



信息科学与技术学院

School of Information Science and Technology

CS 110A

Pipeline II

Chundong Wang

The need of pipeline

- CPU pipelining is a technique that **overlaps the execution of multiple instructions**, splitting one instruction's work into several sequential stages (e.g., fetch → decode → execute → memory access → write-back).
 - Improve throughput (instructions per second)
 - Latency (time for one single instruction) does NOT get shorter, but throughput rises
 - Better hardware utilization
 - Higher performance for program execution
 - Cost-effective performance gain by making the most out of existing hardware
 - Hazards

Hazards ahead!!!

- Structural hazards
- Data hazards
- Control hazards

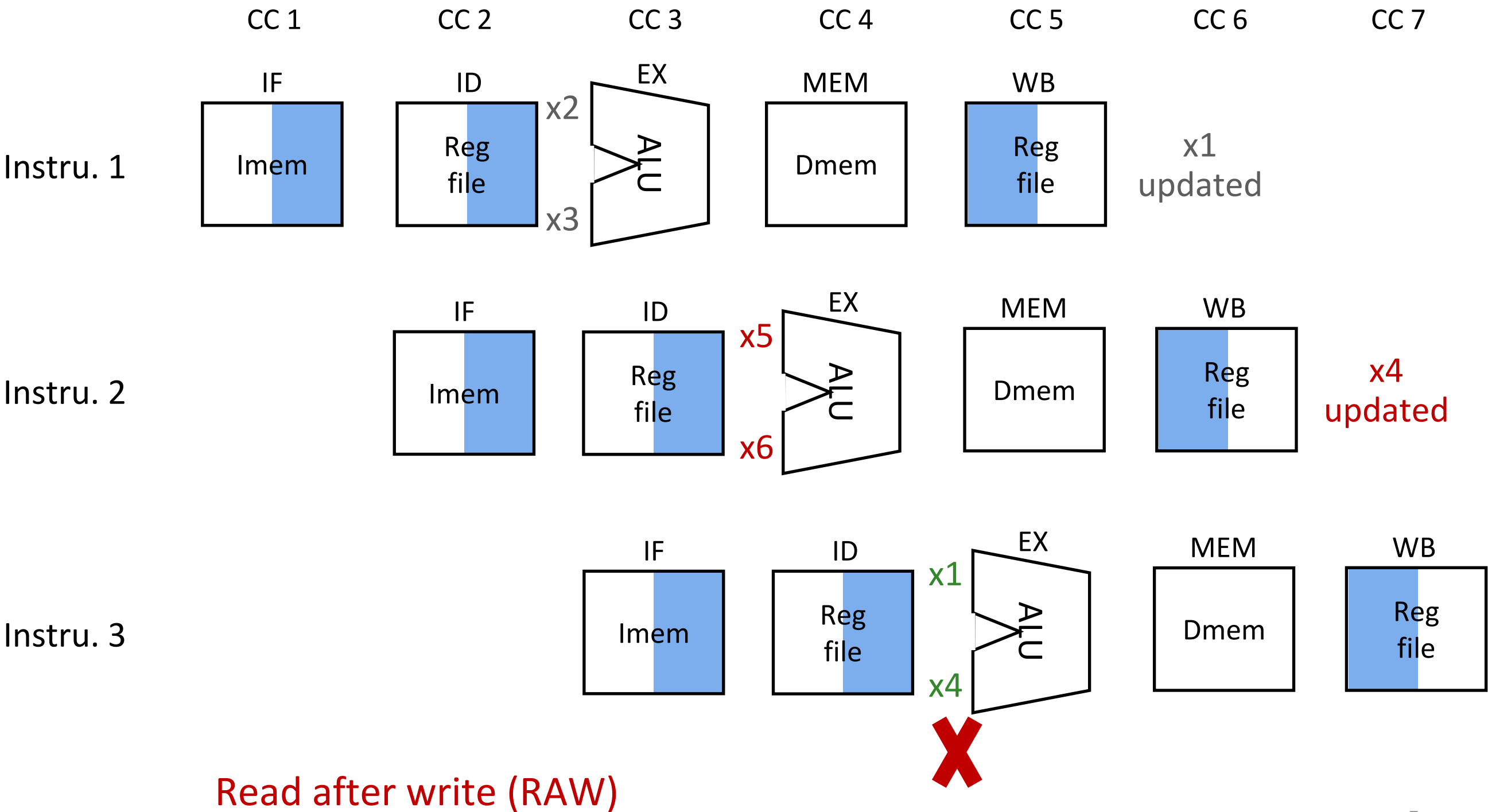
Structural hazards

- Caused by hardware limitations. Two or more instructions in the pipeline compete for a single physical resource
- Can be solved by
 - Separate instruction and data memory (real CPU uses instruction cache and data cache)
 - or using dual-port memory (input multiple addresses, output multiple data) ([general ways to solve structural hazards, add more hardware](#))
 - Assume register file write at rising edge (in the textbook “the first half clock cycle”), read arbitrarily (in the textbook “the second half clock cycle”), and design the hardware with this feature
 - Instructions take turns to use the physical resource (wait/stall)

Data hazards

add x1, x2, x3
add x4, x5, x6
add x7, x1, x4

R-type



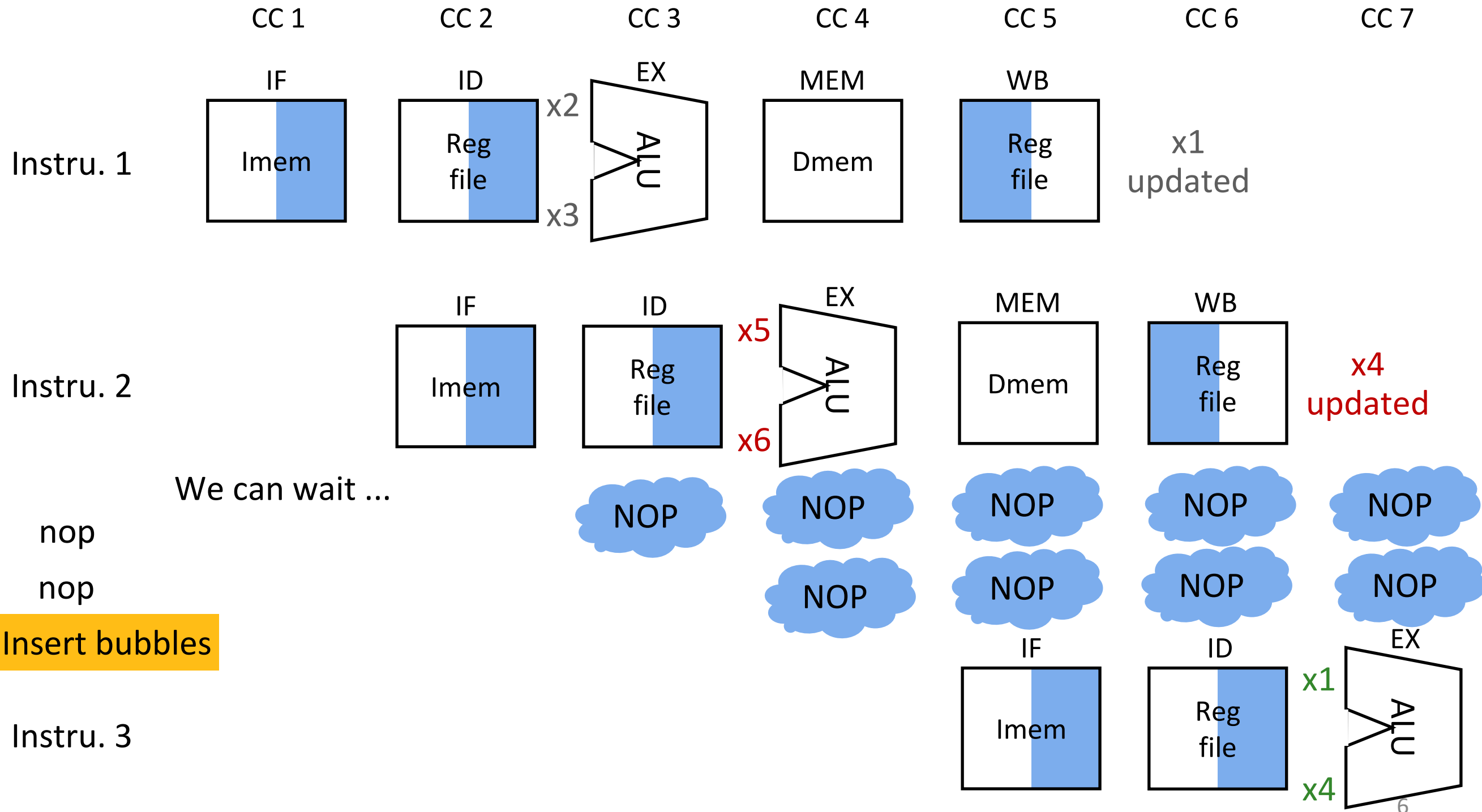
Data hazards -- solution 1

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

R-type



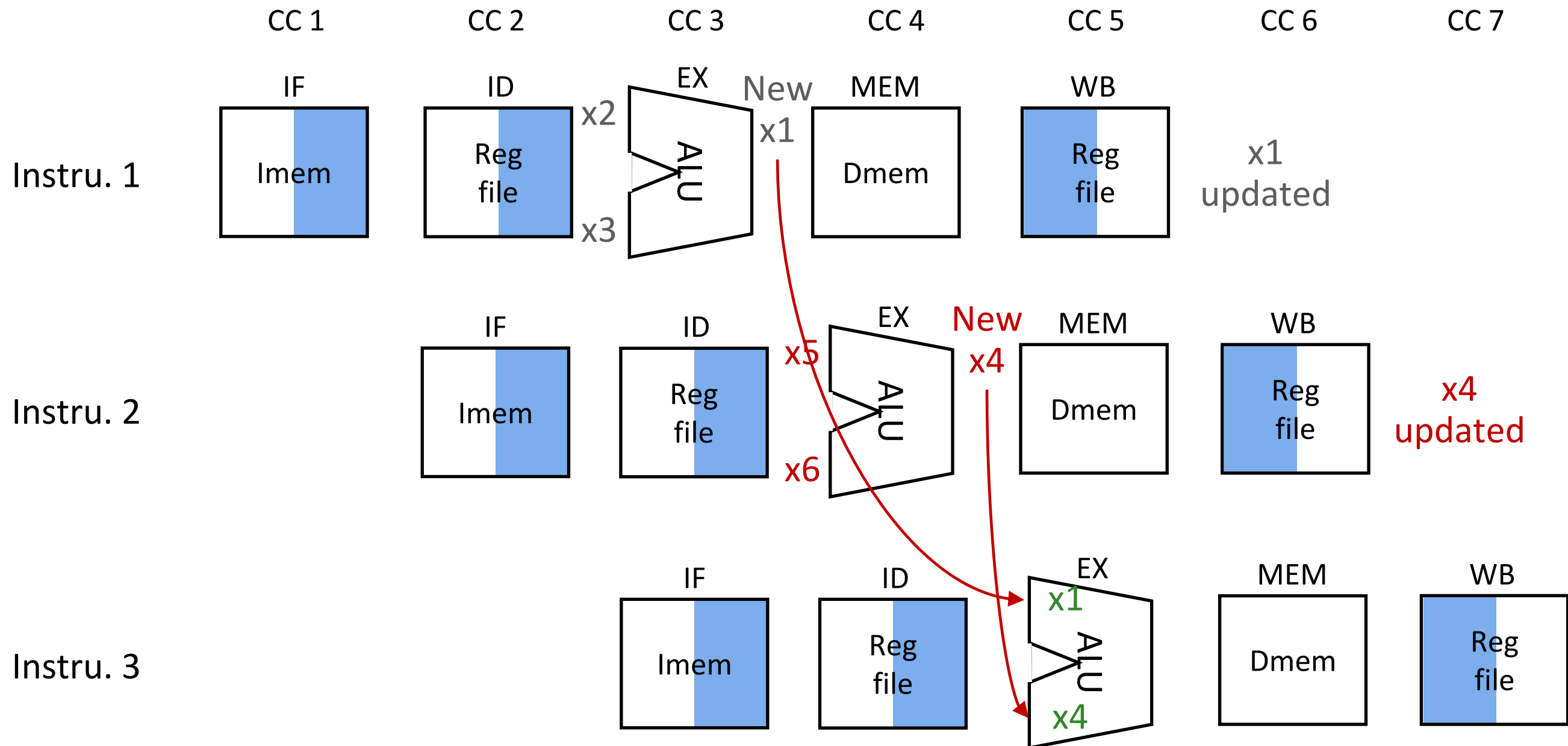
Data hazards -- solution 2

add x1, x2, x3

add x4, x5, x6

add x7, x1, x4

R-type

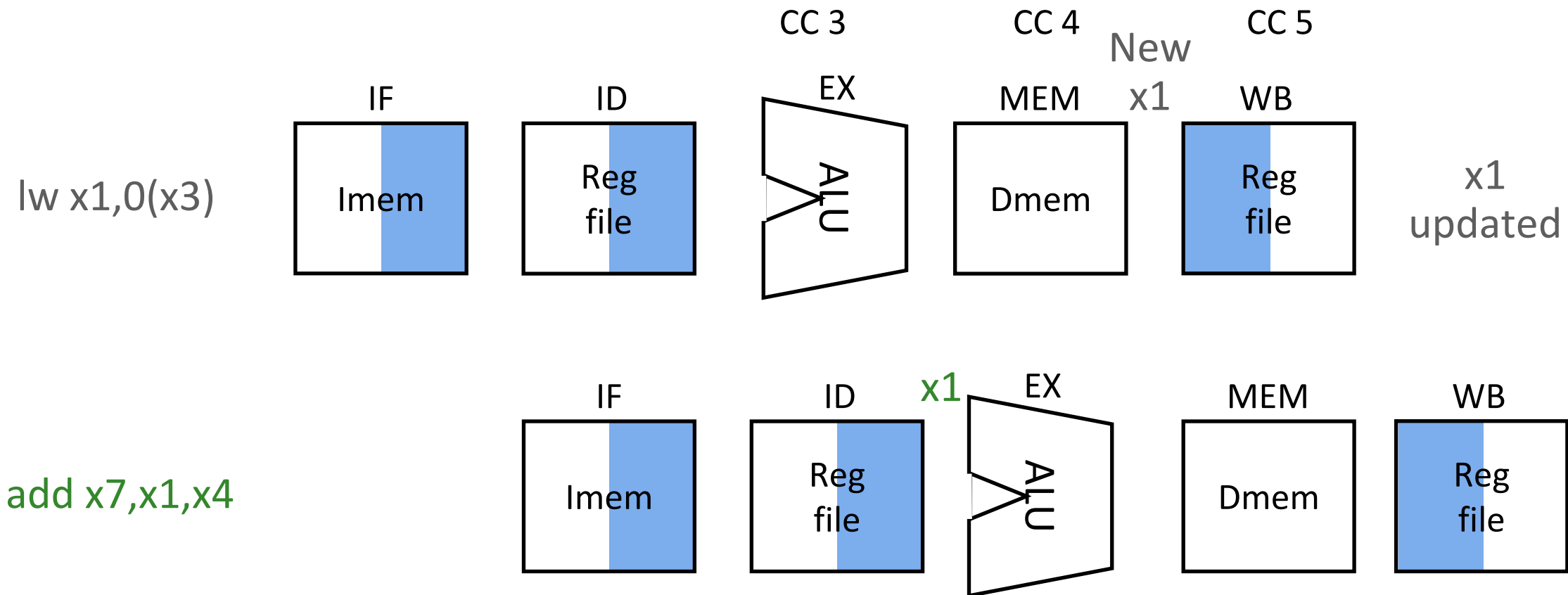


Forwarding or bypass

Data hazards

lw x1,0(x3)
add x7,x1,x4
... ..

Load type RAW

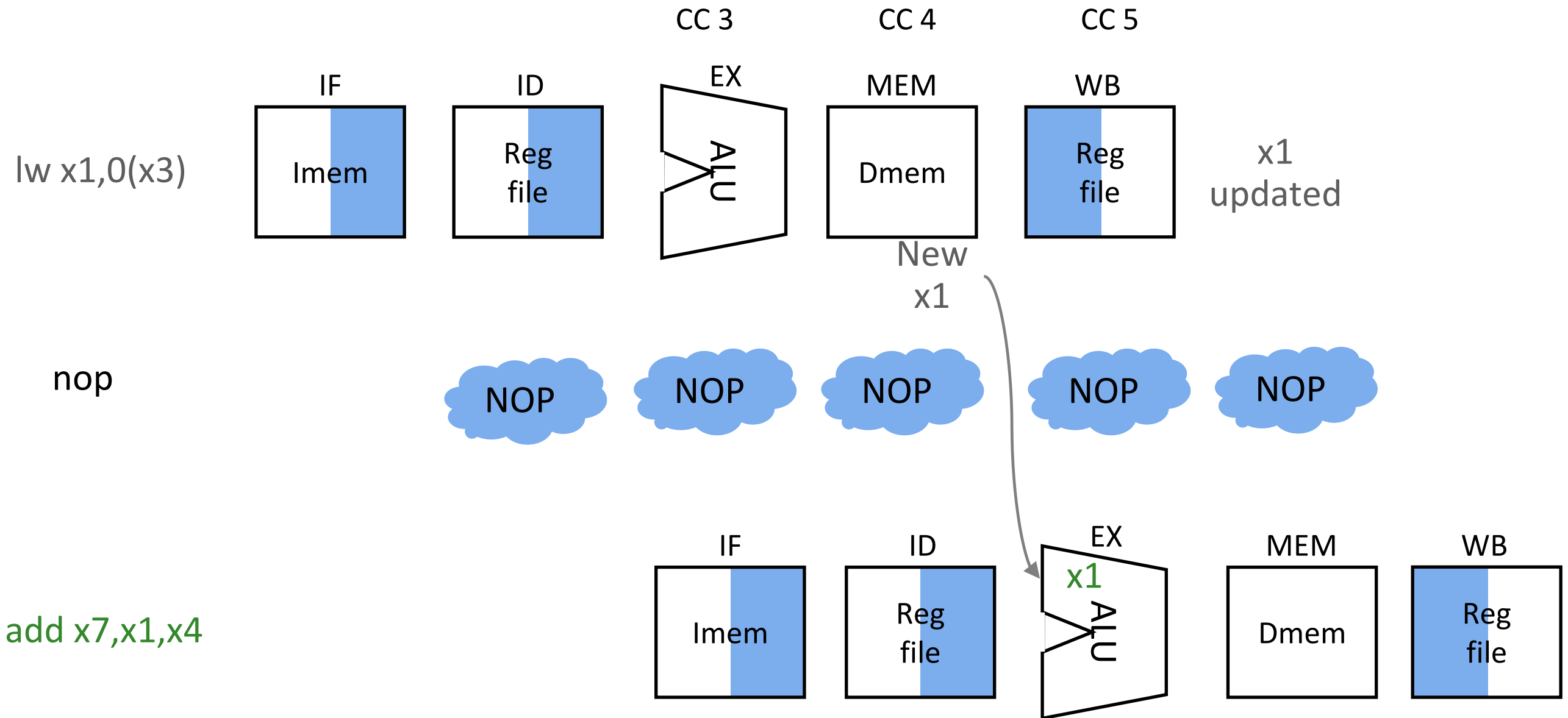


Forwarding cannot solve this, leading to a “load delay slot”

Data hazards -- solution 1

```
lw x1,0(x3)
add x7,x1,x4
... ..
```

Load type RAW



Insert nop and forward x1

Data hazards -- solution 2 warm up

- Identify all the data hazards in the following code
- Which of the hazards cannot be completely solved by forwarding?

Original Order:

```
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)
```

Data hazards -- solution 2

- Code scheduling: Put unrelated instruction into load delay slot
 - No performance loss!

Original Order:

```
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)
```

Code scheduling:

```
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)
```

- Code scheduling can be accomplished by the compiler

Data hazards

- How many clock cycles to complete the code before and after code scheduling? Assume no instruction in the pipeline initially.

Original Order:

```
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)
```

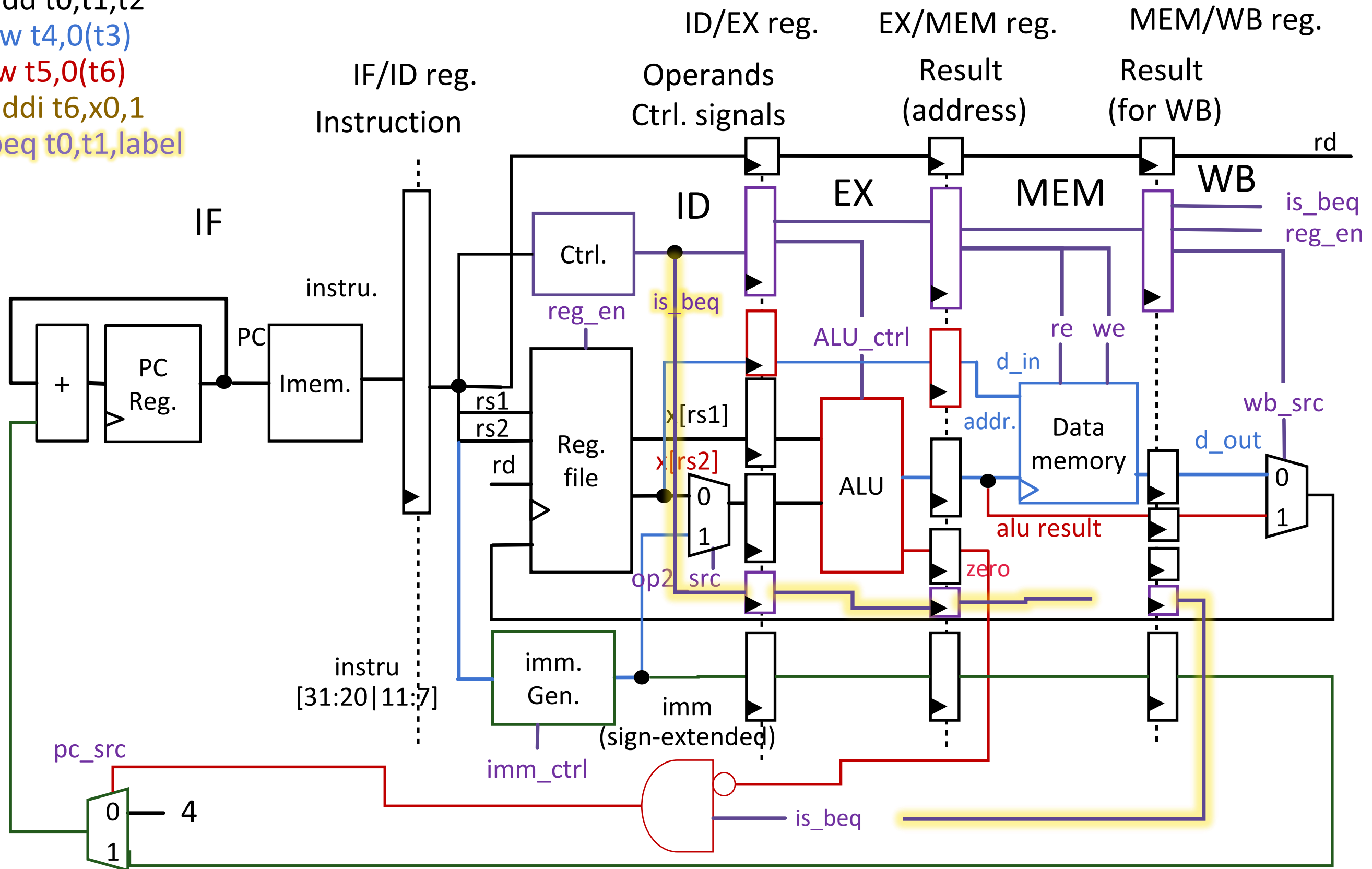
Code scheduling:

```
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)
```

Hint: Do not miss structural hazard(s)

Detailed considerations

add t0,t1,t2
 sw t4,0(t3)
 lw t5,0(t6)
 addi t6,x0,1
 beq t0,t1,label



Detailed considerations

0x0:add t0,t1,t2

0x4:sw t4,0(t3)

0x8:lw t5,0(t6)

0xc:addi t6,x0,1

0x10:beq t0,t1,label

0x14:beq_next1

0x18:beq_next2

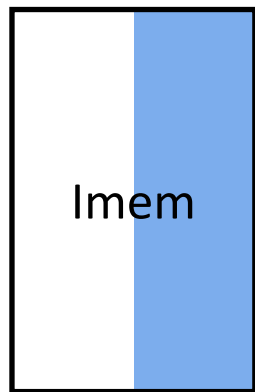
0x1c:beq_next3

0x20:beq_next4

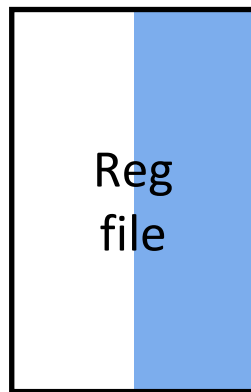
0x24:beq_next5

PC= 0x??

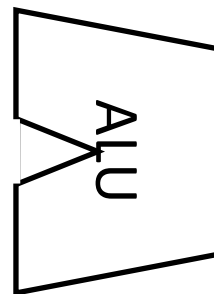
IF



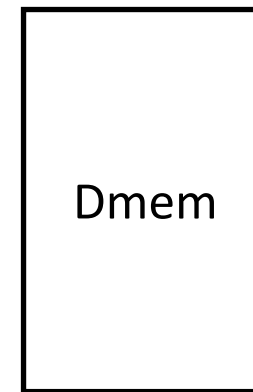
ID/DEC



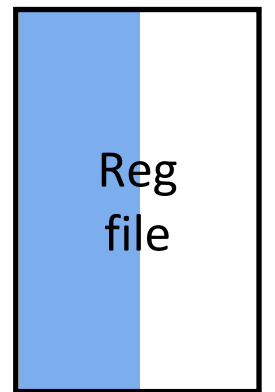
EX



MEM



WB



add

sw

lw

addi

beq

beq_next1

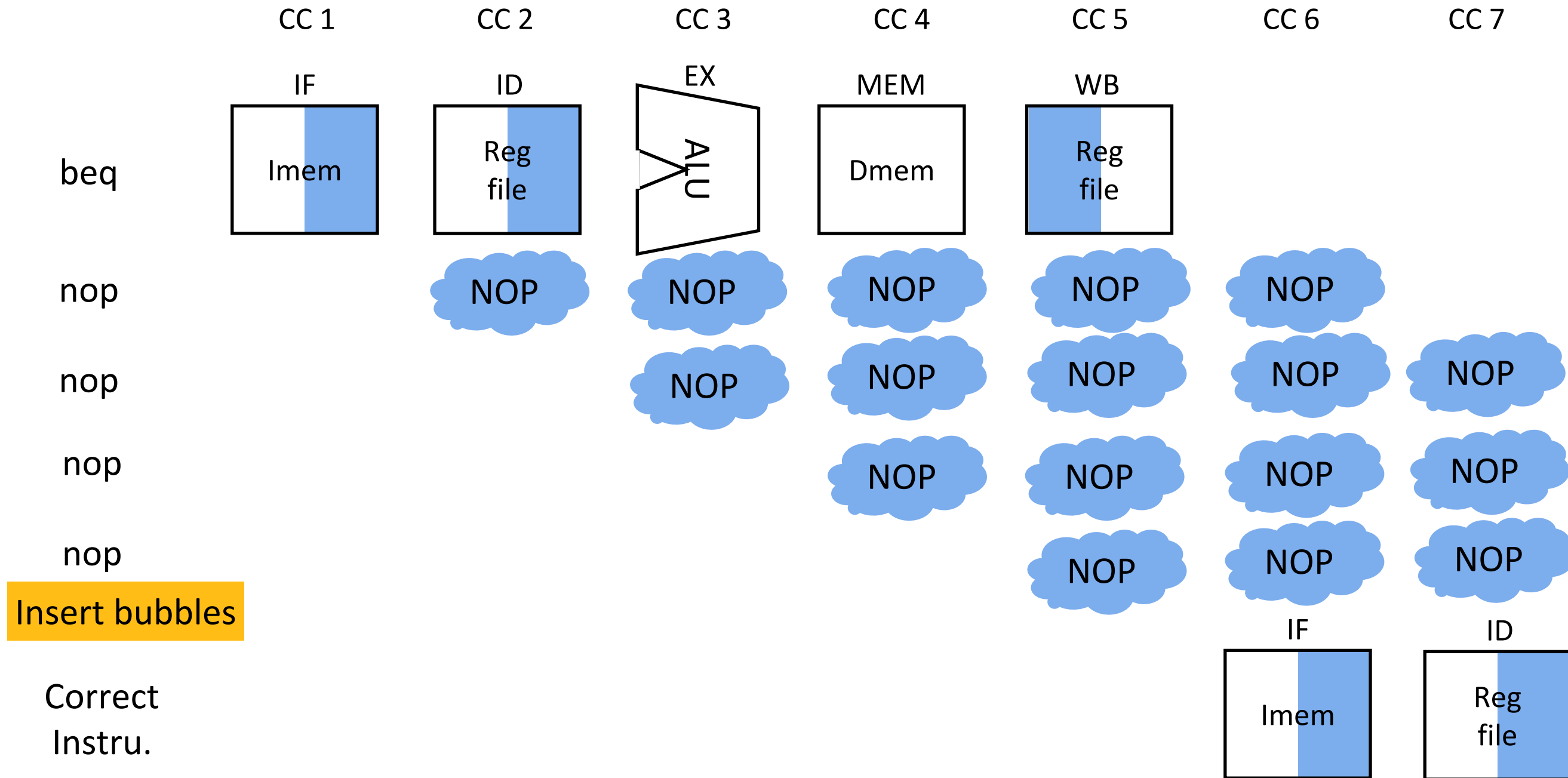
beq_next2

beq_next3

beq_next4

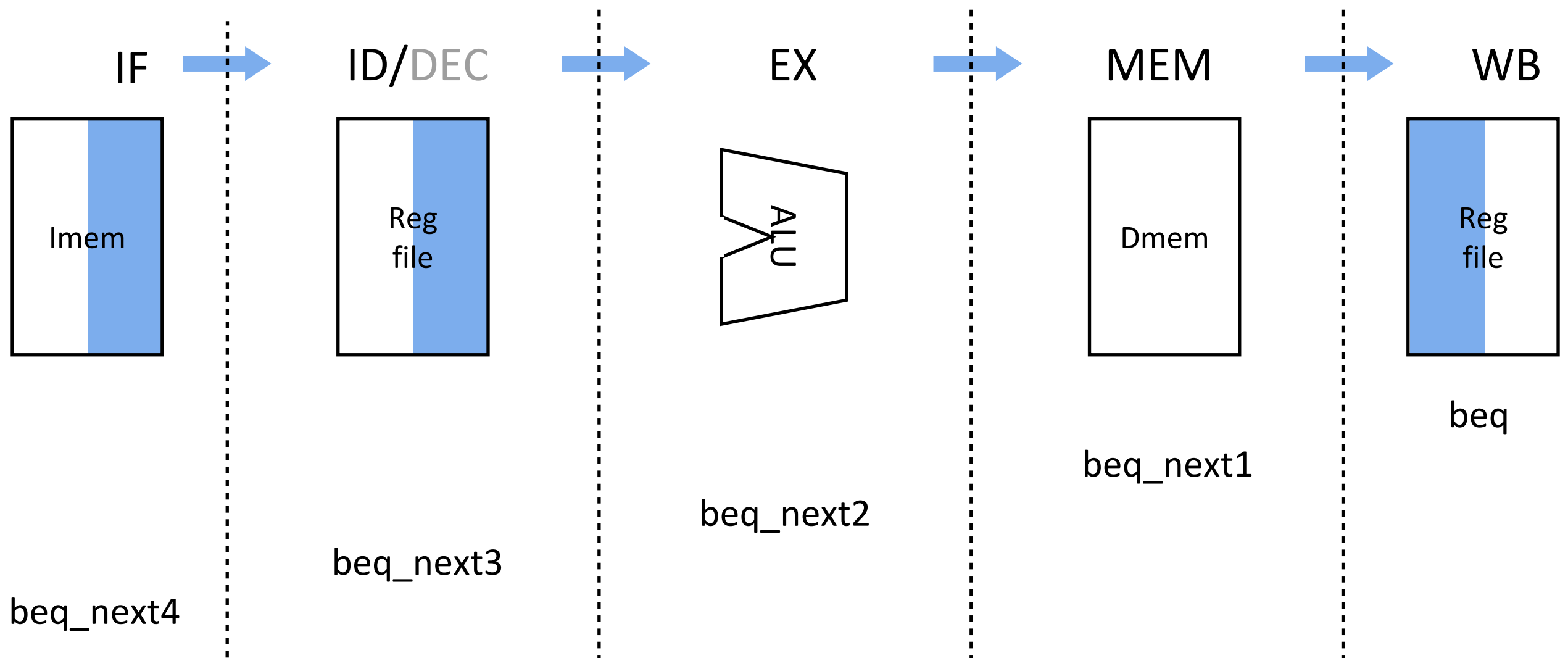
Control hazards--solution 1

We can wait ...



Control hazards -- solution 2

- Assume branch not taken (static)
- Extra control logics to deal with the cases that the branches are taken
 - Flush the pipeline and restore the states



Speculation

Control hazards -- solution 2

- Assume branch not taken (static)
- Not optimal in some cases

```
int A[20];  
int sum = 0;  
for (int i=0; i < 20; i++)  
    sum += A[i];
```

Assume x8 holds pointer to A

Assign x10=sum

add x10, x0, x0 # sum=0

add x11, x8, x0 # ptr = A

addi x12, x11, 80 # end = A + 80

Loop:

lw x13, 0(x11) # x13 = *ptr

add x10, x10, x13 # sum += x13

addi x11, x11, 4 # ptr++

blt x11, x12, Loop # ptr < end

Wrong speculations except the last branch

Control hazards -- solution 2

- Alternatively, **dynamic** branch prediction (when the program is running)

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

Assume x8 holds pointer to A

Assign x10=sum

add x10, x0, x0 # sum=0

add x11, x8, x0 # ptr = A

addi x12, x11, 80 # end = A + 80

Loop:

lw x13, 0(x11) # x13 = *ptr

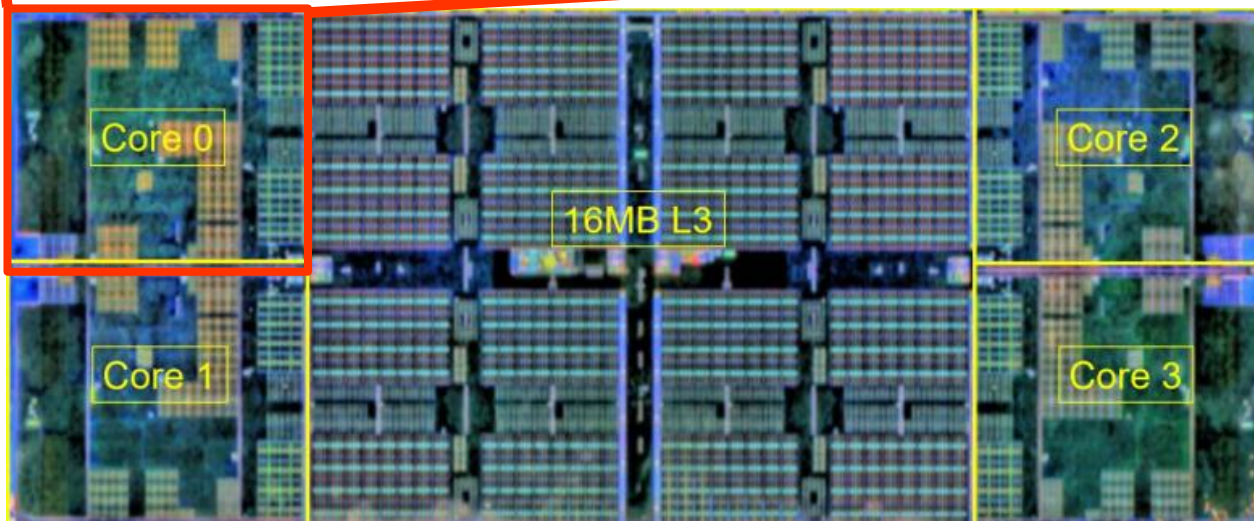
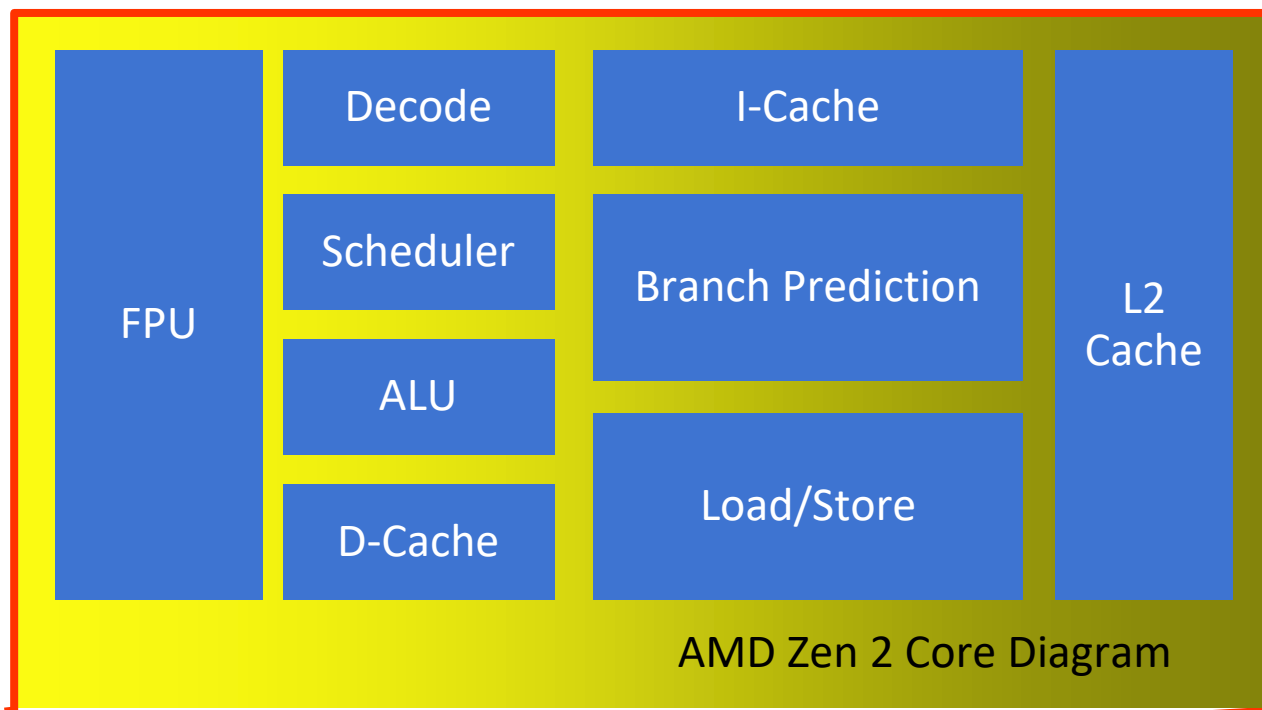
add x10, x10, x13 # sum += x13

addi x11, x11, 4 # ptr++

blt x11, x12, Loop # ptr < end

- Record the position of branch
- Record if the branch is taken for this branch
- Predict if the branch will be taken based on the current record
- Can be modeled as an FSM
- Use one or more bits to represent “(strong) taken” or “(strong not taken)”

Real stuff



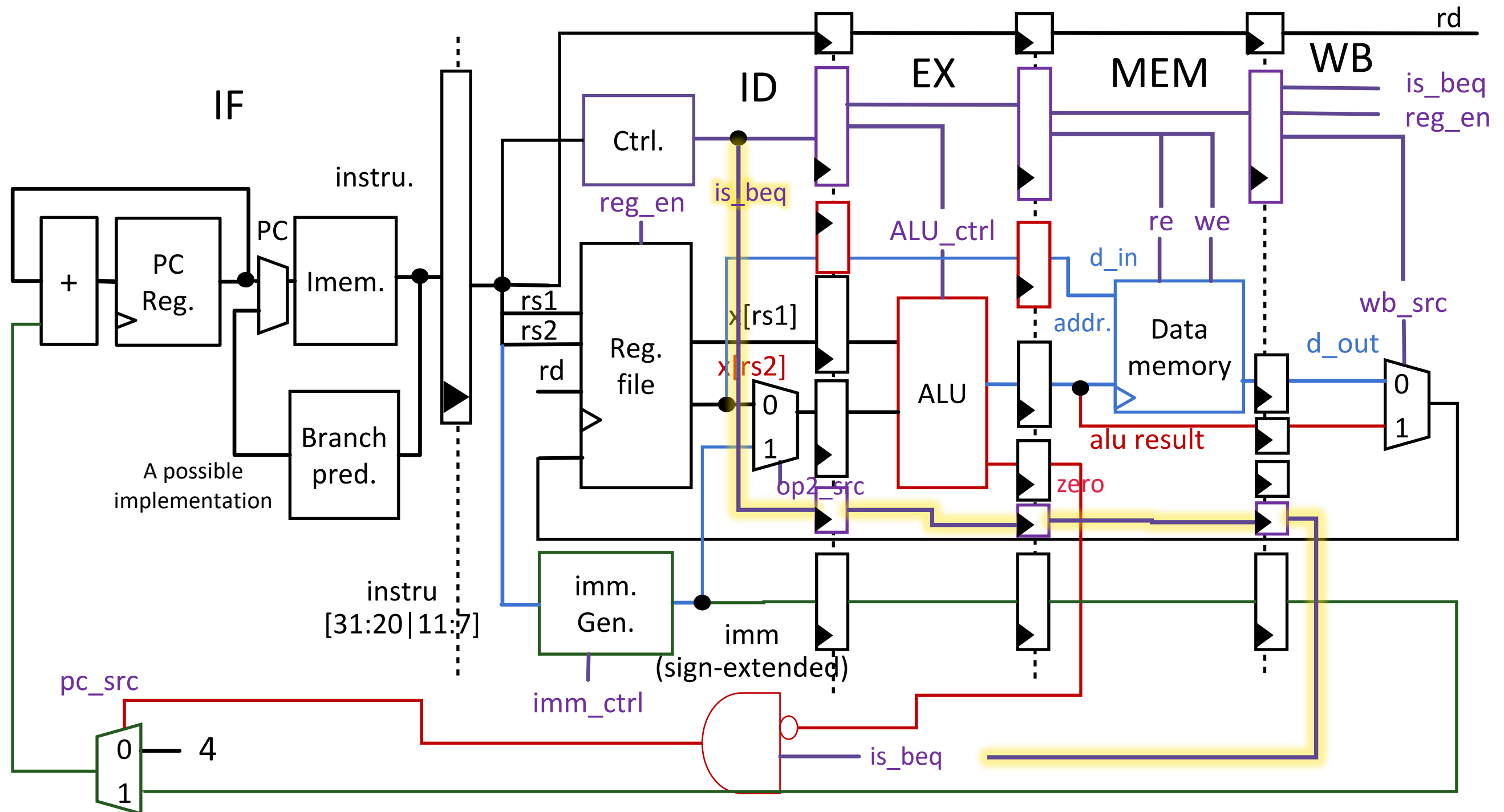
Unit	Zen	Zen 2
Floating Point	128b	256b
L0 Branch Target Buffer	8 entries	16 entries
L1 Branch Target Buffer	256 entries	512 entries
L2 Branch Target Buffer	4K entries	7K entries
Op Cache	2K ops	4K ops
Integer Physical Register File	168 entries	180 entries
Integer Scheduler	84 entries	92 entries
AGEN	2	3
ROB	192 entries	224 entries
L2DTLB	1.5K	2K
L3 Cache Size	8MB	16MB

7.83 mm² per core

[1] T. Singh et al., "2.1 Zen 2: The AMD 7nm Energy-Efficient High-Performance x86-64 Microprocessor Core," IEEE International Solid- State Circuits Conference - (ISSCC), 2020, pp. 42-44.

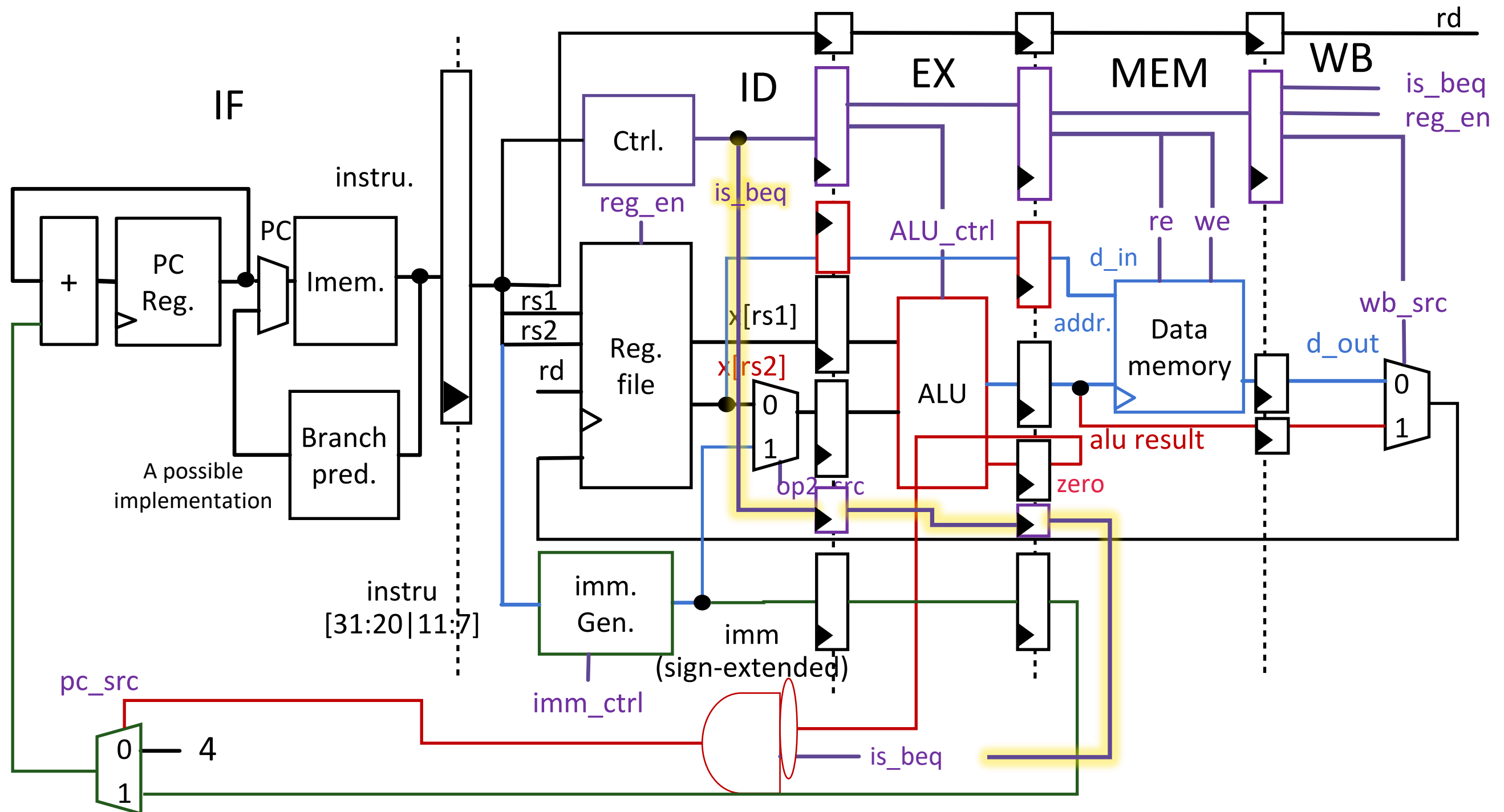
Control hazards -- solution 3

- Use idea similar to forwarding to reduce the delay of branches



Control hazards -- solution 3

- Use idea similar to forwarding to reduce the delay of branches



Summary on control hazards

- The delay in determining the proper instruction to fetch is called a control hazard or branch hazard;
- Methods to deal with control hazards

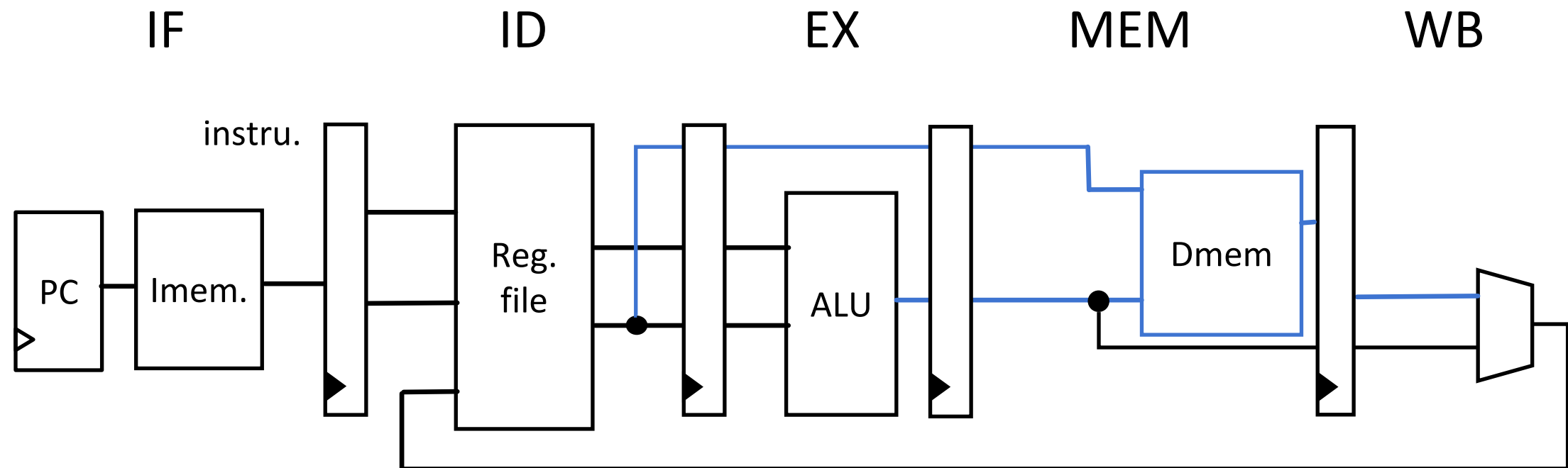
Review

- Overview on parallelism
 - Instruction-level parallelism (ILP)
 - Pipeline, deeper for faster clock, but potentially more hazards
 - Today's lecture (Multi-issue)
 - Data-level parallelism (DLP)
 - SIMD
 - Thread-level parallelism (TLP)
 - Multi-threading/Hardware hyper-threading

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

Single-issue

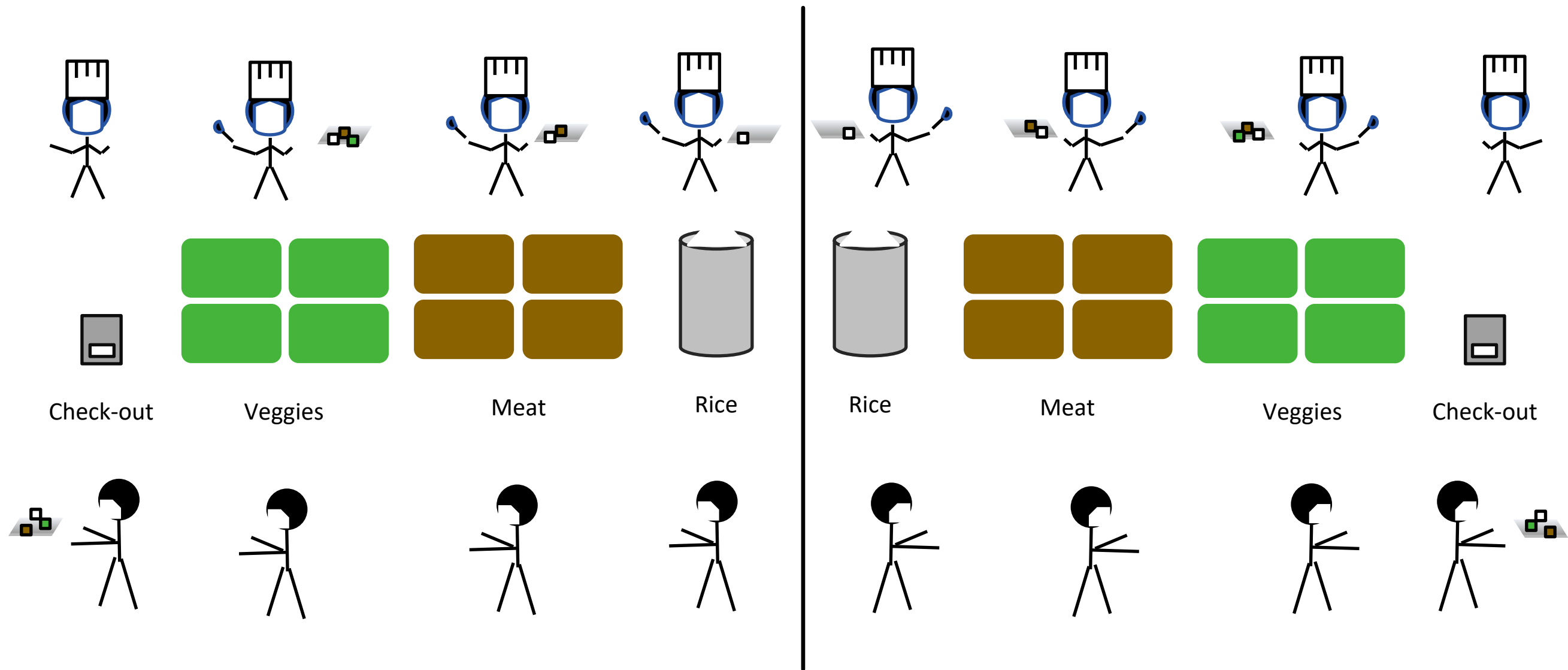
- Simplified 5-stage pipelined single-issue CPU datapath



- At most 1 instruction is “issued” to the datapath at 1 clock cycle

average CPI ≥ 1

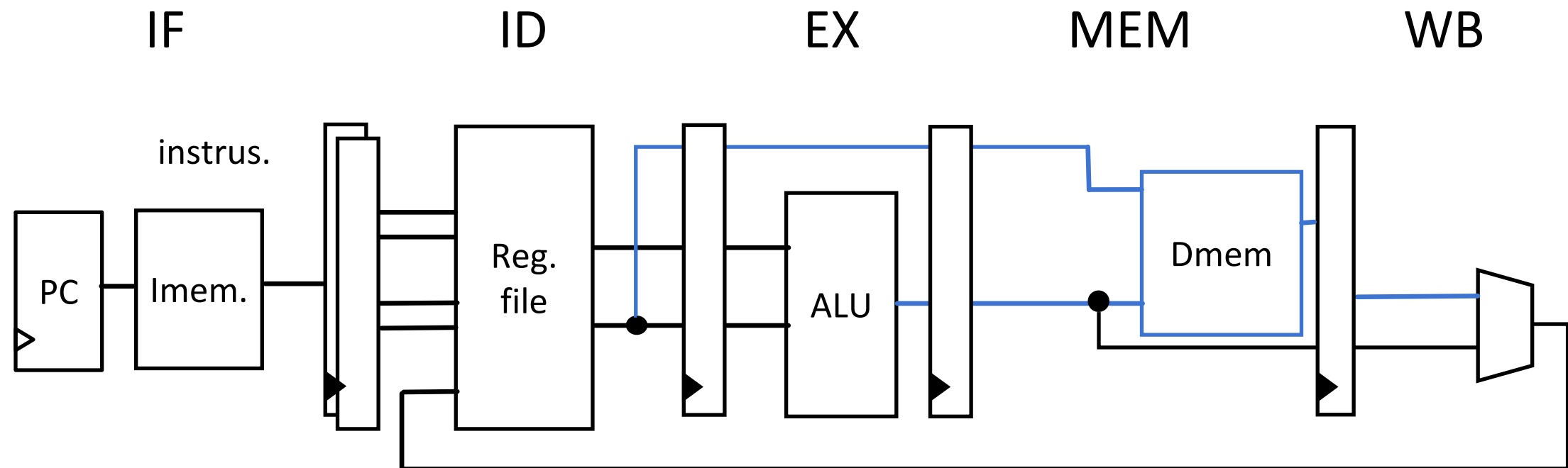
Multi-issue



Canteen analogy
(combined with pipelining)

Multi-issue (Hardware)

- Multi-issue CPU datapath

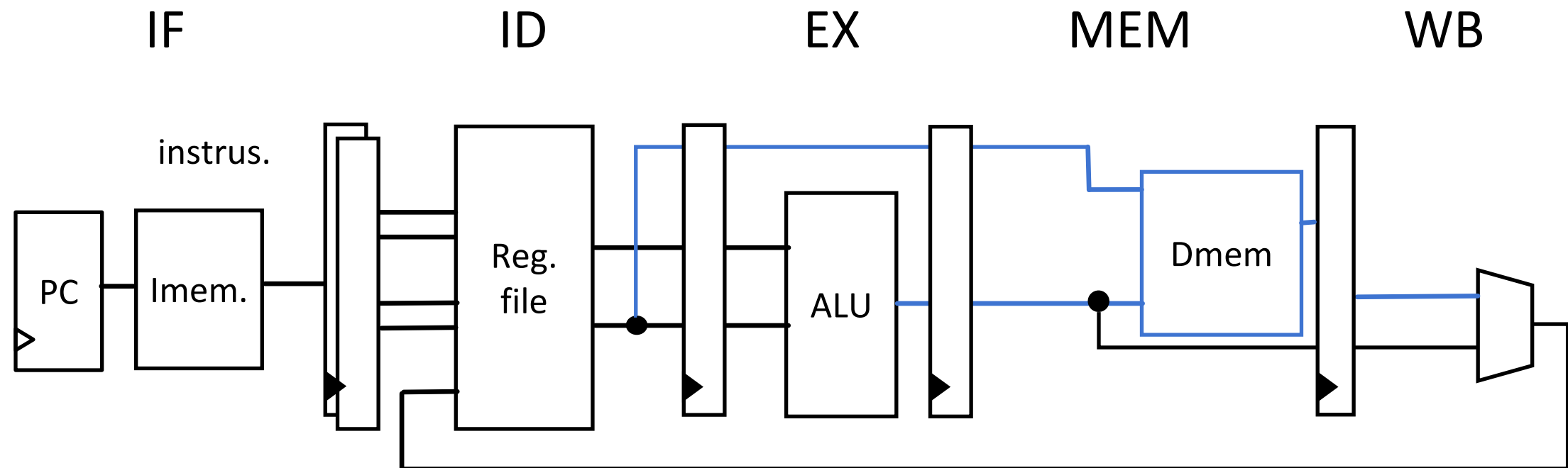


Hardware has to be adapted to avoid structural hazards

- Issue multiple instructions to the datapath in 1 clock cycle, average CPI can be smaller than 1.

Multi-issue (Hardware)

- Multi-issue CPU datapath

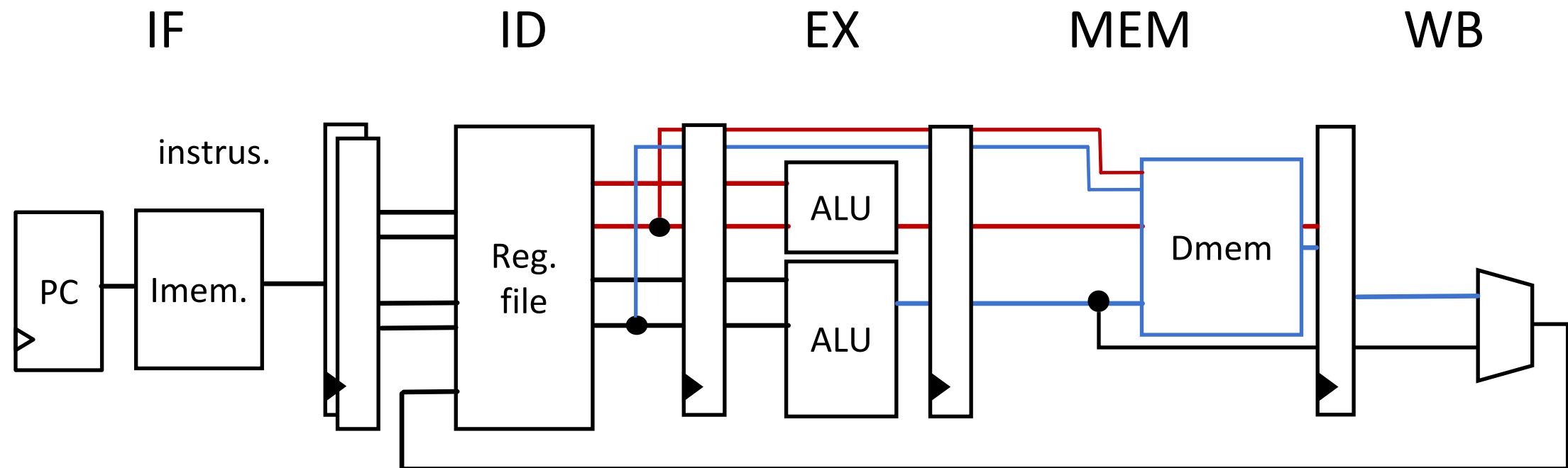


Hardware has to be adapted to avoid structural hazards

- Issue multiple instructions to the datapath in 1 clock cycle, average CPI may be smaller than 1.

Multi-issue (Hardware)

- Multi-issue CPU datapath

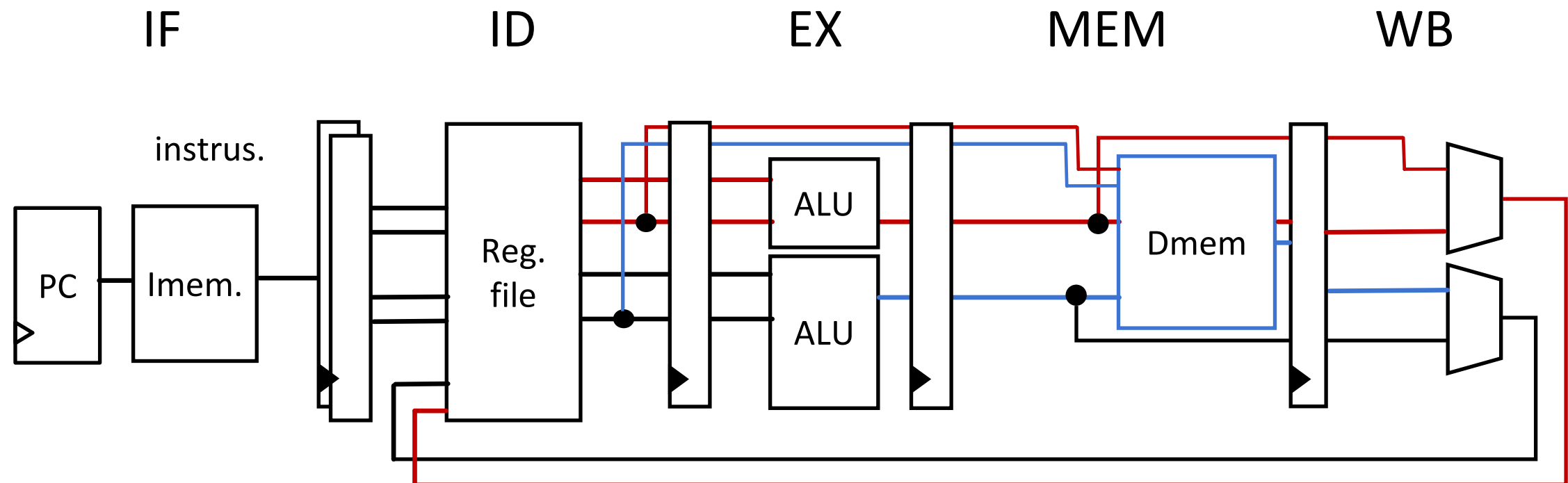


Hardware has to be adapted to avoid structural hazards

- Issue multiple instructions to the datapath in 1 clock cycle, average CPI may be smaller than 1.

Multi-issue (Hardware)

- Multi-issue CPU datapath

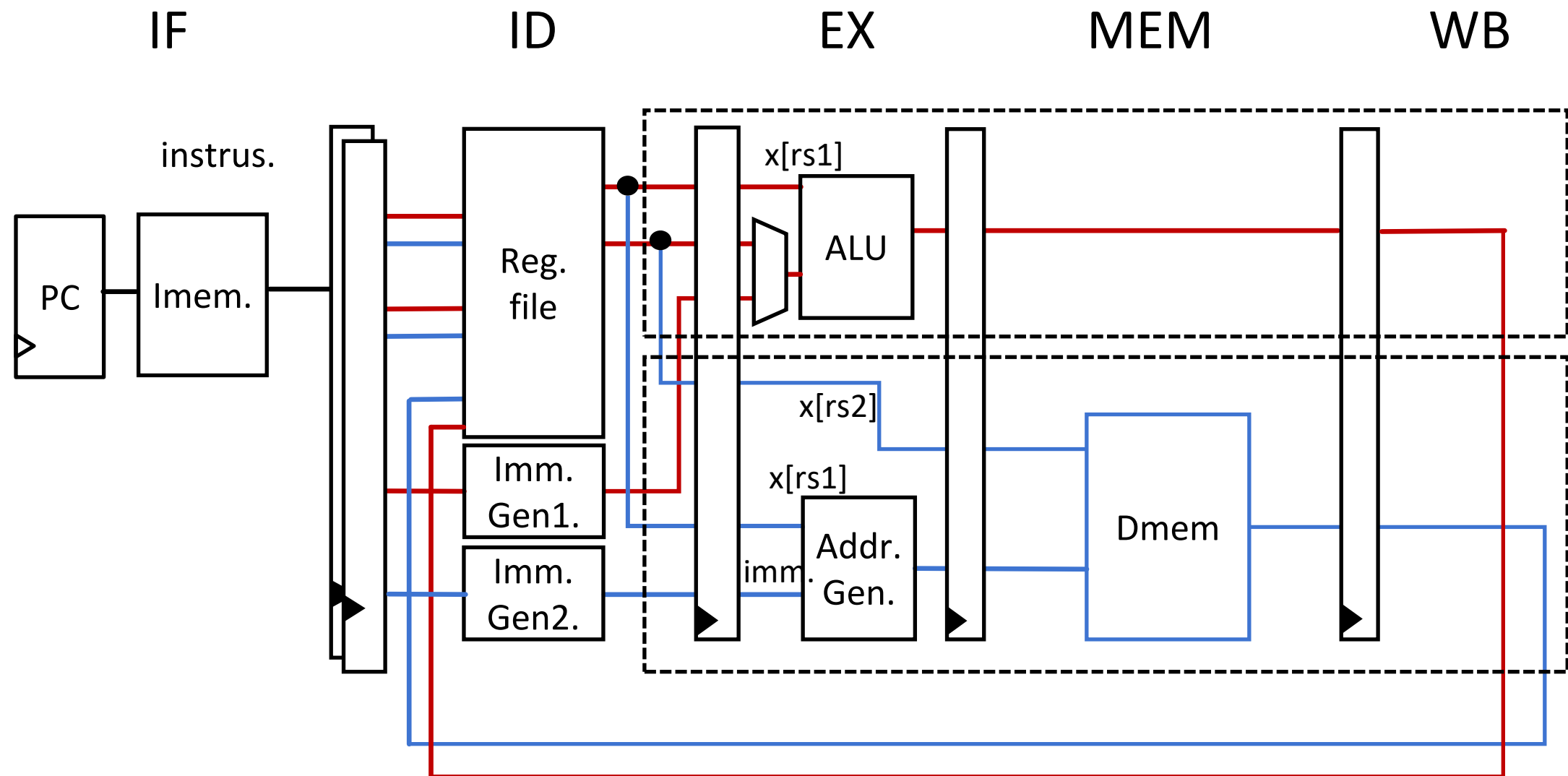


Hardware has to be adapted to avoid structural hazards

- Issue multiple instructions to the datapath in 1 clock cycle, average CPI may be smaller than 1.

Multi-issue (Hardware)

- In practice, we build different datapaths for different types of instructions
- E.g.

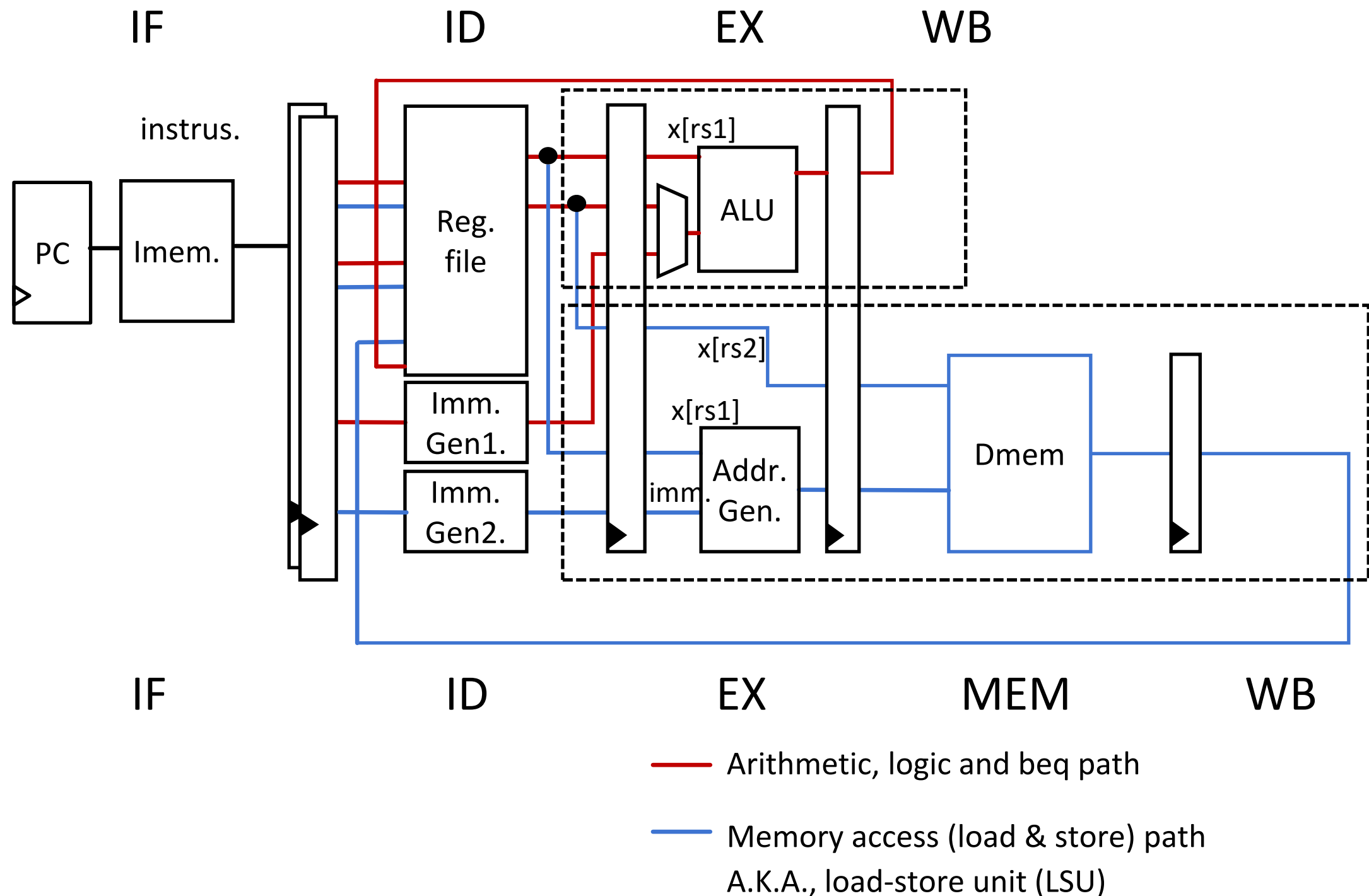


— Arithmetic, logic and beq path

— Memory access (load & store) path
A.K.A., load-store unit (LSU)

Multi-issue (Hardware)

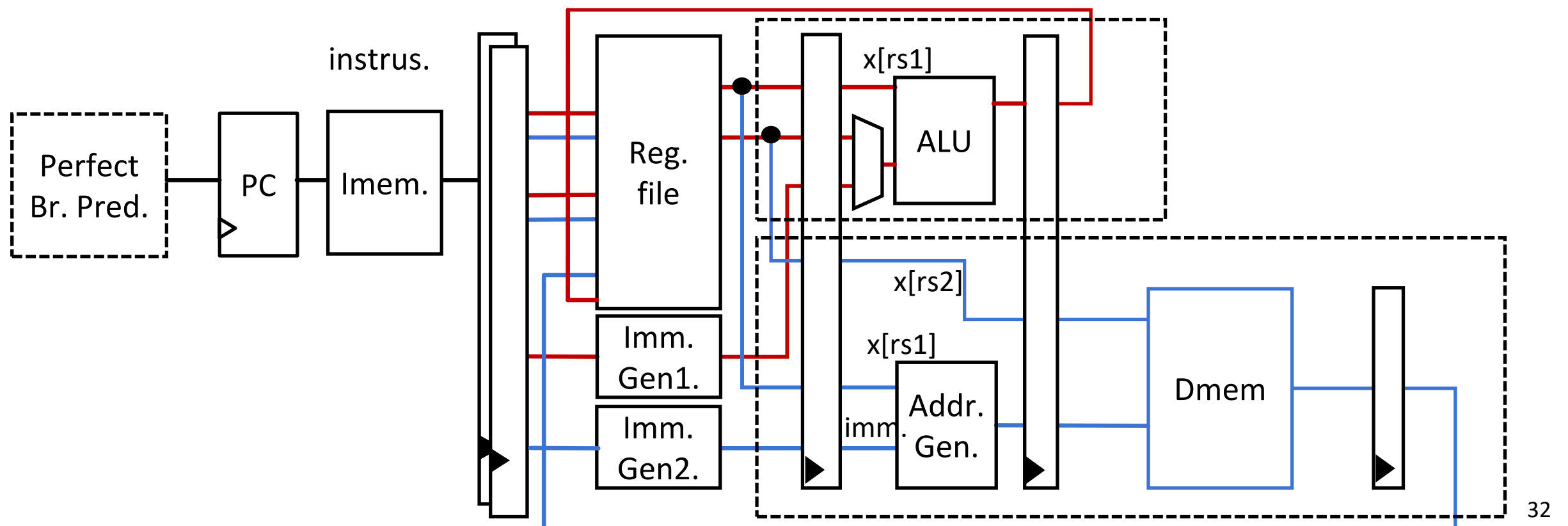
- In practice, we build different datapaths for different types of instructions
- E.g.



Multi-issue

- Ideally, issue two instructions with different type to ALU/mem datapaths

Instrucion type	cc1	cc2	cc3	cc4	cc5	cc6	cc7
ALU or branch	IF	ID	EX	WB			
Load or store	IF	ID	EX	MEM	WB		
ALU or branch		IF	ID	EX	WB		
Load or store		IF	ID	EX	MEM	WB	
ALU or branch			IF	ID	EX	WB	
Load or store			IF	ID	EX	MEM	WB



Multi-issue

```
for (int i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

1. <code>addi</code>	<code>s0,x0,s</code>	#initialize s0
2. <code>Loop:lw</code>	<code>t3,0(t1)</code>	#load array element
3. <code>add</code>	<code>t3,t3,s0</code>	#add s to \$t3
4. <code>sw</code>	<code>t3,0(t1)</code>	#store result
5. <code>addi</code>	<code>t1,t1,-4</code>	#t1 =t1-4
6. <code>bne</code>	<code>t1,t2,Loop</code>	#repeat loop if t1!=t2

Instrucion	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9
1.addi	IF	ID	EX	WB					
2.lw	IF	ID	EX	MEM	WB				
3.add		IF	ID	EX	WB				
4.sw		IF	ID	EX	MEM	WB			
5.addi			IF	ID	EX	WB			
nop			nop	nop	nop	nop	nop		
6.bne			IF	ID	EX	WB			
2.lw			IF	ID	EX	MEM	WB		
3.add				IF	ID	EX	WB		
4.sw				IF	ID	EX	MEM	WB	
5.addi					IF	ID	EX	WB	
nop					nop	nop	nop	nop	nop

Hazards!

Multi-issue

```
for (int i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

1. addi	s0,x0,1	#initialize s0
2. Loop:lw	t3,0(t1)	#load array element
3. add	t3,t3,s0	#add s to \$t3
4. sw	t3,0(t1)	#store result
5. addi	t1,t1,-4	#t1 =t1-4
6. bne	t1,t2,Loop	#repeat loop if t1!=t2

Instrucion	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9
1.addi	IF	ID	EX	WB					
2.lw	IF	ID	EX	MEM	WB				
nop		nop	nop	nop	nop				
nop		nop	nop	nop	nop	nop			
3.add			IF	ID	EX	WB			
4.sw			IF	ID	EX	MEM	WB		
5.addi				IF	ID	EX	WB		
nop				nop	nop	nop	nop	nop	
6.bne					IF	ID	EX	WB	
2.lw					IF	ID	EX	MEM	WB
... ..									

... ..

Multi-issue

- Loop unrolling & register renaming to optimize (also will be used in SIMD)

```

1. addi    s0,x0,s
2.L:lw     t3,0(t1)
3. lw      t4,-4(t1)
4. lw      t5,-8(t1)
5. lw      t6,-12(t1)
6. add     t3,t3,s0
7. add     t4,t4,s0
8. add     t5,t5,s0
9. add     t6,t6,s0
10. sw     t3,0(t1)
11. sw     t4,-4(t1)
12. sw     t5,-8(t1)
13. sw     t6,-12(t1)
14. addi   t1,t1,-16
15. bne    t1,t2,L
    
```

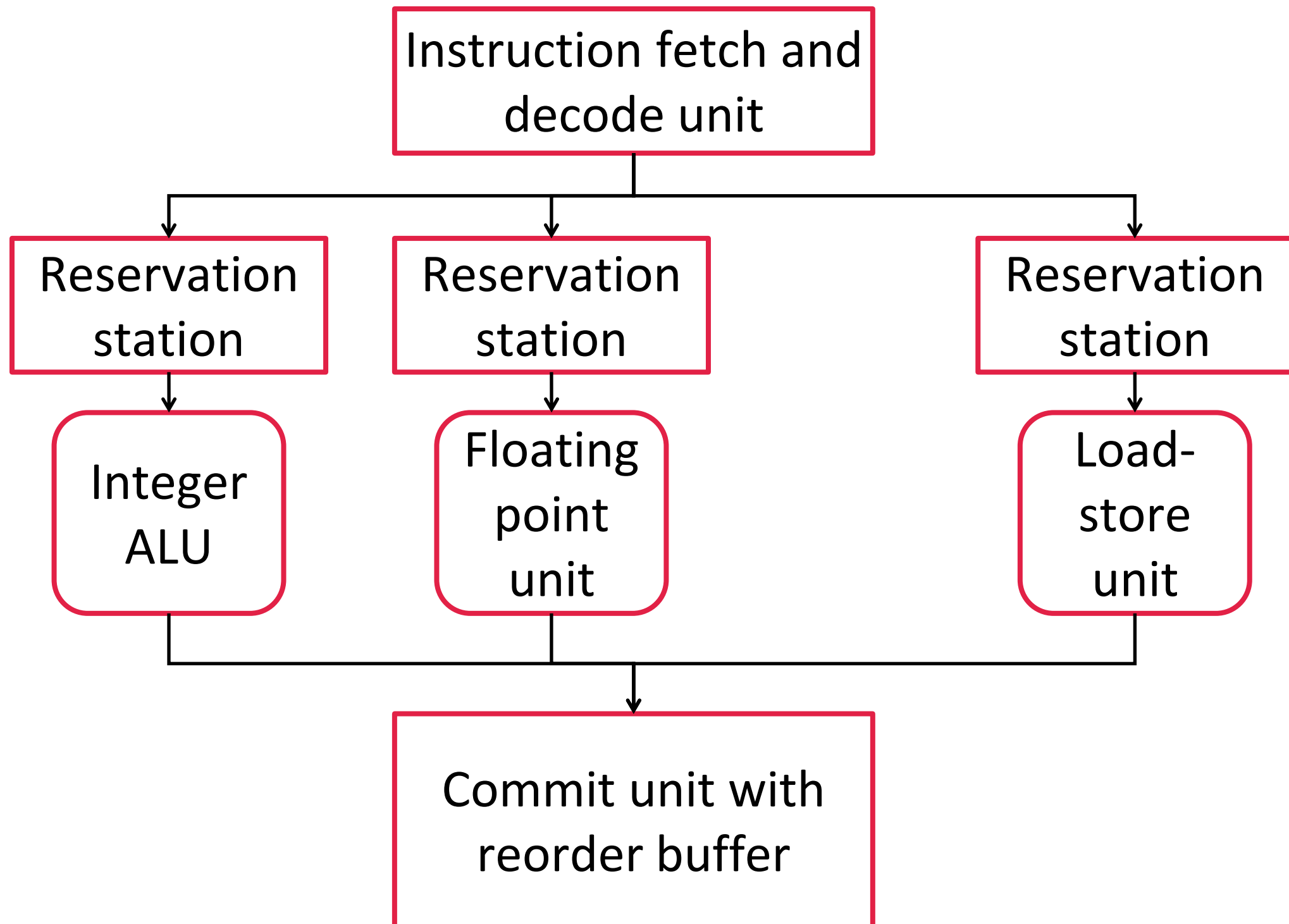
Instrucion	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9
1.addi	IF	ID	EX	WB					
2.lw t3	IF	ID	EX	MEM	WB				
nop		nop	nop	nop	nop				
3.lw t4		IF	ID	EX	MEM	WB			
6.add t3			IF	ID	EX	WB			
4.lw t5			IF	ID	EX	MEM	WB		
7.add t4				IF	ID	EX	WB		
5.lw t6				IF	ID	EX	MEM	WB	
8.add t5					IF	ID	EX	WB	
10.sw t3					IF	ID	EX	MEM	WB
9.add t6						IF	ID	EX	WB
11.sw t4						IF	ID	EX	MEM
14.addi							IF	ID	EX
12.sw t5							IF	ID	EX
15.bne								IF	ID
13.sw t6								IF	ID

Static vs. Dynamic multi-issue

- Static multi-issue
 - Package instructions into issue slots and detect hazards statically (at compile time mostly)
 - Hardware may also detect/resolve hazards
 - Also called VLIW (very long instruction word)
- Dynamic multi-issue
 - Package instructions into issue slots and detect hazards dynamically (during execution by hardware mostly)
 - Compiler may also help avoiding hazards
 - Also called superscalar

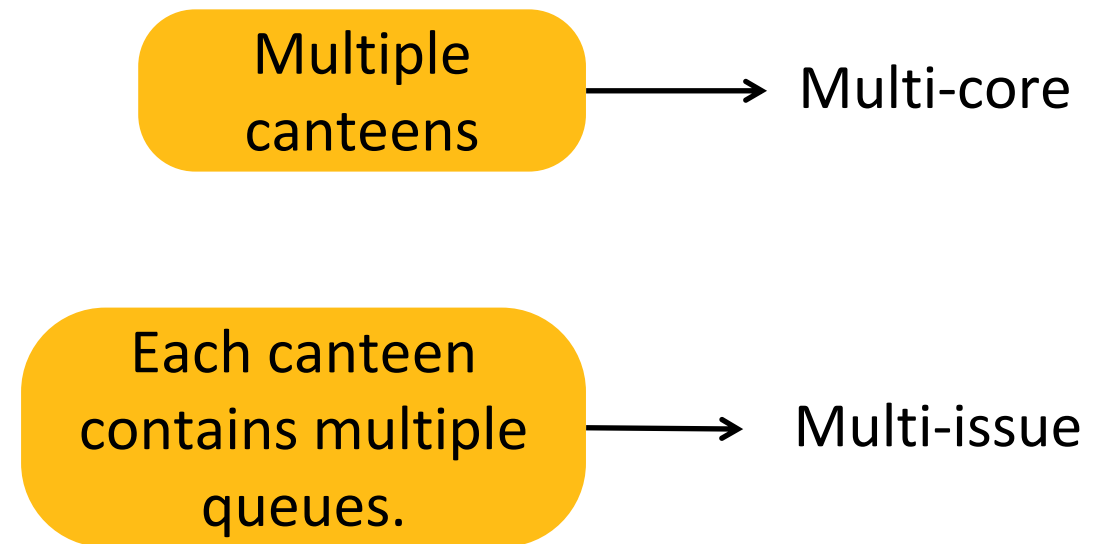
Instrucion type	cc1	cc2	cc3	cc4	cc5	cc6	cc7
ALU or branch	IF	ID	EX	WB			
Load or store	IF	ID	EX	MEM	WB		
ALU or branch		IF	ID	EX	WB		
Load or store		IF	ID	EX	MEM	WB	
ALU or branch			IF	ID	EX	WB	
Load or store			IF	ID	EX	MEM	WB

Hardware implementation of superscalar



Multi-issue Pitfalls

- Multi-issue is not multi-core;
- Multi-issue is not SIMD;
- Multi-issue can be combined with pipelining, SIMD, multi/hyper-threading, etc. to improve the performance of the processor;



Multi-issue Advancements-GPU

One streaming multi-processor inside an H100 GPU

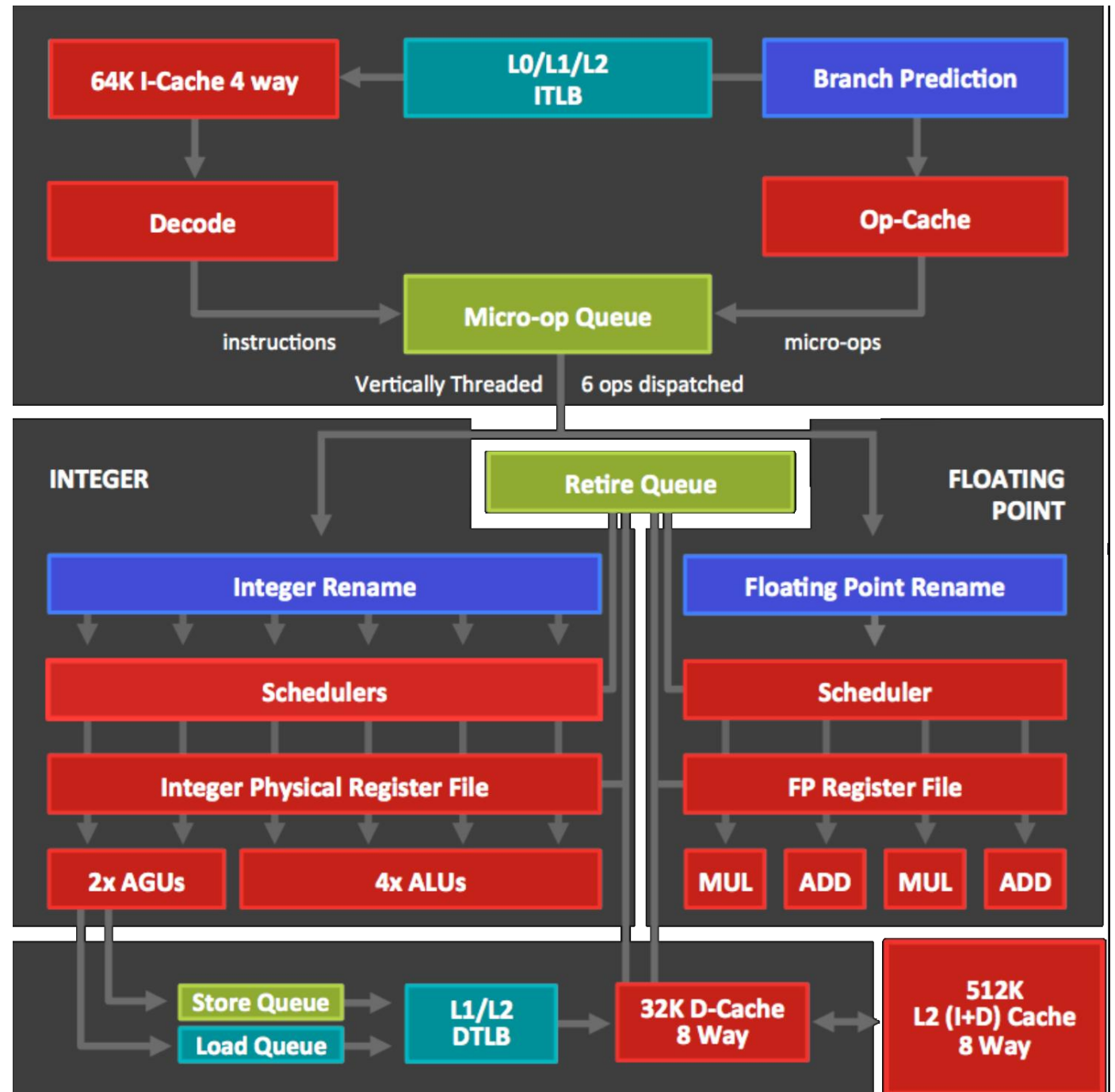
- **Multi-issue** can be combined with **SIMD**;
- In Fermi and later NVIDIA GPU architectures, the scheduler issues
 - 2 INT instructions or 2 single-point FP instructions or
 - 1 mixed INT or FPU or load or store or SFU instructions



Credit to Nvidia

Multi-issue Advancements-CPU

- **Multi-issue** can be combined with **multi/hyper-threading**;
- Thread-level parallelism (TLP)
 - To tolerate latency (e.g., cache miss)
 - To further improve throughput
 - To reduce context switch penalty
- Types of Multithreading
 - Fine-grained: threads scheduled cycle by cycle
 - Coarse-grained: threads scheduled on events (e.g., cache misses)/time quantum
 - SMT: simultaneous multi-threading



Credit to AMD

AMD CPU (Zen architecture) that supports simultaneous multithreading

Summary

- Instruction-level parallelism
 - Pipeline
 - Insert pipeline registers to execute the instructions stage by stage;
 - Multiple instructions co-exist in the pipeline to realize parallelism;
 - Induce (structural/data/control) hazards;
 - Strategies to deal with the hazards (insertion of bubbles, forwarding, hardware re-design, code scheduling, branch prediction, etc.);
 - Multi-issue
 - Multiple datapaths to execute multiple instructions in parallel;
 - Need to consider hazards as well;
 - Static vs. Dynamic multi-issue