HARVARD
School of Engineering and Applied Sciences

Introduction to Assembly Programming

CS61, Lecture 2

Prof. Stephen Chong

September 7, 2010

Announcements

- Sections
  - Section times
    - Tue 4:00-5:30pm
    - Tue 7:00-8:30pm
    - Wed 10:00-11:30am
    - Wed 2:30-4:00pm
    - Thu 1:00-2:30pm
  - Go to https://www.section.fas.harvard.edu/ to sign up for a section by **5pm Friday**
  - Sections start next week (Tue 14 Sept onwards)

Announcements

- GDB tutorial
  - 4pm–5pm today
  - Pierce 307
  - Learn how to use gdb. This will be valuable in the upcoming labs...

Announcements

Announcements

- Highscore contest
  - About 40 people tried it out

Announcements

- Highscore contest
  - About 40 people tried it out
  - Top score was 61, achieved by:
    - Andrew Zhou, Siddarth Chandrasekaran, Ashok Cutkosky, Daniel Margo, Danny Zhu, Charles Herrmann, Herman Gudjonson, Brandon Liu, Mengqi Niu, Robert Nishihara, Robert Bowden, Svilen Kanev, Stefan Muller, Andrew Wang, Winston Luo, Wenchi Zhou, Max Wang, Michael Chen, Tony Ho, David Garcia, Olga Zinoveva, Steven Tricanowicz

Announcements

- Highscore contest
  - About 40 people tried it out
  - Top score was 61, achieved by:
    - Andrew Zhou, Siddarth Chandrasekaran, Ashok Cutkosky, Daniel Margo, Danny Zhu, Charles Herrmann, Herman Gudjonson, Brandon Liu, Mengqi Niu, Robert Nishihara, Robert Bowden, Svilen Kanev, Stefan Muller, Andrew Wang, Winston Luo, Wenchi Zhou, Max Wang, Michael Chen, Tony Ho, David Garcia, Olga Zinoveva, Steven Tricanowicz
  - Even better score achieved by:
    - Edward Gan, Ethan Kruse, Joseph Tassarotti

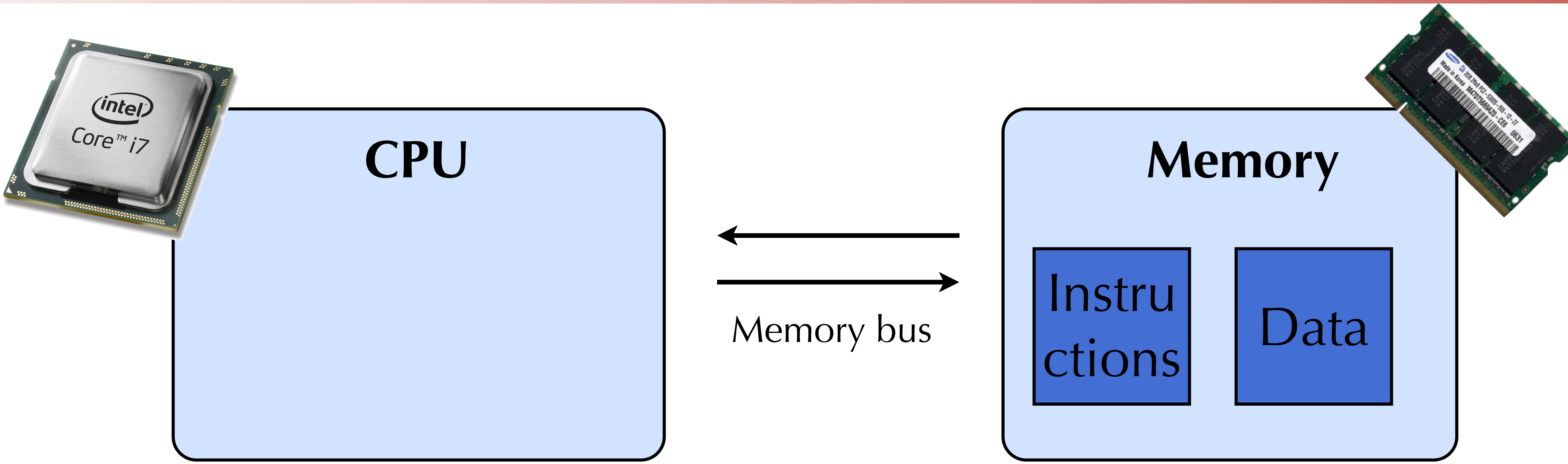
Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Byte ordering
- C, assembly, machine code
  - Basic processor operation
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data

Information

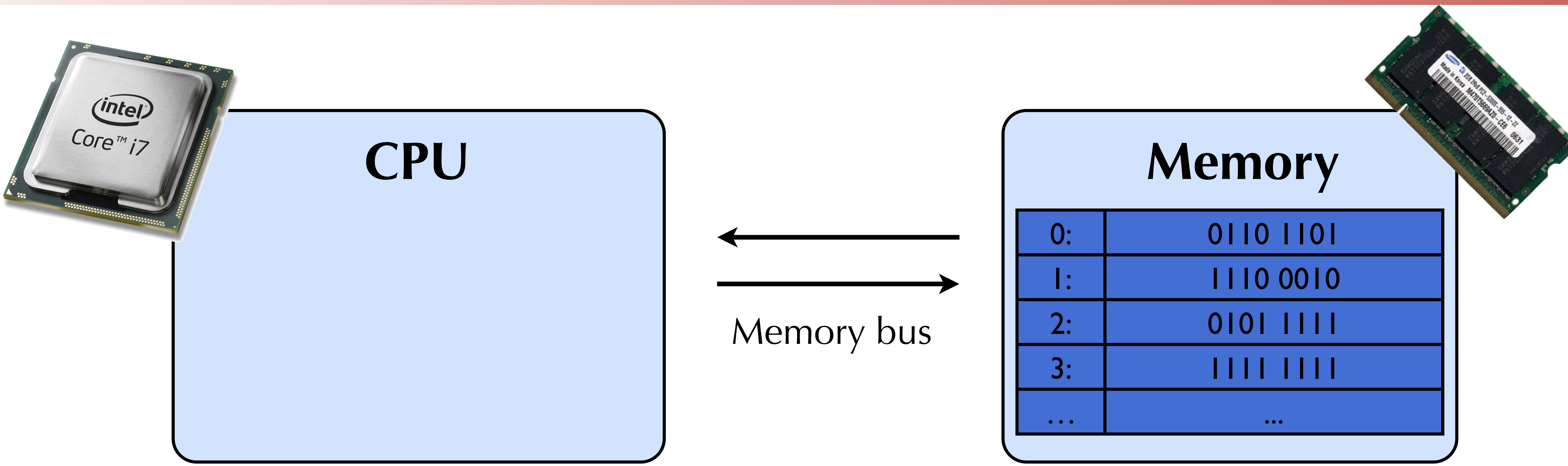
- Computers represent information using **bits**
  - 2-valued signals; binary digits
  - All kinds of information
    - Numbers, memory addresses, instructions, strings, …
- Information is bits plus **context**
  - context = way of interpreting data
  - What do the bits 1100 0011 represent?
    - Could be (unsigned) integer 195
    - Could be (signed) integer -61
    - Could be instruction `ret`
    - Depends on context!

Computers



- Computers store bits in memory
- Information stored in memory is both instructions and data
  - But remember, instructions and data are just bits that get interpreted differently!

Computers



- Rather than accessing individual bits, most computers use blocks of 8 bits, called bytes
- View memory as a very large array of bytes
  - Memory addresses another kind of data

Hexadecimal notation

Hexadecimal notation

- To make it easier to read bits, we use **hexadecimal notation**
  - Decimal notation is base 10
    - Uses digits 0,1,2,3,4,5,6,7,8,9
    - *xyz* represents number $z \cdot 10^0 + y \cdot 10^1 + x \cdot 10^2$

Hexadecimal notation

- To make it easier to read bits, we use **hexadecimal notation**
  - Decimal notation is base 10
    - Uses digits 0,1,2,3,4,5,6,7,8,9
    - *xyz* represents number $z \cdot 10^0 + y \cdot 10^1 + x \cdot 10^2$
  - Binary notation is base 2
    - Uses digits 0,1
    - *xyz* represents number $z \cdot 2^0 + y \cdot 2^1 + x \cdot 2^2$

Hexadecimal notation

- To make it easier to read bits, we use **hexadecimal notation**
  - Decimal notation is base 10
    - Uses digits 0,1,2,3,4,5,6,7,8,9
    - xyz represents number $z \cdot 10^0 + y \cdot 10^1 + x \cdot 10^2$
  - Binary notation is base 2
    - Uses digits 0,1
    - xyz represents number $z \cdot 2^0 + y \cdot 2^1 + x \cdot 2^2$
  - Hexadecimal notation is base 16
    - Use digits 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
    - xyz represents number $z \cdot 16^0 + y \cdot 16^1 + x \cdot 16^2$

Hexadecimal notation

Binary value	Dec. value	Hex. digit
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Hexadecimal notation

- One hexadecimal digit represents 4 bits

Binary value	**Dec. value**	**Hex. digit**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Hexadecimal notation

- One hexadecimal digit represents 4 bits
- One byte is two hexadecimal digits
  - E.g., 0x8c = 10001100

Binary value	Dec. value	Hex. digit
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Hexadecimal notation

- One hexadecimal digit represents 4 bits
- One byte is two hexadecimal digits
  - E.g., 0x8c = 10001100
- We prefix hex. numbers with "0x"
  - E.g., 0x5a = 01011010 = 90 = $10+5\cdot16$
  - E.g., 0x42 = 01000010 = 66 = $2+4\cdot16$
- You will get comfortable and familiar with hexadecimal notation.

Binary value	**Dec. value**	**Hex. digit**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Word size

- Every computer has a **word size**
  - Indicates number of bits that can be used to store integers, memory addresses
- We are in transition between 32-bit machines and 64-bit machines
  - 32-bit machines can name 2^{32} different memory locations
    - 1 byte per memory location = 4 gigabytes (= 4×2^{30} bytes)
  - 64-bit machines can name 2^{64} different memory locations
    - 1 byte per memory location = 16 exabytes (= 16×2^{60} bytes)

Representing integers

Representing integers

- Given 32 bits to store an integer, we can represent 2^{32} different values

Representing integers

- Given 32 bits to store an integer, we can represent 2^{32} different values
- If we just care about non-negative (aka **unsigned**) integers, we can easily store the values
  $0, 1, 2, \ldots, 2^{32}-1$

Representing integers

- Given 32 bits to store an integer, we can represent 2^{32} different values
- If we just care about non-negative (aka **unsigned**) integers, we can easily store the values
  $0, 1, 2, \ldots, 2^{32}\text{-}1$
  - Representation straightforward
    - `0x0000000b` = 11
    - `0xdeadbeef` = 3,735,928,559
    - `0xffffffff` = 2^{32}-1

Integer overflow

Integer overflow

- With 32 bits, we can represent values 0, 1, 2, …, 2^{32}-1
- **Overflow** occurs when we have a result that doesn't fit in the 32 bits
  - E.g., `0xffffffff + 0x1`

Integer overflow

- With 32 bits, we can represent values 0, 1, 2, …, 2^{32}-1
- **Overflow** occurs when we have a result that doesn't fit in the 32 bits
  - E.g., `0xffffffff + 0x1`
- For unsigned integers with *w* bits, addition is

$$x +_w^u y = \begin{cases} x + y & \text{if } x + y < 2^w \\ x + y - 2^w & \text{if } 2^w \leq x + y < 2^{w+1} \end{cases}$$

  - E.g., `0xffffffff + 0x1 = 0x0`

Representing negative integers

Representing negative integers

- If we care about positive and negative integers, have some options

Representing negative integers

- If we care about positive and negative integers, have some options
- **Sign and magnitude**
  - Use one bit to represent sign
  - Remaining bits represent magnitude
  - With 32 bits, have 31 bits for magnitude
    - Can represent integers $-2^{31}+1, \ldots, 0, \ldots, 2^{31}-1$

Representing negative integers

- If we care about positive and negative integers, have some options
- **Sign and magnitude**
  - Use one bit to represent sign
  - Remaining bits represent magnitude
  - With 32 bits, have 31 bits for magnitude
    - Can represent integers $-2^{31}+1$, …, 0, …, $2^{31}-1$
- Key properties of sign and magnitude
  - Straight-forward and intuitive
  - Two different representations of zero!
    - magnitude of zero, +ve sign; magnitude of zero, -ve sign
  - Arithmetic operations need different implementation than for unsigned
    - E.g., addition, right and left shifting, ...

Two's complement

Two's complement

- **Two's-complement** representation for negative numbers is most common
  - With 32 bits, can represent integers -2^{31}, …, 0, …, $2^{31}-1$

Two's complement

- **Two's-complement** representation for negative numbers is most common
  - With 32 bits, can represent integers -2^{31}, …, 0, …, $2^{31}-1$
- Key properties of two's-complement
  - Positive numbers represented in intuitive way
    - e.g., `0x0000000b` = 11

Two's complement

- **Two's-complement** representation for negative numbers is most common
  - With 32 bits, can represent integers -2^{31}, …, 0, …, $2^{31}-1$
- Key properties of two's-complement
  - Positive numbers represented in intuitive way
    - e.g., `0x0000000b` = 11
  - First bit of representation is 1 iff negative
    - e.g., `0xdeadbeef` = -559,038,737

Two's complement

- **Two's-complement** representation for negative numbers is most common
  - With 32 bits, can represent integers -2^{31}, …, 0, …, $2^{31}-1$
- Key properties of two's-complement
  - Positive numbers represented in intuitive way
    - e.g., `0x0000000b` = 11
  - First bit of representation is 1 iff negative
    - e.g., `0xdeadbeef` = -559,038,737
  - -1 always represented by bit string of all 1s
    - e.g., `0xffffffff` = -1
    - Means that (-1) + 1 = 0

Two's complement

- **Two's-complement** representation for negative numbers is most common
  - With 32 bits, can represent integers -2^{31}, …, 0, …, $2^{31}-1$
- Key properties of two's-complement
  - Positive numbers represented in intuitive way
    - e.g., `0x0000000b` = 11
  - First bit of representation is 1 iff negative
    - e.g., `0xdeadbeef` = -559,038,737
  - -1 always represented by bit string of all 1s
    - e.g., `0xffffffff` = -1
    - Means that (-1) + 1 = 0
  - Addition operation works as for unsigned integers!
  - Left shifting and right shifting also work (provided MSB preserved)

Integer overflow

- Overflow can also occur with negative integers
- With 32 bits, maximum integer expressible is 2^{31}-1 = `0x7fffffff`
- `0x7fffffff` + `0x1` = `0x80000000` = -2^{31}
- `0x80000000` + `0x80000000` = `0x0`
- In twos complement, addition is

$$x +_w^t y = \begin{cases} x + y - 2^w & \text{if } 2^{w-1} \leq x + y \quad \text{Positive overflow} \\ x + y & \text{if } -2^{w-1} \leq x + y < 2^{w-1} \quad \text{Normal} \\ x + y + 2^w & \text{if } x + y < -2^{w-1} \quad \text{Negative overflow} \end{cases}$$

Data sizes

- C language has multiple data formats for integer and floating-point data

C declaration	32-bit	64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char *`	4	8
`float`	4	4
`double`	8	8

Byte ordering

- Memory is big array of bytes
  - Address of location is integer index into array
- When we have data that is more than one byte long, which order do we store the bytes?
  - **Big-endian**: most significant bytes first in memory
  - **Little-endian**: least significant bytes first in memory
    - Most Intel-compatible machines are little-endian

Memory

0:	0x6d
1:	0xe2
2:	0x5f
3:	0xff
…	…

Byte ordering example

Byte ordering example

- Consider 32 bit (4 byte) integer `0xff5fe26d`
- Suppose stored at memory address `0x100`
  - i.e., occupies locations `0x100`, `0x101`, `0x102`, `0x103`

Byte ordering example

- Consider 32 bit (4 byte) integer `0xff5fe26d`
- Suppose stored at memory address `0x100`
  - i.e., occupies locations `0x100`, `0x101`, `0x102`, `0x103`
- Big endian: most significant bits first

	0x100	0x101	0x102	0x103	
…	0xff	0x5f	0xe2	0x6d	…

Byte ordering example

- Consider 32 bit (4 byte) integer `0xff5fe26d`
- Suppose stored at memory address `0x100`
  - i.e., occupies locations `0x100`, `0x101`, `0x102`, `0x103`
- Big endian: most significant bits first

	0x100	0x101	0x102	0x103	
…	0xff	0x5f	0xe2	0x6d	…

- Little endian: least significant bits first

	0x100	0x101	0x102	0x103	
…	0x6d	0xe2	0x5f	0xff	…

Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Byte ordering
- C, assembly, machine code
  - Basic processor operation
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data

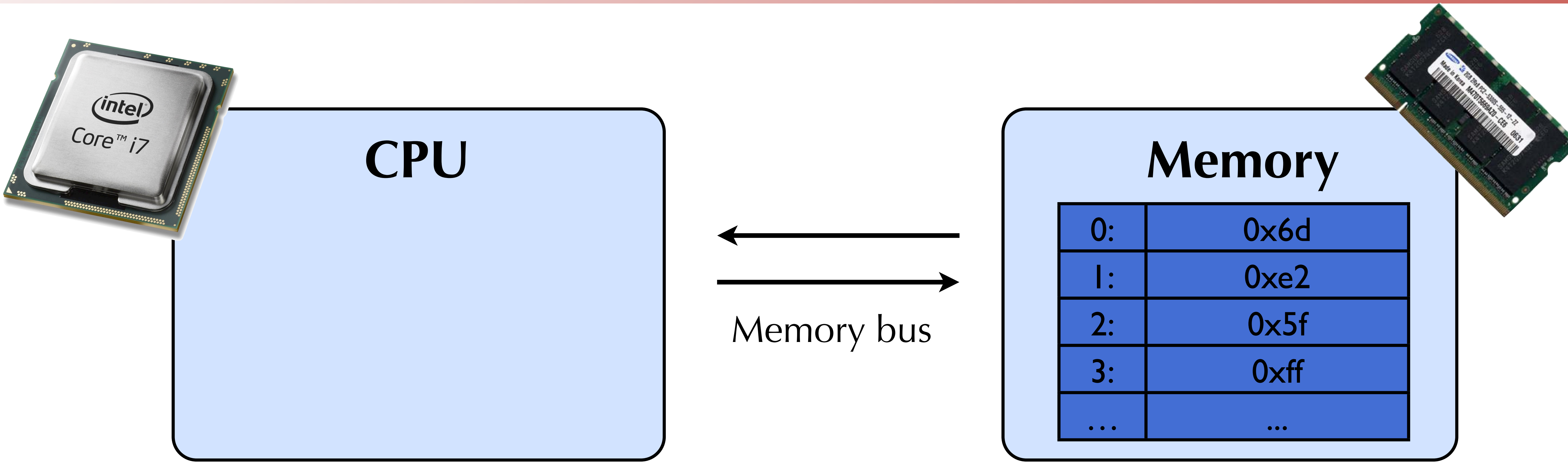
Machine code and assembly

- An **instruction** is a single operation that the CPU can perform
  - add, subtract, copy, call, …
- **Machine code** is bit-level representation of instructions
  - E.g., in x86: `0x83 0xec 0x10` represents "subtract 0x10 from value in the `%esp` register"
  - Different instructions may take different number of bits to represent
  - Hard for humans to read
- **Assembly** is human-readable form of machine code
  - E.g,. `sub 0x10, %esp`

Instruction Set Architecture (ISA)

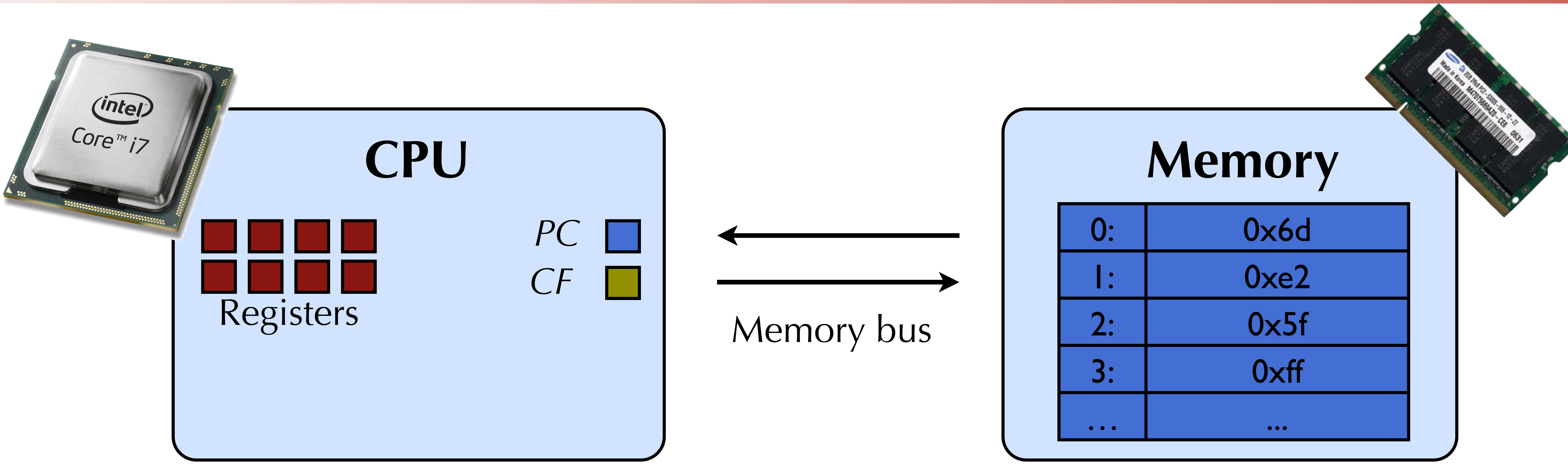
- Definition of machine instructions and format used internally by CPU
  - What instructions the processor can perform, how they are represented, what data types they operate on, etc.
- Specific to the kind of chip and manufacturer
  - Many ISAs, e.g., Alpha, ARM, MIPS, PowerPC, SPARC
- In this course we study the **Intel IA-32 ISA** (aka **x86**)
  - Originated by Intel
  - For 32 bit architectures
  - Evolved (and backward compatible) from earlier ISAs
- Will see some of **x86-64**
  - 64 bit extension of x86

Processor architecture



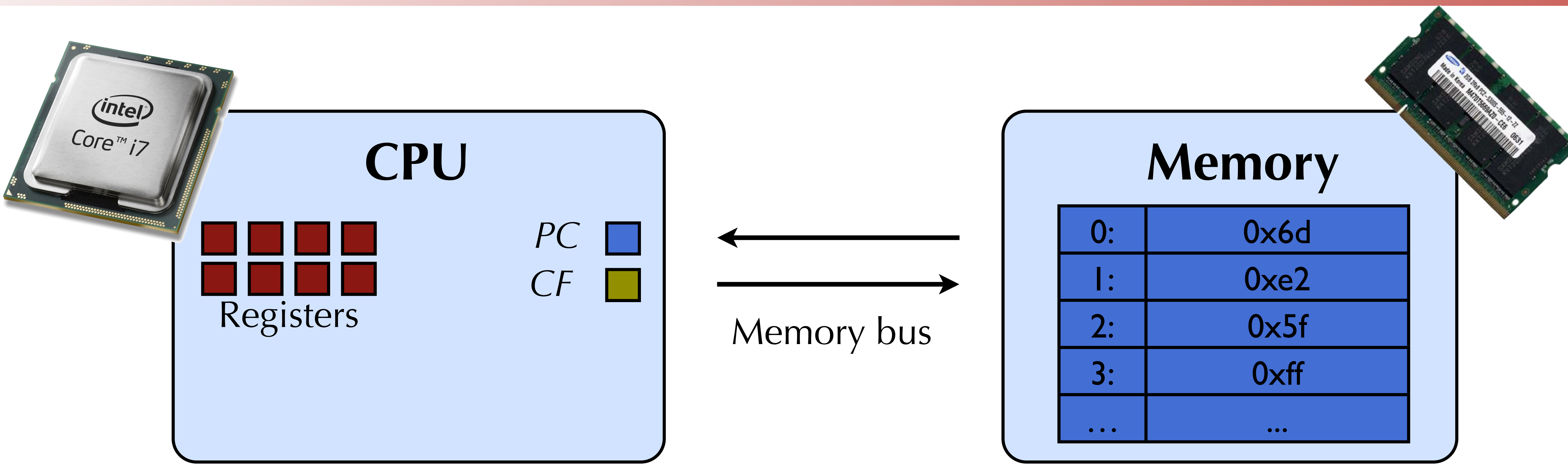
- CPU executes a series of instructions
  - Each instruction is a simple operation: add, load, store, jump, etc.
  - Instructions stored in memory
  - Program Counter (PC) holds memory address of next instruction
- CPU can read or write memory over the memory bus
  - Can generally read or write a single byte or word at a time

Processor operation



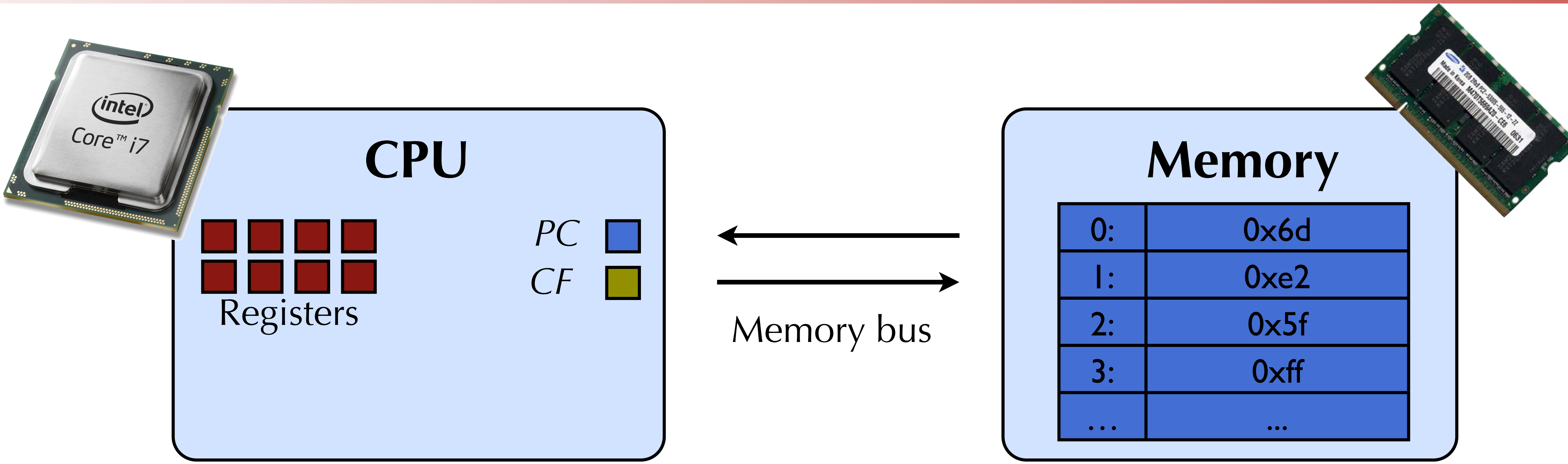
- Basic processor operation:
  - 1) Fetch instruction from memory address pointed to by program counter *PC*
  - 2) Execute instruction
  - 3) Set *PC* to address of next instruction
- Where is the next instruction?
  - Not just "*PC* + 1" – each instruction can be a different size!
  - "Jump" instruction also sets *PC* to new value

Registers



- Registers are used to store "temporary" data on the CPU itself
  - Extremely fast to access a register: 1 clock cycle (0.4 ns on a 2.4 GHz processor)
  - But reading or writing memory can ~40 ns (depends on a lot of factors)
    - Nearly 100x "slowdown" to go to memory!
- The Intel x86 has eight 32-bit registers.
  - Named `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, `%ebp`
  - There are conventions on how certain registers are used

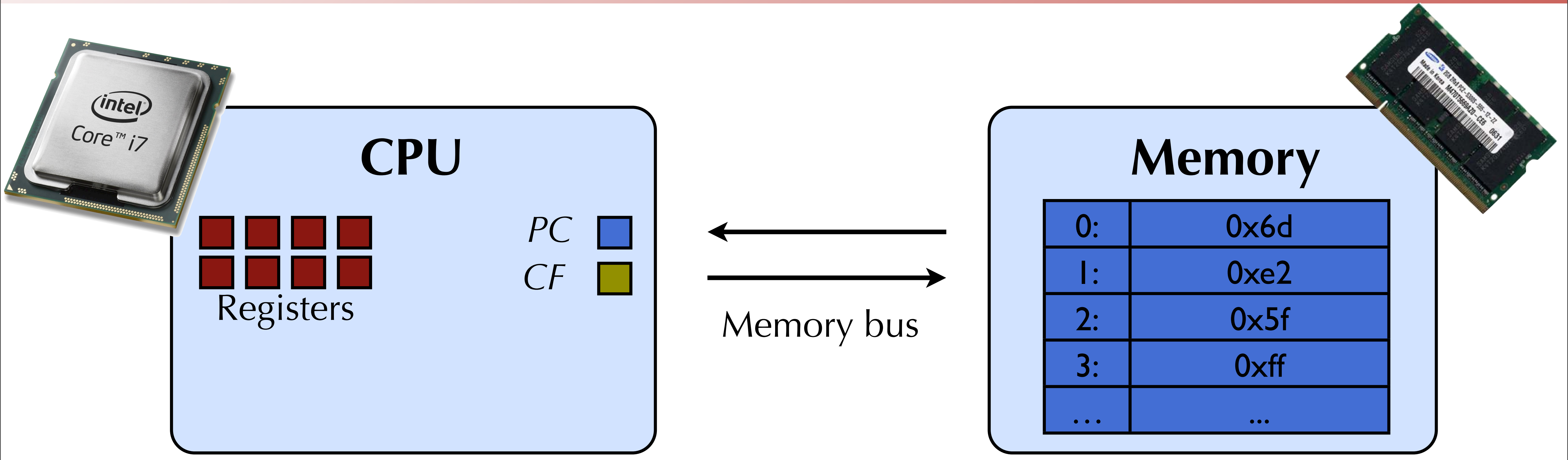
Condition flags



- Condition Flags (CF) hold information on state of last instruction
  - Each flag is one bit.
  - Often used by other instructions to decide what to do.
  - e.g., **Overflow flag** is set to 1 if you add two registers, and the value overflows a word.
  - **Zero flag** set to 1 if result of an operation is zero

```
subl $0x42, %eax   # Subtract 0x42 from value in %eax
jz   $0x80495bc    # If zero flag set, jump to instruction
                   #   at 0x80495bc
```

Accessing memory



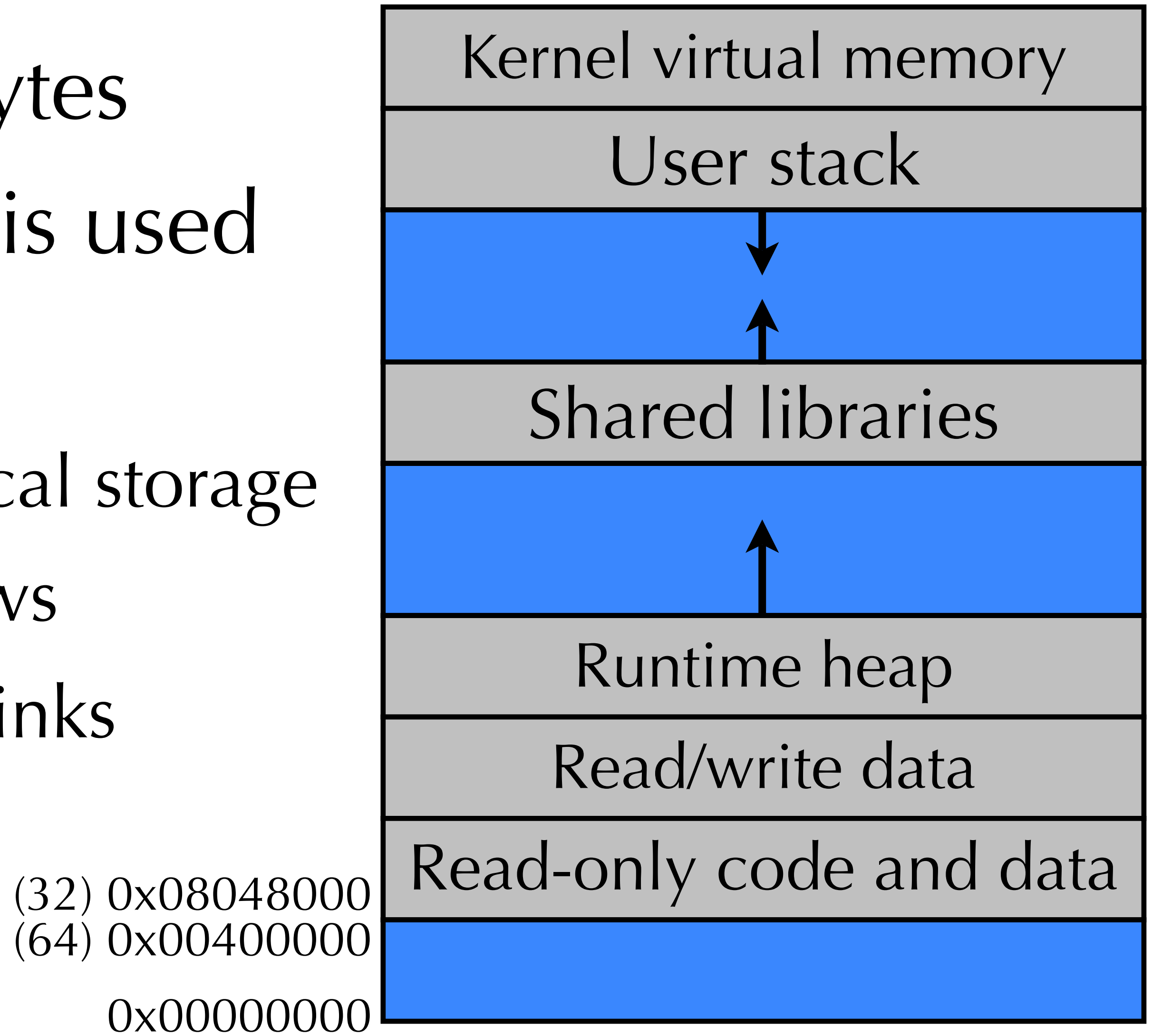
- "Move" instructions are used to read/write registers and memory locations

```
movl $0x00001000, %eax   # Set %eax register to value 0x00001000
movl 0x0000ea5f, %eax    # Set %eax register to contents of
                         #   memory address 0x0000ea5f

movl (%eax), %ebx        # Set %ebx register to contents of
                         #   memory address stored in %eax
```

More on memory

- View memory as large array of bytes
- Some conventions on how array is used
- Stack
  - Used to implement function calls, local storage
  - Every time function called, stack grows
  - Every time function returns, stack shrinks
- Heap
  - Dynamically allocated storage for program
  - Expands and contracts as result of calls to malloc and free



Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Byte ordering
- C, assembly, machine code
  - Basic processor operation
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data

Turning C into machine code

C program
(myprog.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

Turning C into machine code

C program
(myprog.c)

```c
int dosum(int i, int j) {
    return i+j;
}
```

C compiler (gcc)

Assembler program
(myprog.s)

```
dosum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
         ret
```

Turning C into machine code

C program
(myprog.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

C compiler (gcc)

Assembler program
(myprog.s)

```
dosum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
         ret
```

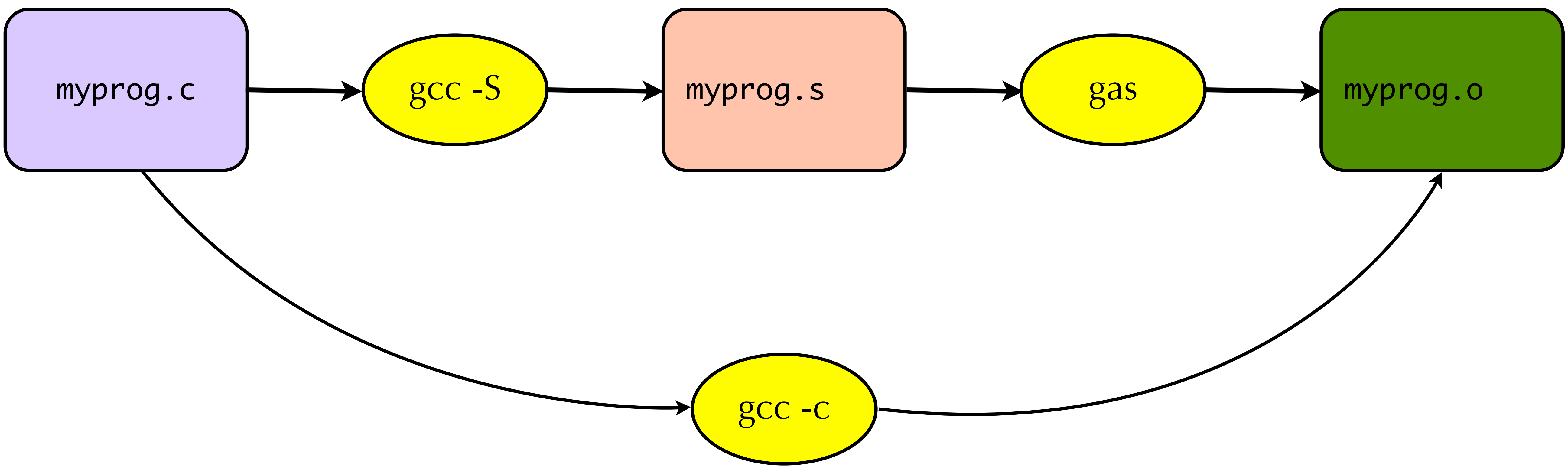
Assembler (gas)

Machine code
(myprog.o)

```
80483b0: 55 89 e5 8b 45 0c 03 45 08 5d c3
```

Skipping assembly language

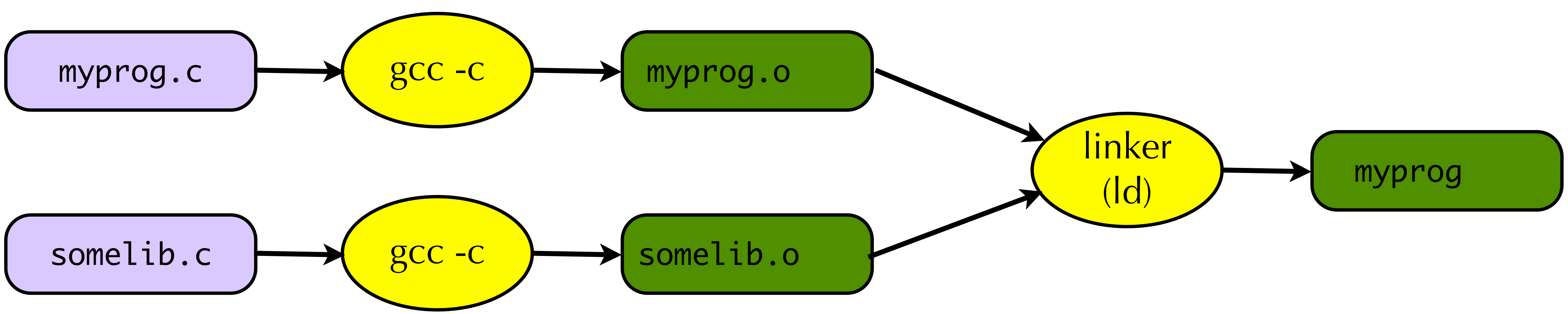
- Most C compilers generate machine code (object files) directly.
  - That is, without actually generating the human-readable assembly file.
  - Assembly language is mostly useful to people, not machines.



- Can generate assembly from C using “gcc -S”
  - And then compile to an object file by hand using “gas”

Object files and executables

- C source file (myprog.c) is compiled into an **object file** (myprog.o)
  - Object file contains the machine code for that C file.
  - It may contain references to external variables and routines
  - E.g., if myprog.c calls printf(), then myprog.o will contain a reference to printf().
- Multiple object files are **linked** to produce an executable file.
  - Typically, standard libraries (e.g., "libc") are included in the linking process.
  - Libraries are just collections of pre-compiled object files, nothing more!



Characteristics of assembly language

- Assembly language is very, very simple.
- Simple, minimal data types
  - Integer data of 1, 2, 4, or 8 bytes
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures!
- Primitive operations
  - Perform arithmetic operation on registers or memory (add, subtract, etc.)
  - Read data from memory into a register
  - Store data from register into memory
  - Transfer control of program (jump to new address)
  - Test a control flag, conditional jump (e.g., jump only if zero flag set)
- More complex operations must be built up as (possibly long) sequences of instructions.

Why you need to understand assembly language

- These days, very few people write assembly code
  - Very *very* few people write significant amounts of assembly code!
  - You won't need to write assembly in this course, and probably won't in future
- But, you will need to be able to read it to understand what a program is really doing, and how the processor works.
- Examples:
  - Understanding strange memory bugs (stack smashing, core dumps, etc.)
  - Understanding what affects the performance of a given piece of code
  - Understanding what the heck the compiler is doing to your precious C program
- Other uses...
  - Writing device drivers: Sometimes need to drop down to assembler
  - Writing an OS or embedded system
  - Writing a compiler

Disassembling

- Assembly is a human readable form of machine code
  - Assemblers (e.g., gas) compile assembly to machine code
- Disassemblers convert machine code to assembly
  - Interprets bits as instructions
  - Useful tools for examining machine code

Disassemblers

- objdump
  - `objdump -d myprog.o`
  - Can be used on object files (.o) or complete executables
- gdb
  - GNU debugger
  - Can disassemble, run, set breakpoints, examine memory and registers
  - gdb tutorial after class today! If you can't make it, go to cs61 website for some resources for learning gdb
- Play around with both! gdb will be especially helpful in labs

What can be disassembled?

- Anything that can be interpreted as executable code
- Disassembler simply examines bits, interprets them as machine code, and reconstructs assembly

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:    55                  push   %ebp
30001001:    8b ec               mov    %esp,%ebp
30001003:    6a ff               push   $0xffffffff
30001005:    68 90 10 00 30      push   $0x30001090
3000100a:    68 91 dc 4c 30      push   $0x304cdc91
```

Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Byte ordering
- C, assembly, machine code
  - Basic processor operation
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data

Addressing modes

Addressing modes

- Most instructions have one or more **operands**
  - Specify input and output for operations
  - Inputs can be registers, memory locations, or immediate (constant) values
  - Outputs can be saved to registers or memory locations
- Collectively, these ways of accessing operands are called **addressing modes**

Addressing modes

- Most instructions have one or more **operands**
  - Specify input and output for operations
  - Inputs can be registers, memory locations, or immediate (constant) values
  - Outputs can be saved to registers or memory locations
- Collectively, these ways of accessing operands are called **addressing modes**
- Different instructions support different addressing modes
  - Need to check the manual to find out which modes are allowed
  - Example: "`movl`" instruction (copy 32-bit value) supports...
    - Immediate to register `movl $0x1000, %eax`
    - Register to register `movl %eax, %ebx`
    - Memory to register (a.k.a. "load") `movl (%eax), %ebx`
    - Register to memory (a.k.a. "store") `movl %eax, (%ebx)`
    - Cannot move from memory to memory!

Immediate and register operands

Immediate and register operands

- Immediate operands are for constant values
  - Written with a `$` followed by integer in standard C notation
  - E.g., `$-577`, `$0x1F`
  - Operand value is simply the immediate value

Immediate and register operands

- Immediate operands are for constant values
  - Written with a `$` followed by integer in standard C notation
  - E.g., `$-577`, `$0x1F`
  - Operand value is simply the immediate value
- Register operands denote content of register
  - Written as the name of the register, which starts with a `%` sign
  - E.g., `%eax`, `%ebx`
  - Operand value is $R[E_a]$ where E_a denotes a register, $R[E_a]$ denotes value stored in register

Register names

- Registers `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, `%ebp` are all 32-bit
- Sometimes we handle data smaller than 32 bits
  - Have names for addressing just some bits of a register
    - Historical, due to development of IA32 from 8 and 16 bit architectures

Register names

General purpose registers: %eax, %ecx, %edx, %ebx, %esi, %edi

32-bit	16-bit	High byte	Low byte	Origin (mostly obsolete)
%eax	%ax	%ah	%al	*accumulate*
%ecx	%cx	%ch	%cl	*counter*
%edx	%dx	%dh	%dl	*data*
%ebx	%bx	%bh	%bl	*base*
%esi	%si			*source index*
%edi	%di			*destination index*
%esp	%sp			***stack pointer***
%ebp	%bp			***base pointer***

16-bit virtual registers
(backwards comaptibility)

Memory operands

Memory operands

- Most general form is $Imm(E_b, E_i, s)$
  - Imm is immediate offset, E_b is base register, E_i is index register, s is scale (must be 1, 2, 4 or 8)
  - Effective address is $Imm + R[E_b] + R[E_i] \times s$
  - Operand value is $M[Imm + R[E_b] + R[E_i] \times s]$

Memory operands

- Most general form is $Imm(E_b, E_i, s)$
  - Imm is immediate offset, E_b is base register, E_i is index register, s is scale (must be 1, 2, 4 or 8)
  - Effective address is $Imm + R[E_b] + R[E_i] \times s$
  - Operand value is $M[Imm + R[E_b] + R[E_i] \times s]$
- Other forms special cases of this general form
  - Imm is an immediate, or **absolute**, address
    - e.g., `0x1a38`
  - (E_b) is an indirect address
    - e.g., `(%eax)` is contents of register `%eax`
  - $Imm(E_b)$ is a base address plus a displacement
    - e.g., `0x8(%ebp)` is contents of register `%ebp` plus 8
  - (E_b, E_i) and $Imm(E_b, E_i)$ are indexed addresses

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)		
(%edx, %ecx)		
(%edx, %ecx, 4)		
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)		
(%edx, %ecx, 4)		
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)		
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	**Computation**	**Address**
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)	0xf000 + 4*0x100	0xf400
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)	0xf000 + 4*0x100	0xf400
0x80(, %edx, 2)	2*0xf000 + 0x80	0x1e080

Moving data

- Copy data from one location to another
  - Heavily used!
- `movx` *source, dest*
  - *x* is one of `b`, `w`, `l`
  - `movb` *source, dest*
    - Move 1-byte "byte"
  - `movw` *source, dest*
    - Move 2-byte "word" (for historical reasons)
  - `movl` *source, dest*
    - Move 4-byte "long word" (for historical reasons)

ATT vs Intel syntax

ATT vs Intel syntax

- Two common ways of formatting IA32 assembly
  - ATT
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools

ATT vs Intel syntax

- Two common ways of formatting IA32 assembly
  - ATT
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools
- Differences:
  - Intel omits size designation: `mov` instead of `movl`

ATT vs Intel syntax

- Two common ways of formatting IA32 assembly
  - ATT
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools
- Differences:
  - Intel omits size designation: `mov` instead of `movl`
  - Intel omits `%` from register names: `ebp` instead of `%ebp`

ATT vs Intel syntax

- Two common ways of formatting IA32 assembly
  - ATT
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools
- Differences:
  - Intel omits size designation: `mov` instead of `movl`
  - Intel omits `%` from register names: `ebp` instead of `%ebp`
  - Intel describes memory locations differently: `[ebp+8]` instead of `8(%ebp)`

ATT vs Intel syntax

- Two common ways of formatting IA32 assembly
  - ATT
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools
- Differences:
  - Intel omits size designation: `mov` instead of `movl`
  - Intel omits % from register names: `ebp` instead of `%ebp`
  - Intel describes memory locations differently: `[ebp+8]` instead of `8(%ebp)`
  - **Intel lists operands in reverse order:** `mov` *dest*`,` *src* instead of `movl` *src*`,` *dest*

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`			
`movl $-147,(%eax)`			
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`			
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

movl examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`	Reg	Mem	`*p = temp;`
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`	Reg	Mem	`*p = temp;`
`movl (%eax),%edx`	Mem	Reg	`temp = *p;`

Note: Cannot move directly from memory to memory with single instruction!

Note: C pointers are just memory addresses