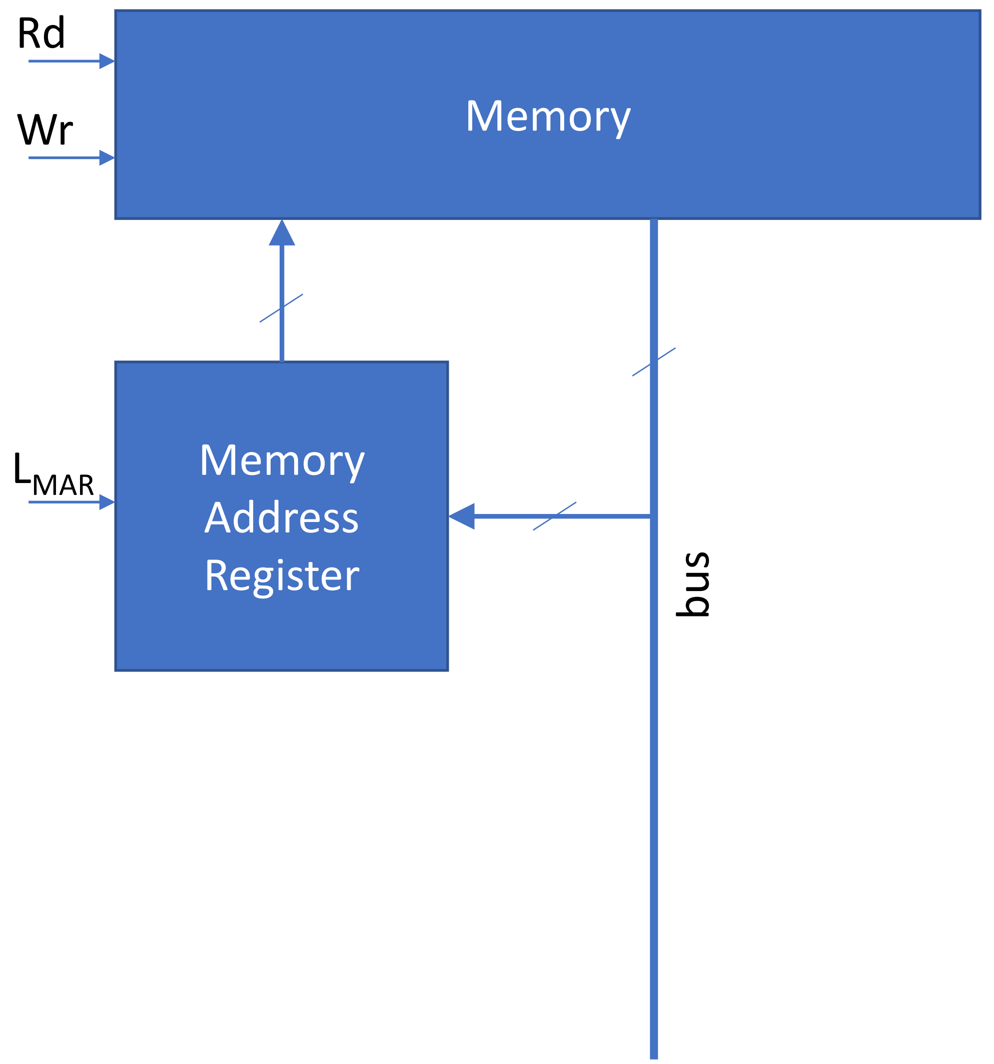
Lecture 30 – Processor design: Infinity War

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

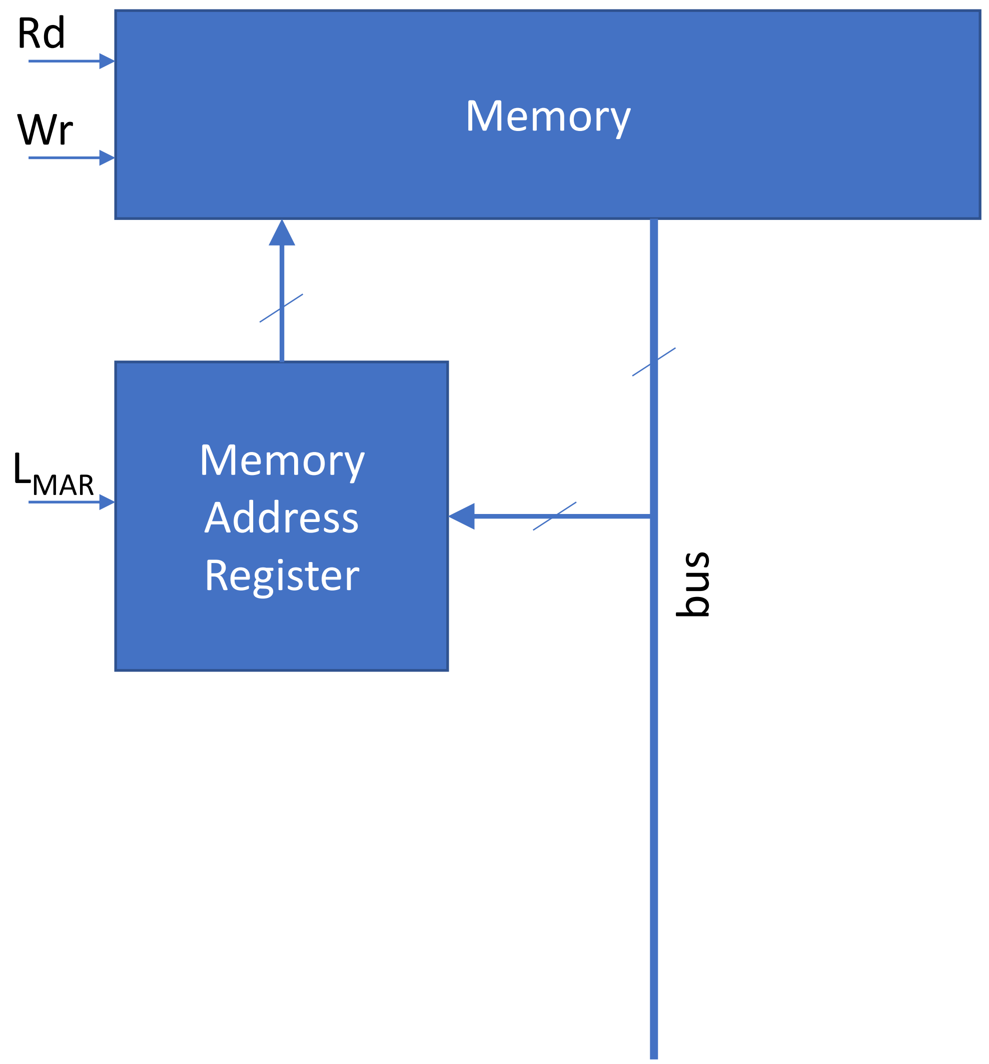
Memory access

- 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



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] $\leftarrow$ [<R>]
`movd <R>`	80-8F	[<R>] $\leftarrow$ [AR]
`movi <R> xx`	90-9F	[<R>] $\leftarrow$ xx
`stor <R>`	A0-AF	[[AR]] $\leftarrow$ [<R>]
`load <R>`	B0-BF	[<R>] $\leftarrow$ [[AR]]

The fetch-execute cycle

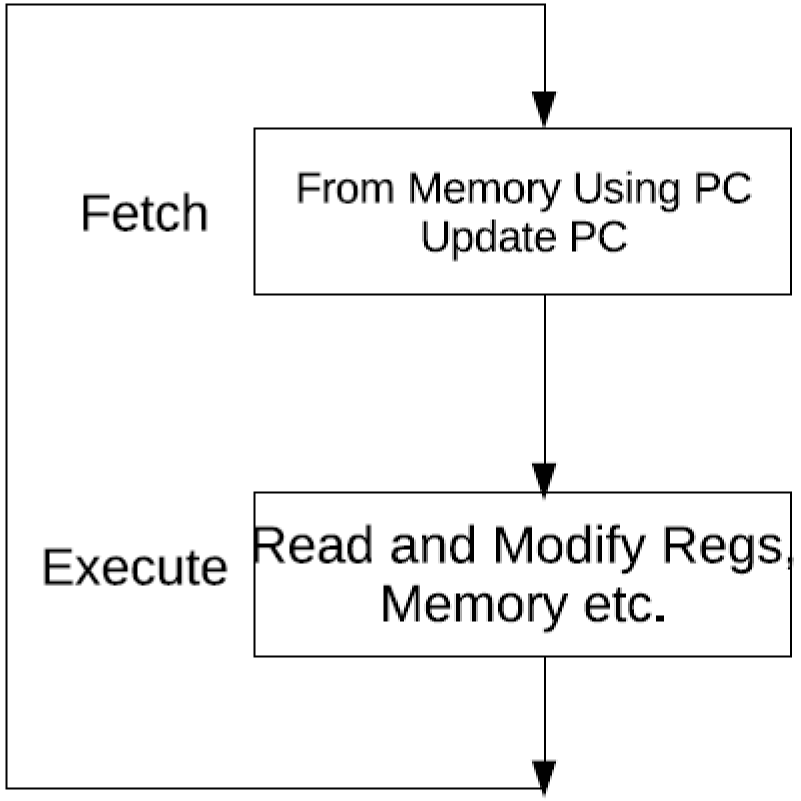
- We will look at the process of instruction fetching and execution
- The processor works autonomously as a continuous fetch-and-execute engine, with no other input than an external clock
- Since instructions as in the machine code are stored in memory, they have to be brought to the processor one by one and executed
- The instruction at address $(i + 1)$ has to be fetched and executed after instruction i, since the instructions of a program are stored consecutively in the memory
- The processor has to do all these by itself

The fetch-execute cycle

- Processors have a special register inside them that manages the process of instruction fetch by keeping track of the address of the next instruction to be fetched at all times
- This register is called the program counter or the PC
- The processing of an instruction begins with fetching its opcode from the memory word whose address is in the PC
- The contents of the PC are incremented while this happens to hold the address of the next instruction in the sequential order
- The opcode is brought to the processor and appropriate action is performed in the execution phase
- Once this is completed, the next instruction is processed by fetching it from the memory using PC as the address
- This goes on for ever inside the processor until a special STOP instruction is encountered
- Executing this instruction stops all activities of the processor

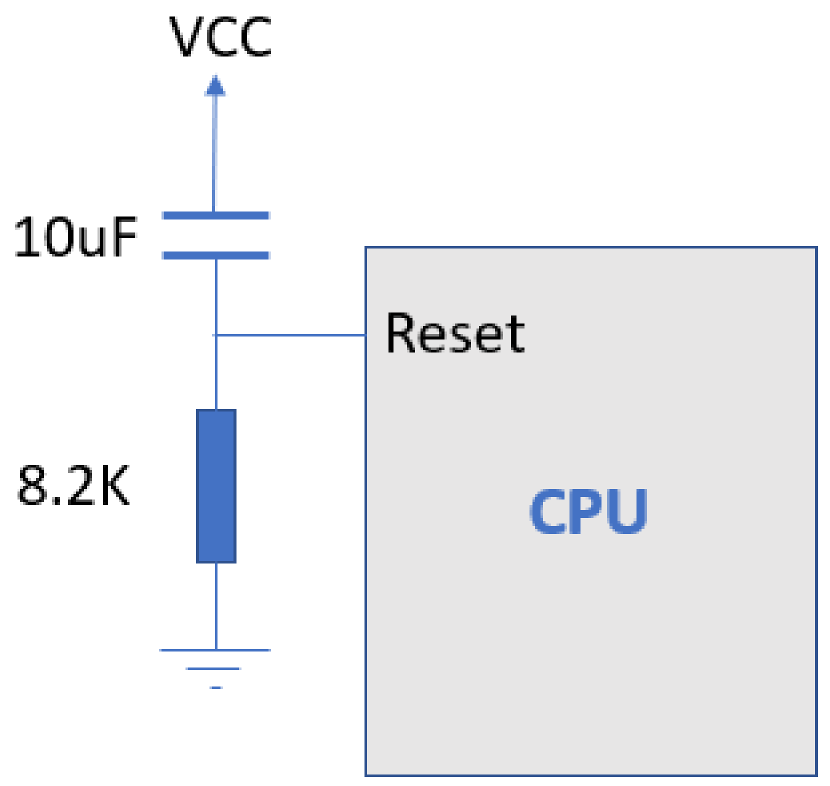
The fetch-execute cycle

- So how do we start the process?
- It is clear that once one instruction is done with, the next one is taken up by incrementing the PC
- Thus, once the execution of a program starts, everything goes on as the program indicates
- So how to start a program?
- A program can be started by loading the address of its first instruction into the PC
- However, how does the very first program start when the computer's power is turned on?



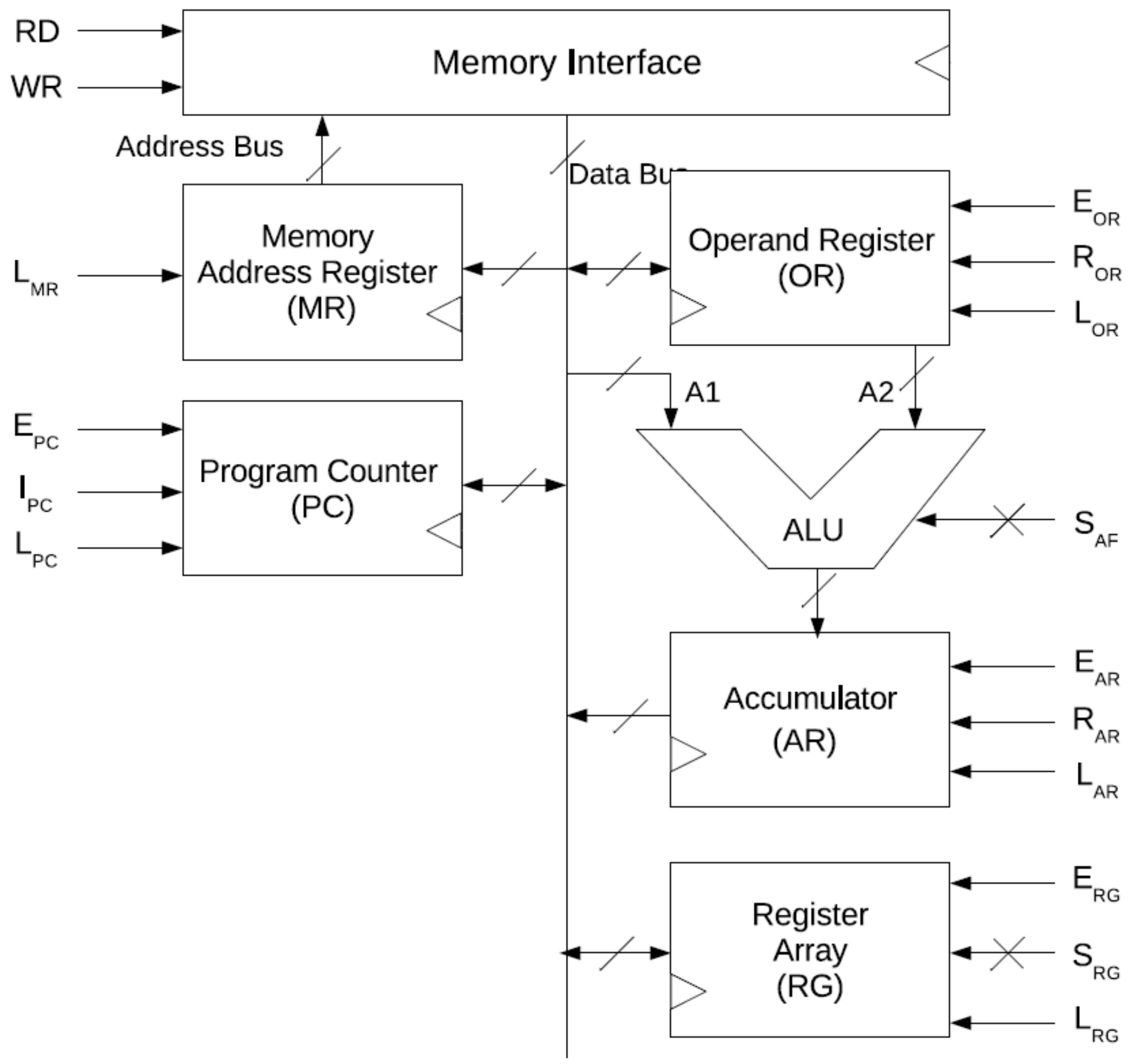
The fetch-execute cycle

- The processor hardware has a special feature to load a value of 0 to the PC when power is turned on or when the reset button of the computer is pressed (*literally* “resets” the “PC”)
- Thus, the very first program that gets control is the one that is saved at memory address 0
- Computer manufacturers place a special program at address 0 that has the BIOS program, which knows how to load the operating system from the boot record and proceed accordingly
- Any corruption in the BIOS can be very detrimental to booting the computer



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



Lecture 30b – Processor design: Guardians of the Instructions

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

Implementing instructions – ALU

- Consider the simple instruction:

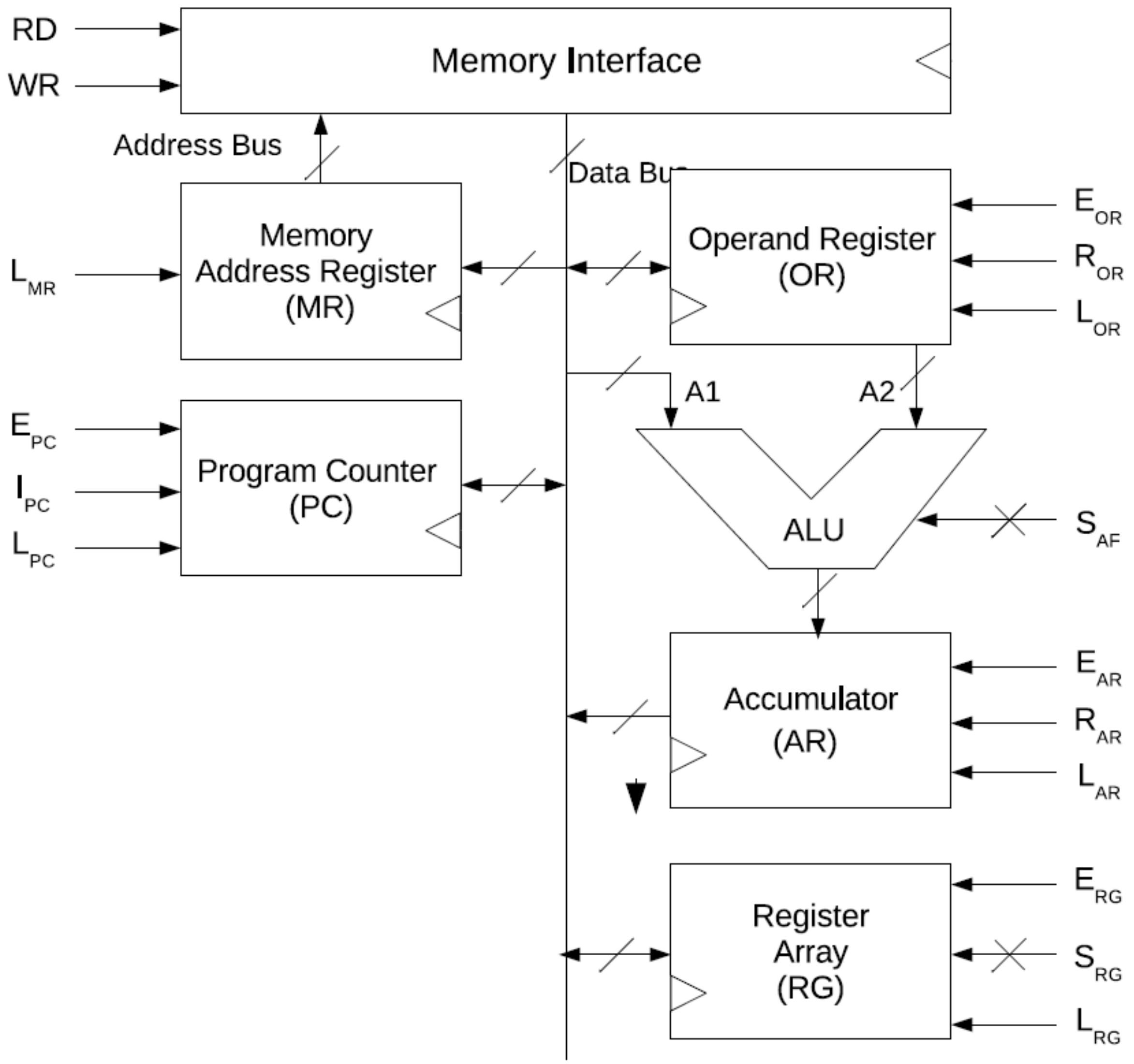
  ADD <R>

- This instruction can be executed in two clock cycles:

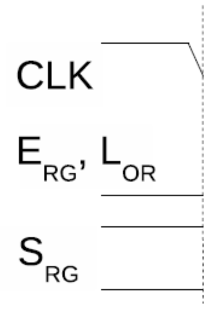
3. We need to enable RG and load the OR with the value [<R>] using the select lines for RG

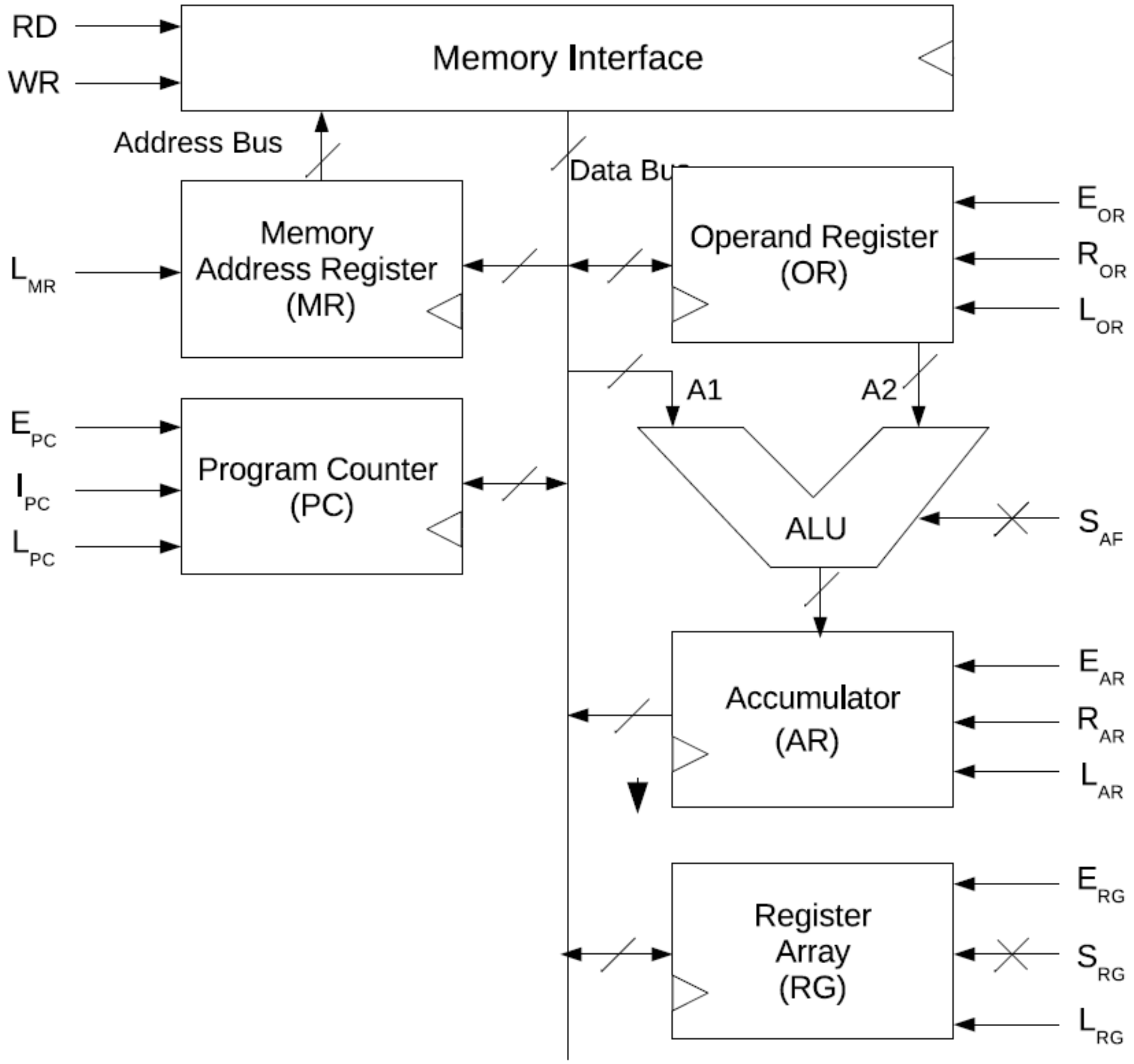
4. Enable AR, load the instruction for ADD in the ALU select lines, activate load AR

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



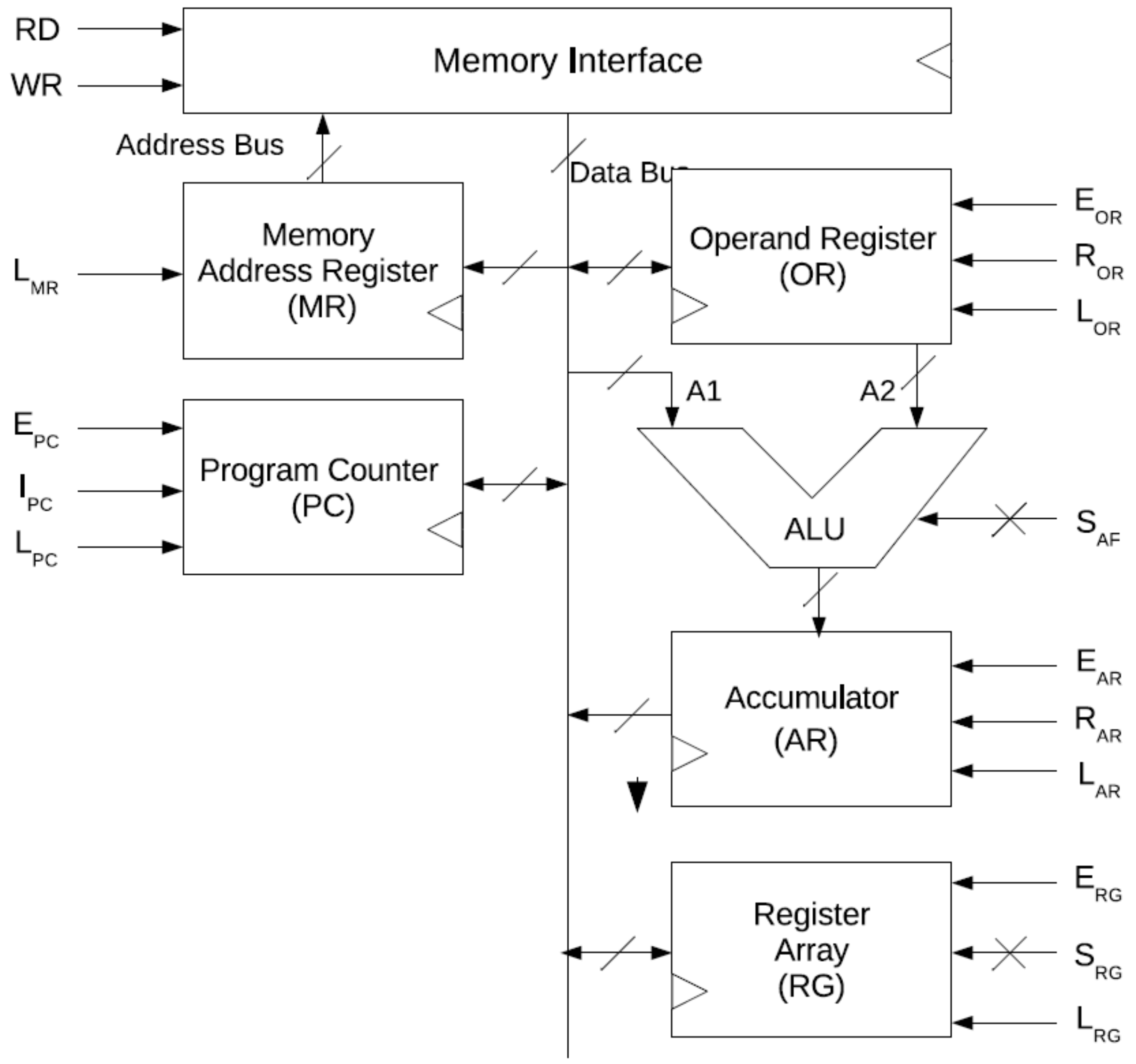
Implementing instructions – ALU





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	-

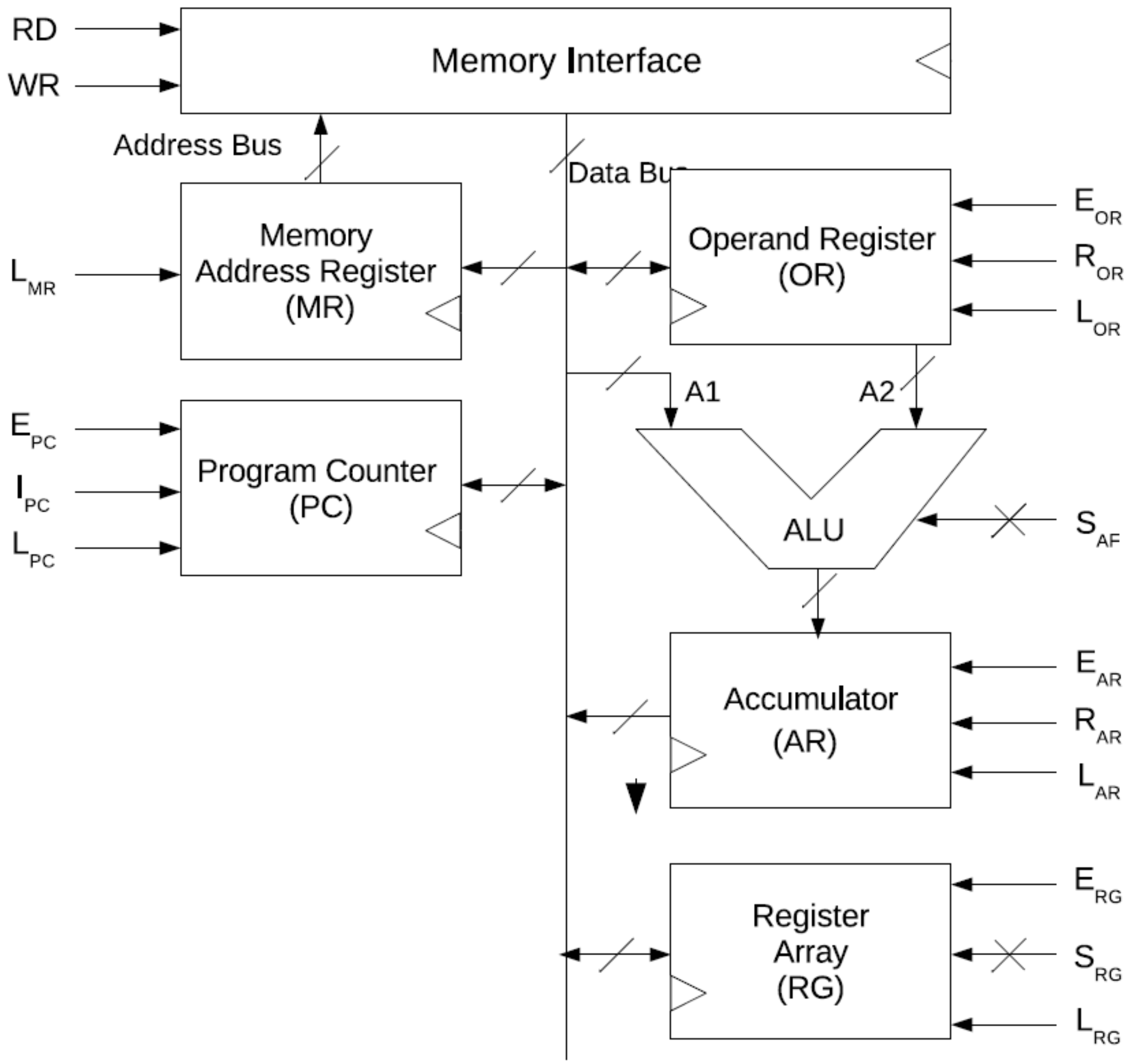


Implementing instructions

- 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*!


RD
WR
Memory Interface
Address Bus
Data Bus
L_{MR}
Memory Address Register (MR)
Operand Register (OR)
E_{OR}
R_{OR}
L_{OR}
A1
A2
E_{PC}
I_{PC}
L_{PC}
Program Counter (PC)
ALU
S_{AF}
Accumulator (AR)
E_{AR}
R_{AR}
L_{AR}
Register Array (RG)
E_{RG}
S_{RG}
L_{RG}