Arithmetic

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 thecarry-out is 1.

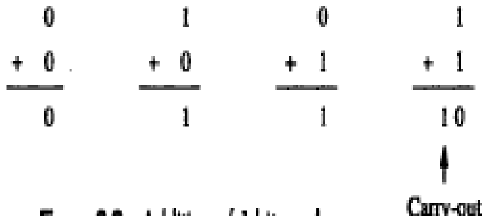


Figure 2.2 Addition of 1-bit numbers.

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 (Figure 1.6).
  - **Rule 1:**
    - ➢ **To Add** two numbers, add their n-bits and ignore the carry-out signal from the MSB position.
    - ➢ Result will be algebraically correct, if it lies in the range -2^{n-1} to $+2^{n-1}-1$.
  - **Rule 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 overflowis said to occur.
- To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit asmany 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 result of an arithmetic operation is outside the representable-range, an **arithmetic overflow** is said to occur.
- For example: If we add two numbers +7 and +4, then the output sum S is 1011(□0111+0100),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 1.6 2's-complement Add and Subtract operations.

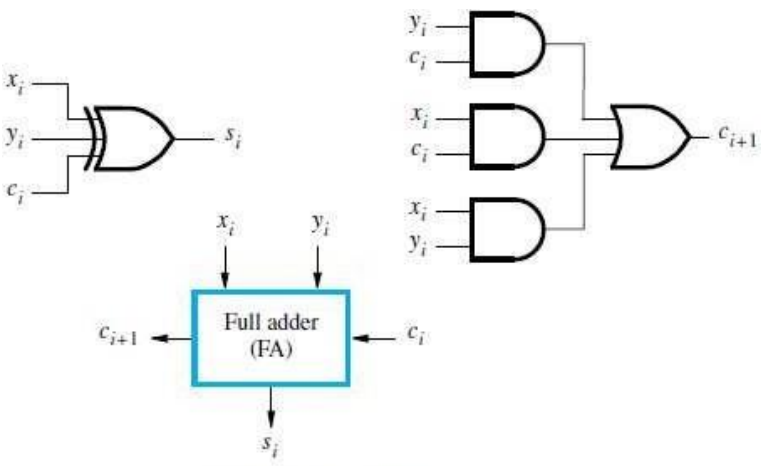
ADDITION & SUBTRACTION OF SIGNED NUMBERSn-BIT RIPPLE CARRY ADDER

- 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 ripplecarry adder (Figure 9.1).

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

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

$$c_{i+1} = y_ic_i + x_ic_i + x_iy_i$$



(a) Logic for a single stage

Example:

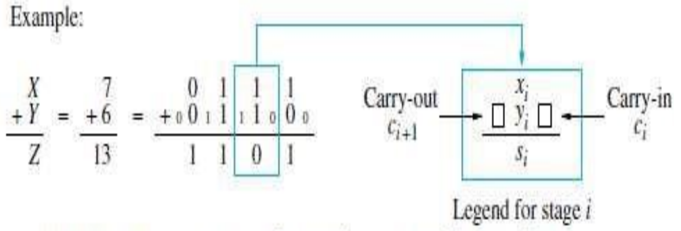
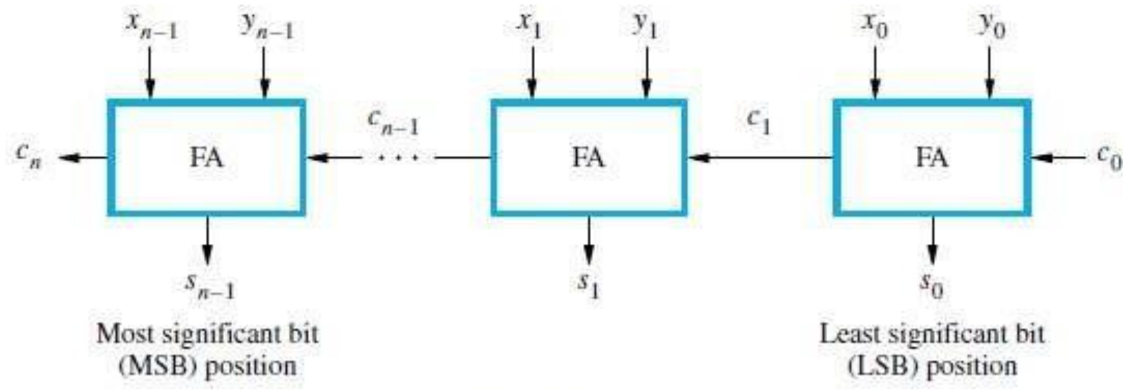
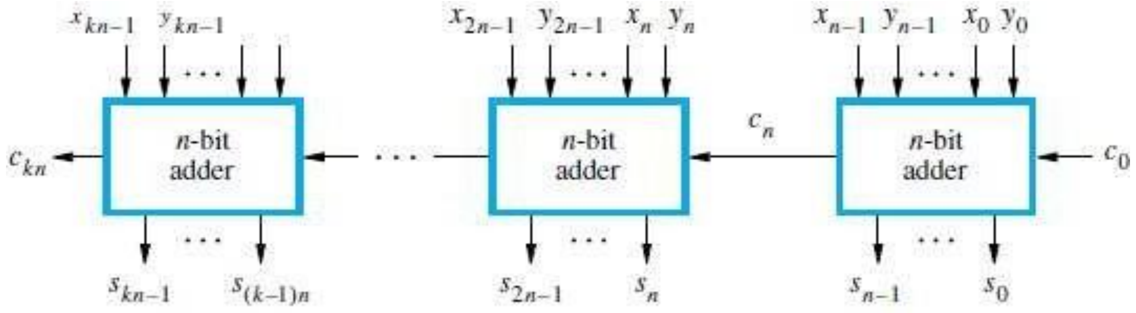


Figure 9.1 Logic specification for a stage of binary addition.



(b) An *n*-bit ripple-carry adder



(c) Cascade of *k* *n*-bit adders

Figure 9.2 Logic for addition of binary numbers.

ADDITION/SUBTRACTION LOGIC UNIT

- The n-bit adder can be used to add 2's complement numbers X and Y (Figure 9.3).
- **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 the2'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.

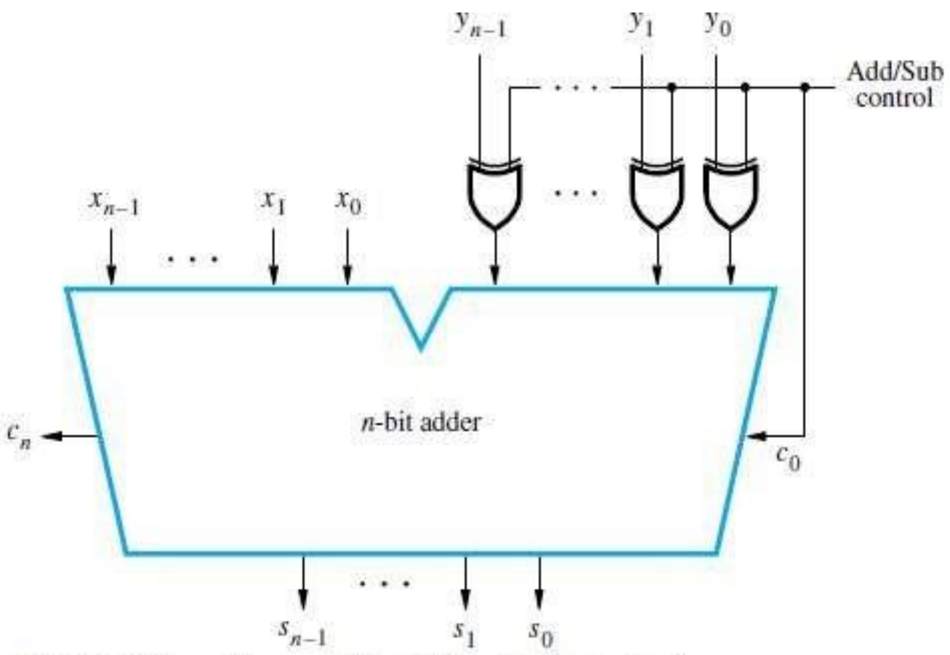
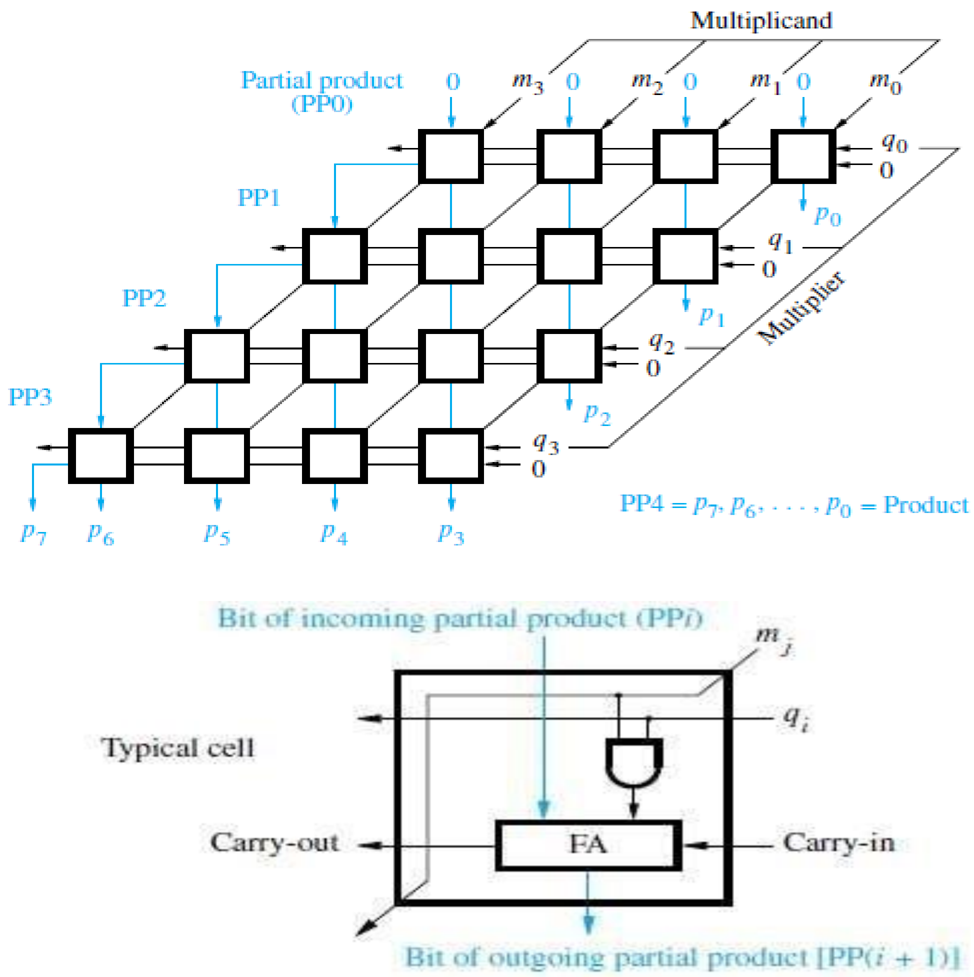


Figure 9.3 Binary addition/subtraction logic circuit.

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

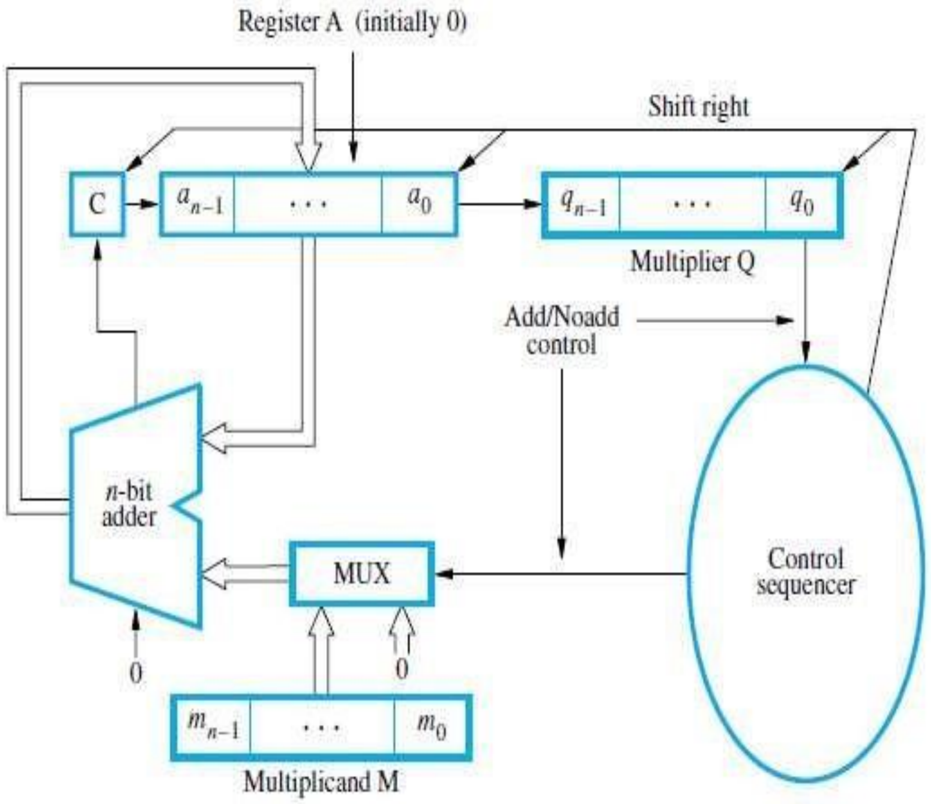
Figure 9.6 Array multiplication of unsigned binary operands.

ARRAY MULTIPLICATION

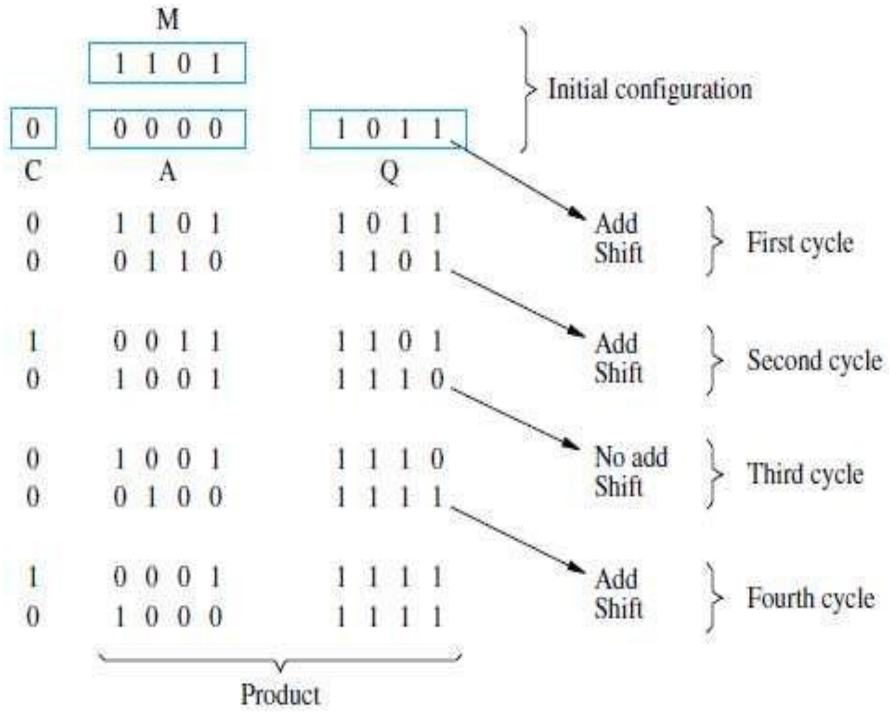
- The main component in each cell is a full adder(FA)..
  The AND gate in each cell determines whether a multiplicand bit mj, is added to the incoming partial-product bit, based on the value of the multiplier bit qi (Figure9.6).

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold PPi(partial product)
  while the multiplier bit qi generates the signal Add/Noadd.
- The carry-out from the adder is stored in flip-flop C (Figure 9.7).
- 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 q0=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position.If q0=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 andthe low-order half is held in register Q.



(a) Register configuration



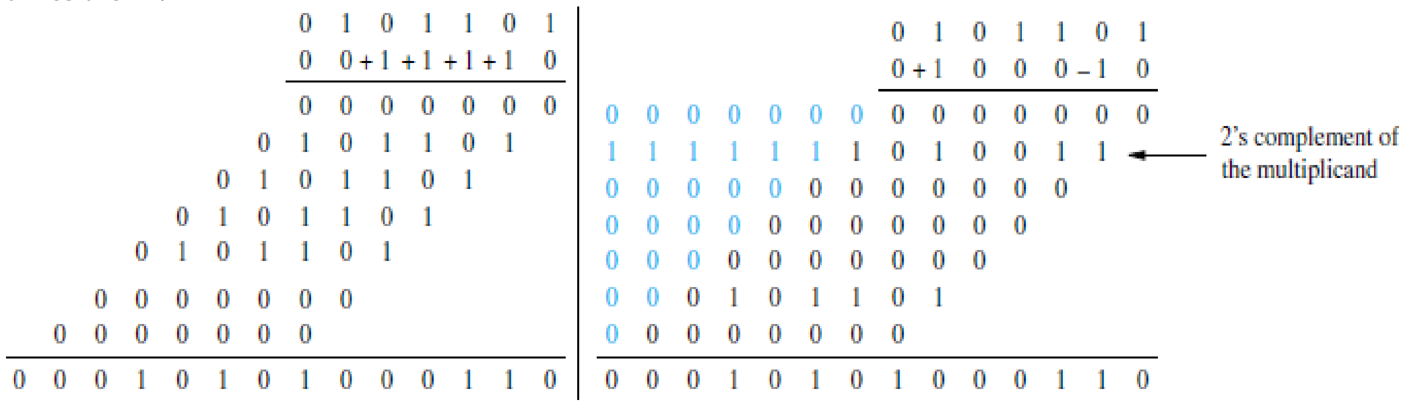
(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

SIGNED OPERAND MULTIPLICATIONBOOTH ALGORITHM

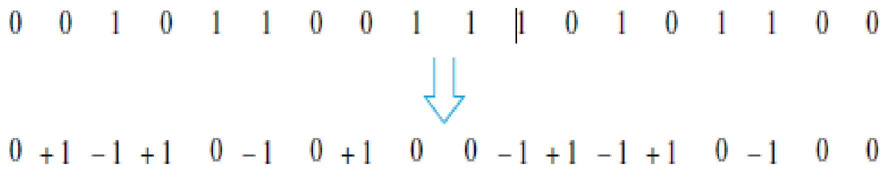
- This algorithm
  - → generates a 2n-bit product
  - → treats both positive & negative 2's-complement n-bit operands uniformly(Figure 9.9-9.12).
- Attractive feature: This algorithm achieves some efficiency in the number of addition required whenthe multiplier has a few large blocks of 1s.
- This algorithm 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 as010000 (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.



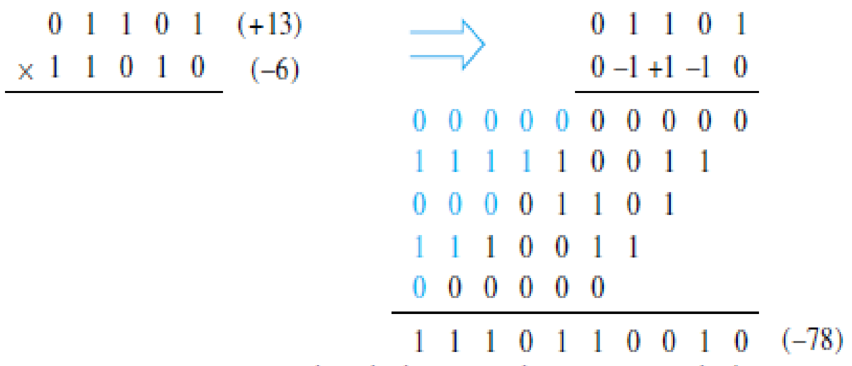
```
                       0  1  0  1  1  0  1   |                     0  1  0  1  1  0  1
                       0  0 +1 +1 +1 +1  0   |                     0 +1  0  0  0 -1  0
                      ---------------------  |                    ---------------------
                       0  0  0  0 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

Figure 9.9 Normal and Booth multiplication schemes.



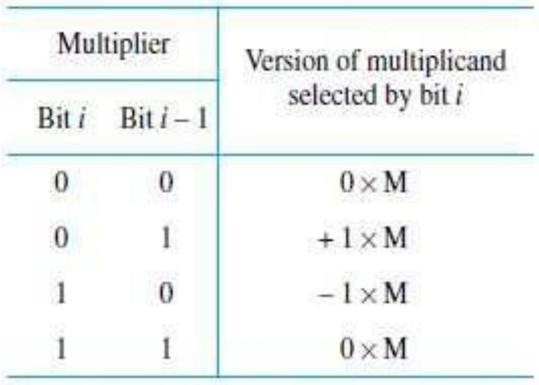
```
 0   0   1   0   1   1   0   0   1   1  |1   0   1   0   1   1   0   0

                                 ⇓

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

Figure 9.10 Booth recoding of a multiplier.



```
    0  1  1  0  1   (+13)                  0  1  1  0  1
 ×  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 9.11 Booth multiplication with a negative multiplier.

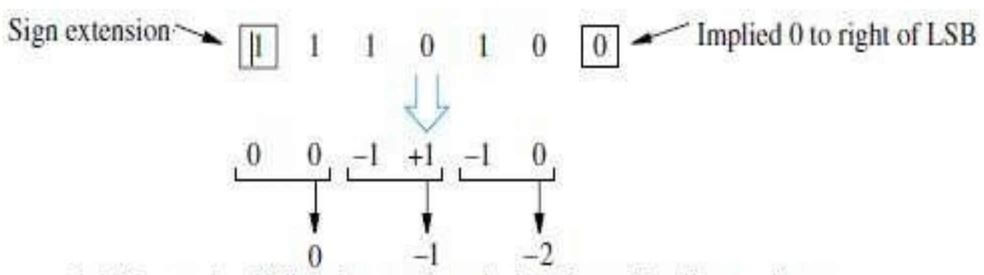


Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i* − 1	
0	0	0 × M
0	1	+ 1 × M
1	0	− 1 × M
1	1	0 × M

Figure 9.12 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 9.14 & 9.15).
- 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	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

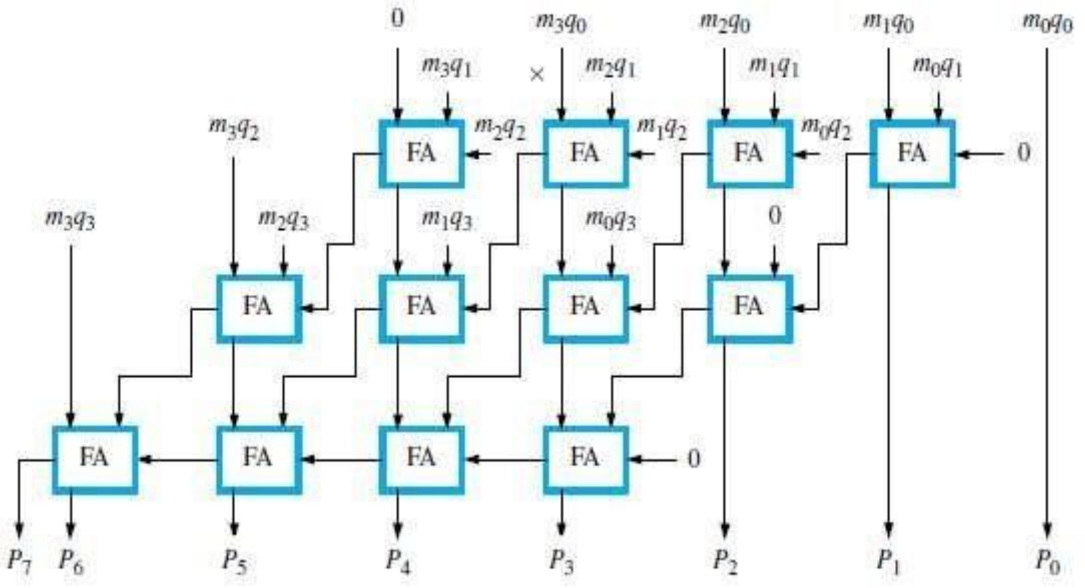
Figure 9.14 Multiplier bit-pair recoding.

```
                    0  1  1  0  1   (+13)
                 ×  1  1  0  1  0   (−6)
                 ----------------
                         ⇓
                    0  1  1  0  1
                    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)
                         ⇓
                    0  1  1  0  1
                    0    −1    −2
                 ----------------
 1  1  1  1  1  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
```

Figure 9.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

- Consider the array for 4*4 multiplication. (Figure 9.16 & 9.18).
- Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.



(b) Carry-save array

Figure 9.16 carry-save arrays for a 4 × 4 multiplier.

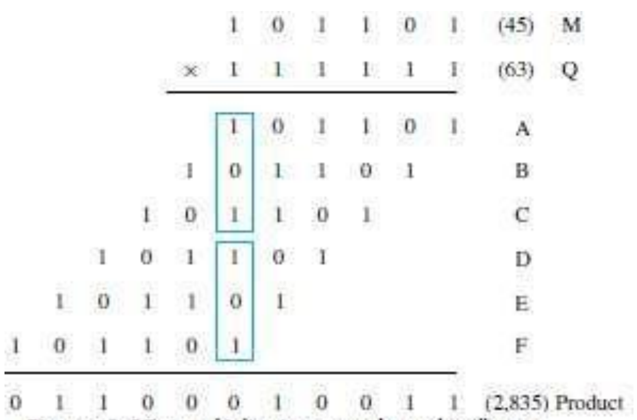
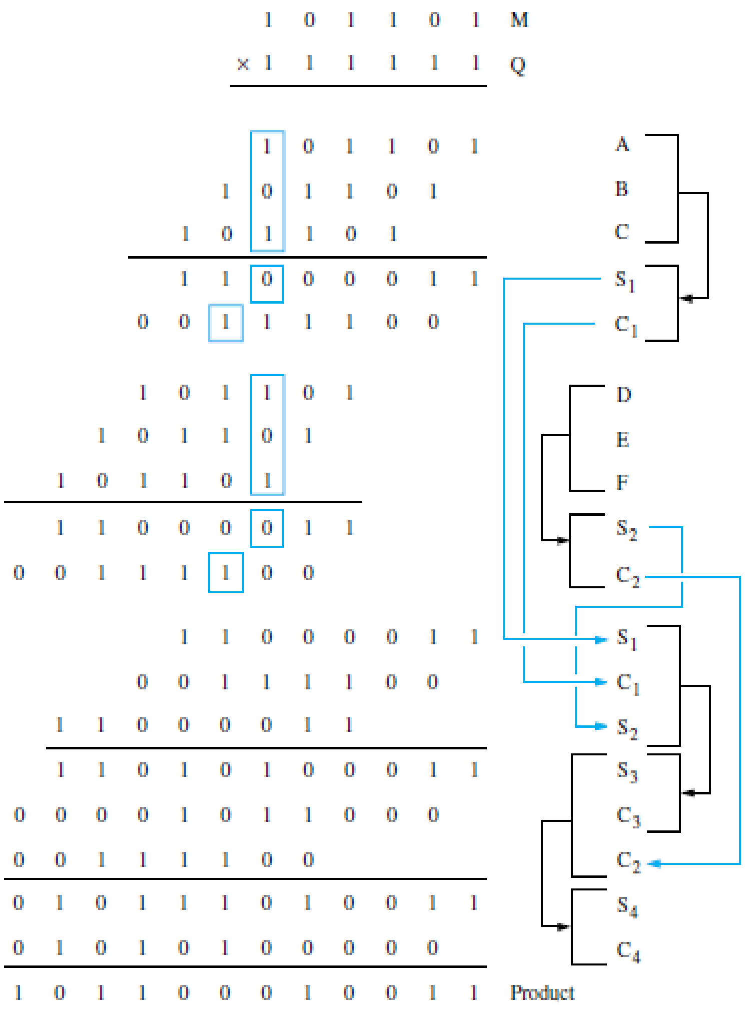


Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.



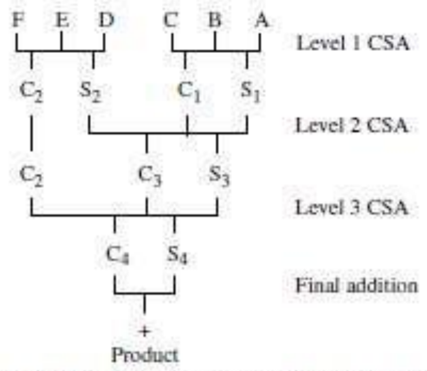


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

- The full adder is input with three partial bit products in the first row.
- Multiplication requires the addition of several summands.
- CSA speeds up the addition process.
- Consider the array for 4x4 multiplication shown in fig 9.16.
- First row consisting of just the AND gates that implement the bit products m3q0, m2q0, m1q0 and m0q0.
- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
- Consider the addition of many summands in fig 9.18.
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a RCA or CLA to produce the desired product.
- When the number of summands is large, the time saved is proportionally much greater.
- Delay: AND gate + 2 gate/CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay,array 6x6 require 6(n-1)-1 = 29 gate Delay.
- In general, CSA takes 1.7 *log*2k-1.7 levels of CSA to reduce k summands.

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 9.21).
- After division operation, the n-bit quotient is in register Q, and
  the remainder is in register A.

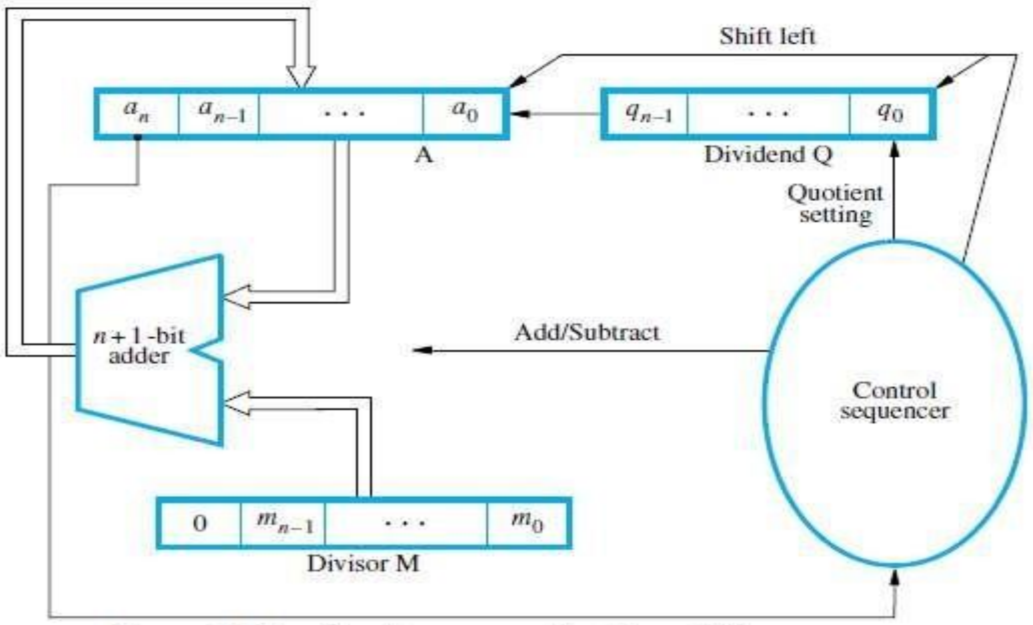


Figure 9.23 Circuit arrangement for binary division.

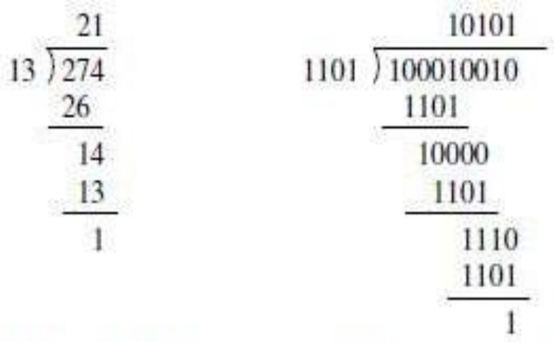


Figure 9.22 Longhand division examples.

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 9.23).

  ii) Now, if the sign of A is 0, set q0 to 1; otherwise set q0 to 0.Step 2: If the sign of A is 1, add M to A (restore).

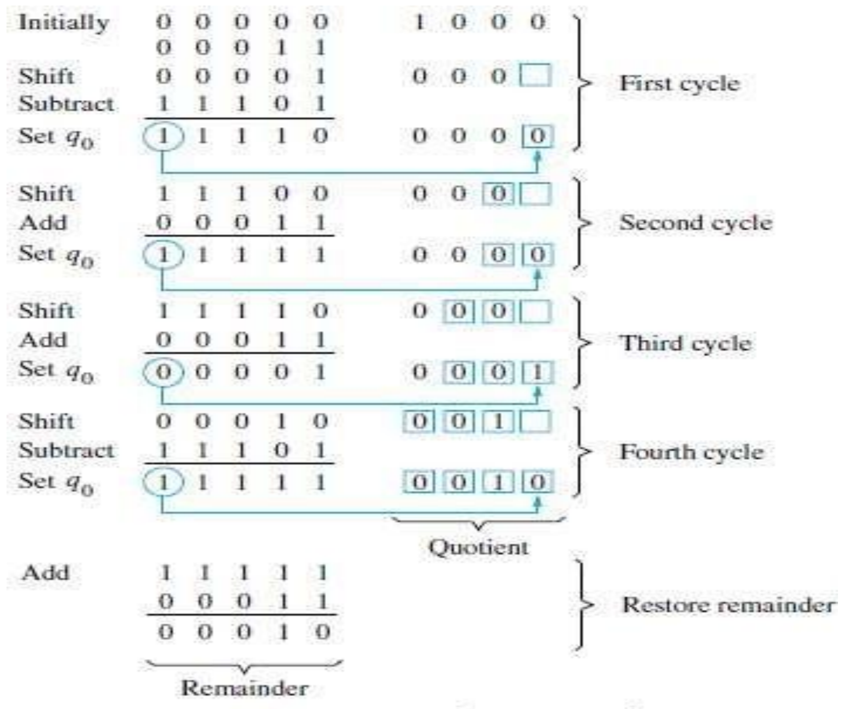


Figure 9.25 A non-restoring division example.

RESTORING DIVISION

- Procedure: Do the following n times
  1) Shift A and Q left one binary position (Figure 9.22).
  2) Subtract M from A, and place the answer back in A
  3) If the sign of A is 1, set q0 to 0 and add M back to A(restore A).If the sign of A is 0, set q0 to 1 and no restoring done.

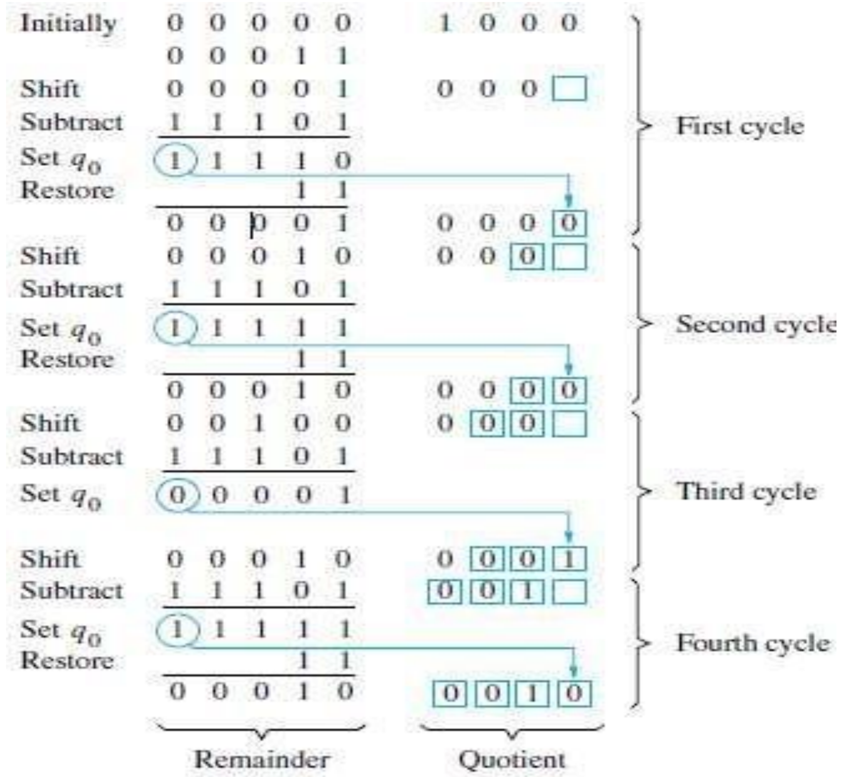
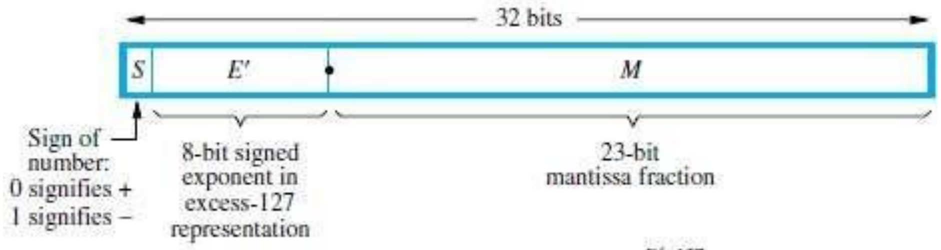


Figure 9.24 A restoring division example.

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

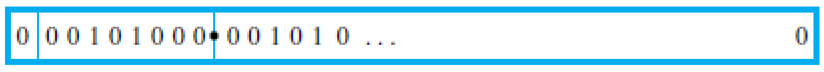
- Single precision representation occupies a single 32-bit word.
  The scale factor has a range of 2^{-126} to 2^{+127} (which is approximately equal to 10^{+38}).
- The 32 bit word is divided into 3 fields: sign(1 bit), exponent(8 bits) and mantissa(23 bits).
- Signed exponent=E.
  Unsigned exponent 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 mantissais always equal to 1. (M represents fractional-part).
- The 24-bit mantissa provides a precision equivalent to about 7 decimal-digits (Figure 9.24).
- Double precision representation occupies a single 64-bit word. And E' is in the range 1<E'<2046.


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 $= \pm 1.M \times 2^{E'-127}$

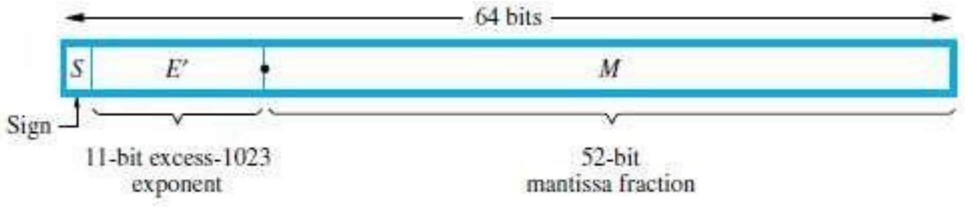
(a) Single precision

- The 53-bit mantissa provides a precision equivalent to about 16 decimal-digits.



Value represented $= 1.001010\ldots0 \times 2^{-87}$

(b) Example of a single-precision number



Value represented $= \pm 1.M \times 2^{E'-1023}$

(c) Double precision

Figure 9.26 IEEE standard floating-point formats.

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 stepsequal 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 9.26).
  If sign=0, then EA>EB and mantissas MA & MB are sent straight through SWAP network. If sign=1, then EA<EB 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 EA if EA>EB or EB if EA<EB
- 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.

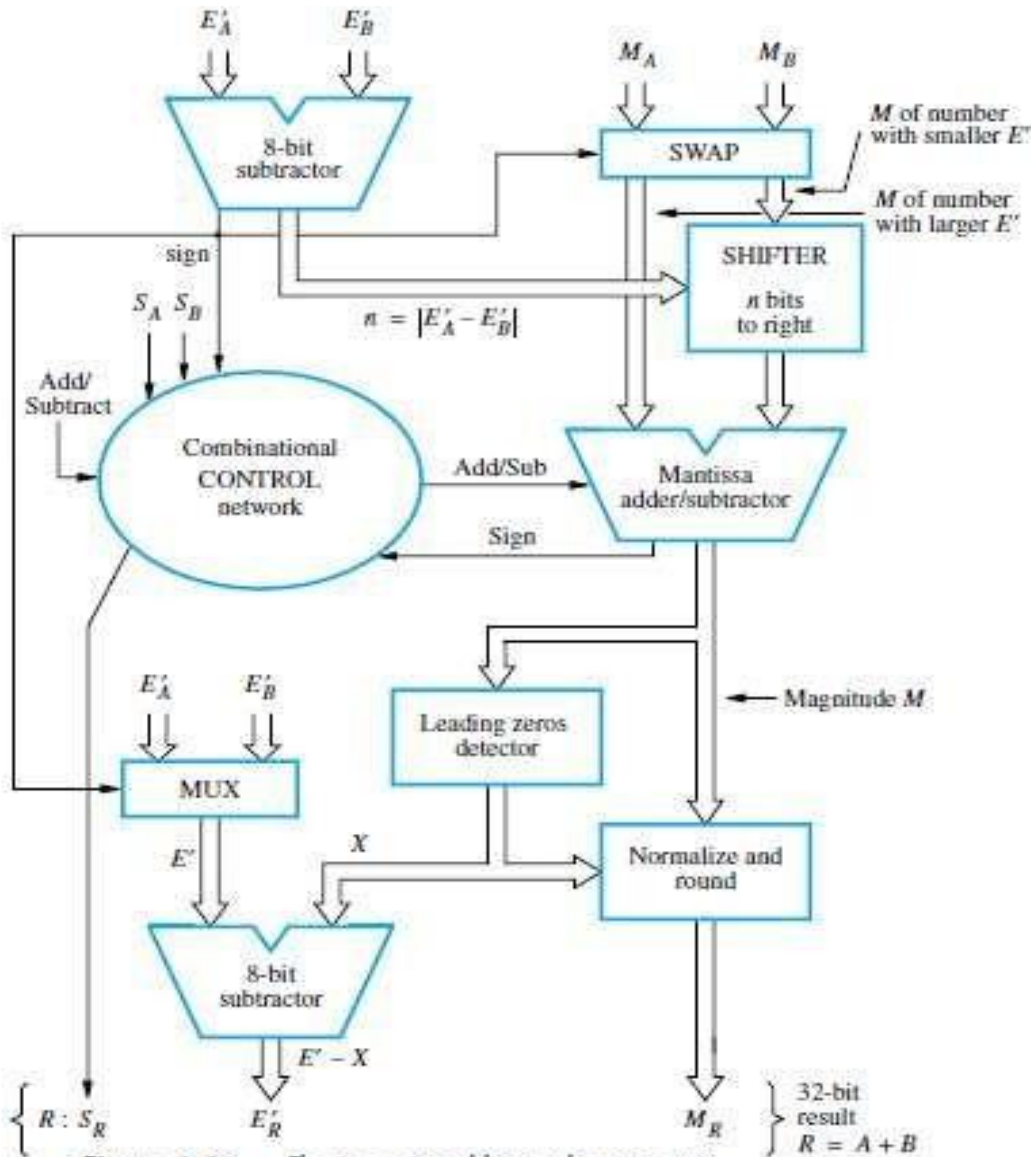


Figure 9.28 Floating-point addition-subtraction unit.

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 is written as:
     IR [[PC]]
  2) Increment PC by 4.
     PC [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**.

- The operation specified by an instruction can be carried out by performing one or more of thefollowing actions:
  1) Read the contents of a given memory-location and load them into a register.
  2) Read data from one or more registers.
  3) Perform an arithmetic or logic operation and place the result into a register.
  4) Store data from a register into a given memory-location.
- The hardware-components needed to perform these actions are shown in Figure 5.1.

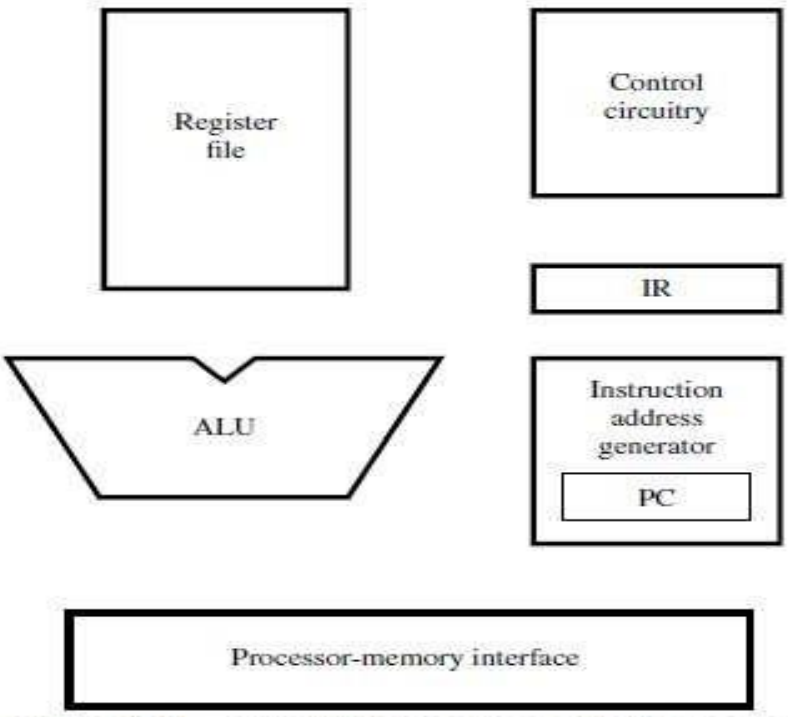


Figure 5.1 Main hardware components of a processor.

SINGLE BUS ORGANIZATION

- ALU and all the registers are interconnected via a **Single Common Bus** (Figure 7.1).
- Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR & MAR respectively. (MDR Memory Data Register, MAR Memory Address Register).
- **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; MAR‟s output is connected to external-bus.
- **Instruction Decoder & Control Unit** is responsible for
  → issuing the control-signals to all the units inside the processor.
  → implementing the actions specified by the instruction (loaded in the IR).
- Register R0 through R(n-1) are the **Processor Registers**.
  The programmer can access these registers for general-purpose use.
- Only processor can access 3 registers **Y**, **Z** & **Temp** for temporary storage during program-execution.
  The programmer cannot access these 3 registers.
- In **ALU**, 1) „A‟ input gets the operand from the output of the multiplexer(MUX).
  2) „B‟ input gets the operand directly from the processor-bus.

- There are 2 options provided for „A" input of the ALU.
- MUX is used to select one of the 2 inputs.
- **MUX** selects either
  - → output of Y or
  - → constant-value 4(which is used to increment PC content).

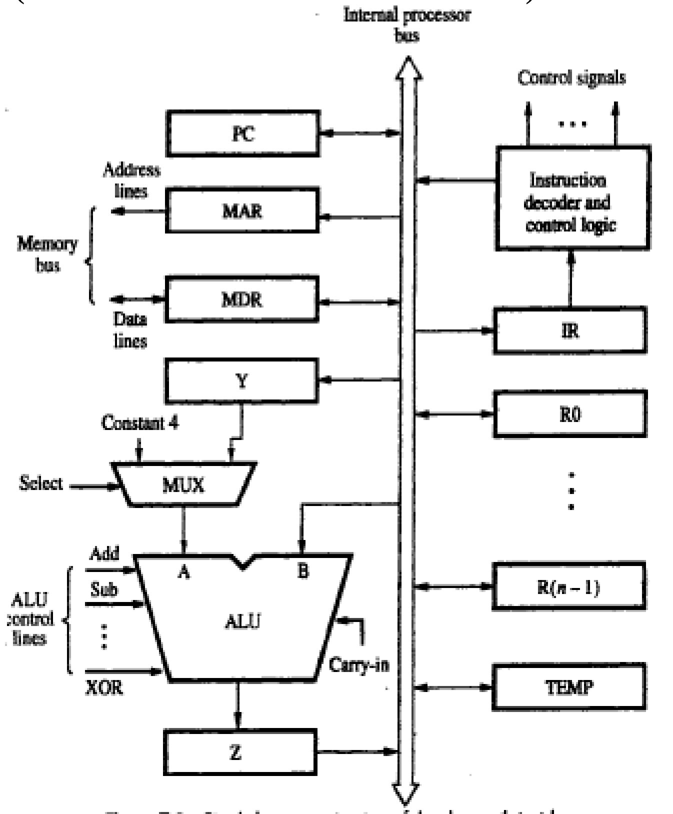


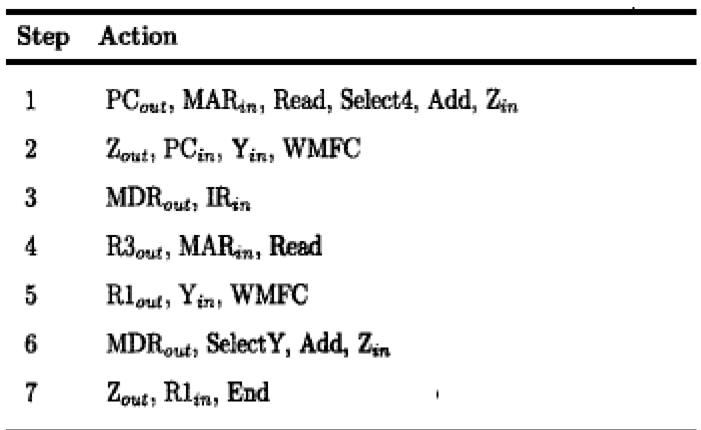
Figure 7.1 Single-bus organization of the datapath inside a processor.

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

Solution: Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel.

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.



Step	Action
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

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

- 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. This completes the PC increment operation andPC will now point to next instruction.

  Step3--> Fetched instruction is moved into MDR and then to IR.The step 1 through 3 constitutes the **Fetch Phase**.

  At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7.

  The step 4 through 7 constitutes the **Execution Phase**.

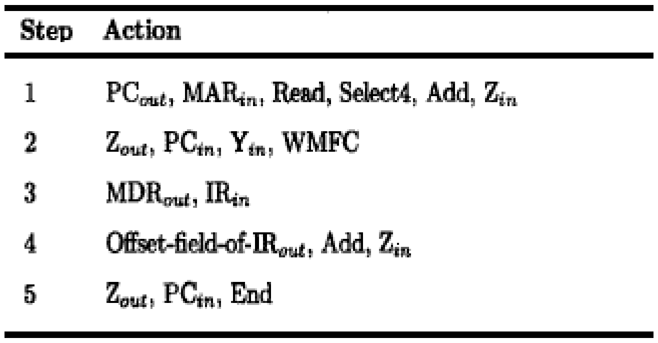
  Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued. Step5--> Contents of R1 are transferred to Y to prepare for addition.

  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 instructionfetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

- Control sequence for an **unconditional branch instruction** is as follows:



Step	Action
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

Figure 7.7 Control sequence for an unconditional Branch instruction.

- Instruction execution proceeds as follows:

  Step 1-3--> The processing starts & the fetch phase ends in step3.

  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.

  Step 5--> the result, which is the branch-address, is loaded into the PC.
- The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address.
- The branch target address is usually obtained by adding the offset in the contents of PC.
- The offset X is usually the difference between the branch target-address and the addressimmediately following the branch instruction.
- In case of **conditional branch**,

  we have to check the status of the condition-codes before loading a new value into the PC.e.g.:

  Offset-field-of-IRout, Add, Zin, If N=0 then End

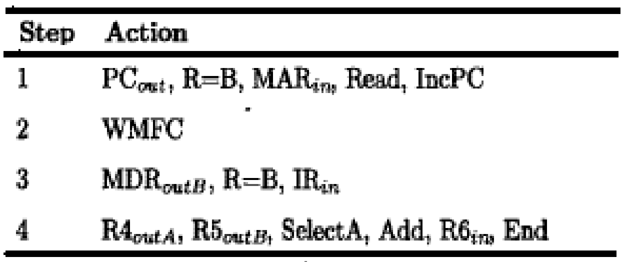
  If N=0, processor returns to step 1 immediately after step 4.If N=1, step 5 is performed to load a new value into PC.

MULTIPLE BUS ORGANIZATION

- **Disadvantage of Single-bus organization:** Only one data-word can be transferred over the bus ina clock cycle. This increases the steps required to complete the execution of the instruction

Solution: To reduce the number of steps, most processors provide multiple internal-paths. Multiplepaths enable several transfers to take place in parallel.

- As shown in fig 7.8, three buses can be used to connect registers and the ALU of the processor.
- All general-purpose registers are grouped into a single block called the **Register File**.
- 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 sameclock-cycle.
- Buses A and B are used to transfer source-operands to A & B inputs of ALU.
- The result is transferred to destination over bus C.



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

Figure 7.9 Control sequence for the instruction Add R4,R5,R6

- **Incrementer Unit** is used to increment PC by 4.
- Instruction execution proceeds as follows:

Step 1--> Contents of PC are

→ passed through ALU using R=B control-signal &

→ loaded into MAR to start 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 takes place in a single step.

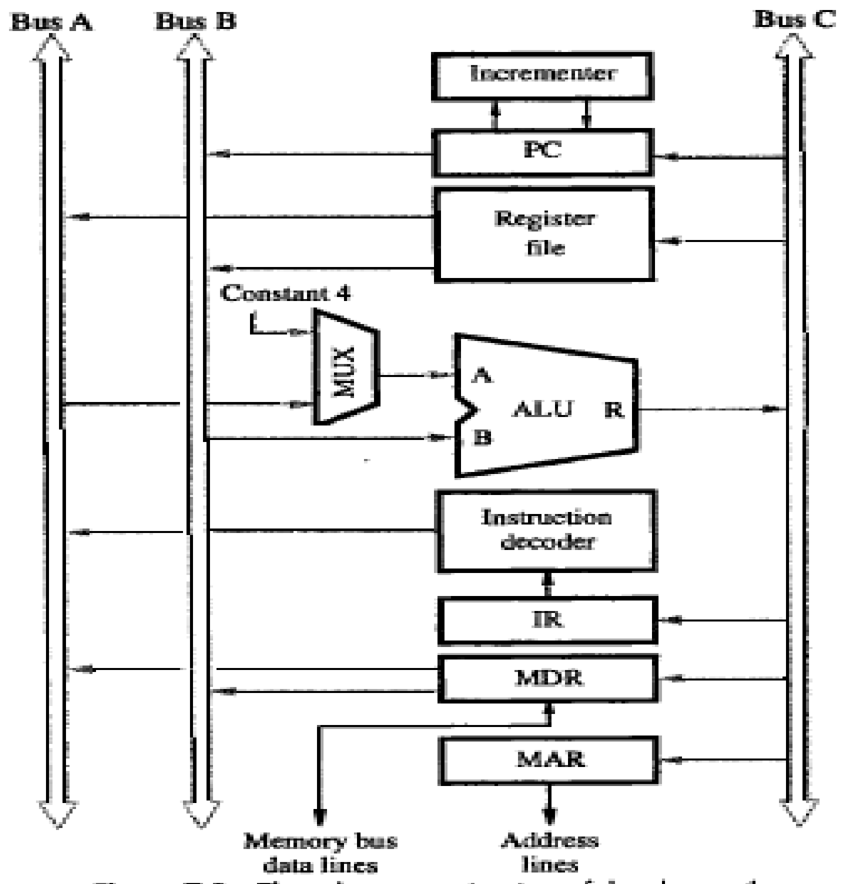


Figure 7.8 Three-bus organization of the datapath.

COMPLETE PROCESSOR

- This has separate processing-units to deal with integer data and floating-point data.
  Integer Unit→ To process integer data. (Figure 7.14).

  Floating Unit→ To process floating –point data.

- **Data-Cache** is inserted between these processing-units & main-memory.The integer and floating unit gets data from data cache.
- **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 concurrentoperations.
- The 80486 processor has 8-kbytes single cache for both instruction and data.
  Whereas the Pentium processor has two separate 8 kbytes caches for instruction and data.

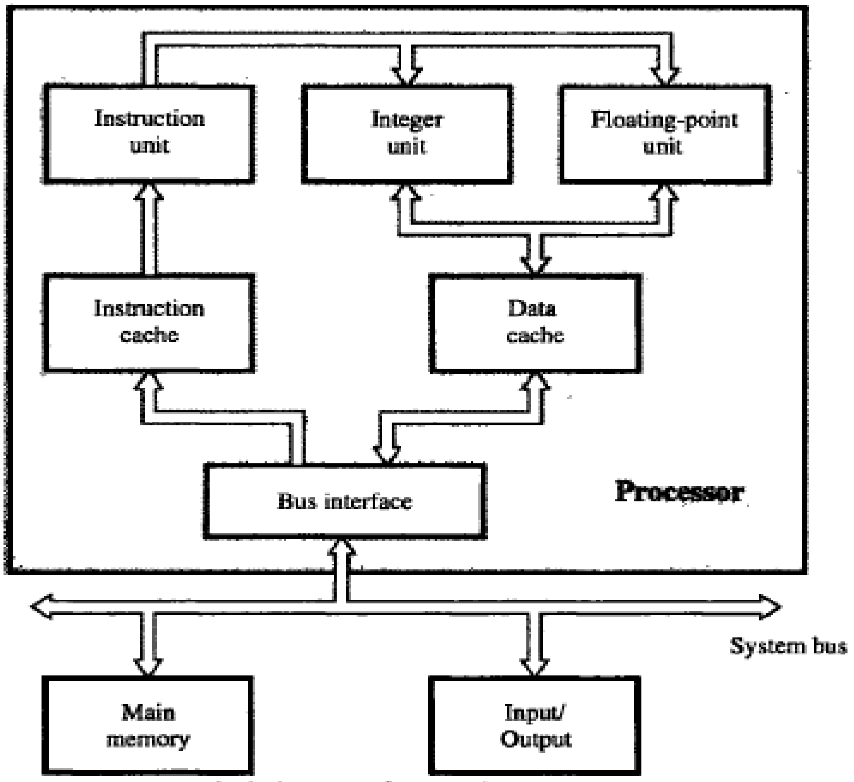


Figure 7.14 Block diagram of a complete processor.

Note:

To execute instructions, the processor must have some means of generating the control-signals. Thereare two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

HARDWIRED CONTROL

- Hardwired control is a method of control unit design (Figure 7.11).
- The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc.
- **Decoder/Encoder Block** is a combinational-circuit that generates required control-outputsdepending on state of all its inputs.
- Instruction Decoder
  - ➢ It decodes the instruction loaded in the IR.
  - ➢ If IR is an 8 bit register, then instruction decoder generates 2^8(256 lines); one for each instruction.
  - ➢ It consists of a separate output-lines INS1 through INSm for each machine instruction.
  - ➢ According to code in the IR, one of the output-lines INS1 through INSm is set to 1, and all

other lines are set to 0.

- **Step-Decoder** provides a separate signal line for each step in the control sequence.
- Encoder
  - ➢ It gets the input from instruction decoder, step decoder, external inputs and condition codes.
  - ➢ It uses all these inputs to generate individual control-signals: Yin, PCout, Add, End and so on.
  - ➢ For example (Figure 7.12), *Zin=T1+T6.ADD+T4.BR*

;This signal is asserted during time-slot T1 for all instructions.
during T6 for an Add instruction.
during T4 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.
- After execution of each instruction, **end** signal is generated. End signal resets step counter.
- Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence the name "**hardwired**".
- **Advantage**: Can operate at high speed.
- Disadvantages:
  1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.
  2) It is costly and difficult to design.
  3) The control unit is inflexible because it is difficult to change the design.

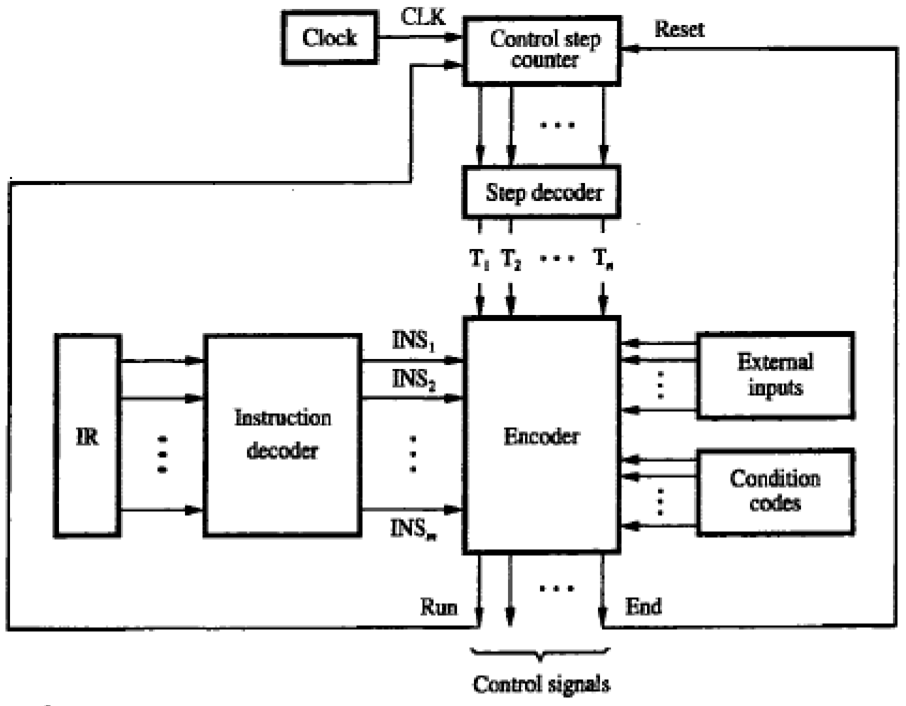


Figure 7.11 Separation of the decoding and encoding functions.

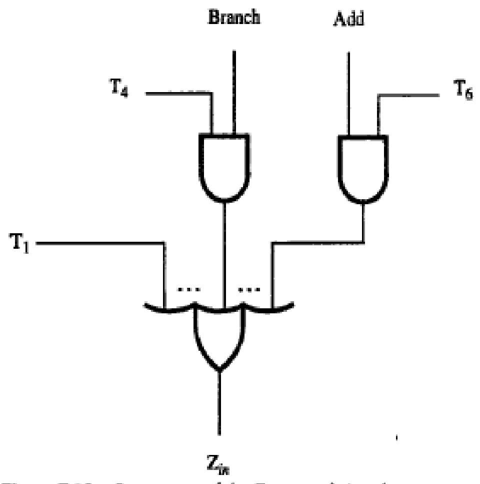


Figure 7.12 Generation of the Z_{in} control signal

MICROPROGRAMMED CONTROL

- Microprogramming is a method of control unit design (Figure 7.16).
- 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, PCin).
- Each of the control-steps in control sequence of an instruction defines a unique combination of 1s &0s in CW.
- Individual control-words in microroutine are referred to as **microinstructions** (Figure 7.15).
- 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 specialmemory called the **Control Store (CS)**.
- Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.
- **µPC** is used to read CWs sequentially from CS. (µPC Microprogram Counter).
- Every time new instruction is loaded into IR, o/p 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.

Advantages

- It simplifies the design of control unit. Thus it is both, cheaper and less error prone implement.
- Control functions are implemented in software rather than hardware.
- The design process is orderly and systematic.
- More flexible, can be changed to accommodate new system specifications or to correct the designerrors quickly and cheaply.
- Complex function such as floating point arithmetic can be realized efficiently.

Disadvantages

- A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.
- The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.

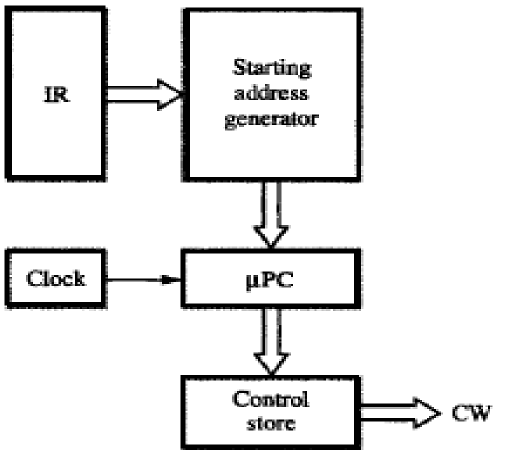


Figure 7.16 Basic organization of a microprogrammed control unit.

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	

Figure 7.15 An example of microinstructions for Figure 7.6.