A-PDF Split DEMO : Purchase from www.A-PDF.com to remove the watermark

11 Sequential Multiplier

This chapter discusses the design, simulation and prototyping of a sequential multiplier. The multiplication process will be done by the shift-and-add sequential multiplication procedure. After a discussion of the multiplication method used, we present the details and interfacing of our design. Then the multiplier will be partitioned into its data and control parts, and each part will be designed separately. The completed design will be simulated in Verilog and tested by programming the FLEX 10K device of the UP2 board.

11.1 Sequential Multiplier Specification

The project is the design of a 2-bit sequential multiplier, with 8-bit *A* and *B* inputs and a 16-bit result. The block diagram of the circuit to be designed is shown in Figure 11.1. This multiplier has an 8-bit bi-directional I/O for inputting its *A* and *B* operands, and outputting its 16-bit output one byte at a time.

Multiplication begins with the *start* pulse, and the *databus* will contain operands *A* and *B* in two consecutive clock pulses. After accepting these data inputs, the multiplier begins its multiplication process and when it is completed, it starts sending the result out on the *databus*. When the least-significant byte is placed on *databus*, the *Lsb_out* output is issued, and for the most-significant byte, *msb_out* is issued. When both bytes are outputted, *done* becomes **1**, and the multiplier is ready for another set of data.

The multiplexed bi-directorial *databus* is used to reduce the total number of pins of the multiplier.

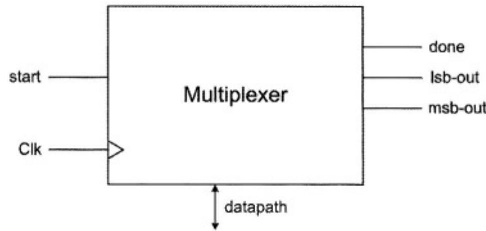

Figure 11.1 Multiplier Block Diagram

11.2 Shift-and-Add Multiplication

When designing multipliers there is always a compromise to be made between how fast the multiplication process is done and how much hardware we are using for its implementation.

A simple multiplication method that is slow, but efficient in use of hardware is the shift-and-add method. In this method, depending on bit i of operand A, either operand B is added to the collected partial result and then shifted to the right (when bit i is **1**), or (when bit i is **0**) the collected partial result is shifted one place to the right without being added to B.

This method is justified by considering how binary multiplication is done manually. Figure 11.2 shows manual multiplication of two 8-bit binary numbers.

We start considering bits of A from right to left. If a bit value is **0** we select 00000000 to be added with the next partial product, and if it is a **1**, the value of B is selected. This process repeats, but each time 00000000 or B is selected, it is written one place to the left with respect to the previous value. When all bits of A are considered, we add all calculated values to come up with the multiplication results.

Understanding hardware implementation of this procedure becomes easier if we make certain modifications to this procedure. First, instead of having to move our observation point from one bit of A to another, we put A in a shift-register, always observe its right-most bit, and after every calculation, we move it one place to the right, making its next bit accessible.

Second, for the partial products, instead of writing one and the next one to its left, when writing a partial product, we move it to the right as we are writing it, and the next one will not have to be shifted.

Finally, instead of calculating all partial products and at the end adding them up, when a partial product is calculated, we add it to the previous partial result and write the newly calculated value as the new partial result.

Therefore, if the bit of A that is being observed is **0,** *00000000* is to be added to the previously calculated partial result, and the new value should be shifted one place to the right. In this case, since the value being added to the partial result is *00000000,* adding is not necessary, and only shifting the partial result is sufficient. This process is called *shift*. However, if bit of A being observed is **1**, B is to be added to the previously calculated partial result, and

the calculated new sum must be shifted one place to the right. This is called *add-and-shift.*

Repeating the above procedure, when all bits of A are shifted out, the partial result becomes the final multiplication result. We use a 4-bit example to clarify the above procedure. As shown in Figure 11.3, A = **1001** and B = **1101** are to be multiplied. Initially at time 0, A is in a shift-register with a register for partial results *(P)* on its left.

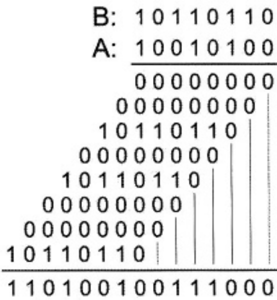

Figure 11.2 Manual Binary Multiplication

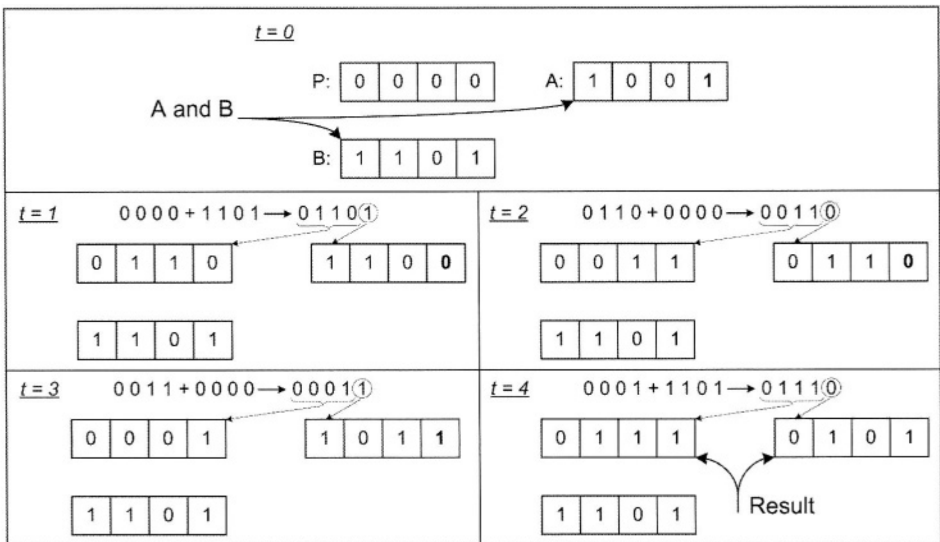

Figure 11.3 Hardware Oriented Multiplication Process

At the time 0, because *A[0]* is **1**, the partial sum of $B + P$ is calculated. This value is **01101** (shown in the upper part of time 1) and has 5 bits to consider carry. The right most bit of this partial sum is shifted into the *A* register, and the other bits replace the old value of *P*. When *A* is shifted, **0** moves into the *A[0]* position. This value is observed at time 1. At this time, because *A[0]* is **0**, **0000** + *P* is calculated (instead of $B + P$). This value is **00110**, the right most bit of which is shifted into *A,* and the rest replace *P*. This process repeats 4 times, and at the end of the 4th cycle, the multiplication result becomes available in *P* and *A*. The least significant 4 bits of the result are in *A* and the most-significant bits are in *P*. The example used here performed 9*13 and 117 was obtained as the result of this operation.

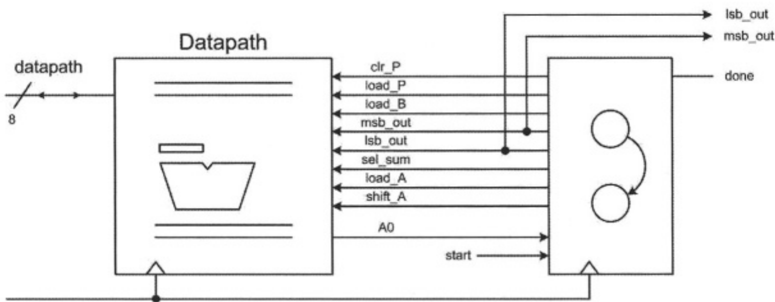

Figure 11.4 Data and Control Parts

11.3 Sequential Multiplier Design

The multiplication process discussed in the previous section justifies the hardware implementation that is being discussed here.

11.3.1 Control Data Partitioning

The multiplier has a datapath and a controller. The data part consists of registers, logic units and their interconnecting busses. The controller is a state machine that issues control signals for control of what gets clocked into the data registers.

As shown in Figure 11.4, the data path registers and the controller are triggered with the same clock signal. On the rising edge of a clock the controller goes into a new state. In this state, several control signals are issued, and as a result the components of the datapath start reacting to these signals. The time given for all activities of the datapath to stabilize is from one edge of the clock to another. Values that are propagated to the inputs of the datapath registers are clocked into these register with every clock edge.

11.3.2 Multiplier Datapath

Figure 11.5 shows the datapath of the sequential multiplier. As shown, *P* and *B* are 8-bit registers and *A* is an 8-bit shift-register. An adder, a multiplexer and a tri-state buffer constitute the other components of this datapath.

Control signals that are outputs of the controller and inputs of the datapath (Figure 11.4), are shown in bold in Figure 11.5 next to the data component that they control. These control signals control register clocking, bus assignments and logic unit output selections.

The input *databus* connects to the inputs of *A* and *B* to load multiplier and multiplicand into these registers. This bi-directional bus is driven by the output of *P* through an octal tri-state buffer, and by the tri-state output of *A*. This bi-directional bus is driven by the output of *P* through an actual tri-state buffer, and by the tri-state output of *A*. These tri-states become active when multiplication result is ready.

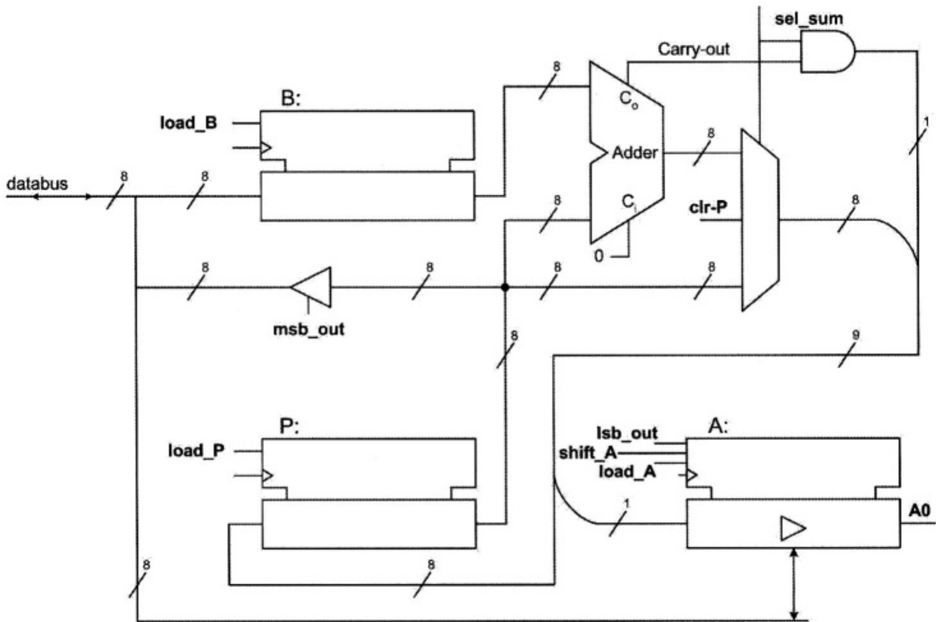

Figure 11.5 Multiplier Block Diagram

The output from *B* and *P* are put into an 8-bit adder for partial result in *P* to be added to *B*. The output of this adder *(P+B)* feeds one side of a multiplexer. The other side of the multiplexer is driven by the *P* output, (P+0). The *sel_sum* control input determines if *P+B* or *P+0* is to go on the multiplexer output.

The AND gate shown in Figure 11.5 selects carry-out from the adder or **0** depending of the value of *sel_sum* control input. This value is concatenated to the left of the multiplexer output to form a 9-bit vector. This vector has *P+B* or *P+0* with a carry to its left. The right-most bit of this 9-bit vector is split and

goes into the serial input of the shift-register that contains *A*, and the other eight bits go into register *P*. Note that concatenation of the AND gate output to the left of multiplexer output and splitting the right bit from this 9-bit vector, effectively produces a shifted result that is clocked into *P*.

11.3.3 Description of Parts

Register *P* and *B* in Figure 11.5 are 8-bit registers with active high load-enable inputs. Module *Reg8*, shown in Figure 11.6 is used for these registers.

The adder used for adding *P* and *B* is a simple 8-bit adder with a carry-in and a carry-out and is shown in Figure 11.7. This description uses an **assign** statement that assigns *a+b+ci* to the concatenation of *co* and *s*. With this assignment, the carry-out from the operation on the right-hand-side is captured in *co*.

Another component of the multiplier design is the 8-bit shift-register of Figure 11.8. The shift-register keeps its contents in its *im_data* intermediate variable. Depending on *{s1,s0}*, *im_data* is either untouched, shifted to the right, loaded with *data* or reset to 0.

```
module Reg8(d_in, clk, en, d_out);

  input [7:0] d_in;
  input clk, en;
  output [7:0] d_out;
  reg [7:0] d_out;

  always @( posedge clk )
    if (en) d_out = d_in;

endmodule
```

Figure 11.6 8-bit Register Used for *P* and *B*

```
module Add8 ( a, b, ci, s, co );

  input [7:0] a, b;
  input ci;
  output [7:0] s;
  output co;

  assign { co, s } = a + b + ci;

endmodule
```

Figure 11.7 8-bit Adder with Carry

```
module Shift8 ( clk, sin, s1, s0, oe, qa, data );

  input clk, sin, s1 , s0, oe;
  output qa;
  inout [7:0] data;

  reg [7:0] im_data;

  always @( posedge clk )
    case ( { s1 , s0 } )
      2'b00 : im_data = im_data;
      2'b01 : im_data = { sin, im_data[7:1] };
      2'b10 : im_data = data;
      2'b11 : im_data = 8'b00;
    endcase

  assign data = ( oe & ~s1 ) ? im_data : 8'hzz;
  assign qa = im_data[0];

endmodule
```

Figure 11.8 Shift-Register with Tri-state Output

```
module Mux8 ( a, b, sel, zero, y );

  input [7:0] a, b;
  input sel, zero;
  output [7:0] y;

  assign y = zero ? 8'h0 : ( ~sel ? a : b );

endmodule
```

Figure 11.9 Multiplexer

```
module Tri8 ( d_in, en, d_out );

  input [7:0] d_in;
  input en;
  output [7:0] d_out;

  assign d_out = en ? d_in : 8'hzz;

endmodule
```

Figure 11.10 Tri-state for Driving *databus*

When the output enable *(oe)* of the shift-register is active, *im_data* is placed on the *data* bi-directional port of the shift-register. Otherwise, *data* is float. Placement of *im_data* on *data* is also conditioned by *~s1,* so that *data* is driven only when not used as input.

Another component of the datapath of Figure 11.5 is the multiplexer of Figure 11.9. This multiplexer selects its *a* or *b* input depending on the value of *sel.* In addition, the multiplexer has a *zero* input that when **1**, it forces its output to *8'h0*. Since the multiplexer output connects to *P,* its zeroing feature is used for initial resetting of the *P* register.

As shown in Figure 11.5, an octal tri-state buffer connects the output of *P* to the bi-directional *databus*. The Verilog Code of this buffer is shown in Figure 11.10. The *en* input of this structure becomes active, when the most significant byte of the result that is in *P* is to go on the multiplier output *(databus).*

11.3.4 Datapath Description

The Verilog Code of the *datapath* of the multiplier is shown in Figure 11.11. In this description components described above are instantiated and wired together according to the block diagram of Figure 11.5.

```
module datapath ( clk, clr_P, load_P, load_B, msb_out,
           lsb_out, sel_sum, load_A, shift_A, data, A0 );

  input clk, clr_P, load_P, load_B, msb_out, lsb_out, sel_sum, load_A, shift_A;
  inout [7:0] data;
  output A0;

  wire [7:0] B, P, sum, ShiftAdd;
  wire co;

  Reg8 latch_B ( data, clk, load_B, B );

  Add8 add_PB ( P, B, 1'b0, sum, co );

  Mux8 P_or_sum ( P, sum, sel_sum, clr_P, ShiftAdd );

  Reg8 latch_P ( {co&sel_sum,ShiftAdd[7:1]}, clk, load_P, P );

  Shift8 latch_A_shift ( clk, ShiftAdd[0], load_A, shift_A, lsb_out, A0, data );

  Tri8 buffer ( P, msb_out, data );

endmodule
```

Figure 11.11 Datapath Verilog Code

11.3.5 Multiplier Controller

The multiplier controller is a finite state machine that has two starting states, eight multiplication states, and two ending states. States and their binary assignments are shown in Figure 11.12. In the *`idle* state the multiplier waits for *`start* while loading *A*. In *`init,* it loads the second operand *B*. In *`m1* to *`m8,* the multiplier performs add-and-shift of *P+P,* or *P+0,* depending on *A0.* In the last two states(*`rslt1* and *1rslt2),* the two halves of the result are put on *databus.*

```
`define idle   4'b0000
`define init   4'b0001
`define m1     4'b0010
`define m2     4'b0011
`define m3     4'b0100
`define m4     4'b0101
`define m5     4'b0110
`define m6     4'b0111
`define m7     4'b1000
`define m8     4'b1001
`define rslt1  4'b1010
`define rslt2  4'b1011
```

Figure 11.12 Multiplier Control States

The Verilog Code of controller is shown in Figure 11.13. This Code declares *datapath* ports, and uses a single **always** block to issue control signals and make state transitions. At the beginning of this **always** block all control signal outputs are set to their inactive values. This eliminates unwanted latches that may be generated by the synthesis tool for these outputs.

The 4-bit *current* variable represents the currently active state of the machine. When *current* is *`idle* and *start* is **0**, the *done* output remains high. In this state if *start* becomes **1**, control signals *load_A*, *clr_P* and *load_P* become active to load *A* with *databus* and clear the *P* register. Clearing *P* requires *clr_P* to put **0**'s on the multiplexer output by disabling it and loading the **0**'s into *P* by asserting *load_P*.

In *`m1 to `m8* states, *A* is shifted, *P* is loaded, and if *A0* is **1**, *sel_sum* is asserted. As discussed in relation to *datapath, sel_sum* controls shifted *P+B* or shifted *P+0*) to go into *P.*

In the result states *lsb_out* and *msb_out* are asserted in two consecutive clocks in order to put *A* and *P* on the *databus,* respectively.

```
module controller ( clk, start, A0, clr_P, load_P, load_B,
             msb_out, lsb_out, sel_sum, load_A, Shift_A, done );

  input clk, start, A0;
  output clr_P, load_P, load_B, msb_out, lsb_out, sel_sum, done;
  output load_A, Shift_A;

  reg clr_P, load_P, load_B, msb_out, lsb_out, sel_sum, done;
  reg load_A, Shift_A;

  reg [3:0] current;

  always @ ( negedge clk ) begin
    clr_P = 0; load_P = 0; load_B = 0; msb_out = 0; lsb_out = 0; sel_sum = 0;
    load_A = 0; Shift_A = 0; done = 0;

    case ( current )
      `idle :
        if (~start) begin
          current = `idle;
          done = 1;
        end else begin
          current= `init;
          load_A = 1;
          clr_P = 1;load_P = 1;
        end
      `init:
        begin
          current=`m1;
          load_B = 1;
        end
      `m1, `m2, `m3, `m4, `m5, `m6, `m6, `m7, `m8 :
        begin
          current = current + 1 ; Shift_A = 1 ; load_P = 1 ;
          if(A0)sel_sum= 1;
        end
      `rslt1 :
        begin
          current=`rslt2; lsb_out = 1;
        end
      `rslt2 :
        begin
          current=`idle; msb_out = 1;
        end
      default : current=`idle;
    endcase

  end

endmodule
```

Figure 11.13 Verilog Code of Controller

```
module Multiplier ( clk, start, databus, lsb_out, msb_out, done );

  input clk, start;
  inout [7:0] databus;
  output done, lsb_out, msb_out;
  wire clr_P, load_P, load_B, msb_out, lsb_out, sel_sum, load_A, Shift_A;

  datapath dpu( clk, clr_P, load_P, load_B,
           msb_out, lsb_out, sel_sum, load_A, Shift_A, databus, A0 );
  controller cu( clk, start, A0, clr_P, load_P, load_B,
           msb_out, lsb_out, sel_sum, load_A, Shift_A, done );

endmodule
```

Figure 11.14 Top-Level Multiplier Code

11.3.6 Top-Level Code of the Multiplier

Figure 11.14 shows the top-level *Multiplier* **module**. The *datapath* and *controller* modules are instantiated here. The input and output ports of this unit are according to the diagram of Figure 11.1. This description is synthesizable, and can be ported into Quartus II for synthesis and device programming.

11.4 Multiplier Testing

This section shows an auto-check verifying testbench for our sequential multiplier. Several forms of data applications and result monitoring are demonstrated by this example. The outline of the *test_multiplier* **module** is shown in Figure 11.15.

In the declarative part of this testbench inputs of the multiplier are declared as **reg** and its outputs as **wire**. Since *databus* of the multiplier is a bidirectional bus, it is declared as **wire** for reading it, and a corresponding *im_data* **reg** is declared for writing into it. An **assign** statement drives *databus* with *im_data*. When writing into this bus from the testbench, the writing must be done into *im_data,* and after the completion of writing the bus must be released by writing 8'hzz into it.

Other variables declared in the testbench of Figure 11.15 are *expected_result* and *multiplier_result.* The latter is for the result read from the multiplier, and the former is what is calculated in the testbench. It is expected that these values are the same.

The testbench shown in Figure 11.15 applies three rounds of test to the *Multiplier* **module**. In each round, data is applied to the module under test and results are read and compared with the expected results. The following are tasks performed by this testbench:

- Read data files *data1.dat* and *data2.dat* and apply data to *databus*
- Apply *start* to start multiplication

- Calculate the expected result
- Wait for multiplication to complete, and collect the calculated result
- Compare expected and calculated results and issue error if they do not match

These tasks are independently timed, and at the same time, an **always** block generates a periodic signal on *clk* that clocks the multiplier.

```
`timescale 1ns/100ps

module test_multiplier;
  reg clk, start, error;
  wire [7:0] databus;
  wire lsb_out, msb_out, done;
  reg [7:0] mem 1[0:2], mem2[0:2];
  reg [7:0] im_data, opnd1, opnd2;
  reg [15:0] expected_result, multiplier_result;
  integer indx;

  Multiplier uut ( clk, start, databus, lsb_out, msb_out, done );

  initial begin: Apply_data   . . .   end                              // Figure 11.16
  initial begin: Apply_Start   . . .   end                             // Figure 11.17
  initial begin: Expected_Result   . . .   end                         // Figure 11.18
  always @(posedge clk) begin: Actual_Result  . . .  end               // Figure 11.19
  always @(posedge clk)  begin: Compare_Results   . . .   end          // Figure 11.20
  always #50 clk = ~clk;
  assign databus=im_data;
endmodule
```

Figure 11.15 Multiplier Testbench Outline

11.4.1 Reading Data Files

Figure 11.16 shows the *Apply_data* **initial** block that is responsible for reading data and applying them to *im_data*, which in turn goes on *databus*. Data from *data1.dat* and *data2.dat* external lines are read into *men1* and *men2*. In each round of test data from *mem1* and *mem2* are put on *im_data*. Data from *mem2* is distanced from that of *mem1* by 100 ns. This way, the latter is interpreted as data for the *A* operand and the former for the *B* multiplication operand. After placing this data, 8'hzz is put on *im_data*. This releases the *databus* so that it can be driven by the multiplier when its result is ready.

11.4.2 Applying Start

Figure 11.17 shows an **initial** block in which variable initializations take place, and *start* signal is issued. Using a **repeat** statement, three 100 ns pulses distanced by 1350 ns are placed on *start*.

11.4.3 Calculating Expected Result

Figure 11.18 shows an **initial** block that reads data that is put on *databus* by the *Apply_data* block (Figure 11.16), and calculates the expected multiplication result. After *start,* when *databus* is updated, the first operand is read into *opnd1*. The next time *databus* changes, *opnd2* is read. The expected result is calculated using these operands.

```
initial begin: Apply_data
  indx=0;
  $readmemh( "data1.dat", mem1 );
  $readmemh( "data2.dat", mem2 );
  repeat(3) begin
    #300 im_data=mem1 [indx];
    #100 im_data=mem2 [indx];
    #100 im_data=8'hzz;
    indx=indx+1;
    #1000;
  end
  #200 $stop;
end
```

Figure 11.16 Reading Data Files

```
initial begin: Apply_Start
  clk=1'b0;start=1'b0; im_data=8'hzz;
  #200 ;
  repeat(3) begin
    #50   start=1'b1;
    #100  start=1'b0;
    #1350;
  end
end
```

Figure 11.17 Initializations and Start

```
initial begin: Expected_Result
  error=1'b0;
  repeat(3) begin
    wait (start==1'b1 );
     @ (databus);
    opnd1=databus;
     @( databus);
    opnd2=databus;
    expected_result = opnd1 * opnd2;
  end
end
```

Figure 11.18 Calculating Expected Result

11.4.4 Reading Multiplier Output

When the multiplier completes its task, it issues *msb_out* and *lsb_out* to signal that it has readied the two bytes of the result. The **always** block of Figure 11.19 is triggered by the rising edge of the circuit clock. After a clock edge, if *msb*_out or *lsb*_out is **1**, it reads the *databus* and puts in its corresponding position in *multiplier_result.*

```
always @(posedge clk) begin: Actual_Result
  if (msb_out) multiplier_result[15:8] = databus;
  if (lsb_out) multiplier_result[7:0] = databus;
end
```

Figure 11.19 Reading Multiplier Results

```
always @(posedge clk) begin: Compare_Results
  if (done)
    if (multiplier_result != expected_result) error = 1 ;
    else error = 0;
end
```

Figure 11.20 Comparing Results

11.4.5 Comparing Results

Figure 11.20 shows the **always** block that is responsible for comparing actual and expected multiplication results. After the active edge of the circuit clock if *done* is **1**, then comparing *multiplier_result* and *expected_result* takes place. If values of these variables do not match *error* is issued.

The self-running testbench presented here verifies RT-level operation of our multiplier. Prototyping this design using the UP2 board is presented in the next section.

11.5 Multiplier Prototyping

We use the FLEX 10K device of UP2 for prototyping our multiplier. This section describes porting the Verilog Code of the multiplier into Quartus II, generating switch and display interfaces for our design and programming the EPF10K70 of the UP2 development board. The Quartus II project used for this part is *SeqMultiplier* in a design directory by the same name. *BookLibrary* is included in the list of libraries available to the project.

11.5.1 Porting Multiplier into Quartus II

The *Multiplier* **module** of Figure 11.14 is the top-level module of our multiplier. To be able to use this design in Quartus II, this and all its related Verilog Files must be copied to the directory of the *SeqMultiplier* project.

In order to use the *Multiplier* **module** in a Quartus II schematic, a symbol has to be created for it. Figure 11.21 shows this symbol created by Quartus II, after some manual editings.

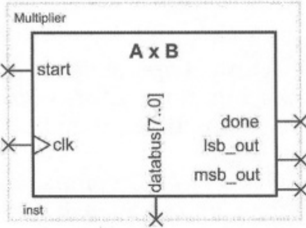

Figure 11.21 Multiplier Symbol

11.5.2 Multiplier Interfaces

Figure 11.22 shows the *Seq Multiplier* schematic that includes the *Multiplier* and its pushbutton and display interfaces. In order to step through the multiplication process, its clock is driven by a pushbutton. The other pushbutton available to FLEX is used for the *start* input. FLEX switch set is used for the *A* and *B* operands. We manually set these switches to values that are to be multiplied.

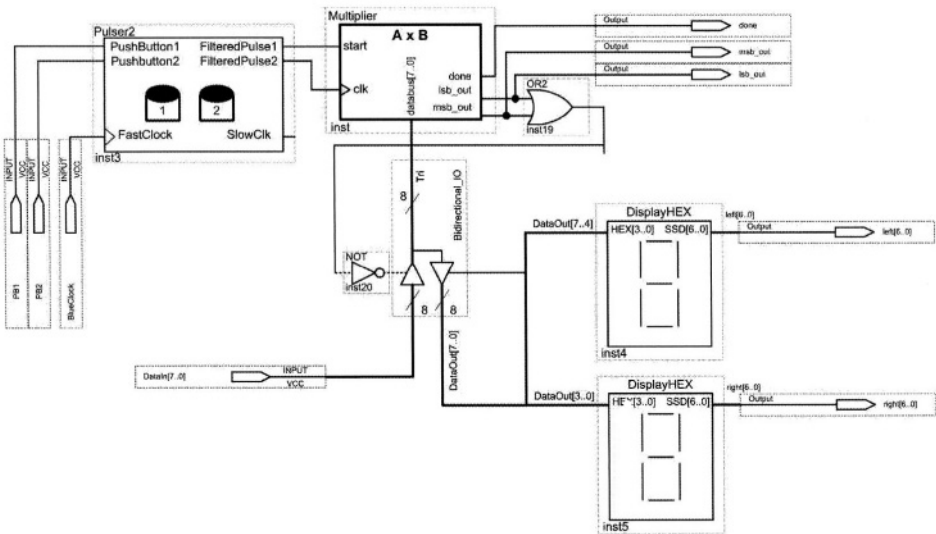

Figure 11.22 ***SeqMultiplier*** **Prototype**

For the display of the output of the multiplier two instances of *DisplayHEX* from *BookLibrary* are used. These outputs display both halves of the 16-bit output of the multiplier.

11.5.3 Bidirectional Databus

The multiplier *databus* is a bidirectional bus used for *A* and *B* operands as well as the two halves of the result. We have used the *lpm_bustri* megafunction of Quartus II, that is available under *gates* category of megafunctions, to split the in-side and out-side of the *databus*.

The FLEX switches connect to the input side of *Bidirectional-IO* component, and the displays connect to its output side. When either *lsb_out* or *msb_out* is issued by the multiplier, the *databus* connects to the displays through the *Bidirectional-IO*. At all other times that the multiplier is not driving its output, the switches drive the *databus*.

11.5.4 Operating the Prototype

Compiling *SeqMultiplier* of Figure 11.22 synthesizes the *Multiplier* **module** and together with the rest of components of this design, generates the *SeqMultiplier.sof* file for programming the FLEX 10K device.

Pin assignments are done according to permanently assigned pins of FLEX. Bits of *DataIn* port of diagram of Figure 11.22 are connected to the switches according to Figure 6.34. The outputs of the *DisplayHEX* components are assigned to the seven segment displays according to Figure 6.35, and inputs PB1 and PB2 are assigned to FLEX pushbuttons as shown in Figure 6.33.

To test the multiplier, the switches are set to a test value for *A* and while *start* is **1**, a clock pulse is given. Then, while *start* is **0**, the switches are set to a value for *B* and another clock pulse is given. Following the leading of *A* and *B,* eight clock pulses are given (releasing and pressing PB2 eight times) to complete the multiplication process. With the next clock, the right-most byte of the result becomes available on the SSDs, and with the next clock the left-most byte of the multiplication result becomes available on the SSDs. Both values of the output are displayed in hexadecimal Code.

Figure 11.23 shows part of the FLEX 10K timing closure floorplan after being programmed with our multiplier. The complete *SeqMultiplier* project uses 230 Logic Elements of the total 3744 available on FLEX 10K. Of the available 36,804 memory bits, none are used. The timing viewer allows cells to be selected and timing between them be viewed. When a cell is selected, its fan-ins and fan-outs are listed and corresponding delay values are shown on the arrows going between the logic elements; Figure 11.23 shows an example.

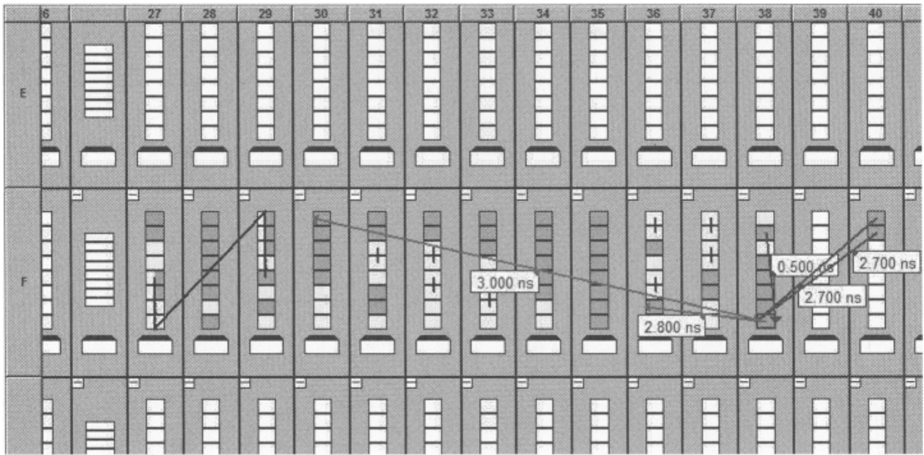

Figure 11.23 Chip Floorplan (Partial View)

11.6 Summary

This chapter showed a complete design of a system with a well-defined datapath and a good-size controller. The design demonstrates top-down design and data/control partitioning. We showed how this design could be implemented by coding lower level RTL parts and then wiring them into a complete system. Concepts of controllers, control signals controlling data activities, bussing, and various forms of unidirectional and bi-directional busses were demonstrated in this design. We demonstrated how the UP2 board could be utilized to test the physical implementation of an HDL based design.

This page intentionally left blank