Instruction Pipelining: Hazard Resolution, Timing Constraints

Mengjia Yan
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Pipeline Diagram – Ideal Pipelining

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) a1 $\leftarrow$ a0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) a3 $\leftarrow$ a2 + 12		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) a5 $\leftarrow$ a4 + 14			IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4) a7 $\leftarrow$ a6 + 16				IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5) s1 $\leftarrow$ s0 + 18					IF_5	ID_5	EX_5	MA_5	WB_5

Over long term, i.e., in steady state, what are the …

Throughput (T) Latency (L)

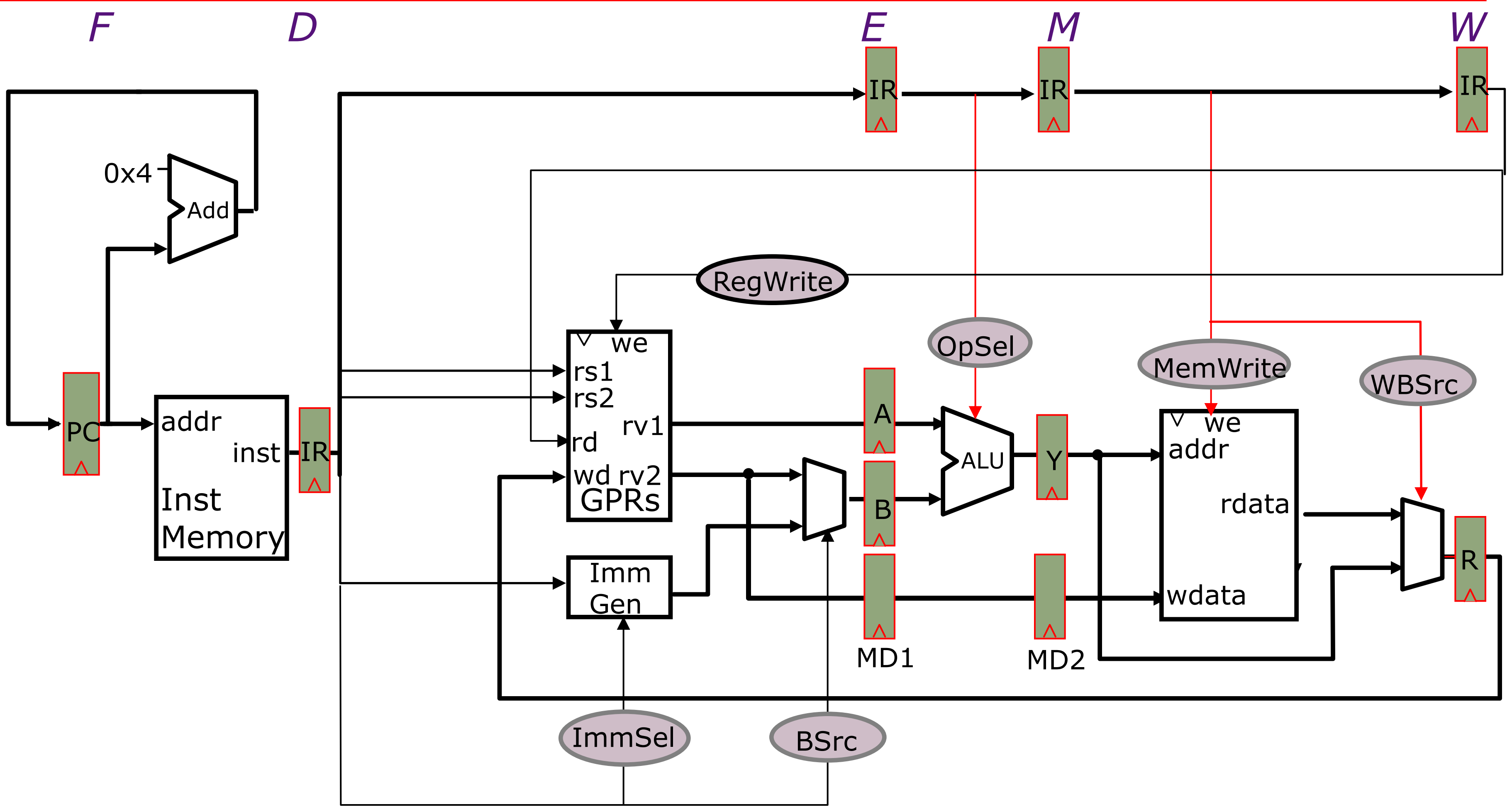
CPI? 1 IPC? 1 Inst exec time? 5

N

Num inst in flight? $\bar{T} = \frac{\bar{N}}{\bar{L}}$ -> 5

Reminder: Pipelined RISC-V Datapath

without jumps



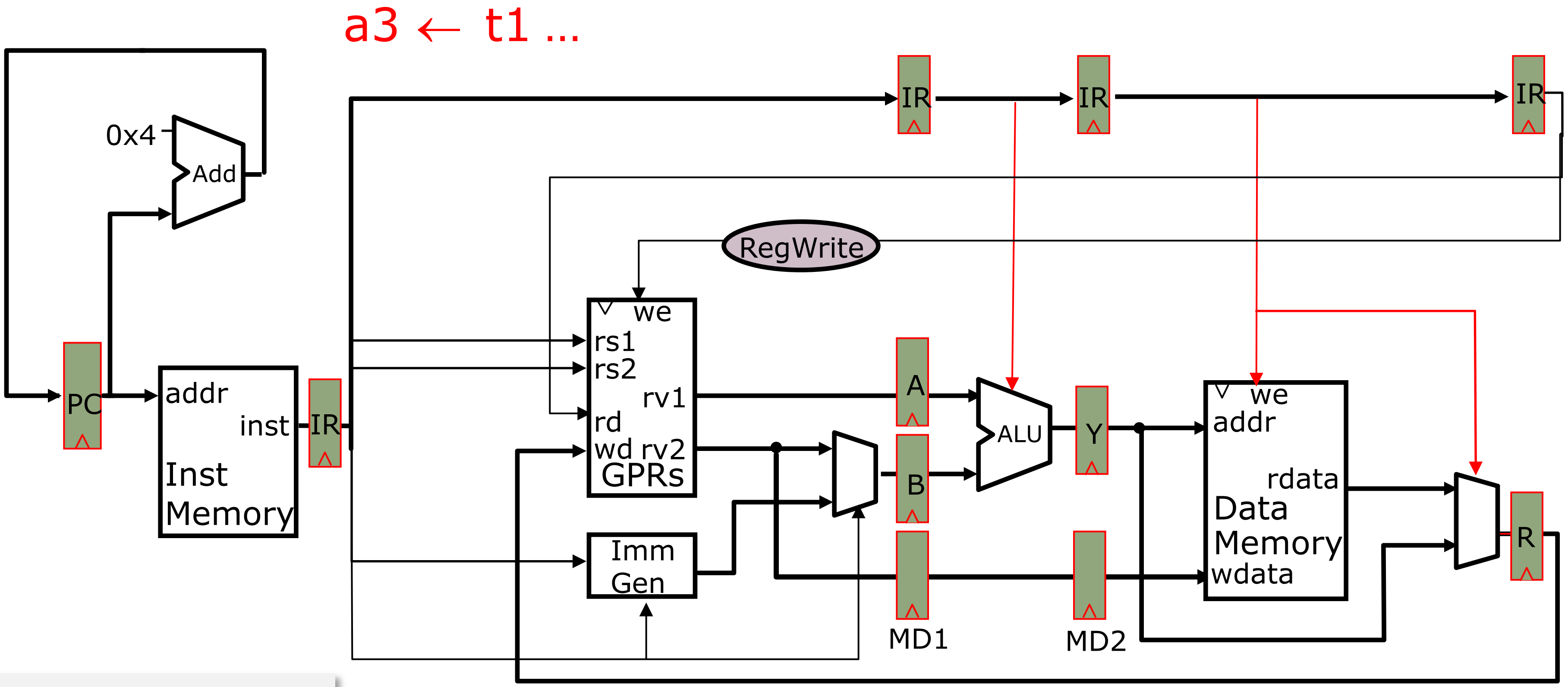
Pipelining increases clock frequency,
but instruction dependences may increase CPI

How instructions can interact with each other in a pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  → *structural hazard*

- An instruction may depend on a value produced by an earlier instruction
  - Dependence may be for a data calculation
    → data hazard
  - Dependence may be for calculating the next PC
    → control hazard (branches, interrupts)

Data Hazards



…
a1 ← a0 + 10
a3 ← a1 + 12
…

Cycle N+1

a1 is stale. Oops!

Pipeline Diagram – Hazard

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) a1 $\leftarrow$ a0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) a3 $\leftarrow$ a1 + 12		IF_2	ID_2	EX_2	MA_2	WB_2			

a1 is stale. Oops!

Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*
Two cases:
  Guessed correctly → no special action required
  Guessed incorrectly → kill and restart

Resolving Data Hazards

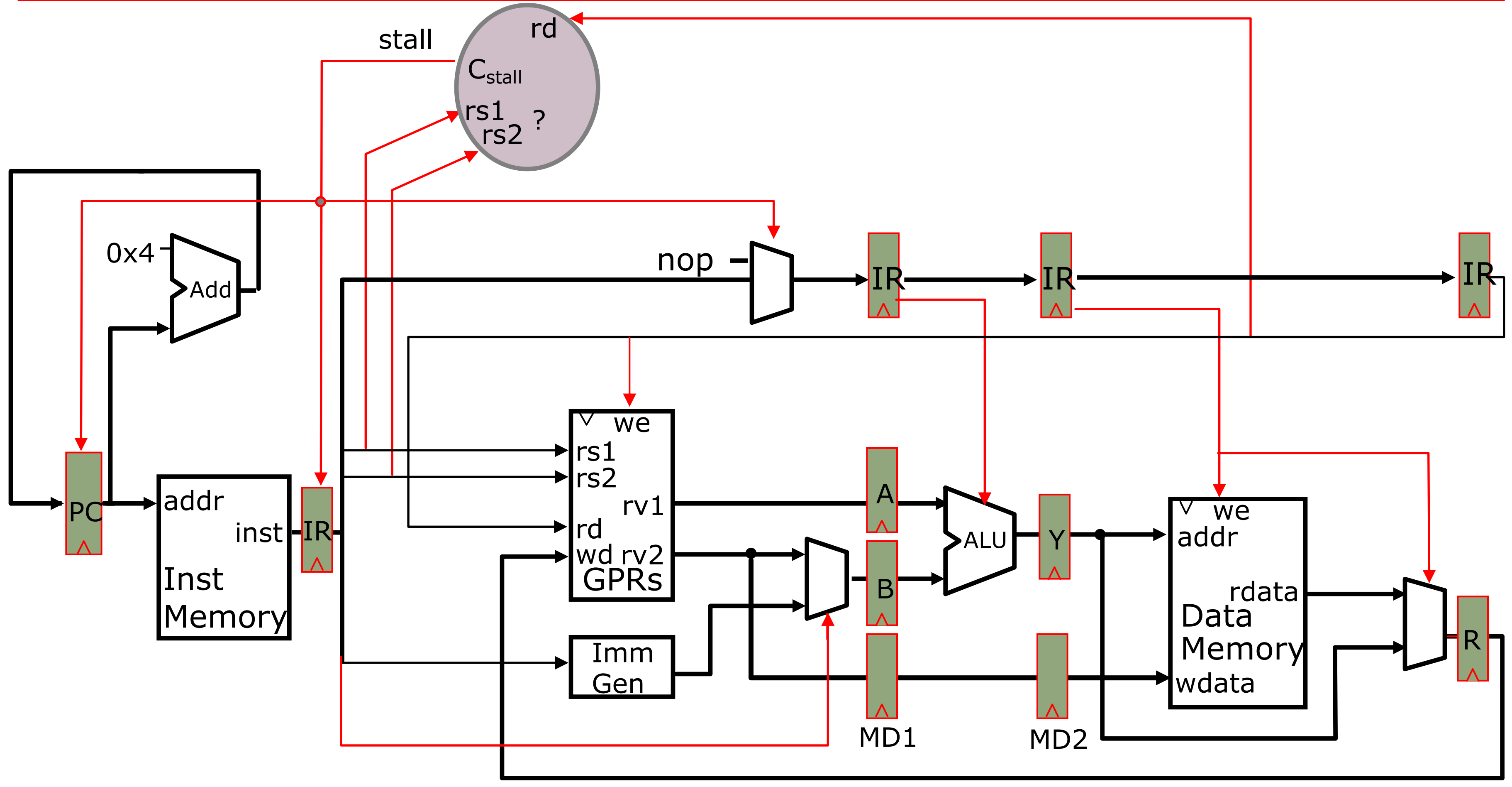
Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) a1 ← a0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) a3 ← a1 + 12		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2

Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)

Reminder: Stall Control Logic

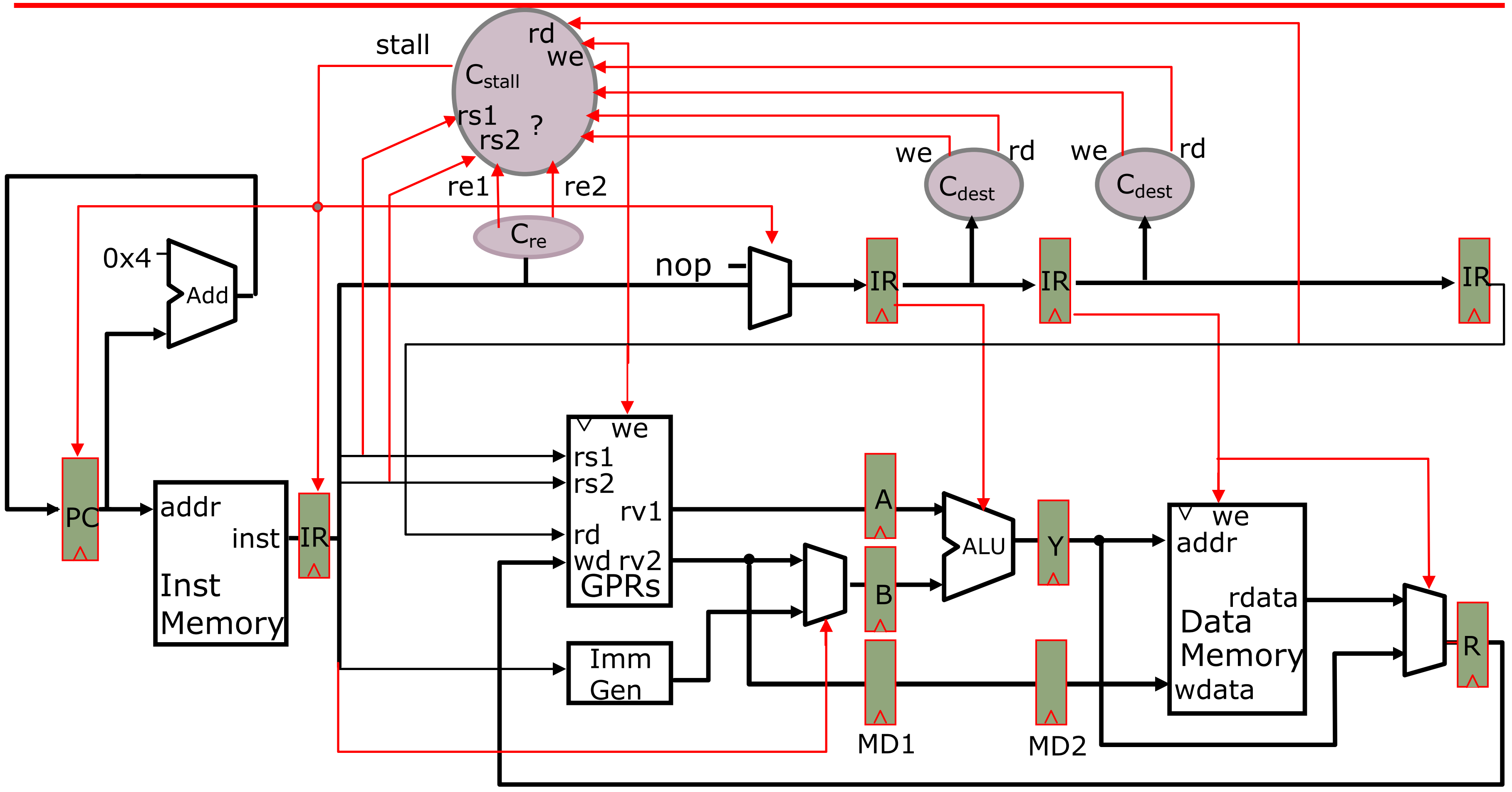
ignoring jumps & branches



Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted* instructions.

Reminder: Stall Control Logic

ignoring jumps & branches



Should we always stall if the rs field(s) matches some rd?

not every instruction writes a register ⇒ we (write enable)

not every instruction reads a register ⇒ re (read enable)

Source & Destination Registers

R-type:	funct7	rs2	rs1	funct3	rd	opcode
I-type:	imm[11:0]		rs1	funct3	rd	opcode
	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

		source(s)	*destination*
ALU	rd ← (rs1) func (rs2)	rs1, rs2	rd
ALUi	rd ← (rs1) funct SXT(imm)	rs1	rd
LW	rd ← M [(rs1) + imm.disp]	rs1	rd
SW	M [(rs1) + imm.disp] ← (rs2)	rs1, rs2	
BEQ	*cond* (rs1)==(rs2)		
	true: PC ← (PC) + SXT(imm)	rs1, rs2	
	false: PC ← (PC) + 4	rs1, rs2	
JAL	rd ← (PC+4), PC ← (PC) + imm		rd
JALR	rd ← (PC+4), PC ← (rs1) + imm*	rs1	rd

**More precise: pc ← {(reg[rs1] + imm)[31:1], 1'b0}*

Deriving the Stall Signal

C_{dest}

```
we = Case opcode
        ALU, ALUi, LW, JAL, JALR
                          ⇒(rd ≠ 0)
        ...               ⇒ off
```

C_{re}

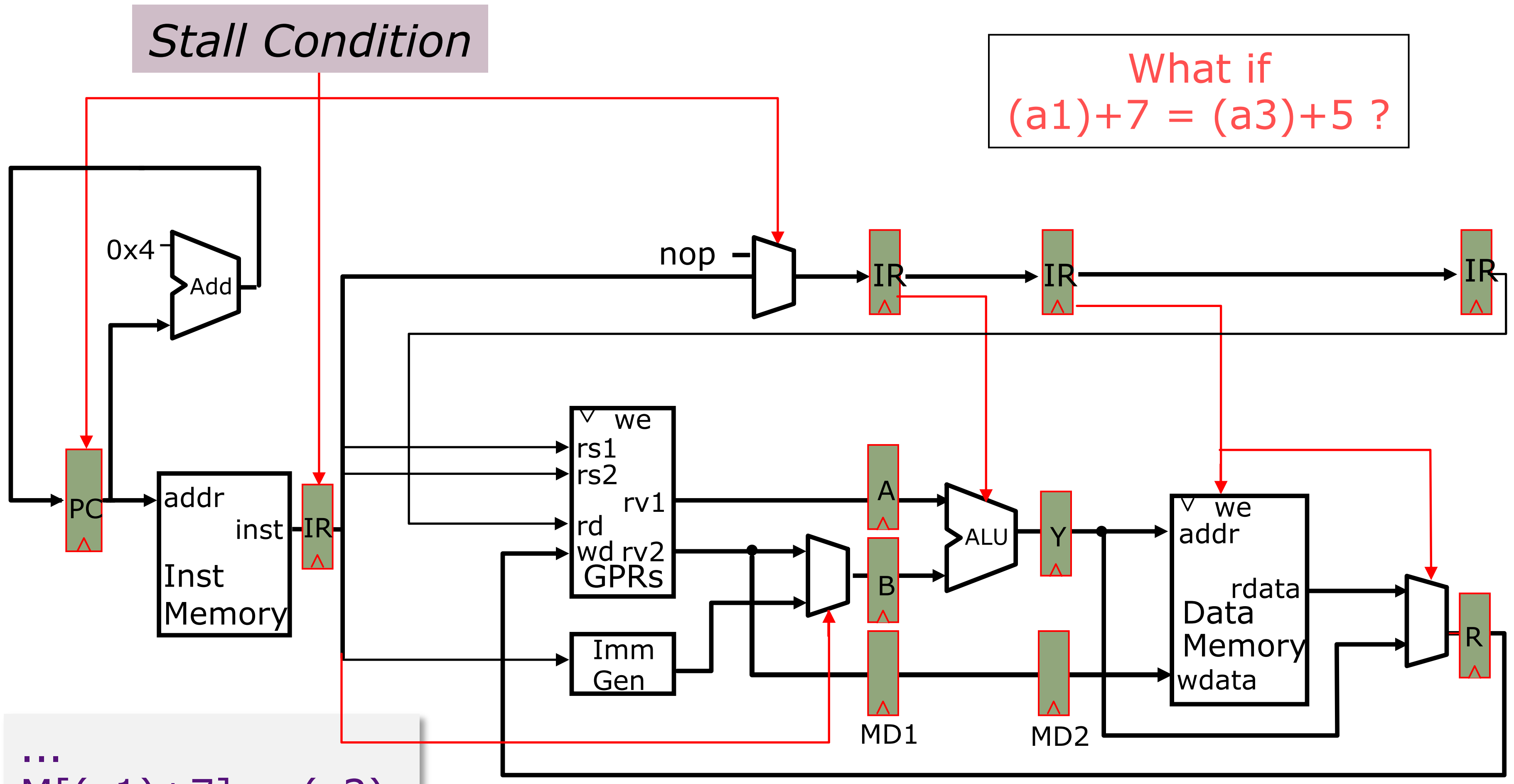
```
re1 = Case opcode
        ALU, ALUi,
        LW, SW, BEQ,
        JR, JALR          ⇒ on

re2 = Case opcode
        ALU, SW, BEQ      ⇒ on
        ...               ⇒ off
```

C_{stall}

$$\begin{aligned} stall = &((rs1_D == rd_E) \cdot we_E + \\ &(rs1_D == rd_M) \cdot we_M + \\ &(rs1_D == rd_W) \cdot we_W) \cdot re1_D \\ &+ \\ &((rs2_D == rd_E) \cdot we_E + \\ &(rs2_D == rd_M) \cdot we_M + \\ &(rs2_D == rd_W) \cdot we_W) \cdot re2_D \end{aligned}$$

This is not the full story !

Hazards due to Loads & Stores



...
M[(a1)+7] ← (a2)
a4 ← M[(a3)+5]
...

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

```
...
M[(a1)+7] ← (a2)
t4 ← M[(a3)+5]
...
```

$\boxed{(a1)+7 = (a3)+5 \;\Rightarrow\; \textit{data hazard}}$

However, the hazard is avoided because *our memory system completes writes in one cycle!*

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

Bypassing

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) a1 ← a0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) a4 ← a1 + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3
(I_4)			*stalled stages*				IF_4	ID_4	EX_4

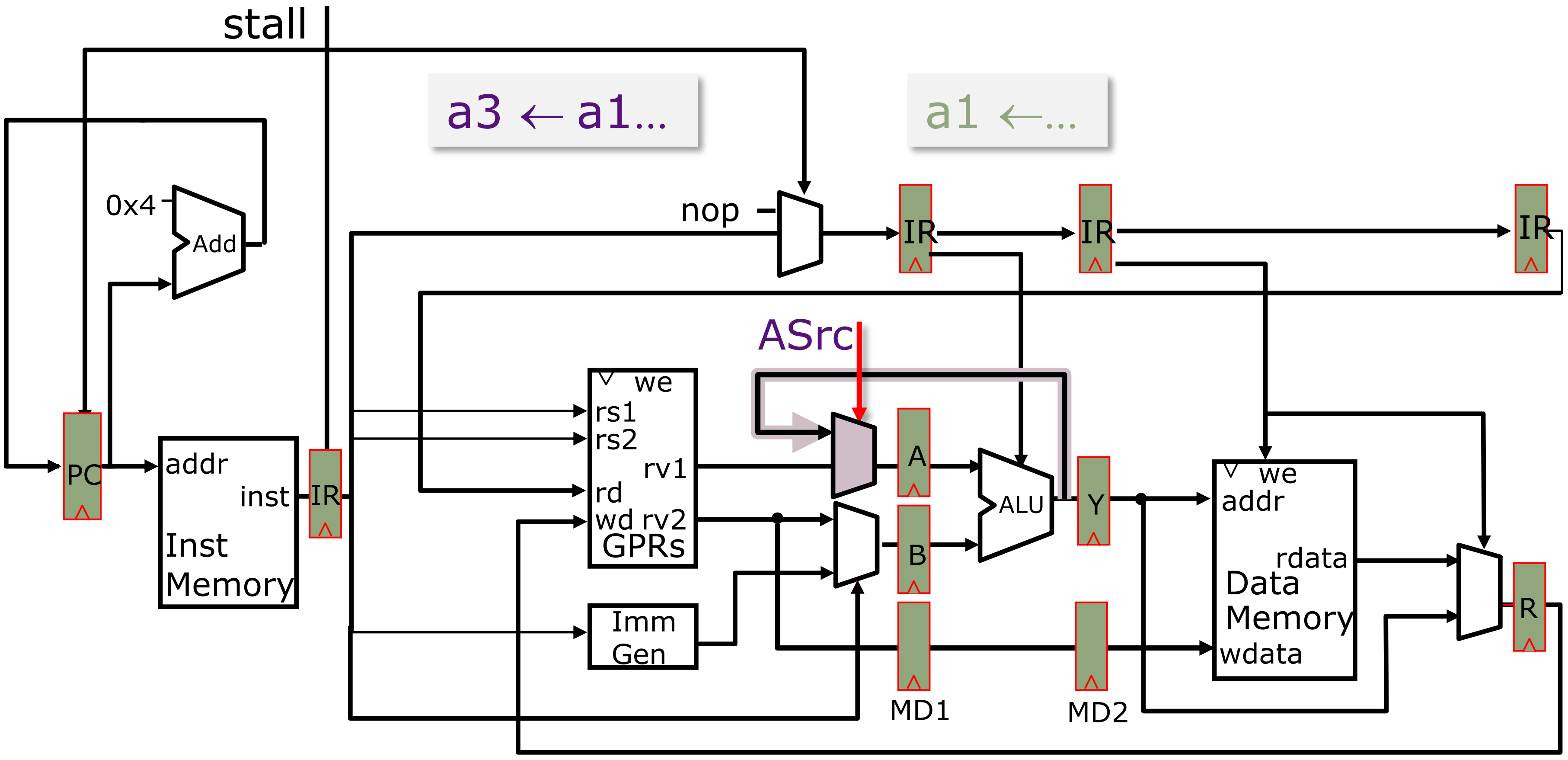
Each *stall or kill* introduces a bubble ⇒ *CPI > 1*

When is data actually available? At Execute

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) a1 ← a0 + 10		IF_1	ID_1	EX_1	MA_1	WB_1			
(I_2) a4 ← a1 + 17			IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)					IF_4	ID_4	EX_4	MA_4	WB_4

A new datapath, i.e., *a bypass*, can get the data from the output of the ALU to its input

Adding a Bypass



When does this bypass help?

(I_1)	a1 ← a0 + 10
(I_2)	a3 ← a1 + 12

yes

a1 ← M[a0 + 10]
a3 ← a1 + 12

no

JAL ra 500
t3 ← ra + 12

no

The Bypass Signal

Deriving it from the Stall Signal

stall = ~~$((rs1_D==rd_E)\cdot we_E$~~ $+ (rs1_D==rd_M)\cdot we_M + (rs1_D==rd_W)\cdot we_W)\cdot re1_D$

$+((rs2_D==rd_E)\cdot we_E + (rs2_D==rd_M)\cdot we_M + (rs2_D==rd_W)\cdot we_W)\cdot re2_D$

```
we = Case opcode
     ALU, ALUi, LW, JAL, JALR
                        ⇒(rd ≠ 0)
     ...                ⇒ off
```

$\boxed{ASrc = (rs1_D==rd_E)\cdot we_E\cdot re1_D}$

Is this correct?

No, only ALU and ALUi instructions can benefit from this bypass. Memory and JAL* instructions cannot.

How might we address this?

Split we_E into two components: we-bypass, we-stall

Bypass and Stall Signals

Split we_E into two components: we-bypass, we-stall

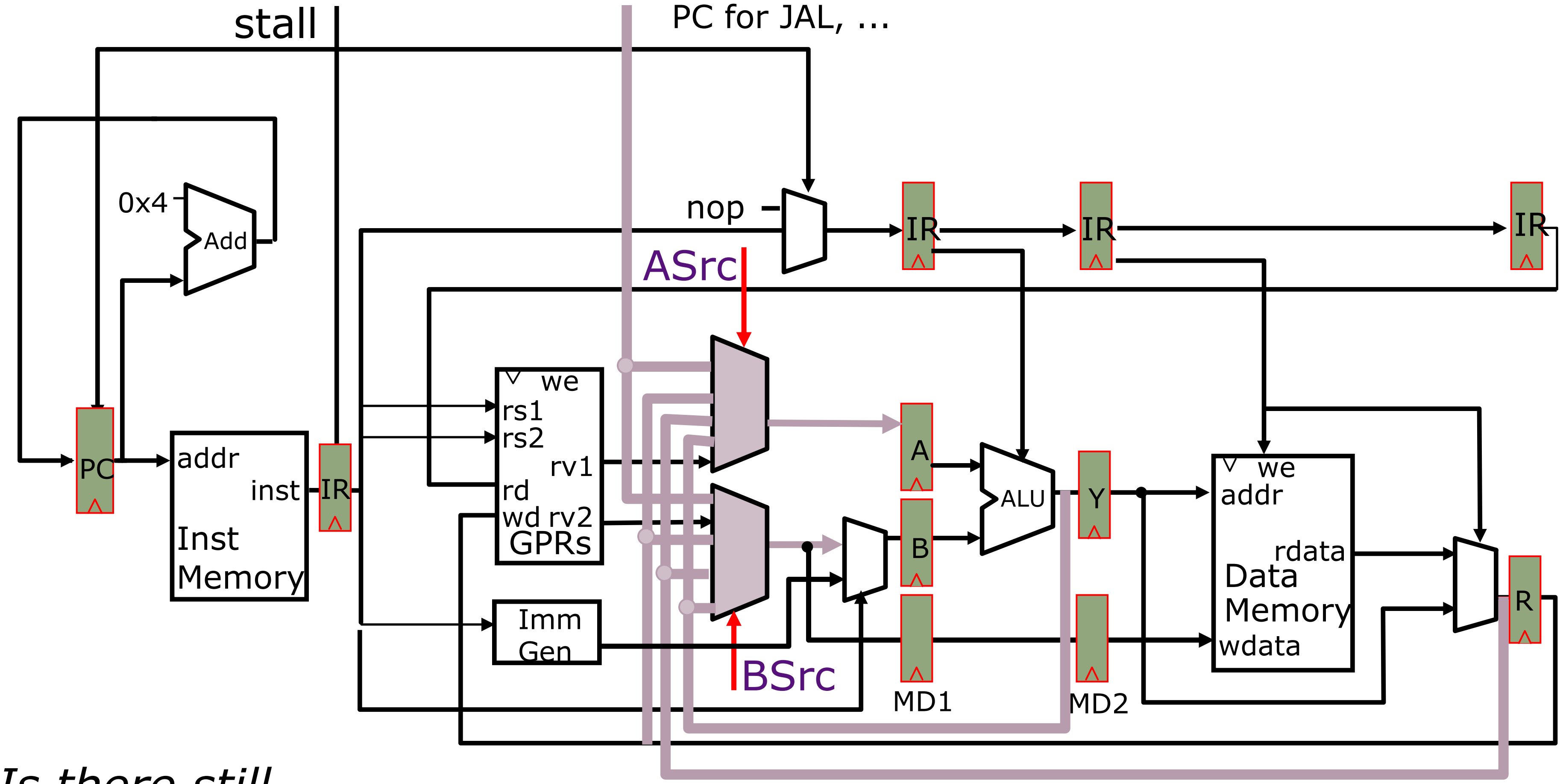
```
we-bypass_E = Case opcode_E
    ALU, ALUi    ⇒ (rd ≠ 0)
    ...          ⇒ off
```

```
we-stall_E = Case opcode_E
    LW, JAL, JALR    ⇒ (rd ≠ 0)
    ...              ⇒ off
```

$$\text{ASrc} = (rs1_D == rd_E) \cdot re1_D \cdot \text{we-bypass}_E$$

$$\begin{aligned}\text{stall} = \ & ((rs1_D == rd_E) \cdot \text{we-stall}_E + \\ & (rs1_D == rd_M) \cdot we_M + (rs1_D == rd_W) \cdot we_W) \cdot re1_D \\ & + ((rs2_D == rd_E) \cdot we_E + (rs2_D == rd_M) \cdot we_M + (rs2_D == rd_W) \cdot we_W) \cdot re2_D\end{aligned}$$

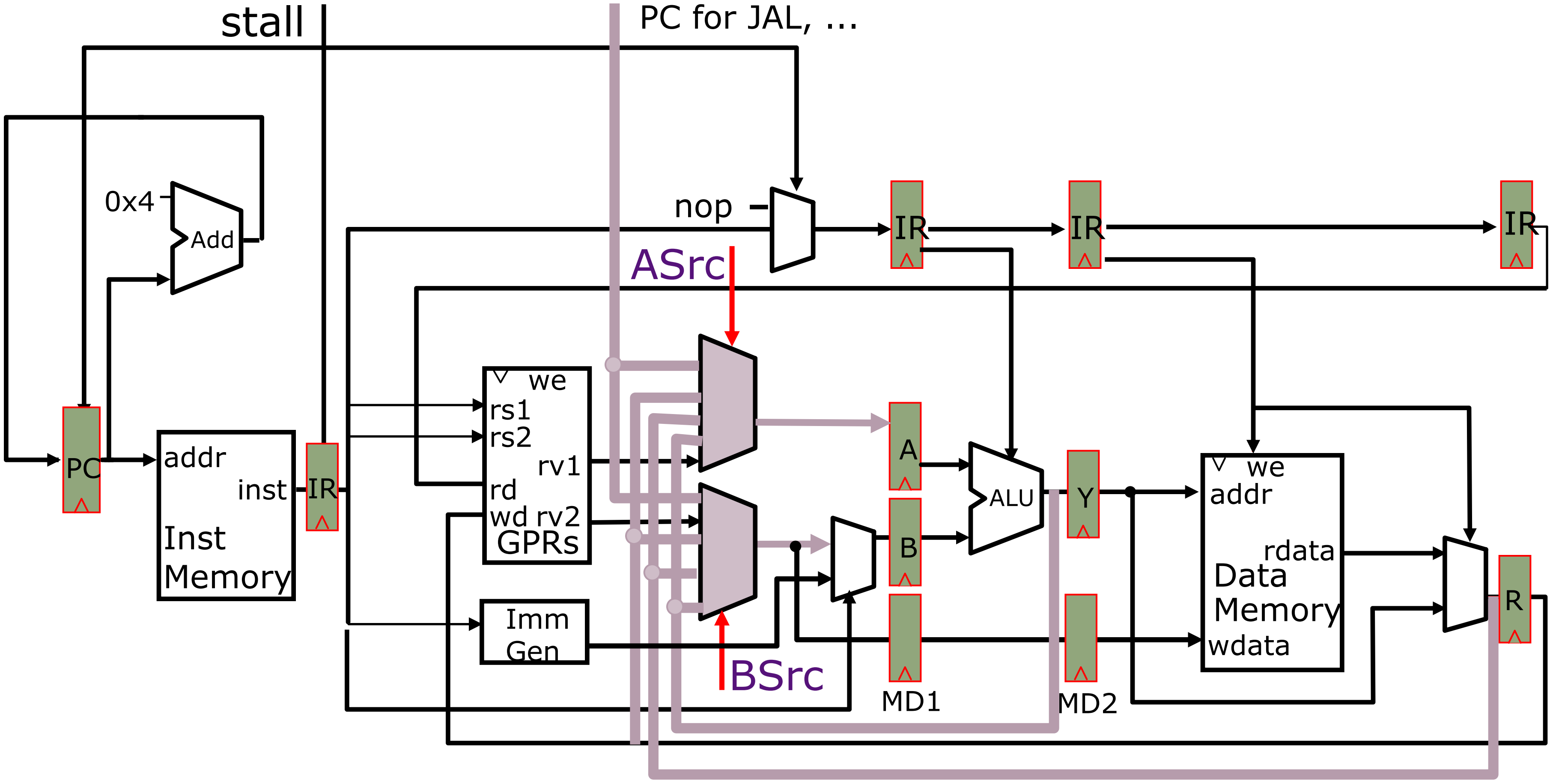
Full Bypass Datapath



Is there still a need for the stall signal?

$$\text{stall} = (rs1_D{=}{=}rd_E)\cdot(opcode_E{=}{=}LW_E)\cdot(rd_E\neq 0)\cdot re1_D + (rs2_D{=}{=}rd_E)\cdot(opcode_E{=}{=}LW_E)\cdot(rd_E\neq 0)\cdot re2_D$$

Full Bypass Datapath



Challenge: How to scale bypass datapaths
for more pipeline stages? More complex ISAs?

Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly → no special action required

Guessed incorrectly → kill and restart

Instruction to Instruction Dependence

- What do we need to calculate next PC?
  - For Jumps (JAL)
    - Opcode, offset, and PC
  - For Jump Register (JALR)
    - Opcode, offset, register value
  - For Conditional Branches (e.g., BEQ)
    - Opcode, offset, PC, and register (for condition)
  - For all others (e.g., arithmetic insts)
    - Opcode and PC
- In what stage do we know these?
  - PC → Fetch
  - Opcode, offset → Decode (or Fetch?)
  - Register value → Decode
  - Branch condition ((rs1)== (rs2)) → Execute (or Decode?)

NextPC Calculation Bubbles

time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) a1 $\leftarrow$ (a0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1						
(I_2) a3 $\leftarrow$ (a2) + 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

time

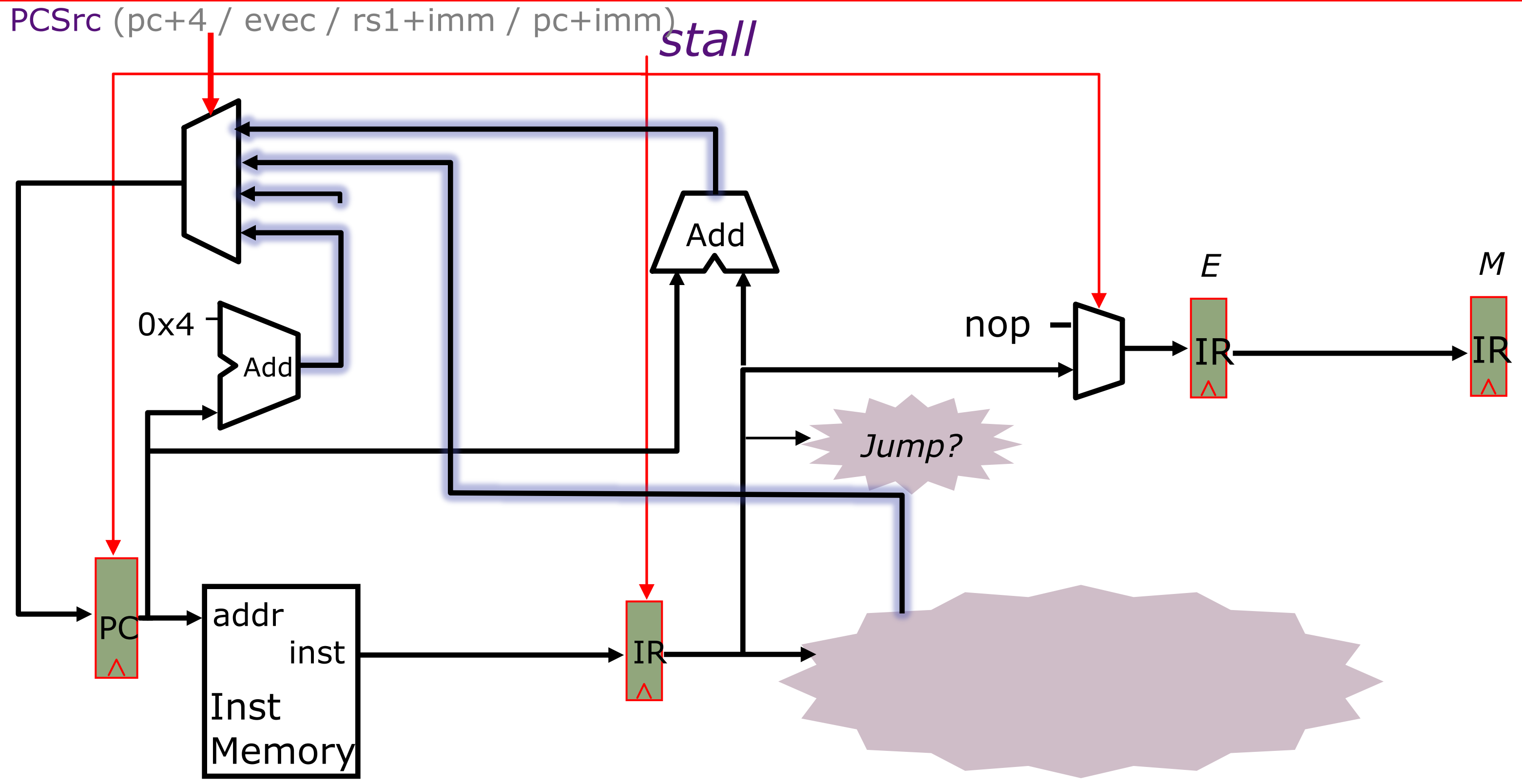
Resource Usage Diagram

	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

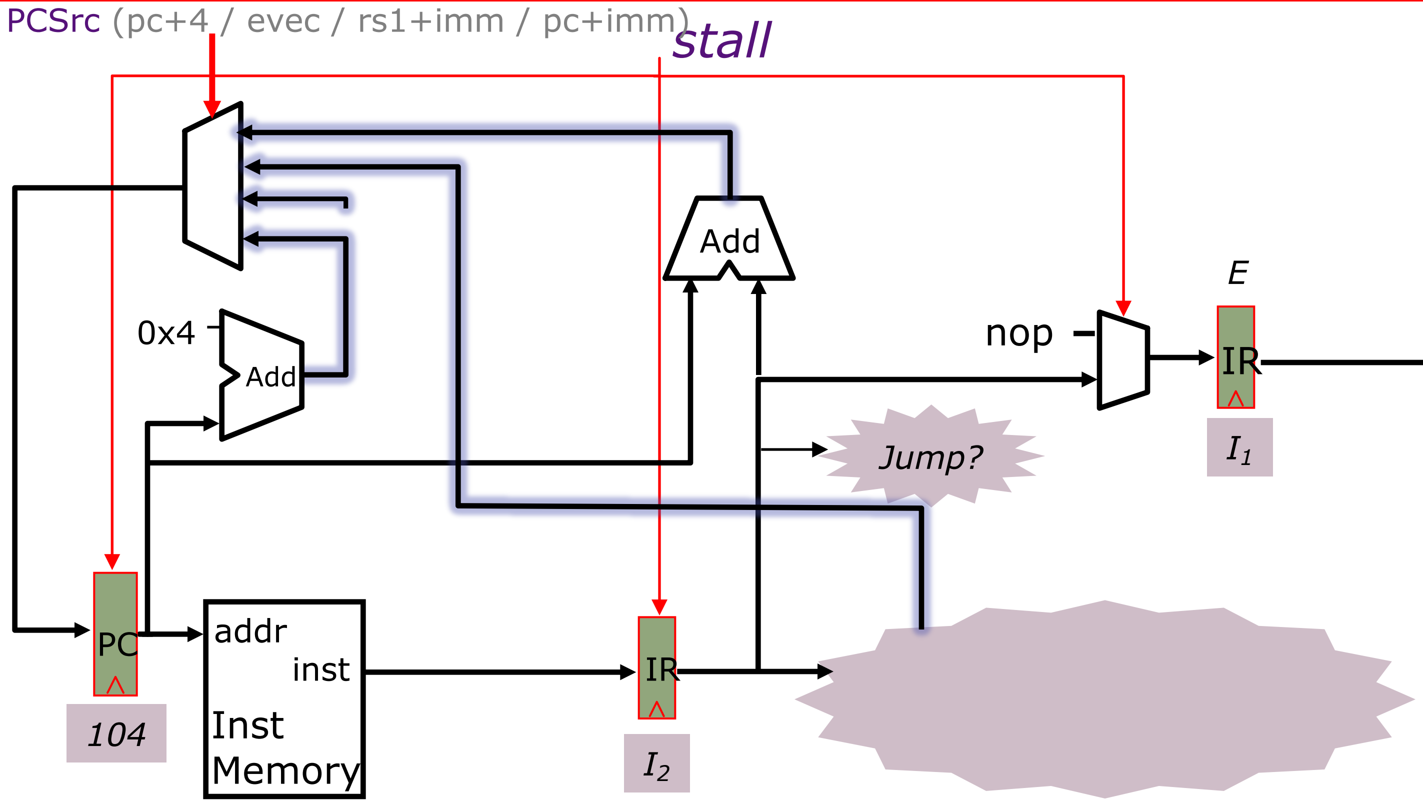
nop $\Rightarrow$ *pipeline bubble*

What's a good guess for next PC? PC+4

Speculate NextPC is PC+4



Speculate NextPC is PC+4

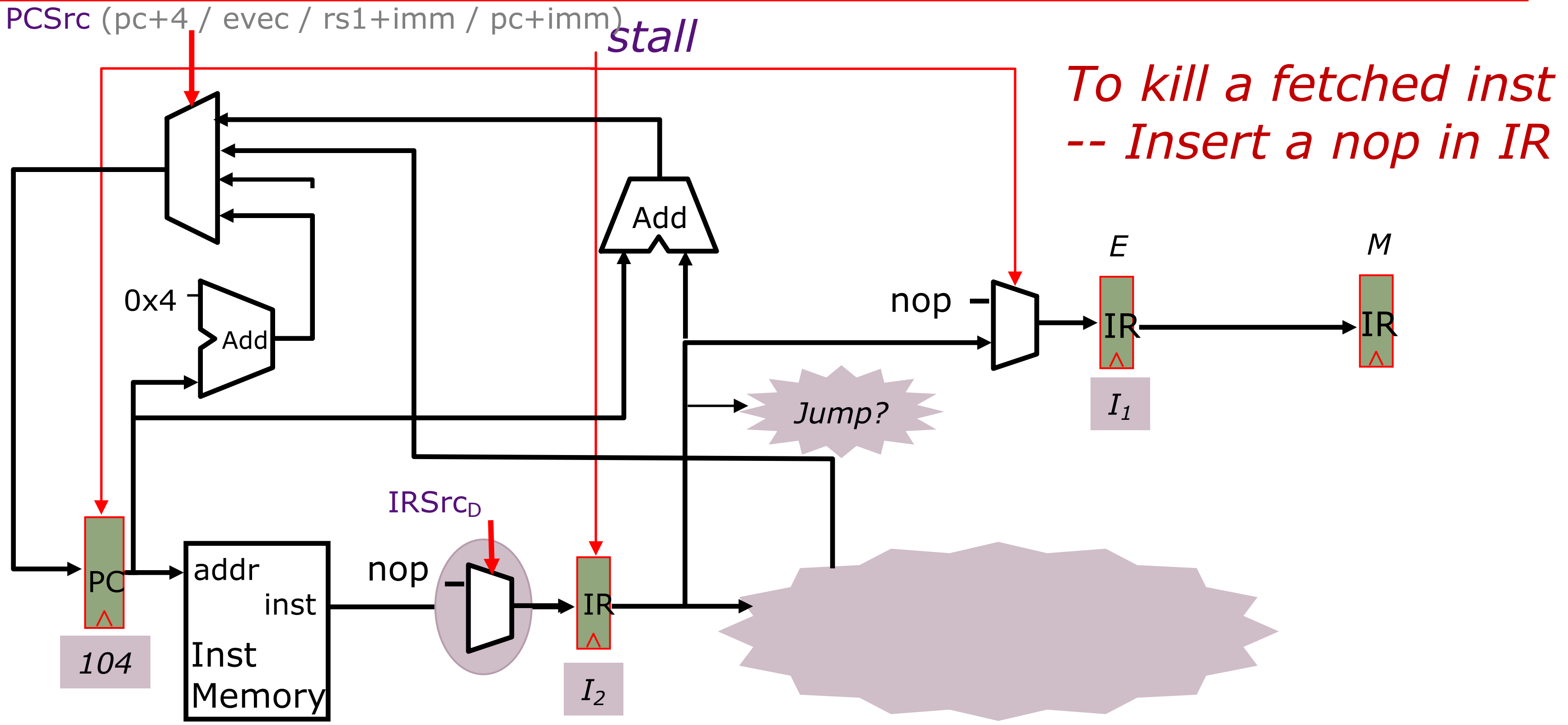


I_1	096	ADD
I_2	100	JAL ra 200
I_3	~~104~~	~~ADD~~ *kill*
I_4	304	ADD

What happens on mis-speculation, i.e., when next instruction is not PC+4?

How?

Pipelining Jumps

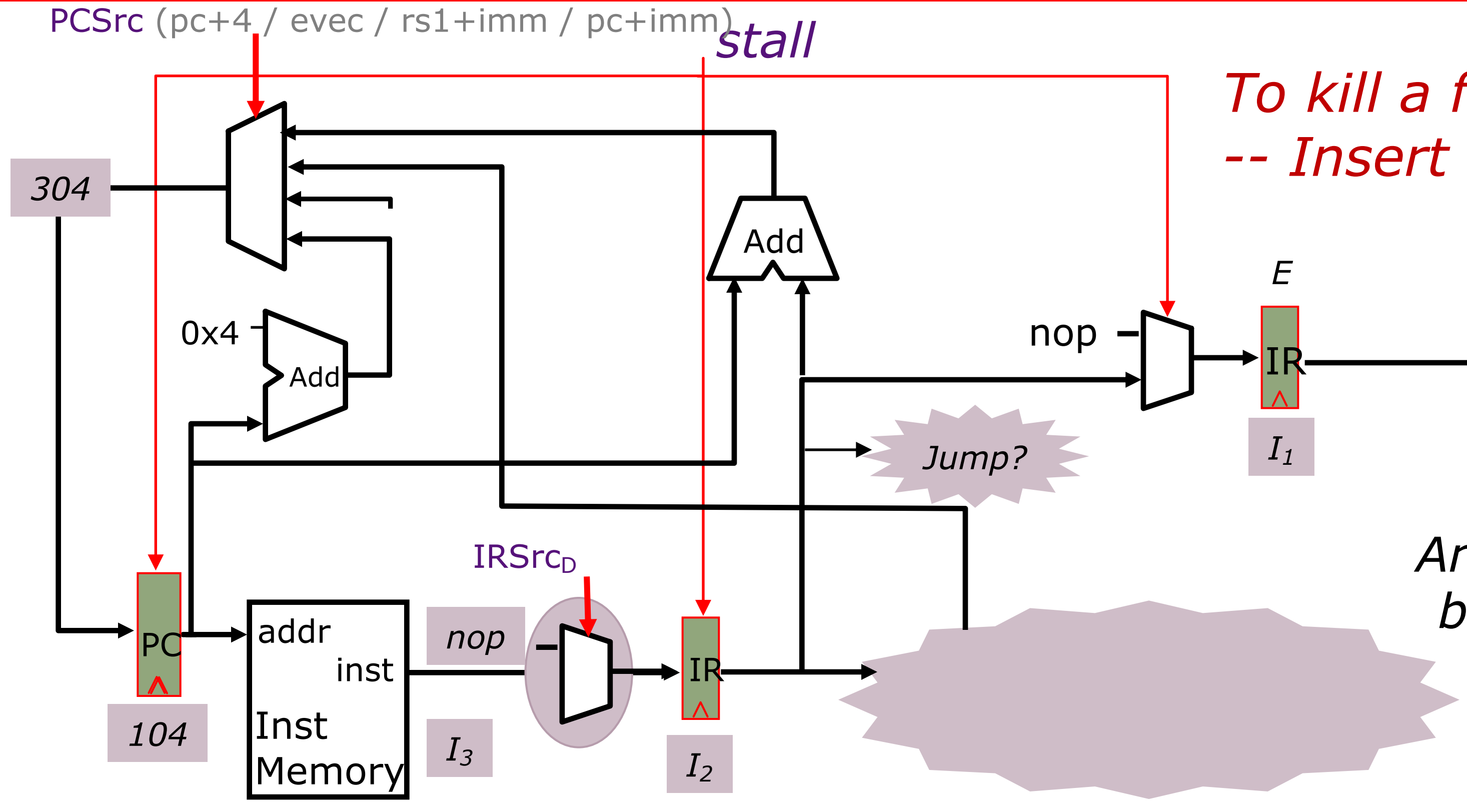


I_1	096	ADD
I_2	100	JAL ra 200
I_3	~~104~~	~~ADD~~
I_4	304	ADD

kill

$IRSrc_D$ = *Case* $opcode_D$
JAL, JALR ⇒ nop
... ⇒ IM

Pipelining Jumps



To kill a fetched inst
-- Insert a nop in IR

Any interaction between stall and jump?

No for now

I_1	096	ADD
I_2	100	JAL ra 200
I_3	~~104~~	~~ADD~~ *kill*
I_4	304	ADD

$IRSrc_D$ = *Case* $opcode_D$
JAL, JALR ⇒ nop
... ⇒ IM

Jump Pipeline Diagrams

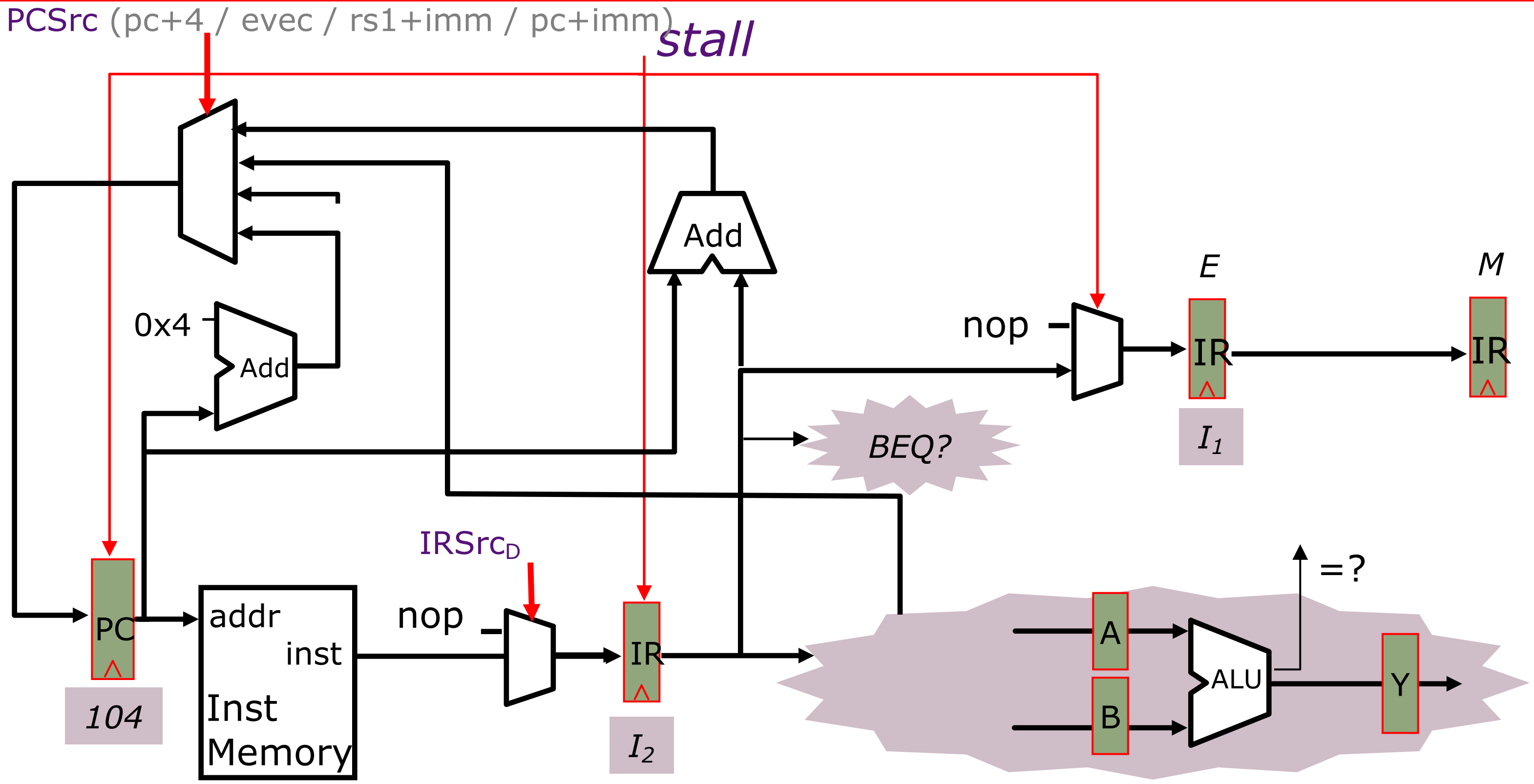
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: JAL ra 200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage Diagram

	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

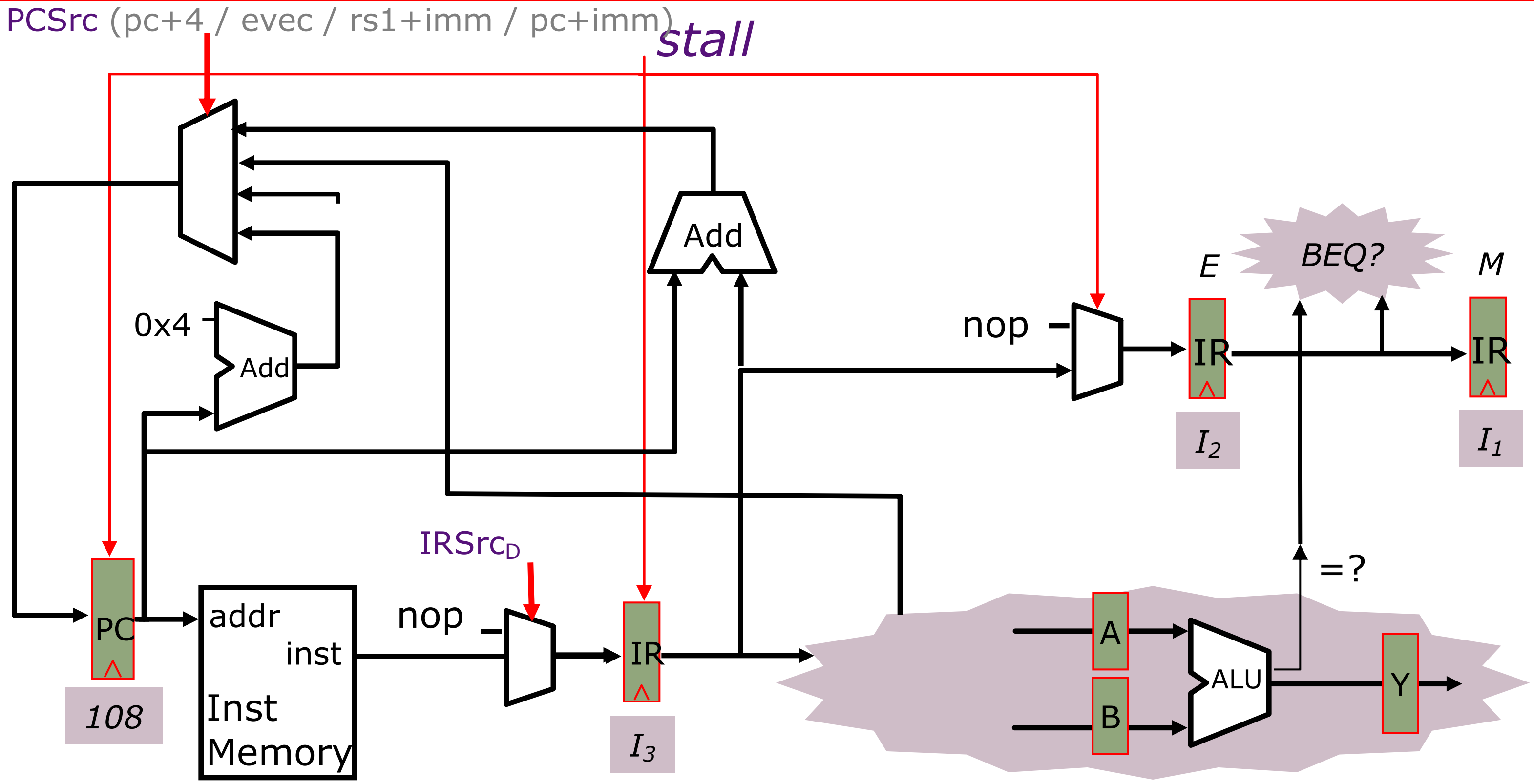


Branch condition is not known until the execute stage. *What action should be taken in the decode stage?*

I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

Continue speculating. Decode I2 and fetch PC+4

Pipelining Conditional Branches

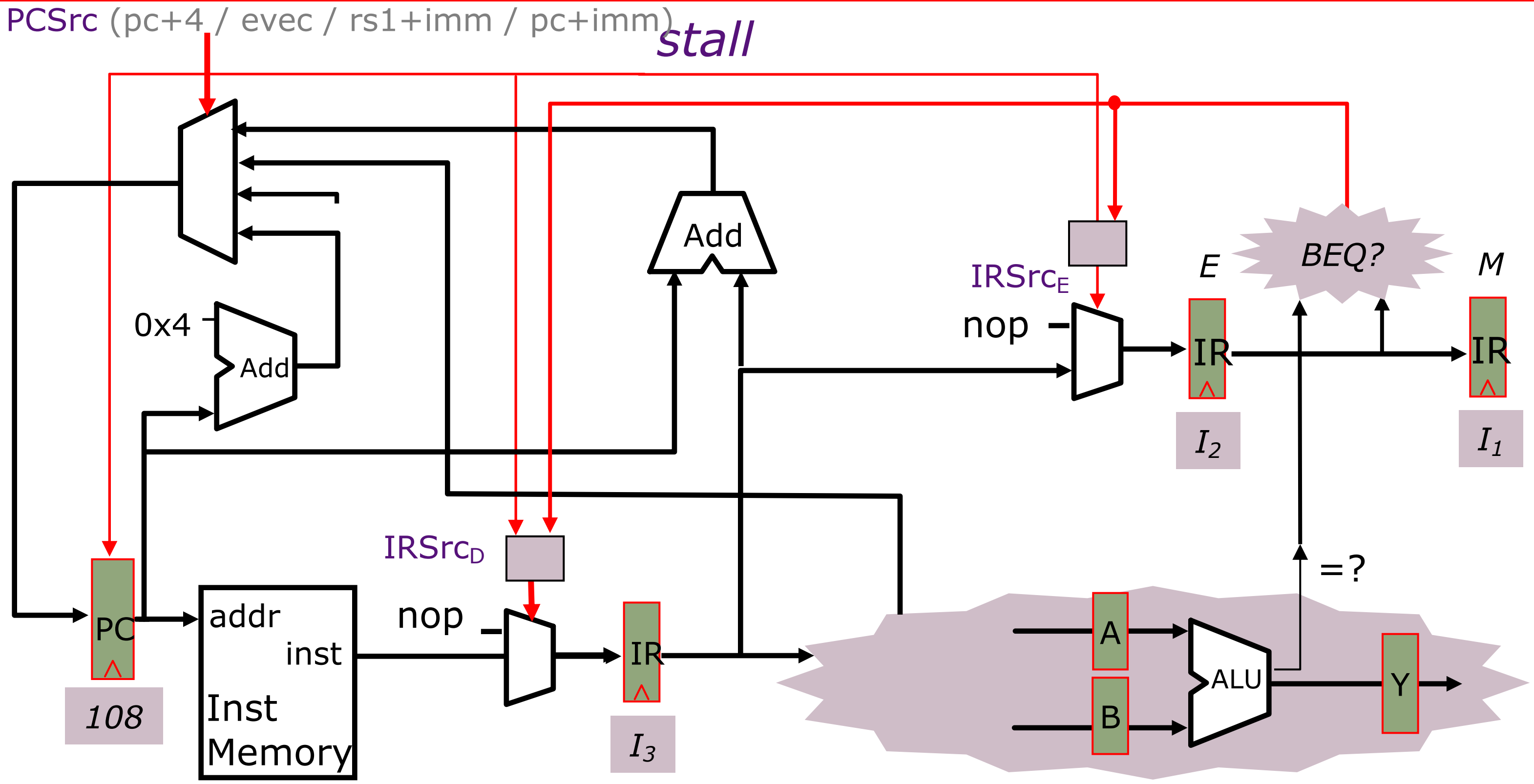


I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

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

$\Rightarrow$ *stall signal is not valid*

Pipelining Conditional Branches



I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

Don’t stall if the branch is taken. Why?

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

```
IRSrc_D = Case opcode_E
      BEQ·z  ⇒ nop
      ...    ⇒ Case opcode_D
                  JAL, JALR ⇒ nop
                  ...       ⇒ IM
```

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction

```
IRSrc_E = Case opcode_E
      BEQ·z ⇒ nop
      ...   ⇒ stall·nop + !stall·IR_D
```

```
PCSrc = Case opcode_E
      BEQ·z ⇒ pc + imm
      ...   ⇒ Case opcode_D
                 JAL    ⇒  pc + imm
                 JALR   ⇒  rs1 + imm
                 ...    ⇒  pc+4
```

pc+4 is a speculative guess

nop ⇒ Kill
pc+imm /rs1+imm ⇒ Restart
pc+4 ⇒ Speculate

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: BEQ a1 a2 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 Diagram

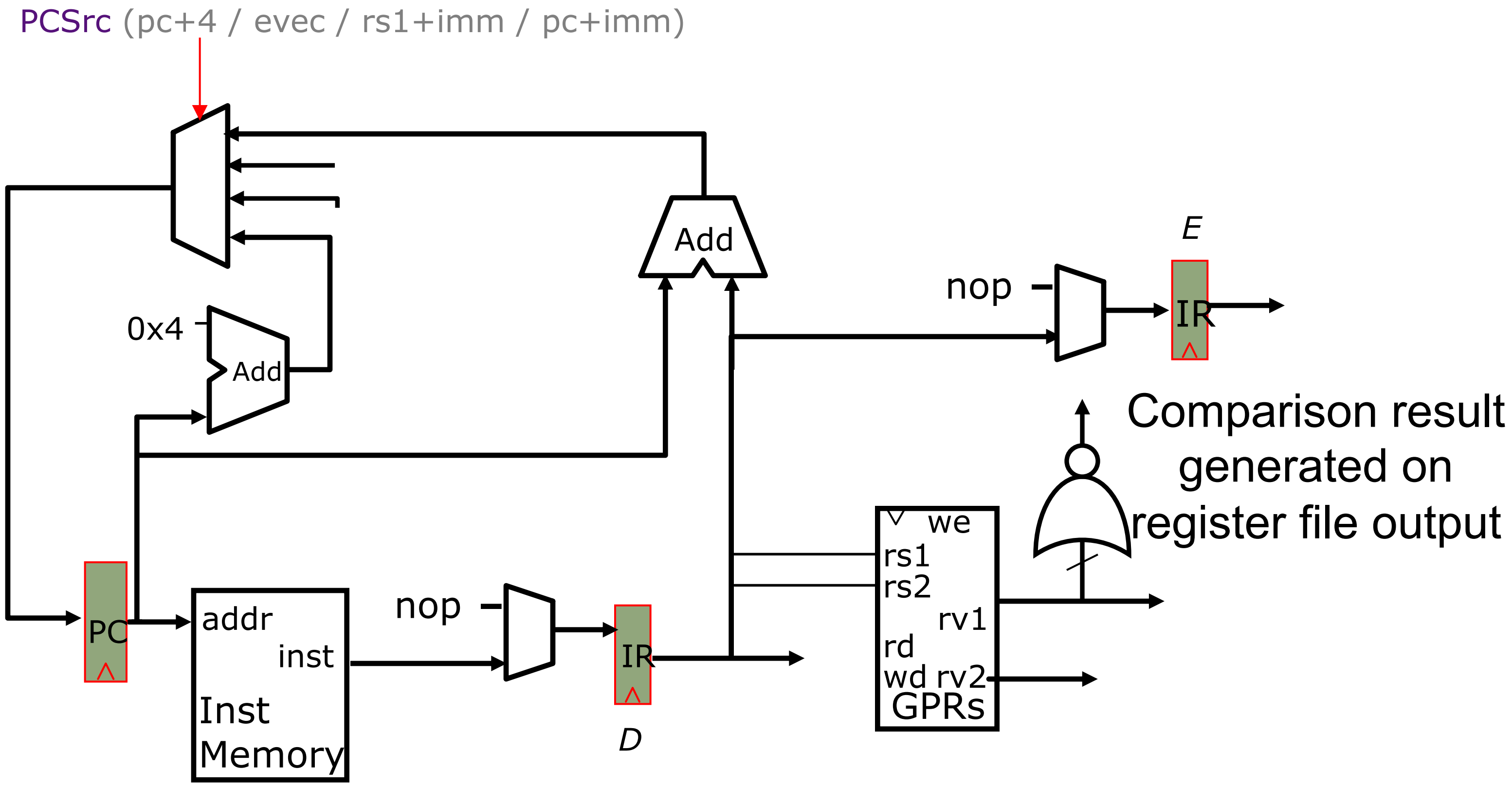
time

	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 $\Rightarrow$ *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



PCSrc (pc+4 / evec / rs1+imm / pc+imm)
Add
0x4
Add
E
nop
IR
Comparison result generated on register file output
we
rs1
rs2
rv1
rd
wd rv2
GPRs
nop
IR
D
PC
addr
inst
Inst Memory

Pipeline diagram now same as for jumps

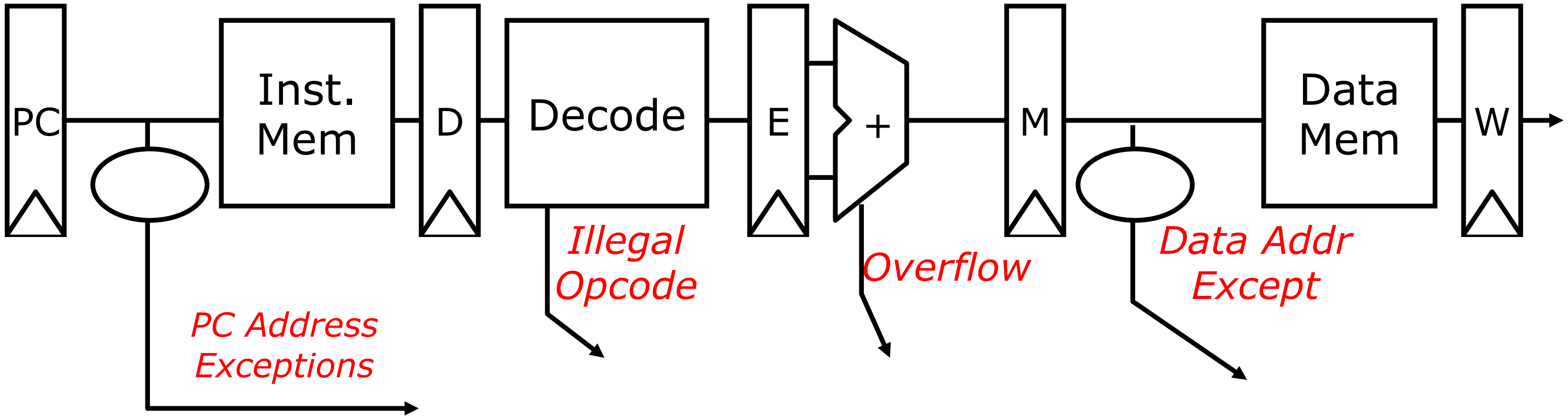
Branch Delay Slots
(expose control hazard to software)

- 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.

I_1	096	ADD	
I_2	100	BEQ a1 a2 200	
I_3	104	ADD ←	*Delay slot instruction executed regardless of branch outcome*
I_4	304	ADD	

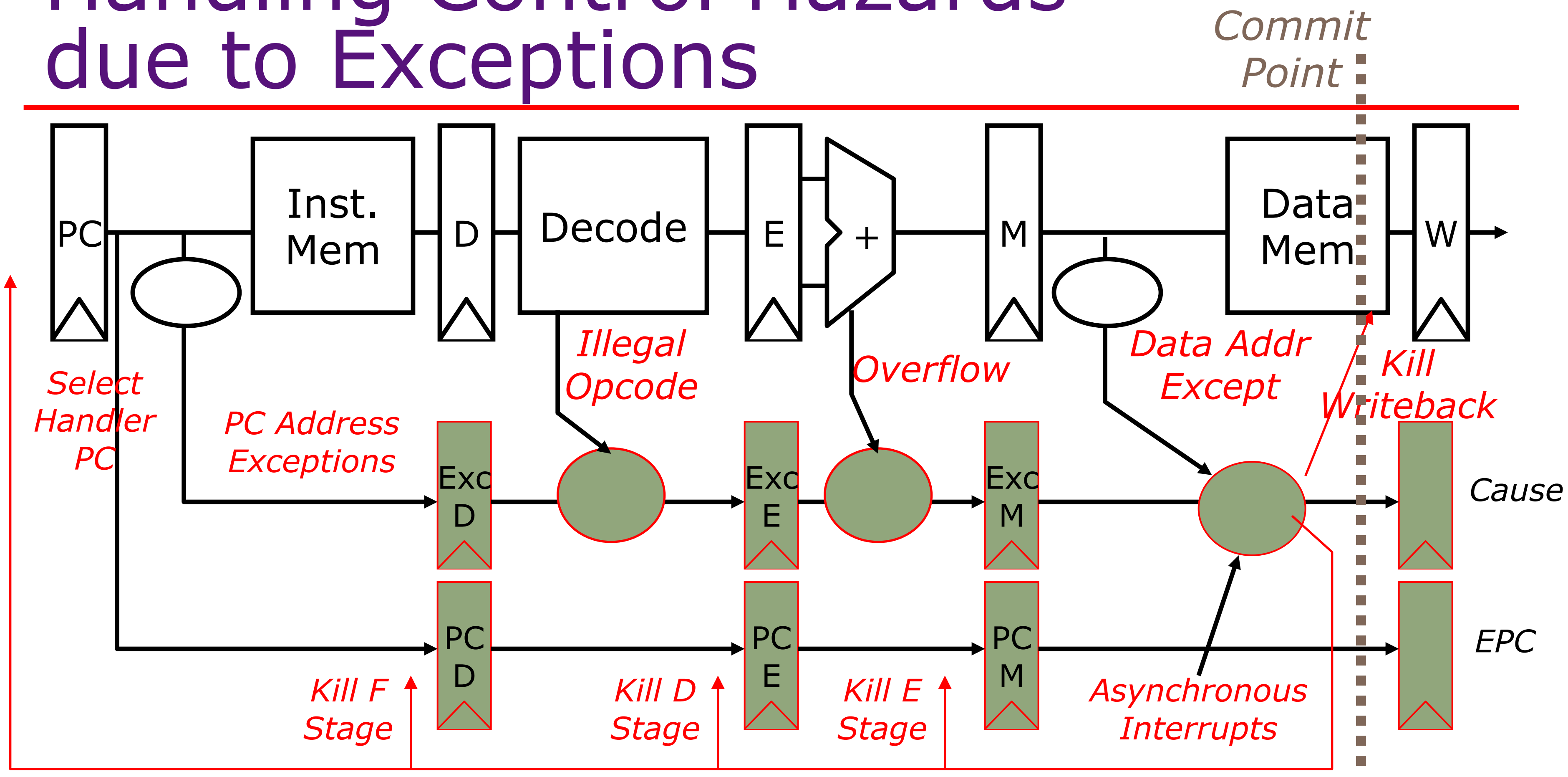
- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

Handling Control Hazards due to Exceptions



- Instructions may suffer exceptions in different pipeline stages
- Must prioritize exceptions from earlier instructions

Handling Control Hazards due to Exceptions



- Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)
  - *Pros/cons vs handling exceptions eagerly, like branches?*

Why an instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - Typically, all frequently used paths are provided
  - Some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches, jumps, and exceptions may cause bubbles
  - Kill instruction(s) following branch if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.

Next lecture:
Superscalar & Scoreboarded Pipelines