UNIT 1 INSTRUCTION SET ARCHITECTURE

Structure **Page No.**

1.0 INTRODUCTION

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler designer. They are the parts of a processor design that need to be understood in order to write assembly language, such as the machine language instructions and registers. Parts of the architecture that are left to the implementation are not part of ISA. The ISA serves as the boundary between software and hardware.

The term instruction will be used in this unit more often. What is an instruction? What are its components? What are different types of instructions? What are the various addressing schemes and their importance? This unit is an attempt to answer these questions. In addition, the unit also discusses the design issues relating to instruction format. We have presented here the instruction set of MIPS (Microprocessor without Interlocked Pipeline Stages) processor (very briefly) as an example.

Other related microprocessors instruction set can be studied from further readings. We will also discuss about the complete instruction set of 8086 micro-processor in unit 1, Block 4 of this course.

1.1 OBJECTIVES

After going through this unit you should be able to:

- describe the characteristics of instruction set;
- discuss various elements of an instruction;
- differentiate various types of operands;

- distinguish various types of instructions and various operations performed by the instructions;
- identify different types of ISAs on the basis of addresses in instruction sets;
- identify various addressing schemes; and
- discuss the instruction format design issues.

1.2 INSTRUCTION SET CHARACTERISTICS

The key role of the Central Processing Unit (CPU) is to perform the calculations, to issue the commands, to coordinate all other hardware components, and executing programs including operating system, application programs etc. on your computer. But CPU is primarily the core hardware component; you must speak to it in the core binary machine language. The words of a machine language are known as *instructions*, and its syntax is known as an *instruction set.*

The Instruction Set Viewpoints

Instruction set is the boundary where the computer designer and the computer programmer see the same computer from different viewpoints. From the designer's point of view, the computer instruction set provides a functional description of a processor, that is:

(i) A detailed list of the instructions that a processor is capable of processing.
(ii) A description of the types/ locations/ access methods for operands.

The common goal of computer designers is to build the hardware for implementing the machine's instructions for CPU. From the programmer's point of view, the user must understand machine or assembly language for low-level programming. Moreover, the user must be aware of the register set, instruction types and the function that each instruction performs.

This unit covers both the viewpoints. However, our prime focus is the programmer's viewpoint with the design of instruction set. Now, let us define the instructions, parts of instruction and so on.

What is an Instruction Set?

Instruction set is the collection of machine language instructions that a particular processor understands and executes. In other words, a set of assembly language mnemonics represents the machine code of a particular computer. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. It should be noted here that the instructions available in a computer are machine dependent, that is, a different processors have different instruction sets. However, a newer processor that may belong to some family may have a compatible but extended instruction set of an old processor of that family. **Instructions can take different formats**. The instruction format involves:

- the instruction length;

- the type;
- length and position of operation codes in an instruction; and
- the number and length of operand addresses etc.

An interesting question for instruction format may be to have uniform length or variable length instructions.

What are the elements of an instruction?

As the purpose of instruction is to communicate to CPU what to do, it requires a minimum set of communication as:

- What operation to perform?
- On what operands?

Thus, each instruction consists of several fields. The most common fields found in instruction formats are:

Opcode: (What operation to perform?)

- An operation code field termed as **opcode** that specifies the operation to be performed.

Operands: (Where are the operands?)

- An address field of operand on which data processing is to be performed.
- An operand can reside in the memory or a processor register or can be incorporated within the operand field of instruction as an **immediate constant**. Therefore a mode field is needed that specifies the way the operand or its address is to be determined.

A sample instruction format is given in figure 1.

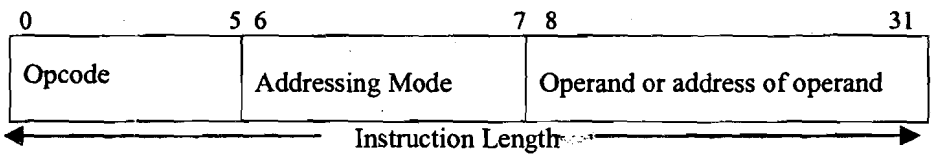


Figure 1: A Hypothetical Instruction Format of 32 bits

Please note the following points in Figure 1:

- The opcode size is 6 bits. So, in general it will have $2^6 = 32$ operations. (However, when you will study more architectures from further readings, you will find even through these bits using special combinations. Instruction set designers have developed much more operations).
- There is only one operand address machine. What is the significance of this? You will find an answer of this question in section 1.3.3 of this unit.
- There are two bits for addressing modes. Therefore, there are $2^2 = 4$ different addressing modes possible for this machine.
- The last field (8 – 31 bits = 24 bits) here is the operand or the address of operand field.

In case of immediate operand the maximum size of the unsigned operand would be 2^{24}.

In case it is an address of operand in memory, then the maximum physical memory size supported by this machine is 2^{24} = 16 MB.

For this machine there may be two more possible addressing modes in addition to the immediate and direct. However, let us not discuss addressing modes right now. They will be discussed in general, details in section 1.4 of this unit.

The opcode field of an instruction is a group of bits that define various processor operations such as LOAD, STORE, ADD, and SHIFT to be performed on some data stored in registers or memory.

The operand address field can be **data**, or can refer to data – that is address of data, or can be labels, which may be the address of an instruction you want to execute next, such labels are commonly used in Subroutine call instructions. An operand address can be:

- The memory address
- CPU register address
- I/O device address

The mode field of an instruction specifies a variety of alternatives for referring to operands using the given address. **Please note that if the operands are placed in processor registers then an instruction executes faster than that of operands placed in memory, as the registers are very high-speed memory used by the CPU. However, to put the value of a memory operand to a register you will require a register LOAD instruction.**

How is an instruction represented?

Instruction is represented as a sequence of bits. A layout of an instruction is termed as *instruction format*. Instruction formats are primarily machine dependent. A CPU instruction set can use many instruction formats at a time. Even the length of opcode varies in the same processor. However, we will not discuss such details in this block. You can refer to further readings for such details.

How many instructions in a Computer?

A computer can have a large number of instructions and addressing modes. The older computers with the growth of Integrated circuit technology have a very large and complex set of instructions. These are called "complex instruction set computers" (CISC). Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

However, later it was found in the studies of program style that many complex instructions found CISC are not used by the program. This lead to the idea of making a simple but faster computer, which could execute simple instructions much faster. These computers have simple instructions, registers addressing and move registers. These are called Reduced Instruction Set Computers (RISC). We will study more about RISC in Unit 5 of this Block.

Check Your Progress 1

State True or False.

T	F

1. An instruction set is a collection of all the instructions a CPU can execute. ☐
2. Instructions can take different formats. ☐
3. The opcode field of an instruction specifies the address field of operand on which data processing is to be performed. ☐

4. The operands placed in processor registers are fetched faster than that of operands placed in memory. ☐

5. Operands must refer to data and cannot be data. ☐

1.3 INSTRUCTION SET DESIGN CONSIDERATIONS

Some of the basic considerations for instruction set design include selection of:

- A set of data types (e.g. integers, long integers, doubles, character strings etc.).
- A set of operations on those data types.
- A set of instruction formats. Includes issues like number of addresses, instruction length etc.
- A set of techniques for addressing data in memory or in registers.
- The number of registers which can be referenced by an instruction and how they are used.

We will discuss the above concepts in more detail in the subsequent sections.

1.3.1 Operand Data Types

Operand is that part of an instruction that specifies the address of the source or result, or the data itself on which the processor is to operate. Operand types usually give operand size implicitly. In general, operand data types can be divided in the following categories. Refer to figure 2:

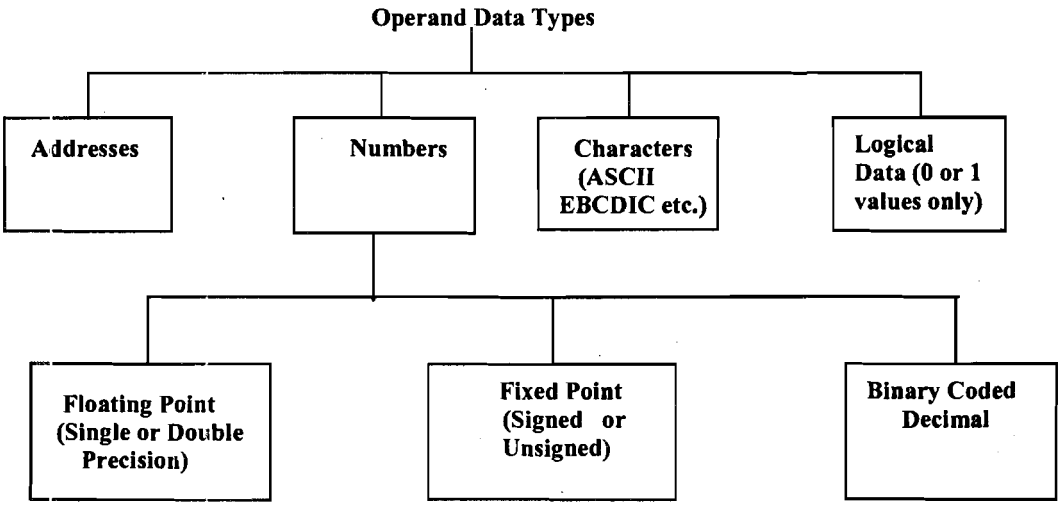


Figure 2: Operand Data Types

- *Addresses:* Operands residing in memory are specified by their memory address and operands residing in registers are specified by a register address. Addresses provided in the instruction are operand references.

- *Numbers*: All machine languages include numeric data types. Numeric data usually use one of three representations:
  - Floating-point numbers-single precision (1 sign bit, 8 exponent bits, 23 mantissa bits) and double precision (1 sign bit, 11 exponent bits, 52 mantissa bits).
  - Fixed point numbers (signed or unsigned).

- Binary Coded Decimal Numbers.

Most of the machines provide instructions for performing arithmetic operations on fixed point and floating-point numbers. However, there is a limit in magnitude of numbers due to underflow and overflow.

- *Characters*: A common form of data is text or character strings. Characters are represented in numeric form, mostly in ASCII (American Standard Code for Information Exchange). Another Code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC).

- *Logical data*: Each word or byte is treated as a single unit of data. When an n-bit data unit is considered as consisting of n 1-bit items of data with each item having the value 0 or 1, then they are viewed as logical data. Such bit-oriented data can be used to store an array of Boolean or binary data variables where each variable can take on only the values 1 (true) and 0 (false). One simple application of such a data may be the cases where we manipulate bits of a data item. For example, in floating-point addition we need to shift mantissa bits.

1.3.2 Types of Instructions

Computer instructions are the translation of high level language code to machine level language programs. Thus, from this point of view the machine instructions can be classified under the following categories. Refer to figure 3:

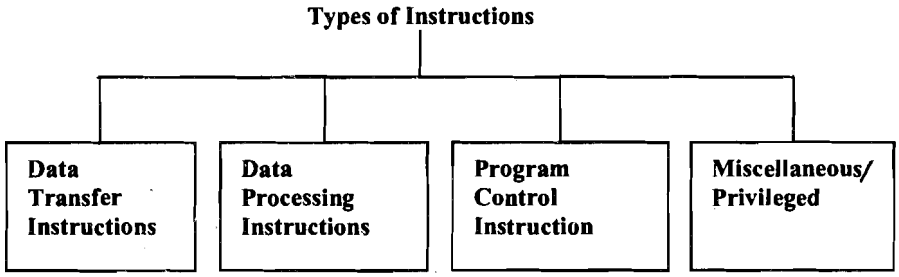


Figure 3: Types of Instructions

Data Transfer Instructions

These instructions transfer data from one location in the computer to another location without changing the data content. The most common transfers are between:

- processor registers and memory,
- processor registers and I/O, and
- processor registers themselves.

These instructions need:

- the location of source and destination operands and
- the mode of addressing for each operand. Given below is a table, which lists eight data transfer instructions with their mnemonic symbols. These symbols are used for understanding purposes only, the actual instructions are binary. Different computers may use different mnemonic for the same instruction.

Operation Name	Mnemonic	Description
Load	LD	Loads the contents from memory to register.
Store	ST	Store information from register to memory location.
Move	MOV	Data Transfer from one register to another or between CPU registers and memory.

Exchange	XCH	Swaps information between two registers or a register and a memory word.
Clear	CLEAR	Causes the specified operand to be replaced by 0's.
Set	SET	Causes the specified operand to be replaced by 1's.
Push	PUSH	Transfers data from a processor register to top of memory stack.
Pop	POP	Transfers data from top of stack to processor register.

Data Processing Instructions

These instructions perform arithmetic and logical operations on data. Data Manipulation Instructions can be divided into three basic types:

Arithmetic: The four basic operations are ADD, SUB, MUL and DIV. An arithmetic instruction may operate on fixed-point data, binary or decimal data etc. The other possible operations include a variety of single-operand instructions, for example ABSOLUTE, NEGATE, INCREMENT, DECREMENT.

The execution of arithmetic instructions requires bringing in the operands in the operational registers so that the data can be processed by ALU. Such functionality is implemented generally within instruction execution steps.

Logical: AND, OR, NOT, XOR operate on binary data stored in registers. For example, if two registers contain the data:

R1 = 1011 0111
R2 = 1111 0000

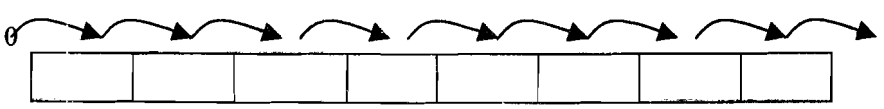
Then,

R1 AND R2 = 1011 0000. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits. With one register is set to all 1's, the XOR operation inverts those bits in R_1 register where R_2 contains 1.

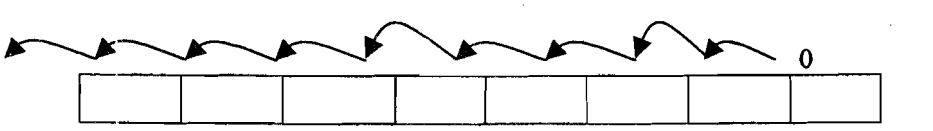
R_1 XOR R_2 = 0100 0111

Shift: Shift operation is used for transfer of bits either to the left or to the right. It can be used to realize simple arithmetic operation or data communication/recognition etc. Shift operation is of three types:

1. Logical shifts LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT insert zeros to the end bit position and the other bits of a word are shifted left or right respectively. The end bit position is the leftmost bit for shift right and the rightmost bit position for the shift left. The bit shifted out is lost.



Logical Shift Right



Logical Shift Left
Figure 4: Logical Shift

2. Arithmetic shifts ARITHMETIC SHIFT LEFT and ARITHMETIC SHIFT RIGHT are the same as LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT

except that the sign bit it remains unchanged. On an arithmetic shift right, the sign bit is replicated into the bit position to its right. On an arithmetic shift left, a logical shift left is performed on all bits but the sign bit, which is retained.

The arithmetic left shift and a logical left shift when performed on numbers represented in two's complement notation cause multiplication by 2 when there is no overflow. Arithmetic shift right corresponds to a division by 2 provided there is no underflow.

3. Circular shifts ROTATE LEFT and ROTATE RIGHT. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.

Character and String Processing Instructions: String manipulation typically is done in memory. Possible instructions include COMPARE STRING, COMPARE CHARACTER, MOVE STRING and MOVE CHARACTER. While compare character usually is a byte-comparison operation, compare string always involves memory address.

Stack and register manipulation: If we build stacks, stack instructions prove to be useful. LOAD IMMEDIATE is a good example of register manipulation (the value being loaded is part of the instruction). Each CPU has multiple registers, when instruction set is designed; one has to specify which register the instruction is referring to.

No operation (or idle) is needed when there is nothing to run on a computer.

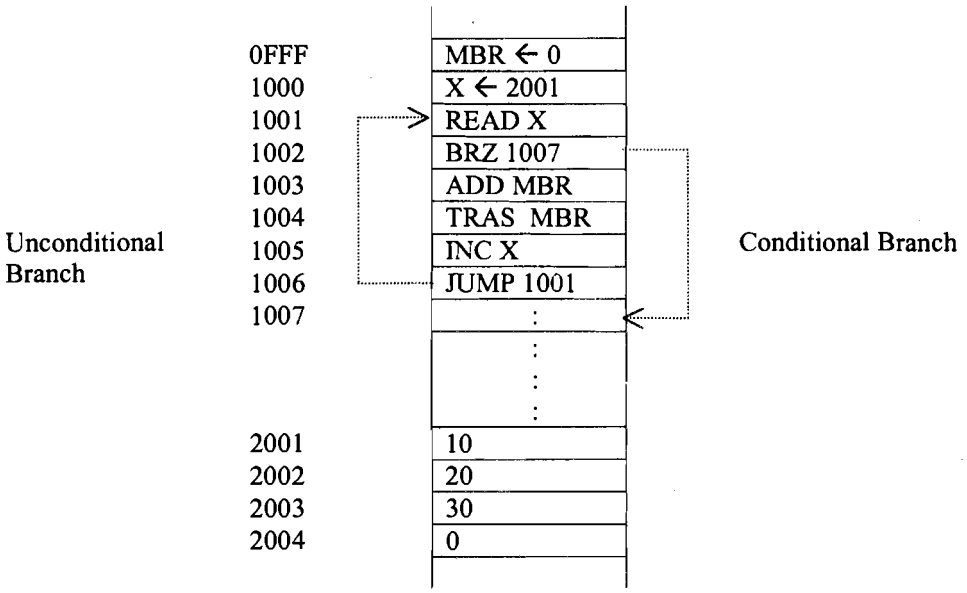
Program Control Instructions

These instructions specify conditions for altering the sequence of program execution or in other words the content of PC (program counter) register. PC points to memory location that holds the next instruction to be executed. The change in value of PC as a result of execution of control instruction like BRANCH or JUMP causes a break in the sequential execution of instructions. The most common control instructions are:

BRANCH and JUMP may be conditional or unconditional. JUMP is an unconditional branch used to implement simple loops. JNE jump not equal is a conditional branch instruction. The conditional branch instructions such as BRP X and BRN X causes a branch to memory location X if the result of most recent operation is positive or negative respectively. If the condition is true, PC is loaded with the branch address X and the next instruction is taken from X, otherwise, PC is not altered and the next instruction is taken from the location pointed by PC. Figure 5 shows an unconditional branch instruction, and a conditional branch instruction if the content of AC is zero.

```
MBR ← 0          ; Assign 0 to MBR register
X← 2001          ; Assume X to be an address location 2001
READ X           ; Read a value from memory location 2001 into AC
BRZ 1007         ; Branch to location 1007 if AC is zero (Conditional branch
                   on zero)
ADD MBR          ; Add the content of MBR to AC and store result to AC
TRAS  MBR        ; Transfer the contents of AC to MBR
INC   X          ; Increment X to point to next location
JUMP 1001        ; Loop back for further processing.
```

(a) A program on hypothetical machine



(b) The Memory of the hypothetical machine

Figure 5: BRANCH & JUMP Instructions

The program given in figure 5 is a hypothetical program that performs addition of numbers stored from locations 2001 onwards till a zero is encountered. Therefore, X is initialized to 2001, while MBR that stores the result is initialized to zero. We have assumed that in this machine all the operations are performed using CPU. The programs will execute instructions as:

1st Cycle:
1001 (with location X = 2001 which is value 10) → 1002 → 1003 → 1004→1005 (X is incremented to 2002)→ 1006

2nd Cycle
1001 (with X = 2002 which is 20) → 1002 → 1003 → 1004 → 1005 (X is 2003) → 1006

3rd Cycle
1001 (with X = 2003 which is 30) → 1002 → 1003 →1004 → 1005 (X is 2004) → 1006

4th Cycle
1001 (with X = 2004 which is 0) → 1002 [AC contains zero so take a branch to 1007]

1007..... (MBR contains the added value)

The **SKIP** instruction is a zero-address instruction and skips the next instruction to be executed in sequence. In other words, it increments the value of PC by one instruction length. The SKIP can also be conditional. For example, the instruction ISZ skips the next instruction only if the result of the most recent operation is zero.

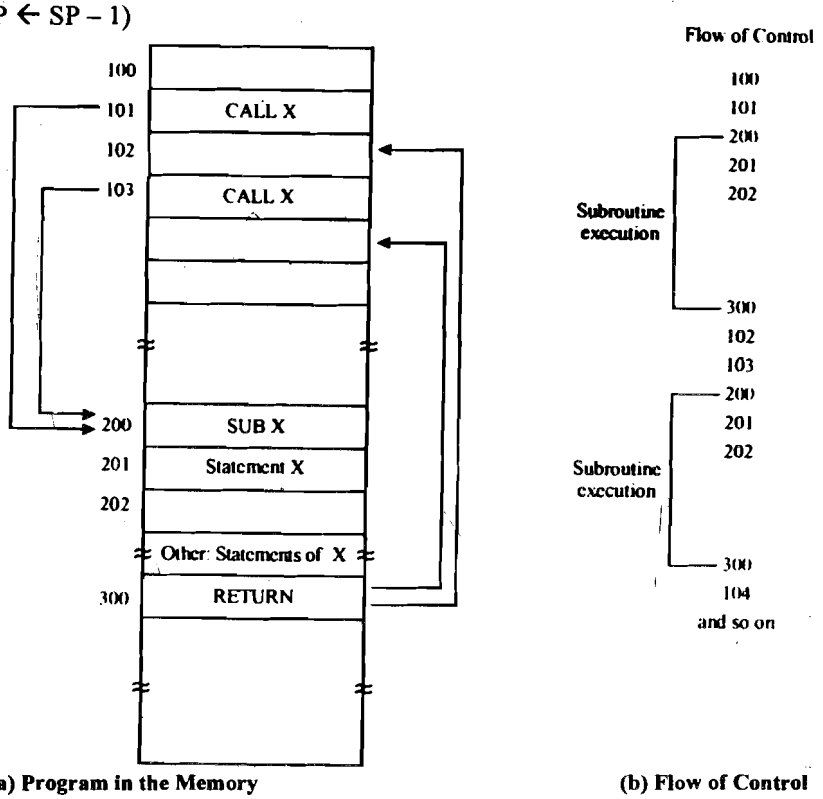
CALL and RETN are used for CALLing subprograms and RETurning from them. Assume that a memory stack has been built such that stack pointer points to a non-empty location stack and expand towards zero address.

CALL:

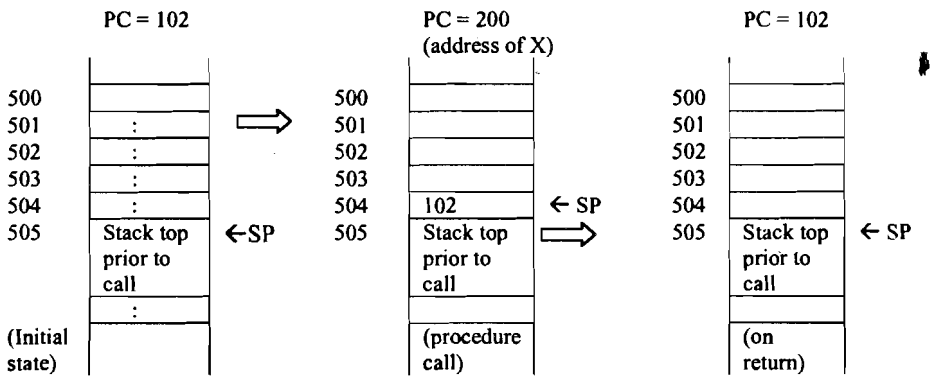
CALL X Procedure Call to function /procedure named X
CALL instruction causes the following to happen:

1. Decrement the stack pointer so that we will not overwrite last thing put on stack,

(SP ← SP – 1)



(a) Program in the Memory **(b) Flow of Control**



(c) Memory Stack Values for first call
Figure 6: Call and Return Statements

2. The contents of PC, which is pointing to NEXT instruction, the one just after the CALL is pushed onto the stack, and, M [SP] ←PC.
3. JMP to X, the address of the start of the subprogram is put in the PC register; this is all a **jump** does. Thus, we go off to the subprogram, but we have to remember where we were in the calling program, i.e. we must remember where we came from, so that we can get back there again.

PC ← X

RETN :

RETN Return from procedure.
RETN instruction causes the following to happen:

1. Pops the stack, to yield an address/label; if correctly used, the top of the stack will contain the address of the next instruction after the call from which we are returning; it is this instruction with which we want to resume in the *calling* program;
2. Jump to the popped address, i.e., put the address into the PC register.

   PC ← top of stack value; Increment SP.

Miscellaneous and Privileged Instructions: These instructions do not fit in any of the above categories. I/O instructions: start I/O, stop I/O, and test I/O. Typically, I/O destination is specified as an address. Interrupts and state-swapping operations: There are two kinds of exceptions, interrupts that are generated by hardware and traps, which are generated by programs. Upon receiving interrupts, the state of current processes will be saved so that they can be restarted after the interrupt has been taken care of.

Most computer instructions are divided into two categories, privileged and non-privileged. A process running in privileged mode can execute all instructions from the instruction set while a process running in user mode can only execute a sub-set of the instructions. I/O instructions are one example of privileged instruction, clock interrupts are another one.

1.3.3 Number of Addresses in an Instruction

In general, the Instruction Set Architecture (ISA) of a processor can be differentiated using five categories:

- Operand Storage in the CPU - Where are the operands kept other than the memory?
- Number of explicitly named operands - How many operands are named in an instruction?
- Operand location - Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU registers?
- Operations - What operations are provided in the ISA?
- Type and size of operands - What is the type and size of each operand and how is it specified?

As far as operations and type of operands are concerned, we have already discussed about these in the previous subsection. In this section let us look into some of the architectures that are common in contemporary computer. But before we discuss the architectures, let us look into some basic instruction set characteristics:

- The operands can be addressed in memory, registers or I/O device address.
- Instruction having less number of operand addresses in an instruction may require lesser bits in the instruction; however, it also restricts the range of functionality that can be performed by the instructions. This implies that a CPU instruction set having less number of addresses has longer programs, which means longer instruction execution time. On the other hand, having more addresses may lead to more complex decoding and processing circuits.
- Most of the instructions do not require more than three operand addresses. Instructions having fewer addresses than three, use registers implicitly for operand locations because using registers for operand references can result in smaller instructions as only few bits are needed for register addresses as against memory addresses.
- The type of internal storage of operands in the CPU is the most basic differentiation.

The three most common types of ISAs are:

1. **Evaluation Stack**: The operands are implicitly on top of the stack.
2. **Accumulator:** One operand is implicitly the accumulator.
3. **General Purpose Register (GPR)**: All operands are explicit, either registers or memory locations.

Evaluation Stack Architecture: A stack is a data structure that implements Last-In-First-Out (LIFO) access policy. You could add an entry to the stack with a PUSH(value) and remove an entry from the stack with a POP( ). No explicit operands are there in ALU instructions, but one in PUSH/POP. Examples of such computers are Burroughs B5500/6500, HP 3000/70 etc.

On a stack machine "C = A + B" might be implemented as:.

```
PUSH A
PUSH B

ADD     // operator POP operand(s) and PUSH result(s) (implicit on top of stack)

POP C
```

Stack Architecture: Pros and Cons

- Small instructions (do not need many bits to specify the operation).
- Compiler is easy to write.
- Lots of memory accesses required - everything that is not on the stack is in memory. Thus, the machine performance is poor.

Accumulator Architecture: An accumulator is a specially designated register that supplies one instruction operand and receives the result. The instructions in such machines are normally one-address instructions. The most popular early architectures were IBM 7090, DEC PDP-8 etc.

On an Accumulator machine "C = A + B" might be implemented as:

```
LOAD A      // Load memory location A into accumulator
ADD B       // Add memory location B to accumulator
STORE C     // Store accumulator value into memory location C
```

Accumulator Architecture: Pros and Cons

- Implicit use of accumulator saves instruction bits.
- Result is ready for immediate reuse, but has to be saved in memory if next computation does not use it right away.
- More memory accesses required than stack. Consider a program to do the expression:
  A = B * C + D * E.

Evaluation of Stack Machine		**Accumulator Machine**	
Program	**Comments**	**Programs**	**Comments**
PUSH B	Push the value B	LOAD B	Load B in AC
PUSH C	Push C	MULT C	Multiply AC with C in AC
MULT	Multiply (B×C) and store result on stack top	STORE T	Store B×C into Temporary T
PUSH D	Push D	LOAD D	Load D in AC
PUSH E	Push E	MULT E	Multiply E in AC

MULT	Multiply D×E and store result on stack top	ADD T	B×C + D×E
ADD	Add the top two values on the stack	STORE A	Store Result in A
POP A	Store the value in A		

General Purpose Register (GPR) Architecture: A register is a word of internal memory like the accumulator. GPR architecture is an extension of the accumulator idea, i.e., use a set of general-purpose registers, which must be explicitly named by the instruction. Registers can be used for anything either holding operands for operations or temporary intermediate values. The dominant architectures are IBM 370, PDP-11 and all Reduced Instant Set Computer (RISC) machines etc. The major instruction set characteristic whether an ALU instruction has two or more operands divides GPR architectures:

"C = A + B" might be implemented on both the architectures as:

Register - Memory	**Load/Store through Registers**
LOAD R1, A	LOAD R1, A
ADD R1, B	LOAD R2, B
STORE C, R1	ADD R3, R1, R2
	STORE C, R3

General Purpose Register Architecture: Pros and Cons

- Registers can be used to store variables as it reduces memory traffic and speeds up execution. It also improves code density, as register names are shorter than memory addresses.
- Instructions must include bits to specify which register to operate on, hence large instruction size than accumulator type machines.
- Memory access can be minimized (registers can hold lots of intermediate values).
- Implementation is complicated, as compiler writer has to attempt to maximize register usage.

While most early machines used stack or accumulator architectures, in the last 15 years all CPUs made are GPR processors. The three major reasons are that registers are faster than memory; the more data that can be kept internally in the CPU the faster the program will run. The third reason is that registers are easier for a compiler to use.

But while CPU's with GPR were clearly better than previous stack and accumulator based CPU's yet they were lacking in several areas. The areas being: Instructions were of varying length from 1 byte to 6-8 bytes. This causes problems with the pre-fetching and pipelining of instructions. ALU instructions could have operands that were memory locations because the time to access memory is slower and so does the whole instruction.

Thus in the early 1980s the idea of **RISC** was introduced. RISC stands for Reduced Instruction Set Computer. Unlike CISC, this ISA uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. The first RISC CPU, the MIPS 2000, has 32 GPRs. MIPS is a load/store architecture, which means that only load and store instructions access memory. All other computational instructions operate only on values stored in registers.

Check Your Progress 2

1. Match the following pairs:

(a)	Zero address instruction	(i)	Accumulator machines
(b)	One address instruction	(ii)	General Purpose Register machine
(c)	Three address instruction	(iii)	Evaluation-Stack machine

2. List the advantages and disadvantages of General Purpose Register machines.

3. Categorize the following operations with the respective instruction types:

(a)	MOVE	(i)	Data Processing Instructions
(b)	DIV	(ii)	Data Transfer Instructions
(c)	STORE	(iii)	Privileged Instructions
(d)	XOR	(iv)	Program Control Instructions
(e)	BRN		
(f)	COMPARE		
(g)	TRAP		

1.4 ADDRESSING SCHEMES

As discussed earlier, an operation code of an instruction specifies the operation to be performed. This operation is executed on some data stored in register or memory. Operands may be specified in one of the three basic forms i.e., immediate, register, and memory.

But, why addressing schemes? The question of **addressing** is concerned with how operands are interpreted. In other words, the term **'addressing schemes'** refers to the mechanism employed for specifying operands. There are a multitude of addressing schemes and instruction formats. Selecting which schemes are available will impact not only the ease to write the compiler, but will also determine how efficient the architecture can be?

All computers employ more than one addressing schemes to give programming flexibility to the user by providing facilities such as pointers to memory, loop control, indexing of data, program relocation and to reduce the number of bits in the operand field of the instruction. Offering a variety of addressing modes can help reduce instruction counts but having more modes also increases the complexity of the machine and in turn may increase the average Cycles per Instruction (CPI). Before we discuss the addressing modes let us discuss the notations being used in this section.

In the description that follows the symbols A, A1, A2 ..... etc. denote the content of an **operand field**. Thus, Ai may refer to a data or a memory address. In case the operand field is a register address, then the symbols R, R1, R2,... etc., are used. If C denotes the contents (either of an operand field or a register or of a memory location), then (C) denotes the content of the memory location whose address is C.

The symbol EA (Effective Address) refers to a physical address in a non-virtual memory environment and refers to a register in a virtual memory address environment. This register address is then mapped to physical memory address.

What is a virtual address? von Neumann had suggested that the execution of a program is possible only if the program and data are residing in memory. In such a situation the program length along with data and other space needed for execution cannot exceed the total memory. However, it was found that at the time of execution, the complete portion of data and instruction is not needed as most of the time only few areas of the program are being referenced. Keeping this in mind a new idea was put

forward where only a required portion is kept in the memory while the rest of the program and data reside in secondary storage. The data or program portion which are stored on secondary storage are brought to memory whenever needed and the portion of memory which is not needed is returned to the secondary storage. Thus, a program size bigger than the actual physical memory can be executed on that machine. This is called virtual memory. Virtual memory has been discussed in greater details as part of the operating system.

The typicality of virtual addresses is that:

- they are longer than the physical addresses as total addressed memory in virtual memory is more than the actual physical memory.
- if a virtual addressed operand is not in the memory then the operating system brings that operand to the memory.

The symbols D, D1, D2,..., etc. refer to actual operands to be used by instructions for their execution.

Most of the machines employ a set of addressing modes. In this unit, we will describe some very common addressing modes employed in most of the machines. A specific addressing mode example, however, is given in Unit 1 of Block 4.

The following tree shows the common addressing modes:

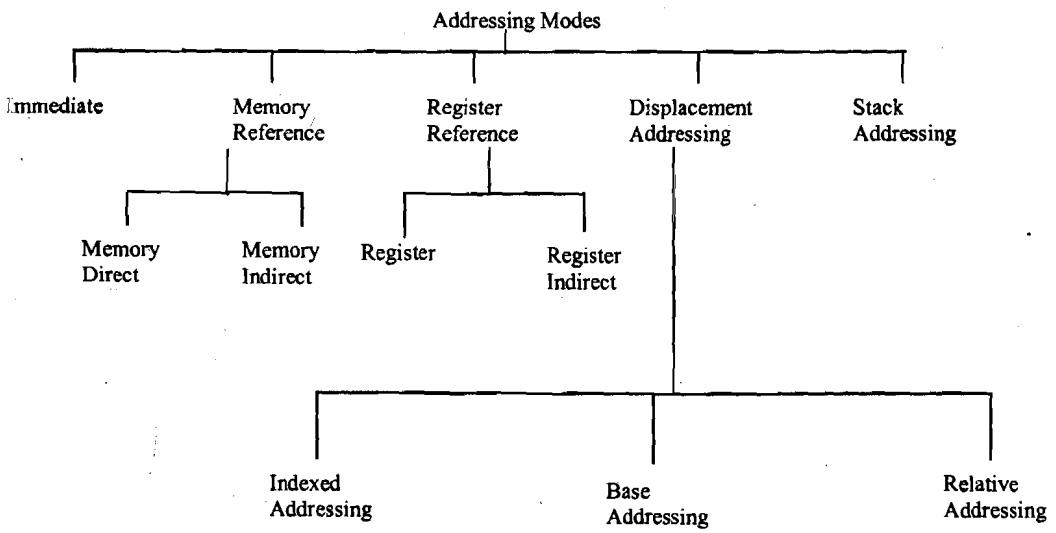


Figure 7: Common Addressing Modes

But what are the uses /applications of these addressing modes?

In general not all of the above modes are used for all applications. However, some of the common areas where compilers of high-level languages use them are:

Addressing Mode	Possible use
Immediate	For moving constants and initialization of variables
Direct	Used for global variables and less often for local variables
Register	Frequently used for storing local variables of procedures
Register Indirect	For holding pointers to structure in programming languages C
Index	To access members of an array
Auto-index mode	For pushing or popping the parameters of procedures
Base Register	Employed to relocate the programs in memory specially in multi-programming systems
Index	Accessing iterative local variables such as arrays
Stack	Used for local variables

1.4.1 Immediate Addressing

When an operand is interpreted as an **immediate** value, e.g. LOAD IMMEDIATE 7, it is the actual value 7 that is put in the CPU register. In this mode the operand is the data in operand address field of the instruction. Since there is no address field at all, and hence no additional memory accesses are required for executing this instruction. In other words, in this addressing scheme, the actual operand D is A, the content of the operand field: i.e. D = A. The effective address in this case is not defined.

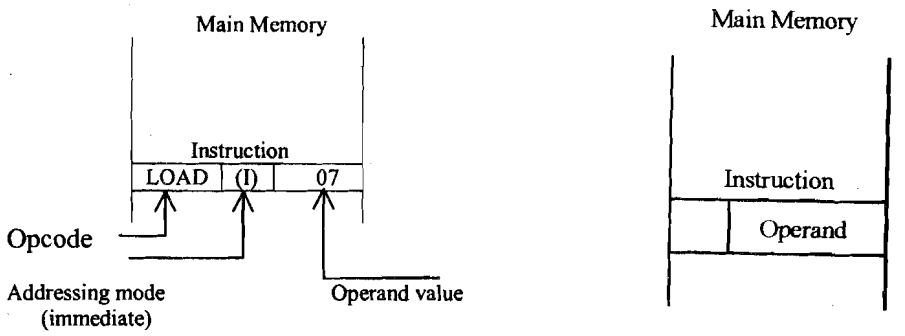


Figure 8: Immediate Addressing

Salient points about the addressing mode are:

- This addressing mode is used to initialise the value of a variable.
- The advantage of this mode is that no additional memory accesses are required for executing the instruction.
- The size of instruction and operand field is limited. Therefore, the type of data specified under this addressing scheme is also restricted. For example, if an instruction of 16 bits uses 6 bits for opcode and 2 bits for addressing mode, then 10 bits can be used to specify an operand. Thus, 2^{10} possible values only can be assigned.

1.4.2 Direct Addressing

In this scheme the operand field of the instruction specifies the **direct address** of the intended operand, e.g., if the instruction LOAD 500 uses direct addressing, then it will result in loading the contents of memory cell 500 into the CPU register. In this mode the intended operand is the address of the data in operation. For example, if memory cell 500 contains 7, as in the diagram below, then the value 7 will be loaded to CPU register.

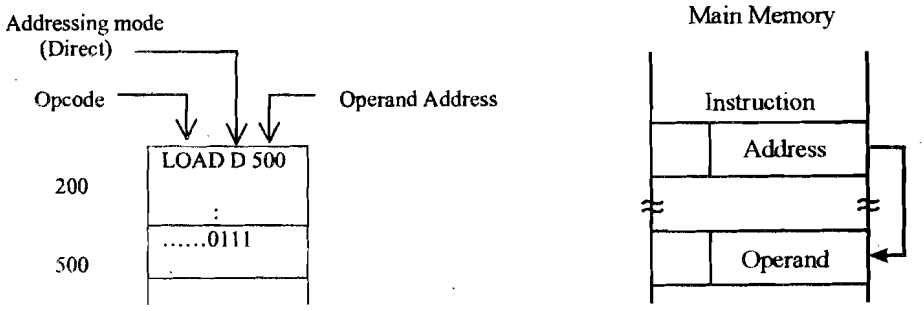


Figure 9: Direct Addressing

Some salient points about this scheme are:

- This scheme provides a limited address space because if the address field has n bits then memory space would contain 2^n memory words or locations. For example, for the example machine of Figure 1, the direct addresses memory space would be 2^{10}.

- The effective address in this scheme is defined as the address of the operand, that is,

  EA ← A and (EA in the above example will be 500)
  D = (EA) (D in the above example will be 7)

  The second statement implies that the data is stored in the memory location specified by effective address.
- In this addressing scheme only one memory reference is required to fetch the operand.

1.4.3 Indirect Addressing

In this scheme the operand field of the instruction specifies the **address** of the **address of** intended operand, e.g., if the instruction LOAD I 500 uses indirect addressing scheme, and contains a value 50A, and memory location 50A contains 7, then the value 7 will get loaded in the CPU register.

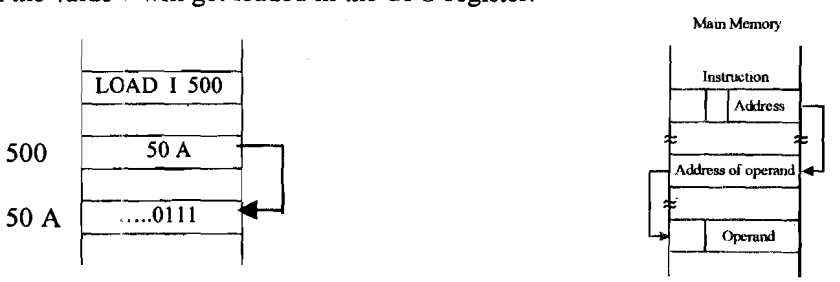


Figure 10: Indirect Addressing

Some salient points about this scheme are:

- In this addressing scheme the effective address EA and the contents of the operand field are related as:

  EA = (A) and (Content of location 500 that is 50A above)
  D = (EA) (Contents of location 50A that is 7)

- The drawback of this scheme is that it requires two memory references to fetch the actual operand. The first memory reference is to fetch the actual address of the operand from the memory and the second to fetch the actual operand using that address.
- In this scheme the word length determines the size of addressable space, as the actual address is stored in a Word. For example, the memory having a word size of 32 bits can have 2^{32} indirect addresses.

1.4.4 Register Addressing

When operands are taken from register(s), implicitly or explicitly, it is called register addressing. These operands are called register operands. If operands are from memory locations, they are called memory operands. In this scheme, a register address is specified in the instruction. That register contains the operand. It is conceptually similar to **direct addressing scheme** except that the register name or number is substituted for memory address. Sometimes the address of register may be assumed implicitly, for example, the Accumulator register in old machines.

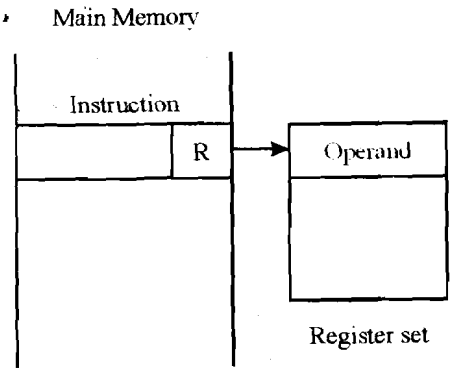


Figure 11: Register Addressing

The major advantages of register addressing are:

- Register access is faster than memory access and hence register addressing results in faster instruction execution. However, register obtains operands only from memory; therefore, the operands that should be kept in registers are selected carefully and efficiently. For example, if an operand is moved into a register and processed only once and then returned to memory, then no saving occurs. However if an operand is used repeatedly after bringing into register then we have saved few memory references. Thus, the task of using register efficiently deals with the task of finding what operand values should be kept in registers such that memory references are minimised. Normally, this task is done by a compiler of a high level language while translating the program to machine language. As a thumb rule the frequently used local variables are kept in the registers.
- The size of register address is smaller than the memory address. It reduces the instruction size. For example, for a machine having 32 general purpose registers only 5 bits are needed to address a register.

In this addressing scheme the effective address is calculated as:

EA = R
D = (EA)

1.4.5 Register Indirect Addressing

In this addressing scheme, the operand is data in the memory pointed to by a register. In other words, the operand field specifies a register that contains the address of the operand that is stored in memory. This is almost same as indirect addressing scheme except it is much faster than indirect addressing that requires two memory accesses.

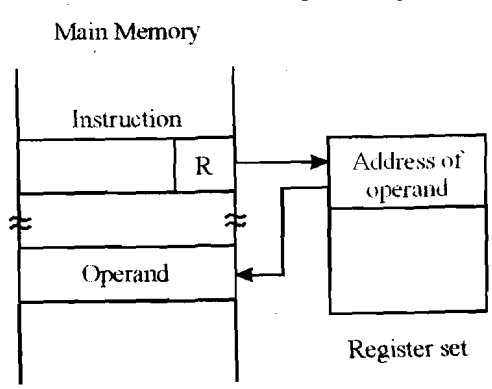


Figure 12: Register Indirect Addressing

The effective address of the operand in this scheme is calculated as:

EA= (R) and

D = (EA)

The address capability of register indirect addressing scheme is determined by the size of the register.

1.4.6 Indexed Addressing Scheme

In this scheme the operand field of the instruction contains an address and an index register, which contains an offset. This addressing scheme is generally used to address the consecutive locations of memory (which may store the elements of an array). The index register is a special CPU register that contains an index value. The contents of the operand field A are taken to be the address of the initial or the reference location (or the first element of array). The index register specifies the distance between the starting address and the address of the operand.

For example, to address of an element B[i] of an array B[1], B[2],....B[n], with each element of the array stored in two consecutive locations, and the starting address of the array is assumed to be 101, the operand field A in the instruction shall contain the number 101 and the index register R will contain the value of the expression $(i - 1) \times 2$.

Thus, for the first element of the array the index register will contain 0. For addressing 5th element of the array, the A=101 whereas index register will contain:

$$(5- 1) \times 2 = 8$$

Therefore, the address of the 5th element of array B is=101+8=109. In B[5], however, the element will be stored in location 109 and 110. To address any other element of the array, changing the content of the index register will suffice.

Thus, the effective address in this scheme is calculated as:

EA = A +(R)
D = (EA)
(DA is Direct address)

As the index register is used for iterative applications, therefore, the value of index register is incremented or decremented after each reference to it. In several systems this operation is performed automatically during the course of an instruction cycle. This feature is known as auto-indexing. Auto indexing can be auto-incrementing or auto-decrementing. The choice of register to be used as an index register differs from machine to machine. Some machines employ general-purpose registers for this purpose while other machines may specify special purpose registers referred to as index registers.

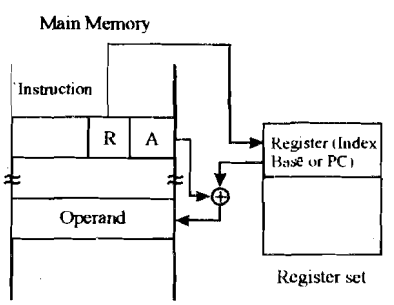


Figure 13: For Displacement Addressing

1.4.7 Base Register Addressing

An addressing scheme in which the content of an instruction specifies base register is added to the displacement field or address field of the instruction. (Refer to Figure

13). The displacement field is taken to be a positive number. For example, if a displacement field is of 8 bits then a memory region of 256 words beginning at the address pointed to by the base register can be addressed by this mode. This is similar to indexed addressing scheme except that the role of Address field and Register is reversed. In indexing Address field of instruction is fixed and index register value is changed, whereas in Base Register addressing, the Base Register is common and Address field of the instruction in various instructions is changed. In this case:

EA = A+ (B)
D = (EA)
(B) Refers to the contents of a base register B.

The contents of the base register may be changed in the privileged mode only. No user is allowed to change the contents of the base register. The base-addressing scheme provides protection of users from one another.

This addressing scheme is usually employed to relocate the programs in memory specially in multiprogramming systems in which only the value of base register requires updating to reflect the beginning of a new memory segment.

Like index register a base register may be a general-purpose register or a special register reserved for base addressing.

1.4.8 Relative Addressing Scheme

In this addressing scheme, the register R is the program counter (PC) containing the address of the current instruction being executed. The operand field A contains the displacement (positive or negative) of an instruction or data with respect to the current instruction. This addressing scheme has advantages if the memory references are nearer to the current instruction being executed. (Please refer to the Figure 13).

Let us give an example of Index, Base and Relative addressing schemes.

Example 1: What would be the effective address and operand value for the following LOAD instructions:

(i) LOAD IA 56 R1 Where IA indicates index addressing, R1 is index register and 56 is the displacement in Hexadecimal.

(ii) LOAD BA 46 B1 Where BA indicates base addressing, B1 is base register and 46 is the displacement specified in instruction in Hexadecimal notation.

(iii) LOAD RA 36 Where RA specifies relative addressing.

The values of registers and memory is given below:

Register	Value
PC	2532_H
Index Register (R1)	2752_H
Base Register (B1)	2260_H

Values of Memory Location

27A8	10_H
	:
	:
2568_H	70_H
	:
	:
$22A6_H$	25_H
	:

The values are shown in the following table:

Addressing Mode	Formulae for addressing mode	EA	Data Value
Index Addressing	EA = A+(R) D= (EA)	$56 + 2752 = 27A8_H$	10_H
Base Addressing	EA = A+ (B)	$46 + 2260 = 22A6_H$	25_H
Relative Addressing	EA = (PC) + A	$2532 + 36 = 2568_H$	70_H

1.4.9 Stack Addressing

In this addressing scheme, the operand is implied as top of stack. It is not explicit, but implied. It uses a CPU Register called Stack Pointer (SP). The SP points to the top of the stack i.e. to the memory location where the last value was **pushed**. A stack provides a sort-of indirect addressing and indexed addressing. This is not a very common addressing scheme. The operand is found on the top of a stack. In some machines the top two elements of stack and top of stack pointer is kept in the CPU registers, while the rest of the elements may reside in the memory. Figure 14 shows the stack addressing schemes.

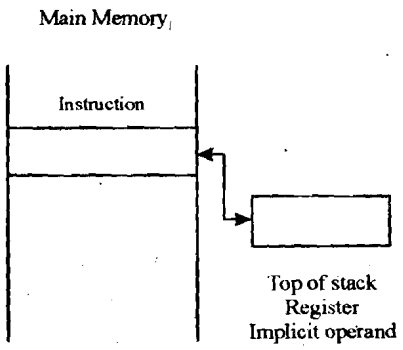


Figure 14: Stack Addressing

Check Your Progress 3

1. What are the numbers of memory references required to get the data for the following addressing schemes:

   (i) Immediate addressing
   (ii) Direct addressing
   (iii) Indirect addressing
   (iv) Register Indirect addressing
   (v) Stack addressing.

2. What are the advantages of Base Register addressing scheme?

3. State True or False. T F

   (i) Immediate addressing is best suited for initialization of variables. ☐

   (ii) Index addressing is used for accessing global variables. ☐

   (iii) Indirect addressing requires fewer memory accesses than that of direct addressing. ☐

   (iv) In stack addressing, operand is explicitly specified. ☐

1.5 INSTRUCTION SET AND FORMAT DESIGN ISSUES

Some of the basic issues of concerns for instruction set design are:

Completeness: For an initial design, the primary concern is that the instruction set should be complete which means there is no missing functionality, that is, it should include instructions for the basic operations that can be used for creating any possible execution and control operation.

Orthogonal: The secondary concern is that the instructions be orthogonal, that is, not unnecessarily redundant. For example, integer operation and floating number operation usually are not considered as redundant but different addressing modes may be redundant when there are more instructions than necessary because the CPU takes longer to decode.

An instruction format is used to define the layout of the bits allocated to these elements of instructions. In addition, the instruction format explicitly or implicitly indicates the addressing modes used for each operand in that instruction.

Designing of instruction format it is a complex art. In this section, we will discuss about the design issues for instruction sets of the machines. We will discuss only point wise details of these issues.

1.5.1 Instruction Length

Significance: It is the basic issue of the format design. It determines the richness and flexibility of a machine.

Basic Tardeoff: Smaller instruction (less space) Versus desire for more powerful instruction repertoire.

Normally programmer desire:

- More op-code and operands: as it results in smaller programs
- More addressing modes: for greater flexibility in implementing functions like table manipulations, multiple branching.

However, a 32 bit instruction although will occupy double the space and can be fetched at double the rate of a 16 bit instruction, but can not be doubly useful.

Factors, which must be considered for deciding about instruction length

Memory size	: if larger memory range is to be addressed, then more bits may be required in address field.
Memory organization	: if the addressed memory is virtual memory then memory range which is to be addressed by the instruction is larger than physical memory size.
Memory transfer length	: instruction length should normally be equal to data bus length or multiple of it.
Memory transfer	: the data transfer rate from the memory ideally should be equivalent to the processor speed. It can become a bottleneck if processor executes instructions faster than the rate of fetching the instructions. One solution for such problem is to use cache memory or another solution can be to keep instruction short.

Normally an instruction length is kept as a multiple of length of a character (that is 8 bits), and equal to the length of fixed-point number. The term word is often used in this context. Usually the word size is equal to the length of fixed point number or equal to memory-transfer size. In addition, a word should store integral number of characters. Thus, word size of 16 bit, 32 bit, 64 bit are to be coming very common and hence the similar length of instructions are normally being used.

1.5.2 Allocation of Bits Among Opcode and Operand

The tradeoff here is between the numbers of bits of opcode versus the addressing capabilities. An interesting development in this regard is the development of variable length opcode.

Some of the factors that are considered for selection of addressing bits:

- **Number of addressing modes**: The more are the explicit addressing modes the more bits are needed for mode selection. However, some machines have implicit modes of addressing.
- **Number of operands**: Fewer number of operand references in an instruction although require less bits yet result in longer programs. Present day machines generally have two operand references in an instruction. Each of these operands may need a addressing mode indicator field.
- **Register addressing versus memory addresses**: The register references require fewer bits in comparison to the memory addresses. In general, the number of user visible registers provided is 16 to 32. Some of these registers may be used for special purposes.
- **Granularity of address**: As far as memory references are concerned, granularity implies whether an address is referencing a byte or a word at a time. This is more relevant for machines, which have 16 bits, 32 bits and higher bits words. Byte addressing although may be better for character manipulation, however, requires more bits in an address. For example, memory of 4K words (1 word = 16 bit) is to be addressed directly then it requires:

WORD Addressing	= 4K words = 2^{12} words $\Rightarrow$ 12 bits are required for word addressing.
Byte Addressing	= 2^{12} words = 2^{13} bytes $\Rightarrow$ 13 bits are required for byte addressing.

1.5.3 Variable-Length of Instructions

With the better understanding of computer instruction sets, the designers came up with the idea of having a variety of instruction formats of different length. What could be the advantages of such a set? The advantages of such a scheme are:

- Large number of operations can be provided which have different lengths of instructions.
- Flexibility in addressing scheme can be provided efficiently and compactly.

However, the basic disadvantage of such a scheme is to have a complex CPU.

An important aspect about these variables length instructions is: "The CPU is not aware about the length of next instruction which is to be fetched". This problem can be handled if each instruction fetch is made equal to the size of the longest instruction. Thus, sometimes in a single fetch multiple instructions can be fetched.

1.6 EXAMPLE OF INSTRUCTION FORMAT

Let us provide you a basic example by which you may be able to define the concept of instruction format.

MIPS 2000

Let's consider the instruction format of a MIPS computer. **MIPS** is an acronym for **Microprocessor without Interlocked Pipeline Stages.** It is a microprocessor architecture developed by MIPS Computer Systems Inc. most widely known for developing the MIPS architecture. The MIPS CPU family was one of the most successful and flexible CPU designs throughout the 1990s. The MIPS CPU has a five-stage CPU pipeline to execute multiple instructions at the same time. Now what we have introduced is a new term Pipelining. What else: the 5 stage pipeline, let us just introduce it here. It defines the 5 steps of execution of instructions that may be performed in an overlapped fashion. The following diagram will elaborate this concept:

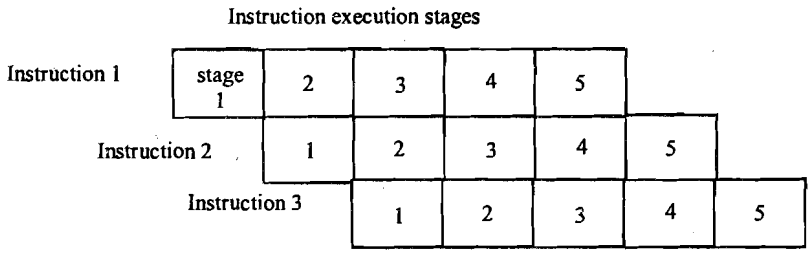


Figure15: Pipeline

Please note that in the above figure:

- All the stages are independent and distinct, that is, the second stage execution of Instruction 1 should not hinder Instruction 2.
- The overall efficiency of the system becomes better.

The early MIPS architectures had 32-bit instructions and later versions have 64-bit implementations.

The first commercial MIPS CPU model, the **R2000**, whose instruction format is discussed below, has thirty-two 32-bit registers and its instructions are 32 bits long.

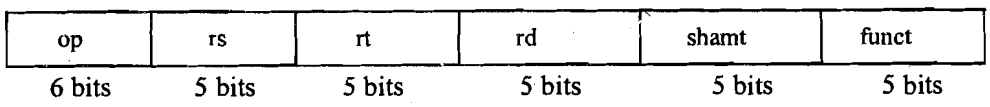


Figure 16: A Sample Instruction Format of MIPS instruction

The meaning of each field in MIPS instruction is given below:

- op : operation code or opcode
- rs : The first register source operand
- rt : The second register source operand
- rd : The destination register operand, stores the result of the operation
- shamt : used in case of shift operations
- funct : This field selects the specific variant of the operation in the opcode field, and is sometimes referred to as function code.

All MIPS instructions are of the same length, requiring different kinds of instruction formats for different types of instructions.

Instruction Format

All MIPS instructions are of the same size and are 32 bits long. MIPS designers chose to keep all instructions of the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, R-type (register) or R-format is used for arithmetic instructions (Figure 16). A second type of instruction format is called i-type or i-format and is used by the data transfer instructions.

Instruction format of I-type instructions is given below:

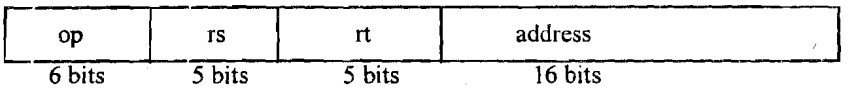


Figure 17: I-format of RISC

The 16-bit address means a load word instruction can load any word within a region of $+2^{15}$ of the base register **rs**. Consider a load word instruction given below:

The rt field specifies the destination register, which receives the result of the load.

MIPS Addressing Modes

MIPS uses various addressing modes:

1. Uses Register and Immediate addressing modes for operations.
2. Immediate and Displacement addressing for Load and Store instructions. In displacement addressing, the operand is at the memory location whose address is the sum of a register.

Check Your Progress 4

1. State True or False. T F

   (i) Instruction length should normally be equal to data bus length or multiple of it. ☐

   (ii) A long instruction executes faster than a short instruction. ☐

   (iii) Memory access is faster than register access. ☐

   (iv) Large number of opcodes and operands result in bigger program. ☐

   (v) A machine can use at the most one addressing scheme. ☐

   (vi) Large number of operations can be provided in the instruction set, which have variable-lengths of instructions. ☐

1.7 SUMMARY

In this unit, we have explained various concepts relating to instructions. We have discussed the significance of instruction set, various elements of an instruction, instruction set design issues, different types of ISAs, various types of instructions and various operations performed by the instructions, various addressing schemes. We have also provided you the instruction format of MIPS machine. Block 4 Unit 1

contains a detailed instruction set of 8086 machine. You can refer to further reading for instruction set of various machines.

1.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. True
2. True
3. False
4. True
5. False

Check Your Progress 2

1. (a) - (iii) (b) - (i) (c) - (ii)
2.
   - Speed up of instruction execution as stores temporary results in registers
   - Less code to execute
   - Larger instruction set
   - Difficult for compiler writing

3. (i) - b), d), f) ; (ii) - a), c) ; (iii) - g) ; (iv) - e)

Check Your Progress 3

1.
   a) Immediate addressing - 0 memory access
   b) Direct addressing - 1 memory access
   c) Indirect addressing - 2 memory accesses
   d) Register Indirect addressing - 1 memory access
   e) Stack addressing - 1 memory access

2. It allows reallocation of program on reloading. It allows protection of users from one another memory space.

3. (i) True.
   (ii) False.
   (iii) False.
   (iv) False

Check Your Progress 4

1.
   (i) True.
   (ii) False.
   (iii) False.
   (iv) False.
   (v) False.
   (vi) True.