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

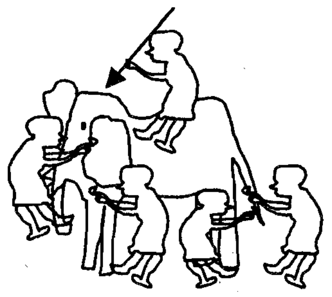
After going through this unit you should be able to:

- 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.

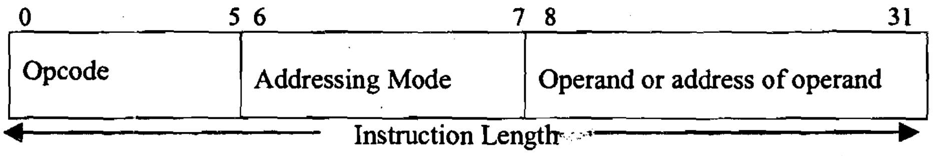


Figure 1: A Hypothetical Instruction Format of 32 bits

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

T	F

State True or False.

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:

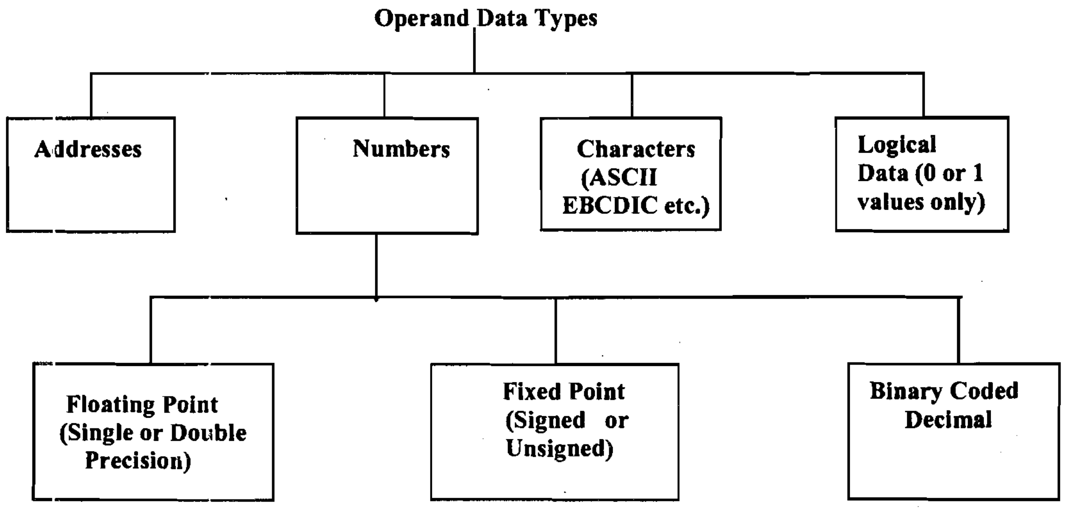

Operand Data Types
Addresses
Numbers
Characters (ASCII EBCDIC etc.)
Logical Data (0 or 1 values only)
Floating Point (Single or Double Precision)
Fixed Point (Signed or Unsigned)
Binary Coded Decimal

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:

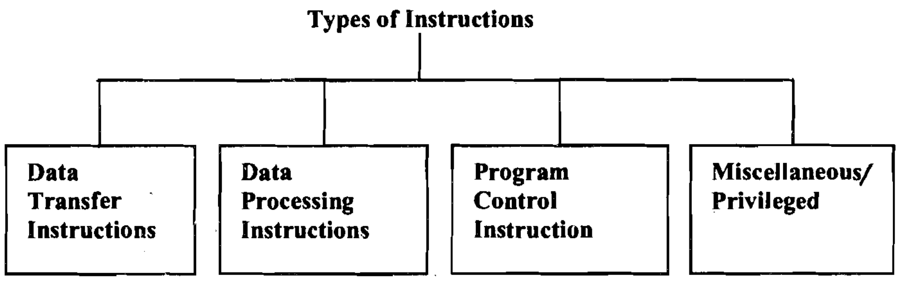

Types of Instructions
Data Transfer Instructions
Data Processing Instructions
Program Control Instruction
Miscellaneous/ Privileged

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

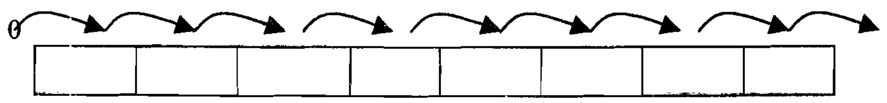
Then,

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.

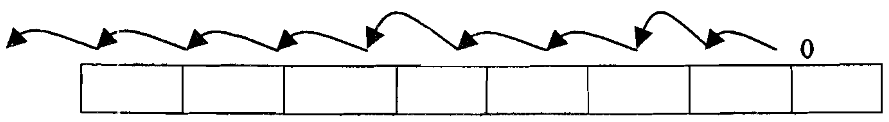
R_1 XOR R_2 = 0100 0111

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.

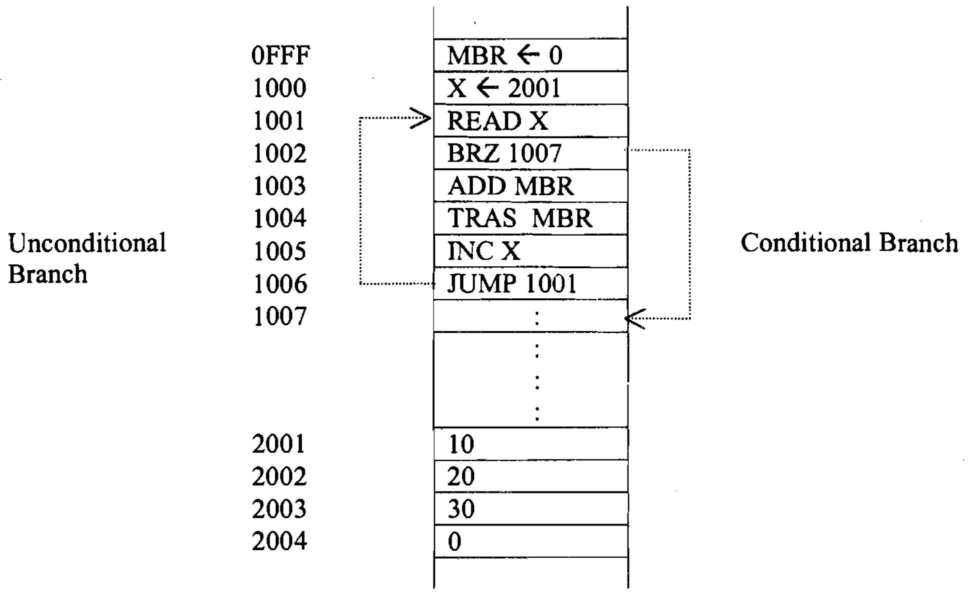
Program Control Instructions

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)

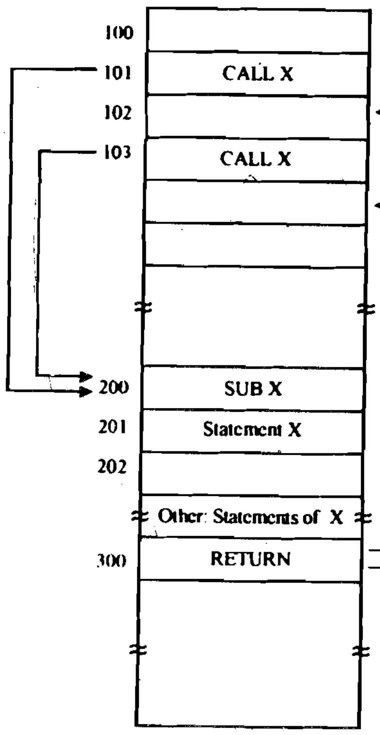
The **SKIP** instruction is a zero-address instruction and skips the next instruction to be executed in sequence. In other words, it increments the value of PC by one instruction length. The SKIP can also be conditional. For example, the instruction ISZ skips the next instruction only if the result of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

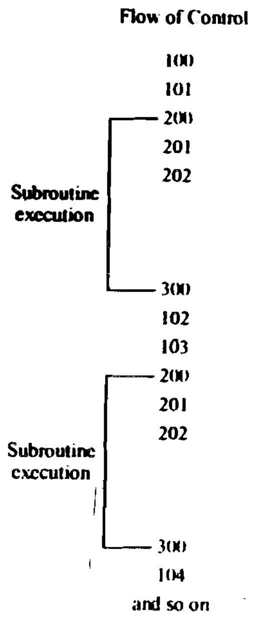
CALL:

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

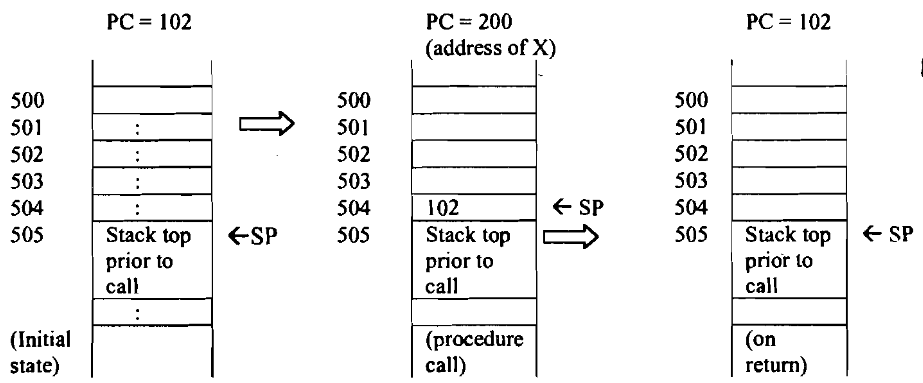
1. Decrement the stack pointer so that we will not overwrite last thing put on stack,
(SP ← SP – 1)



(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:

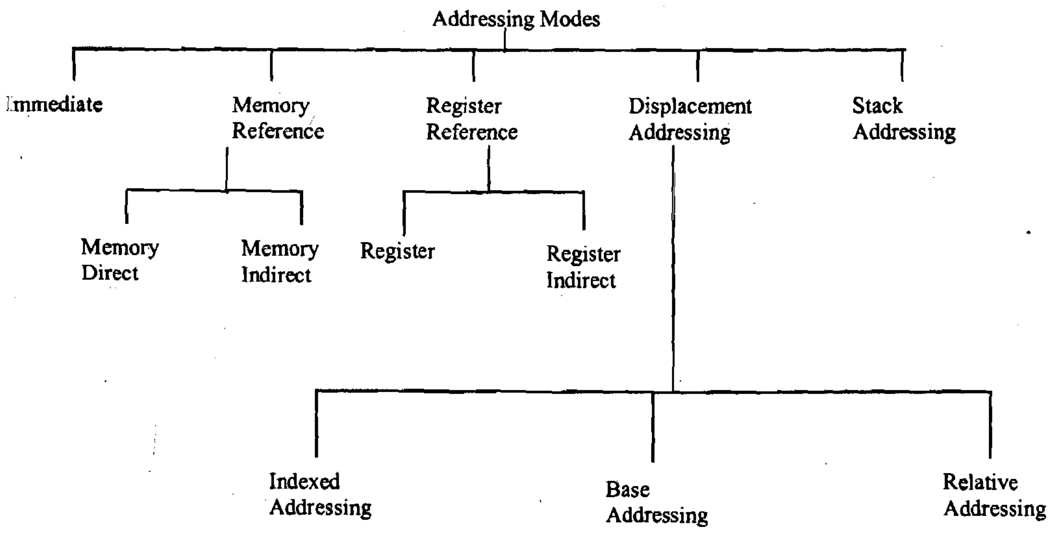


Figure 7: Common Addressing Modes

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.

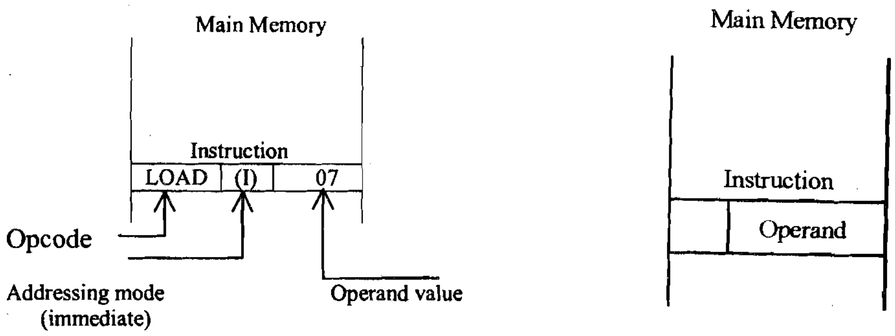


Figure 8: Immediate Addressing

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.

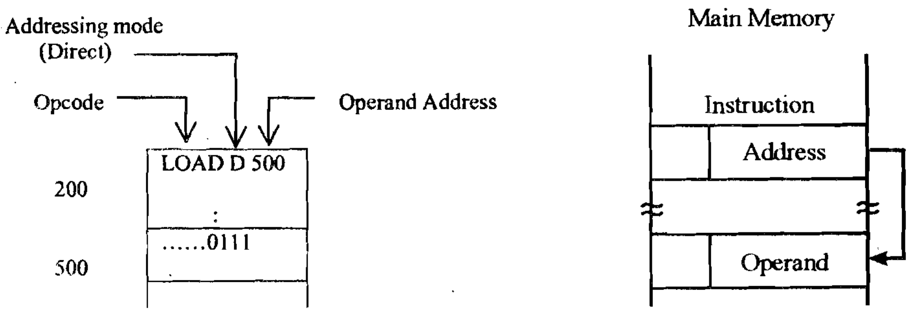


Figure 9: Direct Addressing

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}.

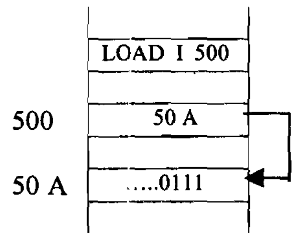
- The effective address in this scheme is defined as the address of the operand, that is,

  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.



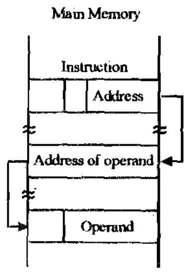


Figure 10: Indirect Addressing

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.

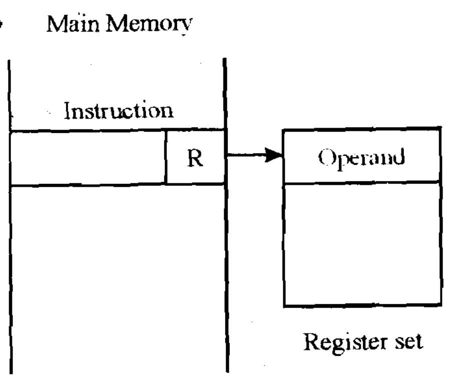


Figure 11: Register Addressing

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.

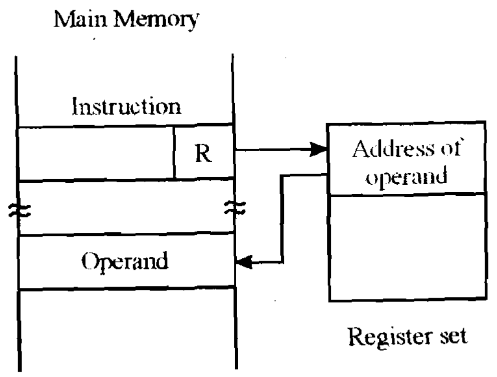


Figure 12: Register Indirect Addressing

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.

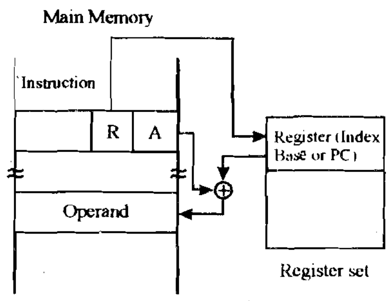


Figure 13: For Displacement Addressing

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.

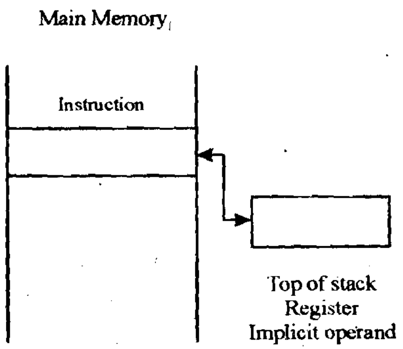


Figure 14: Stack Addressing

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:

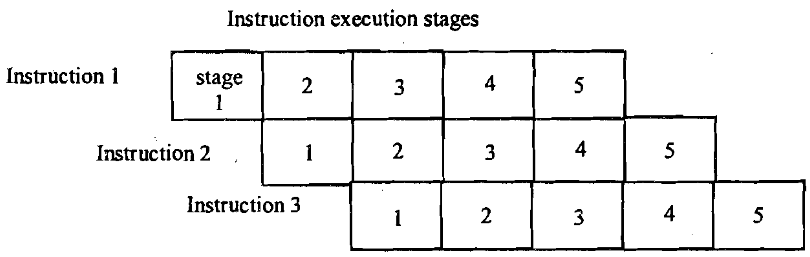


Figure15: Pipeline

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.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	5 bits

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:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

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.

UNIT 2 REGISTERS, MICRO-OPERATIONS AND INSTRUCTION EXECUTION

Structure **Page No.**

2.0 INTRODUCTION

The main task performed by the CPU is the execution of instructions. In the previous unit, we have discussed about the instruction set of computer system. But, one thing, which remained unanswered is: how these instructions will be executed by the CPU?

The above question can be broken down into two simpler questions. These are:

What are the steps required for the execution of an instruction? How are these steps performed by the CPU?

The answer to the first question lies in the fact that each instruction execution consists of several steps. Together they constitute an instruction cycle. A micro-operation is the smallest operation performed by the CPU. These operations put together execute an instruction.

For answering the second question, we must have an understanding of the basic structure of a computer. As discussed earlier, the CPU consists of an Arithmetic Logic Unit, the control unit and operational registers. We will be discussing the register organisation in this unit, whereas the arithmetic-logic unit and control unit organisation are discussed in subsequent units.

In this unit we will first discuss the basic CPU structure and the register organisation in general. This is followed by a discussion on micro-operations and their implementation. The discussion on micro-operations will gradually lead us towards the discussion of a very simple ALU structure. The detail of ALU structure is the topic of the next unit.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the register organisation of the CPU;

- define what is a micro-operation;
- differentiate among various micro-operations;
- discuss an instruction execution using the micro-operations; and
- define the concepts of instruction pipelining.

2.2 BASIC CPU STRUCTURE

A computer manipulates data according to the instructions of a stored program. **Stored program** means the program and data are stored in the same memory unit. The central processing unit, also referred to as CPU, performs the bulk of the data processing operations. It has three main components:

1. A set of registers for holding binary information.
2. An arithmetic and logic unit (ALU) for performing data manipulation, and
3. A control unit that coordinates and controls the various operations and initiates the appropriate sequence of micro-operations for each task.

Computer instructions are normally stored in consecutive memory locations and are executed in sequence one by one. The control unit allows reading of an instruction from a specific address in memory and executes it with the help of ALU and Register.

Instruction Execution and Registers

The basic process of instruction execution is:

1. Instruction is fetched from memory to the CPU registers (called instruction fetch) under the control unit.
2. It is decoded by the control unit and converted into a set of lower level control signals, which cause the functions specified by that instruction to be executed.
3. After the completion of execution of the current instruction, the next instruction fetched is the next instruction in sequence.

This process is repeated for every instruction except for program control instructions, like branch, jump or exception instructions. In this case the next instruction to be fetched from memory is taken from the part of memory specified by the instruction, rather than being the next instruction in sequence.

But why do we need Registers?

If t_{cpu} is the cycle time of CPU that is the time taken by the CPU to execute a well-defined micro-operation using registers, and t_{mem} is the memory cycle time, that is the speed at which the memory can be accessed by the CPU, then (t_{cpu}/t_{mem}) is in the range of 2 to 10, that is CPU is 2 – 10 times faster than memory. Thus, CPU registers are the fastest temporary storage areas. Thus, the instructions whose operands are stored in the fast CPU registers can be executed rapidly in comparison to the instructions whose operands are in the main memory of a computer. Each instruction must designate the registers it will address. Thus, a machine requires a large number of registers.

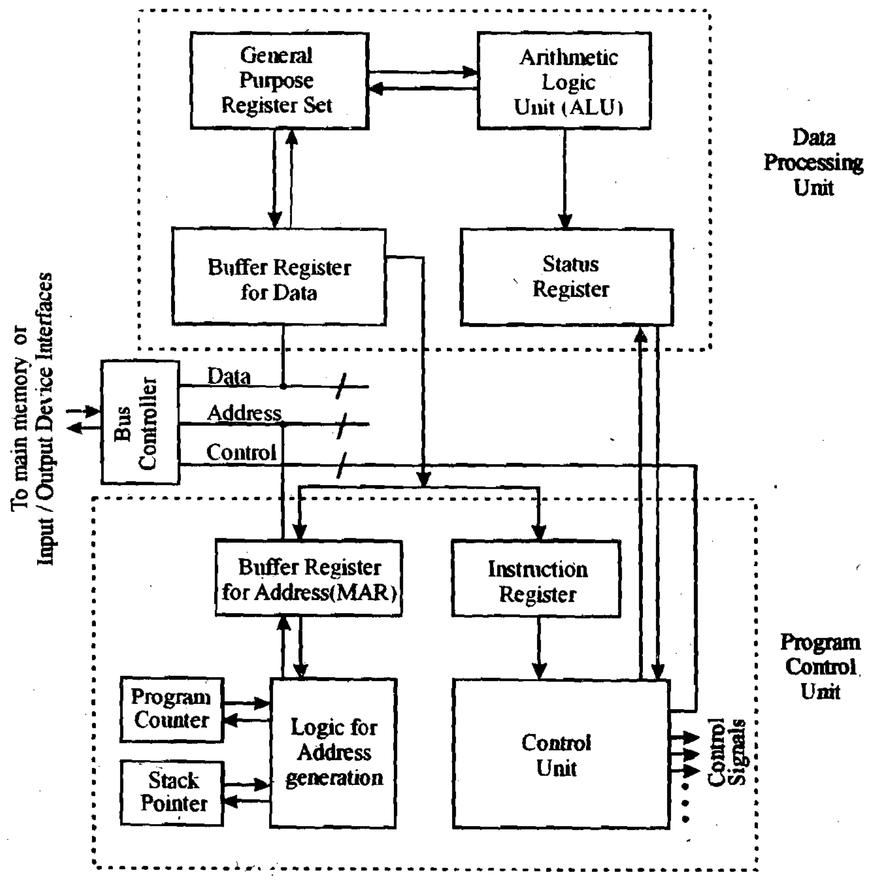


Figure 1: CPU with general register organisation

But how do the registers help in instruction execution? We will discuss this with the help of Figure 1.

Step 1:

The first step of instruction execution is to fetch the instruction that is to be executed. To do so we require:

- Address of the "instruction to be fetched". Normally Program counter (PC) register stores this information.
- Now this address is converted to physical machine address and put on address bus with the help of a buffer register sometimes called Memory Address Register (MAR).
- This, coupled with a request from control unit for reading, fetches the instruction on the data bus, and transfers the instruction to Instruction Register (IR).
- On completion of fetch PC is incremented to point to the next instruction.

In Step 2:

- The IR is decoded; let us assume that Instruction Register contains an instruction. ADD Memory location B with general purpose register R1 and store result in R1, then control unit will first instruct to:
  - Get the data of memory location B to buffer register for data (DR) using buffer address register (MAR) by issuing Memory read operation.
  - This data may be stored in a general purpose register, if so needed let us say R2

- Now, ALU will perform addition of R1 & R2 under the command of control unit and the result will be put back in R1. The status of ALU operation for example result in zero/non zero, overflow/no overflow etc. is recorded in the status register.

- Similarly, the other instructions are fetched and executed using ALU and register under the control of the Control Unit.

Thus, for describing instruction execution, we must describe the registers layout, micro-operations, ALU design and finally the control unit organization. We will discuss registers and micro- operation in this unit. ALU and Control Unit are described in Unit 3 and Unit 4 of this Block.

2.3 REGISTER ORGANISATION

The number and the nature of registers is a key factor that differentiates among computers. For example, Intel Pentium has about 32 registers. Some of these registers are special registers and others are general-purpose registers. Some of the basic registers in a machine are:

- All von-Neumann machines have a program counter (PC) (or instruction counter IC), which is a register that contains the address of the next instruction to be executed.
- Most computers use special registers to hold the instruction(s) currently being executed. They are called instruction register (IR).
- There are a number of general-purpose registers. With these three kinds of registers, a computer would be able to execute programs.
- Other types of registers:
  - Memory-address register (MAR) holds the address of next memory operation (load or store).
  - Memory-buffer register (MBR) holds the content of memory operation (load or store).
  - Processor status bits indicate the current status of the processor. Sometimes it is combined with the other processor status bits and is called the program status word (PSW).

A few factors to consider when choosing the number of registers in a CPU are:

- CPU can access registers faster then it can access main memory.
- For addressing a register, depending on the number of addressable registers a few bit addresses is needed in an instruction. These address bits are definetly quite less in comparison to a memory address. For example, for addressing 256 registers you just need 8 bits, whereas, the common memory size of 1MB requires 20 address bits, a difference of 60%.
- Compilers tend to use a small number of registers because large numbers of registers are very difficult to use effectively. A general good number of registers is 32 in a general machine.
- Registers are more expensive than memory but far less in number.

From a user's point of view the register set can be classified under two basic categories.

Programmer Visible Registers: These registers can be used by machine or assembly language programmers to minimize the references to main memory.

Status Control and Registers: These registers cannot be used by the programmers but are used to control the CPU or the execution of a program.

Different vendors have used some of these registers interchangeably; therefore, you should not stick to these definitions rigidly. Yet this categorization will help in better understanding of register sets of machine. Therefore, let us discuss more about these categories.

2.3.1 Programmer Visible Registers

These registers can be accessed using machine language. In general we encounter four types of programmer visible registers.

- General Purpose Registers
- Data Registers
- Address Registers
- Condition Codes Registers.

A comprehensive example of registers of 8086 is given in Unit 1 Block 4.

The general-purpose registers as the name suggests can be used for various functions. For example, they may contain operands or can be used for calculation of address of operand etc. However, in order to simplify the task of programmers and computers dedicated registers can be used. For example, registers may be dedicated to floating point operations. One such common dedication may be the data and address registers.

The data registers are used only for storing intermediate results or data and not for operand address calculation.

Some dedicated address registers are:

Segment Pointer	: Used to point out a segment of memory.
Index Register	: These are used for index addressing scheme.
Stack Pointer	: Points to top of the stack when programmer visible stack addressing is used.

One of the basic issues with register design is the number of general-purpose registers or data and address registers to be provided in a microprocessor. The number of registers also affects the instruction design as the number of registers determines the number of bits needed in an instruction to specify a register reference. In general, it has been found that the optimum number of registers in a CPU is in the range 16 to 32. In case registers fall below the range then more memory reference per instruction on an average will be needed, as some of the intermediate results then have to be stored in the memory. On the other hand, if the number of registers goes above 32, then there is no appreciable reduction in memory references. However, in some computers hundreds of registers are used. These systems have special characteristics. These are called Reduced Instruction Set Computers (RISC) and they exhibit this property. RISC computers are discussed later in this unit.

What is the importance of having less memory references? As the time required for memory reference is more than that of a register reference, therefore the increased number of memory references results in slower execution of a program.

Register Length: An important characteristic related to registers is the length of a register. Normally, the length of a register is dependent on its use. For example, a register, which is used to calculate address, must be long enough to hold the maximum possible addresses. If the size of memory is 1 MB than a minimum of 20 bits are required to store an instruction address. Please note how this requirement can be optimized in 8086 in the block 4. Similarly, the length of data register should be long enough to hold the data type it is supposed to hold. In certain cases two consecutive registers may be used to hold data whose length is double of the register length.

2.3.2 Status and Control Registers

For control of various operations several registers are used. These registers cannot be used in data manipulation; however, the content of some of these registers can be used by the programmer. One of the control registers for a von-Neumann machine is the Program Counter (PC).

Almost all the CPUs, as discussed earlier, have a status register, a part of which may be programmer visible. A register which may be formed by condition codes is called condition code register. Some of the commonly used flags or condition codes in such a register may be:

Flag	Comments
Sign flag	This indicates whether the sign of previous arithmetic operation was positive (0) or negative (1).
Zero flag	This flag bit will be set if the result of the last arithmetic operation was zero.
Carry flag	This flag is set, if a carry results from the addition of the highest order bits or borrow is taken on subtraction of highest order bit.
Equal flag	This bit flag will be set if a logic comparison operation finds out that both of its operands are equal.
Overflow flag	This flag is used to indicate the condition of arithmetic overflow.
Interrupt	This flag is used for enabling or disabling interrupts. Enable/disable flag.
Supervisor flag	This flag is used in certain computers to determine whether the CPU is executing in supervisor or user mode. In case the CPU is in supervisor mode it will be allowed to execute certain privileged instructions.

These flags are set by the CPU hardware while performing an operation. For example, an addition operation may set the overflow flag or on a division by 0 the overflow flag can be set etc. These codes may be tested by a program for a typical conditional branch operation. The condition codes are collected in one or more registers. RISC machines have several sets of conditional code bits. In these machines an instruction specifies the set of condition codes which is to be used. Independent sets of condition code enable the provisions of having parallelism within the instruction execution unit.

The flag register is often known as Program Status Word (PSW). It contains condition code plus other status information. There can be several other status and control registers such as interrupt vector register in the machines using vectored interrupt, stack pointer if a stack is used to implement subroutine calls, etc.

Check Your Progress 1

1. What is an address register?

....

....

....

2. A machine has 20 general-purpose registers. How many bits will be needed for register address of this machine?

....

....

....

3. What is the advantage of having independent set of conditional codes?

....

....

....

3. Can we store status and control information in the memory?

....

....

....

Let us now look into an example register set of MIPS processor.

2.4 GENERAL REGISTERS IN A PROCESSOR

In Block 4 Unit 1, you would be exposed to 8086 registers. In this section we will provide very brief details of registers of a RISC system called MIPS.

MIPS is a register-to-register or load/store architecture and uses three address instructions for data manipulation. It is because of register-register operands that you can have more operands in an instruction of 32 bits, as register address are smaller. The MIPS have 32 addressable registers = $2^5 \Rightarrow 5$ bits register address. The table given below displays the MIPS general purpose registers.

MIPS register names begin with a $. There are two naming conventions:

- By number:

  $0 $1 $2 ... $31

- By (mostly) two-letter names, such as:

  $a0 - $a3 $t0 - $t7 $s0 - $s7 $gp $fp $sp $ra

Not all of these are general-purpose registers. The following table describes how each general register is treated, and the actions you can take with each register.

Name	Register number	Description	Specify in Expression
ZERO	0	Always has the value 0.	$zero
AT	1	Reserved for the assembler to handle large constants.	$at
V0 - V1	2-3	Function value registers. Values for results and expression evaluation.	$v0 - $v1
A0 - A3	4-7	Argument registers.	$a0 - $a3

T0 - T7	8-15	Temporary registers	$t0 - $t7
S0 - S7	16-23	Saved registers	$s0 - $s7
T8 - T9	24-25	Temporary registers	$t8 - $t9
K0 - K1	26-27	Reserved for the operating system	$k1 - $k2
GP	28	Global pointer register	$gp
SP	29	Stack pointer register	$sp
FP	30	Frame pointer register	$fp
RA	31	Return address register	$ra

You will also study another 8086 based register organization in Block 4 of this course. So, all the computers have a number of registers. But, how exactly is the instruction execution related to registers? To explore this concept, let us first discuss the concept of Micro-operations.

2.5 MICRO-OPERATION CONCEPTS

We have discussed the general architecture and register set of MIPS microprocessor. Our next task is to look at the functionality of ALU, the control unit and how an instruction is executed. In this section, we will define a micro-operation concept, which is the key concept to describe instruction execution.

A micro-operation is an elementary operation performed normally during one clock pulse. On the information stored in one or more registers. The result of the operation may replace the previous content of a register or is transferred to a new register or a memory location.

A digital system performs a sequence of micro-operations on data stored in registers or memory. The specific sequence of micro-operations performed is predetermined for an instruction. Thus, an instruction is a binary code specifying a definite sequence of micro-operations to perform a specific function.

For example, a C program instruction sum = sum + 7, will first be converted to equivalent assembly program:

- Move data from memory location "sum" to register R1 (LOAD R1, sum)
- Add an immediate operand to register (R1) and store the results in R1 (ADD R1, 7)
- Store data from register R1 to memory location "sum" (STORE sum, R1).

Thus, several machine instructions may be needed (this will vary from machine to machine) to execute a simple C statement. But, how will each of these machine statements be executed with the help of micro-operations? Let us try to elaborate the execution steps:

- Fetch the instructions.
    - Pass the address of Program Counter (PC) to Memory Address Register (MAR).
    - Issue the memory read operation to fetch instruction in the Buffer Register for data, such as M(BR).

- Increment Program Counter to refer to next instruction in sequence and bring instruction to Instruction Register (IR).

- Execute the instruction
  - Decode the instruction to ascertain operation.
  - As one of the operands is already available in R1 register and the second operand is an immediate operand so fetch operand step is not required. The immediate operand is available in the address part of the instruction.
  - Perform the ALU based addition with R1 and buffer register, store the result in R1.

Thus, we may have to execute the instruction in several steps. For the subsequent discussion, for simplicity, let us assume that each micro-operation can be completed in one clock period, although some micro-operations require memory read/write that may take more time.

Let us first discuss the type of micro-operations. The most common micro-operations performed in a digital computer can be classified into four categories:

1) Register transfer micro-operations: simply transfer binary information from one register to another.
2) Arithmetic micro-operations: perform simple arithmetic operations on numeric data stored in registers.
3) Logic micro-operations: perform bit manipulation (logic) operations on non-numeric data stored in registers.
4) Shift micro-operations registers: perform shift operations on data stored in registers.

2.5.1 Register Transfer Micro-operations

These micro-operations, as the name suggests transfer information from one register to another. The information does not change during these micro-operations. A register transfer micro-operation may be designed as: R1 ← R2. The ← symbol implies that the contents of register R2 are transferred to register R1. R2 here is a source register while R1 is a destination register. We will use this notation throughout this section. Please note the following important points about register transfer micro-operations.

- For a register transfer micro-operation there must be a path for data transfer from the output of the source register to the input of destination register.
- In addition, the destination register should have a parallel load capability, as we expect the register transfer to occur in a predetermined control condition. We will discuss more about the control unit in Unit 4 of this block.
- A common path for connecting various registers is through a common internal data bus of the processor. In general the size of this data bus should be equal to the number of bits in a general register.

The convention used to represent the micro-operations is:

1. Computer register names are designated by capital letters (sometimes followed by numerals) to denote its function. For example, R1, R2 (General Purpose Registers), AR (Address Register), IR (Instruction Register) etc.

2. The individual bits within a register are numbered from 0 (rightmost bit) to n-1 (leftmost bit) as shown in Figure 2b). Common ways of drawing the block diagram of a computer register are shown below. The name of the 16-bit register is IR (Instruction Register) which is partitioned into two subfields in Figure 2d). Bits 0 through 7 are assigned the symbol L (for Low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The symbol IR (L) refers to the low-order byte and IR (H) refers to high-order byte.

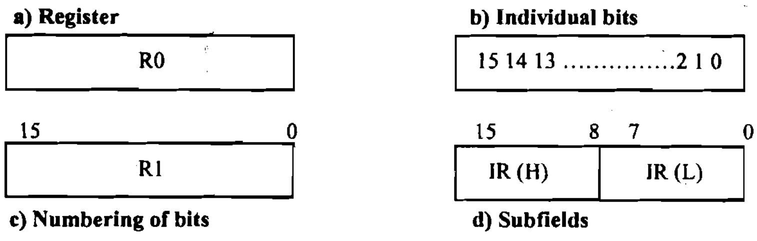


Figure 2: Register Formats

3. Information transfer from one register to another is designated in symbolic notation by a replacement operator. For example, the statement R2 ← R1 denotes a transfer of all bits from the source register R1 to the destination register R2 during one clock pulse and the destination register has a parallel load capacity. However, the contents of register R1 remain unchanged after the register transfer micro-operation. More than one transfer can be shown using a comma operator.

4. If the transfer is to occur only under a predetermined control condition, then this condition can be specified as a control function. For example, if P is a control function then P is a Boolean variable that can have a value of 0 or 1. It is terminated by a colon (:) and placed in front of the actual transfer statement. The operation specified in the statement takes place only when P = 1. Consider the statements:

   If (P =1) then (R2 ← R1)
   or,
   P: R2 ← R1,

   Where P is a control function that can be either 0 or 1.

5. All micro-operations written on a single line are to be executed at the same time provided the statements or a group of statements to be implemented together are free of conflict. A conflict occurs if two different contents are being transferred to a single register at the same time. For example, the statement: new line X: R1← R2, R1← R3 represents a conflict because both R2 and R3 are trying to transfer their contents to R1 at the same time.

6. A clock is not included explicitly in any statements discussed above. However, it is assumed that all transfers occur during the clock edge transition immediately following the period when the control function is 1. All statements imply a hardware construction for implementing the micro-operation statement as shown below:

   Implementation of controlled data transfer from R2 to R1 only when T = 1
   T : R1 ← R2

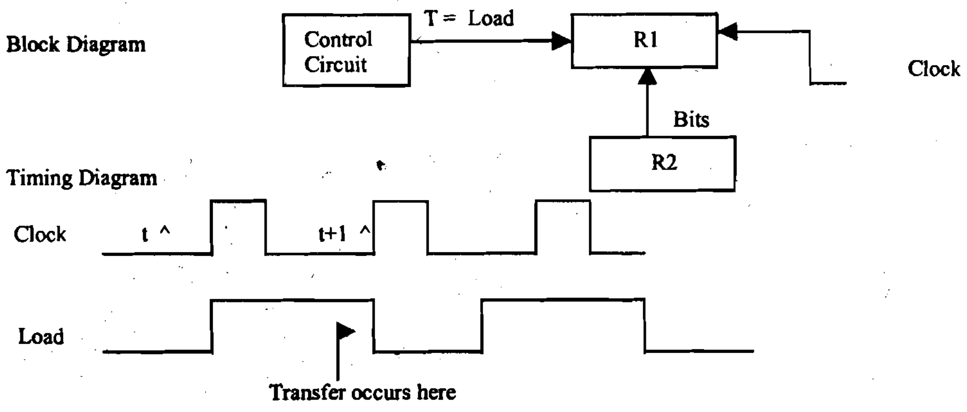


Figure 3: The Register Transfer Time

It is assumed that the control variable is synchronized with the same clock as the one applied to the register. The control function T is activated by the rising edge of the clock pulse at time t. Even though the control variable T becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time t+1. At time t+1, load input is again active and the data inputs of R2 are then loaded into the register R1 in parallel. The transfer occurs with every clock pulse transition while T remains active.

Bus and Memory Transfers

A digital computer has many registers, and rather than connecting wires between all registers to transfer information between them, a common bus is used. Bus is a path (consists of a group of wires) one for each bit of a register, over which information is transferred, from any of several sources to any of several destinations.

From a register to Bus: BUS $\leftarrow$ R. The implementation of bus is explained in Unit 3 of this block.

The transfer from bus to register can be expressed symbolically as:

R1 $\leftarrow$ BUS,

The content of the selected register is placed on the BUS, and the content of the bus is loaded into register R1 by activating its load control input.

Memory Transfer

The transfer of information from memory to outside world i.e., I/O Interface is called a *read* operation. The transfer of new information to be stored in memory is called a *write* operation. These kinds of transfers are achieved via a system bus. It is necessary to supply the address of the memory location for memory transfer operations.

Memory Read

The memory unit receives the address from a register, called the memory address register designated by MAR. The data is transferred to another register, called the data register designated by DR. The read operation can be stated as:

Read: DR $\leftarrow$ [MAR]

Memory Write

The memory write operation transfers the content of a data register to a memory word M selected by the address. Assume that the data of register R1 is to be written to the memory at the address provided in MAR. The write operation can be stated as:

Write: [MAR] $\leftarrow$ R1

Please note, it means that the location pointed by MAR will be written and not MAR.

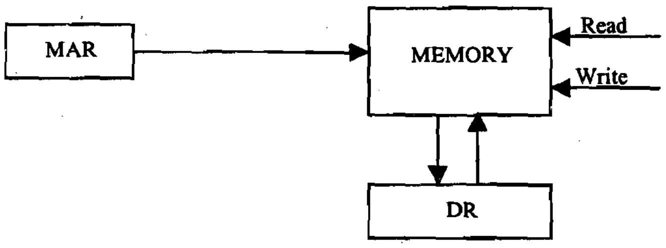


Figure 4: Memory Transfer

2.5.2 Arithmetic Micro-operations

These micro-operations perform simple arithmetic operations on numeric data **stored in registers**. The basic arithmetic micro-operations are addition, subtraction, increment, decrement, and shift.

Addition micro-operation is specified as:

$$R3 \leftarrow R1 + R2$$

It means that the contents of register R1 are added to the contents of register R2 and the sum is transferred to register R3. This operation requires three registers to hold data along with the Binary Adder circuit in the ALU. Binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any lengths and is constructed with full-adder circuits connected in cascade. An n-bit binary adder requires n full-adders. Add micro-operation, in accumulator machine, can be performed as:

$$AC \leftarrow AC + DR$$

Subtraction is most often implemented in machines through complement and adds operations. It is specified as:

R3 ← R1 − R2
R3 ← R1 + (2's complement of R2)
R3 ← R1 + (1's complement of R2 + 1)
R3 ← R1 + R2 + 1 (The bar on top of R2 implies 1's complement of R2 which is bitwise complement)

Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting the contents of R2 from R1 and storing the result in R3. We will describe the basic circuit required for these micro-operations in the next unit.

The increment micro-operation adds one to a number in a register. This operation is designated as:

$$R1 \leftarrow R1 + 1$$

This can be implemented in hardware by using a binary-up counter.

The decrement micro-operation subtracts one from a number in a register. This operation is designated as:

$$R1 \leftarrow R1 - 1$$

This can be implemented using binary-down counter.

What about the multiply and division operations? Are not they micro-operations? In most of the older computers multiply and divisions were implemented using add/subtract and shift micro-operations. If a digital system has implemented division and multiplication by means of combinational circuits, then we can call these as the micro-operations for that system.

2.5.3 Logic Micro-operations

Logic operations are basically binary operations, which are performed on the string of bits stored in the registers. For a logic micro-operation each bit of a register is treated as a variable. A logic micro-operation:

R1 ← R1.R2 specifies AND operation to be performed on the contents of R1 and R2 and store the results in R1. For example, if R1 and R2 are 8 bits registers and:

R1 contains	10010011 and
R2 contains	01010101

Then R1 will contain 00010001 after AND operation.

Some of the common logic micro-operations are AND, OR, NOT or Complement, Exclusive OR, NOR, and NAND. In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented.

Let us now discuss how these four micro-operations can be used in implementing some of the important applications of manipulation of bits of a word, such as, changing some bit values or deleting a group of bits. We are assuming that the result of logic micro-operations go back to Register R1 and R2 contains the second operand.

We will play a trick with the manipulations we are performing. Let us select 1010 as 4 bit data for register R1, and 1100 data for register R2. Why? Because if you see the bit combinations of R2, and R1, they represent the truth table entries (read from right to left and bottom to top) 00, 01, 10 and 11. Thus, the resultant of the logical operation on them will indicate which logic micro-operation is needed to be performed for that data manipulation. The following table gives details on some of these operations:

R1	1	0	1	0
R2	1	1	0	0

Operation name	What is the operation?	Example and Explanation

Selective Set	Sets those bits in Register R1 for which the corresponding R2 bit is 1.	R1 = 1010
		R2 = 1100
		1110
		The value 1110 suggests that selective set can be done using logic OR micro-operation. Please note that all those bits of R1, for which we have 0 bit in R2, have remained unchanged. The bits in R1 which need to be set selectively must have the corresponding R2 bits as 1.

Selective Clear	Clear those bits in register R1 for which corresponding R2 bits are 1.	R1 = 1010
		R2 = 1100
		0010
		The R1 value after the operation is 0010 which suggests that Corresponding micro-operation is R1 AND $\overline{R2}$

Selective Complement	Complement those bits in register R1 for which the corresponding R2 bits are 1.	R1 = 1010
		R2 = 1100
		0110
		The R1, value 0110 after the operation suggests that the selective complement can be done using exclusive - OR micro-operation. The bits in R1 which need to be complemented selectively must have the corresponding R2 bits as 1.

Mask Operations	Clears those bits in Register R1 for which the corresponding R2	R1 = 1010
		R2 = 1100
		1000
		The R1 value after the operation is 1000

	bits are 0.	which suggests that the mask operation can be performed using AND micro-operation. However, the bits in R1 which are cleared or masked correspond to the bits on R2 having a 0 value. The mask operation is preferred over selective clear as most of the computers provide AND micro-operation while the micro-operation required for implementing selective clear is normally not provided in computers
Insert	For inserting a new value in a bit. It is a two-step process: *Step 1*: Mask out the existing bit value *Step 2*: Insert the bit using OR micro-operation with the bits which are to be inserted.	This is a two-step process. *Example:* Say contents of R1 = 0011 1011 Suppose, we want to insert 0110 in place of left most 0011 then: 0011 1011 (R1 before) 0000 1111 (R2 for masking) ——— Perform AND operation (mask) 0000 1011 (R1 after) Now insert: 01100000 (R2 for insertion) ——— Perform OR operation 0110 1011 R1 after insert

Clear	Clear all the bits	R1 = 1101
		R2 = 1101
		0000
		Implemented by taking exclusive OR with the same number. The exclusive OR, thus, can also be used for checking whether two numbers are equal or not.

2.5.4 Shift Micro-operations

Shift is a useful operation, which can be used for serial transfer of data. Shift operations can also be used along with other (arithmetic, logic, etc.) operations. For example, for implementing a multiply operation arithmetic micro-operation (addition) can be used along with shift operation. The shift operation may result in shifting the contents of a register to the left or right. In a shift left operation a bit of data is input at the right most flip-flop while in shift right a bit of data is input at the left most flip-flop. In both the cases a bit of data enters the shift register. Depending on what bit enters the register and where the shift out bit goes, the shifts are classified in three types. These are:

- logical
- arithmetic and
- circular.

In logical shift the data entering by serial input to left most or right most flip-flop (depending on right or left shift operations respectively) is a 0.

If we connect the serial output of a shift register to its serial input then we encounter a circular shift. In circular shift left or circular shift right information is not lost, but is circulated.

In arithmetic shift a signed binary number is shifted to the left or to the right. Thus, an arithmetic shift-left causes a number to be multiplied by 2, on the other hand a shift-right causes a division by 2. But as in division or multiplication by 2 the sign of a number should not be changed, therefore, arithmetic shift must leave the sign bit unchanged. We have already discussed about shift operations in the Unit 1.

Let us summarize micro-operations using the following table:

Sl. No.	Micro-operations	Examples
1.	Register transfer	R1 ← R2 (register transfer) [MAR] ← R1 (Register to memory)
2.	Arithmetic micro-operations	ADD R1 ← R1 + R2 SUBTRACT R1 ← R1 + ($\overline{R2}$ +1) INCREMENT R1 ← R1 +1 DECREMENT R1← R1 – 1
3.	Logical micro operations	AND OR COMPLEMENT XOR
4.	Shift	Left or right shift • Logical • Arithmetic • Circular

Check Your Progress 2

1. How does the memory read / operation carried out using system bus?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. Are multiplication and division arithmetic operations micro-operations?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. What will be the value for R2 operand if:

   (i) Mask operation clears register R1
   (ii) Bits 1011 0001 is to be inserted in an 8 bit R1 register.

4. What are the differences between circular and logical shift micro-operations?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2.6 INSTRUCTION EXECUTIONS AND MICRO - OPERATIONS

Let us now discuss instruction execution using the micro-operations. A simple instruction may require:

- Instruction fetch: fetching the instruction from the memory.
- Instruction decode: decode the instruction.
- Operand address calculation: find out the effective address of the operands.
- Execution: execute the instruction.
- Interrupt Acknowledge: perform an interrupt acknowledge cycle if an interrupt request is pending.

Let us explain how these steps of instruction execution can be broken down to micro-operations. For simplifying the discussion, let us assume that the machine has the structure as shown in Figure 1. In addition, let us also assume that the instruction set of the machine has only two addressing modes direct and indirect memory addresses and a memory access take same time as that of a register access that is one clock cycle.

Instruction fetch: In this phase the instruction is brought from the address pointed by PC to instruction register. The steps required are:

Transfer the address of PC to MAR. (Register Transfer)	MAR ← PC
MAR puts its contents on the address bus for main memory location selection, the control unit instructs the MAR to do so and also uses a memory read signal. The word so read is placed on the data bus where it is accepted by the Data register (Memory-read using bus. It may take more than one clock pulses depending on the t_{cpu} and t_{mem}) The PC is incremented by one memory word length to point to the next instruction in sequence. This micro-operation can be carried out in parallel to the micro-operation above.	DR ← (MAR), PC← PC +1
The instruction so obtained is transferred from data register to the Instruction register for further processing. (Register Transfer)	IR ← DR

Instruction Decode: This phase is performed under the control of the Control Unit of the computer. The Control Unit determines the operation that is to be performed and the addressing mode of the data. In our example, the addressing modes can be direct or indirect.

Operand Address Calculation: In actual machines the effective address may be a memory address, register or I/O port address. The register reference instructions such as complement R1, clear R2 etc. normally do not require any memory reference (assuming register indirect addressing is not being used) and can directly go to the execute cycle. However, the memory reference instruction can use several addressing modes. Depending on the type of addressing the effective address (EA) of operands in the memory is calculated. The calculation of effective address may require more memory fetches (for example in the case of indirect addressing), thus in this step we may calculate the effective address as:

For Direct Address: • Transfer the address portion of instruction is the direct address so no further calculation is needed.	IR (Address) and DR (Address) contain the Effective address.

For Indirect Address:
• Transfer the address bits of instruction to the MAR. This transfer can be achieved using DR, as DR and IR at this point of time contain the same value. (Register Transfer)
• Perform a memory read operation as done in fetch cycle and the desired address of the operand is obtained in the DR. (Memory Read)
• Transfer the address part so obtained in DR as the address part of instruction. (Register Transfer) Thus, the indirect address is now converted to direct address or effective address.
	MAR ← DR (Address)
	
	DR← (MAR)
	
	IR (Address) ← DR (Address)

Thus, the address portion of IR now contains the effective address, which is the direct address of the operand.

Execution: Now the instruction is ready for execution. A different opcode will require different sequence of steps for the execution. Therefore, let us discuss a few examples of execution of some simple instructions for the purpose of identifying some of the steps needed during instruction execution. Let us start the discussions with a simple case of addition instruction. Suppose, we have an instruction: Add R1, A which adds the content of memory location A to R1 register storing the result in R1. This instruction will be executed in the following steps:

Transfer the address portion of the instruction to the MAR. (Register transfer)	MAR ← IR (Address)
Read the memory location A and bring the operand in the DR. (Memory read)	DR ← (MAR)
Add the DR with R1 using ALU and bring the results back to R1. (Add micro-operations)	R1 ← R1 + DR

Now, let us try a complex instruction - a conditional jump instruction. Suppose an instruction:

INCSKIP A

increments A and skips the next instruction if the content of A has become zero. This is a complex instruction and requires intermediate decision-making. The micro operations required for this instruction execution are:

Transfer the address portion of IR to the MAR. (Register transfer)	MAR ← IR (Address)
Read memory. DR on reading will contain the operand A. (Memory read)	DR ← (MAR)
Transfer the contents of DR to R1. We are assuming that DR, although it can be used in computation, it cannot be used as destination of an ALU operation. Thus, we need to transfer its content to a general purpose register R1 where the operation can be performed. (Register transfer)	R1 ← DR
Increment the R1. (Increment micro-operation)	R1 ← R1 +1
Transfer the content of R1 to DR. (Register transfer)	DR ← R1
Store the contents of DR- into the location A using MAR. This operation proceeds through as: Address bits are applied on address bus by MAR. The data is put into the data bus. The control unit providing control signal for memory write, thus resulting in a memory write at a location specified by MAR. (Memory write)	(MAR) ←DR
If the content of R1 is zero then increment PC by one, thus skipping the next instruction. This operation can be performed in parallel to the memory write. Please note in the last step a comparison and an action is taken as a single step. This is possible as it is a simple comparison based on status flags. (Increment on a condition)	If R1 = 0 then PC ← PC + 1

Let us now take an example of branching operation. Suppose we are using the first location of subroutine to store the return address, then the steps involved in this subroutine call (CALL A) can be:

Transfer the contents of address portion of IR to MAR. (Register Transfer)
Transfer the **return address**, that is, the contents of PC to DR. This micro-operation can be performed in parallel to the previous micro-operation. (Register transfer)
	MAR ← IR (Address),
	
	DR ← PC

Transfer the **branch address** that is stored in Address part of the instruction to program counter. (Register transfer)	PC ← IR (Address)
Store the DR using MAR. Thus, the return address is stored at the first location of the subroutine. (This operation normally is done in stack, but in this example we are storing the return address in the first location of the subroutine). This micro-operation can be performed in parallel to previous micro-operation. (Memory write)	(MAR) ←DR
Increment the PC as it contains the first location of subroutine, which is used to store the return address. The first instruction of subroutine starts from the next location. (Increment)	PC ← PC + 1

Thus, the number of steps required in execution may differ from instruction to instruction.

Interrupt Processing: On completion of the execution of an instruction, the machine checks whether there is any pending interrupt request for the interrupts that are enabled. If an enabled interrupt has occurred then that Interrupt may be processed. The nature of interrupt varies from machine to machine. However, let us discuss one simple illustration of interrupt processing events. A simple sequence of steps followed in interrupt phase is:

Transfer the contents of PC to DR, as this is the return address after the interrupt service program has been executed. This address must be saved.	DR ← PC
Place the address of location, where the return address is to be saved, into MAR. Please note that this address is normally predetermined in computers.	MAR ← Address of location for saving return address.
Store the contents of PC in the memory using DR and MAR. (Memory write) Transfer the address of the first instruction of interrupt servicing routine to the PC. This micro-operation can be performed in parallel to the above micro-operation.	(MAR) ←DR PC ← address of the first instruction interrupt service programs

After completing the above interrupt processing, CPU will fetch the next instruction that may be interrupt service program instruction. Thus, during this time CPU might be doing the interrupt processing or executing the user program. Please note each instruction of interrupt service program is executed as an instruction in an instruction cycle.

Please note for a complex machine the instruction cycle will not be as easy as this. You can refer to further readings for more complex instruction cycles.

2.7 INSTRUCTION PIPELINING

After discussing instruction execution, let us now define a concept that is very popular in any CPU implementation. This concept is instruction pipeline.

To extract better performance, as defined earlier, instruction execution can be done through instruction pipeline. The instruction pipelining involves decomposing of an instruction execution to a number of pipeline stages. Some of the common pipeline stages can be instruction fetch (IF), instruction decode (ID), operand fetch (OF), execute (EX), store results (SR). An instruction pipe may involve any combination of such stages. A major design decision here is that the instruction stages should be of equal execution time. Why?

A pipeline allows overlapped execution of instructions. Thus, during the course of execution of an instruction the following may be a scenario of execution.

Time Slot->	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	ID	OF	EX	SR						
Instruction 2		IF	ID	OF	EX	SR					
Instruction 3			IF	ID	OF	EX	SR				
Instruction 4				IF	ID	OF	EX	SR			
Instruction 5					IF	ID	OF	EX	SR		
Instruction 6						IF	ID	OF	EX	SR	
Instruction 7							IF	ID	OF	EX	SR

Figure 5: Instruction Pipeline

Please note the following observations about the above figure:

- The pipeline stages are like steps. Thus, a step of the pipeline is to be complete in a time slot. The size of the time slot will be governed by the stage taking maximum time. Thus, if the time taken in various stages is almost similar, we get the best results.
- The first instruction execution is completed on completion of 5th time slot, but afterwards, in each time slot the next instruction gets executed. So, in ideal conditions one instruction is executed in the pipeline in each time slot.
- Please note that after the 5th time slot and afterwards the pipe is full. In the 5th time slot the stages of execution of five instructions are:

SR	(instruction 1)	(Requires memory reference)
EX	(instruction 2)	(No memory reference)
OF	(instruction 3)	(Requires memory reference)
ID	(instruction 4)	(No memory reference)
IF	(instruction 5)	(Requires memory reference)

The Pipelining Problems:

- On the 5th time slot and later, there may be a register or memory conflict in the instructions that are performing memory and register references that is various stages may refer to same registers/memory location. This will result in slower execution instruction pipeline that is one of the higher number instruction has to wait till the lower number instructions completed, effectively pushing the whole pipelining by one time slot.

- Another important situation in Instruction Pipeline may be the branch instruction. Suppose that instruction 2 is a conditional branch instruction, then by the time the decision to take the branch is taken (at time interval 5) three more instructions have already been fetched. Thus, if the branch is to be taken then the whole pipeline is to be emptied first. Thus, in such cases, pipeline cannot run at full load.

How can we minimize the problems occurring due to the branch instructions?

We can use many mechanisms that may minimize the effect of branch penalty.

- To keep multiple streams in pipeline in case of branch
- Pre-fetching the next as well as instruction to which branch is to take place
- A loop buffer may be used to store the instructions of a loop instruction
- Predicting whether the branch will take place or not and acting accordingly
- Delaying the pipeline fill up till the branch decision is taken.

Check Your Progress 3

State True or False T | F

1) An instruction cycle does not include indirect cycle if the operands are stored in the register. ☐

2) Register transfer micro-operations are not needed for instruction execution. ☐

3) Interrupt cycle results only in jumping to an interrupt service routine. The actual processing of the instructions of this routine is performed in instruction cycle. ☐

2.8 SUMMARY

In this unit, we have discussed in detail the register organisation and a simple structure of the CPU. After this we have discussed in details the micro-operations and their implementation in hardware using simple logical circuits. While discussing micro-operations our main emphasis was on simple arithmetic, logic and shift micro-operations, in addition to register transfer and memory transfer. The knowledge you have acquired about register sets and conditional codes, helps us in giving us an idea that conditional micro-operations can be implemented by simply checking flags and conditional codes. This idea will be clearer after we go through Unit 3 and Unit 4. We have completed the discussions on this unit, with providing a simple approach of instruction execution with micro-operations. We have also defined the concepts of Instruction Pipeline. We will be using this approach for discussing control unit details in Unit 3 and Unit 4. The following table gives the details of various terms used in this unit.

General purpose registers	These registers are used for any address or data computation / storage
Status and control register	Stores the various condition codes
Programmer visible registers	Used by programmers during programming
Micro-operations	Involves register transfer micro operations arithmetic micro-operations like add, subtract, logic micro-operations like AND, OR, NOT, XOR and shift micro-operations left or right shift
Micro-operations and instruction execution	An instruction is executed through a sequence of micro-operations. Thus, a program is executed as a sequence of instruction is executed when a sequence of microinstructions are executed.
Instruction pipeline	Allows overlapped execution of instructions. A good pipe can produce one instruction per clock cycle.

You will also get the details on 8086 microprocessor register sets, conditional codes, instructions etc. in Unit 1 of Block 4.

You can refer to further readings for more register organisation examples and for more details on micro-operations and instruction execution.

2.9 SOLUTIONS /ANSWERS

Check Your Progress 1

1. Registers, which are used only for the calculation of operand addresses, are called address registers.
2. 5 bits
3. It helps in implementing parallelism in the instruction execution unit.
4. Yes. Normally, the first few hundreds of words of memory are allocated for storing control information.

Check Your Progress 2

1. Read operation involves reading of location pointed to by MAR. The address bus is loaded with the contents of MAR
   address BUS ←MAR
   In addition a read signal is issued by control unit, and data is stored to MBR register or data register.
   DR ← data BUS
   The combined operation can be shown as
   DR ← [MAR]

2. Yes, if implemented through circuits.
   No, if implemented through algorithms involving add/ subtract and shift micro-operations.

3. (i) 0000 0000
   (ii) Initially AND with 0000 0000 followed by OR with 1011 0001

4. The bits circulate and after a complete cycle the data is still intact in circular shift. Not so in logical shift.

Check Your Progress 3

1. True
2. False
3. True

UNIT 3 ALU ORGANISATION

Structure **Page No.**

3.0 INTRODUCTION

By now we have discussed the instruction sets and register organisation followed by a discussion on micro-operations and instruction execution. In this unit, we will first discuss the ALU organisation. Then we will discuss the floating point ALU and arithmetic co-processors, which are commonly used for floating point computations.

This unit provides a detailed view on implementation of simple micro-operations that include register–transfer, arithmetic, logic and shift micro-operation. Finally, the construction of a simple ALU is given. Thus, this unit provides you the basic insight into the computer system. The next unit covers details of the control unit. Together these units describe the two most important components of CPU: the ALU and the CU.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- describe the basic organisation of ALU;
- discuss the requirements of a floating point ALU;
- define the term arithmetic coprocessor; and
- create simple arithmetic logic circuits.

3.2 ALU ORGANISATION

As discussed earlier, an ALU performs simple arithmetic-logic and shift operations. The complexity of an ALU depends on the type of instruction set which has been realized for it. The simple ALUs can be constructed for fixed-point numbers. On the other hand the floating-point arithmetic implementation requires more complex control logic and data processing capabilities, i.e., the hardware. Several micro-processor families utilize only fixed-point arithmetic capabilities in the ALUs. For floating point arithmetic or other complex functions they may utilize an auxiliary special purpose unit. This unit is called arithmetic co-processor. Let us discuss all these issues in greater detail in this section.

3.2.1 A Simple ALU Organisation

An ALU consists of circuits that perform data processing micro-operations. But how are these ALU circuits used in conjunction of other registers and control unit? The simplest organisation in this respect for fixed point ALU was suggested by John von Neumann in his IAS computer design (Please refer to Figure 1).

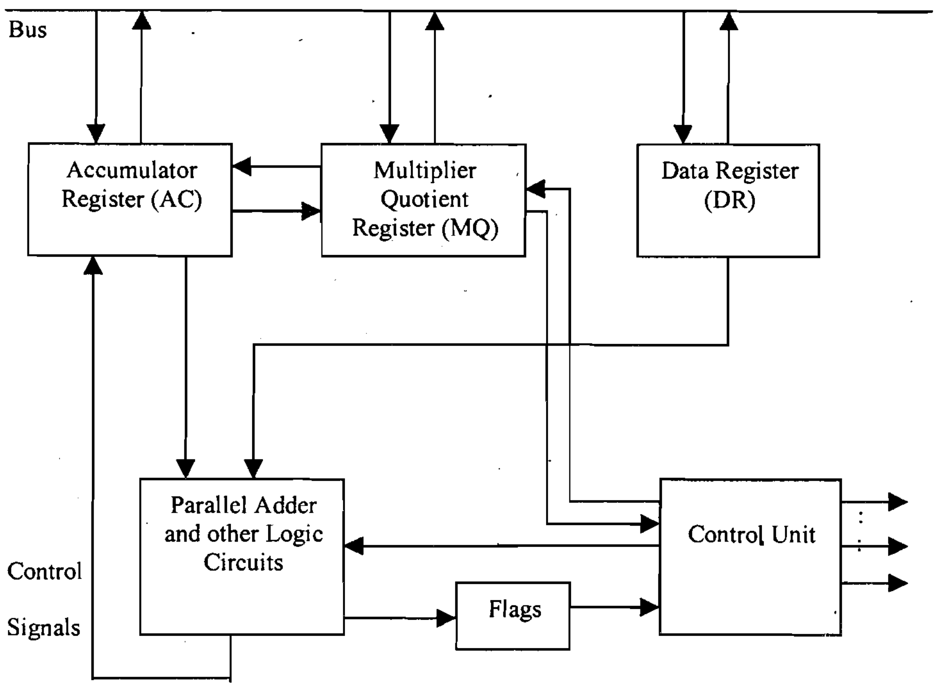

Bus
Accumulator Register (AC)
Multiplier Quotient Register (MQ)
Data Register (DR)
Parallel Adder and other Logic Circuits
Control Unit
Flags
Control
Signals

Figure 1: Structure of a Fixed point Arithmetic logic unit

The above structure has three registers AC, MQ and DR for data storage. Let us assume that they are equal to one word each. Please note that the Parallel adders and other logic circuits (these are the arithmetic, logic circuits) have two inputs and only one output in this diagram. It implies that any ALU operation at most can have two input values and will generate single output along with the other status bits. In the present case the two inputs are AC and DR registers, while output is AC register. AC and MQ registers are generally used as a single AC.MQ register. This register is capable of left or right shift operations. Some of the micro-operations that can be defined on this ALU are:

Addition	: AC ← AC + DR
Subtraction	: AC ← AC – DR
AND	: AC ← AC ^ DR
OR	: AC ← AC v DR
Exclusive OR	: AC ← AC (+) DR
NOT	: AC ← AC

In this ALU organisation multiplication and division were implemented using shift-add/subtract operations. The MQ (Multiplier-Quotient register) is a special register used for implementation of multiplication and division. We are not giving the details of how this register can be used for implementing multiplication and division algorithms. For more details on these algorithms please refer to further readings. One such algorithm is Booth's algorithm and you must refer to it in further readings.

For multiplication or division operations DR register stores the multiplicand or divisor respectively. The result of multiplication or division on applying certain algorithm can finally be obtained in AC.MQ register combination. These operations can be represented as:

Multiplication : AC.MQ $\leftarrow$ DR $\times$ MQ

Division : AC.MQ $\leftarrow$ MQ $\div$ DR

DR is another important register, which is used for storing second operand. In fact it acts as a buffer register, which stores the data brought from the memory for an instruction. In machines where we have general purpose registers any of the registers can be utilized as AC, MQ and DR.

Bit Slice ALUs

It was feasible to manufacture smaller such as 4 or 8 bits fixed point ALUs on a single IC chip. If these chips are designed as expendable types then using these 4 or 8 bit ALU chips we can make 16, 32, 64 bit array like circuits. These are called bit- slice ALUs.

The basic advantage of such ALUs is that these ALUs can be constructed for a desired word size. More details on bit-slice ALUs can be obtained from further readings.

Check Your Progress 1

State True or False

	T	F
1. A multiplication operation can be implemented as a logical operation.	☐	
2. The multiplier-quotient register stores the remainder for a division operation.	☐	
3. A word is processed sequentially on a bit slice ALU.	☐	

3.2.2 A Sample ALU Design

The basis of ALU design starts with the micro-operation implementation. So, let us first explain how the bus can be used for Data transfer micro-operations.

A digital computer has many registers, and rather than connecting wires between all registers to transfer information between them, a common bus is used. Bus is a path (consists of a group of wires) one for each bit of a register, over which information is transferred, from any of several sources to any of several destinations. In general the size of this data bus should be equal to the number of bits in a general purpose register.

A register is selected for the transfer of data through bus with the help of control signals. The common data transfer path, that is the bus, is made using the multiplexers. The select lines are connected to the control inputs of the multiplexers and the bits of one register are chosen thus allowing multiplexers to select a specific source register for data transfer.

The construction of a bus system for four registers using 4×1 multiplexers is shown below. Each register has four bits, numbered 0 through 3. Each multiplexer has 4 data inputs, numbered 0 through 3, and two control or selection lines, C_0 and C_1. The data inputs of 0th MUX are connected to the corresponding 0th input of every register to form four lines of the bus. The 0th multiplexer multiplexes the four 0th bits of the registers, and similarly for the three other multiplexers.

Since the same selection lines C_0 and C_1 are connected to all multiplexers, therefore they choose the four bits of one register and transfer them into the four-line common bus.

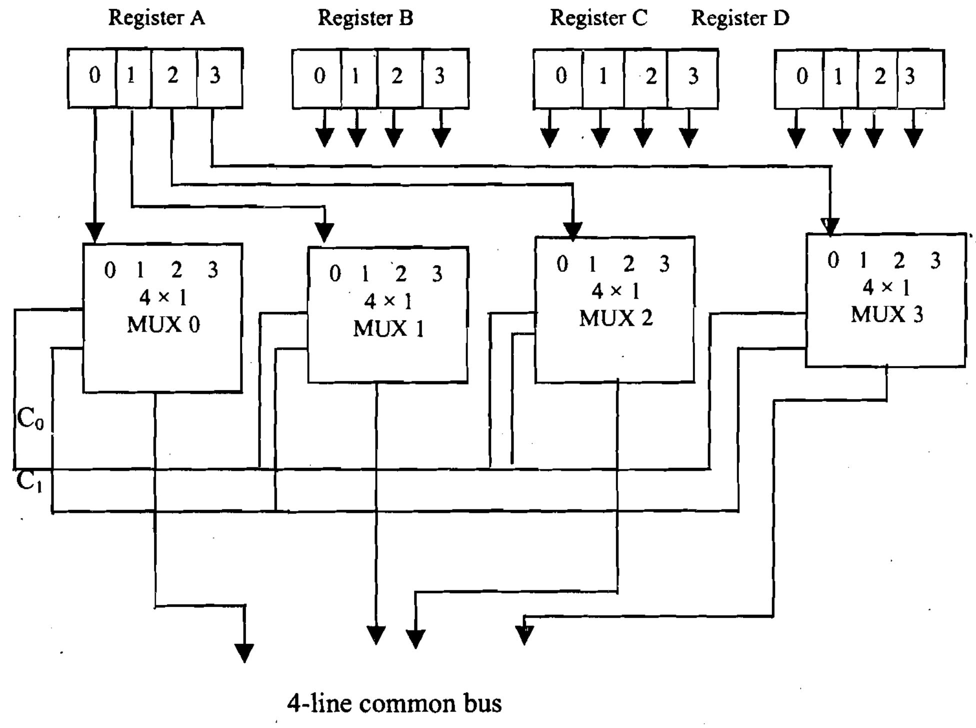


Figure 2: Implementation of BUS

When C_1 C_0 = 00, the 0th data input of all multiplexers are selected and this causes the bus lines to receive the content of register A since the outputs of register A are connected to the 0th data inputs of the multiplexers which is then applied to the output that forms the bus. Similarly, when C_1 C_0 = 01, register B is selected, and so on. The following table shows the register that is selected for each of the four possible values of the selection lines:

C_1	C_0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Figure 3: Bus Line Selection

To construct a bus for 8 registers of 16 bits each, you would require 16 multiplexers, one for each line in the bus. The number of multiplexers needed to construct the bus is equal to the number of bits in each register. Each multiplexer must have eight data input lines and three selection lines ($2^3 = 8$) to multiplex one bit in the eight registers.

Implementation of Arithmetic Circuits for Arithmetic Micro-operation

An arithmetic circuit can be implemented using a number of full adder circuits or parallel adder circuits. Figure 4 shows a logical implementation of a 4-bit arithmetic circuit. The circuit is constructed by using 4 full adders and 4 multiplexers.

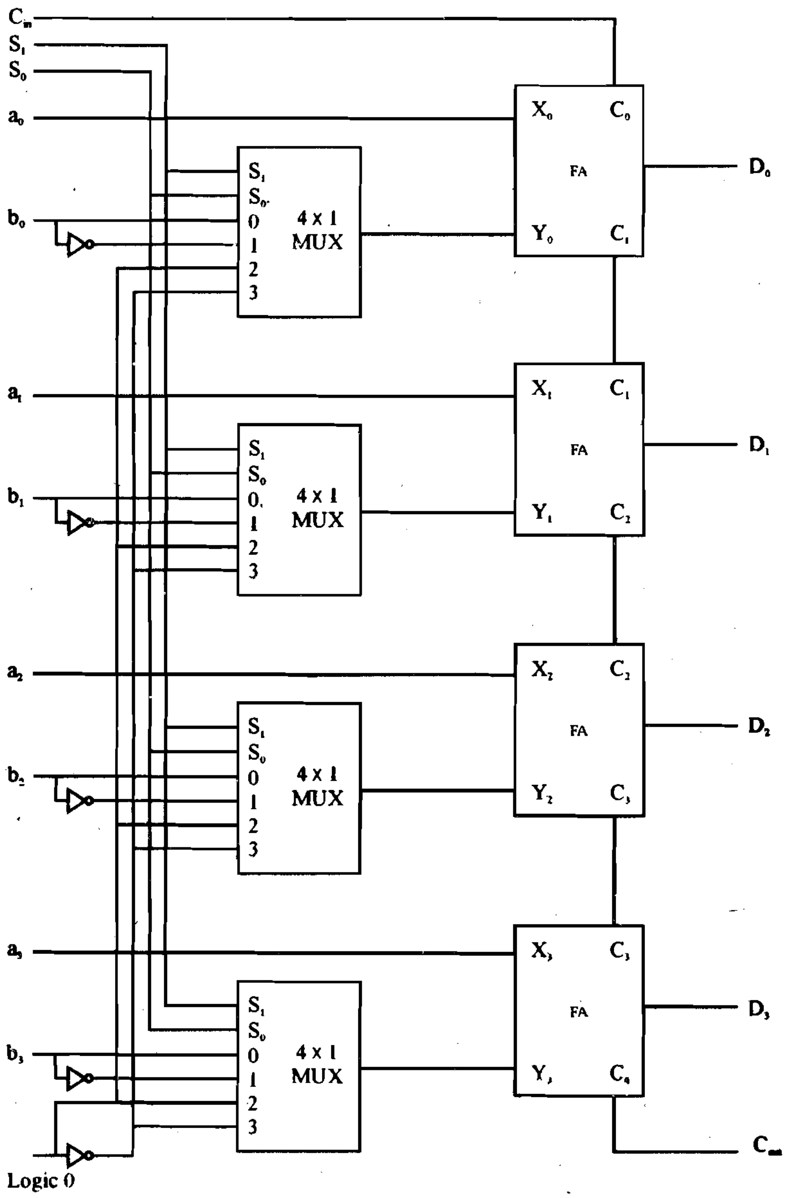


Figure 4: A Four-bit arithmetic circuit

The diagram of a 4-bit arithmetic circuit has four 4×1 multiplexers and four full adders (FA). Please note that the FULL ADDER is a circuit that can add two input bits and a carry-in bit to produce one sum-bit and a carry-out-bit.

So what does the adder do? It just adds three bits. What does the multiplexer do? It controls one of the input bits. Thus, such combination produces a series of micro-operations.

Let us find out how the multiplexer control lines will change one of the Inputs for Adder circuit. Please refer to the following table. (Please note the convention VALID ONLY FOR THE TABLE are that an uppercase alphabet indicates a Data Word, whereas the lowercase alphabet indicates a bit.)

Control Input		Output of 4 × 1 Multiplexers				Y input to Adder	Comments
S_1	S_0	MUX(a)	MUX(b)	MUX(c)	MUX(d)		
0	0	b_0	b_1	b_2	b_3	B	The data word B is input to Full Adders
0	1	$\overline{b_0}$	$\overline{b_1}$	$\overline{b_2}$	$\overline{b_3}$	$\overline{B}$	1's complement of B is input to Full Adders
1	0	0	0	0	0	0	Data word 0 is input to Full Adders
1	1	1	1	1	1	F_H	Data word 1111 = F_H is input to Full Adders

Figure 5: Multiplexer Inputs and Output of the Arithmetic Circuit of Figure 4

Now let us discuss how by coupling carry bit (C_{in}) with these input bits we can obtain various micro-operations.

Input to Circuits

- Register A bits as a_0, a_1, a_2 and a_3 in the corresponding X bits of the Full Adder (FA).
- Register B bits as given in the Figure 5 above as in the corresponding Y bits of the FA.
- Please note each bit of register A and register B is fed to different full adder unit.
- Please also note that each of the four inputs from A are applied to the X inputs of the binary adder and each of the four inputs from B are connected to the data inputs of the multiplexers. It means that the A input directly goes to adder but B input can be manipulated through the Multiplexer to create a number of different input values as given in the figure above. The B inputs through multiplexers are controlled by two selection lines S_1 and S_0. Thus, using various combinations of S_1 and S_0 we can select data bits of B, complement of B, 0 word, or word having All 1's.
- The input carry C_{in}, which can be equal to 0 or 1, goes to the carry input of the full adder in the least significant position. The other carries are cascaded from one stage to the next. Logically it is the same as that of addition performed by us. We do pass the carry of lower digits addition to higher digits. The output of the binary adder is determined from the following arithmetic sum:

$$D = X + Y + C_{in}$$

OR

$$D = A + Y + C_{in}$$

By controlling the value of Y with the two selection lines S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to implement the eight arithmetic micro-operations listed in the truth table.

S_1	S_0	C_{in}	Y val	$D = A+Y+C_{in}$	Equivalent Micro-Operation	Micro-Operation Name
0	0	0	B	D = A + B	R ← R1 + R2	Add
0	0	1	B	D = A + B + 1	R ← R1 + R2 + 1	Add with carry
0	1	0	$\overline{B}$	D =A+B	R ← R1 + $\overline{R2}$	Subtract with borrow
0	1	1	$\overline{B}$	D = A + $\overline{B}$+ 1	R ← R1 + 2's complement of R2	Subtract
1	0	0	0	D = A	R ← R1	Transfer
1	0	1	0	D = A + 1	R ← R1 + 1	Increment
1	1	0	1	D = A – 1	R ← R1 + (All 1s)	Decrement
1	1	1	1	D = A	R ← R1	Transfer

Figure 6: Arithmetic Circuit Function Table

Let us refer to some of the cases in the table above.

When $S_1S_0 = 00$, input line B is enabled and its value is applied to the Y inputs of the full adder. Now,

If input carry $C_{in} = 0$, the output will be D = A + B
If input carry $C_{in} = 1$, the output will be D = A + B + 1.

When $S_1S_0 = 01$, the complement of B is applied to the Y inputs of the full adder. So If $C_{in} = 1$, then output D = A + $\overline{B}$ + 1. This is called subtract micro-operation. (Why?)

Reason: Please observe the following example, where A = 0111 and B=0110, then $\overline{B}$ =1001. The sum will be calculated as:

```
   0111      (Value of A)
   1001      ( Complement of B)
 1 0000 + (Carry in =1) = 0001
```

Ignore the carry out bit. Thus, we get simple subtract operation.

If $C_{in} = 0$, then D = A + $\overline{B}$. This is called subtract with borrow micro-operation. (Why?). Let us look into the same addition as above:

```
   0111     (Value of A)
   1001     ( Complement of B)
 1 0000 + (Carry in =0) = 0000
```

This operation, thus, can be considered as equivalent to:

$$D = A + \overline{B}$$
=> $D = (A - 1) + (\overline{B} + 1)$
=> $D = (A - 1)$ + 2's complement of B
=> $D = (A - 1) - B$ Thus, is the name complement with Borrow

When $S_1S_2 = 10$, input value 0 is applied to Y inputs of the full adder.

If $C_{in} = 0$, then output $D = A + 0 + C_{in}$ => D = A
If $C_{in} = 1$, then D = A + 0 +1 => D = A + 1

The first is a simple data transfer micro-operation; while the second is an increment micro-operation.

When $S_1S_2 = 11$, input word all 1's is applied to Y inputs of the full adder.

If $C_{in} = 0$, then output $D = A + \text{All (1s)} + C_{in} \Rightarrow D = A - 1$ (How? Let us explain with the help of the following example).

Example: Let us assume that the Register A is of 4 bits and contains the value 0101 and it is added to an all (1) value as:

$$\begin{array}{r} 0101 \\ 1111 \\ \hline 1\ 0100 \\ \hline \end{array}$$

The 1 is carry out and is discarded. Thus, on addition with all (1's) the number has actually got decremented by one.

If $C_{in} = 1$, then $D = A + \text{All(1s)} + 1 \Rightarrow D = A$

The first is the decrement micro-operation; while the second is a data transfer micro-operation.

Please note that the micro-operation D = A is generated twice, so there are only seven distinct micro-operations possible through the proposed arithmetic circuit.

Implementation of Logic Micro-operations

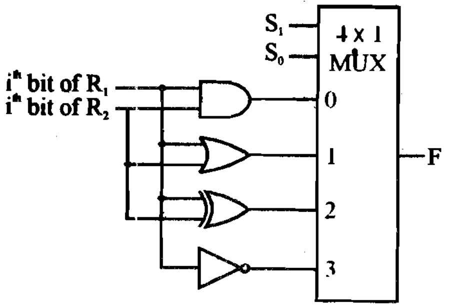
For implementation, let us first ask the questions how many logic operations can be performed with two binary variables. We can have four possible combinations of input of two variables. These are 00, 01, 10, and 11. Now, for all these 4 input combinations we can have $2^4 = 16$ output combinations of truth-values for a function. This implies that for two variables we can have 16 logical operations. The above stated fact will be clearer by going through the following figure.

I_3	I_2	I_1	I_0	Function	Operation	Comments
0	0	0	0	$F_0 = 0$	$R \leftarrow 0$	Clear
0	0	0	1	$F_1 = x.y$	$R \leftarrow R_1 \wedge R_2$	AND
0	0	1	0	$F_2 = x.\overline{y}$	$R \leftarrow R_1 \wedge \overline{R_2}$	R_1 AND with complement R_2
0	0	1	1	$F_3 = x$	$R \leftarrow R_1$	Transfer of R_1
0	1	0	0	$F_4 = \overline{x}.y$	$R \leftarrow \overline{R_1} \wedge R_2$	R_2 AND with complement R_1
0	1	0	1	$F_5 = y$	$R \leftarrow R_2$	Transfer of R_2
0	1	1	0	$F_6 = x \oplus y$	$R \leftarrow R_1 \oplus R_2$	Exclusive OR
0	1	1	1	$F_7 = x + y$	$R \leftarrow R_1 \vee R_2$	OR
1	0	0	0	$F_8 = \overline{(x+y)}$	$R \leftarrow \overline{(R_1 \vee R_2)}$	NOR
1	0	0	1	$F_9 = \overline{(x \oplus y)}$	$R \leftarrow \overline{(R_1 \oplus R_2)}$	Exclusive NOR
1	0	1	0	$F_{10} = \overline{y}$	$R \leftarrow \overline{R_2}$	Complement of R_2
1	0	1	1	$F_{11} = x + \overline{y}$	$R \leftarrow R_1 \vee \overline{R_2}$	R_1 OR with complement R_2
1	1	0	0	$F_{12} = \overline{x}$	$R \leftarrow \overline{R_1}$	Complement of R_1
1	1	0	1	$F_{13} = \overline{x} + y$	$R \leftarrow \overline{R_1} \vee R_2$	R_2 OR with complement R_1
1	1	1	0	$F_{14} = \overline{(x.y)}$	$R \leftarrow \overline{(R_1 \wedge R_2)}$	NAND
1	1	1	1	$F_{15} = 1$	$R \leftarrow$ All 1's	Set all the Bits to 1

Figure 7: Logic micro-operations on two inputs

Please note that in the figure above the micro-operations are derived by replacing the x and y of Boolean function with registers R1 and R2 on each corresponding bit of the registers R1 and R2. Each of these bits will be treated as binary variables.

In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented. The other 12 micro-operations can be derived from these four micro-operations. Figure 8 shows one bit, which is the ith bit stage of the four logic operations. Please note that the circuit consists of 4 gates and a 4 × 1 MUX. The ith bits of Register R1 and R2 are passed through the circuit. On the basis of selection inputs S_0 and S_1 the desired micro-operation is obtained.



(a) Logic Diagram

S_1	S_0	Output	The Operation
0	0	$F = R_1 \wedge R_2$	AND Operation
0	1	$F = R_1 \vee R_2$	OR Operation
1	0	$F = R_1 \oplus R_2$	XOR Operation
1	1	$F = \overline{R_1}$	Complement of Register R_1

(b) Functional representation

Figure 8: Logic diagram of one stage of logic circuit

Implementation of a Simple Arithmetic, Logic and Shift Unit

So, by now we have discussed how the arithmetic and logic micro-operations can be implemented individually. If we combine these two circuits along with shifting logic then we can have a possible simple structure of ALU. In effect ALU is a combinational circuit whose inputs are contents of specific registers. The ALU performs the desired micro-operation as determined by control signals on the input and places the results in an output or destination register. The whole operation of ALU can be performed in a single clock pulse, as it is a combinational circuit. The shift operation can be performed in a separate unit but sometimes it can be made as a part of overall ALU. The following figure gives a simple structure of one stage of an ALU.

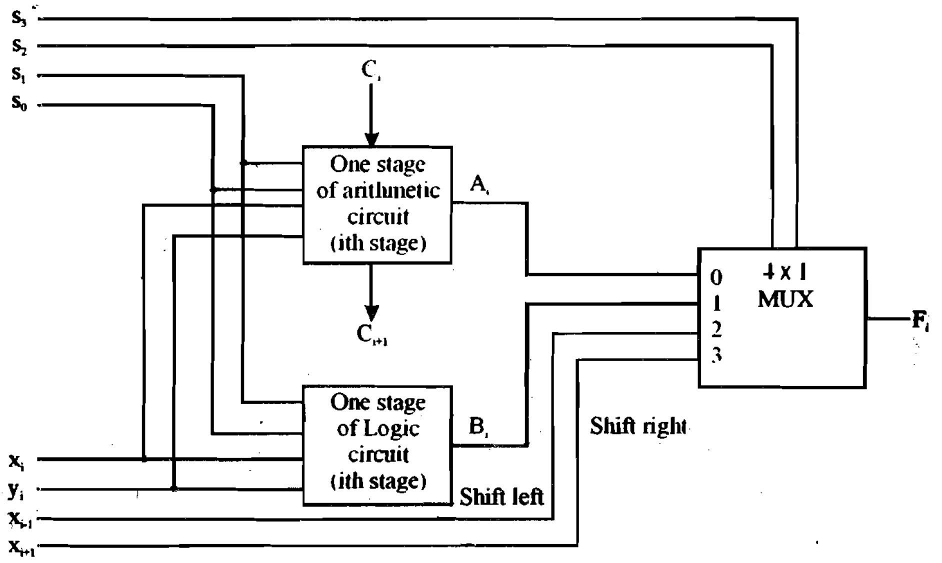


Figure 9: One stage of ALU with shift capability

Please note that in this figure we have given reference to two previous figures for arithmetic and logic circuits. This stage of ALU has two data inputs; the ith bits of the registers to be manipulated. However, the (i – 1)th or (i+1)th bit is also fed for the case of shift micro-operation of only one register. There are four selection lines, which determine what micro-operation (arithmetic, logic or shift) on the input. The F_i is the resultant bit after desired micro-operation. Let us see how the value of F_i changes on the basis of the four select inputs. This is shown in Figure 10:

Please note that in Figure 10 arithmetic micro-operations have both S_3 and S_2 bits as zero. Input C_i is important for only arithmetic micro-operations. For logic micro-operations S_3, S_2 values are 01. The values 10 and 11 cause shift micro-operations. For this shift micro-operation S_1 and S_0 values and C_i values do not play any role.

S_3	S_2	S_1	S_0	C_i	F	Micro-operation	Name	
0	0	0	0	0	$F = x$	$R \leftarrow R_1$	Transfer	Arithmetic Micro-operation
0	0	0	0	1	$F = x+1$	$R \leftarrow R_1+1$	Increment	
0	0	0	1	0	$F = x+y$	$R \leftarrow R_1+R_2$	Addition	
0	0	0	1	1	$F = x+y+1$	$R \leftarrow R_1+R_2+1$	Addition with carry	
0	0	1	0	0	$F = x+\overline{y}$	$R \leftarrow R_1+\overline{R_2}$	Subtract with borrow	
0	0	1	0	1	$F = x+(\overline{y}+1)$	$R \leftarrow R_1 - R_2$	Subtract	
0	0	1	1	0	$F = x-1$	$R \leftarrow R_1 - 1$	Decrement	
0	0	1	1	1	$F = x$	$R \leftarrow R_1$	Transfer	
0	1	0	0	-	$F = x.y$	$R \leftarrow R_1 \wedge R_2$	AND	Logic Micro-operation
0	1	0	1	-	$F = x+y$	$R \leftarrow R_1 \vee R_2$	OR	
0	1	1	0	-	$F = x \oplus y$	$R \leftarrow R_1 \oplus R_2$	Exclusive OR	
0	1	1	1	-	$F = \overline{x}$	$R \leftarrow \overline{R_1}$	Complement	
1	0	-	-	-	$F = Shl(x)$	$R \leftarrow Shl(R_1)$	Shift left	Shift Micro-operations
1	1	-	-	-	$F = Shr(y)$	$R \leftarrow Shr(R_1)$	Shift right	

Figure 10: Micro-operations performed by a Sample ALU

3.3 ARITHMETIC PROCESSORS

The questions in this regard are: "What is an arithmetic processor?" and, "What is the need for arithmetic processors?"

A typical CPU needs most of the control and data processing hardware for implementing non-arithmetic functions. As the hardware costs are directly related to chip area, a floating point circuit being complex in nature is costly to implement. They need not be included in the instruction set of a CPU. In such systems, floating-point operations were implemented by using software routines.

This implementation of floating point arithmetic is definitely slower than the hardware implementation. Now, the question is whether a processor can be constructed only for arithmetic operations. A processor, if devoted exclusively to arithmetic functions, can be used to implement a full range of arithmetic functions in the hardware at a relatively low cost. This can be done in a single Integrated Circuit. Thus, a special purpose arithmetic processor, for performing only the arithmetic operations, can be constructed. This processor physically may be separate, yet can be utilized by the CPU to execute complex arithmetic instructions. Please note in the absence of arithmetic processors, these instructions may be executed using the slower software routines by the CPU itself. Thus, this auxiliary processor enhances the speed of execution of programs having a lot of complex arithmetic computations.

An arithmetic processor also helps in reducing program complexity, as it provides a richer instruction set for a machine. Some of the instructions that can be assigned to arithmetic processors can be related to the addition, subtraction, multiplication, and division of floating point numbers, exponentiation, logarithms and other trigonometric functions.

How can this arithmetic processor be connected to the CPU?

Two mechanisms are used for connecting the arithmetic processor to the CPU.

If an arithmetic processor is treated as one of the Input / Output or peripheral units then it is termed as a peripheral processor. The CPU sends data and instructions to the peripheral processor, which performs the required operations on the data and communicates the results back to the CPU. A peripheral processor has several registers to communicate with the CPU. These registers may be addressed by the CPU as Input /Output register addresses. The CPU and peripheral processors are normally quite independent and communicate with each other by exchange of information using data transfer instructions. The data transfer instructions must be specific instructions in the CPU. This type of connection is called loosely coupled.

On the other hand if the arithmetic processor has a register and instruction set which can be considered an extension of the CPU registers and instruction set, then it is called a tightly coupled processor. Here the CPU reserves a special subset of code for arithmetic processor. In such a system the instructions meant for arithmetic processor are fetched by CPU and decoded jointly by CPU and the arithmetic processor, and finally executed by arithmetic processor. Thus, these processors can be considered a logical extension of the CPU. Such attached arithmetic processors are termed as co-processors.

The concept of co-processor existed in the 8086 machine till Intel 486 machines where co-processor was separate. However, Pentium at present does not have a separate co-processor. Similarly, peripheral processors are not found as arithmetic processors in general. However, many chips are used for specialized I/O architecture. These can be found in further readings.

Check Your Progress 2

1. Draw the logic circuit for a ALU unit.

2. What is an Arithmetic Processor?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

3.4 SUMMARY

In this unit, we have discussed in detail the hardware implementation of micro-operations. The unit starts with an implementation of bus, which is the backbone for any register transfer operation. This is followed by a discussion on arithmetic circuit and micro-operation thereon using full adder circuits. The logic micro-operation implementation has also been discussed. Thus, leading to a logical construction of a simple arithmetic – logic –shift unit. The unit revolves around the basic ALU with the help of the units that are constructed for the implementation of micro-operations.

In the later part of the unit, we discussed the arithmetic processors. Finally, we have presented a few chipsets that support the working of a processor for input/output functions from key board, printer etc.

3.5 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. False
2. False
3. True

Check Your Progress 2

1. The diagram is the same as that of Figure 9.
2. Arithmetic processor performs arithmetic computation. These are support processors to a computer.

UNIT 4 THE CONTROL UNIT

Structure **Page No.**

4.0 INTRODUCTION

By now we have discussed instruction sets and register organisation followed by a discussion on micro-operations and a simple arithmetic logic unit circuit. We have also discussed the floating point ALU and arithmetic processors, which are commonly used for floating point computations.

In this unit we are going to discuss the functions of a control unit, its structure followed by the hardwired type of control unit. We will discuss the micro-programmed control unit, which are quite popular in modern computers because of flexibility in designing. We will start the discussion with several definitions about the unit followed by Wilkes control unit. Finally, we will discuss the concepts involved in micro-instruction execution.

4.1 OBJECTIVES

After going through this unit you will be able to:

- define what is a control unit and its function;
- describe a simple control unit organization;
- define a hardwired control unit;
- define the micro-programmed control unit;
- define the term micro-instruction; and
- identify types and formats of micro-instruction.

4.2 THE CONTROL UNIT

The two basic components of a CPU are the control unit and the arithmetic and logic unit. The control unit of the CPU selects and interprets program instructions and then sees that they are executed. The basic responsibilities of the control unit are **to control**:

a) Data exchange of CPU with the memory or I/O modules.
b) Internal operations in the CPU such as:

   - moving data between registers (register transfer operations)

- making ALU to perform a particular operation on the data
- regulating other internal operations.

But how does a control unit control the above operations? What are the functional requirements of the control unit? What is its structure? Let us explore answers of these questions in the next sections.

Functional Requirements of a Control Unit

Let us first try to define the functions which a control unit must perform in order to get things to happen. But in order to define the functions of a control unit, one must know what resources and means it has at its disposal. A control unit must know about the:

(a) Basic components of the CPU

(b) Micro-operation this CPU performs.

The CPU of a computer consists of the following basic functional components:

- **The Arithmetic Logic Unit (ALU)**, which performs the basic arithmetic and logical operations.
- **Registers** which are used for information storage within the CPU.
- **Internal Data Paths:** These paths are useful for moving the data between two registers or between a register and ALU.
- **External Data Paths:** The roles of these data paths are normally to link the CPU registers with the memory or I/O interfaces. This role is normally fulfilled by the system bus.
- **The Control Unit:** This causes all the operations to happen in the CPU.

The micro-operations performed by the CPU can be classified as:

- Micro-operations for data transfer from register-register, register-memory, I/O-register etc.
- Micro- operations for performing arithmetic, logic and shift operations. These micro-operations involve use of registers for input and output.

The basic responsibility of the control unit lies in the fact that the control unit must be able to guide the various components of CPU to perform a specific sequence of micro-operations to achieve the execution of an instruction.

What are the functions, which a control unit performs to make an instruction execution feasible? The instruction execution is achieved by executing micro-operations in a specific sequence. For different instructions this sequence may be different. Thus the control unit must perform two basic functions:

- Cause the execution of a micro-operation.
- Enable the CPU to execute a proper sequence of micro-operations, which is determined by the instruction to be executed.

But how are these two tasks achieved? The control unit generates control signals, which in turn are responsible for achieving the above two tasks. But, how are these control signals generated? We will answer this question in later sections. First let us discuss a simple structure of control unit.

Structure of Control Unit

A control unit has a set of input values on the basis of which it produces an output control signal, which in turn performs micro-operations. These output signals control the execution of a program. A general model of control unit is shown in Figure 1.

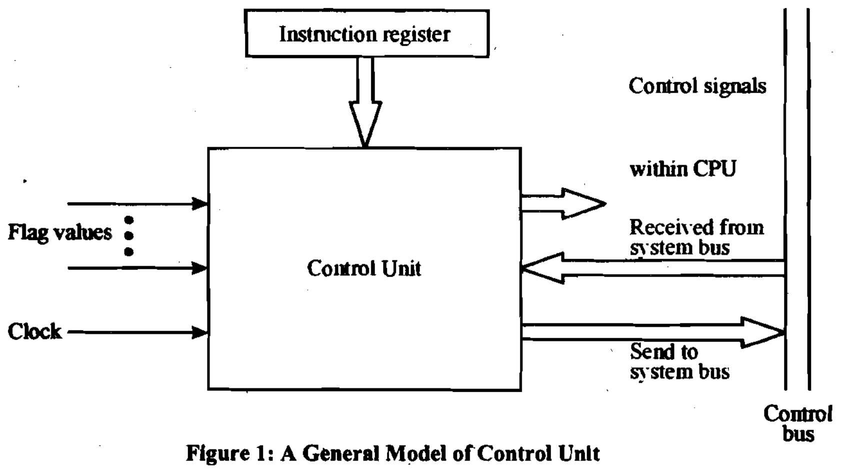


Figure 1: A General Model of Control Unit

In the model given above the control unit is a black box, which has certain inputs and outputs.

The inputs to the control unit are:

- **The Master Clock Signal:** This signal causes micro-operations to be performed in a square. In a single clock cycle either a single or a set of simultaneous micro-operations can be performed. The time taken in performing a single micro-operation is also termed as processor cycle time or the clock cycle time in some machines.

- **The Instruction Register:** It contains the operation code (opcode) and addressing mode bits of the instruction. It helps in determining the various cycles to be performed and hence determines the related micro-operations, which are needed to be performed.

- **Flags:** Flags are used by the control unit for determining the status of the CPU & the outcomes of a previous ALU operation. For example, a zero flag if set conveys to control unit that for instruction ISZ (skip the next instruction if zero flag is set) the next instruction is to be skipped. For such a case control unit cause increment of PC by program instruction length, thus skipping next instruction.

- **Control Signals from Control Bus:** Some of the control signals are provided to the control unit through the control bus. These signals are issued from outside the CPU. Some of these signals are interrupt signals and acknowledgement signals.

On the basis of the input signals the control unit activates certain output control signals, which in turn are responsible for the execution of an instruction. These output control signals are:

- **Control signals, which are required within the CPU:** These control signals cause two types of micro-operations, viz., for data transfer from one register to another; and for performing an arithmetic, logic and shift operation using ALU.

- **Control signals to control bus:** These control signals transfer data from or to CPU register to or from memory or I/O interface. These control signals are issued on the control bus to activate a data path on the data / address bus etc.

Now, let us discuss the requirements from such a unit. A prime requirement for control unit is that it must know how all the instructions will be executed. It should also know about the nature of the results and the indication of possible errors. All this is achieved with the help of flags, op-codes, clock and some control signals to itself.

A control unit contains a clock portion that provides clock-pulses. This clock signal is used for measuring the timing of the micro-operations. In general, the timing signals from control unit are kept sufficiently long to accommodate the proportional delays of signals within the CPU along various data paths. Since within the same instruction cycle different control signals are generated at different times for performing different micro-operations, therefore a counter can be utilised with the clock to keep the count. However, at the end of each instruction cycle the counter should be reset to the initial condition. Thus, the clock to the control unit must provide counted timing signals. Examples, of the functionality of control units along with timing diagrams are given in further readings.

How are these control signals applied to achieve the partieular operation? *The control signals are applied directly as the binary inputs to the logic gates of the logic circuits.* All these inputs are the control signals, which are applied to select a circuit (for example, select or enable input) or a path (for example, multiplexers) or any other operation in the logic circuits.

A program execution consists of a sequence of instruction cycles. Each instruction cycle is made up of a number of sub cycles. One such simple subdivision includes fetch, indirect, execute, and interrupt cycles, with only fetch and execute cycles always occurring. Each sub cycle involves one or more micro-operations.

Let us revisit the micro-operations described in Unit 2 to discuss how the events of any instruction cycle can be described as a sequence of such micro-operations.

Fetch Cycle

The beginning of each instruction cycle is the fetch cycle, and causes an instruction to be fetched from memory.

The fetch cycle consists of four micro-operations that are executed in three timing steps. The fetch cycle can be written as:

$$\begin{array}{lll} T_1: & MAR & \leftarrow PC \\ T_2: & MBR & \leftarrow [MAR] \\ & PC & \leftarrow PC + I \\ T_3: & IR & \leftarrow MBR \end{array}$$

where I is the instruction length. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all the units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (T_1, T_2, T_3) represents successive time units. What is done in these time units?

- In the first time unit the content of PC is moved to MAR.

- In the second time unit the contents of memory location specified by MAR is moved to MBR and the contents of the PC is incremented by I.

- In the third time unit the content of MBR is moved to IR.

The Indirect Cycle

Once an instruction is fetched, the next step is to fetch the operands. Considering the same example as of Unit 2, the instruction may have direct and indirect addressing modes. An indirect address is handled using indirect cycle. The following micro-operations are required in the indirect cycle:

T_1 : MAR ← IR (address)
T_2 : MBR ← [MAR]
T_3 : IR (address) ← MBR (address)

The MAR is loaded with the address field of IR register. Then the memory is read to fetch the address of operand, which is transferred to the address field of IR through MBR as data is received in MBR during the read operation.

Thus, the IR now is in the same state as of direct address, viz., as if indirect addressing had not been used. IR is now ready for the execute cycle.

The Execute Cycle

The fetch and indirect cycles involve a small, fixed sequence of micro-operations. Each of these cycles has fixed sequence of micro-operations that are common to all instructions.

This is not true of the execute cycle. For a machine with N different opcodes, there are N different sequences of micro-operations that can occur. Let us consider some hypothetical instructions:

An add instruction that adds the contents of memory location X to Register R1 with R1 storing the result:

ADD R1, X

The sequence of micro-operations may be:

T_1 : MAR ← IR (address)

T_2 : MBR ← [MAR]

T_3 : R1 ← R1 + MBR

At the beginning of the execute cycle IR contains the ADD instruction and its direct operand address (memory location X). At time T_1, the address portion of the IR is transferred to the MAR. At T_2 the referenced memory location is read into MBR Finally, at T_3 the contents of R1 and MBR are added by the ALU.

Let us discuss one more instruction:

ISZ X, it increments the content of memory location X by 1. If the result is 0, the next instruction in the sequence is skipped. A possible sequence of micro-operations for this instruction may be:

T_1 : MAR ← IR (address)

T_2 : MBR ← [MAR]

T_3 : MBR ← MBR+ 1

T_4 : [MAR] ← MBR

If (MBR = 0) then (PC $\leftarrow$ PC+ I)

Please note that for this machine we have assumed that MBR can be incremented by ALU directly.

The PC is incremented if MBR contains 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory. Such instructions are useful in implementing looping.

The Interrupt Cycle

On completion of the execute cycle the current instruction execution gets completed. At this point a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle is performed. This cycle does not execute an interrupt but causes start of execution of Interrupt Service Program (ISR). Please note that ISR is executed as just another program instruction cycle. The nature of this cycle varies greatly from one machine to another. A typical sequence of micro-operations of the interrupt cycle are:

T_1 : MBR $\leftarrow$ PC

T_2 : MAR $\leftarrow$ Save-Address

PC $\leftarrow$ ISR- Address

T_3 : [MAR] $\leftarrow$ MBR

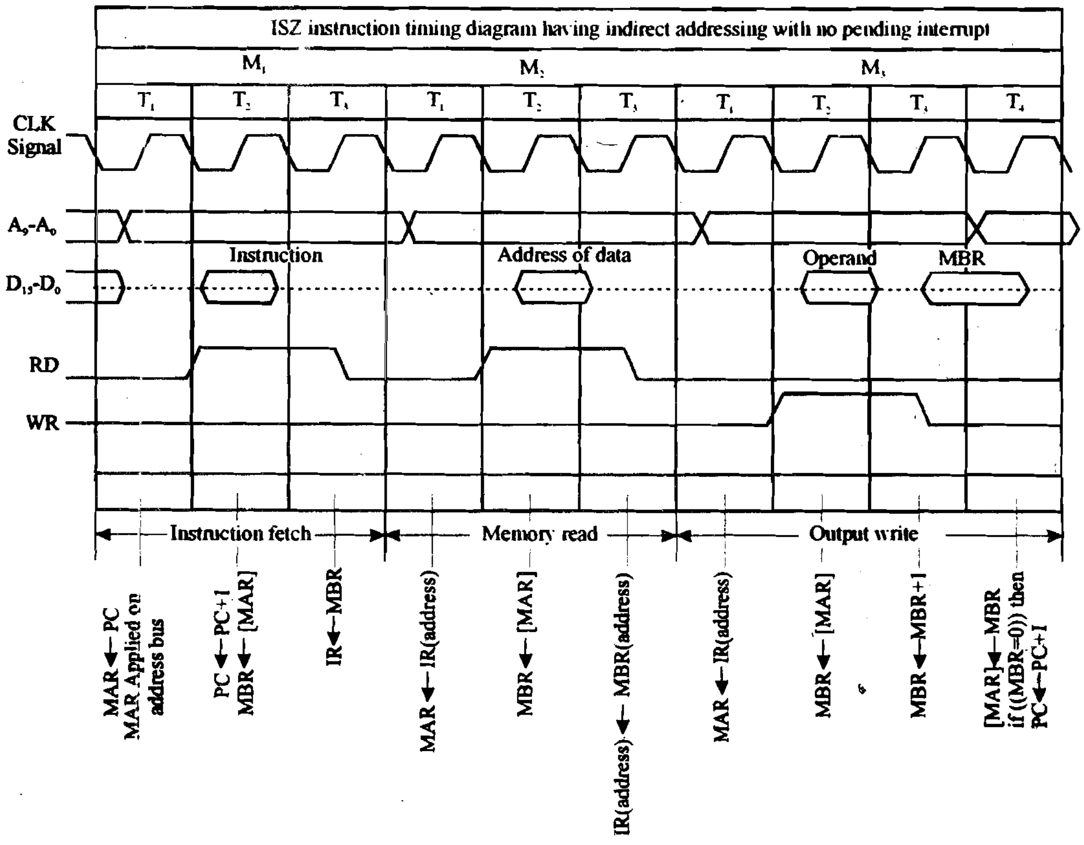
At time T_1, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. At time T_2 the MAR is loaded with the address at which the contents of the PC are to be saved, and PC is loaded with the address of the start of the interrupt-servicing routine. At time T_3 MBR, which contains the old value of the PC, is stored in the memory. The processor is now ready to begin the next instruction cycle.

The Instruction Cycle

The instruction cycle for this given machine consists of four cycles. Assume a 2-bit instruction cycle code (ICC). The ICC can represent the state of the processor in terms of cycle. For example, we can use:

00 :	Fetch
01 :	Indirect
10 :	Execute
11 :	Interrupt

At the end of each of the four cycles, the ICC is set appropriately. Please note that an indirect cycle is always followed by the execute cycle and the interrupt cycle is always followed by the fetch cycle. For both the execute and fetch cycles, the next cycle depends on the state of the system. Let us show an instruction execution using timing diagram and instruction cycles:



Assumptions: **10 bit address bus, 16 bit data bus, size of instruction 16bits - with 10 bit address, 6 bit opcode**

Figure 2: Timing Diagram for ISZ instruction

Please note that the address line determine the location of memory. Read/ write signal controls whether the data is being input or output. For example, at time T_2 in M_2 the read control signal becomes active, $A_9 - A_0$ input contains MAR that value is kept enabled on address bits and the data lines are enabled to accept data from RAM, thus enabling a typical RAM data output on the data bus.

For reading no data input is applied by CPU but it is put on data bus by memory after the read control signal to memory is activated. Write operation is activated along with data bus carrying the output value.

This diagram is used for illustration of timing and control. However, more information on these topics can be obtained from further readings.

4.3 THE HARDWIRED CONTROL

With the last section we have discussed the control unit in terms of its inputs, output and functions. A variety of techniques have been used to organize a control unit. Most of them fall into two major categories:

1. Hardwired control organization
2. Microprogrammed control organization.

In the hardwired organization, the control unit is designed as a combinational circuit. That is, the control unit is implemented by gates, flip-flops, decoder and other digital circuits. Hardwired control units can be optimised for fast operations.

The block diagram of control unit is shown in Figure 3. The major inputs to the circuit are instruction register, the clock, and the flags. The control unit uses the opcode of instruction stored in the IR register to perform different actions for different instructions. The control unit logic has a unique logic input for each opcode This simplifies the control logic. This control line selection can be performed by a decoder. A decoder will have n binary inputs and 2^n binary outputs. Each of these 2^n different input patterns will activate a single unique output line.

The clock portion of the control unit issues a repetitive sequence of pulses for the SS duration of micro-operation(s). These timing signals control the sequence of execution of instruction and determine what control signal needs to applied at what time for instruction execution.

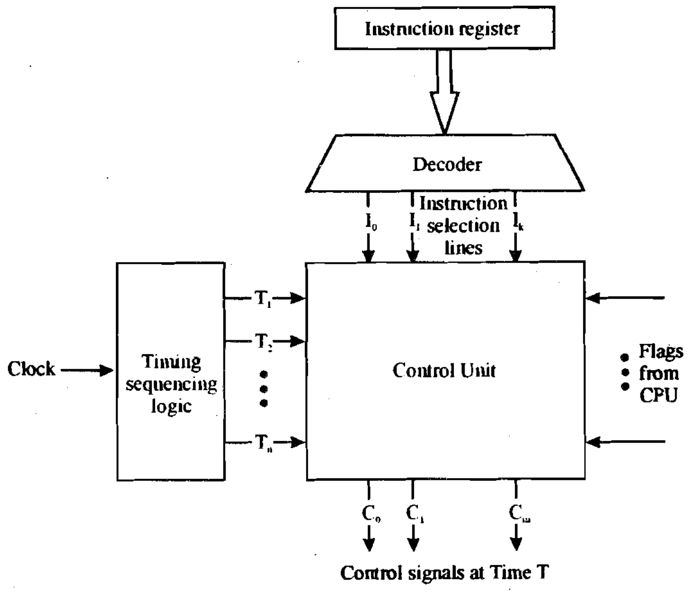


Figure 3: Block Diagram of Control Unit Operation

Check Your Progress 1

1. What are the inputs to control unit?

....

....

....

2. How does a control unit control the instruction cycle?

....

....

....

3. What is a hardwired control unit?

....

....

....

4.4 WILKES CONTROL

Prof. M. V. Wilkes of the Cambridge University Mathematical Laboratory coined the term microprogramming in 1951. He provided a systematic alternative procedure for designing the control unit of a digital computer. During instruction executing a machine instruction, a sequence of transformations and transfer of information from one register in the processor to another take place. These were also called the micro operations. **Because of the analogy between the execution of individual steps in a machine instruction to the execution of the individual instruction in a program, Wilkes introduced the concept of microprogramming.** The Wilkes control unit replaces the sequential and combinational circuits of hardwired control unit by a simple control unit in conjunction with a storage unit that stores the sequence of steps of instruction that is a micro-program.

In Wilkes microinstruction has two major components:

a) Control field which indicates the control lines which are to be activated and
b) Address field, which provides the address of the next microinstruction to be executed.

The figure 4 below is an example of Wilkes control unit design.

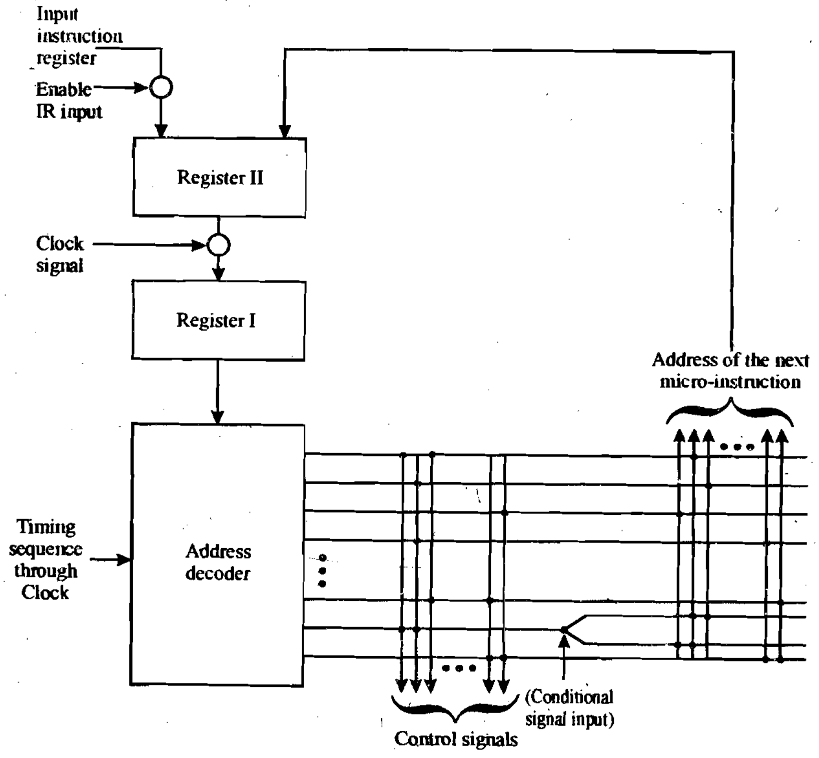

Input instruction register
Enable IR input
Register II
Clock signal
Register I
Address of the next micro-instruction
Timing sequence through Clock
Address decoder
(Conditional signal input)
Control signals

Figure 4: Wilkes Control Unit

The control memory in Wilkes control is organized, as a PLA's like matrix made of diodes. This is partial matrix and consists of two components, the control signals and the address of the next micro-instruction. The register I contains the address of the next micro-instruction that is one step of instruction execution, for example T_1 in M_1 or T_2 in M_2 etc. as in Figure 2. On decoding the control signals are generated that cause execution of micro-operation(s) of that step. In addition, the control unit indicates the address of the next micro-operation which gets loaded through register II to register I. Register I can also be loaded by register II and "enable IR input" control signal. This will pass the address of first micro-instruction of execute cycle. During a machine cycle one row of the matrix is activated. The first part of the row generates the control signals that control the operations of the processor. The second part generates the address of the row to be selected in the next machine cycle.

At the beginning of the cycle, the address of the row to be selected is contained in register I. This address is the input to the decoder, which is activated by a clock pulse.

This activates the row of the control matrix. The two-register arrangement is needed, as the decoder is a combinational circuit; with only one register, the output would become the input during a cycle. This may be an unstable condition due to repetitive loop.

4.5 THE MICRO-PROGRAMMED CONTROL

An alternative to a hardwired control unit is a micro-programmed control unit, in which the logic of the control unit is specified by a micro-program. A micro-program is also called firmware (midway between the hardware and the software). It consists of:

(a) One or more micro-operations to be executed; and
(b) The information about the micro-instruction to be executed next.

The general configuration of a micro-programmed control unit is demonstrated in Figure 5 below:

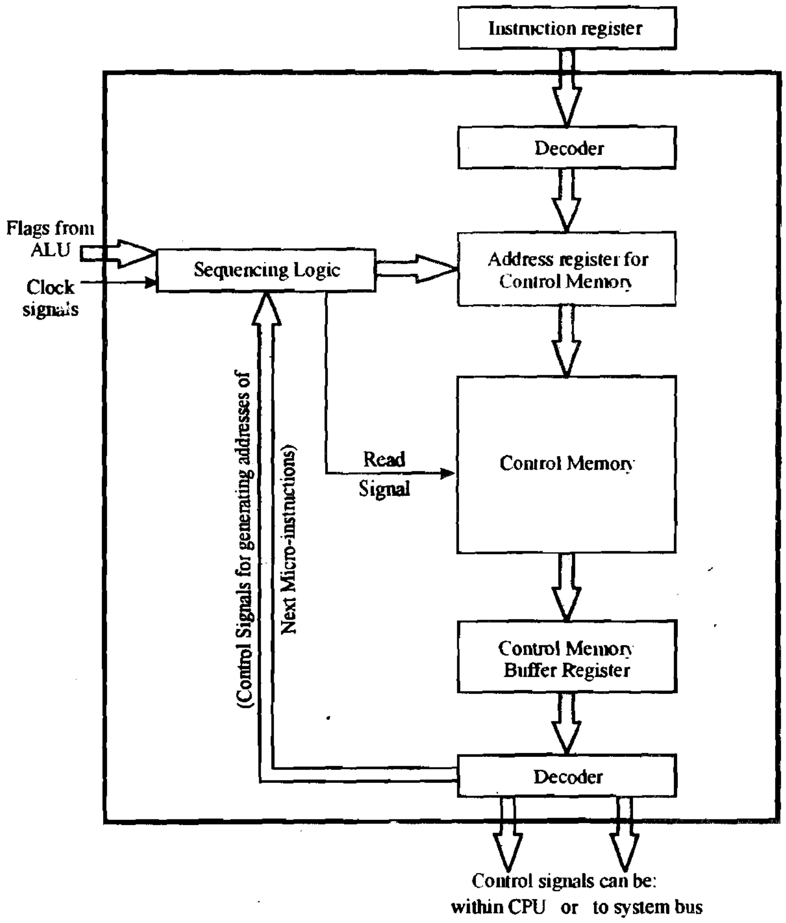


Figure 5: Operation of Micro-Programmed Control Unit

The micro-instructions are stored in the control memory. The address register for the control memory contains the address of the next instruction that is to be read. The control memory Buffer Register receives the micro-instruction that has been read. A micro-instruction execution primarily involves the generation of desired control signals and signals used to determine the next micro-instruction to be executed. The sequencing logic section loads the control memory address register. It also issues a read command to control memory. The following functions are performed by the micro-programmed control unit:

1. The sequence logic unit specifies the address of the control memory word that is to be read, in the Address Register of the Control Memory. It also issues the READ signal.
2. The desired control memory word is read into control memory Buffer Register.
3. The content of the control memory buffer register is decoded to create control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit finds the address of the next control word on the basis of the next-address information from the decoder and the ALU flags.

As we have discussed earlier, the execute cycle steps of micro-operations are different for all instructions in addition the addressing mode may be different. All such information generally is dependent on the opcode of the instruction Register (IR). Thus, IR input to Address Register for Control Memory is desirable. Thus, there exist a decoder from IR to Address Register for control memory. (Refer Figure 5). This decoder translates the opcode of the IR into a control memory address.

Check Your Progress 2

1. What is firmware? How is it different from software?

....

....

....

2. **State True or False**

	T	F
(a) A micro-instruction can initiate only one micro-operation at a time.	☐	
(b) A control word is equal to a memory word.	☐	
(c) Micro-programmed control is faster than hardwired control.	☐	
(d) Wilkes control does not provide a branching micro-instruction.	☐	

3. What will be the control signals and address of the next micro-instruction in the Wilkes control example of Figure 4, if the entry address for a machine instruction selects the last but one (branching control line) and the conditional bit value for branch is true?

....

....

....

4.6 THE MICRO-INSTRUCTIONS

A micro-instruction, as defined earlier, is an instruction of a micro-program. It specifies one or more micro-operations, which can be executed simultaneously. On executing a micro-instruction a set of control signals are generated which in turn cause the desired micro-operation to happen.

4.6.1 Types of Micro-instructions

In general the micro-instruction can be categorised into two general types. These are branching and non-branching. After execution of a non-branching micro-instruction the next micro-instruction is the one following the current micro-instruction.

However, the sequences of micro-instructions are relatively small and last only for 3 or 4 micro-instructions.

A conditional branching micro-instruction tests conditional variable or a flag generated by an ALU operation. Normally, the branch address is contained in the micro-instruction itself.

4.6.2 Control Memory Organization

The next important question about the micro-instruction is: how are they organized in the control memory? One of the simplest ways to organize control memory is to arrange micro-instructions for various sub cycles of the machine instruction in the memory. The Figure 6 shows such an organisation.

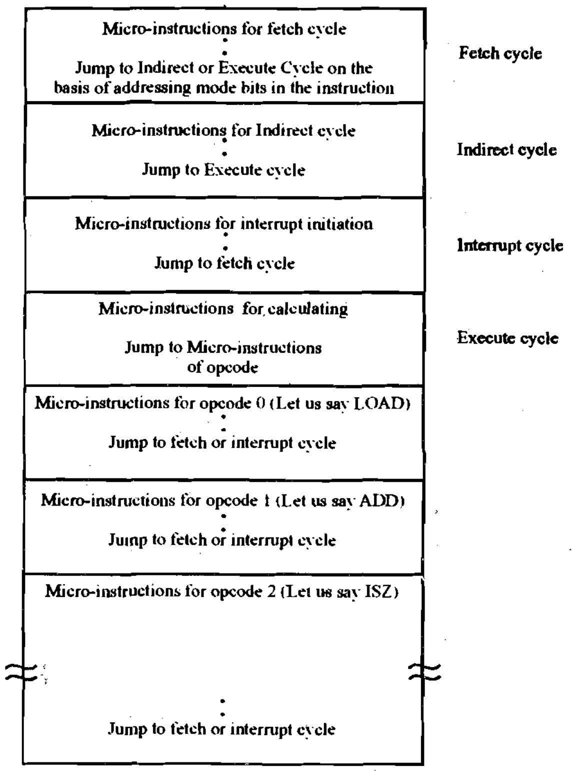

Micro-instructions for fetch cycle
Jump to Indirect or Execute Cycle on the basis of addressing mode bits in the instruction
Fetch cycle
Micro-instructions for Indirect cycle
Jump to Execute cycle
Indirect cycle
Micro-instructions for interrupt initiation
Jump to fetch cycle
Interrupt cycle
Micro-instructions for calculating
Jump to Micro-instructions of opcode
Execute cycle
Micro-instructions for opcode 0 (Let us say LOAD)
Jump to fetch or interrupt cycle
Micro-instructions for opcode 1 (Let us say ADD)
Jump to fetch or interrupt cycle
Micro-instructions for opcode 2 (Let us say ISZ)
Jump to fetch or interrupt cycle

Figure 6: Control Memory Organisation

Let us give an example of control memory organization. Let us take a machine instruction: Branch on zero. This instruction causes a branch to a specified main memory address in case the result of the last ALU operation is zero, that is, the zero flag is set. The pseudocode of the micro-program for this instruction can be;

```
        Test "zero flag" If SET branch to label ZERO
        Unconditional branch to label NON-ZERO
```

ZERO: (Microcode which causes replacement of program counter with the address provided in the instruction)

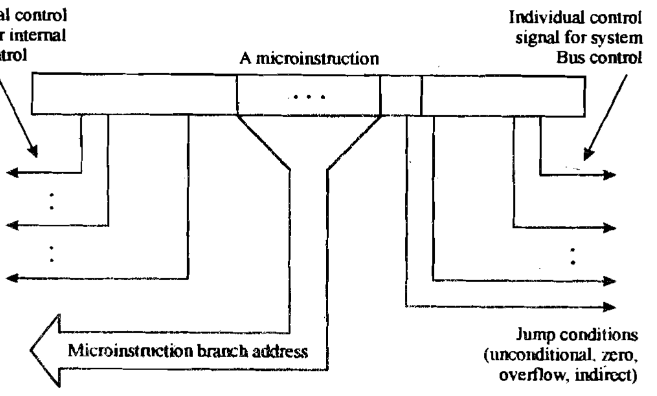
Branch to interrupt or fetch cycle.

NON -ZERO: (Microcode which may set flags if desired indicating the branch has not taken place).

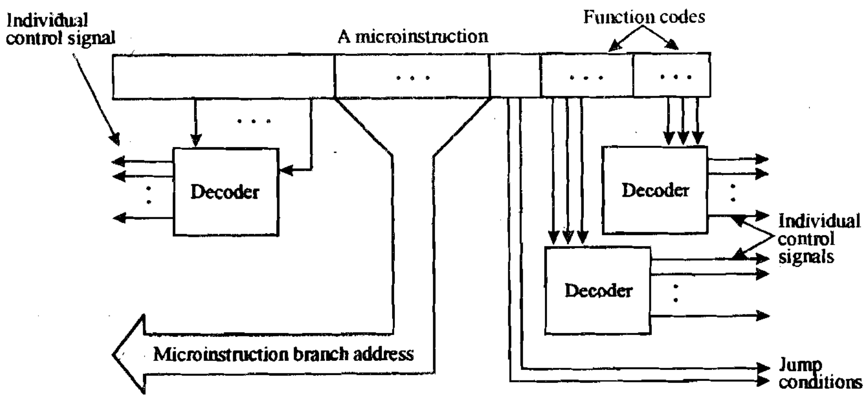
Branch to interrupt or fetch cycle. (For Next- Instruction Cycle)

4.6.3 Micro-instruction Formats

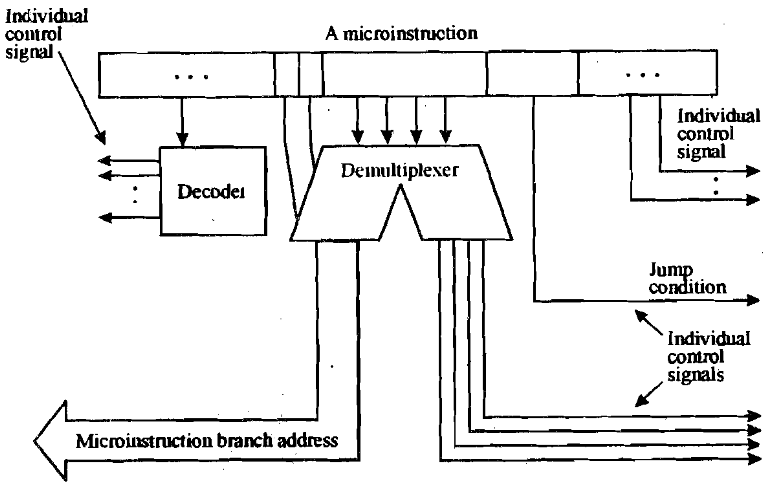
Now let us focus on the format of a micro-instruction. The two widely used formats used for micro-instructi ıs are horizontal and vertical. In the horizontal micro-instruction each bit of t! micro-instruction represents a control signal, which directly controls a single bus line or sometimes a gate in the machine. However, the length of such a micro-instruction may be hundreds of bits. A typical horizontal micro-instruction with its related fields is shown in Figure 7(a).


Individual control signal for internal CPU control
A microinstruction
Individual control signal for system Bus control
Microinstruction branch address
Jump conditions (unconditional, zero, overflow, indirect)

(a) Horizontal Micro-instruction


Individual control signal
A microinstruction
Function codes
Decoder
Decoder
Decoder
Individual control signals
Microinstruction branch address
Jump conditions

(b) Vertical Micro-instructions


Individual control signal
A microinstruction
Decoder
Demultiplexer
Individual control signal
Jump condition
Individual control signals
Microinstruction branch address

(c) A Realistic Micro-instructions
Figure 7: Micro- instruction Formats

In a vertical micro-instruction many similar control signals can be encoded into a few micro-instruction bits. For example, for 16 ALU operations, which may require 16 individual control bits in horizontal micro-instruction, only 4 encoded bits are needed in vertical micro-instruction. Similarly, in a vertical micro-instruction only 3 bits are needed to select one of the eight registers. However, these encoded bits need to be passed from the respective decoders to get the individual control signals. This is shown in figure 7(b).

In general, a horizontal control unit is faster, yet requires wider instruction words, whereas vertical control units, although; require a decoder, are shorter in length. Most of the systems use neither purely horizontal nor purely vertical micro-instructions figure 7(c).

4.7 THE EXECUTION OF MICRO-PROGRAM

The micro-instruction cycle can consist of two basic cycles: the fetch and the execute. Here, in the fetch cycle the address of the micro-instruction is generated and this micro-instruction is put in a register used for the address of a micro-instruction for execution. The execution of a micro-instruction simply means generation of control signals. These control signals may drive the CPU (internal control signals) or the system bus. The format of micro-instruction and its contents determine the complexity of a logic module, which executes a micro-instruction.

One of the key features incorporated in a micro-instruction is the encoding of micro-instructions. What is encoding of micro-instruction? For answering this question let us recall the Wilkes control unit. In Wilkes control unit, each bit of information either generates a control signal or form a bit of next instruction address. Now, let us assume that a machine needs N total number of control signals. If we follow the Wilkes scheme we require N bits, one for each control signal in the control unit.

Since we are dealing with binary control signals, therefore, a 'N' bit micro-instruction can represent 2^N combinations of control signals.

The question is do we need all these 2^N combinations?

No, some of these 2^N combinations are not used because:

1. Two sources may be connected by respective control signals to a single destination; however, only one of these sources can be used at a time. Thus, the combinations where both these control signals are active for the same destination are redundant.
2. A register cannot act as a source and a destination at the same time. Thus, such a combination of control signals is redundant.
3. We can provide only one pattern of control signals at a time to ALU, making some of the combinations redundant.
4. We can provide only one pattern of control signals at a time to the external control bus also.

Therefore, we do not need 2^N combinations. Suppose, we only need 2^K (which is less than 2^N) combinations, then we need only K encoded bits instead of N control signals. The K bit micro-instruction is an extreme encoded micro-instruction. Let us touch upon the characteristics of the extreme encoded and unencoded micro-instructions:

Unencoded micro-instructions

- One bit is needed for each control signal; therefore, the number of bits required in a micro-instruction is high.
- It presents a detailed hardware view, as control signal need can be determined.

- Since each of the control signals can be controlled individually, therefore these micro-instructions are difficult to program. However, concurrency can be exploited easily.
- Almost no control logic is needed to decode the instruction as there is one to one mapping of control signals to a bit of micro-instruction. Thus, execution of micro-instruction and hence the micro-program is faster.
- The unencoded micro-instruction aims at optimising the performance of a machine.

Highly Encoded micro-instructions

- The encoded bits needed in micro-instructions are small.
- It provided an aggregated view that is a higher view of the CPU as only an encoded sequence can be used for micro-programming.
- The encoding helps in reduction in programming burden; however, the concurrency may not be exploited to the fullest.
- Complex control logic is needed, as decoding is a must. Thus, the execution of a micro-instruction can have propagation delay through gates. Therefore, the execution of micro-program takes a longer time than that of an unencoded micro-instruction.
- The highly encoded micro-instructions are aimed at optimizing programming effort.

In most of the cases, the design is kept between the two extremes. The LSI 11 (highly encoded) and IBM 3033 (unencoded) control units are close examples of these two approaches.

Execution/decoding of slightly encoded micro-instructions

In general, the micro-programmed control unit designs are neither completely unencoded nor highly encoded. They are slightly coded. This reduces the width of control memory and micro-programming efforts. The basic technique for encoding is shown in Figure 8. The micro-instruction is organised as a set of fields. Each field contains a code, which, upon decoding, activates one or more control signals. The execution of a micro-instruction means that every field is decoded and generates control signals. Thus, with N fields, N simultaneous actions can be specified. Each action results in the activation of one or more control signals. Generally each control signal is activated by no more than one field. The design of an encoded micro-instruction format can be stated in simple terms:

- Organize the format into independent fields. That is, each field depicts a set of actions such that actions from different fields can occur simultaneously.
- Define each field such that the alternative actions that can be specified by the field are mutually exclusive. That is, only one of the actions specified for a given field could occur at a time.

Another aspect of encoding is whether it is direct or indirect (Figure 8). With indirect encoding, one field is used to determine the interpretation of another field.

Another aspect of micro-instruction execution is the micro-instruction sequencing that involves address calculation of the next micro-instruction. In general, the next micro- instruction can be (refer Figure 6):

- Next micro-instruction in sequence
- Calculated on the basis of opcode
- Branch address (conditional or unconditional).

A detailed discussion on these topics is beyond this unit. You must refer to further readings for more detailed information on Micro-programmed Control Unit Design.

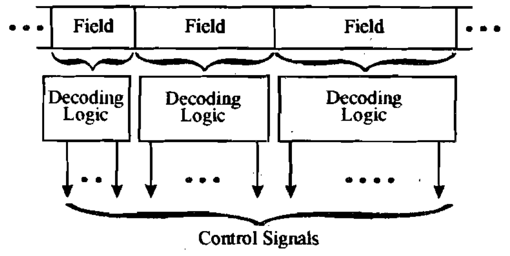


Figure (a): Direct Encoding

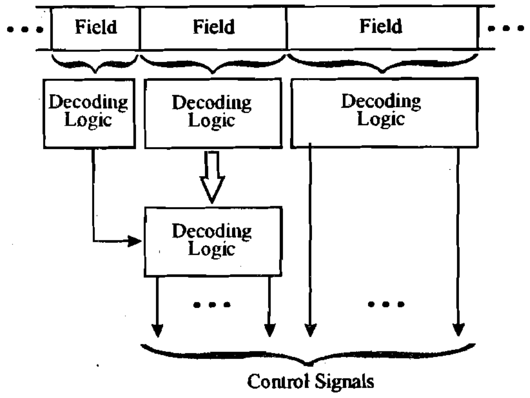


Figure (b): Indirect Encoding

Figure 8: Micro-instruction Encoding

Check Your Progress 3

1. **State True or False** | T | F |

   a) A branch micro-instruction can have only an unconditional jump. ☐

   b) Control store stores opcode-based micro-programs. ☐

   c) A true horizontal micro-instruction requires one bit for every control signal. ☐

   d) A decoder is needed to find a branch address in the vertical micro-instruction. ☐

   e) One of the responsibilities of sequencing logic (Refer Figure 5) is to cause reading of micro-instruction addressed by a micro-program counter into the micro-instruction buffer. ☐

   f) Status bits supplied from ALU to sequencing logic have no role to play with the sequencing of micro-instruction. ☐

2. What art the possibilities for the next instruction address?

   ....

   ....

   ....

   ....

   ....

3. How many address fields are there in Wilkes Control Unit?

....

....

....

4. Compare and contrast unencoded and highly encoded micro-instructions.

....

....

....

4.8 SUMMARY

In this unit we have discussed the organization of control units. Hardwired, Wilkes and micro-programmed control units are also discussed. The key to such control units are micro-instruction, which can be briefly (that is types and formats) described in this unit. The function of a micro-programmed unit, that is, micro-programmed execution, has also been discussed. The control unit is the key for the optimised performance of a computer. The information given in this unit can be further appended by going through further readings.

4.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. IR, Timing Signal, Flags Register
2. The control unit issues control signals that cause execution of micro-operations in a pre-determined sequence. This, enables execution sequence of an instruction.
3. A logic circuit based implementation of control unit.

Check Your Progress 2

1. Firmware is basically micro-programs, which are used in a micro-programmed control unit. Firmwares are more difficult to write than software.

2. (a) False (b) False (C) False (d) False

3. In sequence from left to right as per figure.
   110......00 (control signals ...... indicate more values)
   110......00 (address of next, micro-instruction is found after assuming that bottom line after condition code represent true in the Figure 4)

Check Your Progress 3

1. (a) False (b) False (c) True (d) False (e) True (f) False.

2. The address of the next micro-instruction can be one of the following:

   - the address of the next micro-instruction in sequence.
   - determined by opcode using mapping or any other method.
   - branch address supplied on the internal address bus.

3. Wilkes control typically has one address field. However, for a conditional branching micro-instruction, it contains two addresses. The Wilkes control, in fact, is a hardware representation of a micro-programmed control unit.

4.

Unencoded Micro instructions	Highly encoded
• Large number of bits • Difficult to program • No decoding logic • Optimizes machine performances • Detailed hardware view	Relatively less bits Easy to program Need decoding logic Optimizes programming effort Aggregated view

UNIT 5 REDUCED INSTRUCTION SET COMPUTER ARCHITECTURE

5.0 INTRODUCTION

In the previous units, we have discussed the instruction set, register organization and pipelining, and control unit organization. The trend of those years was to have a large instruction set, a large number of addressing modes and about 16 –32 registers. However, their existed a pool of thought which was in favour of having simplicity in instruction set. This logic was mainly based on the type of the programs, which were being written for various machines. This led to the development of a new type of computers called Reduced Instruction Set Computer (RISC). In this unit, we will discuss about the RISC machines. Our emphasis will be on discussing the basic principles of RISC and its pipeline. We will also discuss the arithmetic and logic units here.

5.1 OBJECTIVES

After going through this unit you should be able to:

- define why complexity of instruction increased?;
- describe the reasons for developing RISC;
- define the basic design principles of RISC;
- describe the importance of having large register file;
- discuss some of the common comments about RISC;
- describe RISC pipelining; and
- define the optimisation in RISC pipelining.

5.2 INTRODUCTION TO RISC

The aim of computer architects is to design computers which are cheaper and more powerful than their predecessors. A cheaper computer has:

- Low hardware manufacturing cost.
- Low Cost for programming scalable/ portable architecture that require low costs for debugging the initial hardware and subsequent programs.

If we review the history of computer families, we find that the most common architectural change is the trend towards even more complex machines.

5.2.1 Importance of RISC Processors

Reduced Instruction Set Computers recognize a relatively limited number of instructions. One advantage of a reduced instruction set is that RISC can execute the instructions very fast because these are so simple. Another advantage is that RISC chips require fewer gates and hence transistors, which makes them cheaper to design and produce.

An instruction of RISC machine can be executed in one cycle, as there exists an instruction pipeline. This may enhance the speed of instruction execution. In addition, the control unit of the RISC processor is simpler and smaller, so much so that it acquires only 6% space for a processor in comparison to Complex Instruction Set Computers (CISC) in which the control unit occupies about 50% of space. This saved space leaves a lot of room for developing a number of registers.

This further enhances the processing capabilities of the RISC processor. It also necessitates that the memory to register "LOAD" and "STORE" are independent instructions.

Various RISC Processors

RISC has fewer design bugs, its simple instructions reduce design time. Thus, because of all the above important reasons RISC processors have become very popular. Some of the RISC processors are:

SPARC Processors

Sun 4/100 series, Sun 4/310 SPARCserver 310, Sun 4/330 SPARCserver 330, Sun 4/350 SPARCserver 350, Sun 4/360 SPARCserver 360, Sun 4/370 SPARCserver 370, Sun 4/20, SPARCstation SLC, Sun 4/40 SPARCstation IPC, Sun 4/75, SPARCstation 2.

verPC Processors

MPC603, MPC740, MPC750, MPC755, MPC7400/7410, MPC745x, MPC7450, MPC8240, MPC8245.

Titanium – IA64 Processor

5.2.2 Reasons for Increased Complexity

Let us see what the reasons for increased complexity are, and what exactly we mean by this.

Speed of Memory Versus Speed of CPU

In the past, there existed a large gap between the speed of a processor and memory. Thus, a subroutine execution for an instruction, for example floating point addition, may have to follow a lengthy instruction sequence. The question is; if we make it a machine instruction then only one instruction fetch will be required and rest will be done with control unit sequence. Thus, a "higher level" instruction can be added to machines in an attempt to improve performance.

However, this assumption is not very valid in the present era where the Main memory is supported with Cache technology. Cache memories have reduced the difference between the CPU and the memory speed and, therefore, an instruction execution through a subroutine step may not be that difficult.

Let us explain it with the help of an example:

Suppose the floating point operation ADD A, B requires the following steps (assuming the machine do not have floating point registers) and the registers being used for exponent are E1, E2, and EO (output); for mantissa M1, M2 and MO (output):

```
•  Load the exponent of A in E1
•  Load the mantissa of A in M1
•  Load the exponent of B in E2
•  Load the mantissa of B in M2
•  Compare E1 and E2
      -  If E1 = E2 then MO ← M1 + M2 and EO ← E1
         Normalise MO and adjust EO
            •  Result will be contained in MO, E1
   else if E1< E2 then find the difference = E2 – E1
            •  Shift Right M1, by difference
            •  MO ← M1 + M2 and EO ← E2
            •  Normalise MO and adjust EO
            •  Result is contained in MO, EO
   else E2 < E1, if so find the difference = E1 – E2
            •  Shift Right M2 by difference above
            •  MO ← M1 + M2 and EO ← E1
            •  Normalise MO and adjust E1 into EO
            •  Result is contained in MO, EO
   Store the above results in A
   Checks overflow underflow if any.
```

If all these steps are coded as one machine instruction, then this simple instruction will require many instruction execution cycles. If this instruction is made as part of the machine instruction set as: ADDF A,B (Add floating point numbers A & B and store results in A) then it will just be a single machine instruction. All the above steps required will then be coded with the help of micro-operations in the form of Control Unit Micro-Program. Thus, just one instruction cycle (although a long one) may be needed. This cycle will require just one instruction fetch. Whereas in the program memory instructions will be fetched.

However, faster cache memory for Instruction and data stored in registers can create an almost similar instruction execution environment. Pipelining can further enhance such speed. Thus, creating an instruction as above may not result in faster execution.

Microcode and VLSI Technology

It is considered that the control unit of a computer be constructed using two ways; create micro-program that execute micro-instructions or build circuits for each instruction execution. Micro-programmed control allows the implementation of complex architectures more cost effective than hardwired control as the cost to expand an instruction set is very small, only a few more micro-instructions for the control store. Thus, it may be reasoned that moving subroutines like string editing, integer to floating point number conversion and mathematical evaluations such as polynomial evaluation to control unit micro-program is more cost effective.

Code Density and Smaller Faster Programs

The memory was very expensive in the older computer. Thus there was a need of less memory utilization, that is, it was cost effective to have smaller compact programs. Thus, it was opined that the instruction set should be more complex, so that programs are smaller. However, increased complexity of instruction sets had resulted in instruction sets and addressing modes requiring more bits to represent them. It is stated that the code compaction is important, but the cost of 10 percent more memory is often far less than the cost of reducing code by 10 percent out of the CPU architecture innovations.

The smaller programs are advantageous because they require smaller RAM space. However, today memory is very inexpensive, this potential advantage today is not so compelling. More important, small programs should improve performance. How? Fewer instructions mean fewer instruction bytes to be fetched.

However, the problem with this reasoning is that it is not certain that a CISC program will be smaller than the corresponding RISC program. In many cases CISC program expressed in symbolic machine language may be smaller but the number of bits of machine code program may not be noticeably smaller. This may result from the reason that in RISC we use register addressing and less instruction, which require fewer bits in general. In addition, the compilers on CISCs often favour simpler instructions, so that the conciseness of complex instruction seldom comes into play.

Let us explain this with the help of the following example:

Assumptions:

- The Complex Instruction is: Add C, A, B having 16 bit addresses and 8 bit data operands
- All the operands are direct memory reference operands
- The machine has 16 registers. So the size of a register address is = 2^4 = 16 = 4 bits.
- The machine uses an 8-bit opcode.

8	16	16	16
Add	C	A	B

8	4	16	
Load	rA	A	
Load	rB	B	
Add	rC	rA	rB
Store	rC	C	

Memory-to-Memory
Instruction size (I) = 56 bits
Data Size (D) = 24 bits
Total Memory Load (M) = 80 bits

Register-to-Register
I = 104 bits
D = 24bits
M = 128 bits

(a) Add A & B to store result in C

8	16	16	16
Add	C	A	B
Add	A	C	D
Sub	D	D	B

8	4	16	
Load	rA	A	
Load	rB	B	
Add	rC	rB	rA
Load	rD	D	
Add	rA	rC	Rd
Sub	rD	rD	rB
Store	rD	D	

Memory-to-Memory
Instruction size (I) = 168 bits
Data Size (D) = 72 bits
Total Memory Load (M) = 240 bits

Register-to-Register
I = 172 bits
D = 32bits
M = 204 bits

(b) Execution of the Instruction Sequence: C = A + B, A = C + D, D = D - B

Figure 1: Program size for different Instruction Set Approaches

So, the expectation that a CISC will produce smaller programs may not be realised.

Support for High-Level Language

With the increasing use of more and higher level languages, manufacturers had provided more powerful instructions to support them. It was argued that a stronger instruction set would reduce the software crisis and would simplify the compilers. Another important reason for such a movement was the desire to improve performance.

However, even though the instructions that were closer to the high level languages were implemented in Complex Instruction Set Computers (CISCs), still it was hard to exploit these instructions since the compilers were needed to find those conditions that exactly fit those constructs. In addition, the task of optimising the generated code to minimise code size, reduce instruction execution count, and enhance pipelining is much more difficult with such a complex instruction set.

Another motivation for increasingly complex instruction sets was that the complex HLL operation would execute more quickly as a single machine instruction rather than as a series of more primitive instructions. However, because of the bias of programmers towards the use of simpler instructions, it may turn out otherwise. CISC makes the more complex control unit with larger microprogram control store to accommodate a richer instruction set. This increases the execution time for simpler instructions.

Thus, it is far from clear that the trend to complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

5.2.3 High Level Language Program Characteristics

Thus, it is clear that new architectures should support high-level language programming. A high-level language system can be implemented mostly by hardware or mostly by software, provided the system hides any lower level details from the programmer. Thus, a cost-effective system can be built by deciding what pieces of the system should be in hardware and what pieces in software.

To ascertain the above, it may be a good idea to find program characteristics on general computers. Some of the basic findings about the program characteristics are:

Variables	Operations	Procedure Calls
Integral Constants 15-25% Scalar Variables 50-60% Array/ structure 20-30%	Simple assignment 35-45% Looping 2-6% Procedure call 10-15% IF 35-45% GOTO FEW Others 1-5%	Most time consuming operation. FACTS: Most of the procedures are called with fewer than 6 arguments. Most of these have fewer than 6 local variables

Figure 2: Typical Program Characteristics

Observations

- Integer constants appeared almost as frequently as arrays or structures.

- Most of the scalars were found to be local variables whereas most of the arrays or structures were global variables.
- Most of the dynamically called procedures pass lower than six arguments.
- The numbers of scalar variables are less than six.
- A good machine design should attempt to optimize the performance of most time consuming features of high-level programs.
- Performance can be improved by more register references rather than having more memory references.
- There should be an optimized instructional pipeline such that any change in flow of execution is taken care of.

The Origin of RISC

In the 1980s, a new philosophy evolved having optimizing compilers that could be used to compile "normal" programming languages down to instructions that were as simple as equivalent micro-operations in a large virtual address space. This made the instruction cycle time as fast as the technology would permit. These machines should have simple instructions such that it can harness the potential of simple instruction execution in one cycle – thus, having reduced instruction sets – hence the reduced instruction set computers (RISCs).

Check Your Progress 1

1. List the reasons of increased complexity.

   ....

   ....

   ....

2. State True or False

	T	F
a) The instruction cycle time for RISC is equivalent to CISC.	☐	
b) CISC yields smaller programs than RISC, which improves its performance; therefore, it is very superior to RISC.	☐	
c) CISC emphasizes optional use of register while RISC does not.	☐	

5.3 RISC ARCHITECTURE

Let us first list some important considerations of RISC architecture:

1. The RISC functions are kept simple unless there is a very good reason to do otherwise. A new operation that increases execution time of an instruction by 10 per cent can be added only if it reduces the size of the code by at least 10 per cent. Even greater reductions might be necessary if the extra modification necessitates a change in design.
2. Micro-instructions stored in the control unit cannot be faster than simple instructions, as the cache is built from the same technology as writable control unit store, a simple instruction may be executed at the same speed as that of a micro-instruction.
3. Microcode is not magic. Moving software into microcode does not make it better; it just makes it harder to change. The runtime library of RISC has all the characteristics of functions in microcode, except that it is easier to change.
4. Simple decoding and pipelined execution are more important than program size. Pipelined execution gives a peak performance of one instruction every step. The longest step determines the performance rate of the pipelined machine, so ideally each pipeline step should take the same amount of time.

Compiler should simplify instructions rather than generate complex instructions. RISC compilers try to remove as much work as possible during compile time so that simple instructions can be used. For example, RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used. RISC compilers keep operands that will be reused in registers, rather than repeating a memory access or a calculation. They, therefore, use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched. (Refer to Figure 1(b)).

Thus, the RISC were designed having the following:

- **One instruction per cycle**: A machine cycle is the time taken to fetch two operands from registers, perform the ALU operation on them and store the result in a register. Thus, RISC instruction execution takes about the same time as the micro-instructions on CISC machines. With such simple instruction execution rather than micro-instructions, it can use fast logic circuits for control unit, thus increasing the execution efficiency further.

- **Register-to-register operands:** In RISC machines the operation that access memories are LOAD and STORE. All other operands are kept in registers. This design feature simplifies the instruction set and, therefore, simplifies the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g. integer add and add with carry); on the other hand a CISC machine can have 25 add instructions involving different addressing modes. Another benefit is that RISC encourages the optimization of register use, so that frequently used operands remain in registers.

- **Simple addressing modes:** Another characteristic is the use of simple addressing modes. The RISC machines use simple register addressing having displacement and PC relative modes. More complex modes are synthesized in software from these simple ones. Again, this feature also simplifies the instruction set and the control unit.

- **Simple instruction formats:** RISC uses simple instruction formats. Generally, only one or a few instruction formats are used. In such machines the instruction length is fixed and aligned on word boundaries. In addition, the field locations can also be fixed. Such an instruction format has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur in parallel. Such a design has many advantages. These are:
  - It simplifies the control unit
  - Simple fetching as memory words of equal size are to be fetched
  - Instructions are not across page boundaries.

Thus, RISC is potentially a very strong architecture. It has high performance potential and can support VLSI implementation. Let us discuss these points in more detail.

- **Performance using optimizing compilers**: As the instructions are simple the compilers can be developed for efficient code organization also maximizing register utilization etc. Sometimes even the part of the complex instruction can be executed during the compile time.
- **High performance of Instruction execution:** While mapping of HLL to machine instruction the compiler favours relatively simple instructions. In addition, the control unit design is simple and it uses little or no micro-instructions, thus could execute simple instructions faster than a comparable CISC. Simple instructions support better possibilities of using instruction pipelining.

- **VLSI Implementation of Control Unit:** A major potential benefit of RISC is the VLSI implementation of microprocessor. The VLSI Technology has reduced the delays of transfer of information among CPU components that resulted in a microprocessor. The delays across chips are higher than delay within a chip; thus, it may be a good idea to have the rare functions built on a separate chip. RISC chips are designed with this consideration. In general, a typical microprocessor dedicates about half of its area to the control store in a micro-programmed control unit. The RISC chip devotes only about 6% of its area to the control unit. Another related issue is the time taken to design and implement a processor. A VLSI processor is difficult to develop, as the designer must perform circuit design, layout, and modeling at the device level. With reduced instruction set architecture, this processor is far easier to build.

5.4 THE USE OF LARGE REGISTER FILE

In general, the register storage is faster than the main memory and the cache. Also the register addressing uses much shorter addresses than the addresses for main memory and the cache. However, the numbers of registers in a machine are less as generally the same chip contains the ALU and control unit. Thus, a strategy is needed that will optimize the register use and, thus, allow the most frequently accessed operands to be kept in registers in order to minimize register-memory operations.

Such optimisation can either be entrusted to an optimising complier, which requires techniques for program analysis; or we can follow some hardware related techniques. The hardware approach will require the use of more registers so that more variables can be held in registers for longer periods of time. This technique is used in RISC machines.

On the face of it the use of a large set of registers should lead to fewer memory accesses, however in general about 32 registers were considered optimum. So how does this large register file further optimize the program execution?

Since most operand references are to local variables of a function in C they are the obvious choice for storing in registers. Some registers can also be used for global variables. However, the problem here is that the program follows function call - return so the local variables are related to most recent local function, in addition this call - return expects saving the context of calling program and return address. This also requires parameter passing on call. On return, from a call the variables of the calling program must be restored and the results must be passed back to the calling program.

RISC register file provides a support for such call- returns with the help of register windows. Register files are broken into multiple small sets of registers and assigned to a different function. A function call automatically changes each of these sets. The use from one fixed size window of registers to another, rather than saving registers in memory as done in CISC. Windows for adjacent procedures are overlapped. This feature allows parameter passing without moving the variables at all. The following figure tries to explain this concept:

Assumptions.

Register file contains 138 registers. Let them be called by register number 0 – 137.

The diagram shows the use of registers: when there is call to function A (f_A) which calls function B (f_B) and function B calls function C (f_C).

Registers Nos.	Used for			
0 – 9	Global variables required by f_A, f_B, and f_C	Function A	Function B	Function C
10 – 83	Unused			
84 – 89 (6 Registers)	Used by parameters of f_C that may be passed to next call			Temporary variables of function C
90 – 99 (10 Registers)	Used for local variable of f_C			Local variables of function C
100 – 105 (6 Registers)	Used by parameters that were passed from $f_B \rightarrow f_C$		Temporary variables of function B	Parameters of function C
106 – 115 (10 Registers)	Local variables of f_B		Local variables of function B	
116 – 121 (6 Registers)	Parameters that were passed from f_A to f_B	Temporary variables of function A	Parameters of function B	
122 – 131 (10 Registers)	Local variable of f_A	Local variables of function A		
132 – 138 (6 Registers)	Parameter passed to f_A	Parameters of function A		

Figure 3: Use of three Overlapped Register Windows

Please note the functioning of the registers: at any point of time the global registers and only one window of registers is visible and is addressable as if it were the only set of registers. Thus, for programming purpose there may be only 32 registers. Window in the above example although has a total of 138 registers. This window consists of:

- Global registers which are shareable by all functions.
- Parameters registers for holding parameters passed from the previous function to the current function. They also hold the results that are to be passed back.
- Local registers that hold the local variables, as assigned by the compiler.
- Temporary registers: They are physically the same as the parameter registers at the next level. This overlap permits parameter passing without the actual movement of data.

But what is the maximum function calls nesting can be allowed through RISC? Let us describe it with the help of a circular buffer diagram, technically the registers as above have to be circular in the call return hierarchy.

This organization is shown in the following figure. The register buffer is filled as function A called function B, function B called function C, function C called function D. The function D is the current function. The current window pointer (CWP) points to the register window of the most recent function (function D in this case). Any register references by a machine instruction is added with the contents of this pointer to determine the actual physical registers. On the other hand the saved window pointer identifies the window most recently saved in memory. This action will be needed if a further call is made and there is no space for that call. If function D now calls function E arguments for function E are placed in D's temporary registers indicated by D temp and the CWP is advanced by one window.

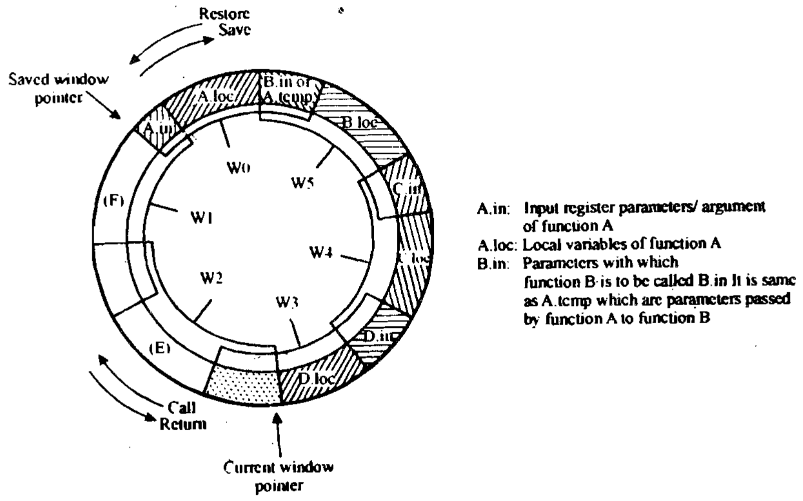

Restore
Save
Saved window pointer
A.in
A.loc
B.in or A.temp
B.loc
C.in
C.loc
D.in
D.loc
(E)
(F)
W0
W1
W2
W3
W4
W5
Call
Return
Current window pointer
A.in: Input register parameters/ argument of function A
A.loc: Local variables of function A
B.in: Parameters with which function B is to be called B.in It is same as A.temp which are parameters passed by function A to function B

Figure 4: Circular-.Buffer Organization of Overlapped Windows

If function E now makes a call to function F, the call cannot be made with the current status of the buffer, unless we free space equivalent to exactly one window. This condition can easily be determined as current window pointer on incrementing will be equal to saved window pointer. Now, we need to create space; how can we do it? The simplest way will be to swap F_A register to memory and use that space. Thus, an N window register file can support N –1 level of function calls.

Thus, the register file, organized in the form as above, is a small fast register buffer that holds most of the variables that are likely to be used heavily. From this point of view the register file acts almost like a cache memory.

So let us find how the two approaches are different:

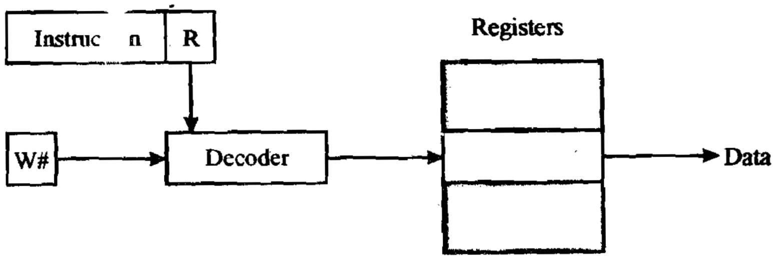
Characteristics of large-register-file and cache organizations

Large Register File	Cache
Hold local variables for almost all functions. This saves time.	Recently used local variables are fetched from main memory for any further use. Dynamic use optimises memory.
The variables are individual.	The transfer from memory is block wise.
Global variables are assigned by the compilers.	It stores recently used variables. It cannot keep track of future use.
Save/restore needed only after the maximum call nesting is over (that is n – 1 open windows) .	Save/restore based on cache replacement algorithms.
It follows faster register addressing.	It is memory addressing.

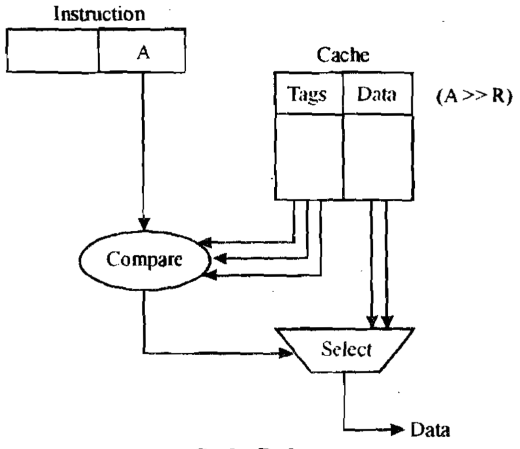
All but one point above basically show comparative equality. The basic difference is due to addressing overhead of the two approaches.

The following figure shows the difference. Small register (R) address is added with current window Pointer W#. This generates the address in register file, which is decoded by decoder for register access. On the other hand Cache reference will be generated from a long memory address, which first goes through comparison logic to ascertain the presence of data, and if the data is present it goes through the select circuit. Thus, for simple variables access register file is superior to cache memory.

However, even in RISC computer, performance can be enhanced by the addition of instruction cache.



(a) Windows based Register file



(b) Cache Reference

Figure 5: Referencing a local Simple Variables

Check Your Progress 2

1. State True or False in the context of RISC architecture: T | F

   a. RISC has a large register file so that more variables can be stored in register or longer periods of time. ☐

   b. Only global variables are stored in registers. ☐

   c. Variables are passed as parameters in registers using temporary registers in a window. ☐

   d. Cache is superior to a large register file as it stores most recently used local scalars. ☐

2. An overlapped register window RISC machine is having 32 registers. Suppose 8 of these registers are dedicated to global variables and the remaining 24 are split for incoming parameters, local and scalar variables and outgoing parameters. What are the ways of allocating these 24 registers in the three categories?

   ....

   ....

   ....

5.5 COMMENTS ON RISC

Let us now try and answer some of the comments that are asked for RISC architectures. Let us provide our suggestions on those.

CISCs provide better support for high-level languages as they include high-level language constructs such as CASE, CALL etc.

Yes CISC architecture tries to narrow the gap between assembly and High Level Language (HLL); however, this support comes at a cost. In fact the support can be measured as the inverse of the costs of using typical HLL constructs on a particular machine. If the architect provides a feature that looks like the HLL construct but runs slowly, or has many options, the compiler writer may omit the feature, or even, the HLL programmer may avoid the construct, as it is slow and cumbersome. Thus, the comment above does not hold.

It is more difficult to write a compiler for a RISC than a CISC.

The studies have shown that it is not so due to the following reasons:

If an instruction can be executed in more ways than one, then more cases must be considered. For it the compiler writer needed to balance the speed of the compilers to get good code. In CISCs compilers need to analyze the potential usage of all available instruction, which is time consuming. Thus, it is recommended that there is at least one good way to do something. In RISC, there are few choices; for example, if an operand is in memory it must first be loaded into a register. Thus, RISC requires simple case analysis, which means a simple compiler, although more machine instructions will be generated in each case.

RISC is tailored for C language and will not work well with other high level languages.

But the studies of other high level languages found that the most frequently executed operations in other languages are also the same as simple HLL constructs found in C, for which RISC has been optimized. Unless a HLL changes the paradigm of programming we will get similar result.

The good performance is due to the overlapped register windows; the reduced instruction set has nothing to do with it.

Certainly, a major portion of the speed is due to the overlapped register windows of the RISC that provide support for function calls. However, please note this register windows is possible due to reduction in control unit size from 50 to 6 per cent. In addition, the control is simple in RISC than CISC, thus further helping the simple instructions to execute faster.

5.6 RISC PIPELINING

Instruction pipelining is often used to enhance performance. Let us consider this in the context of RISC architecture. In RISC machines most of the operations are register-to-register. Therefore, the instructions can be executed in two phases:

F: Instruction Fetch to get the instruction.
E: Instruction Execute on register operands and store the results in register.

In general, the memory access in RISC is performed through LOAD and STORE operations. For such instructions the following steps may be needed:

F: Instruction Fetch to get the instruction
E: Effective address calculation for the desired memory operand
D: Memory to register or register to memory data transfer through bus.

Let us explain pipelining in RISC with an example program execution sample. Take the following program (R indicates register).

LOAD R_A	(Load from memory location A)
LOAD R_B	(Load from memory location B)
ADD R_C ,R_A , R_B	($R_C = R_A + R_B$))
SUB R_D , R_A , R_B	($R_D = R_A - R_B$)
MUL R_E , R_C , R_D	($R_E = R_C \times R_D$)
STOR R_E	(Store in memory location C)
Return to main.	

Load $R_A \leftarrow$ M(A)	F	E	D														
Load $R_B \leftarrow$ M(B)				F	E	D											
Add $R_C \leftarrow R_A + R_B$							F	E									
Sub $R_D \leftarrow R_A - R_B$									F	E							
Mul $R_E \leftarrow R_C \times R_D$											F	E					
Stor $R_E \rightarrow$ M(C)	Time -------------→												F	E	D		
Return	Time = 17 units															F	E
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figure 6: Sequential Execution of Instructions

Figure 7 shows a simple pipelining scheme, in which F and E phases of two different instructions are performed simultaneously. This scheme speeds up the execution rate of the sequential scheme.

Load $R_A \leftarrow$ M(A)	F	E	D								
Load $R_B \leftarrow$ M(B)		F		E	D						
Add $R_C \leftarrow R_A + R_B$				F		E					
Sub $R_D \leftarrow R_A - R_B$						F	E				
Mul $R_E \leftarrow R_C \times R_D$							F	E			
Stor $R_E \rightarrow$ M(C)								F	E	D	
Return									F		E
Time	1	2	3	4	5	6	7	8	9	10	11

Total time = 11 units

Figure 7: Two Way Pipelined Timing

Please note that the pipeline above is not running at its full capacity. This is because of the following problems:

- We are assuming a single port memory thus only one memory access is allowed at a time. Thus, Fetch and Data transfer operations cannot occur at the same time. Thus, you may notice blank in the time slot 3, 5 etc.
- The last instruction is an unconditional jump. Please note that after this instruction the next instruction of the calling program will be executed. Although not visible in this example a branch instruction interrupts the sequential flow of instruction execution. Thus, causing inefficiencies in the pipelined execution.

This pipeline can simply be improved by allowing two memory accesses at a time.

Thus, the modified pipeline would be:

The pipeline may suffer because of data dependencies and branch instructions penalties. A good pipeline has equal phases.

Load $R_A \leftarrow M(A)$	F	E	D					
Load $R_B \leftarrow M(B)$		F	E	D				
Add $R_C \leftarrow R_A + R_B$			F	E				
Sub $R_D \leftarrow R_A - R_B$				F	E			
Mul $R_E = R_C \times R_D$					F	E		
Stor $R_E \rightarrow M(C)$	Time ------→					F	E	D
Return	Time = 8 units						F	E

Figure 8: Three-way Pipelining Timing

Optimization of Pipelining

RISC machines can employ a very efficient pipeline scheme because of the simple and regular instructions. Like all other instruction pipelines RISC pipeline suffer from the problems of data dependencies and branching instructions. RISC optimizes this problem by using a technique called delayed branching.

One of the common techniques used to avoid branch penalty is to pre-fetch the branch destination also. RISC follows a branch optimization technique called delayed jump as shown in the example given below:

Load $R_A \leftarrow M(A)$	F	E	D					
Load $R_B \leftarrow M(B)$		F	E	D				
Add $R_C \leftarrow R_A + R_B$			F	E				
Sub $R_D \leftarrow R_A - R_B$				F	E			
If $R_D < 0$ Return					F	E		
Stor $R_C \rightarrow M(C)$						F	E	D
Return							F	E

(a) The instruction "If $R_D < 0$ Return" may cause pipeline to empty

Load $R_A \leftarrow M(A)$	F	E	D							
Load $R_B \leftarrow M(B)$		F	E	D						
Add $R_C \leftarrow R_A + R_B$			F	E						
Sub $R_D \leftarrow R_A - R_B$				F	E					
If $R_D < 0$ Return					F	E				
NO Operation						F	E			
Stor $R_C \rightarrow M(C)$ Or Return as the case may be								F	E	D
Return									F	E

(b) The No operation instruction causes decision of the If instruction known, thus correct instruction can be fetched.

Load $R_A \leftarrow M(A)$	F	E	D					
Load $R_B \leftarrow M(B)$		F	E	D				
Sub $R_D \leftarrow R_A - R_B$			F	E				
If $R_D < 0$ Return				F	E			
Add $R_C \leftarrow R_A + R_B$					F	E		
Stor $R_C \rightarrow M(C)$						F	E	D
Return							F	E

(c) The branch is calculated before, thus the pipeline need not be emptied. This is delayed branch.

Figure 9: Delayed Branch

Finally, let us summarize the basic differences between CISC and RISC architecture. The following table lists these differences:

CISC	RISC
1. Large number of instructions – from 120 to 350.	1. Relatively fewer instructions - less than 100.
2. Employs a variety of data types and a large number of addressing modes.	2. Relatively fewer addressing modes.
3. Variable-length instruction formats.	3. Fixed-length instructions usually 32 bits, easy to decode instruction format.
4. Instructions manipulate operands residing in memory.	4. Mostly register-register operations. The only memory access is through explicit LOAD/STORE instructions.
5. Number of Cycles Per Instruction (CPI) varies from 1-20 depending upon the instruction.	5. Number of CPI is one as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and instruction formats.
6. GPRs varies from 8-32. But no support is available for the parameter passing and function calls.	6. Large number of GPRs are available that are primarily used as Global registers and as a register based procedural call and parameter passing stack, thus, optimised for structured programming.
7. Microprogrammed Control Unit.	7. Hardwired Control Unit.

Check Your Progress 3

1. What are the problems, which prevent RISC pipelining to achieve maximum speed?

....

....

....

2. How can the above problems be handled?

....

....

....

3. What are the problems of RISC architecture? How are these problems compensated such that there is no reduction in performance?

   ……………………………………………………………………………………………

   ……………………………………………………………………………………………

   ……………………………………………………………………………………………

5.7 SUMMARY

RISC represents new styles of computers that take less time to build yet provide a higher performance. While traditional machines support HLLs with instruction that look like HLL constructs, this machine supports the use of HLLs with instructions that HLL compilers can use efficiently. The loss of complexity has not reduced RISC's functionality; the chosen subset, especially when combined with the register window scheme, emulates more complex machines. It also appears that we can build such a single chip computer much sooner and with much less effort than traditional architectures.

Thus, we see that because of all the features discussed above, the RISC architecture should prove to be far superior to even the most complex CISC architecture.

In this unit we have also covered the details of the pipelined features of the RISC architecture, which further strengthen our arguments for the support of this architecture.

5.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1.
   - Speed of memory is slower than the speed of CPU.
   - Microcode implementation is cost effective and easy.
   - The intention of reducing code size.
   - For providing support for high-level language.

2.
   a) False
   b) False
   c) False

Check Your Progress 2

1.
   (a) True
   (b) False
   (c) True
   (d) False

2. Assume that the number of incoming parameters is equal to the number of outgoing parameters.

   Therefore, Number of locals = 24 –(2 × Number of incoming parameters)

   Return address is also counted as a parameter, therefore, number of incoming parameters is more than or equal to 1 or in other terms the possible combination, are:

Incoming Parameter Registers	Outgoing Parameter Registers	No. of Local Registers
1	1	22
2	2	20
3	3	18
4	4	16
5	5	14
6	6	12
7	7	10
8	8	8
9	9	6
10	10	4
11	11	2
12	12	0

Check Your Progress 3

1. The following are the problems:
   - It has a single port memory reducing the access to one device at a time
   - Branch instruction
   - The data dependencies between the instructions

2. It can be improved by:
   - allowing two memory accesses per phase
   - introducing three phases of approximately equal duration in pipelining
   - causing optimized delayed jumps/loads etc.

3. The problems of RISC architecture are:
   - More instructions to achieve the same amount of work as CISC.
   - Higher instruction traffic
   - However, the cycle time of one instruction per cycle and instruction cache in the chip may compensate for these problems.