CPSC 232
Intro to Assembly Language Programming

Dr. Jochen

Chapter 1: Introduction

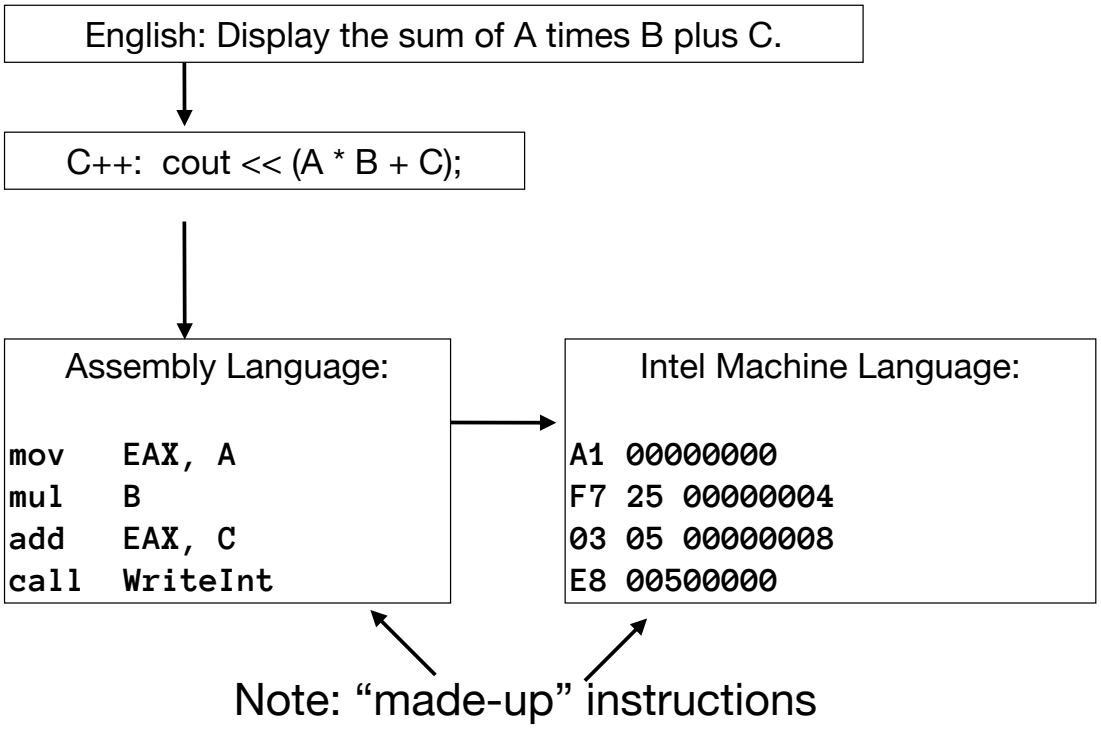
Objectives

- Intro to Assembly Language
- Number Systems
- Computer Organization Refresher
- Basic Assembly
- Writing Assembly

Questions to Ask

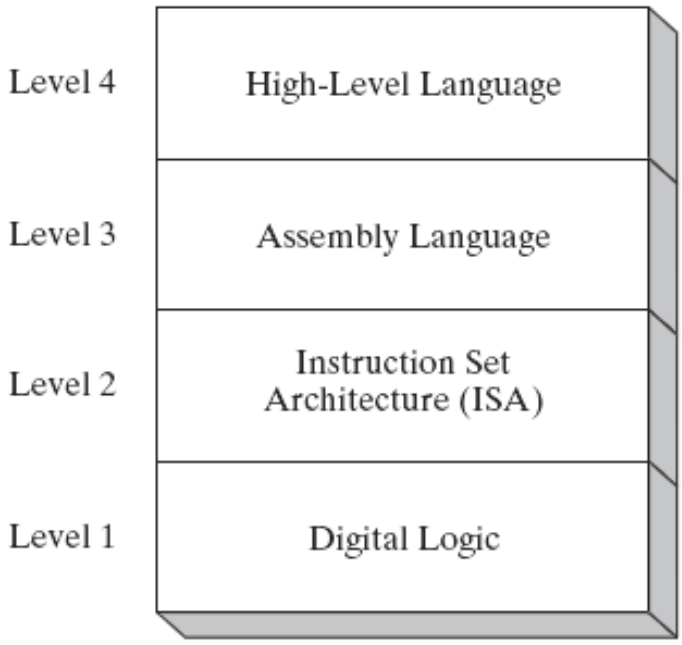
- Why am I learning assembly language?
- What background should I have?
- What is an assembler?
- How does assembly language relate to machine language?
- How do C++ and Java (High Level Languages) relate to assembly?
- Is assembly portable?
- (Back to first question) Why learn assembly language?

What's the Process Look Like?



Specific Machine Levels

- We can think of the compile, assemble, execute process as several “logical” machines
- Each level represents a specific level of abstraction or implementation of the computation/program
- Let’s examine these levels. . .



Level 4: High-Level Language

- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- This is the level within which we humans like to think about the problem
  - Easier for us to think about problems at higher level
  - Focus energy on developing the algorithm
  - Focus less on implementation details
- Programs compile into lower-level assembly language (Level 3)
  - Somewhat less simple to understand & work within
  - Need to understand more specifics of the operating system & the hardware platform (Instruction Set Architecture)

Level 3: Assembly Language

- Instruction mnemonics that have a one-to-one correspondence to machine language
- Believe it or not, this is still somewhat human-readable form of code!
  - Yes, more challenging to understand & work within than high-level languages
  - More understandable than lower-level machine code (layer below)
- Programs are translated into Instruction Set Architecture Level - machine language (Level 2)
- We study levels 2 & 3 in this class (mostly level 3)

Level 2: Instruction Set Architecture (ISA)

- Also known as conventional machine language
- This is literally the ones & zeros (numeric representation of your assembly code)
- For the most part, there exists a one-to-one mapping of a machine code instruction and an assembly instruction
  - Caveat Emptor: some assembler instructions are like macros (they expand into multiple lines of machine code, architecture dependent here!)
  - Executed by Level 1 (Digital Logic) – the hardware!
- We study levels 2 & 3 in this class (again, mostly level 3)
- (We study levels 1 & 2 in Computer Organization)

Level 1: Digital Logic

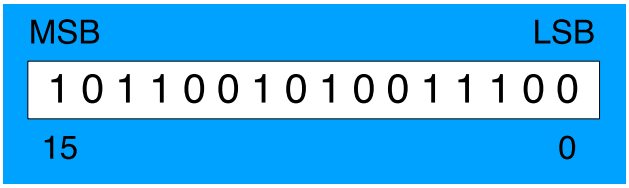
- CPU, constructed from digital logic gates
  - Yes! Those logic operations: AND, OR, NOT, XOR, NOR, NAND, etc!
  - These are the basic building gates of the functional components of the CPU & the computer
  - The machine code instructions the functional units (like the ALU, MMU, etc.) what to do
    - Example: load data from memory, add data to a register, save register to memory, etc.)
  - Some System Components:
    - System bus
    - Memory
    - Implemented using bipolar (on/off) transistors
- We study levels 1 & 2 in Computer Organization

Data Representations: Numbers

- We like/think-about numbers in base 10, computers like/think in base 2 – problem?
- Base ten numbers are stored as Binary Numbers in the machine
  - Translating between binary & decimal
- Binary Addition
  - Performed the same? Differently? How?!
- Integer Storage Sizes
  - What does storage size impact?
- Hexadecimal Integers (where do we use these?)
  - Translating between decimal and hexadecimal
  - Hexadecimal subtraction
- Signed Integers
  - Binary subtraction? Negative values?
- Character Storage – how do we do that?!

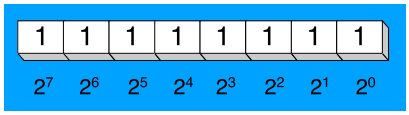
Binary Numbers

- Digits are 1 and 0
  - 1 = true (or on, or high)
  - 0 = false (or off, or low)
- MSB – most significant bit
- LSB – least significant bit
- Bit numbering:



Binary Numbers

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:



Every binary number is a sum of powers of 2

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Translating Binary to Decimal

- Weighted positional notation shows how to calculate the decimal value of each binary bit:

  Decimal = $(D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)$
  D_n = binary digit @ location n

- You did this way back in elementary school when you learned the base-10 number system!
  The One's column, the Ten's column, the Hundreds, . . .

- We apply the same concept for base two (or for any radix that we need, for that matter)

- Binary 00001001 = decimal 9:
  $(1 \times 2^3) + (1 \times 2^0) = 9$

Translating Unsigned Decimal to Binary

- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

37 = 100101

Practice!

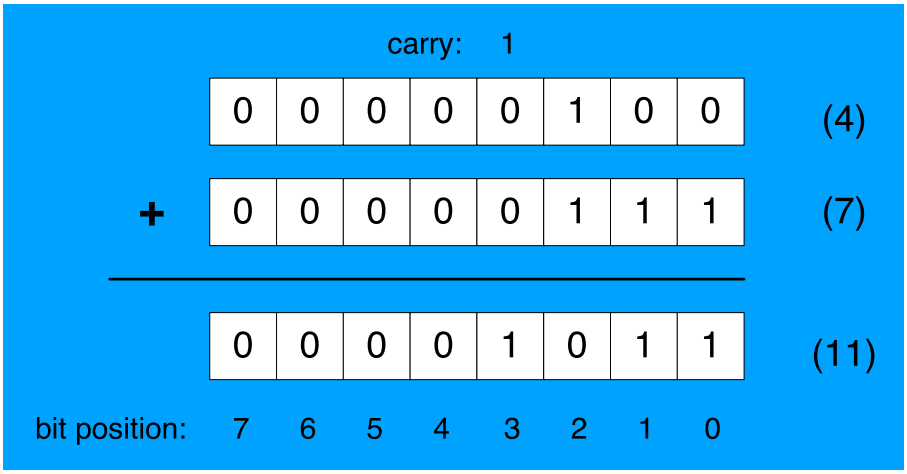
- Convert the following:

Binary to Base Ten	
101	
1011	
1101	
11101	
1010101	
101010	

Base Ten to Binary	
1	
2	
4	
9	
15	
251	

Binary Addition

- Starting with the LSB, add each pair of digits, include the carry if present.



Hexadecimal Integers

Binary values represented in decimal & hexadecimal

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Translating Binary to Hexadecimal

Each hexadecimal digit corresponds to 4 binary bits.

Example: Translate the binary integer
000101101010011110010100 to hexadecimal:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Group four bytes (nibble) together:
0001 0110 1010 0111 1001 0100

Convert each group of four bits to hex value.

Works nicely b/c powers of 2 & 16
can be represented by powers of 2.

Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16
  decimal

- Thus, a 4-digit hex number converted to decimal
  $= (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$

- Hex 1234 equals
  $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660

- Hex 3BA4 equals
  $(3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268

Converting Decimal to Hexadecimal

Division	Quotient	Remainder
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

decimal 422 = 1A6 hexadecimal

Hexadecimal Addition

- Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

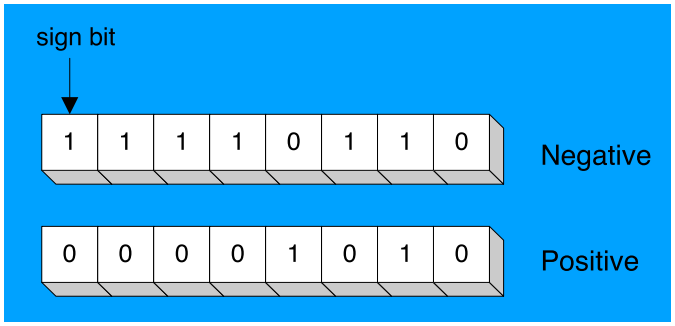
```
          1   1
    36  28  28  6A
+   42  45  58  4B
    78  6D  80  B5
```

A + B = 21;
21 / 16 = 1, rem 5

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

Signed Integers

- The highest bit indicates the sign:
  1 = negative, 0 = positive



If the highest digit of a hexadecimal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D

So we are “stealing” one bit that could be used for “amplitude” and using it to represent sign.

Forming the Two's Complement

- Negative numbers are stored in two's complement notation

- This form represents the additive Inverse

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two’s complement representation	11111111

Note that 00000001 + 11111111 = 00000000

Positive 1: 00000001
Negative 1: 11111111

Two's Compliment Shortcut

- Start from LSB to MSB (right to left)
  - Copy each bit until reach first one (1) bit
  - Copy the first one bit
  - Flip all remaining bits

- Example, make the following negative:
  - `0001` ➜ `1111`
  - `0010` ➜ `1110`
  - `0110` ➜ `1010`
  - `01101` ➜ `10011`

- Practice converting these back, should reach same (starting) value

Binary Subtraction

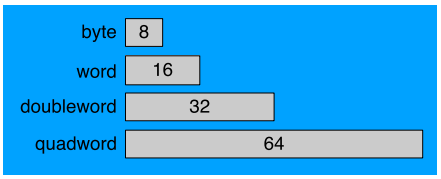
- When subtracting A – B, convert B to its two's complement
  Add A to (–B)

```
  0 0 0 0 1 1 0 0          0 0 0 0 1 1 0 0
- 0 0 0 0 0 0 1 1        + 1 1 1 1 1 1 0 1
-----------------        -----------------
  0 0 0 0 1 0 0 1          0 0 0 0 1 0 0 1
```

Practice:
Subtract 00101 from 01001
Subtract 01111 from 01000

Unsigned Integer Storage Sizes

Standard sizes:



Storage Type	Size	Range (low-high)	Powers of 2
byte	8 bits	0 - 255	0 to (2^8 - 1)
word	2 bytes	0 - 65,535	0 to (2^{16} - 1)
double word	4 bytes	0 - 4,294,967,295	0 to (2^{32} - 1)
quad word	8 bytes	0 - 18,446,744,073,709,551,615	0 to (2^{64} - 1)

Note: Word size is processor/architecture dependent,
can range from 8, 16, to 32 bits (usually 16 or 32)

Exercise:
What is the largest unsigned integer that may be stored in 19 bits?

Ranges of Signed Integers

The highest bit is reserved for the sign.
This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	−128 to +127	-2^7 to $(2^7 - 1)$
Signed word	−32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	−2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the largest positive (signed) value that may be stored in 18 bits?

How about unsigned?

Note Well, the difference between 2s compliment and just using a sign bit:
Just tacking on sign bit gives pos/neg zero (uses two values)
2s compliment does not have “double zeros”

Character Storage

- What about characters?!

- Character sets
  - Standard ASCII, 7-bit (0 – 127)
  - Extended ASCII, 8-bit (0 – 255)
  - ANSI (0 – 255)
  - Unicode, 16-bit (0 – 65,535)

- Null-terminated String
  - Array of characters followed by a null byte

Dec	Hex	Character
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z

Dec	Hex	Character
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z

Numeric Data Representation

- Many ways we can store/represent data:

- Pure binary
  - Can be calculated directly

- ASCII binary
  - String of digits (characters): "01010101"

- ASCII decimal
  - String of digits (characters): "65"

- ASCII hexadecimal
  - String of digits (characters): "9C"

- Can’t do “math” with strings, only pure values

Segue to Boolean Operations

- NOT
- AND
- OR
- Operator Precedence
- Truth Tables

Boolean Algebra

- Based on symbolic logic, designed by George Boole
- Boolean expressions created from:
  - NOT, AND, OR, XOR

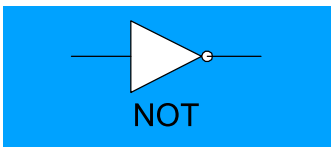
Expression	Description
¬X	NOT X
X ∧ Y	X AND Y
X ∨ Y	X OR Y
¬X ∨ Y	(NOT X) OR Y
¬(X ∧ Y)	NOT (X AND Y)
X ∧ ¬Y	X AND (NOT Y)

NOT

- Inverts (reverses) a boolean value
- Truth table for Boolean NOT operator:

X	¬X
F	T
T	F

Digital gate diagram for NOT:

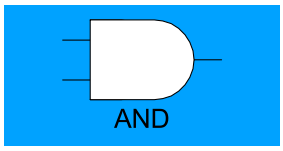


AND

• Truth table for Boolean AND operator:

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

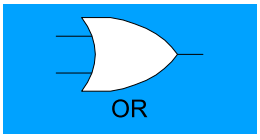


OR

• Truth table for Boolean OR operator:

X	Y	X ∨ Y
F	F	F
F	T	T
T	F	T
T	T	T

Digital gate diagram for OR:

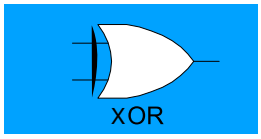


XOR

- Truth table for Boolean XOR operator:

X	Y	X ⊕ Y
F	F	F
F	T	T
T	F	T
T	T	F

Digital gate diagram for XOR:



Operator Precedence

- Examples showing the order of operations:

- Typically precedence is:
  - NOT
  - AND
  - OR

Expression	Order of Operations
¬X ∨ Y	NOT, then OR
¬(X ∨ Y)	OR, then NOT
X ∨ (Y ∧ Z)	AND, then OR

Truth Tables (1 of 3)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.

- A truth table shows all the inputs and outputs of a Boolean function

Example (inputs X, Y): ¬X ∨ Y

X	¬X	Y	¬X ∨ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

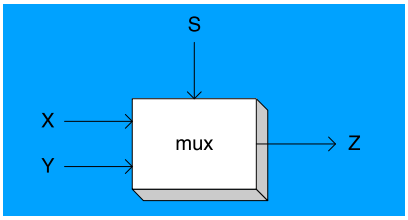
Truth Tables (2 of 3)

- Example: X ∧ ¬Y

X	Y	¬Y	X ∧ ¬Y
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F

Truth Tables (3 of 3)

• Example: (Y ∧ S) ∨ (X ∧ ¬S)



Two-input multiplexer

X	Y	S	**Y ∧ S**	**¬S**	**X ∧ ¬S**	**(Y ∧ S) ∨ (X ∧ ¬S)**
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T

Logic

• Another way to look at the operations:

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, ...} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, ...} contains an odd number of ones.

x86 Family of CPUs

- 8086
  - 16-bit registers
    - AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP
  - Real mode only (memory access, any location)
    - 1MB main memory
- 80286
  - 16-bit registers
  - 16-bit Protected mode (no access memory from other processes)
    - 16 MB main memory
- 80386
  - 32-bit registers (not all, but many)
    - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, FS, GS
  - 32-bit protected mode
    - 4 GB main memory

x86 Family of CPUs

- 80486
  - Same as 80386, just better instruction execution
- MMX
  - CPU w/multimedia instructions to improve graphics performance
- Dual/Quad Core
  - One chip, 2/4 processors on board
- i series (i3, i5, i7, i9)
  - 64-bit Registers
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RIP, FS, GS
- We will play in 32-bit land! (EAX, EBX, . . .)

Intel Register Sets

16bit	32bit	64bit	Description
AX	EAX	RAX	The accumulator register
BX	EBX	RBX	The base register
CX	ECX	RCX	The counter
DX	EDX	RDX	The data register
SP	ESP	RSP	Stack pointer
BP	EBP	RBP	Points to the base of the stack frame
	Rn	RnD	(n = 8...15) General purpose registers
SI	ESI	RSI	Source index for string operations
DI	EDI	RDI	Destination index for string operations
IP	EIP	RIP	Instruction Pointer
FLAGS			Condition codes

Now onto Assembly!!

- Assembly is just human readable machine code
- Instructions usually consist of a mnemonic and possibly some operand(s), of the form:

  mnemonic [operands]

- mnemonics are operators (opcodes), like:
  - add, sub
  - inc, dec
  - mul, div
  - call, ret
  - jmp
  - mov

Operands

- Can usually be one of:
  - Register name
  - Memory address
  - Immediate value
  - Implied

- Example:

```
mov   r2, r1
```

- Moves value stored in r1 to r2

- Might be stored on machine as:
  101 010 001
  (fictitious 9-bit encoding!)

assuming:
“mov” is 101
r1 is 001
r2 is 010

Assembler Process (Refresh)

- Assembly is a “Level 3” language
  - Must be translated to lower lever (instruction set)

- We use compilers for high-level languages (e.g., Java, C)

- We use assembler for assembly
  - Creates object files
  - Objects may need to be linked to create program

Summary

- Assembly language helps you learn how software is constructed at the lowest levels
- Assembly language has a one-to-one relationship with machine language
  - Unless of course, it’s a macro!
- Each layer in a computer's architecture is an abstraction of a machine
  - Layers can be hardware or software
- Boolean expressions are essential to the design of computer hardware and software