ECE 252 / CPS 220
Advanced Computer Architecture I

Lecture 7
Pipelining – Part 2

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

ECE252 Administrivia

29 September – Homework #2 Due

- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

4 October – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. "Optimizing pipelines for power and performance"
2. Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors"
3. Palacharla et al. "Complexity-effective superscalar processors"
4. Yeh et al. "Two-level adaptive training branch prediction"

Data Hazards and Scheduling

Try producing faster code for

- A = B + C; D = E – F;
- Assume A, B, C, D, E, and F are in memory
- Assume pipelined processor

Slow Code

```
LW      Rb, b
LW      Rc, c
ADD     Ra, Rb, Rc
SW      a, Ra
LW      Re e
LW      Rf, f
SUB     Rd, Re, Rf
SW      d, RD
```

Fast Code

```
LW      Rb, b
LW      Rc, c
LW      Re, e
ADD     Ra, Rb, Rc
LW      Rf, f
SW      a, Ra
SUB     Rd, Re, Rf
SW      d, RD
```

Compiler Scheduling

Reduce stalls by moving instructions

- Basic pipeline scheduling eliminates back-to-back load-use pairs
- What are the limitations of scheduling?

Scheduling Scope

- Requires an independent instruction to place between load-use pairs
- Little scope for scheduling, 1-add, 3-ld/st

Slow Code

```
ld      r2, 4(sp)
ld      r3, 8(sp)
add     r3, r2, r1
st      r1, 0(sp)
```

Fast Code

```
ld      r2, 4(sp)
ld      r3, 8(sp)
add     r3, r2, r1
st      r1, 0(sp)
```

Compiler Scheduling

Number of registers

- Registers hold "live" values
- Example code contains 7 different values, including sp
- Before: max 3 values live → 3 registers sufficient
- After: max 4 values live → 3 registers insufficient
- Original code re-uses r1 and r2, re-scheduling causes WAR violations

Before

```
ld    r2, 4 (sp)
ld    r1, 8 (sp)
add   r1, r2, r1   #stall
st    r1, 0 (sp)
ld    r2, 16 (sp)
ld    r1, 20 (sp)
sub   r2, r1, r1   #stall
st    r1, 12 (sp)
```

After

```
ld    r2, 4 (sp)
ld    r1, 8 (sp)
ld    r2, 16 (sp)
add   r1, r2, r1   #WAR
ld    r1, 20 (sp)
st    r1, 0 (sp)   #WAR
sub   r2, r1, r1
st    r1, 12 (sp)
```

Compiler Scheduling

Alias Analysis

- Determine whether load/stores reference same memory locations
- Determines if loads/stores can be re-ordered
- Previous Examples: easy, all loads/stores use the same base register (sp)
- New Example: Can compiler tell that r8 = sp?

Before

```
ld    r2, 4 (sp)
ld    r3, 8 (sp)
add   r3, r2, r1   #stall
st    r1, 0 (sp)
ld    r5, 0 (r8)
ld    r6, 4 (r8)
sub   r5, r6, r4   #stall
st    r4, 8 (r8)
```

After

```
ld    r2, 4 (sp)
ld    r3, 8 (sp)
ld    r5, 0 (r8)
add   r3, r2, r1
ld    r6, 4 (r8)
st    r1, 0 (sp)
sub   r5, r6, r4
st    r4, 8 (r8)
```

Control Hazards

Arises when calculating next program counter (PC)

- Pipeline stalls if required values not yet available

Jumps

- Jumps (immediate) requires opcode, offset, current PC
- Jump (register) requires opcode, register value

Conditional Branches

- Requires opcode, current PC, register (for condition), offset

Sequential Successor Instructions

- Opcode, current PC

Program Counter Calculations

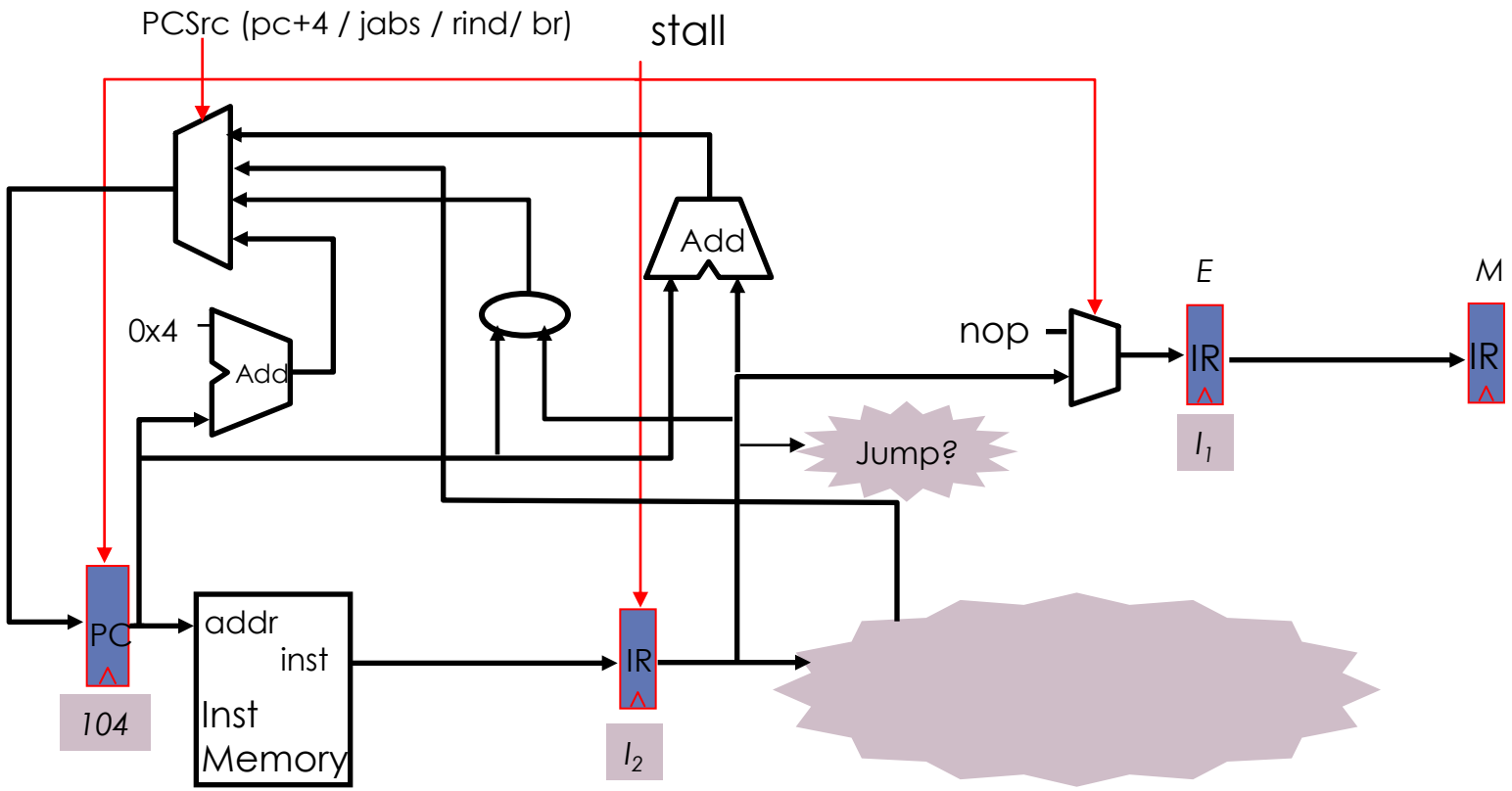
Identify change in control flow during decode

	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

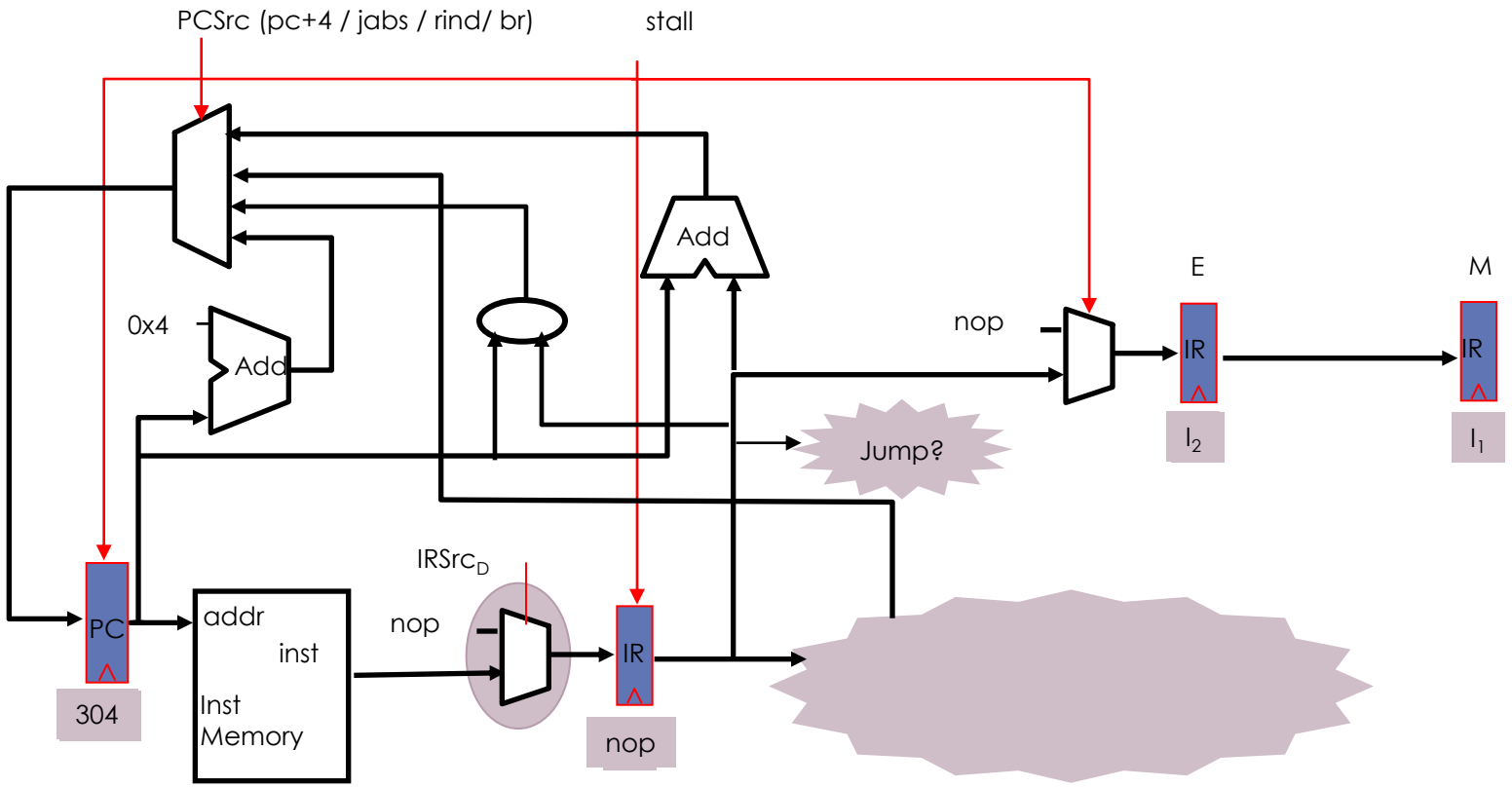
Speculate PC ← PC + 4



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

A jump instruction kills (not stalls) the following instruction. How?

Pipelining Jumps



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

To kill a fetched instruction, add mux before IR to insert "nops"

case(opcodeD)

J, JAL: IR ← nop

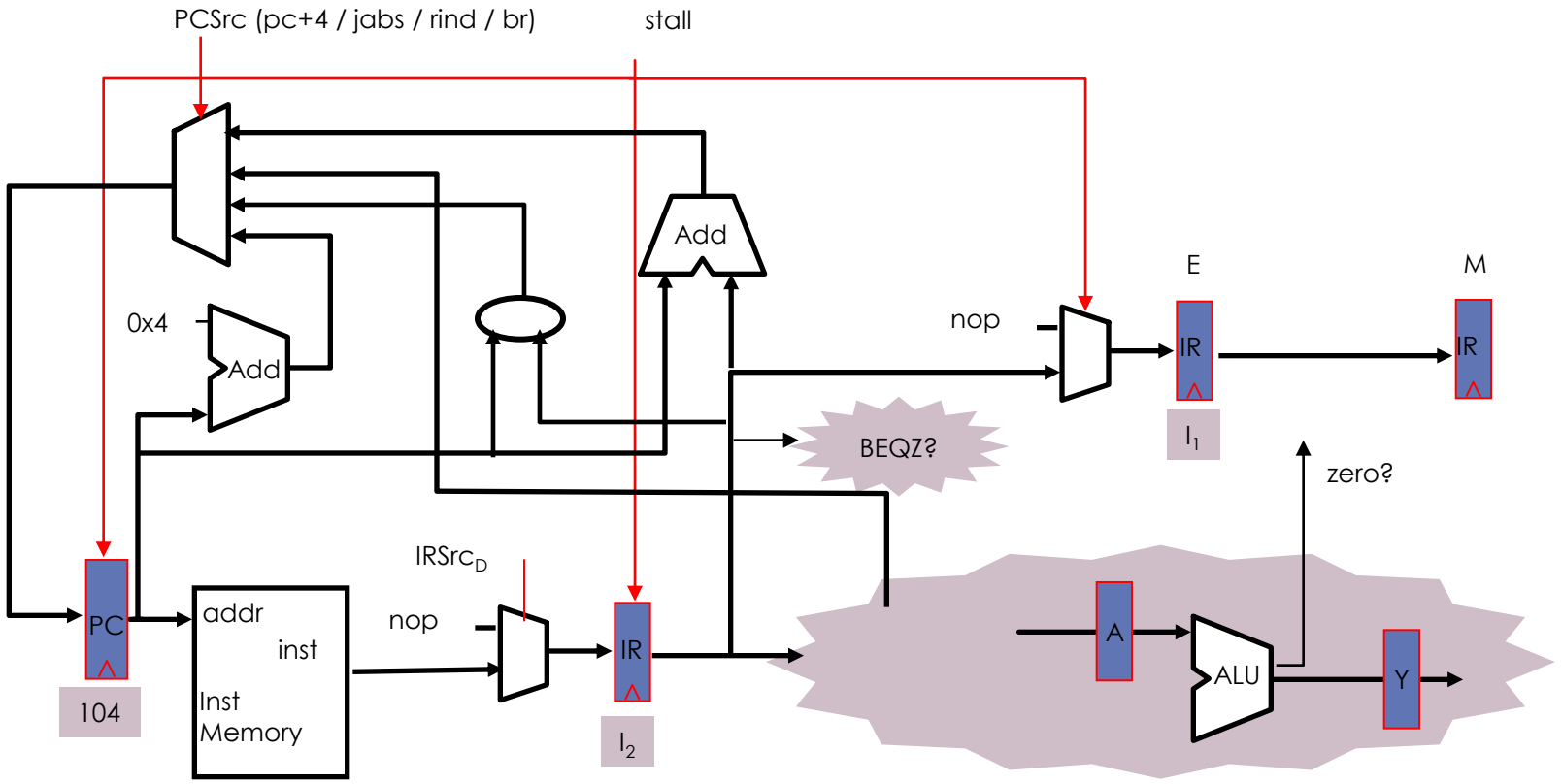
otherwise: IR ← inst

Pipelining Jumps

	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		

		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			
Resource	EX			I_1	I_2	nop	I_4	I_5		
Usage	MA				I_1	I_2	nop	I_4	I_5	
	WB					I_1	I_2	nop	I_4	I_5

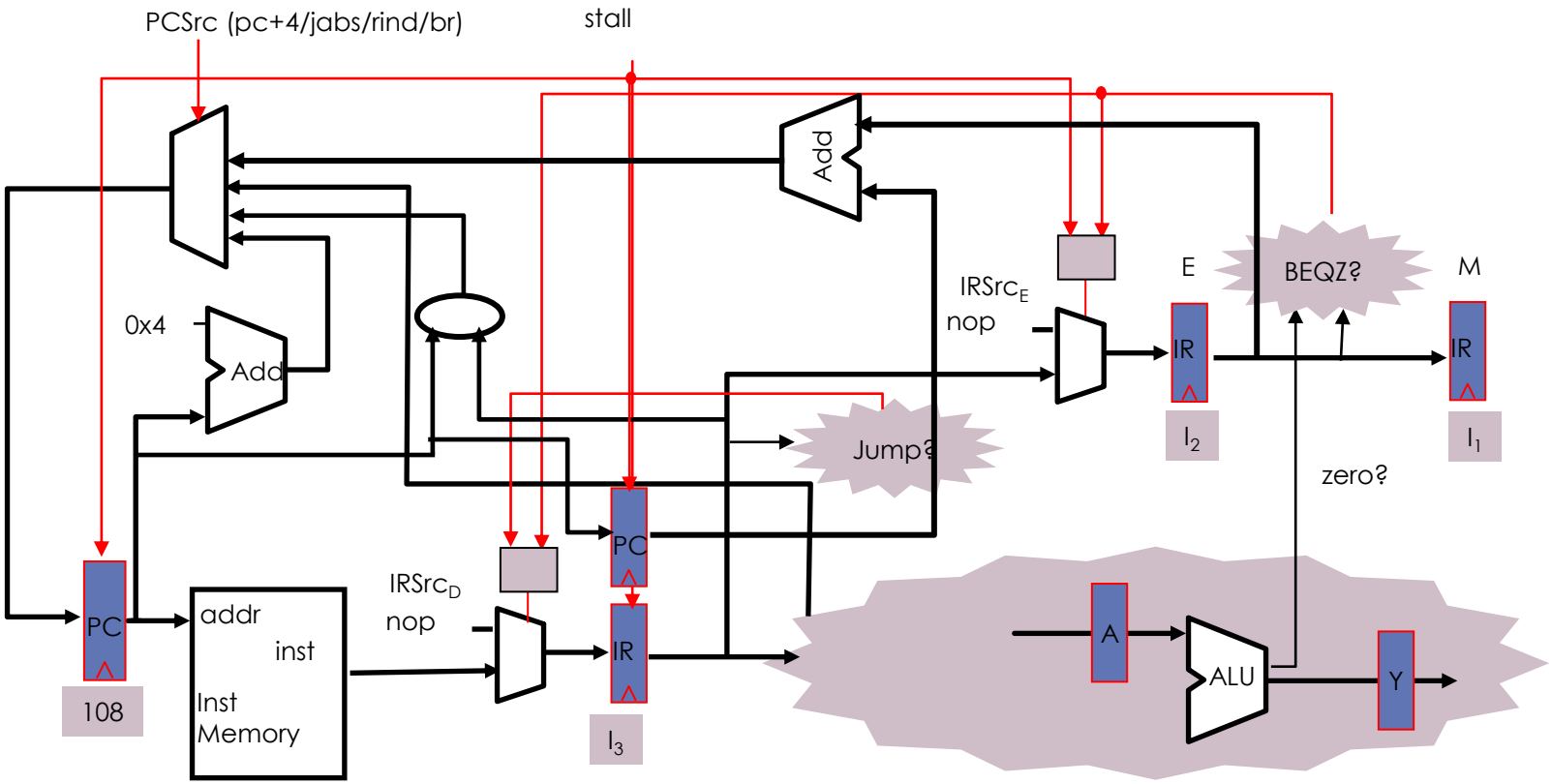
Pipelining Conditional Branches



	addr	instr
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Branch condition computed in execute stage. What should be done in decode stage?

Pipelining Conditional Branches



	addr	instr
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

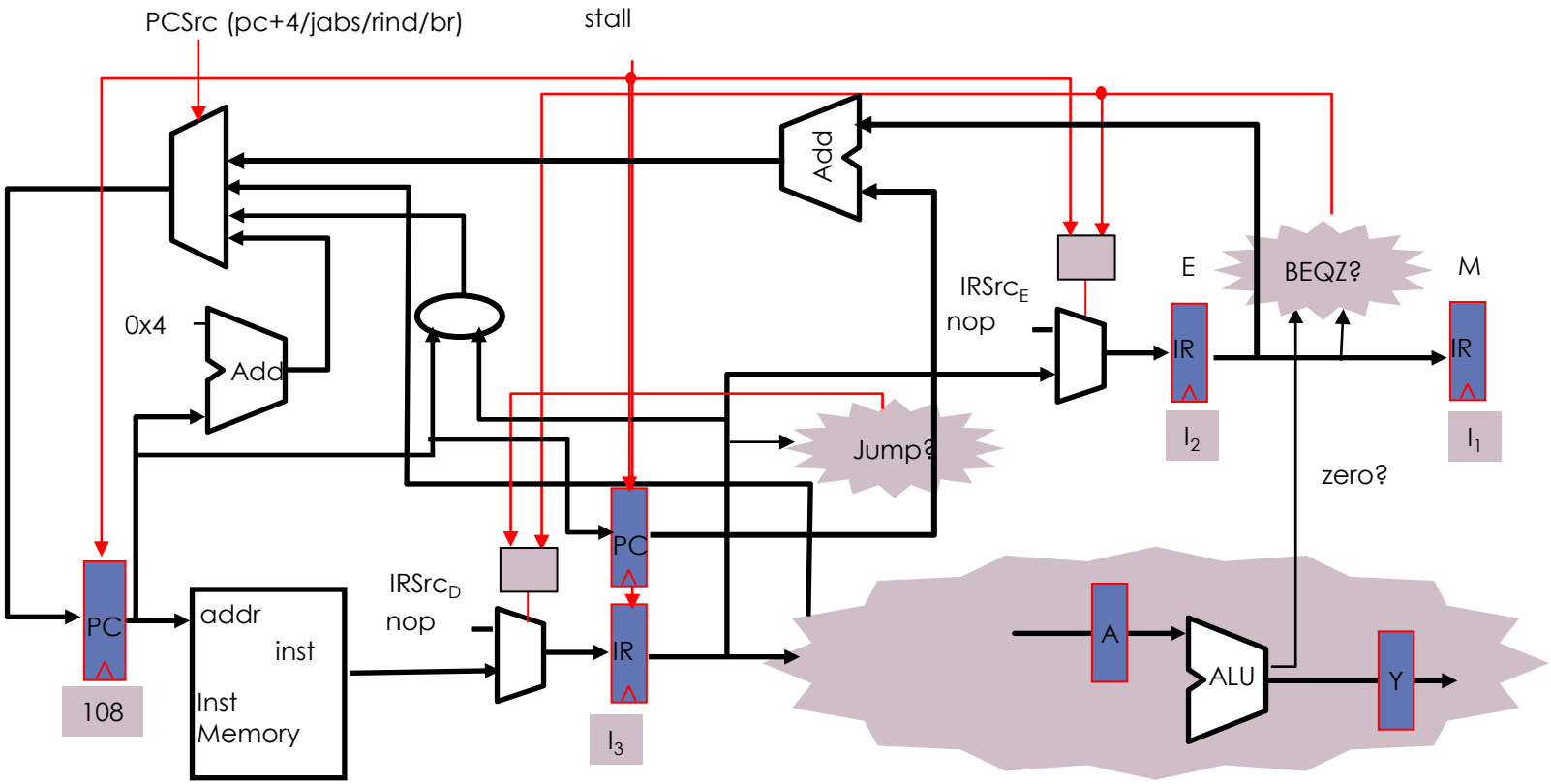
If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

Update Stall Signal

```
Stall ← <<original stall signal>>
        & !(        (opcodeE == BEQZ) & zero?        # branch condition true
                  + (opcodeE == BNEZ) & !zero?       # branch condition true
        )
```

Do not stall if branch is taken. Why?
Instruction in the decode stage is invalid.
Kill instruction instead.

Pipelining Conditional Branches



	addr	instr
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

Derive PCSrc Signal

Derive mux control signal for PCSrc.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), PCSrc ← br
else if ((opcodeD == J) + (opcodeD == JAL)), PCSrc ← jabs
else if ((opcodeD == JR) + (opcodeD == JALR)), PCSrc ← rind
otherwise, PCSrc ← PC + 4

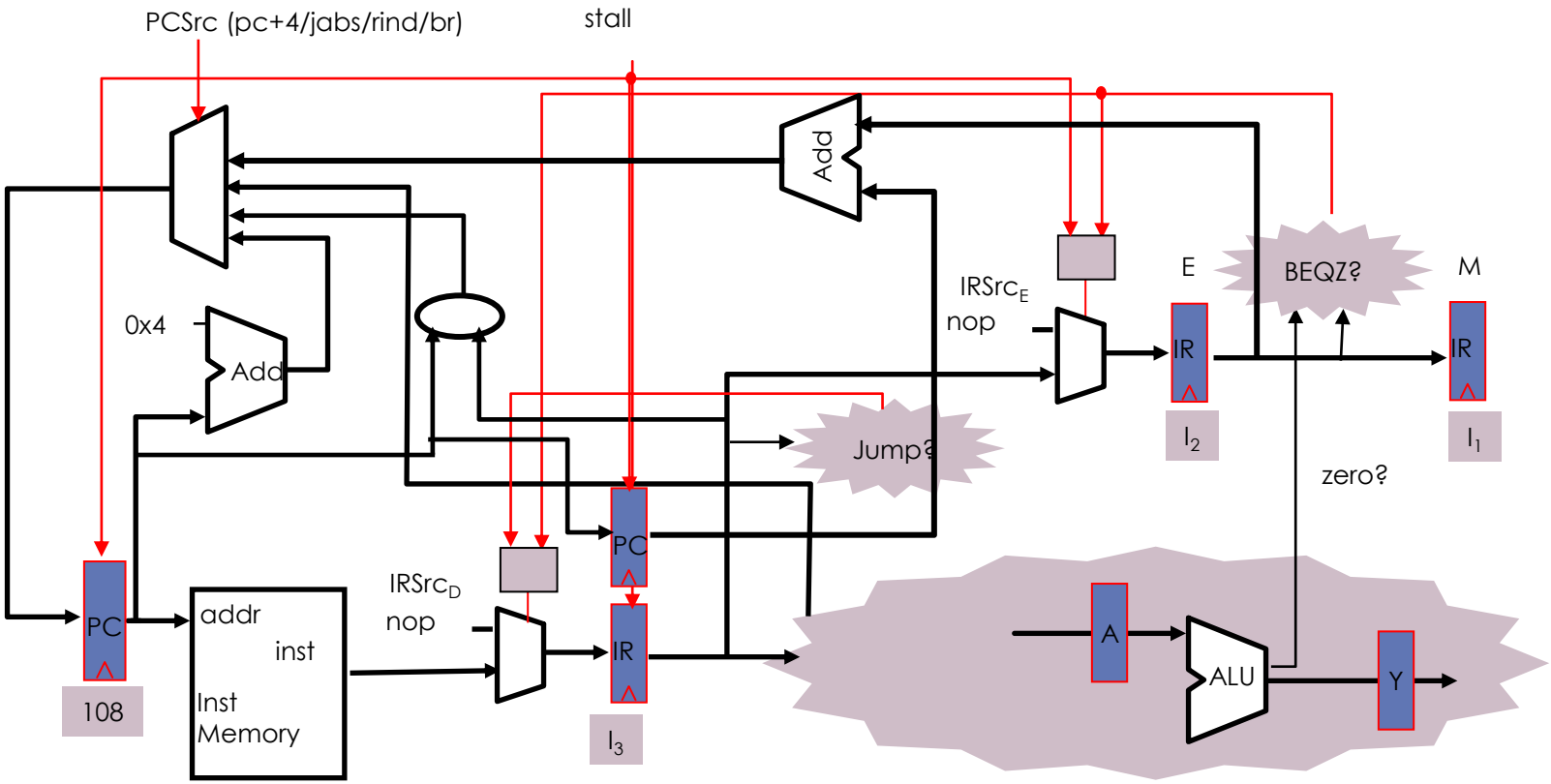
Derive mux control signal for IRSrcD.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), ICSrcD ← nop
else if((opcodeD==J) + (opcodeD==JAL) +
(opcodeD==JAR) + (opcodeD==JALR)), ICSrcD ← nop
otherwise, ICSrcD ← Instr

Derive mux control signal for IRSrcE.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), ICSrcE ← nop
otherwise, IRSrcE ← (stall & nop) + (!stall & IRD)

Pipelining Conditional Branches



	addr	instr
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

Derive IRSrcD Signal

Derive mux control signal for PCSrc.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), PCSrc ← br
else if ((opcodeD == J) + (opcodeD == JAL)), PCSrc ← jabs
else if ((opcodeD == JR) + (opcodeD == JALR)), PCSrc ← rind
otherwise, PCSrc ← PC + 4

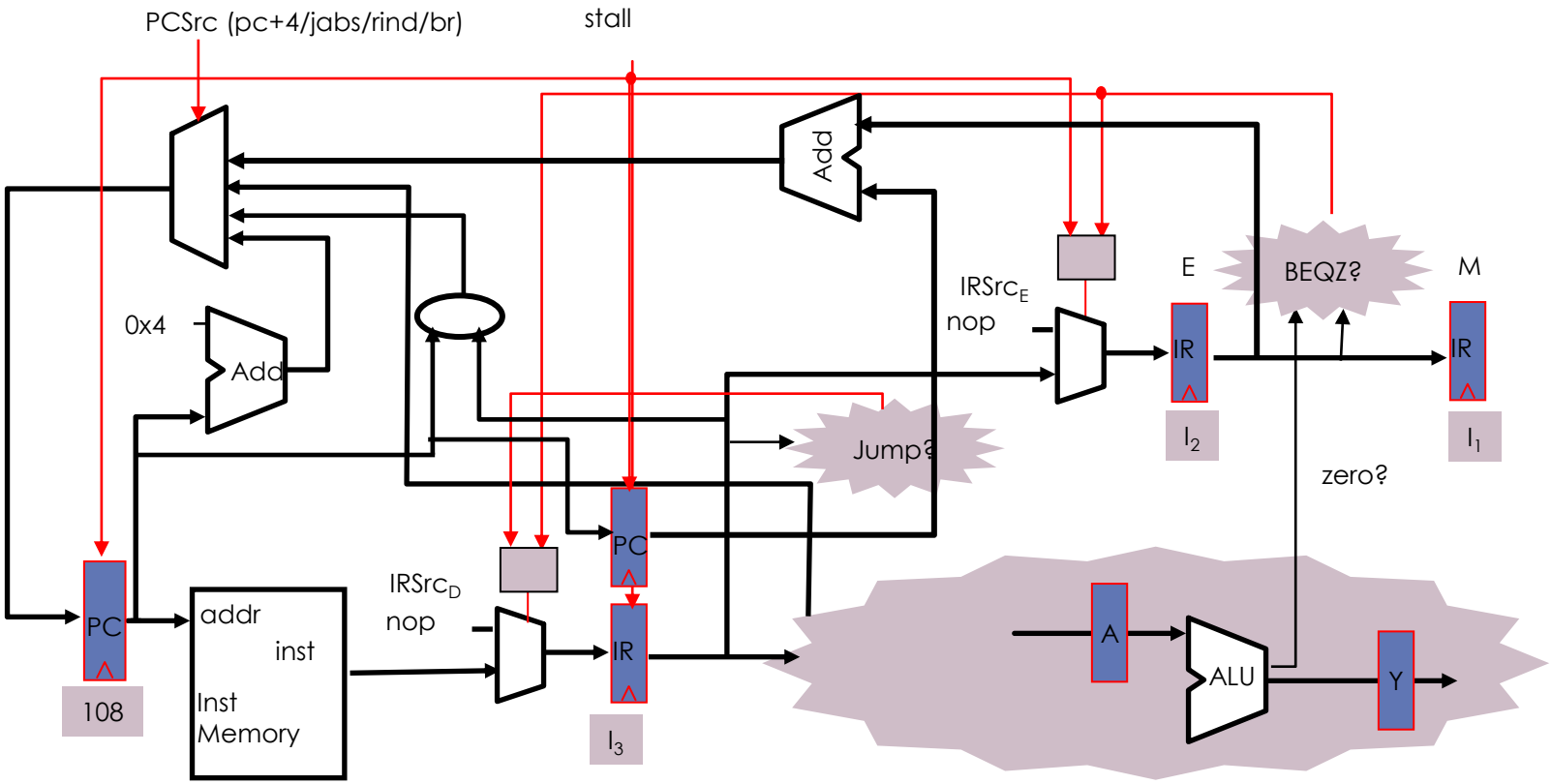
Derive mux control signal for IRSrcD.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), ICSrcD ← nop
else if((opcodeD==J) + (opcodeD==JAL) +
(opcodeD==JAR) + (opcodeD==JALR)), ICSrcD ← nop
otherwise, ICSrcD ← Instr

Derive mux control signal for IRSrcE.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), ICSrcE ← nop
otherwise, IRSrcE ← (stall & nop) + (!stall & IRD)

Pipelining Conditional Branches



	addr	instr
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If branch is taken, kill two following instructions. And because instruction in decode stage is invalid, update stall signal.

Derive IRSrcE Signal

Derive mux control signal for PCSrc.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), PCSrc ← br
else if ((opcodeD == J) + (opcodeD == JAL)), PCSrc ← jabs
else if ((opcodeD == JR) + (opcodeD == JALR)), PCSrc ← rind
otherwise, PCSrc ← PC + 4

Derive mux control signal for IRSrcD.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), IRSrcD ← nop
else if((opcodeD==J) + (opcodeD==JAL) +
(opcodeD==JAR) + (opcodeD==JALR)), IRSrcD ← nop
otherwise, IRSrcD ← Instr

Derive mux control signal for IRSrcE.

if((opcodeE == BEQZ & z) + (opcodeE == BNEZ & !z)), IRSrcE ← nop
otherwise, IRSrcE ← (stall & nop) + (!stall & IRD)

Pipelining Branches

	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	

		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			
Resource	EX			I_1	I_2	nop	nop	I_5		
Usage	MA				I_1	I_2	nop	nop	I_5	
	WB					I_1	I_2	nop	nop	I_5

Resolving Branch Conditions

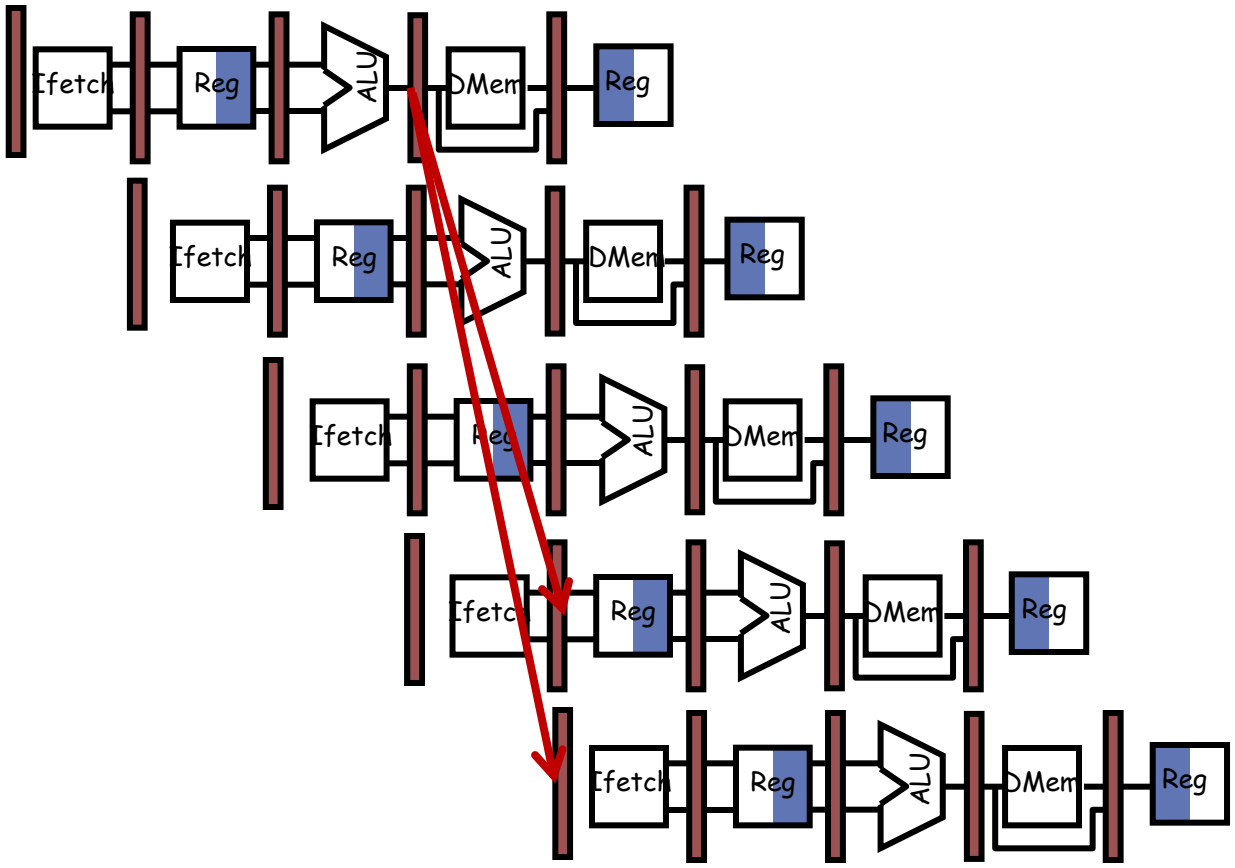
beq r1, r3, 36

and r2, r3, r5

or r6, r1, r7

add r8, r1, r9

xor r10,r1,r11



Three stage stall. What about the 3 instructions in between? Attempt to reduce. Stall/Kill signals in decode stage, reduces stalls from 3 to 2 stages.

Solution 1: Resolve Earlier

Large performance impact

- Suppose CPI = 1, 30% branch
- If branch stalls for 3 cycles, new CPI is 1.9

Solution – Branch Computation

- Determine whether branch is taken or not earlier in pipeline (e.g,. beq)
- Compute target branch address earlier (e.g., PC addition)

Solution – MIPS

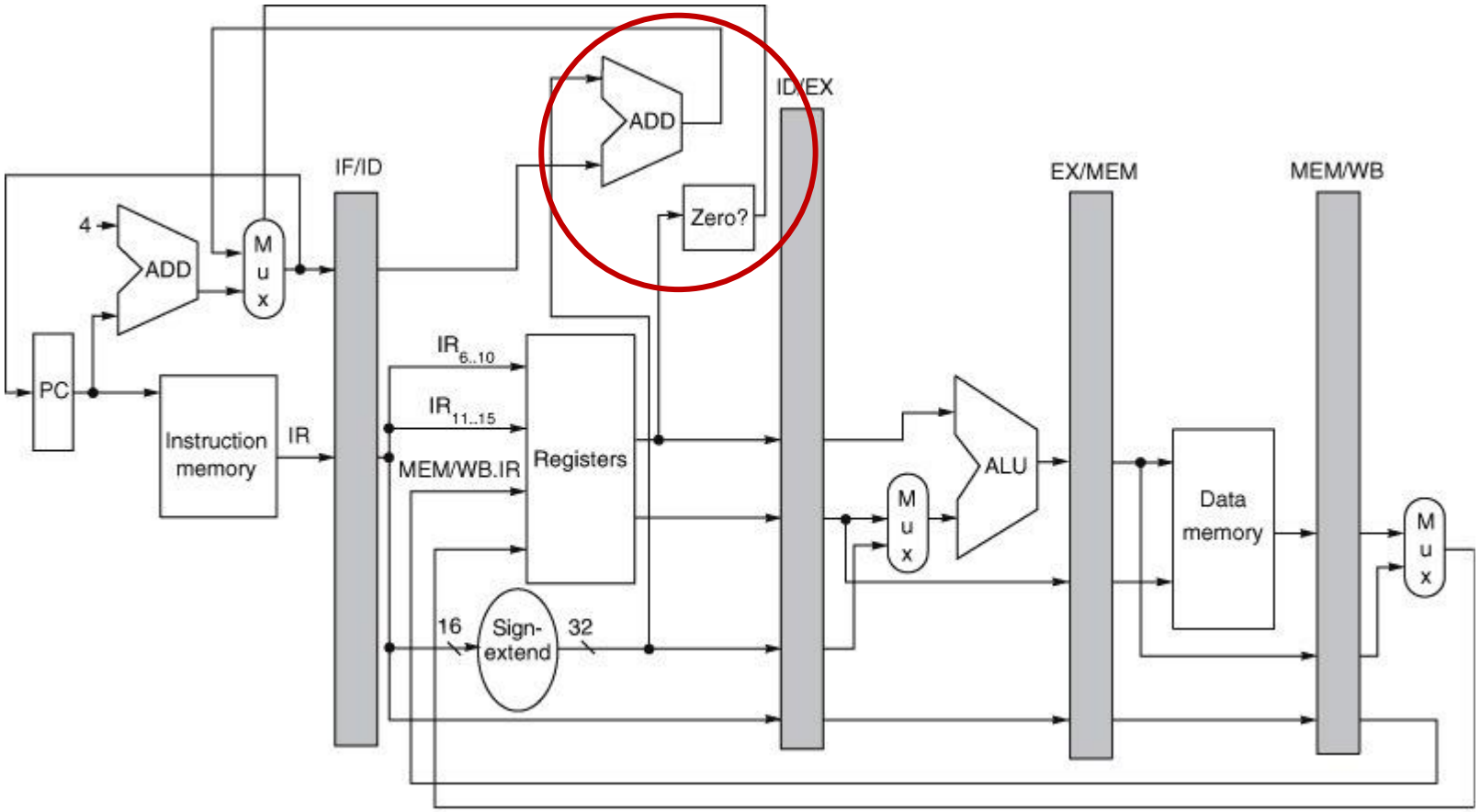
- MIPS branch tests if a register is equal to zero (e.g., beq)
- Move zero test to ID/RF stage
- Introduce adder to calculate new PC in ID/RF stage
- With early branch resolution and kill/stall signals in decode, require 1-cycle per branch, not 3-cycles

Pipelined MIPS Datapath

Figure A.24, Page A-38

Add sufficient logic in decode stage to generate "zero?" signal

Branch is resolved in ID stage instead of EX stage, eliminating one stall cycle



© 2007 Elsevier, Inc. All rights reserved.

Solution 2: Predict Condition

Stall until branch direction is clear

Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch taken
- Advantage: 47% of MIPS branches not taken
- Advantage: PC+4 already calculated for instruction fetch

Predict Branch Taken

- Advantage: 53% of MIPS branches taken
- Disadvantage: Target address not yet calculated, 1-cycle penalty

More sophisticated branch prediction later…

Solution 3: Change ISA Semantics

Delayed Branch

- Change ISA semantics so that instruction following jump/branch always executed.
- Define branch to take place after a following instruction
- Gives compiler flexibility to schedule useful instructions into a branch-induced stall

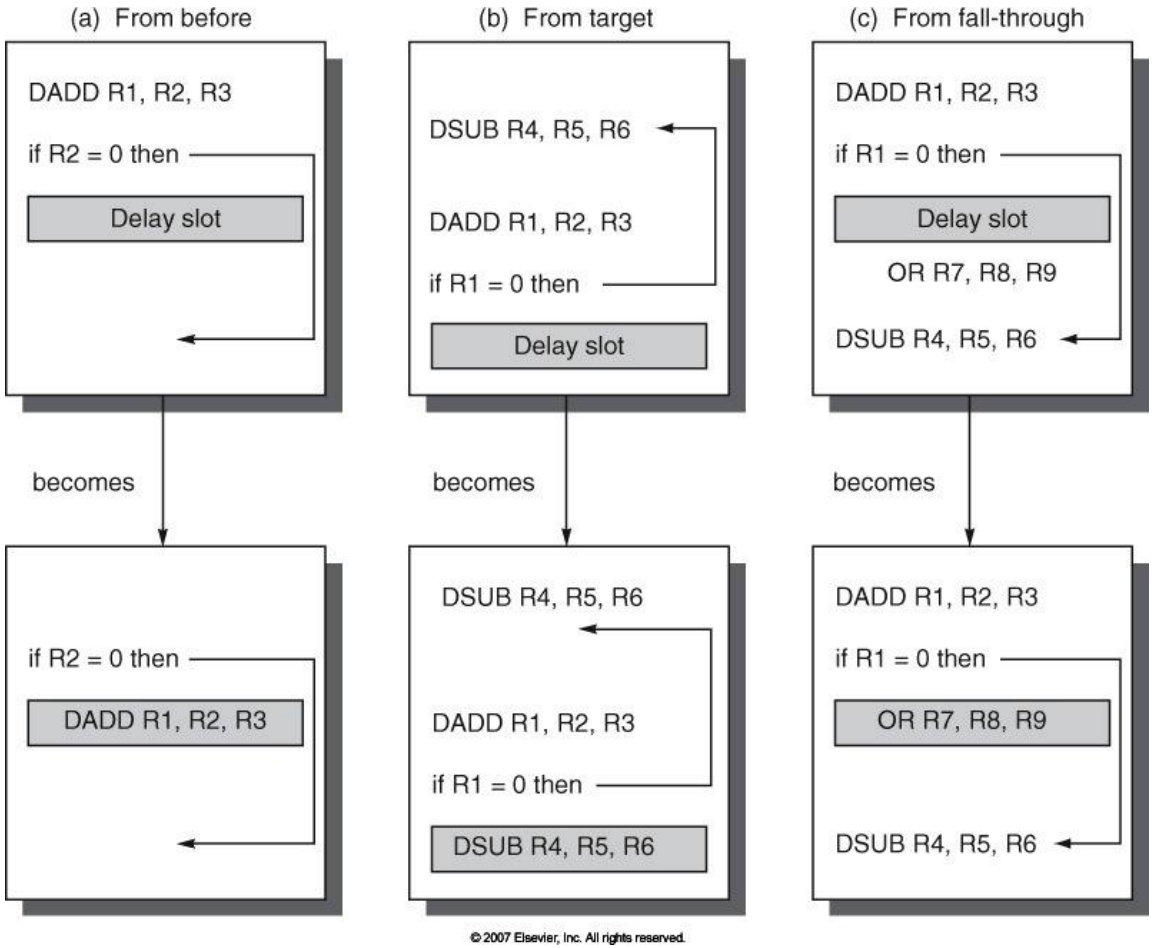
- Branch delay of length n

  Branch instruction
  Sequential successor 1
  Sequential successor 2
  …
  Sequential successor n
  Branch target if taken

- MIPS uses n=1 delay slot to calculate branch outcome, target address

Scheduling Delay Slots

Figure A.24, Page A-38



© 2007 Elsevier, Inc. All rights reserved.

(a) Fills delay slot and reduces instruction count, (b) DSUB needs copying and increases instruction count, (c) OR executes if branch fails so issue speculatively

Delayed Branches

Compiler Effectiveness (n=1 branch delay slot)

- Fill about 60% of branch delay slots
- About 80% of instructions executed are useful computation

Disadvantages of Delayed Branches

- As pipelines deepen, branch delay grows and requires more slots
- Less popular than dynamic approaches (e.g., branch prediction)

Pipelining in Practice

Why is IPC < 1?

Full forwarding may be too expensive to implement

- Implement only frequently used forwarding paths
- Implementing infrequently used forwarding paths might impact length of pipeline stage, increase clock period, and reduce IPC forwarding gains

Multi-cycle Instructions (e.g., loads)

- Instruction following a multi-cycle instruction cannot use its results
- MIPS-I defined load-delay slots, a software-visible pipeline hazard.
- Rely on compiler to schedule useful instructions, nops

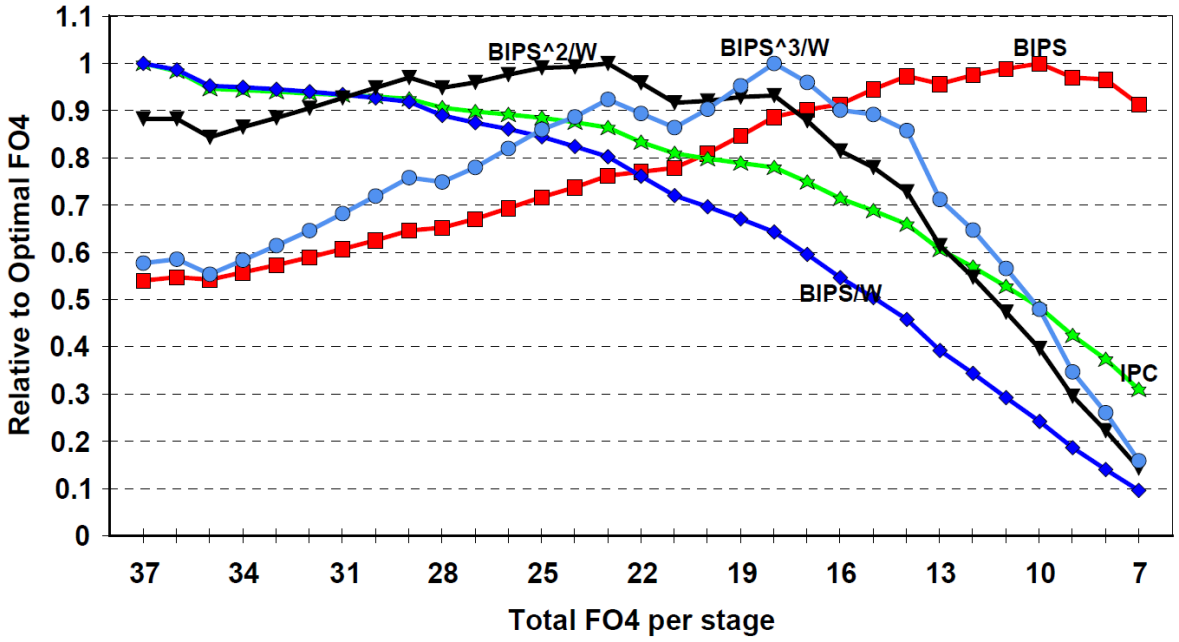
Conditional Branches

- Without delay slots, kill following instructions
- With delay slots, rely on compiler to schedule useful instructions, nops

Optimal Pipeline Depth

- Srinivasan et al. "Optimizing pipelines for power and performance," 2002.
- Performance (BIPS) versus Power (W)

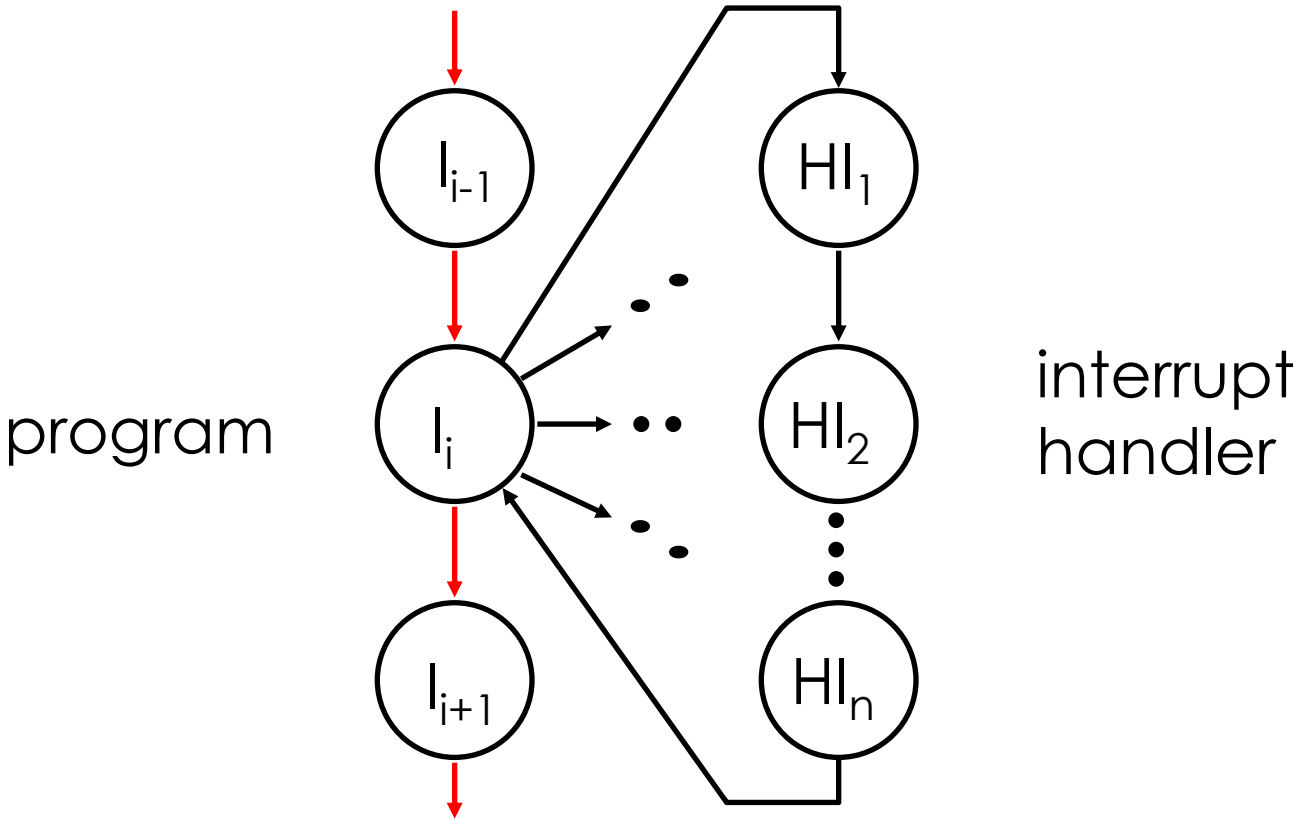
- FO4 is measure of delay: Delay of inverter that is driven by inverter 4x smaller and that is driving inverter 4x larger.
- Quantify amount of logic per pipeline stage in FO4 delays
- (shorter delays → deeper pipelines)



Interrupts and Exceptions

Interrupts alter normal control flow

- Event that needs to be processed by another (system) program
- Event is considered unexpected or rare from program's perspective



Causes of Interrupts

Interrupt

- An event that requests the attention of the processor

Asynchronous Interrupt – External Event

- Input/output device service-request
- Timer expiration
- Power disruptions, hardware failure

Synchronous Interrupt – Internal Event

- Undefined opcode, privileged instruction
- Arithmetic overflow, FPU exception
- Misaligned memory access
- Virtual memory exceptions – page faults, TLB misses, protection violations
- Traps – system calls, jumps into kernel
- Also known as exceptions

Asynchronous Interrupts

Service Request

- An I/O device requests attention by asserting one of the prioritized interrupt request lines

Invoking Interrupt Handler

- Processor decides to process interrupt
- Stops current program at instruction j, completing all instructions up to j-1. Defines a precise interrupt.
- Saves PC of instruction j in a special register (e.g., EPC)
- Disables interrupts and transfers control to designated interrupt handler running in kernel mode.

Interrupt Handler

PC Processing

- Save EPC before re-enabling interrupts, thereby allowing nested interrupts
- Need an instruction to move EPC into general-purpose registers
- Need a way to mask further interrupts until EPC saved

Status Register

- Read status register to determine cause of interrupt
- Executes handler code

Exiting Interrupt Handler

- Use special indirect jump instruction RFE (return-from-exception)
- Enables interrupts
- Restores processor to user mode
- Restores hardware status and control state

Synchronous Interrupts

Exceptions

- A synchronous interrupt (exception) is caused by a particular instruction

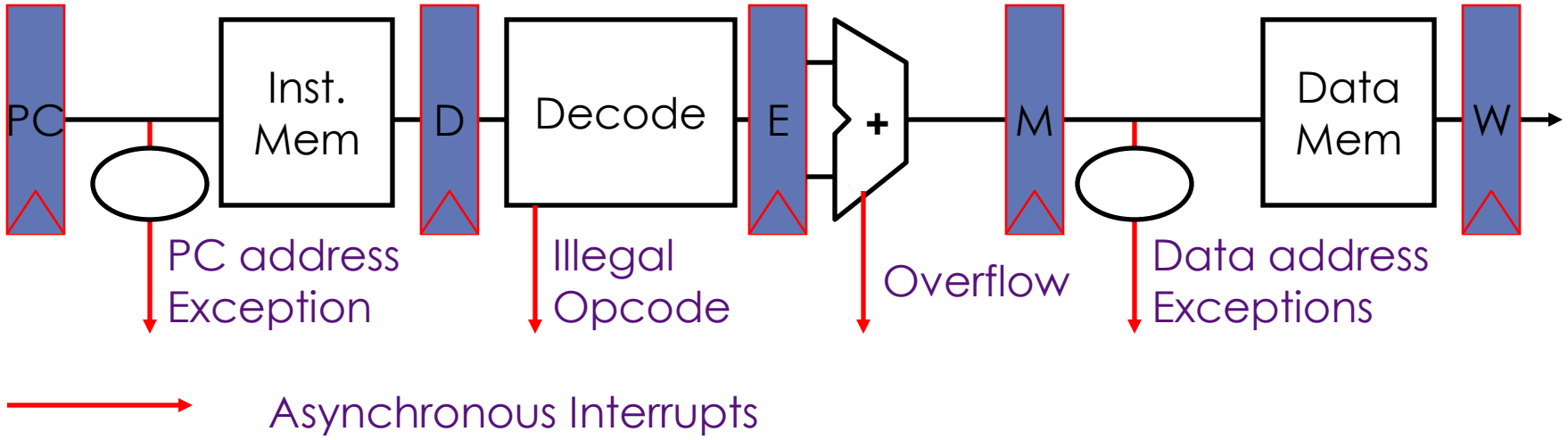
Instruction Re-start

- Generally, instruction cannot be completed
- Instruction needs re-start after exception has been handled
- Processor must undo the effect of partially executed instructions

System Calls

- If the interrupt arises from a system calls (traps), trapping instruction considered complete
- System calls require a special jump instruction and changing into privileged kernel mode

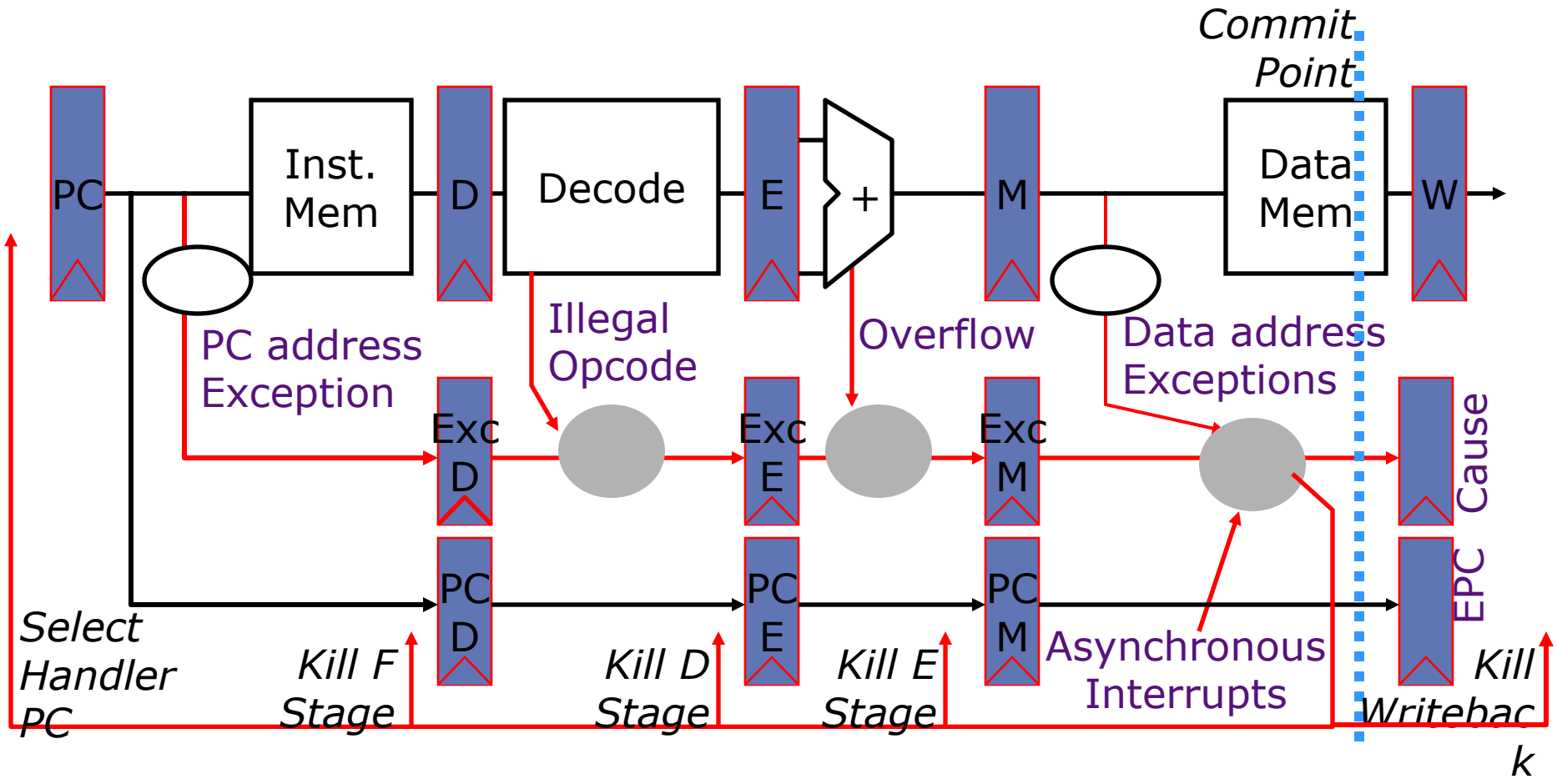
Pipelining and Interrupt Handling



Synchronous: How does the processor handle multiple, simultaneous exceptions in different pipeline stages?

Asynchronous: How does the processor handle external interrupts?

Pipelining and Interrupt Handling



Pipelining and Interrupt Handling

Propagate exception flags through pipeline until commit point

Internal Interrupts

- An instruction might generate multiple exception flags
- For a given instruction, exceptions in earlier pipe stages over-ride those in later pipe stages, thereby prioritizing exceptions earlier in time

Inject external interrupts at commit point

- External interrupts over-ride internal interrupts

Check exception flags at commit point

- If exception flagged, update cause and EPC register
- Kill instructions in all pipeline stages
- Inject handler PC into fetch stage

Speculating about Exceptions

Predict

- Exceptions are rare. Predicting that no exceptions occurred is accurate.

Check Prediction

- Exceptions detected at end of pipeline (commit point). Invoke special hardware for various exception types

Recovery Mechanism

- Architectural state modified at end of pipeline (commit point).
- Discard partially executed instructions after an exception
- Launch exception handler after flushing pipeline

Pipelining and Exceptions

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	nop		*overflow!*		
(I_2) 100: XOR		IF_2	ID_2	EX_2	nop	nop			
(I_3) 104: SUB			IF_3	ID_3	nop	nop	nop		
(I_4) 108: ADD				IF_4	nop	nop	nop	nop	
(I_5) Exc. Handler code					IF_5	ID_5	EX_5	MA_5	WB_5

		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			
Resource	EX			I_1	I_2	nop	nop	I_5		
Usage	MA				I_1	nop	nop	nop	I_5	
	WB					nop	nop	nop	nop	I_5

Acknowledgements

These slides contain material developed and copyright by

- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)