Review

- Last Class:
  - Review of pipeline
  - Single cycle processor → CCT problem
  - Pipelined execution → improved ILP
  - Structural and data hazard
  - Data forwarding
- Today's class:
  - Control hazard
  - Multi-cycle pipelines
- Announcement and reminder
  - Background catch up:
  - Appendix A, B, C in "*Computer Architecture: A Quantitative Approach*"
  - *"Computer Organization and Design - The Hardware/Software Interface"*

Review: Pipelining - What Makes it Hard ?

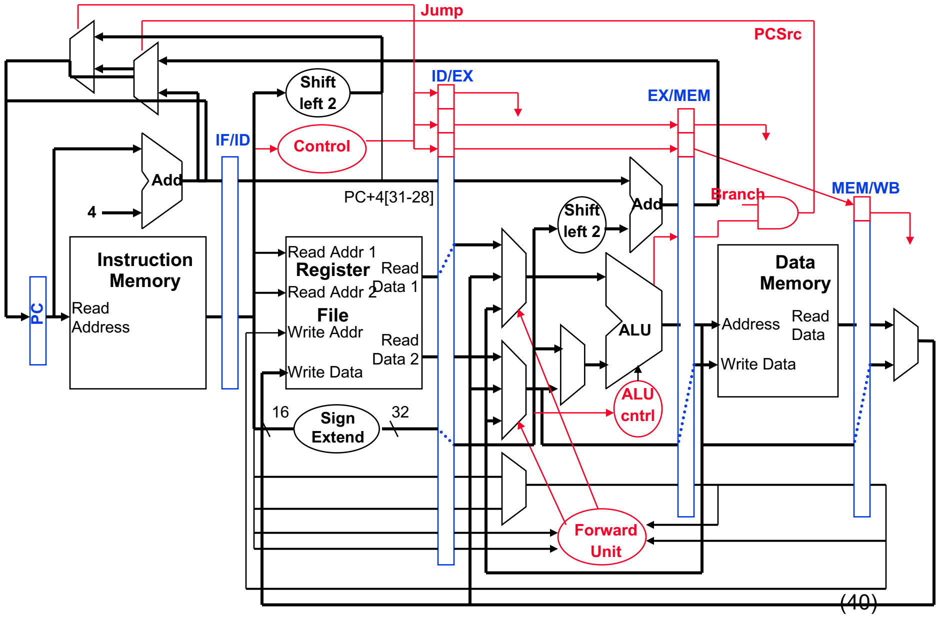
- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

- Pipeline hardware control must detect the hazard and then take action to resolve hazard

Control Hazards

- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is nothing as effective against control hazards as forwarding is for data hazards

Review: Datapath Branch and Jump Hardware

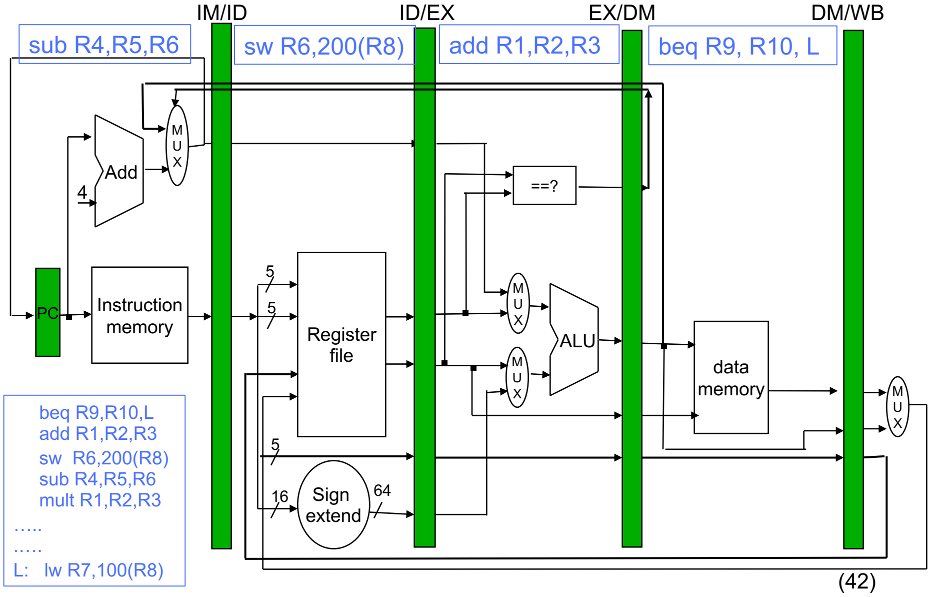


Control (Branch, Jump) Hazards

- What do we need to know?
  - Next instruction target address, maybe sequential (PC+4) *or*
    - » `beq`, PC+4 + branch instruction's sign-extended offset which is computed during EX by the Shift Left 2 – Add logic
    - » `j`, `jal`, constant address field read from IM during IF (26 bits)
    - » `jr`, `jalr`, read from RF during ID (32 bits)
    - » `trap` instruction or exception, obtained from table lookup in the OS (32 bits)
  - Branch decision outcome (ALU zero flag)
    - » continue sequentially? *or* jump to the branch target address?
- When do we need to know it?
  - As early as possible in the pipeline

(41)

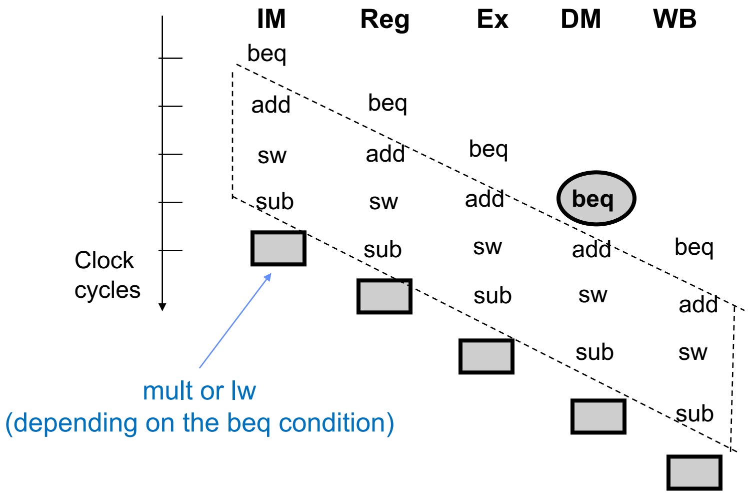
Control Hazard on Branches



```
beq R9,R10,L
add R1,R2,R3
sw  R6,200(R8)
sub R4,R5,R6
mult R1,R2,R3
…..
…..
L:   lw R7,100(R8)
```

(42)

Control Hazard on Branches



Clock cycles	IM	Reg	Ex	DM	WB
	beq				
	add	beq			
	sw	add	beq		
	sub	sw	add	beq	
	▨	sub	sw	add	beq
		▨	sub	sw	add
			▨	sub	sw
				▨	sub
					▨

mult or lw
(depending on the beq condition)

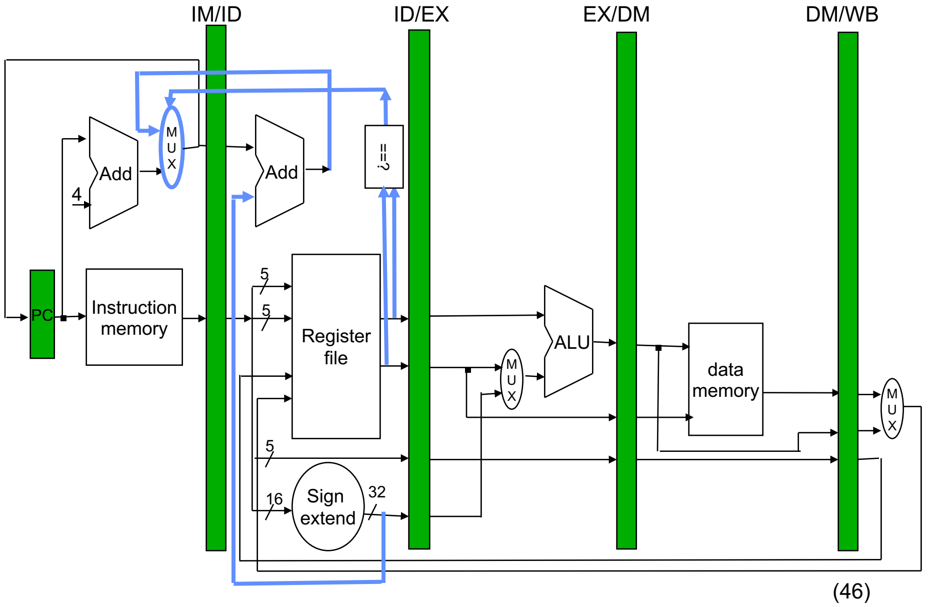
Control (Branch, Jump) Hazards, con’t

- When do we act on the decision?
  - As soon as possible
    - » we decided too late? We already fetched another instruction, and may need to discard it (flush it)
      - Maybe will have to flush more than one instruction?
    - » we guessed, and were right – this is good
    - » we guessed, and were wrong – now need to fix things
  - Guesses require an evaluation of the success rate, by simulation (before the fact) or by measurement (after the fact)
    - » can measurement improve the quality of the guesses?
    - » recent history is often a reasonable predictor of the near future

Branch Stall Impact

- If CPI = 1, 10% branch, Stall 3 cycles => new CPI = 1.3
- Two part solution:
  - Determine branch taken (or not) sooner, and
  - Compute taken branch address earlier
- RISC V branch tests if register = 0
- RISC V Solution:
  - Move Zero test to ID/EX stage
  - Adder to calculate new PC in ID/EX stage
  - 1 clock cycle penalty for branch versus 3

Early determination of Branch condition and target



Four Branch Hazard Alternatives

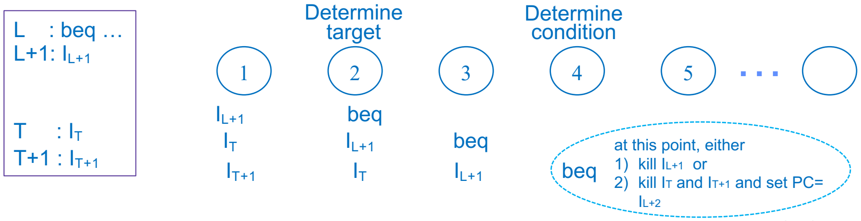
#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence (PC+4 already calculated)
- “Squash” instructions in pipeline if branch actually taken (how?)
- Advantage of late pipeline state update

#3: Predict Branch Taken

- More than 50% of MIPS branches are taken on average
- Start fetching from the branch target as soon as it is available
- Useful if target address is computed earlier than the branch condition (for example in stage 2 and stage 4, respectively, of a deep pipeline).



L : beq …
L+1: I_{L+1}

T : I_T
T+1 : I_{T+1}

Determine target

Determine condition

at this point, either
1) kill I_{L+1} or
2) kill I_T and I_{T+1} and set PC= I_{L+2}

(47)

Four Branch Hazard Alternatives

#4: Delayed Branch

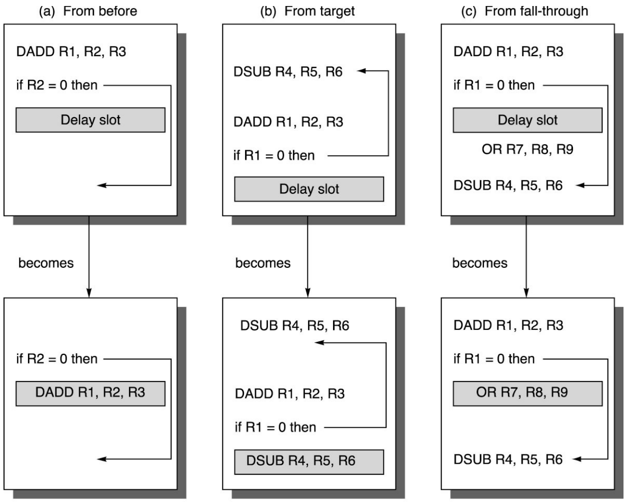
- Change semantics such that branching takes place AFTER the *n* instructions following the branch execute

```
branch instruction
  sequential successor₁
  sequential successor₂
  ........
  sequential successorₙ
branch target if taken
```

Branch delay of length *n*

- One slot delay in the 5-stage pipeline if branch condition and target are resolved in the ID stage.

(48)



(a) From before

DADD R1, R2, R3
if R2 = 0 then
Delay slot

becomes

if R2 = 0 then
DADD R1, R2, R3

(b) From target

DSUB R4, R5, R6
DADD R1, R2, R3
if R1 = 0 then
Delay slot

becomes

DSUB R4, R5, R6
DADD R1, R2, R3
if R1 = 0 then
DSUB R4, R5, R6

(c) From fall-through

DADD R1, R2, R3
if R1 = 0 then
Delay slot
OR R7, R8, R9
DSUB R4, R5, R6

becomes

DADD R1, R2, R3
if R1 = 0 then
OR R7, R8, R9
DSUB R4, R5, R6

- Where do we get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: should repeat instruction for correctness
  - From fall through: correct if R7 is dead after the branch

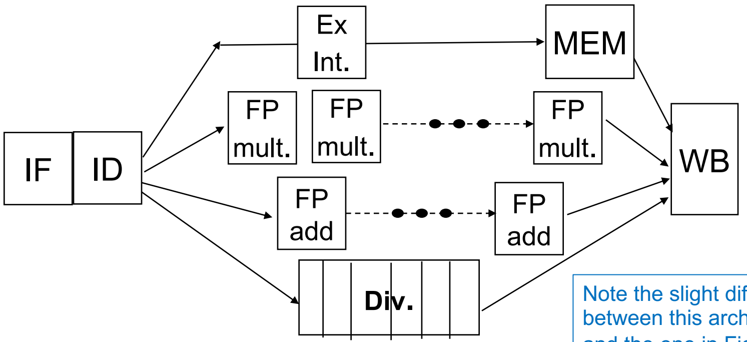
Evaluating Branch Alternatives

When accounting for branch hazards
CPI = 1 + Hazard frequency * Hazard penalty

- **Example**: Assume a pipeline in which the target address is known in stage 3 and the branch condition is known in stage 4. Compare the CPI for the following methods:
  - flush/stall
  - predict taken
  - predict not taken
  - delayed branch

  Assume that you know the percentages of branch instructions, and the probabilities for branch taken/not taken.
- Compilers technology can be used to increase efficiency of code if it knows branch probabilities (static branch prediction - profiling)

Multi-cycle pipelines (Section C.5)



Note the slight difference between this architecture and the one in Figure C.30 of the textbook

- **Assume**
  - 4-stage, pipelined, FP add
  - 7-stage, pipelined, FP multiply
  - 25 stage, non-pipelined divide unit
- **Latency** of an instruction, *I*, in a pipeline, *P*, is the number of bubbles that has to exist in *P* if the instruction following *I* wants to use the result of *I*.

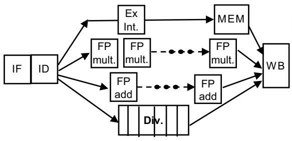
Hazards:

- Two divide instructions will stall the pipe (structural hazards).
- May have more than one register write in one cycle (why?)
  - increase number of ports, or stall the pipeline (interlock)
- May have WAW hazard (why?)
- Out-of-order completion causes problems with exceptions,
- The long pipes causes more RAW hazards (why?)

To deal with structural Hazards and resolve competition for the WB stage:

- Stall a conflicting instruction when entering the WB stage
  - may give priority to longer instructions to reduce RAW hazards.

Examples



Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
fld F4, 0(R2)	IF	ID	EX	MEM	WB												
fmul.d F0, F4, F6		IF	ID	ID	M1	M2	M3	M4	M5	M6	M7	WB					
fadd.d F2, F0, F8			IF	IF	ID	ID	ID	ID	ID	ID	ID	A1	A2	A3	A4	WB	
fsd F2, 0(R2)					IF	IF	IF	IF	IF	IF	IF	ID	EX	EX	EX	MEM	

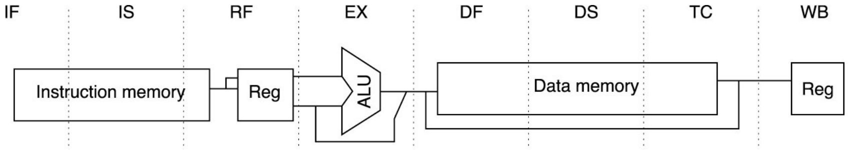
Stall due to RAW hazards (assuming forwarding)

instruction	1	2	3	4	5	6	7	8	9	10	11
fmul.d F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	WB	
…		IF	ID	EX	Mem	WB					
…			IF	ID	EX	Mem	WB				
fadd.d F2,F4,F6				IF	ID	A1	A2	A3	A4	WB	
…					IF	ID	EX	Mem	WB		
fld F2, 0(R2)						IF	ID	EX	Mem	WB	
							IF	ID	Ex	Mem	WB

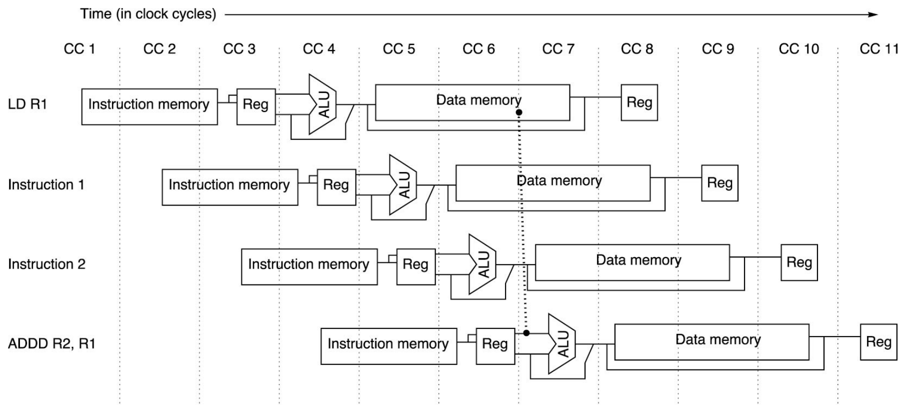
Structural hazards

The MIPS R4000 (Section C.6)

- 64-bit instruction set (MIPS-3 ISA)
- Decomposes memory access into stages (super-pipelining)



- Uses a deeper pipeline
  - IF -- first half of instruction fetch
  - IS -- second half of instruction fetch
  - RF - instruction decode and register fetch (and check cache tag)
  - EX - execution: effective address calculation, ALU operation, branch target computation, branch condition evaluation
  - DF - first half of data fetch
  - DS - second half of data fetch
  - TC - check cache tag (to determine if it is a hit)
  - WB write back



- Forwarding from data memory output is required to instructions that are 3 or 4 cycles later (sooner instructions have to stall)
- Can speculatively forward from data memory (lw R1) to ALU (add R2, R1) before checking the tag to make sure that it is a hit.

(55)

Dealing with Exceptions

- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and “deals” with it

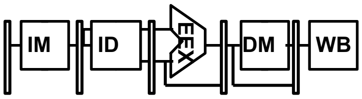
(57)

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
  - simply suspend and resume user program
- Traps (Exception) – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so must stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

(58)

Where in the Pipeline Exceptions Occur

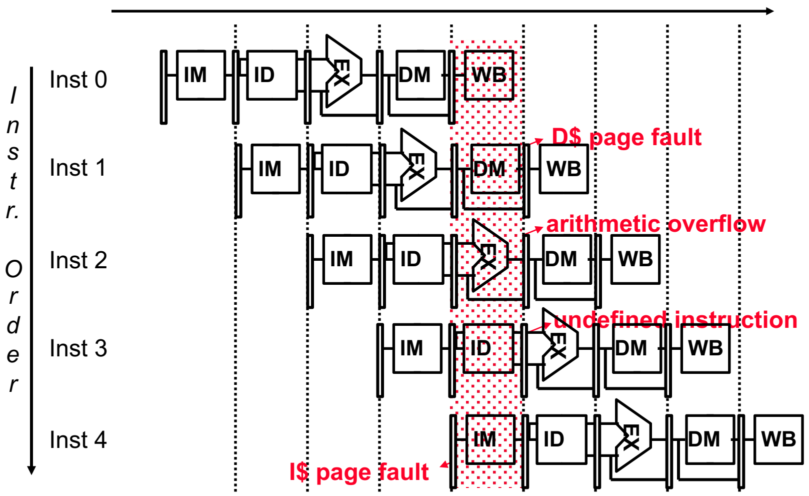


	Stage(s)?	Synchronous?
• Arithmetic overflow	EX	yes
• Undefined instruction	ID	yes
• TLB or page fault	IF, MEM	yes
• I/O service request	any	no
• Hardware malfunction	any	no

Be aware that multiple exceptions can occur simultaneously in a *single* clock cycle!

(59)

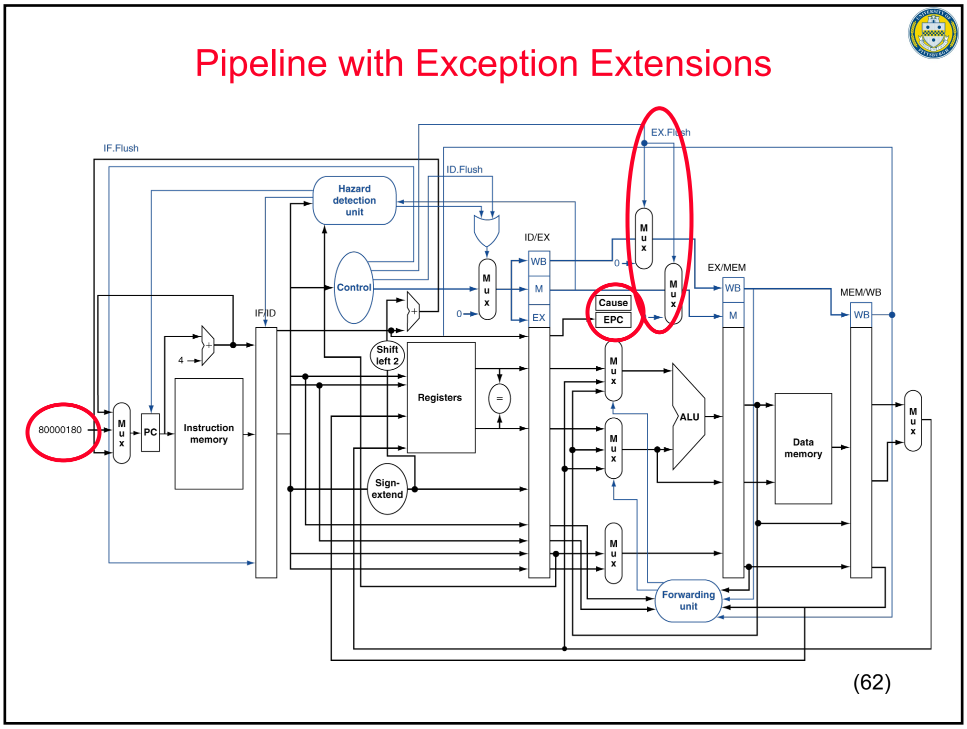
Multiple Simultaneous Exceptions



- ❑ Hardware sorts the exceptions so that the earliest instruction (D$ page fault) is the one interrupted first

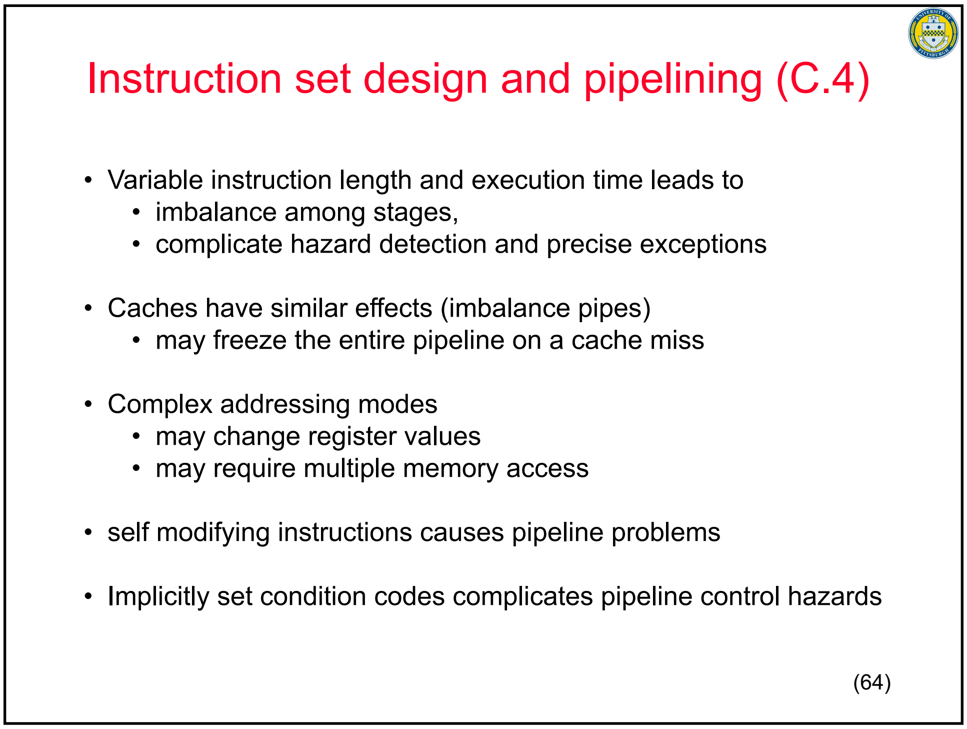
Additions to MIPS to Handle Exceptions

- Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (`CauseWrite`)
- EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (`EPCWrite`)
  - Exception software must match exception to instruction
- A way to load the PC with the address of the exception handler
  - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180$_{hex}$ for arithmetic overflow, 8000 0000$_{hex}$ for undefined instruction)
- A way to flush offending instruction and the ones that follow it



Pipeline with Exception Extensions

(62)



Instruction set design and pipelining (C.4)

- Variable instruction length and execution time leads to
  - imbalance among stages,
  - complicate hazard detection and precise exceptions
- Caches have similar effects (imbalance pipes)
  - may freeze the entire pipeline on a cache miss
- Complex addressing modes
  - may change register values
  - may require multiple memory access
- self modifying instructions causes pipeline problems
- Implicitly set condition codes complicates pipeline control hazards

(64)