8-by-8 Bit Shift/Add Multiplier

Giovanni D'Aliesio
ID: 4860519

Digital Design & Synthesis
COEN 6501
Department of
Electrical & Computer Engineering
Concordia University
December 2003

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

The objective of this project is to go through a design cycle from initial conception to simulation. In this case, it has been taken several steps further and synthesis as well as place & route was also achieved. The goal is to design and simulate an 8-by-8 bit shift/add multiplier. The result is a completely synthesized 8-by-8 bit and 32-by-32 bit shift/add multiplier with various design options for speed and area.

1.1 Design Flow

The VHDL entry, simulation, synthesis and place & route was performed using a variety of high performance, UNIX based CAD tools. The complete design flow is shown in Figure 1-1.

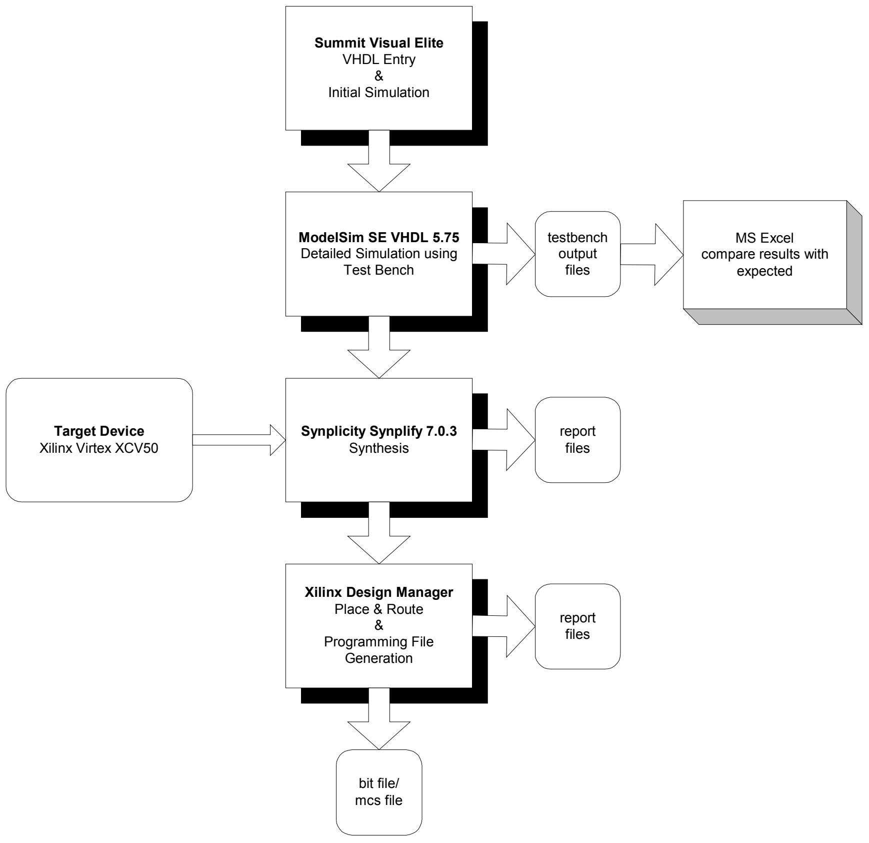


Figure 1-1: FPGA Design Flow

Initial VHDL entry was done in Summit Visual Elite. This tool provided the functionality to compile the code and to perform initial simulation. The simulation performed at this level was mainly to verify block level functionality and timing correctness. Once the blocks were coded and verified, a more advanced simulation tool was used, namely ModelSim. In this case, a complete test bench was developed which performed many multiplications and saved the result in a file. This file was then compared to the expected results in order to confirm proper functionality. Upon successful testing, the design was then synthesized using Synplicity's Synplify and the technology library for the Xilinx Virtex XCV50. Finally, Xilinx Design Manager was used to place and route the design, and generate the appropriate programming files.

2 GENERAL REQUIREMENTS

The requirement is to design an 8-by-8 bit multiplier based on the shift and add method. The overall architecture is shown in Figure 2-1. The multiplier shall accept as inputs an 8-bit multiplier and 8-bit multiplicand as well as a Start signal. The multiplier shall then calculate the result using the shift and add method and provide the 16-bit result along with a Stop signal. The design shall be coded in VHDL and simulated for proper functionality and timing.

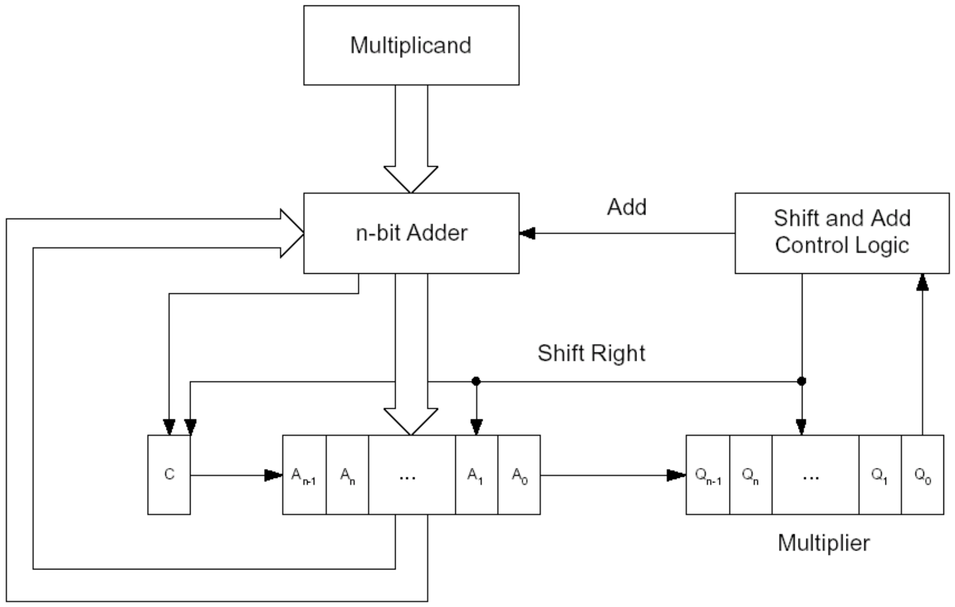


Figure 2-1: Add/Shift Multiplier Block Diagram

3 DESIGN SPECIFICATIONS

The design was implemented using a mixture of both structural design and rtl level design. In each case the choice of style is described. The block diagram shown in Figure 3-1 details the breakdown of VHDL modules. In the following sections each of the modules are described in greater detail and associated diagrams, simulation outputs and timing are provided.

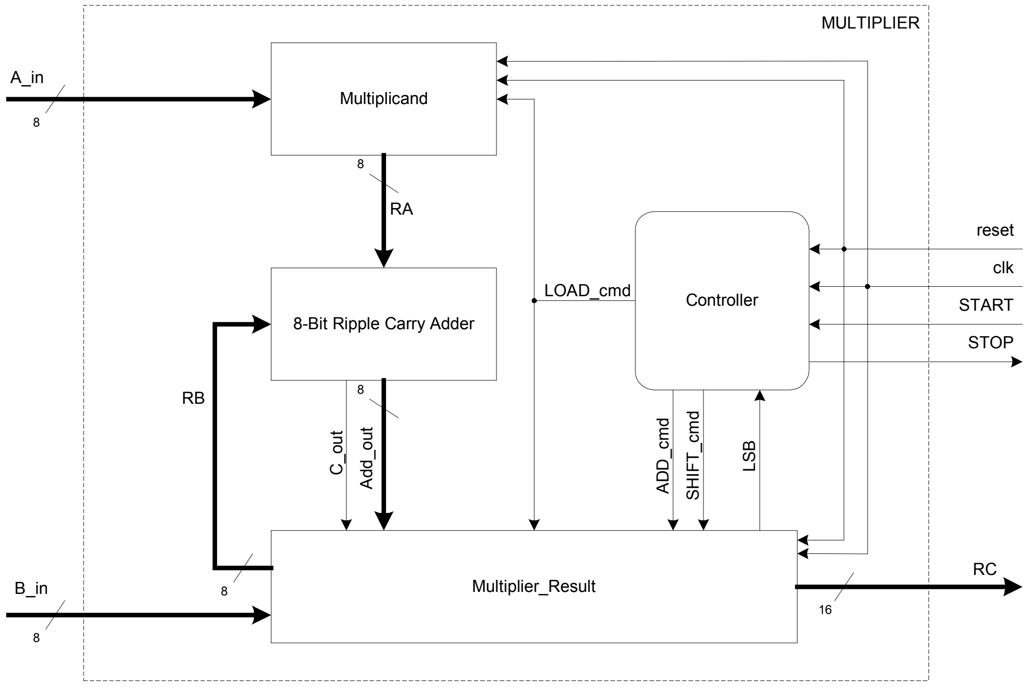


Figure 3-1: Multiplier Design Block Diagram

3.1 Controller Design

The Controller is the control unit of the multiplier. It receives a START signal and consequently commands all other modules until the result is obtained and it outputs a STOP signal.

3.1.1 Design

The design was implemented as a finite state machine with states and transition logic as shown in Figure 3-2. The Start signal transitions the state machine out of the idle state and into the initialize state whereby it commands the multiplicand and multiplier to be loaded into registers. Once loaded, the state machine goes through a series of test and shift, or test, add and shift operations depending on the status of the LSB bit. Upon reaching the maximum count for the multiplication cycle, the state machine goes back to the idle state and outputs a Stop signal.

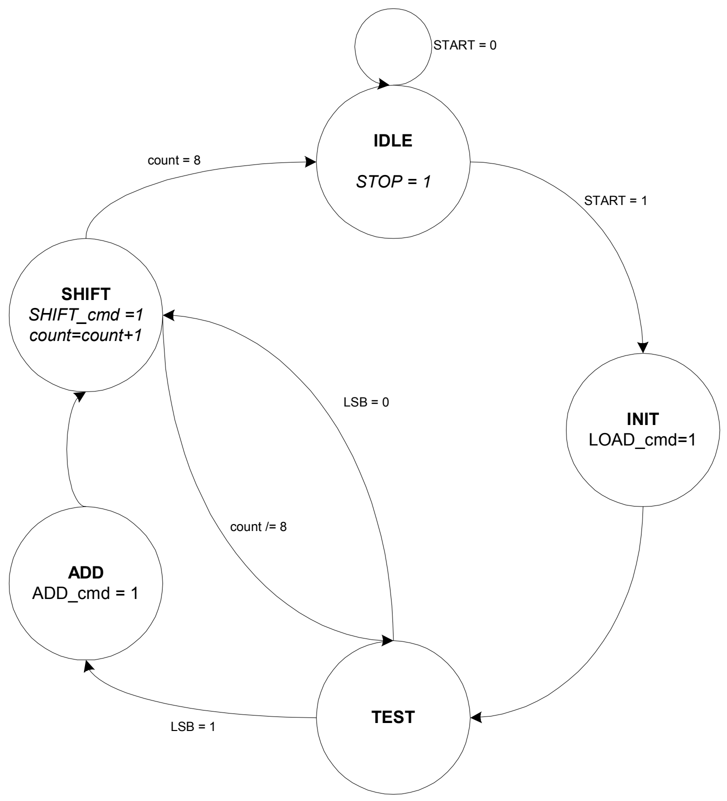


Figure 3-2: Controller FSM Diagram

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.1.2 Simulation & Timing

The controller is synchronous to the clock and transitions through the various states occur on the rising clock edge. As can be seen from the timing diagram in Figure 3-3, the Start signal transitions the state machine out of the idle state only when sampled by the rising clock edge. Upon entering the initialize state, the LOAD_cmd is generated. During each test state, the LSB is sampled. If the LSB was high, the add state is entered and the controller generates the ADD_cmd. If the LSB was low, or once the add state is exited, the shift state is entered and the controller generates the SHIFT_cmd. Upon reaching the maximum count for the multiplication cycle, the state machine goes back to the idle state and outputs a stop signal.

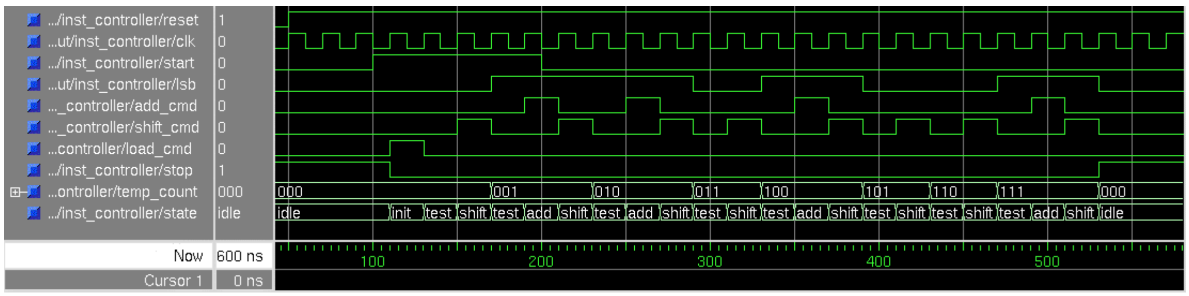


Figure 3-3: Controller Simulation Timing Diagram

3.2 Multiplicand Block Design

The Multiplicand block is composed of 8 D Flip-Flop blocks, which store the "A" byte for processing during the complete multiplication cycle. The register is loaded with the LOAD_cmd signal from the Controller.

3.2.1 Design

The basic design for the Multiplicand block is that of an 8-bit register. The top-level multiplicand module generates an 8-bit register from individual 1-bit D Flip-Flops. The individual D flip-flops have been designed both structurally and behaviorally and the synthesized results are compared in section 4.6. The structural design follows the diagram in Figure 3-4. The detailed diagram of the Multiplicand module is shown in Figure 3-5. The byte is loaded into the register only when the LOAD_cmd is received from the Controller and the register is cleared when a global reset is applied.

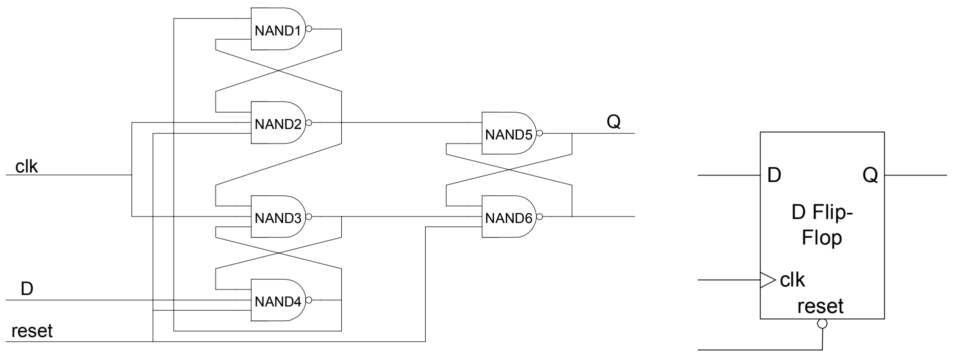


Figure 3-4: Structural D Flip-Flop

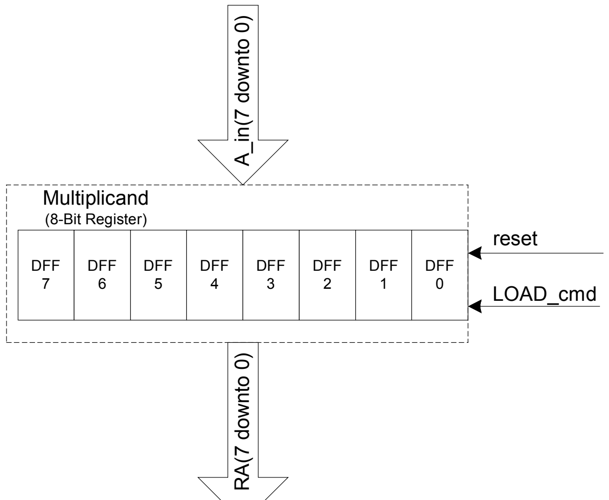


Figure 3-5: Multiplicand Diagram

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.2.2 Simulation & Timing

The timing and simulation results are identical regardless of the flip-flop design (structural or behavioral). The timing diagram in Figure 3-6 displays the functionality of the 8-bit register. The input, A_in, is only loaded into the register on the rising edge of the LOAD_cmd signal and remains in the register until the next time the LOAD_cmd signal is asserted.

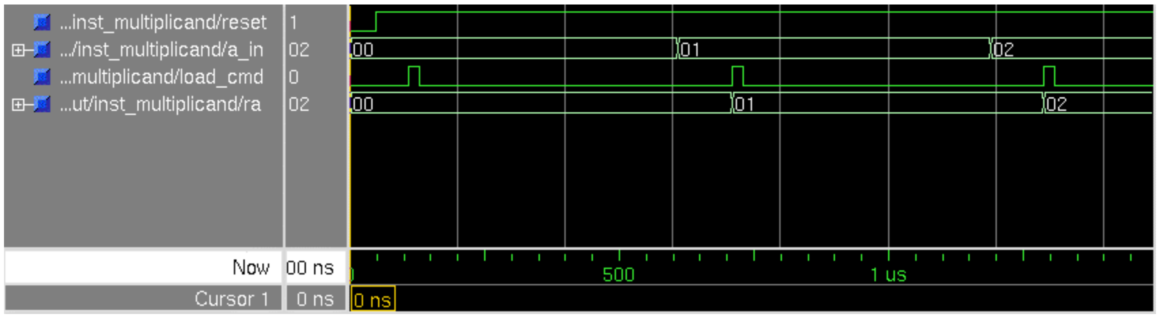

...inst_multiplicand/reset
.../inst_multiplicand/a_in
...multiplicand/load_cmd
...ut/inst_multiplicand/ra
1
02
0
02
00
01
02
00
01
02
Now
00 ns
500
1 us
Cursor 1
0 ns
0 ns

Figure 3-6: Multiplicand Simulation Timing Diagram

3.3 Multiplier/Result Block Design

The Multiplier/Result block stores the multiplier ("B" byte) as well as the accumulated output of the adder. It allows the register to be logically shifted right and provides one of the Adder's inputs. The Multiplier/Result block consists of a 17-bit shift register and a multiplexer in order to provide this functionality.

3.3.1 Design

The design of the Multiplier/Result module was performed at the RTL level. A block diagram of this module is shown in Figure 3-7 and the register assignment details are shown in Figure 3-8. The input to the module is the multiplier, B_in, which is loaded into bits 0 to 7 of the register when the LOAD_cmd is asserted. The adder block outputs (Cout and adder_out) are also inputs to the module. The outputs of the module are, LSB, bit 0 of the register, RB, bits 8 to 15 of the register, and RC, bits 0 to 15 of the register. LSB is fed back to the controller to determine the next state, while RB is fed into the adder in order to be summed with the multiplicand. RC is the final multiplication result and is considered valid only when the controller asserts the STOP signal. If the Multiplier_Result receives a SHIFT_cmd without a prior ADD_cmd, the register will be shifted logically to the right. If the ADD_cmd precedes the SHIFT_cmd, bits 1 to 7 of the register will be placed into positions 0 to 6 while the 9 inputs bits from the adder will be placed into positions 7 to 15 of the register. This is essentially equivalent to storing the adder results and then shifting the entire register.

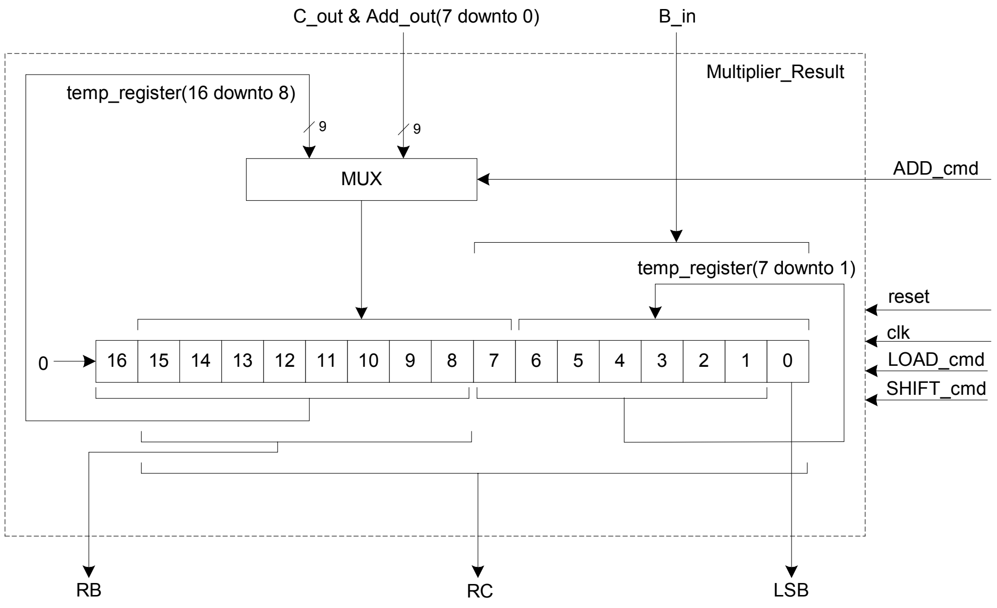


Figure 3-7: Multiplier_Result Block Diagram

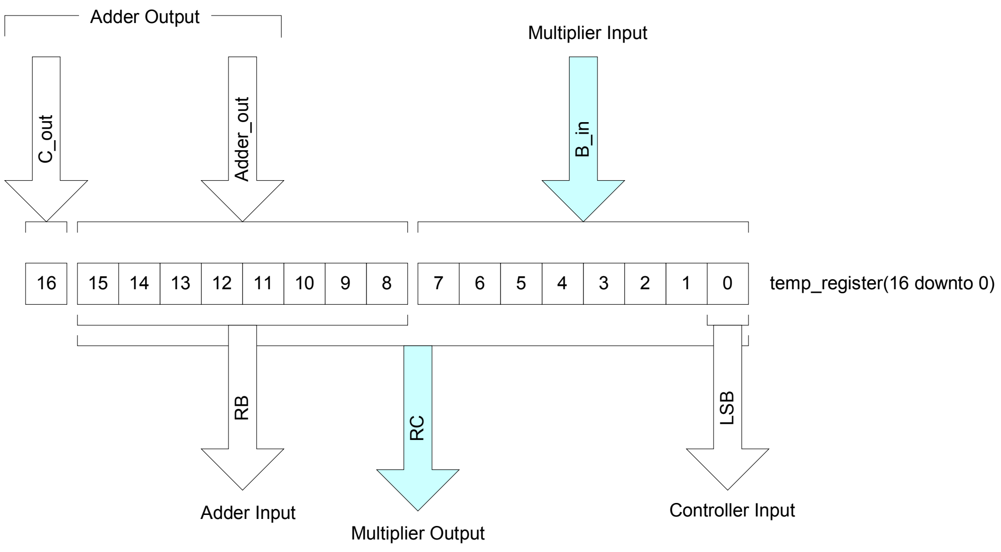


Figure 3-8: Multiplier_Result Register Diagram

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.3.2 Simulation & Timing

The timing diagram for the Multiplier_Result simulation is shown in Figure 3-9. The 17-bit register as shown in Figure 3-8, contains all the data that will be used as outputs of the module. From the timing diagram, it can be seen that once the LOAD_cmd is received, temp_register is loaded with the input byte "b_in", in this case 0x96. If the SHIFT_cmd is received without a prior ADD_cmd, the contents of temp_register will be logically shifted right. This can be seen on the next clock edge when 0x96 is shifted to become 0x4B. If the ADD_cmd is received prior to the SHIFT_cmd, the 8 to 16 bits of temp_register will be loaded with the "add_out" input byte and the register will be logically shifted right. This can be seen on the next clock edge when 0xA is loaded into temp_register which contains 0x4B. This changes temp_register to equal 0xA4B which gets shifted right to become 0x525 as shown in the timing diagram.

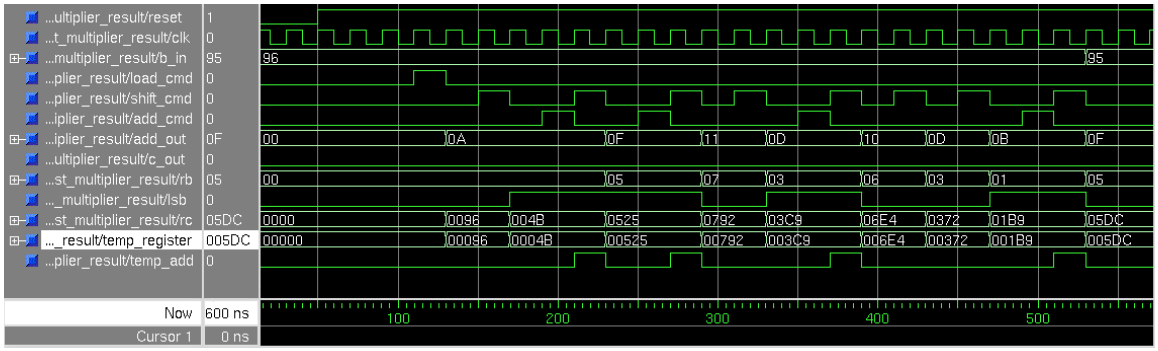


Figure 3-9: Multiplier_Result Simulation Timing Diagram

3.4 Adder Design

The adder design is the most influential part of the multiplier in terms of the area and speed achievable. This is due to the large iterations of additions performed in the multiplication cycle. In this project, two types of adders were designed and compared. In this section, only a description of the design will be presented, while the comparison details will be left to section 4.6.

3.4.1 Design

The main function of the adder block is to sum two bytes together. The main building block of the adders described in the following paragraph is the Full Adder. This block is designed structurally and accepts 2 inputs while generating 1 sum output and 1 carry output.

The 8-bit ripple carry adder is composed of 8 individual full adders connected in a chain as shown in Figure 3-10. In this case, the carry out of each full adder is the carry in of the following full adder. The limiting speed factor in this approach is the delay from the first full adder to the outputs of the final full adder. In order to try and improve the speed of the adder, consequently enhancing the overall speed, the carry-select adder shown in Figure 3-11 was also implemented and compared to the original ripple carry adder. The carry-select adder is composed of three 4-bit ripple carry adder blocks whereby the carry out of the first ripple carry adder selects which of the other two ripple carry adders to send to the output. In this case the limiting factor is the delay from the input of the first 4-bit ripple-carry-adder to its carry out which controls the multiplexer.

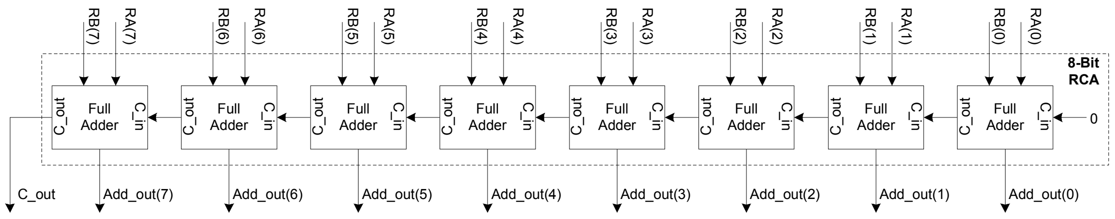

RB(7) RA(7) RB(6) RA(6) RB(5) RA(5) RB(4) RA(4) RB(3) RA(3) RB(2) RA(2) RB(1) RA(1) RB(0) RA(0)
8-Bit RCA
Full Adder
C_out
C_in
0
C_out
Add_out(7) Add_out(6) Add_out(5) Add_out(4) Add_out(3) Add_out(2) Add_out(1) Add_out(0)

Figure 3-10: Ripple Carry Adder Block Diagram

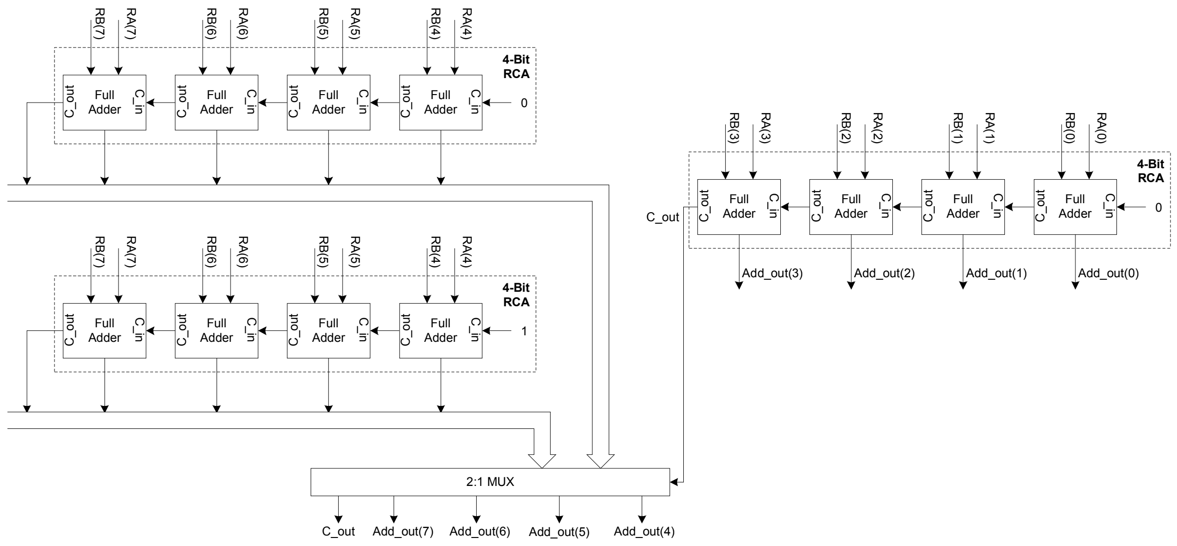

RB(7) RA(7) RB(6) RA(6) RB(5) RA(5) RB(4) RA(4)
4-Bit RCA
Full Adder
C_out
C_in
0
1
RB(3) RA(3) RB(2) RA(2) RB(1) RA(1) RB(0) RA(0)
C_out
Add_out(3) Add_out(2) Add_out(1) Add_out(0)
2:1 MUX
C_out Add_out(7) Add_out(6) Add_out(5) Add_out(4)

Figure 3-11: Carry Select Adder Block Diagram

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.4.2 Simulation & Timing

The simulation of both the ripple-carry-adder and the carry-select-adder are shown in Figure 3-12 and Figure 3-13 respectively. As can be seen from the simulation output of the ripple-carry-adder, the addition of 0xA and 0x5 gives the result 0xF, which is correct. In this particular addition, no carry-out signals were generated by the full adders and as such the internal vector representing the carry-out's, c_temp, remained 0. In the subsequent addition of 0x7 and 0xA, the result is observed to be 0x11, which is also correct. In this addition, carry-out's occur for full adder blocks 2, 3, and 4, hence the carry-out vector should be 0x1C (11100) which is observed in the timing diagram.

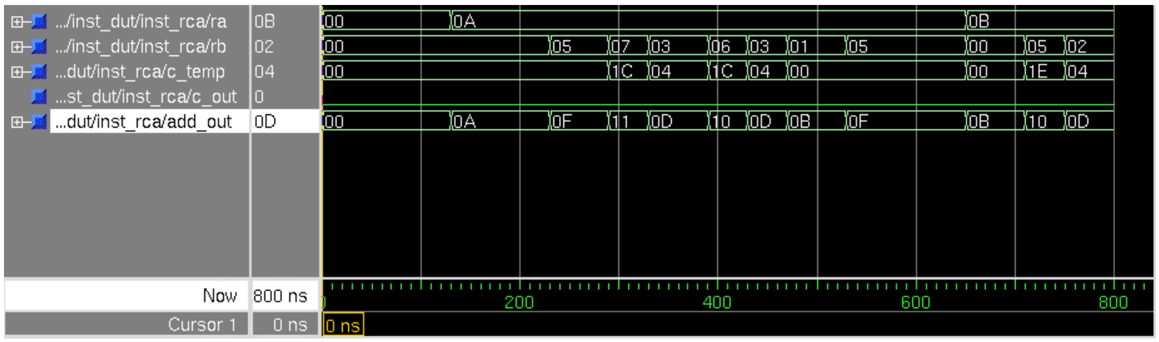

.../inst_dut/inst_rca/ra 0B
.../inst_dut/inst_rca/rb 02
...dut/inst_rca/c_temp 04
...st_dut/inst_rca/c_out 0
...dut/inst_rca/add_out 0D
Now 800 ns
Cursor 1 0 ns
200
400
600
800

Figure 3-12: Ripple Carry Adder Simulation Timing Diagram

The simulation output of the Carry Select Adder shown in Figure 3-13, displays both scenarios when selecting the higher order nibble. In the case of adding 0b11101110 and 0b00000000, the carry out of the first four bit addition is 0 and thus the result is 0b11101110 and not 0b11111110. In the case of adding 0b11101110 and 0b01110111, the carry out of the first four bit addition is 1 and thus the result is 0b01100101 and not 0b01010101.

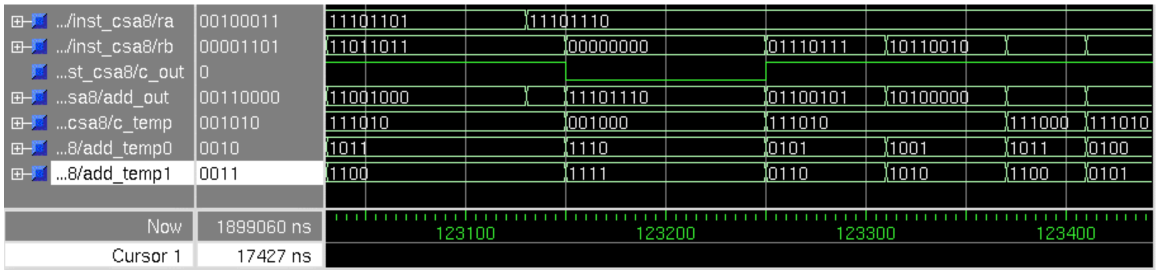


Figure 3-13: Carry Save Adder Simulation Timing Diagram

3.5 Multiplier Design

3.5.1 Design

The complete multiplier is simply a top-level module which instantiates all the lower level functional modules described in the previous sections.

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.5.2 Test Bench

The Test Bench is designed such that it can provide a series of stimuli to the Multiplier and log the results obtained. The log file, included in Appendix B: Simulation Result Files contains the multiplicand and multiplier sent to the Multiplier as well as the result received. In order to quickly validate the results, the log file is imported into an Excel spreadsheet (included on the CD) whereby an automatic comparison is performed between the theoretical results and the experimental results. A block diagram of the Test Bench architecture is shown in Figure 3-14 below.

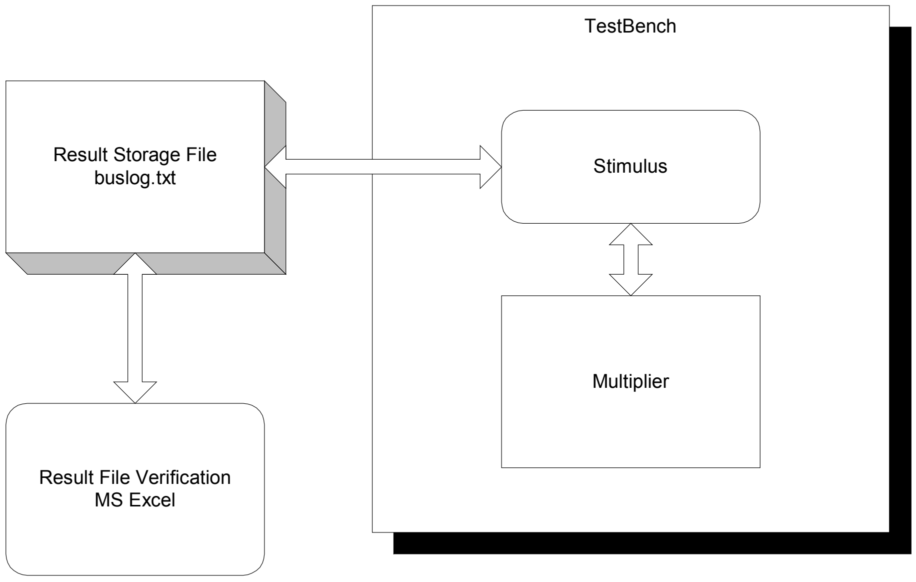


Figure 3-14: TestBench Simulation Block Diagram

The associated VHDL source code is included in Appendix A: VHDL Source Code.

3.5.3 Simulation & Timing

The timing diagram displayed in Figure 3-15 shows one complete multiplication cycle from the Start signal to the Stop signal. In the test case shown, the "A" byte is 0xA (10) and the "B" byte is 0x96 (150). The expected result is 0x5DC (1500) which is obtained once the Stop signal is asserted at the end of the multiplication cycle. While the timing diagram in Figure 3-15 only displays one multiplication cycle, Appendix B: Simulation Result Files contains the results of many multiplication cycles.

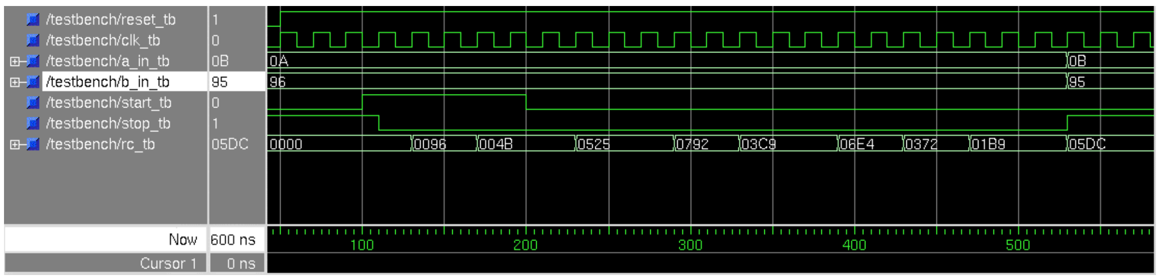


Figure 3-15: Complete Multiplier Simulation Timing Diagram

4 DESIGN ENHANCEMENTS

4.1 Timing

Timing is of great concern in any digital system. In this case there are two ways of looking at the timing issues. The first is the timing requirements of generating the fastest clock while respecting each individual timing requirement such that the functionality is not compromised. The second is the overall latency of the operation. This is the time, or amount of clock cycles that it takes from the Start signal to the Stop signal.

Frequency of operation is determined by analyzing the critical path of the system. That is, the path with the longest delay. From observation, one can guess that the longest path will likely be due to the long string of adders, each of which requires an input from the previous one. This observation is confirmed by analyzing the synthesis report found in Appendix C: Synthesis Report Files. In the case of the design using the ripple-carry-adder, the longest delay would be from the input of the first adder to the output of the last (eighth) adder. As confirmed by the report, the critical path is from temp_register[8] to temp_register[15]. This corresponds to the eight adder string as shown in Figure 3-8. Upon locating the critical path, a new adder design was used to reduce the delay. The timing report shown in Appendix C: Synthesis Report Files, for the case of the carry-select-adder, indicates the critical path to be from temp_register[8] to temp_register[12]. This corresponds to the input of the 4-bit ripple-carry-adder module to the carry-out which selects the appropriate result of the next stage 4-bit ripple-carry-adder. As expected, this delay is shorter than the previous adder design and thus a faster clock frequency is realized.

In the case of latency, depending on the input values, various cycle times are observed. Minimum latency occurs when the multiplier is 0x00 therefore the LSB is always 0 and the add state is never entered. The minimum latency is 17 clock cycles (1 init state + 8 bits X 2 states, test and shift). This scenario is shown in Figure 4-1 below.

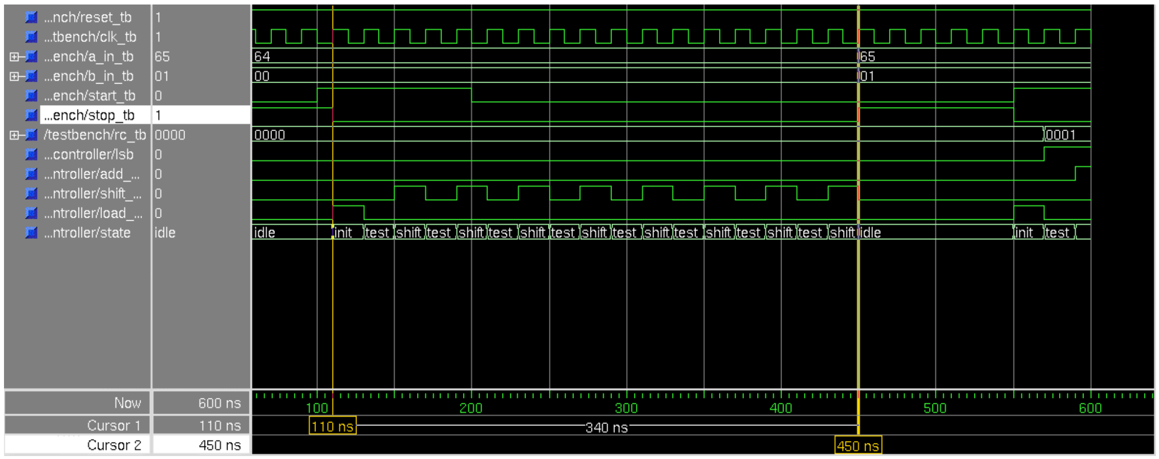


Figure 4-1: Minimum Latency Timing Diagram

Maximum latency occurs when the multiplier is 0xFF therefore the LSB is always one and the add state is always entered. The maximum latency is 25 clock cycles (1 init state + 8 bits X 3 states, test, add and shift). This scenario is shown in Figure 4-2 below.

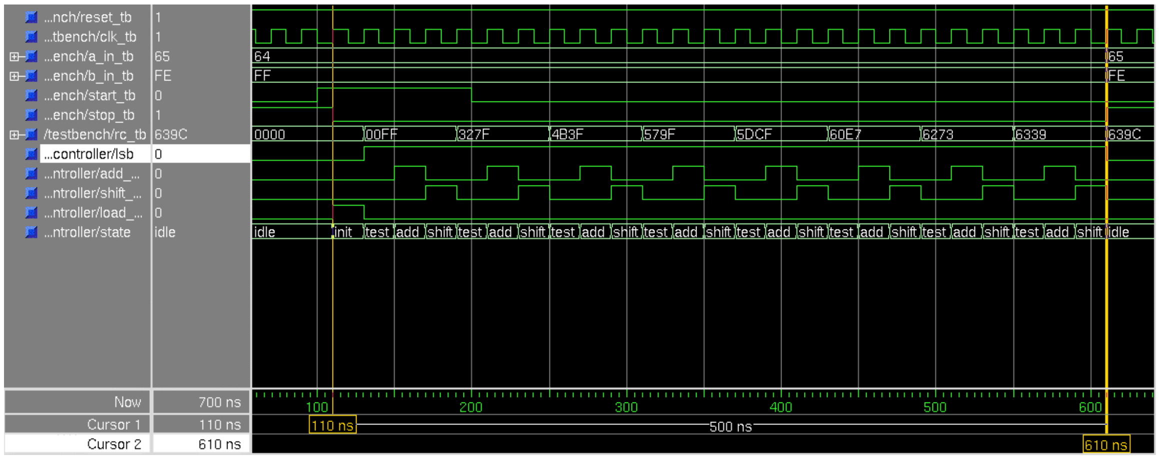


Figure 4-2: Maximum Latency Timing Diagram

4.2 Synthesis

For the purposes of this project, very little constraints were imposed on the design to be synthesized. Neither pin-out, stringent timing, synthesis effort level nor was area optimization specified. Synthesis was performed using Synplicity's Synplify CAD tool. The target device was selected along with the source VHDL file and constraints file. In turn the tool synthesized the design using the target library and generated the netlist as well as several reports. The complete output files are available on the CD-ROM, however an excerpt of the relevant timing and area details are included in Appendix C: Synthesis Report Files herein.

4.3 Place & Route

Place & Route was performed using Xilinx's Design Manager. The netlist file created during the synthesis phase was imported into the tool which routed the design in the appropriate target and generated a bit file which is used to physically program the device. Several reports were also generated which detail post layout timing and area. The complete output files are available on the CD-ROM. A view of the mapped out design using the tool's floorplanner is included as Appendix D: Place & Route Report Files herein.

4.4 Target

The target selected for the synthesis and place & route is the Xilinx Virtex XCV50. The XCV50 is the smallest device in the Virtex family and was chosen simply to outline the place and route process. The device details are outlined in Table 4-1 below. A detailed view of the Virtex slice is shown in Figure 4-3.

Table 4-1: Xilinx Virtex XCV50 Device Details

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+™ Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576

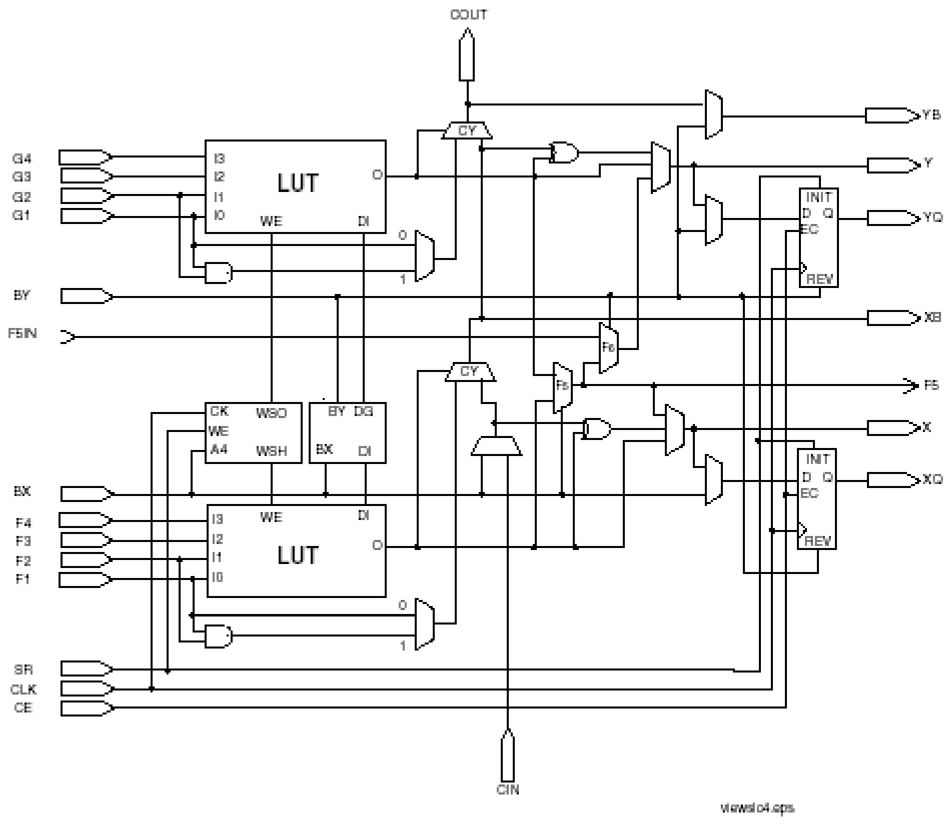


Figure 4-3: Detailed View of Virtex Slice

In order to gain an insight into the variations between devices and the effect on speed and area, one multiplier design was also synthesized using an Actel ACT3 A1415A-3 device. As this comparison can itself be a topic for a large project, the synthesis results are only included here as topic of interest. No detailed analysis was performed on the Actel device.

4.5 Design Improvements

Throughout the project, several improvements were realized in order to enhance the experience of implementing the multiplier. Several adders were implemented in order to investigate the effect on speed and area. In some cases both structural and behavioral VHDL was created for the same module in order to get an insight into its effects on timing and area. A 32-bit version of the multiplier was also implemented. Many further enhancements can be seen coming out of this project. Optimal designs for adders can achieve greater speed, while using several target-specific VHDL modules can reduce the total area.

4.6 Adder Selection

The following table summarizes the area and speed reported by the synthesis and place & route tools for each of the designs. As expected, the design using the carry-select-adder is faster than the ripple-carry-adder version however takes up more area. It is also expected that the 32-bit version of the multiplier is slower and takes nearly four times as much area as its 8-bit counterpart. The interesting observation is that the same 8-bit multiplier using the ripple-carry-adder takes up more area when the D flip-flops of the Multiplicand block are designed structurally as compared to behaviorally. This can be credited to the design of the slice as shown in Figure 4-3. In specifying the structure of the D flip-flop, the CAD tool will generate register as specified instead of using a structure native to the target. Even more interesting is the fact that once this design undergoes place & route, the speed also drops considerably. This is probably due to the fact that the delay in creating the flip-flop structurally is now greater than the adder delay. This is supported by the mapping diagram included as Appendix D: Place & Route Report Files. Included in the table is also the timing result obtained when synthesizing the same ripple-carry-adder version of the multiplier using an Actel ACT 3 device as the target. This demonstrates the effect the device has on actual timing regardless of design being implemented, since in this case it is the same design.

Table 4-2: Area & Speed for Various Adder Designs

	Synthesis			Place & Route		
Adder Type	Max Freq.	Critical Path Delay	Area (Lut)	Max Freq.	Area (SLICEs)	Area (Eq. Gate Count)
Ripple Carry Adder (DFF structural)	92.9 MHz	10.762 ns	77	47.0 MHz	40	662
Ripple Carry Adder (DFF behavioral)	92.9 MHz	10.762 ns	45	81.2 MHz	24	534
Ripple Carry Adder (DFF behavioral) Synthesized Actel	42.9MHz	23.29 ns	*	**	**	**
Carry Select Adder (DFF behavioral)	116.2 MHz	8.602 ns	50	105.2 MHz	27	564
32-Bit RCA (DFF behavioral)	38.1 MHz	26.242 ns	142	22.6 MHz	73	1705

*Area for Actel is in terms of cells: 44 combinational cells & 33 sequential cells
** Place & Route was not performed for Actel Target

4.7 32-Bit Expansion

The standard 8-by-8 bit multiplier design using the ripple-carry-adder and behavioral D flip-flops was expanded to accept 32-bit multipliers and multiplicands. The design can be further optimized, particularly with regards to the 32-bit ripple-carry-adder since the delay introduced in this long string of adders degrades the speed considerably. As shown in Appendix C: Synthesis Report Files, the critical path for the 32-bit design is from temp_reg[32] to temp_reg[63] which corresponds to the inputs of the first full adder in the ripple-carry-adder chain to the output of the last full adder in the chain.

The associated VHDL source code for both the 32-by-32 bit multiplier and its associated test bench is included in Appendix A: VHDL Source Code.

The timing diagram displayed in Figure 4-4 shows one complete multiplication cycle for the 32-by-32 bit version of the multiplier. In the test case shown, the "A" byte is 0xF4240 (1000000) and the "B" byte is 0x1E8480 (2000000). The expected result is 0x1D1A94A2000 (2000000000000) which is obtained once the Stop signal is asserted at the end of the multiplication cycle. While the timing diagram in Figure 4-4 only displays one multiplication cycle, Appendix B: Simulation Result Files contains the results of many multiplication cycles.

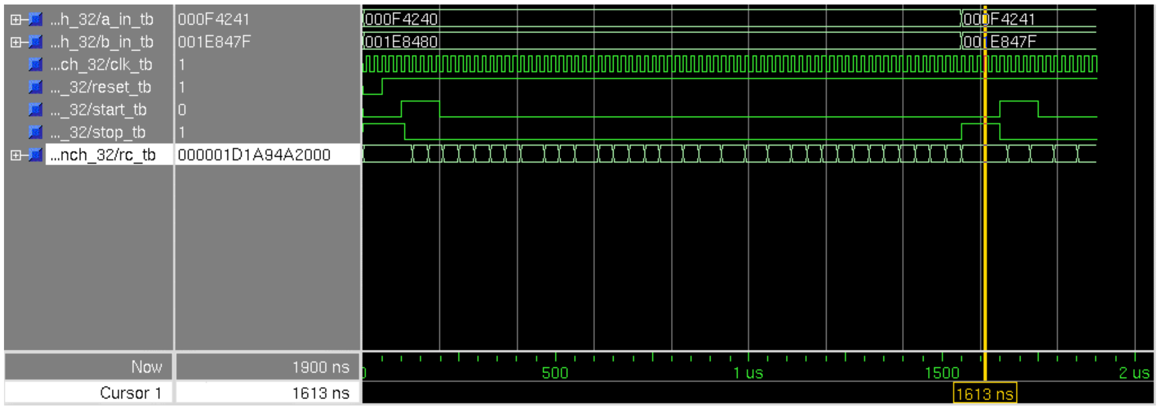


Figure 4-4: Complete 32-Bit Multiplier Simulation Timing Diagram

5 CONCLUSION

The goal of the project, to design an 8-by-8 bit multiplier using the add and shift method, was achieved. The multiplier was designed, coded in VHDL and simulated using the appropriate CAD tools. As an added value to the project, several designs were implemented in order to compare speed and area. Designs were also synthesized using various targets and place & route was also performed for a particular Xilinx device. The design was also expanded to a 32-bit version and similar synthesis and place & route analysis was achieved.

The added work performed for this project, can themselves become a complete topic for another project. For example, the effect of adder design on speed and area can itself be a large research project while optimizing constraints for efficient synthesis is often a critical step when implementing designs in industry. Everyone wants the highest density and fastest speed achievable from their designs.

APPENDIX A: VHDL SOURCE CODE

VHDL: Controller

```
-------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Controller
--
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-- Date        : Mon Oct 27 12:36:47 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Controller is a finite state machine
--               that performs the following in each
--               state:
--        IDLE     > samples the START signal
--        INIT     > commands the registers to be
--                   loaded
--        TEST     > samples the LSB
--        ADD      > indicates the Add result to be stored
--        SHIFT    > commands the register to be shifted
--
-------------------------------------------------------
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity  Controller  is
port (reset     : in std_logic ;
      clk       : in std_logic ;
      START     : in std_logic ;
      LSB       : in std_logic ;
      ADD_cmd   : out std_logic ;
      SHIFT_cmd : out std_logic ;
      LOAD_cmd  : out std_logic ;
      STOP      : out std_logic);
end;

-------------------------------------------------------
architecture  rtl  of  Controller  is

signal temp_count : std_logic_vector(2 downto 0);

-- declare states
type state_typ is (IDLE, INIT, TEST, ADD, SHIFT);
signal state : state_typ;

begin

process (clk, reset)
  begin
    if reset='0' then
      state <= IDLE;
      temp_count <= "000";


    elsif (clk'event and clk='1') then
```

```
      case state is

        when IDLE =>
          if START = '1' then
            state <= INIT;
          else
            state <= IDLE;
          end if;

        when INIT =>
          state <= TEST;

        when TEST =>
          if LSB = '0' then
            state <= SHIFT;
          else
            state <= ADD;
          end if;

        when ADD =>
          state <= SHIFT;

        when SHIFT =>
          if temp_count = "111" then  -- verify if finished
            temp_count <= "000";      -- re-initialize counter
            state <= IDLE;            -- ready for next multiply
          else
            temp_count <= temp_count + 1; -- increment counter
            state <= TEST;
          end if;

      end case;

    end if;

  end process;

  STOP <= '1' when state = IDLE else '0';
  ADD_cmd <= '1' when state = ADD else '0';
  SHIFT_cmd <= '1' when state = SHIFT else '0';
  LOAD_cmd <= '1' when state = INIT else '0';


end rtl;
```

VHDL: Multiplicand

Behavioral D Flip-Flop:

```
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  DFF
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 13:32:59 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : DFF is an active high D flip flop
--               with asynchronous clear.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  DFF  is
port (reset    : in  std_logic ;
      clk      : in  std_logic ;
      D        : in  std_logic ;
      Q        : out std_logic);
end;

------------------------------------------------------
architecture  behav  of  DFF  is
begin

  process (clk, reset)
  begin
    if reset='0' then
      Q <= '0';    -- clear register

    elsif (clk'event and clk='1') then
        Q <= D;    -- load register
    end if;
  end process;
end behav;
```

Structural D Flip-Flop:

```
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 13:32:59 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : DFF is an active high D flip flop
--               with asynchronous clear.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  DFF  is
```

```
port (reset    : in  std_logic ;
      clk      : in  std_logic ;
      D        : in  std_logic ;
      Q        : out std_logic);
end;

------------------------------------------------------
architecture  struc  of  DFF  is

signal NAND_temp : std_logic_vector(6 downto 1);

component NAND_2
      port (
            IN1   : in  std_logic;
            IN2   : in  std_logic;
            OUT1  : out std_logic
            );
end component;

component NAND_3
      port (
            IN1   : in  std_logic;
            IN2   : in  std_logic;
            IN3   : in  std_logic;
            OUT1  : out std_logic
            );
end component;

begin

NAND1: NAND_2 port map (NAND_temp(4), NAND_temp(2), NAND_temp(1));
NAND2: NAND_3 port map (NAND_temp(1), clk, reset, NAND_temp(2));
NAND3: NAND_3 port map (NAND_temp(2), clk, NAND_temp(4), NAND_temp(3));
NAND4: NAND_3 port map (NAND_temp(3), D, reset, NAND_temp(4));
NAND5: NAND_2 port map (NAND_temp(2), NAND_temp(6), NAND_temp(5));
NAND6: NAND_3 port map (NAND_temp(5), NAND_temp(3), reset, NAND_temp(6));

Q <= NAND_temp(5);

end struc;


------------------------------------------------------
--2 Input NAND
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  NAND_2  is
port (IN1   : in   std_logic ;
      IN2   : in   std_logic ;
      OUT1  : out  std_logic);
end;

------------------------------------------------------
architecture  struc  of  NAND_2  is
begin

        OUT1 <= NOT(IN1 AND IN2);

end struc;

------------------------------------------------------
--3 Input NAND
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  NAND_3  is
port (IN1   : in   std_logic ;
      IN2   : in   std_logic ;
      IN3   : in   std_logic ;
      OUT1  : out  std_logic);
end;
------------------------------------------------------------
architecture  struc  of  NAND_3  is
begin

        OUT1 <= NOT(IN1 AND IN2 AND IN3);

end struc;
```

Multiplicand 8-Bit Register:

```
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplicand
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 13:32:59 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Multiplicand is an 8-bit register
--               that is loaded when the LOAD_cmd is
--               received and cleared with reset.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Multiplicand  is
port (reset    : in  std_logic ;
      A_in     : in  std_logic_vector (7 downto 0);
      LOAD_cmd : in  std_logic ;
      RA       : out std_logic_vector (7 downto 0));
end;

------------------------------------------------------
architecture  struc  of  Multiplicand  is

component DFF
      port (
            reset  : in  std_logic;
            clk    : in  std_logic;
            D      : in  std_logic;
            Q      : out std_logic
            );
end component;

begin

DFFs: for i in 7 downto 0 generate

    DFFReg:DFF port map (reset, LOAD_cmd, A_in(i), RA(i));

end generate;

end struc;
```

VHDL: Mulitplier_Result

```
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplier_Result
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 14:13:51 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Multiplier_Result performs the
--               following:
--                 > loads B_in into register upon
--                   receiving LOAD_cmd
--                 > loads Adder output into register
--                   upon receiving ADD_cmd
--                 > shifts register right upon
--                   receiving SHIFT_cmd
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Multiplier_Result  is
port (reset     : in  std_logic ;
      clk       : in  std_logic ;
      B_in      : in  std_logic_vector (7 downto 0);
      LOAD_cmd  : in  std_logic ;
      SHIFT_cmd : in  std_logic ;
      ADD_cmd   : in  std_logic ;
      Add_out   : in  std_logic_vector (7 downto 0);
      C_out     : in  std_logic ;
      RC        : out std_logic_vector (15 downto 0);
      LSB       : out std_logic ;
      RB        : out std_logic_vector (7 downto 0));
end;

------------------------------------------------------
architecture  rtl  of  Multiplier_Result  is
signal temp_register  : std_logic_vector(16 downto 0);
signal temp_Add       : std_logic;

begin

process (clk, reset)
  begin
    if reset='0' then
      temp_register <= (others =>'0');  -- initialize temporary register
      temp_Add <= '0';

    elsif (clk'event and clk='1') then
      if LOAD_cmd = '1' then
        temp_register (16 downto 8) <= (others => '0');
        temp_register(7 downto 0) <= B_in;  -- load B_in into register
      end if;

      if ADD_cmd = '1' then
        temp_Add <= '1';
      end if;

      if SHIFT_cmd = '1' then
        if temp_Add = '1' then
```

```
          -- store adder output while shifting register right 1 bit
          temp_Add <= '0';
          temp_register <= '0' & C_out & Add_out & temp_register (7 downto 1);
        else
          -- no add - simply shift right 1 bit
          temp_register <= '0' & temp_register (16 downto 1);
        end if;
      end if;

    end if;
  end process;

  RB <= temp_register(15 downto 8);
  LSB <= temp_register(0);
  RC <= temp_register(15 downto 0);

end rtl;
```

VHDL: Adder

Full Adder Block:

```
-------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Full_Adder
--
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Basic Full Adder Block
--
-------------------------------------------------------
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Full_Adder  is
port (X     : in  std_logic;
      Y     : in  std_logic;
      C_in  : in  std_logic;
      Sum   : out std_logic ;
      C_out : out std_logic);
end;

-------------------------------------------------------
architecture  struc  of  Full_Adder  is
begin

Sum <= X xor Y xor C_in;
C_out <= (X and Y) or (X and C_in) or (Y and C_in);

end struc;
```

8-Bit Ripple Carry Adder:

```
-------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  RCA
--
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : RCA is an 8-bit ripple carry
--               adder composed of 8 basic full
--               adder blocks.
--
-------------------------------------------------------
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  RCA  is
port (RA      : in  std_logic_vector (7 downto 0);
      RB      : in  std_logic_vector (7 downto 0);
      C_out   : out std_logic ;
      Add_out : out std_logic_vector (7 downto 0));
end;

------------------------------------------------------
architecture  struc  of  RCA  is

signal c_temp : std_logic_vector(7 downto 0);

component Full_Adder
      port (
            X     : in  std_logic;
            Y     : in  std_logic;
            C_in  : in std_logic;
            Sum   : out std_logic;
            C_out : out std_logic
            );
end component;


begin

  c_temp(0) <= '0';  -- carry in of RCA is 0

Adders: for i in 7 downto 0 generate

-- assemble first 7 adders from 0 to 6
  Low: if i/=7 generate
    FA:Full_Adder port map (RA(i), RB(i), c_temp(i), Add_out(i), c_temp(i+1));
  end generate;

-- assemble last adder
  High: if i=7 generate
    FA:Full_Adder port map (RA(7), RB(7), c_temp(i), Add_out(7), C_out);
  end generate;

end generate;

end struc;
```

8-Bit Carry Select Adder (Composed of 4-Bit RCA):

```
----------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  RCA4
--
------------------------------------------------------
------------------------------------------
------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : RCA is an 4-bit ripple carry
--               adder composed of 8 basic full
--               adder blocks.
--
------------------------------------------
------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  RCA4  is
port (C_in    : in  std_logic;
      RA      : in  std_logic_vector (3 downto 0);
      RB      : in  std_logic_vector (3 downto 0);
      C_out   : out std_logic ;
      Add_out : out std_logic_vector (3 downto 0));
```

```
end;

------------------------------------------
architecture  rtl  of  RCA4  is

signal c_temp : std_logic_vector(3 downto 1);

component Full_Adder
      port (
            X     : in  std_logic;
            Y     : in  std_logic;
            C_in  : in std_logic;
            Sum   : out std_logic;
            C_out : out std_logic
            );
end component;


begin

Adders: for i in 3 downto 0 generate

  Low: if i=0 generate
    FA:Full_Adder port map (RA(0), RB(0), C_in, Add_out(0), c_temp(i+1));
  end generate;

  Mid: if (i>0 and i<3) generate
    FA:Full_Adder port map (RA(i), RB(i), c_temp(i), Add_out(i), c_temp(i+1));
  end generate;

  High: if i=3 generate
    FA:Full_Adder port map (RA(3), RB(3), c_temp(i), Add_out(3), C_out);
  end generate;

end generate;

end rtl;

----------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  CSA8
--
------------------------------------------------------
------------------------------------------
------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : CSA8 is an 8 bit carry select adder
--
------------------------------------------
------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  CSA8  is
port (RA      : in  std_logic_vector (7 downto 0);
      RB      : in  std_logic_vector (7 downto 0);
      C_out   : out std_logic ;
      Add_out : out std_logic_vector (7 downto 0));
end;

------------------------------------------
architecture  rtl  of  CSA8  is

signal c_temp : std_logic_vector(5 downto 0);
signal add_temp0 : std_logic_vector(3 downto 0);
signal add_temp1 : std_logic_vector(3 downto 0);
```

```
component RCA4
      port (
            C_in     : in  std_logic;
            RA    : in  std_logic_vector(3 downto 0);
            RB  : in std_logic_vector(3 downto 0);
            Add_out   : out std_logic_vector(3 downto 0);
            C_out : out std_logic
            );
end component;


begin

  c_temp(0) <= '0';
  c_temp(2) <= '0';
  c_temp(3) <= '1';


  inst_RCA1: RCA4
    port map (
              C_in => c_temp(0),
              RA => RA(3 downto 0),
              RB => RB(3 downto 0),
              Add_out => Add_out(3 downto 0),
              C_out => c_temp(1)
              );
  inst_RCA2: RCA4
    port map (
              C_in => c_temp(2),
              RA => RA(7 downto 4),
              RB => RB(7 downto 4),
              Add_out => add_temp0,
              C_out => c_temp(4)
              );
  inst_RCA3: RCA4
    port map (
              C_in => c_temp(3),
              RA => RA(7 downto 4),
              RB => RB(7 downto 4),
              Add_out => add_temp1,
              C_out => c_temp(5)
              );

Add_out (7 downto 4) <= add_temp0 when c_temp(1)='0' else
                        add_temp1 when c_temp(1)='1' else
                        "ZZZZ";


C_out <= (c_temp(1) and C_temp(5)) or c_temp(4);

end rtl;
```

VHDL: Multiplier

```
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplier
--
--  Description : Complete multiplier
--
------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
--library synplify;             -- required for synthesis
--use synplify.attributes.all;  -- required for synthesis


entity Multiplier is
  port (
        A_in  : in  std_logic_vector(7 downto 0 );
        B_in  : in  std_logic_vector(7 downto 0 );
        clk   : in  std_logic;
        reset : in  std_logic;
        START : in  std_logic;
        RC    : out std_logic_vector(15 downto 0 );
        STOP  : out std_logic);

end Multiplier;

use work.all;
architecture rtl of Multiplier is

  signal ADD_cmd   : std_logic;
  signal Add_out   : std_logic_vector(7 downto 0 );
  signal C_out     : std_logic;
  signal LOAD_cmd  : std_logic;
  signal LSB       : std_logic;
  signal RA        : std_logic_vector(7 downto 0 );
  signal RB        : std_logic_vector(7 downto 0 );
  signal SHIFT_cmd : std_logic;

  component RCA
      port (
            RA      : in  std_logic_vector(7 downto 0 );
            RB      : in  std_logic_vector(7 downto 0 );
            C_out   : out std_logic;
            Add_out : out std_logic_vector(7 downto 0 )
            );
  end component;

  component Controller
      port (
            reset     : in  std_logic;
            clk       : in  std_logic;
            START     : in  std_logic;
            LSB       : in  std_logic;
            ADD_cmd   : out std_logic;
            SHIFT_cmd : out std_logic;
            LOAD_cmd  : out std_logic;
            STOP      : out std_logic
            );
  end component;

  component Multiplicand
      port (
            reset    : in  std_logic;
            A_in     : in  std_logic_vector(7 downto 0 );
            LOAD_cmd : in  std_logic;
            RA       : out std_logic_vector(7 downto 0 )
            );
  end component;

  component Multiplier_Result
      port (
            reset     : in  std_logic;
            clk       : in  std_logic;
            B_in      : in  std_logic_vector(7 downto 0 );
            LOAD_cmd  : in  std_logic;
            SHIFT_cmd : in  std_logic;
            ADD_cmd   : in  std_logic;
            Add_out   : in  std_logic_vector(7 downto 0 );
            C_out     : in  std_logic;
            RC        : out std_logic_vector(15 downto 0 );
            LSB       : out std_logic;
            RB        : out std_logic_vector(7 downto 0 )
            );
  end component;


begin

  inst_RCA: RCA
    port map (
              RA => RA(7 downto 0),
              RB => RB(7 downto 0),
              C_out => C_out,
              Add_out => Add_out(7 downto 0)
              );

  inst_Controller: Controller
    port map (
              reset => reset,
              clk => clk,
              START => START,
              LSB => LSB,
              ADD_cmd => ADD_cmd,
              SHIFT_cmd => SHIFT_cmd,
              LOAD_cmd => LOAD_cmd,
              STOP => STOP
              );

  inst_Multiplicand: Multiplicand
    port map (
              reset => reset,
              A_in => A_in(7 downto 0),
              LOAD_cmd => LOAD_cmd,
              RA => RA(7 downto 0)
              );

  inst_Multiplier_Result: Multiplier_Result
    port map (
              reset => reset,
              clk => clk,
              B_in => B_in(7 downto 0),
              LOAD_cmd => LOAD_cmd,
              SHIFT_cmd => SHIFT_cmd,
              ADD_cmd => ADD_cmd,
              Add_out => Add_out(7 downto 0),
              C_out => C_out,
              RC => RC(15 downto 0),
              LSB => LSB,
              RB => RB(7 downto 0)
              );
end rtl;
```

VHDL: TestBench

```
----------------------------------------------------
--
--  TESTBENCH
--
--  Author    : Giovanni D'Aliesio
--
--  Description : Testbench generates clk and reset
--                signals as well as numerous multiplies.
--                It multiplies A and B where A starts from
--                0 to 255 and B starts from 255 down to 0.
--                The inputs and the results are stored in
--                a text file (bus_log.txt) for off-line
--                comparison.
--
----------------------------------------------------
----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;             --required for file I/O
use ieee.std_logic_textio.all;  --required for file I/O


entity  TESTBENCH  is

end TESTBENCH;

architecture BEHAVIORAL of TESTBENCH is

component Multiplier
 port (
        A_in  : in  std_logic_vector(7 downto 0 );
        B_in  : in  std_logic_vector(7 downto 0 );
        clk   : in  std_logic;
        RC    : out std_logic_vector(15 downto 0 );
        reset : in  std_logic;
        START : in  std_logic;
        STOP  : out std_logic
        );
end component;

signal A_in_TB, B_in_TB : std_logic_vector(7 downto 0 );
signal clk_TB, reset_TB, START_TB : std_logic;
signal STOP_TB : std_logic;
signal RC_TB: std_logic_vector(15 downto 0);

begin

-- instantiate the Device Under Test
inst_DUT : Multiplier
  port map (
    A_in => A_in_TB,
    B_in => B_in_TB,
    clk => clk_TB,
    reset => reset_TB,
    RC => RC_TB,
    START => START_TB,
    STOP => STOP_TB);

-- Generate clock stimulus
STIMULUS_CLK : process
begin
  clk_TB <= '0';
  wait for 10 ns;
  clk_TB <= '1';
  wait for 10 ns;
end process STIMULUS_CLK;

-- Generate reset stimulus
STIMULUS_RST : process
begin
  reset_TB <= '0';
  wait for 50 ns;
  reset_TB <= '1';
  wait;
end process STIMULUS_RST;

-- Generate multiplication requests
STIMULUS_START : process

file logFile : text is out "bus_log.txt";  -- set output file name
variable L: line;
variable A_temp, B_temp, i : integer;

begin

  write(L, string'("A  B  Result"));  -- include heading in file
  writeline(logFile,L);
  A_temp := 0;    -- start A at 0
  B_temp := 255;  -- start B at 255
  i := 1;

  for i in 1 to 256 loop
    A_in_TB <= STD_LOGIC_VECTOR(to_unsigned(A_temp,8));
    B_in_TB <= STD_LOGIC_VECTOR(to_unsigned(B_temp,8));

    START_TB <= '0';
    wait for 100 ns;
    START_TB <= '1';  -- request the multiplier to start
    wait for 100 ns;
    START_TB <= '0';
    wait until STOP_TB = '1';  -- wait for the multiplier to finish
    hwrite(L, A_in_TB);        -- insert hex value of A in file
    write(L, string'(" "));
    hwrite(L, B_in_TB);        -- insert hex value of B in file
    write(L, string'(" "));
    hwrite(L, RC_TB);          -- insert hex value of result in file
    writeline(logFile,L);
    A_temp := A_temp + 1;      -- increment value of A (Multiplicand)
    B_temp := B_temp - 1;      -- decrement value of B (Multiplier)
  end loop;
  wait;

end process STIMULUS_START;

end BEHAVIORAL;
```

VHDL: 32-Bit Multiplier

```vhdl
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplier_Result
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 14:13:51 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Multiplier_Result performs the
--               following:
--                 > loads B_in into register upon
--                   receiving LOAD_cmd
--                 > loads Adder output into register
--                   upon receiving ADD_cmd
--                 > shifts register right upon
--                   receiving SHIFT_cmd
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Multiplier_Result  is
port (reset     : in  std_logic ;
      clk       : in  std_logic ;
      B_in      : in  std_logic_vector (31 downto 0);
      LOAD_cmd  : in  std_logic ;
      SHIFT_cmd : in  std_logic ;
      ADD_cmd   : in  std_logic ;
      Add_out   : in  std_logic_vector (31 downto 0);
      C_out     : in  std_logic ;
      RC        : out std_logic_vector (63 downto 0);
      LSB       : out std_logic ;
      RB        : out std_logic_vector (31 downto 0));
end;

------------------------------------------------------
architecture  rtl  of  Multiplier_Result  is
signal temp_register  : std_logic_vector(64 downto 0);
signal temp_Add       : std_logic;

begin

process (clk, reset)
  begin
    if reset='0' then
      temp_register <= (others =>'0');  -- initialize temporary register
      temp_Add <= '0';

    elsif (clk'event and clk='1') then
      if LOAD_cmd = '1' then
        temp_register (64 downto 32) <= (others => '0');
        temp_register(31 downto 0) <= B_in;  -- load B_in into register
      end if;

      if ADD_cmd = '1' then
        temp_Add <= '1';
      end if;

      if SHIFT_cmd = '1' then
        if temp_Add = '1' then
```

```
          -- store adder output while shifting register right 1 bit
          temp_Add <= '0';
          temp_register <= '0' & C_out & Add_out & temp_register (31 downto 1);
        else
          -- no add - simply shift right 1 bit
          temp_register <= '0' & temp_register (64 downto 1);
        end if;
      end if;

    end if;
  end process;

  RB <= temp_register(63 downto 32);
  LSB <= temp_register(0);
  RC <= temp_register(63 downto 0);

end rtl;


------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplicand
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 13:32:59 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Multiplicand is an 8-bit register
--               that is loaded when the LOAD_cmd is
--               received and cleared with reset.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Multiplicand  is
port (reset    : in  std_logic ;
      A_in     : in  std_logic_vector (31 downto 0);
      LOAD_cmd : in  std_logic ;
      RA       : out std_logic_vector (31 downto 0));
end;

------------------------------------------------------
architecture  struc  of  Multiplicand  is

component DFF
      port (
            reset  : in  std_logic;
            clk    : in  std_logic;
            D      : in  std_logic;
            Q      : out std_logic
            );
end component;

begin

DFFs: for i in 31 downto 0 generate

    DFFReg:DFF port map (reset, LOAD_cmd, A_in(i), RA(i));

end generate;

end struc;

------------------------------------------------------
------------------------------------------------------
```

```
------------------------------------------------------
-- Date        : Mon Oct 27 13:32:59 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : DFF is an active high D flip flop
--               with asynchronous clear.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  DFF  is
port (reset    : in  std_logic ;
      clk      : in  std_logic ;
      D        : in  std_logic ;
      Q        : out std_logic);
end;

------------------------------------------------------
architecture  behav  of  DFF  is
begin

  process (clk, reset)
  begin
    if reset='0' then
      Q <= '0';    -- clear register

    elsif (clk'event and clk='1') then
        Q <= D;    -- load register
    end if;
  end process;
end behav;



------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Controller
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Mon Oct 27 12:36:47 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Controller is a finite state machine
--               that performs the following in each
--               state:
--        IDLE     > samples the START signal
--        INIT     > commands the registers to be
--                   loaded
--        TEST     > samples the LSB
--        ADD      > indicates the Add result to be stored
--        SHIFT    > commands the register to be shifted
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity  Controller  is
port (reset     : in std_logic ;
      clk       : in std_logic ;
      START     : in std_logic ;
      LSB       : in std_logic ;
      ADD_cmd   : out std_logic ;
      SHIFT_cmd : out std_logic ;
      LOAD_cmd  : out std_logic ;
      STOP      : out std_logic);
end;

------------------------------------------------------
architecture  rtl  of  Controller  is

signal temp_count : std_logic_vector(4 downto 0);

-- declare states
type state_typ is (IDLE, INIT, TEST, ADD, SHIFT);
signal state : state_typ;

begin

process (clk, reset)
  begin
    if reset='0' then
      state <= IDLE;
      temp_count <= "00000";


    elsif (clk'event and clk='1') then

      case state is

        when IDLE =>
          if START = '1' then
            state <= INIT;
          else
            state <= IDLE;
          end if;

        when INIT =>
          state <= TEST;

        when TEST =>
          if LSB = '0' then
            state <= SHIFT;
          else
            state <= ADD;
          end if;

        when ADD =>
          state <= SHIFT;

        when SHIFT =>
          if temp_count = "11111" then  -- verify if finished
            temp_count <= "00000";      -- re-initialize counter
            state <= IDLE;            -- ready for next multiply
          else
            temp_count <= temp_count + 1; -- increment counter
            state <= TEST;
          end if;

      end case;

    end if;

  end process;

  STOP <= '1' when state = IDLE else '0';
  ADD_cmd <= '1' when state = ADD else '0';
  SHIFT_cmd <= '1' when state = SHIFT else '0';
  LOAD_cmd <= '1' when state = INIT else '0';


end rtl;
```

```
------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Full_Adder
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : Basic Full Adder Block
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  Full_Adder  is
port (X     : in  std_logic;
      Y     : in  std_logic;
      C_in  : in  std_logic;
      Sum   : out std_logic ;
      C_out : out std_logic);
end;

------------------------------------------------------
architecture  rtl  of  Full_Adder  is
begin

Sum <= X xor Y xor C_in;
C_out <= (X and Y) or (X and C_in) or (Y and C_in);

end rtl;

------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  RCA
--
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
-- Date        : Wed Sep 24 12:50:50 2003
--
-- Author      : Giovanni D'Aliesio
--
-- Description : RCA is an 8-bit ripple carry
--               adder composed of 8 basic full
--               adder blocks.
--
------------------------------------------------------
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity  RCA  is
port (RA      : in  std_logic_vector (31 downto 0);
      RB      : in  std_logic_vector (31 downto 0);
      C_out   : out std_logic ;
      Add_out : out std_logic_vector (31 downto 0));
end;

------------------------------------------------------
architecture  rtl  of  RCA  is

signal c_temp : std_logic_vector(31 downto 0);

component Full_Adder
      port (
            X     : in  std_logic;
            Y     : in  std_logic;
            C_in  : in std_logic;
            Sum   : out std_logic;
            C_out : out std_logic
            );
end component;


begin

  c_temp(0) <= '0';  -- carry in of RCA is 0

Adders: for i in 31 downto 0 generate

-- assemble first 31 adders from 0 to 31
  Low: if i/=31 generate
    FA:Full_Adder port map (RA(i), RB(i), c_temp(i), Add_out(i), c_temp(i+1));
  end generate;

-- assemble last adder
  High: if i=31 generate
    FA:Full_Adder port map (RA(31), RB(31), c_temp(i), Add_out(31), C_out);
  end generate;

end generate;

end rtl;

------------------------------------------------------
--
--  Library Name :  DSD
--  Unit    Name :  Multiplier
--
--  Description : Complete multiplier
--
------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
--library synplify;             -- required for synthesis
--use synplify.attributes.all;  -- required for synthesis


entity Multiplier_32 is
  port (
        A_in  : in  std_logic_vector(31 downto 0 );
        B_in  : in  std_logic_vector(31 downto 0 );
        clk   : in  std_logic;
        reset : in  std_logic;
        START : in  std_logic;
        RC    : out std_logic_vector(63 downto 0 );
        STOP  : out std_logic);

end Multiplier_32;

use work.all;
architecture rtl of Multiplier_32 is

  signal ADD_cmd   : std_logic;
  signal Add_out   : std_logic_vector(31 downto 0 );
  signal C_out     : std_logic;
  signal LOAD_cmd  : std_logic;
  signal LSB       : std_logic;
  signal RA        : std_logic_vector(31 downto 0 );
  signal RB        : std_logic_vector(31 downto 0 );
  signal SHIFT_cmd : std_logic;

  component RCA
      port (
            RA      : in  std_logic_vector(31 downto 0 );
            RB      : in  std_logic_vector(31 downto 0 );
            C_out   : out std_logic;
            Add_out : out std_logic_vector(31 downto 0 )
            );
  end component;

  component Controller
      port (
            reset     : in  std_logic;
            clk       : in  std_logic;
            START     : in  std_logic;
            LSB       : in  std_logic;
            ADD_cmd   : out std_logic;
            SHIFT_cmd : out std_logic;
            LOAD_cmd  : out std_logic;
            STOP      : out std_logic
            );
  end component;

  component Multiplicand
      port (
            reset    : in  std_logic;
            A_in     : in  std_logic_vector(31 downto 0 );
            LOAD_cmd : in  std_logic;
            RA       : out std_logic_vector(31 downto 0 )
            );
  end component;

  component Multiplier_Result
      port (
            reset     : in  std_logic;
            clk       : in  std_logic;
            B_in      : in  std_logic_vector(31 downto 0 );
            LOAD_cmd  : in  std_logic;
            SHIFT_cmd : in  std_logic;
            ADD_cmd   : in  std_logic;
            Add_out   : in  std_logic_vector(31 downto 0 );
            C_out     : in  std_logic;
            RC        : out std_logic_vector(63 downto 0 );
            LSB       : out std_logic;
            RB        : out std_logic_vector(31 downto 0 )
            );
  end component;


begin

  inst_RCA: RCA
    port map (
              RA => RA(31 downto 0),
              RB => RB(31 downto 0),
              C_out => C_out,
              Add_out => Add_out(31 downto 0)
              );

  inst_Controller: Controller
    port map (
              reset => reset,
              clk => clk,
              START => START,
              LSB => LSB,
              ADD_cmd => ADD_cmd,
              SHIFT_cmd => SHIFT_cmd,
              LOAD_cmd => LOAD_cmd,
              STOP => STOP
              );

  inst_Multiplicand: Multiplicand
    port map (
              reset => reset,
              A_in => A_in(31 downto 0),
              LOAD_cmd => LOAD_cmd,
              RA => RA(31 downto 0)
              );

  inst_Multiplier_Result: Multiplier_Result
    port map (
              reset => reset,
              clk => clk,
              B_in => B_in(31 downto 0),
              LOAD_cmd => LOAD_cmd,
              SHIFT_cmd => SHIFT_cmd,
              ADD_cmd => ADD_cmd,
              Add_out => Add_out(31 downto 0),
              C_out => C_out,
              RC => RC(63 downto 0),
              LSB => LSB,
              RB => RB(31 downto 0)
              );
end rtl;
```

VHDL: 32-Bit Multiplier TestBench

```
----------------------------------------------------
--
--  TESTBENCH
--
--  Author   : Giovanni D'Aliesio
--
--  Description : Testbench generates clk and reset
--                signals as well as numerous multiplies.
--                It multiplies A and B where A starts from
--                0 to 255 and B starts from 255 down to 0.
--                The inputs and the results are stored in
--                a text file (bus_log.txt) for off-line
--                comparison.
--
----------------------------------------------------
----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;             --required for file I/O
use ieee.std_logic_textio.all;  --required for file I/O


entity  TESTBENCH_32  is

end TESTBENCH_32;

architecture BEHAVIORAL of TESTBENCH_32 is

component Multiplier_32
 port (
        A_in  : in  std_logic_vector(31 downto 0 );
        B_in  : in  std_logic_vector(31 downto 0 );
        clk   : in  std_logic;
        RC    : out std_logic_vector(63 downto 0 );
        reset : in  std_logic;
        START : in  std_logic;
        STOP  : out std_logic
        );
end component;

signal A_in_TB, B_in_TB : std_logic_vector(31 downto 0 );
signal clk_TB, reset_TB, START_TB : std_logic;
signal STOP_TB : std_logic;
signal RC_TB: std_logic_vector(63 downto 0);

begin

-- instantiate the Device Under Test
inst_DUT : Multiplier_32
  port map (
    A_in => A_in_TB,
    B_in => B_in_TB,
    clk => clk_TB,
    reset => reset_TB,
    RC => RC_TB,
    START => START_TB,
    STOP => STOP_TB);

-- Generate clock stimulus
STIMULUS_CLK : process
begin
  clk_TB <= '0';
  wait for 10 ns;
  clk_TB <= '1';
  wait for 10 ns;
end process STIMULUS_CLK;

-- Generate reset stimulus
STIMULUS_RST : process
begin
  reset_TB <= '0';
  wait for 50 ns;
  reset_TB <= '1';
  wait;
end process STIMULUS_RST;

-- Generate multiplication requests
STIMULUS_START : process

file logFile : text is out "bus_log32.txt";  -- set output file name
variable L: line;
variable A_temp, B_temp, i : integer;

begin

  write(L, string'("A  B  Result"));  -- include heading in file
  writeline(logFile,L);
  A_temp := 1000000;    -- start A at 1000000
  B_temp := 2000000;  -- start B at 2000000
  i := 100000;

  for i in 100000 to 200000 loop
    A_in_TB <= STD_LOGIC_VECTOR(to_unsigned(A_temp,32));
    B_in_TB <= STD_LOGIC_VECTOR(to_unsigned(B_temp,32));

    START_TB <= '0';
    wait for 100 ns;
    START_TB <= '1';  -- request the multiplier to start
    wait for 100 ns;
    START_TB <= '0';
    wait until STOP_TB = '1';  -- wait for the multiplier to finish
    hwrite(L, A_in_TB);        -- insert hex value of A in file
    write(L, string'(" "));
    hwrite(L, B_in_TB);        -- insert hex value of B in file
    write(L, string'(" "));
    hwrite(L, RC_TB);          -- insert hex value of result in file
    writeline(logFile,L);
    A_temp := A_temp + 1;      -- increment value of A (Multiplicand)
    B_temp := B_temp - 1;      -- decrement value of B (Multiplier)
  end loop;
  wait;

end process STIMULUS_START;

end BEHAVIORAL;
```

APPENDIX B: SIMULATION RESULT FILES

Simulation output for multiplier:
bus_log.txt:

```
A  B  Result                 A  B  Result                 A  B  Result
00 FF 0000                   57 A8 3918                   AD 52 376A
01 FE 00FE                   58 A7 3968                   AE 51 370E
02 FD 01FA                   59 A6 39B6                   AF 50 36B0
03 FC 02F4                   5A A5 3A02                   B0 4F 3650
04 FB 03EC                   5B A4 3A4C                   B1 4E 35EE
05 FA 04E2                   5C A3 3A94                   B2 4D 358A
06 F9 05D6                   5D A2 3ADA                   B3 4C 3524
07 F8 06C8                   5E A1 3B1E                   B4 4B 34BC
08 F7 07B8                   5F A0 3B60                   B5 4A 3452
09 F6 08A6                   60 9F 3BA0                   B6 49 33E6
0A F5 0992                   61 9E 3BDE                   B7 48 3378
0B F4 0A7C                   62 9D 3C1A                   B8 47 3308
0C F3 0B64                   63 9C 3C54                   B9 46 3296
0D F2 0C4A                   64 9B 3C8C                   BA 45 3222
0E F1 0D2E                   65 9A 3CC2                   BB 44 31AC
0F F0 0E10                   66 99 3CF6                   BC 43 3134
10 EF 0EF0                   67 98 3D28                   BD 42 30BA
11 EE 0FCE                   68 97 3D58                   BE 41 303E
12 ED 10AA                   69 96 3D86                   BF 40 2FC0
13 EC 1184                   6A 95 3DB2                   C0 3F 2F40
14 EB 125C                   6B 94 3DDC                   C1 3E 2EBE
15 EA 1332                   6C 93 3E04                   C2 3D 2E3A
16 E9 1406                   6D 92 3E2A                   C3 3C 2DB4
17 E8 14D8                   6E 91 3E4E                   C4 3B 2D2C
18 E7 15A8                   6F 90 3E70                   C5 3A 2CA2
19 E6 1676                   70 8F 3E90                   C6 39 2C16
1A E5 1742                   71 8E 3EAE                   C7 38 2B88
1B E4 180C                   72 8D 3ECA                   C8 37 2AF8
1C E3 18D4                   73 8C 3EE4                   C9 36 2A66
1D E2 199A                   74 8B 3EFC                   CA 35 29D2
1E E1 1A5E                   75 8A 3F12                   CB 34 293C
1F E0 1B20                   76 89 3F26                   CC 33 28A4
20 DF 1BE0                   77 88 3F38                   CD 32 280A
21 DE 1C9E                   78 87 3F48                   CE 31 276E
22 DD 1D5A                   79 86 3F56                   CF 30 26D0
23 DC 1E14                   7A 85 3F62                   D0 2F 2630
24 DB 1ECC                   7B 84 3F6C                   D1 2E 258E
25 DA 1F82                   7C 83 3F74                   D2 2D 24EA
26 D9 2036                   7D 82 3F7A                   D3 2C 2444
27 D8 20E8                   7E 81 3F7E                   D4 2B 239C
28 D7 2198                   7F 80 3F80                   D5 2A 22F2
29 D6 2246                   80 7F 3F80                   D6 29 2246
2A D5 22F2                   81 7E 3F7E                   D7 28 2198
2B D4 239C                   82 7D 3F7A                   D8 27 20E8
2C D3 2444                   83 7C 3F74                   D9 26 2036
2D D2 24EA                   84 7B 3F6C                   DA 25 1F82
2E D1 258E                   85 7A 3F62                   DB 24 1ECC
2F D0 2630                   86 79 3F56                   DC 23 1E14
30 CF 26D0                   87 78 3F48                   DD 22 1D5A
31 CE 276E                   88 77 3F38                   DE 21 1C9E
32 CD 280A                   89 76 3F26                   DF 20 1BE0
33 CC 28A4                   8A 75 3F12                   E0 1F 1B20
34 CB 293C                   8B 74 3EFC                   E1 1E 1A5E
35 CA 29D2                   8C 73 3EE4                   E2 1D 199A
36 C9 2A66                   8D 72 3ECA                   E3 1C 18D4
37 C8 2AF8                   8E 71 3EAE                   E4 1B 180C
38 C7 2B88                   8F 70 3E90                   E5 1A 1742
39 C6 2C16                   90 6F 3E70                   E6 19 1676
3A C5 2CA2                   91 6E 3E4E                   E7 18 15A8
3B C4 2D2C                   92 6D 3E2A                   E8 17 14D8
```

```
3C C3 2DB4
3D C2 2E3A
3E C1 2EBE
3F C0 2F40
40 BF 2FC0
41 BE 303E
42 BD 30BA
43 BC 3134
44 BB 31AC
45 BA 3222
46 B9 3296
47 B8 3308
48 B7 3378
49 B6 33E6
4A B5 3452
4B B4 34BC
4C B3 3524
4D B2 358A
4E B1 35EE
4F B0 3650
50 AF 36B0
51 AE 370E
52 AD 376A
53 AC 37C4
54 AB 381C
55 AA 3872
56 A9 38C6
```

```
93 6C 3E04
94 6B 3DDC
95 6A 3DB2
96 69 3D86
97 68 3D58
98 67 3D28
99 66 3CF6
9A 65 3CC2
9B 64 3C8C
9C 63 3C54
9D 62 3C1A
9E 61 3BDE
9F 60 3BA0
A0 5F 3B60
A1 5E 3B1E
A2 5D 3ADA
A3 5C 3A94
A4 5B 3A4C
A5 5A 3A02
A6 59 39B6
A7 58 3968
A8 57 3918
A9 56 38C6
AA 55 3872
AB 54 381C
AC 53 37C4
```

```
E9 16 1406
EA 15 1332
EB 14 125C
EC 13 1184
ED 12 10AA
EE 11 0FCE
EF 10 0EF0
F0 0F 0E10
F1 0E 0D2E
F2 0D 0C4A
F3 0C 0B64
F4 0B 0A7C
F5 0A 0992
F6 09 08A6
F7 08 07B8
F8 07 06C8
F9 06 05D6
FA 05 04E2
FB 04 03EC
FC 03 02F4
FD 02 01FA
FE 01 00FE
FF 00 0000
```

Simulation output for 32-bit multiplier:
bus_log.txt:

```
A  B  Result
000F4240 001E8480 000001D1A94A2000
000F4241 001E847F 000001D1A959623F
000F4242 001E847E 000001D1A968A47C
000F4243 001E847D 000001D1A977E6B7
000F4244 001E847C 000001D1A98728F0
000F4245 001E847B 000001D1A9966B27
000F4246 001E847A 000001D1A9A5AD5C
000F4247 001E8479 000001D1A9B4EF8F
000F4248 001E8478 000001D1A9C431C0
000F4249 001E8477 000001D1A9D373EF
000F424A 001E8476 000001D1A9E2B61C
000F424B 001E8475 000001D1A9F1F847
000F424C 001E8474 000001D1AA013A70
000F424D 001E8473 000001D1AA107C97
000F424E 001E8472 000001D1AA1FBEBC
000F424F 001E8471 000001D1AA2F00DF
000F4250 001E8470 000001D1AA3E4300
000F4251 001E846F 000001D1AA4D851F
000F4252 001E846E 000001D1AA5CC73C
000F4253 001E846D 000001D1AA6C0957
000F4254 001E846C 000001D1AA7B4B70
000F4255 001E846B 000001D1AA8A8D87
000F4256 001E846A 000001D1AA99CF9C
000F4257 001E8469 000001D1AAA911AF
000F4258 001E8468 000001D1AAB853C0
000F4259 001E8467 000001D1AAC795CF
000F425A 001E8466 000001D1AAD6D7DC
000F425B 001E8465 000001D1AAE619E7
000F425C 001E8464 000001D1AAF55BF0
000F425D 001E8463 000001D1AB049DF7
000F425E 001E8462 000001D1AB13DFFC
000F425F 001E8461 000001D1AB2321FF
000F4260 001E8460 000001D1AB326400
000F4261 001E845F 000001D1AB41A5FF
000F4262 001E845E 000001D1AB50E7FC
```

```
A  B  Result
000F4295 001E842B 000001D1AE5B0307
000F4296 001E842A 000001D1AE6A449C
000F4297 001E8429 000001D1AE79862F
000F4298 001E8428 000001D1AE88C7C0
000F4299 001E8427 000001D1AE98094F
000F429A 001E8426 000001D1AEA74ADC
000F429B 001E8425 000001D1AEB68C67
000F429C 001E8424 000001D1AEC5CDF0
000F429D 001E8423 000001D1AED50F77
000F429E 001E8422 000001D1AEE450FC
000F429F 001E8421 000001D1AEF3927F
000F42A0 001E8420 000001D1AF02D400
000F42A1 001E841F 000001D1AF12157F
000F42A2 001E841E 000001D1AF2156FC
000F42A3 001E841D 000001D1AF309877
000F42A4 001E841C 000001D1AF3FD9F0
000F42A5 001E841B 000001D1AF4F1B67
000F42A6 001E841A 000001D1AF5E5CDC
000F42A7 001E8419 000001D1AF6D9E4F
000F42A8 001E8418 000001D1AF7CDFC0
000F42A9 001E8417 000001D1AF8C212F
000F42AA 001E8416 000001D1AF9B629C
000F42AB 001E8415 000001D1AFAAA407
000F42AC 001E8414 000001D1AFB9E570
000F42AD 001E8413 000001D1AFC926D7
000F42AE 001E8412 000001D1AFD8683C
000F42AF 001E8411 000001D1AFE7A99F
000F42B0 001E8410 000001D1AFF6EB00
000F42B1 001E840F 000001D1B0062C5F
000F42B2 001E840E 000001D1B0156DBC
000F42B3 001E840D 000001D1B024AF17
000F42B4 001E840C 000001D1B033F070
000F42B5 001E840B 000001D1B04331C7
000F42B6 001E840A 000001D1B052731C
000F42B7 001E8409 000001D1B061B46F
```

```
000F4263 001E845D 000001D1AB6029F7
000F4264 001E845C 000001D1AB6F6BF0
000F4265 001E845B 000001D1AB7EADE7
000F4266 001E845A 000001D1AB8DEFDC
000F4267 001E8459 000001D1AB9D31CF
000F4268 001E8458 000001D1ABAC73C0
000F4269 001E8457 000001D1ABBBB5AF
000F426A 001E8456 000001D1ABCAF79C
000F426B 001E8455 000001D1ABDA3987
000F426C 001E8454 000001D1ABE97B70
000F426D 001E8453 000001D1ABF8BD57
000F426E 001E8452 000001D1AC07FF3C
000F426F 001E8451 000001D1AC17411F
000F4270 001E8450 000001D1AC268300
000F4271 001E844F 000001D1AC35C4DF
000F4272 001E844E 000001D1AC4506BC
000F4273 001E844D 000001D1AC544897
000F4274 001E844C 000001D1AC638A70
000F4275 001E844B 000001D1AC72CC47
000F4276 001E844A 000001D1AC820E1C
000F4277 001E8449 000001D1AC914FEF
000F4278 001E8448 000001D1ACA091C0
000F4279 001E8447 000001D1ACAFD38F
000F427A 001E8446 000001D1ACBF155C
000F427B 001E8445 000001D1ACCE5727
000F427C 001E8444 000001D1ACDD98F0
000F427D 001E8443 000001D1ACECDAB7
000F427E 001E8442 000001D1ACFC1C7C
000F427F 001E8441 000001D1AD0B5E3F
000F4280 001E8440 000001D1AD1AA000
000F4281 001E843F 000001D1AD29E1BF
000F4282 001E843E 000001D1AD39237C
000F4283 001E843D 000001D1AD486537
000F4284 001E843C 000001D1AD57A6F0
000F4285 001E843B 000001D1AD66E8A7
000F4286 001E843A 000001D1AD762A5C
000F4287 001E8439 000001D1AD856C0F
000F4288 001E8438 000001D1AD94ADC0
000F4289 001E8437 000001D1ADA3EF6F
000F428A 001E8436 000001D1ADB3311C
000F428B 001E8435 000001D1ADC272C7
000F428C 001E8434 000001D1ADD1B470
000F428D 001E8433 000001D1ADE0F617
000F428E 001E8432 000001D1ADF037BC
000F428F 001E8431 000001D1ADFF795F
000F4290 001E8430 000001D1AE0EBB00
000F4291 001E842F 000001D1AE1DFC9F
000F4292 001E842E 000001D1AE2D3E3C
000F4293 001E842D 000001D1AE3C7FD7
000F4294 001E842C 000001D1AE4BC170
```

```
000F42B8 001E8408 000001D1B070F5C0
000F42B9 001E8407 000001D1B080370F
000F42BA 001E8406 000001D1B08F785C
000F42BB 001E8405 000001D1B09EB9A7
000F42BC 001E8404 000001D1B0ADFAF0
000F42BD 001E8403 000001D1B0BD3C37
000F42BE 001E8402 000001D1B0CC7D7C
000F42BF 001E8401 000001D1B0DBBEBF
000F42C0 001E8400 000001D1B0EB0000
000F42C1 001E83FF 000001D1B0FA413F
000F42C2 001E83FE 000001D1B109827C
000F42C3 001E83FD 000001D1B118C3B7
000F42C4 001E83FC 000001D1B12804F0
000F42C5 001E83FB 000001D1B1374627
000F42C6 001E83FA 000001D1B146875C
000F42C7 001E83F9 000001D1B155C88F
000F42C8 001E83F8 000001D1B16509C0
000F42C9 001E83F7 000001D1B1744AEF
000F42CA 001E83F6 000001D1B1838C1C
000F42CB 001E83F5 000001D1B192CD47
000F42CC 001E83F4 000001D1B1A20E70
000F42CD 001E83F3 000001D1B1B14F97
000F42CE 001E83F2 000001D1B1C090BC
000F42CF 001E83F1 000001D1B1CFD1DF
000F42D0 001E83F0 000001D1B1DF1300
000F42D1 001E83EF 000001D1B1EE541F
000F42D2 001E83EE 000001D1B1FD953C
000F42D3 001E83ED 000001D1B20CD657
000F42D4 001E83EC 000001D1B21C1770
000F42D5 001E83EB 000001D1B22B5887
000F42D6 001E83EA 000001D1B23A999C
000F42D7 001E83E9 000001D1B249DAAF
000F42D8 001E83E8 000001D1B2591BC0
000F42D9 001E83E7 000001D1B2685CCF
000F42DA 001E83E6 000001D1B2779DDC
000F42DB 001E83E5 000001D1B286DEE7
000F42DC 001E83E4 000001D1B2961FF0
000F42DD 001E83E3 000001D1B2A560F7
000F42DE 001E83E2 000001D1B2B4A1FC
000F42DF 001E83E1 000001D1B2C3E2FF
000F42E0 001E83E0 000001D1B2D32400
000F42E1 001E83DF 000001D1B2E264FF
000F42E2 001E83DE 000001D1B2F1A5FC
000F42E3 001E83DD 000001D1B300E6F7
000F42E4 001E83DC 000001D1B31027F0
000F42E5 001E83DB 000001D1B31F68E7
000F42E6 001E83DA 000001D1B32EA9DC
000F42E7 001E83D9 000001D1B33DEACF
000F42E8 001E83D8 000001D1B34D2BC0
000F42E9 001E83D7 000001D1B35C6CAF
000F42EA 001E83D6 000001D1B36BAD9C
000F42EB 001E83D5 000001D1B37AEE87
000F42EC 001E83D4 000001D1B38A2F70
```

APPENDIX C: SYNTHESIS REPORT FILES

Synthesis Report File for RCA version with Behavioral DFF:

```
Performance Summary
*******************

Worst slack in design: 989.238
                            Requested     Estimated     Requested     Estimated
Clock
Starting Clock              Frequency     Frequency     Period        Period        Slack
Type
------------------------------------------------------------------------------------------
clk                         1.0 MHz       92.9 MHz      1000.000      10.762
989.238     inferred
load_cmd_inferred_clock     1.0 MHz       96.9 MHz      1000.000      10.316
989.684     inferred
==========================================================================================

Worst Paths Information
***********************
Path information for path number 1:
    - Setup time:                        -0.420
    = Required time:                     1000.420

    - Propagation  time:                 11.182
    = Slack (critical) :                 989.238

    Starting point:
inst_Multiplier_Result.temp_register[8] / Q
    Ending point:
inst_Multiplier_Result.temp_register[15] / D
    The start point is clocked by         clk [rising] on pin C
    The end   point is clocked by         clk [rising] on pin C

---------------------------------------
Resource Usage Report for Multiplier

Mapping to part: xcv50bg256-4
Cell usage:
FDCE            17 uses
FDC             15 uses
FDP             1 use
keepbuf         1 use
GND             1 use
VCC             1 use

I/O primitives:
IBUF           18 uses
OBUF           17 uses

BUFGP          1 use

I/O Register bits:                  8
Register bits not including I/Os:   25 (1%)

Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)

Mapping Summary:
Total  LUTs: 45 (2%)

Mapper successful!
```

Synthesis Report File for RCA version with Behavioral DFF Synthesized for Actel ACT3:

```
Performance Summary
*******************
Worst slack in design: 976.710
                            Requested     Estimated     Requested     Estimated
Clock
Starting Clock              Frequency     Frequency     Period        Period        Slack
Type
------------------------------------------------------------------------------------------
clk                         1.0 MHz       42.9 MHz      1000.000      23.290
976.710     inferred
load_cmd_inferred_clock     1.0 MHz       43.7 MHz      1000.000      22.865
977.135     inferred
==========================================================================================
Worst Paths Information
***********************

Path information for path number 1:
    - Setup time:                           0.000
    = Required time:                        1000.000

    - Propagation  time:                    23.290
    = Slack (critical) :                    976.710

    Starting point:                         inst_Multiplier_Result.temp_register[8] / q
    Ending point:                           inst_Multiplier_Result.temp_register[14] / d0
    The start point is clocked by           clk [falling] on pin s00
    The end   point is clocked by           load_cmd_inferred_clock [falling] on pin s00

Synthesized design as a chip
Resource Usage Report of Multiplier

Target Part: a1415a-3
Combinational Cells:    44 of 96 (46%)
Sequential Cells:    33 of 104 (32%)
Total Cells:         77 of 200 (39%)
Clock Buffers:       2
IO Cells:            36

Details:
   and2b:          2  comb:1
   and3:           1  comb:1
   buff:           2  comb:1
   cm8:            22 comb:1
   dfp1:           1  comb:2
   or2b:           1  comb:1
   xnor2:          4  comb:1
   xor2:           10 comb:1

   dfc1b:          8  seq:1
   dfe3c:          4  seq:1
   dfm8a:          21 seq:1

   clkbuf:         1  clock buffer
   inbuf:          18
   outbuf:         17

   clkint:         1  clock buffer

   keepbuf:        1
Found clock clk with period 1000.00ns
Found clock load_cmd_inferred_clock with period 1000.00ns
```

Synthesis Report File for RCA version with Structural DFF:

```
Performance Summary
*******************

Worst slack in design: 989.238
                   Requested     Estimated     Requested     Estimated
Clock
Starting Clock     Frequency     Frequency     Period        Period        Slack
Type
-------------------------------------------------------------------------------
clk                1.0 MHz       92.9 MHz      1000.000      10.762        989.238
inferred

Worst Paths Information
***********************

Path information for path number 1:
    - Setup time:                          -0.420
    = Required time:                       1000.420

    - Propagation  time:                   11.182
    = Slack (critical) :                   989.238

    Starting point:                        inst_Multiplier_Result.temp_register[8] / Q
    Ending point:                          inst_Multiplier_Result.temp_register[15] / D
    The start point is clocked by          clk [rising] on pin C
    The end   point is clocked by          clk [rising] on pin C

---------------------------------------
Resource Usage Report for Multiplier

Mapping to part: xcv50bg256-4
Cell usage:
FDCE            17 uses
FDC             7 uses
FDP             1 use
GND             1 use
VCC             1 use

I/O primitives:
IBUF           18 uses
OBUF           17 uses

BUFGP          1 use

I/O Register bits:                  0
Register bits not including I/Os:   25 (1%)

Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)

Mapping Summary:
Total  LUTs: 77 (5%)

Mapper successful!
```

Synthesis Report File for CSA version with Behavioral DFF:

```
Performance Summary
*******************

Worst slack in design: 991.398
                            Requested     Estimated     Requested     Estimated
Clock
Starting Clock              Frequency     Frequency     Period        Period        Slack
Type
```

```
clk                        1.0 MHz       116.2 MHz     1000.000      8.602      991.398
inferred
load_cmd_inferred_clock    1.0 MHz       122.6 MHz     1000.000      8.156      991.844
inferred

Worst Paths Information
**********************

Path information for path number 1:
    - Setup time:                        -0.420
    = Required time:                     1000.420

    - Propagation  time:                 9.022
    = Slack (critical) :                 991.398

    Starting point:                      inst_Multiplier_Result.temp_register[8] / Q
    Ending point:                        inst_Multiplier_Result.temp_register[12] / D
    The start point is clocked by        clk [rising] on pin C
    The end   point is clocked by        clk [rising] on pin C

---------------------------------------
Resource Usage Report for Multiplier

Mapping to part: xcv50bg256-4
Cell usage:
FDCE            17 uses
FDC             15 uses
FDP             1 use
keepbuf         1 use
GND             1 use
VCC             1 use

I/O primitives:
IBUF           18 uses
OBUF           17 uses

BUFGP          1 use

I/O Register bits:                  8
Register bits not including I/Os:   25 (1%)

Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)


Mapping Summary:
Total  LUTs: 50 (3%)

Mapper successful!
```

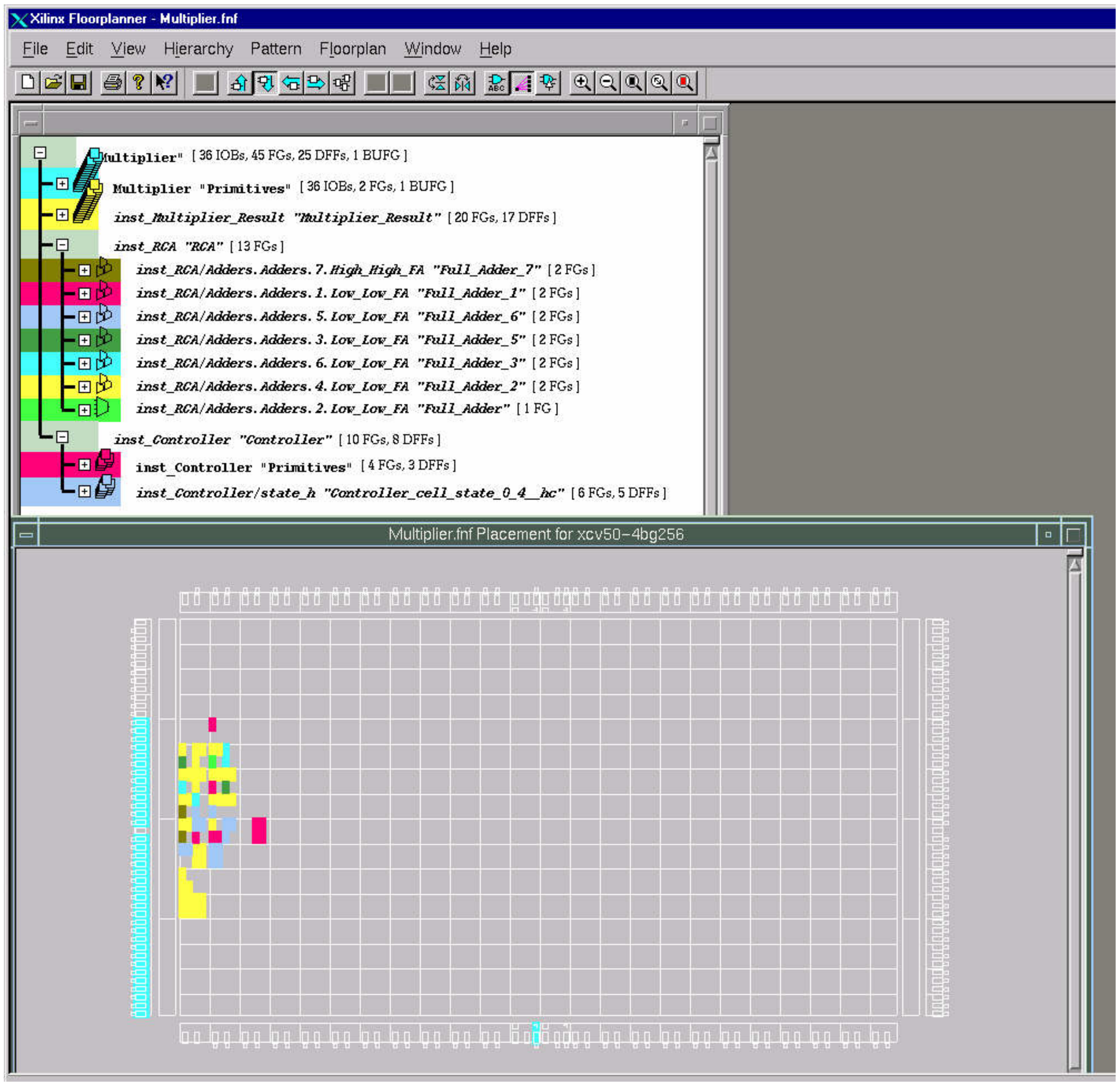
Synthesis Report File for 32-Bit Multiplier with RCA version with Behavioral DFF:

```
Performance Summary
*******************

Worst slack in design: 973.758
                           Requested     Estimated     Requested     Estimated
Clock
Starting Clock             Frequency     Frequency     Period        Period        Slack
Type
------------------------------------------------------------------------------------------
clk                        1.0 MHz       38.1 MHz      1000.000      26.242
973.758     inferred
load_cmd_inferred_clock    1.0 MHz       38.8 MHz      1000.000      25.796
974.204     inferred
==========================================================================================
Worst Paths Information
***********************
```

```
Path information for path number 1:
    - Setup time:                         -0.420
    = Required time:                      1000.420

    - Propagation  time:                  26.662
    = Slack (critical) :                  973.758

    Starting point:                       inst_Multiplier_Result.temp_register[32] / Q
    Ending point:                         inst_Multiplier_Result.temp_register[63] / D
    The start point is clocked by         clk [rising] on pin C
    The end   point is clocked by         clk [rising] on pin C

---------------------------------------
Resource Usage Report for Multiplier_32

Mapping to part: xcv50bg256-4
Cell usage:
FDCE            70 uses
FDC             36 uses
VCC             2 uses
GND             2 uses
FDP             1 use
MUXCY_L         4 uses
XORCY           5 uses
keepbuf         1 use

I/O primitives:
IBUF           66 uses
OBUF           65 uses

BUFG           1 use

BUFGP          1 use

I/O Register bits:                  32
Register bits not including I/Os:   75 (4%)

Global buffer usage summary
BUFGs + BUFGPs: 2 of 4 (50%)


Mapping Summary:
Total  LUTs: 142 (9%)

Mapper successful!
```

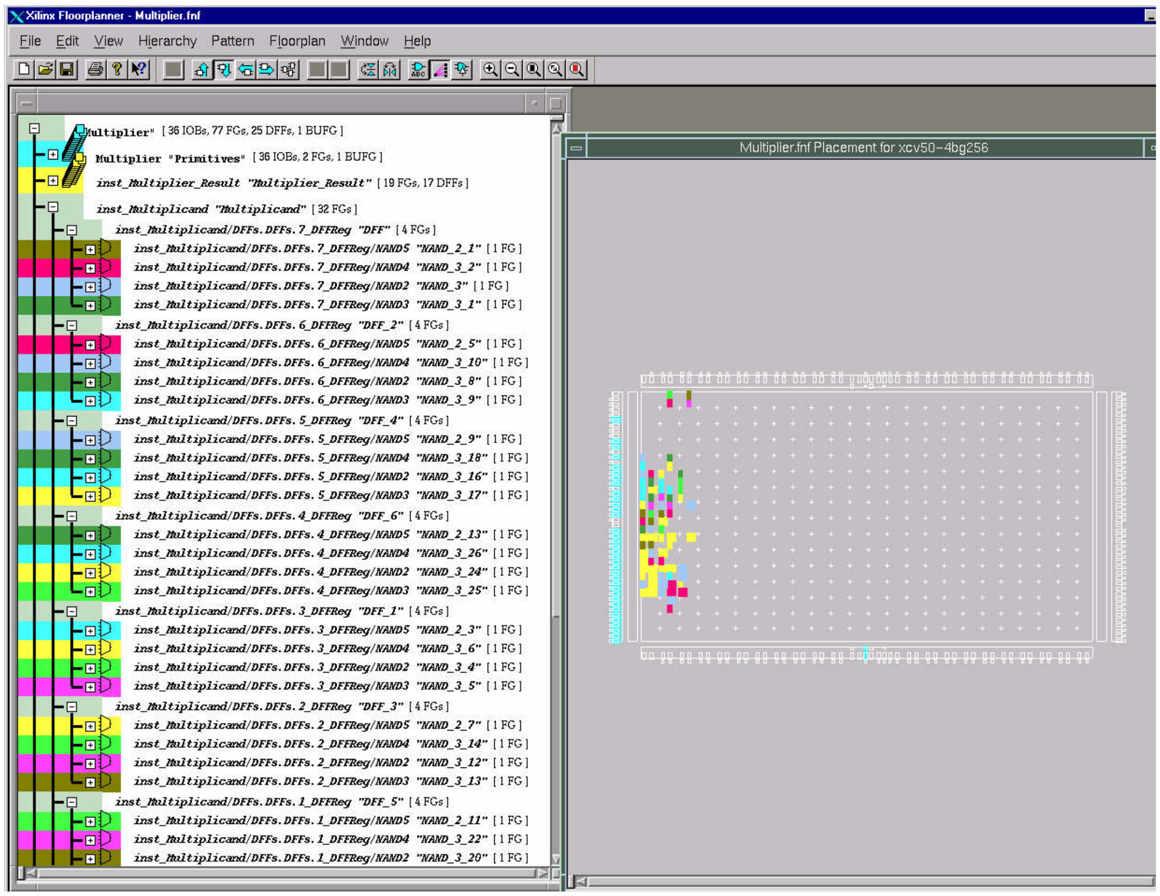
APPENDIX D: PLACE & ROUTE REPORT FILES

The following figures show how the design is mapped into the target device, namely the Xilinx XCV50. It is included here for illustrative purposes only since it was performed beyond the scope of this project and as such not much detail is included. The main goal of including these figures for each of the synthesized designs is such that the timing results and differences between designs becomes more apparent when viewing the physical structure. It can be clearly noted that due to the structure of the FPGA slice, the version of the design that uses a structural D flip-flop actually requires a lot more real estate than does it's behavioral equivalent.


Xilinx Floorplanner - Multiplier.fnf
File
Edit
View
Hierarchy
Pattern
Floorplan
Window
Help
Multiplier.fnf Placement for xcv50-4bg256

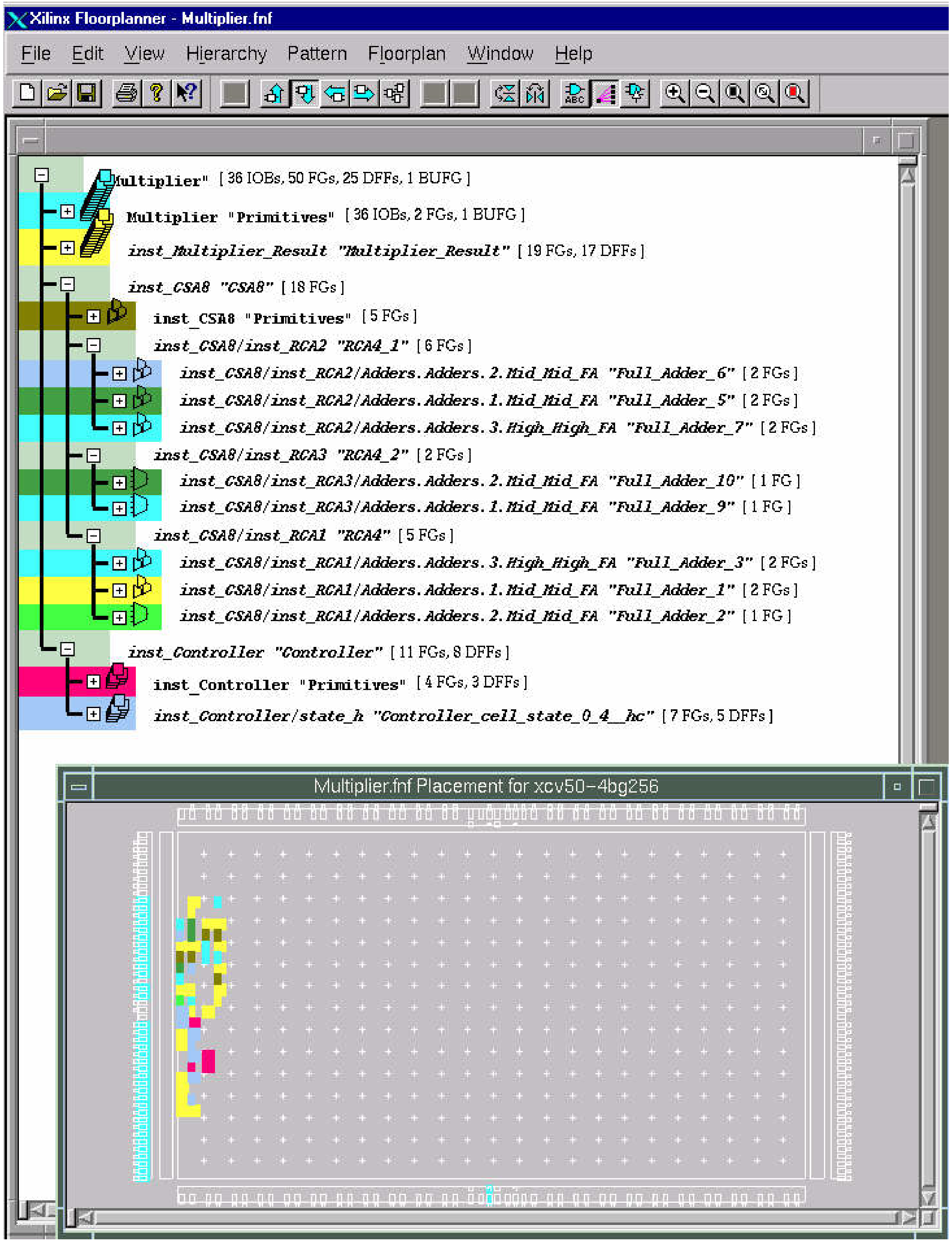
```
Multiplier" [ 36 IOBs, 45 FGs, 25 DFFs, 1 BUFG ]
  Multiplier "Primitives" [ 36 IOBs, 2 FGs, 1 BUFG ]
  inst_Multiplier_Result "Multiplier_Result" [ 20 FGs, 17 DFFs ]
  inst_RCA "RCA" [ 13 FGs ]
    inst_RCA/Adders.Adders.7.High_High_FA "Full_Adder_7" [ 2 FGs ]
    inst_RCA/Adders.Adders.1.Low_Low_FA "Full_Adder_1" [ 2 FGs ]
    inst_RCA/Adders.Adders.5.Low_Low_FA "Full_Adder_6" [ 2 FGs ]
    inst_RCA/Adders.Adders.3.Low_Low_FA "Full_Adder_5" [ 2 FGs ]
    inst_RCA/Adders.Adders.6.Low_Low_FA "Full_Adder_3" [ 2 FGs ]
    inst_RCA/Adders.Adders.4.Low_Low_FA "Full_Adder_2" [ 2 FGs ]
    inst_RCA/Adders.Adders.2.Low_Low_FA "Full_Adder" [ 1 FG ]
  inst_Controller "Controller" [ 10 FGs, 8 DFFs ]
    inst_Controller "Primitives" [ 4 FGs, 3 DFFs ]
    inst_Controller/state_h "Controller_cell_state_0_4__hc" [ 6 FGs, 5 DFFs ]
```

Mapped Design: 8-bit Multiplier with Ripple Carry Adder and behavioral D Flip-Flop


Xilinx Floorplanner - Multiplier.fnf
File Edit View Hierarchy Pattern Floorplan Window Help
Multiplier.fnf Placement for xcv50-4bg256

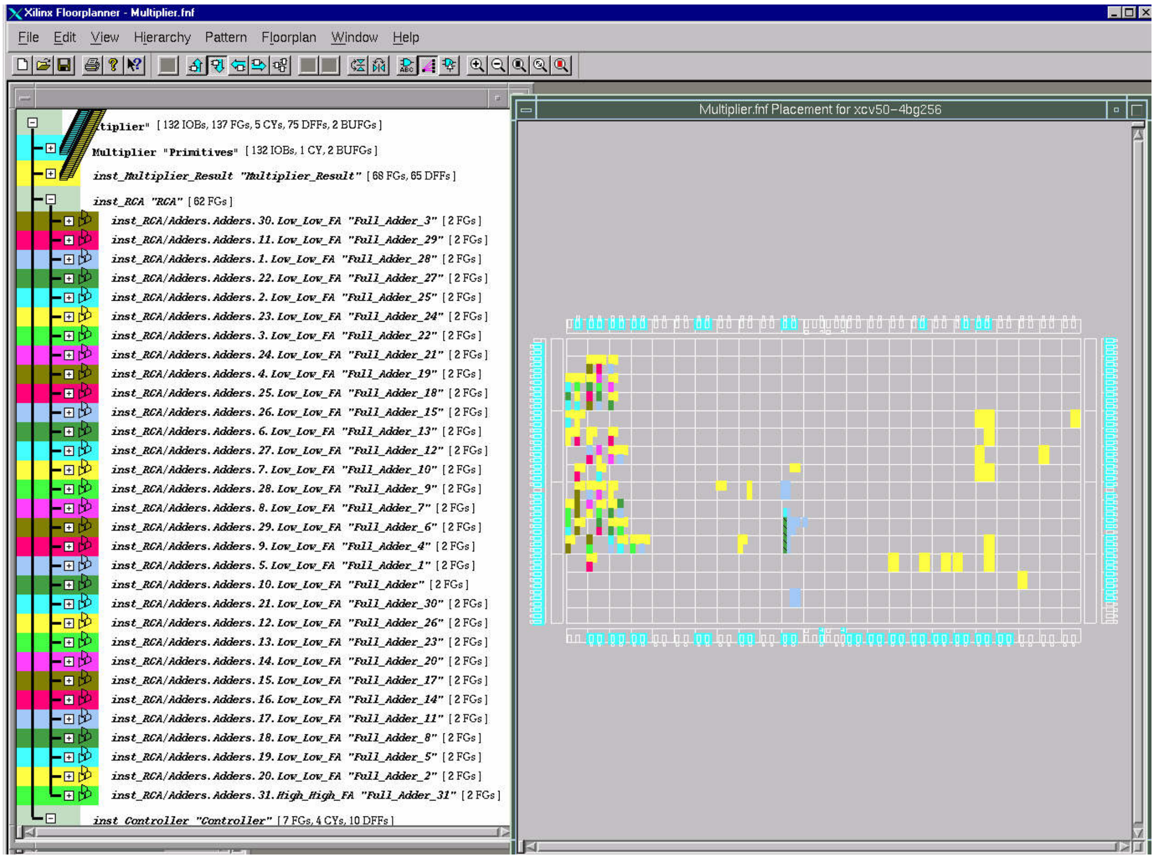
```
"Multiplier" [ 36 IOBs, 77 FGs, 25 DFFs, 1 BUFG ]
  Multiplier "Primitives" [ 36 IOBs, 2 FGs, 1 BUFG ]
  inst_Multiplier_Result "Multiplier_Result" [ 19 FGs, 17 DFFs ]
  inst_Multiplicand "Multiplicand" [ 32 FGs ]
    inst_Multiplicand/DFFs.DFFs.7_DFFReg "DFF" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.7_DFFReg/NAND5 "NAND_2_1" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.7_DFFReg/NAND4 "NAND_3_2" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.7_DFFReg/NAND2 "NAND_3" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.7_DFFReg/NAND3 "NAND_3_1" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.6_DFFReg "DFF_2" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.6_DFFReg/NAND5 "NAND_2_5" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.6_DFFReg/NAND4 "NAND_3_10" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.6_DFFReg/NAND2 "NAND_3_8" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.6_DFFReg/NAND3 "NAND_3_9" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.5_DFFReg "DFF_4" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.5_DFFReg/NAND5 "NAND_2_9" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.5_DFFReg/NAND4 "NAND_3_18" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.5_DFFReg/NAND2 "NAND_3_16" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.5_DFFReg/NAND3 "NAND_3_17" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.4_DFFReg "DFF_6" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.4_DFFReg/NAND5 "NAND_2_13" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.4_DFFReg/NAND4 "NAND_3_26" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.4_DFFReg/NAND2 "NAND_3_24" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.4_DFFReg/NAND3 "NAND_3_25" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.3_DFFReg "DFF_1" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.3_DFFReg/NAND5 "NAND_2_3" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.3_DFFReg/NAND4 "NAND_3_6" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.3_DFFReg/NAND2 "NAND_3_4" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.3_DFFReg/NAND3 "NAND_3_5" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.2_DFFReg "DFF_3" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.2_DFFReg/NAND5 "NAND_2_7" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.2_DFFReg/NAND4 "NAND_3_14" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.2_DFFReg/NAND2 "NAND_3_12" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.2_DFFReg/NAND3 "NAND_3_13" [ 1 FG ]
    inst_Multiplicand/DFFs.DFFs.1_DFFReg "DFF_5" [ 4 FGs ]
      inst_Multiplicand/DFFs.DFFs.1_DFFReg/NAND5 "NAND_2_11" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.1_DFFReg/NAND4 "NAND_3_22" [ 1 FG ]
      inst_Multiplicand/DFFs.DFFs.1_DFFReg/NAND2 "NAND_3_20" [ 1 FG ]
```

Mapped Design: 8-bit Multiplier with Ripple Carry Adder and structural D Flip-Flop


Xilinx Floorplanner - Multiplier.fnf
File
Edit
View
Hierarchy
Pattern
Floorplan
Window
Help
Multiplier.fnf Placement for xcv50-4bg256

```
"Multiplier" [ 36 IOBs, 50 FGs, 25 DFFs, 1 BUFG ]
  Multiplier "Primitives" [ 36 IOBs, 2 FGs, 1 BUFG ]
  inst_Multiplier_Result "Multiplier_Result" [ 19 FGs, 17 DFFs ]
  inst_CSA8 "CSA8" [ 18 FGs ]
    inst_CSA8 "Primitives" [ 5 FGs ]
    inst_CSA8/inst_RCA2 "RCA4_1" [ 6 FGs ]
      inst_CSA8/inst_RCA2/Adders.Adders.2.Mid_Mid_FA "Full_Adder_6" [ 2 FGs ]
      inst_CSA8/inst_RCA2/Adders.Adders.1.Mid_Mid_FA "Full_Adder_5" [ 2 FGs ]
      inst_CSA8/inst_RCA2/Adders.Adders.3.High_High_FA "Full_Adder_7" [ 2 FGs ]
    inst_CSA8/inst_RCA3 "RCA4_2" [ 2 FGs ]
      inst_CSA8/inst_RCA3/Adders.Adders.2.Mid_Mid_FA "Full_Adder_10" [ 1 FG ]
      inst_CSA8/inst_RCA3/Adders.Adders.1.Mid_Mid_FA "Full_Adder_9" [ 1 FG ]
    inst_CSA8/inst_RCA1 "RCA4" [ 5 FGs ]
      inst_CSA8/inst_RCA1/Adders.Adders.3.High_High_FA "Full_Adder_3" [ 2 FGs ]
      inst_CSA8/inst_RCA1/Adders.Adders.1.Mid_Mid_FA "Full_Adder_1" [ 2 FGs ]
      inst_CSA8/inst_RCA1/Adders.Adders.2.Mid_Mid_FA "Full_Adder_2" [ 1 FG ]
  inst_Controller "Controller" [ 11 FGs, 8 DFFs ]
    inst_Controller "Primitives" [ 4 FGs, 3 DFFs ]
    inst_Controller/state_h "Controller_cell_state_0_4_hc" [ 7 FGs, 5 DFFs ]
```

Mapped Design: 8-bit Multiplier with Carry Select Adder and behavioral D Flip-Flop


Xilinx Floorplanner - Multiplier.fnf
File
Edit
View
Hierarchy
Pattern
Floorplan
Window
Help
Multiplier.fnf Placement for xcv50-4bg256

```
tiplier" [ 132 IOBs, 137 FGs, 5 CYs, 75 DFFs, 2 BUFGs ]
Multiplier "Primitives" [ 132 IOBs, 1 CY, 2 BUFGs ]
inst_Multiplier_Result "Multiplier_Result" [ 68 FGs, 65 DFFs ]
inst_RCA "RCA" [ 62 FGs ]
  inst_RCA/Adders.Adders.30.Low_Low_FA "Full_Adder_3" [ 2 FGs ]
  inst_RCA/Adders.Adders.11.Low_Low_FA "Full_Adder_29" [ 2 FGs ]
  inst_RCA/Adders.Adders.1.Low_Low_FA "Full_Adder_28" [ 2 FGs ]
  inst_RCA/Adders.Adders.22.Low_Low_FA "Full_Adder_27" [ 2 FGs ]
  inst_RCA/Adders.Adders.2.Low_Low_FA "Full_Adder_25" [ 2 FGs ]
  inst_RCA/Adders.Adders.23.Low_Low_FA "Full_Adder_24" [ 2 FGs ]
  inst_RCA/Adders.Adders.3.Low_Low_FA "Full_Adder_22" [ 2 FGs ]
  inst_RCA/Adders.Adders.24.Low_Low_FA "Full_Adder_21" [ 2 FGs ]
  inst_RCA/Adders.Adders.4.Low_Low_FA "Full_Adder_19" [ 2 FGs ]
  inst_RCA/Adders.Adders.25.Low_Low_FA "Full_Adder_18" [ 2 FGs ]
  inst_RCA/Adders.Adders.26.Low_Low_FA "Full_Adder_15" [ 2 FGs ]
  inst_RCA/Adders.Adders.6.Low_Low_FA "Full_Adder_13" [ 2 FGs ]
  inst_RCA/Adders.Adders.27.Low_Low_FA "Full_Adder_12" [ 2 FGs ]
  inst_RCA/Adders.Adders.7.Low_Low_FA "Full_Adder_10" [ 2 FGs ]
  inst_RCA/Adders.Adders.28.Low_Low_FA "Full_Adder_9" [ 2 FGs ]
  inst_RCA/Adders.Adders.8.Low_Low_FA "Full_Adder_7" [ 2 FGs ]
  inst_RCA/Adders.Adders.29.Low_Low_FA "Full_Adder_6" [ 2 FGs ]
  inst_RCA/Adders.Adders.9.Low_Low_FA "Full_Adder_4" [ 2 FGs ]
  inst_RCA/Adders.Adders.5.Low_Low_FA "Full_Adder_1" [ 2 FGs ]
  inst_RCA/Adders.Adders.10.Low_Low_FA "Full_Adder" [ 2 FGs ]
  inst_RCA/Adders.Adders.21.Low_Low_FA "Full_Adder_30" [ 2 FGs ]
  inst_RCA/Adders.Adders.12.Low_Low_FA "Full_Adder_26" [ 2 FGs ]
  inst_RCA/Adders.Adders.13.Low_Low_FA "Full_Adder_23" [ 2 FGs ]
  inst_RCA/Adders.Adders.14.Low_Low_FA "Full_Adder_20" [ 2 FGs ]
  inst_RCA/Adders.Adders.15.Low_Low_FA "Full_Adder_17" [ 2 FGs ]
  inst_RCA/Adders.Adders.16.Low_Low_FA "Full_Adder_14" [ 2 FGs ]
  inst_RCA/Adders.Adders.17.Low_Low_FA "Full_Adder_11" [ 2 FGs ]
  inst_RCA/Adders.Adders.18.Low_Low_FA "Full_Adder_8" [ 2 FGs ]
  inst_RCA/Adders.Adders.19.Low_Low_FA "Full_Adder_5" [ 2 FGs ]
  inst_RCA/Adders.Adders.20.Low_Low_FA "Full_Adder_2" [ 2 FGs ]
  inst_RCA/Adders.Adders.31.High_High_FA "Full_Adder_31" [ 2 FGs ]
inst Controller "Controller" [ 7 FGs, 4 CYs, 10 DFFs ]
```

Mapped Design: 32-bit Multiplier with Ripple Carry Adder and behavioral D Flip-Flop