• We can study computer architectures by starting with the basic building blocks
  Transistors and logic gates

• To build more complex circuits
  Adders, decoders, multiplexors, flip-flops, registers, ...

• From which we can build computer components
  Memory, processor, I/O controllers, ...

• Which are used to build a computer system
  Laptop, Printer, PDA, cell phones, ...

This was the approach taken in your first course:
  03-60-265 — Digital Design
The Top-Down Approach

• We’ll study comp. arch. from the programmer’s view

• We study the actions that the processor needs to do to execute tasks written in high-level languages

• But to accomplish this, we need to:

  1. Learn the set of basic actions that the processor can perform: its instruction set

  2. Learn how a HLL compiler decomposes HLL commands into processor instructions

• We can learn the basic instruction set either

  1. At the machine language level

     But reading individual bits is tedious for humans

  2. At the assembly language level

     This is the symbolic equivalent of machine language (understandable by humans)

• Hence we’ll learn how to program a processor in assembly language to perform tasks that are normally written in HLL

     We’ll learn what is going on beneath the HLL interface
Levels and Languages

HLL code $\Rightarrow$ Compiler $\Rightarrow$ ASM code $\Rightarrow$ Assembler $\Rightarrow$ Machine code

- The compiler translates each HLL statement into one or more assembly language instructions

- The assembler translates each assembly language instruction into one machine language instruction

1. Each processor instruction can be written either in machine language form or assembly language form

2. Example, for the Intel Pentium:

   **Assembly language** $\leadsto$ Mov AL, 5;

   **Machine language** $\leadsto$ 10110000 00000101;

- Hence we’ll use assembly language
Assembly Language

- A program written directly in assembly language has the potential to be smaller to run faster than a HLL program.

- But it takes too long to write a large program in assembly language.

  Only time-critical procedures are written in assembly language (optimization for speed).

- Assembly languages are often used in embedded system programs stored in PROM chips.

  Computer cartridge games, micro controllers, ... 

- Remember: you’ll learn assembly language to learn how HLL code gets translated into machine language.

  That is, to learn the details hidden in HLL code.
The Platform we’ll Use

• Assembly language and machine language are processor specific.

  We’ll write code for Intel’s $x_{86}$, $x \geq 3$

• The assembler places its machine code into an object file which is OS specific.

  1. Our code will run (only) on Windows.

     And it will crash on DOS.

  2. Our programs will be Win32 console applications.

     These are programs for which all I/O operations are character-based.

     They run into an MS-DOS box but they are not DOS programs (they do not use DOS calls).
The Intel X86 Family

- The instruction set of the x86 is backward compatible with any of its predecessors

  New additional instructions are introduced with each new processor
Registers

- Registers are the fastest memories
  1. Located directly on the processor
  2. Manipulated directly by processor instructions

- The registers for the 8086 and 80286 are only 16-bit wide

- Most of these registers have been extended to 32 bits for the 80386 and higher processors
  But very few extra registers have been added

- The Pentium has very few registers
  Only 8 registers are available to the programmer (apart from the segment and the FPU registers)
General Purpose Registers

- Used by the programmer for arithmetic and data transfer

- **AX** is the least significant part of **EAX** and can be accessed independently (by its name)

- **AH** and **AL** can also accessed independently

- This is also true for **EBX**, **ECX** and **EDX**

- Only the 16-bit part are present in the 8086 and 80286
Index Registers

- The least significant half can be accessed independently (since it has a name)

- Only the lower 16 bit were present in 8086 and 80286

- Used often to carry the offset part of the logical address (more on that later)

1. ESI and EDI for the data segment

2. ESP and EBP for the stack segment
The Instruction Pointer: EIP

- EIP always contains the offset address of the instruction to be executed next

1. This is the program counter for the x86

2. The offset address is 32-bit wide when the processor runs in 32-bit mode (i.e. for 32-bit segments)

3. It is 16-bit wide in 16-bit mode

- Only the lower 16-bit was present in the 8086 and 80286 (called IP)
EFLAGS Register and Condition Codes

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS</td>
<td>FLAGS</td>
<td></td>
</tr>
</tbody>
</table>

- The conditions codes of the processor are stored in the EFLAGS register

- They consist of individual bits indicating either:

1. The mode of operation of the CPU. Ex:
   - **DF**: indicates if arrays are processed in the direction of increasing addresses

2. The outcome of an arithmetic operation. Ex:
   - **ZF**: indicates if the result is zero
   - **SF**: indicates if the result is negative
   - **CF**: indicates if there is an unsigned overflow
   - **OF**: indicates if there is a signed overflow
Each program is subdivided into logical parts called *SEGMENTS*

1. Code segment: CS
2. Stack segment: SS
3. Data segment: DS, ES, FS and GS

- Segment registers hold the base address of these program segments
- Segment registers are 16-bit wide
Logical and Physical Addresses

• Addresses specify the location of instructions and data

• Addresses that specify an *absolute* location in main memory are *physical addresses*

  They appear on the address bus

• Addresses that specify a location *relative* to a point in the program are *logical (or virtual) addresses*

  They are addresses used in the code and are independent of the structure of main memory

• Each logical address for the x86 consists of two parts:

  1. A segment number used to specify a (logical) part of the program

  2. An offset number used to specify a location relative to the beginning of the segment
Address Translation and Running Modes

- The *translation* from logical to physical addresses is done at run time

- The way in which this address translation is done depends on the *running mode* of the x86

- Two different running modes exist for the x86
  1. *Real mode* (supported by every x86)
  2. *Protected mode* (all x86 except the 8086)

**Address Translation in Real Mode**

- The 16-bit segment number (contained in a segment register) is first multiplied by 16 to give the 20-bit physical address of the first byte of the referenced segment

- Then we add the 16-bit offset address to obtain the 20-bit physical address of the referenced data (or instruction)

1. Ex: if CS contains 15A6h (in hexadecimal), and IP contains 0012h

2. The physical address of the instruction to be executed next is just 15A60h + 0012h = 15A72h
Characteristics of Real Mode

- Can address only up to 1MB of physical memory

- Does not support multitasking

  Only one process is active at a time

- No protection is provided: a program can write anywhere (and corrupt the operating system)

- The 8086 runs only in this mode

- DOS is a real-mode operating system

- Our programs will not run in this archaic mode

  They will run in protected mode, which does not suffer from any of these limitations
Address Translation in Protected Mode

- The virtual address of a referenced word is given by a pair of numbers (segment, offset)

- The segment number is contained in a segment register and is used to select (or index) an entry in a segment table (called a descriptor table)

  Hence a segment register is also called a selector

- The selected entry (the descriptor) contains the base address and the length of the referenced segment

- The 32-bit base address is added to the 32-bit offset to form a 32-bit linear address \((P_1, P_2, D)\)

  \(P_1\) indexes a directory page table (in memory) to obtain the base address of a second page table which is indexed by \(P_2\) to give the physical address of the referenced word
Intel 386 Address Translation

[Diagram showing the process of address translation involving logical address, offset, selector, descriptor table, segment descriptor, linear address, page directory, directory entry, page table, physical address, and page frame.]
The FLAT Memory Model

- The segmentation part is hidden to the programmer when the base address of each segment descriptor is the same

1. Each selector then points to the same segment; so that code, data and stack share the same segment

2. Protection bits (read-only, read-write) in each descriptor can still be used

3. Done by WINDOWS, Linux, FreeBSD, ...

- The offset part of the virtual address is then equivalent to the linear address \((P_1, P_2, D)\)

1. Only the offset part of the virtual address is used to specify the location of a referenced word

2. The address space is then said to be FLAT

3. All our programs will use the FLAT memory model
Memory Units for the x86

- The smallest addressable unit is the BYTE
  
  1 byte = 8 bits

- For the x86, the following units are used
  
  1. 1 word = 2 bytes = 16 bits
  2. 1 double-word = 2 words = 4 bytes
  3. 1 quad-word = 2 double-words = 4 words

Data Representation

- To obtain the value contained in a block of memory we need to choose an interpretation

- Ex: memory content 0100 0001 can either represent:
  
  1. The number $2^6 + 1 = 65$
  2. Or the ASCII code of character "A"

- Only the programmer can provide an interpretation
• Each character is represented by a 7-bit code called the ASCII code

• ASCII codes run from 00h to 7Fh (h = hexadecimal)

  Only codes from 20h to 7Eh represent printable characters. The rest are control codes (used for printing, . . . )

• An extended character set is obtained by setting the most significant bit (MSB) to 1 (codes 80h to FFh) so that each character is stored in 1 byte

  This part of the code depends on the OS used

  For Windows: we find accentuated characters, Greek symbols and some graphic characters

Text Files

• These files contain only printable ASCII characters (for the text) and the non-printable ASCII characters to mark each end of line and the end of file

• But different conventions are used for indicating an end of line

  1. Windows: <CR>+<LF>

  2. UNIX:     <LF>

  3. MAC:      <CR>

• This is at the origin of many problems encountered during transfers of text files from one system to another
Number Systems

• A written number is meaningful only with respect to a \textit{base}

• To tell the assembler which base we use:
  1. Hexadecimal 25 is written as $25h$
  2. Octal 25 is written as $25o$ or $25q$
  3. Binary 1010 is written as $1010b$
  4. Decimal 1010 is written as 1010 or 1010d

• You already know how to convert from one base to another (if not, review 03-60-265 notes)

\textbf{Integer Representations}

• Two different representations exist for integers

  1. Signed representation: MSB is the sign

    \textbf{MSB} = 0 \Rightarrow \text{positive or zero number}

    \textbf{MSB} = 1 \Rightarrow \text{negative number}

  2. Unsigned representation: all the bits are used to represent a \textit{magnitude}

    It is thus always a positive number or zero
Two’s Complement Notation

• Used to represent negative numbers in the signed representation

• The two’s complement of a number $X$, denoted by $\text{NEG}(X)$, is obtained by complementing all its bits and adding +1

• Hence by definition: $\text{NEG}(X) = \text{NOT}(X) + 1$

Ex:

$\text{NEG}(10) = \text{NOT}(10) + 1$

$= \text{NOT}(0000\ 1010_2) + 1$

$= 1111\ 0101_2 + 1 = 1111\ 0110_2$

This is how $-10$ is represented (on 1 byte)

• We always have $X + \text{NEG}(X) = 0$

1. i.e. $\text{NEG}(X)$ is the additive inverse of $X$

2. Hence we have $\text{NEG}(X) = -X$

• To perform the difference $X - Y$:

The machine executes the addition $X + \text{NEG}(Y)$
Two’s Complement Notation
(Continued)

• Note that we have

\[ \text{NEG}(10) = 1111 \ 0110b \]

when we use 1 byte of storage

• But

\[ \text{NEG}(10) = 1111 \ 1111 \ 1111 \ 0110b \]

when we use 1 word of storage

• Exercise #1: Compute the two’s complement of the following numbers and mention if there is an overflow (i.e. when the given storage is not large enough to hold the result). Write the result in binary

1. 16 on 1 byte
2. −16 on 1 byte
3. −128 on 1 byte
4. −128 on 1 word
5. 0 on 1 word
6. Try many other examples for yourself
The MSB of a signed integer is used for its sign. Fewer bits are left for its magnitude. Ex: For a signed byte

1. Smallest negative = $-128 = 1000\ 0000b$
2. Largest negative = $-1 = 1111\ 1111b$
3. Smallest positive = $+0 = 0000\ 0000b$
4. Largest positive = $+127 = 0111\ 1111b$

Exercise #2: Give the smallest and largest positive and negative values for

1. A signed word
2. A signed double-word
3. A signed quad-word

**Signed and Unsigned Interpretations**

To obtain the value of an integer in memory we need to choose an interpretation.

Ex: A byte of memory containing $1111\ 1111b$ can represent either one of these numbers

1. $-1$ if a signed interpretation is used
2. $255$ if an unsigned interpretation is used

Only the programmer can provide an interpretation of the content of memory.