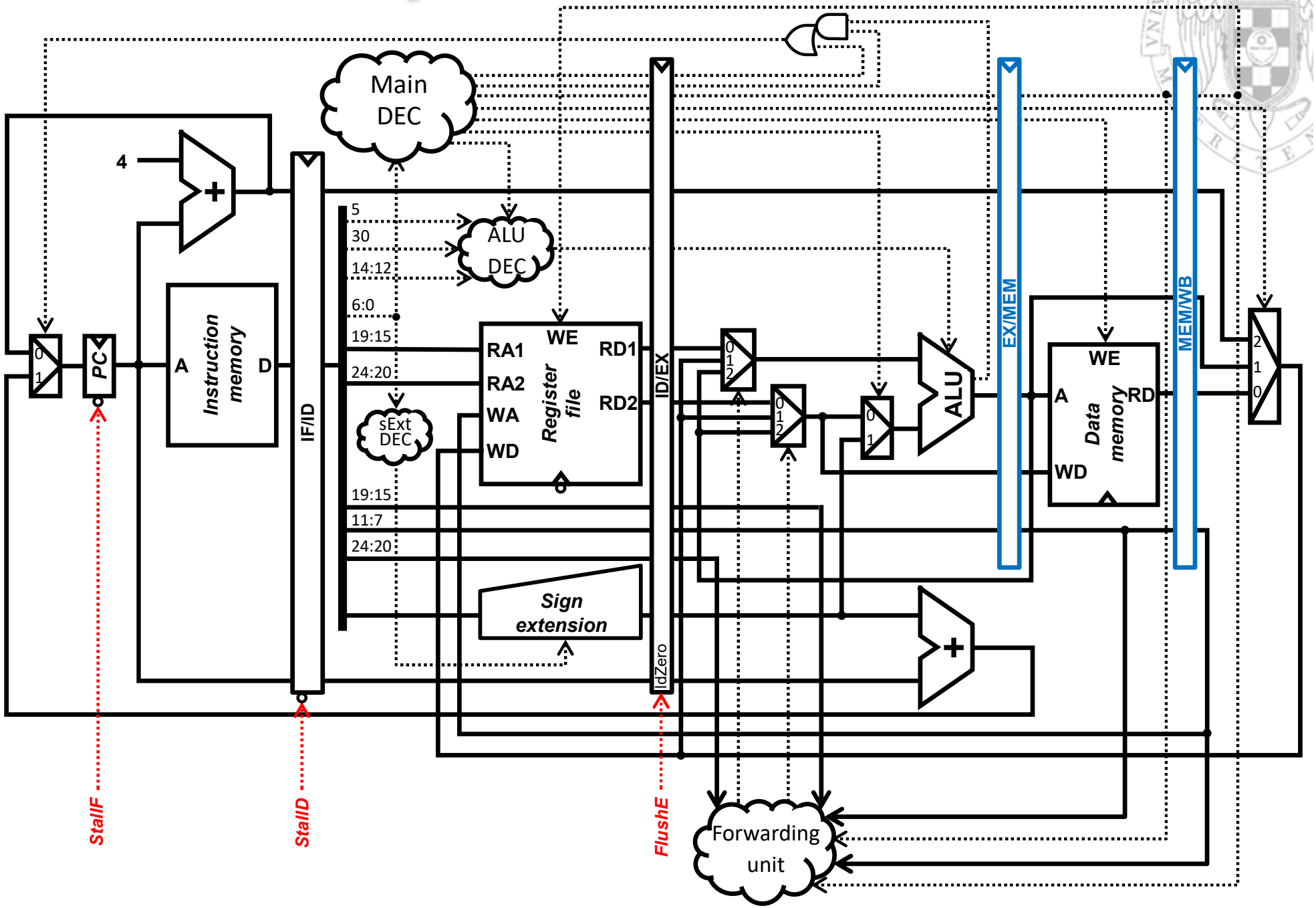
Hazard unit

- The hazard unit is a combinational circuit that determines if the IF and ID stages of the pipeline have to be stalled, controlling whether:
  - The PC and the IF/ID pipeline register have to be loaded or not.
  - The ID/EX pipeline register must be deleted or not.

- In order to behave correctly, it must know:
  - If there is a `lw` instruction in the EX stage.
    - Checking if ResSrcE = 0 and BRwrE = 1 (only `lw` meets this)
  - RdE: number of the destination register of the `lw` instruction in the EX stage.
  - Rs1D: number of source register 1 of the instruction in the ID stage.
  - Rs2D: number of source register 2 of the instruction in the ID stage.

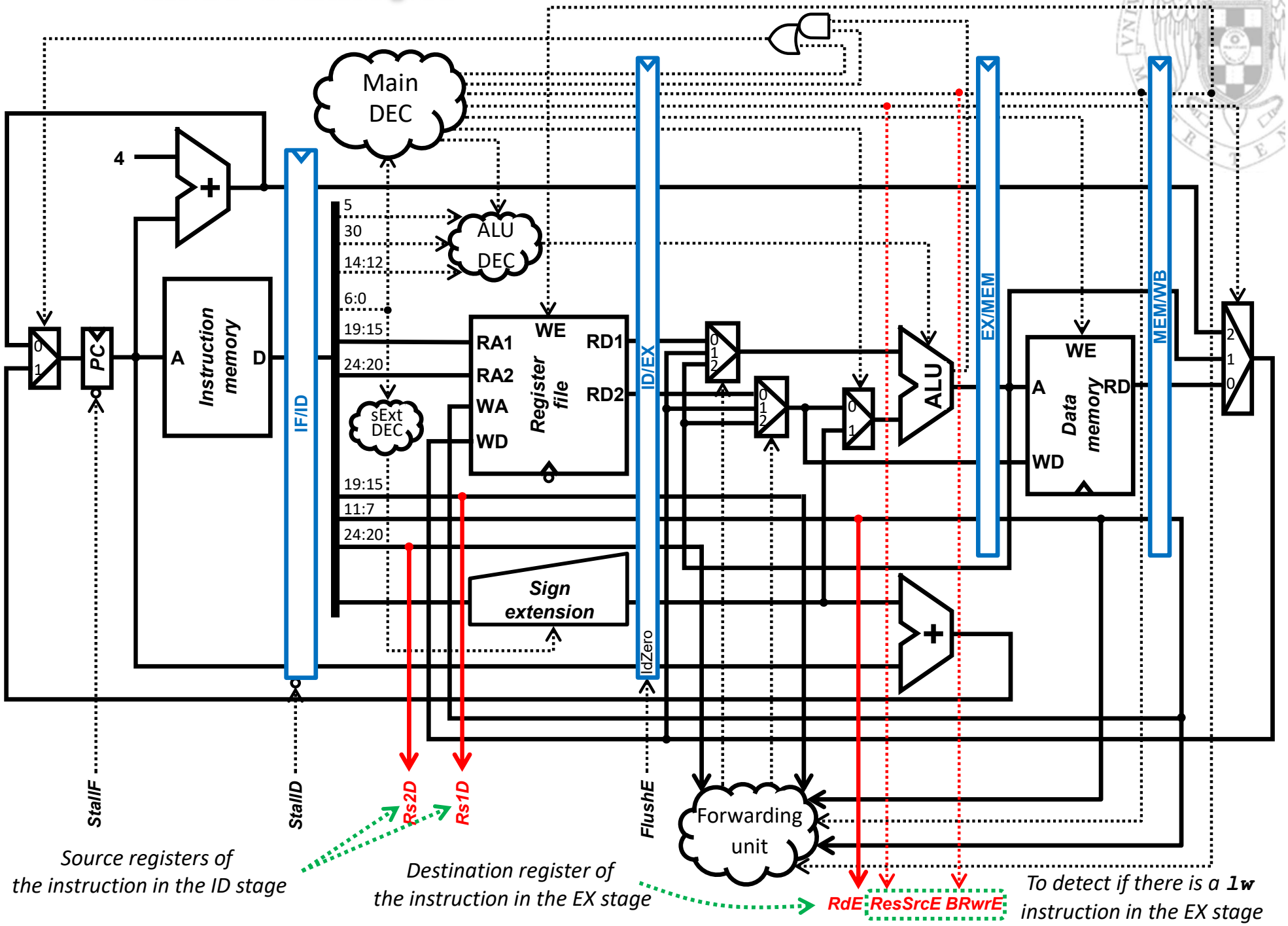
Pipelined processor

+ Hazard unit: control signals



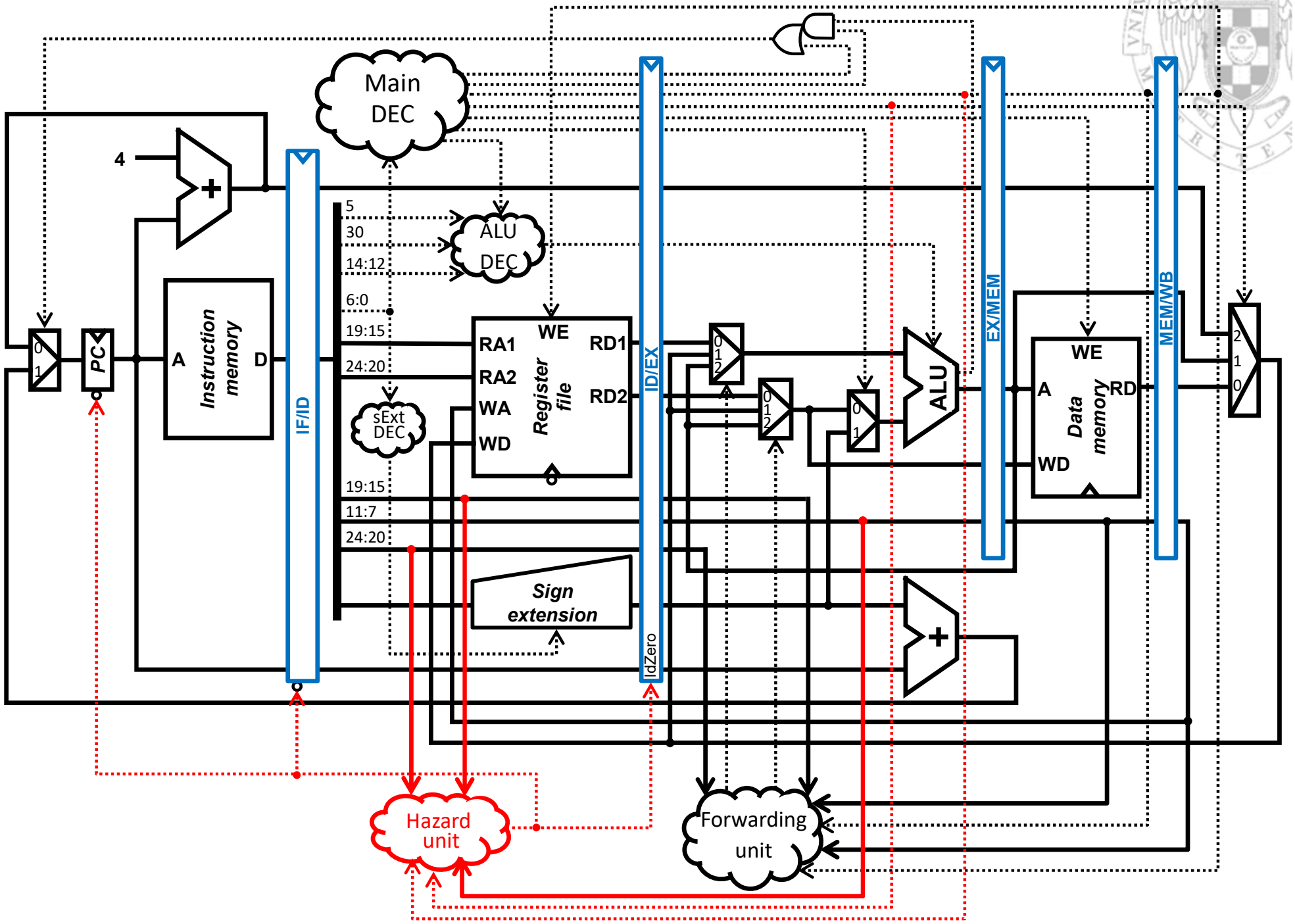
Pipelined processor

+ Hazard unit: status signals



Pipelined processor

+ Hazard unit



Pipelined processor

Hazard unit design (i)

- The pipeline must be stalled during one cycle due to a **`lw`** hazard if:
  - There is a **`lw`** instruction in the EX stage
    - Checking if ResSrcE = 0 and BRwrE = 1 (only **`lw`** meets this)

Stall ← *if* ((ResSrcE = 0) & BRwrE) *then* (1) ⇠ *Stall the pipeline*

Pipelined processor

Hazard unit design (i)

- The pipeline must be stalled during one cycle due to a **`lw`** hazard if:
  - There is a **`lw`** instruction in the EX stage
    - Checking if ResSrcE = 0 and BRwrE = 1 (only **`lw`** meets this)
  - The destination register of the EX stage (RdE) coincides with one of the source registers of the ID stage (Rs1D and/or Rs2D).

Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1) ⇠ *Stall the pipeline*

Pipelined processor

Hazard unit design (i)

- The pipeline must be stalled during one cycle due to a **`lw`** hazard if:
  - There is a **`lw`** instruction in the EX stage
    - Checking if ResSrcE = 0 and BRwrE = 1 (only **`lw`** meets this)
  - The destination register of the EX stage (RdE) coincides with one of the source registers of the ID stage (Rs1D and/or Rs2D).
- Otherwise, the pipeline is not stalled.

Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1) ⇠ *Stall the pipeline*
else (0) ⇠ *Do not stall the pipeline*

Pipelined processor

Hazard unit design (i)

- The pipeline must be stalled during one cycle due to a **`lw`** hazard if:
  - There is a **`lw`** instruction in the EX stage
    - Checking if ResSrcE = 0 and BRwrE = 1 (only **`lw`** meets this)
  - The destination register of the EX stage (RdE) coincides with one of the source registers of the ID stage (Rs1D and/or Rs2D).
- Otherwise, the pipeline is not stalled.
- When a stall happens:
  - Disable the load of the PC and the ID/IF pipeline register.

Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1) ⟵ *Stall the pipeline*
else (0) ⟵ *Do not stall the pipeline*
StallF ← Stall
StallD ← Stall

Pipelined processor

Hazard unit design (i)

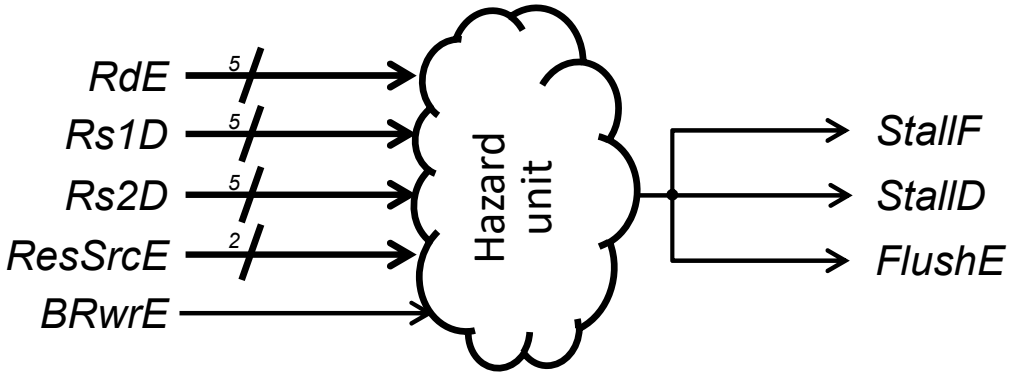
- The pipeline must be stalled during one cycle due to a **`lw`** hazard if:
  - There is a **`lw`** instruction in the EX stage
    - Checking if ResSrcE = 0 and BRwrE = 1 (only **`lw`** meets this)
  - The destination register of the EX stage (RdE) coincides with one of the source registers of the ID stage (Rs1D and/or Rs2D).
- Otherwise, the pipeline is not stalled.
- When a stall happens:
  - Disable the load of the PC and the ID/IF pipeline register.
  - Delete the ID/EX pipeline register.

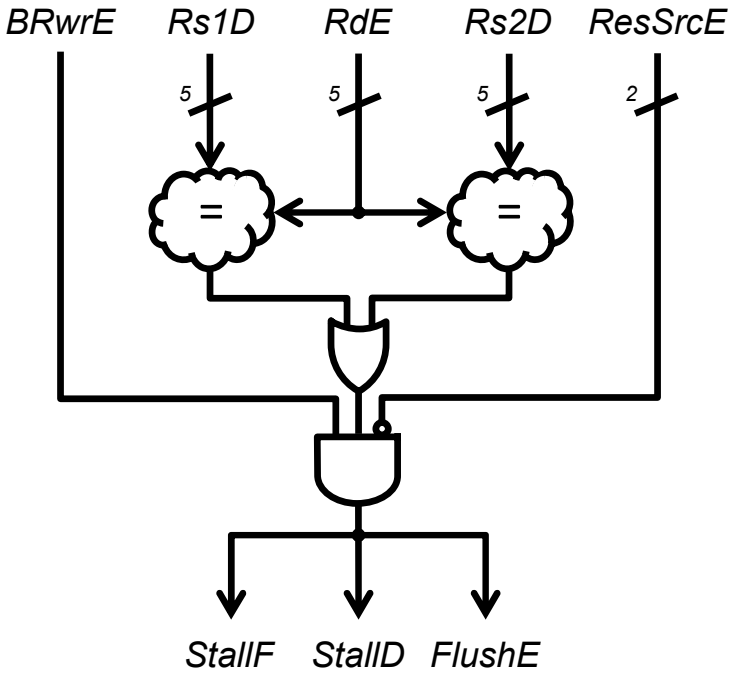
Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1) ⇠ *Stall the pipeline*
else (0) ⇠ *Do not stall the pipeline*
StallF ← Stall
StallD ← Stall
FlushE ← Stall

Pipelined processor

Hazard unit design (ii)

Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1)
else (0)
StallF ← Stall
StallD ← Stall
FlushE ← Stall



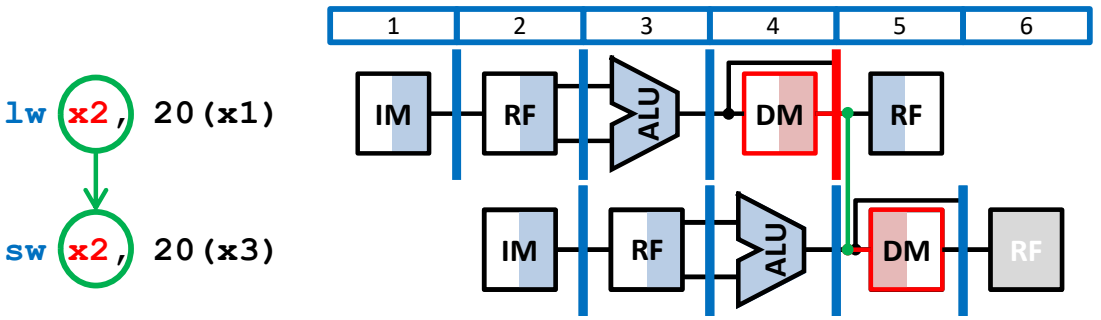


Data hazards

HW solution: additional optimizations (i)

- The proposed solution sometimes performs unnecessary stalls.

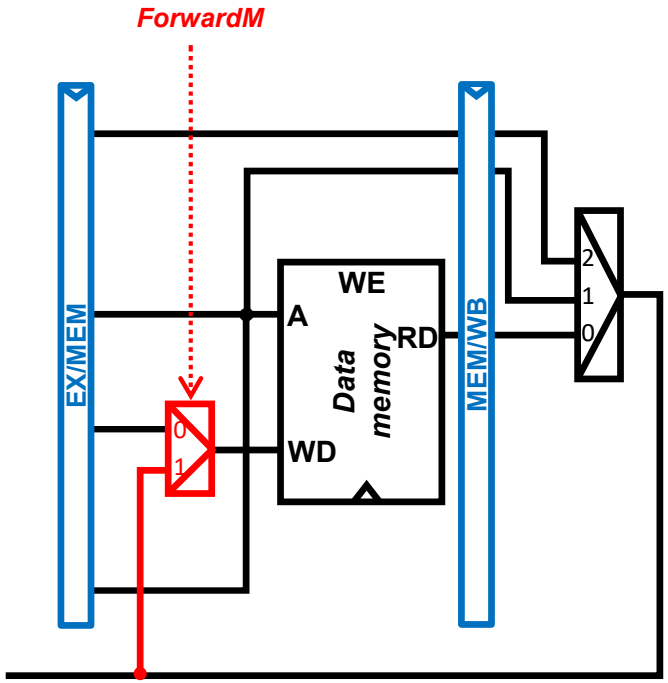
- When `x0` is the destination register of the memory load.
  - Since instructions as `lw x0,20(x1)` are meaningless, it is not worth adding the hardware logic to handle these cases.
- When a `lw` instruction is followed by a `sw` instruction that stores the register loaded from memory by the former.
  - It is a more common case because it is used e.g. to copy arrays.
  - The data, available since cycle 5, could be forwarded without stalling.



Data hazards

HW solution: additional optimizations (ii)

- To avoid the penalty in the **`lw`** → **`sw`** case, it would be enough to:
  - Add a MUX at the memory data input, so that forwarding could be performed from the WB stage to the MEM stage.
  - Redesign the forwarding and hazard units.



Forwarding happens if: there is a **`lw`** *instruction in the WB stage, a* **`sw`** *instruction in MEM and the destination register of the former coincides with the source register of the latter*

ForwardM ← *if* ((ResSrcW = 0) & BRwrW
& MemWrM
& (RdW = Rs2M)) *then* (1)
else (0)

Stall ← *if* ((ResSrcE = 0) & BRwrE
& ((Rs1D = RdE) | (Rs2D = RdE))
& !MemWrD) *then* (1)
else (0)

Do not stall if there is a **`sw`** *instruction in the ID stage*

Data hazards

HW+SW solution: code reordering

- Given an assembly program, stalls due to data hazards by **`lw`** instructions are unavoidable by HW.
  - But they can be avoided by reordering the code, so that a **`lw`** instruction is never followed by another one that uses the loaded register.
  - This is one of the optimizations applied by the compilers.

```
...
int a, b, c;
int d, e;
...
c = a + b;
e = d + b;
...
```

a → x1 *d → x4*
b → x2 *e → x5*
c → x3

Assignment of variables

direct compilation

```
...
lw   x1, 0(x31)
lw   x2, 4(x31)
add  x3, x1, x2
sw   x3, 8(x31)
lw   x4, 12(x31)
add  x5, x4, x2
sw   x5, 16(x31)
...
```

2 stalls

optimized compilation

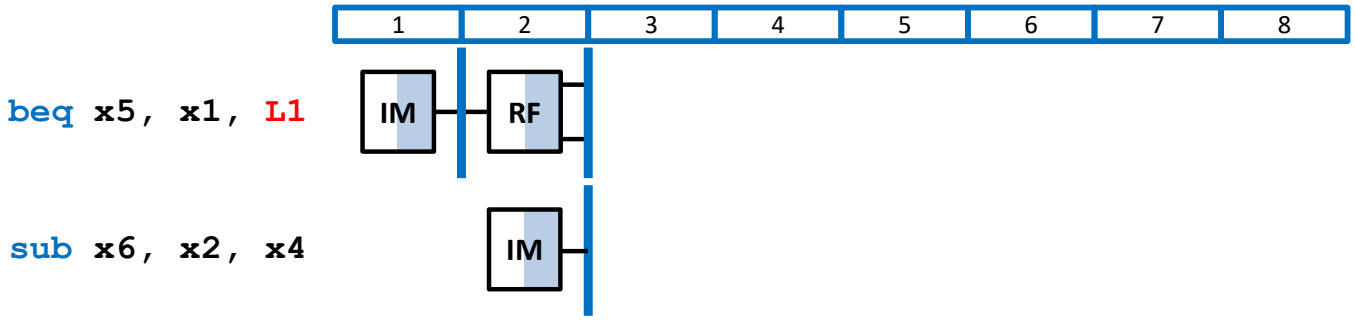
```
...
lw   x1, 0(x31)
lw   x2, 4(x31)
lw   x4, 12(x31)
add  x3, x1, x2
sw   x3, 8(x31)
add  x5, x4, x2
sw   x5, 16(x31)
...
```

0 stalls

Control hazards

HW solution: stalling

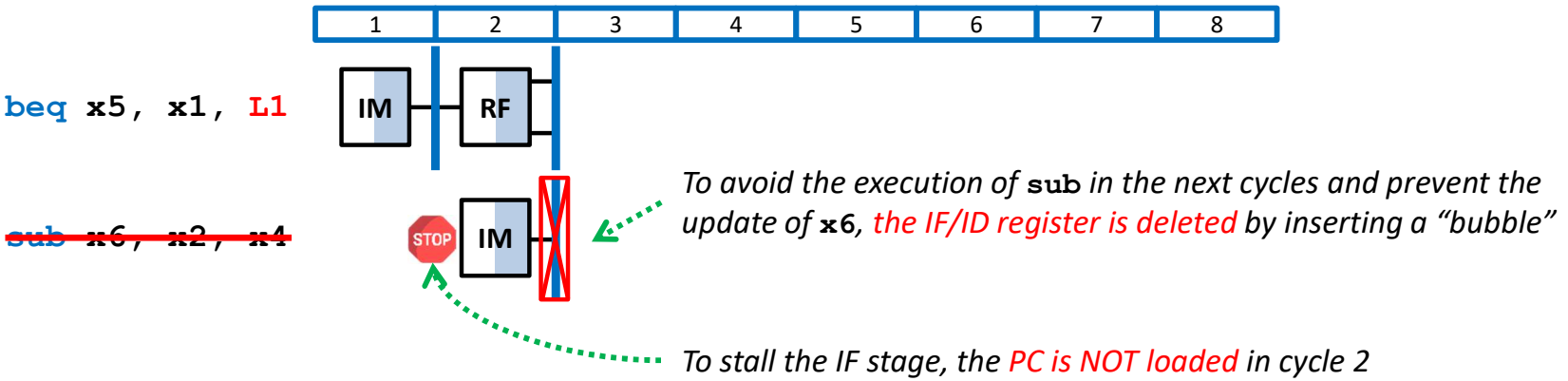
- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **`beq`** (in ID), the control hazard is detected.



Control hazards

HW solution: stalling

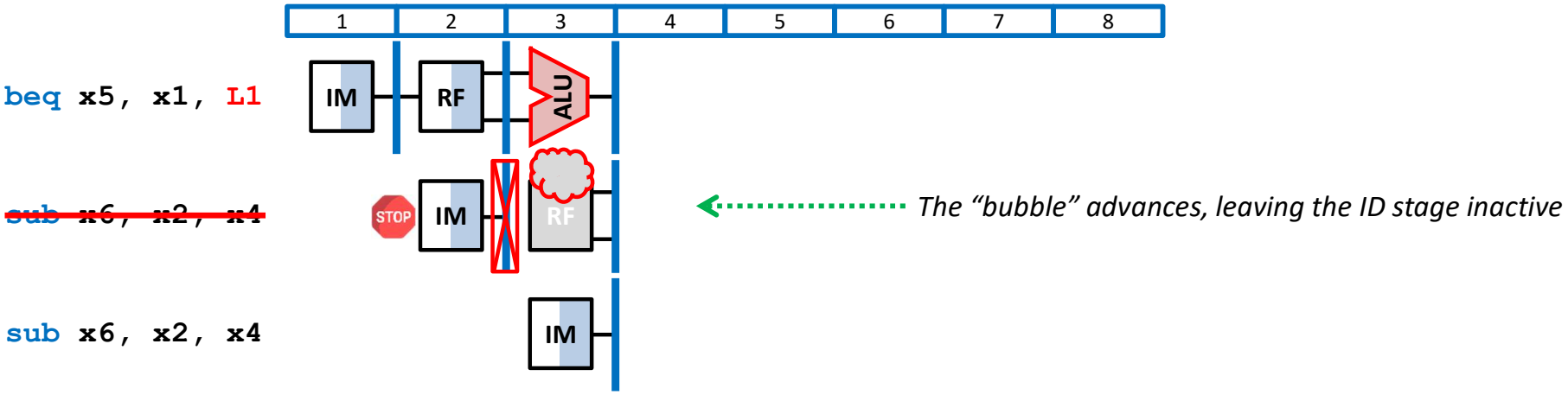
- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **beq** (in ID), the control hazard is detected. The **sub** instruction is stalled and a nop "bubble" is inserted in the ID stage.



Control hazards

HW solution: stalling

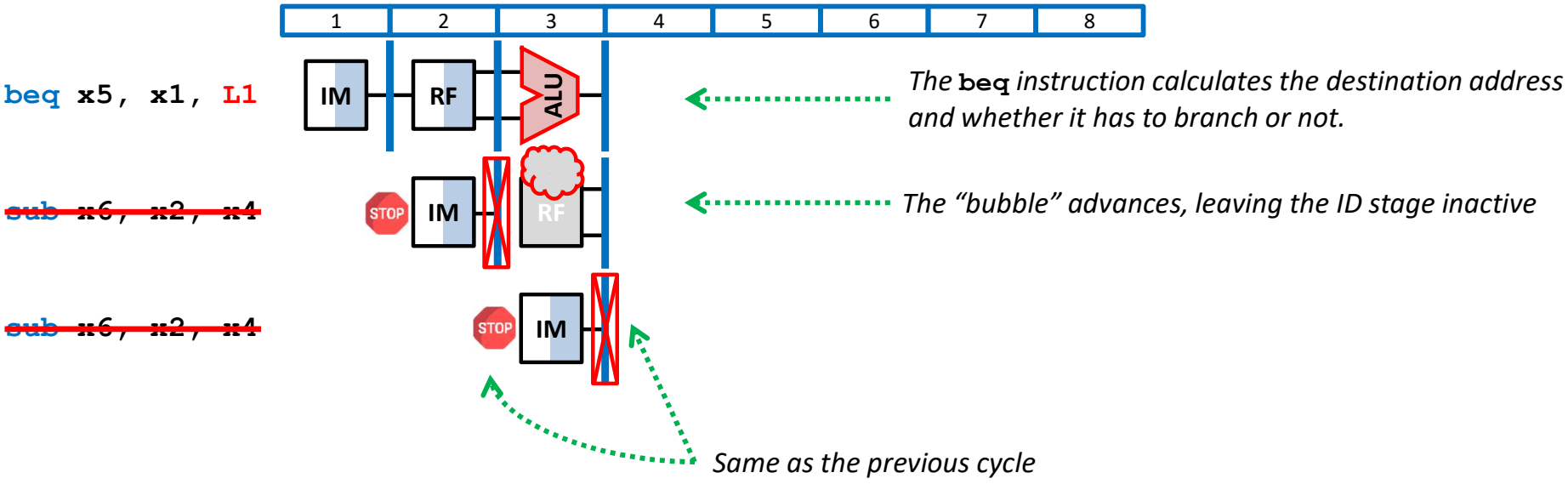
- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **`beq`** (in ID), the control hazard is detected. The **`sub`** instruction is stalled and a nop "bubble" is inserted in the ID stage.
  - Cycle 3: **`beq`** (in EX) decides the branch, but the hazard continues.



Control hazards

HW solution: stalling

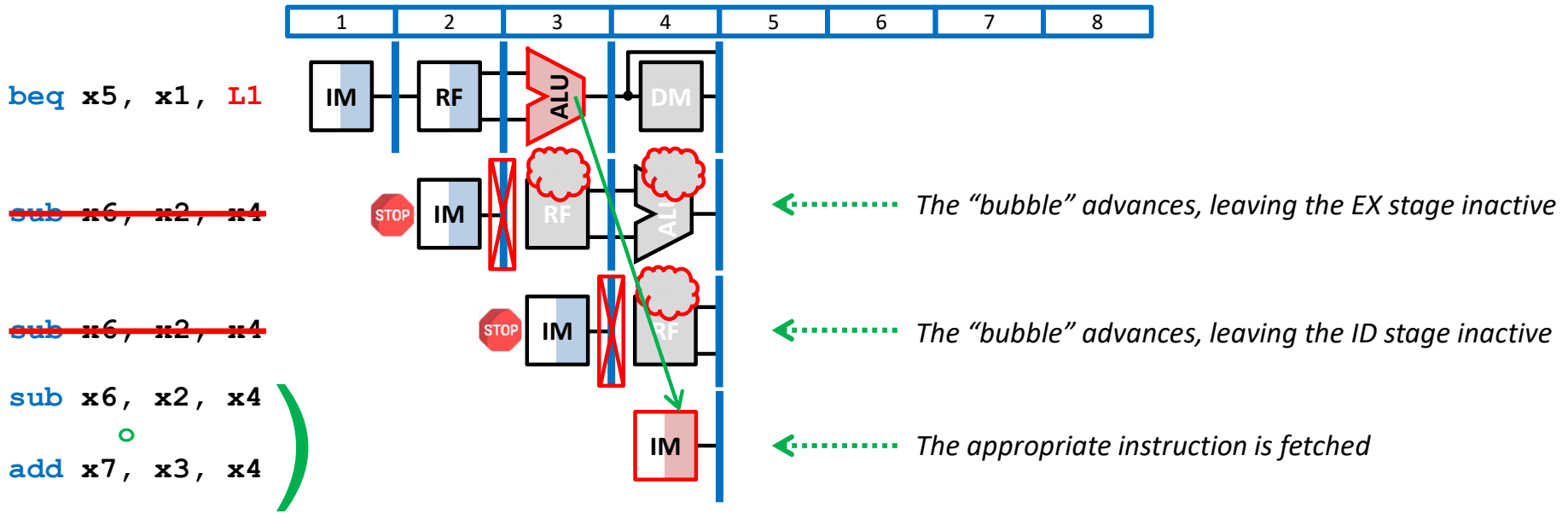
- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **beq** (in ID), the control hazard is detected. The **sub** instruction is stalled and a nop "bubble" is inserted in the ID stage.
  - Cycle 3: **beq** (in EX) decides the branch, but the hazard continues. The **sub** instruction remains stalled and another "bubble" is inserted.



Control hazards

HW solution: stalling

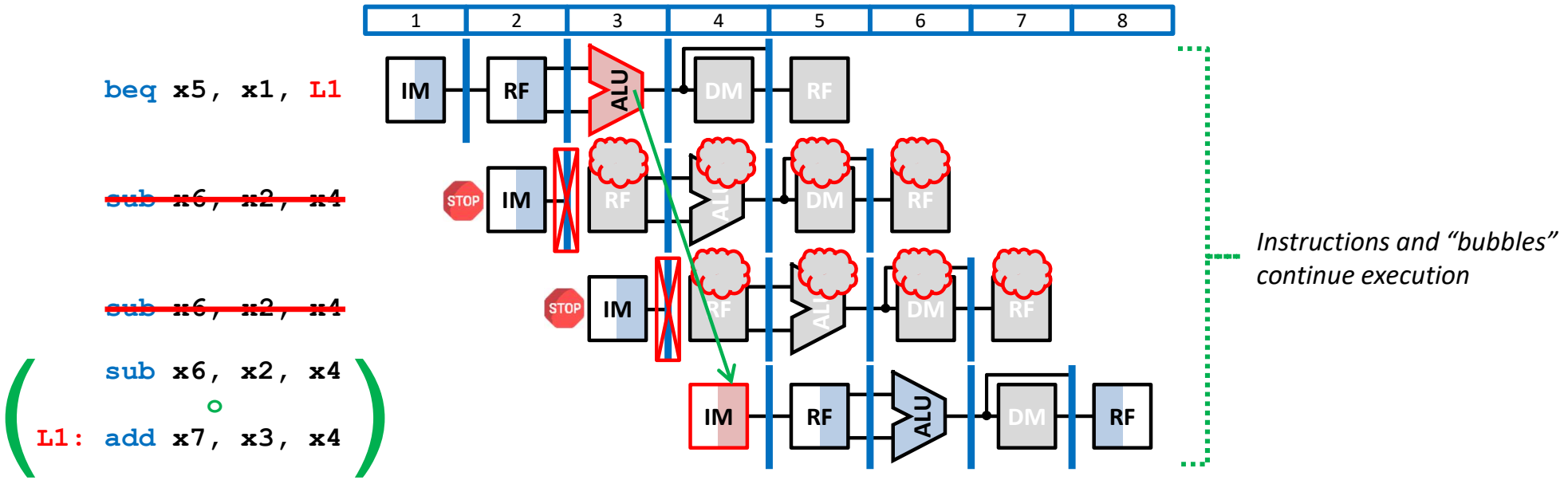
- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **beq** (in ID), the control hazard is detected. The **sub** instruction is stalled and a nop "bubble" is inserted in the ID stage.
  - Cycle 3: **beq** (in EX) decides the branch, but the hazard continues. The **sub** instruction remains stalled and another "bubble" is inserted.
  - Cycle 4: **beq** (in MEM), the appropriate instruction is fetched.



Control hazards

HW solution: stalling

- One solution consists in stalling the pipeline during 2 cycles to delay fetching new instructions until the branch is decided.
  - Cycle 2: **beq** (in ID), the control hazard is detected. The **sub** instruction is stalled and a nop "bubble" is inserted in the ID stage.
  - Cycle 3: **beq** (in EX) decides the branch, but the hazard continues. The **sub** instruction remains stalled and another "bubble" is inserted.
  - Cycle 4: **beq** (in MEM), the appropriate instruction is fetched.
  - There is a penalty of two cycles per branch instruction.



Control hazards

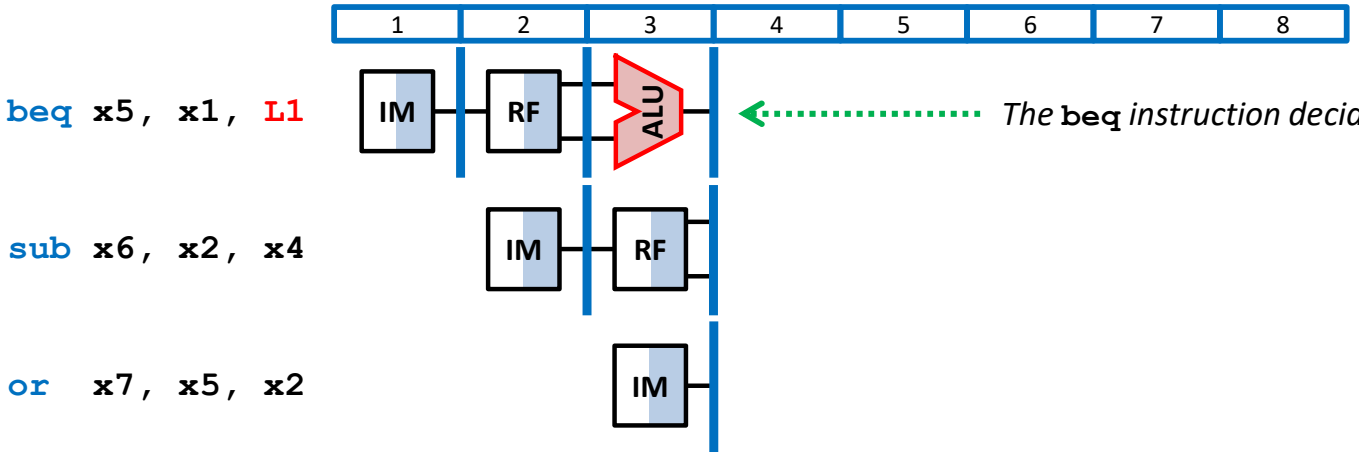
HW solution: branch prediction (i)

- There is a better solution for `beq` instructions, which consists in predicting that the branch will not be taken.
  - The `beq` instruction and the following ones are fetched as normal.
  - When the branch address/decision is known (`beq` will be in the EX stage):
    - If the branch is not taken, do nothing.
    - If the branch is taken, the last 2 fetched instructions are flushed, inserting nop "bubbles" in the ID and EX stages.
  - In the next cycle (`beq` will be in MEM), the appropriate instruction is fetched.
  - There is a penalty of two cycles per taken branch instruction. No penalty if the branch is not taken.

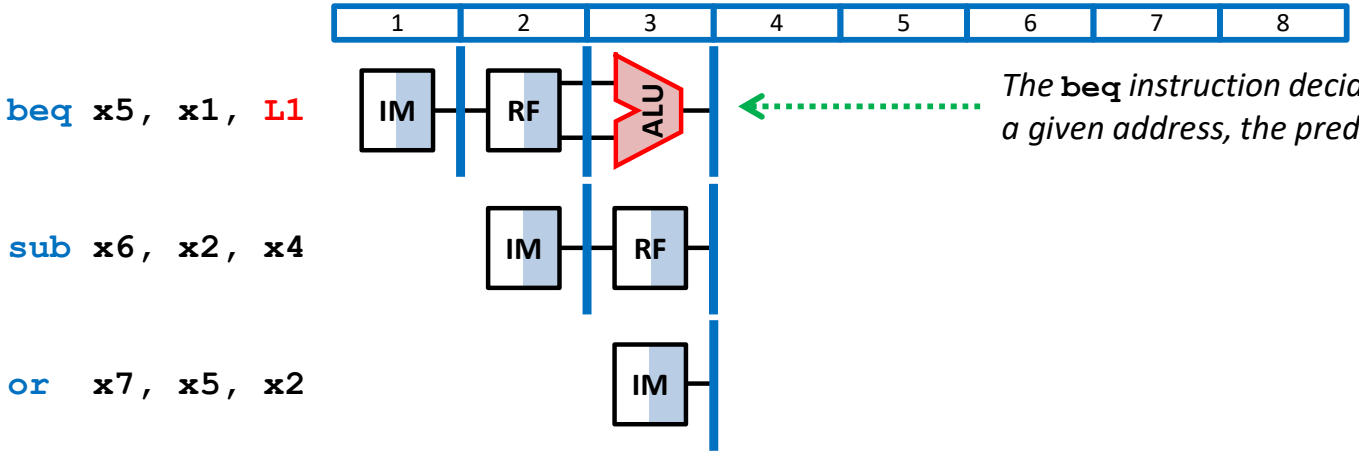
- The opposite operation, i.e. , to predict that the branch is taken, is much more complex since this also requires to predict the destination address.

Control hazards

HW solution: branch prediction (ii)



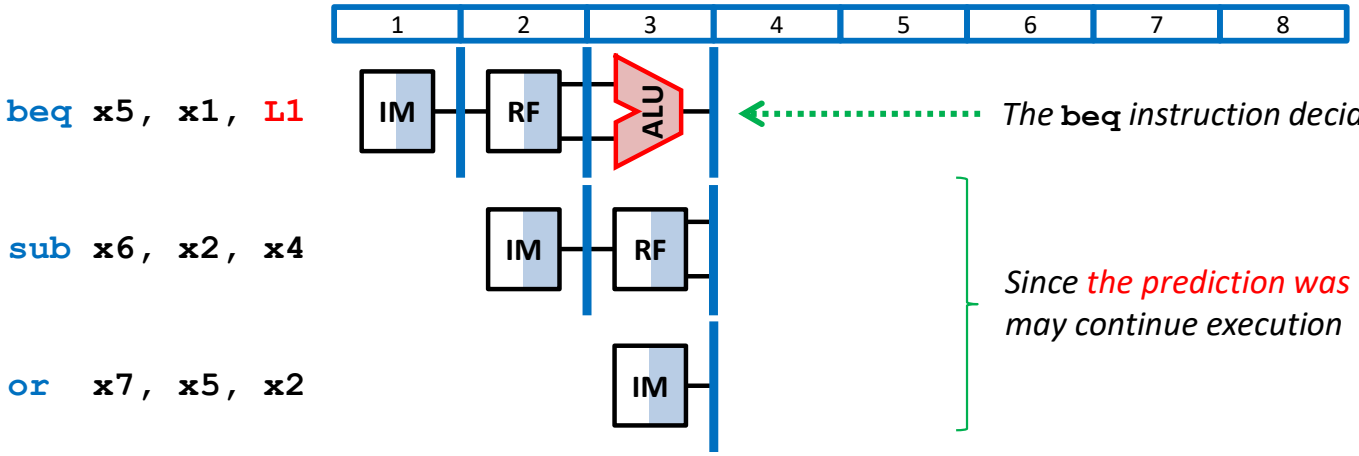
The **beq** *instruction decides that it does not have to branch*



The **beq** *instruction decides that it has to branch to a given address, the prediction was wrong*

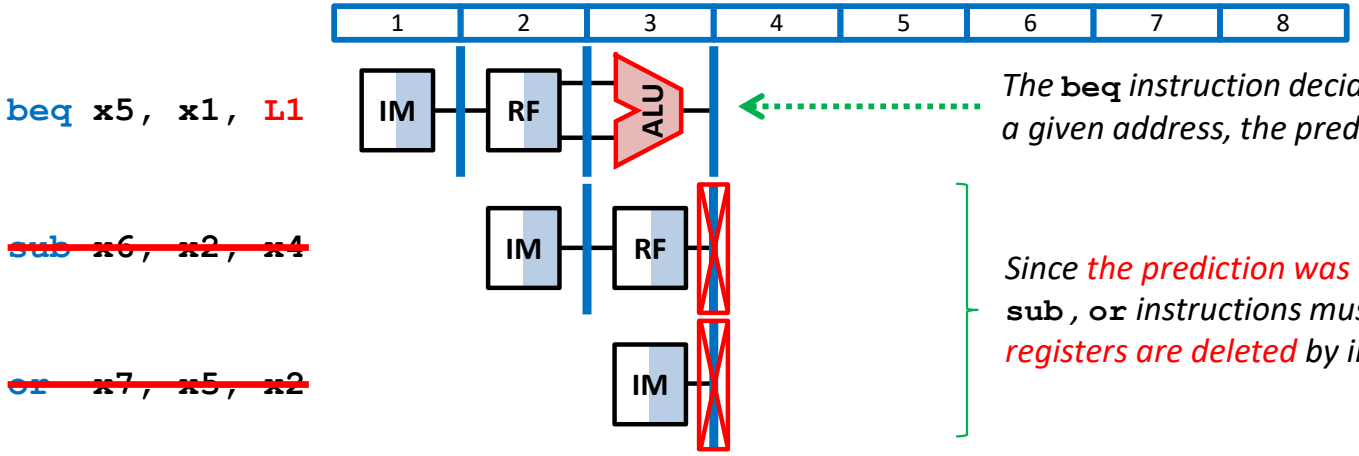
Control hazards

HW solution: branch prediction (ii)



The **beq** *instruction decides that it does not have to branch*

Since the prediction was right, the **sub** *,* **or** *instructions may continue execution*

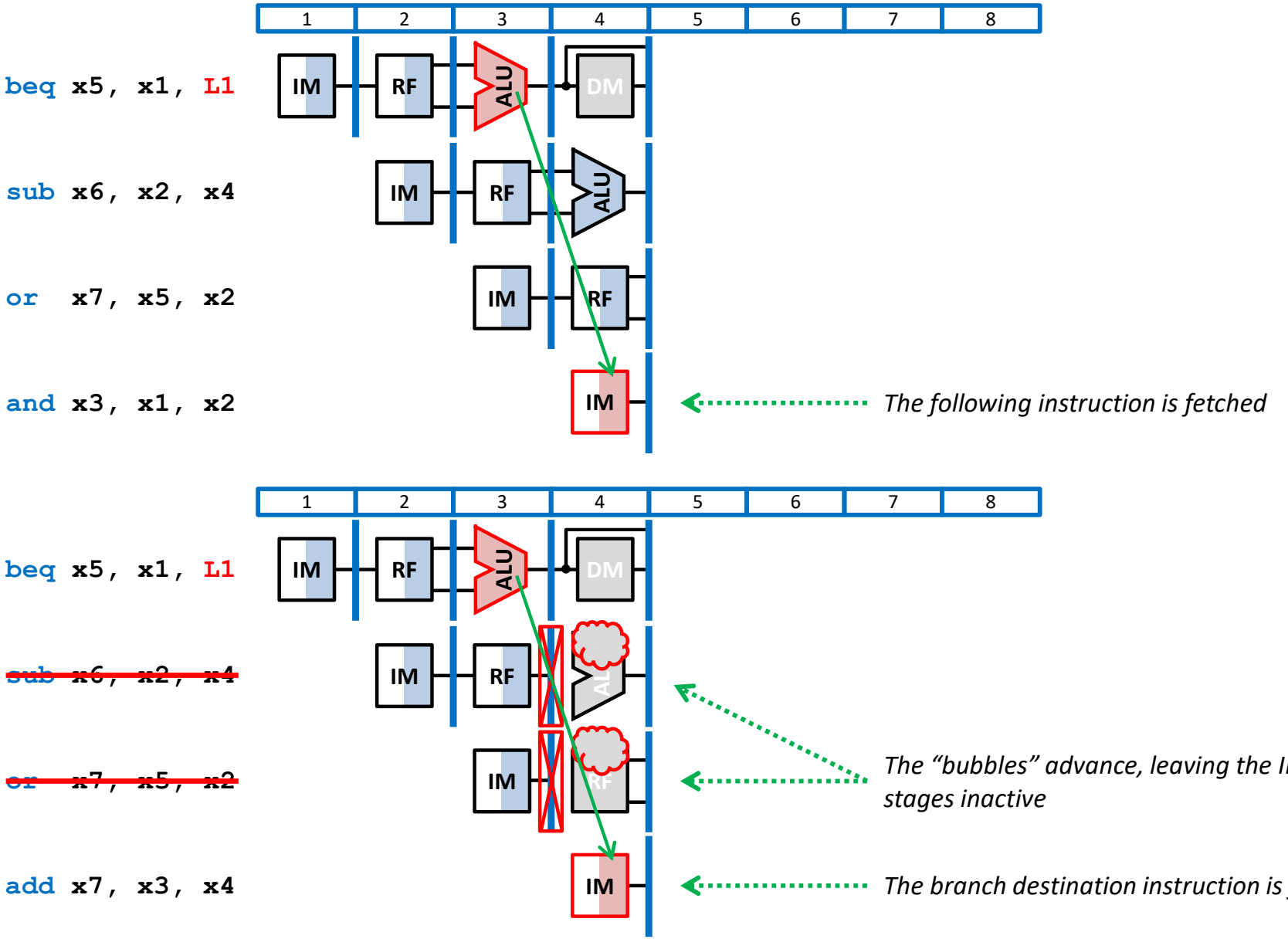


The **beq** *instruction decides that it has to branch to a given address, the prediction was wrong*

Since the prediction was wrong, the **sub** *,* **or** *instructions must be flushed: the IF/ID and ID/EX registers are deleted by inserting "bubbles"*

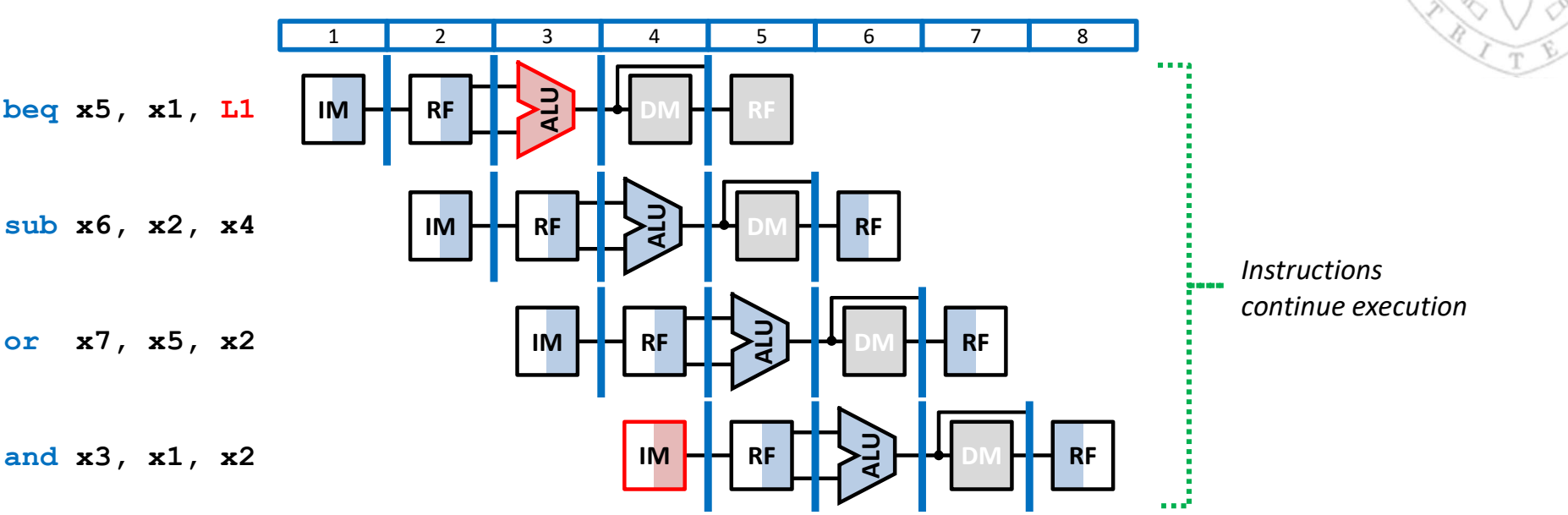
Control hazards

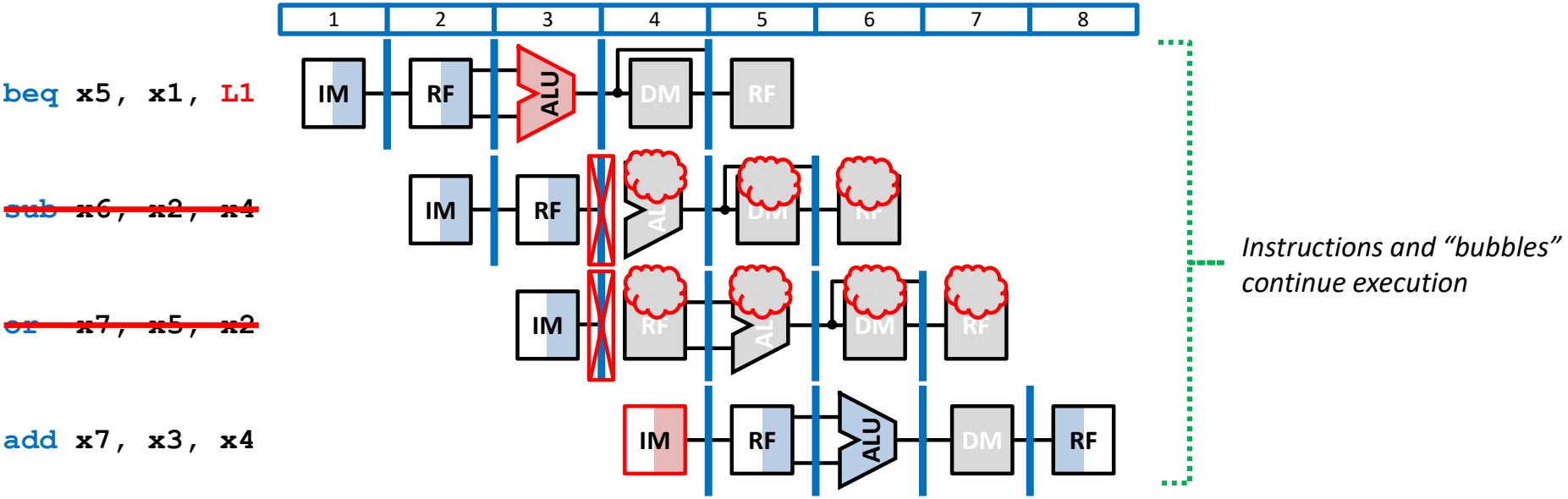
HW solution: branch prediction (ii)



Control hazards

HW solution: branch prediction (ii)

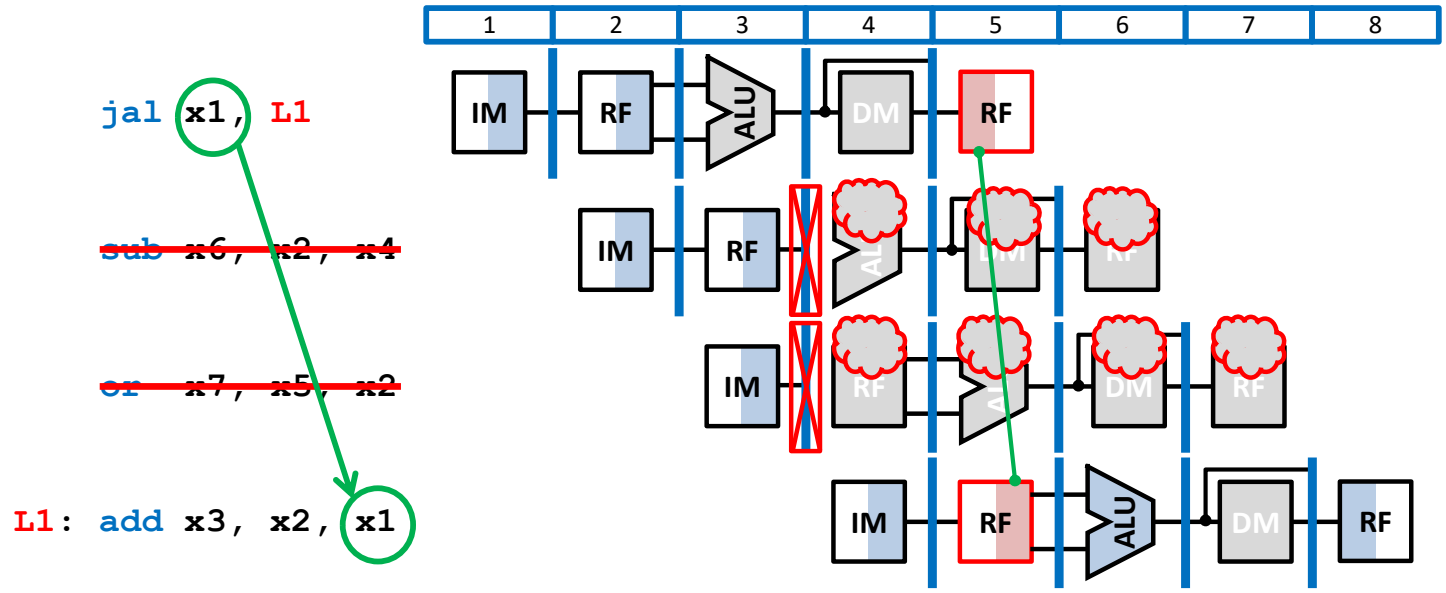




Control hazards

HW solution: branch prediction (iii)

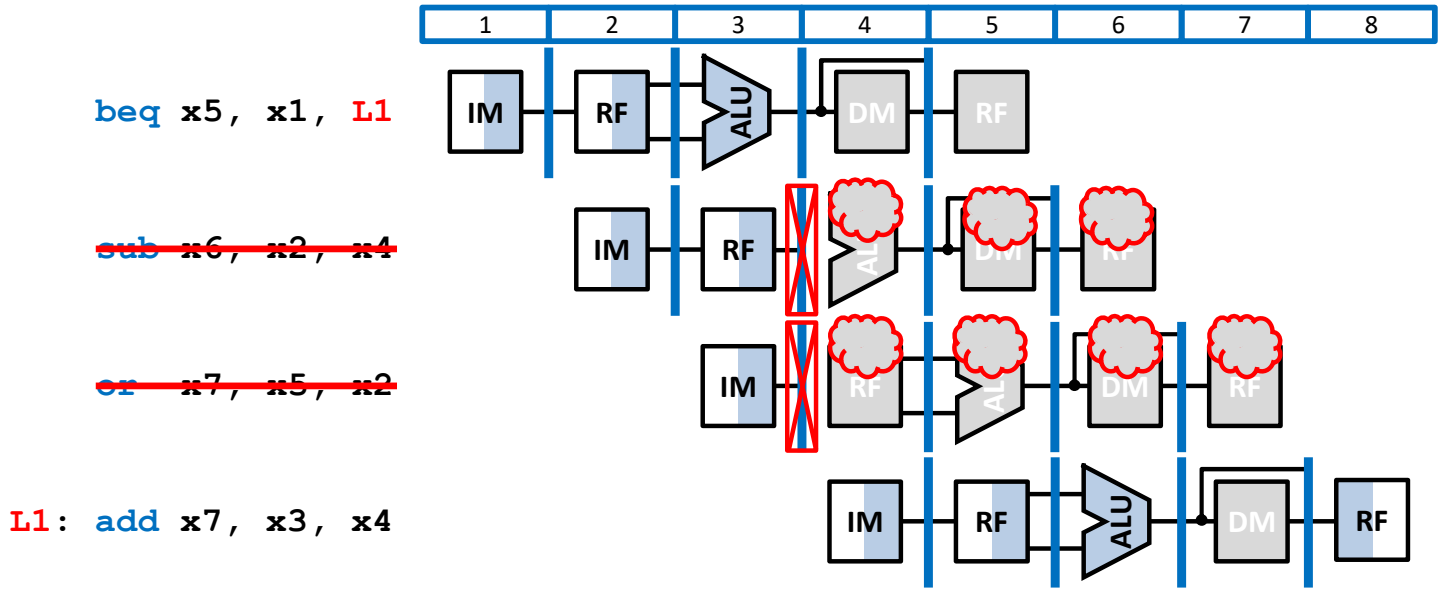
- For `jal` instructions (which always branch), predicting that the branch is not taken is always wrong, but it is used because:
  - The penalty is the same as stalling the pipeline.
  - No additional logic to the one needed for **`beq`** is needed.
  - Implicitly, this solves a special kind of data hazard.
    - The **`jal`** instruction stores PC+4 in **`x1`** during the WB stage, a value that is not in the ALU and therefore it cannot be forwarded using the designed data path.
    - Thanks to the 2-cycle delay, the updated value of **`x1`** can be read from the RF.



Control hazards

HW solution: branch prediction (iv)

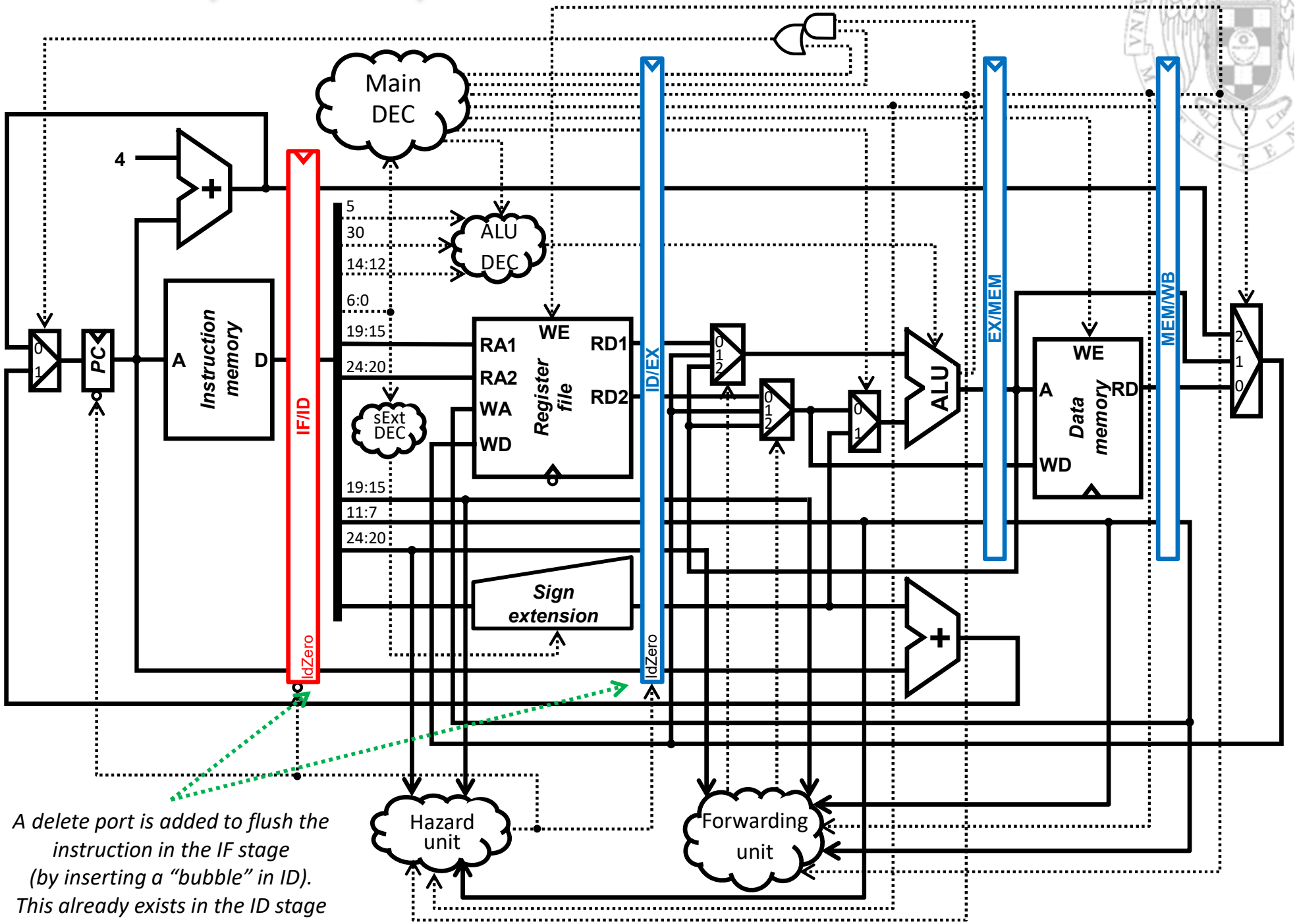
- In the simplified execution diagrams, the flushed instructions are marked explicitly:



	1	2	3	4	5	6	7	8
`beq x5, x1, L1`	IF	ID	EX	M	WB			
`sub x6, x2, x4`		IF	ID	X				
`or x7, x5, x2`			IF	X				
`L1: add x7, x3, x4`				IF	ID	EX	M	WB

Pipelined processor

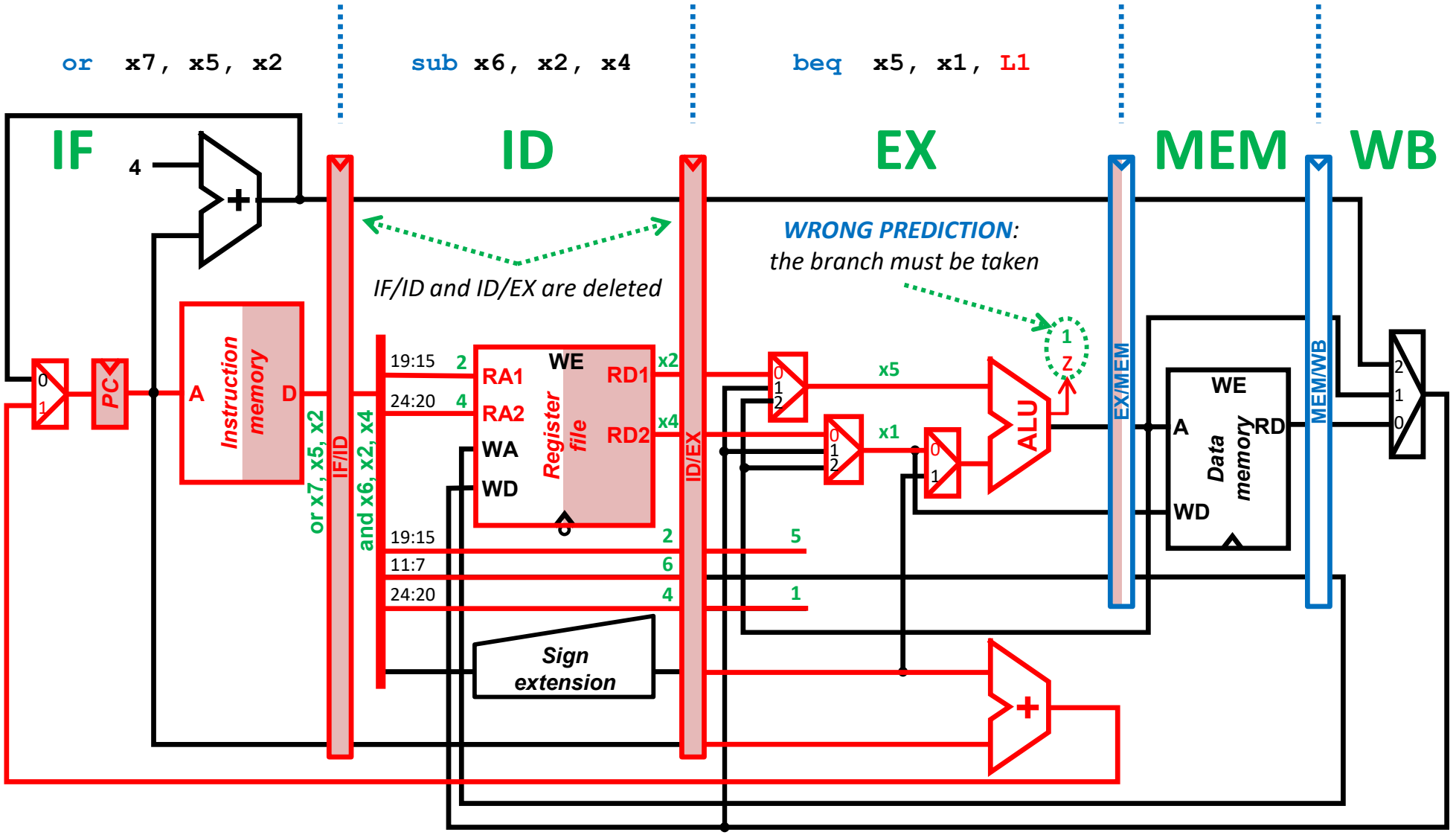
+ branch prediction: data path



A delete port is added to flush the instruction in the IF stage (by inserting a "bubble" in ID). This already exists in the ID stage

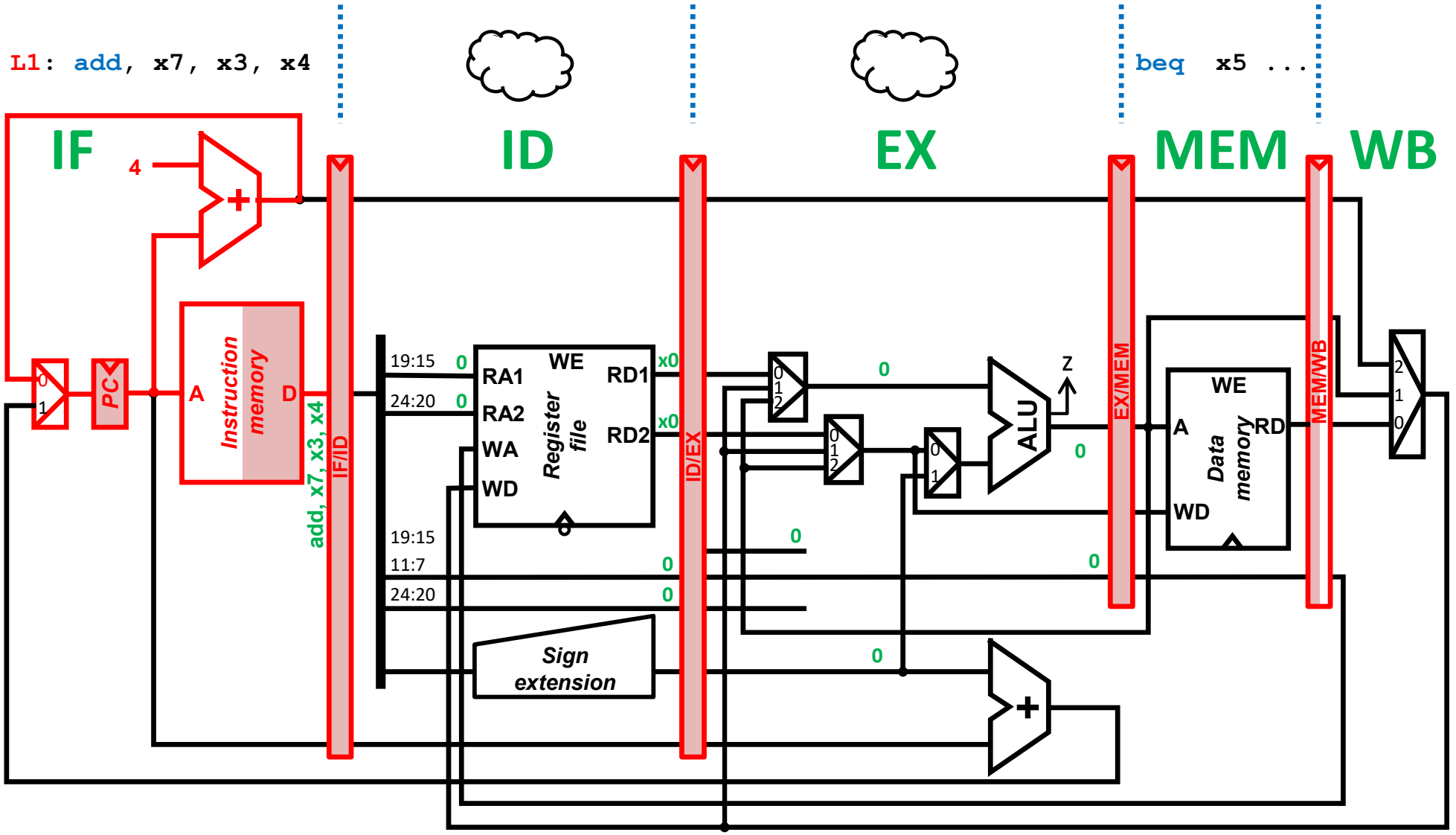
Pipelined processor

Wrong branch prediction simulation: 3rd. cycle



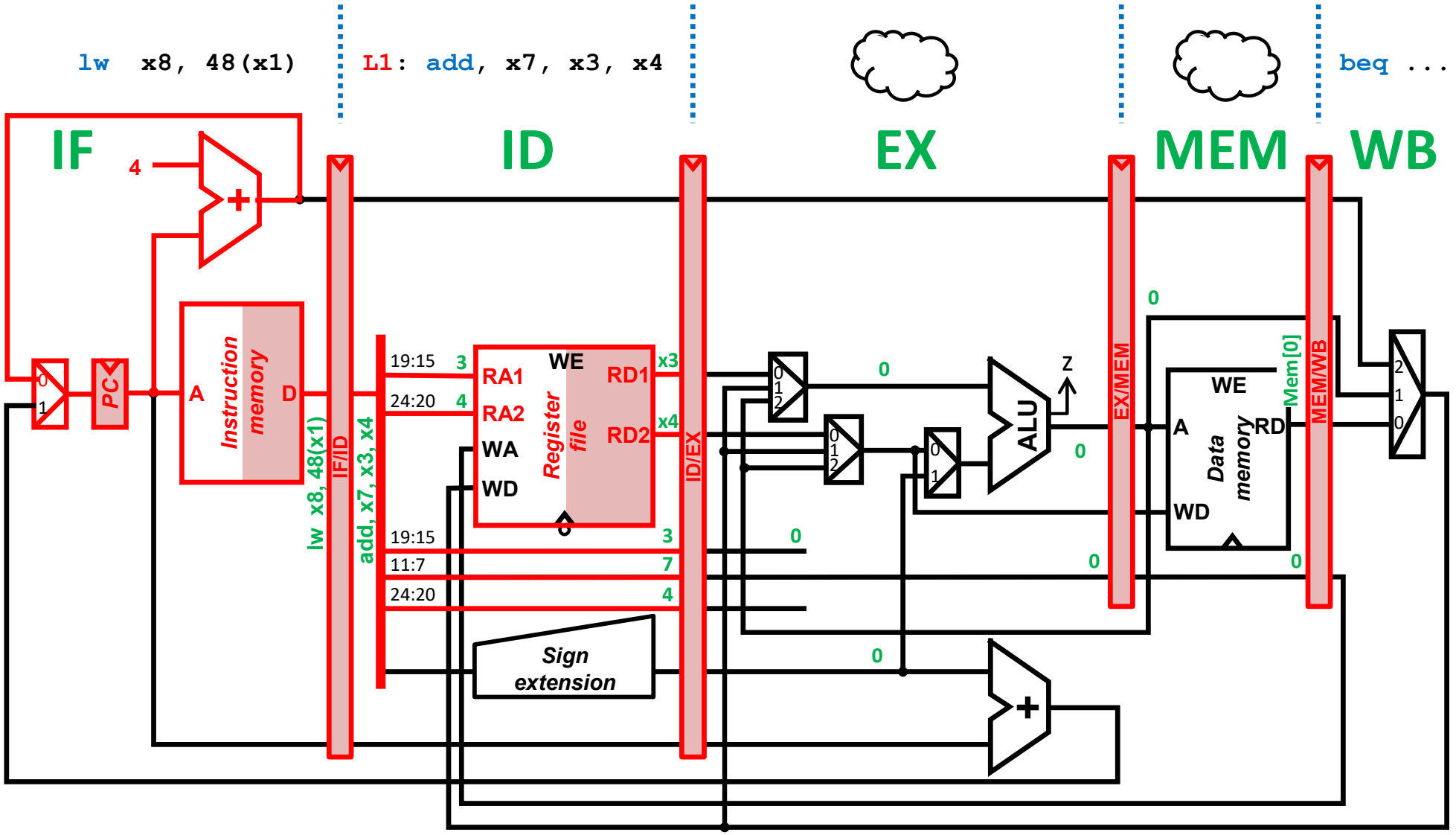
Pipelined processor

Wrong branch prediction simulation: 4th. cycle



Pipelined processor

Wrong branch prediction simulation: 5th. cycle



Pipelined processor

Extended hazard unit design (i)

- The hazard unit is extended in order to flush the instructions in the IF and ID stages if a branch has to be taken:
  - The IF/ID and ID/EX pipeline registers are deleted.

- In order to behave correctly, it must know:
  - PCsrcE: it is only active if the instruction in the EX stage is a branch and it has to be taken.

Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1)
else (0)
StallF ← Stall
StallD ← Stall
FlushE ← Stall | PCsrcE ⟵ *The ID/EX pipeline register is deleted*
FlushD ← PCsrcE ⟵ *The IF/ID pipeline register is deleted*

Pipelined processor

Extended hazard unit design (ii)

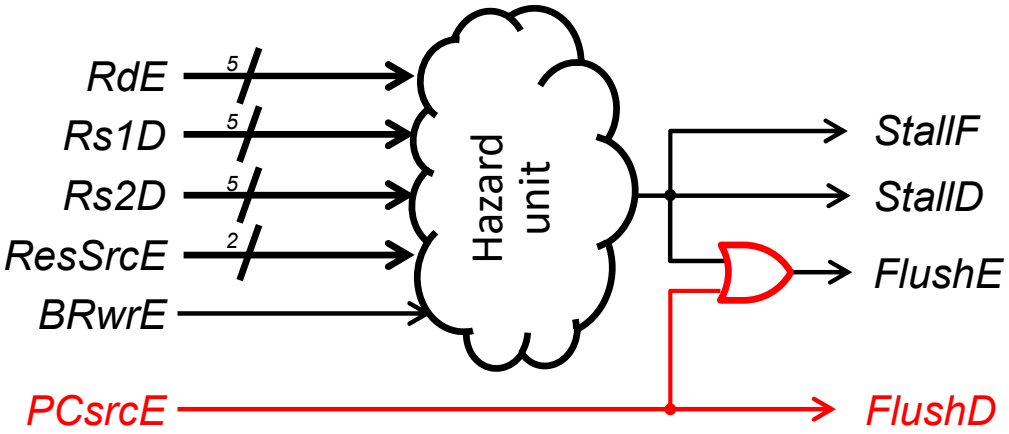
Stall ← *if* ((ResSrcE = 0) & BRwrE & ((Rs1D = RdE) | (Rs2D = RdE))) *then* (1)
else (0)

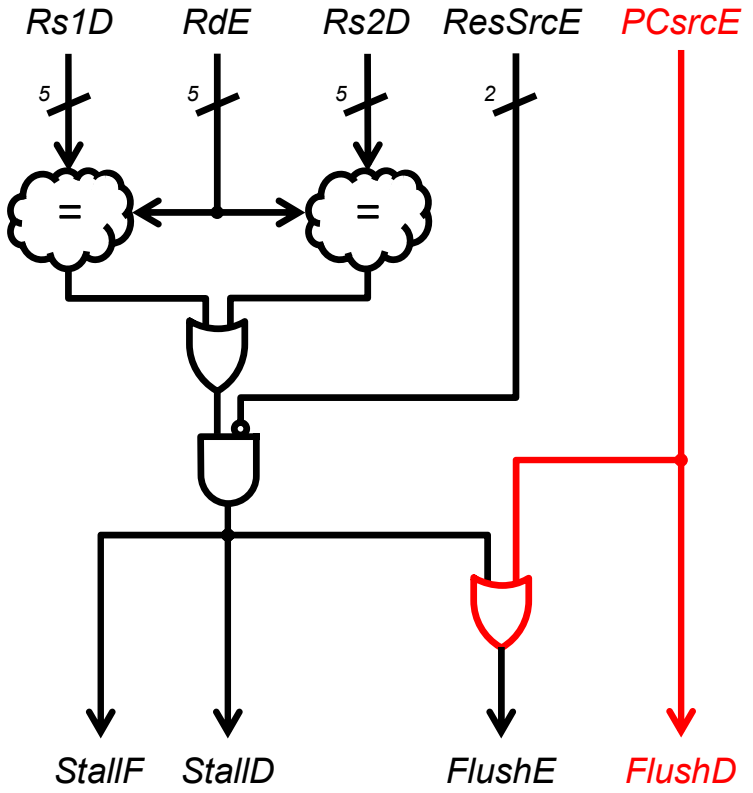
StallF ← Stall

StallD ← Stall

FlushE ← Stall | PCsrcE

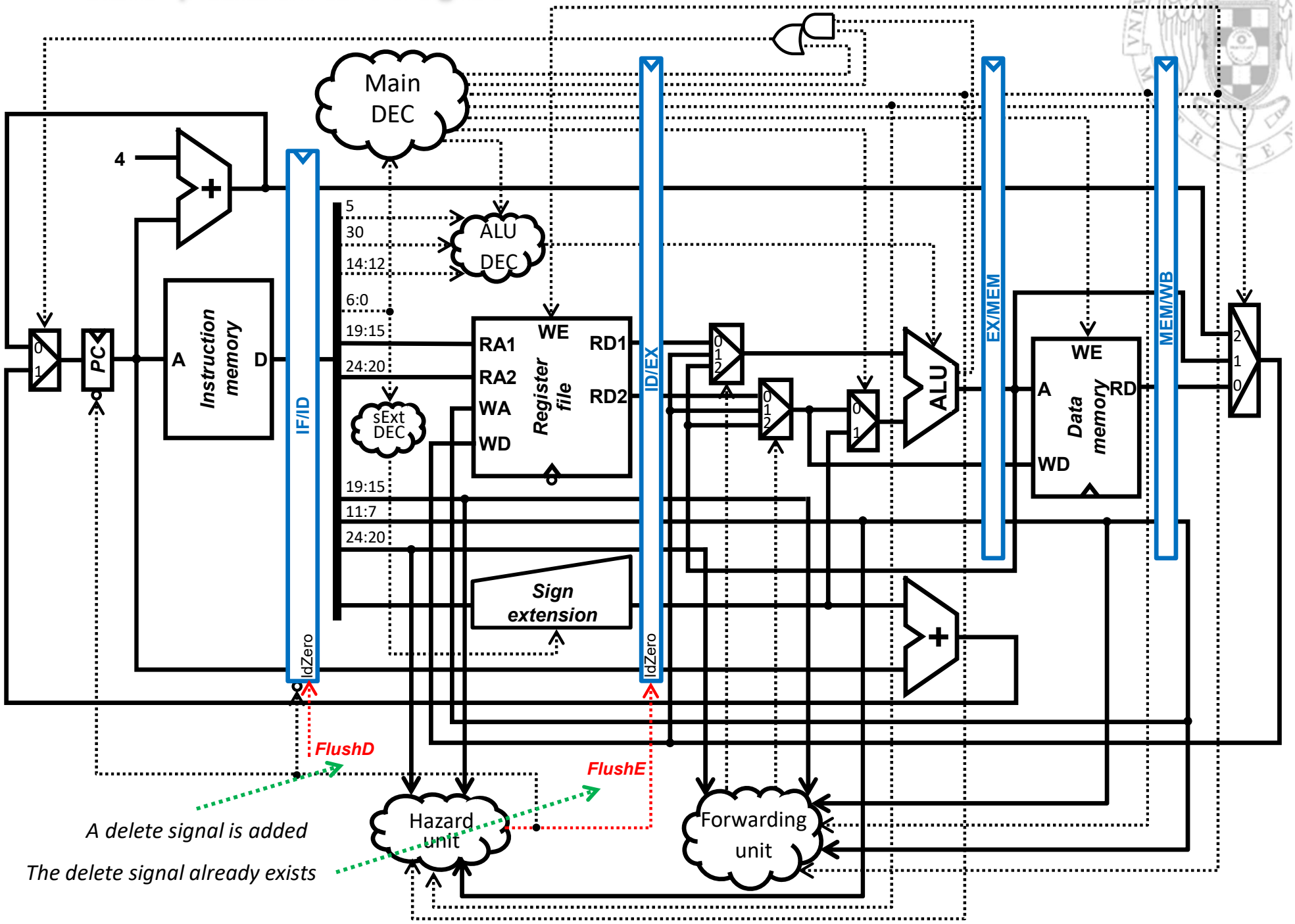
FlushD ← PCsrcE





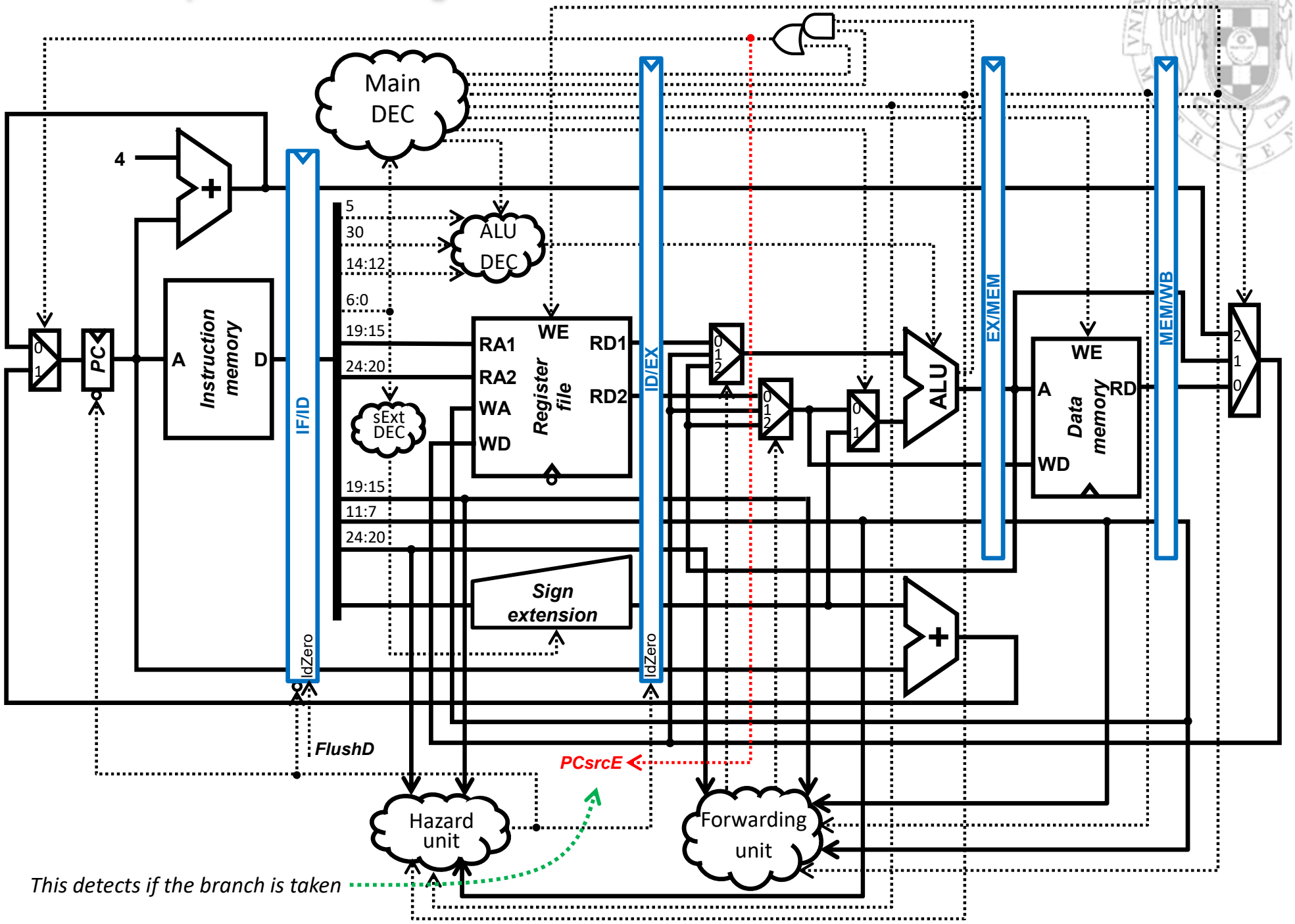
Pipelined processor

+ branch prediction: control signals



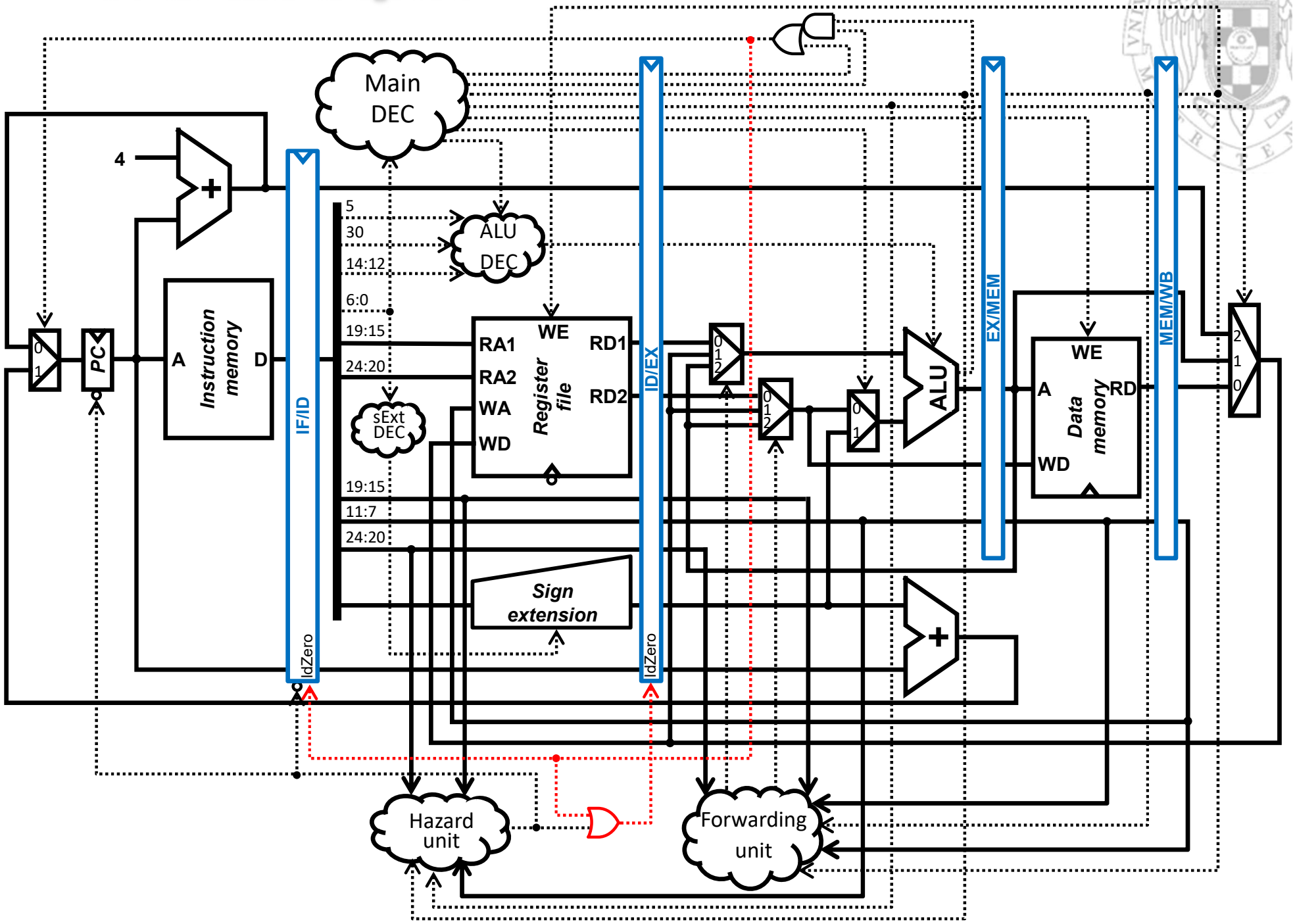
Pipelined processor

+ branch prediction: status signals



Pipelined processor

With full hazard management



Pipelined processor

SW vs. HW hazard management

Hazard type	Involved instructions	Penalty (cycles) SW	Penalty (cycles) HW	*Implemented HW solution*
Structural	*(this does not exist)*	-	-	-
Data	`add/i` –like → others	1, 2 or 3	0	*forwarding*
	`lw` → `add/i` –like	1, 2 or 3	1	*stall + forwarding* *(avoidable by code reordering)*
	`lw` → `lw`	1, 2 or 3	1	*stall + forwarding* *(avoidable by code reordering)*
	`lw` → `sw`	1, 2 or 3	1	*stall + forwarding* *(avoidable by optimized forwarding)*
Control	`beq`	2	0 or 2	*branch prediction*
	`jal`	2	2	*branch prediction*

Pipelined processor

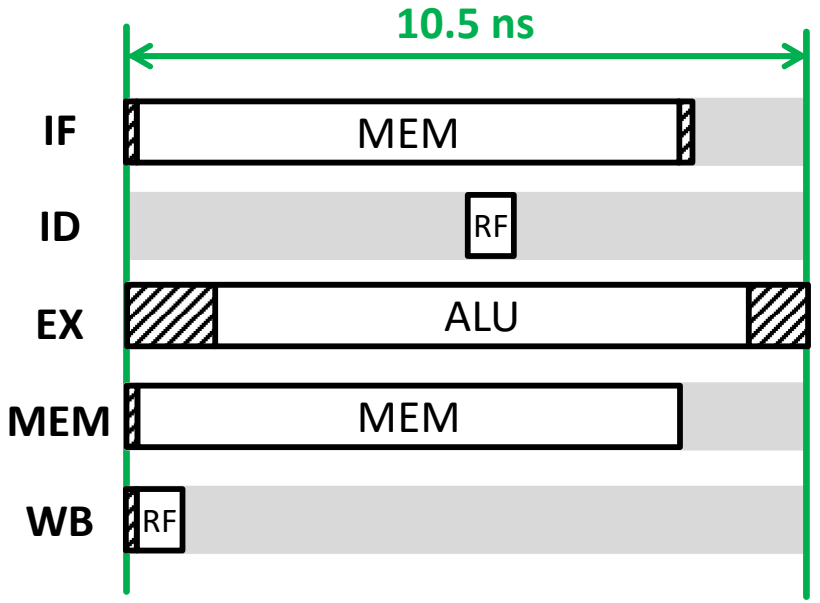
Cost and cycle time (CMOS 90 nm)

$$area = 77,018\ \mu m^2$$

$$t_{clk} = 10.5\ \text{ns}$$

$$f_{clk} = \frac{1}{t_{clk}} = \frac{1}{10.5 \cdot 10^{-6}\text{s}} = 95\ \text{MHz}$$

stage	critical path
IF	8,890 ps
ID	½t_{clk}+723 ps
EX	**10,541 ps**
MEM	8,667 ps
WB	1,122 ps
max.	**10,541 ps**



- Ideal pipelined CPI: without hazard penalty (CPI = 1).

$$CPI = 1$$
$$t_{exec} = 10^8 \cdot 1 \cdot 10.5\ ns = 1.05\ s$$
$$MIPS = 10^8/(10^6 \cdot 1.05\ s) = 95.2\ Minst/s$$

Pipelined processor

Performance metrics

- Given a program that executes 10^8 instructions (100 million) so that:
  - 25% of the instructions are `lw`
    - 40% are followed by an instruction that needs the loaded value: 1-cycle stall.
  - 10% of the instruction are `sw`
  - 11% of the instructions are `beq`
    - 50% are taken branches: wrong prediction and 2 instructions are flushed.
  - 2% of the instructions are `jal`
  - 52% of the instructions are arithmetic-logic
- Actual pipelined CPI: with hazard penalty (CPI > 1).
  - `lw`: 1/2 cycles, `beq`: 1/3 cycles, `jal`: 3 cycles, *other*: 1 cycle

$$\begin{aligned} \boldsymbol{CPI} &= 0.25 \cdot (0.6 \cdot 1 + 0.4 \cdot 2) && \leftarrow \texttt{lw}\ \textit{instructions} \\ &\quad + 0.10 \cdot 1 && \leftarrow \texttt{sw}\ \textit{instructions} \\ &\quad + 0.11 \cdot (0.5 \cdot 1 + 0.5 \cdot 3) && \leftarrow \texttt{beq}\ \textit{instructions} \\ &\quad + 0.02 \cdot 3 && \leftarrow \texttt{jal}\ \textit{instructions} \\ &\quad + 0.52 \cdot 1 = \mathbf{1.25} && \leftarrow \textit{Arithmetic-logic instructions} \end{aligned}$$

$$\boldsymbol{t_{exec}} = 10^8 \cdot 1.25 \cdot 10.5\ ns = \mathbf{1.31\ s}$$

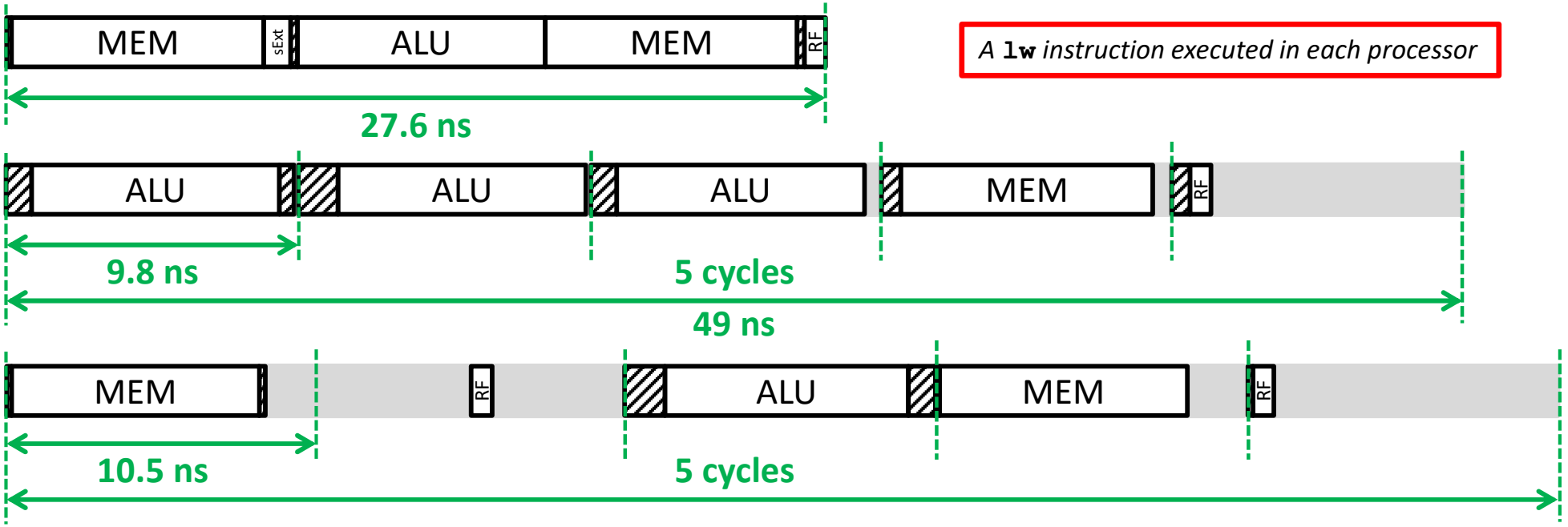
$$\boldsymbol{MIPS} = 10^8 / (10^6 \cdot 1.43\ s) = \mathbf{76.2\ Minst/s}$$

Comparison

Reduced RISC-V: single-cycle vs. multicycle vs. pipelined

- The pipelined processor is more expensive, but it has better performance

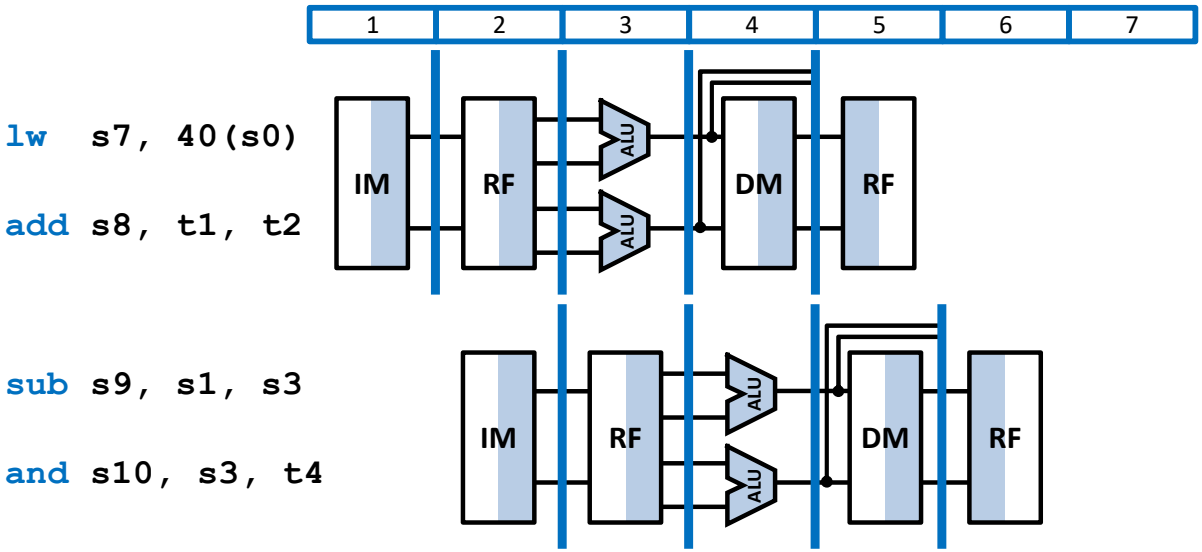
Processor	t_{clk} (ns)	CPI	Cost (μm^2)	t_{exec} (s)	↑ cost	Speedup
Single-cycle	27.6	1	59,181	2.76	1	1
Multicycle	9.8	4.14	65,626	4.06	1.12	0.68
Pipelined	10.5	1.25	77,018	1.31	1.30	2.11



Advanced microarchitectures

Superscalar processors

- A superscalar processor contains several copies of the data path:
  - It executes in parallel several instructions of the same program/thread.
  - A superscalar processor with 2 ways:
    - Has 2 ALUs, and the RF and the memory have duplicated ports.
    - Fetches 2 instructions per cycle (ideal CPI = ½).

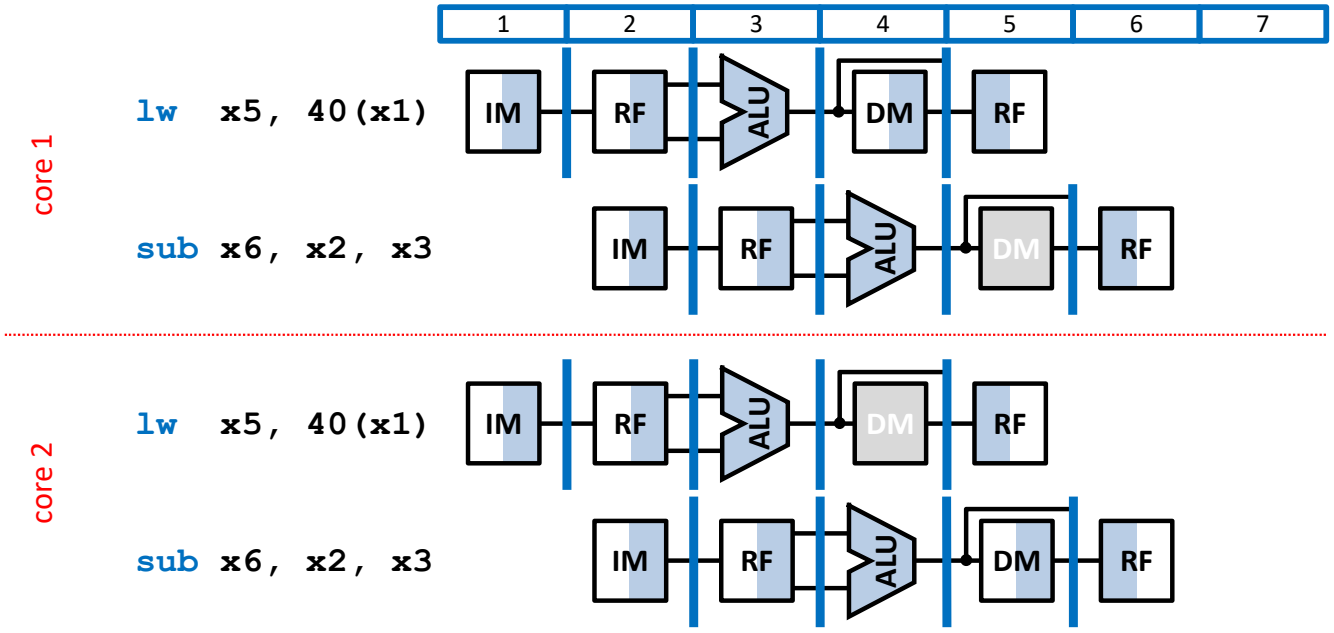


  - The data/control hazard probability increases:
    - To reduce it, it executes instructions in an order different from the one in which they are written in the program (out of order), making sure the result is correct

Advanced microarchitectures

Multicore processors

- A multicore processor contains several copies of the full processor:
  - It executes in parallel several instructions of different programs/threads.
  - A dual core processor:
    - Has 2 full pipelined processors that share the memory.
    - Fetches 2 instructions per cycle (ideal CPI = ½).



  - Each core may also be a superscalar:
    - A dual core superscalar with 2 ways, fetches 4 instructions per cycle (ideal CPI = ¼).

Advanced microarchitectures

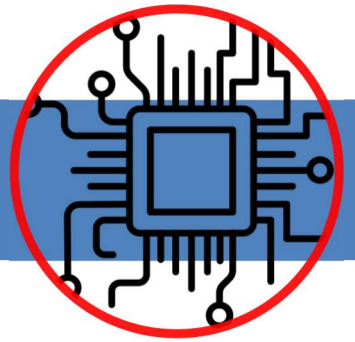
Intel processors

Microprocessor	Year	f_{clk} (MHz)	# Stages	# Ways	Out of order	# cores
486	1989	25	5	1	No	1
Pentium	1993	66	5	2	No	1
Pentium Pro	1997	200	10	3	Yes	1
Pentium 4 Willamette	2001	2000	22	3	Yes	1
Pentium 4 Prescott	2004	3600	31	3	Yes	1
Core	2006	3600	14	4	Yes	2
Core i7 Nehalem	2008	3600	14	4	Yes	2-4
Core Westmere	2010	3730	14	4	Yes	6
Core i7 Ivy Bridge	2012	3400	14	4	Yes	6
Core Broadwell	2014	3700	14	4	Yes	10
Core i9 Skylake	2016	3100	14	4	Yes	14
Ice Lake	2018	4200	14	4	Yes	16

source: A.A. Patterson & J.L. Hennessy, Computer Organization and Design, RISC-V Edition (2nd. edition). (2021)

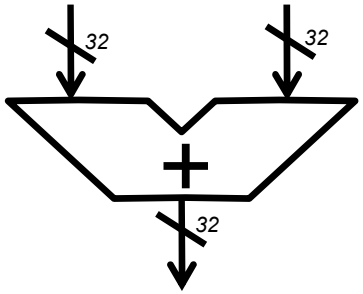
- Cost calculation.
- Cycle time calculation.

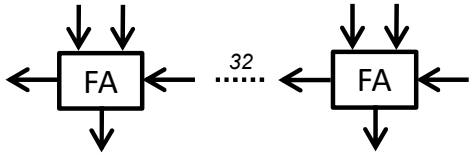
Technology



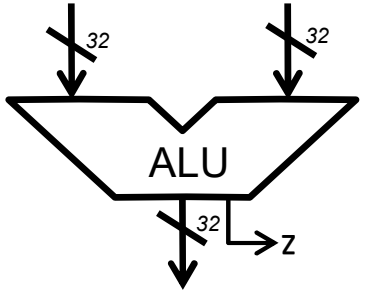
Cost and cycle time calculation

90 nm CMOS

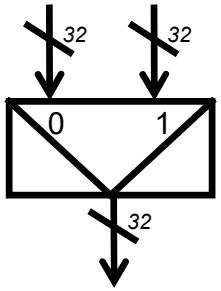


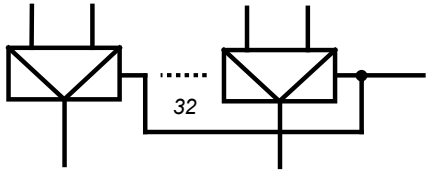


area: 32×29.49 = **944 μm²**
delay: 32×226 = **7,232 ps**

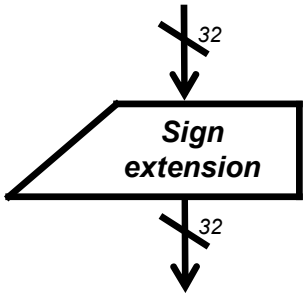


area: **3,052 μm²**
delay: **8,360 ps**

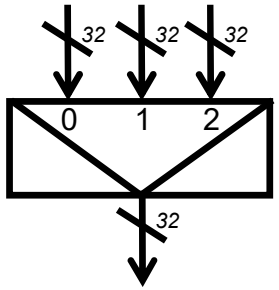


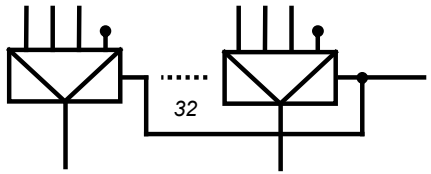


area: 32×11.05 = **354 μm²**
delay: **223 ps**

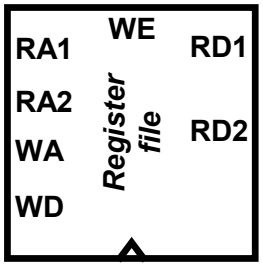


area: **202 μm²**
delay: **460 ps**





area: 32×23.04 = **737 μm²**
delay: **250 ps**



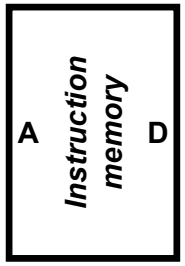
area: **51,405 μm²**
read delay: **723 ps**
write setup: **705 ps**
(due to the DEC address)

90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

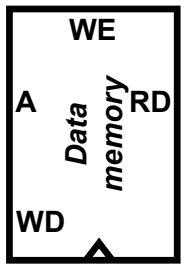
Cost and cycle time calculation

CMOS 90 nm

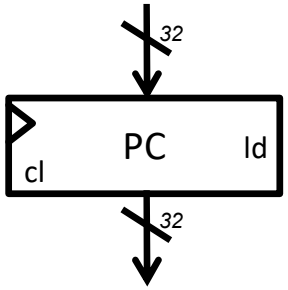
Idealized behavior: delay comparable to the one of the ALU
(so that it can be read in one clock cycle)

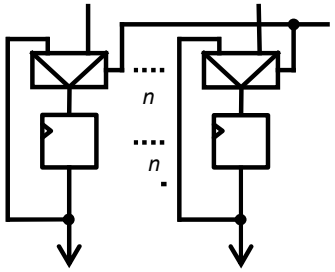


area: –
access time: 8,500 ps



area: –
access time: 8,500 ps





area: 32×11.05 + 32×32.26 = **1386 μm²**
CLK→Q delay: **167 ps**
setup: 1×223 = **223 ps** (due to the load MUX)

area: **56 μm²**
delay: **490 ps**

area: **65 μm²**
delay: **451 ps**

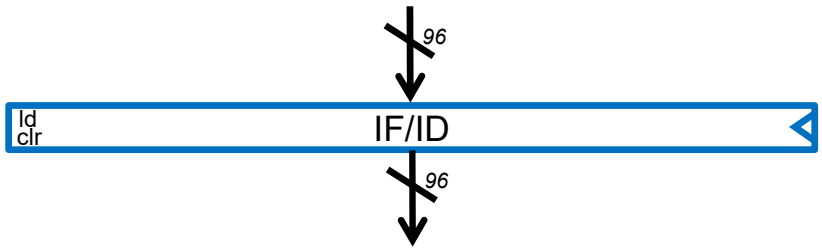
area: **21 μm²**
delay: **451 ps**

area: **15 μm²**
delay: **351 ps**

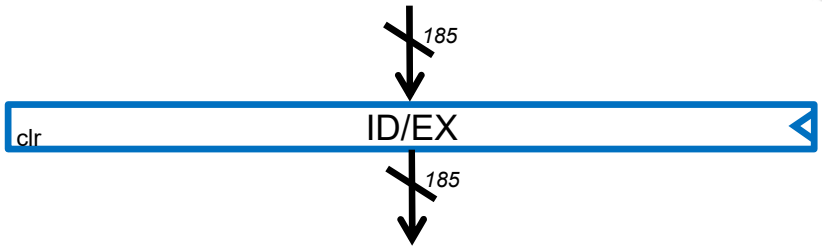
90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

Cost and cycle time calculation

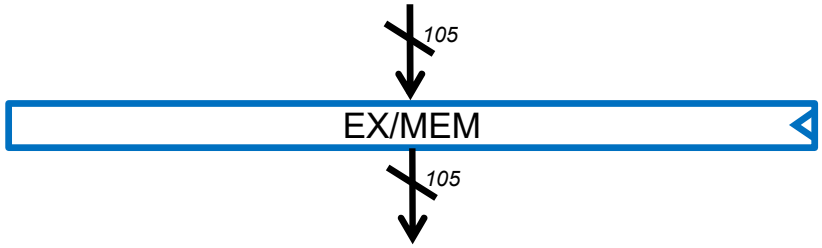
CMOS 90 nm



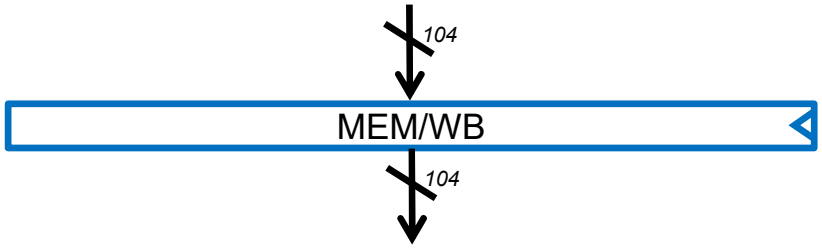
area: 96×11.05 + 96×32.26 = **4,158 μm²**
CLK→Q delay: **167 ps**
setup: 1×223 = **223 ps** (due to the load MUX)



area: 185×32.26 = **5,968 μm²**
CLK→Q delay: 1×167 = **167 ps**
setup: **0 ps**



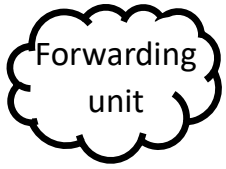
area: 105×24.88 = **2,612 μm²**
CLK→Q delay: **167 ps**
setup: **0 ps**



area: 104×24.88 = **2,588 μm²**
CLK→Q delay: **167 ps**
setup: **0 ps**

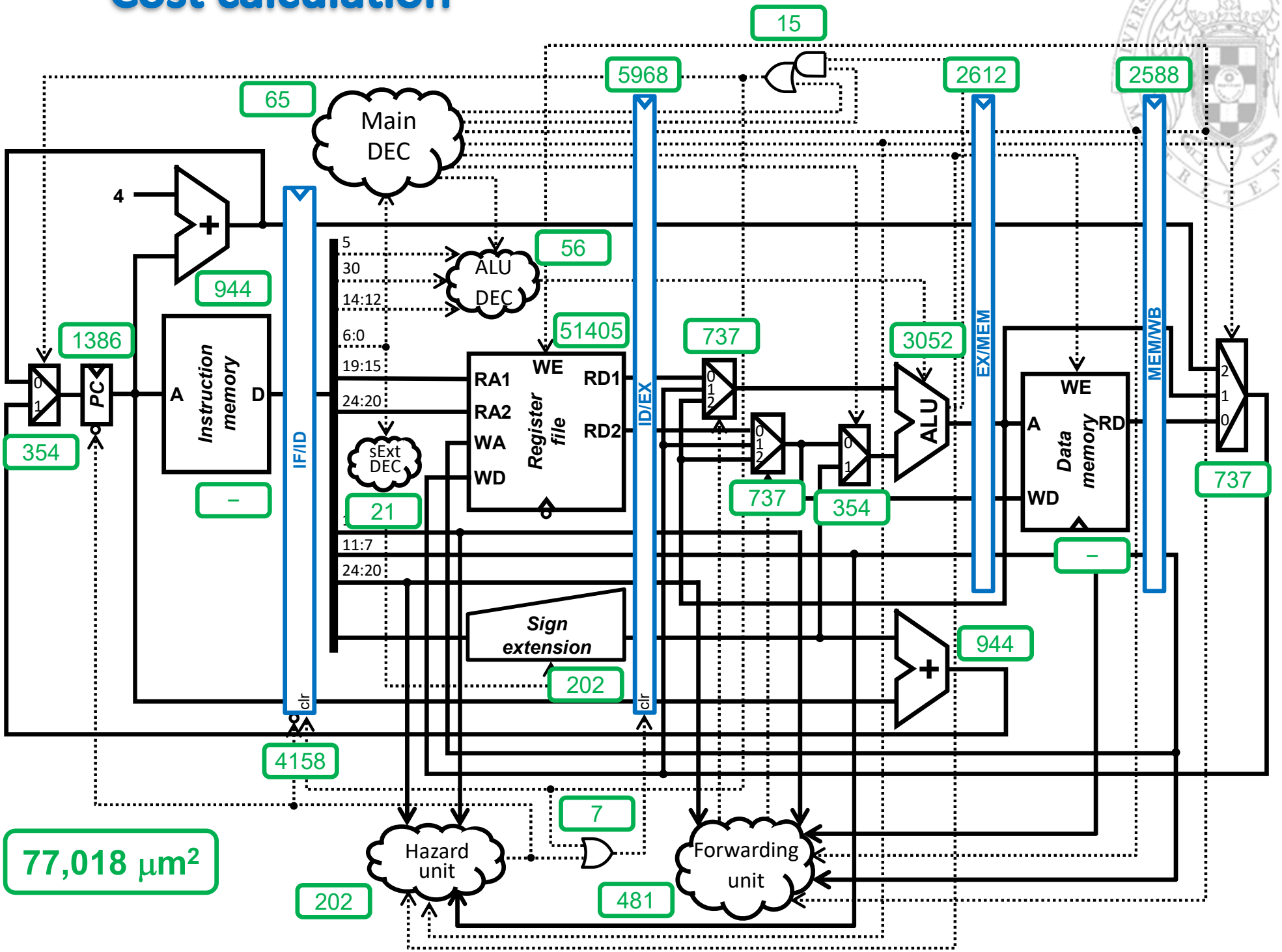
area: **195 μm²**
delay: **881 ps**

area: **7 μm²**
delay: **171 ps**



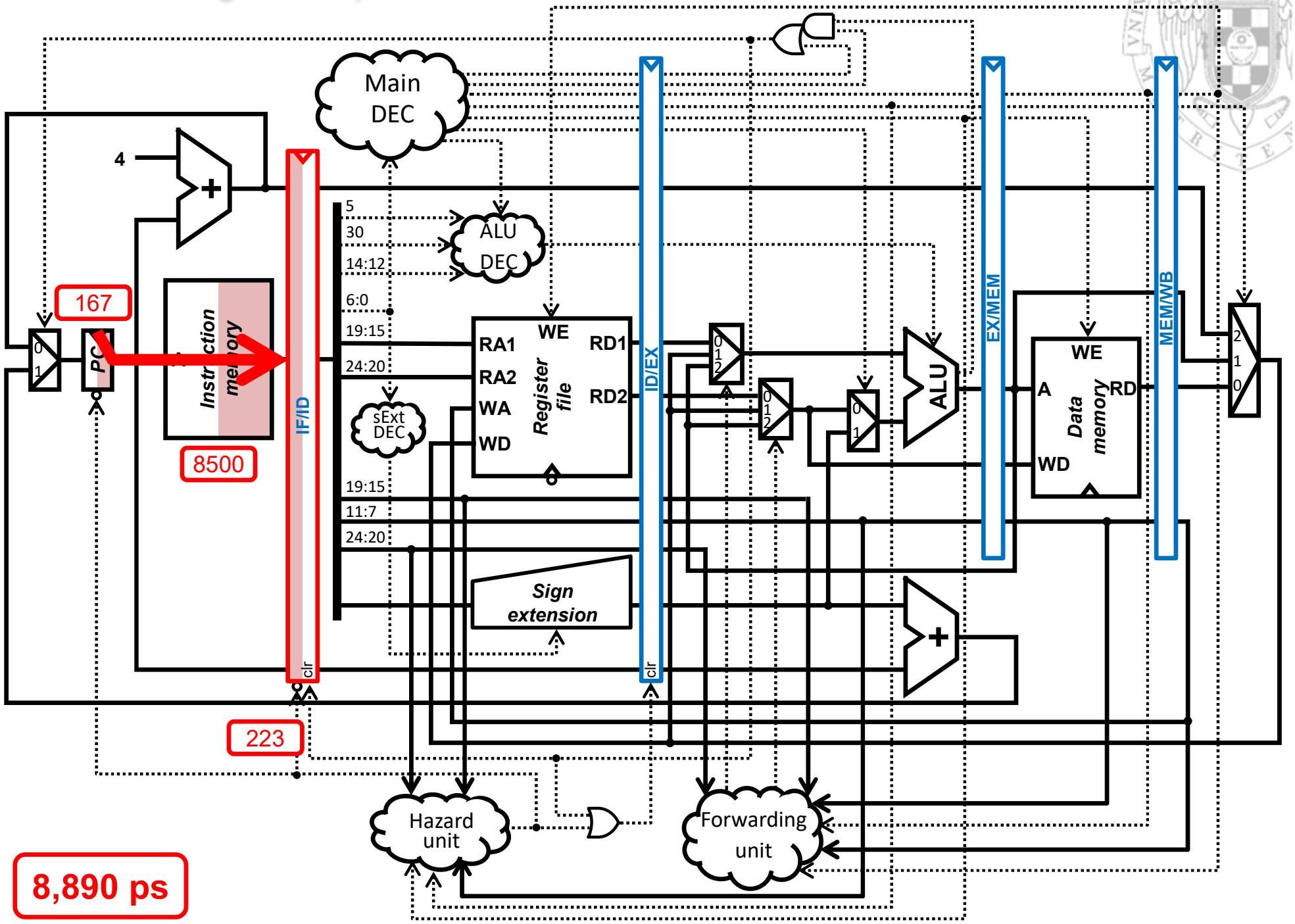
area : **418 μm²**
delay: **744 ps**

Cost calculation



Cycle time calculation

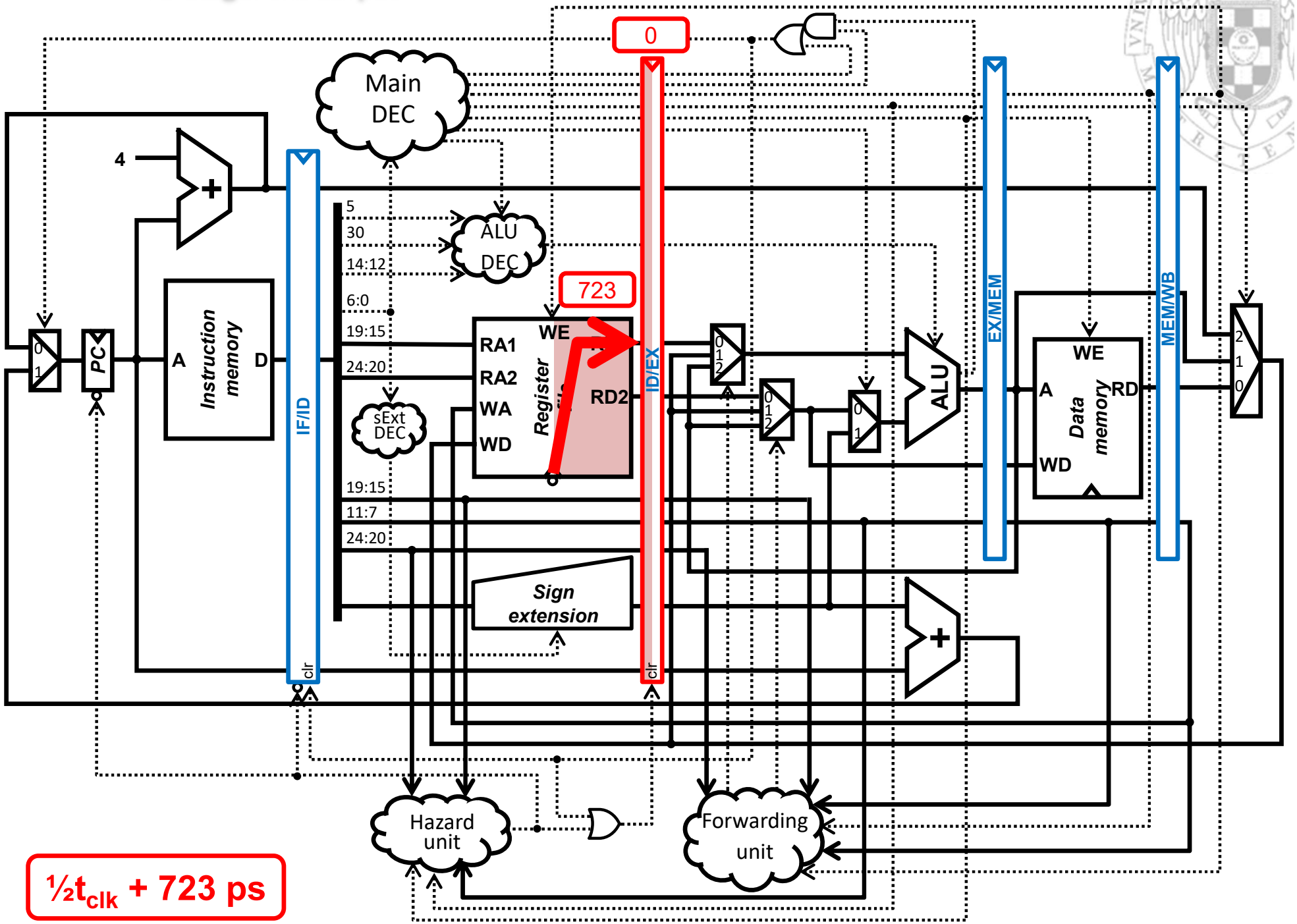
IF stage: critical path



8,890 ps

Cycle time calculation

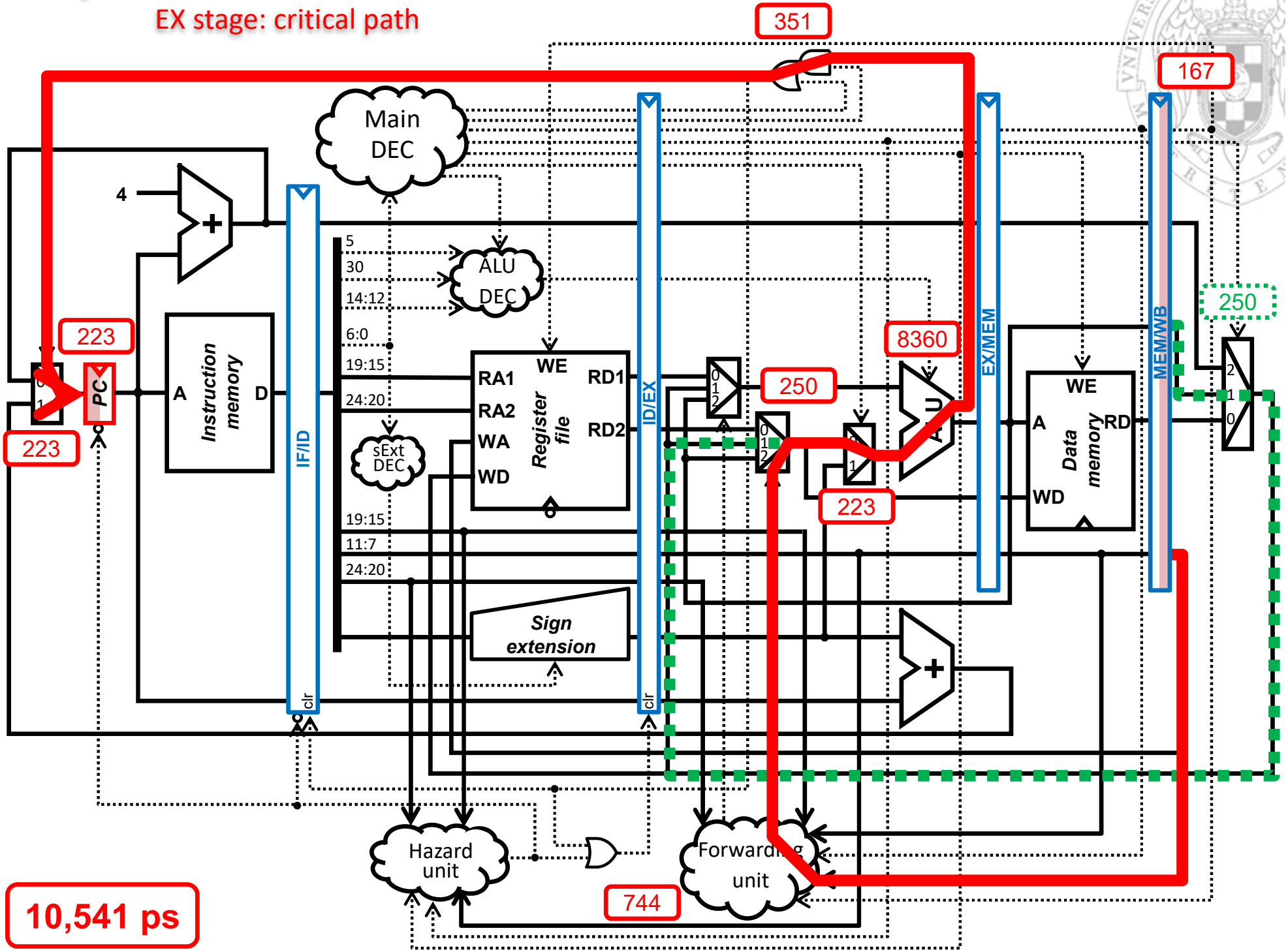
ID stage: critical path



$\frac{1}{2}t_{clk}$ + 723 ps

Cycle time calculation

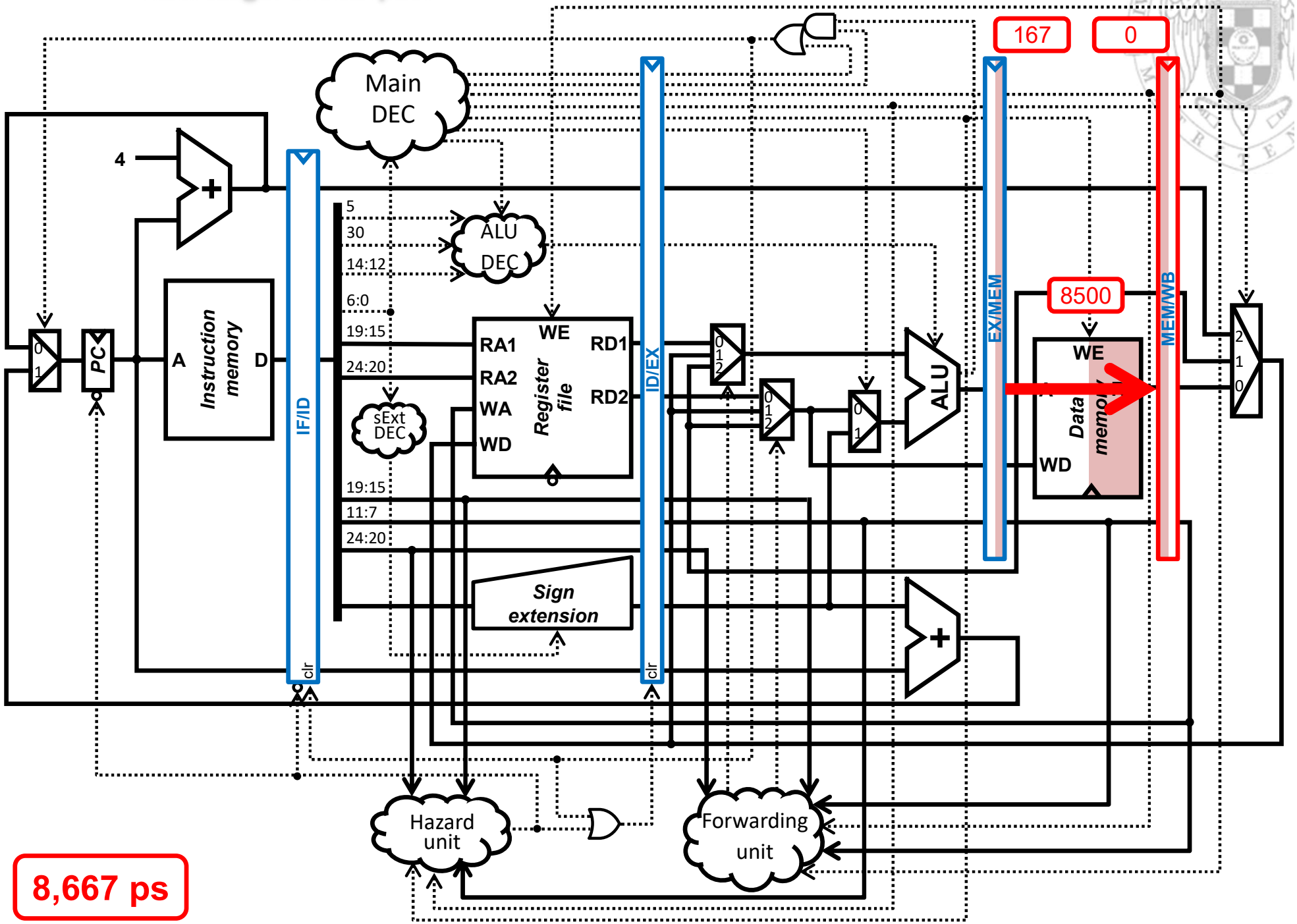
EX stage: critical path



10,541 ps

Cycle time calculation

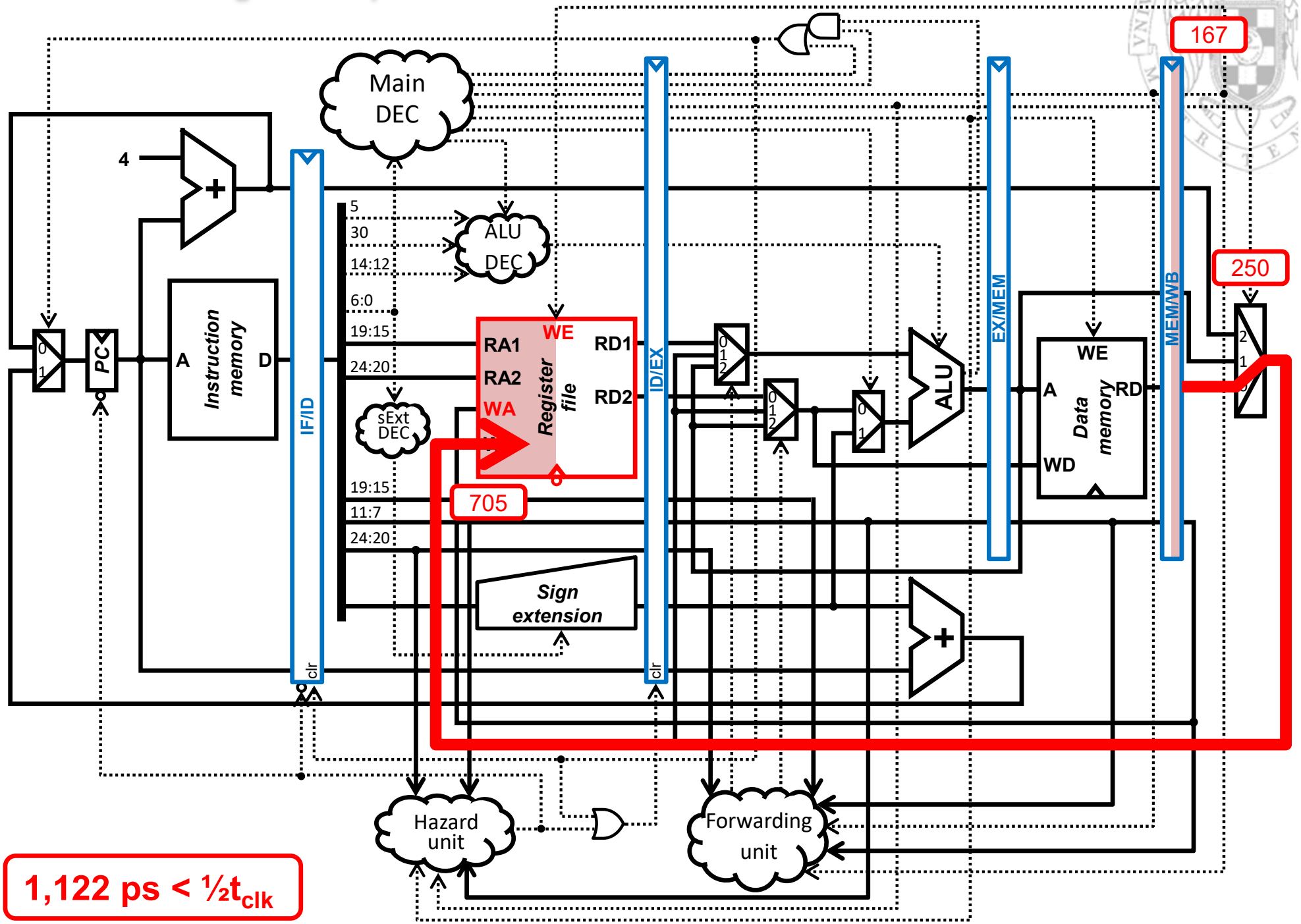
MEM stage: critical path



8,667 ps

Cycle time calculation

WB stage: critical path



$1{,}122\ \text{ps} < \frac{1}{2}t_{clk}$

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/