Department of Electronics and Communication Engineeing
Faculty of Engineering & Technology
University of Lucknow

Notes of

Digital Electronics and Computer Organization
BCA-203

for

BCA
2nd sem(1st year)

Topic

Introduction to CPU

The operation or task that must perform by CPU is:

- ***Fetch Instruction:*** The CPU reads an instruction from memory.
- ***Interpret Instruction:*** The instruction is decoded to determine what action is required.
- ***Fetch Data:*** The execution of an instruction may require reading data from memory or I/O module.
- ***Process data:*** The execution of an instruction may require performing some arithmetic or logical operation on data.
- ***Write 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 being executed. In other words, the CPU needs a small internal memory. These storage locations are generally referred as registers.

The major components of the CPU are an arithmetic 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:

Data Bus:

Data bus is used to transfer the data between main memory and CPU.

Address Bus:

Address bus is used to access a particular memory location by putting the address of the memory location.

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.

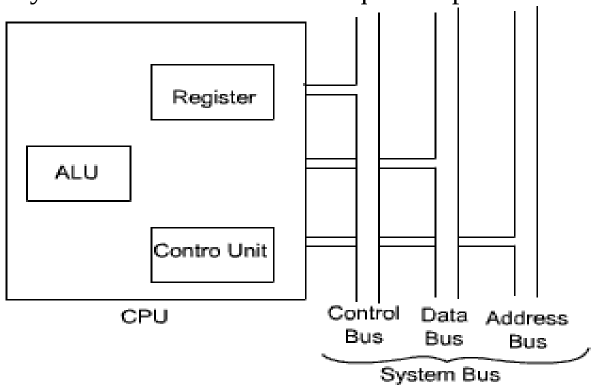


Figure 1: CPU with the system bus.

There are three basic components of CPU: register bank, ALU and Control Unit. There are several data movements between these units and for that an internal CPU bus is used. Internal CPU bus is needed to transfer data between the various registers and the ALU.

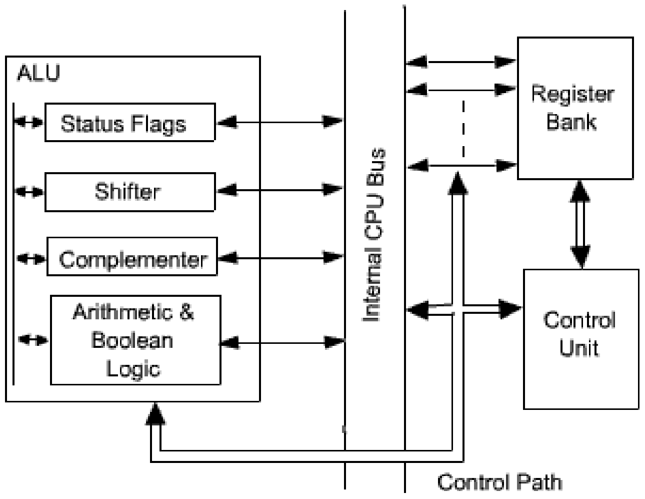


Figure 2 : Internal Structure of CPU

Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last in, first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can only(after an initial value is loaded in to it).The register that hold the address for the stack is called a stack pointer (SP) because its value always points at the top item in stack. Contrary to a stack of trays where the tray it self may be taken out or inserted, the physical registers of a stack are always available for reading or writing.

The two operation of stack are the insertion and deletion of items. The operation of insertion is called *PUSH* because it can be thought of as the result of pushing a new item on top. The operation of deletion is called *POP* because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register stack:

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure X shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C in the order. item C is on the top of the stack so that the content of sp is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of *SP*. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack. Note that item C has read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because 2^6 =64. since SP has only six bits, it cannot exceed a number grater than 63(111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 =1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one bit register Full is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written in to or read out of the stack.

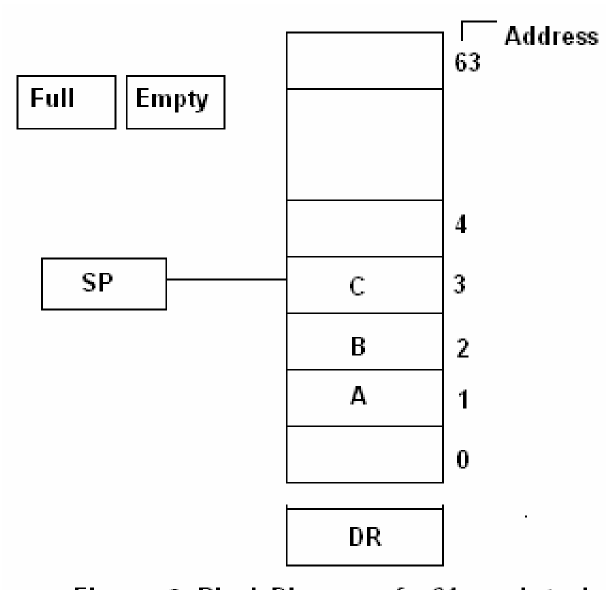


Figure :3 Block Diagram of a 64-word stack

Initially, SP is cleared to 0, Emty is set to 1, and Full is cleared to 0, so that SP points to the word at address o and the stack is marked empty and not full. if the stack is not full , a new item is inserted with a push operation. the push operation is implemented with the following sequence of micro-operation.

SP ←SP + 1	(Increment stack pointer)
M(SP) ← DR	(Write item on top of the stack)
if (sp=0) then (Full ← 1)	(Check if stack is full)
Emty ← 0	(Marked the stack not empty)

The stac pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that M(SP) denotes the memory word specified by the address presently available in SP, the first item stored in the stack is at address 1. The last item is stored at address 0, if SP reaches 0, the stack is full of item, so FULLL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and after increment SP, the last item stored in location 0. Once an item is stored in location 0, there are no more empty register in the stack. If an item is written in the stack, Obviously the stack can not be empty, so EMTY is cleared to 0.

DR← M[SP]	Read item from the top of stack
SP ← SP-1	Decrement stack Pointer
if(SP=0) then (Emty ← 1)	Check if stack is empty
FULL ← 0	Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. if its value reaches zero, the stack is empty, so Emty is set to 1. This condition is reached if the item read was in location 1. once this item is read out , SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equal to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL=1 or popped when EMTY =1.

Memory Stack :

A stack can exist as a stand-alone unit as in figure 4 or can be implemented in a random access memory attached to CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure shows a portion of computer memory partitioned in to three segment program, data and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three register are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or POP items into or from the stack.

As show in figure :4 the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address hat can be used for the stack is 3000. No previous are available for stack limit checks.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows.

SP← SP-1

M[SP] ← DR

The stack pointer is decremented so that it points at the address of the next word. A Memory write operation insertion the word from DR into the top of the stack. A new item is deleted with a pop operation as follows.

DR← M[SP]

SP←SP + 1

The top item is read from the stack in to DR. The stack pointer is then incremented to point at the next item in the stack.

Most computer do not provide hardware to check for stack overflow (FULL) or underflow (Empty). The stack limit can be checked by using two prossor register :

one to hold upper limit and other hold the lower limit. after the pop or push operation SP is compared with lower or upper limit register.

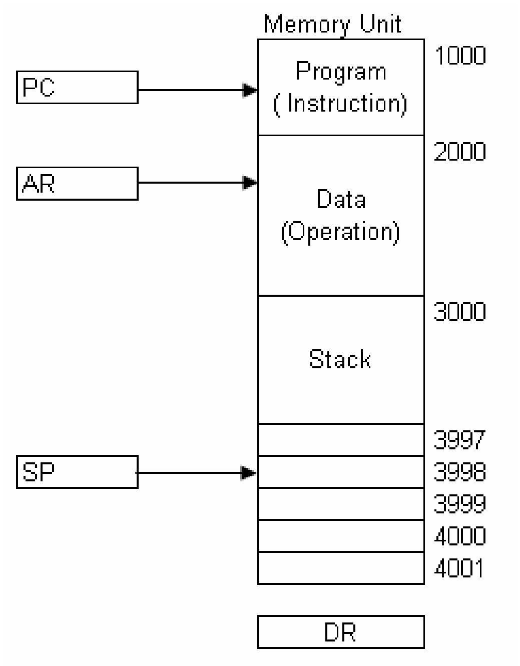


Figure : 4 Computer memory with program, data and stack segments

INSTRUCTION FORMATS

We know that a machine instruction has an opcode and zero or more operands. Encoding an instruction set can be done in a variety of ways. Architectures are differentiated from one another by the number of bits allowed per instruction (16, 32, and 64 are the most common), by the number of operands allowed per instruction, and by the types of instructions and data each can process. More specifically, instruction sets are differentiated by the following features:

1. Operand storage in the CPU (data can be stored in a stack structure or in registers)
2. Number of explicit operands per instruction (zero, one, two, and three being the most common)
3. Operand location (instructions can be classified as register-to-register, register-to-memory or memory-to-memory, which simply refer to the combinations of operands allowed per instruction)
4. Operations (including not only types of operations but also which instructions can access memory and which cannot)
5. Type and size of operands (operands can be addresses, numbers, or even characters)

Number of Addresses:

One of the characteristics of the ISA(Industrial Standard Architecture) that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however.

For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Three-Address Machines :

In three-address machines, instructions carry all three addresses explicitly. The RISC processors use three addresses. Table X1 gives some sample instructions of a three-address machine.

In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

```
mult    T,C,D          ; T = C*D
add     T,T,B          ; T = B + C*D
sub     T,T,E          ; T = B + C*D - E
add     T,T,F          ; T = B + C*D - E + F
add     A,T,A          ; A = B + C*D - E + F + A
```

Table :T1 Sample three-address machine instructions

Instruction	Semantics
add dest,src1,src2	Adds the two values at src1 and src2 and stores the result in dest M(dest) = [src1] + [src2]
sub dest,src1,src2	Subtracts the second source operand at src2 from the first at src1 and stores the result in dest M(dest) = [src1] - [src2]
mult dest,src1,src2	Multiplies the two values at src1 and src2 and stores the result in dest M(dest) = [src1] * [src2]

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table.

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary T and in the last one, it is A. This is the motivation for using two addresses, as we show next.

Two-Address Machines :

In two-address machines, one address doubles as a source and destination. Usually, we use dest to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. Sample instructions of a two-address machine.

On these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

```
load    T,C          ; T = C
mult    T,D          ; T = C*D
add     T,B          ; T = B + C*D
sub     T,E          ; T = B + C*D - E
add     T,F          ; T = B + C*D - E + F
add     A,T          ; A = B + C*D - E + F + A
```

Table :T2 Sample Two-address machine instructions

Instruction	Semantics
load dest,src	Copies the value at src to dest M(dest) = [src]
add dest,src	Adds the two values at src and dest and stores the result in dest M(dest) = [dest] + [src]
sub dest,src	Subtracts the second source operand at src from the first at dest and stores the result in dest M(dest) = [dest] - [src]
mult dest,src	Multiplies the two values at src and dest and stores the result in dest M(dest) = [dest] * [src]

Since we use only two addresses, we use a load instruction to first copy the C value into a temporary represented by T. If you look at these six instructions, you will notice that the operand T is common. If we make this our default, then we don’t need even two addresses: we can get away with just one address.

One-Address Machines :

In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the accumulators. In most machines, there is just a single accumulator register. This kind of design, called accumulator machines, makes sense if memory is expensive.

In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table X3.

In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

```
load C ; load C into the accumulator
mult D ; accumulator = C*D
add B ; accumulator = C*D+B
```

sub E ; accumulator = C*D+B-E
add F ; accumulator = C*D+B-E+F
add A ; accumulator = C*D+B-E+F+A
store A ; store the accumulator contents in A

Table :T3 Sample ONE-address machine instructions

Instruction	Semantics
load addr	Copies the value at address addr into the accumulator accumulator = [addr]
store addr	Stores the value in the accumulator at the memory address addr M(addr) = accumulator
add addr	Adds the contents of the accumulator and value at address addr accumulator = accumulator + [addr]
sub addr	Subtracts the value at memory address addr from the contents of the accumulator accumulator = accumulator - [addr]
mult addr	Multiplies the contents of the accumulator and value at address addr accumulator = accumulator * [addr]

Zero-Address Machines :

In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria.

All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table X4 gives some sample instructions for the stack machines.

Table :T4 Sample Zero-address machine instructions

Instruction	Semantics
push addr	Places the value at address addr on top of the stack push([addr])
pop addr	Stores the top value on the stack at memory address addr M(addr) = pop
add	Adds the top two values on the stack and pushes the result onto the stack push(pop + pop)
sub	Subtracts the second top value from the top value of the stack and pushes the result onto the stack push(pop – pop)
mult	Multiplies the top two values in the stack and pushes the result onto the stack push(pop * pop)

Notice that the first two instructions are not zero-address instructions. These two are special instructions that use a single address and are used to move data between memory and stack.

All other instructions use the zero-address format. Let's see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

```
push E      ; <E>
push C      ; <C, E>
push D      ; <D, C, E>
mult        ; <C*D, E>
push B      ; <B, C*D, E>
add         ; <B+C*D, E>
sub         ; <B+C*D-E>
push F      ; <F, B+D*C-E>
add         ; <F+B+D*C-E>
push A      ; <A, F+B+D*C-E>
add         ; <A+F+B+D*C-E>
pop A       ; < >
```

On the right, we show the state of the stack after executing each instruction. The top element of the stack is shown on the left. Notice that we pushed E early because we need to subtract it from (B+C*D).

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the stack depth. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth.

INSTRUCTION TYPES

Most computer instructions operate on data; however, there are some that do not. Computer manufacturers regularly group instructions into the following categories:

• Data movement
• Arithmetic
• Boolean
• Bit manipulation (shift and rotate)
• I/O
• Transfer of control
• Special purpose

Data movement instructions are the most frequently used instructions. Data is moved from memory into registers, from registers to registers, and from registers to memory, and many machines provide different instructions depending on the source and destination. For example, there may be a MOVER instruction that always requires two register operands, whereas a MOVE instruction allows one register and one memory operand.

Some architectures, such as RISC, limit the instructions that can move data to and from memory in an attempt to speed up execution. Many machines have ariations of load, store, and move instructions to handle data of different sizes. For example, there may be a LOADB instruction for dealing with bytes and a LOADW instruction for handling words.

Arithmetic operations include those instructions that use integers and floating point numbers. Many instruction sets provide different arithmetic instructions for various data sizes. As with the data movement instructions, there are sometimes different instructions for providing various combinations of register and memory accesses in different addressing modes.

Boolean logic instructions perform Boolean operations, much in the same way that arithmetic operations work. There are typically instructions for performing AND, NOT, and often OR and XOR operations.

Bit manipulation instructions are used for setting and resetting individual bits (or sometimes groups of bits) within a given data word. These include both arithmetic and logical shift instructions and rotate instructions, both to the left and to the right. Logical shift instructions simply shift bits to either the left or the right by a specified amount, shifting in zeros from the opposite end. Arithmetic shift instructions, commonly used to multiply or divide by 2, do not shift the leftmost bit, because this represents the sign of the number. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, values are shifted left, zeros are shifted in, but the sign bit is never moved. Rotate instructions are simply shift instructions that shift in the bits that are shifted out. For example, on a rotate left 1 bit, the leftmost bit is shifted out and rotated around to become the rightmost bit.

I/O instructions vary greatly from architecture to architecture. The basic schemes for handling I/O are programmed I/O, interrupt-driven I/O, and DMA devices. These are covered in more detail in Chapter 5.

Control instructions include branches, skips, and procedure calls. Branching can be unconditional or conditional. Skip instructions are basically branch instructions with implied addresses. Because no operand is required, skip instructions often use bits of the address field to specify different situations (recall the Skipcond instruction used by MARIE). Procedure calls are special branch instructions that automatically save the return address. Different machines use different methods to save this address. Some store the address at a specific location in memory, others store it in a register, while still others push the return address on a stack. We have already seen that stacks can be used for other purposes.

Special purpose instructions include those used for string processing, high level language support, protection, flag control, and cache management. Most architectures provide instructions for string processing, including string manipulation and searching.

Addressing Modes

We have examined the types of **operands** and **operations** that may be specified by **machine instructions**. Now we have to see how is the address of an operand specified, and how are the **bits** of an instruction organized to define the **operand addresses** and operation of that instruction.

Addressing Modes: The most common addressing techniques are

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

All computer architectures provide more than one of these addressing modes. The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of *effective address*. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

To explain the addressing modes, we use the following notation:

A	=	contents of an address field in the instruction that refers to a memory
R	=	contents of an address field in the instruction that refers to a register
EA	=	actual (effective) address of the location containing the referenced operand
(X)	=	contents of location X

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:

OPERAND = A

This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

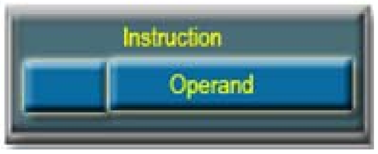


Figure 4.1: Immediate Addressing Mode

The instruction format for Immediate Addressing Mode is shown in the Figure 4.1.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

It requires only one memory reference and no special calculation.

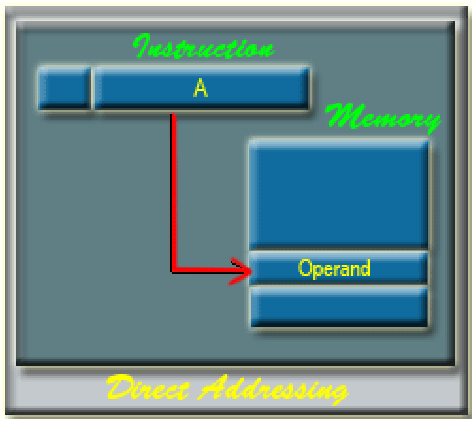


*Figure 4.2**:** Direct Addressing Mode*

Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is know as indirect addressing:

EA = (A)

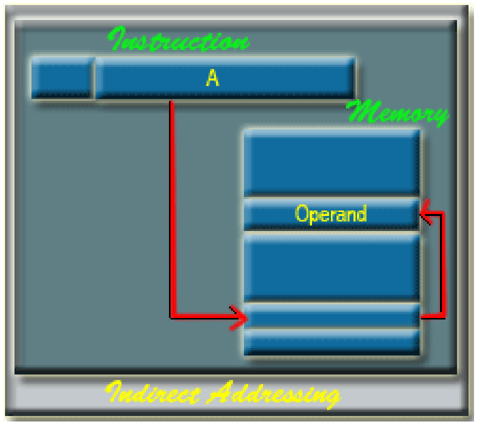


Figure 4.3: Indirect Addressing Mode

Register Addressing:

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.

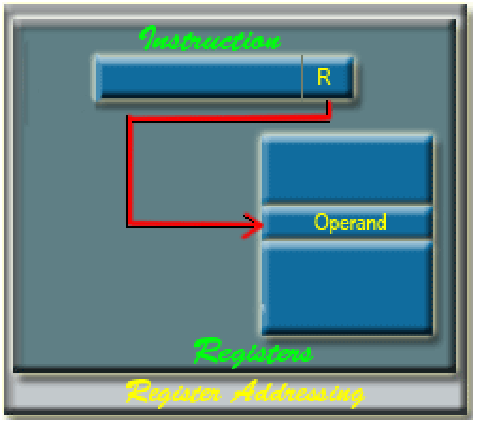


Figure 4.4: Register Addressing Mode.

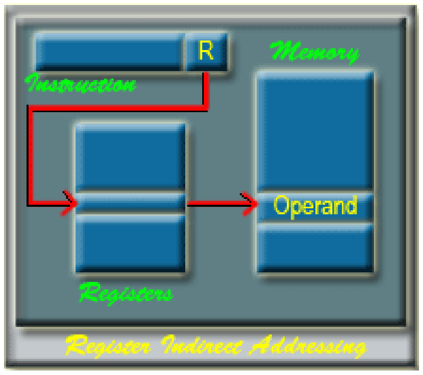
The exact register location of the operand in case of Register Addressing Mode is shown in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

$$EA = (R)$$

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.



Diaplacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address. The general format of Displacement Addressing is shown in the Figure 4.6.

Three of the most common use of displacement addressing are:

- Relative addressing
- Base-register addressing
- Indexing

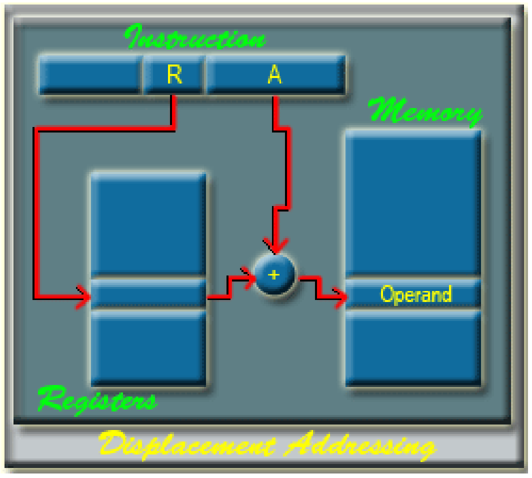


Figure 4.6: Displacement Addressing

Relative Addressing:

For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.

Base-Register Addressing:

The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

Indexing:

The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because

this is such a common operation, some system will automatically do this as part of the same instruction cycle.
This is known as auto-indexing. We may get two types of auto-indexing: -one is auto-incrementing and the other one is -auto-decrementing.

If certain registers are devoted exclusively to indexing, then auto-indexing can be invoked implicitly and automatically. If general purpose register are used, the auto index operation may need to be signaled by a bit in the instruction.

Auto-indexing using *increment* can be depicted as follows:

$$EA = A + (R)$$
$$R = (R) + 1$$

Auto-indexing using *decrement* can be depicted as follows:

$$EA = A + (R)$$
$$R = (R) - 1$$

In some machines, both *indirect addressing* and *indexing* are provided, and it is possible to employ both in the same instruction. There are two possibilities: The indexing is performed either before or after the indirection.
If indexing is performed after the indirection, it is termed post indexing

$$EA = (A) + (R)$$

First, the contents of the address field are used to access a memory location containing an address. This address is then indexed by the register value.

With pre indexing, the indexing is performed before the indirection:

$$EA = (A + (R)$$

An address is calculated, the calculated address contains not the operand, but the address of the operand.

Stack Addressing:

A stack is a linear array or list of locations. It is sometimes referred to as a pushdown list or last-in-first-out queue. A stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.