Multi-cycle Datapath

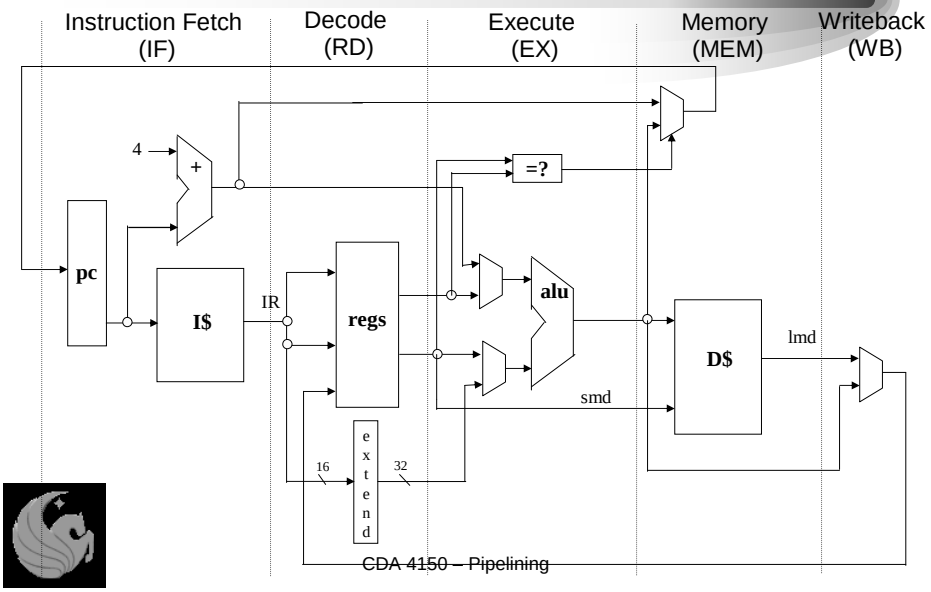
Single-Cycle implementation has poor performance
- Cycle time longer than necessary for all but slowest instruction

Solution: break the instruction into smaller steps
- Execute each step in one clock cycle
- Cycle time: time it takes to execute the longest step
- Design all the steps to have similar length

Advantages of the multiple cycle processor
- Cycle time is much shorter
- Functional units can be used > once/instruction (less HW)

Disadvantages of the multiple cycle processor
- More timing paths to analyze and tune
- Additional registers to store intermediate data values

A Simple Datapath



Execution Time

Time taken for 1 instruction

- Add up the execution times of each phase
- Each phase may take different amounts of time
- One instruction executes at a time

Example

- Pick some execution times out of the air
- t_{fetch}=60ns, t_{decode}=30ns, t_{exec}=50ns, t_{mem}=80ns, t_{wb}=20ns
- Total execution time per instruction = 240ns

Pipelining

We can execute multiple instructions at the same time!

Each instruction will be in a different phase of execution

Throughput will increase by the number of pipeline stages

Overlap different steps for consecutive instructions

- Steps are called *pipeline stages*
- Need latches after each stage to hold control/data for later stages

A new instruction enters the pipeline at IF on each clock

- Takes 5 clocks to complete execution and leave the pipeline
- Potential throughput of 1 CPI

Pipeline Diagram

Instruction	Clock Cycle =>							
I	IF_I	RD_I	EX_I	MEM_I	WB_I			
I+1		IF_{I+1}	RD_{I+1}	EX_{I+1}	MEM_{I+1}	WB_{I+1}		
I+2			IF_{I+2}	RD_{I+2}	EX_{I+2}	MEM_{I+2}	WB_{I+2}	
I+3				IF_{I+3}	RD_{I+3}	EX_{I+3}	MEM_{I+3}	WB_{I+3}
I+4					IF_{I+4}	RD_{I+4}	EX_{I+4}	MEM_{I+4}

Like assembly lines in manufacturing

Execution Time

Pipeline stages execute in parallel

- Must wait for slowest one to finish

Pipeline overhead

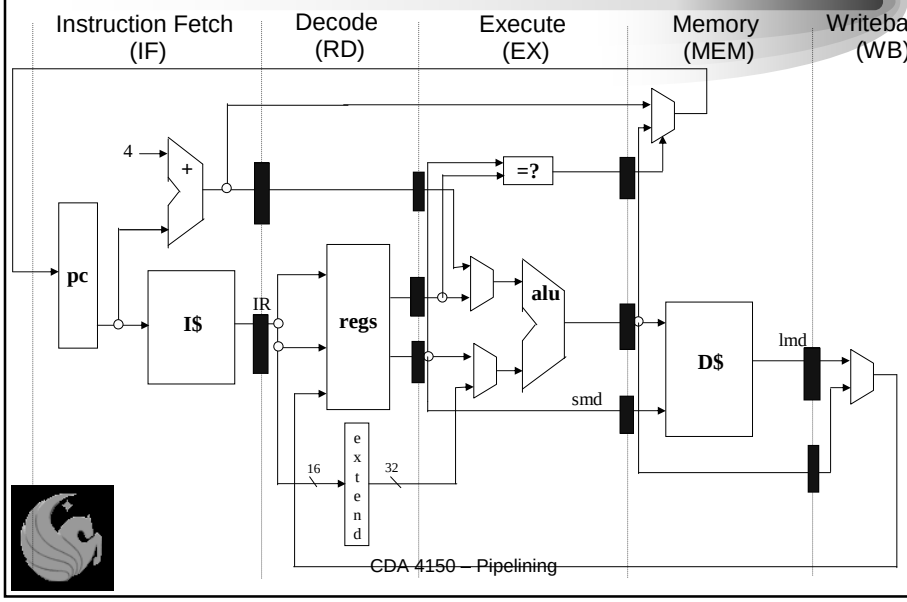
- Introducing pipelining registers adds latency
- Let's assume the overhead is 5ns

Our example

- t_{fetch}=60ns, t_{decode}=30ns, t_{exec}=50ns, t_{mem}=80ns, t_{wb}=20ns
- Longest state is 80ns + 5ns = 85ns
- Instruction executes in 5*85= 425ns
- But, we execute different parts of 5 instructions at same time!

At peak throughput, 1 instruction every 85ns

A Pipelined Datapath



Pipeline Hazards

The major hurdle of pipelining

- Situations where next instruction cannot execute
- Reduce the performance of pipelining

Speedup = Pipeline depth/(1 + pipeline stalls/inst)

Want incredibly long pipelines, with no pipeline stalls

Good luck!

Long pipes increase likelihood of hazards

- Let's look at pipeline resources used by instruction class

Pipeline Resources

Pipe stage	ALU	Memory	Branch
IF	Fetch-PC Inst Cache	Fetch-PC Inst Cache	Fetch-PC Inst Cache
RD	Register Read	Register Read	Register Read
EX	ALU	ALU (address)	ALU (dest addr) Compare logic Fetch-PC (taken)
MEM	N/A	Cache Tags Cache Data	N/A
WB	PC Register Write	PC Register Write (Load)	PC

Types of Hazards

Three classes of hazards

- Data hazards
  - One instruction has a source operand that is the result of a previous instruction in the pipeline (Read-After Write: RAW)
  - There are other types of data hazards (later)
- Control hazards
  - The execution of an instruction depends on the resolution of a previous branch instruction in the pipeline
  - Becomes a big problem with deep pipelines
- Structural hazards
  - Two or more Instructions in the pipeline require the same hardware resource to progress
  - Most common instance is non-pipelined FU (multiplier)

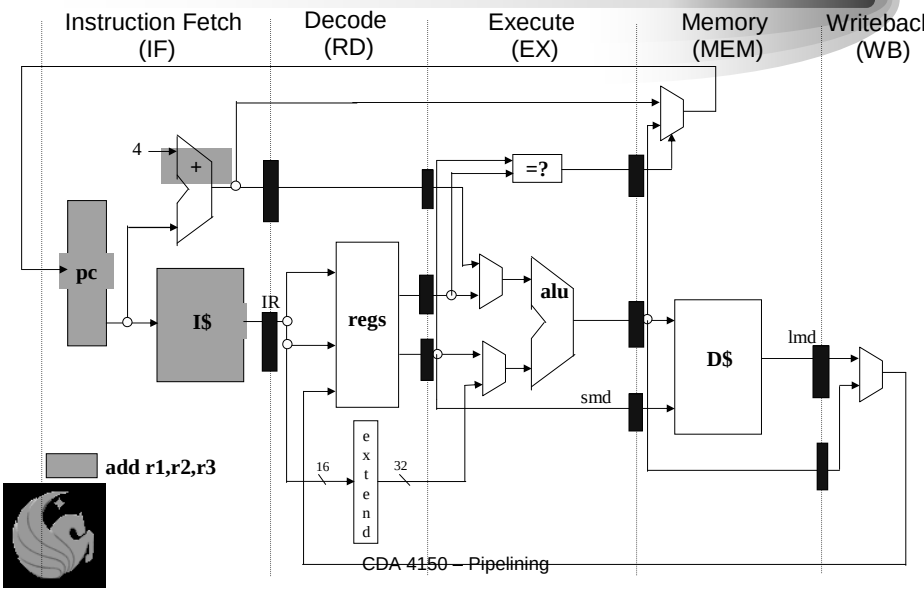
Data Hazards

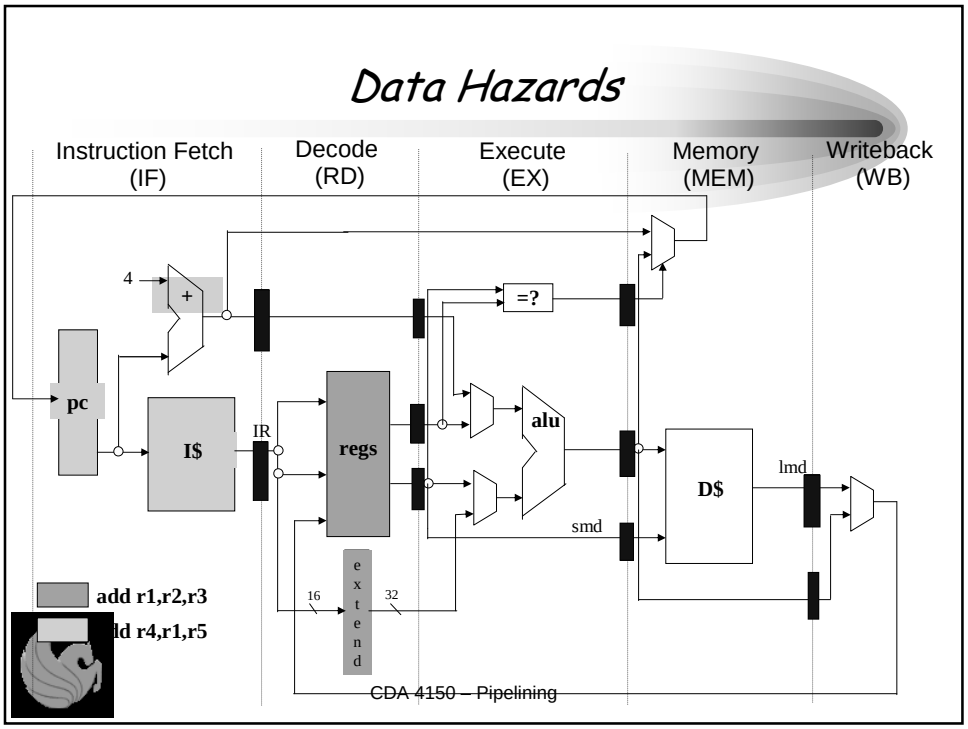
In MIPS R3000 pipeline, a data dependency occurs when an instruction's source register is the destination register for either of the 2 prior instructions

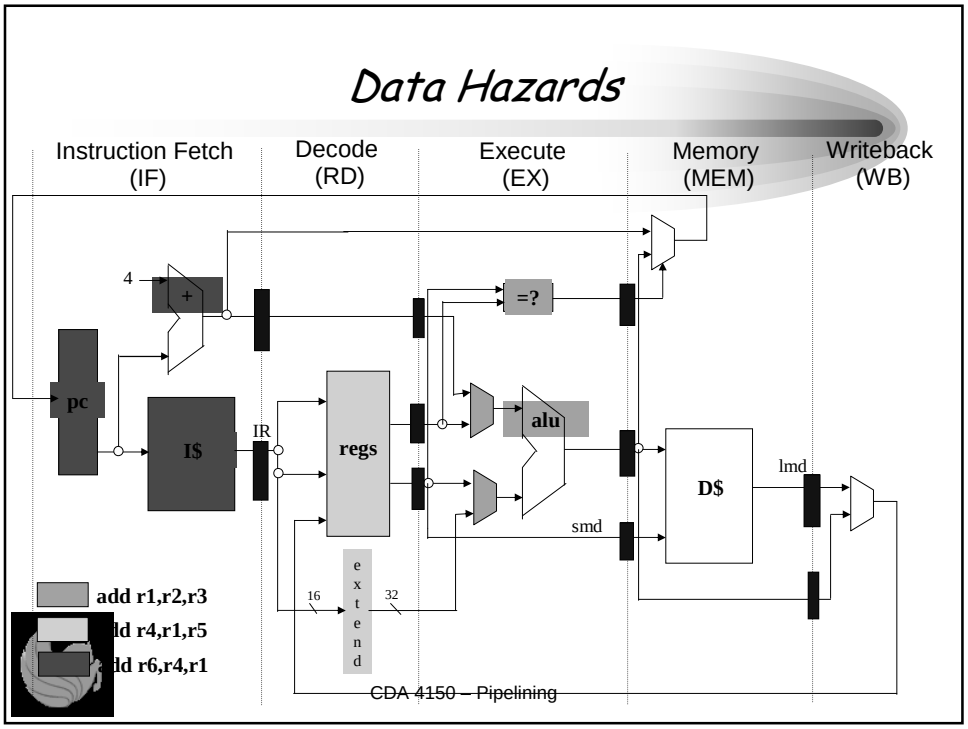
- The simplest way to handle this is to stall the dependent instruction at RD until the required register has been written back
- This would cause a 2-clock delay when the instructions are consecutive

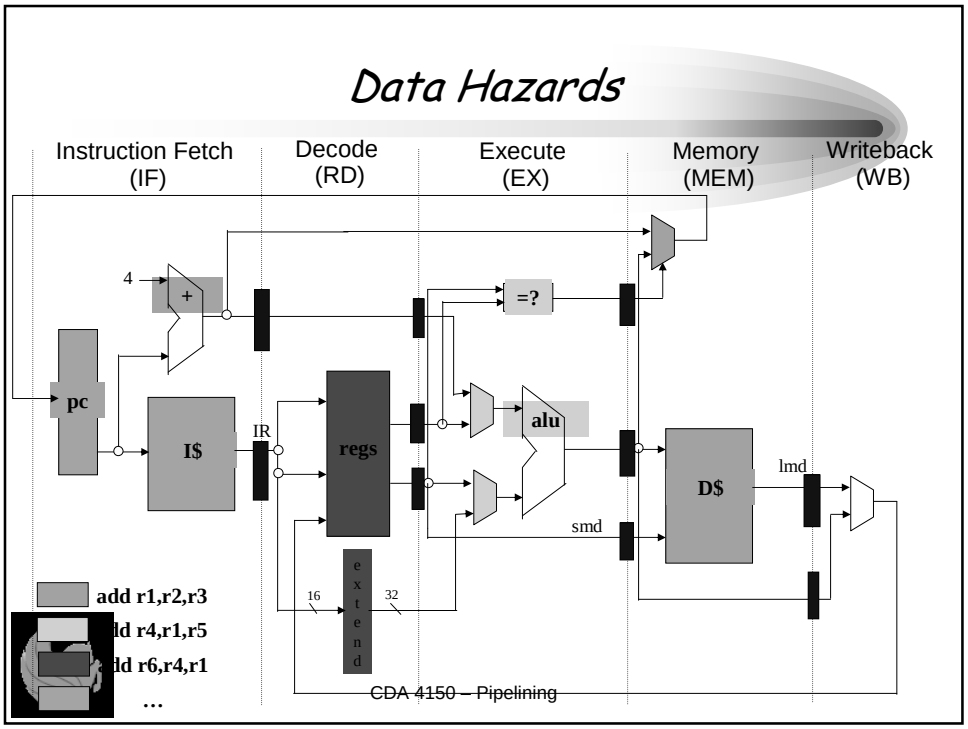
Instruction	Clock Cycle =>							
`add $r3,$r1,$r2`	IF	RD	EX	MEM	WB			
`sub $r5,$r3,$r4`		IF	RD	RD	RD	EX	MEM	WB

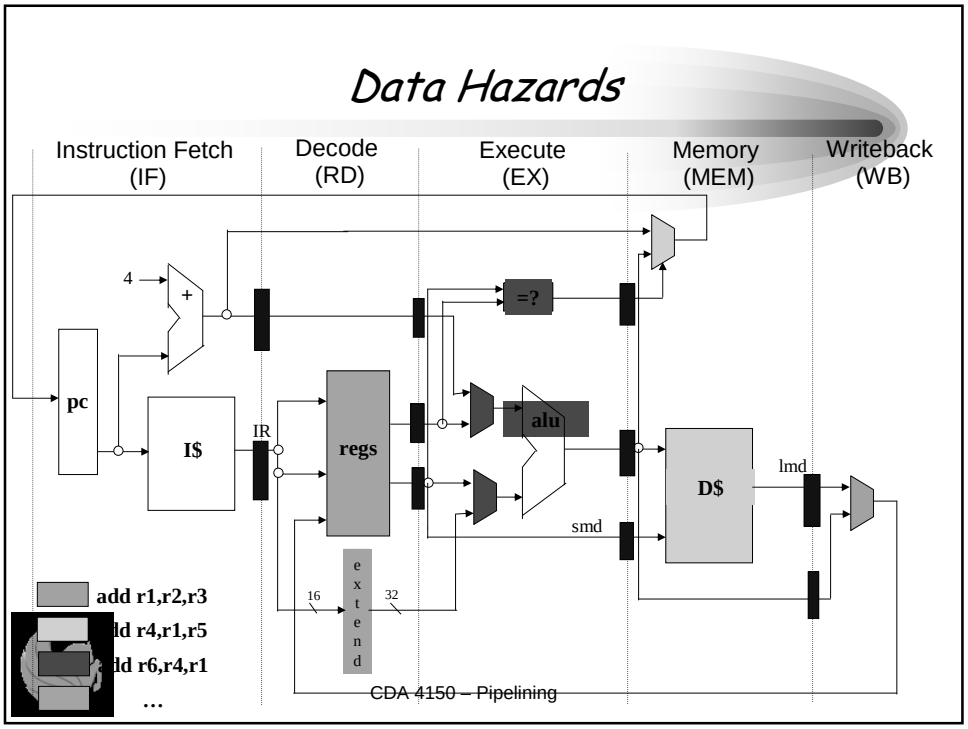
Data Hazards




Data Hazards
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
add r1,r2,r3
ld r4,r1,r5
CDA 4150 – Pipelining


Data Hazards
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
add r1,r2,r3
ld r4,r1,r5
ld r6,r4,r1
CDA 4150 – Pipelining


Data Hazards
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
add r1,r2,r3
ld r4,r1,r5
dd r6,r4,r1
…
CDA 4150 – Pipelining


Data Hazards
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
add r1,r2,r3
ld r4,r1,r5
dd r6,r4,r1
…
CDA 4150 – Pipelining

Bypassing

Performance can be improved by *forwarding* (*bypassing*) a result from a later stage to an earlier stage

- The result of an ALU instruction is known at the end of EX
- The result of a Load instruction is known at the end of MEM

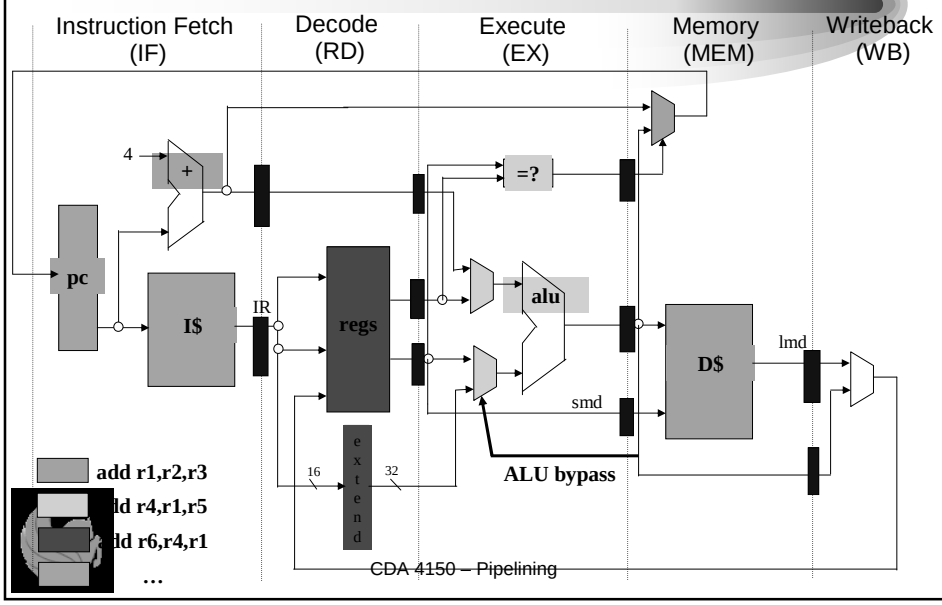
There is no delay when an ALU instruction executes

There is 1 clock delay when a Load instruction is directly followed by a dependent instruction

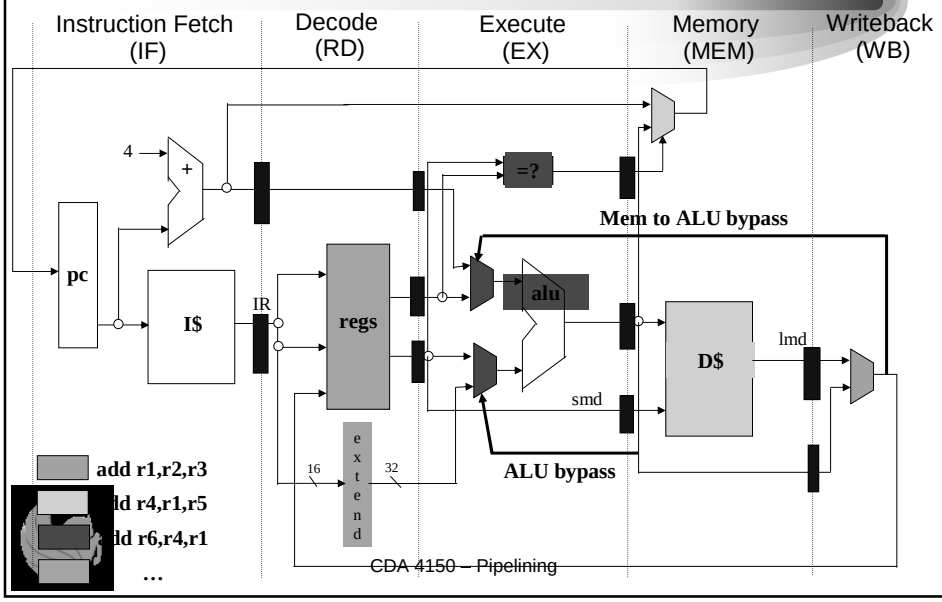
- The Load instruction is said to have a *latency* of 2 clocks

Instruction	Clock Cycle =>								
add $t3,$t1,$t2	IF	RD	EX	MEM	WB				
sub $t5,$t3,$t4		IF	RD	EX	MEM	WB			
lw $s1,0($t3)			IF	RD	EX	MEM	WB		
addi $s2,$s1,1				IF	RD	RD	EX	MEM	WB

ALU Bypass



Mem to ALU Bypass



Control Hazards

When a branch instruction is executed, execution of subsequent instructions depends on whether the branch is taken and the location of the destination

A simple, but effective approach is to assume the branch is not taken and follow the sequential path

The branch is resolved at the end of EX

- If taken, cancel instructions in the sequential path and start fetching from the destination on the next clock
  - this results in a 2-clock delay for taken branches
- If not taken, continue sequentially

Instruction		Clock Cycle =>				
	I_1	IF	RD	EX	MEM	WB
	beq $t0,$t1,L1		IF	RD	EX	MEM	WB
	I_3			IF	RD	--	--
	I_4				IF	--	--
L1:	I_5					IF	RD

ISA Considerations with Pipelining

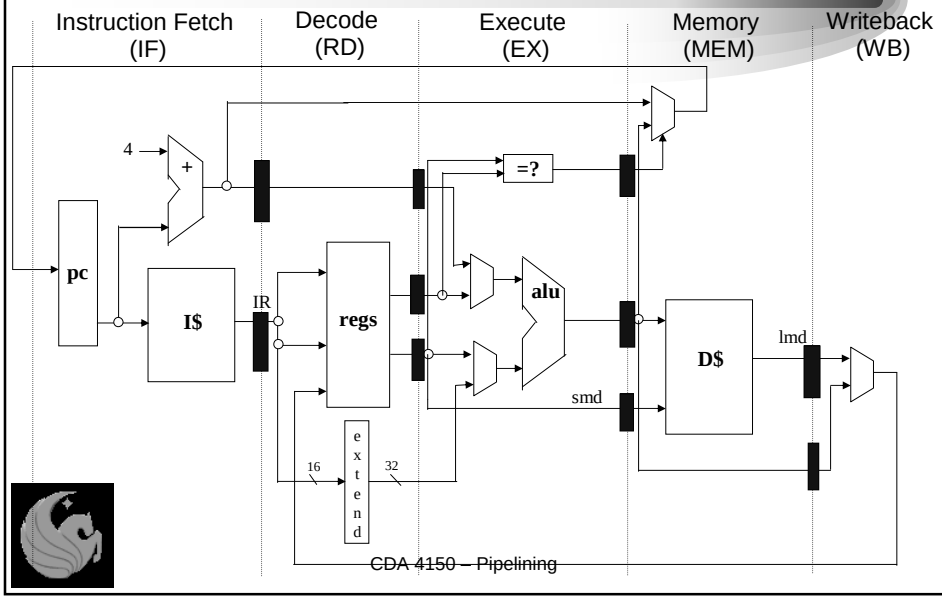
Load Delay

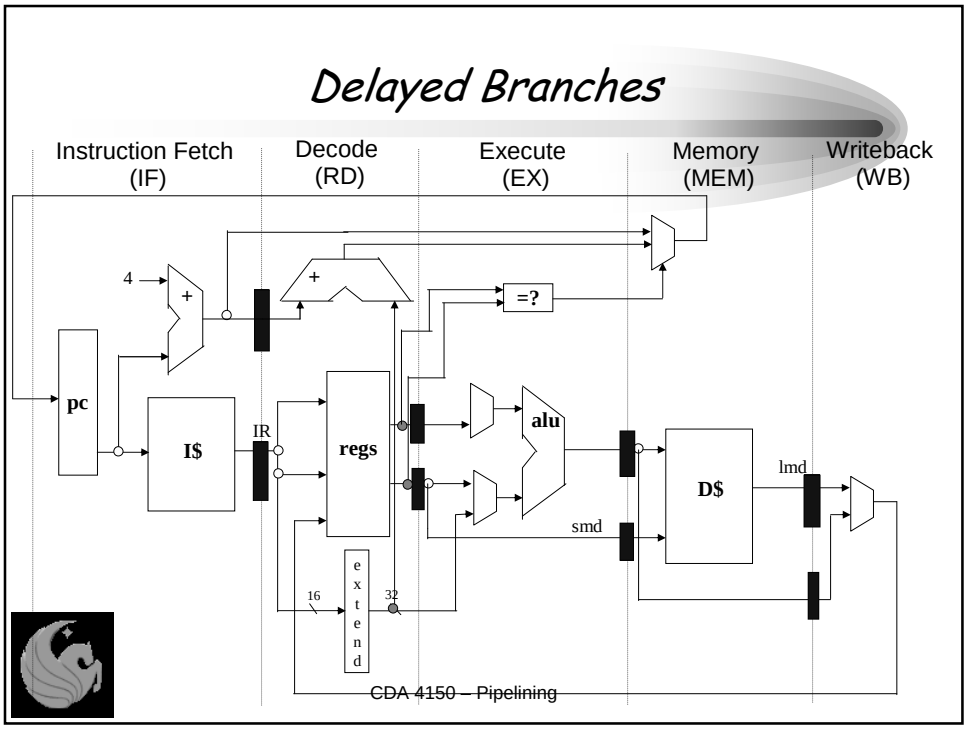
- Explicit 1-instruction delay in MIPS ISA
  - If no instruction can be scheduled following the load, nop required
    - MIPS == "Microprocessor without Interlocked Pipeline Stages
  - But other implementations may have different load delays!

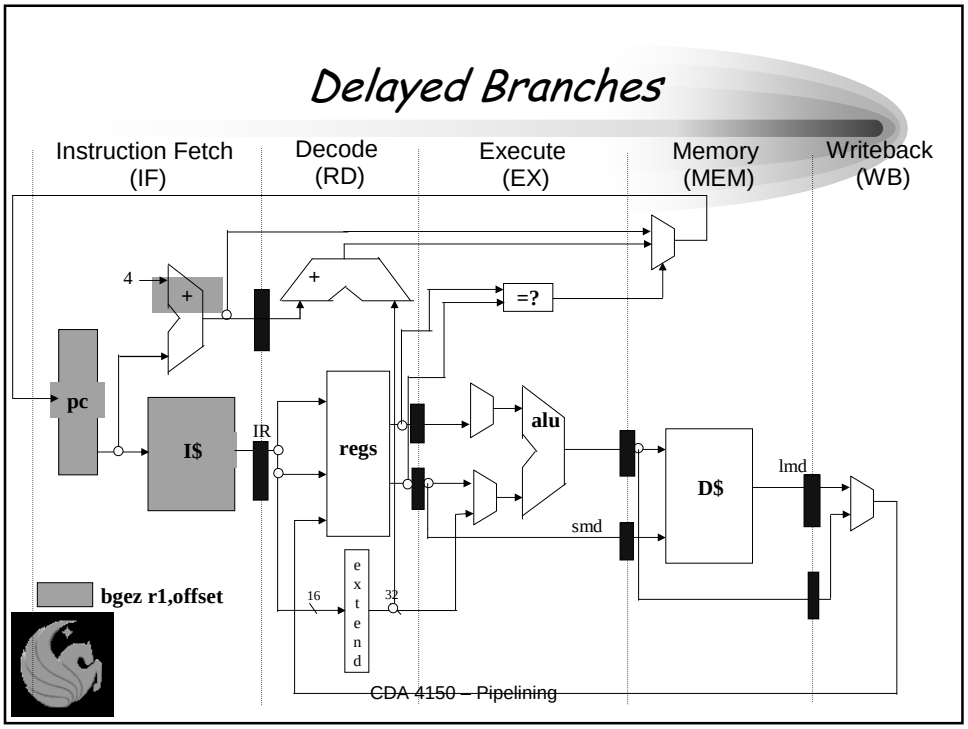
Branch Delay

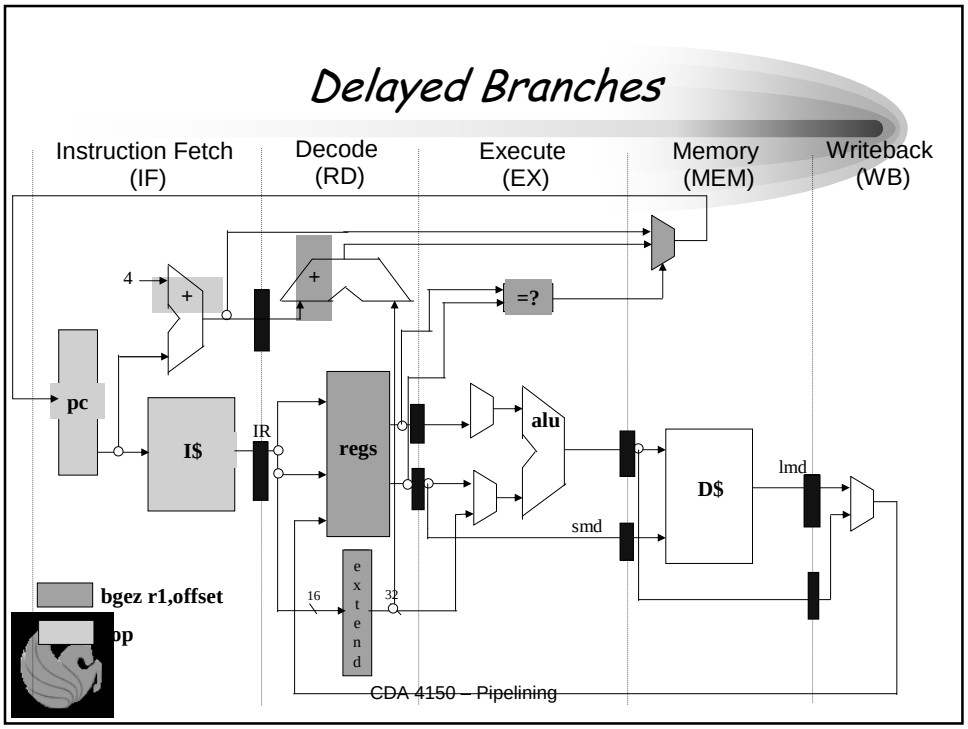
- Explicit 1-instruction delay in MIPS, HP-PA, SPARC
  - For MIPS, if no instruction can be scheduled, NOP required
    - Scheduled instruction must be safe to execute whether or not branch is taken (assembler schedules)
  - For HP-PA/ SPARC the instruction following the branch is conditionally executed or *squashed*

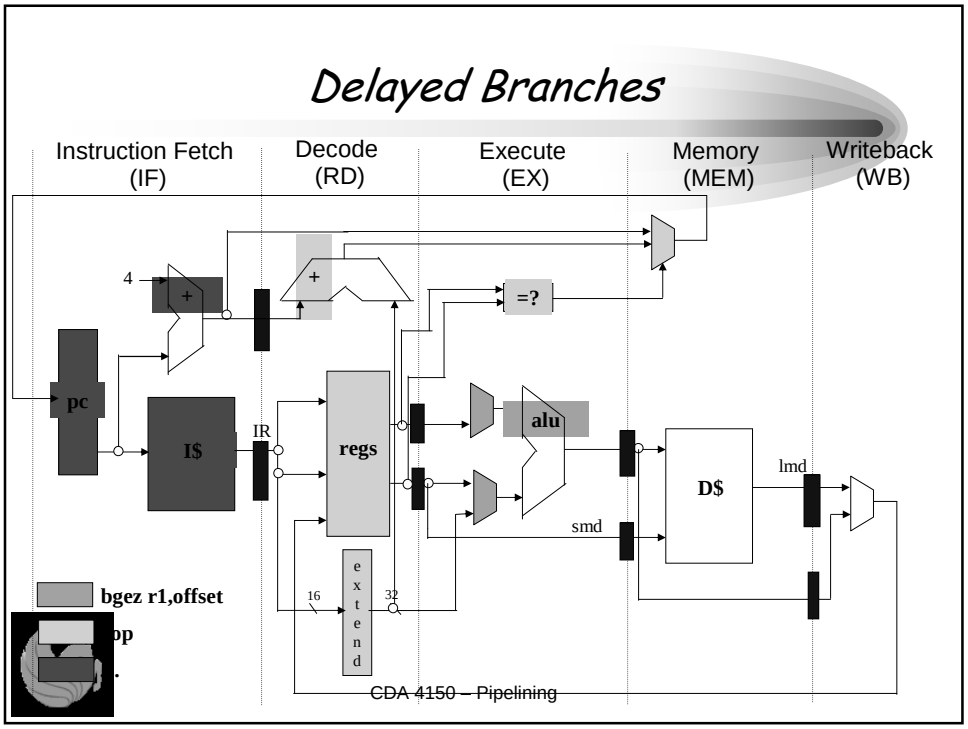
Delayed Branches




Delayed Branches
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
CDA 4150 – Pipelining


Delayed Branches
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
bgez r1,offset
CDA 4150 – Pipelining


Delayed Branches
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
bgez r1,offset
op
CDA 4150 – Pipelining


Delayed Branches
Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)
4
+
+
=?
pc
I$
IR
regs
alu
D$
lmd
smd
16
32
extend
bgez r1,offset
op
CDA 4150 – Pipelining

Structural Hazards

Non-pipelined, multi-cycle functional units
- Integer multiply, divide

Can also have structural hazards on data cache
- Loads access tags/data in MEM
- Stores access tags in MEM, data in WB
- What if a load follows a store?

Structural hazards are detected in decode and stalled there

Only way to remove them is to add functional units
- Or pipeline them
- Or dual port them (caches)

Next Time

More complicated (deeper) pipelines

Data hazards revisited

Code scheduling for pipelines

What makes pipelining hard
- Interrupts
- Precise exceptions
- Branches and long pipes