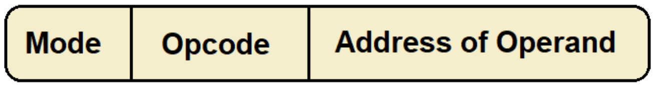
UNIT 4:

Basic computer Organization

What is the Instruction Code?

Instruction codes are bits that instruct the computer to execute a specific operation. An instruction comprises groups called fields. These fields include:
An instruction comprises groups called fields. These fields include:

- The Operation code (Opcode) field determines the process that needs to perform.
- The Address field contains the operand's location, i.e., register or memory location.
- The Mode field specifies how the operand locates.



Instruction Format

Structure of an Instruction Code

The instruction code is also known as an instruction set. It is a collection of binary codes. It represents the operations that a computer processor can perform. The structure of an instruction code can vary. It depends on the architecture of the processor but generally consists of the following parts:

- **Opcode:** The opcode (Operation code) represents the operation that the processor must perform. It might indicate that the instruction is an arithmetic operation such as addition, subtraction, multiplication, or division.

- **Operand(s):** The operand(s) represents the data that the operation must be performed on. This data can take various forms, depending on the processor's architecture. It might be a register containing a value, a memory address pointing to a location in memory where the data is stored, or a constant value embedded within the instruction itself.

- **Addressing mode:** The addressing mode represents how the operand(s) can be interpreted. It might indicate that the operand is a direct address in memory, an indirect address (i.e. a memory address stored in a register), or an immediate value (i.e. a constant embedded within the instruction).

- **Prefixes or modifiers:** Some instruction sets may include additional prefixes or modifiers that modify the behavior of the instruction. For example, they may specify that the operation should be performed only if a specific condition is met or that the instruction should be executed repeatedly until a specific condition is met.

Types of Instruction Code

There are various types of instruction codes. They are classified based on the number of operands, the type of operation performed, and the addressing modes used. The following are some common types of instruction codes:

1. **One-operand instructions:** These instructions have one operand and perform an operation on that operand. For example, the "neg" instruction in the x86 assembly language negates the value of a single operand.

2. **Two-operand instructions:** These instructions have two operands and perform an operation involving both. For example, the "add" instruction in x86 assembly language adds two operands together.

3. **Three-operand instructions:** These instructions have three operands and perform an operation that involves all three operands. For example, the "fma" (fused multiply-add) instruction in some processors multiplies two operands together, adds a third operand, and stores the result in a fourth operand.

4. **Data transfer instructions:** These instructions move data between memory and registers or between registers. For example, the "mov" instruction in the x86 assembly language moves data from one location to another.

5. **Control transfer instructions:** These instructions change the flow of program execution by modifying the program counter. For example, the "jmp" instruction in the x86 assembly language jumps to a different location in the program.

6. **Arithmetic instructions:** These instructions perform mathematical operations on operands. For example, the "add" instruction in x86 assembly language adds two operands together.

7. **Logical instructions:** These instructions perform logical operations on operands. For example, the "and" instruction in x86 assembly language performs a bitwise AND operation on two operands.

8. **Comparison instructions:** These instructions compare two operands and set a flag based on the result. For example, the "cmp" instruction in x86 assembly language compares two operands and sets a flag indicating whether they are equal, greater than, or less than.

9. **Floating-point instructions:** These instructions perform arithmetic and other operations on floating-point numbers. For example, the "fadd" instruction in the x86 assembly language adds two floating-point numbers together.

Opcodes

An opcode is a collection of bits representing the basic operations, including add, subtract, multiply, complement, and shift. The number of bits required for the opcode is determined by the number of functions the computer gives. For ‘2n’ operations, the minimum bits accessible to the opcode should be ‘n’, where n is the number of bits. We implement these operations on information saved in processor registers or memory.

Types of Opcodes

There are three different types of instruction codes on the main computer. The instruction's operation code (opcode) is 3 bits long, and the remaining 13 bits are determined by the operation code encountered.

There are three types of formats:

1. Memory Reference Instruction: It specifies the address with 12 bits and the addressing mode with 1 bit (I). For direct addresses, I equal 0, while for indirect addresses, I equal 1.
2. Register Reference Instruction: The opcode 111 with a 0 in the leftmost bit of the instruction recognizes these instructions. The remaining 12 bits specify the procedure to be carried out.
3. Input-Output Instruction: The operation code 111 with a 1 in the leftmost bit of instruction recognizes these instructions. The input-output action is specified using the remaining 12 bits.

Address

We represent the memory address where a given instruction is built. We use an instruction code's address bits as an operand rather than an address. The instruction in such methods has an immediate operand. The command is directed to a direct address if the second portion contains an address.

In the second half, there is another choice, which includes the operand's address. It points to an oblique address. One bit might indicate whether the instruction code executes the direct or indirect address.

Addressing Modes

We can mainly describe the address field for instruction in the following ways:

- **Direct Addressing** − Uses the address of the operand.
- **Indirect Addressing** − Enables the address as a pointer to the operand.
- **Immediate operand** − The second part of the instruction code specifies an operand.

Accumulator Register (AC): This register is found in single register processors (AC), and it performs all operations with memory operands.

Effective Address (EA) is the address of the operand or the target address. It defines the address that we can execute as a target address for a branch-type instruction or the address we can use directly to create an operand for a computation-type instruction without any changes.

Types of Addressing Modes

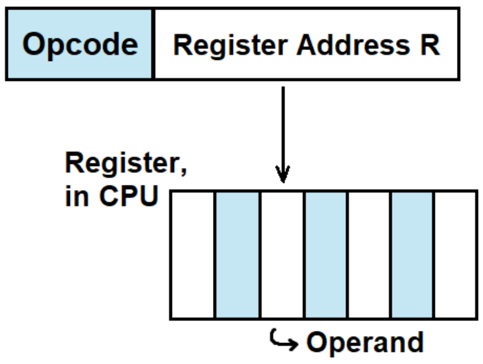
We have discussed below the different types of addressing modes:

Immediate Mode

An immediate mode instruction specifies the operand in the instruction itself rather than the address field. For example, the expression “ADD 100” adds 100 to the contents of the accumulator. 100 here is the operand.

Register Mode

We store operands in the register in the CPU, and instruction addresses are the addresses of the registers in which operands are stored.



Advantages of Addressing Modes

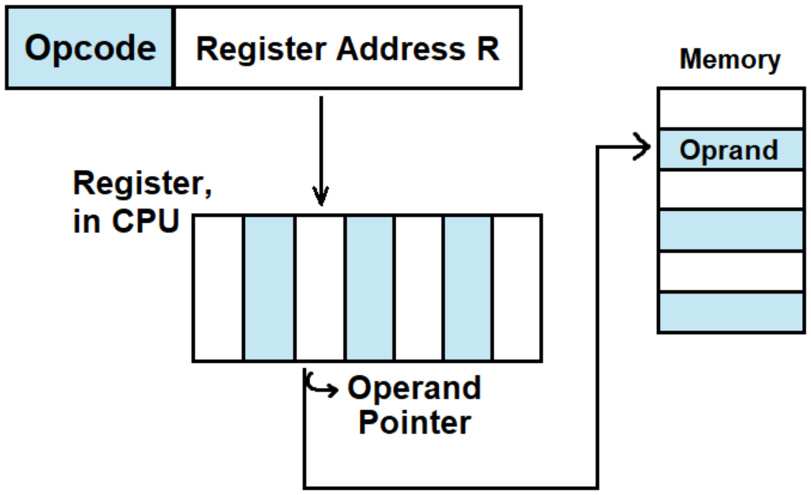
- Shorter instructions and faster instruction fetch.
- Speedier memory access to the operand(s).

Disadvantages of Addressing Modes

- Minimal address space.
- Using multiple registers allows performance, but it complicates the instructions.

Register Indirect Mode

A register that specifies the address of an operand located in memory is used in this mode. Thus, the register possesses the address of the operand rather than the operand itself.

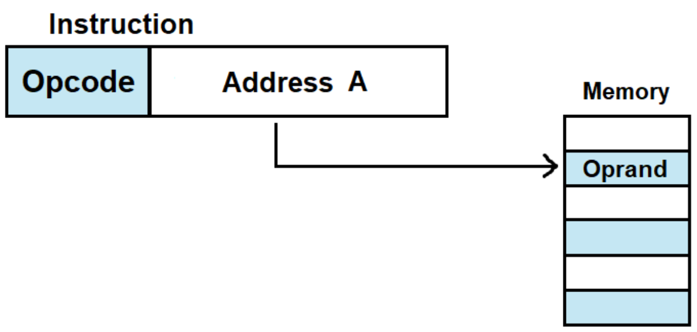


Auto Increment/Decrement Mode

The register increments or decrements after or before it uses the value.

Direct Addressing Mode

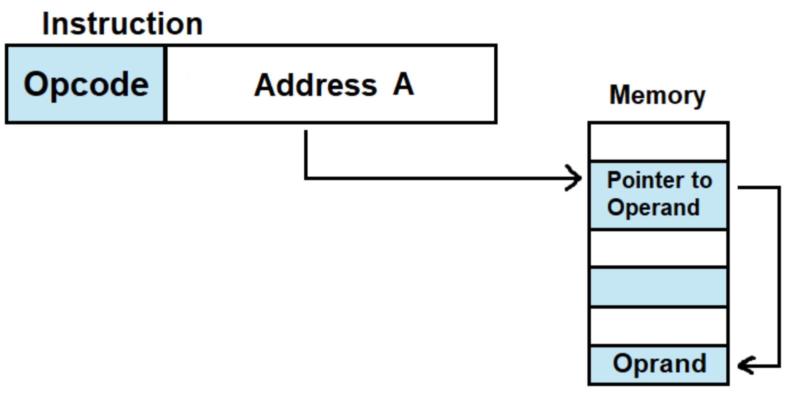
In this, we include the operand's effective address in the instruction.
Data is accessed using a single memory reference.
There are no extra calculations required to determine the operand's effective address.



For example, in ADD C2, 1000, the 1000 represents the operand's effective address.
NOTE: The operand's effective address is where we found it.

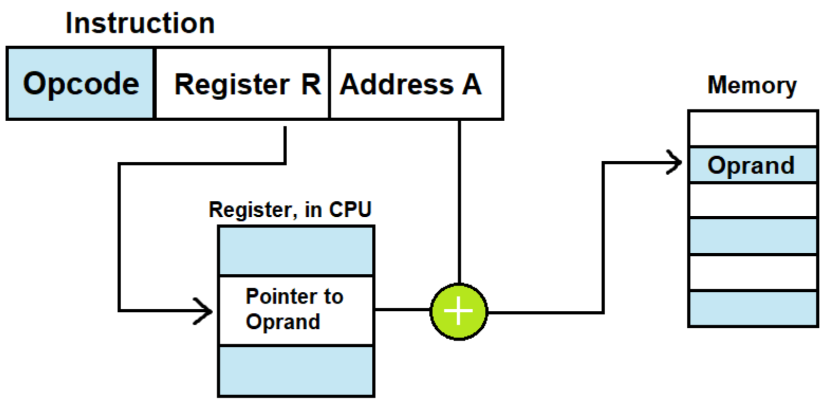
Indirect Addressing Mode

The address component of the instruction defines where the effective address is stored in memory because several memory lookups are required to locate the operand, slowing down the execution.



Displacement Addressing Mode

To acquire the effective address of the operand, we append the contents of the indexed register to the Address section of the instruction.
EA = A + (R), where the address field contains two values: A (the base value) and R (the Displacement), or vice versa.



Relative Addressing Mode

It's a Displacement addressing mode variant. PC (Program Counter) contents are added to the instruction's address component to acquire the effective address.
EA = A + (PC), where PC is the program counter, and EA is the effective address.
A cell apart from the current cell is the operand (the one pointed to by PC).

Base Register Addressing Mode

It's a variation of Displacement addressing mode.
EA = A + (R), where A is the displacement and R is the pointer to the base address.

Stack Addressing Mode

In this addressing mode, the operand is at the top of the stack. For example, ADD; this instruction will POP the top two items off the stack, add them, and then PUSH the result to the top of the stack. This is how ADD instruction works.

Timing and Control signal

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

1. hardwired control and
2. microprogrammed control.

In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed.

In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

The block diagram of the control unit is shown in Fig. 5.6.

It consists of two decoders,

1. a sequence counter, and
2. a number of control logic gates.

An instruction read from memory is placed in the instruction register (IR).position of this register in the common bus system is indicated in Fig 5.4.

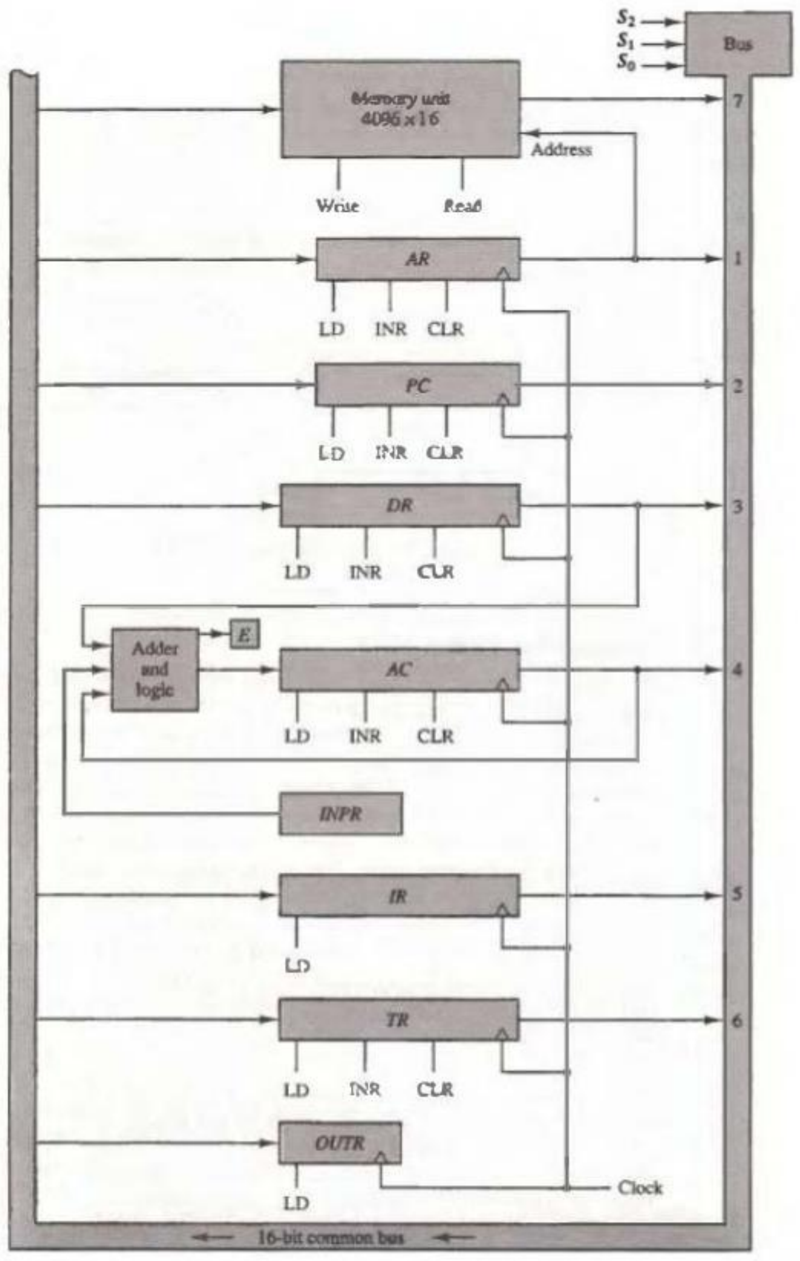


Figure 5-4 Basic computer registers connected to a common bus.

The instruction register is shown again in Fig. 5.6, where it is divided into three parts:

1. the 1 bit,
2. the operation code, and
3. bits 0 through 11.

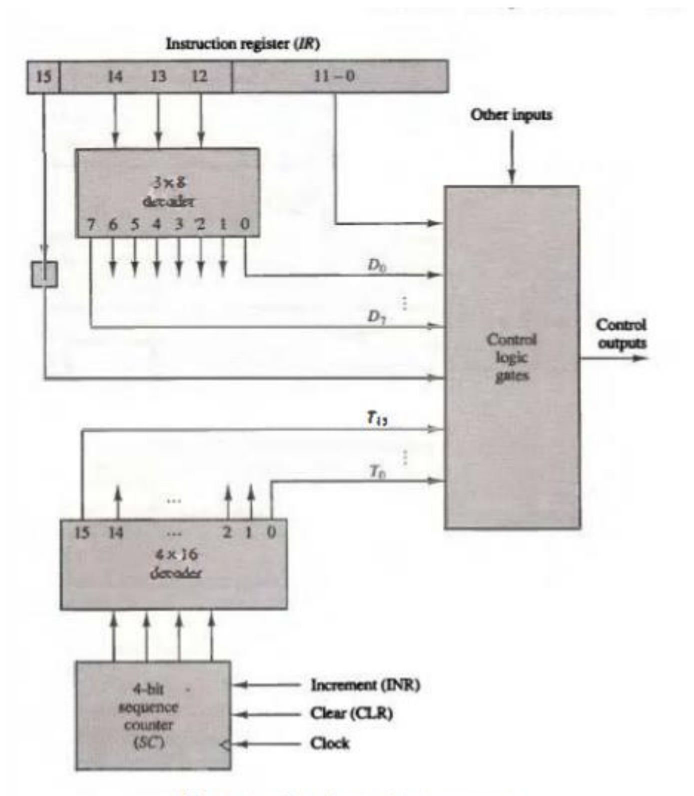


Figure 5-6 Control unit of basic computer.

The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be T_0.

As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

The timing diagram of Fig. 5-7 shows the time relationship of the control signals.

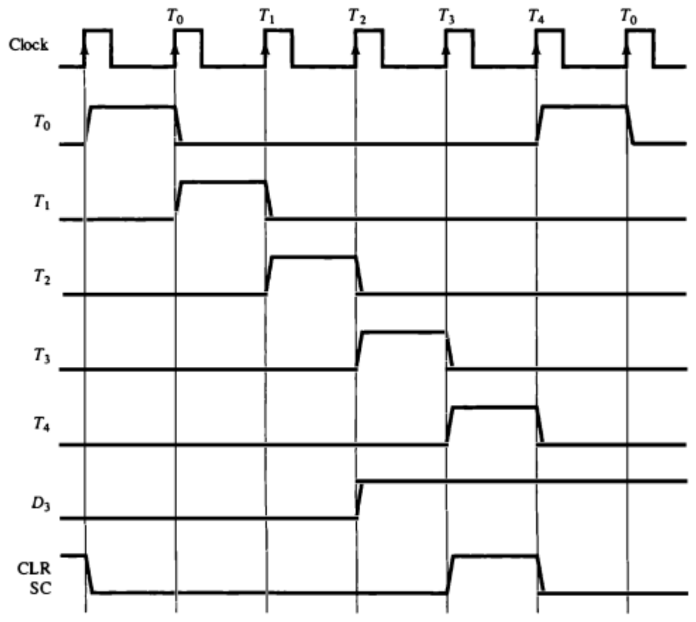


Figure 5-7 Example of control timing signals.

The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in tum activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the dagram will trigger only those registers whose control inputs are transition, to timing signal T_0. SC is incremented with every positive clock transition unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3 ,T_4 and so on, as shown in the dagram. (Note the the relationshuip between the timing signal and and its corresponding positive clock transition.) If SC is not cleared, the timing signals will continue with T5, T_6 up to T_{15} and back to T_0

The last three waveforms in Fig. 5-7 show how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle ini tiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0: AR \leftarrow PC$$

specifies a transfer of the content of PC into AR if timing signal T_0 is active. T_0 is active during an entire clock cycle intervaL During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001 . The next clock cycle has T_1 active and T_0 inactive.

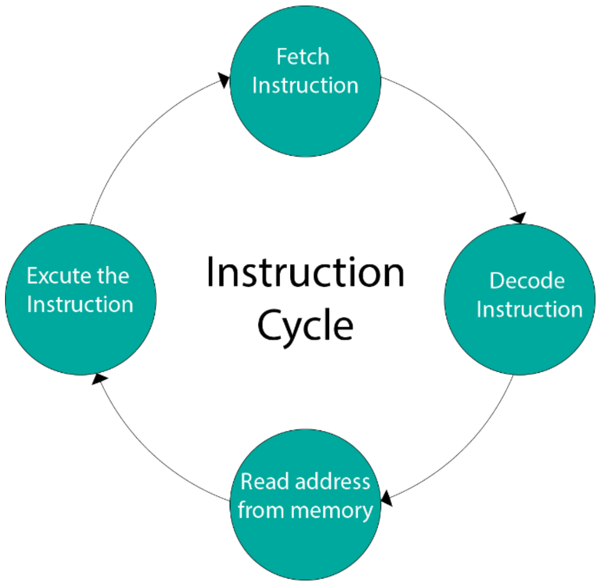
Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

In a basic computer, each instruction cycle consists of the following phases:

1. Fetch instruction from memory.
2. Decode the instruction.

3. Read the effective address from memory.
4. Execute the instruction.

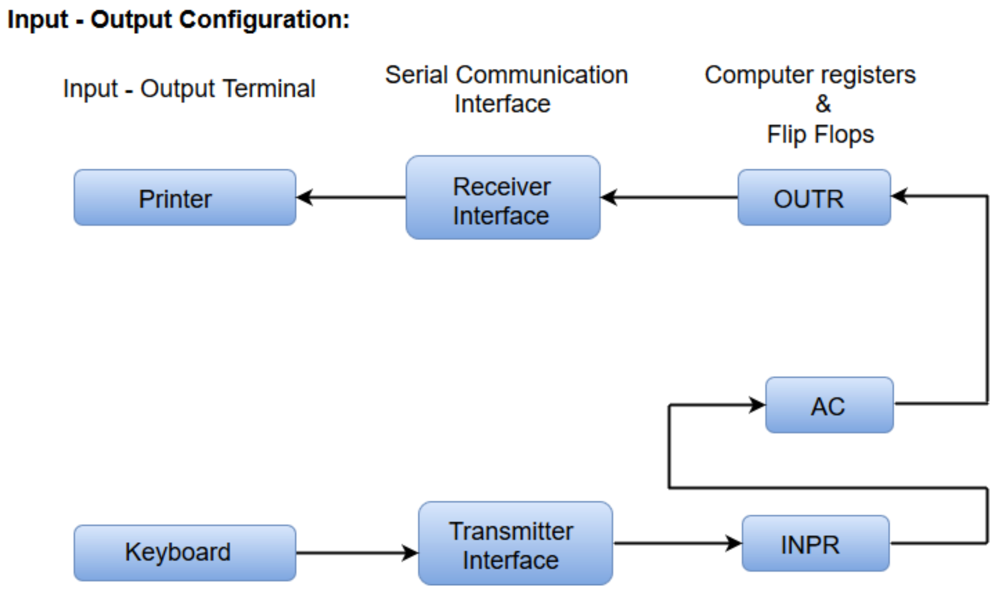


Input-Output Configuration

In computer architecture, input-output devices act as an interface between the machine and the user.

Instructions and data stored in the memory must come from some input device. The results are displayed to the user through some output device.

The following block diagram shows the input-output configuration for a basic computer.


Input - Output Configuration:
Input - Output Terminal
Serial Communication Interface
Computer registers & Flip Flops
Printer
Receiver Interface
OUTR
AC
Keyboard
Transmitter Interface
INPR

- The input-output terminals send and receive information.
- The amount of information transferred will always have eight bits of an alphanumeric code.
- The information generated through the keyboard is shifted into an input register 'INPR'.
- The information for the printer is stored in the output register 'OUTR'.
- Registers INPR and OUTR communicate with a communication interface serially and with the AC in parallel.
- The transmitter interface receives information from the keyboard and transmits it to INPR.
- The receiver interface receives information from OUTR and sends it to the printer serially.

Design of a Basic Computer

A basic computer consists of the following hardware components.

1. A memory unit with 4096 words of 16 bits each
2. Registers: AC (Accumulator), DR (Data register), AR (Address register), IR (Instruction register), PC (Program counter), TR (Temporary register), SC (Sequence Counter), INPR (Input register), and OUTR (Output register).
3. Flip-Flops: I, S, E, R, IEN, FGI and FGO

Note: FGI and FGO are corresponding input and output flags which are considered as control flip-flops.

1. Two decoders: a 3 x 8 operation decoder and 4 x 16 timing decoder
2. A 16-bit common bus
3. Control Logic Gates
4. The Logic and Adder circuits connected to the input of AC.

What are Memory Reference Instructions?

Memory reference instructions are those commands or instructions which are in the custom to generate a reference to the memory and approval to a program to have an approach to the commanded information and that states as to from where the data is cache continually. These instructions are known as Memory Reference Instructions.

There are seven memory reference instructions which are as follows &

AND

The AND instruction implements the AND logic operation on the bit collection from the register and the memory word that is determined by the effective address. The result of this operation is moved back to the register.

ADD

The ADD instruction adds the content of the memory word that is denoted by the effective address to the value of the register.

LDA

The LDA instruction shares the memory word denoted by the effective address to the register.

STA

STA saves the content of the register into the memory word that is defined by the effective address. The output is next used to the common bus and the data input is linked to the bus. It needed only one micro-operation.

BUN

The Branch Unconditionally (BUN) instruction can send the instruction that is determined by the effective address. They understand that the address of the next instruction to be performed is held by the PC and it should be incremented by one to receive the address of the next instruction in the sequence. If the control needs to implement multiple instructions that are not next in the sequence, it can execute the BUN instruction.

BSA

BSA stands for Branch and Save return Address. These instructions can branch a part of the program (known as subroutine or procedure). When this instruction is performed, BSA will store the address of the next instruction from the PC into a memory location that is determined by the effective address.

ISZ

The Increment if Zero (ISZ) instruction increments the word determined by effective address. If the incremented cost is zero, thus PC is incremented by 1. A negative value is saved in the memory word through the programmer. It can influence the zero value after getting incremented repeatedly. Thus, the PC is incremented and the next instruction is skipped.

Input/Output Instruction

The I/O devices are given specific addresses. The processor similarly views the I/O operations as memory operations. It concerns commands that include the address for the device.

The I/O instructions are needed for the following objectives −

- It is used to analyzing flag bits.
- It can transfer data to or from the AC register.
- It can controlling interrupts.

The I/O instructions carry the opcode as 1111. They are recognized through the control when D_7 = 1 and I=1. The operation to be executed is determined by the different remaining bits.

There are various I/O Instructions which are shown in the table −

Symbol	**Description**
INP	The INP instruction address the information from the INPR to AC which has 8 low order bits. It also clears the input flag to 0.
OUT	It can send the 8 low order bits from AC into output register OUTPR. It also clears the output flag to 0.
SKI	These are the status flags. They skip the next instructions when flag = 1, They are primarily branching instructions.
SKO	It is similar to SKI.
ION	Enables (set) interrupt.
IOF	Disables (clear) interrupt.

What is General Register Organization?

Introduction of General Register Organization

General Register Organization is the processor architecture that stores and manipulates data for computations. The main components of a register organization include registers, memory, and instructions. The registers act as memory within the processor and are used to process instructions as they are executed. Core concepts such as memory addressing modes, program counter usage, instruction decoders, and instruction sets are all part of the general register organization.

Registers themselves are composed of several parts. They are typically divided into **GeneralPurpose Registers (GPRs)** and **SpecialPurpose Registers (SPRs)**. GPRs can be used for any data manipulation or input/output operations. At the same time, SPRs are reserved for specific purposes, such as providing locations for storing intermediate results or jump destinations in programs or accessing data from memory.

Memory addressing modes allow us to reference different parts of memory depending on our needs. Program counters let us keep track of where we are in a program as it is executed so that the correct instructions can be fetched when needed. Instruction decoders determine what type of instruction has been fetched and what actions must be taken to carry out those operations. Lastly, instruction sets provide the language through which a processor interprets code written in assembly language or any high-level language compiled into assembly language.

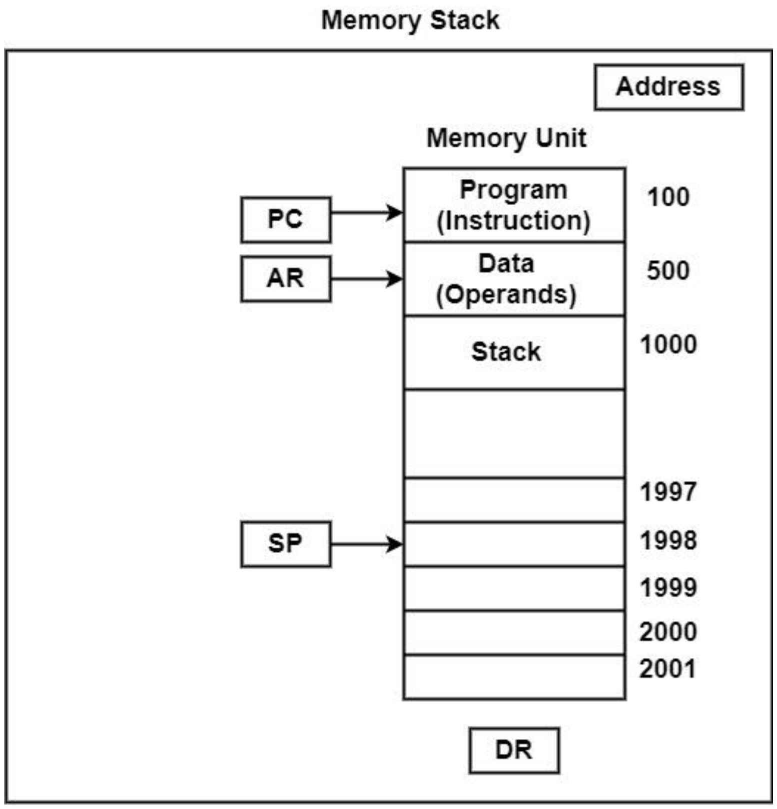
By understanding general register organization components such as registers, memory addressing modes, program counters, instruction decoders, and instruction sets, one can understand how data is stored within a processor, along with how instructions get parsed and executed by that same processor.

Memory Stack

A stack can be executed in the CPU by analyzing an area of the computer memory to a stack operation and utilizing a processor register as a stack pointer. In this method, it is performed in a random access memory connected to the CPU.

An area of the computer memory is broken into three segments such as program, data, and stack. The address of the next instruction in the program is saved in the pointer Program Counter (PC). The Address Register (AR) points to an array of the information. SP continually influences the address of the element present at the top of the stack.

The three registers that are linked to the common bus are PC, AR, and SP. PC can read the instruction during the fetch stage. An operand is read during execute stage using the address register. An element is pushed into or popped from the stack using a stack pointer.



In the figure, the SP points to a beginning value ‘2001’. Therefore, the stack increase with decreasing addresses. The first element is saved at address 2000, the next element is saved at address 1999 and the last element is saved at address 1000.

The data register can read an element into or from the stack. It can use push operation to insert a new element into the stack.

SP ← SP – 1

K ← SP [DR]

It can insert another element into the stack, the stack pointer is decremented by 1. It can point to the address of the next location/word. A word from DR is inserted into the top of the stack using memory write operation.

It can delete an element from the stack. It can use the pop operation which is as follows −

DR ← K [SP]

SP ← SP + 1

The top element is read into the DR and then the stack pointer is decremented to point to the next element in the stack.

Two processor registers can check the stack limits. One processor register influence the upper limit (1000) and the other influence the lower limit (2001). During push operation, the SP is compared with the upper limit to check if the stack is full. During pop operation, the SP is compared with the lower limit to check if the stack is empty.

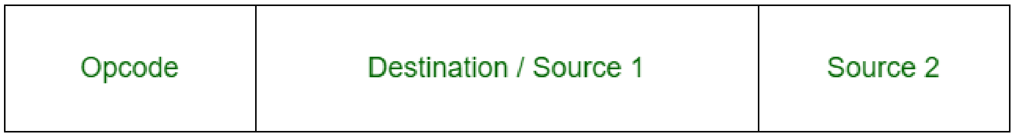
2-address instruction and 1-address instructions

1. Two-Address Instructions : Two-address instruction is a format of machine instruction. It has one opcode and two address fields. One address field is common and can be used for either destination or source and other address field for source.
In computer architecture and assembly language programming, 2-address and 1-address instructions refer to the number of memory operands or memory locations accessed by the instruction. Here are the key differences between 2-address and 1-address instructions:

Number of memory operands: A 2-address instruction involves two memory operands: one source operand and one destination operand. In contrast, a 1-address instruction involves only one memory operand: the operand that is the source or destination of the instruction.

1. Use of registers: 2-address instructions typically require more registers than 1-address instructions. This is because 2-address instructions have both a source and destination operand, which may require additional registers to hold intermediate values or to perform calculations.
2. Code size: 1-address instructions are generally more compact than 2-address instructions, as they require only one memory operand and fewer registers.
3. Flexibility: 2-address instructions are more flexible than 1-address instructions, as they allow for more complex calculations and expressions to be performed. This is because 2-address instructions have both a source and destination operand, which can be used to hold intermediate values or to perform calculations.
4. Execution time: 1-address instructions are generally faster than 2-address instructions, as they require fewer memory accesses and fewer register operations.

In summary, 2-address instructions are more flexible and powerful but require more memory operands and registers, while 1-address instructions are more compact and faster but offer less flexibility. The choice of instruction format depends on the specific requirements of the program, the hardware architecture, and the tradeoffs between code size, execution time, and flexibility.



2-address Instruction Format

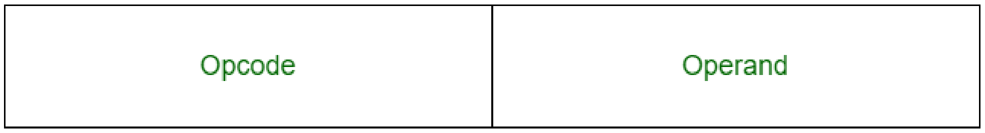
Example:

```
X = (A + B) x (C + D)
```

Solution:

```
MOV R1, A       R1 <- M[A]

ADD R1, B       R1 <- R1 + M[B]

MOV R2, C       R2 <- M[C]

ADD R2, D       R2 <- R2 + D

MUL R1, R2      R1 <- R1 x R2

MOV X, R1       M[X] <- R1
```

2. One-Address Instructions : One-Address instruction is also a format of machine instruction. It has only two fields. One for opcode and other for operand.



1-address Instruction Format

Example:

```
X = (A + B) x (C + D)
```

Solution:

```
LOAD A      AC <- M[A]

ADD B       AC <- AC + M[B]

STORE T     M[T] <- AC

LOAD C      AC <- M[C]

ADD D       AC <- AC + M[D]

MUL T       AC <- AC x M[T]

STORE X     M[X] <- AC
```

Difference between Two-Address Instruction and One-Address Instruction :

TWO-ADDRESS INSTRUCTION	ONE-ADDRESS INSTRUCTION
It has three fields.	It has only two fields.
It has one field for opcode and two fields for address.	It has one field for opcode and one field for address.
It has long instruction length as compared to one-address.	While it has shorter instruction length.
It is slower accessing location inside processor than memory.	It is faster accessing location inside processor than memory.
It generally needs two memory accesses.	It generally needs one memory accesses.
There may be three memory accesses needed for an instruction.	There is a single memory access needed for an instruction.
It can't completely eliminate three memory access.	It eliminates two memory access completely.

Hardware interrupts

The interrupt signal generated from external devices and i/o devices are made interrupt to CPU when the instructions are ready.

For example − In a keyboard if we press a key to do some action this pressing of the keyboard generates a signal that is given to the processor to do action, such interrupts are called hardware interrupts.

Hardware interrupts are classified into two types which are as follows −

- **Maskable Interrupt** − The hardware interrupts that can be delayed when a highest priority interrupt has occurred to the processor.
- **Non Maskable Interrupt** − The hardware that cannot be delayed and immediately be serviced by the processor.

Software interrupts

The interrupt signal generated from internal devices and software programs need to access any system call then software interrupts are present.

Software interrupt is divided into two types. They are as follows −

- **Normal Interrupts** − The interrupts that are caused by the software instructions are called software instructions.
- **Exception** − Exception is nothing but an unplanned interruption while executing a program. For example − while executing a program if we got a value that is divided by zero is called an exception.