13 Multipliers

- Multiplication methods can be classified into three groups:
- bit-serial, word-serial and parallel algorithms.
- Read the appendix at the end of the chapter for further clarification.
- We consider multiplication of two numbers represented by numerals Q and D:

$$P = Q * D$$

Product — P; Multiplier — Q; Multiplicand — D

where

- Q is n-bit **multiplier**: $Q_{n-1:0} = [\, q_{n-1} \ldots q_1\, q_0 \,]$
- D is m-bit **multiplicand**: $D_{m-1:0} = [\, d_{m-1} \ldots d_1\, d_0 \,]$
- P is $(k = m + n)$-bit **product**: $P_{k-1:0} = [\, a_{k-1} \ \ldots \ a_1 \ a_0 \,]$

13.1 Word-Serial Multiplication Processor – the Booth's algorithm

- The word-serial Booth's algorithm is probably the most popular multiplication algorithm used in most of the general purpose processors.
- In a word-serial algorithm a partial product is formed as a shifted sum of the previous partial product and a product of the multiplicand and a i-th digit of the multiplier.
- The one-bit Booth's multiplication algorithm can be described by the following pseudo-code:

```
P[0] = 0 ;  q_{-1} = 0 ;
for (i = 0 ;  i < n ;  i++ ) {
    q̂_i = −q_i + q_{i−1} ;
    P[i + 1] = (P[i] + q̂_i · D) · 2^{−1} ;
}
```

q_i	q_{i-1}	$\hat{q}_i$
0	0	0
0	1	+1
1	0	−1
1	1	0

- Examination of the pseudo-code reveals the following details of the algorithm:
  - The partial product, $P[i]$, is initialised to zero: $(P[0] = 0)$.
  - The additional, q_{-1}, bit is appended to the multiplier and is also initialised to zero.
  - At each step, a Booth's multiplier digit, $\hat{q}_i = -q_i + q_{i-1}$ $(\hat{q}_i \in \{-1, 0, +1\})$, is formed from a pair of adjacent multiplier digits, $q_i, q_{i-1} \in \{0, +1\}$.
  - The Booth's multiplier digit, $\hat{q}_i$ is used to determine the way in which the next partial product, $P[i + 1]$, is calculated from the previous one.

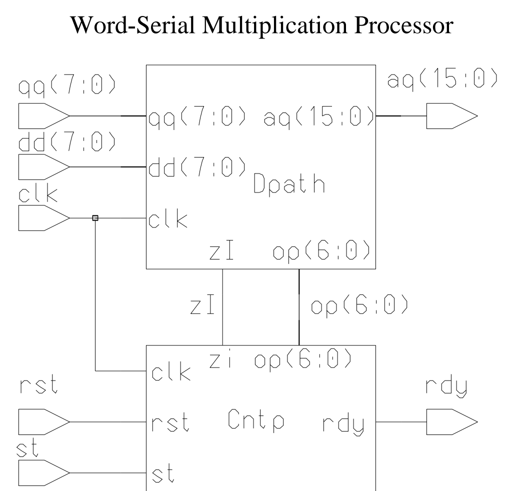
– At each multiplication step one of three possible operations is performed, namely, subtraction of D, no-operation, or addition of D, that is, either $(P[i] - D)$, or $P[i]$, or $(P[i] + D)$.
– The result of this operation $(P[i] + \hat{q}_i \cdot D)$, is then shifted right by one position to form the next partial product, $P[i+1]$.
  The shift-right operation implements multiplication by 2^{-1}.
– The final product is

$$P = P[n] = Q \cdot D$$

- A numerical example is given in Figure **??** and will be examined in detailed in the subsequent section.
- In the next design step, the pseudo-code description of the algorithm is converted into
  – a structure of the **datapath** and
  – a specification of the **control unit** given in a form of a flow-chart of operations performed by the word-serial multiplication processor.
- Remember that all registers are triggered by the positive-edge of the clock signal clk, and their operations are controlled by op-code signals generated by the control unit.

13.2 The top-level structure of the processor

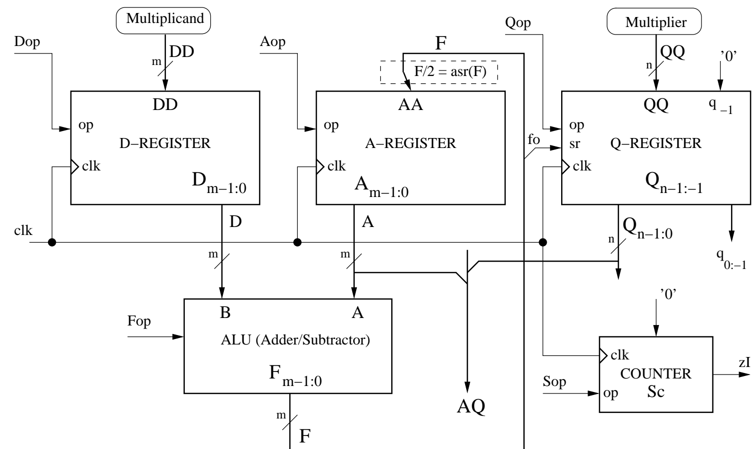
Word-Serial Multiplication Processor



- The top level schematic of the wsm processor consists of two components, namely, the datapath and the control unit:
- Two symbols, dpath and cntp, representing the datapath and the control unit components, respectively, are created and instantiated into the top level schematic wsm
- It is assumed that the multiplier and the multiplicand are 8-bit two's-complement numbers and are entered through the input ports `qq(7:0)` and `dd(7:0)`, respectively.
- The 16-bit result is available at the output port `aq(15:0)`.
- The multiplication operation is initialised with the assertion of the start signal, `st`.
- The completion of the operation is signaled by the **ready** signal, `rdy`.
- The **clock** and **reset** signals are `clk` and `rst`, respectively.
- The control unit generates the required 7-bit op-code `op(6:0)` to specify micro-operations performed by the functional blocks of the datapath.
- Upon completion of the required number of the multiplication steps, the step counter from the data path generates the signal `zI` which is interpreted in the control unit.
- Refer to the flow-chart of operations for details.

13.3 Datapath of the word-serial multiplication processor implementing the 1-bit Booth's algorithm

- The block diagram of the datapath consists of the registers which store variables used in the pseudo-code (sec 13.1), and combinatorial blocks like an ALU (adder/subtractor) which perform required operations on the stored variables.



D — the multiplicand register,

Q — the multiplier register,

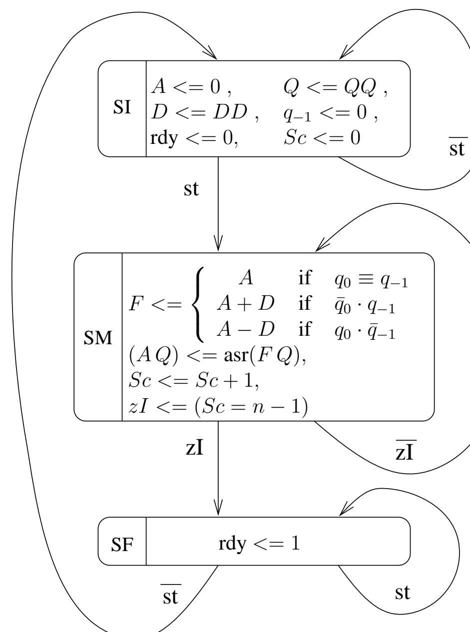
A — the more-significant half of the product register.
The least-significant part of the product is in the Q register, so that, the (partial) products are stored in the concatenated P = (A Q) register.

F — the ALU (Adder/Subtractor) performing operations $F <= A \pm D$, or $F <= A$.
The output F of the ALU is loaded into the A-register after an arithmetic shift-right operation, asr$(F) = F/2$, is performed.
The least significant bit, f_0, is shifted into the Q-register.

Sc — the step counter which counts from '0' to $n-1$ and generates the signal zI when Sc $= n-1$.

13.4 State diagram of the control unit

- The flow-chart of the word-serial multiplier using the one-bit Booth's method is implemented as a state diagram of the control unit.
- In the initial state, **SI**, when st$=0$, the multiplicand and the multiplier are loaded into the registers D and Q respectively, and the signal rdy is reset.
- When the start signal st$=1$ the control unit goes into the state **SM** in which multiplication steps are repeated n times (signal zI).
- In the final state, **SF**, the correct result is available at the port $(A\,Q)$. This is indicated by the signal rdy being asserted.



QQ and DD represent the multiplier and the multiplicand respectively.

Sc is the step counter initialised with zero. The signal zI indicates the $Sc = n-1$, that is, that the last multiplication step is performed.

F is the output of the ALU, which performs conditional operations as described.

$(F\,Q)$ is a concatenation of the outputs from the adder and from the Q register.
$(F\,Q)$ is arithmetically shifted one position right and then loaded into the concatenation of the registers A and Q.

st — the START signal,

rdy — the READY signal, asserted when the multiplication is completed.

- In order to multiply the next numbers, the start signal, st, must go low.

13.5 Numerical example

$D = (101101)_2 = -19$; $QQ = (101001)_2 = -23$

	$A[i], D$	Q	q_{-1}	
		101001	0	$\hat{q}_0 = -q_0 + q_{-1} = -1$
$A[0]$	000000			
$\bar{D}$	010010			
c_0	1			
F	010011			$F = A - D$
		- 10100	1	$\hat{q}_1 = -q_0 + q_{-1} = +1$
$A[1]$	001001	1		
D	101101			
F	110110	1		$F = A + D$
		- - 1010	0	$\hat{q}_2 = 0$
$A[2]$	111011	01		$A[2] = A[1] \cdot 2^{-1}$
		- - -101	0	$\hat{q}_3 = -q_0 + q_{-1} = -1$
$A[3]$	111101	101		
$\bar{D}$	010010			
c_0	1			
F	010000	101		$F = A - D$
		- - - - 10	1	$\hat{q}_4 = -q_0 + q_{-1} = +1$
$A[4]$	001000	0101		
D	101101			
F	110101	0101		$F = A + D$
		- - - - - 1	0	$\hat{q}_5 = -q_0 + q_{-1} = -1$
$A[5]$	111010	10101		
$\bar{D}$	010010			
c_0	1			
F	001101	10101		$F = A - D$
		- - - - - -	1	
$A[6]$	000110	110101		

$$A = A[6] = Q \cdot D = +437$$

- Initially, the multiplicand, DD, is loaded in an m-bit register D, and the multiplier, QQ, is loaded in an n-bit register Q,
- The initial value of the partial product, $P[0] = 0$, is loaded in the register A.
- The adder has the width m, which is equal to the number of bits of the multiplicand, D, and to the number of bits of the more significant part of the partial products which are stored in the register A.
- At each step, i-th, the digit of the Booth's multiplier, $\hat{q}_i$, is determined from the two least-significant bits of the register Q, and the value $\hat{q}_i \cdot D$ is added to the more significant part of the partial product, $A[i]$.
- The result of this addition, F, and the multiplier, Q, are shifted right by one position and loaded back into a combined $A - Q$ register.

  In this way, the least significant bits of the partial products are gradually shifted into a register Q, while the least significant bits of the multiplier are shifted out of the register Q through the additional position Q_{-1}.
- The final product resides in the concatenated register $A - Q$.

13.6 Operations of the datapath blocks

- From the flow-chart we can now compile tables of operations of each block of the datapath assigning opcodes for each elementary operation.

multiplicand register, D		
Dop	**operation**	
0	D ⇐ D	nop
1	D ⇐ DD	load

- The multiplicand register is very simple and performs only two operations as shown in the table

partial product register, A		
Aop	**operation**	
0	A ⇐ A	nop
1	A ⇐ F/2	ldAshr
2, 3	A ⇐ 0	reset

- The partial product register performs three operations.
- The ldAshr operation, that is, "load arithmetically shifted right signal vector $F(3..0)$ into A", can be more precisely described as

$$A <= (F(3)\ ,\ F(3..1))$$

multiplier register, Q		
Qop	**operation**	
0	Q ⇐ Q	nop
1	Q ⇐ shr(F(0),Q)	shrQ
2, 3	Q ⇐ (QQ & 0)	load

- The multiplier register is an (n+1)-bit register, e.g., $Q(3 \,..\, -1)$ (warning: in VHDL negative subscripts are not allowed). It performs three operations.
- The shift operation can be alternatively described as:

$$Q <= (F(0)\ ,\ Q(3..0))$$

step counter, Sc		
Sop	operation	
0	Sc ⇐ Sc	nop
1	Sc ⇐ Sc+1	count
2, 3	Sc ⇐ 0	reset

ALU, F		
Fop	operation	
0, 3	F ⇐ A	pass
1	F ⇐ A + B	add
2	F ⇐ A − B	subtract

- The step counter load the initial value (zero), and counts up until the value $n - 1$ is reached. It generates signal:

$$zI <= (Sc = n - 1)$$

- The ALU performs three operations: addition, subtraction and "pass"
- The op-codes for the ALU, Fop, are equal to the value of the current pair of the least significant multiplier digits. Therefore, we have:

$$\text{Fop} = Q(0 : -1).$$

- Wherever possible we should use mnemonic names for op-codes to retain flexibility of the design
- Binary values of the op-codes can be changed during the design process in order to simplify the internal structure of the components of the datapath.
- All op-codes can be, for convenience, collected in one 7-bit op-code word:

op(6:0) = (Dop, Aop, Qop, Sop)

- The ALU is driven directly by two least significant bits of the multiplier register.

13.7 Designing the datapath

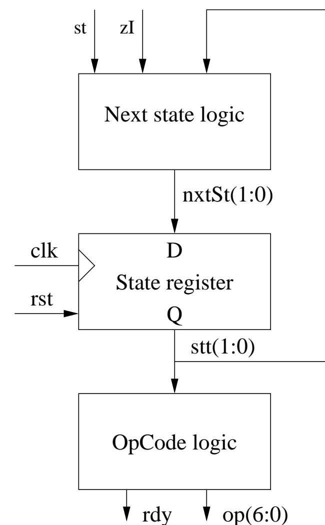
The procedure of designing the datapath consists of two main steps:

- **Synthesis** and **simulation** of all components (functional blocks) of the data path.

  For every block follow the steps described in the previous sections for typical sequential and combinatorial components.
- Connection of the components into a complete datapath.

  This can be achieved in one of following three ways:
  - Use schematic (graphical) entry tools of the CAD package and interconnect blocks as shown in the datapath block-diagram
  - Write a new VHDL entity and architecture for the datapath combining codes for the individual components.
  - Interconnect components using the VHDL structural design method.
- It is a good practice to simulate every new bit of the design, therefore, we should simulate not only all components of the datapath, but the complete datapath as well.
- However, due to its complexity it may be easier to do so after the control unit is designed and connected to the datapath.

13.8 The control unit



- The control unit is a synchronous state machine also knows as an algorithmic state machine.
- It has a typical structure in which we have to specify:
  - Number of bits in the state register (two in this case)
  - Input/output signals
- If we design the control unit manually, we convert the flow-chart (state diagram) into the excitation (next state) table and the output table.
- We can start with the symbolic names of the states SI, SM, SF and perform the state allocation when convenient.
- Similarly, the output table specify the way in which states are re-coded into the opcodes.
- I practical situations we specify the state diagram as behavioural VHDL code.

13.8.1 The VHDL program for the control unit

The program consists of the **entity** cntu in which input/output ports are specified, and an **architecture** behv which describes details of the behaviour of the control unit.

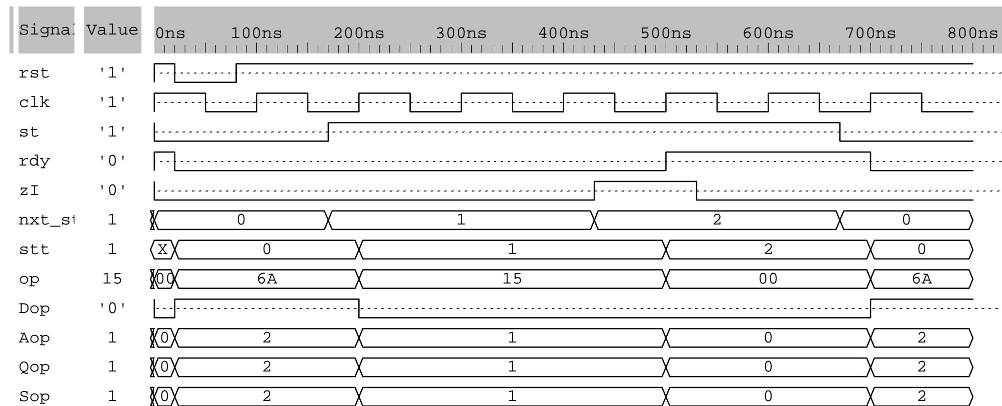
```
--    cntu     the control unit --  by  app
    library IEEE ;
    use IEEE.std_logic_1164.all ;
ENTITY cntu IS   -- the control unit  by  app
       PORT ( rst, clk, st, zi :  IN      STD_LOGIC ;
              op   : OUT   STD_LOGIC_VECTOR(6 DOWNTO 0) ;
           -- stt  : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
              rdy  : OUT   STD_LOGIC
            ) ;
END cntu ;

ARCHITECTURE behv OF cntu IS
--  TYPE states IS ( SI, SM, SF ) ;
--  SIGNAL  stt, nxtSt  :  states := SI ;

   SIGNAL  stt, nxtSt  : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
-- we can use a "hard" encoding of states
   CONSTANT SI : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
   CONSTANT SM : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
   CONSTANT SF : STD_LOGIC_VECTOR(1 DOWNTO 0) := "10" ;

-- Internal op-code signals and related constants
 SIGNAL   Aop, Qop, Sop    : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
 SIGNAL   Dop              : STD_LOGIC ;
 CONSTANT ldD              : STD_LOGIC := '1' ;
 CONSTANT nopD             : STD_LOGIC := '0' ;
 CONSTANT nop              : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
 CONSTANT ldAshr,shrQ,cnt  : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
 CONSTANT reset, load      : STD_LOGIC_VECTOR(1 DOWNTO 0) := "10" ;
```

```
BEGIN
       -- to synthesize edge-triggered flip-flops
       -- with asynchronous reset when  rst = 0
 clkd: PROCESS ( clk, rst)
 BEGIN
         IF (rst = '0') THEN
       stt <= SI ;
    ELSIF ( clk'EVENT AND clk = '1'
                       AND clk'LAST_VALUE = '0' ) THEN
       stt <= nxtSt ;
    END IF ;
 END PROCESS clkd ;

-- the stm process describes the transitions between states
-- and the output signals
 stm: PROCESS ( stt, st, zi )
 BEGIN
--  default assignments
   nxtSt <= stt  ;
     Dop <= nopD ;
     Aop <= nop ;
     Qop <= nop  ;
     Sop <= nop  ;
     rdy <= '0'  ;

--  state transitions and output signals
   CASE stt IS

   WHEN SI =>
     rdy <= '0' ;
     Dop <= ldD ;
     Aop <= reset ;
     Qop <= load ;
     Sop <= reset ;
     IF ( st = '1' ) THEN nxtSt <= SM ;  END IF ;
```



```
   WHEN SM =>
     Aop <= ldAshr ;
     Qop <= shrQ  ;
     Sop <= cnt ;
     IF ( zi = '1' ) THEN nxtSt <= SF ;  END IF ;
   WHEN OTHERS =>    --- when SF
     rdy <= '1' ;
     IF ( st = '0' ) THEN nxtSt <= SI ;  END IF ;
   END CASE ;
 END PROCESS stm ;

           op(6) <= Dop ;
  op(5 DOWNTO 4) <= Aop ;
  op(3 DOWNTO 2) <= Qop ;
  op(1 DOWNTO 0) <= Sop ;
END behv ;
```

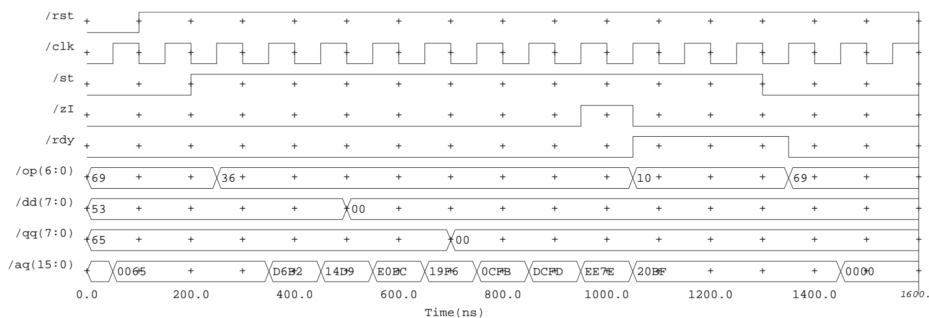
After the control unit is compiled and synthesized it is important to verify its behaviour by simulation. The following waveforms were obtained during simulation of the control unit:



- The signal rst resets the state of the control unit to **SI**.
- If the signal st is low, the control unit remains in the state **SI** until the first rising edge of the clock after the signal st goes high when the state SM is reached.
- From the state SM the transition to the state SF is made on the rising edge when the signal zI from the step counter is asserted.
- It is important to verify that the control unit generates correct op-codes in every state.

13.9 The complete word-serial multiplication processor

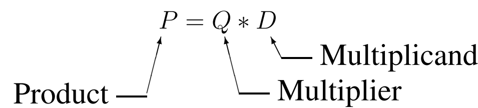
- Once the datapath and the control unit are synthesized and simulated the can be connected them together to form the complete word-serial multiplication processor.
- The following waveforms were obtained during simulation of the complete multiplication processor:



- Note that the processor correctly multiplies $53_H \times 65_H = 20\mathrm{BF}_H$.
- More exhaustive tests are needed to confirm the correctness of the design.

13.10 Appendix: Multiplication methods

Consider multiplication of two numbers represented by numerals Q and D



If the **multiplier**, Q, and the **multiplicand**, D, are represented by the n–digit and m–digit numerals, respectively, then the **product**, P, is represented by the $(n+m-1)$–digit numeral as follows:

$$\begin{aligned} Q_{n-1:0} &= [\, q_{n-1} \ldots q_1\, q_0 \,] \\ D_{m-1:0} &= [\, d_{m-1} \ldots d_1\, d_0 \,] \\ P_{k-1:0} &= [\, a_{k-1}\ \ldots\ a_1\ a_0 \,]\ ;\quad k = m+n \end{aligned}$$

If we use the above notation, then the product numeral, $P_{k-1:0}$, can be elegantly expressed as a matrix product of the multiplier numeral, $Q_{n-1:0}$, and the Sylvester **resultant matrix** of the multiplicand $\langle D_{m-1:0}\rangle_{n-1}$

$$P_{k-1:0} = Q_{n-1:0} \cdot \langle D_{m-1:0}\rangle_{n-1} \tag{13.1}$$

The Sylvester resultant matrix, which also known as the **convolution matrix**, is formed from the shifted numeral $D_{m-1:0}$ in the following way:

$$\langle D_{m-1:0}\rangle_{n-1} = \underbrace{\begin{bmatrix} d_{m-1} & d_{m-2} & \cdots & d_0 & & & \\ & d_{m-1} & d_{m-2} & \cdots & d_0 & \mathbf{0} & \\ & \mathbf{0} & \ddots & \ddots & & \ddots & \\ & & & d_{m-1} & d_{m-2} & \cdots & d_0 \end{bmatrix}}_{n+m-1} \Bigg\updownarrow n \tag{13.2}$$

The angle brackets have been used to denote the resultant matrix. The bold **0**s in the resultant of eqn (13.2), represent appropriate triangles of zeroes.

In general, eqn (13.1) can be thought of as a parallelised description of the multiplication algorithm of two numerals.
Each row of the resultant represents a shifted numeral $D_{m-1:0}$ which is multiplied by the respective digit of the multiplier.
Subsequently, the columns of the resultant are summed up to give the 'pseudo-digits' of the result.
The carry propagation is neglected at this level of the multiplication algorithm.
In this sense that the carry is incorporated into the 'pseudo-digits' of the result.

Example

Consider multiplication of the following decimal numerals

$$P_{5:0} = Q_{2:0} * D_{3:0} = \underbrace{3\,2\,1}_{Q_{2:0}} * \underbrace{1\,2\,3\,4}_{D_{3:0}}$$

This multiplication operation can be described in the following matrix form:

$$\begin{aligned} P_{5:0} &= Q_{2:0} \cdot \langle D_{3:0} \rangle_2 \\ \begin{bmatrix} 3 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 & 4 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 0 \\ 0 & 0 & 1 & 2 & 3 & 4 \end{bmatrix} &= \sum_{\text{cl}} \begin{bmatrix} 3 & 6 & 9 & 12 & 0 & 0 \\ 0 & 2 & 4 & 6 & 8 & 0 \\ 0 & 0 & 1 & 2 & 3 & 4 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 8 & 14 & 20 & 11 & 4 \end{bmatrix} \end{aligned}$$

where $\sum_{\text{cl}}$ denotes the column-wise summation, that is, addition of rows of the matrix.

The product numeral $P_{5:0}$ contains "pseudo-digits", that is, digits which are greater than the base $b = 10$. In order to obtain the 'proper' digits, the following carry propagation operation is required:

$$\begin{array}{rccccccl} & 3 & 8 & 4 & 0 & 1 & 4 & \\ & 0 & 1 & 2 & 1 & 0 & 0 & \\ \hline P_{5:0} = & 3 & 9 & 6 & 1 & 1 & 4 & = 321 * 1234 \end{array}$$

The above example can easily be extended into a **generic multiplication algorithm**. If we combine eqns (13.1) and (13.2) we obtain the following expression describing the first step of a generic multiplication algorithm:

$$P_{k-1:0} = \sum_{\text{cl}} \begin{bmatrix} q_{n-1} \cdot [d_{m-1} & & \cdots & & d_1 & d_0 & 0 & \cdots & 0] \\ \vdots & \vdots & \ddots & & & \ddots & \ddots & \ddots & \vdots \\ q_1 \cdot [\;0 & \cdots & d_{m-1} & & \cdots & & d_1 & d_0 & 0] \\ q_0 \cdot [\;0 & \cdots & 0 & d_{m-1} & & \cdots & & d_1 & d_0] \end{bmatrix} \quad (13.3)$$

Denoting the products of individual digits of multiplier and the multiplicand by

$$r_{ij} = q_i \cdot d_j$$

we obtain from eqn (13.3)

$$P_{k-1:0} = \sum_{\text{cl}} \begin{bmatrix} r_{n-1,m-1} & & \cdots & & r_{n-1,1} & r_{n-1,0} & 0 & \cdots & 0 \\ \vdots & \ddots & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & r_{1,m-1} & & \cdots & & r_{11} & r_{10} & 0 \\ 0 & \cdots & 0 & r_{0,m-1} & & \cdots & & r_{01} & r_{00} \end{bmatrix} \quad (13.4)$$

In a binary case, when, in general, the digits $q_i, d_j \in \{-1, 0, +1\}$, the elementary products are also, $r_{ij} \in \{-1, 0, +1\}$.

In the second step of a generic multiplication algorithm, the elementary products in the matrix (13.4), are to be summed up in columns. For a purely binary case, this operation yields the counts of ones in each column.

Finally, the column sums must be added together, in a step which involves carry propagation.

The above generic multiplication algorithm can be implemented in at least the following ways:

The word-serial algorithm. This is probably the most popular algorithm in which the final product is formed by adding rows of the matrix (13.4) one by one. In practice, we employ a single m–bit adder, and the partial products are shifted one position right between n successive steps of the multiplication process.

Parallel algorithms. These are the fastest implementations of the multiplication operation. In this case we use enough adders (approximately n m-bit adders), so that the multiplication operation is performed in **one** step.

Two groups of algorithms belonging to this class are called the matrix method, and the **Wallace-tree** method, respectively.

The column-serial algorithms. In this case, first elementary products in a column of the matrix (13.4) are added serially, and then operation is repeated for the next more significant column. In other words, there is a single 1-bit adder, and every addition operation is performed in m steps. This is clearly the slowest method, but the amount of hardware required is minimal.

The distributed arithmetic algorithms. We present details of such algorithms in the subsequent sections.

13.11 The Booth's multiplier

In the context of multiplication it is often convenient to convert a two's-complement number into a signed-digit form. The multiplication method using the multiplier in the signed-digit form is known as the Booth's method.

Let

$$Q_{n-1:0} = [\,\overline{q}_{n-1}\ \cdots\ q_1\ q_0\,]\ , \qquad \text{where} \quad q_i \in \{0, 1\}\ .$$

Consider now the following identity:

$$q_i = 2q_i - q_i$$

Using the above identity it is now possible to obtain another numeral representation of the number q in the following way:

$$\begin{array}{rcl|ccccc|l}
 & & & 2^{n-1} & 2^{n-2} & \cdots & 2^1 & 2^0 & \\
\hline
Q_{n-1:0} & = & [& \overline{q}_{n-1} & q_{n-2} & \cdots & q_1 & q_0 &] \\
 & = & [& -q_{n-1} & 2q_{n-2} - q_{n-2} & \cdots & 2q_1 - q_1 & 2q_0 - q_0 &] \\
 & = & [& -q_{n-1} + q_{n-2} & -q_{n-2} + q_{n-3} & \cdots & -q_1 + q_0 & -q_0 + 0 &] \\
\hline
\hat{Q}_{n-1:0} & = & [& \hat{q}_{n-1} & \hat{q}_{n-2} & \cdots & \hat{q}_1 & \hat{q}_0 &]
\end{array}$$

where

$$\hat{q}_i = -q_i + q_{i-1}\ , \quad \text{or,} \quad \begin{array}{|cc|c|}\hline q_i & q_{i-1} & \hat{q}_i \\ \hline 0 & 0 & 0 \\ 0 & 1 & +1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \\ \hline \end{array}\ , \quad \text{and} \quad q_{-1} = 0\ . \tag{13.5}$$

Hence, after re-coding, the Booth's multiplier is

$$\hat{Q}_{n-1:0} = [\,\hat{q}_{n-1}\ \cdots\ \hat{q}_1\ \hat{q}_0\,]\ , \qquad \text{where} \quad \hat{q}_i \in \{-1, 0, +1\}\ .$$

Obviously the value of the multiplier has not changed in the re-coding process and we have:

$$q = -q_{n-1}2^{n-1} + \sum_{i=0}^{n-2} q_i 2^i = \sum_{i=0}^{n-1} \hat{q}_i 2^i$$

Example

$$\begin{aligned}(\bar{1}\ 0\ 0\ 1\ 0\ 1\ 1)_2 &= -2^6 + 2^3 + 2 + 1 = -53 \\ (\bar{1}\ 0\ 1\ \bar{1}\ 1\ 0\ \bar{1})_2 &= -2^6 + 2^4 - 2^3 + 2^2 - 1 = -53\end{aligned}$$