Number Representation and Arithmetic Operations

1.4.1 Integers

Consider an n-bit vector : B = bn−1 . . . b1b0 where bi = 0 or 1 for $0 \leq i \leq n-1$.

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We need to represent both positive and negative numbers.

Three systems are used for representing suchnumbers:

- Sign-and-magnitude
- 1‟s-complement
- 2‟s-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for −3 is obtained by complementing each bit in the vector 0011 to yield 1100.

In the *2's-complement* system, forming the 2‟s-complement of an *n*-bit number is done by subtracting the number from 2^n. Hence, the 2‟s-complement of a number is obtained by adding 1 to the 1‟s-complement of that number.

There are distinct representations for +0 and −0 in both the sign-and magnitude and 1‟s-complement systems, but the 2‟s-complement system has only one representation for 0.

Addition of Unsigned Integers

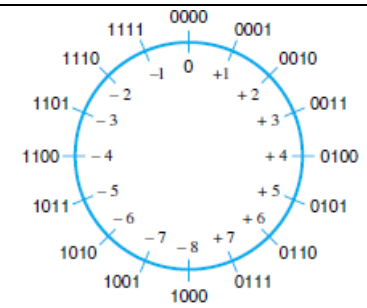
The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in* to the next bit pair to the left.

```
  0      1      0      1
+ 0    + 0    + 1    + 1
---    ---    ---    ---
  0      1      1     1 0
                        ↑
                    Carry-out
```

Addition and Subtraction of Signed Integers

The 2‟s-complement system is the most efficient method for performing addition and subtraction operations.

Unsigned integers mod *N* is a circle with the values 0 through *N* – 1. The decimal values 0 through 15 are represented by their 4-bit binary values 0000 through 1111.



(b) Mod 16 system for 2's-complement numbers

The operation (7 + 5) mod 16 yields the value 12. To perform this operation graphically, locate 7 (0111) on the outside of the circle and then move 5 units in the clockwise direction to arrive at the answer 12 (1100).
Similarly, (9 + 14) mod 16 = 7; this is modeled on the circle by locating 9 (1001) and moving 14 units in the clockwise direction past the zero position to arrive at the answer 7 (0111).
Apply the mod 16 addition technique to the example of adding +7 to −3. The 2"s-complement representation for these numbers is 0111 and 1101, respectively.

To *add* two numbers, add their *n*-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2"s-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2"s-complement of Y, then add it to X using the *add* rule. Again, the result will be the algebraically correct value in 2"s-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

	0010	(+2)	(b) 0100	(+4)
	+ 0011	(+3)	+ 1010	(−6)
	0101	(+5)	1110	(−2)
	1011	(−5)	(d) 0111	(+7)
	+ 1110	(−2)	+ 1101	(−3)
	1001	(−7)	0100	(+4)

Floating-Point Numbers

If we use a full word in a 32-bit word length computer to represent a signed integer in 2"s-complement representation, the range of values that can be represented is -2^{31} to $+2^{31} - 1$.
Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in the representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 3.7291×10^{-27}, -1.0341×10^{2}, -7.3000×10^{-14}. these numbers have been given to 5 *significant digits* of precision.

A binary floating-point number can be represented by:
- a sign for the number
- some significant bits
- a signed scale factor exponent for an implied base of 2

Character Representation

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange). Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes. It is convenient to use an 8-bit *byte* to represent and store a character.
The code occupies the low-order seven bits. The high-order bit is usually set to 0. This facilitates sorting operations on alphabetic and numeric data.
The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system.
This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

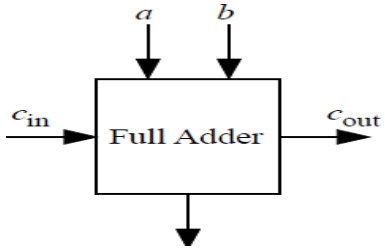


Figure 1: One-bit full adder.

A one-bit full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs(a,b, and cin) and two outputs(s, and cout)___as illustrated in Figure 1.

Table 1: Full adder truth table.

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The truth table of 1-bit full adder is given in the table

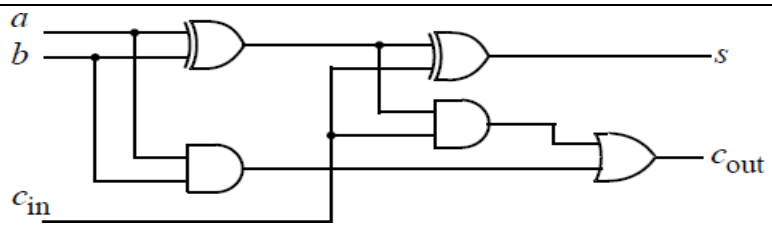


Figure 2: Gate implementation of full adder.

The gate implementation of 1-bit full adder is given in the figure

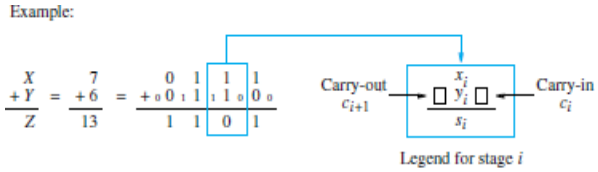


Figure 9.1 Logic specification for a stage of binary addition.

Ripple carry adder

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded. with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure 3 shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder. Notice from Figure 3 that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits a0 and b0 _ in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits s0 and s3.

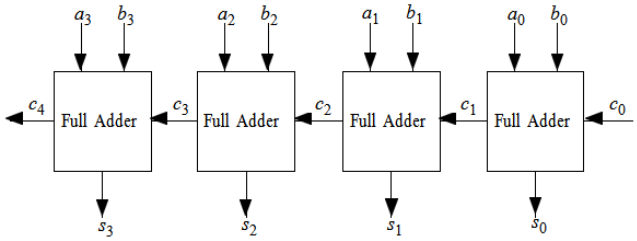


Figure 3: 4-bit full adder.

Ripple carry adder delays

In the ripple carry adder, the output is known after the carry generated by the previous stage is produced. Thus, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage

to the most significant stage. As a result, the final sum and carry bits will be valid after a considerable delay.
Table 2 shows the delays for several CMOS gates assuming all gates are equally loaded for simplicity. All delays are normalized relative to the delay of a simple inverter. The table also shows the corresponding gate areas normalized to a simple minimum-area inverter. Note from the table that multiple-input gates have to use a different circuit technique compared to simple 2-input gates.

Table 2: CMOS gate delays and areas normalized relative to an inverter.

Gate	Delay	Area	Comment
Inverter	1	1	Minimum delay
2-input NOR	1	3	More area to produce delay equal to that of an inverter
2-input NAND	1	3	More area to produce delay equal to that of an inverter
2-input AND	2	4	Composed of NAND followed by inverter
2-input OR	2	4	Composed of NOR followed by inverter
2-input XOR	3	11	Built using inverters and NAND gates
-input OR	2		Uses saturated load ( ).
-input AND	3		Uses -input OR preceded by inverters ( ).

For an n-bit ripple carry adder the sum and carry bits of the most significant bit (MSB) are obtained after a normalized delay of

$$\text{Sum } s_{n-1} \text{ delay} = 4n + 2 \quad (1)$$

$$\text{Carry } c_n \text{ delay} = 4n + 3 \quad (2)$$

For a 32-bit processor, the carry chain normalized delay would be 131. The ripple carry adder can get very slow when many bits need to be added. In fact, the carry chain propagation delay is the determining factor in most microprocessor speeds.

Table 3: Delays for the outputs of a 4-bit ripple carry adder normalized to an inverter delay.

Signal	Delay
,	6, 7
,	10, 11
,	14, 15
,	18, 19

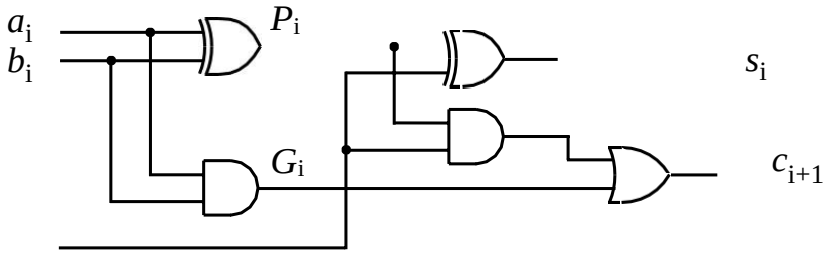
Carry lookahead adder (CLA)

The carry lookahead adder (CLA) solves the carry delay problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases:
(1) when both bits ai and bi are 1, or
(2) when one of the two bits is 1 and the carry-in is 1 .
Thus, one can write,

$$c_{i+1} = a_i.b_i + (a_i \oplus b_i).c_i \quad (3)$$

$$s_i = (a_i \oplus b_i) \oplus c_i \quad (4)$$

The above two equations can be written in terms of two new signals Pi and Gi , which are shown in Figure 4



c_i

Where Pi and Gi are called the carry generate and carry propagate terms, respectively. Notice that the generate and propagate terms only depend on the input bits and thus will be valid after one and two gate delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its proper value. Let‟s apply this to a -bit adder to make it clear.
Notice that the carry-out bit, , of the last stage will be available after four delays: two gate de- lays to calculate the propagate signals and two delays as a result of the gates required to implement Equation 13.

$$c_{i+1} = G_i + P_i.c_i \quad (5)$$
$$s_i = P_i \oplus c_i \quad (6)$$

$$G_i = a_i.b_i \quad (7)$$
$$P_i = a_i \oplus b_i \quad (8)$$
(9)

Putting $i = 0, 1, 2, 3$ in Equation 5, we get

$$c_1 = G_0 + P_0.c_0 \quad (10)$$
$$c_2 = G_1 + P_1.G_0 + P_1.P_0.c_0 \quad (11)$$
$$c_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \quad (12)$$
$$c_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0 \quad (13)$$

Figure 5 shows that a 4-bit CLA is built using gates to generate the and signals and a logic block to generate the carry out signals according to Equations 10–13.

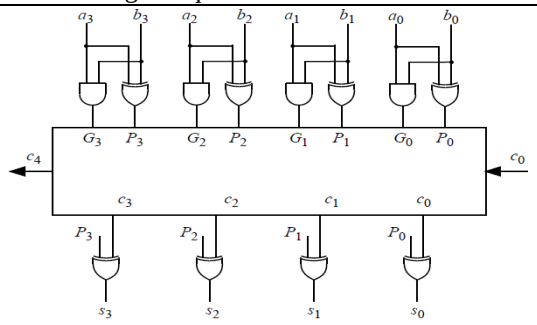
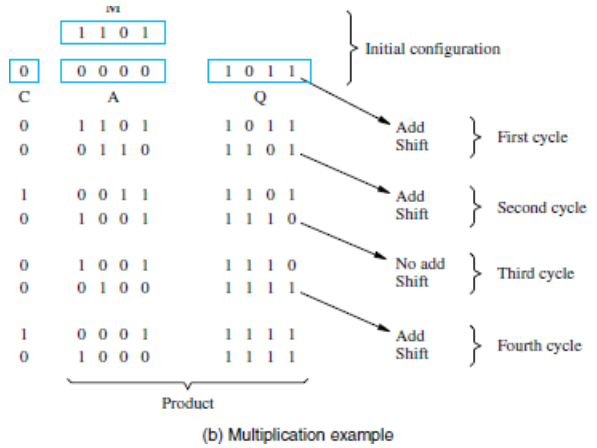


Figure 5: 4-Bit carry lookahead adder implementation detail.

The disadvantage of CLA is that the carry logic block gets very complicated for more than -bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 -bits.

Shift – Add Multiplier

Multiplication is often defined as repeated additions. Thus, to calculate 11 × 23, you would start with 0 and add 11 to it 23 times.



(b) Multiplication example

In this, the 4 bit multiplier is stored in Q register, the 4 bit multiplicand is stored in register B and the register A is initially cleared to zero. The multiplication process starts with checking of the least significant bit of B whether it is 0 or 1.

If the B0 = 1, the number in the multiplicand (B) is added with the least significant bits of the A register and all bits of C, A and Q registers are shifted to the right one bit.

If the bit B0 = 0, the combined C and Q registers are shifted to the right by one bit without performing any addition. This process is repeated for n times for n bit numbers. This method of binary multiplication is called as parallel multiplier.

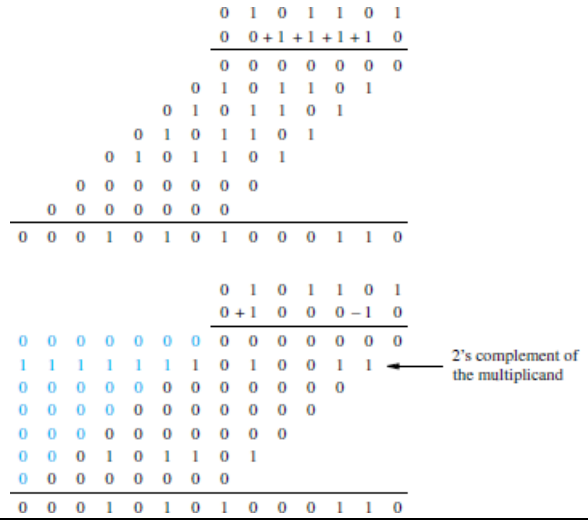
Consider the above figure in which the multiplier and multiplicand values are given as 1011 and 1101 which are loaded into the Q and A registers respectively.

Initially the register C is zero and hence the A register is zero, which stores the carry in addition.

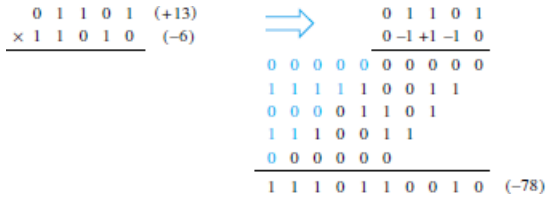
Since the B0 =1, then the number in the B is added to the bits of A and produce the addition result as 1101, and the Q and A register are shifted their values one bit right so the new values during the first cycle are 0110 and 1101 respectively.

This process has to be repeated four times to perform the 4 bit multiplication. The final multiplication result will be available in the A and Q registers as 10001111

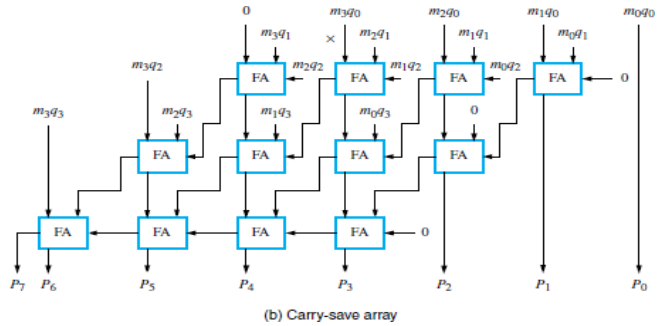
Booth Multiplier



The Booth algorithm generates a 2*n*-bit product and treats both positive and negative 2"scomplement *n*-bit operands uniformly. In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.



Carry-Save Addition



(b) Carry-save array

Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process.

This structure is in the form of the array in which the first row consists of just the AND gates that produce the four inputs m_3q_0, m_2q_0, m_1q_0, and m_0q_0.

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

Integer Division

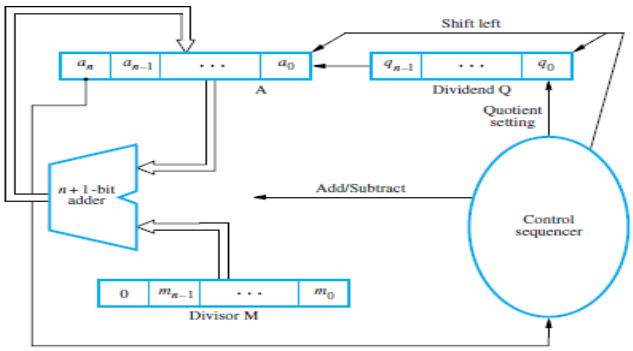
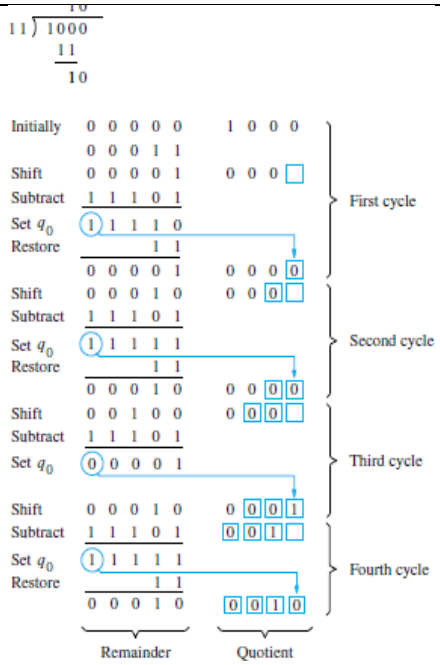


Figure 9.23 Circuit arrangement for binary division.



Restoring Division

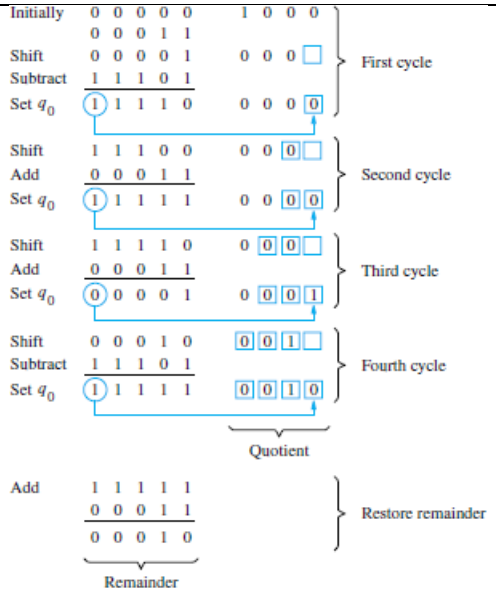


Figure 9.25 A non-restoring division example.

Non restoring division

An *n*-bit positive divisor is loaded into register M and an *n*-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the *n*-bit quotient is in register Q and the remainder is in register A.

The required subtractions are facilitated by using 2"s-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps *n* times:

1. Shift A and Q left one bit position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.

If A is positive, we shift left and subtract M, that is, we perform 2A− M. If Ais negative, we restore it by performing A+ M, and then we shift it left and subtract M.

This is equivalent to performing 2A+ M. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed. following algorithm for *non-restoring division.*

Stage 1: Do the following two steps *n* times:

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

Stage 2: If the sign of A is 1, add M to A.

Stage 2 is needed to leave the proper positive remainder in A after the *n* cycles of Stage 1.