CSCI-564 Advanced Computer Architecture

Pipelining Hazards

Credit to Dr.Bo Wu
Colorado School of Mines

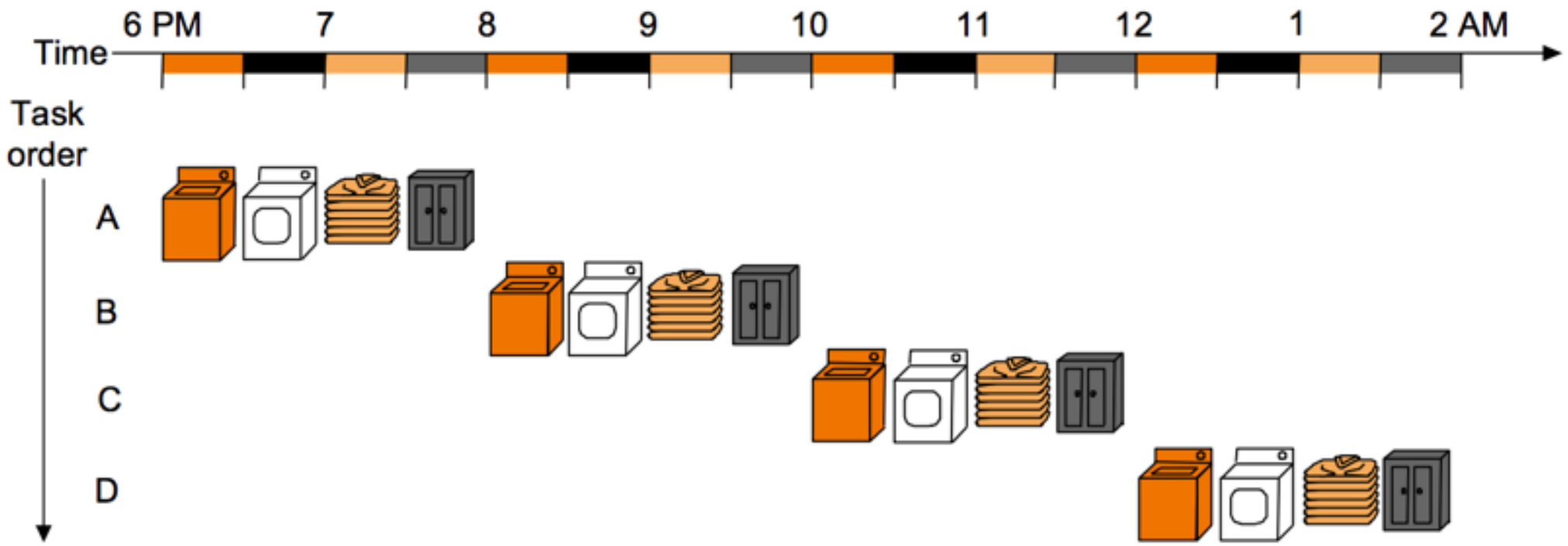
Optional Readings (After class)

Virtual memory III:
https://blog.httrack.com/blog/2014/04/05/a-story-of-realloc-and-laziness/

Virtual memory IV:
http://thebeardsage.com/virtual-memory-translation-lookaside-buffer-tlb/

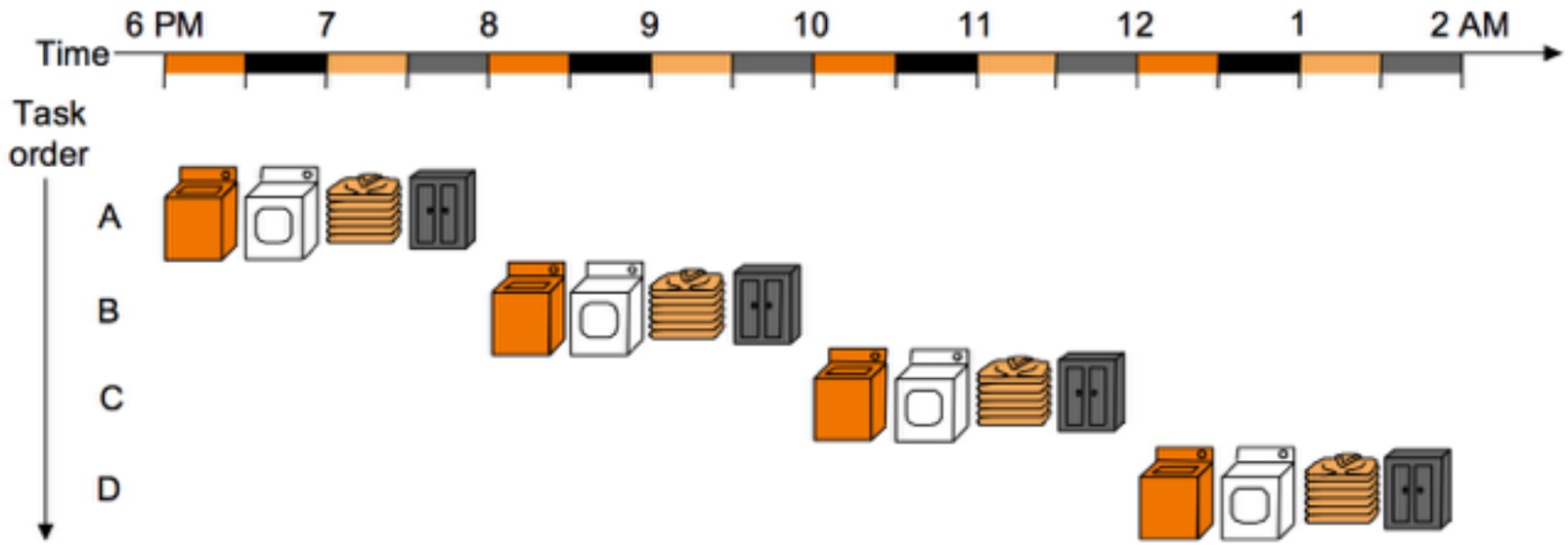
Wake up! Time to do laundry!

The Laundry Analogy

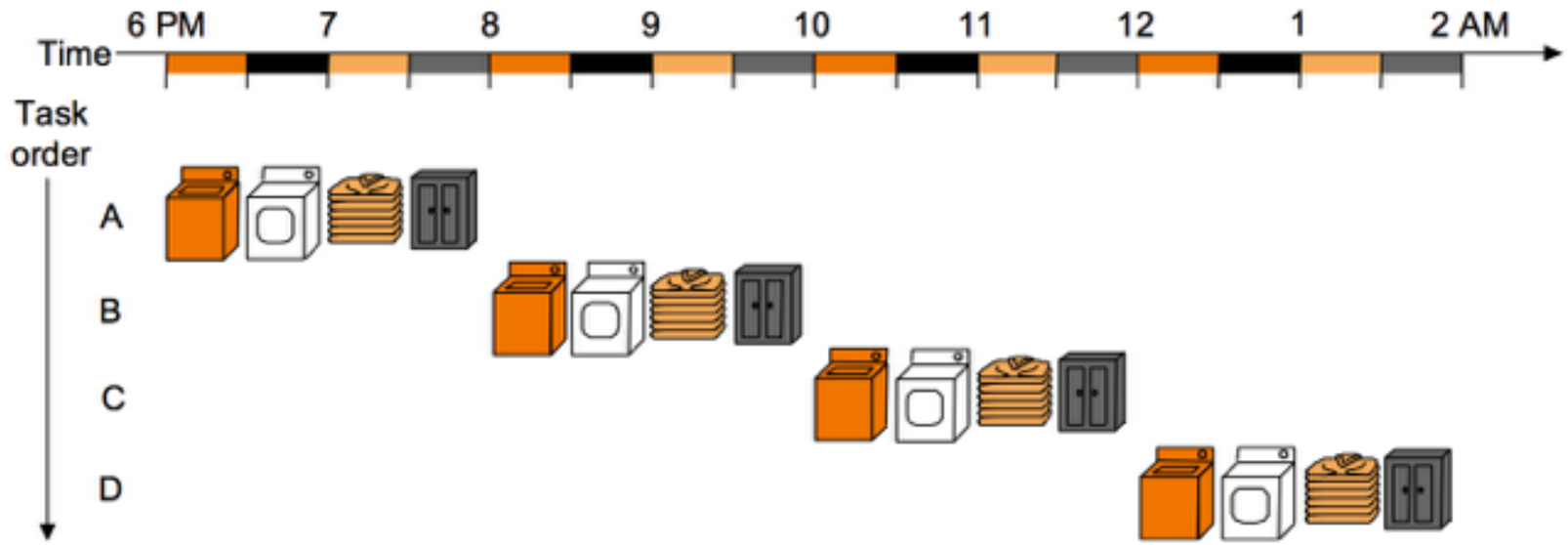


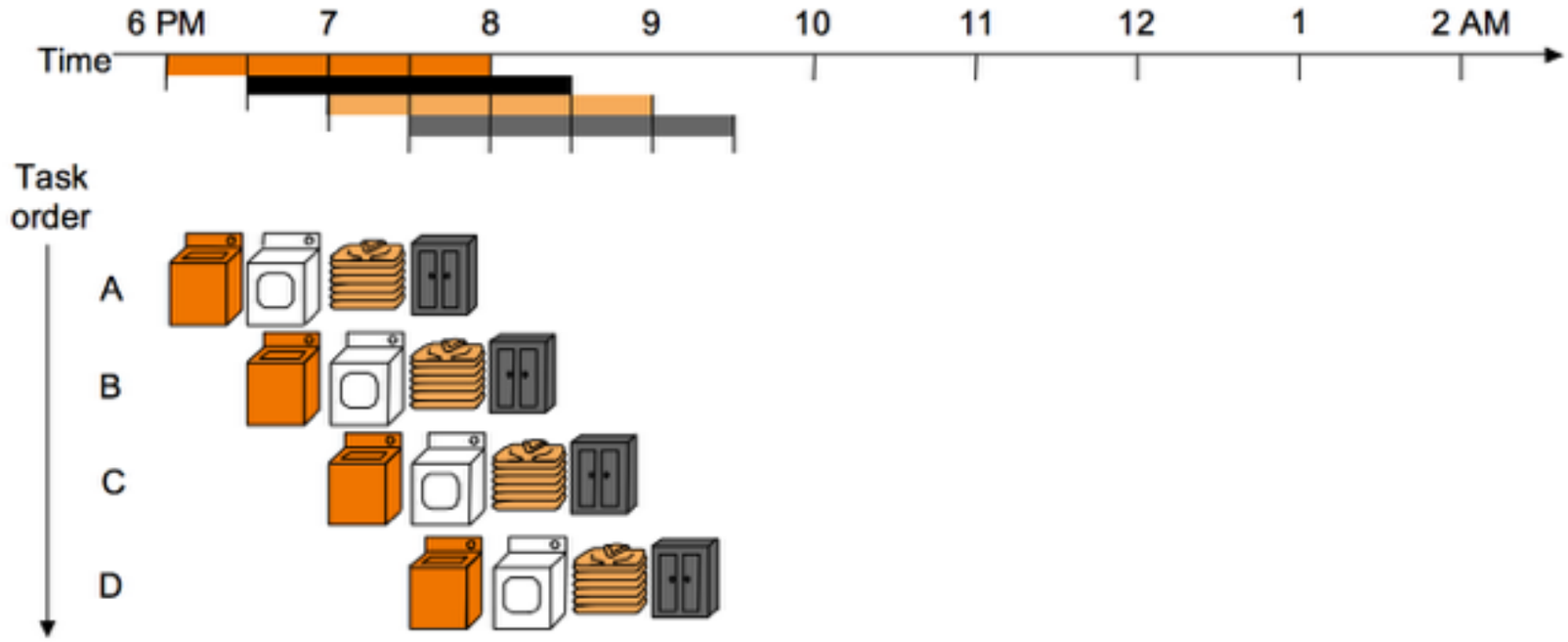
- Place one dirty load of clothes in the washer
- When the washer is finished, place the wet load in the dryer
- When the dryer is finished, take out the dry clothes and fold
- When folding is finished, ask your roommate (?) to put the clothes away

Pipelining Multiple Loads of Laundry

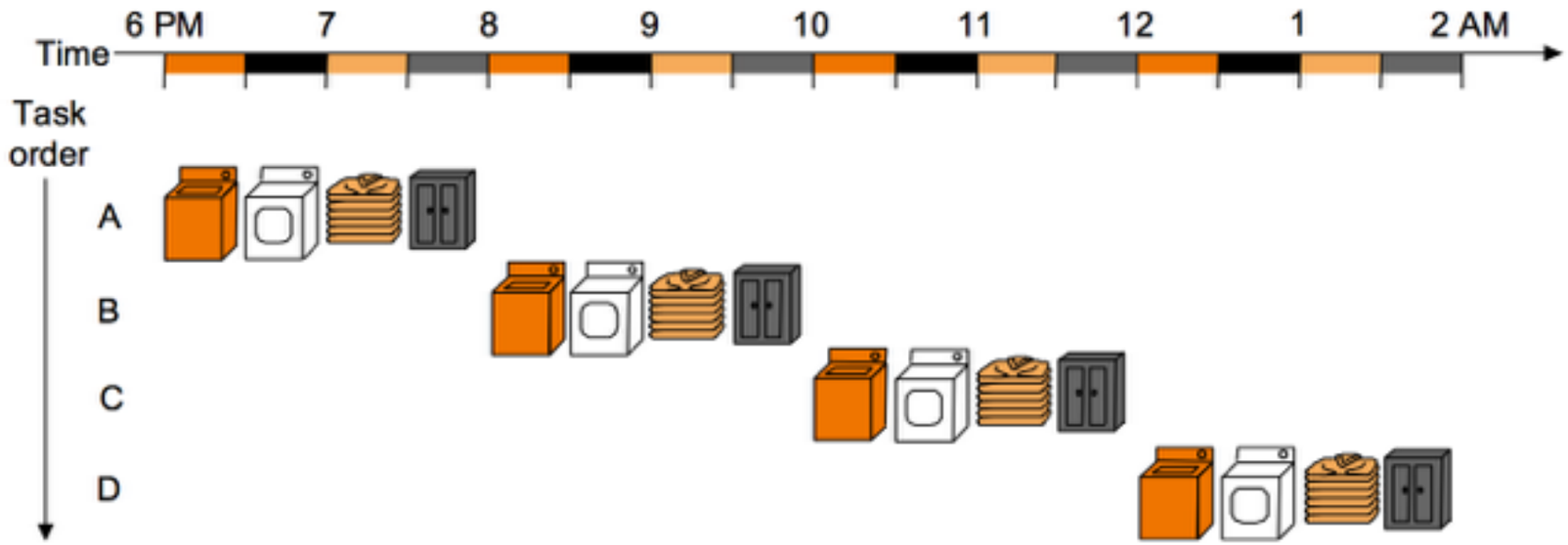


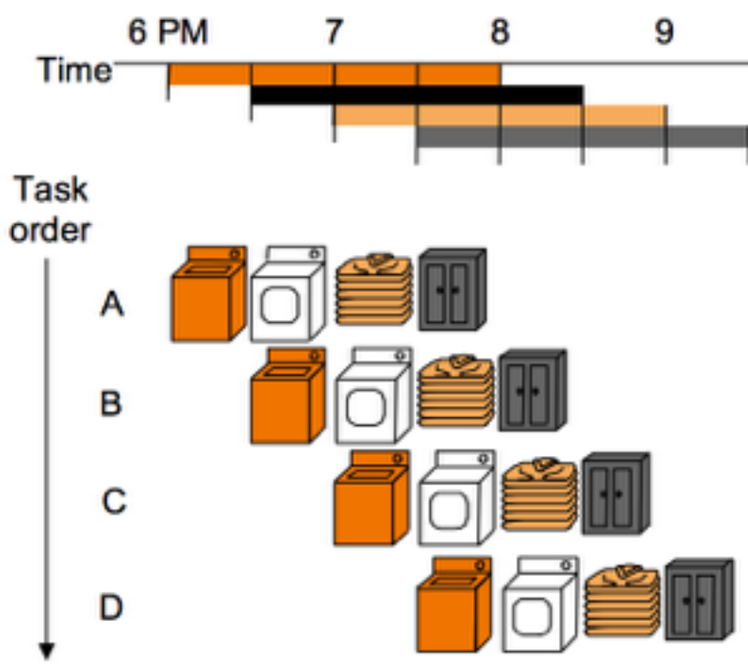
Pipelining Multiple Loads of Laundry





Pipelining Multiple Loads of Laundry





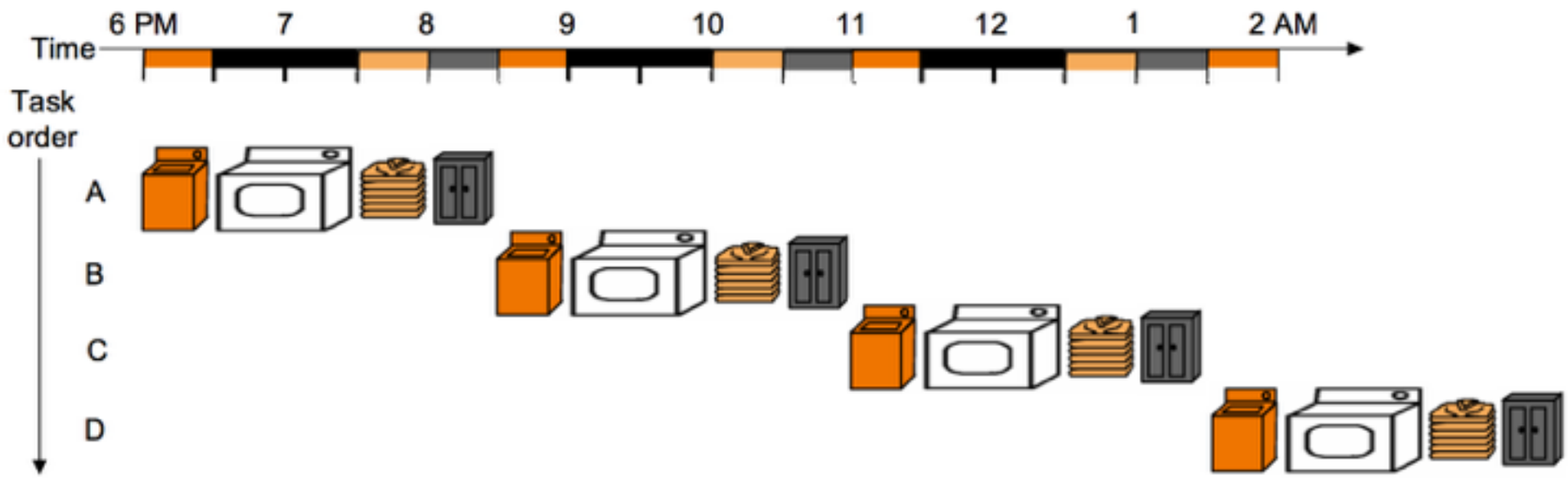
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

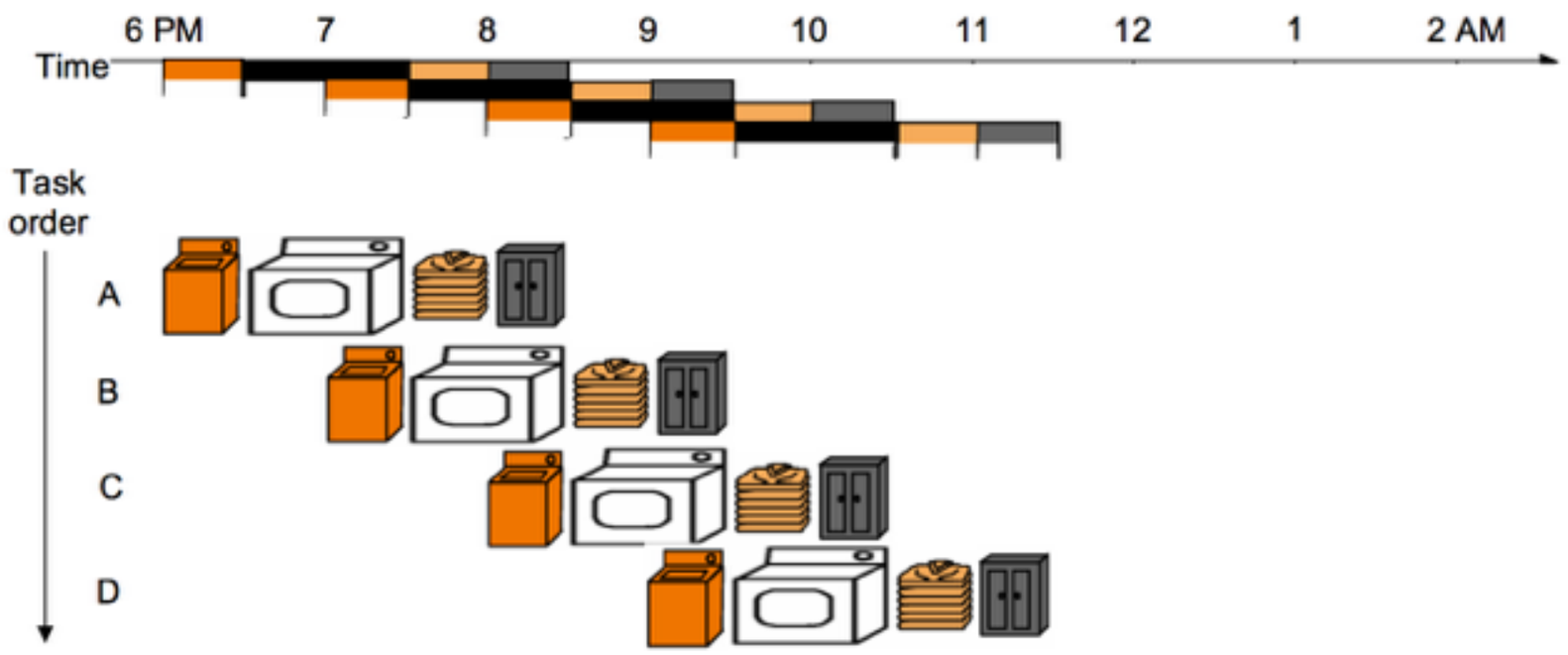
Question: We have a four-stage pipeline. Every stage takes 1 hour. How long does it take to finish 100 loads?

Question: We have a four-stage pipeline. Every stage takes 1 hour. How long does it take to finish 100 loads?

Question: We have a four-stage pipeline. Every stage takes 1 hour. How long does it take to finish N loads?

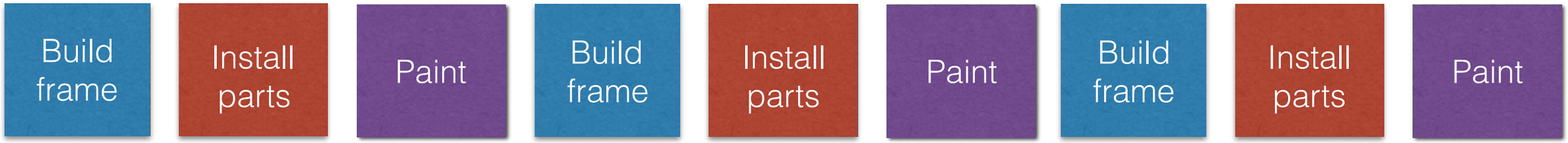
In Reality, maybe…



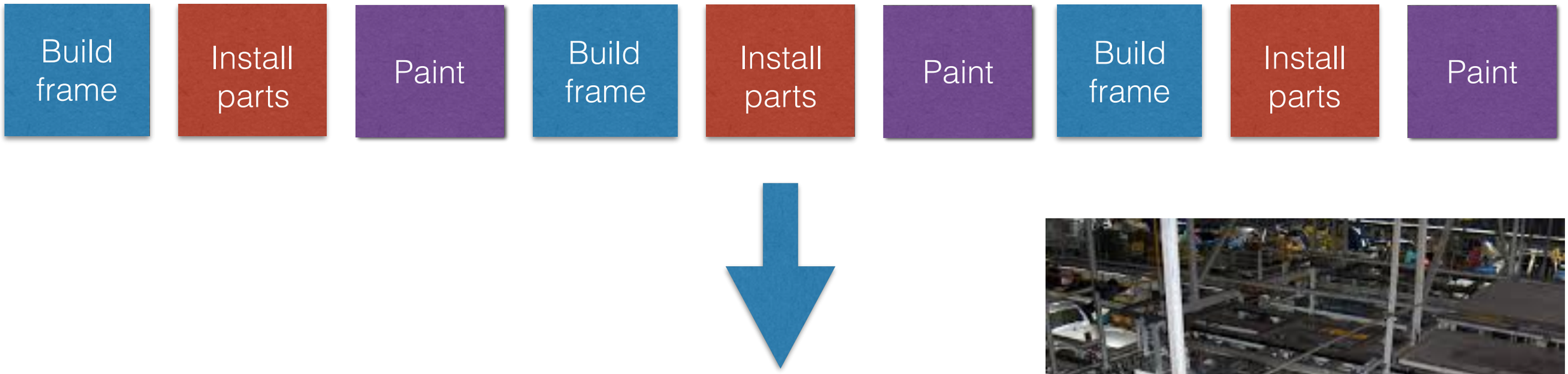


the slowest step decides throughput

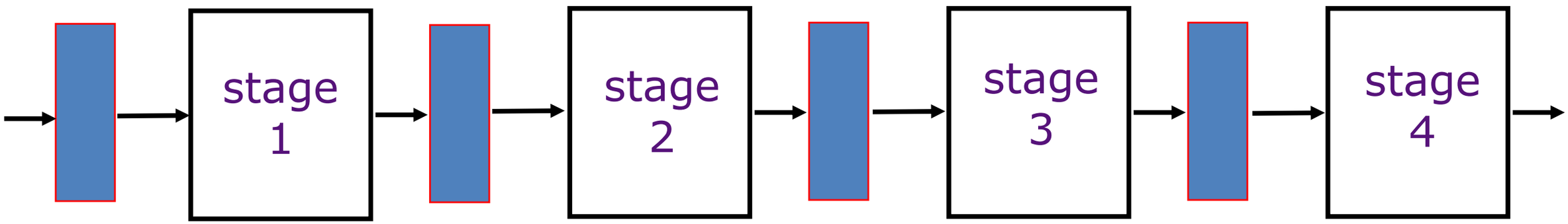
Pipelining is Everywhere



Pipelining is Everywhere

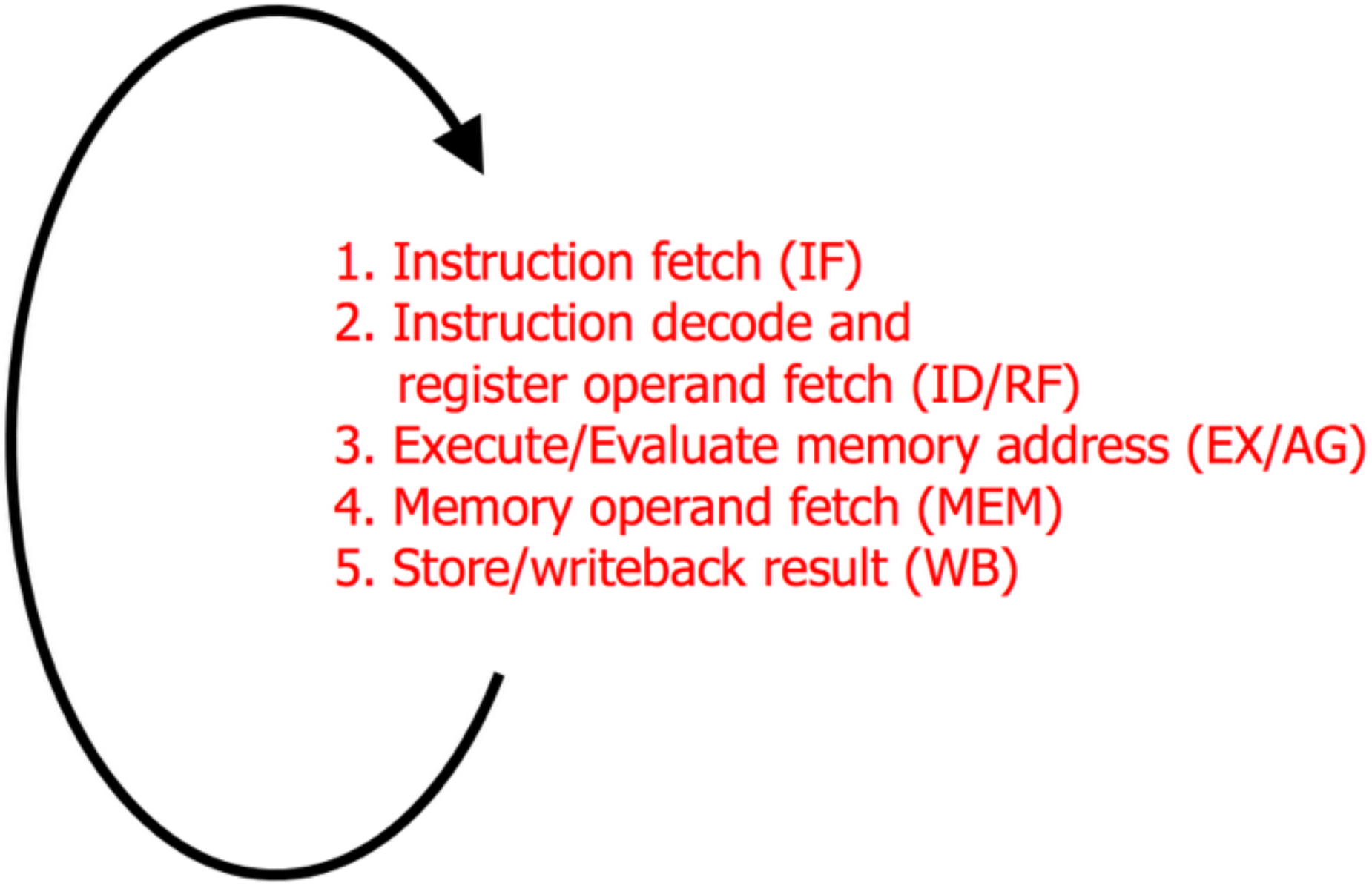


An Ideal Pipeline

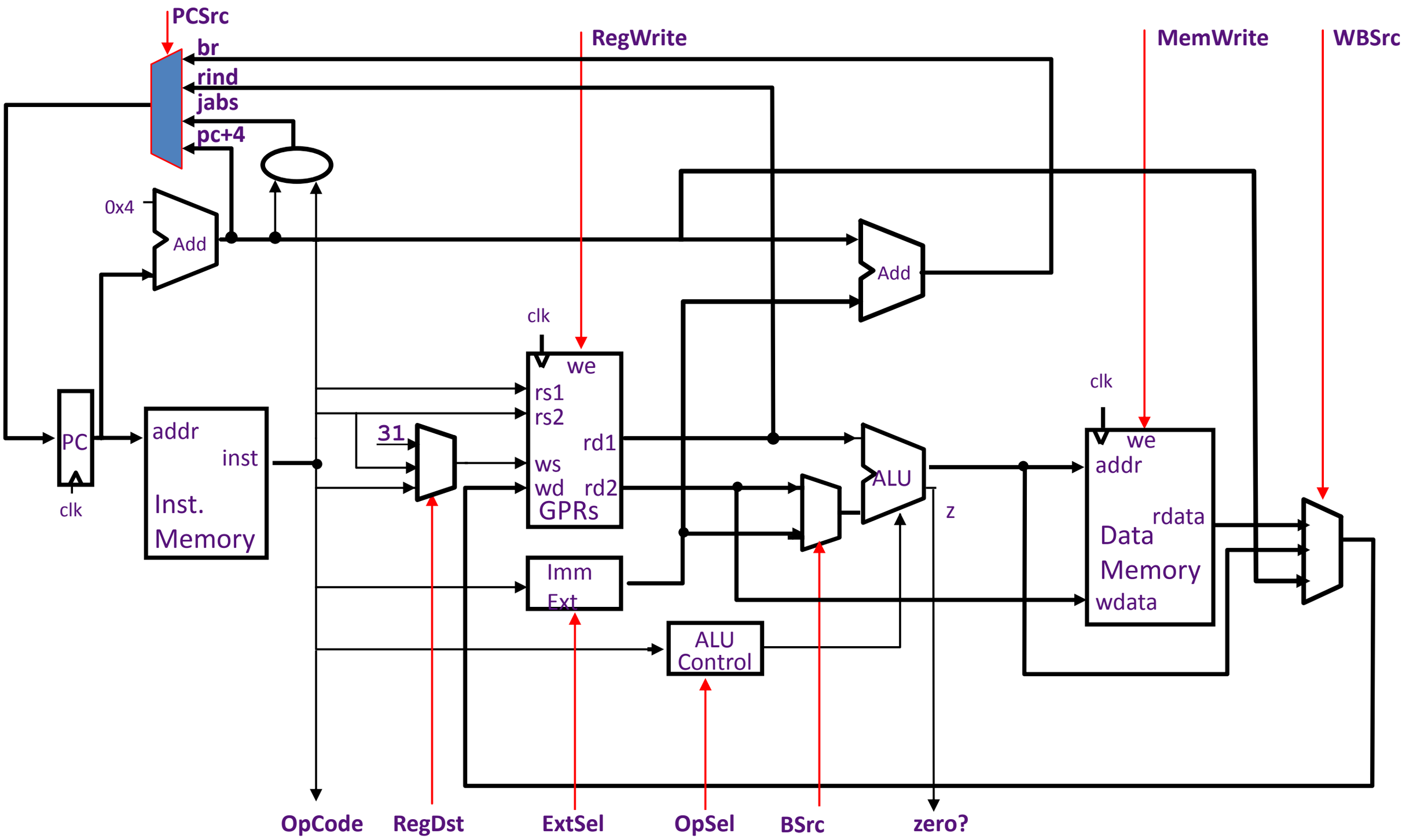


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages

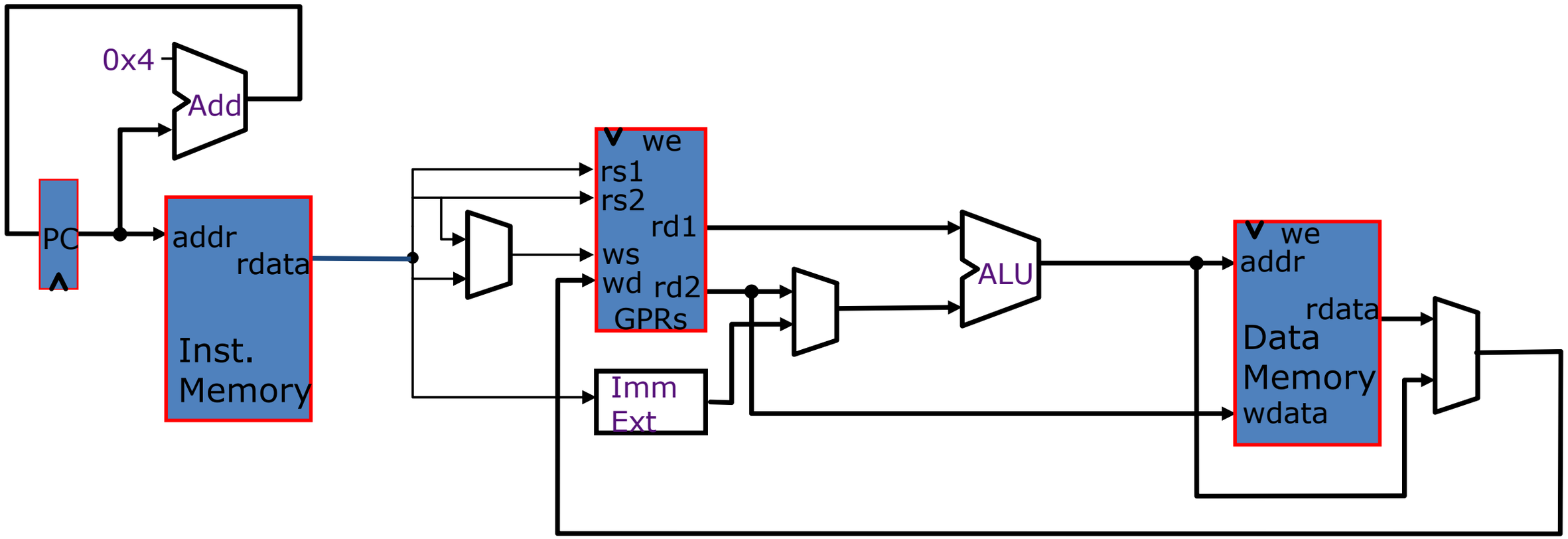
The Instruction Execution Cycle



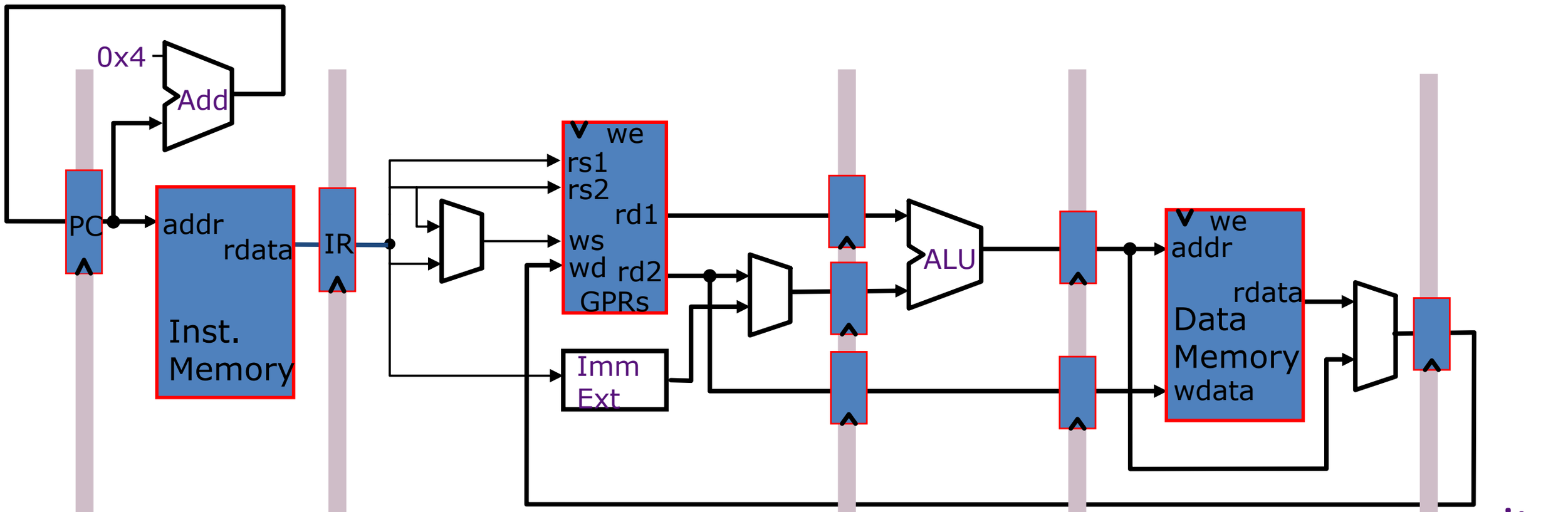
Unpipelined Datapath for MIPS



Simplified Unpipelined Datapath



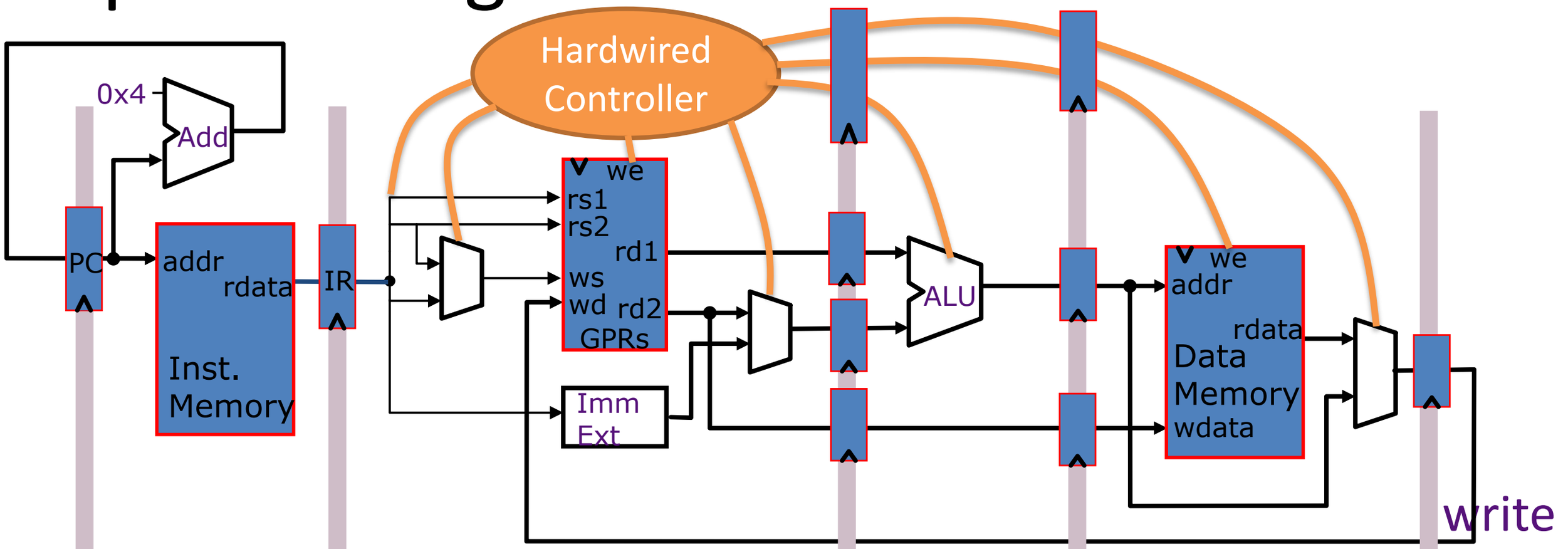
Pipelined Datapath



Clock period can be reduced by dividing the execution of an instruction into multiple cycles

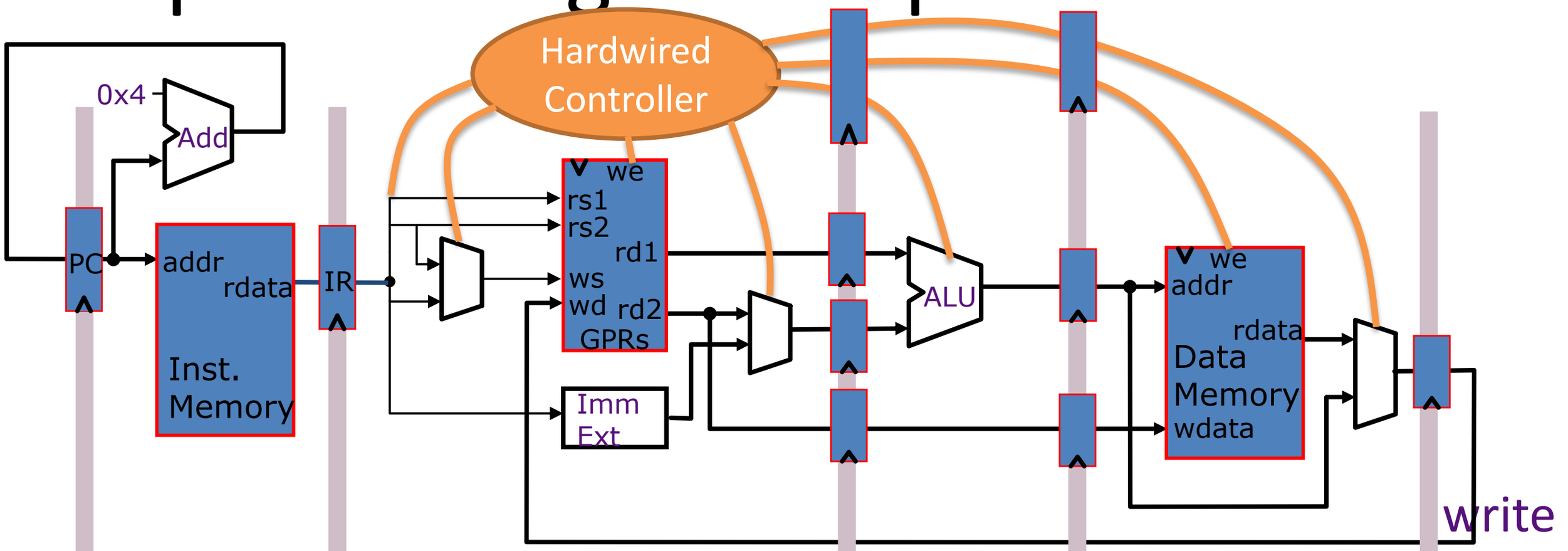
$$t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \; (= t_{DM} \; \textit{probably})$$

Pipeline Diagrams: Transactions vs. Time



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Pipeline Diagrams: Space vs. Time

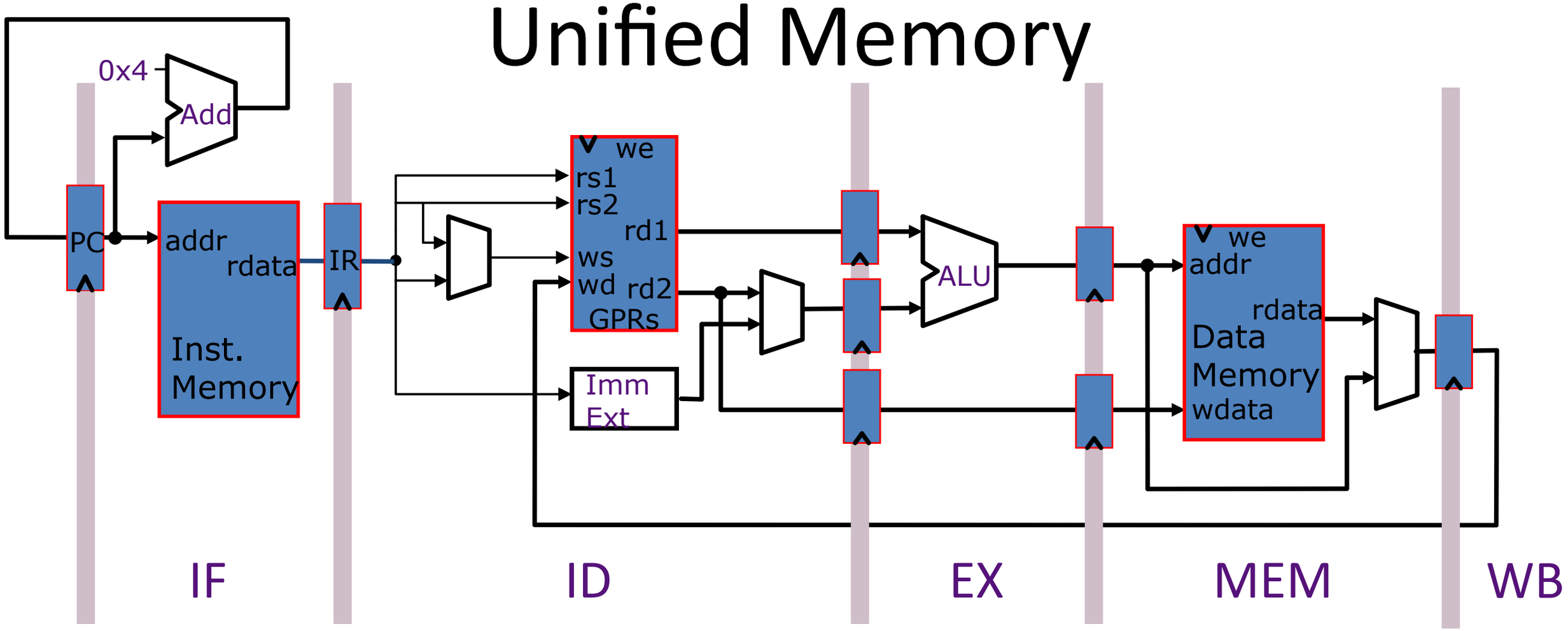


Resources	*time*	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
	IF	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	I_4	I_5			
	EX			I_1	I_2	I_3	I_4	I_5		
	MA				I_1	I_2	I_3	I_4	I_5	
	WB					I_1	I_2	I_3	I_4	I_5

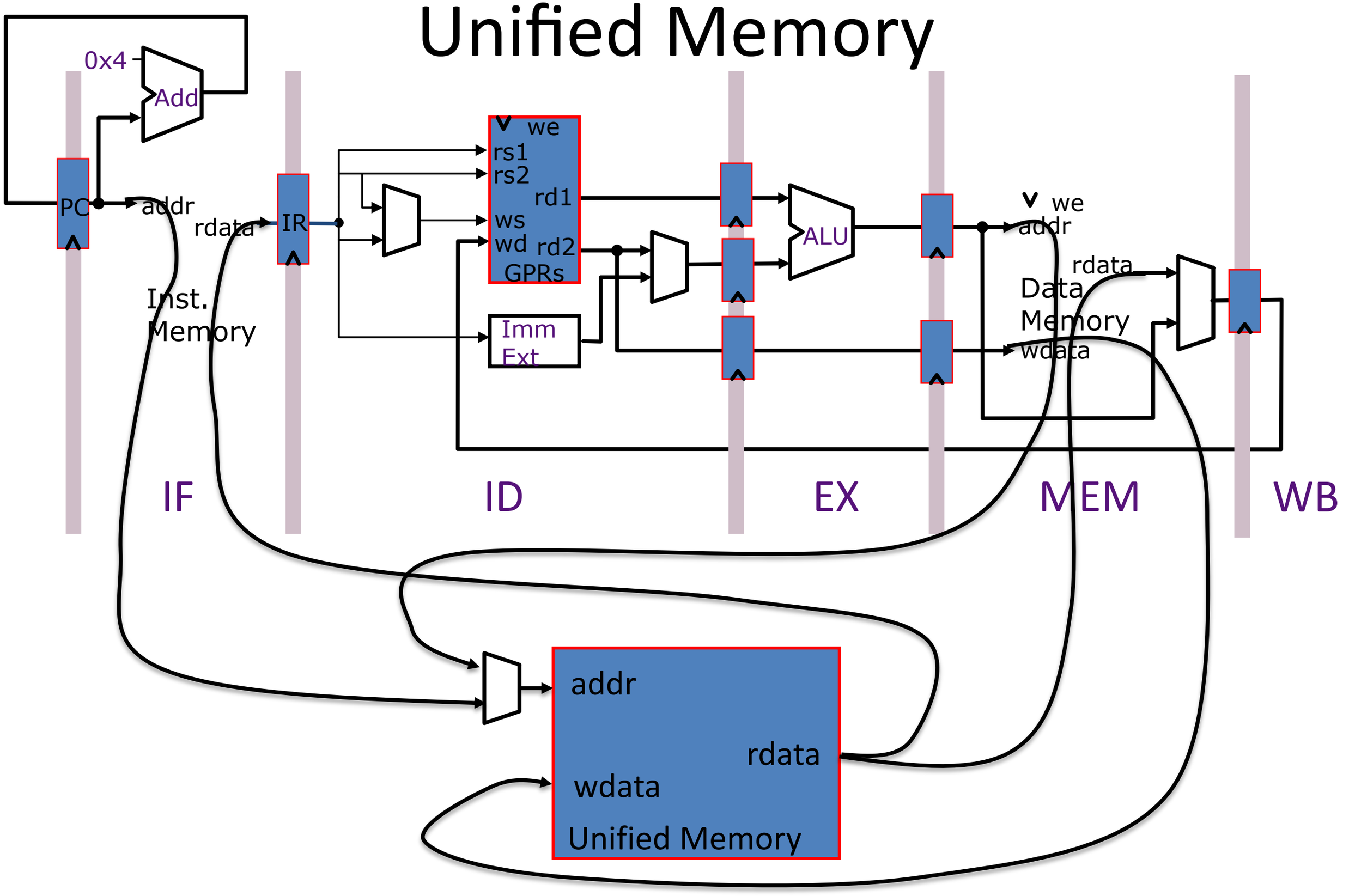
Instructions Interact With Each Other in Pipeline

- **Structural Hazard:** An instruction in the pipeline needs a resource being used by another instruction in the pipeline
- **Data Hazard:** An instruction depends on a data value produced by an earlier instruction
- **Control Hazard:** Whether or not an instruction should be executed depends on a control decision made by an earlier instruction

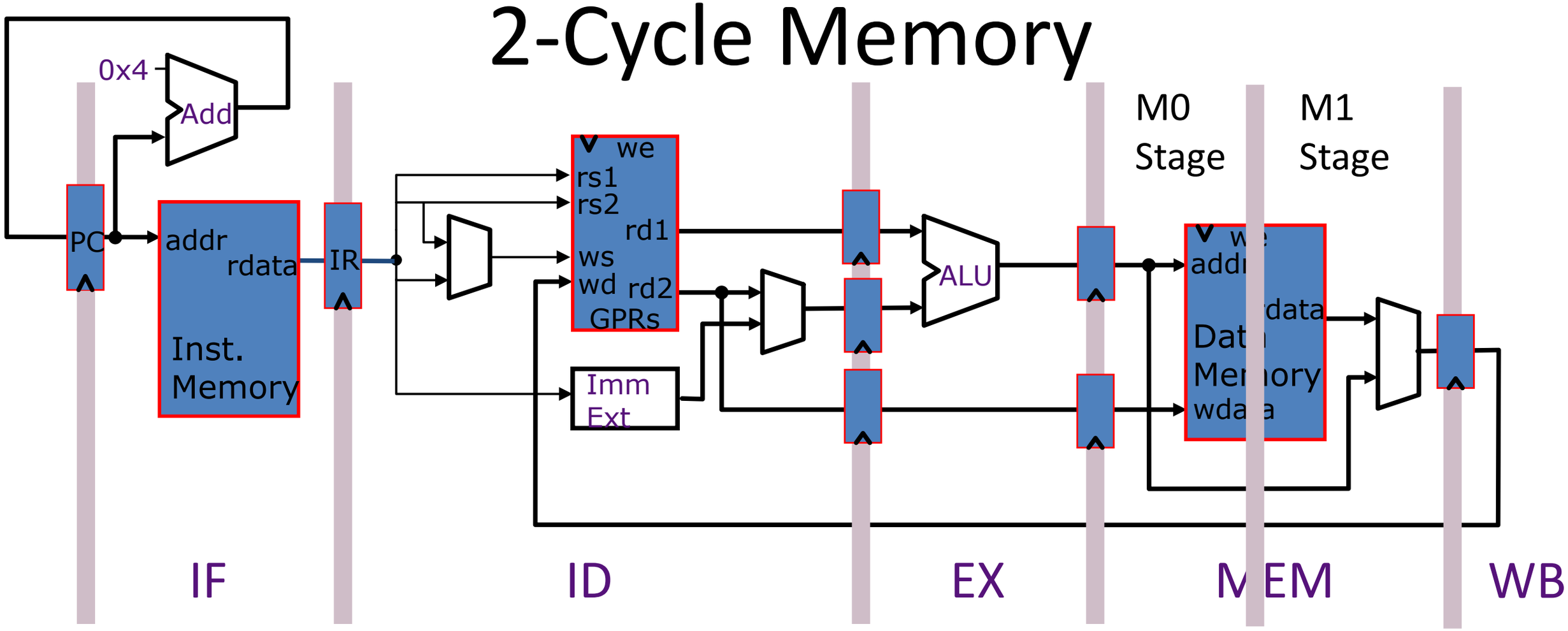
Example Structural Hazard: Unified Memory



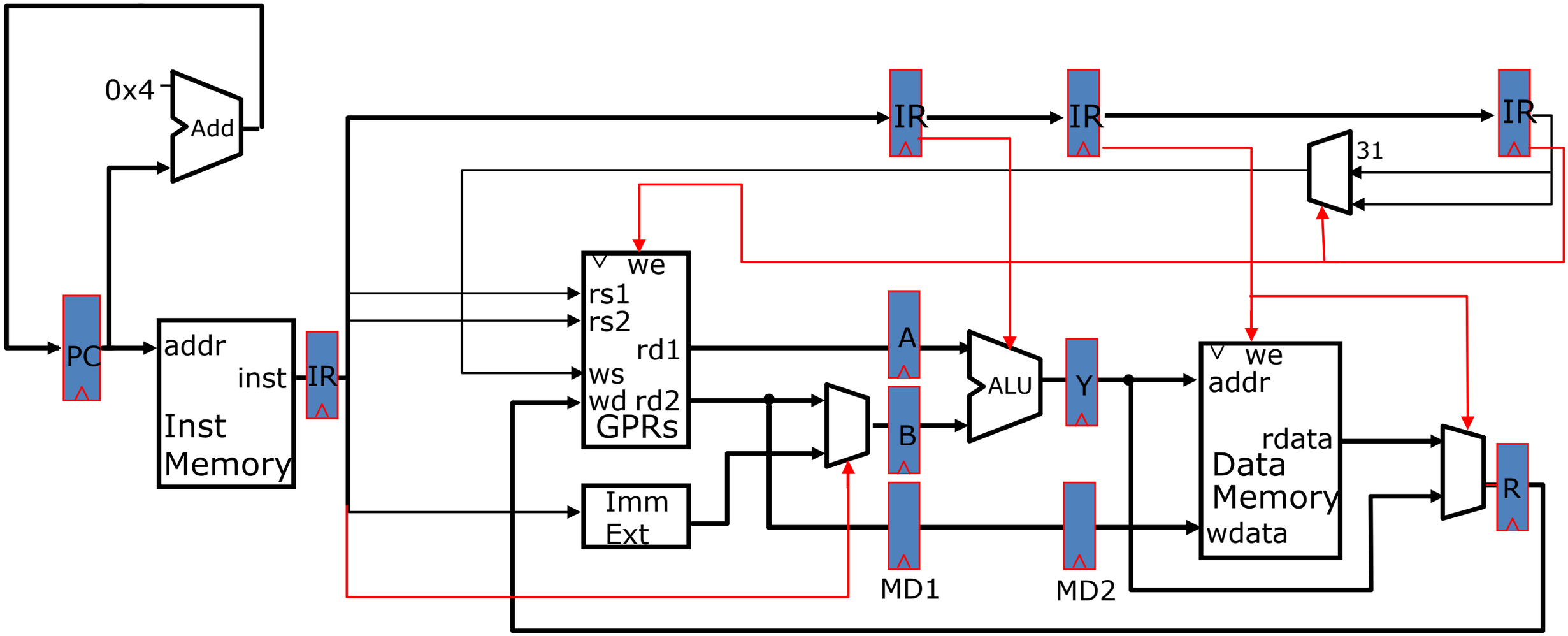
Example Structural Hazard: Unified Memory



Example Structural Hazard: 2-Cycle Memory



Example Data Hazard



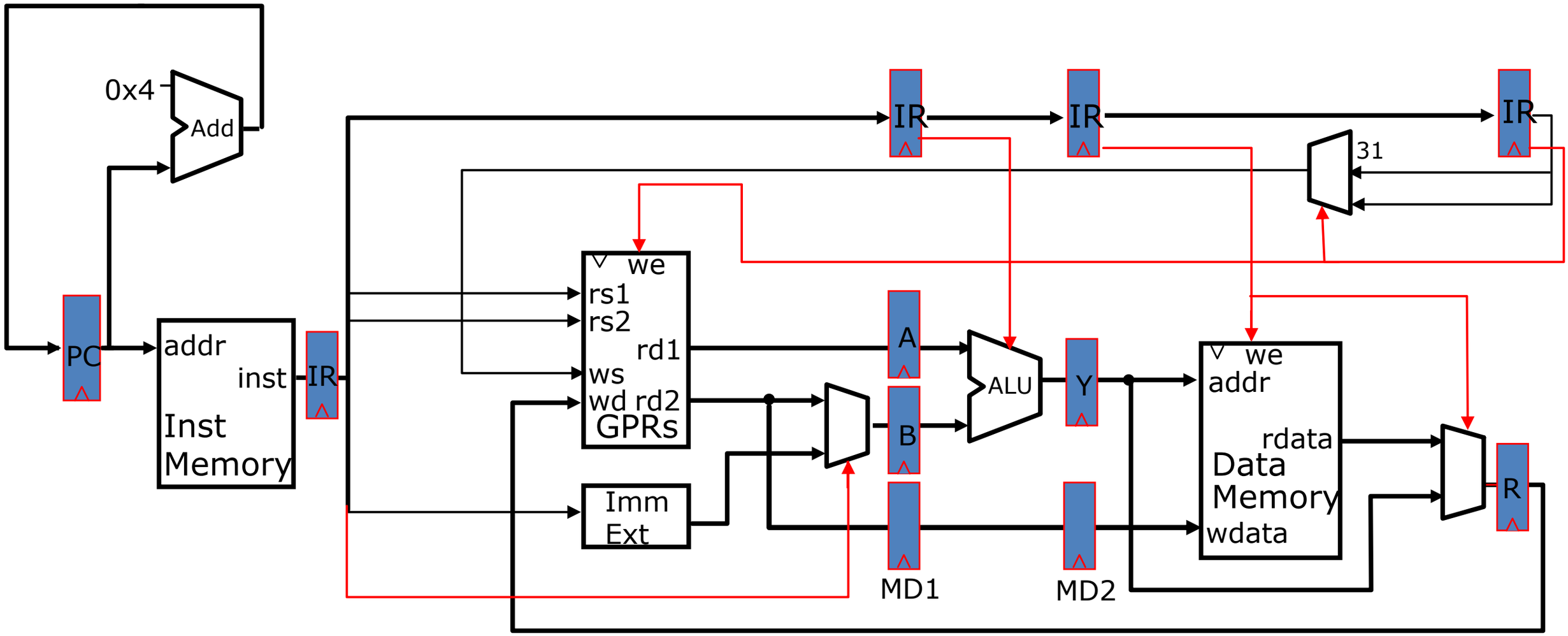
...

r1 ← r0 + 10 (ADDI R1, R0, #10)

r4 ← r1 + 17 (ADDI R4, R1, #17)

...

Example Data Hazard



...

r1 ← r0 + 10 (ADDI R1, R0, #10)

r4 ← r1 + 17 (ADDI R4, R1, #17)

r1 is stale. Oops!

...

Data Dependence

Flow dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2

$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

Output-dependence

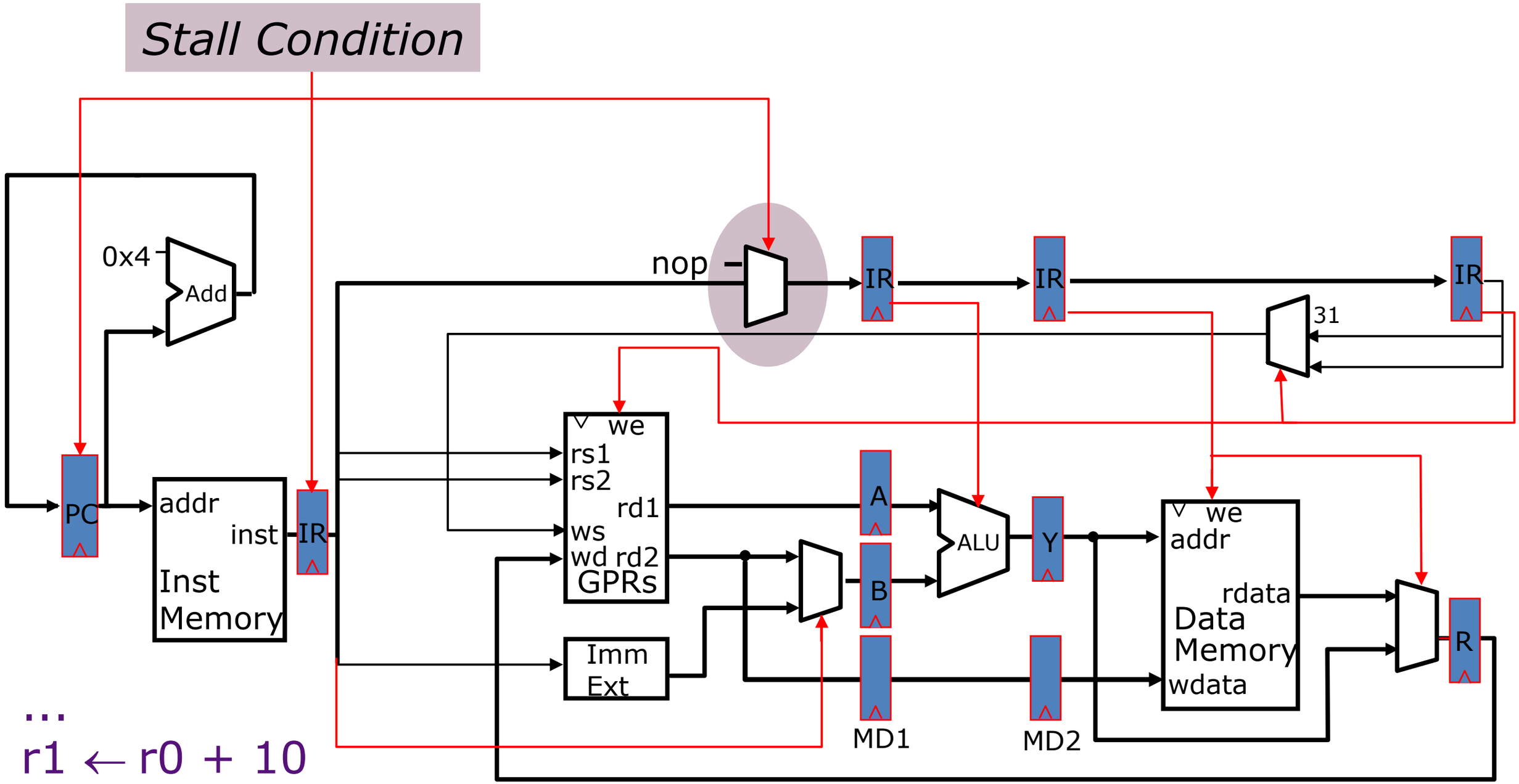
$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Resolving Data Hazards with Stalls



...

r1 ← r0 + 10

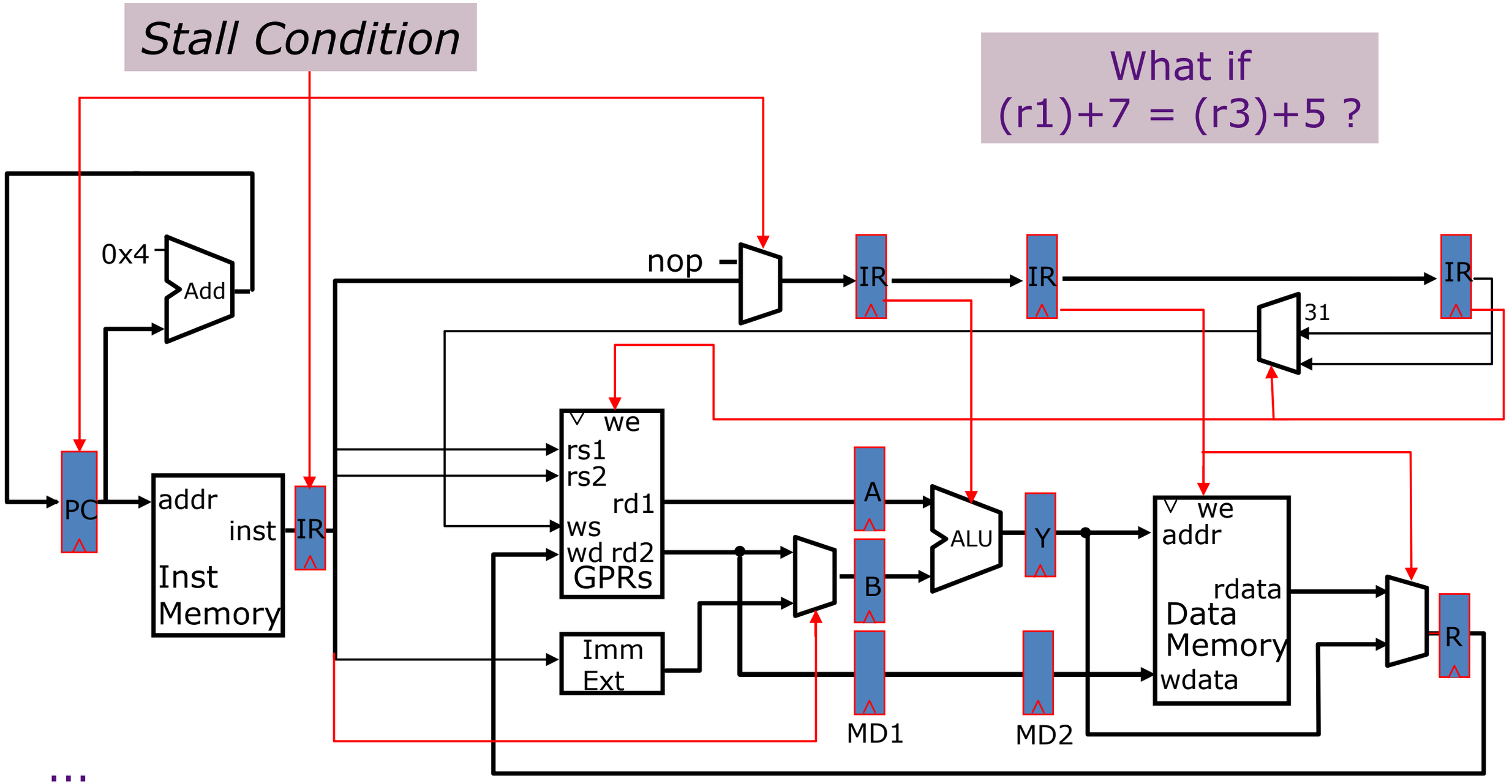
r4 ← r1 + 17

...

Hazards due to Loads & Stores

Stall Condition

What if
(r1)+7 = (r3)+5 ?



...
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
...

Is there any possible data hazard in this instruction sequence?

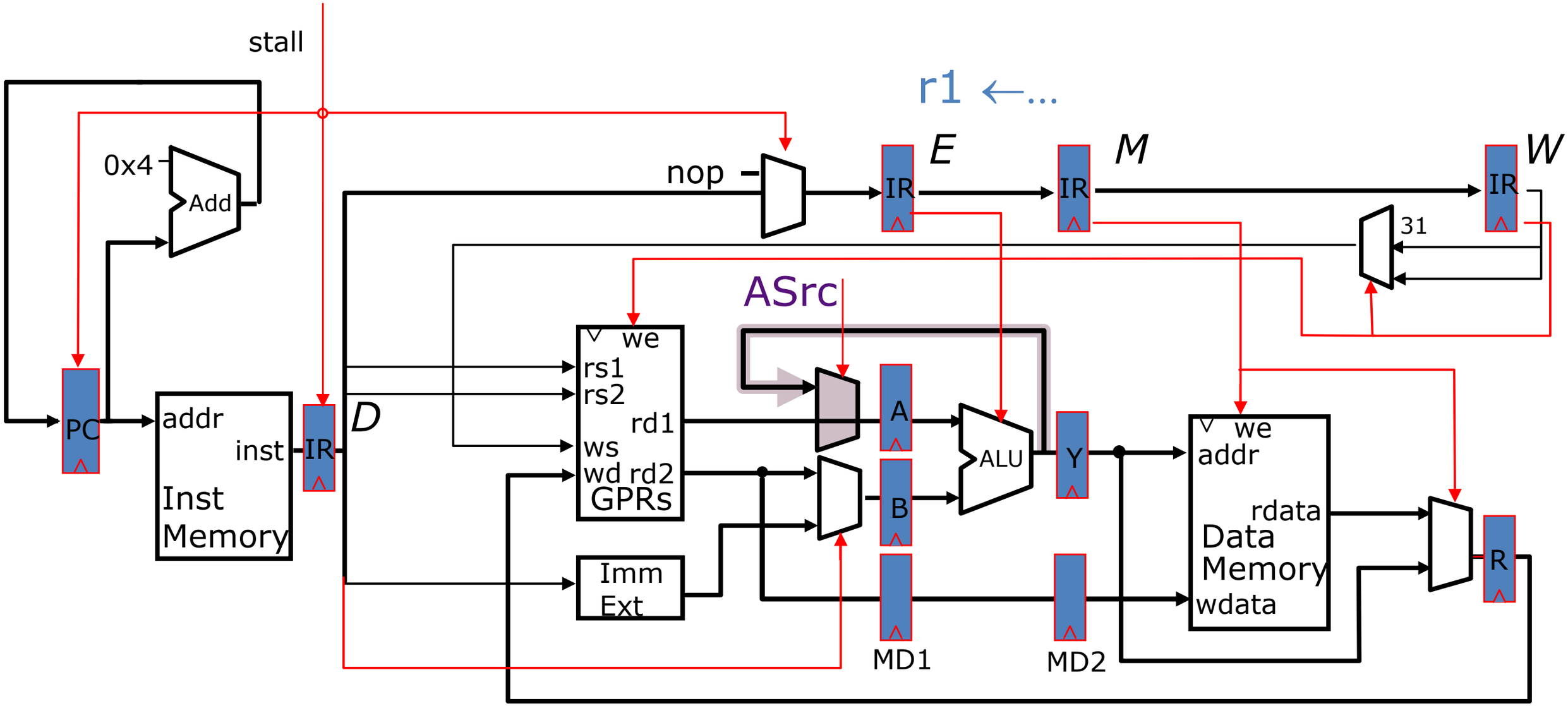
Data Hazards Due to Loads and Store

- Example instruction sequence
  - Mem[Regs[r1] + 7] <- Regs[r2]
  - Regs[r4] <- Mem[Regs[r3] + 5]

Data Hazards Due to Loads and Store

- Example instruction sequence
  - Mem[Regs[r1] + 7] <- Regs[r2]
  - Regs[r4] <- Mem[Regs[r3] + 5]
- What if Regs[r1]+7 == Regs[r3]+5 ?
  - Writing and reading to/from the same address
  - Hazard is avoided because our memory system completes writes in a single cycle
  - More realistic memory system will require more careful handling of data hazards due to loads and stores

Adding Bypassing to the Datapath

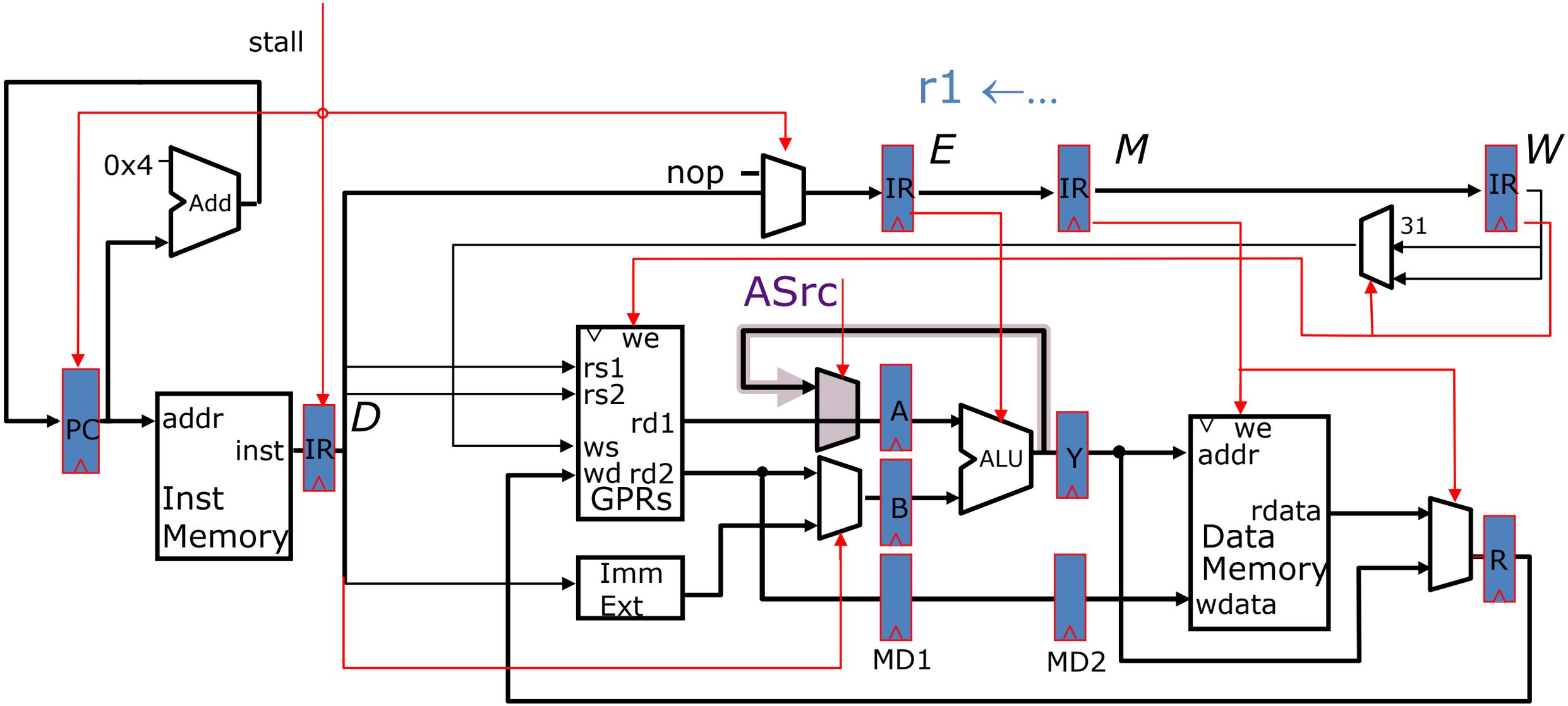


...

(I_1) r1 ← r0 + 10

(I_2) r4 ← r1 + 17

Adding Bypassing to the Datapath



When does <u>this</u> bypass help?

...

(I_1) r1 ← r0 + 10

(I_2) r4 ← r1 + 17

---

r1 ← Mem[r0 + 10]

r4 ← r1 + 17

JAL 500

r4 ← r31 + 1

Bypassing in Action

- Consider this 8-stage pipeline

- For the following pairs of instructions, how many stalls will the second instruction experience (with and without bypassing)?

ADD R1+R2→R3
ADD R3+R4→R5

LD [R1]→R2
ADD R2+R3→R4

LD [R1]→R2
SD [R2]←R3

LD [R1]→R2
SD [R3]←R2

Bypassing in Action

- Consider this 8-stage pipeline

- For the following pairs of instructions, how many stalls will the second instruction experience (with and without bypassing)?

ADD R1+R2→R3 ADD R3+R4→R5	without: 5 with: 1
LD [R1]→R2 ADD R2+R3→R4	without: 5 with: 3
LD [R1]→R2 SD [R2]←R3	without: 5 with: 3
LD [R1]→R2 SD [R3]←R2	without: 5 with: 1

Control Hazards

- What do we need to calculate next PC?

Control Hazards

- What do we need to calculate next PC?
  - For Jumps
    - Opcode, offset and PC
  - For Jump Register
    - Opcode and Register value
  - For Conditional Branches
    - Opcode, PC, Register (for condition), and offset
  - For all other instructions
    - Opcode and PC

Opcode Decoding Bubble

(assuming no branch delay slots for now)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r3 ← (r2) + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3
(I_4)						IF_4	IF_4	ID_4	EX_4 MA_4 WB_4

Opcode Decoding Bubble

(assuming no branch delay slots for now)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r3 ← (r2) + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3
(I_4)						IF_4	IF_4	ID_4	EX_4 MA_4 WB_4

Resource Usage

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	nop	I_2	nop	I_3	nop	I_4		
ID		I_1	nop	I_2	nop	I_3	nop	I_4	
EX			I_1	nop	I_2	nop	I_3	nop	I_4
MA				I_1	nop	I_2	nop	I_3	nop I_4
WB					I_1	nop	I_2	nop	I_3 nop I_4

nop ⇒ *pipeline bubble*

Opcode Decoding Bubble

(assuming no branch delay slots for now)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r3 ← (r2) + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3
(I_4)						IF_4	IF_4	ID_4	EX_4 MA_4 WB_4

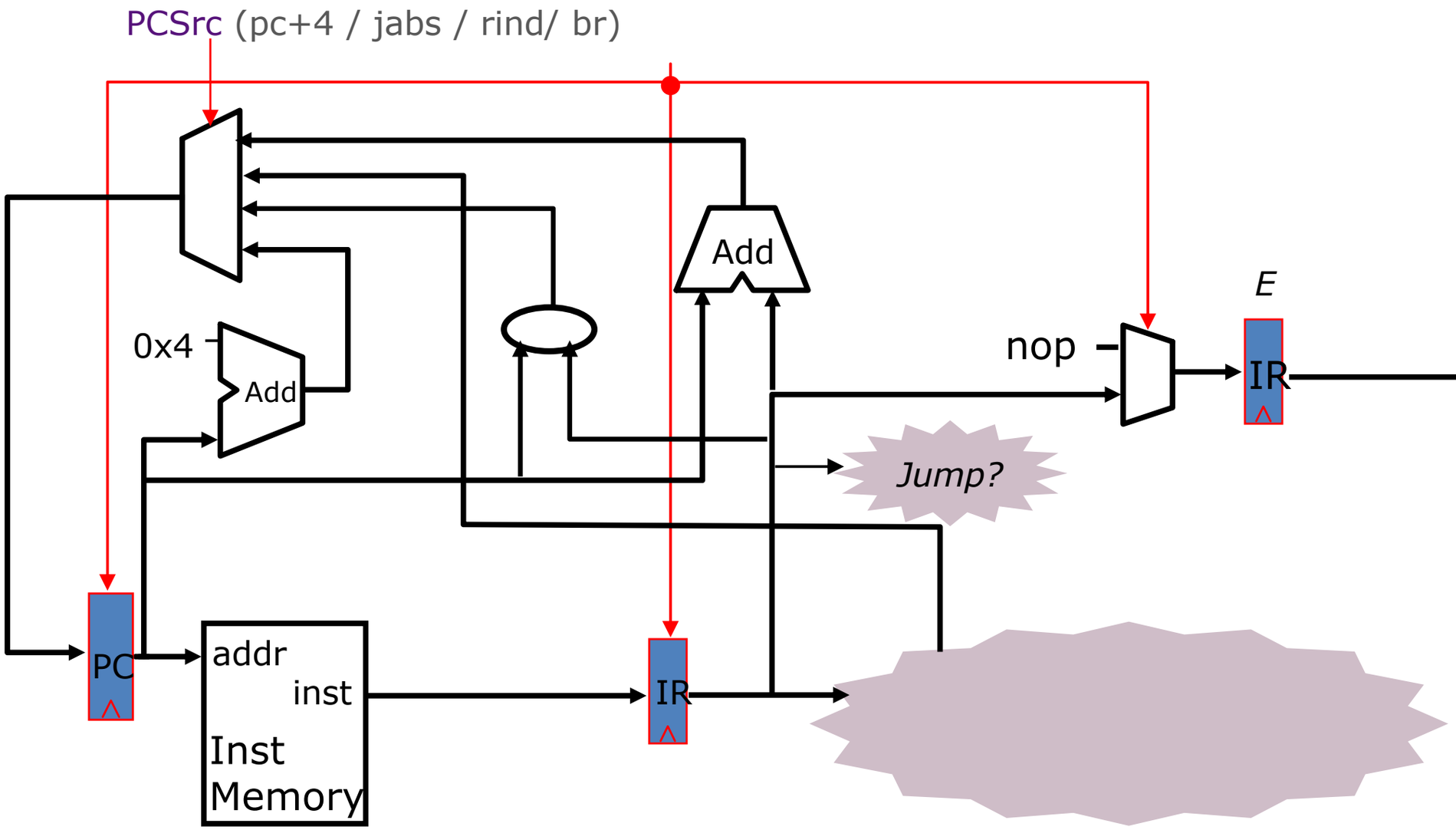
Resource Usage

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	nop	I_2	nop	I_3	nop	I_4		
ID		I_1	nop	I_2	nop	I_3	nop	I_4	
EX			I_1	nop	I_2	nop	I_3	nop	I_4
MA				I_1	nop	I_2	nop	I_3	nop I_4
WB					I_1	nop	I_2	nop	I_3 nop I_4

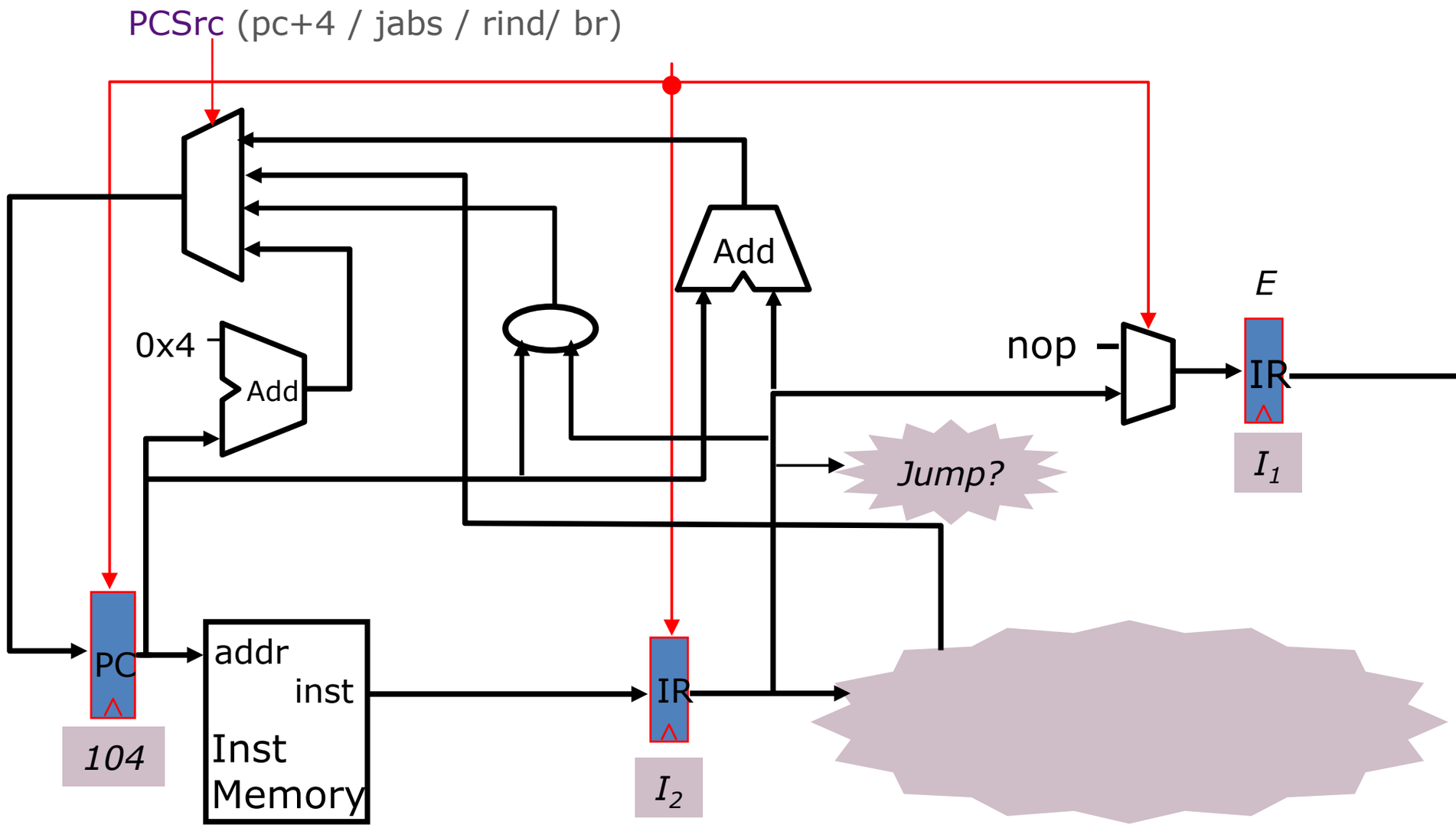
CPI = 2!

nop ⇒ *pipeline bubble*

Speculate next address is PC+4



Speculate next address is PC+4



I_1	096	ADD	
I_2	100	J 304	
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	ADD	

Jump Pipeline Diagrams

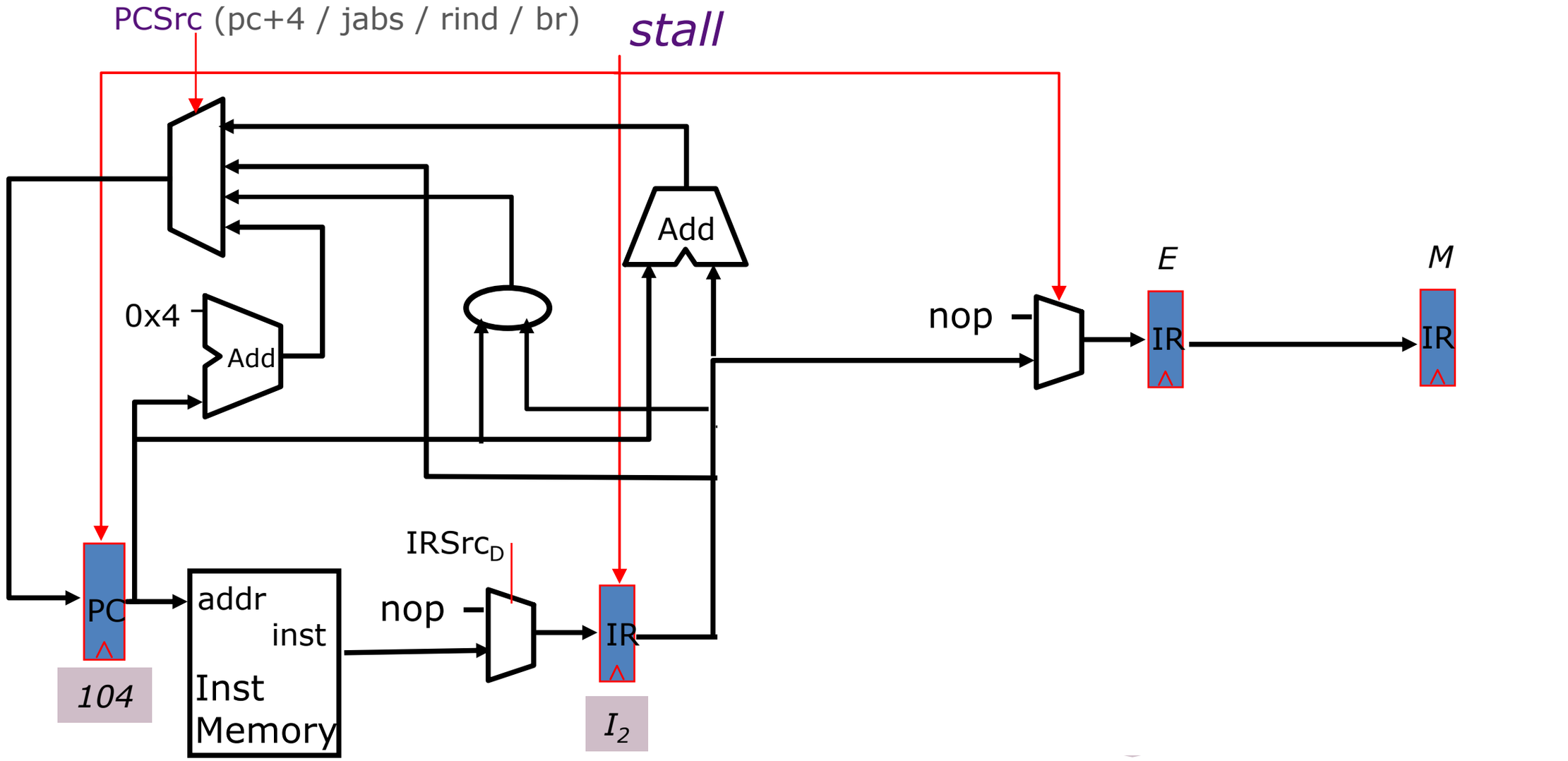
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: J 304		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage

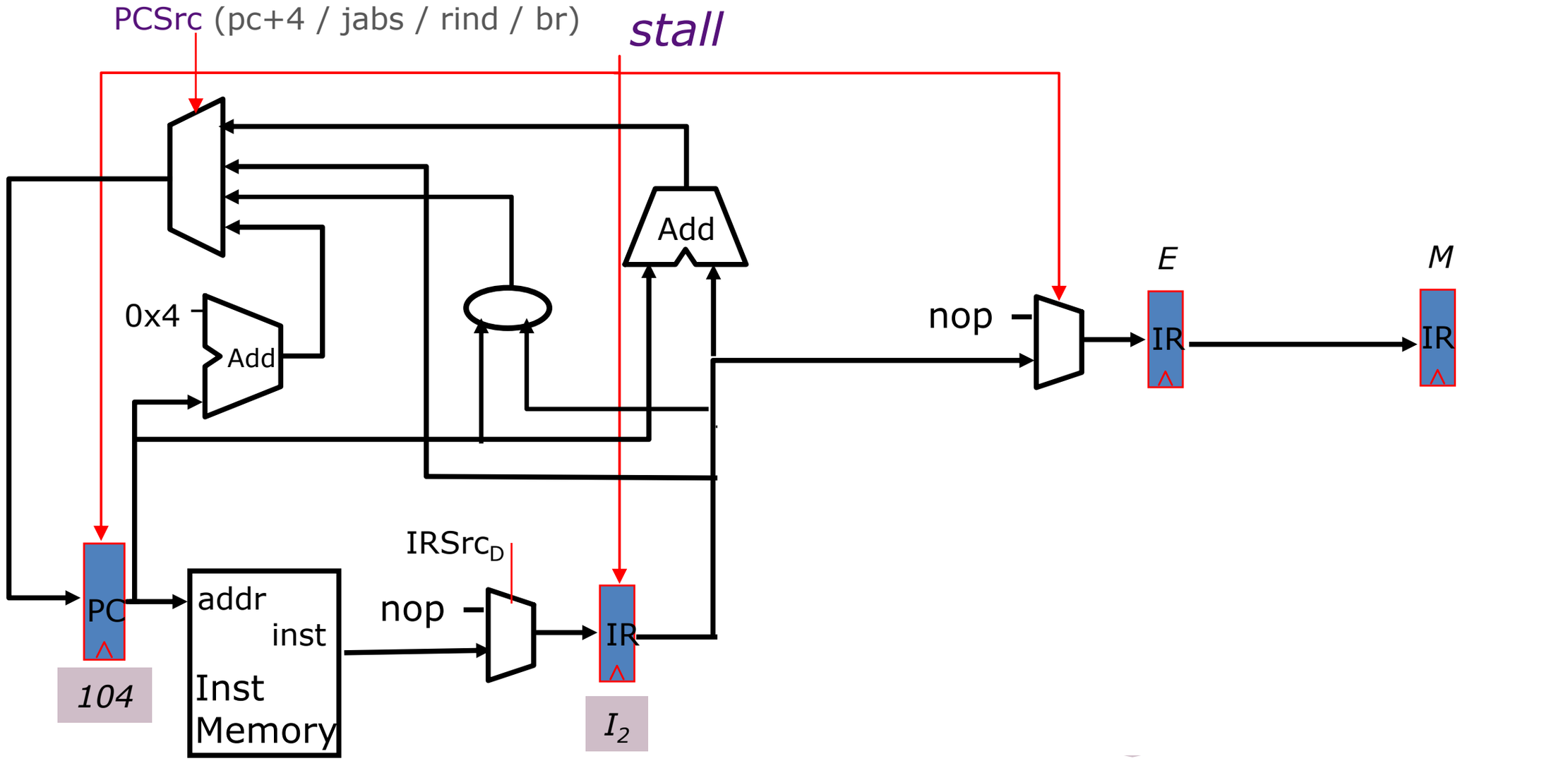
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	nop	I_4	I_5			
EX			I_1	I_2	nop	I_4	I_5		
MA				I_1	I_2	nop	I_4	I_5	
WB					I_1	I_2	nop	I_4	I_5

nop $\Rightarrow$ *pipeline bubble*

Pipelining Conditional Branches



Pipelining Conditional Branches

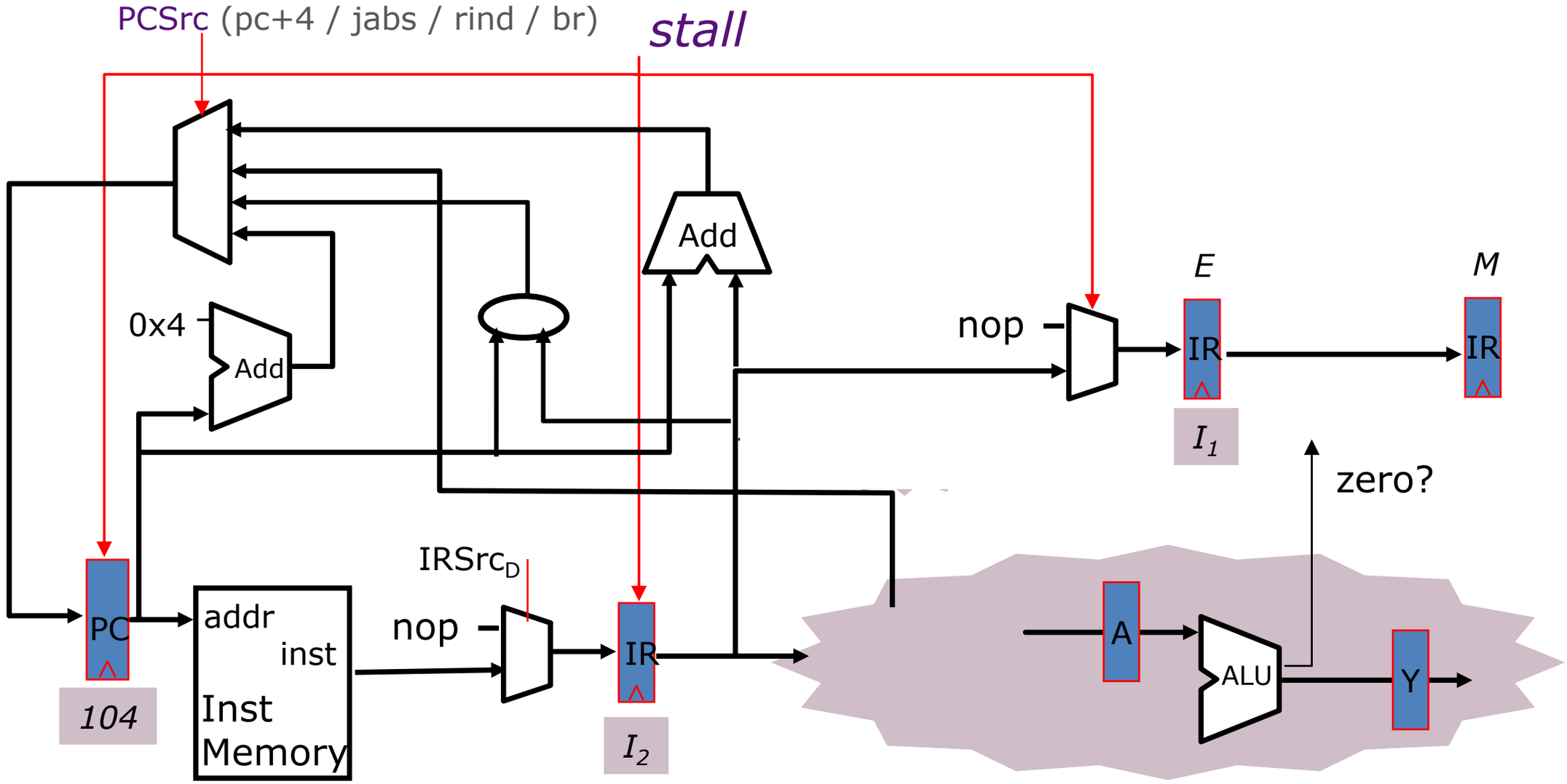


I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	…
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches

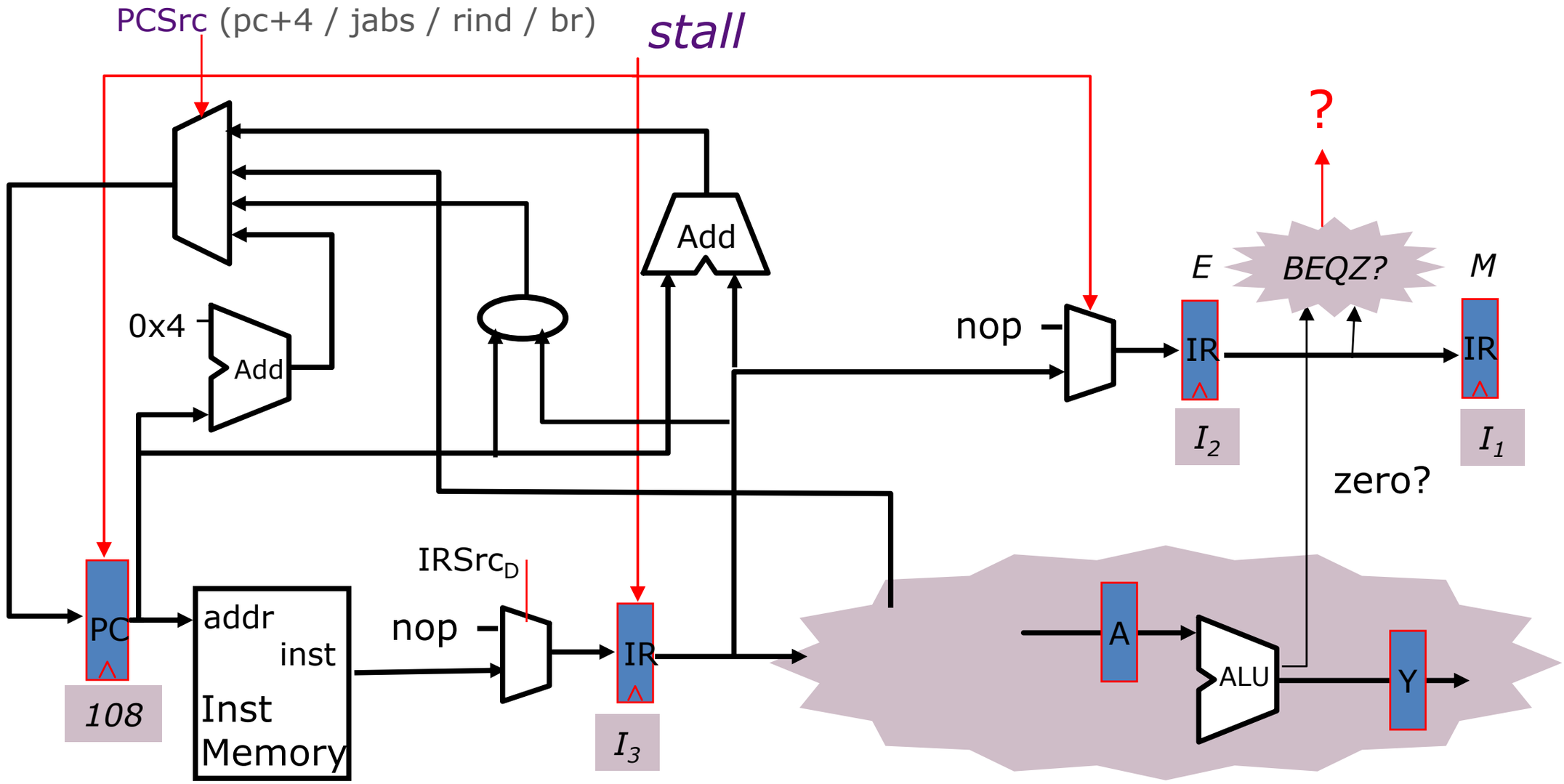


I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	…
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches



I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	…
I_4	304	ADD

If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid

Branch Pipeline Diagrams

(resolved in execute stage)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	nop	nop	nop		
(I_4) 108:				IF_4	nop	nop	nop	nop	
(I_5) 304: ADD					IF_5	ID_5	EX_5	MA_5	WB_5

Resource Usage

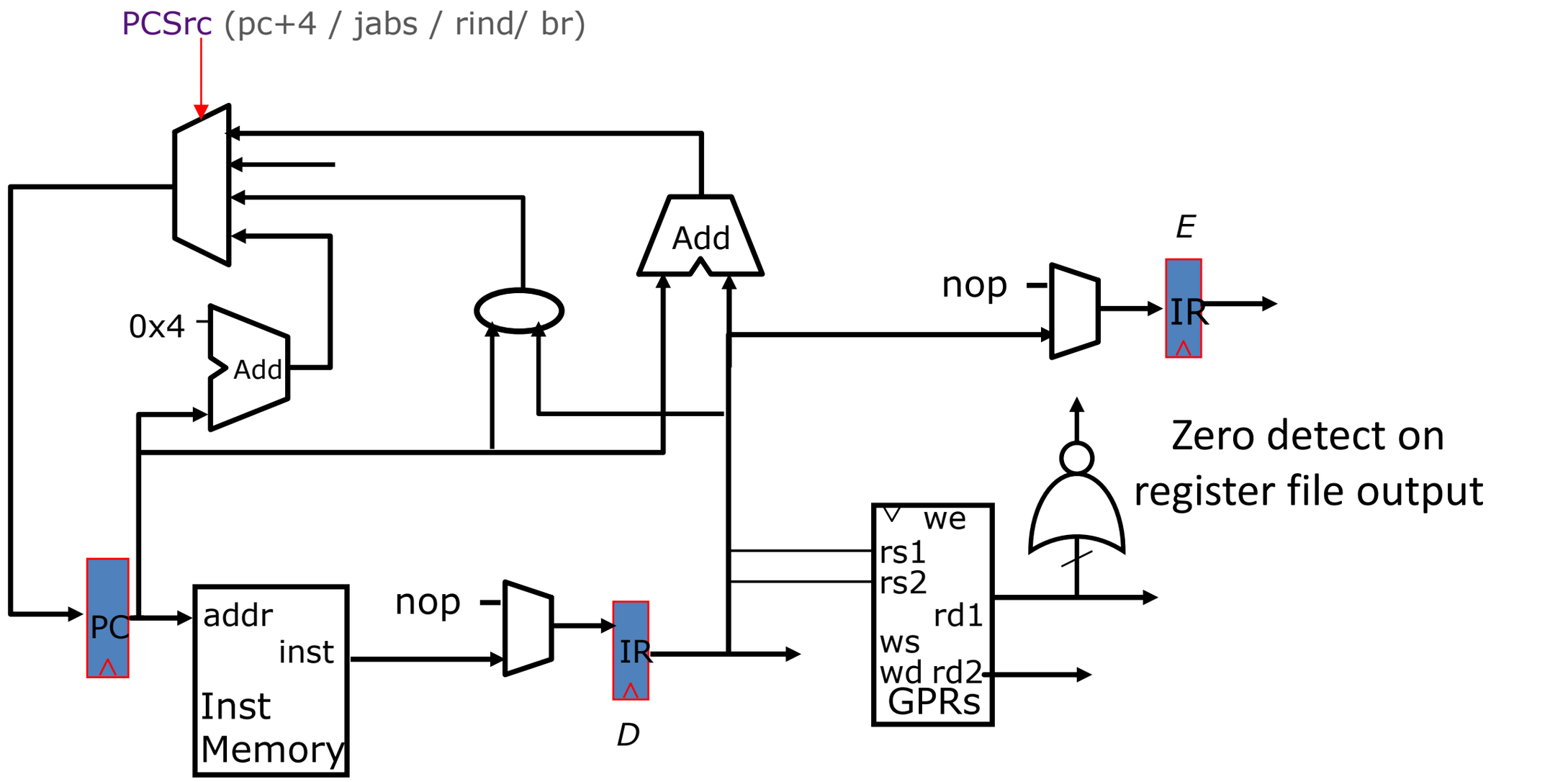
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	I_3	nop	I_5			
EX			I_1	I_2	nop	nop	I_5		
MA				I_1	I_2	nop	nop	I_5	
WB					I_1	I_2	nop	nop	I_5

nop ⇒ *pipeline bubble*

Reducing Branch Penalty

(resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage
  - But might elongate cycle time

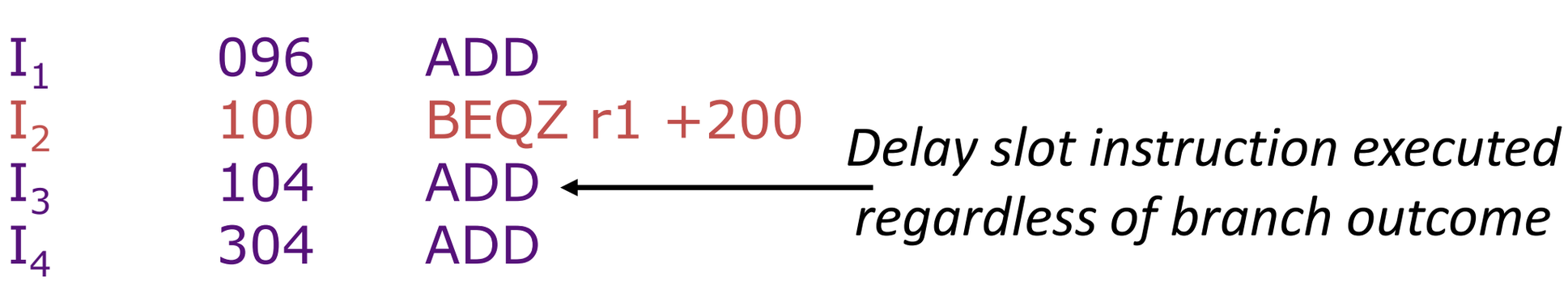


Pipeline diagram now same as for jumps

Branch Delay Slots

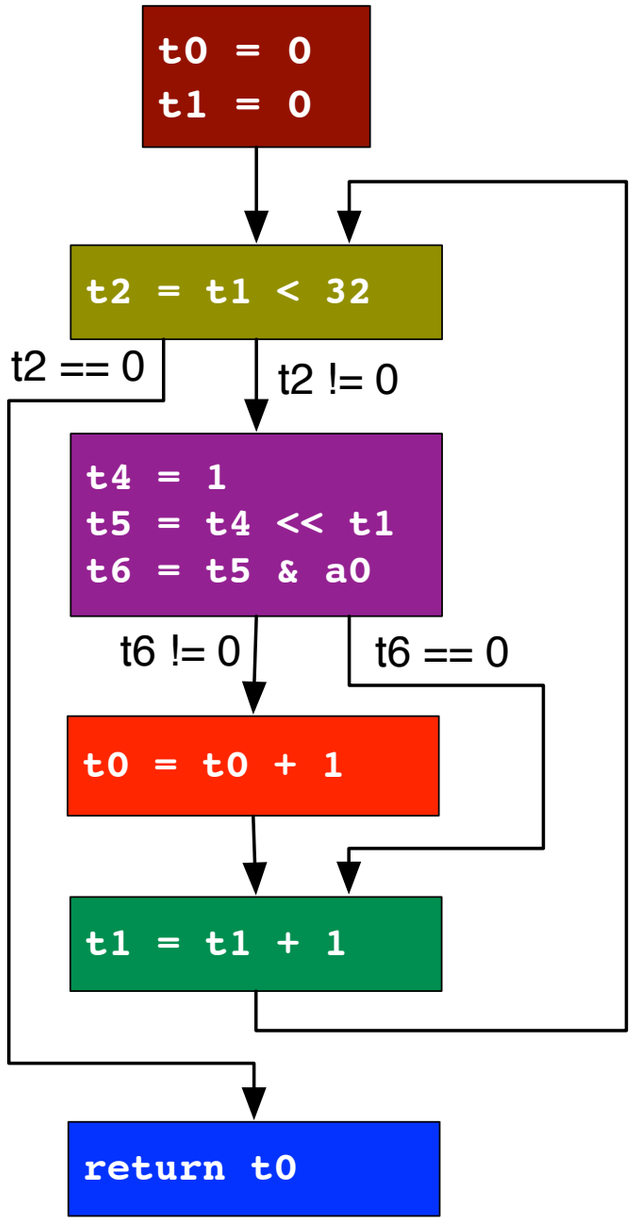
(expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.



- Other techniques include more advanced branch prediction, which can dramatically reduce the branch penalty… *to come later*

In the Compiler



Control flow graph

```
popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
end:
        jr $ra
        nop
```

Assembly

Branch Pipeline Diagrams

(branch delay slot)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4					
ID		I_1	I_2	I_3	I_4				
EX			I_1	I_2	I_3	I_4			
MA				I_1	I_2	I_3	I_4		
WB					I_1	I_2	I_3	I_4	

Scheduling in Action

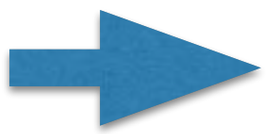
- Suppose every branch brings two stalls, how do you schedule the following instructions?

SUB t2, t3, t5
ADD t4, t3, t5
bneqz t4, 400
ADD t1, t3, t8
ADD t3, t5, t9
beqz t4, 500

Scheduling in Action

- Suppose every branch brings two stalls, how do you schedule the following instructions?

```
SUB t2, t3, t5
ADD t4, t3, t5
bneqz t4, 400
ADD t1, t3, t8
ADD t3, t5, t9
beqz t4, 500
```

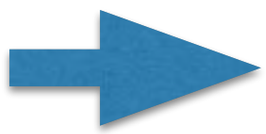


```
ADD t4, t3, t5
bneqz t4, 400
SUB t2, t3, t5
beqz t4, 500
ADD t1, t3, t8
ADD t3, t5, t9
```

Scheduling in Action

- Suppose every branch brings two stalls, how do you schedule the following instructions?

```
SUB t2, t3, t5
ADD t4, t3, t5
bneqz t4, 400
ADD t1, t3, t8
ADD t3, t5, t9
beqz t4, 500
```



```
ADD t4, t3, t5
bneqz t4, 400
SUB t2, t3, t5      <-- stall
beqz t4, 500
ADD t1, t3, t8
ADD t3, t5, t9
```

Exceptions

- Causes
  - Arithmetic overflow
    - In an 8-bit machine, what if you do 255+1?
  - Undefined instruction
  - System call
- When to handle
  - when detected
- Who should handle
  - Process

Interrupts

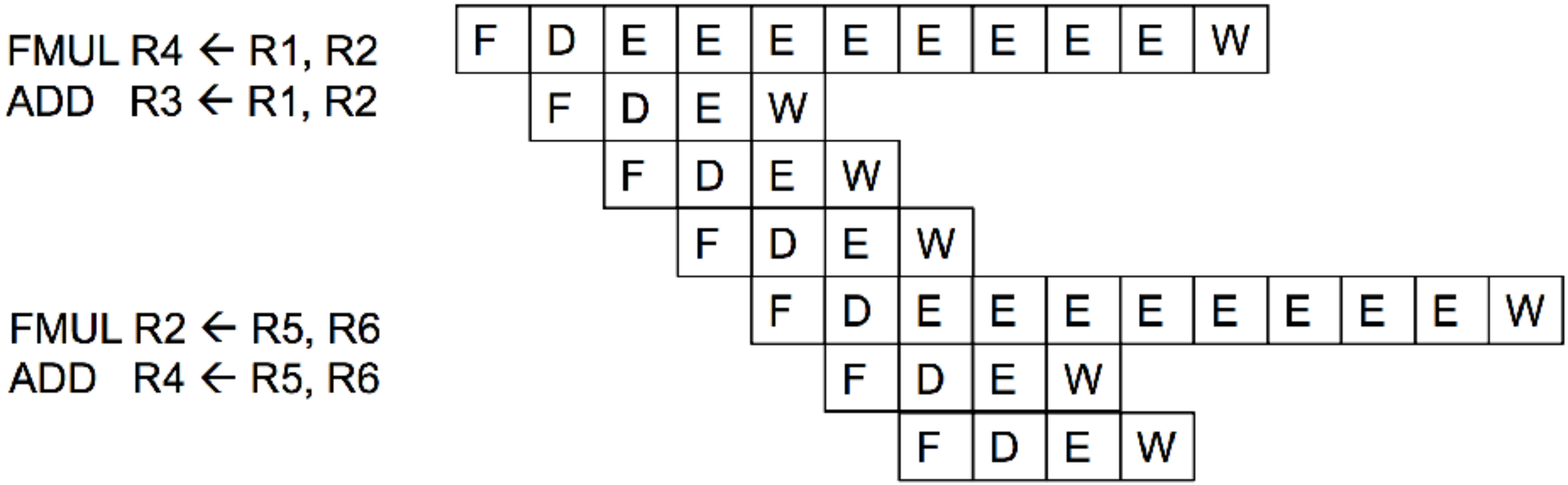
- Causes
  - external events
    - arrival of network package, hard disk, …
- When to handle
  - when convenient except for high priority ones
- Who should handle
  - System

Precise exceptions/interrupts

- The architectural state should be consistent when the exception/interrupt is ready to be handled
  - All previous instructions should be completely retired.
  - No later instruction should be retired.

Retire = commit = finish execution and update arch. state

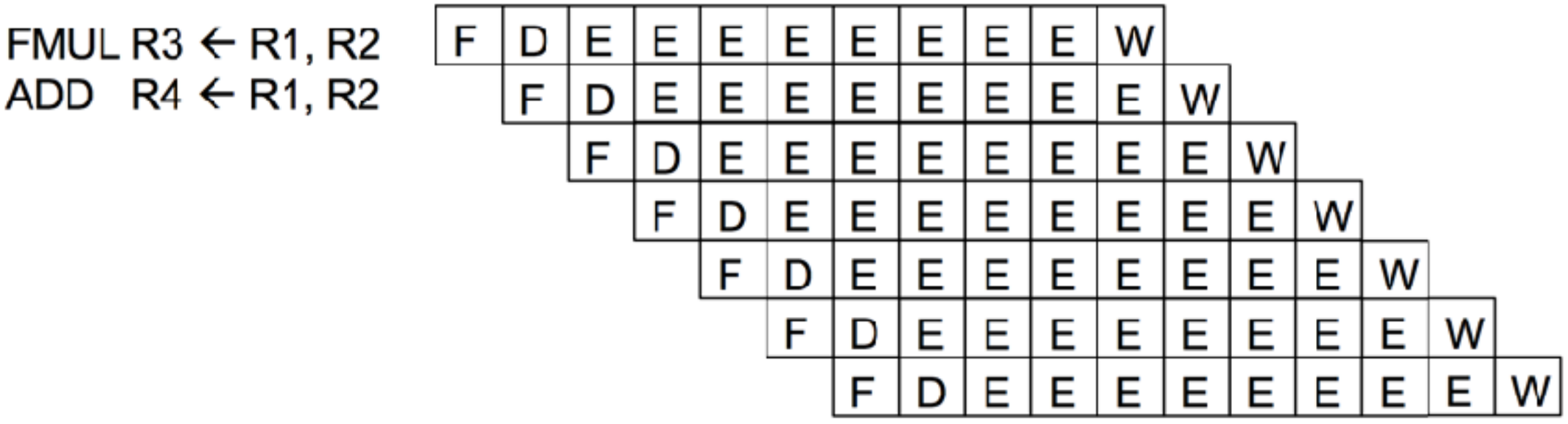
Multi-cycle execute



- instructions may take multiple cycles in ALU
- What's wrong here?
  - what if floating point ALU incurs an exception?

Ensuring precise exceptions in pipelining

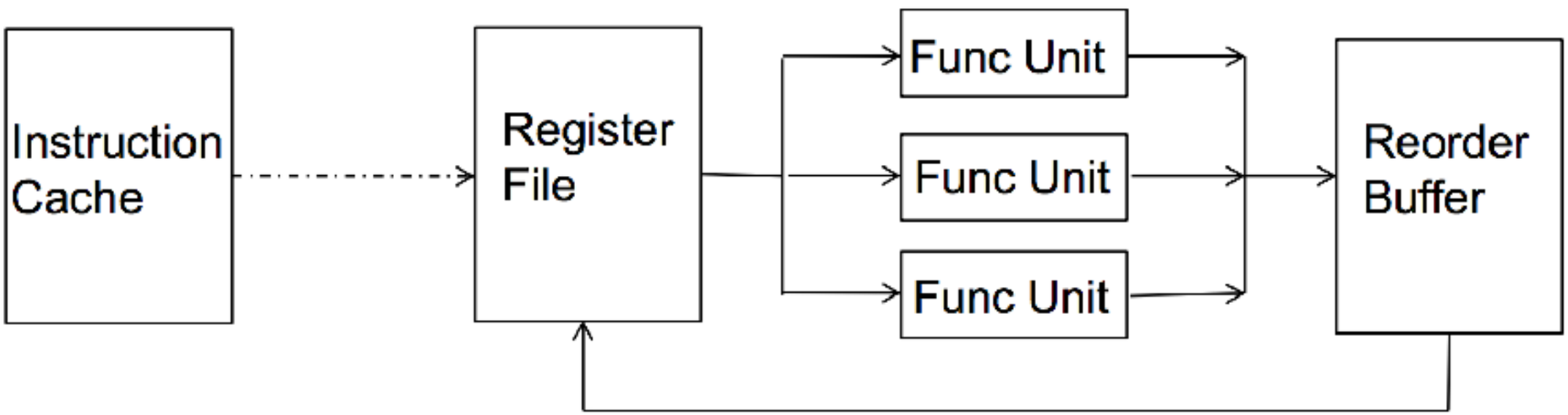
- Idea 1: make each operation take the same amount of time



- Downside
  - worse-case latency determines all instructions' latency
  - chance is high for structural hazards

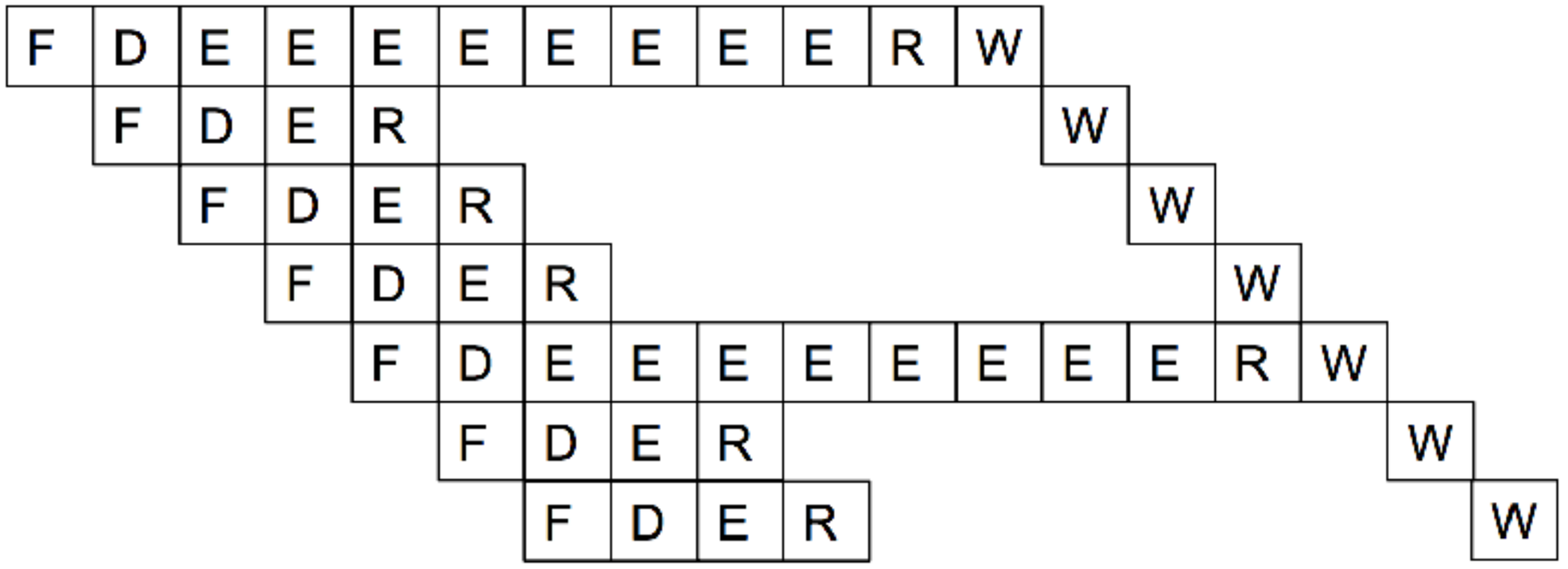
Ensuring precise exceptions in pipelining

- Idea 2: Reorder buffer (ROB)
  - Complete instructions out-of-order, but reorder them before making results visible to architectural state
  - When instruction is decoded it reserves an entry in the ROB
  - When instruction completes, it writes result into ROB entry
  - When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file



Reorder buffer instruction flow

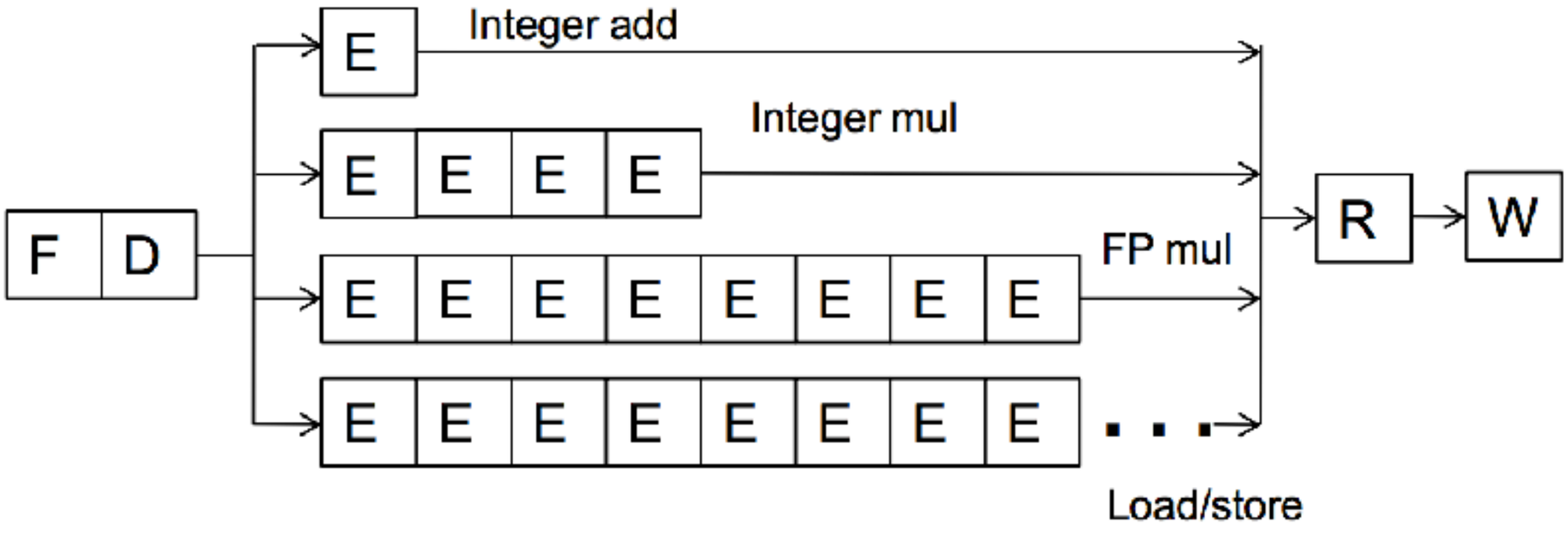
- Results first written to ROB, then to register file at commit time



In-order pipeline with reorder buffer

- Decode (D): Access regfile/ROB, allocate entry in ROB, and dispatch instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler

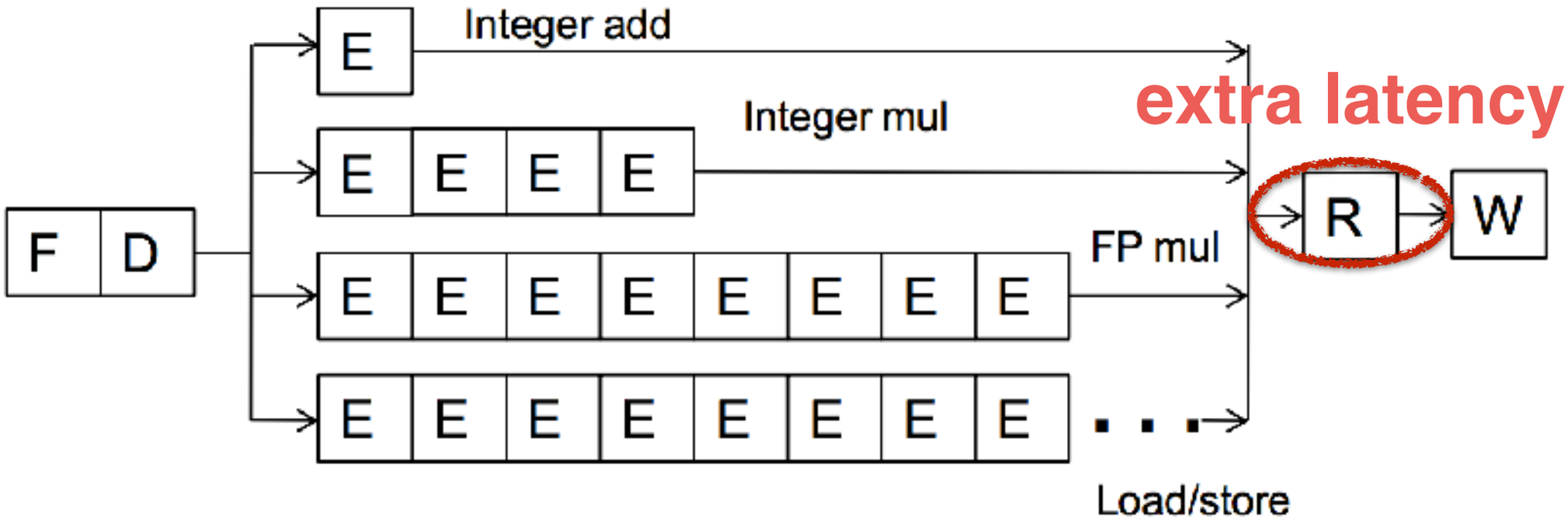
- In-order dispatch/execution, out-of-order completion, in-order retirement



In-order pipeline with reorder buffer

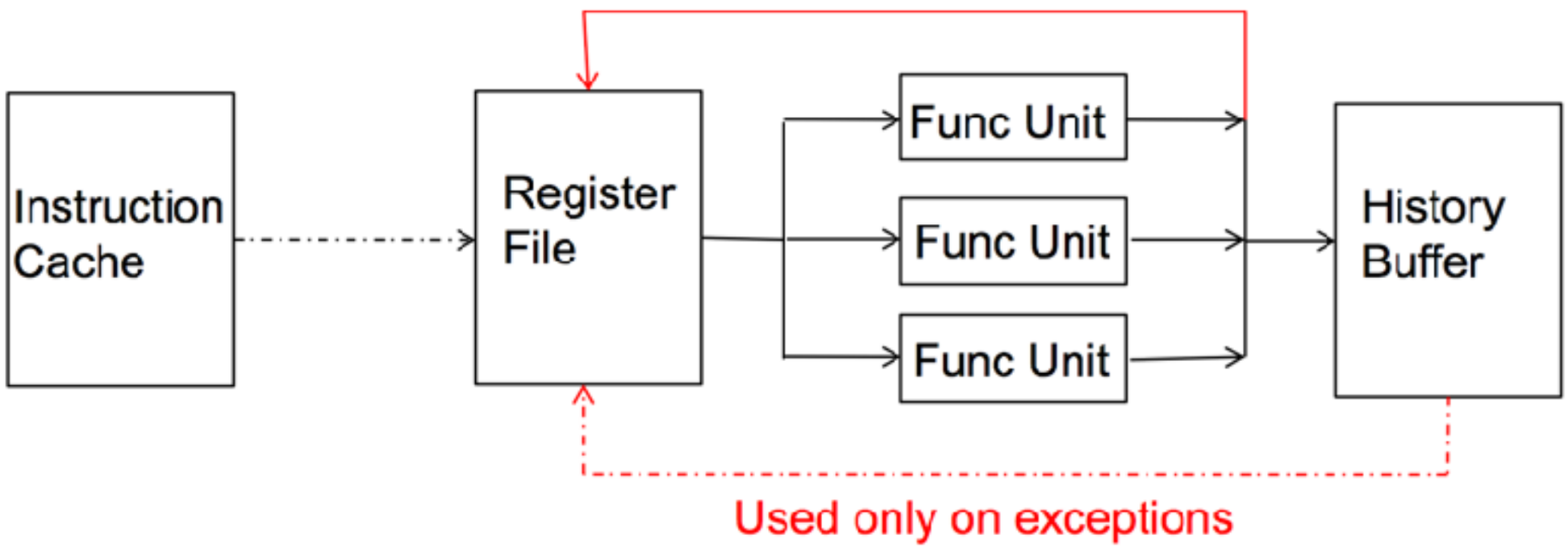
- Decode (D): Access regfile/ROB, allocate entry in ROB, and dispatch instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler

- In-order dispatch/execution, out-of-order completion, in-order retirement



Ensuring precise exceptions in pipelining

- Idea 3: History buffer (HB)
  - When instruction is decoded, it reserves an HB entry
  - When the instruction completes, it stores the old value of its destination in the HB
  - When instruction is oldest and no exceptions interrupts, the HB entry discarded
  - When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head



Superscalar: the essential concept that differentiates undergrad and graduate students

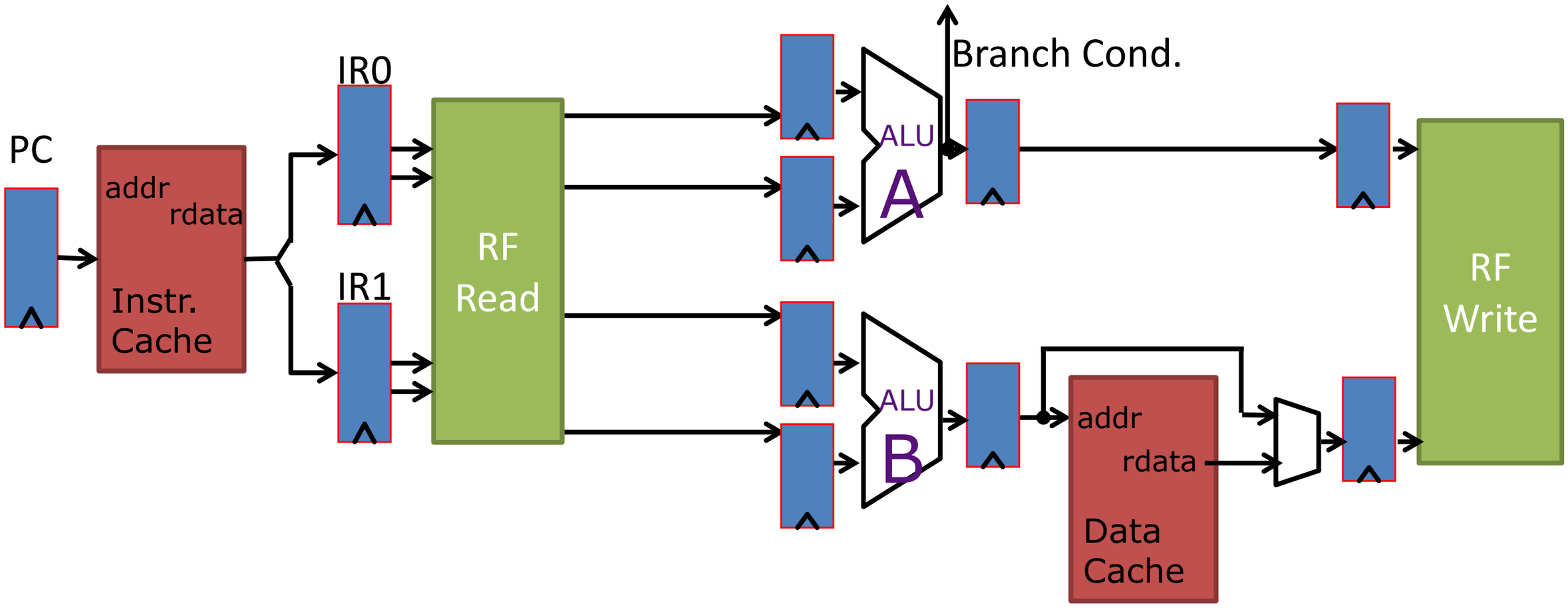
Introduction to Superscalar Processor

- Processors studied so far are fundamentally limited to CPI >= 1

Introduction to Superscalar Processor

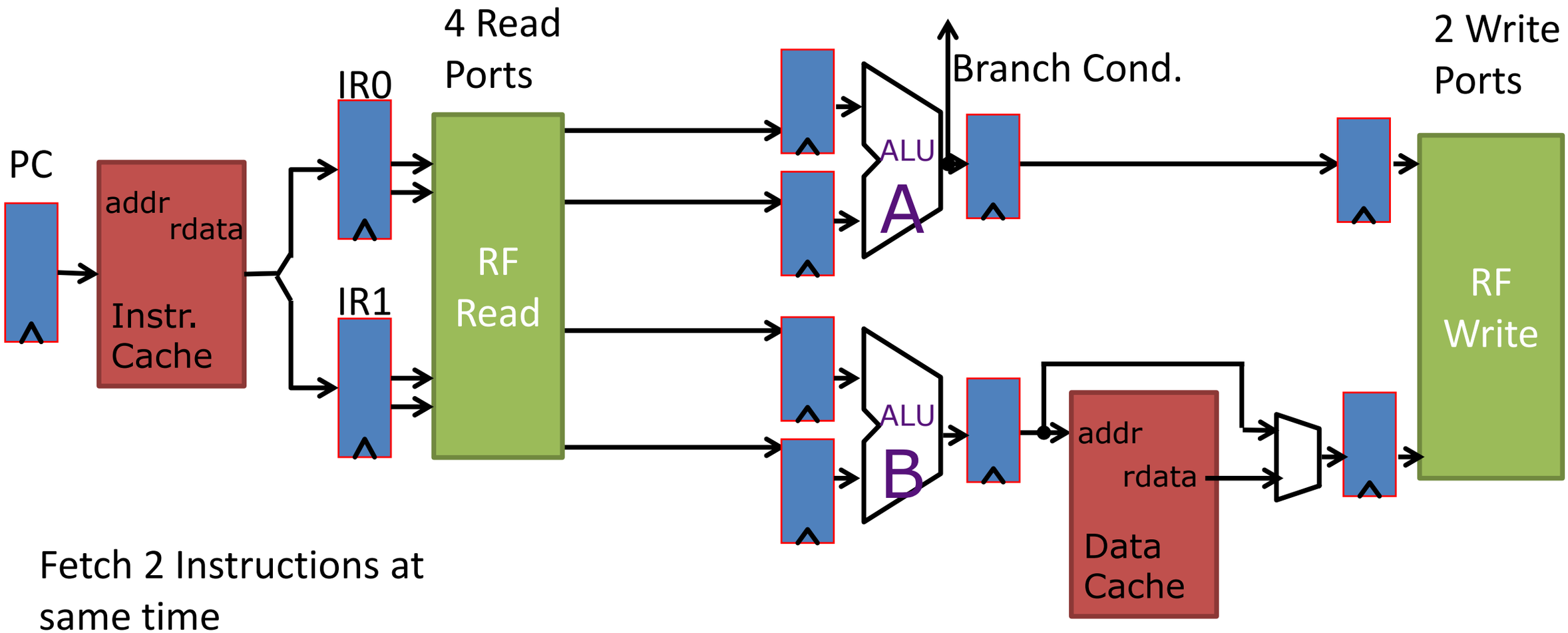
- Processors studied so far are fundamentally limited to CPI >= 1
- Superscalar processors enable CPI < 1 (IPC > 1) by executing multiple instructions in parallel
- Can have both in-order and out-of-order superscalar processors. We will start with in-order.

Baseline 2-Way In-Order Superscalar Processor



Pipe A: Integer Ops., Branches
Pipe B: Integer Ops., Memory

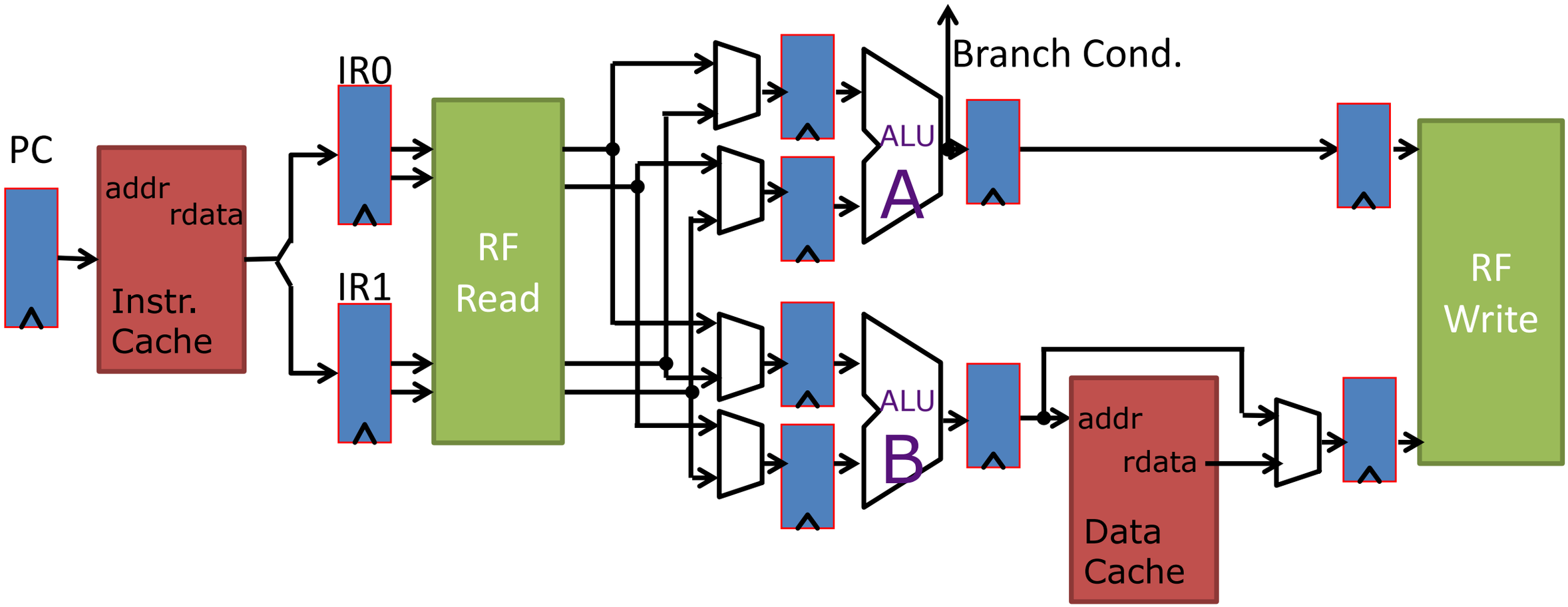
Baseline 2-Way In-Order Superscalar Processor



Fetch 2 Instructions at same time

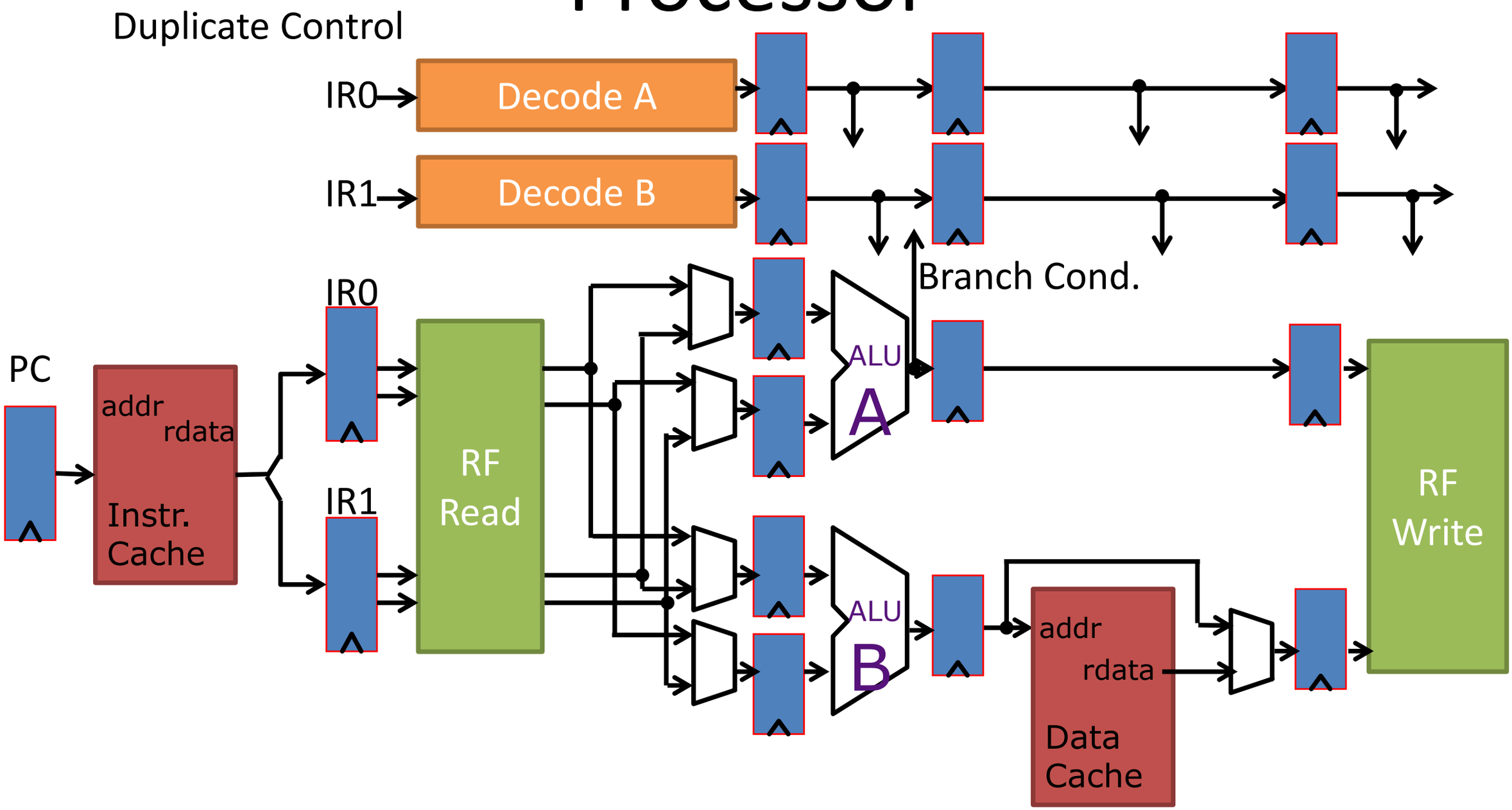
Pipe A: Integer Ops., Branches
Pipe B: Integer Ops., Memory

Baseline 2-Way In-Order Superscalar Processor



Pipe A: Integer Ops., Branches
Pipe B: Integer Ops., Memory

Baseline 2-Way In-Order Superscalar Processor



Pipe A: Integer Ops., Branches
Pipe B: Integer Ops., Memory

Issue Logic Pipeline Diagrams

```
OpA    F   D   A0  A1  W
OpB    F   D   B0  B1  W
OpC        F   D   A0  A1  W
OpD        F   D   B0  B1  W
OpE            F   D   A0  A1  W
OpF            F   D   B0  B1  W
```

CPI = 0.5 (IPC = 2)

Double Issue Pipeline
Can have two instructions in same stage at same time

Dual Issue Data Hazards

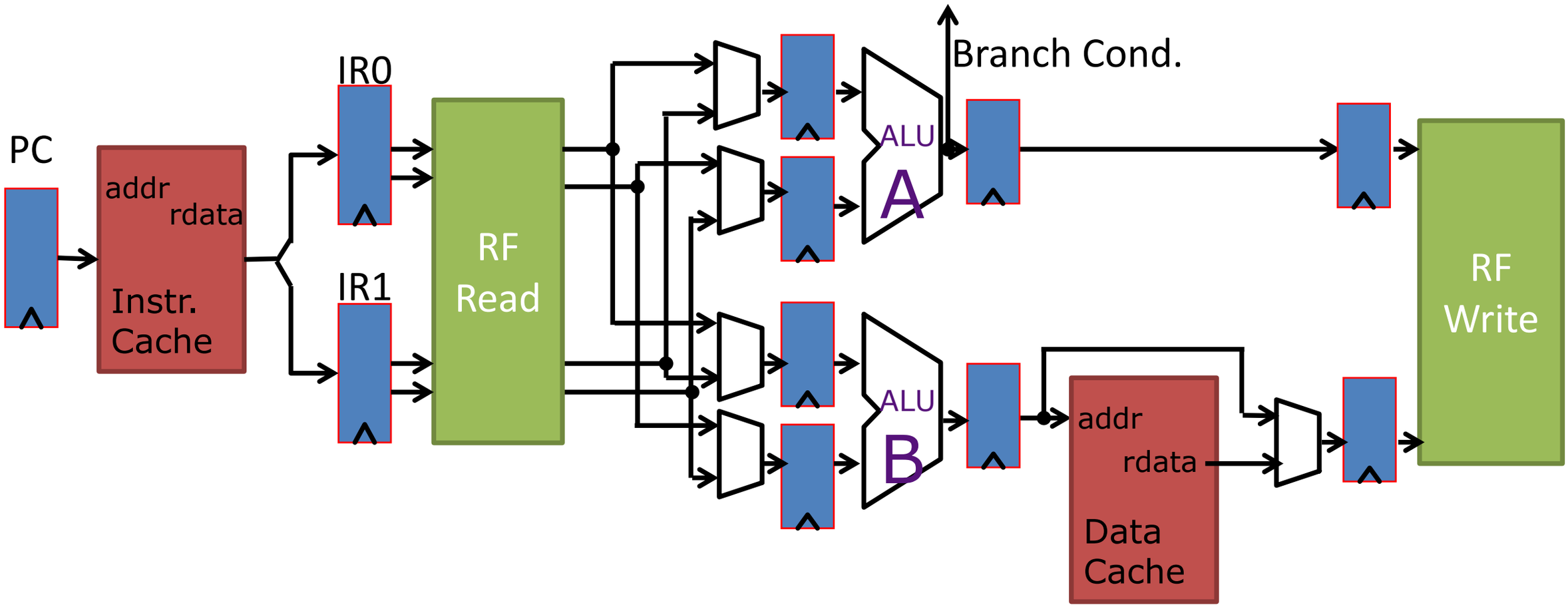
```
No Bypassing:
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R5,R6,1       F   D   A0 A1 W
ADDIU R7,R5,1       F   D   D   D   D   B0 B1 W
```

Dual Issue Data Hazards

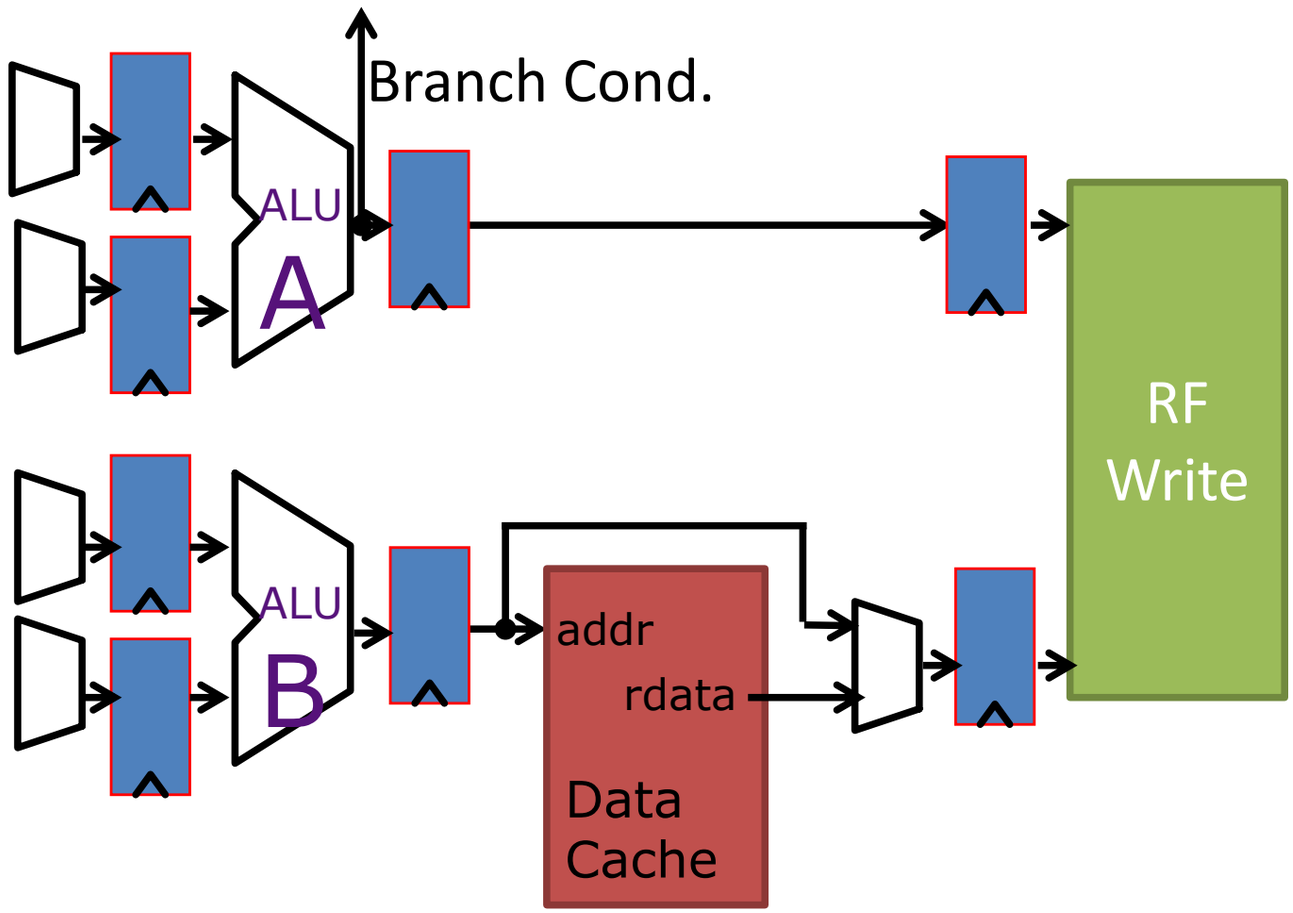
```
No Bypassing:
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R5,R6,1       F   D   A0 A1 W
ADDIU R7,R5,1       F   D   D   D   D   B0 B1 W

Full Bypassing:
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R5,R6,1       F   D   A0 A1 W
ADDIU R7,R5,1       F   D   D   B0 B1 W
```

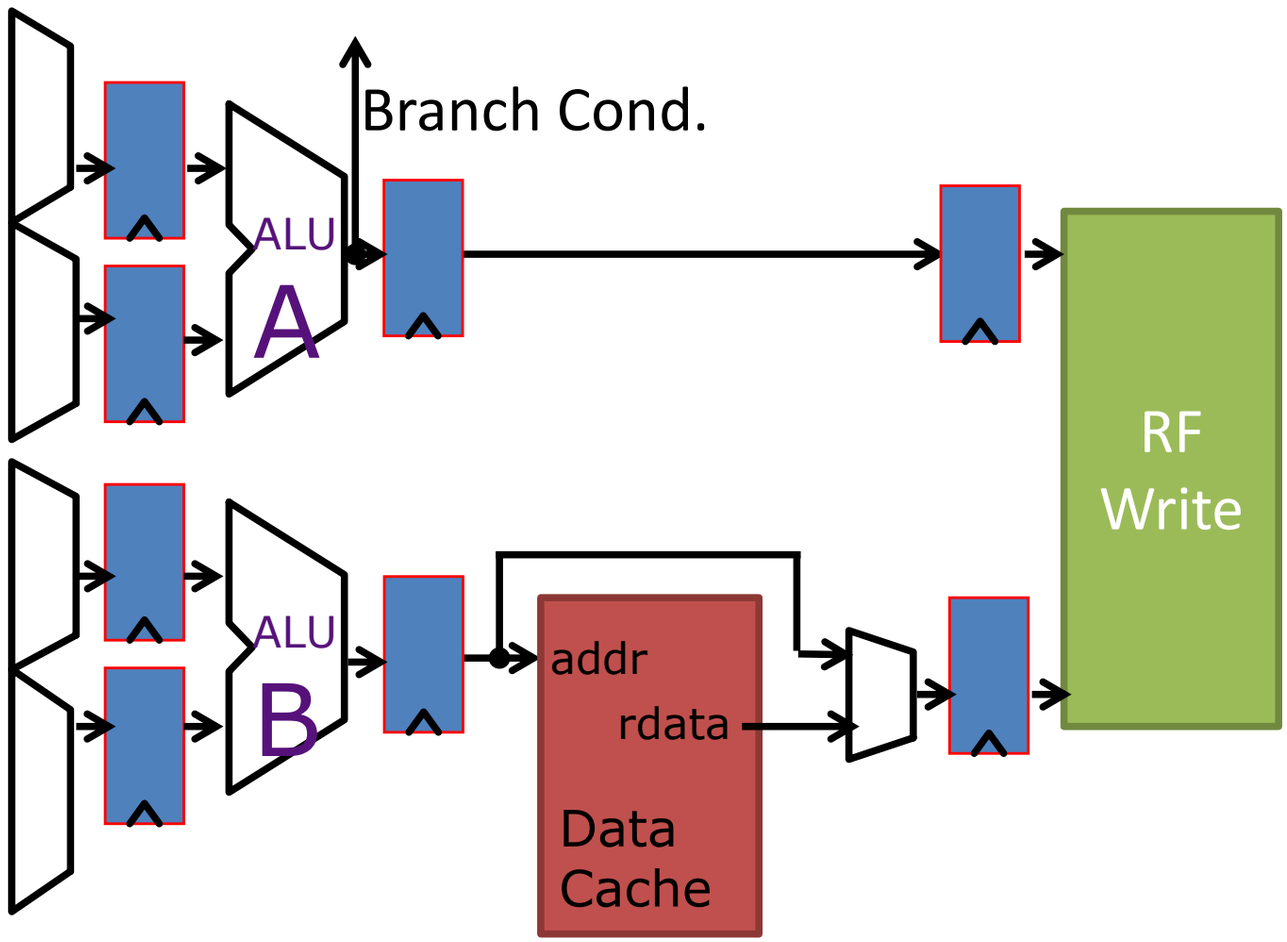
Bypassing in Superscalar Pipelines



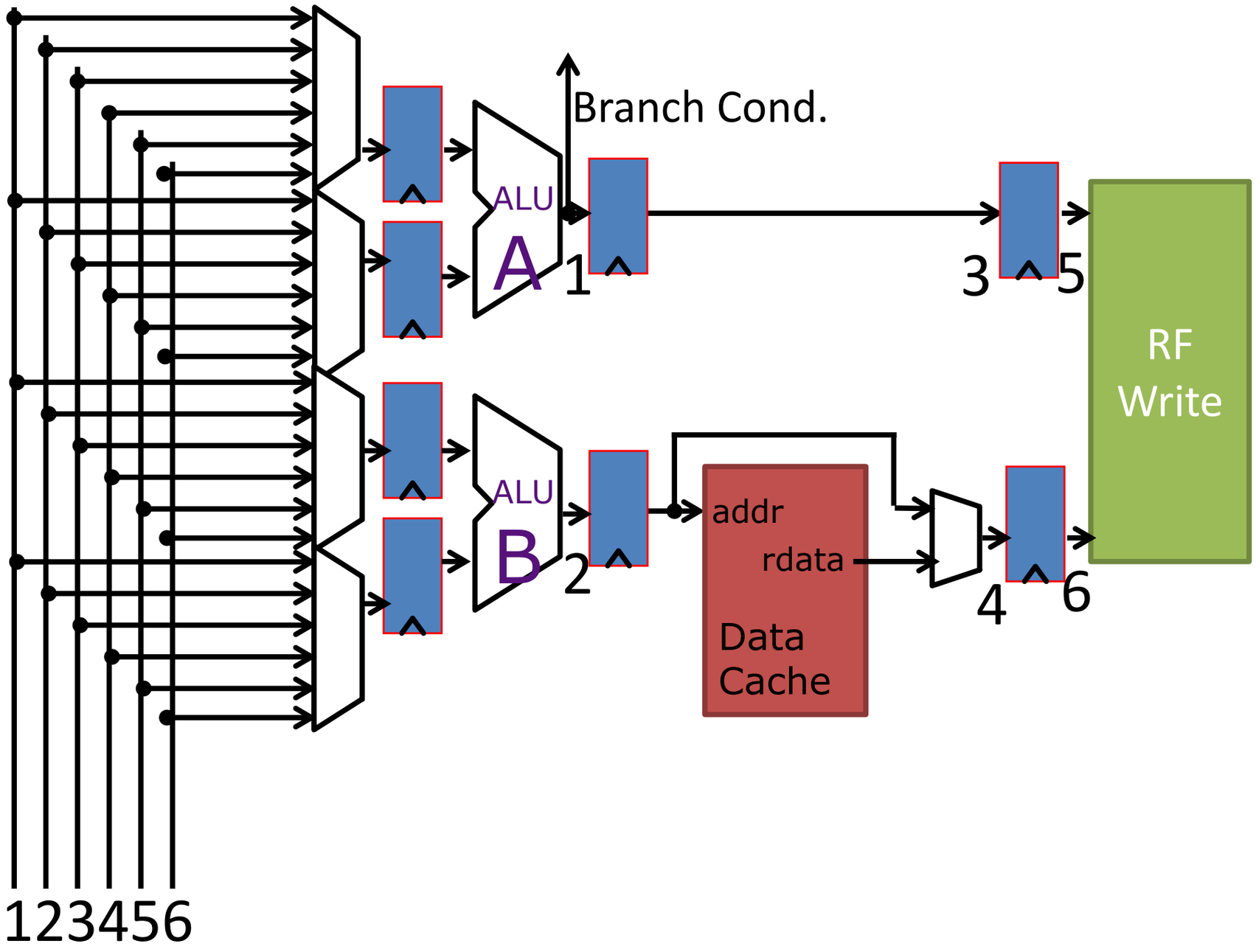
Bypassing in Superscalar Pipelines



Bypassing in Superscalar Pipelines



Bypassing in Superscalar Pipelines



Breaking Decode and Issue Stage

- Bypass Network can become very complex
- Can motivate breaking Decode and Issue Stage

D = Decode, Possibly resolve structural Hazards

I = Register file read, Bypassing, Issue/Steer Instructions to proper unit

```
OpA   F   D   I   A0 A1 W
OpB   F   D   I   B0 B1 W
OpC       F   D   I   A0 A1 W
OpD       F   D   I   B0 B1 W
```

Dual Issue Data Hazards

```
Order Matters:
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R7,R5,1       F   D   A0 A1 W
ADDIU R5,R6,1       F   D   B0 B1 W
```

WAR Hazard Possible?

Fetch Logic and Alignment

```
Cyc Addr Instr
0   0x000 OpA
0   0x004 OpB
1   0x008 OpC
1   0x00C J 0x100
…
2   0x100 OpD
2   0x104 J 0x204
…
3   0x204 OpE
3   0x208 J 0x30C
…
4   0x30C OpF
4   0x310 OpG
5   0x314 OpH
```

0x000	0	0	1	1
…				
0x100	2	2		
…				
0x200		3	3	
…				
0x300				4
0x310	4	5		

Fetching across cache Lines is very hard. May need extra ports

Fetch Logic and Alignment

```
Cyc Addr Instr
0   0x000 OpA
0   0x004 OpB
1   0x008 OpC
1   0x00C J 0x100
…
2   0x100 OpD
2   0x104 J 0x204
…
3   0x204 OpE
3   0x208 J 0x30C
…
4   0x30C OpF
4   0x310 OpG
5   0x314 OpH
```

Ideal, No Alignment Constraints

```
OpA F  D  A0 A1 W
OpB F  D  B0 B1 W
OpC    F  D  B0 B1 W
J      F  D  A0 A1 W
OpD       F  D  B0 B1 W
J         F  D  A0 A1 W
OpE          F  D  B0 B1 W
J            F  D  A0 A1 W
OpF             F  D  A0 A1 W
OpG             F  D  B0 B1 W
OpH                F  D  A0 A1 W
```

With Alignment Constraints

```
Cyc Addr Instr
1   0x000 OpA          F  D  A0 A1 W
1   0x004 OpB          F  D  B0 B1 W
2   0x008 OpC             F  D  B0 B1 W
2   0x00C J 0x100         F  D  A0 A1 W
3   0x100 OpD                F  D  B0 B1 W
3   0x104 J 0x204         F  D  A0 A1 W
4   0x200 ?                  F  -  -  -  -
4   0x204 OpE                F  D  A0 A1 W
5   0x208 J 0x30C               F  D  A0 A1 W
5   0x20C ?                     F  -  -  -  -
6   0x308 ?                        F  -  -  -  -
6   0x30C OpF                      F  D  A0 A1 W
7   0x310 OpG                         F  D  A0 A1 W
7   0x314 OpH                         F  D  B0 B1 W
```

Superscalars Multiply Branch Cost

```
BEQZ   F   D   I   A0 A1 W
OpA    F   D   I   B0 -   -
OpB        F   D   I   -   -   -
OpC        F   D   I   -   -   -
OpD            F   D   -   -   -   -
OpE            F   D   -   -   -   -
OpF                F   -   -   -   -   -
OpG                F   -   -   -   -   -
OpH                    F   D   I   A0 A1 W
OpI                    F   D   I   B0 B1 W
```