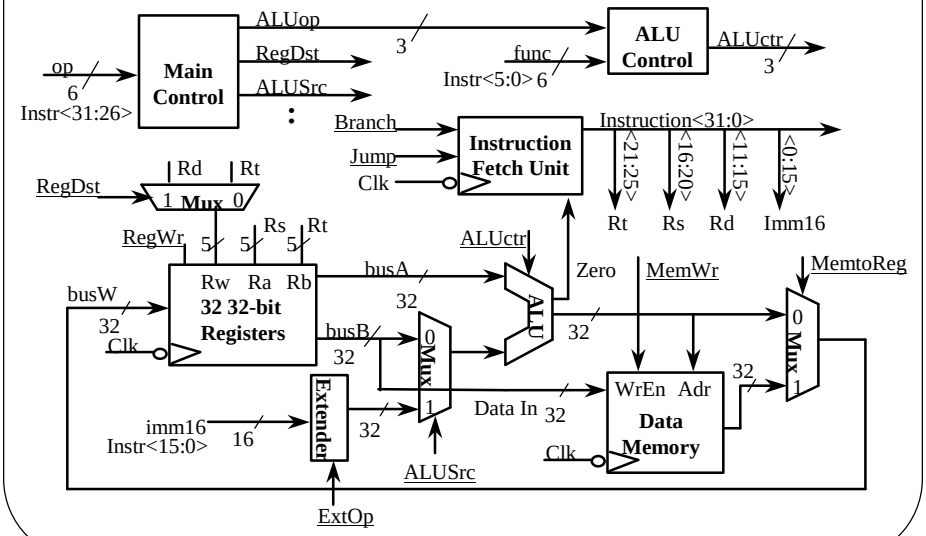
Lecture 4: Pipeline Complications: Data and Control Hazards

Professor Alvin R. Lebeck
Computer Science 220
Fall 2001

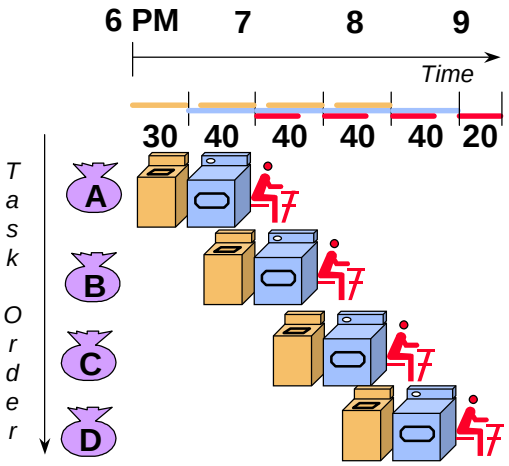
Administrative

- **Homework #1 Due Tuesday, September 11**
- **Start Reading Chapter 4**
- **Projects**

Review: A Single Cycle Processor

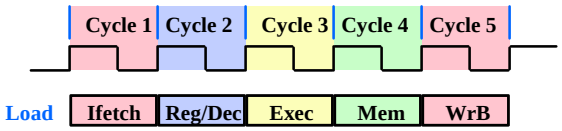


Review: Pipelining Lessons



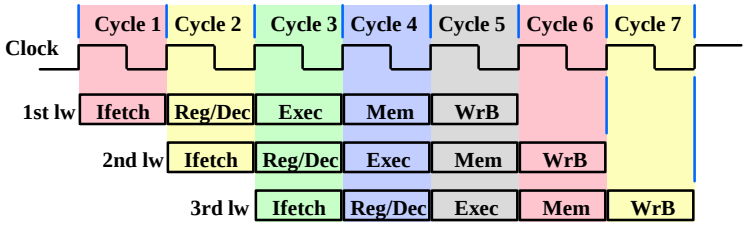
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Review: The Five Stages of a Load



- **Ifetch:** Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec:** Registers Fetch and Instruction Decode
- **Exec:** Calculate the memory address
- **Mem:** Read the data from the Data Memory
- **WrB:** Write the data back to the register file

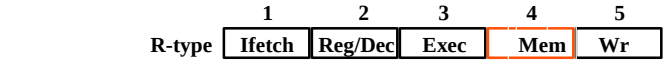
Review: Pipelining the Load Instruction

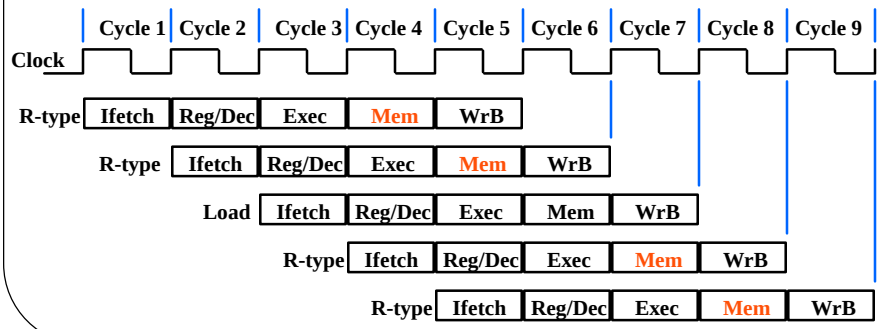


- The five independent pipeline stages are:
  - Read Next Instruction: The Ifetch stage.
  - Decode Instruction and fetch register values: The Reg/Dec stage
  - Execute the operation: The Exec stage.
  - Access Data-Memory: The Mem stage.
  - Write Data to Destination Register: The WrB stage
- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (completed) every cycle
  - The "Effective" Cycles per Instruction (CPI) is 1; ~1/5 cycle time

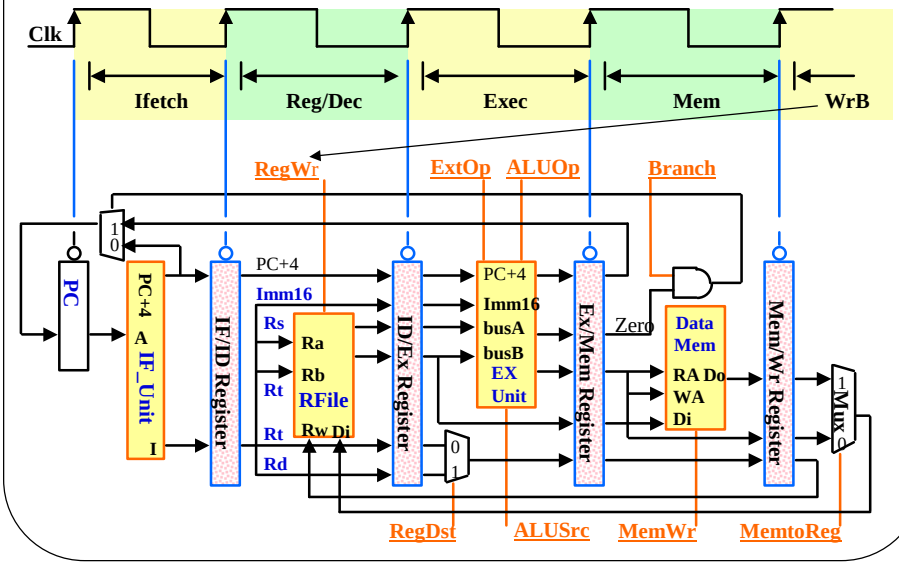
Review: Delay R-type's Write by One Cycle

- **Delay R-type's register write by one cycle:**
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NO-OP stage: nothing is being done. Effective CPI?





Review: A Pipelined Datapath



Its Not That Easy for Computers

- What could go wrong?
- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Pipelining of branches & other instructions

Speed Up Equation for Pipelining

$$\text{Speedup from pipelining} = \frac{\text{Ave Instr Time unpipelined}}{\text{Ave Instr Time pipelined}} = \frac{CPI_{unpipelined} \times \text{Clock Cycle}_{unpipelined}}{CPI_{pipelined} \times \text{Clock Cycle}_{pipelined}} = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

$$\text{Ideal CPI} = CPI_{unpipelined}/\text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{CPI_{pipelined}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{unpipe}/\text{clock}_{pipe})$$
$$= \text{Pipeline Depth}$$

$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{unpipe}/(\text{clock}_{unpipe} / 1.05)$$
$$= (\text{Pipeline Depth}/1.4) \times 1.05$$
$$= 0.75 \times \text{Pipeline Depth}$$

SpeedUp_A / SpeedUp_B = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

Three Generic Data Hazards

- $Instr_I$ followed by $Instr_J$
- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it

Three Generic Data Hazards

- $Instr_I$ followed by $Instr_J$
- **Write After Read (WAR)**
  $Instr_J$ tries to write operand before $Instr_I$ reads it
- **Can't happen in DLX 5 stage pipeline because:**
  - All instructions take 5 stages,
  - Reads are always in stage 2, and
  - Writes are always in stage 5

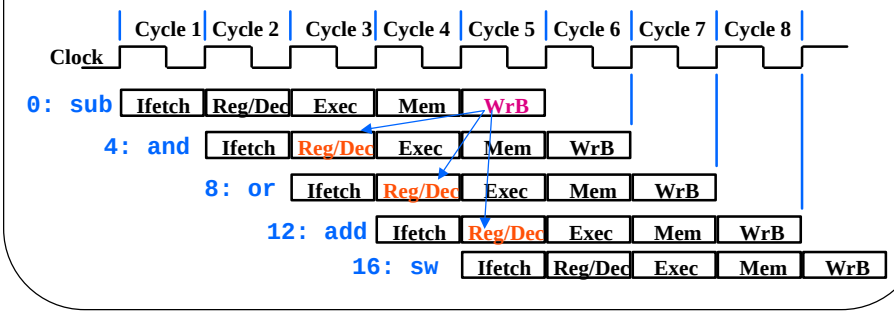
Three Generic Data Hazards

- Instr$_I$ followed by Instr$_J$
- **Write After Write (WAW)**
  Instr$_J$ tries to write operand before Instr$_I$ writes it
  - Leaves wrong result (Instr$_I$ not Instr$_J$)
- Can't happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

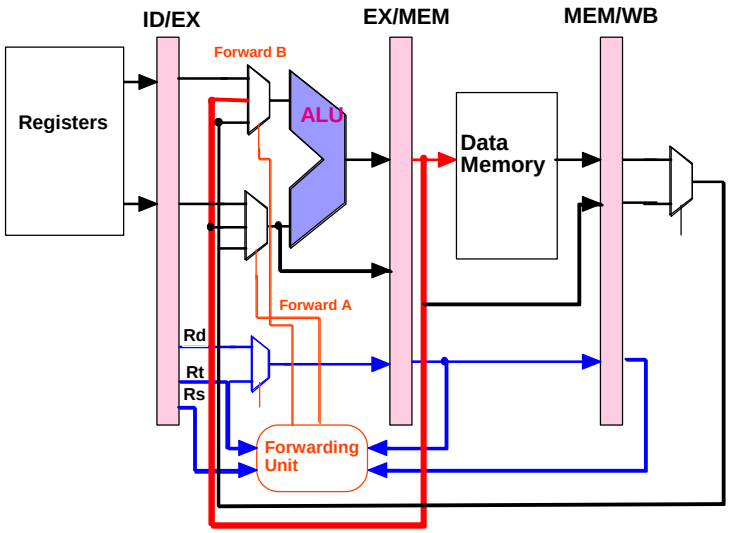
Data Hazards

- We must deal with instruction dependencies.
- Example:

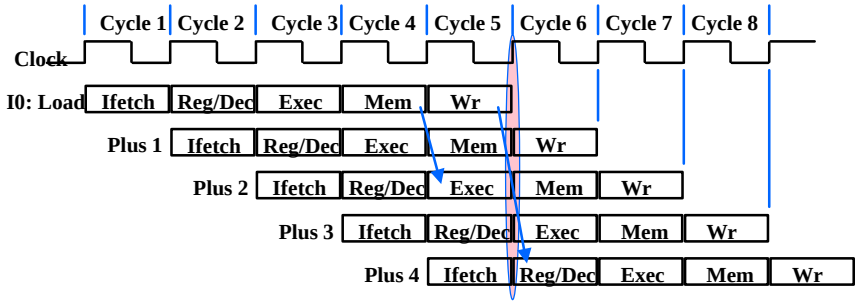
```
sub $2, $1, $3
and $12, $2, $5   # $12 depends on the result in $2
or  $13, $6, $2   # but $2 is updated 3 clock
add $14, $2, $2   # cycles later.
sw  $15, 100($2)  # We have a problem!! Data Hazard
```



RAW Data Hazard Solution: Register Forwarding

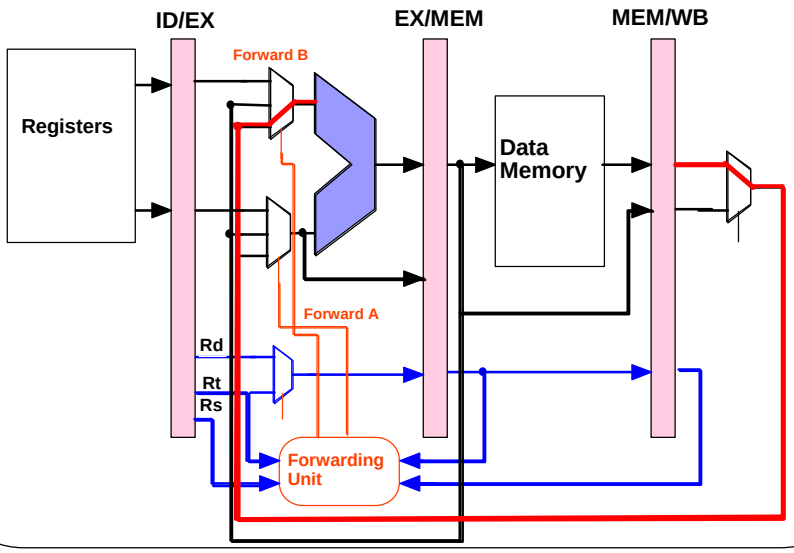


RAW Data Hazard for Load



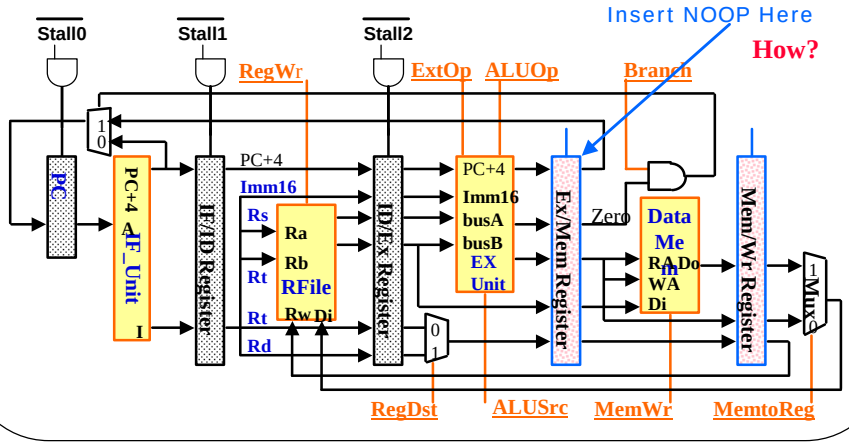
- **Load is fetched during Cycle 1:**
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load takes effect
- **This is a Data Hazard:**
  - Register forwarding reduces the load delay to ONE instruction
  - It is not possible to entirely eliminate the load Data Hazard!

Load Data Forwarding



Dealing with the Load Data Hazard

- There are two ways to deal with the load data hazard:
  - Insert a NOOP bubble into the data path.
  - Use Delayed load semantic (see a next slide)



Delayed Load

- **Load instructions are defined such that immediate successor instruction will not read result of load.**

BAD

```
ld   r1, 8(r2)
sub  r3, r1, r3
add  r2, r2, 4
```

OK

```
ld   r1, 8(r2)
add  r2, r2, 4
sub  r3, r1, r3
```


Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

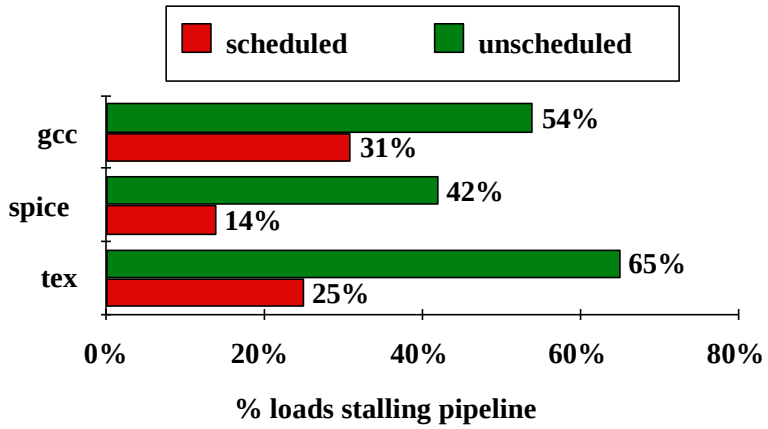
Slow code:

```
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 Avoiding Load Stalls



Review: Data Hazards

- **RAW**
  - **only one that can occur in DLX pipeline**
- **WAR**
- **WAW**
- **Data Forwarding (Register Bypassing)**
  - **send data from one stage to another bypassing the register file**
- **Still have load use delay**

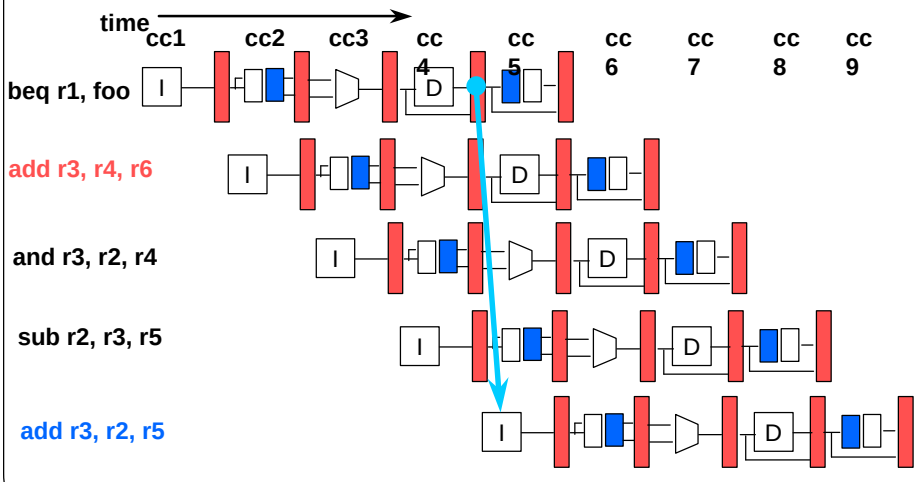
Pipelining Summary

- Just overlap tasks, and easy if tasks are independent
- Speed Up ~ Pipeline Depth; if ideal CPI is 1, then:

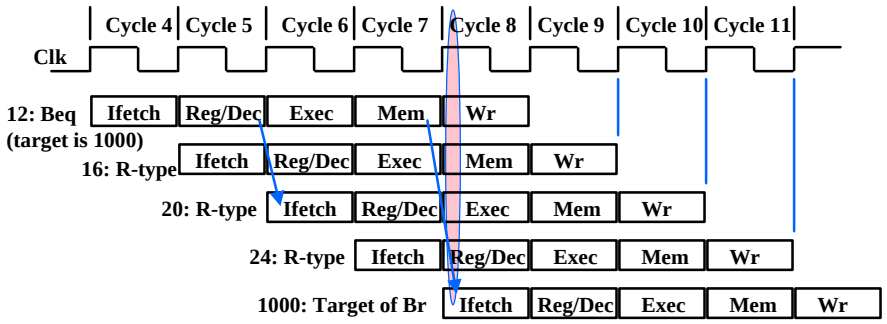
$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: discuss today
- Branches and Other Difficulties
- What makes branches difficult?

Control Hazard on Branches: Three Stage Stall



Control Hazard

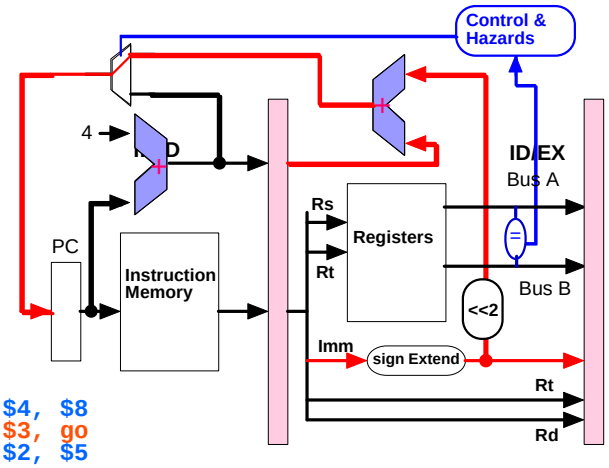


- **Although Beq is fetched during Cycle 4:**
  - **Target address is NOT written into the PC until the end of Cycle 7**
  - **Branch's target is NOT fetched until Cycle 8**
  - **3-instruction delay before the branch take effect**
- **This is called a Control Hazard:**

Branch Stall Impact

- **If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!**
- **How can you reduce this delay?**
- **Two part solution:**
  - **Determine branch taken or not sooner, AND**
  - **Compute taken branch address earlier**
- **DLX branch tests if register = 0 or != 0**
- **DLX Solution:**
  - **Move Zero test to ID/RF stage**
  - **Adder to calculate new PC in ID/RF stage**
  - **1 clock cycle penalty for branch versus 3**

Branch Delays




```
Example:
        sub $10, $4, $8
        beq $10, $3, go
        add $12, $2, $5
        . . .
go:     lw  $4, 16($12)
```


Branch Hazard

- **Can we eliminate the effect of this one cycle branch delay?**

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% DLX branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% DLX branches taken on average
- But haven't calculated branch target address in DLX
  - » DLX still incurs 1 cycle branch penalty
  - » Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

- Define branch to take place AFTER a following instruction

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

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- DLX uses this

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Cancelling branches allows more slots to be filled
- **Compiler effectiveness for single branch delay slot:**
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

Evaluating Branch Alternatives

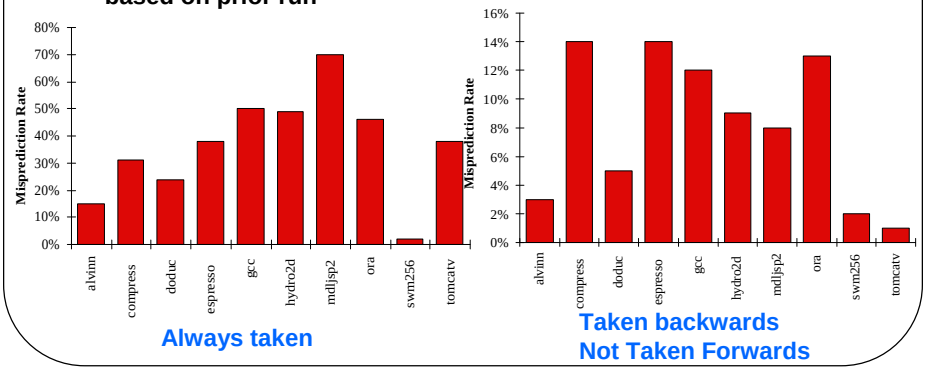
$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} ? \text{Branch penalty}}$$

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Branches = 14% of insts, 65% of them change PC

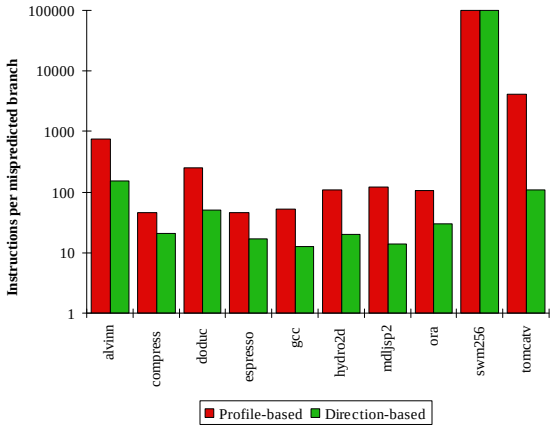
Compiler "Static" Prediction of Taken/Untaken Branches

- Improves strategy for placing instructions in delay slot
- Two strategies
  - Backward branch predict taken, forward branch not taken
  - Profile-based prediction: record branch behavior, predict branch based on prior run



Evaluating Static Branch Prediction

- Misprediction ignores frequency of branch
- "Instructions between mispredicted branches" is a better metric



Pipelining Complications

- **Interrupts (Exceptions)**
  - 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - How to restart the pipeline?
  - Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

Pipelining Complications

- **Simultaneous exceptions in > 1 pipeline stage**
  - Load with data page fault in MEM stage
  - Add with instruction page fault in IF stage
- **Solution #1**
  - Interrupt status vector per instruction
  - Defer check til last stage, kill state update if exception
- **Solution #2**
  - Interrupt ASAP
  - Restart everything that is incomplete
- **Exception in branch delay slot,**
  - SW needs two PCs
- **Another advantage for state update late in pipeline!**

Next Time

- **Next time**
  - **More pipeline complications**
  - **Longer pipelines (R4000) => Better branch prediction, more instruction parallelism?**

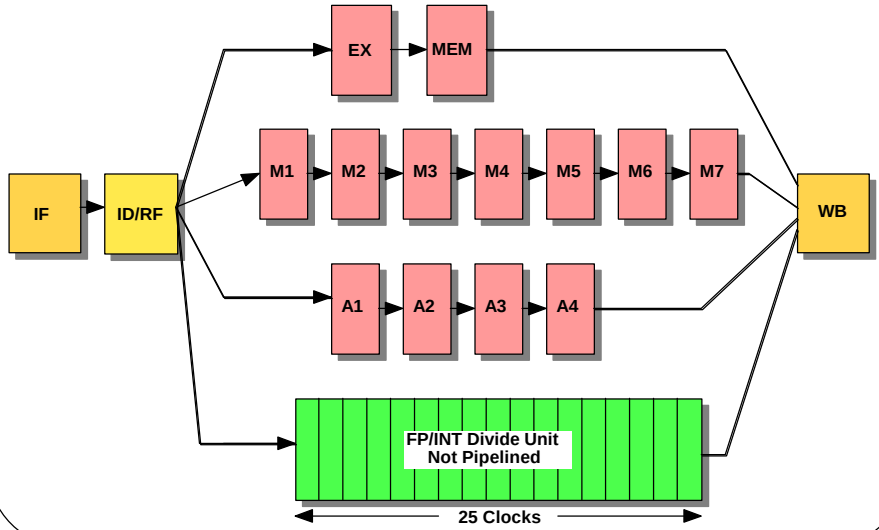
Todo

- **Read Chapter 3 and 4**
- **Homework #1 due**
- **Project selection by September 30**

Pipeline Complications

- **Complex Addressing Modes and Instructions**
- **Address modes: Autoincrement causes register change during instruction execution**
  - **Interrupts? Need to restore register state**
  - **Adds WAR and WAW hazards since writes no longer last stage**
- **Memory-Memory Move Instructions**
  - **Must be able to handle multiple page faults**
  - **Long-lived instructions: partial state save on interrupt**
- **Condition Codes**

Pipeline Complications: Floating Point



Pipelining Complications

- **Floating Point: long execution time**
- **Also, may pipeline FP execution unit so they can initiate new instructions without waiting full latency**

FP Instruction	*Latency*	*Initiation Rate*	*(MIPS R4000)*
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	(interrupts,
Square root	112	111	WAW, WAR)
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

Cycles before use result

Cycles before issue instr of same type

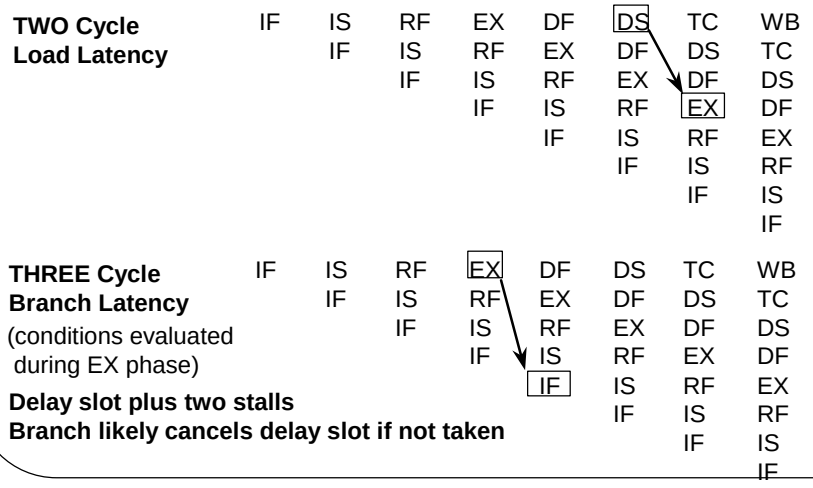
Summary of Pipelining Basics

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
- Interrupts, Instruction Set, FP makes pipelining harder
- Handling context switches.

Case Study: MIPS R4000 (100 MHz to 200 MHz)

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages: What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000



MIPS R4000 Floating Point

- **FP Adder, FP Multiplier, FP Divider**
- **Last step of FP Multiplier/Divider uses FP Adder HW**
- **8 kinds of stages in FP units:**

Stage	*Functional unit*	*Description*
A	**FP adder**	**Mantissa ADD stage**
D	**FP divider**	**Divide pipeline stage**
E	**FP multiplier**	**Exception test stage**
M	**FP multiplier**	**First stage of multiplier**
N	**FP multiplier**	**Second stage of multiplier**
R	**FP adder**	**Rounding stage**
S	**FP adder**	**Operand shift stage**
U		**Unpack FP numbers**

MIPS FP Pipe Stages

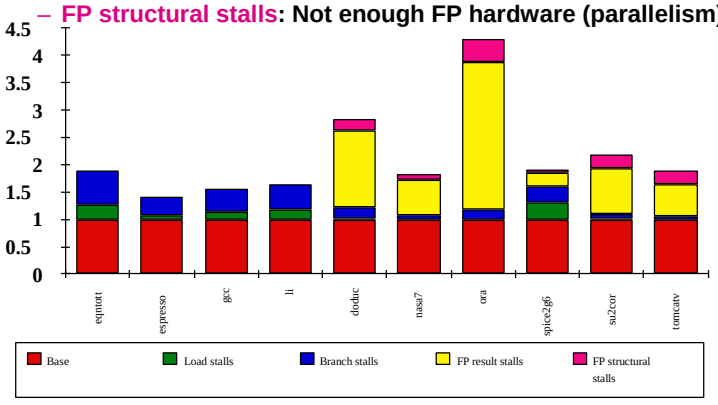
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*…*
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	$(A+R)^{108}$		…	A	R		
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*		
N	*Second stage of multiplier*	*A*	*Mantissa ADD stage*
R	*Rounding stage*	*D*	*Divide pipeline stage*
S	*Operand shift stage*	*E*	*Exception test stage*
U	*Unpack FP numbers*		

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



Next Time

- **Homework #1 is Due**
- **Instruction Level Parallelism (ILP)**
- **Read Chapter 4**