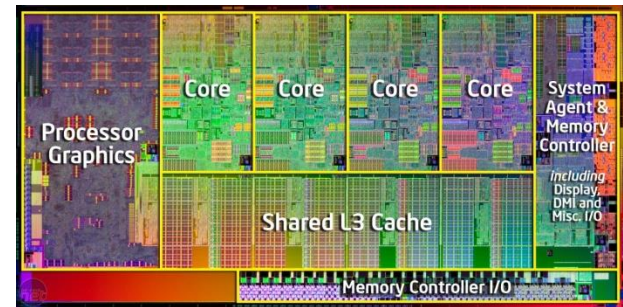




COSE222, COMP212, CYDF210 Computer Architecture

Lecture 2. Instructions and High-level to Machine Code

Prof. Taeweon Suh
Computer Science & Engineering
Korea University

Abstraction

- Abstraction helps us deal with complexity
 - Hide lower-level detail
- Instruction set architecture (ISA)
 - An abstraction interface between the hardware and the low-level software

Abstraction Analogies

Driver



**Abstraction
layer**



**Machine
Details**



**Combustion
Engine in a car**



**Break system
in a car**

Customer



**Abstraction
layer**



**Machine
Details**



**Hardware board in a
vending machine**

Abstractions in Computer



Programming using APIs



facebook



Operating Systems



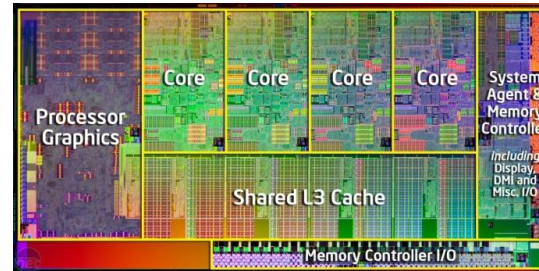
Provides APIs (Application Programming Interface)

Instruction Set Architecture (ISA)



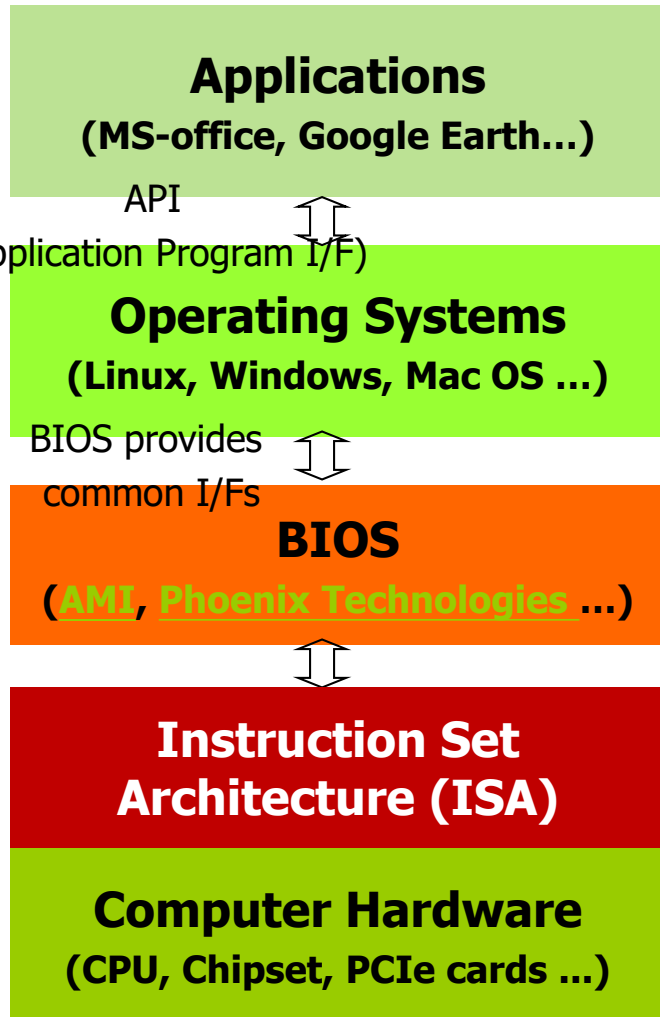
**Assembly language
or
Machine language**

**Hardware Implementation
(x86)**



**Our focus in
this course**

Hardware/Software Stack in Computer



- Application software
 - Written in high-level language
- System software
 - Compilers
 - Translates the code written in high-level language to machine code
 - Operating Systems
 - Handling input/output
 - Managing memory and storage
 - Scheduling tasks & sharing resources
 - BIOS (Basic Input/Output System)
- ISA
 - Interface between hardware and low-level software
- Hardware
 - Processor, memory, I/O controllers

Instructions

- If you want to talk to foreigners, you should be able to speak their languages
 - Likewise, to talk to a computer, you must speak its language
- The words of a computer's language are called **instructions**
- The collection of instructions is called **instruction set**
- **Different CPUs have different instruction sets**
 - x86, MIPS, and ARM have their own instruction sets
 - But, they have many aspects in common

Examples

• MIPS

MIPS operands

Name	Example	Comments
32 registers	\$0, \$1, \$2, ..., \$31, Hi, Lo	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$0 always equals 0. Register \$1 is reserved for the assembler to handle pseudoinstructions and large constants. Hi and Lo are 32-bit registers containing the results of multiply and divide.
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
	subtract	sub \$1,\$2,\$3	\$1 = \$2 - \$3	3 operands; exception possible
	add immediate	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; exception possible
	add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; no exceptions
	subtract unsigned	subu \$1,\$2,\$3	\$1 = \$2 - \$3	3 operands; no exceptions
	add imm. unsign.	addiu \$1,\$2,100	\$1 = \$2 + 100	+ constant; no exceptions
	Move fr. copr. reg.	mfc0 \$1,\$cpc	\$1 = \$cpc	Used to get exception PC
	multiply	mult \$2,\$3	Hi, Lo = \$2 * \$3	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$2,\$3	Hi, Lo = \$2 * \$3	64-bit unsigned product in Hi, Lo
	divide	div \$2,\$3	Lo = \$2 / \$3, Hi = \$2 mod \$3	Lo = quotient, Hi = remainder
Logical	divide unsigned	divu \$2,\$3	Lo = \$2 / \$3, Hi = \$2 mod \$3	Unsigned quotient and remainder
	Move from Hi	mghi \$1	\$1 = Hi	Used to get copy of Hi
	Move from Lo	mfl0 \$1	\$1 = Lo	Used to get copy of Lo
	and	and \$1,\$2,\$3	\$1 = \$2 & \$3	3 register operands; logical AND
	or	or \$1,\$2,\$3	\$1 = \$2 \$3	3 register operands; logical OR
	and immediate	andi \$1,\$2,100	\$1 = \$2 & 100	Logical AND register, constant
	or immediate	ori \$1,\$2,100	\$1 = \$2 100	Logical OR register, constant
	shift left logical	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by constant
	shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
Data transfer	load word	lw \$1,100(\$2)	\$1 = Memory[\$2+100]	Data from memory to register
	store word	sw \$1,100(\$2)	Memory[\$2+100] = \$1	Data from register to memory
	load upper imm.	lui \$1,100	\$1 = 100 x 2 ¹⁶	Loads constant in upper 16 bits
	branch on equal	beq \$1,\$2,100	if (\$1 == \$2) go to PC+4+100	Equal test; PC relative branch
Conditional branch	branch on not eq.	bne \$1,\$2,100	if (\$1 != \$2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0	Compare less than; 2's complement
	set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0	Compare < constant; 2's comp.
	set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0	Compare less than; natural number
Unconditional jump	set l.t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0	Compare < constant; natural
	jump	j 10000	go to 10000	Jump to target address
	jump register	jr \$31	go to \$31	For switch, procedure return
	jump and link	jal 10000	\$31 = PC + 4; go to 10000	For procedure call

Main MIPS assembly language instruction set. The floating-point instructions are shown in Figure 4.44 on page 241. Appendix gives the full MIPS assembly language instruction set.

• x86

Instruction Set Summary

Instr	Addressing Mode	Assembler Format	Operation	Op Code Hex	Dec	Bytes	Clock Cycles	Status Register - P	Instr
ADC	Immediate	ADC #oper	A ← A + C → A, C	69	105	2	2	N V D I Z C	ADC
	Zero Page	ADC addr	A ← [addr] + C → A, C	65	101	2	3	✓ - - - - ✓	
	Zero Page, X	ADC addr, X	A ← [addr + X] + C → A, C	75	117	2	4	✓ - - - - ✓	
	Absolute	ADC ADDR	A ← [ADDR] + C → A, C	6D	109	3	4	✓ - - - - ✓	
	Absolute, X	ADC ADDR, X	A ← [ADDR + X] + C → A, C	7D	125	3	4*	✓ - - - - ✓	
	Absolute, Y	ADC ADDR, Y	A ← [ADDR + Y] + C → A, C	79	121	3	4*	✓ - - - - ✓	
	(Indirect, X)	ADC (addr, X)	A ← [[addr + X + 1, addr + X]] + C → A, C	61	97	2	6	✓ - - - - ✓	
	(Indirect, Y)	ADC (addr, Y)	A ← [[addr + 1, addr + Y] + C → A, C	71	113	2	5*	✓ - - - - ✓	
AND	Immediate	AND #oper	A ← A & A	29	41	2	2	N V D I Z C	AND
	Zero Page	AND addr	A ← [addr] & A	25	37	2	3	✓ - - - - ✓	
	Zero Page, X	AND addr, X	A ← [addr + X] & A	35	53	2	4	✓ - - - - ✓	
	Absolute	AND ADDR	A ← [ADDR] & A	2D	45	3	4	✓ - - - - ✓	
	Absolute, X	AND ADDR, X	A ← [ADDR + X] & A	3D	61	3	4*	✓ - - - - ✓	
	Absolute, Y	AND ADDR, Y	A ← [ADDR + Y] & A	39	57	3	4*	✓ - - - - ✓	
	(Indirect, X)	AND (addr, X)	A ← [[addr + X + 1, addr + X]] & A	21	33	2	6	✓ - - - - ✓	
	(Indirect, Y)	AND (addr, Y)	A ← [[addr + 1, addr + Y] & A	31	49	2	5*	✓ - - - - ✓	
ASL	Accumulator	ASL A	A (←) → A : bit 0, bit 7 → C	0A	10	1	2	N V D I Z C	ASL
	Zero Page	ASL addr	ASL [addr] (←) → [addr]	06	6	2	5	✓ - - - - ✓	
	Zero Page, X	ASL addr, X	[addr + X] (←) → [addr + X]	16	22	2	6	✓ - - - - ✓	
	Absolute	ASL ADDR	[ADDR] (←) → [ADDR]	0E	14	3	6	✓ - - - - ✓	
BCC BCS BEQ BNE BMI BPL BVS BVC	Relative	BCC oper	Branch on C = 0	90	144	2	2*	N V D I Z C	BCC BCS BEQ BNE BMI BPL BVS BVC
	Relative	BCS oper	Branch on C = 1	98	176	2	2*	✓ - - - - ✓	
	Relative	BEQ oper	Branch on Z = 1	F0	240	2	2*	✓ - - - - ✓	
	Relative	BNE oper	Branch on Z = 0	D0	208	2	2*	✓ - - - - ✓	
	Relative	BMI oper	Branch on N = 1	30	48	2	2*	✓ - - - - ✓	
	Relative	BPL oper	Branch on N = 0	10	16	2	2*	✓ - - - - ✓	
	Relative	BVS oper	Branch on V = 1	70	112	2	2*	✓ - - - - ✓	
	Relative	BVC oper	Branch on V = 0	50	80	2	2*	✓ - - - - ✓	
BIT	Zero Page	BIT addr	A & [addr] : bit 7 → N, bit 6 → V	24	36	2	3	N V D I Z C	BIT
	Absolute	BIT ADDR	A & [ADDR]	2C	44	3	4	b ₇ b ₆ - - - -	
BRK	Implied	BRK	1 → B flag PC + 2 + P 4, [FFFF] → PCL, [FFFF] → PCH	00	0	1	7	- - - - - 1 - -	BRK
CLC CLD CLI CLV	Implied	CLC	0 → C	18	24	1	2	N V D I Z C	CLC CLD CLI CLV
	Implied	CLD	0 → D	D8	216	1	2	- - - - - 0	
	Implied	CLI	0 → I	58	88	1	2	- - - - - 0	
	Implied	CLV	0 → V	B8	184	1	2	- - - - - 0	
CMP	Immediate	CMP #oper	A - #	C9	201	2	2	N V D I Z C	CMP
	Zero Page	CMP addr	A - [addr]	C5	197	2	3	✓ - - - - ✓	
	Zero Page, X	CMP addr, X	A - [addr + X]	D5	213	2	4	✓ - - - - ✓	
	Absolute	CMP ADDR	A - [ADDR]	C0	208	3	4	✓ - - - - ✓	
	Absolute, X	CMP ADDR, X	A - [ADDR + X]	DD	221	3	4*	✓ - - - - ✓	
	Absolute, Y	CMP ADDR, Y	A - [ADDR + Y]	D9	217	3	4*	✓ - - - - ✓	
	(Indirect, X)	CMP (addr, X)	A - [[addr + X + 1, addr + X]]	D1	193	2	6	✓ - - - - ✓	
	(Indirect, Y)	CMP (addr, Y)	A - [[addr + 1, addr + Y]]	D0	209	2	5*	✓ - - - - ✓	
CPX	Immediate	CPX #oper	X - #	E0	224	2	2	N V D I Z C	CPX
	Zero Page	CPX addr	X - [addr]	E4	228	2	3	✓ - - - - ✓	
	Absolute	CPX ADDR	X - [ADDR]	EC	236	3	4	✓ - - - - ✓	
CPY	Immediate	CPY #oper	Y - #	C0	192	2	2	N V D I Z C	CPY
	Zero Page	CPY addr	Y - [addr]	C4	196	2	3	✓ - - - - ✓	
	Absolute	CPY ADDR	Y - [ADDR]	CC	204	3	4	✓ - - - - ✓	
DEC	Zero Page	DEC addr	[addr] - 1 → [addr]	C6	198	2	5	N V D I Z C	DEC
	Zero Page, X	DEC addr, X	[addr + X] - 1 → [addr + X]	D6	214	2	6	✓ - - - - ✓	
	Absolute	DEC ADDR	[ADDR] - 1 → [ADDR]	CE	206	3	6	✓ - - - - ✓	
	Absolute, X	DEC ADDR, X	[ADDR + X] - 1 → [ADDR + X]	DE	222	3	7	✓ - - - - ✓	

MIPS and x86 Instruction Sets

- For the instruction sets of MIPS and x86, refer to the following links
 - Intel: <http://www.intel.com/products/processor/manuals/>
 - MIPS: <http://www.mips.com/>
 - We are going to study the MIPS ISA in detail throughout this course

Levels of Program Code

- High-level language
 - Level of abstraction closer to problem domain
 - Provides productivity and portability
- Assembly language
 - Textual and symbolic representation of instructions
- Machine code (object code or binary)
 - Binary bits of instructions and data

High-level
language
program
(in C)

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

Compiler

Assembly
language
program
(for MIPS)

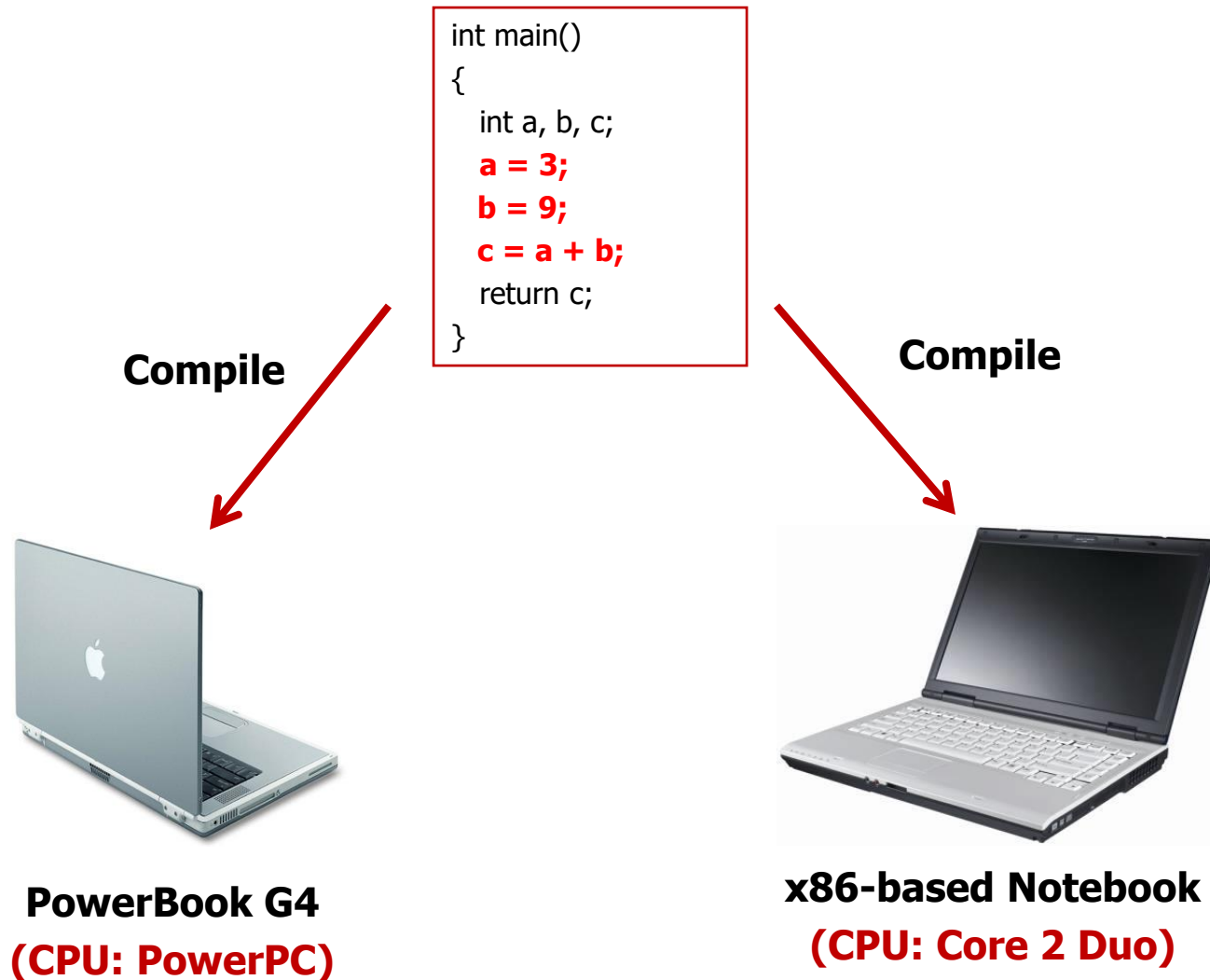
```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

Binary machine
language
program
(for MIPS)

```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
1000110011110010000000000000000100
101011001111001000000000000000000
1010110001100010000000000000000100
00000011111000000000000000001000
```

High-Level Code is Portable



Levels of Program Code

- High-level language program in C

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

one-to-many

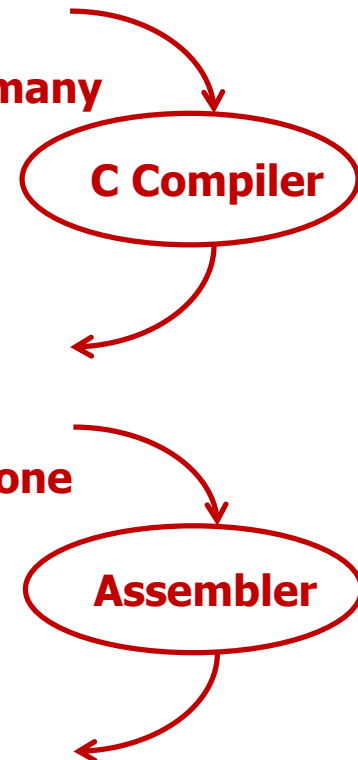
- Assembly language program (MIPS)

```
swap:      sll      $2, $5, 2
           add      $2, $4, $2
           lw       $15, 0($2)
           lw       $16, 4($2)
           sw       $16, 0($2)
           sw       $15, 4($2)
           jr       $31
```

one-to-one

- Machine (object, binary) code (MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```



High-level Code to Executable (Binary)

- What steps did you take to run your program (hello.c) on your Linux machine?

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

```
%gcc -g hello.c -o hello
    // hello is a machine code (binary or executable)
    // -g is optional
%./hello
% Hello World!
%objdump -S -D hello
    // with objdump, you can see human-readable assembly code
```

High-level Code to Executable (Binary)

```
%gcc -g simple_sub.c -o simple_sub
```

```
%objdump -S -D simple_sub
```

Instructions
(human-readable)

```
#include <stdio.h>
```

```
#define A 3
```

```
#define B 5
```

```
int main() {  
    printf("%d - %d = %d",  
        A, B, mysub(A, B));  
    return 0;  
}
```

```
int mysub(int op1, int op2) {  
    int myres;  
    myres = op1 - op2;  
    return myres;  
}
```

simple_sub.c

address

C Compiler



**Representation in
hexadecimal
(machine-readable)**

```
int mysub(int op1, int op2)
```

```
{
```

```
400563: 55
```

```
400564: 48 89 e5
```

```
400567: 89 7d ec
```

```
40056a: 89 75 e8
```

```
int myres;
```

```
myres = op1 - op2;
```

```
40056d: 8b 45 e8
```

```
400570: 8b 55 ec
```

```
400573: 89 d1
```

```
400575: 29 c1
```

```
400577: 89 c8
```

```
400579: 89 45 fc
```

```
return myres;
```

```
40057c: 8b 45 fc
```

```
push    %rbp
```

```
mov     %rsp,%rbp
```

```
mov     %edi,-0x14(%rbp)
```

```
mov     %esi,-0x18(%rbp)
```

```
mov     -0x18(%rbp),%eax
```

```
mov     -0x14(%rbp),%edx
```

```
mov     %edx,%ecx
```

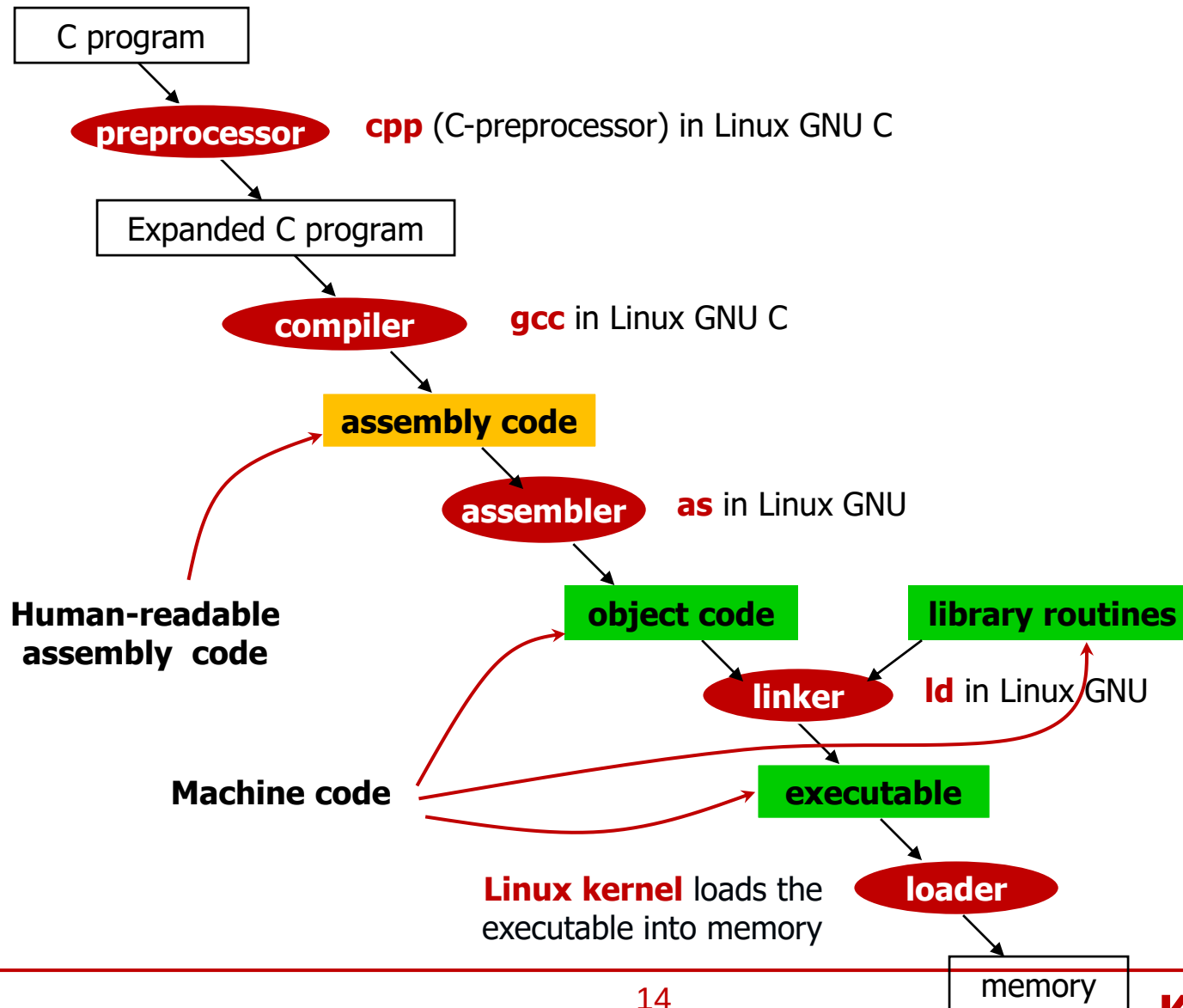
```
sub     %eax,%ecx
```

```
mov     %ecx,%eax
```

```
mov     %eax,-0x4(%rbp)
```

```
mov     -0x4(%rbp),%eax
```

High-level Code to Executable (Binary)



High-level Code to Executable (Binary)

- The command `gcc` hides all the details
- Compile `simple_sub.c` with `gcc -v simple_sub.c -o simple_sub`
 - You will see all the details of what `gcc` does for compilation
- Compilation goes through several steps to generate a machine code
 - Preprocessing
 - Compilation
 - Assembler
 - Linker

```
#include <stdio.h>

#define A 3
#define B 5

int main() {

    printf("%d - %d = %d",
           A, B, mysub(A, B));

    return 0;
}

int mysub(int op1, int op2) {

    int myres;

    myres = op1 - op2;

    return myres;
}
```

High-level Code to Executable (Binary)

- **Preprocessing**

- Use to expand macros and header files included
- `%cpp simple_sub.c > simple_sub.i`
 - open `simple_sub.i` to see what you got

- **Compilation**

- **Actual compilation of the preprocessed code to assembly language** for a specific processor
- `%gcc -S simple_sub.i`
 - Output will be stored in `simple_sub.s`
 - Open `simple_sub.s` to see what you got

- **Assembler**

- Convert **assembly language** into **machine code** and generate an object file
- `%as simple_sub.s -o simple_sub.o`
 - The resulting file `simple_sub.o` contains the machine instructions for the program, with an undefined reference to `printf`

High-level Code to Executable (Binary)

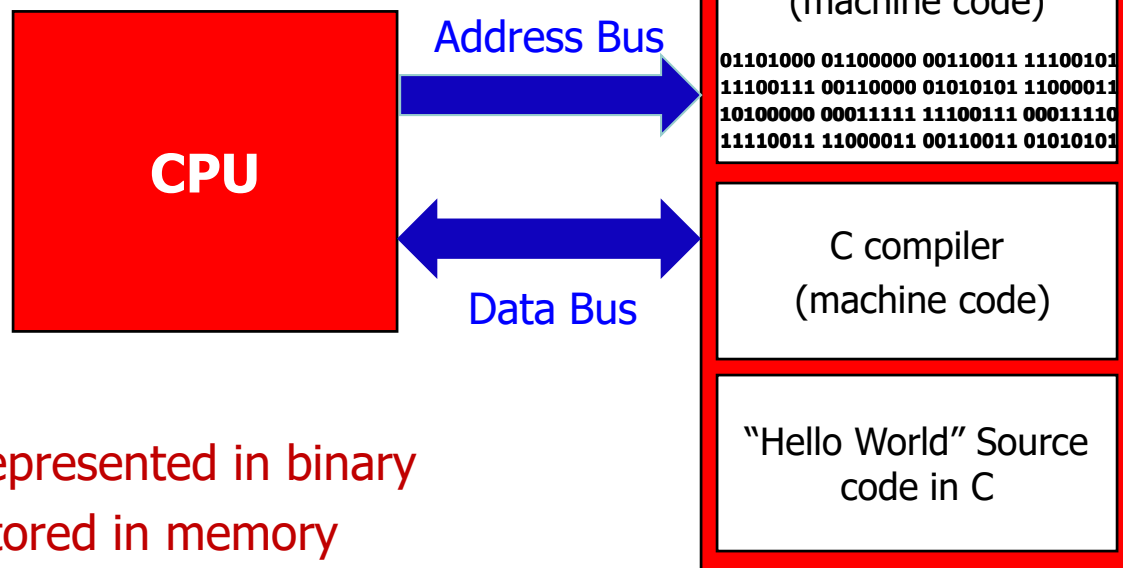
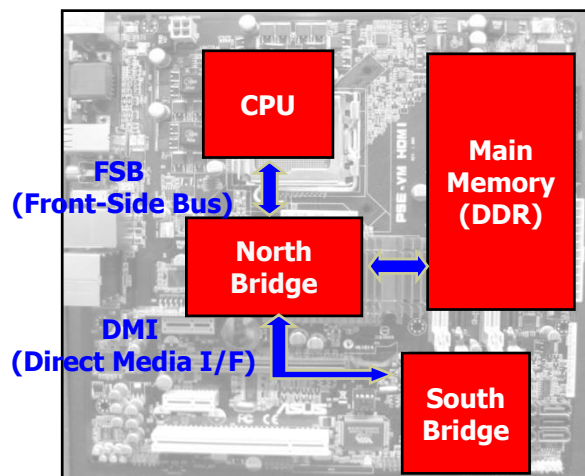
- **Linker**

- Final stage of compilation
- Linking object files to create an executable
- In practice, an executable requires many external functions from system and C run-time (crt) libraries
- Consequently, the actual link commands used internally by GCC are complicated.
- Example
 - `% ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.4.3/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.4.3 -L/usr/lib/gcc/x86_64-linux-gnu/4.4.3 -L/usr/lib -L/lib -L/usr/lib -L/usr/lib/ -L/usr/lib/x86_64-linux-gnu -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.4.3/crtend.o /usr/lib/crtn.o simple_sub.o -o simple_sub`

- **Now run your program**

- `% ./simple_sub // Linux kernel loads the program into memory`
- `% 3 - 5 = -2 // output`

Stored Program Concept



- Instructions and data are represented in binary
- Instructions and data are stored in memory
- CPU fetches instructions and data to execute
- Binary compatibility allows compiled programs to work on different computers with the same ISA
 - Standardized ISAs

Cross Compiler

- Hmm, sound good so far
- But, wait! We are talking about MIPS (not x86). Then, How to generate the MIPS machine code without a MIPS machine?
- You are still able to generate MIPS binaries on an x86 machine
 - How? Use a **cross-compiler**!!!

**Normal
compilation**

```
int main()
{
    int a, b, c;
    a = 3;
    b = 9;
    c = a + b;
    return c;
}
```



x86-based laptop

compile

x86 machine code

```
a = 3;
c7 45 f0 03 00 00 00  movl  $0x3,-0x10(%ebp)
b = 9;
c7 45 f4 09 00 00 00  movl  $0x9,-0xc(%ebp)
c = a + b;
8b 55 f4              mov   -0xc(%ebp),%edx
8b 45 f0              mov   -0x10(%ebp),%eax
01 d0                add   %edx,%eax
89 45 f8              mov   %eax,-0x8(%ebp)
```



MIPS-based laptop
(if exists)

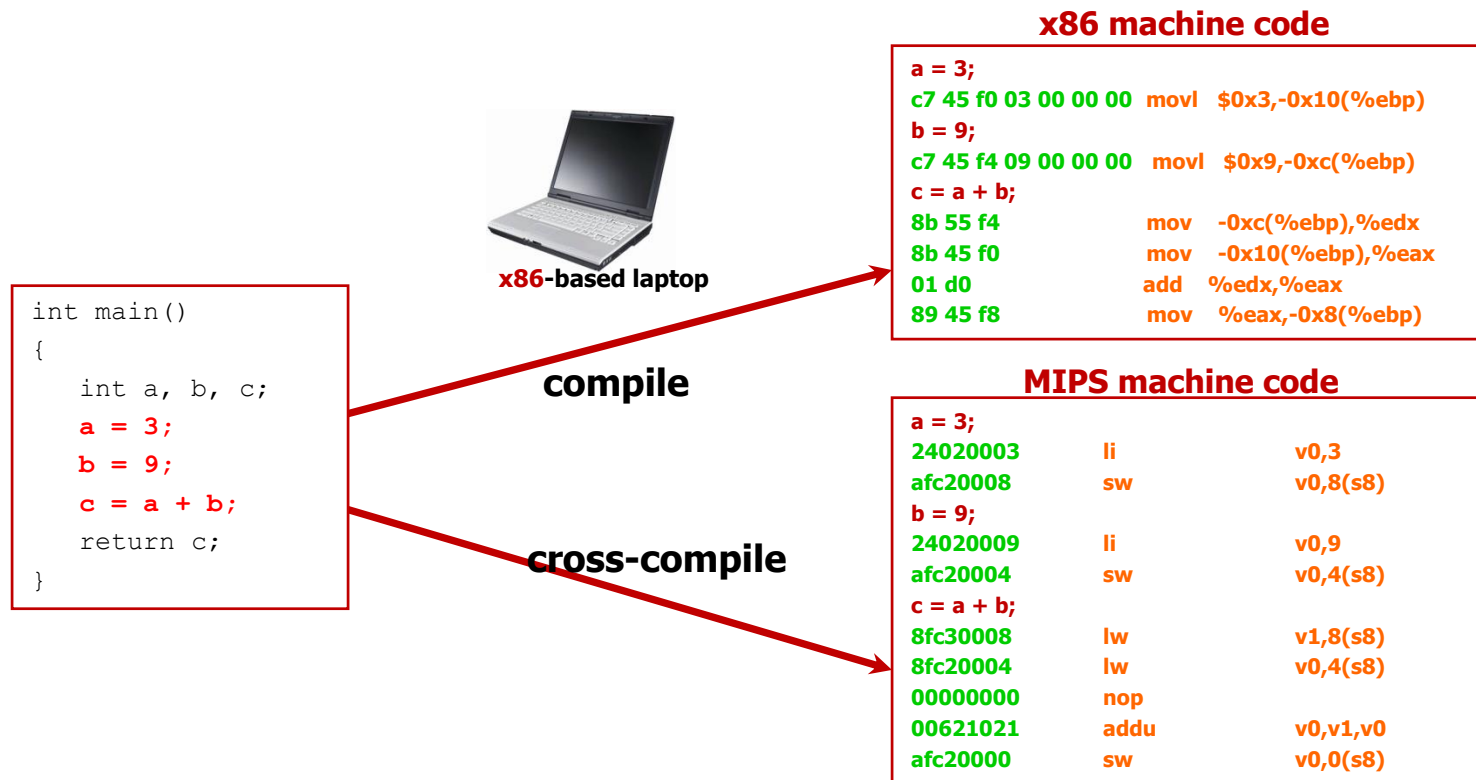
compile

MIPS machine code

```
a = 3;
24020003             li      v0,3
afc20008             sw      v0,8(s8)
b = 9;
24020009             li      v0,9
afc20004             sw      v0,4(s8)
c = a + b;
8fc30008             lw      v1,8(s8)
8fc20004             lw      v0,4(s8)
00000000             nop
00621021             addu    v0,v1,v0
afc20000             sw      v0,0(s8)
```

Cross Compiler (Cont.)

- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is run -- Wiki



MIPS Cross Compiler

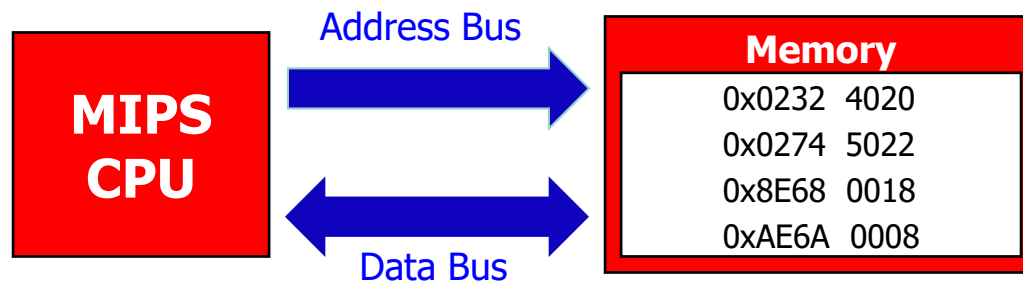
- Check out the class web for instructions on how to set up an environment to generate the MIPS code using Eclipse on Windows
- Test-generate binary from the MIPS assembly program with assembler

```
add    $t0, $s1, $s2 # $t0 <= $s1 + $s2
sub     $t2, $s3, $s4 # $t2 <= $s3 - $s4
lw      $t0, 24($s3)  #load (read) word from memory
                        # $t0 <= [$s3 + 24]
sw      $t2, 8($s3)   # store(write) word to memory
                        # [$s3 + 8] <= $t2
```

assembler

```
0x0232 4020
0x0274 5022
0x8E68 0018
0xAE6A 0008
```

Don't worry. We are going to talk deep about this!



MIPS Instruction examples in 2 forms

- Human-readable form

```
addi $2, $0, 5    // $2 = $0 + 5
sub  $7, $7, $2    // $7 = $7 - $2
and  $5, $3, $4    // $5 = $3 & $4
```

- Machine-readable form

```
0010 0000 0000 0010 0000 0000 0000 0101 = 0x20020005 // addi $2, $0, 5
0000 0000 1110 0010 0011 1000 0010 0010 = 0x00e23822 // sub  $7, $7, $2
0000 0000 0110 0100 0010 1000 0010 0100 = 0x00642824 // and  $5, $3, $4
```

In this Course...

- You need to write some (or many) MIPS assembly programs
 - Use MIPS assembler to assemble your (assembly) program and a linker to generate executable (binary)
- You will also use preprocessor and compiler to generate assembly code from **simple** C code
 - The compiler course hopefully covers details about preprocessing and compiler (and assembler, linker, and loader)
- Then, **run your code in the MIPS simulator called SPIM**
 - **Also, run your code on the CPU you'll design later!**
- We are going to go over MIPS instructions in great detail