Table of Contents

Lab (1): Computer Anatomy ....................................................................................................... 4

Objectives .................................................................................................................................... 4

What is a Computer? .................................................................................................................. 4

System ........................................................................................................................................ 4

1. Motherboard .......................................................................................................................... 5
2. Central Processing Unit (CPU) ............................................................................................... 6
3. Memory .................................................................................................................................. 6
4. Video Adapter .......................................................................................................................... 7
5. Operating System .................................................................................................................... 8
6. Connectivity .............................................................................................................................. 8

Chassis ....................................................................................................................................... 9

1. Full Tower ............................................................................................................................... 9
2. Mid Tower ............................................................................................................................... 9
3. Mini Tower .............................................................................................................................. 9
4. Slim Line Case ........................................................................................................................ 9
5. Small Form Factor (SFF) Case ............................................................................................... 9

Cooling System .......................................................................................................................... 10

Power Supply Unit ..................................................................................................................... 10

Peripherals ................................................................................................................................ 11

1. Input Devices .......................................................................................................................... 11
2. Output Devices ........................................................................................................................ 12
3. Input & Output Devices .......................................................................................................... 13

Review Questions ..................................................................................................................... 14

Lab (2): PC I/O Ports .................................................................................................................. 16

Objectives .................................................................................................................................. 16

Mini-DIN .................................................................................................................................... 16

PS/2 ............................................................................................................................................ 16

Separate Video (S-Video) ............................................................................................................ 16

Universal Serial Bus (USB) ........................................................................................................ 17

Firewire (IEEE 1394) .................................................................................................................. 18

DB (D-subminiature, D-shell) Connectors .................................................................................. 19

Registered Jack (RJ) ................................................................................................................... 19

Audio ....................................................................................................................................... 20
1. Analog .......................................................................................................................... 20
2. Digital .......................................................................................................................... 20
Digital Visual Interface (DVI) ........................................................................................ 20
HDMI (High-Definition Multimedia Interface) ............................................................... 21
DisplayPort ...................................................................................................................... 21
eSata .................................................................................................................................. 22
Summary of Some Bus Speeds ..................................................................................... 22
Summary of External Connector .................................................................................. 23
Review Questions ........................................................................................................... 24
Lab (3): Motherboard .................................................................................................. 27
Objectives ...................................................................................................................... 27
Motherboard Form Factor ............................................................................................. 27
Processors Sockets ........................................................................................................ 28
Chipset ............................................................................................................................. 31
Figure 3-3. Intel Z97 Chipset Block diagram ................................................................... 31
Buses and Expansion slots ............................................................................................ 31
On-Board Ports and Connectors ................................................................................... 33
Review Questions ........................................................................................................... 35
Lab (4): Numbering systems ....................................................................................... 37
Objectives ...................................................................................................................... 37
Review Questions ........................................................................................................... 37
Lab (5): Introduction to Assembly ................................................................................ 39
Objectives ...................................................................................................................... 39
What is assembly language? ......................................................................................... 39
Inside the CPU ............................................................................................................... 39
  1. General purpose registers ....................................................................................... 39
  2. Segment registers .................................................................................................... 40
  3. Special purpose registers ....................................................................................... 41
Variables ........................................................................................................................ 41
  Arrays .......................................................................................................................... 41
Constants ....................................................................................................................... 42
Review Question: .......................................................................................................... 43
Lab (6): EMU8086 ....................................................................................................... 44
Objectives ...................................................................................................................... 44
Using the emulator ........................................................................................................ 45
Review Questions ................................................................................................................. 46
Lab (7): Memory Addressing and Instructions ......................................................................... 47
Objectives ............................................................................................................................. 47
Memory Addressing Modes .................................................................................................... 47
Memory Access Instructions ................................................................................................. 48
1. MOV instruction .............................................................................................................. 48
2. XCHG Instruction ........................................................................................................... 49
Review Questions: ................................................................................................................ 50
Lab (8): Arithmetic and Logic Instructions .............................................................................. 52
Objectives ............................................................................................................................. 52
Arithmetic and Logic Instructions ........................................................................................ 52
1. First group: ADD, SUB, CMP, AND, TEST, OR, XOR ...................................................... 53
2. Second group: MUL, IMUL, DIV, IDIV ............................................................................ 54
3. Third group: INC, DEC, NOT, NEG ................................................................................ 55
Review Questions: ................................................................................................................ 55
Lab (9): Program flow control .................................................................................................. 56
Objectives ............................................................................................................................. 56
Unconditional jumps ............................................................................................................. 56
Short Conditional Jumps ....................................................................................................... 57
1. Jump instructions that test single flag ........................................................................... 57
2. Jump instructions for signed numbers ........................................................................... 58
3. Jump instructions for unsigned numbers ....................................................................... 58
Loops ..................................................................................................................................... 59
Review Questions ................................................................................................................... 61
Lab (10): Procedures, Interrupts and Stack ............................................................................... 62
Objectives ............................................................................................................................. 62
Procedures ............................................................................................................................. 62
Interrupts ............................................................................................................................... 64
The Stack ............................................................................................................................... 66
Review Question .................................................................................................................... 68
Final Project ............................................................................................................................. 69
Lab (1): Computer Anatomy

Objectives

- Identify the basic components of computer and their working
- Explain the importance of various units of a computer

What is a Computer?

A computer is a combination of sets of electronic modules united to meet the needs of its user in a symphony of electronic pulses traversing through printed board connectors, integrated circuits, wires and switches.

Computers come in all shapes and sizes; some are capable of doing millions of calculations in a fraction of a single second, while others may take longer to do the simplest calculations.

Computers generally consist of three major component categories (System, Chassis, and Peripherals) with each category further divides into sub-categories of either essential or optional components.

![Figure (1.1) Major Computer Components](image)

**System**

The System itself is the inner makings of the computer contained within the Chassis. The system consists of a combination of hardware and software coexisting in a symbiotic relationship. That is to use the hardware, a piece of software such as an Operating System (OS) or device driver is needed; vice versa, in order to use a piece of software, existence of compatible hardware is needed. A System must utilize 6 categories of components to classify it as a computer system.

Major Components of a computer system are:
1. Motherboard
2. CPU
3. Memory
4. Video Adapter
5. Operating System
6. Connectivity

1. Motherboard

Even though the processor is the brain of the operation, without a Motherboard (MB) all you will have is a cluster of components, which do not fit each other. Consider the motherboard as the skeleton or a computer where parts, similar to human organs, fit within it.

Figure (1.2) Motherboard Layout

Although different manufacturers build different types of motherboards in a variety of sizes, shapes and integrated features, all motherboards share some basic elements. All motherboards must include an integrated chipset, which establishes the types of components that are compatible with it. All MBs must accommodate at least 1 CPU, capabilities for some type storage device, as well as some common input/output devices such as keyboard and mouse which we’ll discussed later.

Every motherboard includes BIOS (Basic Input/output System) chip, which performs system checks at boot-up time of the system. The BIOS also provides the means of communication between the processor and installed peripherals. Most BIOSes allow a limited amount of customization such as power-saving functions and even password protection for the system even before the operating system (OS) loads as a method of protecting your computer from unauthorized users.
2. Central Processing Unit (CPU)

Also known as, “The Processor”, the CPU receives the user’s instructions from the connected input devices, processes it and sends processed data to the appropriate components and output devices. The CPU is where all complex calculations are carried out, as such the faster the CPU the quicker the programs will function and respond. Intel’s Core™ i7 Extreme Edition 10-core processors, AMD’s FX series 8-cores processors, AMD’s Phenom™ X4 Quad-Core are amongst the fastest processors on the market today. If you are looking for blazing fast speed then you certainly want one of these processors. Without the CPU, the whole computer is rendered useless.

Like any piece of technology, CPUs come in all shape and sizes. There are 3 major aspects to any CPU:

1. Type: There are different standards for a Processor to select from. Some of the most popular ones are Intel, AMD, and Motorola.

2. Class: Different classes have different minimum and maximum speeds. Some examples of Class are: i7, i5, i3, Celeron, Xeon, and Athlon.

3. Speed: Measured in Hertz, it is the speed at which calculations are processed. Examples of values are 4.0GHz and 1.2GHz.

3. Memory

Every time an application is opened, the CPU allocates a specified amount of available RAM space to provide faster reaction time to user commands within that application. RAM is a temporary storage space meaning once the application is done using the RAM space, it will be once again available to be used by the next application that needs it. Once RAM’s capacity is reached, the processor redirects its focus to available HDD space also referred to as Virtual Memory (VM). It should be pointed out that HDDs are slower than RAM and as such once your computer resorts to Virtual Memory, applications begin to slow down. As a rule of
thumb, more RAM allows you to run more applications simultaneously without slowing your system’s performance.

4. **Video Adapter**

The Video adapter is often integrated onto the motherboard or as an add-on video board inserted into motherboard slot connector. It gives the system display capabilities. The video adapter turns the processed and translated user input, if any, along with any other information provided by an application and turns it into a video signal to be sent to a display device. Like any data in need of processing, this is done by utilizing the combined usage of system’ CPU and memory. More recent video adapters include onboard processor and memory to increase graphical processes such as those needed in games and applications used by photographers. Integrated video cards come with a blue jack to connect the monitor. Some add-on video adapters also include S-Video, Digital Video Input (DVI) and even Cable TV connectors.
5. Operating System

Although not a piece of hardware, without an Operating System (OS) a computer is simply a pile of technology put together like Lego blocks. Selecting an OS is a matter of personal choice; however, there are facts and figures that assist us in the decision on what is best for us or our organization. Having dominated the computing market Microsoft’s Windows Operating Systems can be found virtually anywhere. Apple’s overhaul of its systems to match Microsoft’s hardware needs has made the MacOS a worthy competitor to that of Microsoft Windows OS. Linux is a freeware OS majorly used by scientist and programmers around the world. Linux and other variations of it have brought third world countries in the 21st century and provide capabilities matching those of Microsoft and Apple Operating systems.

6. Connectivity

Network connectivity of a computer has become one of the most essential factors as we proceed to integrate computers in all aspects of our lives. All computers poses one or both of 2 methods of connectivity, Wired and Wireless, as means to connect to the Internet, a Local Area Network (LAN), mobile devices or direct (ad-hoc) connection to another computer.
Chassis

The Chassis of the computer is the casing that holds and protects all internal components of the computer system from the hazards of the environment around them.

The most common system case type is the tower. Depending on the specific number of internal drive bays and the height of the tower, these cases can be further classified into mini-size, mid-size and full-size tower cases. One of the biggest considerations when choosing between case sizes is the number of slots and the number of devices we would like to add to those cases.

1. Full Tower
Full-tower cases are generally big with a height that is about or more than 30 inches (more than 76 cm). The number of internal drive bays inside these cases can be between 6 and 10.

2. Mid Tower
Another case that might be a step down, would be classified as a mid tower case. Mid-tower cases are the most widely used computer cases. Mid Tower cases are about 18 to 24 (45 to 60 cm) inches high and they usually contain 2 to 4 internal drive bays and a similar number of external bays (for CD/DVD readers and similar).

3. Mini Tower
Mini-tower usually has up to 2 or sometimes 3 internal drive bays. Mini-cases normally stand at a height of 12 to 18 inches (30 to 45 cm). Expandability is a problem with these cases

4. Slim Line Case
Slim line cases are simply tower cases turned on their sideways. They can hold a monitor on top of the case.

5. Small Form Factor (SFF) Case
Small form factor or SFF cases are custom cases that are designed to minimize the spatial volume of a desktop computer. SFFs are available in a variety of sizes and shapes, including shoe boxes, cubes, and book-sized PCs.
Cooling System
Manufacturers continue trying to develop new devices that take less power to function and generate less heat than the previous generation. However, like any mechanical and electronic device having moving parts and/or using electricity, a computer generates a considerable amount of heat. As such, one of the major parts of any computer case is some type of cooling system.

Spot Cooling (Heat Sink): A primitive form of cooling where a piece of metal specially designed to absorb and disburse heat is attached to a component that generates heat when used.

Air Cooling: Fans are the traditional and most common method to cooling the system as well as a companion to spot cooling of components where spot cooling is not sufficient. Such components include CPU and RAM.

Liquid Cooling: Liquid cooling is the most recent method of cooling a computer. These cooling systems function similar to that of a refrigerator where a coolant circulates through the system by the use of a compressor. Although expensive, it is the most efficient form of cooling system.

Power Supply Unit
The power supply unit (PSU) provides the power needed for each component to function. Regardless of what type of computer you are using, the PSU converts AC power from any source to low voltage DC power to be used for the internal components of the computer.
Peripherals
A peripheral device is an optional device which when attached to a computer expands its abilities to include specific specialized tasks. There are 3 categories of peripherals, those that assist with input of data and instructions such as a mouse or keyboard, those, which output processed data such as a printer or monitor, and lastly those performing both input and output tasks such as storage devices.

1. **Input Devices**
Input devices are conduits of information transfer from the user to the computer.

**Keyboard**
A keyboard typically has characters engraved or printed on the keys (buttons) and each press of a key typically corresponds to a single written symbol. However, to produce some symbols requires pressing and holding several keys simultaneously or in sequence. While most keyboard keys produce letters, numbers or signs (characters), other keys or simultaneous key presses can produce actions or execute computer commands.

![Keyboard Image](image)

**Mouse**
A computer mouse is a pointing device (hand control) that detects two-dimensional motion relative to a surface. This motion is typically translated into the motion of a pointer on a display, which allows a smooth control of the graphical user interface. Physically, a mouse consists of an object held in one's hand, with one or more buttons.
**Scanners**

An image scanner—often abbreviated to just scanner, is a device that optically scans images, printed text, handwriting or an object and converts it to a digital image. Commonly used in offices are variations of the desktop flatbed scanner where the document is placed on a glass window for scanning. Hand-held scanner is moved by hand.

Modern scanners typically use a charge-coupled device (CCD) or a contact image sensor (CIS) as the image sensor. Digital cameras can be used for the same purposes as dedicated scanners. When compared to a true scanner, a camera image is subject to a degree of distortion, reflections, shadows, low contrast, and blur due to camera shake.

**Audio**

Audio devices such as microphones allow the recording of audio such as speech and music for later use in presentations or in creation of a digital compilation media like a Music CD.

2. **Output Devices**

For every input, there is an output to recognize the receipt of an input and to provide a representation of processed data provided via input devices.

**Monitors**

Monitors also referred to as visual display devices are the most common and well-known output devices similar to TVs such that they provide a visual representation of processed data such as images, videos and documents.

**Printers**

Printers are devices that present data shown on a display device on various types of paper in color or black & white by the means of laser, ink or dot matrix technologies.

**Audio**

Audio devices such speakers and headphones act as the computers mouth vocalizing digital audio such as music and video soundtracks and speeches.
3. **Input & Output Devices**

There exist peripherals, which are used as both input and output devices. Most common of such devices are Removable and Fixed storage devices, such as Disk drives, CDs, and DVDs.

**Media Storage**

Storage devices allow us to store our work for future access, might it be from another computer or on the same system. Generally, there are three types of media storage, magnetic, optical and semiconductor. Magnetic drives are such as Floppy and Hard drives store data on magnetic surfaces. Optical drives are such as CRROM, DVD, Blu-ray. Semiconductor drives are flash disks, and memory cards.
Review Questions

1. What are the major components of computer?

2. What is the role of Central Processing Unit?

3. What is the role of BIOS?

4. What are the current types of RAM module in use now?

5. Which video adapter is better integrated onto the motherboard ones or standalone ones? And why?
6. What is the function of Operating Systems?

7. Name three input devices.

8. Name three output devices.
Lab (2): PC I/O Ports

Objectives

- Identify the ports computer and their functions.

The purpose of this Lab is to discuss the I/O ports and jacks commonly used with modern PCs. Receptacles or sockets on computers are known as ports or jacks. The part that is inserted into a port is known as a plug. The plug is commonly referred to as the "male end" of the connection, and the port is often referred to as the "female end."

Mini-DIN

The Mini-DIN standard is a family of small (9 mm) cylindrical connectors with different pin outs for different purposes. The Mini-DIN is similar to the larger, older DIN connector. Both are standards of the Deutsches Institut für Normung, the German standards body.

PS/2

The most widely utilized Mini DIN connector utilized in the computer industry is the 6-pin PS/2 keyboard and mouse connector (shown right). Per the PC 99 System Design Guide specification, the PS/2 keyboard port is colored purple and PS/2 mouse port is colored green. These ports also often have a small keyboard icon and mouse icon next to them, respectively, for identification.

Separate Video (S-Video)

Another mini-DIN connector mentioned in your text is the Separate Video (S-Video) connector. It is an analog video-only connector (colored yellow) similar in size and shape to the PS/2 connector except it has a pin count of four instead of six. It is found on consumer TVs, DVD players, digital TV receivers, DVRs, game consoles, and graphics cards. For the most part, it has been replaced by the DVI and HDMI digital video standards.

One should be careful to properly align the square plastic key on mini-DIN plugs to avoid bending pins when inserting it into a port. Most connectors include an arrow or flat section which is usually oriented to the right or top of the port for proper alignment.
Universal Serial Bus (USB)

The use of USB ports is by far the most popular means of attaching peripherals to computers today. Many of the following peripheral devices have USB capability:

<table>
<thead>
<tr>
<th>Keyboards</th>
<th>Mice</th>
<th>Printers</th>
<th>Scanners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital cameras</td>
<td>Wi-Fi adapters</td>
<td>PDAs</td>
<td>Smart phones</td>
</tr>
<tr>
<td>Flash drives</td>
<td>External hard drives</td>
<td>Floppy drives</td>
<td>Optical drives</td>
</tr>
</tbody>
</table>

Most modern computers are equipped with four or more Type A USB ports. USB connections can provide a small amount of power to a device, and devices are hot-swappable - but remember to stop the device, if applicable, before removing it. USB connectors come in a variety of sizes and configurations, as shown below.

Micro and Mini connectors (shown to the left above) were designed for small devices such as cameras and PDAs. A Type A port as found on the back of a computer is shown to the right. There are three versions of USB as shown in the table below. The latest, USB 3.0, will be deployed on motherboards in Q1 2010. The USB 3.0 connector has the same shape as its predecessors, but plugs are longer and the ports are deeper.
### Firewire (IEEE 1394)

Firewire is used with audio-visual peripherals, such as camcorders, because it is somewhat faster than USB 2.0 and can provide more power to the device. For this same reason, it is also sometimes used with external hard drives. The six-pin plug (shown to the left) is commonly used to connect Firewire peripherals to computers.

As with USB, Firewire is hot-swappable. The table below outlines some Firewire specifications:

<table>
<thead>
<tr>
<th>Firewire Version</th>
<th>Ratification Date</th>
<th>Data Rate (Mbits/s)</th>
<th>Max Cable Length (m)</th>
<th>Output Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>400 - IEEE 1394</td>
<td>200</td>
<td>100, 200, 400</td>
<td>4.5</td>
<td>Up to 45 watts, but 7 - 8 watts is more typical.</td>
</tr>
<tr>
<td>800 - IEEE</td>
<td>200</td>
<td>786, 3200</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1394b S1600 &amp; S3200 modes</td>
<td>200 &amp; 8</td>
<td>1600, 3200</td>
<td>10 &amp; 0</td>
<td></td>
</tr>
</tbody>
</table>

Some computers intended for audio/video use have built-in FireWire ports. Windows Visa and prior only supported the 1394a standard. Microsoft has rewritten the 1394 bus driver for Windows 7 to provide support for advanced versions of the 1394 standard depicted above.

---

1. Sustained USB 2.0 throughput is typically no more than 280 Mbit/s.
2. It is predicted that USB 3.0 with its high data rate will very quickly overshadow and possibly replace other technologies such as Firewire and eSATA.
DB (D-subminiature, D-shell) Connectors

The proper name for DB connectors is actually **D-subminiature**. They are also called **D-shell** connectors because they are shaped like the letter D. The DB-25 connector (pictured below) was the first **D-shell** introduced and subsequently became part of the IEEE-1284 standard. The IEEE-1284 standard also includes the legacy 36 pin Centronics connector that is used to connect to printers (not pictured).

The smaller DE-15, DE-9, and DA-15 subminiature connector (pictured below) were introduced later to provide VGA video, RS-232 serial, and a gaming device interfaces, respectively. The "B" in DB and the "E" in DE and the "A" in DA refer to shell size. The table below summarizes the D-subminiature connectors commonly found on computers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Depiction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB-25</td>
<td>A legacy 25-pin parallel connector commonly used with printers. The newest printer interface is USB.</td>
<td><img src="image" alt="DB-25 Depiction" /></td>
</tr>
<tr>
<td>DE-15 VGA</td>
<td>A 15-pin analog monitor connector referred to in your text as a VESA connector. Also commonly referred to as a VGA (Video Graphics Array) or &quot;DB-15&quot; connector.</td>
<td><img src="image" alt="DE-15 VGA Depiction" /></td>
</tr>
<tr>
<td>DE-9</td>
<td>A nine-pin serial connector used for attaching RS-232 devices such as external modems. Commonly called &quot;DB-9.&quot;</td>
<td><img src="image" alt="DE-9 Depiction" /></td>
</tr>
<tr>
<td>DA-15</td>
<td>A legacy 15-pin connector that was commonly found on audio adapters for connecting gaming devices such as Joysticks. USB and Bluetooth (wireless) are among the newest methods for</td>
<td><img src="image" alt="DA-15 Depiction" /></td>
</tr>
</tbody>
</table>

Registered Jack (RJ)

Two types of Registered Jack (RJ) ports are commonly found with computers: the **RJ-11** and the **RJ-45**. Both are similar in shape with the RJ-45 (pictured right) being slightly larger. The RJ-11 is the standard modular phone connector and is typically found in pairs on modem devices. The RJ-45 is used to connect network adapters to Ethernet networks.
An RJ-45 port is pictured to the left. There are normally a couple of light-emitting diodes (LEDs) found next to it that provide network connectivity and activity status. The RJ-45 plug is usually connected to various types of eight-wire unshielded twisted pair (UTP) cable that can have a maximum segment length of 100 meters when used in an Ethernet network.

A pair of RJ-11 ports, as typically found on modems, is depicted to the right. One port is used to connect the modem to a phone jack and the second port is used to connect to the modem to a phone.

Audio

1. Analog

Three analog 3.5 mm (1/8") cylindrical audio ports, as depicted on the left, are typically found on the back of computers for connecting to a microphone, speakers and line in. These ports are colored pink, lime green, and light blue, respectively - and are formally known as tip, ring, and sleeve (TRS) connectors. However, they are also referred to as "mini-jacks." You are probably familiar with microphones and speakers; however, they may not be familiar with a line in connection. A line in connection is used for inputting a signal from a cassette tape recorder or similar sound source.

2. Digital

Some advanced audio adapters may also have an RCA port (depicted right) to support Sony/Phillips digital interface format (S/PDIF) audio output. This port, color-coded orange, can be used to connect the computer to a home theater receiver equipped with surround-sound speakers.

Digital Visual Interface (DVI)

DVI is a video interface standard used with digital display devices such as flat screen monitors. A typical DVI port (colored white per the PC 99 Design Guide) is shown to the left.

DVI is compatible with the High-Definition Multimedia Interface (HDMI) standard. However, DVI only supports video whereas HDMI supports both video and audio.
HDMI (High-Definition Multimedia Interface)

HDMI, available since 2003, supports both audio and video and is now the de facto standard for digital (flat screen) TVs, including HDTV. It is also widely available on digital (flat screen) PC monitors. Other uses for HDMI include Blue-ray disc players which were introduced in 2006.

HDMI complements DVI in that, with an appropriate adapter, HDMI video can be ported to a DVI interface.

There are four types of HDMI connectors as outlined in the table below. As with USB, the Type A connector is the one primarily used to connect electronic devices to desktop computers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Pins</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>14 mm × 4.55 mm</td>
<td>19</td>
<td>Standard version</td>
</tr>
<tr>
<td>B</td>
<td>21.2 mm × 4.45 mm</td>
<td>29</td>
<td>Future use</td>
</tr>
<tr>
<td>C</td>
<td>10.42 mm × 2.42 mm</td>
<td>19</td>
<td>Mini version</td>
</tr>
<tr>
<td>D</td>
<td>6.4 mm x 2.8 mm</td>
<td>19</td>
<td>Micro version</td>
</tr>
</tbody>
</table>

DisplayPort

Consumer electronics and PC manufacturers incorporating HDMI into their devices must pay a royalty fee of $10,000 per year plus four cents per unit. In 2008, a new audio/visual interface standard named DisplayPort became available that is royalty free and provides a superset of features found in DVI and the core set of features found in HDMI. It was created by Video Electronics Standards Association (VESA) as a more up-to-date alternative to DVI and a cheaper alternative to HDMI. It is targeted primarily towards PC manufactures as a means of connect computers to monitors, and computers to home-theater system. Both Dell and Apple are currently shipping PCs and monitors equipped with Display Port - often as a replacement for DVI.

Display Port video has several technical advantages over DVI video and uses a much smaller connector that doesn't require thumbscrews. Display Port connector pins don't run the risk of bending if improperly handled as is the case with DVI. Further, the DisplayPort connector is easier to connect when guided only by touch. Due to its enhanced capability and connector design, DisplayPort may eventually replace DVI in the PC industry. It may also make inroads into HDMI - primarily due to cost.
eSata

The eSATA connector (shown right) is almost exclusively used to connect external SATA drives to computers. The SATA standard's throughput of 3000 Mbits/s is considerably faster than that of USB 2.0 and Firewire 400 IEEE 1394 - however, it does not provide power. It may be overshadowed by the upcoming deployment of USB 3.0 which has a throughput of 4800 MBits/s and can provide power.

**Summary of Some Bus Speeds**

<table>
<thead>
<tr>
<th>Bus</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB 2.0</td>
<td>480 (theoretical)</td>
</tr>
<tr>
<td>Firewire 400</td>
<td>400</td>
</tr>
<tr>
<td>Firewire 800</td>
<td>786</td>
</tr>
<tr>
<td>Firewire 1600</td>
<td>1600</td>
</tr>
<tr>
<td>Firewire 3200</td>
<td>3200</td>
</tr>
<tr>
<td>eSATA</td>
<td>3000</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>4800 (theoretical)</td>
</tr>
</tbody>
</table>
# Summary of External Connector

<table>
<thead>
<tr>
<th>Connector</th>
<th>Usage</th>
<th>Connection Color</th>
<th>Hot Swappable</th>
<th>Provides Power</th>
<th>Depiction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS/2 Mini-DIN</td>
<td>Keyboard</td>
<td>Purple</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="PS/2 Mini-DIN" /></td>
</tr>
<tr>
<td></td>
<td>Mouse</td>
<td>Green</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="PS/2 Mini-DIN" /></td>
</tr>
<tr>
<td>USB</td>
<td>Numerous Serial devices</td>
<td>Black</td>
<td>Yes</td>
<td>Yes</td>
<td><img src="image" alt="USB" /></td>
</tr>
<tr>
<td>Firewire IEEE 1394</td>
<td>Audio-visual peripherals; external hard drives</td>
<td>Grey</td>
<td>Yes</td>
<td>Yes</td>
<td><img src="image" alt="Firewire IEEE 1394" /></td>
</tr>
<tr>
<td>DB-25 IEEE 1284 Parallel port</td>
<td>Printers</td>
<td>Fuschia</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="DB-25 IEEE 1284 Parallel port" /></td>
</tr>
<tr>
<td>DE-15 VGA port</td>
<td>Monitors</td>
<td>Blue</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="DE-15 VGA port" /></td>
</tr>
<tr>
<td>DE-9 RS-232 serial port</td>
<td>External modems</td>
<td>Teal</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="DE-9 RS-232 serial port" /></td>
</tr>
<tr>
<td>DA-15 Game Port</td>
<td>Gaming devices</td>
<td>Gold</td>
<td>No</td>
<td>No</td>
<td><img src="image" alt="DA-15 Game Port" /></td>
</tr>
<tr>
<td>RJ-45</td>
<td>Network adapters</td>
<td>--</td>
<td>Yes</td>
<td>Possibly</td>
<td><img src="image" alt="RJ-45" /></td>
</tr>
<tr>
<td>RJ-11</td>
<td>Modems</td>
<td>--</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="RJ-11" /></td>
</tr>
<tr>
<td>TRS Mini Jacks</td>
<td>Microphone Speaker Line in</td>
<td>Pink, Lime, Green, Light Blue</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="TRS Mini Jacks" /></td>
</tr>
<tr>
<td>S/PDIF</td>
<td>Digital audio</td>
<td>Orange</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="S/PDIF" /></td>
</tr>
<tr>
<td>DVI</td>
<td>Digital video</td>
<td>White</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="DVI" /></td>
</tr>
<tr>
<td>HDMI</td>
<td>Digital Audio/Video</td>
<td>--</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="HDMI" /></td>
</tr>
<tr>
<td>DisplayPort</td>
<td>Digital Audio/Video</td>
<td>--</td>
<td>Yes</td>
<td>No</td>
<td><img src="image" alt="DisplayPort" /></td>
</tr>
</tbody>
</table>
Review Questions

1. What are two ways that USB version 2.0 ports can be added to a computer?
   A. Implement a USB hub
   B. Add a USB 1.1 expansion card
   C. Add a USB 2.0 expansion card
   D. Use an Ethernet hub

2. A serial port can hook up devices such as external modems and label printers. What is this port usually called?
   A. SCSI port
   B. COM port
   C. PS/2 ports
   D. Parallel port

3. Which of the following devices can be used for a printer port, scanner, or removable media?
   A. PS/2 port
   B. Parallel port
   C. NIC card
   D. I/O port

4. Which device is known as IEEE 1394?
   A. USB
   B. Parallel port
   C. FireWire
   D. PS/2

5. Some desktop systems and many of the older laptop and portable systems include a port to connect a mouse or keyboard. What is this port called?
   A. USB
   B. FireWire
   C. BIOS
   D. PS/2

6. What is the standard size of the audio mini-jack used by sound cards?
   A. 1 1/2 inch
   B. 1/8 inch
   C. 2 1/2 inch
   D. 1 inch

7. There are currently three standards for USB ports. What are they? (Choose three.)
   A. USB 1.1
   B. USB 3.0
   C. USB 1.0
   D. USB 2.0

8. If you run out of USB ports and need more, which of the following devices are available? (Choose all that apply.)
A. Motherboard connectors  
B. USB hubs  
C. Add-on cards  
D. Extra PCI slots  

9. Which device still remains the primary method used to enter data and send commands to the computer?  
   A. Mouse  
   B. Gamepad  
   C. Stylus  
   D. Keyboard  

10. Which of the following is used to transfer data into a computer by pressing on screen icons?  
    A. Touch screen monitors  
    B. CRT monitors  
    C. LCD monitors  
    D. Serial ports  

11. Which of the following are considered multimedia devices? (Choose all that apply.)  
    A. Webcam  
    B. Sound card  
    C. Microphone  
    D. All of these options are correct.  

12. You are in the process of installing a new keyboard to a new PC. Which of the following is the most common type of connector to use?  
    A. Serial connector  
    B. PS/2 connector  
    C. USB connector  
    D. IEEE1394 connector  

13. What is the major difference between a laser printer and an LED printer? (Choose all that apply.)  
    A. LED printers use an LED array to perform the transfer of images.  
    B. LED printers use an LED drum.  
    C. Laser printers are of better print quality.  
    D. Laser printers use a laser to transfer the image to the drum.
14. Most inkjet, laser, and thermal printers use this interface to connect a printer to a computer. (Choose two.)

   A. RJ-45
   B. USB
   C. Parallel
   D. LED

15. What are three major types of display devices that are in use in today's industry? (Choose all that apply.)

   A. CRT monitors
   B. LCD monitors
   C. Data projectors
   D. USB monitors

16. Identify three types of video connectors.

   A. VGA type
   B. DMI type
   C. HDMI type
   D. USB type
Lab (3): Motherboard

Objectives

- Identify the different types and features of motherboards.
- Understand how firmware on the motherboard controls a PC
- Learn to install, configure, and maintain a motherboard

A motherboard is the most complicated component in a computer. When you put together a computer from parts, generally you start with deciding on which processor and motherboard you will use. Everything else follows those decisions. Take a look at the details of Figure 3-1, which shows a motherboard designed with gamers in mind. We’ll also look at how configuration information is stored on a motherboard and the best strategies to use when selecting a motherboard.

Figure 3-1 ASUS Z97 ATX-Motherboard details

Motherboard Form Factor

A motherboard form factor determines the size of the board and its features that make it compatible with power supplies, cases, processors, and expansion cards. The most popular
motherboard form factors are ATX, Mini ATX, EATX, MicroATX, and FlexATX. ATX motherboards have been around for a long time and have seen many improvements. Figure 3-2 shows the differences among ATX family form factor.

![ATX family Form Factors](image)

**Processors Sockets**

Another important feature of a motherboard is the processor socket. This socket and the chipset determine which processors a board can support. A socket will hold either an Intel or AMD processor. Some older processors were installed on the motherboard in a long narrow slot, but all processors sold today use sockets. Table 3-1 lists the sockets currently used by Intel processors for desktop systems.

Earlier Pentiums used a pin grid array (PGA) socket, with pins aligned in uniform rows around the socket. Later sockets used a staggered pin grid array (SPGA), with pins staggered over the socket to squeeze more pins into a small space. Small pins can easily be bent as the processor is installed in the socket. Later Intel sockets use a land grid array (LGA) that uses lands rather than pins. The first LGA socket is the LGA775 socket. It has 775 lands and is shown with the socket lever and top open in Figure 3-4. The lands look like tiny pads that the pins on the processor contact. The latest Intel socket is the LGA1366 socket. It’s called a land grid array socket, but the lands in the socket are actually more like pins that connect with lands on the bottom of the processor. Figure 3-5 shows an LGA1366 socket with the load plate and load lever lifted so that the socket is open and ready to receive the processor.

PGA, SPGA, and LGA sockets are all square or nearly square. So that even force is applied when inserting the processor in the socket, all current processor sockets have a lever on the side of the socket. These sockets are called zero insertion force (ZIF) sockets, and this lever is used to lift the processor up and out of the socket. Push the lever down and the processor moves into its pin or land connectors with equal force over the entire housing. With this method, you can easily remove and replace the processor if necessary. However, know that processors generally
should not be removed or replaced repeatedly because this can damage the delicate pins or socket holes.

<table>
<thead>
<tr>
<th>Socket name</th>
<th>Year</th>
<th>CPU families supported</th>
<th>Computer type</th>
<th>Package</th>
<th>Pin count</th>
<th>Bus clock &amp; transfers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket 478/Socket N</td>
<td>2000</td>
<td>Intel Pentium 4</td>
<td>Desktop</td>
<td>PGA</td>
<td>478</td>
<td>100–200 MHz</td>
<td>400–800 MT/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Celeron</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Pentium 4 EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Pentium 4 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket 495</td>
<td>2000</td>
<td>Intel Celeron</td>
<td>Notebook</td>
<td>PGA</td>
<td>495</td>
<td>66–133MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Pentium III</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket 479</td>
<td>2003</td>
<td>Intel Pentium M</td>
<td>Notebook</td>
<td>PGA</td>
<td>479[iv]</td>
<td>100–133 MHz</td>
<td>400–533 MT/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Celeron M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rPGA 988A/Socket G1</td>
<td>2008</td>
<td>Intel Core i7 (600, 700, 800, 900 series)</td>
<td>Notebook</td>
<td>rPGA</td>
<td>988</td>
<td>2.5 GT/s, 4.8 GT/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i5 (400, 500 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i3 (300 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Pentium (P6000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Celeron (P4000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket 441</td>
<td>2008</td>
<td>Intel Atom</td>
<td>Sub-notebook</td>
<td>PGA</td>
<td>441</td>
<td>400–667 MHz</td>
<td></td>
</tr>
<tr>
<td>LGA 1156/Socket H</td>
<td>2009</td>
<td>Intel Core i7 (800 series)</td>
<td>Desktop</td>
<td>LGA</td>
<td>1156</td>
<td>2.5 GT/s</td>
<td>DMI bus is a (perhaps modified) PCIe x4 v1.1 interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i5 (700, 600 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i3 (500 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Xeon (X3400, L3400 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Pentium (G6000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Celeron (G1000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rPGA 988B/Socket G2</td>
<td>2011</td>
<td>Intel Core i7 (2000, 3000 series)</td>
<td>Notebook</td>
<td>rPGA</td>
<td>988</td>
<td>2.5 GT/s, 4.8 GT/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i5 (2000, 3000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Core i3 (2000, 3000 series)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket name</td>
<td>Year</td>
<td>CPU families supported</td>
<td>Computer type</td>
<td>Package</td>
<td>Pin count</td>
<td>Bus clock &amp; transfers</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
<td>------------------------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>LGA 1150/Socket H3</td>
<td>2013.06.03</td>
<td>Intel Haswell</td>
<td>Desktop</td>
<td>LGA</td>
<td>1150</td>
<td>?</td>
<td>used for Intel's 4th generation (Haswell/Haswell Refresh) processors</td>
</tr>
<tr>
<td></td>
<td>2014.05.11</td>
<td>Intel Haswell Refresh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Broadwell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2015.06.02)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket G3</td>
<td>2013/Q2</td>
<td>Intel Haswell</td>
<td>Notebook</td>
<td>rPGA</td>
<td>946</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Broadwell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGA 1151</td>
<td>2015</td>
<td>Intel Skylake</td>
<td>Desktop</td>
<td>LGA</td>
<td>1151</td>
<td>?</td>
<td>used for Intel's 6th generation (Skylake) and 7th generation (Kaby Lake) processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Kaby Lake</td>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGA 2066/Socket R4</td>
<td>2017</td>
<td>Intel Skylake-X</td>
<td>Desktop</td>
<td>LGA</td>
<td>2066</td>
<td>?</td>
<td>Used for Intel's 7th generation (Skylake-X &amp; Kaby Lake-X) series of Core-X processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Kaby Lake-X</td>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3-1. Intel Sockets
Chipset

Each socket is supported by the chipset. The chipset and associated components provide peripheral connectivity to the CPU. Because the chipset defines the types and limits of most connections between the CPU and peripherals, it may be the most important consideration in a motherboard. A chipset is basically composed of a Northbridge and a Southbridge, although the last few generations of Intel CPUs and recent AMD APUs integrate Northbridge functions within the CPU die. The interfaces requiring the greatest performance are integrated onto the Northbridge, and slower interfaces connect through the Southbridge. Since they are the oldest chipsets available today.

Buses and Expansion slots

Look on the bottom of the motherboard, and you see a maze of circuits that make up a bus. These embedded wires carry four kinds of cargo:

- **Electrical power.** Chips on the motherboard require power to function. These chips tap into a bus’s power lines and draw what they need.
- **Control signals.** Some wires on a bus carry control signals that coordinate all the activity.
- **Memory addresses.** Components pass memory addresses to one another, telling each other where to access data or instructions. The number of wires that make up the memory address lines of the bus determines how many bits can be used for a memory address. The number of wires thus limits the amount of memory the bus can address.
Data. Data passes over a bus in a group of wires, just as memory addresses do. The number of lines in the bus used to pass data determines how much data can be passed in parallel at one time. The number of lines depends on the type of processor and determines the number of bits in the data path. (Remember that a data path is the part of the bus on which the data is placed; it can be 8, 16, 32, 64, or more bits wide.)

The system clock keeps the beat for components. If a component on the motherboard works by the beat, or clock cycle, then it is synchronized, or in sync, with the processor. For example, the back-side bus of the Pentium works at half the speed of the processor. This means that the processor does something on each clock cycle, but the back-side bus is doing something on every other clock cycle.

Some components don’t attempt to keep in sync with the processor, even to work at one-half or one-third of clock cycles. These components work asynchronously with the processor. They might work at a rate determined by the system clock or by another crystal on or off the motherboard. Either way, the frequency is much slower than the processor’s and not in sync with it. If the processor requests something from one of these devices and the device is not ready, the device issues a wait state, which is a command to the processor to wait for slower devices to catch up.

<table>
<thead>
<tr>
<th>Bus Type</th>
<th>Bus Type</th>
<th>Data Path in Bits</th>
<th>Address Lines</th>
<th>Bus Frequency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>System bus</td>
<td>Local</td>
<td>64</td>
<td>32 or 64</td>
<td>Up to 1600 MHz</td>
<td>Up to 3.2 GB/sec</td>
</tr>
<tr>
<td>PCI Express Version 2</td>
<td>Local video and local I/O</td>
<td>Serial with up to 32 lanes</td>
<td>Up to 32 lanes</td>
<td>2.5 GHz</td>
<td>Up to 500 MB/sec per lane in each direction</td>
</tr>
<tr>
<td>PCI Express Version 1.1</td>
<td>Local video and local I/O</td>
<td>Serial with up to 16 lanes</td>
<td>Up to 16 lanes</td>
<td>1.25 GHz</td>
<td>Up to 250 MB/sec per lane in each direction</td>
</tr>
<tr>
<td>PCI Express Version 1</td>
<td>Local video and local I/O</td>
<td>Serial with up to 16 lanes</td>
<td>Up to 16 lanes</td>
<td>1.25 GHz</td>
<td>Up to 250 MB/sec per lane in each direction</td>
</tr>
<tr>
<td>PCI-X</td>
<td>Local I/O</td>
<td>64</td>
<td>32</td>
<td>66, 133, 266, or 533 MHz</td>
<td>Up to 8.5 GB/sec</td>
</tr>
<tr>
<td>PCI</td>
<td>Local I/O</td>
<td>32 or 64</td>
<td>32 or 64</td>
<td>33, 66 MHz</td>
<td>133, 266, or 532 MB/sec</td>
</tr>
<tr>
<td>AGP 1x, 2x, 3x, 4x, 8x</td>
<td>Local video</td>
<td>32</td>
<td>NA</td>
<td>66, 75, 100 MHz</td>
<td>266 MB/sec to 2.1 GB/sec</td>
</tr>
<tr>
<td>FireWire 400 and 800</td>
<td>Local I/O or expansion</td>
<td>1</td>
<td>Serial</td>
<td>NA</td>
<td>Up to 3.2 Gbps (gigabits per second)</td>
</tr>
<tr>
<td>USB 1.1, 2.0, and 3.0</td>
<td>Expansion</td>
<td>1</td>
<td>Serial</td>
<td>3 MHz</td>
<td>12 or 480 Mbps (megabits per second) or 5.0 Gbps (gigabits per second)</td>
</tr>
</tbody>
</table>

Table 3-2 Buses list by throughput

Table 3-2 lists the various buses used on motherboards today, in order of throughput speed from fastest to slowest. (Throughput is sometimes called bandwidth.) Looking at the second column of Table 3-2, you can see that a bus is called an expansion bus, local bus, local I/O bus, or local video bus. A bus that does not run in sync with the system clock is called an expansion bus and always connects to the slow end of the chipset, the South Bridge. Most buses today are local buses, meaning they run in sync with the system clock. If a local bus connects to the slower I/O controller hub or South Bridge of the chipset, it is called a local I/O bus. Because the video card needs to run at a faster rate than other expansion cards, this one slot always connects to the faster end of the chipset,
the North Bridge. This video slot can be either an AGP slot or a PCI Express x16 slot, and the bus is called a local video bus.

32-Bit vs 64-Bit Slots/Boards

Figure 3-4. With PCI Version 2.x, there are four possible types of expansion slots and six differently configured PCI expansion cards to use these slots

**On-Board Ports and Connectors**

In addition to expansion slots, a motherboard might also have several on-board ports and internal connectors. Ports coming directly off the motherboard are called on-board ports or integrated components. Almost all motherboards have two or more USB ports and sound ports. Boards might also offer a network port, modem port, FireWire (IEEE 1394) port, video port, keyboard port, mouse port, parallel port, serial port, one or more eSATA ports (for external SATA hard drives), and a port for a wireless antenna.
When you purchase a motherboard, the package includes an I/O shield, which is the plate that you install in the computer case that provides holes for these I/O ports. The I/O shield is the size designed for the case’s form factor and the holes in the shield are positioned for the motherboard ports (see Figure 3-6). When you first install a motherboard, you might need to install the drivers that come on the CD bundled with the board before some of the motherboard ports will work.

Figure 3-5. A motherboard provides ports for common I/O devices

<table>
<thead>
<tr>
<th>Rear panel connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Optical S/PDIF Out port</td>
</tr>
<tr>
<td>2. HDMI port</td>
</tr>
<tr>
<td>3. USB 2.0 ports 7-10</td>
</tr>
<tr>
<td>4. Intel® LAN port (LAN2)**</td>
</tr>
<tr>
<td>5. Intel® LAN port (LAN1)**</td>
</tr>
<tr>
<td>6. DisplayPort</td>
</tr>
</tbody>
</table>

When you purchase a motherboard, the package includes an I/O shield, which is the plate that you install in the computer case that provides holes for these I/O ports. The I/O shield is the size designed for the case’s form factor and the holes in the shield are positioned for the motherboard ports (see Figure 3-6). When you first install a motherboard, you might need to install the drivers that come on the CD bundled with the board before some of the motherboard ports will work.

Figure 3-6. The I/O shield fits the motherboard ports to the computer case
Review Questions

1. Write the parts names of motherboard?
2. What are the primary functions of the motherboard and list out their various form factors?

3. List out some of the on-board supported features of modern motherboards.

4. What are the important components of a motherboard?

5. What are the differences between the northbridge and southbridge?

6. What is the role of the BIOS and CMOS in the boot process?

7. What is the CPU socket?

8. What is the slot used by graphic cards?
Lab (4): Numbering systems

Objectives
- Review and explain the importance types of numbering systems types.
- Explain the basic operations performed on numbers.

See Appendix A: Numbering Systems Handouts

Review Questions
1. Find the 1st and 2nd complement of the following decimal values?
   a) 154
   b) –413

2. What is the 3-digit 10's complement of 247?
   Answer: ____________

3. What is the 3-digit 10's complement of 17?
   Answer: ____________

4. 777 is a 10’s complement representation of what decimal value?
   Answer: ____________

5. What is -20 expressed as an 8-bit binary number in 2’s complement notation?
   Answer: ____________

6. 1100011 is a 7-bit binary number in 2’s complement notation. What is the decimal value?
   Answer: ____________

7. Add 01001111₂ to 00100011₂ using signed-magnitude arithmetic.

8. Add 10010011₂ (–19) to 00001101₂ (+13) using signed-magnitude arithmetic.

9. Subtract 01001111₂ to 01100011₂ using signed-magnitude arithmetic.

10. Subtract 01100011₂ (99) from 01001111₂ (79) using signed magnitude arithmetic.

11. Subtract 10011000₂ (–24) from 10101011₂ (–43) using signed-magnitude arithmetic.

12. Express 23₁₀ and 9₁₀ in 8-bit binary one's complement form.

13. Add 23₁₀ to –9₁₀ using one's complement arithmetic.

15. Add $9_{10}$ to $-23_{10}$ using two's complement arithmetic.

16. The 2s complement of $(01010)_2$ is

17. The 2s complement of $(0.0010)_2$ is

18. The 10s complement of $(4887)_{10}$ is

19. The 10s complement of $(48.87)_{10}$ is
Objectives
- Identify computer registers and their functions
- Identify the variable declaration and allocation in assembly.

What is assembly language?
Assembly language is a low level programming language. You need to get some knowledge about computer structure in order to understand anything. The simple computer model as I see it:

The system bus (shown in yellow) connects the various components of a computer. The CPU is where most of computations occur. RAM is a place to where the programs are loaded in order to be executed.

Inside the CPU

1. General purpose registers

8086 CPU has 8 general purpose registers; each register has its own name:
   - **AX** - the accumulator register (divided into AH / AL).
   - **BX** - the base address register (divided into BH / BL).
CX - the Count register (divided into CH / CL).
DX - the data register (divided into DH / DL).
SI - Source Index register.
DI - Destination Index register.
BP - Base Pointer register.
SP - Stack Pointer register.

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, it's something like: 0011000000111001b (in binary form), or 12345 in decimal (human) form.

Four general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= 0011000000111001b, then AH=00110000b and AL=00111001b. Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for low part.

Registers are much faster than memory because they are located inside the CPU. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

2. Segment registers
CS – Code Segment register points at the segment containing the current program.
DS – Data Segment register generally points at segment where variables are defined.
ES - Extra segment register, it's up to a coder to define its usage.
SS – Stack Segment register points at the segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.

Segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address 12345h (hexadecimal), we should set the DS = 1230h and SI = 0045h. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values.

CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h * 10h + 45h = 12345h):

\[
\begin{array}{c}
1230b \\
+ 0045b \\
\hline
12345b
\end{array}
\]

The address formed with 2 registers is called a logical address.

By default BX, SI and DI registers work with DS segment register; BP and SP work with SS segment register. Other general purpose registers cannot form a logical address! Also, although BX can form a logical address, BH and BL cannot.
3. Special purpose registers

**IP - the Instruction Pointer.**

**Flags register** - determines the current state of the microprocessor.

**IP register** always works together with CS segment register and it points to currently executing instruction.

**Flags register** is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.

Generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

**Variables**

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "var1" then at the address 5A73:235B, especially when you have 10 or more variables.

Our compiler supports two types of variables: **BYTE** and **WORD**.

Syntax for a variable declaration:

```
name DB value
```

```
name DW value
```

**DB** - stays for **Define Byte**.

**DW** - stays for **Define Word**.

**name** - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

**value** - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or ”?” symbol for variables that are not initialized.

**Arrays**

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

*b* is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:

```
```

```
  48  65  6C  6C  6F  00  48  65  6C  ...
```
You can access the value of any element in array using square brackets, for example:

```
MOV AL, a[3]
```

You can also use any of the memory index registers `BX, SI, DI, BP`, for example:

```
MOV SI, 3
MOV AL, a[SI]
```

If you need to declare a large array you can use `DUP` operator.

The syntax for `DUP`:

```
number DUP ( value(s) )
```

- `number` - number of duplicate to make (any constant value).
- `value` - expression that `DUP` will duplicate.

For example:

```
C DB 5 DUP(9)
```

This is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

One more example:

```
d DB 5 DUP(1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2
```

**Constants**

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants `EQU` directive is used:

```
name EQU < any expression >
```

For example:

```
k EQU 5
MOV AX, k
```
Review Question:

1. List types of data registers?
   1-
   2-
   3-

2. List types of Sections Registers?
   1-
   2-
   3-

3. What are the Differences between AH & AX?
Lab (6): EMU8086

Objectives

 dispon the DEBUG Program.

1. Start Emu8086 by selecting its icon from the start menu, or by running "Emu8086.exe".

After the application starts, the following popup window will be displayed. Select "Code Samples" to see examples or select “Continue” if you will write a new program.

2. From the tool bar select “New” icon.

3. From the following dialog box select “COM template”.

4. A new document file with COM template appears in the application window. Write down your program.

5. Click [Compile] then [Emulate] button (or press F5 hot key).

6. Click [Single Step] button (or press F8 hot key), and watch how the code is being executed.
Using the emulator

If you want to load your code into the emulator, just click "Emulate" button. But you can also use emulator to load executables even if you don't have the original source code. Select Show emulator from the Emulator menu.

Try loading files from "MyBuild" folder. If there are no files in "MyBuild" folder return to source editor, select Examples from File menu, load any sample, compile it and then load into the emulator:

[Single Step] button executes instructions one by one stopping after each instruction. [Run] button executes instructions one by one with delay set by step delay between instructions. Double click on register text-boxes opens "Extended Viewer" window with value of that register converted to all possible forms. You can modify the value of the register directly in this window. Double click on memory list item opens "Extended Viewer" with WORD value loaded from memory list at selected location. Less significant byte is at lower address: LOW BYTE is loaded from selected position and HIGH BYTE from next memory address. You can modify the value of the memory word directly in the "Extended Viewer" window.

You can modify the values of registers on runtime by typing over the existing values. [Flags] button allows you to view and modify flags on runtime.
Review Questions

1. Run the “Hello.asm” sample program and use the `REGISTER` command to display the current contents of all of the 80x86's internal registers. List the initial values held in CS, DS, and SS.
   CS = _____________, DS = ______________, SS = _____________

2. Write the following program that demonstrates the use of MOV instruction and trace it.

   #MAKE COM# ; instruct compiler to make COM file.
   ORG 100h ; directive required for a COM program.
   MOV AX, 0B800h ; set AX to hexadecimal value of B800h.
   MOV DS, AX ; copy value of AX to DS.
   MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.
   MOV CH, 01011111b ; set CH to binary value.
   MOV BX, 15Eh ; set BX to 15Eh.
   MOV [BX], CX ; copy contents of CX to memory at B800:015E
   RET ; returns to operating system.

<table>
<thead>
<tr>
<th>Step</th>
<th>AX</th>
<th>BX</th>
<th>CX</th>
<th>CH</th>
<th>CL</th>
<th>DS</th>
<th>B800:015E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lab (7): Memory Addressing and Instructions

Objectives

- Understand Memory Addressing Modes
- Learn how to use Memory access Instructions

Memory Addressing Modes

To access memory we can use these four registers: BX, SI, DI, BP, combining these registers inside [] symbols, we can get different memory locations. these combinations are supported (addressing modes):

<table>
<thead>
<tr>
<th>BX + SI</th>
<th>SI</th>
<th>BX + SI + d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX + DI</td>
<td>DI</td>
<td>BX + DI + d8</td>
</tr>
<tr>
<td>BP + SI</td>
<td>d16 (variable offset only)</td>
<td>BP + SI + d8</td>
</tr>
<tr>
<td>BP + DI</td>
<td>BX</td>
<td>BP + DI + d8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SI + d8</th>
<th>BX + SI + d16</th>
<th>SI + d16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI + d8</td>
<td>BX + DI + d16</td>
<td>DI + d16</td>
</tr>
<tr>
<td>BP + d8</td>
<td>BP + SI + d16</td>
<td>BP + d16</td>
</tr>
<tr>
<td>BX + d8</td>
<td>BP + DI + d16</td>
<td>BX + d16</td>
</tr>
</tbody>
</table>

- d8 - stays for 8 bit signed immediate displacement (for example: 22, 55h, -1, etc...)
- d16 - stays for 16 bit signed immediate displacement (for example: 300, 5517h, -259, etc...).

Displacement can be an immediate value or offset of a variable, or even both. if there are several values, assembler evaluates all values and calculates a single immediate value.

Displacement can be inside or outside of the [ ] symbols, assembler generates the same machine code for both ways.

Displacement is a signed value, so it can be either positive or negative. Generally the compiler takes care about difference between d8 and d16, and generates the required machine code.

Example: Let's assume that DS = 100, BX = 30, SI = 70.

The following addressing mode:

[BX + SI] + 25

is calculated by processor to this physical address: 100 * 16 + 30 + 70 + 25 = 1725.

By default DS segment register is used for all modes except those with BP register, for these SS segment register is used.

There is an easy way to remember all those possible combinations using this chart:

You can form all valid combinations by taking only one item from each column or skipping the column by not taking anything from it. As you see BX and BP never go together. SI and DI also don't go together. Here are examples of valid addressing modes:

[BX+5],  [BX+SI],  [DI+BX-4]
The value in segment register (CS, DS, SS, ES) is called a **segment**, and the value in purpose register (BX, SI, DI, BP) is called an **offset**. When DS contains value \(1234h\) and SI contains the value \(7890h\) it can be also recorded as \(1234:7890\).

The physical address will be \(1234h \times 10h + 7890h = 19BD0h\).

If zero is added to a decimal number it is multiplied by 10, however \(10h = 16\), so if zero is added to a hexadecimal value, it is multiplied by 16, for example:

\[
7h = 7 \\
70h = 112
\]

In order to say the compiler about data type, these prefixes should be used:

- **byte ptr** - for byte.
- **word ptr** - for word (two bytes).

For example:

```
byte ptr [BX] ; byte access.
or
word ptr [BX] ; word access.
```

In certain cases the assembler can calculate the data type automatically.

### Memory Access Instructions

1. **MOV instruction**

- **Copies the second operand (source) to the first operand (destination).**
- Source operand can be an immediate value, general-purpose register or memory location.
- Destination register can be a general-purpose register, or memory location.
- Both operands must be the same size, which can be a byte or a word.

These types of operands are supported:

- \(\text{MOV REG, memory}\)
- \(\text{MOV memory, REG}\)
- \(\text{MOV REG, REG}\)
- \(\text{MOV memory, immediate}\)
- \(\text{MOV REG, immediate}\)

**REG:** AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP

**Memory:** [BX], [BX+SI+7], variable, etc...

**Immediate:** 5, -24, 3Fh, 10001101b, etc...
For segment registers only these types of MOV are supported:
  - MOV SREG, memory
  - MOV memory, SREG
  - MOV REG, SREG
  - MOV SREG, REG

SREG: DS, ES, SS, and only as second operand: CS.
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
Memory: [BX], [BX+SI+7], variable, etc...

The MOV instruction cannot be used to set the value of the CS and IP registers.

Here is a short program that demonstrates the use of MOV instruction:

```assembly
ORG 100h ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h ; set AX to hexadecimal value of B800h.
MOV DS, AX ; copy value of AX to DS.
MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh ; set BX to 15Eh.
MOV [BX], CX ; copy contents of CX to memory at B800:015E
RET ; returns to operating system.
```

2. XCHG Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHG AX,r16</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>XCHG r16,AX</td>
<td>Exchange AX with r16.</td>
</tr>
<tr>
<td>XCHG EAX, r32</td>
<td>Exchange r32 with EAX.</td>
</tr>
<tr>
<td>XCHG r32, EAX</td>
<td>Exchange EAX with r32.</td>
</tr>
<tr>
<td>XCHG r/m8, r8</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>XCHG r8, r/m8</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
</tr>
<tr>
<td>XCHG r/m16, r16</td>
<td>Exchange r16 with word from r/m16.</td>
</tr>
<tr>
<td>XCHG r16, r/m16</td>
<td>Exchange word from r/m16 with r16.</td>
</tr>
<tr>
<td>XCHG r/m32, r32</td>
<td>Exchange r32 with doubleword from r/m32.</td>
</tr>
<tr>
<td>XCHG r32, r/m32</td>
<td>Exchange doubleword from r/m32 with r32.</td>
</tr>
</tbody>
</table>

XCHG exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation.
Review Questions:

1. Write the syntax of MOV, XCHG?

2. Explain the following statements?
   1- MOV AX, zword
   2- MOV AL, 'A'

3. Explain the following statements?
   1- XCHG AH, BL
   2- XCHG AX, ZWORD

4. Save the Value of A in B

5. Find the error and rewrite the statements?
   1- MOV DS, 1000
   2- MOV DS, ES

6. Implement the Assembly Language Program to find sum of numbers in the array. Use emu8086 emulator.

   **ALGORITHM:**
   1. Start.
   2. Initialize counter = 10.
   3. Initialize array pointer.
   4. Sum = 0.
   5. Get the array element pointed by array pointer.
   6. Add array element in the Sum.
   7. Increment array pointer decrement counter.
   8. Repeat steps 4, 5 & 6 until counter = 0.
   10. Stop.
PROGRAM:
include "emu8086.inc"
ORG 100
.data
SUM DW 0
.code
START: MOV AX, @DATA
MOV DS, AX
MOV CL, 10
XOR DI, DI
MOV BX, OFFSET ARRAY
BACK:   MOV AL, [BX+DI]
MOV AH, 00H
MOV SUM, AX
INC DI
DEC CL
JNZ BACK
; print out the number in ax
CALL print_num ; using procedure
RET
END
Lab (8): Arithmetic and Logic Instructions

Objectives
- Identify Arithmetic Instructions - ADD, SUB, CMP, INC, DEC, NOT, NEG
- Identify Assembly basic Instructions - MUL, DIV, IMUL, IDIV

Arithmetic and Logic Instructions
Most Arithmetic and Logic Instructions affect the processor status register (or Flags):

As you may see there are 16 bits in this register, each bit is called a flag and can take a value of 1 or 0.

- **Carry Flag (CF)** - this flag is set to 1 when there is an unsigned overflow. For example when you add bytes 255 + 1 (result is not in range 0...255). When there is no overflow this flag is set to 0.

- **Zero Flag (ZF)** - set to 1 when result is zero. For none zero result this flag is set to 0.

- **Sign Flag (SF)** - set to 1 when result is negative. When result is positive it is set to 0. Actually this flag take the value of the most significant bit.

- **Overflow Flag (OF)** - set to 1 when there is a signed overflow. For example, when you add bytes 100 + 50 (result is not in range -128...127).

- **Parity Flag (PF)** - this flag is set to 1 when there is even number of one bits in result, and to 0 when there is odd number of one bits. Even if result is a word only 8 low bits are analyzed!

- **Auxiliary Flag (AF)** - set to 1 when there is an unsigned overflow for low nibble (4 bits).

- **Interrupt enable Flag (IF)** - when this flag is set to 1 CPU reacts to interrupts from external devices.

- **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to 0 - the processing is done forward, when this flag is set to 1 the processing is done backward.
There are 3 groups of instructions.

1. First group: ADD, SUB, CMP, AND, TEST, OR, XOR
   These types of operands are supported:
   - REG, memory
   - memory, REG
   - REG, REG
   - memory, immediate
   - REG, immediate
   - REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
   - memory: [BX], [BX+SI+7], variable, etc...
   - immediate: 5, -24, 3Fh, 10001101b, etc...

   After operation between operands, result is always stored in first operand. CMP and TEST instructions affect flags only and do not store a result (these instructions are used to make decisions during program execution).
   These instructions affect these flags only: CF, ZF, SF, OF, PF, AF.
   - ADD - add second operand to first.
   - SUB - Subtract second operand to first.
   - CMP - Subtract second operand from first for flags only.
   - AND - Logical AND between all bits of two operands. These rules apply:
     1 AND 1 = 1
     1 AND 0 = 0
     0 AND 1 = 0
     0 AND 0 = 0
     As you see we get 1 only when both bits are 1.
   - TEST - The same as AND but for flags only.
   - OR - Logical OR between all bits of two operands. These rules apply:
     1 OR 1 = 1
     1 OR 0 = 1
     0 OR 1 = 1
     0 OR 0 = 0
     As you see we get 1 every time when at least one of the bits is 1.
   - XOR - Logical XOR (exclusive OR) between all bits of two operands. These rules apply:
     1 XOR 1 = 0
     1 XOR 0 = 1
     0 XOR 1 = 1
     0 XOR 0 = 0
     As you see we get 1 every time when bits are different from each other.
2. Second group: MUL, IMUL, DIV, IDIV

These types of operands are supported:
- REG
- memory
- REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- memory: [BX], [BX+SI+7], variable, etc...

MUL and IMUL instructions affect these flags only: CF, OF
When result is over operand size these flags are set to 1, when result fits in operand size these flags are set to 0.
For DIV and IDIV flags are undefined.

- **MUL** - Unsigned multiply:
  
  when operand is a **byte**:
  \[ AX = AL \times \text{operand}. \]
  when operand is a **word**:
  \[ (DX AX) = AX \times \text{operand}. \]

- **IMUL** - Signed multiply:
  
  when operand is a **byte**:
  \[ AX = AL \times \text{operand}. \]
  when operand is a **word**:
  \[ (DX AX) = AX \times \text{operand}. \]

- **DIV** - Unsigned divide:
  
  when operand is a **byte**:
  \[ AL = AX / \text{operand} \]
  \[ AH = \text{remainder (modulus)}. \]
  when operand is a **word**:
  \[ AX = (DX AX) / \text{operand} \]
  \[ DX = \text{remainder (modulus)}. \]

- **IDIV** - Signed divide:
  
  when operand is a **byte**:
  \[ AL = AX / \text{operand} \]
  \[ AH = \text{remainder (modulus)}. \]
  when operand is a **word**:
  \[ AX = (DX AX) / \text{operand} \]
  \[ DX = \text{remainder (modulus)}. \]
3. Third group: INC, DEC, NOT, NEG

These types of operands are supported:

- **REG**
- **Memory**
- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **memory**: [BX], [BX+SI+7], variable, etc...

**INC**, **DEC** instructions affect these flags only:

- **ZF**, **SF**, **OF**, **PF**, **AF**.

**NOT** instruction does not affect any flags!

**NEG** instruction affects these flags only:

- **CF**, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** - Reverse each bit of operand.

- **NEG** - Make operand negative (two's complement). Actually it reverses each bit of operand and then adds 1 to it. For example 5 will become -5, and -2 will become 2.

**Review Questions:**

1. Develop a short sequence that add AX, BX, CX, DX and SP, and Save the sum in DI
2. Write a program that adds a column of input values and produces the sum as output. The first input value will contain the number of values that follow as input to be added.
3. Write a program that solve the following equation:

\[ var4 = (var1 \times -7) + (var2 \mod var3) \]
Lab (9): Program flow control

Objectives

- Identify unconditional jump Instructions.
- Identify conditional jump Instructions.
- Identify loop Instructions.

Controlling the program flow is a very important thing; this is where your program can make decisions according to certain conditions.

Unconditional jumps

The basic instruction that transfers control to another point in the program is JMP. The basic syntax of JMP instruction:

```
JMP label
```

To declare a label in your program, just type its name and add ";:" to the end, label can be any character combination but it cannot start with a number, for example here are 3 legal label definitions:

```
label1:
label2:
a:
```

Label can be declared on a separate line or before any other instruction, for example:

```
x1:
    MOV AX, 1

x2: MOV AX, 2
```

Here's an example of JMP instruction:

```
org 100h

mov ax, 5 ; set ax to 5.
mov bx, 2 ; set bx to 2.
jmp calc ; go to 'calc'.
back: jmp stop ; go to 'stop'.
calc:
    add ax, bx ; add bx to ax.
jmp back ; go 'back'.
stop: ret ; return to operating system.
```

Of course there is an easier way to calculate the sum of two numbers, but it's still a good example of JMP instruction. As you can see from this example JMP is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).
Short Conditional Jumps
Unlike *JMP* instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

1. Jump instructions that test single flag

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
<th>Opposite Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ, JE</td>
<td>Jump if Zero (Equal).</td>
<td>ZF = 1</td>
<td>JNZ, JNE</td>
</tr>
<tr>
<td>JC, JB, JNAE</td>
<td>Jump if Carry (Below, Not Above Equal).</td>
<td>CF = 1</td>
<td>JNC, JNB, JAE</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if Sign.</td>
<td>SF = 1</td>
<td>JNS</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if Overflow.</td>
<td>OF = 1</td>
<td>JNO</td>
</tr>
<tr>
<td>JPE, JP</td>
<td>Jump if Parity Even.</td>
<td>PF = 1</td>
<td>JPO</td>
</tr>
<tr>
<td>JNZ, JNE</td>
<td>Jump if Not Zero (Not Equal).</td>
<td>ZF = 0</td>
<td>JZ, JE</td>
</tr>
<tr>
<td>JNC, JNB, JAE</td>
<td>Jump if Not Carry (Not Below, Above Equal).</td>
<td>CF = 0</td>
<td>JC, JB, JNAE</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if Not Sign.</td>
<td>SF = 0</td>
<td>JS</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if Not Overflow.</td>
<td>OF = 0</td>
<td>JO</td>
</tr>
<tr>
<td>JPO, JNP</td>
<td>Jump if Parity Odd (No Parity).</td>
<td>PF = 0</td>
<td>JPE, JP</td>
</tr>
</tbody>
</table>

As you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile *JE* instruction - you will get it disassembled as: *JZ, JC* is assembled the same as *JB* etc... Different names are used to make programs easier to understand, to code and most importantly to remember. Very offset disassembler has no clue what the original instruction was look like that's why it uses the most common name.

If you emulate this code you will see that all instructions are assembled into *JNB*, the operational code (opcode) for this instruction is *73h* this instruction has fixed length of two bytes, the second byte is number of bytes to add to the *IP* register if the condition is true. Because the instruction has only 1 byte to keep the offset it is limited to pass control to -128 bytes back or 127 bytes forward, this value is always signed.

```
jnc a
jnb a
jae a

mov ax, 4
a: mov ax, 5
ret
```
## 2. Jump instructions for signed numbers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
<th>Opposite Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE, JZ</td>
<td>Jump if Equal (=). Jump if Zero.</td>
<td>ZF = 1</td>
<td>JNE, JNZ</td>
</tr>
<tr>
<td>JNE, JNZ</td>
<td>Jump if Not Equal (&lt;&gt;). Jump if Not Zero.</td>
<td>ZF = 0</td>
<td>JE, JZ</td>
</tr>
<tr>
<td>JG, JNLE</td>
<td>Jump if Greater (&gt;). Jump if Not Less or Equal (not &lt;=).</td>
<td>ZF = 0 and SF = OF</td>
<td>JNG, JLE</td>
</tr>
<tr>
<td>JL, JNGE</td>
<td>Jump if Less (&lt;). Jump if Not Greater or Equal (not &gt;=).</td>
<td>SF &lt;&gt; OF</td>
<td>JNL, JGE</td>
</tr>
<tr>
<td>JGE, JNL</td>
<td>Jump if Greater or Equal (&gt;=). Jump if Not Less (not &lt;).</td>
<td>SF = OF</td>
<td>JNGE, JL</td>
</tr>
<tr>
<td>JLE, JNG</td>
<td>Jump if Less or Equal (&lt;=). Jump if Not Greater (not &gt;).</td>
<td>ZF = 1 or SF &lt;&gt; OF</td>
<td>JNLE, JG</td>
</tr>
</tbody>
</table>

<> - sign means not equal.

## 3. Jump instructions for unsigned numbers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
<th>Opposite Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE, JZ</td>
<td>Jump if Equal (=) Jump if Zero.</td>
<td>ZF = 1</td>
<td>JNE, JNZ</td>
</tr>
<tr>
<td>JNE, JNZ</td>
<td>Jump if Not Equal (&lt;&gt;). Jump if Not Zero.</td>
<td>ZF = 0</td>
<td>JE, JZ</td>
</tr>
<tr>
<td>JA, JNBE</td>
<td>Jump if Above (&gt;). Jump if Not Below or Equal (not &lt;=).</td>
<td>CF = 0 and ZF = 0</td>
<td>JNA, JBE</td>
</tr>
<tr>
<td>JB, JNAE, JC</td>
<td>Jump if Below (&lt;). Jump if Not Above or Equal (not &gt;=). Jump if Carry.</td>
<td>CF = 1</td>
<td>JNB, JAE, JNC</td>
</tr>
<tr>
<td>JAE, JNB, JNC</td>
<td>Jump if Above or Equal (&gt;=). Jump if Not Below (not &lt;). Jump if Carry.</td>
<td>CF = 0</td>
<td>JNAE, JB</td>
</tr>
<tr>
<td>JBE, JNA</td>
<td>Jump if Below or Equal (&lt;=). Jump if Not Above (not &gt;).</td>
<td>CF = 1 or ZF = 1</td>
<td>JNBE, JA</td>
</tr>
</tbody>
</table>

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).
The logic is very simple, for example:

It's required to compare 5 and 2,
5 - 2 = 3
The result is not zero (Zero Flag is set to 0).

Another example:

It's required to compare 7 and 7,
7 - 7 = 0
The result is zero! (Zero Flag is set to 1 and JZ or JE will do the jump).

Here's an example of CMP instruction and conditional jump:

```
include "emu8086.inc"

org 100h

mov al, 25 ; set al to 25.
mov bl, 10 ; set bl to 10.
cmp al, bl ; compare al - bl.
je equal ; jump if al = bl (zf = 1).
putc 'n' ; if it gets here, then al <> bl,
jmp stop ; so print 'n', and jump to stop.

equal:
putc 'y' ; if gets here,
then al = bl, so print 'y'.

stop:
ret ; gets here no matter what
```

Try the above example with different numbers for AL and BL, open flags by clicking on flags button, use single step and see what happens. You can use F5 hotkey to recompile and reload the program into the emulator.

### Loops

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation and jump condition</th>
<th>Opposite instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>Decrease CX, jump to label if CX not zero.</td>
<td>DEC CX and JCXZ</td>
</tr>
<tr>
<td>LOOPE</td>
<td>Decrease CX, jump to label if CX not zero and equal (zf = 1).</td>
<td>LOOPNE</td>
</tr>
<tr>
<td>LOOPNE</td>
<td>Decrease CX, jump to label if CX not zero and not equal (zf = 0).</td>
<td>LOOPE</td>
</tr>
<tr>
<td>LOOPNZ</td>
<td>Decrease CX, jump to label if CX not zero and zf = 0.</td>
<td>LOOPZ</td>
</tr>
<tr>
<td>LOOPZ</td>
<td>Decrease CX, jump to label if CX not zero and zf = 1.</td>
<td>LOOPNZ</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump to label if CX is zero.</td>
<td>OR CX, CX and JNZ</td>
</tr>
</tbody>
</table>
Loops are basically the same jumps, it is possible to code loops without using the loop instruction, by just using conditional jumps and compare, and this is just what loop does. All loop instructions use CX register to count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF, however with some agility it is possible to put one loop into another, and another into another two, and three and etc... and receive a nice value of 65535 * 65535 * 65535 ....till infinity.... or the end of ram or stack memory. It is possible store original value of CX register using push cx instruction and return it to original when the internal loop ends with pop cx, for example:

```
org 100h

    mov bx, 0      ; total step counter.

k1:  add bx, 1
    mov al, '1'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5

    add bx, 1
    mov al, '2'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5

k3:  add bx, 1
    mov al, '3'
    mov ah, 0eh
    int 10h
    loop k3       ; internal in internal loop.
   pop cx
    loop k2       ; internal loop.
   pop cx
    loop k1       ; external loop.
                   ; wait any key...
    mov ah, 1
    int 21h

ret
```

BX counts total number of steps, by default emulator shows values in hexadecimal, you can double click the register to see the value in all available bases.

All conditional jumps have one big limitation, unlike JMP instruction they can only jump 127 bytes forward and 128 bytes backward (note that most instructions are assembled into 3 or more bytes).
Review Questions

1. Write a program that prints out the even numbers from 1 to 99. No input is required.
2. Write a program that accepts three values as input and outputs them in order of size, largest to smallest.
Lab (10): Procedures, Interrupts and Stack

Objectives

❖ Learn how to execute Assembly instructions and observe the results

Procedures

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC
 ; here goes the code
 ; of the procedure ...
RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that RET instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and ENDP are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Here is an example:

```
ORG    100h
CALL   m1
MOV    AX, 2
RET                   ; return to operating system.

m1    PROC
MOV    BX, 5
RET                   ; return to caller.

m1    ENDP

END
```

The above example calls procedure m1, does MOV BX, 5, and returns to the next instruction after CALL: MOV AX, 2.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in AL and BL registers, multiplies these parameters and returns the result in AX register:
In the above example value of AL register is updated every time the procedure is called, BL register stays unchanged, so this algorithm calculates 2 in power of 4, so final result in AX register is 16 (or 10h).

Here goes another example,

That uses a procedure to print a *Hello World!* message:

```
ORG 100h

LEA SI, msg ; load address of msg to SI.
CALL print_me
RET ; return to operating system.

; this procedure prints a string, the string should be null
; terminated (have zero in the end),
; the string address should be in SI register:
print_me PROC

next_char:
CMP b.[SI], 0 ; check for zero to stop
JE stop ;
MOV AL, [SI] ; next get ASCII char.
MOV AH, 0EH ; teletype function number.
INT 10h ; using interrupt to print a char in AL.
ADD SI, 1 ; advance index of string array.
JMP next_char ; go back, and type another char.
stop:
RET ; return to caller.
print_me ENDP

msg DB 'Hello World!', 0 ; null terminated string.
END
```

"b." - prefix before [SI] means that we need to compare bytes, not words. When you need to compare words add "w." prefix instead. When one of the compared operands is a register it's not required because compiler knows the size of each register.
Interrupts

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware. We call such functions **software interrupts**.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To make software **interrupt** there is an **INT** instruction, it has very simple syntax:

```
INT value
```

Where **value** can be a number between 0 to 255 (or 0 to 0FFh). Generally we will use hexadecimal numbers. You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions.

To specify a sub-function **AH** register should be set before calling interrupt. Each interrupt may have up to 256 sub-functions (so we get 256 * 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This function displays a character on the screen, advancing the cursor and scrolling the screen as necessary.

---

Example – Custom interrupt

```asm
; interrupt vector (memory from 00000h to 00400h)
; keeps addresses of all interrupts (from 00h to 0ffh).
; you can add new interrupt or modify existing interrupts.
; address of interrupt M is stored in vector at offset M * 4,
; for example: interrupt 10h is stored at offset 10h * 4,
; first goes the offset, then segment (total of 2 bytes).

name "custint"

org 100h

start:

    ; set video mode to 3 - 16 color 80x25
    mov  ah, 0
    mov  al, 3
    int  10h
    ; set es to "0000":
    mov  ax, 0
    mov  es, ax

    mov  bx, 90h     ; calculate vector address for interrupt 90h:
    shl  bx, 2       ; multiply 90h by 4, store result in bx:
    mov  si, offset [test1] ; copy offset into interrupt vector:
    mov  es:[bx], si
    add  bx, 2

    mov  ax, cs     ; copy segment into interrupt vector:
    mov  es:[bx], ax
```

---
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int 90h</code></td>
<td>; test newly created interrupt.</td>
</tr>
<tr>
<td><code>mov ah, 0</code></td>
<td>; wait for any key press:</td>
</tr>
<tr>
<td><code>int 16h</code></td>
<td>; halt execution.</td>
</tr>
<tr>
<td><code>int 20h</code></td>
<td>; interrupt 90h starts here:</td>
</tr>
</tbody>
</table>

### `test1`:

- `pusha` ; store all registers.
- `push cs`
- `pop ds` ; make sure data segment is code segment:
- `mov ax, 0b800h`
- `mov es, ax` ; set segment register to video memory:
- `lea si, msg` ; load offset of msg to si.
- `mov di, 0` ; point to start of the screen.
- `cmp [si], 0` ; if "0" then stop.
- `je stop` ; read ascii code from msg.
- `mov bl, [si]`
- `mov bh, 0f1h` ; set colors: white background, blue text.
- `mov es:[di], bx` ; write to video memory.
- `add di, 2` ; go to next position on screen.
- `inc si` ; next char.
- `jmp print`

### `stop`:

- `popa` ; re-store all registers.
- `iret` ; return from interrupt.

`msg db "test of custom interrupt!", 0`
The Stack
Stack is an area of memory for keeping temporary data. Stack is used by CALL instruction to keep return address for procedure, RET instruction gets this value from the stack and returns to that offset. Quite the same thing happens when INT instruction calls an interrupt, it stores in stack flag register, code segment and offset. IRET instruction is used to return from interrupt call.

We can also use the stack to keep any other data; there are two instructions that work with the stack:
- **PUSH** - stores 16 bit value in the stack.
- **POP** - gets 16 bit value from the stack.

Syntax for **PUSH** instruction:

- **PUSH REG**
- **PUSH SREG**
- **PUSH memory**
- **PUSH immediate**

REG: AX, BX, CX, DX, DI, SI, BP, SP
SREG: DS, ES, SS, CS.
memory: [BX], [BX+SI+7], 16 bit variable, etc...
immediate: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

- **POP REG**
- **POP SREG**
- **POP memory**

REG: AX, BX, CX, DX, DI, SI, BP, SP.
SREG: DS, ES, SS, (except CS).
memory: [BX], [BX+SI+7], 16 bit variable, etc...

**Notes:**
**PUSH** and **POP** work with 16 bit values only!
The stack uses **LIFO** (Last In First Out) algorithm, this means that if we push these values one by one into the stack:
1, 2, 3, 4, 5
The first value that we will get on pop will be 5, then 4, 3, 2, and only then 1.
It is very important to do equal number of **PUSH**s and **POPs**, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).
PUSH and POP instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using PUSH).
- Use the register for any purpose.
- Restore the original value of the register from stack (using POP).

Here is an example:

```assembly
ORG 100h
MOV AX, 1234h
PUSH AX ; store value of AX in stack.
MOV AX, 5678h ; modify the AX value.
POP AX ; restore the original value of AX.
RET
END
```

Another use of the stack is for exchanging the values, here is an example:

```assembly
ORG 100h
MOV AX, 1212h ; store 1212h in AX.
MOV BX, 3434h ; store 3434h in BX
PUSH AX ; store value of AX in stack.
PUSH BX ; store value of BX in stack.
POP AX ; set AX to original value of BX.
POP BX ; set BX to original value of AX.
RET
END
```

The exchange happens because stack uses LIFO (Last In First Out) algorithm, so when we push 1212h and then 3434h, on pop we will first get 3434h and only after it 1212h.

The stack memory area is set by SS (Stack Segment) register, and SP (Stack Pointer) register. Generally, operating system sets values of these registers on program start.

"PUSH source" instruction does the following:

- Subtract 2 from SP register.
- Write the value of source to the address SS:SP.

"POP destination" instruction does the following:

- Write the value at the address SS:SP to destination.
• Add 2 to SP register.

The current address pointed by SS:SP is called the **top of the stack**.

### Review Question

1. Write a procedure “WriteInt”, that receives a 16-bit signed integer in AX, converts it to string and print it on screen using DOS functions.
2. Write a procedure “ReadInt”, that receives a string from the keyboard and convert it to signed integer returned in AX register.
3. Write a program to evaluate the following arithmetic statement using a stack organized computer with zero-address instructions (so only pop and push can access memory).

\[
X = \frac{A - B + C \times H \times (D - F)}{E \times G + K}
\]
## Final Project

Write an assembly language program to simulate the traffic light system implemented in Tabuk at 4-ways cross road. Your system must allow manual override, and online change of delays.

**Time allowed:** 2 weeks

**Group:** Maximum two students.

*All Works must be conducted in the lab.*

**Software:** EMU8086 emulator
Appendix (A)
Why Binary?

- Early computer design was decimal
  - Mark I and ENIAC
- John von Neumann proposed binary data processing (1945)
  - Simplified computer design
  - Used for both instructions and data
- Natural relationship between on/off switches and calculation using Boolean logic

<table>
<thead>
<tr>
<th>On</th>
<th>Off</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Chapter 2 Number Systems
Counting and Arithmetic

- Decimal or base 10 number system
  - Origin: counting on the fingers
  - “Digit” from the Latin word *digitus* meaning “finger”
  - *Base*: the number of different digits including zero in the number system
  - Example: Base 10 has 10 digits, 0 through 9
- *Binary* or *base 2*
- *Bit* (binary digit): 2 digits, 0 and 1
- *Octal* or *base 8*: 8 digits, 0 through 7
- *Hexadecimal* or *base 16*
  - 16 digits, 0 through F
  - Examples: $10_{10} = A_{16}$, $11_{10} = B_{16}$

Keeping Track of the Bits

- Bits commonly stored and manipulated in groups
  - 8 bits = 1 *byte*
  - 4 bytes = 1 word (in many systems)
- Number of bits used in calculations
  - Affects accuracy of results
  - Limits size of numbers manipulated by the computer
Numbers: Physical Representation

- Different numerals, same number of oranges
  - Cave dweller: I II I I
  - Roman: V
  - Arabic: 5

- Different bases, same number of oranges
  - \(5_{10}\)
  - \(101_2\)
  - \(12_3\)

Number System

- Roman: position independent
- Modern: based on positional notation (place value)
  - Decimal system: system of positional notation based on powers of 10.
  - Binary system: system of positional notation based on powers of 2
  - Octal system: system of positional notation based on powers of 8
  - Hexadecimal system: system of positional notation based on powers of 16
**Positional Notation: Base 10**

43 = 4 x 10^1 + 3 x 10^0

<table>
<thead>
<tr>
<th>Place</th>
<th>10^1</th>
<th>10^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Evaluate</td>
<td>4 x 10</td>
<td>3 x 1</td>
</tr>
<tr>
<td>Sum</td>
<td>40</td>
<td>3</td>
</tr>
</tbody>
</table>

527 = 5 x 10^2 + 2 x 10^1 + 7 x 10^0

<table>
<thead>
<tr>
<th>Place</th>
<th>10^2</th>
<th>10^1</th>
<th>10^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>100</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Evaluate</td>
<td>5 x 100</td>
<td>2 x 10</td>
<td>7 x 1</td>
</tr>
<tr>
<td>Sum</td>
<td>500</td>
<td>20</td>
<td>7</td>
</tr>
</tbody>
</table>
### Positional Notation: Octal

\[ 624_8 = 404_{10} \]

<table>
<thead>
<tr>
<th>Place</th>
<th>Value</th>
<th>Evaluate</th>
<th>Sum for Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8^2)</td>
<td>64</td>
<td>6 \times 64</td>
<td>384</td>
</tr>
<tr>
<td>(8^1)</td>
<td>8</td>
<td>2 \times 8</td>
<td>16</td>
</tr>
<tr>
<td>(8^0)</td>
<td>1</td>
<td>4 \times 1</td>
<td>4</td>
</tr>
</tbody>
</table>

### Positional Notation: Hexadecimal

\[ 6,704_{16} = 26,372_{10} \]

<table>
<thead>
<tr>
<th>Place</th>
<th>Value</th>
<th>Evaluate</th>
<th>Sum for Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16^3)</td>
<td>4,096</td>
<td>6 \times 4,096</td>
<td>24,576</td>
</tr>
<tr>
<td>(16^2)</td>
<td>256</td>
<td>7 \times 256</td>
<td>1,792</td>
</tr>
<tr>
<td>(16^1)</td>
<td>16</td>
<td>0 \times 16</td>
<td>0</td>
</tr>
<tr>
<td>(16^0)</td>
<td>1</td>
<td>4 \times 1</td>
<td>4</td>
</tr>
</tbody>
</table>
### Positional Notation: Binary

1101 0110<sub>2</sub> = 214<sub>10</sub>

<table>
<thead>
<tr>
<th>Place</th>
<th>2&lt;sup&gt;7&lt;/sup&gt;</th>
<th>2&lt;sup&gt;6&lt;/sup&gt;</th>
<th>2&lt;sup&gt;5&lt;/sup&gt;</th>
<th>2&lt;sup&gt;4&lt;/sup&gt;</th>
<th>2&lt;sup&gt;3&lt;/sup&gt;</th>
<th>2&lt;sup&gt;2&lt;/sup&gt;</th>
<th>2&lt;sup&gt;1&lt;/sup&gt;</th>
<th>2&lt;sup&gt;0&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Evaluate</td>
<td>1 x 128</td>
<td>1 x 64</td>
<td>0 x 32</td>
<td>1 x 16</td>
<td>0 x 8</td>
<td>1 x 4</td>
<td>1 x 2</td>
<td>0 x 1</td>
</tr>
<tr>
<td>Sum for Base 10</td>
<td>128</td>
<td>64</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

### Estimating Magnitude: Binary

1101 0110<sub>2</sub> = 214<sub>10</sub>

1101 0110<sub>2</sub> > 192<sub>10</sub> (128 + 64 + additional bits to the right)

<table>
<thead>
<tr>
<th>Place</th>
<th>2&lt;sup&gt;7&lt;/sup&gt;</th>
<th>2&lt;sup&gt;6&lt;/sup&gt;</th>
<th>2&lt;sup&gt;5&lt;/sup&gt;</th>
<th>2&lt;sup&gt;4&lt;/sup&gt;</th>
<th>2&lt;sup&gt;3&lt;/sup&gt;</th>
<th>2&lt;sup&gt;2&lt;/sup&gt;</th>
<th>2&lt;sup&gt;1&lt;/sup&gt;</th>
<th>2&lt;sup&gt;0&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Evaluate</td>
<td>1 x 128</td>
<td>1 x 64</td>
<td>0 x 32</td>
<td>1 x 16</td>
<td>0 x 8</td>
<td>1 x 4</td>
<td>1 x 2</td>
<td>0 x 1</td>
</tr>
<tr>
<td>Sum for Base 10</td>
<td>128</td>
<td>64</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Range of Possible Numbers

- $R = B^K$ where
  - $R$ = range
  - $B$ = base
  - $K$ = number of digits

- **Example #1:** Base 10, 2 digits
  - $R = 10^2 = 100$ different numbers (0…99)

- **Example #2:** Base 2, 16 digits
  - $R = 2^{16} = 65,536$ or 64K
  - 16-bit PC can store 65,536 different number values

Decimal Range for Bit Widths

<table>
<thead>
<tr>
<th>Bits</th>
<th>Digits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0+</td>
<td>2 (0 and 1)</td>
</tr>
<tr>
<td>4</td>
<td>1+</td>
<td>16 (0 to 15)</td>
</tr>
<tr>
<td>8</td>
<td>2+</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1,024 (1K)</td>
</tr>
<tr>
<td>16</td>
<td>4+</td>
<td>65,536 (64K)</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>1,048,576 (1M)</td>
</tr>
<tr>
<td>32</td>
<td>9+</td>
<td>4,294,967,296 (4G)</td>
</tr>
<tr>
<td>64</td>
<td>19+</td>
<td>Approx. $1.6 \times 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>38+</td>
<td>Approx. $2.6 \times 10^{38}$</td>
</tr>
</tbody>
</table>
### Base or Radix

- **Base:**
  - The number of different symbols required to represent any given number
  - The *larger* the base, the *more* numerals are required
  - Base 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2: 0, 1
  - Base 8: 0, 1, 2, 3, 4, 5, 6, 7
  - Base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

### Number of Symbols vs. Number of Digits

- For a given number, the *larger* the base
  - the *more* symbols required
  - but the *fewer* digits needed
- **Example #1:**
  - $65_{16}$  10110  1458  110 01012
- **Example #2:**
  - $11C_{16}$  28410  4348  1 0001 11002
Counting in Base 2

<table>
<thead>
<tr>
<th>Binary Number</th>
<th>Equivalent</th>
<th>Decimal Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 x 2¹</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1 x 2¹</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1 x 2¹ + 0 x 2³</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1 x 2¹ + 1 x 2³</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>1 x 2²</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>1 x 2² + 1 x 2³</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>1 x 2² + 1 x 2¹</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>1 x 2² + 1 x 2¹ + 1 x 2³</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>1 x 2³</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>1 x 2³ + 1 x 2²</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>1 x 2³ + 1 x 2¹</td>
<td>10</td>
</tr>
</tbody>
</table>

Base 10 Addition Table

\[3_{10} + 6_{10} = 9_{10}\]
### Base 8 Addition Table

$$3_8 + 6_8 = 11_8$$

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

(no 8 or 9, of course)

---

### Base 10 Multiplication Table

$$3_{10} \times 6_{10} = 18_{10}$$

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

etc.
Base 8 Multiplication Table

\[
3_8 \times 6_8 = 22_8
\]

Addition

<table>
<thead>
<tr>
<th>Base</th>
<th>Problem</th>
<th>Largest Single Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>6 + 3</td>
<td>9</td>
</tr>
<tr>
<td>Octal</td>
<td>6 + 1</td>
<td>7</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>6 + 9</td>
<td>F</td>
</tr>
<tr>
<td>Binary</td>
<td>1 + 0</td>
<td>1</td>
</tr>
</tbody>
</table>
### Addition

<table>
<thead>
<tr>
<th>Base</th>
<th>Problem</th>
<th>Carry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>6 + 4</td>
<td>Carry the 10</td>
<td>10</td>
</tr>
<tr>
<td>Octal</td>
<td>6 + 2</td>
<td>Carry the 8</td>
<td>10</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>6 + A</td>
<td>Carry the 16</td>
<td>10</td>
</tr>
<tr>
<td>Binary</td>
<td>1 + 1</td>
<td>Carry the 2</td>
<td>10</td>
</tr>
</tbody>
</table>

### Binary Arithmetic

```
  1 1 1 1 1
+ 1 1 0 1 1 0 1
  ___________
  1 0 0 0 0 1 1 1 1
```
Binary Arithmetic

- **Addition**
  - Boolean using XOR and AND
    
    \[
    \begin{array}{c|c|c}
    + & 0 & 1 \\
    \hline
    0 & 0 & 1 \\
    \end{array}
    \]
  - **Boolean Logic**
    - **EXCLUSIVE-OR**
      - Output is “1” only if either input, but **not** both inputs, is a “1”
    - **AND (carry bit)**
      - Output is “1” if and only both inputs are a “1”

- **Multiplication**
  - **AND**
    
    \[
    \begin{array}{c|c|c}
    & 1 & 1 \\
    \hline
    x & 0 & 1 \\
    \end{array}
    \]
  - **Shift**
    
    \[
    \begin{array}{c|c|c}
    & 0 & 0 \\
    \hline
    1 & 0 & 1 \\
    \end{array}
    \]

- **Division**
  
  \[
  \begin{array}{c|c|c|c|c}
  1 & 0 & 0 & 0 & 0 \\
  \hline
  + & 1 & 1 & 1 & 0 \\
  \hline
  1 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]
Binary Multiplication

- Boolean logic without performing arithmetic
  - AND (carry bit)
    - Output is “1” if and only both inputs are a “1”
  - Shift
    - Shifting a number in any base left one digit multiplies its value by the base
    - Shifting a number in any base right one digit divides its value by the base
    - Examples:
      - $10_{10}$ shift left $= 100_{10}$
      - $10_{10}$ shift right $= 1_{10}$
      - $10_{2}$ shift left $= 100_{2}$
      - $10_{2}$ shift right $= 1_{2}$

Binary Multiplication

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & \text{1’s place} \\
0 & \text{2’s place} \\
1 & 1 & 0 & 1 & \text{4’s place (bits shifted to line up with 4’s place of multiplier)} \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1 & \text{Result (AND)}
\end{array}
\]
### Binary Multiplication

\[
\begin{array}{cccccccc}
& 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\times & 1 & 0 & 0 & 1 & 1 & 0 \\
\hline
& 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

2’s place (bits shifted to line up with 2’s place of multiplier)

\[
\begin{array}{cccccccc}
& 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\times & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
& 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

4’s place

\[
\begin{array}{cccccccc}
& 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\times & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Result (AND)

Note the 0 at the end, since the 1’s place is not brought down.

Note: multiple carries are possible.

### Converting from Base 10

#### Powers Table

<table>
<thead>
<tr>
<th>Power Base</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>32,768</td>
<td>4,096</td>
<td>512</td>
<td>64</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>65,536</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 2 Number Systems
From Base 10 to Base 2

42\textsubscript{10} = 101010\textsubscript{2}

<table>
<thead>
<tr>
<th>Power</th>
<th>Base</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>42/32</td>
<td>= 1</td>
<td>10/16</td>
<td>= 0</td>
<td>10/8</td>
<td>= 1</td>
<td>2/4</td>
<td>= 0</td>
</tr>
<tr>
<td>Remainder</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Base 10 to Base 2

Base 10 42

Quotient 2 ) 42 ( 0 Least significant bit
2 ) 21 ( 1
2 ) 10 ( 0
2 ) 5 ( 1
2 ) 2 ( 0
2 ) 1 ( Most significant bit

Base 2 101010
From Base 10 to Base 16

5,735\textsubscript{10} = 1667\textsubscript{16}

<table>
<thead>
<tr>
<th>Power</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Integer
- 5,735 / 4,096 = 1
- 1,639 / 256 = 6
- 103 / 16 = 6

Remainder
- 5,735 - 4,096 = 1,639
- 1,639 - 1,536 = 103
- 103 - 96 = 7

From Base 10 to Base 16

Base 10 5,735

Quotient
- 16 ) 5,735 ( 7  Least significant bit
- 16 ) 358 ( 6
- 16 ) 22 ( 6
- 16 ) 1 ( 1  Most significant bit
- 16 ) 0

Base 16 1667
**From Base 10 to Base 16**

Base 10 8,039

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
<th>Least significant bit</th>
<th>Most significant bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 ) 8,039</td>
<td>7</td>
<td>16 ) 502</td>
<td>6</td>
</tr>
</tbody>
</table>

Base 16 1F67

---

**From Base 8 to Base 10**

72638 = 3,76310

<table>
<thead>
<tr>
<th>Power</th>
<th>8^3</th>
<th>8^2</th>
<th>8^1</th>
<th>8^0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>512</td>
<td>64</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>x 7</td>
<td>x 2</td>
<td>x 6</td>
<td>x 3</td>
</tr>
<tr>
<td>Sum for Base 10</td>
<td>3,584</td>
<td>128</td>
<td>48</td>
<td>3</td>
</tr>
</tbody>
</table>
From Base 8 to Base 10

7263\textsubscript{8} = 3763\textsubscript{10}

7 \times 8 = 56 + 2 = 58

\begin{align*}
56 \times 8 & = 464 + 6 = 470 \\
470 \times 8 & = 3760 + 3 = 3763
\end{align*}

From Base 16 to Base 2

- The nibble approach
  - Hex easier to read and write than binary

<table>
<thead>
<tr>
<th>Base 16</th>
<th>1</th>
<th>F</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 2</td>
<td>0001</td>
<td>1111</td>
<td>0110</td>
<td>0111</td>
</tr>
</tbody>
</table>

- Why hexadecimal?
  - Modern computer operating systems and networks present variety of troubleshooting data in hex format
Fractions

- Number point or radix point
  - Decimal point in base 10
  - Binary point in base 2
- No exact relationship between fractional numbers in different number bases
  - Exact conversion may be impossible

Decimal Fractions

- Move the number point one place to the right
  - Effect: multiplies the number by the base number
  - Example: $139.0_{10} \rightarrow 1390_{10}$
- Move the number point one place to the left
  - Effect: divides the number by the base number
  - Example: $139.0_{10} \rightarrow 13.9_{10}$
Fractions: Base 10 and Base 2

### .2589\textsubscript{10}

<table>
<thead>
<tr>
<th>Place</th>
<th>10\textsuperscript{-1}</th>
<th>10\textsuperscript{-2}</th>
<th>10\textsuperscript{-3}</th>
<th>10\textsuperscript{-4}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1/10</td>
<td>1/100</td>
<td>1/1000</td>
<td>1/10000</td>
</tr>
<tr>
<td>Evaluate</td>
<td>2 x 1/10</td>
<td>5 x 1/100</td>
<td>8 x 1/1000</td>
<td>9 x 1/10000</td>
</tr>
<tr>
<td>Sum</td>
<td>.2</td>
<td>.05</td>
<td>.008</td>
<td>.0009</td>
</tr>
</tbody>
</table>

\[ .101011\textsubscript{2} = 0.671875\textsubscript{10} \]

<table>
<thead>
<tr>
<th>Place</th>
<th>2\textsuperscript{-1}</th>
<th>2\textsuperscript{-2}</th>
<th>2\textsuperscript{-3}</th>
<th>2\textsuperscript{-4}</th>
<th>2\textsuperscript{-5}</th>
<th>2\textsuperscript{-6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/32</td>
<td>1/64</td>
</tr>
<tr>
<td>Evaluate</td>
<td>1 x 1/2</td>
<td>0 x 1/4</td>
<td>1x 1/8</td>
<td>0 x 1/16</td>
<td>1 x 1/32</td>
<td>1 x 1/64</td>
</tr>
<tr>
<td>Sum</td>
<td>.5</td>
<td>0.125</td>
<td>0.03125</td>
<td>0.015625</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Fractions: Base 10 and Base 2

- No general relationship between fractions of types 1/10\textsuperscript{k} and 1/2\textsuperscript{k}
  - Therefore a number representable in base 10 may not be representable in base 2
  - But: the converse is true: all fractions of the form 1/2\textsuperscript{k} can be represented in base 10

- Fractional conversions from one base to another are stopped
  - If there is a rational solution or
  - When the desired accuracy is attained
Mixed Number Conversion

- Integer and fraction parts must be converted separately
- Radix point: fixed reference for the conversion
  - Digit to the left is a unit digit in every base
  - \( B^0 \) is always 1 regardless of the base

Copyright 2003 John Wiley & Sons

All rights reserved. Reproduction or translation of this work beyond that permitted in Section 117 of the 1976 United States Copyright Act without express permission of the copyright owner is unlawful. Request for further information should be addressed to the permissions Department, John Wiley & Sons, Inc. The purchaser may make back-up copies for his/her own use only and not for distribution or resale. The Publisher assumes no responsibility for errors, omissions, or damages caused by the use of these programs or from the use of the information contained herein.”
Appendix (B)
### TRANSFER Flags

| Name     | Comment                  | Code       | Operation                                                                 | O | D | I | T | S | Z | A | P | C |
|----------|--------------------------|------------|---------------------------------------------------------------------------|---|---|---|---|---|---|---|---|---|---|
| MOV      | Move (copy)              | MOV        | Dest:=Source                                                              |   |   |   |   |   |   |   |   |   |   |
| STC      | Set Carry                | STC        | CF:=1                                                                     |   |   |   |   |   |   |   |   |   |   |
| CLC      | Clear Carry              | CLC        | CF:=0                                                                     |   |   |   |   |   |   |   |   |   |   |
| CMC      | Complement Carry         | CMC        | CF:=¬CF                                                                  |   |   |   |   |   |   |   |   |   |   |
| STD      | Set Direction            | STD        | DF:=1 (string op's downwards)                                             |   |   |   |   |   |   |   |   |   |   |
| CLD      | Clear Direction          | CLD        | DF:=0 (string op's upwards)                                               |   |   |   |   |   |   |   |   |   |   |
| STI      | Set Interrupt            | STI        | IF:=1                                                                     |   |   |   |   |   |   |   |   |   |   |
| CLI      | Clear Interrupt          | CLI        | IF:=0                                                                     |   |   |   |   |   |   |   |   |   |   |
| PUSH     | Push onto stack          | PUSH       | DEC SP, [SP]:=Source                                                      |   |   |   |   |   |   |   |   |   |   |
| PUSHF    | Push flags               | PUSHF      | O, D, I, T, S, Z, A, P, C                                               |   |   |   |   |   |   |   |   |   |   |
| PUSHA    | Push all general registers | PUSHAH  | AX, CX, DX, BX, SP, BP, SI, DI                                          |   |   |   |   |   |   |   |   |   |   |
| POP      | Pop from stack           | POP        | Dest:=[SP]                                                               | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| POPF     | Pop flags                | POPF       | O, D, I, T, S, Z, A, P, C                                               | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| POPA     | Pop all general registers | POPA      | DI, SI, BP, SP, BX, DX, CX, AX                                          | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| CBW      | Convert byte to word     | CBW        | AX:=AL (signed)                                                          |   |   |   |   |   |   |   |   |   |   |
| CWD      | Convert word to double   | CWD        | DX:AX:=AX (signed)                                                       | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| CWDE     | Conv word extended double | CWDE     | EAX:=AX (signed)                                                          | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| IN       | Input                    | IN         | AL/AX/EAX := byte/word/double of specified port                          |   |   |   |   |   |   |   |   |   |   |
| OUT      | Output                   | OUT        | AL/AX/EAX := byte/word/double of specified port                          | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |

### ARITHMETIC Flags

| Name     | Comment                  | Code       | Operation                                                                 | O | D | I | T | S | Z | A | P | C |
|----------|--------------------------|------------|---------------------------------------------------------------------------|---|---|---|---|---|---|---|---|---|---|
| ADD      | Add                      | ADD        | Dest:=Dest+Source                                                         | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| ADC      | Add with Carry           | ADC        | Dest:=Dest+Source+CF                                                      | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SUB      | Subtract                 | SUB        | Dest:=Dest-Source                                                         | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SBB      | Subtract with borrow     | SBB        | Dest:=Dest-Source(CF)                                                    | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| DIV      | Divide (unsigned)        | DIV        | Op:=byte: AL:=AX / Op                                                     | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| DIVD     | Divide (unsigned)        | DIV        | Op:=word: AX:=DX:AX / Op                                                  | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| MUL      | Multiply (unsigned)      | MUL        | Op:=byte: AX:=AL*Op if AH=0                                              | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| MUL386   | Multiply (unsigned)      | MUL        | Op:=word: DX:AX:=AX*Op if DX=0                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| IMUL     | Signed Integer Multiply  | IMUL       | Op:=byte: AX:=AL*Op if AL sufficient                                     | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| IMUL386  | Signed Integer Multiply  | IMUL       | Op:=word: DX:AX:=AX*Op if AL sufficient                                   | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| INC      | Increment                | INC        | Op:=Op+1 (Carry not affected!)                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| DEC      | Decrement                | DEC        | Op:=Op-1 (Carry not affected!)                                          | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| CMP      | Compare                  | CMP        | Op1,Op2                                                                   | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SAL      | Shift arithmetic left    | SAL        | SAL Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SAR      | Shift arithmetic right   | SAR        | SAR Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| RCL      | Rotate left through Carry| RCL        | RCL Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| RCR      | Rotate right through Carry| RCR       | RCR Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| ROL      | Rotate left              | ROL        | ROL Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| ROR      | Rotate right             | ROR        | ROR Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |

### LOGIC Flags

| Name     | Comment                  | Code       | Operation                                                                 | O | D | I | T | S | Z | A | P | C |
|----------|--------------------------|------------|---------------------------------------------------------------------------|---|---|---|---|---|---|---|---|---|---|
| NEG      | Negate (two-complement)  | NEG        | Op:=0-Op if Op=0 then CF:=1                                               | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| NOT      | Invert each bit          | NOT        | Op:¬Op (invert each bit)                                                 | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| AND      | Logical and              | AND        | Dest:=Dest+Source                                                         | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| OR       | Logical or               | OR         | Dest:=Dest-Source                                                         | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| XOR      | Logical exclusive or     | XOR        | Dest:=Dest (exor) Source                                                  | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SHL      | Shift logical left       | SHL        | SHL Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SHR      | Shift logical right      | SHR        | SHR Op,Quantity                                                           | ± | ± | ± | ± | ± | ± | ± | ± | ± | ± |

---

Download latest version free of charge from [www.jegerlehner.ch/intel](http://www.jegerlehner.ch/intel) This page may be freely distributed without cost provided it is not changed. All rights reserved.
### MISC

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
<td>LEA</td>
<td>Dest,.Source</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>Interrupt</td>
<td>INT</td>
<td>Nr</td>
<td>0</td>
</tr>
</tbody>
</table>

### JUMPS (flags remain unchanged)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Call subroutine</td>
<td>CALL</td>
<td>Proc</td>
<td>RET</td>
<td>Return from subroutine</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>JMP</td>
<td>Dest</td>
<td>JNE</td>
<td>Jump if not Equal</td>
<td>JNE</td>
<td>Dest</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if Equal</td>
<td>JE</td>
<td>Dest (= JZ)</td>
<td>JZ</td>
<td>Jump if Zero</td>
<td>JZ</td>
<td>Dest</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX Zero</td>
<td>JCXZ</td>
<td>Dest</td>
<td>JNP</td>
<td>Jump if no Parity (Parity Odd)</td>
<td>JNP</td>
<td>Dest</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if Parity (Parity Even)</td>
<td>JP</td>
<td>Dest (= JPE)</td>
<td>JPO</td>
<td>Jump if Parity Odd</td>
<td>JPO</td>
<td>Dest</td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if Parity Even</td>
<td>JPE</td>
<td>Dest (= JP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JUMPS Unsigned (Cardinal)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above</td>
<td>JA</td>
<td>Dest (= JNBE)</td>
<td>JG</td>
<td>Jump if Greater</td>
<td>JG</td>
<td>Dest</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>JAE</td>
<td>Dest (= JNB = JNC)</td>
<td>JGE</td>
<td>Jump if Greater or Equal</td>
<td>JGE</td>
<td>Dest</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below</td>
<td>JB</td>
<td>Dest (= JNAE = JC)</td>
<td>JL</td>
<td>Jump if Less</td>
<td>JL</td>
<td>Dest</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>JBE</td>
<td>Dest (= JNA)</td>
<td>JLE</td>
<td>Jump if Less or Equal</td>
<td>JLE</td>
<td>Dest</td>
</tr>
<tr>
<td>JN</td>
<td>Jump if not Above</td>
<td>JNA</td>
<td>Dest (= JBE)</td>
<td>JNG</td>
<td>Jump if not Greater</td>
<td>JNG</td>
<td>Dest</td>
</tr>
<tr>
<td>JNJE</td>
<td>Jump if not Above or Equal</td>
<td>JNJE</td>
<td>Dest (= JB = JC)</td>
<td>JNGE</td>
<td>Jump if not Greater or Equal</td>
<td>JNGE</td>
<td>Dest</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not Below</td>
<td>JNB</td>
<td>Dest (= JAE = JNC)</td>
<td>JNL</td>
<td>Jump if Less or Equal</td>
<td>JNL</td>
<td>Dest</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not Below or Equal</td>
<td>JNBE</td>
<td>Dest (= JA)</td>
<td>JNLE</td>
<td>Jump if not Less or Equal</td>
<td>JNLE</td>
<td>Dest</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if Carry</td>
<td>JC</td>
<td>Dest</td>
<td>JO</td>
<td>Jump if Overflow</td>
<td>JO</td>
<td>Dest</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if no Carry</td>
<td>JNC</td>
<td>Dest</td>
<td>JNS</td>
<td>Jump if no Sign (= positive)</td>
<td>JNS</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### General Registers:

- **EAX**
  - AH
  - AL
- **EDX**
  - DX
  - DL
- **ECX**
  - CX
  - CL
- **EBX**
  - BH
  - BL

### Example:

- `.MODEL SEG` ; Demo program
- `.MODEL SMALL` ; Const
- `.STACK 1024` ; Data
- `.DATA` ; define Byte, any value
- `.DB ?` ; define Word, binary
- `.DW 1010b` ; define Word, decimal
- `.DD OFFFFh` ; define Doubleword, hex
- `.SDB "Hello !",0` ; define String
- `.CODE` ; resolved by linker
- `.MOVA,DGROUP` ; init dataseg reg
- `.MOVA,[VarB].42` ; init VarB
- `.MOVA,[VarB].7` ; set VarD
- `.MOVA,Offset[S]` ; addr of "*" of "Hello !"
- `.MOVA,[VarW]` ; get value into accumulator
- `.ADD A,VarW2` ; add VarW2 to AX
- `.MOV [VarW2].AX` ; store AX in VarW2
- `.MOV AX,4C00h` ; back to system
- `.INT 21h` ; back to system
- `.END main` ; back to system

### Status Flags (result of operations):

- **C:** Carry
- **O:** Overflow
- **S:** Sign
- **Z:** Zero
- **A:** Aux. carry
- **P:** Parity

### Control Flags (how instructions are carried out):

- **D:** Direction
- **I:** Interrupt
- **T:** Trap