COA

Ch2- Part1
Introduction to CPU Architecture

By

Dr. Raghad Samir Al Najim

2017@SPU

Introduction to CPU Architecture

The operation or task that must perform by CPU are:

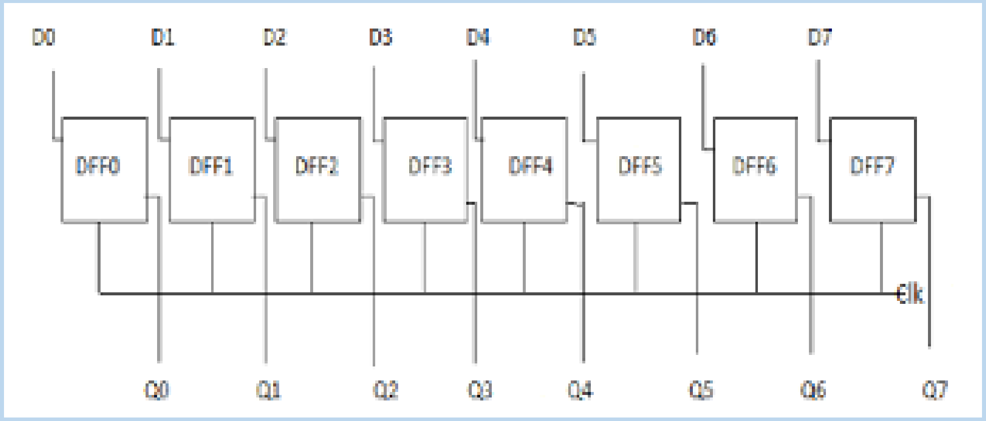
1. Fetch Instruction: The CPU reads an instruction from memory.
2. Interprete(decode) Instruction: The instruction is decoded to determine what action is required.
3. Fetch Data: The execution of an instruction may require reading data from memory or I/O module.
4. Process data: The execution of an instruction may require performing some arithmatic or logical operation on data.
5. Write(store) data: The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is beign executed. In other words, the CPU needs a small internal memory.
These storage location are generally referred as registers.

The Registers

Registers are used in computer systems as places to store a wide variety of data, such as addresses, program counters, or data necessary for program execution.
Simply, a register is a hardware device that stores binary data.
We Know that D flip-flops can be used to implement registers. One D flip-flop is equivalent to a 1-bit register, so a collection of D flip-flops is necessary to store multi-bit values.

For example, to build a 8-bit register, we need to connect 8 D flip-flops together. At each pulse of the clock, input enters the register and cannot be changed (and thus is stored) until the clock pulses again.



Data processing on a computer is usually done on fixed size binary words that are stored in registers. Therefore, most computers have registers of a certain size. Common sizes include 16, 32, and 64 bits. The number of registers in a machine varies from architecture to architecture, but is typically a power of 2.

Registers contain data, addresses, or control information. Some registers are specified as "special purpose" and may contain only data, only addresses, or only control information. Other registers are more generic and may hold data, addresses, and control information at various times. Information is written to registers, read from registers, and transferred from register to register. Registers are addressed and manipulated by the control unit itself.

The major components of the CPU are:

an Arithmatic and Logic Unit (ALU) and a Control Unit (CU) .

The ALU does the actual computation or processing of data .

The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU.

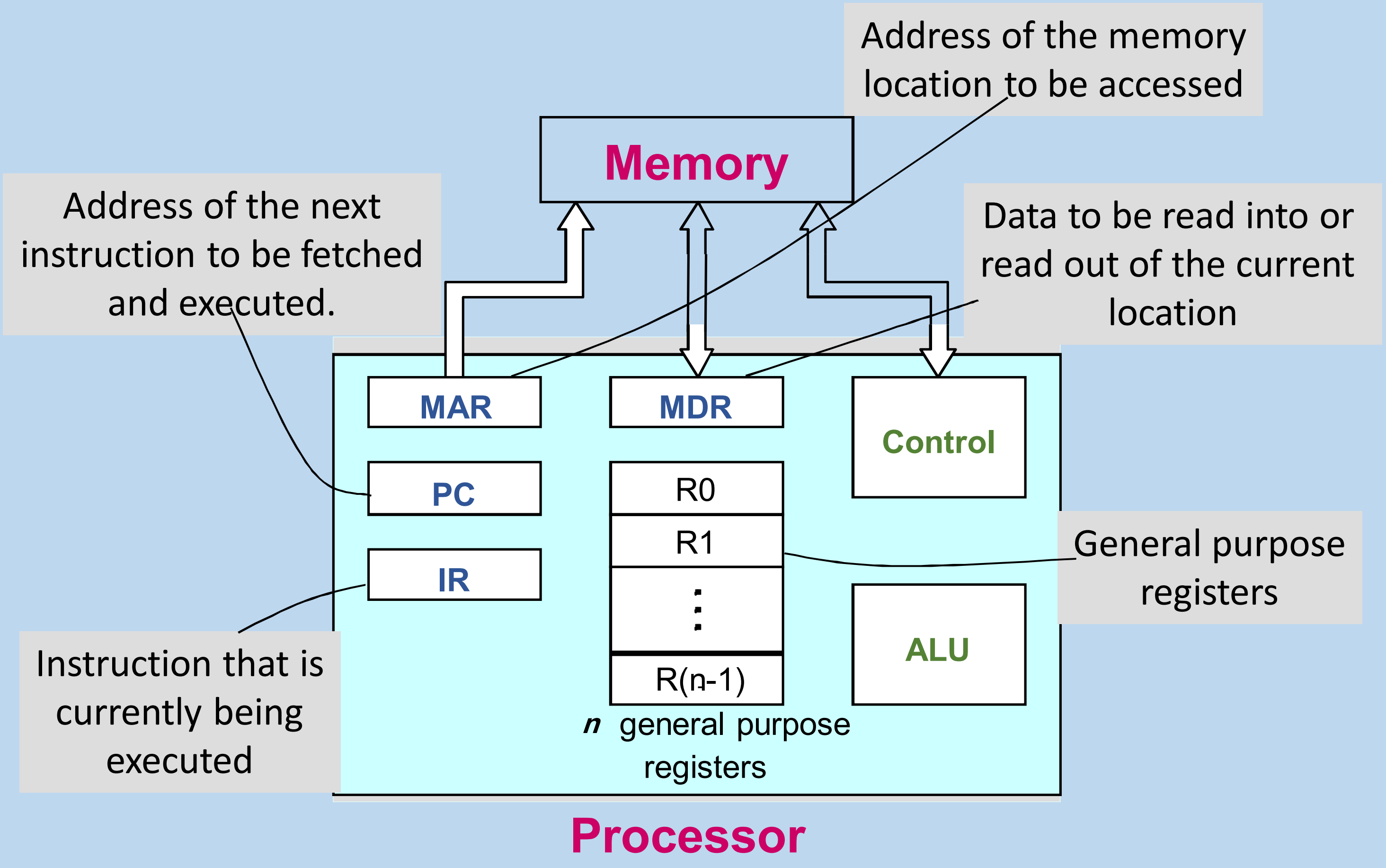
The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components:

1. Data Bus: Data bus is used to transfer the data between main memory and CPU.
2. Address Bus: Address bus is used to access a particular memory location by putting the address of the memory location.
3. Control Bus: Control bus is used to provide the different control signal generated by CPU to different part of the system .

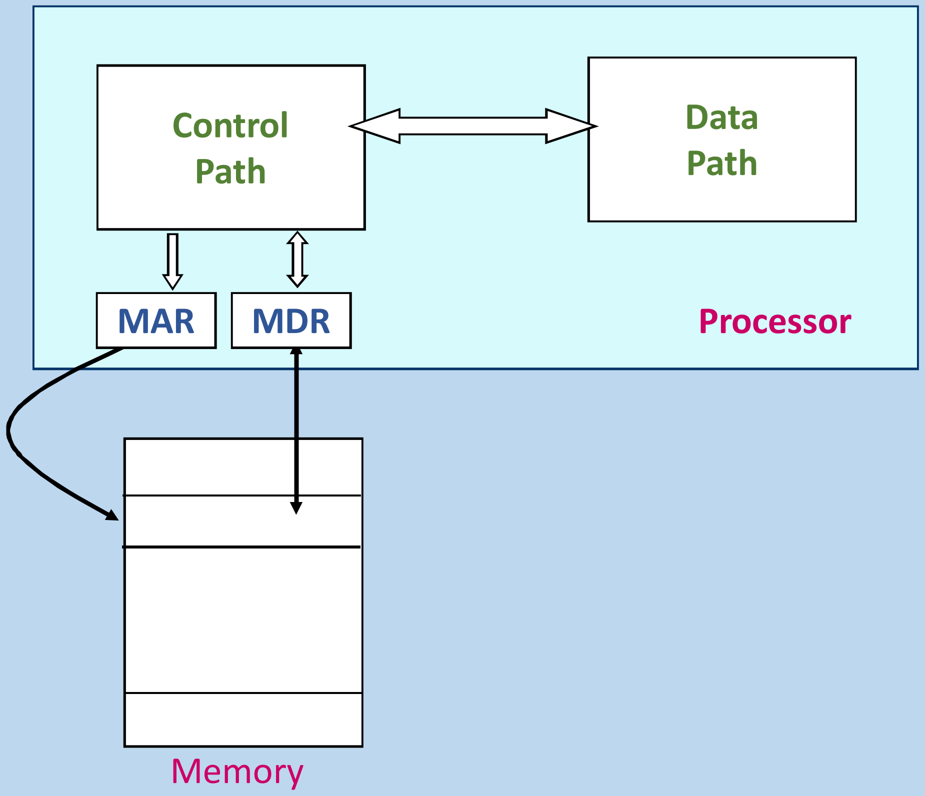
As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

CPU Architecture

Basic processor architecture


Address of the memory location to be accessed
Memory
Address of the next instruction to be fetched and executed.
Data to be read into or read out of the current location
MAR
MDR
Control
PC
R0
R1
⋮
R(n-1)
IR
General purpose registers
ALU
Instruction that is currently being executed
n general purpose registers
Processor

Basic processor architecture

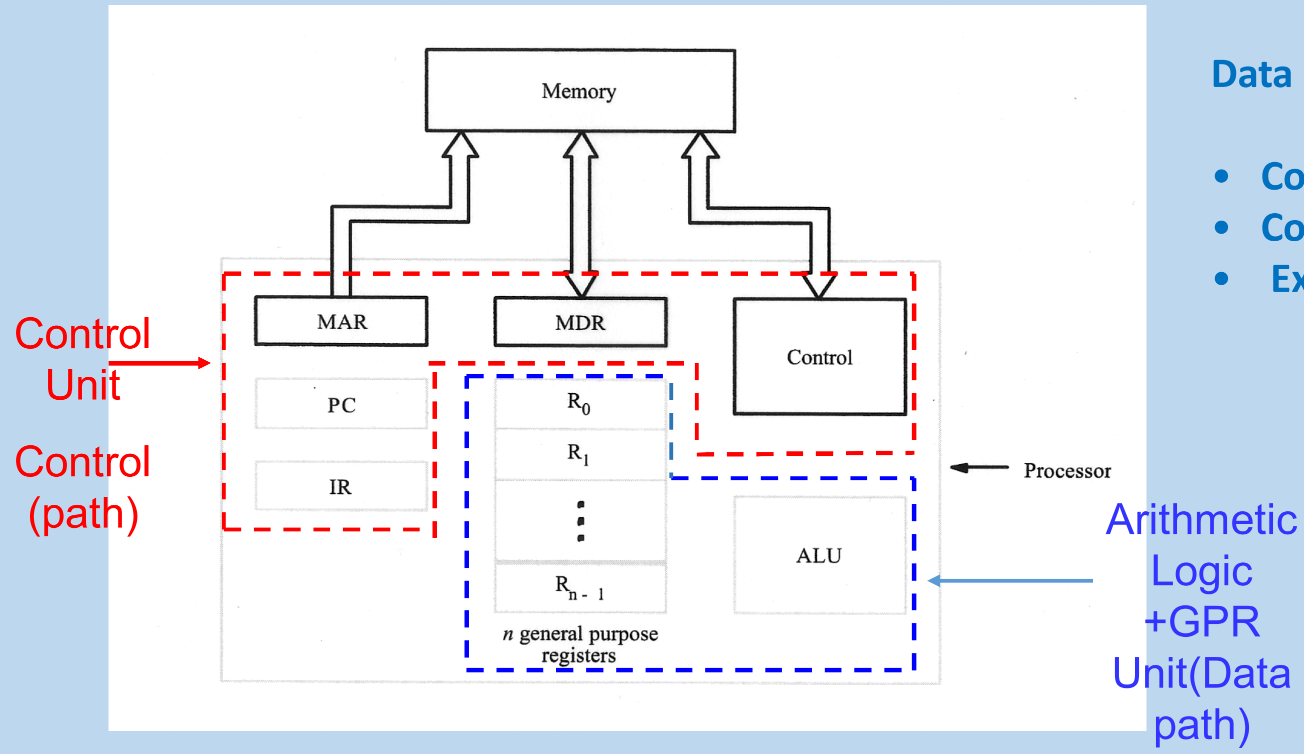


Control path is responsible for:

- Instruction fetch and execution sequencing
- Operand fetch
- Saving results

Data path:

- Contains general purpose registers
- Contains ALU
- Executes instructions



Data path:

- **Contains general purpose registers**
- **Contains ALU**
- **Executes instructions**

PC - Program Counter

IR - Instruction Register

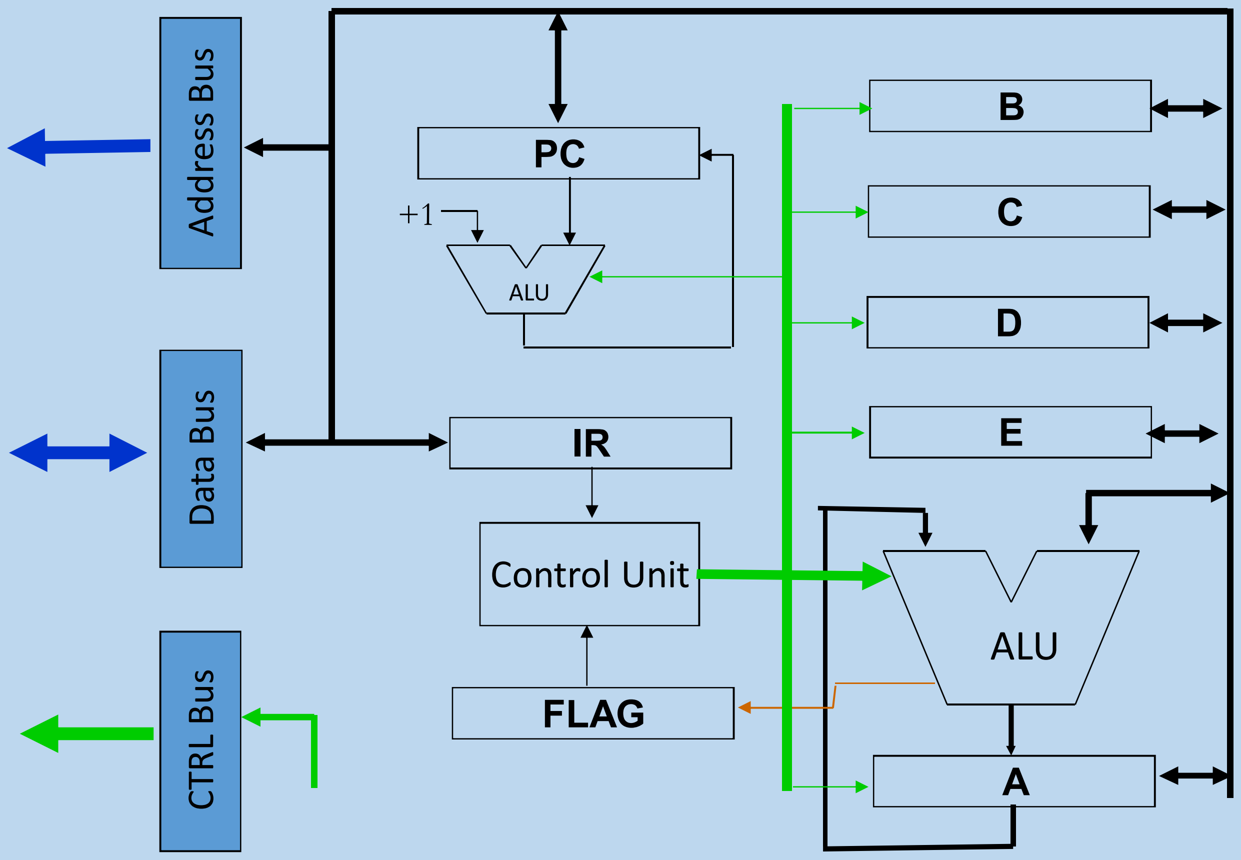
MAR - Memory Address Register

MDR - Memory Data Register

Control path is responsible for:

- **Instruction fetch and execution sequencing**
- **Operand fetch**
- **Saving results**

Overall Internal Structure of a simple CPU



MARIE

To understand Computer architecture, we will use MARIE, a Machine Architecture that is Really Intuitive and Easy, is a simple architecture consisting of memory (to store programs and data) and a CPU (consisting of an ALU and several registers).

It has all the functional components necessary to be a real working computer.

We describe MARIE's architecture in the following sections.

The Architecture MARIE has the following characteristics:

- Fixed word length, Word (but not byte) addressable. 16-bit data (words have 16 bits)

- 4K words of main memory (this implies 12 bits per address)
- 16-bit instructions, 4 for the opcode and 12 for the address
- A 16-bit accumulator (AC)

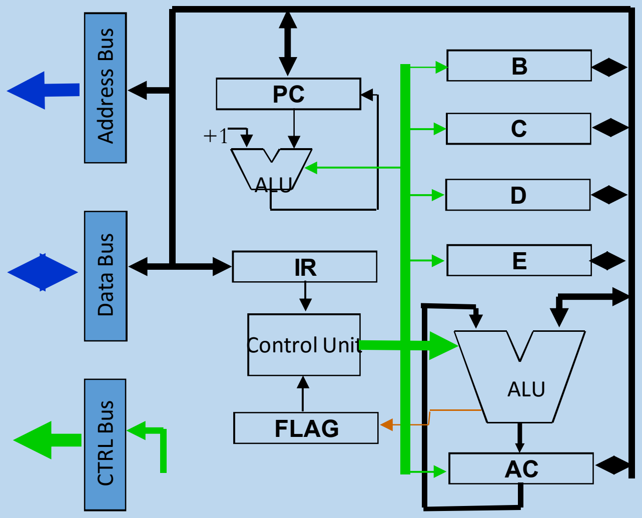
- A 16-bit instruction register (IR)
- A 16-bit memory buffer register (MBR)

$\left.\right\}$ Instruction related

- A 12-bit program counter (PC)
- A 12-bit memory address register (MAR).

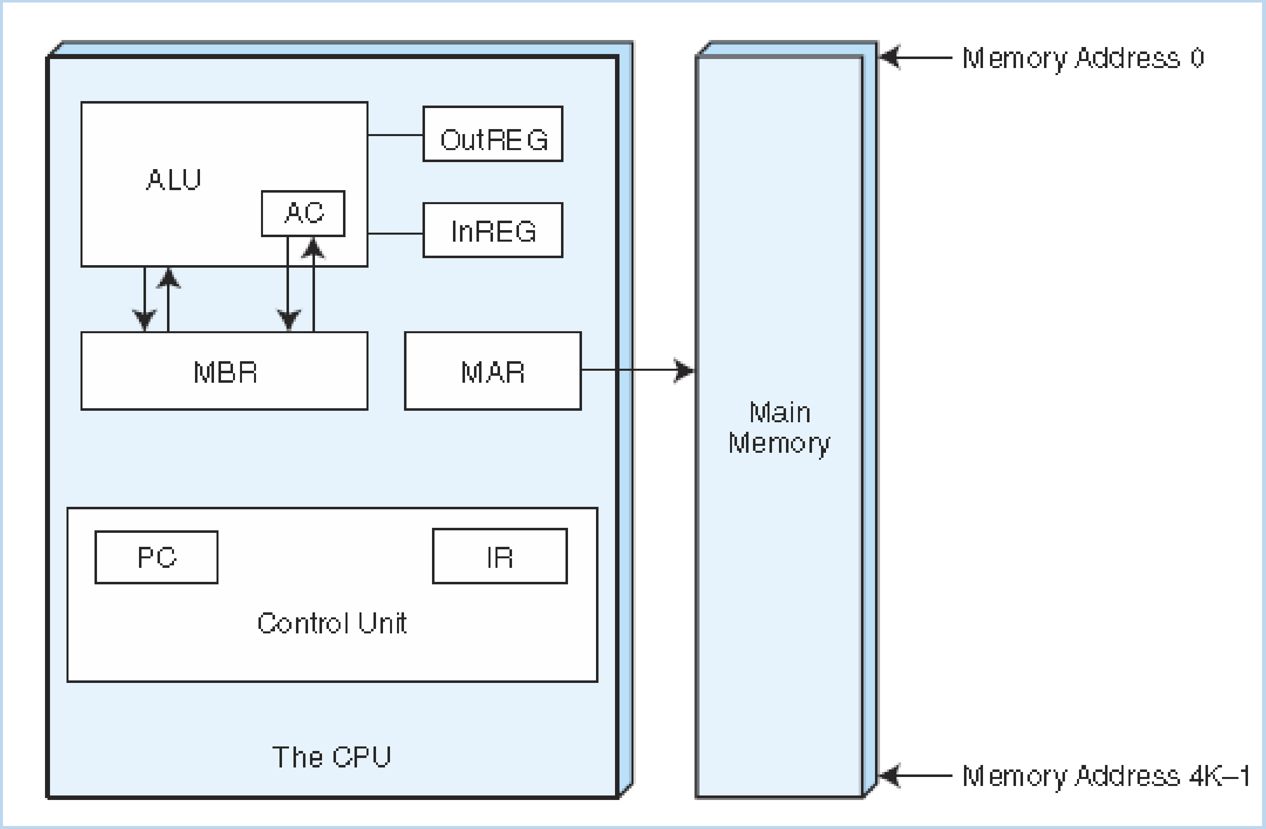
$\left.\right\}$ Address related

- An 8-bit input register & An 8-bit output register



Marie’s Architecture

- A 16-bit (AC)
- A 16-bit (IR)
- A 16-bit (MBR)
- A 12-bit (PC)
- A 12-bit (MAR)



In MARIE, there are seven registers, as follows:

- AC: The accumulator, holds data. This is a general purpose register holds data that the CPU needs to process. Most computers today have multiple general purpose registers.
- MAR: The memory address register, holds the memory address of the data being referenced.
- MBR(MDR): The memory buffer(or data) register, which holds either the data just read from memory or the data ready to be written to memory.
- PC: The program counter, holds the address of the next instruction to be executed.
- IR: The instruction register, which holds the next instruction to be executed.
- InREG: The input register, which holds data from the input device.
- OutREG: The output register, which holds data for the output device.

In addition, there is a status or flag register that holds information indicating various conditions, such as an overflow in the ALU.

MARIE is a very simple computer with a limited register set. Modern CPUs have multiple general purpose registers

Registers in the control path

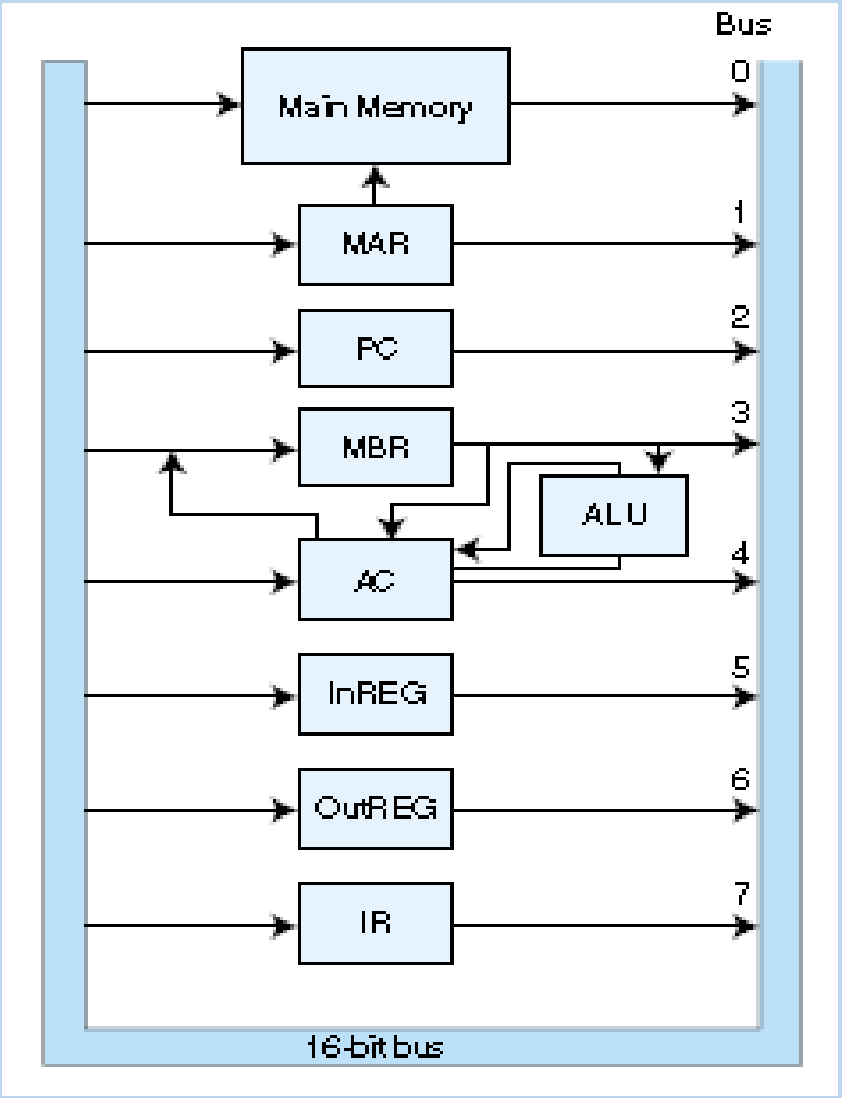
- Instruction Register (IR):
  - Instruction that is currently being executed.
- Program Counter (PC):
  - Address of the next instruction to be fetched and executed.
- Memory Address Register (MAR):
  - Address of the memory location to be accessed.
- Memory Data Register (MDR):
  - Data to be read into or read out of the current memory location, whose address is in the Memory Address Register (MAR).

Path through the bus

we assume a common bus scheme. Each device connected to the bus has a number, and before the device can use the bus, it must be set to that identifying number. We also have some pathways to speed up execution.

- a communication path between the MAR and memory.
- a separate path from the MBR to the AC.
- a special path from the MBR to the ALU to allow the data in the MBR to be used in arithmetic operations.

Information can also flow from the AC through the ALU and back into the AC without being put on the common bus.

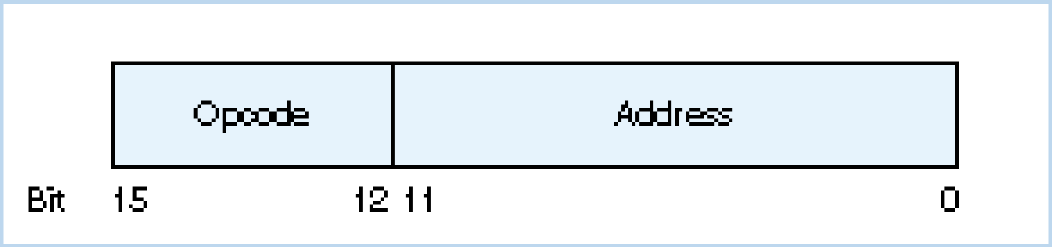


The Instruction Set Architecture (ISA)

MARIE has a very simple, instruction set. The instruction set architecture (ISA) of a machine specifies the instructions that the computer can perform and the format for each instruction. The ISA is essentially an interface between the software and the hardware.

Each instruction for MARIE consists of 16 bits. The most significant 4 bits, bits 12–15, make up the opcode that specifies the instruction to be executed (which allows for a total of 16 instructions).

The least significant 12 bits, bits 0–11, form an address, which allows for a maximum memory size of 2^{12}–1.



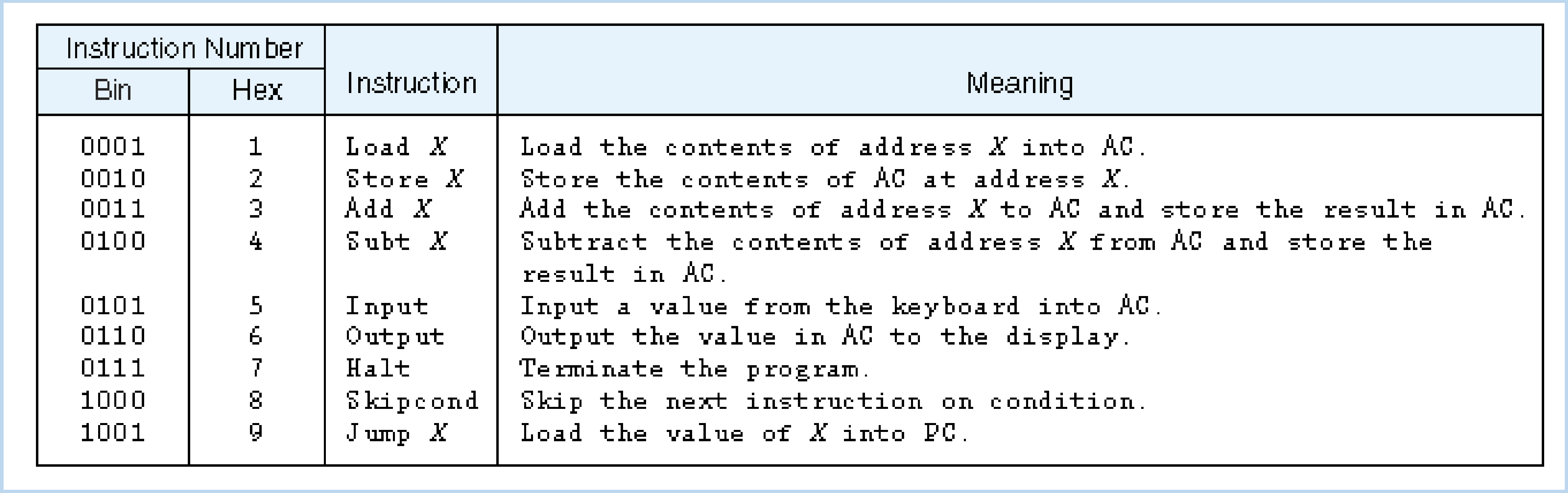
Instruction

Instruction types

Most ISAs Computer instructions must be capable of performing 4 types of operations:

- Data transfer/movement between memory and processor registers.
  - E.g., memory read, memory write
- Arithmetic and logic operations:
  - E.g., addition, subtraction, comparison between two numbers.
- Program sequencing and flow of control:
  - Branch instructions
- Input/output transfers to transfer data to and from the real world.

MARIE's instruction set consists of the instructions shown:



Instruction Number		Instruction	Meaning
Bin	Hex		
0001	1	Load *X*	Load the contents of address *X* into AC.
0010	2	Store *X*	Store the contents of AC at address *X*.
0011	3	Add *X*	Add the contents of address *X* to AC and store the result in AC.
0100	4	Subt *X*	Subtract the contents of address *X* from AC and store the result in AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate the program.
1000	8	Skipcond	Skip the next instruction on condition.
1001	9	Jump *X*	Load the value of *X* into PC.

MARIE’s instruction set are:

The Load instruction allows us to move data from memory into the CPU (via the MBR and the AC). All data (which includes anything that is not an instruction) from memory must move first into the MBR and then into either the AC or the ALU; there are no other options in this architecture.
Notice that the Load instruction does not have to name the AC as the final destination; this register is implicit in the instruction. Other instructions reference the AC register in a similar fashion.

The Store instruction allows us to move data from the CPU back to memory.

The Add and Subt instructions add and subtract, respectively, the data value found at address X to or from the value in the AC. The data located at address X is copied into the MBR where it is held until the arithmetic operation is executed.

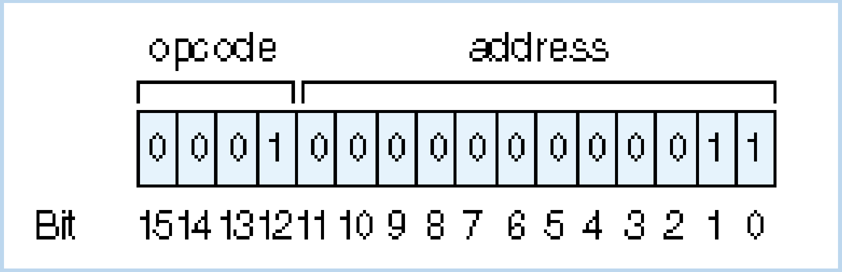
Input and Output allow MARIE to communicate with the outside world.
The Halt command causes the current program execution to terminate.

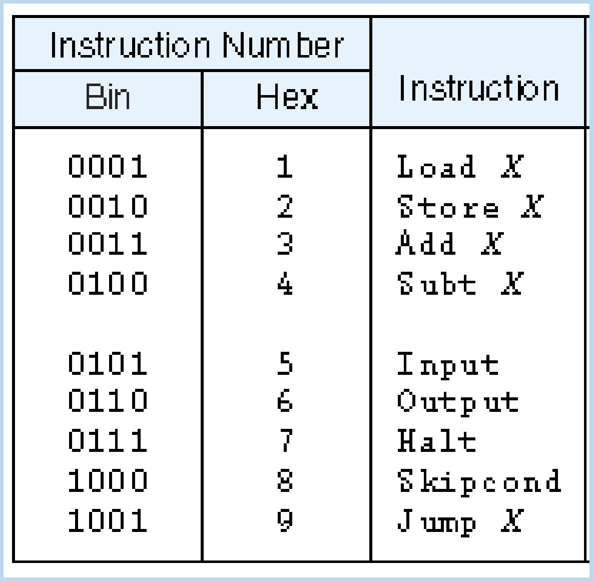
The Skipcond instruction allows us to perform conditional branching (as is done with "while" loops or "if" statements). The Skipcond instruction is executed, based on the value stored in the AC.
Two of the address bits (let's assume we always use the two address bits closest to the opcode field, bits A10 and A11) specify the condition to be tested.

If the two address bits are 00, this translates to "skip if the AC is negative."
If the two address bits are 01 (bit eleven is 0 and bit ten is 1), this translates to "skip if the AC is equal to 0."
Finally, if the two address bits are 10 , this translates to "skip if the AC is greater than 0."

By "skip" we simply mean jump over the next instruction. This is accomplished by incrementing the PC by 1, essentially ignoring the following instruction, which is never fetched.
The Jump instruction, an unconditional branch, also affects the PC.
This instruction causes the contents of the PC to be replaced with the value of X, which is the address of the next instruction to fetch.

Let’s examine the instruction format used in MARIE. Suppose we have the following 16-bit instruction:



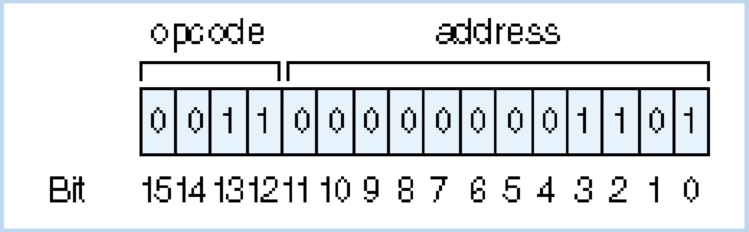


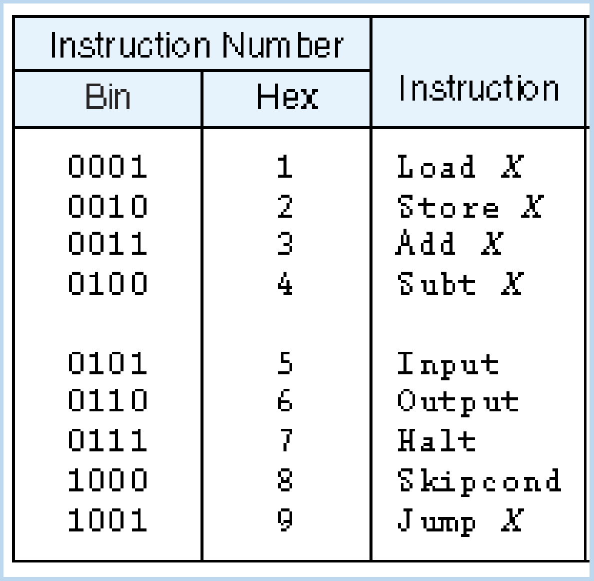
Instruction Number		Instruction
Bin	Hex	
0001	1	Load *X*
0010	2	Store *X*
0011	3	Add *X*
0100	4	Subt *X*
0101	5	Input
0110	6	Output
0111	7	Halt
1000	8	Skipcond
1001	9	Jump *X*

The leftmost 4 bits indicate the opcode, or the instruction to be executed.

This instruction causes the data value found in main memory, address 3, to be copied into the AC.

Consider another instruction:

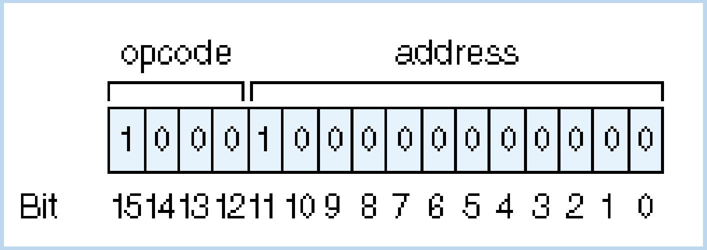


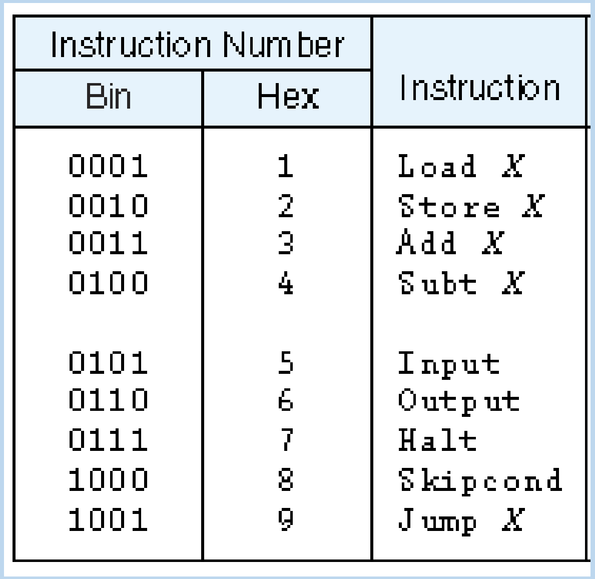


Instruction Number		Instruction
Bin	Hex	
0001	1	Load *X*
0010	2	Store *X*
0011	3	Add *X*
0100	4	Subt *X*
0101	5	Input
0110	6	Output
0111	7	Halt
1000	8	Skipcond
1001	9	Jump *X*

We go to main memory, get the data value at address 00D, and add this value to the AC. The value in the AC would then change to reflect this sum.

One more example follows:





Instruction Number		Instruction
Bin	Hex	
0001	1	Load X
0010	2	Store X
0011	3	Add X
0100	4	Subt X
0101	5	Input
0110	6	Output
0111	7	Halt
1000	8	Skipcond
1001	9	Jump X

The opcode for this instruction represents the Skipcond instruction. Bits ten and eleven (10). This implies a “skip if AC greater than 0.”
If the value in the AC is less than zero, this instruction is ignored and we simply go on to the next instruction.

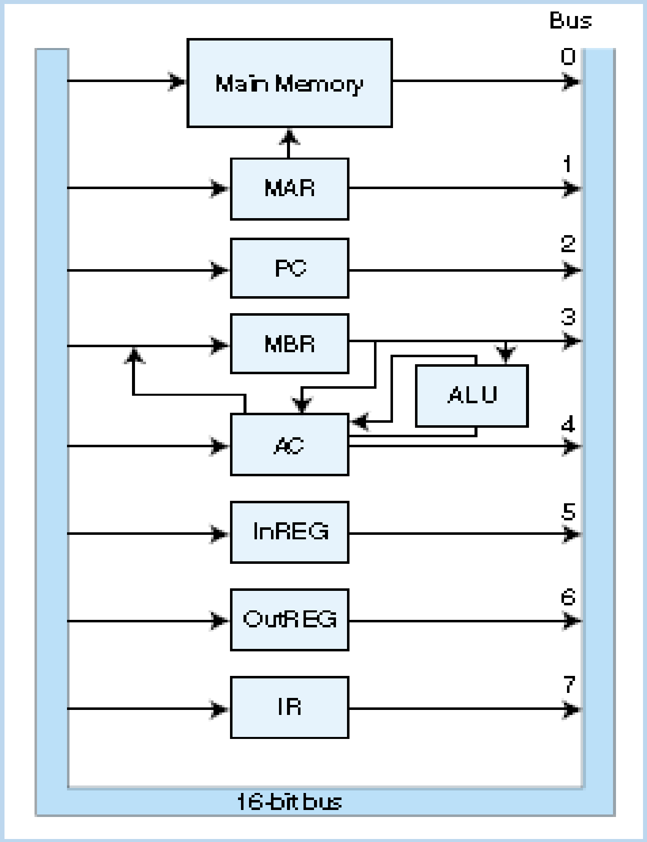
If the value in the AC is greater than zero, this instruction causes the PC to be incremented by 1, thus causing the instruction immediately following this instruction in the program to be ignored (keep this in mind as you read the following section on the instruction cycle).

Register Transfer Notation

In the Load instruction loads the contents of the given memory location into the AC register. But, if we observe what is happening at the component level, we see that multiple “mini-instructions” are being executed.

- First, the address from the instruction must be loaded into the MAR.
- Then the data in memory at this location must be loaded into the MBR.
- Then the MBR must be loaded into the AC.

These mini-instructions are called microoperations and specify the elementary operations that can be performed on data stored in registers.



The symbolic notation used to describe the behavior of microoperations is called register transfer notation (RTN) or register transfer language (RTL).

We use the notation M[X] to indicate the actual data stored at location X in memory, and → to indicate a transfer of information.

In reality, a transfer from one register to another always involves a transfer onto the bus from the source register, and then a transfer off the bus into the destination register

We now present the register transfer notation for each of the instructions in the ISA for MARIE.

❖Load X

This instruction loads the contents of memory location X into the AC.
However, the address X must first be placed into the MAR. Then the data at location M[MAR] (or address X) is moved into the MBR. Finally, this data is placed in the AC.

$$\mathrm{MAR} \xleftarrow{1} \mathrm{X}$$
$$\mathrm{MBR} \xleftarrow{2} \mathrm{M[MAR]},\ \mathrm{AC} \xleftarrow{3} \mathrm{MBR}$$

❖ Store X

This instruction stores the contents of the AC in memory location X:

$$\mathrm{MAR} \xleftarrow{1} \mathrm{X},\ \mathrm{MBR} \xleftarrow{2} \mathrm{AC}$$
$$\mathrm{M[MAR]} \xleftarrow{3} \mathrm{MBR}$$

❖ Add X

The data value stored at address X is added to the AC. This can be accomplished as follows:

$$\begin{aligned} &\mathrm{MAR} \leftarrow \mathrm{X} \\ &\mathrm{MBR} \leftarrow \mathrm{M[MAR]} \\ &\mathrm{AC} \leftarrow \mathrm{AC} + \mathrm{MBR} \end{aligned}$$

❖ Subt X

Similar to Add, this instruction subtracts the value stored at address X from the accumulator and places the result back in the AC:

$$\begin{aligned} &\mathrm{MAR} \leftarrow \mathrm{X} \\ &\mathrm{MBR} \leftarrow \mathrm{M[MAR]} \\ &\mathrm{AC} \leftarrow \mathrm{AC} - \mathrm{MBR} \end{aligned}$$

❖ Input

Any input from the input device is first routed into the InREG. Then the data is transferred into the AC.

$$\mathrm{AC} \leftarrow \mathrm{InREG}$$

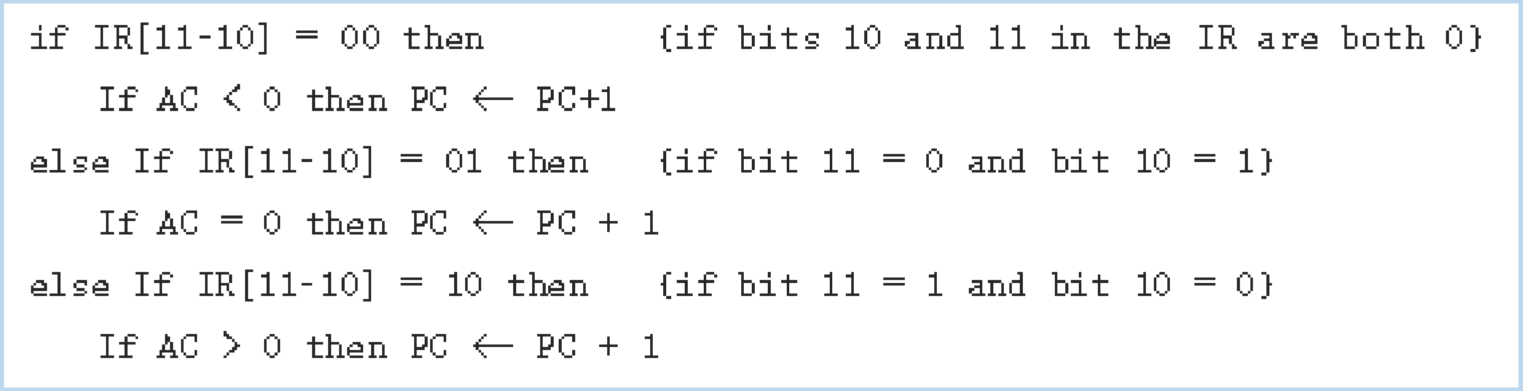
❖ Output

This instruction causes the contents of the AC to be placed into the OutREG, where it is eventually sent to the output device.

$$\mathrm{OutREG} \leftarrow \mathrm{AC}$$

❖ Skipcond

This instruction uses the bits in positions 10 and 11 in the address field (A10,A11)to determine what comparison to perform on the AC. The AC is checked to see whether it is negative(-ve), equal to zero, or greater than zero. If the given condition is true, then the next instruction is skipped. This is performed by incrementing the PC register by 1.



```
if IR[11-10] = 00 then          {if bits 10 and 11 in the IR are both 0}
    If AC < 0 then PC ← PC+1
else If IR[11-10] = 01 then     {if bit 11 = 0 and bit 10 = 1}
    If AC = 0 then PC ← PC + 1
else If IR[11-10] = 10 then     {if bit 11 = 1 and bit 10 = 0}
    If AC > 0 then PC ← PC + 1
```

If the bits in positions ten and eleven (A10,A11)are both ones, an error condition results. However, an additional condition could also be defined using these bit values.

❖ Jump X

This instruction causes an unconditional branch to the given address, X. Therefore, to execute this instruction, X must be loaded into the PC.

$$PC \leftarrow X$$

In reality, the lower or least significant 12 bits of the instruction register (or IR[11–0]) reflect the value of X. So this transfer is more accurately depicted as:

$$PC \leftarrow IR[11\text{-}0]$$

INSTRUCTION PROCESSING

All computers follow a basic machine cycle(Instruction Cycle): the fetch, decode, and execute cycle.

The Fetch-Decode-Execute Cycle (Instruction cycle)

The CPU fetches an instruction (transfers it from main memory to the instruction register IR), decodes it (determines the opcode and fetches any data necessary to carry out the instruction), and executes it (performs the operation(s) indicated by the instruction).

Notice that a large part of this cycle is spent copying data from one location to another. When a program is initially loaded, the address of the first instruction must be placed in the PC. The steps in this cycle, which take place in specific clock cycles, are listed below. Note that Steps 1 and 2 make up the fetch phase, Step 3 makes up the decode phase, and Step 4 is the execute phase.

Fetch/Execute cycle (Instruction Cycle)

- Execution of an instruction takes place in two phases:
  - Instruction fetch.
  - Instruction execute(decoding+execution).
- Instruction fetch:
  - Fetch the instruction from the memory location whose address is in the Program Counter (PC).
  - Place the instruction in the Instruction Register (IR).
- Instruction execute:
  - Instruction in the IR is examined (decoded) to determine which operation is to be performed.
  - Fetch the operands from the memory or registers.
  - Execute the operation.
  - Store the results in the destination location.
- Basic fetch/execute cycle repeats indefinitely.

1. Copy the contents of the PC to the MAR: `MAR ← PC.`

2. Go to main memory and fetch the instruction found at the address in the MAR, placing this instruction in the IR; `IR ← M[MAR]` increment PC by 1 (PC now points to the next instruction in the program. `PC ← PC+1.`

Fetch

(Note: Because MARIE is word-addressable, the PC is incremented by one, which results in the next word's address occupying the PC. If MARIE were byte addressable, the PC would need to be incremented by 2 to point to the address of the next instruction, because each instruction would require two bytes. On a byte-addressable machine with 32-bit words, the PC would need to be incremented by 4.)

Decoding

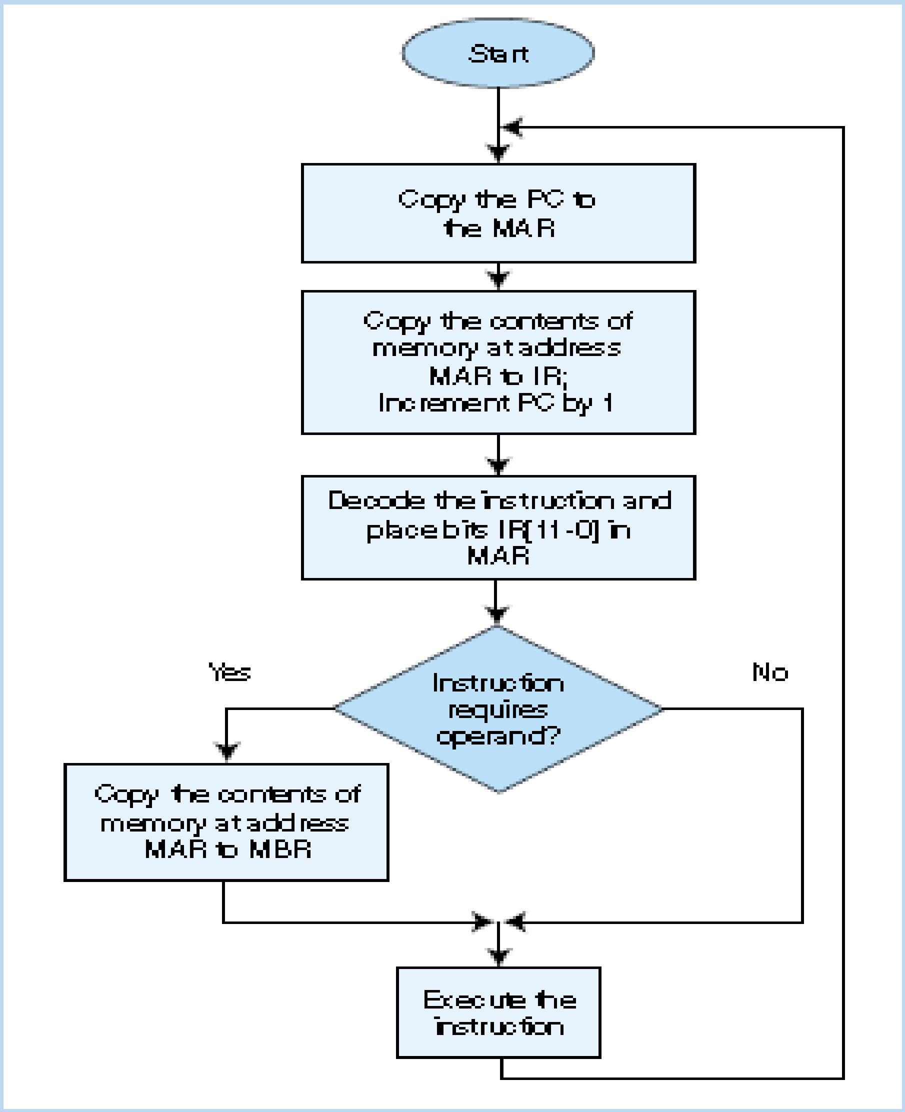
3. Copy the rightmost 12 bits of the IR into the MAR; decode the leftmost four bits to determine the opcode, `MAR ← IR[11-0],` and decode IR[15–12].

executing

4. If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR (and possibly the AC), and then execute the instruction `MBR ← M[MAR]` and execute the actual instruction.

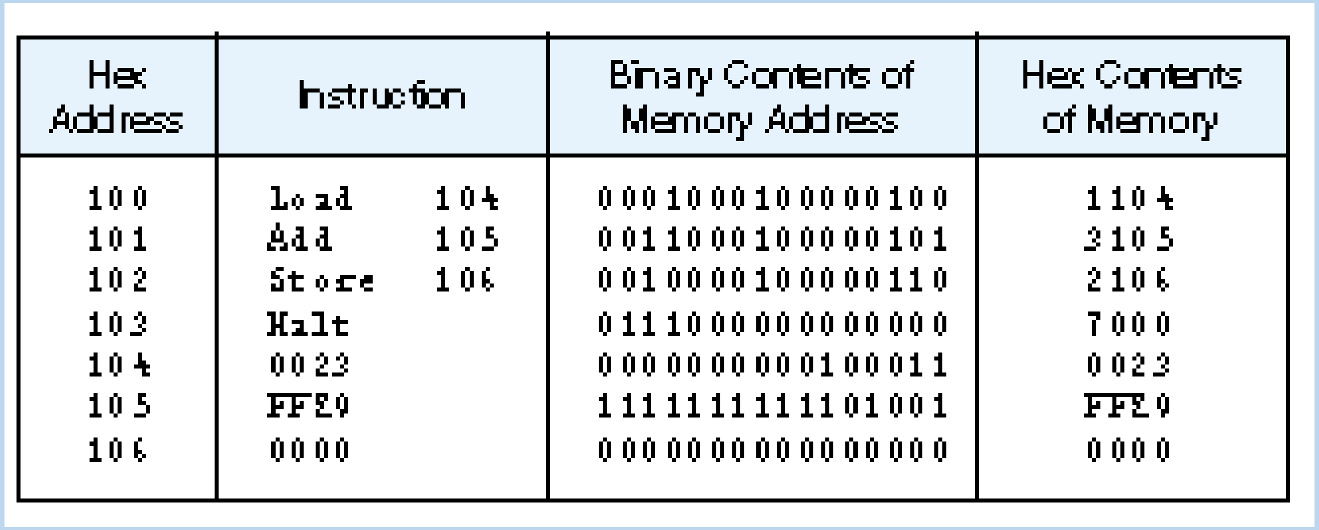
This cycle is illustrated in the flowchart shown

Note that computers today, even with large instruction sets, long instructions,
and huge memories, can execute millions of these fetch-decode-execute cycles in the blink of an eye.

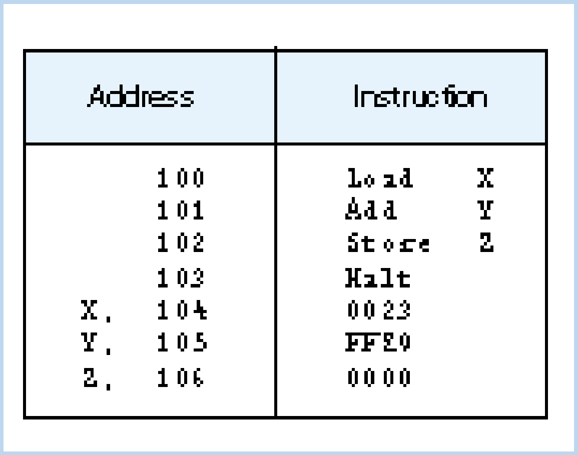

Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR; Increment PC by 1
Decode the instruction and place bits IR[11-0] in MAR
Instruction requires operand?
Yes
No
Copy the contents of memory at address MAR to MBR
Execute the instruction

Simple Program in MARIE

A simple program to add two numbers together (both are in main memory), storing the sum in memory. An assembly language program to do this, along with its corresponding, machine-language program.



Hex Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0010000100000110	2106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000



Address		Instruction
	100	Load X
	101	Add Y
	102	Store Z
	103	Halt
X,	104	0023
Y,	105	FFE9
Z,	106	0000

Lets see how to execute

Hex Address	Instruction	Hex Contents of Memory
100	Load 104	1104
101	Add 105	3105
102	Store 106	2106
103	Halt	7000
104	0023	0023
105	FFE9	FFE9
106	0000	0000

a) Load 104

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ← PC	100	------	100	------	------
	IR ← M[MAR]	100	1104	100	------	------
	PC ← PC + 1	101	1104	100	------	------
Decode	MAR ← IR[11–0]	101	1104	104	------	------
	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR ← M[MAR]	101	1104	104	0023	------
Execute	AC ← MBR	101	1104	104	0023	0023

b) Add 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ⟵ PC	101	1104	101	0023	0023
	IR ⟵ M[MAR]	101	3105	101	0023	0023
	PC ⟵ PC + 1	102	3105	101	0023	0023
Decode	MAR ⟵ IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ⟵ M[MAR]	102	3105	105	FFE9	0023
Execute	AC ⟵ AC + MBR	102	3105	105	FFE9	000C

c) Store 106

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ⟵ PC	102	3105	102	FFE9	000C
	IR ⟵ M[MAR]	102	2106	102	FFE9	000C
	PC ⟵ PC + 1	103	2106	102	FFE9	000C
Decode	MAR ⟵ IR[11–0]	103	2106	106	FFE9	000C
	(Decode IR[15–12])	103	2106	106	FFE9	000C
Get operand	(not necessary)	103	2106	106	FFE9	000C
Execute	MBR ⟵ AC	103	2106	106	000C	000C
	M[MAR] ⟵ MBR	103	2106	106	000C	000C

Hex Address	Instruction	Hex Contents of Memory
100	Load 104	1104
101	Add 105	3105
102	Store 106	2106
103	Halt	7000
104	0023	0023
105	FFE9	FFE9
106	0000	0000

Number of Operands and Instruction Length

MARIE uses a fixed length instruction with a 4-bit opcode and a 12-bit operand.

The instruction length must also be compared to the word length on the machine.

The most common instruction formats include zero, one, two, or three operands.
some instructions for MARIE have no operands, whereas others have one operand.
Arithmetic and logic operations typically have two operands, but can be executed with one operand (as we saw in MARIE), if the accumulator is implicit.

Instruction Format:

The following are some common instruction formats:

- **OPCODE only (zero addresses)**
- **OPCODE + 1 Address (usually a memory address)**
- **OPCODE + 2 Addresses (usually registers, or one register and one memory address)**
- **OPCODE + 3 Addresses (usually registers, or combinations of registers and memory)**

Data Format

Little versus Big Endian: The term endian refers to a computer architecture's "byte order," or the way the computer stores the bytes of a multiple-byte data element.

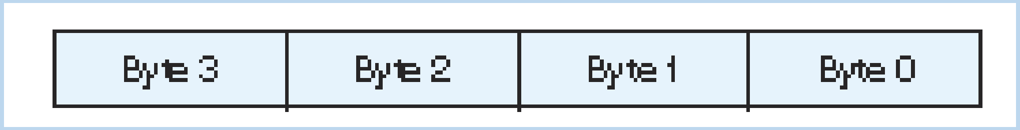
Virtually all computer architectures today are byte-addressable and must, therefore, have a standard for storing information requiring more than a single byte.
Some machines store a two-byte integer, for example, with the least significant byte first (at the lower address) followed by the most significant byte. Therefore, a byte at a lower address has lower significance. These machines are called little endian machines.

Other machines store this same two-byte integer with its most significant byte first, followed by its least significant byte. These are called big endian machines because they store the most significant bytes at the lower addresses.

Most UNIX machines are big endian, whereas most PCs are little endian machines. Most newer RISC architectures are also big endian.

For example, Intel has always done things the "little endian" way whereas Motorola has always done things the "big endian" way.

For example, consider an integer requiring 4 bytes:

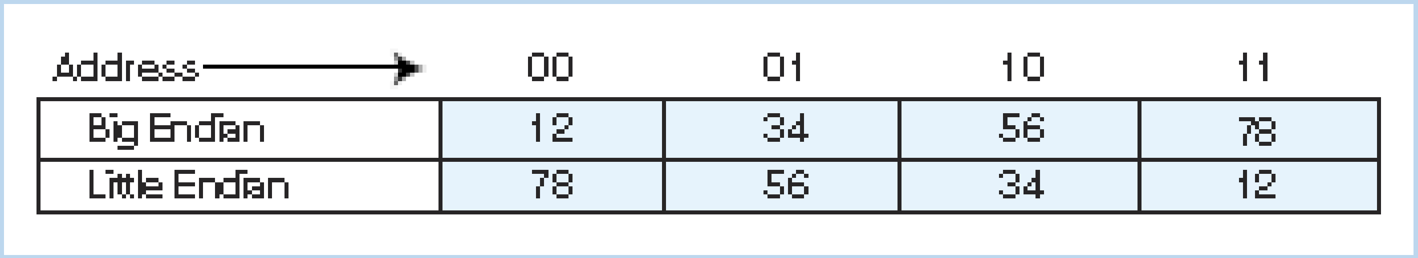


On a little endian machine, this is arranged in memory as follows:

Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3

On a big endian machine, this long integer would then be stored as:
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0

For example: Let's assume that on a byte-addressable machine, the 32-bit hex value 12345678 is stored at address 0.
Each digit requires a nibble, so one byte holds two digits.
This hex value is stored in memory as shown in Figure below, where the shaded cells represent the actual contents of memory.



Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Summery of Instruction Execution Sequence

1. **Fetch next instruction from memory to IR**
2. **Change PC to point to next instruction**
3. **Determine type of instruction just fetched**
4. **If instruction needs data from memory, determine where it is**
5. **Fetch data if needed into register**
6. **Execute instruction**
7. **Go to step 1 & continue with next instruction**

Interrupts

All computers provide a mechanism by which other module (I/O, memory etc.) may interrupt the normal processing of the processor. The most common classes of interrupts are:

- Program: Generated by some condition that occurs as a result of an instruction execution, such as arithmatic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside the user's allowed memory space.
- I/O: Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
- Hardware failure: Generated by a failure such as power failure .

Interupts are provided primarily as a way to improve processing effeciency. For example, most external devices are much slower than the processor. With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

For I/O operation, like printing some information by a printer. Printer is much slower device than the CPU. The CPU puts some information on the output buffer. While printer is busy printing these information from output buffer ,during this time CPU can perform some other task which does not involve the memory bus.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service the particular I/O device (known as an interrupt handler), and resuming the original execution after the device is serviced.

Then with the interrupts ,an interrupt cycle is added to the instruction cycle, which is shown in the figure D .

In the interrupt cycle, the processor checks if any interrupt have occurred, indicated by the presence of an interrupt signals.

If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following :

1. Saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
2. It sets the program counter to the starting address of an interrupt service routine.
   simply branches to its starting location –Calling the subroutine.
3. After a subroutine has been executed, the calling program must resume execution,
   continuing immediately after the instruction that called the subroutine –Returning from subroutine.

Calling – Store the contents of the PC in the link register.
– Branch to the target address specified by the instruction. Returning – Branch to the address contained in the link register

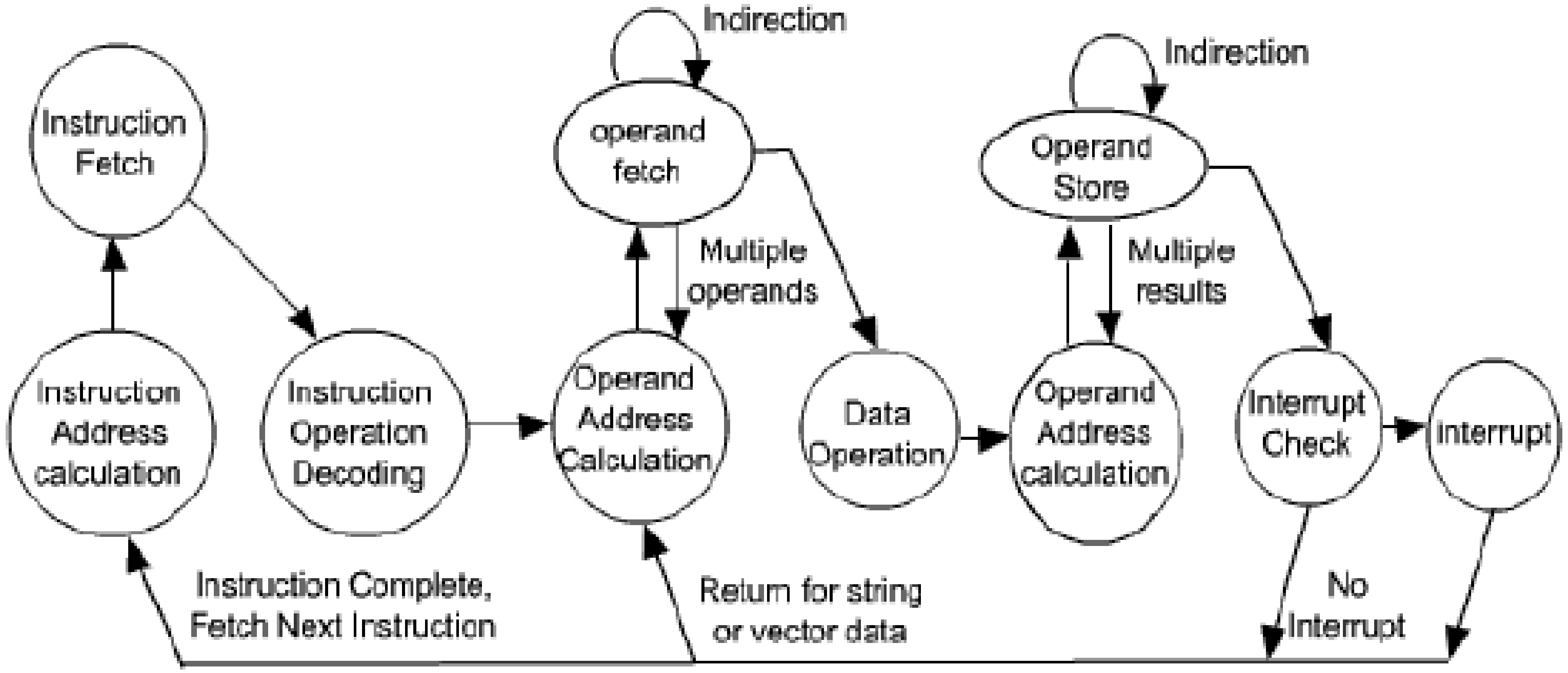
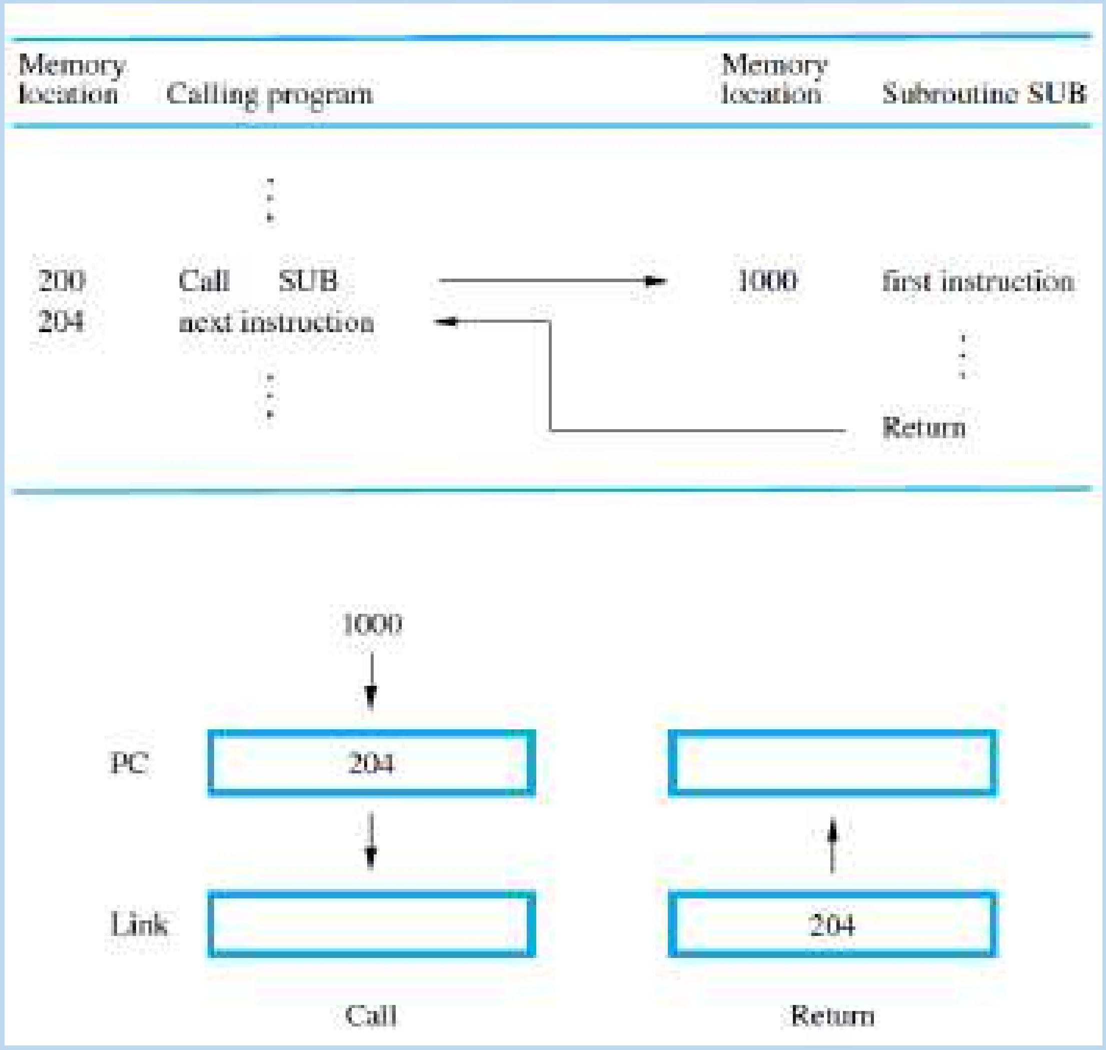

Instruction Fetch
Indirection
operand fetch
Indirection
Operand Store
Multiple operands
Multiple results
Instruction Address calculation
Instruction Operation Decoding
Operand Address Calculation
Data Operation
Operand Address calculation
Interrupt Check
Interrupt
Instruction Complete, Fetch Next Instruction
Return for string or vector data
No Interrupt

Figure D : Instruction cycle state diagram with interrupt.

Calling & Returning Example



Nested subroutine call

One subroutine calls another, which calls another and so on.

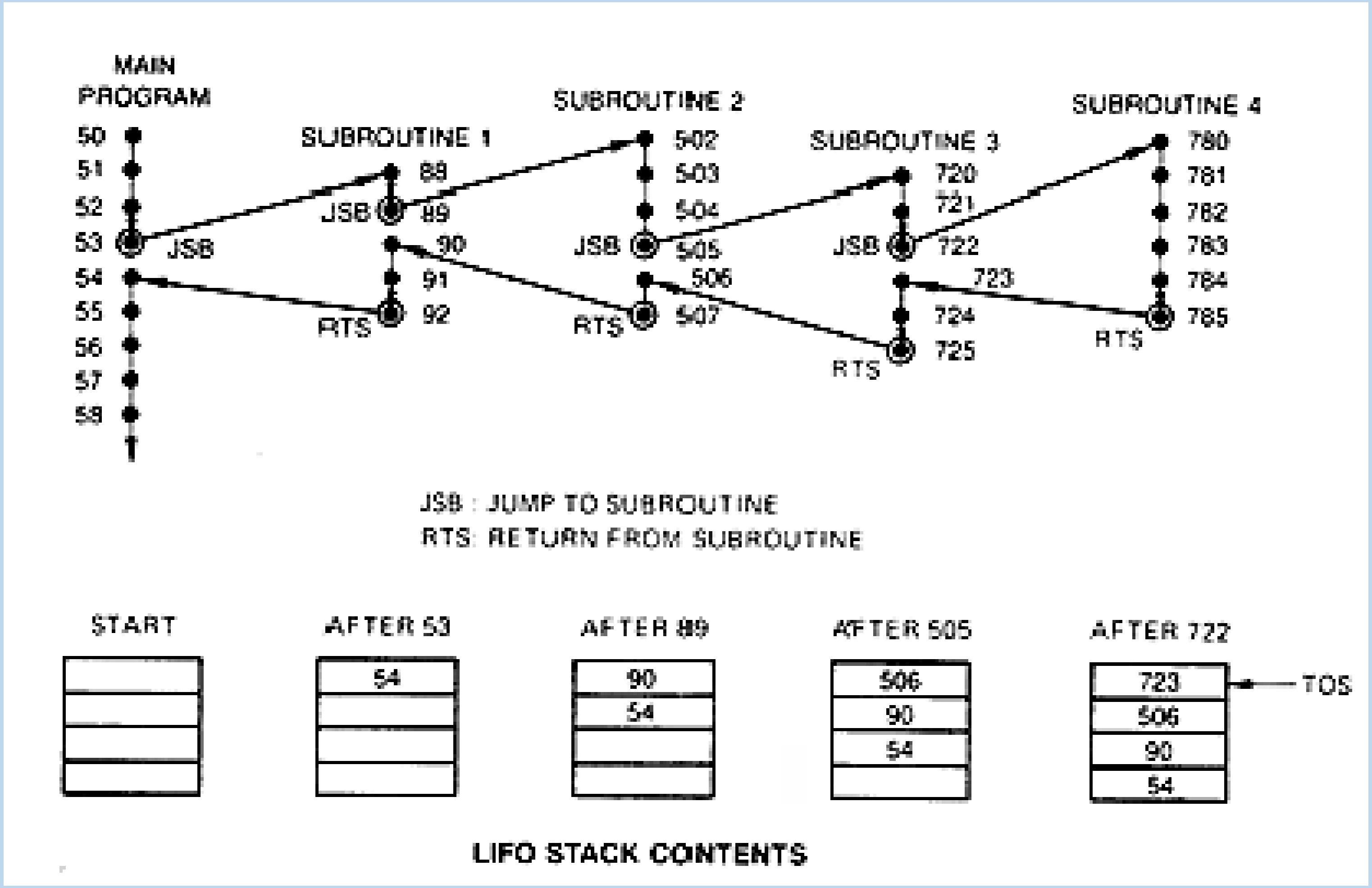
• In this case, the return address of the second call is also stored in the link register, destroying its previous contents.
• Solution – The last subroutine returns first.

– **LIFO**: Use Processor stack for storing return addresses.
• SP points the top of stack.

• The **Call** instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC.

• The **Return** instruction pops the return address from the processor stack into the PC.

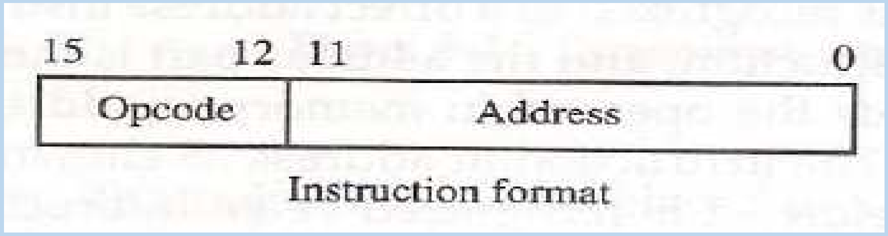
Nested subroutine calls -example

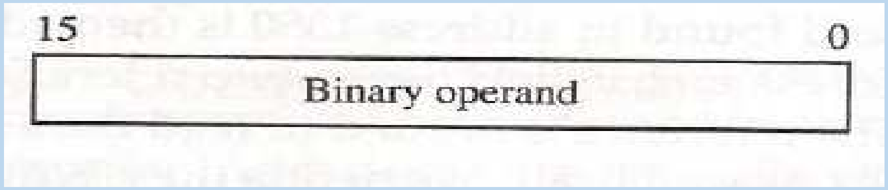


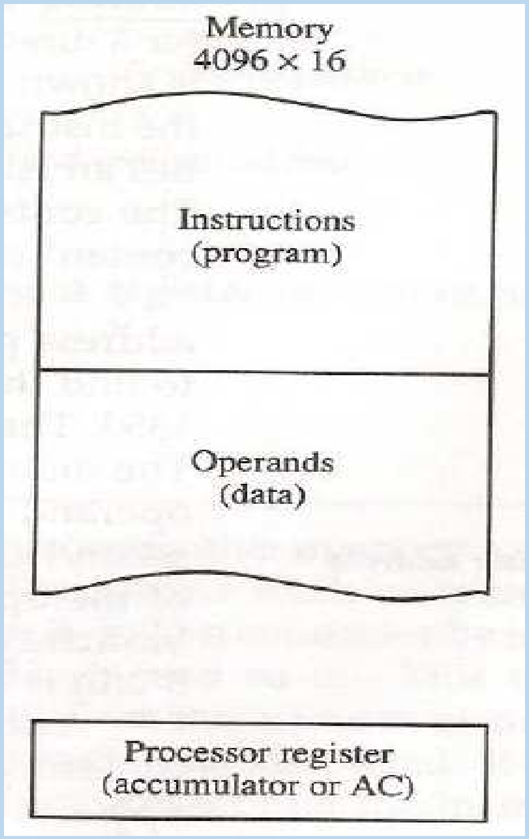
Some examples of Instruction cycle

Example: Given a memory 4096x16 , how will the execution cycle will be to add two numbers stored in memory locations 940, 941 and storing the result at location 941, as shown:

The most four bits of memory location is the opcode, and the rest 12 bits are the specified address(in program segment)

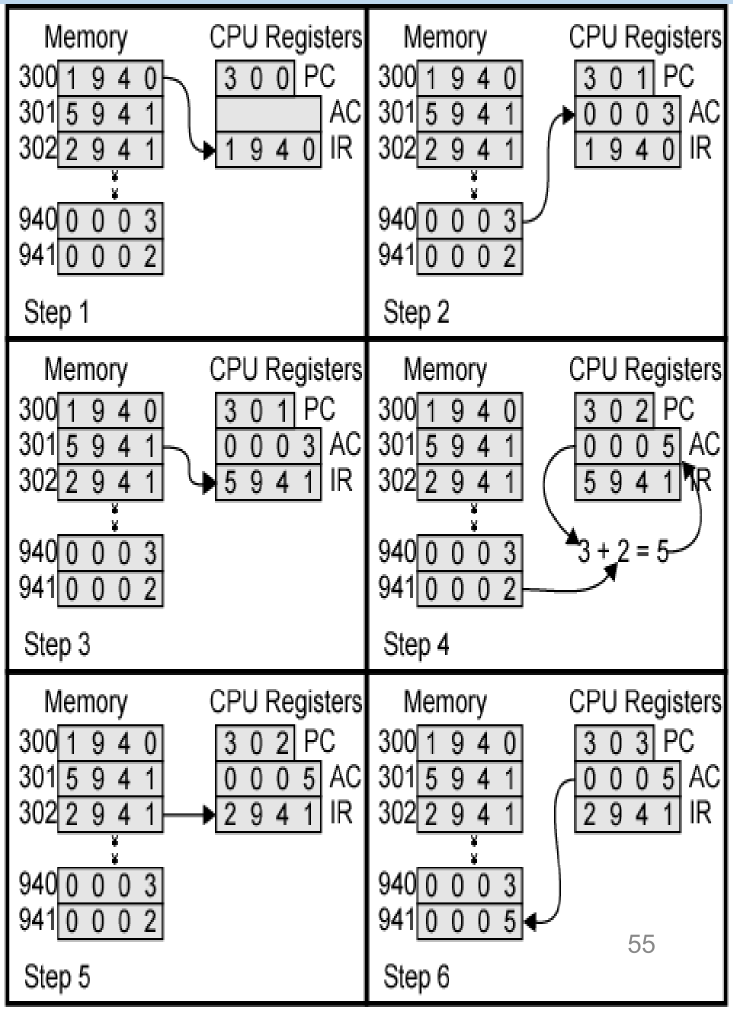






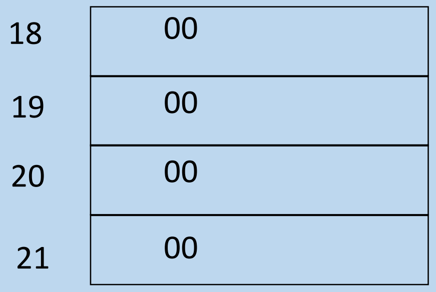
According to the figure, the instruction cycle will be:

1. The PC contains 300, the address of the first instruction. This instruction (1940H) is loaded into IR and the PC is incremented. This process involves the use of MAR and MDR.
2. The MSB 4 bits (1H) in the IR indicate that the AC is to be loaded. The remaining 12 bits specify the address (940H) from which data are to be loaded.
   Load AC, [940], AC $\leftarrow$ 3, end of 1st instruction cycle
3. The next instruction (5941) is fetched and 5H meaning add , and the PC is incremented. PC =302
4. This means , the contents of AC and the contents of location 941 are added , the result is stored in AC. ADD AC,[941], [941]+AC $\longrightarrow$ AC, end of 2nd I. C
5. Last instruction (2941) is fetched and 2 means store and the PC is incremented. AC $\longrightarrow$ 941, The contents of the AC are stored in location 941. end of program

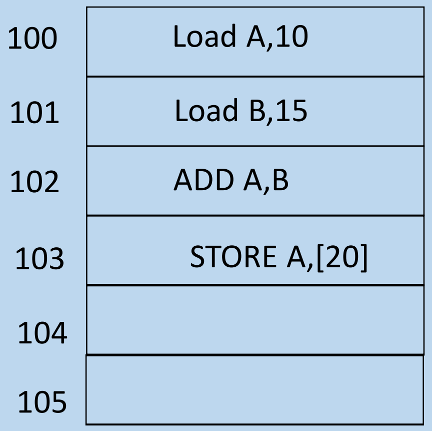


Example: simple program

```
100: Load  A,10
101: Load  B,15
102: Add   A,B
103: STORE A,[20]
```

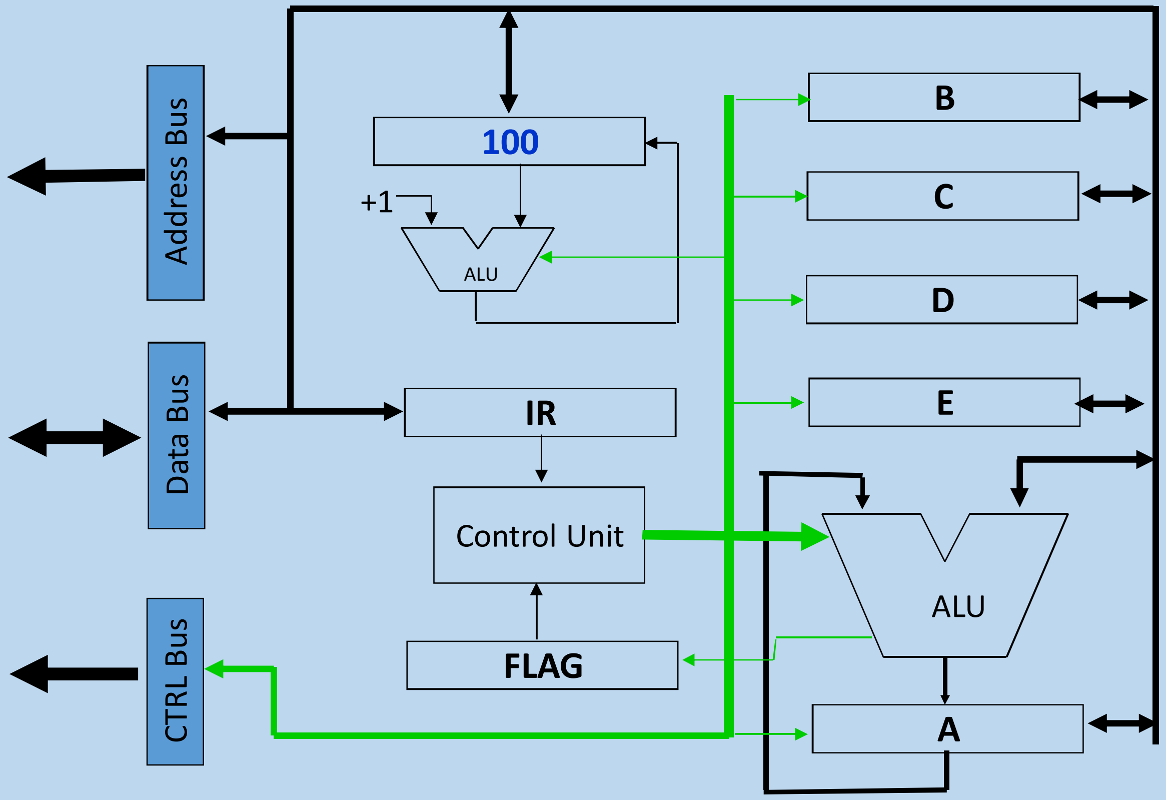


Data memory



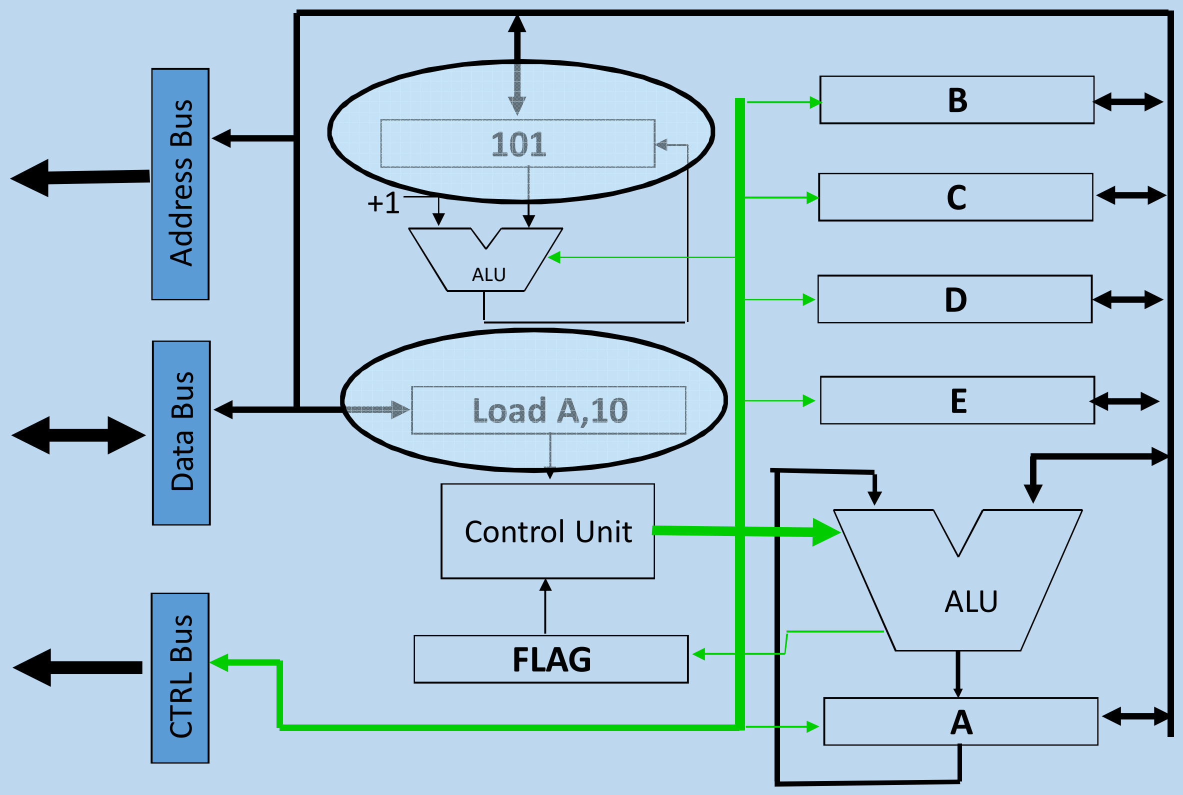
Program memory

Before execution of 1st fetch cycle



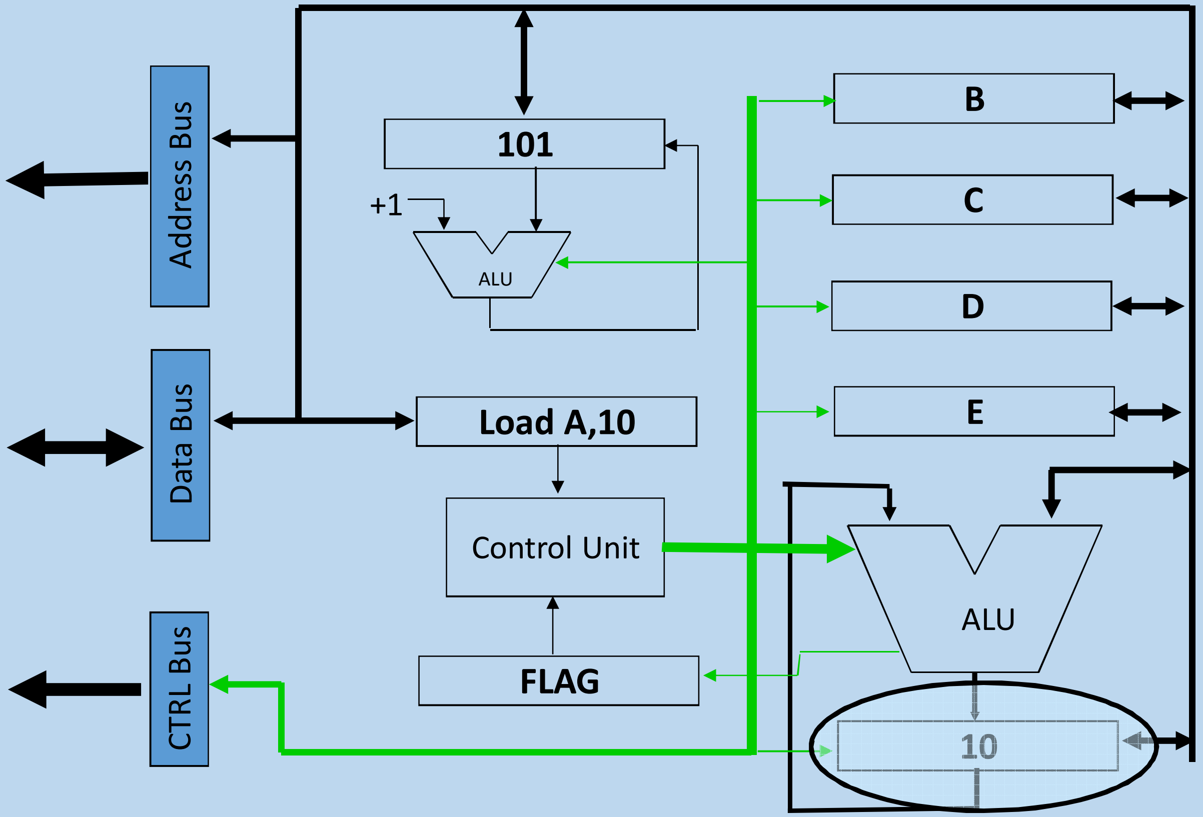
```
100: Load  A,10
```

After 1st fetch cycle …



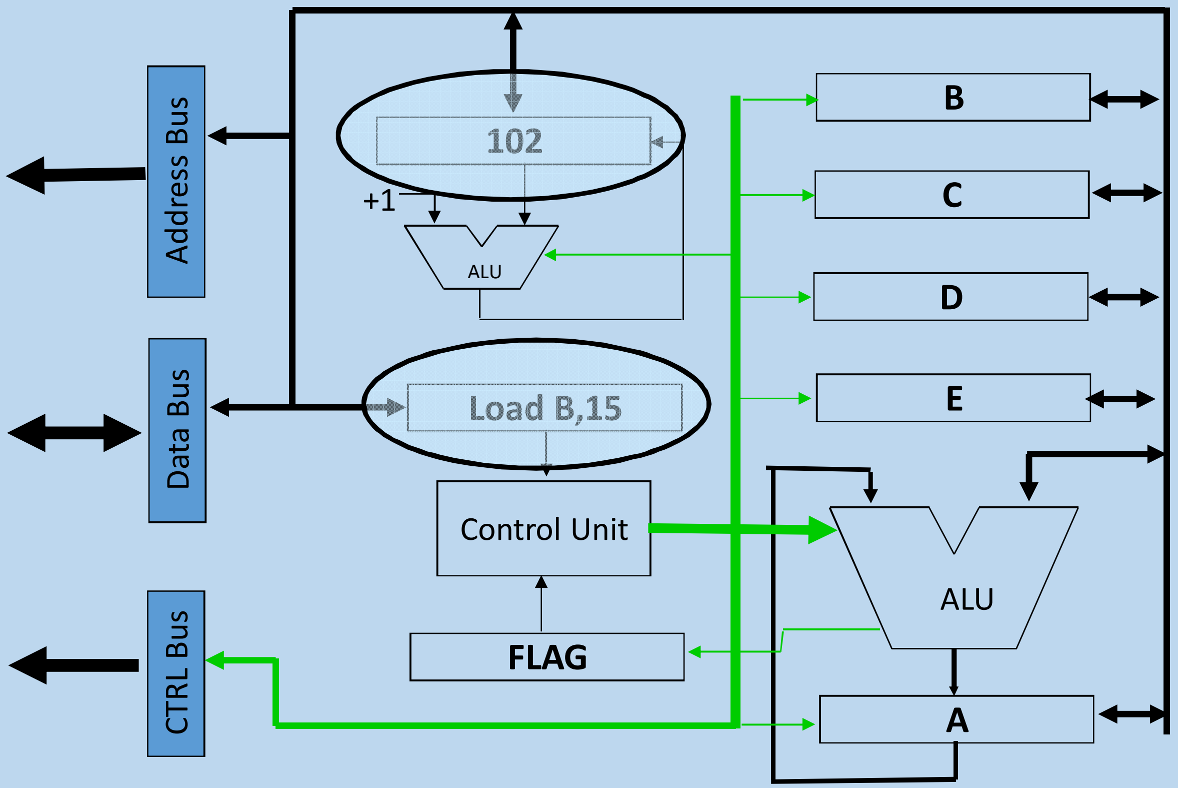
```
100: Load  A,10
```

After 1st instruction cycle …

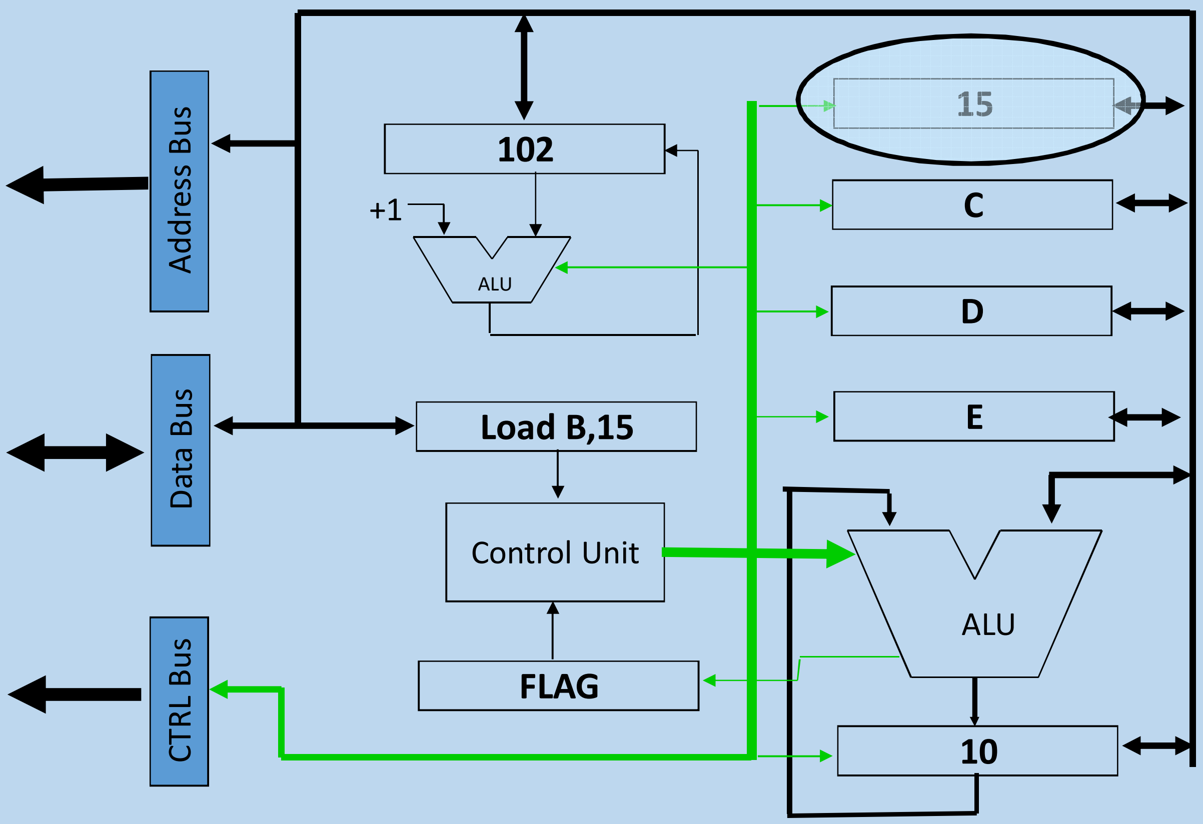


After 2nd fetch cycle …

```
101: Load  B,15
```

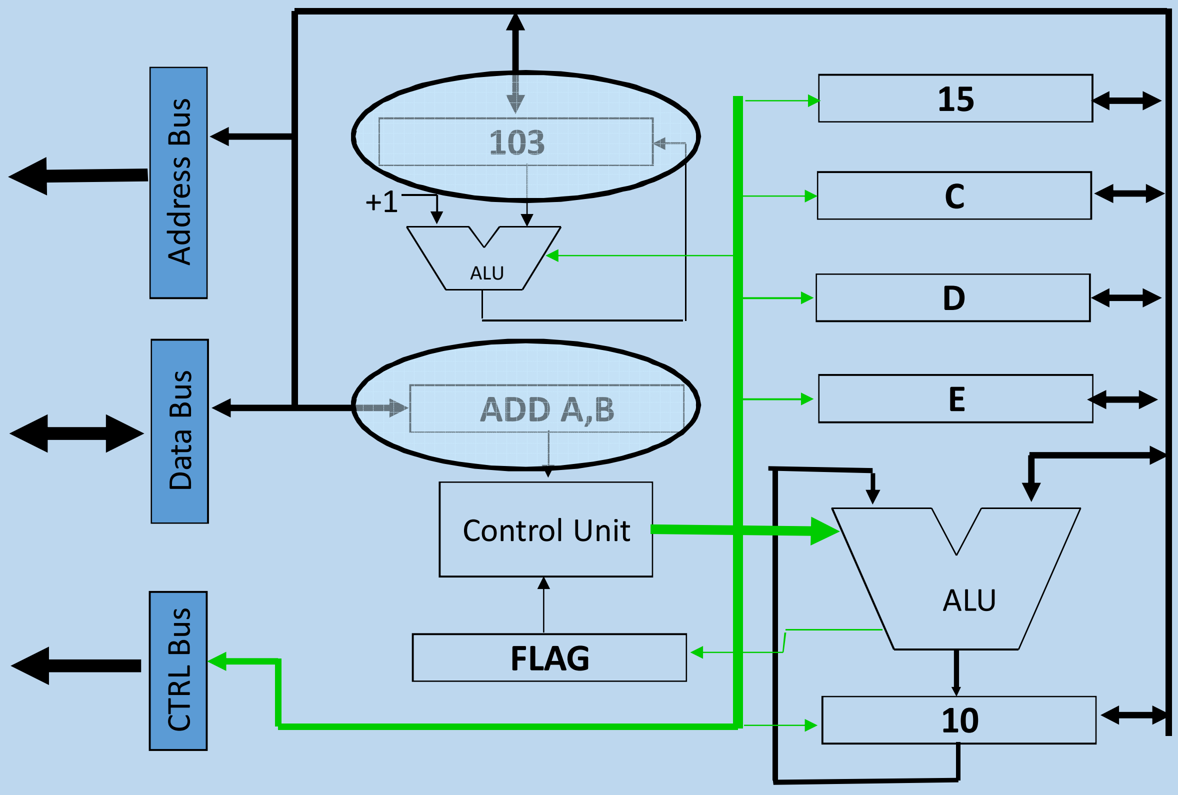


After 2nd instruction cycle …

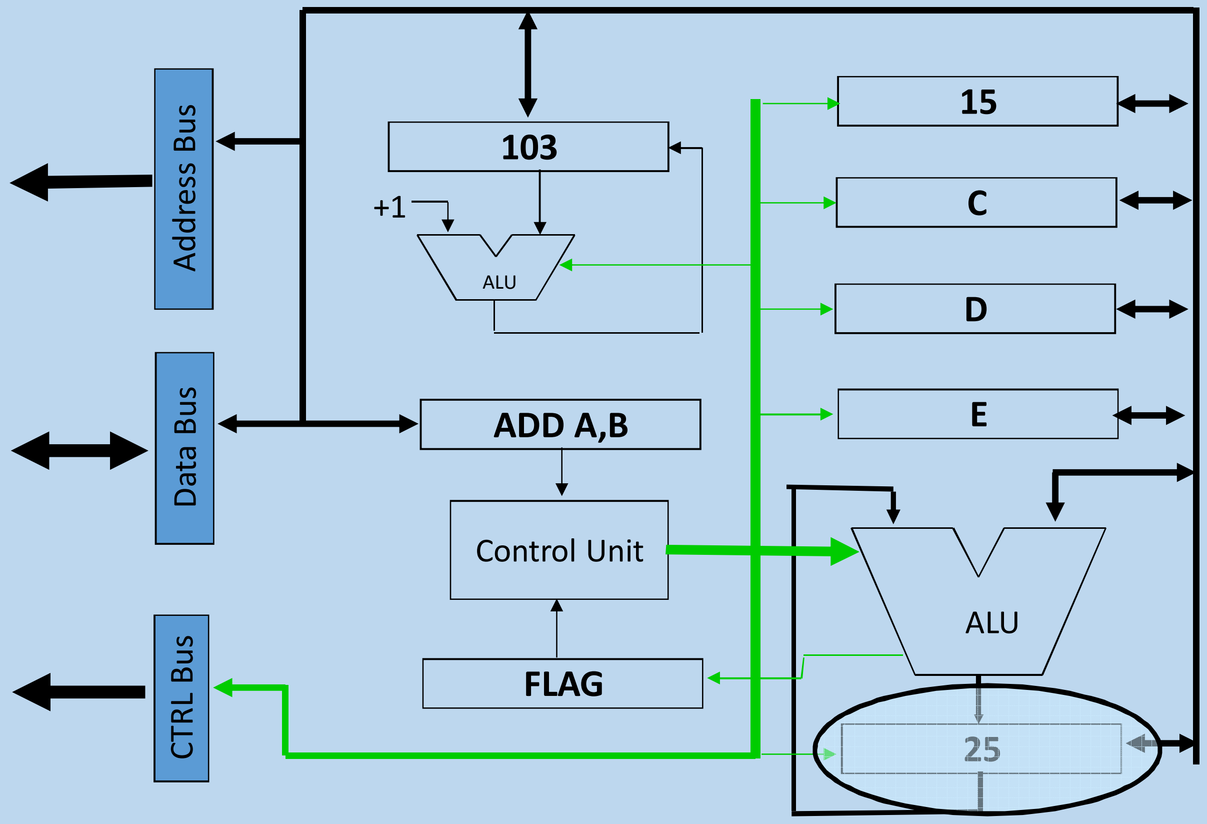


After 3rd fetch cycle …

```
102: Add    A,B
```



After 3rd instruction cycle …



Architectural Differences

- Length of microprocessors' data word
  - 4, 8, 16, 32, 64, & 128 bit
- Speed of instruction execution
  - Clock rate & processor speed
- Size of direct addressable memory
- CPU architecture
- Instruction set
- Number & types of registers
- Support circuits
- Compatibility with existing software & hardware development systems

So the design is not unique , but depends on the choice of the above points