Objectives

- To provide an introduction to the internal architecture of an INTEL 80x86-based microprocessor used in the IBM & compatible PCs.

• Supplementary Readings:
  - Irvine Chapter 2: Processor Architecture
  - Linux Assembly Language Programming: Chapter 1.1
  - PC in a Nutshell: Chapter 1.1, 1.2, 4.1 and 4.2
  (Note: you do not need to cover every aspect of these chapters. They provide you additional background materials on the topic.)

Hardware dependencies

- Software written for a machine with an Intel Processor would not run on a machine with a Motorola processor
  - Why?
  - The machine instructions are different. The bit string which means one thing on one processor means something else on another processor.
  - Eg, Software written for an i80386 will run on an i80586 (Pentium) but the reverse may not be true. The i80586 processor has a much larger instruction set and it includes the instruction set for the i80386
Operating System dependencies

- Programs written (i.e., the machine code versions) for Windows or DOS will not run under another operating system like Unix or Linux, although the hardware may be identical.

Compiler dependency

- Different compiler developers often introduce their own extensions to the language resulting in different variants of the language.
- A program written in Borland C using the Borland C extensions will not compile under Microsoft C or the ANSI C compiler.
- These dependencies reduce the portability of software.
- If the compiler dependencies were not there, then the other 2 dependencies can be overcome by recompiling the source.

Major Hardware Components

- System Components
- Generic CPU Components
- 80x86 specific CPU Components (IMPORTANT)
- The CPU Fetch/Execute Cycle (VERY IMPORTANT)

Generic Computer System Components

- Processor (CPU)
- Main memory
- Peripherals (I/O devices)
- System Bus
CPU - Central Processing Unit

- The “Brain” of a computer
- Performs the actual execution
- Executes and understands only machine language
- Controls access and interaction of all other system components
- The CPU consists of:
  - Control Unit – for decoding and executing program instructions
  - ALU – for executing arithmetic and Boolean operations
  - Registers and Accumulators – internal memory units for storing intermediate results

Registers

- Registers are high-speed storage cells directly inside the CPU, designed to be accessed at much higher speed than memory cells
- PC: program counter – contains the address of the instruction that is currently executing

Registers

- SP: stack pointer – contains the address of the top item in a memory region called the stack. Computer uses stack to keep track of function calls and other info (see later for more details)
- XR: index register – used by programs to access arrays
- PS: processor status register – contains information such as condition codes (flags), eg, indicating whether the most recent operation produced +ve, -ve, or zero result
- DRs: data registers – used as fast temporary memory to store data/results
- MAR: memory address register – used as the CPU’s connection to the address bus. Eg, if an instruction needs contents of the memory word 40050, the CPU will put 40050 into MAR, from where it will flow to the address bus
- IR: instruction register – contains the instruction currently being executed
• **MDR:** memory data register - used as the CPU’s connection to the data bus. Eg,
  - if an instruction needs (to read) contents of the memory word 40050, the memory will put \( c(40050) \) onto data bus, from where it will flow into the MDR in the CPU.
  - If an instruction needs to write data, eg the value 35, to the memory word 40050, the CPU will put 35 in the MDR, from where it will flow out onto the data bus and then on to memory.

• **Sizes of the various registers**
  - PC, SP, XR, and MAR all contain addresses – so their sizes are the address size of the machine
  - DRs and MDRs typically have sizes equal to the word size of the machine
  - PS stores miscellaneous information (eg, flags) – its size has no particular relation to the machine’s address size or word size

• **Memory (primary/main)**
  - Contains both the programs to execute and their data
  - Consists of cells (bytes & words) containing the data identified by an address
  - CPU *fetches/gets/reads* from the contents of cells of memory
  - CPU *stores/puts/writes* to the contents of cells of memory
Memory

- Typically two types of memory – RAM and ROM
  - RAM - random access memory (most is volatile, some non-volatile battery backed up - termed CMOS RAM)
  - Contains operating system, user programs and data
- ROM - read only memory (non-volatile)
  - Contains boot program, system diagnostics, low level driver routines (BIOS), system parameters, information/data/code fixed for the lifetime of computer or rarely updated

Input/Output Devices (I/O)

- Typical devices: keyboard, screen, mouse, disc drives, speakers, microphones etc.
- Specialised devices: eg. sensors, actuators
- Nature of original data could be binary, discrete or continuous.
- If continuous signals are required, converters are used to translate between the analogue and digital signals.

System Bus (sets of parallel wires/lines used for communication)

- For communication of CPU with memory & I/O
- At any fixed time, each wire has only a 0 or a 1 (one bit of information) - either high or low voltage
- The size of the bus indicates the no. of parallel wires used, eg, 20-bit address bus will use 20 wires

Three components of System Bus

- Data Bus – used for transferring information. Its size determines the base unit of transfer (16 bit, 32 bit, etc)
  - CPU can put/drive data bus with data to be put into memory or sent to output device
  - CPU can get/read data bus with data put onto there by memory or by an input device
• Address Bus - to indicate which memory cell
  – CPU nearly always puts/drives the address bus with
    the address of memory cell to reference

• Control Bus – consists of assorted signals. For example:
  – MEMR - memory read
  – MEMW - memory write
  – IOR - input/output read
  – IOW - input/output write

• Since the system bus mentioned above is
  directly associated with the CPU, it is often
called the local bus these days.

• Modern computer systems have several
  system buses (called external buses),
  which provide a standard mechanism to
  interface I/O devices. Their functions are:
  – Data transfer
  – Timing and synchronisation of data

Examples of external buses

• ISA (Industry Standard Architecture) bus
  – was originally developed for the IBM-PC computer
    (8086, 8-bit)
  – provides a simple bus and connector system for
    interfacing I/O devices to the computer system
  – was extended to support 16 bits (80286)
  – operates at a slow speed (8MHz)

• EISA (Extended Industry Standard Architecture)
  – is a 32 bit bus extension to ISA
  – was designed for the 80386 and 80486 processors
• PCI (Peripherals Component Interconnect (PCI) bus
  – is the more recent bus found in newer machines
  – supports 32-bit address and 64-bit data transfers
  – operates at 33 MHz
  – also supports the new plug and play facility.

• USB (Universal Serial Bus)
  – USB 2.0 interface technology enables the computer to be connected to multiple devices such as hubs, printers, mice, joysticks, keyboards, etc. This technology is also referred to as Hi-Speed USB.
  – USB 2.0 transfers data between the computer and peripherals 40 times faster than original USB at transfer rates up to 480Mbps (megabits per second) whereas USB 1.1 devices transfer at speeds of 12Mbps.

What is plug and play (PnP)?

• The concept received considerable attention with the advent of Windows 95 operating system, although it has already been in use before (eg, in Apple’s Macintosh computers).
• Each I/O device which is plugged into the PCI bus has information about itself on its own board.
• The computer can query the I/O device to determine its type, function etc and then appropriately configure and use the I/O device without human intervention
• For it to happen properly, the BIOS, the device to be added and the operating system need to be designed/extended to handle PnP
Direct Memory Access

- **DMA**: It is also possible for the I/O devices to communicate directly with memory (i.e. without CPU involvement), called **direct memory access (DMA)**
- This is done via special electronic circuitry and involves a **DMA controller**.

Reasons for using the DMA controller

- Information transfer rate is superior to that achievable by using CPU as an intermediary
- It frees up the CPU to do other things while the DMA controller is handling data transfer
- Example: transfer of information to/from hard disk always involves DMA because of the large amounts of information transferred

80x86 CPU Components

- MAR, MDR
- Segment Registers: Code (CS), Data (DS), Stack (SS) and Extra (ES)
- Pointers: Instruction (IP), Stack (SP), Base (BP)
- Index Registers: Source (SI), Destination (DI)
- Data Registers: AX, BX, CX, DX
- Instruction Register: (IR)
- Arithmetic Logic Unit: (ALU)
The above components form a subset of the more sophisticated 286, 386, ..., 686 subfamilies.

The additional components of newer machines are used only when a 286/386/486 CPU runs in **protected mode**.

When such a CPU runs in **real mode**, it is essentially equivalent to a fast version of an 80x86 CPU.

**Non-segment Registers**

- The MAR and MDR are part of a more complex entity called the **bus interface unit**.
- The registers BP, BX, SI, DI all serve as index registers similar to XR but with much more variety then XR.

**General Registers**

- AX and DX are mainly used as Data Registers similar to the DRs.
- AX, BX, CX and DX are further broken down into
  - **high bites** - bits 15 through 8,
  - **low bites** - bits 7 through 0.
- The high byte of AX is called AH, the lower byte is called AL.
- Similarly for BX, CX and DX registers.

**8086 Segmented Memory**

- Intel 8086 CPU allows memory addresses to consist of 20 bits
- this will require the address bus to have 20 lines (wires), one for the transmission of each bit in the address
- 20-bit address bus has the ability to access \( 2^{20} = 1,048,576 \) (1MB) different memory locations, numbered from:
  - 00000 to FFFFF in Hexadecimal
In the 8086, memory is actually thought of as consisting of two parts:
- A 16-bit segment address, and
- a 16-bit offset from the start of that segment.

Each segment (chunk of memory) consists of $2^{16}$ locations (= 64KB of memory), giving the largest possible offset: FFFF₁₆, which is 65535₁₀.

The 8086 keeps track of four different segments: **code segment**, **data segment**, **stack segment**, and **extra segment**.

The 8086 has four 16-bit segment registers (one for each segment): CS, DS, SS, ES. These point to code, data, stack, and extra segments respectively.

The address of an instruction is formed by adding values from the CS and IP registers:
- CS contains the base (start) address of code segment (program instructions)
- IP contains the displacement (offset), i.e., how far the current instruction is from the base.

The <segment:offset> pair are often written as xxxx:yyyy, where xxxx is the value in the segment register and yyyy is the offset.

The DS register defines the beginning of the data segment (where program’s data items are stored).

**NOTE**: Segments must begin at an address that is divisible by 16, i.e., one whose least significant digit in hex representation is 0.

When storing a segment’s address in the segment register, this rightmost 0 is kept implicit and not physically stored. This 0 is appended at the time the address is calculated. This technique allows a 20-bit address to be stored in a 16-bit register. The bottom 4 bits of the 20-bit address are always assumed to be 0.
• Note that appending a 0 in hex means multiplying by 16, just as appending a 0 in base 10 arithmetic means multiplying by 10.

• Thus the address is always computed as:
  16 * c(segment register) + offset value
  i.e., 16 * xxxx + yyyy

Example

Value in the DS register = 75B4\text{_{16}}
Offset = 124\text{_{16}}
Actual address of data = 75B40+0124 = 75C64\text{_{16}}
Address calculation is done as follows:
Shift the value of segment register 4 bits to the left (equivalent to one hex digit), then add in the offset
The resulting 20-bit value is the actual address as shown below:

\[
\begin{align*}
0111 & \quad 0101 & \quad 1011 & \quad 0100 & \quad 0000 & = 75B40 \\
0000 & \quad 0001 & \quad 0010 & \quad 0100 & \quad \quad = 00124 \\
0111 & \quad 0101 & \quad 1100 & \quad 0110 & \quad 0100 & = 75C64
\end{align*}
\]

The CPU Fetch/Execute Cycle (FEC)

• Once switched on, all computers enter the FEC
• For a computer which is already on, it is always in the FEC mode of operation
• Irrespective of whether your program is working correctly, it is in the FEC
• Computer after a crash is usually still in the FEC mode - just not doing what you expect (eg. no output, "random" output, ...)
• Exception: A few very special instructions will actually completely stop or halt the computer

Algorithm followed by the CPU while executing a program

initialise PC with the address of program’s first instruction;
do
fetch instruction from the location given by PC:
  place it in a holding area;
  increment the PC;
decode instruction to determine the operation:
  perform address translation;
  fetch operands from memory;
execute the instruction:
  carry out the required operation;
  store results in memory or registers;
  set status flags attached to the CPU;
while (not end-of-program)
Basic steps in the FEC are as follows

• **Generic steps applicable to most CPUs:**
  1. Fetch Instruction (memory read operation)
     - PC -> MAR -> Address Bus
     - MEMR control line asserted
     - CPU waits for memory to send contents of requested address
     - PC updated to point to next instruction (ready for next FEC)
     - Memory puts requested stuff (code in this case) on the Data Bus
     - Data -> MDR

2. Instruction Load and Decode (data interpreted as machine instruction)
   - CPU copies contents of MDR into IR
   - MDR now free for further memory accesses
   - CPU inspects data, interprets it as an instruction and decodes it

3. Instruction Execution
   - Actual execution of instruction
   - Instruction may need data from memory (memory read operation)
   - Instruction may store data back to memory (memory write operation)
   - After last FEC step, another FEC is started
   - All steps in FEC are performed by hardware

80x86 specifics: modifications to FEC for *real mode*

• PC -> MAR replaced with 16*c(CS) + c(IP)
• In step 1, only first bytes of instruction fetched (80x86 instructions can be from 1 to 6 bytes in length)
• In step 2, CPU determines from bit pattern whether instruction consists of more bytes
• In step 2, if needed, CPU fetches more bytes until entire instruction is in IR
• IP incremented by total number of bytes in instruction
Software Components of the Computer

- Hardware alone is not enough for a computer system
- Must program the hardware to do something useful
- Many programs have core components which are exactly the same
- ==> Most typical example is I/O

Consider writing your programs without an OS

- Turn on the computer, nothing happens
- Can't use desired computer to enter program (or at least very difficult)
- Can't use desired computer to compile program
- Suppose that somehow, you managed to load a program from disk into memory and set the IP of the CPU with the address of your program. The program would still be unable to run because requests by the program to do I/O would not succeed unless you wrote all the low level output routines within your program.

- Rather than each programmer/user writing their own I/O routines (among other things)
  - ==> Common core of routines are collected together
- Operating System (OS) has a collection of common core routines which provide an interface to the hardware thus making it usable.

- Important to remember: Operating system is a program and/or collection of programs. However, in most cases it is very large and very complex
  - ==> Provides a range of services to the user
  - As a naive user it is sometimes difficult to know what is done by hardware and what is done by software (e.g. character I/O, editing, etc.)
User programs make use of many OS services

• Executable version of a user program has machine instructions for the host computer PLUS calls to the OS (i.e. programs/routines in the OS)
• Therefore, executable programs for same machines don’t usually run on the same machines having different operating systems (or even different versions) (e.g. PC - MSDOS & Windows - A Windows program will not run in MSDOS)

OS itself consists of machine instructions for specific CPU & its hardware (memory, disks, screen output, etc)
• Different OS’s can be run on same hardware, eg. IBM-PC: MS-DOS, Windows 95/98 & Windows NT, OS/2, Linux
• Same OS (providing the same set of services and similar user interface) can be developed for different hardware systems. Executable versions will be different - different machine instructions, addresses, etc. Examples include Unix, Windows-NT, OS/2

Speed of a Computer "Engine"

• Speed of a computer is affected by many factors
• Speed of computer and/or program?
1. CPU Architecture:
   – Instructions - simple or complex: simple instructions take less time to run
   – registers - few or lots
   – memory structure (data and address bus sizes)

2. Parallel Operations (more computing per unit time)
   – instruction prefetch
   – During execution of current instruction, CPU will attempt to fetch one or more sequential instructions
   – Depends on bus activity
   – pipelining (more general form)
   – more CPUs (parallel computers)
3. **Clock Rate**
   - Each instruction is implemented as microsteps
   - Microsteps are performed one per clock cycle
   - Different instructions require different number of microsteps (i.e., clock cycles). Eg
     - Add two registers (e.g., 2 cycles)
     - Multiply (e.g., 21 cycles)
     - Fetching data from memory takes longer than from register or cache
   - Typically clock rates: Megahertz (e.g., 400 MHz - 400 million times a second)

4. **Memory Access times**
   - CPU asserts address and control lines
   - Memory may not be as fast as CPU, so need to wait for memory to respond
   - Wait states inserted into FEC while waiting: (poor CPU & memory match)
   - Memory cache - temporary copy within CPU of a small subset of memory

5. **OS Efficiency**
   - Same program may run at different speeds on same computer with different OSs
     - disk caching
     - memory management (virtual memory)
     - OS implementation

   • **RISC architectures** (Reduced Instruction Set) - often has high clock rates with fast simple instructions - but many of these “reduced instructions” are needed as compared with a typical instruction on a CISC machine (Complex Instruction Set)
Memory and Windows Operating System

- Under the DOS operating system - CPU runs in **real mode**
- Under the Windows operating system - CPU runs in **protected mode**
- In protected mode, the 80x86 processor allows **multi-tasking**

Multi-tasking environment

- The code (instructions) and data for two or more programs (tasks) can reside in main memory at the same time
- Each program (task) is executed in turn for a small period of time thus giving the impression that several tasks are being executed simultaneously

Protected mode of operation vs Real mode

- In protected mode, there are
  - More instructions
  - More registers
  - Different functionality of some registers
  - Memory structure & access can be different (e.g. certain memory regions can be protected from a user program)

Protected Mode

- At this stage of unit, we are concerned here only with changed functionality of CS & DS, and memory structure
- Contents of segment registers are actually a selector (index) to a descriptor table which in turn contains actual memory location
- Segment registers no longer “point” to actual memory
- More than one program can exist in memory at any one time including the operating system
Each task/program is assigned its own regions of code & data - setup and managed by Windows.

Each task/program can access only that memory which has been “assigned” to that program.

Accessing memory not assigned to a program results in a General Protection Fault.

Another look at the 80x86 CPU

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>8-bit name</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>AH AL</td>
<td>AX EAX</td>
<td></td>
</tr>
<tr>
<td>Base index</td>
<td>BH BL</td>
<td>BX EBX</td>
<td></td>
</tr>
<tr>
<td>Counter</td>
<td>CH CL</td>
<td>CX ECX</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>DH DL</td>
<td>DX EDX</td>
<td></td>
</tr>
<tr>
<td>Stack pointer</td>
<td>SP</td>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>Base pointer</td>
<td>BP</td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>Source index</td>
<td>SI</td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>Destination index</td>
<td>DI</td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>Instruction pointer</td>
<td>IP</td>
<td>EIP</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>FL</td>
<td>EFL</td>
<td></td>
</tr>
</tbody>
</table>

Segments

- All segment registers are 16 bits:
  - Code segment: CS
  - Data segment: DS
  - Stack segment: SS
  - Extra segment: ES
  - F Segment: FS
  - G Segment: GS

The above 80x86 registers can generally be categorised as follows:

- General-purpose Registers
  - These are: AX, BX, CX, DX
  - Also called data registers – used for arithmetic and data movement
  - AX: the **accumulator register** – is favoured by the CPU for arithmetic operations
  - BX: the **base register** – can hold the address of a function or variable. Three other registers with this ability are SI, DI, and BP. BX can also be used for arithmetic and data movement.
• CX: the **counter register** – acts as a counter for loop instructions. These instructions automatically repeat and decrement CX.
• DX: the **data register** – has a special role in multiply and divide operations. Eg, when multiplying, DX holds the high 16 bits of the product.

### Segment Registers

• CPU contains four **segment registers** which are used as base addresses for program instructions, data, and the stack
• All memory references on the IBM-PC involve a segment register used as a base address
• CS: the **code segment register** – holds the base address of all executable instructions in a program

### Index Registers

• DS: the **data segment register** – is the default base address for variables. The CPU calculates their addresses using the segment value in DS
• SS: the **stack segment register** – contains the base address of the stack
• ES: the **extra segment register** – is an additional base address for variables
• BP: the **base pointer** register – contains an offset from the SS register (as does the SP). The BP register is often used by a function/procedure to locate variables that were passed on the stack by a calling func
• **SP**: the *stack pointer* register – contains the offset of the top of the stack. The SP and SS registers combine to form the complete address of the top of the stack.

• **SI**: the *source index* register – takes its name from the string movement instructions, in which the source string is pointed to by the SI register.

• **DI**: *destination index* register – acts as the destination for string movement instructions.

### Status and control

• **IP**: *instruction pointer* register – always contains the offset of the next instruction to be executed within the current code segment. The IP and CS (code segment) registers combine to form the complete address of the next instruction.

### Two types of flags

• **Status flags**:
  - O = overflow
  - S = Sign
  - Z = Zero
  - A = Auxiliary Carry
  - P = Parity
  - C = Carry

• **Control Flags**:
  - D = direction
  - I = Interrupt
  - T = Trap
• **Note:** You don’t have to memorise each flag position!! Fortunately there are special instructions designed to test and manipulate the flags. A flag (or bit) is set when it equals 1; it is clear (or reset) when it equals 0. The CPU sets flags by turning on individual bits in the Flags register.