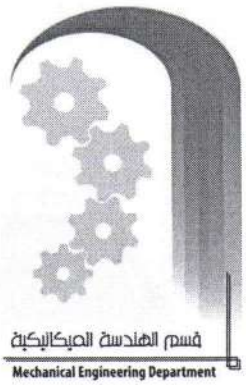


University of Technology

Mechanical Engineering Department

Microprocessors and Microcontrollers

Prepared by:

Dr. Alaa Abdulhady Jaber

20039@uotechnology.edu.iq

Baghdad/Iraq

2017-2018

Course Outline

Course Name:	Microprocessors and Microcontrollers	المعالجات و المسيطرات الدقيقة	اسم المقرر:
Course Code:	ME395-2	همك395-2	رمز المقرر:
Units:	2	2	الوحدات:

Hours per Week			الساعات الأسبوعية		
Theoretical	Experimental	Tutorial	مناقشة	عملي	نظري
1	2	-	-	2	1

Assessment criteria		معايير التقييم	
Final exam	60%	%60	الامتحان النهائي
Midterm exam	15%	%15	امتحان نصف فصلي
Quizzes	10%	%10	امتحانات مفاجئة
Lab	10%	%10	درجة المختبر
Continuous assessment	5%	%5	تقييم مستمر

Week	Contents	المحتويات	الاسبوع
1 2	**Introduction to microprocessor:** - Microprocessor architecture, 4 to 64 bit architecture with detail on 32bit - Microprocessor bus system, registers organization, and pin configuration. - Comparison between Complex instruction computers CISC and reduced instruction set computer RISC, Princeton architecture and Harvard architecture. - Memory system organization, ROM, RAM chips and their varieties. - Addressing modes and their features, paging and segmentation.	مدخل الى المعالجات الدقيقة:	1 2
3 4 5 6	**Microprocessor instruction set:** - Software instruction set, instruction cycle, machine cycle and timing diagram. - Instruction set groups according to function: data transfer, arithmetic, logic, branching, Stack operations, I/O operations and machine control	طقم الأوامر في المعالجات الدقيقة:	3 4 5 6

	instructions. - Assembly language programming, assembler directives, assembly process, LST and HEX files. - Debugging assembly language programs. - Introduction to C programming for microprocessors and microcontrollers. - Assembly and C language programming.		
7 8	**Hardware interfacing:** - Interfacing memory ROM and RAM. - Hardware I/O ports design. - Memory mapped and I/O mapped I/O.	تعشيق المعالج الدقيق مع بقية العتاد المطلوب لعمل منظومات الحاسوب:	7 8
9	**Interrupts:** - Introduction to interrupt system. - Interrupt controllers with examples on 8259 programmable interrupt controller.	المقاطعة و فوائدها في منظومات الحاسوب:	9
10	**Direct memory access:** - Introduction to DMA system and its features - DMA controller with examples on 8237 DMA controller.	تقنية الولوج المباشر للذاكرة:	10
11	**Analogue to digital and Digital to analogue conversion** - Types of ADC - Types of DAC	اساليب تحويل الإشارات التماثلية الى رقمية، و الإشارات الرقمية الى تماثلية:	11
12 13	**Programmable peripheral devices:** - The 8255 programmable peripheral interface device - The 8279 display and keyboard / I/O interface device. - The 8251 universal synchronous/asynchronous receiver transmitter	النبائط الطرفية المبرمجة:	12 13
14	**Examples on computer system design using microprocessors**	امثلة على منظومات الحاسوب المصممة باستخدام المعالجات الدقيقة:	14
15	**Examples on embedded systems design using microcontrollers.**	امثلة على تطبيقات المسيطرات الدقيقة في الأنظمة المتضمنة.	15

References

1. Introduction to Microprocessors and Microcontrollers, by: John Crisp, 2nd Edition, Elsevier Newnes, 2004, USA.
2. The Intel Microprocessors, Architecture, Programming, and Interfacing, by: Barry B. Brey, 8th Edition, Pearson Education, 2009, USA.
3. Inside the Machine, by: Jon Stokes, 2nd Edition, no Starch Press Inc., 2007, USA.

Lecture One

1.1 Introduction

The minimal hardware configuration of a microcomputer system is composed of three fundamental components that are a **central processing unit** (CPU), the **system memory**, and some form of **input/output (I/O) interface**. These components are interconnected by multiple sets of lines grouped according to their functions, and globally denominated the ***system buses***. An additional set of components provide the necessary power and timing synchronization for system operation. Figure 1.1 illustrates the integration of such a basic structure.

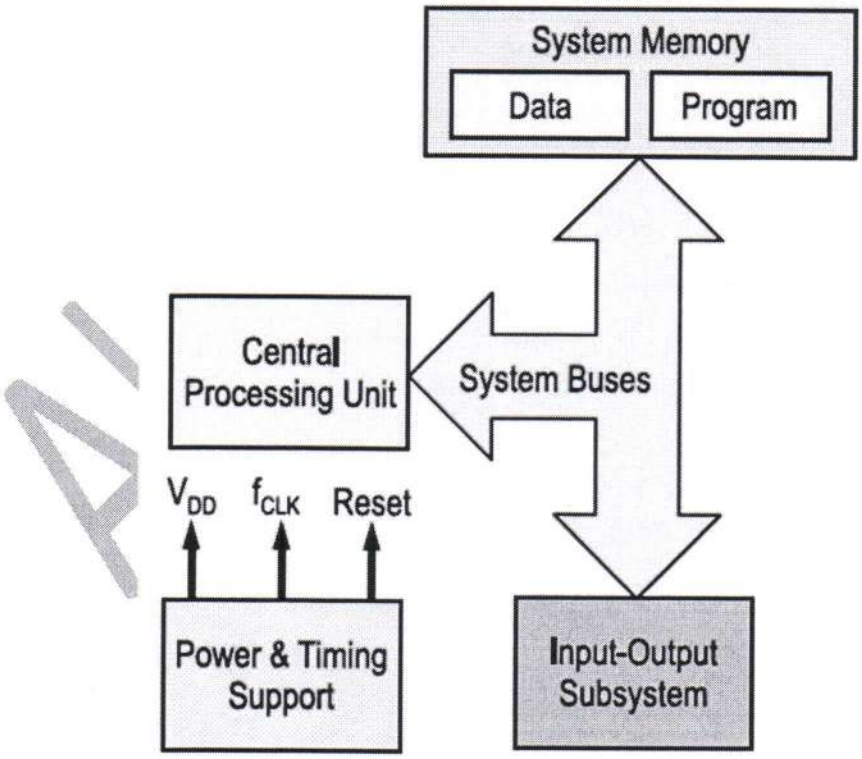


Figure 1.1: General architecture of a microcomputer system

The components of a microcomputer can be implemented in diverse ways. They could be deployed with multiple chips on a board-level microcomputer or

integrated into a single chip, in a structure named microcomputer-on-a-chip, or simply a ***Microcontroller***. Nowadays most embedded systems are developed around microcontrollers. Regardless of the implementation style, each component of a microcomputer has the same specific function, as described below.

***Central Processing Unit* (CPU):** The CPU forms the heart of the microcontroller system. It retrieves instructions from program memory, decodes them, and accordingly operates on data and/or on peripherals devices in the Input-Output subsystem to give functionality to the system.

***System Memory*:** The place where programs and data are stored to be accessed by the CPU is the system memory. Two types of memory elements are identified within the system: **Program Memory** and **Data Memory**. Program memory stores programs in the form of a sequence of instructions. Programs dictate the system operation. Data Memory stores data to be operated on by programs.

Input/Output subsystem: The I/O subsystem, also called ***Peripheral Subsystem*** includes all the components or peripherals that allow the CPU to exchange information with other devices, systems, or the external world. The I/O subsystem includes all the components that complement the CPU and memory to form a computer system.

System Buses: The set of lines interconnecting CPU, Memory, and I/O subsystem are denominated the system buses. Groups of lines in the system buses perform different functions. Based on their function the system bus lines are sub-divided into ***address*** bus, ***data*** bus, and ***control*** bus.

1.2 Microcontroller Versus Microprocessor

Before we delve any deeper into the structure of the different components of a microcomputer system, let's first establish the fundamental difference between microprocessors and microcontrollers.

1.2.1 Microprocessor Units

A Microprocessor Unit, commonly abbreviated MPU, fundamentally contains a general purpose CPU in its die. To develop a basic system using a MPU, all components depicted in Figure 1.1 other than the CPU, i.e., the buses, memory, and I/O interfaces, are implemented externally.

The most common examples of systems designed around an MPUs are personal computers and mainframes. But these are not the only ones. There are many other systems developed around traditional MPUs. Manufacturers of MPU's include INTEL, Freescale, Zilog, Fujitsu, Siemens, and many others. Microprocessor design has advanced from the initial models in the early 1970s to present day technology.

Figure 1.2: Currentally available microprocessor

3.2.2 Microcontroller Units

A microcontroller unit, abbreviated MCU, is developed using a microprocessor core or central processing unit (CPU), usually less complex than that of an MPU. This basic CPU is then surrounded with memory of both types (program and data) and several types of peripherals, all of them embedded into a single integrated circuit, or chip. This blending of CPU, memory, and I/O within a single chip is what we call a ***microcontroller***. However, ***Micro*** suggests that the device is small, and ***controller*** suggests that the device can be used in control applications.

The assortment of components embedded into a microcontroller allows for implementing complete applications requiring only a minimal number of external components or in many cases solely using the MCU chip. Peripheral timers, input/output (I/O) ports, interrupt handlers, and data converters are among those commonly found in most microcontrollers. The provision of such an assortment of resources inside the same chip is what has gained them the denomination of ***computers-on-a-chip***. Figure 1.3 shows a typical microcontroller configuration.

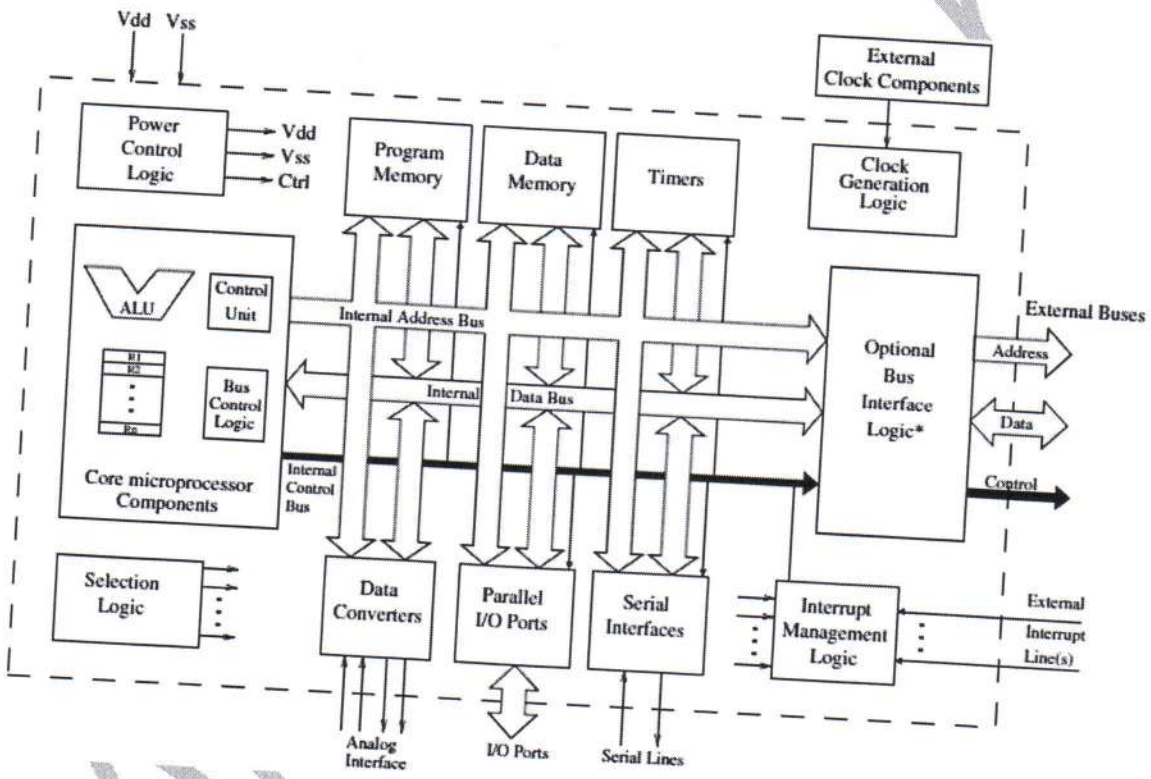


Figure 1.3: Structure of a typical microcontroller

Microcontrollers share a number of characteristics with general purpose microprocessors. Yet, the core architectural components in a typical MCU are less complex and more application oriented than those in a general purpose microprocessor. Microcontrollers are usually marketed as family members. Each family is developed around a basic architecture which defines the common characteristics of all members. These include, among others, the data and program path widths, architectural style, register structure, base instruction set, and addressing modes. Features differentiating family members include the

amount of on-chip data and program memory and the assortment of on-chip peripherals. There are literally hundreds, perhaps thousands, of microprocessors and microcontrollers on the market. Table 1.1 shows a very small sample of microcontroller family models of different sizes from six companies.

Table 1.1: A sample of MCU families/series

Company	4-bits	8-bits	16-bits	32-bits
EM Microelectronic	EM6807	EM6819		
Samsung	S3P7xx	S3F9xxx	S3FCxx	S3FN23BXZZ
Freescale semiconductor		68HC11	68HC12	
Toshiba		TLCS-870	TLCS-900/L1	TLCS-900/H1
Texas instruments			MSP430	TMS320C28X
			TMS320C24X	Stellaris line
Microchip		PIC1X	PIC2x	PIC32

1.3 Microcontroller Applications

Basically, a microcomputer (or microcontroller) executes a user program which is loaded in its program memory. Under the control of this program, data are received from external devices (inputs), manipulated and then sent to external devices (outputs). For example, in a microcontroller-based fluid-level control system, the fluid level is read by the microcomputer via a level-sensor device and the microcontroller attempts to control the fluid level at the required value. If the fluid level is low, the microcomputer operates a pump to draw more fluid from the reservoir in order to keep the fluid at the required level. Figure 1.4 shows the block diagram of our simple fluid-level control system. The system shown in Figure 1.4 is a very simplified fluid-level control system. In a more sophisticated system, we may have a keypad to set the required fluid level, and a liquid-crystal display (LCD) to display the current level in the tank. Figure 1.5 shows the block diagram of this more sophisticated fluid-level control system.

We can make our design even more sophisticated (Figure 1.6) by adding an audible alarm to inform us if the fluid level is outside the required value. Also, the actual level at any time can be sent to a PC every second for archiving and further processing. For example, a graph of the daily fluid-level changes can be plotted on the PC. As you can see, because the microcontrollers are

programmable, it is very easy to make the final system as simple or as complicated as we like.

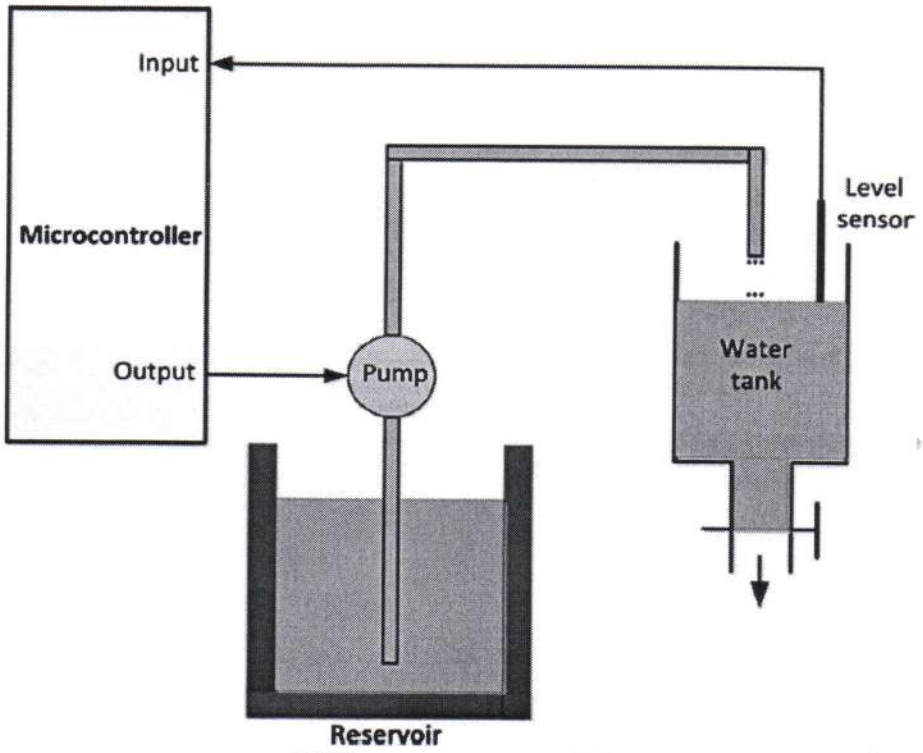


Figure 1.4: Microcontroller-based fluid-level control system

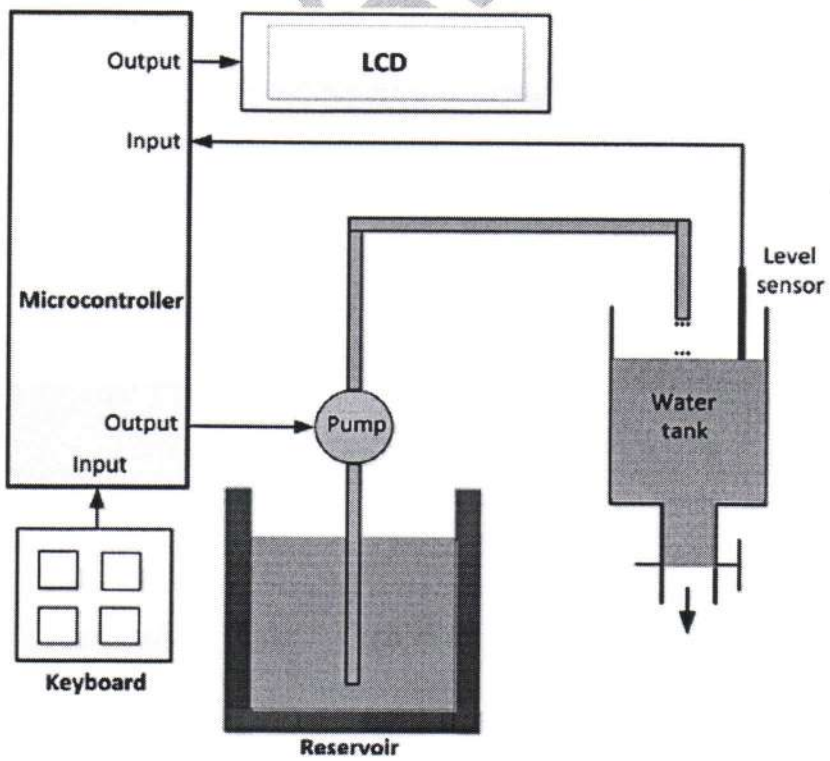


Figure 1.5: Fluid-level control system with a keypad and an LCD

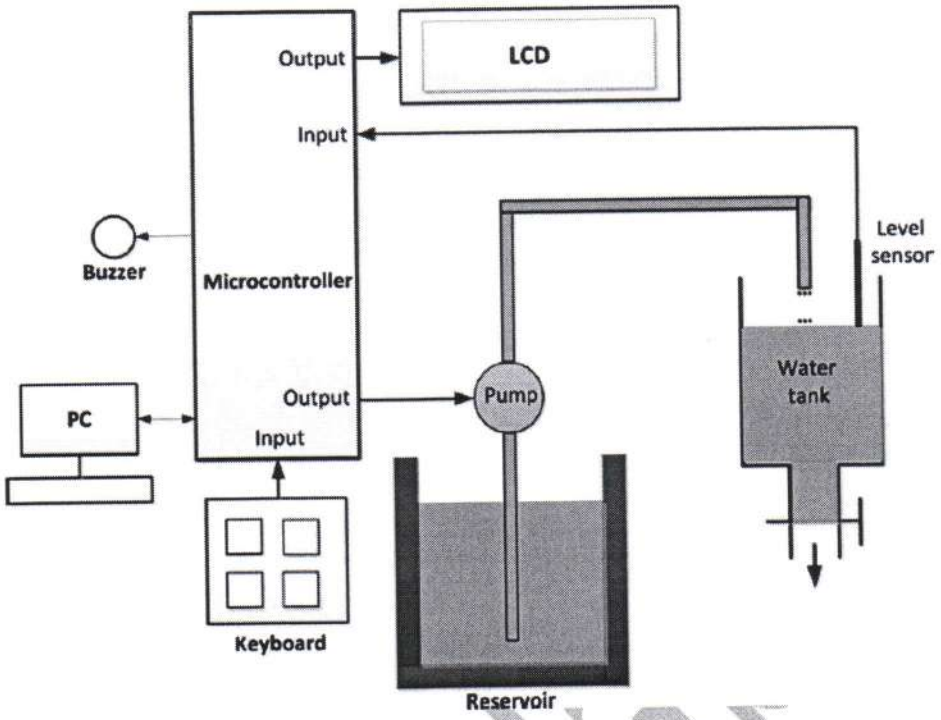


Figure 1.6: More sophisticated fluid-level controller

1.4 Explanation of Terms

Before we go on, it is necessary to understand some basic terms.

- ***Bit*** is an abbreviation for the term ***binary digit***. A binary digit can have only two values, which are represented by the symbols **0** and **1**, whereas a decimal digit can have 10 values, represented by the symbols **0** through **9**. The bit values are easily implemented in electronic and magnetic media by two-state devices whose states portray either of the binary digits 0 and 1. Examples of such two-state devices are a transistor that is conducting or not conducting and a capacitor that is charged or discharged.
- ***Address*** is a pattern of 0's and 1's that represents a specific location in memory or a particular I/O device. An 8-bit microcontroller with 16 address bits can produce 2^{16} unique 16-bit patterns from 0000000000000000 to 1111111111111111, representing 65,536 different address combinations (addresses 0 to 65,535).

- ***Arithmetic -logic unit (ALU)*** is a digital circuit that performs arithmetic and logic operations on two n-bit digital words. The value of n for microcontrollers can be 8-bit or 16-bit. Typical operations performed by an ALU are addition, subtraction, and comparison of two n-bit digital words. The size of the ALU defines the size of the microcontroller. For example, an 8-bit microcontroller contains an 8-bit ALU.
- ***Bit size*** refers to the number of bits that can be processed simultaneously by the basic arithmetic circuits of a microcontroller. A number of bits taken as a group in this manner is called a word. For example, an 8-bit microcontroller can process an 8-bit word.
- ***Bus*** consists of a number of conductors (wires) organized to provide a means of communication among different elements in a microcontroller system. The conductors in a bus can be grouped in terms of their functions. A microcontroller normally has an address bus, a data bus, and a control bus. Address bits are sent to memory or to an external device on the *address bus*. Instructions from memory, and data to/from memory or external devices, normally travel on the *data bus*. Control signals for the other buses and among system elements are transmitted on the *control bus*. Buses are sometimes ***bidirectional***; that is, information can be transmitted in either direction on the bus, but normally in only one direction at a time.
- ***Clock*** is analogous to human heart beats. The microcontroller requires synchronization among its components, and this is provided by a *clock* or timing circuits.
- ***Timers*** are important parts of any microcontroller. A timer is basically a counter which can be stopped or started by program control. Most timers can be configured to generate an interrupt when they reach a certain

count. The interrupt can be used by the user program to carry out accurate timing-related operations inside the microcontroller.

- ***Reset Input*** is used to reset a microcontroller externally. Resetting puts the microcontroller into a known state such that the program execution starts from a known address. An external reset action is usually achieved by connecting a push-button switch to the reset input such that the microcontroller can be reset when the switch is pressed.
- ***Interrupts*** are very important concepts in microcontrollers. An interrupt causes the microcontroller to respond to external and internal (e.g. a timer) events very quickly. When an interrupt occurs, the microcontroller leaves its normal flow of program execution and jumps to a special part of the program, known as the interrupt service routine (ISR). The program code inside the ISR is executed and upon return from the ISR, the program resumes its normal flow of execution.
- Brown-out Detector are also in many microcontrollers and they reset a microcontroller if the supply voltage falls below a nominal value. Brown-out detectors are safety features and they can be employed to prevent unpredictable operation at low voltages.
- ***Supply Voltage:*** Most microcontrollers operate with the standard logic voltage of +5 V. Some microcontrollers can operate at as low as +2.7 V and some will tolerate +6 V without any problems. The manufacturers' data sheets have to be checked check about the allowed limits of the power supply voltage.

1.5 Bits, bytes and other things

All the information entering or leaving a microprocessor is in the form of a binary signal, a voltage switching between the two bit levels 0 and 1. Bits are passed through the microprocessor at very high speed and in large numbers and we find it easier to group them together.

Nibble: A group of four bits handled as a single lump. It is half a byte.

Byte: A byte is simply a collection of 8 bits. Whether they are ones or zeros or what their purpose is does not matter.

Word: A number of bits can be collected together to form a '**word**'. Unlike a byte, a word does not have a fixed number of bits in it. The length of the word or the number of bits in the word depends on the microprocessor being used. If the microprocessor accepts binary data in groups of 32 at a time, then the word in this context would include 32 bits. If a different microprocessor used data in smaller handfuls, say 16 at a time, then the word would have a value of 16 bits.

Long word: In some microprocessors where a word is taken to mean say 16 bits, a long word would mean a group of twice the normal length, in this case 32 bits.

Kilobyte (Kb or KB or kbyte): A kilobyte is 1024 or $\mathbf{2^{10}}$ bytes. In normal use, kilo means 1000 so a kilovolt or kV is exactly 1000 volts. The difference between 1000 and 1024 is fairly slight when we have only 1 or 2 Kb and the difference is easily ignored. However, as the numbers increase, so does the difference. The actual number of bytes in 42 Kb is actually 43 008 bytes (42*1024).

Megabyte (MB or Mb): This is a kilo kilobyte or 1024 x 1024 bytes. Numerically this is $\mathbf{2^{20}}$ or 1 048 576 bytes. Be careful not to confuse this with mega as in megavolts (MV) which is exactly one million (10^6).

Gigabyte (Gb): This is 1024 megabytes which is $\mathbf{2^{30}}$ or 1 073 741 824 bytes. In general engineering, giga means one thousand million (10^9).

Terabyte (TB or Tb): Terabyte is a mega megabyte or 2^{40} or 1 099 511 600 000 bytes (it is not normal Tera which is = 10^{12}).

Lecture Two

2.1 Central Processing Unit

The Central Processing Unit (CPU) in a microcomputer system is typically a microprocessor unit (MPU) or core. The CPU is where instructions become signals and hardware actions that command the microcomputer operation. The minimal list of components that define the architecture of a CPU include the following:

- Hardware Components:
  - An Arithmetic Logic Unit (ALU)
  - A Control Unit (CU)
  - A Set of Registers
  - Bus Interface Logic (BIL)
- Software Components:
  - Instruction Set
  - Addressing Modes

The instructions and addressing modes will be defined by the specifics of the hardware ALU and CU units. In this section we concentrate on the hardware components. Figure 2.1 illustrates a simplified model view of the CPU with its internal hardware components. These components allow the CPU to access programs and data stored somewhere in memory or input/output subsystem, and to operate as a stored program computer. The sequence of instructions that make a program are chosen from the processor's instruction set. A memory stored program dictates the sequence of operations to be performed by the system. In the processing of data, each CPU component plays a necessary role that complements those of the others.

The collection of hardware components within the CPU performing data operations is called the processor's ***Data Path***. The CPU data path includes the

ALU, the internal data bus, and other functional components such as floating-point units, hardware multipliers, and so on. The hardware components performing system control operations are designated ***Control Path***. The control unit is at the heart of the CPU control path. The bus control unit and all timing and synchronization hardware components are also considered part of the control path.

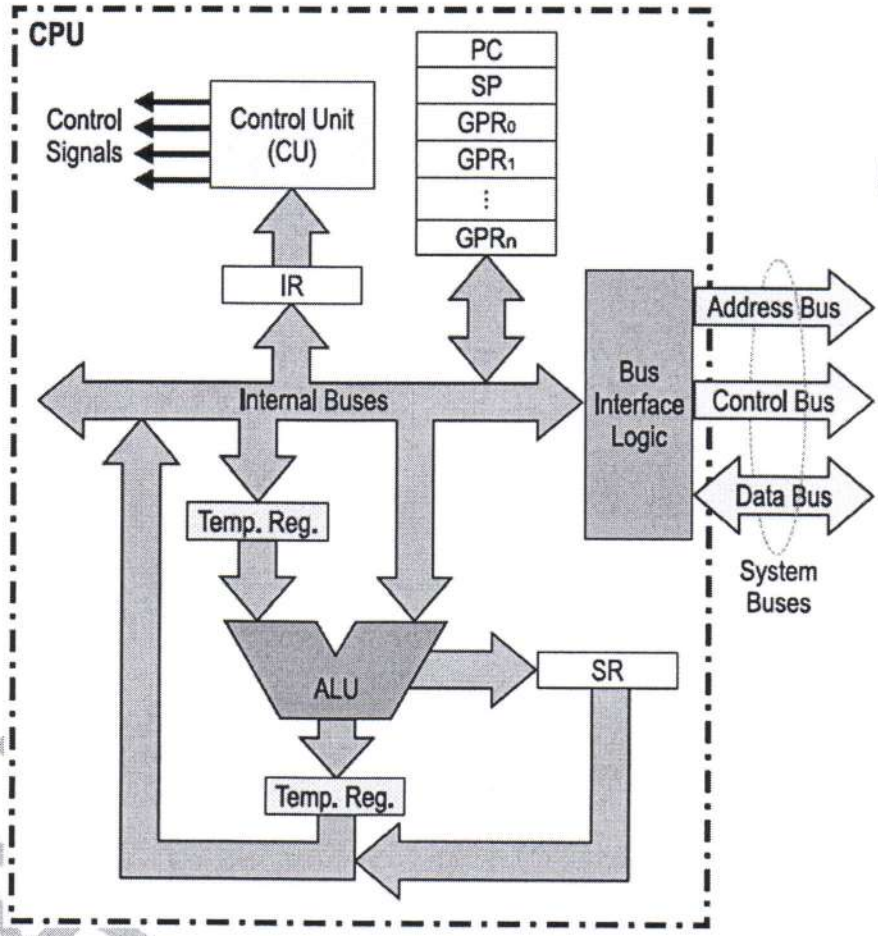


Figure 2.1: Minimal architectural components in a simple CPU

2.1.1 Control Unit

The control unit (CU) governs the CPU operation working like a finite state machine that cycles forever through three states: fetch, decode, and execute, as illustrated in Figure 2.2. This fetch-decode-execute cycle is also known as ***instruction cycle*** or ***CPU cycle***. The complete cycle will generally take several clock cycles, depending on the instruction and operands. It is usually assumed

as a rule of the thumb that it takes at least four clock cycles.3 Since the instruction may contain several words, or may require several intermediate steps, the actual termination of the execution process may require more than one instruction cycle.

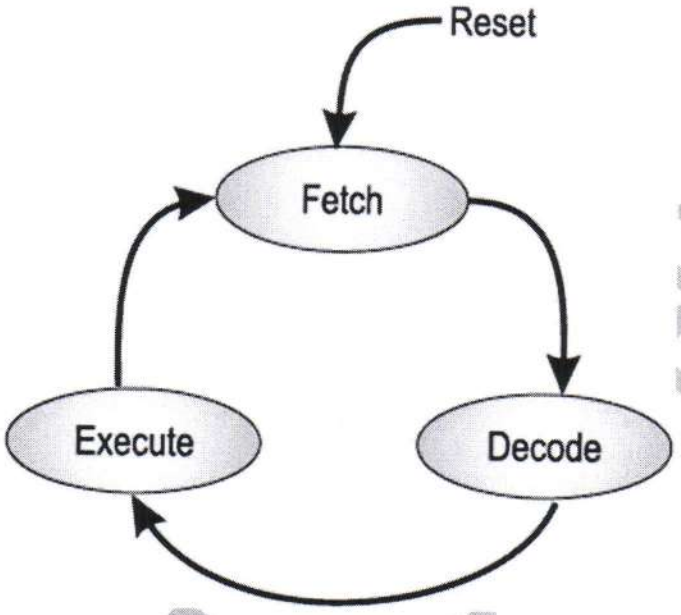


Figure 2.2: States in control unit operation: fetch, decode, and execute

Several CPU blocks participate in the fetch-decode-execute process, among which we find special purpose registers **PC** (program counter) and **IR** (instruction register) (see Figure 2.1). The cycle can be described as follows:

1- **Fetch State**: During the fetch state a new instruction is brought from memory into the CPU through the bus interface logic (BIL). The *program counter* (PC) provides the address of the instruction to be fetched from memory. The newly fetched instruction is read along the data bus and then stored in the *instruction register* (IR).

2- **Decoding State**: After fetching the instruction, the CU goes into a decoding state, where the instruction meaning is deciphered. The decoded information is used to send signals to the appropriate CPU components to execute the actions specified by the instruction.

3- **Execution State**: In the execution state, the CU commands the corresponding CPU functional units to perform the actions specified by

the instruction. At the end of the execution phase, the PC has been incremented to point to the address of the next instruction in memory.

After the execution phase, the CU commands the BIL to use the information in the program counter to fetch the next instruction from memory, initiating the cycle again. The cycle may require intermediate cycles similar to this one whenever the decoding phase requires reading (fetching) other values from memory. This is dictated by the addressing mode use in the instruction, as it will be explained later.

Being the CU a finite state machine, it needs a ***Reset signal*** to initiate the cycle for the first time. The program counter is hardwired to, upon reset, load the memory address of the first instruction to be fetched. That is how the first cycle begins operation. The address of this first instruction is called ***reset vector***.

2.1.2 Arithmetic Logic Unit

The arithmetic logic unit (ALU) is the CPU component where all logic and arithmetic operations supported by the system are performed. Basic arithmetic operations such as addition, subtraction, and complement, are supported by most ALUs. So may also include hardware for more complex operations such as multiplication and division, although in many cases these operations are supported via software or by other peripherals, such as a hardware multiplier.

The CU governs the ALU by specifying which particular operation is to be performed, the source operands, and the destination of the result. The width of the operands accepted by the ALU of a particular CPU (data path width) is typically used as an indicator of the CPU computational capacity. When for example, a microprocessor is referred to as a 16-bit unit, its ALU has the capability of operating on 16-bit data. The ALU data width shapes the CPU data path architecture establishing the width of data bus and data registers.

2.1.3 Bus Interface Logic

The Bus Interface Logic (BIL) refers to the CPU structures that coordinate the interaction between the internal buses and the system buses. The BIL defines

how the external address, data, and control buses operate. In small embedded systems the BIL is totally contained within the CPU and transparent to the designer. In distributed and high performance systems the BIL may include dedicated peripherals devoted to establish the CPU interface to the system bus. Examples of such extensions are bus control peripherals, bridges, and bus arbitration hardware included in the chip set of contemporary microprocessor systems.

2.1.4 Registers

CPU registers provide temporary storage for data, memory addresses, and control information in a way that can be quickly accessed. They are the fastest form of information storage in a computer system, while at the same time they are the smallest in capacity. Register contents is *volatile*, meaning that it is lost when the CPU is de-energized. CPU registers can be broadly classified as general purpose and specialized.

General purpose registers (GPR) are those not tied to specific processor functions and may be used to hold data, variables, or address pointers as needed. Based on this usage, some authors classify them also as *data* or *address registers*. Depending on the processor architecture, a CPU can contain from as few as two to several dozen

GPRs.

Special purpose registers perform specific functions that give functionality to the CPU. The most basic CPU structure includes the following four specialized registers:

- Instruction Register (IR)
- Program Counter (PC), also called Instruction Pointer (IP)
- Stack Pointer (SP)
- Status Register (S)

Instruction Register (IR)

This register holds the instruction that is being currently decoded and executed in the CPU. The action of transferring an instruction from memory into the IR is called *instruction fetch*. In many small embedded systems, the IR holds one instruction at a time. CPUs used in distributed and high-performance systems usually have multiple instruction registers arranged in a queue, allowing for concurrently *issuing* instructions to multiple functional units. In these architectures the IR is commonly called an *instruction queue*.

Program Counter (PC)

This register holds the address of the instruction to be fetched from memory by the CPU. It is sometimes also called the *instruction pointer (IP)*. Every time an instruction is fetched and decoded, the control unit increments the value of the PC to point to the next instruction in memory. Being the PC an address register, its width may also determine the size of the largest program memory space directly addressable by the CPU.

Stack Pointer (SP)

The *stack* is a specialized memory segment used for temporarily storing data items in a particular sequence.

Status Register (SR)

The status register, also called the *Processor Status Word (PSW)*, or *Flag Register* contains a set of indicator bits called *flags*, as well as other bits controlling the CPU status. A flag is a single bit that indicates the occurrence of a particular condition.

2.2 System Buses

Memory and I/O devices are accessed by the CPU through the system buses. A bus is simply a group of lines that perform a similar function. Each line carries a bit of information and the group of bits may be interpreted as a whole. The system buses are grouped in three classes: *address*, *data*, and *control* buses. These are described next.

2.2.1 Data Bus

The set of lines carrying data and instructions to or from the CPU is called the *data bus*. A *read* operation occurs when information is being transferred *into* the CPU. A data bus transfers *out from* the CPU into memory or into a peripheral device, is called a *write* operation. Note that the designation of a transfer on the data bus as read or write is always made with respect to the CPU. This convention holds for every system component. Data bus lines are generally bi-directional because the same set of lines allows us to carry information to or from the CPU. One transfer of information is referred to as *data bus transaction*.

The number of lines in the data bus determines the maximum *data width* the CPU can handle in a single transaction; wider data transfers are possible, but require multiple data bus transactions. For example, an 8-bit data bus can transfer at most one byte (or two nibbles) in a single transaction, and a 16-bit transaction would require two data bus transactions. Similarly, a 16-bit data bus would be able to transfer at most, two bytes per transaction; transferring more than 16 bits would require multiple transactions.

2.2.2 Address Bus

The CPU interacts with only one memory register or peripheral device at a time. Each register, either in memory or a peripheral device, is uniquely identified with an identifier called *address*. The set of lines transporting this address information form the *address bus*. These lines are usually unidirectional and coming out from the CPU. Addresses are usually named in hexadecimal notation.

The width of the address bus determines the size of the largest memory space that the CPU can address. An address bus of m bits will be able to address at most 2^m different memory locations, which are referred to by hex notation. For example, with a 16-bit address bus, the CPU can access up to 2^{16} = 64K locations named 0x0000, 0x0001, . . ., 0xFFFF. Notice that the bits of the

address bus lines work as a group, called *address word*, and are not considered meaningful individually.

2.2.3 Control Bus

The *control bus* groups all the lines carrying the signals that regulate the system activity. Unlike the address and data buses lines which are usually interpreted as a group (address or data), the control bus signals usually work and are interpreted separately. Control signals include those used to indicate whether the CPU is performing a read or write access, those that synchronize transfers saying when the transaction begins and ends, those requesting services to the CPU, and other tasks. Most control lines are unidirectional and enter or leave the CPU, depending on their function. The number and function of the lines in a control bus will vary depending on the CPU architecture and capabilities.

Lecture Three

3.1 Memory Organization

The memory subsystem stores instructions and data. Memory consists of a large number of hardware components which can store one bit each. These bits are organized in *n*-bit words, working as a register, usually referred to as cell or location. The contents of cells is a basic unit of information called *memory word*. In addition, each memory location is identified by a unique identifier, its *memory address*, which is used by the CPU to either read or write over the memory word stored at the location. In general, a memory unit consisting of m cells of n bits each is referred to as an $m \times n$ memory; for $n = 1$ and $n = 8$ it is customary to indicate a number followed by b or B, respectively. Thus, we speak of 1Mb (one Mega Bits) and 1MB (one Mega Bytes) memories to refer to $1M \times 1$ and $1M \times 8$ cases.

Usually, addresses are sequentially numbered as illustrated in Figure 3.1. However, in a specific MCU model some addresses may not be present. The example in the figure shows a memory module of 64K cells, each storing an 8-bit word (byte), forming a 64 kilo-byte (64KB) memory. In the illustration, for example, the cell at address 0FFFEh contains the value 27 h. The CPU uses the address bus to select only the cell with which it will interact. The interaction with the contents is realized through the data bus. In a write operation, the CPU modifies the information contained in the cell, while in a read operation it retrieves this word without changing the contents. The CPU uses control bus signals to determine the type of operation to be realized, as well the direction in which the data bus will operate—remember that the data bus lines are bidirectional.

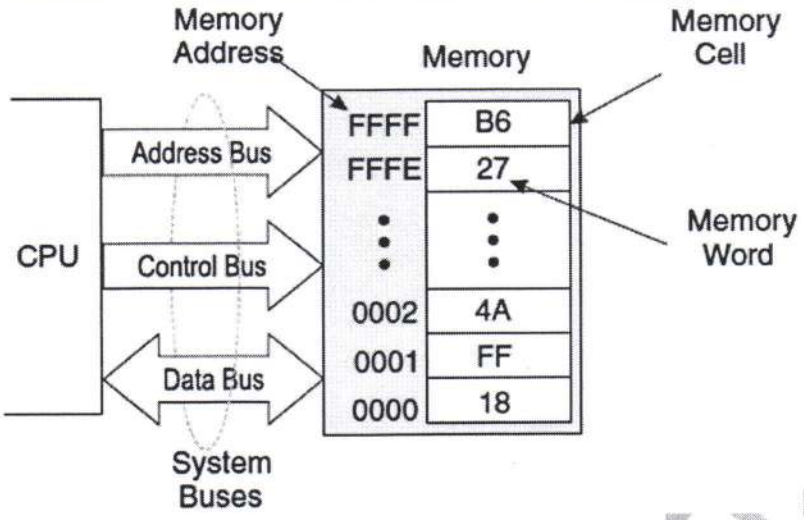


Figure 3.1: Memory structure

3.2 Memory Types

Memory is an important part of a microcontroller system. Depending on the type used, we can classify memories into two groups: program memory and data memory. Program memory stores the programs written by the programmer and this memory is usually non-volatile, i.e. data are not lost after the removal of power. Data memory is where the temporary data used in a program are stored and this memory is usually volatile, i.e. data are lost after the removal of power.

There are basically six types of memories, summarized as follows:

Traditionally, memory technology has been divided into two categories:

- ***Volatile:*** This is memory that only works as long as it is powered. It loses its stored value when power is removed, but can be used as memory for temporary data storage. Generally, this type of memory uses simple semiconductor technology and is easier to write to from an electrical point of view. For historical reasons it has commonly been called RAM (Random Access Memory). A slightly more descriptive name is simply 'data memory'.
- ***Non-volatile:*** This is memory that retains its stored value even when power is removed. On a desktop computer this function is achieved primarily via the hard disk, a huge non-volatile store of data. In an embedded system it is

achieved using non-volatile semiconductor memory. It is a greater challenge to make non-volatile memory, and sophisticated semiconductor technology is applied. Generally, this type of memory has been more difficult to write to electrically, for example in terms of time or power taken, or complexity of the writing process. Non-volatile memory is used for holding the computer program and for historical reasons has commonly been called ROM (Read-Only Memory). A more descriptive name is 'program memory'.

However, based on the above mentioned memories there are six different types of memories as follows:

3.2.1 Random Access Memory

Random access memory (RAM) is a general-purpose memory which usually stores the user data in a program. RAM memory is volatile in the sense that it cannot retain data in the absence of power, i.e. data are lost after the removal of power. Most microcontrollers have some amount of internal RAM. Several kilobytes are a common amount, although some microcontrollers have much more, and some less. RAM chips can be designed in two different forms which we call static RAM (SRAM) and dynamic RAM (DRAM), as seen in Figure 3.2. Static memory (SRAM) is faster than dynamic memory (DRAM). Also, SRAM consumes less power than DRAM. SRAM is more expensive, since it takes more space on silicon.

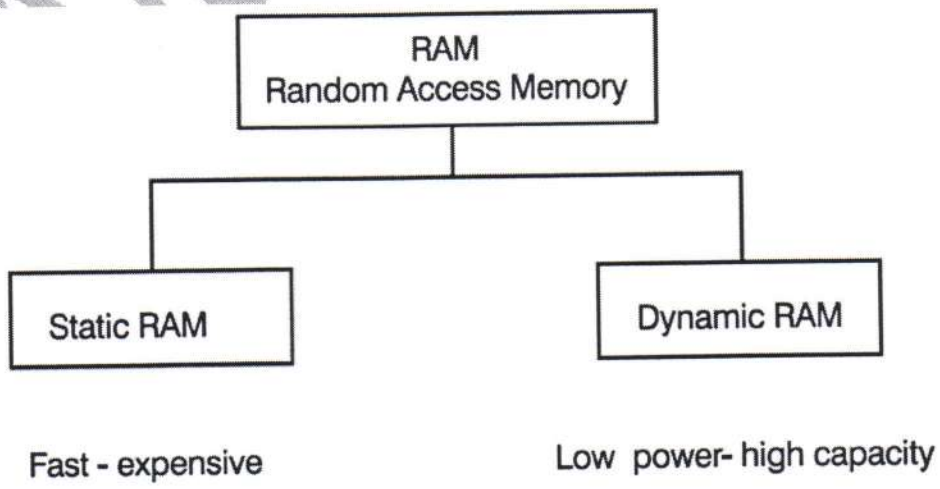


Figure 3.2: The two types of RAM

3.2.2 Read Only Memory

Read only memory (ROM) usually holds program or fixed user data. ROM is non-volatile. If power is removed from ROM and then reapplied, the original data will still be there. ROM memories are programmed at factory during the manufacturing process and their contents cannot be changed by the user. ROM memories are only useful if we have developed a program and wish to order several thousand copies of it, or if we wish to store some configuration data.

3.2.3 Programmable Read Only Memory

Programmable read only memory (PROM) is a type of ROM that can be programmed in the field, often by the end user, using a device called a PROM programmer. Once a PROM has been programmed, its contents cannot be changed. PROMs are usually used in low-production applications where only several such memories are required.

3.2.4 Erasable Programmable Read Only Memory

Erasable programmable read only memory (EPROM) is similar to ROM, but the EPROM can be programmed using a suitable programming device. EPROM memories have a small, clear glass window on the top of the chip where the data can be erased under strong ultraviolet light. Once the memory is programmed, the window can be covered with dark tape to prevent accidental erasure of the data. An EPROM memory must be erased before it can be reprogrammed. Many development versions of microcontrollers are manufactured with EPROM memories where the user program can be stored. These memories are erased and reprogrammed until the user is satisfied with the program. Some versions of EPROMs, known as one-time programmable (OTP), can be programmed using a suitable programmer device but these memories cannot be erased. OTP memories cost much less than the EPROMs. OTP is useful after a project has been developed completely and it is required to make many copies of the program memory.

Figure 3.3: Erasable Programmable Read Only Memory (EPROM)

3.2.5 Electrically Erasable Programmable Read Only Memory

Electrically erasable programmable read only memory (EEPROM) is a non-volatile memory. These memories can be erased and also be reprogrammed using suitable programming devices. EEPROMs are used to save configuration information, maximum and minimum values, identification data, etc.

3.2.6 Flash EEPROM

This is another version of EEPROM-type memory. This memory has become popular in microcontroller applications and is generally used to store the user program. Flash EEPROM is non-volatile and is usually very fast. The data can be erased and then reprogrammed using a suitable programming device. These memories can also be programmed without removing them from their circuits. Some microcontrollers have only 1 KB flash EEPROM while some others have 32 KB or more.

3.3 Microcontroller Architectures

To interact with memory, there must be two types of number moved around: the address of the memory location required and the actual data that belongs in the location. These are connected in two sets of interconnections, called the **address bus** and the **data bus**.

A simple way of meeting the need just described is shown in Figure 3.4. It is called the Von Neumann structure or architecture, after its inventor. The

computer has just one address bus and one data bus, and the same address and data buses serve both program and data memories. The input/output may also be interconnected in this way and made to behave like memory as far as the CPU is concerned. The Von Neumann structure is simple and logical, and gives a certain type of flexibility. The addressable memory area can be divided up in any way between program memory and data memory. However, it suffers from two disadvantages. One is that it is a 'one size fits all' approach. It uses the same data bus for all areas of memory, even if one area deals with large words and another deals with small. It also has the problem of all things that are shared. If one person is using it, another can't. Therefore, if the CPU is accessing program memory, then data memory must be idle and vice versa.

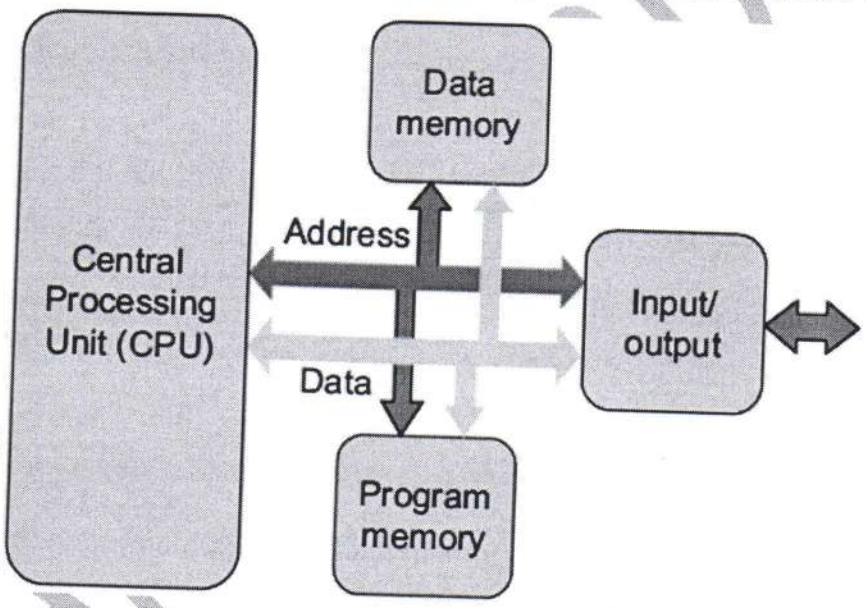


Figure 3.4: Memory accessing based on Von Neumann architecture

An alternative to the Von Neumann structure is seen in Figure 3.5. Every memory area gets its own address bus and its own data bus. Because this structure was invented in the university of the same name, this is called a Harvard structure. In the Harvard approach we get greater flexibility in bus size, but pay for it with a little more complexity. With program memory and data memory each having their own address and data buses, each can be a different size, appropriate to their needs, and data and program can be accessed

simultaneously. On the minus side, the Harvard structure reinforces the distinction between program and data memory, even when this distinction is not wanted. This disadvantage may be experienced, for example, when data is stored in program memory as a table, but is actually needed in the data domain.

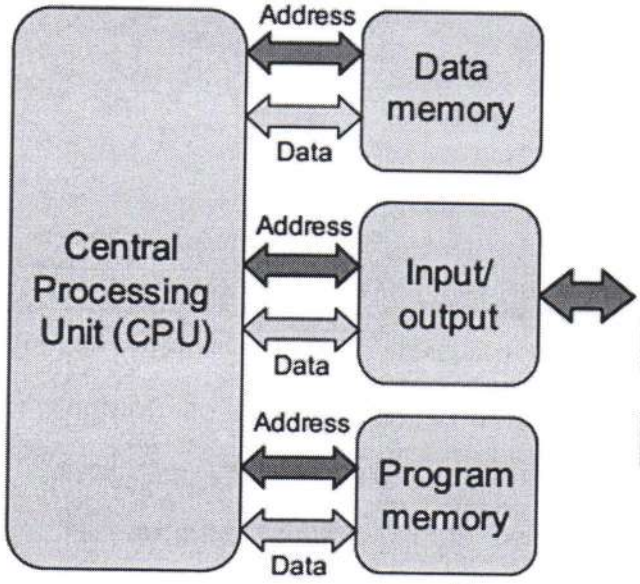


Figure 3.5: Memory accessing based on Harvard architecture

3.4 Complex Instruction Set Computer and the Reduced Instruction Set Computer

Any CPU has a set of instructions that it recognises and responds to; all programs are built up in one way or another from this instruction set. We want computers to execute code as fast as possible, but how to achieve this aim is not always an obvious matter. One approach is to build sophisticated CPUs with vast instruction sets, with an instruction ready for every foreseeable operation. This leads to the CISC, the Complex Instruction Set Computer. A CISC has many instructions and considerable sophistication. Yet the complexity of the design needed to achieve this tends to lead to slow operation. One characteristic of the CISC approach is that instructions have different levels of complexity. Simple ones can be expressed in a short instruction code, say one byte of data, and execute quickly. Complex ones may need several bytes of code to define

them and take a long time to execute. Another approach is to keep the CPU very simple and have a limited instruction set. This leads to the RISC approach – the Reduced Instruction Set Computer. The instruction set, and hence overall design, is kept simple. This leads to fast operation. One characteristic of the RISC approach is that each instruction is contained within a single binary word. That word must hold all information necessary, including the instruction code itself, as well as any address or data information also needed. A further characteristic, an outcome of the simplicity of the approach, is that every instruction normally takes the same amount of time to execute.

3.5 8, 16, or 32 Bits?

People are usually confused for making a decision between 8, 16, or 32 bits of microcontrollers. It is important to realize that the number of bits just refers to the width of the data handled by the processor. This number actually limits the precision of mathematical operations carried out by the CPU.

In general, 8-bit microcontrollers have been around since the first days of the microcontroller development. They are cheap, easy to use (only small package size), low speed, and can be used in most general-purpose control and data manipulation operations. For example, it is still very efficient to design low- to medium-speed control systems (e.g. temperature control,

fluid-level control, or robotics applications) using 8-bit microcontrollers. In such applications, low cost is more important than high speed. Many commercial and industrial applications fall into this category and can easily be designed using standard 8-bit microcontrollers.

Microcontrollers of 16 and 32 bit on the other hand usually cost more, but they offer much higher speeds, and much higher precision in mathematical operations. These microcontrollers are usually housed in larger packages (e.g. 64 or 100 pins) and offer much more features, such as larger data and program memories, more timer/counter modules, more and faster A/D channels, more I/O ports, and so on. Microcontrollers of 32 bit are usually used in high-speed,

real-time digital signal processing applications, where also high precision is a requirement, such as digital image processing, digital audio processing, and so on. Most consumer products, such as electronic games and mobile phones, are based on 32-bit processors as they demand high-speed real-time operation with colour graphical displays and with touch-screen panels. Other high-speed applications such as video capturing, image filtering, video editing, video streaming, speech recognition, and speech processing all require very fast 32-bit processors with lots of data and program memories, and very high precision while implementing the digital signal processing algorithms.

Lecture Four

4.1 Programming Languages

The inputs that cause a microprocessor to perform a specific action are called **instructions** and the collection of instructions that the microprocessor will recognize is its **instruction set**. The form of the instruction set depends on the microprocessor concerned. The series of instructions that are needed to carry out a particular task is called a **program**.

Microprocessors work in binary code; however, instructions written in binary code are referred as **machine code**. Writing a program in such a code requires good skills and it is very tedious process. It is prone to errors because the program is just a series of 0s and 1s and the instructions are not easily understood from just looking at the patterns. An alternative way is to use an easy to understand form of shorthand code for the patterns of 0s and 1s. for example, the operation of adding data might be represented by just ADDA. Such a shorthand code is referred to as a **mnemonic code.** The term **assembly language** is used for such a code. Writing a program using assembly language is easier because they are an abbreviated version of the operation performed by the instruction. Also, using assembly language is less likely to be disposed to errors than the binary patterns of machine code programming. However, the program written in assembly language is still needed to be converted into machine code since it is the only code that the microprocessor will recognize. This conversion can be achieved by hand using manufacture's datasheets, which list the binary code for each assembly code. However, computer programs are available to do the conversion, such programs are referred to as **assembler programs**.

High-level languages are available, which provide a type of programming language that are easily describing the required operation. Examples of such languages are **BASIC**, **C**, **FORTRAN**, and **PASCAL**. Such languages have still required to be converted into machine code, by a computer program, in order for the microprocessor to be able to use. Additionally, programs written in high-level languages usually need more memory to store them when they have been converted into machine code and thus tend to take longer to run than programs written in assembly language.

4.2 The Five Programming Steps

Every program that we want to design can be reduced to five basic program steps, these are:

1. **Initialization Step**

   The purpose of the Initialization Step is to establish the environment in which the program will run.

2. **Input Step**

   Almost every computer program has a task that is designed to take some existing state of information, process it in some way, and show the new state of that information.

3. **Process Step**

   once the input from the sensors is received, some parts of the code must be responsible for determining whether the sensors are detecting what it used for or not.

4. **Output Step**

   After the Process Step has finished its work, the new value is typically output on some display device.

5. **Termination Step**

   The Termination Step has the responsibility of "cleaning up" after the program is finished performing its task.

4.3 Programming Arduino Microcontroller

Arduino is an open-source microcontroller that enables programming and interaction; it is programmed in C/C++ with an Arduino library to allow it to access the hardware. This allows for more flexible programmability and the ability to use electronics that can interface with Arduino. The basic Arduino programming functions are discussed in the following.

structure

The basic structure of the Arduino programming language is fairly simple and runs in at least two parts. These two required parts, or functions, enclose blocks of statements.

```
void setup ()
{
statements;
}
void loop ()
{
statements;
}
```

Where **setup ()** is the preparation, **loop ()** is the execution. Both functions are required for the program to work. The **setup function** should follow the declaration of any variables at the very beginning of the program. It is the first function to run in the program, is run only once, and is used to set **pinMode** or initialize **serial communication**.

The **loop function** follows next and includes the code to be executed continuously – reading inputs, triggering outputs, etc. This function is the core of all Arduino programs.

setup ()

The **setup ()** function is called once when your program starts. Use it to initialize pin modes, or begin serial. It must be included in a program even if there are no statements to run.

```
void setup ()
{
pinMode (pin, OUTPUT);     // sets the 'pin' as output
}
```

loop ()

After calling the **setup ()** function, the **loop ()** function does precisely what its name suggests, and loops consecutively, allowing the program to change, respond, and control the Arduino board.

```
void loop ()
{
digitalWrite (pin, HIGH);     // turns 'pin' on
delay (1000);                 // pauses for one second
digitalWrite (pin, LOW);      // turns 'pin' off
delay (1000);                 // pauses for one second
}
```

functions

A function is a block of code that has a name and a block of statements that are executed when the function is called. The functions **void setup()** and **void loop()** have already been discussed and other built-in functions will be discussed later.

Custom functions can be written to perform repetitive tasks and reduce clutter in a program. **Functions are declared by first declaring the function type. This is the type of value to be returned by the function such as 'int' for an**

integer type function. If no value is to be returned the function type would be void. After type, declare the name given to the function and in parenthesis any parameters being passed to the function.

```
type functionName (parameters)
{
statements;
}
```

The following integer type function **delayVal()** is used to set a delay value in a program by reading the value of a potentiometer. It first declares a local variable v, sets v to the value of the potentiometer which gives a number between 0-1023, then divides that value by 4 for a final value between 0-255, and finally returns that value back to the main program.

```
int delayVal()
{
int v;                          // create temporary variable 'v'
 v = analogRead (pot);          // read potentiometer value
v /= 4;                         // converts 0-1023 to 0-255
return v;                       // return final value
}
```

{} curly braces

Curly braces (also referred to as just "braces" or "curly brackets") define the beginning and end of function blocks and statement blocks such as the **void loop ()** function and the for and if statements.

```
type function()
{
statements;
```

}

An opening curly brace { must always be followed by a closing curly brace }. This is often referred to as the braces being balanced. Unbalanced braces can often lead to cryptic, impenetrable compiler errors that can sometimes be hard to track down in a large program.

The Arduino environment includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

; semicolon

A semicolon must be used to end a statement and separate elements of the program. A semicolon is also used to separate elements in a for loop.

```
int x = 13; // declares variable 'x' as the integer 13
```

Note: Forgetting to end a line in a semicolon will result in a compiler error. The error text may be obvious, and refer to a missing semicolon, or it may not. If an impenetrable or seemingly illogical compiler error comes up, one of the first things to check is a missing semicolon, near the line where the compiler complained.

/*... */ block comments

Block comments, or multi-line comments, are areas of text ignored by the program and are used for large text descriptions of code or comments that help others understand parts of the program. They begin with /* and end with */ and can span multiple lines.

```
/*  this is an enclosed  block  comment don' t forget the closing
comment - they have to  be balanced!
```

```
*/
```

Because comments are ignored by the program and take no memory space they should be used generously and can also be used to "comment out" blocks of code for debugging purposes.

// line comments

Single line comments begin with // and end with the next line of code. Like block comments, they are ignored by the program and take no memory space.

```
// this is a single line comment
```

Single line comments are often used after a valid statement to provide more information about what the statement accomplishes or to provide a future reminder.

variables

A variable is a way of naming and storing a numerical value for later use by the program. As their namesake suggests, variables are numbers that can be continually changed as opposed to constants whose value never changes. A variable need to be declared and optionally assigned to the value needing to be stored. The following code declares a variable called inputVariable and then assigns it the value obtained on analog input pin 2:

```
int inputVariable = 0;                  // declares a variable and
                                        // assigns value of 0
inputVariable = analogRead (2);         // set variable to value of
                                        // analog pin 2
```

'inputVariable' is the variable itself. The first line declares that it will contain an **int**, short for integer. The second line sets the variable to the value at **analog pin 2**. This makes the value of pin 2 accessible elsewhere in the code.

Once a variable has been assigned, or re-assigned, you can test its value to see if it meets certain conditions, or you can use its value directly. As an example to illustrate three useful operations with variables, the following code tests whether the **inputVariable** is less than 100, if true it assigns the value 100 to **inputVariable**, and then sets a delay based on **inputVariable** which is now a minimum of 100:

```
if (inputVariable < 100) // tests variable if less than 100
{
inputVariable = 100;   // if true assigns value of 100
}
delay(inputVariable);   // uses variable as delay
```

Note: Variables should be given descriptive names, to make the code more readable. Variable names like **tiltSensor** or **pushButton** help the programmer and anyone else reading the code to understand what the variable represents. Variable names like **var** or **value**, on the other hand, do little to make the code readable and are only used here as examples. A variable can be named any word that is not already one of the keywords in the Arduino language.

variable declaration

All variables have to be declared before they can be used. Declaring a variable means defining its value type, as in **int, long, float**, etc., setting a specified name, and optionally assigning an initial value. This only needs to be done once in a program but the value can be changed at any time using arithmetic and various assignments.

The following example declares that **inputVariable** is an **int**, or integer type, and that its initial value equals zero. This is called a simple assignment.

```
int inputVariable = 0;
```

A variable can be declared in a number of locations throughout the program and where this definition takes place determines what parts of the program can use the variable.

variable scope

A variable can be declared at the beginning of the program before **void setup()**, **locally inside of functions, and sometimes within a statement block such as for loops**. Where the variable is declared determines the variable scope, or the ability of certain parts of a program to make use of the variable.

A global variable is one that can be seen and used by every function and statement in a program. This variable is declared at the beginning of the program, before the **setup ()** function.

A local variable is one that is defined inside a function or as part of a for loop. It is only visible and can only be used inside the function in which it was declared. It is therefore possible to have two or more variables of the same name in different parts of the same program that contain different values. Ensuring that only one function has access to its variables simplifies the program and reduces the potential for programming errors.

The following example shows how to declare a few different types of variables and demonstrates each variable's visibility:

```
int value;          // 'value' is visible
                    // to any function
void setup()
{
```

```
        // no  setup needed
}
void loop()
{
for (int  i=0; i<20;)                    // 'i' is only visible
{                                        // inside the for-loop i++;
}
float f;                                 // 'f' is only visible
}                                        // inside loop
```

byte

Byte stores an 8-bit numerical value without decimal points. They have a range of 0-255.

```
byte  someVariable =  180;    // declares 'someVariable'
                              // as a byte type
```

int

Integers are the primary datatype for storage of numbers without decimal points and store a 16-bit value with a range of 32,767 to -32,768.

```
int someVariable =  1500;     // declares 'someVariable'
                              // as an integer  type
```

Note: Integer variables will roll over if forced past their maximum or minimum values by an assignment or comparison. For example, if x = 32767 and a subsequent statement adds 1 to x, x = x + 1 or x++, x will then rollover and equal -32,768.

long

Extended size datatype for long integers, without decimal points, stored in a 32-bit value with a range of 2,147,483,647 to -2,147,483,648.

```
long someVariable = 90000; // declares 'someVariable'
                           // as a long type
```

float

A datatype for floating-point numbers, or numbers that have a decimal point. Floating- point numbers have greater resolution than integers and are stored as a 32-bit value with a range of 3.4028235E+38 to -3.4028235E+38.

```
float someVariable = 3.14;   // declares 'someVariable'
                             // as a floating-point type
```

Note: Floating-point numbers are not exact, and may yield strange results when compared. Floating point math is also much slower than integer math in performing calculations, so should be avoided if possible.

Variable Names in C

A keyword is any word that has special meaning to the C compiler and they cannot be used for variable or function names. There are three general rules for naming variables or functions in C, however, valid variable names may contain:

1. Characters a through z and A through Z
2. The underscore character (_)
3. Digit characters 0 through 9, provided they are not used as the first character in the name.

Valid variable names might include:

jane Jane ohm ampere volt money day1 Week50 _system XfXf

Using the same rules, the following would not be valid names:

^create 4March -positive @URL %percent not-Good This&That which?

Some good examples of variable are:

myFriend **toggleLED** **reloadEmptyPaperTray** **closeDriveDoor**

arrays

An array is a collection of values that are accessed with an index number. Any value in the array may be called upon by calling the name of the array and the index number of the value. Arrays are zero indexed, with the first value in the array beginning at index number 0. An array needs to be declared and optionally assigned values before they can be used.

```
int myArray [] = {value0, value1, value2...}
```

Likewise, it is possible to declare an array by declaring the array type and size and later assign values to an index position:

```
int myArray[5];          // declares integer array w/ 6 positions
myArray[3] = 10;         // assigns the 4th index the value 10
```

To retrieve a value from an array, assign a variable to the array and index position:

```
x = myArray[3]; // x now equals 10
```

Arrays are often used in for loops, where the increment counter is also used as the index position for each array value. The following example uses an array to flicker an LED. Using a for loop, the counter begins at 0, writes the value contained at index position 0 in the array flicker[], in this case 180, to the PWM pin 10, pauses for 200ms, then moves to the next index position.

```
int ledPin = 10;  // LED on pin 10
```

```
byte flicker [] = {180, 30, 255, 200, 10, 90, 150, 60};
                                    // above array of 8
void setup()                        // different values
{
pinMode (ledPin, OUTPUT);  // sets OUTPUT pin
}
void loop()
{
for (int i=0; i<7; i++)             // loop equals number
{                                   // of values in array
analogWrite(ledPin, flicker[i]);  // write index value
delay(200) ;                        // pause 200ms
}
}
```

arithmetic

Arithmetic operators include addition, subtraction, multiplication, and division. They return the sum, difference, product, or quotient (respectively) of two operands.

```
y = y + 3; x = x - 7; i = j * 6; r = r / 5;
```

The operation is conducted using the data type of the operands, so, for example, 9 / 4 results in 2 instead of 2.25 since 9 and 4 are **ints** and are incapable of using decimal points. This also means that the operation can overflow if the result is larger than what can be stored in the data type.

If the operands are of different types, the larger type is used for the calculation. For example, if one of the numbers (operands) are of the type float and the other of type integer, floating point math will be used for the calculation.

Choose variable sizes that are large enough to hold the largest results from your calculations. Know at what point your variable will rollover and also what

happens in the other direction e.g. (0 - 1) OR (0 - - 32768). For math that requires fractions, use float variables, but be aware of their drawbacks: large size and slow computation speeds.

Note: Use the cast operator e.g. **(int) myFloat** to convert one variable type to another on the fly. For example, i = (int) 3.6 will set i equal to 3.

Lecture Five

Notice: This lecture is a continuation to lecture Four.

5.1 compound assignments

Compound assignments combine an arithmetic operation with a variable assignment. These are commonly found in for loops as described later. The most common compound assignments include:

```
x ++        // same as x = x + 1, or increments x by +1
x --        // same as x = x - 1, or decrements x by -1
x += y      // same as x = x + y, or increments x by +y
x -= y      // same as x = x - y, or decrements x by -y
x *= y      // same as x = x * y, or multiplies x by y
x /= y      // same as x = x / y, or divides x by y
```

Note: For example, **x* = 3** would triple the old value of x and re-assign the resulting value to x.

comparison operators

Comparisons of one variable or constant against another are often used in if statements to test if a specified condition is true. In the examples found on the following pages, ?? is used to indicate any of the following conditions:

```
x == y      // x is equal to y
x != y      // x is not equal to y
x < y       // x is less than y
x > y       // x is greater than y
x <= y      // x is less than or equal to y
```

```
x >= y    // x is greater than or equal to y
```

logical operators

Logical operators are usually a way to compare two expressions and return a TRUE or FALSE depending on the operator. There are three logical operators, AND, OR, and NOT, that are often used in if statements:

Logical AND:

```
if (x > 0 && x < 5)   // true only if both
                      // expressions are true
```

Logical OR:

```
if (x > 0 || y > 0)   // true if either
                      // expression is true
```

Logical NOT:

```
if (!x > 0)           // true only if
                      // expression is false
```

constants

The Arduino language has a few predefined values, which are called constants. They are used to make the programs easier to read. Constants are classified in groups.

true/false

These are Boolean constants that define logic levels. FALSE is easily defined as 0 (zero) while TRUE is often defined as 1, but can also be anything else except zero. So in a Boolean sense, -1, 2, and -200 are all also defined as TRUE.

```
if (b == TRUE);
{
```

```
doSomething;
}
```

high/low

These constants define pin levels as HIGH or LOW and are used when reading or writing to digital pins. HIGH is defined as logic level 1, ON, or 5 volts while LOW is logic level 0, OFF, or 0 volts.

```
digitalWrite (13, HIGH);
```

input/output

Constants used with the pinMode() function to define the mode of a digital pin as either INPUT or OUTPUT.

```
pinMode (13, OUTPUT);
```

if

if statements test whether a certain condition has been reached, such as an analog value being above a certain number, and executes any statements inside the brackets if the statement is true. If false the program skips over the statement. The format for an if test is:

```
if (someVariable ?? value)
{
do Something;
}
```

The above example compares **someVariable** to another value, which can be either a variable or constant. If the comparison, or condition in parentheses is true, the statements inside the brackets are run. If not, the program skips over them and continues on after the brackets.

Note: Beware of accidentally using '=', as in if (x=10), while technically valid, defines the variable x to the value of 10 and is as a result always true. Instead use '==', as in if (x==10), which only tests whether x happens to equal the value 10 or not. Think of '=' as "equals" opposed to '==' being "is equal to".

if... else

if... else allows for 'either-or' decisions to be made. For example, if you wanted to test a digital input, and do one thing if the input went HIGH or instead do another thing if the input was LOW, you would write that this way:

```
if (inputPin == HIGH)
{
doThingA;
}
else
{
doThingB;
}
```

else can also precede another if test, so that multiple, mutually exclusive tests can be run at the same time. It is even possible to have an unlimited number of these else branches. Remember though, only one set of statements will be run depending on the condition tests:

```
if (inputPin < 500)
{
doThingA;
}
else if (inputPin >= 1000)
{
doThingB;
}
```

```
else
{
doThingC;
}
```

Note: An if statement simply tests whether the condition inside the parenthesis is true or false. This statement can be any valid C statement as in the first example, if (inputPin == HIGH). In this example, the if statement only checks to see if indeed the specified input is at logic level high, or +5v.

for

The for statement is used to repeat a block of statements enclosed in curly braces a specified number of times. An increment counter is often used to increment and terminate the loop. There are three parts, separated by semicolons (;), to the for loop header:

```
for (initialization; condition; expression)
{
doSomething;
}
```

The initialization of a local variable, or increment counter, happens first and only once. Each time through the loop, the following condition is tested. If the condition remains true, the following statements and expression are executed and the condition is tested again. When the condition becomes false, the loop ends.

The following example starts the integer i at 0, tests to see if i is still less than 20 and if true, increments i by 1 and executes the enclosed statements:

```
for (int  i=0; i<20; i++)          //     declares i, tests if less
{                                  //      than 20, increments i by  1
```

```
digitalWrite (13, HIGH);    //    turns pin 13  on
delay (250);                //    pauses for 1/4 second
digitalWrite (13, LOW);     //    turns pin 13  off
delay (250);                //    pauses for 1/4 second
}
```

while

while loops will loop continuously, and infinitely, until the expression inside the parenthesis becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

```
while (someVariable ?? value)
{
doSomething;
}
```

The following example tests whether **'someVariable'** is less than 200 and if true executes the statements inside the brackets and will continue looping until 'someVariable' is no longer less than 200.

```
while (someVariable < 200)   // tests if less  than 200
{
doSomething;                 // executes enclosed statements
someVariable++;              // increments variable by  1
}
```

do... while

The do loop is a bottom driven loop that works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run **at least once**.

```
do
{
doSomething;
} while (someVariable ?? value);
```

The following example assigns readSensors () to the variable 'x', pauses for 50 milliseconds, then loops indefinitely until 'x' is no longer less than 100:

```
do
{
x = readSensors ();        // assigns the value of readSensors() to
delay (50);                // pauses 50 milliseconds
} while (x < 100);         // loops if x is less than 100
```

pinMode (pin, mode)

Used in **void setup ()** to configure a specified pin to behave either as an INPUT or an OUTPUT.

```
pinMode (pin, OUTPUT);     // sets 'pin' to output
```

Arduino digital pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode(). Pins configured as INPUT are said to be in a high-impedance state.

digitalRead (pin)

Reads the value from a specified digital pin with the result either HIGH or LOW. The pin can be specified as either a variable or constant (0-13).

```
value = digitalRead (Pin);     // sets 'value' equal to
                               // the input pin
```

digitalWrite (pin, value)

Outputs either logic level HIGH or LOW at (turns on or off) a specified digital pin. The pin can be specified as either a variable or constant (0-13).

```
digitalWrite (pin, HIGH);      // sets 'pin' to high
```

The following example reads a pushbutton connected to a digital input and turns on an LED connected to a digital output when the button has been pressed:

```
int led = 13;        // connect LED to  pin 13
int pin = 7;         // connect pushbutton to  pin 7
int value =  0;      // variable to store the read value

void setup ()
{
pinMode (led, OUTPUT);     // sets pin 13   as output
pinMode (pin,  INPUT);     // sets pin 7  as input
}

void loop ()
{
value =  digitalRead (pin);     //      sets 'value'  equal to
                                //      the input pin
digitalWrite (led, value);      //      sets 'led'  to the
                                //      button's  value
}
```

analogRead (pin)

Reads the value from a specified analog pin with a 10-bit resolution. This function only works on the analog in pins (0-5). The resulting integer values range from 0 to 1023.

```
value = analogRead (pin);     // sets 'value' equal to 'pin'
```

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

analogWrite (pin, value)

Writes a pseudo-analog value using hardware enabled pulse width modulation (PWM) to an output pin marked PWM. On newer Arduinos with the ATmega168 chip, this function works on pins 3, 5, 6, 9, 10, and 11. Older Arduinos with an ATmega8 only support pins 9, 10, and 11. The value can be specified as a variable or constant with a value from 0-255.

```
analogWrite (pin, value);     // writes 'value' to analog 'pin'
```

A value of 0 generates a steady 0 volts output at the specified pin; a value of 255 generates a steady 5 volts output at the specified pin. For values in between 0 and 255, the pin rapidly alternates between 0 and 5 volts - the higher the value, the more often the pin is HIGH (5 volts). For example, a value of 64 will be 0 volts three-quarters of the time, and 5 volts one quarter of the time; a value of 128 will be at 0 half the time and 255 half the time; and a value of 192 will be 0 volts one quarter of the time and 5 volts three-quarters of the time.
Because this is a hardware function, the pin will generate a steady wave after a call to analogWrite in the background until the next call to analogWrite (or a call to digitalRead or digitalWrite on the same pin).

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

The following example reads an analog value from an analog input pin, converts the value by dividing by 4, and outputs a PWM signal on a PWM pin:

```
int led = 10;        // LED with 220 resistor on pin 10
int pin =  0;        // potentiometer on analog pin 0
int value;           // value for reading
void setup (){  } // no setup needed
void loop()
{
value =    analogRead (pin);// sets 'value' equal to 'pin'
value /= 4; // converts 0-1023 to 0-255
analogWrite (led, value);      // outputs PWM signal to led
}
```

delay (ms)

Pauses a program for the amount of time as specified in milliseconds, where 1000 equals 1 second.

```
delay(1000);      // waits for one second
```

Serial.begin (rate)

Opens serial port and sets the baud rate for serial data transmission. The typical baud rate for communicating with the computer is 9600 although other speeds are supported.

```
void setup ()
{
Serial.begin (9600);      // opens serial port
}                         // sets data rate to 9600 bps
```

Serial.println (data)

Prints data to the serial port, followed by an automatic carriage return and line feed. This command takes the same form as **Serial.print ()**, but is easier for reading data on the Serial Monitor.

```
Serial.println (analogValue); // sends the value of
                              // 'analogValue'
```

The following simple example takes a reading from analog pin0 and sends this data to the computer every 1 second.

```
void setup ()
{
Serial.begin (9600);      // sets serial to 9600bps
}

void loop ()
{
Serial.println (analogRead(0)); // sends analog value
delay (1000);                   // pauses for 1 second
}
```


5.2 Libraries

The Arduino programming environment comes with a standard library, a library of functions that are included in every sketch, which are discussed in the above paragraphs. By default, Arduino can handle basic mathematical operations, and set pins to digital or analog input and output, but it cannot write data to an SD card, connect to WiFi, or use a LCD. These devices that are not

standard on Arduino boards. Of course, an Arduino can use these devices when they are available, but to use these devices, a library for the specific device must be imported into a sketch. Otherwise, there is no point in having the extra functionality that could potentially take up space on a device where space is critical. Adding a library to your sketch adds more functionality and allows you, the programmer, to use new functions. For example, by importing the EEPROM library, you can access the internal EEPROM by using two new functions: read () and write ().

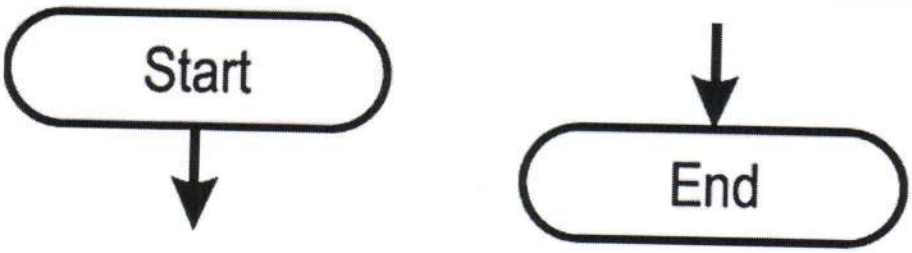
5.3 Software Design Using Flowcharts

Flowcharts are one of the oldest and most popular tools used to express an algorithm, regardless of whether it is a software or a hardware algorithm. Probably the reason flowcharts are so popular is because they give a general picture of the algorithm, as opposed to the sometimes obscure picture provided by other tools. Although, as mentioned before, flowcharts are used to express an algorithm that could later be implemented in either software or hardware. **Flowcharts are not intended to include every detail of the implementation of the algorithm because that is left to the actual coding of the instructions in the particular programming language of choice**. Thus, flowcharts are expected to give a general idea of the algorithm and it will help to keep this in mind when developing algorithms using flowcharts.

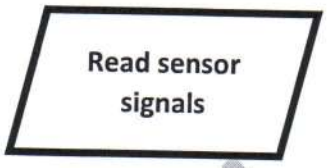
5.3.1 Used Symbols in Flowcharts Design

There are five (5) basic symbols used to develop flowcharts; these are:

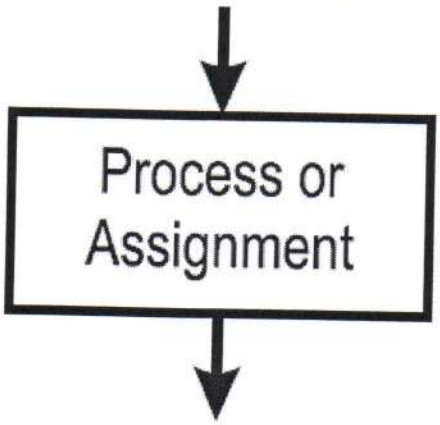
1- **Start/Finish:** The Start/Finish symbol consists of an oval shape as illustrated in figure below. As its name implies, the Start/Finish symbol is used to show an entry point into the algorithm or an exit point out of the algorithm.



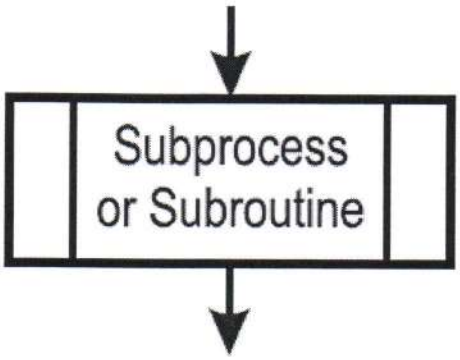
2- **Input/output:** Used for input and output operation; it is represented in the figure below.



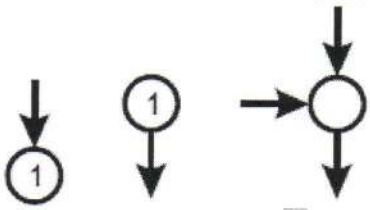
3- **Assignment or process:** This is drawn as a rectangle and inside of the rectangle we indicate the corresponding arithmetic or logic assignments including any expressions.



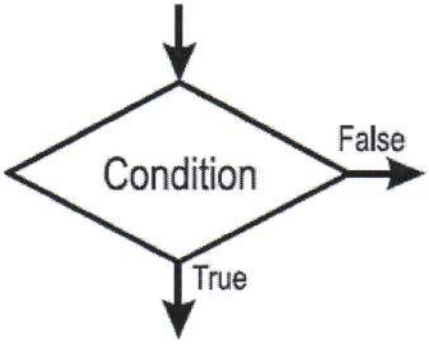
4- **Sub-process:** The Sub-process symbol is shown in the figure below. It looks a lot like the assignment symbol since it is also a rectangle, but it includes an additional vertical line on both the left and right side. This symbol is used to indicate whenever a subroutine or function call is invoked. After the sub-process returns, the algorithm continues with the next symbol in sequence in the flowchart.



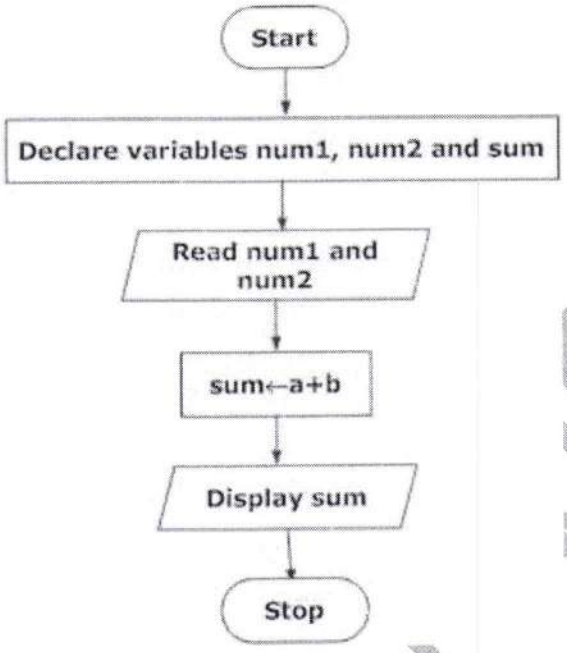
5- **Connector:** The Connector symbol shown in the figure below is used whenever we run out of space in the page where we are developing our flowchart and need to continue on another part of the page or on a completely different page.



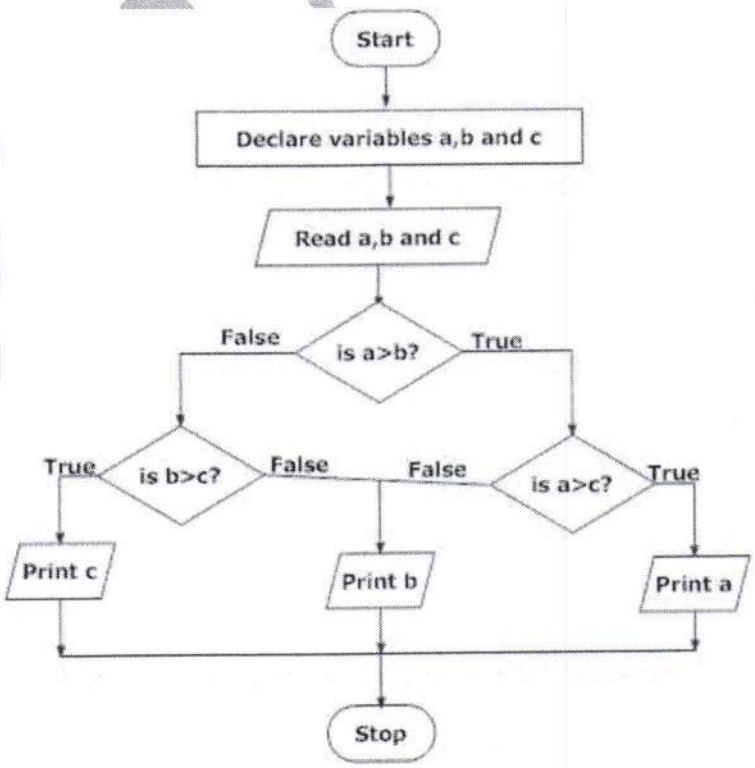
6- **Decision**: Its shape resembles a diamond and, as its name implies, it is used to alter the flow of the algorithm based on the value of some variable after a particular decision is made. The ability to make decisions allow for powerful algorithms as opposed to purely sequential ones. Decision blocks are the only ones that feature two outputs, one corresponding to each of the two possible decision outcomes: true or false, yes or no, left or right, etc.



Example 5.1: Draw a flowchart to add two numbers entered by user.



Example 5.2: Draw flowchart to find the largest among three different numbers entered by user.



Lecture Six

6.1 Number Systems

The efficient use of a microprocessor or microcontroller requires a working knowledge of binary, decimal, and hexadecimal numbering systems. Number systems are classified according to their bases. The numbering system used in everyday life is base 10 or the decimal number system. The most commonly used numbering system in microprocessor and microcontroller applications is base 16, or hexadecimal. In addition, base 2 (binary), or base 8 (or octal) number systems are also used.

6.1.1 Decimal Number System

As you all know, the numbers in this system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We can use the subscript 10 to indicate that a number is in decimal format. For example, we can show decimal number 235 as 235_{10}. In general, a decimal number is represented as follows:

$$a_n \times 10^n + a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} + \cdots\cdots + a_0 \times 10^0$$

For example, decimal number $\mathbf{825_{10}}$ can be shown as follows:

$$825_{10} = 8 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$$

Similarly, decimal number $\mathbf{26_{10}}$ can be shown as follows:

$$26_{10} = 2 \times 10^1 + 6 \times 10^0$$

Or

$$3359_{10} = 3 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 9 \times 10^0$$

6.1.2 Binary Number System

In binary number system, there are two numbers: 0 and 1.We can use the subscript 2 to indicate that a number is in binary format. For example, we can show binary number 1011 as 1011_2. In general, a decimal number is represented as follows:

$$a_n \times 2^n + a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots\cdots + a_0 \times 2^0$$

For example, binary number $\mathbf{1110_2}$ can be shown as follows:

$$1110_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

Similarly, binary number 10001110_2 can be shown as follows:

$$10001110_2 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

Each one or zero in the binary representation is called a **bit**. A collection of 8-bits is called a **byte** and a collection of 4-bits is called a **nibble**. Two nibbles make a **byte**. The bit associated with the highest power of two is called the most significant bit (**MSB**), and the bit associated with the lowest power of two is the least significant bit (**LSB**). The 4-bit nibble forms the basis of hex numbers, which will be discussed a little later.

| 1 |

bit

| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

byte (1 byte = 8-bits)

| 1 | 0 | 0 | 1 |

nibble (hex number) (1 nibble = 4-bits)

| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

2 nibbles = 1 byte

6.1.3 *Octal Number System*

In octal number system, the valid numbers are 0, 1, 2, 3, 4, 5, 6, and 7. We can use the subscript 8 to indicate that a number is in octal format. For example, we can show octal number 23 as 23_8.

In general, an octal number is represented as follows:

$$a_n \times 8^n + a_{n-1} \times 8^{n-1} + a_{n-2} \times 8^{n-2} + \cdots\cdots + a_0 \times 8^0$$

For example, octal number 237_8 can be shown as follows:

$$237_8 = 2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0$$

Similarly, octal number 17778 can be shown as follows:

$$1777_8 = 1 \times 8^3 + 7 \times 8^2 + 7 \times 8^1 + 7 \times 8^0$$

6.1.4 *Hexadecimal Number System*

In hexadecimal number system, the valid numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. We can use the subscript 16 of H to indicate that a number is in hexadecimal format. For example, we can show hexadecimal number 1F as $1F_{16}$ or as $1F_H$.

In general, a hexadecimal number is represented as follows:

$$a_n \times 16^n + a_{n-1} \times 16^{n-1} + a_{n-2} \times 16^{n-2} + \cdots\cdots + a_0 \times 16^0$$

For example, hexadecimal number $2AC_{16}$ can be shown as follows:

$$2AC_{16} = 2 \times 16^2 + 10 \times 16^1 + 12 \times 16^0$$

Similarly, hexadecimal number 3FFE16 can be shown as follows:

$$3FFE_{16} = 3 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 14 \times 16^0$$

6.2 Converting Binary Numbers into Decimal

To convert a binary number into decimal, write the number as the sum of the powers of 2.

Example 1.1

Convert binary number 1011_2 into decimal.

Solution:

Write the number as the sum of the powers of 2:

$$\begin{aligned} 1011_2 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 8 + 0 + 2 + 1 \\ &= 11 \end{aligned}$$

or, $1011_2 = 11_{10}$

Example 1.2

Convert binary number 11001110_2 into decimal.

Solution:

Write the number as the sum of the powers of 2 as follows:

$$\begin{aligned} 11001110_2 &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 128 + 64 + 0 + 0 + 8 + 4 + 2 + 0 \\ &= 206 \end{aligned}$$

or, $11001110_2 = 206_{10}$

6.3 Converting Decimal Numbers into Binary

To convert a decimal number into binary, divide the number repeatedly by two and take the remainders. The first remainder is the least significant digit (LSD), and the last remainder is the most significant digit (MSD).

Example 1.3

Convert decimal number 2810 into binary.

Solution:

Divide the number by two repeatedly and take the remainders:

```
28/2 → 14  Remainder 0 (LSD)
14/2 → 7   Remainder 0
7/2  → 3   Remainder 1
3/2  → 1   Remainder 1
1/2  → 0   Remainder 1 (MSD)
```

The required binary number is 11100_2.

Example 1.4

Convert decimal number 6510 into binary.

Solution:

Divide the number by two repeatedly and take the remainders:

```
65/2 → 32  Remainder 1 (LSD)
32/2 → 16  Remainder 0
16/2 → 8   Remainder 0
8/2  → 4   Remainder 0
4/2  → 2   Remainder 0
2/2  → 1   Remainder 0
1/2  → 0   Remainder 1 (MSD)
```

The required binary number is 1000001_2.

Example 1.5

Convert decimal number 12210 into binary.

Solution:

Divide the number by two repeatedly and take the remainders:

```
122/2 → 61  Remainder 0 (LSD)
61/2  → 30  Remainder 1
30/2  → 15  Remainder 0
15/2  → 7   Remainder 1
7/2   → 3   Remainder 1
3/2   → 1   Remainder 1
1/2   → 0   Remainder 1 (MSD)
```

The required binary number is 1111010_2.

6.4 Converting Binary Numbers into Hexadecimal

Binary numbers quickly become long and hard to remember. For this reason, it is more convenient to convert the binary values into hexadecimal numbers (hex). The reason hex notations are used is that it allows for a one to one correspondence between the 4-bit binary nibble and a single hexadecimal value. If a binary number is broken down into 4-bit nibbles, then each nibble can be replaced with the corresponding hexadecimal number, and the compression is complete. If the number cannot be divided exactly into groups of four, insert zeroes to the left-hand side of the number.

Example 1.6

Convert binary number 10011111_2 into hexadecimal.

Solution:

First, divide the number into groups of four and then find the hexadecimal equivalent of each group:

$$\begin{array}{l} 10011111 = 1001\ 1111 \\ \qquad\qquad\quad\ \ 9 \quad F \end{array}$$

The required hexadecimal number is $9F_{16}$

Example 1.7

Convert binary number 1110111100001110_2 into hexadecimal.

Solution:

First, divide the number into groups of four and then find the equivalent of each group:

$$\begin{array}{l} 1110111100001110 = 1110\ 1111\ 0000\ 1110 \\ \qquad\qquad\qquad\qquad\quad\ \ E \quad F \quad 0 \quad E \end{array}$$

The required hexadecimal number is $EF0E_{16}$.

Example 1.8

Convert binary number 111110_2 into hexadecimal.

Solution:

Since the number cannot be divided exactly into groups of four, we have to insert zeroes to the left of the number:

$$111110 = 0011\ 1110$$
$$3\quad E$$

The required hexadecimal number is $3E_{16}$.

Table 1.2 shows the hexadecimal and binary equivalents of numbers 0 to 31. One notes that the numbers in Table 1.2 are all positive, i.e., they are ***unsigned integers***. If one also wants to consider negative numbers, then one would have to deal with ***signed integers***, which are not discussed in this course.

Table 1.2: Hexadecimal and Binary Equivalents of Decimal Numbers

Decimal (Base 10)	Hexadecimal (Base 16)	Binary (Base 2)	Decimal (Base 10)	Hexadecimal (Base 16)	Binary (Base 2)
0	0	0000	16	10	00010000
1	1	0001	17	11	00010001
2	2	0010	18	12	00010010
3	3	0011	19	13	00010011
4	4	0100	20	14	00010100
5	5	0101	21	15	00010101
6	6	0110	22	16	00010110
7	7	0111	23	17	00010111
8	8	1000	24	18	00011000
9	9	1001	25	19	00011001
10	A	1010	26	1A	00011010
11	B	1011	27	1B	00011011
12	C	1100	28	1C	00011100
13	D	1101	29	1D	00011101

14	E	1110	30	1E	00011110
15	F	1111	31	1F	00011111

6.5 Converting Hexadecimal Numbers into Binary

To convert a hexadecimal number into binary, write the 4-bit binary equivalent of each hexadecimal digit.

Example 1.9

Convert hexadecimal number $A9_{16}$ into binary.

Solution:

Writing the binary equivalent of each hexadecimal digit:

$A = 1010_2$ $9 = 1001_2$

The required binary number is 10101001_2.

Example 1.10

Convert hexadecimal number $FE3C_{16}$ into binary.

Solution:

Writing the binary equivalent of each hexadecimal digit:

$F = 1111_2$ $E = 1110_2$ $3 = 0011_2$ $C = 1100_2$

The required binary number is 1111111000111100_2.