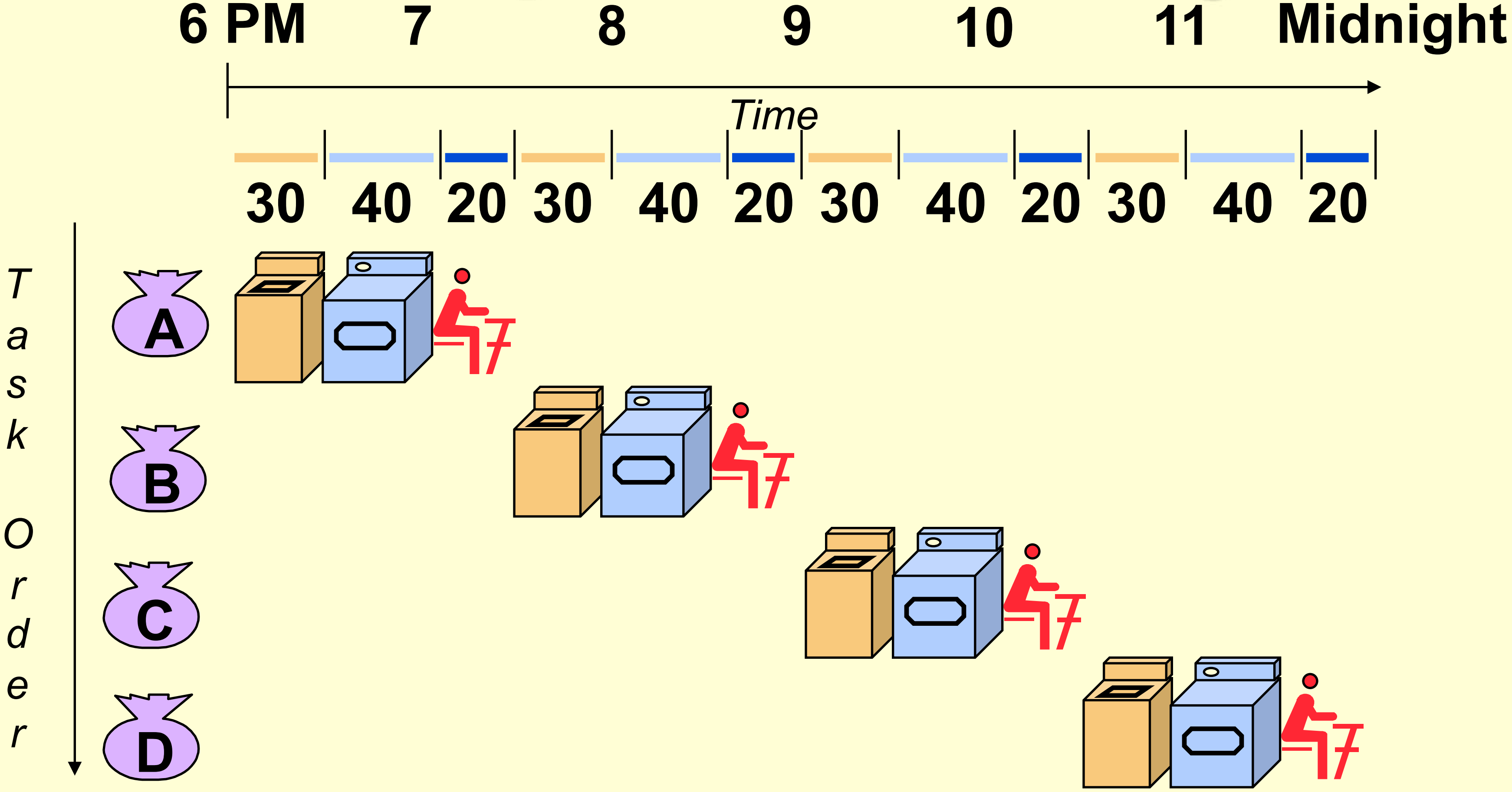
CMSC 611: Advanced Computer Architecture

Pipelining

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science

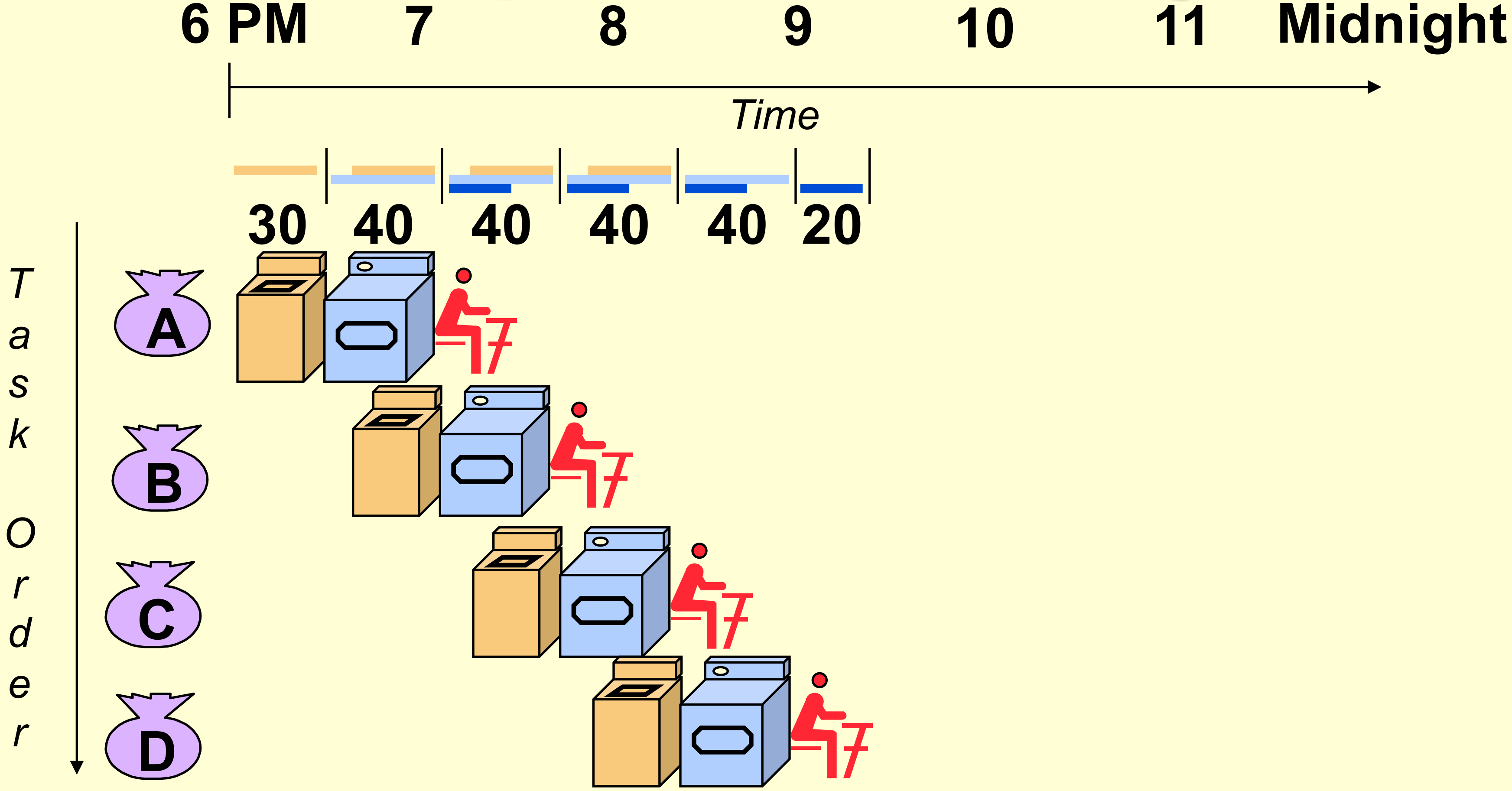
Sequential Laundry



- 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

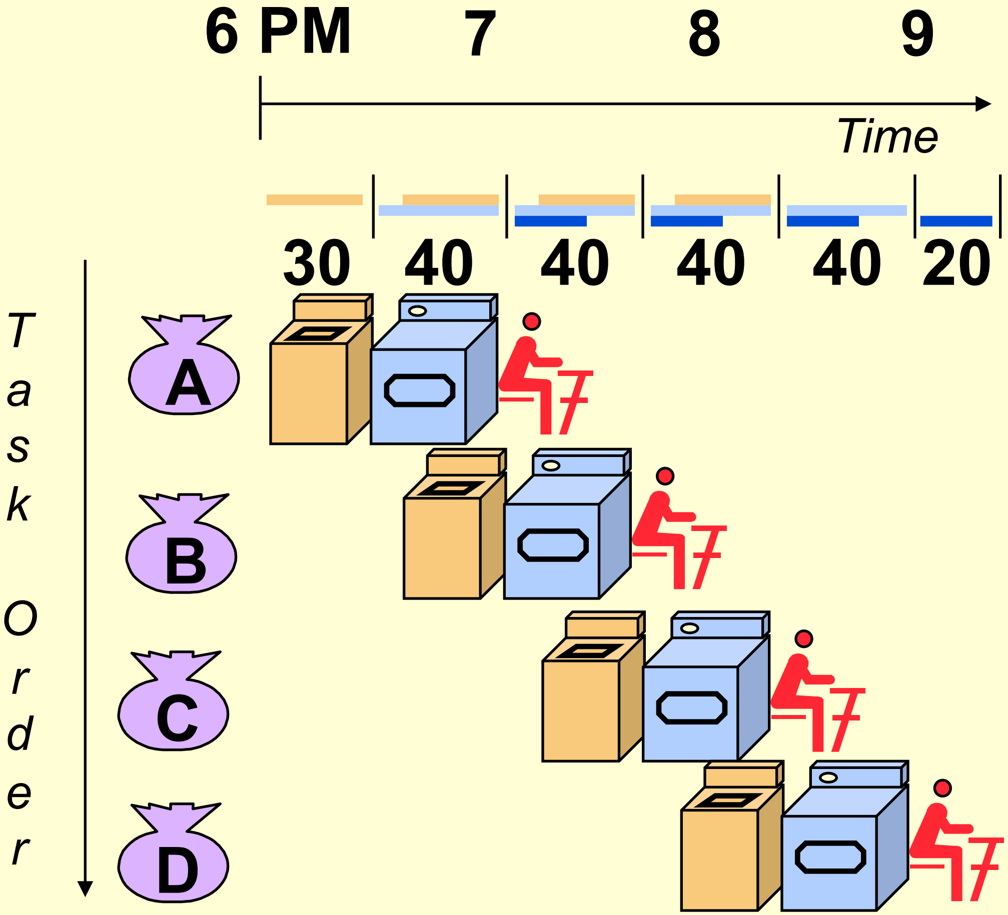
Pipelined Laundry



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

Slide: Dave Patterson

Pipelining Lessons

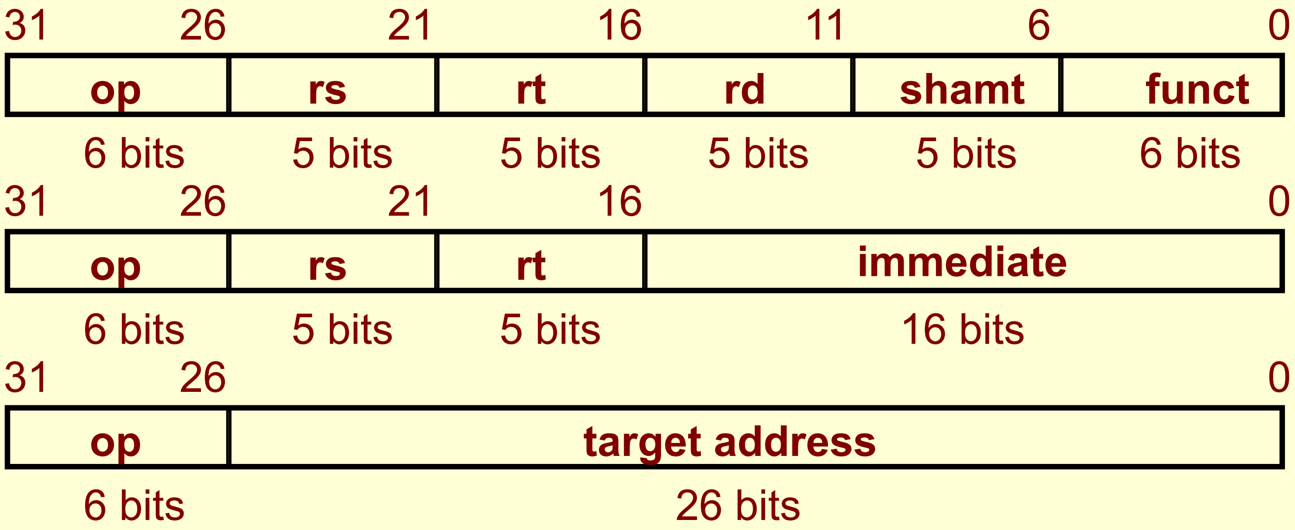


- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- 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

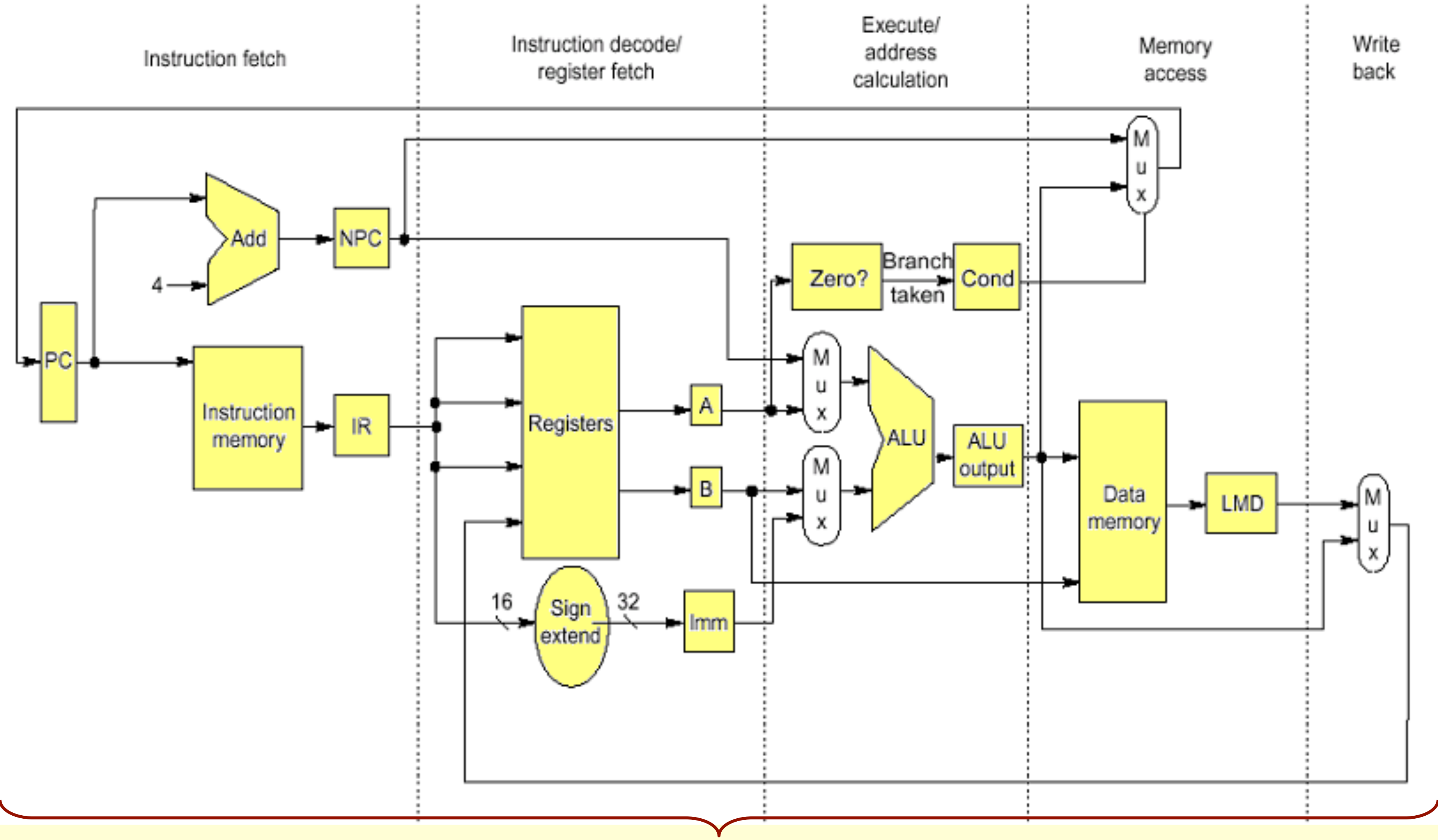


31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

31 26	0
op	**target address**
6 bits	26 bits

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

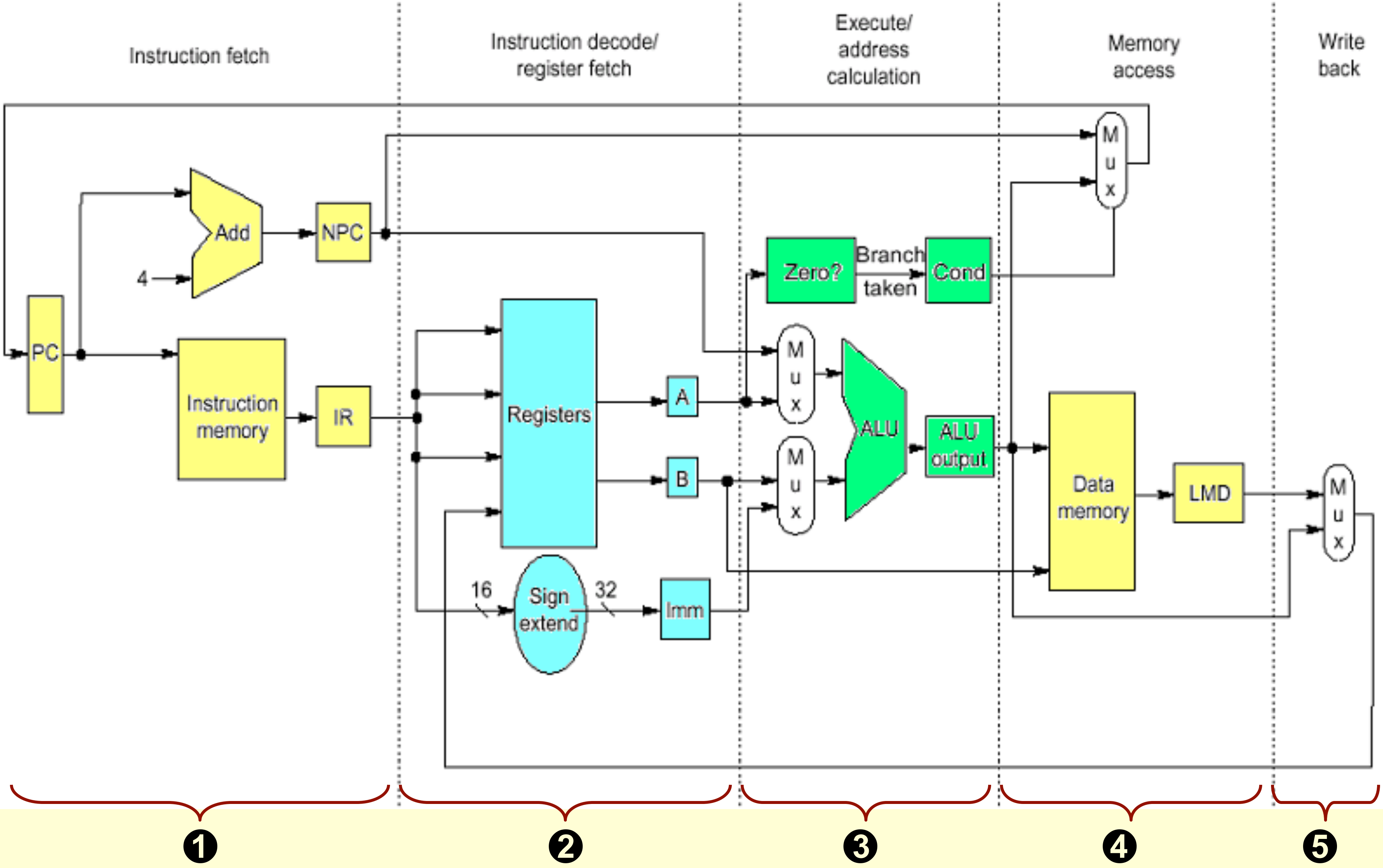
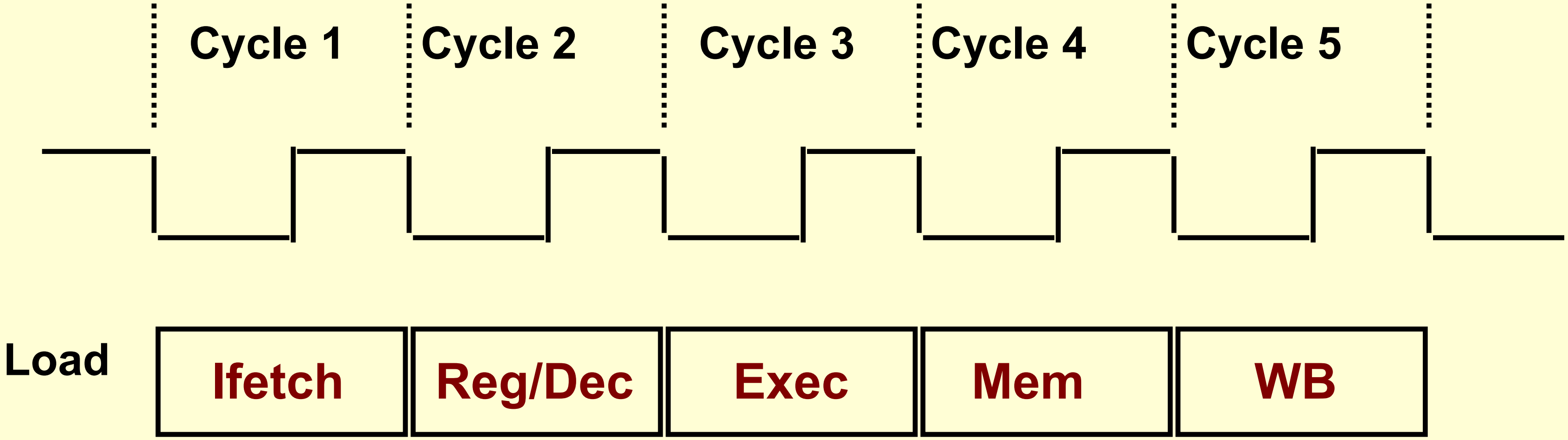


Figure: Dave Patterson

Stages of Instruction Execution



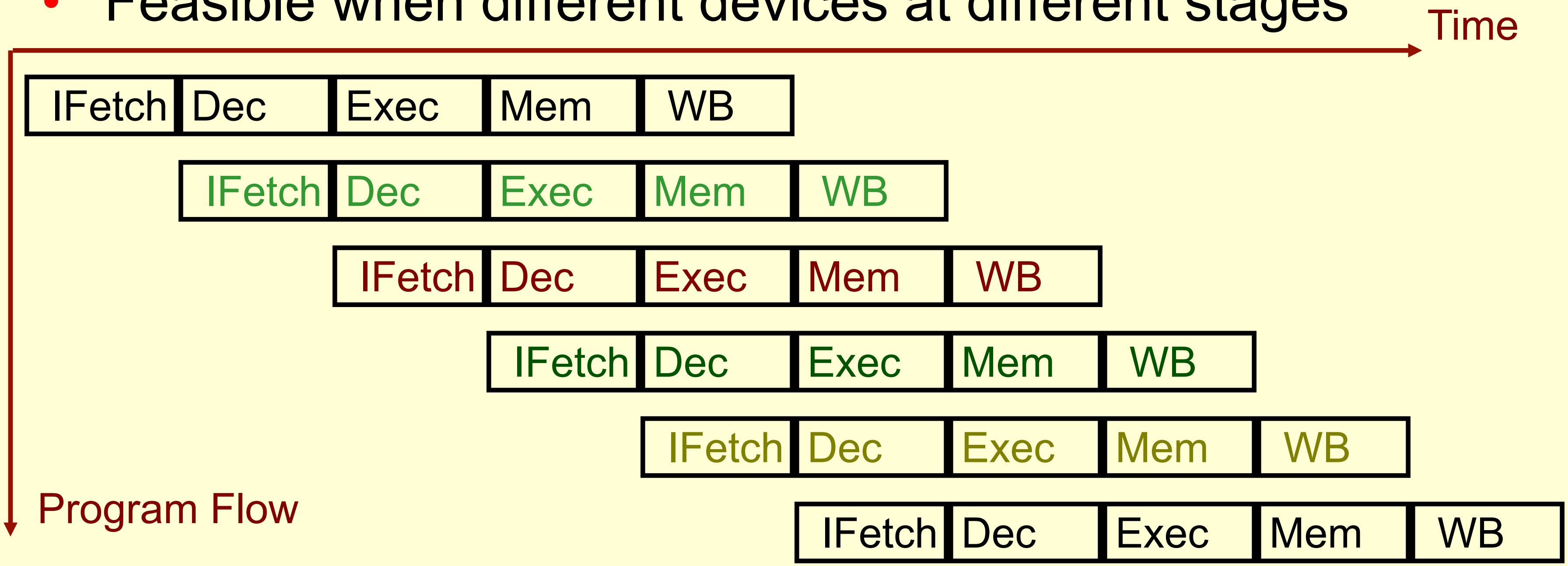
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Load	**Ifetch**	**Reg/Dec**	**Exec**	**Mem**	**WB**

- 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

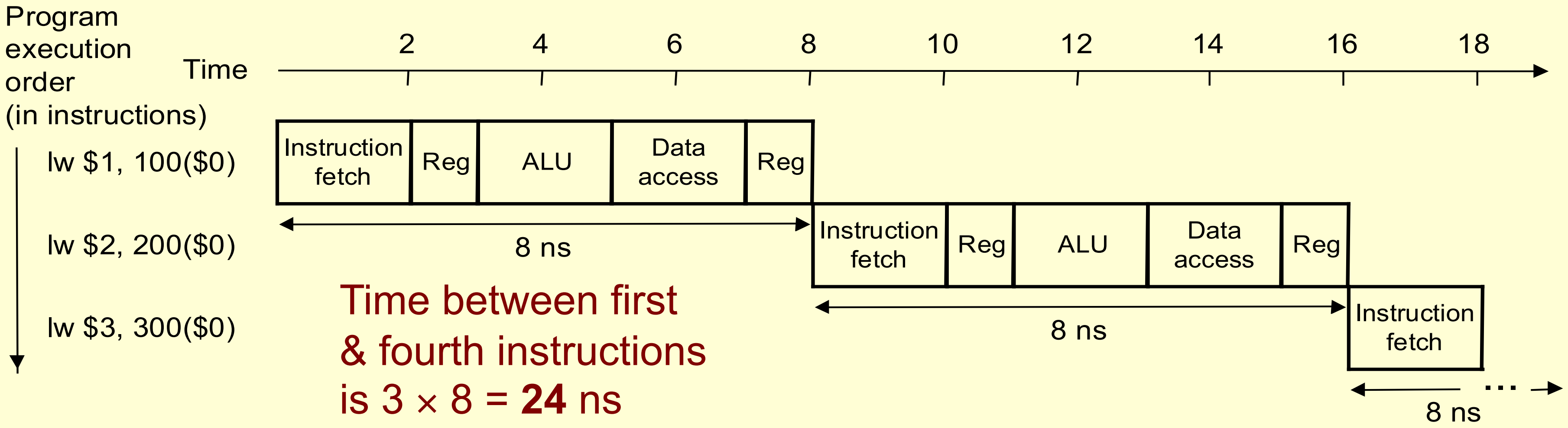


IFetch	Dec	Exec	Mem	WB					
	IFetch	Dec	Exec	Mem	WB				
		IFetch	Dec	Exec	Mem	WB			
			IFetch	Dec	Exec	Mem	WB		
				IFetch	Dec	Exec	Mem	WB	
					IFetch	Dec	Exec	Mem	WB

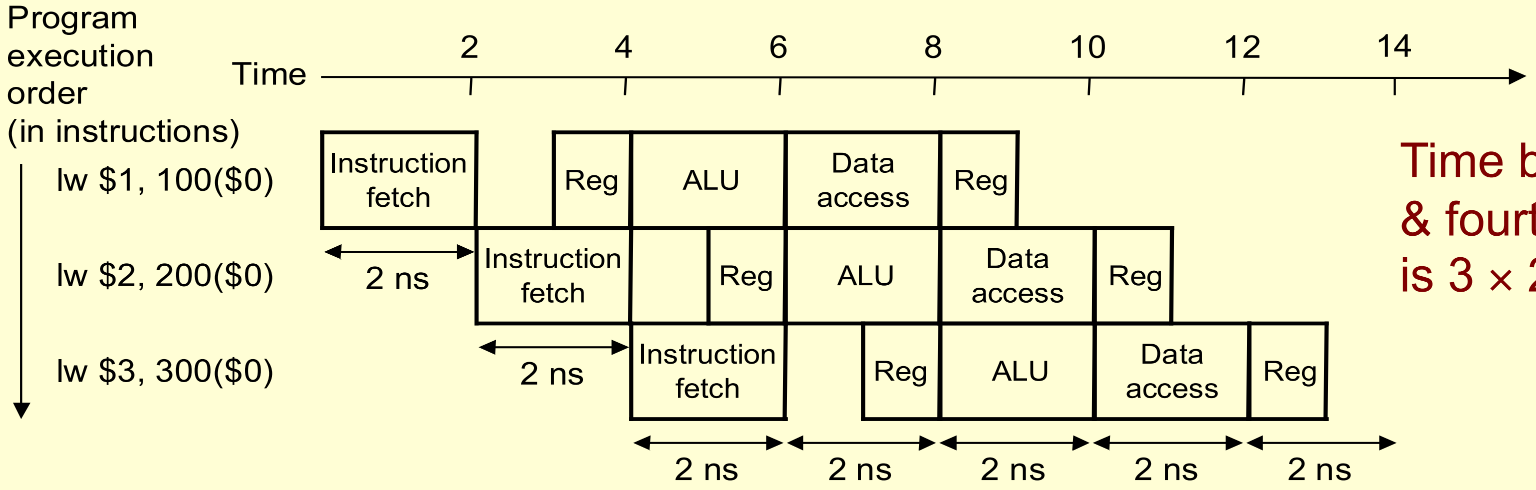
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput

Example of Instruction Pipelining



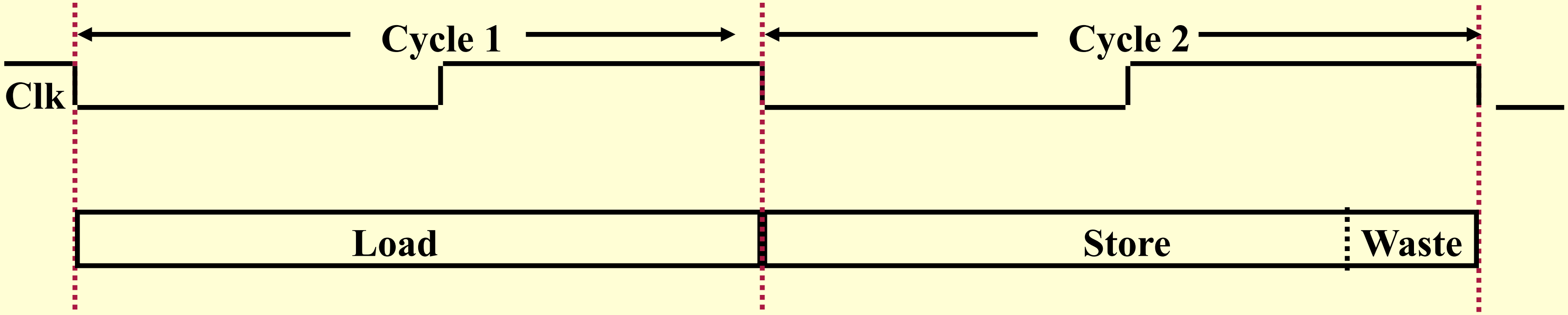
Time between first & fourth instructions is 3 × 8 = **24** ns



Time between first & fourth instructions is 3 × 2 = **6** ns

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

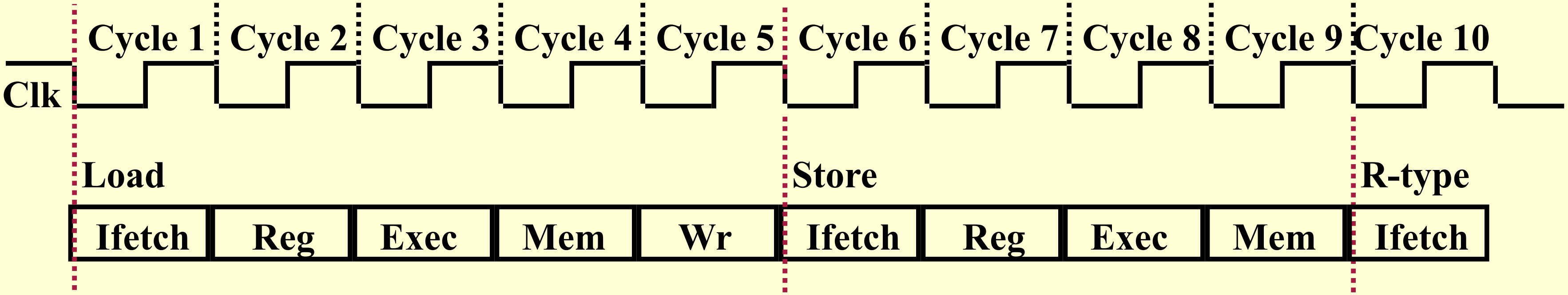
Single Cycle



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

Figure: Dave Patterson

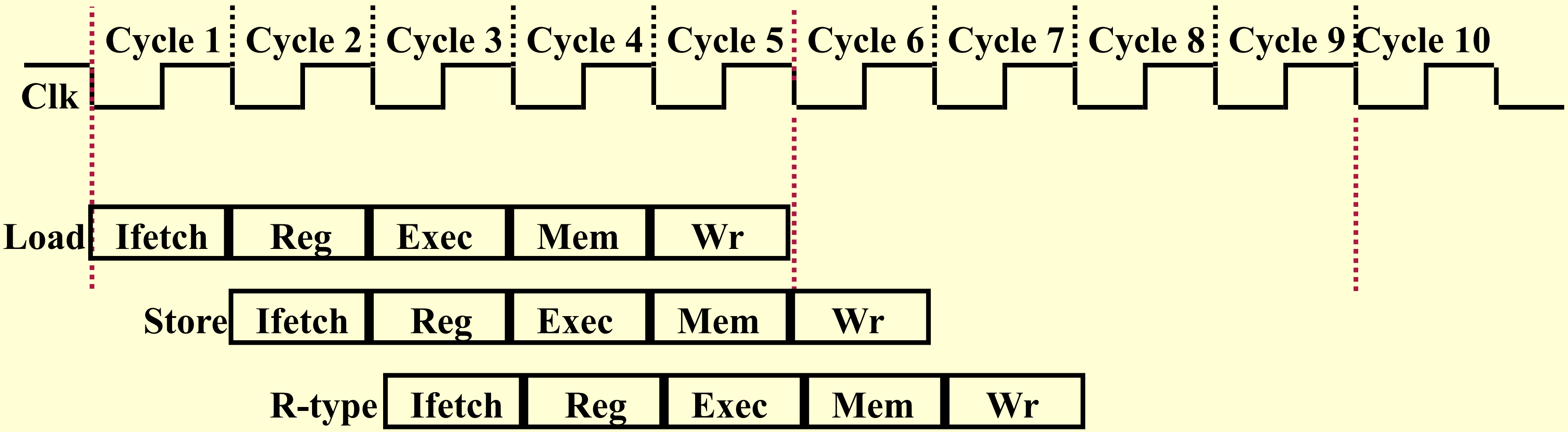
Multiple Cycle



- 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

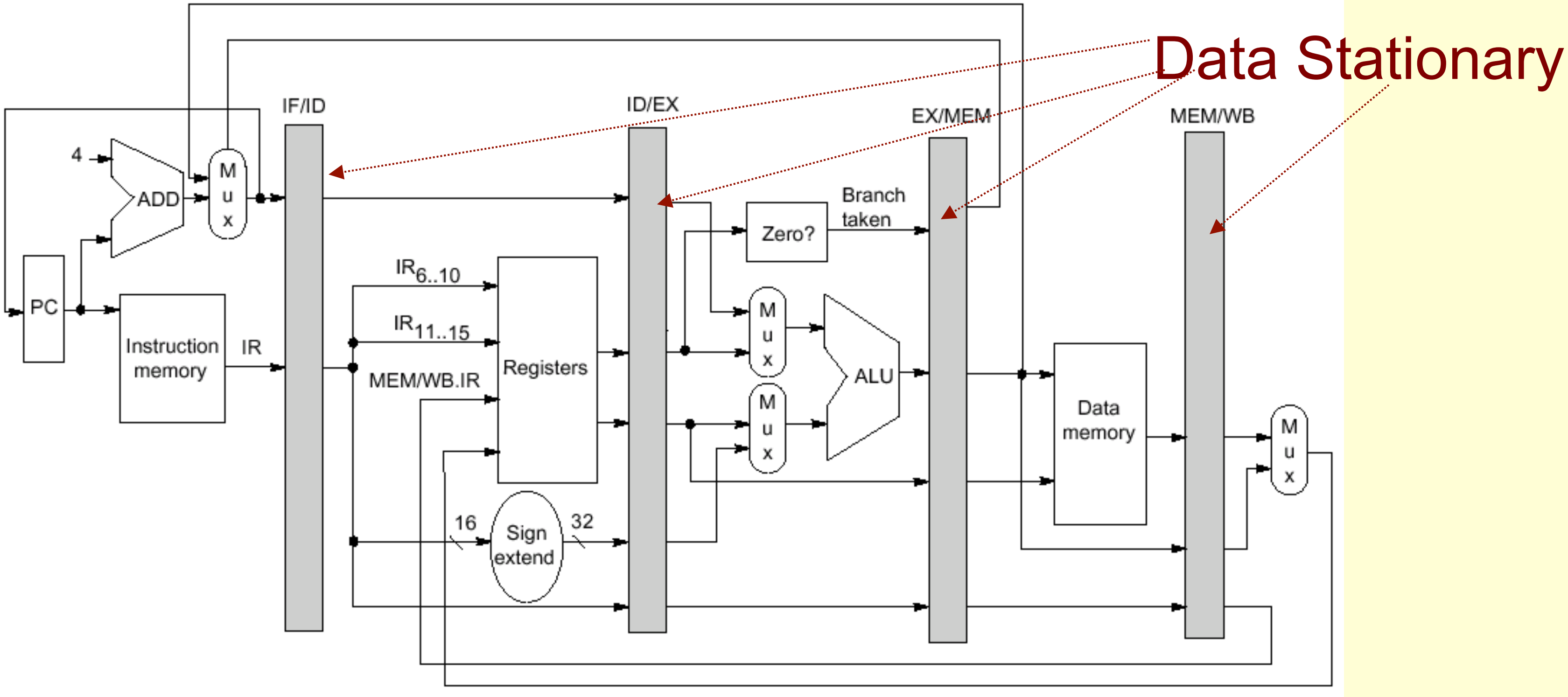
Figure: Dave Patterson

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)



Data Stationary

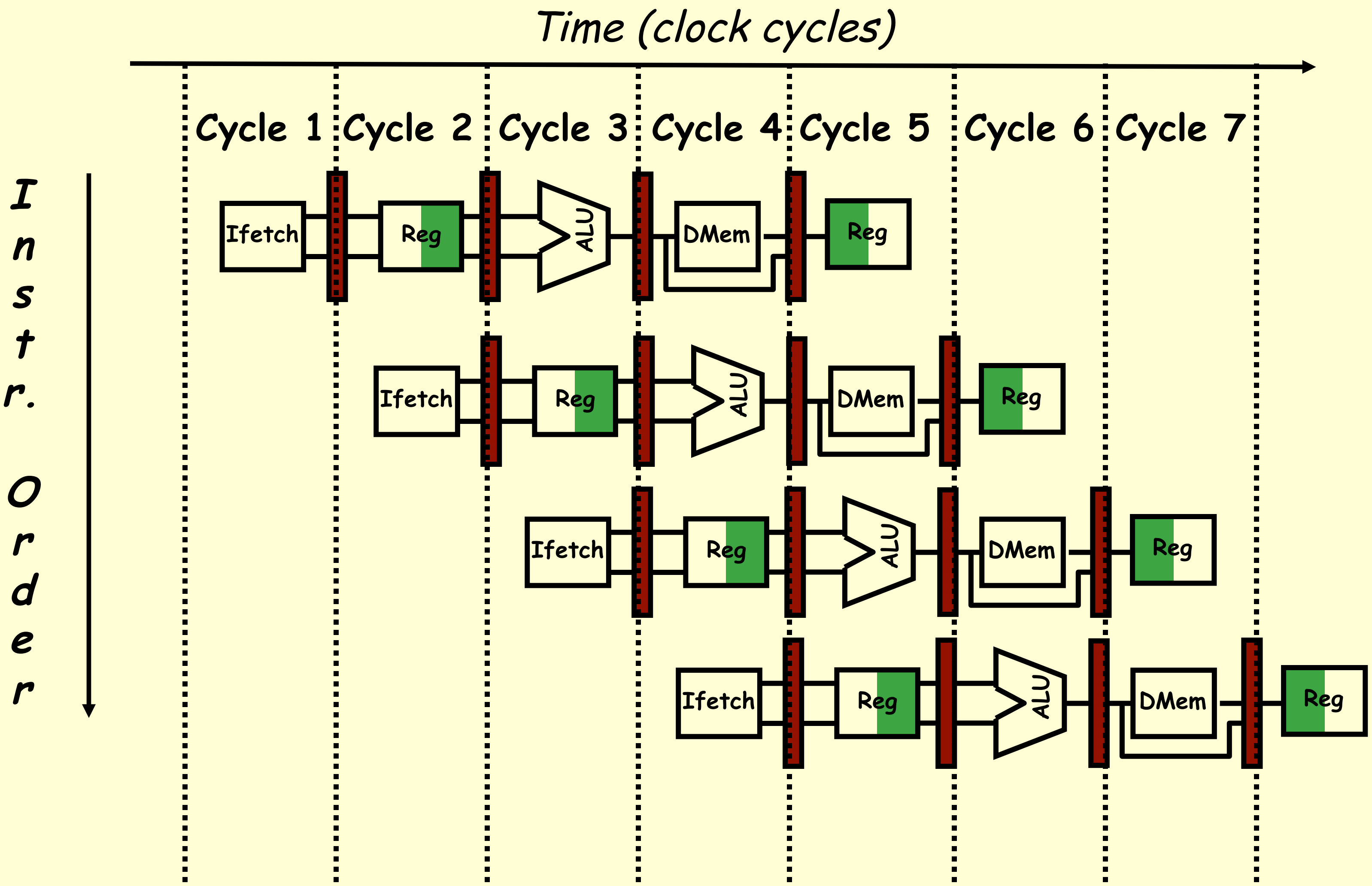
Pipeline Stage Interface

Stage	Any Instruction		
IF	IF/ID.IR ←MEM[PC] ; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond) {EX/MEM.ALUOutput } else { PC + 4 }) ;		
ID	ID/EX.A = Regs[IF/ID. IR $_{6..10}$]; ID/EX.B ←Regs[IF/ID. IR $_{11..15}$]; ID/EX.NPC ←IF/ID.NPC ; ID/EX.IR ←IF/ID.IR; 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; EX/MEM. ALUOutput ← ID/EX.A func ID/EX.B; Or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ←ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + ID/EX.Imm; EX/MEM.cond ← (ID/EX.A op 0);
MEM	MEM/WB.IR ←EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] ; Or Mem[EX/MEM.ALUOutput] ← EX/MEM.B ;	
WB	Regs[MEM/WB. IR $_{16..20}$] ← EM/WB.ALUOutput; Or Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.ALUOutput ;	For load only: Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.LMD;	

Pipeline Hazards

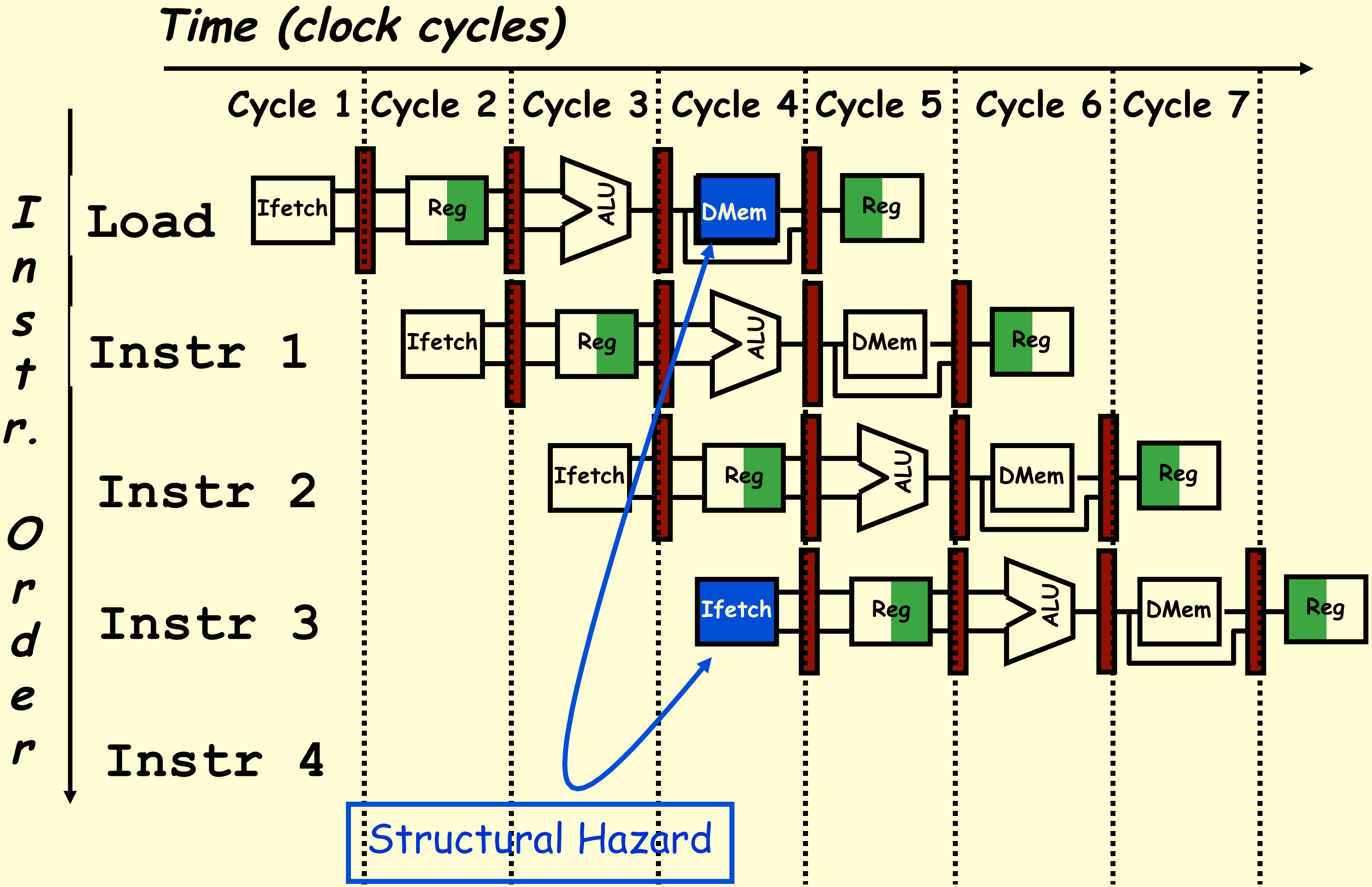
- Cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types
  - Structural hazard: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - Data hazard: attempt to use item before it is ready
    - Instruction 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

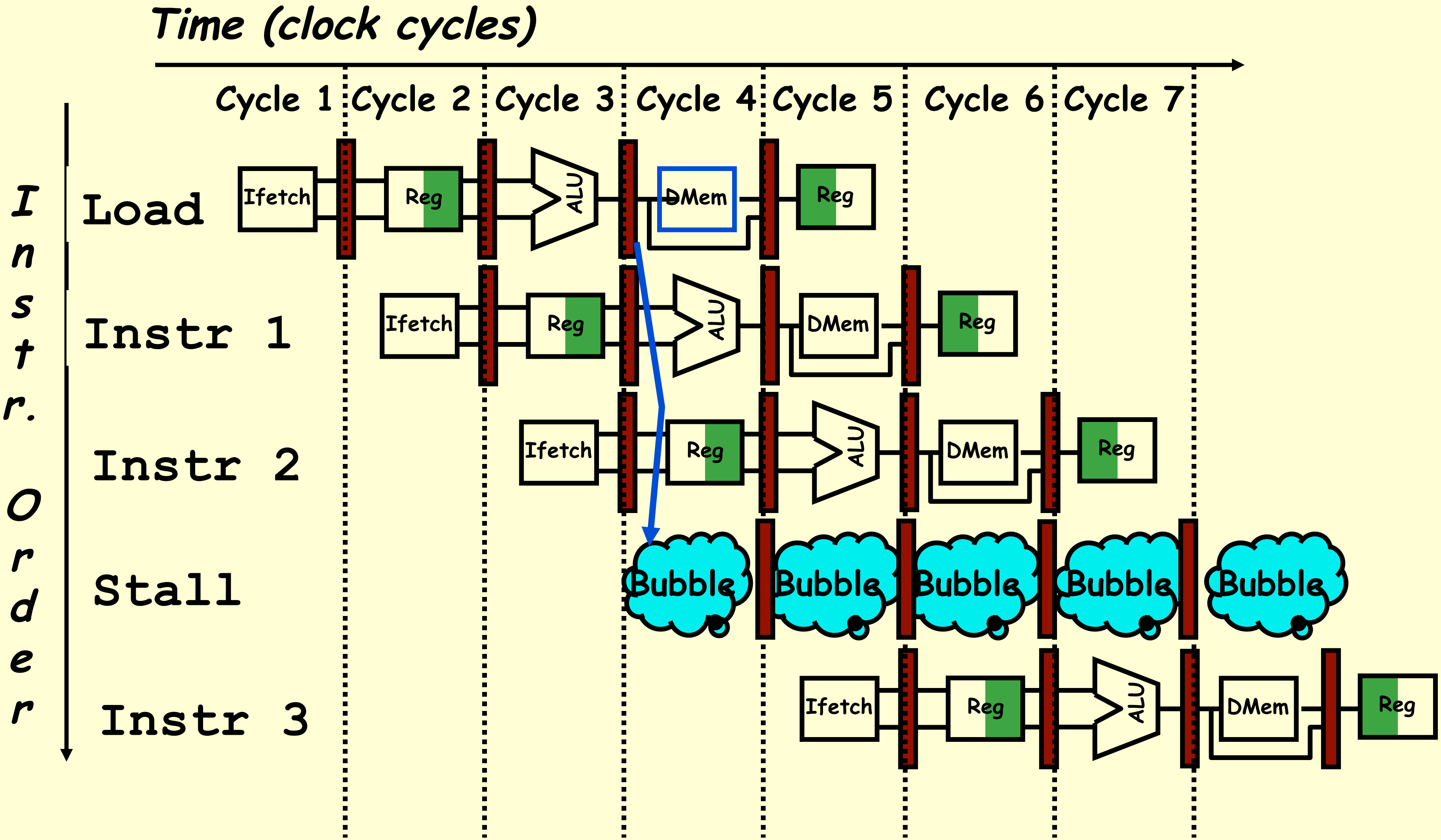


Slide: David Culler

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

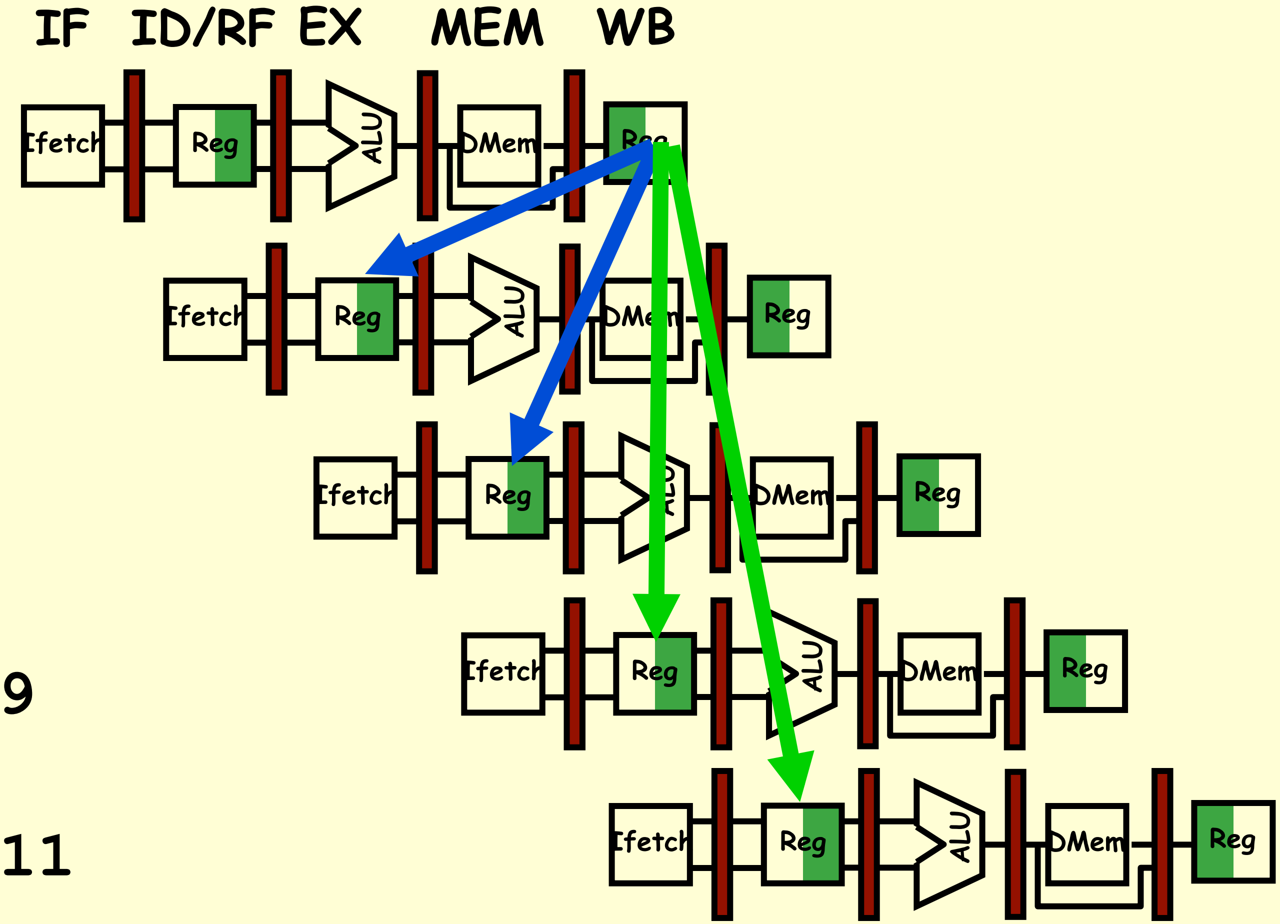
$$\boxed{\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}}$$

Data Hazards

Time (clock cycles)

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

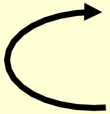
```
I: add r1,r2,r3
J: sub r4,r1,r3
```

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

Slide: David Culler

Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand before Instr$_I$ reads it



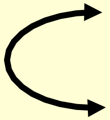
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- 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.

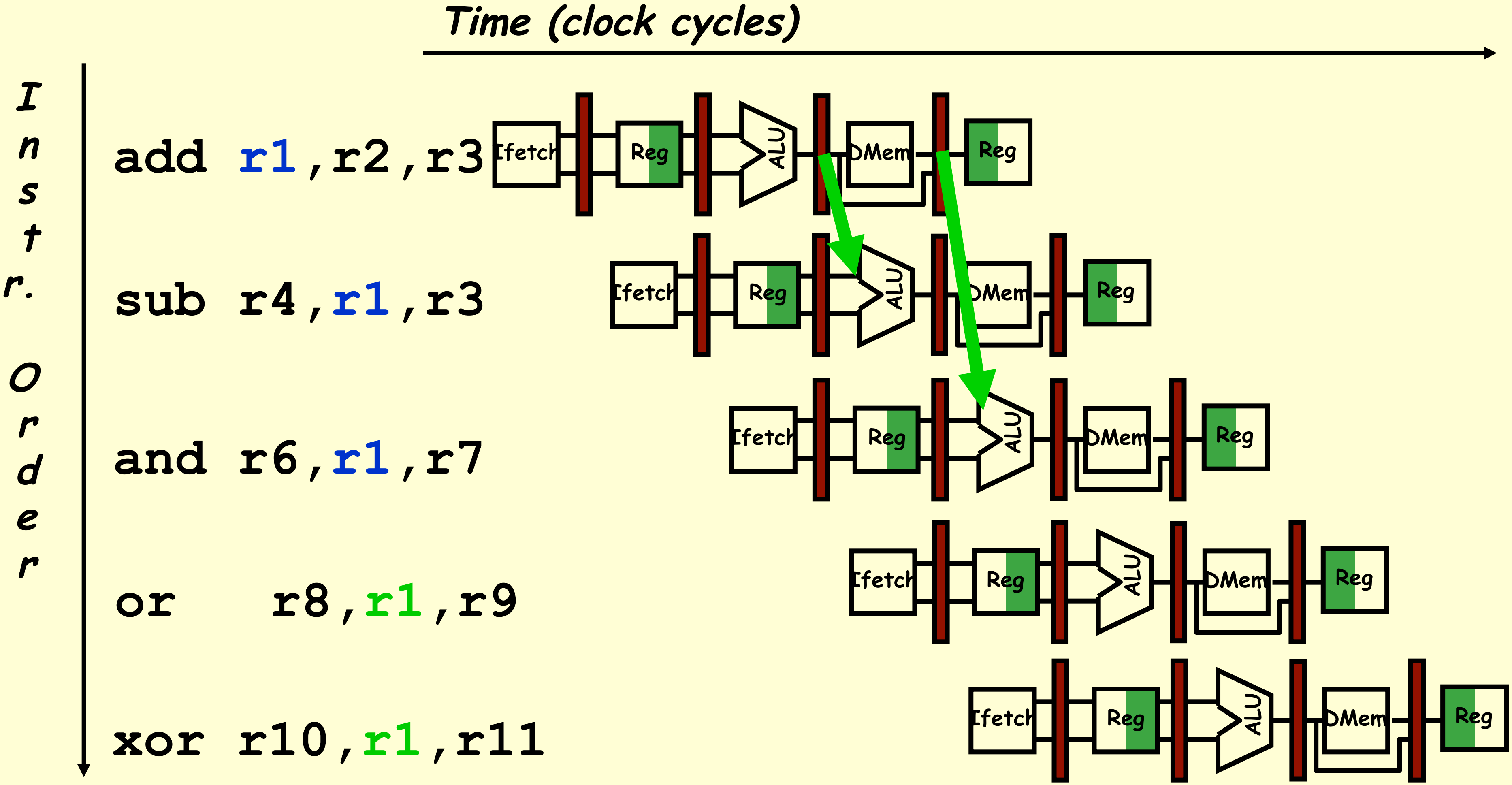


```
I: mul r1,r4,r3
J: add r1,r2,r3
K: sub r6,r1,r7
```

- 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



add r1,r2,r3

sub r4,r1,r3

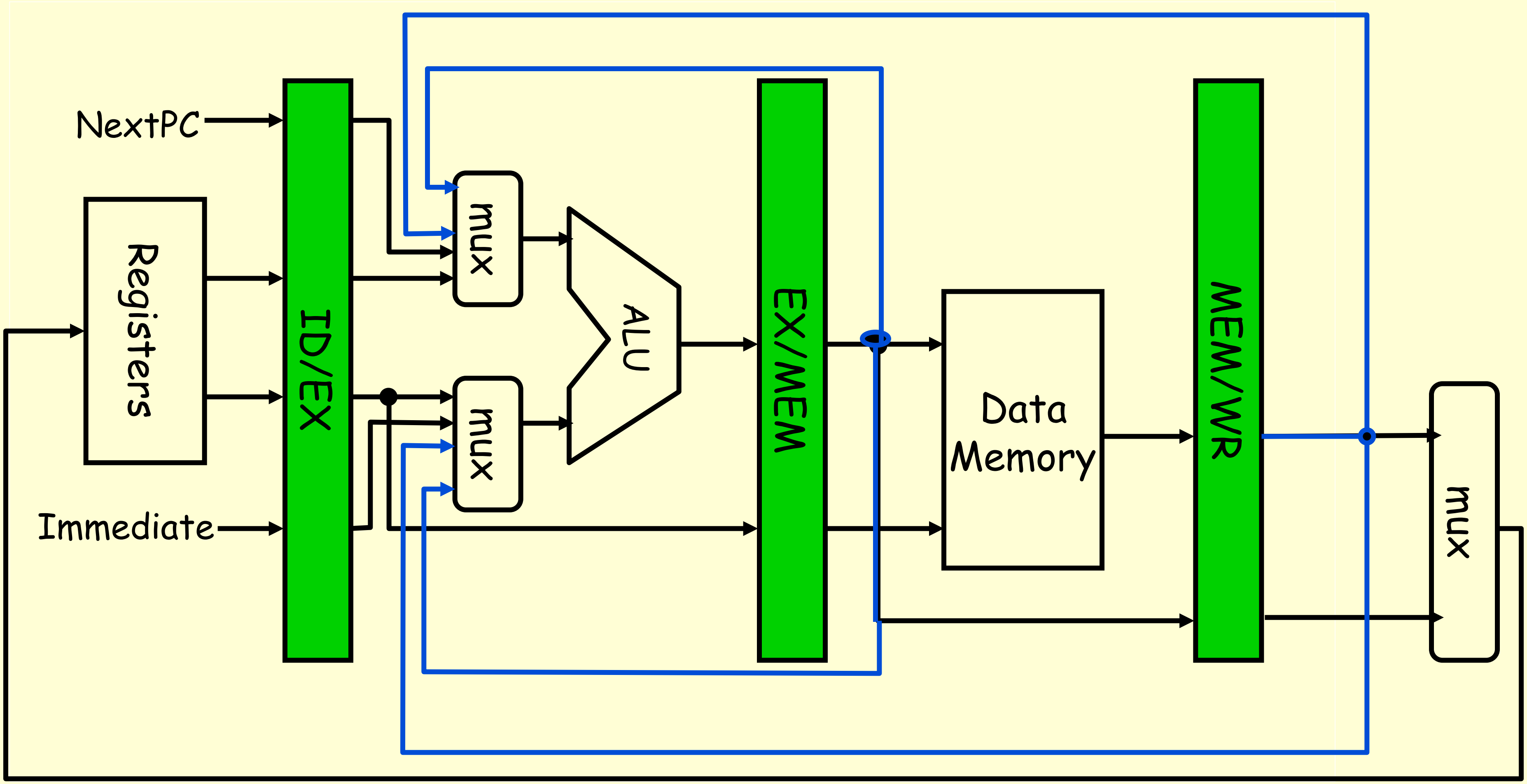
and r6,r1,r7

or   r8,r1,r9

xor r10,r1,r11

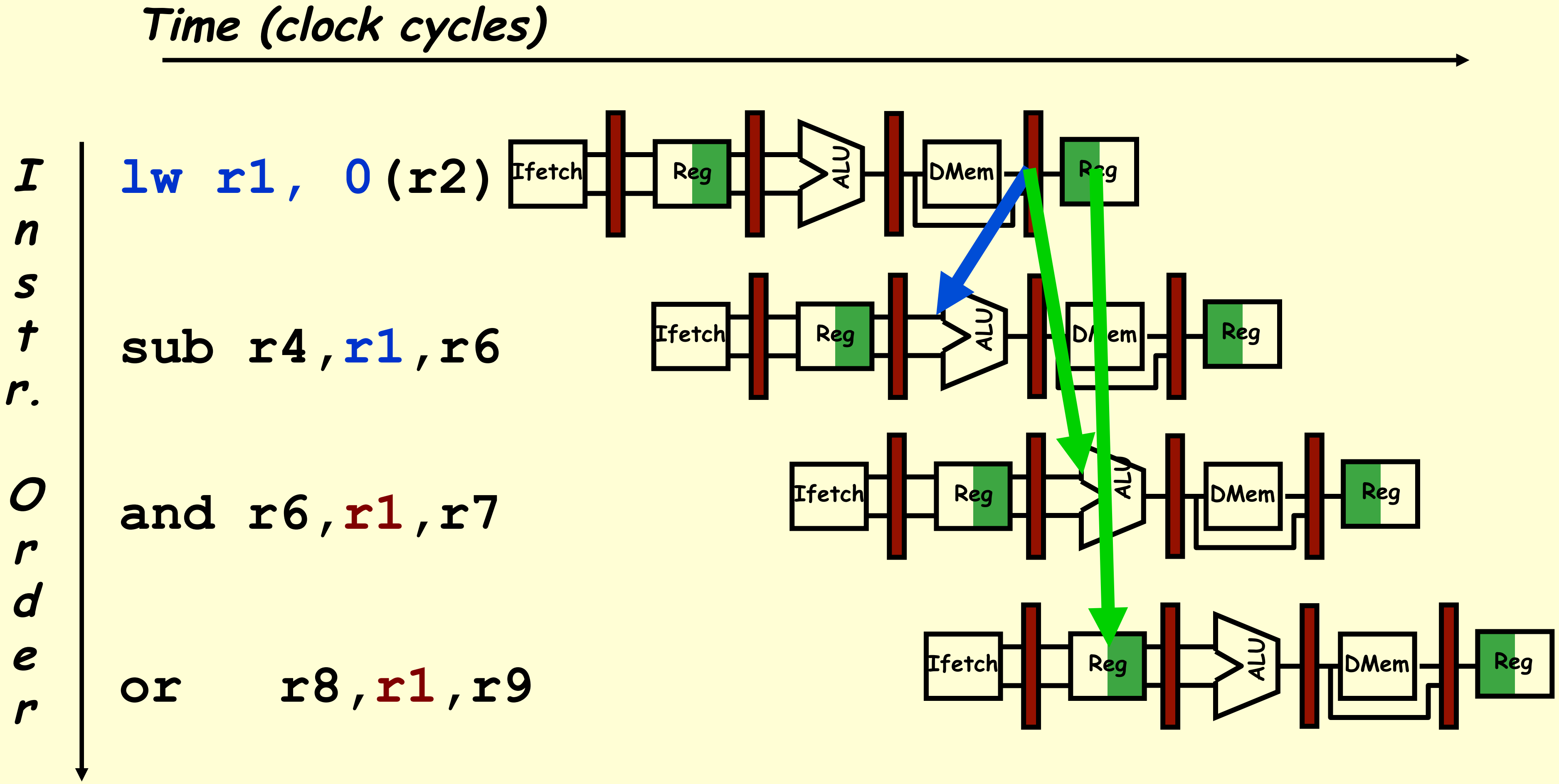
Slide: David Culler

HW Change for Forwarding



Slide: David Culler

Data Hazard Even with Forwarding



Time (clock cycles)

Instr. Order

`lw r1, 0(r2)`

`sub r4,r1,r6`

`and r6,r1,r7`

`or   r8,r1,r9`

Slide: David Culler

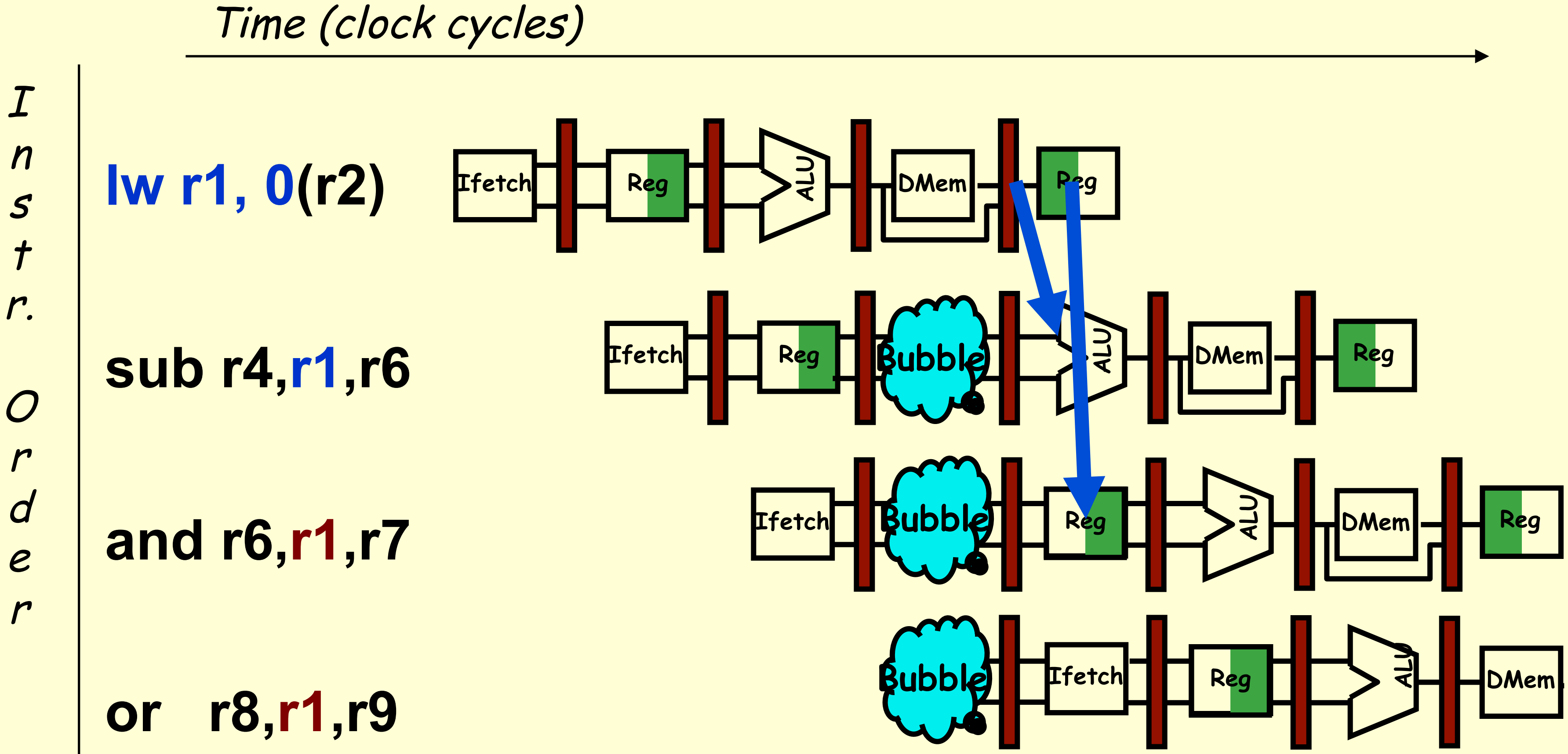
Resolving Load Hazards

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

- What is the cost of load delays?

Slide: David Culler

Resolving the Load Data Hazard



Time (clock cycles)

Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or r8,r1,r9

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

Slide: David Culler

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

Control Hazard on Branches
Three Stage Stall

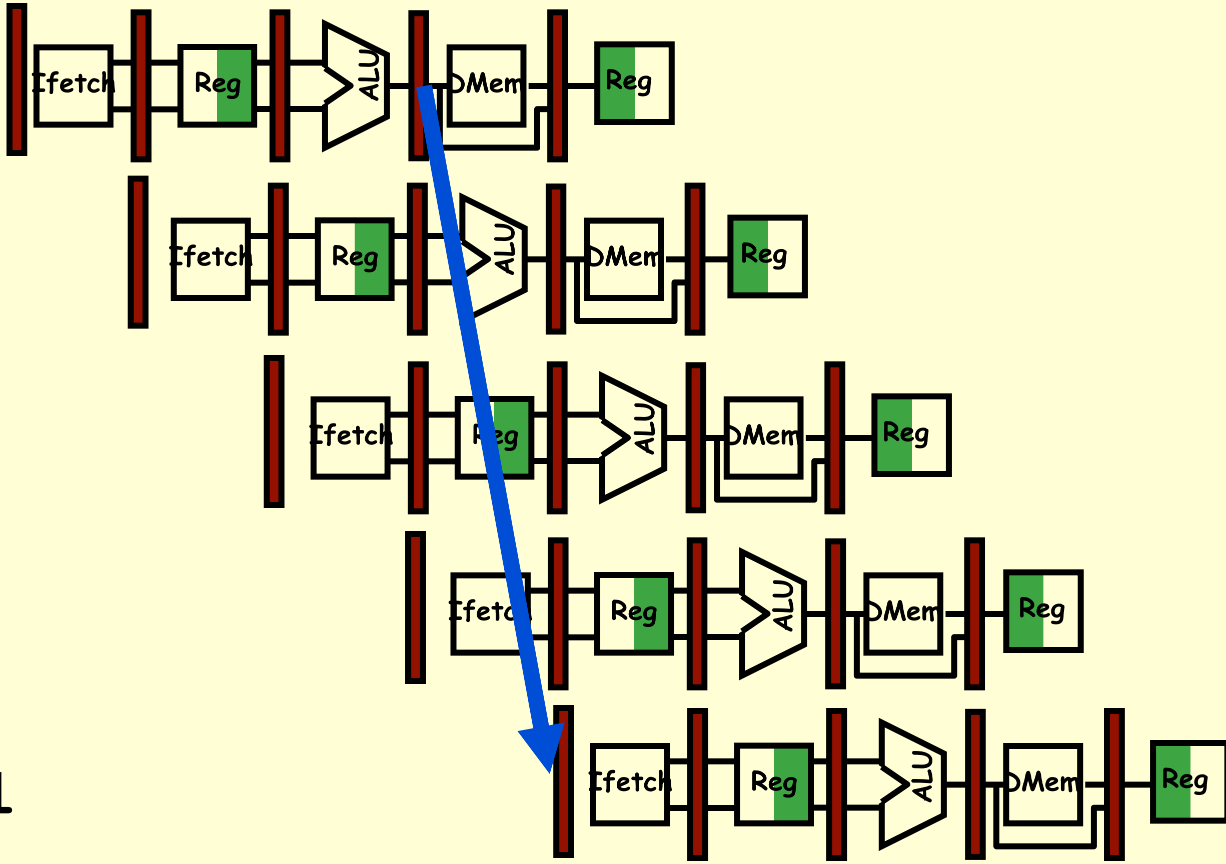
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



Slide: David Culler

Example: Branch Stall Impact

- If 30% branch, 3-cycle stall 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

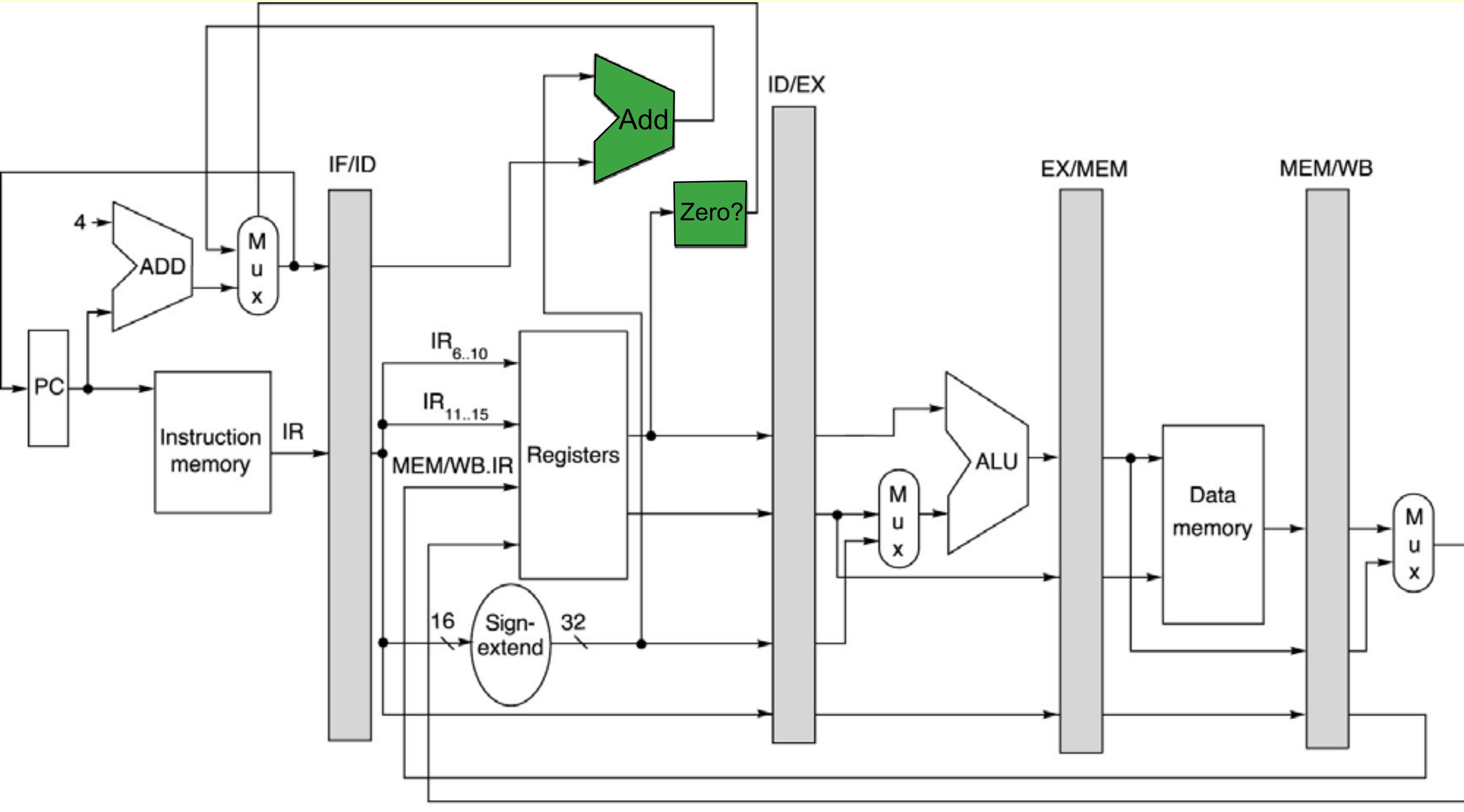


Figure: Dave Patterson

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

Slide: David Culler

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 target if taken

     Branch delay of length *n*

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

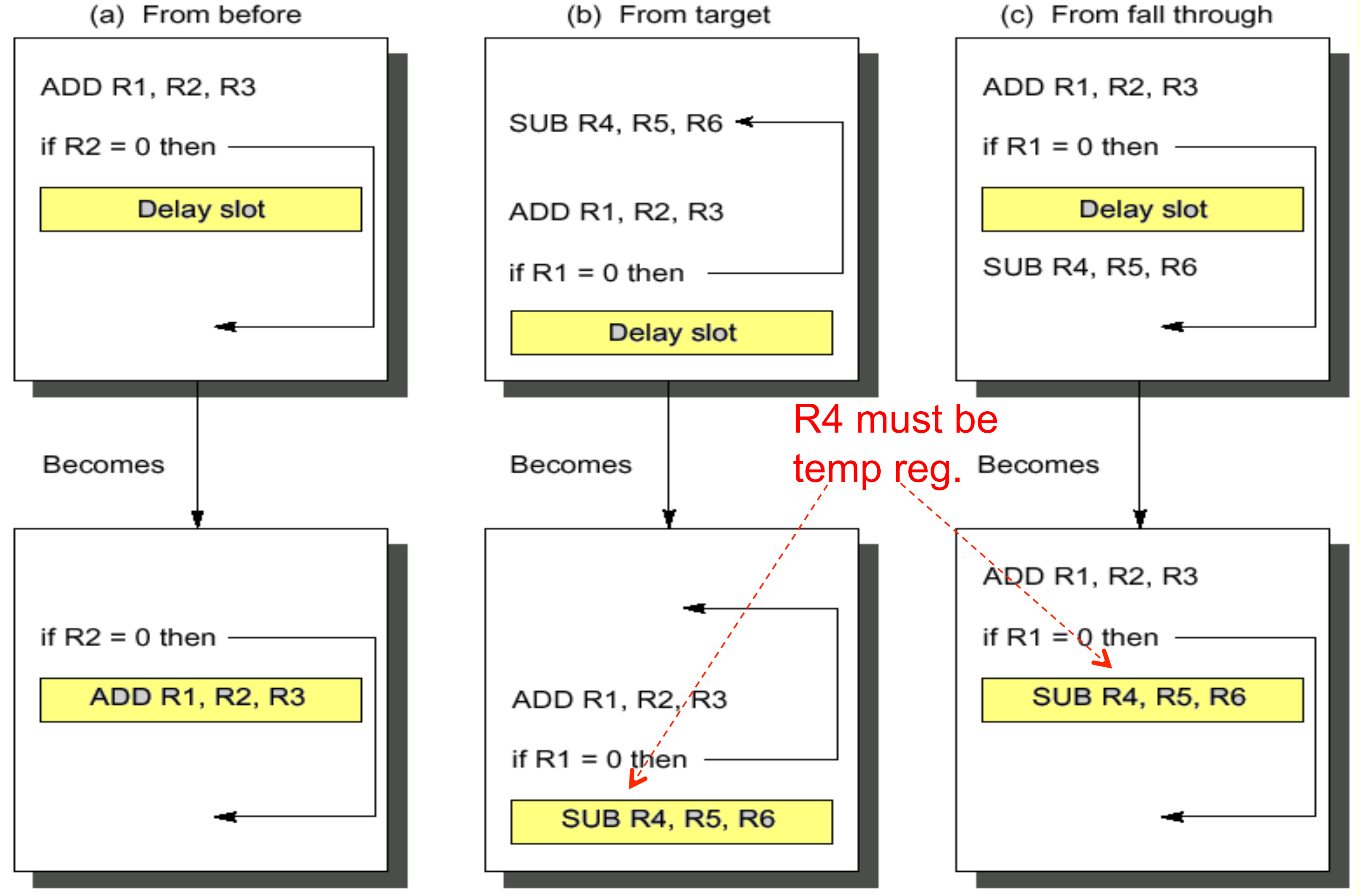
Slide: David Culler

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
  - 48% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Slide: David Culler

Scheduling Branch-Delay Slots



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)

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% Taken; 52% Delay slots not usefully filled

Scheduling Scheme	***Branch Penalty***	***CPI***	***Pipeline Speedup***	***Speedup vs stall***
Stall pipeline	3.00	1.42	3.52	1.00
Predict taken	1.00	1.14	4.39	1.25
Predict not taken	1.00	1.09	4.58	1.30
Delayed branch	0.52	1.07	4.66	1.32

Slide: David Culler

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