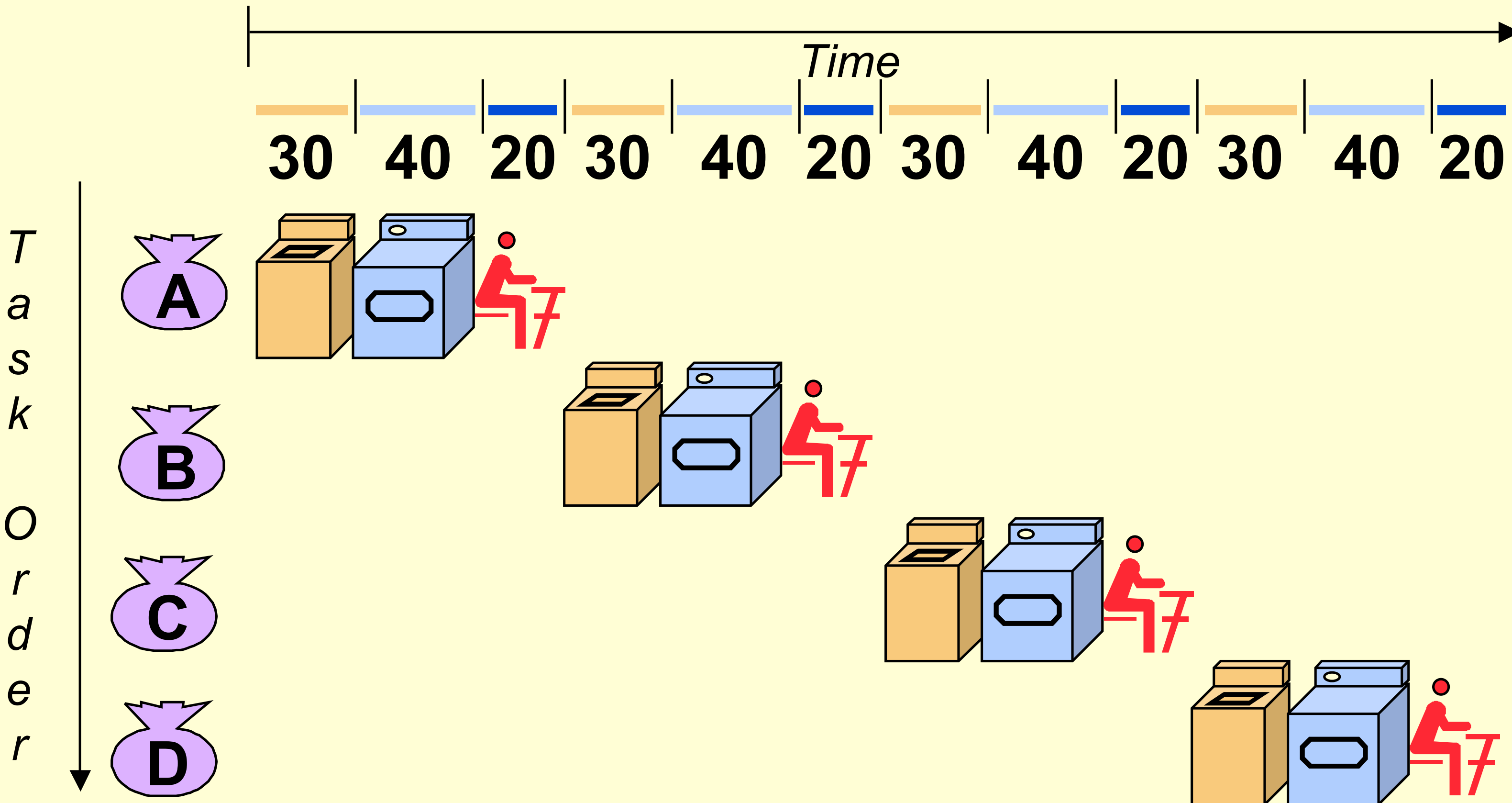
# CMSC 611: Advanced Computer Architecture

## Pipelining & Pipeline Hazards

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from David Culler, UC Berkeley CS252, Spr 2003 course slides, ©2002 UC Berkeley
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science

# Sequential Laundry




- Washer takes 30 min, Dryer takes 40 min, folding takes 20 min
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Slide: Dave Patterson

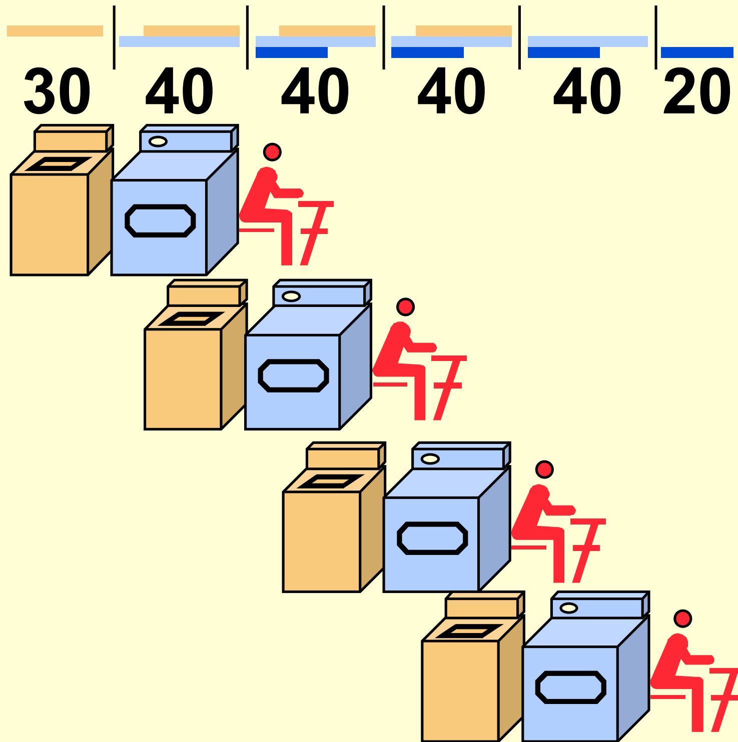
# Pipelined Laundry

6 PM 7 8 9 10 11 Midnight

*Time*

*Task Order*

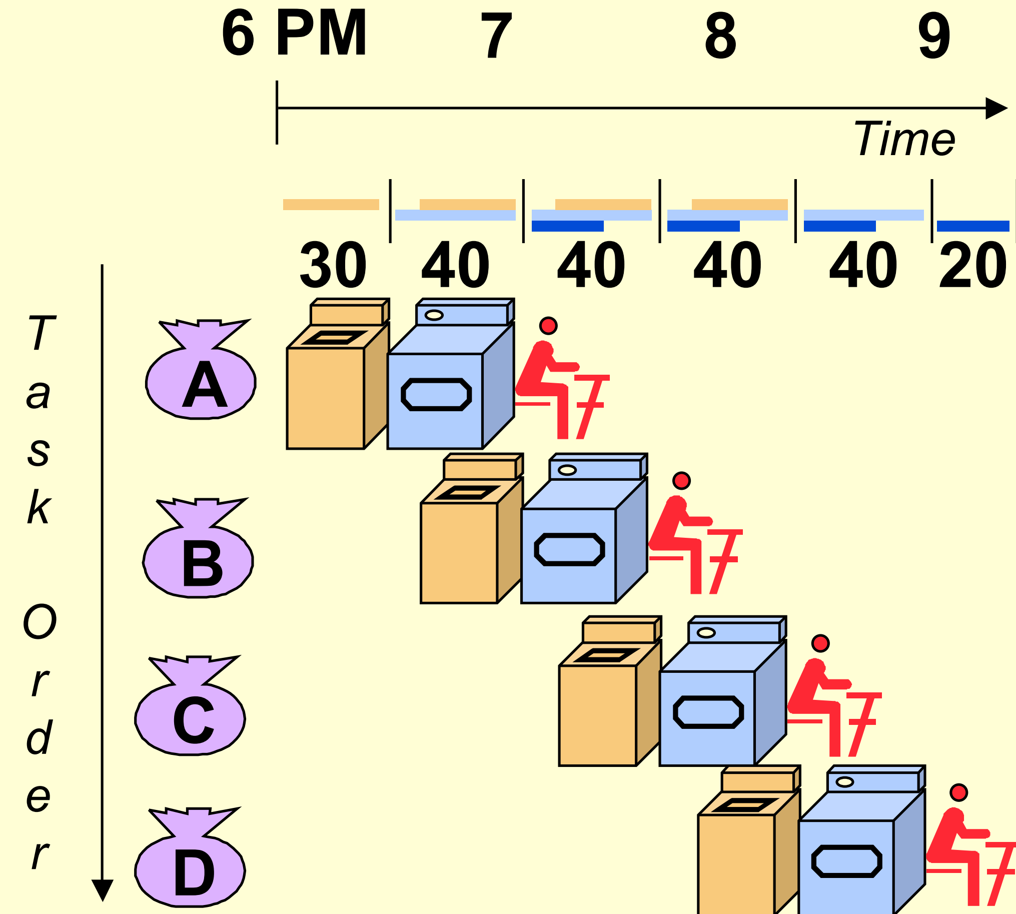
A
B
C
D




- Pipelining means start work as soon as possible
- Pipelined laundry takes 3.5 hours for 4 loads

Slide: Dave Patterson

# Pipelining Lessons

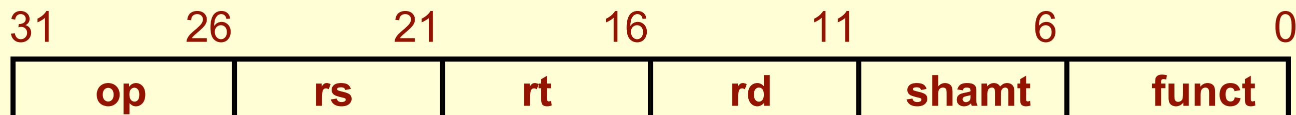



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

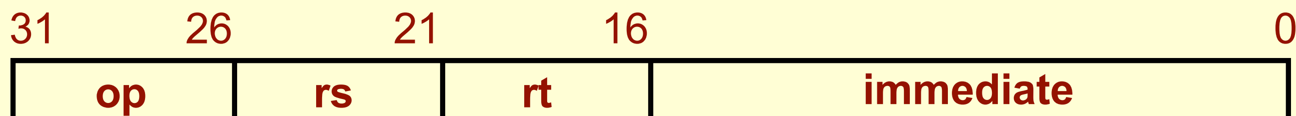
Slide: Dave Patterson

# MIPS Instruction Set

- RISC characterized by the following features that simplify implementation:
  - All ALU operations apply only on registers
  - Memory is affected only by load and store
  - Instructions follow very few formats and typically are of the same size




6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

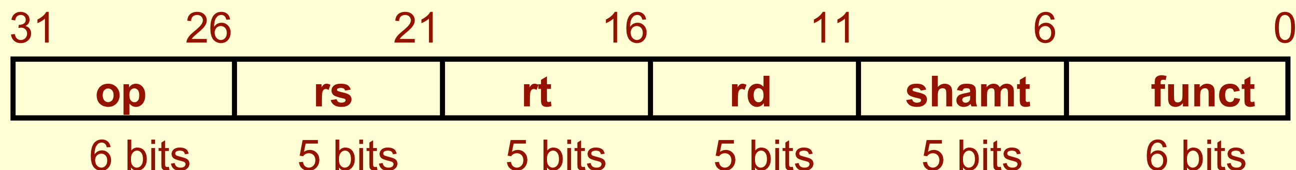



6 bits 5 bits 5 bits 16 bits

6 bits 26 bits

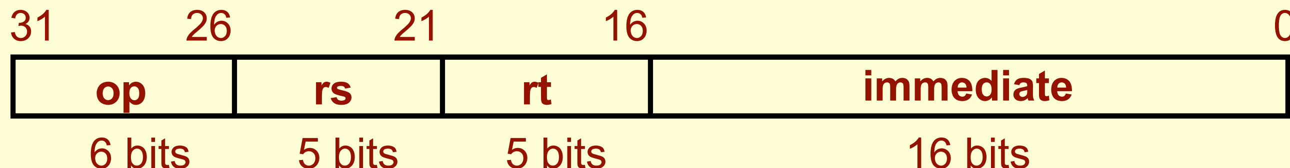
# MIPS Instruction Formats

- R-type (register)
  - Most operations
    - add $t1, $s3, $s4   # $t1 = $s3 + $s4
  - rd, rs, rt all registers
  - op always 0, funct gives actual function



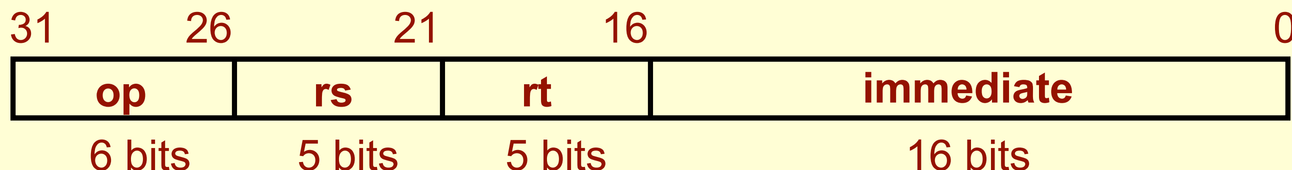
# MIPS Instruction Formats

- I-type (immediate)
  - ALU with one immediate operand
    - addi $t1, $s2, 32 # $t1 = $s2 + 32
  - Load, store within $\pm 2^{15}$ of register
    - lw $t0, 32($s2) # $s1 = $s2[32] or *(32+s2)
  - Load immediate values
    - lui $t0, 255 # $t0 = (255<<16)
    - li $t0, 255



# MIPS Instruction Formats

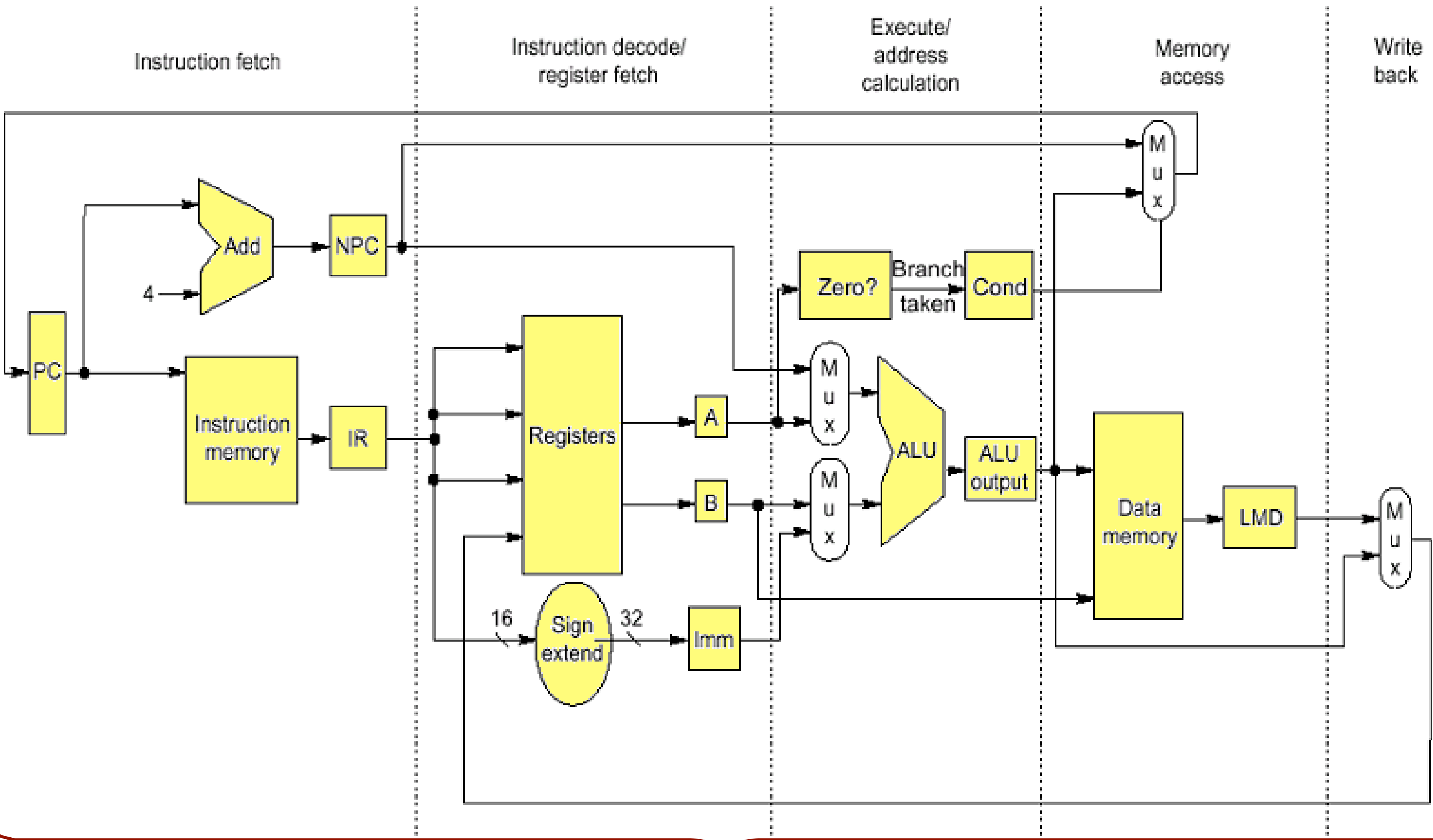
- I-type (immediate)
  - PC-relative conditional branch
  - $\pm 2^{15}$ from PC **after** instruction
    - beq $s1, $s2, L1 # goto L1 if ($s1 = $s2)
    - bne $s1, $s2, L1 # goto L1 if ($s1 ≠ $s2)



# MIPS Instruction Formats

- J-type (jump)
  - unconditional jump
    - j L1 # goto L1
  - Address is concatenated to top bits of PC
    - Fixed addressing within $2^{26}$

# Single-cycle Execution




❶

Figure: Dave Patterson

# Multi-Cycle Implementation of MIPS

❶ **Instruction fetch cycle (IF)**

IR ← Mem[PC];  NPC ← PC + 4

❷ **Instruction decode/register fetch cycle (ID)**

A ← Regs[$IR_{6..10}$];  B ← Regs[$IR_{11..15}$];  Imm ← $((IR_{16})^{16}$ ##$IR_{16..31})$

❸ **Execution/effective address cycle (EX)**

Memory ref:  ALUOutput ← A + Imm;

Reg-Reg ALU:  ALUOutput ← A *func* B;

Reg-Imm ALU:  ALUOutput ← A *op* Imm;

Branch:  ALUOutput ← NPC + Imm;  Cond ← (A op 0)

❹ **Memory access/branch completion cycle (MEM)**

Memory ref:  LMD ← Mem[ALUOutput]  or  Mem(ALUOutput] ← B;

Branch:  if (cond) PC ←ALUOutput;

❺ **Write-back cycle (WB)**

Reg-Reg ALU:  Regs[$IR_{16..20}$] ← ALUOutput;

Reg-Imm ALU:  Regs[$IR_{11..15}$] ← ALUOutput;

Load:  Regs[$IR_{11..15}$] ← LMD;

# Multi-cycle Execution

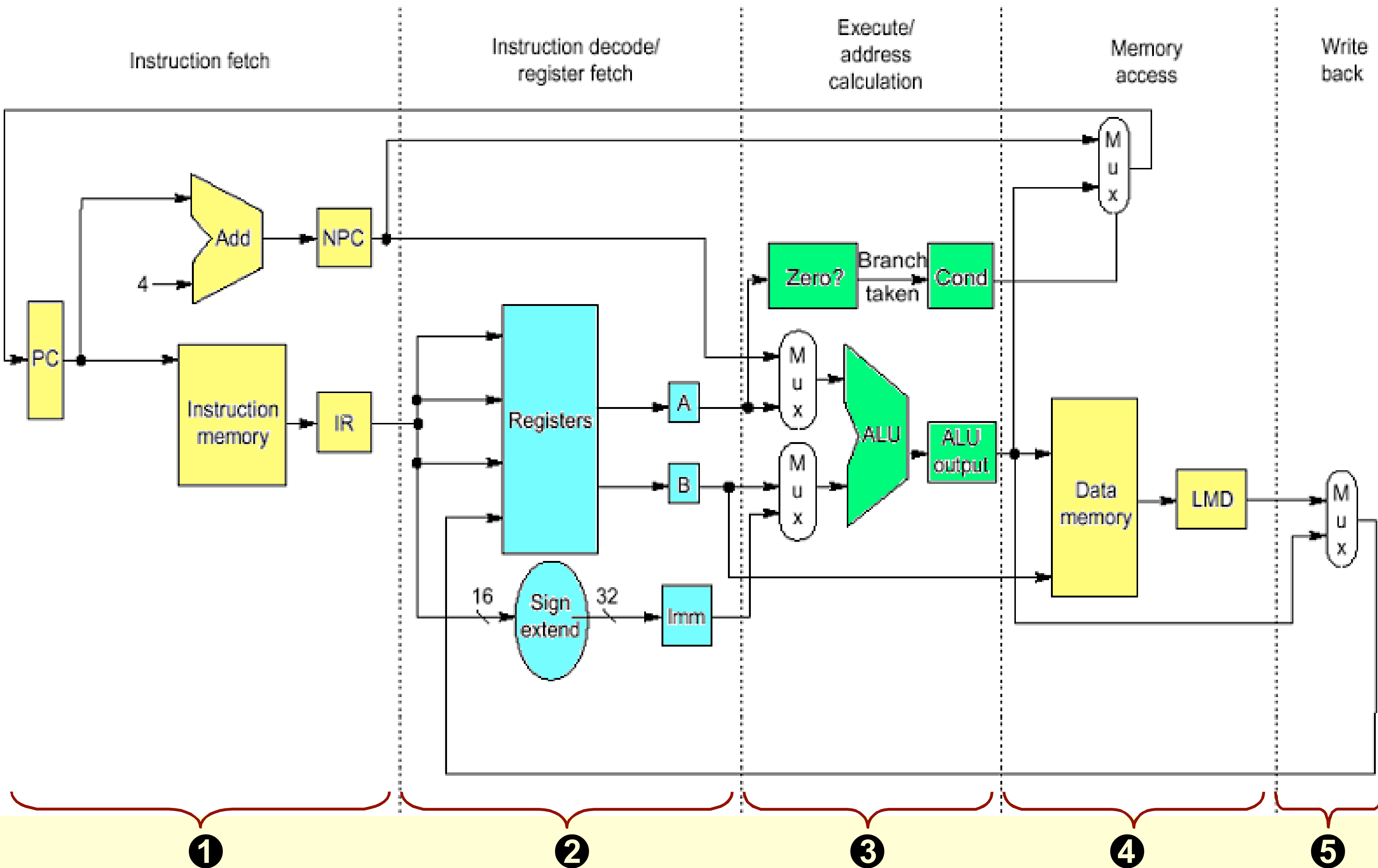
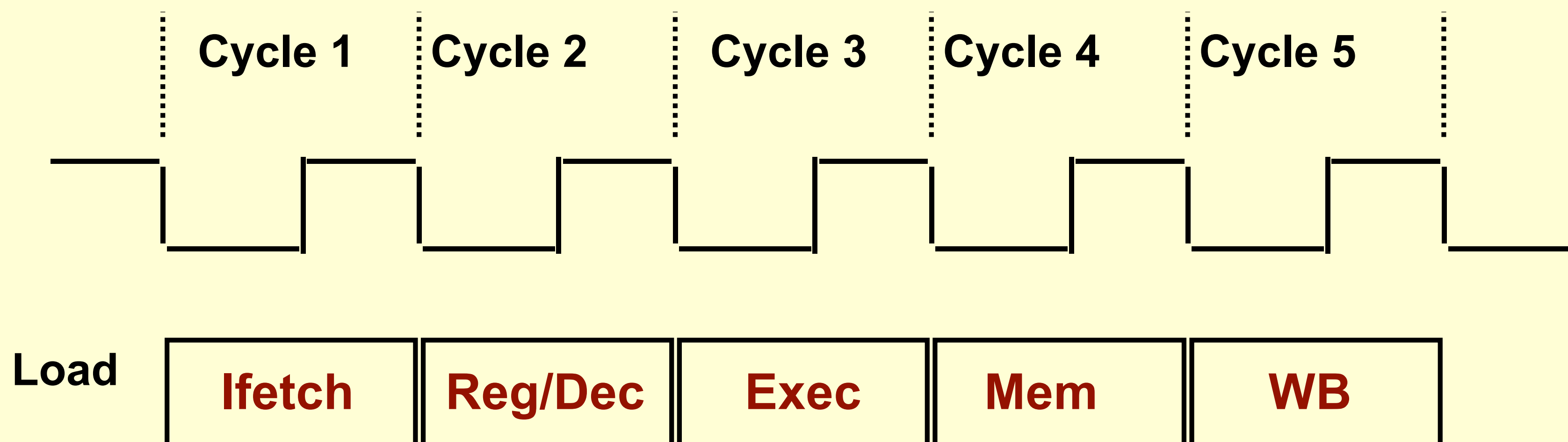



Figure: Dave Patterson

# Stages of Instruction Execution

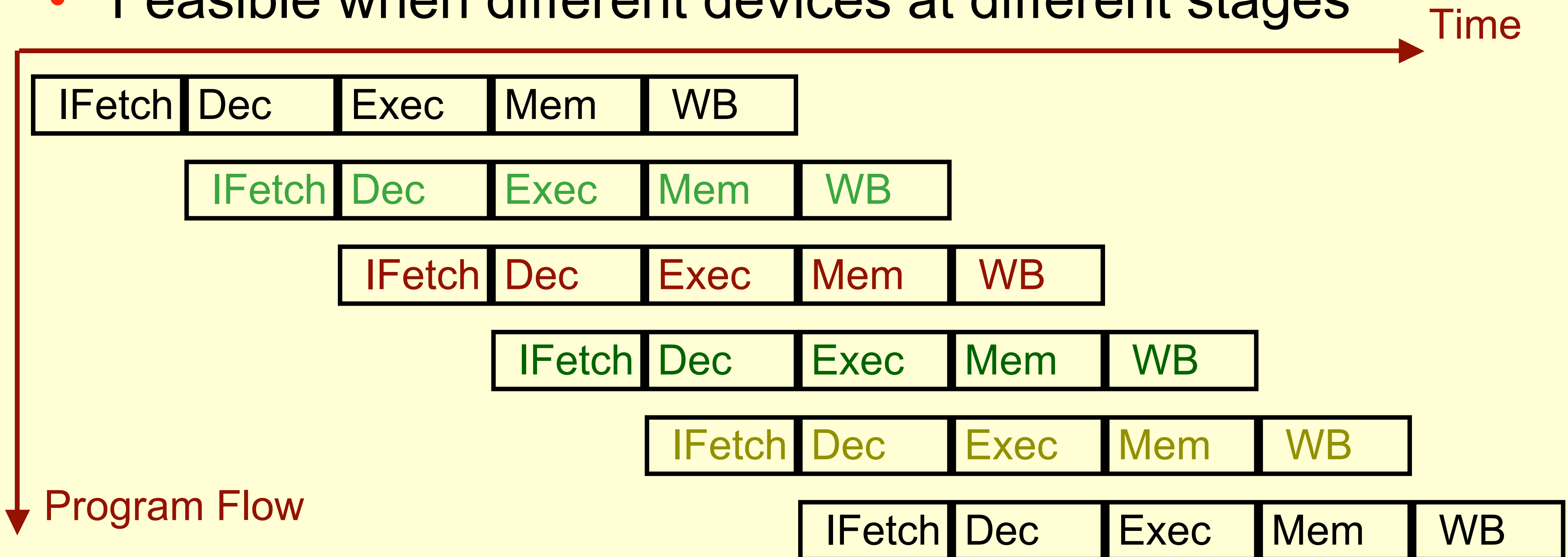



- The load instruction is the longest
- All instructions follows at most the following five steps:
  - Ifetch: Instruction Fetch
    - Fetch the instruction from the Instruction Memory and update PC
  - Reg/Dec: Registers Fetch and Instruction Decode
  - Exec: Calculate the memory address
  - Mem: Read the data from the Data Memory
  - WB: Write the data back to the register file

Slide: Dave Patterson

# Instruction Pipelining

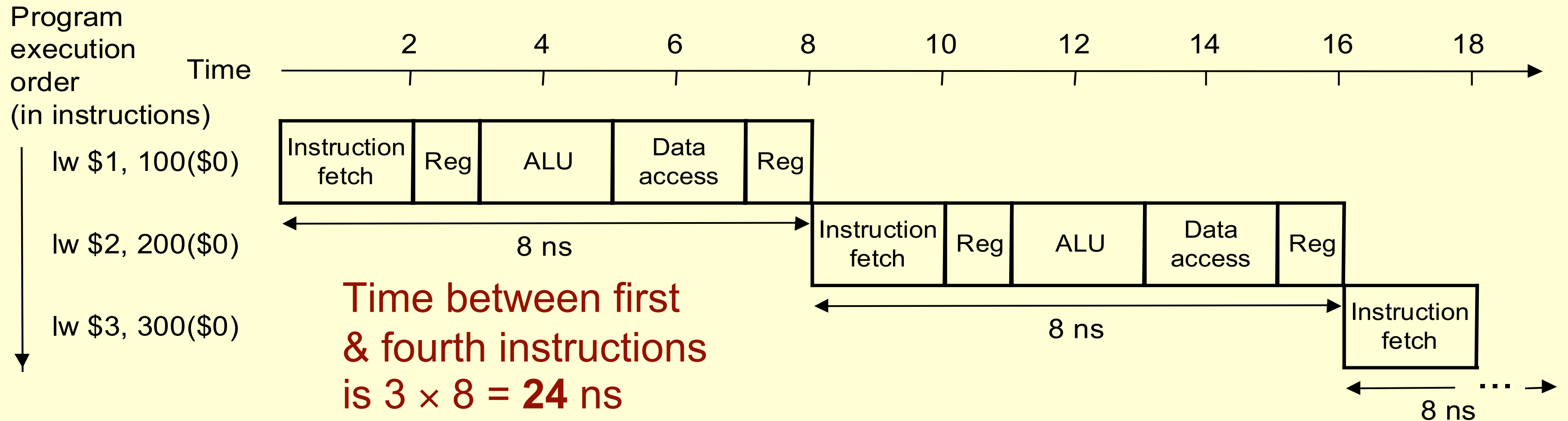
- Start handling next instruction while the current instruction is in progress
- Feasible when different devices at different stages

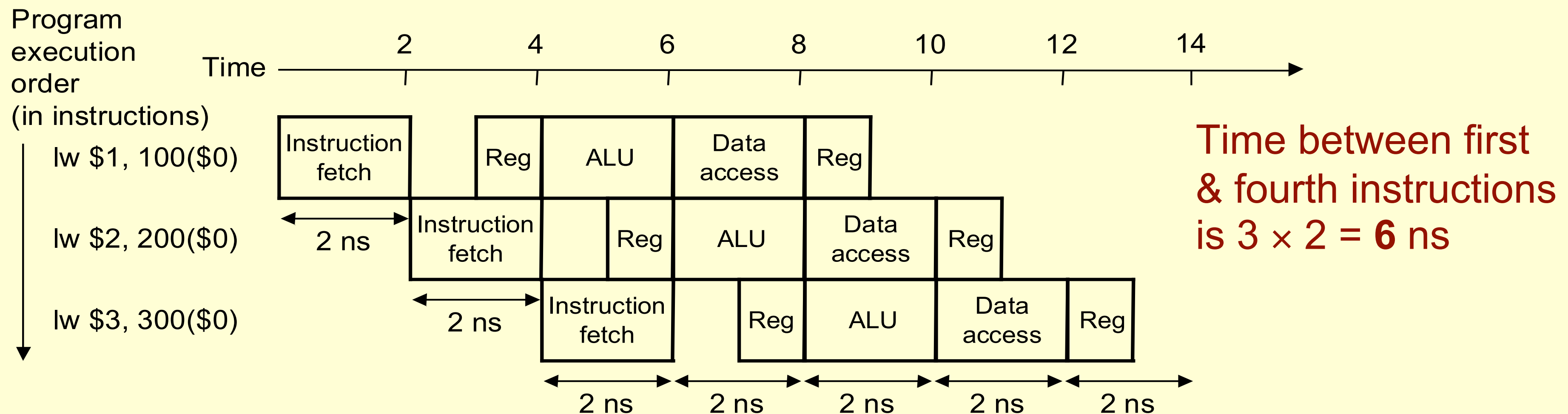



$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput

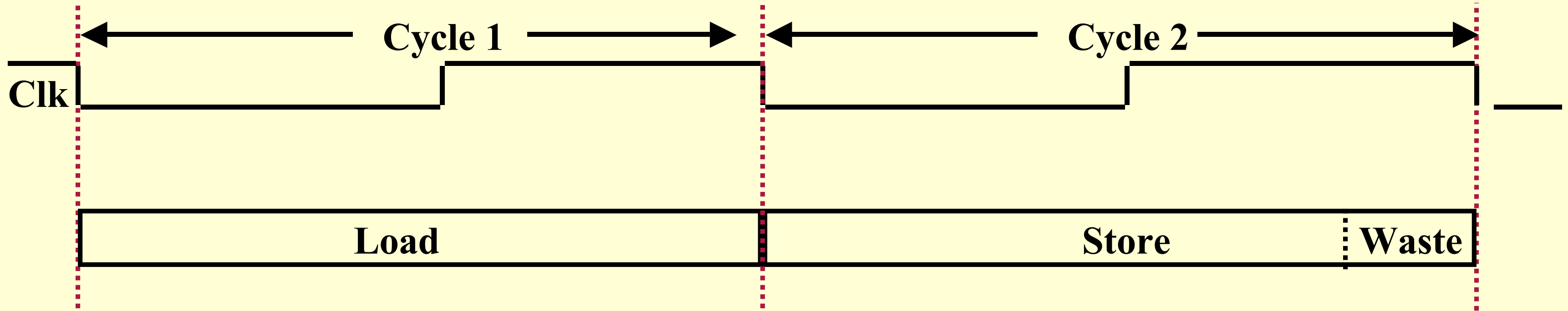
# Example of Instruction Pipelining




*Ideal and upper bound for speedup is number of stages in the pipeline*

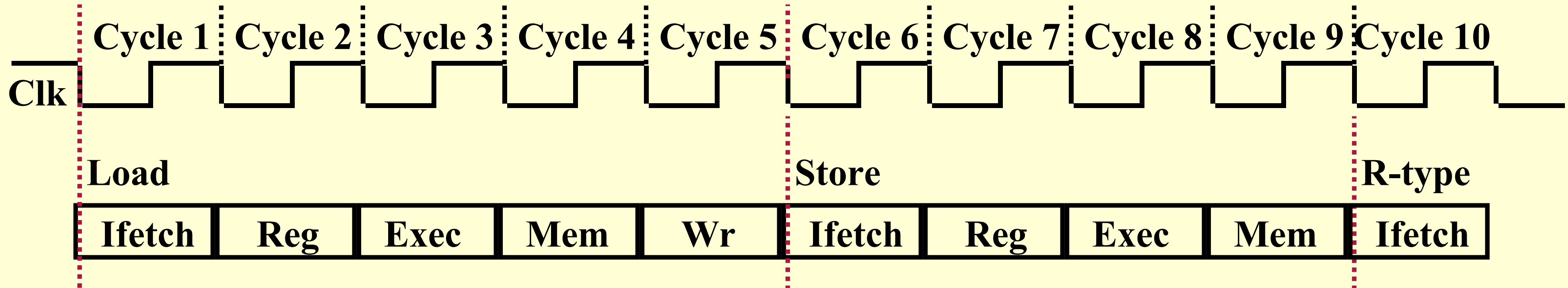
# Single Cycle




- Cycle time long enough for longest instruction
- Shorter instructions waste time
- No overlap

Figure: Dave Patterson

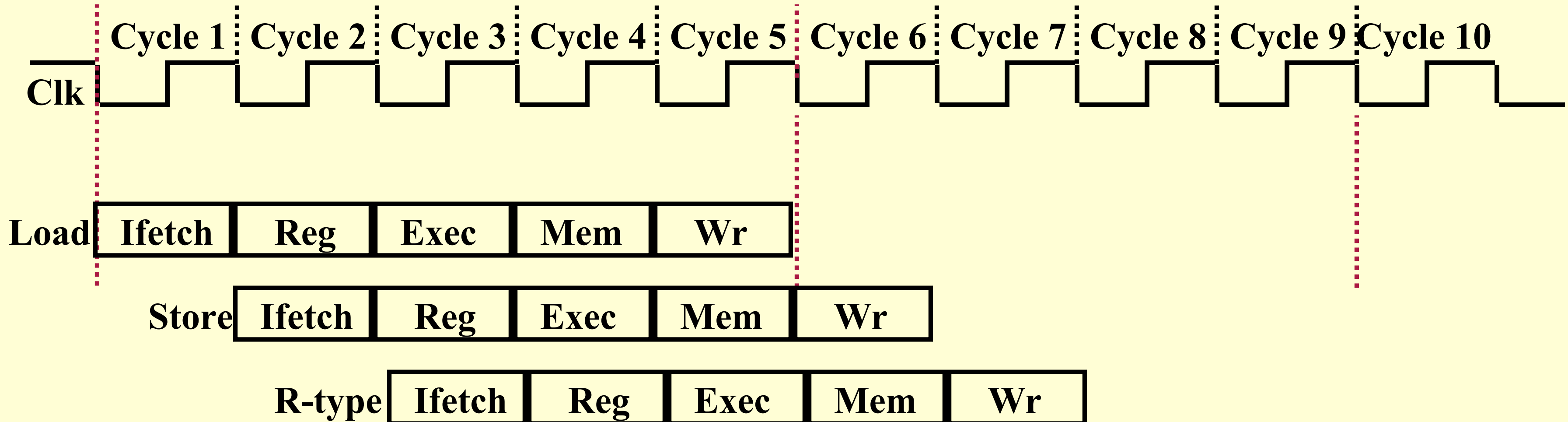
# Multiple Cycle




- Cycle time long enough for longest stage
- Shorter stages waste time
- Shorter instructions can take fewer cycles
- No overlap

Figure: Dave Patterson

# Pipeline




- Cycle time long enough for longest stage
- Shorter stages waste time
- No additional benefit from shorter instructions
- Overlap instruction execution

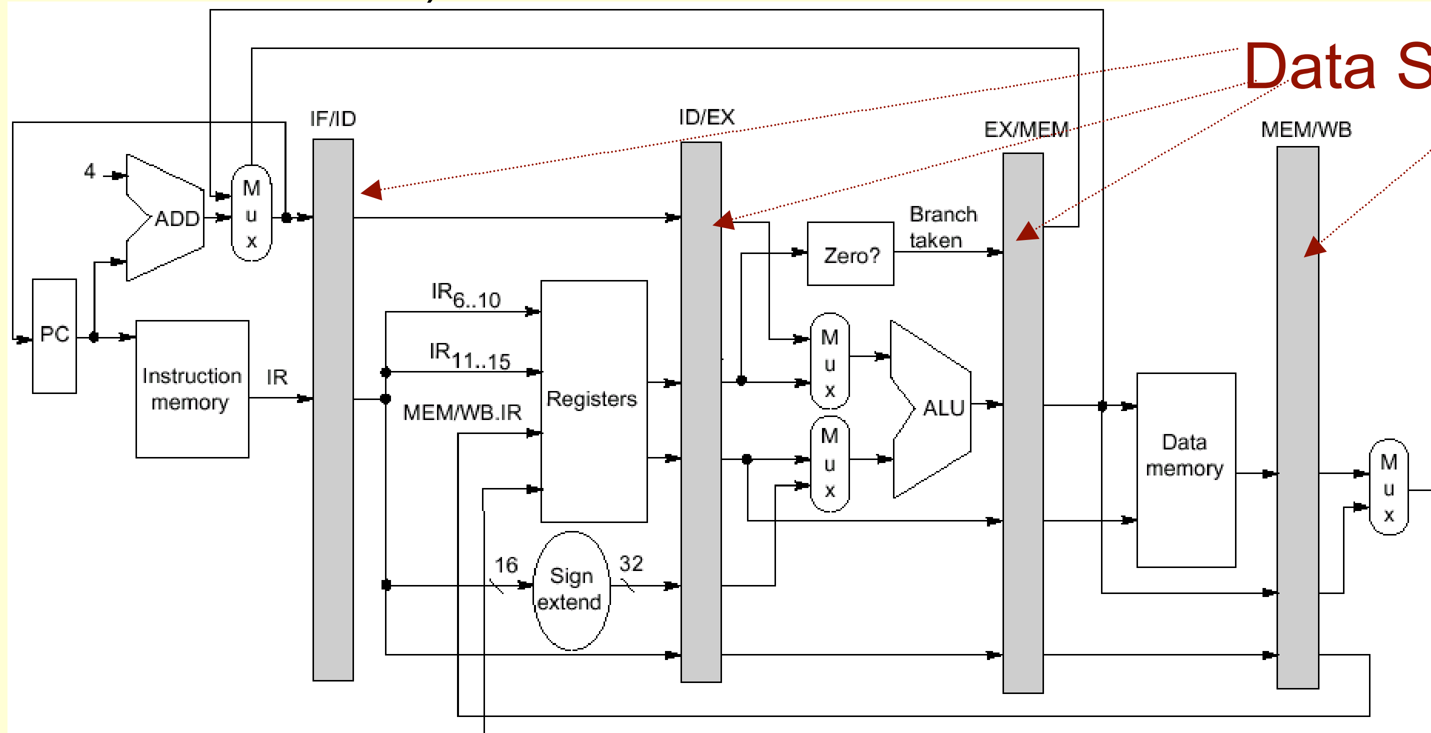
Figure: Dave Patterson

# Pipeline Performance

- Pipeline increases the instruction throughput
  - not execution time of an individual instruction
- An individual instruction can be **slower**:
  - Additional pipeline control
  - Imbalance among pipeline stages
- Suppose we execute 100 instructions:
  - Single Cycle Machine
    - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
  - Multi-cycle Machine
    - 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
  - Ideal 5 stages pipelined machine
    - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
- Lose performance due to fill and drain

# Pipeline Datapath

- Every stage must be completed in one clock cycle to avoid stalls
- Values must be latched to ensure correct execution of instructions
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction)



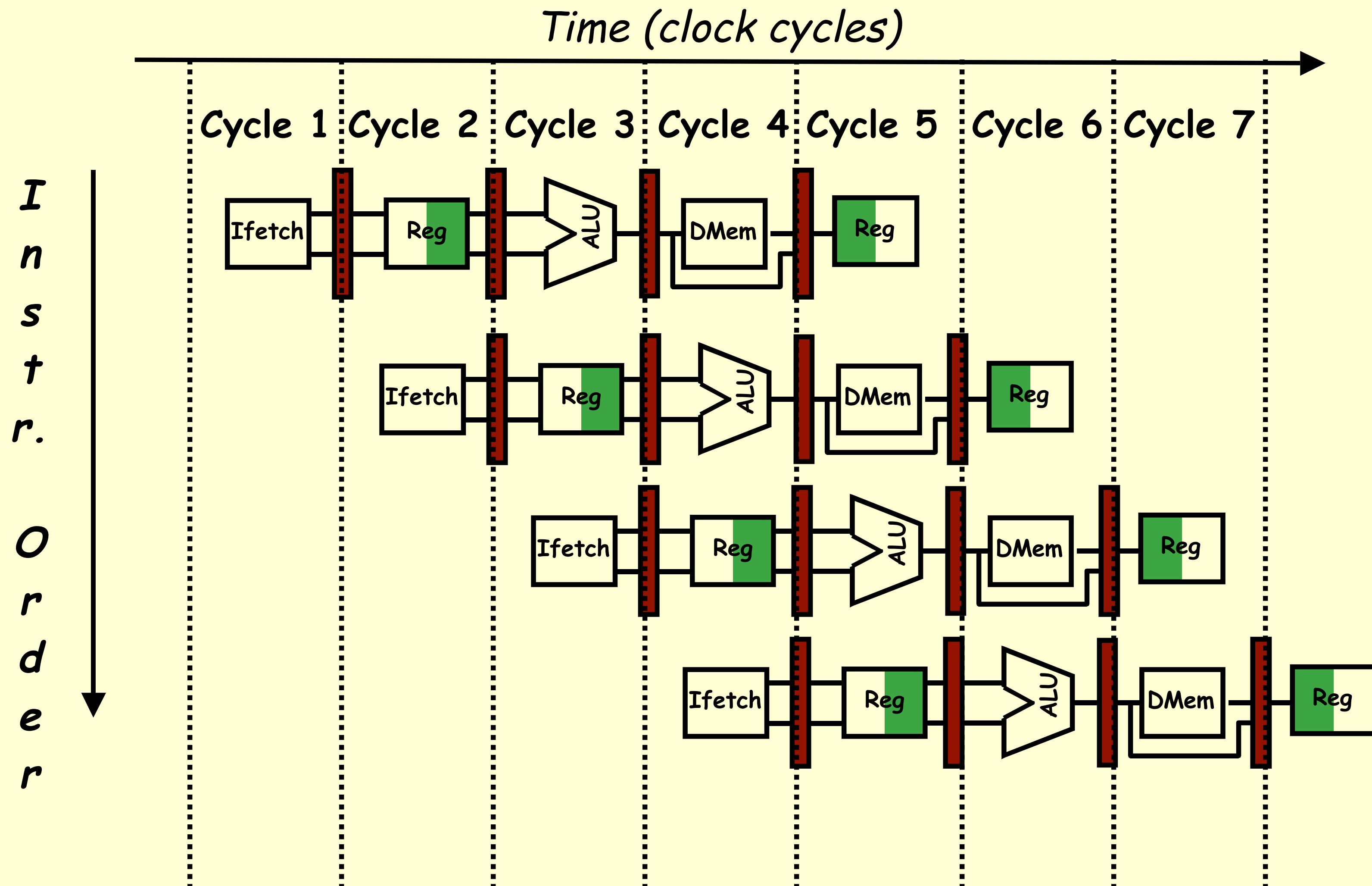
# Pipeline Stage Interface

| Stage | Any Instruction | | |
|---|---|---|---|
| **IF** | IF/ID.IR ←MEM[PC] ;<br>IF/ID.NPC,PC ← ( if ( (EX/MEM.opcode == branch) & EX/MEM.cond)<br>{EX/MEM.ALUOutput } else { PC + 4 } ) ; | | |
| **ID** | ID/EX.A = Regs[IF/ID. IR $_{6..10}$]; ID/EX.B ←Regs[IF/ID. IR $_{11..15}$];<br>ID/EX.NPC ←IF/ID.NPC ; ID/EX.IR ←IF/ID.IR;<br>ID/EX.Imm ← (IF/ID. IR $_{16}$) $^{16}$ ## IF/ID. IR $_{16..31}$; | | |
| | ***ALU*** | ***Load or Store*** | ***Branch*** |
| **EX** | EX/MEM.IR = ID/EX.IR;<br>EX/MEM. ALUOutput ← ID/EX.A func ID/EX.B;<br>Or<br>EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;<br>EX/MEM.cond ← 0; | EX/MEM.IR ← ID/EX.IR;<br>EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;<br>EX/MEM.cond ← 0;<br>EX/MEM.B ←ID/EX.B; | EX/MEM.ALUOutput ← ID/EX.NPC + ID/EX.Imm;<br>EX/MEM.cond ← (ID/EX.A op 0); |
| **MEM** | MEM/WB.IR ←EX/MEM.IR;<br>MEM/WB.ALUOutput ← EX/MEM.ALUOutput; | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] ;<br>Or<br>Mem[EX/MEM.ALUOutput] ← EX/MEM.B ; | |
| **WB** | Regs[MEM/WB. IR $_{16..20}$] ← EM/WB.ALUOutput;<br>Or<br>Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.ALUOutput ; | For load only:<br>Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.LMD; | |

# Pipeline Hazards

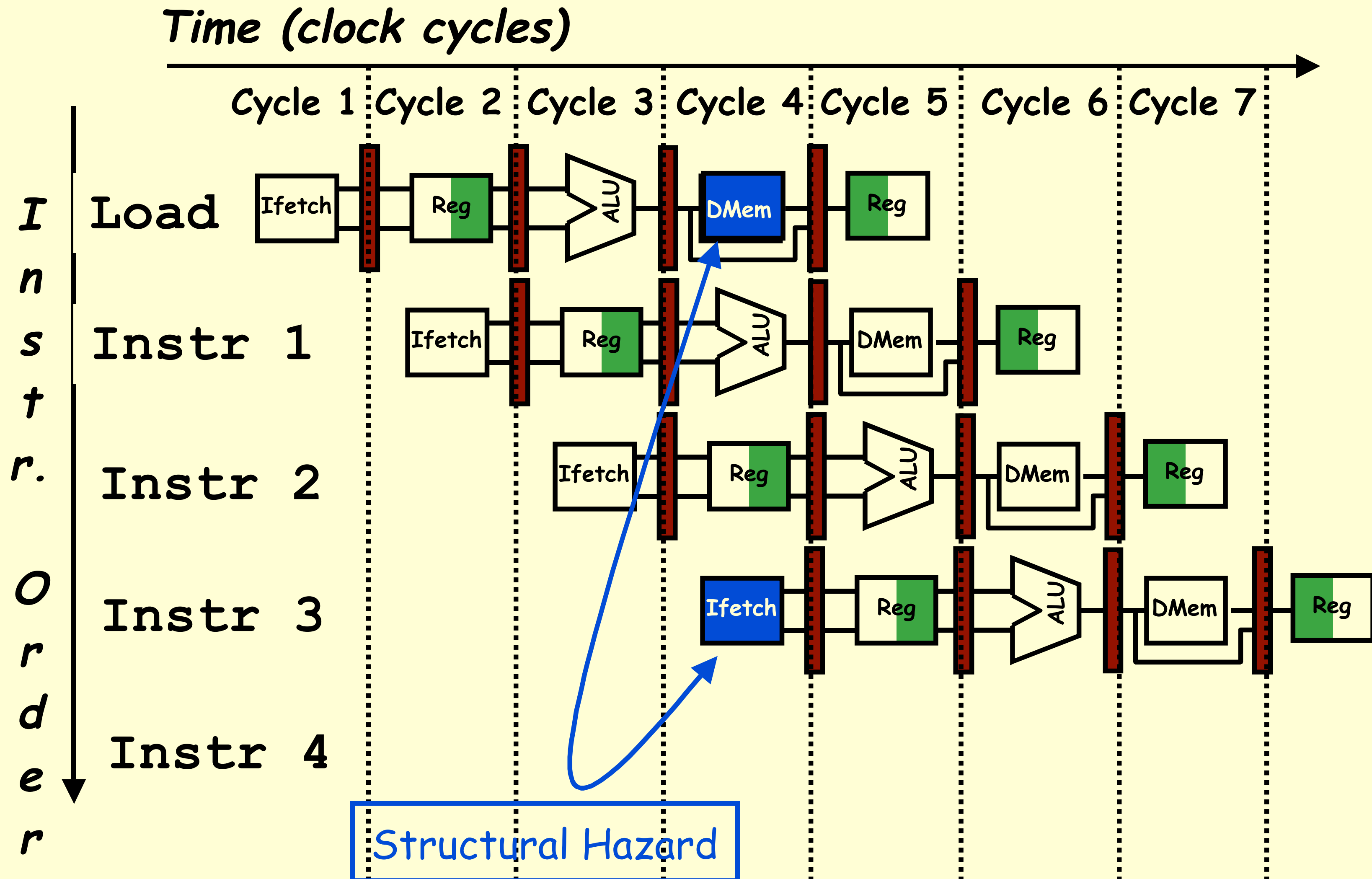
- Cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types
  - Structural hazard: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - Data hazard: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazard: attempt to make a decision before condition is evaluated
    - branch instructions
- Hazards can always be resolved by waiting

# Visualizing Pipelining




Slide: David Culler

# Example: One Memory Port/Structural Hazard

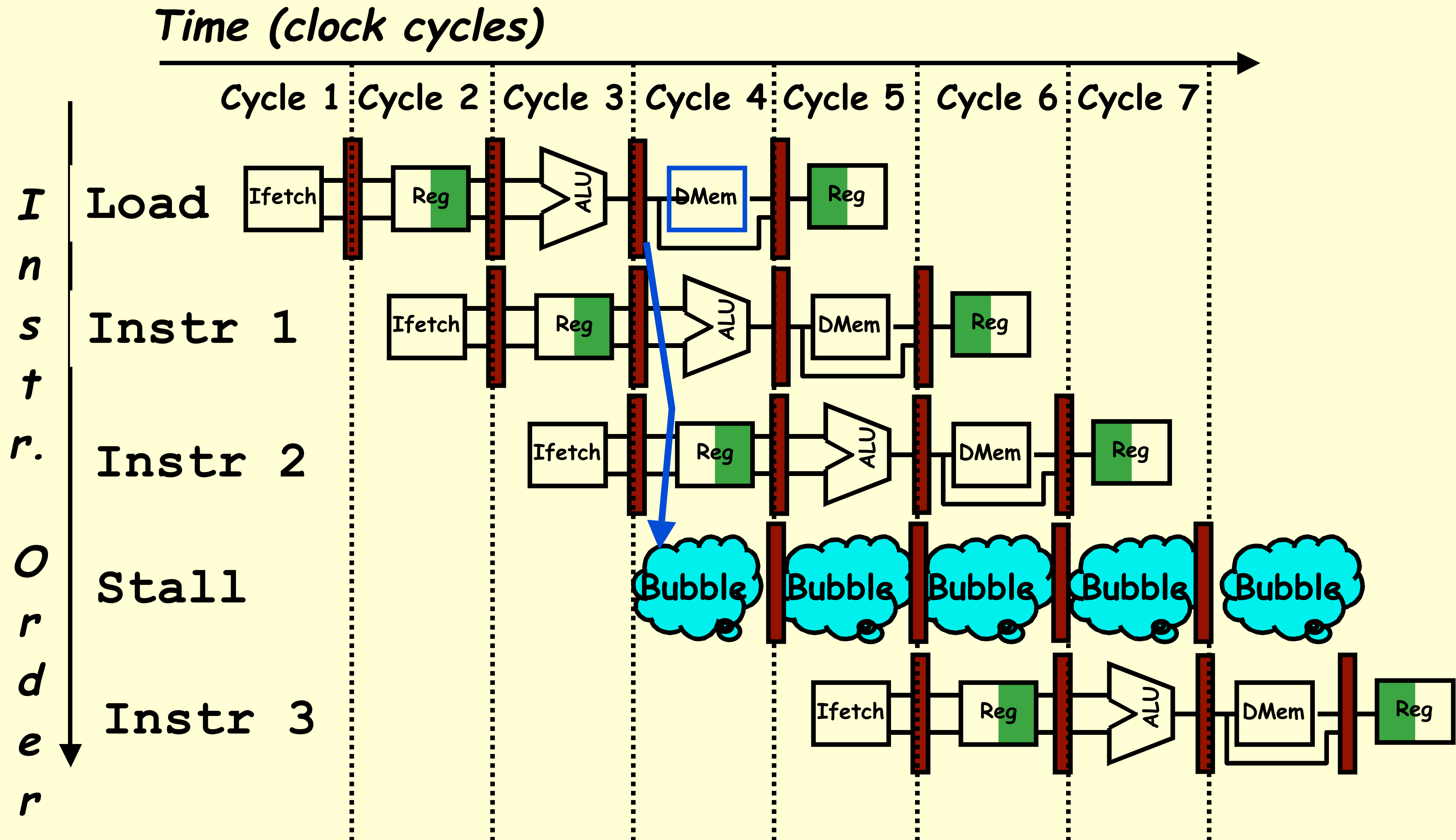



Slide: David Culler

# Resolving Structural Hazards

1. Wait
   – Must detect the hazard
     • Easier with uniform ISA
   – Must have mechanism to stall
     • Easier with uniform pipeline organization
2. Throw more hardware at the problem
   – Use instruction & data cache rather than direct access to memory

# Detecting and Resolving Structural Hazard




Slide: David Culler

# Stalls & Pipeline Performance

$$\text{Pipelining Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Ideal CPI pipelined} = 1$$

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}$$

$$= 1 + \text{Pipeline stall cycles per instruction}$$

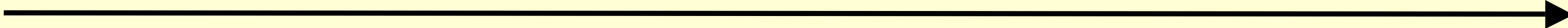
$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Assuming all pipeline stages are balanced

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

# Data Hazards

*Time (clock cycles)*



IF ID/RF EX MEM WB

*Instr. Order*

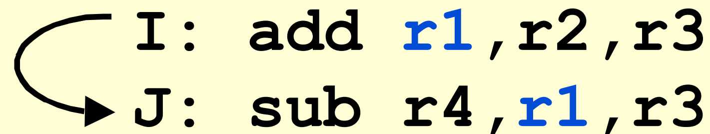
```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or  r8,r1,r9
xor r10,r1,r11
```



Slide: David Culler

# Three Generic Data Hazards

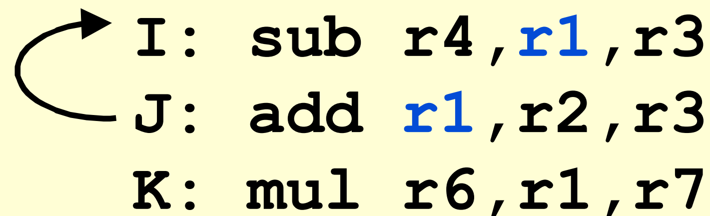
- Read After Write (RAW)
  $Instr_J$ tries to read operand before $Instr_I$ writes it




- Caused by a "Data Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

# Three Generic Data Hazards

- Write After Read (WAR)
  $Instr_J$ writes operand before $Instr_I$ reads it

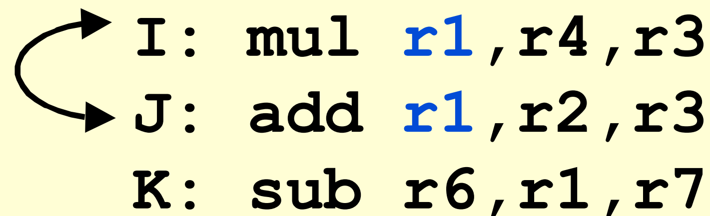



- Called an "anti-dependence" in compilers.
  - This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Slide: David Culler

# Three Generic Data Hazards

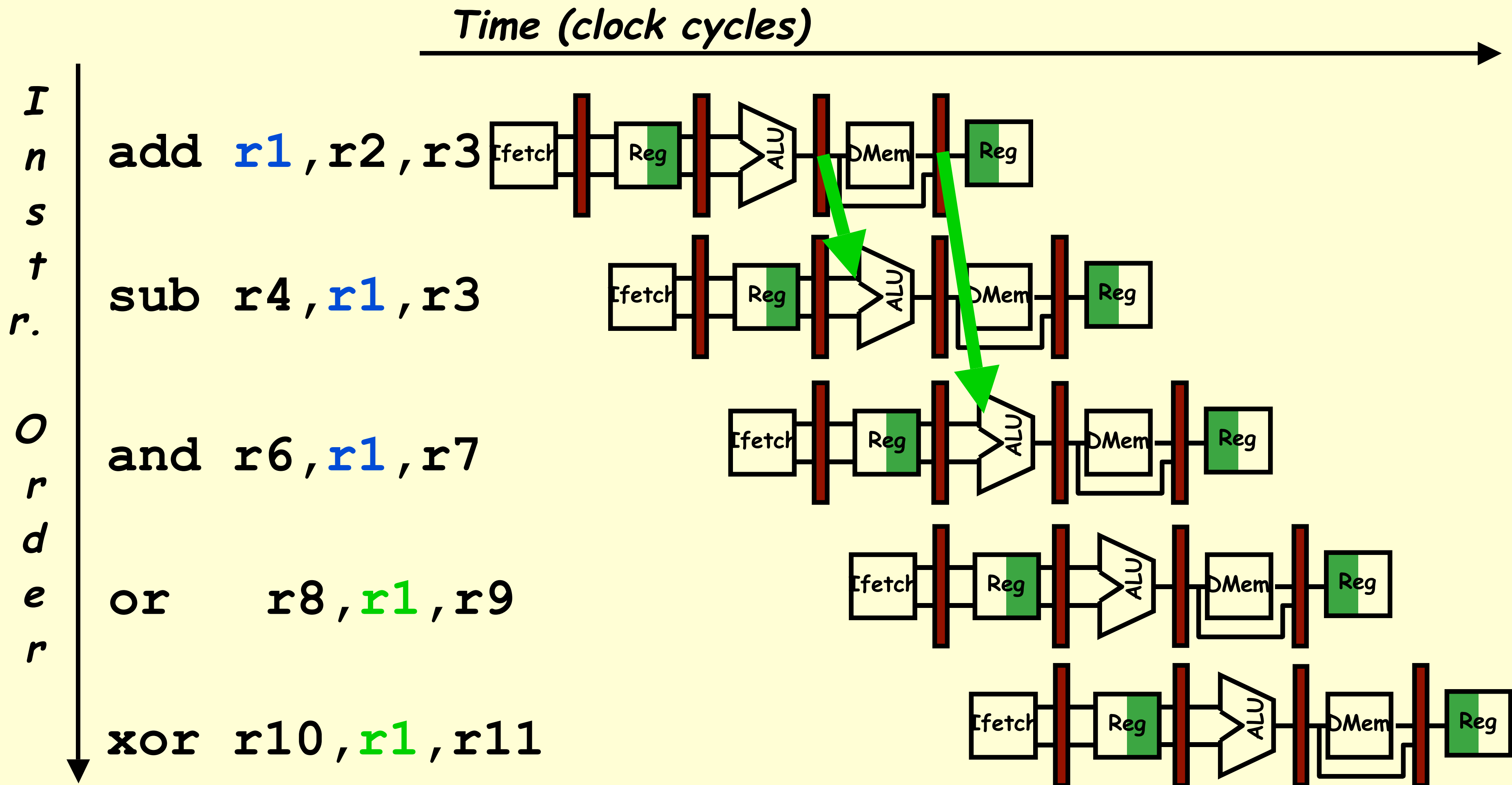
- Write After Write (WAW)
  $Instr_J$ writes operand before $Instr_I$ writes it.




- Called an "output dependence" in compilers
  - This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Do see WAR and WAW in more complicated pipes

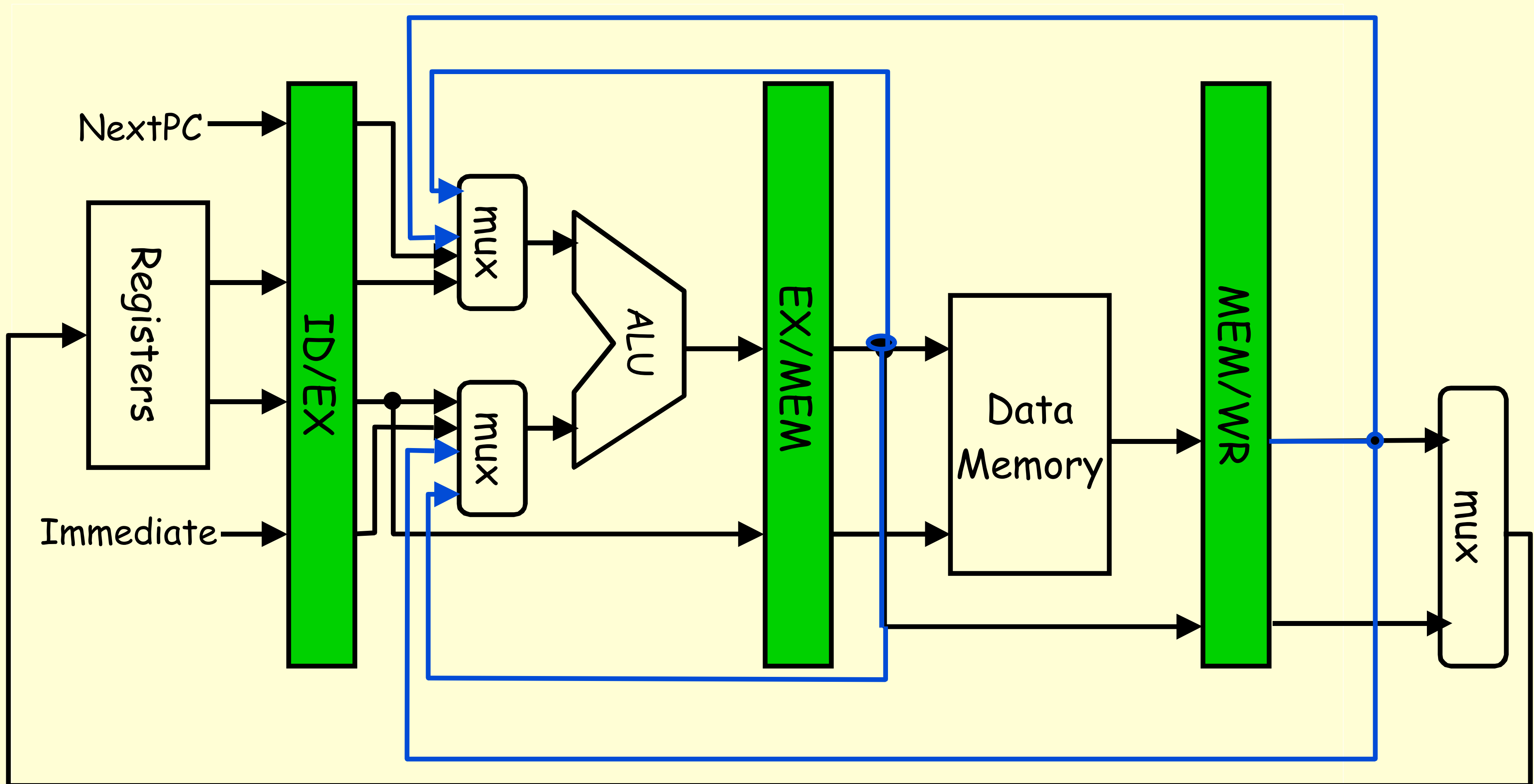
Slide: David Culler

# Forwarding to Avoid Data Hazard



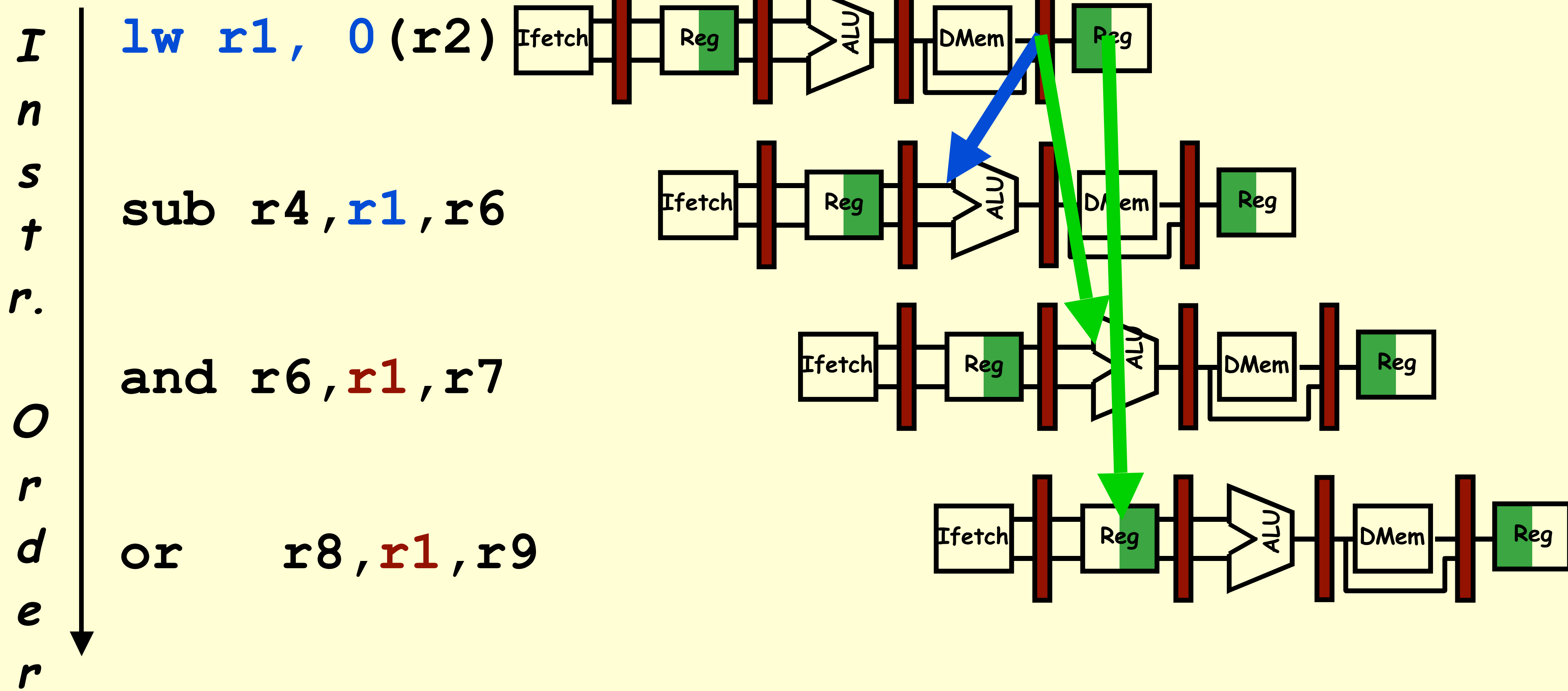

Slide: David Culler

# HW Change for Forwarding




Slide: David Culler

# Data Hazard Even with Forwarding




Slide: David Culler

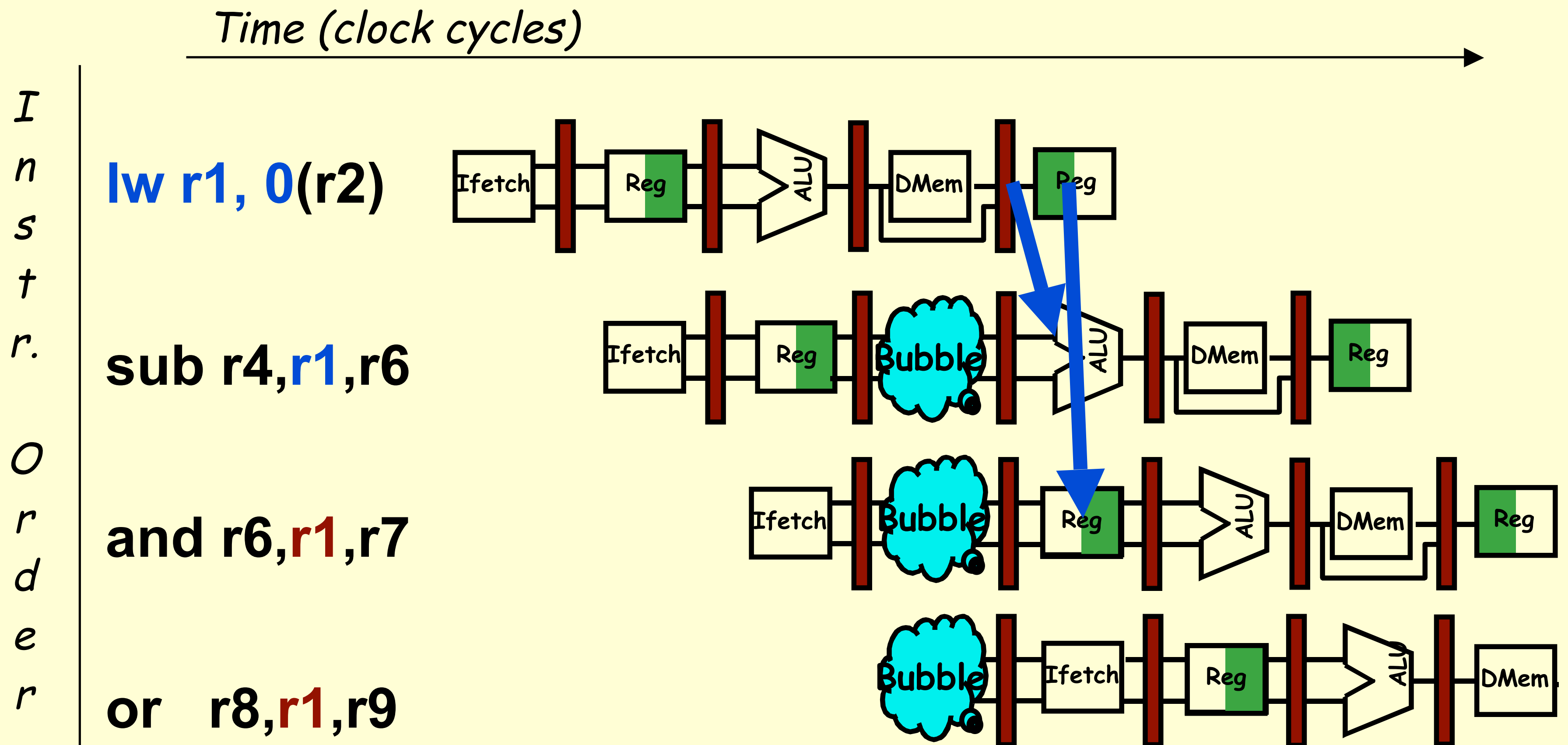
# Resolving Load Hazards

- Adding hardware? How? Where?
- Detection?
- Compilation techniques?

- What is the cost of load delays?

Slide: David Culler

# Resolving the Load Data Hazard




How is this different from the instruction issue stall?

Slide: David Culler

# Software Scheduling to Avoid Load Hazards

Try producing fast code for

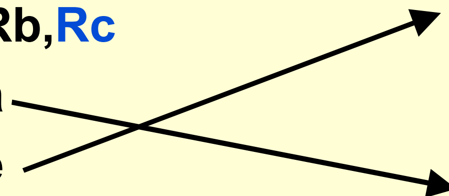
a = b + c;

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:

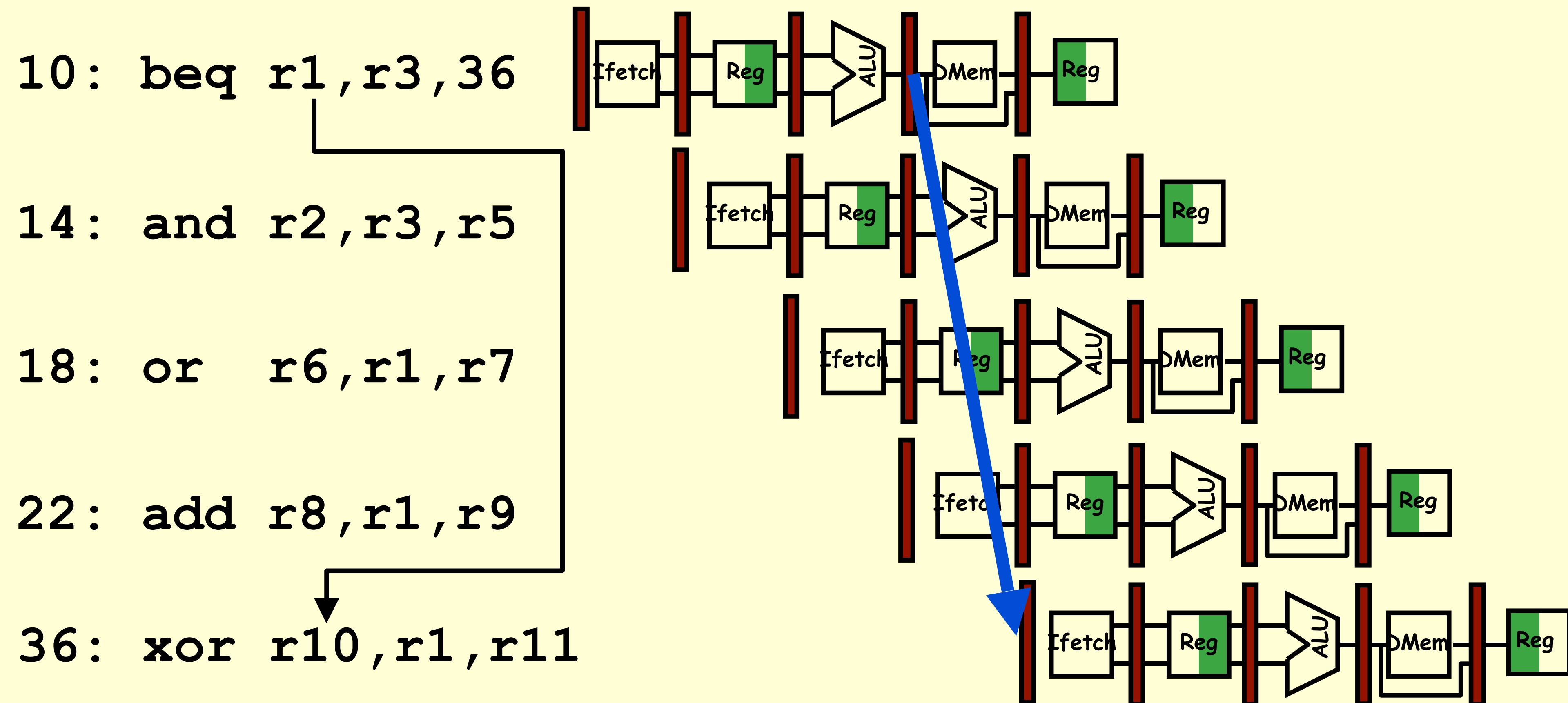
```
LW   Rb,b
LW   Rc,c
LW   Re,e
ADD  Ra,Rb,Rc
LW   Rf,f
SW   a,Ra
SUB  Rd,Re,Rf
SW   d,Rd
```

Slide: David Culler

# Instruction Set Connection

- What is exposed about this organizational hazard in the instruction set?
- k cycle delay?
  - bad, CPI is not part of ISA
- k instruction slot delay
  - load should not be followed by use of the value in the next k instructions
- Nothing, but code can reduce run-time delays
- MIPS did the transformation in the assembler

# Control Hazard on Branches Three Stage Stall




Slide: David Culler

# 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 ≠ 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

Slide: David Culler

# Pipelined MIPS Datapath

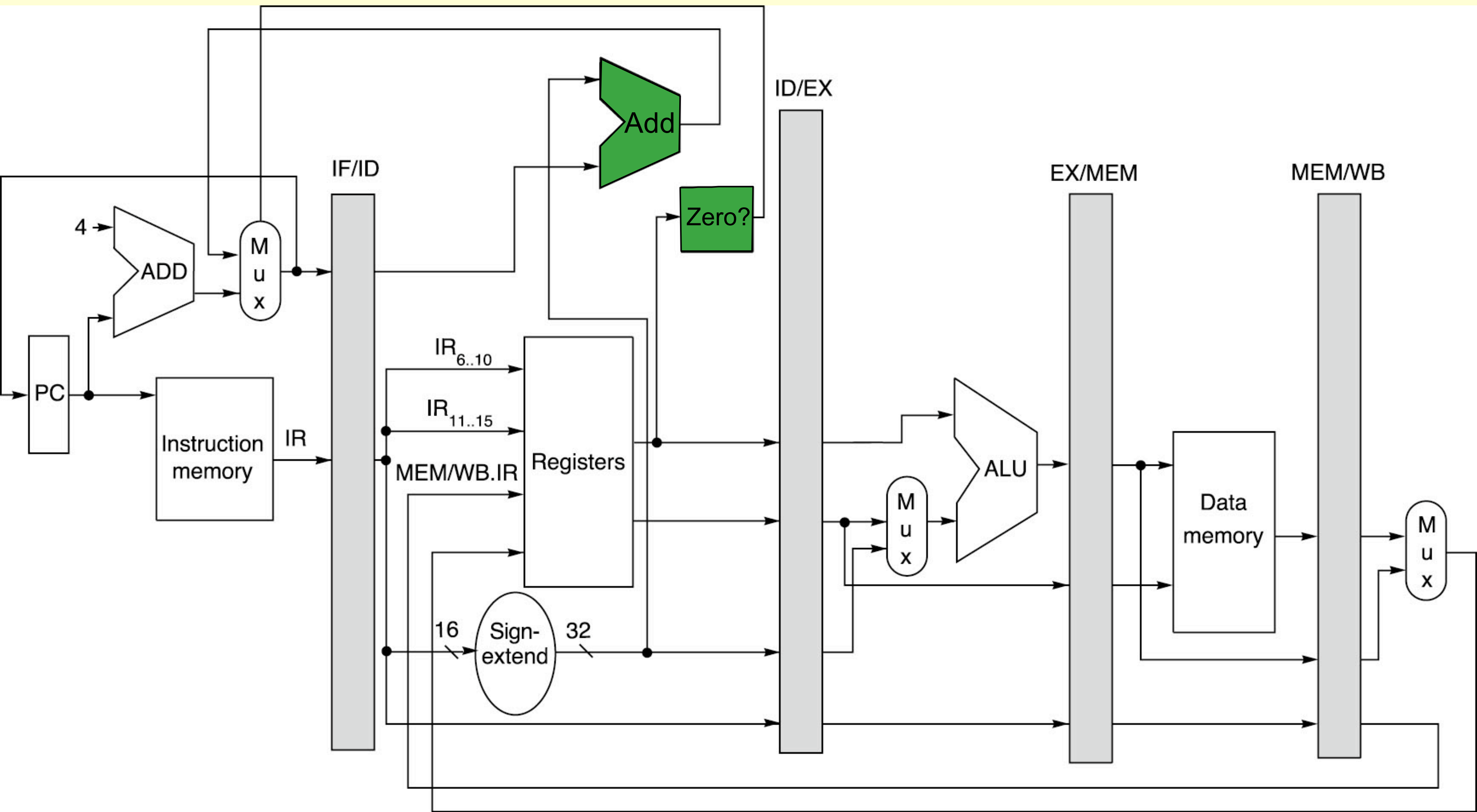



Figure: Dave Patterson

# 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 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

4. Delayed Branch
   - Define branch to take place AFTER a following instruction

     branch instruction
     
     sequential successor$_1$
     
     sequential successor$_2$
     
     ........
     
     sequential successor$_n$
     
     **Branch delay of length *n***
     
     ........
     
     branch target if taken

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

# Delayed Branch

- Where to get branch delay slot instructions?
  - 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
- Compiler effectiveness for single 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
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Slide: David Culler

# Example: Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}}$$

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

Assume:

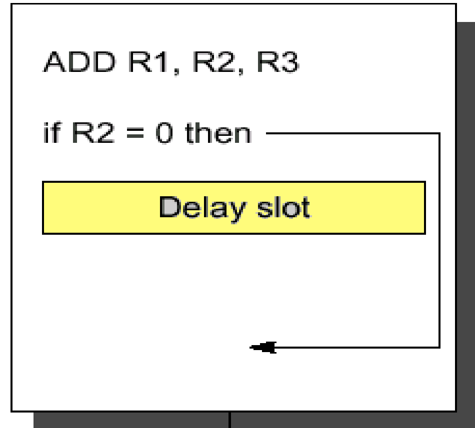
14% Conditional & Unconditional

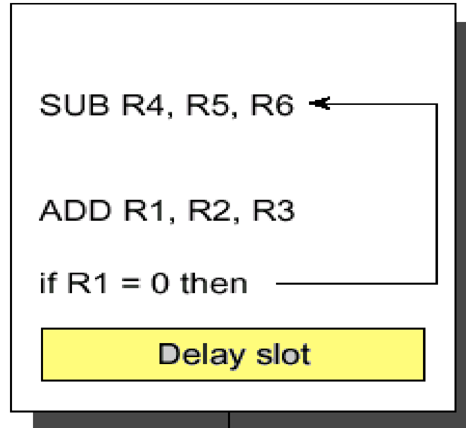
65% Change PC

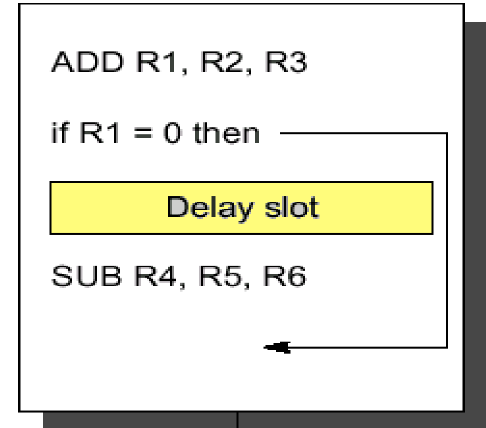
| ***Scheduling Scheme*** | ***Branch Penalty*** | ***CPI*** | ***Pipeline Speedup*** | ***Speedup vs stall*** |
|---|---|---|---|---|
| Stall pipeline | 3.0 | 1.420 | 3.52 | 1.00 |
| Predict taken | 1.0 | 1.140 | 4.39 | 1.25 |
| Predict not taken | 1.0 | 1.091 | 4.58 | 1.30 |
| Delayed branch | 0.5 | 1.007 | 4.97 | 1.41 |

Slide: David Culler

# Scheduling Branch-Delay Slots

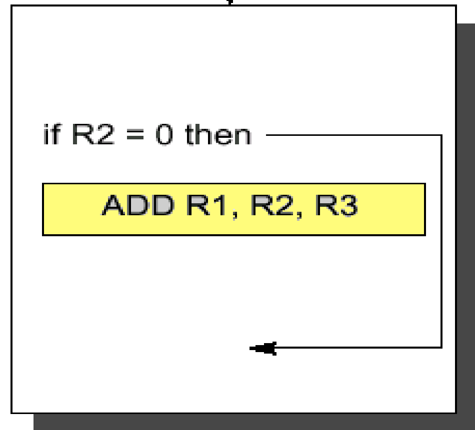


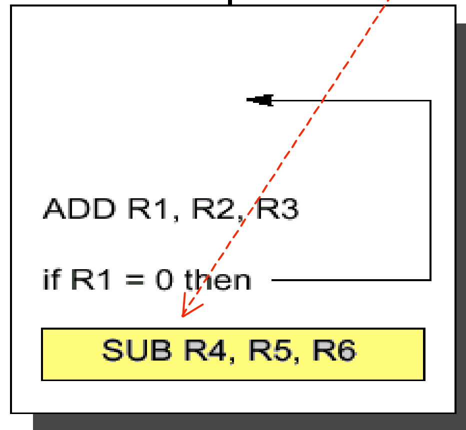



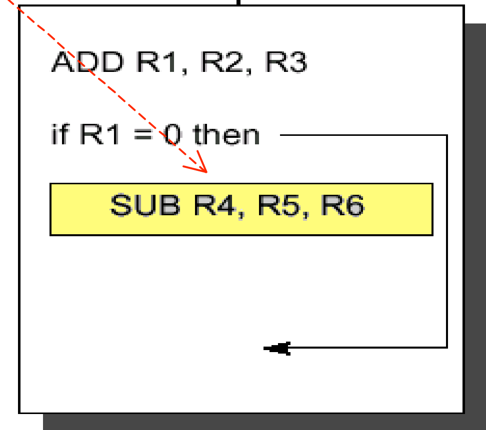




Becomes

R4 must be temp reg.




**Best scenario** | **Good for loops** | **Good taken strategy**

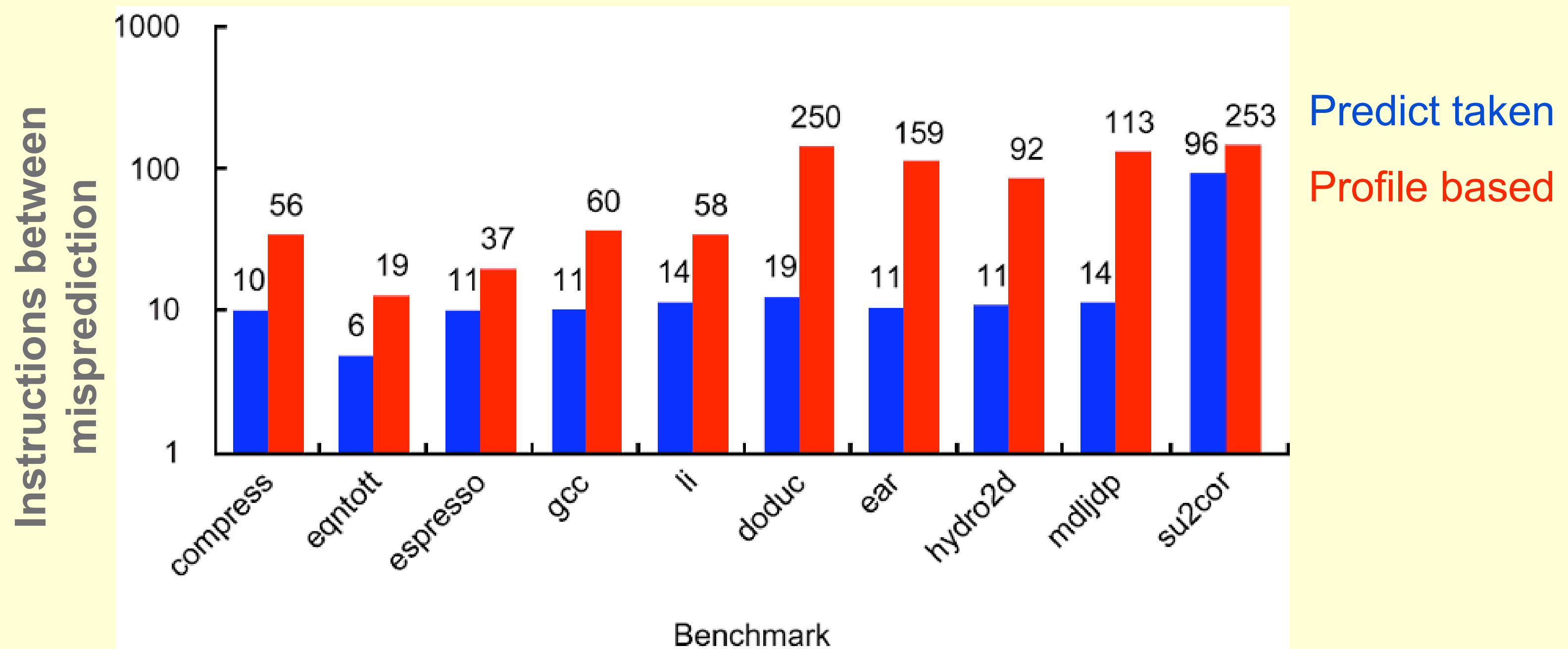
# Branch-Delay Scheduling Requirements

| Scheduling Strategy | Requirements | Improves performance when? |
|---|---|---|
| (a) From before | Branch must not depend on the rescheduled instructions | Always |
| (b) From target | Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions. | When branch is taken. May enlarge programs if instructions are duplicated. |
| (c) From fall through | Must be okay to execute instructions if branch is taken. | When branch is not taken. |

- Limitation on delayed-branch scheduling arise from:
  - Restrictions on instructions scheduled into the delay slots
  - Ability to predict at compile-time whether a branch is likely to be taken
- May have to fill with a no-op instruction
  - Average 30% wasted
- Additional PC is needed to allow safe operation in case of interrupts (more on this later)

# Static Branch Prediction

- Examination of program behavior
  - Assume branch is usually taken based on statistics but misprediction rate still 9%-59%
- Predict on branch direction forward/backward based on statistics and code generation convention
  - Profile information from earlier program runs



# Exception Types

- I/O device request
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Privilege violation
- Hardware and power failure

# Exception Requirements

- Synchronous vs. asynchronous
  - I/O exceptions: Asyncronous
    - Allow completion of current instruction
  - Exceptions within instruction: Synchronous
    - Harder to deal with
- User requested vs. coerced
  - Requested predictable and easier to handle
- User maskable vs. unmaskable
- Resume vs. terminate
  - Easier to implement exceptions that terminate program execution

# Stopping & Restarting Execution

- Some exceptions require restart of instruction
  - e.g. Page fault in MEM stage
- When exception occurs, pipeline control can:
  - Force a trap instruction into the pipeline next IF
  - Until the trap is taken, turn off all writes for the faulting (and later) instructions
  - OS exception-handling routine saves faulting instruction PC

# Stopping & Restarting Execution

- Precise exceptions
  - Instructions before the faulting one complete
  - Instructions after it restart
  - As if execution were serial
- Exception handling complex if faulting instruction can change state before exception occurs
- Precise exceptions simplifies OS
- Required for demand paging

# Exceptions in MIPS

| Pipeline Stage | Problem exceptions occurring |
|---|---|
| IF | Page fault on instruction fetch; misaligned memory access; memory protection violation |
| ID | Undefined or illegal opcode |
| EX | Arithmetic exception |
| MEM | Page fault on data fetch; misaligned memory access; memory protection violation |
| WB | None |

- Multiple exceptions might occur since multiple instructions are executing
  - (LW followed by DIV might cause page fault and an arith. exceptions in same cycle)
- Exceptions can even occur out of order
  - IF page fault before preceeding MEM page fault

**Pipeline exceptions must follow order of execution of faulting instructions not according to the time they occur**

# Precise Exception Handling

- The MIPS Approach:
  - Hardware posts all exceptions caused by a given instruction in a status vector associated with the instruction
  - The exception status vector is carried along as the instruction goes down the pipeline
  - Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off
  - Upon entering the WB stage the exception status vector is checked and the exceptions, if any, will be handled according the time they occurred
  - Allowing an instruction to continue execution till the WB stage is not a problem since all write operations for that instruction will be disallowed

# Instruction Set Complications

- Early-Write Instructions
  - MIPS only writes late in pipeline
  - Machines with multiple writes usually require capability to rollback the effect of an instruction
    - e.g. VAX auto-increment,
  - Instructions that update memory state during execution, e.g. string copy, may need to save & restore temporary registers
- Branching mechanisms
  - Complications from condition codes, predictive execution for exceptions prior to branch
- Variable, multi-cycle operations
  - Instruction can make multiple writes