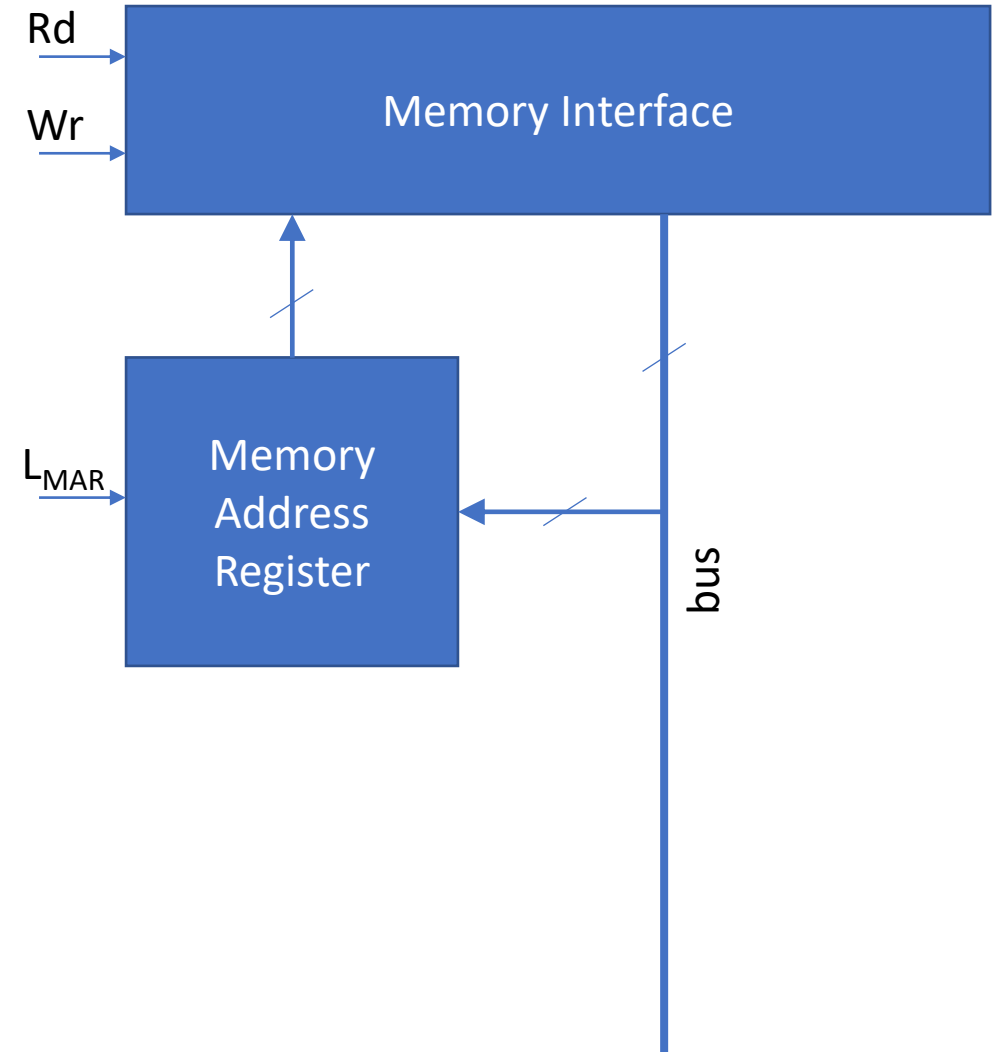


# Lecture 29 – Processor design 4

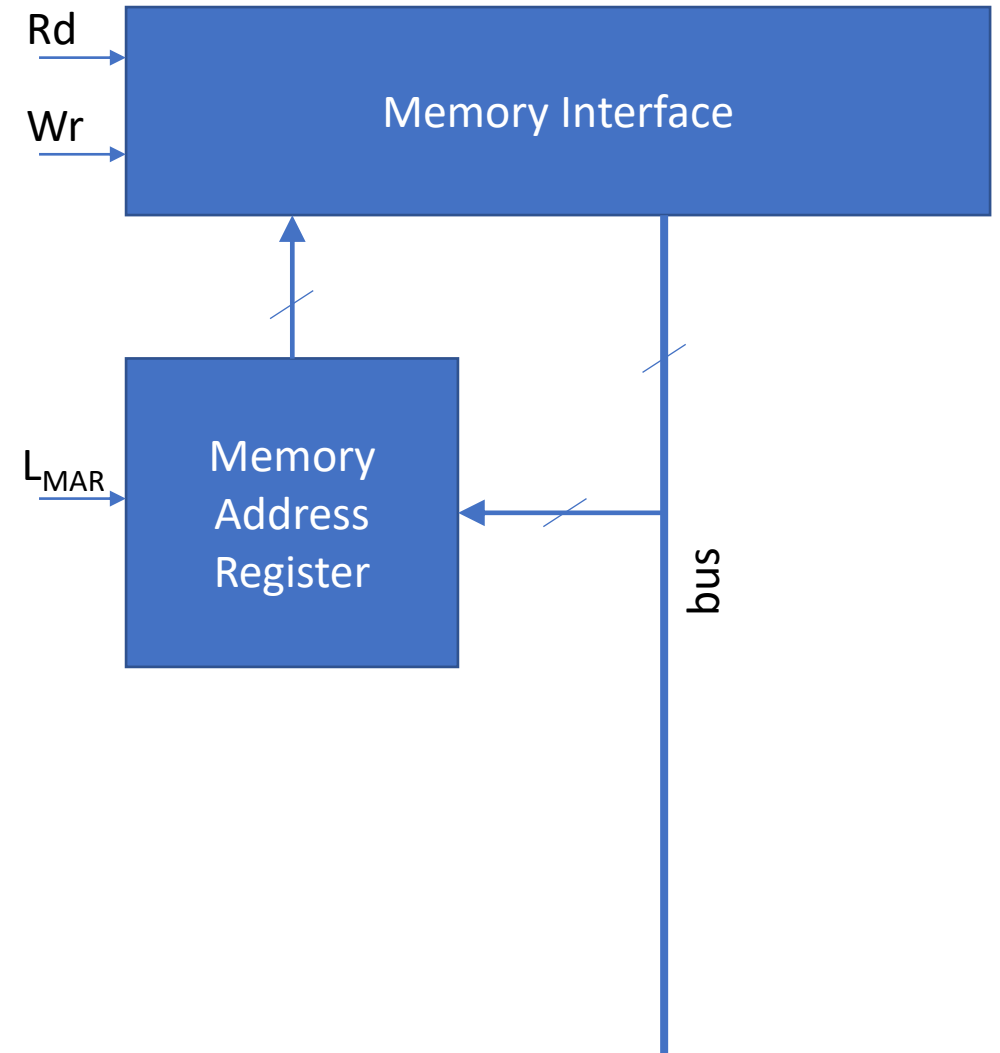
# Memory access

- 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



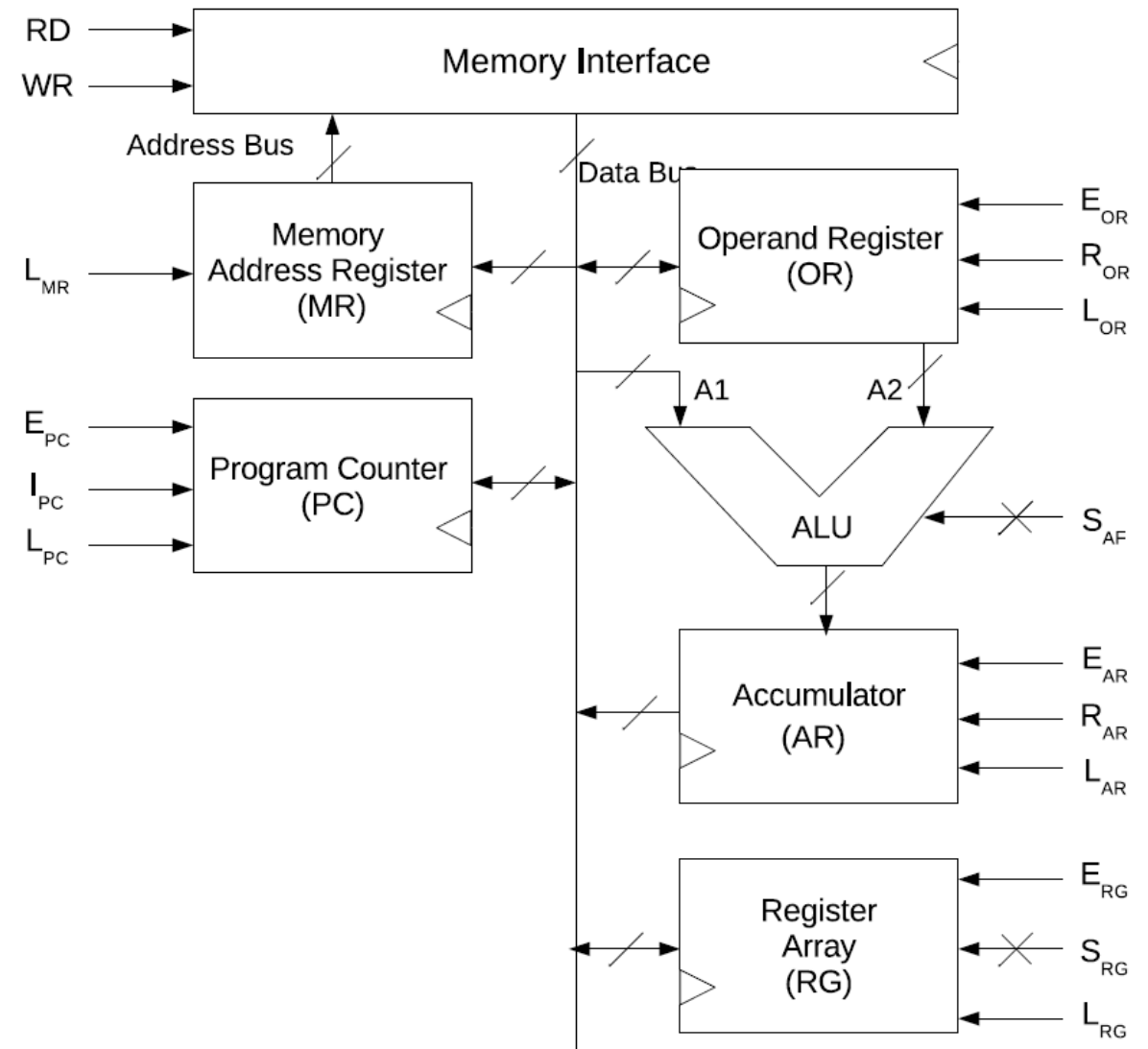
# Memory access

- 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



# Two more instructions

- We will introduce two more simple instructions: **NOP** instruction for no operation and **STOP** instruction to stop the processor.
- NOP does nothing: when it is executed nothing at all changes in the processor or memory.
- It may seem superfluous, but comes into use when nothing is required from the processor other than spending the required clocks to fetch and execute this instruction.
- The STOP instruction terminates the endless fetch-execute cycle that the processor is engaged in.
- The processor enters a state of complete inaction when the STOP instruction is executed.
- There is no way to come out of this state through a program as the processor has stopped looking at programs!
- Thus, the only way to come out is a hard reset through a reset button or through power cycling.

Assembly Instruction	Machine Code	Action
nop	00	-
stop	07	(Stops fetch)

# Implementing instructions – ALU

- Consider the simple instruction:

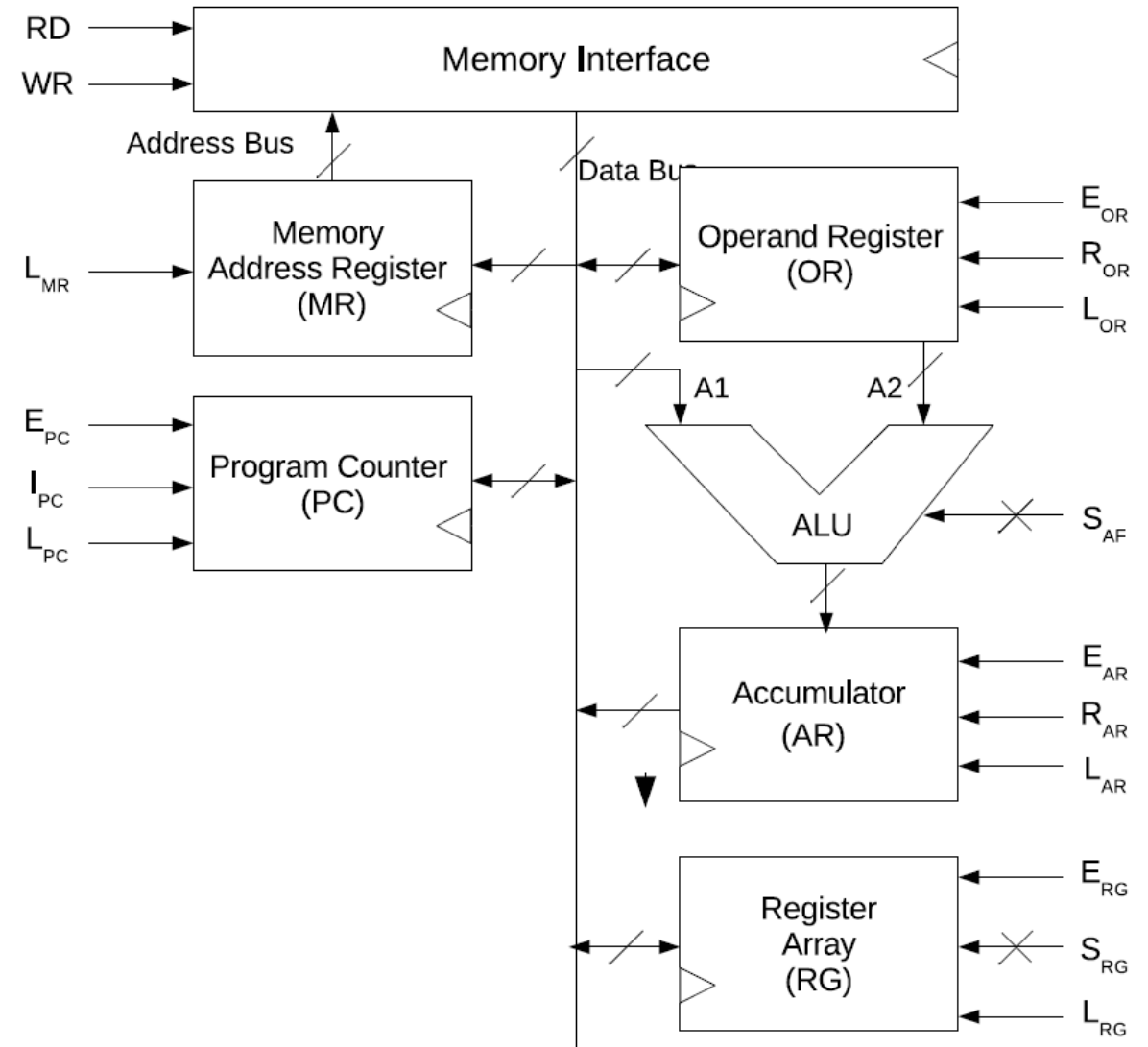
**ADD <R>**

- 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

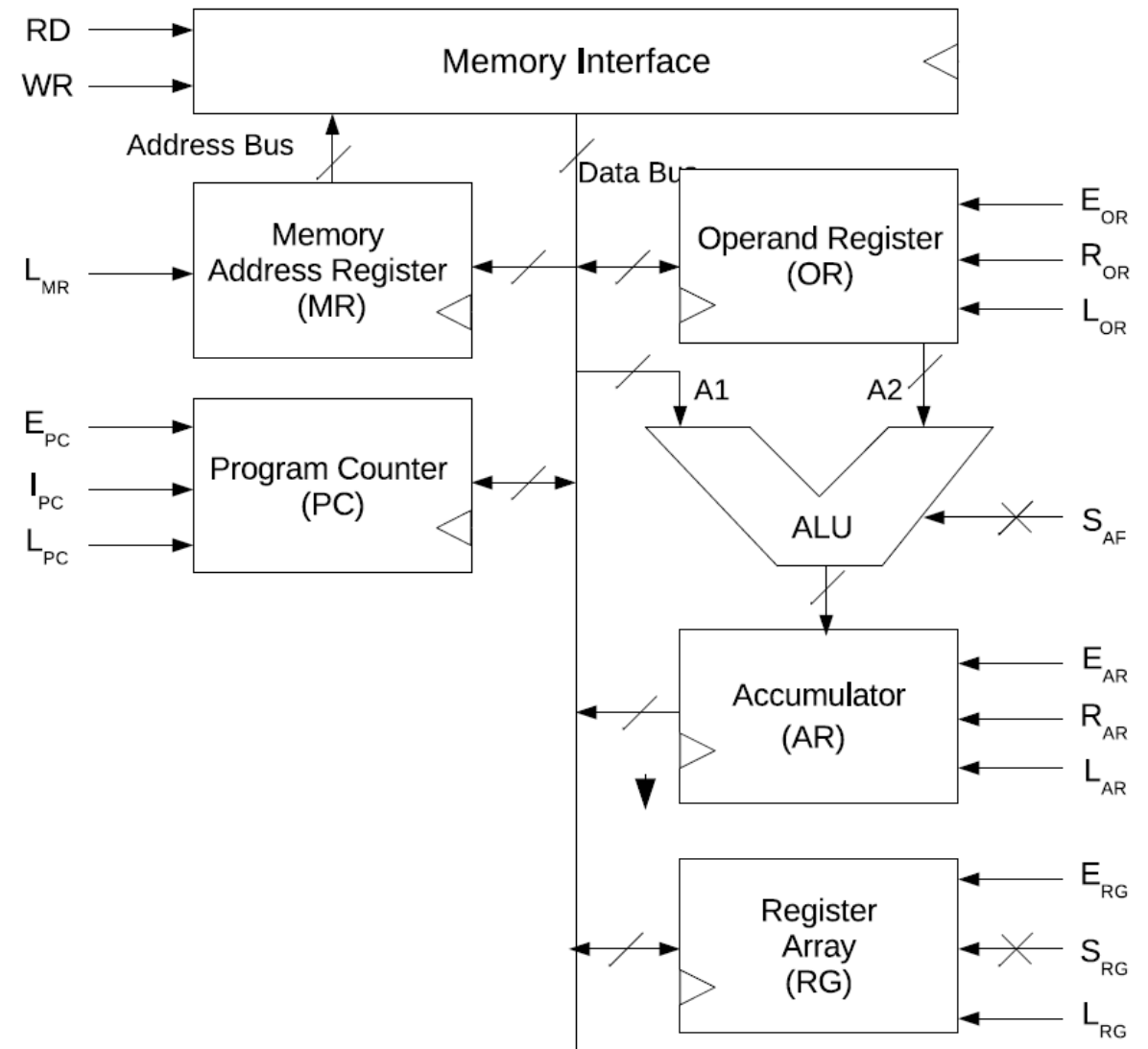
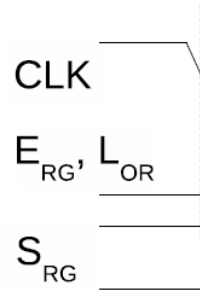
**Ck 4:** Enable AR, set the ALU select lines for ADD operation, activate load AR

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



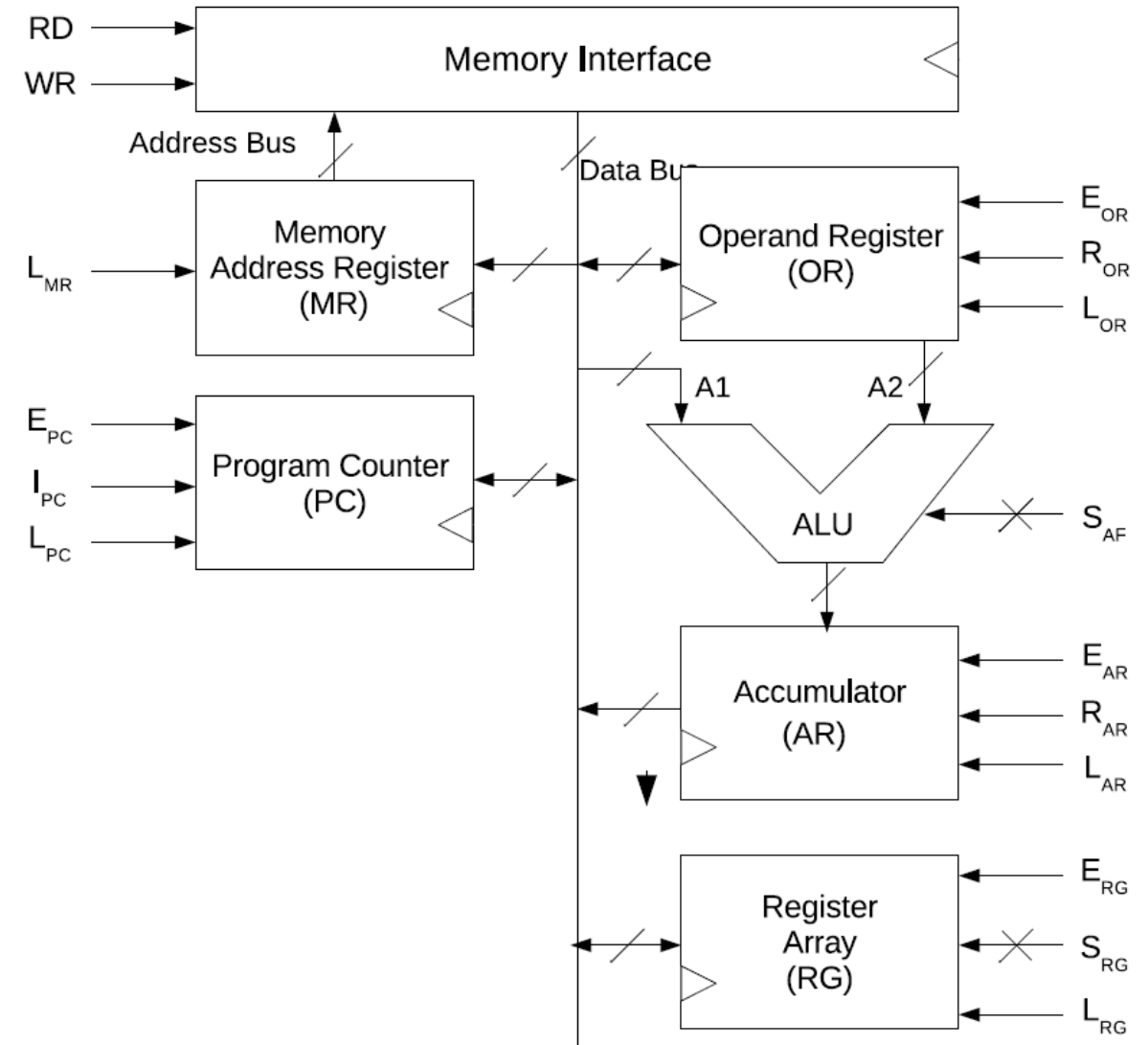
# Implementing instructions – ALU

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	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{ADD}$
sub <R>	Ck 3: $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , $L_{AR}$ , End	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{SUB}$
xor <R>	Ck 3: $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , $L_{AR}$ , End	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{XOR}$
and <R>	Ck 3: $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , $L_{AR}$ , End	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{AND}$
or <R>	Ck 3: $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , $L_{AR}$ , End	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{OR}$
cmp <R>	Ck 3: $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , End	$S_{RG} \leftarrow \langle R \rangle$ $S_{ALU} \leftarrow \text{CMP}$
nop	Ck 3: End	-



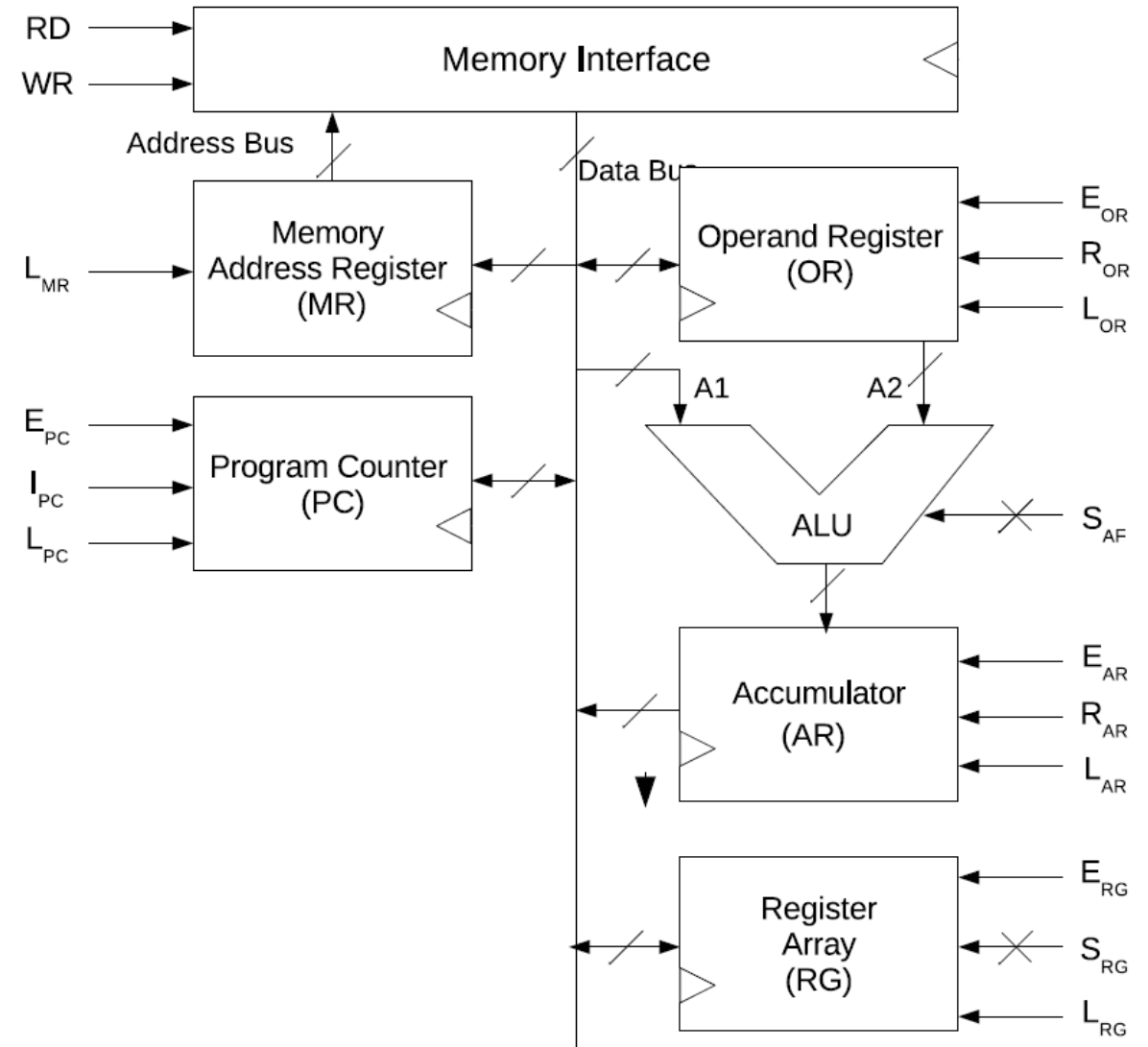


# Implementing instructions

- 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

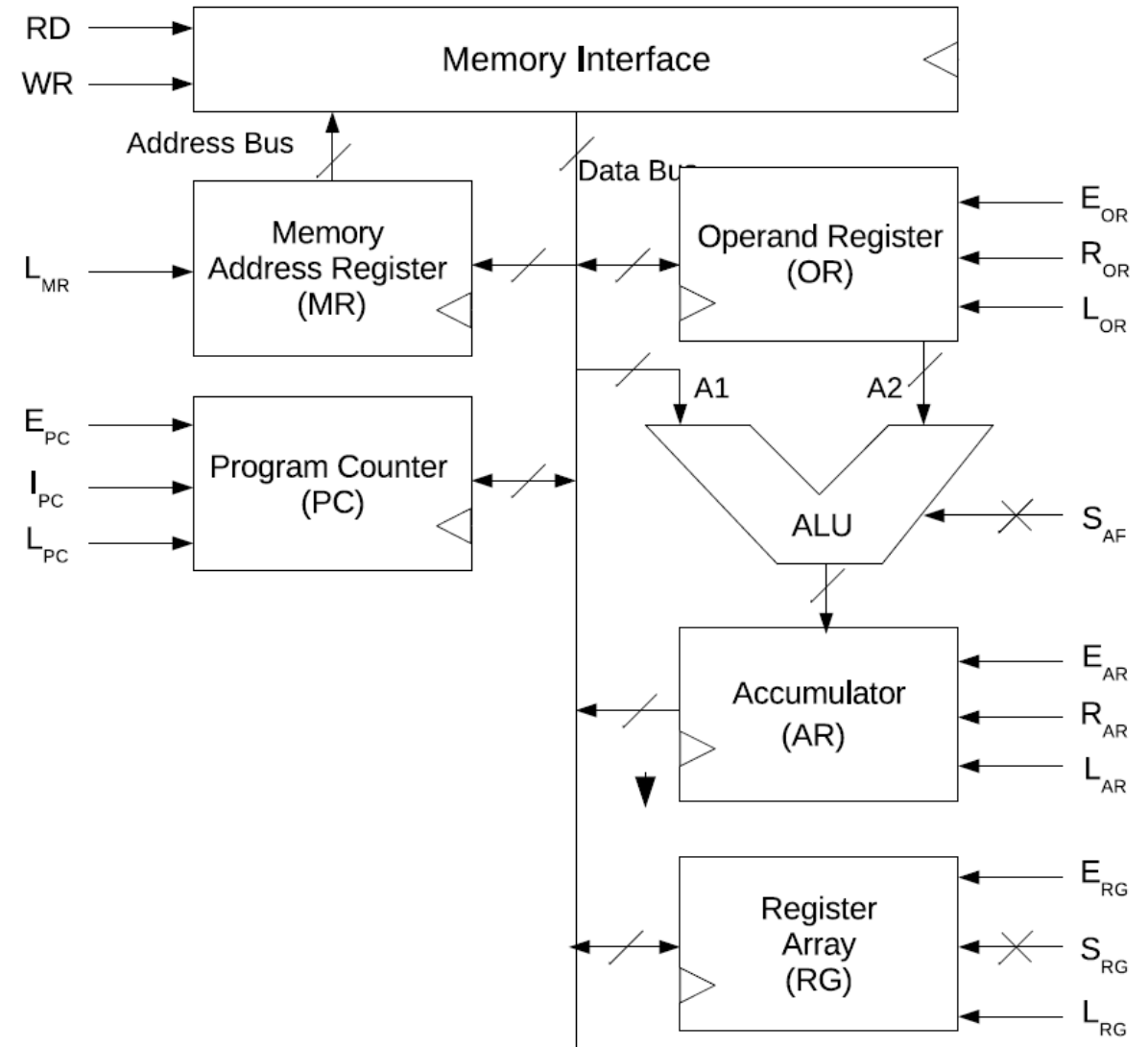
# Implementing instructions – data movement

- 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 from 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*!



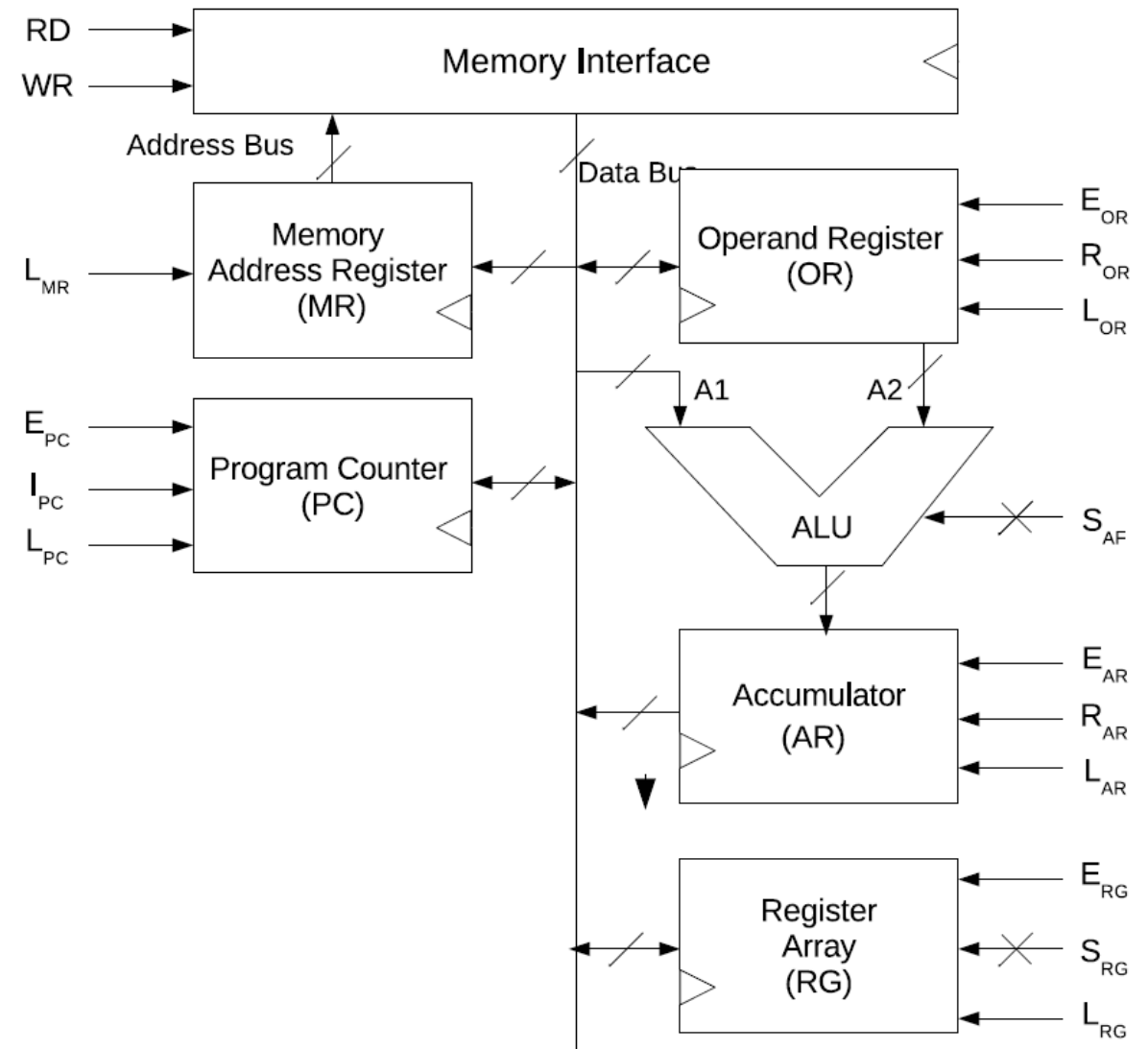
# Implementing instructions – data movement

- 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



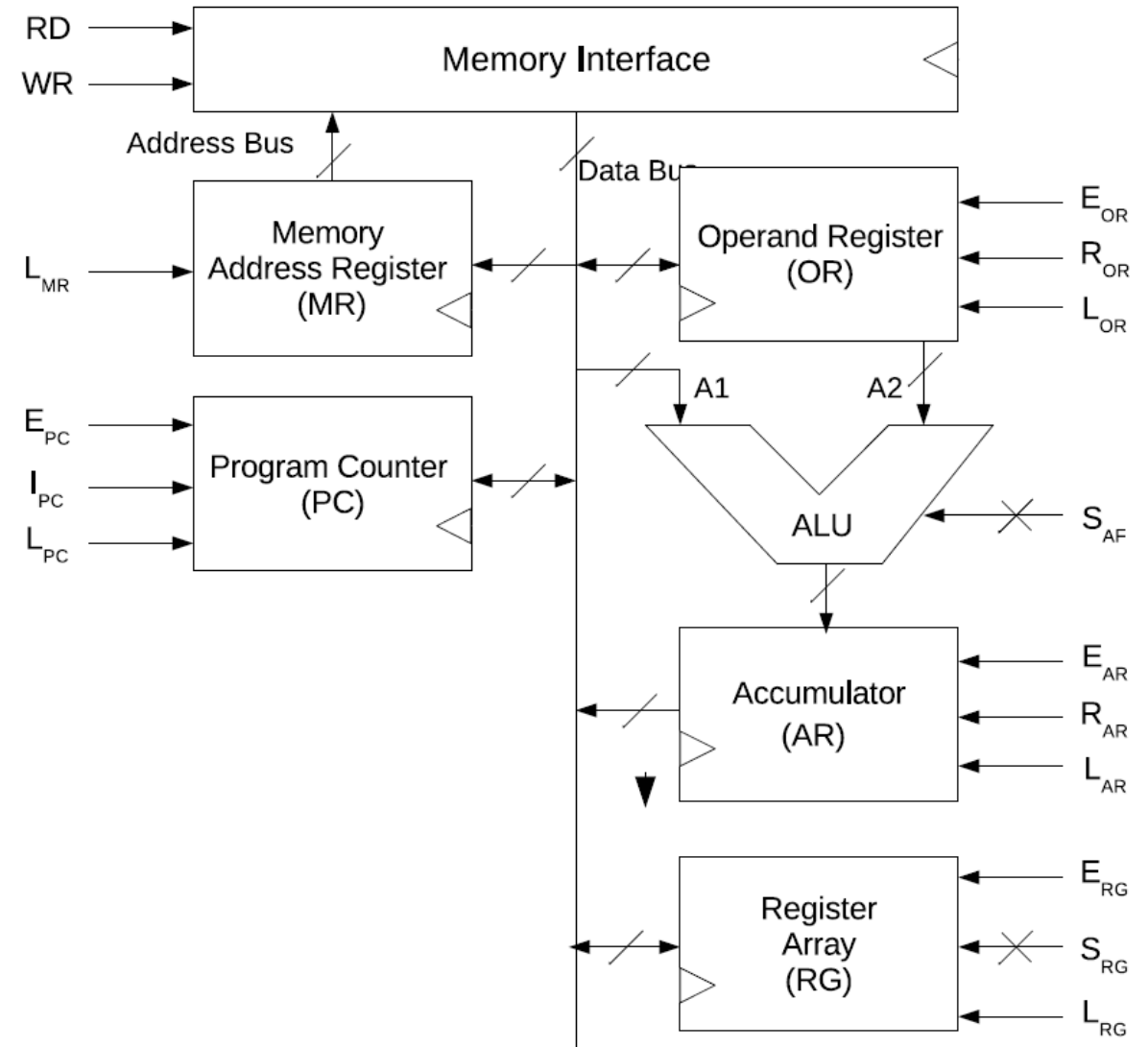
# Implementing instructions – data movement

- 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

