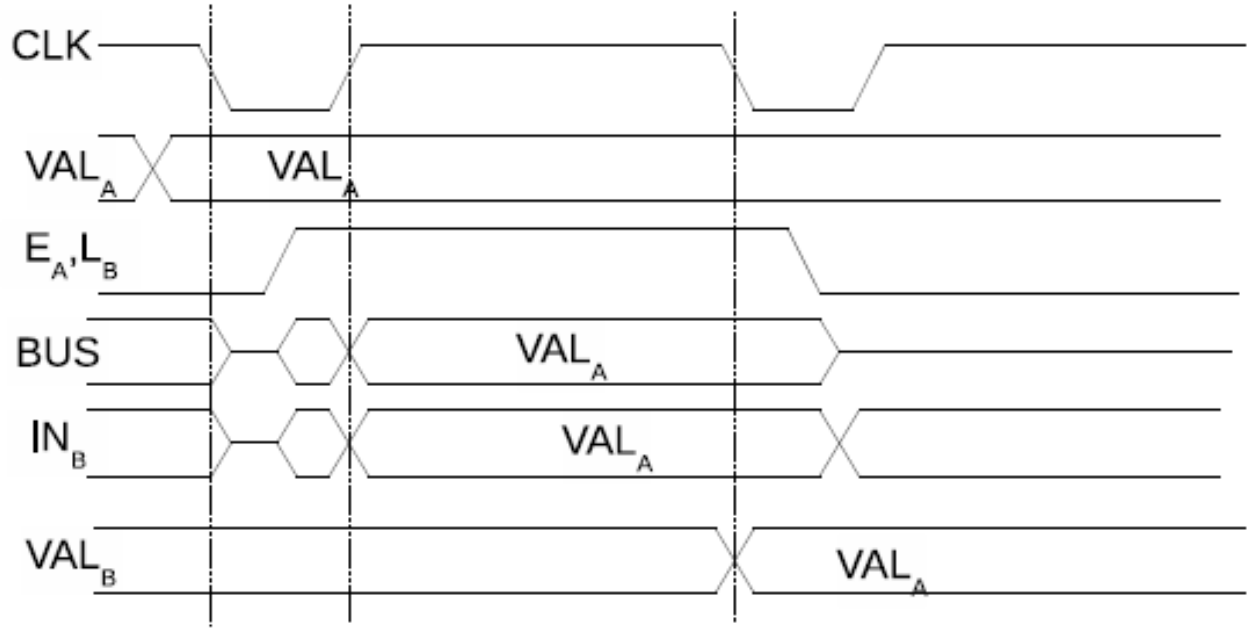
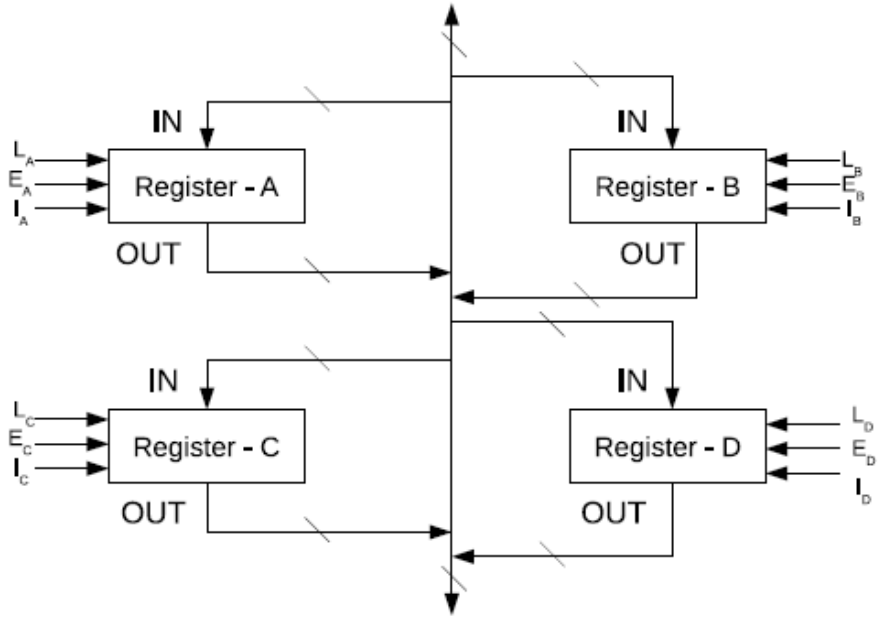
Lecture 27 – Processor design 2

Creating a common bus

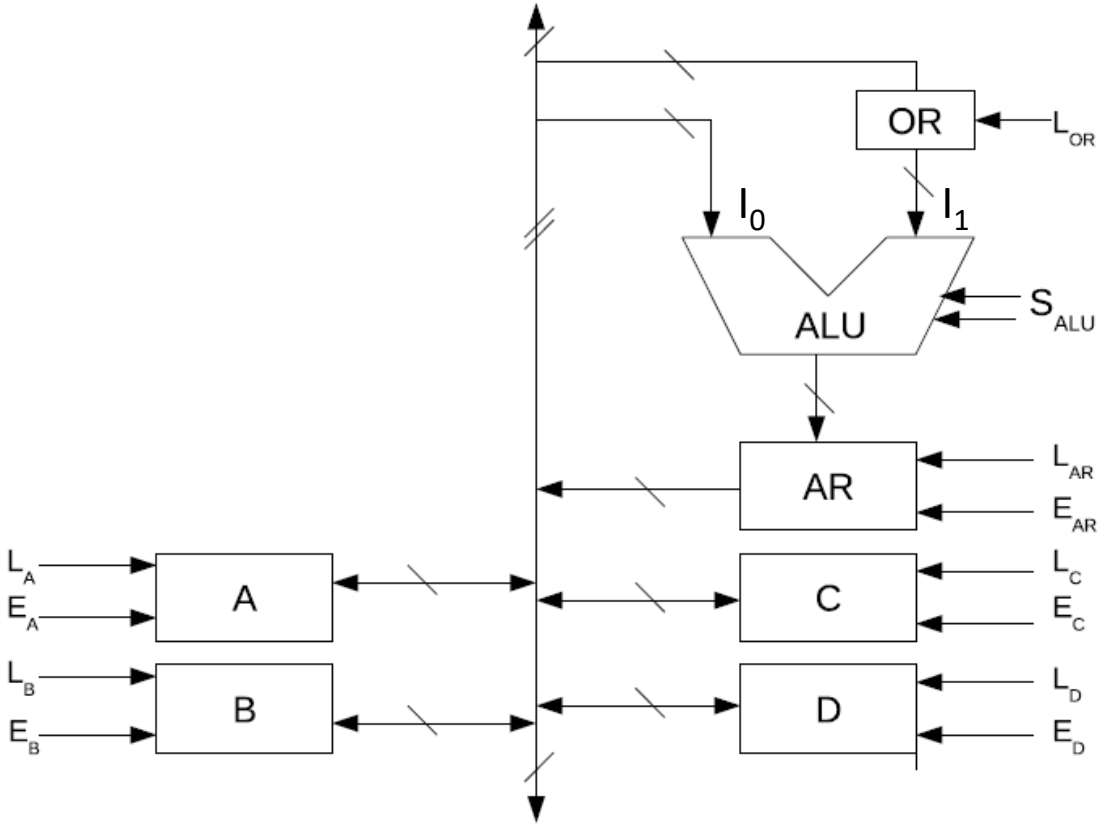
- If we enable E_A and L_B and in one clock cycle, the contents of register A is copied to register B
- What if we activate E_A, L_B, I_A





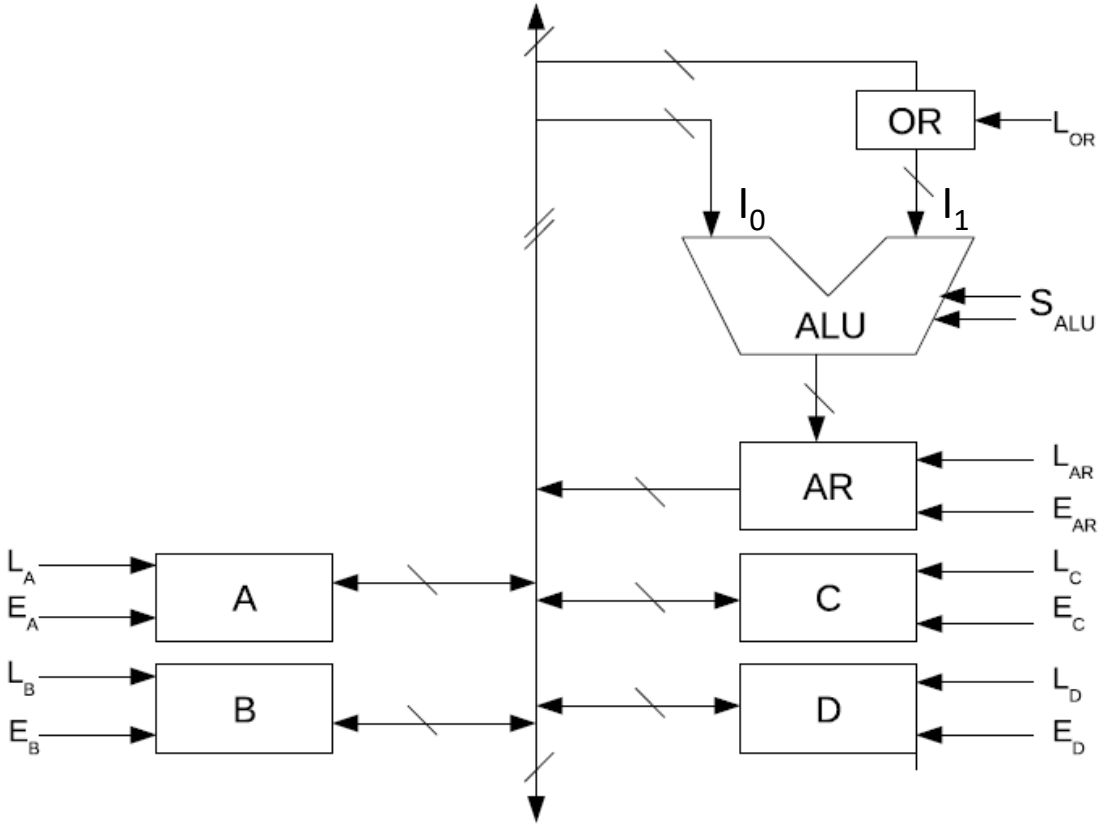
ALU on a bus

- Consider the configuration which has a few registers and an ALU
- Assume all the registers are made using the multipurpose registers
- The ALU is a combinational circuit with 2 inputs and two lines to select the function to be performed
- The two select lines (S_{ALU}) can be in one of 4 combinations:
  - We refer to them symbolically as ADD, SUB, AND, PASS0



ALU on a bus

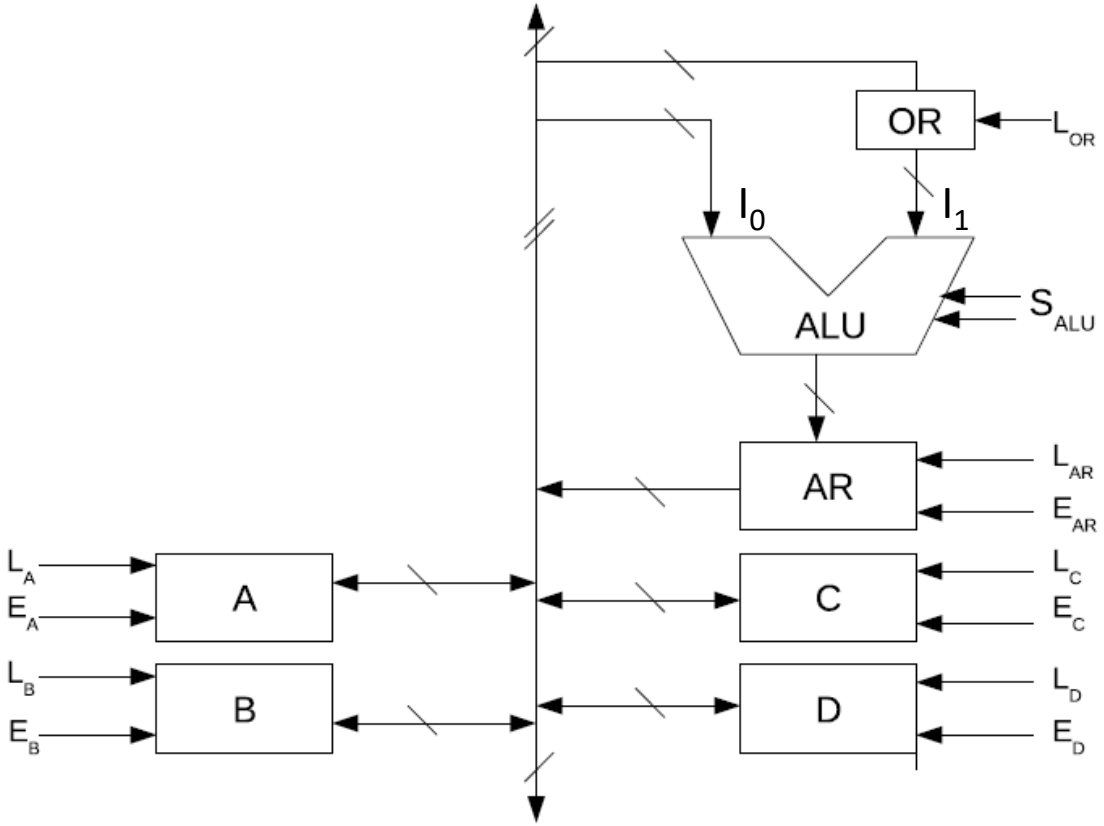
- The ALU has its left input connected to the bus and the right input to a register called OR or operand register, whose inputs are connected to the bus
- The output of the ALU is connected to the input of a register called AR or accumulator register, whose output is connected to the bus
- Every register has all the control signals of our generic multipurpose register



ALU on a bus

- What will happen if we activate the following controls: $E_{OR}, E_A, L_{AR}, S_{ALU\_ADD}$
- Two enables are simultaneously active
- Does it cause conflict at the bus?
- No, since the output of OR is not connected to the bus, there will be no conflict
- The last term above indicates that the select lines of the ALU are set to the bits corresponding to "ADD"
- The ALU performs addition of its inputs, which are the contents of OR register and register A
- The sum is written to AR register

AR ← A + OR



ALU on a bus

- Consider this multiclock instruction:

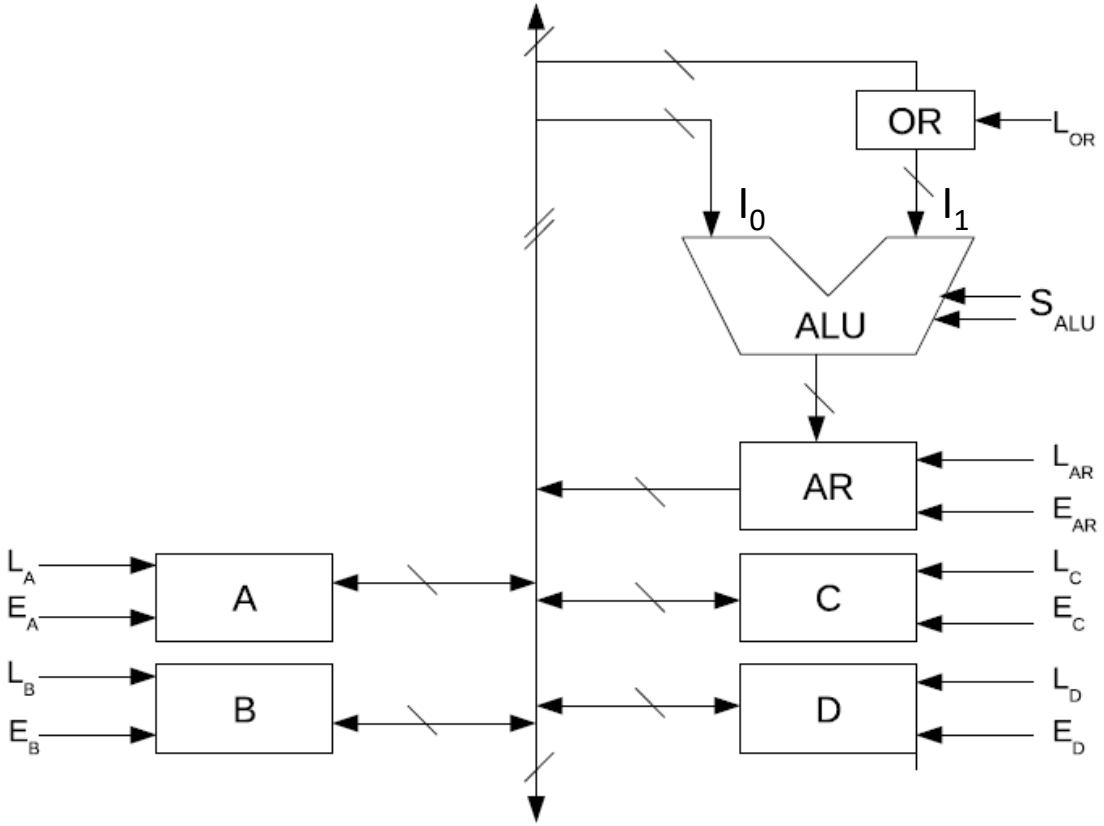
  Ck 10: E_A, L_{OR}

  Ck 11: E_B, E_{OR}, L_{AR}, $SALU_{SUB}$

  Ck 12: E_{AR}, L_C

- What does the 3 clock cycle combination achieve?
- The contents of A are copied to OR in cycle-10
- The contents of OR are subtracted from the contents of B and the result is loaded to AR in cycle-11
- The contents of AR are copied to register C in cycle-12

$C \leftarrow B - A$



ALU on a bus

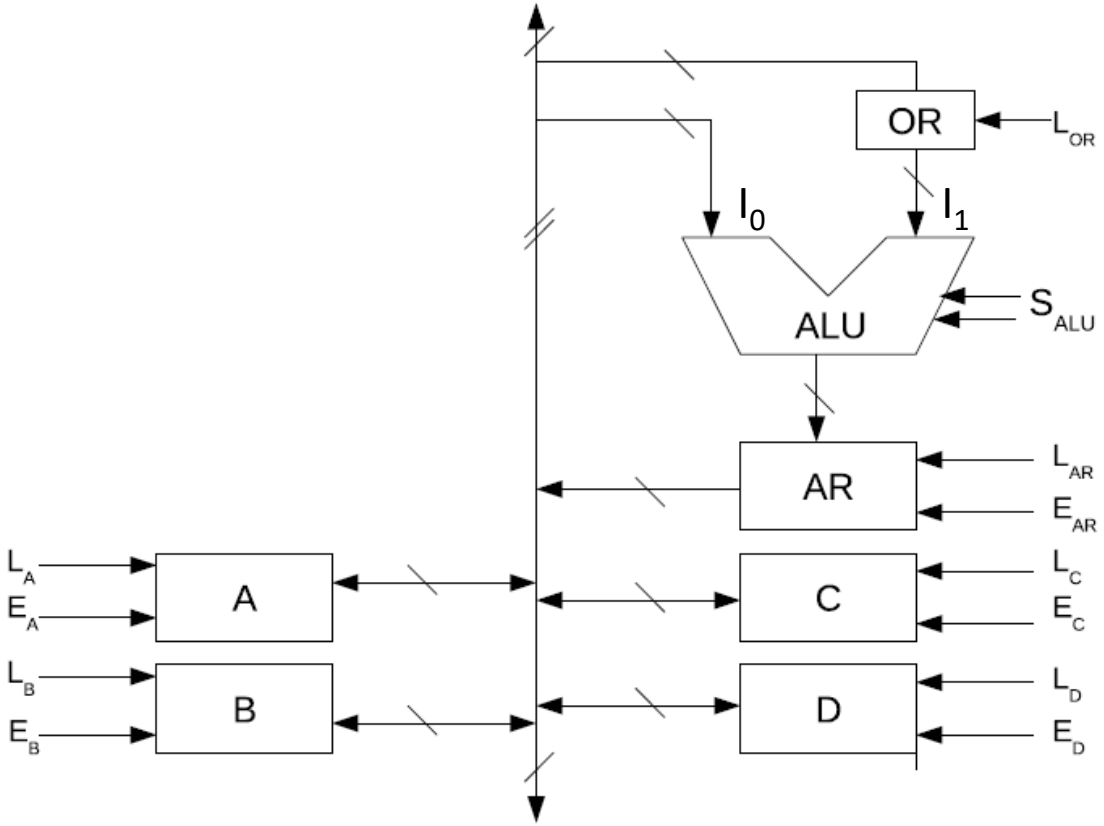
- It is easy to see how addition, subtraction and logical AND with any combination of 2 registers as input and any register as output can be implemented using a very similar 3-cycle sequence of combinations of signals
- Let us see if we can write control signals for this instruction:

$D \leftarrow C$ AND D

Ck 1: E_C, L_{OR}

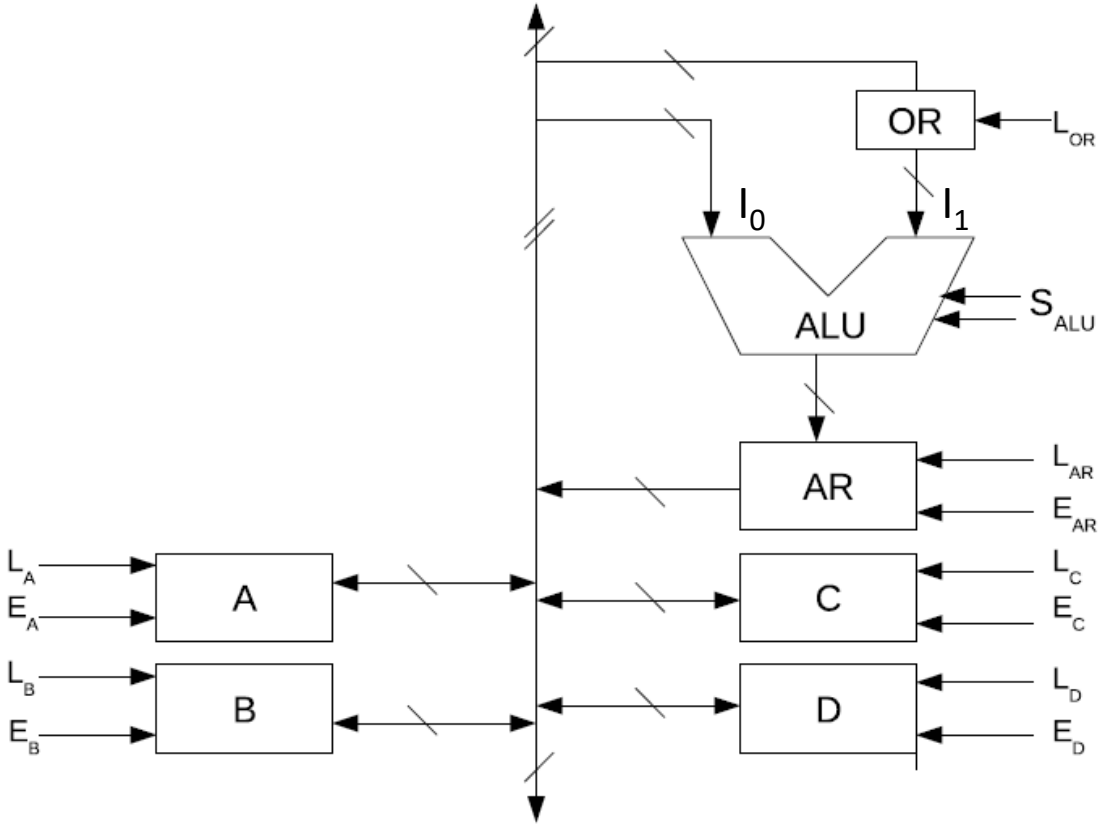
Ck 2: E_D, E_{OR}, L_{AR}, S_{ALU_AND}

Ck 3: E_{AR}, L_D



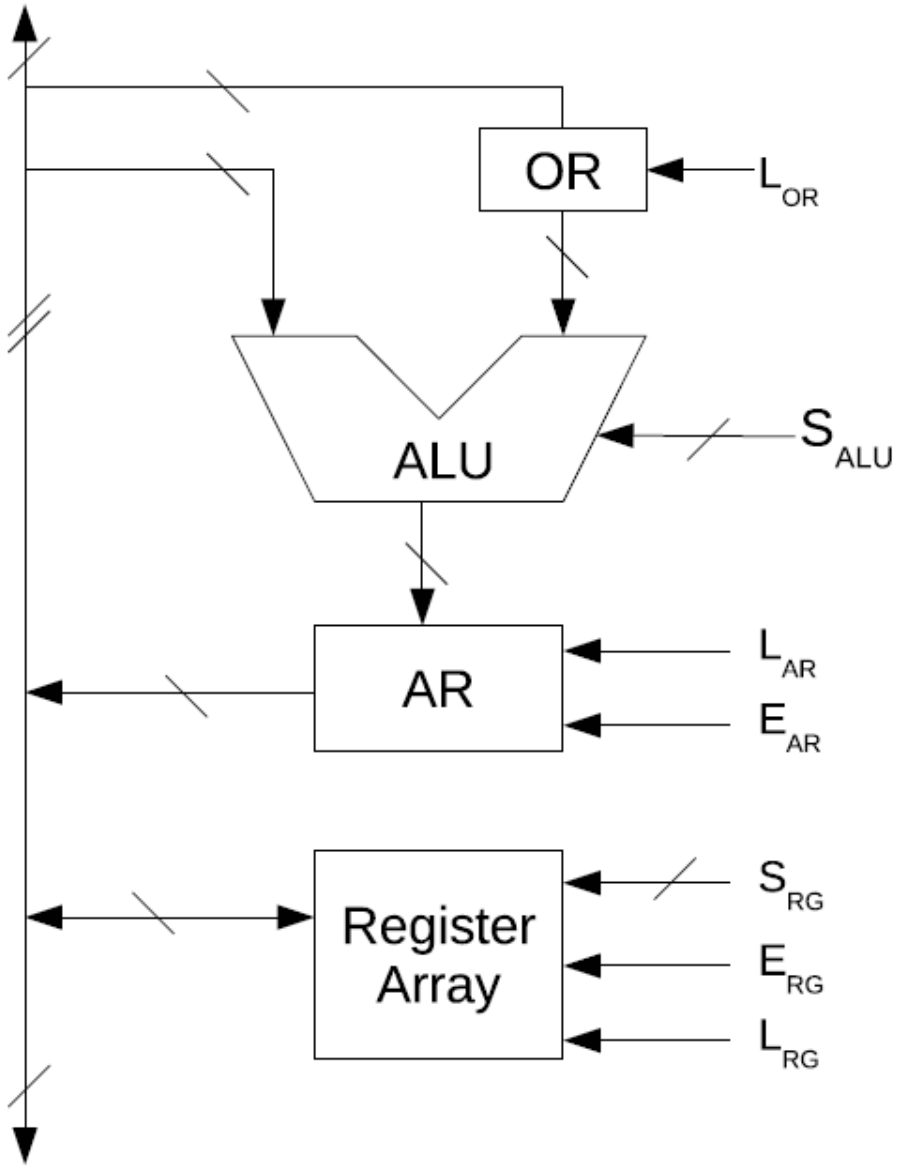
ALU on a bus

- It is clear that we can make the hardware perform several steps by carefully selecting the control signals to different units that are active in each clock cycle
- We can also get larger “operations” implemented using multiclock sequences of such combinations
- Each such clock cycle is typically referred to as a microcycle, which is the basic time unit in which something happens within the processor



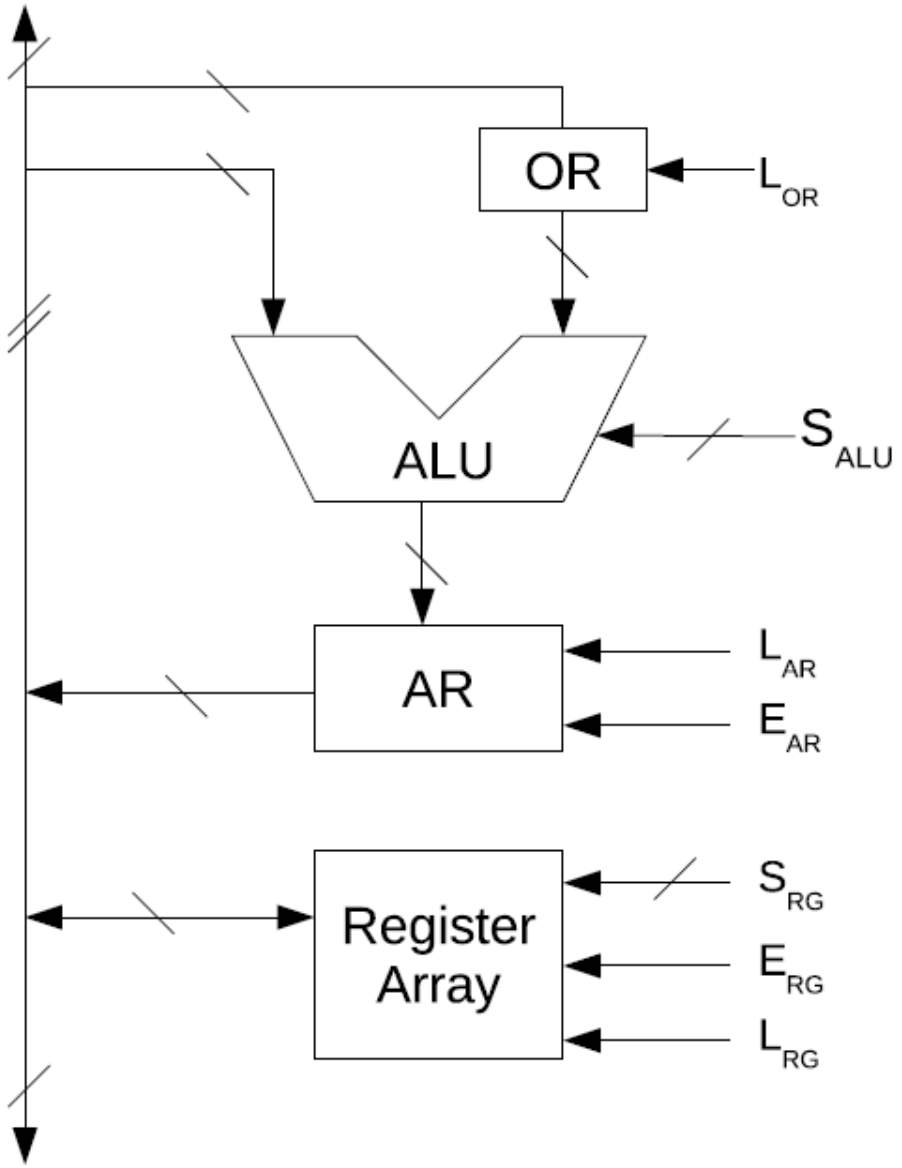
Enhanced singlebus architecture

- 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 24 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



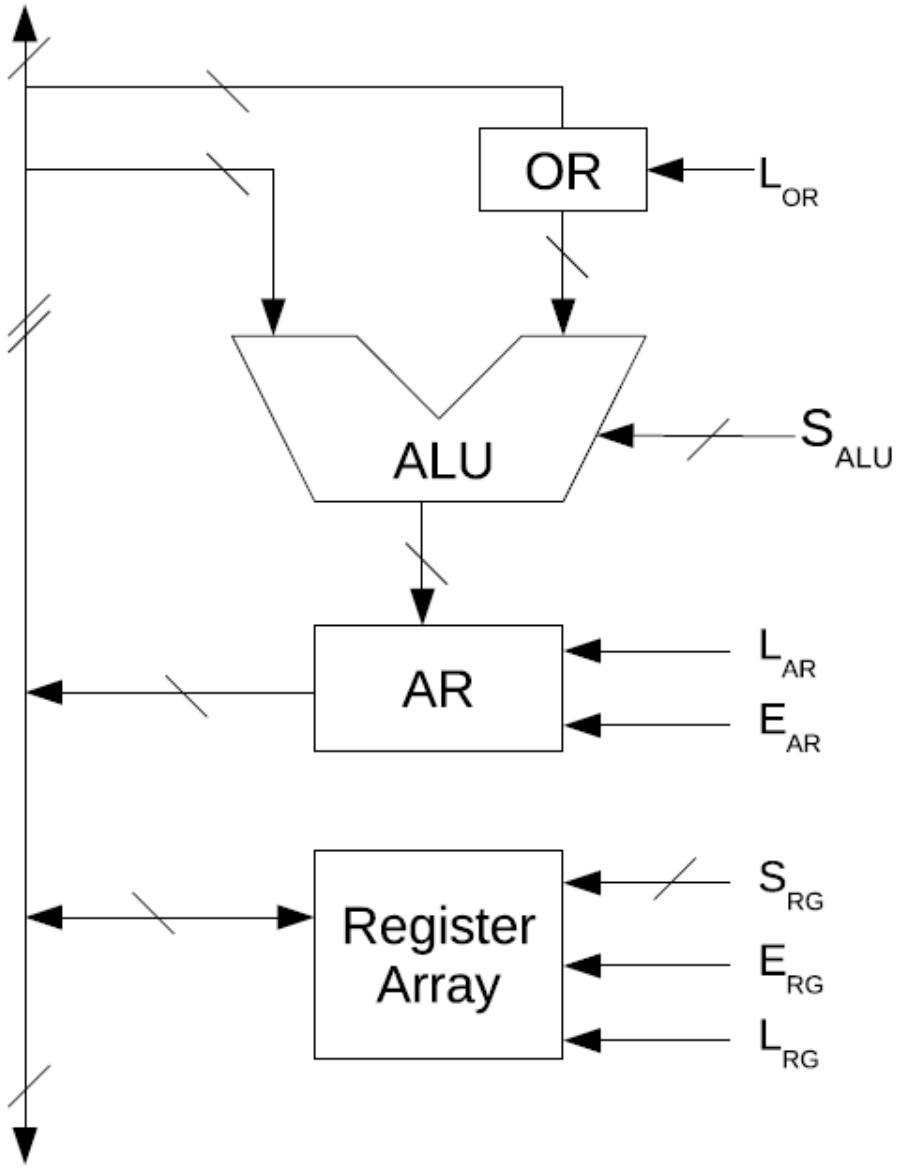
Enhanced singlebus architecture

- We also decide to make the registers 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: none, add, subtract, logical AND, OR, XOR, and pass left. The ALU takes the left operand from the bus and the right one from OR



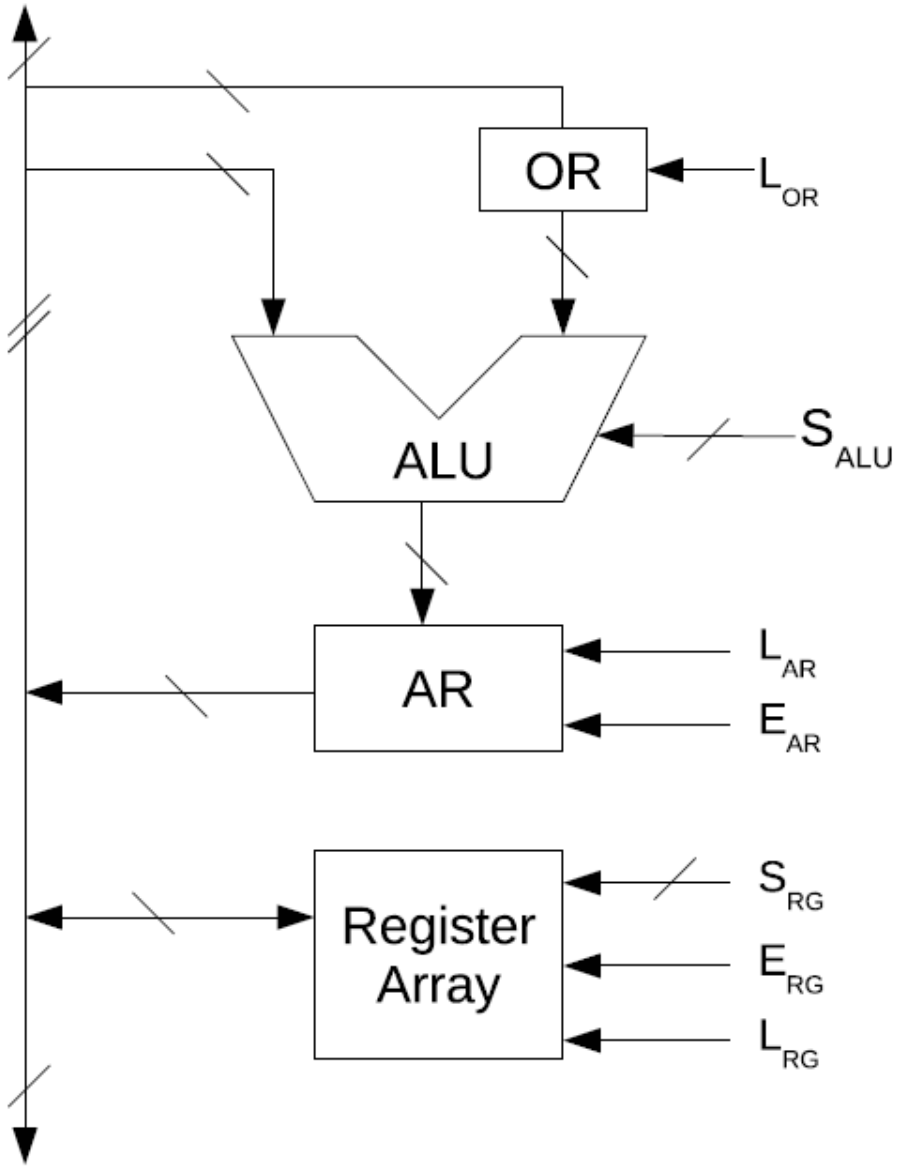
Enhanced singlebus architecture

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



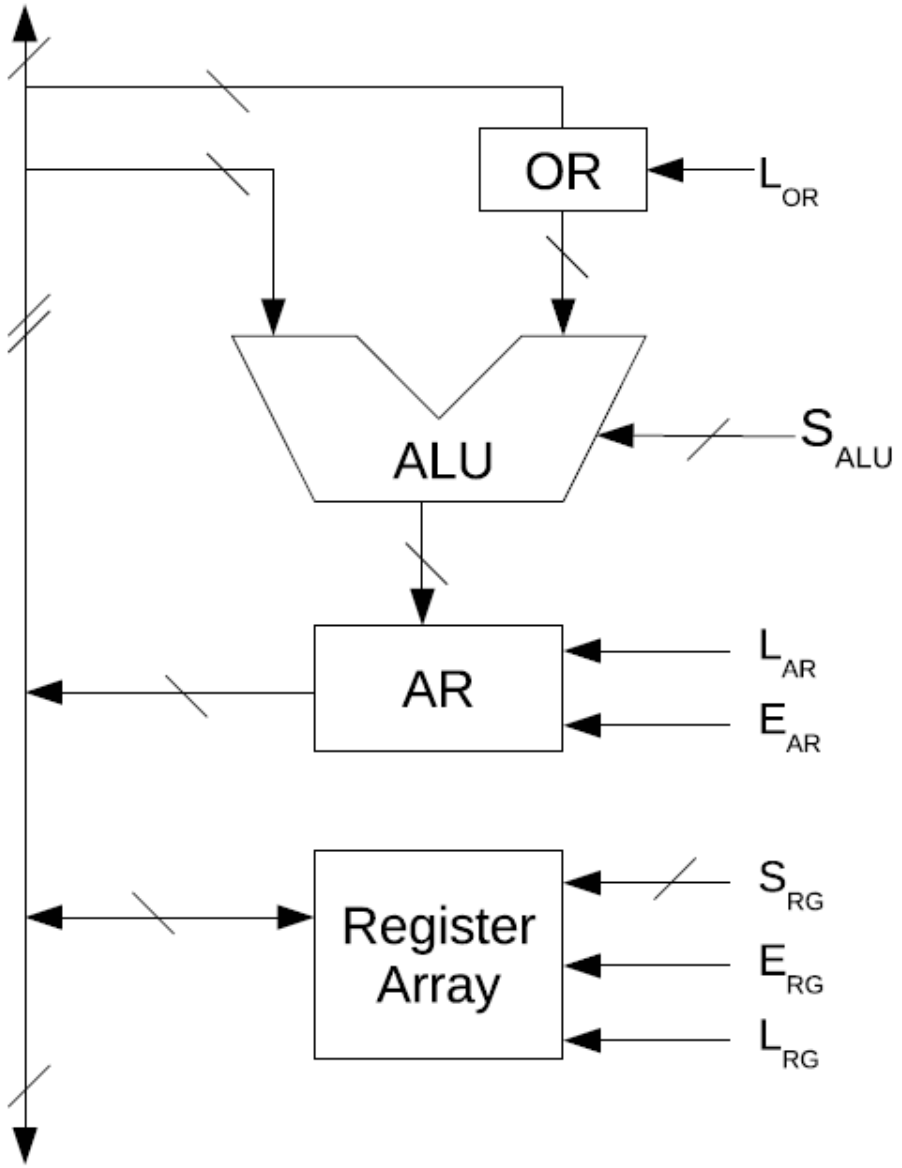
Enhanced singlebus architecture

- 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 load and store respectively
- Eg: LOAD R4 and STOR R9 can be implemented as follows:
- LOAD R4:
  - Ck0: E_{RG}, L_{AR}, $S_{RG} \leftarrow 4$, SALU $\leftarrow$ PASS0
- STOR R9:
  - Ck0: E_{AR}, L_{RG}, $S_{RG} \leftarrow 9$



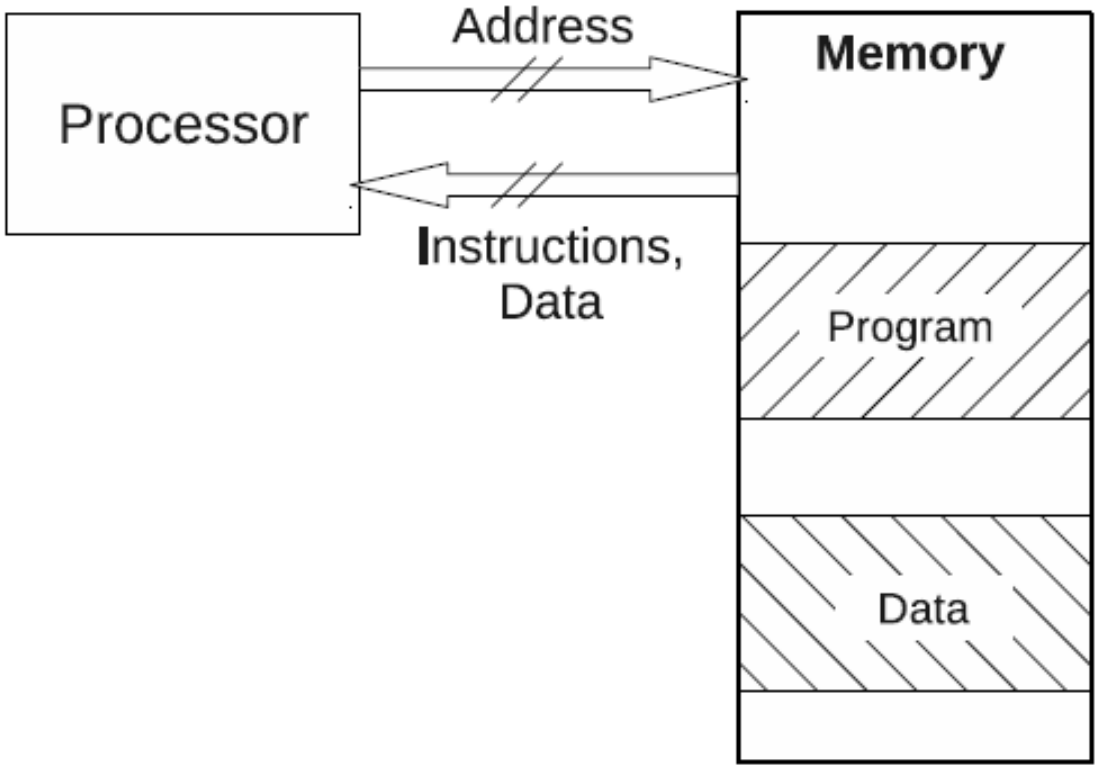
Instructions

- 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



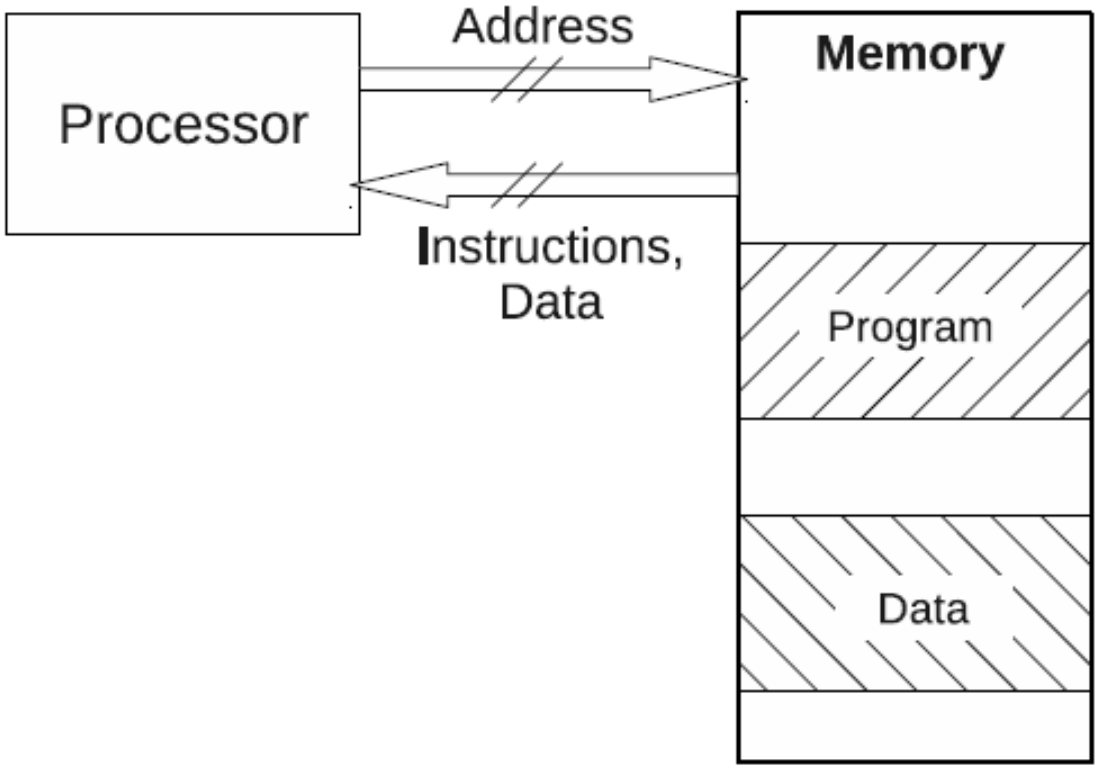
Instructions

- Both the data to be processed and the program can be stored in the same memory (the *von Neumann* or stored-program 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