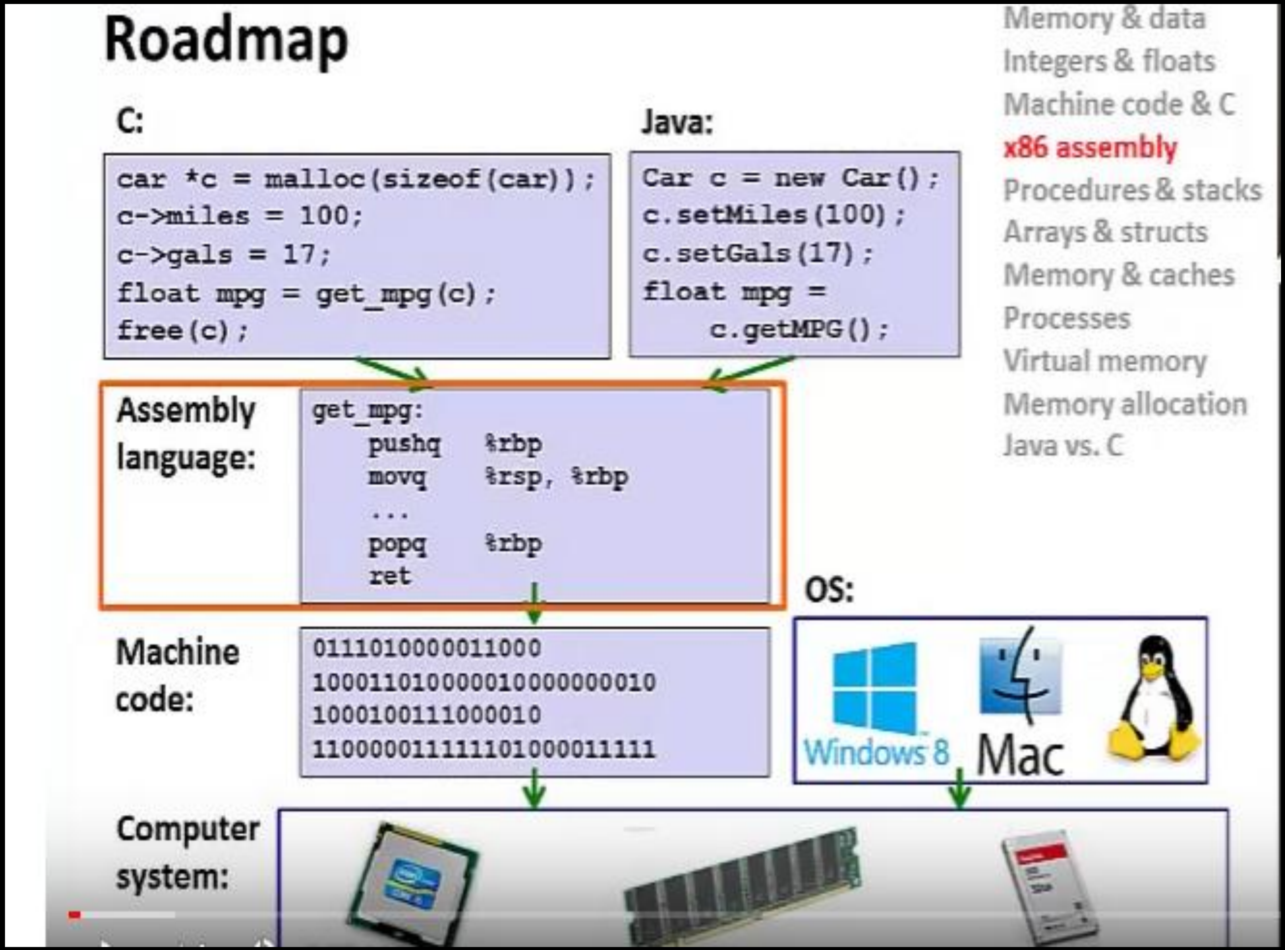
Assembly Programming

Lesson 14

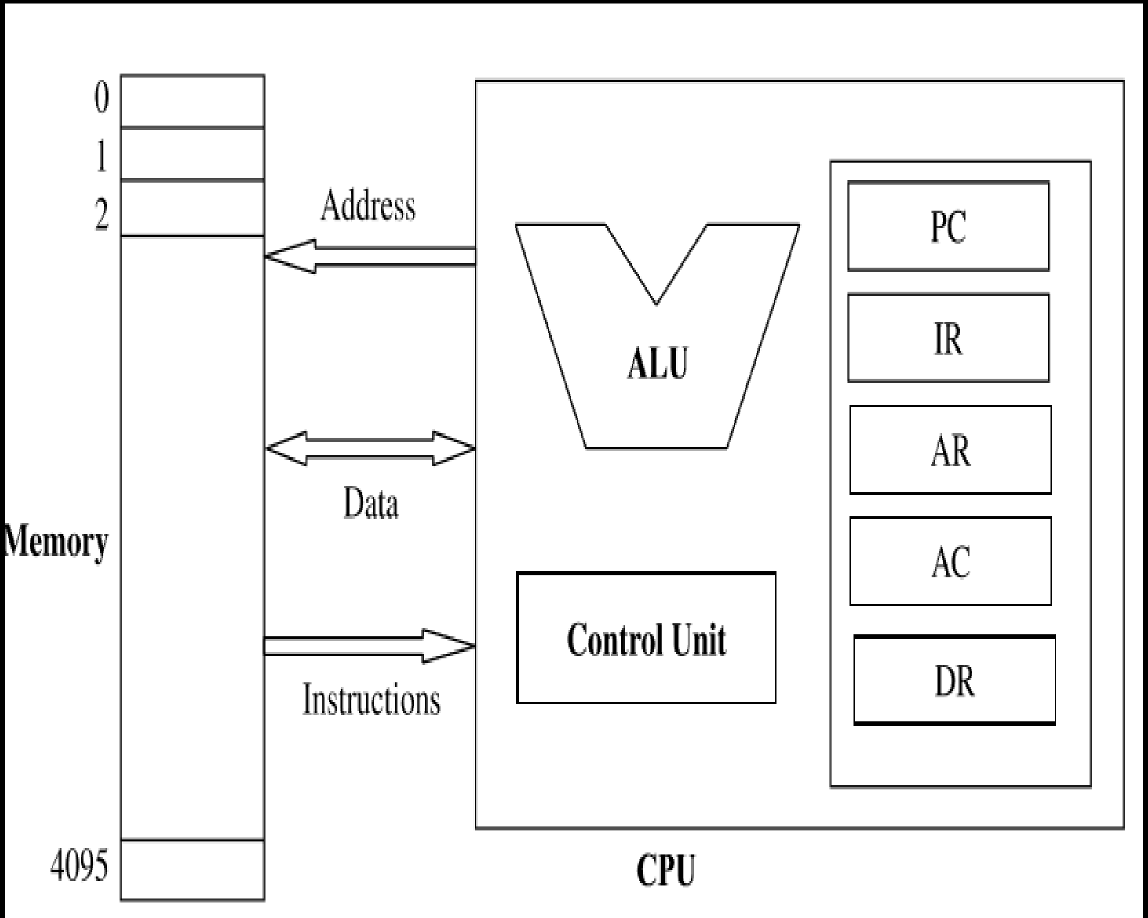
- A computer program can be represented at different levels of abstraction.
- A program could be written in a machine-independent high-level language, such as Java or C.
- A computer can execute programs only when they are represented in machine language specific to **its architecture.**
- A machine language program for a given architecture is a collection of machine instructions represented in binary form.



- Programs written at any level higher than the machine language must be translated to the binary representation before a computer can execute them.
- An assembly language program is a symbolic representation of the machine language program.
- Machine language is pure binary code, whereas assembly language is a direct mapping of the binary code onto a symbolic form that is easier for humans to understand and manage.
- Converting the symbolic representation into machine language is performed by a special program called the assembler.
- An assembler is a program that accepts a symbolic language program (source) and produces its machine language equivalent (target).

- In translating a program into binary code, the assembler will replace symbolic addresses by numeric addresses, replace symbolic operation codes by machine operation codes, reserve storage for instructions and data, and translate constants into machine representation.
- Machine language is the native language of a given processor.
- Assembly language is the symbolic form of machine language - each different type of processor has its own unique assembly language.

- Before we study the assembly language of a given processor:
  - we need first to understand the details of that processor (e.g. MIPS).
  - We need to know the memory size and organization, the processor registers, the instruction format, and the entire instruction set.



Example machine is an accumulator-based processor, which has five 16-bit registers:

1) Program Counter (PC),
2) Instruction Register (IR),
3) Address Register (AR),
4)Accumulator (AC),
5) Data Register (DR).

- The PC contains the address of the next instruction to be executed.
- The IR contains the operation code portion of the instruction being executed.
- The AR contains the address portion (if any) of the instruction being executed.
- The AC serves as the implicit source and destination of data.
- The DR is used to hold data.
- The memory unit is made up of 4096 words of storage. The word size is 16 bits.

- We assume that processors support three types of instructions:
  - Data transfer,
  - Data processing,
  - Program control.
- The data transfer operations are:
  - load, store, and move data between the registers AC and DR.
- The data processing instructions are:
  - add, subtract, and, and not.
- The program control instructions are:
  - jump and conditional jump.

Example 1

Write a machine language program that adds the contents of memory location 12 (00C-hex), initialized to 350 and memory location 14 (00E-hex), initialized to 96, and store the result in location 16 (010-hex), initialized to 0.

The program is given in binary instructions. The first column gives the memory location in binary for each instruction and operand. The second column

Instruction Set of the Simple Processor

Operation code	Operand	Meaning of instruction
0000		Stop execution
0001	*adr*	Load operand from memory (location adr) into AC
0010	*adr*	Store contents of AC in memory (location adr)
0011		Copy the contents AC to DR
0100		Copy the contents of DR to AC
0101		Add DR to AC
0110		Subtract DR from AC
0111		And bitwise DR to AC
1000		Complement contents of AC
1001	*adr*	Jump to instruction with address adr
1010	*adr*	Jump to instruction adr if AC = 0

Simple Machine Language Program in Binary

Memory location (bytes)	Binary instruction	Description
0000 0000 0000	0001 0000 0000 1100	Load the contents of location 12 in AC
0000 0000 0010	0011 0000 0000 0000	Move contents of AC to DR
0000 0000 0100	0001 0000 0000 1110	Load the contents of location 14 into AC
0000 0000 0110	0101 0000 0000 0000	Add DR to AC
0000 0000 1000	0010 0000 0001 0000	Store contents of AC in location 16
0000 0000 1010	0000 0000 0000 0000	Stop
0000 0000 1100	0000 0001 0101 1110	Data value 350
0000 0000 1110	0000 0000 0110 0000	Data value is 96
0000 0001 0000	0000 0000 0000 0000	Data value is 0

INSTRUCTION MNEMONICS AND SYNTAX

- Assembly programs are written with short abbreviations called **mnemonics**.
- A mnemonic is an abbreviation that represents the actual machine instruction.
- Assembly language programming is the writing of machine instructions in mnemonic form, where each machine instruction (binary or hex value) is replaced by a mnemonic.
- Clearly the use of mnemonics is more meaningful than that of hex or binary values, which would make programming at this low level easier and more manageable.

- An assembly program consists of a sequence of assembly statements, where statements are written one per line.
- Each line of an assembly program is split into the following four fields:

1) label, 2) operation code (opcode), 3) operand, and 4) comments.

- Figure shows the four-column format of an assembly instruction:

Label (Optional)	Operation Code (Required)	Operand (Required in some instructions)	Comment (Optional)

- **A label** is an identifier that can be used on a program line in order to branch to the labeled line.
- It can also be used to access data using symbolic names.
- Assembly languages for some processors require a colon after each label while others do not.
- For example, **SPARC** assembly requires a colon after every label, but **Motorola** assembly does not. The **Intel** assembly requires colons after code labels but not after data labels.

- The **operation code** (opcode) field contains the symbolic abbreviation of a given operation.
- The **operand** field consists of additional information or data that the opcode requires.
- The operand field may be used to specify constant, label, immediate data, register, or an address.
- The **comments** field provides a space for documentation to explain what has been done for the purpose of debugging and maintenance.

Instruction's binary representation

Operation code	Operand	Meaning of instruction
0000		Stop execution
0001	*adr*	Load operand from memory (location adr) into AC
0010	*adr*	Store contents of AC in memory (location adr)
0011		Copy the contents AC to DR
0100		Copy the contents of DR to AC
0101		Add DR to AC
0110		Subtract DR from AC
0111		And bitwise DR to AC
1000		Complement contents of AC
1001	*adr*	Jump to instruction with address adr
1010	*adr*	Jump to instruction adr if AC = 0

Instructions' mnemonic representation

Mnemonic	Operand	Meaning of instruction
STOP		Stop execution
LD	*x*	Load operand from memory (location x) into AC
ST	*x*	Store contents of AC in memory (location x)
MOVAC		Copy the contents AC to DR
MOV		Copy the contents of DR to AC
ADD		Add DR to AC
SUB		Subtract DR from AC
AND		And bitwise DR to AC
NOT		Complement contents of AC
BRA	*adr*	Jump to instruction with address *adr*
BZ	*adr*	Jump to instruction *adr* if AC = 0

- The following is the assembly code of the machine language program of Example in the previous section.

The example in mnemonic form

```
      LD X                \ AC ← X
      MOVAC               \ DR ← AC
      LD Y                \ AC ← Y
      ADD                 \ AC ← AC + DR
      ST Z                \ Z ← AC
      STOP
X     W        350        \ reserve a word initialized to 350
Y     W        96         \ reserve a word initialized to 96
Z     W        0          \ result stored here
```

x86 Assembly Programming

- Move instructions, registers, and operands
- Memory addressing modes
- swap example: 32-bit vs. 64-bit
- Arithmetic operations
- Condition codes
- Conditional and unconditional branches
- Loops
- Switch statements

Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - %reg = Mem[address]
  - *Store* register data into memory
    - Mem[address] = %reg

Remember:
memory is indexed
just like an array[]!

- **Perform arithmetic function on register or memory data**
  - c = a + b;

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

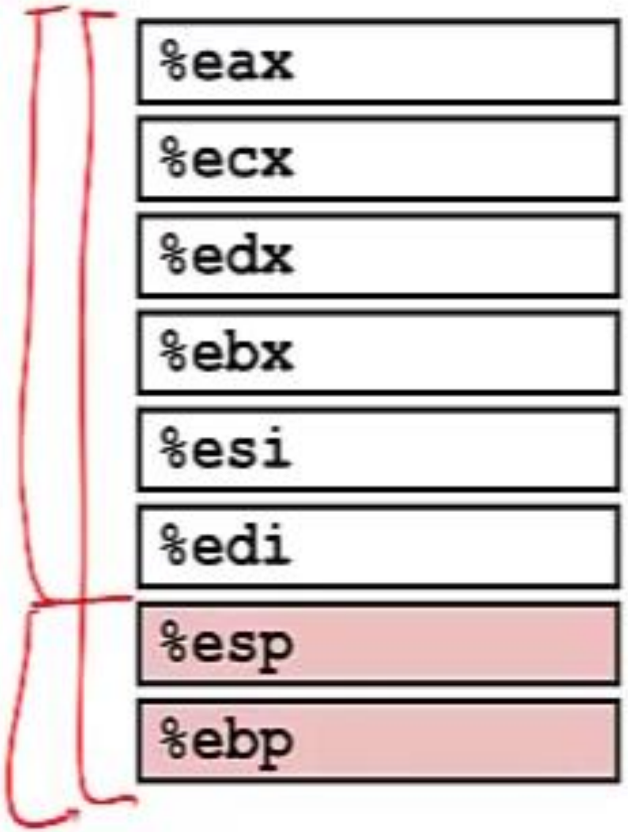
Moving Data: IA32

- **Moving Data**
  - `movx` ***Source, Dest***
  - `x` is one of {`b`, `w`, `l`}

  - `movl` ***Source, Dest*:**
    Move 4-byte "long word"
  - `movw` ***Source, Dest*:**
    Move 2-byte "word"
  - `movb` ***Source, Dest*:**
    Move 1-byte "byte"

General Registers

Specia



Moving Data: IA32

- **Moving Data**
  `movl` *Source*, *Dest*:
- **Operand Types**
  - *Immediate:* Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - *Register:* One of 8 integer registers
    - Example: `%eax`, `%edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  - *Memory:* 4 consecutive bytes of memory at address given by register
    - Simplest example: `(%eax)`
    - Various other "address modes"

`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`var_a = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p_a = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`var_d = var_a;`
		Mem	`movl %eax,(%edx)`	`*p_d = var_a;`
	Mem	*Reg*	`movl (%eax),%edx`	`var_d = *p_a;`

Cannot do memory-memory transfer with a single instruction.

Memory Addressing Modes: Basic

- **Indirect** **(R)** **Mem[Reg[R]]**
  - Register R specifies the memory address

```
movl (%ecx),%eax
```

- **Displacement** **D(R)** **Mem[Reg[R]+D]**
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address

```
movl 8(%ebp),%edx
```

%edx = Mem[%ebp + 8]

11:28 / 17:38

Using Basic Addressing Modes

```c
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl 12(%ebp),%ecx
   movl 8(%ebp),%edx
   movl (%ecx),%eax
   movl (%edx),%ebx
   movl %eax,(%edx)
   movl %ebx,(%ecx)

   movl -4(%ebp),%ebx
   movl %ebp,%esp
   popl %ebp
   ret
```

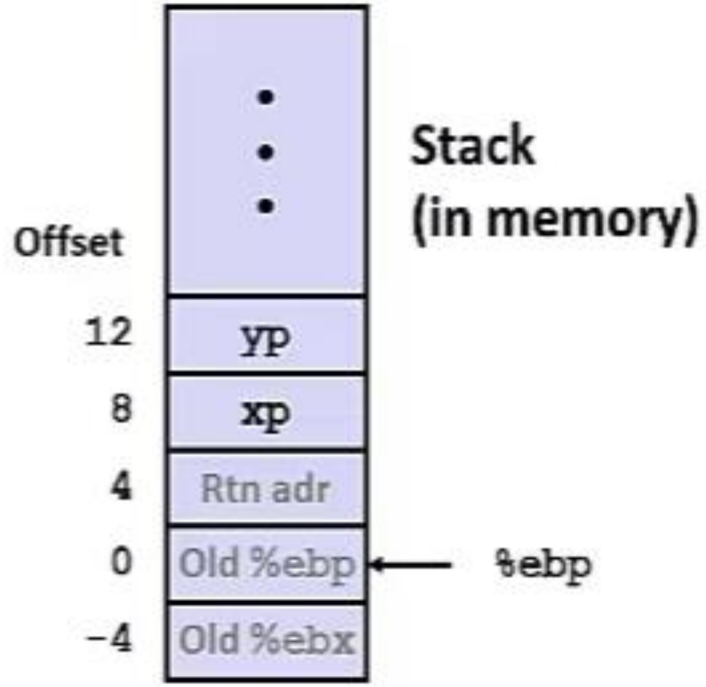
Set Up

Body

Finish

Understanding Swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

Register	Value
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

Mem[%ebp+12]

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

Download and install MASM32 software