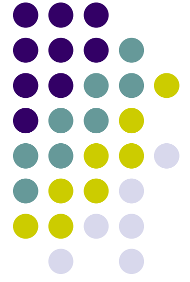
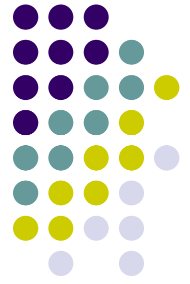
Review of Pipelining Basics

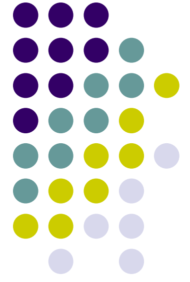


What Drives Architectural Developments?



Demanding Applications which require higher and higher Performance at lower cost and lower power

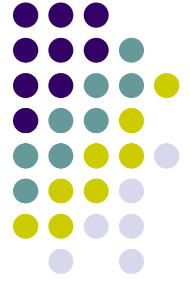
Performance Measures



- Time to run the task
  - Execution time, response time, latency
- Tasks per day, hour, week, sec, ns …
  - Throughput, bandwidth

Execution time is the ultimate measure of computer performance!

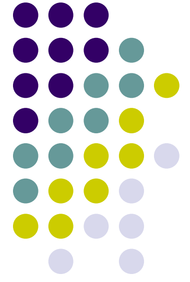
Speed Up



"X is n times faster than Y" means

$$\text{Speedup} = \frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$

Estimating Speedup
Amdahl's Law



Speedup due to enhancement E:

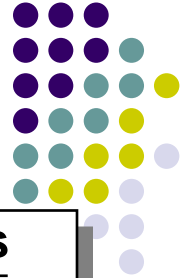
$$\text{Speedup(E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

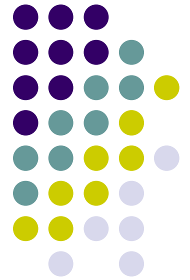
Aspects of CPU Performance



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

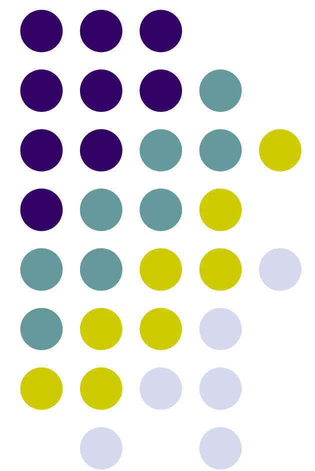
	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization	X		X
Technology			X

Architectural Enhancements

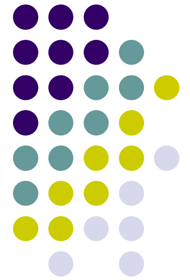


- Instruction Level Parallelism (ILP)
  - Pipelining
  - Dynamic Scheduling
  - Superscalar, VLIW and Vector processors
  - Compiler support (EPIC Architecture)
- Thread-level Parallelism
- Multiprocessors

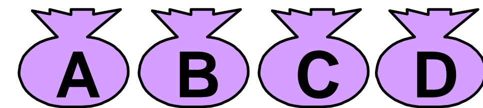
Pipelining

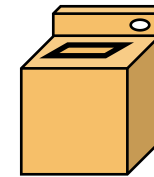


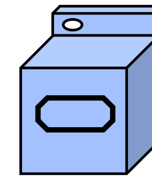
What Is Pipelining



- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes







What Is Pipelining

6 PM 7 8 9 10 11 Midnight

Time

30 40 20 30 40 20 30 40 20 30 40 20

Task Order

A

B

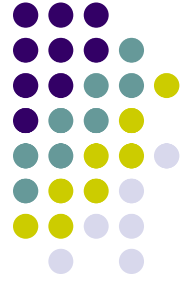
C

D

Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

What Is Pipelining



6 PM 7 8 9 10 11 Midnight

Time

30 40 40 40 40 20

Task Order

A

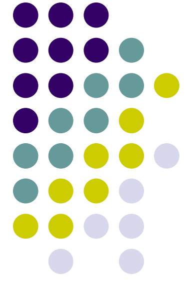
B

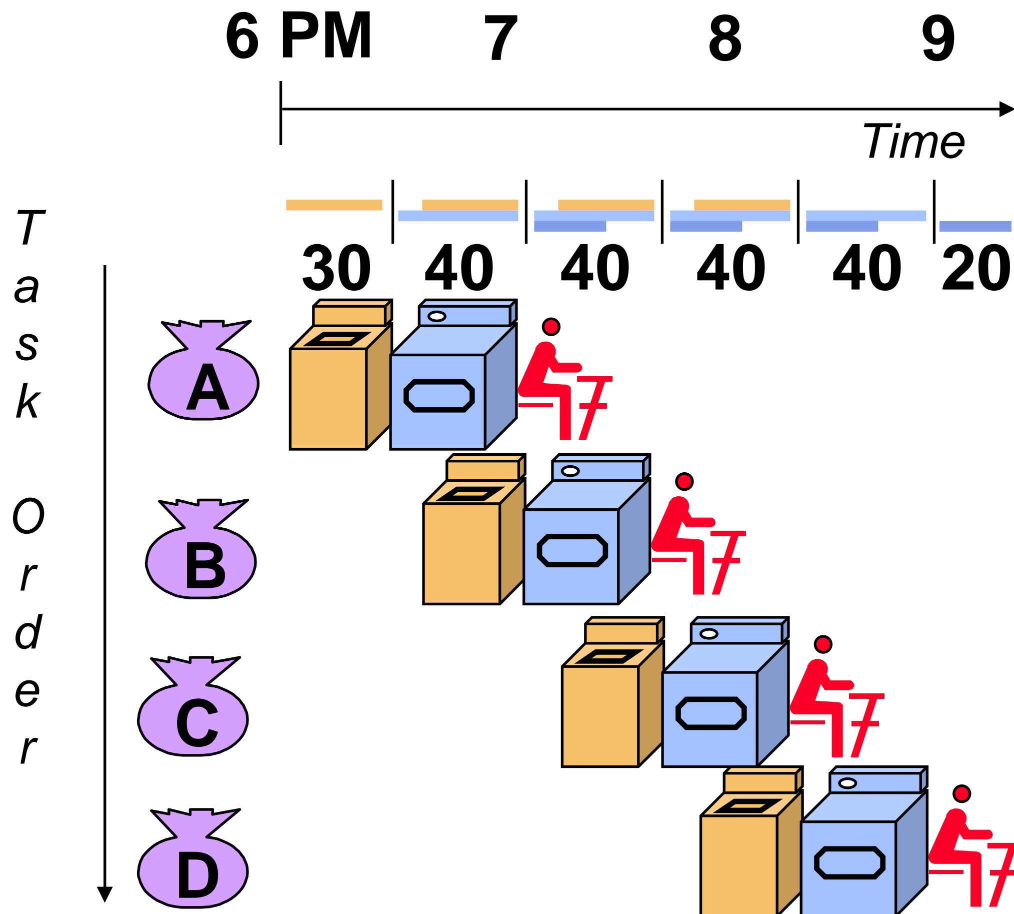
C

D

- Pipelined laundry takes 3.5 hours for 4 loads

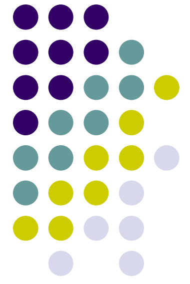
Pipelining Lessons





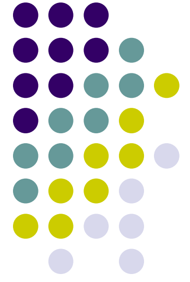
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and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

Computer Pipelines



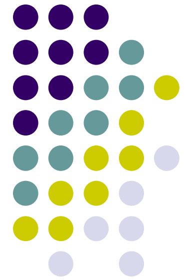
- Execute billions of instructions, so *throughput* is what matters
- What is desirable in instruction sets for pipelining?
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand many places in instruction format vs. registers located in same place?

A "Typical" RISC

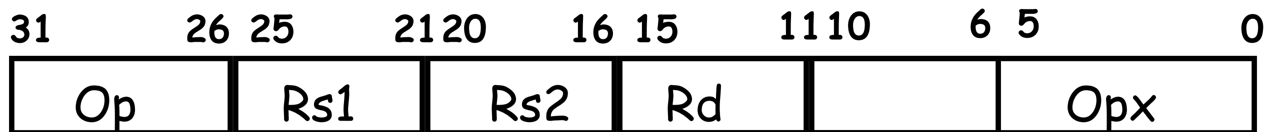


- 32-bit fixed format instruction (3 formats)
- Memory access only via load/store instrutions
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction; registers in same place
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

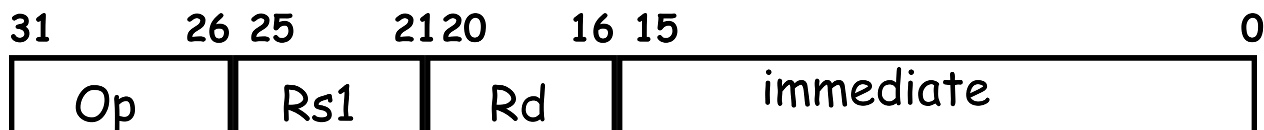
Example: MIPS (Note register location)



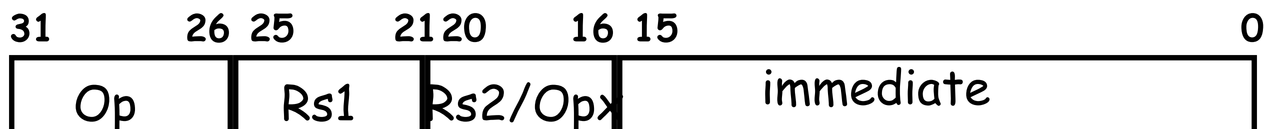
Register-Register



Register-Immediate



Branch



Jump / Call

5 Steps of MIPS Datapath Without Pipelining

Instruction Fetch | Instr. Decode Reg. Fetch | Execute Addr. Calc | Memory Access | Write Back

Next PC

Adder

4

Address

Memory

Inst

Next SEQ PC

RS1

RS2

RD

Imm

Reg File

Sign Extend

MUX

Zero?

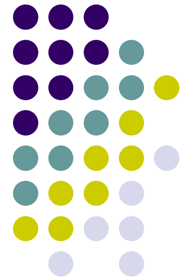
ALU

Data Memory

LMD

WB Data

MIPS Functions



Passed To Next Stage

IR <- Mem[PC]

NPC <- PC + 4

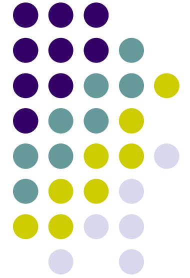
Instruction Fetch (IF):

Sends out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

IR holds the instruction that will be used in the next stage.

NPC holds the value of the next PC.

MIPS Functions



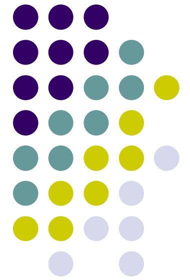
Passed To Next Stage

A <- Regs[IR6..IR10];
B <- Regs[IR10..IR15];
Imm <- ((IR16) ##IR16-31

Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers.
The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.
We extend the sign of the lower 16 bits of the Instruction Register.

MIPS Functions



Passed To Next Stage

A <- A func. B

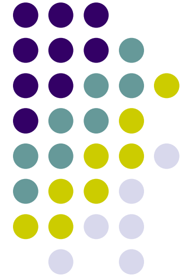
cond = 0;

Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

MIPS Functions



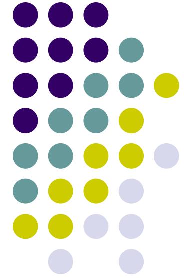
Passed To Next Stage

A = Mem[prev. B]
or
Mem[prev. B] = A

MEMORY ACCESS (MEM):

If this is an ALU, do nothing.
If its a load or store, then access memory.

MIPS Functions



Passed To Next Stage

Regs <- A, B;

WRITE BACK (WB):

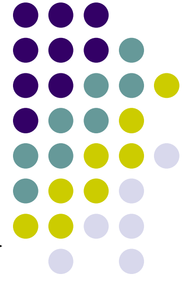
Update the registers from either the ALU or from the data loaded.

The Basic Pipeline For MIPS

Instruction Fetch | Instr. Decode Reg. Fetch | Execute Addr. Calc | Memory Access | Write Back

Next PC
Next SEQ PC
Next SEQ PC
Adder
4
Address
Memory
IF/ID
RS1
RS2
Reg File
ID/EX
MUX
Zero?
MUX
MUX
ALU
EX/MEM
MUX
Memory
MEM/WB
MUX
Sign Extend
Imm
RD
RD
RD
WB Data
Datapath
Control Path

Visualizing Pipeline For MIPS



Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Instr. Order

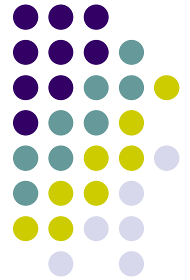
Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Speed Up Equation for Pipelining



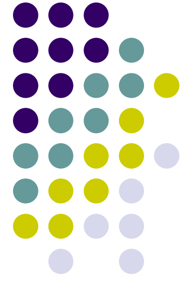
$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

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

For simple RISC pipeline, CPI = 1, therefore

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

An Example



We want to compare the performance of two machines. Which machine is faster?

- Machine A: Dual ported memory - so there are no memory stalls
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Assume:

- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

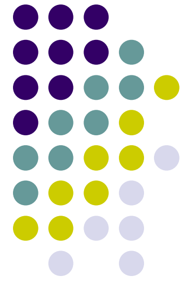
$SpeedUp_A$ = Pipeline Depth/(1 + 0) x ($clock_{unpipe}$/$clock_{pipe}$)
= Pipeline Depth

$SpeedUp_B$ = Pipeline Depth/(1 + 0.4 x 1)
x ($clock_{unpipe}$/($clock_{unpipe}$ / 1.05)
= (Pipeline Depth/1.4) x 1.05
= 0.75 x Pipeline Depth

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

- Machine A is 1.33 times faster

Limits to Pipelining



- **Hazards:** Circumstances that would cause incorrect execution if the next instruction was launched
  - Structural hazards: Attempting to use the same hardware to do two different things at the same time
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
    - Arises due to data dependences in compiler nomenclature
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Structural Hazards

Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Instr. Order

Load: Ifetch, Reg, ALU, DMem, Reg

Instr 1: Ifetch, Reg, ALU, DMem, Reg

Instr 2: Ifetch, Reg, ALU, DMem, Reg

Instr 3: Ifetch, Reg, ALU, DMem, Reg

Instr 4: Ifetch, Reg, ALU, DMem, Reg

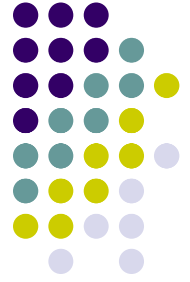
Structural Hazard

When two or more different instructions want to use same hardware resource in same cycle

e.g., MEM uses the same memory port as IF as shown in this slide.

Solution: To stall the pipeline

Structural Hazards – Stalling the pip



Time (clock cycles)

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7

Instr. Order

Load — Ifetch, Reg, ALU, DMem, Reg

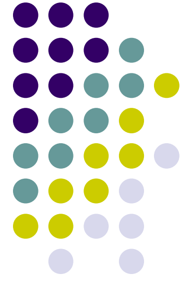
Instr 1 — Ifetch, Reg, ALU, DMem, Reg

Instr 2 — Ifetch, Reg, ALU, DMem, Reg

Stall — Bubble, Bubble, Bubble, Bubble, Bubble

Instr 3 — Ifetch, Reg, ALU, DMem, Reg

Dealing with Structural Hazards

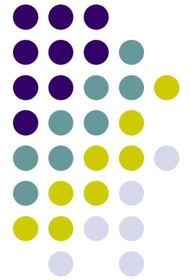


Stall
- **low cost, simple**
- **increases CPI**
- **used for rare cases since stalling affects performance**

Pipeline hardware resource
- **useful for multi-cycle resources**
- **good performance**
- **sometimes complex e.g., RAM**

Replicate resource
- **good performance**
- **increases cost (+ maybe interconnect delay)**
- **useful for cheap or divisible resources**

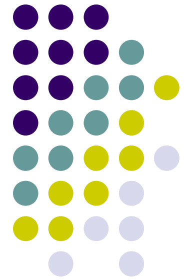
Data and Control hazards



Arise due to

- Dependences between instructions in a program
  - Data dependence
  - Control dependence
- Dependences are properties of programs
- Whether the dependences turn out to be hazards and cause stalls in the pipeline are properties of the pipeline organization

Data Dependences



- Three types of dependences: **data** dependences (also called true data dependences), **name** dependences and **control** dependences

- An instruction *j* is *data dependent* on instruction *i* if either of the following holds:
  - Instruction *i* produces a result that may be used by instruction *j*, or
  - Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*

Name Dependences

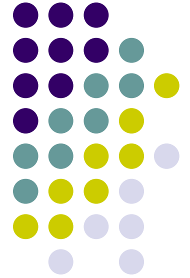
- Occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name

- Two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:
  - An *antidependence* between instruction i and instruction j occurs when instruction *j* writes a register or memory location that instruction *i* reads. The original ordering must be preserved.
  - An *output dependence* occurs when instruction *i* and instruction *j* write the same register or memory location. The ordering between the instructions must be preserved.

Data Hazards

- Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions:
- RAW (*read after write)*
  - Corresponds to a true data dependence
  - Program order must be preserved
- WAW (*write after write)*
  - Corresponds to an output dependence
  - Occurs when there are multiple writes or a short integer pipeline and a longer floating-point pipeline or when an instruction proceeds when a previous instruction is stalled
- WAR (*write after read)*
  - Arises from an anti dependence
  - Cannot occur in most static issue pipelines
  - Occurs either when there are early writes *and* late reads, or when instructions are re-ordered.

Data Hazards



Read After Write (RAW)

$Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

Write After Read (WAR)

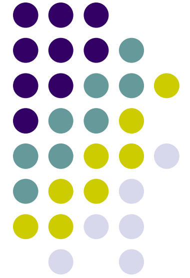
$Instr_J$ tries to write operand *before* $Instr_I$ reads i

- Can get wrong operand

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Data Hazards



Write After Write (WAW)

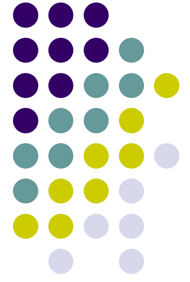
$Instr_J$ tries to write operand *before* $Instr_I$ writes it

- Leaves wrong result ($Instr_I$ not $Instr_J$)

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

Data Hazards



Simple Solution to RAW

- Stall
  - Hardware detects RAW and stalls
  - Low cost to implement, simple
  - Reduces IPC
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/short-circuit (We will use the word “forward”)
- Use data before it is in the register
  + reduces/avoids stalls
  -- complex
- Crucial for common RAW hazards

Data Hazards

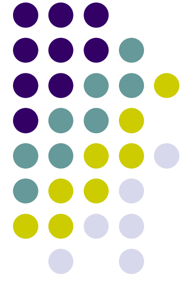
Time (clock cycles)

IF ID/RF EX MEM WB

Instr. Order

```
dadd r1,r2,r3
dsub r4,r1,r5
and  r6,r1,r7
or   r8,r1,r9
xor  r10,r1,r11
```

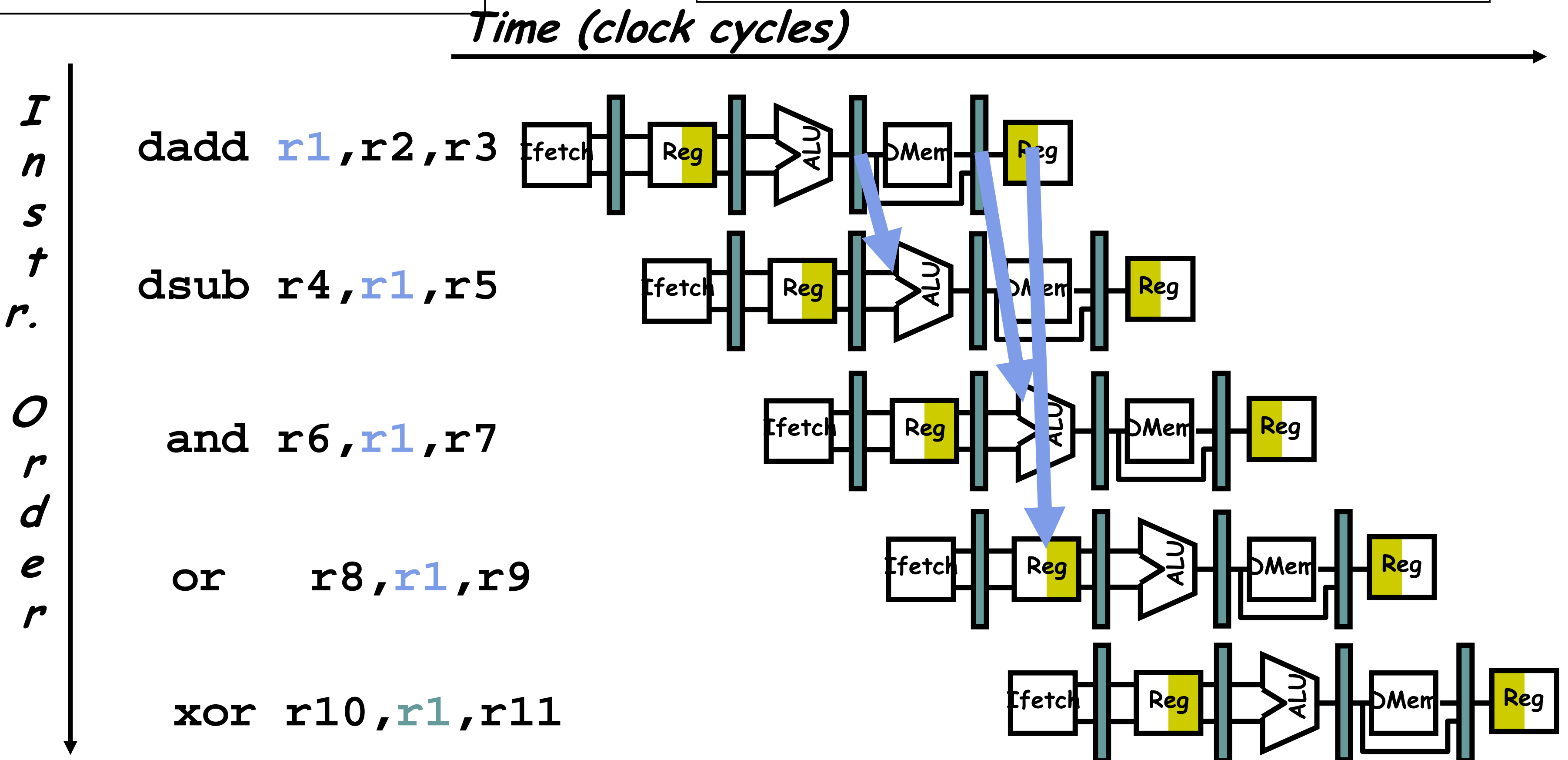

The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.



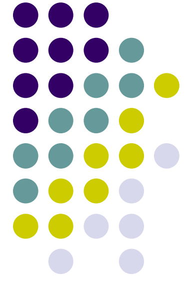
Data Hazards

Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

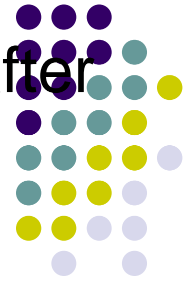


Hardware Change for Forwarding



Data Hazards

The data isn't loaded until after the MEM stage.



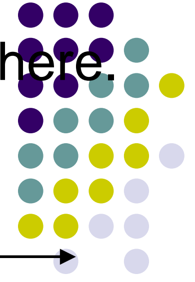
Time (clock cycles)

Instr. Order

```
ld r1,0(r2)
dsub r4,r1,r5
and r6,r1,r7
or   r8,r1,r9
```


There are some instances where hazards occur, even with forwarding.

The stall is necessary as shown here.



Data Hazards

Time (clock cycles)

Instr. Order

ld r1, 0(r2)

dsub r4,r1,r5

and r6,r1,r7

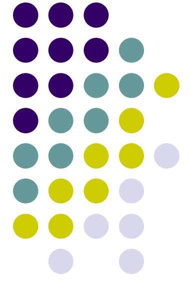
or r8,r1,r9

There are some instances where hazards occur, even with forwarding.

Control Hazards

- **A <u>control hazard</u> is due to control dependence i.e., when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.**
- Control Dependence
  - Two general constraints imposed by control dependences:
    - An instruction that is control dependent on its branch cannot be moved *before* the branch so that its execution is *no longer controlled* by the branch.
    - An instruction that is not control dependent on its branch cannot be moved *after* the branch so that its execution *is controlled* by the branch.

Control Hazard on Branches Three Stage Stall



10: beq r1,r3,36

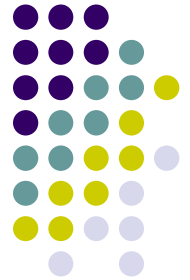
14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

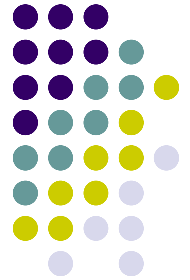
36: xor r10,r1,r11

Example: Branch Stall Impact



- If 30% branch, Stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or $\neq$ 0
- MIPS 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

Control Hazards –Four Alternatives



#1: Stall until branch direction is clear

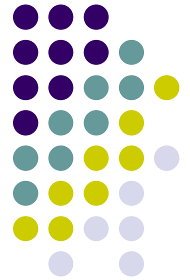
#2: Predict Branch Not Taken

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

#3: Predict Branch Taken

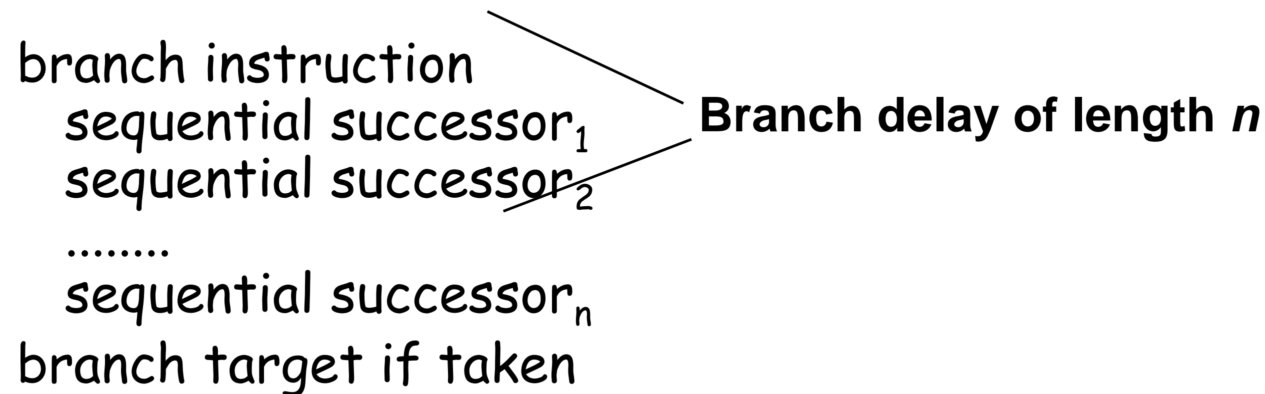
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Four Branch Hazard Alternatives Contd.



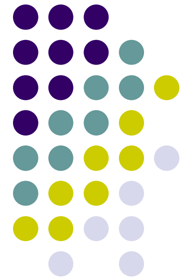
#4: Delayed Branch

- Define branch to take place AFTER a following instruction



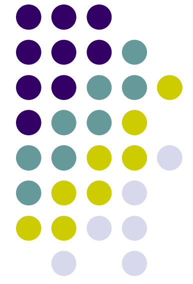
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

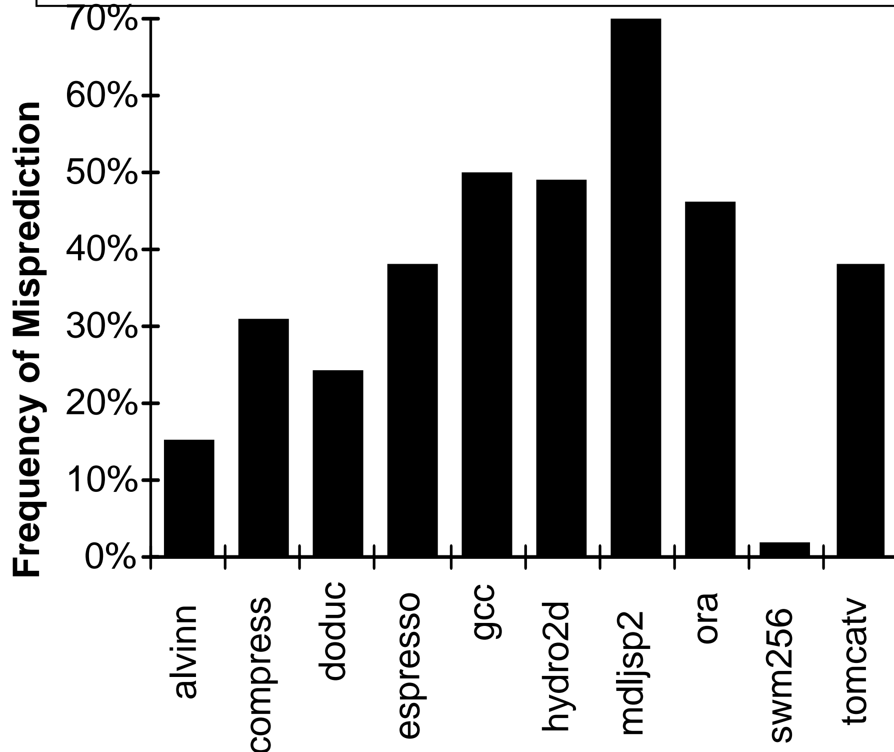


- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Canceling branches allow more slots to be filled
- Delayed Branch downside: Difficult to find instructions.

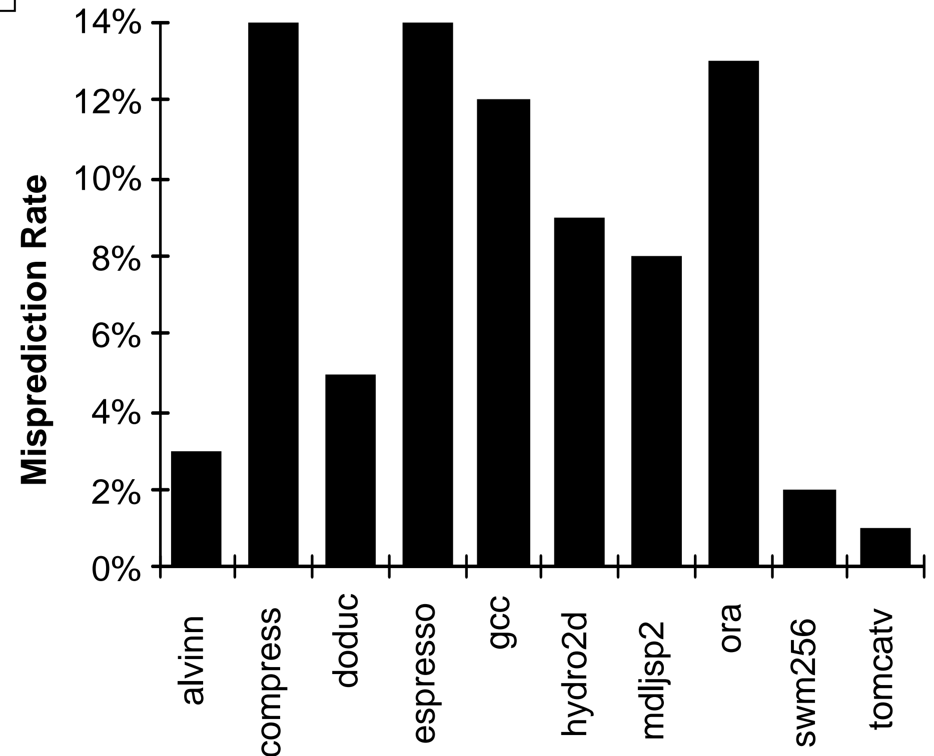
Compiler "Static" Prediction of Taken/Untaken Branches



The compiler can program what it thinks the branch direction will be. Here are the results when it does so.



Always taken

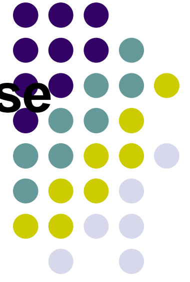


Taken backwards
Not Taken Forwards

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

What Makes Pipelining Hard?



Interrupts cause great havoc!

Examples of interrupts:

- Power failing,
- Arithmetic overflow,
- I/O device request,
- OS call,
- Page fault

Interrupts (also known as: faults, exceptions, traps) often require

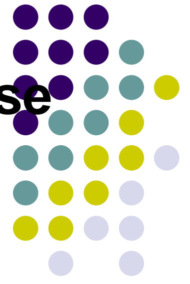
- surprise jump (to vectored address)
- linking return address
- saving of PSW (including CCs)
- state change (e.g., to kernel mode)

There are 5 instructions executing in 5 stage pipeline when an interrupt occurs:

- **How to stop the pipeline?**
- **How to restart the pipeline?**
- **Who caused the interrupt?**

What Makes Pipelining Hard?

Interrupts cause great havoc!



What happens on interrupt while in delay slot ?

- Next instruction is not sequential

solution #1: save multiple PCs

- Save current and next PC
- Special return sequence, more complex hardware

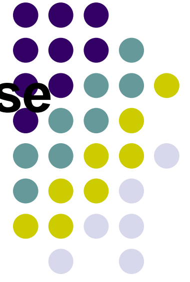
solution #2: single PC plus Branch delay bit

- PC points to branch instruction

Stage	*Problem that causes the interrupt*
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

What Makes Pipelining Hard?

Interrupts cause great havoc!

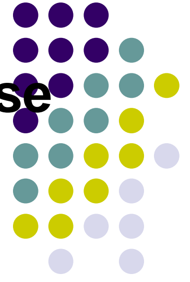


- Simultaneous exceptions in more than one pipeline stage,
- e.g.,
  - Load with data page fault in MEM stage
  - Add with instruction page fault in IF stage
  - Add fault will happen BEFORE load fault
- Solution #1
  - Interrupt status vector per instruction
  - Defer check until last stage, kill state update if exception
- Solution #2
  - Interrupt ASAP
  - Restart everything that is incomplete

Another advantage for state update late in pipeline!

What Makes Pipelining Hard?

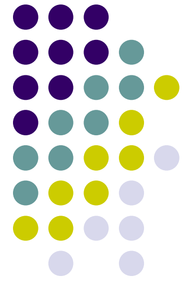
Interrupts cause great havoc!



Here's what happens on a data page fault.

	1	2	3	4	5	6	7	8	9		
i	F	D	X	M	W						
i+1		F	D	X	M	W <- page fault					
i+2			F	D	X	M	W <- squash				
i+3				F	D	X	M	W <- squash			
i+4					F	D	X	M	W <- squash		
i+5 trap ->						F	D	X	M	W	
i+6 trap handler ->								F	D	X	M W

What Makes Pipelining Hard?



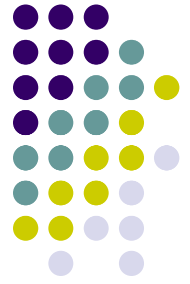
Complex Addressing Modes and Instructions

- Address modes: Auto increment causes register change during instruction execution
  - Interrupts? Need to restore register state
  - Adds WAR and WAW hazards since writes are no longer the last stage.

- Memory-Memory Move Instructions
  - Must be able to handle multiple page faults
  - Long-lived instructions: partial state save on interrupt

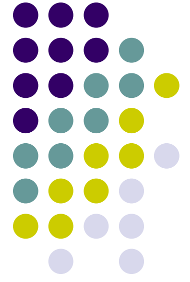
- Condition Codes

Floating Point Pipeline – Multi-cycle Operations



- Long Latency instructions
- More complex pipeline
- Multiple functional units
  - Examples:
    - Floating Point Divider Unit
    - Floating Point Multiplier Unit
    - Floating Point Adder Unit
    - Floating Point Integer Unit

Multi-Cycle Operations



Floating point gives long execution time.

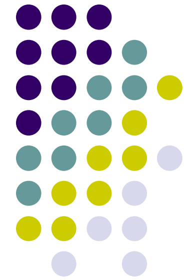
This causes a stall of the pipeline.

It's possible to pipeline the FP execution unit so it can initiate new instructions without waiting full latency. Can also have multiple FP units.

Example:

FP Instruction	*Latency*	*Initiation interval*
Add, Subtract	4	1
Multiply	8	1
Divide	36	35
Square root	112	111
Negate	2	1
Absolute value	2	1
FP compare	3	2

Multi-Cycle Operations



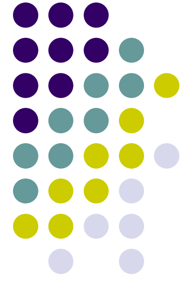
Divide, Square Root take -10X to -30X longer than Add

- Interrupts?
- Adds WAR and WAW hazards since pipelines are no longer same length

	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	EX	MEM	WB						
I + 1		IF	ID	EX	EX	EX	EX	MEM	WB		
I + 2			IF	ID	EX	MEM	WB				
I + 3				IF	ID	EX	EX	EX	EX	MEM	WB
I + 4					IF	ID	EX	MEM	WB		
I + 5						IF	ID	--	--	EX	EX
I + 6							IF	--	--	ID	EX

Notes:
I + 2: WAW, but this complicates an interrupt
I + 4: WB conflict
I + 5: stall forced by structural hazard
I + 6: stall forced by in-order issue

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
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction