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 $D$:

$$P = Q \times D$$

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:

$$P[0] = 0; \quad q_{-1} = 0;$$

for $(i = 0; \quad i < n; \quad i++)$

$$\hat{q}_i = -q_i + q_{i-1};$$

$$P[i + 1] = (P[i] + \hat{q}_i \cdot D) \cdot 2^{-1};$$

- 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 top-level structure of the processor

- The top level schematic of the \textit{wsm} processor consists of two components, namely, the datapath and the control unit:
  - Two symbols, \texttt{dpath} and \texttt{cntp}, representing the datapath and the control unit components, respectively, are created and instantiated into the top level schematic \texttt{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 \textit{ready} signal, \( rdy \).
  - The \textit{clock} and \textit{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, $\text{asr}(F) = F/2$, 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 $Sc = n - 1$.

A.P. Papliński

13–5

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 ($AQ$). This is indicated by the signal $rdy$ being asserted.

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

\[ D = (101101)_2 = -19, \quad QQ = (101001)_2 = -23 \]

<table>
<thead>
<tr>
<th>( A[i], D )</th>
<th>( Q )</th>
<th>( q_{i-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>101101</td>
<td>1</td>
</tr>
<tr>
<td>( F )</td>
<td>010011</td>
<td></td>
</tr>
<tr>
<td>( A[1] )</td>
<td>101101</td>
<td>1</td>
</tr>
<tr>
<td>( F )</td>
<td>110110</td>
<td>1</td>
</tr>
<tr>
<td>( F )</td>
<td>010000</td>
<td>101</td>
</tr>
<tr>
<td>( F )</td>
<td>110101</td>
<td>1010</td>
</tr>
<tr>
<td>( F )</td>
<td>001101</td>
<td>1010</td>
</tr>
</tbody>
</table>

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

A.P. Papliński

13–7

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.

<table>
<thead>
<tr>
<th>multiplicant register, D</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( D \leftarrow D ) \quad \text{nop}</td>
</tr>
<tr>
<td>1</td>
<td>( D \leftarrow DD ) \quad \text{load}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>partial product register, A</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( A \leftarrow A ) \quad \text{nop}</td>
</tr>
<tr>
<td>1</td>
<td>( A \leftarrow F/2 ) \quad \text{ldAshr}</td>
</tr>
<tr>
<td>2, 3</td>
<td>( A \leftarrow 0 ) \quad \text{reset}</td>
</tr>
</tbody>
</table>

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

| The partial product register performs three operations. |

| The \text{ldAshr} operation, that is, \( "\text{load arithmetically shifted right signal vector } F(3..0) \text{ into } A" \), can be more precisely described as \( A \leftarrow (F(3), F(3..1)) \) |

| 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: \( Q \leftarrow (F(0), Q(3..0)) \) |

A.P. Papliński

13–8
### 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 known 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.
- In 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.

```
-- cnu 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);
      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";
```

A.P. Papliński

13–11
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:

<table>
<thead>
<tr>
<th>Signals</th>
<th>Value</th>
<th>0ns</th>
<th>100ns</th>
<th>200ns</th>
<th>300ns</th>
<th>400ns</th>
<th>500ns</th>
<th>600ns</th>
<th>700ns</th>
<th>800ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>rst</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdy</td>
<td>'0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zI</td>
<td>'0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nat_st</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stt</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>15</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>6A</td>
<td>6A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dop</td>
<td>'0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aop</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qop</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sop</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 they can be connected 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 $53_{16} \times 65_{16} = 20BF_{16}$.
- 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 $D$

\[
P = Q \cdot D
\]

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:

\[
Q_{n-1:0} = [q_{n-1} \ldots q_1 q_0]
D_{m-1:0} = [d_{m-1} \ldots d_1 d_0]
\]

\[
P_{k-1:0} = [a_{k-1} \ldots a_1 a_0]; \quad k = m + n
\]

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 $\langle D_{m-1:0} \rangle_{n-1}$

\[
P_{k-1:0} = Q_{n-1:0} \cdot \langle D_{m-1:0} \rangle_{n-1}
\]

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:

\[
\langle D_{m-1:0} \rangle_{n-1} = \begin{bmatrix}
    d_{m-1} & d_{m-2} & \cdots & d_0 \\
    d_{m-1} & d_{m-2} & \cdots & d_0 \\
    0 & \ddots & \ddots & \vdots \\
    0 & \cdots & d_{m-1} & d_{m-2} & \cdots & d_0
\end{bmatrix}
\]

The angle brackets have been used to denote the resultant matrix. The bold 0s 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

\[ P_{5:0} = Q_{2:0} \cdot D_{3:0} = 321 \cdot 1234 \]

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

\[
P_{5:0} = Q_{2:0} \cdot \langle D_{3:0} \rangle_2
\]

\[
\begin{bmatrix}
3 & 2 & 1 \\
0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 2 & 3 & 4
\end{bmatrix}
= \sum_{cl} \begin{bmatrix}
3 & 6 & 9 & 12 & 0 & 0 \\
0 & 2 & 4 & 6 & 8 & 0 \\
0 & 0 & 1 & 2 & 3 & 4
\end{bmatrix}
= \begin{bmatrix}
3 & 8 & 14 & 20 & 11 & 4
\end{bmatrix}
\]

where \( \sum_{cl} \) 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:

\[
P_{5:0} = 3\,9\,6\,1\,1\,4 = 321 \times 1234
\]

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:

\[
P_{k-1:0} = \sum_{cl} \begin{bmatrix}
q_{n-1} [d_{m-1}] \cdots d_1 \, d_0 \cdots 0 \\
\vdots \, \cdots \, \cdots \, \cdots \, \cdots \, \cdots \\
q_0 \, 0 \cdots d_{m-1} \cdots d_1 \, d_0 \cdots 0 \\
\end{bmatrix}
\]

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)

\[
P_{k-1:0} = \sum_{cl} \begin{bmatrix}
r_{n-1,m-1} \cdots r_{n-1,1} \cdots r_{n-1,0} \cdots 0 \\
\vdots \, \cdots \, \cdots \, \cdots \, \cdots \, \cdots \\
0 \cdots r_{1,m-1} \cdots r_{1,1} \cdots r_{1,0} \cdots 0 \\
\end{bmatrix}
\]

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

\[
Q_{n-1:0} = \begin{bmatrix} \hat{q}_{n-1} & \cdots & q_1 & q_0 \end{bmatrix}, \quad \text{where} \quad q_i \in \{0, 1\}.
\]

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:

\[
Q_{n-1:0} = \begin{bmatrix} 2^{n-1} & 2^{n-2} & \cdots & 2^1 & 2^0 \\ 2q_{n-1} & 2q_{n-2} & \cdots & q_1 & q_0 \\ -q_{n-1} & 2q_{n-2} - q_{n-2} & \cdots & 2q_1 - q_1 & 2q_0 - q_0 \\ -q_{n-1} + q_{n-2} & -q_{n-2} + q_{n-3} & \cdots & -q_1 + q_0 & -q_0 + 0 \end{bmatrix}
\]

\[
\hat{Q}_{n-1:0} = \begin{bmatrix} \hat{q}_{n-1} & \hat{q}_{n-2} & \cdots & \hat{q}_1 & \hat{q}_0 \end{bmatrix}
\]

where
\[ \hat{q}_i = -q_i + q_{i-1}, \quad \text{or}, \quad q_i \quad q_{i-1} \quad \hat{q}_i \\
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 1 & +1 \\
1 & 0 & -1 \\
1 & 1 & 0 \\
\end{array} \]

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

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

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

\[ (1001011)_2 = -2^6 + 2^3 + 2 + 1 = -53 \]
\[ (10111011)_2 = -2^6 + 2^4 - 2^3 + 2^2 - 1 = -53 \]