RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 14
Final Review

Nagi Naganathan
December 8, 2016

Course Goals

- Develop an understanding digital logic which forms a foundation for Computer Architecture, Computer Engineering, Embedded Systems, and VLSI Design
- Learn to build digital circuits
- The course builds on "**Digital Logic Design**" and will introduce logic design using hardware description language such as SystemVerilog and provide insight into Computer Architecture
- Aim of the course is to provide a practical view of building digital circuits through several real examples (hardware and software) with hands on Altera DE2-115 FPGA implementation in the labs
- Develop a comprehensive understanding of the underlying technologies and design techniques used to build a RISC microprocessor (MIPS, ARM)

Course Description

- Combines digital logic design and computer architecture
- It starts with a discussion of combinational logic: logic gates, minimization techniques, arithmetic circuits, combinational logic, sequential circuits and state machines
- Understand what's under the hood of a computer
- Study of MIPS architecture (First RISC Processor)
- Study of ARM architecture (**New to this course**) – Popular processor for all smart phones and tablets
- Overview of advanced microarchitecture (branch prediction, superscalar multithreading etc.,
- Study of memory hierarchy
- Develop debugging skills by designing, building and testing digital circuits using commercially available CAD tools (Altera) – Lab

Course Outcomes

- **Digital Logic/System Design**
  - Understand digital logic - Combinational and Sequential circuits
  - Apply the principles of abstraction, modularity, hierarchy and regularity in digital design
  - Design complex state machines that's present in all computers
- **Hardware Description Language**
  - Design Digital Logic using SystemVerilog
  - Design high speed arithmetic circuits using FPGA
  - Use of CAD Tools
  - Simulation
- **Computer Architecture**
  - Understand what's under the hood of a computer
  - Understand RISC Architecture
  - Develop a good understanding of MIPS and ARM Architectures

Topic	Book Chapters	Dates
1. Organizational issues. Information revolution. Basic hardware concepts. 2. Number systems, Binary addition, subtraction, Representation of negative numbers, 2's complement addition/subtraction. 3. Logic Signals and Gates	Chapter 1 (From Zero to One) Chapter 2 (Combinational Logic Design) Slides - Lectures #1, #2, #3	9/8/2016
1. Switching algebra, Theorems, Standard representation of logic functions 2. Combinational circuits, Truth table, Karnaugh maps, Minimization techniques	Chapter 2 (Combinational Logic Design) Slides - Lectures #3, #4	9/15/2016
1. Latches and Flip-Flops 2. Synchronous Logic Design 3. Finite State Machine (FSM) 4. Timing	Chapter 3 (Sequential Logic Design Slides - Lecture #5	9/22/2016
1. Digital Building Blocks (Arithmetic Circuits) a. Adder b. Subtracter c. Comparator d. Multiplier e. Counters, Shift Registers	Chapter 5 (Digital Building Block) Slides – Lectures #6	9/29/2016
1. Hardware Description Language – SystemVerilog	Chapter 4 (Hardware Description Language) Slides - Lecture #7	10/6/2016
Mid Term 1 – 10/13/2016		

Mid Term 1 – 10/13/2016		
1. Guest Lecture – Verification		10/20/2016
2. RISC Architecture – MIPS 3. Assembly Language 4. Programming	Chapter 6 (Architecture) Slides – Lecture #8)	10/27/2016
MIPS Microarchitecture	Chapter 7 (Microarchitecture) (Slides - Lecture #9)	11/3/2016
1. RISC Architecture – ARM 2. Assembly Language 3. Programming 4. ARM Microarchitecture	Chapter 6, 7 (Supplemental Book – Architecture) (Slides – Lecture #10, #11)	11/10/2016
1. Review 2. Memory and I/O Subsystems – Cache (MIPS, ARM)	Chapter 8 (Memory & I/O Sub-Systems) Slides – Lectures #12, #13	11/17/2016
Thanks Giving Recess – 11/24/2016 – 11/27/2016		
Mid Term 2 – 12/1/2016		

Chapter 1 :: Topics

- **Background**
- **The Game Plan**
- **The Art of Managing Complexity**
- **The Digital Abstraction**
- **Number Systems**
- **Logic Gates**
- **Logic Levels**
- **CMOS Transistors**
- **Power Consumption**

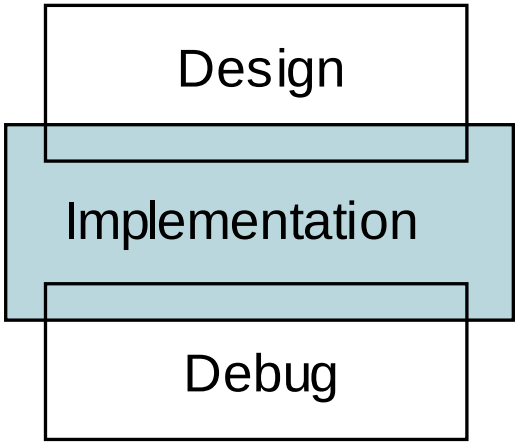
FROM ZERO TO ONE

The Game Plan

- Purpose of course:
  - Understand what's under the hood of a computer
  - Learn the principles of digital design
  - Learn to systematically debug increasingly complex designs
  - Design and build a microprocessor

FROM ZERO TO ONE

The Process of Design



Design

Initial concept: what is the function performed by the object?
Constraints: How fast? How much area? How much cost?
Refine abstract functional blocks into more concrete realizations

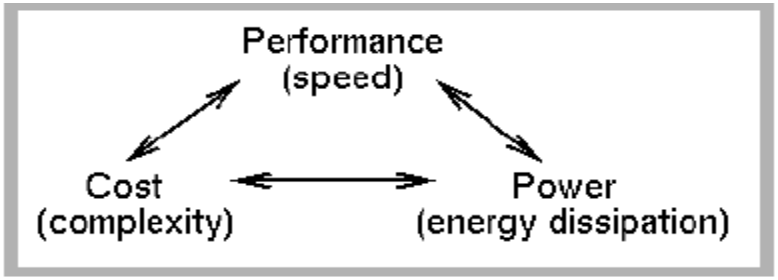
Implementation

Assemble primitives into more complex building blocks
Composition via wiring
Choose among alternatives to improve the design

Debug

Faulty systems: design flaws, composition flaws, component flaws
Design to make debugging easier
Hypothesis formation and troubleshooting skills

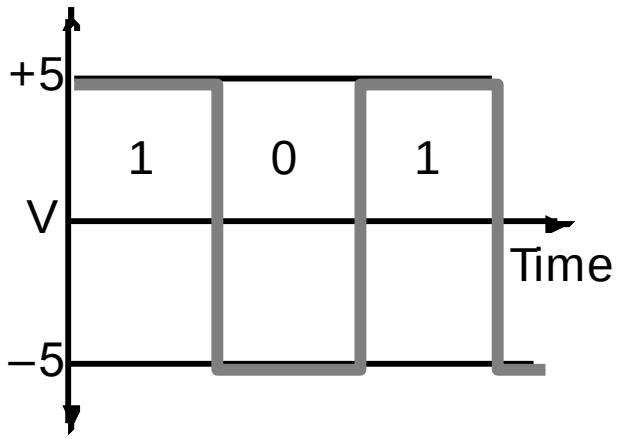
Design Tradeoffs

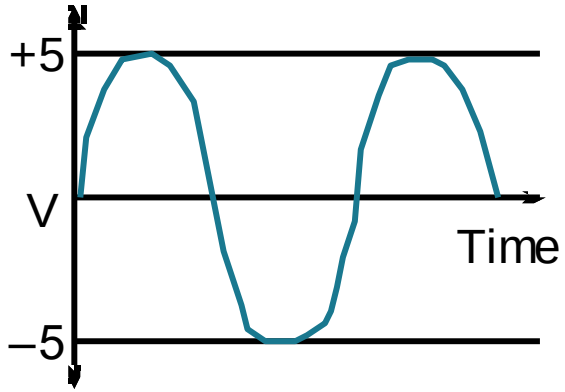


- You can improve on one at the expense of worsening one or both of the others.
- These tradeoffs exist at every level in the system design - every sub-piece and component.
- Design Specification -
  - Functional Description.
  - Performance, cost, power constraints.
- As a designer you must make the tradeoffs necessary to achieve the function within the constraints.

Digital Systems

Digital vs. Analog Waveforms



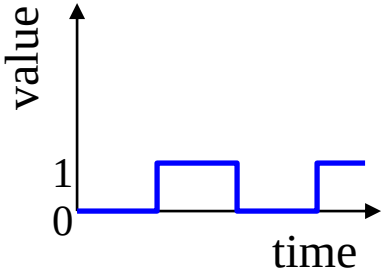


Digital:
only assumes discrete values

Analog:
values vary over a broad range continuously

Digital Signals with Only Two Values: Binary

- ***Binary*** digital signal -- only *two* possible values
  - Typically represented as **0** and **1**
  - One *b*inary dig*it* is a ***bit***
  - We'll only consider *binary* digital signals
  - Binary is popular because
    - Transistors, the basic digital electric component, operate using *two* voltages (more in Chpt. 2)
    - Storing/transmitting one of *two* values is easier than three or more (e.g., loud beep or quiet beep, reflection or no reflection)



Examples of Digital Revolution

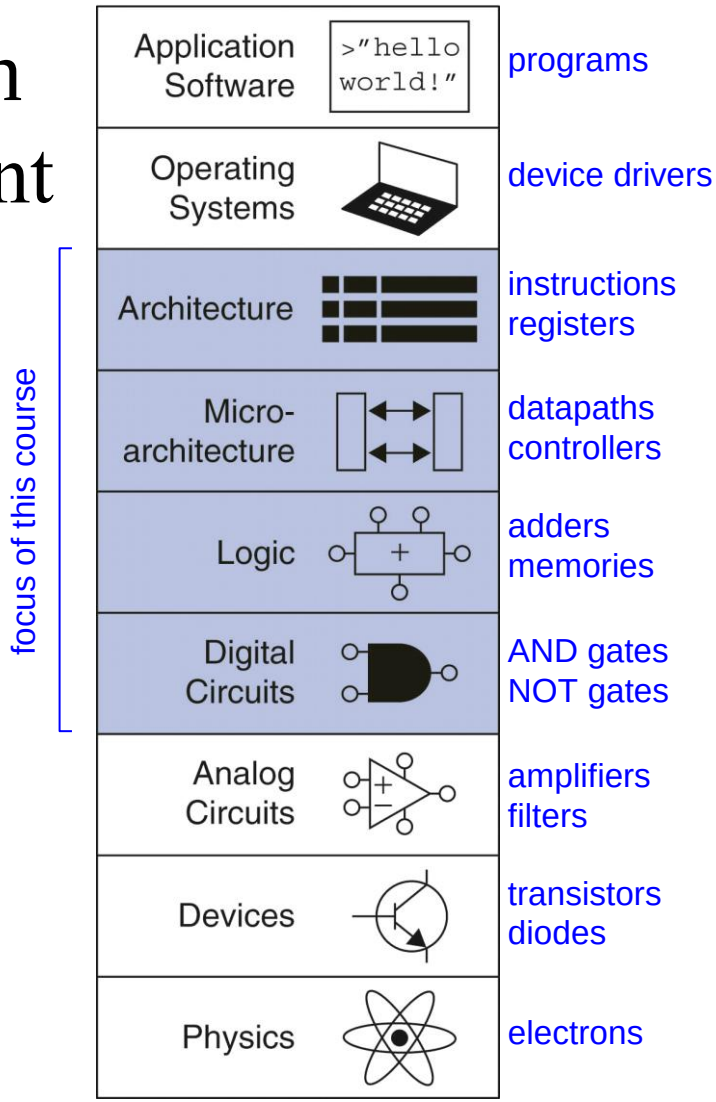
- Digital Cameras – Still Pictures
- Video Recordings
- Audio Recordings
- Traffic Lights
- Movie Effects

Advantages of Digital

- Reproducibility of results – Analog depends on temperature, power supply voltage, component aging etc.,
- Ease of design
  - Logic Design – No Math – Operation viewed mentally without insights about the operation of capacitors, transistors etc.,
- Flexibility and Functionality – scrambling voice
- Programmable
- Speed
- Economy
- Steadily advancing technology

Abstraction

- Hiding details when they aren't important



The Three -Y's

- Hierarchy
  - A system divided into modules and submodules
- Modularity
  - Having well-defined functions and interfaces
- Regularity
  - Encouraging uniformity, so modules can be easily reused

Boolean Algebra and its Relation to Digital Circuits

- To understand the benefits of “logic gates” vs. switches, we should first understand Boolean algebra
- “Traditional” algebra
  - Variables represent real numbers (x, y)
  - Operators operate on variables, return real numbers (2.5*x + y - 3)
- ***Boolean Algebra***
  - Variables represent 0 or 1 only
  - Operators return 0 or 1 only
  - Basic operators
    - AND: *a AND b* returns 1 only when both a=1 and b=1
    - OR: *a OR b* returns 1 if either (or both) a=1 or b=1
    - NOT: *NOT a* returns the opposite of a (1 if a=0, 0 if a=1)

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	1
1	0

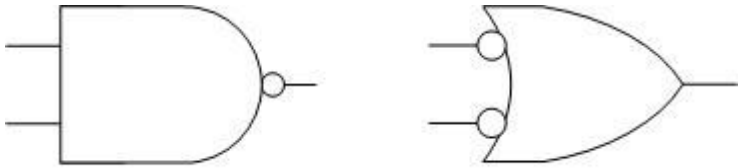
Logical Operations

- The three basic logical operations are:
  - AND
  - OR
  - NOT
- AND is denoted by a dot (·).
- OR is denoted by a plus (+).
- NOT is denoted by an overbar ($\bar{}$), a single quote mark (') after, or (~) before the variable.

- DeMorgan's Theorem

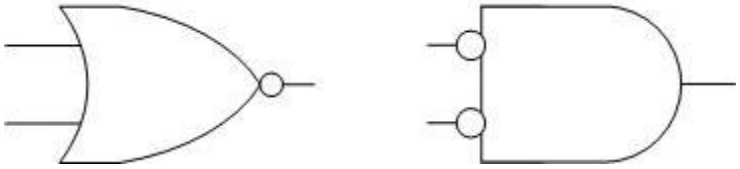
Part 1: an AND gate whose output is complemented is equivalent to

an OR gate whose inputs are complemented



Part 2: an OR gate whose output is complemented is equivalent to

an AND gate whose inputs are complemented



Converting to Boolean Equations

- Q1. A fire sprinkler system should spray water if high heat is sensed and the system is set to enabled.
  - Answer: Let Boolean variable h represent “high heat is sensed,” e represent “enabled,” and F represent “spraying water.” Then an equation is: F = h AND e.
- Q2. A car alarm should sound if the alarm is enabled, and either the car is shaken or the door is opened.
  - Answer: Let a represent “alarm is enabled,” s represent “car is shaken,” d represent “door is opened,” and F represent “alarm sounds.” Then an equation is: F = a AND (s OR d).
  - (a) Alternatively, assuming that our door sensor d represents “door is closed” instead of open (meaning d=1 when the door is closed, 0 when open), we obtain the following equation: F = a AND (s OR NOT(d)).

Complement Numbers

- 1's complement –

  One's complement: One obtains 1's complement[4] of a binary number by subtracting the number from the binary number consisting of the same number of bits, with all 1's. This is the same as switching all of the bits (substituting every 0 by a 1 and vice-versa).

- Example 011 = 111 – 011 = 100
- 2's complement

Example: The 2's Complement of binary 101100 is :

010011 ← 1's Complement

\+ 1

010100

The 2's Complement can be obtained by leaving all least significant 0's and the first 1 unchanged and replacing 1's with 0's and 0's with 1's in all other significant digits.

Example:

The 2's Complement of 111001100 is: 000110 1 00

All other significant bits are changed

First 1 unchanged

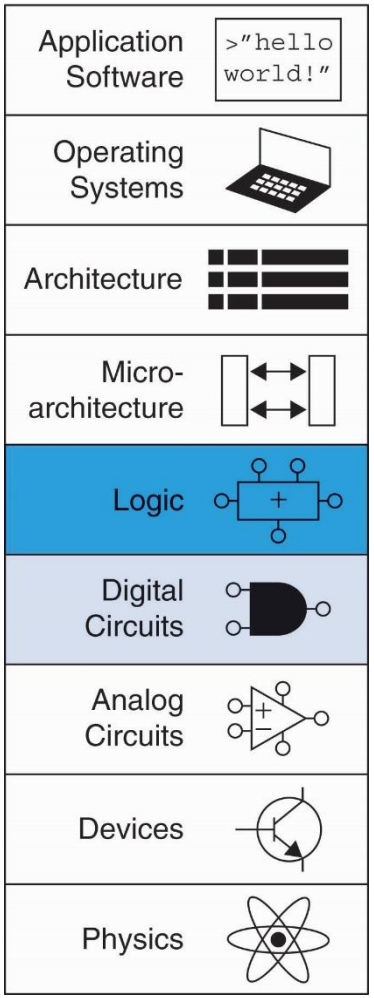
Two least significant 0: unchanged

“Taking the Two’s Complement”

- Flip the sign of a two’s complement number
- Method:
  1. Invert the bits
  2. Add 1
- Example: Flip the sign of $3_{10} = 0011_2$

Chapter 2 :: Topics

- Introduction
- Boolean Equations
- Boolean Algebra
- From Logic to Gates
- Multilevel Combinational Logic
- X's and Z's, Oh My
- Karnaugh Maps
- Combinational Building Blocks
- Timing



Some Definitions

- Complement: variable with a bar over it
  $\bar{A}, \bar{B}, \bar{C}$
- Literal: variable or its complement
  $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- Implicant: product of literals
  $AB\bar{C}, \bar{A}C, BC$
- Minterm: product that includes all input variables
  $AB\bar{C}, A\bar{B}\bar{C}, ABC$
- Maxterm: sum that includes all input variables
  $(A+\bar{B}+C), (\bar{A}+B+\bar{C}), (\bar{A}+\bar{B}+C)$

SOP & POS Form

- SOP – sum-of-products

O	C	E	minterm
0	0	0	$\overline{O}\ \overline{C}$
0	1	0	$\overline{O}\ C$
1	0	1	$O\ \overline{C}$
1	1	0	$O\ C$

$$E = O\overline{C}$$
$$= \Sigma(2)$$

- POS – product-of-sums

O	C	E	maxterm
0	0	0	$O + C$
0	1	0	$O + \overline{C}$
1	0	1	$\overline{O} + C$
1	1	0	$\overline{O} + \overline{C}$

$$E = (O + C)(O + \overline{C})(\overline{O} + \overline{C})$$
$$= \Pi(0, 1, 3)$$

Boolean Function Minimization

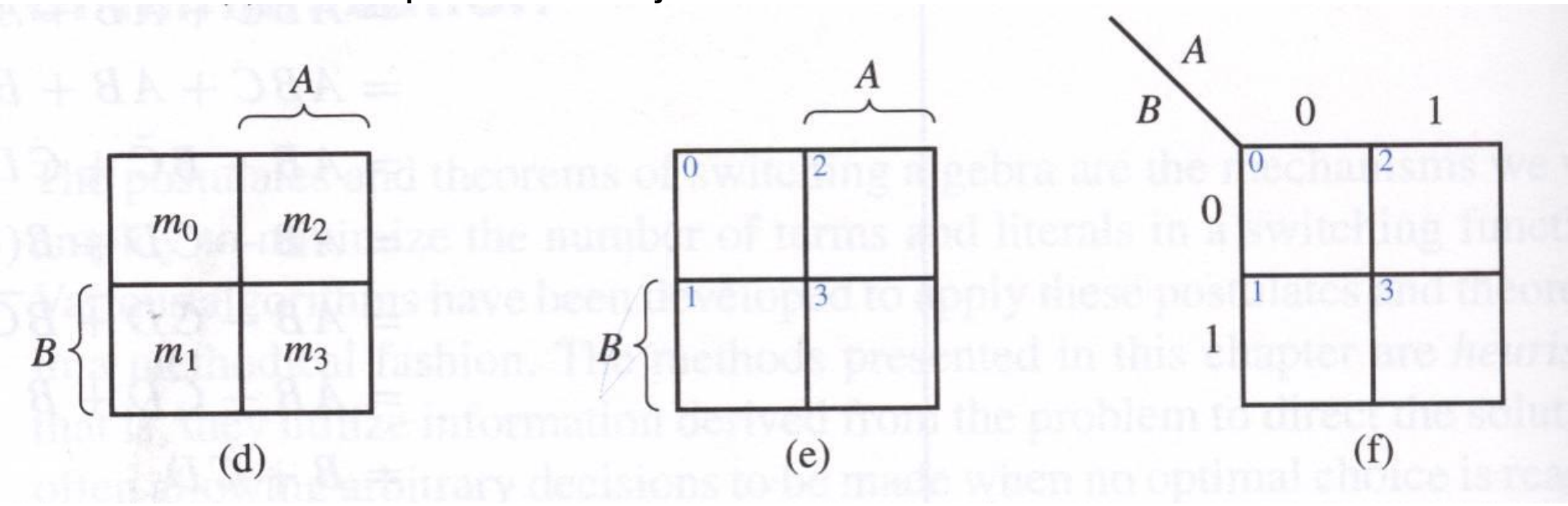
- Simpler expression leads to minimum number of gates
- Cheaper, Faster, Smaller
- Algebraic methods
  - Using theorems
  - Harder
- Karnaugh Map (K-map)
  - Pictorial/Graphical representation and similar to Venn Diagram
  - Easier due to pattern matching
  - Limited to 5 variables
  - Quine-Mccluskey for large combinational circuits

Simplification of switching functions

- Simplify – what?
  - SOP/POS form has products/sums and literals
    - **Literal: each appearance of a variable or its complement**
  - Minimize number of sums/products
    - Reduces total gate count
  - Minimize number of variables in each sum/product
    - Reduces number of inputs to each gate
    - PLDs have fixed # of inputs; only the number of terms need to be minimized there

Karnaugh maps

– Adjacencies are preserved when going from c) to d)
  - They are the same, only the areas are made equal in d), which preserves adjacencies



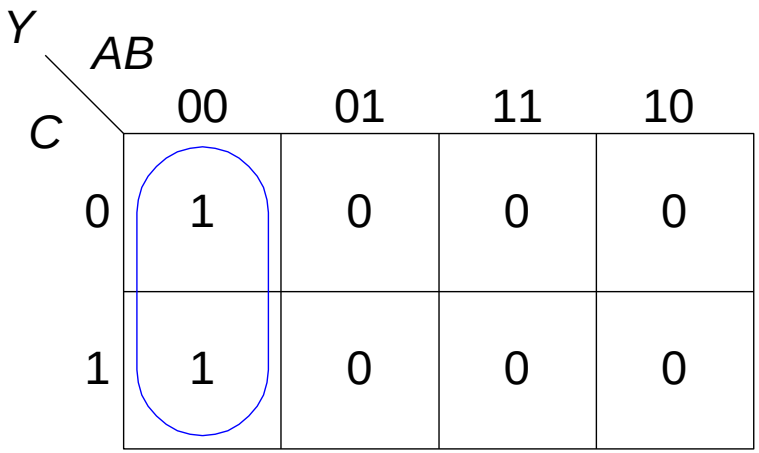
  - Subscripts are dropped in e); realize that 2&3 is A; 1&3 is B
  - In f) the labels change and become 0 and 1

– Each square of the K-map is 1 row of the TT

K-Map

- Circle 1's in adjacent squares
- In Boolean expression, include only literals whose true and complement form are ***not*** in the circle

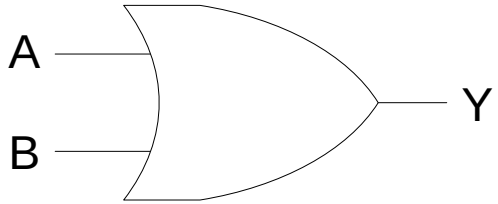
A	B	C	Y
0	0	0	**1**
0	0	1	**1**
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



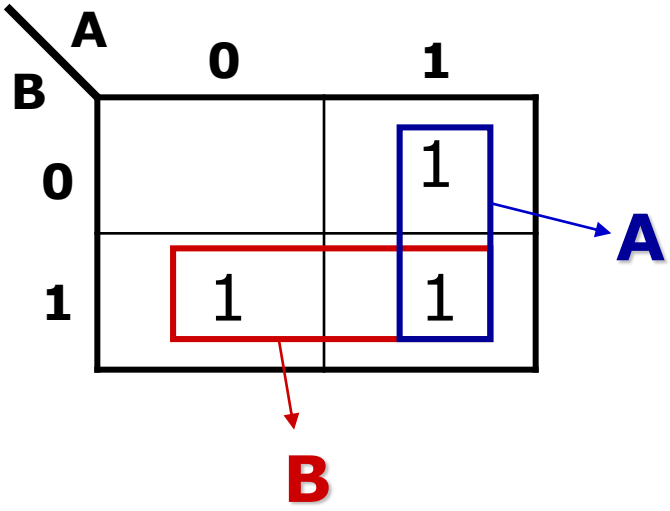
$$Y = \bar{A}\bar{B}$$

Example

2-variable Karnaugh maps are trivial but can be used to introduce the methods you need to learn. The map for a 2-input OR gate looks like this:



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

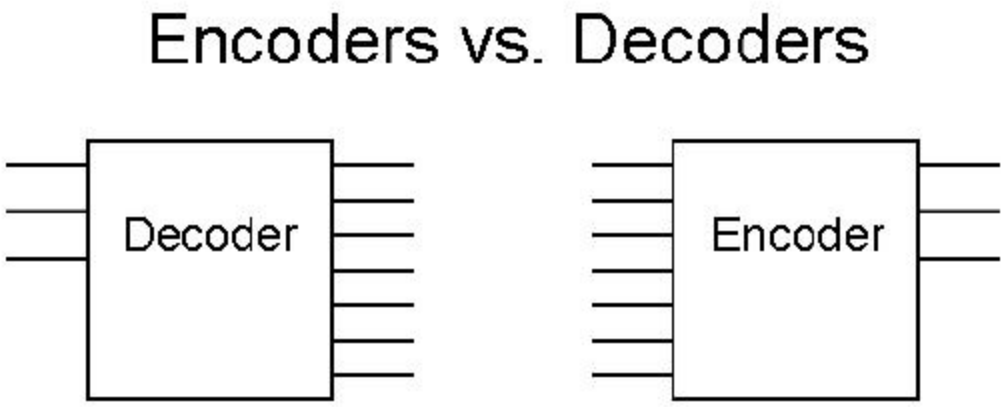


A+B

Combinational Building Blocks

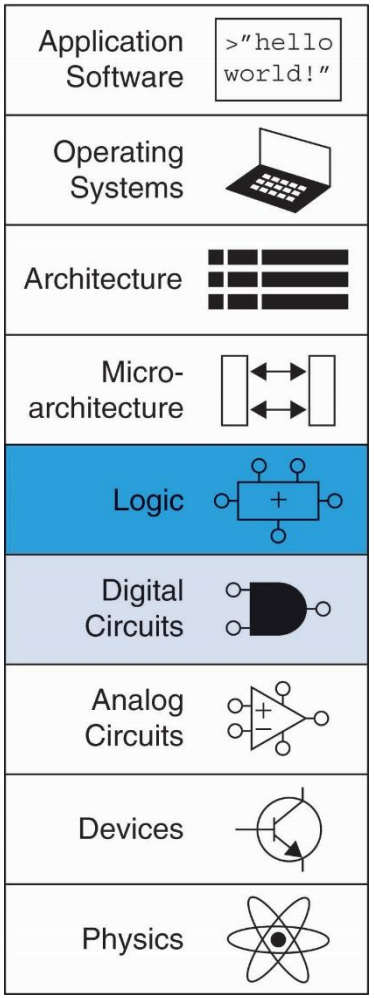
- Multiplexers
- De-Multiplexers
- Decoders
- Encoders

Encoder vs Decoder



Chapter 3 :: Topics

- **Introduction**
- **Latches and Flip-Flops**
- **Synchronous Logic Design**
- **Finite State Machines**
- **Timing of Sequential Logic**
- **Parallelism**



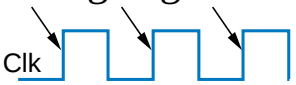
Introduction

- Outputs of sequential logic depend on current *and* prior input values – it has ***memory***.
- Some definitions:
  - **State:** all the information about a circuit necessary to explain its future behavior
  - **Latches and flip-flops:** state elements that store one bit of state
  - **Synchronous sequential circuits:** combinational logic followed by a bank of flip-flops

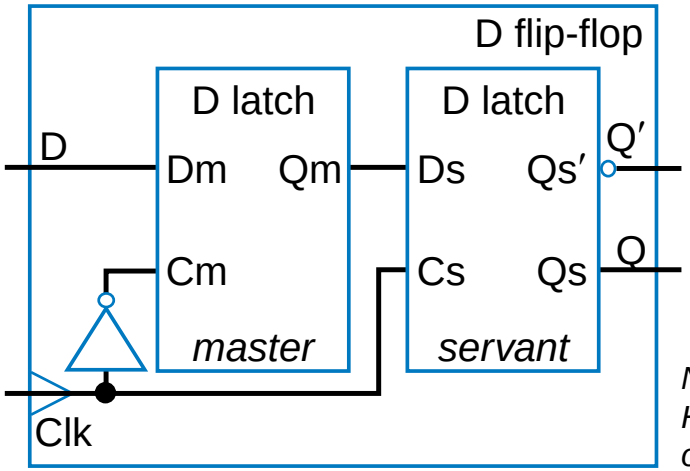
D Flip-Flop

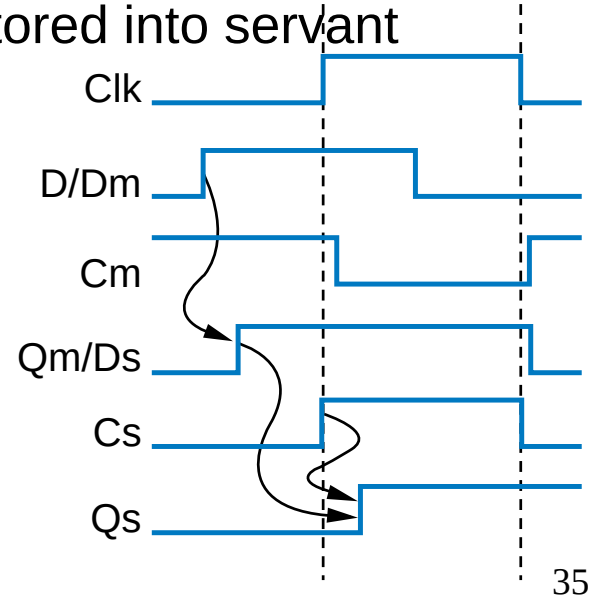
Can we design bit storage that only stores a value on the *rising* edge of a clock signal?

rising edges



- ***Flip-flop***: Bit storage that stores on clock *edge*
- One design – master-servant
  - Clk = 0 – master enabled, loads D, appears at Qm. Servant disabled.
  - Clk = 1 – Master disabled, Qm stays same. Servant latch enabled, loads Qm, appears at Qs.
  - Thus, value at *D (and hence at Qm) when Clk changes from 0 to 1* gets stored into servant

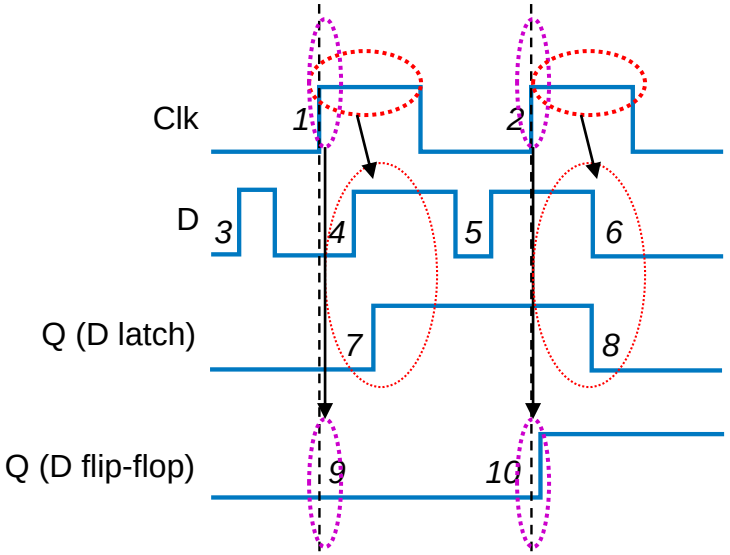




Note: Hundreds of different flip-flop designs exist

D Latch vs. D Flip-Flop

- Latch is level-sensitive
  - Stores D when C=1
- Flip-flop is edge triggered
  - Stores D when C changes from 0 to 1
- Saying “level-sensitive latch” or “edge-triggered flip-flop” is redundant
- Comparing behavior of latch and flip-flop:

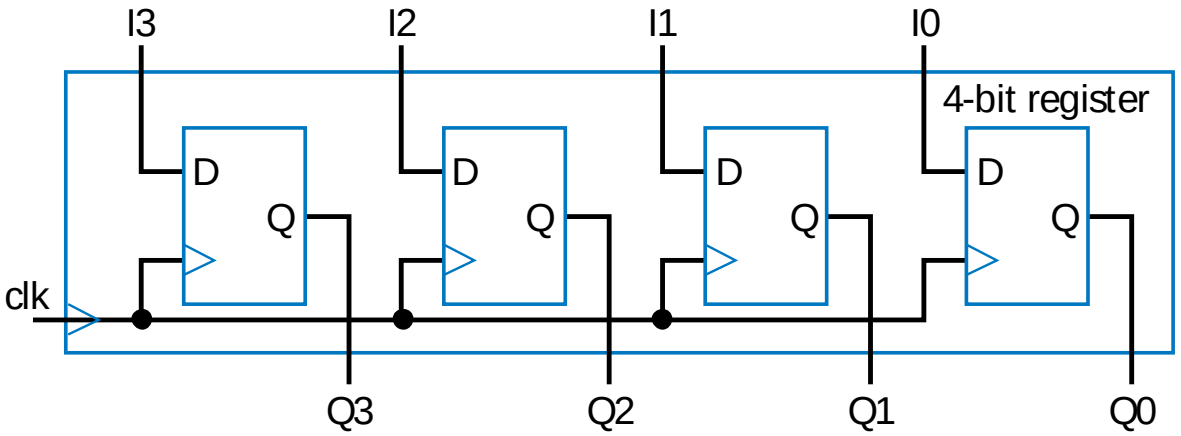


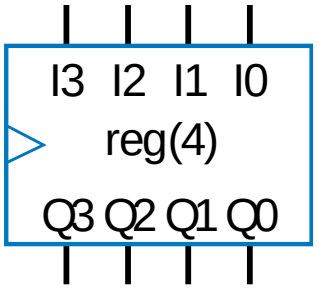
Latch follows D while Clk is 1

Flip-flop only loads D during Clk rising edge

Basic Register

- Typically, we store multi-bit items
  - e.g., storing a 4-bit binary number
- ***Register***: multiple flip-flops sharing clock signal
  - From this point, we'll use registers for bit storage
    - No need to think of latches or flip-flops
    - But now you know what's inside a register

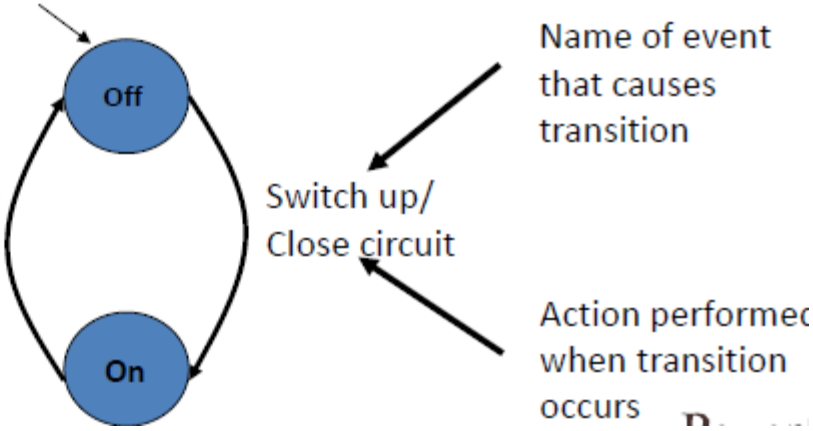




Finite State Machine (Adapted from various sources)

represented by arrows between the states

e.g. Light Switch



- **Event**: Something that happens to the object instantaneously. It is often implemented as an operation on an object.
- **State:** Defines behavior and attribute values held by an object.
- **State transition:** A change of state (in response to an event).

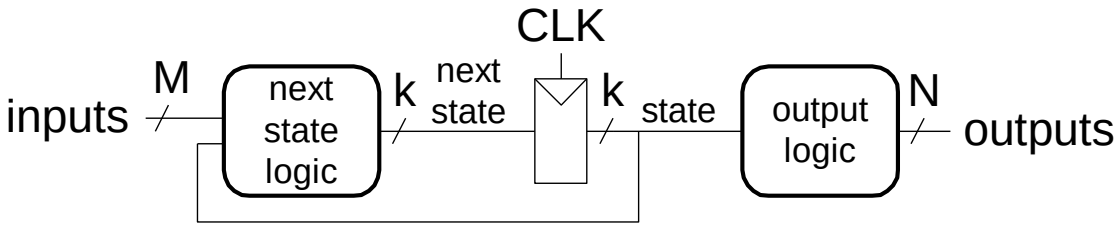
Moore and Mealy Models

- **Sequential Circuits or Sequential Machines are also called *Finite State Machines* (FSMs). Two formal models exist:**
  - **Moore Model**
    - **Named after E.F. Moore**
    - **Outputs are a function ONLY of states**
    - **Usually specified on the states.**
  - **Mealy Model**
    - **Named after G. Mealy**
    - **Outputs are a function of inputs AND states**
    - **Usually specified on the state transition arcs.**

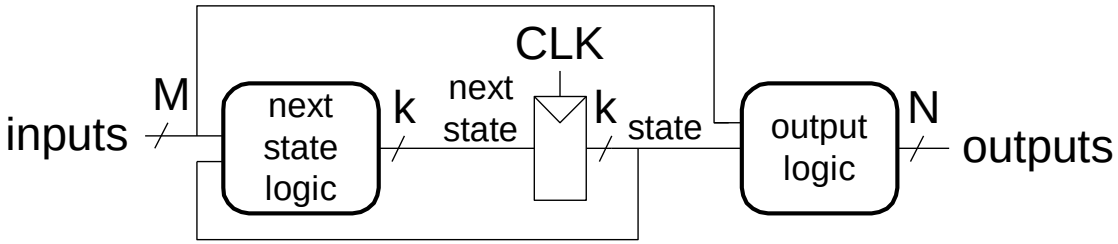
Finite State Machines (FSMs)

- Next state determined by current state and inputs
- Two types of finite state machines differ in output logic:
  - **Moore FSM:** outputs depend only on current state
  - **Mealy FSM:** outputs depend on current state *and* inputs

Moore FSM

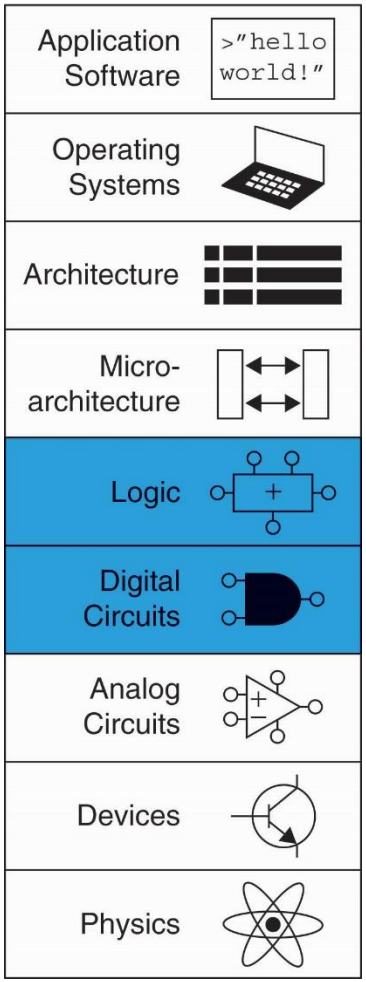


Mealy FSM



Chapter 4 :: Topics

- Introduction
- Combinational Logic
- Structural Modeling
- Sequential Logic
- More Combinational Logic
- Finite State Machines
- Parameterized Modules
- Testbenches

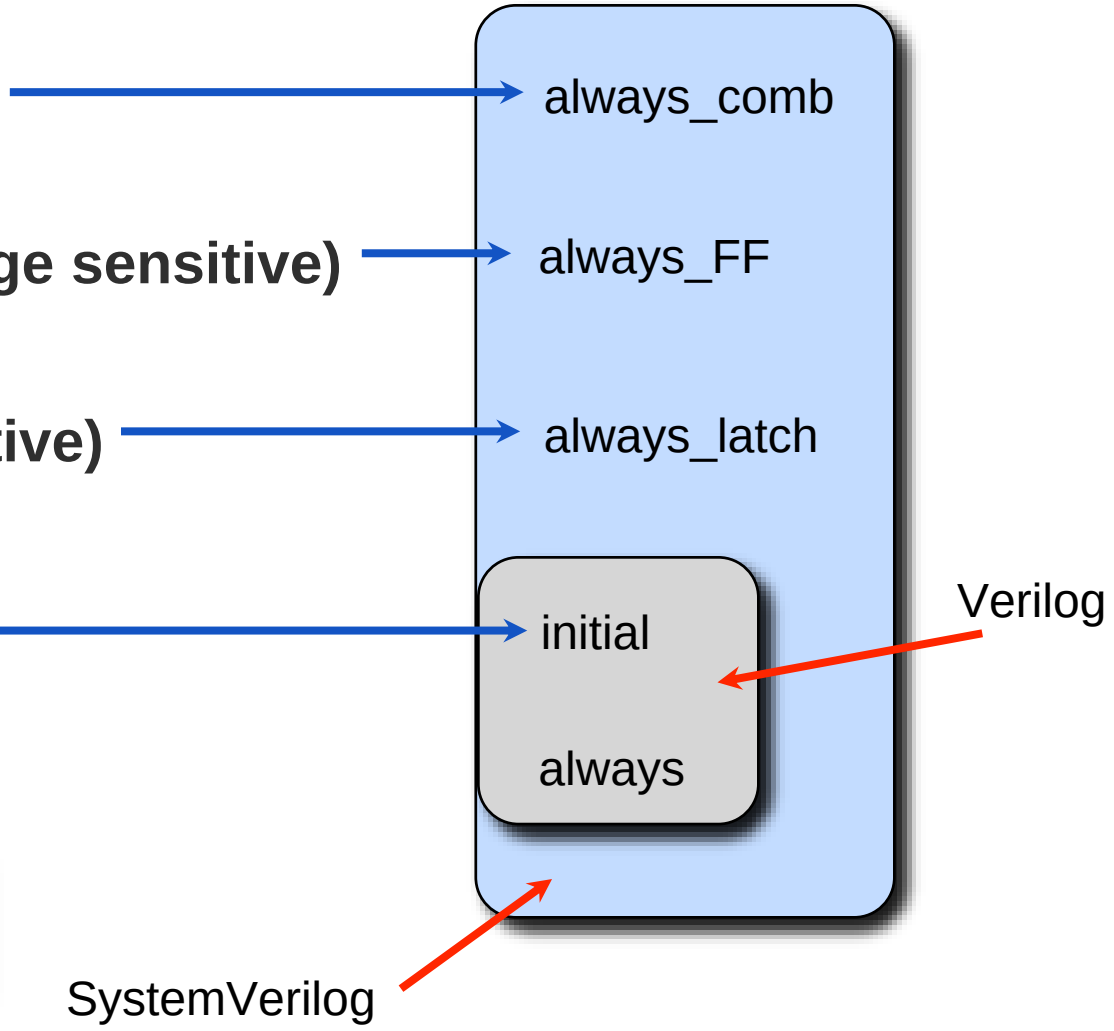


Introduction

- Hardware description language (HDL):
  - specifies logic function only
  - Computer-aided design (CAD) tool produces or *synthesizes* the optimized gates
- Most commercial designs built using HDLs
- Two leading HDLs:
  - **SystemVerilog**
    - developed in 1984 by Gateway Design Automation
    - IEEE standard (1364) in 1995
    - Extended in 2005 (IEEE STD 1800-2009)
  - **VHDL 2008**
    - Developed in 1981 by the Department of Defense
    - IEEE standard (1076) in 1987
    - Updated in 2008 (IEEE STD 1076-2008)

Summary of what we're building

- ❑ Combinational
- ❑ FSM/registers (edge sensitive)
- ❑ Latch (level sensitive)
- ❑ Testbench
- ❑ Other



What is the "how to design" view of this?

There are other things to build, but we'll get to them. They're built up from these basic modeling blocks.

Two types of modeling

- ***Structural*** **modeling**
  - focuses on interconnections of things
  - gate level designs — interconnections of gates
  - module hierarchy and interconnection
    - regardless of what's inside each module

```
module mux1
 (output logic   f,
input            a, b, sel);

 and #5   g1 (f1, a, nsel),
          g2 (f2, b, sel);
  or   #5  g3 (f, f1, f2);
 not       g4 (nsel, sel);
endmodule: mux1
```

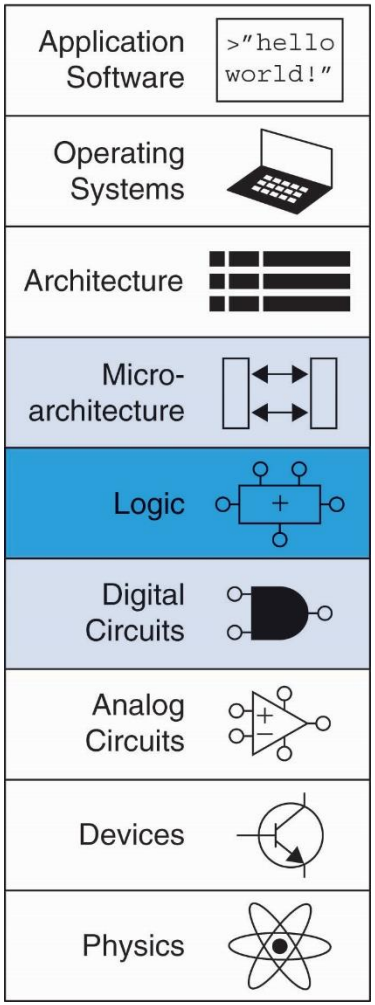
- ***Procedural*** **modeling**
  - modeling the functionality of logic blocks with statements that *look like* a programming language
    - look like, and execute like…
  - using always_i, initial, assign

```
module mux2
 (output logic  f,
input            a, b, sel);

    always_comb
          if (sel)
            f = b;
         else
            f = a;
endmodule: mux2
```

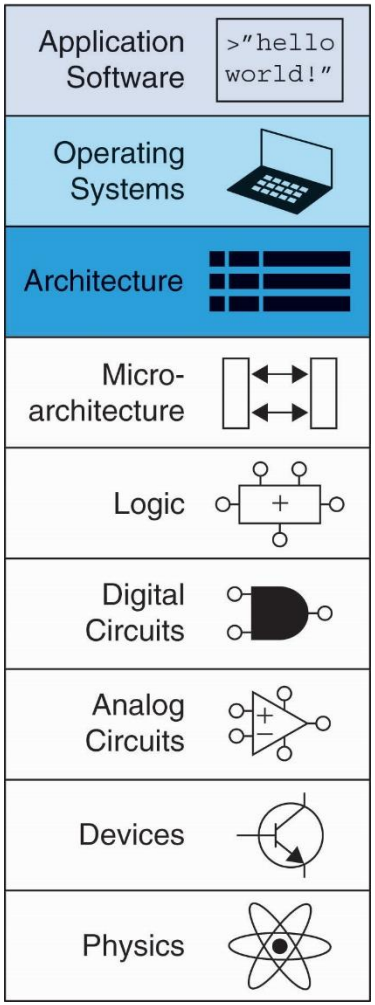
Chapter 5 :: Topics

- **Introduction**
- **Arithmetic Circuits**
- **Number Systems**
- **Sequential Building Blocks**
- **Memory Arrays**
- **Logic Arrays**



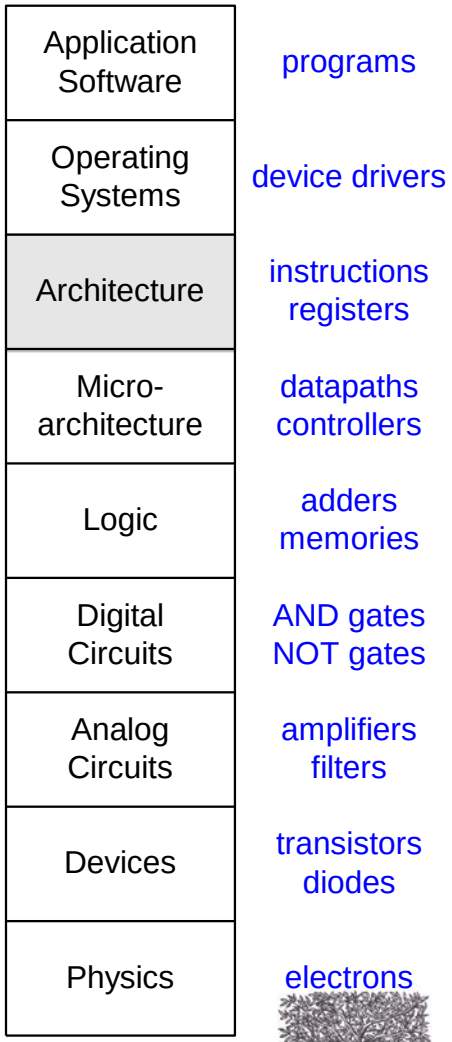
Chapter 6 :: Topics

- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembling, & Loading
- Odds and Ends



Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)

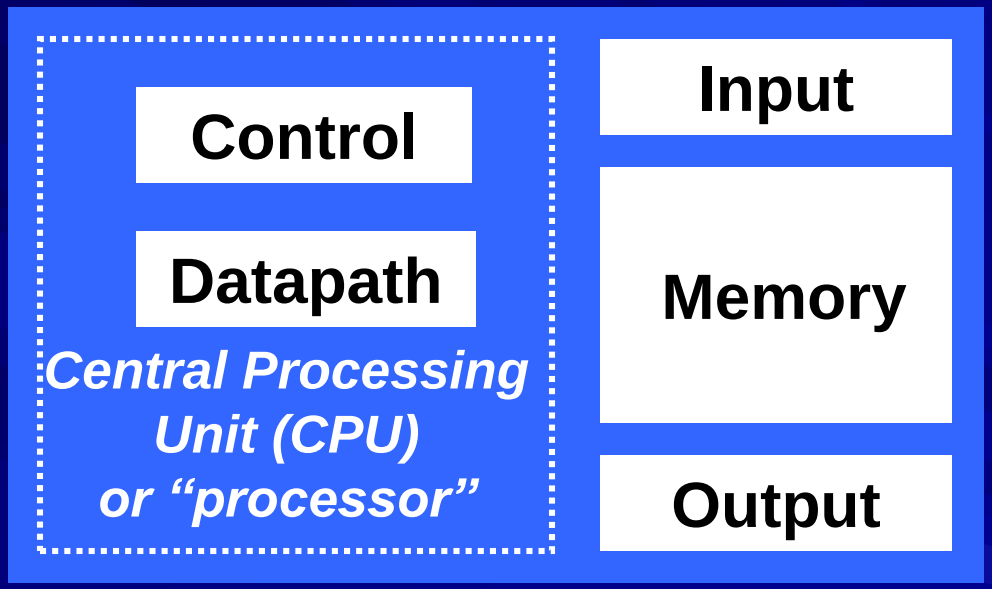


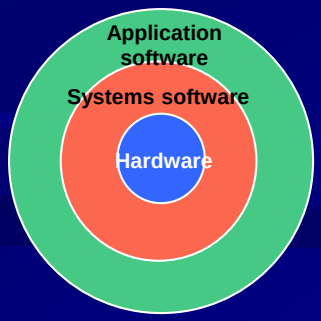
Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

The Hardware of a Computer





FIVE EASY PIECES

Abstractions

The BIG Picture

- Abstraction helps us deal with complexity
  - Hide lower-level detail
- Instruction set architecture (ISA)
  - The hardware/software interface
- Application binary interface
  - The ISA plus system software interface
- Implementation
  - The details underlying and interface

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

Architecture Design Principles

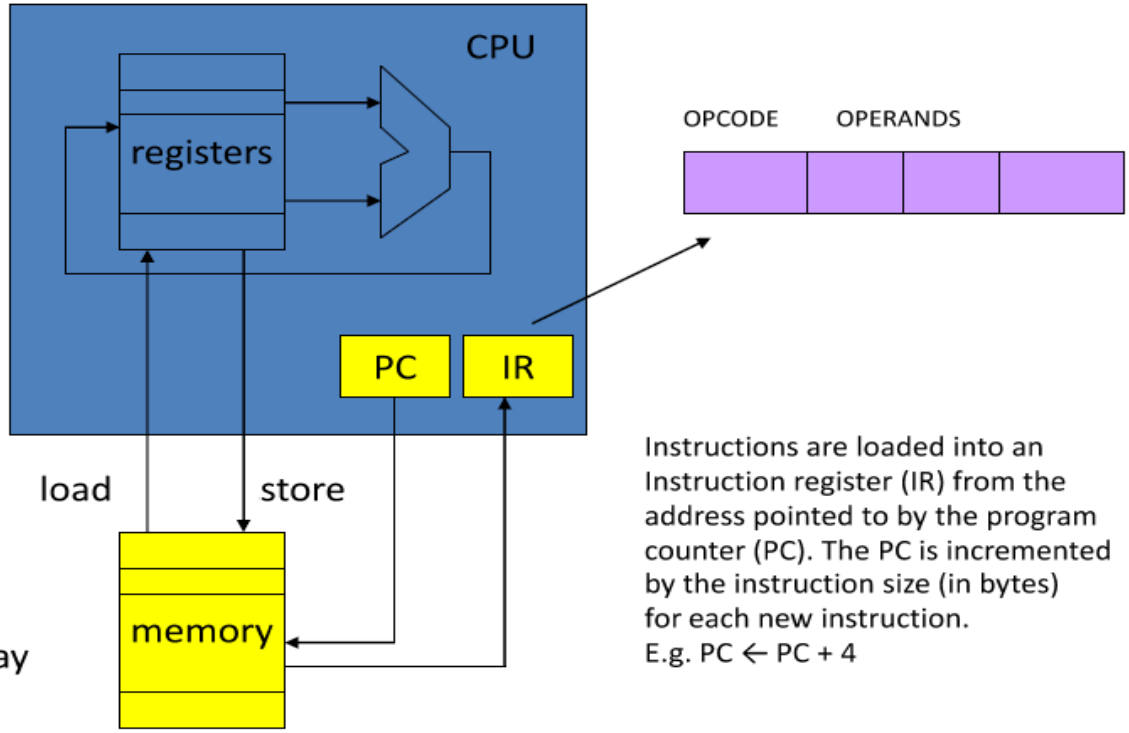
Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - a collection of instructions expressed in "textual" format e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code (mostly true: compound instructions, address labels ....)

Basic Computer Organization

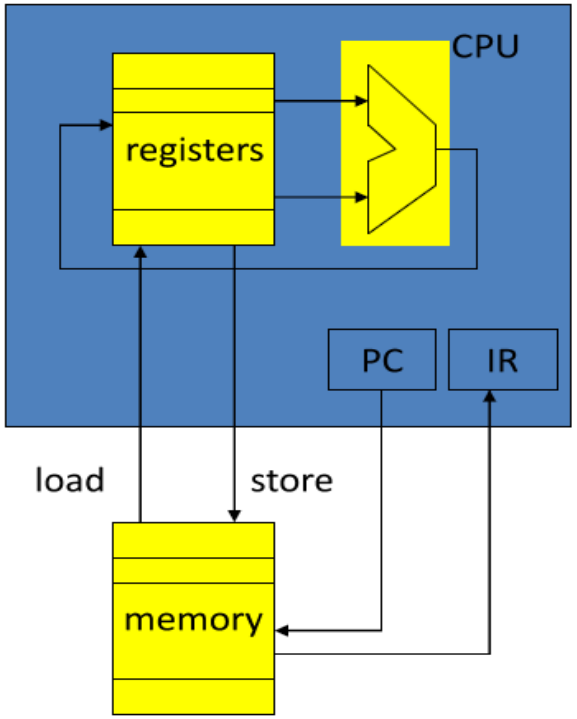
Limited number of **fast** registers for temporary storage



Large amount of **slow** memory Arranged as an array of bytes

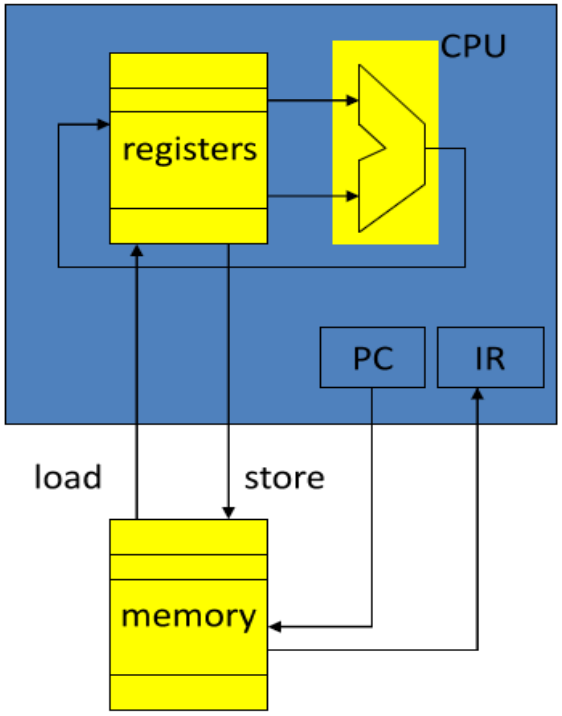
Instructions are loaded into an Instruction register (IR) from the address pointed to by the program counter (PC). The PC is incremented by the instruction size (in bytes) for each new instruction.
E.g. PC ← PC + 4

Load/Store Architecture (Reg-Reg)



- Instructions can **ONLY** get their data and write their result from/to registers.
- The register numbers are specified in the operand fields of the instruction
- Since data is stored in memory, we need special "load" and "store" instructions for transfers between registers and memory. These two instructions are the ONLY ones allowed to access memory

Load/Store Architecture (Reg-Reg)



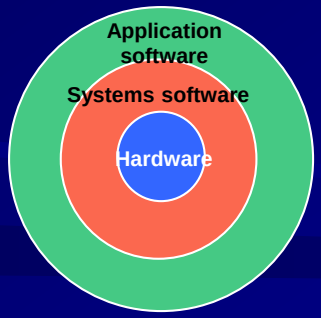
- **RISC** architectures are load/store. The regularity of this architecture enables fast organizations using **pipelining**.

- **CISC** machines (e.g. Intel IA-32) permit instructions to get their data from both registers and memory (mem-reg). These highly irregular architectures (mem-reg, variable-length instructions) are practically impossible to pipeline.
  - The advantage of them is that they produce shorter programs (no loads or stores needed, variable-length instr.), but memory today is cheap and compilers can't really use complex instructions anyways.

- Modern "CISC" machines really just translate the CISC instructions to a set of RISC instructions and run those.
  - Done purely for compatibility reasons.

Instruction Set Architecture (ISA)

- A set of assembly language instructions (ISA) provides a link between software and hardware.
- Given an instruction set, software programmers and hardware engineers work more or less independently.
- ISA is designed to extract the most performance out of the available hardware technology.



Software

Instruction set

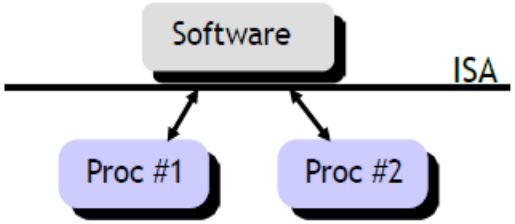
Hardware

ISA

- Defines registers
- Defines data transfer modes between registers, memory and I/O
- Types of ISA: RISC, CISC, VLIW, Superscalar
- Examples:
  - IBM370/X86/Pentium/K6 (CISC)
  - PowerPC (Superscalar)
  - Alpha (Superscalar)
  - MIPS (RISC and Superscalar)
  - Sparc (RISC), UltraSparc (Superscalar)

What's an ISA?

- The ISA is the interface between hardware and software.
- The ISA serves as an abstraction layer between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent



- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars.
- Before ISAs, software was re-written for each new machine.

Types of ISA

- Complex instruction set computer (CISC)
  - Many instructions (several hundreds)
  - An instruction takes many cycles to execute
  - Example: Intel Pentium
- Reduced instruction set computer (RISC)
  - Small set of instructions (typically 32)
  - Simple instructions, each executes in one clock cycle – ***REALLY? Well, almost.***
  - Effective use of pipelining
  - Example: ARM

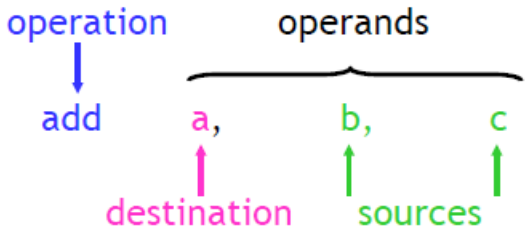
MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains *opcode* and *arguments*.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.

MIPS

- MIPS is a register-to-register, or load/store, architecture.
  - The destination and sources must all be registers.
  - Special instructions, which we'll see later, are needed to access main memory.

- MIPS uses three-address instructions for data manipulation.
  - Each ALU instruction contains a destination and two sources.
  - For example, an addition instruction (a = b + c) has the form:



MIPS

- An Instruction Set Architecture, or ISA, is an **interface** between the hardware and the software.
- An ISA consists of: (for MIPS)
  - a set of operations (instructions) ← arithmetic, logical, conditional, branch, etc.
  - data units (sized, addressing modes, etc.) ← 32-bit data word
  - processor state (registers) ← 32, 32-bit registers
  - input and output control (memory operations) ← load and store
  - execution model (program counter) ← 32-bit program counter

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

C Code

```
a = b + c - d;
```

MIPS assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Arithmetic Instr. (Continued)

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

- Operands must be registers (why?) *Remember von Neumann bottleneck.*
- 32 registers provided
- Each register contains 32 bits

Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- MIPS is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

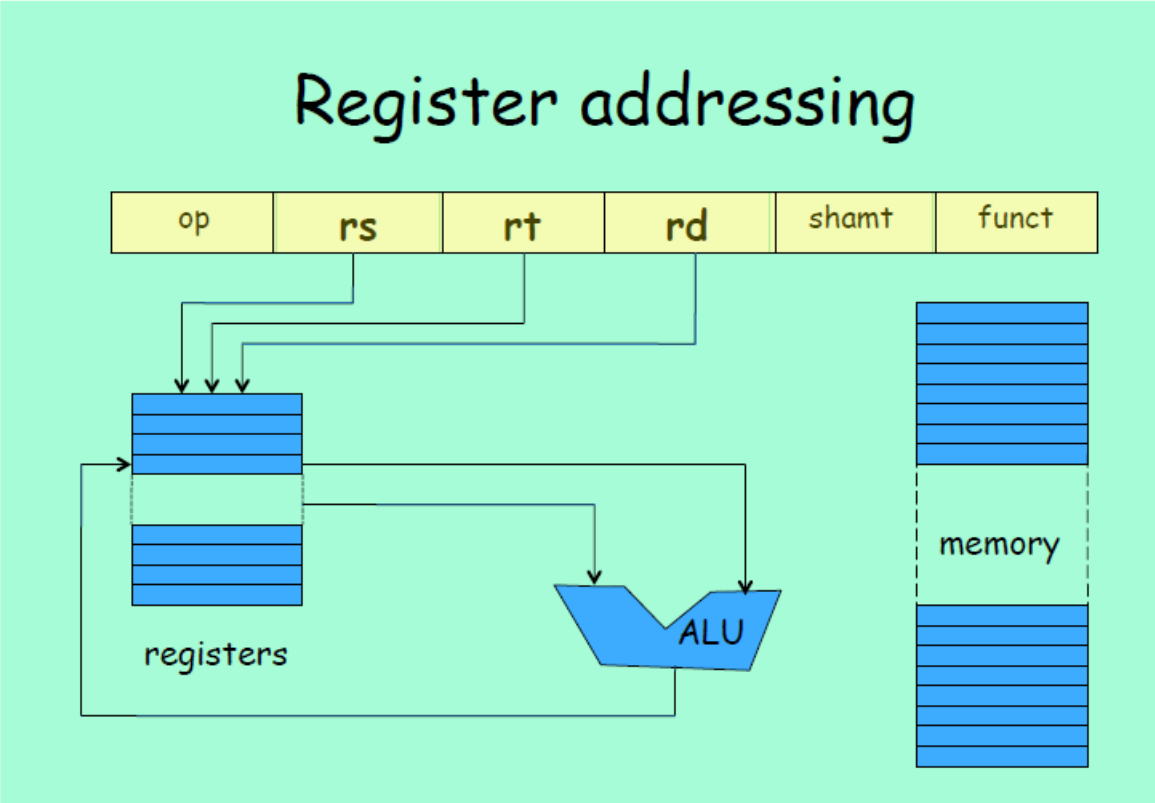
Operands: Registers

- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers


Register addressing
op
rs
rt
rd
shamt
funct
registers
ALU
memory

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Operands: Registers

- Registers:
  - `$` before name
  - Example: `$0`, “register zero”, “dollar zero”
- Registers used for specific purposes:
  - `$0` always holds the constant value 0.
  - the *saved registers*, `$s0`–`$s7`, used to hold variables
  - the *temporary registers*, `$t0` - `$t9`, used to hold intermediate values during a larger computation
  - Discuss others later

MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - By number:

    $0 $1 $2 … $31
  - By (mostly) two-character names, such as:

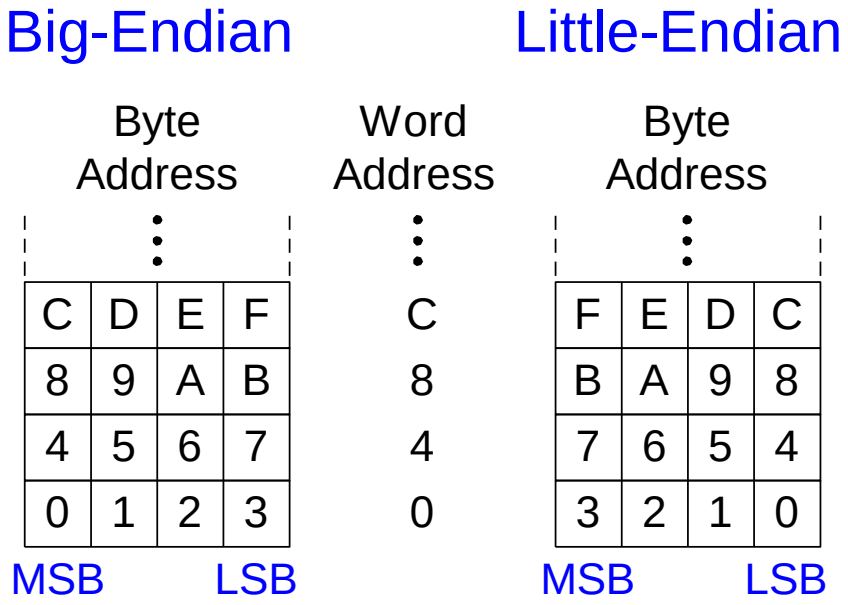
    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra

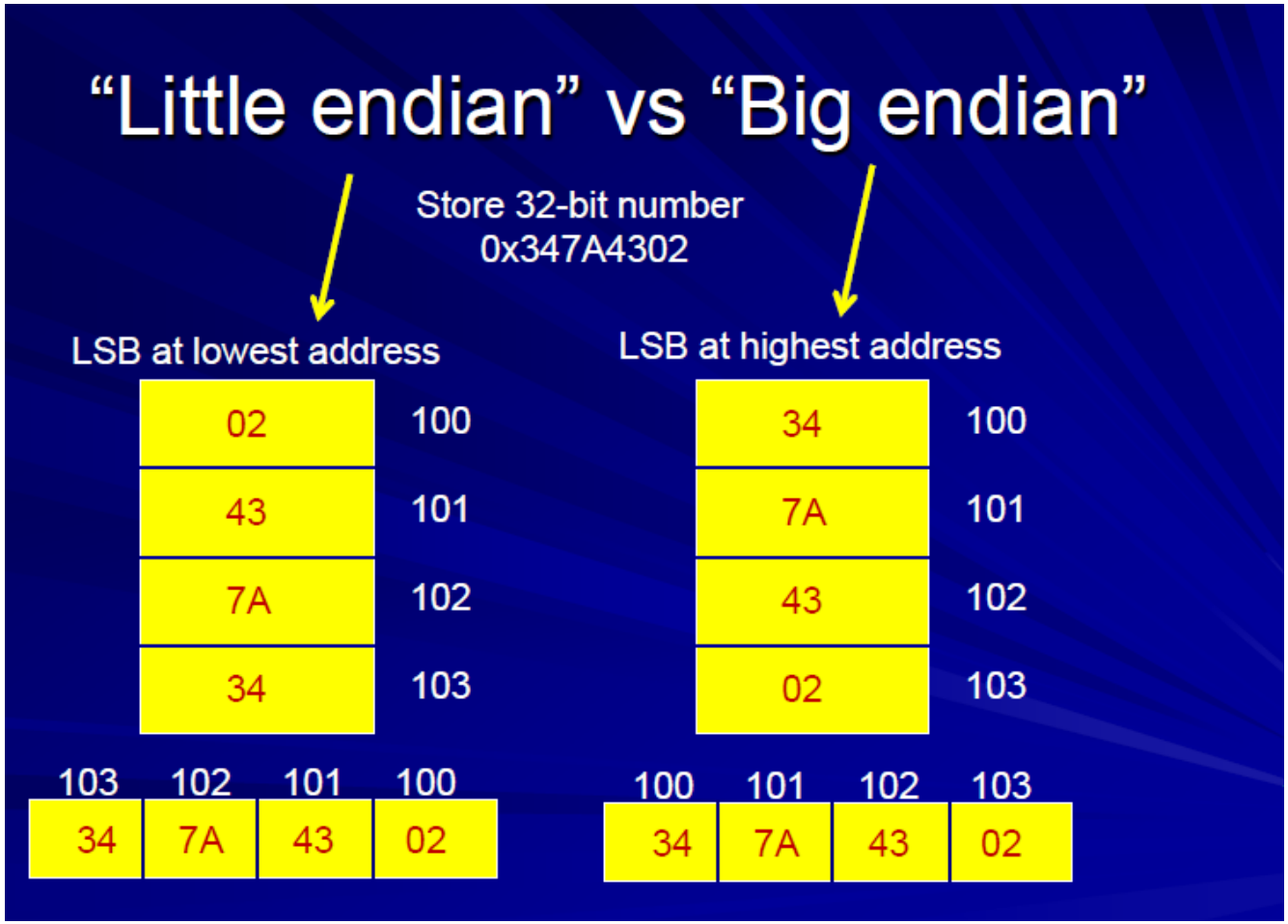
- Not all of the registers are equivalent:
  - E.g., register $0 or $zero always contains the value 0
    - (go ahead, try to change it)

- Other registers have special uses, by convention:
  - E.g., register $sp is used to hold the "stack pointer"

Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian




"Little endian" vs "Big endian"
Store 32-bit number
0x347A4302
LSB at lowest address
02
43
7A
34
100
101
102
103
103 102 101 100
34 7A 43 02
LSB at highest address
34
7A
43
02
100
101
102
103
100 101 102 103
34 7A 43 02

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function* with opcode, tells computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type

R-Type Examples

Assembly Code

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

I-Type

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-Type

I-Type Examples

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

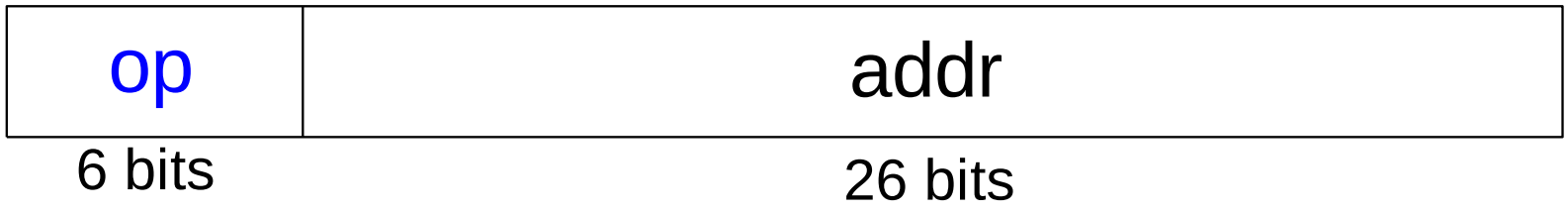
Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Machine Language: J-Type

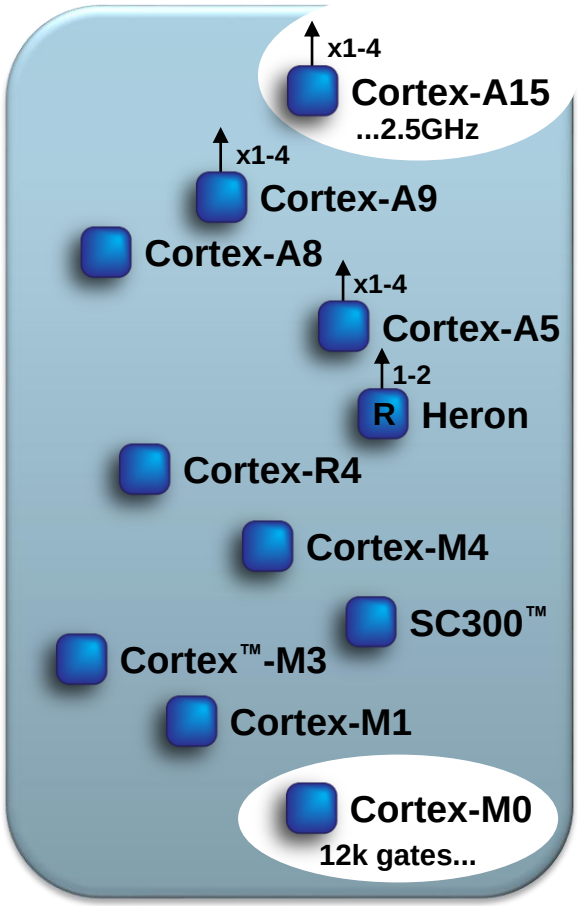
- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

J-Type



ARM Cortex Processors (v7)

- ARM Cortex-**A** family (v7-A):
  - Applications processors for full OS and 3rd party applications
- ARM Cortex-**R** family (v7-R):
  - Embedded processors for real-time signal processing, control applications
- ARM Cortex-**M** family (v7-M):
  - Microcontroller-oriented processors for MCU and SoC applications



Cortex family

Cortex-A8

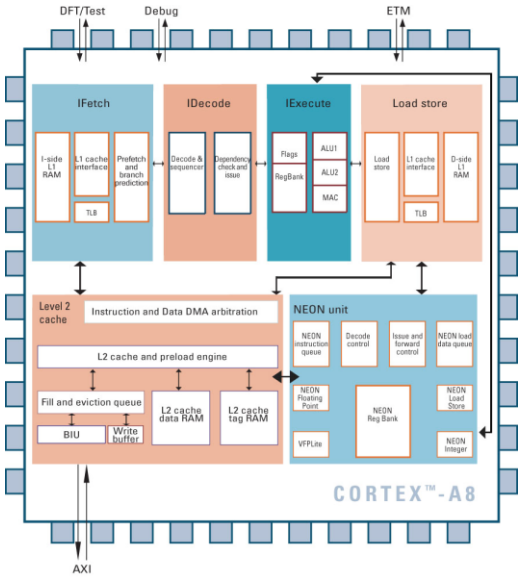
- Architecture v7A
- MMU
- AXI
- VFP & NEON support

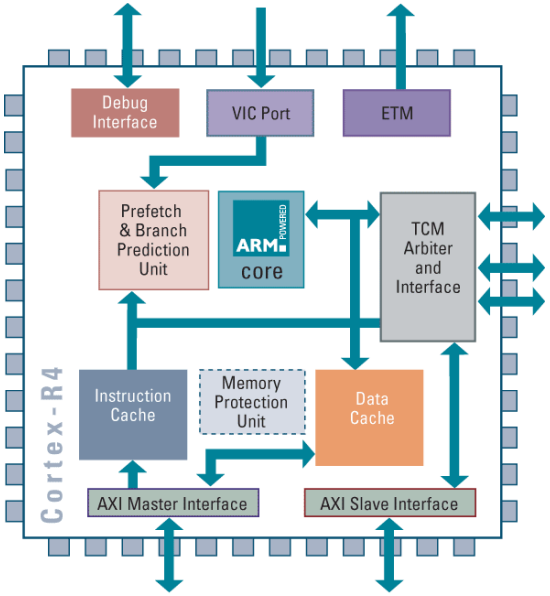
Cortex-R4

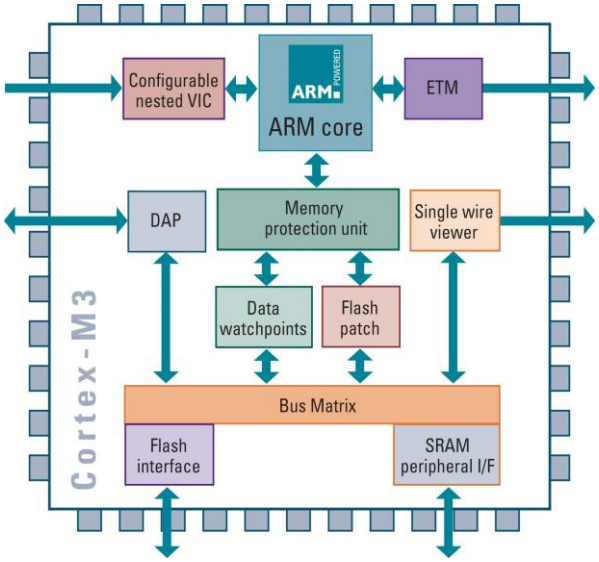
- Architecture v7R
- MPU (optional)
- AXI
- Dual Issue

Cortex-M3

- Architecture v7M
- MPU (optional)
- AHB Lite & APB







Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Regularity supports design simplicity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** mnemonic – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

- Registers
- Constants (also called *immediates*)
- Memory

Operands: Registers

- ARM has 16 registers
- Registers are faster than memory
- Each register is 32 bits
- ARM is called a “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

Data Sizes and Instruction Sets

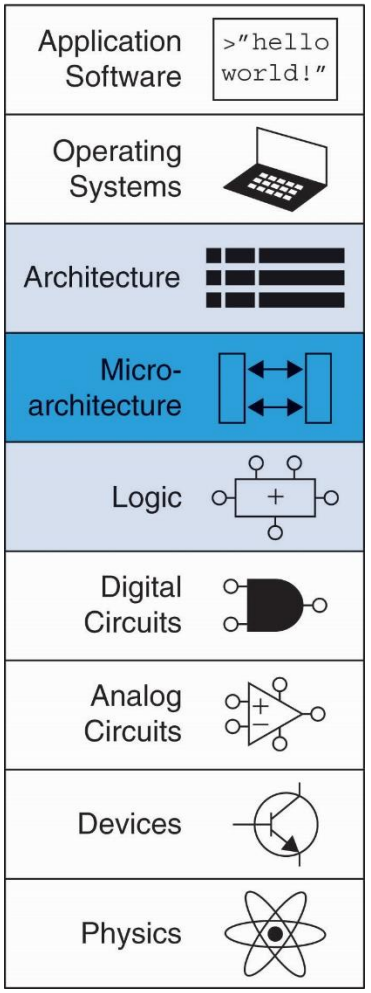
- **The ARM is a 32-bit architecture.**
- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
- **Jazelle cores can also execute Java bytecode**

ARM Register Set

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

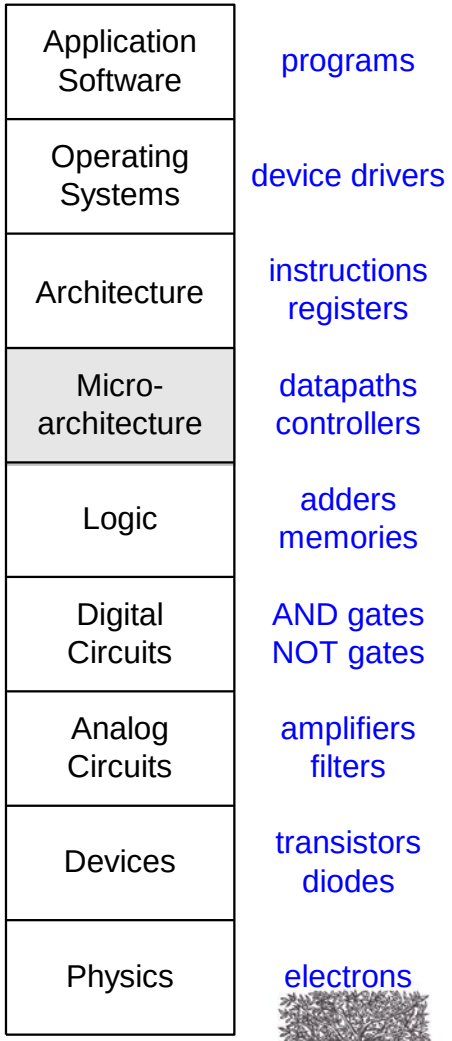
Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Exceptions**
- **Advanced Microarchitecture**



Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals



Microarchitecture

- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

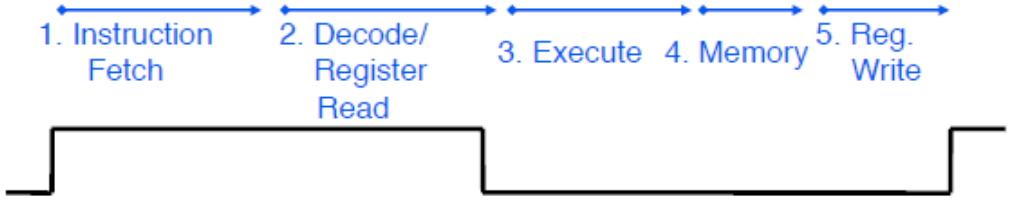
Microarchitecture

1. Analyze instruction set architecture (ISA) ⟹ datapath requirements
   - meaning of each instruction is given by the *data transfers (register transfers)*
   - datapath must include storage element for ISA registers
   - datapath must support each data transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the data transfer.
5. Assemble the control logic.

Microarchitecture

CPU clocking (1/2)

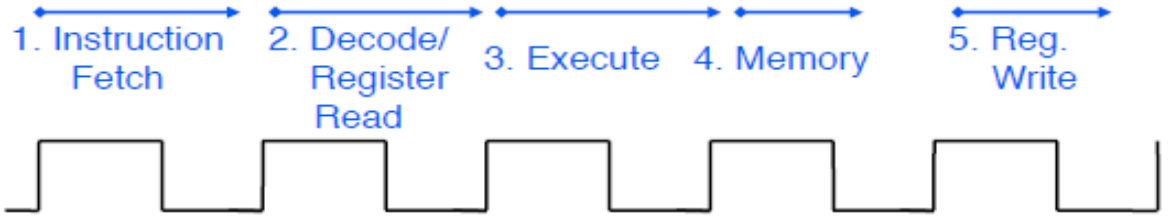
- Single Cycle CPU: All stages of an instruction are completed within one ***long*** clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.



Microarchitecture

CPU clocking (2/2)

- Multiple-cycle CPU: Only one stage of instruction per clock cycle.
  - The clock is made as long as the slowest stage.

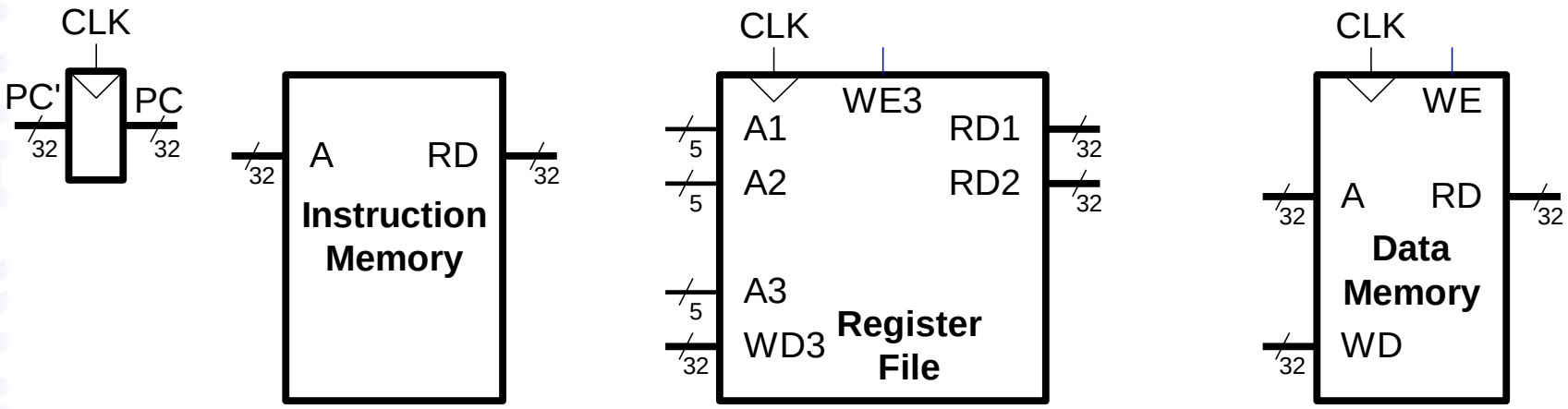


Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).

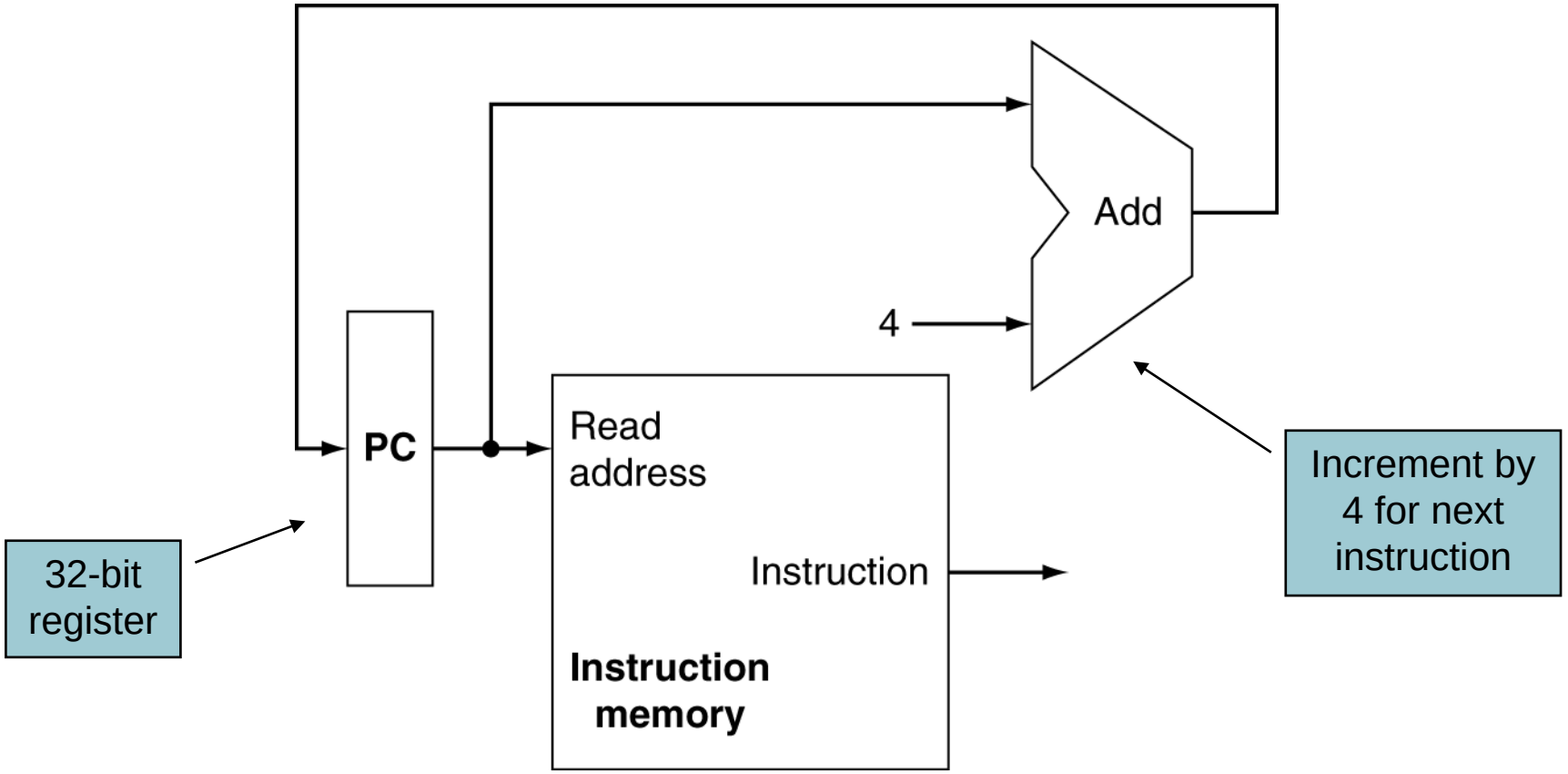
Architectural State

- Determines everything about a processor:
  - PC
  - 32 registers
  - Memory

MIPS State Elements

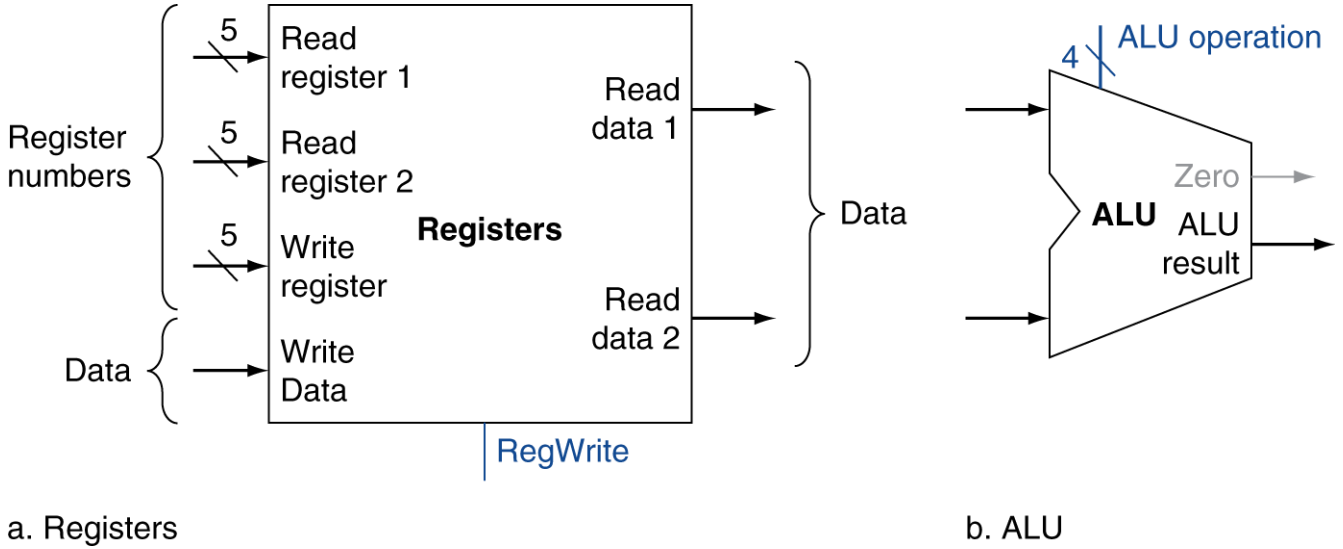


Instruction Fetch



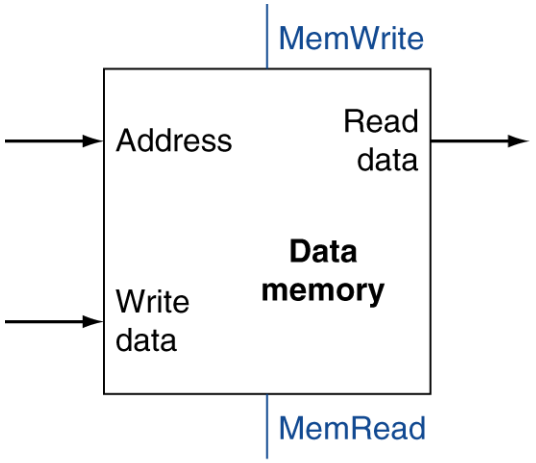
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

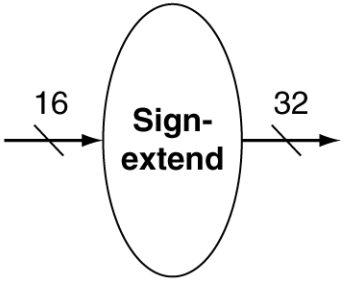


Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

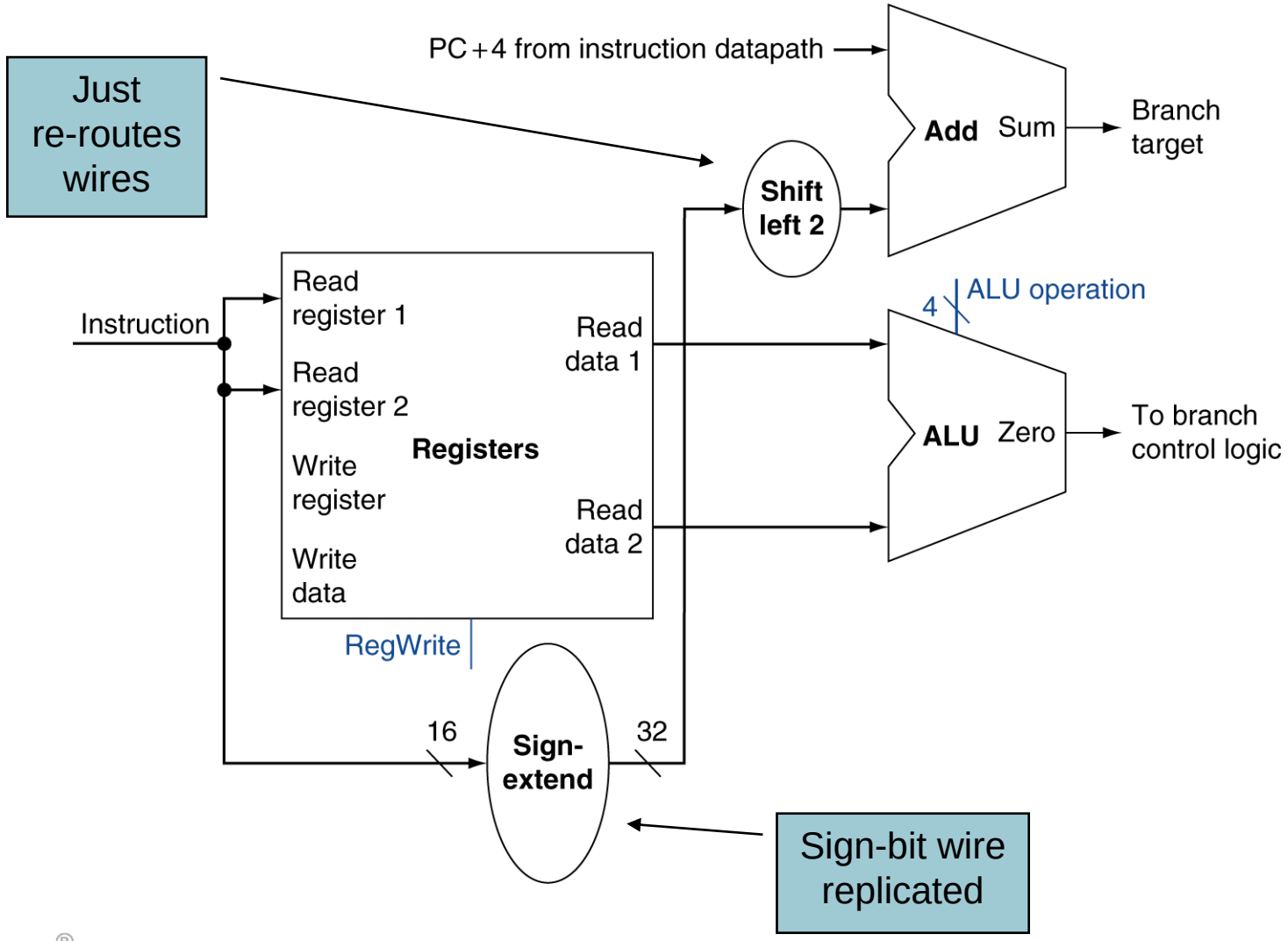


b. Sign extension unit

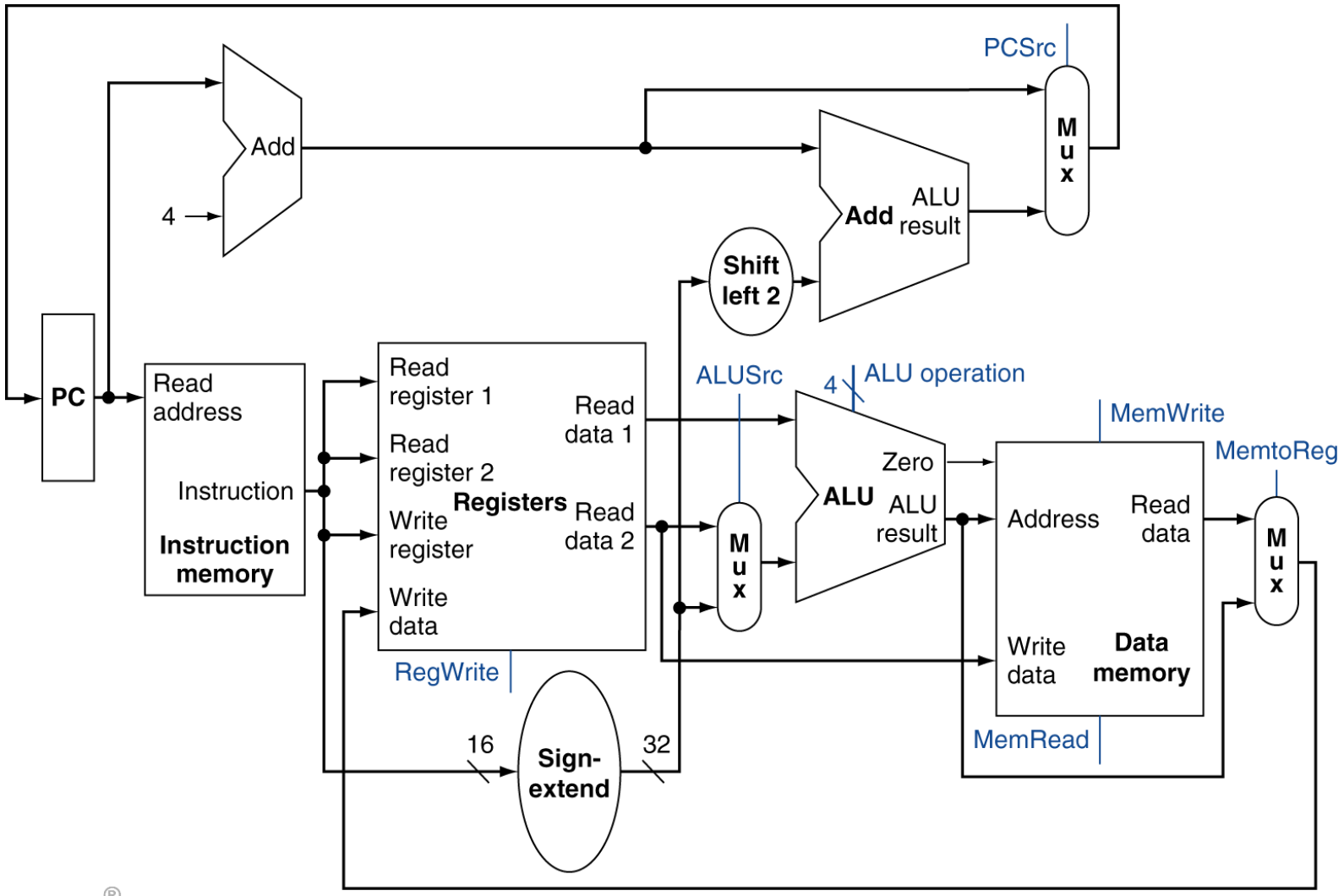
Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions



Full Datapath



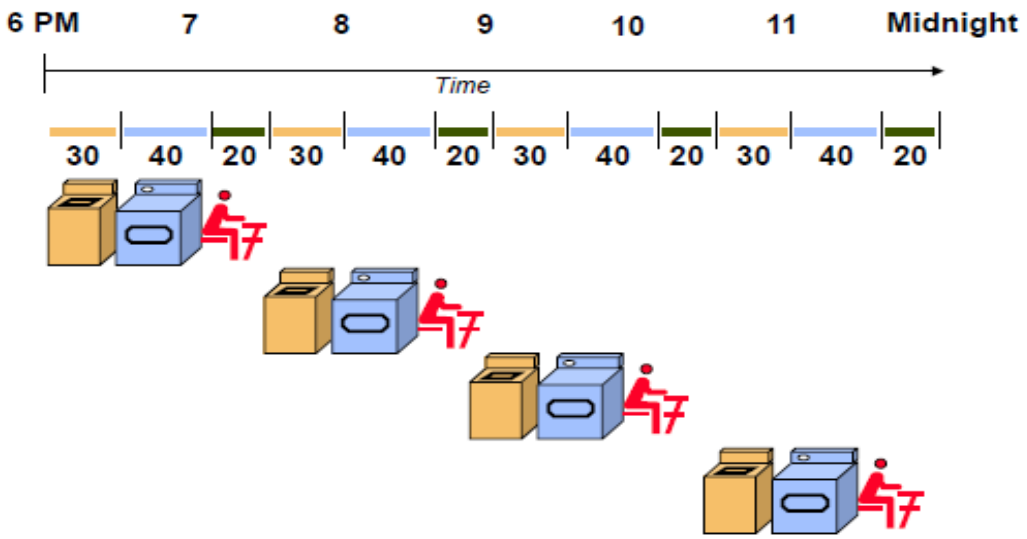
Pipelining

- Assuming you've got:
  - One washer (takes 30 minutes)
  - One drier (takes 40 minutes)
  - One "folder" (takes 20 minutes)

- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  - How long does 4 loads take?

Pipelining

The slow way

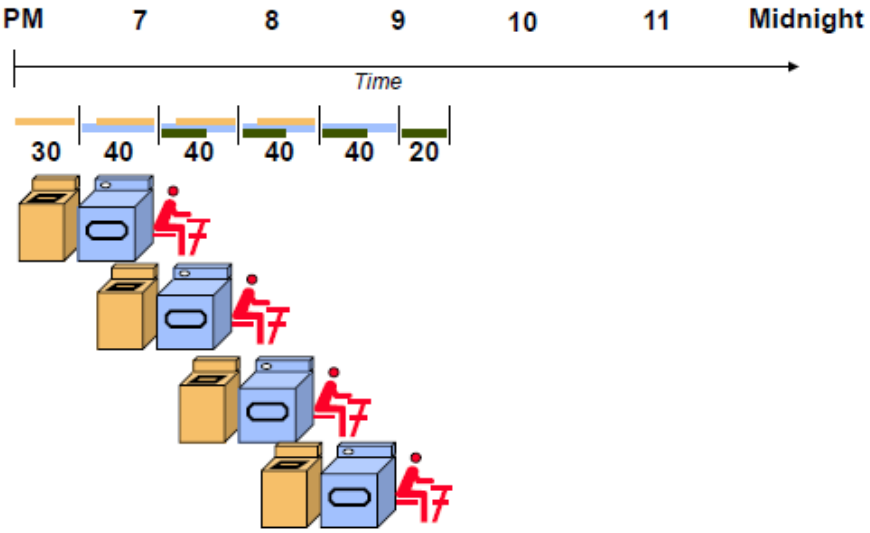


- If each load is done sequentially it takes 6 hours

Pipelining

Laundry Pipelining

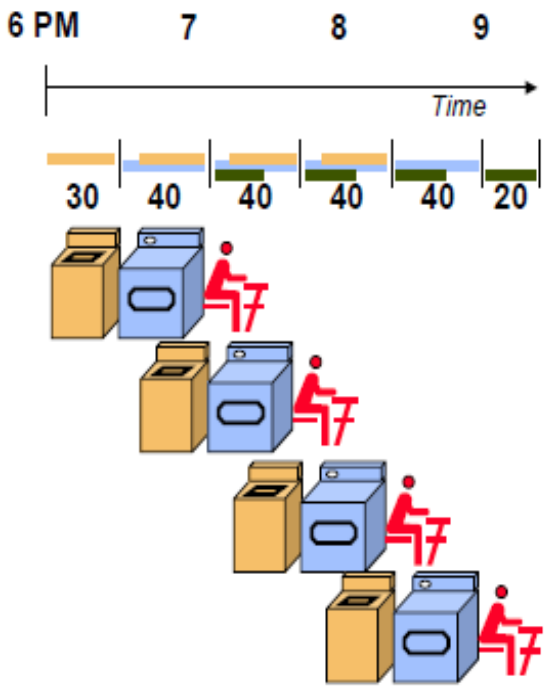
- Start each load as soon as possible
  - Overlap loads



- Pipelined laundry takes 3.5 hours

Pipelining

Pipelining Lessons



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

Pipelining

Instruction execution review

- Executing a MIPS instruction can take up to five steps.

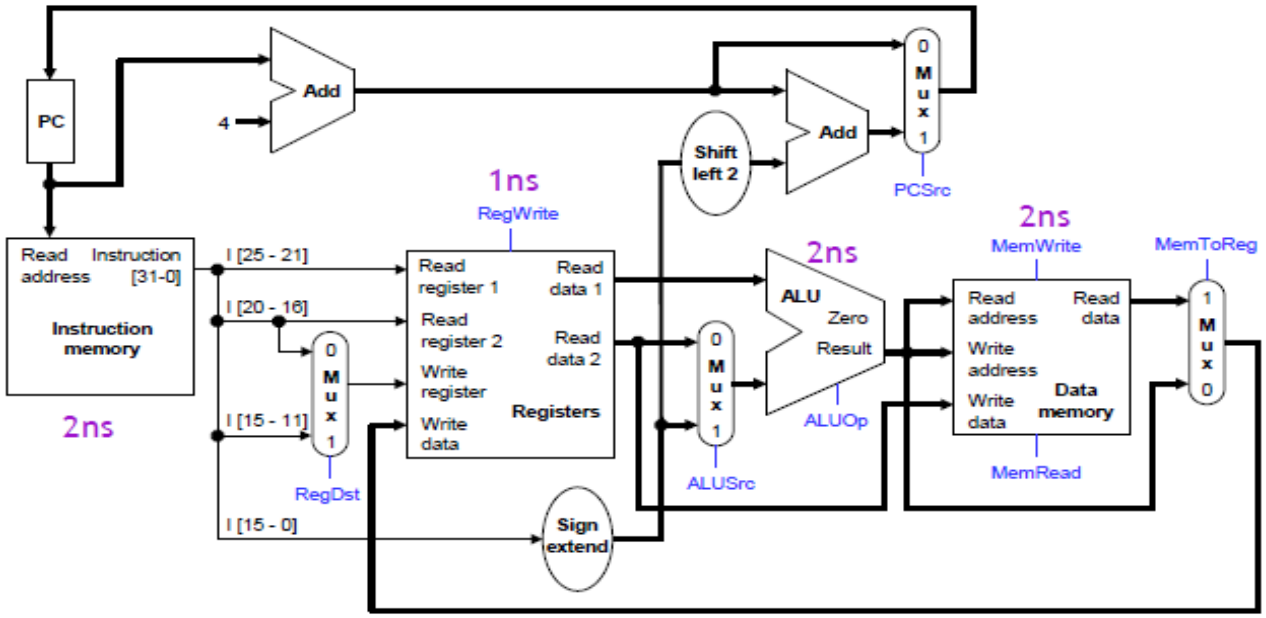
Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Pipelining

Single-cycle datapath diagram

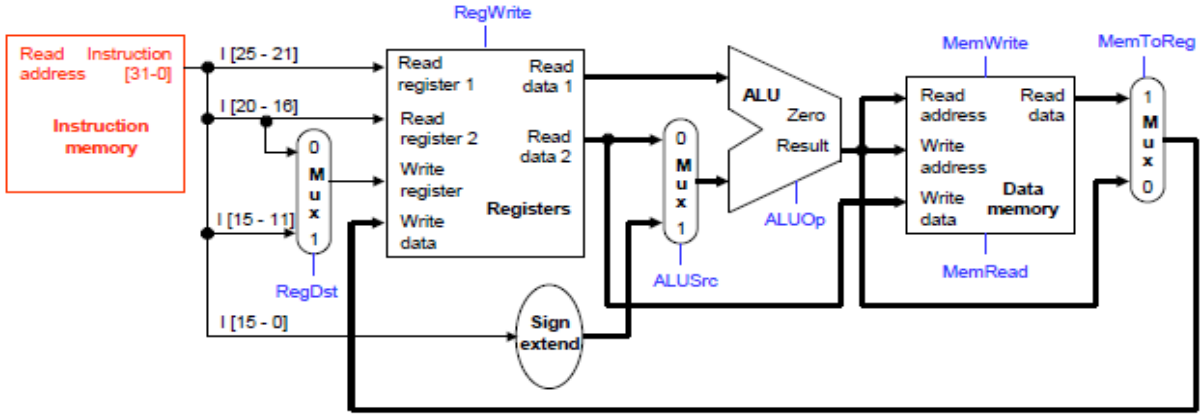


- How long does it take to execute each instruction?

Pipelining

Review: Instruction Fetch (IF)

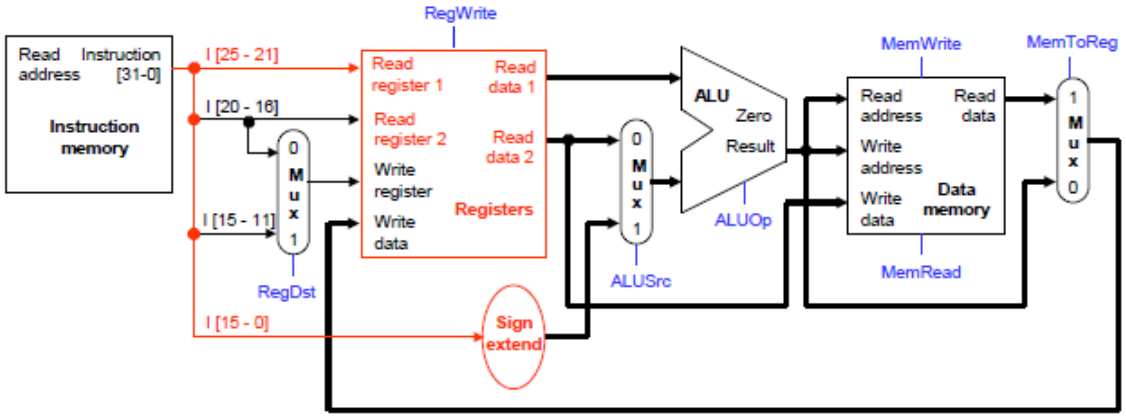
- Let's quickly review how lw is executed in the single-cycle datapath.
- We'll ignore PC incrementing and branching for now.
- In the Instruction Fetch (IF) step, we read the instruction memory.



Pipelining

Instruction Decode (ID)

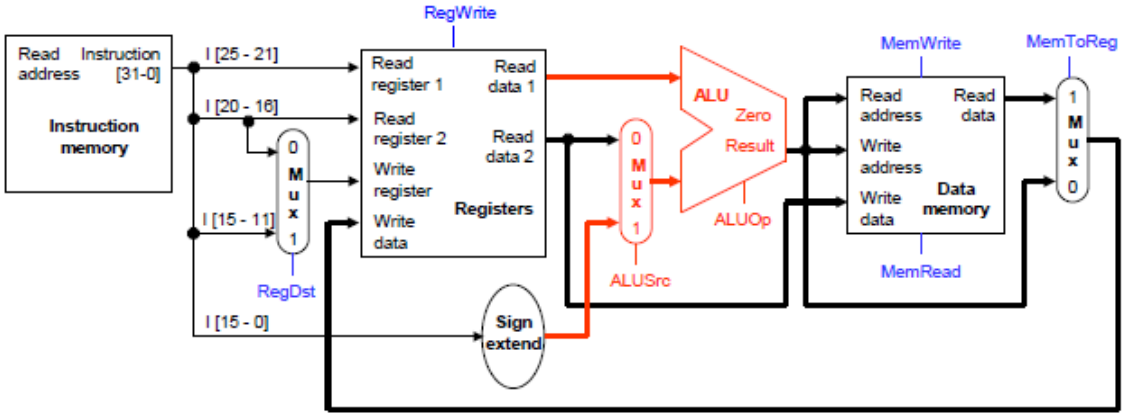
- The Instruction Decode (ID) step reads the source register from the register file.



Pipelining

Execute (EX)

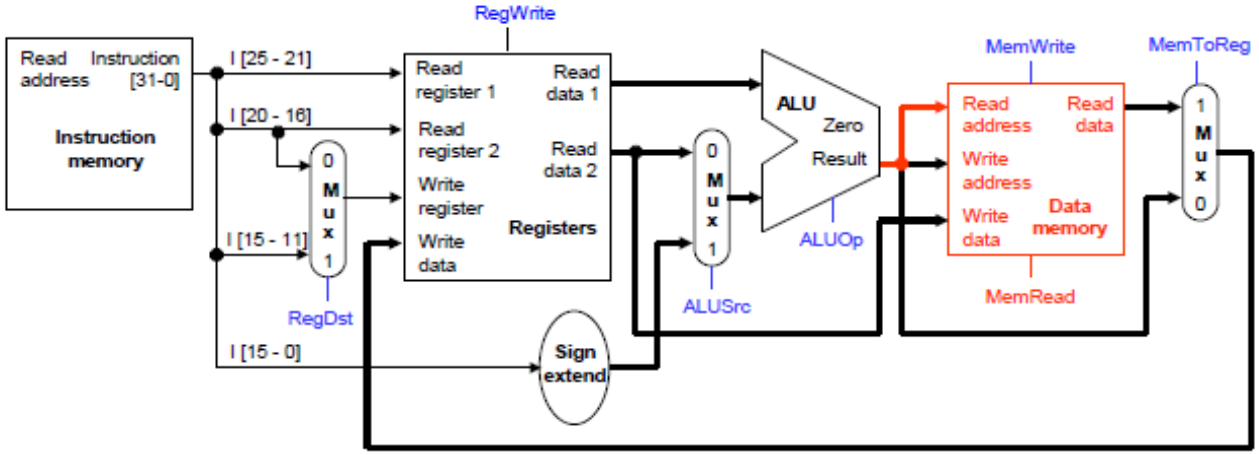
- The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.



Pipelining

Memory (MEM)

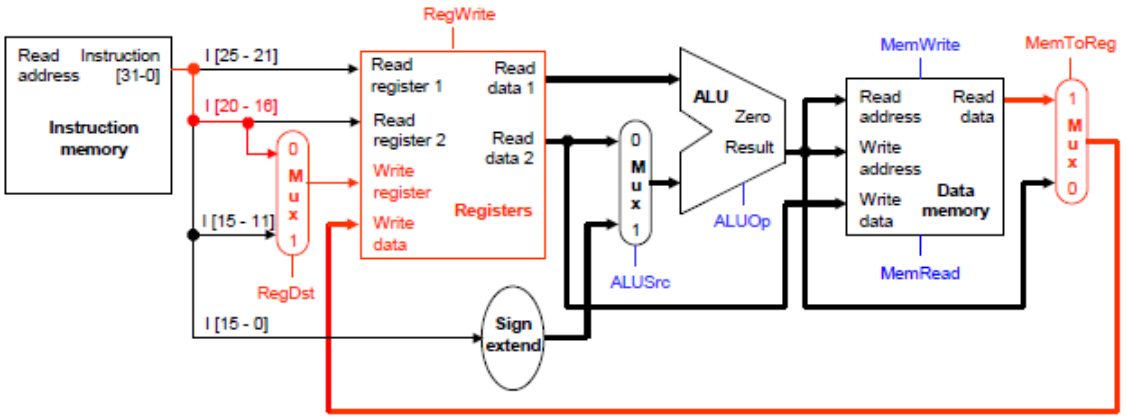
- The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.



Pipelining

Writeback (WB)

- Finally, in the Writeback (WB) step, the memory value is stored into the destination register.



Pipelining

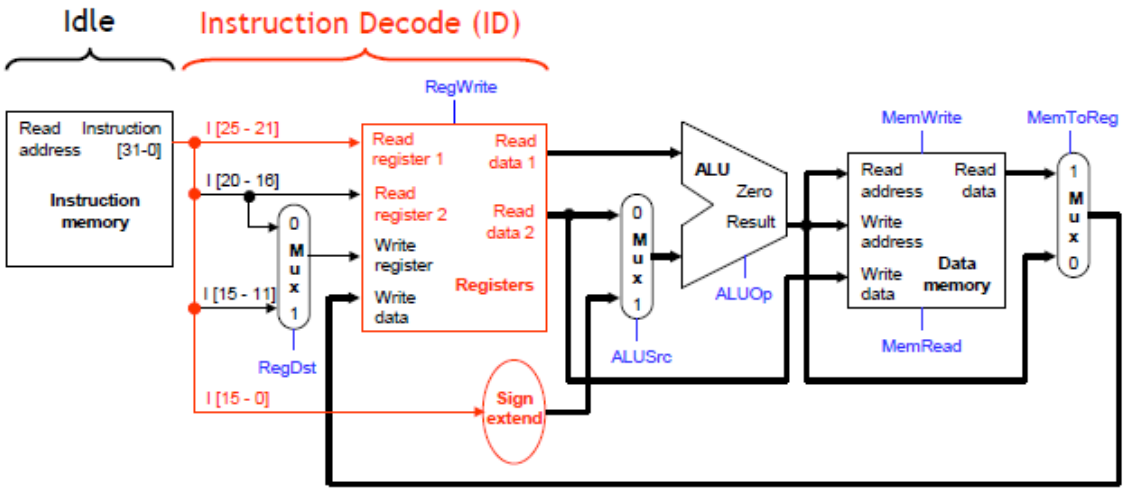
A bunch of lazy functional units

- Notice that each execution step uses a different functional unit.
- In other words, the main units are idle for most of the 8ns cycle!
  - The instruction RAM is used for just 2ns at the start of the cycle.
  - Registers are read once in ID (1ns), and written once in WB (1ns).
  - The ALU is used for 2ns near the middle of the cycle.
  - Reading the data memory only takes 2ns as well.
- That's a lot of hardware sitting around doing nothing.

Pipelining

Putting those slackers to work

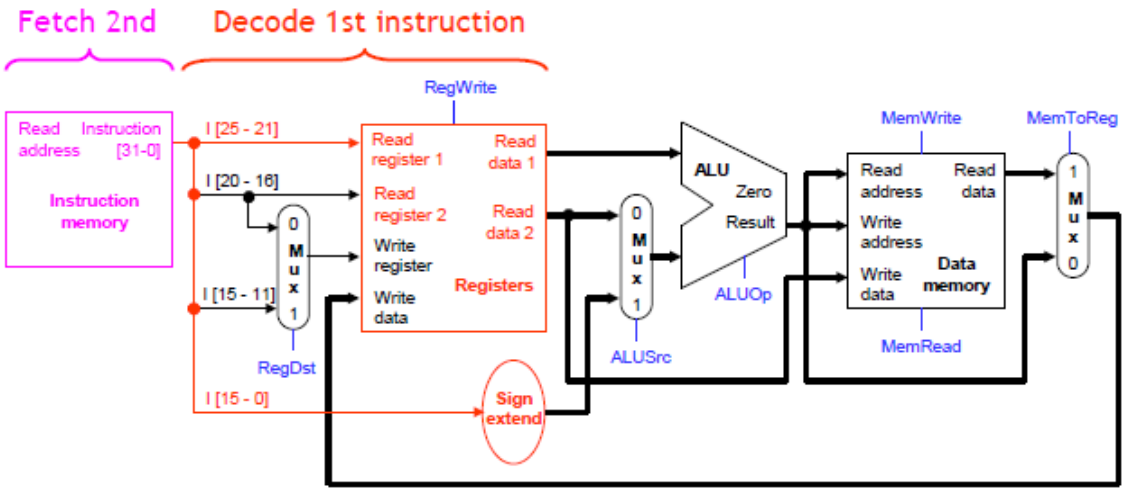
- We shouldn't have to wait for the entire instruction to complete before we can re-use the functional units.
- For example, the instruction memory is free in the Instruction Decode step as shown below, so...



Pipelining

Decoding and fetching together

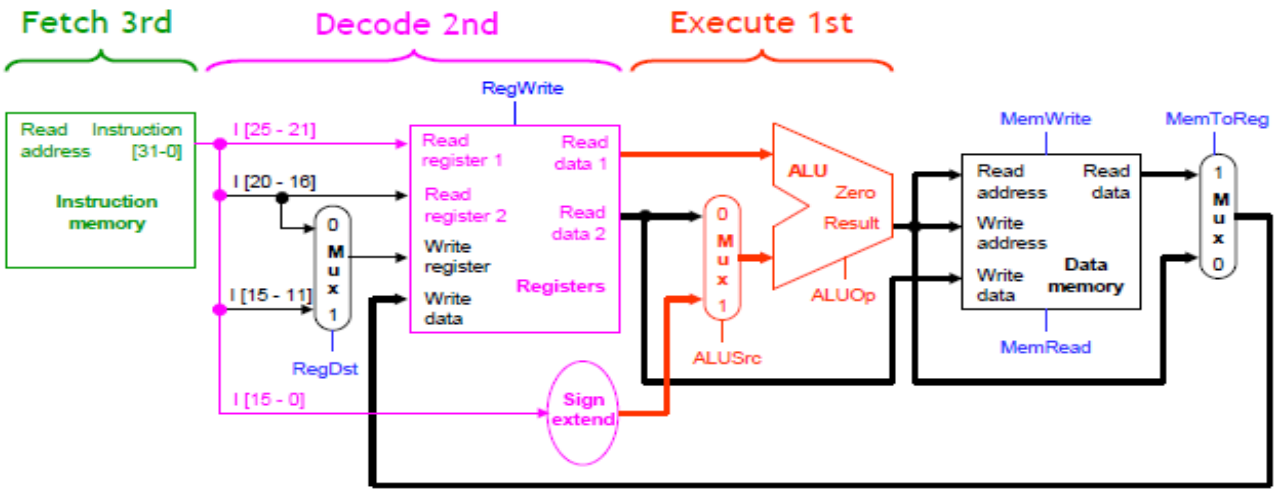
- Why don't we go ahead and fetch the *next* instruction while we're decoding the first one?



Pipelining

Executing, decoding and fetching

- Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- But now the instruction memory is free again, so we can fetch the third instruction!



Pipelining

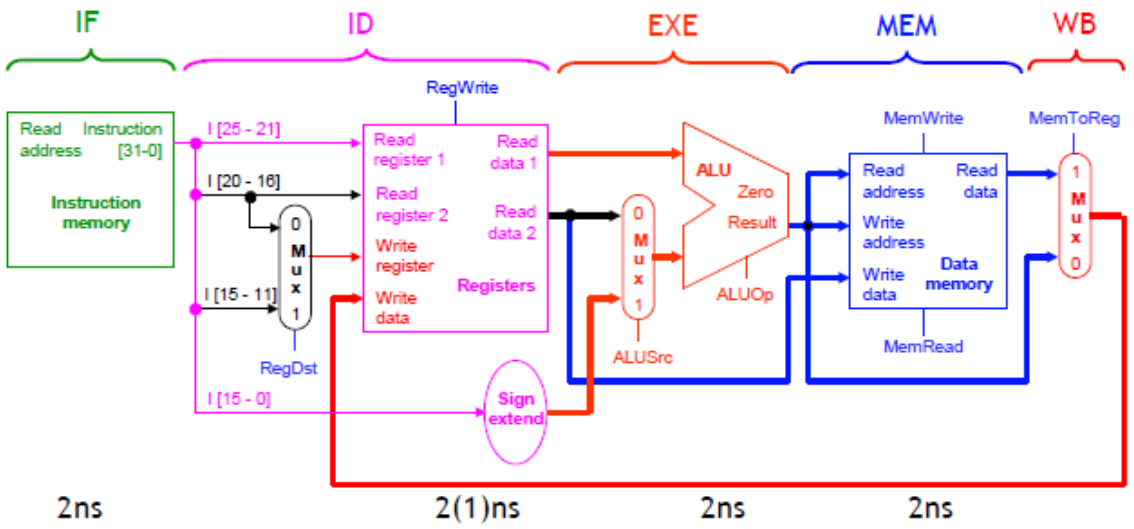
Making Pipelining Work

- We'll make our pipeline 5 stages long, to handle load instructions
  - Stages are: IF, ID, EX, MEM, and WB
- We want to support executing 5 instructions simultaneously: one in each stage.

Pipelining

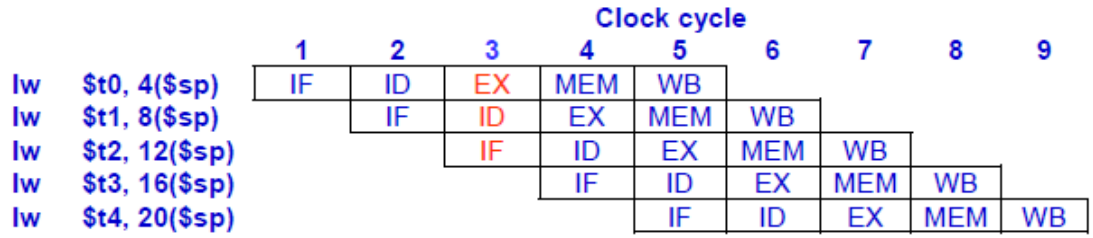
Break datapath into 5 stages

- Each stage has its own functional units.
- Each stage can execute in 2ns



Pipelining

Pipelining Loads

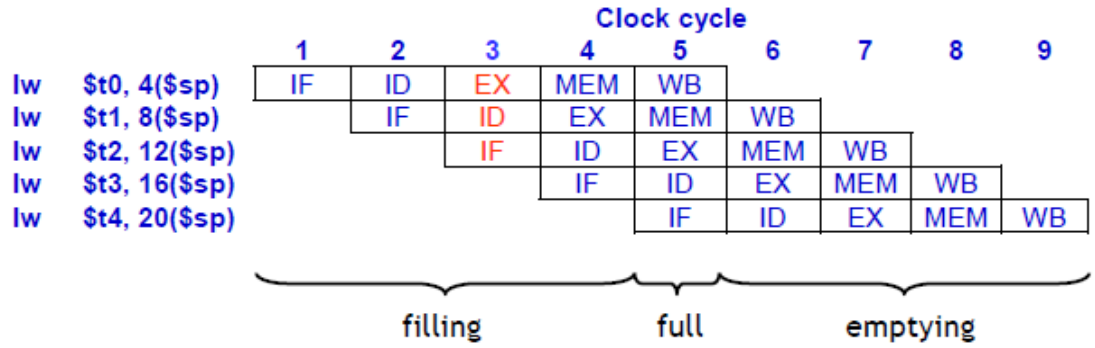


- **A pipeline diagram shows the execution of a series of instructions.**
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

- **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**
  - The "lw $t0" instruction is in its Execute stage.
  - Simultaneously, the "lw $t1" is in its Instruction Decode stage.
  - Also, the "lw $t2" instruction is just being fetched.

Pipelining

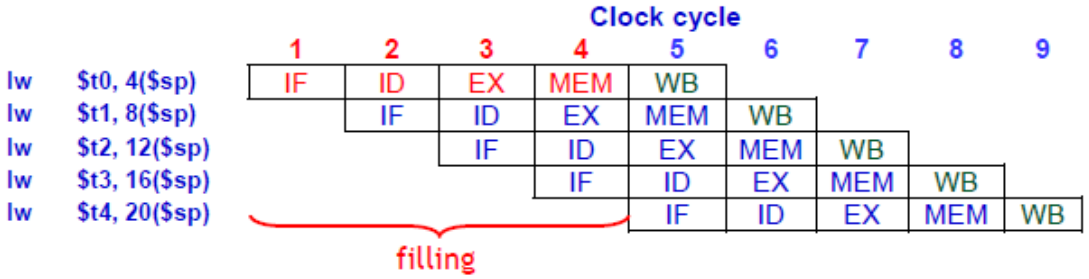
Pipelining terminology



- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.

Pipelining

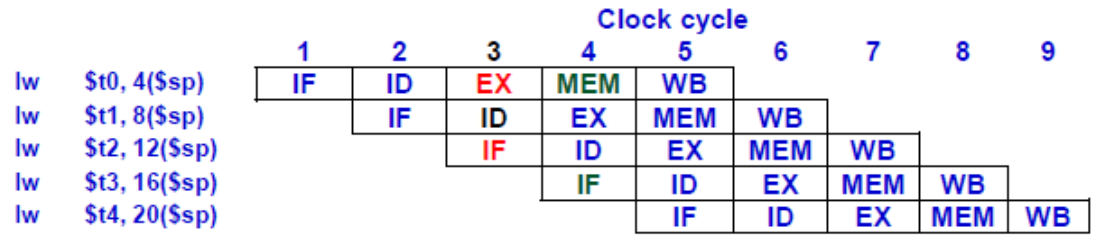
Pipelining Performance



- **Execution time on ideal pipeline:**
  - time to fill the pipeline + one cycle per instruction
  - How long for N instructions?
- **Compared to single-cycle design, how much faster is pipelining for N=1000 ?**

Pipelining

Pipeline Datapath: Resource Requirements



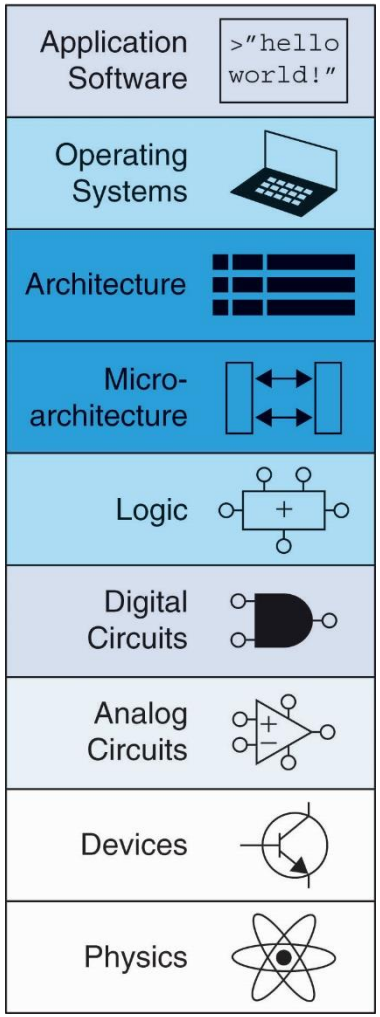
- We need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.
- What does that mean for our hardware?

Pipelined MIPS Processor

- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Chapter 8 :: Topics

- **Introduction**
- **Memory System Performance Analysis**
- **Caches**
- **Virtual Memory**
- **Memory-Mapped I/O**
- **Summary**



Good Luck

Become Super Heroes of Digital Design