

# **CS152**

## **Computer Architecture and Engineering**

### **Lecture 1**

**January 18, 1995**

**Dave Patterson & Shing Kong**

cs 152 Intro.1

©DAP & SIK 1995

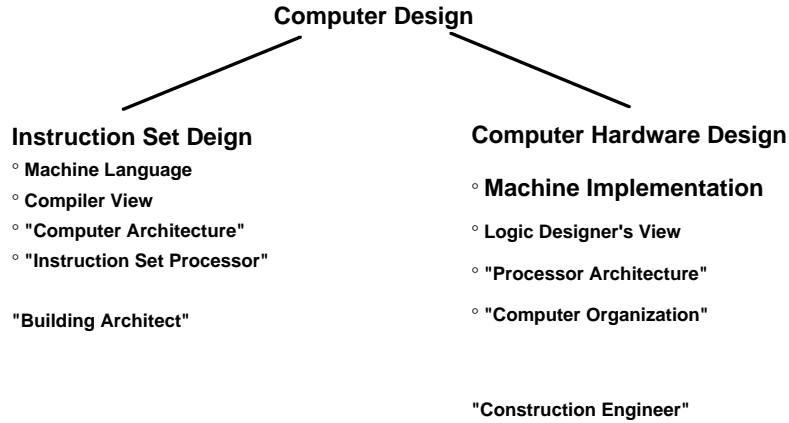
## **Overview of Today's Lecture**

- **Course Overview (20 minutes: Dave Patterson)**
- **Administrative Matters (3 minutes: DP)**
- **Course Philosophy and Structure (10 min: DP)**
- **Level of Representation (15 min: DP)**
- **Break (5 min)**
- **Levels of Organization (25 min: Kong)**

cs 152 Intro.2

©DAP & SIK 1995

## CS152: Course Overview



cs 152 Intro.3

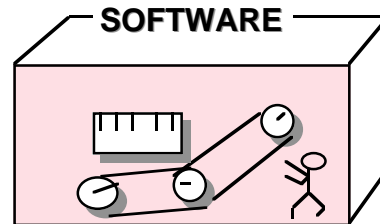
©DAP & SIK 1995

## Instruction Set Architecture

... the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

Amdahl, Blaaw, and Brooks, 1964

- Organization of Programmable Storage
- Data Types & Data Structures: Encodings & Representations
- Instruction Formats
- Instruction (or Operation Code) Set
- Modes of Addressing and Accessing Data Items and Instructions
- Exceptional Conditions



cs 152 Intro.4

©DAP & SIK 1995

## Organization

ISA Level

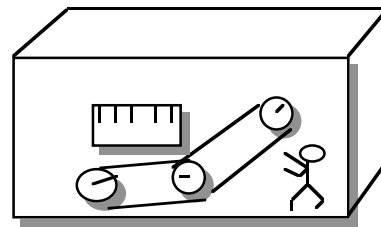
FUs & Interconnect

### *Logic Designer's View*

- Capabilities & Performance Characteristics of Principal Functional Units  
(e.g., Registers, ALU, Shifters, Logic Units, etc.
- Ways in which these components are interconnected
- nature of information flows between components
- logic and means by which such information flow is controlled.

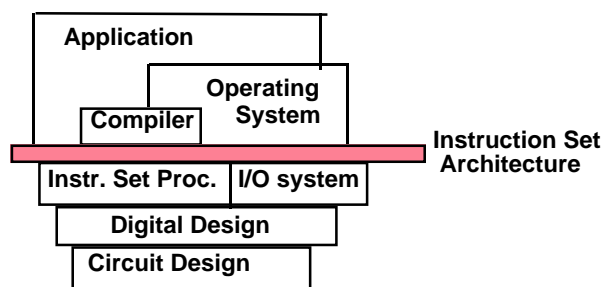
Choreography of FUs to realize the ISA

Register Transfer Level Description



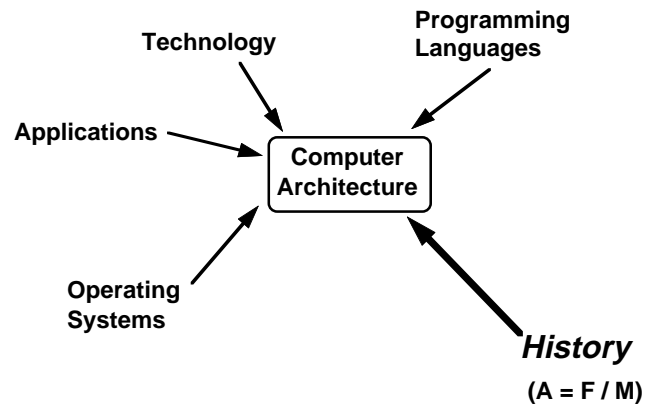
## What is "Computer Architecture"

- Co-ordination of *levels of abstraction*



- Under a set of rapidly changing *Forces*

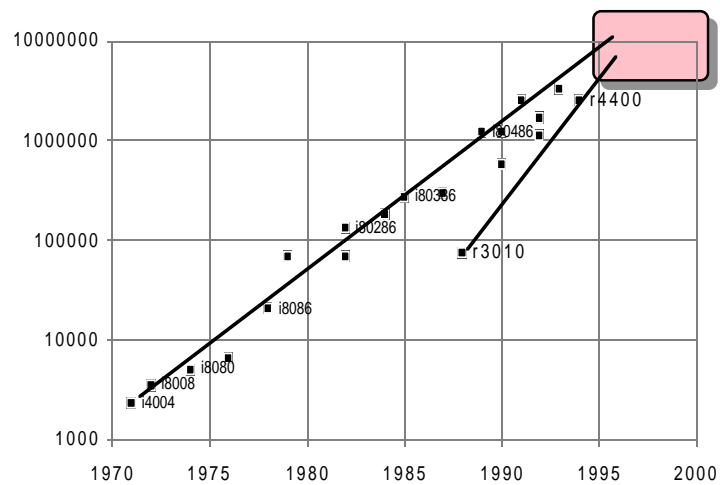
## Forces on Computer Architecture



cs 152 Intro.7

©DAP & SIK 1995

## Technology: Microprocessor Logic Density

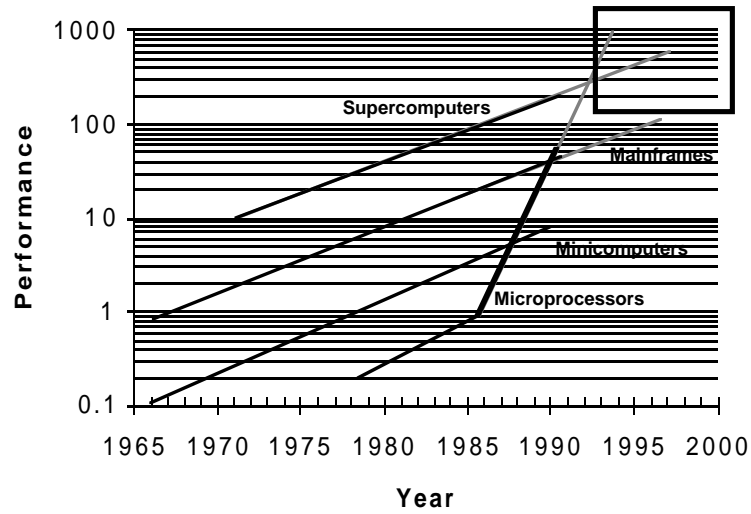


Memory: 4x every 3 years

cs 152 Intro.8

©DAP & SIK 1995

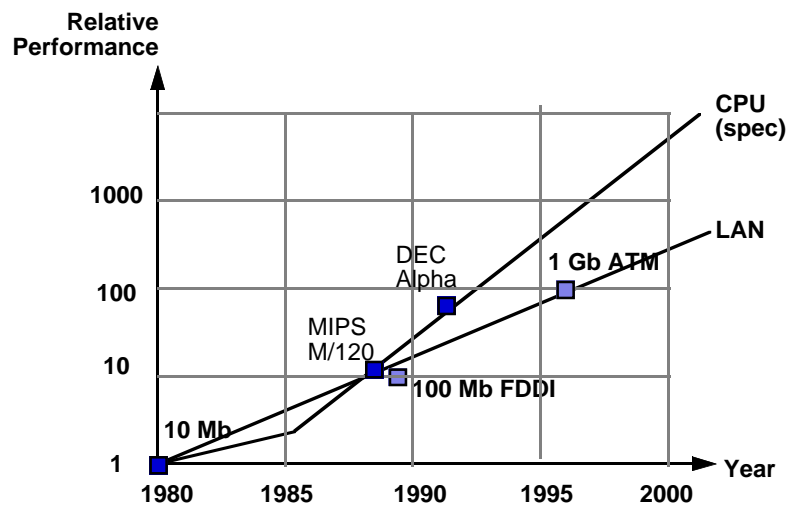
## Performance Trends



cs 152 Intro.9

©DAP & SIK 1995

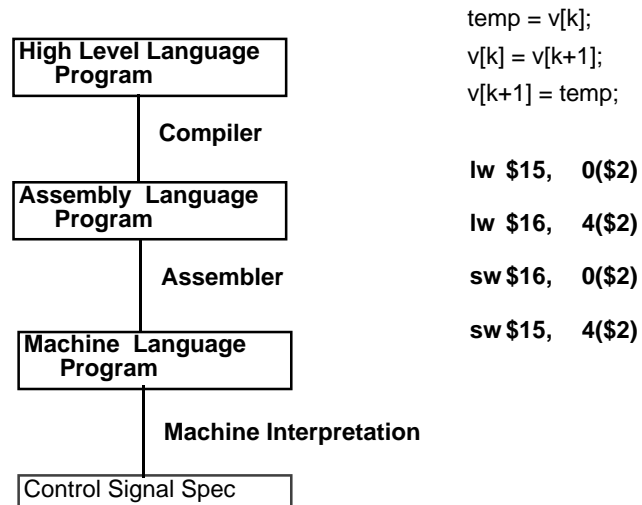
## CPU and LAN Performance



cs 152 Intro.10

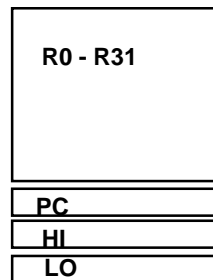
©DAP & SIK 1995

## Levels of Representation

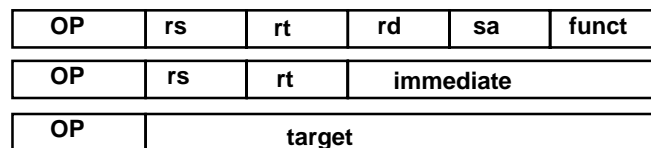


## MIPS R3000 Instruction Set Architecture

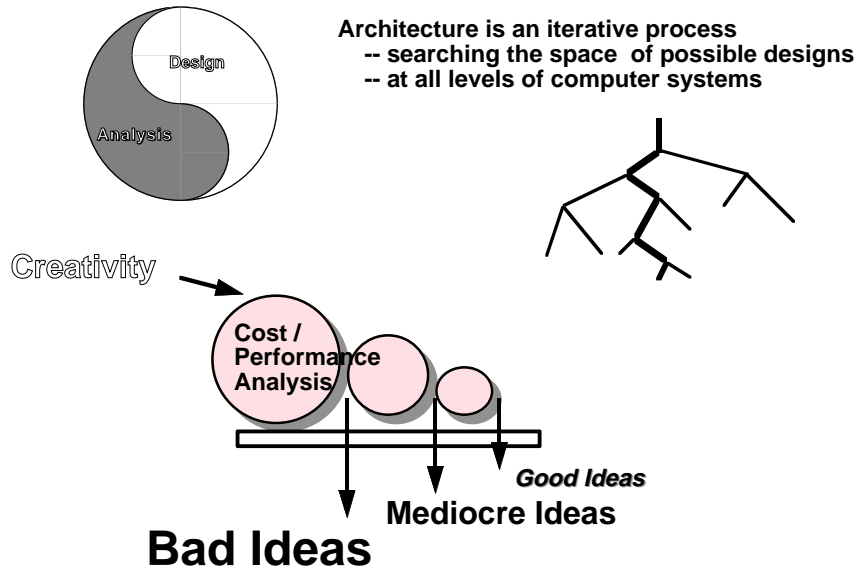
- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



### Instruction Format



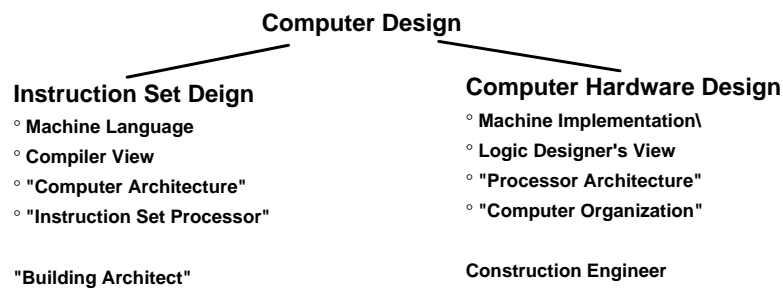
## Measurement and Evaluation



cs 152 Intro.13

©DAP & SIK 1995

## CS152: Course Overview (cont)



*Few people design computers! Very few design instruction sets!*  
*Many people design computer components.*  
*Very many people are concerned with computer function, in detail.*

cs 152 Intro.14

©DAP & SIK 1995

## CS152: So what's in it for me?

- In-depth understanding of the inner-workings of modern computers, their evolution, and trade-offs present at the hardware/software boundary.
  - Insight into fast/slow operations that are easy/hard to implement in hardware
- Experience with the *design process* in the context of a large complex (hardware) design.
  - Functional Spec --> Control & Datapath --> Physical implementation
  - Modern CAD tools
- Designer's "Intellectual" toolbox.

## CS152: Computer Architecture and Engineering

**Instructor:** David A. Patterson      Shing I. Kong ("Kong")  
DAP Office: 635 Soda Hall, 642-6587 [patterson@cs.berkeley.edu](mailto:patterson@cs.berkeley.edu)  
DAP Office Hours: Wed Fri. 2-3 or by appt.  
SIK Office: 615 Soda Hall, 415-786-6377  
[shing.kong@eng.sun.com](mailto:shing.kong@eng.sun.com)  
SIK Office Hours: Wed Fri. 2-3 (lecture day) or by phone

<b>T. A's:</b>	Young Hyun Cho	<a href="mailto:young@uclink.berkeley.edu">young@uclink.berkeley.edu</a>
	Kim Liu	<a href="mailto:kliu@cs.berkeley.edu">kliu@cs.berkeley.edu</a>
	Lloyd Huang	<a href="mailto:lh Huang@cs.berkeley.edu">lh Huang@cs.berkeley.edu</a>
	Nikunj Oza	<a href="mailto:oza@cs.berkeley.edu">oza@cs.berkeley.edu</a>
	Trevor Pering	<a href="mailto:pering@eecs.berkeley.edu">pering@eecs.berkeley.edu</a>
	Mark Spiller	<a href="mailto:mds@ic.eecs.berkeley.edu">mds@ic.eecs.berkeley.edu</a>

**Text:** Computer Organization & Design  
The Hardware / Software Interface



## Course Philosophy

- **Lecture style**
  - 20-Minute Lecture
  - 3- Minute Administrative Matters
  - 25-Minute Lecture
  - 5-Minute Break
  - 25-Minute Lecture
- **Reduce the workload of the class**
  - Final project has been simplified
  - Lab 3 - 6 are directly related to the project
  - Project teams must have at least 4 members
  - No final exam. Only Two mid-terms.
- **Reduce the pressure of taking exams**
  - Both mid-terms will be open book
  - You will have 3 hrs to take the 2-hr test (5-8 PM, Sibley Auditorium)
  - Our goal: test your knowledge

## Simulate Industrial Environment

- **Project teams must have at least 4 members**
- **Communicate with colleagues (team members)**
  - What have you done?
  - What answers you need from others?
  - You must document your work!!!
  - Everyone must keep an on-line notebook
- **Communicate with supervisor (TAs)**
  - How is the team's plan?
  - Short progress reports are required:
    - What is the team's game plan?
    - What is each member's responsibility?

## Course Structure

- Design Intensive Class --- 75 to 150 hours per student  
MIPS Instruction Set ---> Standard-Cell implementation

- Modern CAD System (VIEW<sub>logic</sub>):

### Schematic capture and Simulation

Design Description

Computer-based "breadboard"

- Behavior over time
- Before construction

- Lectures:
  - 1 week on Overview
  - 2 weeks on ISA Design
  - 5 weeks on Proc. Design
  - 4 weeks on Memory and I/O
  - 3 weeks on special topics

## Homework Assignments and Project

- Each assignment consists of two parts
  - Individual Effort: Exercises from the text book
  - Team Effort (After Lab 3): Lab assignments
- All assignments are assigned on Friday and due on a later Monday
- Here is the list of lab assignments:
  - Lab 1 Measure real machines' performance (1 week)
  - Lab 2 MIPS R3000 ISA and SPIM (2 weeks)
  - Lab 3 ALU Design (2 weeks)
  - Lab 4 Single Cycle Processor Design (2 weeks)
  - Lab 5 Pipelined Processor Design (4 weeks - 1 week Spr. Break)
  - Lab 6 Cache Design (2 weeks)
- Final project: Integrate Lab 5 and Lab 6 together
- Safety Net: Cache modules will be provided

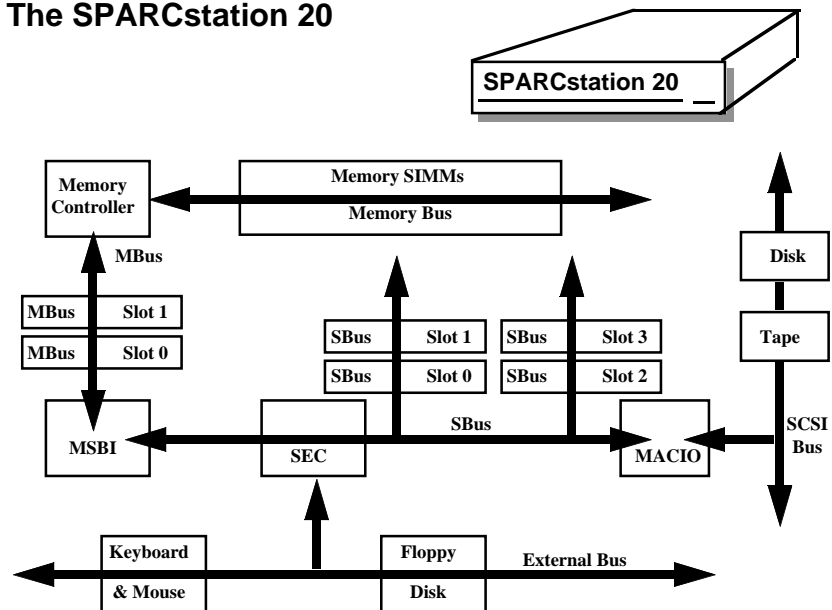
## Course Problems

- No late homeworks or labs (except for lab 5)
- What is cheating?
  - Studying together in groups is encouraged
  - Work must be your own
  - Common examples of cheating: running out of time on a assignment and then pick up output, take homework from box and copy, person asks to borrow solution “just to take a look”, copying an exam question, ...

## Decide on penalties for cheating

- Exercises (book):
  - 0 for problem
  - 0 for homework assignment
  - subtract full value for assignment
  - subtract 2X full value for assignment
- Labs leading to project (groups: only penalize individuals?)
  - 0 for problem
  - 0 for homework assignment
  - subtract full value for assignment
  - subtract 2X full value for assignment
- Exams
  - 0 for problem
  - 0 for exam

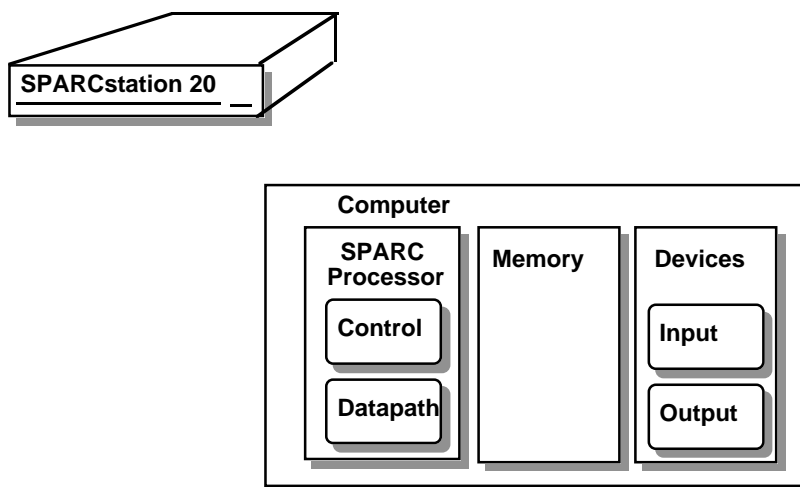
## The SPARCstation 20



cs 152 Intro.23

©DAP & SIK 1995

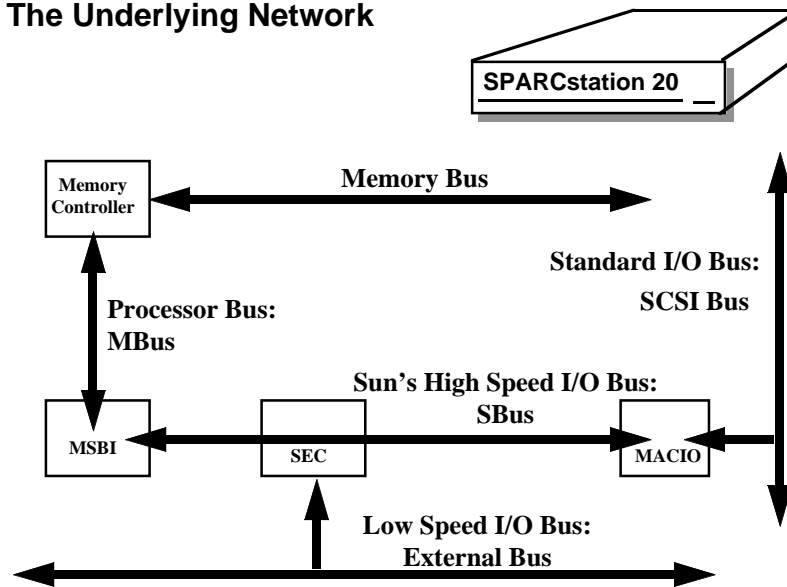
## Levels of Organization



cs 152 Intro.24

©DAP & SIK 1995

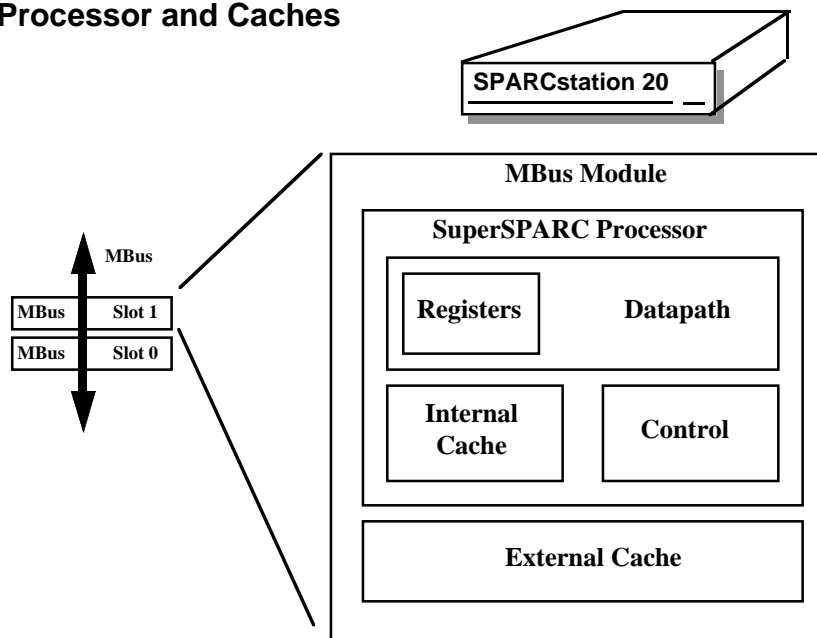
## The Underlying Network



cs 152 Intro.25

©DAP & SIK 1995

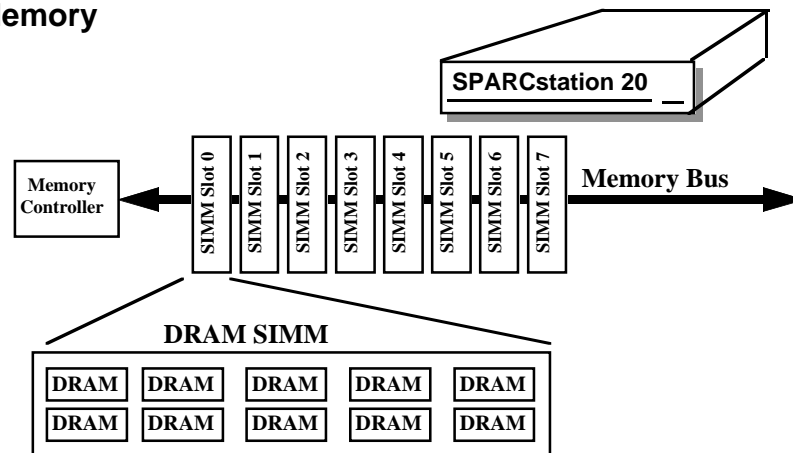
## Processor and Caches



cs 152 Intro.26

©DAP & SIK 1995

## Memory

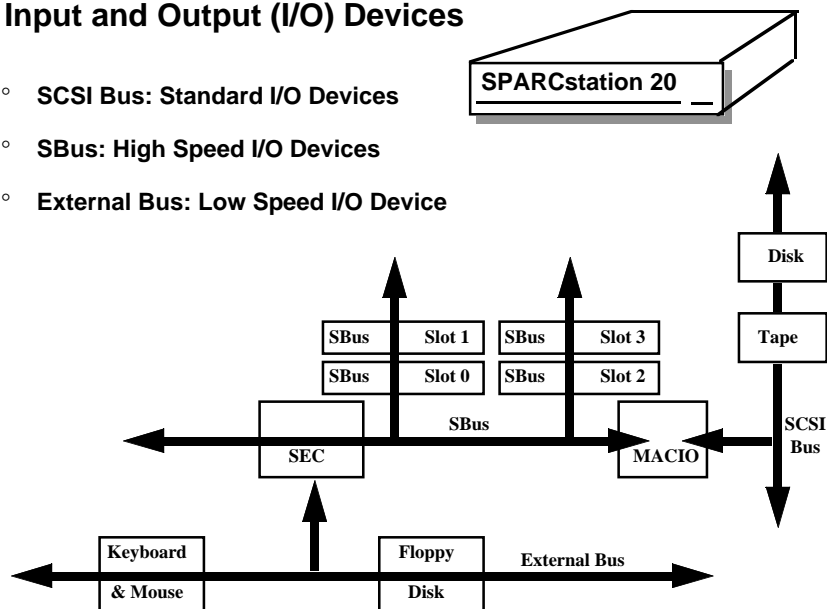


cs 152 Intro.27

©DAP & SIK 1995

## Input and Output (I/O) Devices

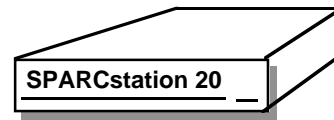
- SCSI Bus: Standard I/O Devices
- SBus: High Speed I/O Devices
- External Bus: Low Speed I/O Device



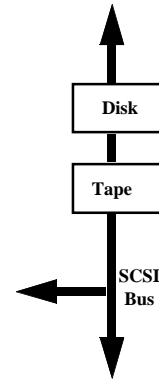
cs 152 Intro.28

©DAP & SIK 1995

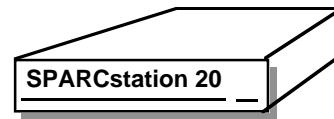
## Standard I/O Devices



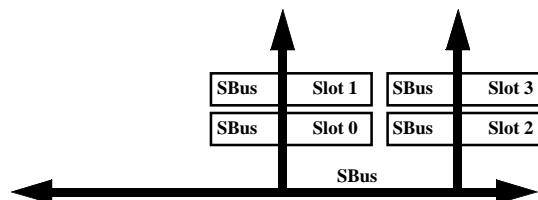
- **SCSI = Small Computer Systems Interface**
- **A standard interface (IBM, Apple, HP, Sun ... etc.)**
- **Computers and I/O devices communicate with each other**
- **The hard disk is one I/O device resides on the SCSI Bus**



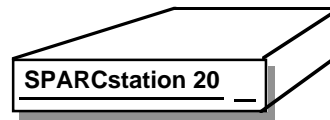
## High Speed I/O Devices



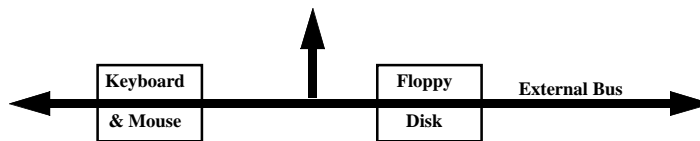
- **SBus is SUN's own high speed I/O bus**
- **SS20 has four SBus slots where we can plug in I/O devices**
- **Example: graphics accelerator, video adaptor, ... etc.**
- **High speed and low speed are relative terms**



## Slow Speed I/O Devices



- There are only four SBus slots in SS20--"seats" are expensive
- The speed of some I/O devices is limited by human reaction time--very very slow by computer standard
- Examples: Keyboard and mouse
- No reason to use up one of the expensive SBus slot



cs 152 Intro.31

©DAP & SIK 1995

## Summary

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all "memory" are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory--we can have more
- Input and output (I/O) devices has the messiest organization
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ... etc.
  - Least amount of research (so far)

cs 152 Intro.32

©DAP & SIK 1995



**CS152**  
**Computer Architecture and Engineering**  
**Lecture 2: Cost and Performance**

January 20, 1995

Dave Patterson (patterson@cs) & Shing Kong (kong@cs)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 Lec2.1

©DAP & SIK 1995

**Overview of Today's Lecture: Cost and Performance**

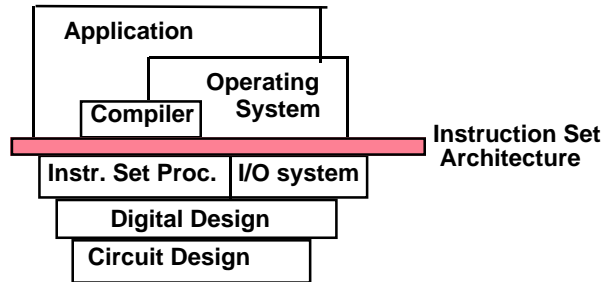
- Review from Last Lecture (2 minutes)
- Cost of Integrated Circuits (20 minutes)
- Administrative Matters (3 minutes)
- Definition and Measures of Performance (25 minutes)
- Break (5 minutes)
- Summarizing Performance and Performance Pitfalls (25 minutes)

cs 152 Lec2.2

©DAP & SIK 1995

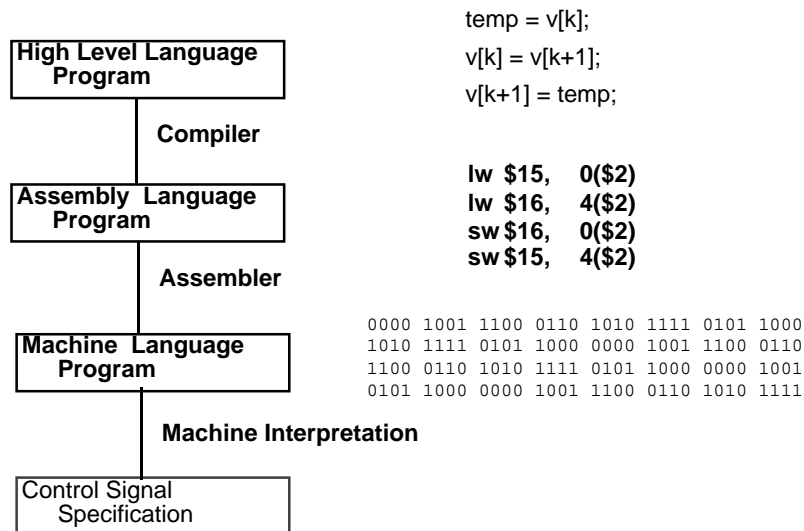
## Review: What is "Computer Architecture"

- Co-ordination of *levels of abstraction*

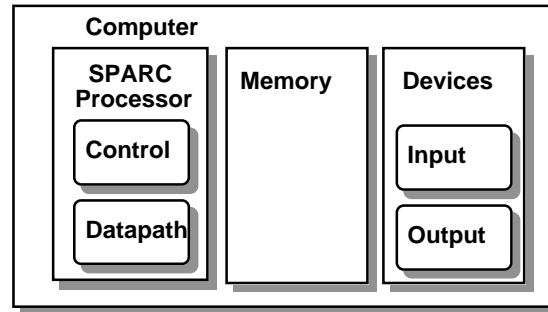
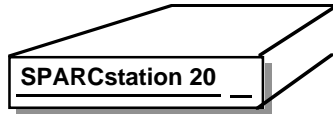


- Under a set of rapidly changing *Forces*

## Review: Levels of Representation



## Review: Levels of Organization



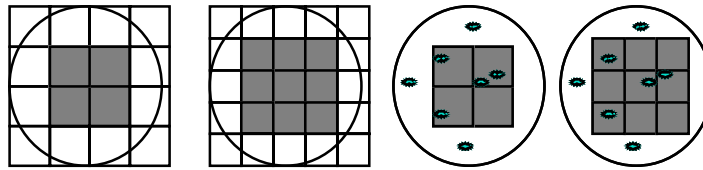
## Review: Summary from Last Lecture

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all “memory” are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory--we can have more
- Input and output (I/O) devices has the messiest organization
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ... etc.
  - Least amount of research (so far)

## Integrated Circuits Costs

$$\text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies per Wafer} * \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi * (\text{Wafer diam} / 2)^2}{\text{Die Area}} - \frac{\pi * \text{Wafer diam}}{\sqrt{2 * \text{Die Area}}} - \text{Test dies} \approx \frac{\text{Wafer Area}}{\text{Die Area}}$$



$$\text{Die Yield} = \text{Wafer yield} * \left\{ 1 + \frac{\text{Defects\_per\_unit\_area} * \text{Die\_Area}}{\alpha} \right\}^{-\alpha}$$

*Die Cost is goes roughly with the cube of the area.*

cs 152 Lec2.7

©DAP & SIK 1995

## Real World Examples

Chip	Metal layers	Line width	Wafer cost	Defect /cm <sup>2</sup>	Area mm <sup>2</sup>	Dies/wafer	Yield	Die Cost
386DX	2	0.90	\$900	1.0	43	360	71%	\$4
486DX2	3	0.80	\$1200	1.0	81	181	54%	\$12
PowerPC 601	4	0.80	\$1700	1.3	121	115	28%	\$53
HP PA 7100	3	0.80	\$1300	1.0	196	66	27%	\$73
DEC Alpha	3	0.70	\$1500	1.2	234	53	19%	\$149
SuperSPARC	3	0.70	\$1700	1.6	256	48	13%	\$272
Pentium	3	0.80	\$1500	1.5	296	40	9%	\$417

From "Estimating IC Manufacturing Costs," by Linley Gwennap, *Microprocessor Report*, August 2, 1993, p. 15

cs 152 Lec2.8

©DAP & SIK 1995

## Other Costs

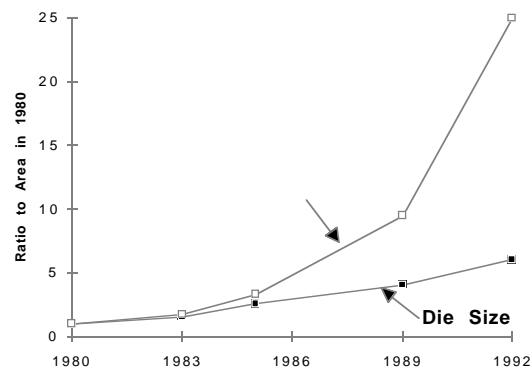
$$\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yield}}$$

Packaging Cost: depends on pins, heat dissipation , ...

<i>Chip</i>	<i>Die cost</i>	<i>Package pins</i>	<i>Package type</i>	<i>cost</i>	<i>Test &amp; Assembly</i>	<i>Total</i>
386DX	\$4	132	QFP	\$1	\$4	\$9
486DX2	\$12	168	PGA	\$11	\$12	\$35
PowerPC 601	\$53	304	QFP	\$3	\$21	\$77
HP PA 7100	\$73	504	PGA	\$35	\$16	\$124
DEC Alpha	\$149	431	PGA	\$30	\$23	\$202
SuperSPARC	\$272	293	PGA	\$20	\$34	\$326
Pentium	\$417	273	PGA	\$19	\$37	\$473

## CMOS improvements

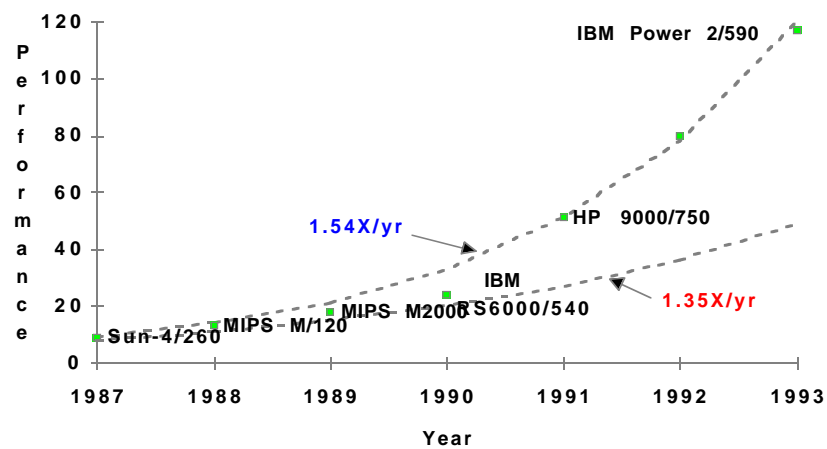
- ° Die size 2X / 3 years; Line widths halve / 7 years



## Technology Trends

	<u>Capacity</u>	<u>Speed</u>
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	1.4x in 10 years
disk	4x in 3 years	1.4x in 10 years

## Processor Performance



## The bottom line: Performance (and cost)

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concorde	3 hours	1350 mph	132	178,200

- **Time to do the task (Execution Time)**
  - execution time, response time, latency
- **Tasks per day, hour, week, sec, ns. .. (Performance)**
  - throughput, bandwidth

## The bottom line: Performance (and cost)

"X is n times faster than Y" means

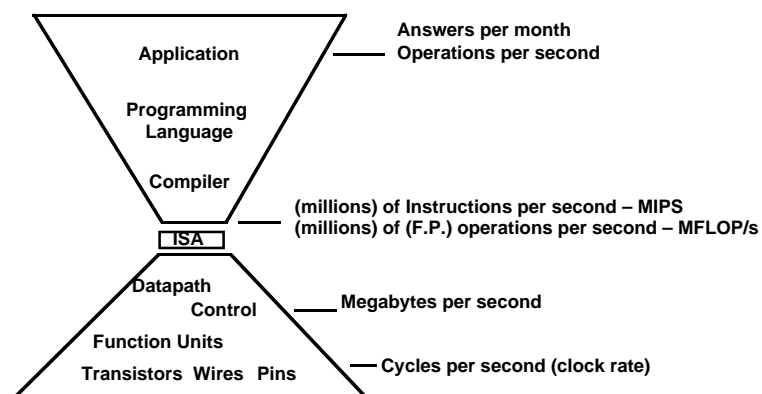
$$\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$

- Time of Concorde vs. Boeing 747?
- Throughput of Boeing 747 vs. Concorde?

## Administrative Matters

- CS152 news group: [ucb.class.cs152](http://ucb.class.cs152)
- Slides available via Mosaic: <http://http.cs.berkeley.edu/~patterson>
- Initial Assignment: Read Chapters 1 and 2 of “Computer Organization and Design”, Exercises 1.1 to 1.26, 1.50 to 1.52, 2.5 to 2.7, 2.14-2.17, 2.33 + run Linpack program on two machines and measure performance
- Class decided on penalties for cheating
  - Exercises (book): 0 for homework assignment
  - Labs leading to project (groups: penalize individuals in groups)
    - 0 for assignment
  - Exams: 0 for exam
- Book is hardcover version
- Other topics?

## Metrics of performance





## Relating Processor Metrics

- CPU execution time = CPU clock cycles/pgm X clock cycle time
- or CPU execution time = CPU clock cycles/pgm ÷ clock rate
- CPU clock cycles/pgm = Instructions/pgm X avg. clock cycles per instr.
- or CPI = CPU clock cycles/pgm ÷ Instructions/pgm
- CPI tells us something about the Instruction Set Architecture, the Implementation of that architecture, and the program measured

## Aspects of CPU Performance

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

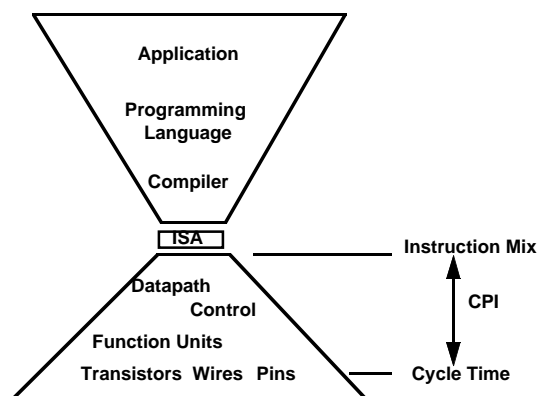
	instr. count	CPI	clock rate
Program			
Compiler			
Instr. Set Arch.			
Organization			
Technology			

## Aspects of CPU Performance

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

	instr count	CPI	clock rate
Program	X		
Compiler	X	(x)	
Instr. Set.	X	X	
Organization		X	X
Technology			X

## Organizational Trade-offs



## CPI

"Average cycles per instruction"

$$\begin{aligned}\text{CPI} &= \text{Instruction Count} / (\text{CPU Time} * \text{Clock Rate}) \\ &= \text{Instruction Count} / \text{Cycles}\end{aligned}$$

$$\text{CPU time} = \text{CycleTime} * \sum_{i=1}^n \text{CPI}_i * I_i$$

"instruction frequency"

$$\text{CPI} = \sum_{i=1}^n \text{CPI}_i * F_i \quad \text{where } F_i = \frac{I_i}{\text{Instruction Count}}$$

**Invest Resources where time is Spent!**

## Example

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	33%
Load	20%	2	.4	27%
Store	10%	2	.2	13%
Branch	20%	2	.4	27%
			<u>1.5</u>	

Typical Mix

## Marketing Metrics

$$\begin{aligned}\text{MIPS} &= \text{Instruction Count} / \text{Time} * 10^6 \\ &= \text{Clock Rate} / \text{CPI} * 10^6\end{aligned}$$

- machines with different instruction sets ?
- programs with different instruction mixes ?
  - dynamic frequency of instructions
- uncorrelated with performance

$$\text{MFLOP/S} = \text{FP Operations} / \text{Time} * 10^6$$

- machine dependent
- often not where time is spent

## Why Do Benchmarks?

- How we evaluate differences
  - Different systems
  - Changes to a single system
- Provide a target
  - Benchmarks should represent large class of important programs
  - Improving benchmark performance should help many programs
- For better or worse, benchmarks shape a field
- Good ones accelerate progress
  - good target for development
- Bad benchmarks hurt progress
  - help real programs v. sell machines/papers?
  - Inventions that help real programs don't help benchmark

## Programs to Evaluate Processor Performance

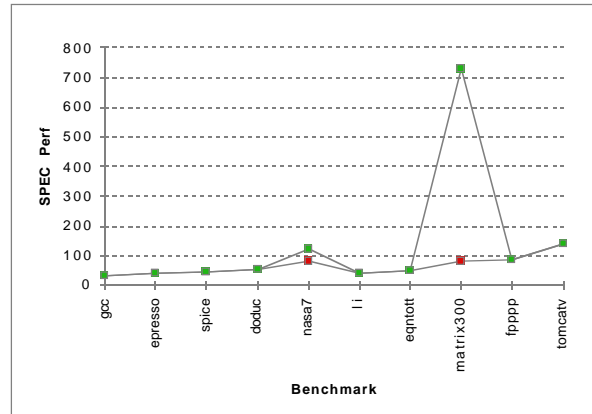
- (Toy) Benchmarks
  - 10-100 line
  - e.g.,: sieve, puzzle, quicksort
- Synthetic Benchmarks
  - attempt to match average frequencies of real workloads
  - e.g., Whetstone, dhrystone
- Kernels
  - Time critical excerpts of real programs
  - e.g., Livermore loops
- Real programs
  - e.g., gcc, spice

## Successful Benchmark: SPEC

- 1987 RISC industry mired in “bench marketing”:  
 (“That is 8 MIPS machine, but they claim 10 MIPS!”)
- EE Times + 5 companies band together to perform Systems Performance Evaluation Committee (SPEC) in 1988:  
 Sun, MIPS, HP, Apollo, DEC
- Create standard list of programs, inputs, reporting: some real programs, includes OS calls, some I/O

## SPEC first round

- First round 1989; 10 programs, single number to summarize performance
- One program: 99% of time in single line of code
- New front-end compiler could improve dramatically



cs 152 Lec2.27

©DAP & SIK 1995

## SPEC Evolution

- Second round; SpecInt92 (6 integer programs) and SpecFP92 (14 floating point programs)
  - Compiler Flags unlimited. March 93 of DEC 4000 Model 610:
  - `spice: unix.c:/def=(sysv,has_bcopy,"bcopy(a,b,c)=memcpy(b,a,c)"`
  - `wave5: /ali=(all,dcom=nat)/ag=a/ur=4/ur=200`
  - `nasa7: /norecu/ag=a/ur=4/ur2=200/lc=blas`
- Add SPECbase: one flag setting for integer programs & 1 for FP
- Third round; 1995; new set of programs
  - "benchmarks useful for 3 years"

cs 152 Lec2.28

©DAP & SIK 1995

## How to Summarize Performance

- Arithmetic mean (or weighted arithmetic mean) tracks execution time:  
 $\text{SUM}(T_i)/n$  or  $\text{SUM}(W_i \cdot T_i)$
- Harmonic mean (or weighted harmonic mean) of rates (e.g., MFLOPS) tracks execution time:  
 $n/\text{SUM}(1/R_i)$  or  $n/\text{SUM}(W_i/R_i)$
- Normalized execution time is handy for scaling performance  
(e.g., time on reference machine  $\div$  time on measured machine)
- But do not take the arithmetic mean of normalized execution time, use the geometric mean ( $\text{prod}(R_i)^{1/n}$ )
- Alas, geometric mean rewards all improvements equally:  
program A going from 2 seconds to 1 second as important as  
program B going from 2000 seconds to 1000 seconds

cs 152 Lec2.29

©DAP & SIK 1995

## Impact of Means on SPECmark89 for IBM 550

Program	Ratio to VAX:Time:				Weighted Time:	
	Before	After	Before	After	Before	After
gcc	30	29	49	51	8.91	9.22
espresso	35	34	65	67	7.64	7.86
spice	47	47	510	510	5.69	5.69
doduc	46	49	41	38	5.81	5.45
nasa7	78	144	258	140	3.43	1.86
li	34	34	183	183	7.86	7.86
eqntott	40	40	28	28	6.68	6.68
matrix300	78	730	58	6	3.43	0.37
fpppp	90	87	34	35	2.97	3.07
tomcatv	133	138	20	19	2.01	1.94
Mean	54	72	124	108	54.42	49.99
		Geometric	Arithmetic	Weighted Arith.		
		Ratio	Ratio	Ratio	Ratio	Ratio
		1.33	1.16	1.09		

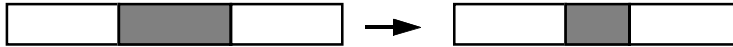
cs 152 Lec2.30

©DAP & SIK 1995

## 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 then,

$$\text{ExTime}(\text{with E}) = ((1-F) + F/S) \times \text{ExTime}(\text{without E})$$

$$\text{Speedup}(\text{with E}) = \frac{\text{ExTime}(\text{without E})}{((1-F) + F/S) \times \text{ExTime}(\text{without E})}$$

## Cost Summary

- Integrated circuits driving computer industry
- Die costs goes up with the cube of die area

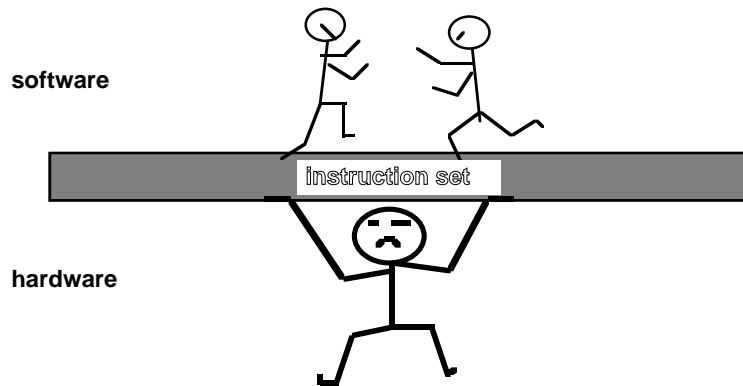


## Performance Evaluation Summary

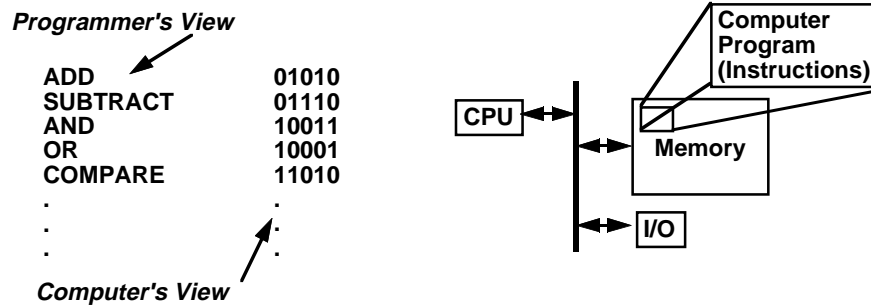
CPU time	=	$\frac{\text{Seconds}}{\text{Program}}$	=	$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$
----------	---	---	---	---

- Time is the measure of computer performance!
- Good products created when have:
  - Good benchmarks
  - Good ways to summarize performance
- If not good benchmarks and summary, then choice between improving product for real programs vs. improving product to get more sales=> sales almost always wins
- Remember Amdahl's Law: Speedup is limited by unimproved part of program

## Instruction Set Design



## Instruction Set Architecture



### Princeton (Von Neumann) Architecture

- Data and Instructions mixed in same memory ("stored program computer")
- Program as data (dubious advantage)
- Storage utilization
- Single memory interface

### Harvard Architecture

- Data & Instructions in separate memories
- Has advantages in certain high performance implementations

## Basic Issues in Instruction Set Design

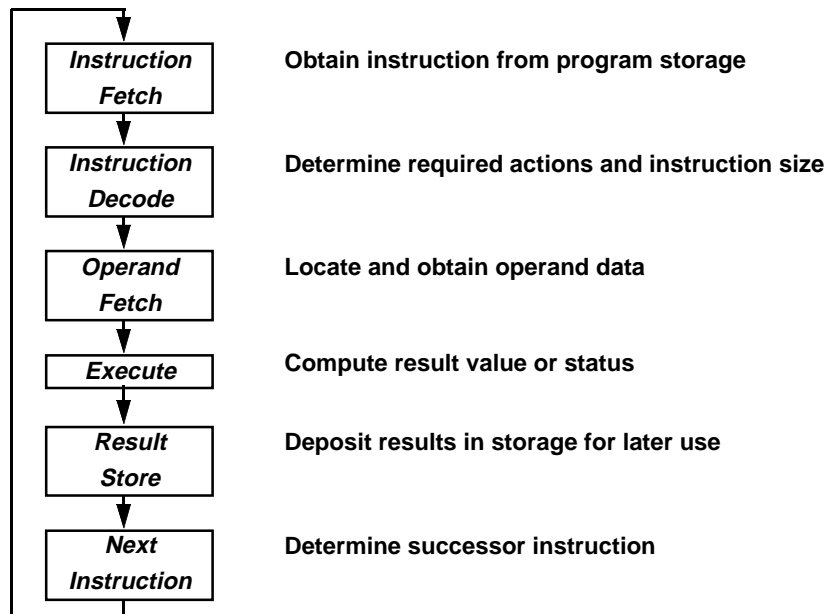
- What operations (and how many) should be provided
  - LD/ST/INC/BRN sufficient to encode any computation
  - But not useful because programs too long!
- How (and how many) operands are specified
  - Most operations are dyadic (eg,  $A \leftarrow B + C$ )
  - Some are monadic (eg,  $A \leftarrow \sim B$ )
- How to encode these into consistent instruction formats
  - Instructions should be multiples of basic data/address widths

*Typical instruction set:*

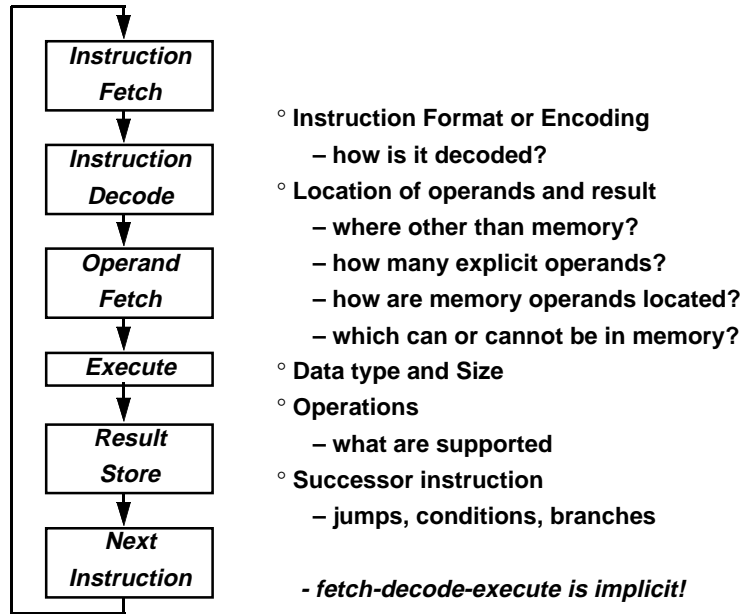
- 32 bit word
- basic operand addresses are 32 bits long
- basic operands, like integers, are 32 bits long
- in general case, instruction could reference 3 operands ( $A := B + C$ )

challenge: encode operations in a small number of bits!

## Execution Cycle



## What Must be Specified?



cs 152 Lec 3 ISA.5

© Kong/Patterson 1995

## Topics to be covered

- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Operations
- Instruction Format or Encoding
  - how is it decoded?
- Data type and Size
  - what are supported

cs 152 Lec 3 ISA.6

© Kong/Patterson 1995

## Basic ISA Classes

---

### Accumulator:

1 address	add A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$
1+x address	addx A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A + x]$

### Stack:

0 address	add	$\text{tos} \leftarrow \text{tos} + \text{next}$
-----------	-----	--

### General Purpose Register:

2 address	add A B	$\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$
3 address	add A B C	$\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

### Load/Store:

3 address	add Ra Rb Rc	$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$
	load Ra Rb	$\text{Ra} \leftarrow \text{mem}[\text{Rb}]$
	store Ra Rb	$\text{mem}[\text{Rb}] \leftarrow \text{Ra}$

### Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

---

## Comparing Number of Instructions

◦ Code sequence for  $C = A + B$  for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

## General Purpose Registers Dominate

- Since 1975 all machines use general purpose registers
- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g.,  $(A*B) - (C*D) - (E*F)$  can do multiplies in any order vs. stack
  - registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

## Examples of Register Usage

Number of memory addresses per typical ALU instruction

Maximum number of operands per typical ALU instruction

Examples

0	3	SPARC, MIPS, Precision Architecture, Power PC
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

## Pros and Cons of Number Memory Operands/Operands

- **Register–register: 0 memory operands/instr, 3 (register) operands/instr**
  - + Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute
  - Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.
- **Register–memory (1,2)**
  - + Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.
  - Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.
- **Memory–memory (3,3)**
  - + Most compact. Doesn't waste registers for temporaries.
  - Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

## Summary on Instruction Classes

- Expect new instruction set architecture to use general purpose register
- Pipelining => Expect it to use load store variant of GPR ISA

## Administrative Matters

---

- CS152 news group: [ucb.class.cs152](http://ucb.class.cs152)
- Slides available via Mosaic: <http://http.cs.berkeley.edu/~patterson>
- Video tapes of lectures available for viewing in 205 McLaughlin, Mon-Fri 8AM to 5PM
- 1st Assignment due Monday; Second Assignment handed out Friday: Questions, Problems?
- Other topics?

## Memory Addressing

---

- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
  - Since could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto words?
  - Can a word be placed on any byte boundary?



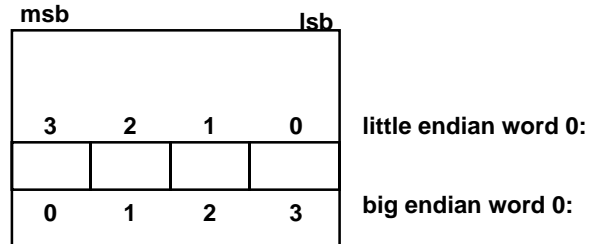
## Addressing Objects

**Big Endian:** address of most significant byte = word address  
(xx00 = Big End of word)

- IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

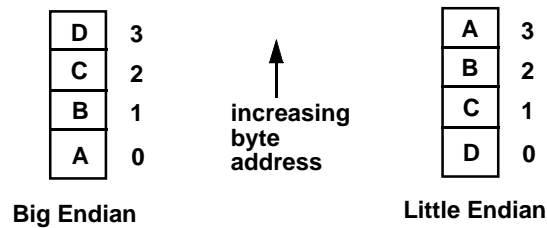
**Little Endian:** address of least significant byte = word address  
(xx00 = Little End of word)

- Intel 80x86, DEC Vax



**Alignment:** require that objects fall on address that is multiple of their size.

## Byte Swap Problem



When words are transferred between Big Endian and Little Endian machines, you must permute the bytes to successfully copy the data

Each system is self-consistent, but causes problems when they need communicate!

## Addressing Modes

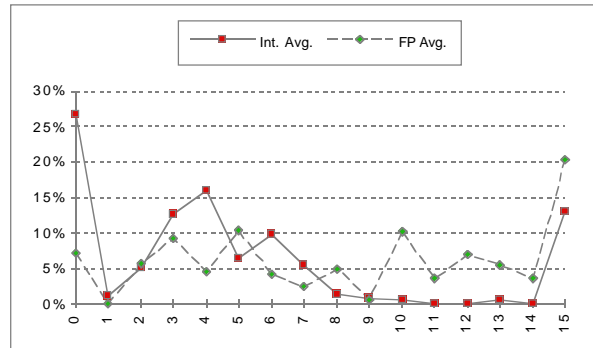
Addressing mode	Example	Meaning
Register	Add R4,R3	$R4 \leftarrow R4 + R3$
Immediate	Add R4,#3	$R4 \leftarrow R4 + 3$
Displacement	Add R4,100(R1)	$R4 \leftarrow R4 + \text{Mem}[100 + R1]$
Register indirect	Add R4,(R1)	$R4 \leftarrow R4 + \text{Mem}[R1]$
Indexed	Add R3,(R1+R2)	$R3 \leftarrow R3 + \text{Mem}[R1 + R2]$
Direct or absolute	Add R1,(1001)	$R1 \leftarrow R1 + \text{Mem}[1001]$
Memory indirect	Add R1,@(R3)	$R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]]$
Auto-increment	Add R1,(R2)+	$R1 \leftarrow R1 + \text{Mem}[R2]; R2 \leftarrow R2 + d$
Auto-decrement	Add R1,-(R2)	$R2 \leftarrow R2 - d; R1 \leftarrow R1 + \text{Mem}[R2]$
Scaled	Add R1,100(R2)[R3]	$R1 \leftarrow R1 + \text{Mem}[100 + R2 + R3 * d]$

## Addressing Mode Usage

3 programs measured on machine with all address modes (VAX)

--- Displacement:	42% avg, 32% to 55%
--- Immediate:	33% avg, 17% to 43%
--- Register deferred (indirect):	13% avg, 3% to 24%
--- Scaled:	7% avg, 0% to 16%
--- Memory indirect:	3% avg, 1% to 6%
--- Misc:	2% avg, 0% to 3%

## Displacement Address Size



- Average of 5 programs from SPECint92 and Average of 5 programs from SPECfp92
- X-axis is in powers of 2: 4  $\Rightarrow$  addresses  $> 2^3$  (8) and  $\leq 2^4$  (16)
- 1% of addresses  $> 16$ -bits

## Immediate Size

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits

## Addressing Summary

---

- Data Addressing modes that are important:  
Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate size should be 8 to 16 bits

## Typical Operations

---

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Logical	not, and, or, set, clear
Shift	shift left/right, rotate left/right
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate

## Top 10 80x86 Instructions

---

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

Simple instructions dominate instruction frequency

## Methods of Testing Condition

---

### Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3  
bz label

### Condition Register

Ex: cmp r1, r2, r3  
bgt r1, label

### Compare and Branch

Ex: bgt r1, r2, label

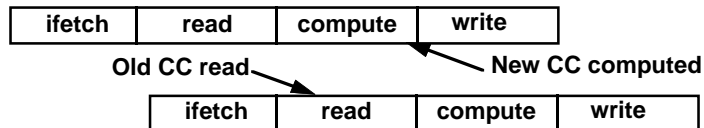
## Condition Codes

Setting CC as side effect can reduce the # of instructions

X: . . . SUB r0, #1, r0 BRP X	vs.	X: . . . SUB r0, #1, r0 CMP r0, #0 BRP X
---	-----	---

But also has disadvantages:

- not all instructions set the condition codes which do and which do not often confusing!  
*e.g., shift instruction sets the carry bit*
- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC



## Branches

--- Conditional control transfers

*Four basic conditions:*

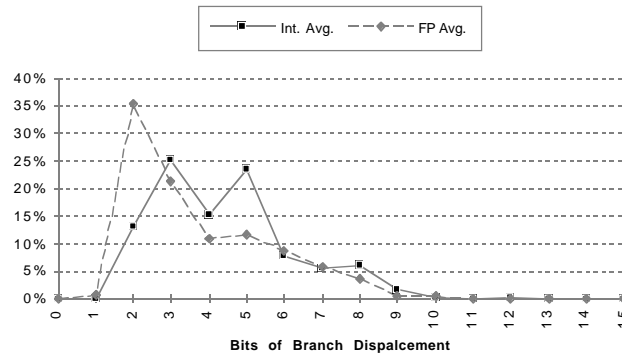
N -- negative  
Z -- zero

V -- overflow  
C -- carry

Sixteen combinations of the basic four conditions:

Always	Unconditional
Never	NOP
Not Equal	$\sim Z$
Equal	Z
Greater	$\sim[Z + (N \oplus V)]$
Less or Equal	$Z + (N \oplus V)$
Greater or Equal	$\sim(N \oplus V)$
Less	$N \oplus V$
Greater Unsigned	$\sim(C + Z)$
Less or Equal Unsigned	$C + Z$
Carry Clear	$\sim C$
Carry Set	C
Positive	$\sim N$
Negative	N
Overflow Clear	$\sim V$
Overflow Set	V

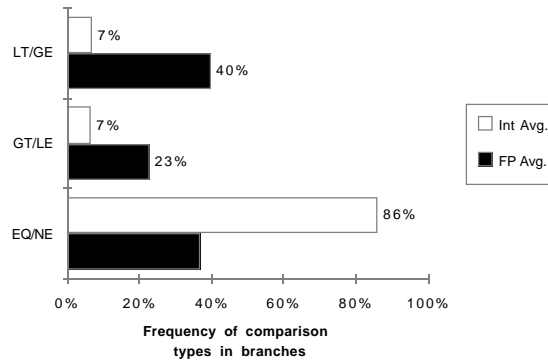
## Conditional Branch Distance



- Distance from branch in instructions  $2^i \Rightarrow \leq \pm 2^{i-1} \ \& \ > 2^{i-2}$
- 25% of integer branches are  $> 2 \ \& \ \leq 4$  or  $-2$  to  $-4 \ \&$

## Conditional Branch Addressing

- PC-relative since most branches are relatively close to the current PC address
- At least 8 bits suggested ( $\pm 128$  instructions)
- Compare Equal/Not Equal most important for integer programs



## Operation Summary

---

- Support these simple instructions, since they will dominate the number of instructions executed:

load,  
store,  
add,  
subtract,  
move register-register,  
and,  
shift,  
compare equal, compare not equal,  
branch (with a PC-relative address at least 8-bits long),  
jump,  
call,  
return;

## Data Types

---

**Bit:** 0, 1

**Bit String:** sequence of bits of a particular length

4 bits is a nibble

8 bits is a byte

16 bits is a half-word (VAX: word)

32 bits is a word (VAX: long word)

**Character:**

ASCII 7 bit code

EBCDIC 8 bit code

**Decimal:**

digits 0-9 encoded as 0000b thru 1001b

two decimal digits packed per 8 bit byte

**Integers:**

Sign & Magnitude: 0X vs. 1X

1's Complement: 0X vs. 1(~X)

2's Complement: 0X vs. (1's comp) + 1

Positive #'s same in all

First 2 have two zeros

Last one usually chosen

**Floating Point:**

Single Precision

Double Precision

Extended Precision

M x RE

mantissa

exponent

base

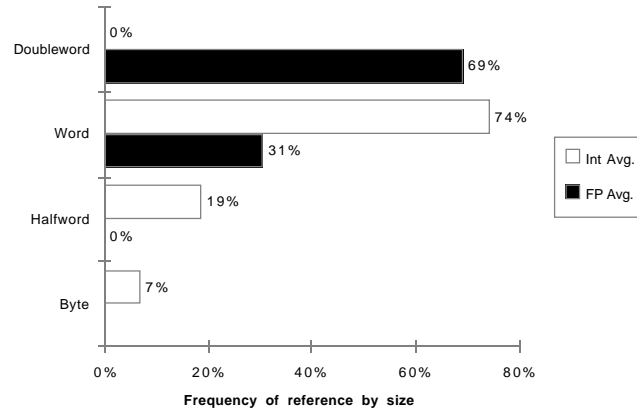
How many +/- #'s?

Where is decimal pt?

How are +/- exponents represented?



## Operand Size Usage



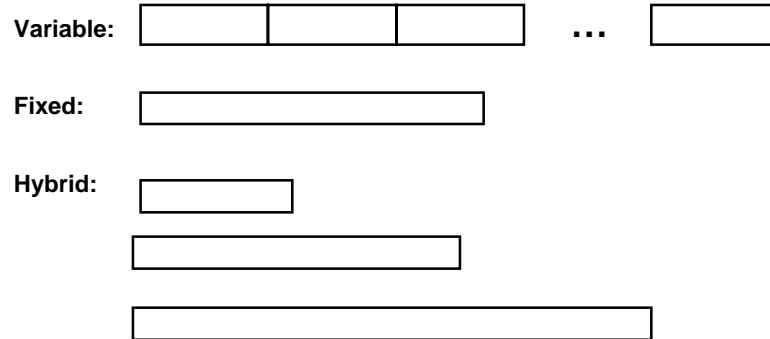
- Support these data sizes and types:  
8-bit, 16-bit, 32-bit integers and  
32-bit and 64-bit IEEE 754 floating point numbers

## Instruction Format

- If have many memory operands per instructions and many addressing modes, need an Address Specifier per operand
- If have load-store machine with 1 address per instr. and one or two addressing modes, then just encode addressing mode in the opcode

## Generic Examples of Instruction Formats

---



## Summary of Instruction Formats

---

- If code size is most important,  
use variable length instructions
- If performance is over is most important,  
use fixed length instructions

## Compilers and Instruction Set Architectures

---

- **Ease of compilation**

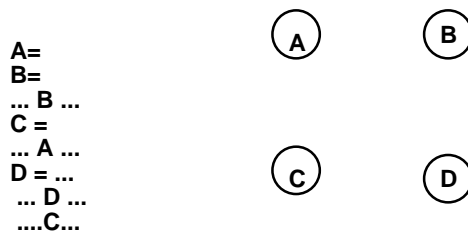
- orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
- completeness: support for a wide range of operations and target applications
- regularity: no overloading for the meanings of instruction fields
- streamlined: resource needs easily determined

- **Register Assignment is critical too**

## Modern Register Assignment

---

- Keep args and local variables in registers
    - unless their "location" is obtained.
  - Assign registers by "graph coloring"
- Works well if at least 16 registers



## Summary of Compiler Considerations

---

- Provide at least 16 general purpose registers plus separate floating-point registers,
- Be sure all addressing modes apply to all data transfer instructions,
- Aim for a minimalist instruction set.

## Instruction Set Metrics

---

### *Design-time metrics:*

- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

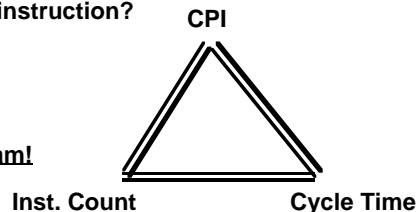
### *Static Metrics:*

- How many bytes does the program occupy in memory?

### *Dynamic Metrics:*

- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

**Best Metric:** Time to execute the program!



**NOTE:** this depends on instructions set, processor organization, and compilation techniques.

## Lecture Summary: ISA

---

- Use general purpose registers with a load-store architecture;
- Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred;
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 64-bit IEEE 754 floating point numbers;
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size;
- Provide at least 16 general purpose registers plus separate floating-point registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.

**CS152**  
**Computer Architecture and Engineering**  
**Lecture 4: MIPS Instruction Set Architecture**

**January 27, 1995**

**Dave Patterson (patterson@cs) & Shing Kong (kong@cs)**

**Slides available on <http://http.cs.berkeley.edu/~patterson>**

cs 152 Lec4.1

©DAP & SIK 1995

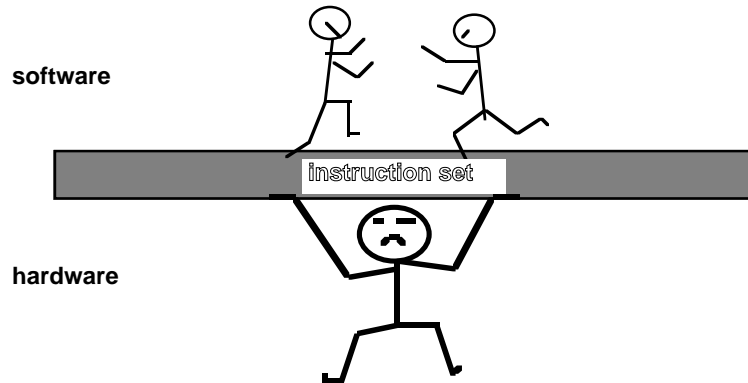
**Overview of Today's Lecture: MIPS et al**

- **Review from Last Lecture (3 minutes)**
- **MIPS ISA (20 minutes)**
- **Administrative Matters (3 minutes)**
- **More MIPS (25 minutes)**
- **Break (5 minutes)**
- **MIPS (VAX, 80x86?) (25 minutes)**

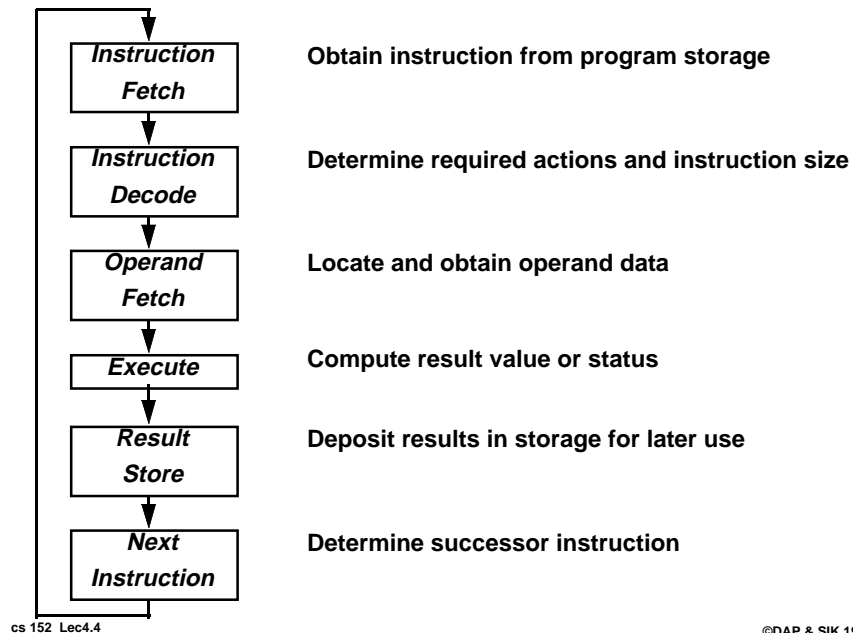
cs 152 Lec4.2

©DAP & SIK 1995

## Review: Instruction Set Design



## Execution Cycle

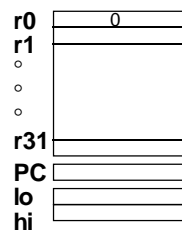


## Review: Summary

- Use general purpose registers with a load-store architecture;
- Provide at least 16 general purpose registers plus separate floating-point registers,
- Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred;
- Be sure all addressing modes apply to all data transfer instructions,
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size;
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers;
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
- Aim for a minimalist instruction set.

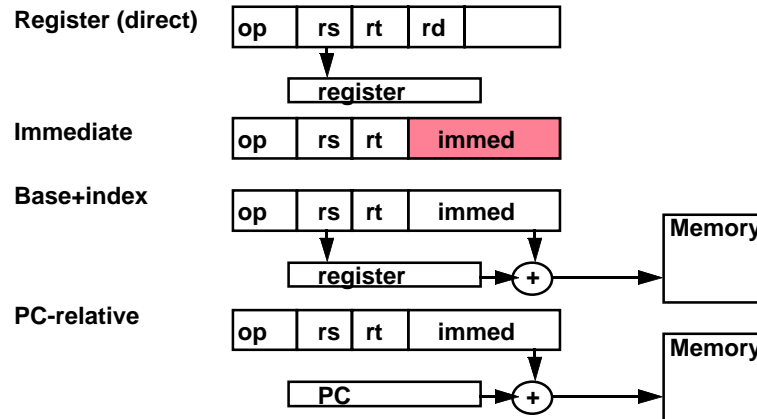
## MIPS R2000 / R3000 Registers

- Programmable storage
  - $2^{32}$  x bytes
  - 31 x 32-bit GPRs ( $R0 = 0$ )
  - 32 x 32-bit FP regs (paired DP)
  - HI, LO, PC





## MIPS Addressing Modes/Instruction Formats

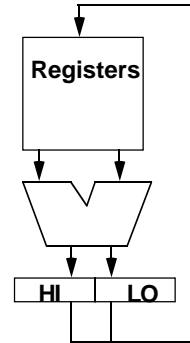


## MIPS R2000 / R3000 Operation Overview

- Arithmetic logical
  - Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
  - Addl, AddIU, SLTI, SLTIU, Andl, Orl, Xorl, Lui
  - SLL, SRL, SRA, SLLV, SRLV, SRAV
- Memory Access
  - LB, LBU, LH, LHU, LW, LWL, LWR
  - SB, SH, SW, SWL, SWR

## Multiply / Divide

- Start multiply, divide
  - MULT rs, rt
  - MULTU rs, rt
  - DIV rs, rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd
  - MFLO rd
- Move to HI or LO
  - MTHI rd
  - MTLO rd



## MIPS arithmetic instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comments</u>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; exception possible
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; exception possible
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \$3$	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \$3$	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \$3$ , Hi = $\$2 \bmod \$3$	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = $\$2 \div \$3$ , Hi = $\$2 \bmod \$3$	Unsigned quotient & remainder
Move from Hi	mfhi \$1	$\$1 = \text{Hi}$	Used to get copy of Hi
Move from Lo	mflo \$1	$\$1 = \text{Lo}$	Used to get copy of Lo

## MIPS logical instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comment</u>
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	$\$1 = \$2   \$3$	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	$\$1 = \$2 \oplus \$3$	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	$\$1 = \sim(\$2   \$3)$	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	$\$1 = \$2 \& 10$	Logical AND reg, constant
or immediate	ori \$1,\$2,10	$\$1 = \$2   10$	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	$\$1 = \sim\$2 \& \sim 10$	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant
shift right arithm.	sra \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	$\$1 = \$2 \ll \$3$	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right arith. by variable

## MIPS data transfer instructions

<u>Instruction</u>	<u>Comment</u>
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

## Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset    if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset    <>
- Compare to zero and Branch
  - BLEZ rs, offset    if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset    >
  - BLT    <
  - BGEZ    >=
  - BLTZAL rs, offset    if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL    >=
- Remaining set of compare and branch take two instructions
- Almost all comparisons are against zero!

## MIPS jump, branch, compare instructions

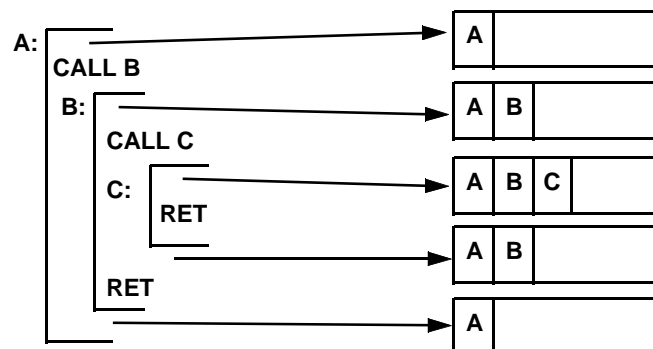
<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>
branch on equal	beq \$1,\$2,100	if (\$1 == \$2) go to PC+4+100 <i>Equal test; PC relative branch</i>
branch on not eq.	bne \$1,\$2,100	if (\$1!= \$2) go to PC+4+100 <i>Not equal test; PC relative</i>
set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; 2's comp.</i>
set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 <i>Compare &lt; constant; 2's comp.</i>
set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; natural no.</i>
set l. t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 <i>Compare &lt; constant; natural</i>
jump	j 10000	go to 10000 <i>Jump to target address</i>
jump register	jr \$31	go to \$31 <i>For switch, procedure return</i>
jump and link	jal 10000	\$31 = PC + 4; go to 10000 <i>For procedure call</i>

## Administrative Matters

- First Assignment due Monday at 4PM
  - Please put discussion section number or time on homework!
- Second Assignment: Read Chapter 3 and Appendix A of “Computer Organization and Design”, do Exercises + run MIPS program on SPIM and debug broken SPIM
  - Exercises due following Monday (2/6)
  - Lab (Problems 0,1,2,3) due 2 Mondays later (2/13)
  - Start soon! 2 weeks because its more than one weeks work
  - Problems 0, 1, 2 & Exercises by yourself
  - Problem 3 only of lab in pairs
    - Pairs randomly assigned in discussion sections
  - Estimate time spent on prior assignment for feedback
- Other topics?

## Why Are Stacks So Great?

*Stacking of Subroutine Calls & Returns and Environments:*



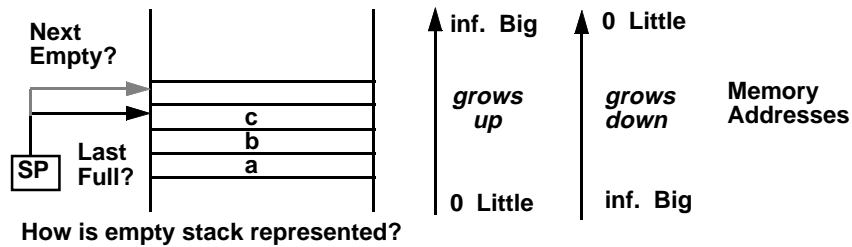
Some machines provide a memory stack as part of the architecture (e.g., the VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

## Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

*Stacks that Grow Up vs. Stacks that Grow Down:*



Little --> Big/Last Full

POP: Read from Mem(SP)  
Decrement SP

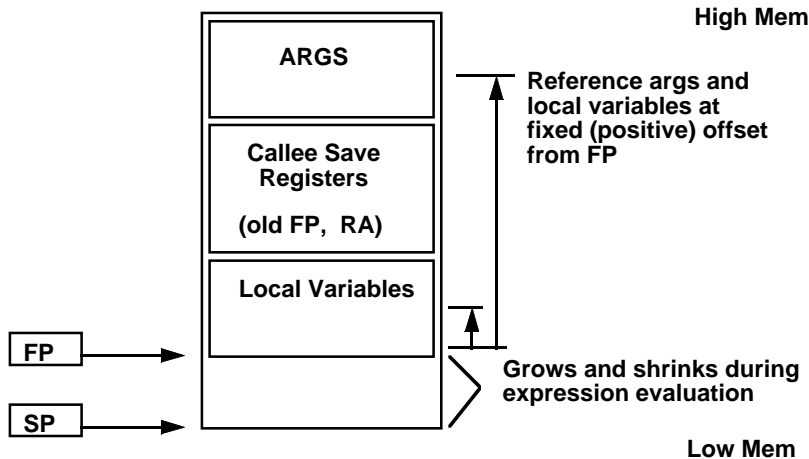
PUSH: Increment SP  
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP  
Read from Mem(SP)

PUSH: Write to Mem(SP)  
Increment SP

## Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next )
- Block structured languages contain link to lexically enclosing frame.

## MIPS: Software conventions

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation and
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
...		
15	t7	
16	s0	callee saves
...		
23	s7	
24	t8	
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

cs 152 Lec4.19

©DAP & SIK 1995

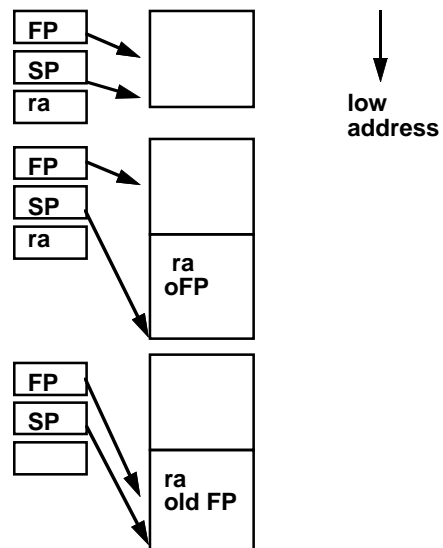
## MIPS / GCC Calling Conventions

fact:

```

addiu $sp, $sp, -32
sw    $ra, 20($sp)
sw    $fp, 16($sp)
addu  $fp, $sp, 32
...
sw    $a0, 0($fp)
...
lw    $31, 20($sp)
lw    $fp, 16($sp)
addiu $sp, $sp, 32
jr    $31

```



First four args passed in registers.

cs 152 Lec4.20

©DAP & SIK 1995

## Example in C: swap

```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Assume temp is register \$15; arguments in \$a1, \$a2; \$16 is scratch reg:
- Write MIPS code

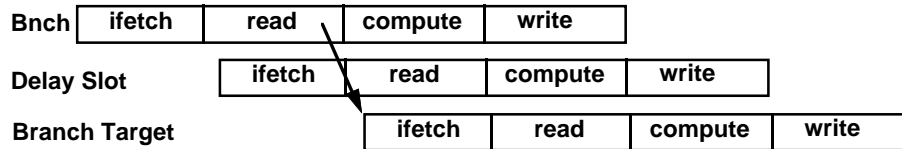
## swap: MIPS

swap:

```
addi $sp,$sp,-4
sw   $16, 8($sp)
sll  $t2, $a2, 2
add  $t2, $a1, $t2
lw   $15, 0($t2)
lw   $16, 4($t2)
sw   $16, 0($t2)
sw   $15, 4($t2)
lw   $16, 8($sp)
addiu $sp,$sp, 4
jr   $31
```

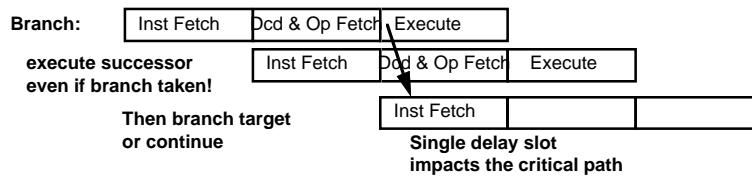


## Branch & Pipelines



By the end of the read stage of the Branch instruction, the CPU knows whether or not the branch will take place. However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

## Delayed Branches: Redefine behavior



- Compiler can fill a single delay slot with a useful instruction 50% of the time.
- try to move down from above jump
- move up from target, if safe

```
add r3, r1, r2
sub r4, r4, 1
bz  r4, LL
...

LL: add rd, ...
```

## Standard and Delayed Interpretation

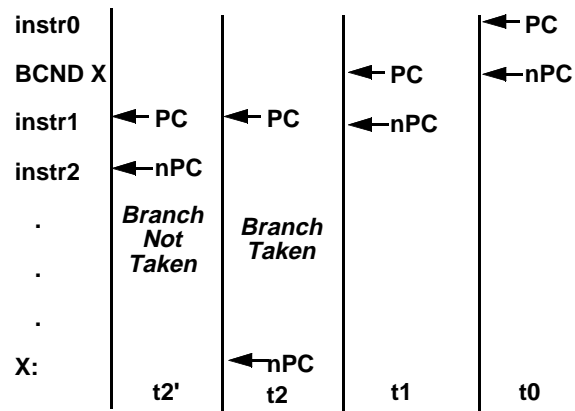
PC	add rd, rs, rt	$R[rd] \leftarrow R[rs] + R[rd];$ $PC \leftarrow PC + 4;$
	beq rs, rt, offset	if $R[rd] == R[rt]$ then $PC \leftarrow PC + SX(offset)$ else $PC \leftarrow PC + 4;$
	sub rd, rs, rt	...
	...	
	L1: target	

---

PC	add rd, rs, rt	$R[rd] \leftarrow R[rs] + R[rd];$ $PC \leftarrow nPC; \quad nPC \leftarrow nPC + 4;$
nPC	beq rs, rt, offset	if $R[rd] == R[rt]$ then $nPC \leftarrow nPC + SX(offset)$ else $nPC \leftarrow nPC + 4;$
		$PC \leftarrow nPC$
	sub rd, rs, rt	...
	...	
	L1: target	<u>Delayed Loads?</u>

## Delayed Branches (cont.)

### Execution History



Branches are the bane of pipelined machines  
 Delayed branches complicate the compiler slightly, but make pipelining easier to implement and more effective  
 Good strategy to move some complexity to compile time

## Miscellaneous MIPS instructions

- **break**                      A breakpoint trap occurs, transfers control to exception handler
- **syscall**                    A system trap occurs, transfers control to exception handler
- **coprocessor instrs.**      Support for floating point: discussed later
- **TLB instructions**        Support for virtual memory: discussed later
- **restore from exception**   Restores previous interrupt mask & kernel/user mode bits into status register
- **load word left/right**      Supports misaligned word loads
- **store word left/right**     Supports misaligned word stores

## Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Branch and jump instructions put the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, ...)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates are zero extended to 32 bits
  - arithmetic immediates are sign extended to 32 bits
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

## Other ISAs

- Intel 8086/88 => 80286 => 80386 => 80486 => Pentium => P6
  - 8086 few transistors to implement 16-bit microprocessor
  - tried to be somewhat compatible with 8-bit microprocessor 8080
  - successors added features which were missing from 8086 over next 15 years
  - product several different intel enigneers over 10 to 15 years
  - Announced 1978
- VAX simple compilers & small code size =>
  - efficient instruction encoding
  - powerful addressing modes
  - powerful instructions
  - few registers
  - product of a single talented architect
  - Announced 1977

## Machine Examples: Address & Registers

Intel 8086	$2^{20}$ x 8 bit bytes AX, BX, CX, DX SP, BP, SI, DI CS, SS, DS IP, Flags	acc, index, count, quot stack, string code,stack,data segment
VAX 11	$32$ 2 x 8 bit bytes 16 x 32 bit GPRs	r15-- program counter r14-- stack pointer r13-- frame pointer r12-- argument ptr
MC 68000	$2^{24}$ x 8 bit bytes 8 x 32 bit GPRs 7 x 32 bit addr reg 1 x 32 bit SP 1 x 32 bit PC	
MIPS	$32$ 2 x 8 bit bytes 32 x 32 bit GPRs 32 x 32 bit FPRs HI, LO, PC	

## VAX Operations

- **General Format:**  
**(operation) (datatype) (2, 3)**  
2 or 3 explicit operands

- **For example**

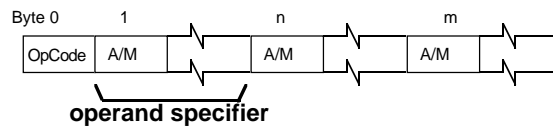
**add (b, w, l, f, d) (2, 3)**

**Yields**

**addb2 addw2 addl2 addf2 addd2**  
**addb3 addw3 addl3 addf3 addd3**

## VAX format, addressing modes

### General Instruction Format



register	5	r	
autoinc	8	r	
disp	A	r	byte
	C	r	half word
	E	r	word
index	4	r	m r displacement

## swap: MIPS vs. VAX

swap:

addiu \$sp,\$sp, -4	.word ^m<r0,r1,r2,r3>
sw \$16, 8(\$sp)	
sll \$t2, \$a2,2	movl r2, 4(a)
add \$t2, \$a1,\$t2	movl r1, 8(a)
lw \$15, 0(\$t2)	movl r3, (r2)[r1]
lw \$16, 4(\$t2)	addl3 r0, #1,8(ap)
sw \$16, 0(\$t2)	movl (r2)[r1],(r2)[r0]
sw \$15, 4(\$t2)	movl (r2)[r0],r3
lw \$16, 8(\$sp)	
addiu \$sp,\$sp, 4	
jr \$31	ret

## Summary

- Use general purpose registers with a load-store architecture: YES
- Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR
- Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)
- All addressing modes apply to all data transfer instructions : YES
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b
- Aim for a minimalist instruction set: YES

## Summary: Salient features of MIPS R3000

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =
  - no condition codes
- **Delayed branch**
  - execute instruction after the branch (or jump) even if the branch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)

**CS152**  
**Computer Architecture and Engineering**  
**Lecture 5: Technology & Delay Modeling**

**February 1, 1995**

**Dave Patterson (patterson@cs) and**  
**Shing Kong (shing.kong@eng.sun.com)**

**Slides available on <http://http.cs.berkeley.edu/~patterson>**



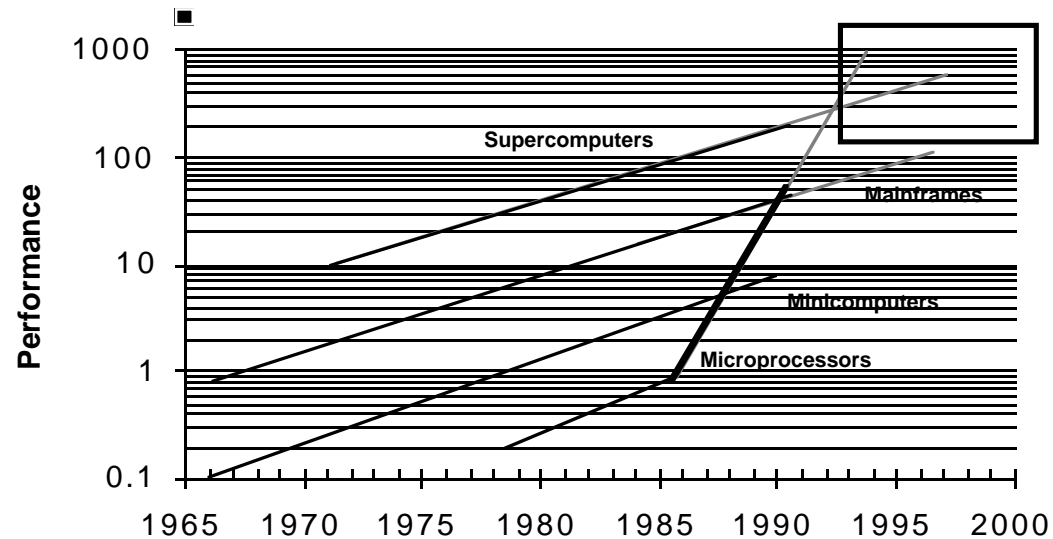
# Recap of Last Lecture

- Use general purpose registers with a load-store architecture
  - Provide at least 16 GP registers and separate FP registers
- Support the following addressing modes:
  - Displacement (with an address offset size of 12 to 16 bits)
  - Immediate with size 8 to 16 bits
  - Register deferred
- All addressing modes apply to all data transfer instructions
- Use fixed instruction encoding if interested in performance  
Use variable instruction encoding if interested in code size
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers  
32-bit and 64-bit IEEE floating point numbers
- Support these simple instructions: load, store, add, subtract, move  
register-register, and, shift, compare equal, compare not equal, branch,  
jump, call, and return
- Aim for a minimalist instruction set

# Outline of Today's Lecture

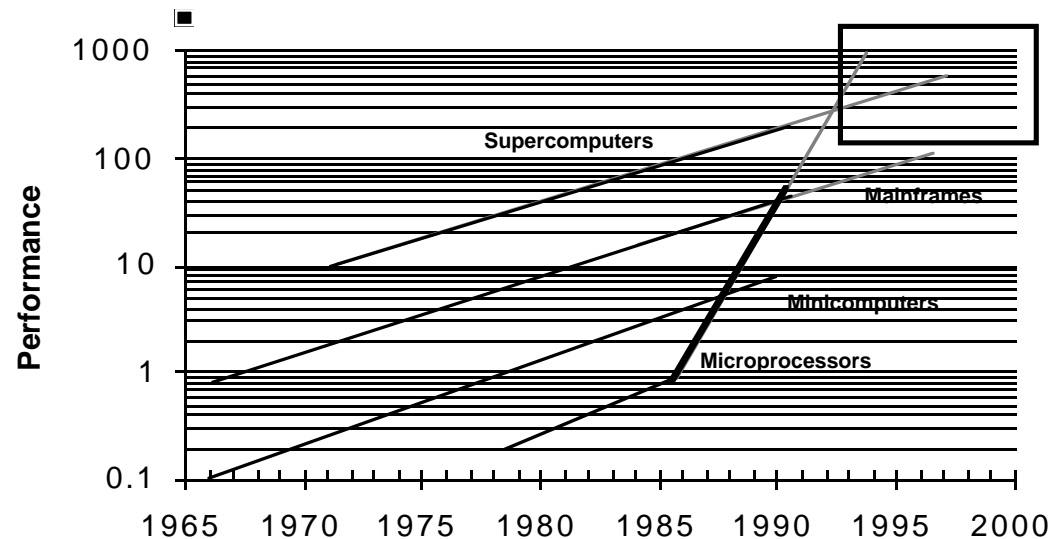
- **Recap of Last Lecture and Introduction of Today's Lecture (4 min.)**
- **Performance and Technology trends (16 minutes)**
- **Administrative Matters and Questions (5 minutes)**
- **Delay Modeling and Gate Characterization (25 minutes)**
- **Questions and Break (5 minutes)**
- **Clocking Methodologies and Timing Considerations (25 minutes)**

# Performance and Technology Trends



- **Feature Size: shrinks 10% / yr.**
  - **Switching speed improves 1.2 / yr.**
- **Density: improves 1.2x / yr.**
- **Die Area: 1.2x / yr.**
- **Technology Power:  $1.2 \times 1.2 \times 1.2 = 1.7 \text{ x / year}$**

# Performance and Technology Trends (Continue)



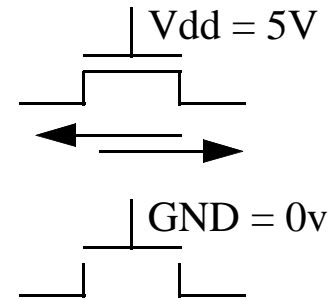
- **Compute Power:**
  - Prior to 1985: less than 1.7 x / year
  - After 1985: Microprocessor improves greater than 1.7 x / year
- The lesson of RISC is to keep the ISA as simple as possible:
  - Shorter design cycle => fully exploit the advancing technology.
  - Advance pipeline techniques
  - Bigger and more sophisticated on-chip caches

# Basic Technology: CMOS

- **CMOS: Complementary Metal Oxide Semiconductor**
  - **NMOS (N-Type Metal Oxide Semiconductor) transistors**
  - **PMOS (P-Type Metal Oxide Semiconductor) transistors**

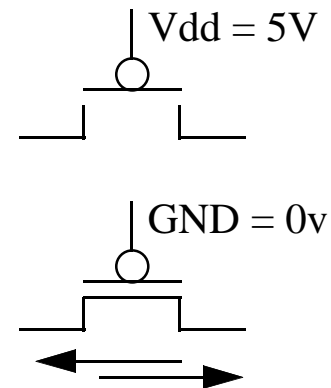
- **NMOS Transistor**

- **Apply a HIGH ( $V_{dd}$ , 5V) to its gate turns the transistor into a “conductor”**
- **Apply a LOW (GND, 0v) to its gate shuts off the conduction path**



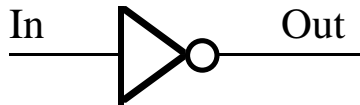
- **PMOS Transistor**

- **Apply a HIGH ( $V_{dd}$ , 5V) to its gate shuts off the conduction path**
- **Apply a LOW (GND, 0v) to its gate turns the transistor into a “conductor”**

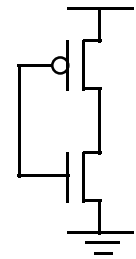
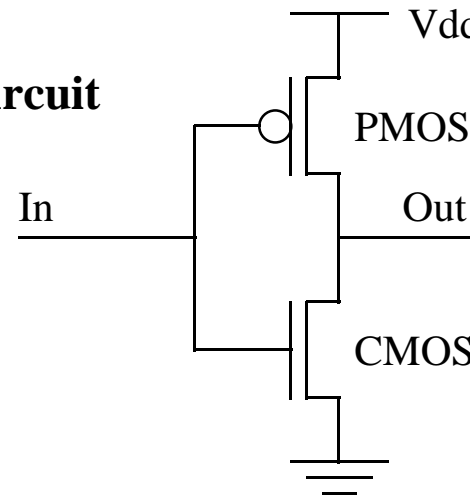


# Basic Components: CMOS Inverter

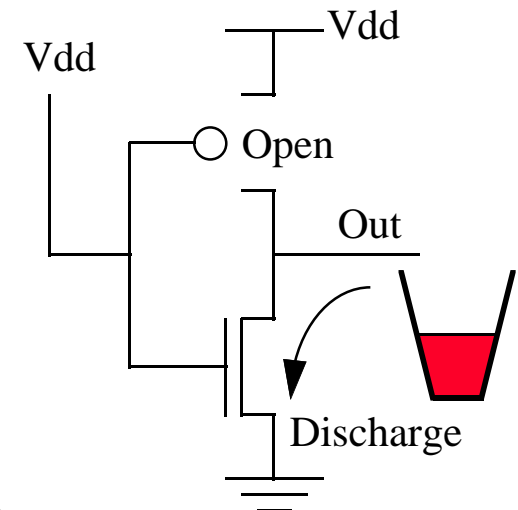
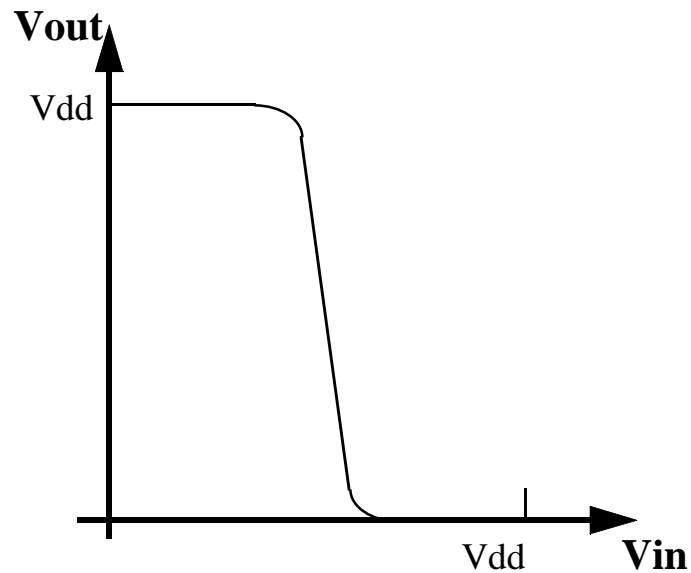
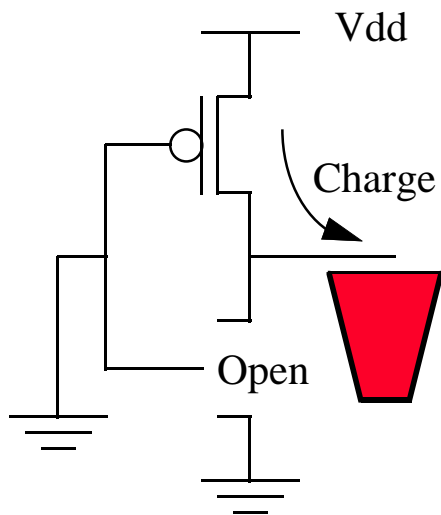
Symbol



Circuit

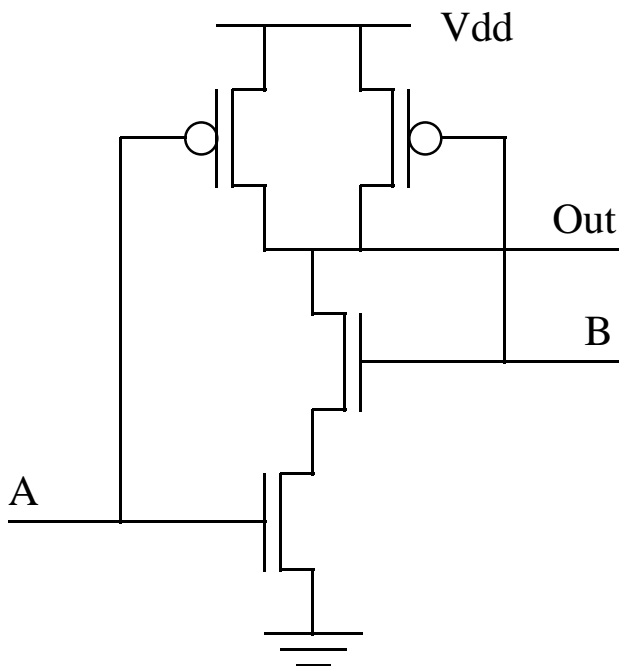
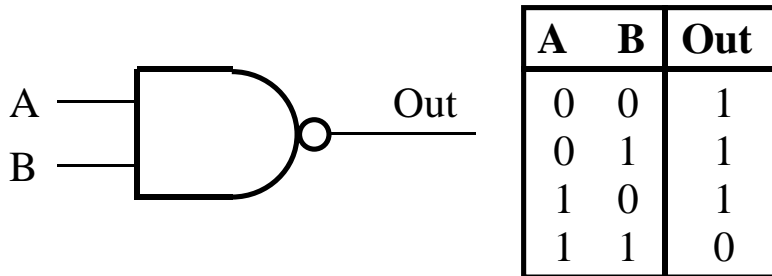


## ◦ Inverter Operation

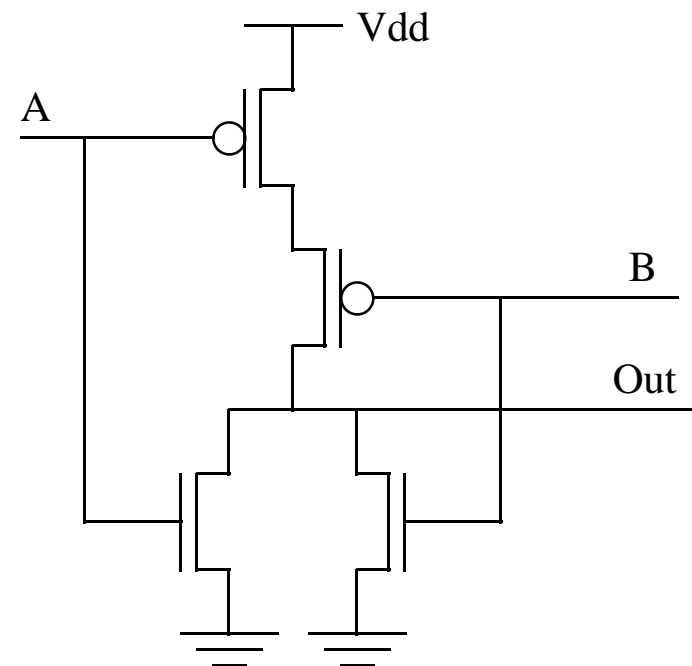
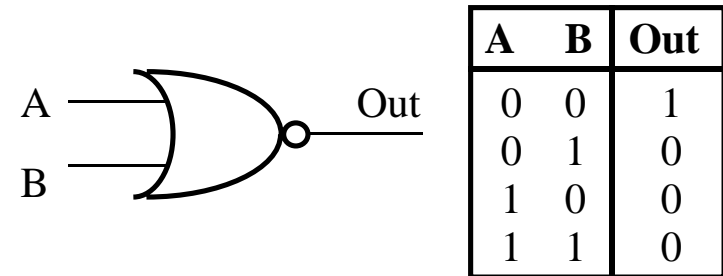


# Basic Components: CMOS Logic Gates

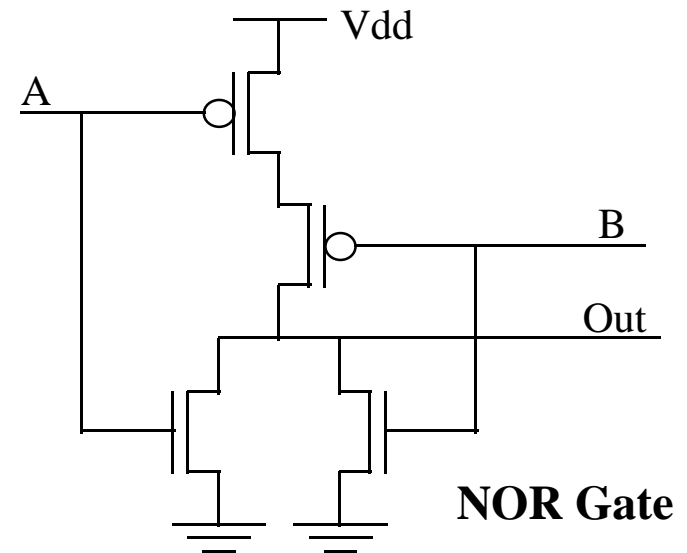
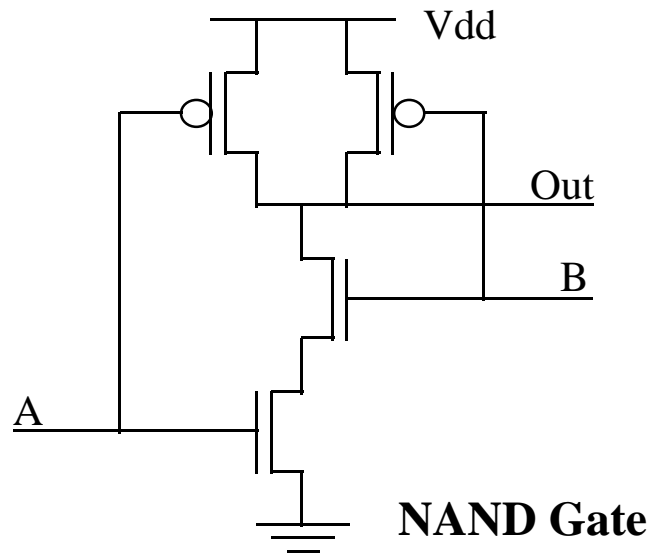
## NAND Gate



## NOR Gate



# Gate Comparison

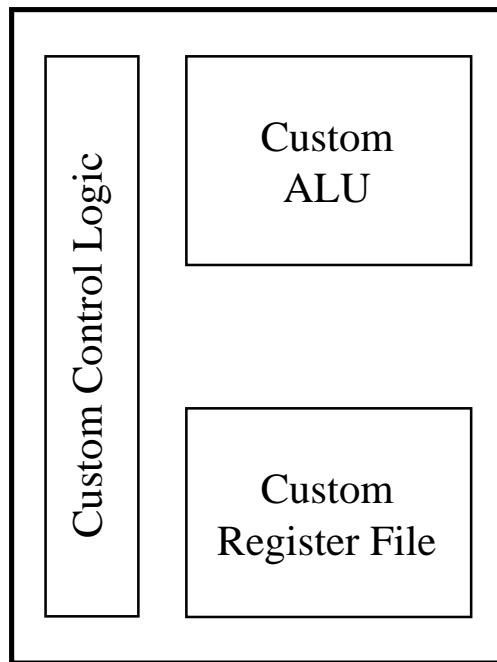


- If PMOS transistors is faster:
  - It is OK to have PMOS transistors in series
  - NOR gate is preferred
  - NOR gate is preferred also if  $H \rightarrow L$  is more critical than  $L \rightarrow H$
- If NMOS transistors is faster:
  - It is OK to have NMOS transistors in series
  - NAND gate is preferred
  - NAND gate is preferred also if  $L \rightarrow H$  is more critical than  $H \rightarrow L$

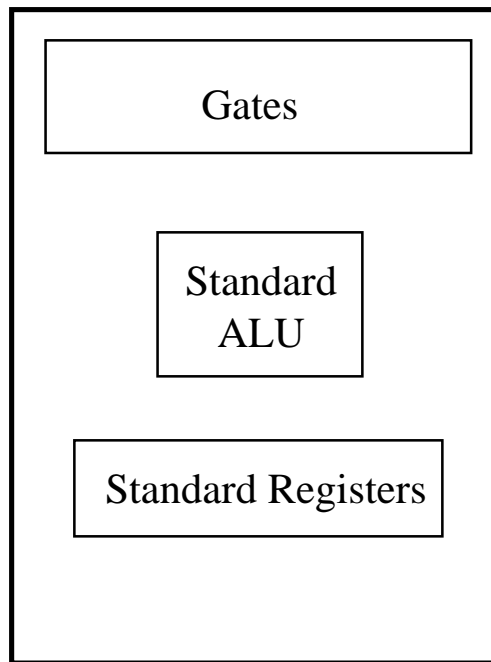


# Range of Design Style

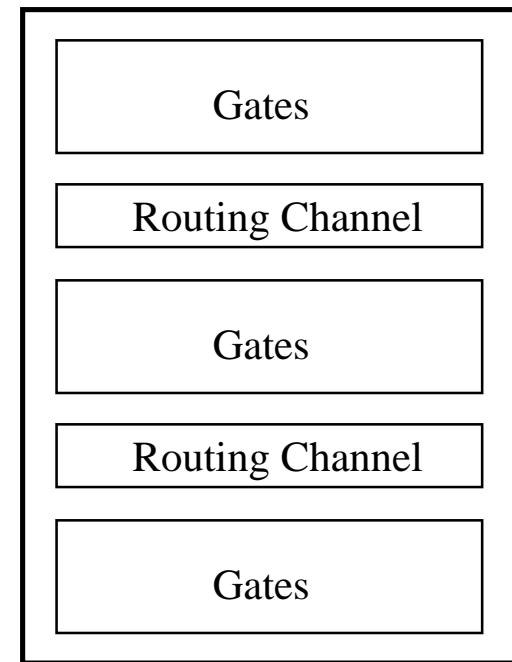
**Custom Design**



**Standard Cell**



**Gate Array**

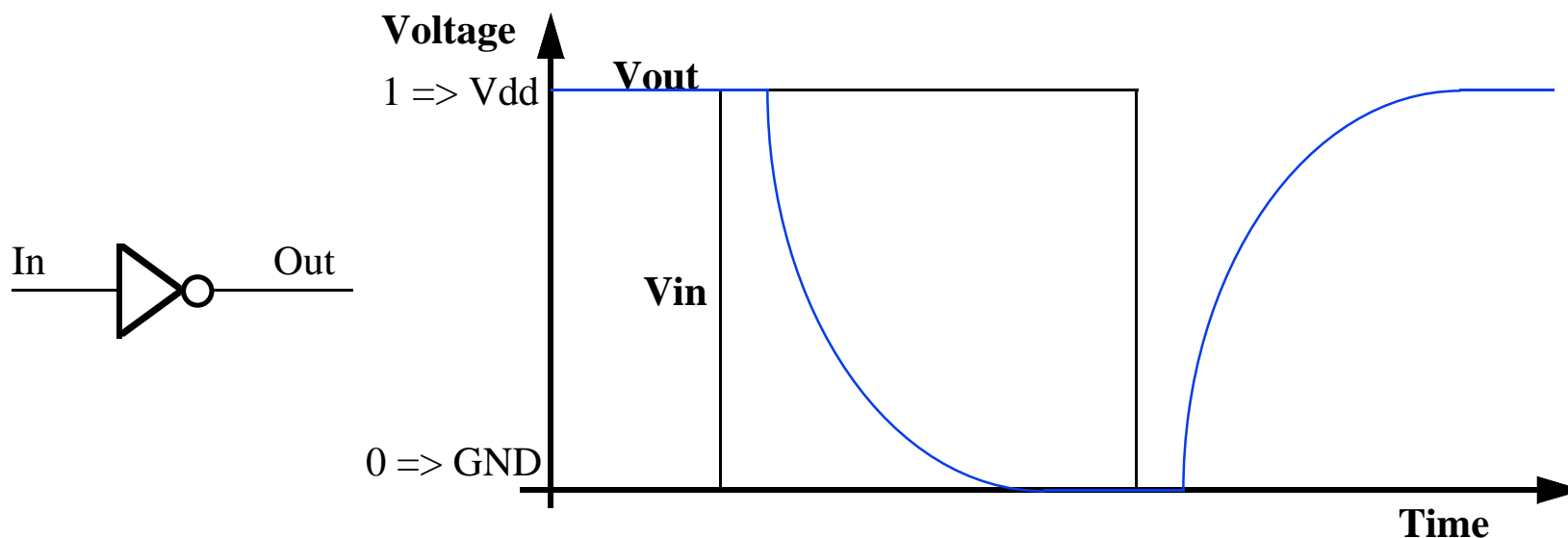


# Questions and Administrative Matters

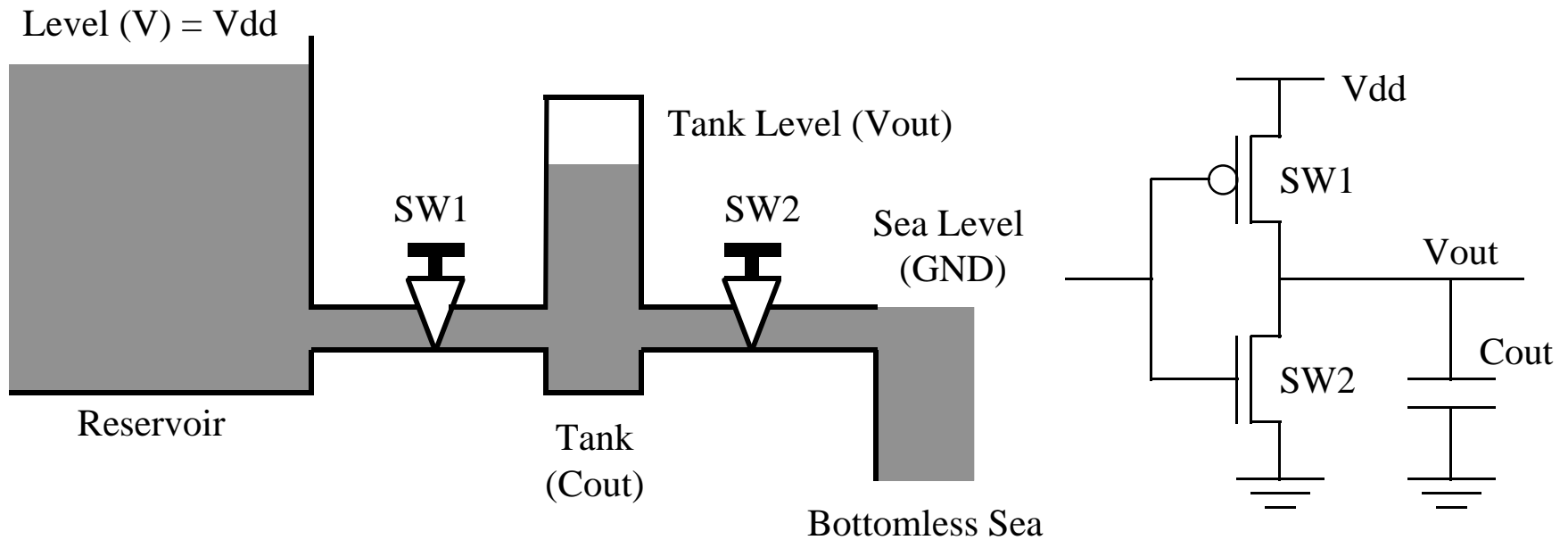
- **CS152 Logic and Storage Components**

# Ideal (CS) versus Reality (EE)

- When input 0  $\rightarrow$  1, output 1  $\rightarrow$  0 but NOT instantly
  - Output goes 1  $\rightarrow$  0: output voltage goes from Vdd (5v) to 0v
- When input 1  $\rightarrow$  0, output 0  $\rightarrow$  1 but NOT instantly
  - Output goes 0  $\rightarrow$  1: output voltage goes from 0v to Vdd (5v)
- Voltage does not like to change instantaneously

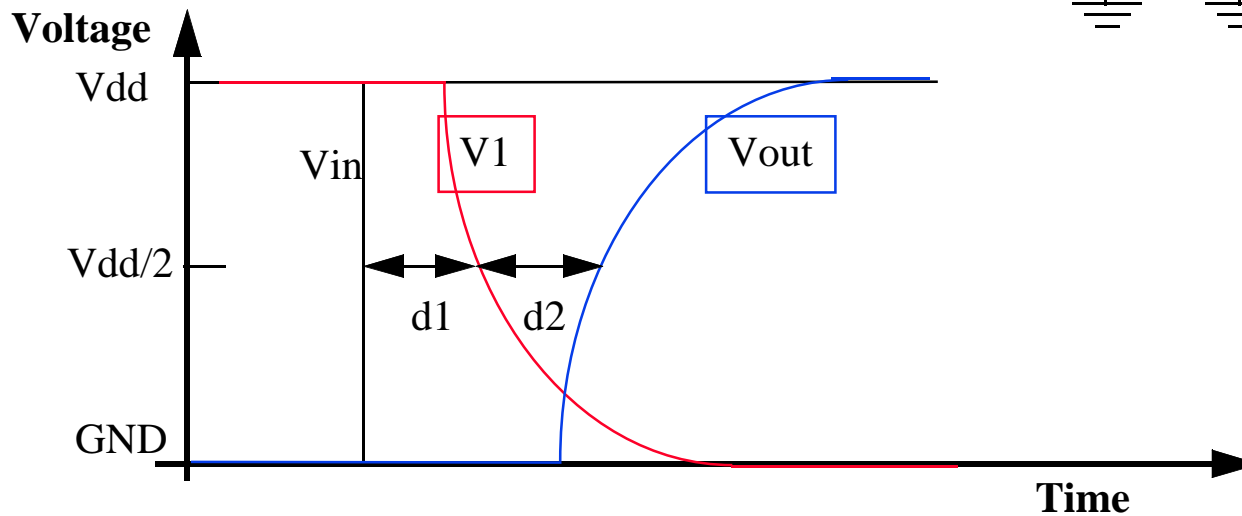
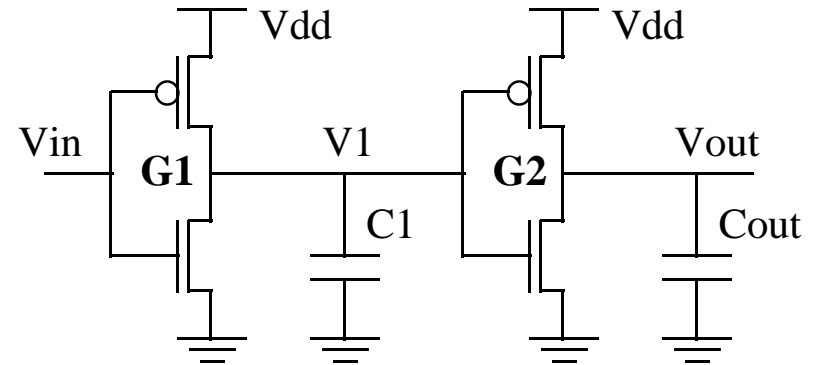
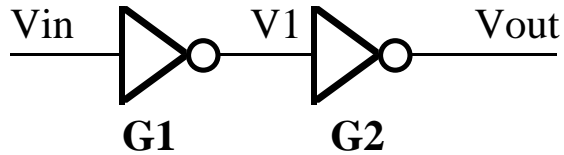


# Fluid Timing Model



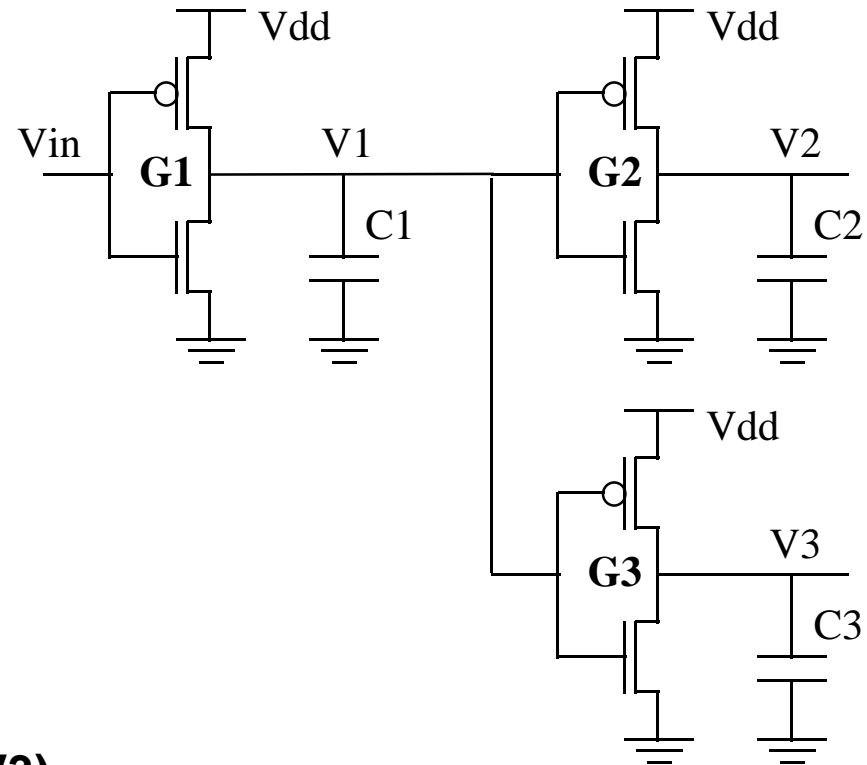
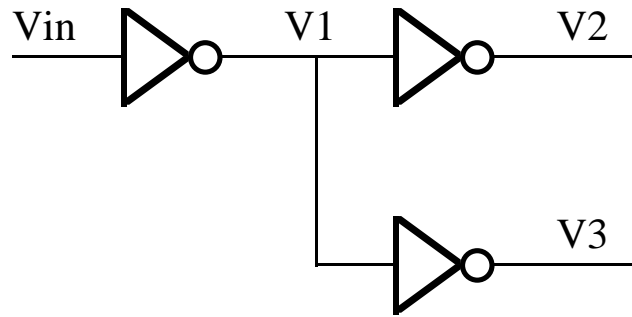
- **Water <-> Electrical Charge      Tank Capacity <-> Capacitance (C)**
- **Water Level <-> Voltage      Water Flow <-> Charge Flowing (Current)**
- **Size of Pipes <-> Strength of Transistors (G)**
- **Time to fill up the tank ~ C / G**

# Series Connection



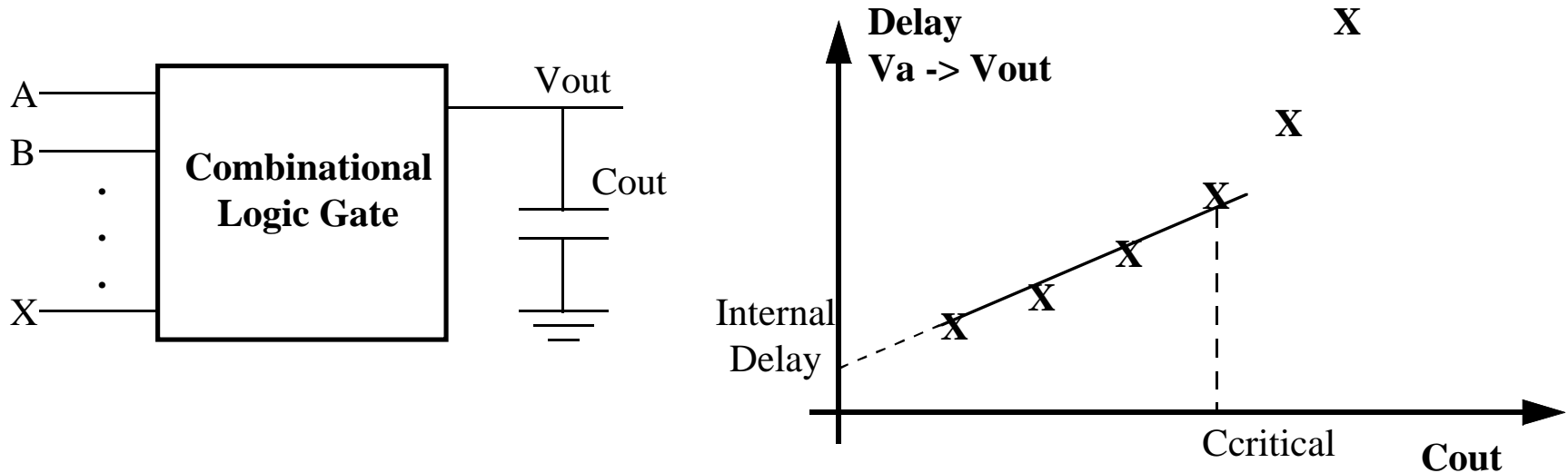
- **Total Propagation Delay = Sum of individual delays =  $d1 + d2$**
- **Capacitance  $C1$  has two components:**
  - **Capacitance of the wire connecting the two gates**
  - **Input capacitance of the second inverter**

# Parallel Connection



- **Delay ( $V_{in} \rightarrow V2$ )  $\neq$  Delay ( $V_{in} \rightarrow V3$ )**
  - **Delay ( $V_{in} \rightarrow V2$ ) = Delay ( $V_{in} \rightarrow V1$ ) + Delay ( $V1 \rightarrow V2$ )**
  - **Delay ( $V_{in} \rightarrow V3$ ) = Delay ( $V_{in} \rightarrow V1$ ) + Delay ( $V1 \rightarrow V3$ )**
- **Critical Path = The longest among the N parallel paths**
- **$C1 = \text{Wire C} + C_{in} \text{ of Gate 2} + C_{in} \text{ of Gate 3}$**

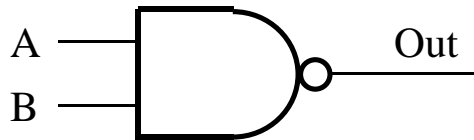
# General Cell Delay Model



- Tied inputs B ... X such that a change in A will causes Vout to change
  - Example: If this is a AND gate, tie Input B, C, ... X to “1”
- Do not use this gate to drive any  $C_{out} > C_{critical}$
- Keep the model simple by using a linear equation:
  - $\text{Delay (A} \rightarrow \text{Out)} = \text{Internal Delay} + (\text{Load Dependent Delay}) \times C_{out}$

# Characterize a Gate

- Input capacitance for each input
- For each input-to-output path:
  - For each output transition type (H->L, L->H, H->Z, L->Z ... etc.)
    - Internal delay (ns)
    - Load dependent delay (ns / fF)
- Example: 2-input NAND Gate

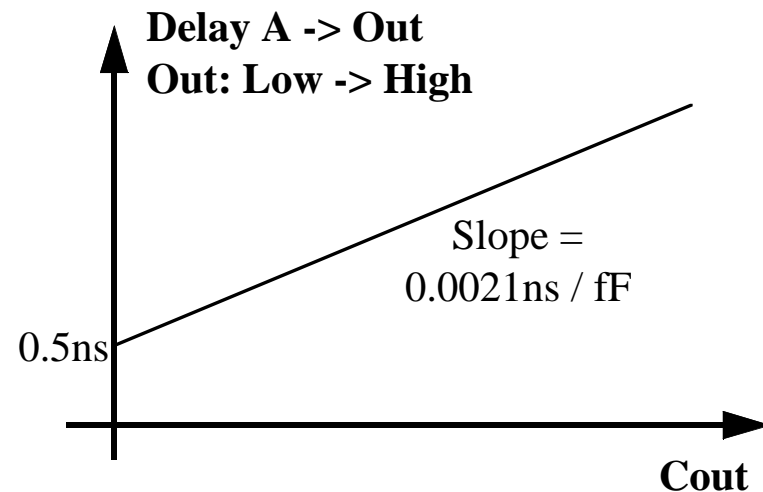


For A and B: Input Load = 61 fF

For either A -> Out or B -> Out:

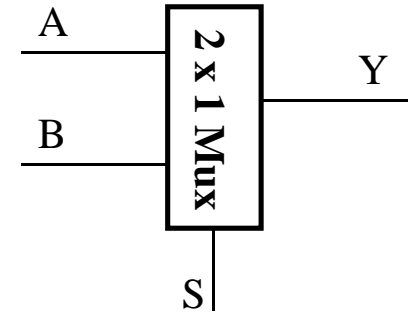
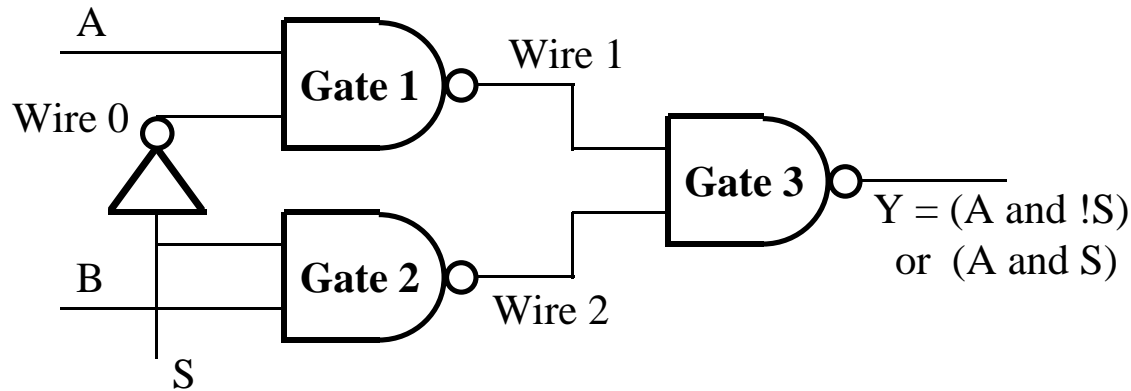
$TP_{lh} = 0.5\text{ns}$     $T_{plhf} = 0.0021\text{ns} / \text{fF}$

$TP_{hl} = 0.1\text{ns}$     $T_{phlf} = 0.0020\text{ns} / \text{fF}$





## A Specific Example: 2 to 1 MUX



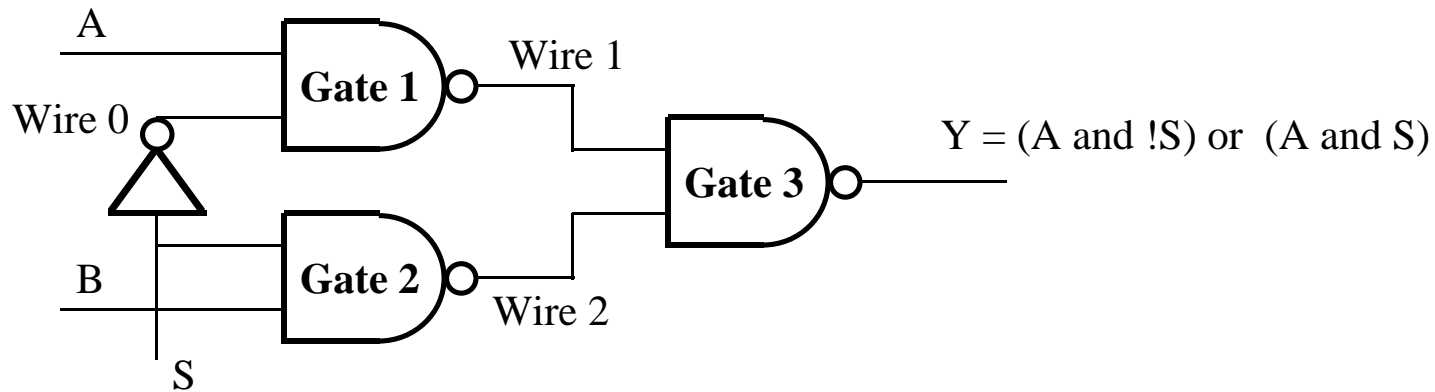
### ◦ Input Load

- A, B: I.L. (NAND) = 61 fF
- S: I.L. (INV) + I.L. (NAND) = 50 fF + 61 fF = 111 fF

### ◦ Load Dependent Delay: Same as Gate 3

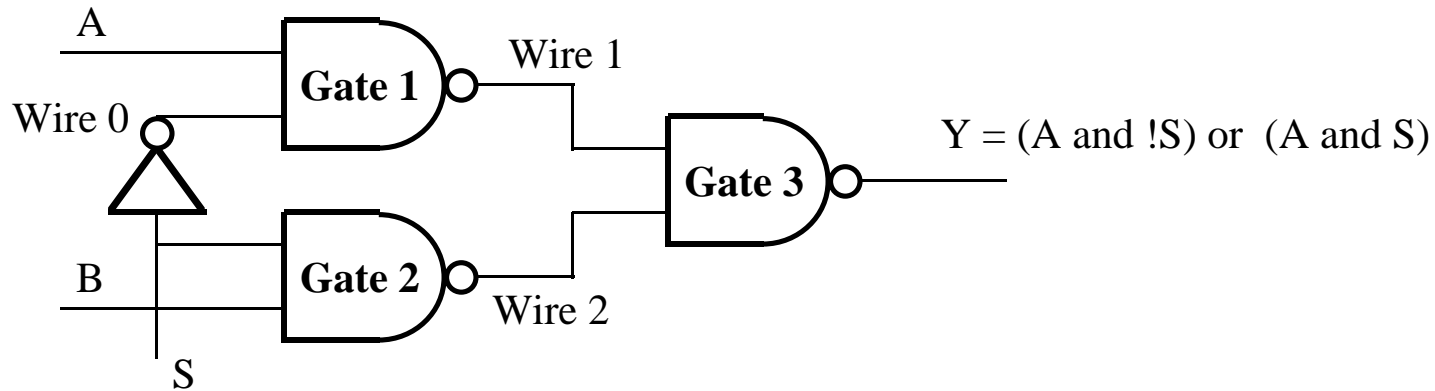
- $T_{AY|hf} = 0.021 \text{ ns / fF}$        $T_{AY|hf} = 0.020 \text{ ns / fF}$
- $T_{BY|hf} = 0.021 \text{ ns / fF}$        $T_{BY|hf} = 0.020 \text{ ns / fF}$
- $T_{SY|hf} = 0.021 \text{ ns / fF}$        $T_{SY|hf} = 0.020 \text{ ns / fF}$

## 2 to 1 MUX: Internal Delay Calculation



- **Internal Delay:**
  - **A to Y:** I.D. G1 + (Wire 1 C + G3 Input C) \* L.D.D G1 + I.D. G3
  - **B to Y:** I.D. G2 + (Wire 2 C + G3 Input C) \* L.D.D. G2 + I.D. G3
  - **S to Y (Worst Case):** I.D. Inv + (Wire 0 C + G1 Input C) \* L.D.D. Inv + Internal Delay A to Y
- **We can approximate the effect of “Wire 1 C” by:**
  - **Assume Wire 1 has the same C as all the gate C attache to it.**
  - **Total C Gate 1 need to drive: 2.0 x Input C of Gate 3**

## 2 to 1 MUX: Internal Delay Calculation (continue)



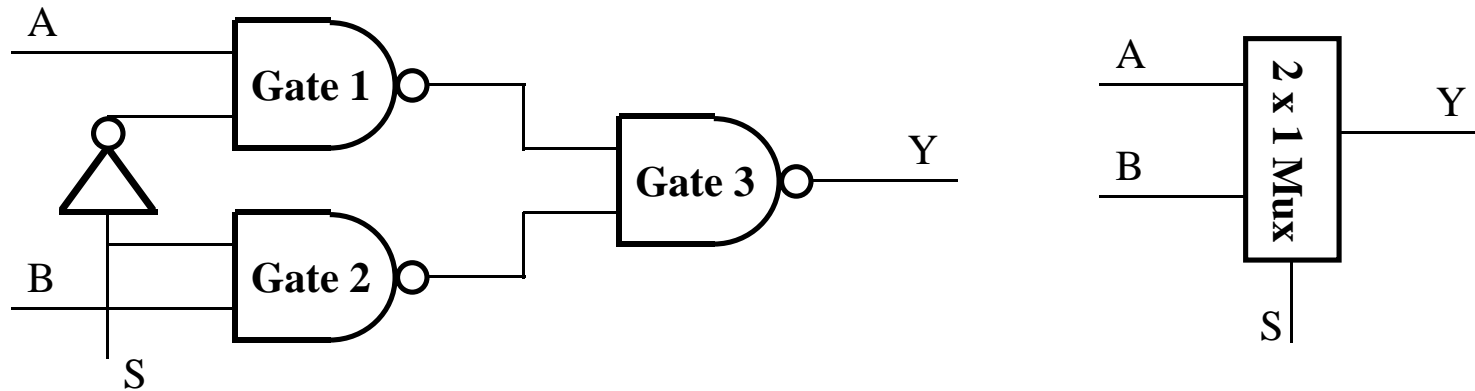
### Internal Delay:

- A to Y: I.D. G1 + (Wire 1 C + G3 Input C) \* L.D.D G1 + I.D. G3
- B to Y: I.D. G2 + (Wire 2 C + G3 Input C) \* L.D.D. G2 + I.D. G3
- S to Y (Worst Case): I.D. Inv + (Wire 0 C + G1 Input C) \* L.D.D. Inv + Internal Delay A to Y

### Specific Example:

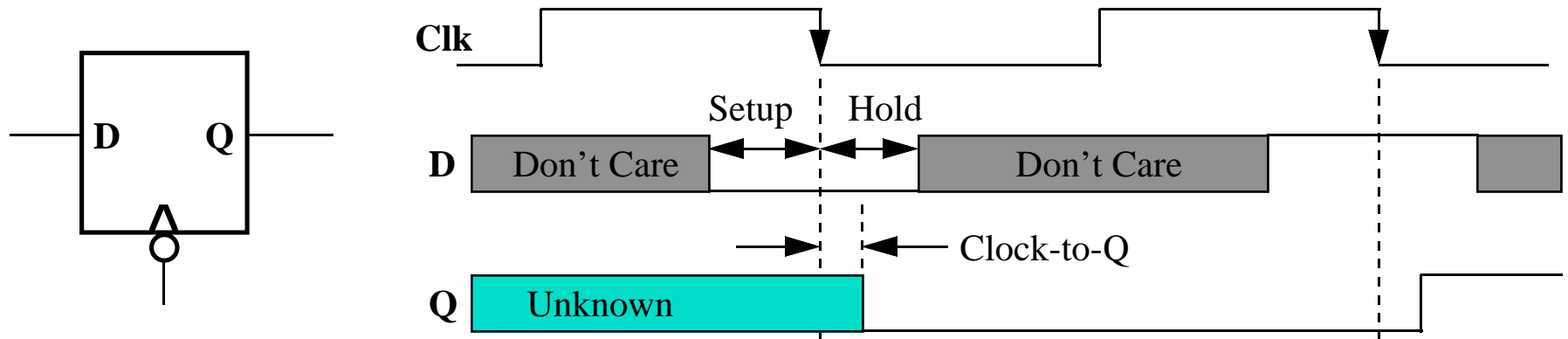
- $T_{AY|H} = T_{PhI} \text{ G1} + (2.0 * 61 \text{ fF}) * T_{PhI} \text{ G1} + T_{PlH} \text{ G3}$   
 $= 0.1\text{ns} + 122 \text{ fF} * 0.0020\text{ns/fF} + 0.5\text{ns} = 0.844\text{ns}$

## Abstraction: 2 to 1 MUX



- **Input Load:**  $A = 61 \text{ fF}$ ,  $B = 61 \text{ fF}$ ,  $S = 111 \text{ fF}$
- **Load Dependent Delay:**
  - $T_{AY|hf} = 0.021 \text{ ns / fF}$        $T_{AYhlf} = 0.020 \text{ ns / fF}$
  - $T_{BY|hf} = 0.021 \text{ ns / fF}$        $T_{BYhlf} = 0.020 \text{ ns / fF}$
  - $T_{SY|hf} = 0.021 \text{ ns / fF}$        $T_{SYhlf} = 0.020 \text{ ns / fF}$
- **Internal Delay:**
  - $T_{AY|h} = T_{Phl \text{ G1}} + (2.0 * 61 \text{ fF}) * T_{Phl \text{ G1}} + T_{Plh \text{ G3}}$   
 $= 0.1\text{ns} + 122 \text{ fF} * 0.0020\text{ns/fF} + 0.5\text{ns} = 0.844\text{ns}$
  - **Fun Exercises:**  $T_{AYhl}$ ,  $T_{BYlh}$ ,  $T_{SYlh}$ ,  $T_{SYlh}$

# Storage Element's Timing Model



- **Setup Time:** Input must be stable **BEFORE** the trigger clock edge
- **Hold Time:** Input must **REMAIN** stable after the trigger clock edge
- **Clock-to-Q time:**
  - Output cannot change instantaneously at the trigger clock edge
  - Similar to delay in logic gates, two components:
    - Internal Clock-to-Q
    - Load dependent Clock-to-Q

# Break (5 Minutes)

# CS152 Logic Elements

- **NAND2, NAND3, NAND 4**
- **NOR2, NOR3, NOR4**
- **INV1x (normal inverter)**
- **INV4x (inverter with big output drive)**

# CS152 Logic Elements (Continue)

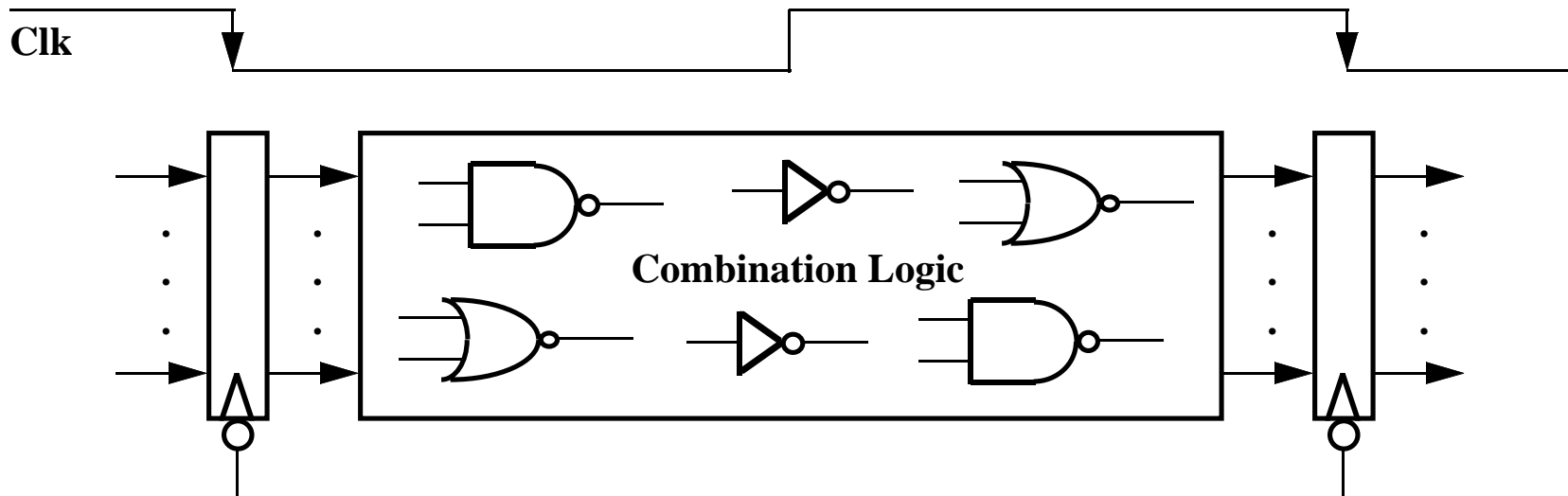
- **XOR2**
- **XNOR2**
- **PWR: Source of 1's**
- **GND: Source of 0's**



# CS152 Storage Element

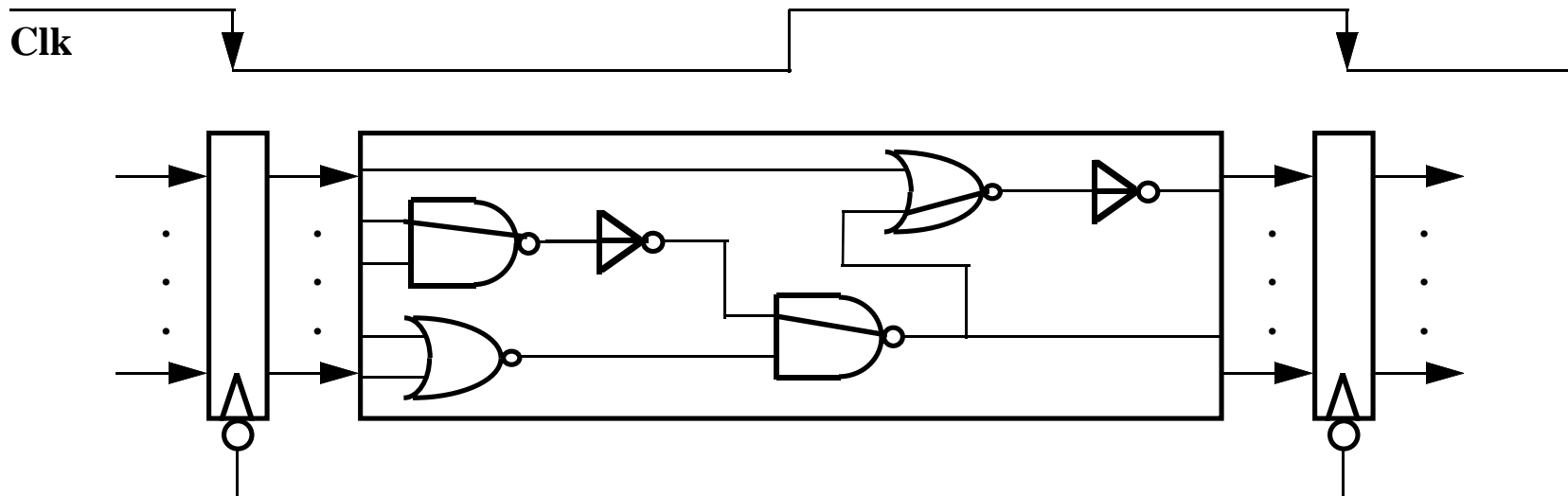
- D flip flop with negative edge triggered

# Clocking Methodology



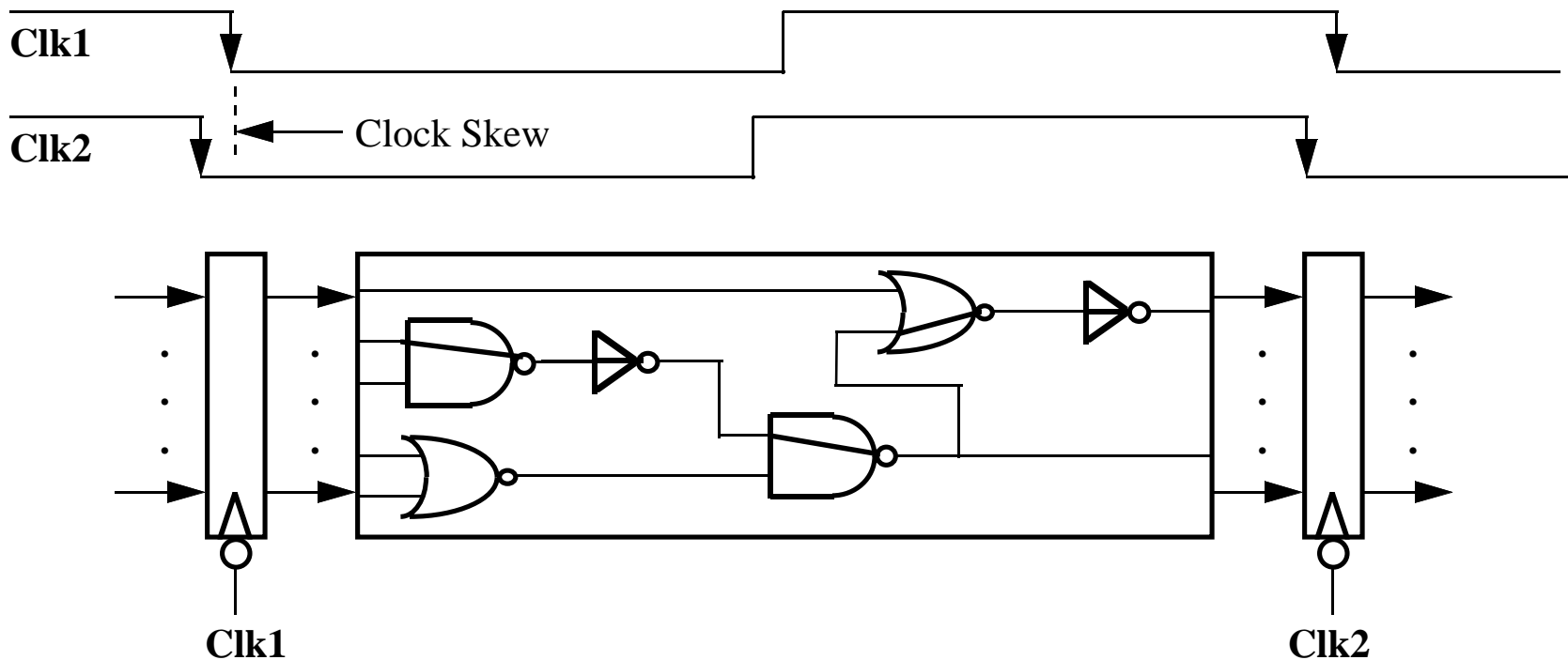
- All storage elements are clocked by the same clock edge
- The combination logic block's:
  - Inputs are updated at each clock tick
  - All outputs **MUST** be stable before the next clock tick

# Critical Path & Cycle Time



- **Critical path: the slowest path between any two storage devices**
- **Cycle time is a function of the critical path**
- **More specifically, the cycle time must be greater than:**
  - **Clock-to-Q + Longest Path through the Combination Logic + Setup**

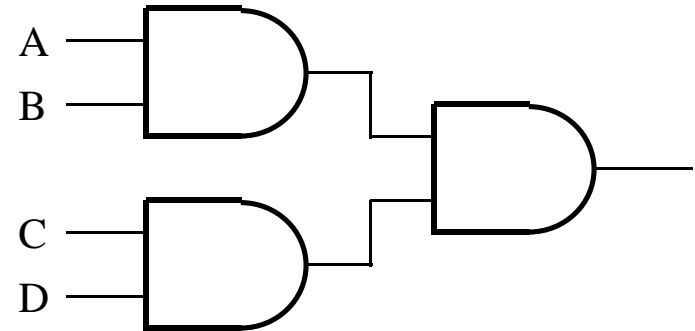
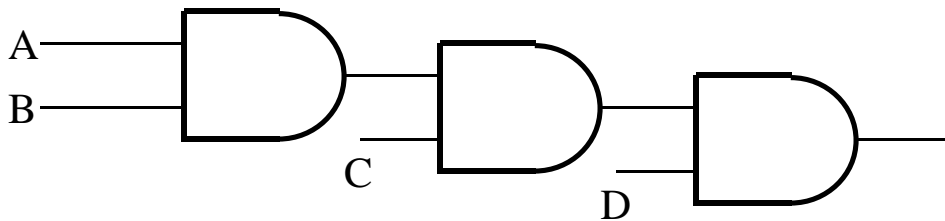
# Clock Skew's Effect on Cycle Time



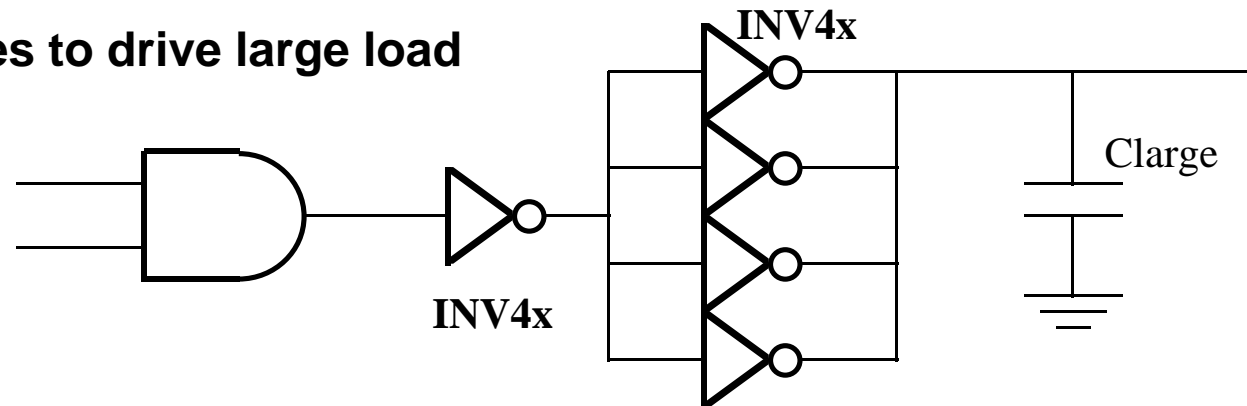
- The worst case scenario for cycle time consideration:
  - The input register sees CLK1
  - The output register sees CLK2
- **Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew**

# Tricks to Reduce Cycle Time

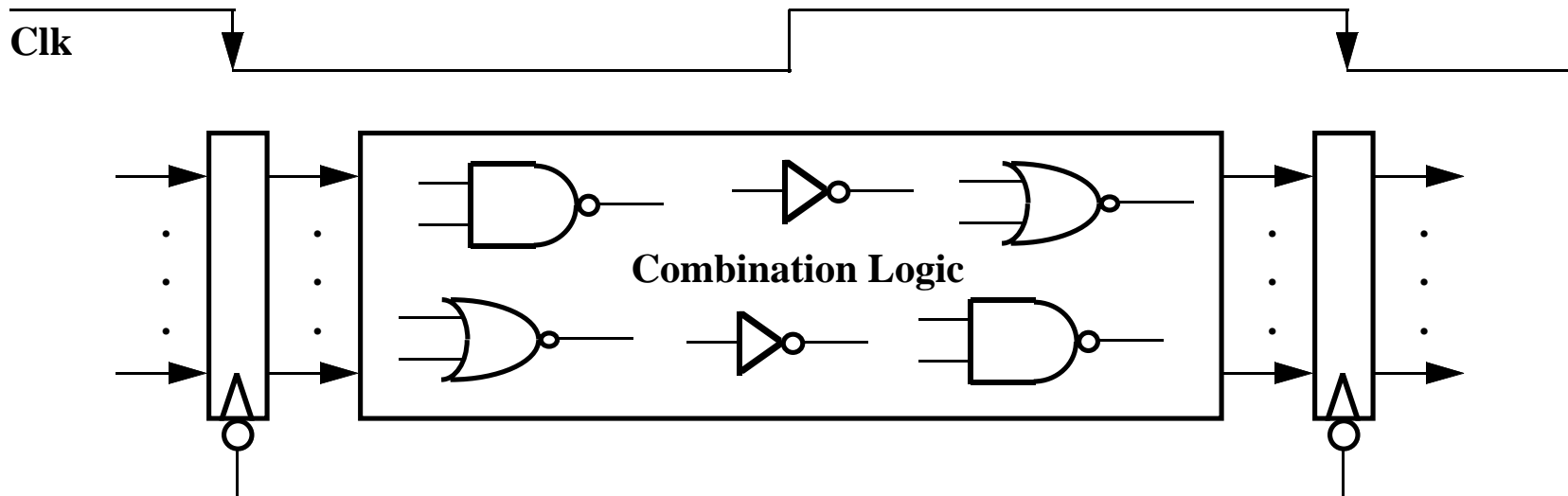
- Reduce the number of gate levels



- Pay attention to loading
  - One gate driving many gates is a bad idea
  - Avoid using a small gate to drive a long wire
- Use multiple stages to drive large load

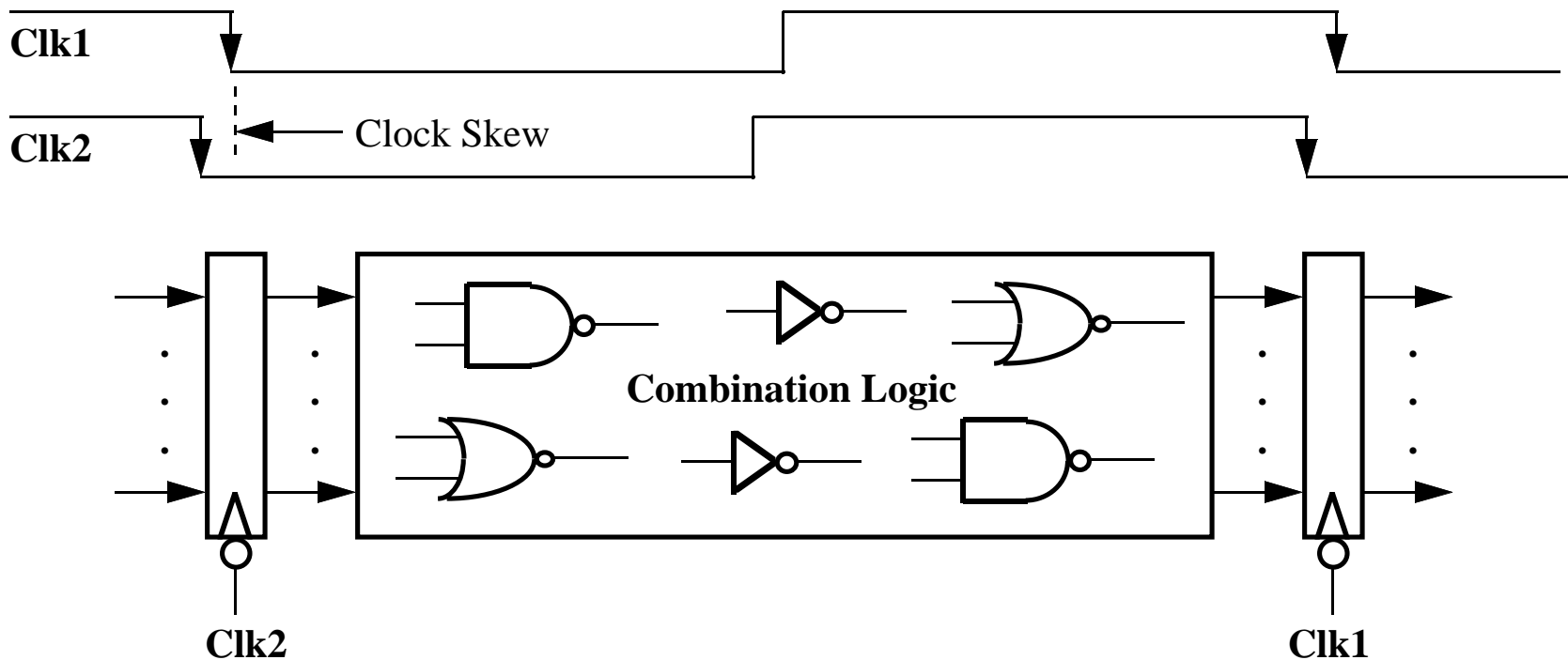


# How to Avoid Hold Time Violation?



- **Hold time requirement:**
  - **Input to register must NOT change immediately after the clock tick**
- **This is usually easy to meet in the “edge trigger” clocking scheme**
- **CLK-to-Q + Shortest Delay Path must be greater than Hold Time**

# Clock Skew's Effect on Hold Time



- The worst case scenario for hold time consideration:
  - The input register sees CLK2
  - The output register sees CLK1
- $(\text{CLK-to-Q} + \text{Shortest Delay Path} - \text{Clock Skew}) > \text{Hold Time}$

# Summary

- **Performance and Technology Trends**
  - **Keep the design simple to take advantage of the latest technology**
  - **CMOS inverter and CMOS logic gates**
- **Delay Modeling and Gate Characterization**
  - **Delay = Internal Delay + (Load Dependent Delay x Output Load)**
- **Clocking Methodology and Timing Considerations**
  - **Simplest clocking methodology**
    - **All storage elements use the SAME clock edge**
  - **Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew**
  - **CLK-to-Q + Shortest Delay Path - Clock Skew < Hold Time**



# To Get More Information

- **A Classic Book that Started it All:**
  - **Carver Mead and Lynn Conway, “Introduction to VLSI Systems,” Addison-Wesley Publishing Company, October 1980.**
- **A Good VLSI Circuit Design Book**
  - **Lance Glasser & Daniel Dobberpuhl, “The Design and Analysis of VLSI Circuits,” Addison-Wesley Publishing Company, 1985.**
    - **Mr. Dobberpuhl is responsible for the DEC Alpha chip design.**
- **A Book by Dean Hodges:**
  - **David Hodges & Horace Jackson, “Analysis and Design of Digital Integrated Circuits,” McGraw-Hill Book Company, 1983.**

# **Computer Architecture and Engineering**

## **Lecture 6: The Design Process & ALU Design**

February 3, 1995

Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 design.1

©DAP & SIK 1995

### **Recap of Last Lecture**

- **Performance and Technology Trends**
  - Keep the design simple to take advantage of the latest technology
  - CMOS inverter and CMOS logic gates
- **Delay Modeling and Gate Characterization**
  - $\text{Delay} = \text{Internal Delay} + (\text{Load Dependent Delay} \times \text{Output Load})$
- **Clocking Methodology and Timing Considerations**
  - Simplest clocking methodology
    - All storage elements use the **SAME** clock edge
  - $\text{Cycle Time} = \text{CLK-to-Q} + \text{Longest Delay Path} + \text{Setup} + \text{Clock Skew}$
  - $\text{CLK-to-Q} + \text{Shortest Delay Path} - \text{Clock Skew} < \text{Hold Time}$

cs 152 design.2

©DAP & SIK 1995

## Outline of Today's Lecture

- Recap of Last Lecture and Introduction of Today's Lecture (4 min.)
- An Overview of the Design Process (16 min.)
- Questions and Administrative Matters (5 min.)
- An Review of Binary Arithmetics (5 min.)
- Designing a Simple 4-bit ALU (20 min.)
- Questions and Break (5 min.)
- Other ALU Construction Techniques (5 min.)
- Keeping an On-line Design Notebook (20 min.)

## The Design Process

### *"To Design Is To Represent"*

Design activity yields description/representation of an object

- Traditional craftsman does not distinguish between the conceptualization and the artifact
- Separation comes about because of complexity
- The concept is captured in one or more *representation languages*
- This process IS design

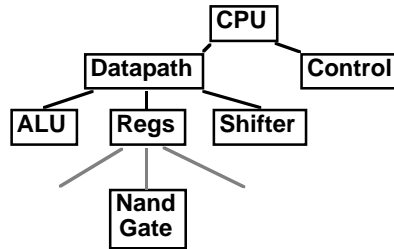
### *Design Begins With Requirements*

- Functional Capabilities: what it will do
- Performance Characteristics: Speed, Power, Area, Cost, . . .

## Design Process (cont.)

### *Design Finishes As Assembly*

- Design understood in terms of components and how they have been assembled
- Top Down *decomposition* of complex functions (behaviors) into more primitive functions
- bottom-up *composition* of primitive building blocks into more complex assemblies



*Design is a "creative process," not a simple method*

cs 152 design.5

©DAP & SIK 1995

## Design Refinement

Informal System Requirement

Initial Specification

Intermediate Specification

Final Architectural Description

Intermediate Specification of Implementation

Final Internal Specification

Physical Implementation

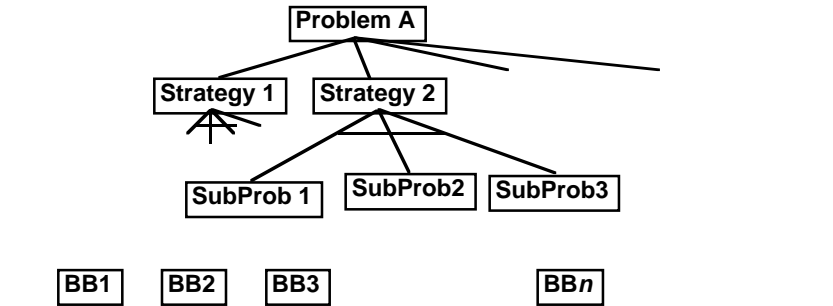
refinement  
increasing level of detail



cs 152 design.6

©DAP & SIK 1995

## Design as Search



*Design involves educated guesses and verification*

- Given the goals, how should these be prioritized?
- Given alternative design pieces, which should be selected?
- Given design space of components & assemblies, which part will yield the best solution?

Feasible (good) choices vs. Optimal choices

cs 152 design.7

©DAP & SIK 1995

## Design as Representation (example)

### (1) Functional Specification

"VHDL Behavior"

Inputs: 2 x 16 bit operands- A, B; 1 bit carry input- Cin.

Outputs: 1 x 16 bit result- S; 1 bit carry output- Co.

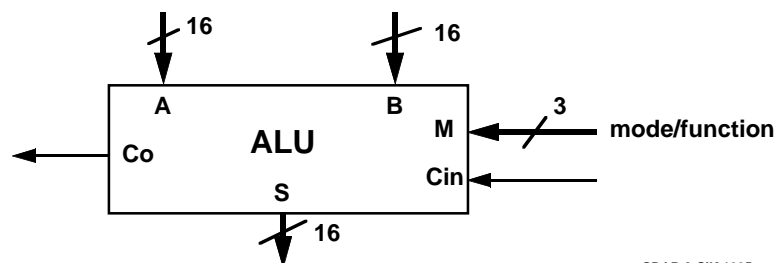
Operations: PASS, ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, XOR, OR, COMPARE (equality)

Performance: left unspecified for now!

### (2) Block Diagram

"VHDL Entity"

Understand the data and control flows



cs 152 design.8

©DAP & SIK 1995

## Elements of the Design Process

- **Divide and Conquer**
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)
- **Generate and Test**
  - Given a collection of building blocks, look for ways of putting them together that meets requirement
- **Successive Refinement**
  - Solve "most" of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.
- **Formulate High-Level Alternatives**
  - Articulate many strategies to "keep in mind" while pursuing any one approach.
- **Work on the Things you Know How to Do**
  - The unknown will become "obvious" as you make progress.

cs 152 design.9

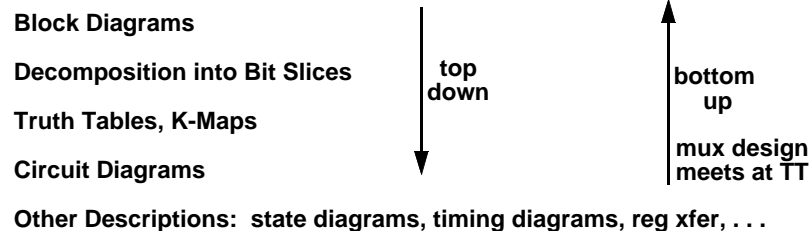
©DAP & SIK 1995

## Summary of the Design Process

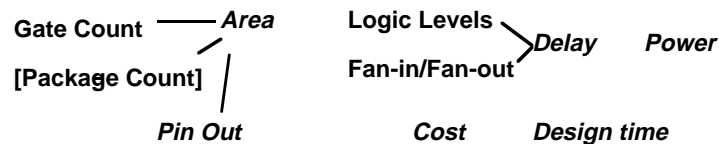
Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

Importance of Design Representations:



Optimization Criteria:



cs 152 design.10

©DAP & SIK 1995

## Administrative Matters

- **A new “tentative” schedule**
  - **Good News:** You will have more time to work on the project after the 2nd mid-term
  - **Bad News:** The 2nd mid-term is moved up one week.
- **After the 2nd mid-terms, we will try to arrange some guest lecturers to talk about some topics that are interesting**

# Introduction to Binary Numbers

- **Consider a 4-bit binary number**

	Decimal	Binary	Decimal	Binary
○	0	0000	4	0100
○	1	0001	5	0101
○	2	0010	6	0110
○	3	0011	7	0111

- **Examples:**

- $3 + 2 = 5$

$$\begin{array}{rcccc} & & 1 & & \\ & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 0 & 1 \end{array}$$

**3 + 3 = 6**

$$\begin{array}{cccc}
 & & 1 & 1 \\
 & 0 & 0 & 1 & 1 \\
 + & 0 & 0 & 1 & 1 \\
 \hline
 & 0 & 1 & 1 & 0
 \end{array}$$

## Two's Complement Representation

- 2's complement representation of negative numbers
  - Bitwise inverse and add 1
  - The MSB is always "1" for negative number => sign bit
- Biggest 4-bit Binary Number: 7      Smallest 4-bit Binary Number: -8

Decimal	Binary	Decimal	Bitwise Inverse	2's Complement
0	0000	0	1111	0000
1	0001	-1	1110	1111
2	0010	-2	1101	1110
3	0011	-3	1100	1101
4	0100	-4	1011	1100
5	0101	-5	1010	1011
6	0110	-6	1001	1010
7	0111	-7	1000	1001
8	1000	-8	0111	1000

“Illegal” Positive Number!

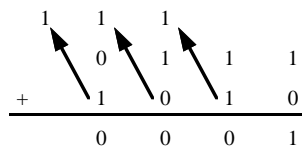
cs 152 design.13

©DAP & SIK 1995

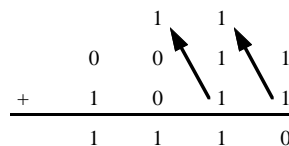
## Two's Complement Arithmetic

Decimal	Binary	Decimal	2's Complement
0	0000	0	0000
1	0001	-1	1111
2	0010	<u>-2</u>	<u>1110</u>
3	0011	-3	1101
4	0100	-4	1100
5	0101	<u>-5</u>	<u>1011</u>
6	0110	<u>-6</u>	<u>1010</u>
7	0111	-7	1001
		-8	1000

◦ Examples:  $7 - 6 = 7 + (-6) = 1$



$3 - 5 = 3 + (-5) = -2$

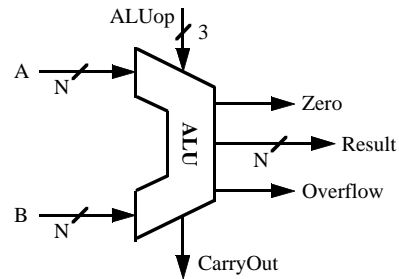


cs 152 design.14

©DAP & SIK 1995



## Functional Specification of the ALU



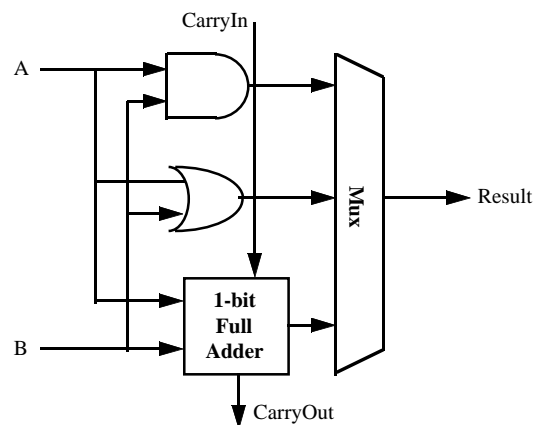
ALU Control Lines (ALUop)	Function
• 000	And
• 001	Or
• 010	Add
• 110	Subtract
• 111	Set-on-less-than

cs 152 design.15

©DAP & SIK 1995

## A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD

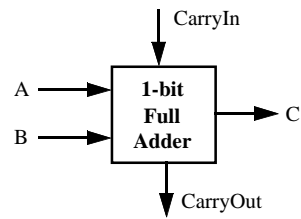


cs 152 design.16

©DAP & SIK 1995

## A One-bit Full Adder

- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:



Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

cs 152 design.17

©DAP & SIK 1995

## Logic Equation for CarryOut

Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

- $\text{CarryOut} = (!A \& B \& \text{CarryIn}) \mid (A \& !B \& \text{CarryIn}) \mid (A \& B \& !\text{CarryIn}) \mid (A \& B \& \text{CarryIn})$
- $\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$

cs 152 design.18

©DAP & SIK 1995

## Logic Equation for Sum

Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

$$\begin{aligned} \text{Sum} = & (!A \& !B \& \text{CarryIn}) \mid (!A \& B \& !\text{CarryIn}) \mid (A \& !B \& !\text{CarryIn}) \\ & \mid (A \& B \& \text{CarryIn}) \end{aligned}$$

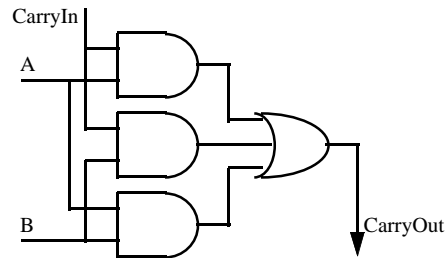
## Logic Equation for Sum (continue)

- Sum =  $(!A \& !B \& \text{CarryIn}) \mid (!A \& B \& !\text{CarryIn}) \mid (A \& !B \& !\text{CarryIn}) \mid (A \& B \& \text{CarryIn})$
- Sum = A XOR B XOR CarryIn
- Truth Table for XOR:

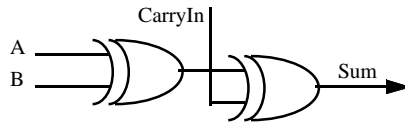
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

## Logic Diagrams for CarryOut and Sum

◦  $\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$



◦  $\text{Sum} = A \text{ XOR } B \text{ XOR } \text{CarryIn}$

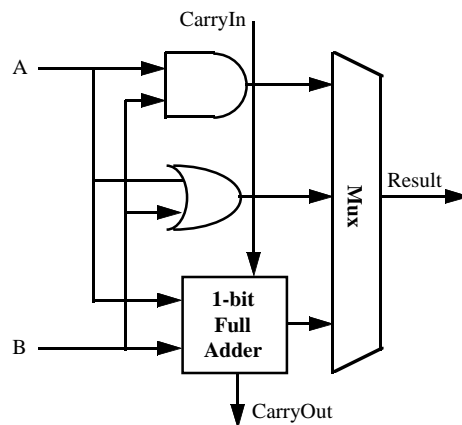


cs 152 design.21

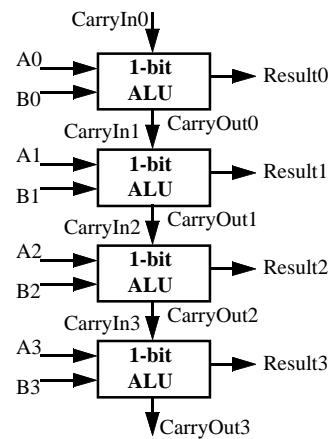
©DAP & SIK 1995

## A 4-bit ALU

◦ **1-bit ALU**



**4-bit ALU**

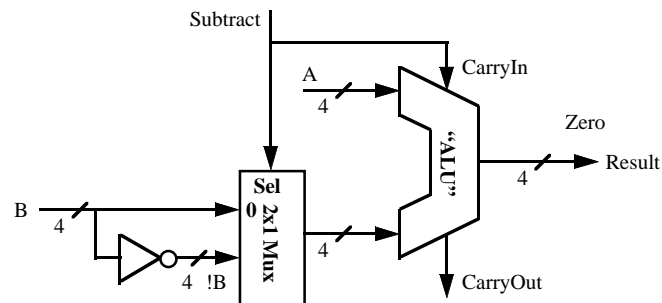


cs 152 design.22

©DAP & SIK 1995

## How About Subtraction?

- Keep in mind the followings:
  - $(A - B)$  is the that as:  $A + (-B)$
  - 2's Complement: Take the inverse of every bit and add 1
- Bit-wise inverse of B is !B:
  - $A + !B + 1 = A + (!B + 1) = A + (-B) = A - B$



cs 152 design.23

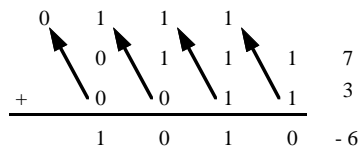
©DAP & SIK 1995

## Overflow

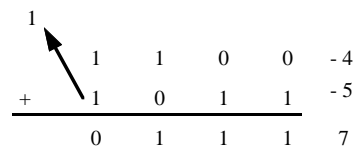
Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

- Examples:  $7 + 3 = 10$  but ...



- $-4 - 5 = -9$  but ...

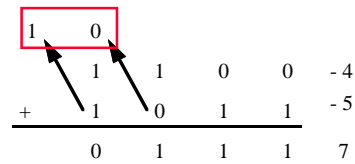
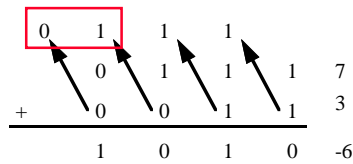


cs 152 design.24

©DAP & SIK 1995

## Overflow Detection

- **Overflow:** the result is too large (or too small) to represent properly
  - Example:  $-8 < \leq$  4-bit binary number  $\leq 7$
- When adding operands with different signs, overflow cannot occur!
- **Overflow occurs when adding:**
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- **Homework exercise:** Prove you can detect overflow by:
  - Carry into MSB  $\neq$  Carry out of MSB

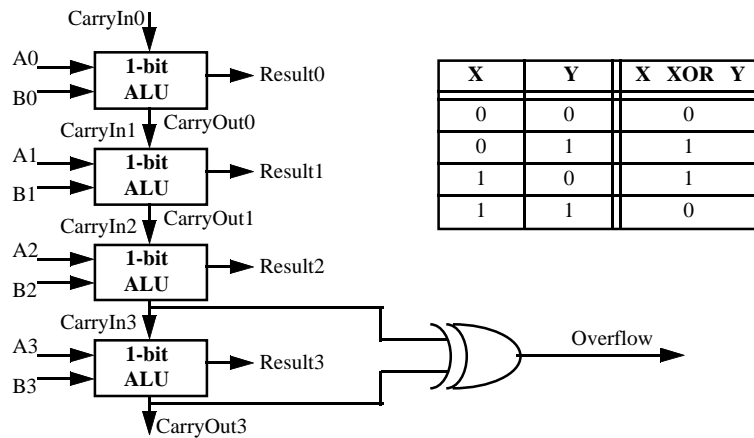


cs 152 design.25

©DAP & SIK 1995

## Overflow Detection Logic

- **Carry into MSB  $\neq$  Carry out of MSB**
  - For a N-bit ALU:  $\text{Overflow} = \text{CarryIn}[N - 1] \text{ XOR } \text{CarryOut}[N - 1]$

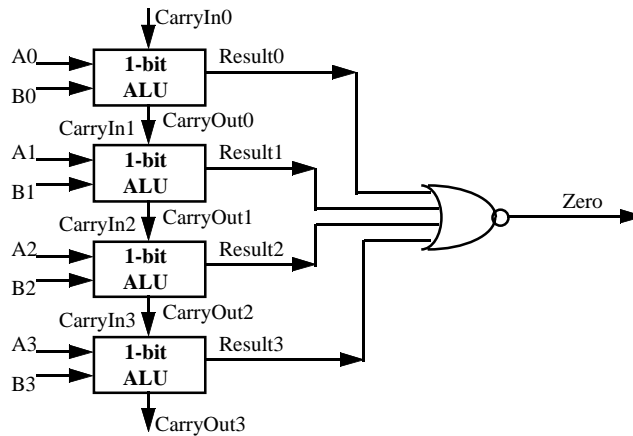


cs 152 design.26

©DAP & SIK 1995

## Zero Detection Logic

- Zero Detection Logic is just a one BIG NOT gate
  - Any non-zero input to the NOR gate will cause its output to be zero

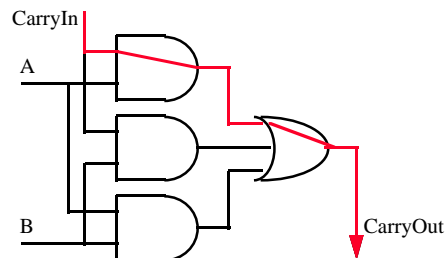
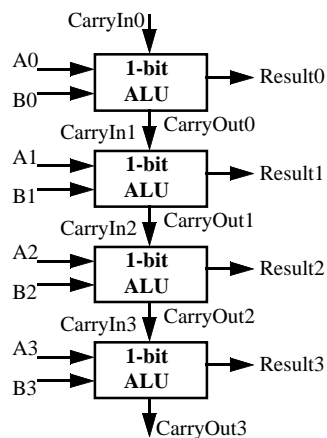


cs 152 design.27

©DAP & SIK 1995

## The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay



cs 152 design.28

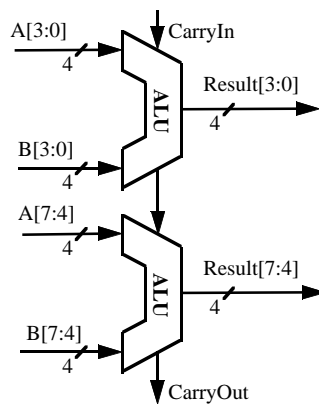
©DAP & SIK 1995

## Break

- 5-minute Break

## Carry Select Header

- Consider building a 8-bit ALU
  - Simple: connects two 4-bit ALUs in series

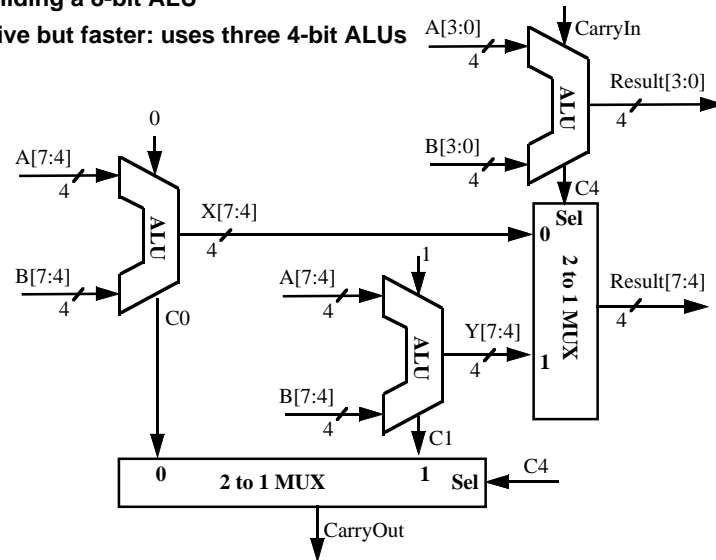




## Carry Select Header (Continue)

### Consider building a 8-bit ALU

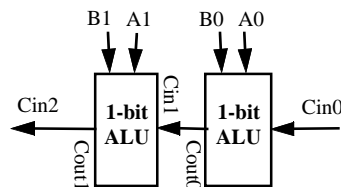
- Expensive but faster: uses three 4-bit ALUs



cs 152 design.31

©DAP & SIK 1995

## The Theory Behind Carry Lookahead



### Recalled: CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)

- $Cin2 = Cout1 = (B1 \& Cin1) | (A1 \& Cin1) | (A1 \& B1)$
- $Cin1 = Cout0 = (B0 \& Cin0) | (A0 \& Cin0) | (A0 \& B0)$

### Substituting Cin1 into Cin2:

- $Cin2 = (A1 \& A0 \& B0) | (A1 \& A0 \& Cin0) | (A1 \& B0 \& Cin0) | (B1 \& A0 \& B0) | (B1 \& A0 \& Cin0) | (B1 \& A0 \& Cin0) | (A1 \& B1)$

### Now define two new terms:

- Generate Carry at Bit i  $gi = Ai \& Bi$
- Propagate Carry via Bit i

cs 152 design.32

©DAP & SIK 1995

## The Theory Behind Carry Lookahead (Continue)

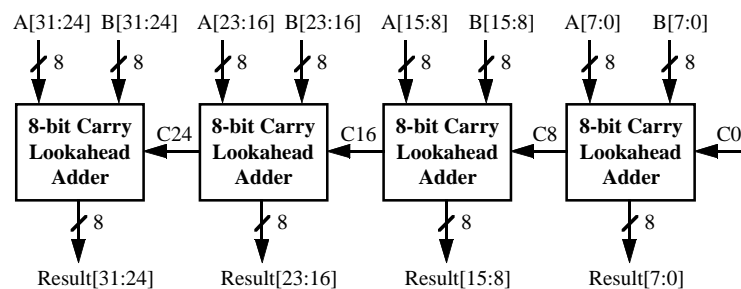
- Using the two new terms we just defined:
  - Generate Carry at Bit  $i$       $g_i = A_i \& B_i$
  - Propagate Carry via Bit  $i$     $p_i = A_i \text{ or } B_i$
- We can rewrite:
  - $Cin1 = g_0 \mid (p_0 \& Cin0)$
  - $Cin2 = g_1 \mid (p_1 \& g_0) \mid (p_1 \& p_0 \& Cin0)$
  - $Cin3 = g_2 \mid (p_2 \& g_1) \mid (p_2 \& p_1 \& g_0) \mid (p_2 \& p_1 \& p_0 \& Cin0)$
- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 ( $g_2$ )
  - Or we generate a carry at bit 1 ( $g_1$ ) and bit 2 allows it to propagate ( $p_2 \& g_1$ )
  - Or we generate a carry at bit 0 ( $g_0$ ) and bit 1 as well as bit 2 allows it to propagate ( $p_2 \& p_1 \& g_0$ )
  - Or we have a carry input at bit 0 ( $Cin0$ ) and bit 0, 1, and 2 all allow it to propagate ( $p_2 \& p_1 \& p_0 \& Cin0$ )

cs 152 design.33

©DAP & SIK 1995

## A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
  - Just imagine the length of the equation for  $Cin_{31}$
- Common practices:
  - Connects several N-bit Lookahead Adders to form a big adder
  - Example: connects four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder



cs 152 design.34

©DAP & SIK 1995

## **Why should you keep an design notebook?**

- **Keep track of the design decisions and the reasons behind them**
  - **Otherwise, it will be hard to debug and/or refine the design**
- **Insights you have on certain aspect of the design**
- **Results of the different design & debug experiments**

## **Why do we keep it on-line?**

- **You need to force yourself to take notes**
  - **Open an widow and leave an editor running while you work**
- **Take advantage of the window system's "cut and paste" features**
- **It is much easier to read your typing than your writing**

## How should you do it?

- **Keep it simple**
- **Separate the entries by dates**
- **Index: write a one-line summary of what you did each day**

## On-line Notebook Example

- **Refer to the handout**

## Summary

- **An Overview of the Design Process**
  - Design is an iterative process-- successive refinement
  - Do NOT wait until you know everything before you start
- **An Introduction to Binary Arithmetics**
  - If you use 2's complement representation, subtract is easy.
- **ALU Design**
  - Designing a Simple 4-bit ALU
  - Other ALU Construction Techniques
- **On-line Design Notebook**
  - Open a window and keep an editor running while you work
  - Refer to the handout as an example

## To Get More Information

- **Chapter 4 of your text book:**
  - David Patterson & John Hennessy, "Computer Organization & Design," Morgan Kaufmann Publishers, 1994.
- **A book I really like:**
  - David Winkel & Franklin Prosser, "The Art of Digital Design: An Introduction to Top-Down Design," Prentice-Hall, Inc., 1980.
- **My master thesis has a chapter on carry lookahead adder design:**
  - Shing Kong, "Some Design Techniques for High Performance MOS Circuits," Master Report, EECS Department, UC Berkeley, 1985.

# Computer Architecture and Engineering

## Lecture 7: ALU Design

February 8, 1995

Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

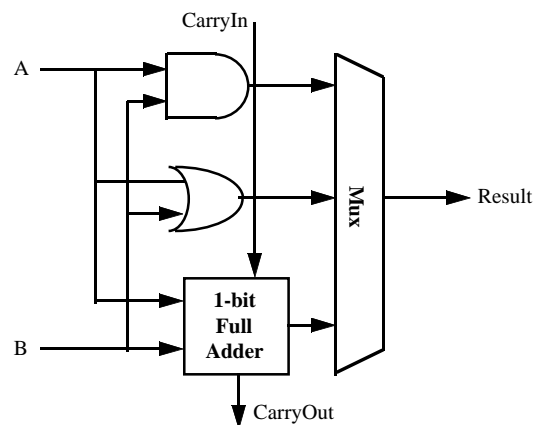
Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 I7 ALU.1

©DAP & SIK 1995

### Review: A One Bit ALU

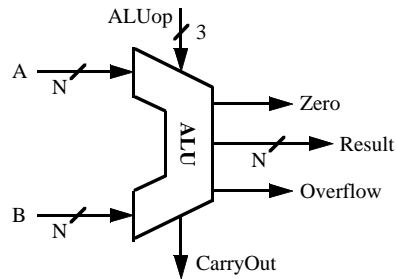
- This 1-bit ALU will perform AND, OR, and ADD



cs 152 I7 ALU.2

©DAP & SIK 1995

## Review: Functional Specification of the ALU



ALU Control Lines (ALUop)	Function
• 000	And
• 001	Or
• 010	Add
• 110	Subtract
• 111	Set-on-less-than

## Recap of Last Lecture

- **An Overview of the Design Process**
  - Design is an iterative process-- successive refinement
  - Do NOT wait until you know everything before you start
- **An Introduction to Binary Arithmetics**
  - If you use 2's complement representation, subtract is easy.
- **ALU Design**
  - Designing a Simple 4-bit ALU
  - Other ALU Construction Techniques
- **On-line Design Notebook**
  - Open a window and keep an editor running while you work
  - Refer to the handout as an example

## Outline of Today's Lecture

- **Recap of Last Lecture and Introduction of Today's Lecture (2 min.)**
- **Deriving the ALU from the Instruction Set & Shift (25 min.)**
- **Questions and Administrative Matters (3 min.)**
- **Multiply (20 min.)**
- **Questions and Break (5 min.)**
- **More Multiply (25 min.)**

## Deriving requirements of ALU

- **Start with instruction set architecture: must be able to do all operations in ISA**
- **Tradeoffs of cost and speed based on frequency of occurrence, hardware budget**
- **MIPS ISA**



## MIPS arithmetic instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comments</u>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; exception possible
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; exception possible
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \$3$	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \$3$	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \$3$ , Hi = $\$2 \bmod \$3$	Lo = quotient, Hi = remainder
divide unsigned remainder	divu \$2,\$3	Lo = $\$2 \div \$3$ , Hi = $\$2 \bmod \$3$	Unsigned quotient &
Move from Hi	mfhi \$1	$\$1 = \text{Hi}$	Used to get copy of Hi
Move from Lo	mflo \$1	$\$1 = \text{Lo}$	Used to get copy of Lo

cs 152 I7 ALU.7

©DAP & SIK 1995

## MIPS logical instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comment</u>
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	$\$1 = \$2 \mid \$3$	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	$\$1 = \$2 \oplus \$3$	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	$\$1 = \sim(\$2 \mid \$3)$	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	$\$1 = \$2 \& 10$	Logical AND reg, constant
or immediate	ori \$1,\$2,10	$\$1 = \$2 \mid 10$	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	$\$1 = \sim\$2 \& \sim 10$	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant
shift right arithm.	sra \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	$\$1 = \$2 \ll \$3$	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right arith. by variable

cs 152 I7 ALU.8

©DAP & SIK 1995

## Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset    if  $R[rs] == R[rt]$  then PC-relative branch
  - BNE rs, rt, offset     $<>$
- Compare to zero and Branch
  - BLEZ rs, offset    if  $R[rs] \leq 0$  then PC-relative branch
  - BGTZ rs, offset     $>$
  - BLT     $<$
  - BGEZ     $\geq$
  - BLTZAL rs, offset    if  $R[rs] < 0$  then branch and link (into R 31)
  - BGEZAL     $\geq$

## MIPS ALU requirements

- Add, AddU, Sub, SubU, Addl, AddIU  
=> 2's complement adder with overflow detection & inverter
- SLTI, SLTIU (set less than)  
=> 2's complement adder with inverter, check sign bit of result
- BEQ, BNE (branch on equal or not equal)  
=> 2's complement adder with inverter, check if result = 0
- And, Or, Andl, Orl  
=> Logical AND, logical OR
- ALU from last lecture supports these ops

## Additional MIPS ALU requirements

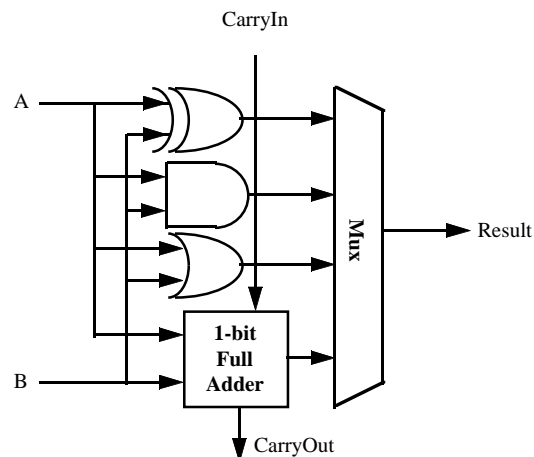
- Xor, Nor, Xorl  
=> Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- Sll, Srl, Sra  
=> Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- Mult, MultU, Div, DivU  
=> Need 32-bit multiply and divide, signed and unsigned

cs 152 I7 ALU.11

©DAP & SIK 1995

## Add XOR to ALU

- Expand Multiplexor



cs 152 I7 ALU.12

©DAP & SIK 1995

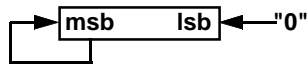
## Shifters

Three different kinds:

*logical*-- value shifted in is always "0"



*arithmetic*-- on right shifts, sign extend



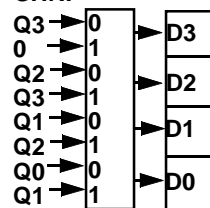
*rotating*-- shifted out bits are wrapped around (not in MIPS)



Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

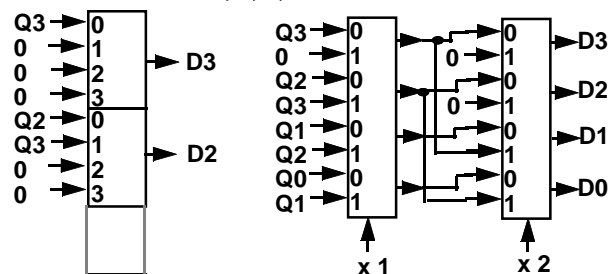
## Multiplexor/Shifter

**SHR:**



SHR/  
don't  
shift  
( 5 inputs)

**SHR 0, 1, 2, 3 bits:**



8 x 2:1 Mux  
2 stages

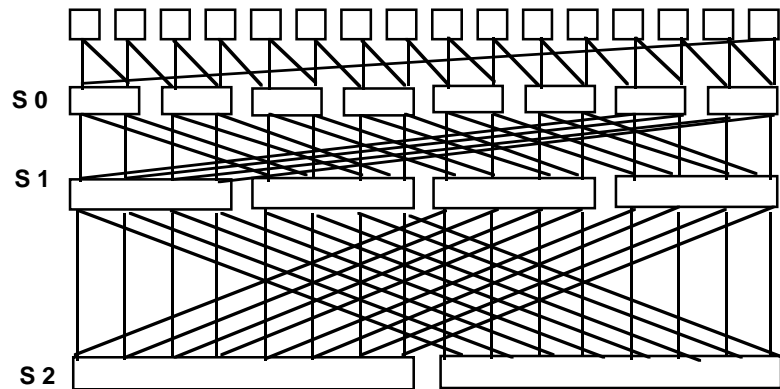
shift amount  
(0,1,2,3)

4 x 4:1 Mux  
1 stage

( 7 inputs)

How do arithmetic shift right?

### General Scheme



Right-to-left connections support Rotate (not in MIPS but found in others)

### 32 Bit Shifter

Using this scheme for 32 bit data with 0-31 bit shifts would result in

5 stages of mux's (x1, x2, x4, x8, x16) if 2:1 are used

$$32 \text{ bits} \times 5 \text{ stages} = 160 \text{ 2:1 mux's!}$$

3 stages of mux's (x4, x4, x2) if 2 levels of 4:1 and 1 level of 2:1 used

$$32 \times 4:1 + 32 \times 4:1 + 32 \times 2:1$$

2 stages of mux's (x8, x4) if 1 level of 8:1 and 1 level of 4:1 used

$$32 \times 8:1 + 32 \times 4:1$$

### Multibit Shifts (continued)

Mixed strategy, multiple control loops with more than one bit per loop

31 bit shift:

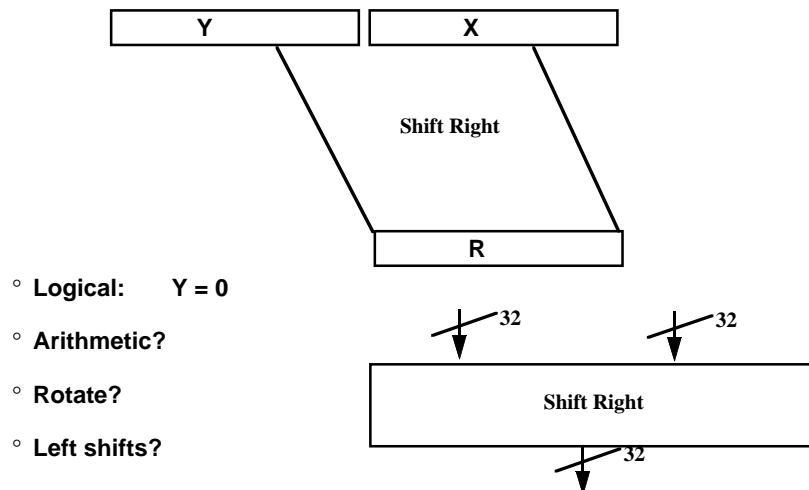
- 31 iterations with a 0,1 position shifter
- 11 iterations with a 0,1,2,3 position shifter
- 5 iterations with a 0,1,2,3,4,5,6,7 position shifter
- 3 iterations with an 0-15 position shifter

Fortunately, most shifts are relatively short (0-3 often implemented)

*Extra Complexity:* can only do shift right so far

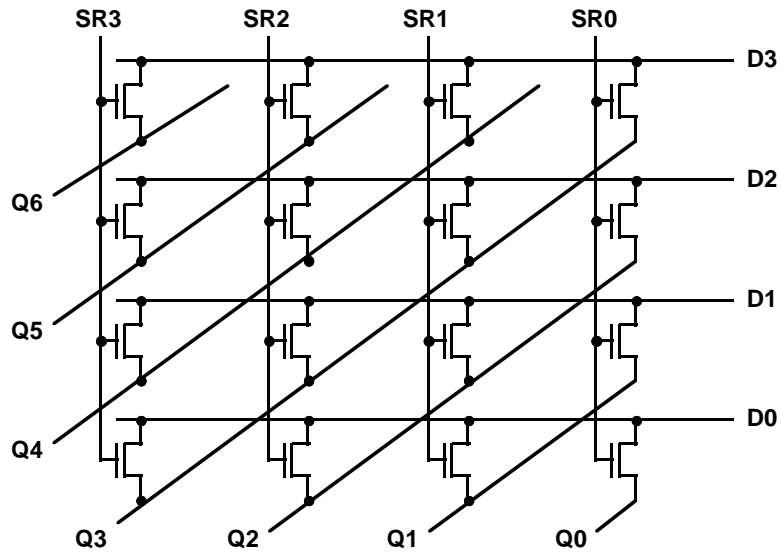
### Funnel Shifter

Instead Extract 32 bits of 64.



## Barrel Shifter

Technology-dependent solutions:



cs 152 I7 ALU.19

©DAP & SIK 1995

## Administrative Matters

- Video tapes of lectures available for viewing in 205 McLaughlin, Mon. to Fri. 9 AM to 5 PM; Wed & Fri. 6 PM to 10 PM
- 1st Midterm 2 weeks, Feb. 22; 5 PM to 8 PM for 1.5 hour test in Sibley Auditorium;  
2 hand drawn pieces of paper  
(will be able to bring to second midterm too)
- LaVal's Afterwards to meet students, TAs, profs
- Other topics?

cs 152 I7 ALU.20

©DAP & SIK 1995

## MULTIPLY

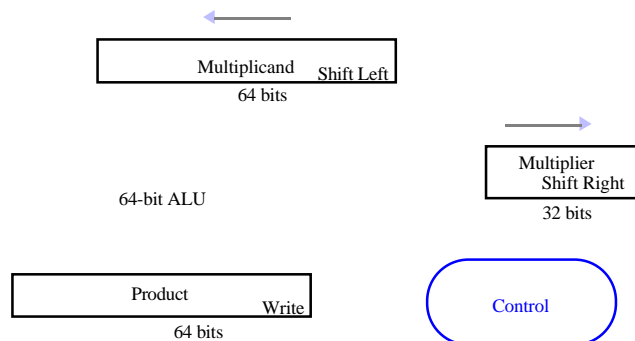
- Paper and pencil example:

Multiplicand	1000
Multiplier x	1001
	<hr/>
	1000
	0000
	0000
	<hr/>
Product	1001000

- $m$  bits  $\times$   $n$  bits =  $m+n$  bit product
- Binary makes it easy:
  - 0  $\Rightarrow$  place 0 ( 0  $\times$  multiplicand)
  - 1  $\Rightarrow$  place 0 ( 1  $\times$  multiplicand)
- 3 versions of multiply hardware & algorithm: successive refinement

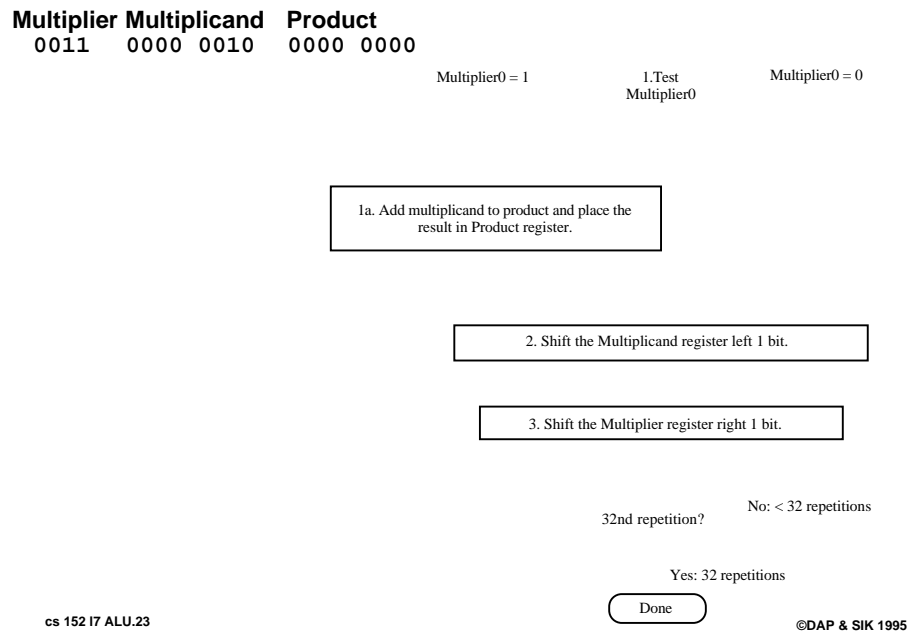
## MULTIPLY HARDWARE Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg





## Multiply Algorithm Version 1

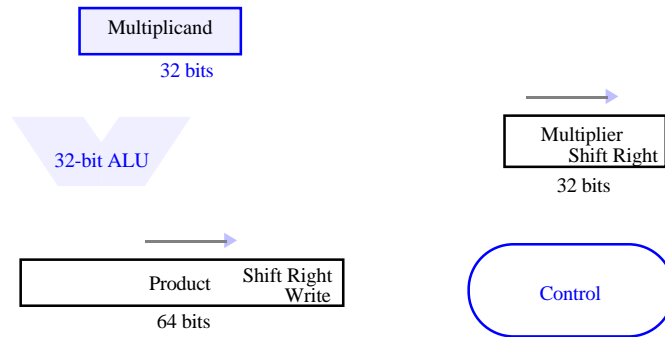


## Observations on Multiply Version 1

- 1 clock per cycle => ≈ 100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- 1/2 bits in multiplicand always 0  
=> 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted  
=> least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

## MULTIPLY HARDWARE Version 2

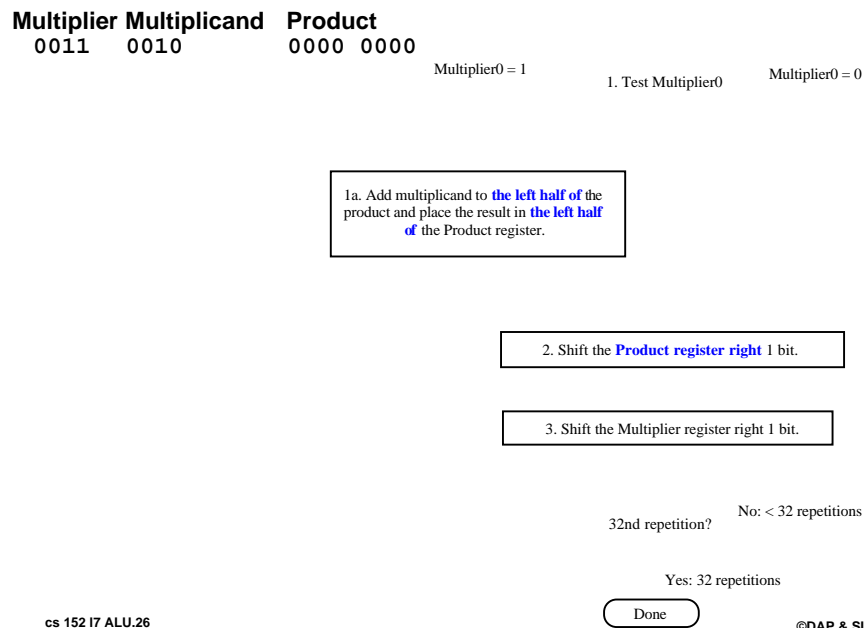
- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg



cs 152 I7 ALU.25

©DAP & SIK 1995

## Multiply Algorithm Version 2



cs 152 I7 ALU.26

©DAP & SIK 1995

## Observations on Multiply Version 2

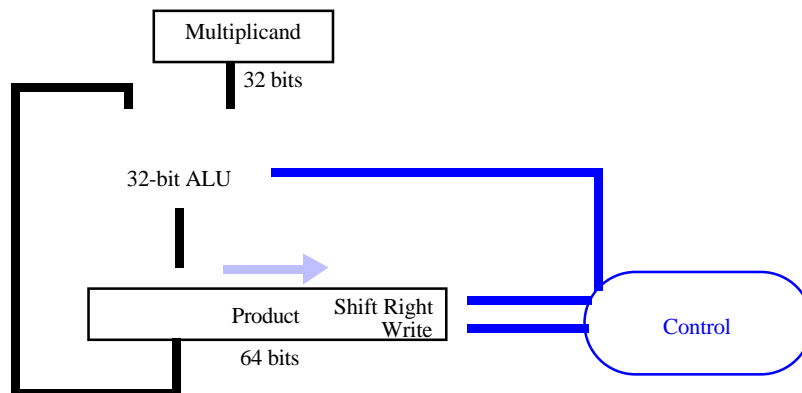
- ° Product register wastes space that exactly matches size of multiplier  
=> combine Multiplier register and Product register

cs 152 I7 ALU.27

©DAP & SIK 1995

## MULTIPLY HARDWARE Version 3

- ° 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)

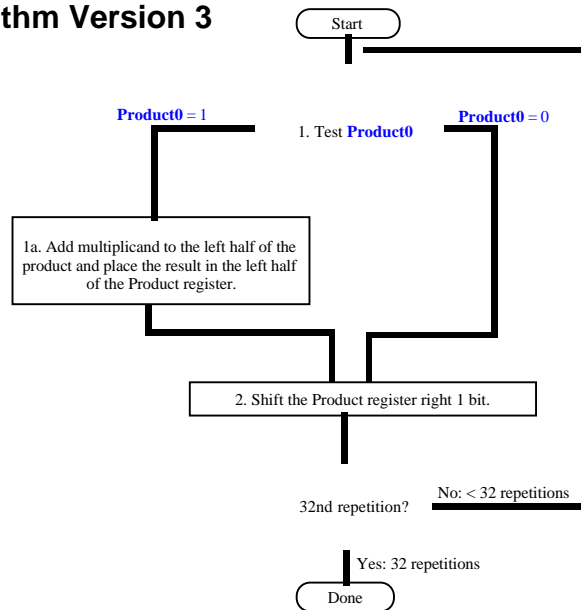


cs 152 I7 ALU.28

©DAP & SIK 1995

## Multiply Algorithm Version 3

Multiplicand    Product  
0010            0000 0011



## Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - Booth's Algorithm is more elegant way to multiply signed numbers using same hardware as before

## Motivation for Booth's Algorithm

- Example  $2 \times 6 = 0010 \times 0110$ :

x	0010	
+	0110	
+	0000	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0100	add (1 in multiplier)
+	0000	shift (0 in multiplier)
	00001100	

- ALU with add or subtract gets same result in more than one way:

$$\begin{array}{lcl}
 6 & = & -2 + 8 \\
 0110 & = & -0010 + 1000
 \end{array}
 , \text{ or}$$

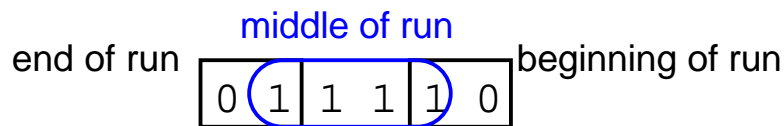
- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one. For example

x	0010	
+	0110	
+	0000	shift (0 in multiplier)
-	0010	sub (first 1 in multiplier)
+	0000	shift (middle of string of 1s)
+	0010	add (prior step had last 1)
	00001100	

cs 152 I7 ALU.31

©DAP & SIK 1995

## Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	000111 <u>1</u> 000
1	1	Middle of a run of 1s	000111 <u>1</u> 000
0	1	End of a run of 1s	000 <u>1</u> 111000
0	0	Middle of a run of 0s	000 <u>1</u> 111000

Originally for Speed since shift faster than add for his machine

cs 152 I7 ALU.32

©DAP & SIK 1995

## Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:

- 00: a. Middle of a string of 0s, so no arithmetic operations.
- 01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
- 10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
- 11: d. Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

<b>Multiplicand</b>	<b>Product (2 x 3)</b>	<b>Multiplicand</b>	<b>Product (2 x -3)</b>
0010	0000 0011 0	0010	0000 1101 0

## Summary

- Instruction Set drives the ALU design
- Shifter: success refinement from 1/bit at a time shift register to barrel shifter
- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
- There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)
- What's Missing from MIPS is Divide & Floating Point Arithmetic: Next time the Pentium Bug

**CS152**  
**Computer Architecture and Engineering**  
**Lecture 9: Designing a Single Cycle Datapath**

February 15, 1995

Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 datapath.1

©DAP & SIK 1995

## Recap of the On-line Design Note Book

- Top 10 things to put in your on-line design notebook
  - 10. Start: type “date” and copy & paste into your notebook.
  - 9. What is the goal/objective of today?
  - 8. Description of any problem: what did you see? what did you do?
  - 7. Keep track of the time whenever you do a new “compile.”
  - 6. Procedures for testing and running experiments.
  - 5. Outputs of tests and experiments.
  - 4. Insights and thoughts you have while you work.
  - 3. Copy & paste headers of important emails.
  - 2.. Last thing of the day: One line summary => Notebook Index.
  - 1. Finish: type “date” and copy & paste into your notebook.

cs 152 datapath.2

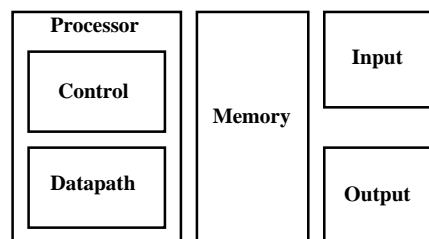
©DAP & SIK 1995

## Outline of Today's Lecture

- **Recap and Introduction (5 minutes)**
- **Where are we with respect to the BIG picture? (15 minutes)**
- **Questions and Administrative Matters (5 minutes)**
- **The Steps of Designing a Processor (10 minutes)**
- **Datapath and timing for Reg-Reg Operations (15 minutes)**
- **Break (5 minutes)**
- **Datapath for Logical Operations with Immediate (5 minutes)**
- **Datapath for Load and Store Operations (10 minutes)**
- **Datapath for Branch and Jump Operations (10 minutes)**

## The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**



- **Today's Topic: Datapath Design**



## The Big Picture: The Performance Perspective

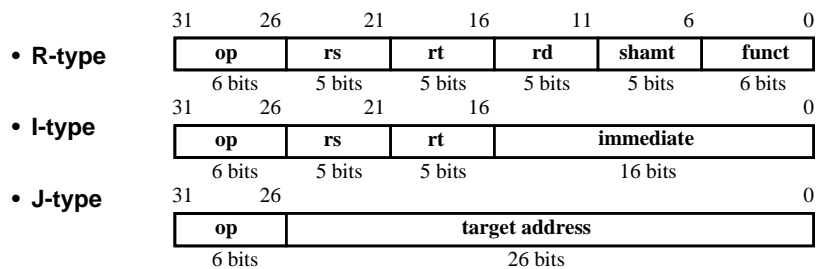
- Performance of a machine was determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- In the next two lectures:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time

cs 152 datapath.5

©DAP & SIK 1995

## The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

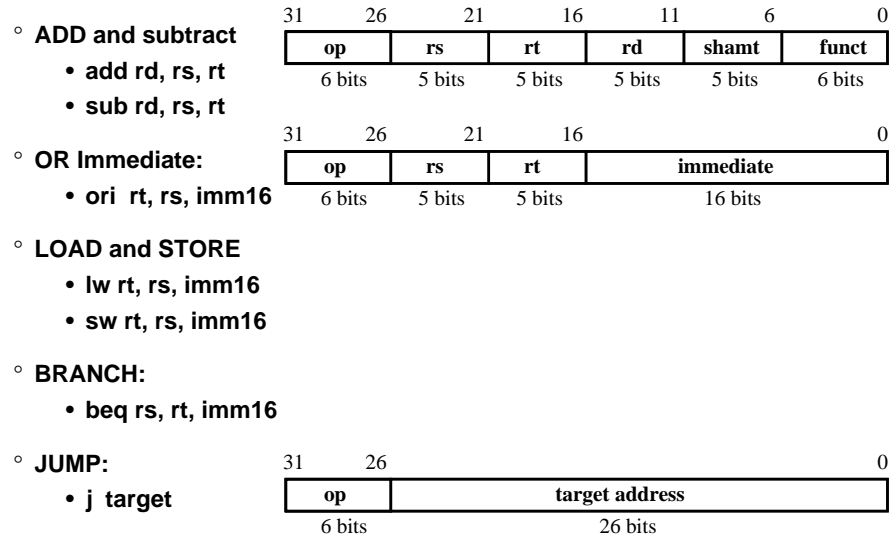


- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

cs 152 datapath.6

©DAP & SIK 1995

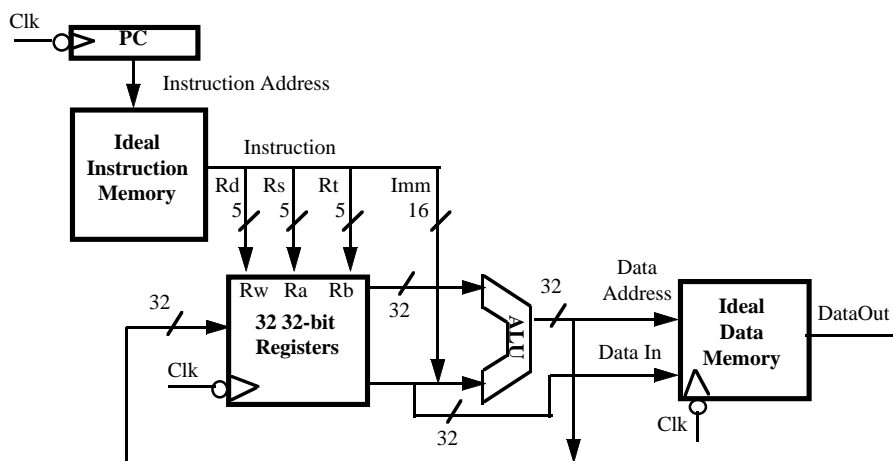
## The MIPS Subset



cs 152 datapath.7

©DAP & SIK 1995

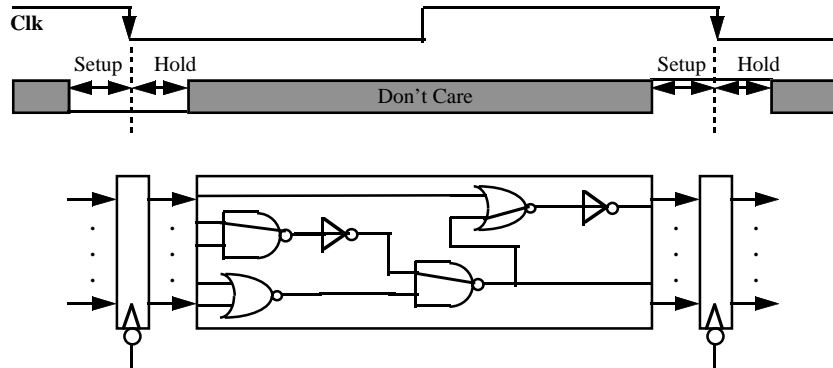
## An Abstract View of the Implementation



cs 152 datapath.8

©DAP & SIK 1995

## Clocking Methodology



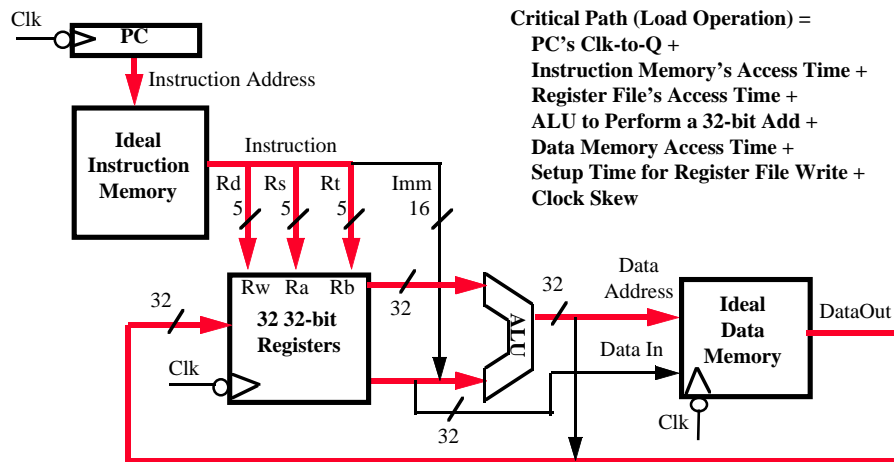
- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- $(\text{CLK-to-Q} + \text{Shortest Delay Path} - \text{Clock Skew}) > \text{Hold Time}$

cs 152 datapath.9

©DAP & SIK 1995

## An Abstract View of the Critical Path

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”



cs 152 datapath.10

©DAP & SIK 1995

## Questions and Administrative Matters (5 Minutes)

- Discussion Section Room Change for Thursday:
  - Before 109 Morgan => New 373 Soda
  - Effective: 2/16/1995, Thursday
- One more time:
  - All teams must be at least four people
  - We want you to learn to work in a big team

## The Steps of Designing a Processor

- Instruction Set Architecture => Register Transfer Language
- Register Transfer Language =>
  - Datapath components
  - Datapath interconnect
- Datapath components => Control signals
- Control signals => Control logic

## RTL: The ADD Instruction

◦ **add rd, rs, rt**

- **mem[PC]**                      **Fetch the instruction from memory**
- **$R[rd] \leftarrow R[rs] + R[rt]$**               **The ADD operation**
- **$PC \leftarrow PC + 4$**                       **Calculate the next instruction's address**

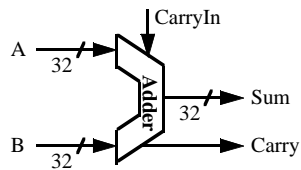
## RTL: The Load Instruction

◦ **lw rt, rs, imm16**

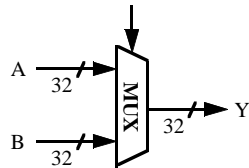
- **mem[PC]**                      **Fetch the instruction from memory**
- **$Addr \leftarrow R[rs] + \text{SignExt}(imm16)$**   
   **Calculate the memory address**
- **$R[rt] \leftarrow \text{Mem}[Addr]$**               **Load the data into the register**
- **$PC \leftarrow PC + 4$**                       **Calculate the next instruction's address**

## Combinational Logic Elements

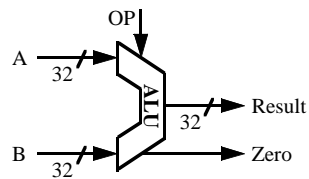
### ◦ Adder



### ◦ MUX



### ◦ ALU



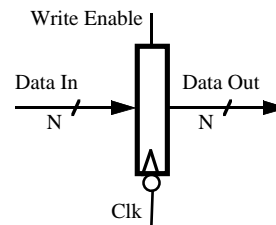
cs 152 datapath.15

©DAP & SIK 1995

## Storage Element: Register

### ◦ Register

- Similar to the D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - 0: Data Out will not change
  - 1: Data Out will become Data In



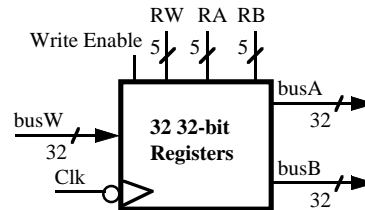
cs 152 datapath.16

©DAP & SIK 1995

## Storage Element: Register File

- Register File consists of 32 registers:

- Two 32-bit output busses: busA and busB
- One 32-bit input bus: busW



- Register is selected by:

- RA selects the register to put on busA
- RB selects the register to put on busB
- RW selects the register to be written via busW when Write Enable is 1

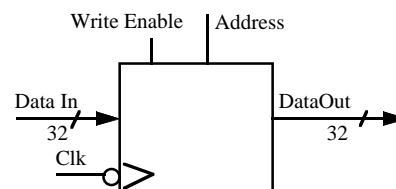
- Clock input (CLK)

- The CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - RA or RB valid => busA or busB valid after “access time.”

## Storage Element: Idealized Memory

- Memory (idealized)

- One input bus: Data In
- One output bus: Data Out



- Memory word is selected by:

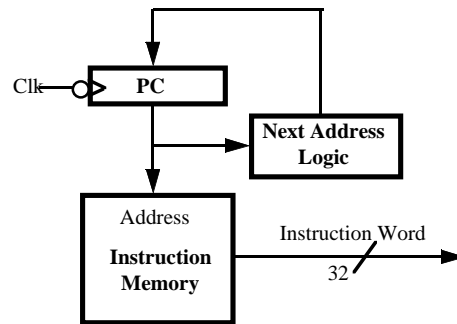
- Address selects the word to put on Data Out
- Write Enable = 1: address selects the memory memory word to be written via the Data In bus

- Clock input (CLK)

- The CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - Address valid => Data Out valid after “access time.”

## Overview of the Instruction Fetch Unit

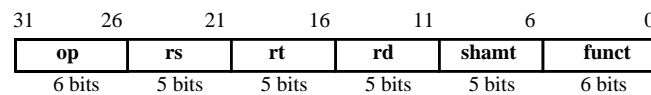
- The common RTL operations
  - Fetch the Instruction:  $\text{mem}[\text{PC}]$
  - Update the program counter:
    - Sequential Code:  $\text{PC} \leftarrow \text{PC} + 4$
    - Branch and Jump  $\text{PC} \leftarrow \text{"something else"}$



cs 152 datapath.19

©DAP & SIK 1995

## RTL: The ADD Instruction



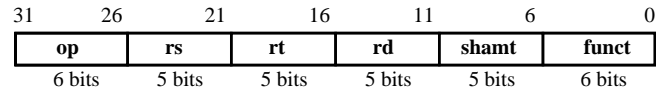
- **add rd, rs, rt**
  - $\text{mem}[\text{PC}]$                       Fetch the instruction from memory
  - $\text{R}[\text{rd}] \leftarrow \text{R}[\text{rs}] + \text{R}[\text{rt}]$               The actual operation
  - $\text{PC} \leftarrow \text{PC} + 4$                       Calculate the next instruction's address

cs 152 datapath.20

©DAP & SIK 1995



## RTL: The Subtract Instruction

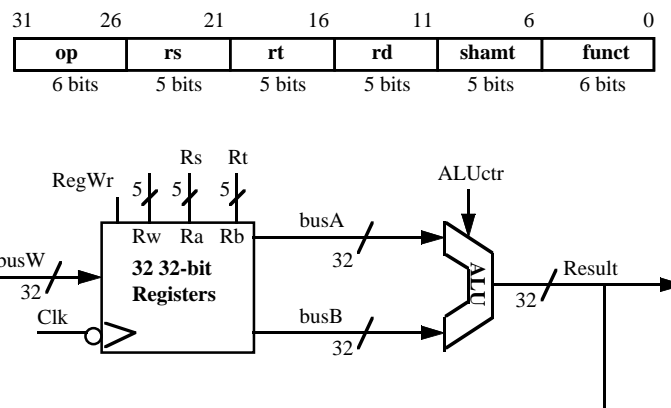


◦ **sub rd, rs, rt**

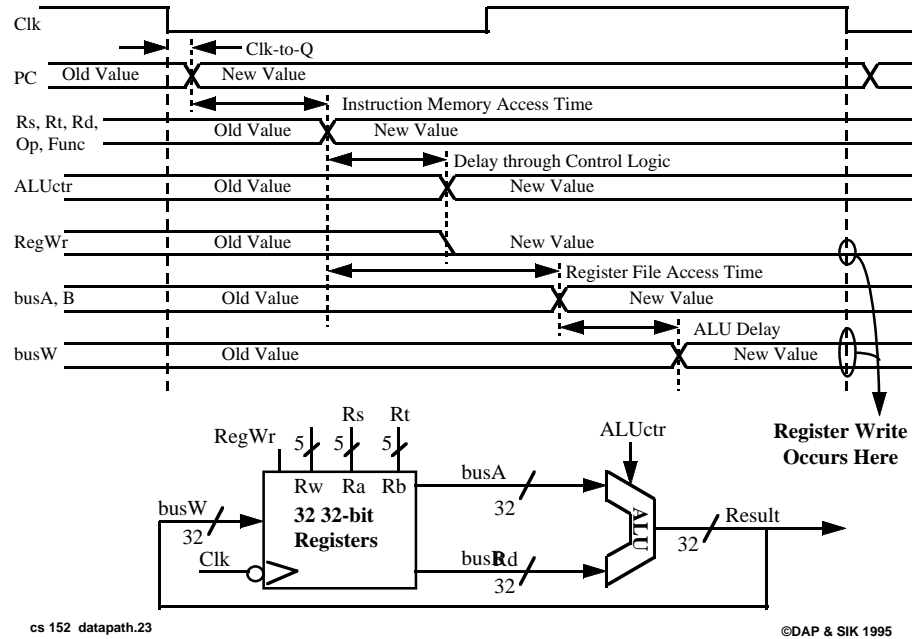
- **mem[PC]**                      **Fetch the instruction from memory**
- **R[rd] <- R[rs] - R[rt]**              **The actual operation**
- **PC <- PC + 4**                      **Calculate the next instruction's address**

## Datapath for Register-Register Operations

- **R[rd] <- R[rs] op R[rt]**              **Example: add rd, rs, rt**
- **Ra, Rb, and Rw** comes from instruction's **rs, rt, and rd** fields
  - **ALUctr** and **RegWr**: control logic after decoding the instruction

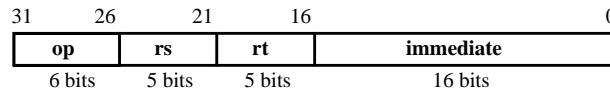


## Register-Register Timing



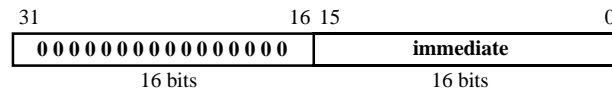
**Break (5 Minutes)**

## RTL: The OR Immediate Instruction



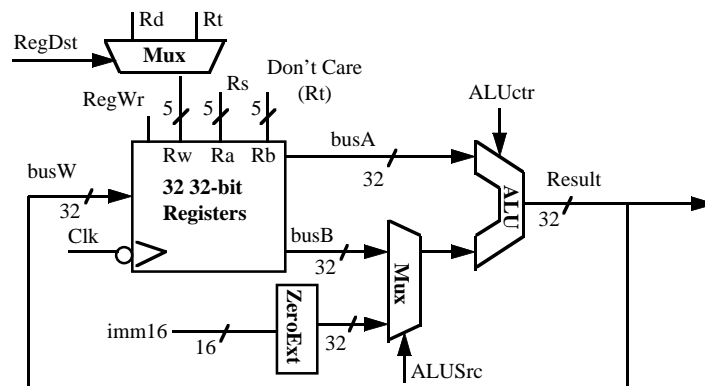
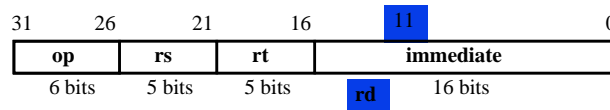
◦ **ori rt, rs, imm16**

- **mem[PC]** **Fetch the instruction from memory**
  
- **R[rt] <- R[rs] or ZeroExt(imm16)** **The OR operation**
  
- **PC <- PC + 4** **Calculate the next instruction's address**

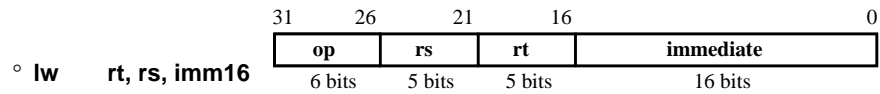


## Datapath for Logical Operations with Immediate

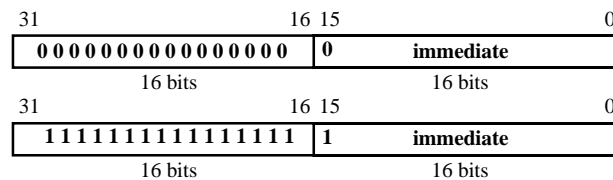
◦ **R[rt] <- R[rs] op ZeroExt[imm16]]** **Example: ori rt, rs, imm16**



## RTL: The Load Instruction



- mem[PC]                      Fetch the instruction from memory
- Addr <- R[rs] + SignExt(imm16)                      Calculate the memory address
- R[rt] <- Mem[Addr]                      Load the data into the register
- PC <- PC + 4                      Calculate the next instruction's address

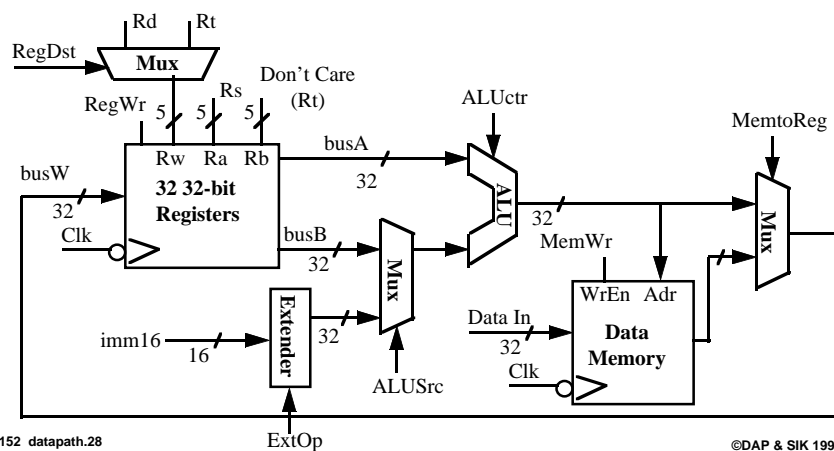
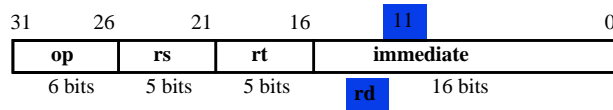


cs 152 datapath.27

©DAP & SIK 1995

## Datapath for Load Operations

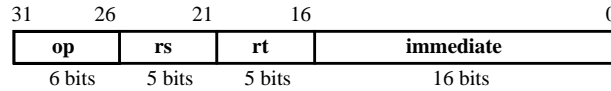
- ° R[rt] <- Mem[R[rs] + SignExt[imm16]]     Example: lw     rt, rs, imm16



cs 152 datapath.28

©DAP & SIK 1995

## RTL: The Store Instruction

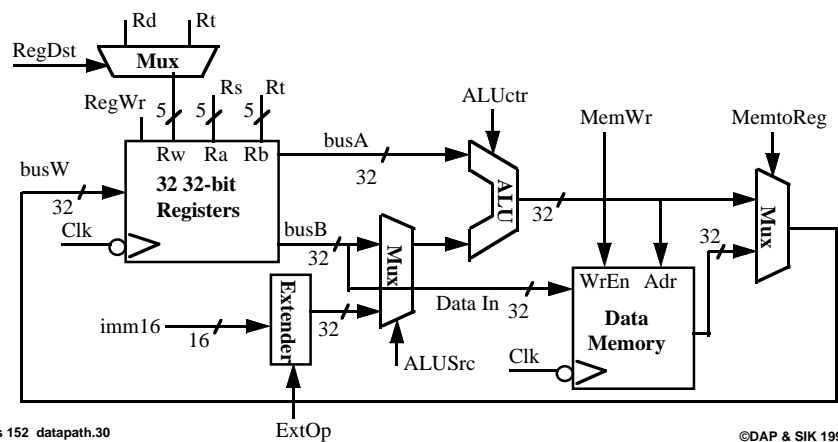
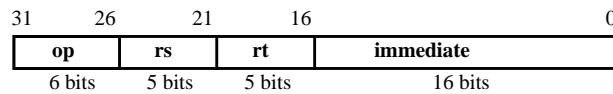


◦ **sw**     **rt, rs, imm16**

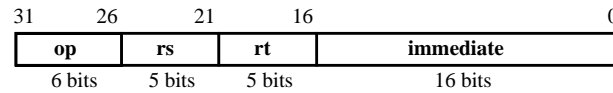
- **mem[PC]**                                      **Fetch the instruction from memory**
  
- **Addr <- R[rs] + SignExt(imm16)**                                      **Calculate the memory address**
  
- **Mem[Addr] <- R[rt]**                                      **Store the register into memory**
  
- **PC <- PC + 4**                                      **Calculate the next instruction's address**

## Datapath for Store Operations

◦ **Mem[R[rs] + SignExt[imm16] <- R[rt]]**     **Example: sw    rt, rs, imm16**



## RTL: The Branch Instruction



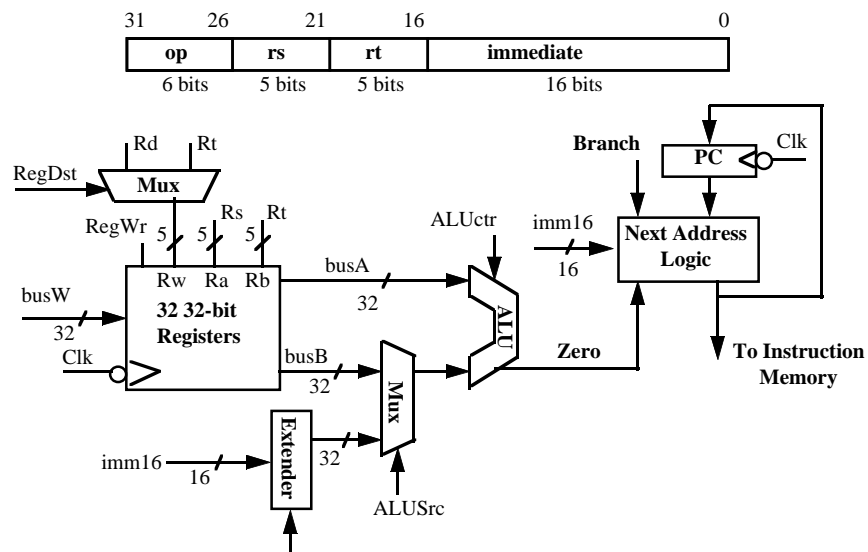
◦ **beq** rs, rt, imm16

- **mem[PC]**                                      **Fetch the instruction from memory**
- **Cond <- R[rs] - R[rt]**                      **Calculate the branch condition**
- **if (COND eq 0)**                              **Calculate the next instruction's address**
  - **PC <- PC + 4 + ( SignExt(imm16) x 4 )**
- **else**
  - **PC <- PC + 4**

## Datapath for Branch Operations

◦ **beq** rs, rt, imm16

**We need to compare Rs and Rt!**



## Binary Arithmetics for the Next Address

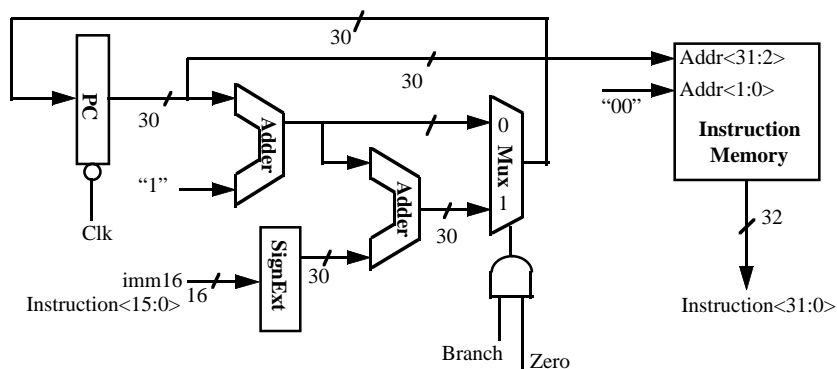
- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation:  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4$
  - Branch operation:  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4 + \text{SignExt}[\text{Imm16}] * 4$
- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit  $PC\langle 31:2 \rangle$ :
  - Sequential operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
  - Branch operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
  - In either case: Instruction Memory Address =  $PC\langle 31:2 \rangle$  concat “00”

cs 152 datapath.33

©DAP & SIK 1995

## Next Address Logic: Expensive and Fast Solution

- Using a 30-bit PC:
  - Sequential operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
  - Branch operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
  - In either case: Instruction Memory Address =  $PC\langle 31:2 \rangle$  concat “00”

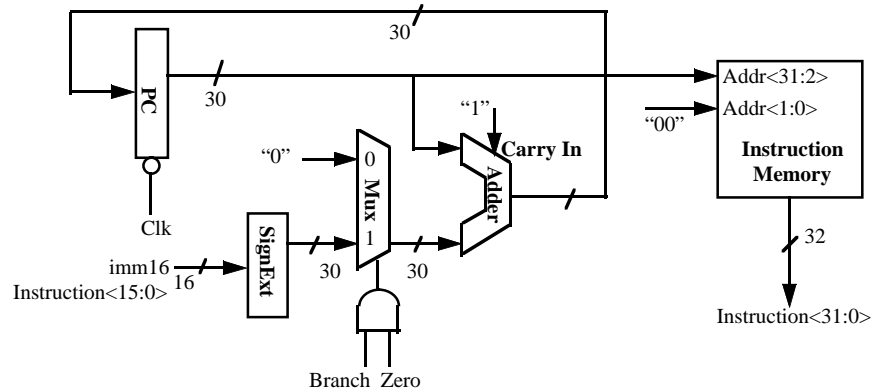


cs 152 datapath.34

©DAP & SIK 1995

## Next Address Logic: Cheap and Slow Solution

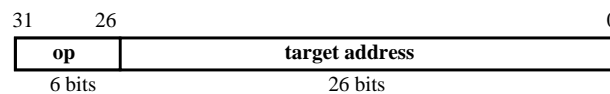
- Why is this slow?
  - Cannot start the address add until Zero (output of ALU) is valid
- Does it matter that this is slow in the overall scheme of things?
  - Probably not here. Critical path is the load operation.



cs 152 datapath.35

©DAP & SIK 1995

## RTL: The Jump Instruction



◦ j target

- mem[PC] Fetch the instruction from memory
- PC<31:2> <- PC<31:29> concat target<25:0> Calculate the next instruction's address

cs 152 datapath.36

©DAP & SIK 1995

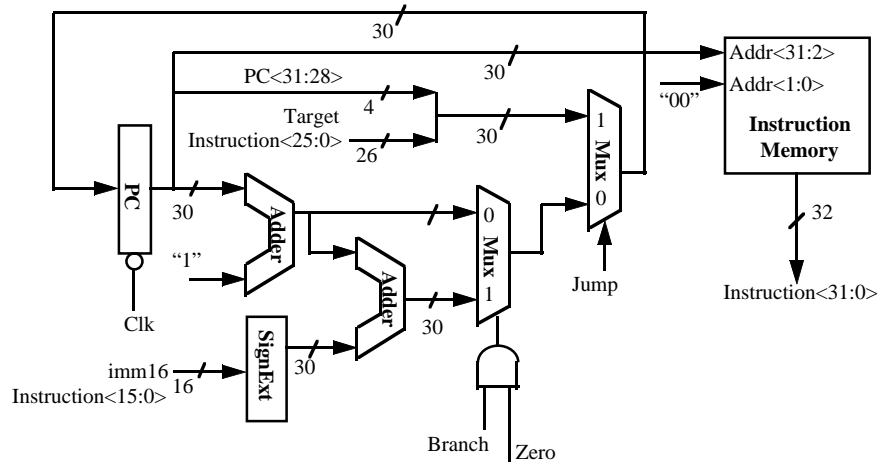


## Instruction Fetch Unit

- ```

° j      target
• PC<31:2> <- PC<31:29> concat target<25:0>

```

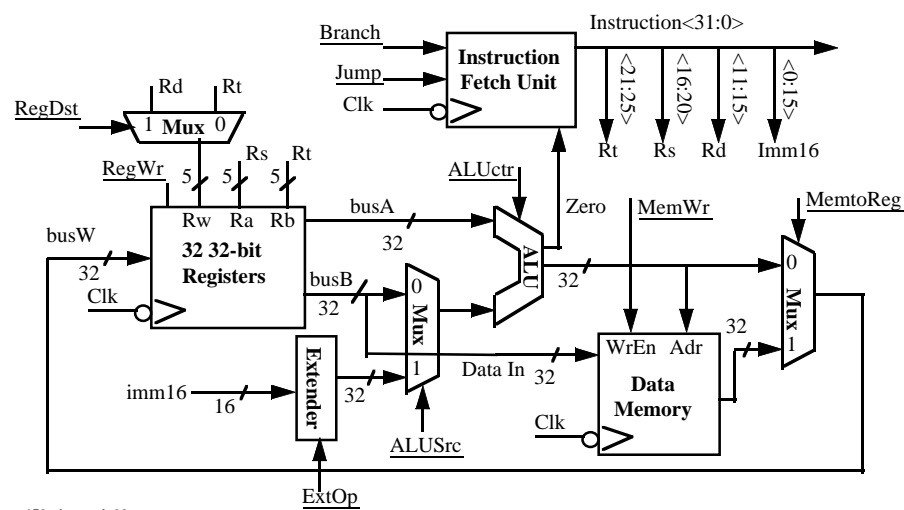


cs 152 datapath.37

©DAP & SIK 1995

## Putting it All Together: A Single Cycle Datapath

- We have everything except control signals (underline)



cs 152 datapath.38

©DAP & SIK 1995

## Where to get more information?

- To be continued ...

# CS152

## Computer Architecture and Engineering

### Lecture 10: Designing a Single Cycle Control

February 17, 1995

Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

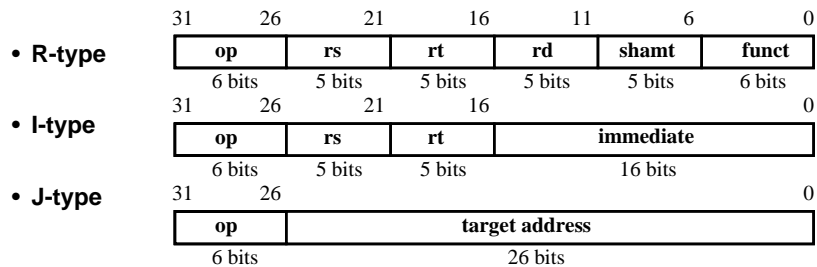
Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 control.1

©DAP & SIK 1995

### Recap: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

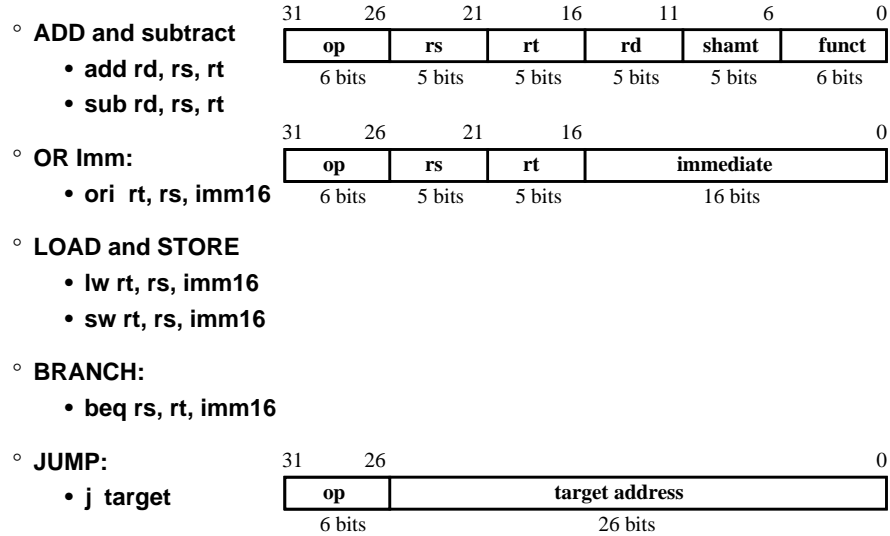


- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination registers specifier
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

cs 152 control.2

©DAP & SIK 1995

## Recap: The MIPS Subset

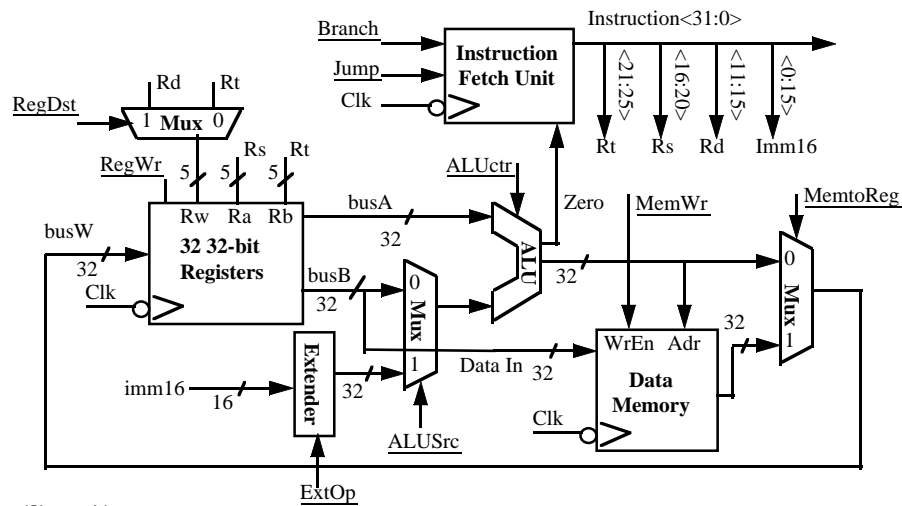


cs 152 control.3

©DAP & SIK 1995

## Recap: A Single Cycle Datapath

- ° We have everything except control signals (underline)
- Today's lecture will show you how to generate the control signals

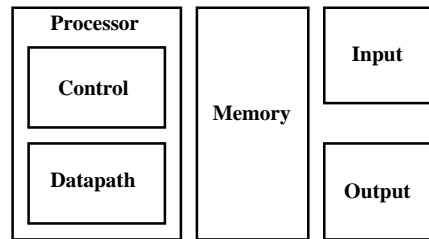


cs 152 control.4

©DAP & SIK 1995

## The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

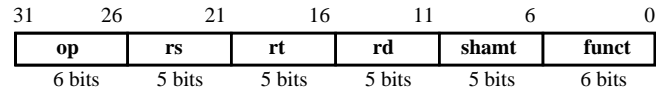


- Today's Topic: Designing the Control for the Single Cycle Datapath

## Outline of Today's Lecture

- Recap and Introduction (10 minutes)
- Control for Register-Register & Or Immediate instructions (10 minutes)
- Questions and Administrative Matters (5 minutes)
- Control signals for Load, Store, Branch, & Jump (15 minutes)
- Building a local controller: ALU Control (10 minutes)
- Break (5 minutes)
- The main controller (20 minutes)
- Summary (5 minutes)

## RTL: The ADD Instruction

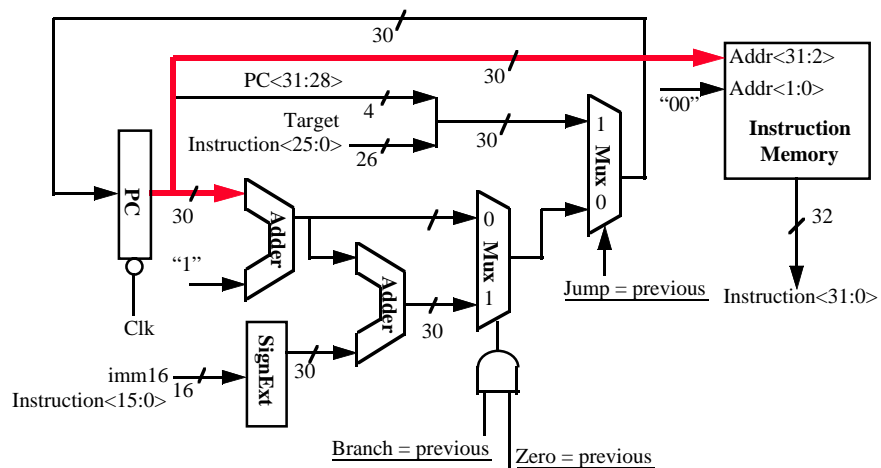


◦ **add rd, rs, rt**

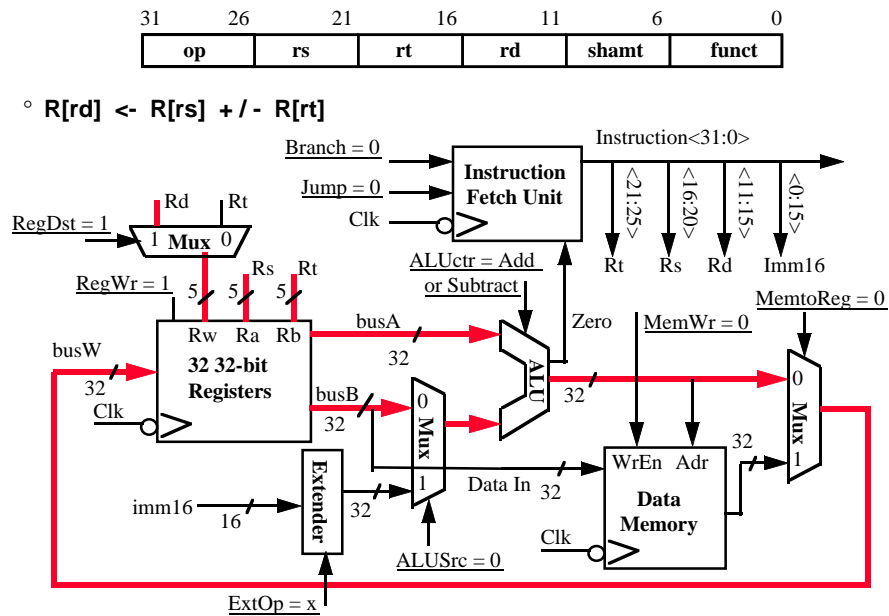
- **mem[PC]**                      **Fetch the instruction from memory**
- **R[rd] <- R[rs] + R[rt]**            **The actual operation**
- **PC <- PC + 4**                      **Calculate the next instruction's address**

## Instruction Fetch Unit at the Beginning of Add / Subtract

- **Fetch the instruction from Instruction memory: Instruction <- mem[PC]**
  - **This is the same for all instructions**



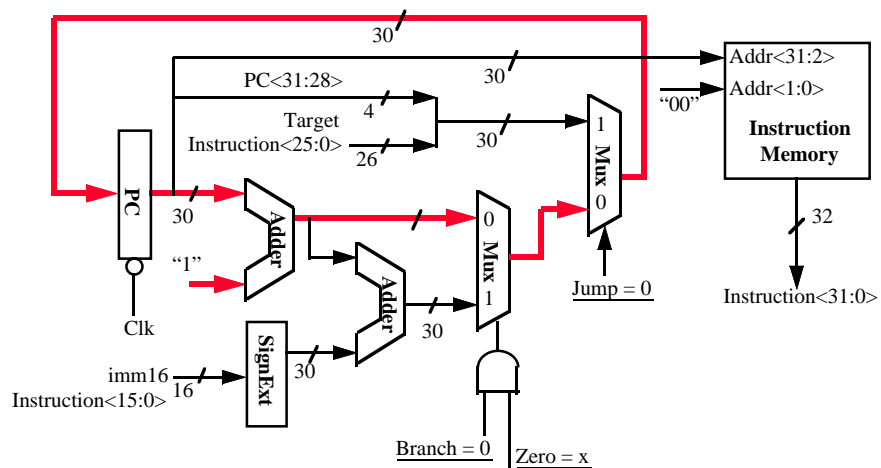
## The Single Cycle Datapath during Add and Subtract



©DAP & SIK 1995

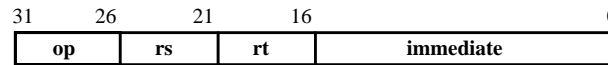
### Instruction Fetch Unit at the End of Add and Subtract

- **PC ← PC + 4**
  - This is the same for all instructions except: Branch and Jump

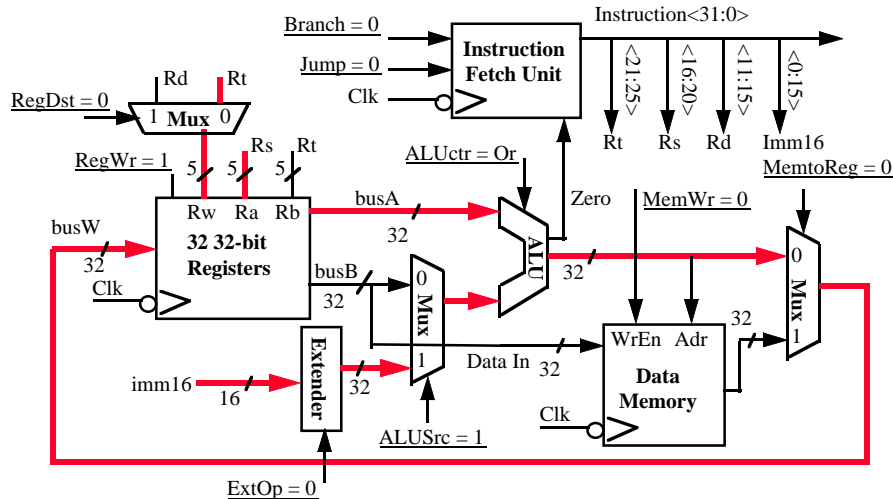


©DAP & SIK 1995

## The Single Cycle Datapath during Or Immediate



- $R[rt] \leftarrow R[rs]$  or  $\text{ZeroExt}[Imm16]$



cs 152 control.11

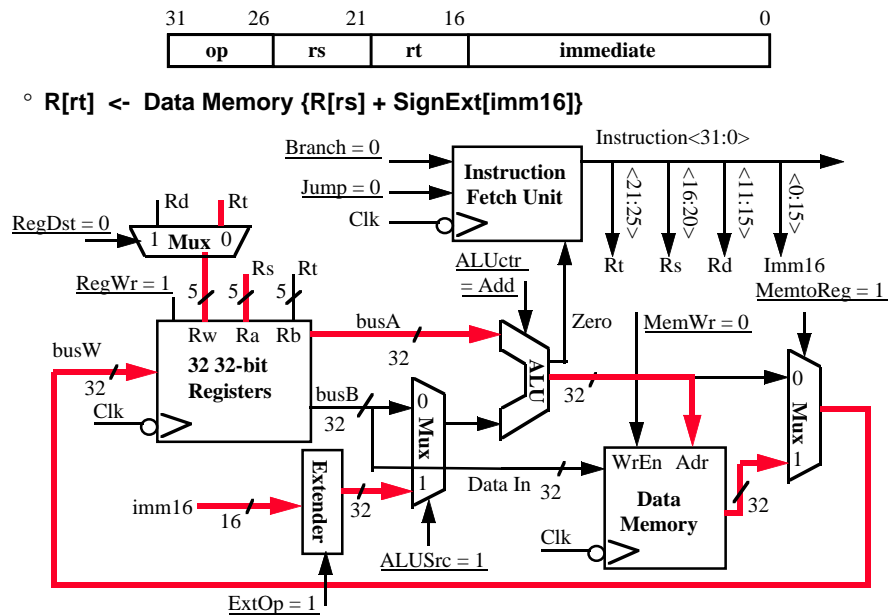
©DAP & SIK 1995

## Questions and Administrative Matters

- Midterm next Wednesday 2/22/95:
  - 5:00pm to 8:00pm, Sibley Auditorium
  - No class on that day
- Things to bring to midterm:
  - Pencil, calculator, two 8.5" x 11" pages of handwritten notes
  - Sit every other chair, every other row (odd row & odd seat)
  - Meet at LaVal's pizza after the midterm (\$5/person)
    - Need a headcount. How many are definitely coming?
- Next homework assignment due Tuesday, 2./21/95
  - Monday is a holiday

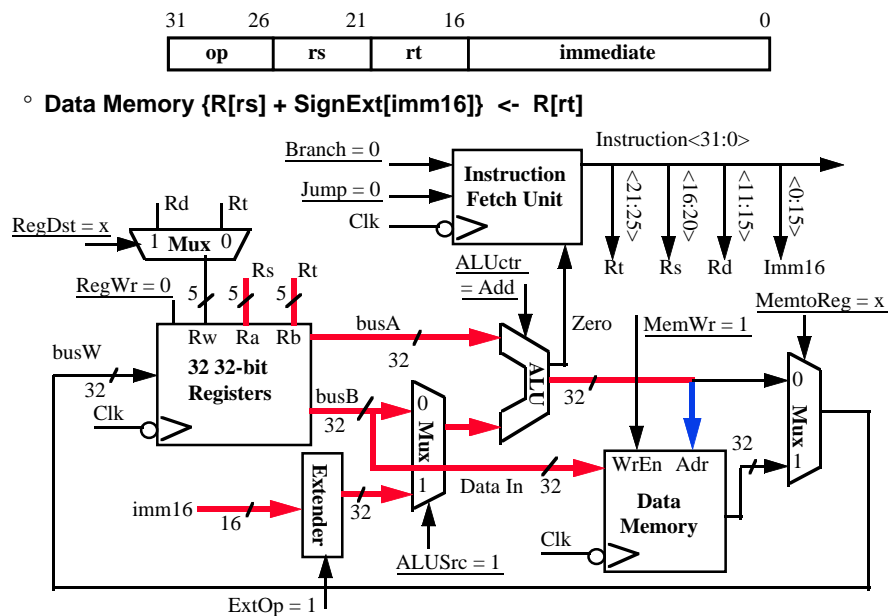


## The Single Cycle Datapath during Load



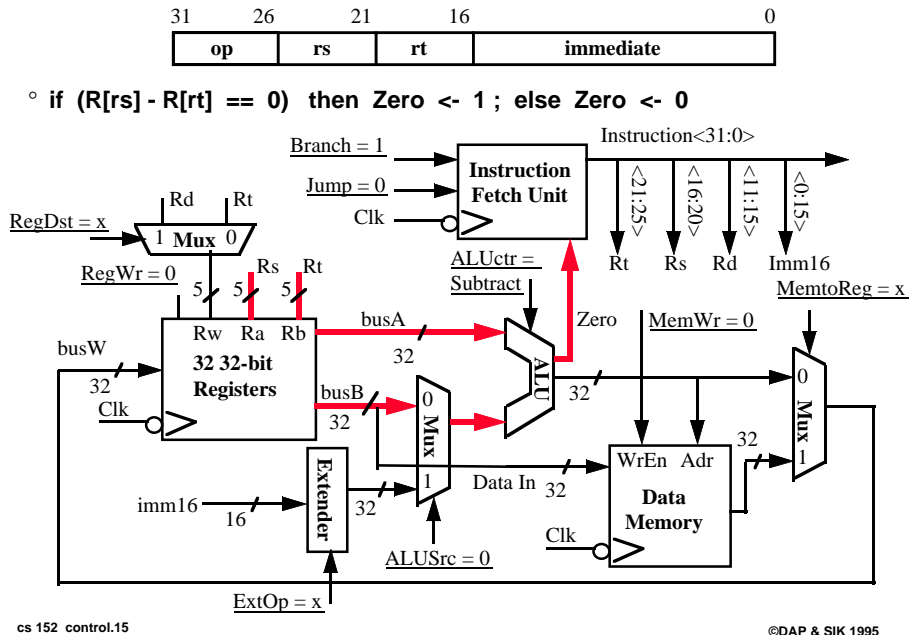
©DAP & SIK 1995

## The Single Cycle Datapath during Store

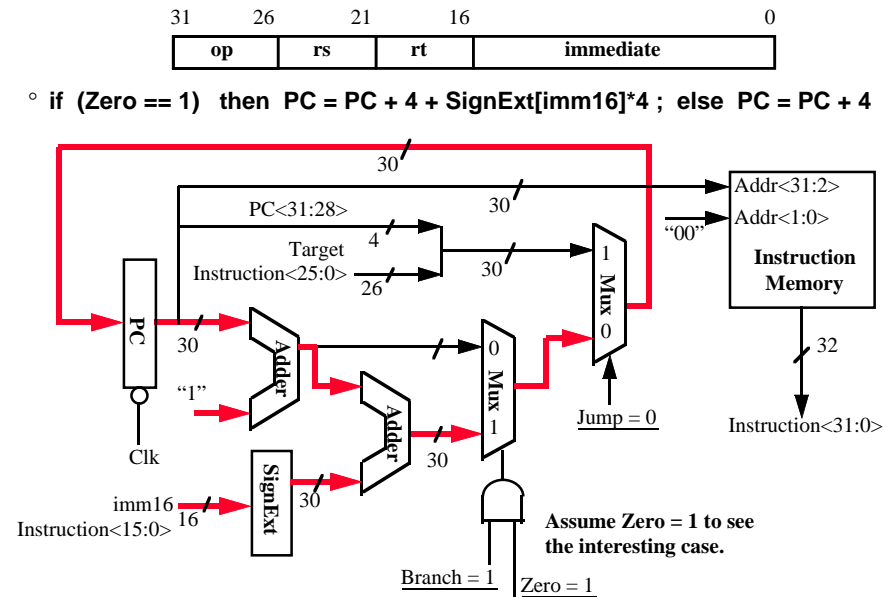


©DAP & SIK 1995

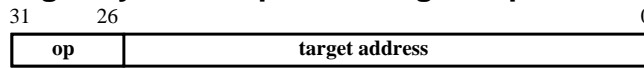
## The Single Cycle Datapath during Branch



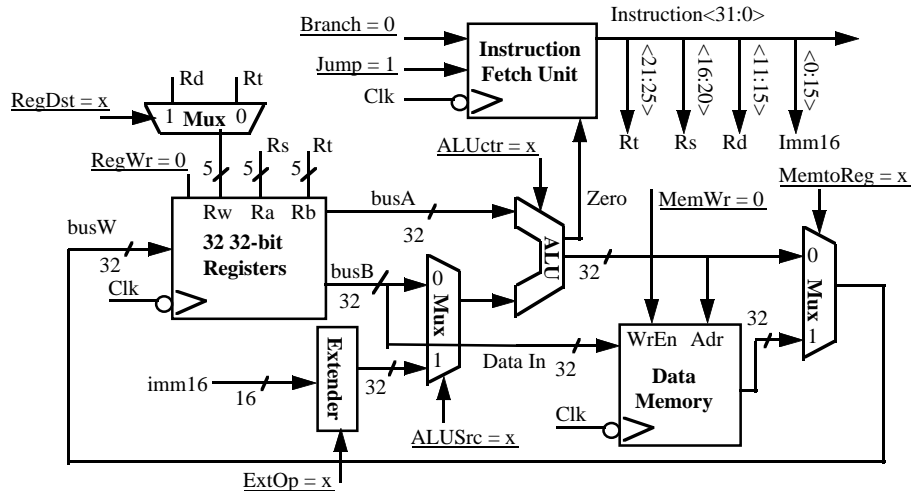
## Instruction Fetch Unit at the End of Branch



## The Single Cycle Datapath during Jump



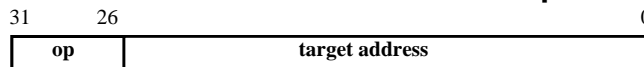
- **Nothing to do! Make sure control signals are set correctly!**



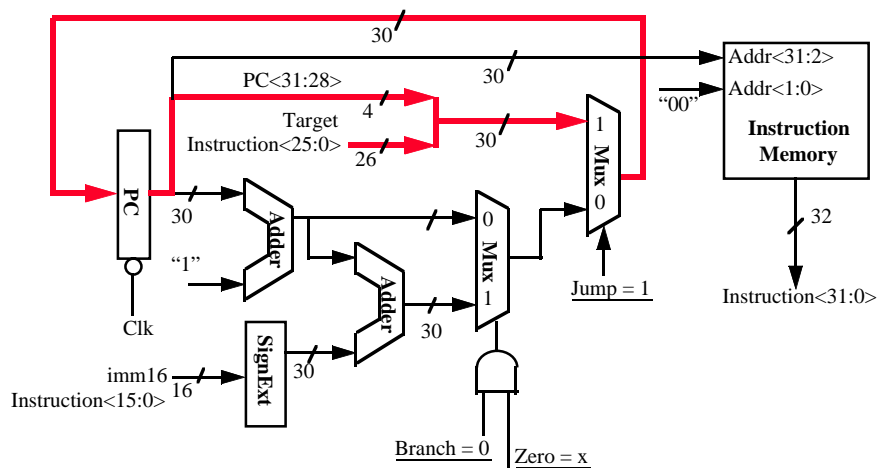
cs 152 control.17

©DAP & SIK 1995

### Instruction Fetch Unit at the End of Jump



- ```
PC <- PC<31:29> concat target<25:0> concat "00"
```



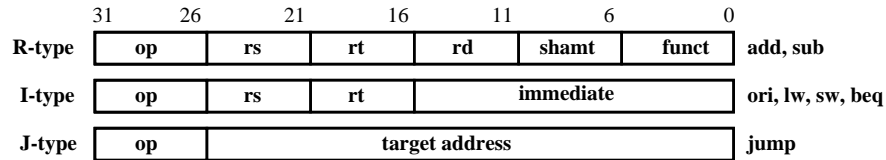
cs 152 control.18

©DAP & SIK 1995

## A Summary of the Control Signals

See Appendix A

	func	10 0000	10 0010	We Don't Care :-)			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100 00 0010
		add	sub	ori	lw	sw	beq jump
RegDst		1	0	0	0	x	x
ALUSrc		0	0	1	1	1	0
MemtoReg		0	0	0	1	x	x
RegWrite		1	1	1	1	0	0
MemWrite		0	0	0	0	1	0
Branch		0	0	0	0	0	1
Jump		0	0	0	0	0	0
ExtOp		x	x	0	1	1	x
ALUctr<2:0>		Add	Subtract	Or	Add	Add	Subtract xxx

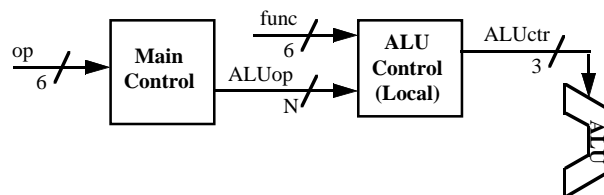


cs 152 control.19

©DAP & SIK 1995

## The Concept of Local Decoding

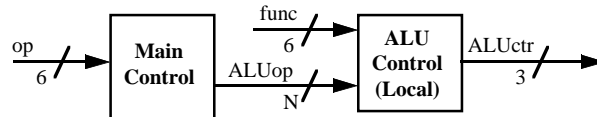
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



cs 152 control.20

©DAP & SIK 1995

## The Encoding of ALUop



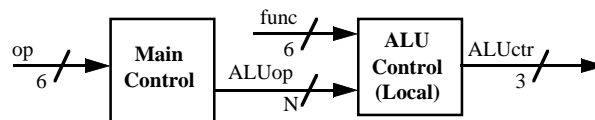
- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) “R-type” instructions
  - “I-type” instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) “R-type” instructions
  - “I-type” instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

cs 152 control.21

©DAP & SIK 1995

## The Decoding of the “func” Field



	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

	31	26	21	16	11	6	0
R-type	op	rs	rt	rd	shamt	funct	

Recall ALU Homework (also P. 286 text):

funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

cs 152 control.22



ALUctr<2:0>	ALU Operation
000	Add
001	Subtract
010	And
110	Or
111	Set-on-less-than

©DAP & SIK 1995

## The Truth Table for ALUctr

ALUop (Symbolic)	R-type "R-type"	ori	lw	sw	beq	func<3:0>	Instruction Op.
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	0000	add
						0010	subtract
						0100	and
						0101	or
						1010	set-on-less-than

ALUop bit<2> bit<1> bit<0>			func bit<3> bit<2> bit<1> bit<0>				ALU Operation	ALUctr bit<2> bit<1> bit<0>		
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

**Break (5 Minutes)**

## The Logic Equation for ALUctr<2>

ALUop			func				ALUctr<2>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

$$\begin{aligned} \text{ALUctr<2>} = & \text{!ALUop<2>} \& \text{ALUop<0>} + \\ & \text{ALUop<2>} \& \text{!func<2>} \& \text{func<1>} \& \text{!func<0>} \end{aligned}$$

## The Logic Equation for ALUctr<1>

ALUop			func				ALUctr<1>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

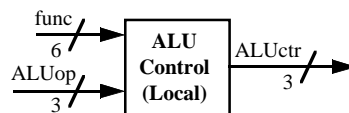
$$\begin{aligned} \text{ALUctr<1>} = & \text{!ALUop<2>} \& \text{!ALUop<0>} + \\ & \text{ALUop<2>} \& \text{!func<2>} \& \text{!func<0>} \end{aligned}$$

## The Logic Equation for ALUctr<0>

ALUop			func				ALUctr<0>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

- $ALUctr<0> = !ALUop<2> \& ALUop<0>$   
 $+ ALUop<2> \& !func<3> \& func<2> \& !func<1> \& func<0>$   
 $+ ALUop<2> \& func<3> \& !func<2> \& func<1> \& !func<0>$

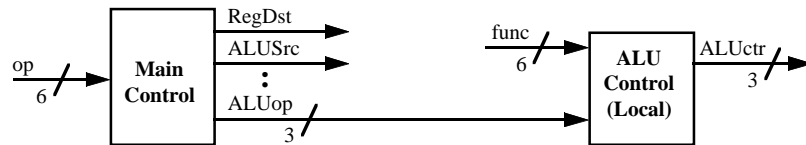
## The ALU Control Block



- $ALUctr<2> = !ALUop<2> \& ALUop<0> +$   
 $ALUop<2> \& !func<2> \& func<1> \& !func<0>$
- $ALUctr<1> = !ALUop<2> \& !ALUop<0> +$   
 $ALUop<2> \& !func<2> \& !func<0>$
- $ALUctr<0> = !ALUop<2> \& ALUop<0>$   
 $+ ALUop<2> \& !func<3> \& func<2> \& !func<1> \& func<0>$   
 $+ ALUop<2> \& func<3> \& !func<2> \& func<1> \& !func<0>$



## The “Truth Table” for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUOp (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUOp <2>	1	0	0	0	0	x
ALUOp <1>	0	1	0	0	0	x
ALUOp <0>	0	0	0	0	1	x

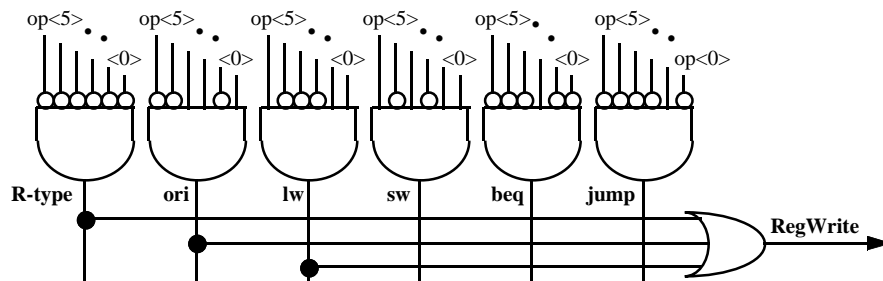
cs 152 control.29

©DAP & SIK 1995

## The “Truth Table” for RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	x	x	x

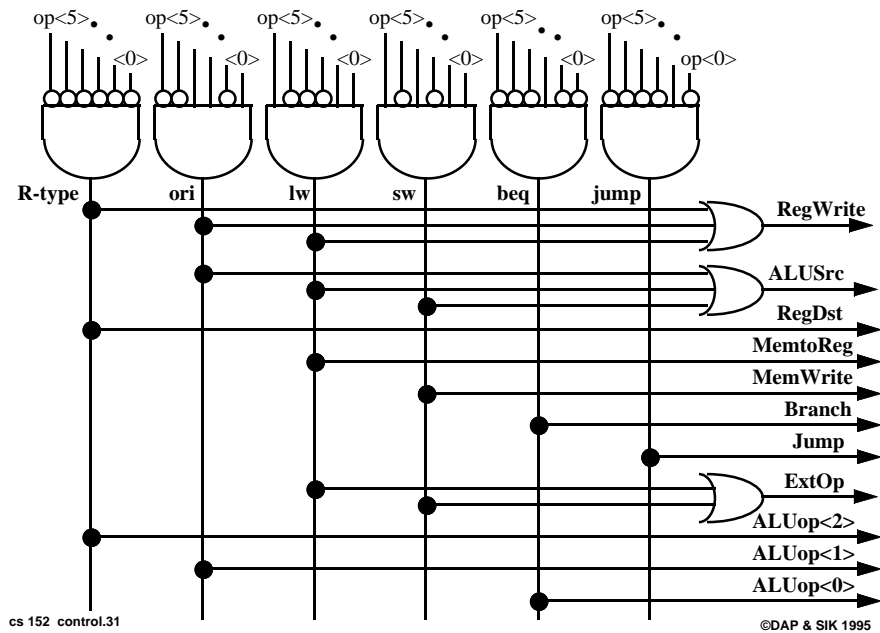
- $\text{RegWrite} = \text{R-type} + \text{ori} + \text{lw}$ 
  - $= !\text{op}<5> \& !\text{op}<4> \& !\text{op}<3> \& !\text{op}<2> \& !\text{op}<1> \& !\text{op}<0>$  (R-type)
  - $+ !\text{op}<5> \& !\text{op}<4> \& \text{op}<3> \& \text{op}<2> \& !\text{op}<1> \& \text{op}<0>$  (ori)
  - $+ \text{op}<5> \& !\text{op}<4> \& !\text{op}<3> \& !\text{op}<2> \& \text{op}<1> \& \text{op}<0>$  (lw)



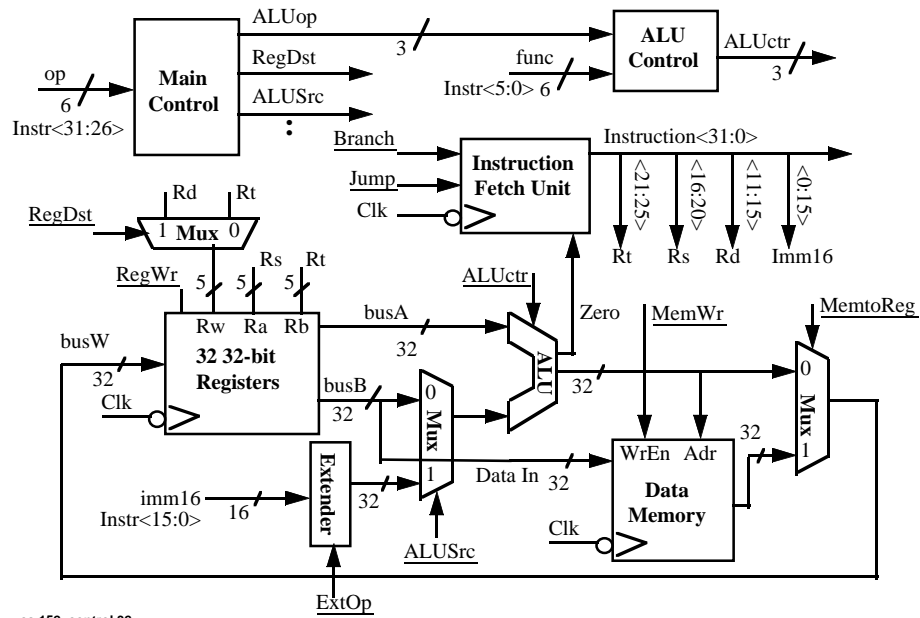
cs 152 control.30

©DAP & SIK 1995

## PLA Implementation of the Main Control



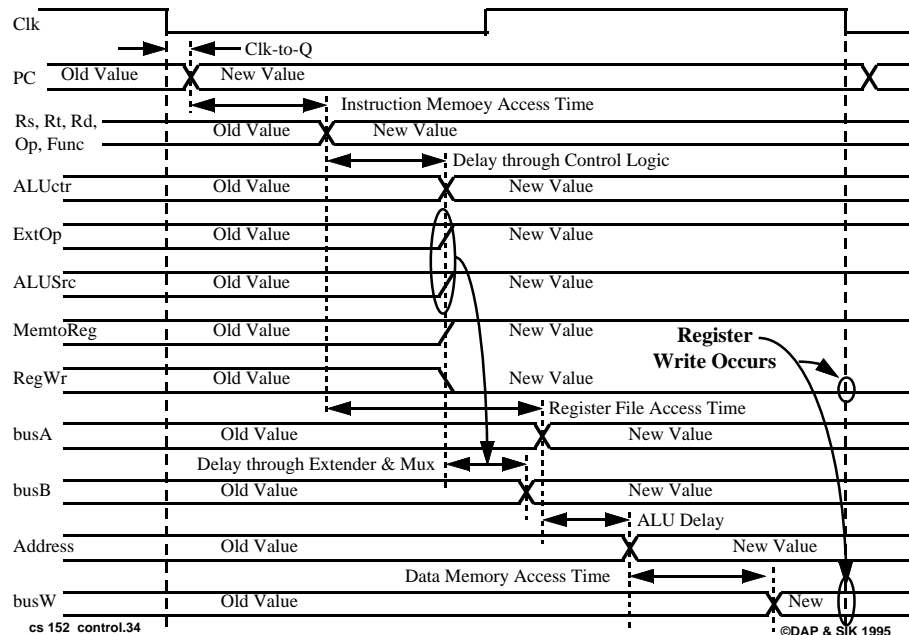
## Putting it All Together: A Single Cycle Processor



## How is this Different from a Real MIPS Processor?

- The effect of load in a real MIPS Processor is delayed:
  - lw \$1, 100 (\$2) // Load Register R1
  - add \$3, \$1, \$0 // Move “old” R1 into R3
  - add \$4, \$1, \$0 // Move “new” R1 into R4
- The effect of load in our single cycle process is NOT delayed
  - lw \$1, 100 (\$2) // Load Register R1
  - add \$3, \$1, \$0 // Move “new” R1 into R3
- The effect of branch and jump in a real MIPS Processor is delayed:
  - Instruction Address: 0x00 j 1000
  - Instruction Address: 0x04 add \$1, \$2, \$3
  - Instruction Address: 0x1000 sub \$1, \$2, \$3
- Branch and jump in our single cycle process is NOT delayed
  - Instruction Address: 0x00 j 1000
  - Instruction Address: 0x1000 sub \$1, \$2, \$3

## Worst Case Timing



## Drawback of this Single Cycle Processor

- Long cycle time:
  - Cycle time must be long enough for the load instruction:  
PC's Clock -to-Q +  
Instruction Memory Access Time +  
Register File Access Time +  
ALU Delay (address calculation) +  
Data Memory Access Time +  
Register File Setup Time +  
Clock Skew
- Cycle time is much longer than needed for all other instructions

## Where to get more information?

- Chapter 5.1 to 5.3 of your text book:
  - Daid Patterson and John Hennessy, "Computer Organization & Design: The Hardware / Software Interface," Morgan Kaufman Publishers, San Mateo, California, 1994.
- One of the best PhD thesis on processor design:
  - Manolis Katevenis, "Reduced Instruction Set Computer Architecture for VLSI," PhD Dissertation, EECS, U C Berkeley, 1982.
- For a reference on the MIPS architecture:
  - Gerry Kane, "MIPS RISC Architecture," Prentice Hall.

# CS152

## Computer Architecture and Engineering

### Lecture 11: Designing a Multiple Cycle Processor

February 24, 1995

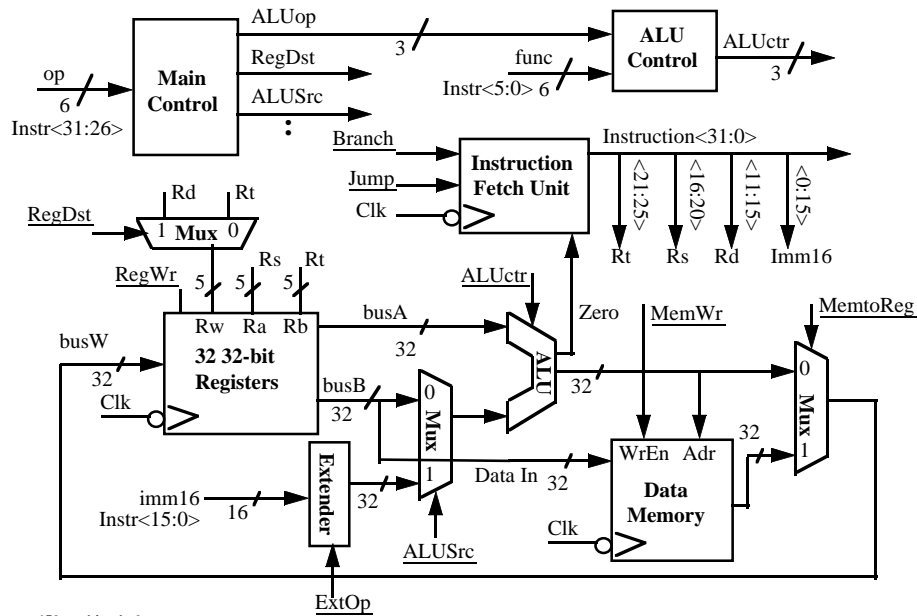
Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 multipath..1

©DAP & SIK 1995

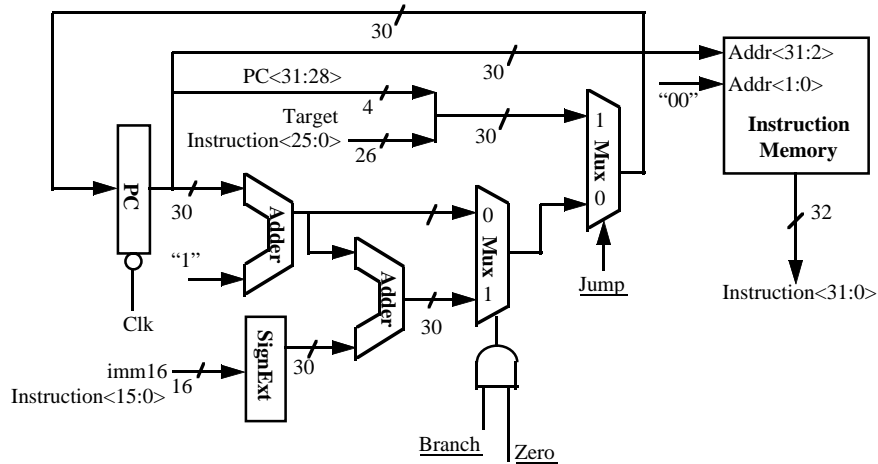
### A Single Cycle Processor



cs 152 multipath..2

©DAP & SIK 1995

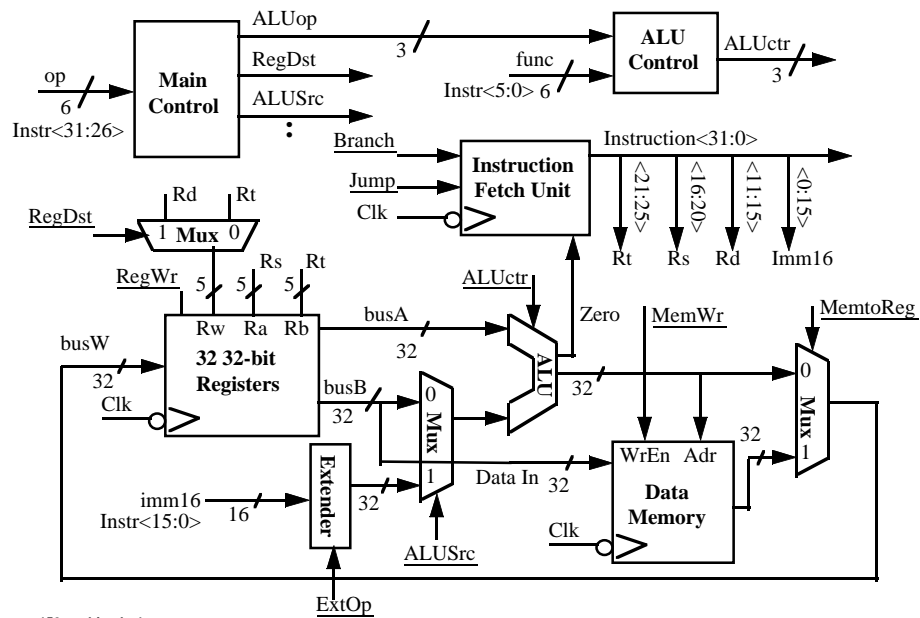
## Push: Instruction Fetch Unit



cs 152 multipath..3

©DAP & SIK 1995

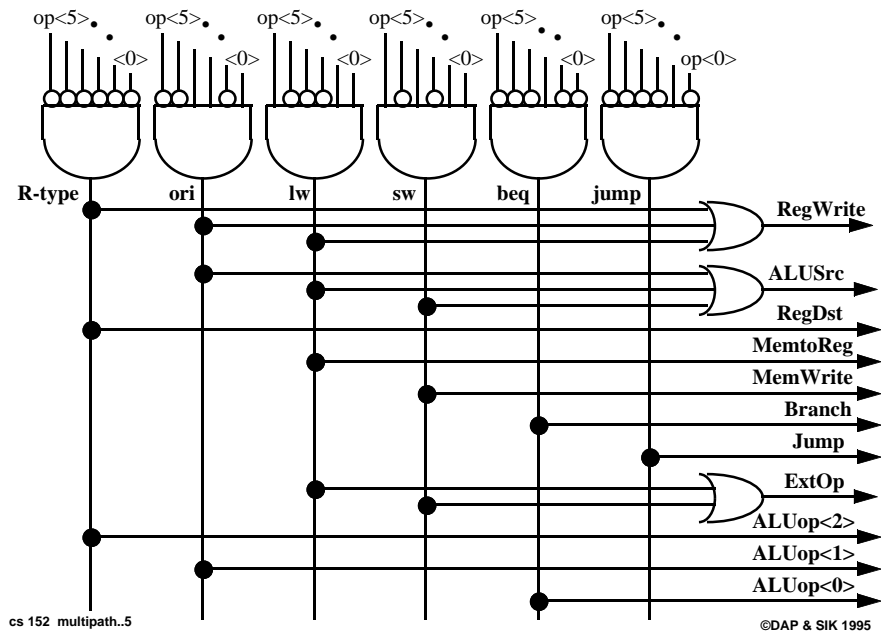
## Pop: A Single Cycle Processor



cs 152 multipath..4

©DAP & SIK 1995

## Push: The Main Control



## Outline of Today's Lecture

- Recap and Introduction (5 minutes)
- Introduction to the Concept of Multiple Cycle Processor (15 minutes)
- Questions and Administrative Matters (5 minutes)
- Multiple Cycle Implementation of R-type Instructions (15 minutes)
- What is a Multiple Cycle Delay Path and Why is it Bad? (10 minutes)
- Break (5 minutes)
- Multiple Cycle Implementation of Or Immediate (5 minutes)
- Multiple Cycle Implementation of Load and Store (15 minutes)
- Putting it all Together (5 minutes)

## Drawbacks of this Single Cycle Processor

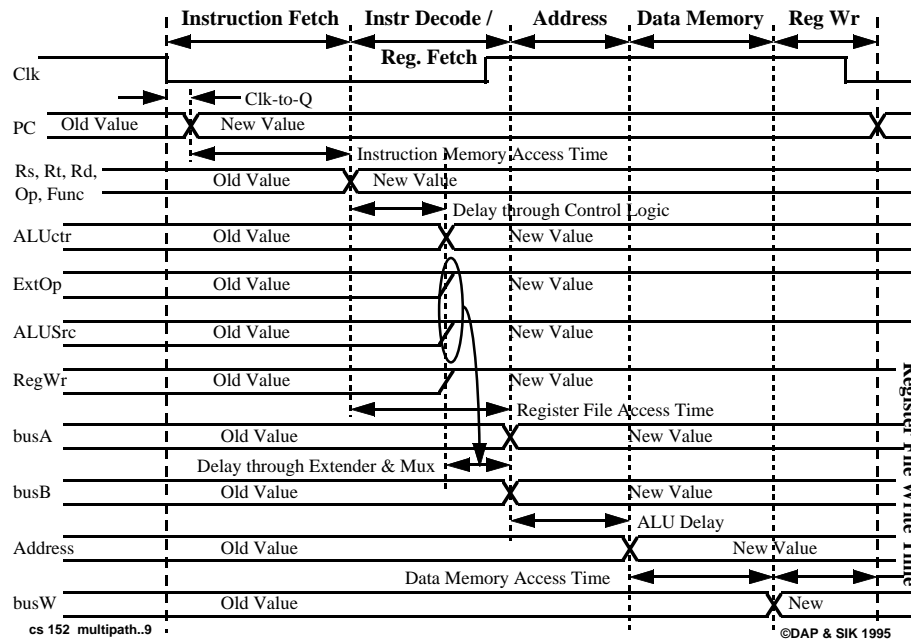
- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    - PC's Clock -to-Q +
    - Instruction Memory Access Time +
    - Register File Access Time +
    - ALU Delay (address calculation) +
    - Data Memory Access Time +
    - Register File Setup Time +
    - Clock Skew
- Cycle time is much longer than needed for all other instructions.  
Examples:
  - R-type instructions do not require data memory access
  - Jump does not require ALU operation nor data memory access

## Overview of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction



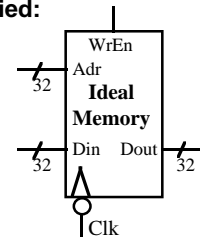
## The Five Steps of a Load Instruction



## Register File & Memory Write Timing: Ideal vs. Reality

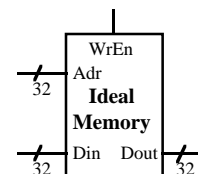
◦ In previous lectures, register file and memory are simplified:

- Write happens at the clock tick
- Address, data, and write enable must be stable one “set-up” time before the clock tick



◦ In real life:

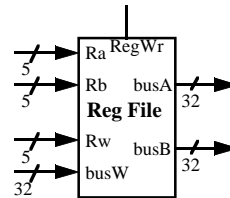
- Neither register file nor ideal memory has the clock input
- The write path is a combinational logic delay path:
  - Write enable goes to 1 and Din settles down
  - Memory write access delay
  - Din is written into mem[address]
- Important: Address and Data must be stable BEFORE Write Enable goes to 1



## Race Condition Between Address and Write Enable

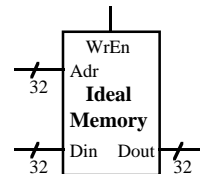
- This “real” (no clock input) register file may not work reliably in the single cycle processor because:

- We cannot guarantee  $Rw$  will be stable BEFORE  $RegWr = 1$
- There is a “race” between  $Rw$  (address) and  $RegWr$  (write enable)



- The “real” (no clock input) memory may not work reliably in the single cycle processor because:

- We cannot guarantee Address will be stable BEFORE  $WrEn = 1$
- There is a race between  $Adr$  and  $WrEn$

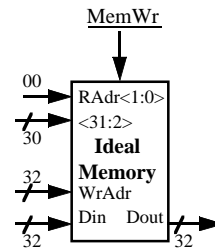


## How to Avoid this Race Condition?

- Solution for the multiple cycle implementation:
  - Make sure Address is stable by the end of Cycle N
  - Assert Write Enable signal ONE cycle later at Cycle (N + 1)
  - Address cannot change until Write Enable is disasserted

## Dual-Port Ideal Memory

- Dual Port Ideal Memory
  - Independent Read (RAdr, Dout) and Write (WAdr, Din) ports
  - Read and write (to different location) can occur at the same cycle
- Read Port is a combinational path:
  - Read Address Valid -->
  - Memory Read Access Delay -->
  - Data Out Valid
- Write Port is also a combinational path:
  - MemWrite = 1 -->
  - Memory Write Access Delay -->
  - Data In is written into location[WAdr]

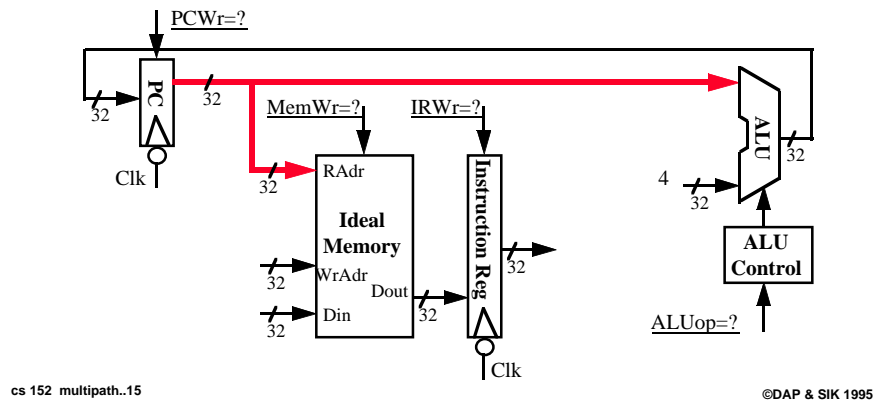
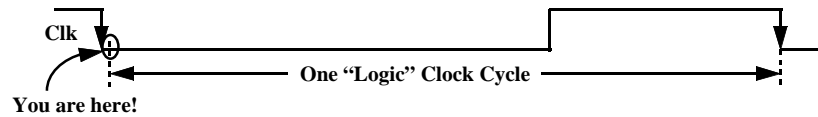


## Questions and Administrative Matters

## Instruction Fetch Cycle: In the Beginning

- Every cycle begins right AFTER the clock tick:

- $\text{mem}[\text{PC}] \quad \text{PC} < 31:0 > + 4$



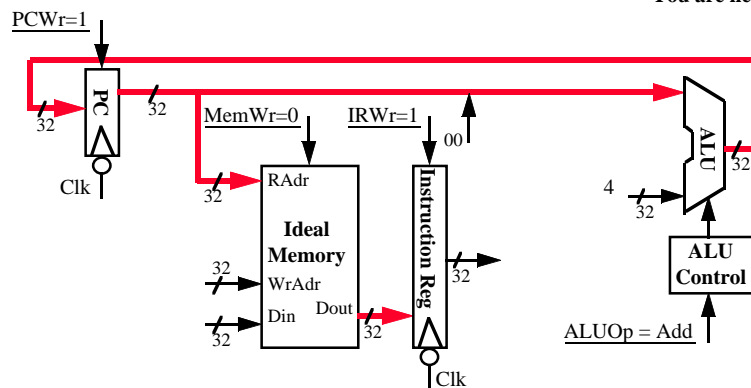
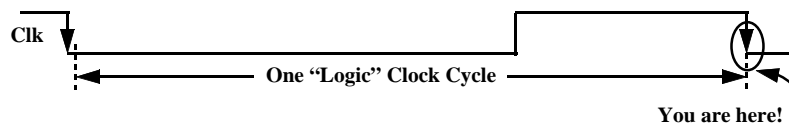
cs 152 multipath..15

©DAP & SIK 1995

## Instruction Fetch Cycle: The End

- Every cycle ends AT the next clock tick (storage element updates):

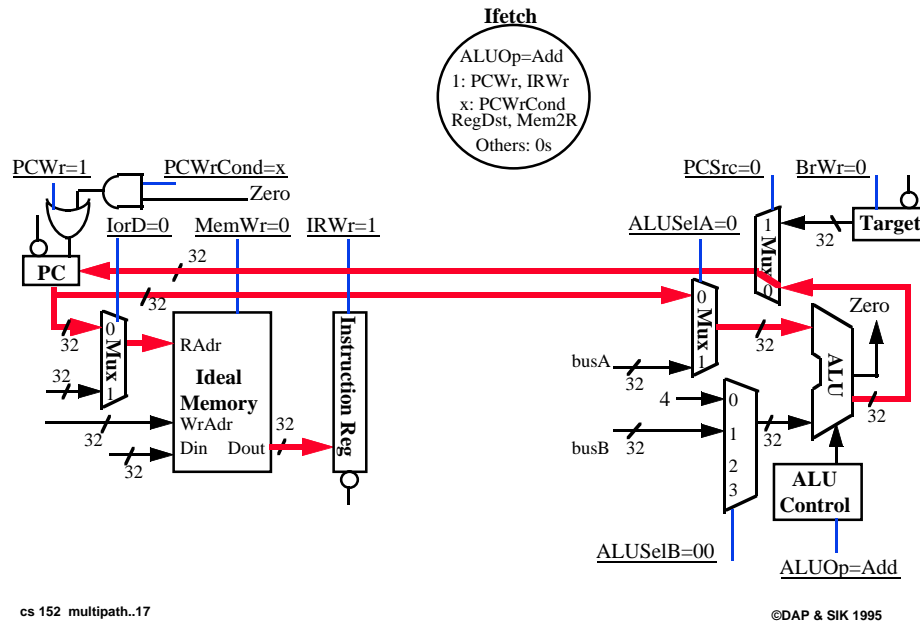
- $\text{IR} \leftarrow \text{mem}[\text{PC}] \quad \text{PC} < 31:0 > \leftarrow \text{PC} < 31:0 > + 4$



cs 152 multipath..16

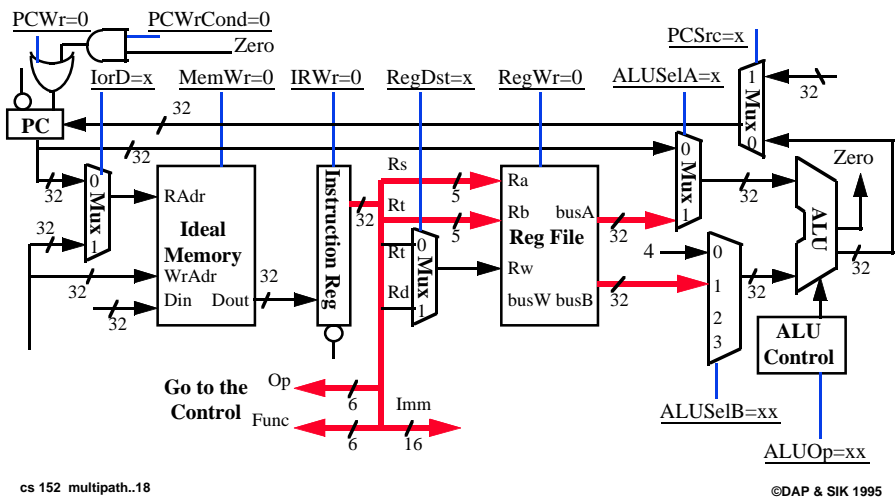
©DAP & SIK 1995

## Instruction Fetch Cycle: Overall Picture



## Register Fetch / Instruction Decode

- **busA <- RegFile[rs] ; busB <- RegFile[rt] ;**
- **ALU is not being used: ALUctr = xx**

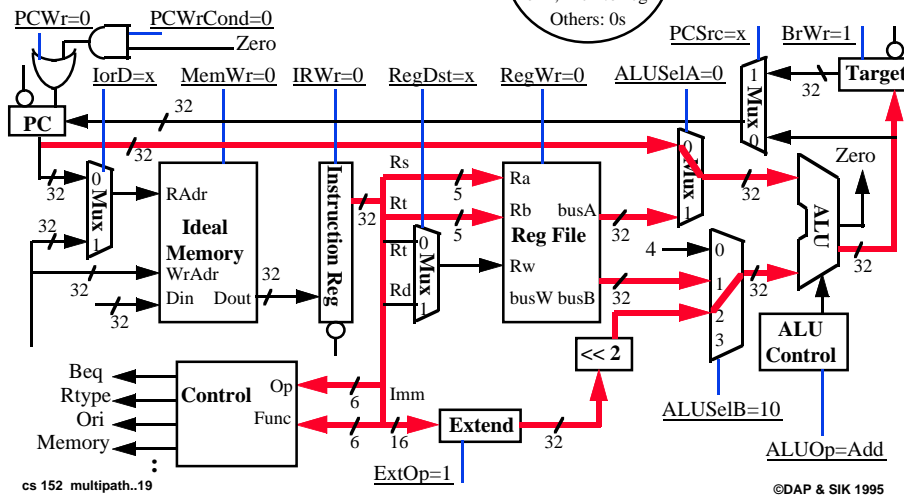


## Register Fetch / Instruction Decode (Continue)

- $\text{busA} \leftarrow \text{Reg}[\text{rs}] ; \text{busB} \leftarrow \text{Reg}[\text{rt}] ;$
- $\text{Target} \leftarrow \text{PC} + \text{SignExt}(\text{Imm16}) * 4$

### Rfetch/Decode

ALUOp=Add  
1: BrWr, ExtOp  
ALUSelB=10  
x: RegDst, PCSrc  
IorD, MemtoReg  
Others: 0s

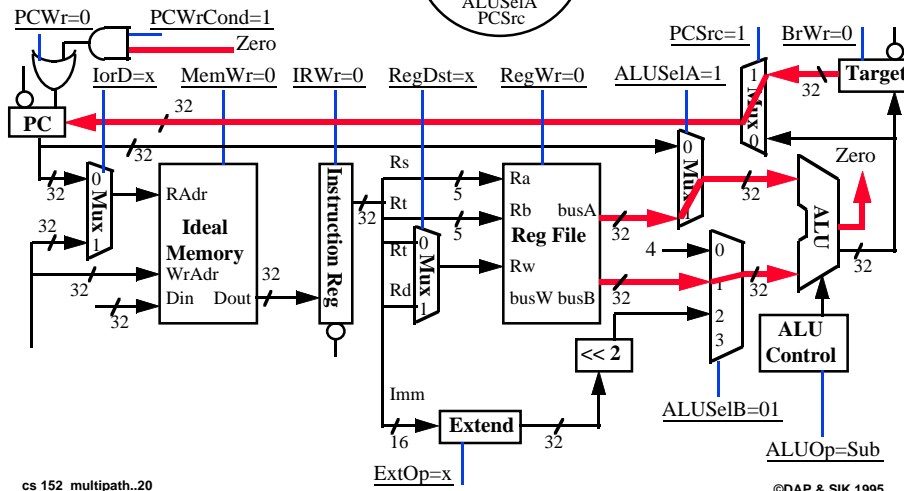


## Branch Completion

- if ( $\text{busA} == \text{busB}$ )
  - $\text{PC} \leftarrow \text{Target}$

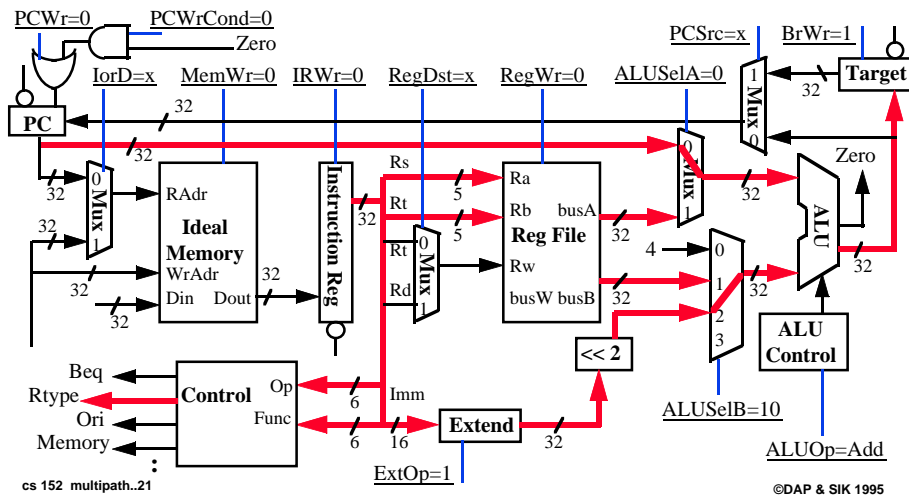
### BrComplete

ALUOp=Sub  
ALUSelB=01  
x: IorD, Mem2Reg  
RegDst, ExtOp  
1: PCWrCond  
ALUSelA  
PCSrc



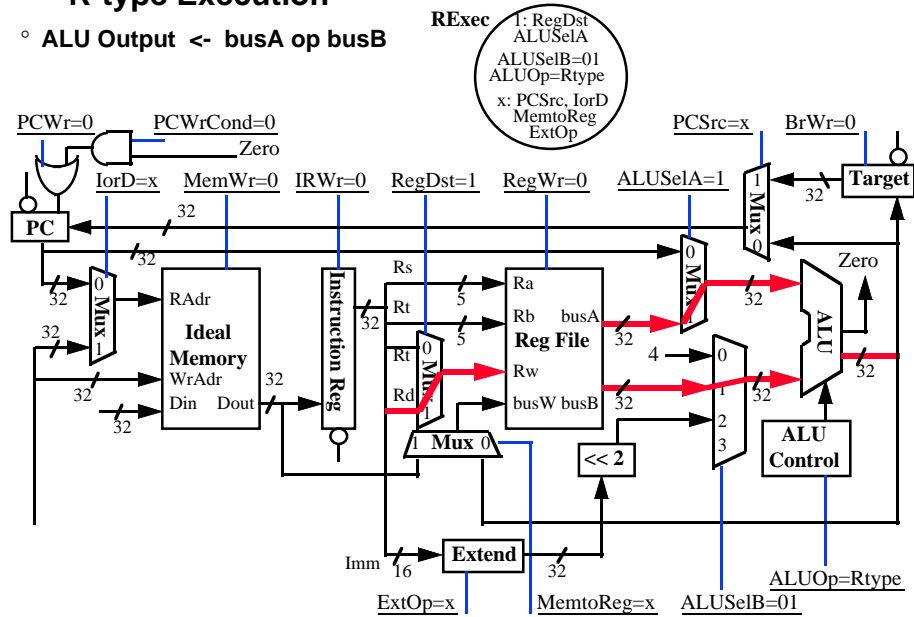
## Instruction Decode: We have a R-type!

- Next Cycle: R-type Execution



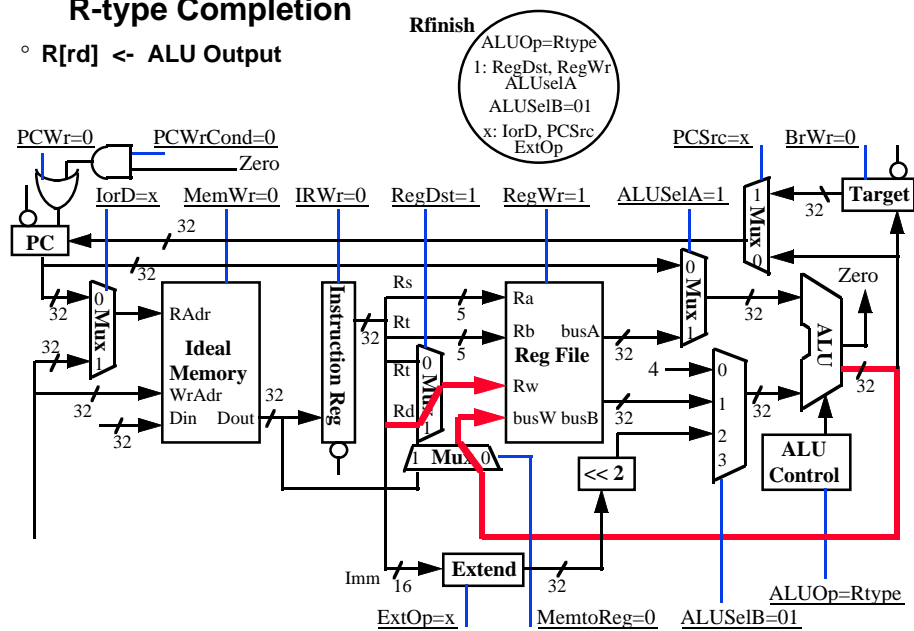
## R-type Execution

- ALU Output <- busA op busB



## R-type Completion

- $R[rd] \leftarrow \text{ALU Output}$

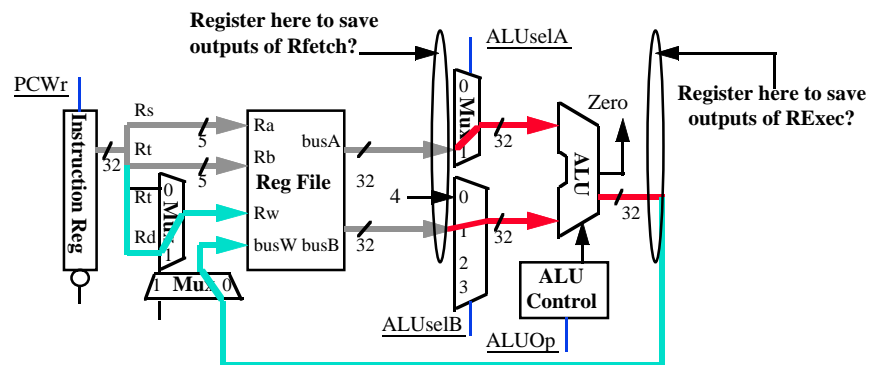


cs 152 multipath..23

©DAP & SIK 1995

## A Multiple Cycle Delay Path

- There is no register to save the results between:
  - Register Fetch:  $\text{busA} \leftarrow \text{Reg}[rs] ; \text{busB} \leftarrow \text{Reg}[rt]$  —
  - R-type Execution:  $\text{ALU output} \leftarrow \text{busA op busB}$  —
  - R-type Completion:  $\text{Reg}[rd] \leftarrow \text{ALU output}$  —



cs 152 multipath..24

©DAP & SIK 1995



## A Multiple Cycle Delay Path (Continue)

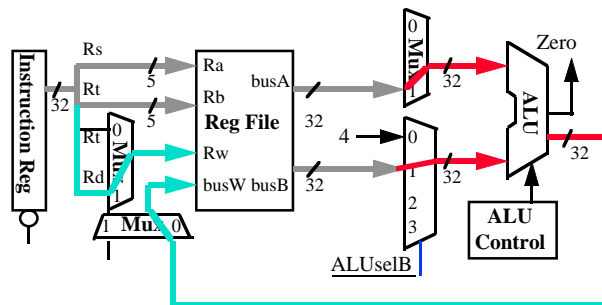
- Register is NOT needed to save the outputs of Register Fetch:
  - $IRWr = 0$ : busA and busB will not change after Register Fetch
- Register is NOT needed to save the outputs of R-type Execution:
  - busA and busB will not change after Register Fetch
  - Control signals  $ALUSelA$ ,  $ALUSelB$ , and  $ALUOp$  will not change after R-type Execution
  - Consequently ALU output will not change after R-type Execution
- In theory (P. 316, P&H), you need a register to hold a signal value if:
  - (1) The signal is computed in one clock cycle and used in another.
  - (2) AND the inputs to the functional block that computes this signal can change before the signal is written into a state element.
- You can save a register if Cond 1 is true BUT Cond 2 is false:
  - But in practice, this will introduce a multiple cycle delay path:
    - A logic delay path that takes multiple cycles to propagate from one storage element to the next storage element

cs 152 multipath..25

©DAP & SIK 1995

## Pros and Cons of a Multiple Cycle Delay Path

- A 3-cycle path example:
  - IR (storage) -> Reg File Read -> ALU -> Regr File Write (storage)
- Advantages:
  - Register savings
  - We can share time among cycles:
    - If ALU takes longer than one cycle, still “a OK” as long as the entire path takes less than 3 cycles to finish



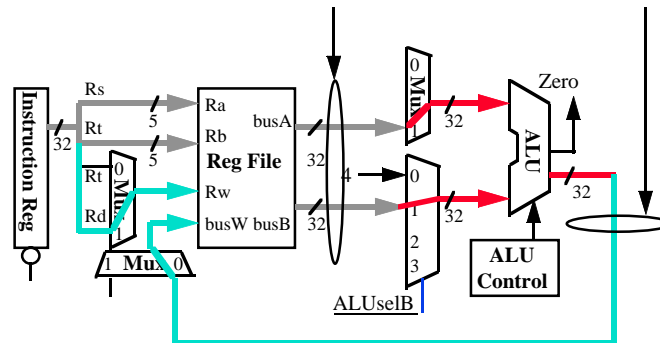
cs 152 multipath..26

©DAP & SIK 1995

## Pros and Cons of a Multiple Cycle Delay Path (Continue)

- Disadvantage:

- Static timing analyzer, which **ONLY** looks at delay between two storage elements, will report this as a timing violation
- You have to ignore the static timing analyzer's warnings
- But you may end up ignoring real timing violations
- I always TRY to put in registers between cycles to avoid MCP



cs 152 multipath..27

©DAP & SIK 1995

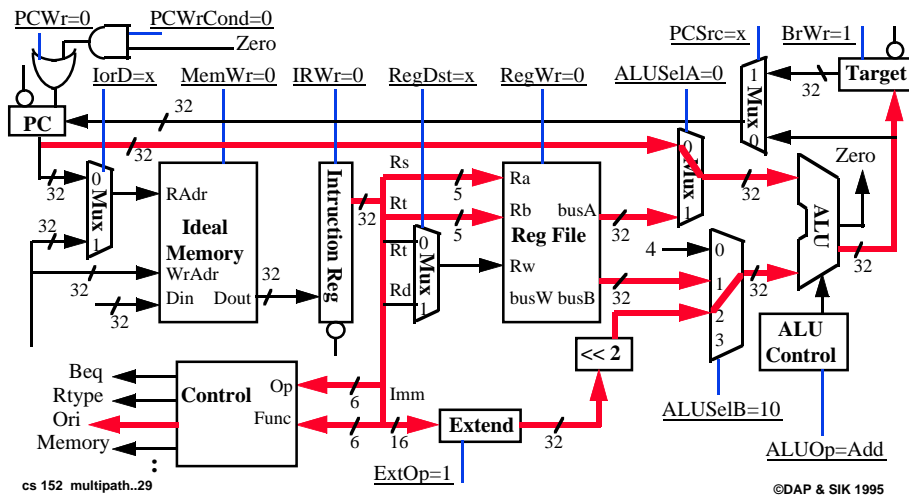
**Break (5 Minutes)**

cs 152 multipath..28

©DAP & SIK 1995

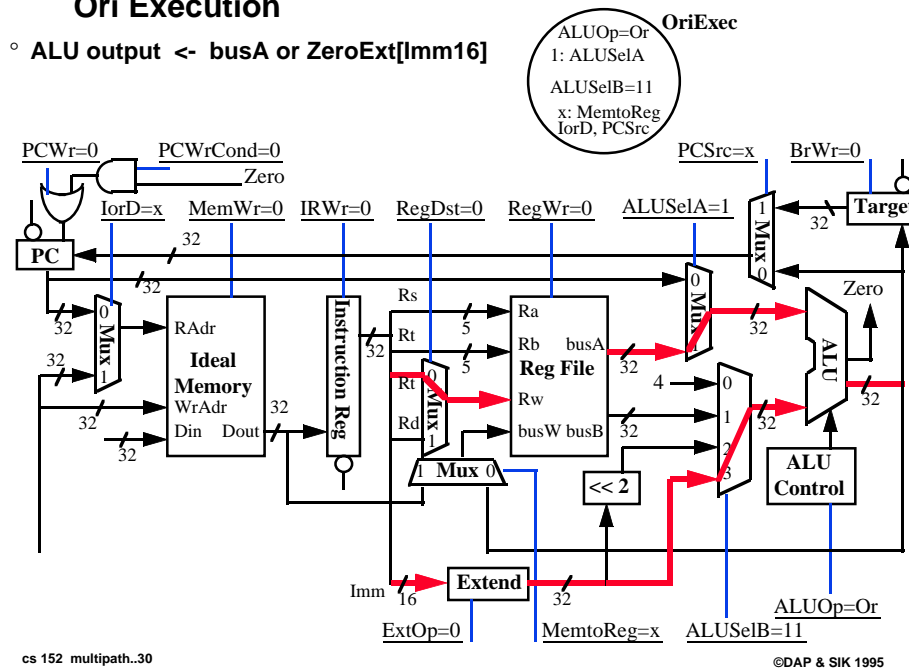
### Instruction Decode: We have an Ori!

- **Next Cycle: Ori Execution**



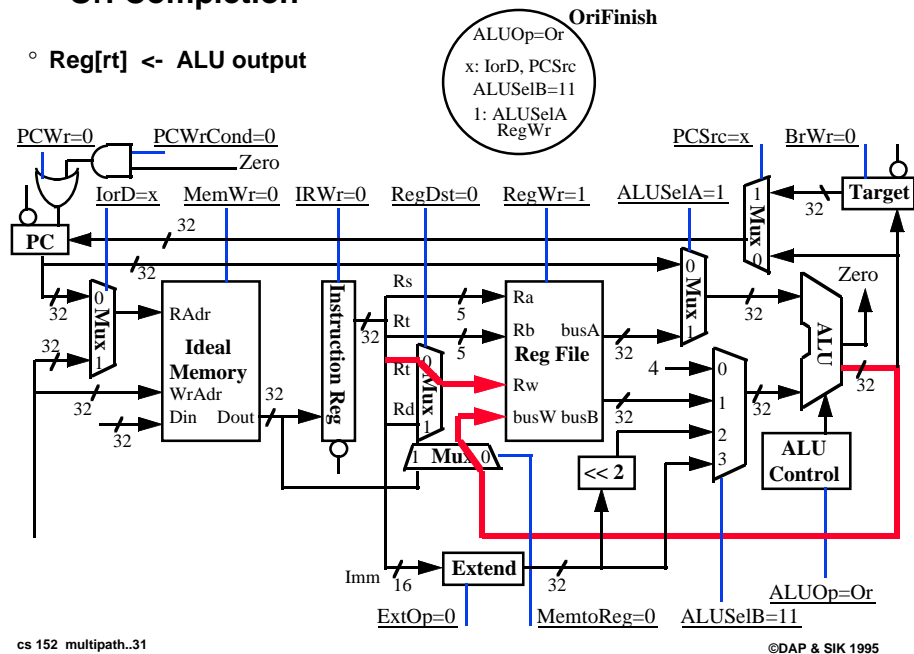
## Ori Execution

- **ALU output <- busA or ZeroExt[Imm16]**



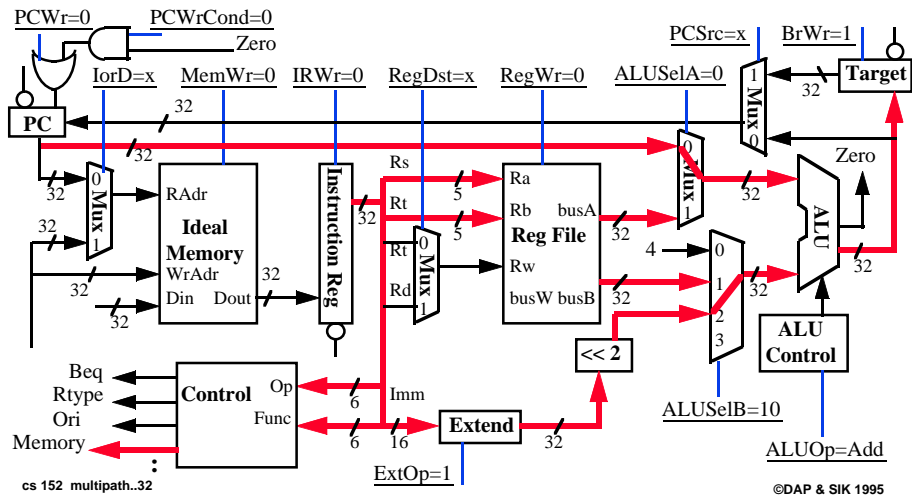
## Ori Completion

- $\text{Reg}[rt] \leftarrow \text{ALU output}$



## Instruction Decode: We have a Memory Access!

- Next Cycle: Memory Address Calculation



**AdrCal**

1: ExtOp  
ALUSelA  
ALUSelB=11  
ALUOp=Add  
x: MemtoReg  
PCSrc

[illegible]

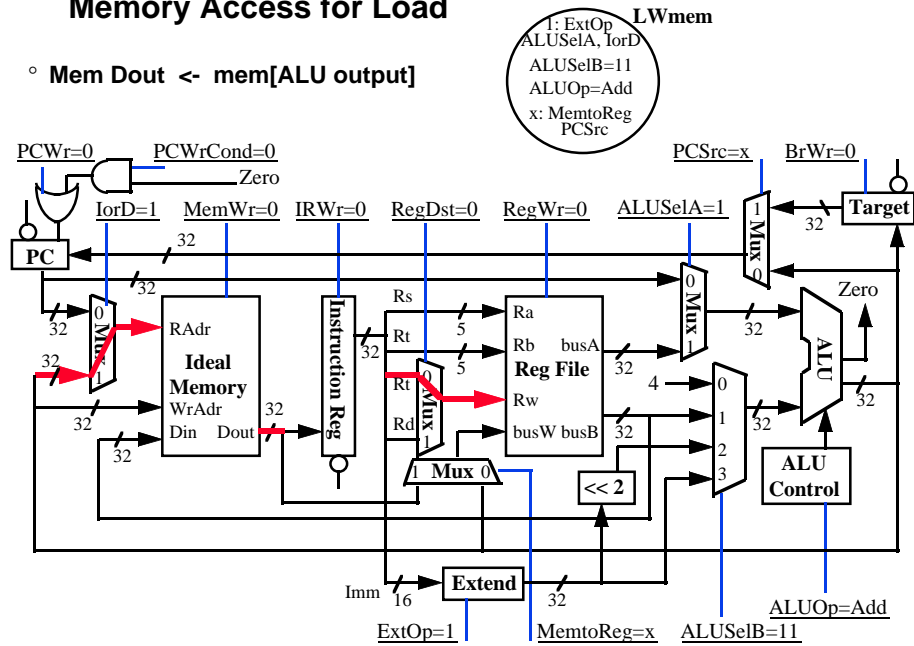
©DAP & SIK 1995

Figure 1: Block diagram of the RISC-V processor. The diagram illustrates the internal components and data flow of the processor. Key components include the PC (Program Counter), Ideal Memory, Instruction Register, Register File, ALU, and ALU Control. Data paths are labeled with widths (e.g., 32, 16, 5, 4). Control signals are shown at the top and bottom. A red path highlights the ALU input from the register file and the immediate extension.

©DAP & SIK 1995

## Memory Access for Load

◦ Mem Dout <- mem[ALU output]

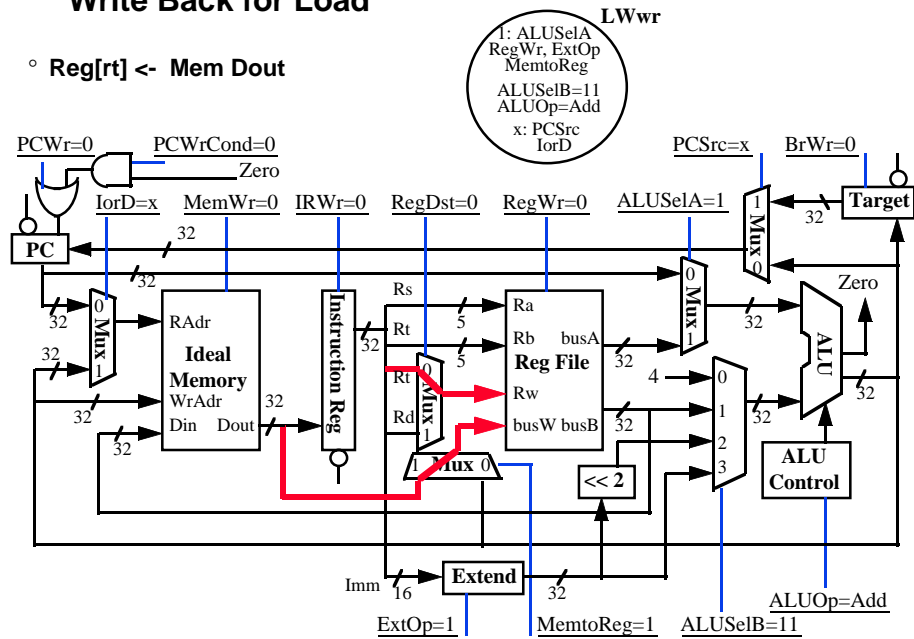


cs 152 multipath..35

©DAP & SIK 1995

## Write Back for Load

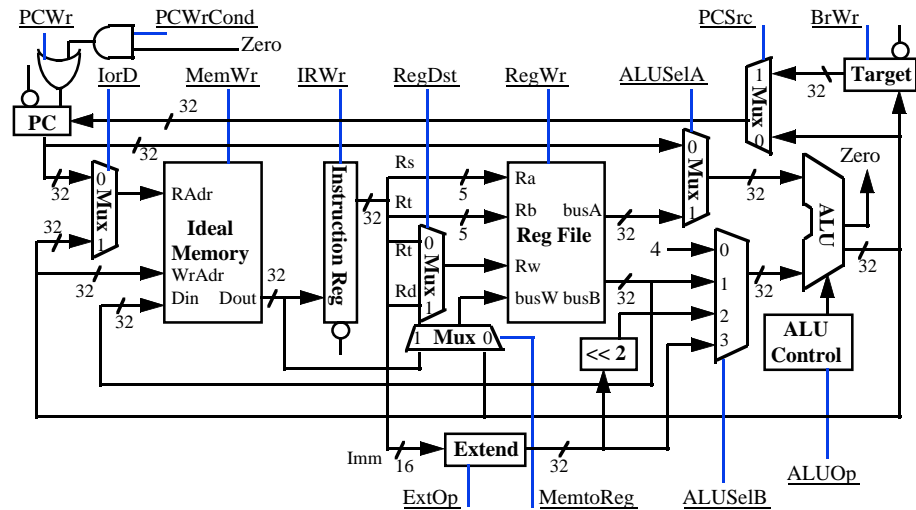
◦ Reg[rt] <- Mem Dout



cs 152 multipath..36

©DAP & SIK 1995

## Putting it all together: Multiple Cycle Datapath



cs 152 multipath..37

©DAP & SIK 1995

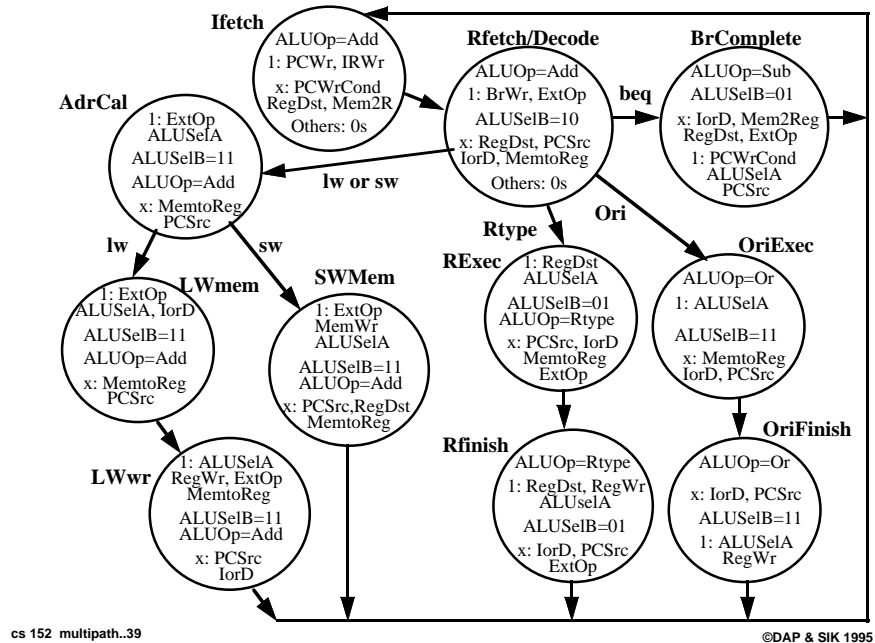
## Summary

- **Disadvantages of the Single Cycle Processor**
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- **Multiple Cycle Processor:**
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- **Do NOT confuse Multiple Cycle Processor with Multiple Cycle Delay Path**
  - Multiple Cycle Processor executes each instruction in multiple clock cycles
  - Multiple Cycle Delay Path: a combinational logic path between two storage elements that takes more than one clock cycle to complete
- **It is possible (desirable) to build a MC Processor without MCDP:**
  - Use a register to save a signal's value whenever a signal is generated in one clock cycle and used in another cycle later

cs 152 multipath..38

©DAP & SIK 1995

## Putting it all together: Control State Diagram



cs 152 multipath..39

©DAP & SIK 1995

## Where to get more information?

- **Next two lectures:**
  - **Multiple Cycle Controller:** Appendix C of your text book.
  - **Microprogramming:** Section 5.5 of your text book.
- **D. Patterson, "Microprograming," Scientific America, March 1983.**
- **D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," Computer Architecture News 8, 6 (October 15, 1980)**



**CS152**  
**Computer Architecture and Engineering**  
**Lecture 12: Designing a Multiple Cycle Controller**

March 1, 1995

Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)

Slides available on <http://http.cs.berkeley.edu/~patterson>

cs 152 multicontroller..1

©DAP & SIK 1995

## Review of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction

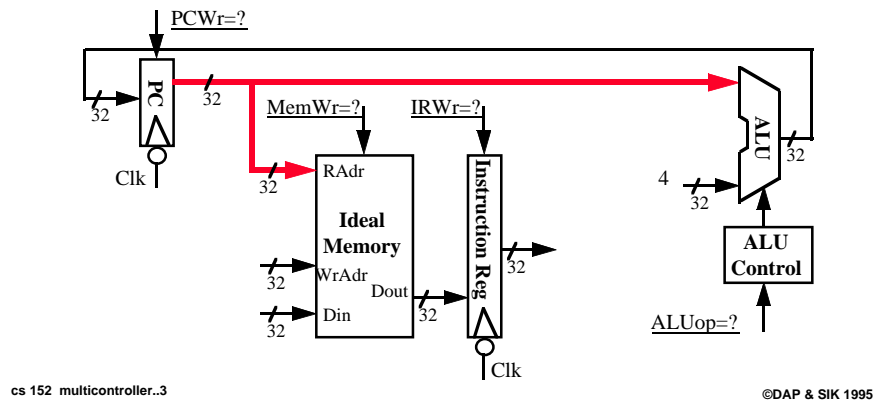
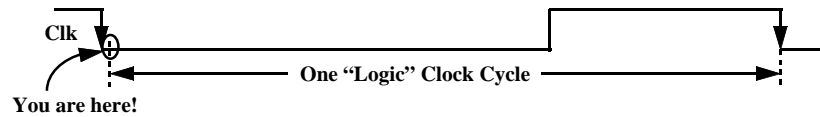
cs 152 multicontroller..2

©DAP & SIK 1995

## Review: Instruction Fetch Cycle, In the Beginning

- Every cycle begins right AFTER the clock tick:

- $\text{mem}[\text{PC}] \quad \text{PC} < 31:0 > + 4$



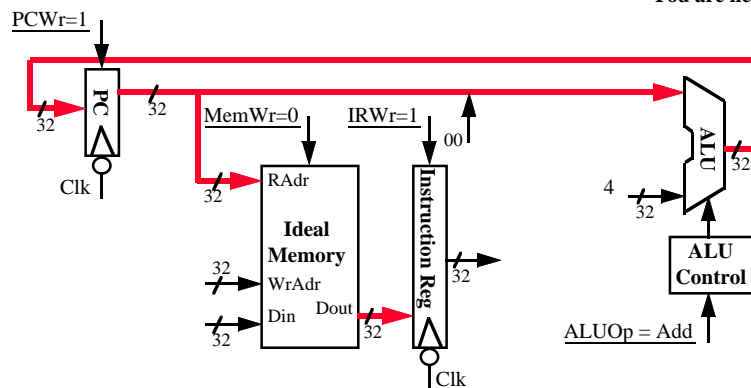
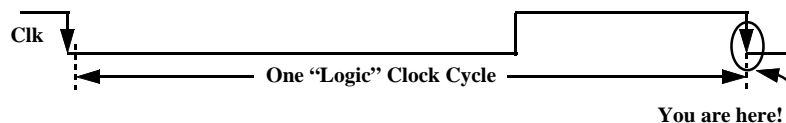
cs 152 multicontroller..3

©DAP & SIK 1995

## Review: Instruction Fetch Cycle, The End

- Every cycle ends AT the next clock tick (storage element updates):

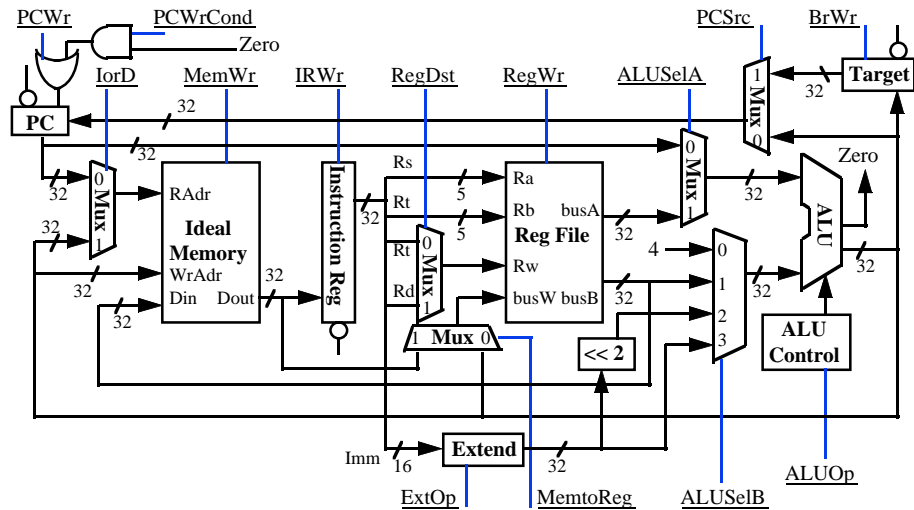
- $\text{IR} \leftarrow \text{mem}[\text{PC}] \quad \text{PC} < 31:0 > \leftarrow \text{PC} < 31:0 > + 4$



cs 152 multicontroller..4

©DAP & SIK 1995

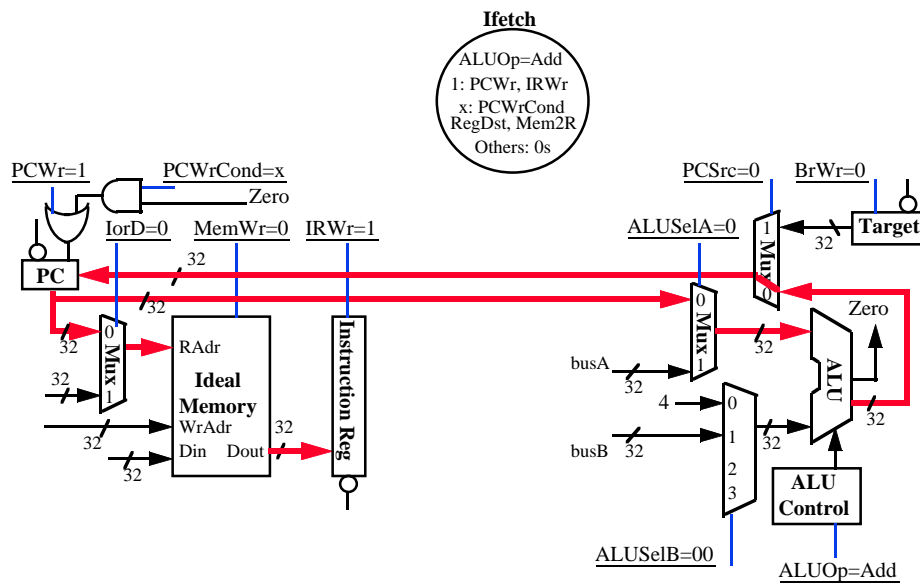
## Putting it all together: Multiple Cycle Datapath



cs 152 multicontroller..5

©DAP & SIK 1995

## Instruction Fetch Cycle: Overall Picture



cs 152 multicontroller..6

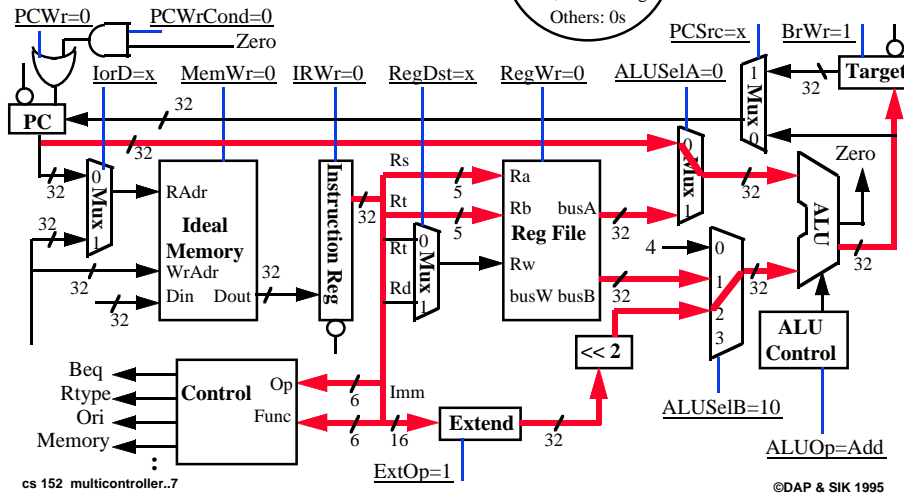
©DAP & SIK 1995

## Register Fetch / Instruction Decode (Continue)

- $\text{busA} \leftarrow \text{Reg}[\text{rs}] ; \text{busB} \leftarrow \text{Reg}[\text{rt}] ;$
- $\text{Target} \leftarrow \text{PC} + \text{SignExt}(\text{Imm16}) * 4$

**Rfetch/Decode**

ALUOp=Add  
1: BrWr, ExtOp  
ALUSelB=10  
x: RegDst, PCSrc  
IorD, MemtoReg  
Others: 0s

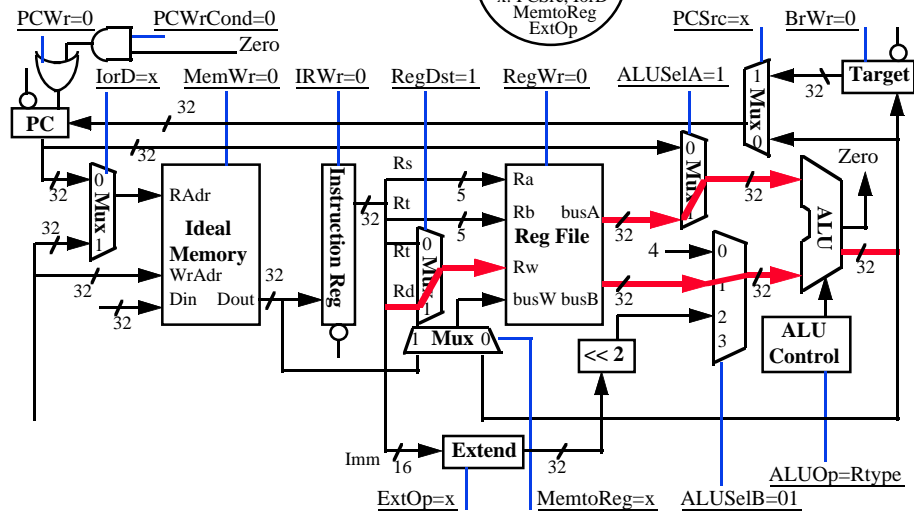


## R-type Execution

- $\text{ALU Output} \leftarrow \text{busA op busB}$

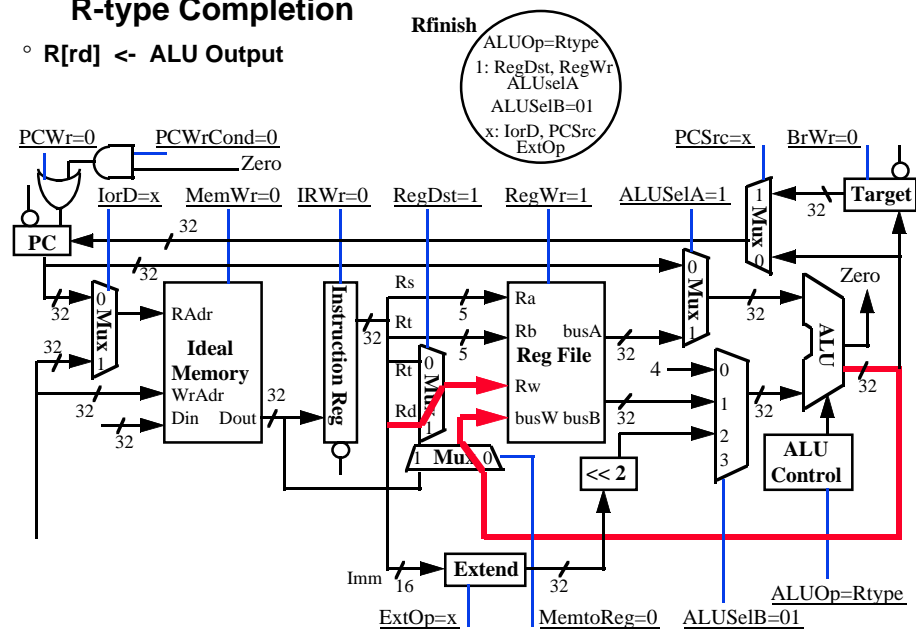
**RExec**

1: RegDst  
ALUSelA  
ALUSelB=01  
ALUOp=Rtype  
x: PCSrc, IorD  
MemtoReg  
ExtOp



## R-type Completion

- **R[rd] <- ALU Output**



cs 152 multicontroller..9

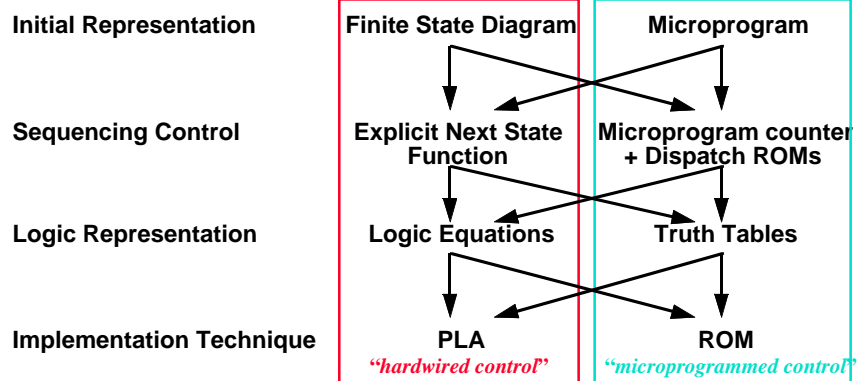
©DAP & SIK 1995

## Outline of Today's Lecture

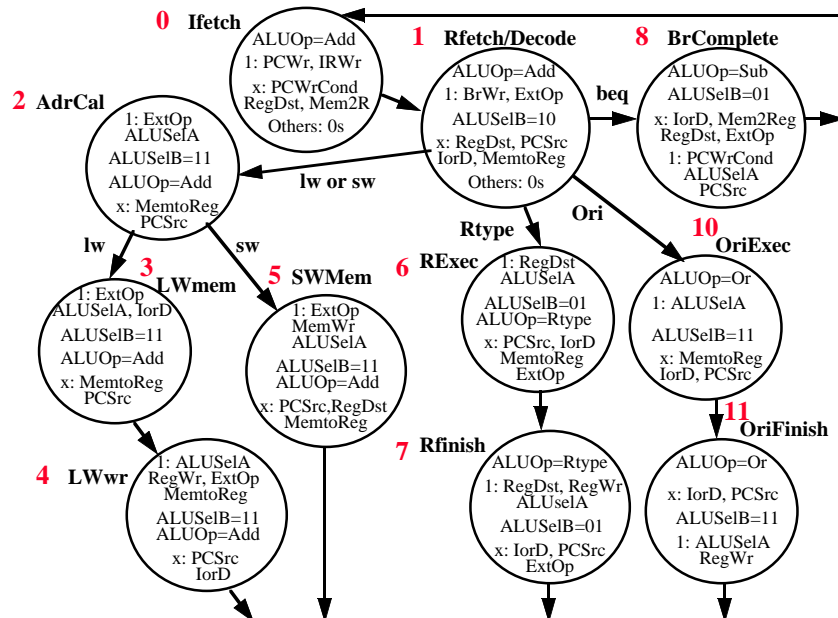
- **Recap (5 minutes)**
- **Review of FSM control (15 minutes)**
- **Questions and Administrative Matters (5 minutes)**
- **From Finite State Diagrams to Microprogramming (25 minutes)**
- **Break (5 minutes)**
- **ABCs of microprogramming (25 minutes)**

## Overview of Next Two Lectures

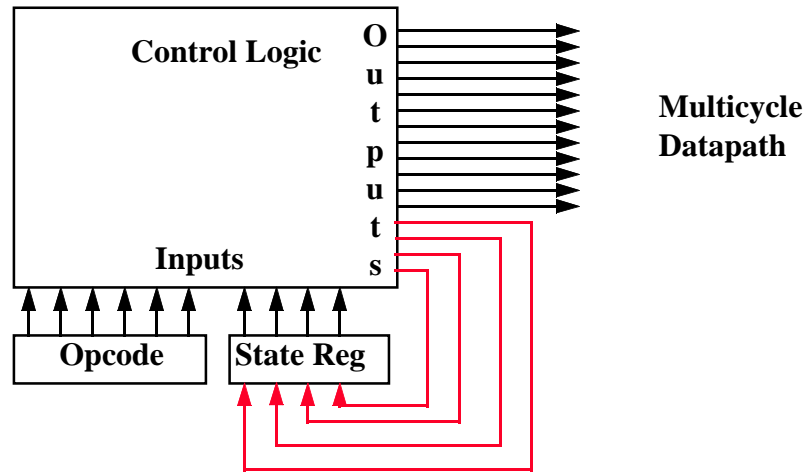
- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



## Initial Representation: Finite State Diagram



## Sequencing Control: Explicit Next State Function



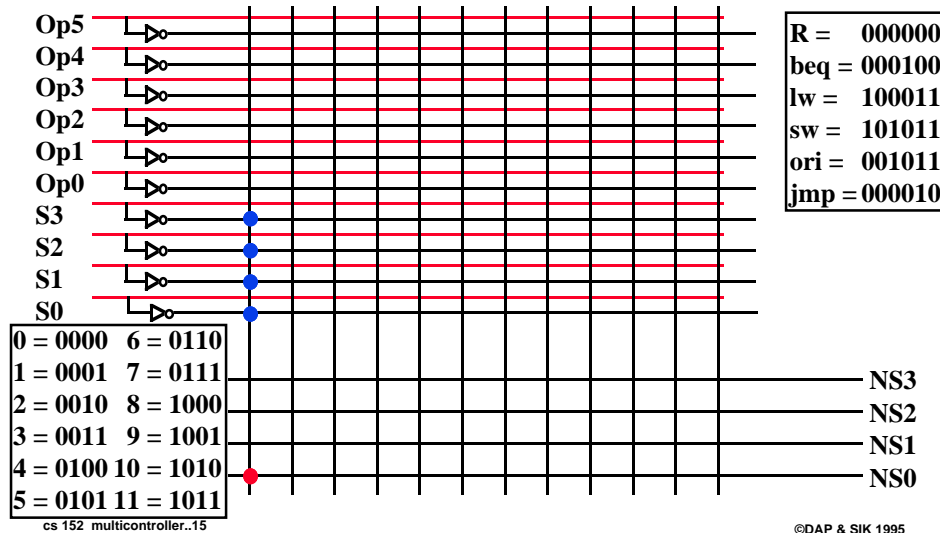
- Next state number is encoded just like datapath controls

## Logic Representative: Logic Equations

- Next state from current state
  - State 0 -> State1
  - State 1 -> S2, S6, S8, S10
  - State 2 -> \_\_\_\_\_
  - State 3 -> \_\_\_\_\_
  - State 4 -> State 0
  - State 5 -> State 0
  - State 6 -> State 7
  - State 7 -> State 0
  - State 8 -> State 0
  - State 9 -> State 0
  - State 10 -> State 11
  - State 11 -> State 0
- Alternatively, prior state & condition
  - S4, S5, S7, S8, S9, S11 -> State0
  - \_\_\_\_\_ -> State 1
  - \_\_\_\_\_ -> State 2
  - \_\_\_\_\_ -> State 3
  - \_\_\_\_\_ -> State 4
  - State2 & op = sw -> State 5
  - \_\_\_\_\_ -> State 6
  - State 6 -> State 7
  - \_\_\_\_\_ -> State 8
  - State2 & op = jmp -> State 9
  - \_\_\_\_\_ -> State 10
  - State 10 -> State 11

## Implementation Technique: Programmed Logic Arrays

- Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane



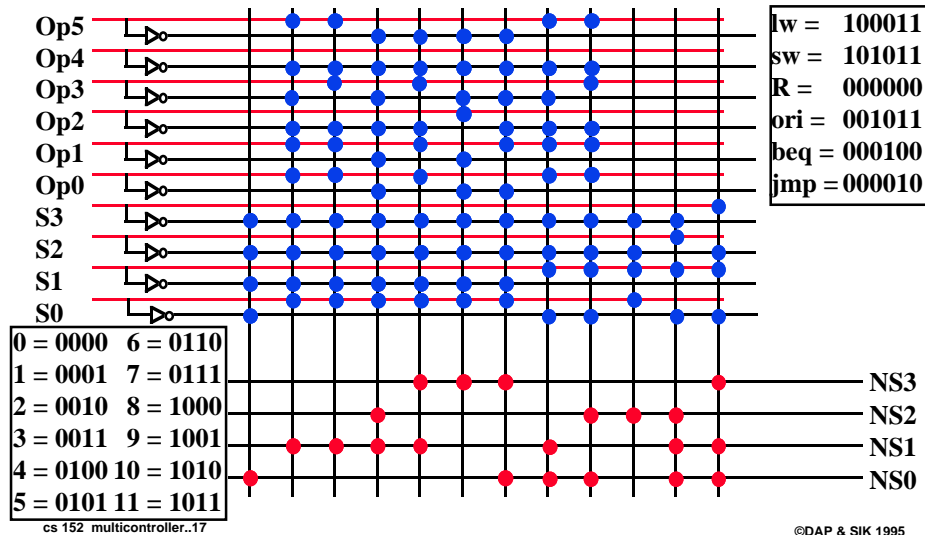
## Questions and Administrative Matters

- Apologize for problems with the franklin.cs
  - Lessons learned for your career?
- “Midterm” for instructors and TAs: constructive criticism by Friday
  - Please put your name, as I want to hear from everyone
  - If you want to submit an anonymous form, just take a second copy
  - Be careful what you wish for, it may come true
    - this semester we switched to the faster HP workstations (which is the cause of the instability) and doubled the number of Powerview licenses and disk space per group
  - Return in class Friday, right after 5 minute break
- Email progress reports 4PM Friday
- Assume reasonable delays for modules for next assignment
- Go to discussion section so that you can meet with your group!!!



## Implementation Technique: Programmed Logic Arrays

- Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane

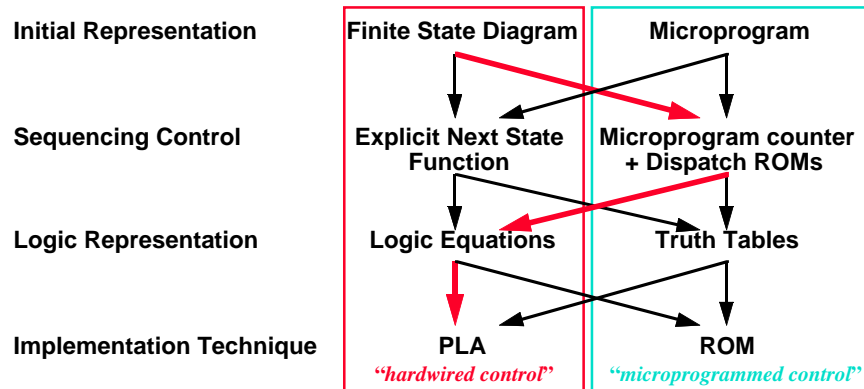


## Multicycle Control

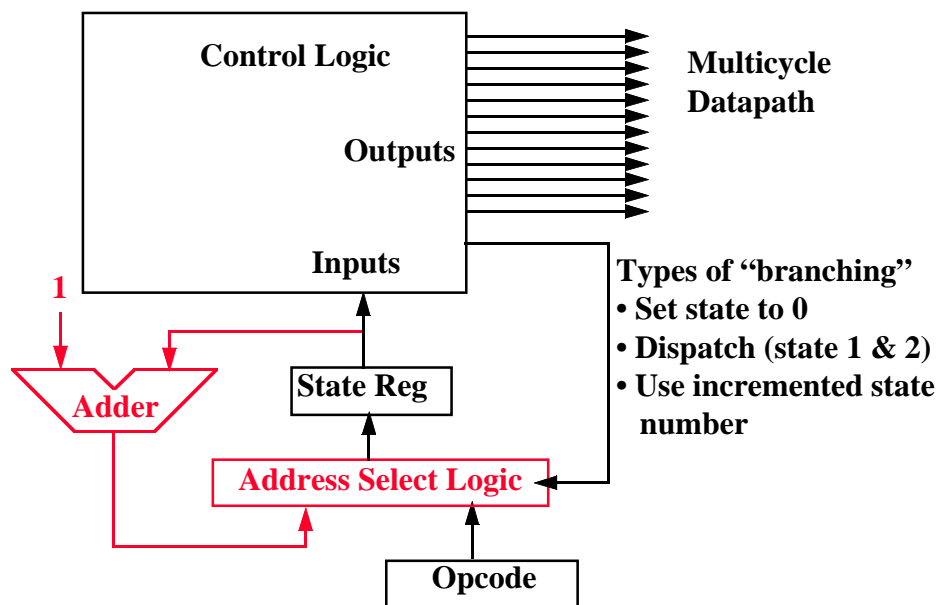
- Given numbers of FSM, can turn determine next state as function of inputs, including current state
- Turn these into Boolean equations for each bit of the next state lines
- Can implement easily using PLA
- What if many more states, many more conditions?
- What if need to add a state?

## Next Iteration: Using Sequencer for Next State

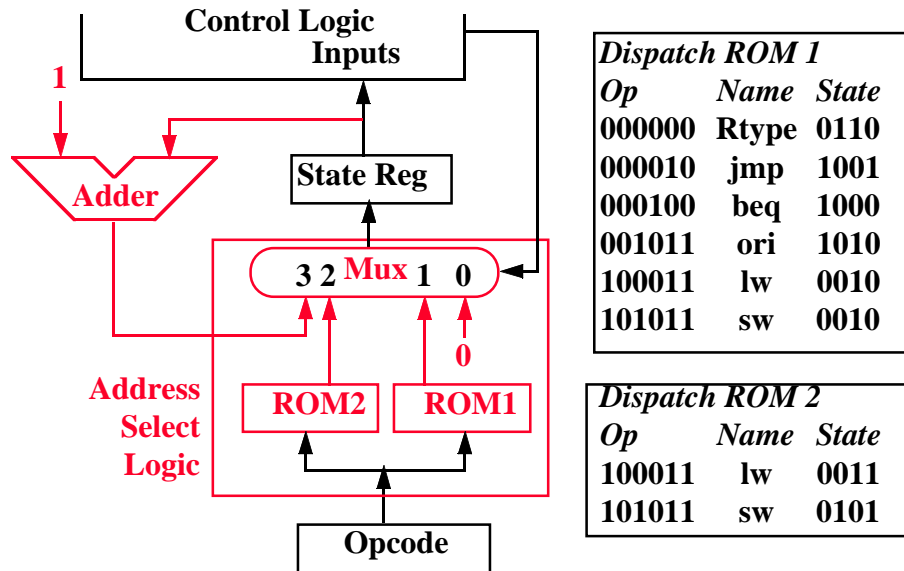
- Before Explicit Next State: Next try variation 1 step from right hand side
- Few sequential states in small FSM: suppose added floating point?
- Still need to go to non-sequential states: e.g., state 1 => 2, 6, 8, 10



## Sequencer-based control unit



## Sequencer-based control unit details



©DAP & SIK 1995

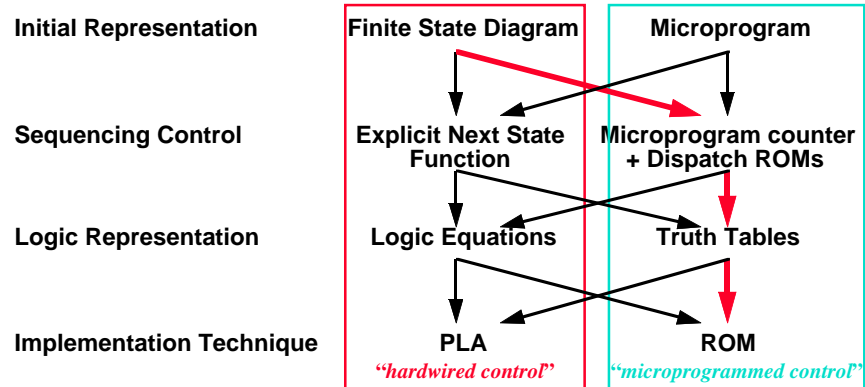
## Implementing Control with a ROM

- Instead of a PLA, use a ROM with one word per state (“Control word”)

<i>State number</i>	<i>Control Word Bits 18-2</i>	<i>Control Word Bits 1-0</i>
0	10010100000001000	11
1	00000000010011000	01
2	00000000000010100	10
3	00110000000010100	11
4	00110010000010110	00
5	00101000000010100	00
6	00000000001000100	11
7	00000000001000111	00
8	01000000100100100	00
9	10000001000000000	00
10	...	11
11	...	00

©DAP & SIK 1995

## Next Iteration: Using Microprogram for Representation



- ROM can be thought of as a sequence of control words
- Control word can be thought of as instruction: "microinstruction"
- Rather than program in binary, use assembly language

**Break (5 Minutes)**

## Microprogramming

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global

### Microprogramming:

- A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

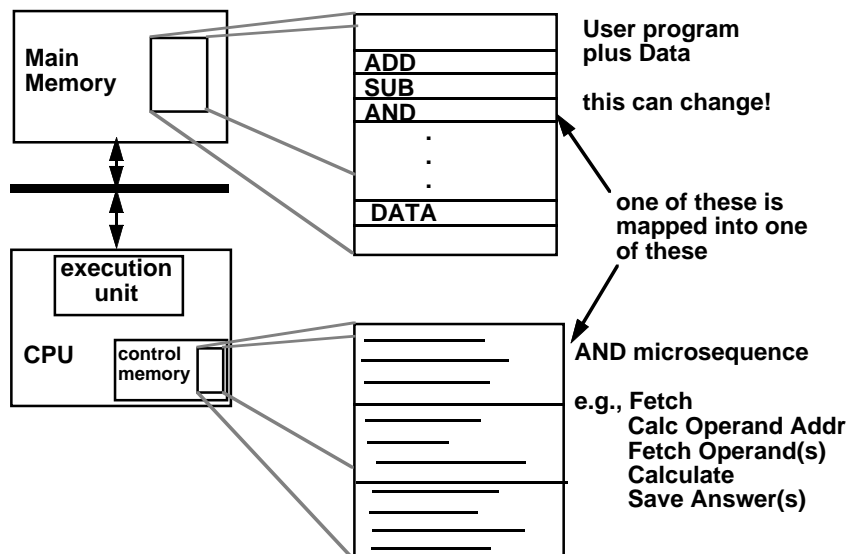
### Microarchitecture:

- Logical structure and functional capabilities of the hardware as seen by the microprogrammer

### Historical Note:

IBM 360 Series first to distinguish between architecture & organization  
Same instruction set across wide range of implementations, each with different cost/performance

### Macroinstruction Interpretation



## Variations on Microprogramming

- **Horizontal Microcode**

- control field for each control point in the machine



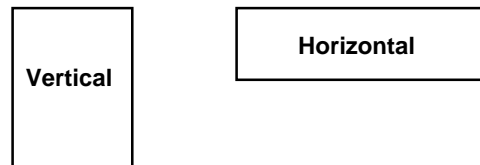
- **Vertical Microcode**

- compact microinstruction format for each class of microoperation

**branch:**  $\mu\text{seq-op}$   $\mu\text{add}$

**execute:**      **ALU-op A,B,R**

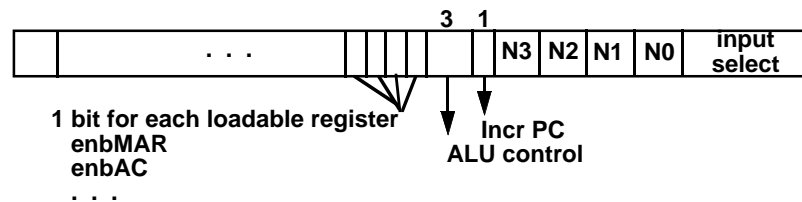
**memory:** mem-op S, D



cs 152 multicontroller..27

©DAP & SIK 1995

### Extreme Horizontal



**Depending on bus organization, many potential control combinations simply not wrong, i.e., implies transfers that can never happen at the same time.**

**Makes sense to encode fields to save ROM space**

**Example: gate Rx and gate Ry to same bus should never happen  
encoded in single bit which is decoded rather than two separate bits**

**NOTE:** encoding should be just sufficient that parallel actions that the datapath supports should still be specifiable in a single microinstruction

cs 152 multicontroller..28

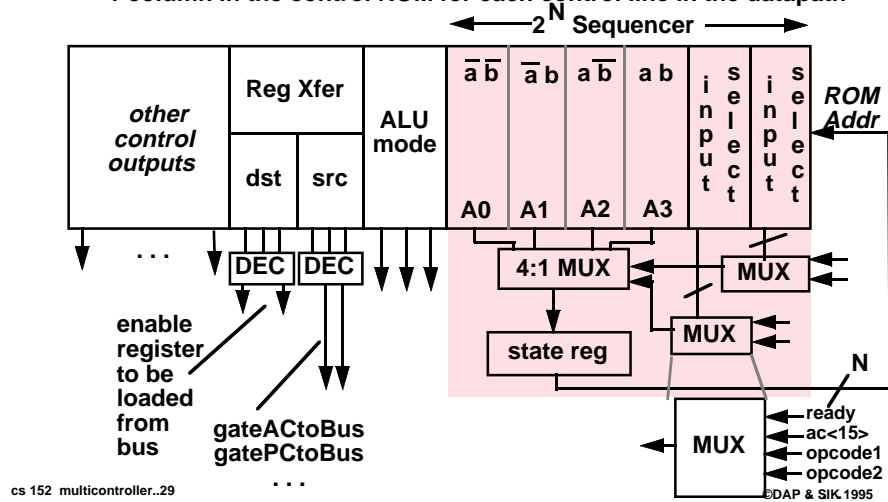
©DAP & SIK 1995

### Microprogramming (example)

- use of ROM/RAM to generate control points rather than through discrete logic
- including control of the next-state logic (the microsequencer)

### Horizontal Microprogramming

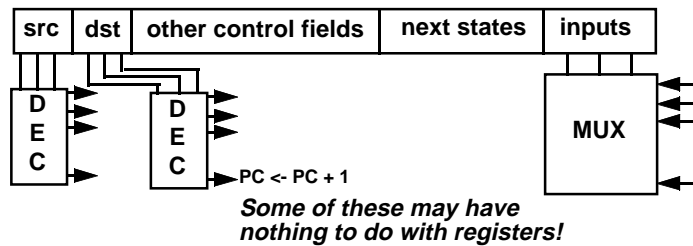
1 column in the control ROM for each control line in the datapath



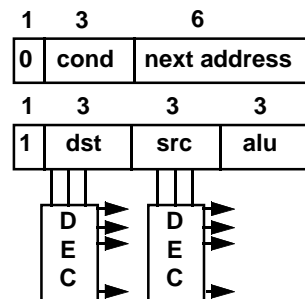
cs 152 multicontroller..29

©DAP & SIK 1995

### More Vertical Format



### Multiformat Microcode:



cs 152 multicontroller..30

©DAP & SIK 1995

[illegible]

### Not all critical control information is derived from control logic

enable signals from control

R 1 R 2 R D

D E C D E C D E C

Register File

IR op rs1 rs2 rd

to control



## Horizontal vs. Vertical Microprogramming

**NOTE:** previous organization is not TRUE horizontal microprogramming;  
register decoders give flavor of *encoded* microoperations

Most microprogramming-based controllers vary between:

*horizontal* organization (1 control bit per control point)

*vertical* organization (fields encoded in the control memory and must be decoded to control something)

### Horizontal

- + more control over the potential parallelism of operations in the datapath
- uses up lots of control store

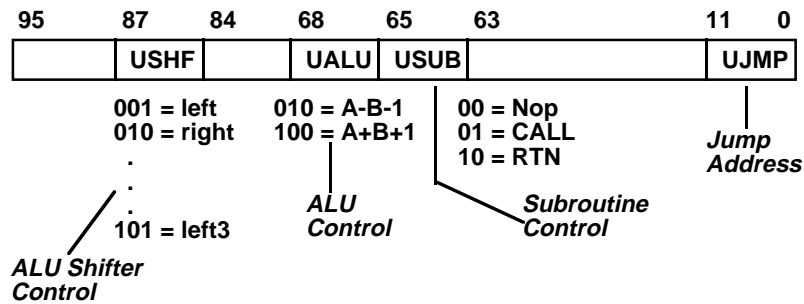
### Vertical

- + easier to program, not very different from programming a RISC machine in assembly language
- extra level of decoding may slow the machine down

## Vax Microinstructions

VAX Microarchitecture:

96 bit control store, 30 fields, 4096  $\mu$ instructions for VAX ISA  
encodes concurrently executable "microoperations"



## Legacy Software and Microprogramming

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
- If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn't multiple microprograms do multiple instruction sets on the same microarchitecture?
- Coined term "emulation": instruction set interpreter in microcode for non-native instruction set
- Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used

## Microprogramming Pros and Cons

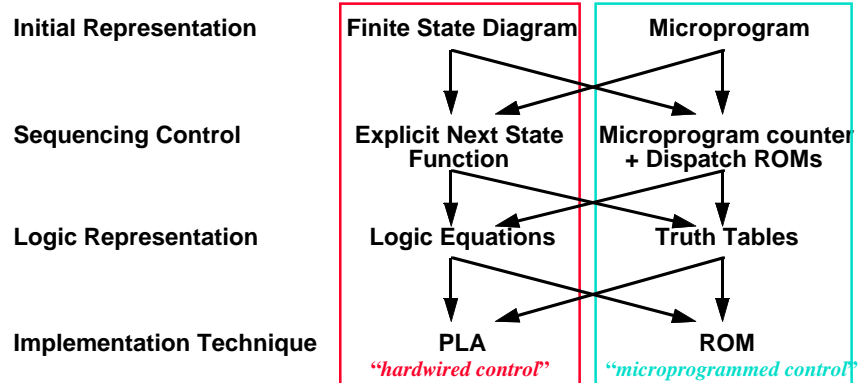
- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

## Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate...
- If you could even write compilers to produce microinstructions...
- If most programs use simple instructions and addressing modes...
- If microcode is kept in RAM instead of ROM so as to fix bugs ...
- If same memory used for control memory could be used instead as cache for “macroinstructions”...
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine?

## Summary: Multicycle Control

- Microprogramming and hardwired control have many similarities, perhaps biggest difference is initial representation and ease of change of implementation, with ROM generally being easier than PLA



# **CS152**

## **Computer Architecture and Engineering**

### **Lecture 13: Microprogramming and Exceptions**

**March 3, 1995**

**Dave Patterson (patterson@cs) and  
Shing Kong (shing.kong@eng.sun.com)**

**Slides available on <http://http.cs.berkeley.edu/~patterson>**

cs 152  $\mu$ prog..1

©DAP & SIK 1995

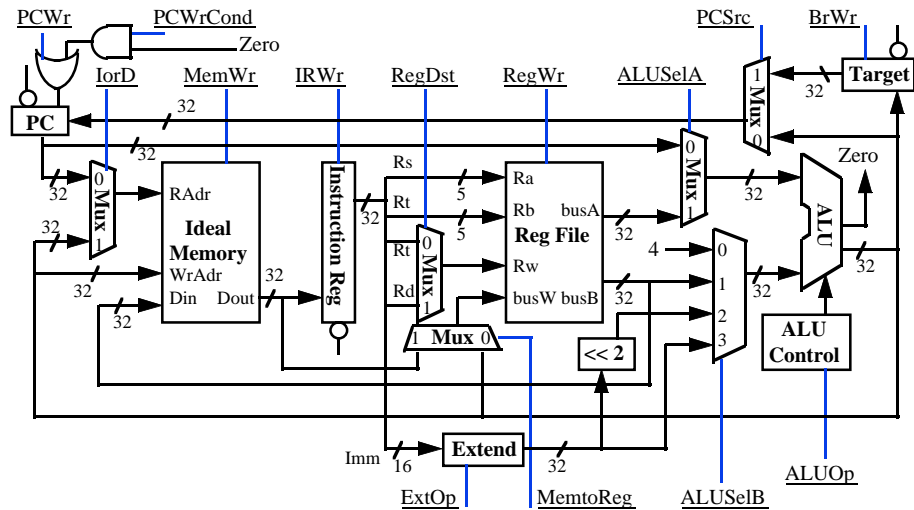
## **Review of a Multiple Cycle Implementation**

- **The root of the single cycle processor's problems:**
  - **The cycle time has to be long enough for the slowest instruction**
- **Solution:**
  - **Break the instruction into smaller steps**
  - **Execute each step (instead of the entire instruction) in one cycle**
    - **Cycle time: time it takes to execute the longest step**
    - **Keep all the steps to have similar length**
  - **This is the essence of the multiple cycle processor**
- **The advantages of the multiple cycle processor:**
  - **Cycle time is much shorter**
  - **Different instructions take different number of cycles to complete**
    - **Load takes five cycles**
    - **Jump only takes three cycles**
  - **Allows a functional unit to be used more than once per instruction**

cs 152  $\mu$ prog..2

©DAP & SIK 1995

## Review: Multiple Cycle Datapath



cs 152 μprog..3

©DAP & SIK 1995

## Overview of the Two Lectures

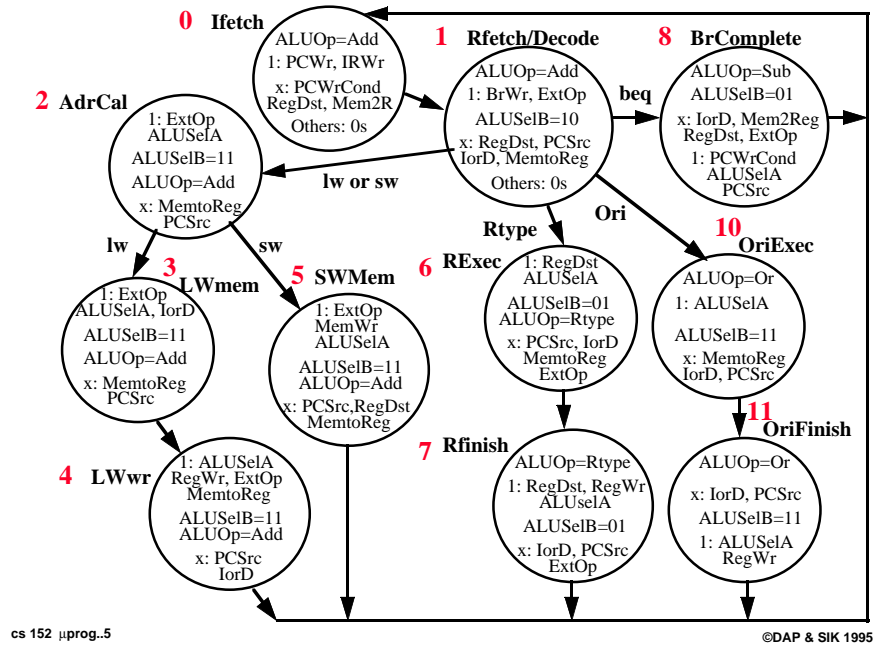
- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.

	Finite State Diagram	Microprogram
Initial Representation		
Sequencing Control	Explicit Next State Function	Microprogram counter + Dispatch ROMs
Logic Representation	Logic Equations	Truth Tables
Implementation Technique	PLA <i>"hardwired control"</i>	ROM <i>"microprogrammed control"</i>

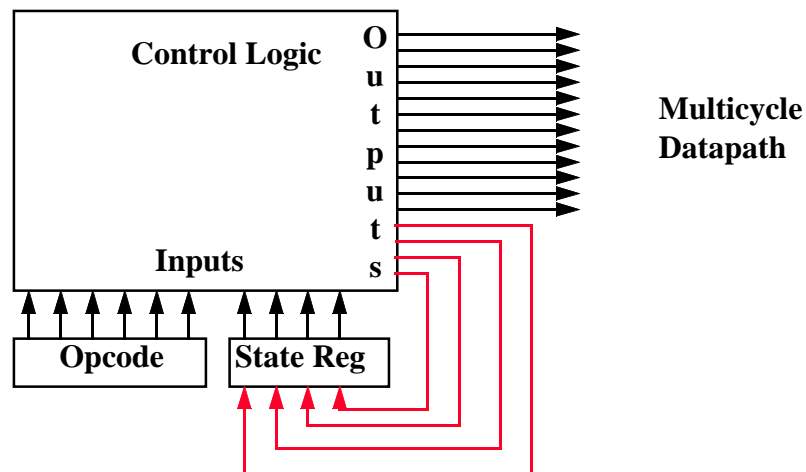
cs 152 μprog..4

©DAP & SIK 1995

## Initial Representation: Finite State Diagram



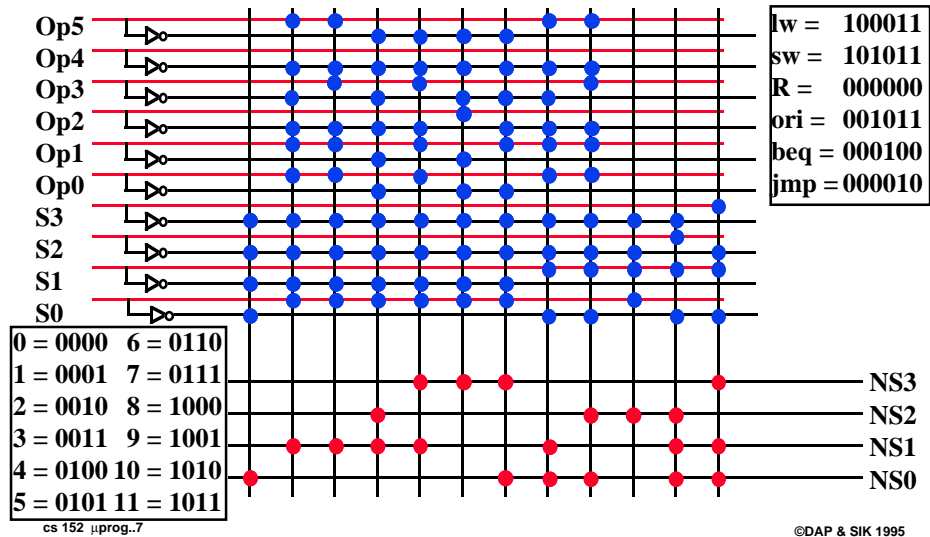
## Sequencing Control: Explicit Next State Function



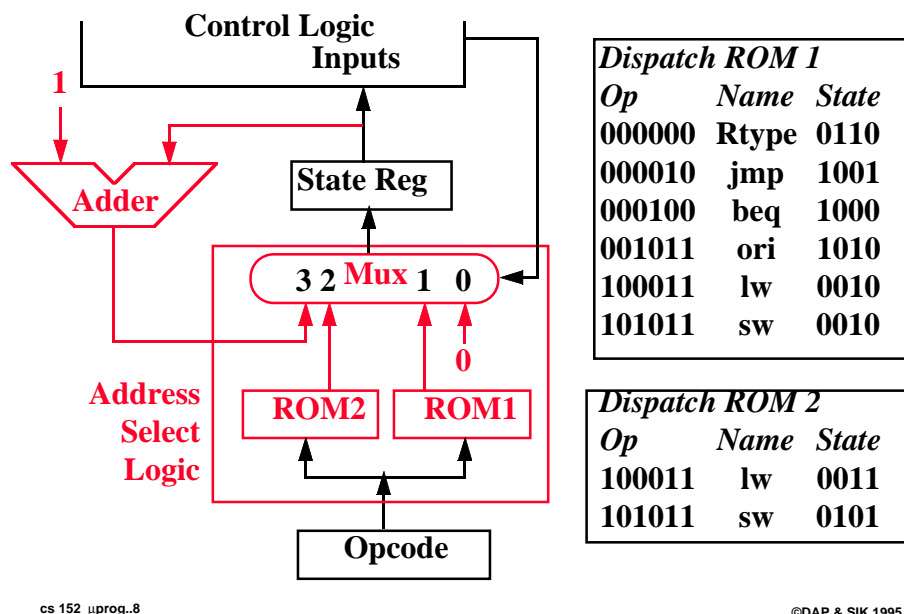
- Next state number is encoded just like datapath controls

## Implementation Technique: Programmed Logic Arrays

- Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane



## Sequencer-based control unit details

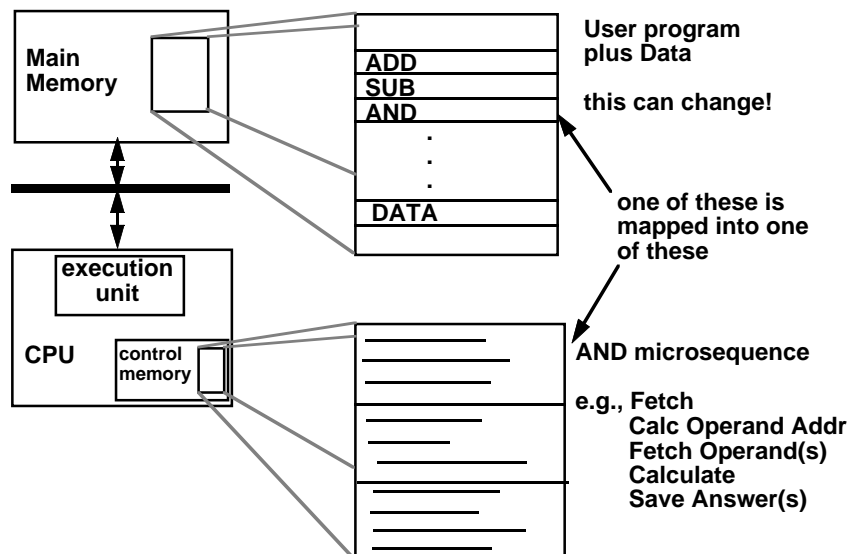


## Implementing Control with a ROM

- Instead of a PLA, use a ROM with one word per state (“Control word”)

State number	Control Word Bits 18-2	Control Word Bits 1-0
0	10010100000001000	11
1	00000000010011000	01
2	00000000000010100	10
3	00110000000010100	11
4	00110010000010110	00
5	00101000000010100	00
6	00000000001000100	11
7	00000000001000111	00
8	01000000100100100	00
9	10000001000000000	00
10	...	11
11	...	00

## Macroinstruction Interpretation





## Variations on Microprogramming

- **Horizontal Microcode**

- control field for each control point in the machine

$\mu$ seq	$\mu$ addr	A-mux	B-mux	bus enables	register enables	
-----------	------------	-------	-------	-------------	------------------	--

- **Vertical Microcode**

- compact microinstruction format for each class of microoperation

branch:         $\mu$ seq-op  $\mu$ add

execute:       ALU-op A,B,R

memory:       mem-op S, D

## Microprogramming Pros and Cons

- **Ease of design**
- **Flexibility**
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- **Can implement very powerful instruction sets (just more control memory)**
- **Generality**
  - Can implement multiple instruction sets on same machine. (Emulation)
  - Can tailor instruction set to application.
- **Compatibility**
  - Many organizations, same instruction set
- **Costly to implement**
- **Slow**

## Outline of Today's Lecture

- **Recap (5 minutes)**
- **Microinstruction Format Example (15 minutes)**
- **Questions and Administrative Matters (5 minutes)**
- **Do-it-yourself Microprogramming (25 minutes)**
- **Break (5 minutes)**
- **Exceptions (25 minutes)**

## Designing a Microinstruction Set

- **Start with list of control signals**
- **Group signals together that make sense: called “fields”**
- **Places fields in some logical order (ALU operation & ALU operands first and microinstruction sequencing last)**
- **Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals**
- **To minimize the width, encode operations that will never be used at the same time**

## Start with list of control signals, grouped into fields

<u>Signal name</u>	<u>Effect when deasserted</u>	<u>Effect when asserted</u>
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
TargetWrite	None	Target reg. = ALU
MemRead	None	Memory at address is read
MemWrite	None	Memory at address is written
lorD	Memory address = PC	Memory address = ALU
IRWrite	None	IR = Memory
PCWrite	None	PC = PCSource
PCWriteCond	None	IF ALUzero then PC = PCSource

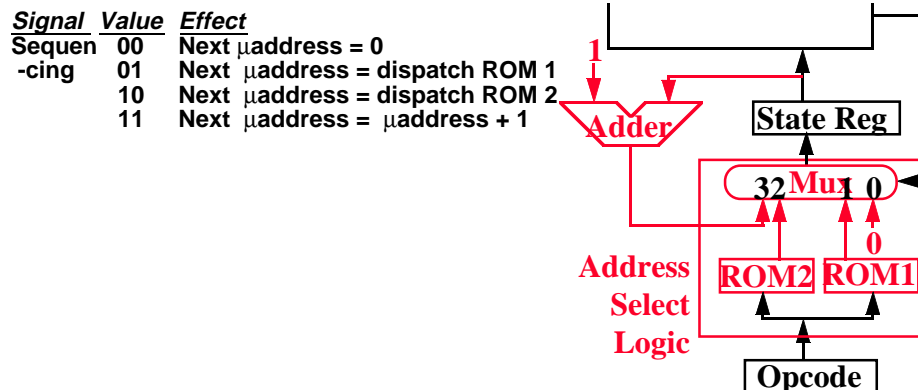
<u>Signal name</u>	<u>Value</u>	<u>Effect</u>
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU does function code
	11	ALU does logical OR
ALUSelB	000	2nd ALU input = Reg[rt]
	001	2nd ALU input = 4
	010	2nd ALU input = sign extended IR[15-0]
	011	2nd ALU input = sign extended, shift left 2 IR[15-0]
	100	2nd ALU input = zero extended IR[15-0]
PCSource	00	PC = ALU
	01	PC = Target
	10	PC = PC+4[29-26] : IR[25-0] << 2

cs 152 µprog..15

©DAP & SIK 1995

## Start with list of control signals, cont'd

- For next state function (next microinstruction address), use Sequencer-based control unit from last lecture



cs 152 µprog..16

©DAP & SIK 1995

## Microinstruction Format

<u>Field Name</u>	<u>Width</u>	<u>Control Signals Set</u>
ALU Control	2	ALUOp
SRC1	1	ALUSelA
SRC2	3	ALUSelB
ALU Destination	4	RegWrite, MemtoReg, RegDst, TargetWrite
Memory	3	MemRead, MemWrite, IorD
Memory Register	1	IRWrite
PCWrite Control	4	PCWrite, PCWriteCond, PCSource
Sequencing	2	AddrCtl
Total	20	

cs 152 µprog..17

©DAP & SIK 1995

## Legend of Fields and Symbolic Names

<u>Field Name</u>	<u>Values for Field</u>	<u>Function of Field with Specific Value</u>
ALU	Add	ALU adds
	Func code	ALU subtracts
	Subt.	ALU does function code
	Or	ALU does logical OR
SRC1	PC	1st ALU input = PC
	rs	1st ALU input = Reg[rs]
SRC2	4	2nd ALU input = 4
	Extend	2nd ALU input = sign ext. IR[15-0]
	Extend0	2nd ALU input = zero ext. IR[15-0]
	Extshft	2nd ALU input = sign ex., sl IR[15-0]
	rt	2nd ALU input = Reg[rt]
ALU destination	Target	Target = ALU
	rd	Reg[rd] = ALU
Memory	Read PC	Read memory using PC
	Read ALU	Read memory using ALU output
	Write ALU	Write memory using ALU output
Memory register	IR	IR = Mem
	Write rt	Reg[rt] = Mem
	Read rt	Mem = Reg[rt]
PC write	ALU	PC = ALU output
	Target-cond.	IF ALU Zero then PC = Target
Sequencing	jump addr.	PC = PCSource
	Seq	Go to sequential µinstruction
	Fetch	Go to the first microinstruction
	Dispatch i	Dispatch using ROMi (1 or 2)

cs 152 µprog..18

©DAP & SIK 1995

## Microprogram it yourself!

<u>Label</u>	<u>ALU</u>	<u>SRC1</u>	<u>SRC2</u>	<u>ALU Dest.</u>	<u>Memory</u>	<u>Mem. Req.</u>	<u>PC Write</u>	<u>Sequencing</u>
Fetch	Add	PC	4		Read PC	IR	ALU	Seq

## Questions and Administrative Matters

- “Midterm” for instructors and TAs: constructive criticism by Friday
  - Please put your name, as I want to hear from everyone
  - If you want to submit an anonymous form, just take a second copy
  - Be careful what you wish for, it may come true
  - Return in class today, right after 5 minute break (take another if don't have one)
- Lecture next Wednesday March 8 is moved from 306 Soda to ?? because of a conference that day
- Lab 4 progress report in discussion section March 8-10; project also due in discussion sections March 15 to 17; everyone needs to be there for both meetings
- Read the course newsgroup to keep uptodate on latest news

## Break (5 Minutes)

- Turn in class surveys!

## Exceptions and Interrupts

- Control is hardest part of the design
- Hardest part of control is exceptions and interrupts
  - events other than branches or jumps that change the normal flow of instruction execution
  - exception is an unexpected event from within the processor; e.g., arithmetic overflow
  - interrupt is an unexpected event from outside the processor; e.g., I/O
- MIPS convention: exception means any unexpected change in control flow, without distinguishing internal or external; use the term interrupt only when the event is externally caused.

<u>Type of event</u>	<u>From where?</u>	<u>MIPS terminology</u>
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

## How are Exceptions Handled?

- Machine must save the address of the offending instruction in the EPC (exception program counter)
- Then transfer control to the OS at some specified address
  - OS performs some action in response, then terminates or returns using EPC
- 2 types of exceptions in our current implementation: undefined instruction and an arithmetic overflow
- Which Event caused Exception?
  - Option 1 (used by MIPS): a Cause register contains reason
  - Option 2 Vectored interrupts: address determines cause.
    - addresses separated by 32 instructions
    - E.g.,

<u>Exception Type</u>	<u>Exception Vector Address (in Binary)</u>
Undefined instruction	01000000 00000000 00000000 00000000 <sub>two</sub>
Arithmetic overflow	01000000 00000000 00000000 01000000 <sub>two</sub>

cs 152 μprog..24

©DAP & SIK 1995

## Additions to MIPS ISA to support Exceptions

- EPC—a 32-bit register used to hold the address of the affected instruction.
- Cause—a register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, though some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1.
- 2 control signals to write EPC and Cause
- Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 00000000<sub>two</sub>
- May have to undo  $PC = PC + 4$ , since want EPC to point to offending instruction (not its successor);  $PC = PC - 4$

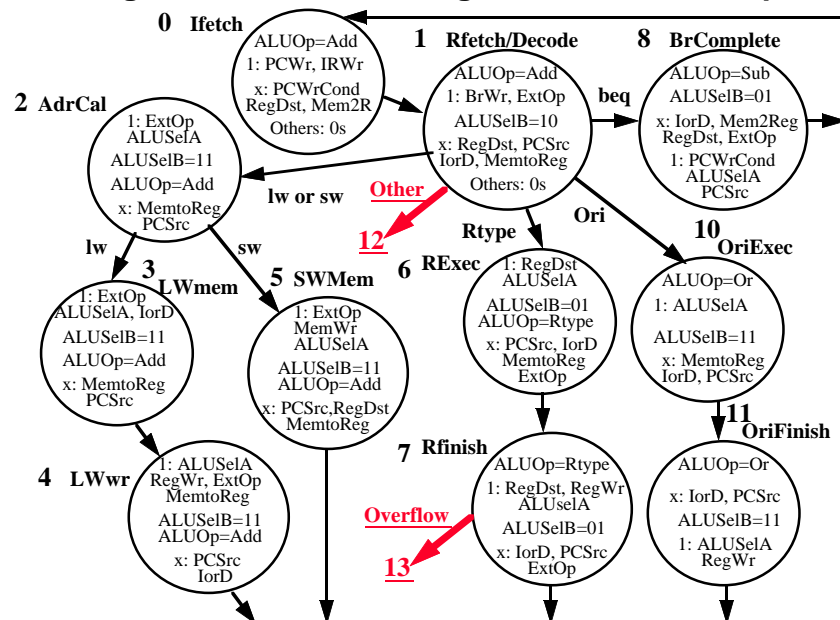
cs 152 μprog..25

©DAP & SIK 1995

## How Control Detects Exceptions

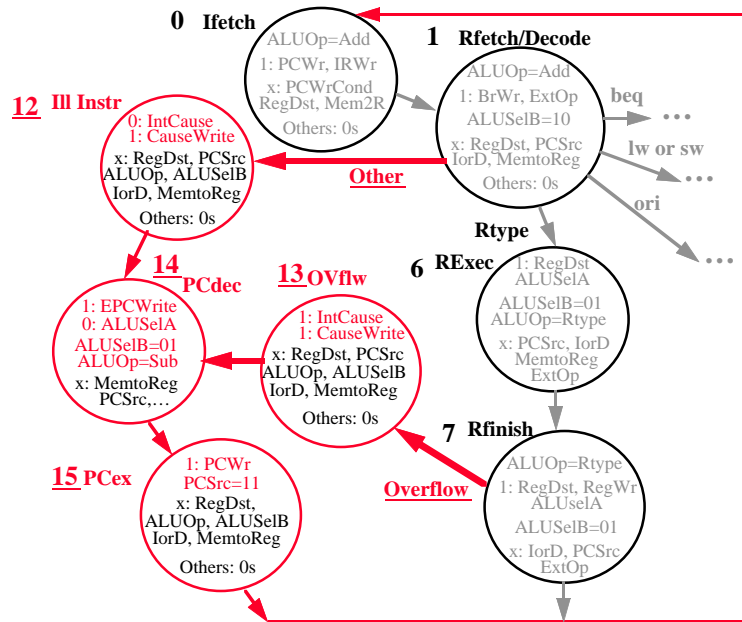
- **Undefined Instruction**—detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
  - Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- **Arithmetic overflow**—Chapter 4 included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state for state 7
- **Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.**
  - Complex interactions makes the control unit the most challenging aspect of hardware design

## Changes to Finite State Diagram to Detect Exceptions





## Extra States to Handle Exceptions



cs 152 μprog..28

©DAP & SIK 1995

## What happens to Instruction with Exception?

- Some problems could occur in the way the exceptions are handled.
- For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result, because the overflow branch is in the state when the write completes.
- However, the architecture may define the instruction as having no effect if the instruction causes an exception; MIPS specifies this.
- When get to virtual memory we will see that certain classes of exceptions prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance.

cs 152 μprog..29

©DAP & SIK 1995

## Summary

- Control is hard part of computer design
- Microprogramming specifies control like assembly language programming instead of finite state diagram
- Next State function, Logic representation, and implementation technique can be the same as finite state diagram, and vice versa
- Exceptions are the hard part of control
- Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
- As we get pipelined CPUs that support page faults on memory accesses which means that the instruction cannot complete AND you must be able to restart the program at exactly the instruction with the exception, it gets even harder