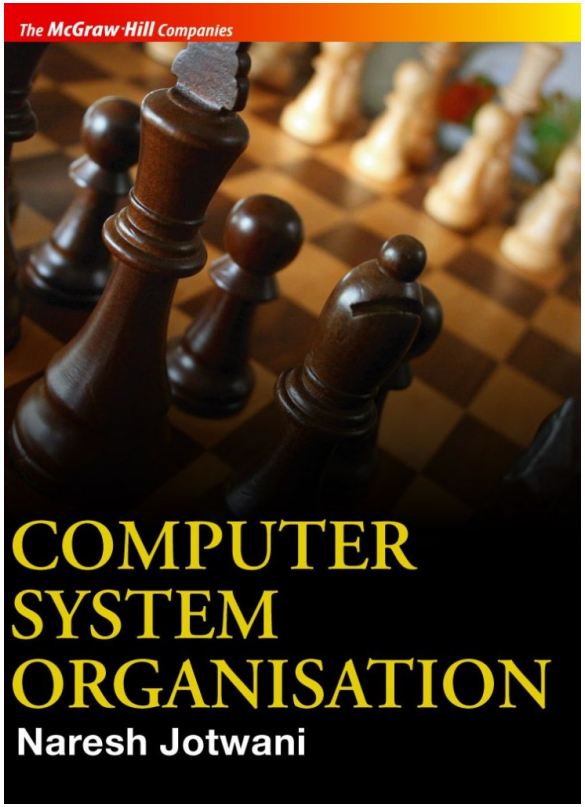
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



Chapter 8

COMPUTER ARITHMETIC

Introduction

- Computer arithmetic is in principle very similar to usual decimal arithmetic.

- The main differences are:

  (a) Computer circuits work on binary rather than decimal data

  (b) Computer circuits are designed to operate at high speeds

Integer addition and subtraction:

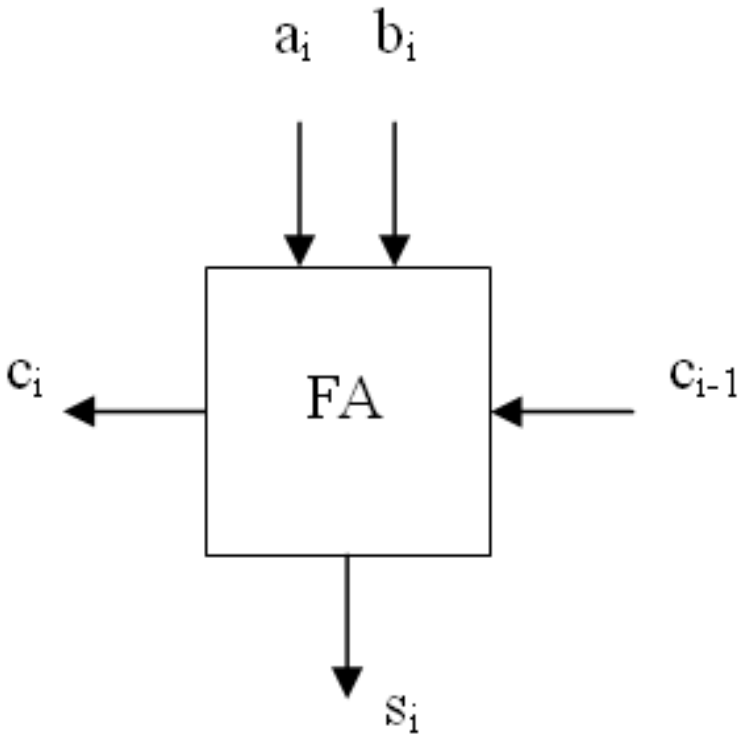
In the process of digit-by-digit addition:

- *Carry out* is an 'overflow' from a digit position to the next higher digit position (on its left).
- *Carry out* from one digit position becomes *carry in* for the position to its left.
- If a digit position does not have *carry in*, we may set its *carry in* to 0.

These concepts apply to binary addition just as they do to the usual decimal addition.

- In the next figure, assume a_i, b_i and c_{i-1} as inputs, and s_i and c_i as outputs of the digital circuit shown.
- This circuit is known as a *full adder.*
- Inputs a_i, b_i and c_{i-1} will be either '0' or '1', and the circuit will add up their values.

One stage of integer addition in binary using a *full adder* -



- Between the input bits, there is no relative place value – i.e. '1' has numerical value one, whether it occurs as the value of a_i, or b_i, or c_{i-1}.

- i.e. we need to simply count how many of these input bits are '1'.

- This count of '1' bits on the three inputs is in the range from 0 to 3. We encode this count into two bits, which are the output bits c_i and s_i.

- There is relative place value between the output bits. Since carry out c_i is taken into the next position on the left, its place value is twice that of s_i.

- The next slide shows the complete truth table of the *full adder*.
- The table has $2^3 = 8$ rows.
- The first row corresponds to all inputs being '0', i.e. the total count of '1' bits is zero.
  - c_i and s_i remain '0'.
- The next three rows correspond to any one input to the circuit being '1'.
  - No *carry out* is generated in these three cases, and the outputs are c_i = '0' and s_i = '1'.

Truth table of full adder:

Inputs			Outputs	
A_i	b_i	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0		
1	0	0		
1	1	0	1	0
1	0	1		
0	1	1		
1	1	1	1	1

- The next three rows correspond to any two inputs to the circuit being '1'.
  - A *carry out* is generated in these three cases, and the outputs are c_i = '1' and s_i = '0'.

- The last row corresponds to all three inputs to the circuit being '1'.
  - A *carry out* is generated and the sum bit s_i becomes '1'. Outputs are c_i = '1' and s_i = '1'.

- The circuit is a *combinational circuit* – the output at any time is a function only of the input at that time. Circuit has no memory of prior inputs or outputs.

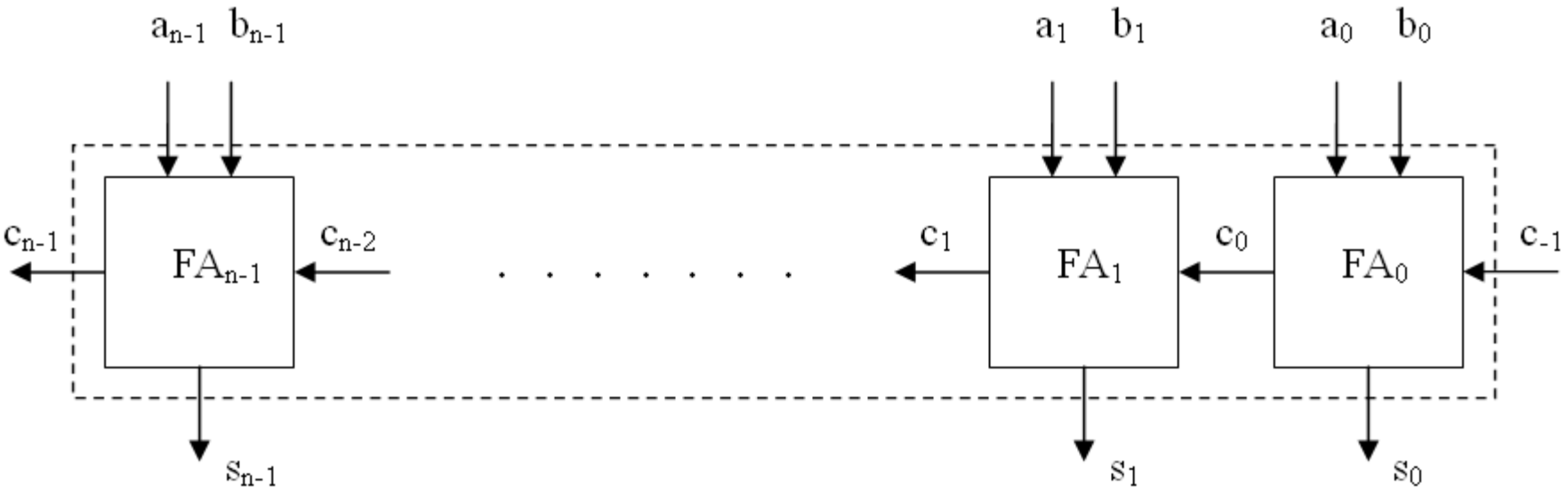
- Logic gates can be used for a realization of the circuit, with maximum gate delay of 2.

- This *full adder* circuit can perform the work of binary addition at one bit position.

- In the next slide, *n* of these full adders are connected for full *n*-bit addition.

- The only 'cause-and-effect' relationship between the stages is:
  - *Carry out* c_i from bit position *i* becomes *carry in* c_{j-1} at bit position $j = i+1$ to its left

Circuit for *n*-bit addition, using *n* full adders



- We need to 'string together' *n* instances of the full adder, i.e. *carry out* bit of one stage becomes the *carry in* bit of the next stage to its left.

- The *carry in* bit of the rightmost stage – i.e. bit position 0 – has been numbered as c_{-1}.

- The *carry out* bit of the leftmost stage becomes the final *carry out* of the *n*-bit adder.

- At the end of addition, this bit is made available to programmer as the *carry* condition flag C.

- The dashed outer rectangle encloses the *n* full adders connected together. Known as *ripple carry adder*.
- This complete circuit has as its inputs the two *n*-bit integers A and B, and the *carry in* c_{-1} shown on the right.
- Outputs of the circuit are the *n*-bit sum S, and *carry out* c_{n-1} shown on the left.
- But, although this circuit is fully functional, it has a shortcoming.

- In any digital circuit, there is a *time delay* between the time inputs are applied and the time when outputs become stable.
- In the *n*-bit adder, the maximum number of logic gates is encountered in the paths between inputs the a_0, b_0 and c_{-1} on the right, and the outputs s_{n-1} and c_{n-1} on the left.
- These circuit paths pass through each of the *n* full adders in the circuit, from right to left.
- Each full adder on the path contributes 2 gate delays, and therefore the total gate delay of the circuit is 2*n*, unacceptable from a performance point of view.

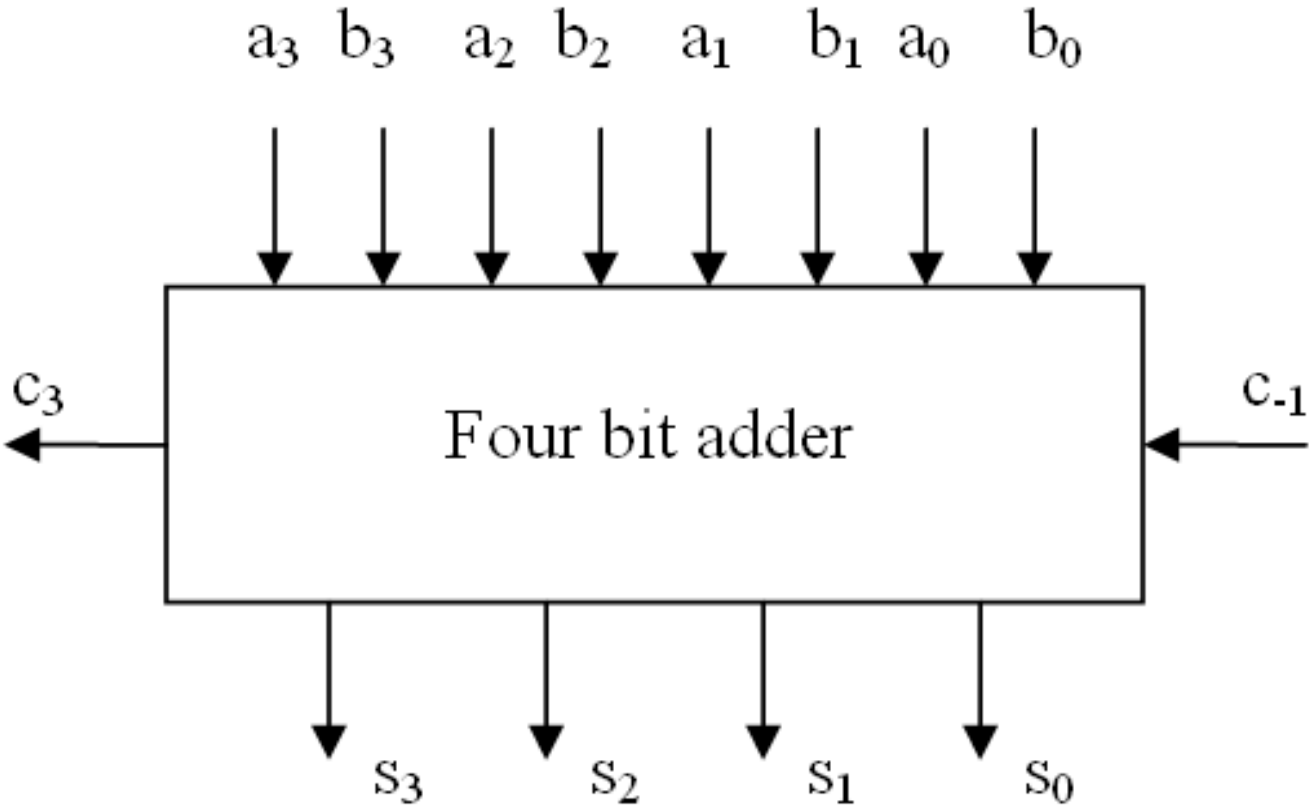
Carry Look-ahead:

For the *n*-bit adder:

- Suppose the building block is a 2-bit adder which also provides for *carry in* and *carry out*.
- Connect *n*/2 of such building blocks together to obtain the required *n*-bit adder. Carry ripples through these *n*/2 stages,.
- Each 2-bit adder can be designed with 2 gate delays
- The complete *n*-bit adder will now have *n*/2x2 = *n* gate delays
- Faster by a factor of 2, as compared to the previous *ripple carry* circuit with 2*n* gate delays

- We can also use 4-bit adders.
  - *n*/4 of the 4-bit adders can be connected for an *n*-bit adder. with a total gate delay of *n*/4x2 = *n*/2.
- The next slide shows such a 4-bit adder.
- The two 4-bit inputs are A = $a_3a_2a_1a_0$ and B = $b_3b_2b_1b_0$; the 4-bit sum is S = $s_3s_2s_1s_0$
- *Carry in* is c_{-1} and *carry out* is c_3
- This 4-bit adder can also be thought of as a single digit hexadecimal adder.

A 4-bit adder as a combinational building block:



- *Carry look-ahead* is a systematic technique for designing multiple bit adders.

- Consider the 1-bit full adder discussed earlier.

- In the i^{th} stage of an *n*-bit adder, we define the following two functions of input bits a_i and b_i:

  Generate function $g_i = a_i \cdot b_i$

  Propagate function $p_i = a_i + b_i$

- Then *carry out* bit c_i is given by:

  $c_{i+1} = g_i + p_i \cdot c_{i-1}$

- Generate bit g_i indicates that the i^{th} stage *generates carry out*, regardless of whether there is carry coming into this stage.

- Propagate bit p_i indicates that the i^{th} stage *propagates* incoming carry, regardless of whether there is carry generated in this stage.

- In stage i, *carry out* c_i is '1' if:

  (a) stage *i* generates a carry, AND/ OR

  (b) stage *i* receives incoming carry c_{i-1}, which it propagates to the stage on its left

- For the left-most stage of a 4-bit adder, we write:

$$c_3 = g_3 + p_3 \cdot c_2$$

- Substituting in this the expression for c_2:

$$c_3 = g_3 + p_3 \cdot (g_2 + p_2 \cdot c_1)$$

- By substituting for c_1, then for c_0 in the resulting expression, we obtain an expression for c_3 in terms of *carry in* c_{-1} and the input bits a_i and b_i, i = 0 to 3.

- THUS → Carry bits c_i, for i = 0 to 3, depend on input bits only, and we do not require a carry to ripple forward stage by stage from right to left.

- Sum bit s_i of each stage is a function of a_i, b_i and c_{i-1}, as before in the 1-bit full adder of the i^{th} stage.

- The slow 'ripple carry' from right to left has been eliminated.

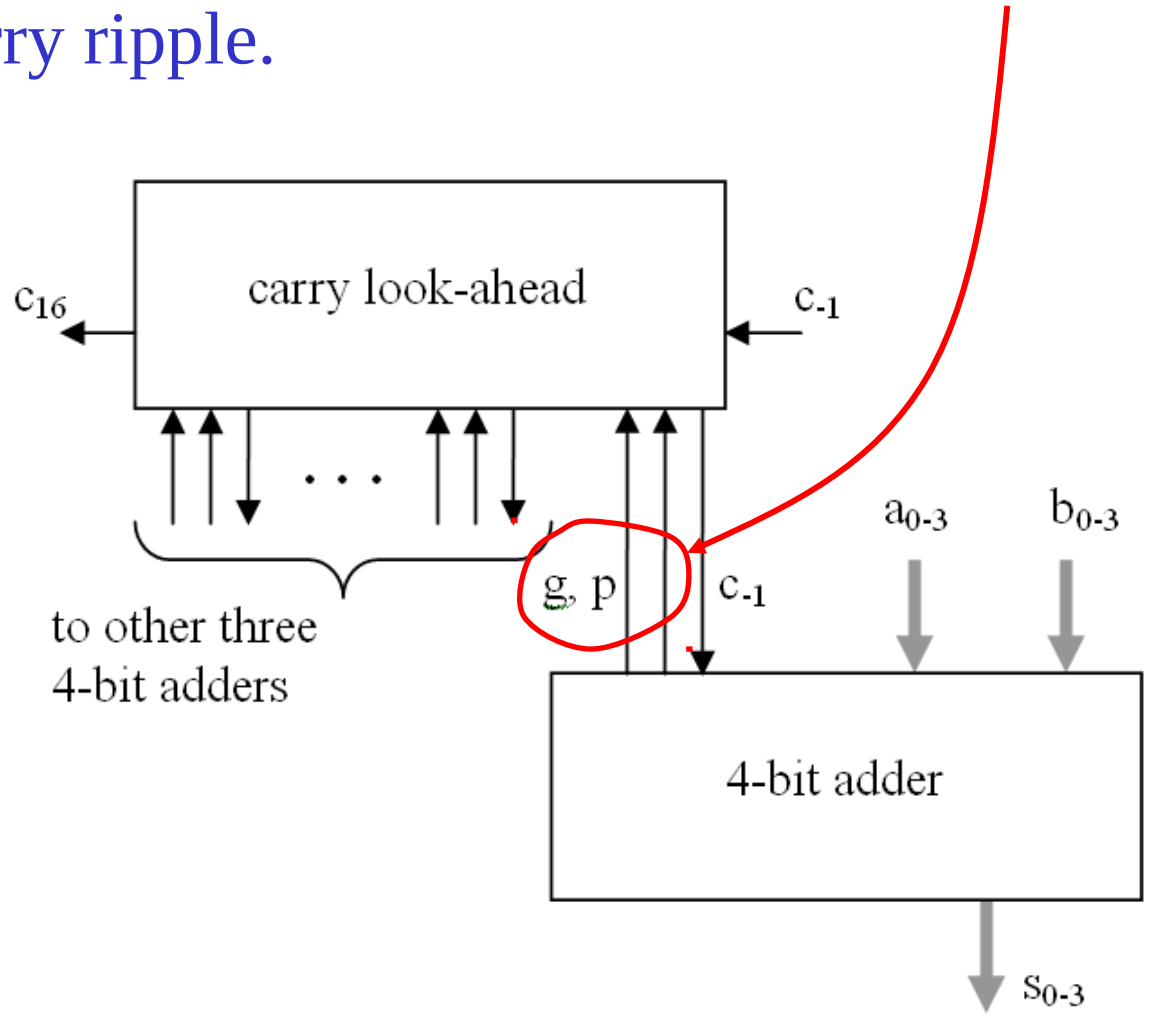
- In this way, a 16-bit adder can be built by connecting together four of the 4-bit adders, which are built using carry look-ahead.

- But, if this is done, a carry still has to ripple through from right to left, from one 4-bit adder to its left neighbour.

  - This is because, while each 4-bit adder internally makes use of carry look-ahead, there is no carry look-ahead between stages.

- Fortunately, as shown below, the principle of carry look-ahead also applies to a set of 4-bit adders connected together.

- For this, *generate* and *propagate* functions must be calculated for each 4-bit adder, and utilized to avoid the carry ripple.



- In the diagram, notation ‘g’ and ‘p’ indicates *generate* and *propagate* functions, respectively, from the single 4-bit adder shown.

- Carry look-ahead logic is shown above it, as a separate sub-circuit, to which four such 4-bit adders are connected.

Unsigned integer multiplication:

- R*epeated addition* proves to be a very inefficient method for larger integers.

- *multiply-shift-and-add* method is faster and more efficient. It involves multiplication by one multiplier digit at a time, as illustrated below:

```
      2 1 3
    x 3 7 9
  ---------
    1 9 1 7
  1 4 9 1 0
  6 3 9 0 0
-----------
  8 0 7 2 7
```

} Partial products

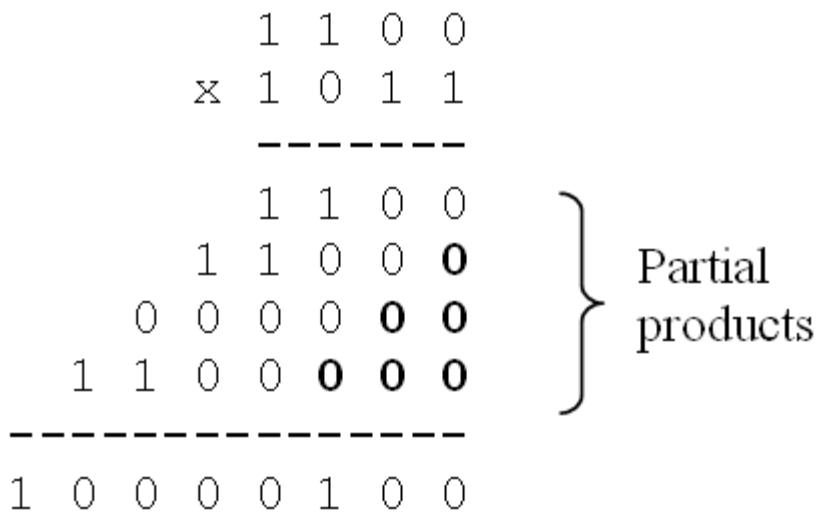
- In the second partial product, note that the number 7x213 = 1491 is shifted one position to the left.
- This is because this shift multiplies 7x213 by 10, to give us the required partial product 70x213.
- The third partial product involves a left shift by two digits.
- Thus the multiplier 379 is broken up as 9+70+300, and each partial product represents the product of one of these terms with the multiplicand 213.
- Multiplication of two *n* digit integers will in general produce a product which has 2*n* digits.

- *Multiply-shift-and-add* is an efficient method of multiplication; multiplication of any two *n* digit numbers requires the addition of *n* partial products.

- In binary, this is the method employed in computer systems for integer multiplication.

- *Multiply-shift-and-add* method applied to binary digits is shown below:

  – Let the four bits of the multiplier 1011 be denoted by $b_3b_2b_1b_0$ respectively.



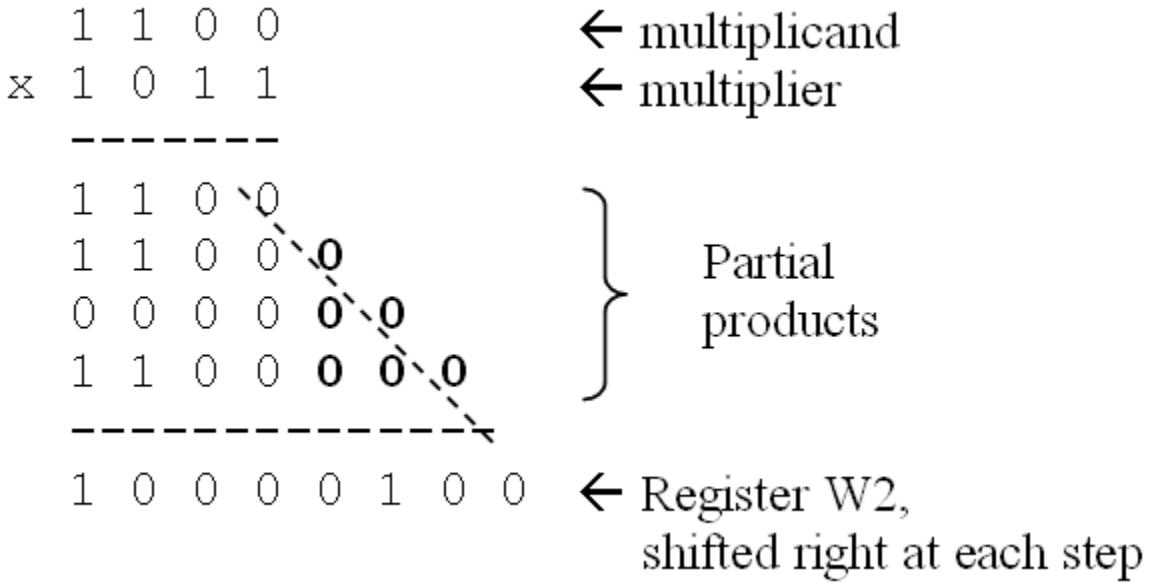
- Observe that:
  - (a) When a multiplier bit b_i is '0', the corresponding partial product is zero, and
  - (b) When multiplier bit b_i is '1', the corresponding partial product is simply a shifted version of the multiplicand 1100.

- Bold-face zeros in partial products show number of positions the multiplicand has been shifted to the left.

- For implementation on a computer:

  (i) We reserve a result register, say W2, of size 2*n* bits.

  (ii) To keep the multiplicand unchanged through the multiplication process, we shift W2 to the right at each step, instead of shifting the multiplicand to the left.



- The dashed inclined line above corresponds to the rightmost bit position of W2.

- Note that we do not need separate registers for the *n* partial products.
- Of course, we do need two separate *n*-bit registers for the multiplicand and multiplier.
- And we need the 2*n*-bit register W2 into which the partial products get added, and which stores the result.

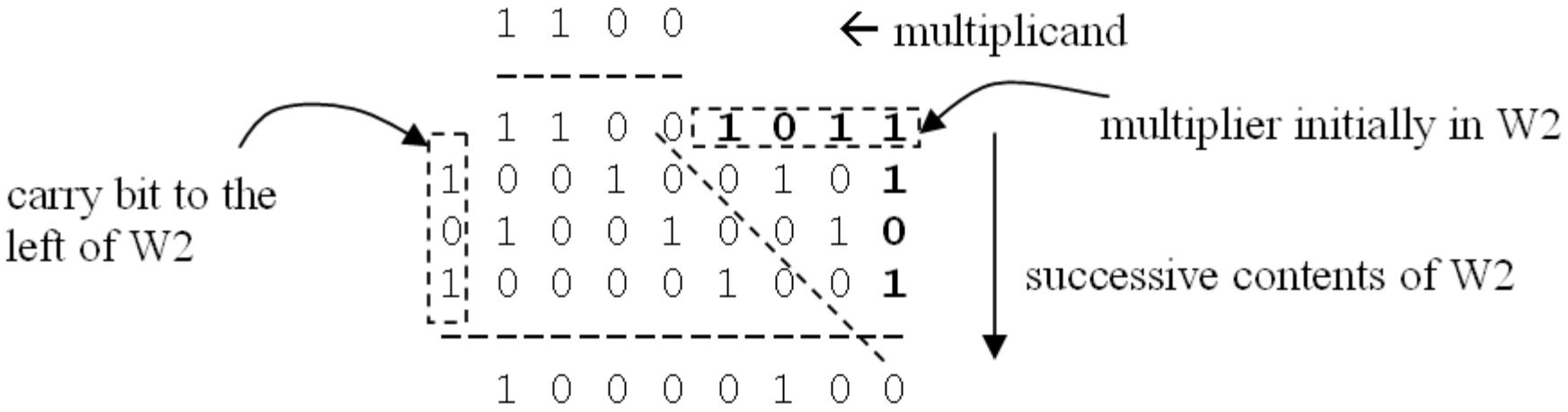
- Note also that:

  W2 is initially zero, and only *n* of its 2*n* bits are utilized in the first *shift-and-add* step carried out.

  Once a bit b_i of the multiplier has been examined, it is not needed again during the multiplication process.

- Thus the right half of W2 can be used to initially hold the multiplier. As a bit of the multiplier gets shifted out on the right, one more bit of W2 is available to store the accumulating product.
- Carry is a part of the accumulating product, and must be shifted in from the left during the next right shift.

- This arrangement is shown below:



- The four bits 1101 in bold-face shown on the right are successive bits of the multiplier.
- One by one they get shifted out. At any step, the multiplicand is added into W2 only if the particular multiplier bit is ‘1’.
- The inclined dashed line shows how the rightmost ‘0’ bit of the multiplicand gets shifted right one bit position at a time, to finally become the rightmost bit of W2.
- The next slide shows this technique in algorithmic notation. The inner loop of the algorithm is repeated *n* times, for *n*-bit multiplication.

Algorithm for unsigned integer multiplication

```
initialize W2(L) to zero, and
           W2(R) to multiplier
multiplicand is in register M
repeat n times
{
   if rightmost bit of W2 is '1' then
      add M to W2(L), saving carry
   shift carry & W2 one bit to right
}
final product is in W2
```

- In a typical processor, the 2*n*-bit register W2 will be a pair of *n* bit general purpose registers, e.g. R3 and R4, and M a third such register.

- (a) ADD and SHIFT can be used to provide the multiply function in software.

- (b) For a hardware integer multiplier, the algorithm above must be translated into an appropriate *sequential* digital circuit, and provided as a functional unit.

- Integer multiply will then be a machine instruction, similar to ADD and SUB.

- (c) If the processor does not provide an integer multiply instruction, then it is possible to implement the multiply algorithm in *microcode*. Multiply instruction is thereby added to the instruction set.

- Microcoded implementation provides execution time faster than software, but slower than that of a hardware multiplier.

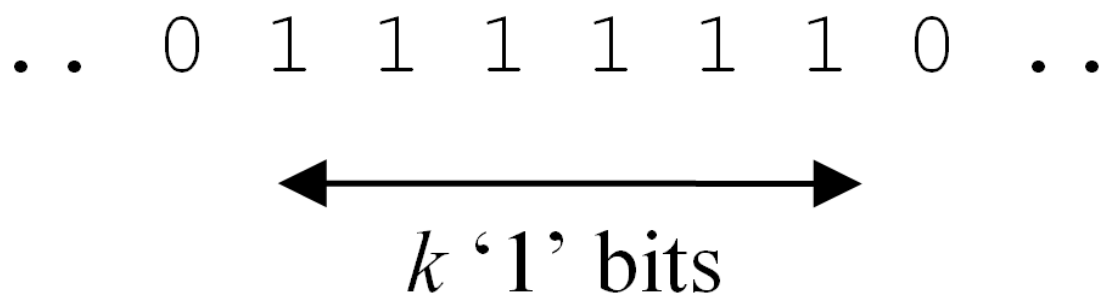
Booth's algorithm and integer division

- The previous algorithm cannot be used directly for multiplying signed integers.
- *Booth's algorithm* provides a faster alternative which also takes care of signed multiplication in a natural way.
  - Consider 8-bit multiplier 00111111, which equals decimal 63. This can be written as 64 - 1, which in binary is: 01000000 - 00000001.
  - The first term represents a left shift of the multiplicand by 6 bit positions, whereby it is multiplied by 64.
  - The second term represents a subtraction of the multiplicand.

- Now consider the multiplier 00111100, which has decimal value 60. This can be written in binary as 01000000 – 00000100.

- The second term now represents left shift by two positions and a subtraction.

- In general, suppose the multiplier has a *k* consecutive '1' bits, within its total size of *n* bits. Then the relevant part of the multiplier has the form:



- Let the rightmost of these *k* '1' bits be b_j. Then the leftmost of these *k* bits is b_{j+k-1}. The numerical value of these *k* bits is given by:

$$\sum_{i=j}^{j+k-1} 2^i$$

- This is a summation of *k* different powers of 2, which would need *k* steps of *shift-and-add*.

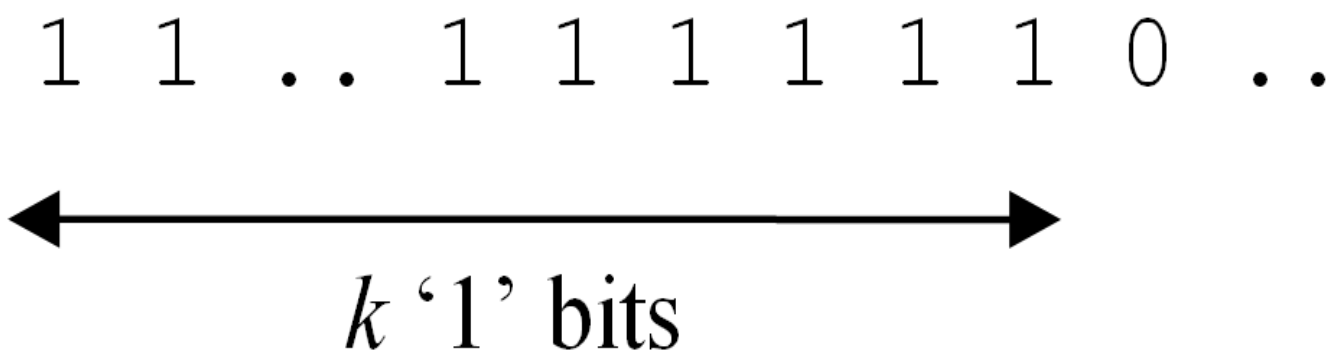
- But the summation equals $2^{j+k} - 2^j$, which involves only one addition and one subtraction.

- If $k > 2$, then instead of k additions, we get by with one addition and one subtraction, for a total of only 2 operations.

- This is the essence of the Booth's algorithm for multiplication.

- The algorithm has the additional benefit that it naturally takes care of signed operands in 2's complement form.

- To see that, consider the following multiplier:



- The *k* '1' bits are now the leftmost *k* bits in the *n*-bit multiplier, i.e. *j*+*k*-1 = *n*-1. Now we get the equivalent value of the multiplier as: $2^{j+k} - 2^j = 2^n - 2^j$

- Value 2^n corresponds to bit b_n – which is <u>not</u> within the *n* bits $b_{n-1}..b_0$ of the multiplier. Without this bit, the value of summation reduces to -2^j, which is the numerical value of the leftmost *k* bits of the multiplier <u>in 2's complement</u>.

- Thus Booth's algorithm has the following advantages:
  (a) sequences of '1' bits in multiplier are handled more efficiently
  (b) signed operands are handled in a natural manner, in 2's complement representation.

Integer division:

(i) integer division can be viewed as repeated subtraction;

(ii) in binary, the 'long-hand' method of division amounts to a sequence of *shift-and-subtract* operations.

- *Shift-and-subtract* steps of division can generate the integer division algorithm.

- It is possible to implement the division algorithm in software, hardware, or microcode.

Floating point operations:

- To add two floating point numbers, it is necessary to make their exponents equal, before the mantissas can be added.
- A floating point number in binary remains unchanged in value if its mantissa is shifted right by k bit positions and the exponent is increased by k.
- By applying this transformation to the operand with the smaller exponent, we can make the two exponents equal. Then the two mantissas can be added.
- The sum will in general be un-normalized, and must be normalized as the final step of the operation.

- Addition & subtraction of floating point numbers go through the same series of steps. Sign bits of the two operands must be taken into account during the process.

- For multiplication and division, the two respective exponents need not be equal.

- In multiplication, exponents of the two operands are added; in division, exponent of the divisor is subtracted from that of the dividend.

- Example: See floating point operation 32 - 15.5 on next slide.

```
+ 32:    0  10000100  00000000000000000000000
- 15.5:  1  10000010  11110000000000000000000
```

- We un-normalize the second number, i.e. shift its mantissa to the right by 2 bits, and increase its exponent by 2, to give:

```
  1  10000100  01111000000000000000000
```

- Note there is no longer a hidden '1' bit, since the former hidden bit is now included in the 23+2 = 25 bits of this mantissa. Two numbers have opposite signs, so we perform mantissa subtraction:

```
    1. 000000000 . . .
  -  0. 011111000 . . .
     -----------------
=    0. 100001000 . . .
```

← result mantissa, to be re-normalized

Now we normalize the result, giving:

`0 10000011 00001000000000000000000  = 16.5 (correct result)`

- Three options for implementing floating point operations on a system: *software*, *dedicated hardware*, or *microcode*.

- On many processors, floating point operations are provided using a dedicated hardware unit known as *floating point unit* (FPU).

- The two functional units ALU and FPU are usually capable of operating *in parallel*.

- Such parallelism of operations is a major potential source of performance enhancement.

Binary Coded Decimal (BCD) Arithmetic:

- *Packed* BCD format uses a nibble – i.e. 4 bits – for each decimal digit. For example, the eight digit number 31052008 would be packed into four bytes as:

0011 0001	0000 0101	0010 0000	0000 1000

- On some processors, special machine instructions operate on numbers in packed BCD format. 'Auxiliary carry' must be provided from right nibble to left.

- Programming languages in earlier years provided a packed BCD type of number representation.

Conditions, exceptions, and faults:

- Condition flags are needed to provide required conditional transfers of control within a program.
- Instruction set of the processor provides for conditional transfers based on these flags. [Recall *NICE* instruction set seen in Chapters 4 & 5.]
- These condition flags do not represent '*errors*', since they occur routinely in any running program.
- Programmer responds to these flags according to program requirements. Branch decisions made on that basis make up the *flow of control* within a program.

- But other conditions can arise in a running program which need to be handled in a different manner.

- *Floating point overflow, underflow, divide by zero* etc. are examples of such conditions.

- Such conditions do not cause any condition flags to be set in PSW, but they are treated as 'run-time' errors, and they also known as *exceptions*.

- Yet another type of condition which can arise within a processor is termed as a *fault*.
- Examples: *memory access violation, page fault*

  [to be studied in Chapter 9].

- In such cases, it is not possible for the program itself to take corrective action.

- Rather, exceptions and faults are handled by *interrupts*. The source of an interrupt may be:

  (a) an external input or output device, such as keyboard or disk [to be studied in Chapter 10],

  (b) An exception of fault occurring within the processor, such as floating point overflow or memory violation,

  (c) a *software interrupt*, which is a machine instruction executed by the running program.

  [Note: At present, *NICE* instruction set does not provide for software interrupt].

Summary

- Full adder, *n*-bit ripple carry adder
- Use of carry look-ahead technique
- Shift-and-add technique for multiplication
- Booth's algorithm
- Floating point operations
- Conditions, exceptions, and faults encountered in a running program