CMPE 102 - Assembly Language Programming

Week 3 - Computer Basics

Processor components:

Central processing unit (CPU)

- A hardware component that performs computing functions utilizing the ALU, control unit and registers.

Arithmetic/logic unit (ALU)

- Performs mathematical calculations and makes logical comparisons

Control unit

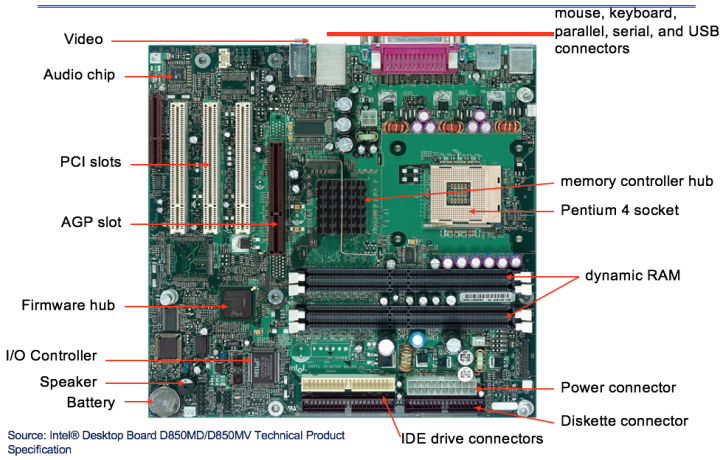
- Sequentially accesses program instructions, decodes them, coordinates flow of data in/out of ALU, registers, primary and secondary storage, and various output devices.

Registers

- High-speed storage areas used to temporarily hold small units of program instructions and data immediately before during, and after execution by the CPU.

Motherboard consists of:

- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips
- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)



Memory:

- ROM: read-only memory
- EPROM: erasable programmable read-only memory
- Dynamic RAM (DRAM): inexpensive; must be refreshed constantly
- Static RAM (SRAM): expensive; used for cache memory; no refresh required
- CMOS RAM: Complementary metal-oxide semiconductor. System setup information (battery backup)

Input-Output Ports:

- USB (universal serial bus)
    - intelligent high-speed connection to devices

        - up to 12 megabits/second
        - USB hub connects multiple devices
        - *enumeration*: computer queries devices
        - supports *hot* connections
- Parallel
    - short cable, high speed
    - once common for printers
    - bidirectional, parallel data transfer
    - Intel 8255 controller chip
- Serial
    - RS-232 serial port
    - one bit a time
    - uses long cables and modems
    - 16550 UART (universal asynchronous receiver transmitter)
    - programmable in assembly language
-

Week 4 - Memory

The ***memory*** is that part of the computer where programs and data are stored.
The basic unit of memory is the binary digit called a ***bit***. A bit may contain a 0 or a 1.
Memories consist of a number of ***cells***. (Each cell represents a binary value.

There are two types of main memory:
Random Access Memory (RAM)

- Holds its data as long as the computer is switched on.
- All data in RAM is lost when the computer is switched off (*volatile*).
- It is ***direct access*** as it can be both written to or read from in any order.
- Its purpose is to temporarily hold programs and data for processing. In modern computers it also holds the operating system.

Read only memory (ROM)

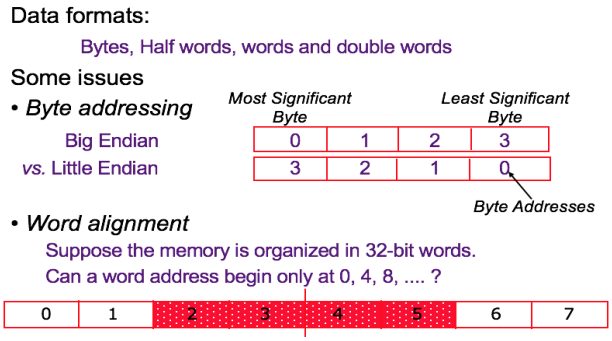
- ROM holds programs and data permanently even when the computer is switched off.
- Data can be read by the CPU in any order so ROM is also direct access.
- The contents of ROM are fixed at the time of manufacture
- Stores a program called the ***bootstrap loader*** that helps start up the computer

Main memory consists of a number of storage locations, each of which is identified by a ***unique address***. The ability of the CPU to identify each location is known as its ***addressability***.
Each location stores a **word**, i.e., the number of bits that can be processed by the CPU in a single operation. **Word length** may be typically 16, 32 or as many as 64 bits.

Memory locations are addressed in units of bytes. (One byte = eight bits).
Modern computers can read more than one byte at a time, typically 4 or 8 bytes in a *word*.

Computers express memory addresses as a binary number, the maximum number of bytes addressable is 2^m

Big Endian: Most significant
Little Endian: Least significant



There are no variables in assembly, you use the address. We only have memory. The correct notation would be.

a ← [a] + 1

A Struct, Array or other data structures exist in the theoretical world, however memory addressing is the real deal.

Registers may contain either addresses or data values.
Assembly programs typically spend as much time manipulating addresses as the do data values. That means of course, that the programmer must remain aware at all times whether the contents of a register represents an address or a data value. (ONLY THE PROGRAMMER KNOWS THE DIFFERENCE). To the processor, it's just a binary number either way.

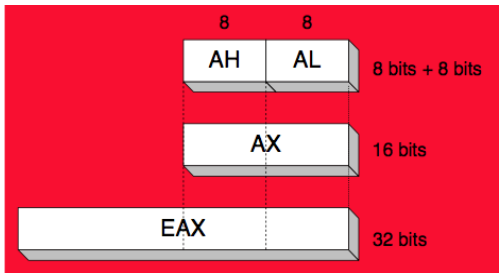
Week 5 - IA-32 Architecture

Special uses of registers:

- EAX - **Accumulator** register
  - Automatically used by multiplication and division instructions
  - Integer Function return values
- ECX - **Counter** register
  - Automatically used by LOOP instructions
  - Loop counter values
- ESP - **Stack Pointer** register
  - Used by PUSH and POP instructions, points to top of stack
- ESI and EDI - **Source Index** and **Destination Index** register
  - Used by string instructions
  - ESI: Source address for memory moves / compare instruction
  - EDI: Destination address for memory moves / compare instructions
- EBP - **Base pointer** register
  - Used to reference parameters and local variables on the stack

- Base frame pointer (used to access function parameters, local variables)

EAX, EBX, ECX, and EDX are 32-bit Extended registers



- Programmers can access their 16-bit and 8-bit parts
- Lower 16-bit of EAX is named AX
- AX is further divided into
  - AL = lower 8 bits
  - AH = upper 8 bits

ESI, EDI, EBP, ESP have only 16-bit names for lower half

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Special-Purpose & Segment Registers

EIP= Extended Instruction Pointer

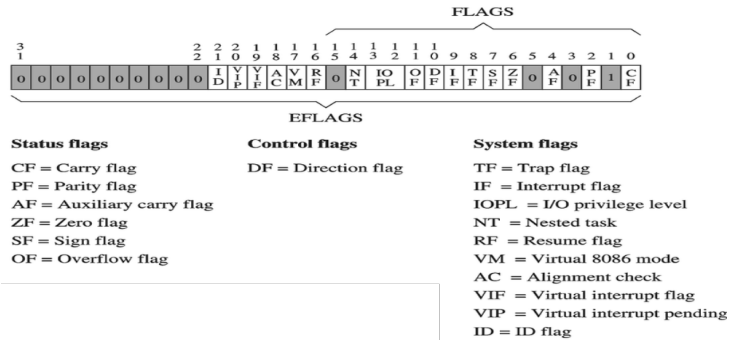
- Contains address of next instruction to be executed

EFLAGS = Extended Flags Register

- Contains status and control flags
- Each flag is a single binary bit

Six 16-bit Segment Registers

- Support segmented memory
- Six segments accessible at a time
- Segments contain distinct contents
  - Code, Data, Stack



EFLAGS register

Status Flags

- Status of arithmetic and logical operations

Control and System flags

- Control the CPU operation

Commonly-Used Flags

Some parts of EFLAGS (register that holds all flags):

- C: true if last math operation carried
- Z: true if last math operation gave a zero
- O: true if last math operation overflowed

- S: true if result of last operation was negative
- I: true if interrupts enabled

Flag commands:

- STC: set carry flag to true
- CLC: clear carry flag (set to false)
- Similarly: STI, CLI, etc.

Status Flags

Carry Flag

- Set when **unsigned** arithmetic result is out of range

Overflow Flag

- Set when **signed** arithmetic result is out of range

Sign Flag

- Copy of **sign bit**, set when result is **negative**

Zero Flag

- Set when result is **zero**

Auxiliary Carry Flag

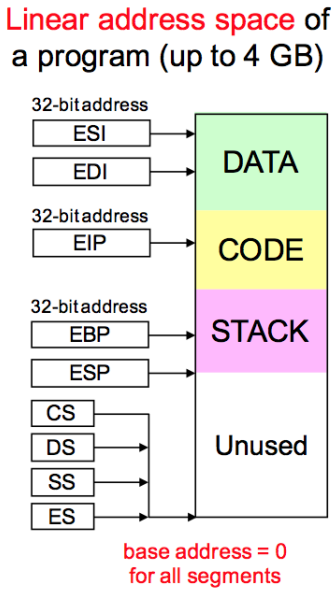
- Set when there is a **carry from bit 3 to bit 4**

Parity Flag

- Set when parity is **even**
- Least-significant **byte** in result contains **even number of 1s**

Flat Memory Model

Modern operating systems turn segmentation off. Each program uses **one 32-bit linear address space**. All segments are mapped to the **same linear address space**. A **linear address** is also called a **virtual address**. Operating system maps **virtual address** onto **physical addresses** - using a technique called paging.



Programmer View of Flat Memory

- Same base address for all segments
  - All segments are mapped to the **same linear address space**
- EIP Register
  - Points at next instruction
- ESI and EDI Registers Contain data addresses
  - Used also to index arrays
- ESP and EBP Registers
  - ESP points at top of stack
  - EBP is used to address parameters and variables on the stack

Protected Mode Architecture

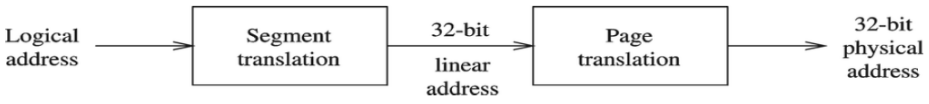
Logical address consists of

- 16-bit segment selector (CS, SS, DS, ES, FS, GS)
- 32-bit offset (EIP, ESP, EBP, ESI ,EDI, EAX, EBX, ECX, EDX)

Segment unit translates **logical address** to **linear address**

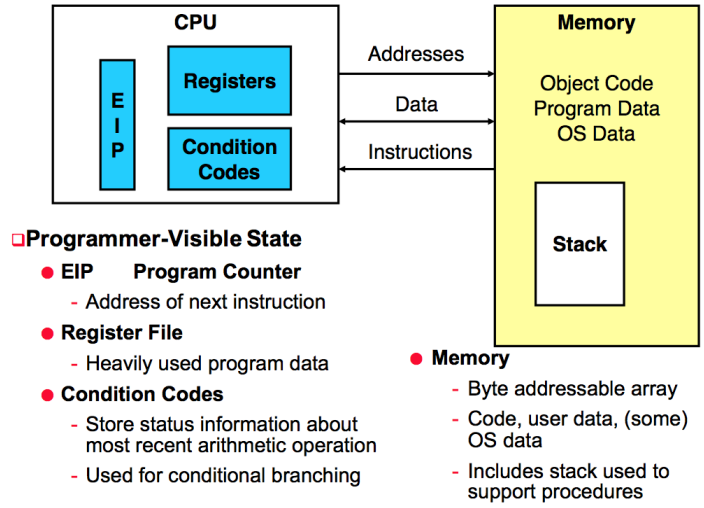
- Using a **segment descriptor table**
- Linear address is 32 bits (called also a **virtual address**)

Paging unit translates **linear address** to **physical address**



- Using a **page directory** and a **page table**

Assembly Programmer's View



Programmer-Visible State

- **EIP Program Counter**
  - Address of next instruction
- **Register File**
  - Heavily used program data
- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

Assembly characteristics

- Data structures does not exist on this architecture level. Only memory and adressing

Instruction format

- Three types of statements in assembly language Typically, one statement should appear on a line

- Executable Instructions
  - Generate machine code for the processor to execute at runtime
  - Instructions tell the processor what to do

- Assembler Directives
  - Provide information to the assembler while translating a program
  - Used to define data, select memory model, etc. Non-executable: directives are not part of instruction set

- Macros
  - Shorthand notation for a group of statements Sequence of instructions, directives, or other macros

Machine Instruction Example

```
int t = x + y;
```

C Code
- **Add two signed integers**

Assembly
- **Add 2 4-byte integers**
  - "Double" words in IA-32 speak
  - Same instruction whether signed or unsigned
- **Operands:**
  - x: Register eax
  - y: Memory M[ebp+8]
  - t: Register eax

```
add eax,[ebp+8]
```

Similar to expression
x += y

```
0x401046:   03 45 08
```

Object Code
- **3-byte instruction**
- **Stored at address 0x401046**

Instructions

Assembly language instructions have the format:

[*label:*] *mnemonic* [*operands*] [*;comment*]

Instruction Label (optional)
- Marks the address of an instruction, must have a colon :
- Used to transfer program execution to a labeled instruction

Mnemonic
- Identifies the operation (e.g. MOV, ADD, SUB, JMP, CALL)

Operands
- Specify the data required by the operation
- Executable instructions can have zero to three operands
- Operands can be registers, memory variables, or constants

General Instructions

- **MOV**: copies value from second argument to first argument: e.g., “mov eax, ebx” copies the value in EBX into EAX. (MOV should really be called “Copy”, since that’s what it does. Ah well.)
- **ADD**: adds its two arguments together and stores answer in the first one: “add ebx, ecx” means “let EBX = EBX + ECX”
- **SUB**: works just like ADD.
- **CMP**: like SUB, but doesn’t store the result anywhere. (we’ll see why it’s still useful later)
- **AND**: takes bitwise AND of both args, and stores answer in first arg. So, “and eax, edx” means “let EAX = EAX & EDX”. OR and XOR work the same.
- **MUL** and **DIV** are complicated, so we’ll ignore them for now.
- **PUSH**: push its argument onto the stack: e.g., “push eax”
- **POP**: pop stuff off the stack into the argument: e.g., “pop eax”.
- **CALL**: call a function: e.g., “*call 500*” calls the function at memory address 500.
- **RET**: return from function. Works like “*return*;”
- **JMP**: like ***goto***. “*jmp 100*” goes to 100.

Conditional jumps: only jumps if a condition is met. E.g., “jz 100” jumps only if last instruction produced a (z)ero result.
Use CMP and conditional jumps to do ***if***s (as we’ll see later.)