Course Info

- Lab 7 next week, prepare before lab sessions!
- Project 1.2 ddl soon (March 31st).
- Project 2.1 coming soon! April 14th ddl.
- Next week discussion on pipeline & superscalar.
- Mid-term I solution & score released. If you have questions about the solution, feel free to ask on Piazza; If you have questions regarding your marks, email the instructors **BEFORE April 2nd**. We will get back to you ASAP.
- Any regrade request after April 2nd **WOULD NOT** be considered.

Course Info

- HW4 released. Submit your paper homework to the box below (at SIST 3-322). DDL April 7th.
- Remember to add your name. You have only one chance to submit and cannot be withdrawn.

信息科学与技术学院
School of Information Science and Technology

CS 110 Computer Architecture Hazards & Advanced Techniques

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/3/23

Review

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

- We have built a pipelined processor!
  - Each instruction might consumes a longer time, but the overall throughput is improved
  - $T_{clk} =$ Max delay($t_{IF}, t_{ID}, t_{EX}, t_{MEM}, t_{WB}$)
  - Max Frequency = 1/Max delay
- A hazard is a situation in which a planned instruction cannot execute in the “proper” clock cycle.
  - Structural hazard: does not exist in our current design
  - Data hazard:
    - Solution 1: insert nop/bubble/stall, CPI increased

Three Types of Pipeline Hazards

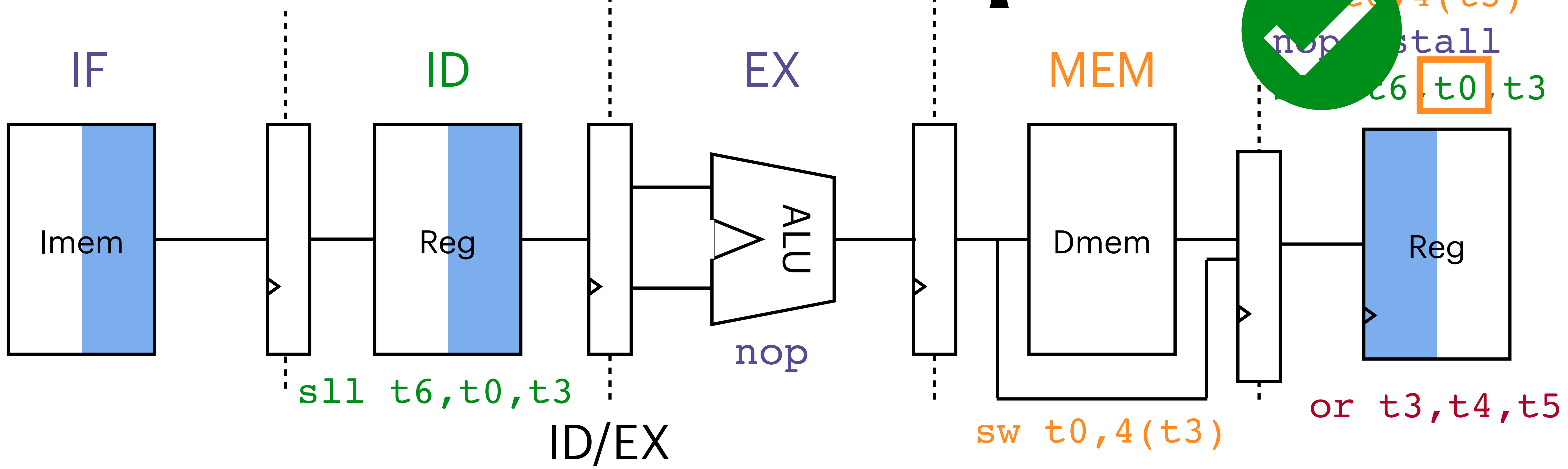
A hazard is a situation in which a planned instruction cannot execute in the “proper” clock cycle.

1. Structural hazard:
   - Hardware does not support access across multiple instructions in the same cycle
2. **Data hazard**:
   - Instructions have data dependency
   - Occurs when an instruction reads a register before a previous instruction has finished writing to that register

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
```

`t0` as an Example

```
add t0,t1,t2
lw  t0,8(t3)
or  t3,t4,t5
    ,4(t3)
nop  stall
   t6,t0,t3
```



Clock cycle 5

IF nop **ID sw** **EX or** **MEM lw** **WB add**

Clock cycle 6

IF sll **ID nop** **EX sw** **MEM or** **WB lw**

Clock cycle 7: `t0` updated at the beginning of this clock cycle (posedge)

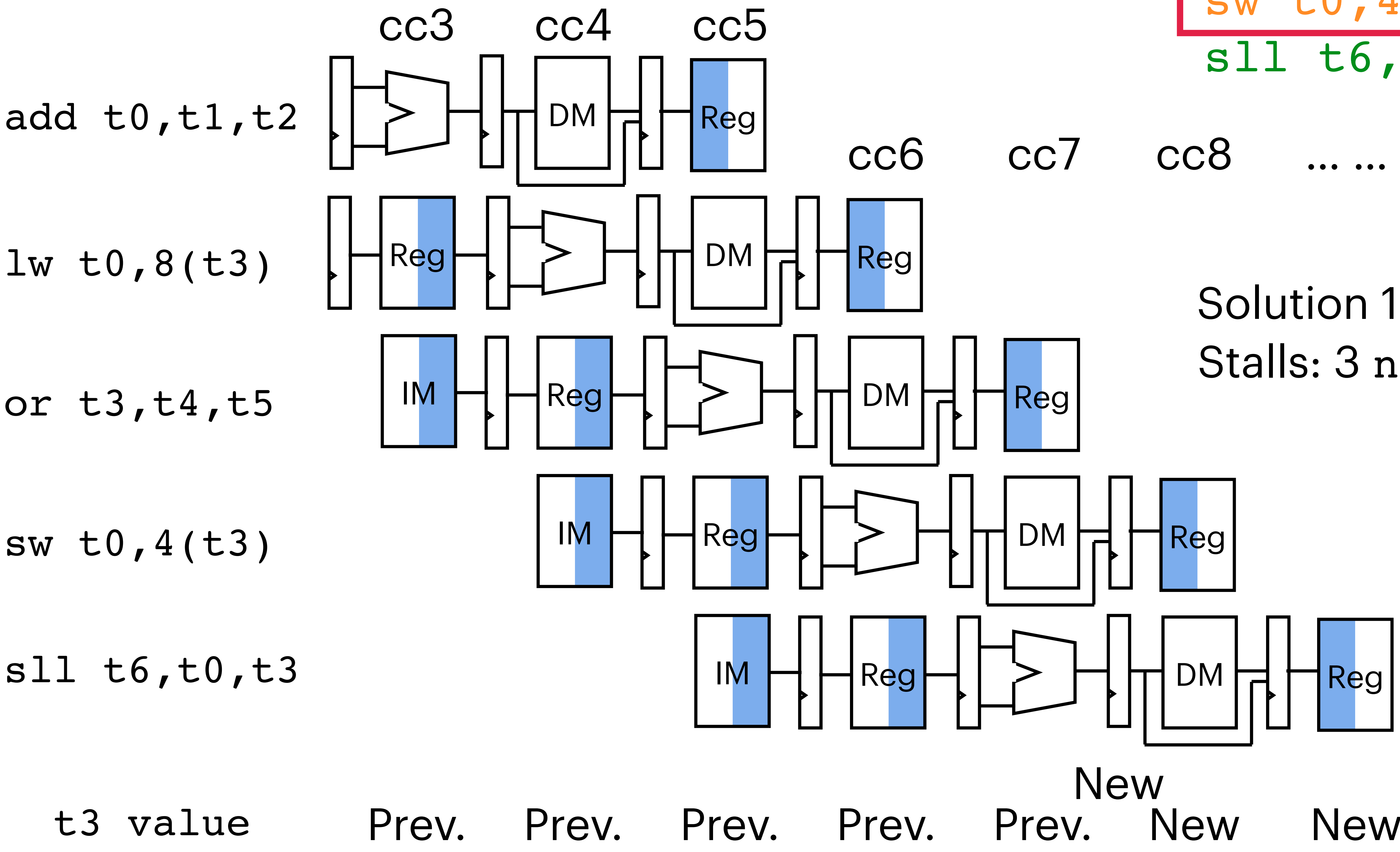
IF next **ID sll** **EX nop** **MEM sw** **WB or**

Covered by RegFile write-then-read in the same clock cycle

Solution 1

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
```

Starting from cc3



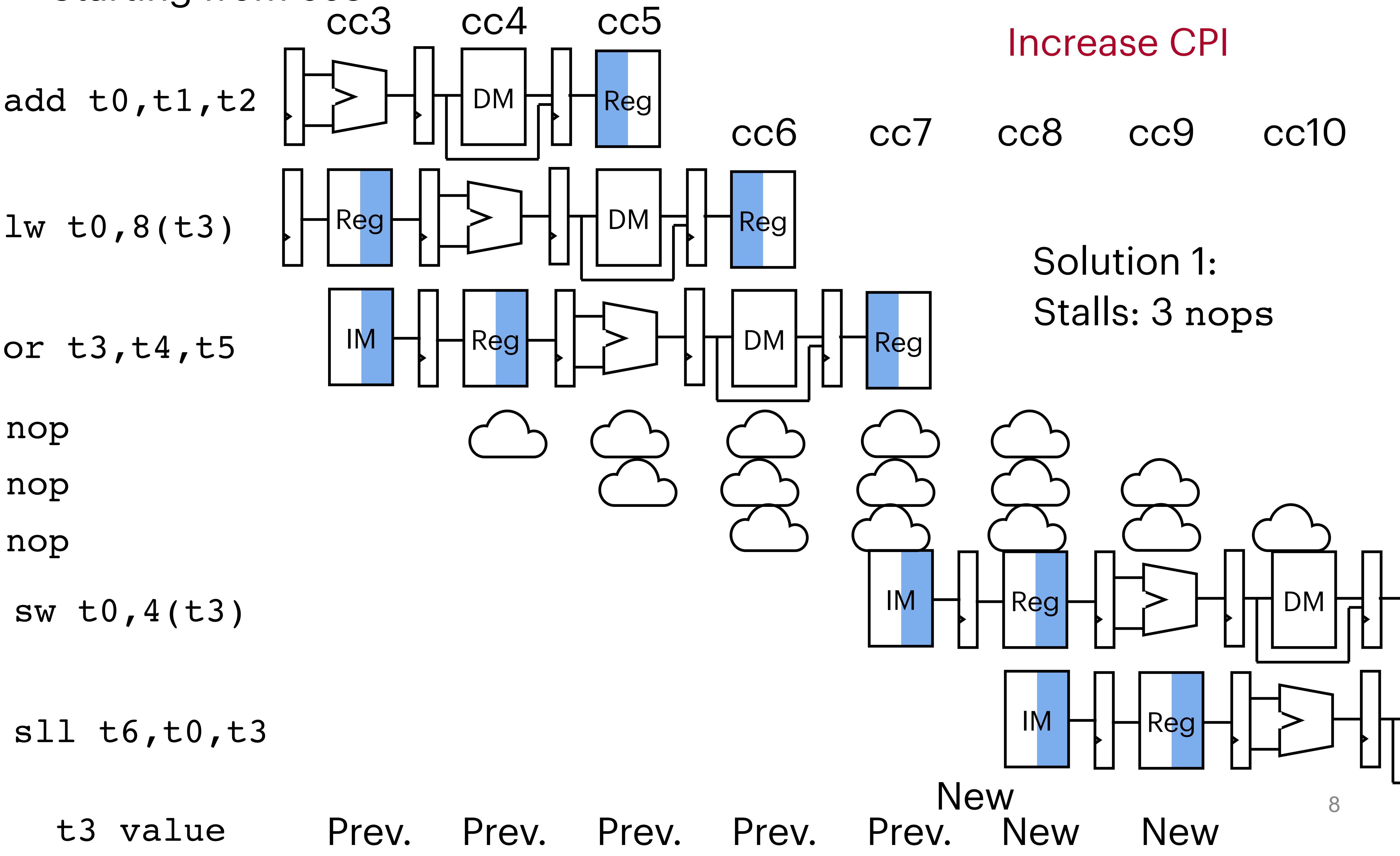
Solution 1:
Stalls: 3 nops

Stalls and Performance

Starting from cc3

Increase CPI

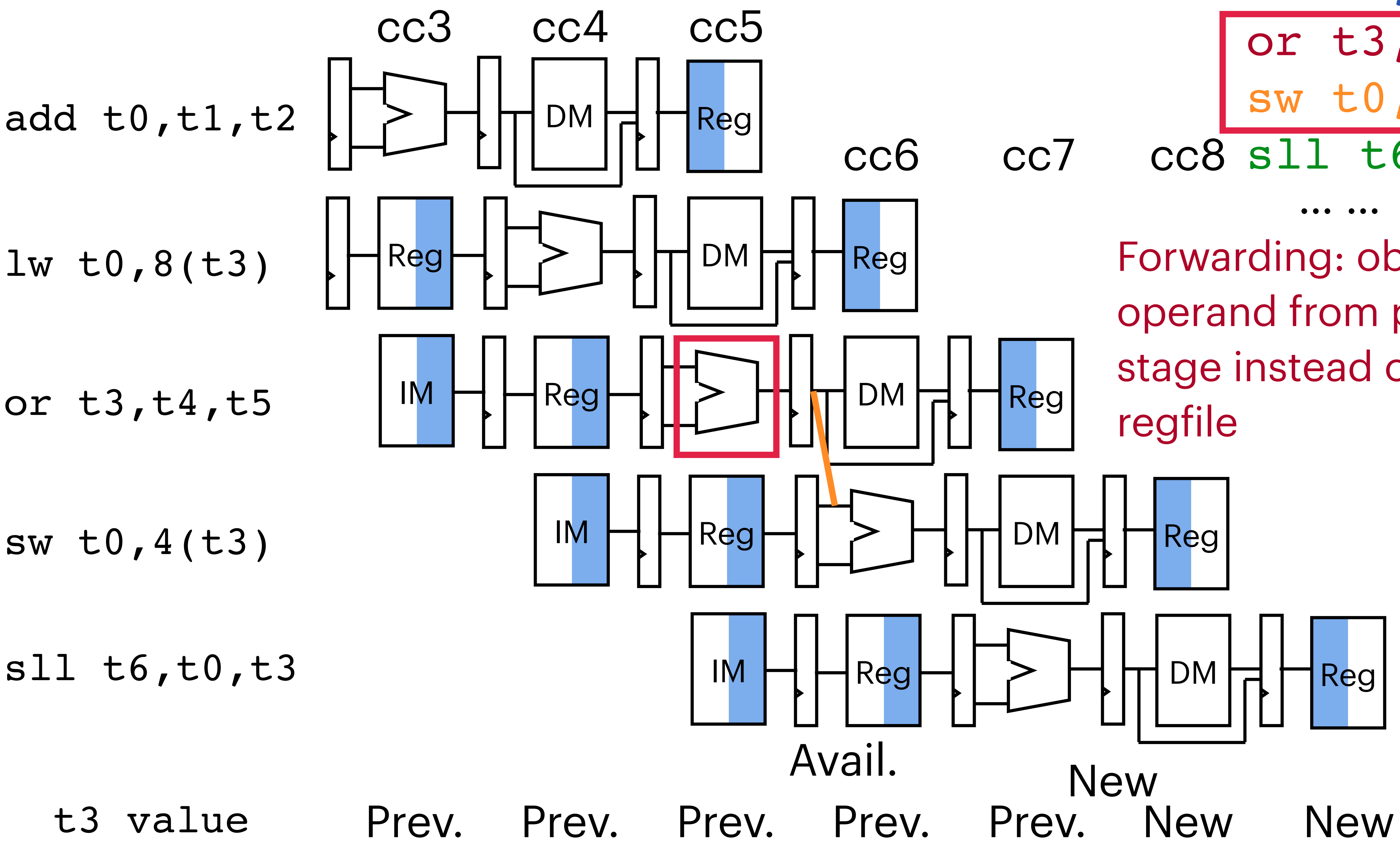
Solution 1:
Stalls: 3 `nops`



Solution 2: Forwarding/Bypassing

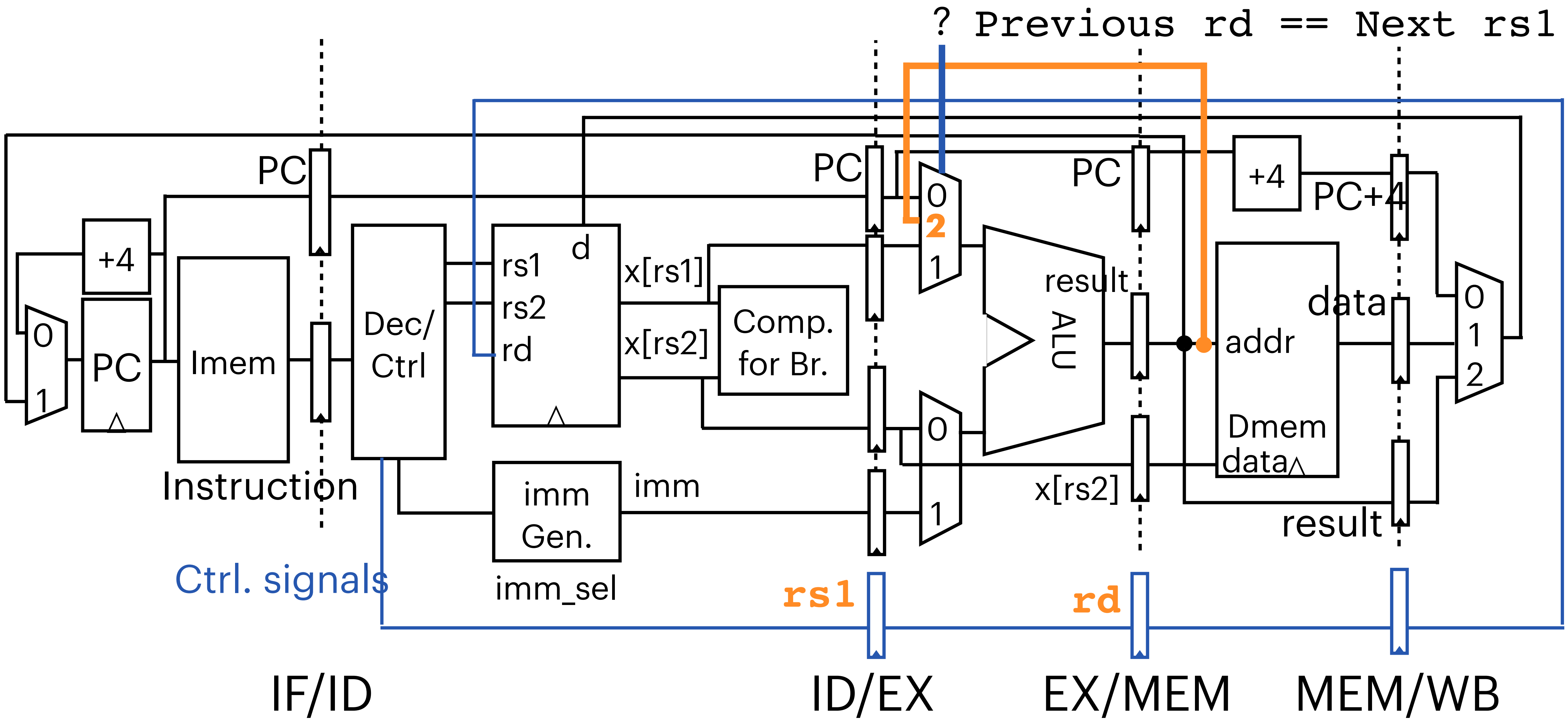
Starting from cc3

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
… …
```



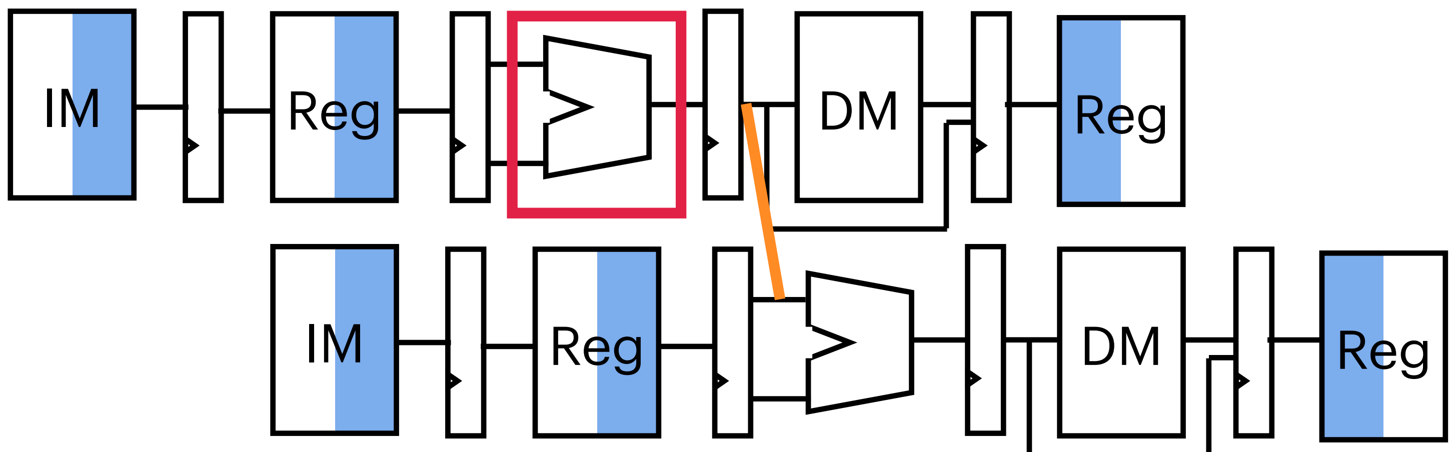
Forwarding: obtain operand from pipeline stage instead of the regfile

Adjust the Datapath: Add Extra Connections

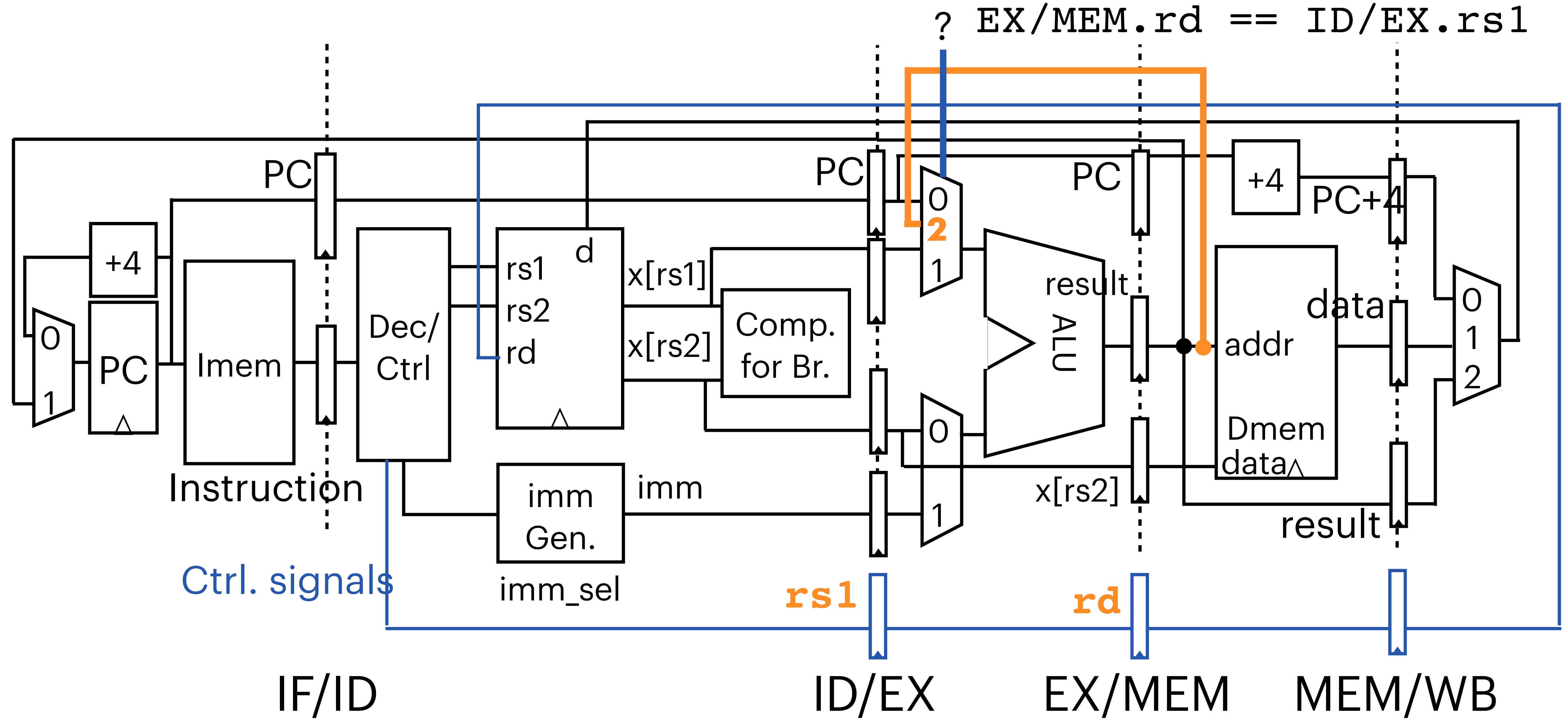


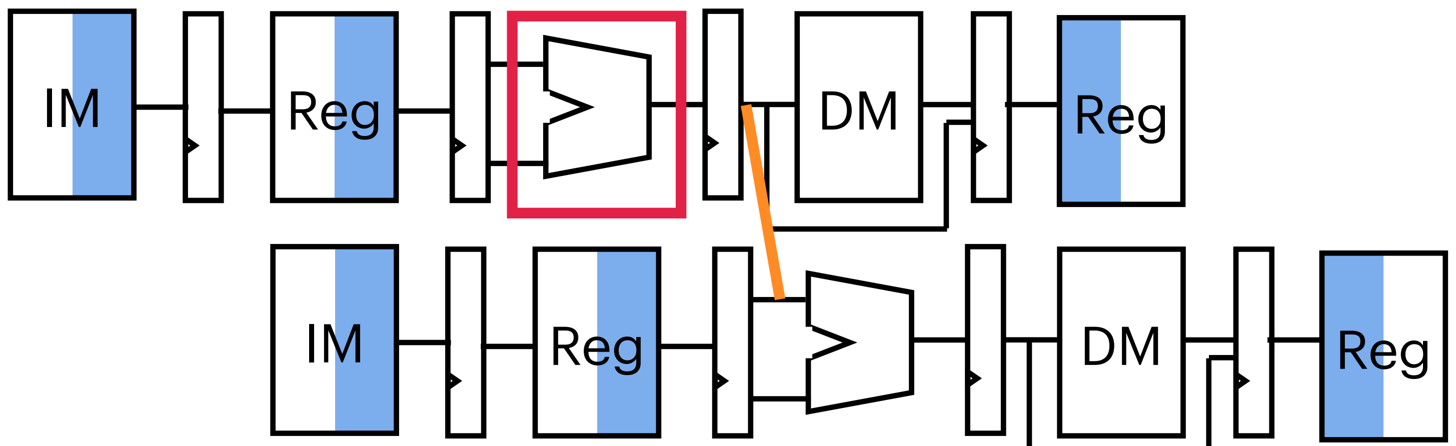
`or t3,t4,t5`

`sw t0,4(t3)`



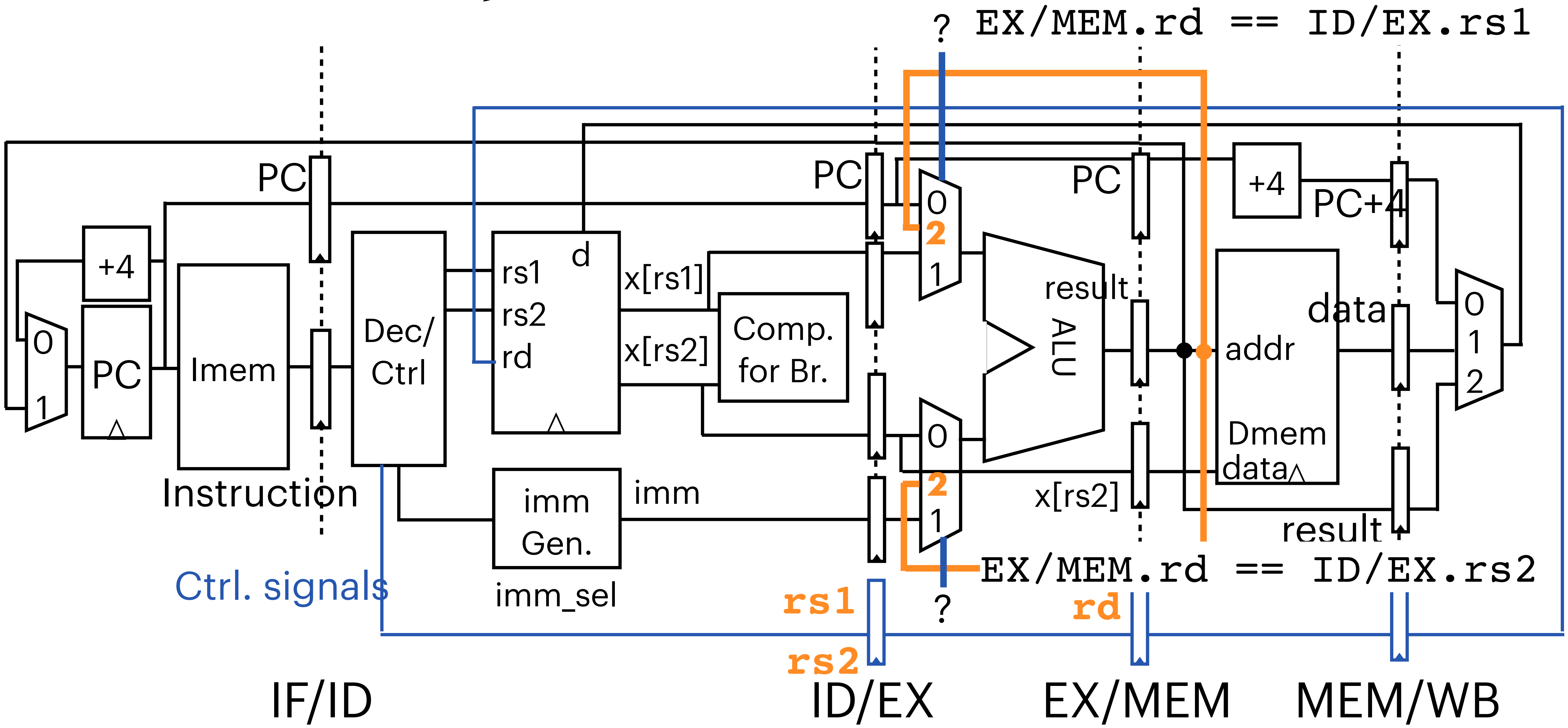
Adjust the Datapath



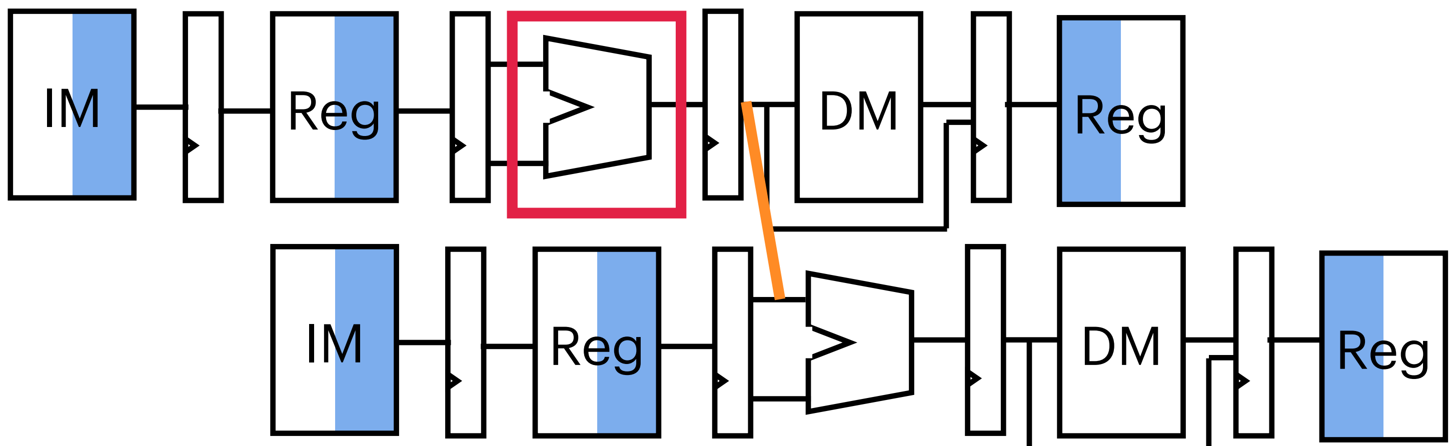


```
or t3,t4,t5

sw t0,4(t3)
```

Adjust the Datapath



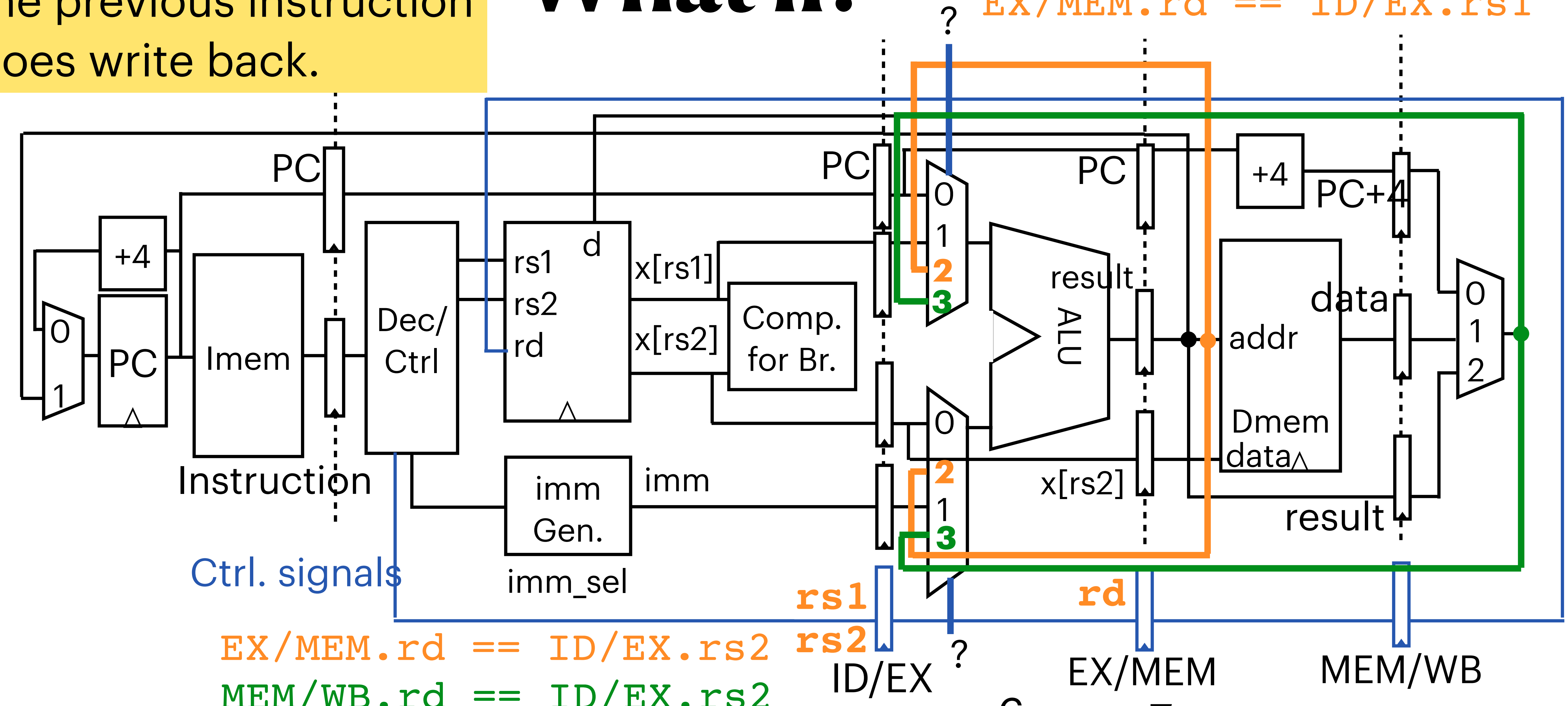
```
or t3,t4,t5

add t0,t2,t3
```



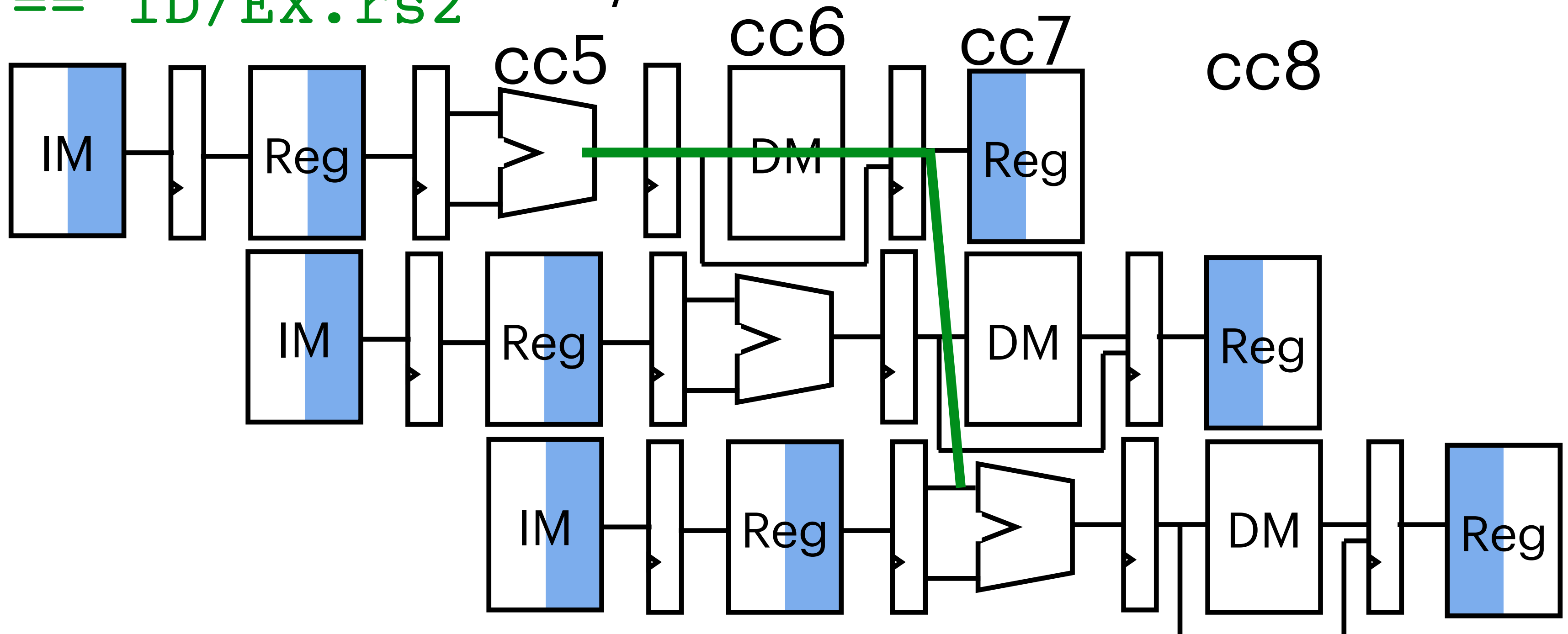
Also have to make sure the previous instruction does write back.

What if?

MEM/WB.rd == ID/EX.rs1
EX/MEM.rd == ID/EX.rs1



EX/MEM.rd == ID/EX.rs2
MEM/WB.rd == ID/EX.rs2

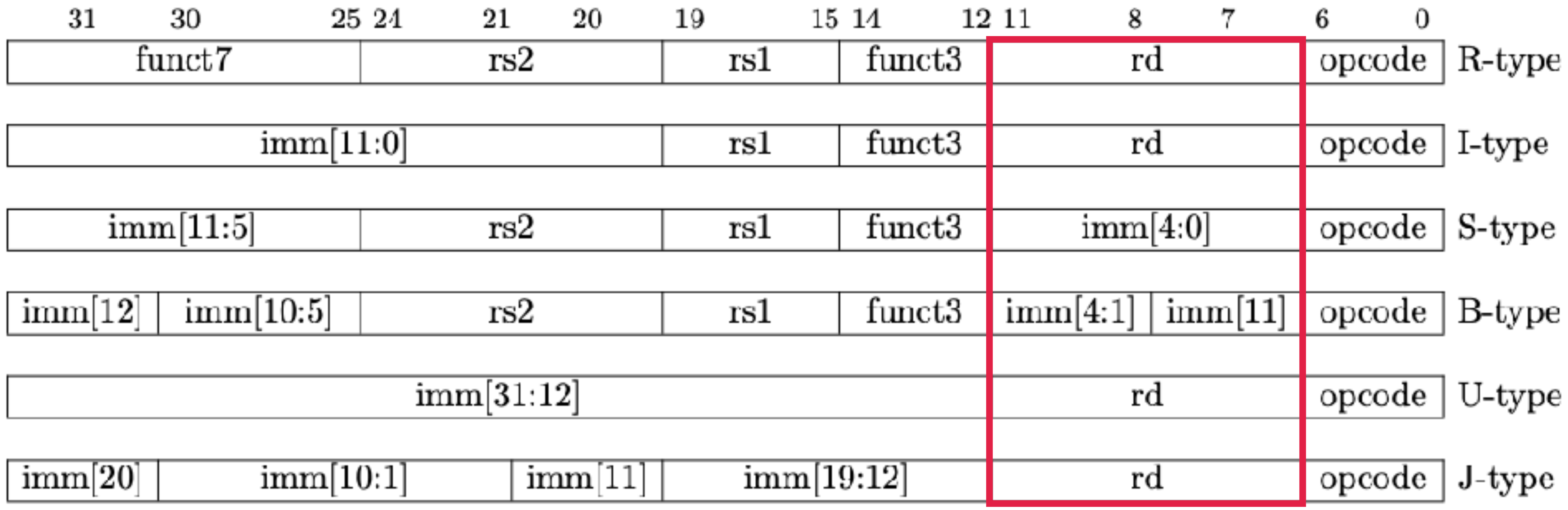


```
or t3,t4,t5
```

One irrelevant instruction
e.g. sll t1,t2,t6

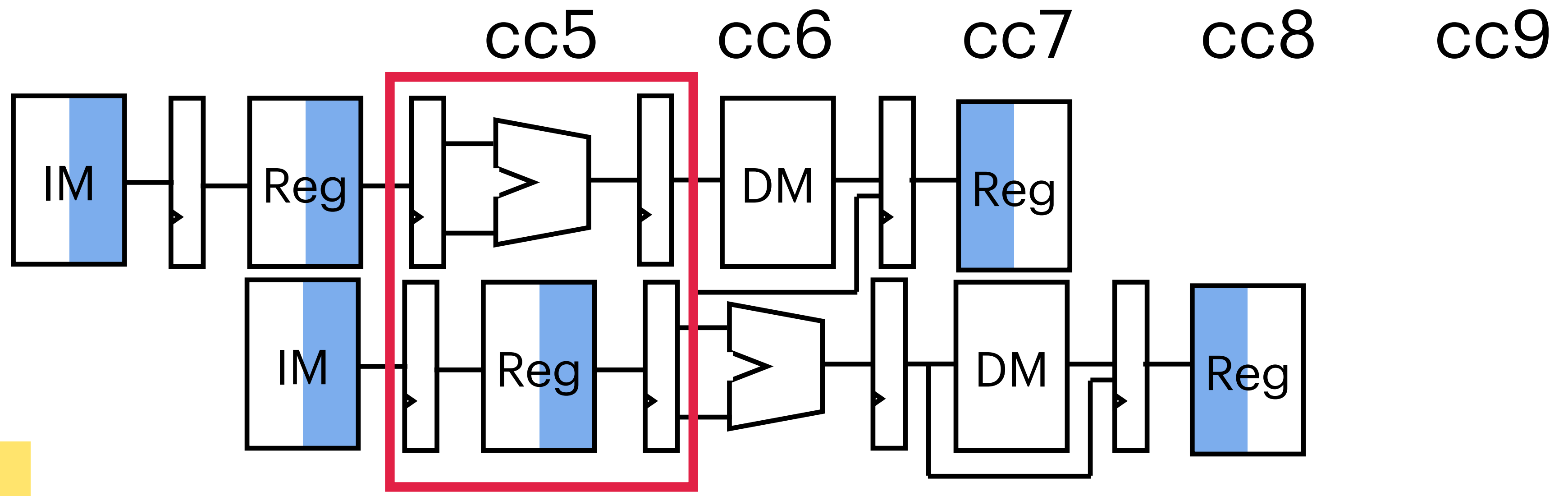
```
add t0,t2,t3
```

Extra Considerations



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

```
sw t0,28(t1)

add t0,t2,t3
```



Also have to make sure the previous instruction does write back.

EX/MEM.rd=28

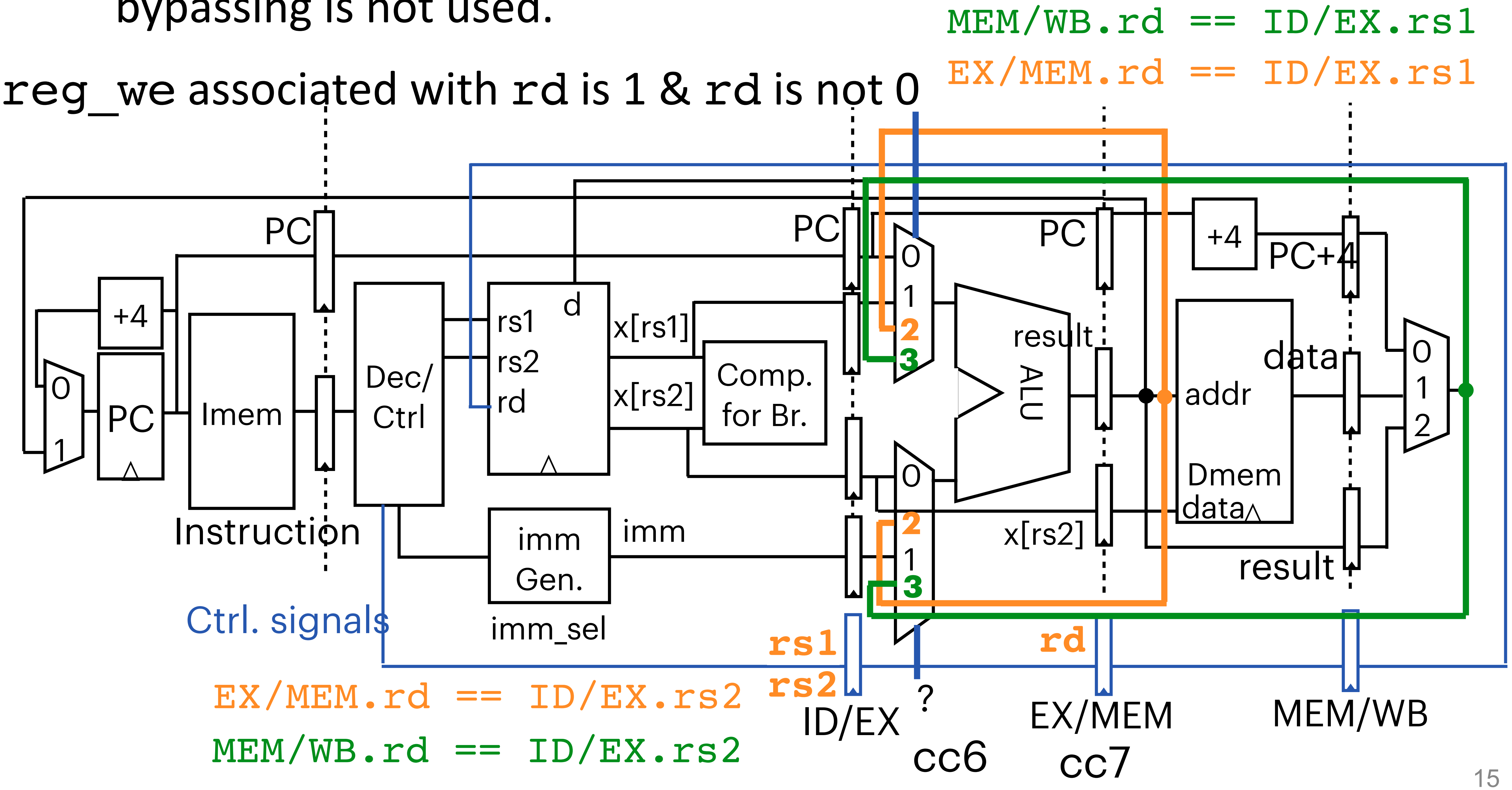
ID/EX.rs2=t3=x28=28

✓ False positive

What about x0?

x0 register stores zero, and its value won't be modified, i.e., if rd is x0, there will never be data hazard on x0. Thus, forwarding/bypassing is not used.

reg_we associated with rd is 1 & rd is not 0



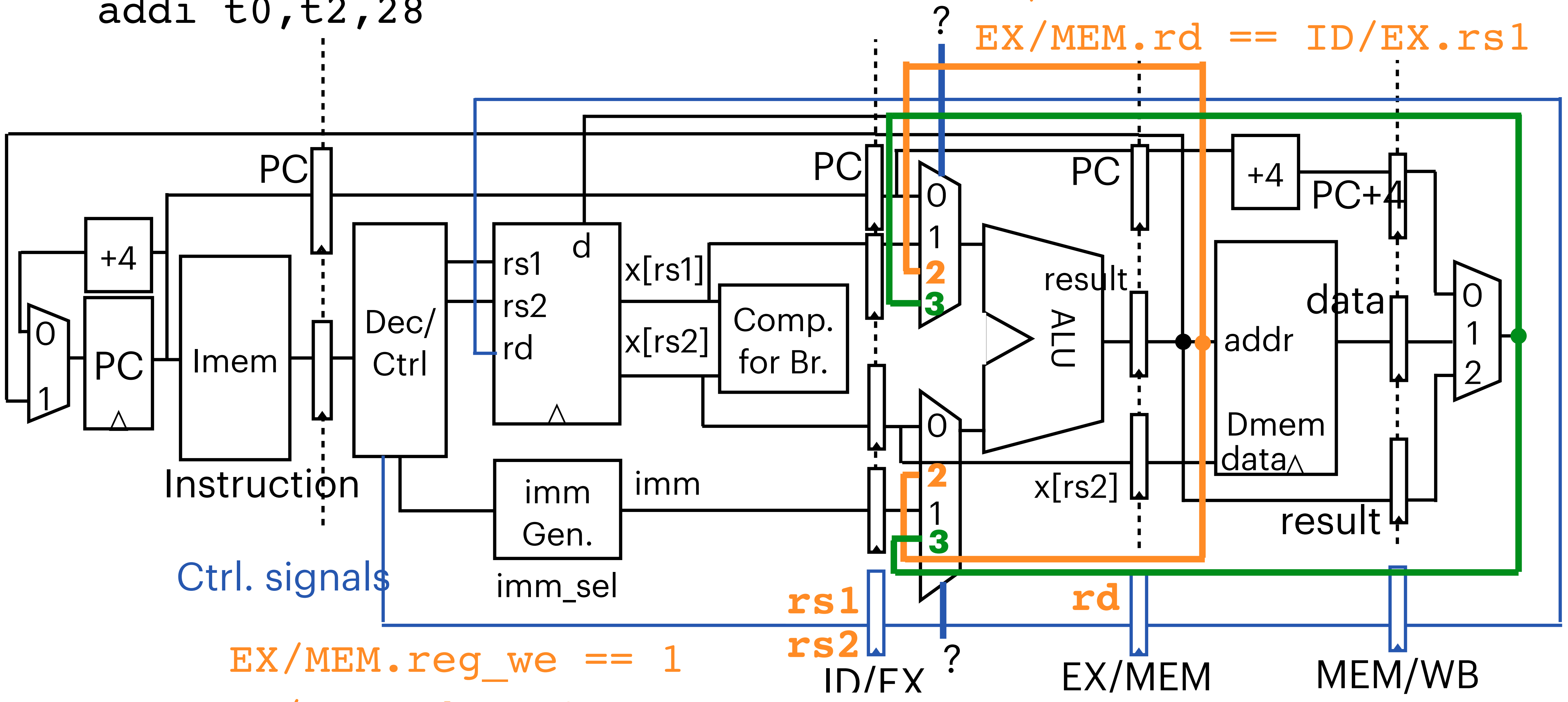
Extra Considerations

What about the input operands?

Do we have to check either?

```
add t3,t4,t5
addi t0,t2,28
```

```
MEM/WB.reg_we == 1
MEM/WB.rd !=0
MEM/WB.rd == ID/EX.rs1
EX/MEM.reg_we == 1
EX/MEM.rd !=0
EX/MEM.rd == ID/EX.rs1
```



```
EX/MEM.reg_we == 1
EX/MEM.rd != 0
EX/MEM.rd == ID/EX.rs2
```

```
MEM/WB.reg_we == 1
MEM/WB.rd != 0
MEM/WB.rd == ID/EX.rs2
```

Extra Considerations

What about the input operands?

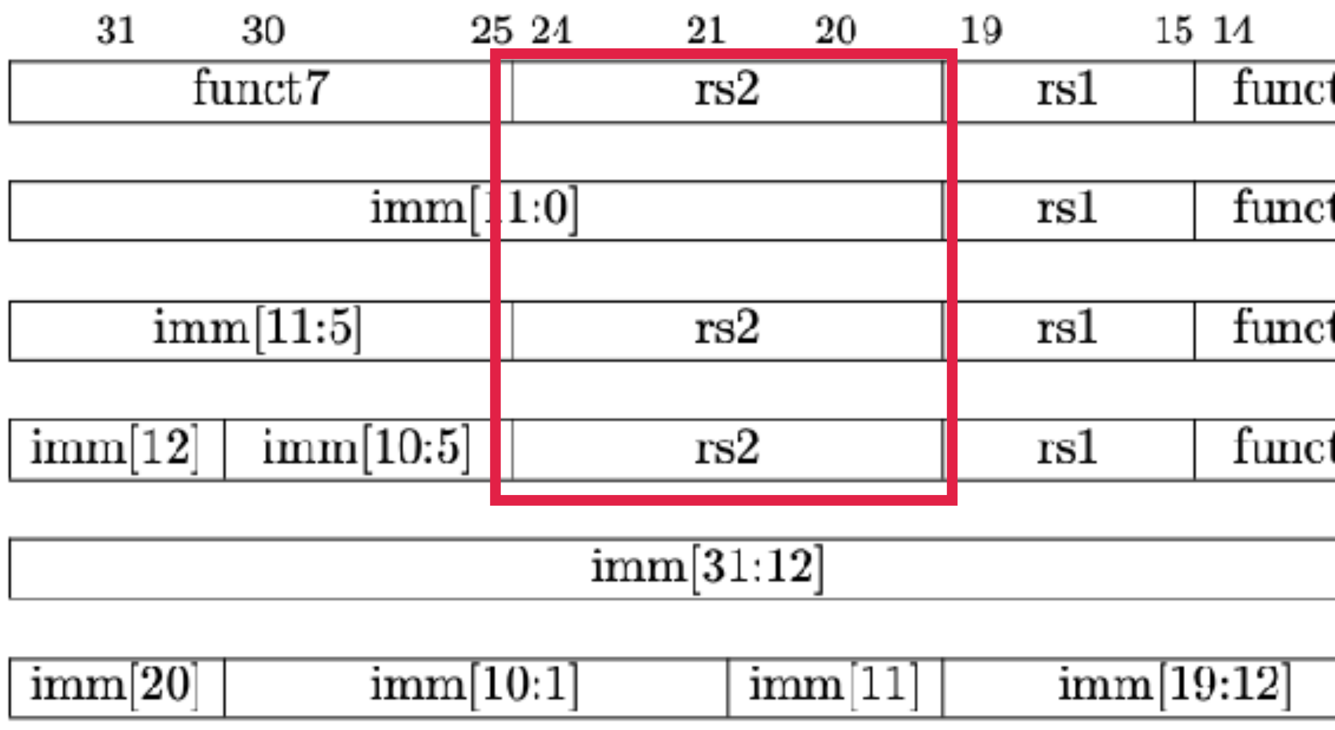
Do we have to check either?

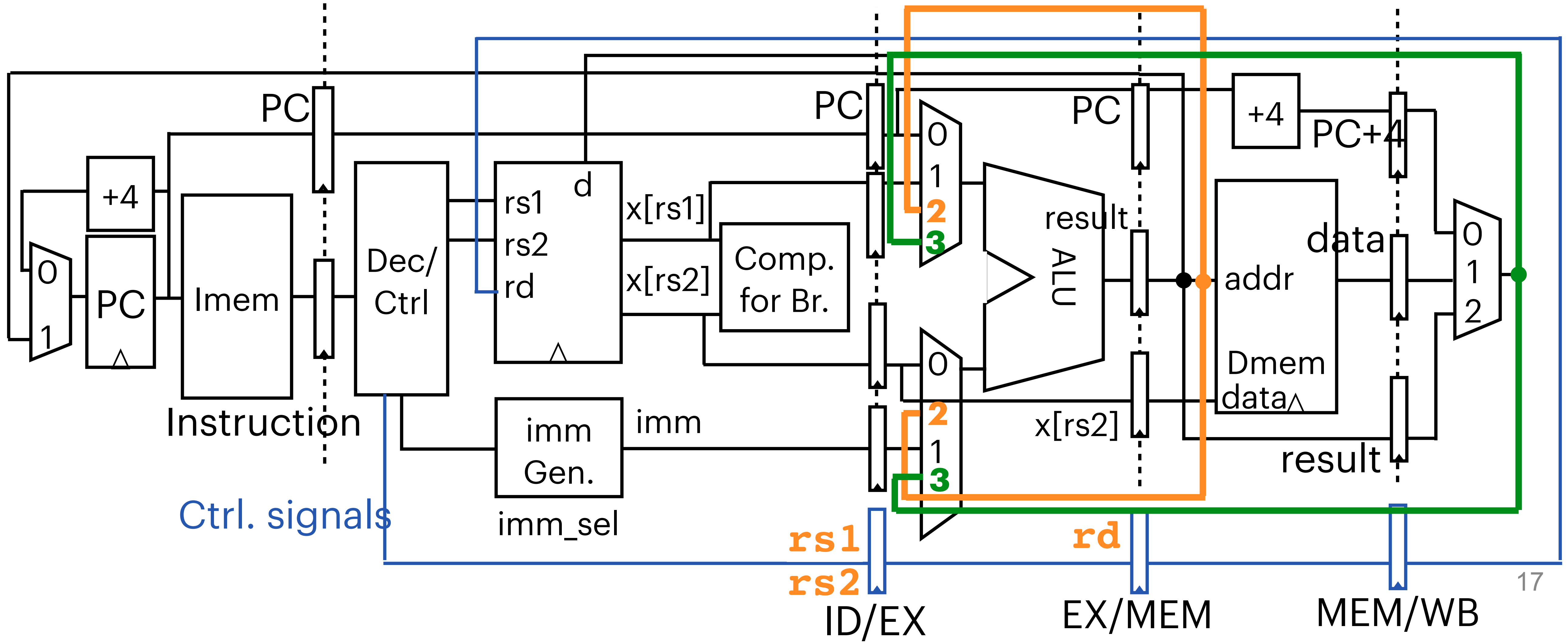
```
add t3,t4,t5
addi t0,t2,28
```

EX/MEM.rd=28

ID/EX.rs2=t3=x28=28

False positive





Extra Considerations

What about the input operands?

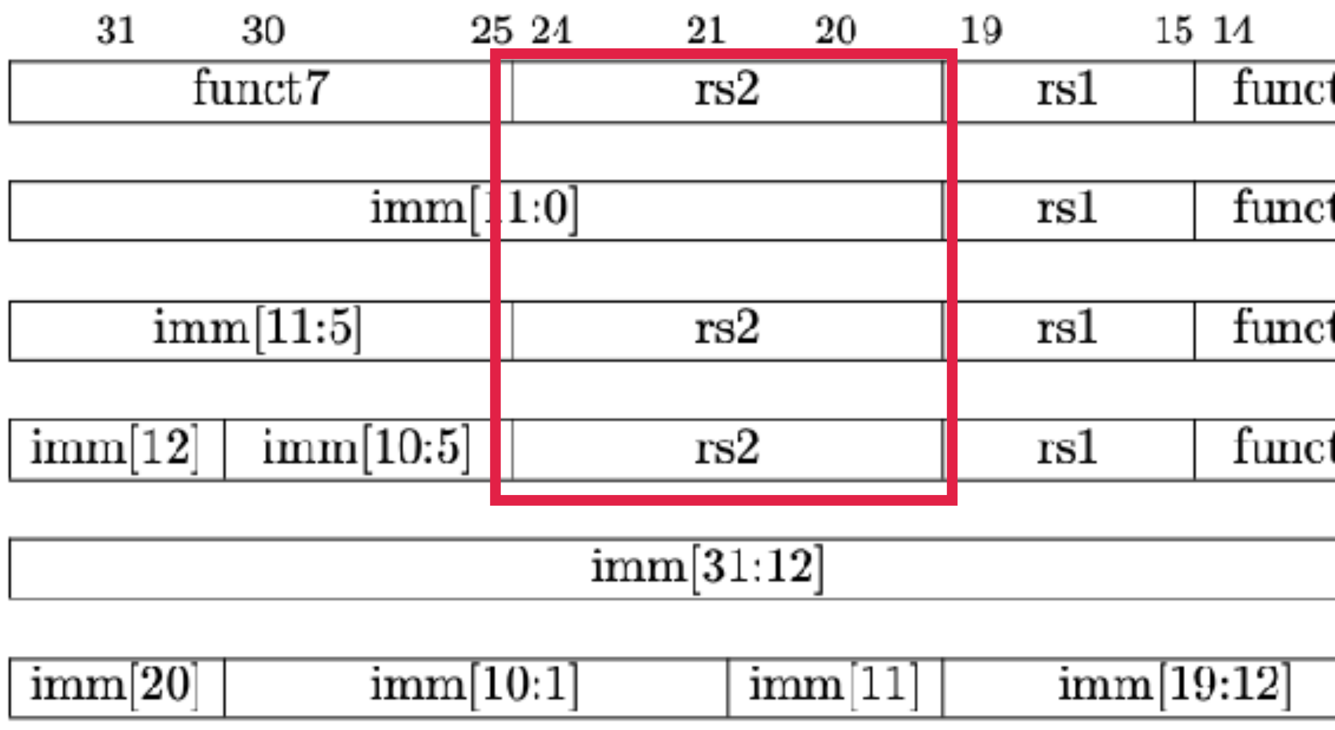
Do we have to check either?

```
add t3,t4,t5
addi t0,t2,28
```

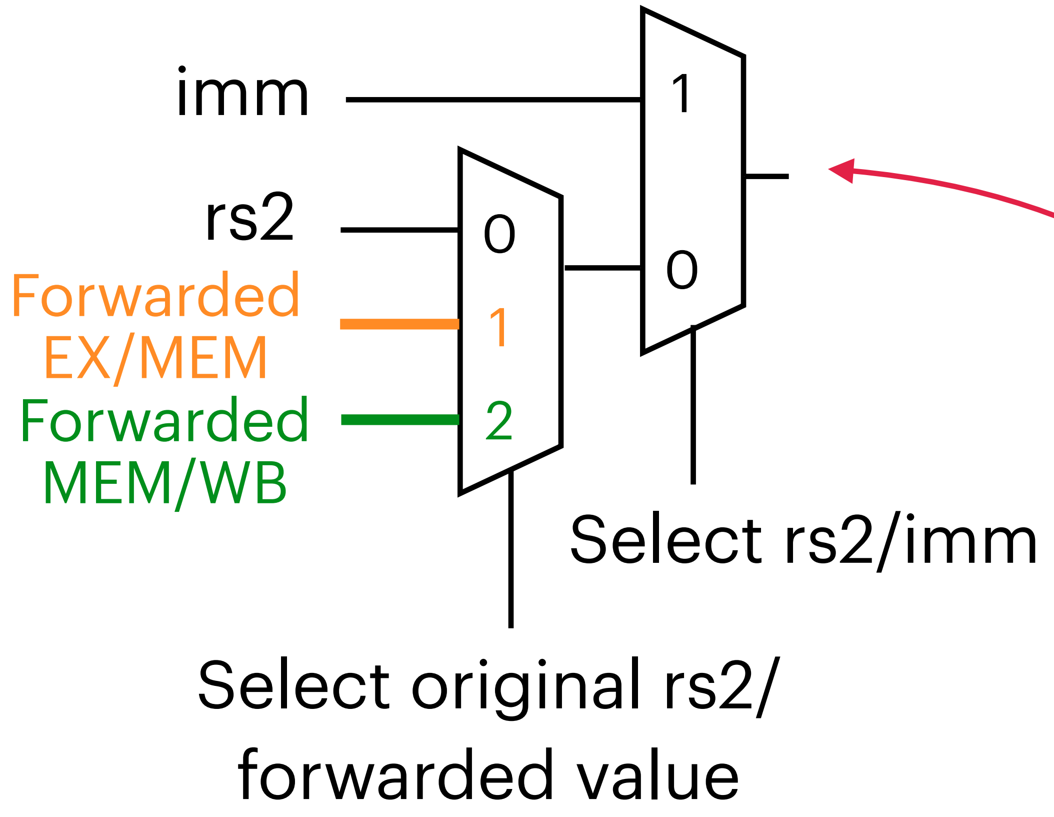
EX/MEM.rd=28

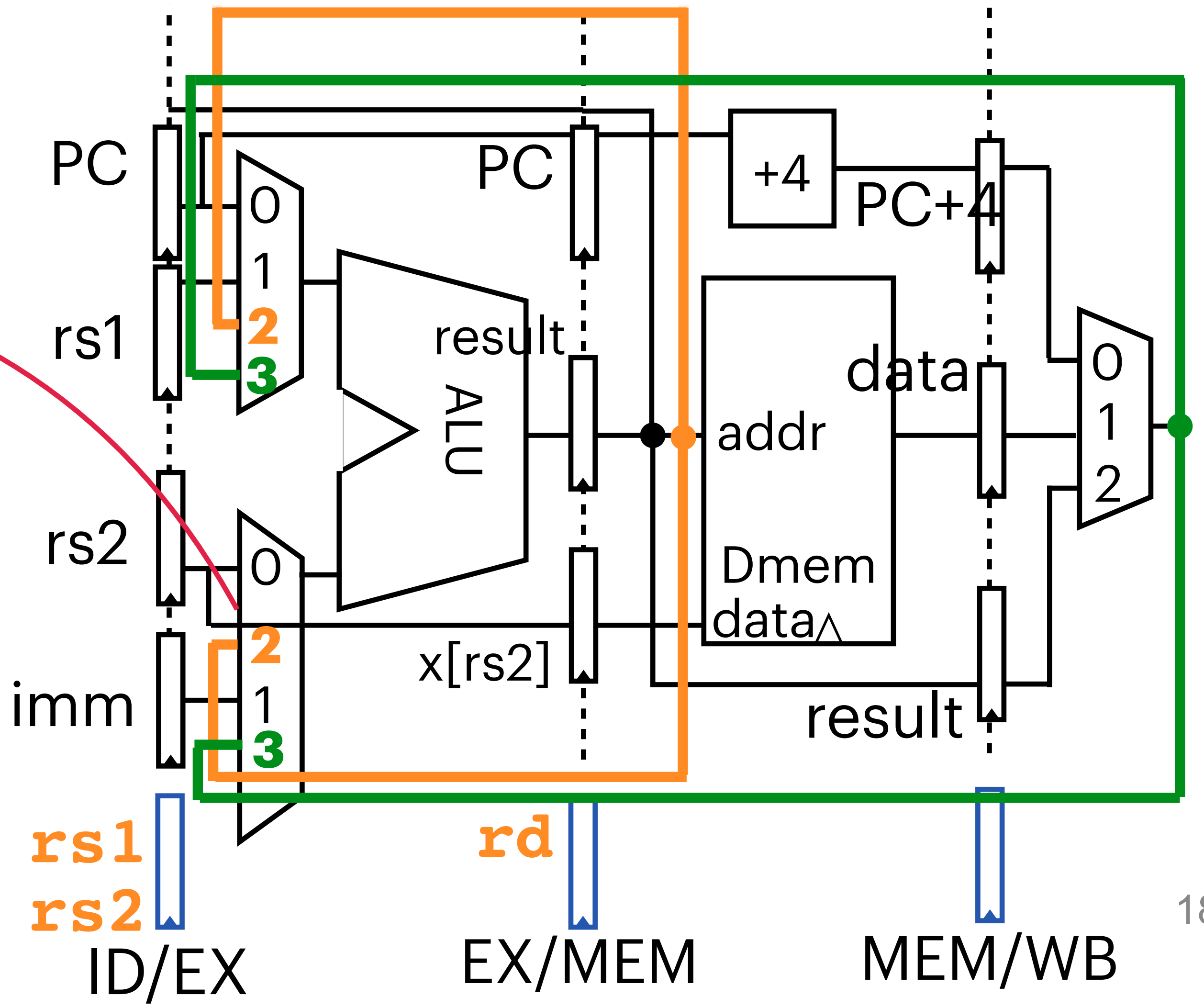
ID/EX.rs2=t3=x28=28

✓ False positive



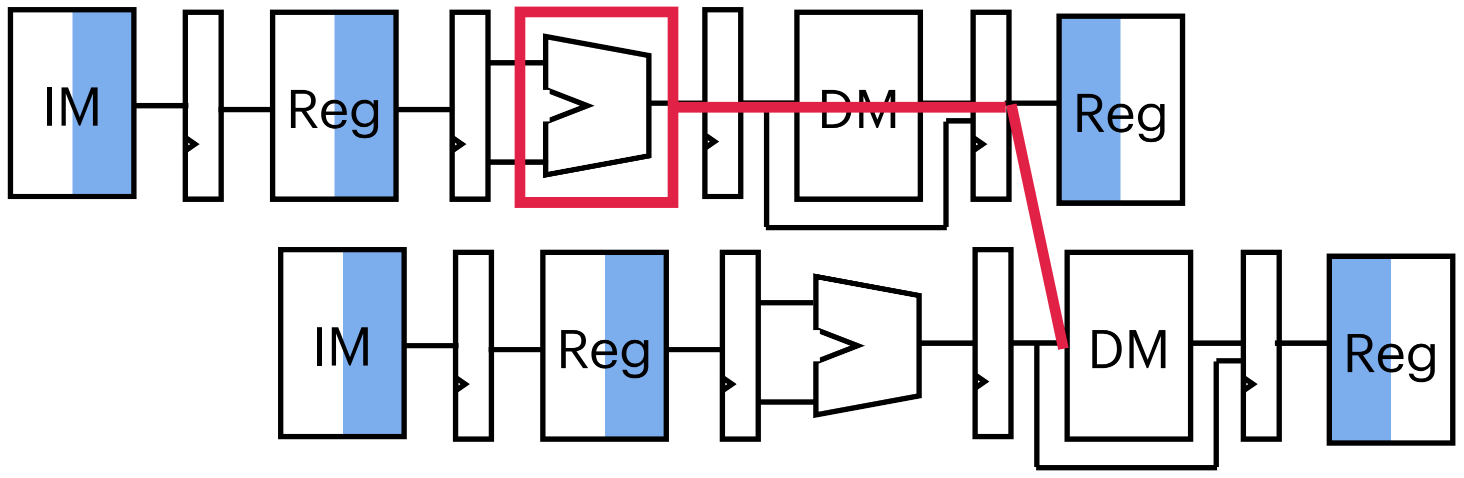
Give priority to rs2/imm judgement

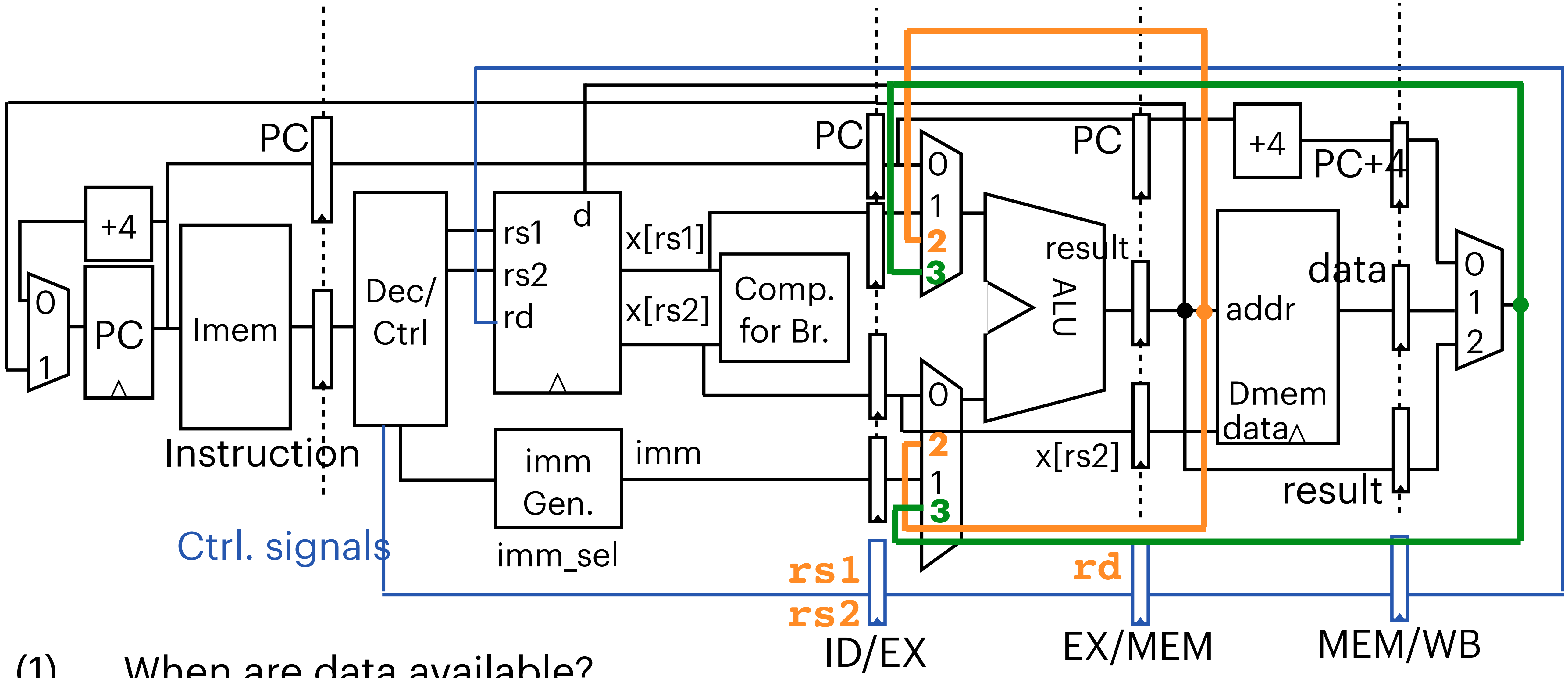




Forwarding also Resolves `sw` Hazards

```
add t3,t1,t2

sw t3,t4,t5
```



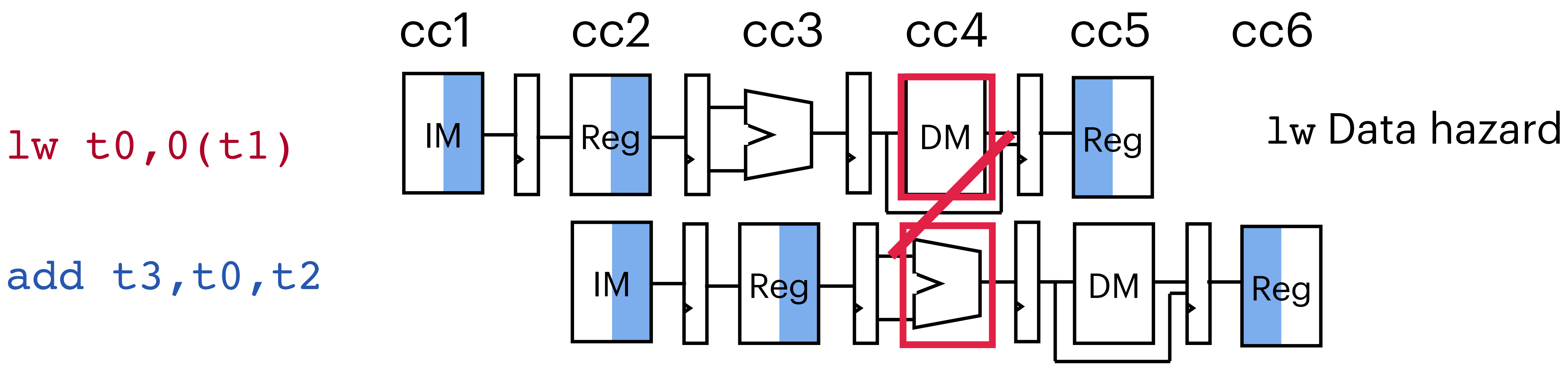


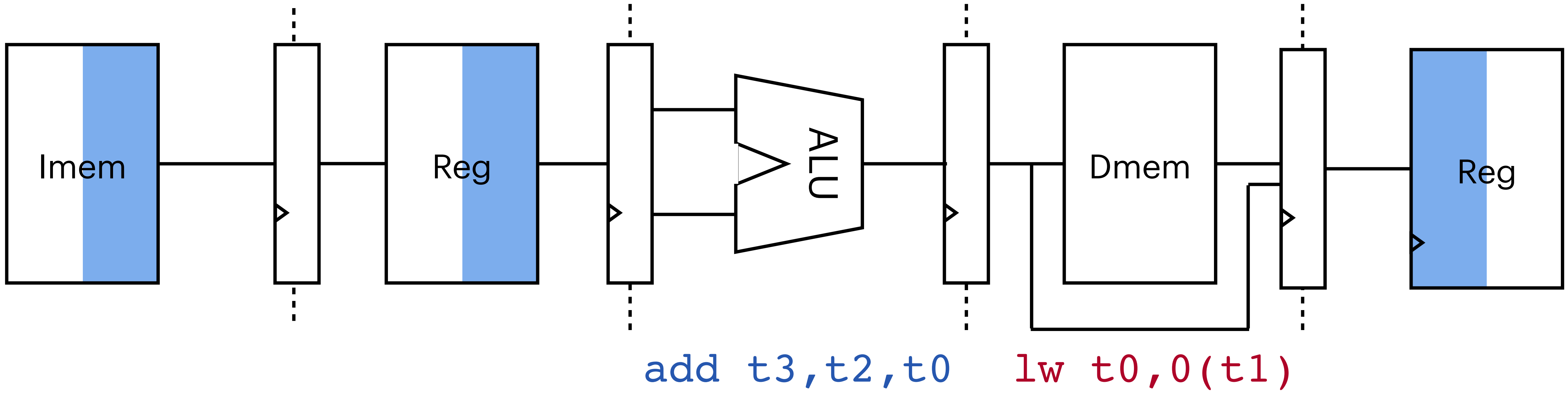
(1) When are data available?

(2) When are data consumed?

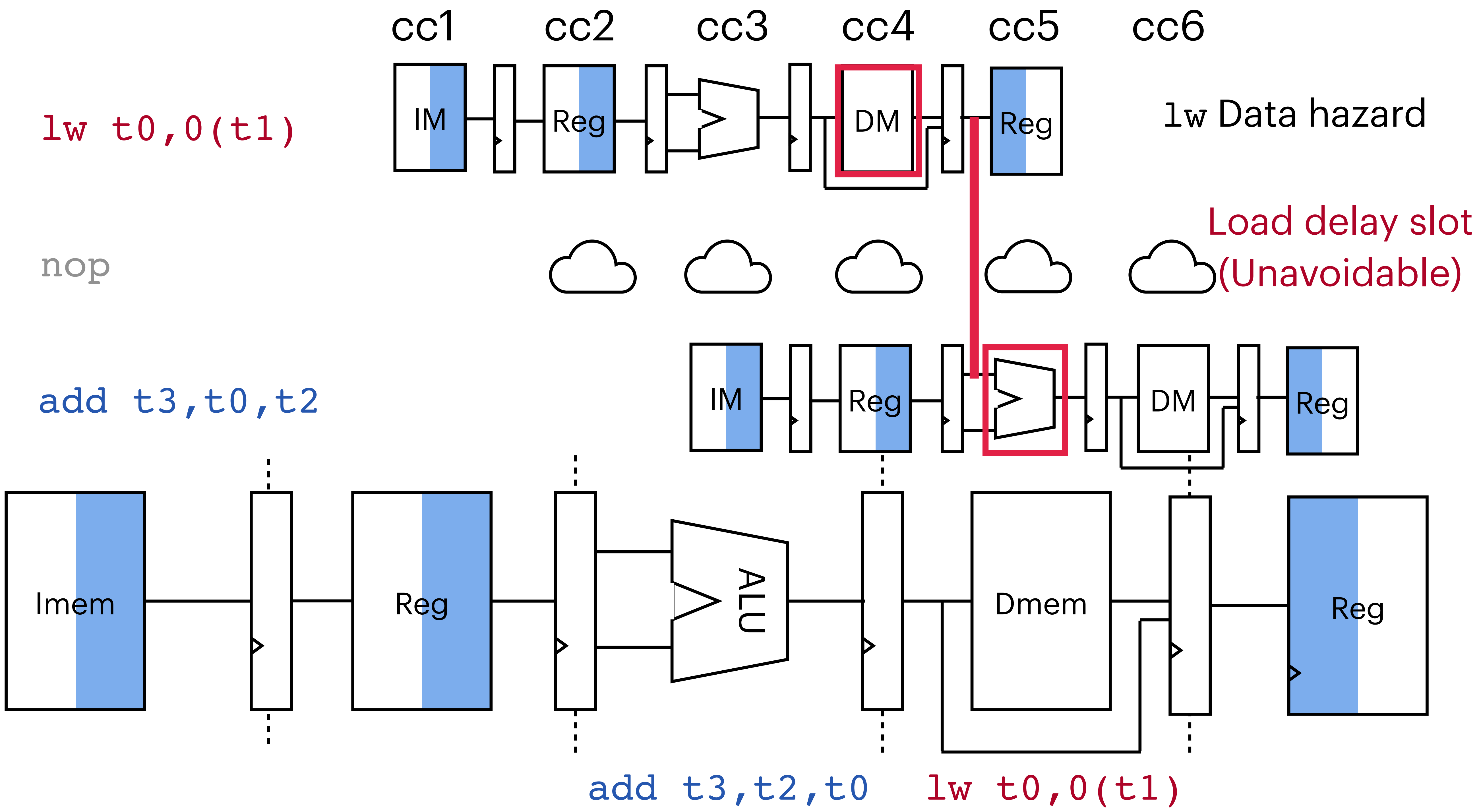
Try yourself!

Extra Consideration on `lw`





Extra Consideration on `lw`



Code Scheduling to Avoid Stalls

Assume C code: `D=A+B; E=A+C`

A, B, C addresses are `0(t0), 4(t0), 8(t0)`
D, E addresses are `12(t0), 16(t0)`

Naïve RISC-V compiler

```
lw t1,0(t0)
lw t2,4(t0)
add t3,t1,t2
sw t3,12(t0)
lw t4,8(t0)
add t5,t1,t4
sw t5,16(t0)
```

A smart compiler reorders the instructions:

```
lw t1,0(t0)
lw t2,4(t0)
lw t4,8(t0)
add t3,t1,t2
sw t3,12(t0)
add t5,t1,t4
sw t5,16(t0)
```

Q: How many clk cycles?

Code Scheduling to Avoid Stalls

Code scheduling: With knowledge of the underlying CPU pipeline, the compiler reorders code to improve performance.

Up-to-Now: Data Hazards

A hazard is a situation in which a planned instruction cannot execute in the "proper" clock cycle.

1. Structural hazard:
   - Hardware does not support access across multiple instructions in the same cycle

2. **Data hazard**:
   - Instructions have data dependency
   - Occurs when an instruction reads a register before a previous instruction has finished writing to that register
   - Solution 1: stall
   - Solution 2: forwarding/bypassing
   - Solution 3: Code scheduling

Control Hazards

A hazard is a situation in which a planned instruction cannot execute in the “proper” clock cycle.

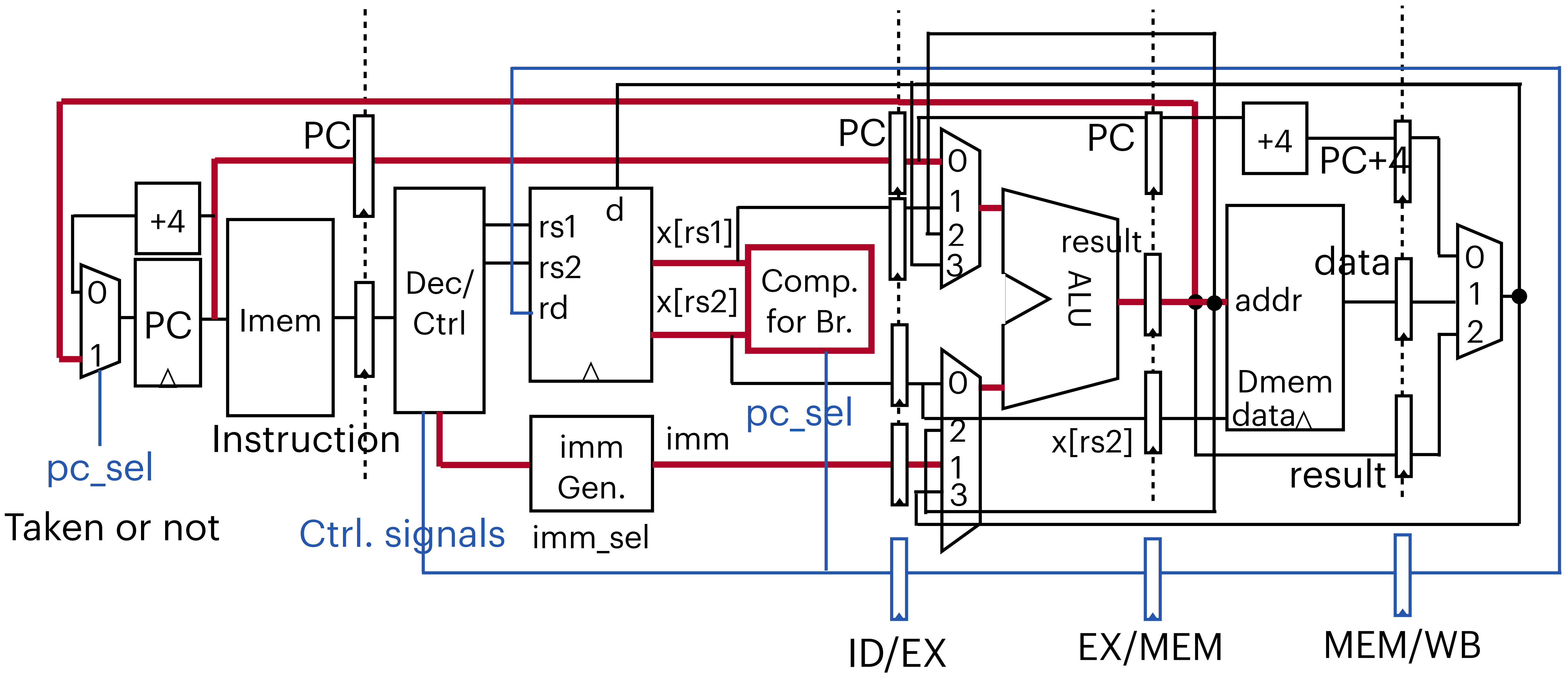
1. Structural hazard:
    - Hardware does not support access across multiple instructions in the same cycle

2. Data hazard:
    - Instructions have data dependency
    - Occurs when an instruction reads a register before a previous instruction has finished writing to that register

3. Control hazard:
    - Flow of execution depends on previous instruction

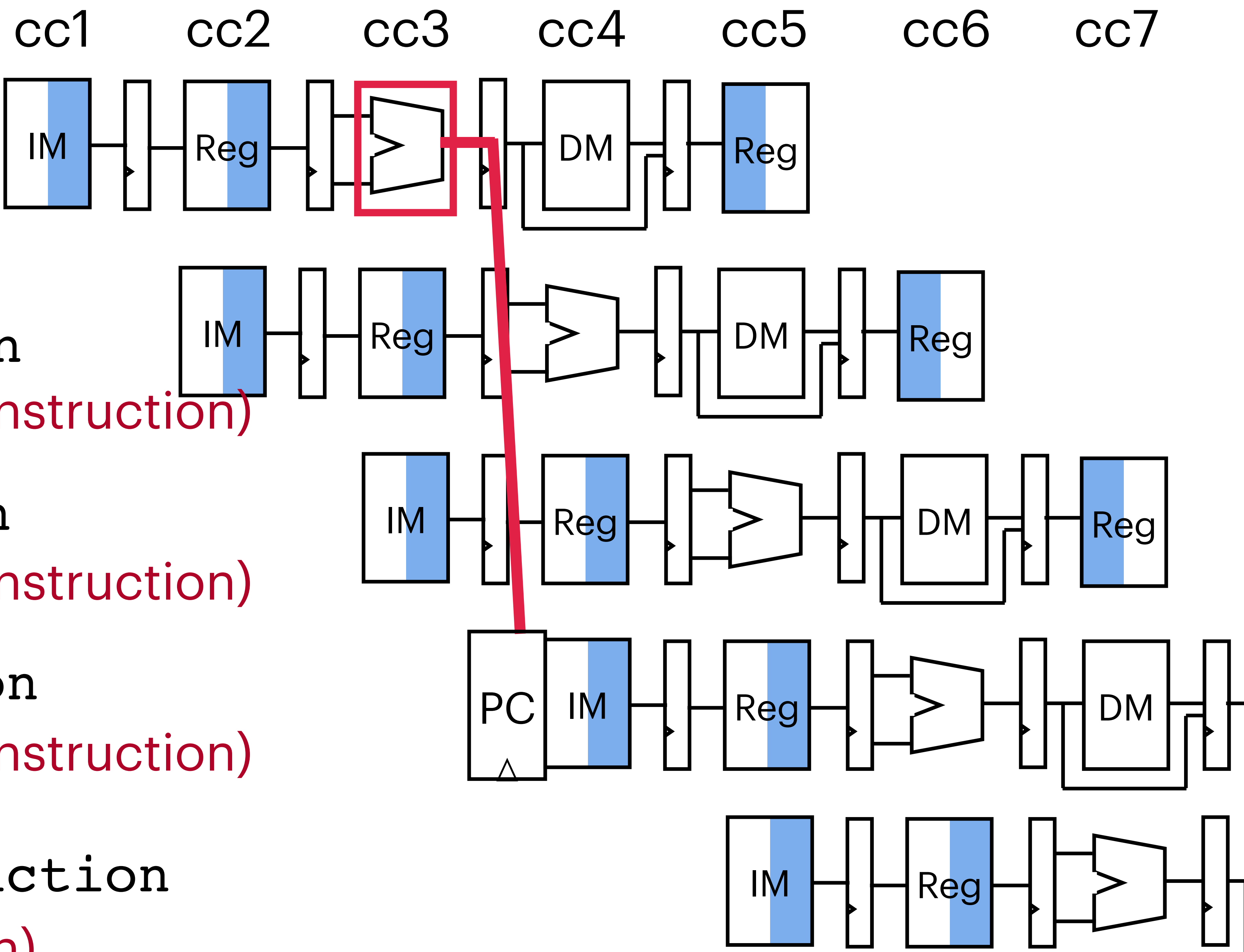
Control Hazard Example: Branch

```
beq rs1,rs2,imm/label
```



```
1.rs1/rs2 comparison
2.PC = PC + imm
```

Control Hazard Example: Branch



`beq rs1,rs2,imm`

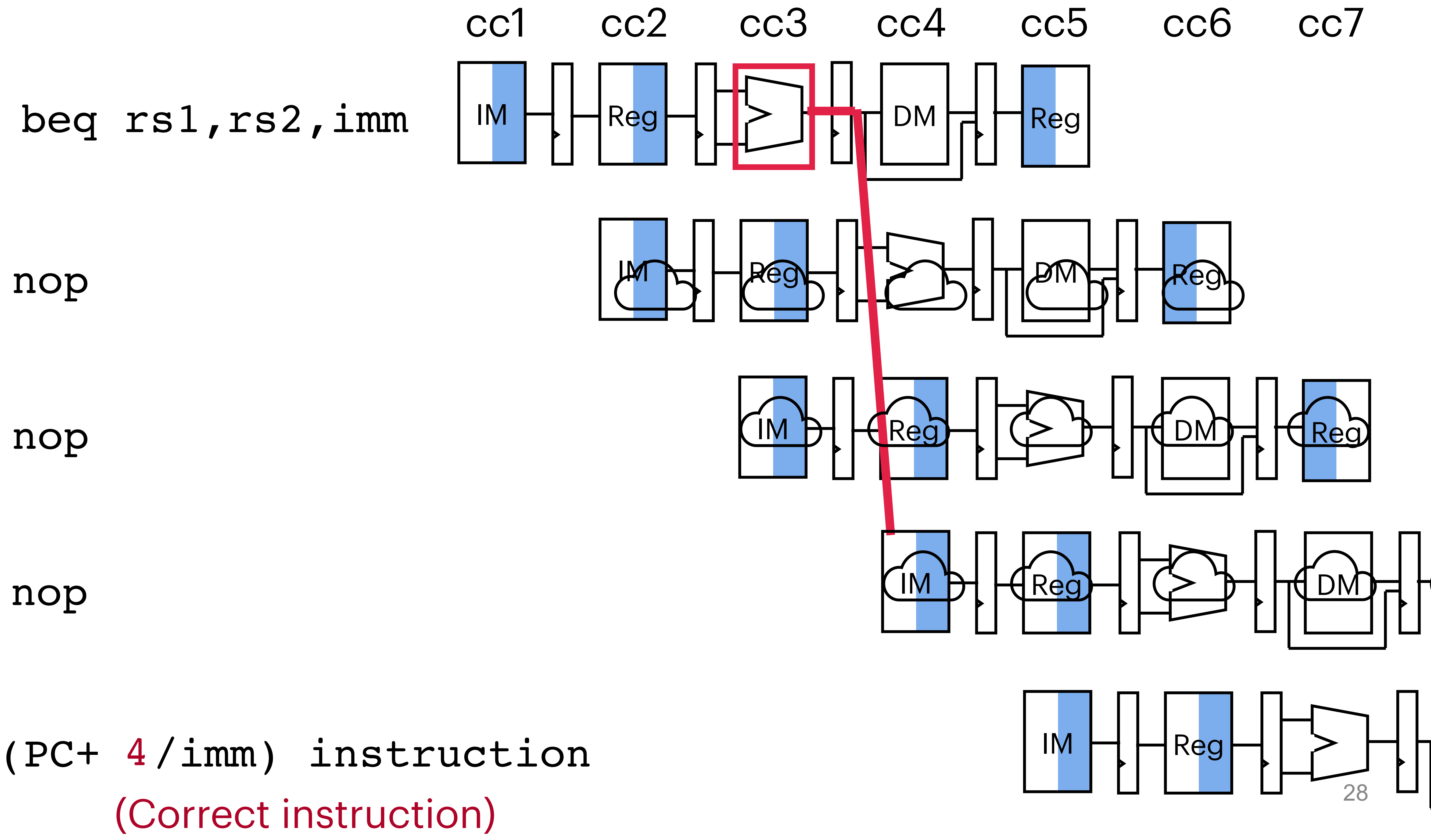
`(PC+4) instruction`
(Might not be the right instruction)

`(PC+8) instruction`
(Might not be the right instruction)

`(PC+12) instruction`
(Might not be the right instruction)

`(PC+16/imm) instruction`
(Correct instruction)

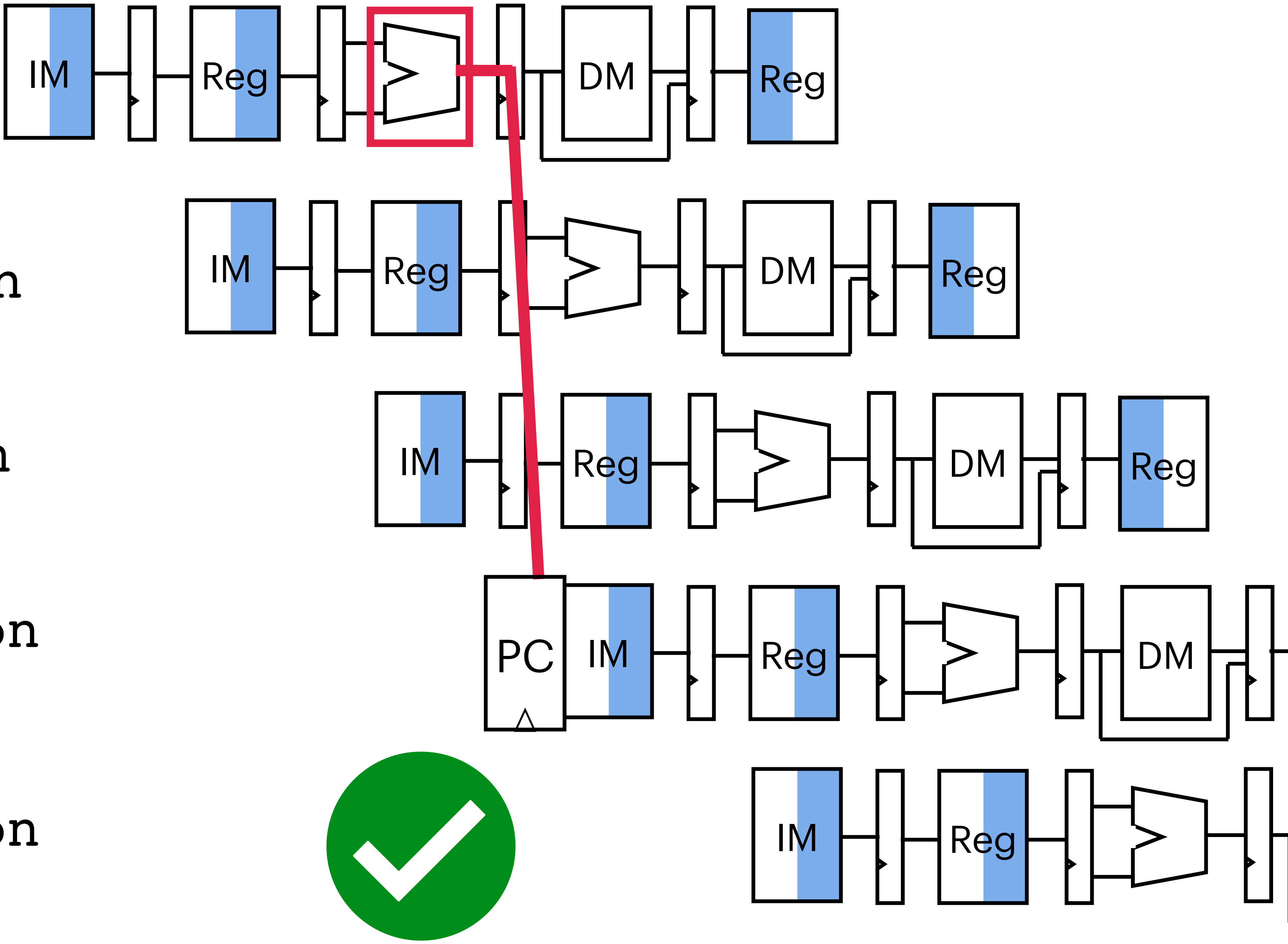
Control Hazard Example: Branch



Solution 1: Branch Prediction

- For example, a naïve predictor: always guess branch not taken.

```
beq rs1,rs2,imm
 (x[rs1]!=x[rs2])

(PC+4) instruction

(PC+8) instruction

(PC+12) instruction

(PC+16) instruction
```



Solution 1: Branch Prediction

- For example, a naïve predictor: always guess branch not taken.

`beq rs1,rs2,imm`
`(x[rs1]==x[rs2])`

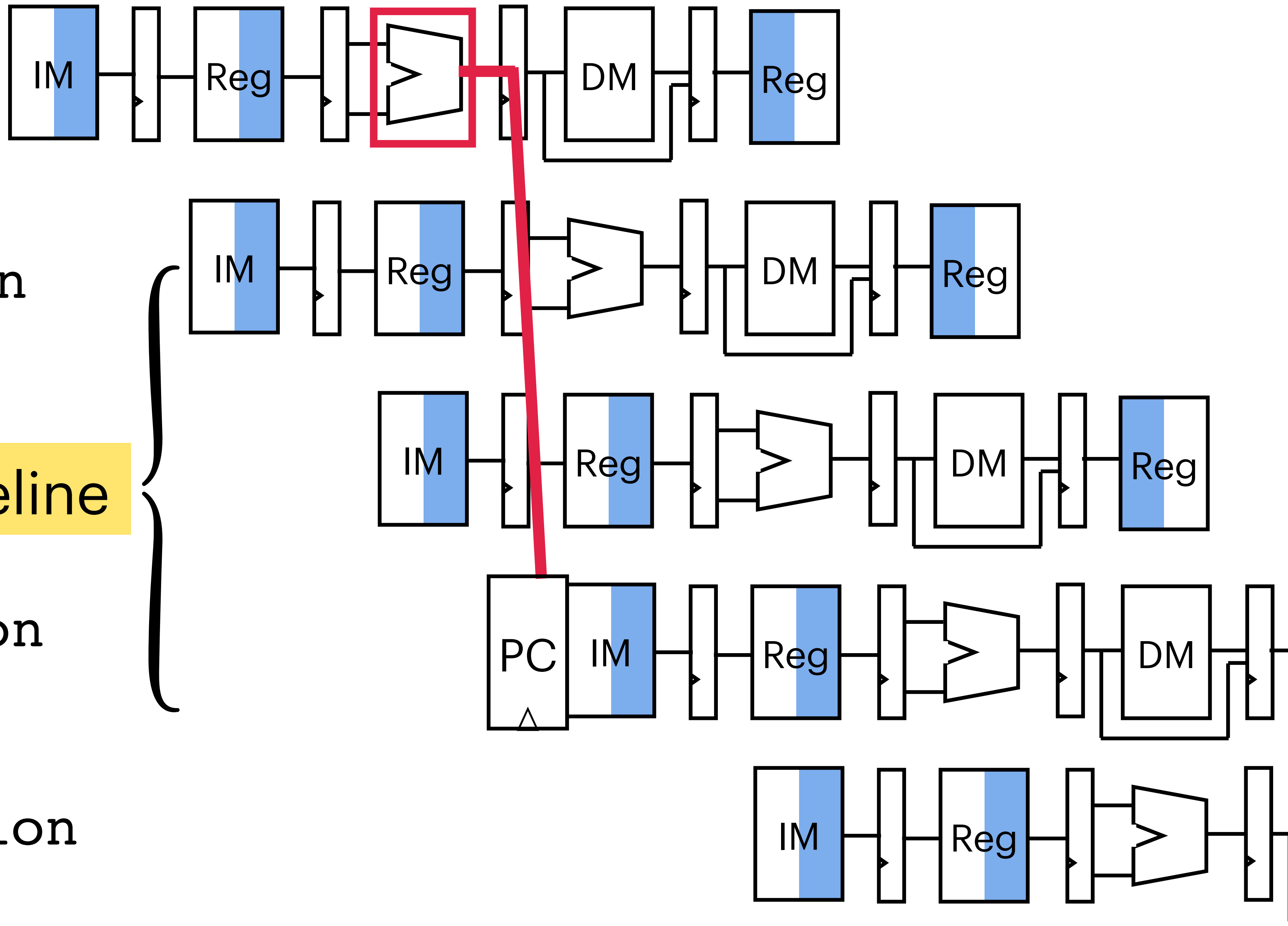
`(PC+4) instruction`

`(PC+8) in`

Flush pipeline

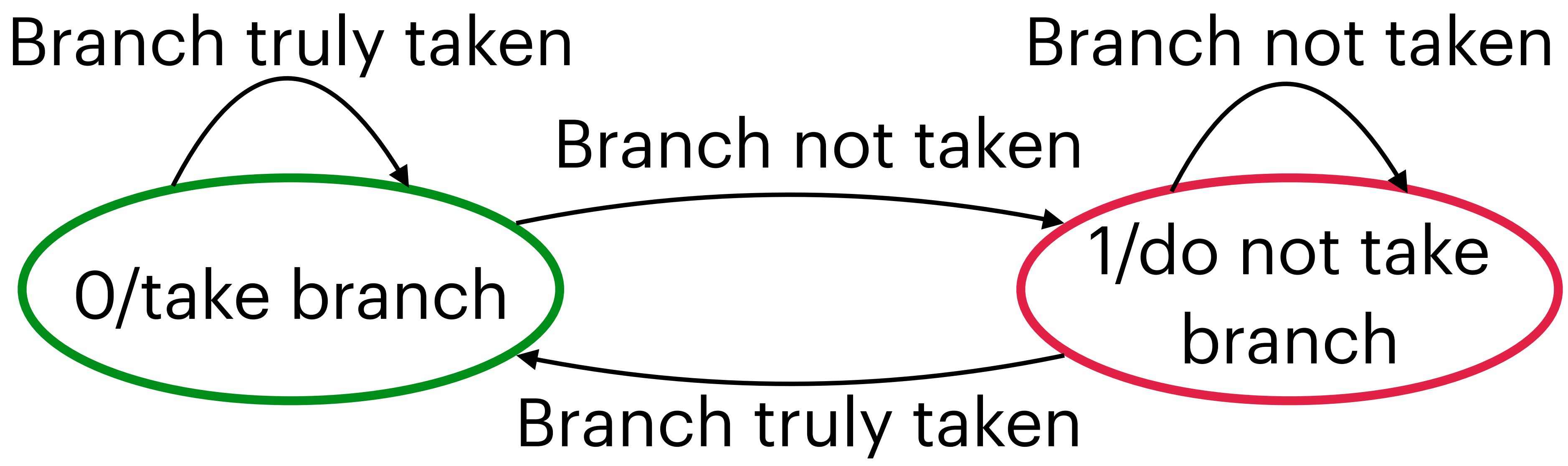
`(PC+12) instruction`

`(PC+imm) instruction`

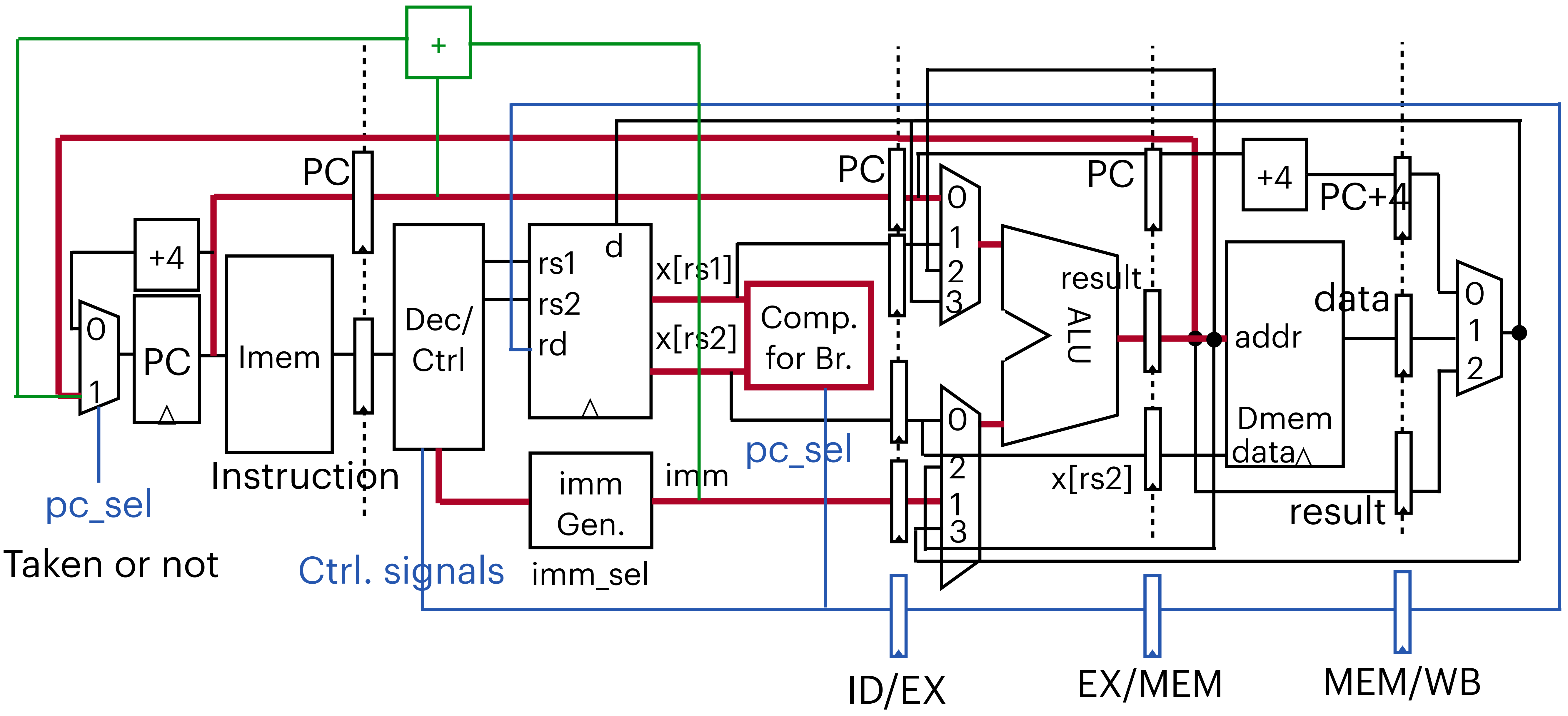


Solution 1: Branch Prediction

- Other scheme: dynamic branch prediction using runtime info
  - predict based on history
  - Build an FSM to perform the prediction



Solution 2: Modify the Hardware



Pipeline Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance/hardware modifications

Increasing Processor Performance

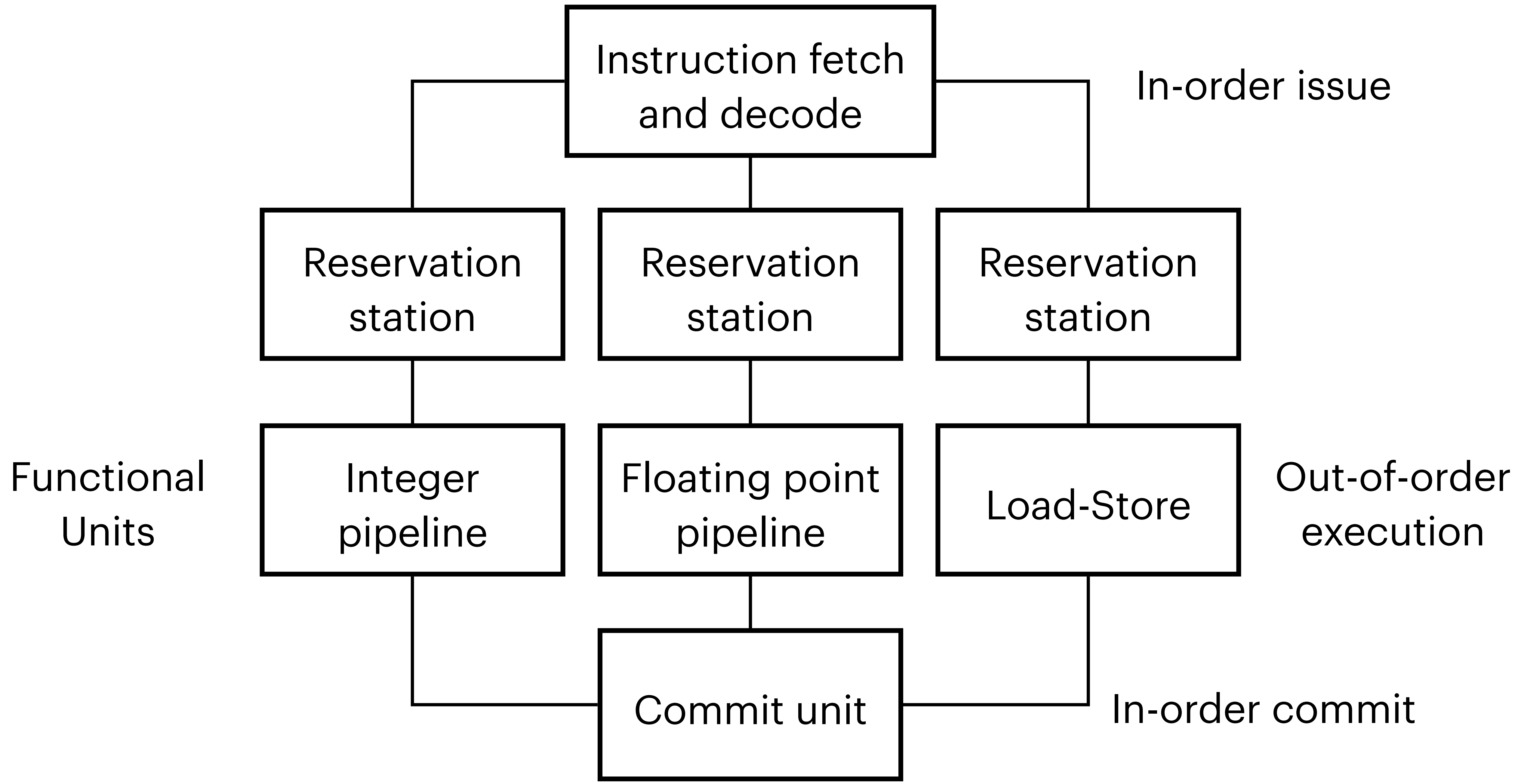
- Pipelining
  - "Overlap" instruction execution
  - Deeper pipeline: 5 => 10 => 15 stages
    - Less work per stage → shorter clock cycle
    - But more potential for hazards
    - Multi-issue processor
- CPI measurement: benchmark to obtain time/program

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

Greater Instruction-Level Parallelism (ILP)

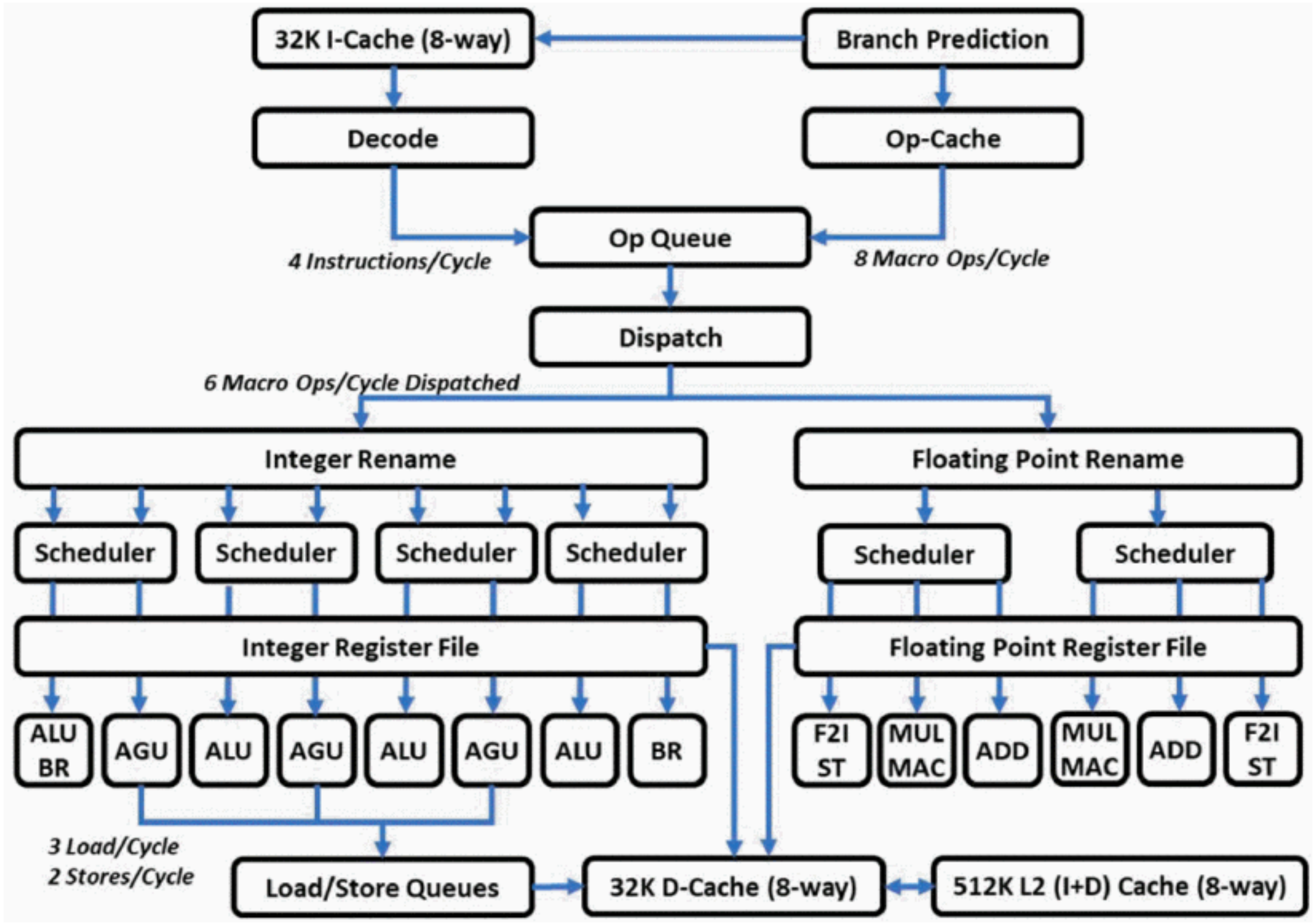
- Multiple issue
  - Replicate pipeline stages => multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, also use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice
  - Superscalar
  - “Out-of-Order” execution
    - Reorder instructions dynamically in hardware to reduce impact of hazards
- More in CAII & EE219 (about AI chips and HW/SW co-design)

Examples



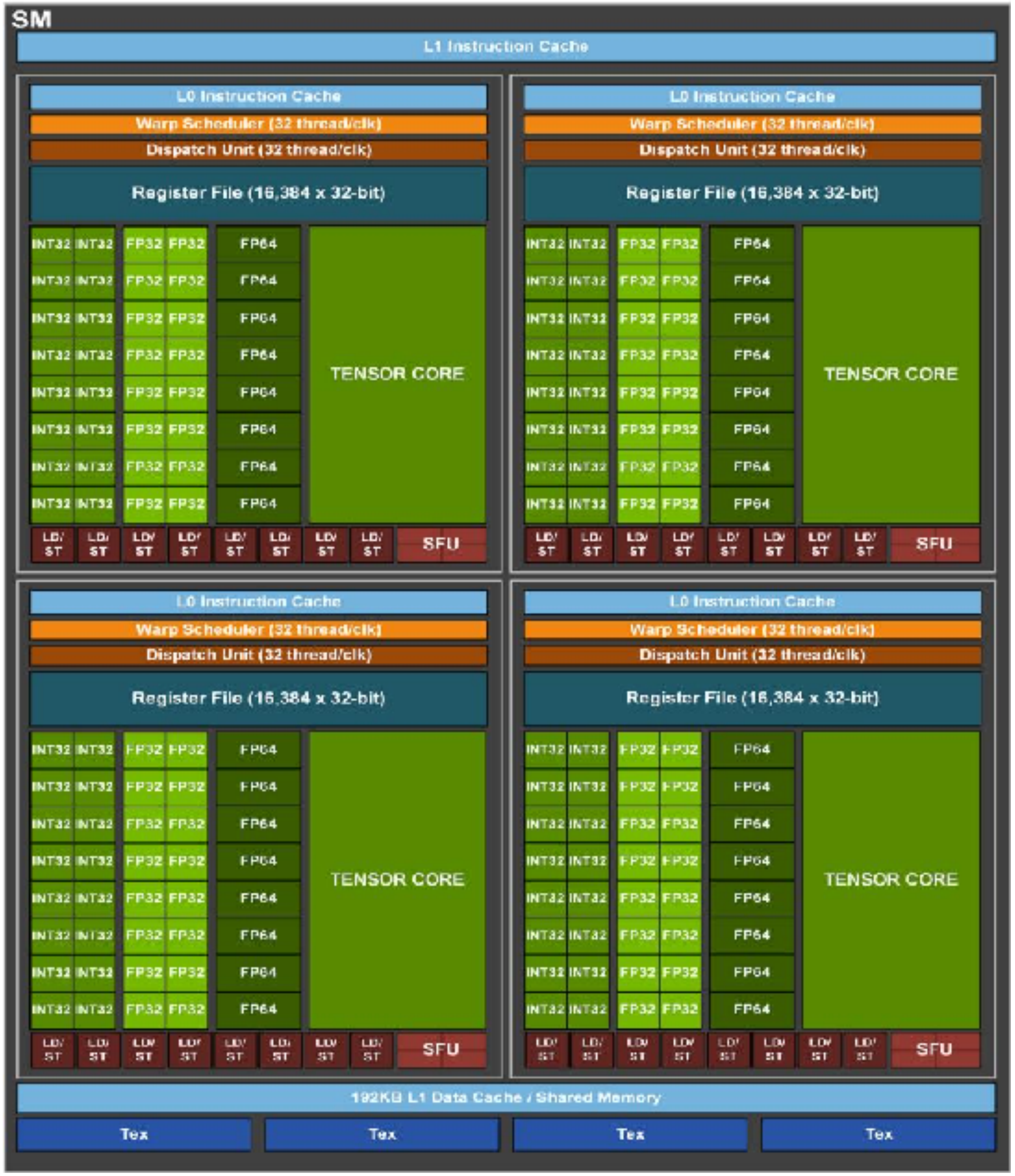
Multi-issue/superscalar is not multicore

Examples: Superscalar (also Multi-issue)



AMD Zen 3, 7 nm process, a single core

Examples: GPU


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

NVIDIA A100

Super"vector"

or

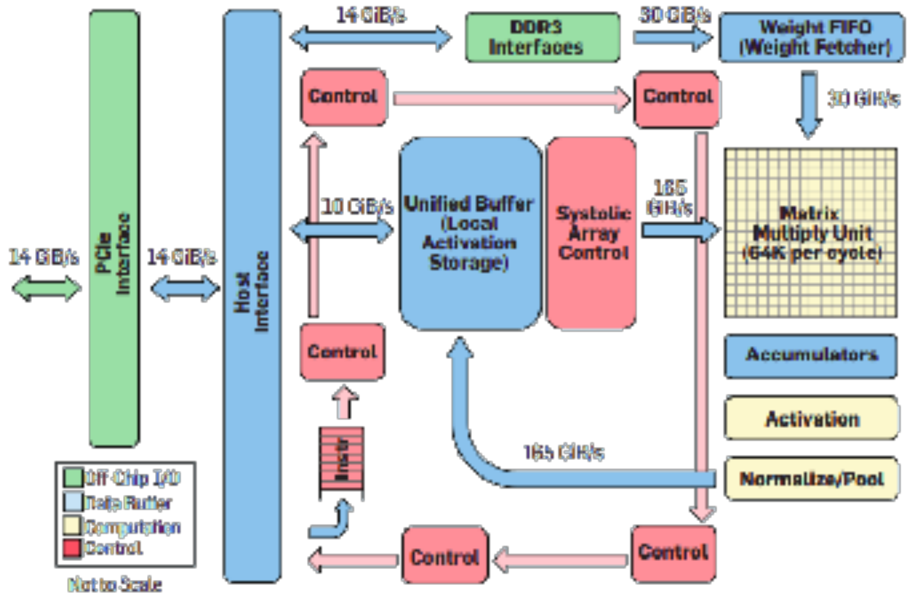
SIMD/MIMD

More on future lectures

(cover by Prof. Wang)

Static Multiple Issue

- aka.: Very Long Instruction Word (VLIW)
- Compiler bundles instructions together
- Compiler takes care of hazards
- Used in Google TPU (an AI chip)



ALU or branch	IF	ID	EX	MEM	WB			
Load or store	IF	ID	EX	MEM	WB			
ALU or branch		IF	ID	EX	MEM	WB		
Load or store		IF	ID	EX	MEM	WB		
ALU or branch			IF	ID	EX	MEM	WB	
Load or store			IF	ID	EX	MEM	WB	
ALU or branch				IF	ID	EX	MEM	WB
Load or store				IF	ID	EX	MEM	WB

CPU Design & Manufacturing

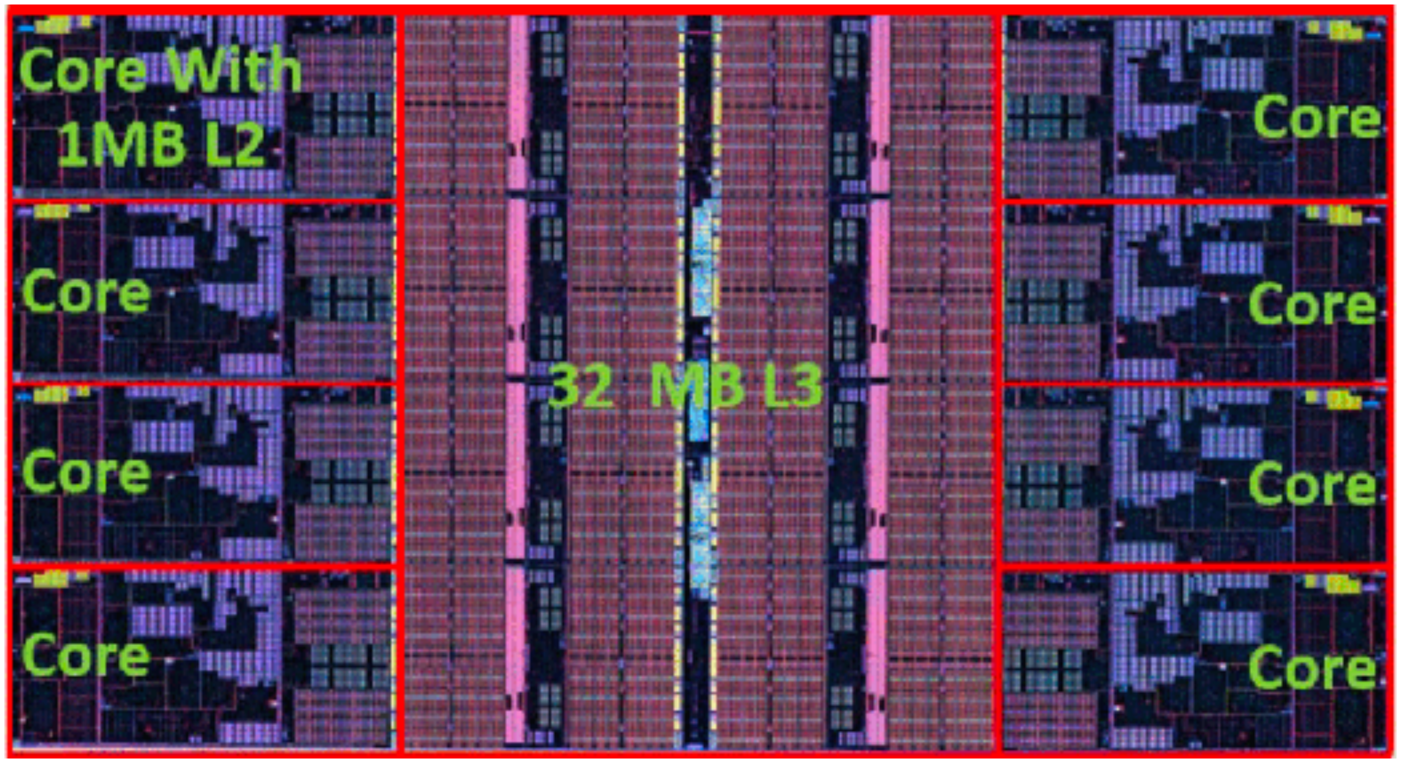
- Specifications defined.
  - Target performance, TPD (power consumption), cost, etc.
- Microarchitecture design
  - Decide ISA, multi-issue/VLIW/superscalar/out-of-order execution, etc.
  - Hardware design using hardware description language (HDL)

```
module alu(opA, opB, aluop, result, zero);
parameter width=32;
input [1:0] aluop; input [width-1:0] opA, opB;
output reg [width-1:0] result; output reg zero;
always @(*) zero = (result == 0);
always @(opA, opB, aluop) begin
case (aluop) 0: result = opA + opB; 1: result = opA - opB;
2: result = opA & opB; 3: result = opA | opB;
default: result = 0;
endcase
end
endmodule
```

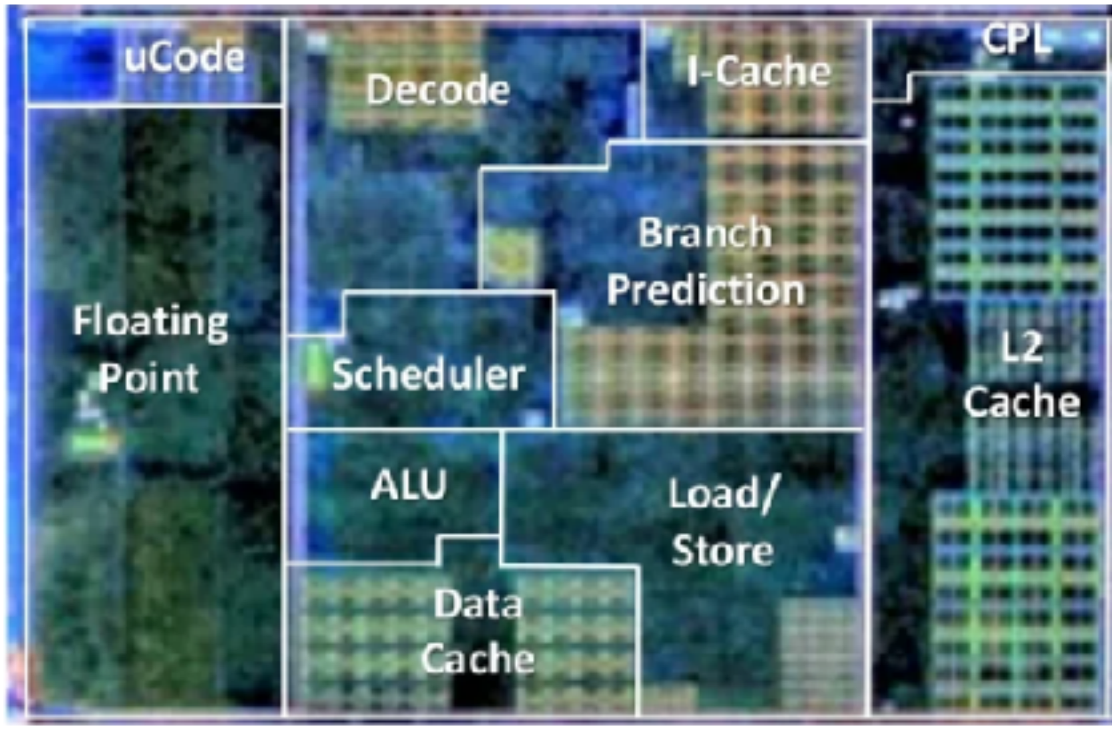
An RTL-level description (verilog) of an ALU.

CPU Design & Manufacturing

- Using EDA tools
  - To generate gate netlist automatically
  - To perform timing analysis
  - To simulate, verify, clock tree generation, etc.
  - To generate layout (GDSII)



AMD Zen 4 https://ieeexplore.ieee.org/document/10067540

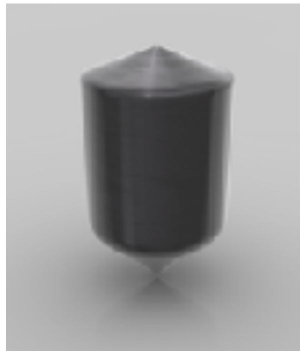


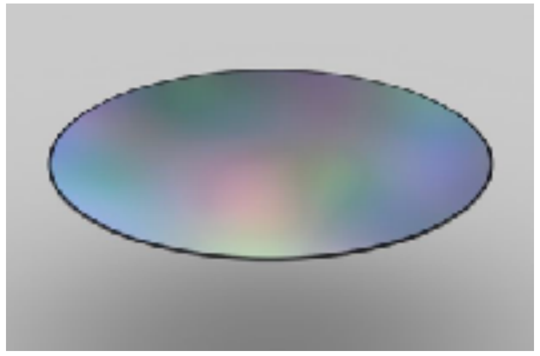
Zen 2: The AMD 7nm Energy-Efficient High-Performance x86-64 Microprocessor Core, 2020 ISSCC.

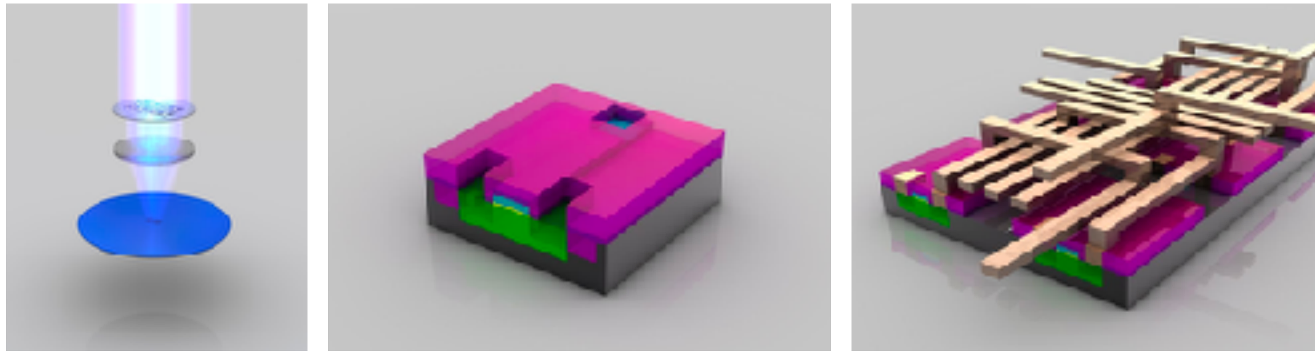
CPU Design & Manufacturing

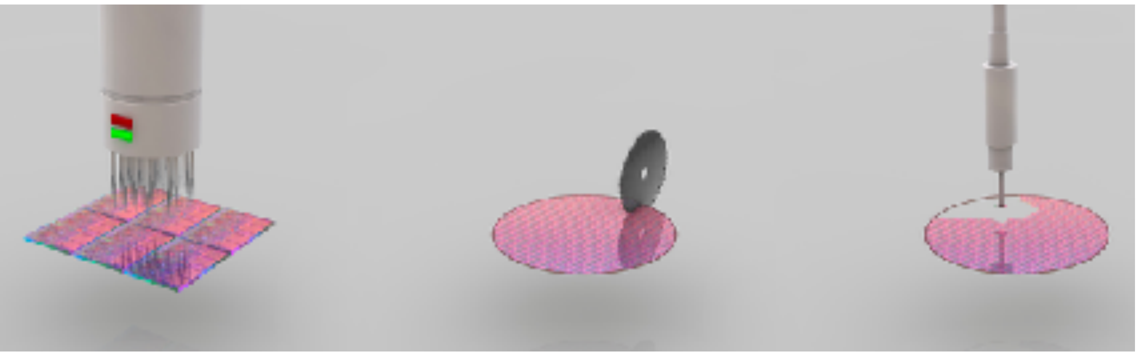
- Manufacturing
  - Different vendors have different strategies (fabless/IDM/foundry)

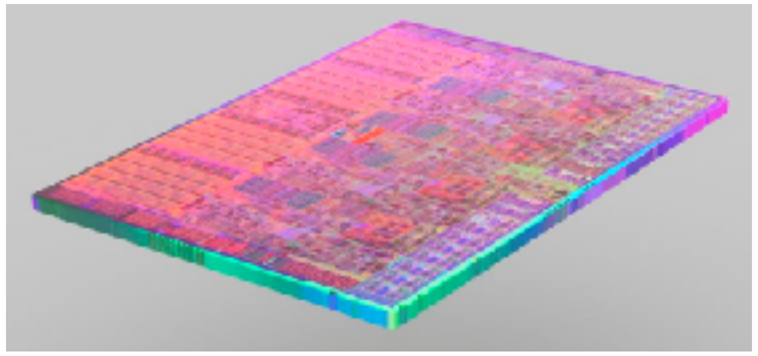
Sand

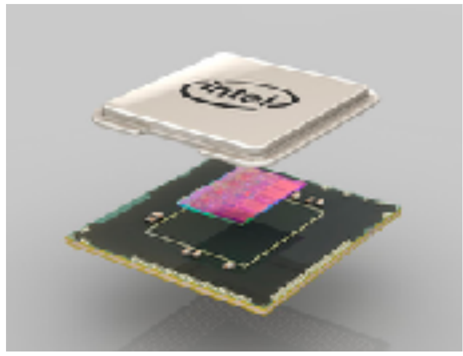

Ingot


Wafer

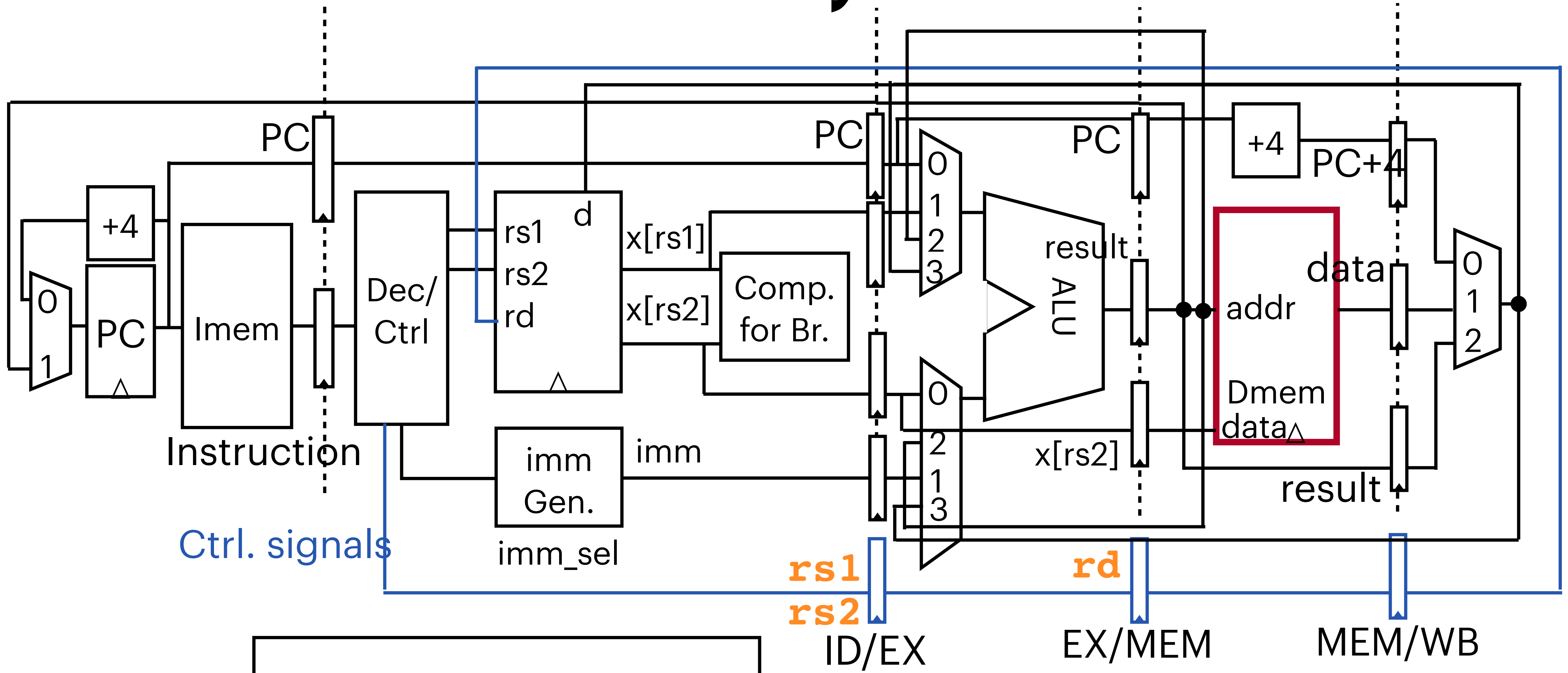

UV light/etching/doping/layering, etc.


Testing/wafer slicing

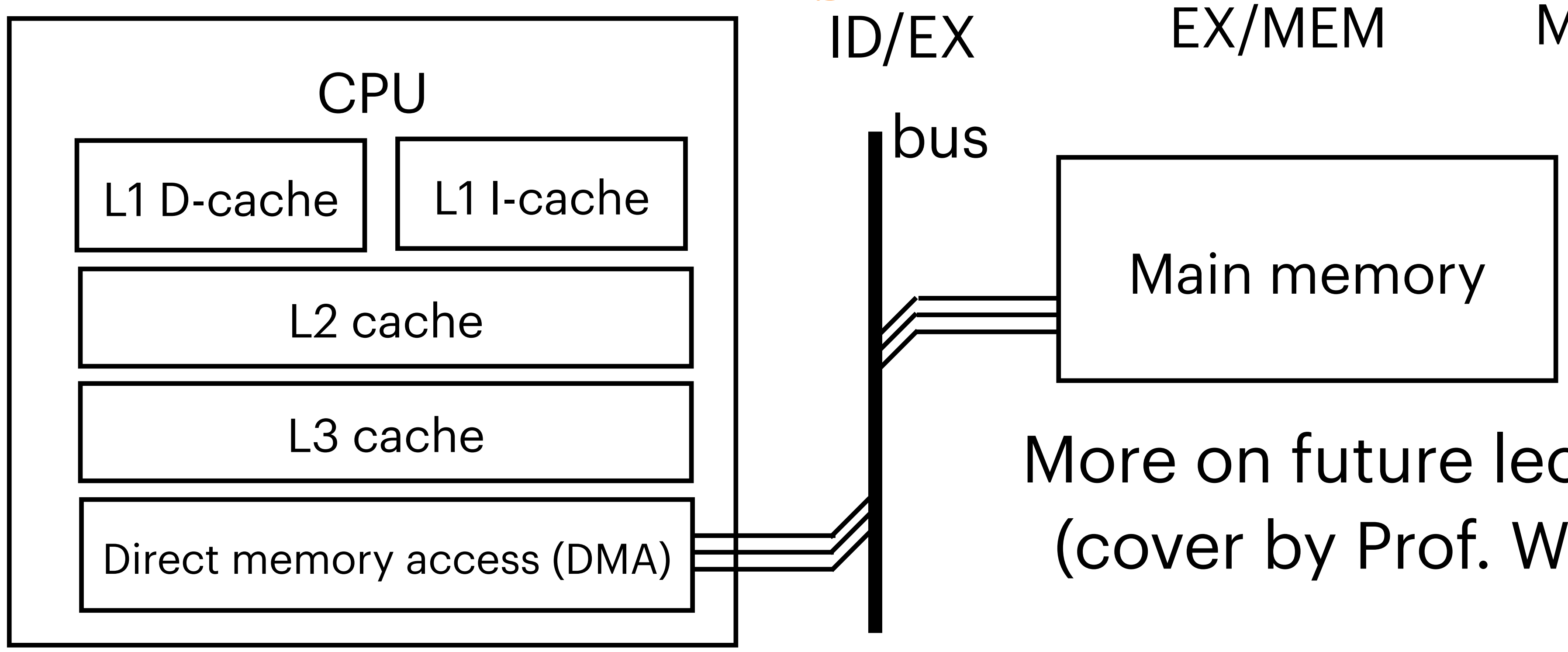

A single die/CPU


Packaging

CPU-Memory Interface



Reality



More on future lectures
(cover by Prof. Wang)

Summer Interns Needed

- Possible topics
  - AI chip design (for convolutional neural network accelerator)
  - RISC-V CPU design (refer to https://ysyx.oscc.cc/)
- You will learn and practice
  - What you have learnt in the first half CAI course and so on
  - Hardware design using hardware description language (HDL) such as verilog HDL or chisel HDL