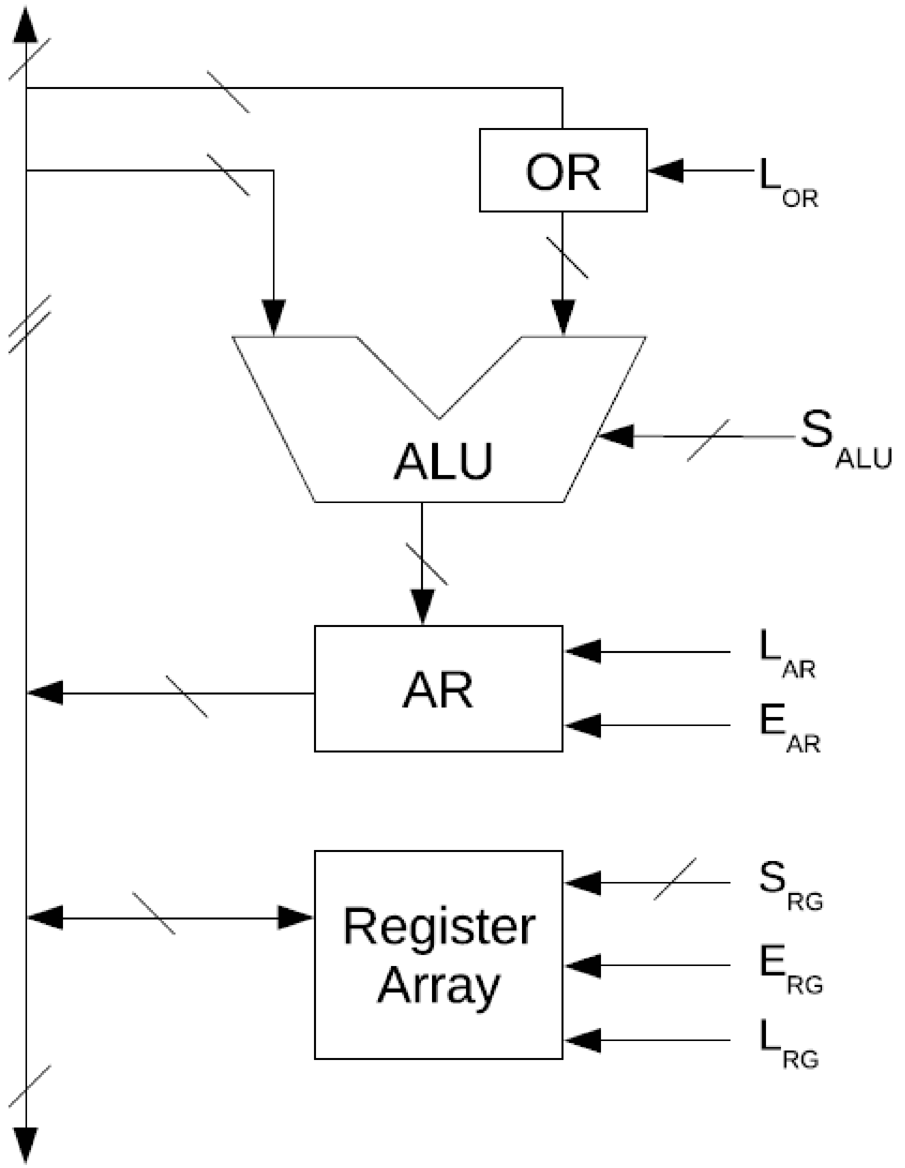
Lecture 19 – Processor design 2

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

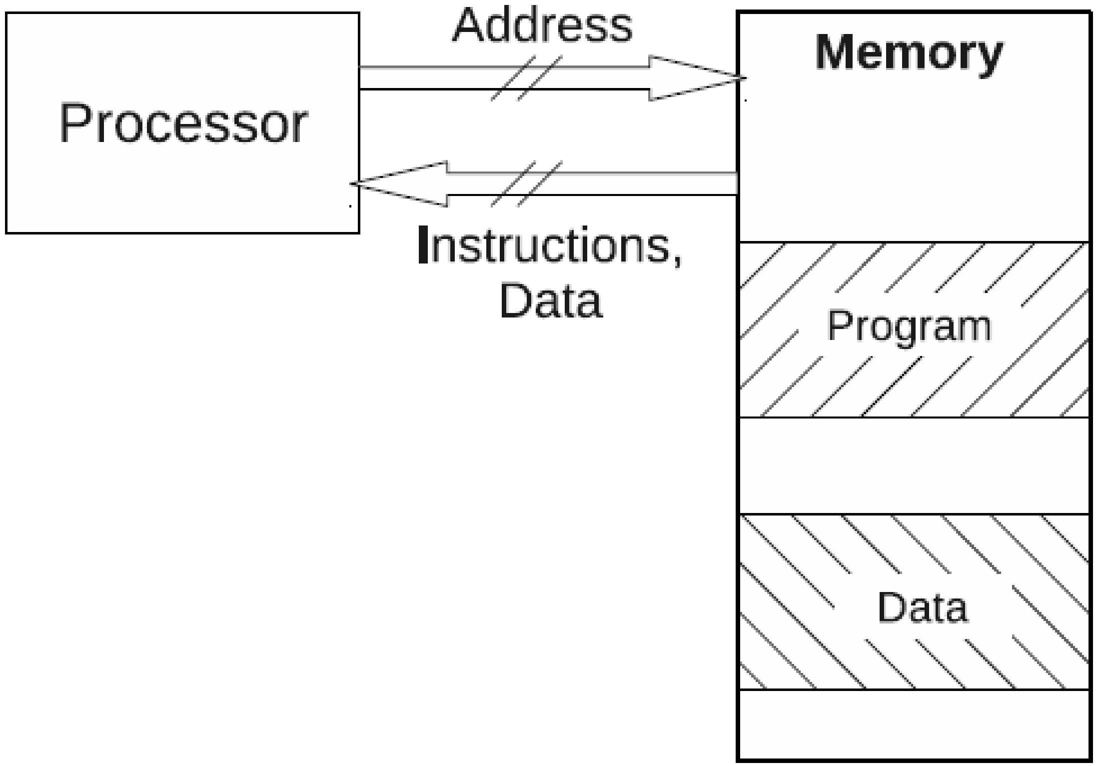
Instructions

- A digital processor can handle only binary strings at the very lowest level
- Thus, all instructions to be carried out by a digital processor needs to be coded or represented as binary strings
- Different basic instructions have to be coded as unambiguous binary strings
- The hardware is capable of looking at a string, decoding it, and carrying out the corresponding instruction
- A sequence of such strings forms a program



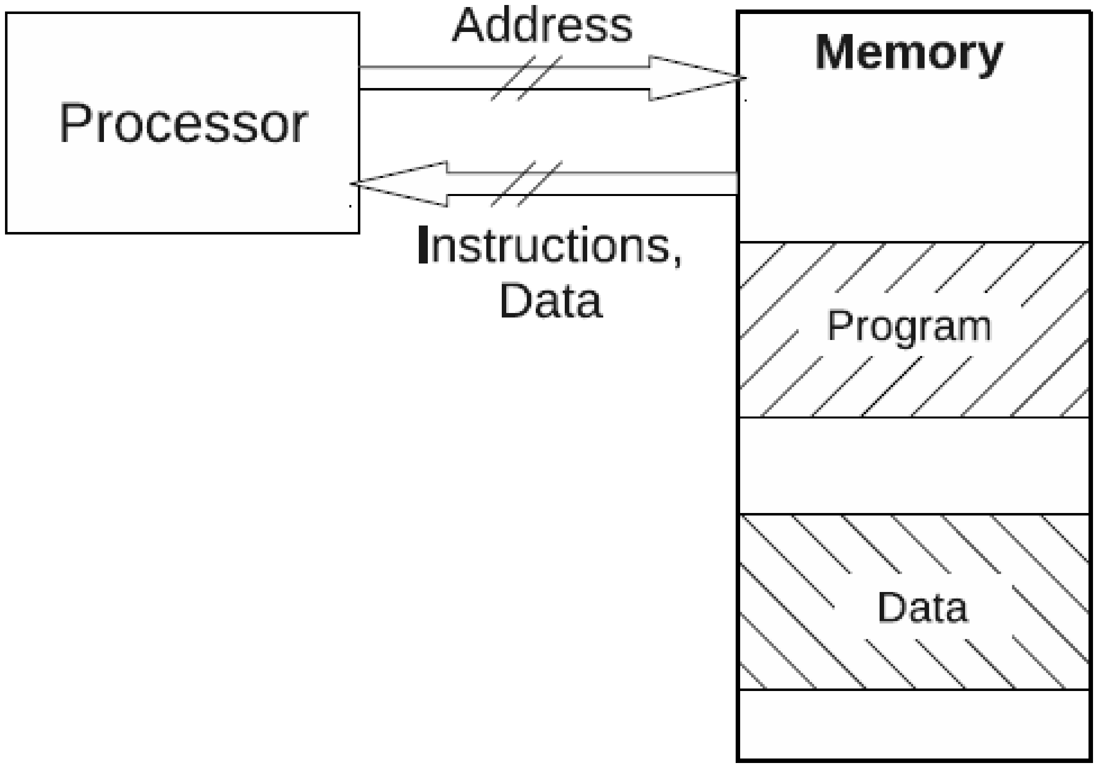
Instructions

- Both the data to be processed and the program can be stored in the same memory (the van Neumann model)
- Instructions that make up the program are stored sequentially in memory
- Each instruction is a binary string that encodes the operations to be performed without ambiguity
- The processor fetches the instructions one by one from the memory and executes it or carries out the corresponding actions



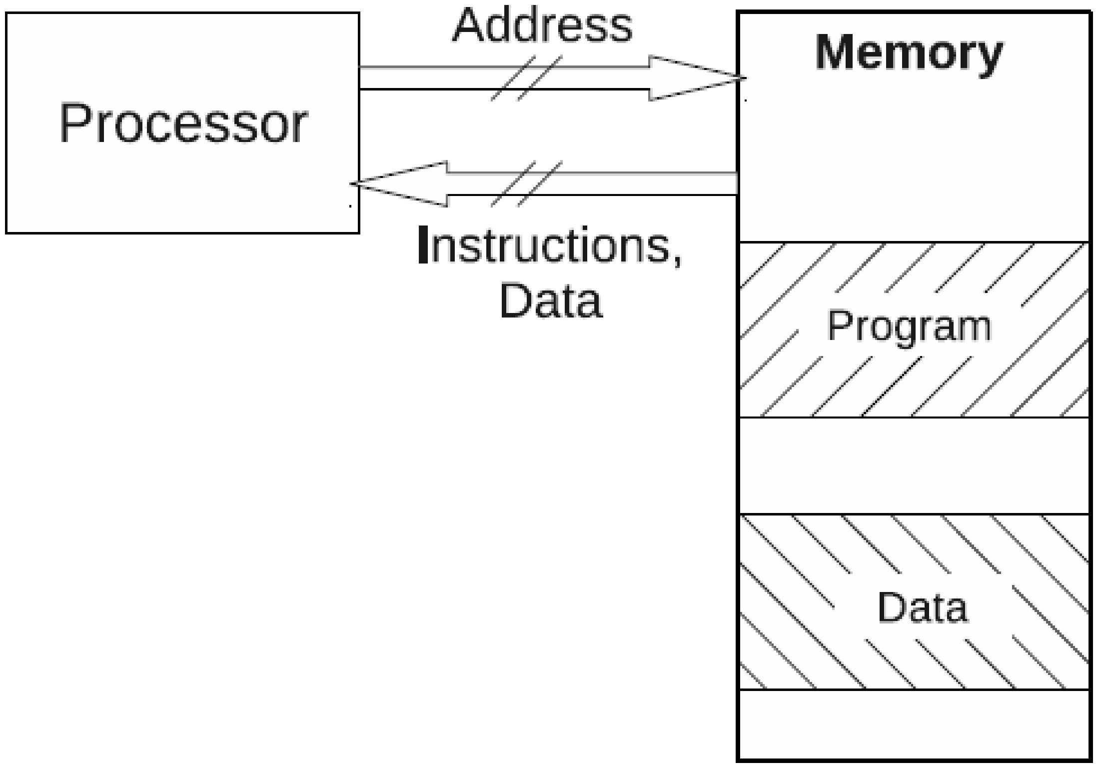
Instructions

- Meaningful work gets done as a side-effect of executing these instructions, as the instructions can read data stored in memory, perform arithmetic, logic, and other operations on the data, and store the results back into the memory
- In fact, the processor is engaged in a perpetual loop of fetch and execute, with the real work done as the side effect of executing the instructions
- Special instructions can also control the input and output from the processor, but we will not consider those for the simple processor



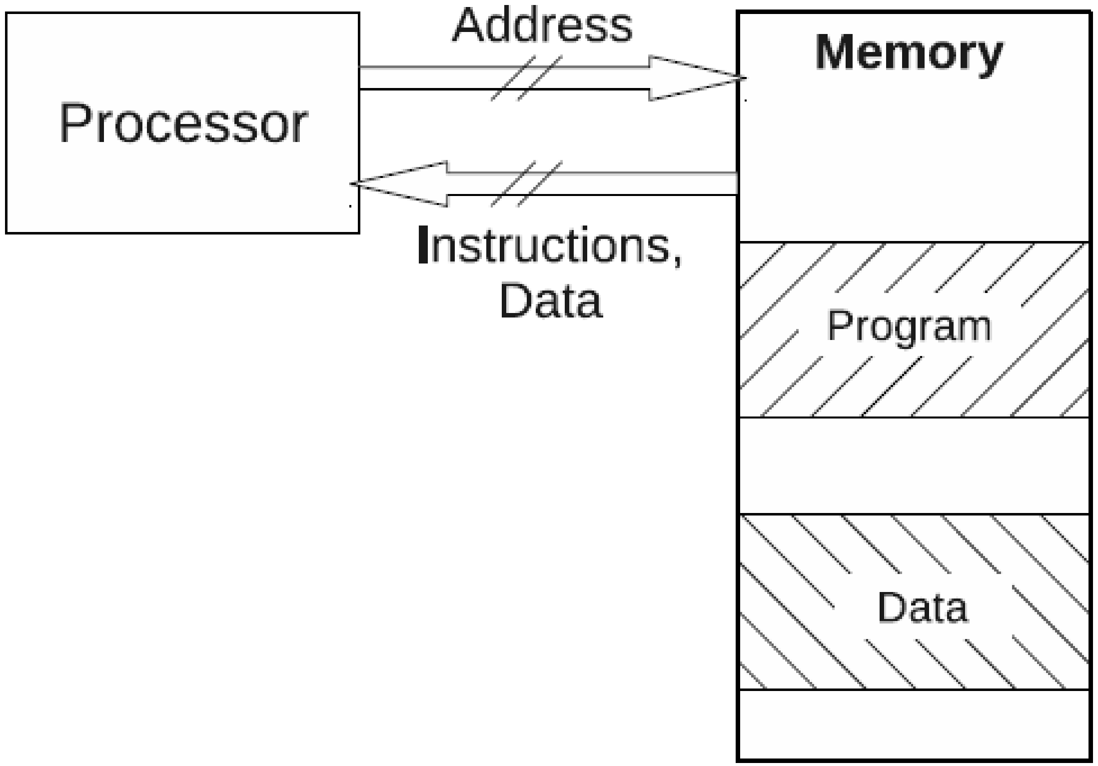
Instructions

- We will follow this scheme for our simple processor
- Coded instructions are fetched from memory and executed by our processor
- We will first look at the execute step of the processor
- We will discuss the mechanism of fetching later
- We will also discuss how the endless fetch-execute loop is realized within the processor



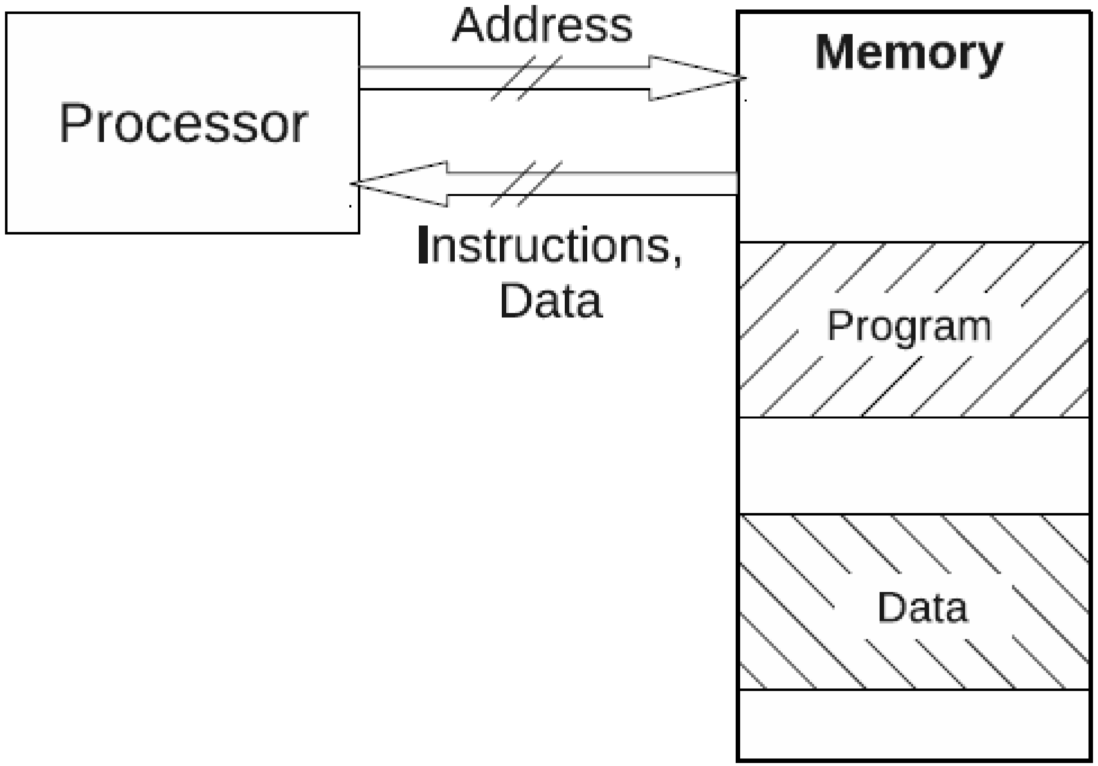
Machine language

- The binary coded instructions are referred to as machine instructions, following the machine language
- This is really no “language” but an encoding scheme that which makes unique decoding of the instructions possible
- Encoded instructions are called machine code or opcode for operation code
- These are understood by the processor naturally
- Importantly, that is the only “language” understood by the processor as it cannot understand high level language (like C++/Python)



Assembly language

- Machine instructions are meant only for the processor; they require tremendous effort to interpret by us
- A mapping of the machine instructions for easier grasp by humans is used widely by processors
- This representation is essentially a one-to-one mapping from machine instructions, using mnemonics or nearly comprehensible short words and symbolic representation of internal resources like the registers
- Such a representation of the basic instructions is called the assembly language



The instruction set – ALU

- Let us assume the word length of our processor is 8 bits
- Thus, all entities we will handle are 8-bits wide, which includes the coded instructions as well as data elements
- Our instruction set will have the arithmetic and logic instructions, namely, add, subtract, and, or, and xor
- We can come up with an arbitrary assembly to machine code mapping as shown in the table

Assembly Instruction	Machine Code	Action
add <R>	10-1F	[AR] $\leftarrow$ [AR] $+$ [<R>]
sub <R>	20-2F	[AR] $\leftarrow$ [AR] $-$ [<R>]
xor <R>	30-3F	[AR] $\leftarrow$ [AR] $\oplus$ [<R>]
and <R>	40-4F	[AR] $\leftarrow$ [AR] $\wedge$ [<R>]
or <R>	50-5F	[AR] $\leftarrow$ [AR] $\vee$ [<R>]
cmp <R>	60-6F	[AR] $-$ [<R>]

The instruction set – ALU

- The <R> in the first column of the table is a parameter that can be replaced by one of R0 to R11, with the corresponding number appearing in the lower significant half of the machine code, given in the second column
- Thus, ADD R1 will be coded as 0x11, XOR R8 as 0x38, and OR R11 as 0x5B
- Any opcode in that range can be unambiguously understood too
- Thus, 0x27 stands for SUB R7, 0x42 for AND R2, etc.

Assembly Instruction	Machine Code	Action
add <R>	10-1F	[AR] $\leftarrow$ [AR] $+$ [<R>]
sub <R>	20-2F	[AR] $\leftarrow$ [AR] $-$ [<R>]
xor <R>	30-3F	[AR] $\leftarrow$ [AR] $\oplus$ [<R>]
and <R>	40-4F	[AR] $\leftarrow$ [AR] $\wedge$ [<R>]
or <R>	50-5F	[AR] $\leftarrow$ [AR] $\vee$ [<R>]
cmp <R>	60-6F	[AR] $-$ [<R>]

The instruction set – ALU

- The last instruction performs a comparison of the register and AR without changing the value of the accumulator
- This may seem pointless as the results are not used
- However, the arithmetic and logic operations have other side-effects based on the results of the operation
- This could include overflow, carry generation, value being negative, etc. These find use in controlling loops in conjunction with conditional branching instructions we will encounter later

Assembly Instruction	Machine Code	Action
add <R>	10-1F	[AR] ← [AR] + [<R>]
sub <R>	20-2F	[AR] ← [AR] − [<R>]
xor <R>	30-3F	[AR] ← [AR] ⊕ [<R>]
and <R>	40-4F	[AR] ← [AR] ∧ [<R>]
or <R>	50-5F	[AR] ← [AR] ∨ [<R>]
cmp <R>	60-6F	[AR] − [<R>]

The instruction set – ALU

- We should have another variation of the above arithmetic and logic instructions in which the actual operand is specified in the instruction itself as a constant
- Such instructions are frequently needed to initialize variables to a constant, such as the loop counter to 0
- Such instructions are set to provide their arguments in the immediate mode
- Here is the problem – all our registers are 8 bits, so these constants should be 8 bits
- We assume the operand is stored in the word that immediately follows the machine code that indicates such an operation

Assembly Instruction	Machine Code	Action
adi xx	01	[AR] $\leftarrow$ [AR] $+$ xx
sbi xx	02	[AR] $\leftarrow$ [AR] $-$ xx
xri xx	03	[AR] $\leftarrow$ [AR] $\oplus$ xx
ani xx	04	[AR] $\leftarrow$ [AR] $\wedge$ xx
ori xx	05	[AR] $\leftarrow$ [AR] $\vee$ xx
cmi xx	06	[AR] $-$ xx

The instruction set – data movement

- We need instructions to move data from and to the accumulator to get our work done
- We have seen the instructions to move contents of AR from or to a register
- We also need instructions to move from AR to and from the memory, which lies outside the processor
- Registers are not sufficient to hold all our data, such as the array of marks obtained by all students
- These are kept in the memory and is brought in and out of the processor as needed
- The *movs* instruction moves a register to the accumulator and the *movd* instruction moves the accumulator to a register
- The register number involved is embedded into the opcode as a parameter as before
- An additional instruction *movi* is provided to move an immediate constant directly to a register

The instruction set – data movement

- The *load* and *stor* instructions involve a register and a memory location
- Memory resides outside of the processor
- To access the memory, one needs to give it an address to indicate which of its words is to be accessed
- The contents of the corresponding memory word will be given to the processor on a read
- The processor has to supply the contents to be written memory for a write
- The number of bits of address determines the maximum capacity of memory that can be used
- We assume that the memory address is represented using one word of 8 bits in our processor
- Thus, the maximum memory capacity is $2^8 = 256$ words in our simple processor

The instruction set – data movement

- The *load* and *stor* instructions use the contents of AR as the address

- The word read from the memory is stored into the register specified in the instruction for the *load* instruction

- The value to be written is available in such a register for the *stor* instruction

Assembly Instruction	Machine Code	Action
movs <R>	70-7F	[AR] $\leftarrow$ [<R>]
movd <R>	80-8F	[<R>] $\leftarrow$ [AR]
movi <R> xx	90-9F	[<R>] $\leftarrow$ xx
stor <R>	A0-AF	[[AR]] $\leftarrow$ [<R>]
load <R>	B0-BF	[<R>] $\leftarrow$ [[AR]]

The fetch-execute cycle

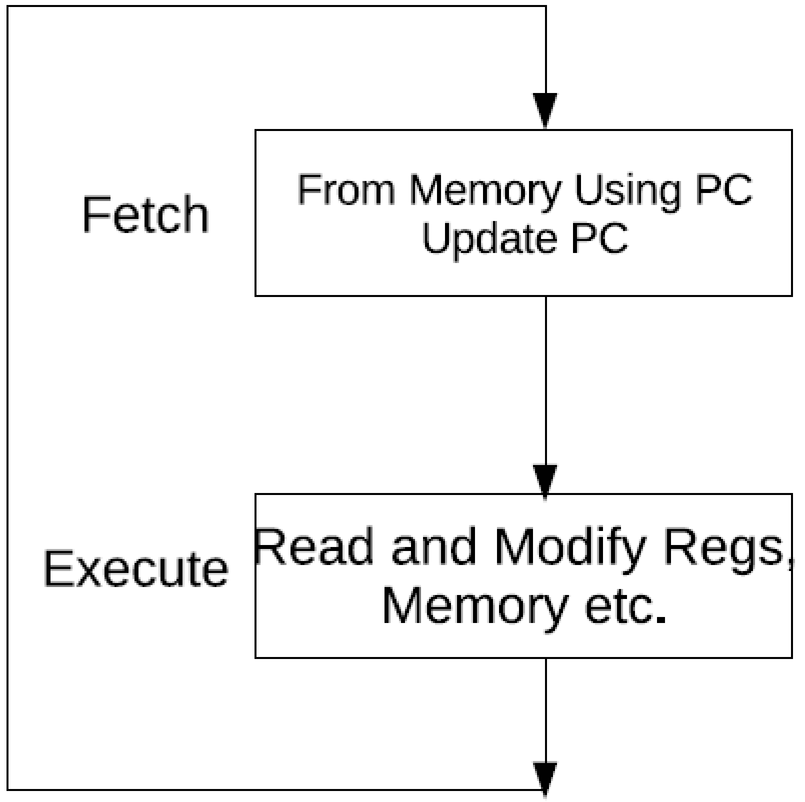
- We will look at the process of instruction fetching and execution
- The processor works autonomously as a continuous fetch-and-execute engine, with no other input than an external clock
- Since instructions as in the machine code are stored in memory, they have to be brought to the processor one by one and executed
- The instruction at address $(i + 1)$ has fetched and executed after instruction i, since the instructions of a program are stored consecutively in the memory
- The processor has to do all these by itself

The fetch-execute cycle

- Processors have a special register inside them that manages the process of instruction fetch by keeping track of the address of the next instruction to be fetched at all times
- This register is called the program counter or the PC
- The processing of an instruction begins with fetching its opcode from the memory word whose address is in the PC
- The contents of the PC are incremented while this happens to hold the address of the next instruction in the sequential order
- The opcode is brought to the processor and appropriate action is performed in the execution phase
- Once this is completed, the next instruction is processed by fetching it from the memory using PC as the address
- This goes on for ever inside the processor until a special STOP instruction is encountered
- Executing this instruction stops all activities of the processor

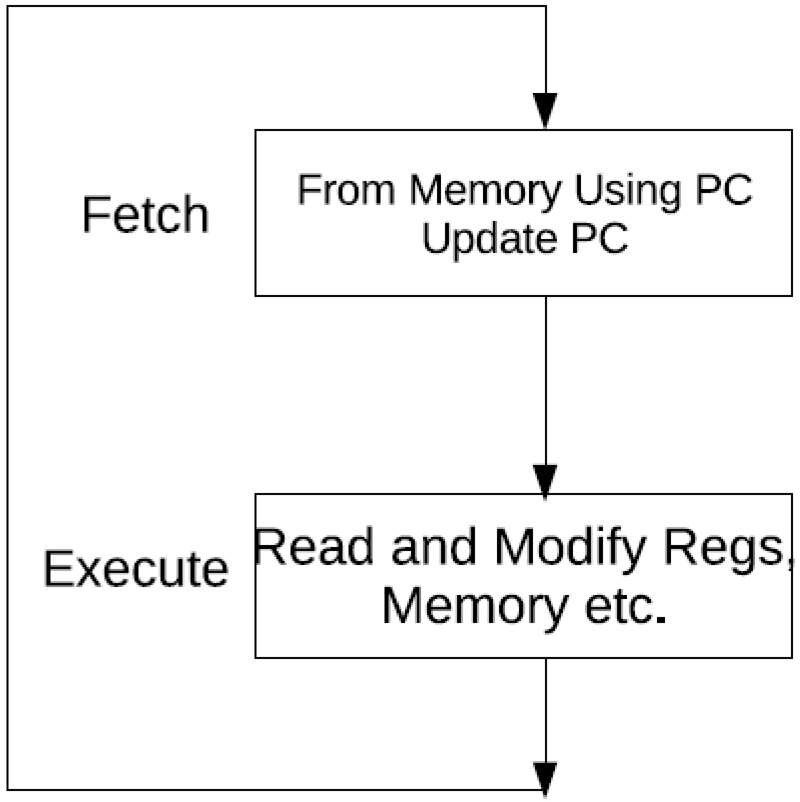
The fetch-execute cycle

- So how do we start the process?
- It is clear that once one instruction is done with, the next one is taken up by incrementing the PC until the STOP instruction is encountered
- Thus, once the execution of a program starts, everything goes on as the program indicates
- So how to start a program?
- A program can be started by loading the address of its first instruction into the PC
- However, how does the very first program start when the computer's power is turned on?



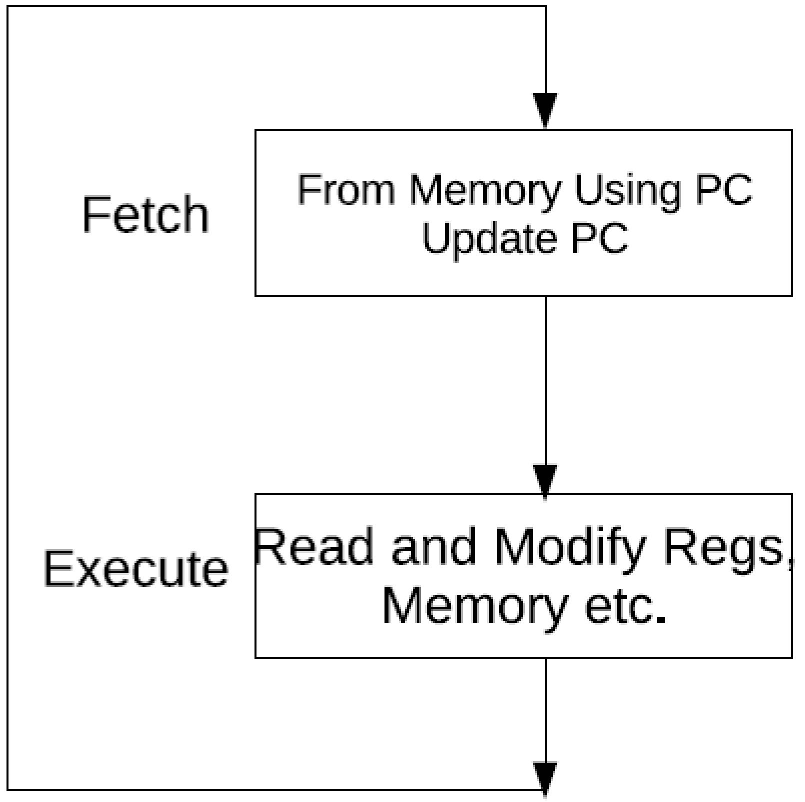
The fetch-execute cycle

- We know Operating System (OS) is the program that controls our computer
- The OS itself is loaded into the processor’s memory from the hard disk on boot up prior to taking over the system
- Which program loads the operating system? How does that program get the control at the very beginning?
- Modern PCs have a program called the BIOS (Basic Input Output System), which is the very first one to get control of the processor
- How does the BIOS get control?



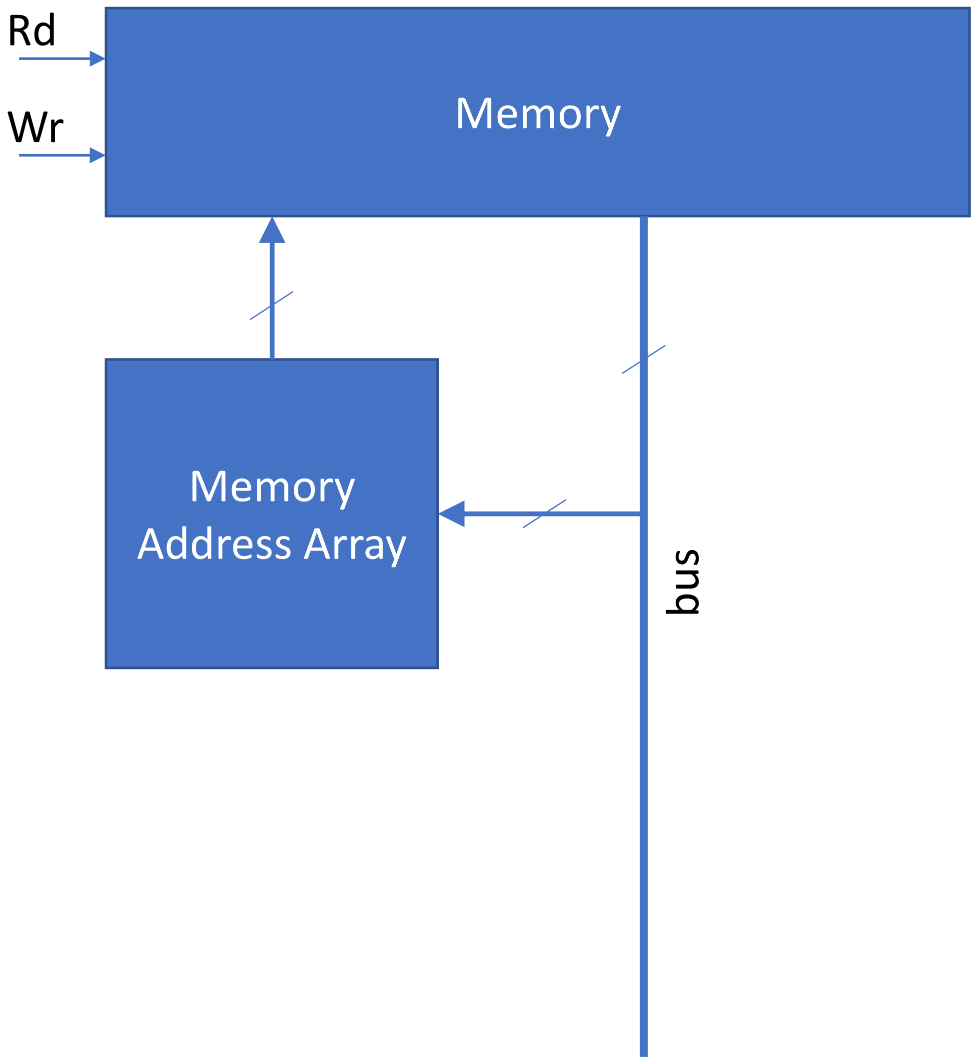
The fetch-execute cycle

- The processor hardware has a special feature to load a value of 0 to the PC when power is turned on or when the reset button of the computer is pressed (literally resets the PC)
- Thus, the very first program that gets control is the one that is saved at memory address 0
- Computer manufacturers have placed a special read only memory at address 0 that has the BIOS program, which knows how to load the operating system from the boot record and proceed accordingly



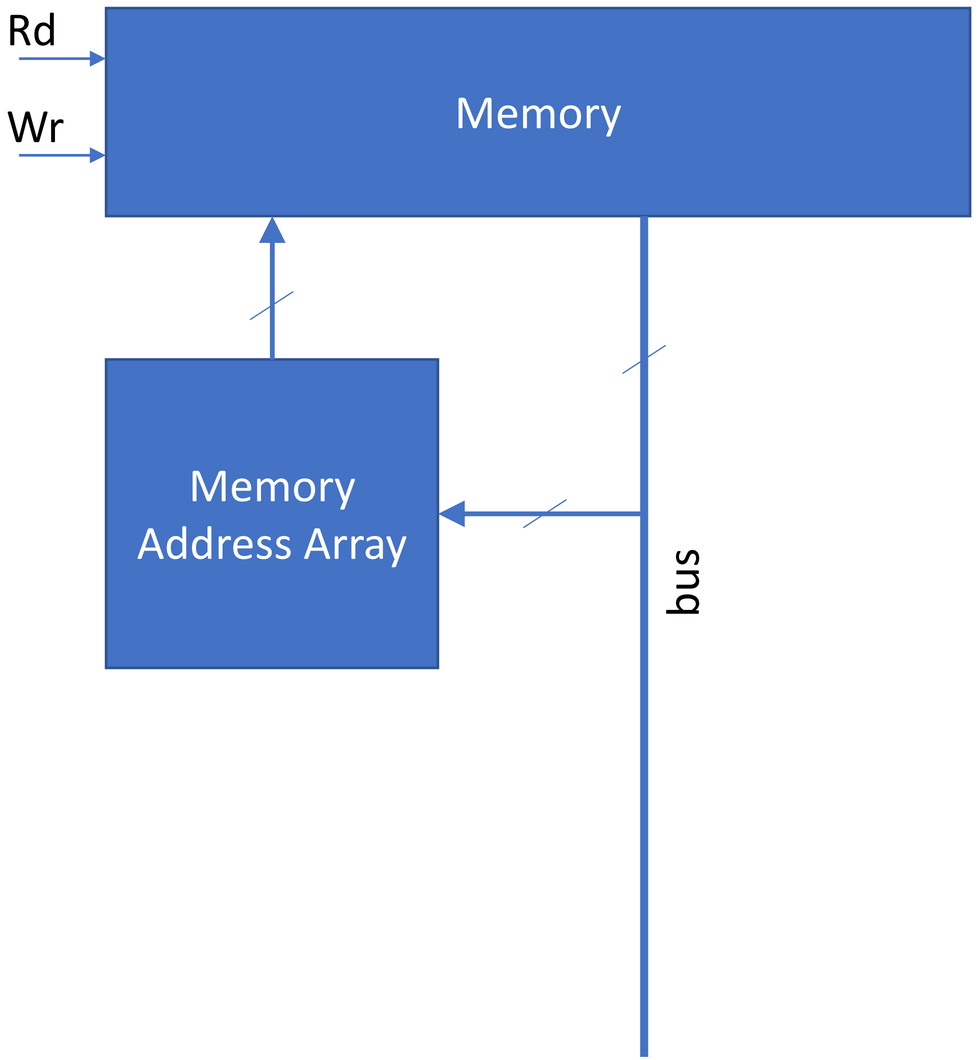
Memory access

- How does the simple processor access the memory?
- We will use a memory interface that supplies the address to the memory along with the signals to indicate if a read or a write is desired
- Data should be presented separately for writes; data supplied by the memory should be used inside the processor for reads
- We assume an external memory interface consisting of address lines, data lines, and two control lines
- The data lines are connected directly to the data lines of the bus, as if the memory is a large register array, but outside of the processor



Memory access

- The address has to be supplied separately, prior to the read or write operation
- We assign a memory address register (MAR) to hold the address
- The MAR is connected to the bus like other registers and can be written to from the bus
- There is usually no need to enable the MAR to the internal bus
- It can be assumed to be enabled always to the external memory interface
- Two control lines RD and WR are sent to the memory to indicate memory read and write respectively



Enhanced enhanced single bus architecture

- With this information, the enhanced single bus architecture is modified to include additional components
- The memory address register to store the next memory address to be accessed
- The program counter to store the current address of the instruction being performed