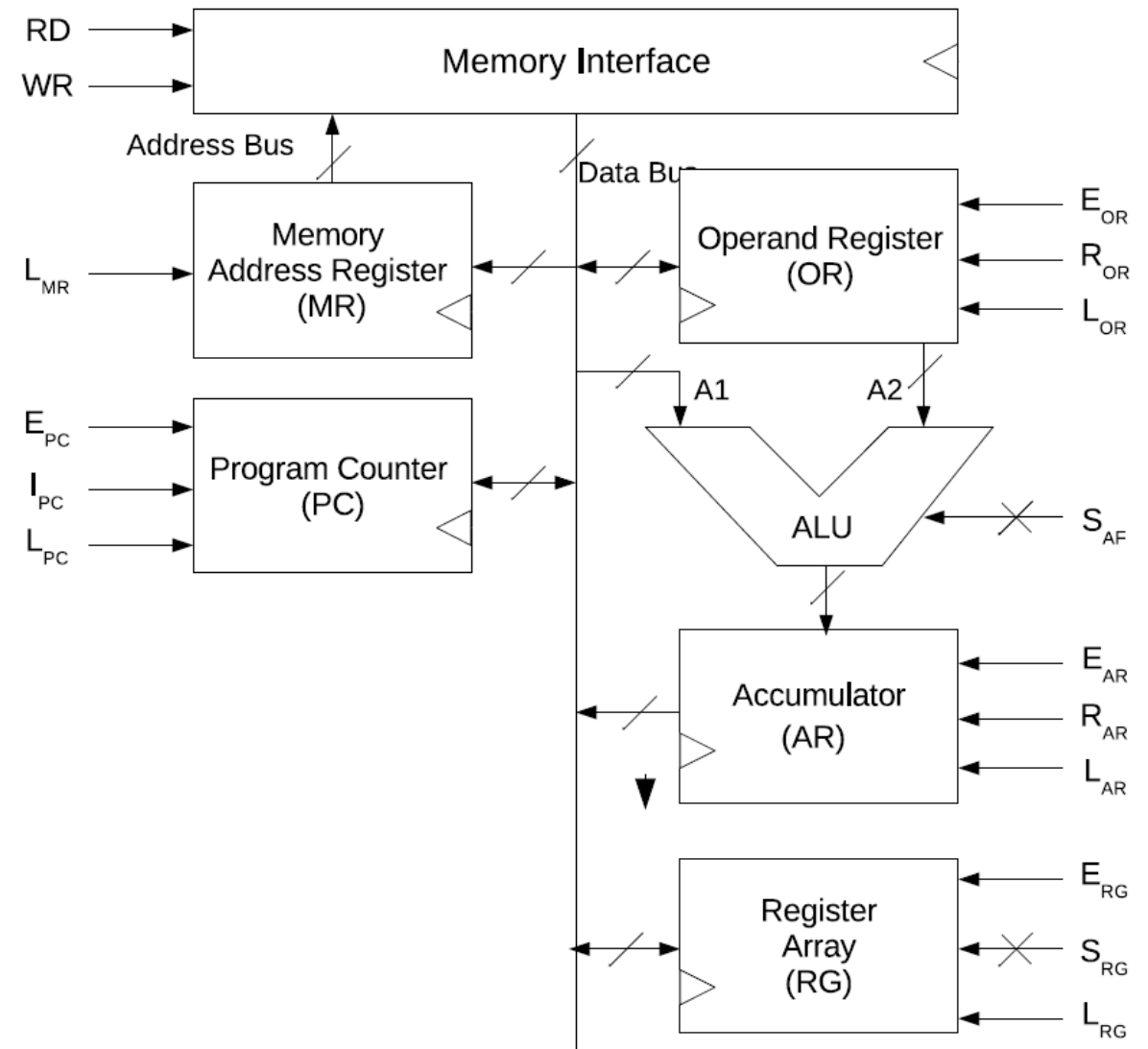


# Lecture 31 – Processor design: Homecoming

Dr. Aftab M. Hussain,  
Assistant Professor, PATRIOT Lab, CVEST

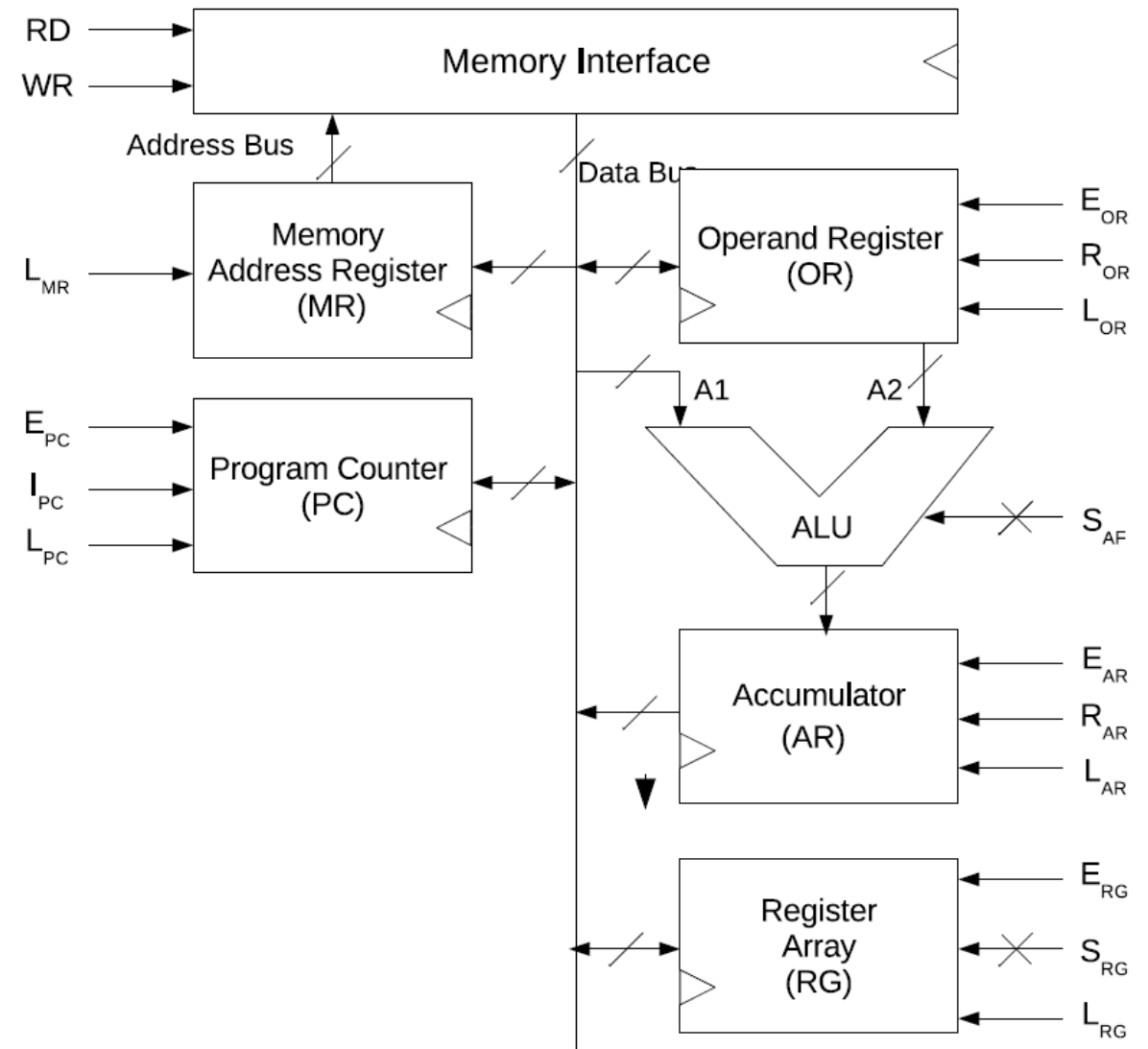
# Implementing instructions – data movement

- 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



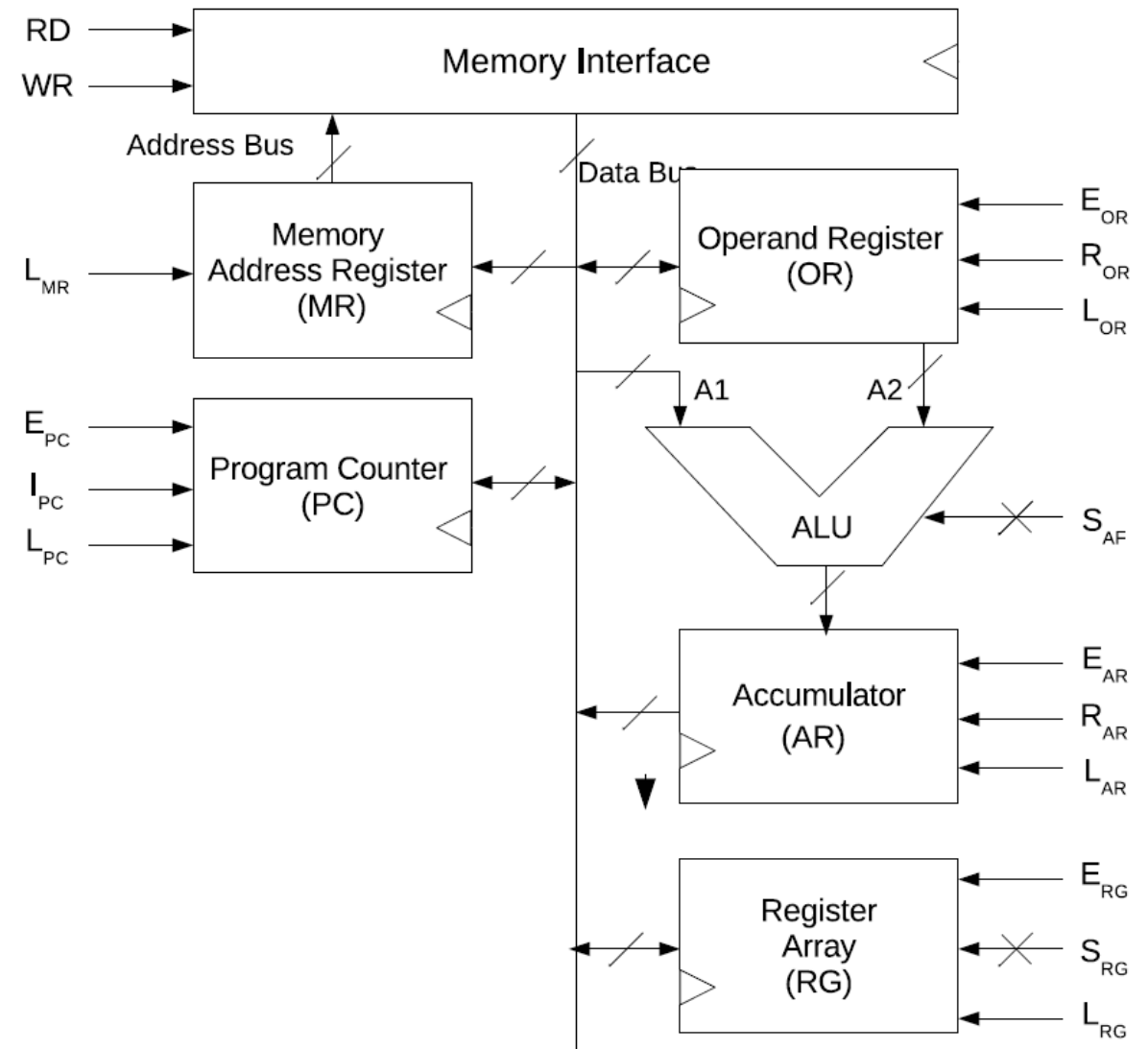
# Implementing instructions – data movement

- 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



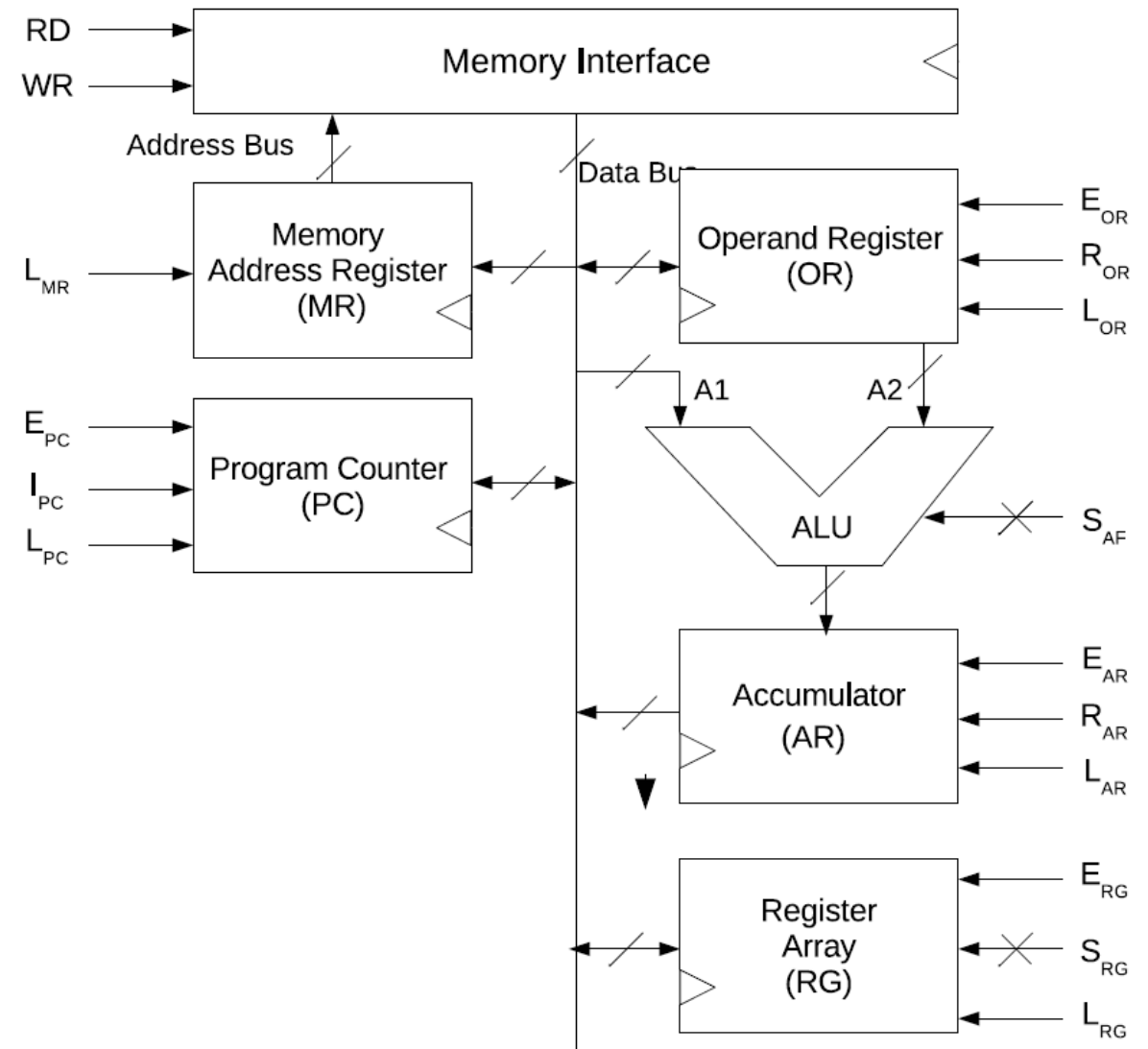
# Implementing instructions – data movement

- 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 (*addr* + 1) if the opcode for *movi* is stored in a memory location with address *addr*
- Moreover, we assume that as the opcode for *movi* is fetched from *addr*, the PC value is incremented by 1 to point to the next instruction



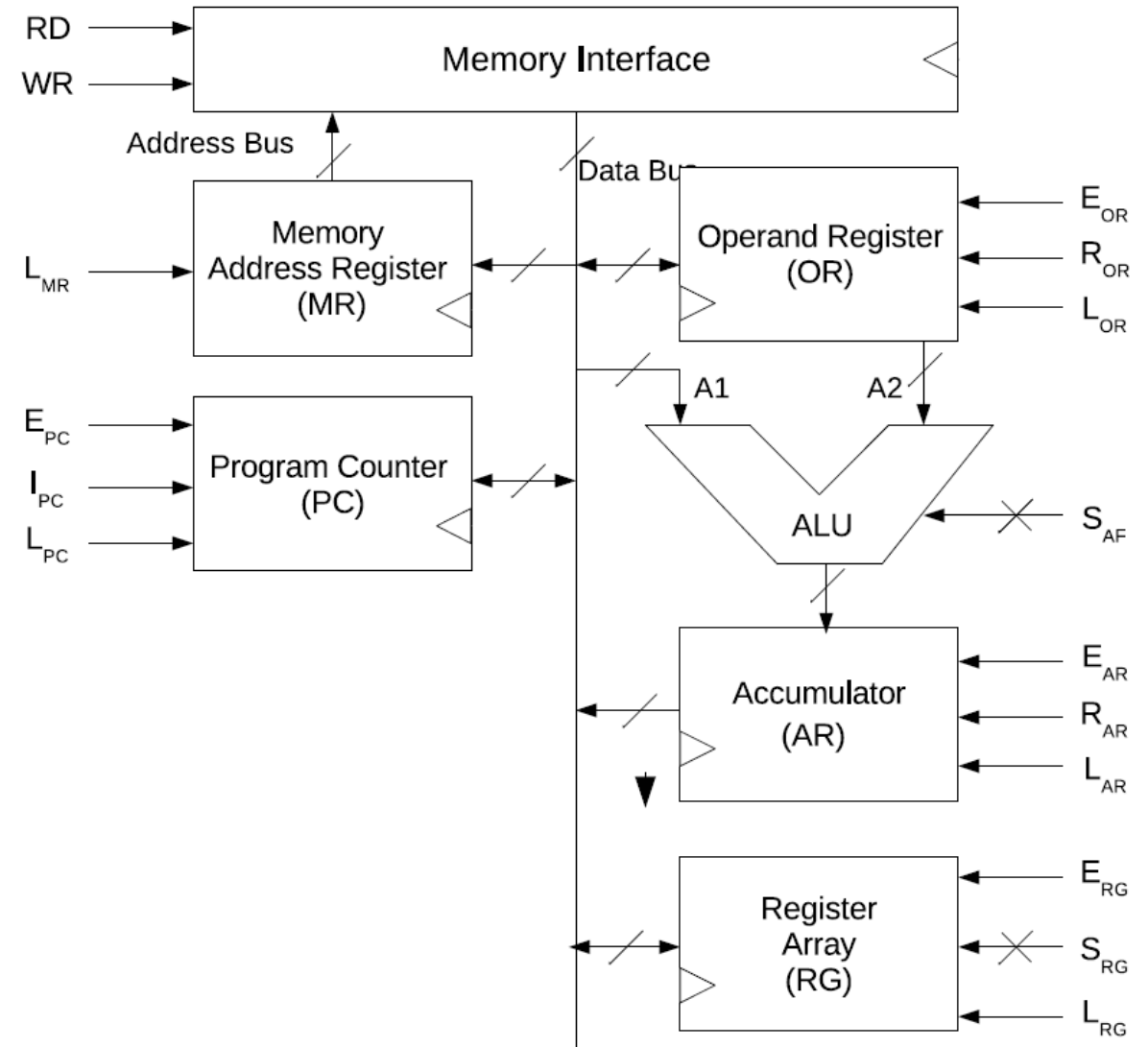
# Implementing instructions – data movement

- 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



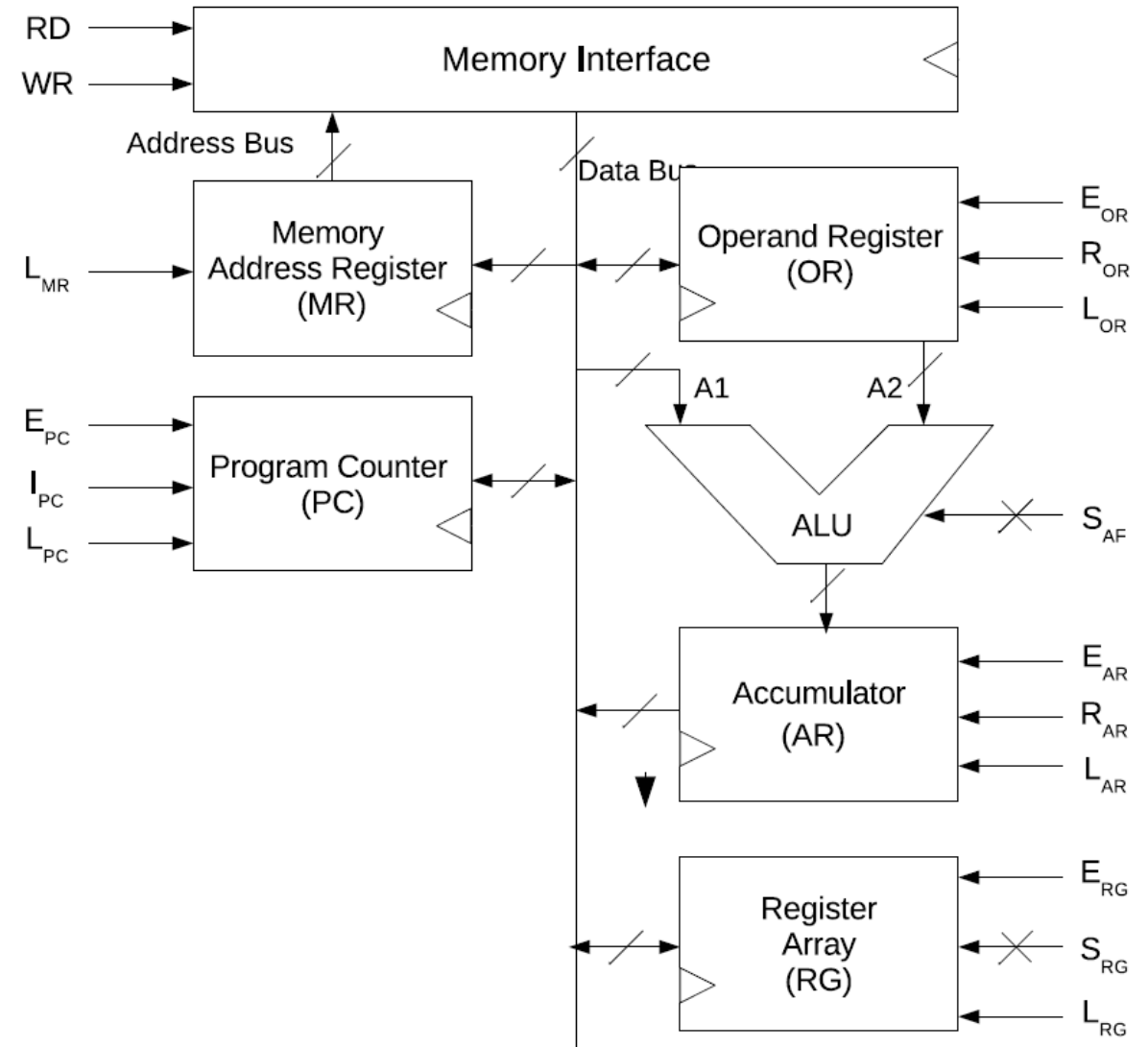
# Implementing instructions – ALU immediate

- 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 <sub>RG</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R>, S <sub>ALU</sub> ← PASS0
movd <R>	Ck 3: E <sub>AR</sub> , L <sub>RG</sub> , End	S <sub>RG</sub> ← <R>
load <R>	Ck 3: E <sub>AR</sub> , L <sub>MR</sub> Ck 4: RD, L <sub>RG</sub> , End	S <sub>RG</sub> ← <R>
stor <R>	Ck 3: E <sub>AR</sub> , L <sub>MR</sub> Ck 4: E <sub>RG</sub> , WR, End	S <sub>RG</sub> ← <R>
movi <R> xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>RG</sub> , End	S <sub>RG</sub> ← <R>
adi xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>ALU</sub> ← ADD
sbi xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>ALU</sub> ← SUB
xri xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>ALU</sub> ← XOR
ani xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>ALU</sub> ← AND
ori xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>ALU</sub> ← OR
cmi xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , End	S <sub>ALU</sub> ← CMP



# Jump instructions

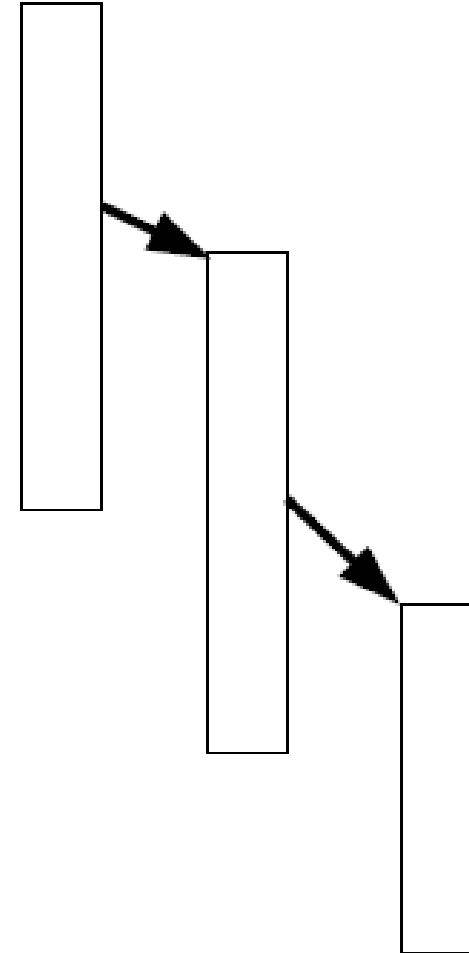
---

- We will now look at the final few instructions to make our processor complete
- We had no branching instructions so far; all programs have to be a strictly linear sequence of instructions
- That is obviously not a very desirable situation
- We need the capability to branch or to break the sequential flow of instructions to implement any sort of loops (for loops, while)
- Additionally, we need the capability to branch based on some condition based on the values of registers (if-else statements etc)
- We use two forms of branching: Conditional and unconditional



# Jump instructions

- Branching involves the shifting of the program execution from one point in the program to another
- This is a change in the control flow of the program and may be used to perform different actions on the basis of the results achieved so far
- Jump is a type of branching where the control is transferred absolutely, without any memory of the branching point
- Branching is natural in our every day activities



# The flag register

- The branching may be conditional, based on a current state of the processor
- For this, we include a flag register in the processor to indicate particular conditions
- The condition of one of the flag register bits can be used for branching
- If the jump is conditioned on a flag bit being set, the branching happens only if that flag has a value of 1 and the program proceeds with the instruction at the branch address
- If the flag is at state 0, execution proceeds normally with the next instruction
- The flag register may include aspects like: Did the last arithmetic operation result in an overflow? Did it result in a carry from the most significant bit? Was the result of the previous operation a zero?
- These, in conjunction with branching, are essential to control the program based on the results of operations

# The flag register

---

- For example, if we want to run a loop ten times, we can repeat the linear code 10 times, which makes the code long
- It also allows no flexibility to run the code 12 times, if we desire it
- An alternative is to use a count (typically stored in a register) that is initialized to 10
- After one set of computations is over, the count is reduced by 1
- The program can branch to the start of the computations if the count is still not zero
- It is clear that the second option results in shorter code
- Even better, if the count is initialized to 12 or 25, the code remains exactly the same

# The flag register

---

- For our simple processor we chose the following 4 flag bits: zero, carry, sign, and parity, with respective flags Z, C, S, and P
- The zero flag is set if the previous ALU operation produced an exact 0 as the result (i.e., if AR is zero)
- Similarly, the carry flag is set if the previous operation resulted in a carry-out or borrow-in from the most significant bit
- The S bit copies the sign bit of the last arithmetic operation and becomes 1 if the result was negative
- The parity bit counts the number of 1 bits in the result of the last operation (AR)

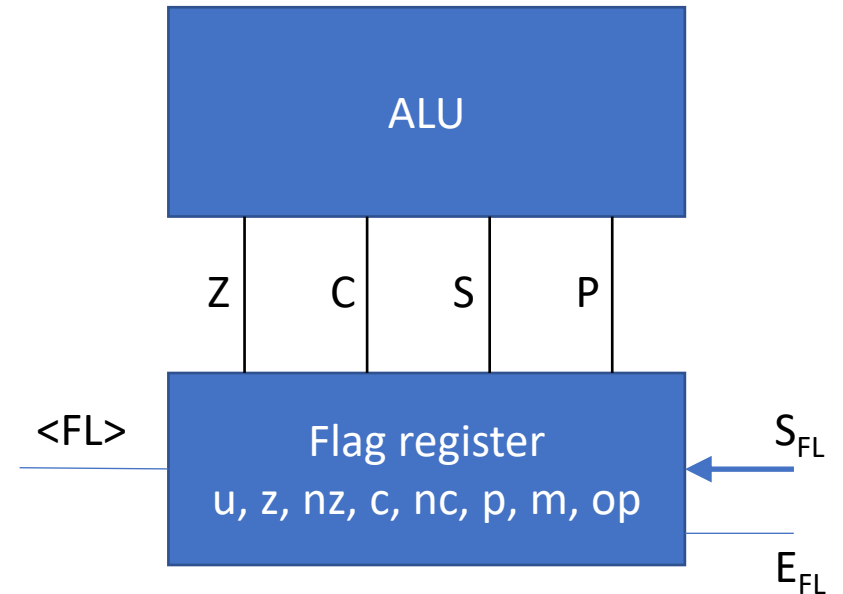
# The flag register

---

- Instructions that do not use the ALU – such as the data movement instructions, branching instructions, and the like – do not change the flag values
- Some processors group all flags into a special register word known as the Program Status Word (PSW)
- Special instructions may move the PSW to or from internal registers or memory
- This allows their manipulation as data

# Jump instruction

- For the simple flag register defined, the <FL> flag for conditional instructions can take one of the following values: u, z, nz, c, nc, p, m, op
- These respectively stand for unconditional, zero, non-zero, carry, no-carry, positive, minus, and odd-parity
- Unconditional case is always true, irrespective of the state of the flag bits
- Zero condition is true when the Z bit is set and the non-zero condition is true otherwise
- Similarly, the carry and no-carry conditions are true when the C bit of the flags is 1 and 0 respectively
- The plus condition is true when the sign bit S is 0 and the minus condition is true otherwise
- The odd-parity condition is true if the flag bit P is 1



# Implementing the jump instruction

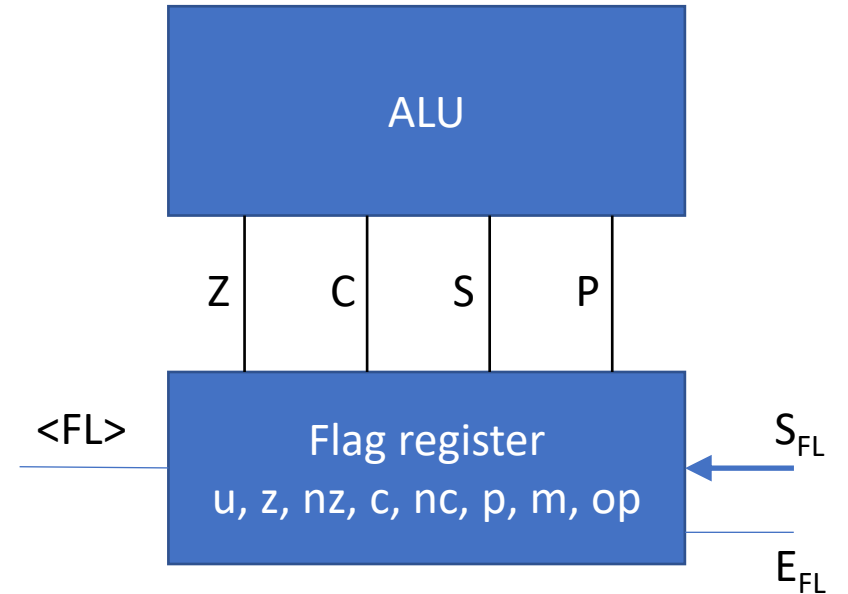
- Lets define two jump instructions:

`jmpd<FL> xx`

- This instruction makes the program jump to address XX if <FL> is true
- 8 different cases are there

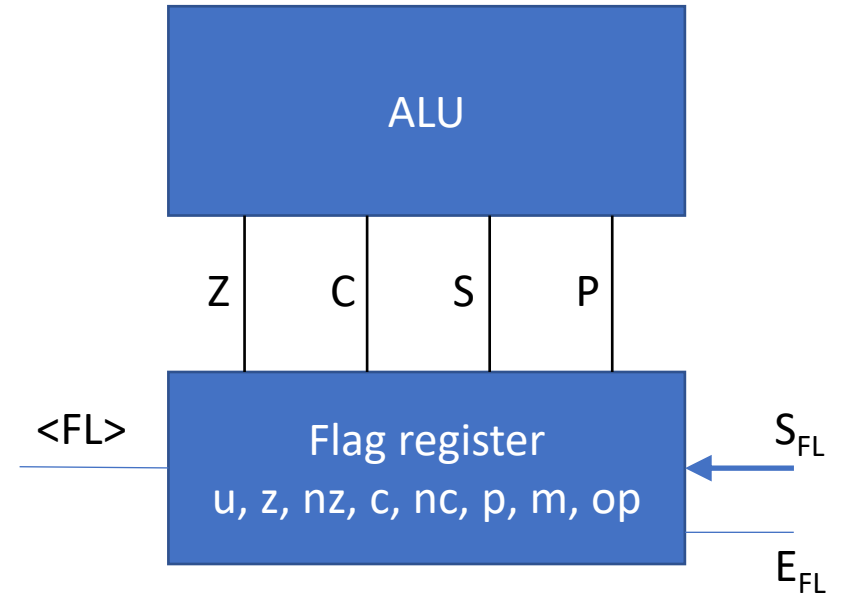
`jmpr<FL>`

- This instruction makes the program jump to address [AR] if <FL> is true
- 8 different cases are there



# Implementing the jump instruction

- To implement the jump function, we need to change the PC to the contents of the AR if the flag condition is satisfied
- If not, it should have no effect
- The conditional jump instructions use a modified control signal, labelled “*End if <FL>*”
- This means that the End control signal is activated only if the selected flag is at a 0 level
- This is the case where the condition is not satisfied and hence the instruction has no impact

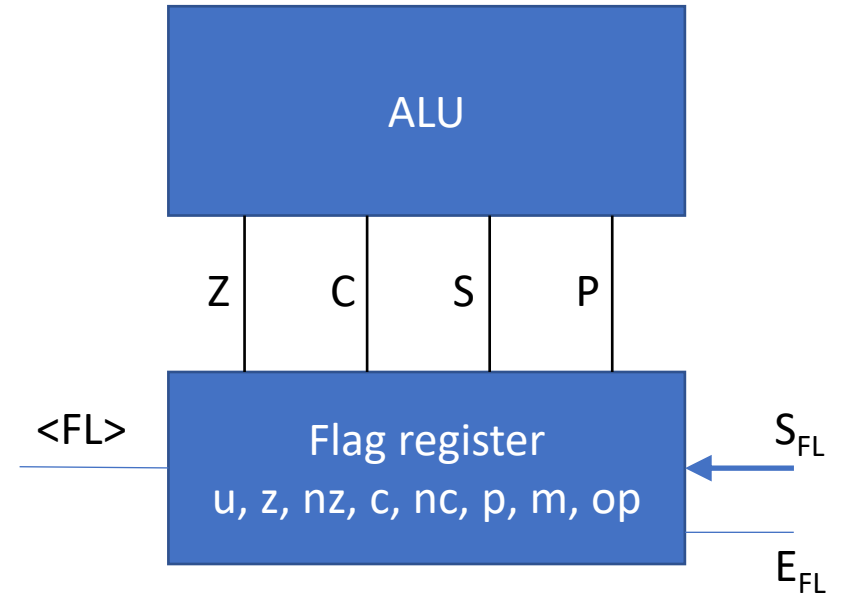


Assembly Instruction	Machine Code	Action
jmpd<FL> xx	E0-E7	[PC] $\leftarrow$ xx if <FL> = 1
jmpr<FL>	E8-EF	[PC] $\leftarrow$ [AR] if <FL> = 1



# Implementing the jump instruction

- However, for instructions with immediate operands (such as `jumpd`), the next word has to be jumped over so that the PC points to the next real instruction
- At the same time, the value of PC should be incremented whether the value of flag is true or not
- Hence, the PC value is saved in MR and PC is incremented
- Then if the flag condition is satisfied, the value at the address is loaded into PC



Assembly Instruction	Machine Code	Action
<code>jumpd&lt;FL&gt; xx</code>	E0-E7	$[PC] \leftarrow xx$ if $\langle FL \rangle = 1$
<code>jmprr&lt;FL&gt;</code>	E8-EF	$[PC] \leftarrow [AR]$ if $\langle FL \rangle = 1$

Instruction	Control Signals	Select Signals
<code>jumpd&lt;FL&gt; xx</code>	Ck 3: $E_{PC}, L_{MR}, I_{PC}, E_{FL}$ , End if $\langle FL \rangle$ , Ck 4: $RD, L_{PC}$ , End	$S_{FL} \leftarrow \langle FL \rangle$ -
<code>jmprr&lt;FL&gt;</code>	Ck 3: $E_{FL}$ , End if $\langle FL \rangle$ , Ck 4: $E_{AR}, L_{PC}$ , End	$S_{FL} \leftarrow \langle FL \rangle$ -