# Lecture 28 – Processor design 3

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

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)
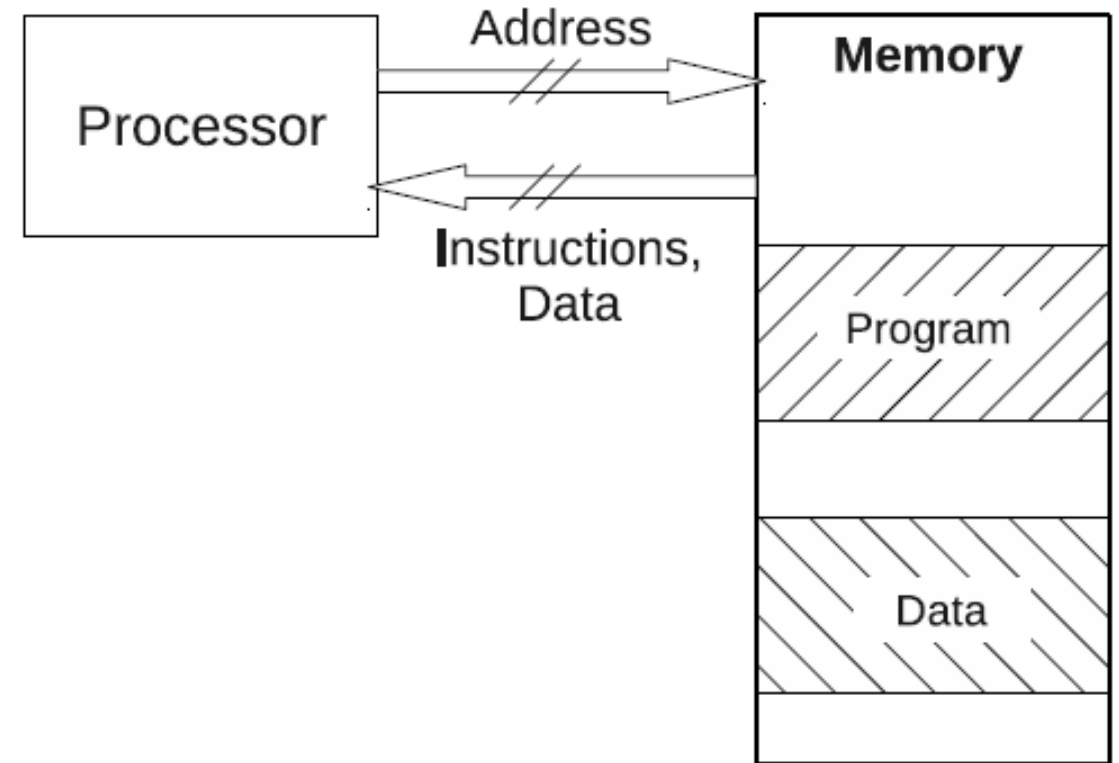
Assembly language

- 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
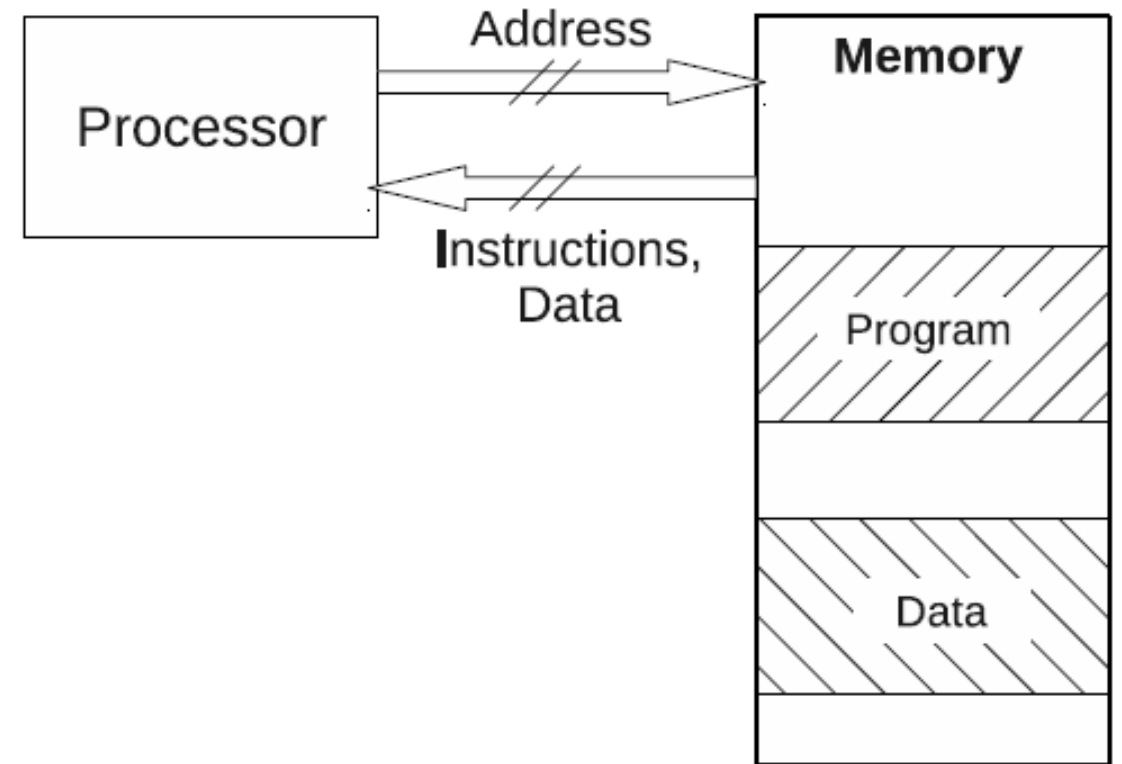
The instruction set – ALU

- 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] ← [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

- 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] ← [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

- 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] ← [<R>]
movd <R>	80–8F	[<R>] ← [AR]
movi <R> xx	90–9F	[<R>] ← xx
stor <R>	A0–AF	[[AR]] ← [<R>]
load <R>	B0–BF	[<R>] ← [[AR]]