UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 4: Signed Numbers, Hexadecimal, Instructions, and Endianess

Taylor Johnson

Important Concepts from Previous Lectures

- How do computers compute?
- Binary to decimal, decimal to binary, ASCII
- Structured computers
  - Multilevel computers and architectures
  - Abstraction layers
- Performance metrics
  - Clock rates, cycle time/period, CPI, response time, throughput

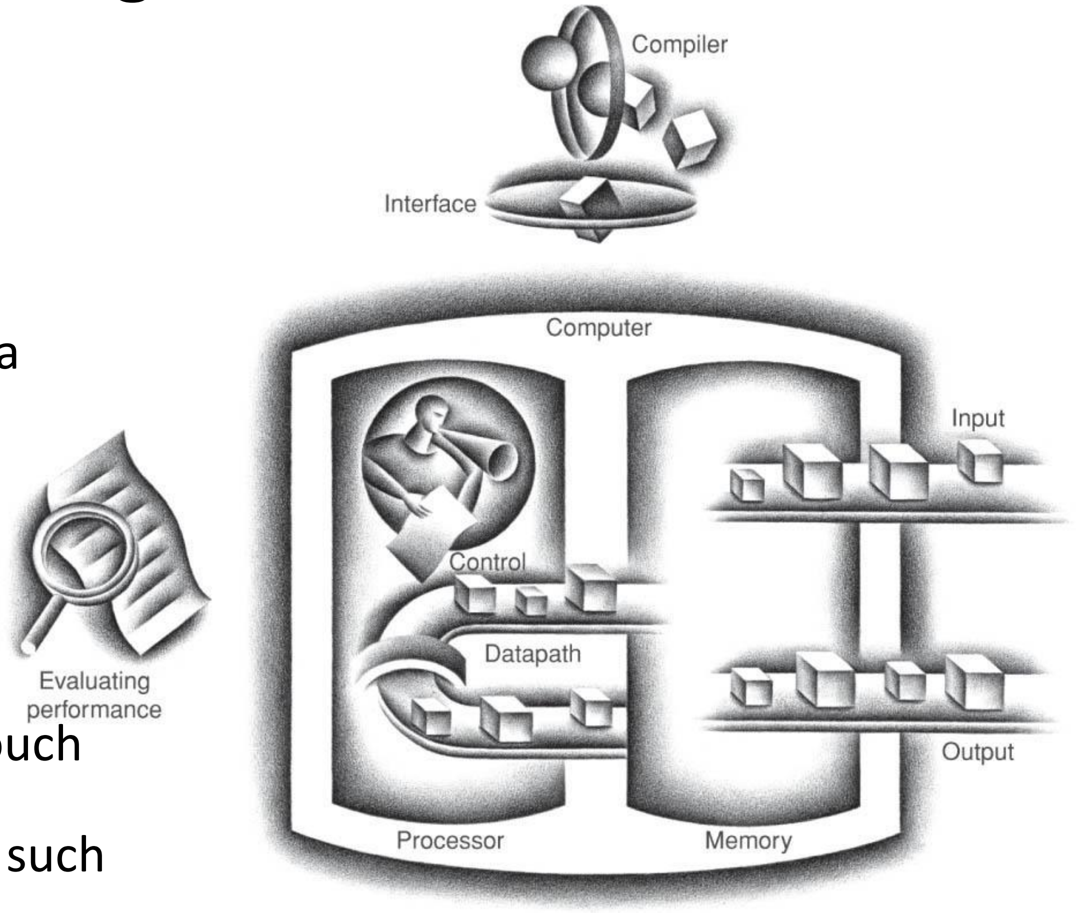
Announcements and Outline

- Quiz 2 on Blackboard site (due 11:59PM Friday)
  - Review binary arithmetic, Boolean operations, and representing signed and unsigned numbers in binary
- Homework 1 due Thursday
  - Read chapter 1
- Homework 2 assigned Thursday
  - Start reading chapter 2 (ARM version on Blackboard site)

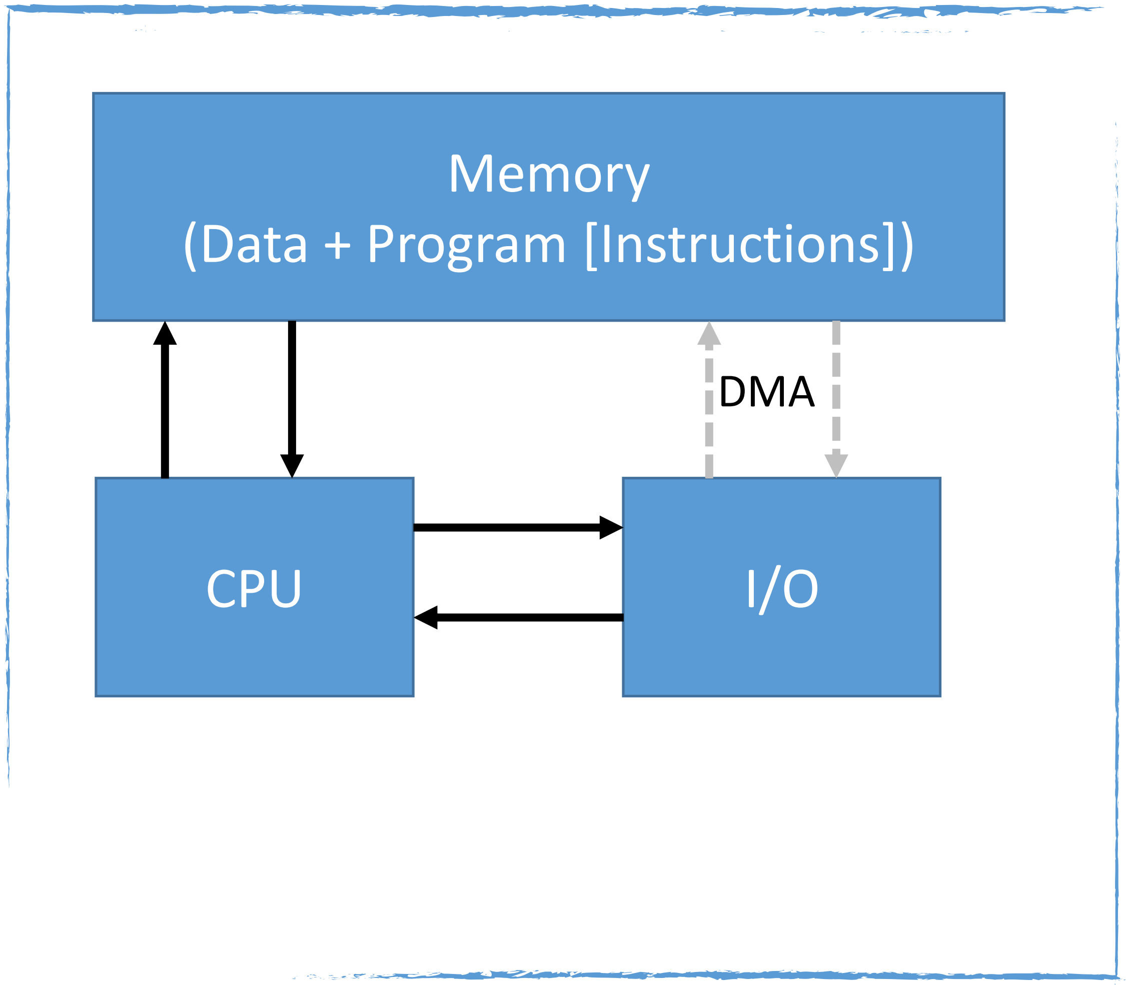
- Review from last time / Chapter 1
  - Performance metrics
- Signed vs. Unsigned Numbers (Two's Complement)
- Instructions: the Language of the Computer

Review: Computer Organization Overview

- CPU
  - Executes instructions
- Memory
  - Stores programs and data
- Buses
  - Transfers data
- Storage
  - Permanent
- I/O devices
  - Input: keypad, mouse, touch
  - Output: printer, screen
  - Both (input and output), such as:
    - USB, network, Wifi, touch screen, hard drive

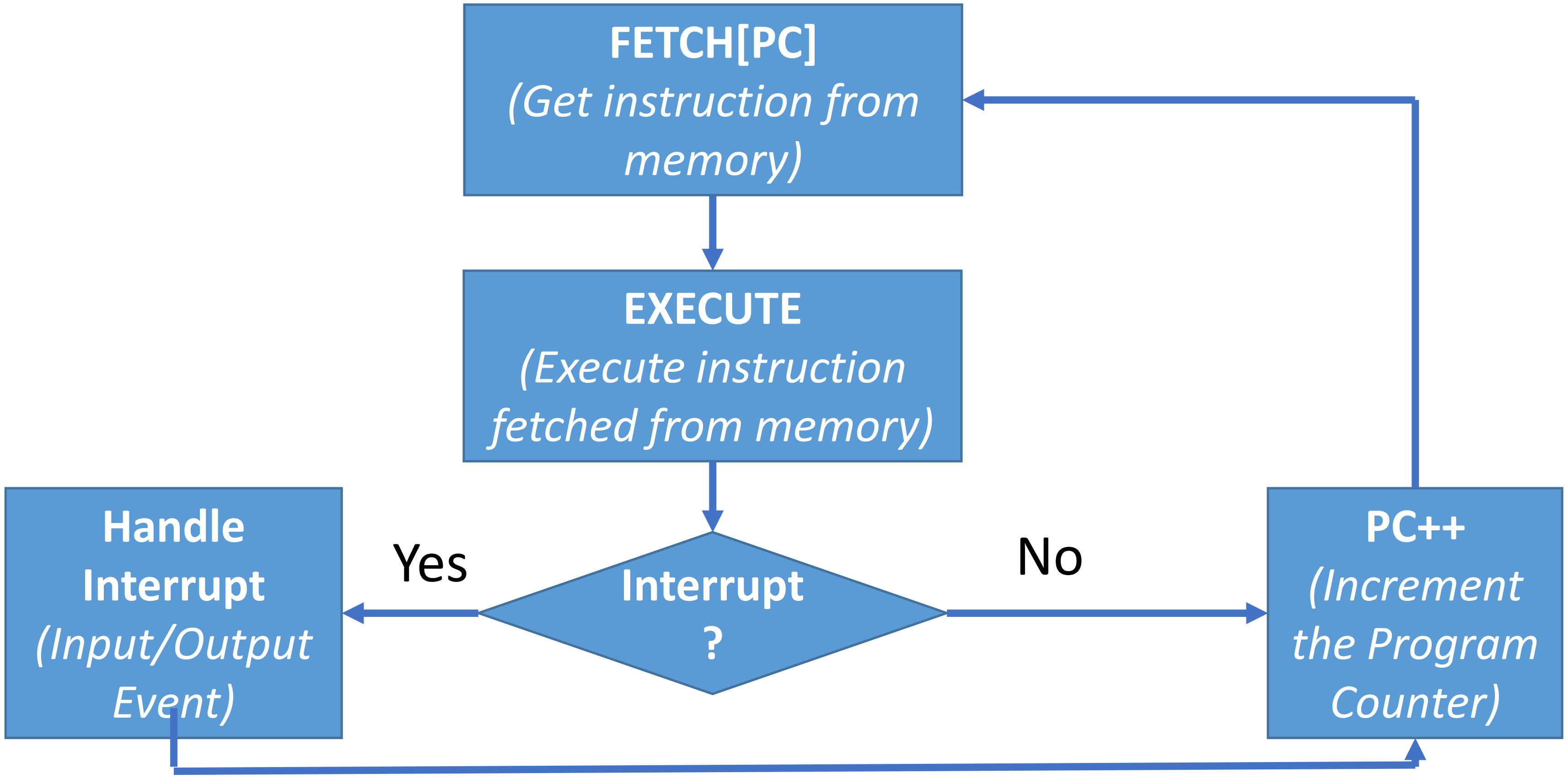


Review: Von Neumann Architecture



- Both data and program stored in memory
- Allows the computer to be "re-programmed"
- Input/output (I/O) goes through CPU
- I/O part is not representative of modern systems (direct memory access [DMA])
- Memory layout is representative of modern systems

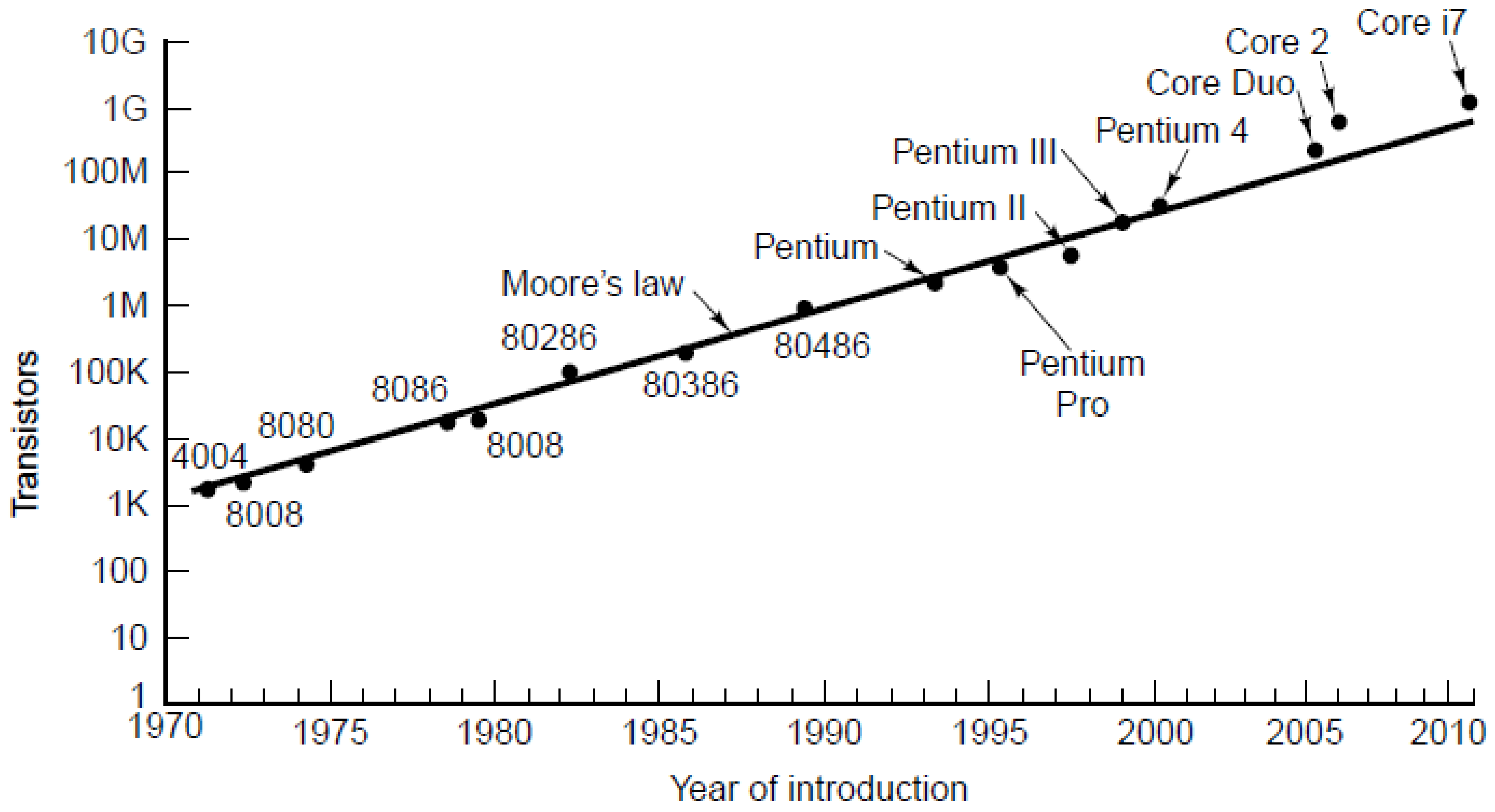
Review: Abstract Processor Execution Cycle


FETCH[PC]
(Get instruction from memory)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt
(Input/Output Event)
PC++
(Increment the Program Counter)

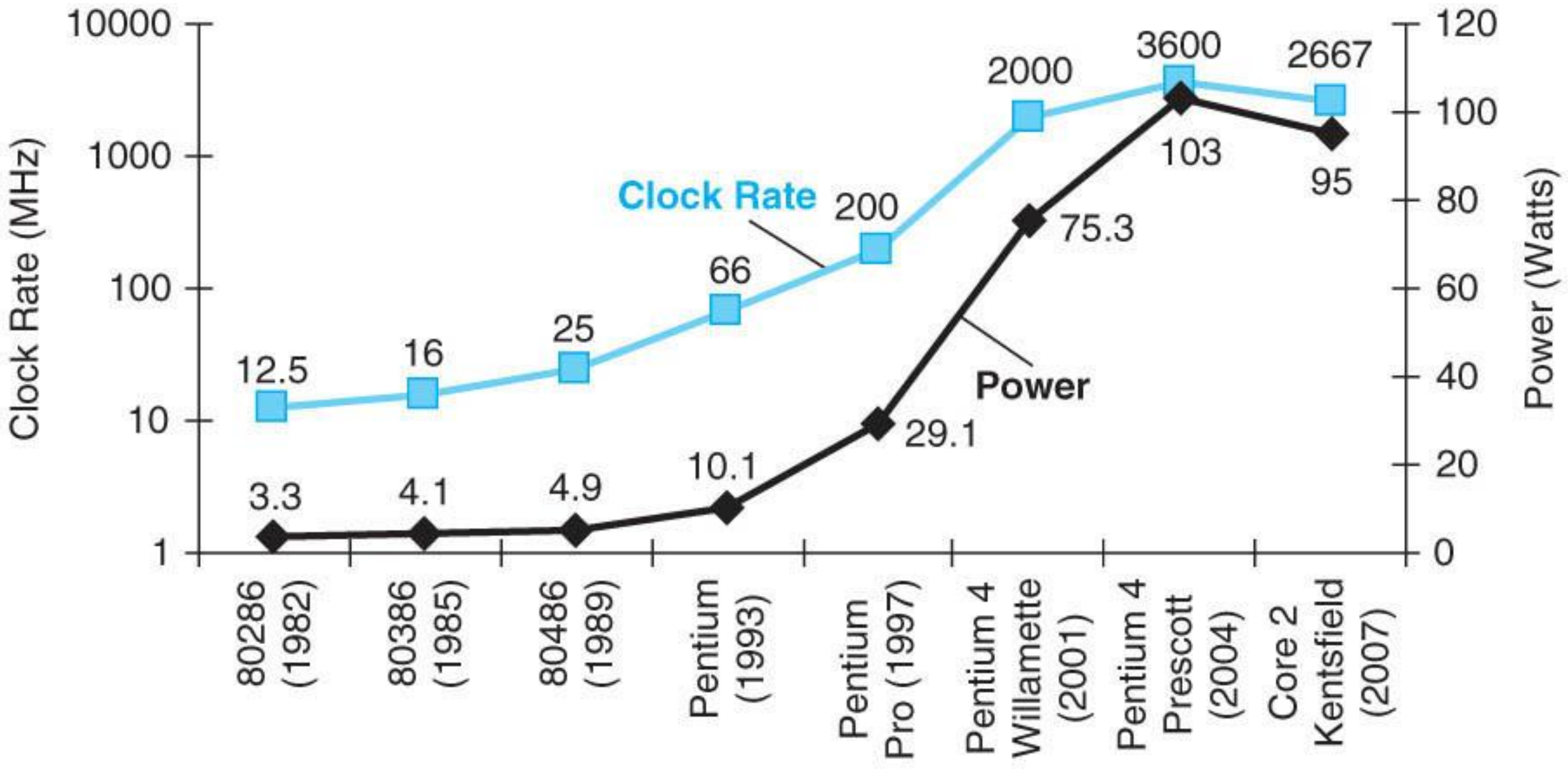
Review: Units of Memory

- One bit (binary digit): the smallest amount of information that we can store:
  - Either a 1 or a 0
  - Sometimes refer to 1 as high/on/true, 0 as low/off/false
- One byte = 8 bits
  - Can store a number from 0 to 255
- Kilobyte (KB): $10^3 = 1000$ bytes
- Kibibyte (KiB): $2^{10} = 1024$ bytes
- Kilobit: (Kb): $10^3 = 1000$ bits (125 bytes)
- Kibibit: (Kib): $2^{10} = 1024$ bits (128 bytes)

Review: Moore's Law for the Intel Family



Review: The Power Wall



Review: Relative Performance

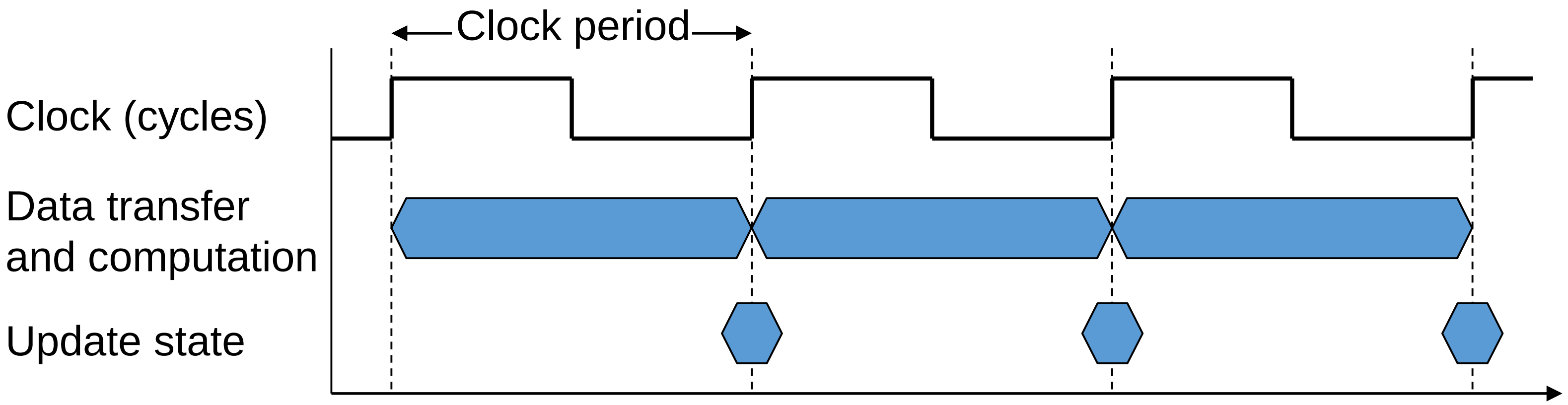
- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Review: CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

Review: Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\begin{aligned}\text{CPU Time} &= \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time} \\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}\end{aligned}$$

- Instruction Count for a program = number of instructions in program
  - Determined by program, ISA and compiler
- Average cycles per instruction (CPI) = number of cycles to execute an instruction (on average)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

Review: Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

Review: Chapter 1 Summary

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

Announcements and Outline

- Quiz 2 on Blackboard site (due 11:59PM Friday)
  - Review binary arithmetic, Boolean operations, and representing signed and unsigned numbers in binary
- Homework 1 due Thursday
  - Read chapter 1
- Homework 2 assigned Thursday
  - Start reading chapter 2 (ARM version on Blackboard site)

- Review from last time / Chapter 1
  - Performance metrics
- Signed vs. Unsigned Numbers (Two’s Complement)
- Instructions: the Language of the Computer

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- –(–2n – 1) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Two's Complement Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
  - Representation called one's complement

$$x + \overline{x} = 1111\ldots111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits (also called a nibble or nybble) per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0xECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Announcements and Outline

- Quiz 2 on Blackboard site (due 11:59PM Friday)
  - Review binary arithmetic, Boolean operations, and representing signed and unsigned numbers in binary
- Homework 1 due Thursday
  - Read chapter 1
- Homework 2 assigned Thursday
  - Start reading chapter 2 (ARM version on Blackboard site)

- Review from last time / Chapter 1
  - Performance metrics
- Signed vs. Unsigned Numbers (Two's Complement)
- Instructions: the Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
  - Will discuss a few in this course, primarily will focus on ARM for assignments
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

MIPS and ARM Instruction Sets

- MIPS
  - Used as examples throughout the book
  - Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Large share of embedded core market
    - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - Typical of many modern ISAs
    - See MIPS Reference Data tear-out card, and Appendixes B and E
- ARM
  - Commercially much more successful (nearly every phone)
  - Similar to MIPS
  - ARM version of chapters on Blackboard
  - Use this for programming assignments

Arithmetic Operations

- Add and subtract, three ***operands***
  - ***Operand:*** *quantity on which an operation is performed*
  - Two sources and one destination

```
add a, b, c  # a updated to b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

- Compiled ARM code:

```
add r0, g, h   # temp r0 = g + h
add r1, i, j   # temp r1 = i + j
sub f, r0, r1  # f = t0 - t1
```

- Notice: registers "=" variables

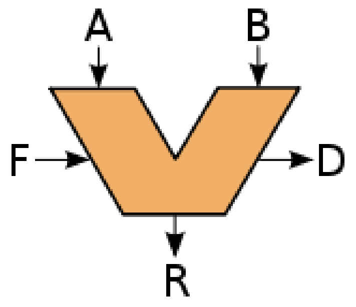
Some Processor Components

CPU

Register File

- Program Counter (PC)
- Instruction Register (IR)
- General Purpose Registers
  - Word size
  - Typically 16-32 of these
  - PC sometimes one of these
- Floating Point Registers

Arithmetic logic unit (ALU)



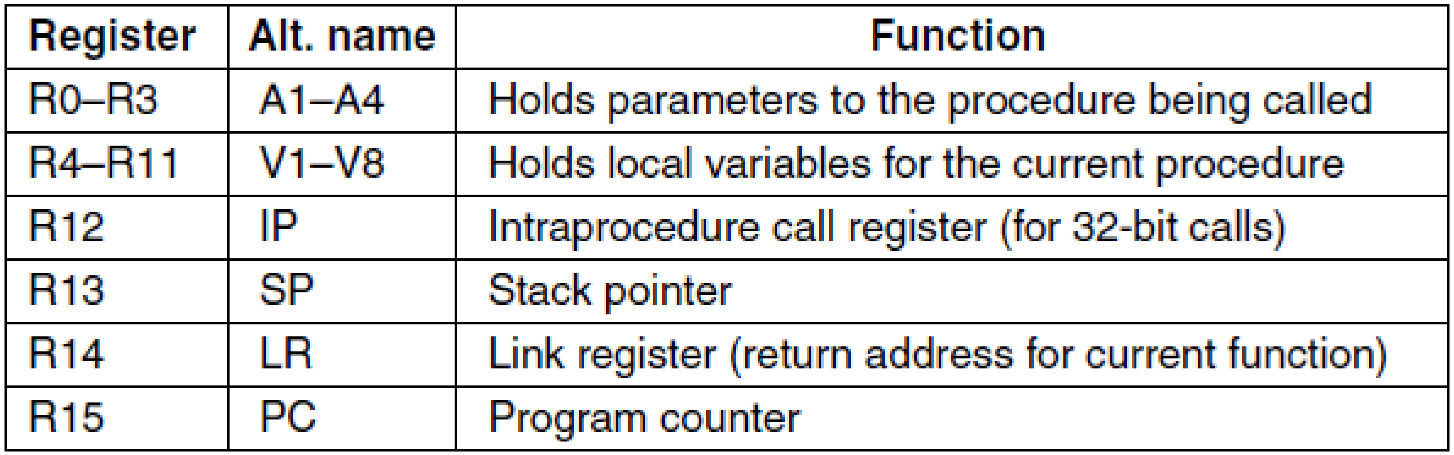
Floating Point Unit (FPU)

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - \$t0, \$t1, …, \$t9 for temporary values
  - \$s0, \$s1, …, \$s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

ARM 7 Registers

- 16 32-bit general purpose registers
- 32 32-bit floating-point registers (not available on every device)

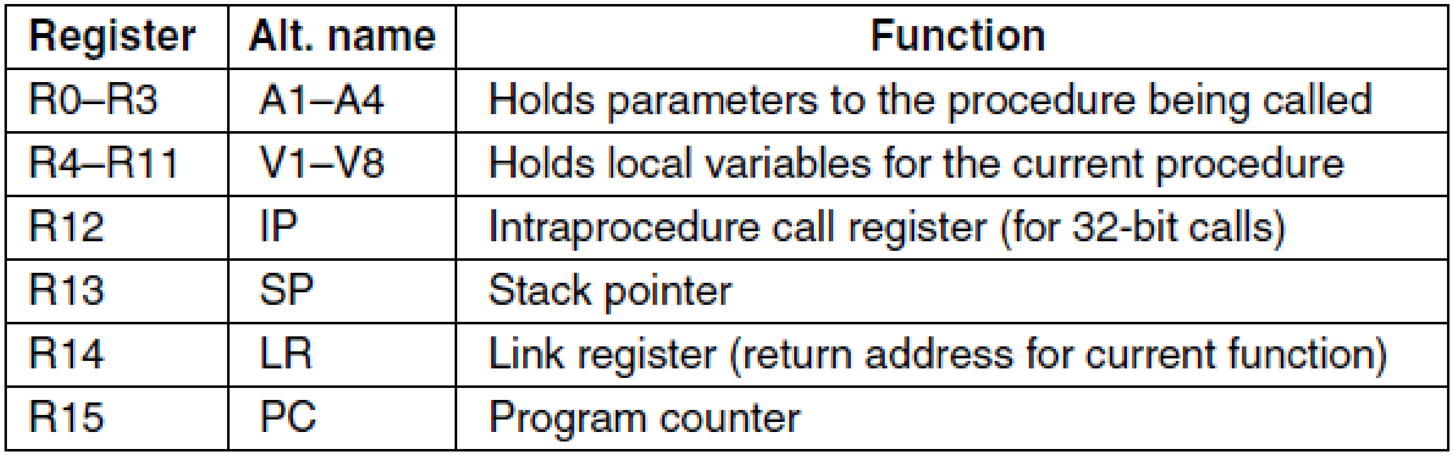


Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

ARM 7 Registers

- The Vx registers hold data needed by procedures (functions)
- They should be stored in memory when calling another procedure
- They should be restored from memory when returning from another procedure

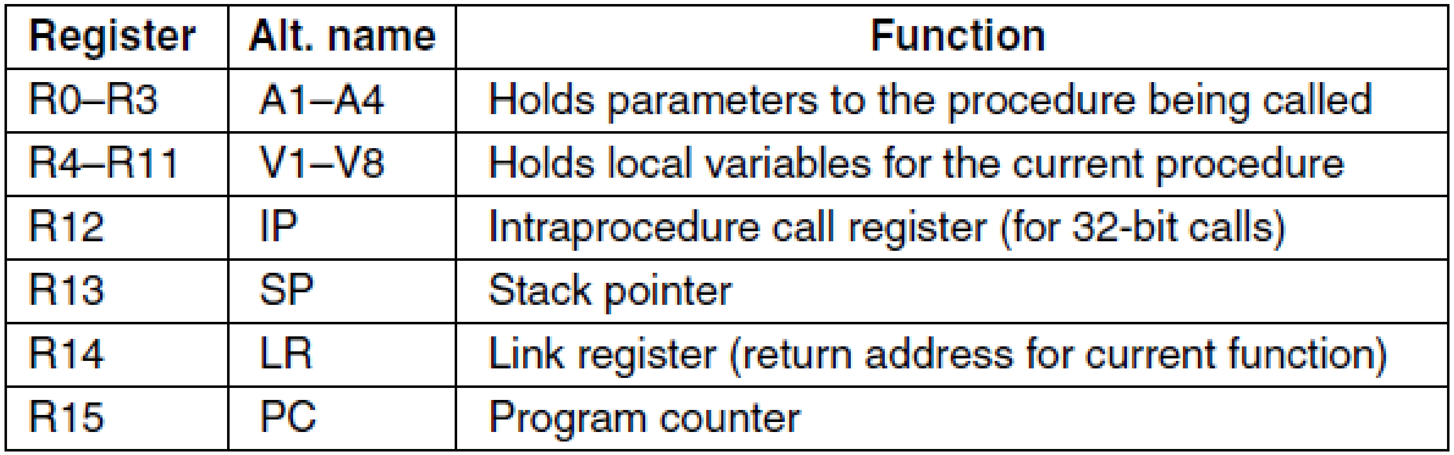


Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

ARM 7 Registers

- The Ax registers are used for passing parameters to procedures
- Four dedicated registers have special roles: IP, SP, LR, PC.
  - We will see more details on these registers are later.
- Who ensures that these registers are used as specified here?



Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM’s general registers.

ARM 7 Registers

- The Ax registers are used for passing parameters to procedures
- Four dedicated registers have special roles: IP, SP, LR, PC.
  - We will see more details on these registers are later
- Who ensures that these registers are used as specified here?
  - **You!!! (The programmer)**

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

ARM: Load/Store Architecture

- ARM is a load/store architecture
- This means that memory can only be accessed by load and store instructions
- All arguments for arithmetic and logical instructions must either:
    - Come from registers
    - Be constants specified within the instruction
        - (more examples of that later)
- This may not seem like a big deal to you, as you have not experienced the alternative
    - However, it makes life much easier
    - This is one reason why we chose ARM 7 for this course

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS/ARM are Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - `f, …, j` in:
    - `$s0, …, $s4 (MIPS)`
    - `r0, …, r4 (ARM)`

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- Compiled ARM code:

```
add r0, r1, r2
add r1, r3, r4 // overwrite r1
sub r0, r0, r1
```

- Note: syntax and semantics (meaning) differences

ARM Instructions in Machine Language

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bit	4 bits	1 bit	6 bits	6 bits	12 bits

- Opcode: Basic operation of the instruction
- Rd: The register destination operand. It gets the result of the operation
- Rn: The first register source operand
- Operand2: The second source operand
- I: Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate
- S: Set Condition Code. This field is related to conditional branch instructions
- Cond: Condition. Related to conditional branch instructions
- F: Instruction Format. This field allows ARM to different instruction formats when needed

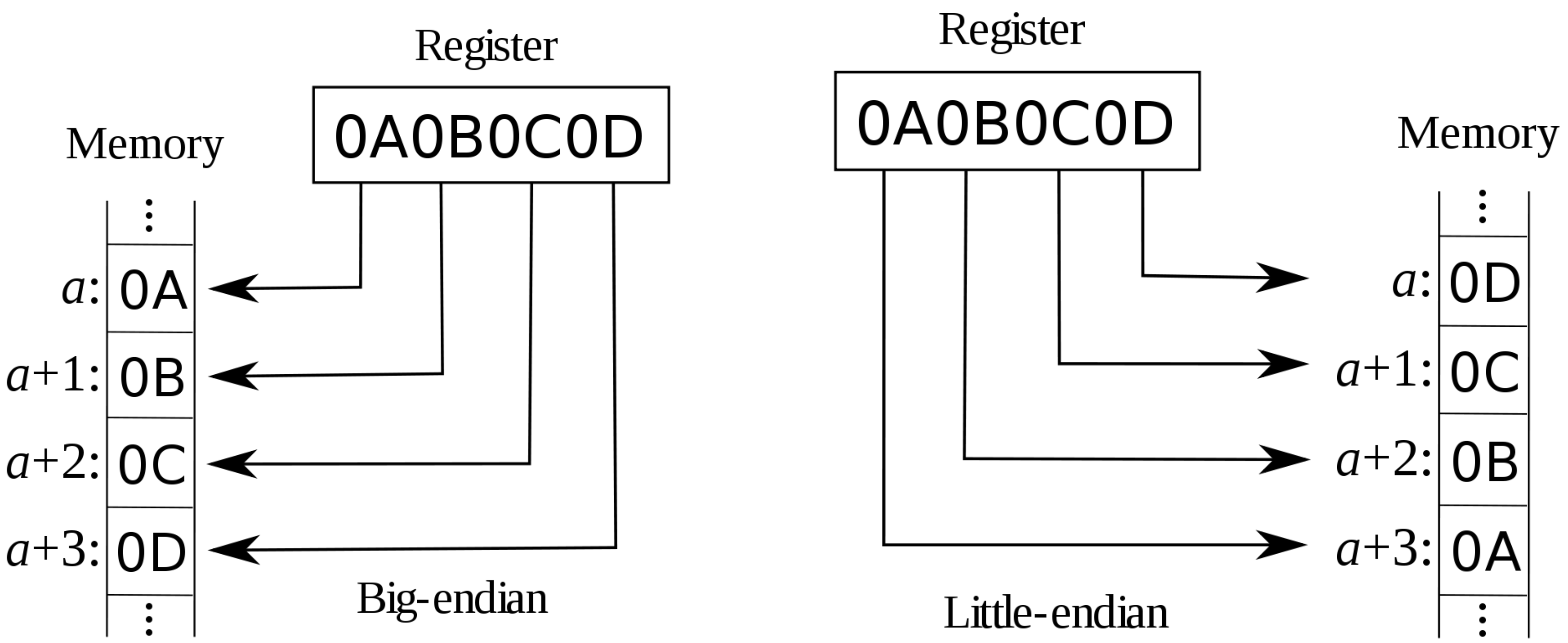
Byte Ordering - Endianness

- How do we store an integer in memory?
- Simple answer: in binary
- Actual answer: yes, in binary, but this does not fully specify how we store the number
- Unfortunately, we have two choices
- Common architectures may follow either choice, and mess ensues, unless we are aware of this issue and we deal with it explicitly
- This is the problem of **endianness**

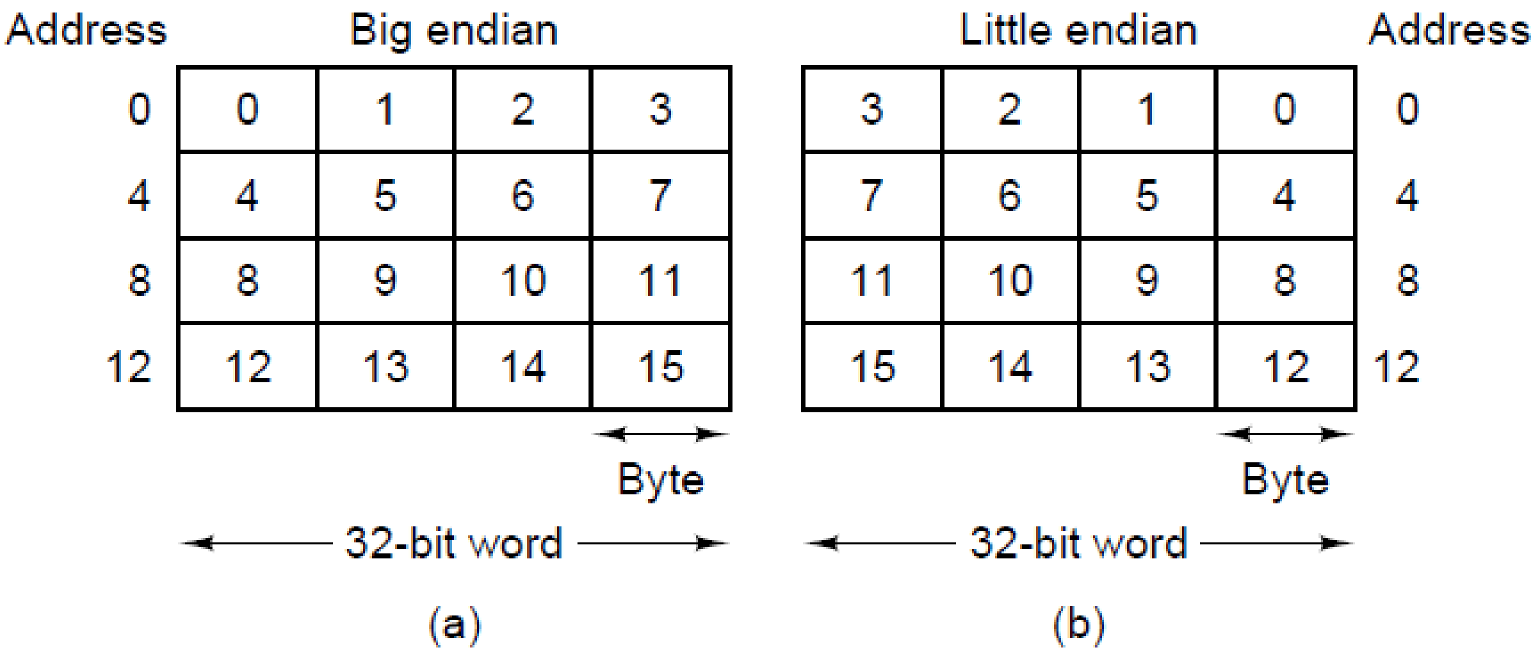
Endianness

- Little-endian: increasing numeric significance with increasing memory addresses
- Big-endian: decreasing numeric significance with increasing memory addresses
- Little-Endian Examples
  - x86, x86-64, 8051, DEC Alpha, Atmel AVR
- Big-Endian Examples
  - Motorola 6800 and 68k series, Xilinx Microblaze, IBM POWER, and System/360
- Bi-Endianness
  - Ability for computer to operate using either
  - SPARC
  - ARM architecture: little-endian before version 3, now bi-endian

Endianness Example



Byte Ordering Visualization



(a) Big endian memory. (b) Little endian memory.

Main difference: ordering of bytes in a word

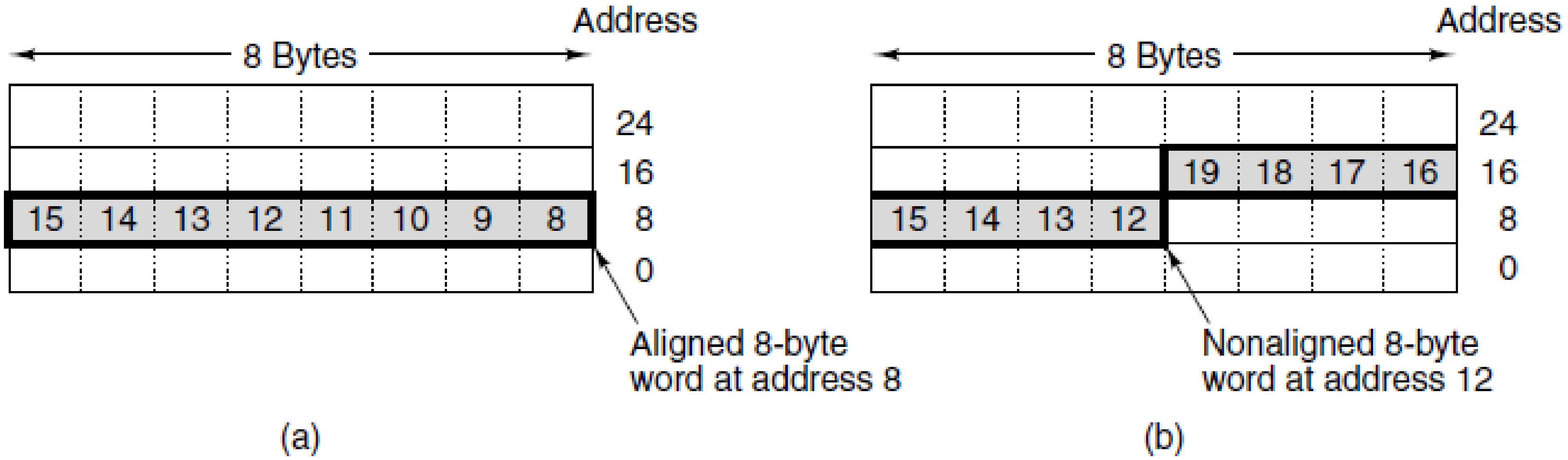
- Left-to-right in big endian.

- Right-to-left in little-endian.

Memory: Words and Alignment

- Bytes are grouped into **words**
- Depending on the machine, a word can be:
  - 32 bits (4 bytes) , or
  - 64 bits (8 bytes), or … (16-bits, 128 bits, etc.)
- Oftentimes it is required that words are aligned
- This means that:
  - 4-byte words can only begin at memory addresses that are multiples of 4: 0, 4, 8, 12, 16…
  - 8-byte words can only begin at memory addresses that are multiples of 8: 0, 8, 16, 24, 32, …

Memory Models



An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

Memory Cells and Addresses

- Memory cell: a piece of memory that contains a specific number of bits
    - How many bits depends on the architecture
    - In modern architectures, it is almost universal that a cell contains 8 bits (1 byte), and that will be also our convention in this course
- Memory address: a number specifying a location of a memory cell containing data
    - Essentially, a number specifying the location of a byte of memory

Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell
- For example, suppose we have a 96-bit memory.
- If each cell is 8 bits, we have ??? addresses?
- If each cell is 12 bits, we have ??? addresses?
- If each cell is 16 bits, we have ??? addresses?

Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell
- For example, suppose we have a 96-bit memory.
- If each cell is 8 bits, we have 12 addresses?
- If each cell is 12 bits, we have 8 addresses?
- If each cell is 16 bits, we have 6 addresses?

- Convention used almost everywhere, and in this course: if a memory has ***n*** cells, the addresses of these cells will be from **0** to ***n-1***.

Address Spaces For Instructions and Data

- Typically memory can be accessed using a single address space
  - For example, if we have 4 GB of memory, each byte has an address from 0 to 2^{32} - 1.
  - Each memory location may store instructions at some point and data at some other point
- An alternative is to have separate address spaces for instructions and data
  - In that case, a memory location is permanently dedicated to either storing instructions or to storing data
  - Instead of a single **load** instruction, we have **load_instructions** and **load_data**

Effects of Separate Address Spaces

- If A is a valid memory address, load_instructions A and load_data A access different memory locations.
  - load_instructions A accesses address A in the instructions space.
  - load_data A accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?

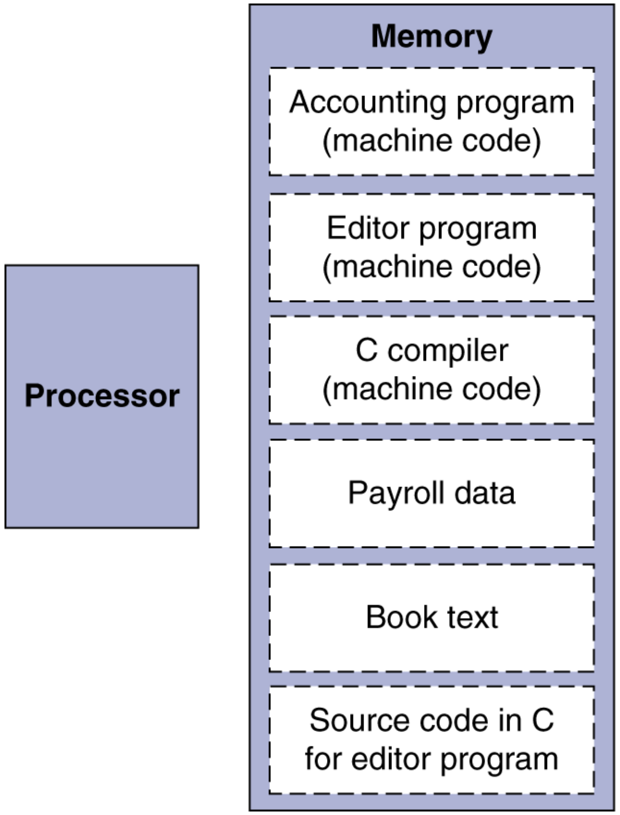
Effects of Separate Address Spaces

- If A is a valid memory address, load_instructions A and load_data A access different memory locations.
  - load_instructions A accesses address A in the instructions space.
  - load_data A accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?
- A common way for malware to attack is to:
  - Run as regular program.
  - Modify memory locations that store instructions, thus modifying other programs (such as the operating system).
- If instruction memory is accessed with different instructions, such behavior can easily be prevented.

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Stored Program Computers


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Announcements and Outline

- Quiz 2 on Blackboard site (due 11:59PM Friday)
  - Review binary arithmetic, Boolean operations, and representing signed and unsigned numbers in binary
- Homework 1 due Thursday
  - Read chapter 1
- Homework 2 assigned Thursday
  - Start reading chapter 2 (ARM version on Blackboard site)

- Review from last time / Chapter 1
  - Performance metrics
- Signed vs. Unsigned Numbers (Two's Complement)
- Instructions: the Language of the Computer

Questions?

?

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in \$s2, base address of A in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

  ```
  addi $s2, $s1, -1
  ```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- ARM (and MIPS) instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible