COMPUTER ORGANIZATION

(Common to CSE & ISE)

Subject Code: 10CS46	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART – A

UNIT - 1 **6 Hours**

Basic Structure of Computers: Computer Types, Functional Units, Basic Operational Concepts, Bus Structures, Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement, Historical Perspective

Machine Instructions and Programs: Numbers, Arithmetic Operations and Characters, Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing,

UNIT - 2 **7 Hours**

Machine Instructions and Programs *contd.*: Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions

UNIT - 3 **6 Hours**

I**nput/Output Organization:** Accessing I/O Devices, Interrupts – Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Exceptions, Direct Memory Access, Buses

UNIT - 4 **7 Hours**

Input/Output Organization *contd.*: Interface Circuits, Standard I/O Interfaces – PCI Bus, SCSI Bus, USB

PART – B

UNIT - 5 **7 Hours**

Memory System: Basic Concepts, Semiconductor RAM Memories, Read Only Memories, Speed, Size, and Cost, Cache Memories – Mapping Functions, Replacement Algorithms, Performance Considerations, Virtual Memories, Secondary Storage

UNIT - 6 **7 Hours**

Arithmetic: Addition and Subtraction of Signed Numbers, Design of Fast Adders, Multiplication of Positive Numbers, Signed Operand Multiplication, Fast Multiplication, Integer Division, Floating-point Numbers and Operations

UNIT - 7 **6 Hours**

Basic Processing Unit: Some Fundamental Concepts, Execution of a Complete Instruction, Multiple Bus Organization, Hard-wired Control, Microprogrammed Control

UNIT - 8 **6 Hours**

Multicores, Multiprocessors, and Clusters: Performance, The Power Wall, The Switch from Uniprocessors to Multiprocessors, Amdahl's Law, Shared Memory Multiprocessors, Clusters and other Message Passing Multiprocessors, Hardware Multithreading, SISD, IMD, SIMD, SPMD, and Vector.

Text Books:

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.
   (Listed topics only from Chapters 1, 2, 4, 5, 6, 7)
2. David A. Patterson, John L. Hennessy: Computer Organization and Design – The Hardware / Software Interface ARM Edition, 4th Edition, Elsevier, 2009.

TABLE OF CONTENTS

UNIT 1: BASIC STRUCTURE OF COMPUTERS

TYPES OF COMPUTERS

Desktop Computers

• These are most commonly used computers in home, schools and offices.
• This has
  → processing- & storage-units
  → video & audio output-units
  → keyboard & mouse input-units.

Notebook Computers (Laptops)

• This is a compact version of a personal-computer(PC) made as a portable-unit.

Workstations

• These have more computational-power than PC.

Enterprise Systems (Mainframes)

• These are used for business data-processing.
• These have large computational-power and larger storage-capacity than workstations.
• These are referred to as
  → server at low-end and
  → super-computers at high end.

Servers

• These have large database storage-units and can also execute requests from other computers.
• These are used in banks & educational institutions.

Super Computers

• These are used for very complex numerical-calculations.
• These are used in weather forecasting, aircraft design and military applications.

FUNCTIONAL UNITS

• A computer consists of 5 functionally independent main parts: 1)input, 2)memory,3)arithmetic & logic, 4)output and 5)control units.

Input Unit

• The computer accepts the information in the form of program & data through an input-device.
Eg: keyboard

• Whenever a key is pressed, the corresponding letter/digit is automatically translated into its corresponding binary-code and transmitted over a cable to either the memory or the processor.

Memory Unit

• This unit is used to store programs & data.

• There are 2 classes of storage:

1) Primary-storage is a fast-memory that operates at electronic-speed. Programs must be stored in the memory while they are being executed.
2) Secondary-storage is used when large amounts of data & many programs have to be stored. Eg: magnetic disks and optical disks(CD-ROMs).

• The memory contains a large number of semiconductor storage cells(i.e. flip-flops), each capable of storing one bit of information.

• The memory is organized so that the contents of one word can be stored or retrieved in one basic operation.

• Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called RAM (Random Access Memory).

ALU (Arithmetic & Logic Unit)

• This unit is used for performing arithmetic & logical operations.

• Any arithmetic operation is initiated by bringing the required operand into the processor (i.e. registers), where the operation is performed by the ALU.

Output Unit

• This unit is used to send processed-results to the outside world.
Eg: printer, graphic displays etc.

Control Unit

• This unit is used for controlling the activities of the other units (such as memory, I/O device).

• This unit sends control-signals (read/write) to other units and senses their states.

• Data transfers between processor and memory are also controlled by the control-unit through timing-signals.

• Timing-signals are signals that determine when a given action is to take place.

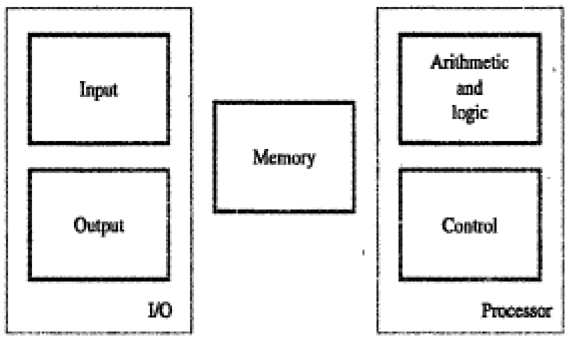


Figure 1.1 Basic functional units of a computer.

BASIC OPERATIONAL CONCEPTS

• The processor contains ALU, control-circuitry and many registers.
• The instruction-register(IR) holds the instruction that is currently being executed.
• The instruction is then passed to the control-unit, which generates the timing-signals that determine when a given action is to take place
• The PC(Program Counter) contains the memory-address of the next-instruction to be fetched & executed.
• During the execution of an instruction, the contents of PC are updated to point to next instruction.
• The processor also contains 'n' general-purpose registers R0 through Rn-1.
• The MAR (Memory Address Register) holds the address of the memory-location to be accessed.
• The MDR (Memory Data Register) contains the data to be written into or read out of the addressed location.

Following are the steps that take place to execute an instruction

• The address of first instruction(to be executed) gets loaded into PC.
• The contents of PC(i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
• After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
• Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
• To fetch an operand, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
• Likewise required number of operands is fetched into processor.
• Finally, ALU performs the desired operation.
• If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
• The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
• At some point during execution, contents of PC are incremented to point to next instruction in the program. [The instruction is a combination of opcode and operand].

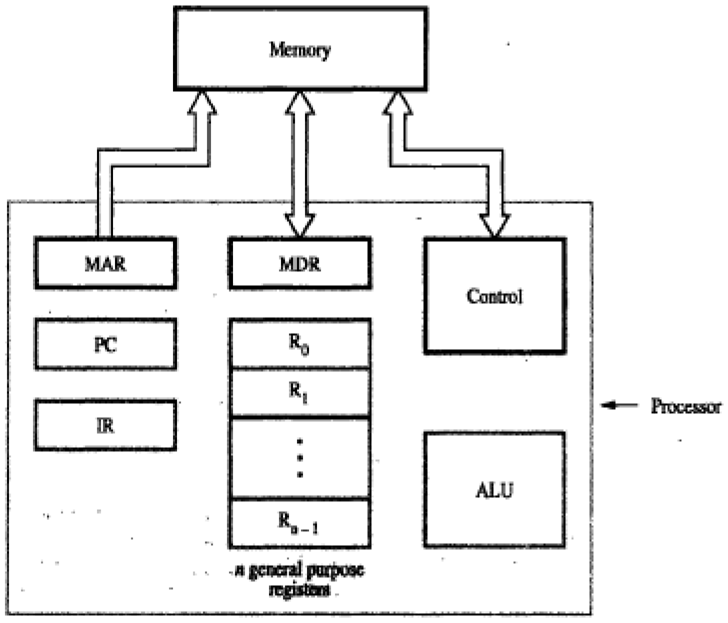


Figure 1.2 Connections between the processor and the memory.

BUS STRUCTURE

• A bus is a group of lines that serves as a connecting path for several devices.
• Bus must have lines for data transfer, address & control purposes.
• Because the bus can be used for only one transfer at a time, only 2 units can actively use the bus at any given time.
• Bus control lines are used to arbitrate multiple requests for use of the bus.
• Main advantage of single bus: Low cost and flexibility for attaching peripheral devices.
• Systems that contain multiple buses achieve more concurrency in operations by allowing 2 or more transfers to be carried out at the same time. Advantage: better performance. Disadvantage: increased cost.
• The devices connected to a bus vary widely in their speed of operation. To synchronize their operational speed, the approach is to include buffer registers with the devices to hold the information during transfers.
• Buffer registers prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers.

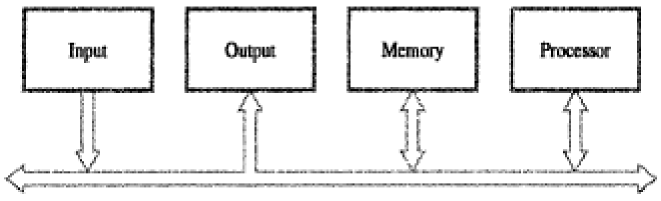


Figure 1.3 Single-bus structure.

SOFTWARE

• System software is a collection of programs that are executed to perform functions such as
  1) Receiving and interpreting user commands
  2) Entering and editing application programs and storing them as files in secondary storage devices
  3) Managing the storage and retrieval of files in secondary storage devices
  4) Controlling I/O units to receive input information and produce output results
  5) Translating programs from source form in object form consisting of machine instructions
  6) Linking and running user-written application programs with existing standard library routines.
• Application programs are usually written in a high-level programming language(such as C,C++)
• A system software program called a compiler translates the high-level language program into a suitable machine language program.
• Text editor is a system program used for entering and editing application programs.
• Operating system is a collection of routines that is used to control the sharing of and interaction among varous computer units as they execute application programs.

PROCESSOR CLOCK

• Processor circuits are controlled by a timing signal called a clock.
• The clock defines regular time intervals called *clock cycles.*
• To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps such that each step can be completed in one clock cycle.
• Let P=length of one clock cycle R=clock rate. Relation between P and R is given by R=1/P which is measured in cycles per second.
• Cycles per second is also called hertz(Hz)

BASIC PERFORMANCE EQUATION

• Let T=processor time required to executed a program
N=actual number of instruction executions
S=average number of basic steps needed to execute one machine instruction
R=clock rate in cycles per second
• The program execution time is given by

$$T = \frac{N \times S}{R} \quad ---(1)$$

• Equ1 is referred to as the basic performance equation.
• To achieve high performance, the computer designer must reduce the value of T, which means reducing N and S, and increasing R.
→ The value of N is reduced if source program is compiled into fewer machine instructions.
→ The value of S is reduced if instructions have a smaller number of basic steps to perform.
→ The value of R can be increased by using a higher frequency clock.
• Care has to be taken while modifying the values since changes in one parameter may affect the other.

PIPELINING & SUPERSCALAR OPERATION

• Normally, the computer executes the instruction in sequence one by one. An improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called *pipelining.*
• Consider the instruction
Add R1,R2,R3 ;instruction 1
Move R4,R5 ;instruction 2
Let us assume that both operations take 3 clock cycles each for completion.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2				Fetch	Decode	Execute

• As shown in above figure, 6 clock cycles are required to complete two operations.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		

• As shown in above figure, if we use pipelining & prefetching, only 4 cycles are required to complete same two operations.
• While executing the Add instruction, the processor can read the Move instruction from memory.
• In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.
• A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor i.e. multiple functional units can be used to execute different instructions parallely. This mode of operation is known as *superscalar execution.*
• With Superscalar arrangement, it is possible to complete the execution of more than one instruction in every clock cycle.

PERFORMANCE MEASUREMENT

• SPEC(System Performance Evaluation Corporation) selects & publishes the standard programs along with their test results for different application domains

• The SPEC rating is computed as follows

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

• If SPEC rating=50 means that the computer under test is 50times as fast as reference computer.

• The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed.

• Let $SPEC_i$ be the rating for program i in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i\right)^{\frac{1}{n}}$$

where n=number of programs in the suite

INSTRUCTION SET: CISC AND RISC

Architectural characteristics	***CISC***	***RISC***
Instruction formats	instructions with variable formats	instructions with fixed format and most are register based instruction
addressing modes	12-24	limited to 3-5
General purpose register and cache design	8-24 GPRs. Mostly with a unified cache for instruction and data	large number of GPRs with mostly split data cache and instruction cache
Clock rate & CPI	33-50 Mhz with CPI between 2 and 15	50-150Mhz with 1 clock cycle for almost all instruction and an average CPI<15
CPU control	mostly microcoded using control memory	mostly hardwired without control memory

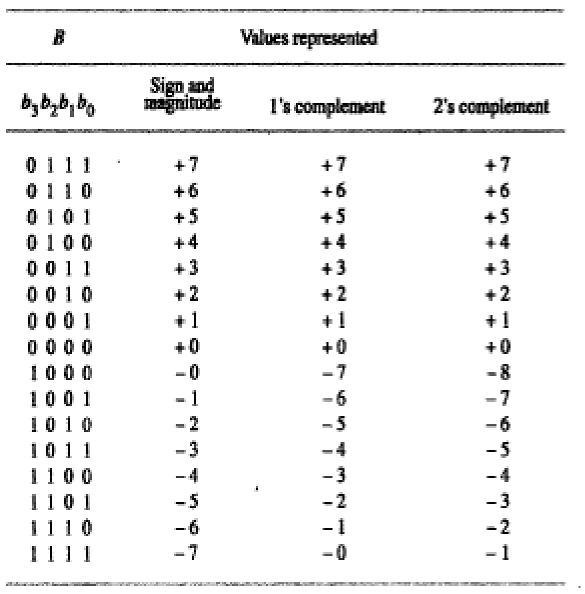
UNIT 1(CONT.): MACHINE INSTRUCTIONS & PROGRAMS

NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

NUMBER REPRESENTATION

• Numbers can be represented in 3 formats:
1) Sign and magnitude
2) 1's complement
3) 2's complement

• In all three formats, MSB=0 for +ve numbers & MSB=1 for -ve numbers.
• In the sign-and-magnitude system, negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value. For example, +5 is represented by 0101 &
-5 is represented by 1101.
• In 1's complement representation, negative values are obtained by complementing each bit of the corresponding positive number. For example, -3 is obtained by complementing each bit in 0011 to yield 1100. (In other words, the operation of forming the 1's complement of a given number is equivalent to subtracting that number from 2^n-1).
• In the 2's complement system, forming the 2's complement of a number is done by subtracting that number from 2^n.
(In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).
• The 2's complement system yields the most efficient way to carry out addition and subtraction operations.



B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Figure 2.1 Binary, signed-integer representations.

ADDITION OF POSITIVE NUMBERS

• Consider adding two 1-bit numbers.
• The sum of 1 & 1 requires the 2-bit vector 10 to represent the value 2. We say that sum is 0 and the carry-out is 1

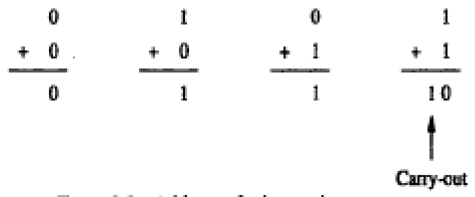


Figure 2.2 Addition of 1-bit numbers.

ADDITION & SUBTRACTION OF SIGNED NUMBERS

• Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system.

1) To add two numbers, add their n-bits and ignore the carry-out signal from the MSB position. The sum will be algebraically correct value as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$ (Figure 2.4).

2) To subtract two numbers X and Y(that is to perform X-Y),take the 2's complement of Y and then add it to X as in rule 1.Result will be algebraically correct, if it lies in the range (2^{n-1}) to $+(2^{n-1}-1)$.

• When the result of an arithmetic operation is outside the representable-range, an *arithmetic overflow* is said to occur.

• To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called *sign extension.*

• In 1's complement representation, the result obtained after an addition operation is not always correct. The carry-out(c_n) cannot be ignored. If c_n=0, the result obtained is correct. If c_n=1, then a 1 must be added to the result to make it correct.

OVERFLOW IN INTEGER ARITHMETIC

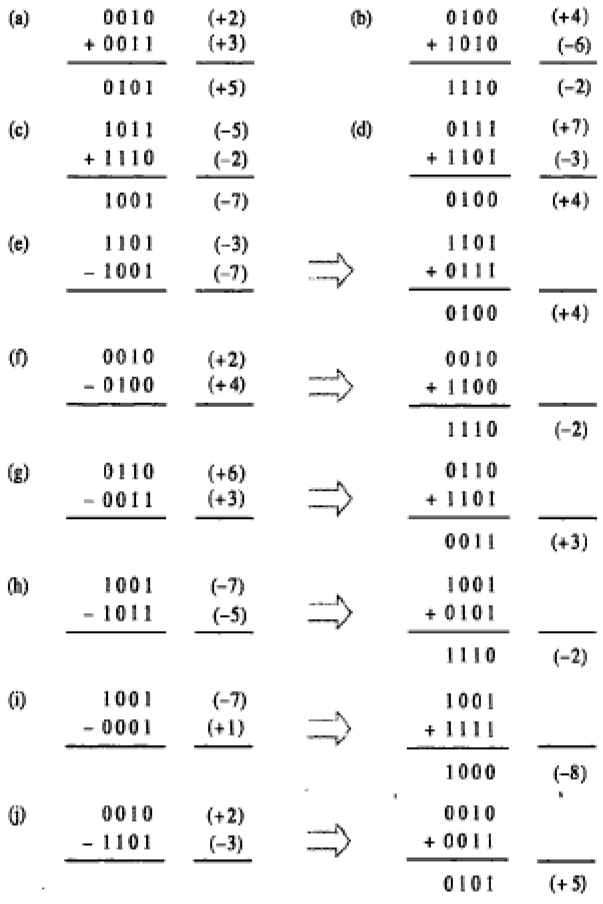
• When the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.

• For example, when using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output sum S is 1011, which is the code for -5, an incorrect result.

• An overflow occurs in following 2 cases

1) Overflow can occur only when adding two numbers that have the same sign.

2) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.



```
(a)    0010   (+2)         (b)    0100   (+4)
     + 0011   (+3)              + 1010   (-6)
     ------   ----              ------   ----
       0101   (+5)                1110   (-2)

(c)    1011   (-5)         (d)    0111   (+7)
     + 1110   (-2)              + 1101   (-3)
     ------   ----              ------   ----
       1001   (-7)                0100   (+4)

(e)    1101   (-3)                1101
     - 1001   (-7)     ⇒        + 0111
     ------   ----              ------   ----
                                  0100   (+4)

(f)    0010   (+2)                0010
     - 0100   (+4)     ⇒        + 1100
     ------   ----              ------   ----
                                  1110   (-2)

(g)    0110   (+6)                0110
     - 0011   (+3)     ⇒        + 1101
     ------   ----              ------   ----
                                  0011   (+3)

(h)    1001   (-7)                1001
     - 1011   (-5)     ⇒        + 0101
     ------   ----              ------   ----
                                  1110   (-2)

(i)    1001   (-7)                1001
     - 0001   (+1)     ⇒        + 1111
     ------   ----              ------   ----
                                  1000   (-8)

(j)    0010   (+2)                0010
     - 1101   (-3)     ⇒        + 0011
     ------   ----              ------   ----
                                  0101   (+5)
```

Figure 2.4 2's-complement add and subtract operations.

MEMORY LOCATIONS & ADDRESSES

• The memory consists of many millions of storage cells (flip-flops), each of which can store a bit of information having the value 0 or 1 (Figure 2.5).
• Each group of n bits is referred to as a word of information, and n is called the word length.
• The word length can vary from 8 to 64bits.
• A unit of 8 bits is called a byte.
• Accessing the memory to store or retrieve a single item of information (either a word or a byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through 2^k-1 as the addresses of successive locations in the memory).
• If 2^k=number of addressable locations, then 2^k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2^{24} locations (16MB).

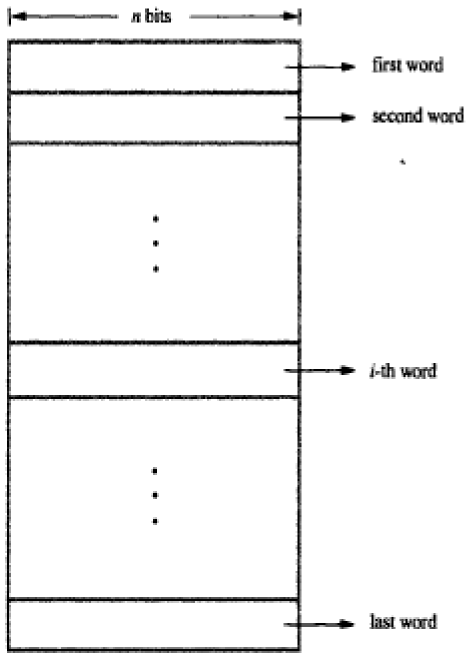


Figure 2.5 Memory words.

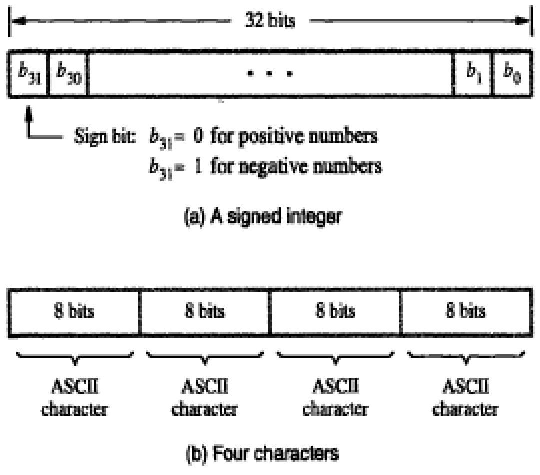


Figure 2.6 Examples of encoded information in a 32-bit word.

Note:
• Characters can be letters of the alphabet, decimal digits, punctuation marks and so on.
• Characters are represented by codes that are usually 8 bits long. E.g. ASCII code
• The three basic information quantities are: bit, byte and word.
• A byte is always 8 bits, but the word length typically ranges from 1 to 64 bits.
• It is impractical to assign distinct addresses to individual bit locations in the memory.

BYTE ADDRESSABILITY

• In byte addressable memory, successive addresses refer to successive byte locations in the memory.
• Byte locations have addresses 0, 1, 2. . . . .
• If the word length is 32 bits, successive words are located at addresses 0, 4, 8. .with each word having 4 bytes.

BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

• There are two ways in which byte addresses are arranged.

1) Big-endian assignment: lower byte addresses are used for the more significant bytes of the word (Figure 2.7).
2) Little-endian: lower byte addresses are used for the less significant bytes of the word

• In both cases, byte addresses 0, 4, 8. . . . . are taken as the addresses of successive words in the memory.

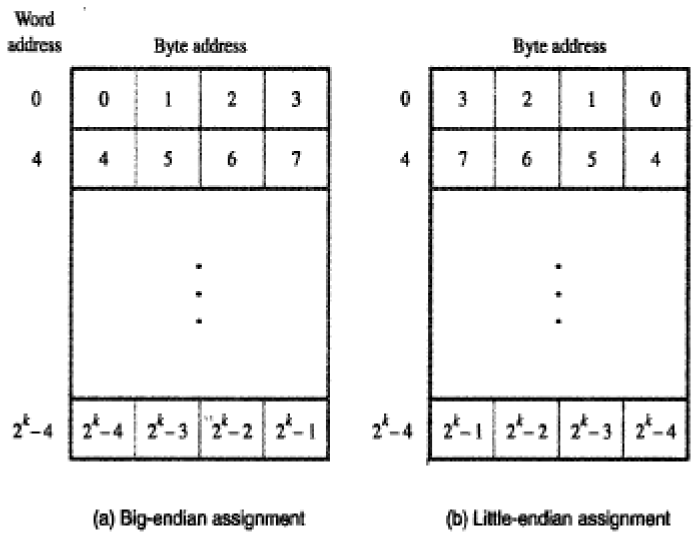

Word address
Byte address
0
4
2^k-4
0 1 2 3
4 5 6 7
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
4
2^k-4
3 2 1 0
7 6 5 4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

Figure 2.7 Byte and word addressing.

WORD ALIGNMENT

• Words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.
• For example, if the word length is 16(2 bytes), aligned words begin at byte addresses 0, 2, 4 . . . . . And for a word length of 64, aligned words begin at byte addresses 0, 8, 16. . . . .
• Words are said to have unaligned addresses, if they begin at an arbitrary byte address.

ACCESSING NUMBERS, CHARACTERS & CHARACTERS STRINGS

• A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.
• There are two ways to indicate the length of the string

→ a special control character with the meaning "end of string" can be used as the last character in the string, or

→ a separate memory word location or processor register can contain a number indicating the length of the string in bytes

MEMORY OPERATIONS

• Two basic operations involving the memory are: Load(Read/Fetch) and Store(Write).

• The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.

• The steps for Load operation:

1) Processor sends the address of the desired location to the memory
2) Processor issues 'read' signal to memory to fetch the data
3) Memory reads the data stored at that address
4) Memory sends the read data to the processor

• The Store operation transfers the information from the processor register to the specified memory location. This will destroy the original contents of that memory location.

• The steps for Store operation are:

1) Processor sends the address of the memory location where it wants to store data
2) Processor issues 'write' signal to memory to store the data
3) Content of register(MDR) is written into the specified memory location

INSTRUCTIONS & INSTRUCTION SEQUENCING

• A computer must have instructions capable of performing 4 types of operations:

1) Data transfers between the memory and the processor registers (MOV, PUSH, POP, XCHG),
2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT),
3) Program sequencing and control (CALL.RET, LOOP, INT),
4) I/0 transfers (IN, OUT),

REGISTER TRANSFER NOTATION (RTN)

• We identify a memory location by a symbolic name (in uppercase alphabets).

For example, LOC, PLACE, NUM etc indicate memory locations.
R0, R5 etc indicate processor register.
DATAIN, OUTSTATUS etc indicate I/O registers.

• For example,

R<-[LOC] means that the contents of memory location LOC are transferred into processor register R1 (The contents of a location are denoted by placing square brackets around the name of the location).

R3<-[R1]+[R2] indicates the operation that adds the contents of registers R1 and R2 ,and then places their sum into register R3.

• This type of notation is known as RTN(Register Transfer Notation).

ASSEMBLY LANGUAGE NOTATION

• To represent machine instructions and programs, assembly language format can be used.

• For example,

Move LOC, R1; This instruction transfers data from memory-location LOC to processor-register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

Add R1, R2, R3; This instruction adds 2 numbers contained in processor-registers R1 and R2, and places their sum in R3.

Note:

• A computer performs its task according to the program stored in memory. A program is a collection of instructions which tell the processor to perform a basic operation like addition, reading from keyboard etc.

• Possible locations that may be involved in data transfers are memory locations, processor registers or registers in the I/O subsystem.

BASIC INSTRUCTION TYPES

• *C=A+B;* This statement is a command to the computer to add the current values of the two variables A and B, and to assign the sum to a third variable C.

• When the program is compiled, each variable is assigned a distinct address in memory.

• The contents of these locations represent the values of the three variables

• The statement *C<-[A]+[B]* indicates that the contents of memory locations A and B are fetched from memory, transferred to the processor, sum is computed and then result is stored in memory location C.

Three-Address Instruction

• The instruction has general format

Operation Source1, Source2, Destination

• For example, *Add A, B, C;* operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed.

Two-Address Instruction

• The instruction has general format

Operation Source, Destination

• For example, *Add A, B;* performs the operation B<-[A]+[B].

• When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.

• The operation C<-[A]+[B] can be performed by the two-instruction sequence

```
Move B, C
Add A, C
```

One-Address Instruction

• The instruction has general format

Operation Source/Destination

• For example, *Add A ;* Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator.

• *Load A;* This instruction copies the contents of memory location A into the accumulator and

Store A; This instruction copies the contents of the accumulator into memory location A.

• The operation C<-[A]+[B] can be performed by executing the sequence of instructions

```
Load A
Add B
Store C
```

• The operand may be a source or a destination depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Zero-Address Instruction

• The locations of all operands are defined implicitly. The operands are stored in a structure called pushdown stack. In this case, the instructions are called zero-address instructions.

Note:

• Access to data in the registers is much faster than to data stored in memory locations because the registers are inside the processor.

• Let Ri represent a general-purpose register. The instructions

```
Load A,Ri
Store Ri,A
Add A,Ri
```

are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

• In processors where arithmetic operations as allowed only on operands that are in processor registers, the C=A+B task can be performed by the instruction sequence

```
Move A,Ri
Move B,Rj
Add Ri,Rj
Move Rj,C
```

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

• The program is executed as follows:

1) Initially, the address of the first instruction is loaded into PC (Program counter is a register which holds the address of the next instruction to be executed)
2)Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing* (Figure 2.8)
3) During the execution of each instruction, the PC is incremented by 4 to point to the next instruction.
4) Executing given instruction is a two-phase procedure.
    i) In fetch phase, the instruction is fetched from the memory location (whose address is in the PC) and placed in the IR of the processor
    ii) In execute phase, the contents of IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor.

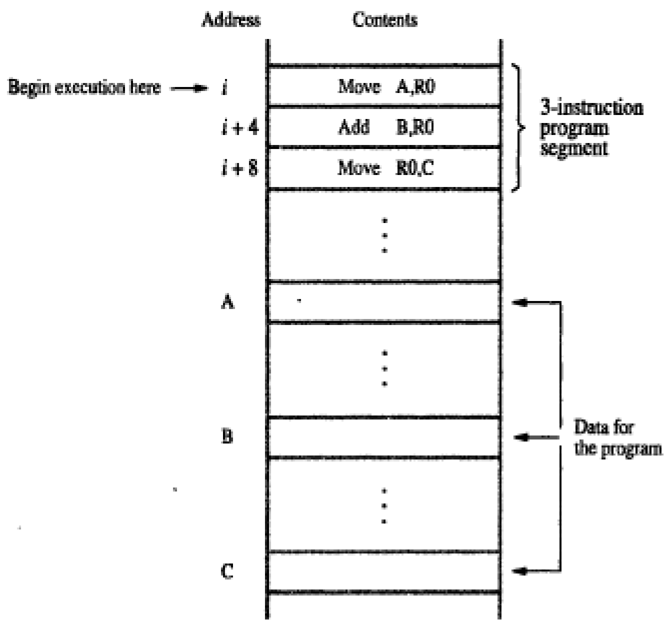


Figure 2.8 A program for C ← [A] + [B].

BRANCHING

• Consider the task of adding a list of n numbers (Figure 2.10).
• The loop is a straight line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch>0.
• During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0.
• Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program.
• Within the body of the loop, the instruction *Decrement R1* reduces the contents of R1 by 1 each time through the loop.
• Then Branch instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the branch target.
• A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

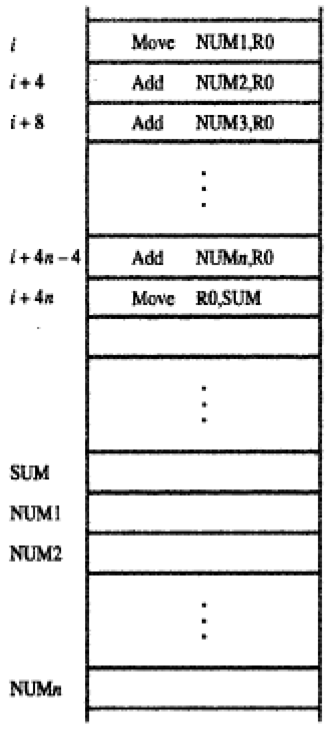


Figure 2.9 A straight-line program for adding *n* numbers.

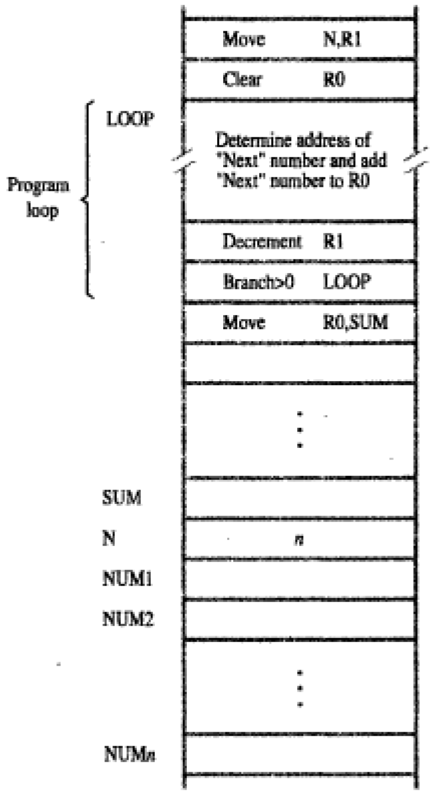


Figure 2.10 Using a loop to add *n* numbers.

CONDITION CODES

• The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called *condition code flags.*
• These flags are grouped together in a special processor-register called the *condition code register* (or statue register).
• Four commonly used flags are

→ N (negative) set to 1 if the result is negative, otherwise cleared to 0
→ Z (zero) set to 1 if the result is 0; otherwise, cleared to 0
→ V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
→ C (carry) set to 1 if a carry-out results from the operation; otherwise cleared to 0

UNIT 2: MACHINE INSTRUCTIONS & PROGRAMS (CONT.)

ADDRESSING MODES

• The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes* (Table 2.1).

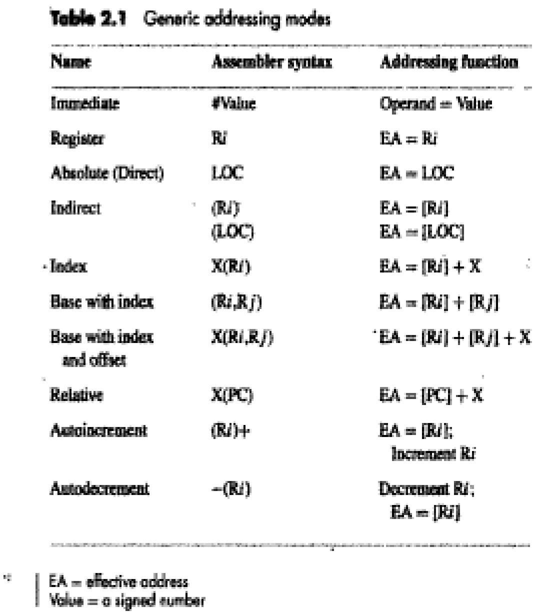


Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLE AND CONSTANTS

• Variables & constants are the simplest data-types and are found in almost every computer program.
• In assembly language, a variable is represented by allocating a register (or memory-location) to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Register Mode

• The operand is the contents of a register.
• The name (or address) of the register is given in the instruction.
• Registers are used as temporary storage locations where the data in a register are accessed.
• For example, the instruction,

Move R1, R2 ;Copy content of register R1 into register R2

Absolute (Direct) Mode

• The operand is in a memory-location.
• The address of memory-location is given explicitly in the instruction.
• For example, the instruction,

Move LOC, R2 ;Copy content of memory-location LOC into register R2

Immediate Mode

• The operand is given explicitly in the instruction.
• For example, the instruction,

Move #200, R0 ;Place the value 200 in register R0

• Clearly, the immediate mode is only used to specify the value of a source-operand.

INDIRECTION AND POINTERS

• In this case, the instruction does not give the operand or its address explicitly; instead, it provides information from which the memory-address of the operand can be determined. We refer to this address as the *effective address(EA)* of the operand.

Indirect Mode

• The EA of the operand is the contents of a register(or memory-location) whose address appears in the instruction.

• The register (or memory-location) that contains the address of an operand is called a *pointer*. {The indirection is denoted by ( ) sign around the register or memory-location}.

E.g: *Add (R1),R0* ;The operand is in memory. Register R1 gives the effective-address(B) of the operand. The data is read from location B and added to contents of register R0

* To execute the Add instruction in fig (a), the processor uses the value which is in register R1, as the EA of the operand.

* It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0.

* Indirect addressing through a memory location is also possible as shown in fig (b). In this case, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand

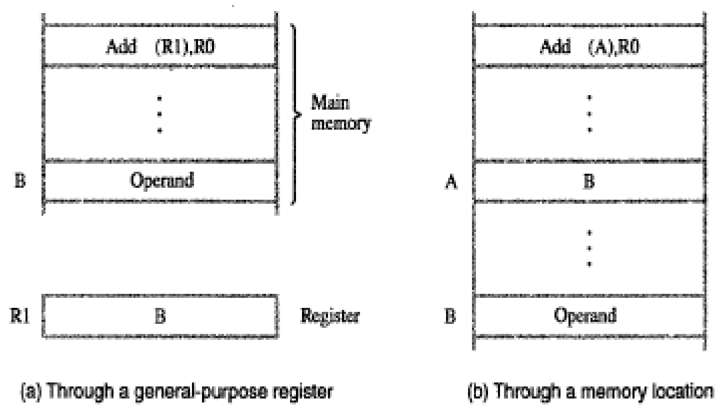
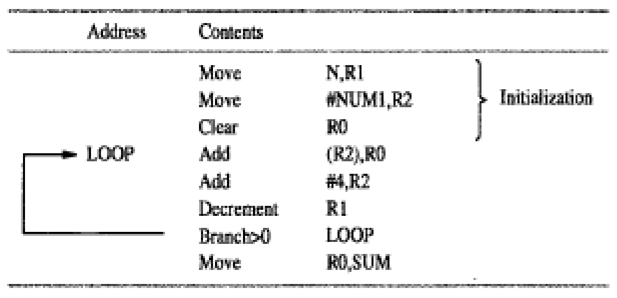

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure 2.11 Indirect addressing.



```
Address   Contents
          Move        N,R1        }
          Move        #NUM1,R2    } Initialization
          Clear       R0          }
LOOP      Add         (R2),R0
          Add         #4,R2
          Decrement   R1
          Branch>0    LOOP
          Move        R0,SUM
```

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

• In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.

• The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.

• The first two instructions in the loop implement the unspecified instruction block starting at LOOP.

• The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0.

• The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

INDEXING AND ARRAYS

• A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode

• The operation is indicated as X(Ri)
where X=the constant value contained in the instruction
Ri=the name of the index register

• The effective-address of the operand is given by EA=X+[Ri]

• The contents of the index-register are not changed in the process of generating the effective-address.

• In an assembly language program, the constant X may be given either
→ as an explicit number or
→ as a symbolic-name representing a numerical value.

* Fig(a) illustrates two ways of using the Index mode. In fig(a), the index register, R1, contains the address of a memory location, and the value X defines an offset(also called a displacement) from this address to the location where the operand is found.

* An alternative use is illustrated in fig(b). Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

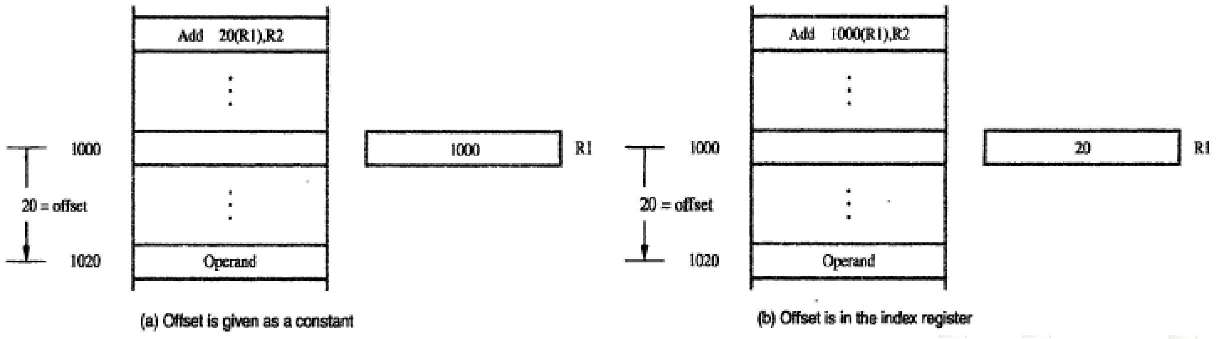


Figure 2.13 Indexed addressing.

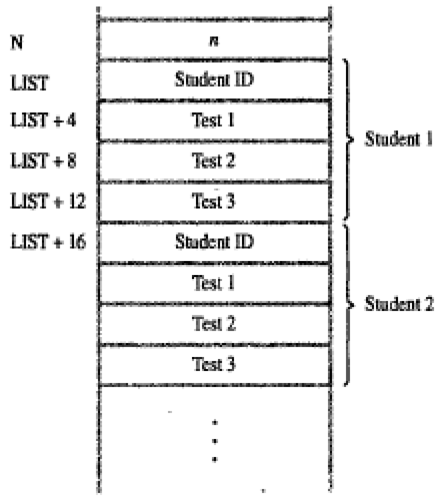
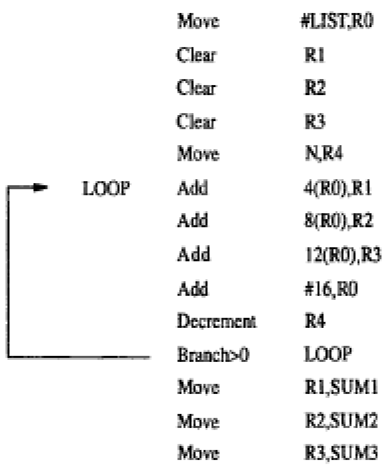


Figure 2.14 A list of students' marks.



```
        Move       #LIST,R0
        Clear      R1
        Clear      R2
        Clear      R3
        Move       N,R4
LOOP    Add        4(R0),R1
        Add        8(R0),R2
        Add        12(R0),R3
        Add        #16,R0
        Decrement  R4
        Branch>0   LOOP
        Move       R1,SUM1
        Move       R2,SUM2
        Move       R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

Base with Index Mode

• Another version of the Index mode uses 2 registers which can be denoted as
(Ri, Rj)

• Here, a second register may be used to contain the offset X.

• The second register is usually called the *base register*.

• The effective-address of the operand is given by EA=[Ri]+[Rj]

• This form of indexed addressing provides more flexibility in accessing operands, because
both components of the effective address can be changed.

Base with Index & Offset Mode

• Another version of the Index mode uses 2 registers plus a constant, which can be denoted as
X(Ri, Rj)

• The effective-address of the operand is given by EA=X+[Ri]+[Rj]

• This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

RELATIVE MODE

• This is similar to index-mode with an exception: The effective address is determined using the PC in place of the general purpose register Ri.
• The operation is indicated as X(PC).
• X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
• Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
• This mode is used commonly in conditional branch instructions.
• An instruction such as

```
        Branch > 0 LOOP         ;Causes program execution to go to the branch target location
                                 identified by name LOOP if branch condition  is satisfied.
```

ADDITIONAL ADDRESSING MODES

• The following 2 modes are useful for accessing data items in successive locations in the memory.

Auto-increment Mode

• The effective-address of operand is the contents of a register specified in the instruction (Fig: 2.16).
• After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
• Implicitly, the increment amount is 1.
• This mode is denoted as

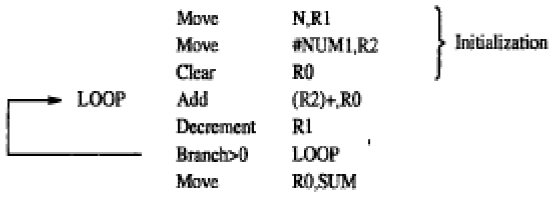
```
                        (Ri)+     ;where Ri=pointer register
```

Auto-decrement Mode

• The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
• This mode is denoted as

```
                        -(Ri)     ;where Ri=pointer register
```

• These 2 modes can be used together to implement an important data structure called a stack.



```
            Move        N,R1        }
            Move        #NUM1,R2    } Initialization
            Clear       R0          }
    LOOP    Add         (R2)+,R0
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

ASSEMBLY LANGUAGE

• A complete set of symbolic names and rules for their use constitute an assembly language.
• The set of rules for using the mnemonics in the specification of complete instructions and programs is called the *syntax* of the language.
• Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*.
• The user program in its original alphanumeric text formal is called a *source program*, and the assembled machine language program is called an *object program*.
• Move instruction is written is

MOVE R0,SUM ;The mnemonic MOVE represents the binary pattern, or OP code, for the operation performed by the instruction.

• The instruction

ADD #5,R3 ;Adds the number 5 to the contents of register R3 and puts the result back into register R3.

ASSEMBLER DIRECTIVES

• EQU informs the assembler about the value of an identifier (Figure: 2.18).

Ex*: SUM EQU 200* ; This statement informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program.

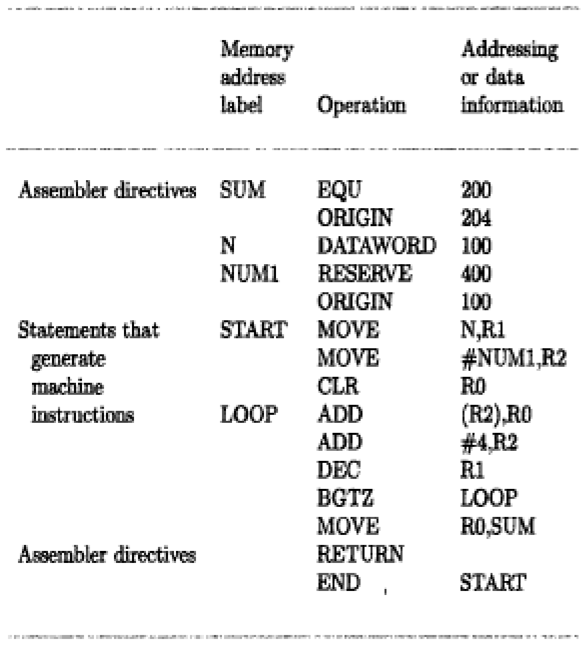
• ORIGIN tells the assembler about the starting-address of memory-area to place the data block.
• DATAWORD directive tells the assembler to load a value (say 100) into the location (say 204).

Ex: *N DATAWORD 100*

• RESERVE directive declares that a memory-block of 400 bytes is to be reserved for data and that the name NUM1 is to be associated with address 208.

Ex: *NUM1 RESERVE 400*

• END directive tells the assembler that this is the end of the source-program text.
• RETURN directive identifies the point at which execution of the program should be terminated.
• Any statement that makes instructions or data being placed in a memory-location may be given a label.
• The label(say N or NUM1) is assigned a value equal to the address of that location.



	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Figure 2.18 Assembly language representation for the program in Figure 2.17.

GENERAL FORMAT OF A STATEMENT

• Most assembly languages require statements in a source program to be written in the form:

Label Operation Operands Comment

→ Label is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.
→ The Operation field contains the OP-code mnemonic of the desired instruction or assembler → The Operand field contains addressing information for accessing one or more operands, depending on the type of instruction.
→ The Comment field is used for documentation purposes to make the program easier to understand.

ASSEMBLY AND EXECUTION OF PRGRAMS

• Programs written in an assembly language are automatically translated into a sequence of machine instructions by the assembler.

• Assembler program

→ replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.
→ replaces all names and labels with their actual values.
→ assigns addresses to instructions & data blocks, starting at the address given in the ORIGIN directive.
→ inserts constants that may be given in DATAWORD directives.
→ reserves memory-space as requested by RESERVE directives.

• As the assembler scans through a source-program, it keeps track of all names of numerical-values that correspond to them in a symbol-table. Thus, when a name appears a second time, it is replaced with its value from the table. Hence, such an assembler is called a *two-pass assembler.*

• The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a *loader program* is used.

• *Debugger program* is used to help the user find the programming errors.

• Debugger program enables the user

→ to stop execution of the object-program at some points of interest and
→ to examine the contents of various processor registers and memory-location

BASIC INPUT/OUTPUT OPERATIONS

• Consider the problem of moving a character-code from the keyboard to the processor.

For this transfer, buffer-register(DATAIN) & a status control flags(SIN) are used.

• Striking a key stores the corresponding character-code in an 8-bit buffer-register(DATAIN) associated with the keyboard (Figure: 2.19).

• To inform the processor that a valid character is in DATAIN, a SIN is set to 1.

• A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN.

• When the character is transferred to the processor, SIN is automatically cleared to 0.

• If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

• An analogous process takes place when characters are transferred from the processor to the display. A buffer-register, DATAOUT, and a status control flag, SOUT are used for this transfer.

• When SOUT=1, the display is ready to receive a character.

• The transfer of a character to DATAOUT clears SOUT to 0.

• The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a *device interface*.

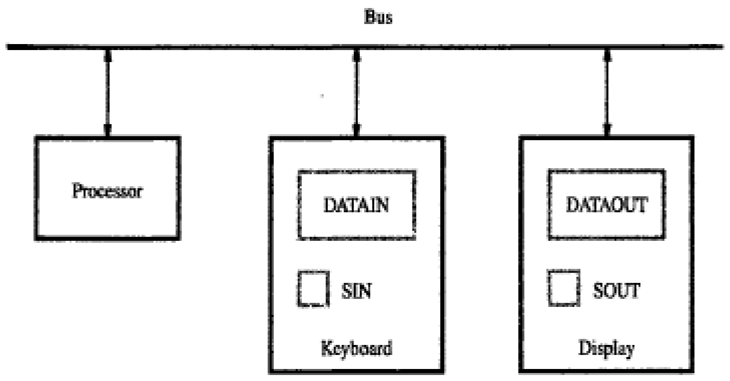


Figure 2.19 Bus connection for processor, keyboard, and display.

• Following is a program to read a line of characters and display it

```
        Move       #LOC,R0         Initialize pointer register R0 to point to the
                                     address of the first location in memory
                                     where the characters are to be stored.
READ    TestBit    #3,INSTATUS     Wait for a character to be entered
        Branch=0   READ              in the keyboard buffer DATAIN.
        MoveByte   DATAIN,(R0)     Transfer the character from DATAIN into
                                     the memory (this clears SIN to 0).
ECHO    TestBit    #3,OUTSTATUS    Wait for the display to become ready.
        Branch=0   ECHO
        MoveByte   (R0),DATAOUT    Move the character just read to the display
                                     buffer register (this clears SOUT to 0).
        Compare    #CR,(R0)+       Check if the character just read is CR
                                     (carriage return). If it is not CR, then
        Branch≠0   READ              branch back and read another character.
                                     Also, increment the pointer to store the
                                     next character.
```

MEMORY-MAPPED I/O

• Some address values are used to refer to peripheral device buffer-registers such as DATAIN and DATAOUT.

• No special instructions are needed to access the contents of the registers; data can be transferred between these registers and the processor using instructions such as Move, Load or Store.

• For example, contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

MoveByte DATAIN,R1

• The MoveByte operation code signifies that the operand size is a byte.

• The Testbit instruction tests the state of one bit in the destination, where the bit position to be tested is indicated by the first operand.

STACKS

• A stack is a list of data elements with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom (Figure: 2.21).

• The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

• A processor-register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the SP (Stack Pointer).

• If we assume a byte-addressable memory with a 32-bit word length,

→ The push operation can be implemented as

Subtract #4,SR
Move NEWITEM,(SP)

→ The pop operation can be implemented as

Move (SP),ITEM
Add #4,SP

• Routine for a safe pop and push operation as follows

```
SAFEPOP   Compare    #2000,SP      Check to see if the stack pointer contains
          Branch>0   EMPTYERROR      an address value greater than 2000. If it
                                     does, the stack is empty. Branch to the
                                     routine EMPTYERROR for appropriate
                                     action.
          Move       (SP)+,ITEM    Otherwise, pop the top of the stack into
                                     memory location ITEM.
```

(a) Routine for a safe pop operation

```
SAFEPUSH  Compare    #1500,SP       Check to see if the stack pointer
          Branch≤0   FULLERROR        contains an address value equal
                                      to or less than 1500. If it does, the
                                      stack is full. Branch to the routine
                                      FULLERROR for appropriate action.
          Move       NEWITEM,-(SP)  Otherwise, push the element in memory
                                      location NEWITEM onto the stack.
```

(b) Routine for a safe push operation

Figure 2.23 Checking for empty and full errors in pop and push operations.

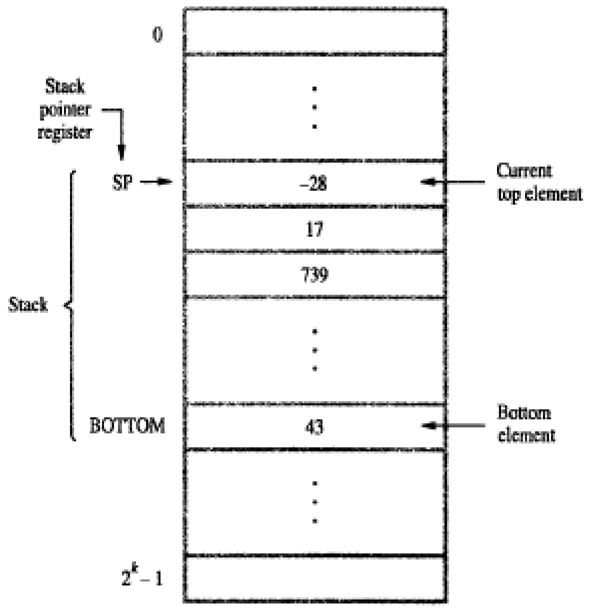


Figure 2.21 A stack of words in the memory.

QUEUE

• Data are stored in and retrieved from a queue on a FIFO basis.
• Difference between stack and queue?

1) One end of the stack is fixed while the other end rises and falls as data are pushed and popped.
2) A single pointer is needed to point to the top of the stack at any given time.
On the other hand, both does of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of the two does of the queue.
3) Without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer.

SUBROUTINES

• A subtask consisting of a set of instructions which is executed many times is called a *subroutine.*
• The program branches to a subroutine with a Call instruction (Figure: 2.24).
• Once the subroutine is executed, the calling-program must resume execution starting from the instruction immediately following the Call instructions i.e. control is to be transferred back to the calling-program. This is done by executing a Return instruction at the end of the subroutine.
• The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method.
• The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*.
• When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.
• The Call instruction is a special branch instruction that performs the following operations:

→ Store the contents of PC into link-register.
→ Branch to the target-address specified by the instruction.

• The Return instruction is a special branch instruction that performs the operation:

→ Branch to the address contained in the link-register.

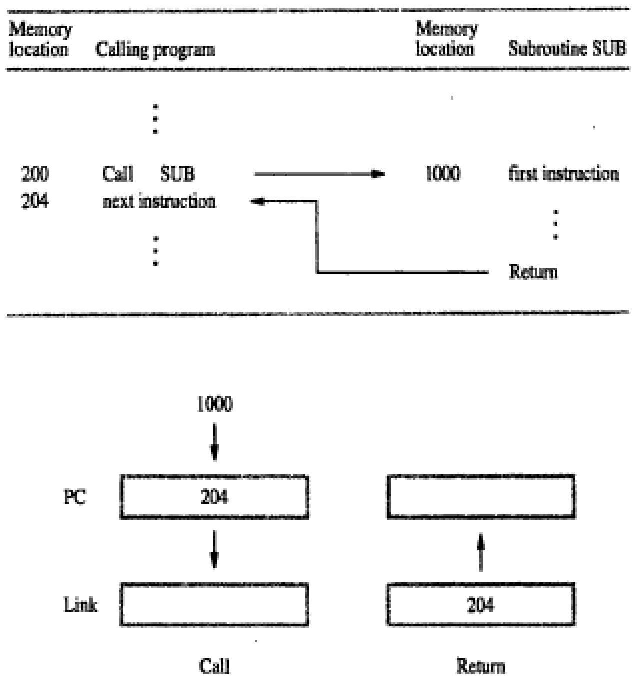


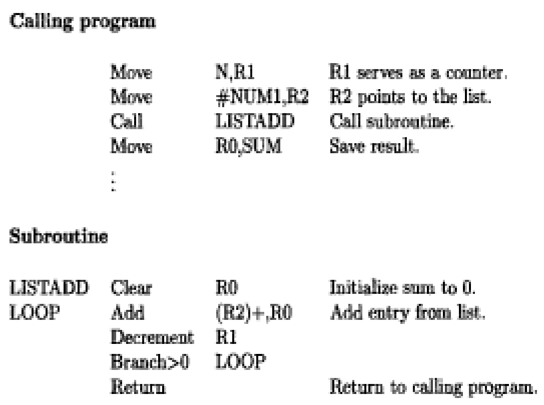
Figure 2.24 Subroutine linkage using a link register.

SUBROUTINE NESTING AND THE PROCESSOR STACK

• *Subroutine nesting* means one subroutine calls another subroutine.
• In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
• Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
• Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
• The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
• This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
• SP is used to point to the processor-stack.
• Call instruction pushes the contents of the PC onto the processor-stack.
Return instruction pops the return-address from the processor-stack into the PC.

PARAMETER PASSING

• The exchange of information between a calling-program and a subroutine is referred to as *parameter passing* (Figure: 2.25).
• The parameters may be placed in registers or in memory-location, where they can be accessed by the subroutine.
• Alternatively, parameters may be placed on the processor-stack used for saving the return-address • Following is a program for adding a list of numbers using subroutine with the parameters passed through registers.



```
Calling program

            Move        N,R1        R1 serves as a counter.
            Move        #NUM1,R2    R2 points to the list.
            Call        LISTADD     Call subroutine.
            Move        R0,SUM      Save result.
            ⋮

Subroutine

LISTADD     Clear       R0          Initialize sum to 0.
LOOP        Add         (R2)+,R0    Add entry from list.
            Decrement   R1
            Branch>0    LOOP
            Return                  Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

STACK FRAME

• *Stack frame* refers to locations that constitute a private work-space for the subroutine(Figure:2.26).

• The work-space is

→ created at the time the subroutine is entered &

→ freed up when the subroutine returns control to the calling-program.

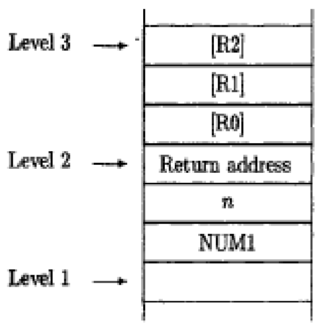
• Following is a program for adding a list of numbers using subroutine with the parameters passed to stack

```
Assume top of stack is at level 1 below.

            Move           #NUM1,-(SP)   Push parameters onto stack.
            Move           N,-(SP)
            Call           LISTADD       Call subroutine
                                           (top of stack at level 2).
            Move           4(SP),SUM     Save result.
            Add            #8,SP         Restore top of stack
                                           (top of stack at level 1).
            :

LISTADD     MoveMultiple   R0-R2,-(SP)   Save registers
                                           (top of stack at level 3).
            Move           16(SP),R1     Initialize counter to n.
            Move           20(SP),R2     Initialize pointer to the list.
            Clear          R0            Initialize sum to 0.
LOOP        Add            (R2)+,R0      Add entry from list.
            Decrement      R1
            Branch>0       LOOP
            Move           R0,20(SP)     Put result on the stack.
            MoveMultiple   (SP)+,R0-R2   Restore registers.
            Return                       Return to calling program.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

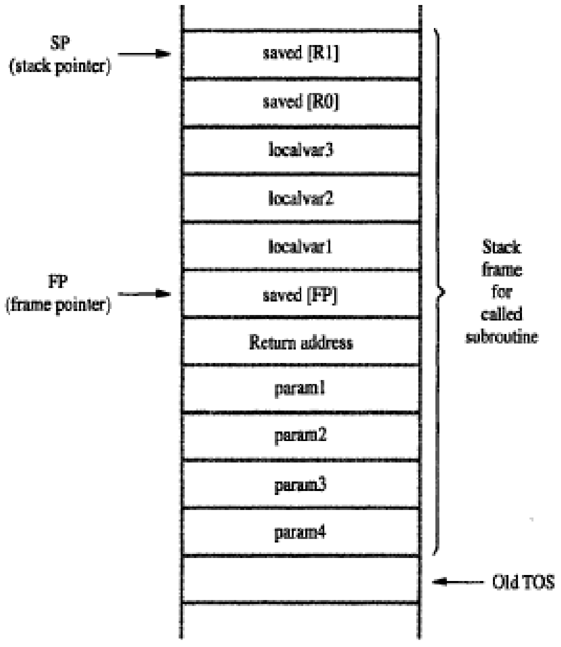


Figure 2.27 A subroutine stack frame example.

• Fig: 2.27 show an example of a commonly used layout for information in a stack-frame.

• *Frame pointer(FP)* is used to access the parameters passed

→ to the subroutine &

→ to the local memory-variables

• The contents of FP remains fixed throughout the execution of the subroutine, unlike stack-pointer SP, which must always point to the current top element in the stack.

Operation on Stack Frame

• Initially SP is pointing to the address of oldTOS.
• The calling-program saves 4 parameters on the stack (Figure 2.27).
• The Call instruction is now executed, pushing the return-address onto the stack.
• Now, SP points to this return-address, and the first instruction of the subroutine is executed.
• Now, FP is to be initialized and its old contents have to be stored. Hence, the first 2 instructions in the subroutine are:

Move FP,-(SP)
Move SP,FP

• The FP is initialized to the value of SP i.e. both FP and SP point to the saved FP address.
• The 3 local variables may now be pushed onto the stack. Space for local variables is allocated by executing the instruction

Subtract #12,SP

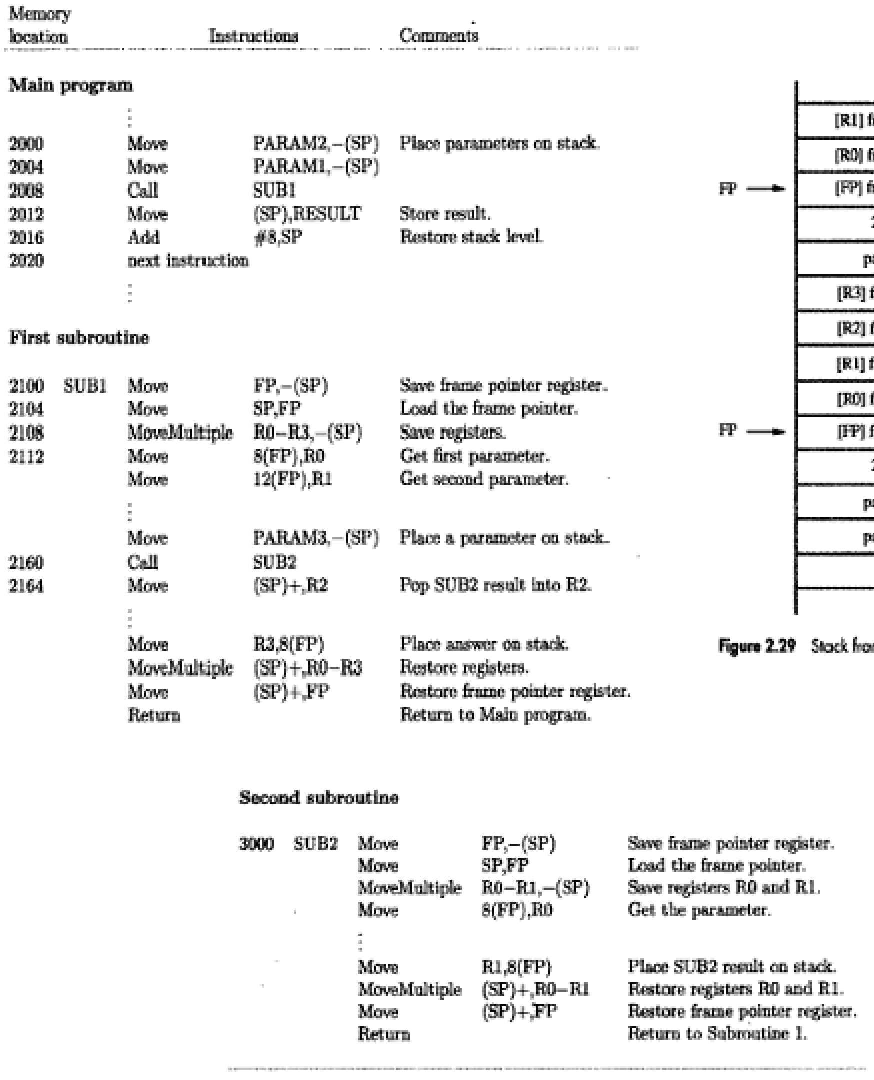
• Finally, the contents of processor-registers R0 and R1 are saved in the stack. At this point, the stack-frame has been set up as shown in the fig 2.27.
• The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

Add #12, SP

• And subroutine pops saved old value of FP back into FP. At this point, SP points to return-address, so the Return instruction can be executed, transferring control back to the calling-program.

STACK FRAMES FOR NESTED SUBROUTINES

• Stack is very useful data structure for holding return-addresses when subroutines are nested.
• When nested subroutines are used; the stack-frames are built up in the processor-stack.
• Consider the following program to illustrate stack frames for nested subroutines



```
Memory
location               Instructions               Comments

Main program
                       :
2000         Move            PARAM2,-(SP)     Place parameters on stack.
2004         Move            PARAM1,-(SP)
2008         Call            SUB1
2012         Move            (SP),RESULT      Store result.
2016         Add             #8,SP            Restore stack level.
2020         next instruction
                       :

First subroutine

2100   SUB1  Move            FP,-(SP)         Save frame pointer register.
2104         Move            SP,FP            Load the frame pointer.
2108         MoveMultiple    R0-R3,-(SP)      Save registers.
2112         Move            8(FP),R0         Get first parameter.
             Move            12(FP),R1        Get second parameter.
                       :
             Move            PARAM3,-(SP)     Place a parameter on stack.
2160         Call            SUB2
2164         Move            (SP)+,R2         Pop SUB2 result into R2.
                       :
             Move            R3,8(FP)         Place answer on stack.
             MoveMultiple    (SP)+,R0-R3      Restore registers.
             Move            (SP)+,FP         Restore frame pointer register.
             Return                           Return to Main program.

Second subroutine

3000   SUB2  Move            FP,-(SP)         Save frame pointer register.
             Move            SP,FP            Load the frame pointer.
             MoveMultiple    R0-R1,-(SP)      Save registers R0 and R1.
             Move            8(FP),R0         Get the parameter.
                       :
             Move            R1,8(FP)         Place SUB2 result on stack.
             MoveMultiple    (SP)+,R0-R1      Restore registers R0 and R1.
             Move            (SP)+,FP         Restore frame pointer register.
             Return                           Return to Subroutine 1.
```

Figure 2.28 Nested subroutines.

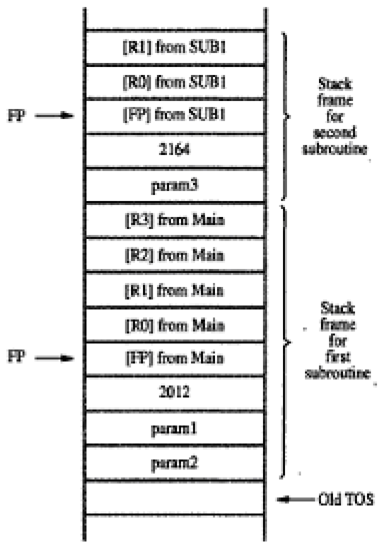


Figure 2.29 Stack frames for Figure 2.28.

The Flow of Execution is as follows:

• Main program pushes the 2 parameters param2 and param1 onto the stack and then calls SUB1.
• SUB1 has to perform an operation & send result to the main-program on the stack (Fig:2.28& 29).
• During the process, SUB1 calls the second subroutine SUB2 (in order to perform some subtask).
• After SUB2 executes its Return instruction; the result is stored in register R2 by SUB1.
• SUB1 then continues its computations & eventually passes required answer back to main-program on the stack.
• When SUB1 executes return statement, the main-program stores this answers in memory-location RESULT and continues its execution.

LOGIC INSTRUCTIONS

• Logic operations such as AND, OR, and NOT applied to individual bits.
• These are the basic building blocks of digital-circuits.
• This is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel.
• For example, the instruction

Not dst

SHIFT AND ROTATE INSTRUCTIONS

• There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.
• The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
• For general operands, we use a logical shift.

For a number, we use an arithmetic shift, which preserves the sign of the number.

LOGICAL SHIFTS

• Two logical shift instructions are needed, one for shifting left(LShiftL) and another for shifting right(LShiftR).
• These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

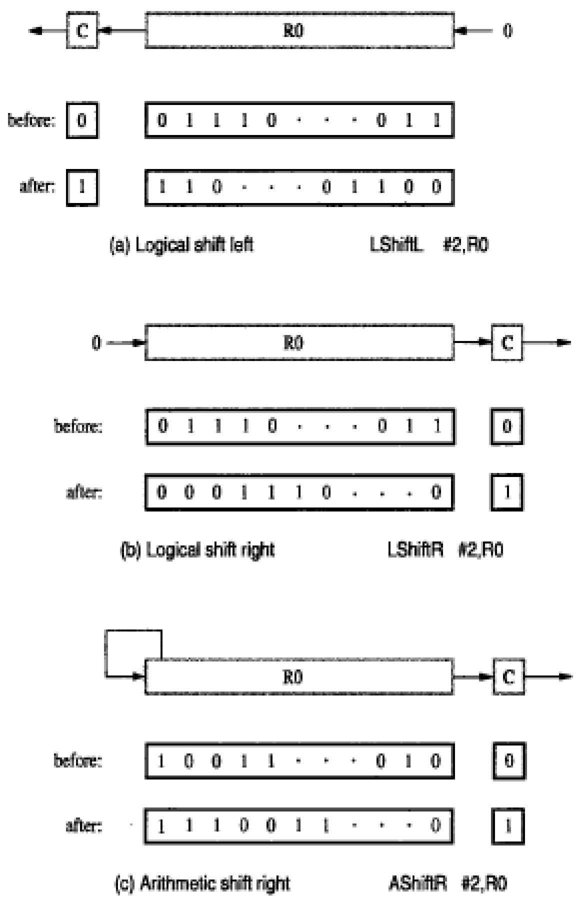


Figure 2.30 Logical and arithmetic shift instructions.

```
Move        #LOC,R0      R0 points to data.
MoveByte    (R0)+,R1     Load first byte into R1.
LShiftL     #4,R1        Shift left by 4 bit positions.
MoveByte    (R0),R2      Load second byte into R2.
And         #$F,R2       Eliminate high-order bits.
Or          R1,R2        Concatenate the BCD digits.
MoveByte    R2,PACKED    Store the result.
```

Figure 2.31 A routine that packs two BCD digits.

ROTATE OPERATIONS

• In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
• To preserve all bits, a set of rotate instructions can be used.
• They move the bits that are shifted out of one end of the operand back into the other end.
• Two versions of both the left and right rotate instructions are usually provided.
In one version, the bits of the operand is simply rotated.
In the other version, the rotation includes the C flag.

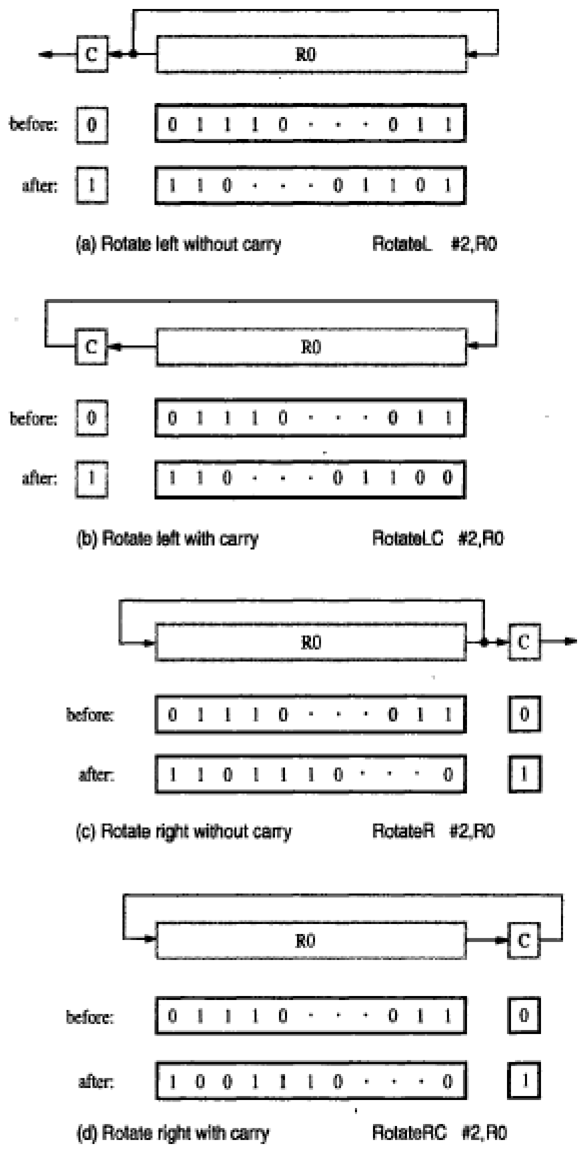


Figure 2.32 Rotate instructions.

ENCODING OF MACHINE INSTRUCTIONS

• To be executed in a processor, an instruction must be encoded in a binary-pattern. Such encoded instructions are referred to as *machine instructions*.
• The instructions that use symbolic names and acronyms are called *assembly language instructions*.
• We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers.
• Let us examine some typical cases.

The instruction

Add R1, R2 ;Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing-mode is used for each operand.

The instruction

Move 24(R0), R5 ;Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

• In all these examples, the instructions can be encoded in a 32-bit word (Fig 2.39).
• The OP code for given instruction refers to type of operation that is to be performed.
• Source and destination field refers to source and destination operand respectively.
• The "Other info" field allows us to specify the additional information that may be needed such as an index value or an immediate operand.
• Using multiple words, we can implement complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computers (CISC) refers to processors that use
• CISC approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used.
• In RISC (reduced instruction set computers), any instruction occupies only one word.
• The RISC approach introduced other restrictions such as that all manipulation of data must be done on operands that are already in processor registers.

Ex: Add R1,R2,R3

• In RISC type machine, the memory references are limited to only Load/Store operations.

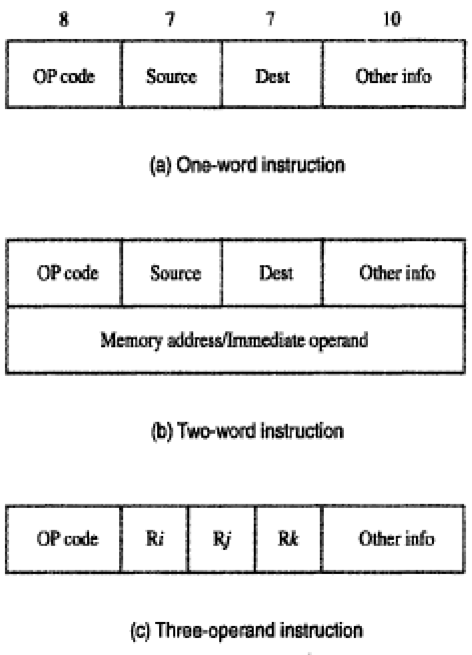


Figure 2.39 Encoding instructions into 32-bit words.

UNIT 3: INPUT/OUTPUT ORGANIZATION

ACCESSING I/O DEVICES

• There are 2 ways to deal with I/O devices (Figure 4.1).

1) Memory mapped I/O
   - • Memory and I/O devices share a common address-space.
   - • Any data-transfer instruction(like Move, Load) can be used to exchange information.
   - • For example, Move DATAIN, R0 ;this instruction reads data from DATAIN(input-buffer associated with keyboard) & stores them into processor-register R0.

2) In *I/O mapped I/O*, memory and I/0 address-spaces are different.
   - • A special instructions named IN and OUT are used for data transfer.
   - • Advantage of separate I/O space: I/O devices deal with fewer address-lines.

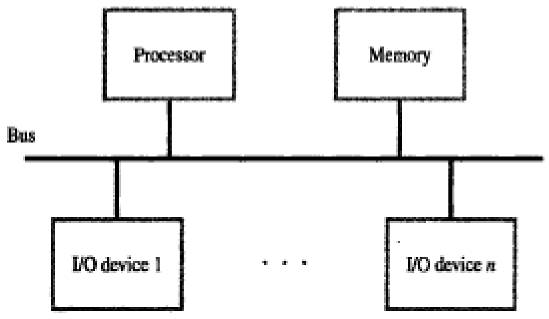


Figure 4.1 A single-bus structure.

I/O Interface for an Input Device

• Address decoder: decodes address sent on bus, so as to enable input-device (Figure 4.2).
• Data register: holds data being transferred to or from the processor.
• Status register: contains information relevant to operation of I/O device.
• Address decoder, data- and status-registers, and control-circuitry required to coordinate I/O transfers constitute device's interface-circuit.

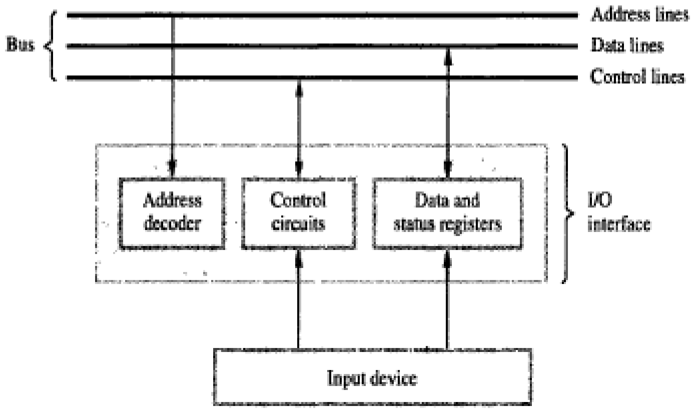


Figure 4.2 I/O interface for an input device.

MECHANISMS USED FOR INTERFACING I/O DEVICES

1) Program Controlled I/O
   - Processor repeatedly checks a status-flag to achieve required synchronization between processor & input/output device. (We say that the processor polls the device).
   - Main drawback: The processor wastes its time in checking the status of the device before actual data transfer takes place.

2) Interrupt I/O
   - Synchronization is achieved by having I/O device send a special signal over bus whenever it is ready for a data transfer operation.

3) Direct Memory Access (DMA)
   - This involves having the device-interface transfer data directly to or from the memory without continuous involvement by the processor.

INTERRUPTS

• I/O device initiates the action instead of the processor. This is done by sending a special hardware signal to the processor called as *interrupt*(INTR), on the interrupt-request line.
• The processor can be performing its own task without the need to continuously check the I/O device.
• When device gets ready, it will "alert" the processor by sending an interrupt-signal (Figure 4.5).
• The routine executed in response to an interrupt-request is called ISR(Interrupt Service Routine).
• Once the interrupt-request signal comes from the device, the processor has to inform the device that its request has been recognized and will be serviced soon. This is indicated by a special control signal on the bus called *interrupt-acknowledge*(INTA).

Difference between subroutine & ISR

• A subroutine performs a function required by the program from which it is called.
However, the ISR may not have anything in common with the program being executed at the time the interrupt-request is received. Before starting execution of ISR, any information that may be altered during the execution of that routine must be saved. This information must be restored before the interrupted-program resumed.
• Another difference is that an interrupt is a mechanism for coordinating I/O transfers
whereas a subroutine is just a linkage of 2 or more function related to each other.

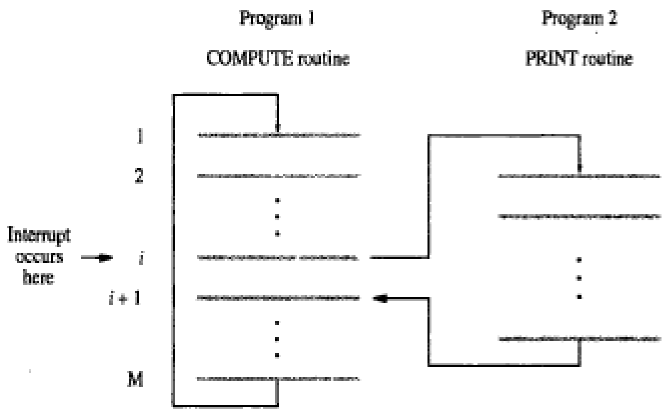


Figure 4.5 Transfer of control through the use of interrupts.

Note:

• The speed of operation of the processor and I/O devices differ greatly. Also, since I/O devices are manually operated in many cases (like pressing a key on keyboard), there may not be synchronization between the CPU operations and I/O operations with reference to CPU clock. To cater to the different needs of I/O operations, 3 mechanisms have been developed for interfacing I/O devices. 1) Program controlled I/O 2) Interrupt I/O 3) Direct memory access (DMA).
• Saving registers increases the delay between the time an interrupt request is received and the start of execution of the ISR. This delay is called interrupt latency.
• Since interrupts can arrive at any time, they may alter the sequence of events. Hence, facility must be provided to enable and disable interrupts as desired.
• Consider the case of a single interrupt request from one device. The device keeps the interrupt request signal activated until it is informed that the processor has accepted its request. This activated signal, if not deactivated may lead to successive interruptions, causing the system to enter into an infinite loop.

INTERRUPT HARDWARE

• An I/O device requests an interrupt by activating a bus-line called interrupt-request(IR).
• A single IR line can be used to serve 'n' devices (Figure 4.6).
• All devices are connected to IR line via switches to ground.
• To request an interrupt, a device closes its associated switch. Thus, if all IR signals are inactive(i.e. if all switches are open), the voltage on the IR line will be equal to V_{dd}.
• When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the INTR received by the processor to goto 1.
• The value of INTR is the logical OR of the requests from individual devices

$$INTR=INTR_1+ INTR_{2+} \ldots\ldots +INTR_n$$

• A special gates known as open-collector or open-drain are used to drive the INTR line.
• Resistor R is called a *pull-up resistor* because
  it pulls the line voltage up to the high-voltage state when the switches are open.

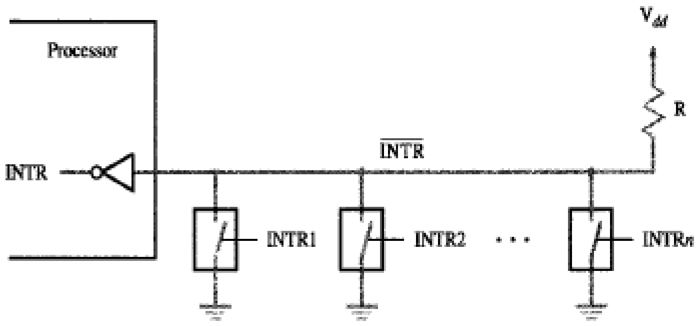


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

ENABLING & DISABLING INTERRUPTS

• To prevent the system from entering into an infinite-loop because of interrupt, there are 3 possibilities:
  1) The first possibility is to have the processor-hardware ignore the interrupt-request line until the execution of the first instruction of the ISR has been completed.
  2) The second option is to have the processor automatically disable interrupts before starting the execution of the ISR.
  3) In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be edge-triggered.
• Sequence of events involved in handling an interrupt-request from a single device is as follows:
  1) The device raises an interrupt-request.
  2) The program currently being executed is interrupted.
  3) All interrupts are disabled(by changing the control bits in the PS).
  4) The device is informed that its request has been recognized, and
     in response, the device deactivates the interrupt-request signal.
  5) The action requested by the interrupt is performed by the ISR.
  6) Interrupts are enabled again and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

Polling

• Information needed to determine whether a device is requesting an interrupt is available in its status-register.
• When a device raises an interrupt-request, it sets IRQ bit to 1 in its status-register (Figure 4.3).
• KIRQ and DIRQ are the interrupt-request bits for keyboard & display.
• Simplest way to identify interrupting device is to have ISR poll all I/O devices connected to bus.
• The first device encountered with its IRQ bit set is the device that should be serviced. After servicing this device, next requests may be serviced.
• Main advantage: Simple & easy to implement.
Main disadvantage: More time spent polling IRQ bits of all devices (that may not be requesting any service).

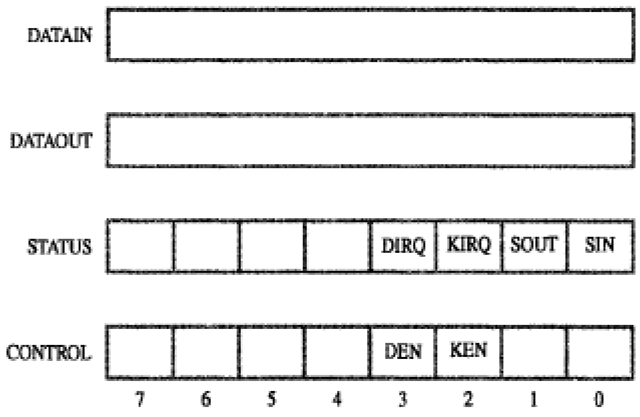


Figure 4.3 Registers in keyboard and display interfaces.

```
        Move        #LINE,R0        Initialize memory pointer.
WAITK   TestBit     #0,STATUS       Test SIN.
        Branch=0    WAITK           Wait for character to be entered.
        Move        DATAIN,R1       Read character.
WAITD   TestBit     #1,STATUS       Test SOUT.
        Branch=0    WAITD           Wait for display to become ready.
        Move        R1,DATAOUT      Send character to display.
        Move        R1,(R0)+        Store charater and advance pointer.
        Compare     #$0D,R1         Check if Carriage Return.
        Branch≠0    WAITK           If not, get another character.
        Move        #$0A,DATAOUT    Otherwise, send Line Feed.
        Call        PROCESS         Call a subroutine to process the
                                      the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

Vectored Interrupts

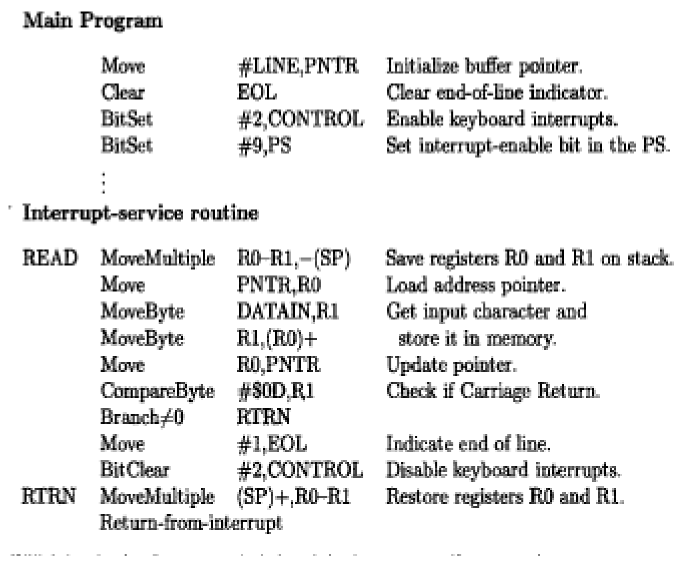
• A device requesting an interrupt identifies itself by sending a special-code to processor over bus. (This enables processor to identify individual devices even if they share a single interrupt-request line).
• The code represents starting-address of ISR for that device.
• ISR for a given device must always start at same location.
• The address stored at the location pointed to by interrupting-device is called the interrupt-vector.
• Processor
⟶ loads interrupt-vector into PC &
⟶ executes appropriate ISR
• Interrupting-device must wait to put data on bus only when processor is ready to receive it.
• When processor is ready to receive interrupt-vector code, it activates INTA line.
• I/O device responds by sending its interrupt-vector code & turning off the INTR signal.

CONTROLLING DEVICE REQUESTS

• There are 2 independent mechanisms for controlling interrupt requests.

• At device-end, an interrupt-enable bit in a control register determines whether device is allowed to generate an interrupt request.

• At processor-end, either an interrupt-enable bit in the PS register or a priority structure determines whether a given interrupt-request will be accepted.



```
Main Program

        Move           #LINE,PNTR     Initialize buffer pointer.
        Clear          EOL            Clear end-of-line indicator.
        BitSet         #2,CONTROL     Enable keyboard interrupts.
        BitSet         #9,PS          Set interrupt-enable bit in the PS.
          ⋮
Interrupt-service routine

READ    MoveMultiple   R0–R1,–(SP)    Save registers R0 and R1 on stack.
        Move           PNTR,R0        Load address pointer.
        MoveByte       DATAIN,R1      Get input character and
        MoveByte       R1,(R0)+         store it in memory.
        Move           R0,PNTR        Update pointer.
        CompareByte    #$0D,R1        Check if Carriage Return.
        Branch≠0       RTRN
        Move           #1,EOL         Indicate end of line.
        BitClear       #2,CONTROL     Disable keyboard interrupts.
RTRN    MoveMultiple   (SP)+,R0–R1    Restore registers R0 and R1.
        Return-from-interrupt
```

Figure 4.9 Using interrupts to read a line of characters from a keyboard via the registers in Figure 4.3.

INTERRUPT NESTING

• A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device
• Each of the INTR lines is assigned a different priority-level (Figure 4.7).
• Priority-level of processor is the priority of program that is currently being executed.
• During execution of an ISR, interrupt-requests will be accepted from some devices but not from others depending upon device's priority.
• Processor accepts interrupts only from devices that have priority higher than its own.
• At the time of execution of an ISR for some device is started, priority of processor is raised to that of the device
• Processor's priority is encoded in a few bits of processor-status(PS) word. This can be changed by program instructions that write into PS. These are called *privileged instructions.*
• Privileged-instructions can be executed only while processor is running in supervisor-mode.
• Processor is in supervisor-mode only when executing operating-system routines. (An attempt to execute a privileged-instruction while in the user-mode leads to a special type of interrupt called a *privileged exception*).

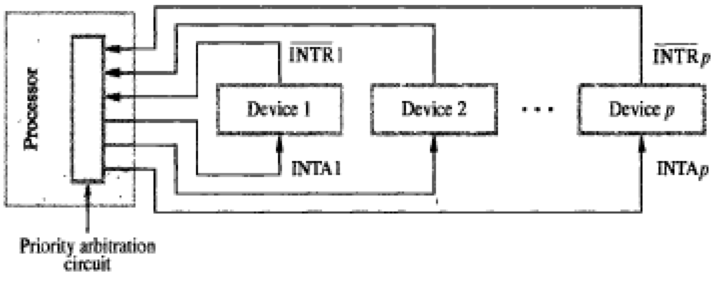

Processor
$\overline{\text{INTR}}$ 1
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA 1
INTA p
Priority arbitration circuit

Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

SIMULTANEOUS REQUESTS

• INTR line is common to all devices (Figure 4.8).
• INTA line is connected in a daisy-chain fashion such that INTA signal propagates serially through devices.
• When several devices raise an interrupt-request and INTR line is activated, processor responds by setting INTA line to 1. This signal is received by device 1.
• Device 1 passes signal on to device 2 only if it does not require any service.
• If device 1 has a pending-request for interrupt, it blocks INTA signal and proceeds to put its identifying code on data lines.
• Device that is electrically closest to processor has highest priority.
• Main advantage: This allows the processor to accept interrupt-requests from some devices but not from others depending upon their priorities.

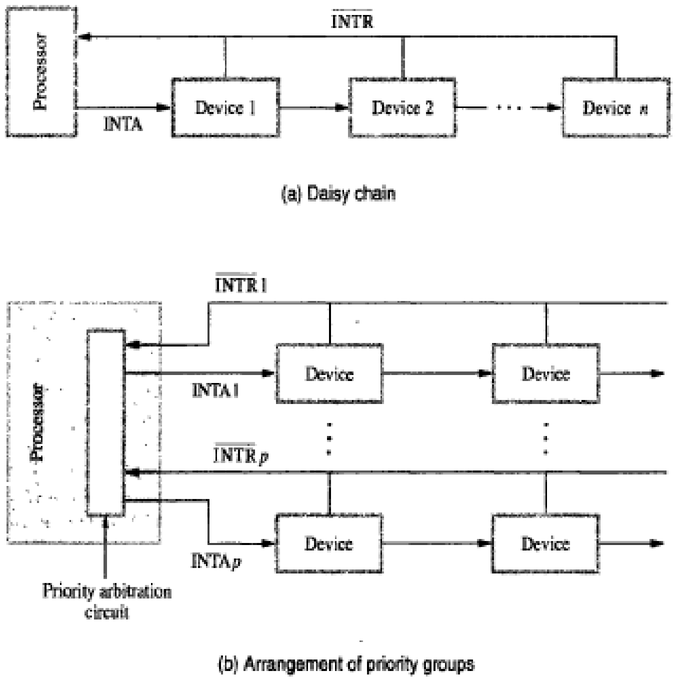


Figure 4.8 Interrupt priority schemes.

EXCEPTIONS

• An interrupt is an event that causes

→ execution of one program to be suspended &

→ execution of another program to begin.

• *Exception* refers to any event that causes an interruption.

I/O interrupts are one example of an exception.

Recovery from Errors

• Computers use a variety of techniques to ensure that all hardware-components are operating properly. For e.g. many computers include an error-checking code in main-memory which allows detection of errors in stored-data.

• If an error occurs, control-hardware detects it & informs processor by raising an interrupt.

• When exception processing is initiated (as a result of errors), processor

→ suspends program being executed &

→ starts an ESR(Exception Service Routine). This routine takes appropriate action to recover from the error to inform user about it.

Debugging

• Debugger

→ helps programmer find errors in a program and

→ uses exceptions to provide 2 important facilities: 1) Trace & 2) Breakpoints

• When a processor is operating in trace-mode, an exception occurs after execution of every instruction (using debugging-program as ESR).

• Debugging-program enables user to examine contents of registers (AX, BX), memory-locations and so on.

• On return from debugging-program,

next instruction in program being debugged is executed,

then debugging-program is activated again.

• Breakpoints provide a similar facility except that program being debugged is interrupted only at specific points selected by user. An instruction called Trap(or Software interrupt) is usually provided for this purpose.

Privilege Exception

• To protect OS of computer from being corrupted by user-programs, certain instructions can be executed only while processor is in supervisor-mode. These are called *privileged instructions.*

• For e.g. when the processor is running in user-mode, it will not execute an instruction that changes priority-level of processor.

• An attempt to execute such an instruction will produce a privilege-exception. As a result, processor switches to supervisor-mode & begins to execute an appropriate routine in OS.

DIRECT MEMORY ACCESS (DMA)

• The transfer of a block of data directly between an external device & main memory without continuous involvement by processor is called as *DMA*.
• DMA transfers are performed by a control-circuit that is part of I/O device interface. This circuit is called as a *DMA controller* (Figure 4.19).
• DMA controller performs the functions that would normally be carried out by processor
• In controller, 3 registers are accessed by processor to initiate transfer operations (Figure 4.18):

1) Two registers are used for storing starting-address & word-count
2) Third register contains status- & control-flags

• The R/W bit determines direction of transfer.

When R/W=1, controller performs a read operation(i.e. it transfers data from memory to I/O),
Otherwise it performs a write operation (i.e. it transfers data from I/O device to memory).

• When Done=1, controller

→ completes transferring a block of data &
→ is ready to receive another command.

• When IE=1, controller raises an interrupt after it has completed transferring a block of data (IE=Interrupt Enable).
• Finally, when IRQ=1, controller requests an interrupt. (Requests by DMA devices for using the bus are always given higher priority than processor requests).
• There are 2 ways in which the DMA operation can be carried out:

2) In one method, processor originates most memory-access cycles. DMA controller is said to "steal" memory cycles from processor. Hence, this technique is usually called *cycle stealing.*
3) In second method, DMA controller is given exclusive access to main-memory to transfer a block of data without any interruption. This is known as *block mode* (or burst mode).

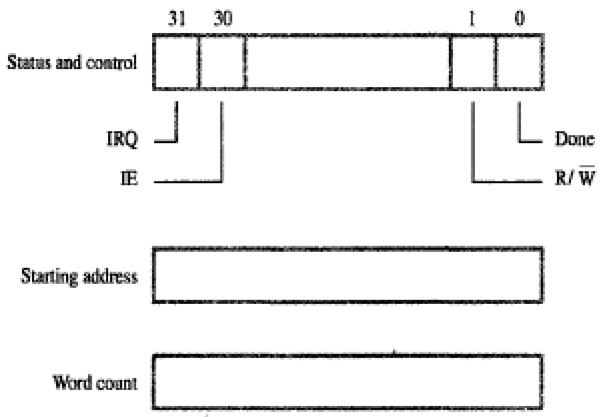


Figure 4.18 Registers in a DMA interface.

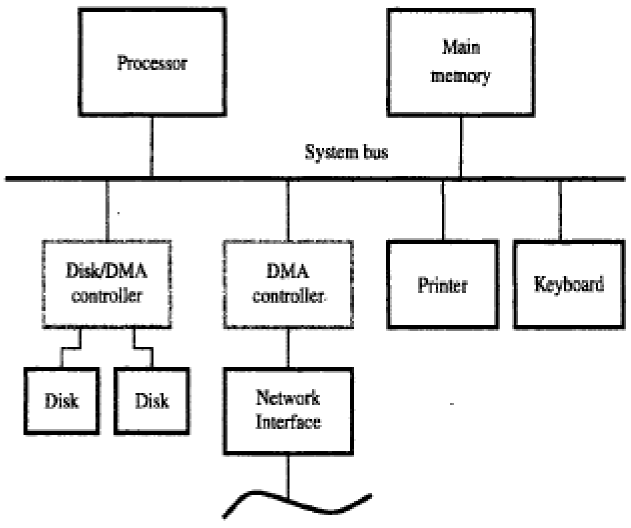


Figure 4.19 Use of DMA controllers in a computer system.

BUS ARBITRATION

• The device that is allowed to initiate data transfers on bus at any given time is called *bus-master.*
• There can be only one bus master at any given time.
• *Bus arbitration* is the process by which next device to become the bus-master is selected and bus-mastership is transferred to it.
• There are 2 approaches to bus arbitration:
  1) In centralized arbitration, a single bus-arbiter performs the required arbitration.
  2) In distributed arbitration, all device participate in selection of next bus-master.

CENTRALIZED ARBITRATION

• A single bus-arbiter performs the required arbitration (Figure: 4.20 & 4.21).
• Normally, processor is the bus.master unless it grants bus.mastership to one of the DMA controllers.
• A DMA controller indicates that it needs to become bus.master by activating Bus-Request line(BR).
• The signal on the BR line is the logical OR of bus-requests from all devices connected to it.
• When BR is activated, processor activates Bus-Grant signal(BG1) indicating to DMA controllers that they may use bus when it becomes free. (This signal is connected to all DMA controllers using a daisy-chain arrangement).
• If DMA controller-1 is requesting the bus, it blocks propagation of grant-signal to other devices.
  Otherwise, it passes the grant downstream by asserting BG2.
• Current bus-master indicates to all devices that it is using bus by activating Bus-Busy line (BBSY).
• Arbiter circuit ensures that only one request is granted at any given time according to a predefined priority scheme

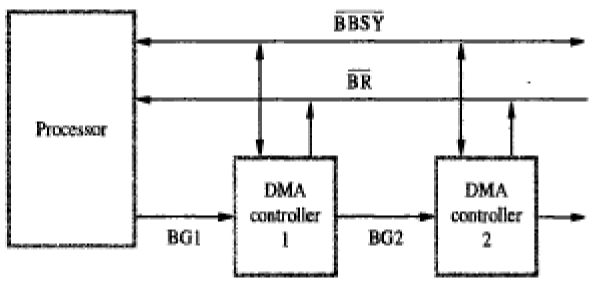


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

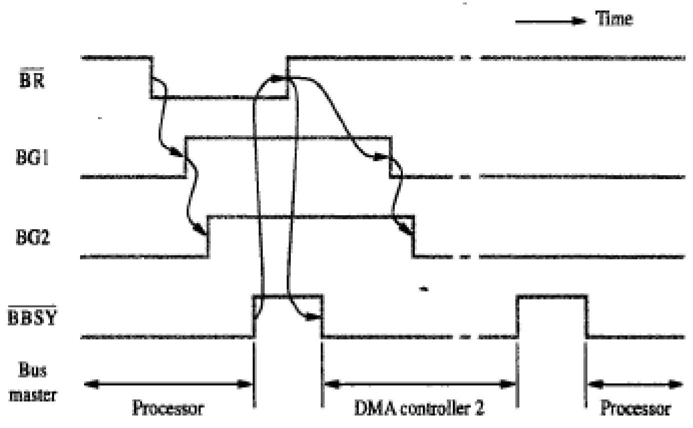


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

Note:
A conflict may arise if both the processor and a DMA controller try to use the bus at the same time to access the main memory. To resolve these conflicts, a special circuit called the bus arbiter is provided to coordinate the activities of all devices requesting memory transfers

DISTRIBUTED ARBITRATION

• All device participate in the selection of next bus-master (Figure 4.22)
• Each device on bus is assigned a 4-bit identification number (ID).
• When 1 or more devices request bus, they

→ assert Start-Arbitration signal &

→ place their 4-bit ID numbers on four open-collector lines $\overline{ARB0}$ through $\overline{ARB3}$.

• A winner is selected as a result of interaction among signals transmitted over these lines by all contenders.
• Net outcome is that the code on 4 lines represents request that has the highest ID number.
• Main advantage: This approach offers higher reliability since operation of bus is not dependent on any single device.

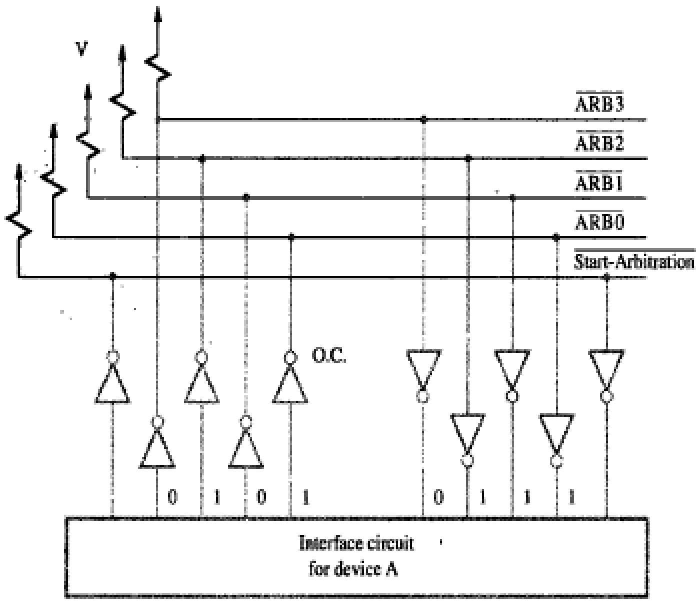


Figure 4.22 A distributed arbitration scheme.

BUSES

• Bus
→ is used to inter-connect main-memory, processor & I/O devices
→ includes lines needed to support interrupts & arbitration

• Primary function: To provide a communication-path for transfer of data.

• *Bus protocol* is set of rules that govern the behavior of various devices connected to the buses.

• Bus-protocol specifies parameters such as:
→ asserting control-signals
→ timing of placing information on bus
→ rate of data-transfer

• A typical bus consists of 3 sets of lines: 1) Address, 2) Data and 3) Control lines.

• Control-signals specify whether a read or a write operation is to be performed.

• R/W line specifies
→ read operation when R/W=1
→ write operation when R/W=0

• In data-transfer operation, one device plays the role of a bus-master which initiates data transfers by issuing Read or Write commands on bus (Hence it may be called an initiator).

• Device addressed by master is referred to as a slave (or target).

• Timing of data transfers over a bus is classified into 2 types:
1) Synchronous and 2) Asynchronous

SYNCHRONOUS BUS

• All devices derive timing-information from a common clock-line.

• Equally spaced pulses on this line define equal time intervals.

• Each of these intervals constitutes a bus-cycle during which one data transfer can take place.

A sequence of events during a read operation:

• At time t_0, the master (processor)
→ places the device-address on address-lines &
→ sends an appropriate command on control-lines (Figure 4.23).

• Information travels over bus at a speed determined by its physical & electrical characteristics.

• Clock pulse width(t_1-t_0) must be longer than the maximum propagation-delay between 2 devices connected to bus.

• Information on bus is unreliable during the period t_0 to t_1 because signals are changing state.

• Slave places requested input-data on data-lines at time t_1.

• At end of clock cycle(at time t_2), master strobes(captures) data on data-lines into its input-buffer

• For data to be loaded correctly into any storage device (such as a register built with flip-flops), data must be available at input of that device for a period greater than setup-time of device.

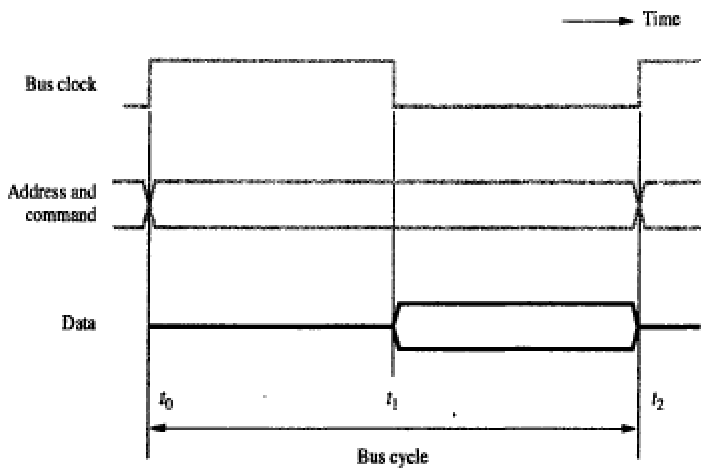


Figure 4.23 Timing of an input transfer on a synchronous bus.

ASYNCHRONOUS BUS

• This method uses handshake-signals between master and slave for coordinating data transfers.

• There are 2 control-lines:

1) Master-ready(MR) to indicate that master is ready for a transaction
2) Slave-ready(SR) to indicate that slave is ready to respond

The read operation proceeds as follows:

• At t0, master places address- & command-information on bus. All devices on bus begin to decode this information.

• At t1, master sets MR-signal to 1 to inform all devices that the address- & command-information is ready.

• At t2, selected slave performs required input-operation & sets SR signal to 1 (Figure 4.26).

• At t3, SR signal arrives at master indicating that the input-data are available on bus skew.

• At t4, master removes address- & command-information from bus.

• At t5, when the device-interface receives the 1-to-0 transition of MR signal, it removes data and SR signal from the bus. This completes the input transfer

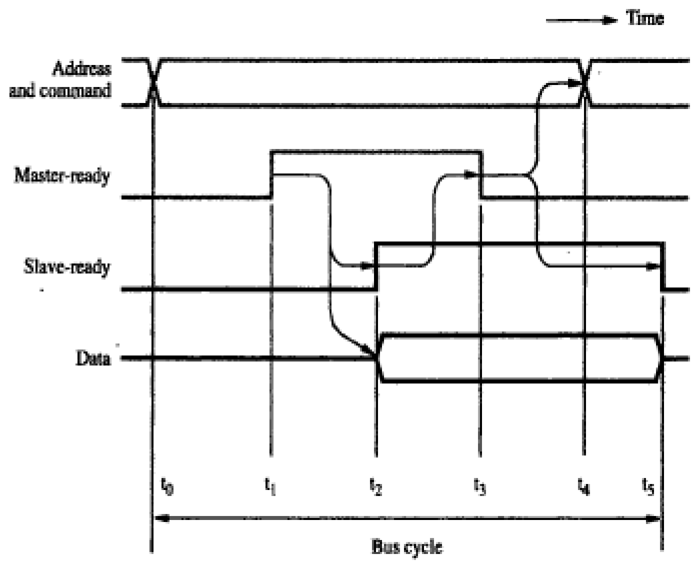


Figure 4.26 Handshake control of data transfer during an input operation.

UNIT 6: ARITHMETIC

ADDITION & SUBTRACTION OF SIGNED NUMBERS

n-BIT RIPPLE CARRY ADDER

• A cascaded connection of n full-adder blocks can be used to add 2-bit numbers. Since carries must propagate(or ripple) through cascade, the configuration is called an n-bit ripple carry adder.(Fig 6.2).

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

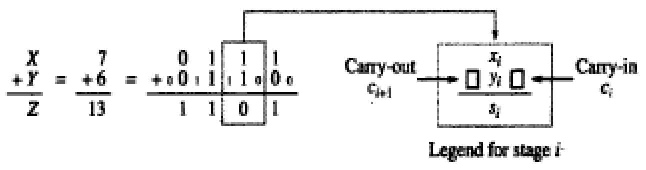
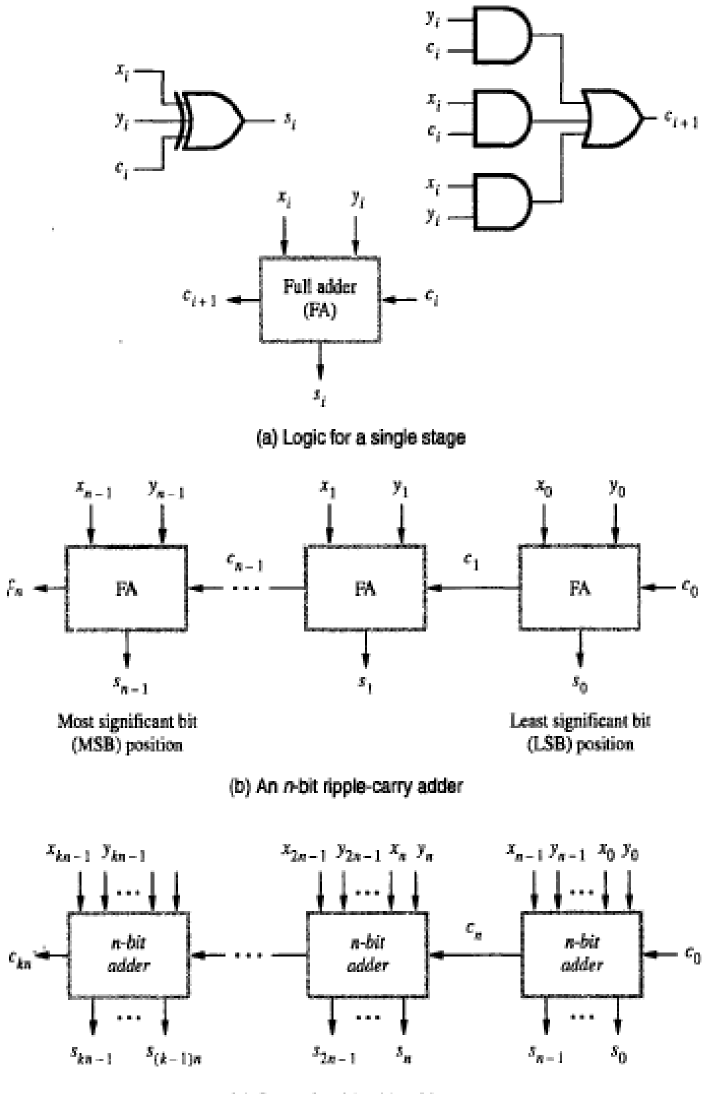


Figure 6.1 Logic specification for a stage of binary addition.


Full adder (FA)
FA
Most significant bit (MSB) position
Least significant bit (LSB) position
n-bit adder
(a) Logic for a single stage

(b) An n-bit ripple-carry adder

(c) Cascade of k n-bit adders

Figure 6.2 Logic for addition of binary vectors.

ADDITION/SUBTRACTION LOGIC UNIT

• The n-bit adder can be used to add 2's complement numbers X and Y (Figure 6.3).
• Overflow can only occur when the signs of the 2 operands are the same.
• In order to perform the subtraction operation X-Y on 2's complement numbers X and Y; we form the 2's complement of Y and add it to X.
• Addition or subtraction operation is done based on value applied to the Add/Sub input control-line.
• Control-line=0 for addition, applying the Y vector unchanged to one of the adder inputs.
Control-line=1 for subtraction, the Y vector is 2's complemented.

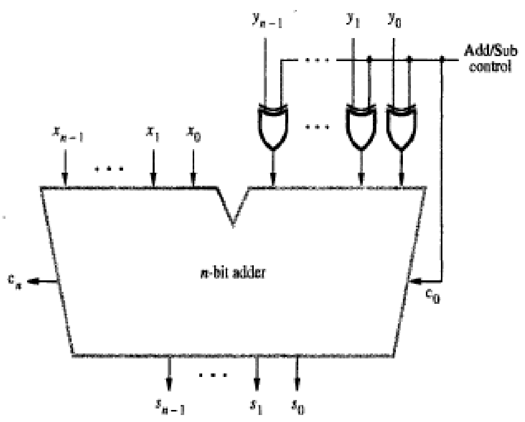


Figure 6.3 Binary addition-subtraction logic network.

DESIGN OF FAST ADDERS

• Drawback of ripple carry adder: If the adder is used to implement the addition/subtraction, all sum bits are available in 2n gate delays.

• Two approaches can be used to reduce delay in adders:

i) Use the fastest possible electronic-technology in implementing the ripple-carry design

ii) Use an augmented logic-gate network structure

CARRY-LOOKAHEAD ADDITIONS

• The logic expression for s_i(sum) and c_{i+1}(carry-out) of stage i are

$s_i=x_i+y_i+c_i$ ------(1) $c_{i+1}=x_iy_i+x_ic_i+y_ic_i$ ------(2)

• Factoring (2) into

$c_{i+1}=x_iy_i+(x_i+y_i)c_i$

we can write

$c_{i+1}=G_i+P_iC_i$ where $G_i=x_iy_i$ and $P_i=x_i+y_i$

• The expressions G_i and P_i are called generate and propagate functions (Figure 6.4).

• If G_i=1, then c_{i+1}=1, independent of the input carry c_i. This occurs when both x_i and y_i are 1. Propagate function means that an input-carry will produce an output-carry when either x_i=1 or y_i=1.

• All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay.

• Expanding c_i terms of i-1 subscripted variables and substituting into the c_{i+1} expression, we obtain

$c_{i+1}=G_i+P_iG_{i-1}+P_iP_{i-1}G_{i-2}. \ . \ . \ . \ . \ .+P_1G_0+P_iP_{i-1} \ . \ . \ . \ P_0c_0$

• Conclusion: Delay through the adder is 3 gate delays for all carry-bits &
4 gate delays for all sum-bits.

• Consider the design of a 4-bit adder. The carries can be implemented as

$c_1=G_0+P_0c_0$

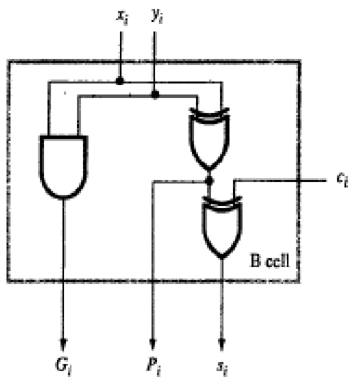
$c_2=G_1+P_1G_0+P_1P_0c_0$

$c_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0c_0$

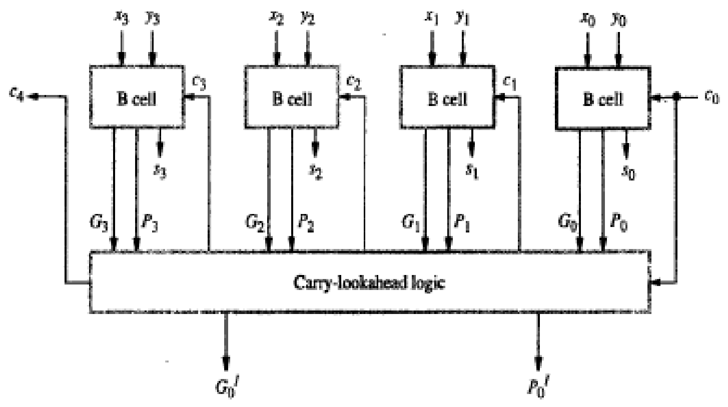
$c_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$

• The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a *carry-lookahead adder*.

• Limitation: If we try to extend the carry-lookahead adder for longer operands, we run into a problem of gate fan-in constraints.



(a) Bit-stage cell



(b) 4-bit adder

Figure 6.4 4-bit carry-lookahead adder.

HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

• 16-bit adder can be built from four 4-bit adder blocks (Figure 6.5).
• These blocks provide new output functions defined as G_k and P_k,
  where k=0 for the first 4-bit block,
  k=1 for the second 4-bit block and so on.
• In the first block,

$P_0=P_3P_2P_1P_0$

&

$G_0=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$

• The first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry, and the second level G_k and P_k functions determine whether block k generates or propagates a carry.
• Carry c_{16} is formed by one of the carry-lookahead circuits as

$c_{16}=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$

• Conclusion: All carries are available 5 gate delays after X, Y and c_0 are applied as inputs.

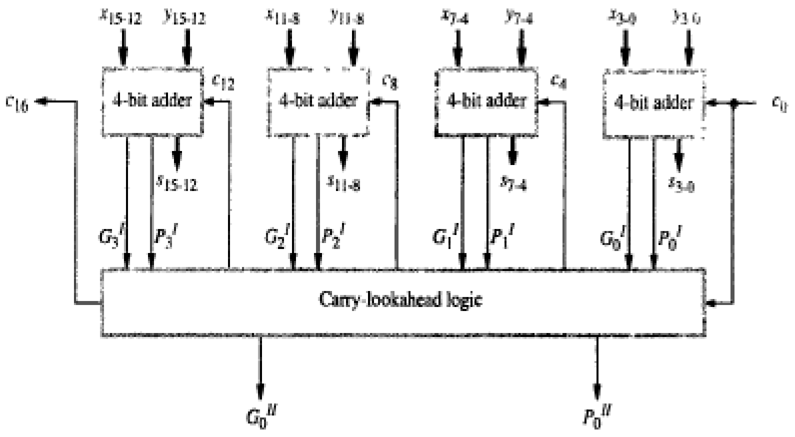
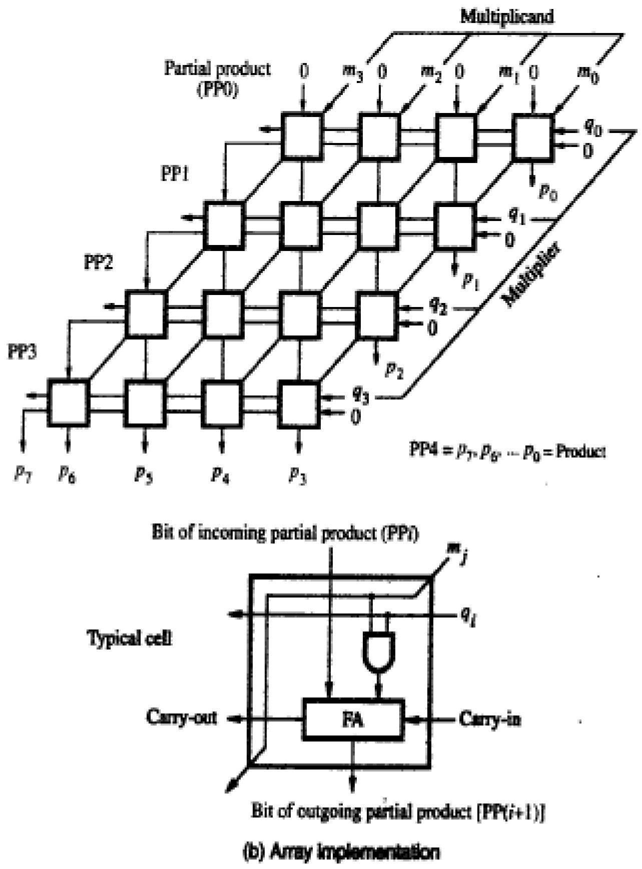


Figure 6.5 16-bit carry-lookahead adder built from 4-bit adders (see Figure 6.4b).

MULTIPLICATION OF POSITIVE NUMBERS

```
          1 1 0 1       (13) Multiplicand M
        × 1 0 1 1       (11) Multiplier Q
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
  ---------------
  1 0 0 0 1 1 1 1       (143) Product P
```

(a) Manual multiplication algorithm



(b) Array implementation

Figure 6.6 Array multiplication of positive binary operands.

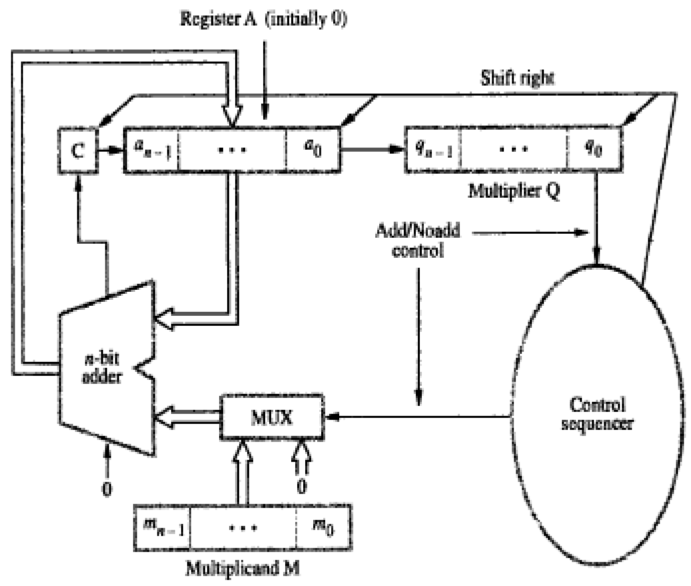
ARRAY MULTIPLICATION

• The main component in each cell is a full adder(FA)..

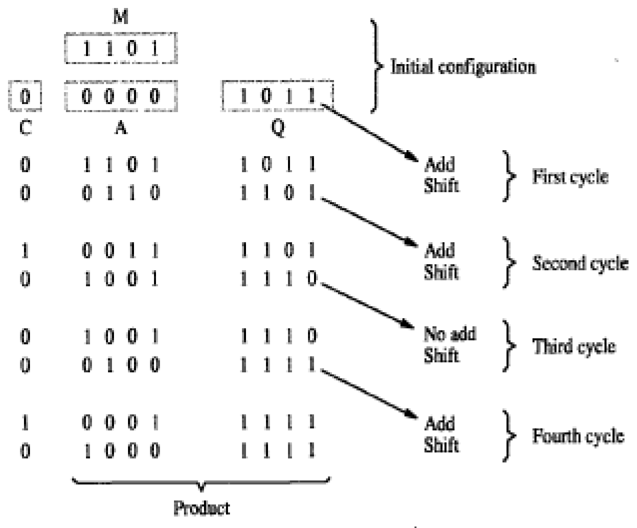
• The AND gate in each cell determines whether a multiplicand bit m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit q_i (Figure 6.6).

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold PP_i(partial product)
  while the multiplier bit q_i generates the signal Add/Noadd.
- The carry-out from the adder is stored in flip-flop C (Figure 6.7).
- Procedure for multiplication:
  1) Multiplier is loaded into register Q,
     Multiplicand is loaded into register M and
     C & A are cleared to 0.
  2) If q_0=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position.
     If q_0=0, no addition performed and C, A & Q are shifted right one bit-position.
  3) After n cycles, the high-order half of the product is held in register A and
     the low-order half is held in register Q.



(a) Register configuration



C	A	Q		
	M: 1 1 0 1			Initial configuration
0	0 0 0 0	1 0 1 1		
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

(b) Multiplication example

Figure 6.7 Sequential circuit binary multiplier.

SIGNED OPERAND MULTIPLICATION

BOOTH ALGORITHM

• This algorithm

→ generates a 2n-bit product

→ treats both positive & negative 2's-complement n-bit operands uniformly(Figure 6.9-6.12).

• Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.

• This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

For e.g. multiplier(Q) 14(001110) can be represented as

```
 010000 (16)
-000010 (2)
 001110 (14)
```

• Therefore, product P=M*Q can be computed by adding 2^4 times the M to the 2's complement of 2^1 times the M

```
                    0  1  0  1  1  0  1
                    0  0 +1 +1 +1 +1  0
                    --------------------
                    0  0  0  0  0  0  0
                 0  1  0  1  1  0  1
              0  1  0  1  1  0  1
           0  1  0  1  1  0  1
        0  1  0  1  1  0  1
     0  0  0  0  0  0  0
  0  0  0  0  0  0  0
-----------------------------------------
  0  0  0  1  0  1  0  1  0  0  0  1  1  0

                    0  1  0  1  1  0  1
                    0 +1  0  0  0 -1  0
                    --------------------
  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  1  1  1  1  1  1  1  0  1  0  0  1  1      <----- 2's complement of
  0  0  0  0  0  0  0  0  0  0  0  0                the multiplicand
  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  1  0  1  1  0  1
  0  0  0  0  0  0  0  0
-----------------------------------------
  0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Figure 6.9 Normal and Booth multiplication schemes.

```
0   0   1   0   1   1   0   0   1   1   1   0   1   0   1   1   0   0
                              ⇓
0  +1  -1  +1   0  -1   0  +1   0   0  -1  +1  -1  +1   0  -1   0   0
```

Figure 6.10 Booth recoding of a multiplier.

```
  0 1 1 0 1  (+13)            0 1 1 0 1
x 1 1 0 1 0  (-6)    ==>      0 -1 +1 -1 0
-----------              ---------------------
                          0 0 0 0 0 0 0 0 0 0
                          1 1 1 1 1 0 0 1 1
                          0 0 0 0 1 1 0 1
                          1 1 1 0 0 1 1
                          0 0 0 0 0 0
                         ---------------------
                          1 1 1 0 1 1 0 0 1 0  (-78)
```

Figure 6.11 Booth multiplication with a negative multiplier.

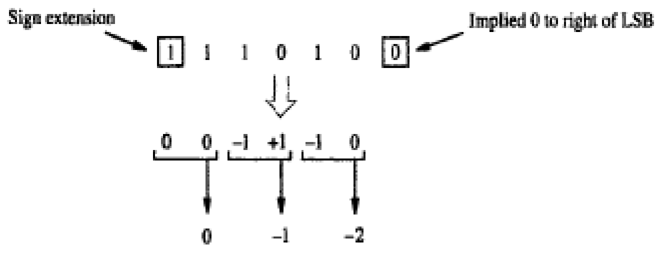
Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i* − 1	
0	0	0×M
0	1	+1×M
1	0	−1×M
1	1	0×M

Figure 6.12 Booth multiplier recoding table.

FAST MULTIPLICATION

BIT-PAIR RECODING OF MULTIPLIERS

- This method
  - → derived from the booth algorithm
  - → reduces the number of summands by a factor of 2
- Group the Booth-recoded multiplier bits in pairs. (Figure 6.14 & 6.15).
- The pair (+1 -1) is equivalent to the pair (0 +1).



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	−2 × M
1	0	1	−1 × M
1	1	0	−1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

Figure 6.14 Multiplier bit-pair recoding.

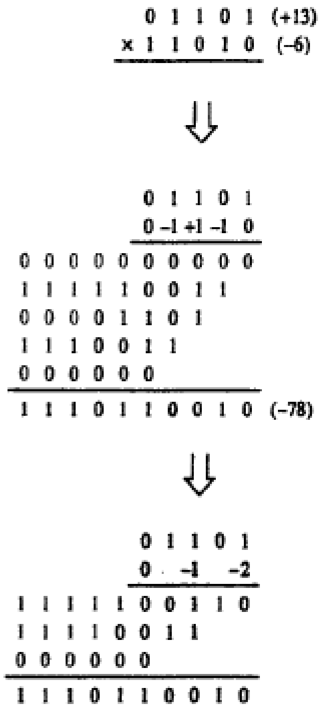


Figure 6.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

• Consider the array for 4*4 multiplication. (Figure 6.16 & 6.18).

• Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.

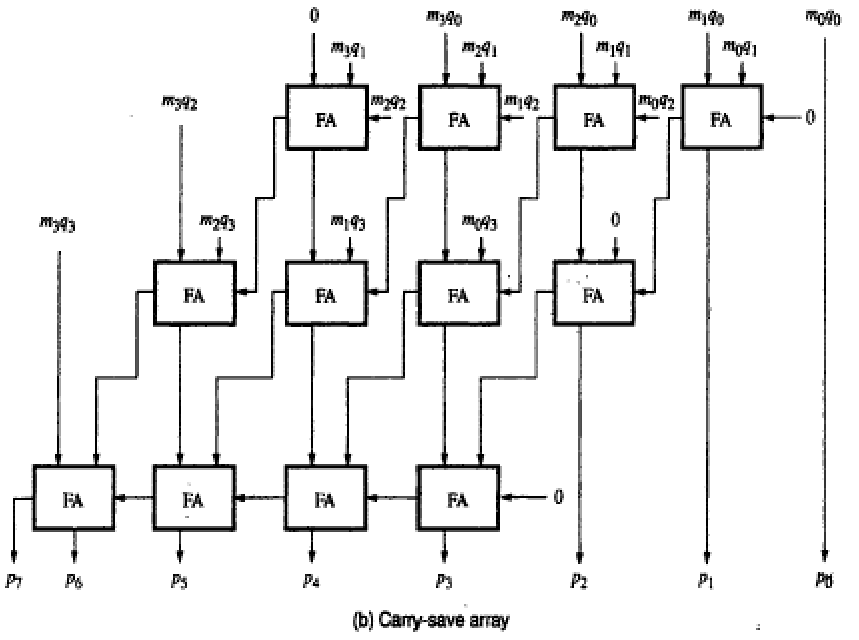


Figure 6.16 Ripple-carry and carry-save arrays for the multiplication operation M × Q = P for 4-bit operands.

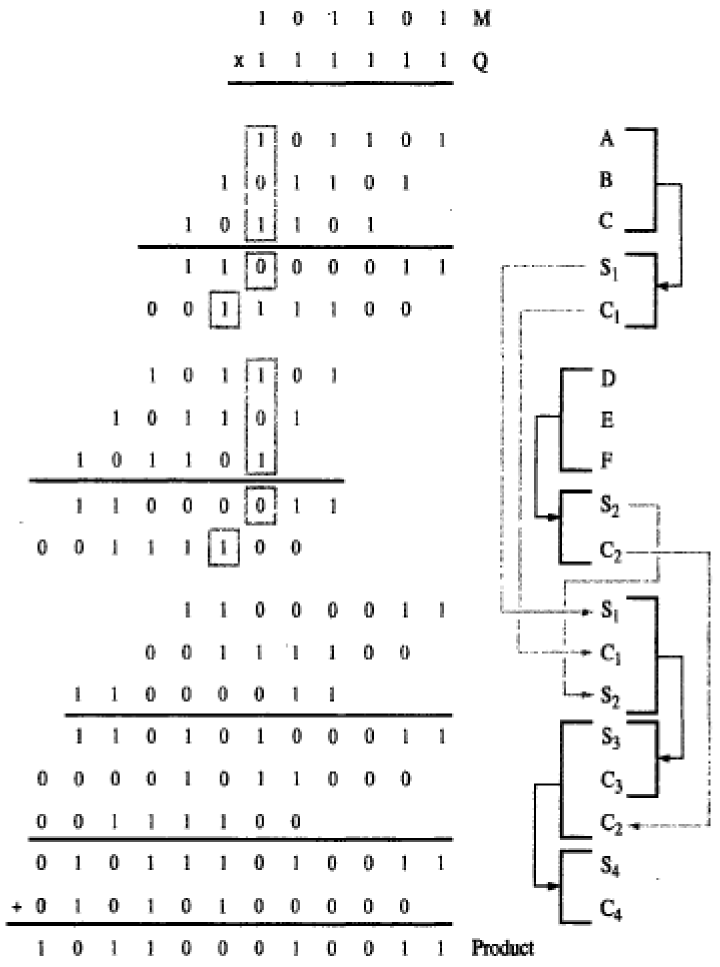


Figure 6.18 The multiplication example from Figure 6.17 performed using carry-save addition.

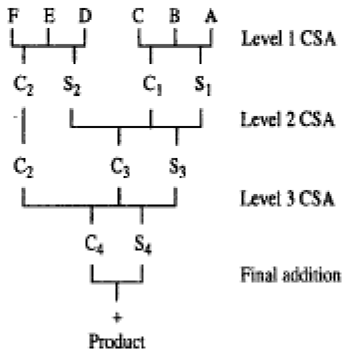


Figure 6.19 Schematic representation of the carry-save addition operations in Figure 6.18.

INTEGER DIVISION

• An n-bit positive-divisor is loaded into register M.
An n-bit positive-dividend is loaded into register Q at the start of the operation.
Register A is set to 0 (Figure 6.21).

• After division operation, the n-bit quotient is in register Q, and
the remainder is in register A.

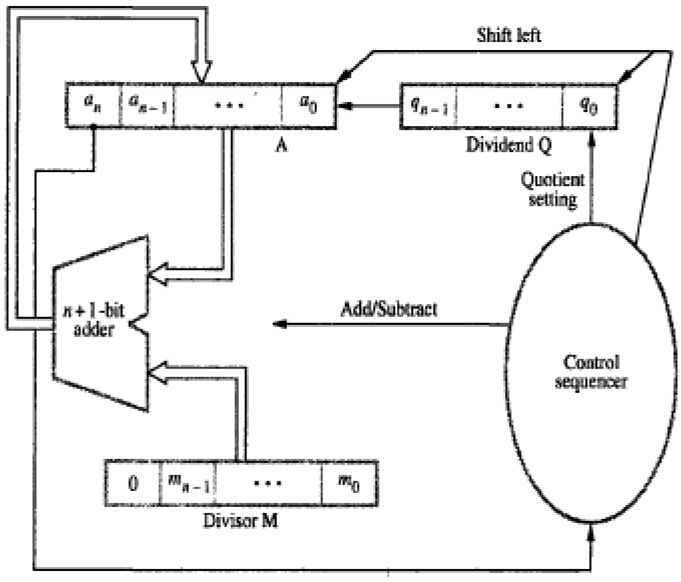


Figure 6.21 Circuit arrangement for binary division.

NON-RESTORING DIVISION

• Procedure:

Step 1: Do the following n times

i) If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A (Figure 6.23).

ii) Now, if the sign of A is 0, set q_0 to 1; otherwise set q_0 to 0.

Step 2: If the sign of A is 1, add M to A (restore).

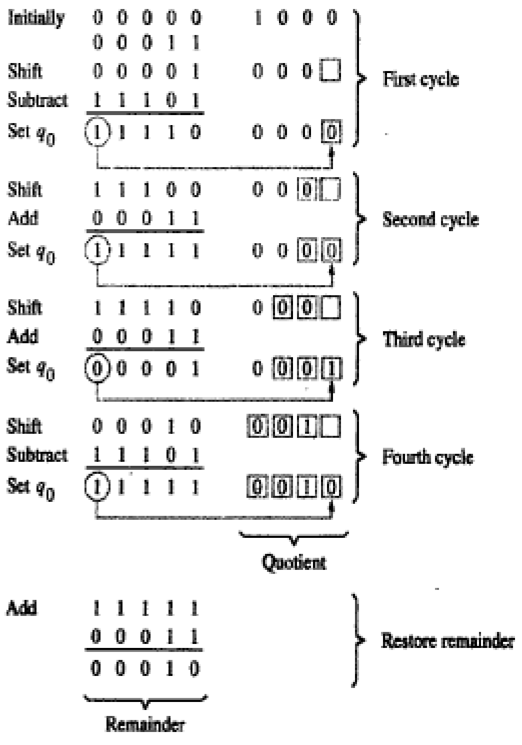


Figure 6.23 A nonrestoring-division example.

RESTORING DIVISION

• Procedure: Do the following n times

1) Shift A and Q left one binary position (Figure 6.22).
2) Subtract M from A, and place the answer back in A
3) If the sign of A is 1, set q_0 to 0 and add M back to A(restore A).
   If the sign of A is 0, set q_0 to 1 and no restoring done.

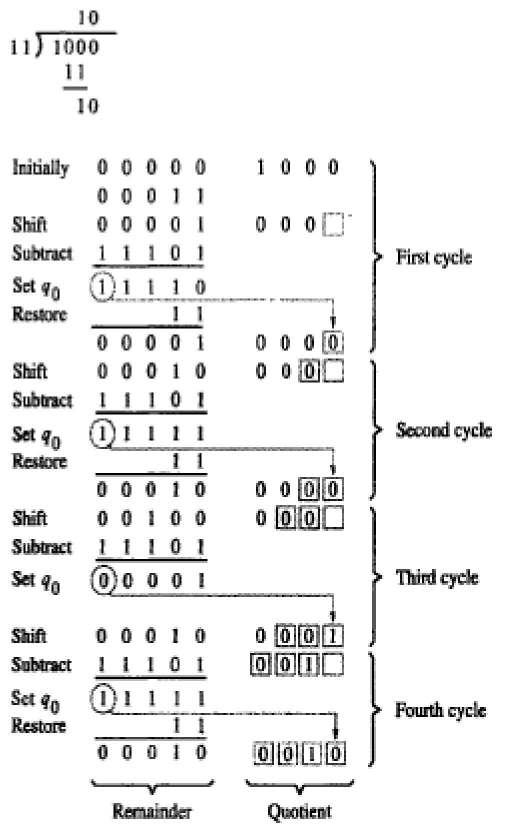


Figure 6.22 A restoring-division example.

c

FLOATING-POINT NUMBERS & OPERATIONS
IEEE STANDARD FOR FLOATING POINT NUMBERS

• Single precision representation occupies a single 32-bit word.
The scale factor has a range of 2^{-126} to 2^{+127} (which is approximately equal to 10^{+38}).
• The 32 bit word is divided into 3 fields: sign(1 bit), exponent(8 bits) and mantissa(23 bits).
• Signed exponent=E
Unsigned exponent E'=E+127. Thus, E' is in the range 0<E'<255.
• The last 23 bits represent the mantissa. Since binary normalization is used, the MSB of the mantissa is always equal to 1. (M represents fractional-part).
• The 24-bit mantissa provides a precision equivalent to about 7 decimal-digits (Figure 6.24).
• Double precision representation occupies a single 64-bit word. And E' is in the range 1<E'<2046.
• The 53-bit mantissa provides a precision equivalent to about 16 decimal-digits.

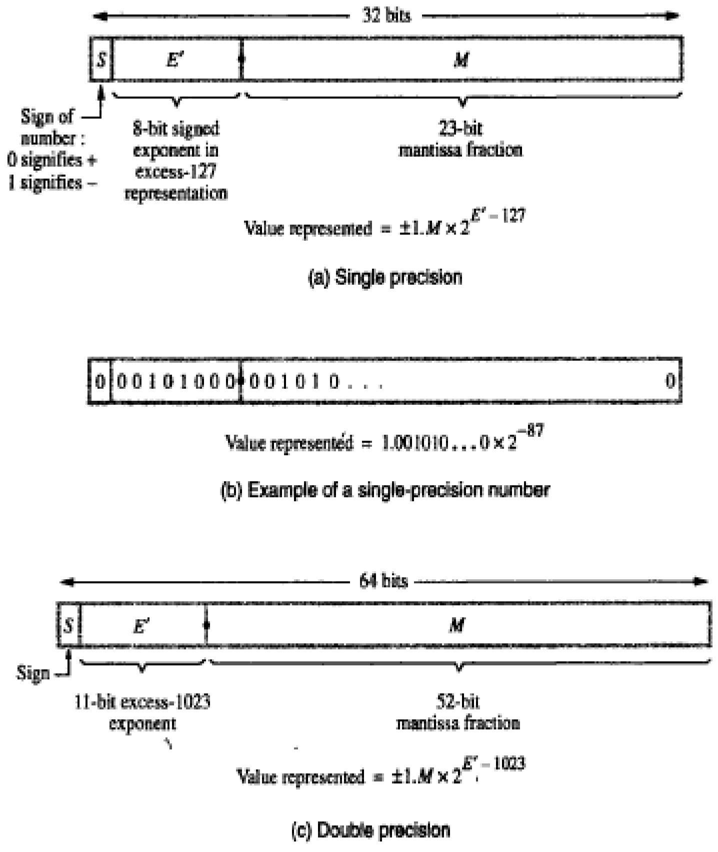

32 bits
S
E'
M
Sign of number : 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
Value represented = ±1.M × 2^(E' − 127)
(a) Single precision
0 00101000 001010 . . . 0
Value represented = 1.001010...0 × 2^−87
(b) Example of a single-precision number
64 bits
S
E'
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction
Value represented = ±1.M × 2^(E' − 1023)
(c) Double precision

Figure 6.24 IEEE standard floating-point formats.

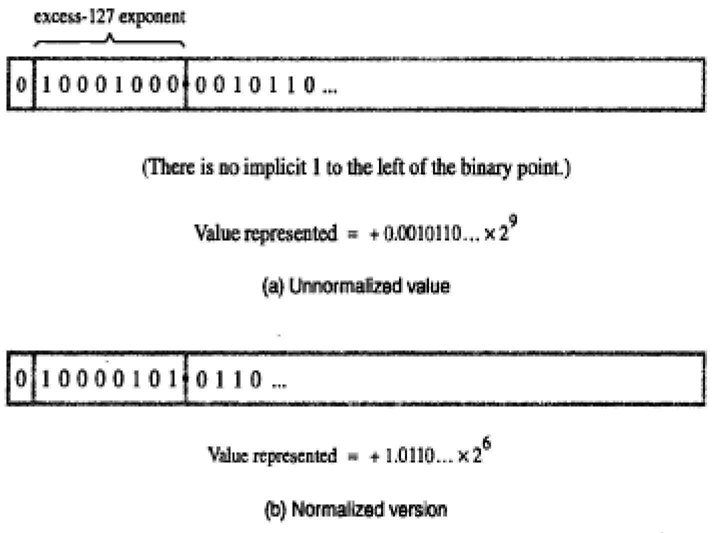

excess-127 exponent
0 10001000 0010110 ...
(There is no implicit 1 to the left of the binary point.)
Value represented = +0.0010110... × 2^9
(a) Unnormalized value
0 10000101 0110 ...
Value represented = +1.0110... × 2^6
(b) Normalized version

Figure 6.25 Floating-point normalization in IEEE single-precision format.

ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

Multiply Rule

1) Add the exponents & subtract 127.
2) Multiply the mantissas & determine sign of the result.
3) Normalize the resulting value if necessary.

Divide Rule

1) Subtract the exponents & add 127.
2) Divide the mantissas & determine sign of the result.
3) Normalize the resulting value if necessary.

Add/Subtract Rule

1) Choose the number with the smaller exponent & shift its mantissa right a number of steps equal to the difference in exponents(n).
2) Set exponent of the result equal to larger exponent.
3) Perform addition/subtraction on the mantissas & determine sign of the result.
4) Normalize the resulting value if necessary.

IMPLEMENTING FLOATING-POINT OPERATIONS

• First compare exponents to determine how far to shift the mantissa of the number with the smaller exponent.
• The shift-count value n
→ is determined by 8 bit subtractor &
→ is sent to SHIFTER unit.
• In step 1, sign is sent to SWAP network (Figure 6.26).
If sign=0, then $E_A > E_B$ and mantissas M_A & M_B are sent straight through SWAP network.
If sign=1, then $E_A < E_B$ and the mantissas are swapped before they are sent to SHIFTER.
• In step 2, 2:! MUX is used. The exponent of result E is tentatively determined as E_A if $E_A > E_B$ or E_B if $E_A < E_B$
• In step 3, CONTROL logic
→ determines whether mantissas are to be added or subtracted.
→ determines sign of the result.
• In step 4, result of step 3 is normalized. The number of leading zeros in M determines number of bit shifts(X) to be applied to M.

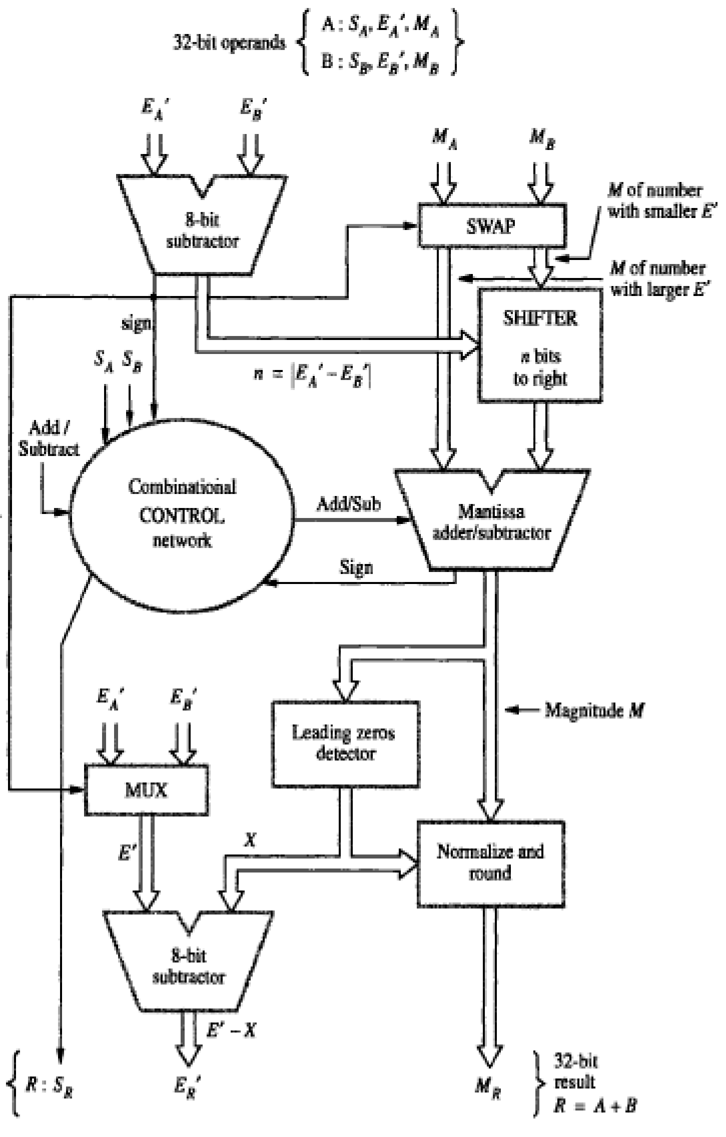


Figure 6.26 Floating-point addition-subtraction unit.

UNIT 7: BASIC PROCESSING UNIT

SOME FUNDAMENTAL CONCEPTS

• To execute an instruction, processor has to perform following 3 steps:

1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation can be written as

   IR$\leftarrow$ [[PC]]

2) Increment PC by 4

   PC$\leftarrow$ [PC] +4

3) Carry out the actions specified by instruction (in the IR).

• The first 2 steps are referred to as fetch phase;

   Step 3 is referred to as execution phase.

SINGLE BUS ORGANIZATION

• MDR has 2 inputs and 2 outputs. Data may be loaded
 → into MDR either from memory-bus (external) or
 → from processor-bus (internal).

• MAR's input is connected to internal-bus, and
 MAR's output is connected to external-bus.

• Instruction-decoder & control-unit is responsible for
 → issuing the signals that control the operation of all the units inside the processor (and for interacting with memory bus).
 → implementing the actions specified by the instruction (loaded in the IR)

• Registers R0 through R(n-1) are provided for general purpose use by programmer.

• Three registers Y, Z & TEMP are used by processor for temporary storage during execution of some instructions. These are transparent to the programmer i.e. programmer need not be concerned with them because they are never referenced explicitly by any instruction.

• MUX(Multiplexer) selects either
 → output of Y or
 → constant-value 4(is used to increment PC content).This is provided as input A of ALU.

• B input of ALU is obtained directly from processor-bus.

• As instruction execution progresses, data are transferred from one register to another, often passing through ALU to perform arithmetic or logic operation.

• An instruction can be executed by performing one or more of the following operations:
 1) Transfer a word of data from one processor-register to another or to the ALU.
 2) Perform arithmetic or a logic operation and store the result in a processor-register.
 3) Fetch the contents of a given memory-location and load them into a processor-register.
 4) Store a word of data from a processor-register into a given memory-location.

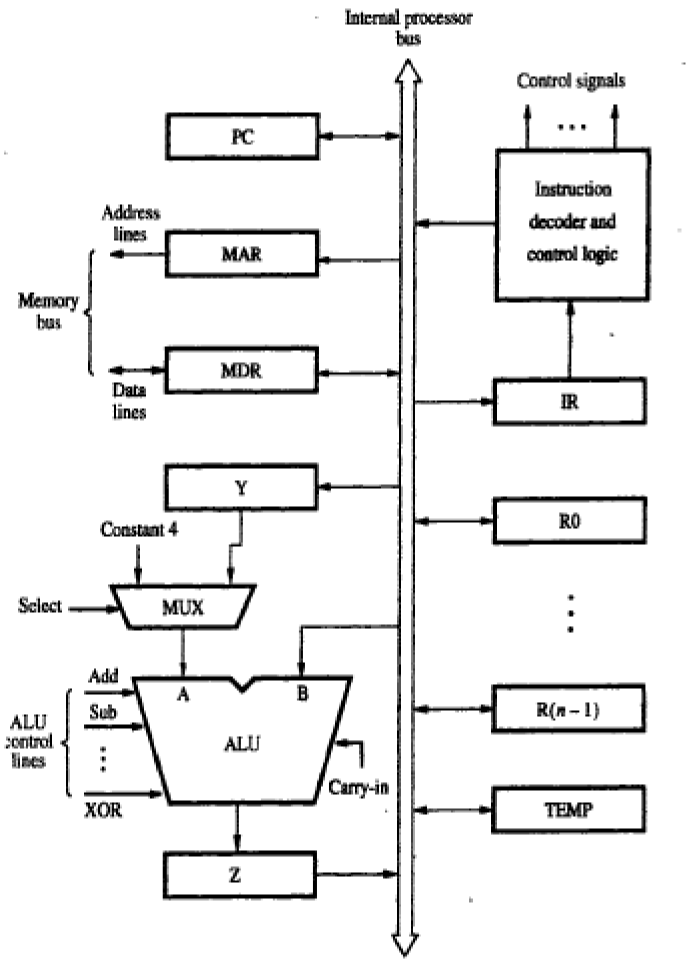


Figure 7.1 Single-bus organization of the datapath inside a processor.

REGISTER TRANSFERS

• Instruction execution involves a sequence of steps in which data are transferred from one register to another.
• Input & output of register Ri is connected to bus via switches controlled by 2 control-signals: Ri_{in} & Ri_{out}. These are called *gating signals.*
• When Ri_{in}=1, data on bus is loaded into Ri.
  Similarly, when Ri_{out}=1, content of Ri is placed on bus.
• When Ri_{out}=0, bus can be used for transferring data from other registers.
• All operations and data transfers within the processor take place within time-periods defined by the processor-clock.
• When edge-triggered flip-flops are not used, 2 or more clock-signals may be needed to guarantee proper transfer of data. This is known as *multiphase clocking*.

Input & Output Gating for one Register Bit

• A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
• When Ri_{in}=1, mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock.
  When Ri_{in}=0, mux feeds back the value currently stored in flip-flop.
• Q output of flip-flop is connected to bus via a tri-state gate.
  When Ri_{out}=0, gate's output is in the high-impedance state. (This corresponds to the open-circuit state of a switch).
  When Ri_{out}=1, the gate drives the bus to 0 or 1, depending on the value of Q.

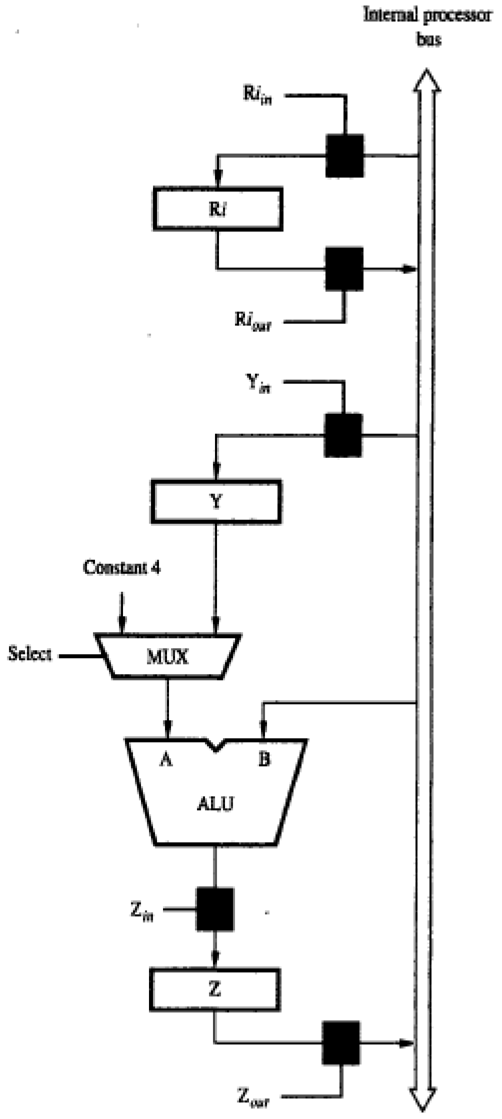


Figure 7.2 Input and output gating for the registers in Figure 7.1.

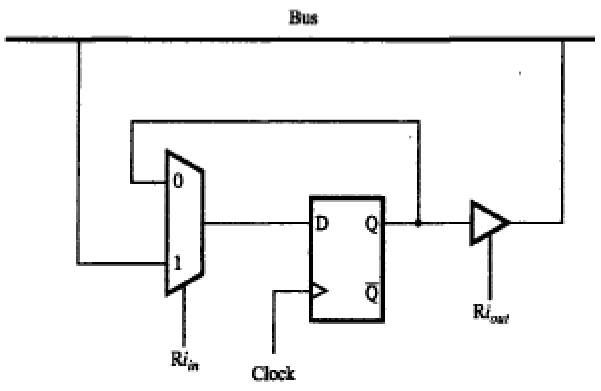


Figure 7.3 Input and output gating for one register bit.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

• The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
• One of the operands is output of MUX &
 the other operand is obtained directly from bus.
• The result (produced by the ALU) is stored temporarily in register Z.
• The sequence of operations for [R3]🡐[R1]+[R2] is as follows

```
1) R1out, Yin                 //transfer the contents of R1 to Y register
2) R2out, SelectY, Add, Zin   //R2 contents are transferred directly to B input of ALU.
                              // The numbers of added. Sum stored in register Z
3) Zout, R3in                 //sum is transferred to register R3
```

• The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

Write the complete control sequence for the instruction : Move (R_s),R_d

• This instruction copies the contents of memory-location pointed to by R_s into R_d. This is a memory read operation. This requires the following actions

→ fetch the instruction

→ fetch the operand (i.e. the contents of the memory-location pointed by R_s).

→ transfer the data to R_d.

• The control-sequence is written as follows

1) PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2) Z_{out}, PC_{in}, Y_{in}, WMFC
3) MDR_{out}, IR_{in}
4) R_s, MAR_{in}, Read
5) MDR_{inE}, WMFC
6) MDR_{out}, R_d, End

FETCHING A WORD FROM MEMORY

• To fetch instruction/data from memory, processor transfers required address to MAR (whose output is connected to address-lines of memory-bus).

At the same time, processor issues Read signal on control-lines of memory-bus.

• When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers

• MFC (Memory Function Completed): Addressed-device sets MFC to 1 to indicate that the contents of the specified location

→ have been read &

→ are available on data-lines of memory-bus

• Consider the instruction Move (R1),R2. The sequence of steps is:

1) $R1_{out}$, MAR_{in}, Read ;desired address is loaded into MAR & Read command is issued

2) MDR_{inE}, WMFC ;load MDR from memory bus & Wait for MFC response from memory

3) MDR_{out}, $R2_{in}$;load R2 from MDR

where WMFC=control signal that causes processor's control circuitry to wait for arrival of MFC signal

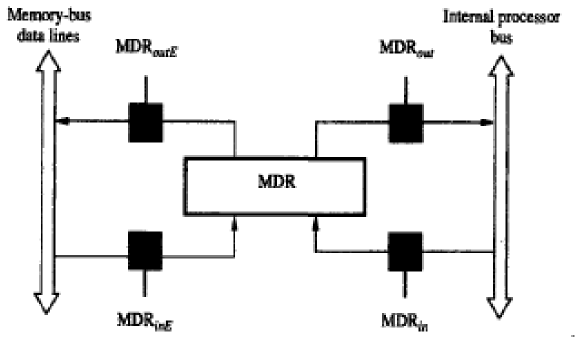


Figure 7.4 Connection and control signals for register MDR.

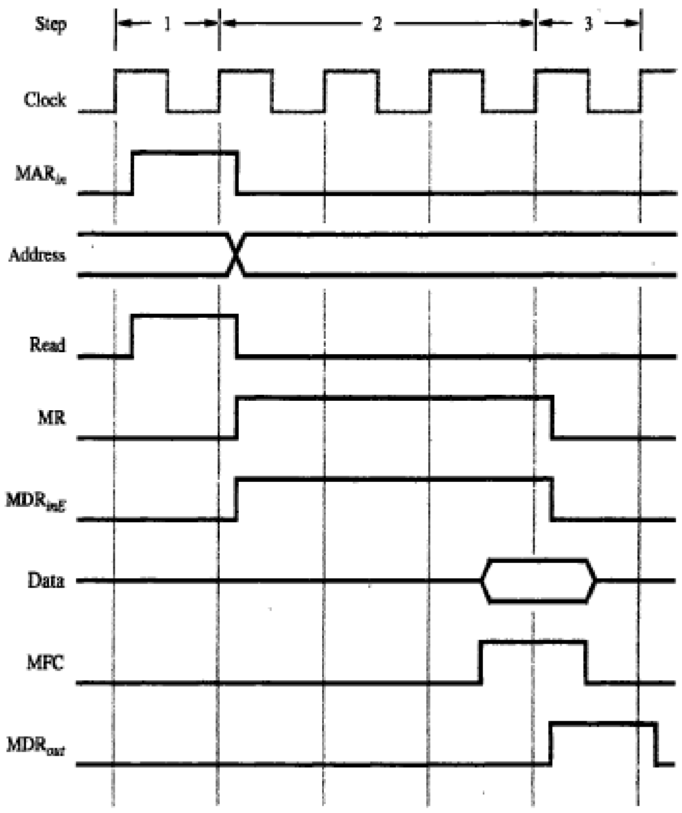


Figure 7.5 Timing of a memory Read operation.

Storing a Word in Memory

• Consider the instruction *Move R2,(R1).* This requires the following sequence:

1) $R1_{out}$, MAR_{in} ;desired address is loaded into MAR

2) $R2_{out}$, MDR_{in}, Write ;data to be written are loaded into MDR & Write command is issued

3) MDR_{outE}, WMFC ;load data into memory location pointed by R1 from MDR

EXECUTION OF A COMPLETE INSTRUCTION

• Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instruction.
2) Fetch the first operand.
3) Perform the addition.
4) Load the result into R1.

• Control sequence for execution of this instruction is as follows

1) PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2) Z_{out}, PC_{in}, Y_{in}, WMFC
3) MDR_{out}, IR_{in}
4) $R3_{out}$, MAR_{in}, Read
5) $R1_{out}$, Y_{in}, WMFC
6) MDR_{out}, SelectY, Add, Z_{in}
7) Z_{out}, $R1_{in}$, End

• Instruction execution proceeds as follows:

Step1--> The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC's content), and the result is stored in Z
Step2--> Updated value in Z is moved to PC.
Step3--> Fetched instruction is moved into MDR and then to IR.
Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued.
Step5--> Contents of R1 are transferred to Y to prepare for addition.
Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.
Step7--> Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

• Control sequence for an unconditional branch instruction is as follows:

1) PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2) Z_{out}, PC_{in}, Y_{in}, WMFC
3) MDR_{out}, IR_{in}
4) Offset-field-of-IR_{out}, Add, Z_{in}
5) Z_{out}, PC_{in}, End

• The processing starts, as usual, the fetch phase ends in step3.
• In step 4, the offset-value is extracted from IR by instruction-decoding circuit.
• Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.
• In step 5, the result, which is the branch-address, is loaded into the PC.
• The offset X used in a branch instruction is usually the difference between the branch target-address and the address immediately following the branch instruction. (For example, if the branch instruction is at location 1000 and branch target-address is 1200, then the value of X must be 196, since the PC will be containing the address 1004 after fetching the instruction at location 1000).
• In case of conditional branch, we need to check the status of the condition-codes before loading a new value into the PC.

e.g.: Offset-field-of-IR_{out}, Add, $Z_{in,}$ If N=0 then End
If N=0, processor returns to step 1 immediately after step 4.
If N=1, step 5 is performed to load a new value into PC.

MULTIPLE BUS ORGANIZATION

• All general-purpose registers are combined into a single block called the *register file*.

• Register-file has 3 ports. There are 2 outputs allowing the contents of 2 different registers to be simultaneously placed on the buses A and B.

• Register-file has 3 ports.

1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.
2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.

• Buses A and B are used to transfer source-operands to A & B inputs of ALU.

• Result is transferred to destination over bus C.

• Incrementer-unit is used to increment PC by 4.

• Control sequence for the instruction *Add R4,R5,R6* is as follows

1) PC_{out}, R=B, MAR_{in}, Read, IncPC
2) WMFC
3) MDR_{out}, R=B, IR_{in}
4) $R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

• Instruction execution proceeds as follows:

Step 1--> Contents of PC are passed through ALU using R=B control-signal and loaded into MAR to start a memory Read operation. At the same time, PC is incremented by 4.

Step2--> Processor waits for MFC signal from memory.

Step3--> Processor loads requested-data into MDR, and then transfers them to IR.

Step4--> The instruction is decoded and add operation take place in a single step.

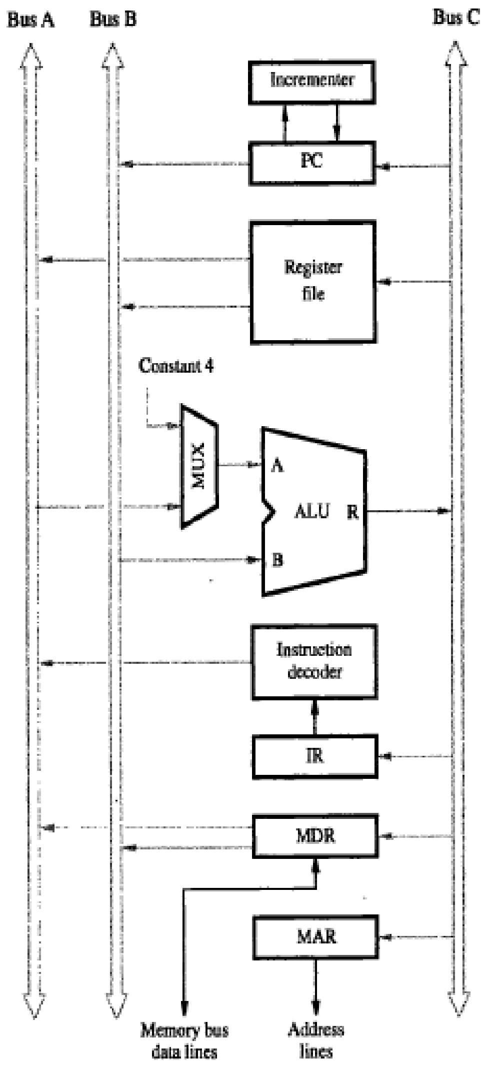


Figure 7.8 Three-bus organization of the datapath.

Note:

To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. There are two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

HARDWIRED CONTROL

• Decoder/encoder block is a combinational-circuit that generates required control-outputs depending on state of all its inputs.

• Step-decoder provides a separate signal line for each step in the control sequence.
Similarly, output of instruction-decoder consists of a separate line for each machine instruction.

• For any instruction loaded in IR, one of the output-lines INS_1 through INS_m is set to 1, and all other lines are set to 0.

• The input signals to encoder-block are combined to generate the individual control-signals Y_{in}, PC_{out}, Add, End and so on.

• For example, $Z_{in}=T_1+T_6.ADD+T_4.BR$;This signal is asserted during time-slot T_1 for all instructions,
during T_6 for an Add instruction
during T_4 for unconditional branch instruction

• When RUN=1, counter is incremented by 1 at the end of every clock cycle.
When RUN=0, counter stops counting.

• Sequence of operations carried out by this machine is determined by wiring of logic elements, hence the name "hardwired".

• Advantage: Can operate at high speed.
Disadvantage: Limited flexibility.

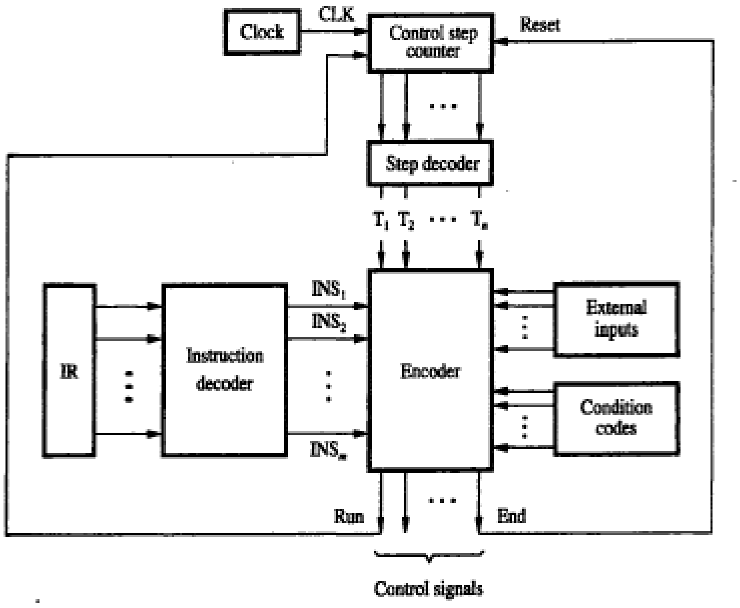


Figure 7.11 Separation of the decoding and encoding functions.

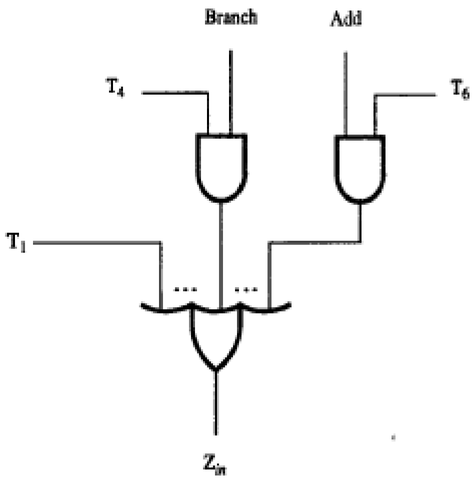


Figure 7.12 Generation of the Z_{in} control signal for the processor in Figure 7.1.

COMPLETE PROCESSOR

- This has separate processing-units to deal with integer data and floating-point data.
- A data-cache is inserted between these processing-units & main-memory.
- Instruction-unit fetches instructions
  → from an instruction-cache or
  → from main-memory when desired instructions are not already in cache
- Processor is connected to system-bus &
  hence to the rest of the computer by means of a bus interface
- Using separate caches for instructions & data is common practice in many processors today.
- A processor may include several units of each type to increase the potential for concurrent operations.

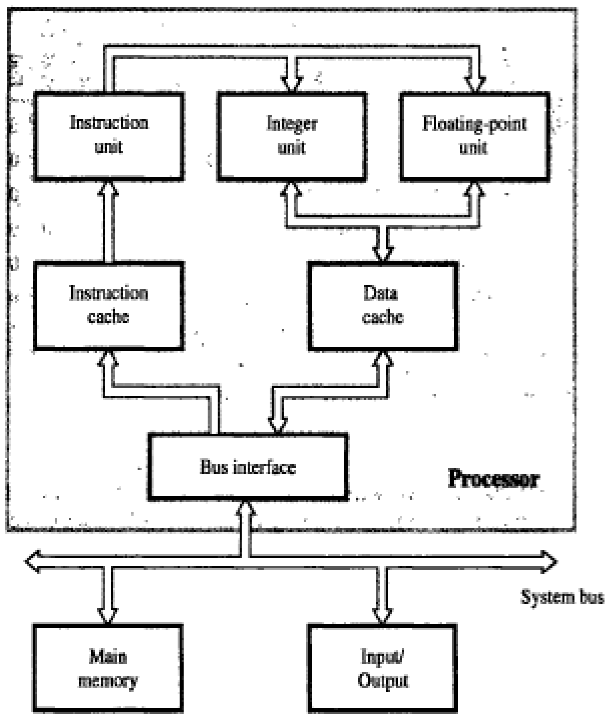
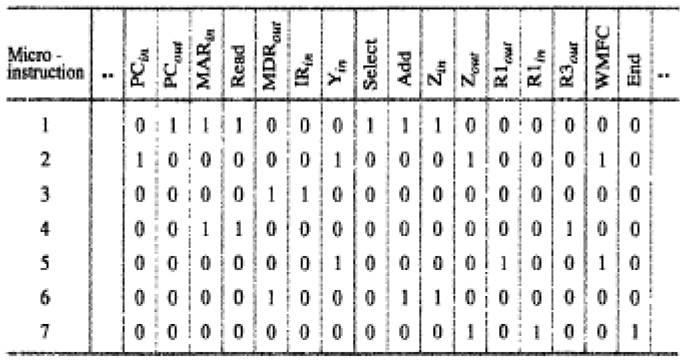


Figure 7.14 Block diagram of a complete processor.

MICROPROGRAMMED CONTROL

• Control-signals are generated by a program similar to machine language programs.
• *Control word(CW)* is a word whose individual bits represent various control-signals(like Add, End, Z_{in}). {Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in the CW}.
• Individual control-words in microroutine are referred to as *microinstructions.*
• A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the *microroutine*.
• The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the *control store(CS).*
• Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.
• *Microprogram counter(μPC)* is used to read CWs sequentially from CS.
• Every time a new instruction is loaded into IR, output of "starting address generator" is loaded into μPC.
• Then, μPC is automatically incremented by clock,
causing successive microinstructions to be read from CS.
Hence, control-signals are delivered to various parts of processor in correct sequence.



Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

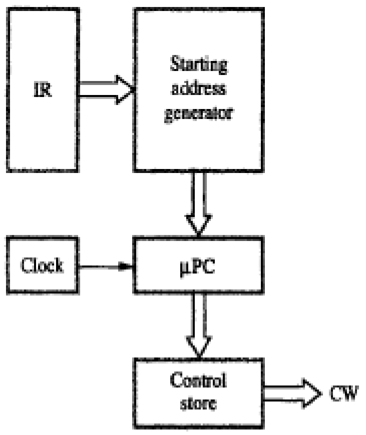
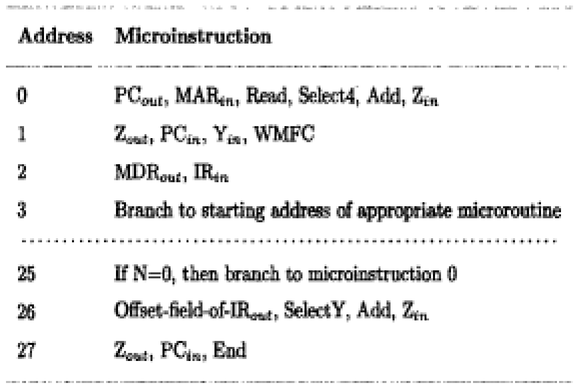


Figure 7.16 Basic organization of a microprogrammed control unit.

ORGANIZATION OF MICROPROGRAMMED CONTROL UNIT (TO SUPPORT CONDITIONAL BRANCHING)

• In case of conditional branching, microinstructions specify which of the external inputs, condition-codes should be checked as a condition for branching to take place.
• The *starting and branch address generator block* loads a new address into μPC when a microinstruction instructs it to do so.
• To allow implementation of a conditional branch, inputs to this block consist of
→ external inputs and condition-codes
→ contents of IR
• μPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations
i) When a new instruction is loaded into IR, μPC is loaded with starting-address of microroutine for that instruction.
ii) When a Branch microinstruction is encountered and branch condition is satisfied, μPC is loaded with branch-address.
iii) When an End microinstruction is encountered, μPC is loaded with address of first CW in microroutine for instruction fetch cycle.



Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
...	...
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

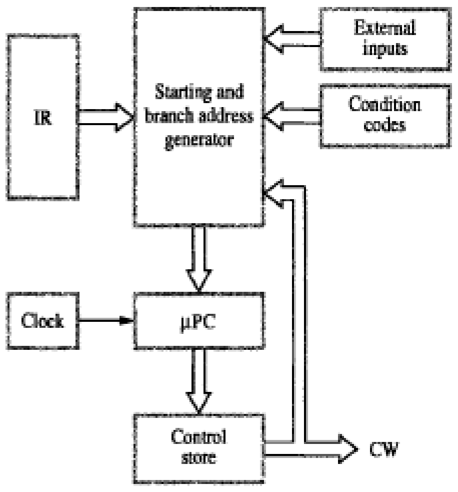


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

MICROINSTRUCTIONS

• Drawbacks of microprogrammed control:

1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.
2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.

• Solution: Signals can be grouped because

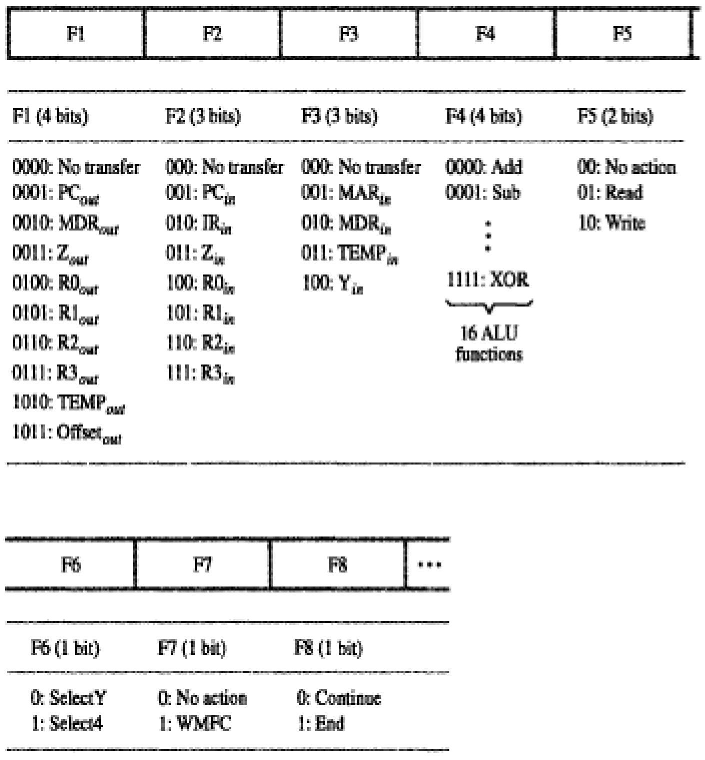
1) Most signals are not needed simultaneously.
2) Many signals are mutually exclusive.

• Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control signals.

• Advantage: This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals).

Vertical organization	Horizontal organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization	The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization
This approach results in considerably slower operating speeds because more micrinstructions are needed to perform the desired control functions	This approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources

Microinstruction



F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functions	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	•••

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Figure 7.19 An example of a partial format for field-encoded microinstructions.

MICROPROGRAM SEQUENCING

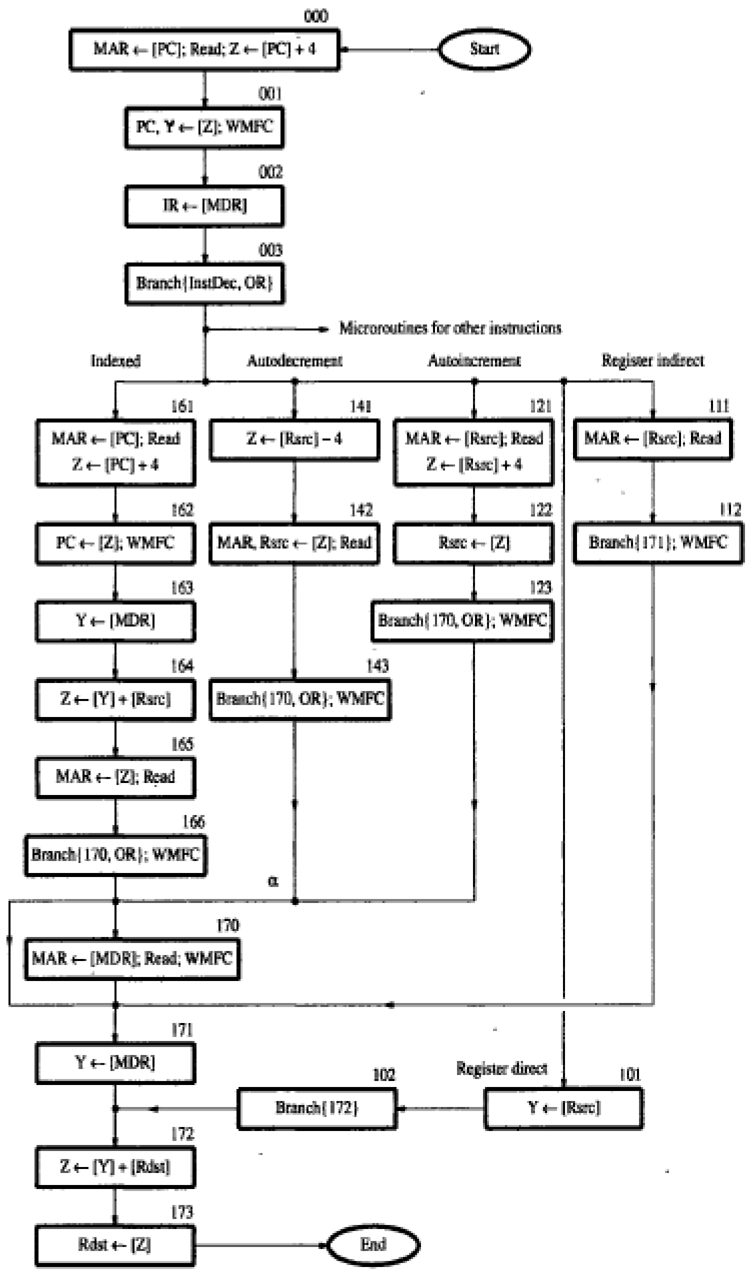


Figure 7.20 Flowchart of a microprogram for the Add src,Rdst instruction.

• Two major disadvantage of microprogrammed control is:

1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.

2) Execution time is longer because it takes more time to carry out the required branches.

• Consider the instruction *Add src,Rdst* ;which adds the source-operand to the contents of Rdst and places the sum in Rdst.

• Let source-operand can be specified in following addressing modes: register, autoincrement, autodecrement and indexed as well as the indirect forms of these 4 modes.

• Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box.

• The microinstruction is located at the address indicated by the octal number (001,002).

BRANCH ADDRESS MODIFICATION USING BIT-ORING

• Consider the point labeled α in the figure. At this point, it is necessary to choose between direct and indirect addressing modes.
• If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.
If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171.
• The most efficient way to bypass microinstruction 170 is to have the preceding branch microinstructions specify the address 170 and then use an OR gate to change the LSB of this address to 1 if the direct addressing mode is involved. This is known as the *bit-ORing* technique.

WIDE BRANCH ADDRESSING

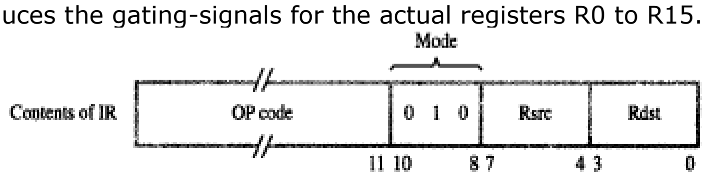
• The instruction-decoder(InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR.
• Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC).
• The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

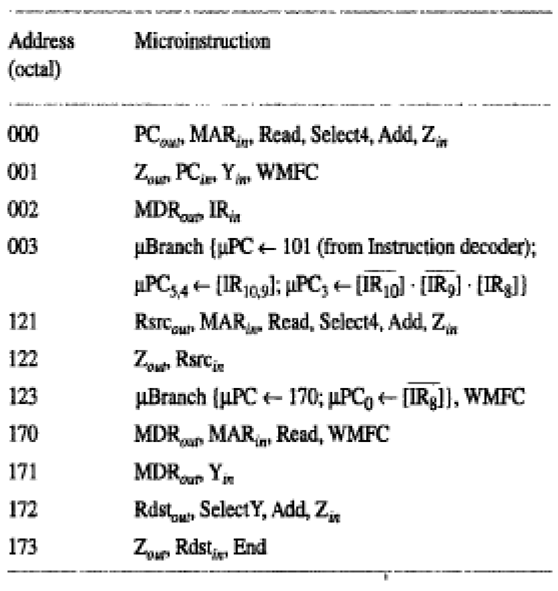
Use of WMFC

• WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171.
• WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting-period.

Detailed Examination

• Consider *Add (Rsrc)+,Rdst;* which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode).
• In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version.
• The processor has 16 registers that can be used for addressing purposes; each specified using a 4-bit-code.
• There are 2 stages of decoding:
1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.
2) The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15.





Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {$\mu PC \leftarrow 170$; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21 Microinstruction for Add (Rsrc)+,Rdst.

MICROINSTRUCTIONS WITH NEXT-ADDRESS FIELDS

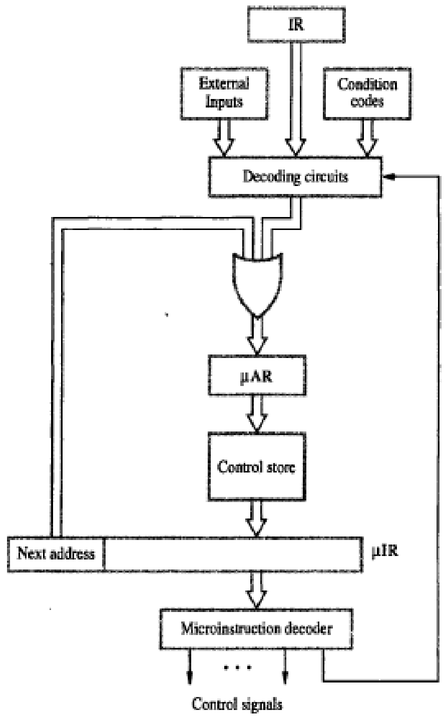
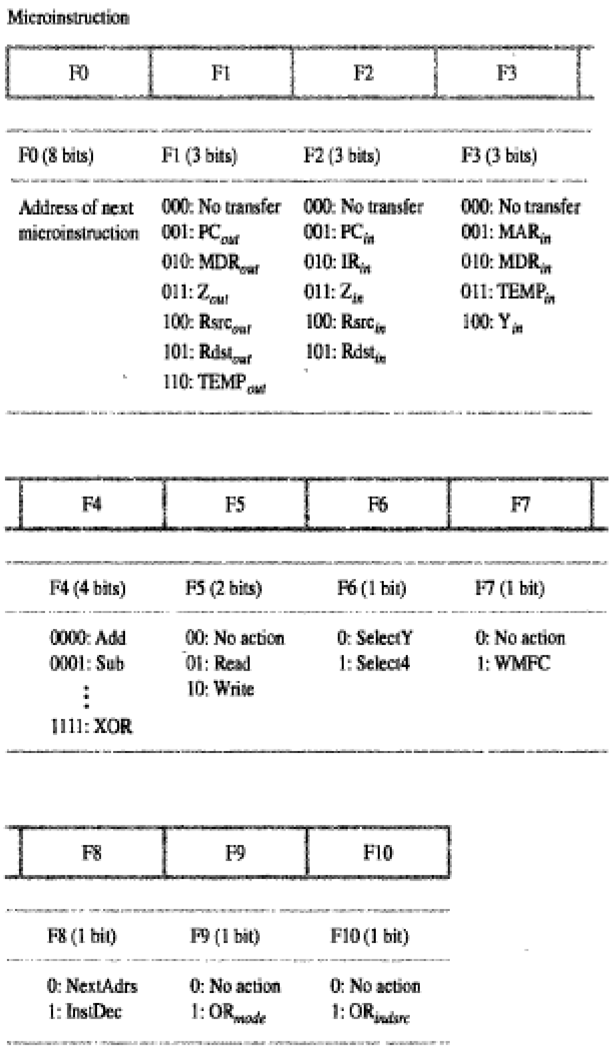


Figure 7.22 Microinstruction-sequencing organization.



Microinstruction

F0	F1	F2	F3
F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7
F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10
F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0: NextAdrs 1: InstDec	0: No action 1: OR_{mode}	0: No action 1: OR_{indsrc}

Figure 7.23 Format for microinstructions in the example of Section 7.5.3.

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
000	00000001	001	011	001	0000	01	1	0	0	0	0
001	00000010	011	001	100	0000	00	0	1	0	0	0
002	00000011	010	010	000	0000	00	0	0	0	0	0
003	00000000	000	000	000	0000	00	0	0	1	1	0
121	01010010	100	011	001	0000	01	1	0	0	0	0
122	01111000	011	100	000	0000	00	0	1	0	0	1
170	01111001	010	000	001	0000	01	0	1	0	0	0
171	01111010	010	000	100	0000	00	0	0	0	0	0
172	01111011	101	011	000	0000	00	0	0	0	0	0
173	00000000	011	101	000	0000	00	0	0	0	0	0

Figure 7.24 Implementation of the microroutine of Figure 7.21 using a next-microinstruction address field. (See Figure 7.23 for encoded signals.)

• The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating speed of the computer.
• Solution: Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (This means every microinstruction becomes a branch microinstruction).
• The flexibility of this approach comes at the expense of additional bits for the address-field.
• Advantage: Separate branch microinstructions are virtually eliminated. There are few limitations in assigning addresses to microinstructions. There is no need for a counter to keep track of sequential addresse. Hence, the μPC is replaced with a μAR (Microinstruction Address Register). {which is loaded from the next-address field in each microinstruction}.
• The next-address bits are fed through the OR gate to the μAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.
• The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR.

PREFETCHING MICROINSTRUCTIONS

• Drawback of microprogrammed control: Slower operating speed because of the time it takes to fetch microinstructions from the control-store.
• Solution: Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

Emulation

• The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.
• Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.
• Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.
• Programs written in the machine language of M2 can be then be run on computer M1 i.e. M1 emulates M2.
• Emulation allows us to replace obsolete equipment with more up-to-date machines.
• If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.
• Emulation is easiest when the machines involved have similar architectures.