Lecture 3:
A Top-Level View Of Computer Function & Interconnection

Computer Components

As discussed in Lecture 01, all contemporary computers are based on concepts developed by John von Neumann referred to as the *von Neumann architecture*. It has three key concepts

- data and instructions are stored in a single read-write memory.
- the contents of this memory are addressable by location, without regard to the type of data contained there.
- execution occurs in a sequential fashion, unless it is explicitly modified.

The reasoning behind these concepts is that there are small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data.

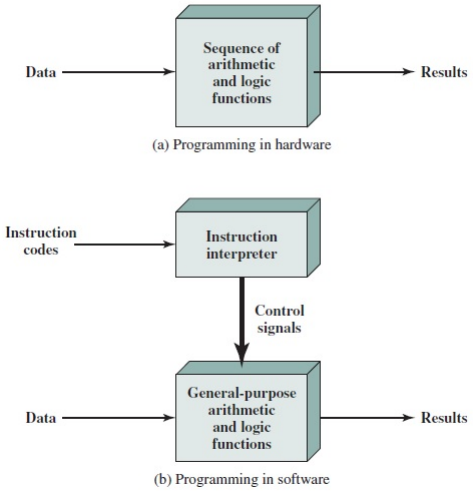
There are two approaches for designing a computer. For the first approach, if there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a *hardwired program*.

For the second approach, suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware.

With the former approach of customizing hardware, the system accepts data and produces results. With the latter approach of general-purpose hardware, the system accepts data and control signals, and then,produces results. The significant difference between the approaches is that the second approach does not need to reconfigure the hardware for new programs. Instead the programmer only needs to supply a new set of control signals.

For the second approach, the control signals should be simple but subtle. So an entire program can be a sequence of steps such that for each step some arithmetic or logical operation is performed on some data. Which means a new set of control signals are needed for each step. Hence, we should provide a unique code for each possible set of control signals. Furthermore, we should add to the general-purpose hardware a segment that can accept a code and generate control signals.

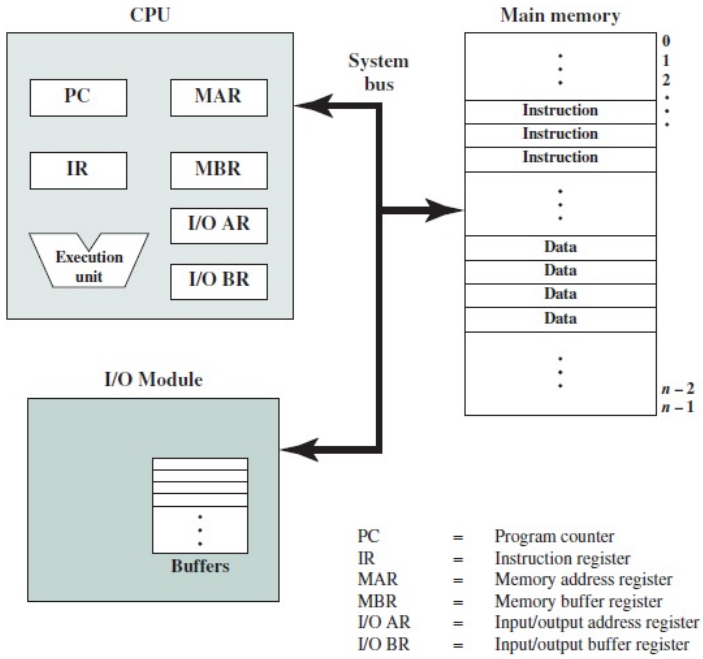
Therefore, with a few conditions, the second approach is significantly easier to implement than the first approach. Each code is an instruction and part of the hardware interprets them and generate control signals. This method of programming is called *software*.



(a) Programming in hardware

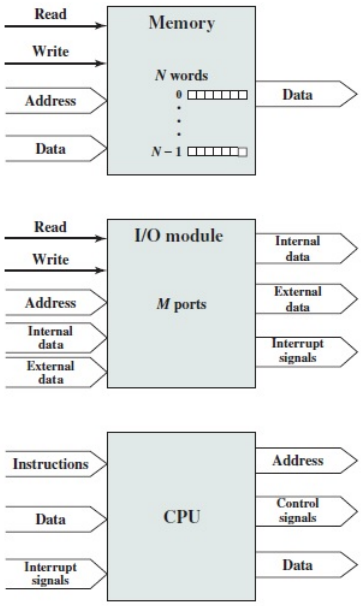
(b) Programming in software

The second diagram in the above figure indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic function. These two components constitute the CPU. However, several other components are needed to yield a functioning computer. Data and instructions must be put in the computer; meanwhile, the computer must be able to report results as well. These capabilities are accomplished with the use of an input module and output module respectively; ultimately, together they form the *I/O module*. Last, the program is not invariably executed sequentially. Thus there must be a place to temporarily store both instruction and data. This module is called *memory*, or *main memory*.



Interconnection Structure

Computers consists of a set of components or modules of three basic types (processor, memory, and I/O) that communicate with each other. Thus, there must be a path for connecting the modules. A collection of paths connecting the various modules is called the *interconnection structure*. The design of the structure will depend on the exchanges that must be made among the modules. The figure below suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type.



For the memory module, it will consist of a predetermined number of words, let us say N, of equal length. Each word is assigned a unique numerical address (typically $0, \ldots, N-1$). A word of data can be read from or written into memory. The nature of the operation is indicated by the read and write control signals. The location for the operation is specified by an address.

For the I/O module from an internal (to the computer system) point of view, it is functionally similar to memory. There are two operations; read and write. Furthermore, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a *port* and give each of them a unique address ($0, \ldots, M-1$ where M is the total number of ports). In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send *interrupt* signals, which are control signals that pauses the normal sequence of operations of the processor.

For the processor module, it reads in instructions and data, writes out data after processing, and use control signals to control the overall operation of the system. It also receives interrupt signals.

Hence, the interconnection structure must support the following types of transfers:

- **Memory to Processor**: The processor reads an instruction or a unit of data from memory.
- **Processor to Memory**: The processor writes a unit of data to memory.
- **I/O to Processor**: The processor reads data from an I/O device via an I/O module.
- **Processor to I/O**: The processtor send data to an I/O device.
- **I/O to or from Memory**: An I/O module is allowed to directly exchange data with memory without going through the processor, using direct memory access.

The two most common interconnection structures are the *bus* and various multiple bus structures, and *point-to-point* interconnection structures with packetized data transfer.

Bus Interconnection

The bus was the dominant means of component interconnection for decades until multicore computers became mainstream. Although general-purpose computers have gradually given way to various point-to-point interconnection structures which now dominates computer system design, buses are commonly used for embedded systems, in particular, microcontrollers. A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to

the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. However, if two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or *lines*. Each line is capable of transmitting a signal representing a binary 1 or a binary 0. Over time, a sequence of bunary digits can be transmitted across a single line. But together, serveral lines of the bus can be used to transmit binary digits simultaneously (in parallel). For instance, an 8-bit unit of data can be transmitted over eight bus lines.

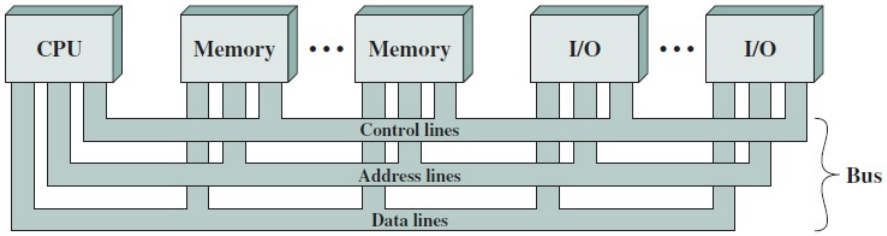
A computer system contains a number of different buses that provides pathways between components. The bus that connects the major computer conmponents (processor, memory, and I/O) is called a *system bus*. A system bus consists, typically, of from about fifty to hundreds of separate lines. In general, each line is assigned a particular meaning or function. On any bus, the lines can be classified into three functional groups: data, address, and control lines. Additionally, there may be power distribution lines that supply power to the attached modules.

The data lines provide a path for moving data amoung the system modules. These lines, collectively, are called the *data bus*. The data bus may consist of 32, 64, 128 or even more separate lines. The number of lines is referred to as the *width* of the data bus. Since each line can carry a single but at a time, the number of lines determines how many bits can be transferred at a time. For instance, if the width of the data bus is 8 and a unit of data is 16-bits, two transmissions will be needed to read the data. Hence, the width of the data bus is the key factor in determining the overall system performance.

The address lines are used to designate the source or destinatin of the data on the data bus. The width of the address bus determines the maximum possible memory capacity of the system. Furthermore, the address bus are generally used to address I/O ports as well. When used for I/O, the higher-order bits are used to select a particular module on the bus, and the lower-order bits selects a memory location or I/O port withing the module.

The control lines are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing information among system modules. The timing signals indicates the validity of the data and address information. The command signals specify operations to be performed. Typical control lines are

- **Memory write**: causes data on the bus to be written into the addressed location.
- **Memory read**: causes data from the addressed location to be placed on the bus.
- **I/O write**: causes data on the bus to be output to the addressed I/O port.
- **I/O read**: causes data from the addressed I/O port to be placed on the bus.
- **Transfer ACK**: indicates that data have been accepted from or placed on the bus.
- **Bus request**: indicates that module needs to gain control of the bus.
- **Bus grant**: indicates that a requesting module has been granted control of the bus.
- **Interrupt request**: indicates that an interrupt is pending.
- **Interrupt ACK**: acknowledges that the pending interrupt has been recognized.
- **Clock**: is used to synchronize operations.
- **Reset**: initializes all modules.

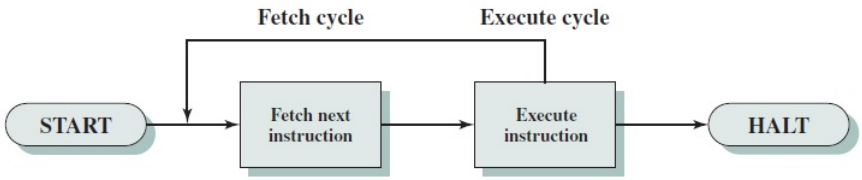


In conclusion, the operation of the bus is as follows

+ If one module wishes to send data to another module, it must do two things:
  - obtain the use of the bus
  - transfer data via the bus
+ If one module wishes to request data from another module, it must
  - obtain the use of the bus
  - transfer a request to the other module over the appropriate control and address lines
  - wait for the another module to send the data

Computer Functions

The basic function performed by a computer is execution of a program, which conists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. In its simplest form, the instruction process consists of two steps; the processor reads (fetches) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution, which may involve several operations and depends on the nature of the instruction. The processing of a single instruction is called an *instruction cycle*. It consists of the *fetch cycle* and the *execute cycle*.



However, a program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer in encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the *program counter* (PC) holds the address of the instruction to be fetched next. If not told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in the sequence. When an instruction is fetched, it is loaded into a register in the processor known as the *instruction register* (IR). The instruction contains bits that specify the action the processor is to perform, which the processor interprets and executes. In general, these actions fall into four categories

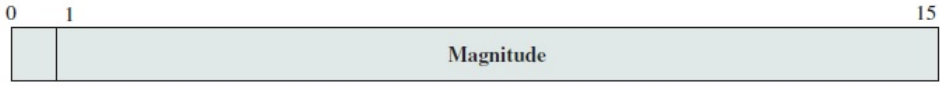
- **Processor-memory**: data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O**: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.

- **Data processing**: The processor may perform some arithmetic or logic operation on data.
- **Control**: An instruction may specify that the sequence of execution be altered.

An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical machine that includes the charactistics listed below

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

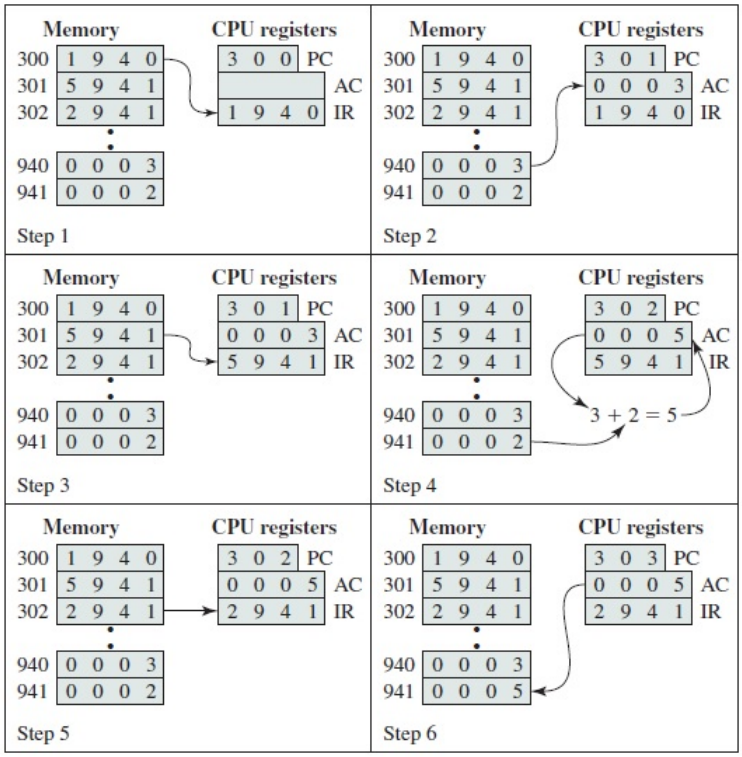
(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

The processor contains a single data register called an accumulator (AC). Both instructions and data are 16-bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides a 4-bit opcode, so there can be at most $2^4 = 16$ different opcodes, and up to $2^{12} = 4096(4K)$ words of memory can be directly addressed.

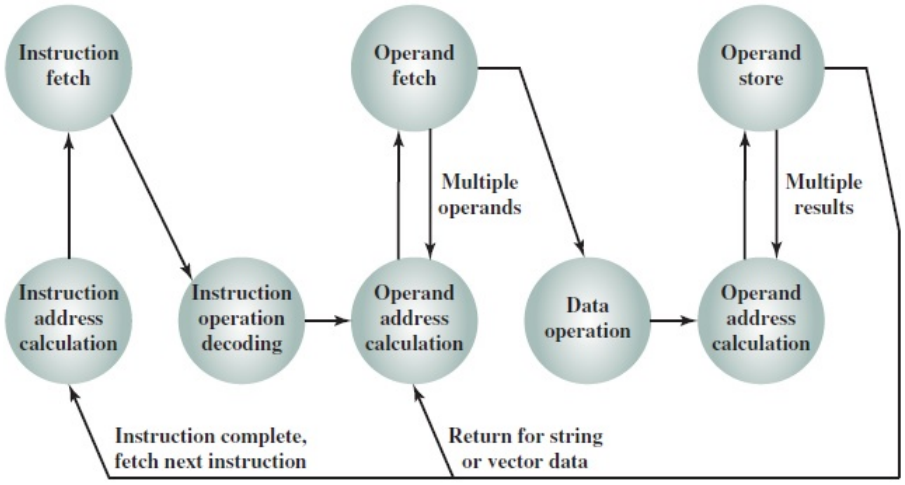
The following figure illustrates a partical program execution, showing the relevant portions f memory and processor registers.



The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register.
2. The first 4 bits in the IR indicate that AC is to be loaded. The maining 12 bits specifiy the address from which data are to be loaded.
3. The next instruction is fetched from location 301, and the PC is incremented.
4. The old content of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of location 941. With a more complex set of instructions, fewer cycles would be needed. Some older processors, for example, included instructions that containe more than one memory address. Thus, the execution cycle for a particular instruction on such processors could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations, the following figure provides a more detailed look at the instruction cycle. The figure is in the form of a state diagram.



For any given instruction cycle, some states may be null and others may be visited more than once. The states can be described as follows:

- **Instruction address calculation (iac)**: Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction.
- **Instruction fetch (if)**: Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod)**: Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac)**: If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.

- **Operand fetch (of)**: Fetch the operand from memory or read it in from I/O.
- **Data operation (do)**: Perform the operation indicated in the instruction.
- **Operand store (os)**: Write the result into memory or out to I/O.

States in the upper part of the above figure involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed. Also notice that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. Last, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) or characters. As indicated in the diagram, repetitive operand fetch andor store operations will be required.