EC2.101 – Digital Systems and Microcontrollers

Lecture 30 – Processor design 5

Implementing instructions – ALU

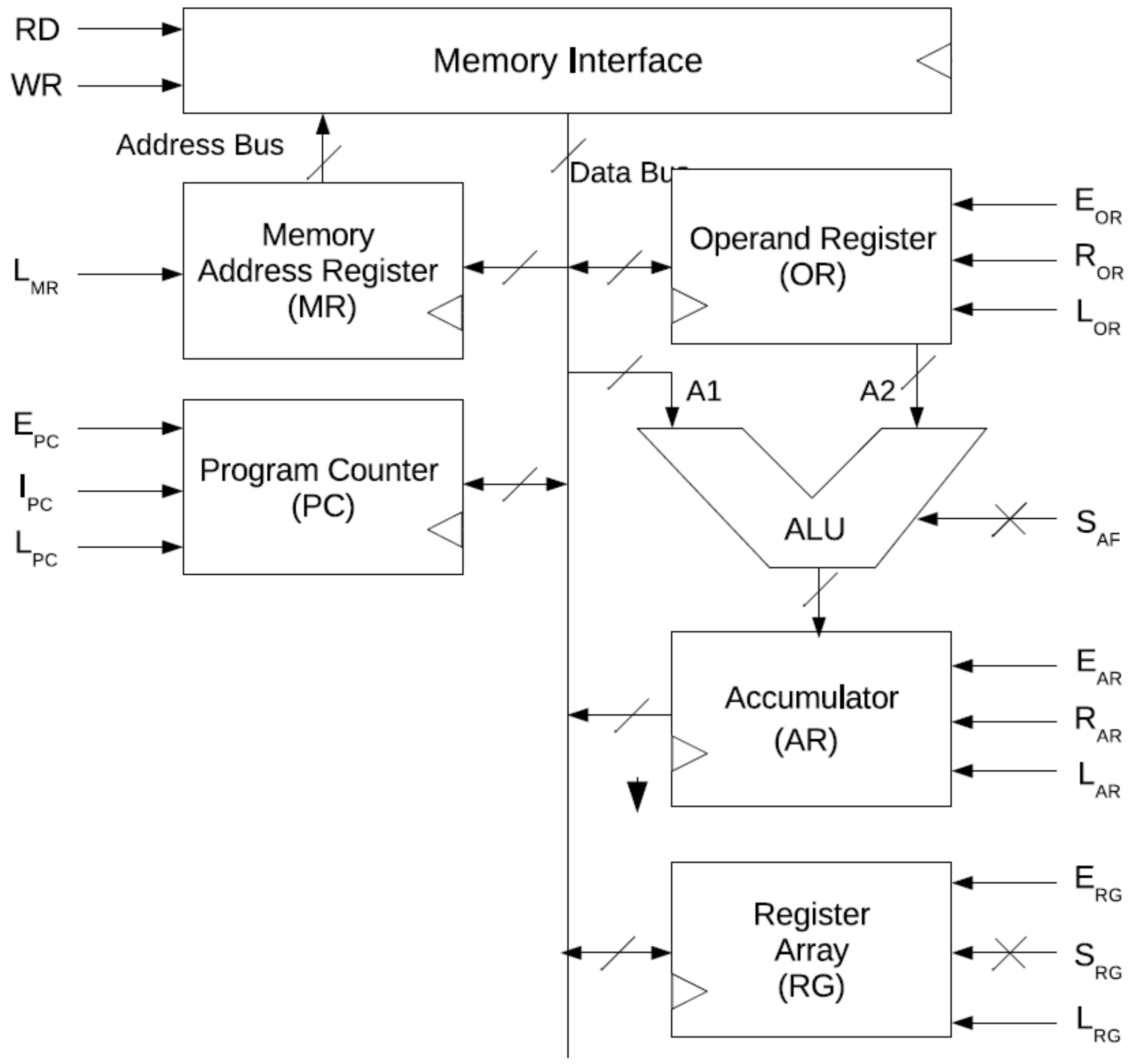
- Consider the simple instruction:

  ADD <R>

- This instruction can be executed in two clock cycles:

  Ck 3: We need to enable RG and load the OR with the value [<R>] using the select lines for RG

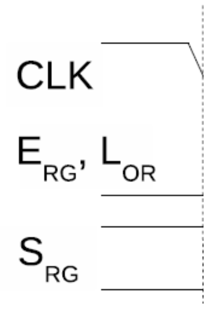
  Ck 4: Enable AR, set the ALU select lines for ADD operation, activate load AR

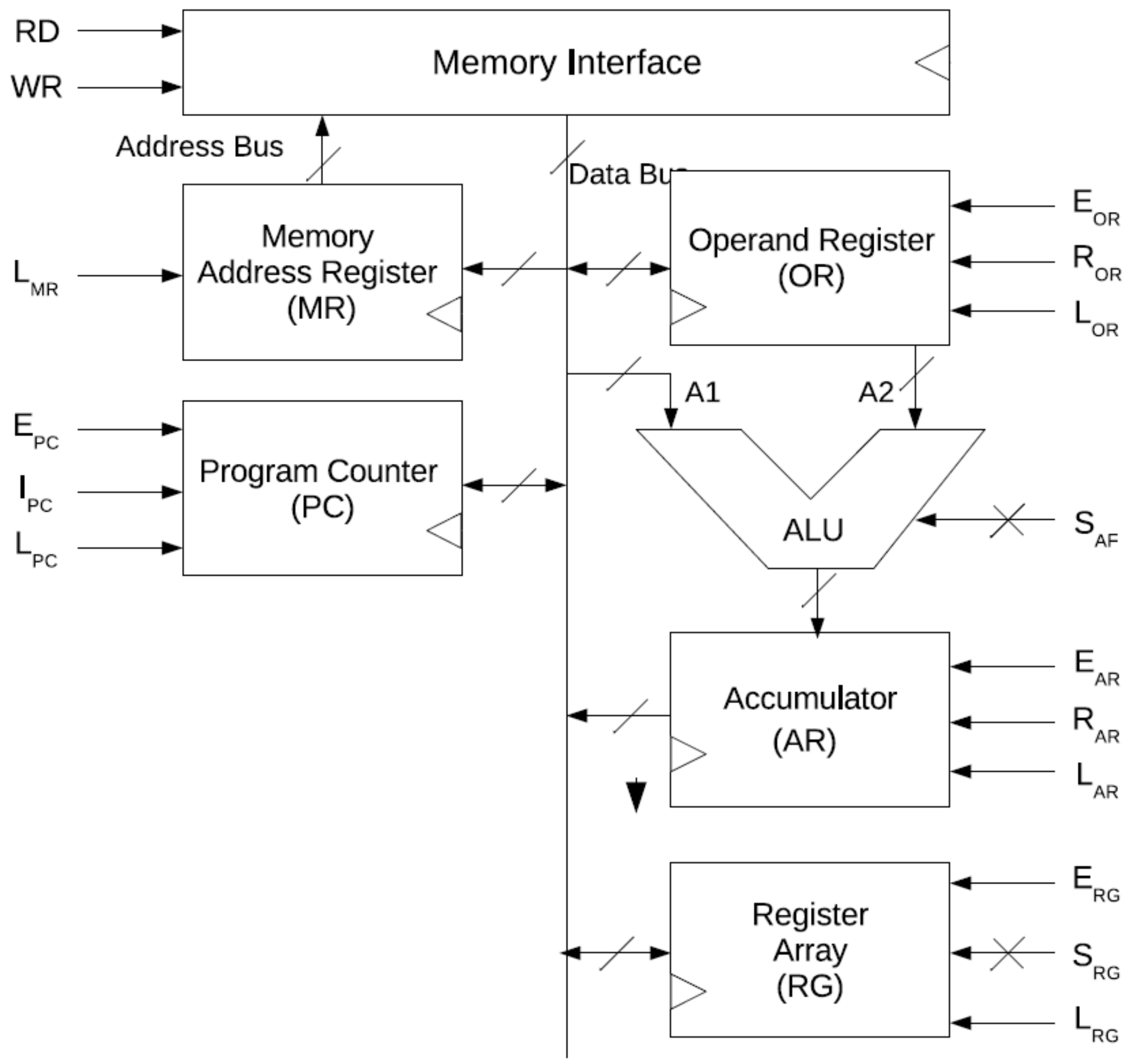


add <R>	Ck 3. E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ ADD

Implementing instructions – ALU

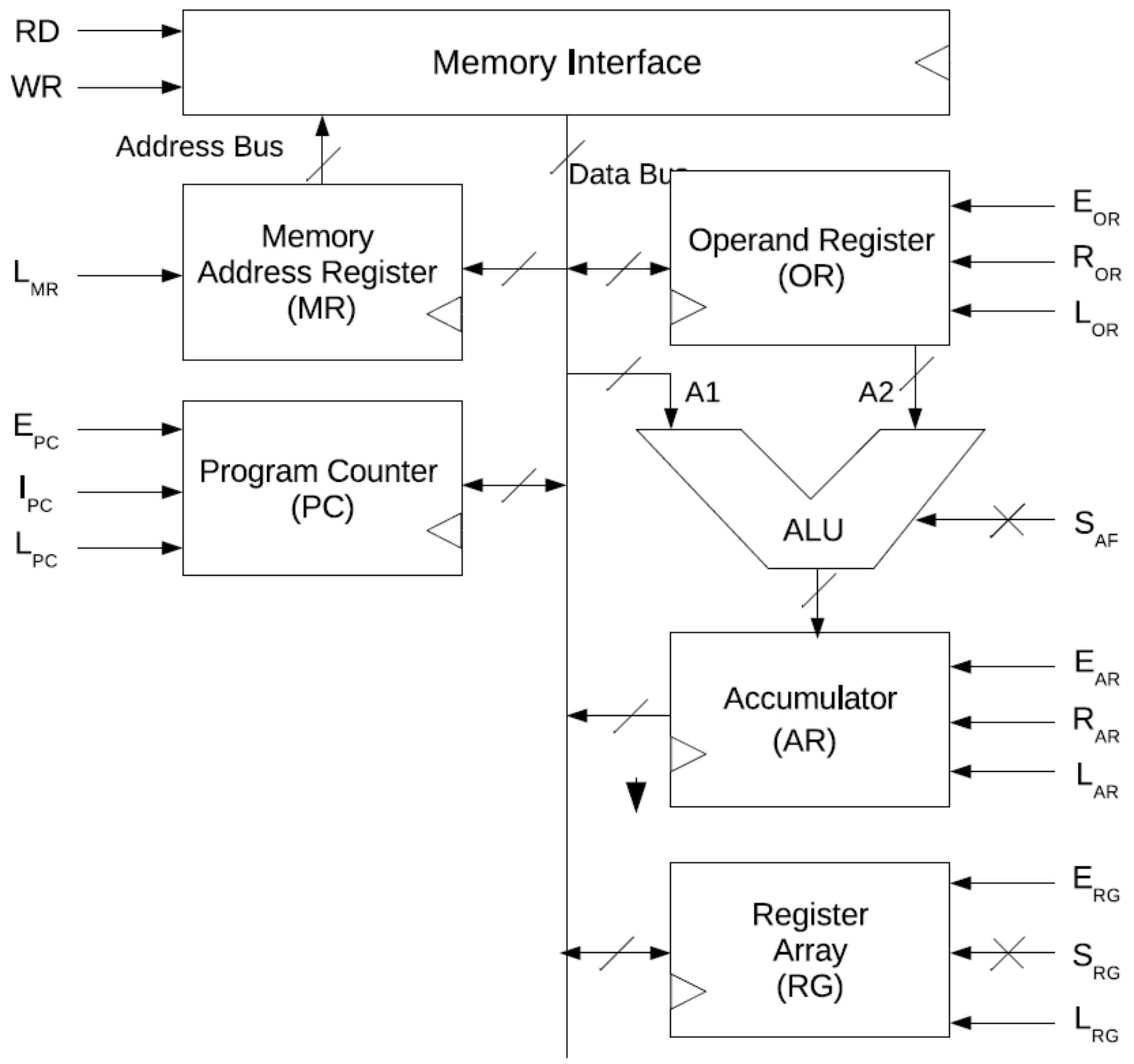
ADD <R>





Implementing instructions – ALU

Instruction	Control Signals	Select Signals
add <R>	Ck 3. E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ ADD
sub <R>	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ SUB
xor <R>	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ XOR
and <R>	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ AND
or <R>	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ OR
cmp <R>	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ CMP
nop	Ck 3: End	-

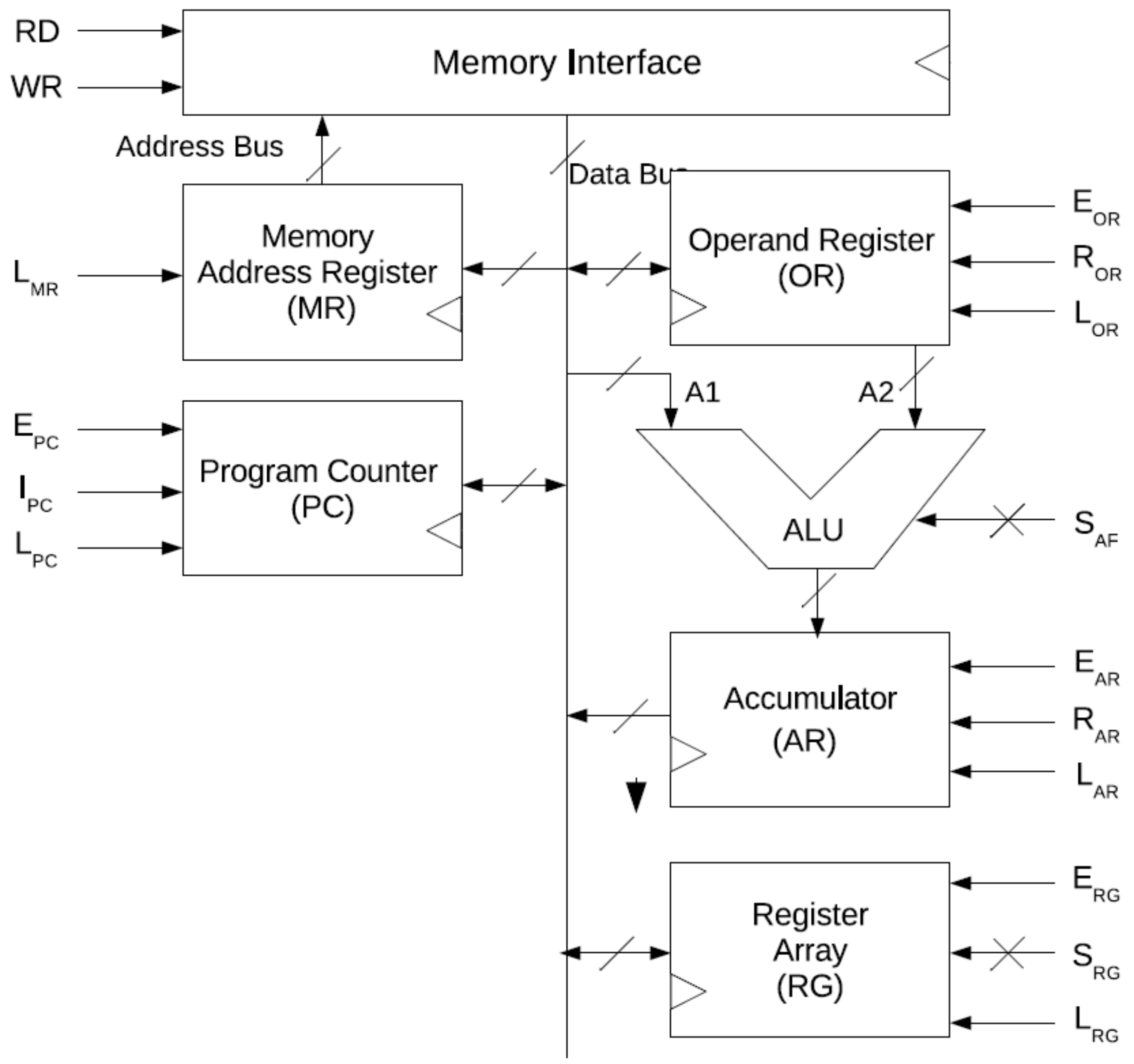


Implementing instructions

- A clock cycle in which one basic operation is performed is called a *microcycle*
- The combination of control signals that are active (or at level 1) in a microcycle determines what operation is performed in that cycle
- The operation performed in a microcycle is often referred to as a *microinstruction*
- The execution of each machine instruction (such as ADD <R>) needs one or more microcycles
- Faster instructions take fewer microcycles and vice versa
- The number of microcycles needed for different machine instructions depends on the processor architecture – some processors are “hardwired” to perform certain instructions very rapidly

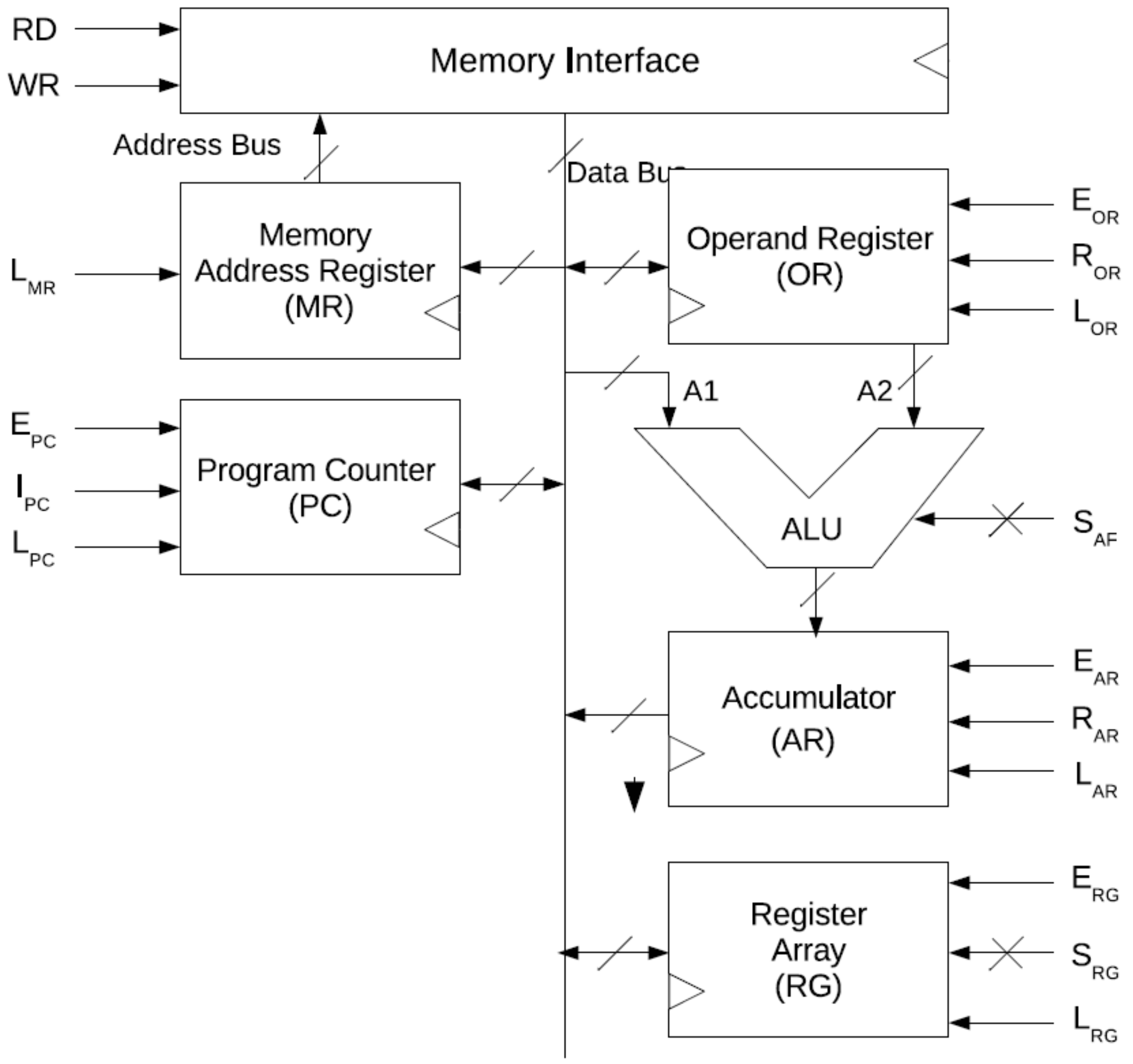
Implementing instructions – data movement

- Moving from AR to a register is achieved using the *movd* instruction
- It is quite straightforward to implement, enable AR and load RG, and needs only one cycle to execute
- To load form register to ALU: we load the register value to the bus by setting S_{RG} and choosing the pass option of the ALU
- The register contents are available at the input of AR in the same clock cycle
- If L_{AR} is also active in that clock, the data will go from the register to ALU input through the bus, pass through the ALU to AR and be stored into it all in *one clock cycle*!



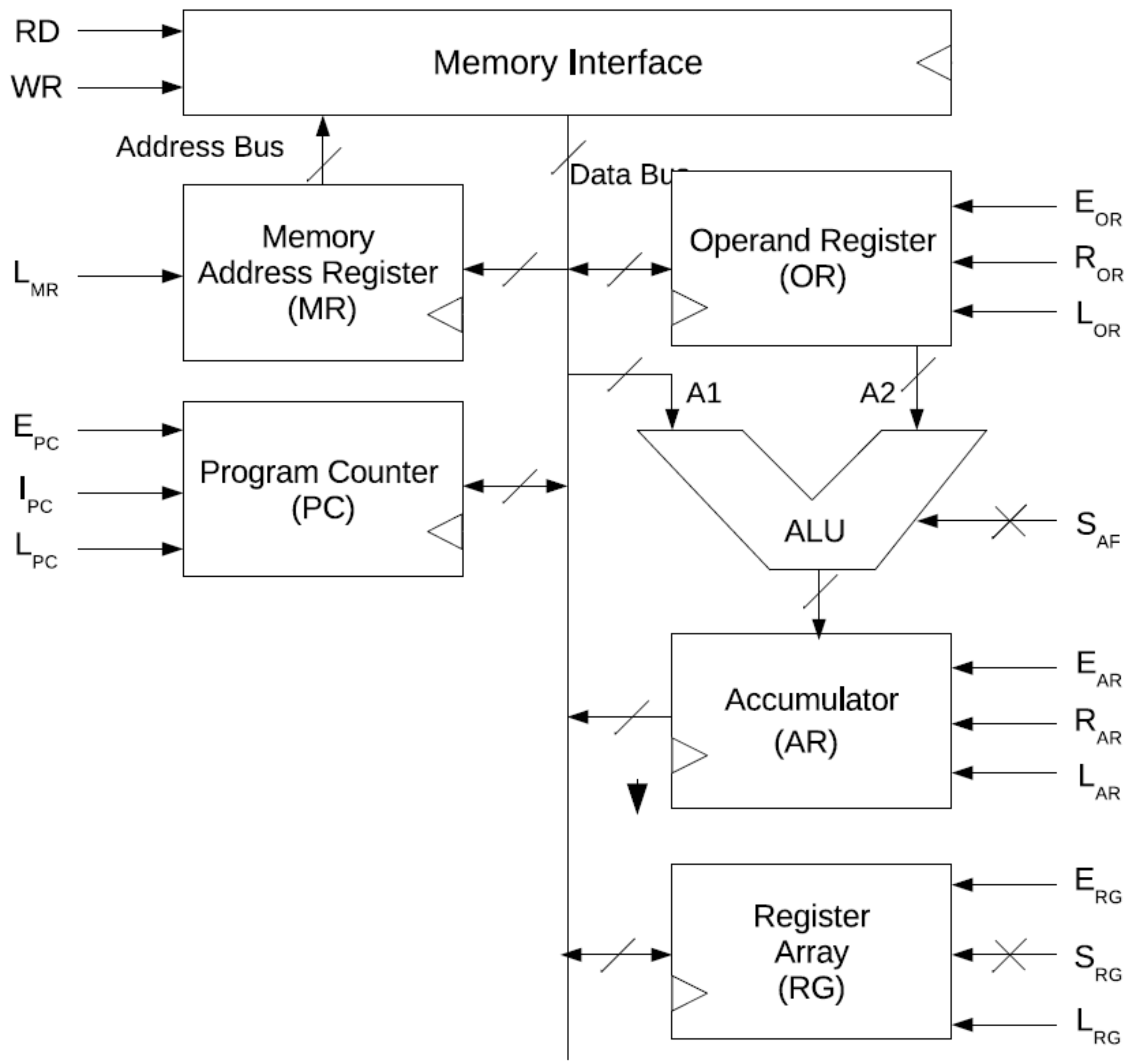
Implementing instructions – data movement

- Now, we look at the two data movement instructions, namely, *load* and *stor*
- The load instruction reads a value from the memory to a register
- The address of the memory location is given in AR
- The memory sits outside of the processor and is accessed by giving it an address through the MAR register and a command through the RD and WR lines, as is appropriate
- The first microcycle of execution moves the address from AR to MAR for both instructions
- This is done using the E_{AR} and L_{MR} signals



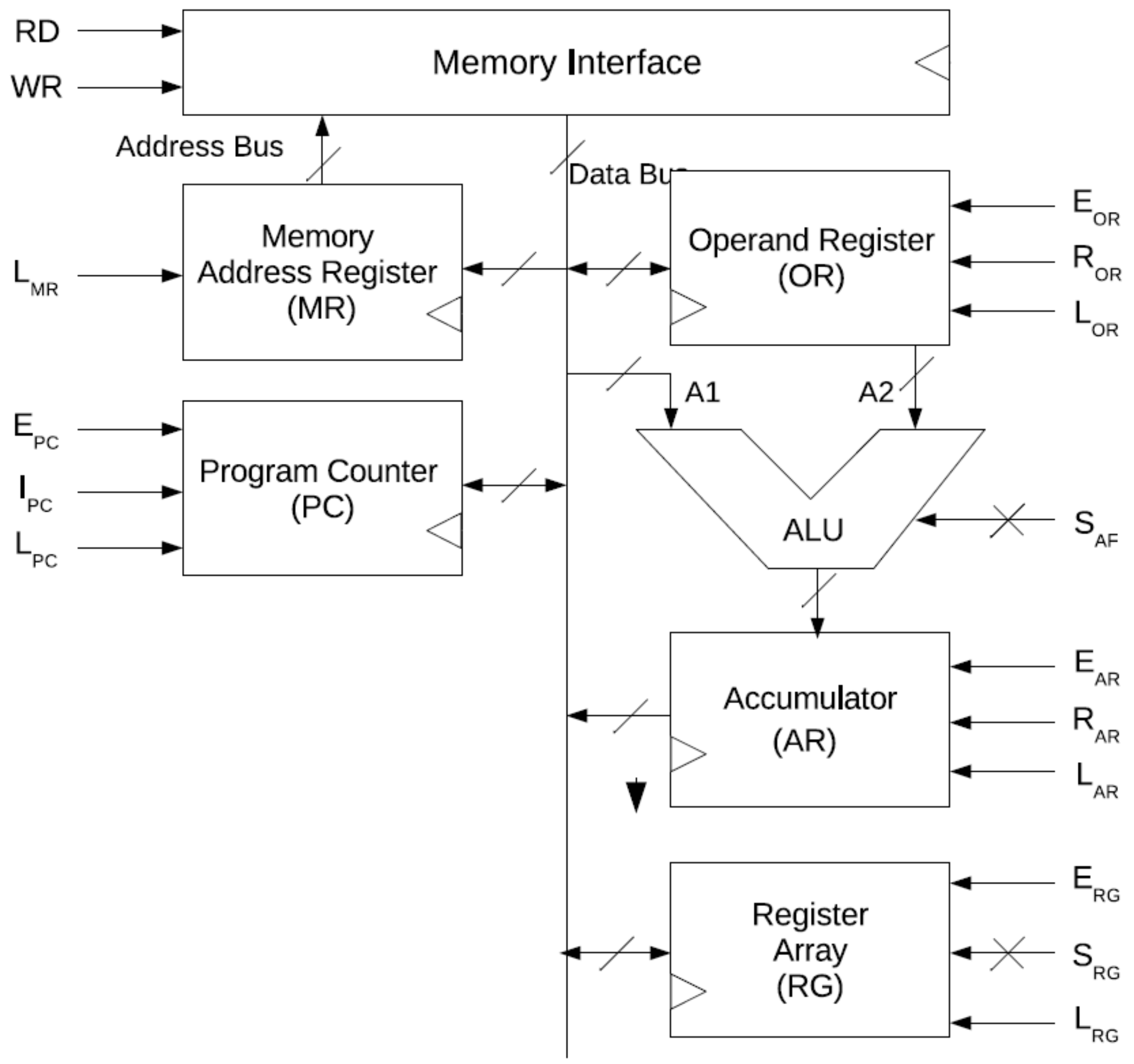
Implementing instructions – data movement

- In case of *load*, the next microcycle asks the memory to read the location using RD and L_{RG} is activated
- In case of stor, the next microcycle activates WR and E_{RG}
- In case of load, we assume the value will be available on the data bus before the end of the clock cycle
- Thus, the memory is treated like an external register file, whose address (or select) is given through MAR
- However, in practice, the memory is significantly slower than the registers and the read cannot complete in the same clock cycle
- We will ignore that aspect as we are designing a very simple processor
- Thus, the data movement instructions only take 2 clock cycles for their execution on our architecture



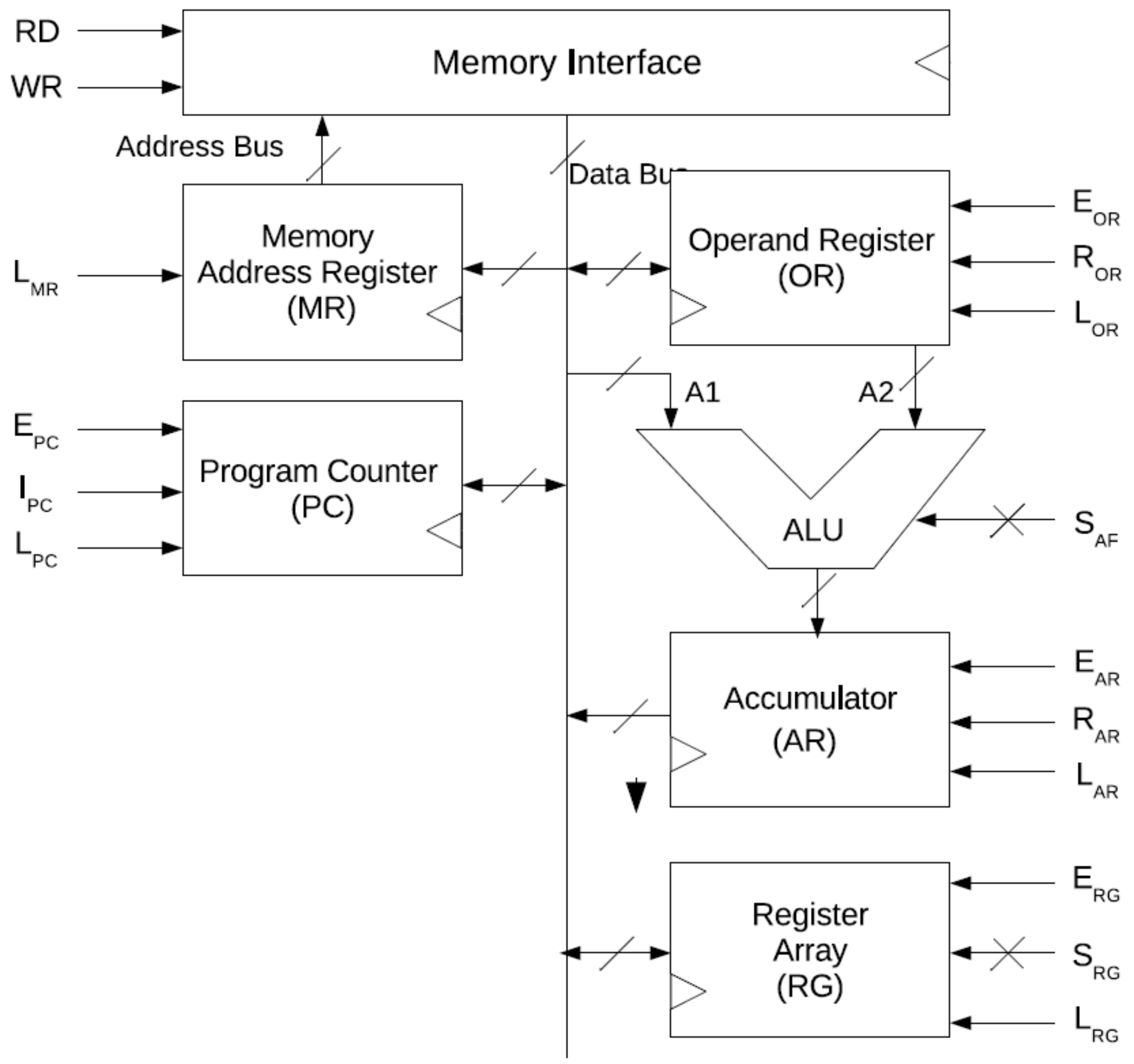
Implementing instructions – data movement

- The third data movement operation uses an immediate argument
- This is very similar to *load* except for the specification of the source memory address
- The source value is stored immediately along with the instruction
- As we have seen before, the immediate argument xx is stored in a memory location with address (*k+1*) if the opcode for *movi* is stored in a memory location with address *k*
- Moreover, we assume that as the opcode for *movi* is fetched from *k*, the PC value is incremented by 1 to point to the next instruction



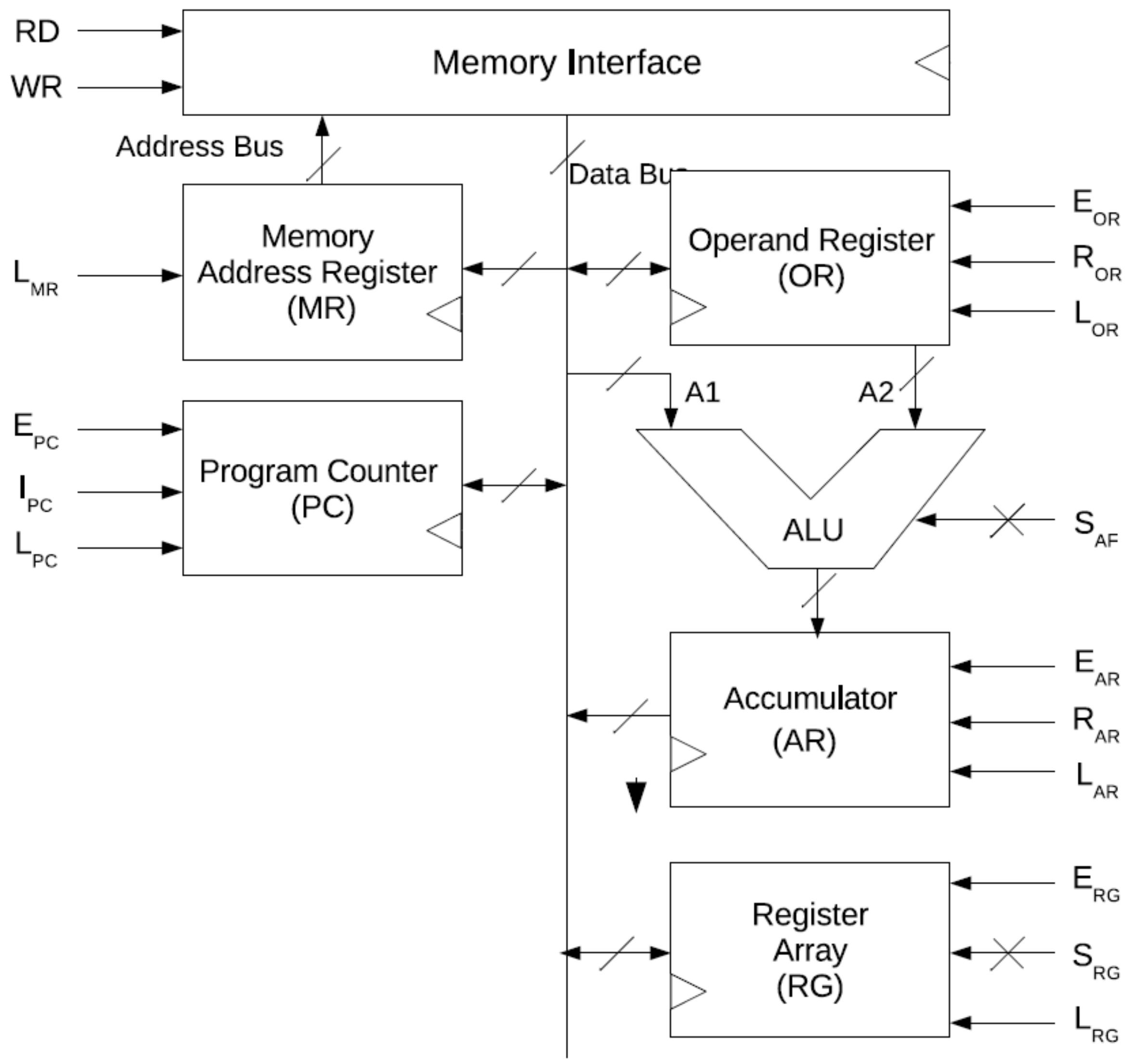
Implementing instructions – data movement

- Thus, when the execution of *movi* starts, the PC is pointing to the word following the opcode
- This word holds the immediate operand xx
- Thus, the situation is similar to *load*, except for the PC supplying the address of the operand instead of AR
- Thus, the execution of *movi* proceeds very similarly: E_{PC}, L_{MR}
- However, the PC needs to point to the next real opcode at the end of executing *movi*
- We achieve this by incrementing PC while it is loaded onto MAR, by enabling the I_{PC} control signal
- Thus, the microcycle activates: E_{PC}, L_{MR}, I_{PC}
- Then the memory can be read as before



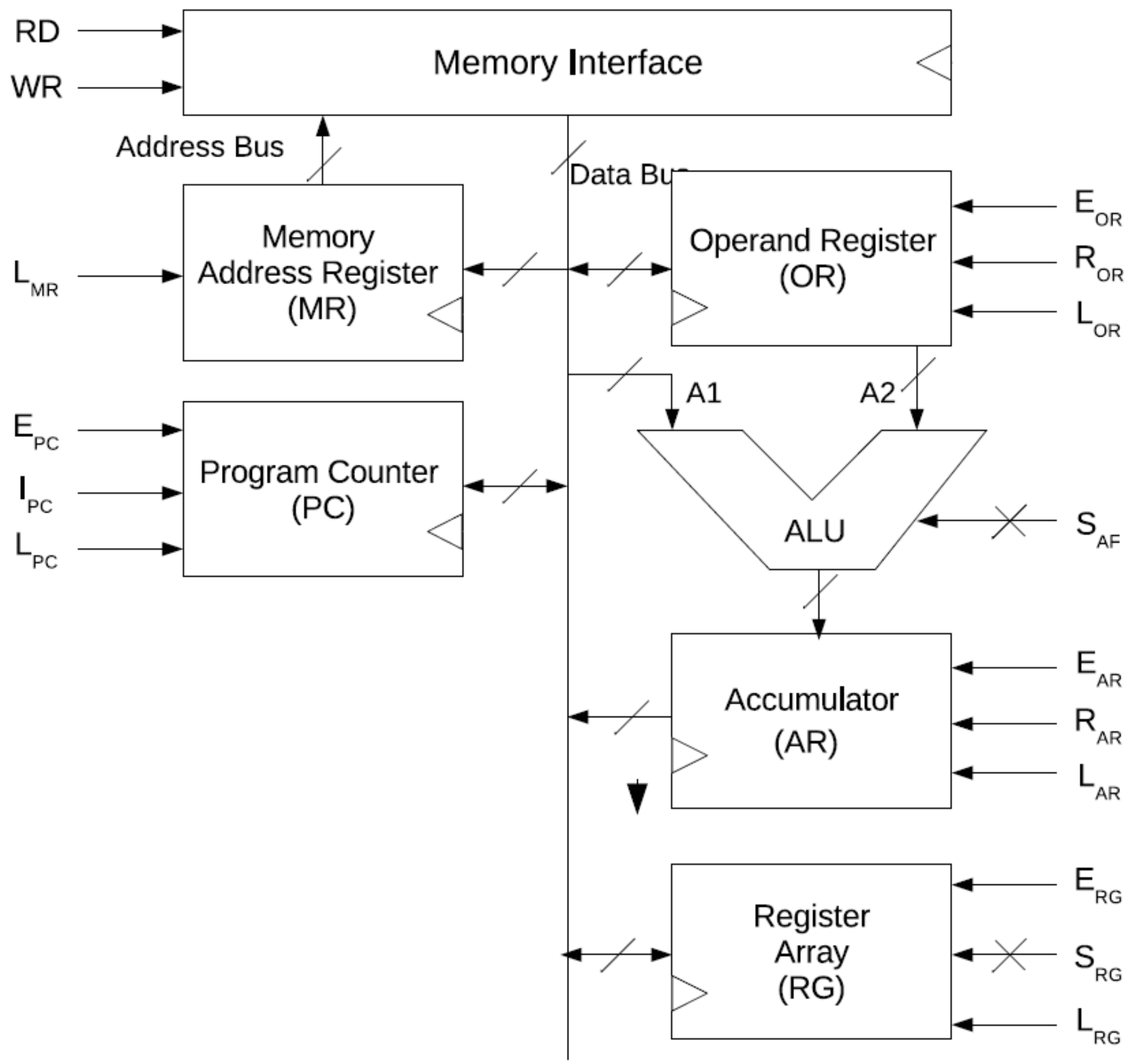
Implementing instructions – ALU immediate

- The only difference between an instruction that uses a register argument and one that uses an immediate argument is the source of the argument
- Earlier, we loaded the source from the selected register in one clock to OR through the bus
- In immediate case, we have to get it from the memory, and we know that the PC holds the operand's address when execution starts
- All arithmetic and logic instructions can be implemented keeping this in mind
- Note that these instructions require 3 clock cycles each for their execution



Implementing instructions

Instruction	Control Signals	Select Signals
movs <R>	Ck 3: E_{RG}, L_{AR}, End	S_{RG} ⟵ <R>, S_{ALU} ⟵ PASS0
movd <R>	Ck 3: E_{AR}, L_{RG}, End	S_{RG} ⟵ <R>
load <R>	Ck 3: E_{AR}, L_{MR} Ck 4: RD, L_{RG}, End	- S_{RG} ⟵ <R>
stor <R>	Ck 3: E_{AR}, L_{MR} Ck 4: E_{RG}, WR, End	- S_{RG} ⟵ <R>
movi <R> xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{RG}, End	- S_{RG} ⟵ <R>
adi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - S_{ALU} ⟵ ADD
sbi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - S_{ALU} ⟵ SUB
xri xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - S_{ALU} ⟵ XOR
ani xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - S_{ALU} ⟵ AND
ori xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - S_{ALU} ⟵ OR
cmi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, End	- - S_{ALU} ⟵ CMP



Instruction fetch

- We are now ready to tackle instruction fetch - we know it involves reading a word from the memory, using the PC value as the address.
- This can be achieved using the following two microinstructions:

Ck 1: E_{PC}, L_{MR}, I_{PC}

Ck 2: RD, L_{IR}

- In the first cycle, PC value is loaded to MAR and the PC is simultaneously incremented, so that the next fetch will be from the next word in memory.
- In the second cycle, the memory word at the address given by MAR is read and the value obtained is loaded into a special *instruction register or IR*.
- The instruction register holds the entire opcode, which then needs to be decoded or deciphered to select one of the possible actions.