CMSC 611: Advanced Computer Architecture

Instruction Set Architecture & Pipelining

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

Instruction Representation

- All data in computer systems is represented in binary
- Instructions are no exception
- The program that translates the human-readable code to numeric form is called an *Assembler*
- Hence *machine-language* or *assembly-language*

Example:

Assembly: `ADD $t0, $s1, $s2`

Note: by default MIPS $t0..$t7 map to reg. 8..15, $s0..$s7 map to reg. 16-23

M/C language (hex by field): `0x0 0x11 0x12 0x8 0x020`

M/C language (binary):

0000	0010	0011	0010	0100	0000	0010	0000

M/C language (hex): `0x02324020`

Encoding an Instruction Set

- Affects the size of the compiled program
- Also complexity of the CPU implementation
- Operation in one field called opcode
- Addressing mode in opcode or separate field
- Must balance:
  - Desire to support as many registers and addressing modes as possible
  - Effect of operand specification on the size of the instruction (and program)
  - Desire to simplify instruction fetching and decoding during execution
- Fixed size instruction encoding simplifies CPU design but limits addressing choices

Encoding Examples

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier	Address field

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

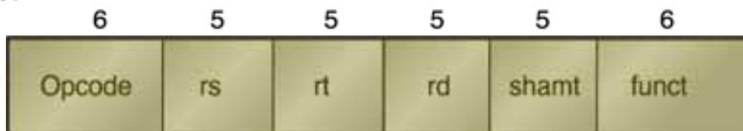
MIPS Instruction Formats

I-type instruction

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

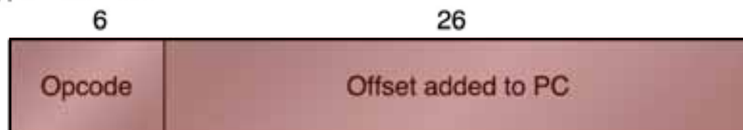
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction



Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction



Jump and jump and link
Trap and return from exception

opcodes

	000	001	010	011	100	101	110	111
000	R-type		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	
010								
011	llo	lhi	trap					
100	lb	lh		lw	lbu	lhu		
101	sb	sh		sw				
110								
111								

funct codes

	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

GPU Shading ISA

- Data
  - IEEE-like floating point
  - 4-element vectors
    - Most instructions perform operation on all four
- Addressing
  - No addresses
  - ATTRIB, PARAM, TEMP, OUTPUT
  - Limited arrays
  - Element selection (read & write)
    - C.xyw, C.rgba

GPU Shading ISA

- Instructions:

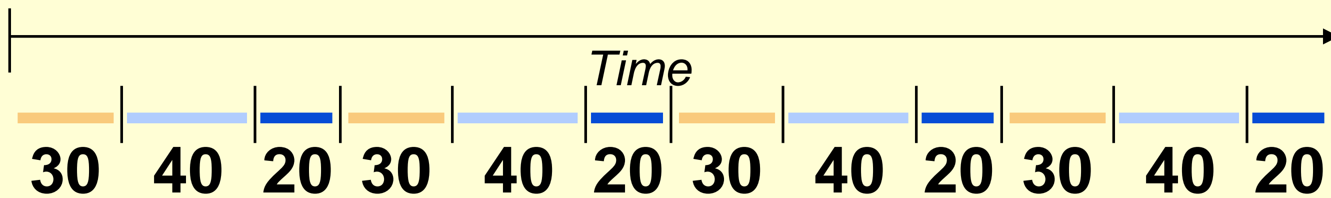
Instruction	Operation	Instruction	Operation
ABS r,s	r = abs(s)	MIN r,s1,s2	r = min(s1,s2)
ADD r,s1,s2	r = s1+s2	MOV r,s1	r = s1
CMP r,c,s1,s2	r = c<0 ? s1 : s2	MUL r,s1,s2	r = s1*s2
COS r,s	r = cos(s)	POW r,s1,s2	r ≈ $s1^{s2}$
DP3 r,s1,s2	r = s1.xyz • s2.xyz	RCP r,s1	r = 1/s1
DP4 r,s1,s2	r = s1 • s2	RSQ r,s1	r = 1/sqrt(s1)
DPH r,s1,s2	r = s1.xyz1 • s2	SCS r,s1	r = (cos(s),sin(s),?,?)
DST r,s1,s2	r = (1,s1.y*s2.y,s1.z,s2.w)	SGE r,s1,s2	r = s1≥s2 ? 1 : 0
EX2 r,s	r ≈ 2^s	SIN r,s	r = sin(s)
FLR r,s	r = floor(s)	SLT r,s1,s2	r = s1<s2 ? 1 : 0
FRC r,s	r = s - floor(s)	SUB r,s1,s2	r = s1-s2
KIL s	if (s<0) discard	SWZ r,s,cx,cy,cz,cw	r = swizzle(s)
LG2 r,s	r ≈ $\log_2(s)$	TEX r,s,name,nD	r = texture(s)
LIT r,s	r = lighting computation	TXB r,s,name,nD	r = textureLOD(s)
LRP r,t,s1,s2	r = t*s1 + (1-t)*s2	TXP r,s,name,nD	r = texture(s/s.w)
MAD r,s1,s2,s3	r = s1*s2 + s3	XPD r,s1,s2	r = s1×s2
MAX r,s1,s2	r = max(s1,s2)		

GPU Shading ISA

- Notable:
  - Many special-purpose instructions
  - No binary encoding, interface is text form
    - No ISA limits on future expansion
    - No ISA limits on registers
    - No ISA limits on immediate values
  - Originally no branching! (exists now)

Sequential Laundry

6 PM 7 8 9 10 11 Midnight



Task Order

A

B

C

D

- 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

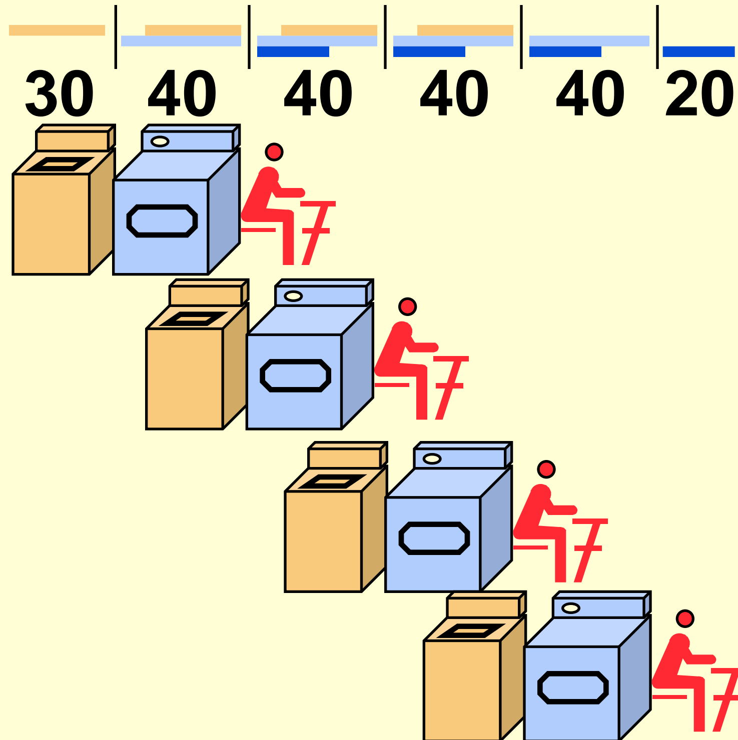
Task Order

A

B

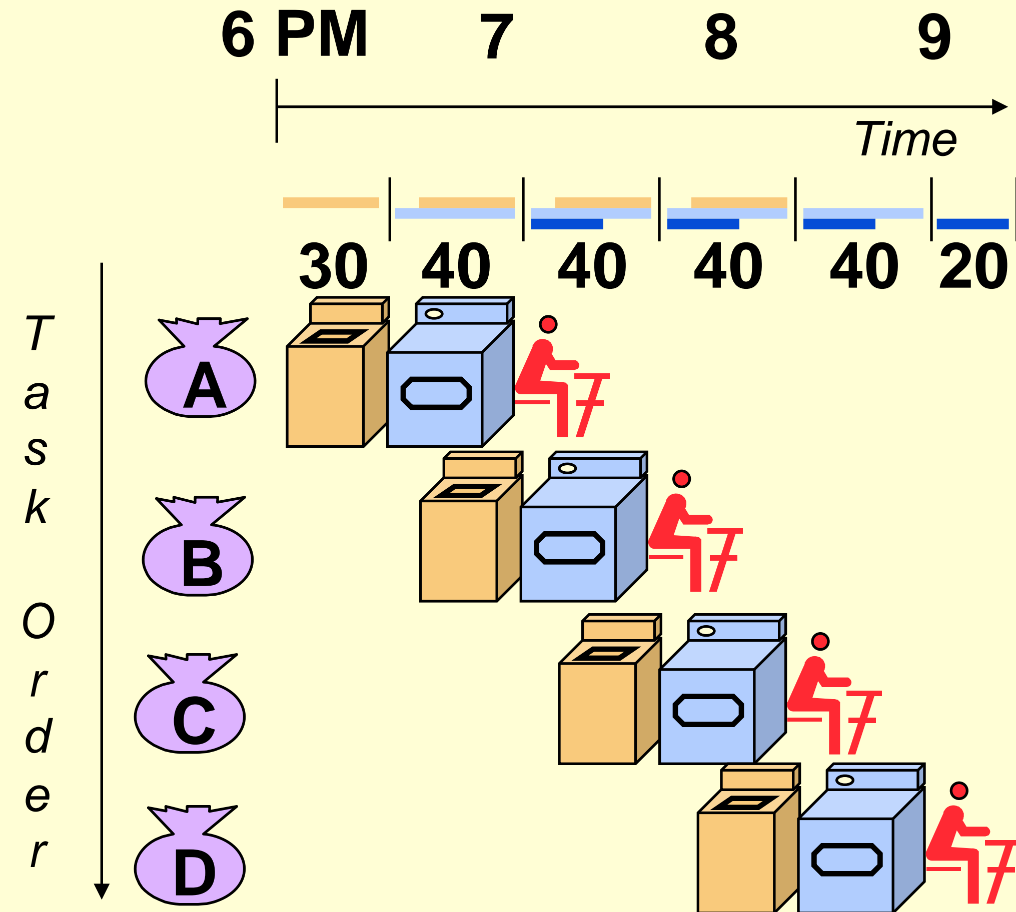
C

D



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

Pipelining Lessons

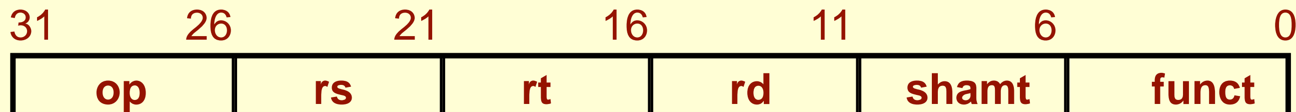


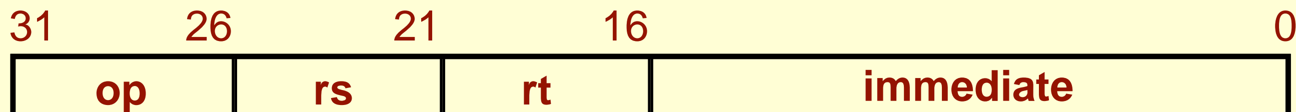
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

Slide: Dave Patterson

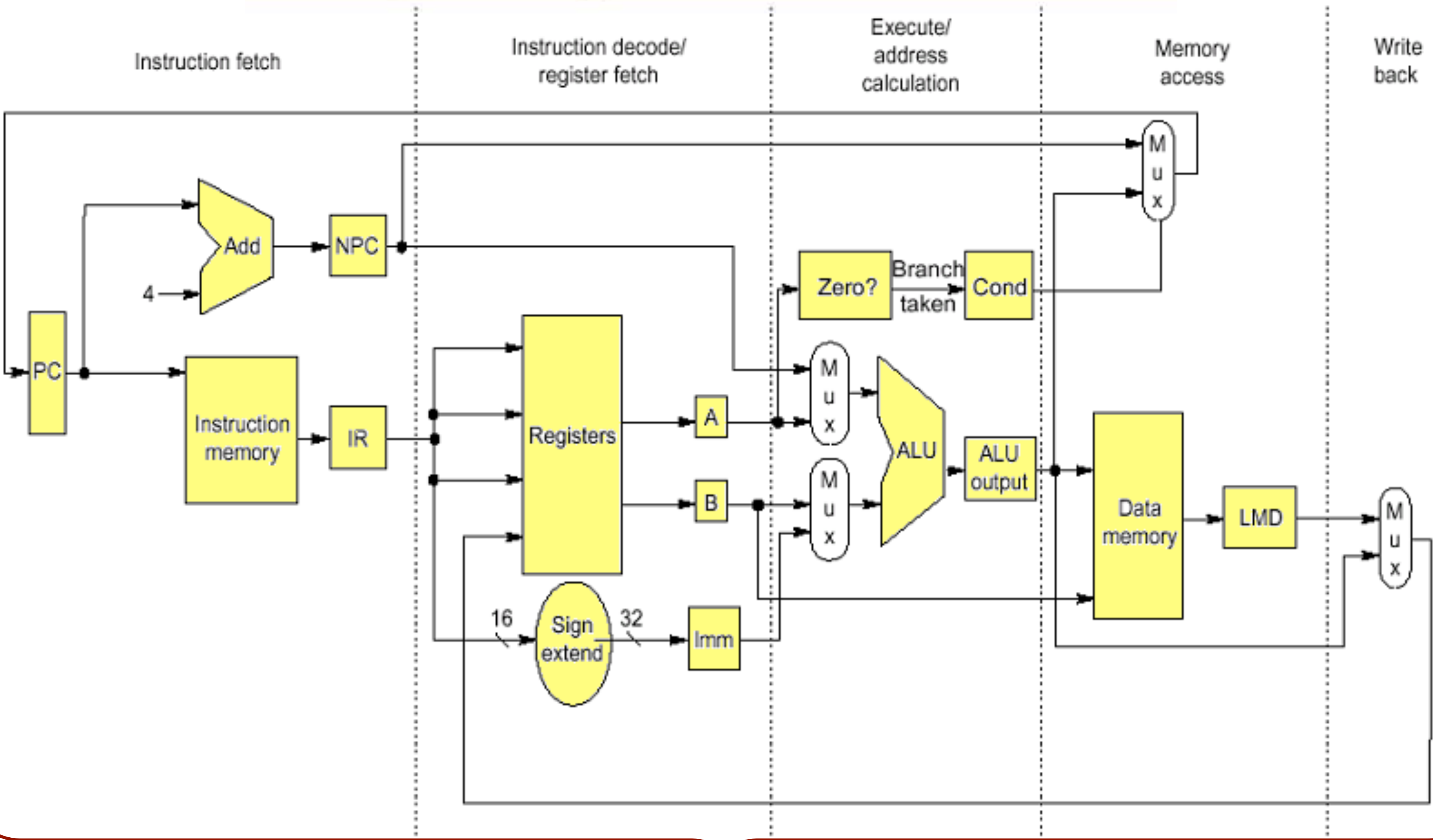
MIPS Instruction Set

- 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





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

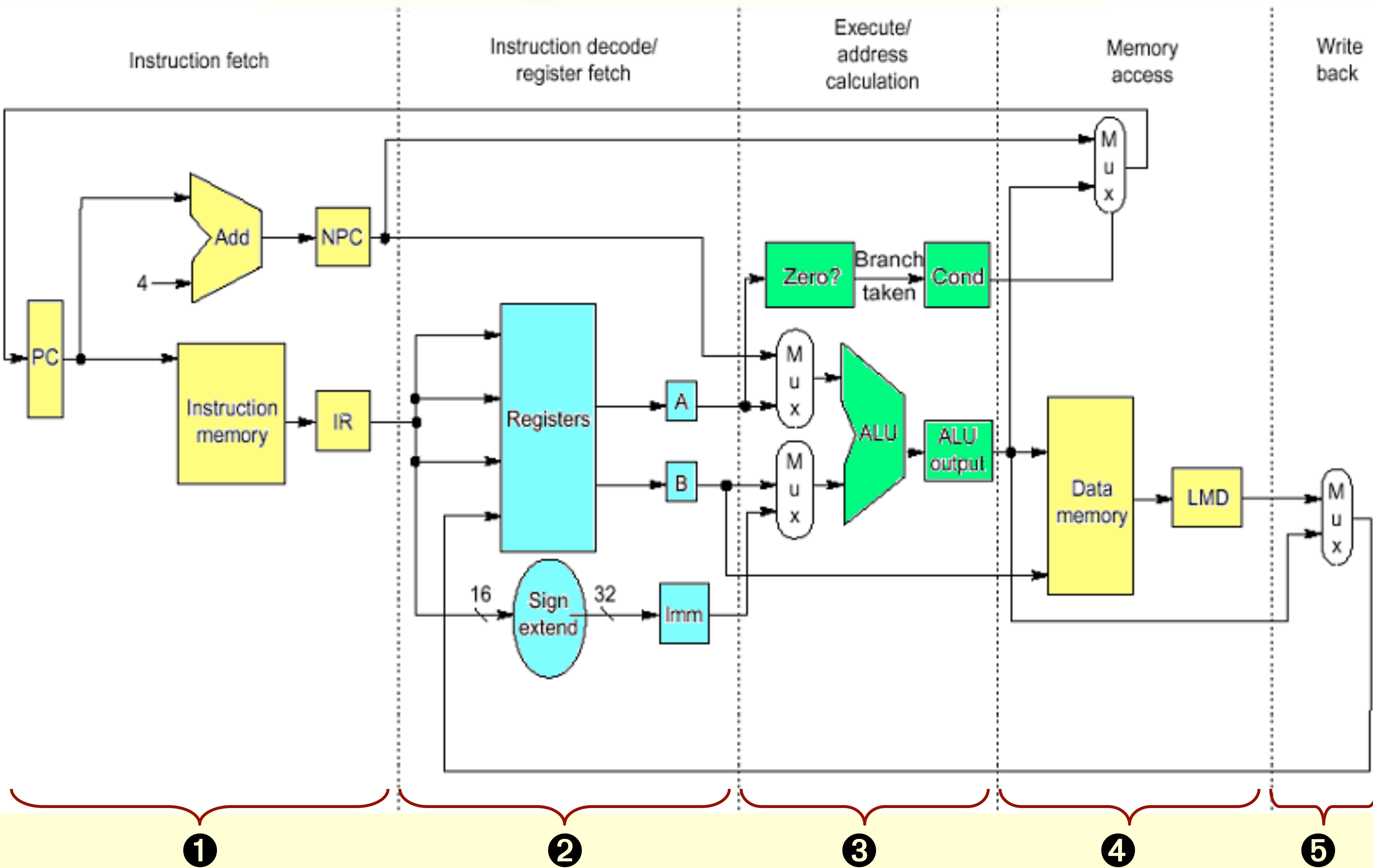
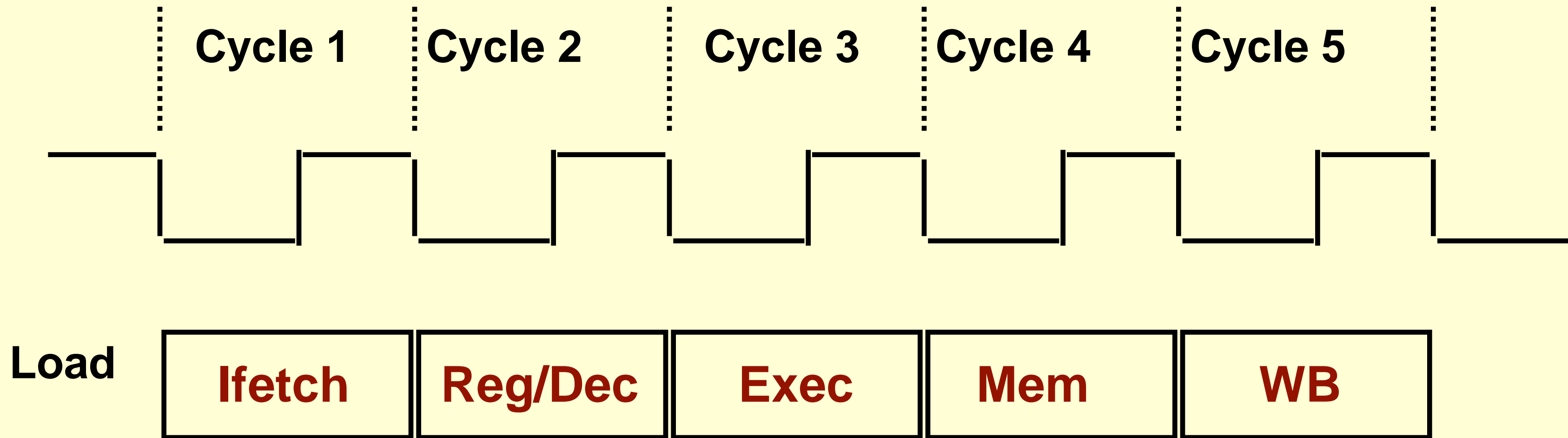


Figure: Dave Patterson

Stages of Instruction Execution

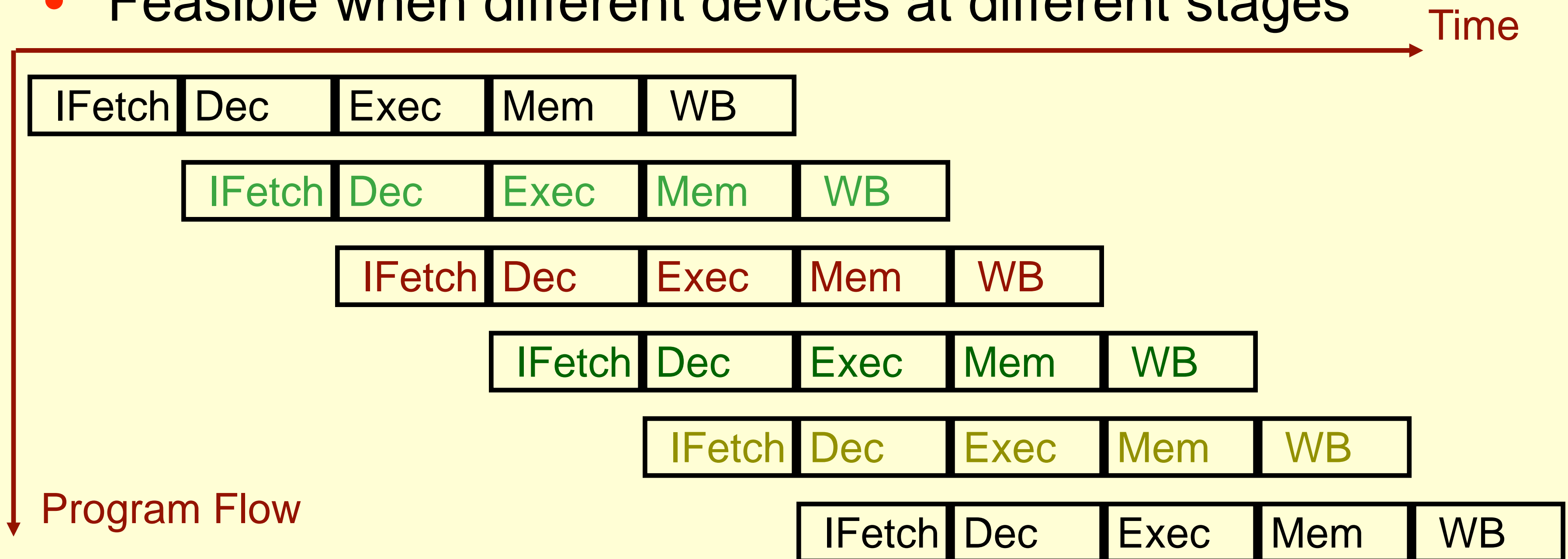


- The load instruction is the longest
- All instructions follows at most the following five steps:
  - Ifetch: Instruction Fetch
    - Fetch the instruction from the Instruction Memory and update PC
  - Reg/Dec: Registers Fetch and Instruction Decode
  - Exec: Calculate the memory address
  - Mem: Read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

Slide: Dave Patterson

Instruction Pipelining

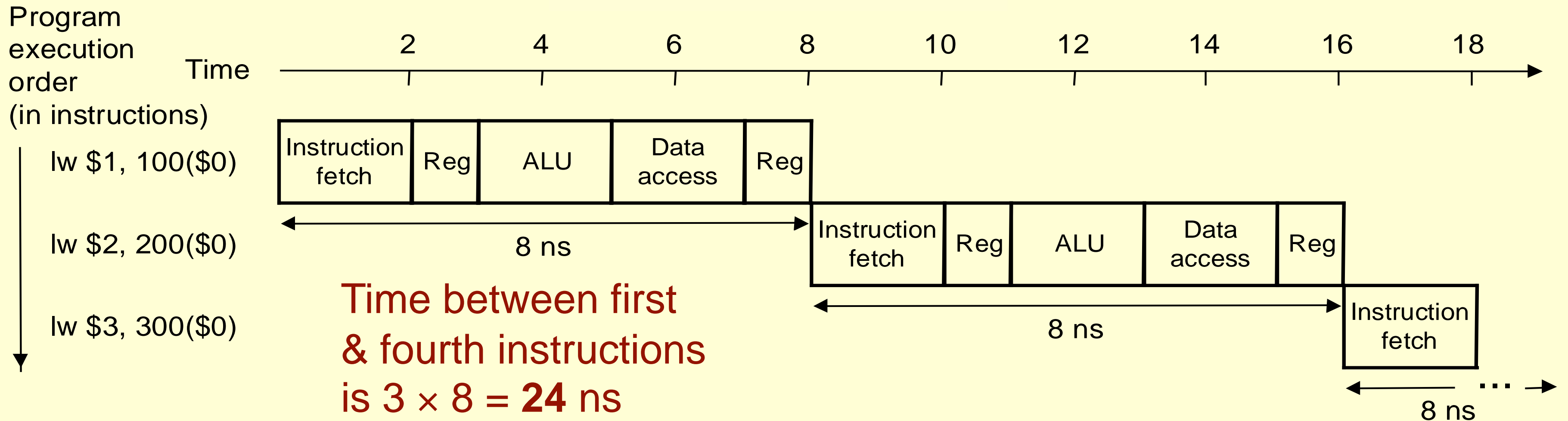
- Start handling next instruction while the current instruction is in progress
- Feasible when different devices at different stages

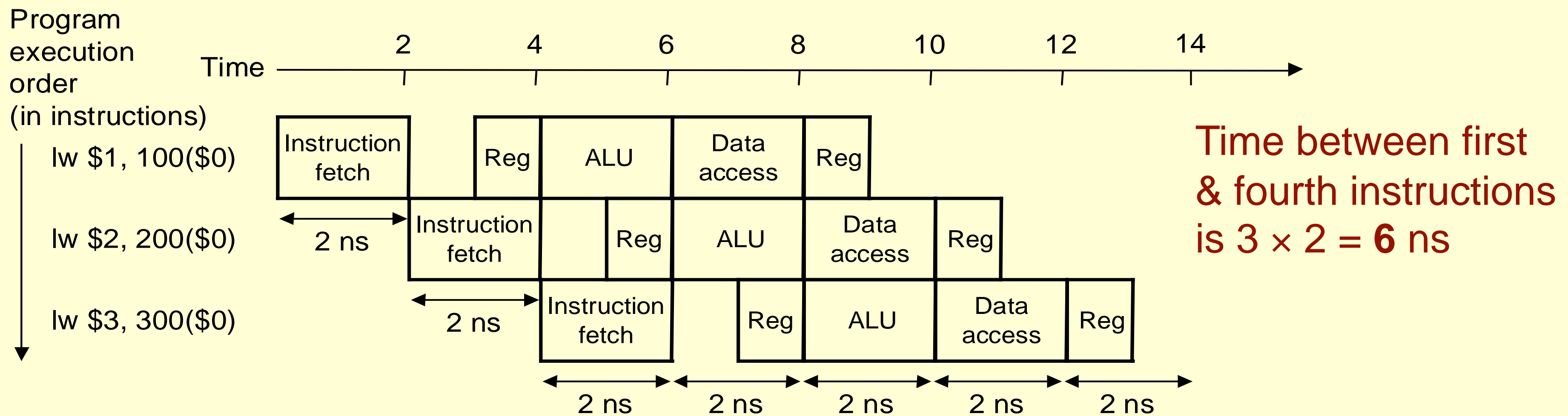


$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput

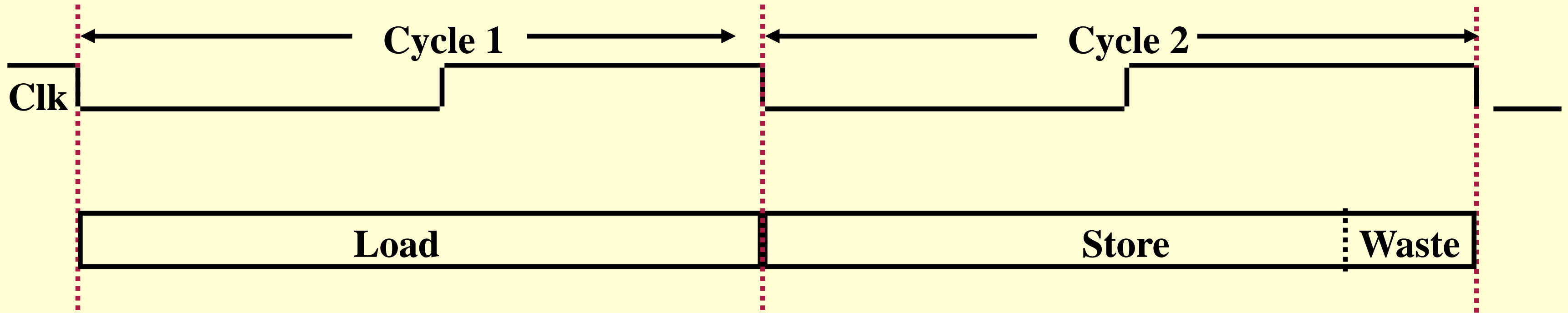
Example of Instruction Pipelining





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

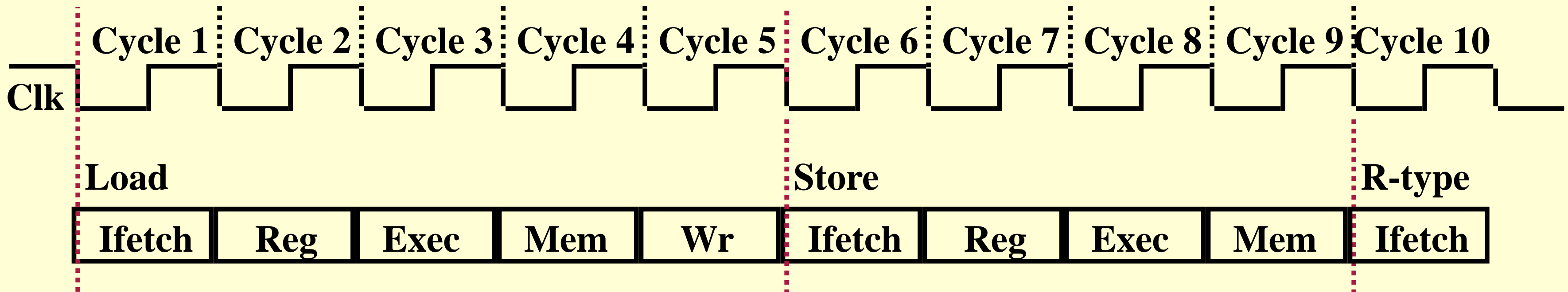
Single Cycle



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

Figure: Dave Patterson

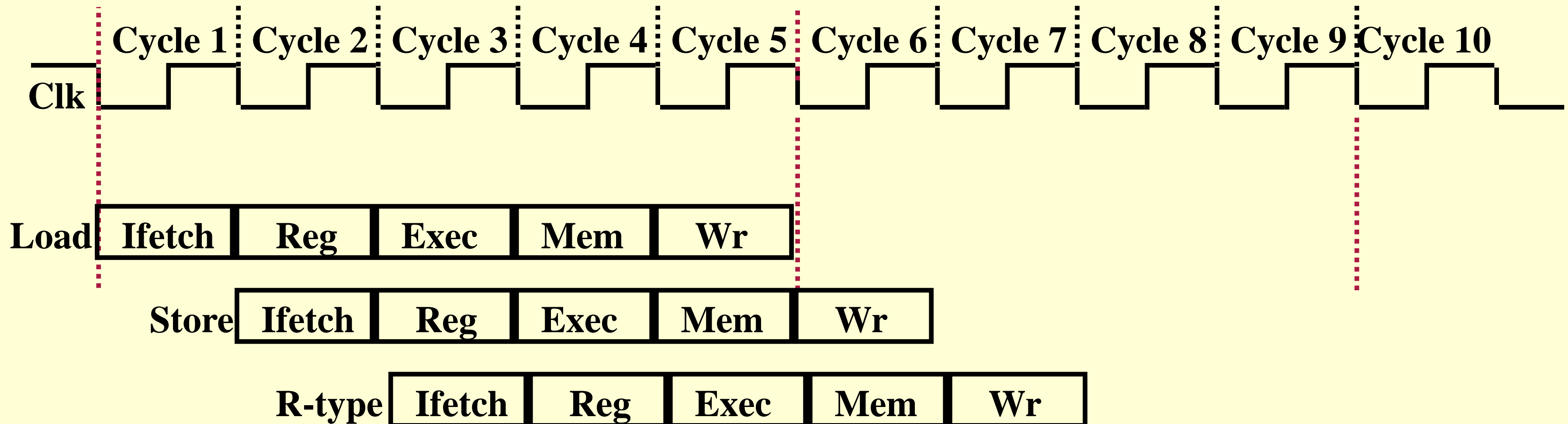
Multiple Cycle



- 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

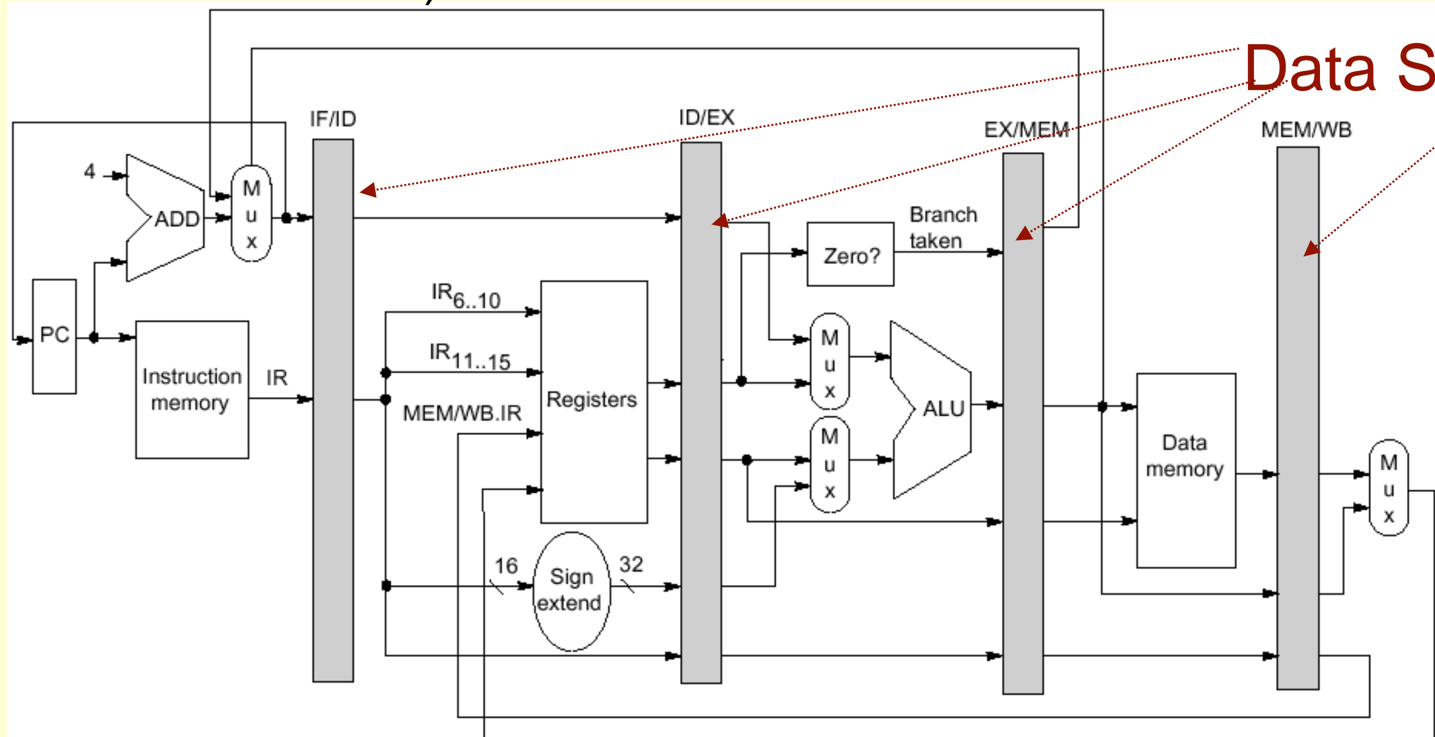
Figure: Dave Patterson

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)



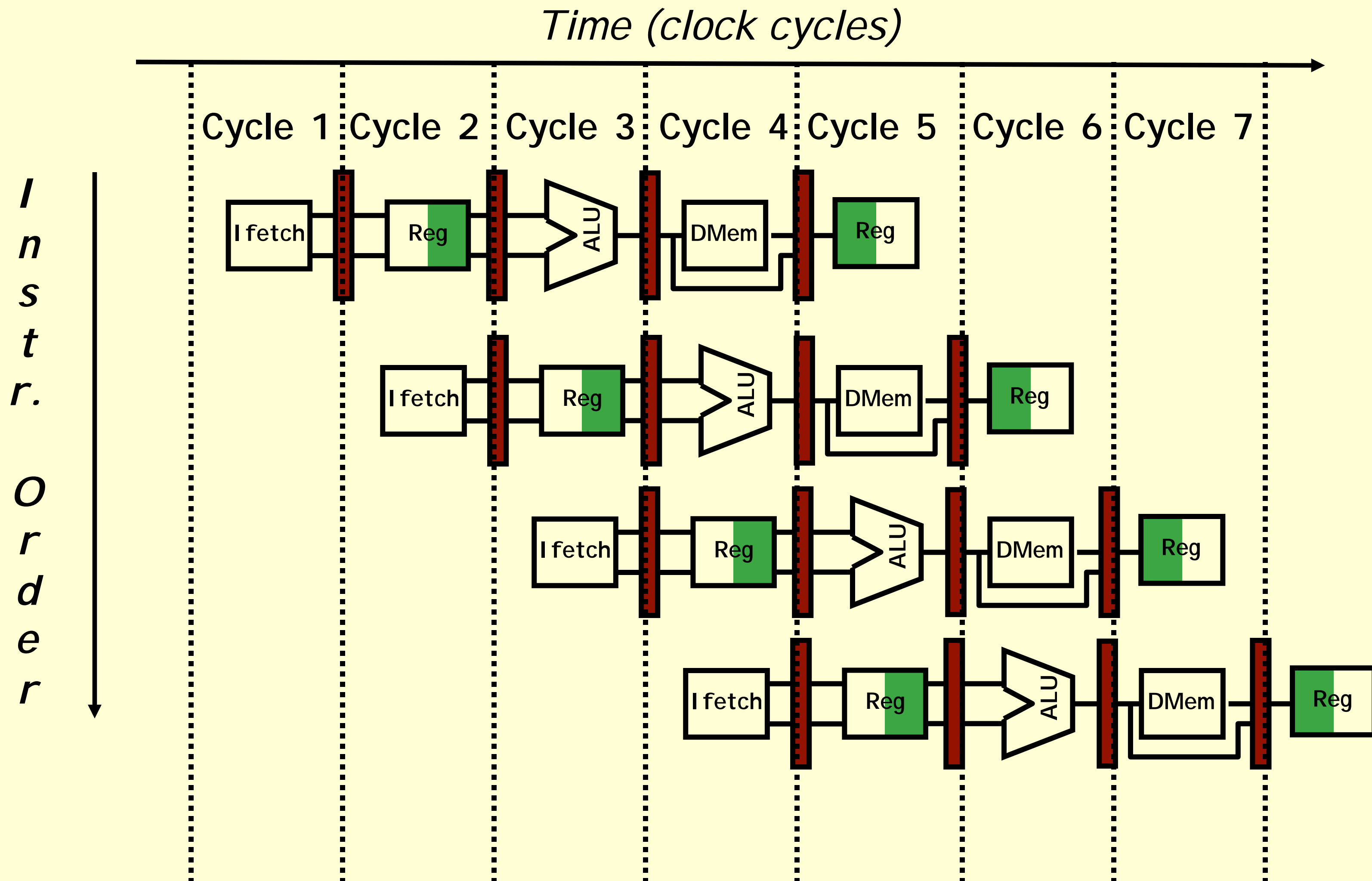
Pipeline Stage Interface

Stage	Any Instruction		
IF	IF/ID.IR ←MEM[PC] ; 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

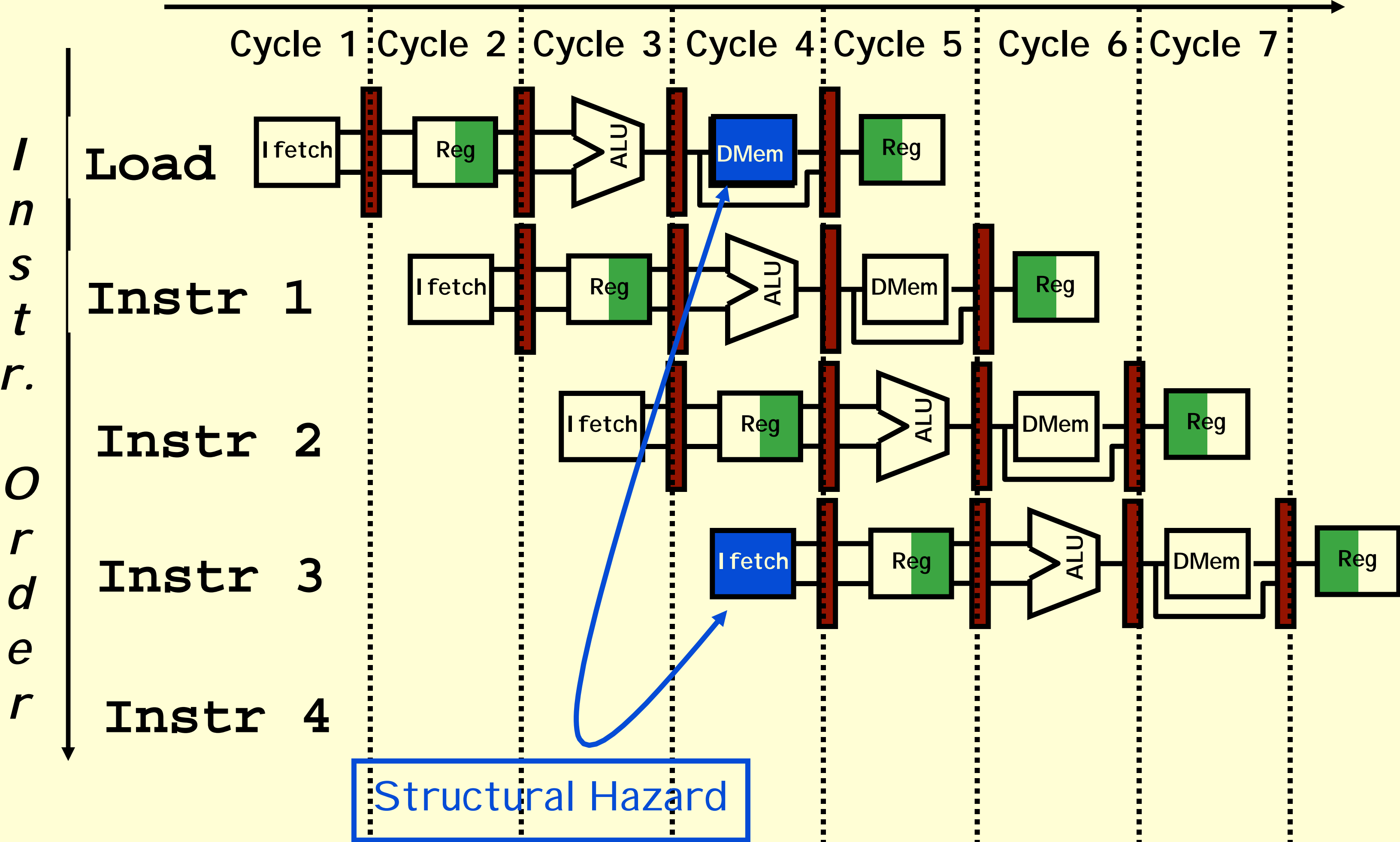
- Cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types
  - Structural hazard: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - Data hazard: attempt to use item before it is ready
    - Instruction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

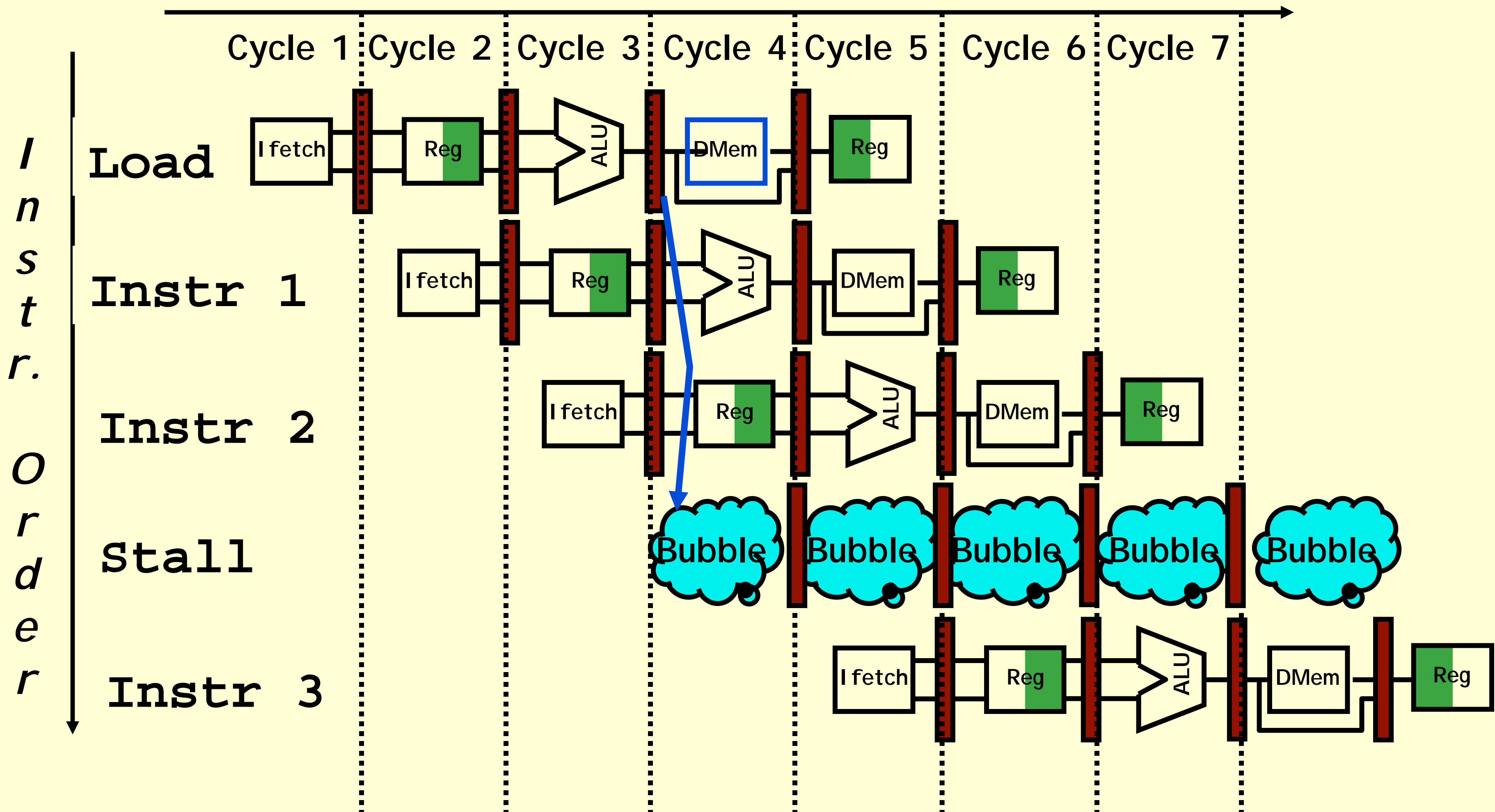


Slide: David Culler

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

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