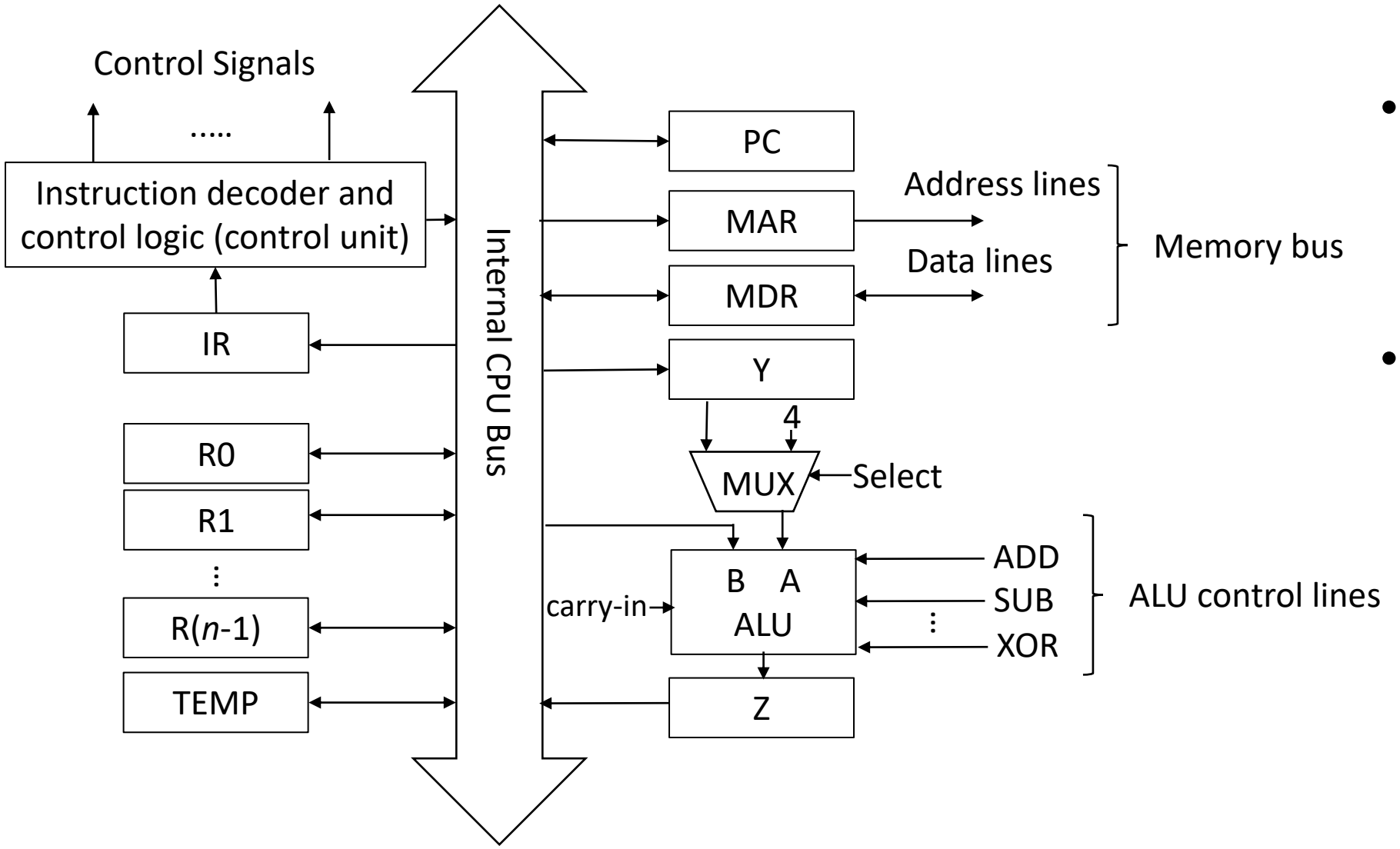
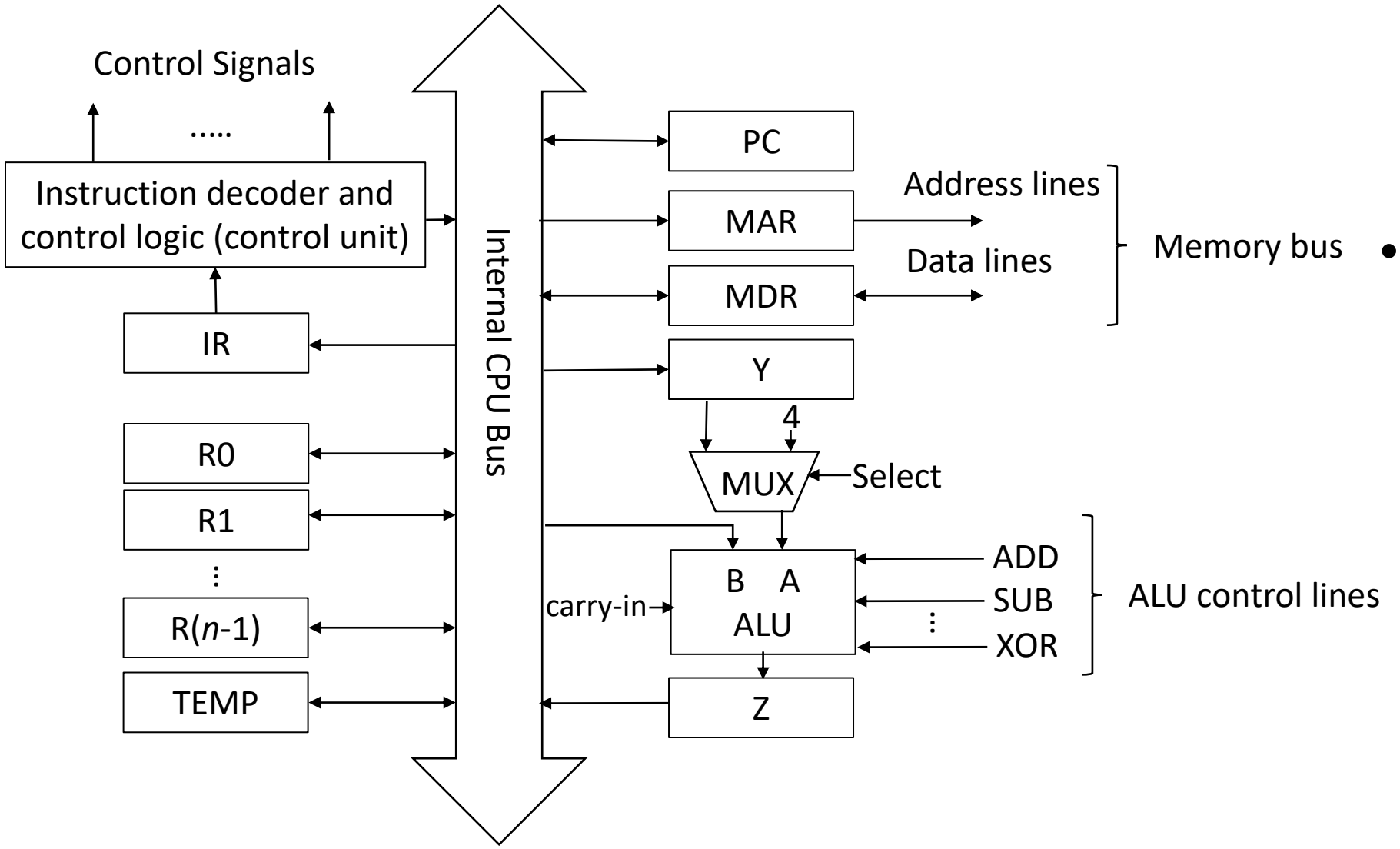
Single-Bus CPU Architecture



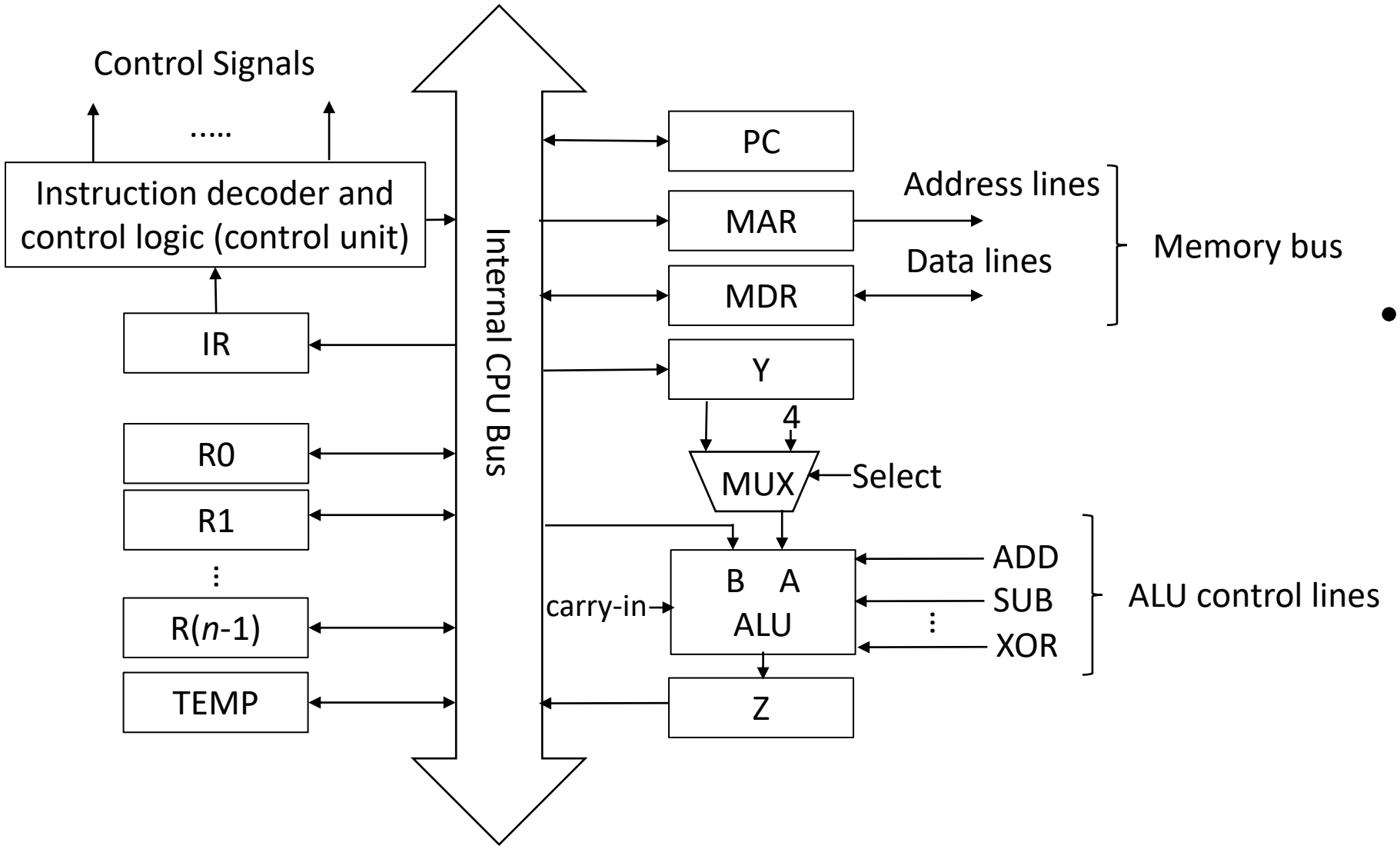
- The *Program Counter* (*PC*) stores the address of the next instruction to execute.
- The *Instruction Register* (*IR*) stores the instruction currently being executed.
- The *Control Unit* (*CU*) generates the control signals needed to direct the operation of the different components of the CPU.

Single-Bus CPU Architecture



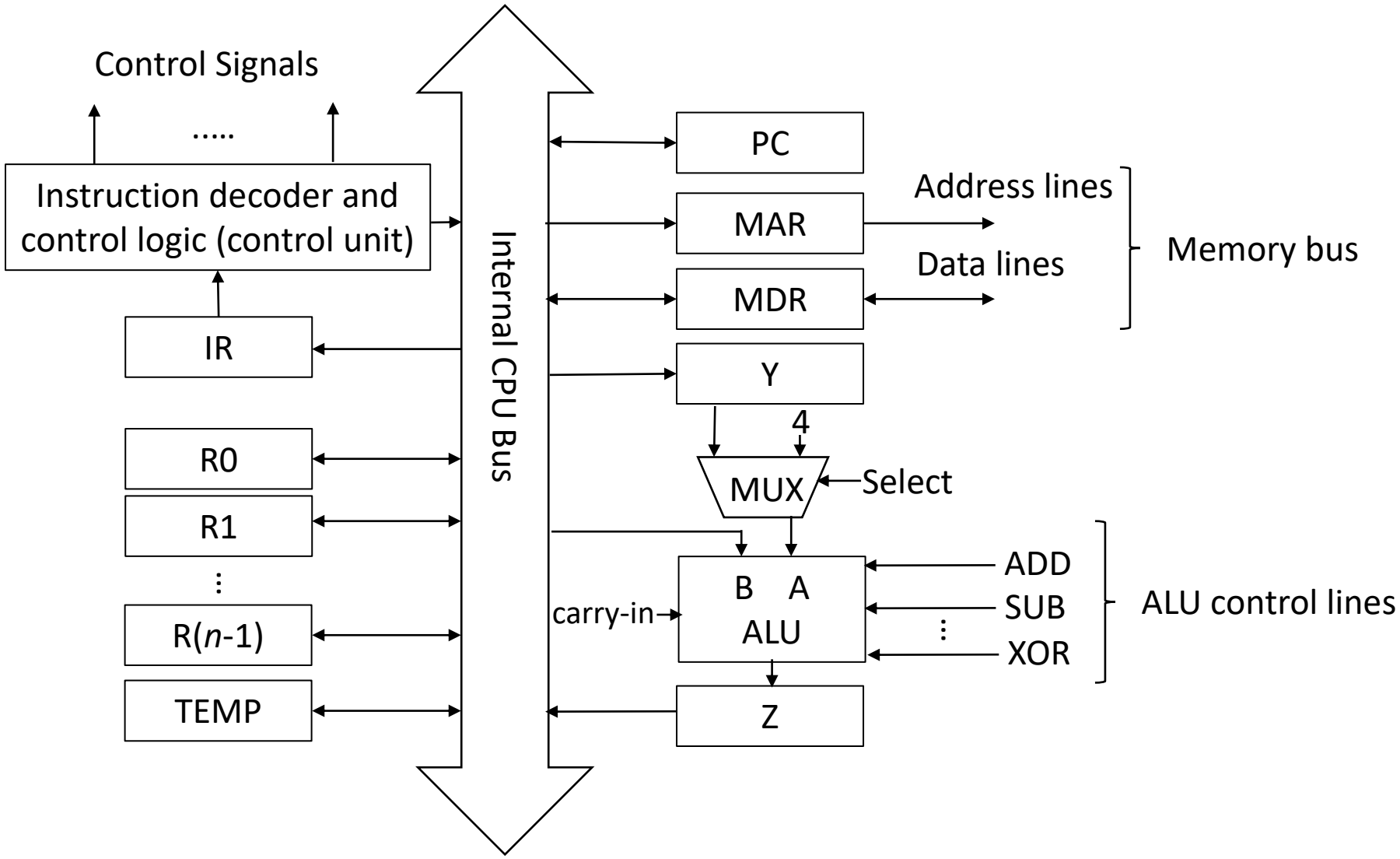
- The *Memory Address Register* (*MAR*) holds the address of the memory location to be accessed.
- The *Memory Data Register* (*MDR*) holds the data to be stored or retrieved from memory.

Single-Bus CPU Architecture



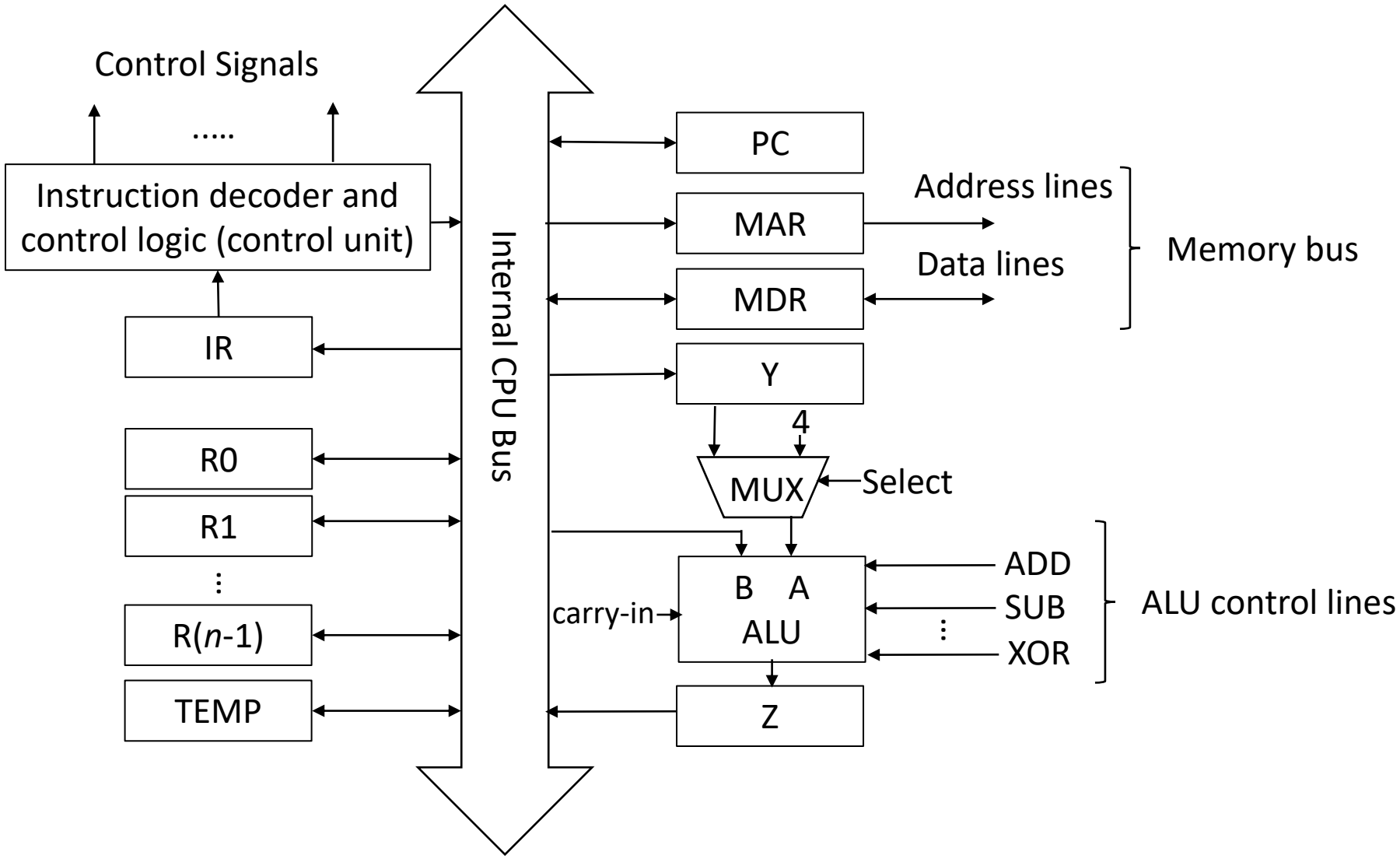
- CPU registers *R0* to *R*(*n*–1) are "general-purpose" registers for data storage. In x86, these are EAX, EBX, ECX, EDX, etc. while for RISC these are x1, x2, etc.
- *TEMP* register is transparent to the programmer and used by the processor for temporary storage during execution of some instruction

Single-Bus CPU Architecture



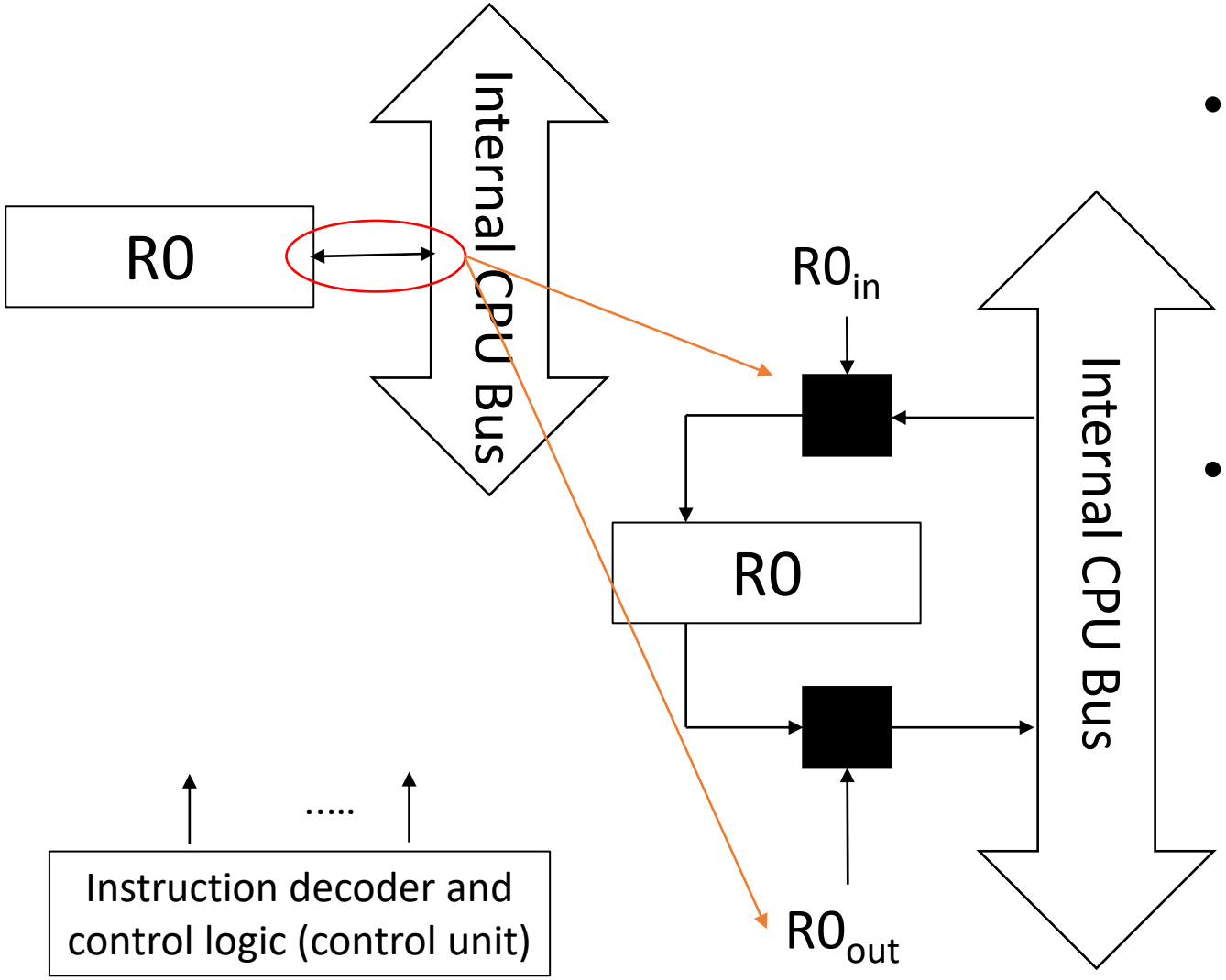
- The *Arithmetic and Logic Unit* (*ALU*) performs specified operations on the data.
- The *Y* register is used to hold a second operand for binary ALU operations.
- The *Z* register is used to hold the result of an ALU operation.

Single-Bus CPU Architecture



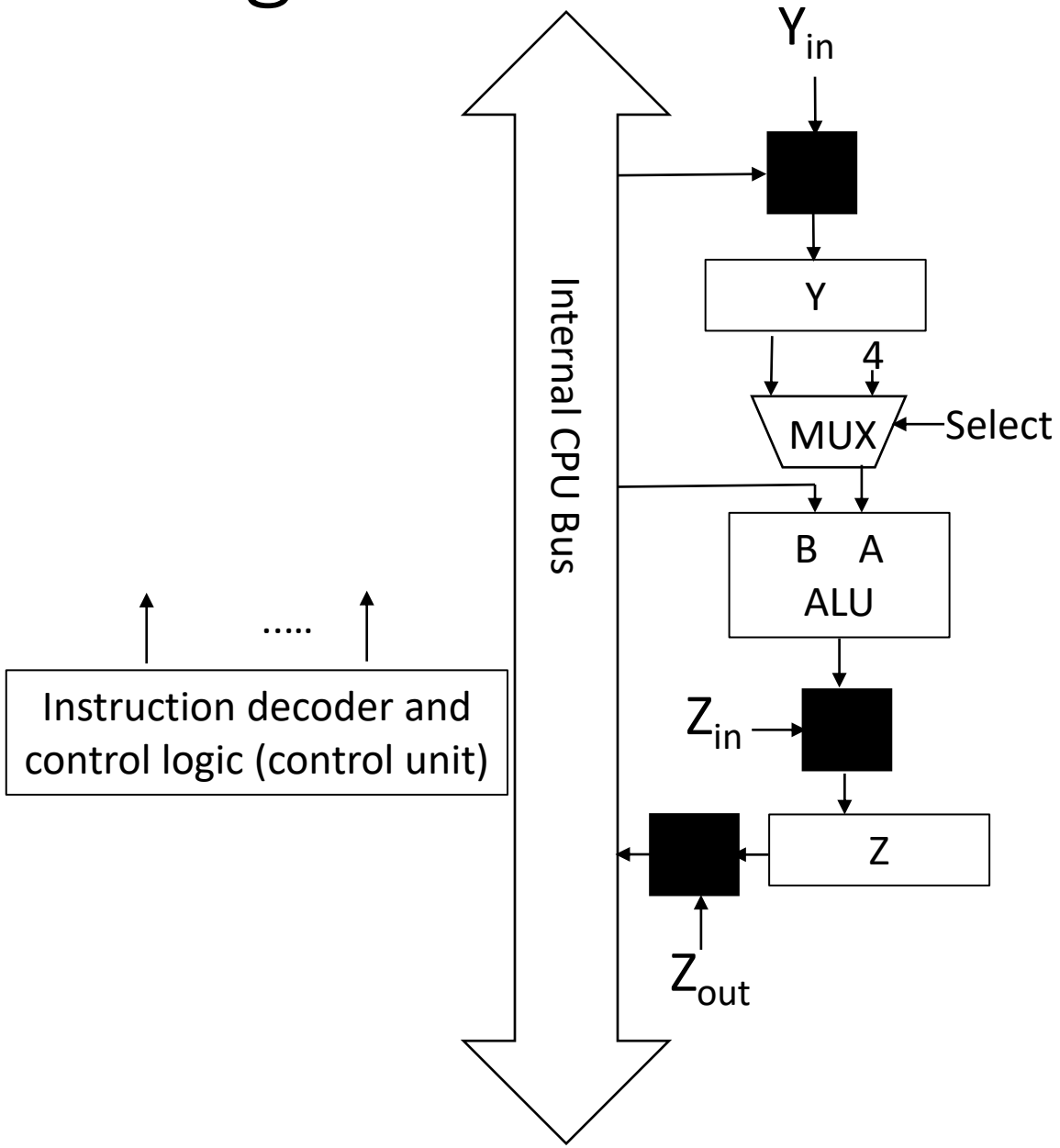
- The multiplexer *MUX* selects either the output of register *Y* or a constant value 4 as input *A* of the ALU. The constant 4 is used to increment the contents of the program counter.

Gating Considerations & Signals (register)



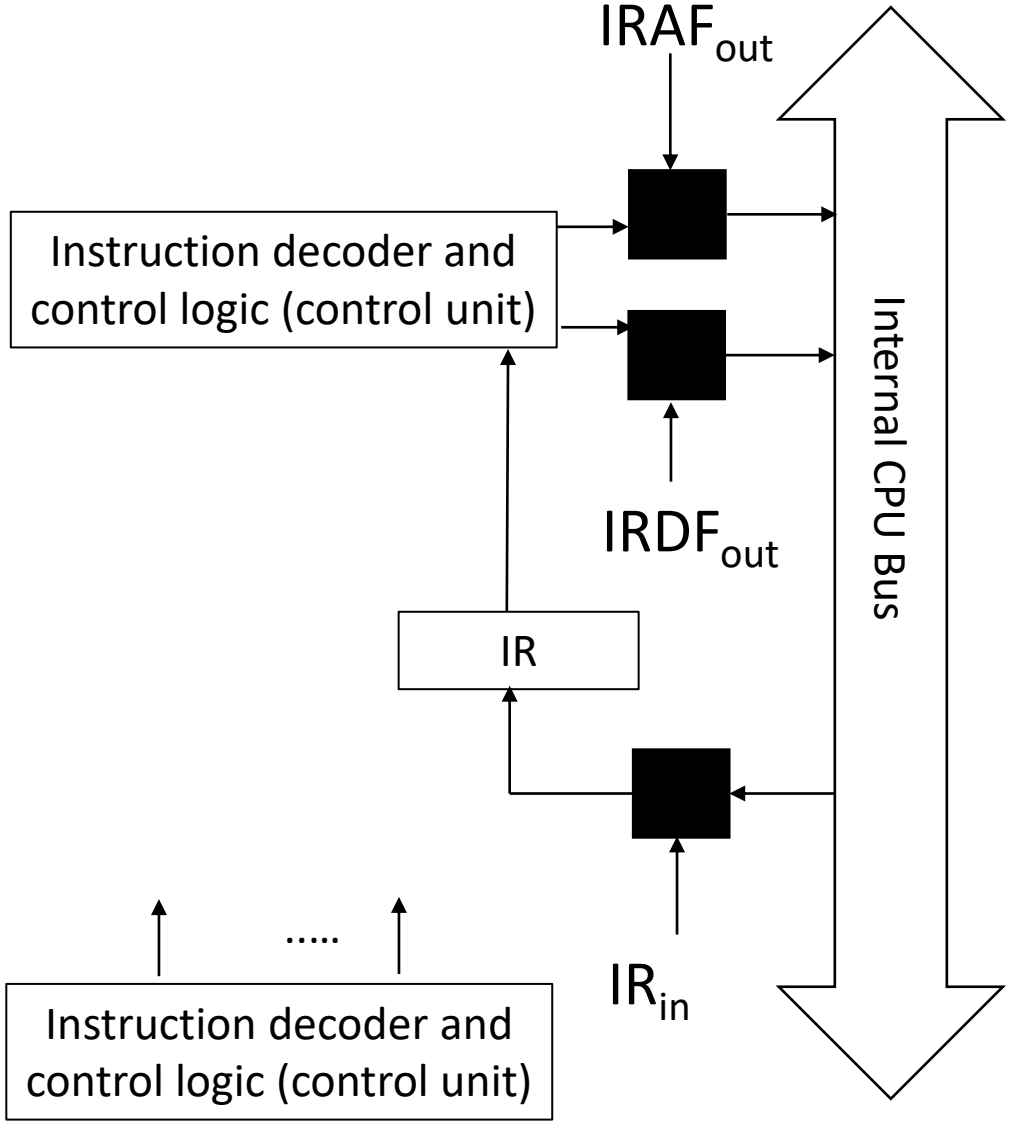
- Data flow within the CPU is controlled by gating signals.
  - Bus access requires that only one component may output to the bus at any instant. It is possible to have more than one "listeners" to the bus.
- Gating signals shall be identified using the register name with subscripts denoting the direction of data flow.
  - An "in" subscript indicates that the input data will be copied onto the register.
  - An "out" subscript indicates that the content of the register will be copied onto the bus.

Gating Considerations & Signals (ALU)



- Data flow within the CPU is controlled by gating signals.
  - Bus access requires that only one component may output to the bus at any instant. It is possible to have more than one "listeners" to the bus.
- The content of Y and the bus are always visible to the ALU.
- The output of the ALU is gated to the Z register.
- Two possible values of the MUX Select control is Select4 and SelectY for selecting constant 4 or register Y respectively

Gating Considerations & Signals (IR and CU)



- Data flow within the CPU is controlled by gating signals.
  - Bus access requires that only one component may output to the bus at any instant. It is possible to have more than one "listeners" to the bus.
- The content of IR is always visible to the control unit
- In some instances, data is included in the instruction passed to the IR. This data is either used as actual data for an operation or an address for a memory access. Hence, the instruction decoder has two output lines, IR_address_field$_{out}$ (IRAF$_{out}$) and IR_data_field$_{out}$ (IRDF$_{out}$).

Basic Operations

- All CPU instructions, with few exceptions, can be decomposed into a combination of 4 basic operations.
  - Register Transfer
  - ALU Operation
  - Memory Fetch
  - Memory Store

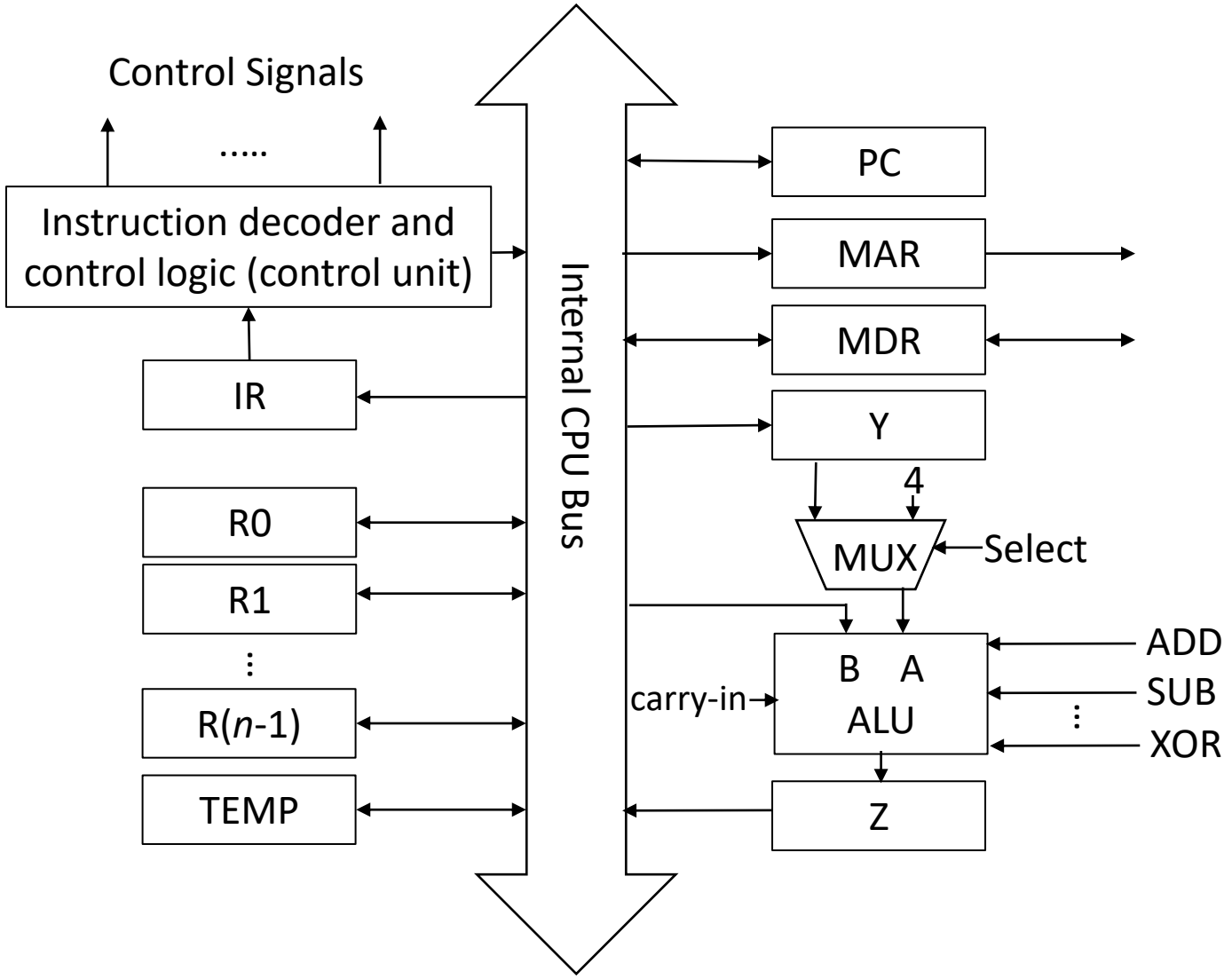
Register Transfer

- This involves the copying of data from one register (*source*) to another register (*destination*).
- In terms of gating signals, this operation is performed via
  - $source_{out}$, $destination_{in}$
- where the *source* outputs the data onto the bus while the *destination* reads the bus.

Register Transfer

Example: MOV EAX, EBX

- EBX_{out}, EAX_{in}



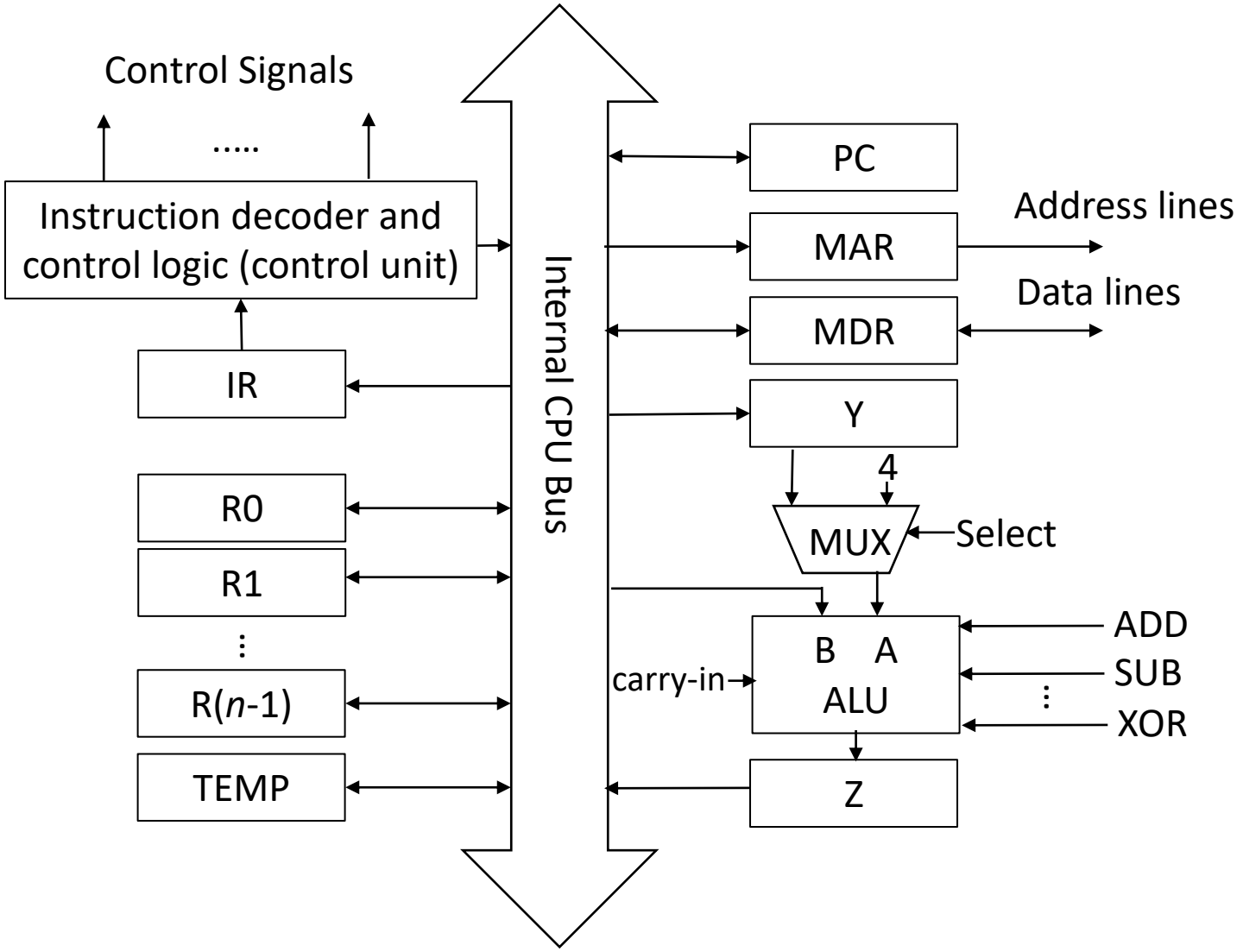
ALU Operation

- This involves operation of the ALU on data.
- For a binary operation, one operand needs to be placed on the Y register first, *i.e.*,
  - *source1*$_{out}$, Y$_{in}$
  - source2$_{out}$, Select Y, ALU_command, Z$_{in}$
  - Z$_{out}$, destination$_{in}$

ALU Operation

Example: ADD EAX, EBX

- EAX_{out}, Y_{in}
- EBX_{out}, SelectY, ADD, Z_{in}
- Z_{out}, EAX_{in}



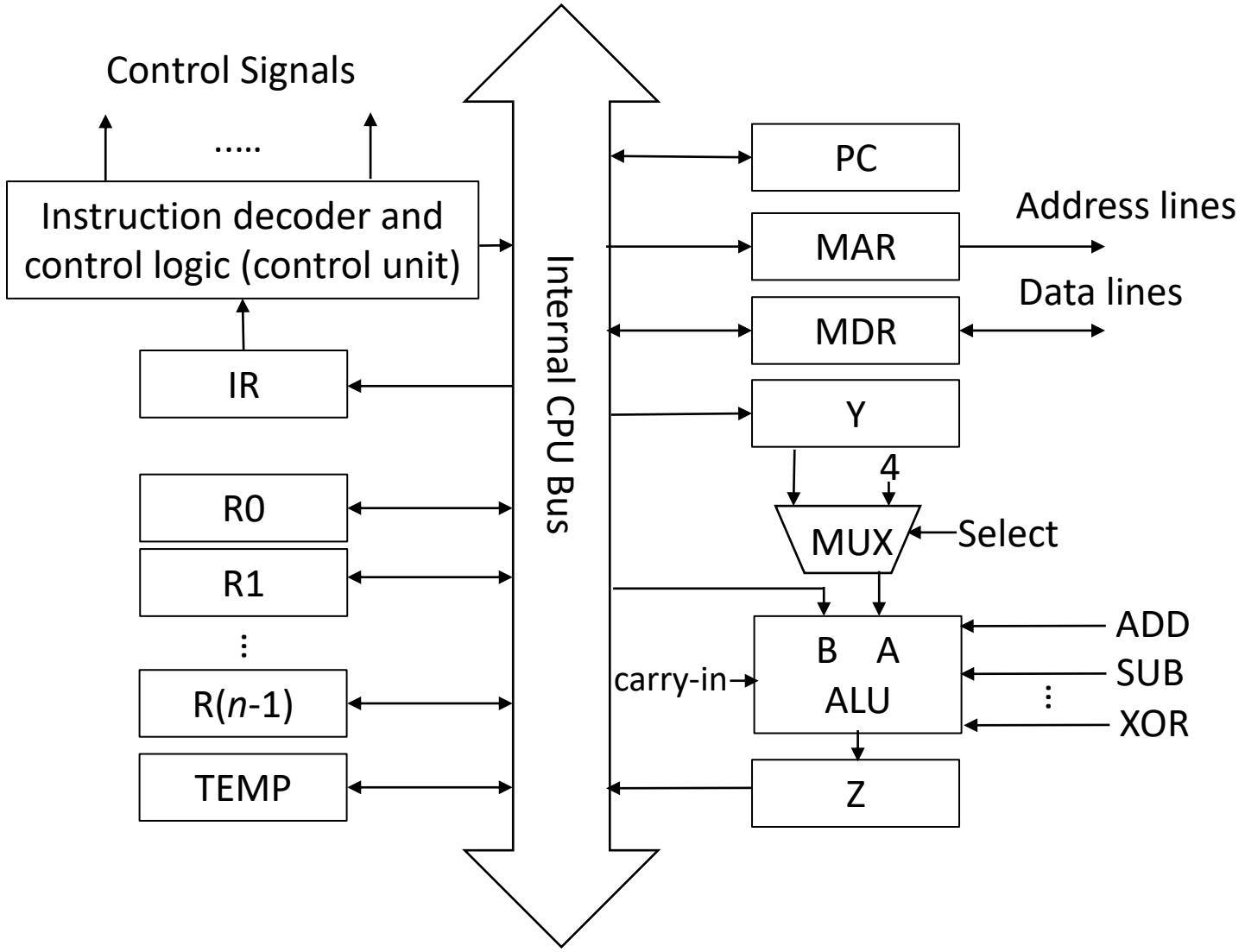
ALU Operation

- For a unary operation (i.e. INC EAX), this involves a source writing data onto the bus, the ALU being informed of what operation to perform, and the Z register copying the output, *i.e.*,
  - $source_{out}$, set carry-in, ALU_command, Z_{in}
  - Z_{out}, $destination_{in}$

ALU Operation

Example: INC EAX

- EAX_{out}, Set carry-in, ADD, Z_{in}
- Z_{out}, EAX_{in}



Memory Fetch

- This operation requires data to be read from the memory.
- This is performed by placing the memory address into the MAR, sending a Read control signal to the memory device, and getting the data from the MDR.
- When using an asynchronous memory device, the memory device indicates that the data is available on the MDR by sending a Memory Function Complete (MFC) signal to the CPU.

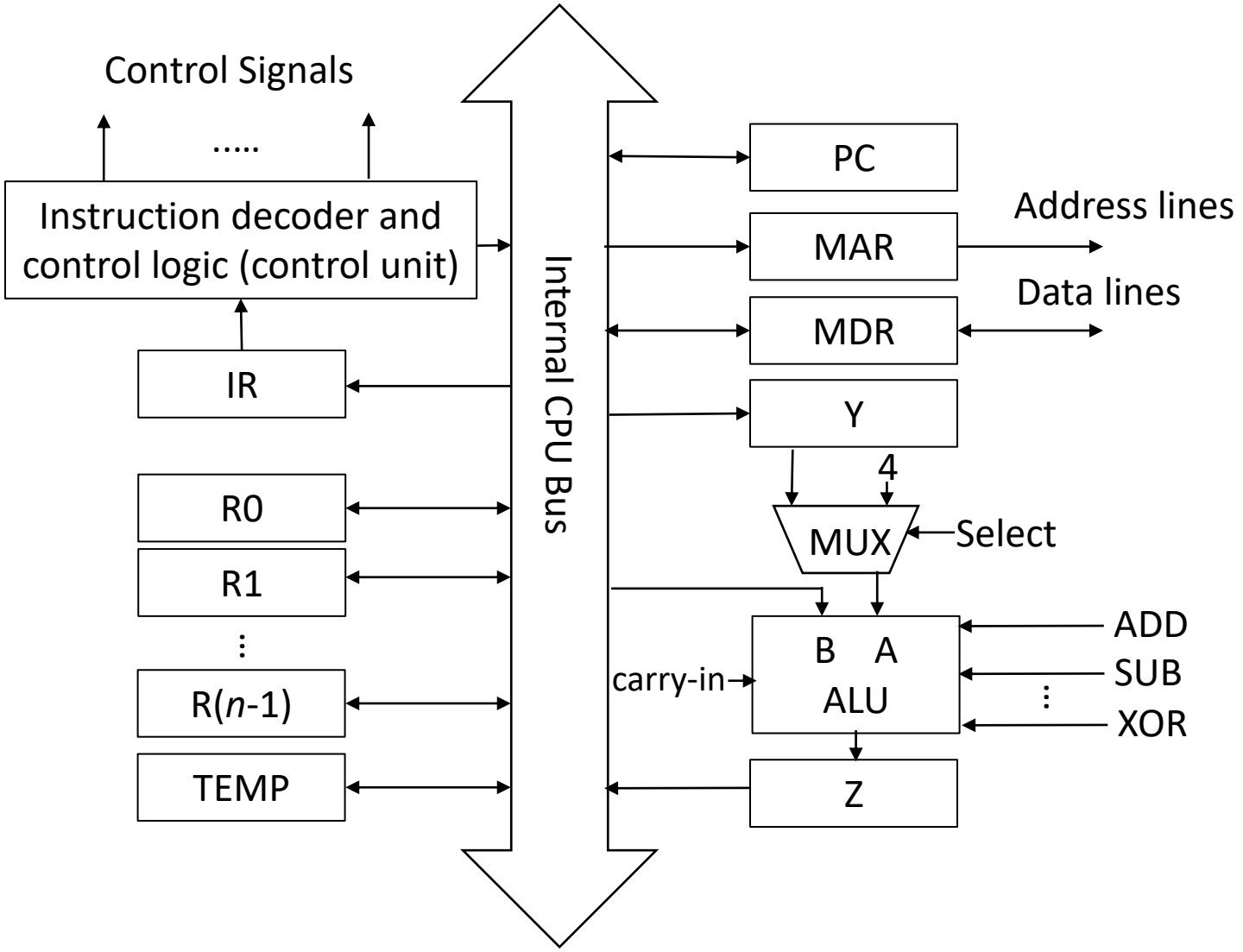
Memory Fetch

- Hence, in terms of gating signals, a memory fetch is performed as follows:
  - $address\_source_{out}$, MAR_{in}, Read, Wait for MFC (WMFC)
  - MDR_{out}, $destination_{in}$
- Note that during a WMFC, the CPU resumes execution only after MFC arrives.

Memory Fetch

Example: MOV EAX, [EBX]

- EBX_{out}, MAR_{in}, READ, WMFC
- MDR_{out}, EAX_{in}



Memory Store

- This operation requires data to be written in memory.
- This is performed by placing the memory address into the MAR, the data into the MDR, sending a Write signal to the memory device, and waiting for completion, *i.e.*,
  - address_source$_{out}$, MAR$_{in}$
  - *data_source*$_{out}$, MDR$_{in}$, Write, WMFC

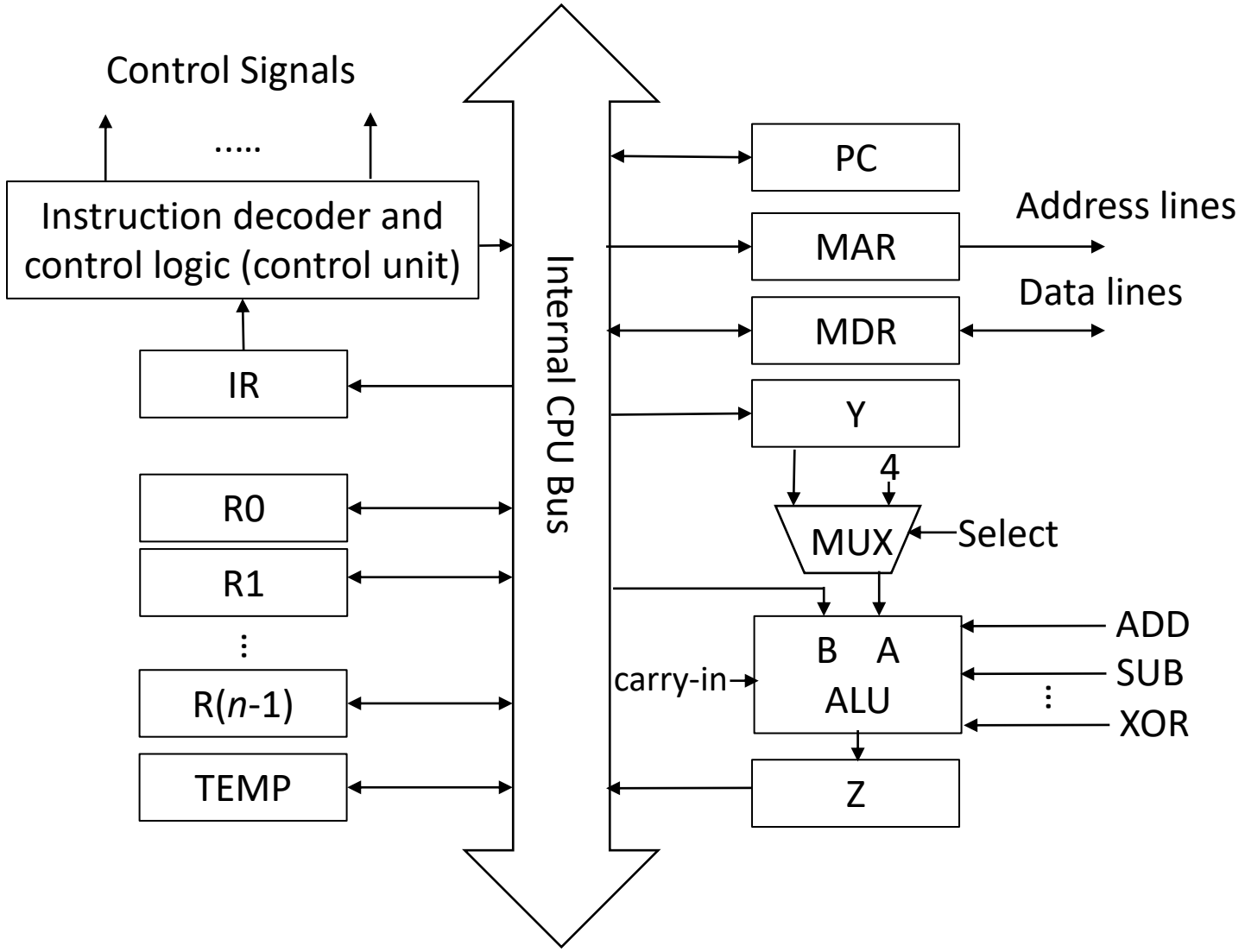
Memory Store

- Note that WMFC is not necessarily on the same line. Arrival of the MFC indicates that both the MAR and MDR may already be changed. Hence, the WMFC may be placed at a later line if neither the MAR nor the MDR are require at that time
- Moreover, placing of information in the MDR may precede that of the MAR.

Memory Store

Example: MOV [EAX], EBX

- EBX_{out}, MDR_{in}
- EAX_{out}, MAR_{in}, WRITE, WMFC



Instruction Execution

- Instruction execution is performed in two steps or phases
  - Fetch Phase
  - Execution Phase
- The decode operation is performed asynchronously by the Instruction Decoder once the instruction is copied into the IR.

Fetch Phase

- All instructions have the same fetch phase:
  - The instruction to be executed is taken from the memory location “pointed” to by the PC.
  - The content of the PC is incremented.

Fetch Phase

- The first step requires a memory fetch.
  - PC_{out}, MAR_{in}, Read, WMFC
  - MDR_{out}, IR_{in}

- The second step requires an increment operation. This is performed by adding the content of Y register with the bus
  - PC_{out}, Select4, Add, Z_{in}
  - Z_{out}, PC_{in}

Fetch Phase

- From these, it can be noted that:
  - The first lines of both steps place the content of the PC onto the bus and perform independent operations. Hence, these can be combined into one line.
  - The second lines are independent of each other and may be interchanged.

The optimal control sequence would then be:

1. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2. Z_{out}, PC_{in}, WMFC
3. MDR_{out}, IR_{in}

Execute Phase

- The execution phase is dependent on the instruction being executed

Microprogramming

- Once an instruction is fetched, it is decoded and translated into control signals
- The process of decoding and translation an instruction into control signals is known as microprogramming
- All control signals in each clock cycle (as shown in the previous line as line numbers) are “activated” at the same time
- There can only be one “out” control signal but can be multiple “in” control signals

Microprogramming

- Each instruction control sequence is terminated by an End signal. This causes the CU to start the fetch phase of the next instruction.
- For the purposes of this course, addition & subtraction is defined as follows:
  - Addition is via *bus* + Y (+ Carry-in)
  - Subtraction is via *bus* – Y (– Carry-in)

Microprogramming & Addressing Mode (x86)

Register Addressing mode

Example: MOV EAX, EBX

1. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2. Z_{out}, PC_{in}, WMFC
3. MDR_{out}, IR_{in}
4. EBX_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Immediate Addressing mode

Example: MOV EAX, 0x12345678

4. $IRDF_{out}$, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (displacement or absolute)

Example: MOV EAX, [ALPHA]

4. $IRAF_{out}$, MAR_{in}, READ, WMFC
5. MDR_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (base or register indirect)

Example: MOV EAX, [ESI]

4. ESI_{out}, MAR_{in}, READ, WMFC
5. MDR_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (base+displacement)

Example: MOV EAX, [ALPHA+ESI]

4. ESI_{out}, Yin
5. $IRAF_{out}$, SelectY, ADD, Zin
6. Z_{out}, MAR_{in}, READ, WMFC
7. MDR_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (base+index+displacement)

Example: MOV EAX, [ALPHA+ESI+EBX]

4. EBX_{out}, Y_{in}
5. ESI_{out}, SelectY, ADD, Z_{in}
6. Z_{out}, Y_{in}
7. $IRAF_{out}$, SelectY, ADD, Z_{in}
8. Z_{out}, MAR_{in}, READ, WMFC
9. MDR_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (index*scale+displacement)

Example: MOV EAX, [ALPHA+ESI*4]

4. ESI_{out}, Select4, MUL, Z_{in}
5. Z_{out}, Y_{in}
6. $IRAF_{out}$, SelectY, ADD, Z_{in}
7. Z_{out}, MAR_{in}, READ, WMFC
8. MDR_{out}, EAX_{in}, END

Microprogramming & Addressing Mode (x86)

Memory Addressing mode (base+(index*scale)+displacement)
Example: MOV EAX, [ALPHA+EBX+ESI*4]

4. ESI_{out}, Select4, MUL, Z_{in}
5. Z_{out}, Y_{in}
6. EBX_{out}, SelectY, ADD, Z_{in}
7. Z_{out}, Y_{in}
8. $IRAF_{out}$, SelectY, ADD, Z_{in}
9. Z_{out}, MAR_{in}, READ, WMFC
10. MDR_{out}, EAX_{in}, END

More Example (x86)

ADD EAX, [ALPHA+ESI]

4. ESI_{out}, Y_{in}
5. $IRAF_{out}$, SelectY, ADD, Z_{in}
6. Z_{out}, MAR_{in}, READ, WMFC
7. MDR_{out}, Y_{in}
8. EAX_{out}, SelectY, ADD, Z_{in}
9. Z_{out}, EAX_{in}, END

More Example (x86)

IMUL dword [ESI]

4. ESI_{out}, MAR_{in}, READ, WMFC
5. MDR_{out}, Y_{in}
6. EAX_{out}, SelectY, MUL, Z_{in}
7. Z_{out}, EAX_{in}, EDX_{in}, END

Microprogramming & Branching

- Branching refers to the changing of the sequence of execution by changing the content of the PC.
- Branching can either be:
  - Absolute: new content of PC is given explicitly
  - Relative: new content of PC is obtained by adding an offset to the current content

Microprogramming & Branching

Absolute Branch

Example:

JMP L1 ; assume L1 contains the address

4. $IRAF_{out}$, PC_{in}, END

Microprogramming & Branching

Relative Branch

Example:

JMP L1 ; assume L1 contains the offset

4. $IRDF_{out}$, Y_{in}
5. PC_{out}, SelectY, ADD, Z_{in}
6. Z_{out}, PC_{in}, END

Microprogramming & Branching

Branching can also be:

- Unconditional: PC is always changed
- Conditional: PC is changed if a certain condition is satisfied. Conditions are based on the status flags (i.e., ZF, CF, OF, PF, SF)

Microprogramming & Branching

Unconditional Branch (x86)

Example:

JMP L1

4. $IRDF_{out}$, Y_{in}
5. PC_{out}, SelectY, ADD, Z_{in}
6. Z_{out}, PC_{in}, END

Microprogramming & Branching

Conditional Branch (x86)

Example: JC L1

4. If (CF==0) then END
   $IRDF_{out}$, Y_{in}
5. PC_{out}, SelectY, ADD, Z_{in}
6. Z_{out}, PC_{in}, END