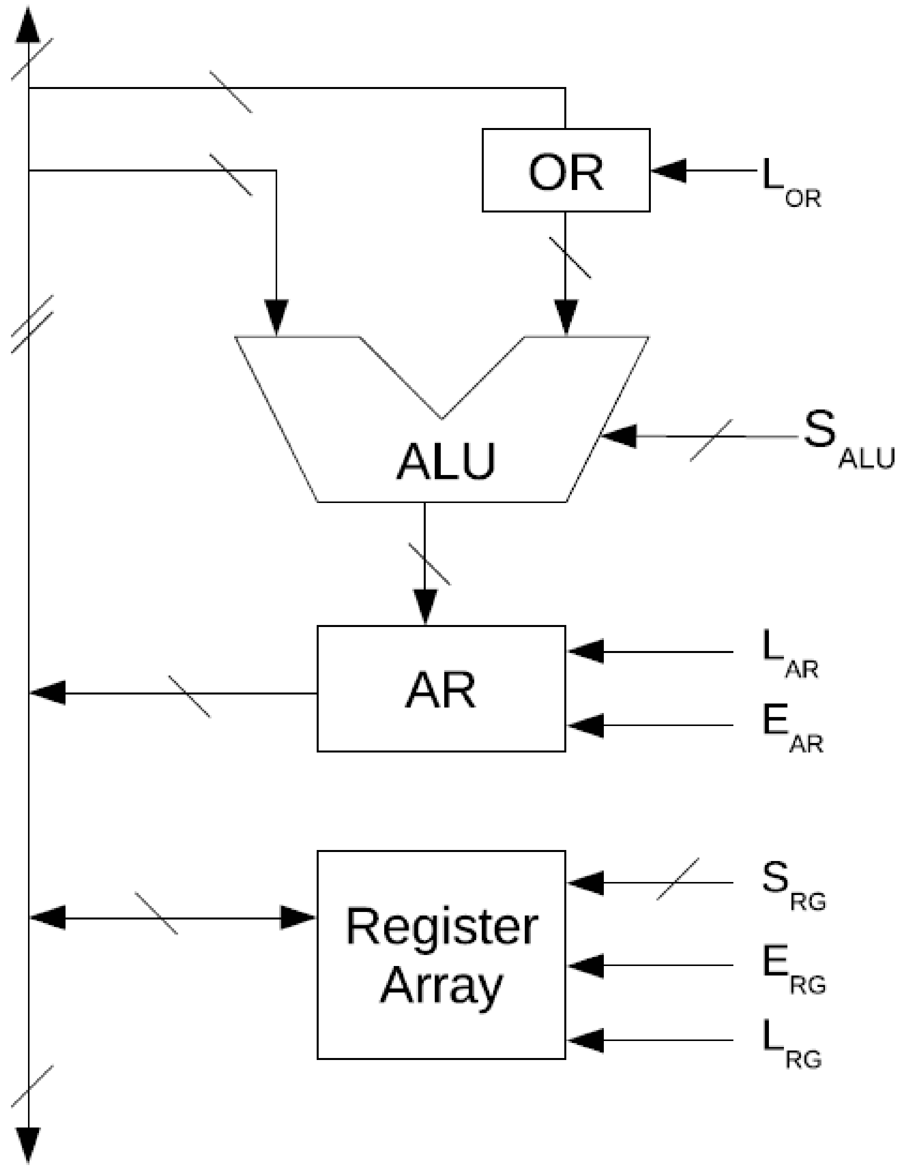
Lecture 29 – Processor design: The Age of ALU

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

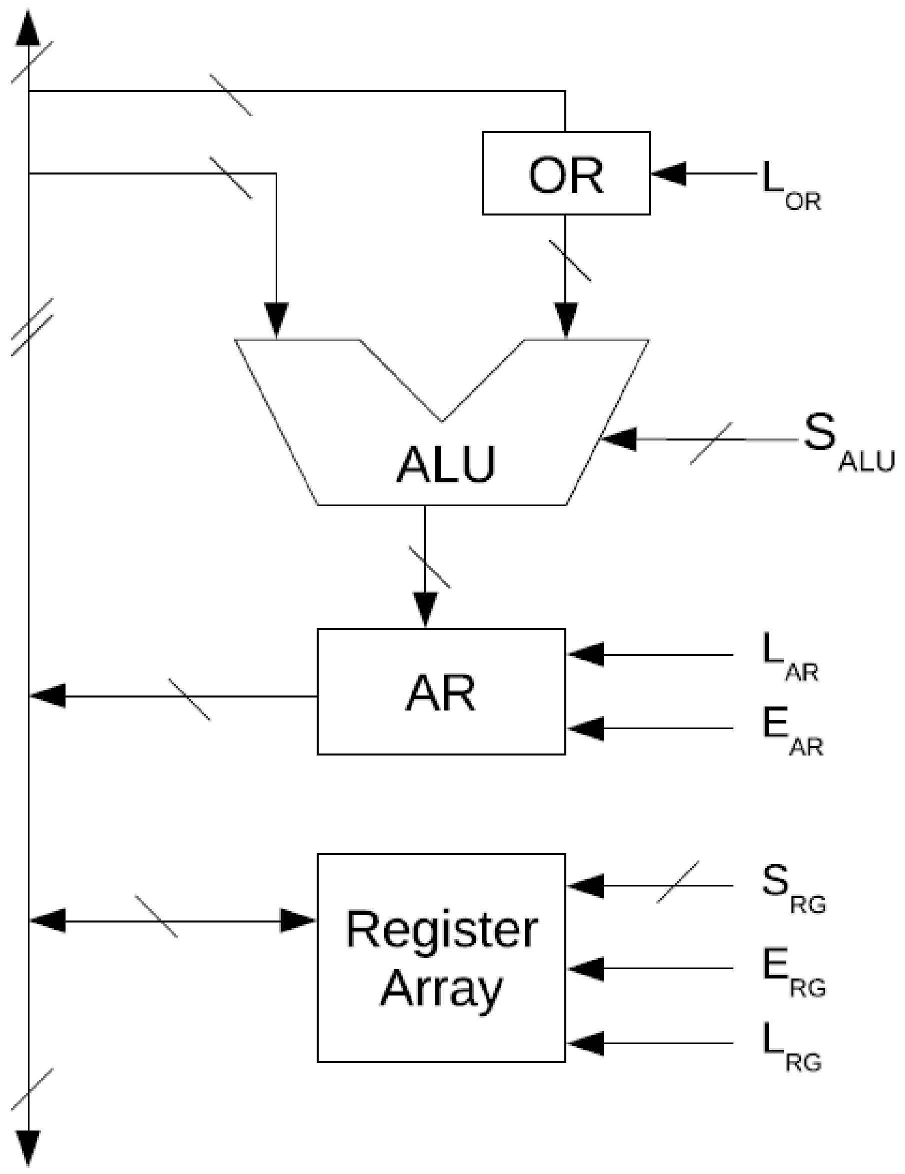
Enhanced singlebus architecture

- We can consolidate the registers into a register array or a register file
- These are numbered R_0 through R_{11}
- The register file has a single enable input E_{RG} and a single load L_{RG} and 4 select lines S_{RG}
- The select lines identify which register of the file is being operated on, with enable or load controlling the action performed on it
- The new design of the register array needs only 6 control signals – 4 for select and 2 for enable/load – as opposed to 22 that would be needed were each register to have its individual enable and load signals
- Some generality is, however, lost as only one register can be selected for writing



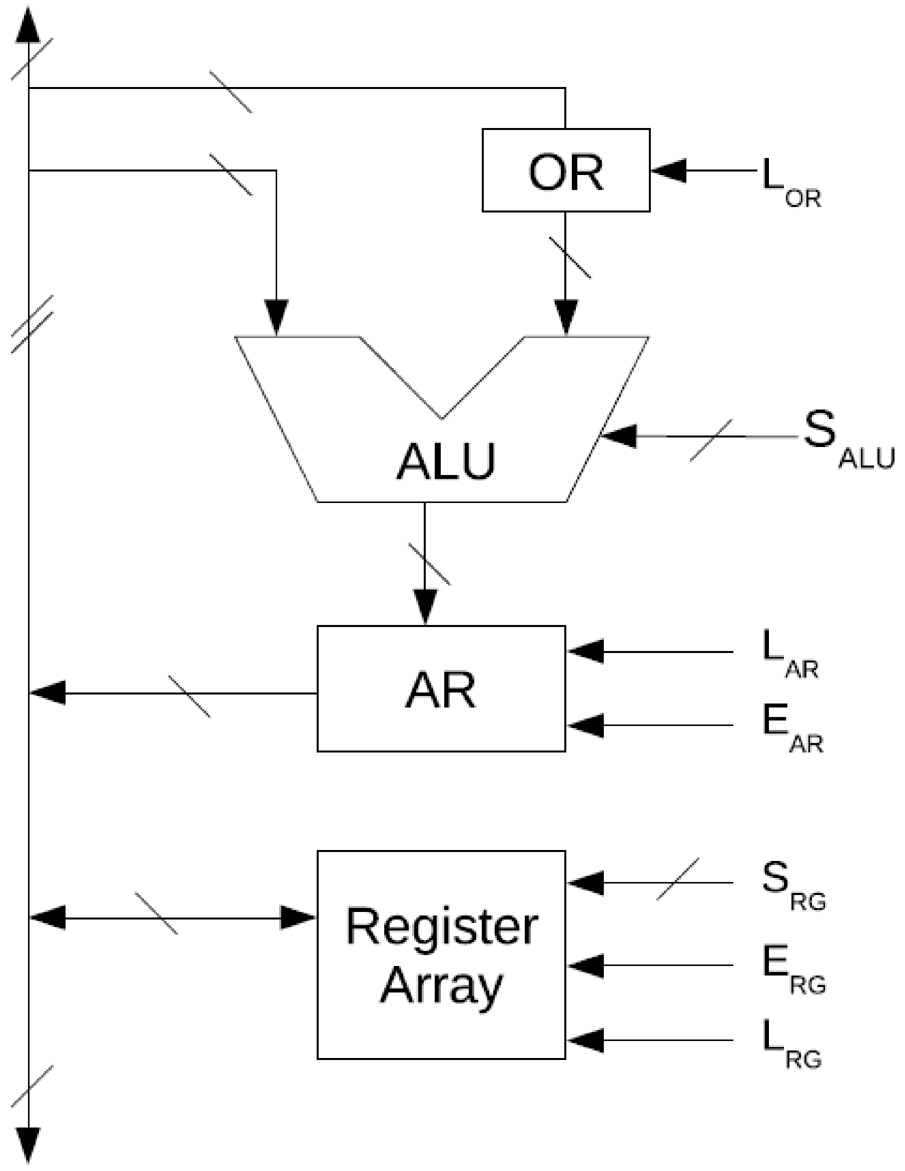
Enhanced singlebus architecture

- We also decide to make the registers also simpler devices as they need to do only parallel load and read
- Reset, increment, shift, etc., are not possible on them
- These registers are temporary store of information
- We also have an enhanced ALU in place, with 3 select lines and supporting 8 operations: zero, add, subtract, logical AND, OR, XOR, and pass left. The ALU takes the left operand from the bus and the right one from OR



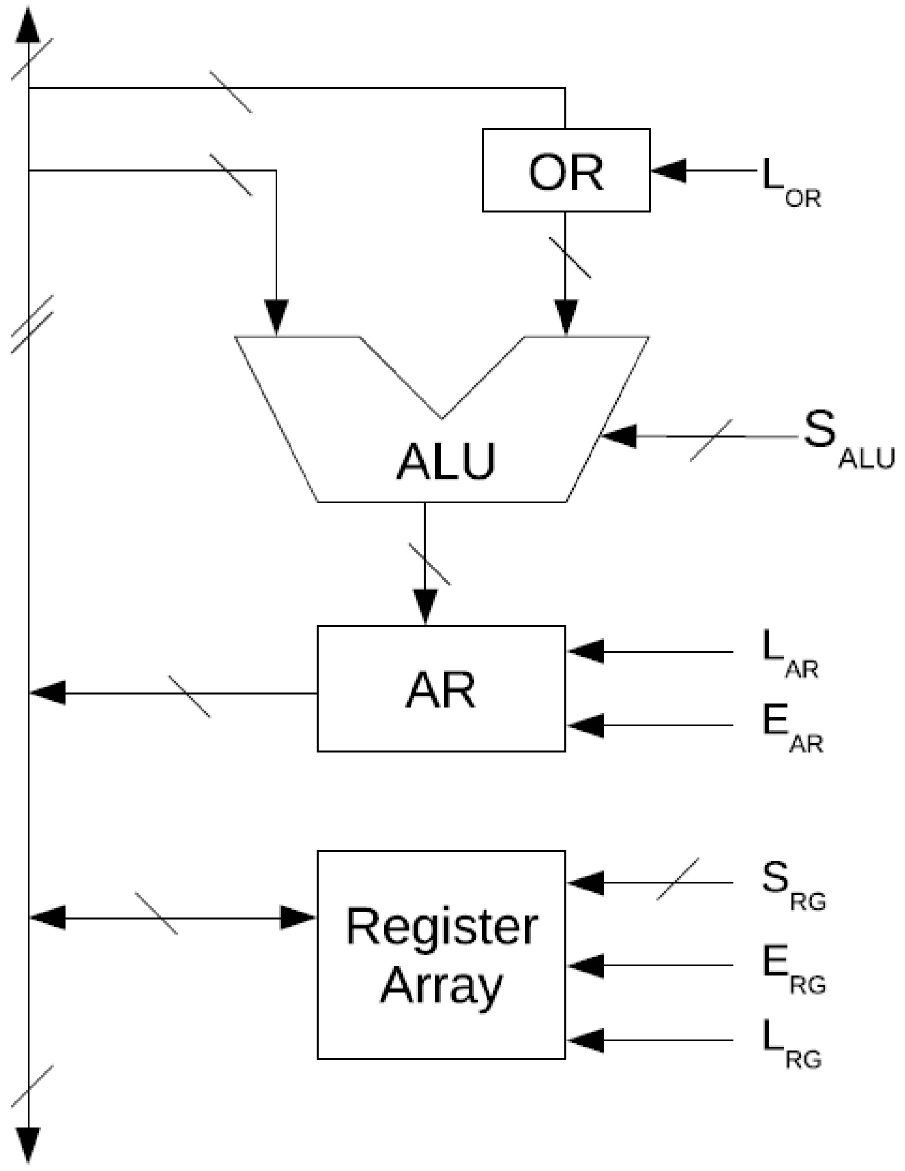
Enhanced singlebus architecture

- ADD R1: (AR = AR+R1)
    - Ck 0: E_{RG}, L_{OR}, S_{RG} ← 1
    - Ck 1: E_{AR}, L_{AR}, SALU ← ADD
- SUB R7: (AR = AR – R7)
    - Ck 0: E_{RG}, L_{OR}, S_{RG} ← 7
    - Ck 1: E_{AR}, L_{AR}, SALU ← SUB
- XOR R11: (AR = AR XOR R11)
    - Ck 0: E_{RG}, L_{OR}, S_{RG} ← 11
    - Ck 1: E_{AR}, L_{AR}, SALU ← XOR
- OR R0: (AR = AR OR R0)
    - Ck 0: E_{RG}, L_{OR}, SRG ← 0
    - Ck 1: E_{AR}, L_{AR}, SALU ← OR



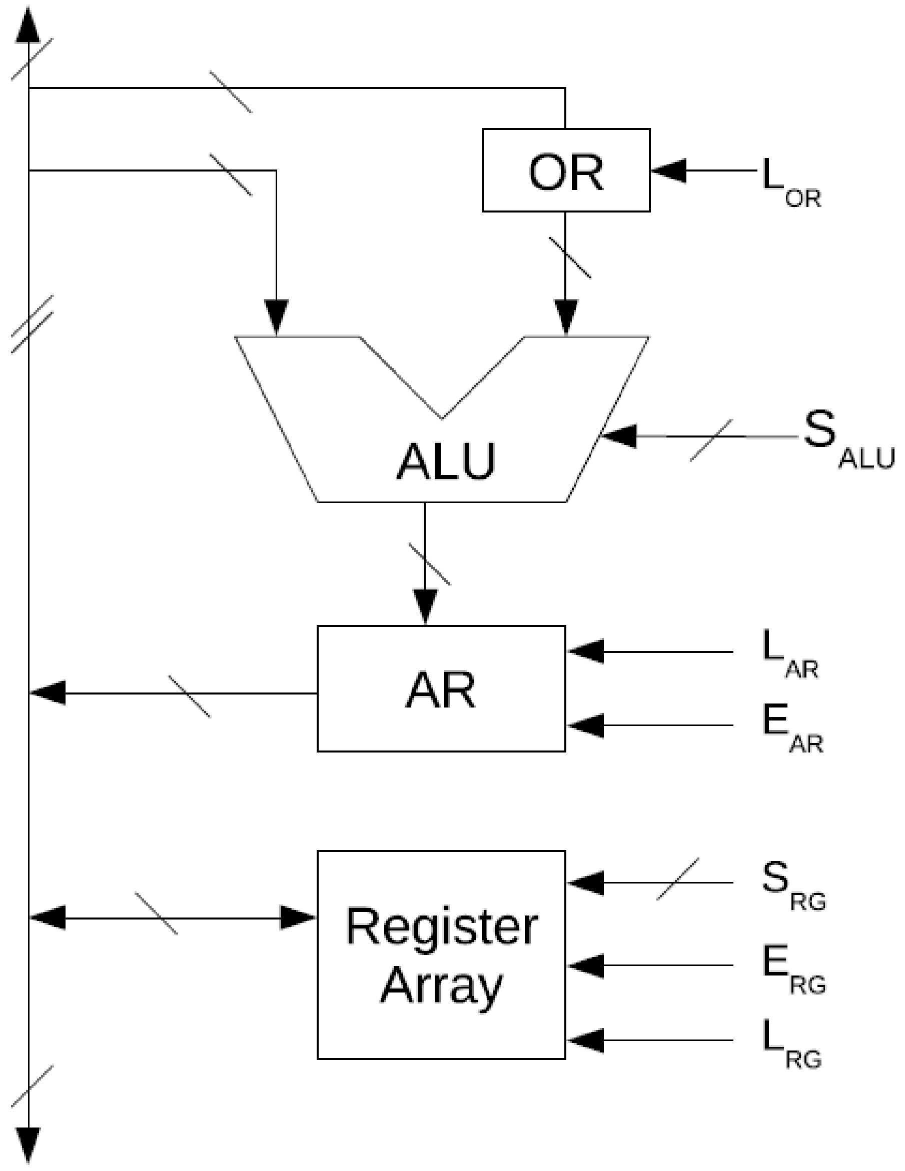
Enhanced singlebus architecture

- We also need a way to load values from register to AR and to save results from AR to a different register
- We call these MOVS and MOVD respectively
- MOVS R4 and MOVD R9 can be implemented as follows:
- MOVS R4:
  - Ck0: E_{RG}, L_{AR}, $S_{RG} \leftarrow 4$, SALU $\leftarrow$ PASS0
- MOVD R9:
  - Ck0: E_{AR}, L_{RG}, $S_{RG} \leftarrow 9$



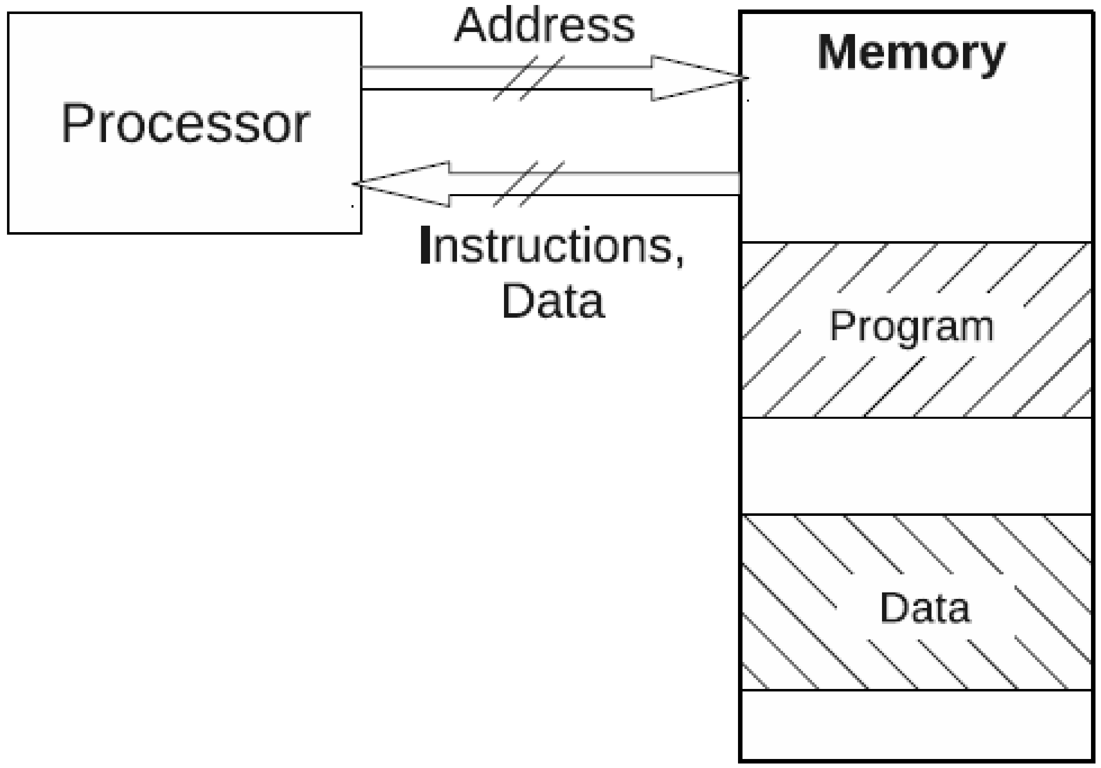
Instructions

- 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



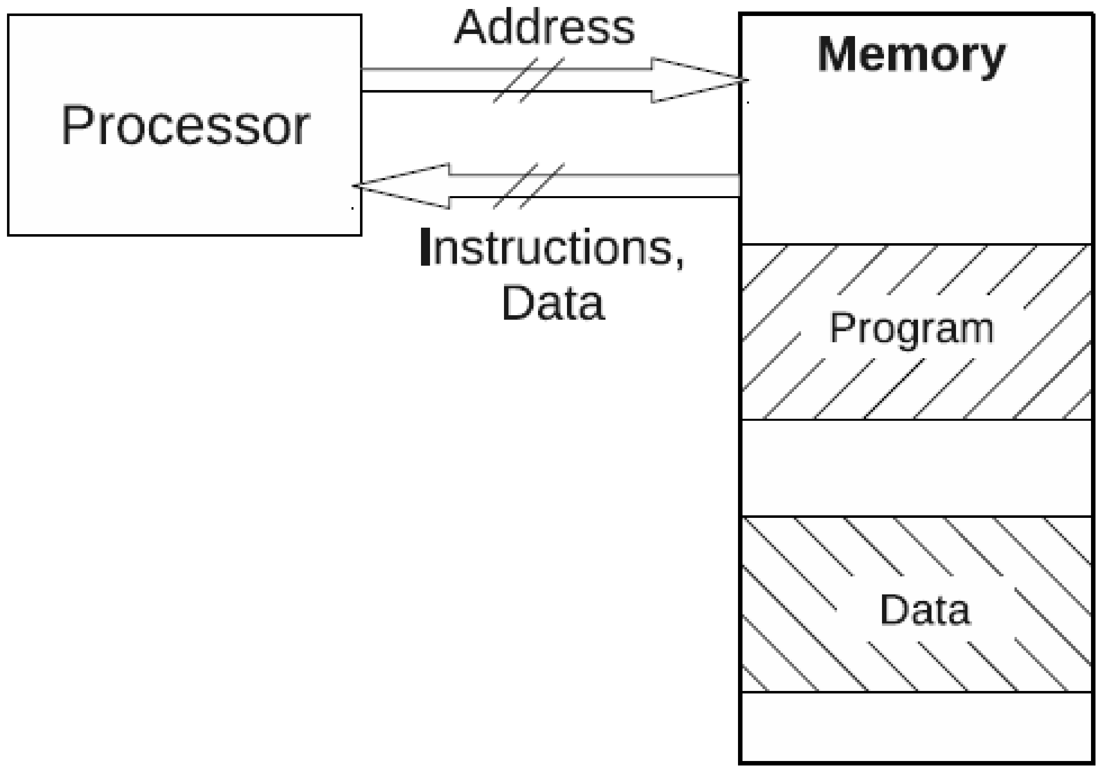
Instructions

- Both the data to be processed and the program are stored in the same memory
- 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



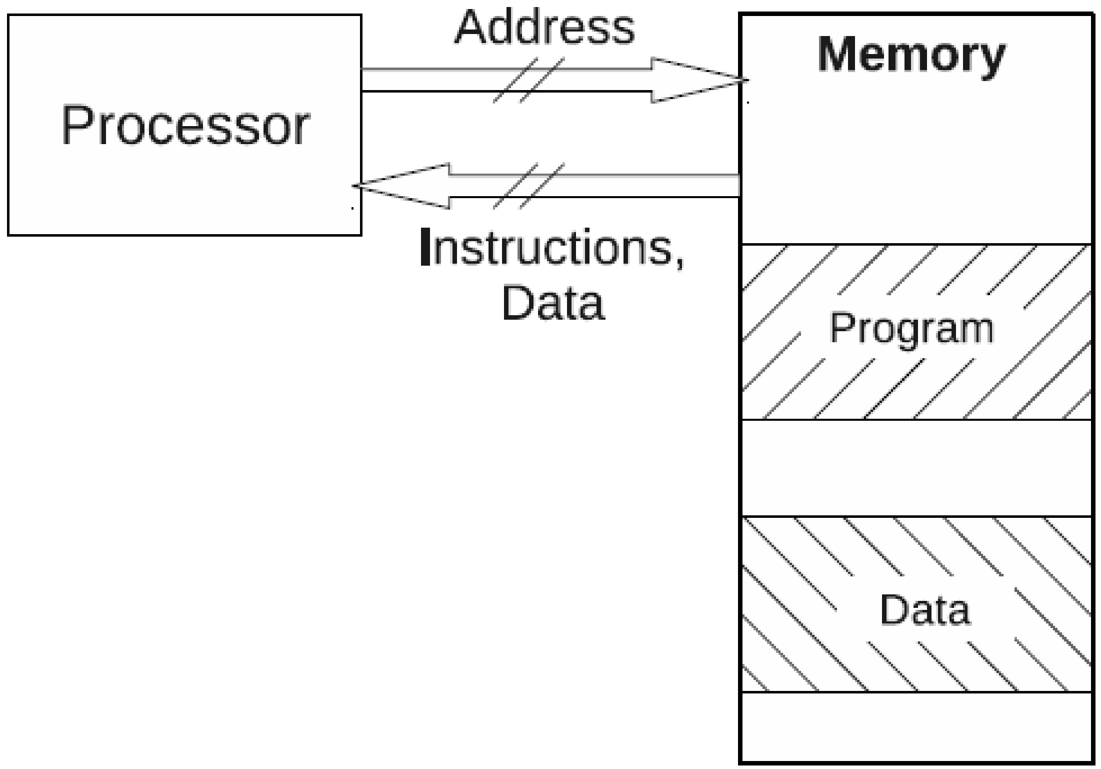
Instructions

- 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



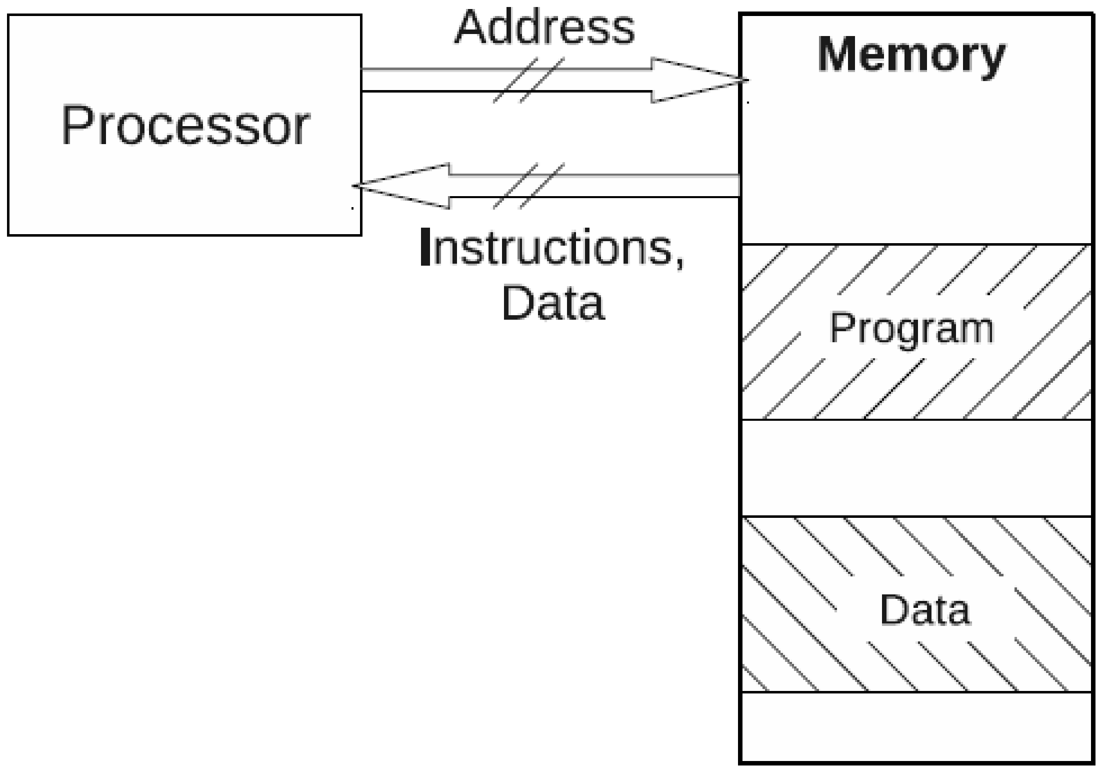
Machine language

- 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)



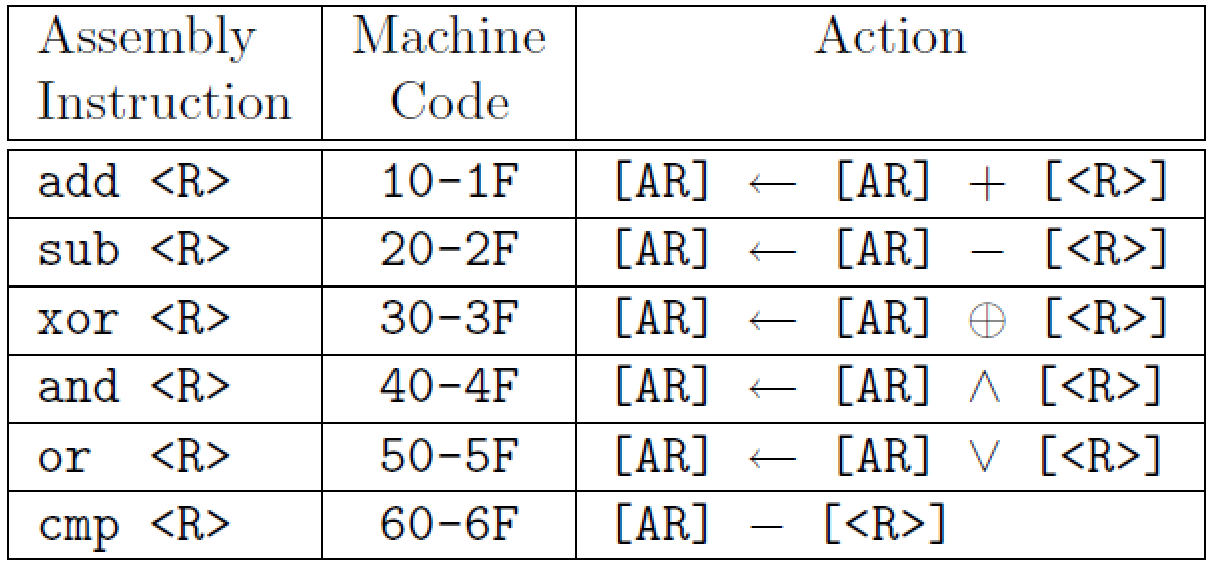
Assembly language

- 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 (flags) 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] $\leftarrow$ [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

- 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