UNIT –III - CHAPTER 5 – CENTRAL PROCESSING UNIT

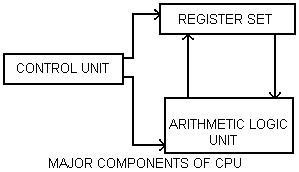
## Introduction

The part of the computer that performs the bulk of data processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Fig. 1.1

The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

## Figure 1.1 Major components of CPU



### **General Register Organization**

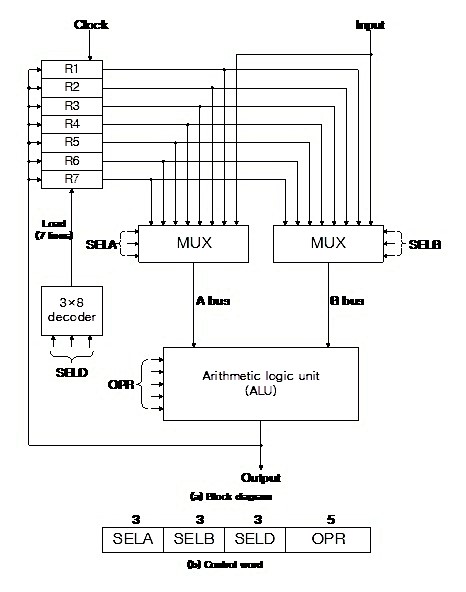
When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic. Logic, and shift microoperations in the processor.

A bus organization for seven CPU registers is shown in Fig. 1.2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU) . The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. To perform the operation

R1 ← R2 + R3

## Figure 1.2 Register set with common ALU



The control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.

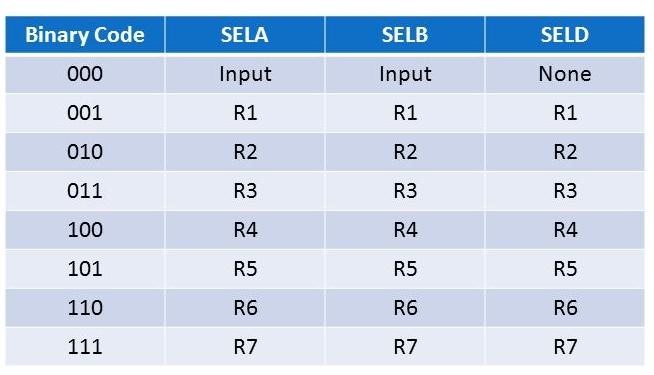
## Control word

There are 14 binary selection inputs in the unit and their combined value specifies a control word. The 14-bit control word is defined in ―Fig. 1.2 (b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14bit control word when applied to the selection inputs specify a particular microoperation

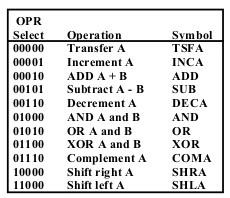
The encoding of the register selections is specified in Table 1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a pre shift capability, or at the output of the ALU to provide post shifting capability. In some cases, the shift operations are included with the ALU.

### Table 1 Encoding of Register Selection Fields



### Table 2 Encoding of ALU Operations



## Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

R1 ← R2 – R3

Specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A – B. The binary control word for the subtract microoperationis 010 011 001 00101 and is obtained as follows:

Field: SELA SELB SELD OPR

Symbol: R2 R3 R1 SUB

Control word: 010 011 001 00101

### **Stack Organization**

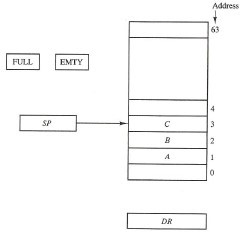
A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last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

The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top.

**Register Stack**

The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A,B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack pushed by incrementing SP and writing a word in the next higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.



In a 64-word stack the stack pointer contains 6 bits because 26= 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

SP← SP + 1 Increment stack pointer

M[SP] ← DR Write item on top of the stack

If (SP = 0) then (FULL ← 1) Check if stack is full

EMTY ← 0 Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack.

SP holds the address of the top of the stack and the M[SP] donates the memory word specified by the address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. If an item is written in the stack. obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if EMPY = 0). The pop operation consists of the following sequence of micro operations:

DR←M[SP] Read item from the top of stack

SP←SP – 1 Decrement stack pointer

If (SP=0) then (EMTY←1) Check is stack is empty

FULL ← 0 Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. An erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMPTY = 1.

## Memory Stack

The implementation of a stack is the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. In memory stack,the portion of computer memory partitioned into three segments program, data, and stack. The program counter PC points at the address of the nest instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

The initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

A new item is inserted with the push operation as follows.

SP←SP – 1

M[SP] ← DR

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

DR ←M[SP]

SP←SP + 1

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

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000), and the other to hold the lower limit (4001). After a push operations, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower–limit register.

## Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in infix notation with each operator written between the operands. Consider the simple arithmetic expression.

A\*B + C\*D

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product A\*B, store this product while computing C\*D, and the sum the two products.

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation, often referred to as Polish notation. places the operator before the operands. The postfix notatio referred to as reverse Polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

A +B Infix notation

+AB Prefix or Polish notation

AB+ Postfix or reverse Polish notation.

The reverse Polish notation is in a form suitable for stack manipulation. The expression

A\*B + C\* D

is written in reverse Polish notation as

AB \* CD \* +

and is evaluated as follows. Scan the expression from left to right. When an operator is reached, perform the operation with the two operands found on the left side of the operator. Remove the two operands and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

For the expression above we find the operator \* after A and B. We perform the operation A\*B and replace A,B, and \* by the product to obtain

(A\*B) CD\* +

The next operator is a \* and its previous two operands are C and D, so we perform C\*D and obtain an expression with two operands and one operator.

(A\*B) (C\*D) +

The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

The conversation from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

Consider the expression (A+B)\*[C\*(D+E)+F]

The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

AB + DE + C\* F+\*

Proceeding from left to right, it adds A and B, then add D and E. Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.

The following numerical example may clarify this procedure. Consider the arithmetic expression

(3\*4) + (5\*6)

In reverse Polish notation, it is expressed as 34 \* 56 \* +

First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator\*. This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next\* replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

### **Instruction Formats**

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are;

1. An operation code field that specifics the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

A register address is a binary number of k bits that defines one of 2k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

ADD X where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M [X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

The instruction format in this type of computer needs three register address fields.Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1, R2, R3

to denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

MOV R1, R2

denotes the transfer R1 ← R2 ( or R2 ← R1, depending on the particular computer). Thus transfer – type instructions need two address fields to specify the source and the destination.

General register – type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

would specify the operation R1 ← R1 + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

Computers with stack organized would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack.

The instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effects of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

All arithmetic and logic instructions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack – organized CPU.

The arithmetic statement

X = (A + B) \* (C + D)

using zero, one, two, or three address instructions. The symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer–type operation and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operand are in memory address A, B, C and D, and the result must be stored in memory at address X.

## Three – Address Instructions

Computers with three–address instruction formats can use each address field to specify whether a processor register or a memory operand. The program in assembly either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) \* (C + D) is shown below, together with comments that explain the register operation of each instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| ADD | R1, A , B |  | R1 ← M [A] + M [B] |
| ADD | R2, C, D |  | R2 ← M [C] + M [D] |
| MUL | X, R1, R2 |  | M [X] ← R1 \* R2 |

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A.

The advantage of the three – address format is that it result in short programs when evaluating arithmetic expressions. The disadvantage is that the binary–coded instructions require too many bits to specify three addresses.

## Two–Address Instructions

Two–address instructions are the most common in commercial computers. Here again each address field can specify either a procedure register or a memory word. The program to evaluate X = (A+B) \* (C+D) is as follows.

|  |  |
| --- | --- |
| MOV | R1, A R1 ← M [A] |
| ADD | R1, B R1 ← R1 + M [ B] |
| MOV | R2, C R2 ← M [ C ] |
| ADD | R2, D R2 ←R2 + M [ D] |
| MUL | R1, R2 R1 ←R1 \* R2 |

MOV X , R1` R1←M [ X ] ← R1

The MOV instruction moves or transfers the operands to and from memory and processor registers.

## One–Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate X = (A + B) \* (C + D) is

|  |  |
| --- | --- |
| LOAD A | AAC ← M [A] |
| ADD | B AC ←AC + M [ B] |
| STORE | T M [ T] ← AC |
| LOAD C | C AC ← M [ C] |
| ADD | D AC ← M [ C] |
| MUL | T AC ← AC + M [ T ] |
| STORE | X M [X] ← AC |

All operations are done between the AC register and a memory operand.

## Zero – Address Instructions

A stack–organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions need an address field to specify the operand that communicates with the stack. The following program shows how X = (A+ B) \* (C + D) will be written for a stack organized computer. (TOS stands for top of stack.)

|  |  |  |
| --- | --- | --- |
| PUSH | A | TOS←A |
| PUSH | B | TOS←B |
| ADD |  | TOS← (A + B) |
| PUSH | C | TOS←C |
| PUSH | D | TOS←D |
| ADD |  | TOS ←(C + D) |
| MUL |  | TOS← (C + D) \* (A + B) |
| POP | X | M [X]←TOS |

## RISC Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC –type CPU consists of LOAD and STORE instructions that have one memory and one memory and one register address, and computational–type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate X = (A+B) \* (C + D)

|  |  |  |
| --- | --- | --- |
| LOAD | R1, A | R1← M [A] |
| LOAD | R2, B | R2 ←M [B] |
| LOAD | R3, C | R3← M [C] | |
| LOAD | R4, D | R4 ←M [D] | |
| ADD | R1, R1, R2 | R1 ← R1 + R2 | |
| ADD | R3, R3, R4 | R3 ← R3 + R4 | |
| MUL | R1, R1, R3 | R1 ←R1 + R3 | |
| STORE | X, R1 | M [X] ← R1 | |

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

### **Addressing Modes**

The addressing mode specifies a rule for interpreting or modifying the address field of the instructions before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction. The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases.
   1. Fetch the instruction from memory
   2. Decode the instruction 3.Execute the instruction.

There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address filed, it may designate a memory address or a processor register. Most addressing modes modify the address field of the instruction there are two modes that need no address field at all. There are the implied and immediate modes.

**Implied Mode** : In this mode the operands are specified implicitly in the definition of the instruction. For example the instruction complement accumulator is an implied mode instruction, all register reference instructions that use an accumulator are implied mode instructions. Zero address instructions in a stack organized computer are implied mode instructions since the operands are implied to be on top of the stack.

### Figure : Instruction format with mode field

|  |  |  |
| --- | --- | --- |
| Opcode | Mode | Address |

**Immediate Mode**: In this mode the operand is specified in the instruction itself, In order words, an immediate mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.

**Register Mode:** In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

**Register Indirect Mode**: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register.

**Auto increment or Auto decrement Mode:**

This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

The effective address is defined to be the memory address obtained from the computation by the given addressing mode. The effective address is the address of the operand in a computational type instruction.

**Direct Address Mode:**

In this mode the effective address is equal to the address part of the instruction.

**Indirect Address Mode:**

In the mode the address field of the instruction gives the address where the effective address is stored in memory.

**Relative Address Mode:**

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

**Indexed Addressing Mode:**

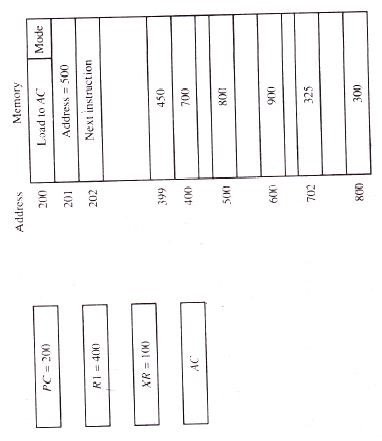
In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

**Based Register Addressing Mode:**

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

## Numerical Example

### Numerical example for addressing modes



The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode that effective address is500 + 202= 702 and the operand is 325. In the index mode the effective address is XR + 500 = 100+500 = 600 and the operand is 900. In the register mode the operand is in R1 and 400is loaded into AC. In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700. The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

#### Reduced Instruction Set

A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a reduced instruction set computer or RISC.

## CISC Characteristics

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.

Another characteristic of CISC architecture is the incorporation of variable-length instruction formats. Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code.

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct addressing. Another memory location may be included in the instruction to store the sum. This requires three memory references during execution of the instruction. As more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. Major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes – typically from 5 to 20 different modes.
4. Variable – length instruction formats
5. Instructions that manipulate operands in memory.

## RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are,

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instruction
4. All operations done within the registers of the CPU
5. Fixed–length, easily decoded instruction format.
6. Single–cycle instruction execution
7. Hardwired rather than micro programmed control.

The small set of instructions of a typical RISC processor consists mostly or register– to-register operations, with only simple load and store operations for memory access.

Results are transferred to memory by means of store instructions. The use of only a few addressing modes results from the fact that almost all instructions have simple register addressing. The instruction length can be fixed and aligned on word boundaries. By simplifying the instructions and their format, it is possible to simplify the control logic. For faster operations, a hardwired control is presented in micro programmed control.

Other characteristics attributed to RISC architecture are:

* 1. A relatively large number of registers in the processor unit
  2. Use of overlapped register windows to speed-up procedure call and return
  3. Efficient instruction pipeline
  4. Compiler support for efficient translation of high-level language programs into machine language programs

A large number of registers is useful for storing intermediate results and for optimizing operand references. The advantage of register storage as op-posed to memory storage is that registers can transfer information to other registers much faster than the transfer of information to and from memory.

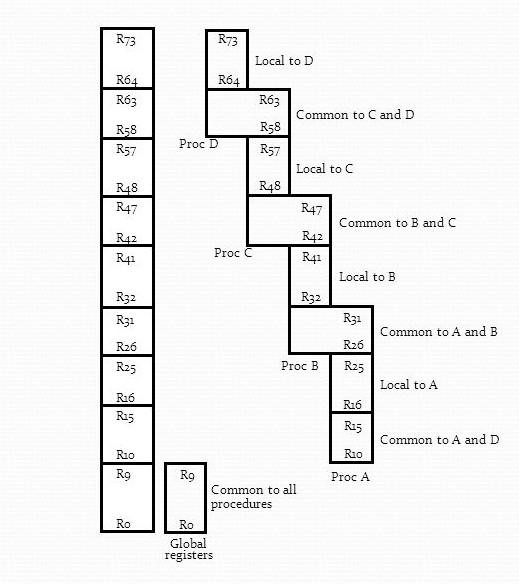
## Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time consuming operation.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlapped register windows is illustrated in Fig. system has a total of 74 registers. Registers RO through R9 are global that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low register of the called procedure, and therefore the parameters automatically transfer from calling to called procedure .

### Overlapped Register Windows



Suppose that procedure A calls procedure B. Registers R26 through R31 are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers. Procedure B uses local registers R32 through R41 for local variable storage. If procedure B calls procedure C, it will pass the parameters through registers R42 through R47. When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers R26 through R31 and transfers back to the register window of procedure A. Note that registers R10 through common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

The 10 global registers R0 through R9 are available to all procedures. This includes 10 global registers, 10 local registers, six low overlapping register, and six high overlapping registers.

number of global registers = G number of local registers in each window = L number of registers common to two windows = C number of windows = W

The number registers available for each window is calculated as folk window size = L + 2C + G

The total number of registers needed in the processor is register file = (L + C)W + G

## Berkeley RISC I

The Berkeley RISC is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, -16 -, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows registers in each. Since only one set of 32 registers in a window is

Figure Berkeley RISC I instruction formats.

### 31 24 23 19 18 14 13 12 5 4 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Opcode | Rd | Rs | | 0 | Not used | S2 |
|  | 8 5 5 1 8 5  (a) Register mode: (S2 specifies a register)  31 24 23 19 18 14 13 12 0 | | | | | |
| Opcode | Rd | Rs | | 1 | S2 | |
|  | 8 5 5 1 13  (b) Register—immediate mode: (S2 specifies an operand)  31 24 23 19 18 0 | | | | | |
| Opcode | COND | | Y | | | |

### 8 5 19 (C) P C relative mode

accessed at any given time, the instruction format can specify a processor register with a register field of five bits.

The given figure shows the 32-bit instruction formats used for register-to-register instructions and memory access instructions. Seven of the bits in the operation code specify an operation, and the eighth bit indicates whether to update the status bits after an ALU operation. For register-to-register instructions, the 5-bit Rd field selects one of the 32 registers as a destination for the result of the operation. The operation is performed with the data specified in fields Rs and S2. Rs is one of the source registers. If bit 13 of the instruction is 0, the low-order 5 bits of S2 specify register. If bit 13 of the instruction is 1, S2 specifies a sign-extended 13-bit constant. Thus the instruction has a three-address format, but the second source may be either a register or an immediate operand. Memory access instructions use Rs to specify a 32-bit address in a register and S2 to specify an offset. Register R0 contains all 0's, so it can be used in any field to specify a zero quantity. The third instruction format combines the last three fields to form a 19-bit relative address Y and is used primarily with the jump and call instructions. The COND field replaces the Rd field for jump instructions and is used to specify one of 16 possible branch conditions.

**COMPUTER ARITHMETIC**

## Addition and Subtraction

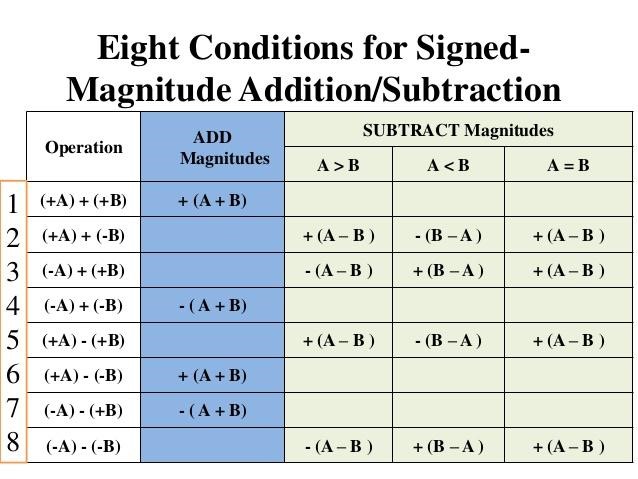
There are three ways of representing negative fixed–point binary numbers : signed – magnitude, signed -1‘s complement, or signed-2‘s complement. Most computers use the signed – 2‘s complement representation when performing arithmetic operations with integers. For floating–point operations, most computers use the signed–magnitude representation for the mantissa.

## Addition and Subtraction with Signed–Magnitude Numbers

The representation of numbers in signed – magnitude is familiar because it is used in everyday arithmetic calculations. We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 5-1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0

The algorithms for addition and subtraction are derived as follows. Addition (Subtraction) algorithm : when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result when the signs of A and B are different (identical), compare the magnitudes and

## TABLE-1 Addition and Subtraction of Signed – Magnitude Numbers



Subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

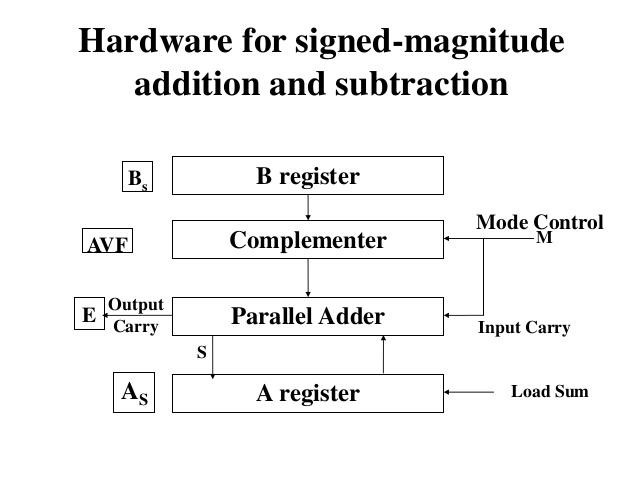
## Hardware Implementation

Let A and B be two registers that hold the magnitudes of the numbers, and As and Bs be two flip–flops that hold the corresponding signs. The result of the operation may be transferred to a third register,however, a saving is achieved if the result is transferred into A and As. Thus A and As together form an accumulator register.

Consider now the hardware implementation of the algorithms above. First, a parallel adder is need to perform the micro operation A + B. Second, a comparator circuit is needed to establish if A > B, A =B, or A<B. Third, two parallel–subtract or circuits are needed to perform the micro operations A –B and B –A. The sign relationship can be determined from an exclusive–OR gate with As and Bs as inputs.

Figure shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip–flops As and Bs. Subtraction is done by adding A to the 2‘s complement of B. The output carry is transferred to flip–flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add–overflow flip–flop AVF holds the overflow bit when A and B are added.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B based on the state of the mode control M. The complementer consists of exclusive –OR gates and the parallel adder consists of full–adder circuits. The M signal is also applied to the input carry of the adder. When M =0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B.



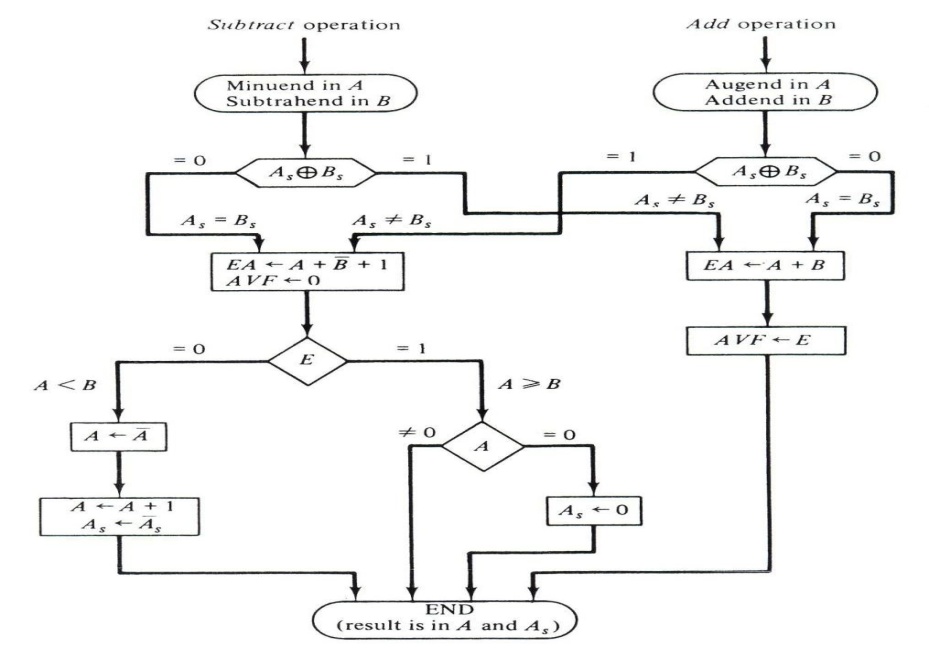
When M =1, the 1‘s complement of B is applied to the adder, the input carry is 1, and output S = A + + 1. This is equal to A plus the 2‘s complement of B, whi~~ch~~ is equivalent to the subtraction A –B.

The two signs As and Bs are compared by an exclusive–OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation , where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip – flop AVF.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2‘s complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that 𝐴≥𝐵 and the number in A is the correct result. If this number is zero, the sign As must be made positive to avoid a negative zero. A 0 in E indicates that

𝐴<𝐵. For this case it is necessary to take the 2‘s complement of the value in A. This operation can be done with one microoperation 𝐴←Ā+ 1. However, we assume that the A register has circuits for microoperations complement and increment, so the 2‘s complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no in As is required. When 𝐴<𝐵,the sign of the result is the complement of the original sign of A. It is then necessary to complement As to obtain the correct sign.

### Flow chart for add and subtract operations



The final result is found in register A and its sign in As. The value in AVF provides an overflow indication.

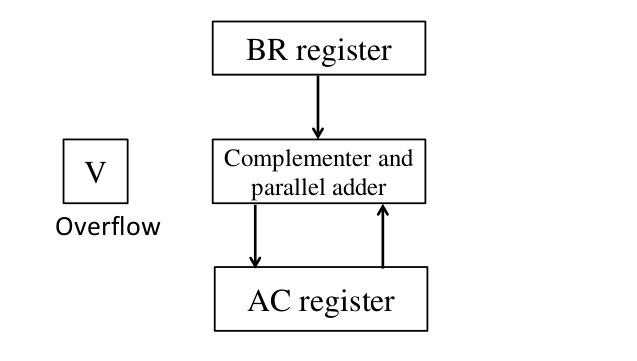
The leftmost bit of a binary number represents the sign bit : 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2‘s complement form. The addition of two numbers in signed -2‘s complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry – out of the sign – bit position is discarded. The subtraction consists of first taking the 2‘s complement of the subtrahend and then adding it to the minuend.

When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occurred. An overflow can be detected by inspecting the last two carried out of the addition. When the two carries are applied to an exclusive–OR gate, the overflow is detected when the output of the gate is equal to 1.

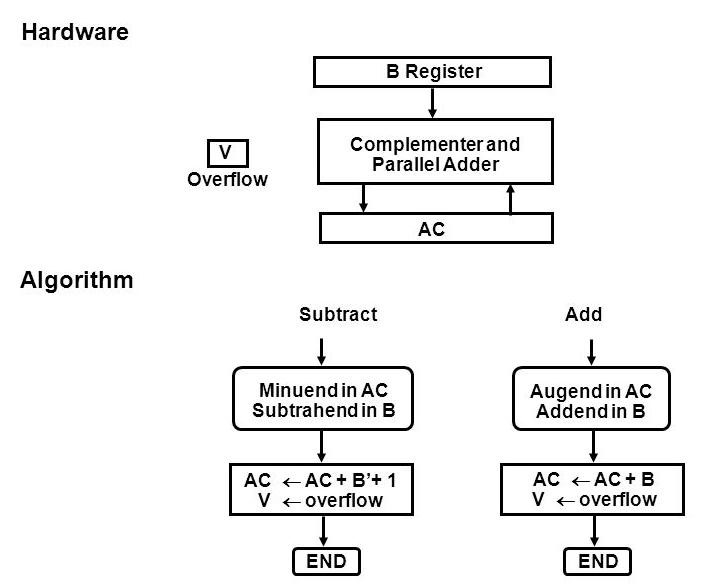
The register configuration for the hardware implementation is given below.The A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip – flop V is set to 1 if there is an overflow.

The algorithm for adding and subtracting two binary numbers in signed 2‘s complement representation is shown in the flowchart of Fig. 5.4. The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive–OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2‘s complement of BR. Taking the 2‘s complement of BR has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

### Hardware for signed – 2‘s complement addition and subtraction



Algorithm for adding and subtracting numbers in signed -2‘s complement representation.



Comparing this algorithm with its signed – magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed -2‘s complement representation. For this reason most computers adopt this representation over the more familiar signed–magnitude.

#### Multiplication Algorithms

Multiplication of two fixed–point binary numbers in signed – magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

23 10111 Multiplicand

19 x 10011 Multiplier

10111

10111

00000 +

00000

10111

437 110110101 Product

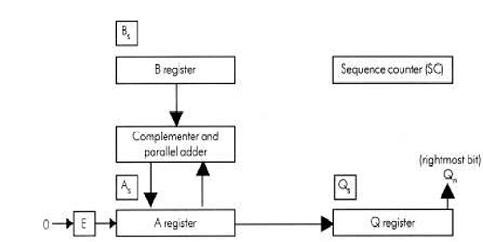
If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

These registers together with registers A and B are shown in the given Figure. The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

The multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. this shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 5.5. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip –flop in register Q, designated by Qn, will hold the bit of the multiplier, which must be inspected next.

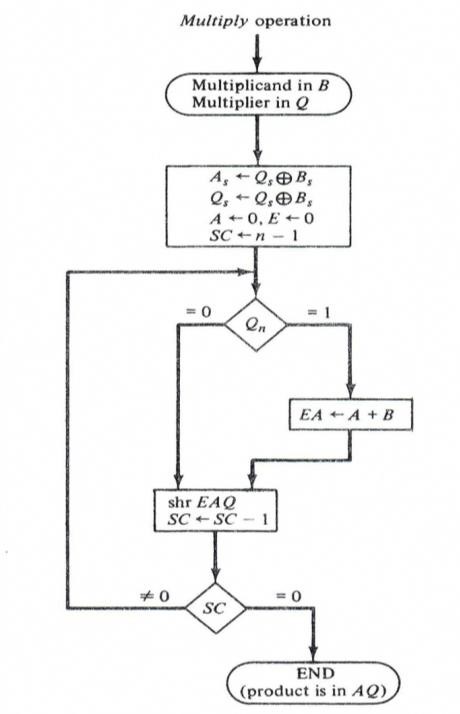
Hardware for multiply operation.



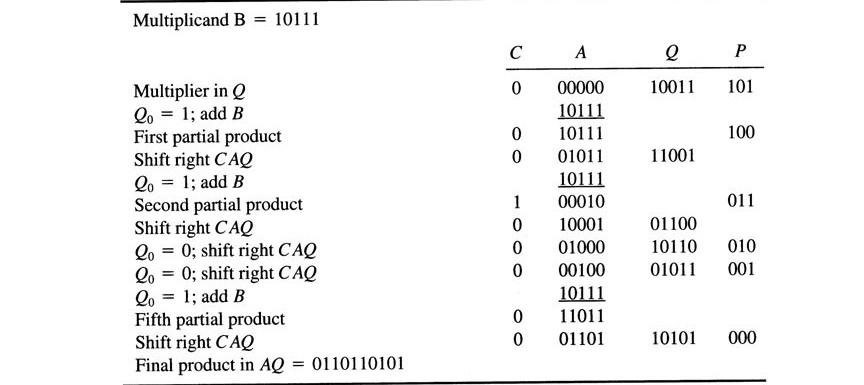
In the flowchart,Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs, respectively. The signs are compared , and both A and Q are set to correspond to the sign of the product since a double length product will be stored in registers A and Q. Registers A and E and cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist on n -1 bits.

The low – order bit of the multiplier in Qn is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC =0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

### Flowchart for multiply operation



### Table-2 Numerical example for binary multiplier



## Booth Multiplication Algorithm

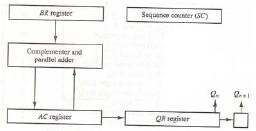
Booth algorithm gives a procedure for multiplying binary integers in signed -2‘s complement representation. It operates on the fact that strings of 0‘s in the multiplier require no addition but just shifting, and a string of 1‘s in the multiplier from bit weight 2k to 2m can be treated2k+1-2m. for example for binary number 001110(+14) when (k=3, m=1), the number can be represented as 2k+1– 2m = 24-21=16- 2=14. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier, can be done as M x 24 –M x 21. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

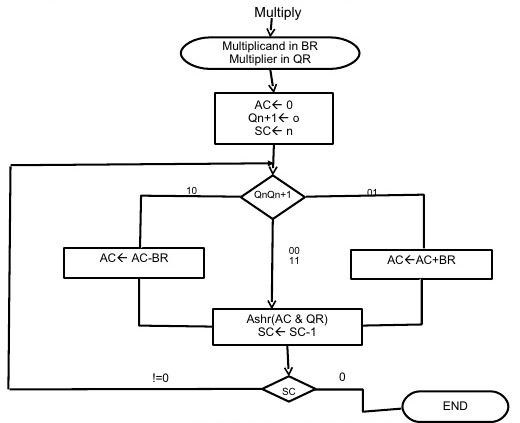
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1‘s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0‘s in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 5.7. We rename registers A, B and Q, as AC, BR, and QR, respectively. Qndesignates the least significant bit of the multiplier in register QR. An extra flip – flop Qn+1is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. 5.8. AC and the appendedbitQn+1 are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.

### Hardware for Booth algorithm



The two bits of the multiplier in Qn and Qn+1are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1‘s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0‘s has been encountered. this requires the addition of the multiplicand to the partial product in AC.



Booth algorithm for multiplication of signed -2‘s complement numbers

The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged . The sequence counter is decremented and the computational loop is repeated n times.

## TABLE 3 Example of Multiplication with Booth Algorithm

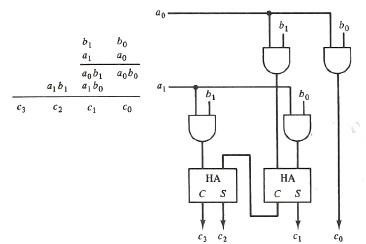
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **QnQn+1** | **BR=10111 BR+ 1=01001** | **AC** | **QR** | **Qn+1** | **SC** |
| 1 0  1 1  0 1   1. 0 2. 0 | Initial  Subtract BR  ashrashr  Add BR  ashrashr  Subtract BR  ashr | 00000  01001 | 10011  11001  01100  10110  01011  10101 | 0  1  1  0  0  1 | 101  100  011  010  001  000 |
| **01001**  00100  00010  10111 |
| **11001**  11100  11110  01001 |
| **00111**  00011 |

## Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro operations. The multiplication of two binary numbers can be done with one micro operation by means of a combinational circuit that forms the product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2 – bit numbers as shown in Fig.The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3c2c1c0. The first partial product is formed by multiplying a0 by b1b0. The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left. The two partial products are added with two half – adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full–adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the product does not have to go through an adder since it is formed by the output of the first AND gate.

### Figure – bit by 2- bit array multiplier



#### Division Algorithms

Division of two fixed – point binary numbers in signed – magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Fig. 5.10. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bits of A and compare this number with B. The 6 bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

In a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2‘s complement of B. The information about the relative magnitudes is then available from the end – carry.

The hardware for implementing the division operation is identical to the

multiplication operation.. Register EAQ is now shifted to the left with 0 inserted into Qnand the previous value of E lost. The numerical example is repeated in figure.

Figure Example of binary division Divisor:

B=10001

### 10001)0111000000(11010

01110

011100 -10001

\_\_\_\_\_\_\_

-010110

--10001

\_\_\_\_\_\_\_

--001010

---010100

----10001

\_\_\_\_\_\_\_\_

----000110 -----00110

Figure 5.11 Example of binary division with

digital hardware

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | E | A | Q | SC |
| Dividend :  ShlEAQ addB+1  E=1  Set Qn =1  ShlEAQ  Add B +1  E=1  Set Qn =1  ShlEAQ  Add B+1  E=0; leave Qn=0  Add B  Restore remainder  ShlEAQ  Add B +1  E=1  Set Qn=1  ShlEAQ  Add B+1  E=0; leave Qn=0  Add B  Restore remainder  Neglect E  Remainder in A  Quotient in Q |  | 0  1  1  0  1  1  0  0  1  0  1  1  0  0  1 | 01110  11100  01111  01011  01011  01010  01111  00101  00101  01010 01111  11001  10001  01010  10100  01111  00011  00011  00110 01111  10101 10001  00110  00110 | 00000 00000  00001 00010  00011  00110  00110    01100  01101  11010  11010  11010  11010 | 5  4  3  2  1  0 |

The divisor is stored in register B and the double length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding the 2‘s complement value. The information about the relative magnitude is available in E. If E=1 it signifies that A≥B. a quotient bit 1 is inserted into Qnand the partial reminder is shifted to the left to repeat the process. If E=0 it signifies that A<B so the quotient in Qn remains 0. The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. While the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

## Divide Overflow

The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.

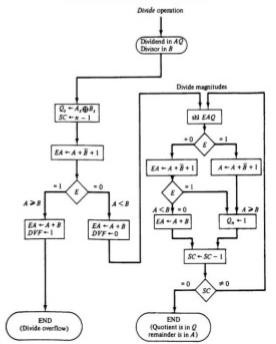
When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide–overflow condition occurs if the high order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip–flop is set. We will call it a divide–overflow flip–flop and label it DVF.

The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow. In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a divide stop. Stopping the operation of the computer is not recommended because it is time consuming. The procedure in most computers is to provide an interrupt request when DVF is set. The interrupt causes the computers is to provide an interrupt request when DVF is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. The best way to avoid a divide overflow is to use floating point data.

## Hardware Algorithm

The hardware divide algorithm is shown in the flowchart of Fig. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words in n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

### Figure -Flowchart for divide operation



A divide – overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If A B, the divide-overflow flip–flop DVF is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in AQ to the left with the high–order bit shifted into E. If the bit shifted into E is 1, we know that EA> B because EA consists of a 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Qn for the quotient bit. Since register A is missing the high – order bit of the dividend (which is in E), its value is EA –2n-1. Adding to this value the 2‘s complement of B results in

(EA -2n-1) + (2n-1 –B) =EA –B

The carry from this addition is not transferred to E if we want E to remain a1.

If the shift – left operation inserts a 0 into E, the divisor is subtracted by adding its 2‘s complement value and the carry is transferred into E. If E = 1, it signifies that A  B; therefore, Qn is set to 1. If E =0, it signifies that A < B and the original number is restored by adding B to A. In the latter case we leave a 0 in Qn (0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After n- 1 times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Qs and the sign of the remainder in As is the same as the original sign of the dividend.

## Other Algorithms

The hardware method just described is called the restoring method. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference. Two other methods are available for dividing numbers, the comparison method and the non restoring method. In the comparison method A and B are compared prior to the subtraction operation. Then If A  B, B is subtracted from A. If A < B nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end–carry out of the parallel–adder prior to its transfer to register E.

In the non restoring method, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. In the non restoring method, B is subtracted if the previous value of Qn was a 1, but B is added if the previous value of Qn was a 0 and no restoring of the partial remainder is required. This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.