

7 Multipliers and their VHDL representation

7.1 Introduction to arithmetic algorithms

If a is a **number**, then a vector of **digits**

$A_{n-1:0} = [a_{n-1} \dots a_1 a_0]$ is a **numeral** representing the number in the radix- b system:

$$a = \underbrace{(a_{n-1} \dots a_1 a_0)}_{A_{n-1:0}}_b = \sum_{i=0}^{n-1} a_i b^i = [a_{n-1} \dots a_1 a_0] \begin{bmatrix} b^{n-1} \\ \vdots \\ b^1 \\ b^0 \end{bmatrix} \quad (7.1)$$

Eqn (7.1) says that a number is a sum of products of digits and **weights**.

Examples

The two's complement system

$$a = \underbrace{[\bar{1} 0 1 1 0]}_{A_{4:0}} \begin{bmatrix} 2^4 \\ 2^3 \\ 2^2 \\ 2^1 \\ 2^0 \end{bmatrix} = -2^4 + 2^2 + 2^1 = -10 ; \quad \begin{matrix} a_i \in \{0, 1\} \\ b = 2 \end{matrix}$$

Note, that in the two's complement system the most significant digit is negative (or zero). The bar over the digit is its sign not the logic complement.

A signed-digit ternary system

$$a = [\underbrace{2 \bar{1} 0 1}_{A_{3:0}}] \begin{bmatrix} 3^3 \\ 3^2 \\ 3^1 \\ 3^0 \end{bmatrix} = 2 \cdot 3^3 - 3^2 + 1 = 46 ; \quad a_i \in \{\pm 2, \pm 1, 0\} \\ b = 3$$

A signed-digit binary system

$$a_i \in \{-1, 0, 1\} ; \quad b = 2$$

$$a = (\underbrace{-1 \ 1 \ 0 \ -1}_{A_{3:0}})_2 = -2^3 + 2^2 - 1 = -5 ;$$

7.2 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

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

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:

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

where

$$\hat{q}_i = -q_i + q_{i-1} \text{ , or, } \begin{array}{|c|c|c|} \hline q_i & q_{i-1} & \hat{q}_i \\ \hline 0 & 0 & 0 \\ 0 & 1 & +1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \\ \hline \end{array} \text{ , and } q_{-1} = 0 \text{ .} \quad (7.2)$$

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

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

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

$$(\bar{1} 0 0 1 0 1 1)_2 = -2^6 + 2^3 + 2 + 1 = -53$$

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

7.3 Multiplication methods

Consider multiplication of two numbers represented by numerals Q and D

$$\begin{array}{c}
 P = Q * D \\
 \swarrow \quad \searrow \quad \nwarrow \\
 \text{Product} \quad \quad \quad \text{Multiplier} \quad \quad \text{Multiplicand}
 \end{array}$$

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:

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

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

$$P_{k-1:0} = Q_{n-1:0} \cdot \langle D_{m-1:0} \rangle_{n-1} \quad (7.3)$$

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:

$$\langle D_{m-1:0} \rangle_{n-1} = \begin{bmatrix} d_{m-1} & d_{m-2} & \cdots & d_0 & & & \\ & d_{m-1} & d_{m-2} & \cdots & d_0 & \mathbf{0} & \\ & \mathbf{0} & \ddots & \ddots & & \ddots & \\ & & & d_{m-1} & d_{m-2} & \cdots & d_0 \end{bmatrix} \quad \begin{matrix} \uparrow \\ n \\ \downarrow \end{matrix} \quad (7.4)$$

$\longleftarrow n+m-1 \longrightarrow$

The angle brackets have been used to denote the resultant matrix. The bold **0**s in the resultant of eqn (7.4), represent appropriate triangles of zeroes.

In general, eqn (7.3) 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

$$P_{5:0} = Q_{2:0} * D_{3:0} = \underbrace{321}_{Q_{2:0}} * \underbrace{1234}_{D_{3:0}}$$

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:

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

The above example can easily be extended into a **generic multiplication algorithm**. If we combine eqns (7.3) and (7.4) we obtain the following expression describing the first step of a generic multiplication algorithm:

$$P_{k-1:0} = \sum_{cl} \begin{bmatrix} q_{n-1} \cdot [d_{m-1} & \cdots & d_1 & d_0 & 0 & \cdots & 0] \\ \vdots & \ddots & & & & \ddots & \ddots & \vdots \\ q_1 \cdot [0 & \cdots & d_{m-1} & \cdots & & d_1 & d_0 & 0] \\ q_0 \cdot [0 & \cdots & 0 & d_{m-1} & \cdots & & d_1 & d_0] \end{bmatrix} \quad (7.5)$$

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

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

we obtain from eqn (7.5)

$$P_{k-1:0} = \sum_{cl} \begin{bmatrix} r_{n-1,m-1} & \cdots & r_{n-1,1} & r_{n-1,0} & 0 & \cdots & 0 \\ \vdots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & r_{1,m-1} & \cdots & & r_{11} & r_{10} & 0 \\ 0 & \cdots & 0 & r_{0,m-1} & \cdots & & r_{01} & r_{00} \end{bmatrix} \quad (7.6)$$

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 (7.6), 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 (7.6) 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 (7.6) 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.

7.4 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.

Let the product P be

$$P = Q * D$$

Product
Multiplier
Multiplicand

and the **multiplier**, $Q = \{q_i\}_{n-1:0}$, and the **multiplicand**, $D = \{d_i\}_{m-1:0}$ be represented in the two’s-complement system by the n -digit and m -digit numerals, respectively. The **product**, P , is represented by the $(n+m-1)$ -digit numeral.

The one-bit Booth’s multiplication algorithm can be described by the following pseudo-code:

$P[0] = 0 ; \quad q_{-1} = 0 ;$	q_i	q_{i-1}	$\hat{q}_i$
for $(i = 0 ; \quad i < n ; \quad i++) \{$	0	0	0
$\hat{q}_i = -q_i + q_{i-1} ;$	0	1	+1
$P[i + 1] = (P[i] + \hat{q}_i \cdot D) \cdot 2^{-1} ;$	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 7–4 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 **data-path** and a specification of the **control unit** given in a form of a flow-chart of operations performed by the word-serial multiplication processor.

7.5 The top-level structure of the processor

The top level schematic of the WSM processor consists of two components, namely, the datapath and the control unit, and its structure is given in Figure 7–1.

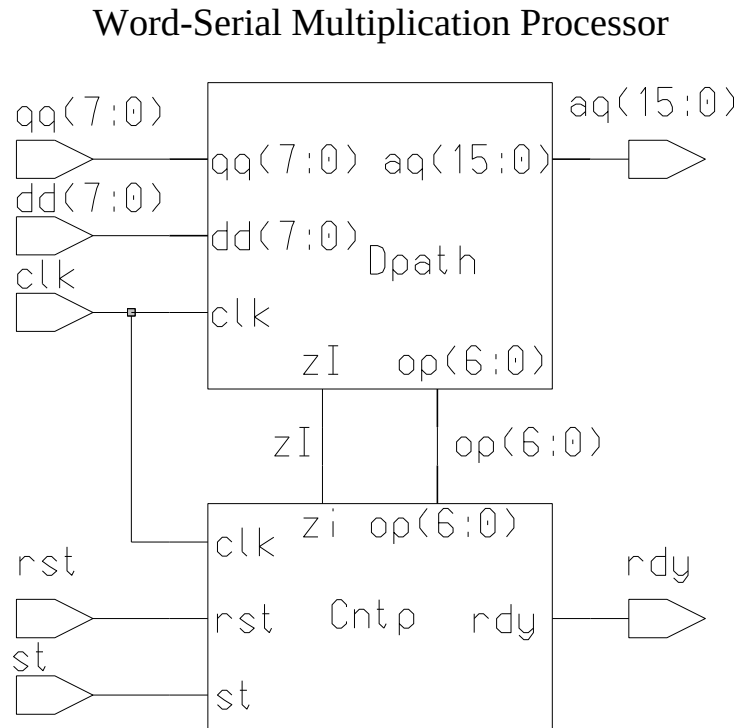


Figure 7–1: The top level schematic of the word-serial multiplication processor

Two symbols, **dpath** and **cntp**, representing the datapath and the control unit components, respectively, are created and instantiated into the top level schematic **wsM** as in Figure 7–1. The brief description of the schematic is as follows.

- 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 , being asserted.
- Refer to the flow-chart of operations for details.
- 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.

F — the ALU (Adder/Subtractor) performing operations

$$F \leq A \pm D, \text{ or } F \leq A.$$

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.

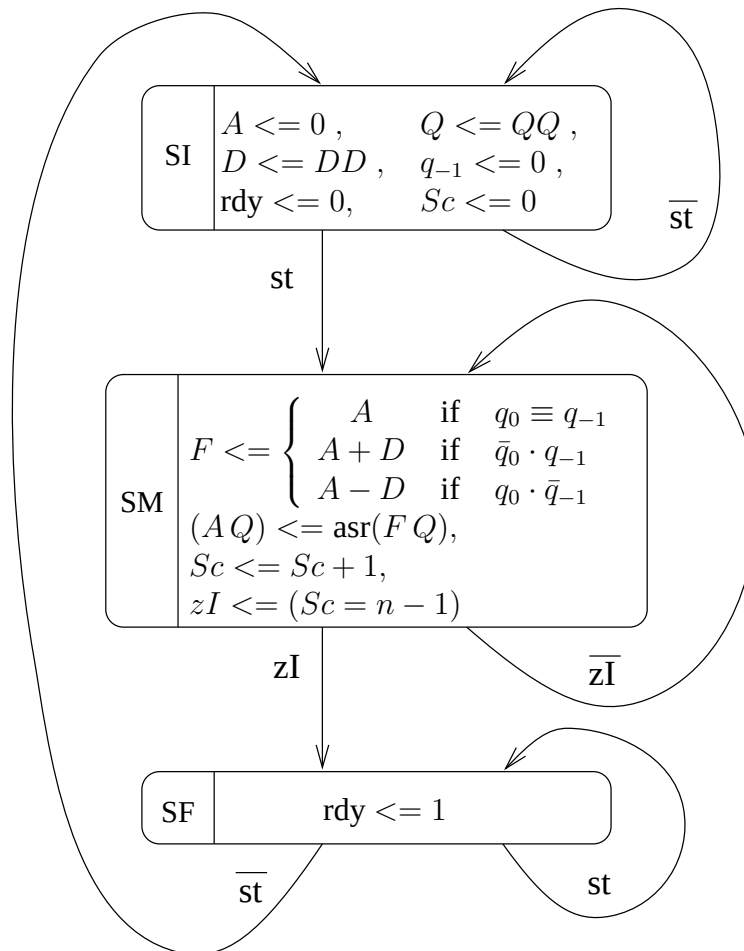
The least significant bit, f_0 , is shifted into the Q-register.

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

All registers are triggered by the positive-edge of the clock signal clk , and their operations are controlled by **op**-code signals which will be examined in detail in the subsequent sections.

7.7 State diagram of the control unit

The flow-chart of operations performed by the control unit of the word-serial multiplier using the Booth's method, that is, the state (transition) diagram describing the control unit as the finite state machine, is presented in Figure 7–3.



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.

(FQ) is a concatenation of the outputs from the adder and from the Q register. (FQ) 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.

Figure 7–3: The flow-chart of the word-serial multiplier using the one-bit Booth's method

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.

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

7.8 Numerical example

The operations performed by the multiplication processor can be best explained by a numerical example presented in Figure 7–4. This example of the one-bit Booth's multiplication algorithm should be examined in conjunction with the datapath and the flow-chart of the control unit. The multiplication steps are as follows:

- 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 $A - Q$.

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

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

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

Figure 7–4: A one-bit, word-serial Booth's multiplication algorithm. A numerical example.

7.9 Details of hardware implementation – Operations of the data-path blocks

The top-level structure of the multiplication processor was already discussed in sec.7.5 and presented in Figure 7–1.

In the subsequent sections we will discuss details of hardware implementation of the datapath and the control unit.

A detailed block-diagram, or schematic of the datapath has been presented in Figure 7–2.

Before we can attempt the detailed design of the datapath we have to specify operations performed by its blocks in a form of the function tables as we did it for typical sequential and combinatorial blocks in previous sections.

This can be done by examination of the operations described in the flow-chart of Figure 7–3.

With every operation which is required to perform the multiplication steps as specified in the flow-chart, we must now associate an operation of a specific functional block of the datapath.

The result of this examination is presented in Table 3.

For each block of the datapath we specify an operation (function) table which lists all operations performed by the block.

Every operation is assigned a specific operation code (an op-code) using mnemonic names like `nop` which stands for ‘no operation’.

Binary values of op-codes are also given in the function tables to fix our attention, but they can be changed during the design process in order to simplify the internal structure of the components of the datapath.

multiplicand register, D			partial product register, A		
Dop	operation		Aop	operation	
0	$D \leftarrow D$	nop	0	$A \leftarrow A$	nop
1	$D \leftarrow DD$	load	1	$A \leftarrow F/2$	ldAshr
			2, 3	$A \leftarrow 0$	reset

multiplier register, Q		
Qop	operation	
0	$Q \leftarrow Q$	nop
1	$Q \leftarrow \text{shr}(F(0), Q)$	shrQ
2, 3	$Q \leftarrow (QQ \ \& \ 0)$	load

step counter, Sc		
Sop	operation	
0	$Sc \leftarrow Sc$	nop
1	$Sc \leftarrow Sc+1$	count
2, 3	$Sc \leftarrow 0$	reset

ALU, F		
Fop	operation	
0, 3	$F \leftarrow A$	pass
1	$F \leftarrow A + B$	add
2	$F \leftarrow A - B$	subtract

Table 3: Specification of operations performed by the components of the data-path

Some easy-to-miss details of operation need to be repeated from sec.7.5:

- The ldAshr operation, that is, “load arithmetically shifted right signal vector $F(3..0)$ into A”, can be more precisely described as

$$A \leftarrow (F(3) \ \& \ F(3..1))$$

- The Q register is an $(n+1)$ -bit register, e.g., $Q(3..-1)$ (warning: in VHDL negative subscripts are not allowed). During the load operation the additional position $Q(-1)$ should be cleared.

The shift operation can be alternatively described as:

$$Q \leftarrow (F(0) \ \& \ Q(3..0))$$

- 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:

$$Fop = Q(0:-1).$$

All other op-code signals can be, for convenience, collected in one 7-bit op-code word:

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

7.10 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 tools of the CAD package and interconnect blocks as in Figure 7–2.
 - 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.

7.11 The control unit

The first step in designing the control unit is the re-interpretation of the multiplication flowchart from Figure 7–3 which describes details of the algorithm as an equivalent **state diagram** of the control unit in which we refer to operations performed by the components of the datapath.

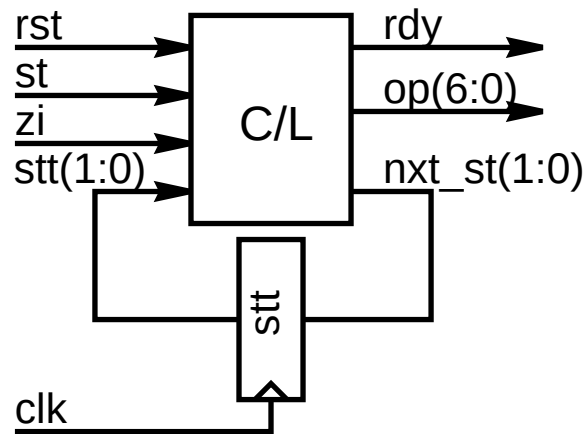


Figure 7–5: The control unit as a finite state machine

The control unit is a finite state machine as in Figure 7–5 consisting of the state register, *stt*, and combinational logic, *C/L*, implementing the state transition and output functions. The state register stores the codes of three states, **SI**, **SM**, **SF** encoded using two flip-flops.

The state diagram describes transitions between the states, and the output signals generated by the control unit. These output signals generated by the control unit are primarily the op-codes which specify operations performed by the blocks of the datapath.

7.11.1 The VHDL program for the control unit

The behavioural specification of the control unit can be given by the following VHDL program. This 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
    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, nxt_st : states := SI ;

    SIGNAL stt, nxt_st : 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, count : 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 <= nxt_st ;
        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
        nxt_st <= 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 nxt_st <= SM ; END IF ;

            WHEN SM =>
                Aop <= ldAshr ;
                Qop <= shrQ ;
                Sop <= count ;
                IF ( zi = '1' ) THEN nxt_st <= SF ; END IF ;

```

```
    WHEN OTHERS =>      --- when SF
        rdy <= '1' ;
        IF ( st = '0' ) THEN nxt_st <= 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 ;
```

The ‘clocked’ process `clkd` describes the state D flip-flops of the control unit.

The flip-flops are asynchronously reset when `rst = 0`.

The state machine process `stm` describes combinatorial logic of the “next address” (excitation) circuit and the “output signals” circuit of the control unit.

After the control unit is compiled and synthesized it is important to verify its behaviour by simulation. The results of the simulation of the control unit could be of the form as presented in Figure 7–6.

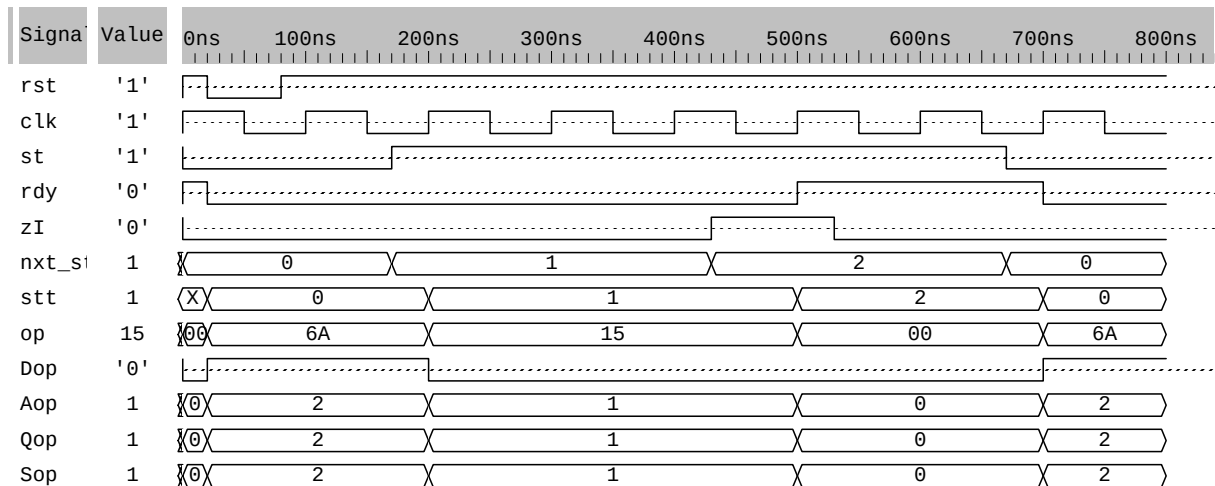


Figure 7–6: Simulation waveforms for the control unit

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.

7.12 The complete word-serial multiplication processor

Once the datapath and the control unit are synthesized and simulated the can be connected together to form the complete word-serial multiplication processor as in Figure 7–1. Then the processor should be tested by simulation.

An example of simulation results is presented in Figure 7–7.

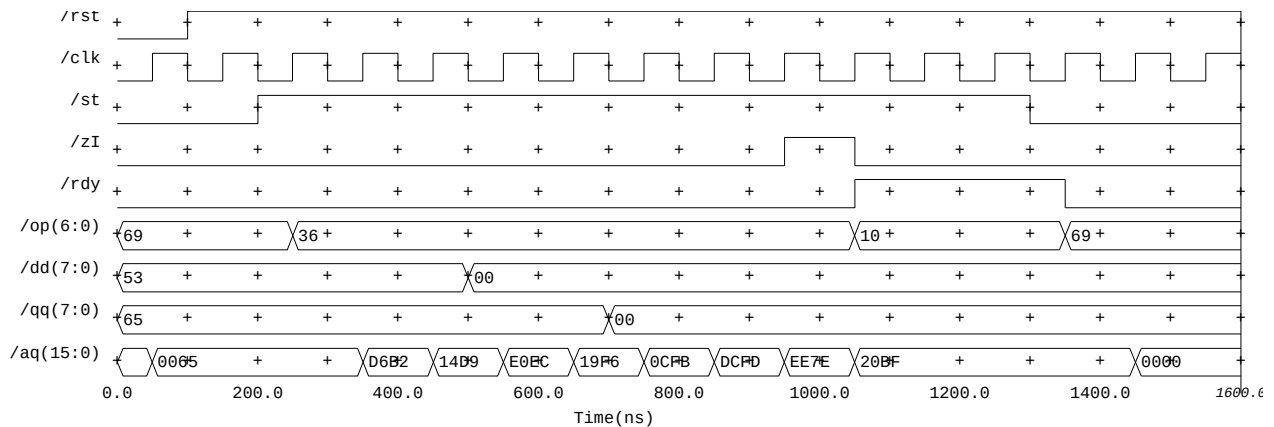


Figure 7–7: Simulation waveforms for the word-serial multiplication processor.

Note that $53_H \times 65_H = 20BF_H$

Note that the processor correctly multiplies two positive numbers. It should be also tested for negative numbers.