

APPENDIX A

VHDL DESCRIPTION AND SYNTHESIS OF MIPS R2000 DATAPATH BASIC BUILDING BLOCKS

This appendix presents a brief review of the Register Transfer Level (RTL) description of the basic building blocks for the datapath of the MIPS R2000 microprocessor followed by my own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of these datapath basic building blocks includes simulation and synthesis onto the target Xilinx Virtex-II FPGA chip. Again, this appendix is based on and complements the material presented in [47] and [48] and is annotated with my comments and tailored adaptation for the context of this research. This appendix is the basis on which Appendix B builds upon to create the VHDL description and synthesis of the finalized full MIPS R2000 microprocessor in chapter 6.

A.1 Introduction

This appendix presents the basic building blocks for the datapath functional units necessary for the hardware implementation of the MIPS instruction subset presented in chapter 5 and the finalized full MIPS R2000 microprocessor outlined in chapter 6.

The format for presentation of the material in this appendix is to take each functional unit (also referred to as *component*), and briefly review its RTL description as described in [47] and [48], then follow it with my own work implementing this component in VHDL, along with its synthesis and simulation. This process follows the design cycle described in chapter 4. Following is how the material is presented for each component:

➤ *RTL Description*

This is based on material from [47] and [48] and is annotated with my comments from my own work. This covers material like the behavioural/functional description of the component, relevant block and RTL diagrams, and design specifications. This is the first step in the design cycle for the component.

➤ *Design Entry and Synthesis*

This is my own work, and involves creating the component (called “Design” in this context) in Xilinx ISE software tools using either Schematic Editor (for synthesis from a schematic diagram) or HDL Editor (for synthesis from VHDL code). This is, then followed by running the Xilinx synthesis compiler XST (Xilinx Synthesis Tools) to infer (implement) the hardware components and layout onto the FPGA.

➤ *Synthesis Results*

After using the Xilinx ISE synthesis tools (XST) in the previous step (Design Synthesis) to generate the hardware implementation for the component, this step shows the resulting Top Level RTL symbol, Top Level Schematic Diagram, and subsequent lower levels (Gate Level Schematic Diagram) in the schematic hierarchy for the synthesized component.

➤ *FPGA Device Synthesis Summary*

This covers the most important FPGA Device Synthesis statistics from the Synthesis Report generated by XST after inferring the hardware in the Design Synthesis step. It shows the FPGA chip resources used in the hardware implementation of the component being designed.

These statistics are [72, 74, 71]:

Design Statistics:

IOs : number of total I/O ports

Macro Statistics:

RAM : number of total block RAMs
 # 256x32-bit dual-port block RAM : number of 256x32-bit block RAMs
 # 32x32-bit dual-port block RAM : number of 32x32-bit block RAMs
 # Tristates : number of total tristate buffers
 # 32-bit tristate buffer : number of 32-bit tristate buffers
 # 5-bit tristate buffer : number of 5-bit tristate buffers
 # 8-bit tristate buffer : number of 8-bit tristate buffers

Cell Usage:

This section reports the count of all the primitives used in the design. These primitives are classified in 8 groups [72]:

BELS : number of total logical cells that are basic elements of Virtex technology
 # and2 : number of 2-input AND gates
 # and2b1 : number of 2-input AND gates with inverted second input
 # and3 : number of 3-input AND gates
 # and3b1 : number of 3-input AND gates with inverted third input
 # gnd : number of ground connection signal tags
 # inv : number of single inverters
 # LUT1 : number of 1-bit Look-UP Tables with general output
 # LUT2_L : number of 2-bit Look-UP Tables with local output
 # LUT3 : number of 3-bit Look-Up Tables with general output
 # LUT3_L : number of 3-bit Look-Up Tables with local output
 # LUT4 : number of 4-bit Look-Up Tables with general output
 # LUT4_L : number of 4-bit Look-Up Tables with local output
 # muxcy : 2-to-1 multiplexer for carry logic with general output
 # muxcy_1 : 2-to-1 multiplexer for carry logic with local output
 # or2 : number of 2-input OR gates
 # vcc : number of Vcc connections
 # xor2 : number of 2-input XOR gates
 # xor3 : number of 3-input XOR gates
 # RAMS : number of total RAMs

```

#          RAMB16_S36_S36      :      number of 16-bit RAMs
# Tri-States                    :      number of total tristate primitives
#          BUFT                 :      number of tristate buffers with active low
                                   enable
#          BUFE                 :      number of tristate buffers with active high
                                   enable
# Clock Buffers                 :      number of total clock buffers
#          BUFGP                :      number of primary global buffers fro driving
                                   clocks or longlines
# IO Buffers                   :      number of total standard I/O buffers, except
                                   the clock buffer
#          IBUF                 :      number of single input buffers
#          OBUF                 :      number of single output buffers
# Logical                      :      number of total logical cells primitives that
                                   are not basic elements
#          nor4                 :      number of 4-input NOR gates
# Others                       :      This group contains all the cells that have
                                   not been classified in the previous groups.
#          fmap                 :      number of function generator partitioning
                                   control symbols

```

Device utilization summary:

Selected Device: 2v8000ff1517-4 (This is the 8-million-gate Virtex-2 FPGA device used)

```

Number of Slices:              How many used out of a total of 46592
Number of 4 input LUTs:       How many used out of a total of 93184
Number of bonded IOBs:        How many used out of a total of 1108
Number of TBUFs:              How many used out of a total of 23296
Number of BRAMs:              How many used out of a total of 168
Number of GCLKs:              How many used out of a total of 16

```

➤ **Place-and-Route onto the FPGA**

After synthesizing the component (the design), the synthesized hardware is placed using ISE FloorPlanner and then routed onto the target FPGA chip using ISE FPGA Editor.

➤ **Simulation Results**

Mentor Graphics ModelSim is used to simulate the behaviour of the synthesized hardware. The resulting waveforms generated by ModelSim are used to verify that the synthesized hardware functions according to the specified behaviour and functionality of the component (the design). This step concludes the design cycle for the component.

This appendix starts with a review of digital logic conventions and clocking methodology in section A.2, thereby setting the scene for presentation of the material to follow, which is covered in section A.3. Section

A.3 elaborates on the building blocks (basic components) needed for the MIPS datapath. Section A.4 concludes this appendix with a summary.

A.2 A Review of Logic Conventions and Clocking Methodology

This section briefly reviews a few key ideas in digital logic that will be used extensively in this chapter. They are as follows:

- ❑ For the sake of consistency, the word *asserted* will be used to indicate a signal that is logically high (i.e. logic value = 1) and *assert* to indicate that a signal should be driven logically high [47]. Conversely, the word *de-asserted* will be used to indicate a signal that is logically low (i.e. logic value = 0) and *de-assert* to specify that a signal should be driven logically low (= 0).
- ❑ The components, also known as “the functional units in the MIPS implementation consist of two different types of logic elements” [47, p.340]:
 - *Combinational Elements*: These operate on data values [47]. Examples include decoders, multiplexors, adders, and ALUs (Arithmetic Logic Units). These are all discussed in more detail in section A.3.
 - *State (Sequential) Elements*: These contain state as they have some internal storage [47]. Examples include the instruction and data memories as well as the registers in the register file. These are all discussed in more detail in section A.3.
- ❑ All the simulations of the synthesized hardware covered in this appendix are run at a rate at which the input changes at 10 ns time intervals for combinational elements and a clock cycle time of 10 ns for sequential (state elements).
- ❑ The MIPS implementation is a synchronous digital system, which uses an *edge-triggered* clocking methodology for its state (sequential) elements [47]. The design convention implemented in this research uses a rising-edge clocking methodology.
- ❑ Nearly all of these state and logic elements are 32 bits wide [47].

In the next section, the datapath building blocks for the MIPS hardware implementation are covered.

A.3 The Datapath Building Blocks (Basic Components)

This section presents the results of my research on the VHDL implementation, synthesis and simulation of the basic building blocks (basic components) needed for the datapath sections and complete datapath (covered in Appendix B) and the finalized MIPS R2000 microprocessor hardware implementation covered in chapter 6.

A.3.1 Decoders

➤ *RTL Description*

The decoder is a combinational logic component. The decoder is described in detail on page B-8 of [47].

➤ *Design Entry and Synthesis*

Below is the VHDL code for synthesizing a 5-bit Decoder (5-to-32) with a reset signal, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

-- 5-bit Decoder (5-to-32) with reset:

entity Decoder_5to32_vhd is
    Port (
        Reset : in  STD_LOGIC;
        D_in  : in  STD_LOGIC_VECTOR(4 downto 0);
        D_out : out STD_LOGIC_VECTOR(31 downto 0)
    );
end Decoder_5to32_vhd;

architecture Behavioral of Decoder_5to32_vhd is

begin
```

```

process(Reset,D_in)
begin
    if ( Reset = '1') then
        D_out <= "00000000000000000000000000000000";
    else
        case D_in is
            when "00000" => D_out <= "00000000000000000000000000000001";
            when "00001" => D_out <= "00000000000000000000000000000010";
            when "00010" => D_out <= "000000000000000000000000000000100";
            when "00011" => D_out <= "0000000000000000000000000000001000";
            when "00100" => D_out <= "00000000000000000000000000000010000";
            when "00101" => D_out <= "000000000000000000000000000000100000";
            when "00110" => D_out <= "0000000000000000000000000000001000000";
            when "00111" => D_out <= "00000000000000000000000000000010000000";
            when "01000" => D_out <= "000000000000000000000000000000100000000";
            when "01001" => D_out <= "0000000000000000000000000000001000000000";
            when "01010" => D_out <= "00000000000000000000000000000010000000000";
            when "01011" => D_out <= "000000000000000000000000000000100000000000";
            when "01100" => D_out <= "0000000000000000000000000000001000000000000";
            when "01101" => D_out <= "00000000000000000000000000000010000000000000";
            when "01110" => D_out <= "000000000000000000000000000000100000000000000";
            when "01111" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10000" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10001" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10010" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10011" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10100" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10101" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10110" => D_out <= "0000000000000000000000000000001000000000000000";
            when "10111" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11000" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11001" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11010" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11011" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11100" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11101" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11110" => D_out <= "0000000000000000000000000000001000000000000000";
            when "11111" => D_out <= "0000000000000000000000000000001000000000000000";
            when others => NULL;
        end case;
    end if;
end process;

end Behavioral;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 5-to-32 decoder, was generated. Figure A.2 shows the resulting top level RTL symbol for the synthesized 5-to-32 decoder. Figures A.3 to A.5 show the resulting top level schematic diagram, while figure A.6 shows the resulting gate level schematic diagram.

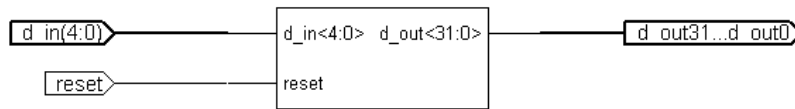


Figure A.2 Resulting top level RTL symbol for the synthesized 5-to-32 decoder.

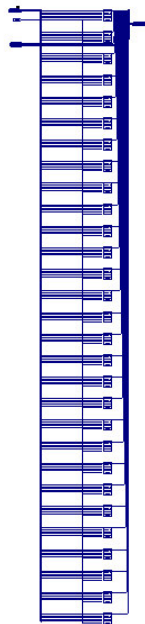


Figure A.3 Resulting top level RTL schematic diagram (Part 1 of 2) for the synthesized 5-to-32 decoder of figure A.2, and showing only the upper 29 output bits of the total 32, due to space limitations.

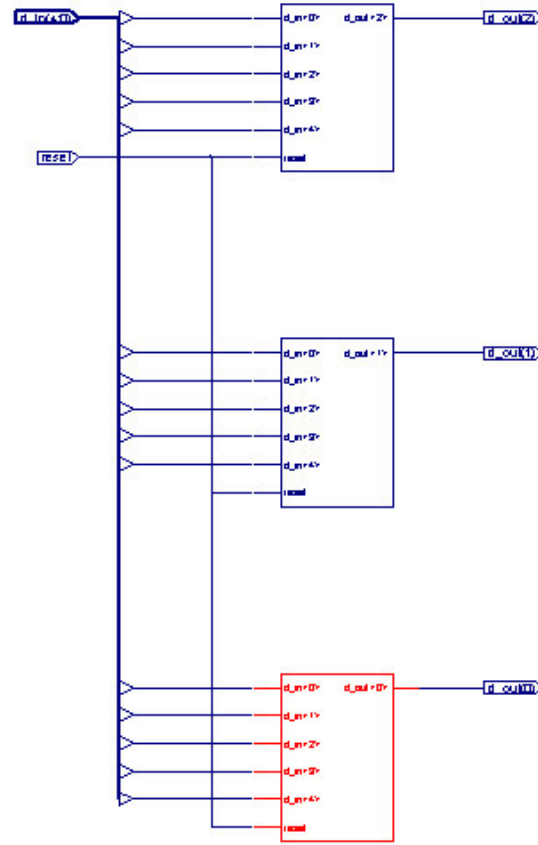


Figure A.4 Resulting top level RTL schematic diagram (Part 2 of 2) for the synthesized 5-to-32 decoder of figure A.2, and showing only the lower 3 output bits of the total 32, due to space limitations.

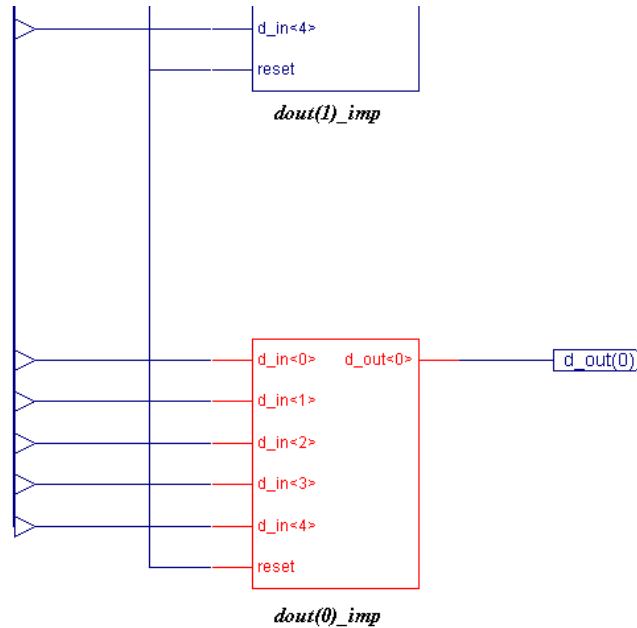


Figure A.5 Magnification of the bottom section of figure A.4 (highlighted in red).

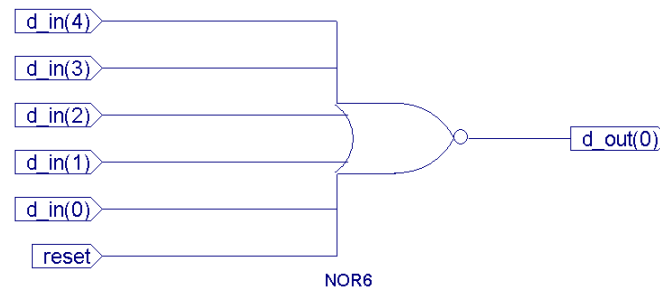


Figure A.6 Resulting gate level schematic diagram for the bottom section of figure A.5 (highlighted in red).

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the above 5-to-32 decoder using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 38

Cell Usage:

BELS : 40

LUT3 : 32

LUT4 : 8

IO Buffers : 38

IBUF : 6

OBUF : 32

Device utilization summary:

Number of Slices: 23 out of 46592 0%

Number of 4 input LUTs: 40 out of 93184 0%

Number of bonded IOBs: 38 out of 1108 3%

➤ *Place-and-Route onto the FPGA*

In figure A.7, FPGA Editor shows the synthesized 5-to-32 decoder after place-and-route onto the target Virtex-II FPGA chip, where the lighter-color (blue) clutter in the middle left represents the actual FPGA chip resourced used (placed-and-routed) while the lighter-colored lines running across the chip represent the actual interconnection between these resources and the input/output pins.

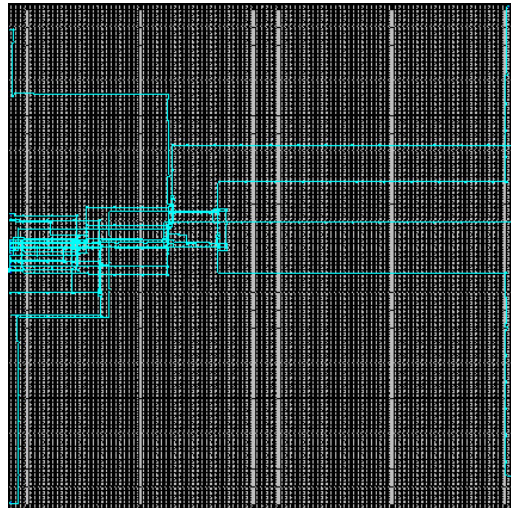


Figure A.7 FPGA Editor showing the synthesized 5-to-32 decoder after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure A.8 shows the waveform results of simulating the 5-to-32 decoder VHDL behavioural model in Mentor Graphics ModelSim. These waveforms are in decimal format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the decoder. This concludes the design cycle for this component.

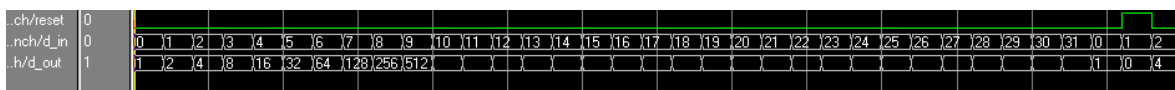


Figure A.8 Results of simulating the synthesized decoder using ModelSim.

A.3.2 Multiplexors (Implemented with Logic Gates)

➤ RTL Description

One basic logic function that is used quite often in the MIPS hardware implementation is the *multiplexer*. A multiplexer is a combinational logic component. The multiplexer is described in detail on page B-9 of [47].

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing a 1-bit 2-to-1 multiplexer from the Xilinx ISE library using Schematic Editor:

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1e.sch
```

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1.sch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY M2_1_MXILINX IS
    PORT ( D0      :      IN      STD_LOGIC;
           D1      :      IN      STD_LOGIC;
           S0      :      IN      STD_LOGIC;
           O        :      OUT     STD_LOGIC);

end M2_1_MXILINX;

ARCHITECTURE SCHEMATIC OF M2_1_MXILINX IS
    SIGNAL M0      :      STD_LOGIC;
    SIGNAL M1      :      STD_LOGIC;

    ATTRIBUTE BOX_TYPE : STRING;

    COMPONENT AND2
        PORT ( I0      :      IN      STD_LOGIC;
               I1      :      IN      STD_LOGIC;
               O        :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF AND2 : COMPONENT IS "BLACK_BOX";
    COMPONENT AND2B1
        PORT ( I0      :      IN      STD_LOGIC;
               I1      :      IN      STD_LOGIC;
               O        :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF AND2B1 : COMPONENT IS "BLACK_BOX";
    COMPONENT OR2
        PORT ( I0      :      IN      STD_LOGIC;
               I1      :      IN      STD_LOGIC;
               O        :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
BEGIN

```

```

I_36_9 : AND2
    PORT MAP (I0=>D1, I1=>S0, O=>M1);

I_36_7 : AND2B1
    PORT MAP (I0=>S0, I1=>D0, O=>M0);

I_36_8 : OR2
    PORT MAP (I0=>M1, I1=>M0, O=>O);

END SCHEMATIC;

```

The Xilinx ISE library schematic symbol for a 1-bit 2-to-1 multiplexer is shown in figure A.10.

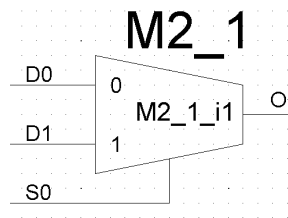


Figure A.10 RTL Schematic symbol for a 1-bit 2-to-1 multiplexer in Xilinx ISE library.

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 1-bit 2-to-1 multiplexer, was generated. Figure A.11 shows the resulting top level RTL symbol for the synthesized multiplexer while figure A.12 shows the resulting top level schematic diagram which is also the gate level schematic.

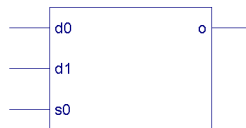


Figure A.11 Resulting top level RTL symbol for the synthesized 1-bit 2-to-1 multiplexer.

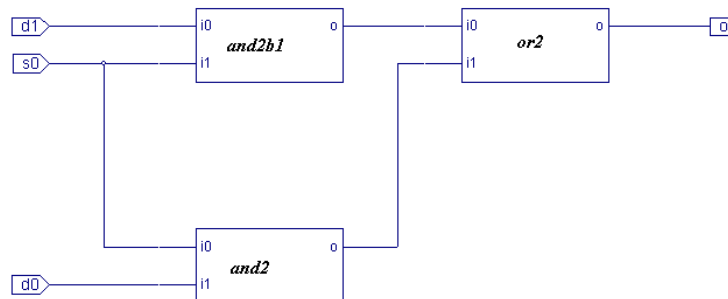


Figure A.12 Resulting top level (is also gate level) schematic diagram for the synthesized 1-bit 2-to-1 multiplexer of figure A.11.

It is worth noting that, as seen in figure A.12, the gate level schematic diagram shows the individual logic gates as rectangular symbols rather than the usual logic symbol shapes (as in figure A.6 for the decoder). This is due to the fact that in Xilinx ISE tools, if the design entry is VHDL code written in HDL Editor, then the resulting synthesized gate level schematic would show the logic gates in their proper symbols. On the other hand, if the design entry is a schematic diagram drawn in Schematic Editor, then the resulting synthesized gate level schematic would show the logic gates all in rectangular shapes! Whether this is an intentional feature or a software glitch in Xilinx ISE tools remains a mystery!

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the above 1-bit 2-to-1 multiplexer using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

Ios : 4

Cell Usage:

BELS : 3

and2 : 1

and2b1 : 1

or2 : 1

IO Buffers : 4

IBUF : 3

OBUF : 1

Device utilization summary:

Number of bonded IOBs: 4 out of 1108 0%

➤ *Place-and-Route onto the FPGA*

In figure A.13, FPGA Editor shows the synthesized 1-bit 2-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the small blue interconnections at the lower left corner in the figure.

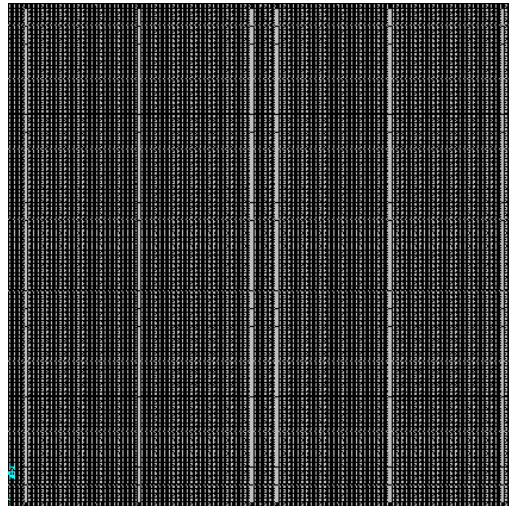


Figure A.13 FPGA Editor showing the synthesized 1-bit 2-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure A.14 shows the waveform results of simulating the 1-bit 2-to-1 multiplexer VHDL behavioural model in Mentor Graphics ModelSim. All these waveform are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the multiplexer. This concludes the design cycle for this component.

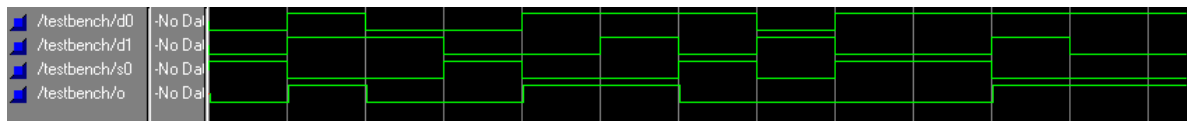


Figure A.14 Results of simulating the synthesized 1-bit 2-to-1 multiplexer using ModelSim.

A.3.3 Multiplexors (Implemented with Tri-state Buffers)

➤ RTL Description

It is practical to build a multiplexer from logic gates only when the number of inputs is not large. For example, in the case of implementing the *register file* (to be covered in later in section A.3.10), a 32-to-1 multiplexer is required, and is implemented with logic gates. However, large memories (memories are covered in section A.3.11) cannot be built in the same way a register file is built because, unlike a register file where a 32-to-1 multiplexer might be practical, a 64K-to-1 multiplexer that would be needed for a 64K x 1 RAM is totally impractical [47].

“Rather than use a giant multiplexer, large memories are implemented with a shared output line, called a *bit line*, which multiple memory cells in the memory array can assert. To allow multiple sources to drive a single line, a *three-state buffer* (or *tri-state buffer*) is used” [47, p.B-28]. Multiplexers implemented with tri-state buffers are described in detail on pages B-28 and B-29 of [47].

Adapting the above arguments to my research; wherever possible I have implemented multiplexers with tri-state buffers (even for a small number of inputs) more than with logic gates mainly due to the fact that the Virtex-II FPGA chip has a very large pool of available tri-state buffers (23296 to be exact!) which makes sense to use them while sparing the relatively more scarce logic gates; the latter chip resources being used for synthesis of other more complex hardware functions.

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing a 32-bit 32-to-1 Multiplexer design with Tri-State Buffers and a built-in 5-to-32 Decoder, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

-- Saturday 24th May 2003

-- 32bit 32-to-1 Multiplexor design with Tri-State Buffers and a built-in 5-to-32
Decoder:

entity Multiplexor_Tri_State_32bit_32to1_vhd is
    Port (
        Sel      : in  STD_LOGIC_VECTOR(4  downto 0);
        D_in0    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in1    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in2    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in3    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in4    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in5    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in6    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in7    : in  STD_LOGIC_VECTOR(31 downto 0);
        D_in8    : in  STD_LOGIC_VECTOR(31 downto 0);
```



```

D_in9 : in STD_LOGIC_VECTOR(31 downto 0);
D_in10 : in STD_LOGIC_VECTOR(31 downto 0);
D_in11 : in STD_LOGIC_VECTOR(31 downto 0);
D_in12 : in STD_LOGIC_VECTOR(31 downto 0);
D_in13 : in STD_LOGIC_VECTOR(31 downto 0);
D_in14 : in STD_LOGIC_VECTOR(31 downto 0);
D_in15 : in STD_LOGIC_VECTOR(31 downto 0);
D_in16 : in STD_LOGIC_VECTOR(31 downto 0);
D_in17 : in STD_LOGIC_VECTOR(31 downto 0);
D_in18 : in STD_LOGIC_VECTOR(31 downto 0);
D_in19 : in STD_LOGIC_VECTOR(31 downto 0);
D_in20 : in STD_LOGIC_VECTOR(31 downto 0);
D_in21 : in STD_LOGIC_VECTOR(31 downto 0);
D_in22 : in STD_LOGIC_VECTOR(31 downto 0);
D_in23 : in STD_LOGIC_VECTOR(31 downto 0);
D_in24 : in STD_LOGIC_VECTOR(31 downto 0);
D_in25 : in STD_LOGIC_VECTOR(31 downto 0);
D_in26 : in STD_LOGIC_VECTOR(31 downto 0);
D_in27 : in STD_LOGIC_VECTOR(31 downto 0);
D_in28 : in STD_LOGIC_VECTOR(31 downto 0);
D_in29 : in STD_LOGIC_VECTOR(31 downto 0);
D_in30 : in STD_LOGIC_VECTOR(31 downto 0);
D_in31 : in STD_LOGIC_VECTOR(31 downto 0);

D_out : out STD_LOGIC_VECTOR(31 downto 0)

);

end Multiplexor_Tri_State_32bit_32to1_vhd;

architecture Behavioral of Multiplexor_Tri_State_32bit_32to1_vhd is

begin

    D_out <= D_in0 when (Sel="00000") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in1 when (Sel="00001") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in2 when (Sel="00010") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in3 when (Sel="00011") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in4 when (Sel="00100") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in5 when (Sel="00101") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in6 when (Sel="00110") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in7 when (Sel="00111") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in8 when (Sel="01000") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in9 when (Sel="01001") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in10 when (Sel="01010") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in11 when (Sel="01011") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in12 when (Sel="01100") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in13 when (Sel="01101") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    D_out <= D_in14 when (Sel="01110") else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

```

This multiplexer is what is used in implementing a 32-bit register file with 32 registers.

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 32-bit 32-to-1 Multiplexer, was generated. Figure A.17 shows the resulting top level RTL symbol for the synthesized multiplexer.

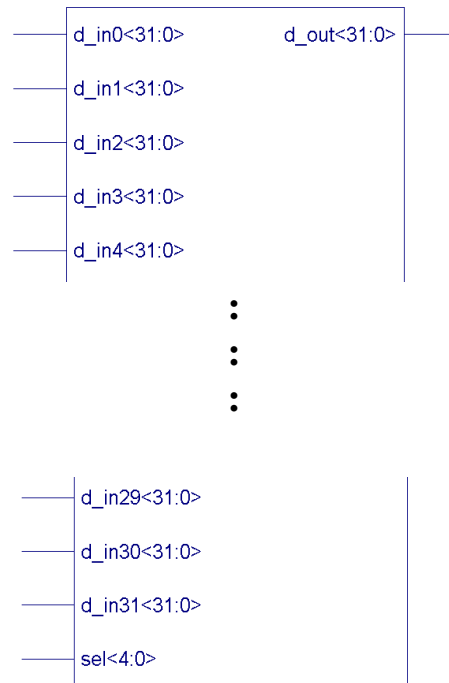


Figure A.17 Resulting top level RTL symbol for the synthesized 32-bit 32-to-1 multiplexer implemented with tri-state buffers.

Figures A.18 to A.20 show more schematic diagrams for the synthesis results of the design hierarchy.

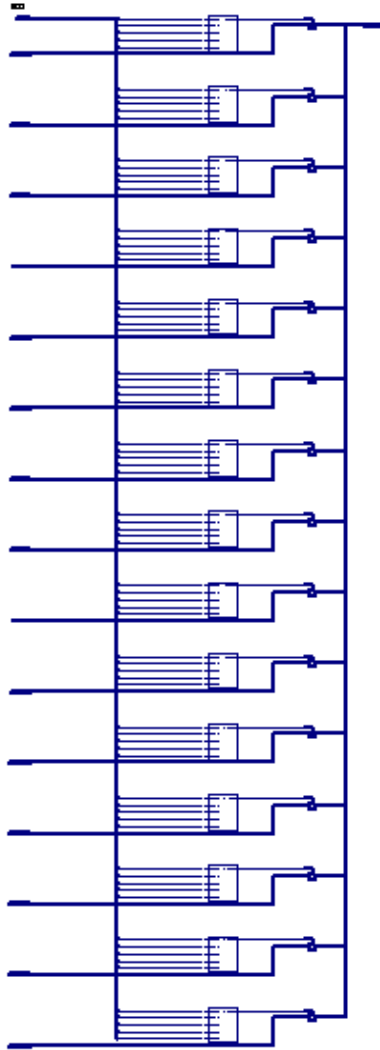


Figure A.18 Resulting top level RTL schematic diagram for the synthesized multiplexer of figure A.17, showing only the first 16 input signals of the total 33, due to space limitations.

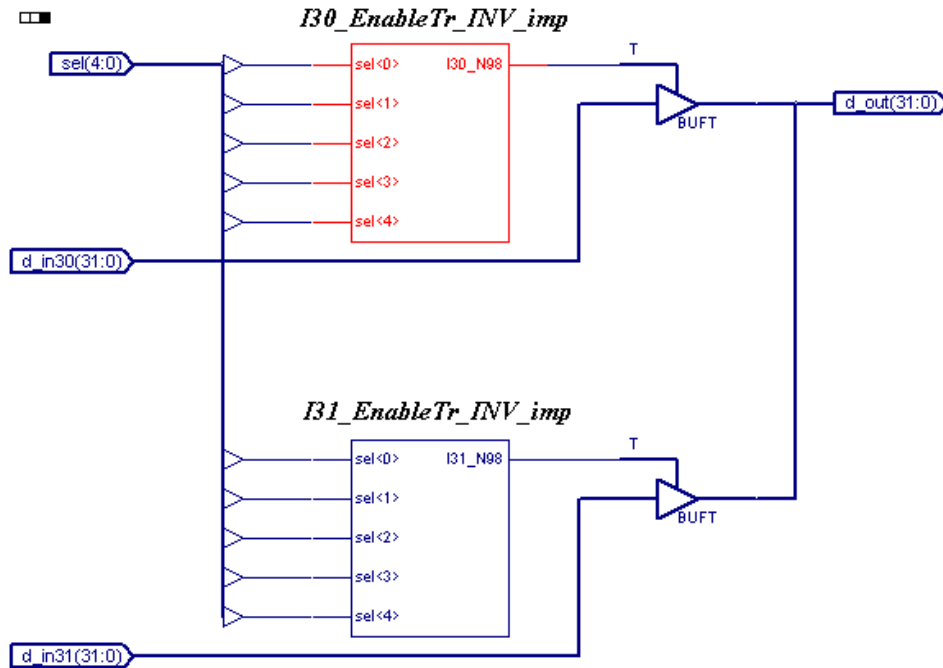


Figure A.19 Resulting top level RTL schematic diagram for the synthesized multiplexer of figure A.17 and extension of figure A.18, showing only the last 2 input signals of the total 33, due to space limitations.

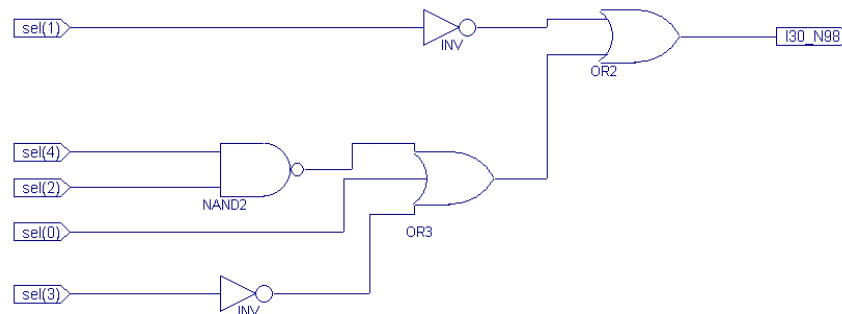


Figure A.20 Resulting gate level schematic diagram for the synthesized top block (highlighted in red) of figure A.19.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 32-bit 32-to-1 multiplexer using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```

Design Statistics:
# I/Os                      : 1061

Macro Statistics:
# Tristates                  : 32
# 32-bit tristate buffer : 32

Cell Usage:
# BELS                      : 40
# LUT3                      : 40
# Tri-States                : 1024
# BUFT                      : 1024
# IO Buffers                : 1061
# IBUF                      : 1029
# OBUF                      : 32

Device utilization summary:

Number of Slices:           23      out of 46592      0%
Number of 4 input LUTs:     40      out of 93184      0%
Number of bonded IOBs:      1061    out of 1108       95%
Number of TBUFs:            1024    out of 23296      4%

```

➤ *Place-and-Route onto the FPGA*

In figure A.21, FPGA Editor shows the synthesized 32-bit 32-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the FPGA chip.

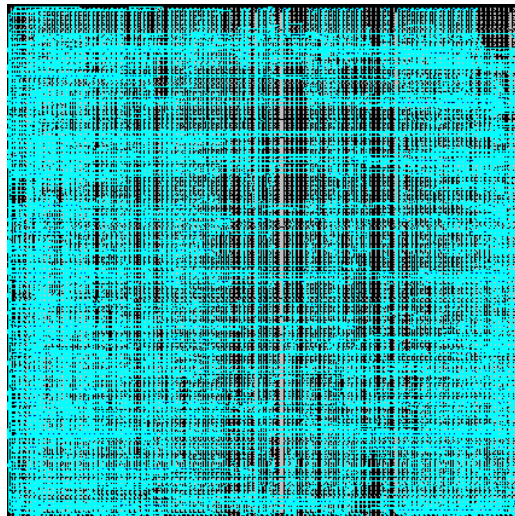


Figure A.21 *FPGA Editor showing the synthesized 32-bit 32-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip.*

➤ Simulation Results

Figure A.22 shows the waveform results of simulating the 32-bit 32-to-1 multiplexer VHDL behavioural model in Mentor Graphics ModelSim. All these waveforms are in decimal format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the multiplexer. This concludes the design cycle for this component.

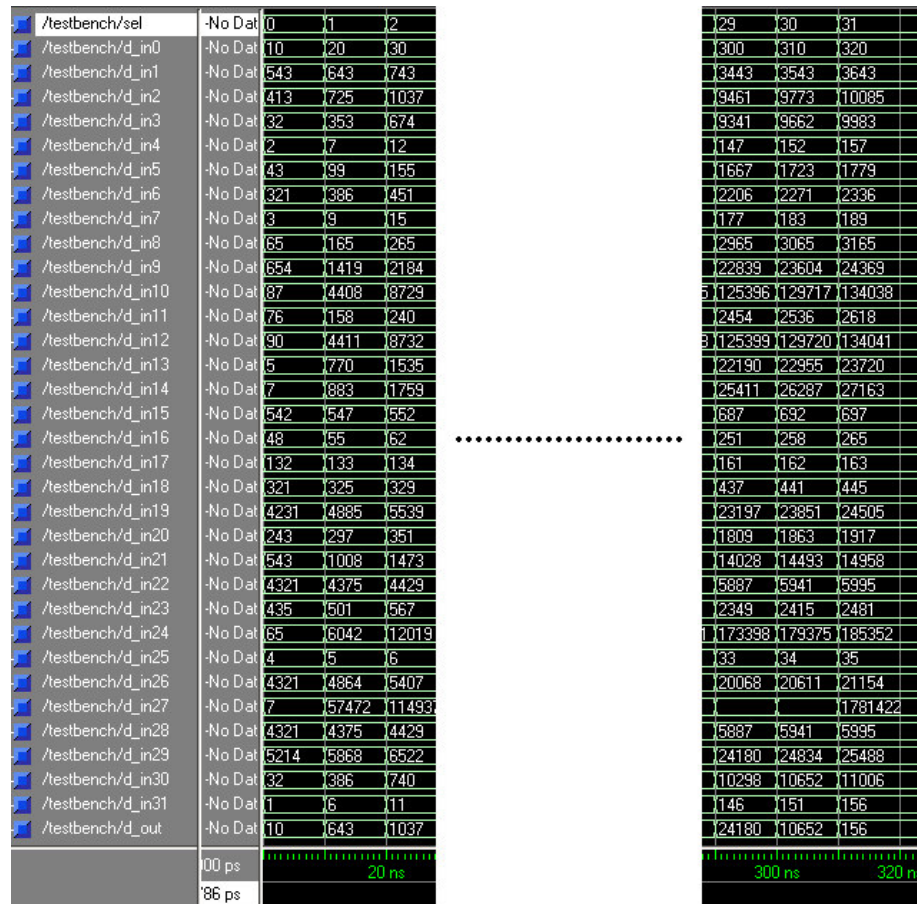


Figure A.22 Results of simulating the synthesized 32-bit 32-to-1 multiplexer using ModelSim. Note that all the signals are shown in unsigned decimal format rather than binary for reasons of clarity.

A.3.4 Arithmetic Logic Unit (ALU)

The *Arithmetic Logic Unit (ALU)* is a combinational logic unit and “is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR” [47, p.230].

In this subsection, an ALU is constructed from four hardware building blocks (AND and OR gates, inverters, and multiplexers) [47]. The complete details for designing an ALU, starting off with a basic 1-bit ALU and ending at a more complex 32-bit ALU using *Carry Look Ahead*, are covered on pages 230 to 249 of [47].

The Big Picture First: Basic 32-Bit ALU

➤ RTL Description

A basic 32-bit ALU is created by connecting adjacent 1-bit ALUs (as black boxes) [48]. This results in the creation of a 32-bit ALU using *Ripple Carry (RC)*. This is covered in more detail on pages B-29 and B-30 of [48].

I chose to cover in detail the ripple carry 32-bit ALU first because of its relative simplicity (compared to the carry lookahead 32-bit ALU to be discussed at the end) thereby enabling a clear transition in presenting the material and research results leading up to the final more complex carry lookahead 32-bit ALU. This also happens to be the flow in describing the ripple carry ALU first leading up to the carry lookahead in both [47] and [48].

Starting Point: Basic 1-Bit ALU

➤ RTL Description

The starting point is a basic 1-bit ALU. The detailed procedure for designing this basic 1-bit ALU is elaborated in both [47] and [48]. However, figure A.24 below shows that this ALU is constructed from an AND gate, an OR gate, an inverter, two multiplexers and a *Full Adder (FA)*.

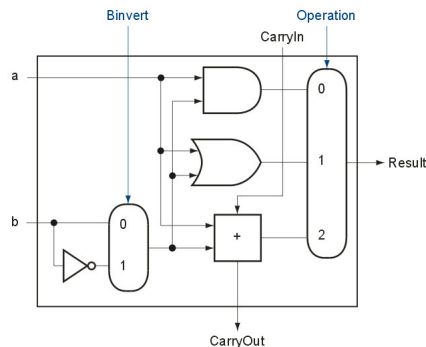


Figure A.24 Basic 1-bit ALU that performs AND, OR, addition, and subtraction [48, p.B-31].

The table in figure A.25 shows the operations (functions) that this ALU performs on the 1-bit input signals a , b , and $CarryIn$.

Control Inputs:		Function	Datapath Output:	
<i>Binvert</i>	<i>Operation</i>		<i>Result</i>	
0	00	AND (Logical)	a	AND b
1	00	AND (Logical)	a	AND b'
0	01	OR (Logical)	a	OR b
1	01	OR (Logical)	a	OR b'
0	10	ADD (Arithmetic)	a	$+$ b $+$ $CarryIn$
1	10	SUB (Arithmetic)	a	$+$ b' $+$ $CarryIn$

Figure A.25 Function table for the basic 1-bit ALU of figure A.24.

Referring to the function table in figure A.25, it is worth noting the following:

- When Function = ADD:
 - For LSB 1-bit ALU only, we assign:

$$CarryIn = 0$$
 - For each of the upper 31 1-bit ALUs (including MSB):

$$CarryIn = CarryOut \text{ from the FA of the previous 1-bit ALU}$$
- When Function = SUB (2's complement subtraction), the final 32-bit result is:

$$\begin{aligned}
 Result_{(32-bit)} &= A_{(32-bit)} + B'_{(32-bit)} + CarryIn_{(1-bit)} \\
 &= A_{(32-bit)} + B'_{(32-bit)} + 1 \\
 &= A_{(32-bit)} + (B'_{(32-bit)} + 1) \\
 &= A_{(32-bit)} + (-B_{(32-bit)}) \\
 &= A_{(32-bit)} - B_{(32-bit)} \\
 &= A_{(32-bit)} + 2's \text{ complement of } B_{(32-bit)}
 \end{aligned}$$

This is achieved by performing the following:

- For LSB 1-bit ALU only, we assign:

$$CarryIn = Binvert = 1$$
- For each of the upper 31 1-bit ALUs (including MSB):

$$CarryIn = CarryOut \text{ from the FA of the previous 1-bit ALU}$$

As seen now, this is the basic ALU and is, therefore, the starting point from which the final fully functional 32-bit ALU is designed.

Customizing the Basic 32-Bit ALU for MIPS

➤ RTL Description

“These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most MIPS instructions can be performed by this ALU. But the design of the MIPS ALU is yet incomplete. One instruction, which still needs support is the set on less than instruction (SLT)” [48, p.B-32]. The detailed procedure for adding this added functionality to the basic 32-bit ALU is described on page B-32 of [48]. The resulting 32-bit ALU supporting slt instruction tailored to MIPS is shown in figure A.26 below.

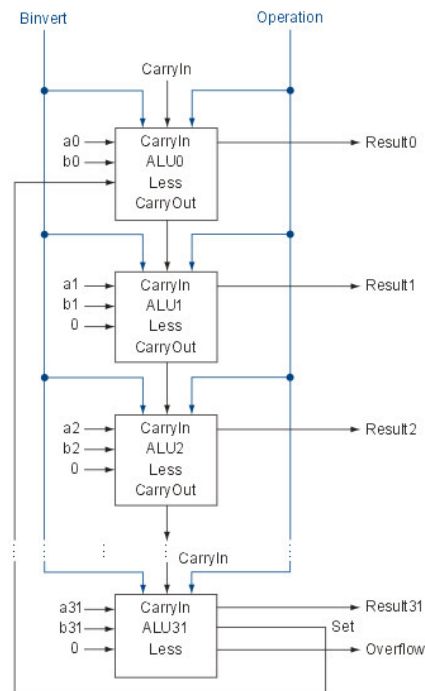


Figure A.26 A 32-bit ALU tailored for MIPS is constructed from 31 copies of the same 1-bit ALU and one special 1-bit ALU for the MSB [47, p.239].

However, the added functionality for the SLT instruction can be summarized as follows:

- ❑ SLT will always set all bits of the subtraction result ($A_{(32-bit)} - B_{(32-bit)}$) except the LSB to 0, with the LSB set according to the following comparison:
 - ❑ $LSB = 1$ if $A_{(32-bit)} < B_{(32-bit)}$ (i.e. $A_{(32-bit)} - B_{(32-bit)}$ is negative) $\Rightarrow 1$ means -ve
 - ❑ $LSB = 0$ otherwise (i.e. $A_{(32-bit)} - B_{(32-bit)}$ is positive) $\Rightarrow 0$ means +ve

□ Therefore, the input *Less* to each of the 1-bit ALUs will be set as follows:

□ For upper 31 bits (bits 1 to 31): $Less = 0$

□ For LSB (bit 0): $Less = Set$ output signal from MSB ALU (bit 31)

The following two subsections show the modifications made to the basic 1-bit ALU to support this added functionality of the SLT instruction.

Final 1-Bit MIPS ALU

➤ RTL Description

The detailed procedure for finalizing the design for the final 1-bit MIPS ALU is described on pages 236 to 238 of [47]. Figure A.27 below shows the finalized 1-bit ALU tailored to MIPS and supporting the SLT instruction by having a fourth input signal *Less* connected to input port number 3 of the output multiplexer [47].

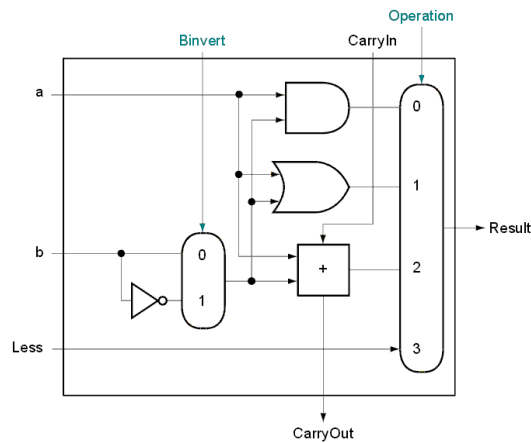


Figure A.27 Finalized 1-bit ALU tailored to MIPS and supporting SLT instruction [47, p.238].

The added slt functionality of this finalized 1-bit ALU for MIPS is shown in the function table in figure A.28.

Control Inputs:		Function		Datapath Output:	
Binvert	Operation			Result	
0	00	AND	(Logical)	a	AND b
1	00	AND	(Logical)	a	AND b'
0	01	OR	(Logical)	a	OR b
1	01	OR	(Logical)	a	OR b'
0	10	ADD	(Arithmetic)	$a + b$	+ $CarryIn$
1	10	SUB	(Arithmetic)	$a + b'$	+ $CarryIn$
X	11	SLT	(Arithmetic)	Less	

Figure A.28 Function table for the finalized 1-bit MIPS ALU of figure A.27.

In the table above, note that when Function = SLT, the value of the control input signal *Binvert* has no effect on the datapath output signal *Result*. This is due to the fact that this table reflects only the functionality of this 1-bit ALU as a stand-alone unit. However, the upcoming subsection on **Final 32-Bit MIPS ALU using Ripple Carry** will show the table for the finalized and full functionality.

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the finalized 1-bit ALU of figure A.27. The final schematic diagram is shown in figure A.29. Note that in figure A.29, the *Enable* input signal to the output multiplexer M4_1E is always asserted (i.e. is set to 1). This convention will be maintained throughout this work for this ALU and all subsequent ALU designs.

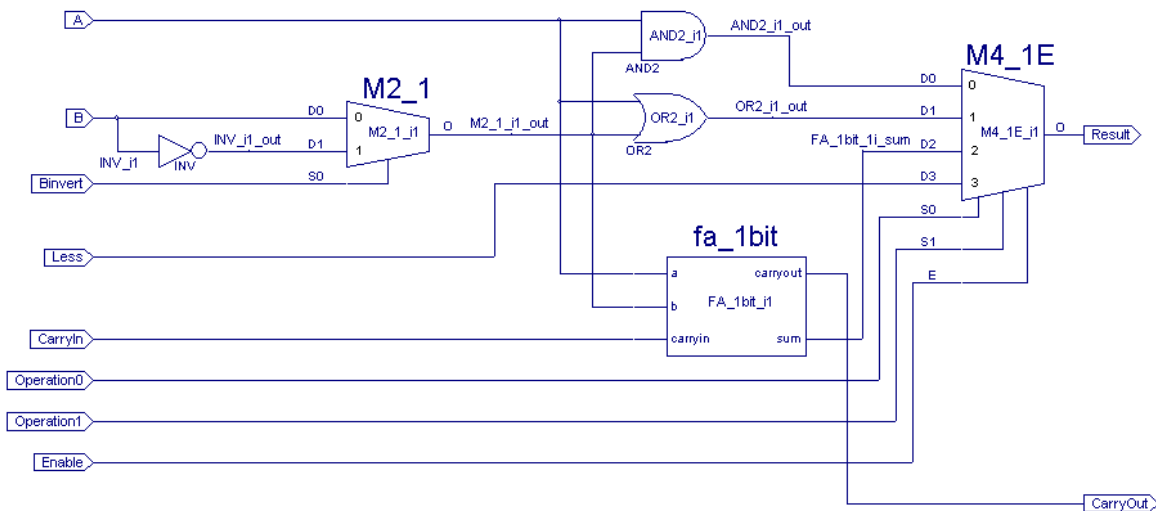


Figure A.29 Schematic diagram design entry for the finalized 1-bit MIPS ALU in Schematic Editor.

After synthesis of the schematic diagram in figure A.29 using XST, the resulting VHDL code shown below was generated.

```
-- Vhdl model created from schematic alu_1bit_sch.sch - Tue Apr 18 07:45:24 2006
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY alu_1bit_sch IS
    PORT ( A          :      IN      STD_LOGIC;
           B          :      IN      STD_LOGIC;
           Binvert    :      IN      STD_LOGIC;
           CarryIn    :      IN      STD_LOGIC;
           Enable     :      IN      STD_LOGIC;
           Less       :      IN      STD_LOGIC;
           Operation0 :      IN      STD_LOGIC;
           Operation1 :      IN      STD_LOGIC;
           CarryOut   :      OUT     STD_LOGIC;
           Result     :      OUT     STD_LOGIC);

end alu_1bit_sch;

ARCHITECTURE SCHEMATIC OF alu_1bit_sch IS
    SIGNAL AND2_i1_out :      STD_LOGIC;
    SIGNAL FA_1bit_1i_sum :      STD_LOGIC;
    SIGNAL INV_i1_out :      STD_LOGIC;
    SIGNAL M2_1_i1_out :      STD_LOGIC;
    SIGNAL OR2_i1_out :      STD_LOGIC;

    ATTRIBUTE BOX_TYPE : STRING;
    ATTRIBUTE U_SET :      STRING ;
    ATTRIBUTE U_SET OF M2_1_i1 :      LABEL IS "M2_1_i1_3";
    ATTRIBUTE U_SET OF M4_1E_i1 :      LABEL IS "M4_1E_i1_2";

    COMPONENT AND2
        PORT ( I0      :      IN      STD_LOGIC;
               I1      :      IN      STD_LOGIC;
               O        :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF AND2 : COMPONENT IS "BLACK_BOX";

    COMPONENT fa_1bit
        PORT ( a          :      IN      STD_LOGIC;
               b          :      IN      STD_LOGIC;
               carryin    :      IN      STD_LOGIC;
               carryout   :      OUT     STD_LOGIC;
               sum        :      OUT     STD_LOGIC);
```

```

END COMPONENT;

COMPONENT INV
  PORT ( I      :      IN      STD_LOGIC;
        O      :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF INV : COMPONENT IS "BLACK_BOX";
COMPONENT M2_1_MXILINX_alu_1bit_sch
  PORT ( D0      :      IN      STD_LOGIC;
        D1      :      IN      STD_LOGIC;
        S0      :      IN      STD_LOGIC;
        O      :      OUT     STD_LOGIC);
END COMPONENT;

COMPONENT M4_1E_MXILINX_alu_1bit_sch
  PORT ( D0      :      IN      STD_LOGIC;
        D1      :      IN      STD_LOGIC;
        D2      :      IN      STD_LOGIC;
        D3      :      IN      STD_LOGIC;
        E      :      IN      STD_LOGIC;
        S0      :      IN      STD_LOGIC;
        S1      :      IN      STD_LOGIC;
        O      :      OUT     STD_LOGIC);
END COMPONENT;

COMPONENT OR2
  PORT ( I0      :      IN      STD_LOGIC;
        I1      :      IN      STD_LOGIC;
        O      :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
BEGIN

AND2_i1 : AND2
  PORT MAP (I0=>M2_1_i1_out, I1=>A, O=>AND2_i1_out);

FA_1bit_i1 : fa_1bit
  PORT MAP (a=>A, b=>M2_1_i1_out, carryin=>CarryIn, carryout=>CarryOut,
            sum=>FA_1bit_i1_sum);

INV_i1 : INV
  PORT MAP (I=>B, O=>INV_i1_out);

M2_1_i1 : M2_1_MXILINX_alu_1bit_sch
  PORT MAP (D0=>B, D1=>INV_i1_out, S0=>Binvert, O=>M2_1_i1_out);

M4_1E_i1 : M4_1E_MXILINX_alu_1bit_sch

```

```

PORT MAP (D0=>AND2_i1_out, D1=>OR2_i1_out, D2=>FA_1bit_i1_sum, D3=>Less,
          E=>Enable, S0=>Operation0, S1=>Operation1, O=>Result);

OR2_i1 : OR2
PORT MAP (I0=>M2_1_i1_out, I1=>A, O=>OR2_i1_out);

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 1-bit ALU, was generated. Figure A.30 shows the resulting top level RTL symbol for the synthesized ALU while figure A.31 shows the resulting top level RTL schematic diagram. Figure A.32 shows the resulting gate level schematic diagram for the synthesized 1-bit full adder of figure A.31 (highlighted in red).

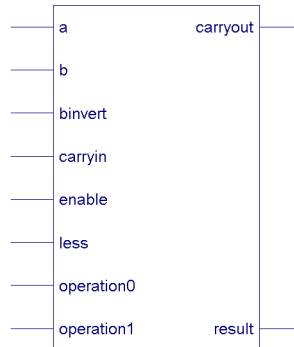


Figure A.30 Resulting top level RTL symbol for the synthesized final 1-bit MIPS ALU.

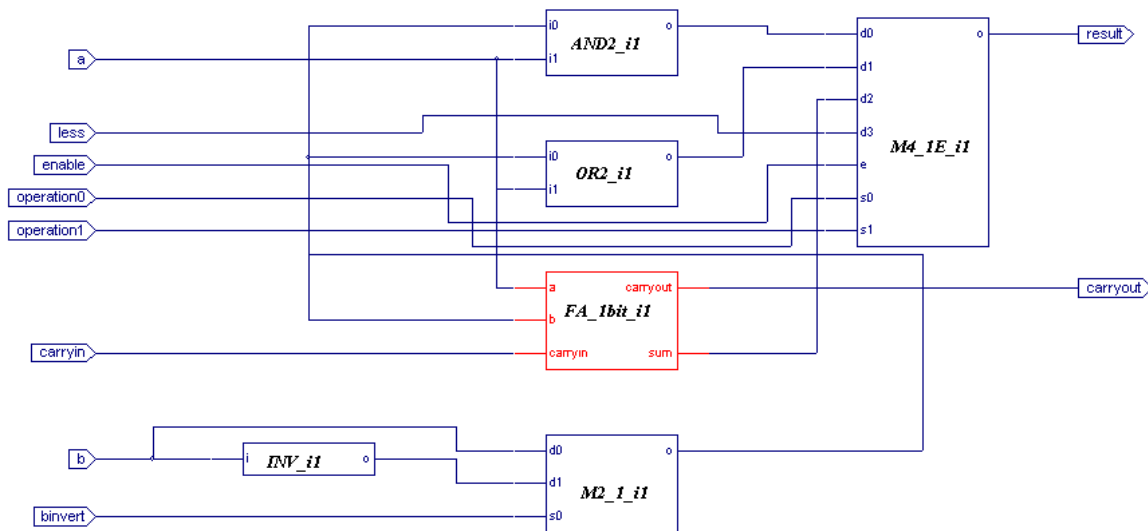


Figure A.31 Resulting top level RTL schematic diagram for the synthesized final 1-bit MIPS ALU.

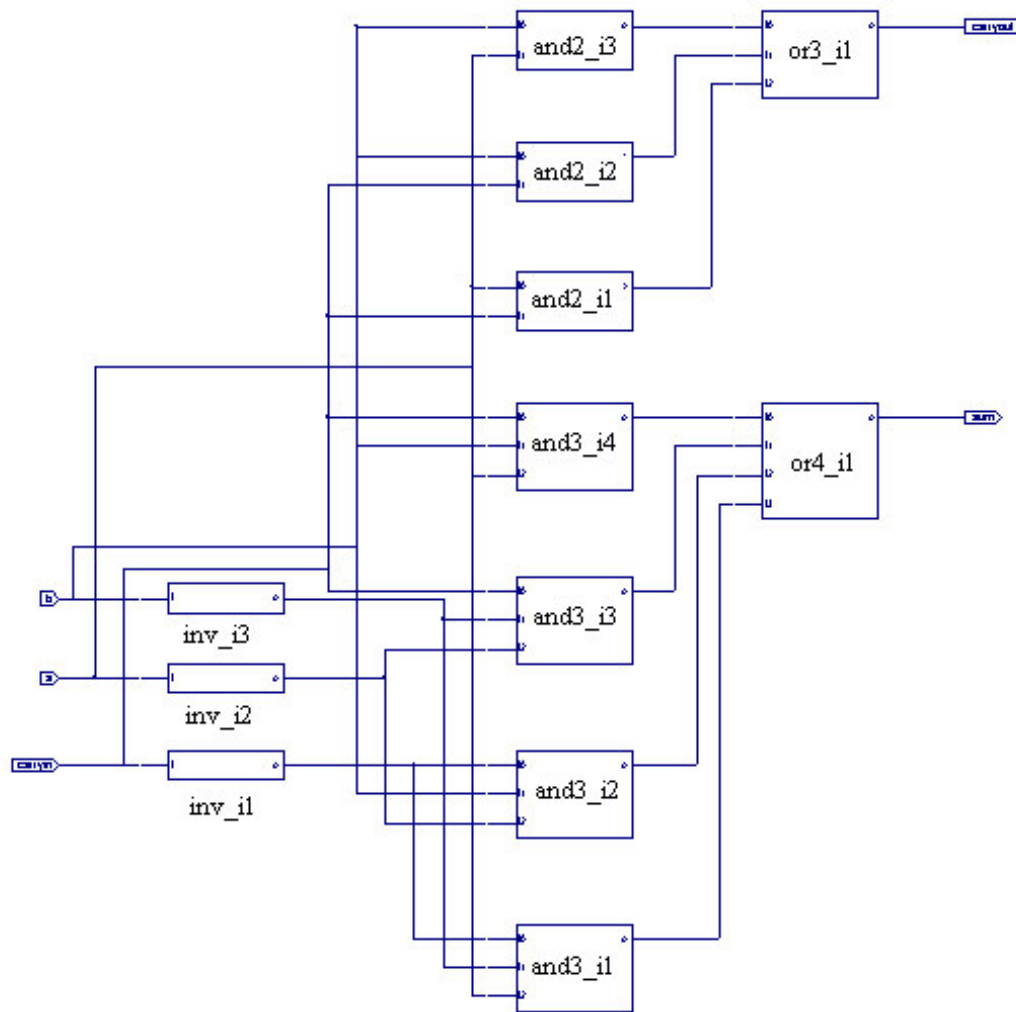


Figure A.32 Resulting gate level schematic diagram for the synthesized 1-bit full adder in figure A.31 (highlighted in red).

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 1-bit ALU using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 10

Cell Usage:

BELS : 27

and2 : 5

and2b1 : 1


```

#          and3                : 6
#          and3b1              : 2
#          inv                 : 4
#          LUT1                : 2
#          muxf5               : 1
#          or2                 : 4
#          or3                 : 1
#          or4                 : 1
# IO Buffers                   : 10
#          IBUF                : 8
#          OBUF                : 2

```

Device utilization summary:

```

Number of Slices:                1      out of 46592      0%
Number of 4 input LUTs:         2      out of 93184      0%
Number of bonded IOBs:         10      out of 1108       0%

```

➤ Place-and-Route onto the FPGA

In figure A.33, FPGA Editor shows the synthesized 1-bit ALU after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the FPGA chip and more concentrated at the lower right corner.

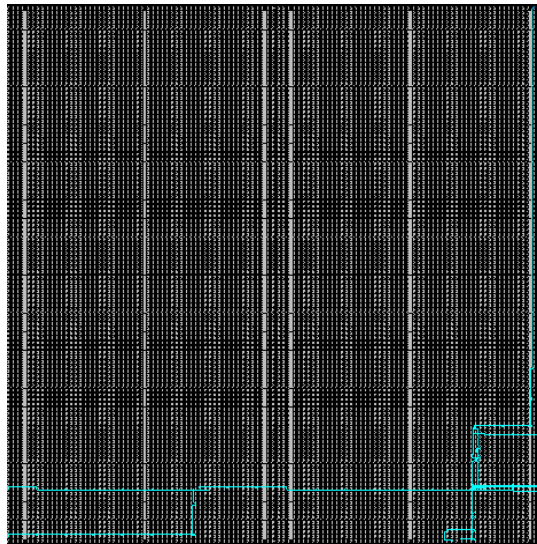


Figure A.33 FPGA Editor showing the synthesized final 1-bit MIPS ALU after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure A.34 shows the waveform results of simulating the 1-bit ALU VHDL behavioural model in Mentor Graphics ModelSim. All the waveforms are in binary format. When checking these waveforms, the following points are worth noting:

- ❑ These waveforms are compared against the specified behavior of the ALU as shown earlier in the function table of figure A.28.
- ❑ When the ALU Function is either AND, OR, or SLT, the only output signal of interest is *result*.
- ❑ Only when the ALU Function is either ADD or SUB, then both output signals *carryout* and *result* are of interest.

It is clear from figure A.34 that the resulting synthesized hardware functions correctly and according to specification. This concludes the design cycle for this component.

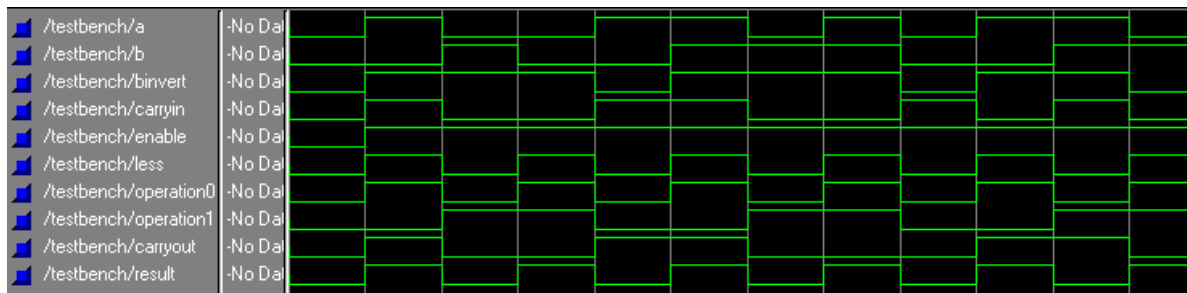


Figure A.34 Results of simulating the synthesized 1-bit ALU using ModelSim.

Final 1-Bit MIPS ALU for Most Significant Bit (MSB)

➤ RTL Description

Referring back to figure A.26, it is evident that the last 1-bit ALU (ALU31) is a special ALU designed specifically to handle the MSB. The detailed procedure for finalizing the design for this special MSB ALU for MIPS is described on pages 237 and 238 of [47]. Figure A.35 below shows the finalized 1-bit ALU tailored to MIPS and supporting the SLT instruction by having a fourth input signal *Less* connected to input port number 3 of the output multiplexer. Also, this ALU has an *Overflow Detection Unit (ODU)*, which is shown in more detail in figure A.36.

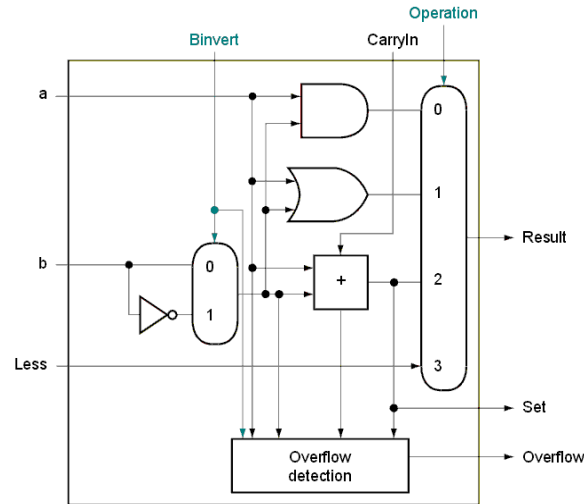


Figure A.35 Finalized special 1-bit MSB ALU tailored to MIPS and supporting SLT and overflow [47, p.238]

The function table for this MSB ALU is the same as that shown in figure A.28 in the previous subsection on the **Final 1-Bit MIPS ALU**.

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the finalized special 1-bit MSB ALU of figure A.35. The final schematic diagram is shown in figure A.36. Note that there is additional circuitry for generating the *Overflow* and *Set* signals to ensure that the *Set* signal now factors in the overflow condition [47, 48]. Also, the output signal *SetLessThan* in figure A.36 is actually the output signal *Set* in figure A.35.

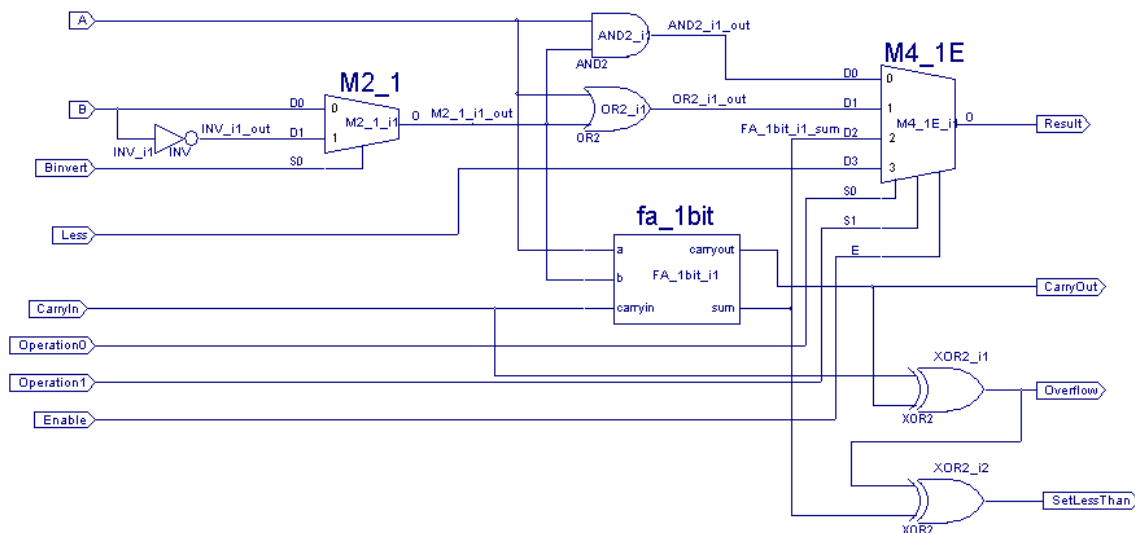


Figure A.36 Schematic diagram design entry for the finalized special 1-bit MIPS ALU for MSB in Schematic Editor.

After synthesis of the schematic diagram in figure A.36 using XST, the resulting VHDL code shown below was generated.

```
-- Vhdl model created from schematic alu_msb_sch.sch - Wed Apr 26 18:11:24 2006

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY alu_msb_sch IS
    PORT ( A          :      IN      STD_LOGIC;
           B          :      IN      STD_LOGIC;
           Binvert     :      IN      STD_LOGIC;
           CarryIn     :      IN      STD_LOGIC;
           Enable      :      IN      STD_LOGIC;
           Less        :      IN      STD_LOGIC;
           Operation0   :      IN      STD_LOGIC;
           Operation1   :      IN      STD_LOGIC;
           CarryOut     :      OUT     STD_LOGIC;
           Overflow     :      OUT     STD_LOGIC;
           Result       :      OUT     STD_LOGIC;
           SetLessThan  :      OUT     STD_LOGIC);

end alu_msb_sch;

ARCHITECTURE SCHEMATIC OF alu_msb_sch IS

    SIGNAL AND2_i1_out      :      STD_LOGIC;
    SIGNAL CarryOut_DUMMY   :      STD_LOGIC;
    SIGNAL FA_1bit_i1_sum   :      STD_LOGIC;
    SIGNAL INV_i1_out       :      STD_LOGIC;
    SIGNAL M2_1_i1_out      :      STD_LOGIC;
    SIGNAL OR2_i1_out       :      STD_LOGIC;
    SIGNAL Overflow_DUMMY   :      STD_LOGIC;

    ATTRIBUTE BOX_TYPE      :      STRING;
    ATTRIBUTE U_SET         :      STRING ;
    ATTRIBUTE U_SET OF M2_1_i1 : LABEL IS "M2_1_i1_3";
    ATTRIBUTE U_SET OF M4_1E_i1 : LABEL IS "M4_1E_i1_2";

    COMPONENT AND2
        PORT ( I0      :      IN      STD_LOGIC;
              I1      :      IN      STD_LOGIC;
              O        :      OUT     STD_LOGIC);
    END COMPONENT;
```

```

ATTRIBUTE BOX_TYPE OF AND2 : COMPONENT IS "BLACK_BOX";
COMPONENT fa_1bit
    PORT ( a      :      IN      STD_LOGIC;
           b      :      IN      STD_LOGIC;
           carryin :      IN      STD_LOGIC;
           carryout :     OUT     STD_LOGIC;
           sum     :     OUT     STD_LOGIC);
END COMPONENT;

COMPONENT INV
    PORT ( I      :      IN      STD_LOGIC;
           O      :     OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF INV : COMPONENT IS "BLACK_BOX";
COMPONENT M2_1_MXILINX_alu_msb_sch
    PORT ( D0      :      IN      STD_LOGIC;
           D1      :      IN      STD_LOGIC;
           S0      :      IN      STD_LOGIC;
           O        :     OUT     STD_LOGIC);
END COMPONENT;

COMPONENT M4_1E_MXILINX_alu_msb_sch
    PORT ( D0      :      IN      STD_LOGIC;
           D1      :      IN      STD_LOGIC;
           D2      :      IN      STD_LOGIC;
           D3      :      IN      STD_LOGIC;
           E        :      IN      STD_LOGIC;
           S0      :      IN      STD_LOGIC;
           S1      :      IN      STD_LOGIC;
           O        :     OUT     STD_LOGIC);
END COMPONENT;

COMPONENT OR2
    PORT ( I0      :      IN      STD_LOGIC;
           I1      :      IN      STD_LOGIC;
           O        :     OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
COMPONENT XOR2
    PORT ( I0      :      IN      STD_LOGIC;
           I1      :      IN      STD_LOGIC;
           O        :     OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XOR2 : COMPONENT IS "BLACK_BOX";
BEGIN
    CarryOut <= CarryOut_DUMMY;

```

```

Overflow <= Overflow_DUMMY;

AND2_i1 : AND2
  PORT MAP (I0=>M2_1_i1_out, I1=>A, O=>AND2_i1_out);

FA_1bit_i1 : fa_1bit
  PORT MAP (a=>A, b=>M2_1_i1_out, carryin=>CarryIn,
    carryout=>CarryOut_DUMMY, sum=>FA_1bit_i1_sum);

INV_i1 : INV
  PORT MAP (I=>B, O=>INV_i1_out);

M2_1_i1 : M2_1_MXILINX_alu_msb_sch
  PORT MAP (D0=>B, D1=>INV_i1_out, S0=>Binvert, O=>M2_1_i1_out);

M4_1E_i1 : M4_1E_MXILINX_alu_msb_sch
  PORT MAP (D0=>AND2_i1_out, D1=>OR2_i1_out, D2=>FA_1bit_i1_sum, D3=>Less,
    E=>Enable, S0=>Operation0, S1=>Operation1, O=>Result);

OR2_i1 : OR2
  PORT MAP (I0=>M2_1_i1_out, I1=>A, O=>OR2_i1_out);

XOR2_i2 : XOR2
  PORT MAP (I0=>FA_1bit_i1_sum, I1=>Overflow_DUMMY, O=>SetLessThan);

XOR2_i1 : XOR2
  PORT MAP (I0=>CarryOut_DUMMY, I1=>CarryIn, O=>Overflow_DUMMY);

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 1-bit ALU, was generated. Figure A.37 shows the resulting top level RTL symbol for the synthesized ALU while figure A.38 shows the resulting top level RTL schematic diagram.

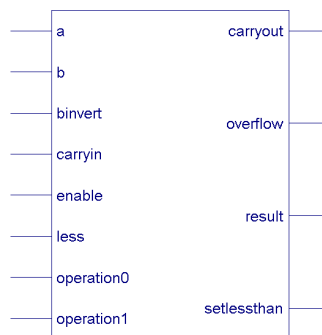


Figure A.37 Resulting top level RTL symbol for the finalized special 1-bit MIPS ALU for MSB.

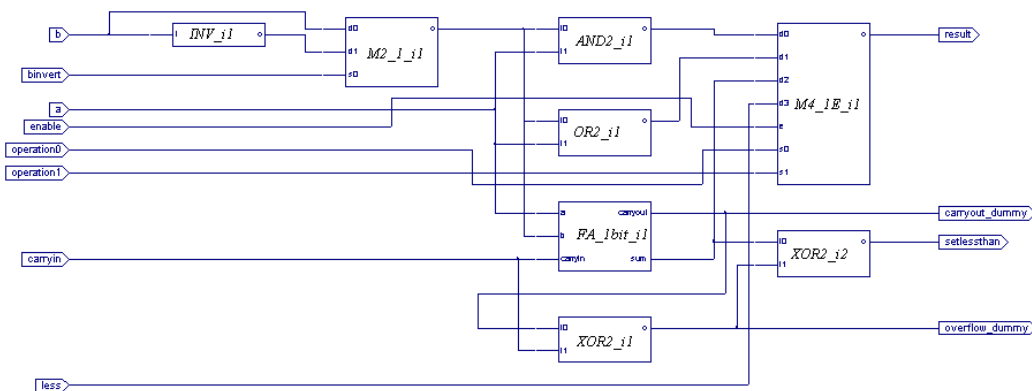


Figure A.38 Resulting top level RTL schematic diagram for the finalized special 1-bit MIPS ALU for MSB.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 1-bit ALU using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 12

Cell Usage:

```
# BELS : 29
# and2 : 5
# and2b1 : 1
# and3 : 6
# and3b1 : 2
# inv : 4
# LUT1 : 2
# muxf5 : 1
# or2 : 4
# or3 : 1
# or4 : 1
# xor2 : 2
# IO Buffers : 12
# IBUF : 8
# OBUF : 4
```

Device utilization summary:

Number of Slices:	1 out of 46592	0%
Number of 4 input LUTs:	2 out of 93184	0%
Number of bonded IOBs:	12 out of 1108	1%

➤ *Place-and-Route onto the FPGA*

In figure A.39, FPGA Editor shows the synthesized 1-bit ALU after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the FPGA chip and more concentrated at the lower right corner.

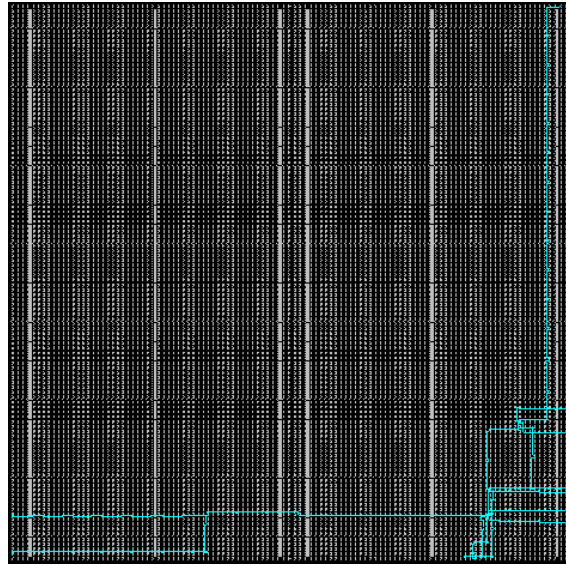


Figure A.39 *FPGA Editor showing the synthesized final 1-bit MIPS ALU for MSB after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figure A.40 shows the waveform results of simulating the 1-bit ALU VHDL behavioural model in Mentor Graphics ModelSim. All the waveforms are in binary format. When checking these waveforms, the following points are worth noting:

- ❑ These waveforms are compared against the specified behavior of the ALU as shown earlier in the function table of figure A.28.
- ❑ When the ALU Function is either AND, OR, or SLT, the only output signal of interest is *result*.
- ❑ Only when the ALU Function is either ADD or SUB, then both output signals *carryout* and *result* are of interest.

- ❑ The output signal *setlessthan* is not meaningful at this stage but will be when the final 32-bit ALU is put together in the next subsection, and will therefore be checked only then.
- ❑ Although the output signal *overflow* is implemented, it will not be utilized in any aspect within the context of this research, but rather for future research.
- ❑ The waveforms for all four output signals (*carryout*, *overflow*, *result* and *setlessthan*) display narrow spikes caused by hazards which settle down to the correct expected values directly afterwards. Hazards occur when two inputs change at the same time, which drives the output temporarily to intermediate unwanted values. They become problematic when the output signal values are to be clocked (stored) into state elements (registers or memory), as the value stored could be the unwanted spike/hazard rather than the stabilized correct signal value. Delayed clocking is used to avoid the effect of the spike/hazard being propagated to subsequent logic hardware. This solution is the method used in the context of this research as will be shown later.

It is clear from figure A.40 that the resulting synthesized hardware functions correctly and according to specification. This concludes the design cycle for this component.

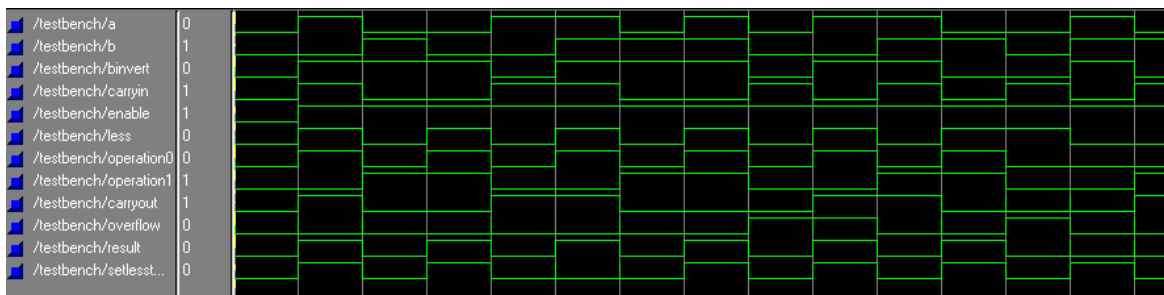


Figure A.40 Results of simulating the synthesized final 1-bit MIPS ALU for MSB using ModelSim.

Final 32-Bit MIPS ALU using Ripple Carry

➤ RTL Description

It is worth noting that every time the ALU is to perform a subtract operation, both *CarryIn* and *Binvert* signals (also known as *control lines*) are set to 1, while for addition and logical operations, both these control lines are set to 0. Therefore, control of the ALU can be simplified by combining these *CarryIn* and *Binvert* signals into a single control line called *Bnegate* [47, p.237]. This is shown in figure A.41 below.

In order to further tailor this ALU to the MIPS instruction set, conditional branch instructions must be supported [47]. This is described in further detail on pages 237 and 238 of [47].

Figure A.41 below shows the revised and final 32-bit ALU. The combination of the 1-bit *Bnegate* line and the 2-bit *Operation* lines makes up the 3-bit control lines for the ALU, commanding it to perform add, subtract, AND, OR, or set on less than [47]. Figure A.42 shows the function table outlining the ALU control lines and the corresponding ALU operations.

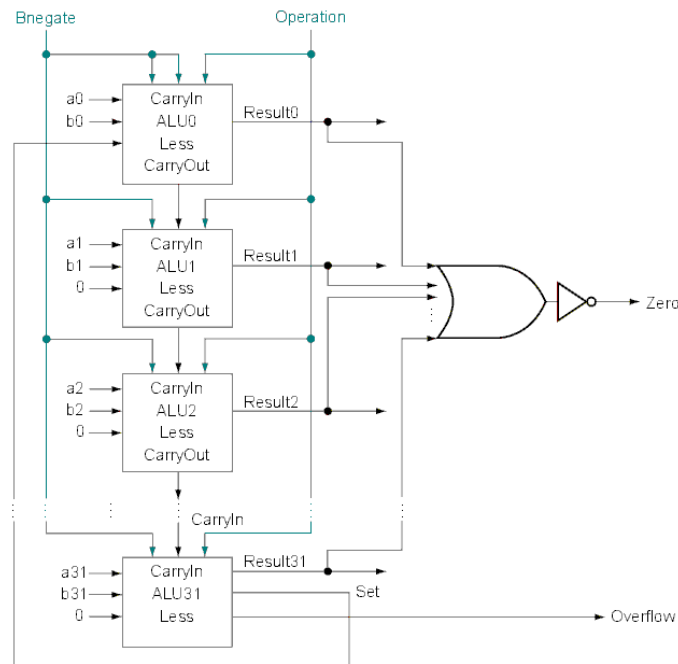


Figure A.41 Final 32-bit ripple carry ALU tailored to MIPS [47, p.240].

ALU Control Inputs:			Function	Datapath Outputs:	
Bit2 <i>Bnegate</i>	Bit1 <i>Operation1</i>	Bit0 <i>Operation0</i>		Result (31:0)	Zero (for BEQ and BNE only)
0	0	0	AND	$A(31:0) \text{ AND } B(31:0)$	x
0	0	1	OR	$A(31:0) \text{ OR } B(31:0)$	x
0	1	0	ADD	$A(31:0) + B(31:0)$	x
1	1	0	SUB	$A(31:0) - B(31:0)$	x
1	1	0	BEQ (=SUB)	x	1 if $A(31:0) = B(31:0)$ 0 if otherwise
1	1	0	BNE (=SUB)	x	1 if $A(31:0) \neq B(31:0)$ 0 if otherwise
1	1	1	SLT	1 if $A(31:0) < B(31:0)$ 0 if otherwise	x

Figure A.42 Function table for the finalized 32-bit ripple carry ALU tailored to MIPS [47, 48].

Referring to the function table in figure A.42, it is worth noting the following:

- When Function = ADD:

- For LSB 1-bit ALU only, we assign:

$$CarryIn = 0$$

- For each of the upper 31 1-bit ALUs (including MSB):

$$CarryIn = CarryOut \text{ from the FA of the previous 1-bit ALU}$$

- When Function = SUB (2's complement subtraction), the final 32-bit result is:

$$\begin{aligned} Result_{(32-bit)} &= A_{(32-bit)} + B'_{(32-bit)} + CarryIn_{(1-bit)} \\ &= A_{(32-bit)} + B'_{(32-bit)} + 1 \\ &= A_{(32-bit)} + (B'_{(32-bit)} + 1) \\ &= A_{(32-bit)} + (-B_{(32-bit)}) \\ &= A_{(32-bit)} - B_{(32-bit)} \\ &= A_{(32-bit)} + 2's \text{ complement of } B_{(32-bit)} \end{aligned}$$

This is achieved by performing the following:

- For LSB 1-bit ALU only, we assign:

$$CarryIn = Binvert = 1 \quad (\text{i.e. } Bnegate = 1)$$

- For each of the upper 31 1-bit ALUs (including MSB):

$$CarryIn = CarryOut \text{ from the FA of the previous 1-bit ALU}$$

- When Function = SLT, this will always set all bits of the subtraction result ($A_{(32-bit)} - B_{(32-bit)}$) except the LSB to 0, with the LSB set according to the following comparison:

- LSB = 1 if $A_{(32-bit)} < B_{(32-bit)}$ (i.e. $A_{(32-bit)} - B_{(32-bit)}$ is negative) \Rightarrow 1 means -ve

- LSB = 0 otherwise (i.e. $A_{(32-bit)} - B_{(32-bit)}$ is positive) \Rightarrow 0 means +ve

- Therefore, the input *Less* to each of the 1-bit ALUs will be set as follows:

- For upper 31 bits (bits 1 to 31): $Less = 0$

- For LSB (bit 0): $Less = Set \text{ output signal from MSB ALU (bit 31)}$

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the finalized 32-bit ripple carry ALU of figure A.41. The final schematic diagram is shown in figure A.44.

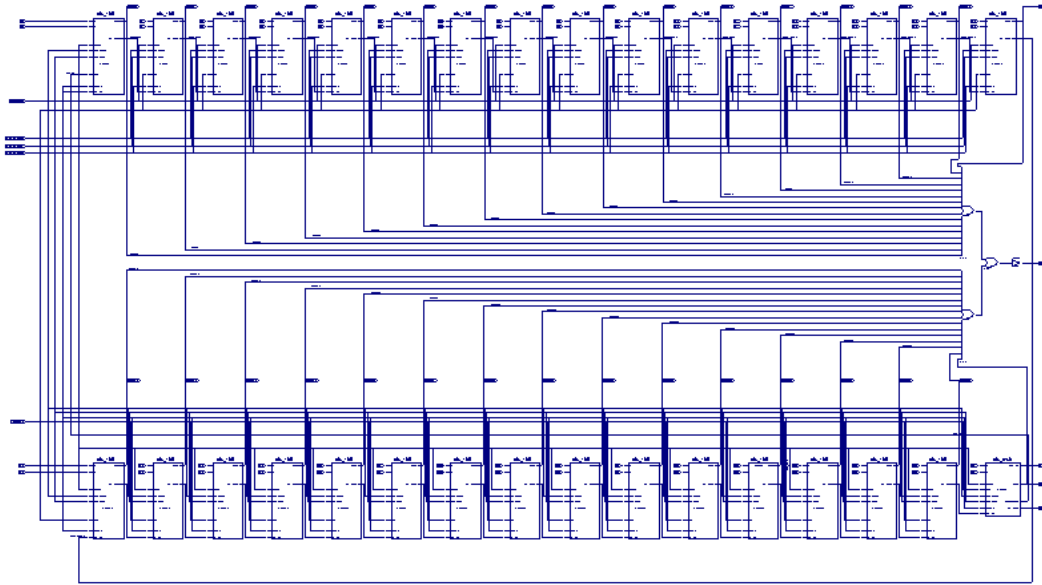


Figure A.44 Schematic diagram design entry for the finalized 32-bit ripple carry MIPS ALU in Schematic Editor.

After synthesis of the schematic diagram in figure A.44 using XST, the resulting VHDL code shown below was generated.

```
-- Vhdl model created from schematic alu_32bit_rc_sch.sch - Fri Oct 31 14:43:47 2003
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on
```

```
ENTITY alu_32bit_rc_sch IS
    PORT ( ALUOperation0 : IN STD_LOGIC;
          ALUOperation1 : IN STD_LOGIC;
          ALUOperation2 : IN STD_LOGIC;
          Less_Zero      : IN STD_LOGIC;
          Mux_Enable     : IN STD_LOGIC;
          a0              : IN STD_LOGIC;
          a1              : IN STD_LOGIC;
          .               : .;
          .               : .;
          a30             : IN STD_LOGIC;
          a31             : IN STD_LOGIC;
          b0              : IN STD_LOGIC;
          b1              : IN STD_LOGIC;
          .               : .;
          .               : . );
```

```

        b30      :    IN    STD_LOGIC;
        b31      :    IN    STD_LOGIC;
        CarryOut :    OUT   STD_LOGIC;
        Overflow :    OUT   STD_LOGIC;
        Result0  :    OUT   STD_LOGIC;
        Result1  :    OUT   STD_LOGIC;

        .
        .
        .

        Result30 :    OUT   STD_LOGIC;
        Result31 :    OUT   STD_LOGIC;
        Zero      :    OUT   STD_LOGIC);

end alu_32bit_rc_sch;

ARCHITECTURE SCHEMATIC OF alu_32bit_rc_sch IS
    SIGNAL OR16_i1_out :    STD_LOGIC;
    SIGNAL OR16_i2_out :    STD_LOGIC;
    SIGNAL OR2_i1_out  :    STD_LOGIC;
    SIGNAL Result0_DUMMY :    STD_LOGIC;

    .
    .
    .

    SIGNAL Result30_DUMMY :    STD_LOGIC;
    SIGNAL Result31_DUMMY :    STD_LOGIC;
    SIGNAL cout0_cin1    :    STD_LOGIC;

    .
    .
    .

    SIGNAL cout30_cin31 :    STD_LOGIC;
    SIGNAL slt31_0      :    STD_LOGIC;

    ATTRIBUTE BOX_TYPE :    STRING;
    ATTRIBUTE U_SET    :    STRING ;
    ATTRIBUTE U_SET OF OR16_i1 :    LABEL IS "OR16_i1_0";
    ATTRIBUTE U_SET OF OR16_i2 :    LABEL IS "OR16_i2_1";

    COMPONENT alu_1bit
        PORT ( a      :    IN    STD_LOGIC;
              b      :    IN    STD_LOGIC;
              binvert :    IN    STD_LOGIC;
              carryin :    IN    STD_LOGIC;
              enable  :    IN    STD_LOGIC;
              less    :    IN    STD_LOGIC;
              operation0 :    IN    STD_LOGIC;
              operation1 :    IN    STD_LOGIC;
              carryout :    OUT   STD_LOGIC;
              result  :    OUT   STD_LOGIC);

    END COMPONENT;

    COMPONENT alu_msb

```

```

        PORT ( a      :      IN      STD_LOGIC;
              b      :      IN      STD_LOGIC;
              binvert :      IN      STD_LOGIC;
              carryin :      IN      STD_LOGIC;
              enable  :      IN      STD_LOGIC;
              less    :      IN      STD_LOGIC;
              operation0 :      IN      STD_LOGIC;
              operation1 :      IN      STD_LOGIC;
              carryout :      OUT     STD_LOGIC;
              overflow :      OUT     STD_LOGIC;
              result  :      OUT     STD_LOGIC;
              setlessthan :      OUT     STD_LOGIC);
    END COMPONENT;

    COMPONENT INV
        PORT ( I      :      IN      STD_LOGIC;
              O      :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF INV : COMPONENT IS "BLACK_BOX";
    COMPONENT OR16_MXILINX_alu_32bit_rc_sch
        PORT ( I0      :      IN      STD_LOGIC;
              I1      :      IN      STD_LOGIC;
              .        :      .
              .        :      .
              I15     :      IN      STD_LOGIC;
              O      :      OUT     STD_LOGIC);
    END COMPONENT;

    COMPONENT OR2
        PORT ( I0      :      IN      STD_LOGIC;
              I1      :      IN      STD_LOGIC;
              O      :      OUT     STD_LOGIC);
    END COMPONENT;

    ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
    BEGIN
        Result0 <= Result0_DUMMY;
        Result1 <= Result1_DUMMY;
        .
        .
        Result30 <= Result30_DUMMY;
        Result31 <= Result31_DUMMY;

        ALU_bit0 : alu_1bit
            PORT MAP (a=>a0, b=>b0, binvert=>ALUOperation2, carryin=>ALUOperation2,
                    enable=>Mux_Enable, less=>slt31_0, operation0=>ALUOperation0,
                    operation1=>ALUOperation1, carryout=>cout0_cin1, result=>Result0_DUMMY);
    
```

```

ALU_bit1 : alu_1bit
    PORT MAP (a=>a1, b=>b1, binvert=>ALUOperation2, carryin=>cout0_cin1,
              enable=>Mux_Enable, less=>Less_Zero, operation0=>ALUOperation0,
              operation1=>ALUOperation1, carryout=>cout1_cin2, result=>Result1_DUMMY);
    .
    .

ALU_bit31 : alu_msb
    PORT MAP (a=>a31, b=>b31, binvert=>ALUOperation2, carryin=>cout30_cin31,
              enable=>Mux_Enable, less=>Less_Zero, operation0=>ALUOperation0,
              operation1=>ALUOperation1, carryout=>CarryOut, overflow=>Overflow,
              result=>Result31_DUMMY, setlessthan=>slt31_0);

INV_i1 : INV
    PORT MAP (I=>OR2_i1_out, O=>Zero);

OR16_i1 : OR16_MXILINX_alu_32bit_rc_sch
    PORT MAP (I0=>Result0_DUMMY, I1=>Result1_DUMMY, I10=>Result10_DUMMY,
              I11=>Result11_DUMMY, I12=>Result12_DUMMY, I13=>Result13_DUMMY,
              I14=>Result14_DUMMY, I15=>Result15_DUMMY, I2=>Result2_DUMMY,
              I3=>Result3_DUMMY, I4=>Result4_DUMMY, I5=>Result5_DUMMY,
              I6=>Result6_DUMMY, I7=>Result7_DUMMY, I8=>Result8_DUMMY,
              I9=>Result9_DUMMY, O=>OR16_i1_out);

OR16_i2 : OR16_MXILINX_alu_32bit_rc_sch
    PORT MAP (I0=>Result31_DUMMY, I1=>Result30_DUMMY, I10=>Result21_DUMMY,
              I11=>Result20_DUMMY, I12=>Result19_DUMMY, I13=>Result18_DUMMY,
              I14=>Result17_DUMMY, I15=>Result16_DUMMY, I2=>Result29_DUMMY,
              I3=>Result28_DUMMY, I4=>Result27_DUMMY, I5=>Result26_DUMMY,
              I6=>Result25_DUMMY, I7=>Result24_DUMMY, I8=>Result23_DUMMY,
              I9=>Result22_DUMMY, O=>OR16_i2_out);

OR2_i1 : OR2
    PORT MAP (I0=>OR16_i2_out, I1=>OR16_i1_out, O=>OR2_i1_out);

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above finalized 32-bit ripple carry MIPS ALU, was generated. Figures A.45 and A.46 show the resulting top level RTL symbols for the synthesized ALU. Note that in both figures A.45 and A.46 there is an input signal named *less_zero*. This signal is actually the *Less* input signal to all the upper 31 1-bit ALUs (i.e. ALU1 to ALU31) shown in figure A.41. This signal must always be assigned a value of 0 (therefore the name *less_zero* in my implementation) in order for the SLT instruction to execute properly.

Figures A.47 and A.48 show the resulting top level RTL schematic diagrams.

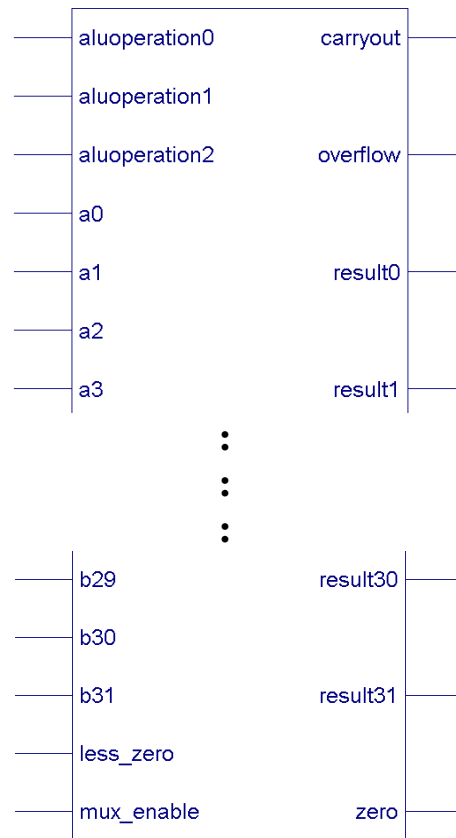


Figure A.45 Resulting top level RTL symbol for the finalized 32-bit ripple carry MIPS ALU.

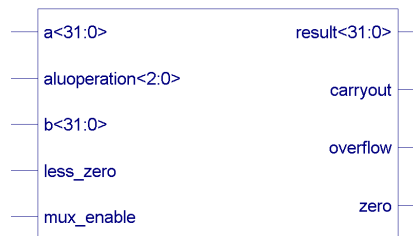


Figure A.46 A compact version of the resulting top level RTL symbol for the finalized 32-bit ripple carry MIPS ALU.



Figure A.47 First section of the resulting top level RTL schematic diagram for the finalized 32-bit ripple carry MIPS ALU, showing the lower 31 1-bit ALUs.

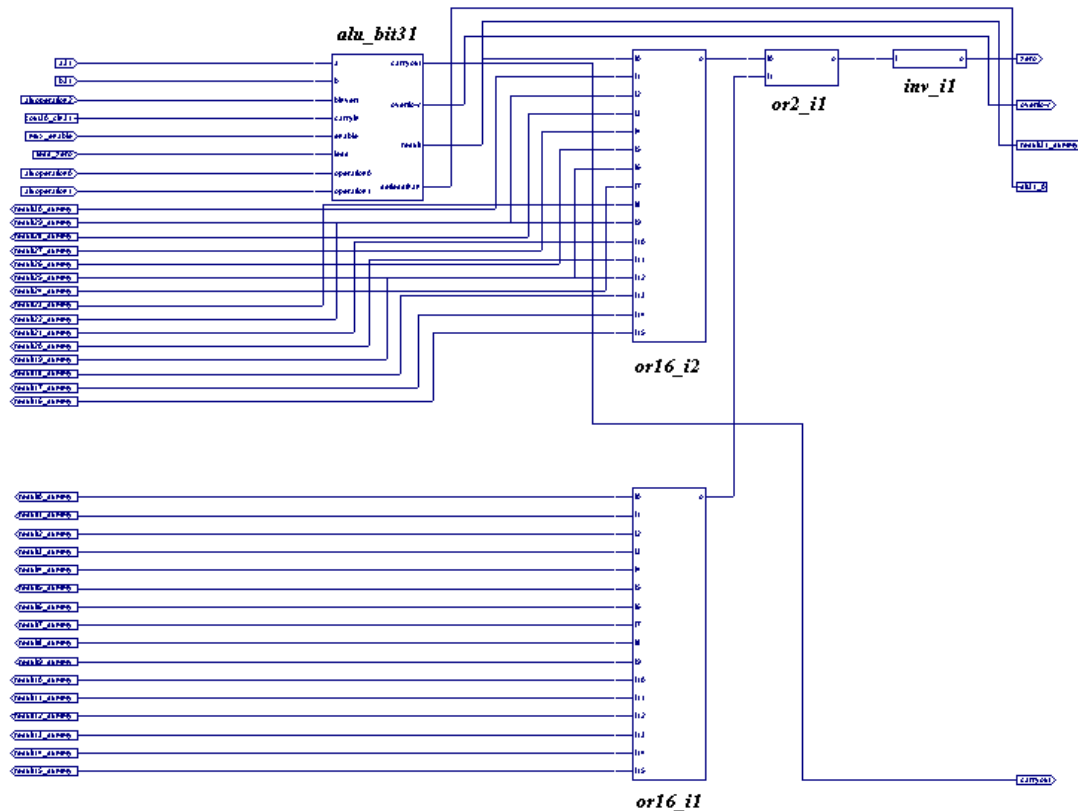


Figure A.48 Second and last section of the resulting top level RTL schematic diagram for the finalized 32-bit ripple carry MIPS ALU, showing the MSB ALU.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 32-bit ALU using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```

Design Statistics:
# I/Os                                     : 104

Cell Usage:
# BELS                                     : 880
# and2                                    : 160
# and2b1                                  : 32
# and3                                    : 192
# and3b1                                  : 64
# gnd                                     : 2
# inv                                     : 129
# LUT1                                    : 64
# muxcy                                   : 2
# muxcy_1                                : 6

```

```

#      muxf5                : 32
#      or2                  : 129
#      or3                  : 32
#      or4                  : 32
#      vcc                  : 2
#      xor2                 : 2
# IO Buffers                : 104
#      IBUF                 : 69
#      OBUF                 : 35
# Logical                   : 8
#      nor4                 : 8
# Others                    : 8
#      fmap                 : 8

```

Device utilization summary:

Number of Slices:	36	out of	46592	0%
Number of 4 input LUTs:	64	out of	93184	0%
Number of bonded IOBs:	104	out of	1108	9%

➤ *Place-and-Route onto the FPGA*

In figure A.49, FPGA Editor shows the synthesized 1-bit ALU after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the left side of the FPGA chip.

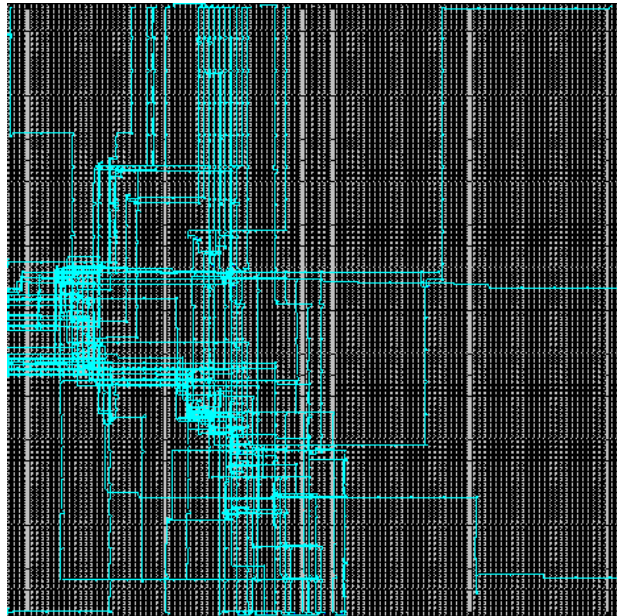


Figure A.49 *FPGA Editor showing the synthesized final 32-bit ripple carry MIPS ALU after place-and-route onto the target Virtex-II FPGA chip.*

➤ Simulation Results

Figures A.50 to A.55 show the waveform results of simulating the finalized 32-bit ripple carry MIPS ALU VHDL behavioural model in Mentor Graphics ModelSim. In both figures A.50 and A.51, all the waveforms are in binary format. However, in figures A.52 through to A.55, all the waveforms are in binary format except for the signals *a*, *b*, and *result* which are in signed decimal format.

When checking these waveforms, the following points are worth noting:

- ❑ These waveforms are compared against the specified behavior of the ALU as shown earlier in the function table of figure A.42.
- ❑ In each figure, the time delay is shown for every value of the output signal(s) in question. This information will be used later on for comparison between the 32-bit ripple carry ALU and the carry lookahead one.
- ❑ When the ALU Function is AND (figure A.50):
 - The only output signal of interest is *result*.
 - Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexer is disabled (input signal *mux_enable* = 0).

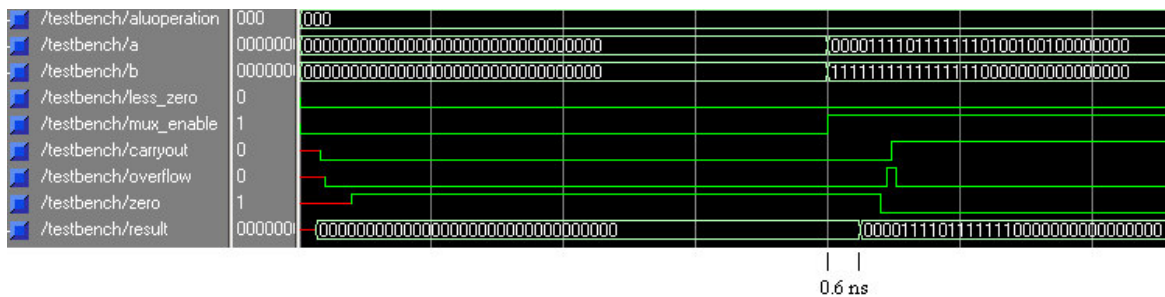


Figure A.50 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = AND.

- ❑ When the ALU Function is OR (figure A.51):
 - The only output signal of interest is *result*.
 - Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexer is disabled (input signal *mux_enable* = 0).

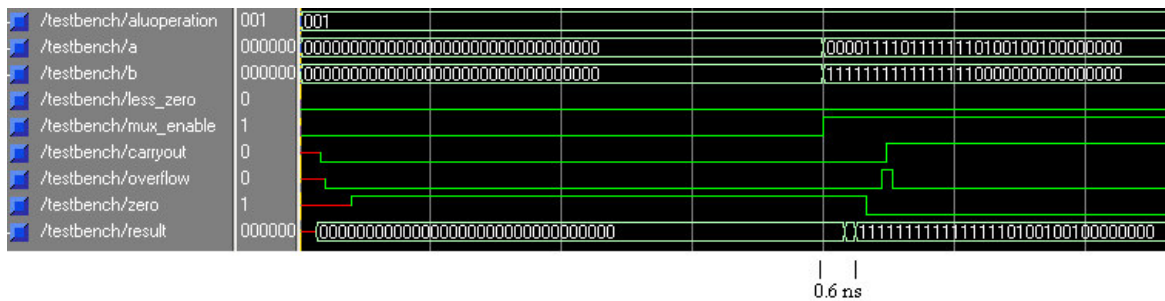


Figure A.51 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = OR.

- When the ALU Function is ADD (figure A.52):
 - Both output signals *carryout* and *result* are of interest.
 - Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexer is disabled (input signal *mux_enable* = 0).
 - Due to the fact that this ALU is based on ripple carry addition, the *result* output goes through a number of unwanted intermediate values until settling at the correct value. For example, in figure A.52, during the third 10 ns interval, the *result* output settles at the correct value of decimal 995339259 after 1.3 ns from the time the inputs *a* and *b* have been applied.

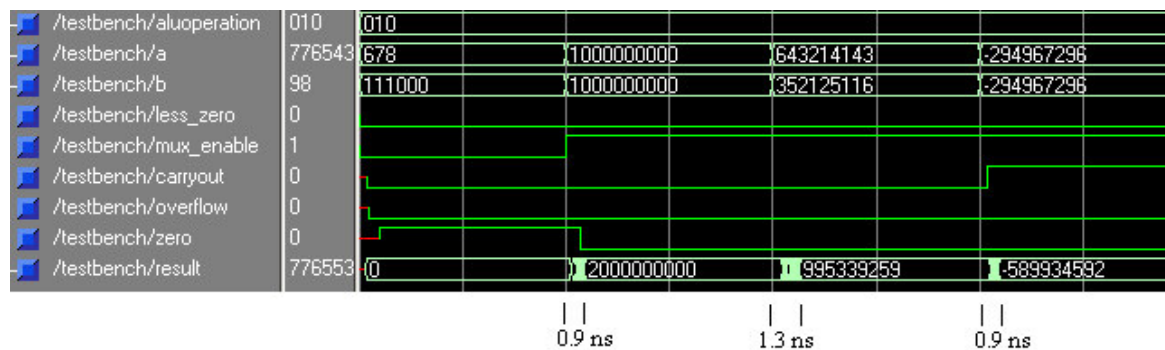


Figure A.52 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = ADD.

- When the ALU Function is SUB (figure A.53):
 - Both output signals *carryout* and *result* are of interest.

- The same ripple carry argument as the above ADD case applies here. The *result* output does not settle down to the correct value only until some noticeable time after the inputs *a* and *b* have been applied.

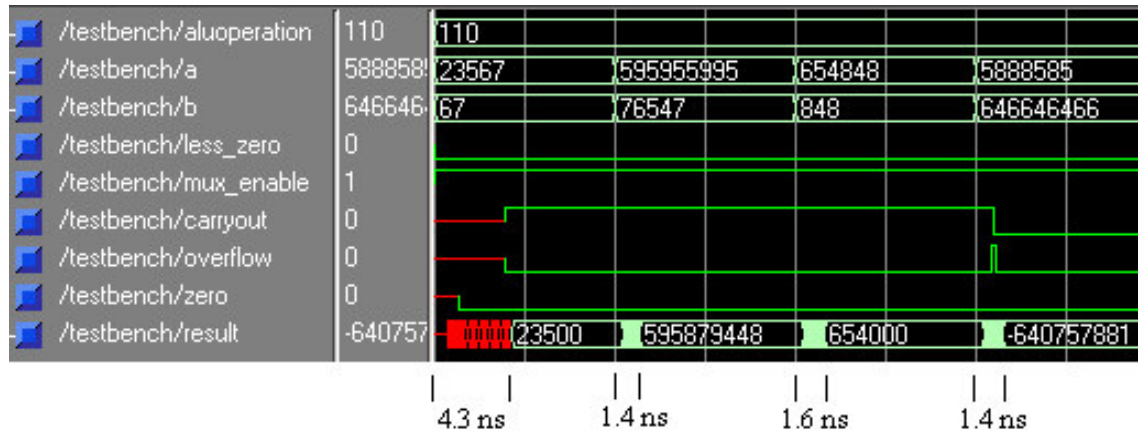


Figure A.53 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = SUB.

- When the ALU Function is either BEQ or BNE (figure A.54):
 - The only output signal of interest is *zero*.
 - However, the effects of the ripple carry addition/subtraction are quite severe in this case. Figure A.54 shows that it can take up to 8 ns (this is during the second 10 ns time interval) for the *zero* output to be generated which is only shortly (about 1 ns) after the result output has settled to the correct value.

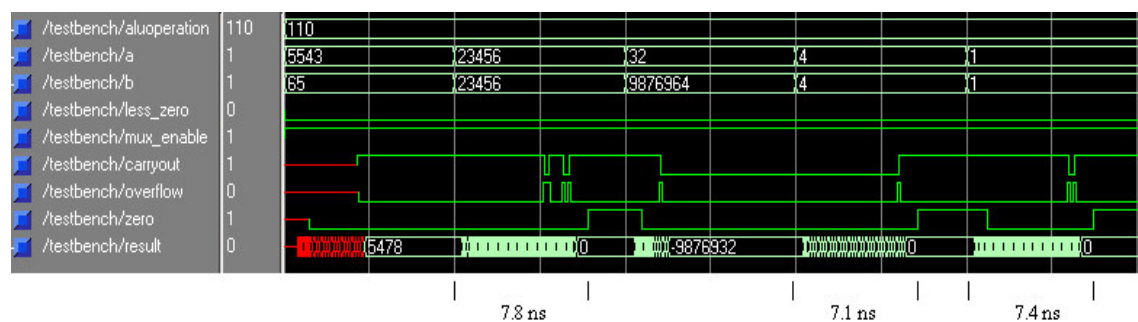


Figure A.54 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = BEQ/BNE.

- When the ALU Function is SLT (figure A.55):
 - The only output signal of interest is *result*.

- Again, as is the case above with BEQ/BNE, the *result* output might not stabilize to the correct value only until 7 ns after the *a* and *b* inputs have been applied.
- ❑ Although the output signal *overflow* is implemented, it will not be utilized in any aspect within the context of this research, but rather for future research.

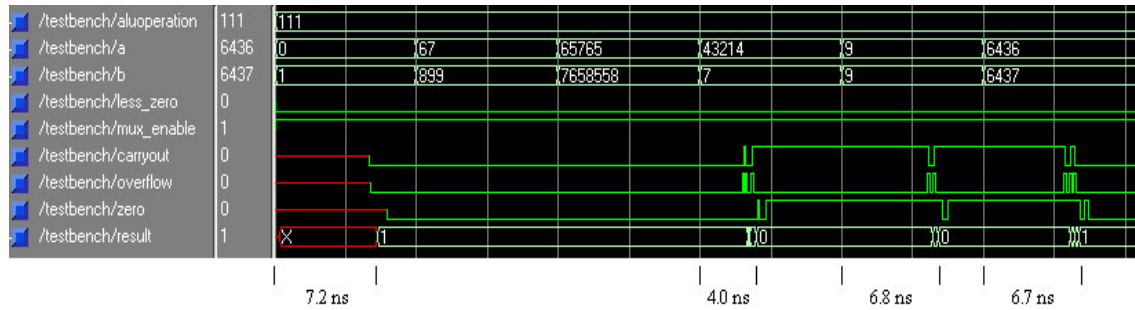


Figure A.55 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = SLT.

It is clear from these figures (A.50 to A.55) that the resulting synthesized hardware functions correctly and according to specification. This concludes the design cycle for this component.

However, the various output signals in all these figures (A.50 to A.55) display hazards which result in a combination of spikes, unwanted intermediate values, and noticeably delayed correct values. In the context of this research, one or more of the following remedies have been implemented:

- ❑ Increase the time interval (for combinational logic) or clock cycle time (for sequential logic) from 10 ns to 20 ns or even to 100 ns or more.
- ❑ Delayed Clocking: When the output value from a combinational element is to be clocked in (stored into) a state element (register or memory), the rising edge of the clock is delayed to ensure that the combinational output has settled and stabilized to the correct value.
- ❑ Or, a combination of both of the above techniques can be implemented at the same time.

Carry Lookahead (CLA) for Faster Addition

So far, we have seen how a standard 32-bit ALU using ripple carry works. The problem with ripple carry is that the chain reaction is too slow to be used in time-critical hardware [47]. However, there is a solution.

The key to speeding up addition is determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the \log_2 of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry [48, p.B-38].

One such fast carry scheme is the Carry Lookahead (CLA) scheme, which is described in much detail on pages 241 to 249 of [47].

The Big Picture First: 16-Bit ALU using Carry Lookahead

➤ *RTL Description*

Figure A.56 below shows four 4-bit adders connected with a 4-bit carry-lookahead unit to form a 16-bit ALU. It is worth mentioning here that the *carry* signals are generated from the carry-lookahead unit and not from the 4-bit ALUs [47].

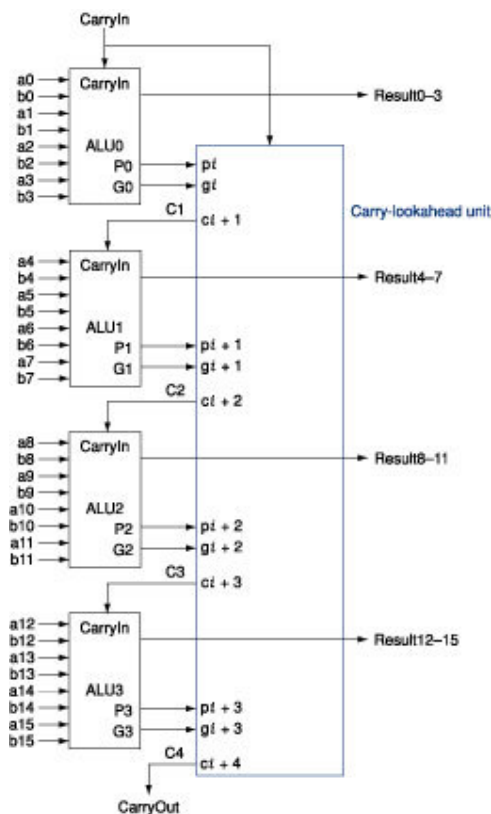


Figure A.56 Four 4-bit ALUs using a 4-bit carry lookahead unit to form a 16-bit ALU [47, p.246].

The Bigger Picture: 32-Bit ALU using Carry Lookahead

For the context of this research, the ALU to be used in implementing the MIPS processor onto the FPGA is a 32-bit ALU using CLA rather than ripple carry. To elaborate on the details of the design hierarchy, figure A.57 shows the layers of abstraction for such ALU. This 32-bit ALU using CLA consists of one 2-bit CLA unit and two 16-bit ALUs both using CLA. However, the second 16-bit ALU using CLA is modified to cater for the MSB. Each 16-bit ALU using CLA, in turn, is made up of one 4-bit CLA unit and four 4-bit ALUs all using CLA. Also, as shown in figure A.58, each 4-bit ALU using CLA is made up of one 4-bit CLA unit and four 1-bit ALUs; all of which have support for the *propagate* and *generate* signals (as will be shown later). However, figure A.57 shows that the last 4-bit ALU using CLA (contained within the 16-bit ALU for MSB using CLA) is modified to cater for the MSB and figure A.59 elaborates more on this 4-bit ALU for MSB using CLA by showing its design hierarchy and that this ALU is made up of one 4-bit CLA unit and four 1-bit ALUs; all four of which have support for the *propagate* and *generate* signals: the first three 1-bit ALUs are similar (as will be shown later) while the last 1-bit ALU is modified to cater for the MSB (as will be shown later too).

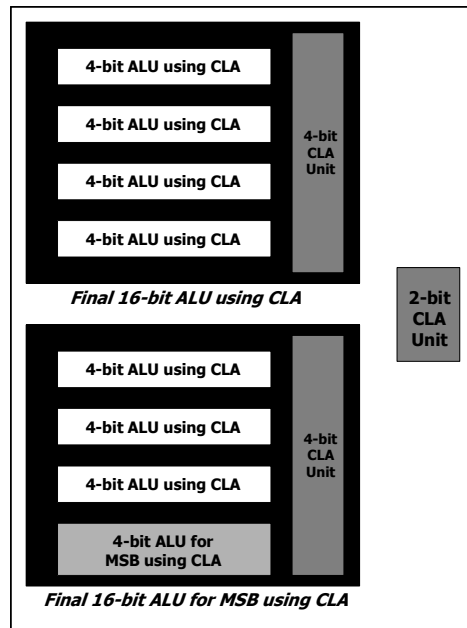


Figure A.57 Design hierarchy and abstraction for a 32-bit ALU using carry lookahead.

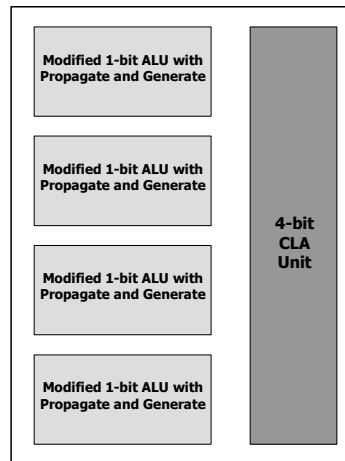


Figure A.58 Design hierarchy and abstraction for a 4-bit ALU using carry lookahead.

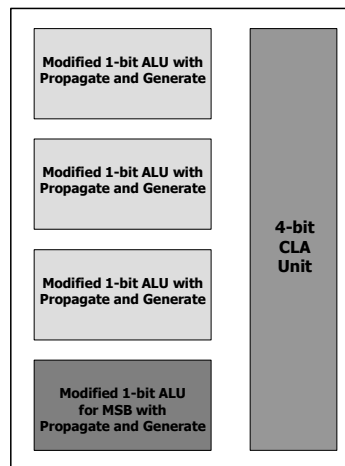


Figure A.59 Design hierarchy and abstraction for a 4-bit ALU for MSB using carry lookahead.

To summarize, “carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, *generate* and *propagate*. The former creates a carry regardless of the carry input, and the other passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity” [47, p.249].

In the following subsections, I will first introduce the VHDL synthesis and simulation of the modified 1-bit ALU with *propagate* and *generate* signals, then the 4-bit CLA unit, followed by the 4-bit ALU using CLA, then the final 16-bit ALU using CLA. Afterwards, the VHDL synthesis and simulation of the modified 1-bit ALU for MSB with *propagate* and *generate* signals is covered, followed by the 4-bit ALU for MSB using CLA, then the final 16-bit ALU for MSB using CLA. Finally, this section ends by

combining all these blocks together (as shown earlier in figure A.57) to present the VHDL synthesis and simulation of the final 32-bit MIPS ALU using CLA.

Starting Point: Modified 1-Bit ALU with Propagate and Generate

➤ *RTL and Boolean Description*

The derivation of the two signals *propagate* and *generate* for a 1-bit ALU using CLA is detailed on pages 241 and 242 of [47]. This is repeated here for convenience:

$$gi = ai \cdot bi$$

$$pi = ai + bi$$

These two signals *propagate* and *generate* replace the need for a full-adder in the ALU. The rest of the design for this modified 1-bit ALU with *propagate* and *generate* is the same as the final 1-bit MIPS ALU discussed earlier and shown in figure A.27.

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the modified 1-bit ALU with *propagate* and *generate* signals. The final schematic diagram is shown in figure A.60. Note that the Sum function is an XOR gate and the Carry function is replaced by the *generate* and *propagate* signals.

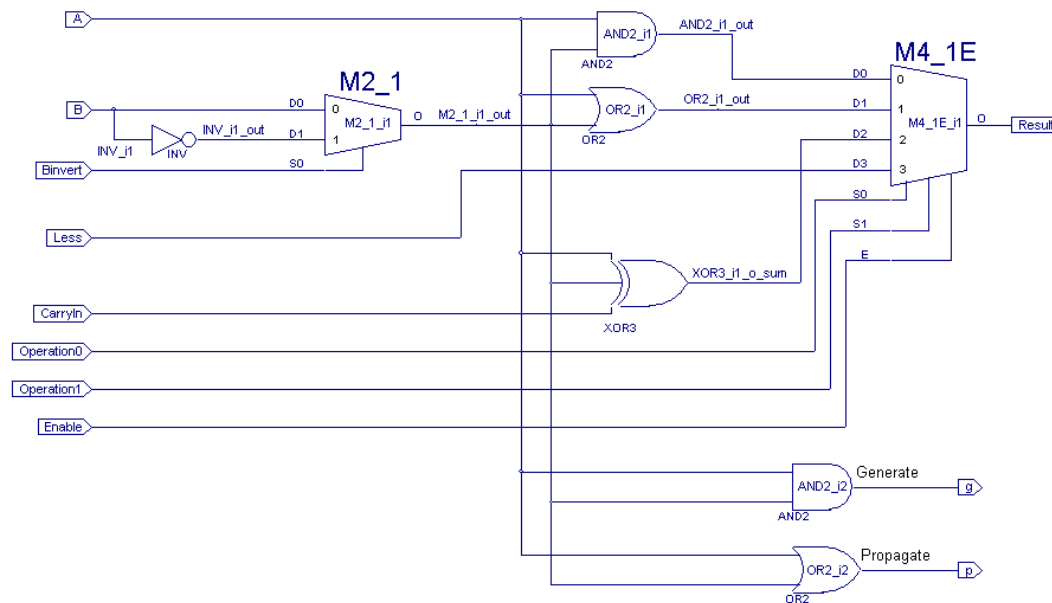


Figure A.60 Schematic diagram design entry for the modified 1-bit ALU with propagate and generate signals (instead of a full adder) in Schematic Editor.

➤ Simulation Results

No simulation has been carried out for this component as it is similar (except for the *generate* and *propagate* output signals) to the previous ALUs synthesized and simulated. However, the final 32-bit ALU using carry lookahead will definitely be simulated and tested thoroughly.

4-bit Carry Look-Ahead Unit

➤ RTL and Boolean Description

The CLA is a combinational logic component. This 4-bit CLA unit synthesized here is what is shown in figures A.56, A.57, A.58 and A.59. The derivation for the Boolean expressions for generating the output signals is covered in detail on pages 241 to 247 of [47] and, therefore, will not be repeated here. However, the VHDL code below is based on these Boolean expressions.

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing a 4-bit CLA unit, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

-- 4-bit Carry Look-Ahead Unit:

entity CLA_4bit_vhd is
    Port ( c0 : in std_logic;
          g0 : in std_logic;
          g1 : in std_logic;
          g2 : in std_logic;
          g3 : in std_logic;
          p0 : in std_logic;
          p1 : in std_logic;
          p2 : in std_logic;
          p3 : in std_logic;
          c1 : out std_logic;
          c2 : out std_logic;
          c3 : out std_logic;
          -- c4 : out std_logic; -- No need for it at this time!
          sg : out std_logic;
          sp : out std_logic);
end CLA_4bit_vhd;

architecture Behavioral of CLA_4bit_vhd is

begin

    c1 <= g0 or (p0 and c0);
    c2 <= g1 or (p1 and g0) or (p1 and p0 and c0);
    c3 <= g2 or (p2 and g1) or (p2 and p1 and g0) or (p2 and p1 and p0 and c0);

    -- No need for the following CarryOut signal at this time:
    -- c4 <= g3 or (p3 and g2) or (p3 and p2 and g1) or (p3 and p2 and p1 and g0)
    --      or (p3 and p2 and p1 and p0 and c0);

    sp <= p3 and p2 and p1 and p0;
    sg <= g3 or (p3 and g2) or (p3 and p2 and g1) or (p3 and p2 and p1 and g0);

end Behavioral;
```

➤ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 4-bit CLA unit, was generated. Figure A.62 shows the resulting top level RTL symbol for the synthesized 4-bit CLA unit while figure A.63 shows the resulting top level RTL schematic diagram. Figures A.64 to A.73 elaborate more on the top level RTL schematic diagram and show the resulting gate level schematic diagrams.

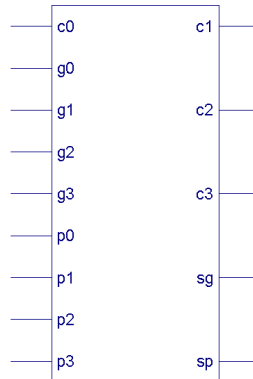


Figure A.62 Resulting top level RTL symbol for the synthesized 4-bit CLA unit.

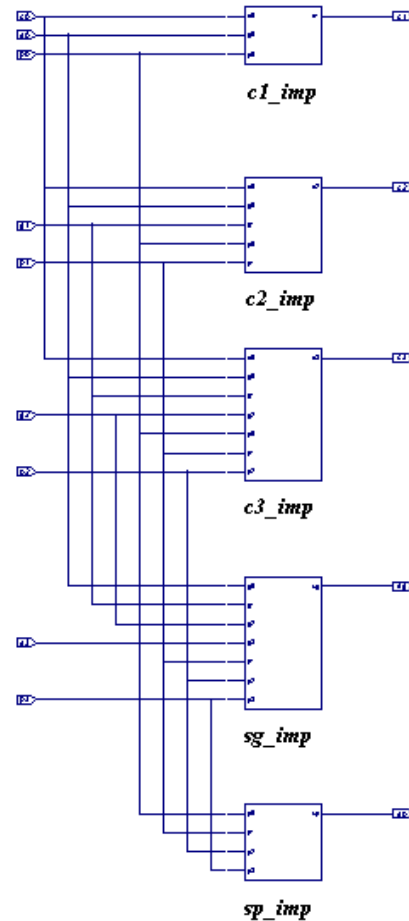


Figure A.63 Resulting top level RTL schematic for the synthesized 4-bit CLA unit.

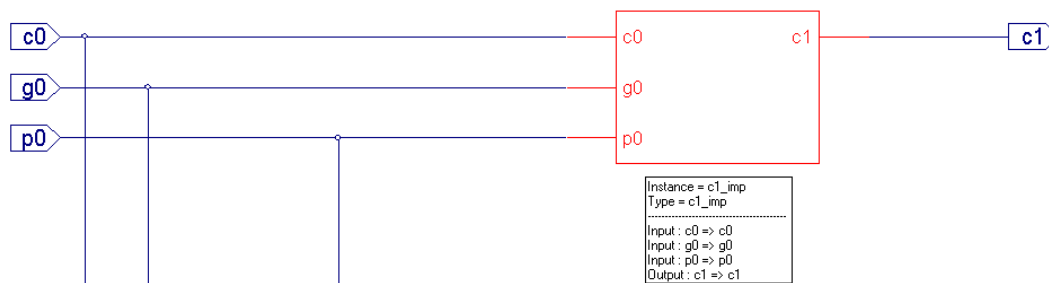


Figure A.64 Magnification of block *c1_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

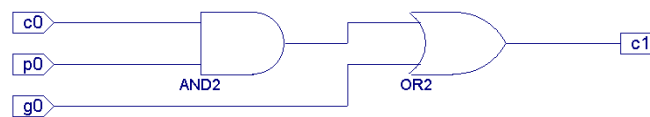


Figure A.65 Gate level schematic for block *c1_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

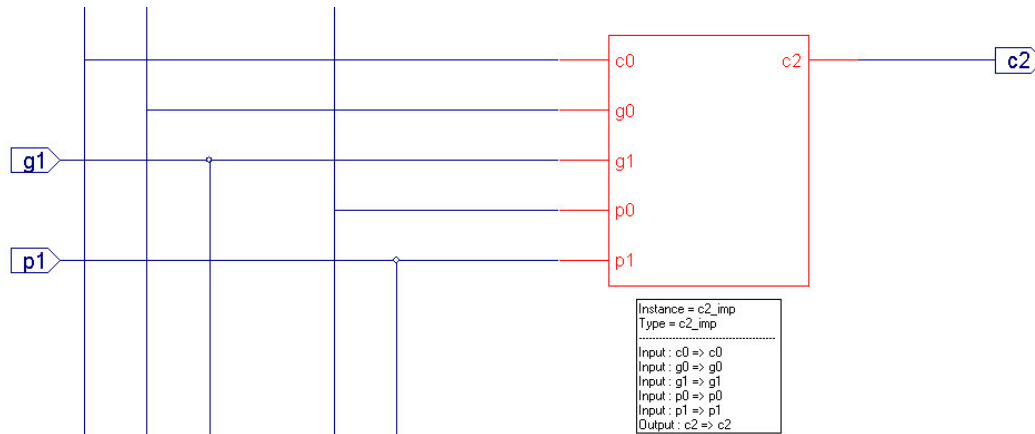


Figure A.66 Magnification of block *c2_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

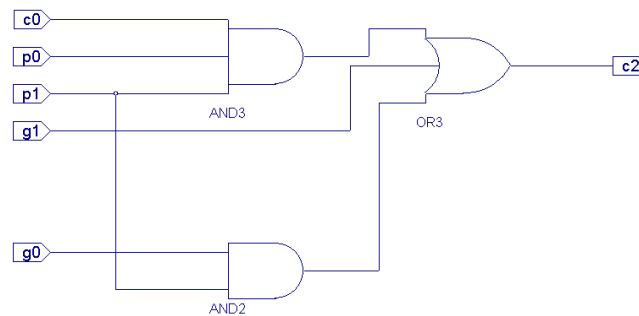


Figure A.67 Gate level schematic for block *c2_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

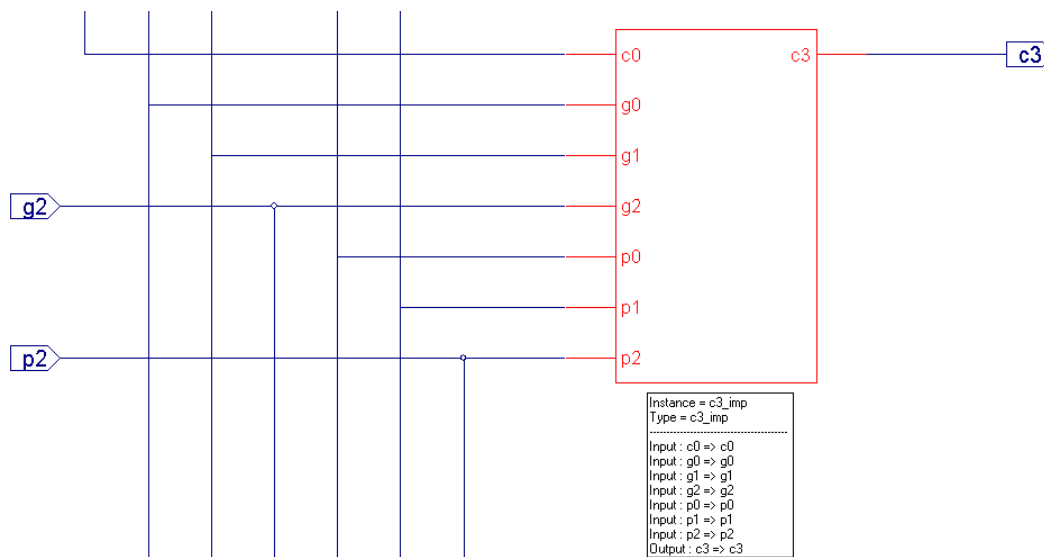


Figure A.68 Magnification of block *c3_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

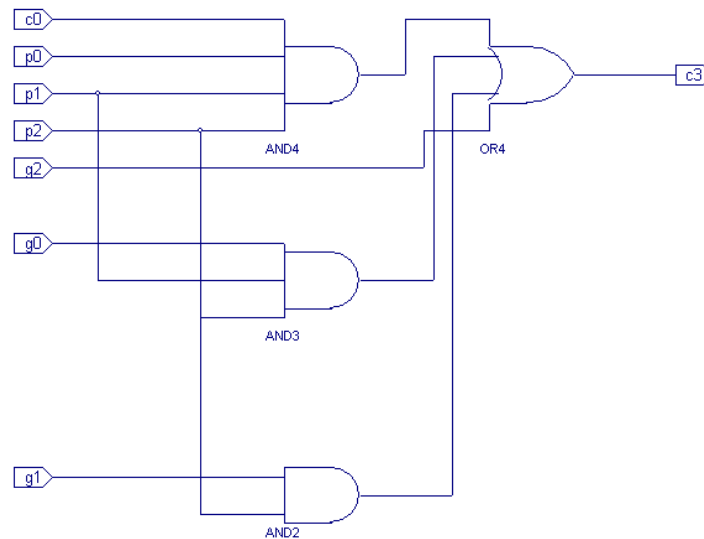


Figure A.69 Gate level schematic for block *c3_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

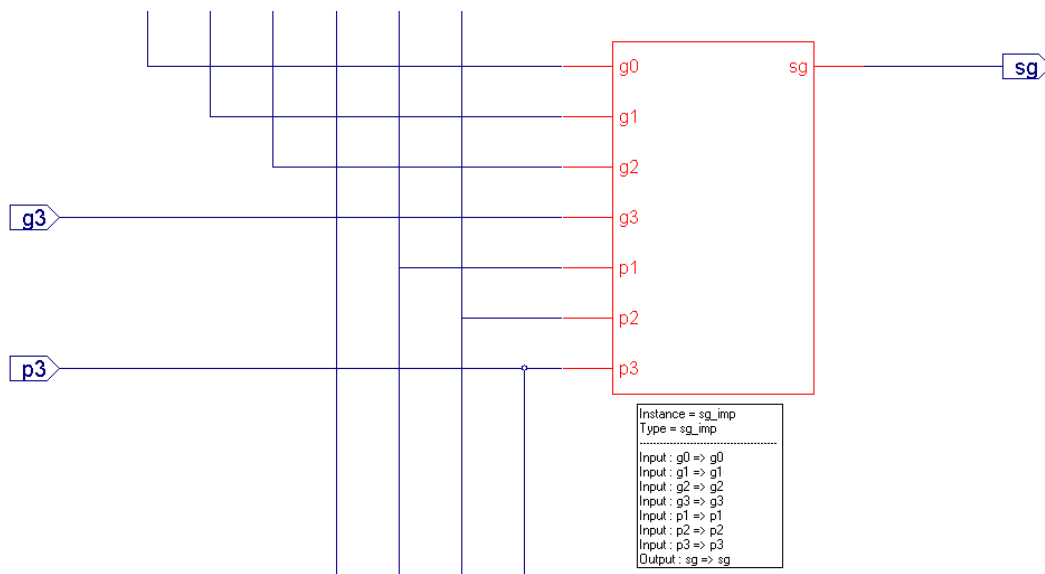


Figure A.70 Magnification of block *sg_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

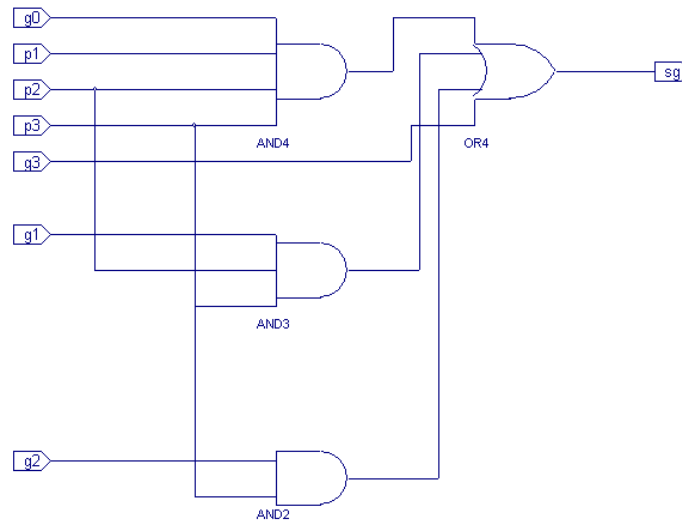


Figure A.71 Gate level schematic for block *sg_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

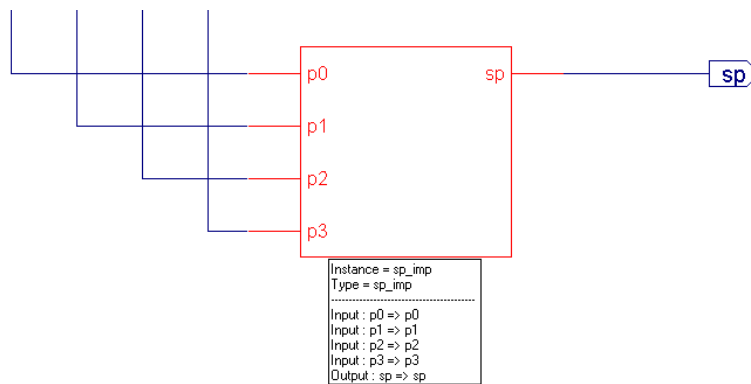


Figure A.72 Magnification of block *sp_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

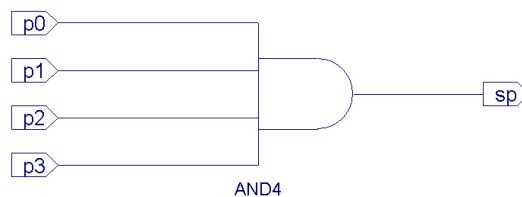


Figure A.73 Gate level schematic for block *sp_imp* in the resulting top level RTL schematic for the synthesized 4-bit CLA unit of figure A.63.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the above 4-bit CLA unit using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

# IOs	:	14
-------	---	----

Cell Usage:

# BELS	:	6
# LUT3	:	3
# LUT4	:	3
# IO Buffers	:	14
# IBUF	:	9
# OBUF	:	5

Device utilization summary:

Number of Slices:	3	out of	46592	0%
Number of 4 input LUTs:	6	out of	93184	0%
Number of bonded IOBs:	14	out of	1108	1%

➤ *Place-and-Route onto the FPGA*

In figure A.74, FPGA Editor shows the synthesized 4-bit CLA unit after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the lower section of the FPGA chip.

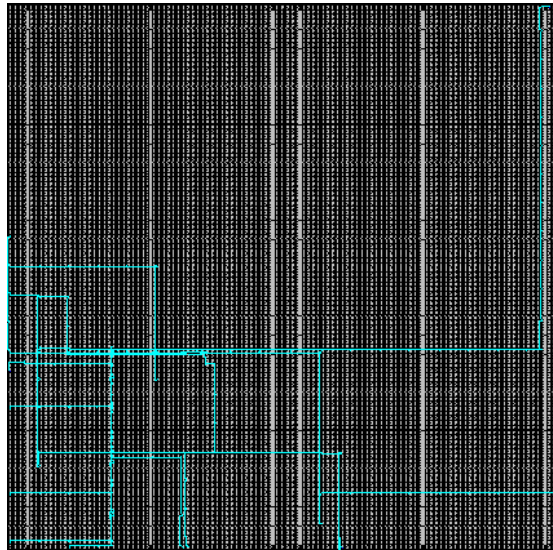


Figure A.74 *FPGA Editor showing the synthesized 4-bit CLA unit after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Simulating and testing a CLA unit on its own does not provide much insight into the correctness of its functionality as opposed to simulating and testing it as part of a multi-bit ALU block that implements CLA. Therefore, there is no need for this individual simulation to be performed at this stage.

4-Bit ALU using Carry Lookahead

➤ *RTL Description*

Figures A.56 to A.58 have previously elaborated on the RTL description for the 4-bit ALU using CLA, which is also described in [47] and [48].

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the 4-bit ALU using CLA, which was shown in figures A.56, A.57 and A.58. The final schematic diagram is shown in figure A.75. Figures A.76 to A.80 show magnifications of different sections of the schematic diagram in figure A.75.

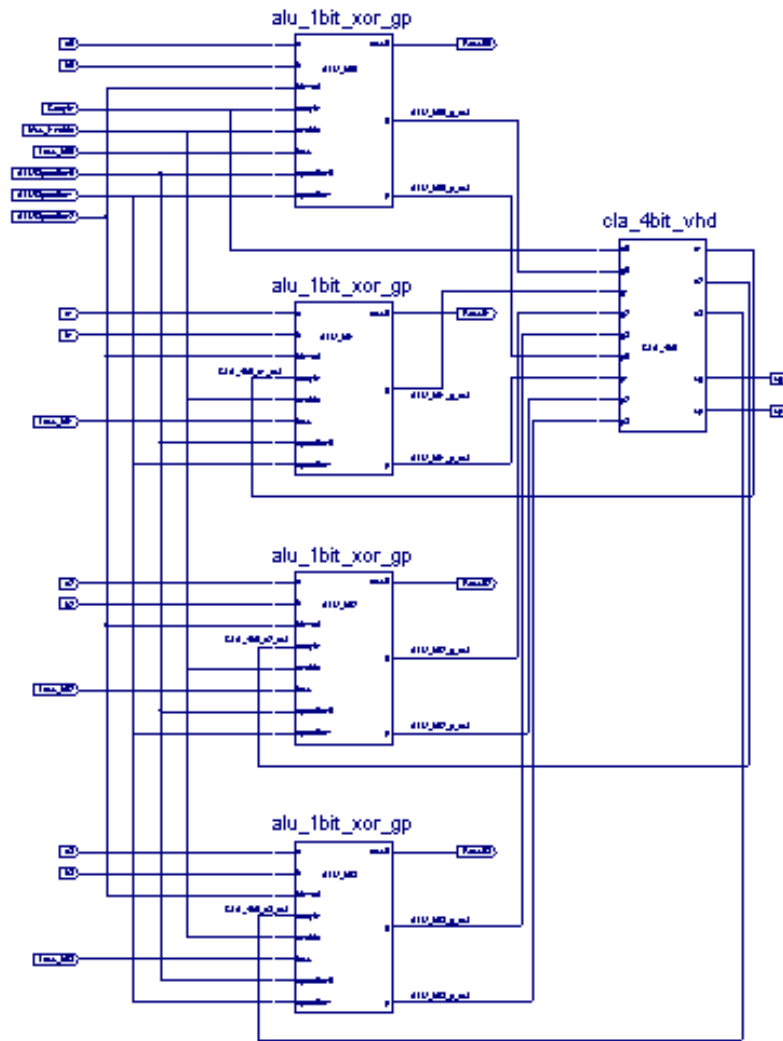


Figure A.75 Schematic diagram design entry for the 4-bit ALU with CLA in Schematic Editor.

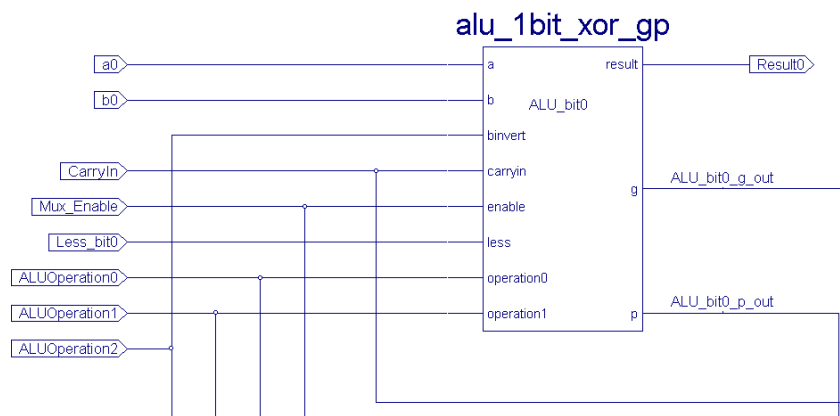


Figure A.76 Magnification of a section of figure A.75 showing ALU_bit0 in the schematic diagram for the 4-bit ALU with CLA.

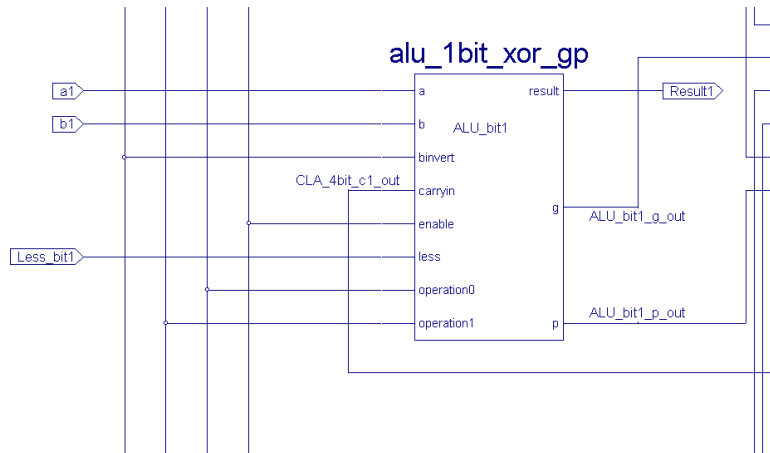


Figure A.77 Magnification of a section of figure A.75 showing ALU_bit1 in the schematic diagram for the 4-bit ALU with CLA.

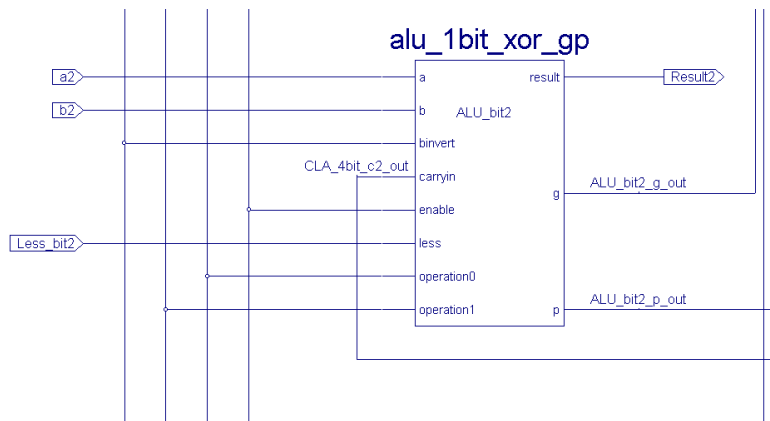


Figure A.78 Magnification of a section of figure A.75 showing ALU_bit2 in the schematic diagram for the 4-bit ALU with CLA.

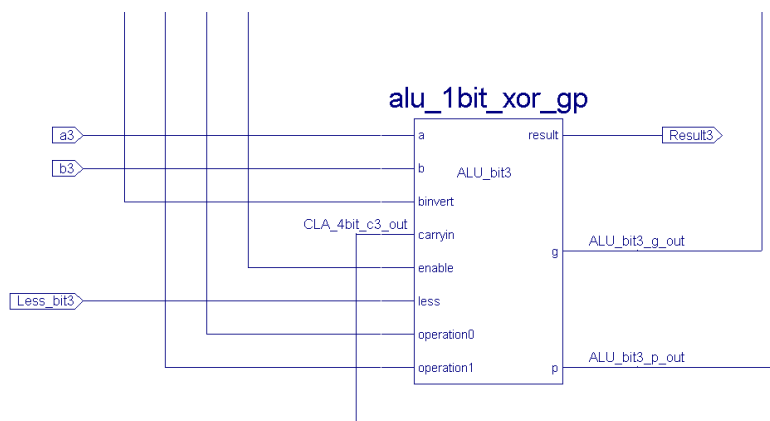


Figure A.79 Magnification of a section of figure A.75 showing ALU_bit3 in the schematic diagram for the 4-bit ALU with CLA.

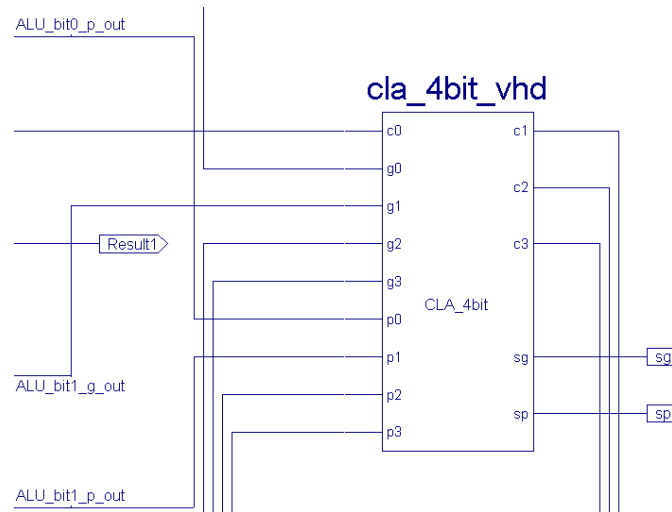


Figure A.80 Magnification of a section of figure A.75 showing CLA_4bit in the schematic diagram for the 4-bit ALU with CLA.

After synthesis of the schematic diagram in figure A.75 using XST, the resulting VHDL code shown below was generated.

-- Vhdl model created from schematic alu_4bit_cla_nz_sch.sch - Mon May 05 12:56:13 2003

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on
```

```
ENTITY alu_4bit_cla_nz IS
    PORT ( ALUOperation0 : IN STD_LOGIC;
          ALUOperation1 : IN STD_LOGIC;
          ALUOperation2 : IN STD_LOGIC;
          CarryIn       : IN STD_LOGIC;
          Less_bit0     : IN STD_LOGIC;
          Less_bit1     : IN STD_LOGIC;
          Less_bit2     : IN STD_LOGIC;
          Less_bit3     : IN STD_LOGIC;
          Mux_Enable    : IN STD_LOGIC;
          a0            : IN STD_LOGIC;
          a1            : IN STD_LOGIC;
          a2            : IN STD_LOGIC;
          a3            : IN STD_LOGIC;
          b0            : IN STD_LOGIC;
          b1            : IN STD_LOGIC;
          b2            : IN STD_LOGIC;
```

```

        b3          :      IN      STD_LOGIC;
        Result0     :      OUT     STD_LOGIC;
        Result1     :      OUT     STD_LOGIC;
        Result2     :      OUT     STD_LOGIC;
        Result3     :      OUT     STD_LOGIC;
        sg          :      OUT     STD_LOGIC;
        sp          :      OUT     STD_LOGIC);

```

```

end alu_4bit_cla_nz;

```

```

ARCHITECTURE SCHEMATIC OF alu_4bit_cla_nz IS

```

```

    SIGNAL ALU_bit0_g_out :      STD_LOGIC;
    SIGNAL ALU_bit0_p_out :      STD_LOGIC;
    SIGNAL ALU_bit1_g_out :      STD_LOGIC;
    SIGNAL ALU_bit1_p_out :      STD_LOGIC;
    SIGNAL ALU_bit2_g_out :      STD_LOGIC;
    SIGNAL ALU_bit2_p_out :      STD_LOGIC;
    SIGNAL ALU_bit3_g_out :      STD_LOGIC;
    SIGNAL ALU_bit3_p_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c1_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c2_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c3_out :      STD_LOGIC;

```

```

    ATTRIBUTE BOX_TYPE : STRING;

```

```

    COMPONENT alu_1bit_xor_gp

```

```

        PORT ( a          :      IN      STD_LOGIC;
              b          :      IN      STD_LOGIC;
              binvert     :      IN      STD_LOGIC;
              carryin     :      IN      STD_LOGIC;
              enable      :      IN      STD_LOGIC;
              less        :      IN      STD_LOGIC;
              operation0   :      IN      STD_LOGIC;
              operation1   :      IN      STD_LOGIC;
              result      :      OUT     STD_LOGIC;
              g           :      OUT     STD_LOGIC;
              p           :      OUT     STD_LOGIC);

```

```

    END COMPONENT;

```

```

    COMPONENT cla_4bit_vhd

```

```

        PORT ( c0      :      IN      STD_LOGIC;
              g0       :      IN      STD_LOGIC;
              g1       :      IN      STD_LOGIC;
              g2       :      IN      STD_LOGIC;
              g3       :      IN      STD_LOGIC;
              p0       :      IN      STD_LOGIC;
              p1       :      IN      STD_LOGIC;
              p2       :      IN      STD_LOGIC;
              p3       :      IN      STD_LOGIC;

```

```

        c1      :      OUT      STD_LOGIC;
        c2      :      OUT      STD_LOGIC;
        c3      :      OUT      STD_LOGIC;
        sg      :      OUT      STD_LOGIC;
        sp      :      OUT      STD_LOGIC;
END COMPONENT;

BEGIN

    ALU_bit0 : alu_1bit_xor_gp
        PORT MAP (a=>a0, b=>b0, binvert=>ALUOperation2, carryin=>CarryIn,
            enable=>Mux_Enable, less=>Less_bit0, operation0=>ALUOperation0,
            operation1=>ALUOperation1, result=>Result0, g=>ALU_bit0_g_out,
            p=>ALU_bit0_p_out);

    ALU_bit1 : alu_1bit_xor_gp
        PORT MAP (a=>a1, b=>b1, binvert=>ALUOperation2, carryin=>CLA_4bit_c1_out,
            enable=>Mux_Enable, less=>Less_bit1, operation0=>ALUOperation0,
            operation1=>ALUOperation1, result=>Result1, g=>ALU_bit1_g_out,
            p=>ALU_bit1_p_out);

    ALU_bit2 : alu_1bit_xor_gp
        PORT MAP (a=>a2, b=>b2, binvert=>ALUOperation2, carryin=>CLA_4bit_c2_out,
            enable=>Mux_Enable, less=>Less_bit2, operation0=>ALUOperation0,
            operation1=>ALUOperation1, result=>Result2, g=>ALU_bit2_g_out,
            p=>ALU_bit2_p_out);

    ALU_bit3 : alu_1bit_xor_gp
        PORT MAP (a=>a3, b=>b3, binvert=>ALUOperation2, carryin=>CLA_4bit_c3_out,
            enable=>Mux_Enable, less=>Less_bit3, operation0=>ALUOperation0,
            operation1=>ALUOperation1, result=>Result3, g=>ALU_bit3_g_out,
            p=>ALU_bit3_p_out);

    CLA_4bit : cla_4bit_vhd
        PORT MAP (c0=>CarryIn, g0=>ALU_bit0_g_out, g1=>ALU_bit1_g_out,
            g2=>ALU_bit2_g_out, g3=>ALU_bit3_g_out, p0=>ALU_bit0_p_out,
            p1=>ALU_bit1_p_out, p2=>ALU_bit2_p_out, p3=>ALU_bit3_p_out,
            c1=>CLA_4bit_c1_out, c2=>CLA_4bit_c2_out, c3=>CLA_4bit_c3_out, sg=>sg,
            sp=>sp);

END SCHEMATIC;

```

➤ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 4-bit ALU using CLA, was generated. Figure A.81 shows the resulting top level RTL symbol for the synthesized 4-bit ALU using CLA while figure A.82 shows the resulting top level RTL schematic diagram. However, there is

no need for delving into deeper levels of the hierarchy as these have already been covered previously in various preceding subsections.

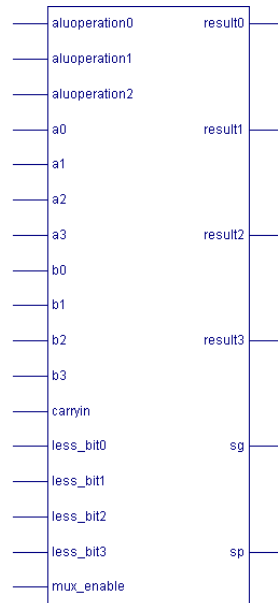


Figure A.81 Resulting top level RTL symbol for the 4-bit ALU using CLA.

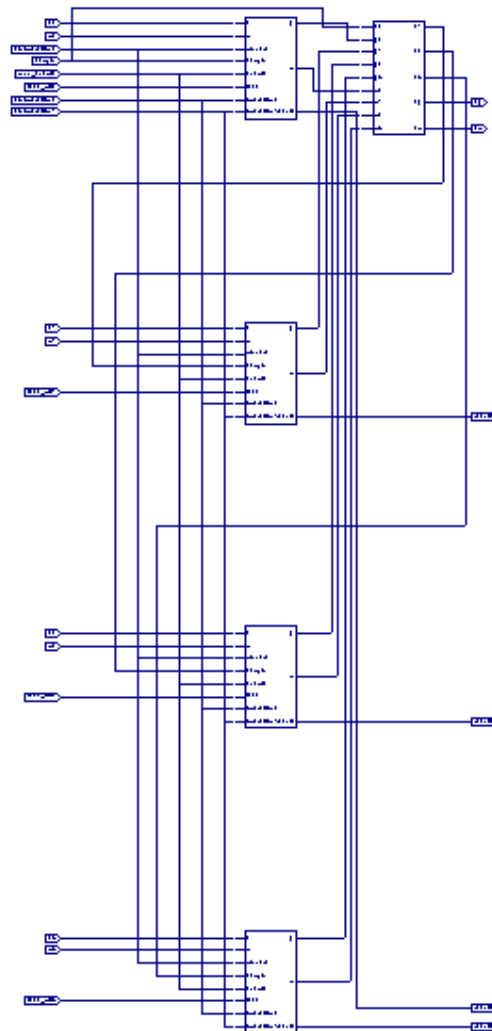


Figure A.82 Resulting top level RTL schematic for the 4-bit ALU using CLA.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 4-bit ALU with CLA using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

I/Os : 23

Cell Usage:

BELS : 78

and2 : 12

and2b1 : 4

```

#      and3                : 8
#      and3b1              : 8
#      inv                 : 4
#      LUT1                : 8
#      LUT3                : 3
#      LUT4                : 3
#      muxf5               : 4
#      or2                 : 20
#      xor3                : 4
# IO Buffers               : 23
#      IBUF                : 17
#      OBUF                : 6

```

Device utilization summary:

```

Number of Slices:                7 out of 46592    0%
Number of 4 input LUTs:          14 out of 93184   0%
Number of bonded IOBs:           23 out of 1108    2%

```

➤ *Place-and-Route onto the FPGA*

In figure A.83, FPGA Editor shows the synthesized 4-bit ALU with CLA after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running across the FPGA chip.

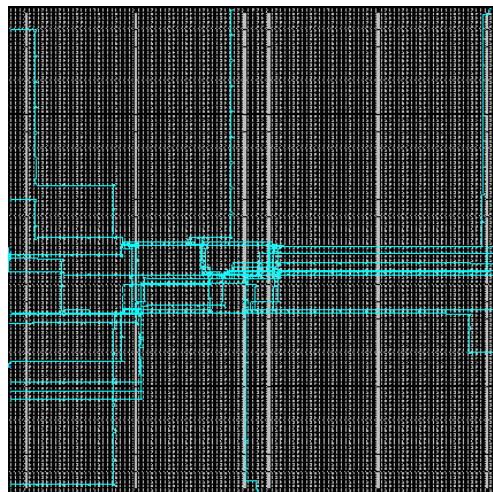


Figure A.83 *FPGA Editor showing the synthesized 4-bit ALU with CLA after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Simulating and testing a 4-bit ALU with CLA unit on its own does not provide much insight into the correctness of its functionality as opposed to simulating and testing it as part of the final 32-bit ALU block that implements CLA. Therefore, there is no need for this individual simulation to be performed at this stage.

Final 16-Bit ALU using Carry Lookahead

➤ *RTL Description*

Figures A.56 to A.58 have previously elaborated on the RTL description for the 16-bit ALU using CLA, which is also described in [47] and [4].

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the 16-bit ALU using CLA, which was shown in figures A.56, A.57 and A.58. The final schematic diagram is shown in figure A.84. Figures A.85 to A.88 show magnifications of different sections of the schematic diagram in figure A.84.

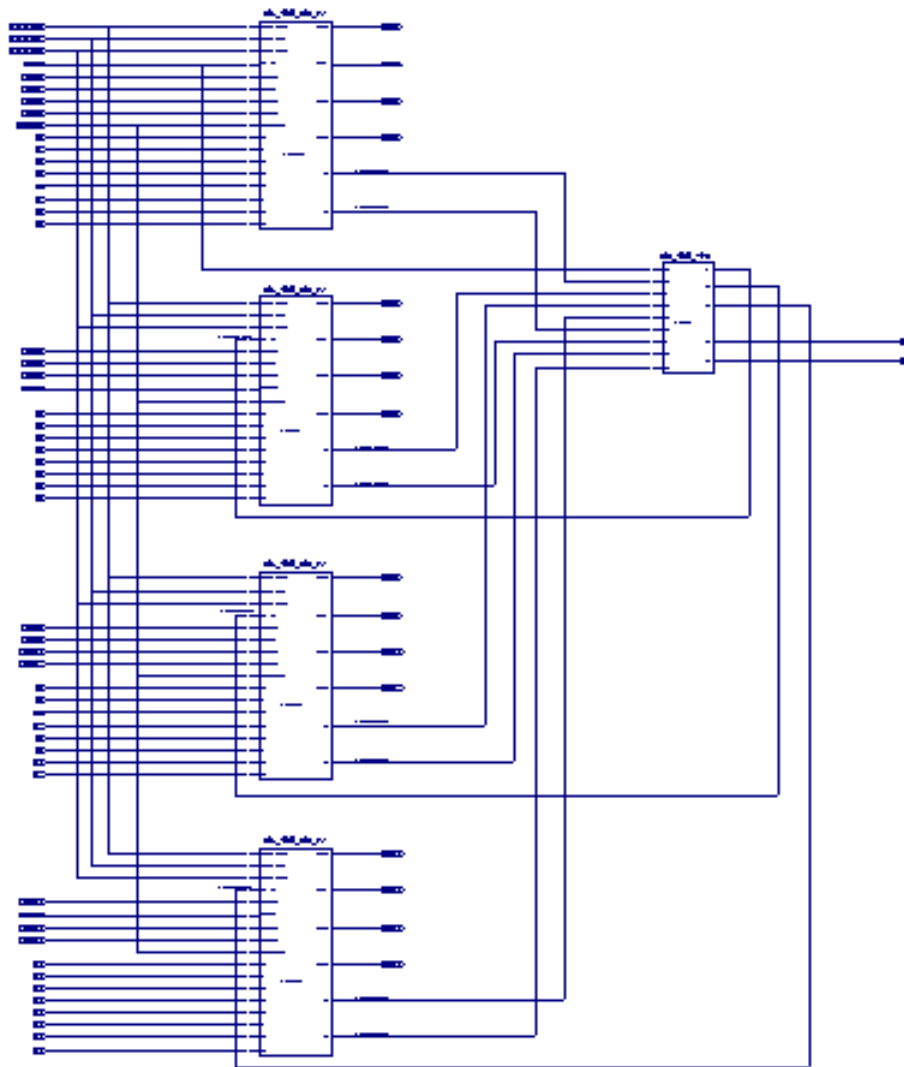


Figure A.84 Schematic diagram design entry for the 16-bit ALU with CLA in Schematic Editor.

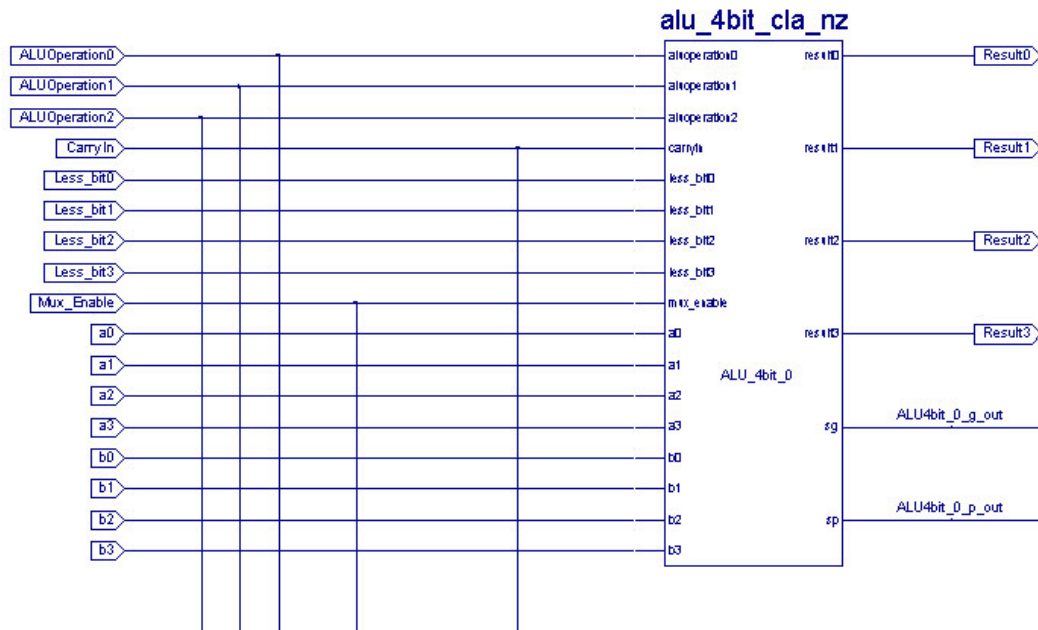


Figure A.85 Magnification of a section of figure A.84 showing ALU_4bit_0 in the schematic diagram for the 16-bit ALU with CLA.

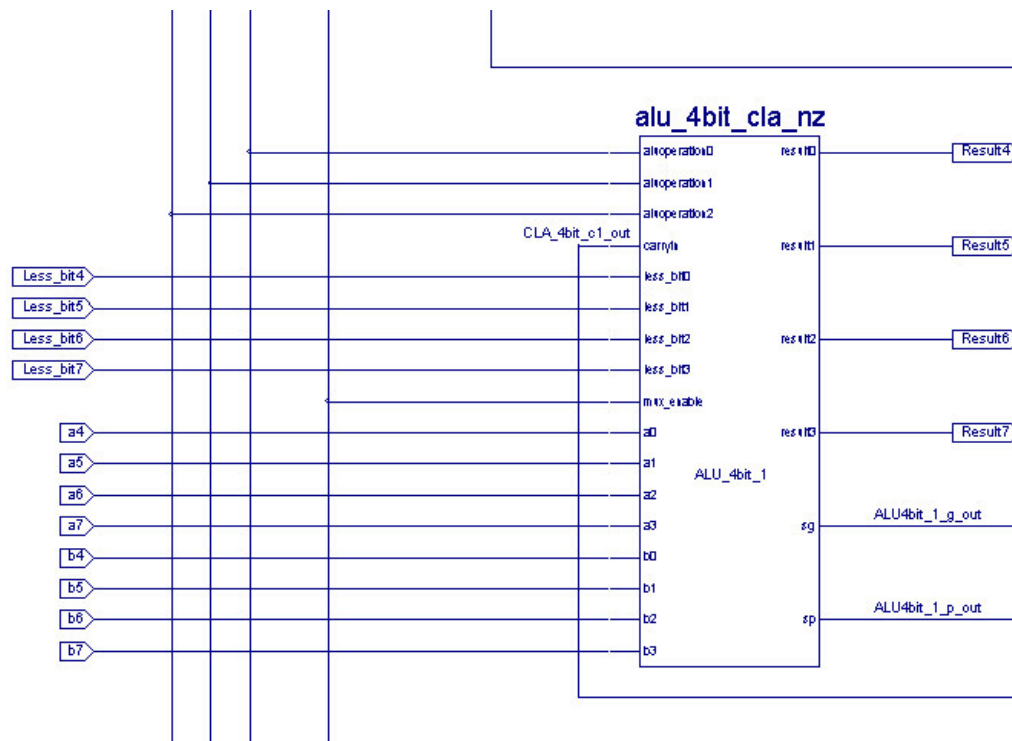


Figure A.86 Magnification of a section of figure A.84 showing ALU_4bit_1 in the schematic diagram for the 16-bit ALU with CLA.

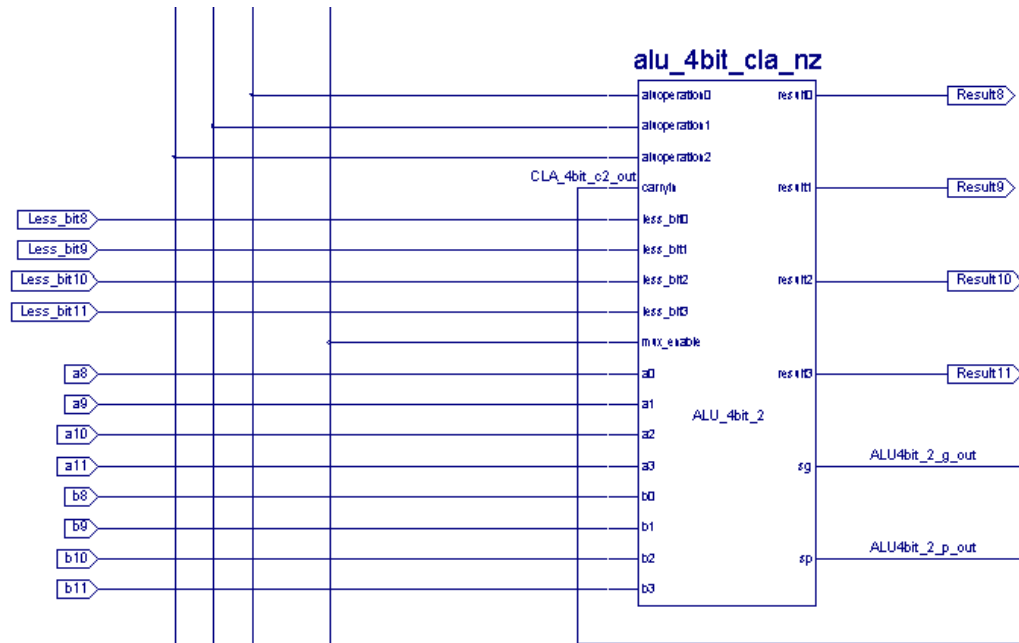


Figure A.87 Magnification of a section of figure A.84 showing ALU_4bit_2 in the schematic diagram for the 16-bit ALU with CLA.

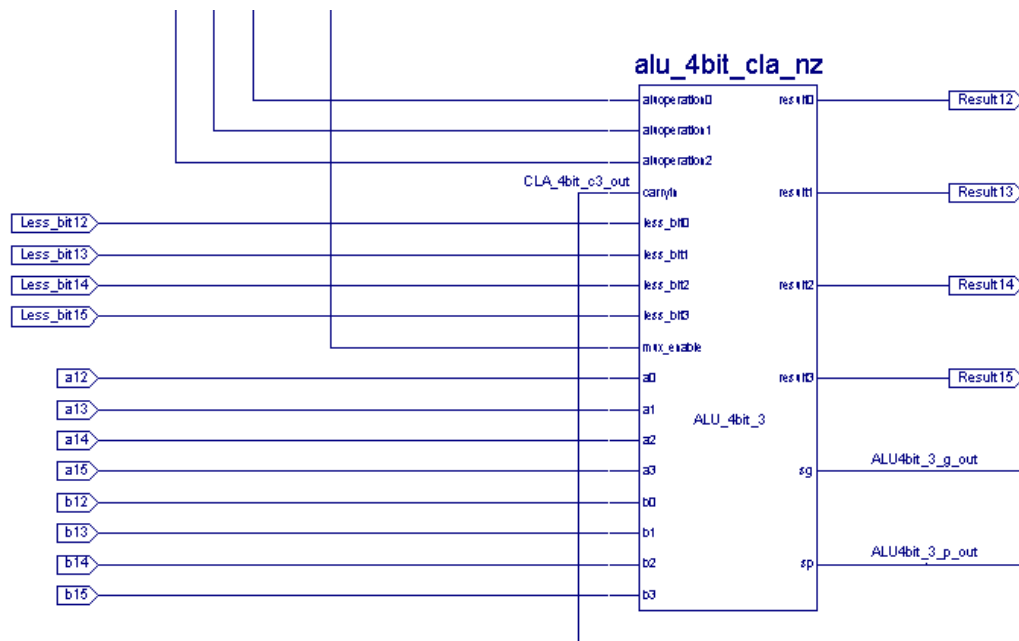


Figure A.88 Magnification of a section of figure A.84 showing ALU_4bit_3 in the schematic diagram for the 16-bit ALU with CLA.

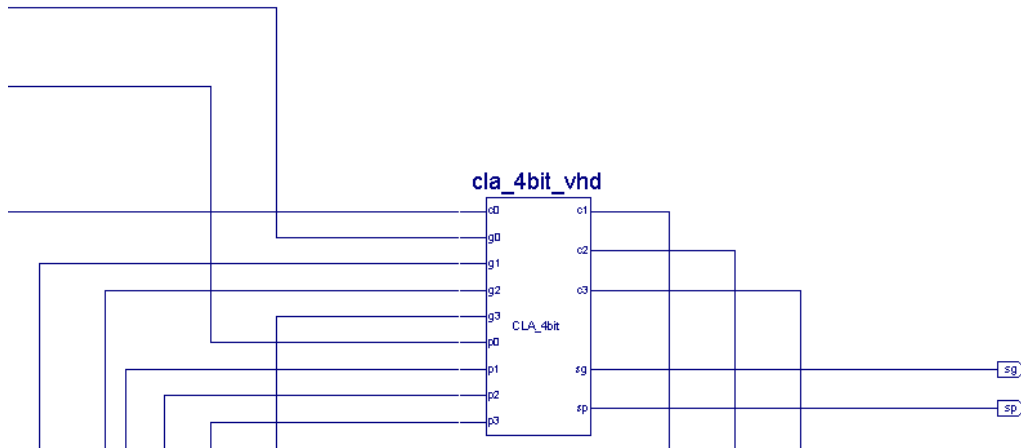


Figure A.88 Magnification of a section of figure A.84 showing CLA_4bit in the schematic diagram for the 16-bit ALU with CLA.

After synthesis of the schematic diagram in figure A.84 using XST, the resulting VHDL code shown below was generated.

- Vhdl model created from schematic alu_16bit_cla_nz_sch.sch - Tue May 06 10:16:19 2003

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY alu_16bit_cla_nz IS
    PORT (
        ALUOperation0 : IN STD_LOGIC;
        ALUOperation1 : IN STD_LOGIC;
        ALUOperation2 : IN STD_LOGIC;
        CarryIn       : IN STD_LOGIC;
        Less_bit0      : IN STD_LOGIC;
        Less_bit1      : IN STD_LOGIC;
        Less_bit15     : IN STD_LOGIC;
        Mux_Enable     : IN STD_LOGIC;
        a0             : IN STD_LOGIC;
        a1             : IN STD_LOGIC;
        a15            : IN STD_LOGIC;
        b0             : IN STD_LOGIC;
        b1             : IN STD_LOGIC;
        b15            : IN STD_LOGIC;
    );

```



```

        Result0      :      OUT      STD_LOGIC;
        Result1      :      OUT      STD_LOGIC;
        :
        Result15     :      OUT      STD_LOGIC;
        sg           :      OUT      STD_LOGIC;
        sp           :      OUT      STD_LOGIC);

end alu_16bit_cla_nz;

ARCHITECTURE SCHEMATIC OF alu_16bit_cla_nz IS

    SIGNAL ALU4bit_0_g_out :      STD_LOGIC;
    SIGNAL ALU4bit_0_p_out :      STD_LOGIC;
    SIGNAL ALU4bit_1_g_out :      STD_LOGIC;
    SIGNAL ALU4bit_1_p_out :      STD_LOGIC;
    SIGNAL ALU4bit_2_g_out :      STD_LOGIC;
    SIGNAL ALU4bit_2_p_out :      STD_LOGIC;
    SIGNAL ALU4bit_3_g_out :      STD_LOGIC;
    SIGNAL ALU4bit_3_p_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c1_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c2_out :      STD_LOGIC;
    SIGNAL CLA_4bit_c3_out :      STD_LOGIC;

    ATTRIBUTE BOX_TYPE : STRING;

    COMPONENT alu_4bit_cla_nz
        PORT ( aluoperation0 :      IN      STD_LOGIC;
              aluoperation1 :      IN      STD_LOGIC;
              aluoperation2 :      IN      STD_LOGIC;
              carryin       :      IN      STD_LOGIC;
              less_bit0     :      IN      STD_LOGIC;
              less_bit1     :      IN      STD_LOGIC;
              less_bit2     :      IN      STD_LOGIC;
              less_bit3     :      IN      STD_LOGIC;
              mux_enable     :      IN      STD_LOGIC;
              a0            :      IN      STD_LOGIC;
              a1            :      IN      STD_LOGIC;
              a2            :      IN      STD_LOGIC;
              a3            :      IN      STD_LOGIC;
              b0            :      IN      STD_LOGIC;
              b1            :      IN      STD_LOGIC;
              b2            :      IN      STD_LOGIC;
              b3            :      IN      STD_LOGIC;
              result0       :      OUT      STD_LOGIC;
              result1       :      OUT      STD_LOGIC;
              result2       :      OUT      STD_LOGIC;
              result3       :      OUT      STD_LOGIC;
              sg            :      OUT      STD_LOGIC;
              sp            :      OUT      STD_LOGIC);

    END COMPONENT;

```

```

COMPONENT cla_4bit_vhd
    PORT ( c0      :      IN      STD_LOGIC;
           g0      :      IN      STD_LOGIC;
           g1      :      IN      STD_LOGIC;
           g2      :      IN      STD_LOGIC;
           g3      :      IN      STD_LOGIC;
           p0      :      IN      STD_LOGIC;
           p1      :      IN      STD_LOGIC;
           p2      :      IN      STD_LOGIC;
           p3      :      IN      STD_LOGIC;
           c1      :      OUT     STD_LOGIC;
           c2      :      OUT     STD_LOGIC;
           c3      :      OUT     STD_LOGIC;
           sg      :      OUT     STD_LOGIC;
           sp      :      OUT     STD_LOGIC);
END COMPONENT;

BEGIN

ALU_4bit_0 : alu_4bit_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
              aluoperation2=>ALUOperation2, carryin=>CarryIn, less_bit0=>Less_bit0,
              less_bit1=>Less_bit1, less_bit2=>Less_bit2, less_bit3=>Less_bit3,
              mux_enable=>Mux_Enable, a0=>a0, a1=>a1, a2=>a2, a3=>a3, b0=>b0, b1=>b1,
              b2=>b2, b3=>b3, result0=>Result0, result1=>Result1, result2=>Result2,
              result3=>Result3, sg=>ALU4bit_0_g_out, sp=>ALU4bit_0_p_out);

ALU_4bit_1 : alu_4bit_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
              aluoperation2=>ALUOperation2, carryin=>CLA_4bit_c1_out,
              less_bit0=>Less_bit4, less_bit1=>Less_bit5, less_bit2=>Less_bit6,
              less_bit3=>Less_bit7, mux_enable=>Mux_Enable, a0=>a4, a1=>a5, a2=>a6,
              a3=>a7, b0=>b4, b1=>b5, b2=>b6, b3=>b7, result0=>Result4,
              result1=>Result5, result2=>Result6, result3=>Result7,
              sg=>ALU4bit_1_g_out, sp=>ALU4bit_1_p_out);

ALU_4bit_2 : alu_4bit_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
              aluoperation2=>ALUOperation2, carryin=>CLA_4bit_c2_out,
              less_bit0=>Less_bit8, less_bit1=>Less_bit9, less_bit2=>Less_bit10,
              less_bit3=>Less_bit11, mux_enable=>Mux_Enable, a0=>a8, a1=>a9, a2=>a10,
              a3=>a11, b0=>b8, b1=>b9, b2=>b10, b3=>b11, result0=>Result8,
              result1=>Result9, result2=>Result10, result3=>Result11,
              sg=>ALU4bit_2_g_out, sp=>ALU4bit_2_p_out);

ALU_4bit_3 : alu_4bit_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
              aluoperation2=>ALUOperation2, carryin=>CLA_4bit_c3_out,
              less_bit0=>Less_bit12, less_bit1=>Less_bit13, less_bit2=>Less_bit14,

```

```

less_bit3=>Less_bit15, mux_enable=>Mux_Enable, a0=>a12, a1=>a13, a2=>a14,
a3=>a15, b0=>b12, b1=>b13, b2=>b14, b3=>b15, result0=>Result12,
result1=>Result13, result2=>Result14, result3=>Result15,
sg=>ALU4bit_3_g_out, sp=>ALU4bit_3_p_out);

CLA_4bit : cla_4bit_vhd
PORT MAP (c0=>CarryIn, g0=>ALU4bit_0_g_out, g1=>ALU4bit_1_g_out,
g2=>ALU4bit_2_g_out, g3=>ALU4bit_3_g_out, p0=>ALU4bit_0_p_out,
p1=>ALU4bit_1_p_out, p2=>ALU4bit_2_p_out, p3=>ALU4bit_3_p_out,
c1=>CLA_4bit_c1_out, c2=>CLA_4bit_c2_out, c3=>CLA_4bit_c3_out, sg=>sg,
sp=>sp);

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 16-bit ALU using CLA, was generated. Figure A.89 shows the resulting top level RTL symbol for the synthesized 16-bit ALU using CLA while figure A.90 shows a magnification of the resulting top level RTL symbol. Figure A.91 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these have already been covered previously in various preceding subsections.

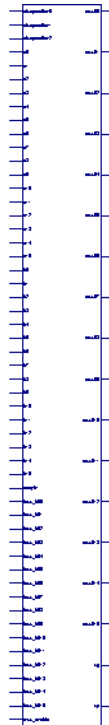


Figure A.89 Resulting top level RTL symbol for the 16-bit ALU using CLA.

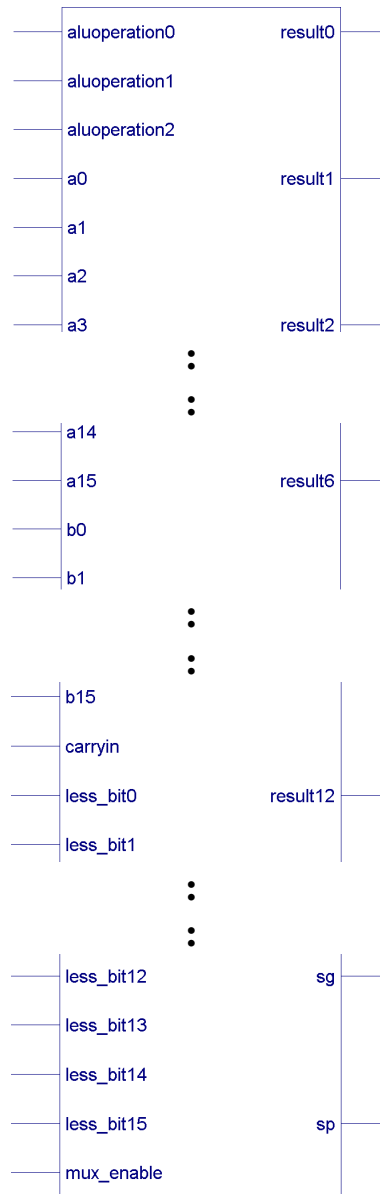


Figure A.90 Magnification of the resulting top level RTL symbol of figure A.89 for the 16-bit ALU using CLA.

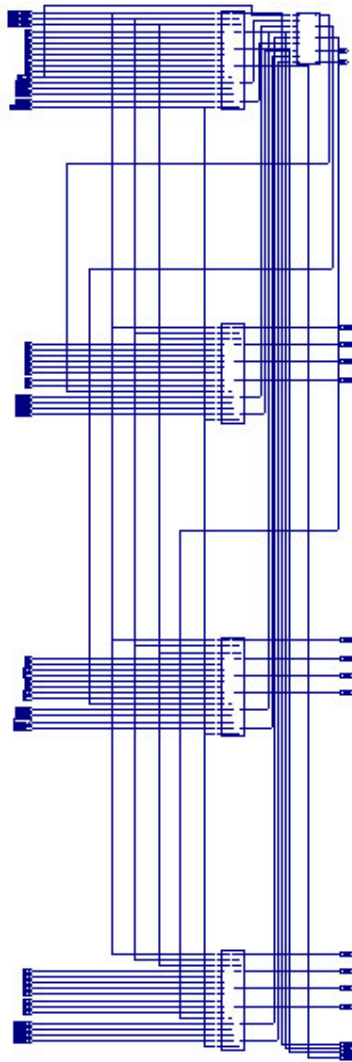


Figure A.91 Resulting top level RTL schematic for the 16-bit ALU using CLA.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the above 16-bit ALU with CLA using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```
Design Statistics
# IOs                      : 71

Cell Usage:
# BELS                     : 318
#      and2                 : 48
#      and2b1               : 16
```

```

#      and3                      : 32
#      and3b1                    : 32
#      inv                       : 16
#      LUT1                      : 32
#      LUT3                      : 15
#      LUT4                      : 15
#      muxf5                     : 16
#      or2                       : 80
#      xor3                      : 16
# IO Buffers                     : 71
#      IBUF                     : 53
#      OBUF                     : 18

```

Device utilization summary

```

Number of Slices:                33 out of 46592    0%
Number of 4 input LUTs:         62 out of 93184    0%
Number of bonded IOBs:         71 out of 1108     6%

```

➤ *Place-and-Route onto the FPGA*

In figure A.92, FPGA Editor shows the synthesized 16-bit ALU with CLA after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the middle and lower left section of the FPGA chip.

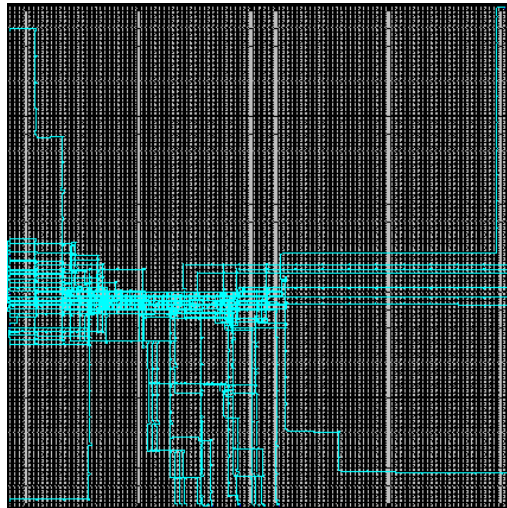


Figure A.92 *FPGA Editor showing the synthesized 16-bit ALU with CLA after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Simulating and testing a 16-bit ALU with CLA unit on its own does not provide much insight into the correctness of its functionality as opposed to simulating and testing it as part of the final 32-bit ALU block that implements CLA. Therefore, there is no need for this individual simulation to be performed at this stage.

Modified 1-Bit ALU for MSB with Propagate and Generate

➤ *RTL Description*

Figure A.59 shows the specially modified 1-bit ALU with *propagate* and *generate* signals for the most-significant bit (of the final 32-bit MIPS ALU with CLA).

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the modified 1-bit ALU for MSB with *propagate* and *generate* signals. The final schematic diagram is shown in figure A.93. This is very similar to the modified 1-bit ALU with *propagate* and *generate* signals (for the first 31 ALUs in a 32-bit ALU with CLA) shown in figure A.60 except for the fact that the *Sum* signal which is the output from the XOR gate is also provided as an output port. Also, neither the *Overflow* nor the *SetLessThan* signals are generated within this MSB ALU. They are generated outside (when putting together the final 32-bit MIPS ALU with CLA). That is why the *Sum* signal is also provided as an output port.

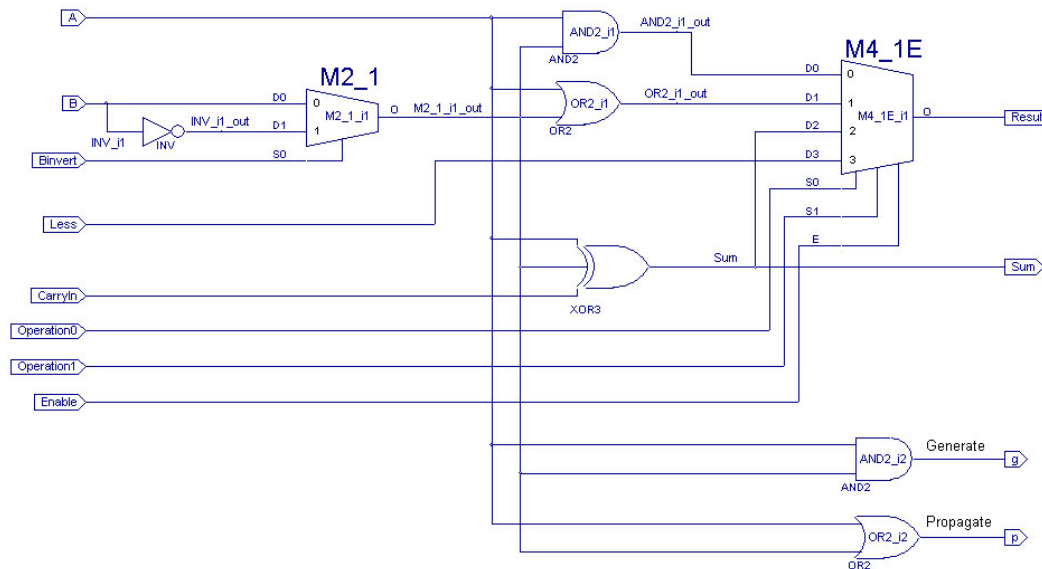


Figure A.93 Schematic diagram design entry for the modified 1-bit ALU for MSB with propagate and generate signals (instead of a full adder) in Schematic Editor.

➤ Simulation Results

No simulation has been carried out for this component as it is similar (except for the *sum*, *generate* and *propagate* output signals) to the previous ALUs synthesized and simulated. However, the final 32-bit ALU using carry lookahead will definitely be simulated and tested thoroughly.

4-Bit ALU for MSB using Carry Lookahead

➤ RTL Description

Figure A.59 shows the design hierarchy and abstraction for a 4-bit ALU for MSB using carry lookahead. It is similar to the 4-bit ALU with CLA (described previously) except for the fact that it contains the ALU for the MSB.

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the 4-bit ALU for MSB using CLA, which was shown in figures A.57 and A.59. The final schematic diagram is shown in figure A.94 while figure A.95 shows a magnified section of the schematic highlighting the 1-bit ALU for MSB.

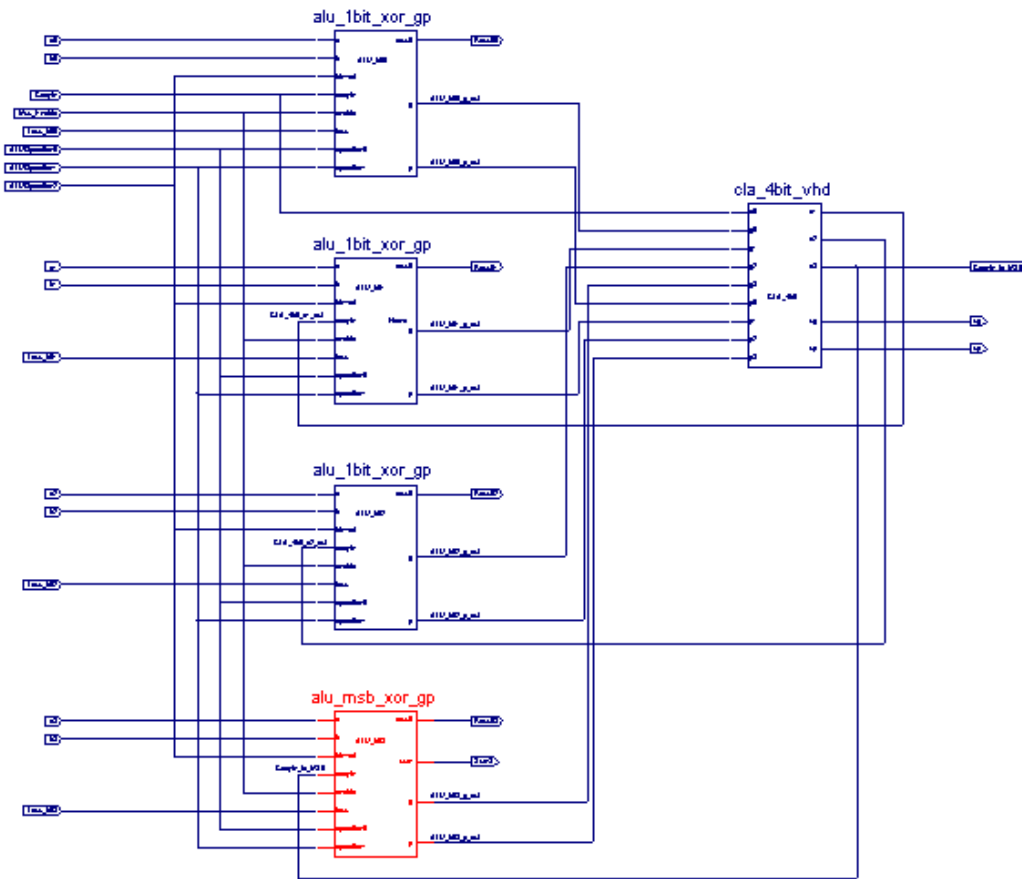


Figure A.94 Schematic diagram design entry for the 4-bit ALU for MSB with CLA in Schematic Editor.

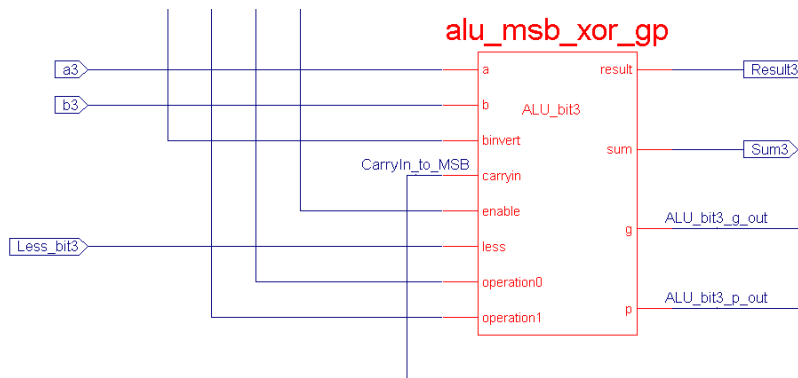


Figure A.95 Magnification of a section of figure A.94 (highlighted in red) showing ALU_bit3 for MSB in the schematic diagram for the 4-bit ALU for MSB with CLA.

➤ *Simulation Results*

Simulating and testing a 4-bit ALU for MSB with CLA unit on its own does not provide much insight into the correctness of its functionality as opposed to simulating and testing it as part of the final 32-bit ALU block that implements CLA. Therefore, there is no need for this individual simulation to be performed at this stage.

Final 16-Bit ALU for MSB using Carry Lookahead

➤ *RTL Description*

Figure A.57 shows the placement of the 16-bit ALU for MSB with CLA in the design hierarchy of the final 32-bit MIPS ALU with CLA.

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the final 16-bit ALU for MSB using CLA, which was shown in figure A.59. The final schematic diagram is shown in figure A.96 while figure A.97 shows a magnified section of the schematic highlighting the 4-bit ALU for MSB.

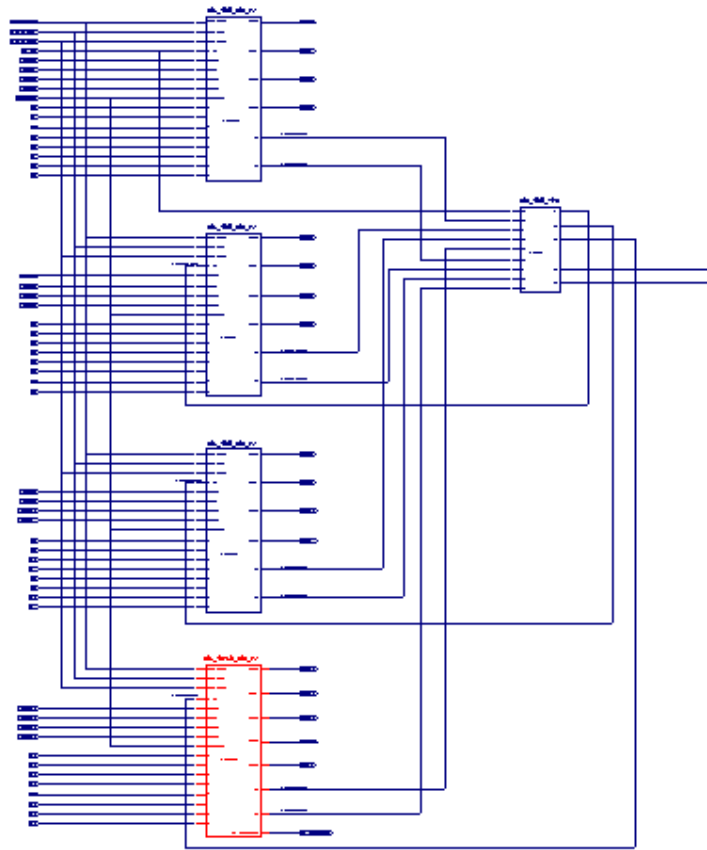


Figure A.96 Schematic diagram design entry for the final 16-bit ALU for MSB with CLA in Schematic Editor.

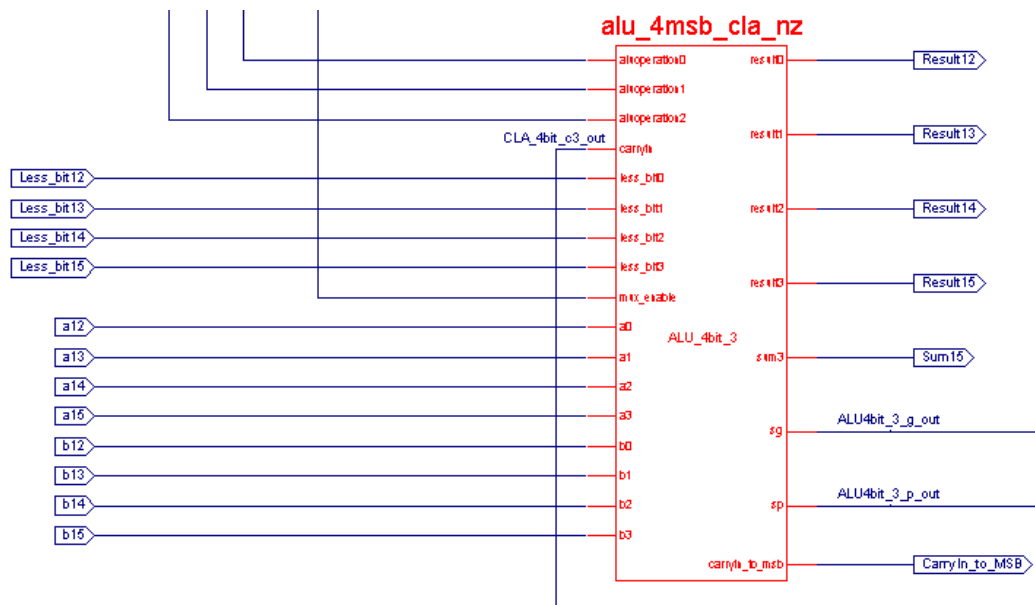


Figure A.97 Magnification of a section of figure A.96 (highlighted in red) showing ALU_4bit_3 for MSB in the schematic diagram for the final 16-bit ALU for MSB with CLA.

➤ *Simulation Results*

Simulating and testing the final 16-bit ALU for MSB with CLA unit on its own does not provide much insight into the correctness of its functionality as opposed to simulating and testing it as part of the final 32-bit ALU block that implements CLA. Therefore, there is no need for this individual simulation to be performed at this stage.

Final 32-Bit MIPS ALU using Carry Lookahead

➤ *RTL Description*

This is the finalized ALU used in building the full RTL model for the MIPS processor in this research. Figure A.57 is an elaboration of the design hierarchy and abstraction for this ALU.

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the final 32-bit MIPS ALU with CLA, which was shown in figure A.57. The final schematic diagram is shown in figure A.98 while figures A.99 to A.103 show the different magnified sections of this schematic.

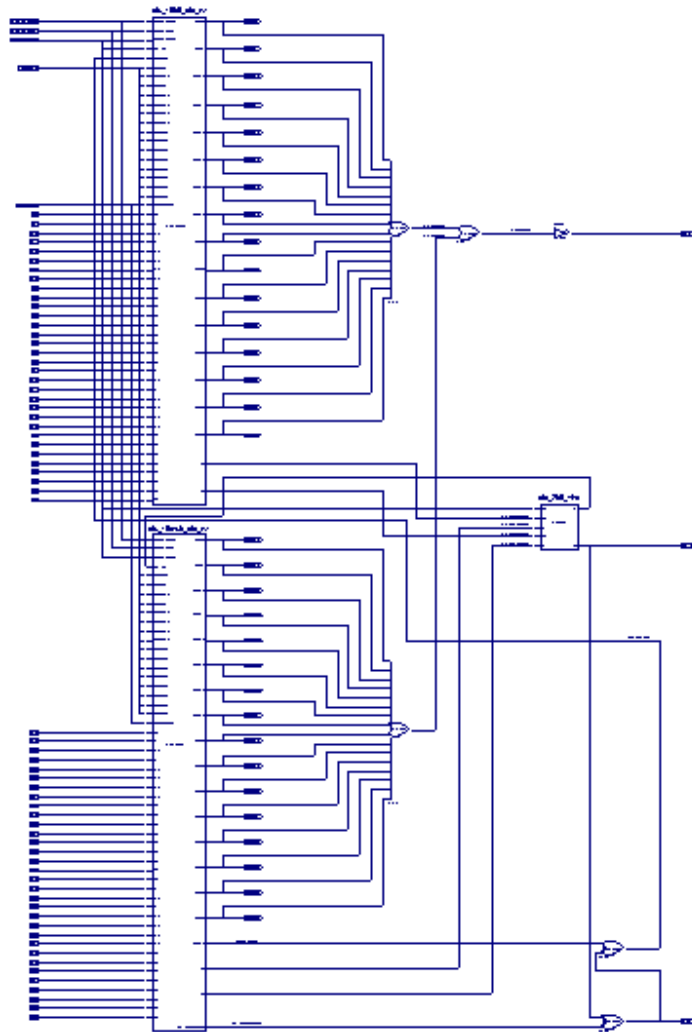


Figure A.98 Schematic diagram design entry for the final 32-bit MIPS ALU with CLA in Schematic Editor.

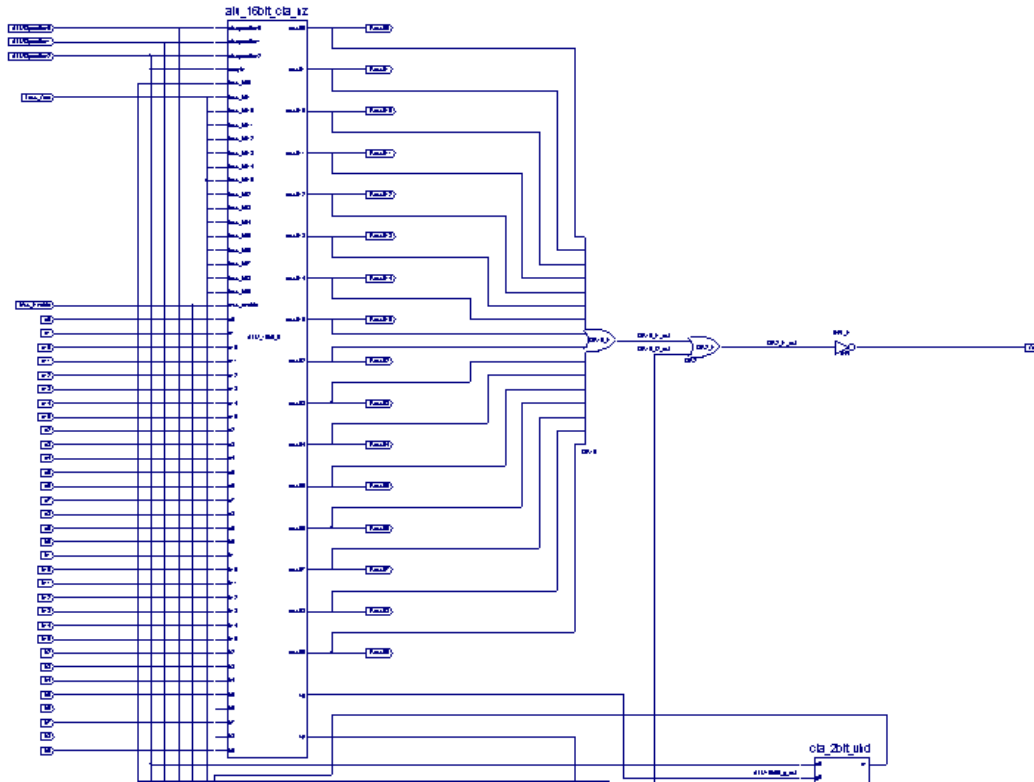


Figure A.99 Magnification of the top section of figure A.98 showing the first 16-bit ALU with CLA in the schematic diagram for the final 32-bit MIPS ALU with CLA.

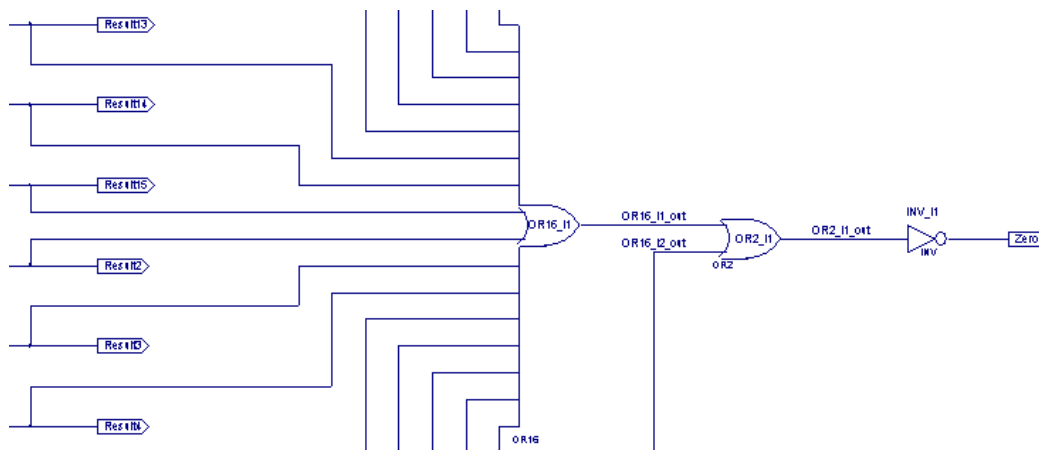
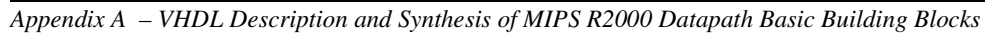
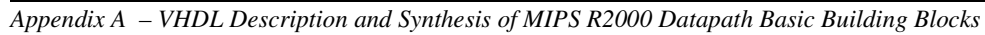


Figure A.100 Magnification of the right top section of figure A.98 showing the generation of the zero output signal in the schematic diagram for the final 32-bit MIPS ALU with CLA.



Appendix A – VHDL Description and Synthesis of MIPS R2000 Datapath Basic Building Blocks



Appendix A – VHDL Description and Synthesis of MIPS R2000 Datapath Basic Building Blocks

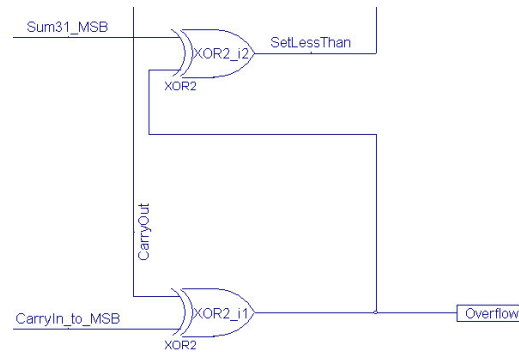


Figure A.103 Magnification of the right bottom section of figure A.98 showing the generation of the *SetLessThan* internal signal and the *Overflow* output signal in the schematic diagram for the final 32-bit MIPS ALU with CLA.

After synthesis of the schematic diagram in figure A.98 using XST, the resulting VHDL code shown below was generated.

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\or16.sch - Fri Oct
24 12:36:29 2003
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.vcomponents.ALL;
-- synopsys translate_on

ENTITY OR16_MXILINX_alu_32bit_cla_sch IS
    PORT ( I0      :      IN      STD_LOGIC;
           :
           I15     :      IN      STD_LOGIC;
           O       :      OUT     STD_LOGIC);

end OR16_MXILINX_alu_32bit_cla_sch;

ARCHITECTURE SCHEMATIC OF OR16_MXILINX_alu_32bit_cla_sch IS
    SIGNAL C0      :      STD_LOGIC;
    SIGNAL C1      :      STD_LOGIC;
    SIGNAL C2      :      STD_LOGIC;
    SIGNAL CIN     :      STD_LOGIC;
    SIGNAL S0      :      STD_LOGIC;
    SIGNAL S1      :      STD_LOGIC;
    SIGNAL S2      :      STD_LOGIC;
    SIGNAL S3      :      STD_LOGIC;
    SIGNAL XLXN_46 :      STD_LOGIC;

    ATTRIBUTE BOX_TYPE : STRING;
```



```

ATTRIBUTE RLOC : STRING ;
ATTRIBUTE RLOC OF I_36_170 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_142 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_138 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_29 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_165 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_147 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_129 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_2 : LABEL IS "X0Y0";

COMPONENT FMAP
    PORT ( I1      :      IN      STD_LOGIC;
           I2      :      IN      STD_LOGIC;
           I3      :      IN      STD_LOGIC;
           I4      :      IN      STD_LOGIC;
           O       :      IN      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF FMAP : COMPONENT IS "BLACK_BOX";
COMPONENT GND
    PORT ( G       :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF GND : COMPONENT IS "BLACK_BOX";
COMPONENT MUXCY
    PORT ( CI      :      IN      STD_LOGIC;
           DI      :      IN      STD_LOGIC;
           S       :      IN      STD_LOGIC;
           O       :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY : COMPONENT IS "BLACK_BOX";
COMPONENT MUXCY_L
    PORT ( CI      :      IN      STD_LOGIC;
           DI      :      IN      STD_LOGIC;
           S       :      IN      STD_LOGIC;
           LO      :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY_L : COMPONENT IS "BLACK_BOX";
COMPONENT NOR4
    PORT ( I0      :      IN      STD_LOGIC;
           I1      :      IN      STD_LOGIC;
           I2      :      IN      STD_LOGIC;
           I3      :      IN      STD_LOGIC;
           O       :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF NOR4 : COMPONENT IS "BLACK_BOX";

```

```

COMPONENT VCC
    PORT ( P      :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF VCC : COMPONENT IS "BLACK_BOX";
BEGIN

I_36_170 : FMAP
    PORT MAP (I1=>I12, I2=>I13, I3=>I14, I4=>I15, O=>S3);

I_36_142 : FMAP
    PORT MAP (I1=>I8, I2=>I9, I3=>I10, I4=>I11, O=>S2);

I_36_138 : FMAP
    PORT MAP (I1=>I4, I2=>I5, I3=>I6, I4=>I7, O=>S1);

I_36_29 : FMAP
    PORT MAP (I1=>I0, I2=>I1, I3=>I2, I4=>I3, O=>S0);

I_36_174 : GND
    PORT MAP (G=>CIN);

I_36_165 : MUXCY
    PORT MAP (CI=>C2, DI=>XLXN_46, S=>S3, O=>O);

I_36_147 : MUXCY_L
    PORT MAP (CI=>C1, DI=>XLXN_46, S=>S2, LO=>C2);

I_36_129 : MUXCY_L
    PORT MAP (CI=>C0, DI=>XLXN_46, S=>S1, LO=>C1);

I_36_2 : MUXCY_L
    PORT MAP (CI=>CIN, DI=>XLXN_46, S=>S0, LO=>C0);

I_36_161 : NOR4
    PORT MAP (I0=>I12, I1=>I13, I2=>I14, I3=>I15, O=>S3);

I_36_151 : NOR4
    PORT MAP (I0=>I8, I1=>I9, I2=>I10, I3=>I11, O=>S2);

I_36_127 : NOR4
    PORT MAP (I0=>I4, I1=>I5, I2=>I6, I3=>I7, O=>S1);

I_36_110 : NOR4
    PORT MAP (I0=>I0, I1=>I1, I2=>I2, I3=>I3, O=>S0);

I_36_172 : VCC
    PORT MAP (P=>XLXN_46);

```

```

END SCHEMATIC;

-- Vhdl model created from schematic alu_32bit_cla_sch.sch - Fri Oct 24 12:36:29 2003

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY alu_32bit_cla_sch IS
    PORT ( ALUOperation0 : IN STD_LOGIC;
          ALUOperation1 : IN STD_LOGIC;
          ALUOperation2 : IN STD_LOGIC;
          Less_Zero      : IN STD_LOGIC;
          Mux_Enable     : IN STD_LOGIC;
          a0              : IN STD_LOGIC;
                  :      :
          a31            : IN STD_LOGIC;
          b0              : IN STD_LOGIC;
                  :      :
          b31            : IN STD_LOGIC;
          CarryOut        : OUT STD_LOGIC;
          Overflow        : OUT STD_LOGIC;
          Result0         : OUT STD_LOGIC;
                  :      :
          Result31       : OUT STD_LOGIC;
          Zero            : OUT STD_LOGIC);

end alu_32bit_cla_sch;

ARCHITECTURE SCHEMATIC OF alu_32bit_cla_sch IS
    SIGNAL ALU16bit0_g_out : STD_LOGIC;
    SIGNAL ALU16bit0_p_out : STD_LOGIC;
    SIGNAL ALU16bit1_g_out : STD_LOGIC;
    SIGNAL ALU16bit1_p_out : STD_LOGIC;
    SIGNAL CarryIn_to_MSB : STD_LOGIC;
    SIGNAL CarryOut_DUMMY : STD_LOGIC;
    SIGNAL OR16_i1_out     : STD_LOGIC;
    SIGNAL OR16_i2_out     : STD_LOGIC;
    SIGNAL OR2_i1_out      : STD_LOGIC;
    SIGNAL Overflow_DUMMY : STD_LOGIC;
    SIGNAL Result0_DUMMY  : STD_LOGIC;
        :
    SIGNAL Result31_DUMMY : STD_LOGIC;
    SIGNAL SetLessThan    : STD_LOGIC;
    SIGNAL Sum31_MSB      : STD_LOGIC;

```

```

SIGNAL XLXN_79 :      STD_LOGIC;

ATTRIBUTE BOX_TYPE : STRING;
ATTRIBUTE U_SET : STRING ;
ATTRIBUTE U_SET OF OR16_i1 : LABEL IS "OR16_i1_0";
ATTRIBUTE U_SET OF OR16_i2 : LABEL IS "OR16_i2_1";

COMPONENT alu_16bit_cla_nz
  PORT ( aluoperation0 :      IN      STD_LOGIC;
        aluoperation1 :      IN      STD_LOGIC;
        aluoperation2 :      IN      STD_LOGIC;
        carryin       :      IN      STD_LOGIC;
        less_bit0      :      IN      STD_LOGIC;
              :           :
        less_bit15     :      IN      STD_LOGIC;
        mux_enable     :      IN      STD_LOGIC;
        a0             :      IN      STD_LOGIC;
              :           :
        a15            :      IN      STD_LOGIC;
        b0             :      IN      STD_LOGIC;
              :           :
        b15            :      IN      STD_LOGIC;
        result0        :      OUT     STD_LOGIC;
              :           :
        result15       :      OUT     STD_LOGIC;
        sg             :      OUT     STD_LOGIC;
        sp             :      OUT     STD_LOGIC);
END COMPONENT;

COMPONENT alu_16msb_cla_nz
  PORT ( aluoperation0 :      IN      STD_LOGIC;
        aluoperation1 :      IN      STD_LOGIC;
        aluoperation2 :      IN      STD_LOGIC;
        carryin       :      IN      STD_LOGIC;
        less_bit0      :      IN      STD_LOGIC;
              :           :
        less_bit15     :      IN      STD_LOGIC;
        mux_enable     :      IN      STD_LOGIC;
        a0             :      IN      STD_LOGIC;
              :           :
        a15            :      IN      STD_LOGIC;
        b0             :      IN      STD_LOGIC;
              :           :
        b15            :      IN      STD_LOGIC;
        result0        :      OUT     STD_LOGIC;
              :           :
        result15       :      OUT     STD_LOGIC;
        sum15          :      OUT     STD_LOGIC;
        sg             :      OUT     STD_LOGIC);

```

```

        sp          :      OUT      STD_LOGIC;
        carryin_to_msb :      OUT      STD_LOGIC);
END COMPONENT;

COMPONENT cla_2bit_vhd
    PORT ( c0      :      IN      STD_LOGIC;
          g0      :      IN      STD_LOGIC;
          g1      :      IN      STD_LOGIC;
          p0      :      IN      STD_LOGIC;
          p1      :      IN      STD_LOGIC;
          c1      :      OUT      STD_LOGIC;
          c2      :      OUT      STD_LOGIC);
END COMPONENT;

COMPONENT INV
    PORT ( I      :      IN      STD_LOGIC;
          O      :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF INV : COMPONENT IS "BLACK_BOX";
COMPONENT OR16_MXILINX_alu_32bit_cla_sch
    PORT ( I0      :      IN      STD_LOGIC;
          :          :
          I15     :      IN      STD_LOGIC;
          O      :      OUT      STD_LOGIC);
END COMPONENT;

COMPONENT OR2
    PORT ( I0      :      IN      STD_LOGIC;
          I1      :      IN      STD_LOGIC;
          O      :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
COMPONENT XOR2
    PORT ( I0      :      IN      STD_LOGIC;
          I1      :      IN      STD_LOGIC;
          O      :      OUT      STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XOR2 : COMPONENT IS "BLACK_BOX";
BEGIN
    CarryOut <= CarryOut_DUMMY;
    Overflow <= Overflow_DUMMY;
    Result0 <= Result0_DUMMY;
    :
    Result31 <= Result31_DUMMY;

```

```

ALU_16bit_0 : alu_16bit_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
    aluoperation2=>ALUOperation2, carryin=>ALUOperation2,
    less_bit0=>SetLessThan, less_bit1=>Less_Zero, less_bit10=>Less_Zero,
    less_bit11=>Less_Zero, less_bit12=>Less_Zero, less_bit13=>Less_Zero,
    less_bit14=>Less_Zero, less_bit15=>Less_Zero, less_bit2=>Less_Zero,
    less_bit3=>Less_Zero, less_bit4=>Less_Zero, less_bit5=>Less_Zero,
    less_bit6=>Less_Zero, less_bit7=>Less_Zero, less_bit8=>Less_Zero,
    less_bit9=>Less_Zero, mux_enable=>Mux_Enable, a0=>a0, a1=>a1, a10=>a10,
    a11=>a11, a12=>a12, a13=>a13, a14=>a14, a15=>a15, a2=>a2, a3=>a3, a4=>a4,
    a5=>a5, a6=>a6, a7=>a7, a8=>a8, a9=>a9, b0=>b0, b1=>b1, b10=>b10,
    b11=>b11, b12=>b12, b13=>b13, b14=>b14, b15=>b15, b2=>b2, b3=>b3, b4=>b4,
    b5=>b5, b6=>b6, b7=>b7, b8=>b8, b9=>b9, result0=>Result0_DUMMY,
    result1=>Result1_DUMMY, result10=>Result10_DUMMY,
    result11=>Result11_DUMMY, result12=>Result12_DUMMY,
    result13=>Result13_DUMMY, result14=>Result14_DUMMY,
    result15=>Result15_DUMMY, result2=>Result2_DUMMY, result3=>Result3_DUMMY,
    result4=>Result4_DUMMY, result5=>Result5_DUMMY, result6=>Result6_DUMMY,
    result7=>Result7_DUMMY, result8=>Result8_DUMMY, result9=>Result9_DUMMY,
    sg=>ALU16bit0_g_out, sp=>ALU16bit0_p_out);

ALU_16bit_1 : alu_16msb_cla_nz
    PORT MAP (aluoperation0=>ALUOperation0, aluoperation1=>ALUOperation1,
    aluoperation2=>ALUOperation2, carryin=>XLXN_79, less_bit0=>Less_Zero,
    less_bit1=>Less_Zero, less_bit10=>Less_Zero, less_bit11=>Less_Zero,
    less_bit12=>Less_Zero, less_bit13=>Less_Zero, less_bit14=>Less_Zero,
    less_bit15=>Less_Zero, less_bit2=>Less_Zero, less_bit3=>Less_Zero,
    less_bit4=>Less_Zero, less_bit5=>Less_Zero, less_bit6=>Less_Zero,
    less_bit7=>Less_Zero, less_bit8=>Less_Zero, less_bit9=>Less_Zero,
    mux_enable=>Mux_Enable, a0=>a16, a1=>a17, a10=>a26, a11=>a27, a12=>a28,
    a13=>a29, a14=>a30, a15=>a31, a2=>a18, a3=>a19, a4=>a20, a5=>a21,
    a6=>a22, a7=>a23, a8=>a24, a9=>a25, b0=>b16, b1=>b17, b10=>b26, b11=>b27,
    b12=>b28, b13=>b29, b14=>b30, b15=>b31, b2=>b18, b3=>b19, b4=>b20,
    b5=>b21, b6=>b22, b7=>b23, b8=>b24, b9=>b25, result0=>Result16_DUMMY,
    result1=>Result17_DUMMY, result10=>Result26_DUMMY,
    result11=>Result27_DUMMY, result12=>Result28_DUMMY,
    result13=>Result29_DUMMY, result14=>Result30_DUMMY,
    result15=>Result31_DUMMY, result2=>Result18_DUMMY,
    result3=>Result19_DUMMY, result4=>Result20_DUMMY,
    result5=>Result21_DUMMY, result6=>Result22_DUMMY,
    result7=>Result23_DUMMY, result8=>Result24_DUMMY,
    result9=>Result25_DUMMY, sum15=>Sum31_MSB, sg=>ALU16bit1_g_out,
    sp=>ALU16bit1_p_out, carryin_to_msb=>CarryIn_to_MSB);

CLA_2bit : cla_2bit_vhd
    PORT MAP (c0=>ALUOperation2, g0=>ALU16bit0_g_out, g1=>ALU16bit1_g_out,
    p0=>ALU16bit0_p_out, p1=>ALU16bit1_p_out, c1=>XLXN_79,
    c2=>CarryOut_DUMMY);

```

```

INV_i1 : INV
  PORT MAP (I=>OR2_i1_out, O=>Zero);

OR16_i1 : OR16_MXILINK_alu_32bit_cla_sch
  PORT MAP (I0=>Result9_DUMMY, I1=>Result8_DUMMY, I10=>Result13_DUMMY,
    I11=>Result12_DUMMY, I12=>Result11_DUMMY, I13=>Result10_DUMMY,
    I14=>Result1_DUMMY, I15=>Result0_DUMMY, I2=>Result7_DUMMY,
    I3=>Result6_DUMMY, I4=>Result5_DUMMY, I5=>Result4_DUMMY,
    I6=>Result3_DUMMY, I7=>Result2_DUMMY, I8=>Result15_DUMMY,
    I9=>Result14_DUMMY, O=>OR16_i1_out);

OR16_i2 : OR16_MXILINK_alu_32bit_cla_sch
  PORT MAP (I0=>Result25_DUMMY, I1=>Result24_DUMMY, I10=>Result29_DUMMY,
    I11=>Result28_DUMMY, I12=>Result27_DUMMY, I13=>Result26_DUMMY,
    I14=>Result17_DUMMY, I15=>Result16_DUMMY, I2=>Result23_DUMMY,
    I3=>Result22_DUMMY, I4=>Result21_DUMMY, I5=>Result20_DUMMY,
    I6=>Result19_DUMMY, I7=>Result18_DUMMY, I8=>Result31_DUMMY,
    I9=>Result30_DUMMY, O=>OR16_i2_out);

OR2_i1 : OR2
  PORT MAP (I0=>OR16_i2_out, I1=>OR16_i1_out, O=>OR2_i1_out);

XOR2_i1 : XOR2
  PORT MAP (I0=>CarryIn_to_MSB, I1=>CarryOut_DUMMY, O=>Overflow_DUMMY);

XOR2_i2 : XOR2
  PORT MAP (I0=>Overflow_DUMMY, I1=>Sum31_MSB, O=>SetLessThan);

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the final 32-bit MIPS ALU with CLA, was generated. Figure A.104 shows the resulting top level RTL symbol while figure A.105 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these have already been covered previously in various preceding subsections.

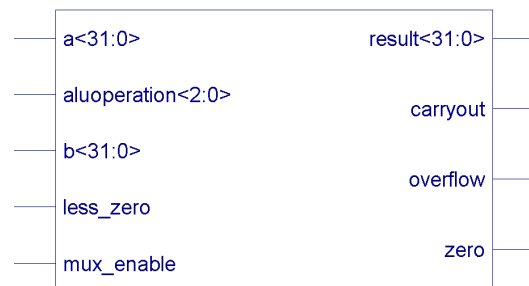


Figure A.104 Compact version of the resulting top level RTL symbol for the final 32-bit MIPS ALU using CLA.

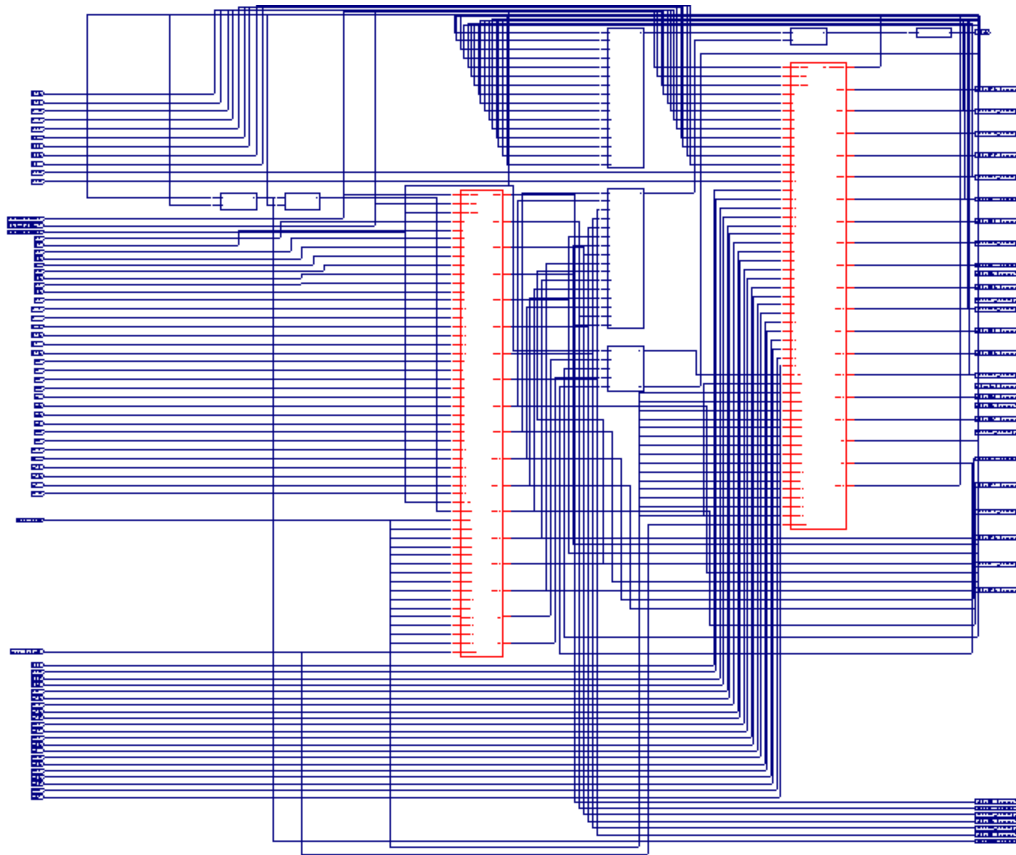


Figure A.105 Resulting top level RTL schematic for the final 32-bit MIPS ALU using CLA.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the final 32-bit MIPS ALU with CLA using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```
Design Statistics:
# I/Os                      : 104

Cell Usage:
# BELS                      : 654
#   and2                    : 96
#   and2b1                  : 32
#   and3                    : 64
#   and3b1                  : 64
#   gnd                     : 2
#   inv                     : 33
#   LUT1                    : 64
#   LUT3                    : 32
```



```

#      LUT4                      : 30
#      muxcy                     : 2
#      muxcy_1                   : 6
#      muxf5                     : 32
#      or2                       : 161
#      vcc                       : 2
#      xor2                      : 2
#      xor3                      : 32
# IO Buffers                     : 104
#      IBUF                     : 69
#      OBUF                     : 35
# Logical                       : 8
#      nor4                     : 8
# Others                        : 8
#      fmap                     : 8

```

Device utilization summary:

```

Number of Slices:                67 out of 46592    0%
Number of 4 input LUTs:         126 out of 93184    0%
Number of bonded IOBs:         104 out of 1108     9%

```

➤ Place-and-Route onto the FPGA

In figure A.106, FPGA Editor shows the synthesized final 32-bit MIPS ALU with CLA after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the middle upper section of the FPGA chip.

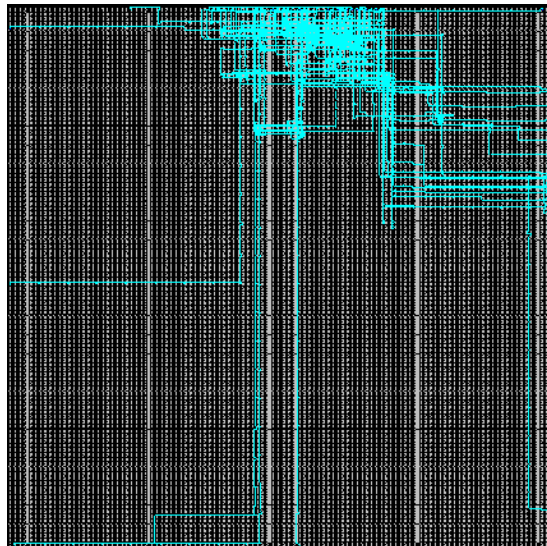


Figure A.106 FPGA Editor showing the synthesized final 32-bit MIPS ALU with CLA after place-and-route onto the target Virtex-II FPGA chip.

➤ *Simulation Results*

Figures A.107 to A.112 show the waveform results of simulating the VHDL behavioural model for the finalized 32-bit MIPS ALU with CLA in Mentor Graphics ModelSim. In both figures A.107 and A.108, all the waveforms are in binary format. However, in figures A.108 through to A.112, all the waveforms are in binary format except for the signals *a*, *b*, and *result* which are in signed decimal format. Also, figures A.50 to A.55 have been repeated here for the sake of comparing the output waveforms as will be discussed in the next subsection “**Final 32-Bit MIPS ALU: Ripple Carry versus Carry Lookahead**”.

When checking these waveforms, the following points are worth noting:

- ❑ The sets of input test signals are exactly the same as those applied previously to the final 32-bit MIPS ALU using ripple carry (figures A.50 to A.55). This facilitates the comparison of both ALUs to confirm that they both generate exactly the same outputs given the same inputs. The upcoming subsection “**Final 32-Bit MIPS ALU: Ripple Carry versus Carry Lookahead**” elaborates more on this point.
- ❑ These waveforms are compared against the specified behavior of the ALU as shown earlier in the function table of figure A.42.
- ❑ In each figure, the time delay is shown for every value of the output signal(s) in question. This information will be used shortly for comparison between the 32-bit ripple carry ALU and the carry lookahead one.

□ When the ALU Function is AND (figure A.107):

- The only output signal of interest is *result*.
- Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexer is disabled (input signal *mux_enable* = 0).

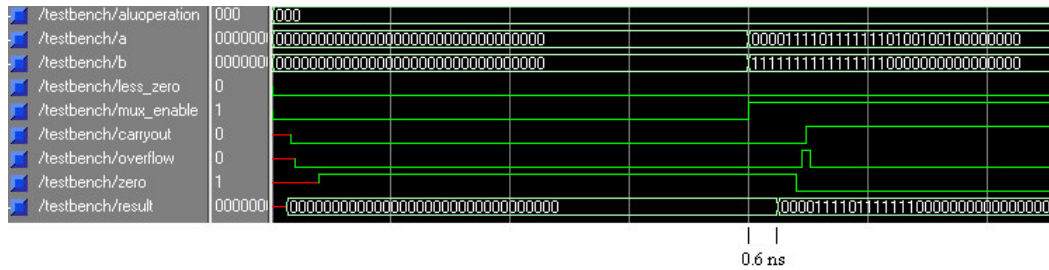


Figure A.50 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = AND.

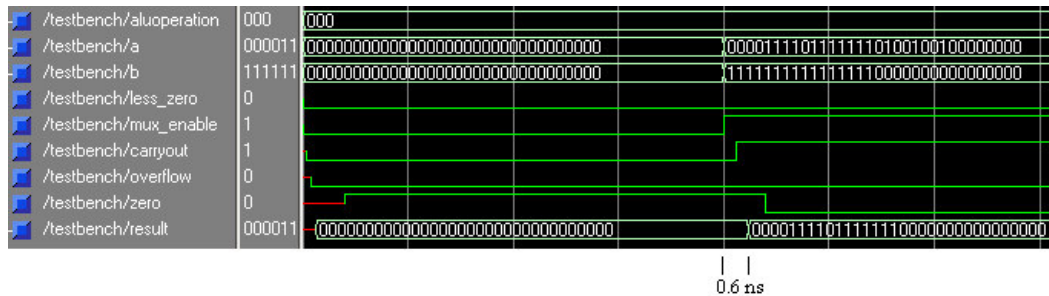


Figure A.107 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = AND.

□ When the ALU Function is OR (figure A.108):

- The only output signal of interest is *result*.
- Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexer is disabled (input signal *mux_enable* = 0).

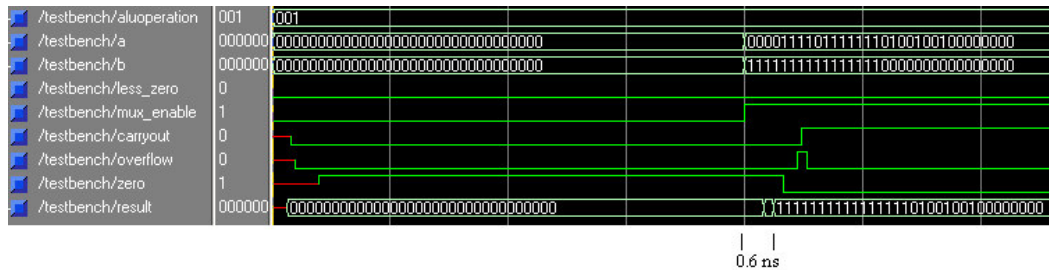


Figure A.51 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = OR.

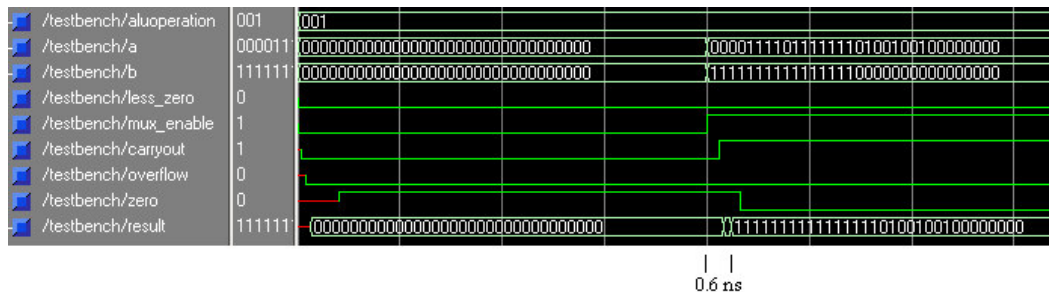


Figure A.108 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = OR.

- When the ALU Function is ADD (figure A.109):
 - Both output signals *carryout* and *result* are of interest.
 - Only for the first 10 ns time interval, the output signal *result* = 0. This is because the ALU internal multiplexor is disabled (input signal *mux_enable* = 0).
 - Due to the fact that this ALU is based on carry lookahead addition, the *result* output goes through a lesser number of unwanted intermediate values (compared to ripple carry addition) until settling at the correct value. In figure A.109, the output *result* is always settling down to the correct value after a fixed delay of 0.7 ns from the time the input signals *a* and *b* have been applied.

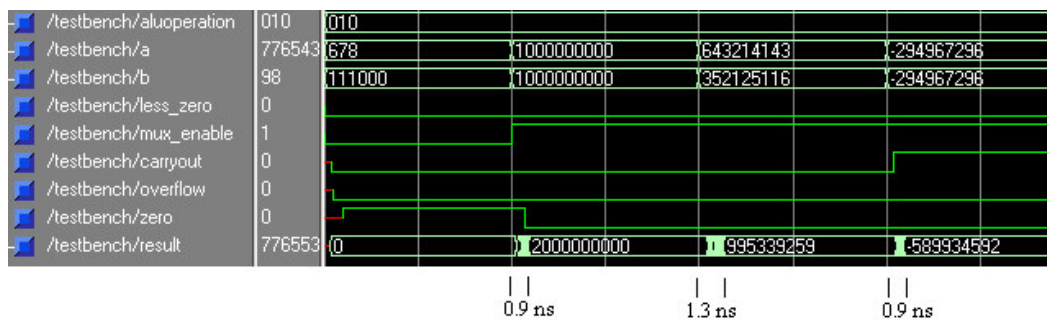


Figure A.52 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = ADD.

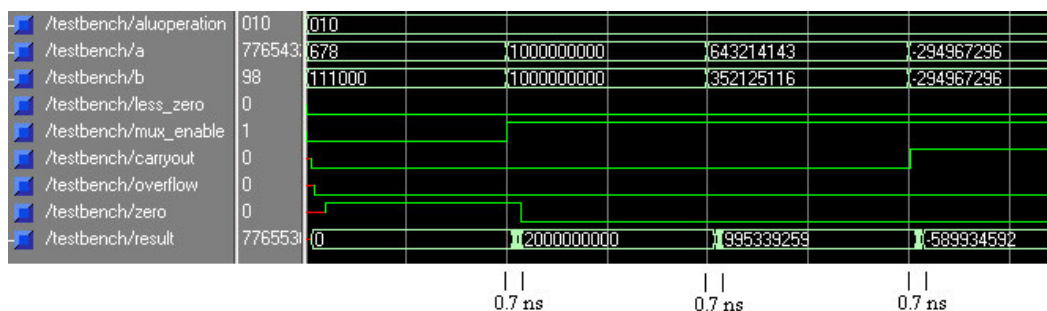


Figure A.109 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = ADD.

- When the ALU Function is SUB (figure A.110):
 - Both output signals *carryout* and *result* are of interest.
 - The same argument relating to the unwanted intermediate values as the above ADD case applies here. In figure A.110, the output *result* is always settling down to the correct value after a fixed delay of 0.8 ns from the time the input signals *a* and *b* have been applied.

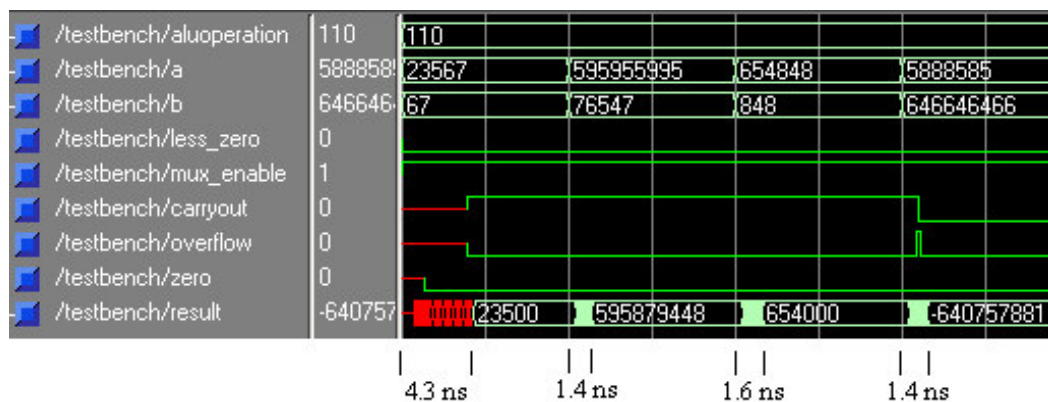


Figure A.53 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = SUB.

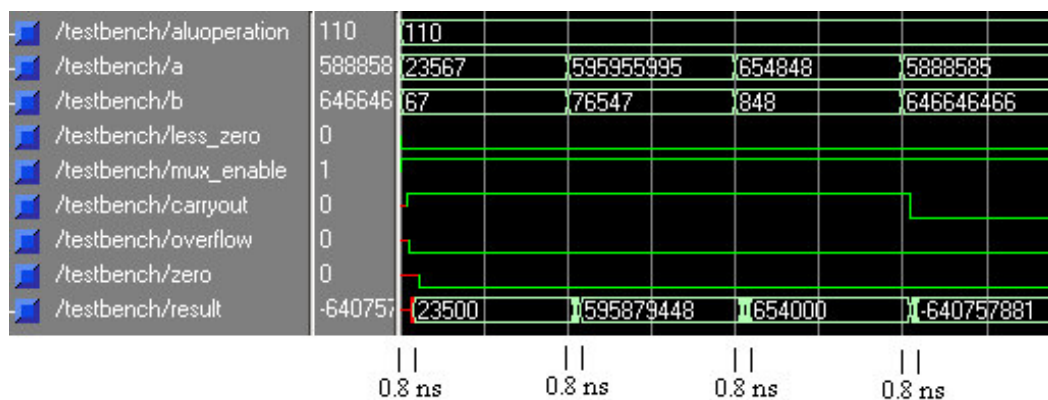


Figure A.110 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = SUB.

□ When the ALU Function is either BEQ or BNE (figure A.111):

- The only output signal of interest is *zero*.
- Again, due to the fact that this ALU is based on carry lookahead addition/subtraction, the *zero* output undergoes a shorter time delay compared to ripple carry. In figure A.111, the output *zero* is always producing the correct value after a fixed delay of 1.5 ns from the time the input signals *a* and *b* have been applied.

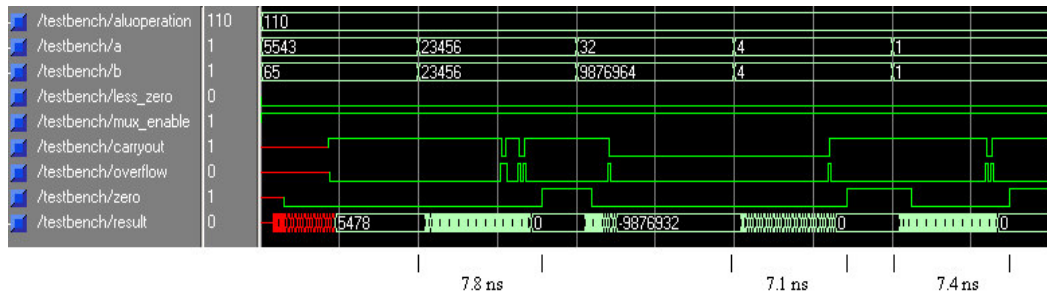


Figure A.54 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = BEQ/BNE.

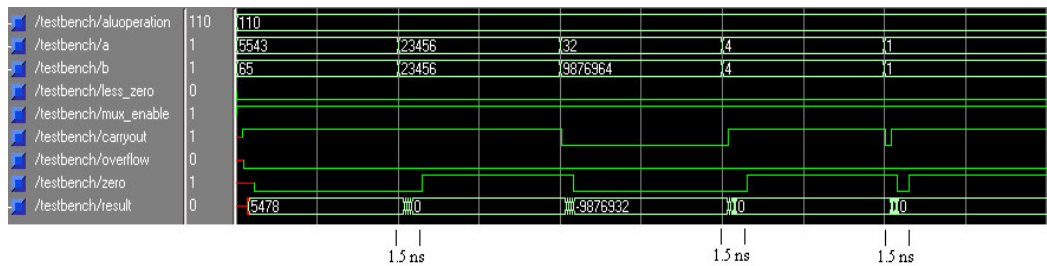


Figure A.111 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = BEQ/BNE.

- ❑ When the ALU Function is SLT (figure A.112):
 - The only output signal of interest is *result*.
 - Again, due to the fact that this ALU is based on carry lookahead addition/subtraction, the *result* output undergoes a shorter time delay compared to ripple carry. In figure A.112, the output *result* is always producing the correct value after a fixed delay of 0.9 ns from the time the input signals *a* and *b* have been applied.
- ❑ Although the output signal *overflow* is implemented, it will not be utilized in any aspect within the context of this research, but rather for future research.

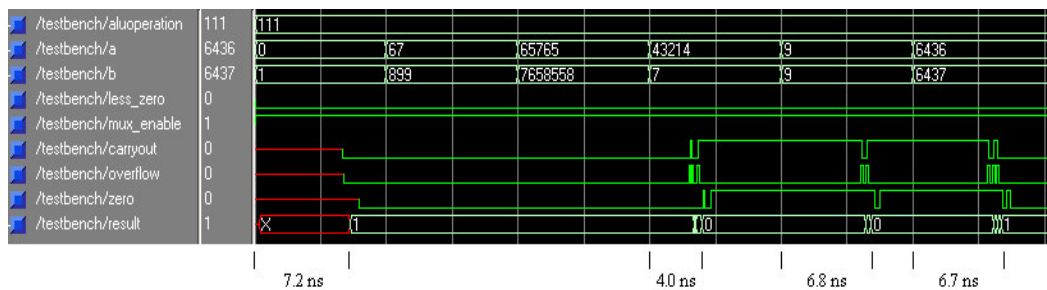


Figure A.55 Results of simulating the synthesized final 32-bit ripple carry MIPS ALU using ModelSim, with ALU Function = SLT.

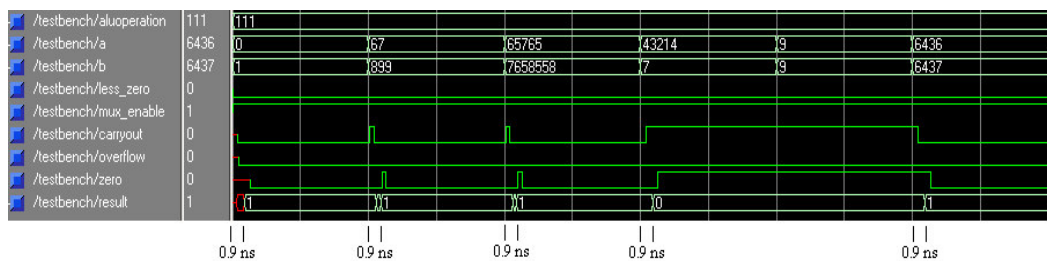


Figure A.112 Results of simulating the synthesized final 32-bit MIPS ALU with CLA using ModelSim, with ALU Function = SLT.

It is clear from these figures (A.107 to A.112) that the resulting synthesized hardware functions correctly and according to specification. This concludes the design cycle for this component.

Even though the various output signals in all these figures (A.107 to A.112) still display hazards which result in a combination of spikes, unwanted intermediate values, and noticeably delayed correct values. In the context of this research, one or more of the following remedies have been implemented:

- ❑ Increase the time interval (for combinational logic) or clock cycle time (for sequential logic) from 10 ns to 20 ns or even to 100 ns or more.

- ❑ Delayed Clocking: When the output value from a combinational element is to be clocked in (stored into) a state element (register or memory), the rising edge of the clock is delayed to ensure that the combinational output has settled and stabilized to the correct value.
- ❑ Or, a combination of both of the above techniques can be implemented at the same time.

Final 32-Bit MIPS ALU: Ripple Carry versus Carry Lookahead

This final summary concludes with a comparison between the 32-bit MIPS ALU using ripple carry and its counterpart using carry lookahead. This is based on comparing the hardware resources utilization and performance speed-up.

❑ *Hardware Resources Utilization:*

Comparing the FPGA synthesis reports of both ALUs, the carry look-ahead 32-bit ALU utilizes only twice as much hardware resources (73 slices and 128 4-input LUTs) as compared to the ripple carry 32-bit ALU (36 slices and 64 4-input LUTs)!

❑ *Performance Speed-up:*

For the purpose of this comparison, this very basic and simple formula will be used to measure the speedup in the carry lookahead ALU vs. the ripple carry ALU:

$$\text{Speed-up} = \frac{\text{Max. Time Delay for ALU with RC}}{\text{Max. Time Delay for ALU with CLA}}$$

Comparing the simulation results waveforms of both ALUs (figures A.50 to A.55 against figures A.107 to A.112), the following observations are made regarding speed-up and the *maximum time delay* between the time when the inputs *a* and *b* are applied and the time when the relevant output signals stabilize to the correct values:

➤ *When ALU Function = AND:*

Average Time Delay:

For ripple carry ALU = 0.6 ns

For carry lookahead ALU = 0.6 ns

Speed-up = 0.6 ns / 0.6 ns = 1

➤ When ALU Function = OR:

Average Time Delay:

For ripple carry ALU = 0.6 ns

For carry lookahead ALU = 0.6 ns

Speed-up = 0.6 ns / 0.6 ns = 1

➤ When ALU Function = ADD:

Max. Time Delay:

For ripple carry ALU = 1.3 ns

For carry lookahead ALU = 0.7 ns

Speed-up = 1.3 ns / 0.7 ns = 1.86

➤ When ALU Function = SUB:

Max. Time Delay:

For ripple carry ALU = 4.3 ns

For carry lookahead ALU = 0.8 ns

Speed-up = 4.3 ns / 0.8 ns = 5.38

➤ When ALU Function = BEQ/BNE:

Max. Time Delay:

For ripple carry ALU = 7.8 ns

For carry lookahead ALU = 1.5 ns

$$\text{Speed-up} = 7.8 \text{ ns} / 1.5 \text{ ns} = \underline{5.2}$$

➤ When ALU Function = SLT:

Max. Time Delay:

For ripple carry ALU = 7.2 ns

For carry lookahead ALU = 0.9 ns

$$\text{Speed-up} = 7.2 \text{ ns} / 0.9 \text{ ns} = \underline{8}$$

➤ Conclusion:

Speed-up ranges from 1 in logical operations (where the CLA unit does not come into play) to 8 in the case of the SLT operation. However:

$$\text{Average Speed-up} = (1 + 1 + 1.86 + 5.38 + 5.2 + 8) / 6 = \underline{3.74}$$

❑ **Final Conclusion:**

A 32-bit CLA ALU performs 1 to 8 times (3.74 on average) faster than a 32-bit RC ALU for only twice as much hardware resources!

A.3.6 Instruction Splitter

➤ *RTL Description*

The Instruction Splitter is a combinational logic component designed specially for the customized requirements of implementing the MIPS processor RTL model in this research. It is actually a bus tap to extract specific bit fields from the 32-bit instruction and feed these bits into the relevant datapath components and/or control unit. The details of the different bit fields making up the MIPS 32-bit instruction are covered in chapter 5.

➤ *Design Entry and Synthesis*

Below is the VHDL code for synthesizing the instruction splitter unit, entirely from VHDL constructs by using HDL Editor.

```

-- Instruction Splitter:
-- This unit is equivalent to a bus tap to extract specific bits from the instruction.
-- Source: COD2E Ch 3.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Instruction_Splitter is
    Port (
        Instruction : in  std_logic_vector(31 downto 0);
        Bits31_26   : out std_logic_vector(5  downto 0);
        Bits25_21   : out std_logic_vector(4  downto 0);
        Bits20_16   : out std_logic_vector(4  downto 0);
        Bits15_11   : out std_logic_vector(4  downto 0);
        Bits10_6    : out std_logic_vector(4  downto 0);
        Bits5_0     : out std_logic_vector(5  downto 0);
        Bits15_0    : out std_logic_vector(15 downto 0);
        Bits25_0    : out std_logic_vector(25 downto 0)
    );
end Instruction_Splitter;

architecture Behavioral of Instruction_Splitter is

begin

    Bits31_26 <= Instruction(31 downto 26); -- opcode
    Bits25_21 <= Instruction(25 downto 21); -- rs
    Bits20_16 <= Instruction(20 downto 16); -- rt
    Bits15_11 <= Instruction(15 downto 11); -- rd
    Bits10_6  <= Instruction(10 downto  6); -- shamt
    Bits5_0   <= Instruction(5  downto  0); -- funct
    Bits15_0  <= Instruction(15 downto  0); -- 16-bit address
    Bits25_0  <= Instruction(25 downto  0); -- 26-bit address

end Behavioral;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the instruction splitter, was generated. Figure A.113 shows the resulting top level RTL symbol while figure A.114 shows the resulting top level RTL schematic diagram.

Note that in figure A.114, the synthesized top level RTL schematic does not show any labelling for six of the eight output bus taps. This is obviously a shortcoming in the ISE software. However, when testing and simulating the instruction splitter (as will be seen shortly), the behaviour was 100% as expected.

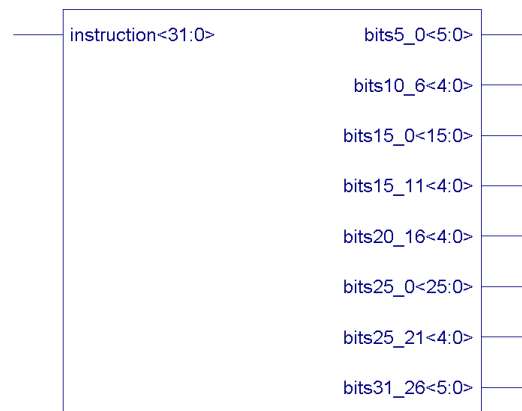


Figure A.113 Resulting top level RTL symbol for the synthesized instruction splitter.

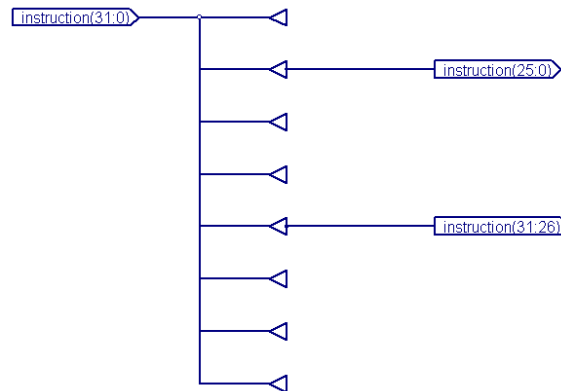


Figure A.114 Resulting top level RTL schematic for the synthesized instruction splitter.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the instruction splitter using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 106

Cell Usage:

IO Buffers : 106

IBUF : 32

OBUF : 74

Device utilization summary:

Number of bonded IOBs: 106 out of 1108 9%

➤ *Place-and-Route onto the FPGA*

In figure A.115, FPGA Editor shows the synthesized instruction splitter after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running alongside the outer edges of the FPGA chip.

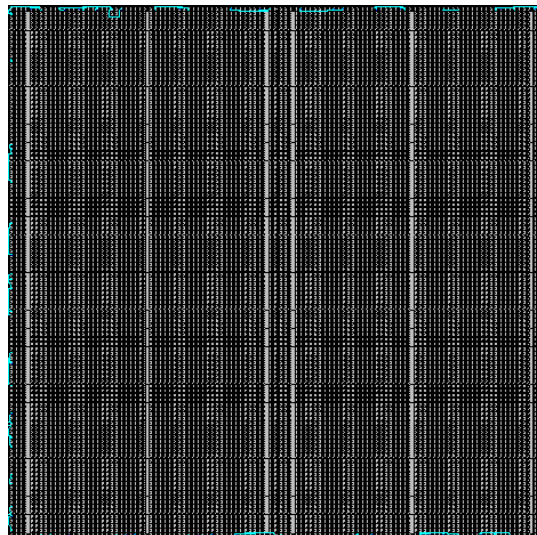


Figure A.115 *FPGA Editor showing the synthesized instruction splitter after place-and-route onto the target Virtex-II FPGA chip.*

➤ Simulation Results

Figure A.116 shows the waveform results of simulating the instruction splitter VHDL behavioural model in Mentor Graphics ModelSim. All these waveform are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the instruction splitter. This concludes the design cycle for this component.

/instruction	000000	00000000000010000100101100110111				00000000000000000000111000111011				00000000000000000000000000000000
/bits31_26	000000	000000								
/bits25_21	000000	000000								
/bits20_16	000000	010000				000000				
/bits15_11	000000	01001				000001			000000	
/bits10_6	000000	01100				111000			000000	
/bits5_0	000000	110111				1111011			0000000	
/bits15_0	000000	0100101100110111				00000111000111011			000000000000000000	
/bits25_0	000000	00000010000100101100110111				000000000000000111000111011			000000000000000000000000	

Figure A.116 Results of simulating the synthesized instruction splitter using ModelSim.

A.3.7 Address Truncator

➤ RTL Description

The address truncator is a combinational logic component designed specially for the customized requirements of implementing the MIPS processor RTL model in this research. It is actually a bus tap to truncate a 16-bit address field to an 8-bit address field by using only the lower 8 of the 16 bits. This address truncation is mandatory because the address field embedded within the 32-bit MIPS instruction or generated by some datapath sections is 16-bit wide while the memory (instruction and data memories) implemented in this research is based on 8-bit addressing (limiting the synthesized memory size due to limited FPGA resources).

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing the address truncator unit, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
```

```

-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Sign_Dextension is
    Port ( Addr_in  : in  std_logic_vector(15 downto 0);
          Addr_out  : out std_logic_vector(7  downto 0));
end Sign_Dextension;

architecture Behavioral of Sign_Dextension is

begin

    Addr_out(7)  <= Addr_in(7);
    Addr_out(6)  <= Addr_in(6);
    Addr_out(5)  <= Addr_in(5);
    Addr_out(4)  <= Addr_in(4);
    Addr_out(3)  <= Addr_in(3);
    Addr_out(2)  <= Addr_in(2);
    Addr_out(1)  <= Addr_in(1);
    Addr_out(0)  <= Addr_in(0);

end Behavioral;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the address truncator, was generated. Figure A.117 shows the resulting top level RTL symbol while figure A.118 shows the resulting top level RTL schematic diagram.



Figure A.117 Resulting top level RTL symbol for the synthesized address truncator.



Figure A.118 Resulting top level RTL schematic for the synthesized address truncator.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the address truncator using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:


```

Design Statistics:
# I/Os                                     : 24

Cell Usage:
# IO Buffers                             : 16
#      IBUF                               : 8
#      OBUF                               : 8

Device utilization summary:

Number of bonded IOBs:                    16 out of 1108    1%

```

➤ *Place-and-Route onto the FPGA*

In figure A.119, FPGA Editor shows the synthesized address truncator after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running alongside the outer edges of the FPGA chip.

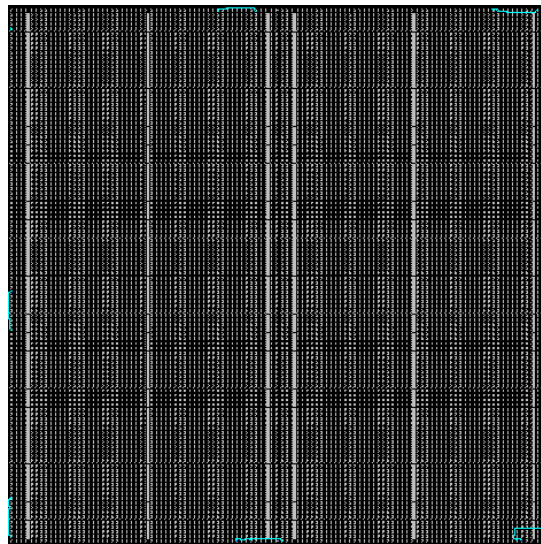


Figure A.119 *FPGA Editor showing the synthesized address truncator after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figure A.120 shows the waveform results of simulating the synthesized address truncator VHDL behavioural model in Mentor Graphics ModelSim. All these waveform are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the address truncator. This concludes the design cycle for this component.

/testbench/addr_in	100101	1111111111111111	0010101000111100	0000000000000000	1001010100010111
/testbench/addr_out	000101	11111111	00111100	00000000	00010111

Figure A.120 Results of simulating the synthesized address truncator using ModelSim.

A.3.8 Sign Extender

➤ RTL Description

The Sign Extender is a combinational logic component designed specially for the customized requirements of implementing the MIPS processor RTL model in this research. It is actually a bus tap to convert a 16-bit address to a 32-bit address by replicating the sign bit (which is the MSB) of the 16-bit address field to fill the upper 16 bits of the extended 32-bit address. This sign extension is also a requirement of the MIPS ISA design.

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing the sign extender unit, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Sign_Extension is
    Port ( Addr_in  : in  std_logic_vector(15 downto 0);
          Addr_out  : out std_logic_vector(31 downto 0));
end Sign_Extension;

architecture Behavioral of Sign_Extension is

begin

    Addr_out(31) <= Addr_in(15);
    Addr_out(30) <= Addr_in(15);
    Addr_out(29) <= Addr_in(15);
    Addr_out(28) <= Addr_in(15);
```

```

Addr_out (27) <= Addr_in(15);
Addr_out (26) <= Addr_in(15);
Addr_out (25) <= Addr_in(15);
Addr_out (24) <= Addr_in(15);
Addr_out (23) <= Addr_in(15);
Addr_out (22) <= Addr_in(15);
Addr_out (21) <= Addr_in(15);
Addr_out (20) <= Addr_in(15);
Addr_out (19) <= Addr_in(15);
Addr_out (18) <= Addr_in(15);
Addr_out (17) <= Addr_in(15);
Addr_out (16) <= Addr_in(15);
Addr_out (15) <= Addr_in(15);
Addr_out (14) <= Addr_in(14);
Addr_out (13) <= Addr_in(13);
Addr_out (12) <= Addr_in(12);
Addr_out (11) <= Addr_in(11);
Addr_out (10) <= Addr_in(10);
Addr_out (9)  <= Addr_in(9);
Addr_out (8)  <= Addr_in(8);
Addr_out (7)  <= Addr_in(7);
Addr_out (6)  <= Addr_in(6);
Addr_out (5)  <= Addr_in(5);
Addr_out (4)  <= Addr_in(4);
Addr_out (3)  <= Addr_in(3);
Addr_out (2)  <= Addr_in(2);
Addr_out (1)  <= Addr_in(1);
Addr_out (0)  <= Addr_in(0);

end Behavioral;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the sign extender, was generated. Figure A.121 shows the resulting top level RTL symbol while figure A.122 shows the resulting top level RTL schematic diagram.



Figure A.121 Resulting top level RTL symbol for the synthesized sign extender.



Figure A.122 Resulting top level RTL schematic for the synthesized sign extender.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the sign extender using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 48

Cell Usage:

IO Buffers : 48

IBUF : 16

OBUF : 32

Device utilization summary:

Number of bonded IOBs: 48 out of 1108 4%

➤ *Place-and-Route onto the FPGA*

In figure A.123, FPGA Editor shows the synthesized sign extender after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running alongside the outer edges of the FPGA chip.



Figure A.123 *FPGA Editor showing the synthesized sign extender after place-and-route onto the target Virtex-II FPGA chip.*

➤ Simulation Results

Figure A.124 shows the waveform results of simulating the synthesized sign extender VHDL behavioural model in Mentor Graphics ModelSim. All these waveform are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the sign extender. This concludes the design cycle for this component.

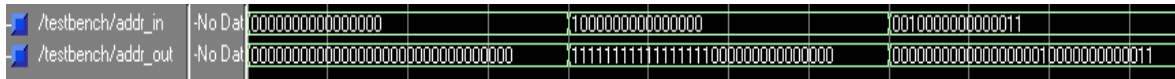


Figure A.124 Results of simulating the synthesized sign extender, using ModelSim.

A.3.9 32-to-8 Bit Converter

➤ RTL Description

The 32-to-8 bit converter is a combinational logic component designed specially for the customized requirements of implementing the MIPS processor RTL model in this research. Similar to the address truncator discussed previously in section A.3.7, it is actually a bus tap to truncate a 32-bit address field to an 8-bit address field by using only the lower 8 of the 32 bits. This address truncation is mandatory because the address field generated by some datapath sections is 32-bit wide while the memory (instruction and data memories) implemented in this research is based on 8-bit addressing (limiting the synthesized memory size due to limited FPGA resources).

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing the 32-to-8 bit converter, entirely from VHDL constructs by using HDL Editor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```

entity Conv_32to8b is
    Port ( din   : in std_logic_vector(31 downto 0);
          dout  : out std_logic_vector(7 downto 0));
end Conv_32to8b;

architecture Behavioral of Conv_32to8b is

begin

    dout <= din(7 downto 0);

end Behavioral;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the 32-to-8 bit converter, was generated. Figure A.125 shows the resulting top level RTL symbol while figure A.126 shows the resulting top level RTL schematic diagram.



Figure A.125 Resulting top level RTL symbol for the synthesized 32-to-8 bit converter.

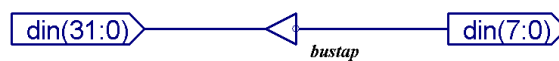


Figure A.126 Resulting top level RTL schematic for the synthesized 32-to-8 bit converter.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the 32-to-8 bit converter using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```

Design Statistics:
# IOs                               : 40

Cell Usage:
# IO Buffers                        : 16
# IBUF                             : 8

```

```
#          OBUF          : 8

Device utilization summary:

Number of bonded IOBs:          16  out of   1108    1%
```

➤ *Place-and-Route onto the FPGA*

In figure A.127, FPGA Editor shows the synthesized 32-to-8 bit converter after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections running alongside the outer edges of the FPGA chip.

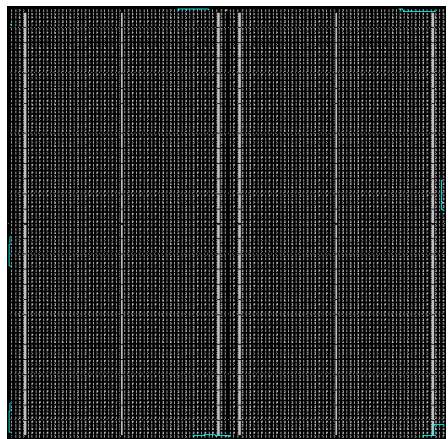


Figure A.127 *FPGA Editor showing the synthesized 32-to-8 bit converter after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figure A.127 shows the waveform results of simulating the synthesized 32-to-8 bit converter VHDL behavioural model in Mentor Graphics ModelSim. All these waveform are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the 32-to-8 bit converter. This concludes the design cycle for this component.

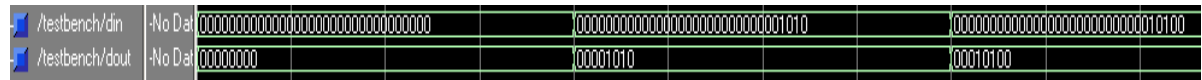


Figure A.127 *Results of simulating the synthesized 32-to-8 bit converter using ModelSim.*

A.3.10 Registers

Registers and Flip Flops (the building blocks for registers) are the basic building blocks for state (sequential) logic components. Flip flops and registers are explained in detail in [47, 40, 48], while their VHDL descriptions are elaborated in [75, 40]. However, the implementation and synthesis of flip flops and registers onto FPGAs in general and the Virtex-II FPGA in particular are discussed in [72, 73, 70].

In this section, the two registers of importance to this research are the *Program Counter (PC)* and the *Register File (RF)*, both of which are discussed here.

The Program Counter (PC) Register

➤ *RTL Description*

The PC is a sequential logic component. The role of the PC is discussed in detail in [1, 16]. Basically, it is a register, which holds the address of the next instruction to be executed. In the MIPS ISA design, it is 32 bits wide. However, in my implementation it is only 8 bits wide and can therefore address only 256 memory locations, which is more than enough to run a reasonably sized footprint code. This design compromise has been adopted due to the fact that implementing memory in the FPGA chip is quite demanding in hardware resources.

➤ *Design Entry and Synthesis*

Below is the VHDL code for synthesizing the program counter, entirely from VHDL constructs by using HDL Editor.

```
-- 32-bit Program Counter with Asynchronous Reset

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PC is
```



```

Port (
    CLK:   in   STD_LOGIC;
    RESET: in   STD_LOGIC;
    DIN:   in   STD_LOGIC_VECTOR(7 downto 0);
    DOUT:  out  STD_LOGIC_VECTOR(7 downto 0)
);
end PC;

architecture Behavioral of PC is

begin

    process (CLK, RESET)
    begin
        if RESET='1' then    --asynchronous RESET active High
            DOUT <= "00000000";
        elsif (CLK'event and CLK='1') then    --CLK rising edge
            DOUT <= DIN;
        end if;
    end process;

end Behavioral;

```

➤ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the program counter, was generated. Figure A.128 shows the resulting top level RTL symbol while figure A.129 shows the resulting top level RTL schematic diagram and illustrates that the program counter is synthesized from the Virtex-II library as an 8-bit register with an active high asynchronous reset signal.

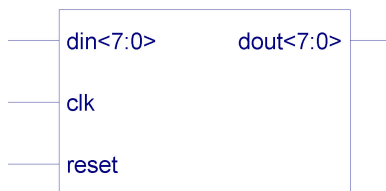


Figure A.128 Resulting top level RTL symbol for the synthesized program counter.

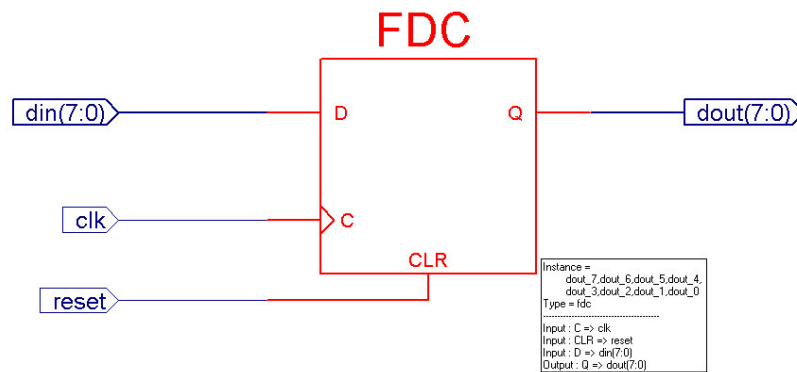


Figure A.129 Resulting top level RTL schematic for the synthesized program counter.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the program counter using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 18

Macro Statistics:

Registers : 1

8-bit register : 1

Cell Usage:

FlipFlops/Latches : 8

FDC : 8

Clock Buffers : 1

BUFPG : 1

IO Buffers : 17

IBUF : 9

OBUF : 8

Device utilization summary:

Number of Slices: 5 out of 46592 0%

Number of Slice Flip Flops: 8 out of 93184 0%

Number of bonded IOBs: 17 out of 1108 1%

Number of GCLKs: 1 out of 16 6%

➤ *Place-and-Route onto the FPGA*

In figure A.130, FPGA Editor shows the synthesized program counter after place-and-route onto the target Virtex-II FPGA chip. Notice that this is the blue interconnections at the middle of the top of the FPGA chip.

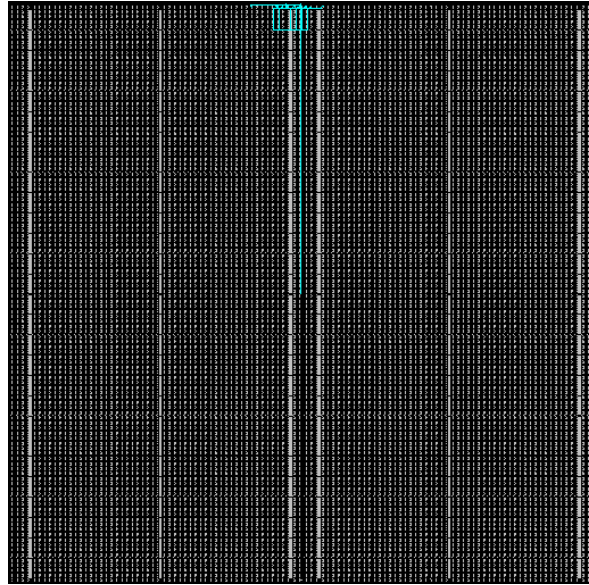


Figure A.130 *FPGA Editor showing the synthesized program counter after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figure A.131 shows the waveform results of simulating the synthesized program counter VHDL behavioural model in Mentor Graphics ModelSim. The waveforms for input signals *clk* and *reset* are in binary format while those for signals *din* and *dout* are in signed decimal format.

When checking these waveforms, the following points are worth noting:

- ❑ When the asynchronous *reset* is asserted high (= 1), the output *dout* is reset to the value of 0.
- ❑ When the asynchronous *reset* is de-asserted to low (= 0), the value of *din* is clocked in at the rising edge of the clock signal *clk* and would show up immediately at the output *dout*.

It is clear that the resulting synthesized hardware functions according to the specified behavior of the program counter. This concludes the design cycle for this component.

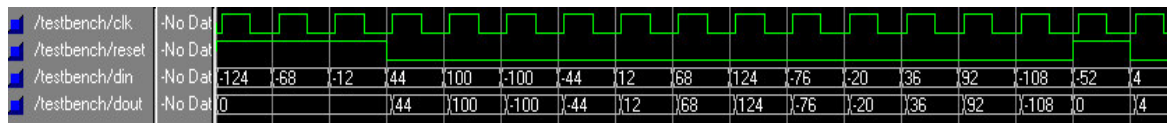


Figure A.131 Results of simulating the synthesized program counter using ModelSim.

The Register File (RF)

➤ RTL Description

The RF is a sequential logic component. The role of the RF is discussed in detail in [47, 48]. Basically, it is a set of 32 32-bit registers and is central to the datapath in the MIPS architecture [47]. Figure A.132 shows the input and output ports of a MIPS RF.

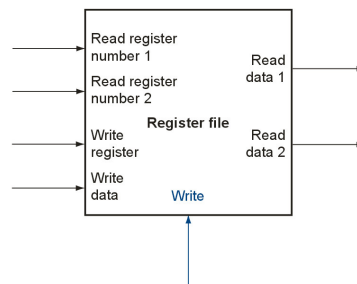


Figure A.132 The Register File (RF) with two read ports and one write port has five inputs and two outputs [47, p.B-25].

Following the well-known and agreed-upon convention among digital designers that the design of digital circuits encompassing sequential logic is recommended to be *synchronous*, the register file implemented in this research is a *synchronous* register file in which both the input (write) and output (read) operations are synchronized with the rising edge of the clock. This means that a write operation performed in the current clock cycle (during the *clk* rising edge) is available for read only in the next clock cycle. Similarly, a read operation performed in the current clock cycle (during the *clk* rising edge) yields the value written in the previous cycle.

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing the register file, entirely from VHDL constructs by using HDL Editor.

```
-- Register File with both i/p and o/p synchronized with clk
-- This means:
```

```

-- Write in current cycle is available for read in next cycle.
-- Read in current cycle is from write in previous cycle.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RF_synch is
    Port ( read_reg1   : in  std_logic_vector(4  downto 0);
          read_reg2   : in  std_logic_vector(4  downto 0);
          write_data   : in  std_logic_vector(31 downto 0);
          write_reg    : in  std_logic_vector(4  downto 0);
--          clear      : in  std_logic;
          clk          : in  std_logic;
          enable       : in  std_logic;
          read_data_1  : out std_logic_vector(31 downto 0);
          read_data_2  : out std_logic_vector(31 downto 0)
        );
end RF_synch;

architecture Behavioral of RF_synch is

    type reg_type is array (31 downto 0) of std_logic_vector (31 downto 0);
    signal REG : reg_type;

    begin

        process (clk) -- Write and Read process: edge-triggered.
            begin
                if (clk'event and clk = '1') then
                    if (enable = '1') then
                        REG(conv_integer(write_reg)) <= write_data;
                        read_data_1 <= REG(conv_integer(read_reg1));
                        read_data_2 <= REG(conv_integer(read_reg2));
                    end if;
                end if;
            end process;

        end Behavioral;

```

An important issue to highlight here is that the above VHDL code cannot properly synthesize the *clear* signal (because XST does not support synthesizing BlockRAM with *asynchronous reset*), which is for

resetting the contents of all registers. However, one way around this is that the registers could be reset using explicit preload with zero values during initialization. This method is implemented in this research.

However, as seen in the above VHDL code, this register file supports the *enable* signal. Only when *enable* is asserted high (and during a rising *clk* edge) can the register file read and/or write. Therefore, this register file is based on *synchronous write* and *synchronous read*.

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the register file, was generated. Figure A.133 shows the resulting top level RTL symbol while figure A.134 shows the resulting top level RTL schematic diagram. Figures A.135 and A.136 show magnified versions of figure A.134. These figures illustrate that this synchronous register file is synthesized from the Virtex-II library using *BlockRAM (BRAM)*, which is relatively abundant in the Virtex-II FPGA chip and utilizes no discrete gates nor CLBs. Actually, this register file is synthesized using two 32x32-bit dual-port block RAMs because each BRAM has a maximum of two input ports while the register file requires three.

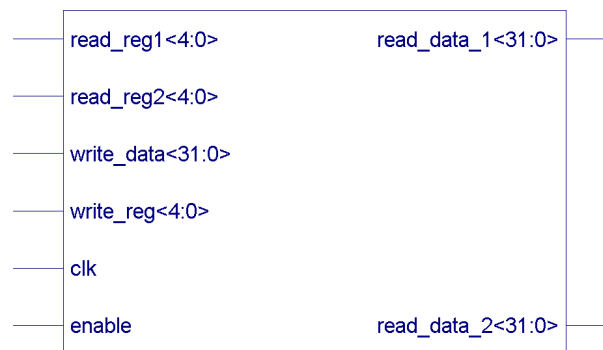


Figure A.133 Resulting top level RTL symbol for the synthesized register file.

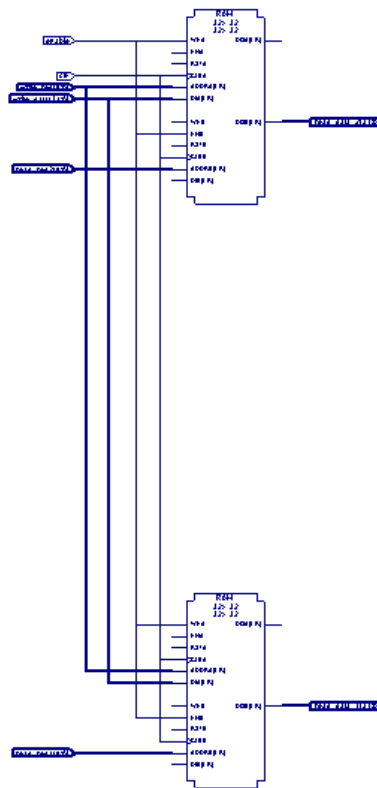


Figure A.134 Resulting top level RTL schematic for the synthesized register file.

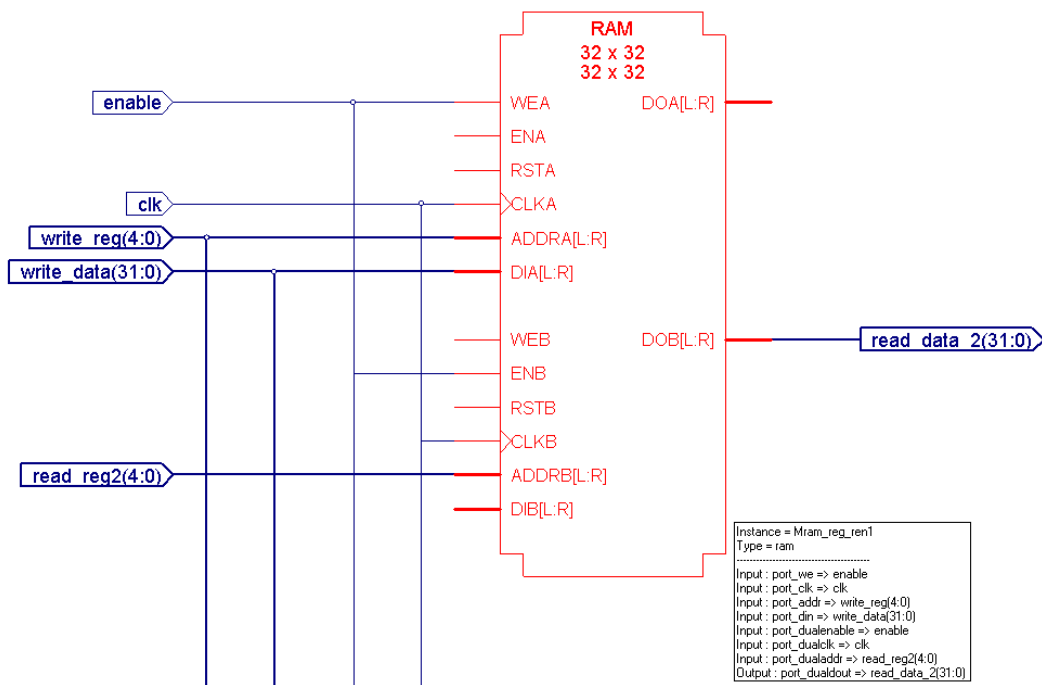


Figure A.135 Magnification of the upper section of figure A.134 for the resulting top level RTL schematic for the synthesized register file.

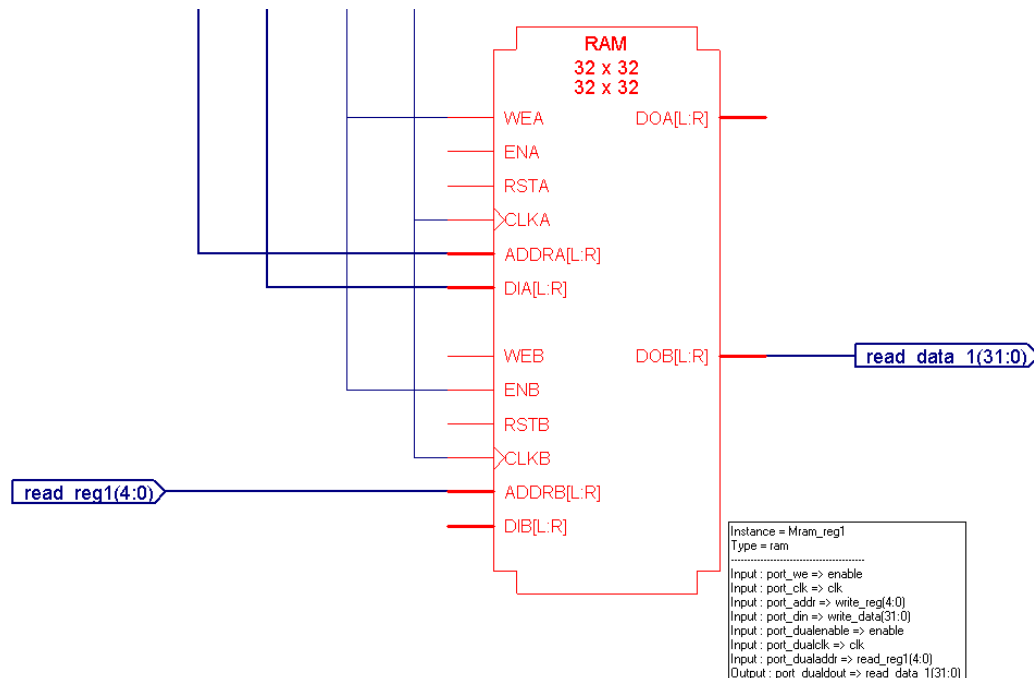


Figure A.136 Magnification of the lower section of figure A.134 for the resulting top level RTL schematic for the synthesized register file.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the register file using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```
Design Statistics:
# I/Os                                     : 113

Macro Statistics:
# RAM                                       : 2
# 32x32-bit dual-port block RAM: 2

Cell Usage :
# BELS                                     : 2
# GND                                     : 1
# VCC                                     : 1
# RAMS                                    : 2
# RAMB16_S36_S36                         : 2
# Clock Buffers                          : 1
# BUFGP                                  : 1
# IO Buffers                             : 112
# IBUF                                  : 48
# OBUF                                  : 64
```


Device utilization summary:

Number of bonded IOBs:	112	out of	1108	10%
Number of BRAMs:	2	out of	168	1%
Number of GCLKs:	1	out of	16	6%

➤ *Place-and-Route onto the FPGA*

In figure A.137, FPGA Editor shows the synthesized register file after place-and-route onto the target Virtex-II FPGA chip. Notice that this is the cluster of blue interconnections at the middle of the top of the FPGA chip.

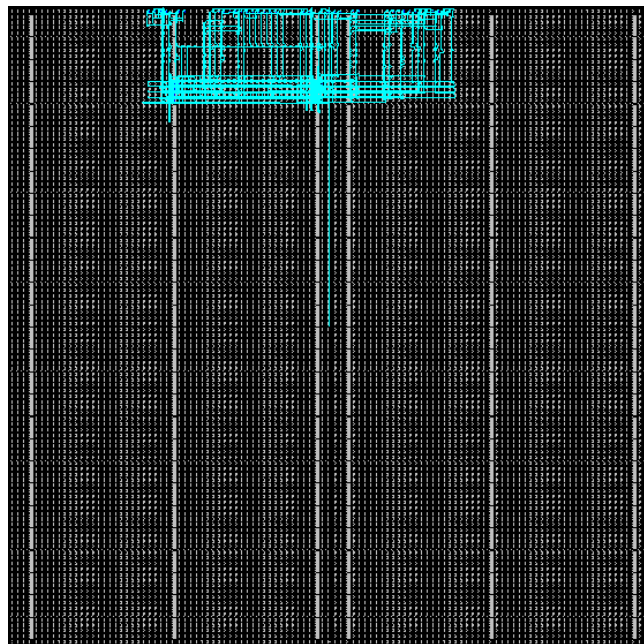


Figure A.137 *FPGA Editor showing the synthesized register file after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figures A.138 and A.139 show the waveform results of simulating the synthesized register file VHDL behavioural model in Mentor Graphics ModelSim. In both figures, all waveforms are in signed decimal format except for input signals *clk* and *enable* which are in binary format.

When checking these waveforms, the following points are worth noting:

□ In figure A.138:

- When *enable* = 0, no value is written to nor read out from the register file.
- When *enable* = 1, the register file is having each register (in sequence) written to in one clock cycle then that same value is being read out in the next clock cycle.

□ In figure A.139:

- Register no. 1 (reg1) in the register file is being written to in one clock cycle, then that same value is being read out in the next clock cycle.
- During the first clock cycle, the value read out from reg1 is XXXXXXXXXXXXXXXXXXXXXXXXXXXX because it is not initialized to any deterministic value yet.

It is clear that the resulting synthesized hardware functions according to the specified behaviour of the register file. This concludes the design cycle for this component.

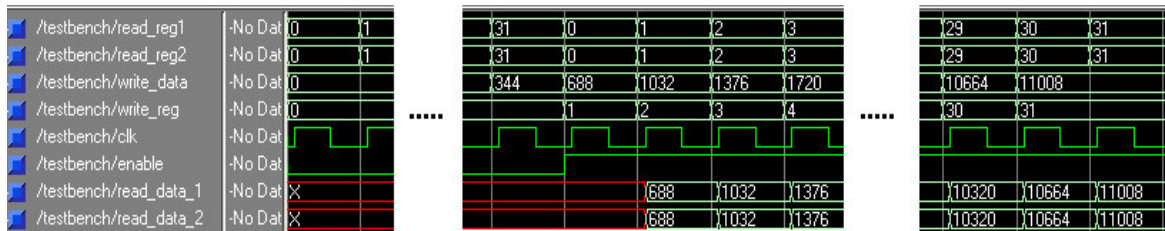


Figure A.138 Results of simulating the synthesized register file, using ModelSim. This shows the writing to and then reading from all 32 registers in the register file in sequence.

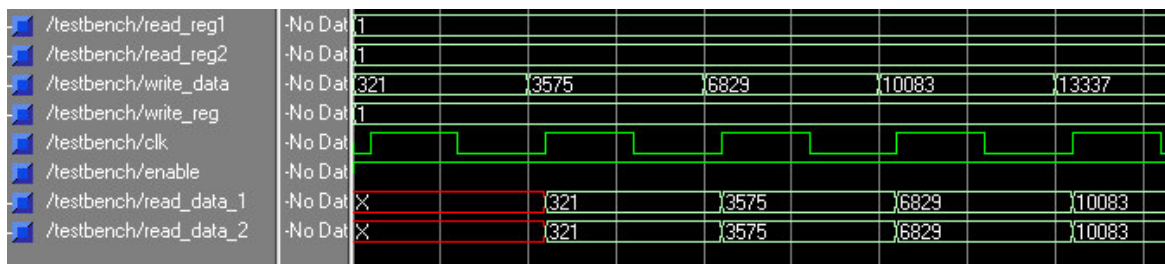


Figure A.139 Results of simulating the synthesized register file, using ModelSim. This shows the writing to and reading from reg1 in the same clock cycle.

A.3.11 Memory

Memory is a sequential logic component. Flip flops, registers and register files are the basic building blocks for small memories whereas larger memories are built using either *SRAMs* (*Static Random Access Memories*) or *DRAMs* (*Dynamic Random Access Memories*) [47, 48]. Memories in both forms SRAM and DRAM and their implementation in the MIPS R2000 are discussed in detail in [47, 48] while their VHDL descriptions are elaborated in [75, 40]. However, the implementation and synthesis of memories onto FPGAs in general and the Virtex-II FPGA in particular are discussed in [72, 73, 70].

In this section, the two memories of importance to this research are the *Instruction Memory (IM)* and the *Data Memory (DM)*, both of which are discussed here. However, in the actual MIPS R2000 processor, the instruction memory is implemented as *Instruction Cache (IC)* while the data memory is implemented as *Data Cache (DC)*. Both these types of cache involve much more complicated logic and are therefore beyond the scope of this research.

Instruction Memory

➤ RTL Description

The instruction memory is a sequential logic component and its MIPS implementation is discussed in detail in [47, 48]. The instruction memory shown in figure A.140 is very basic and provides only read access because the datapath does not write instructions back to it [47, 48]. However, the customized implementation for the instruction memory in the context of this research involves modifying its design to add functionality for writing to (pre-loading) it with the code instructions necessary for running the simulator for the synthesized RTL model of the MIPS processor.

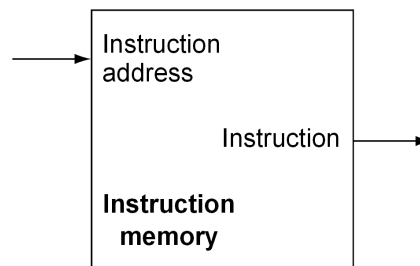


Figure A.140 The Instruction Memory (IM) in the MIPS architecture [47, p.344].

➤ *Design Entry and Synthesis*

Below is the VHDL code for synthesizing the instruction memory, entirely from VHDL constructs by using HDL Editor.

```
-- Only XST supports RAM inference
-- Infers single Port Block RAM
-- Both Read and Write are synchronized with the clock edge:

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Inst_RAM is
    Port (
        clk : in  std_logic;
        we  : in  std_logic;
        a   : in  std_logic_vector(7 downto 0); -- address (read/write)
        di  : in  std_logic_vector(31 downto 0);
        do  : out std_logic_vector(31 downto 0)
    );
end Inst_RAM;

architecture Behavioral of Inst_RAM is

    type ram_type is array (255 downto 0) of std_logic_vector (31 downto 0);
    signal RAM : ram_type;

    begin

        process (clk)
        begin
            if (clk'event and clk = '1') then
                if (we = '1') then
                    RAM(conv_integer(a)) <= di;
                end if;
            end if;
        end process;

        process (clk)
        begin
```

```

                                if (clk'event and clk = '1') then
                                    do <= RAM(conv_integer(a));
                                end if;
                                end process;

end Behavioral;

```

As seen in the above VHDL code, this instruction memory supports the *write enable* (*we*) signal. Only when *we* is asserted high (and during a rising clock edge) can the instruction memory be written. However, the instruction memory can still be read without the need for any enabling signal but the read is still synchronous with the rising clock edge. Therefore, similar to the register file, this instruction memory is based on *synchronous write* and *synchronous read*.

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the instruction memory, was generated. Figure A.141 shows the resulting top level RTL symbol while figure A.142 shows the resulting top level RTL schematic diagram. Figure A.142 illustrates that this instruction memory is synthesized from the Virtex-II library using one 256x32-bit single-port block RAM.

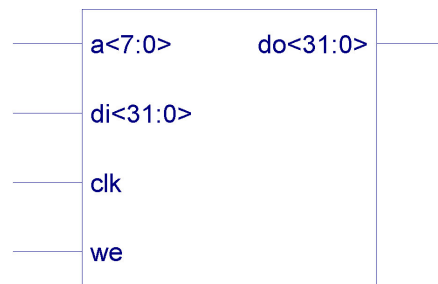


Figure A.141 Resulting top level RTL symbol for the synthesized instruction memory.

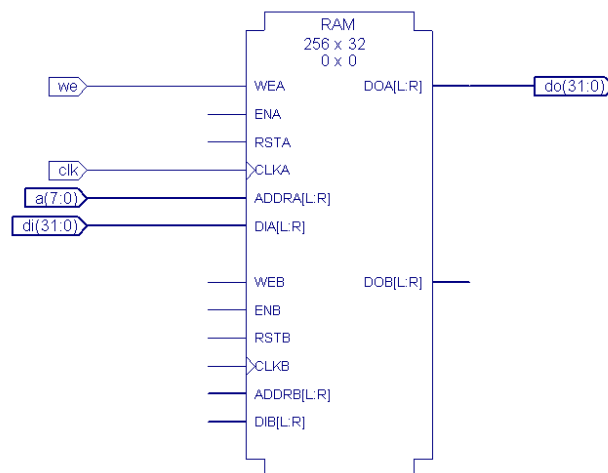


Figure A.142 Resulting top level RTL schematic for the synthesized instruction memory.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the instruction memory using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 74

Macro Statistics:

RAM : 1

256x32-bit single-port block RAM: 1

Cell Usage:

BELS : 2

GND : 1

VCC : 1

RAMS : 1

RAMB16_S36 : 1

Clock Buffers : 1

BUFGP : 1

IO Buffers : 73

IBUF : 41

OBUF : 32

Device utilization summary:

Number of bonded IOBs: 73 out of 1108 6%

Number of BRAMs: 1 out of 168 0%

Number of GCLKs:

1 out of 16 6%

➤ *Place-and-Route onto the FPGA*

In figure A.143, FPGA Editor shows the synthesized instruction memory after place-and-route onto the target Virtex-II FPGA chip. Notice that this is the cluster of blue interconnections at the middle of the top of the FPGA chip.

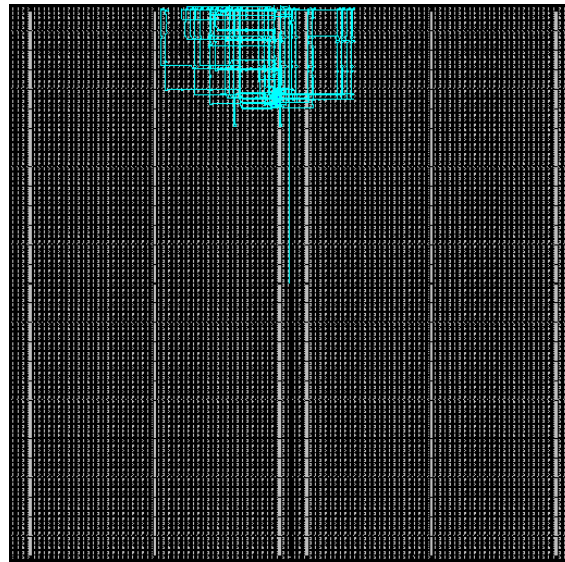


Figure A.143 FPGA Editor showing the synthesized instruction memory after place-and-route onto the target Virtex-II FPGA chip.

➤ *Simulation Results*

Figure A.144 shows the waveform results of simulating the synthesized instruction memory VHDL behavioural model in Mentor Graphics ModelSim. In this figure, the input signals *clk* and *we* are in binary format while the input signals *a* and *di* and the output signal *do* are in signed decimal format.

When checking these waveforms, the following points are worth noting:

➤ **During part 1 of the waveform (clock cycles 1 - 12):**

- The write enable signal *we* is asserted high for clock cycles 5 – 12.
- So memory locations 0 – 7 are being written to.

- At the same time, the value read out from these locations is `xxxxxxxxxxxxxxxxxxxxxxxxxxxx` because they are not initialized to any deterministic value yet.

➤ **During part 2 of the waveform (clock cycles 13 - 19):**

- The write enable signal *we* is de-asserted.
- So memory locations 7, 5 and 4 are only being read.
- No writing is enabled to any location.

➤ **During part 3 of the waveform (clock cycles 20 onwards):**

- The write enable signal *we* is asserted high.
- Writing to and reading from the same memory location (location 4) at the same time:
 - Writing a value in one clock cycle.
 - Reading that same value in the next clock cycle.

It is clear that the resulting synthesized hardware functions according to the specified behavior of the instruction memory. This concludes the design cycle for this component.

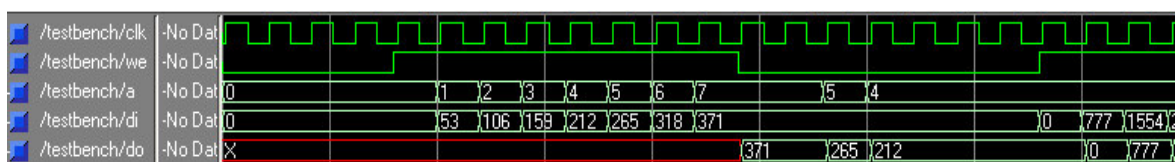


Figure A.144 Results of simulating the synthesized instruction memory, using ModelSim.

Data Memory

➤ *RTL Description*

The data memory is a sequential logic component and its MIPS implementation is discussed in detail in [47, 48]. The data memory shown in figure A.145 is what is implemented in [47, 48] and has one Address input, which can become a write address if the control signal *MemWrite* is asserted high or

alternatively a read address if *MemRead* is the one that is asserted high. Either *MemWrite* or *MemRead* is allowed to be asserted high at any one time [47, 48].

However, the customized implementation for the data memory in the context of this research involves modifying its design to accommodate the specific requirements and limitations of how it will synthesize onto the Virtex-II FPGA chip and would, therefore, deviate from what is shown in figure A.145 and discussed in [47, 48]. This is discussed in more detail very shortly.

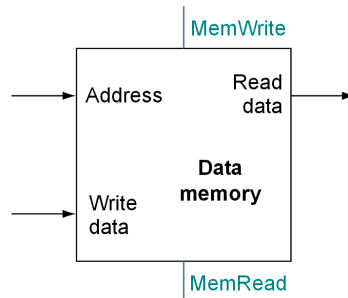


Figure A.145 The Data Memory (DM) in the MIPS architecture [47, p.348].

➤ Design Entry and Synthesis

Below is the VHDL code for synthesizing the instruction memory, entirely from VHDL constructs by using HDL Editor.

```

-- Only XST supports RAM inference
-- Infers Dual Port Block RAM
-- Data Cache
-- P&H pages 348 and B-28
-- Both read and write are sync with the clock edge
-- Cannot have OE signal, else will synthesize either dist RAM or Block RAM with
additional
-- o/p flip flops giving rise to an additional 1 cycle delay in read mode!

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Data_BRAM_Sync is

```

```

Port (
    clk : in  std_logic;
    we  : in  std_logic; -- write enable (MemWrite)
    wa  : in  std_logic_vector(7 downto 0);
    ra  : in  std_logic_vector(7 downto 0);
    di  : in  std_logic_vector(31 downto 0);
    do  : out std_logic_vector(31 downto 0)
);
end Data_BRAM_Sync;

architecture Behavioral of Data_BRAM_Sync is

    type ram_type is array (255 downto 0) of std_logic_vector (31 downto 0);
    signal RAM : ram_type;

begin

    process (clk) -- Write process: sync edge-triggered.
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(wa)) <= di;
            end if;
        end if;
    end process;

    process (clk) -- Read process: sync edge-triggered.
    begin
        if (clk'event and clk = '1') then
            do <= RAM(conv_integer(ra));
        end if;
    end process;

end Behavioral;

```

As seen in the above VHDL code and comparing it with figure A.145, this data memory has the following:

- Two address inputs: a *read address (ra)* and a *write address (wa)*, unlike figure A.145, which has only one *Address* input.
- The *write enable (we)* input signal. Only when *we* is asserted high (and during a rising clock edge) can the data memory be written. This is equivalent to the *MemWrite* control input in figure A.145

- There is no *read enable (re)* input signal (this would've been equivalent to the *MemRead* control input in figure A.145) otherwise XST will synthesize either distributed RAM (from discrete gates) or Block RAM but with additional output flip flops giving rise to an additional delay of 1 clock cycle in read mode, both of these are unwanted scenarios. However, in the current scenario the data memory can still be read without the need for any enabling control signal but the read is still synchronous with rising clock edge.

Therefore, similar to the register file and instruction memory, this data memory is based on *synchronous write* and *synchronous read*.

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the data memory, was generated. Figure A.146 shows the resulting top level RTL symbol while figure A.147 shows the resulting top level RTL schematic diagram. Figure A.147 illustrates that this data memory is synthesized from the Virtex-II library using one 256x32-bit dual-port block RAM.

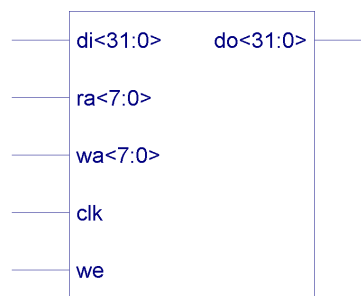


Figure A.146 Resulting top level RTL symbol for the synthesized data memory.

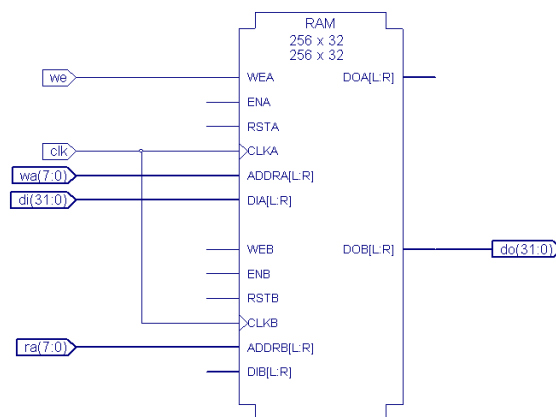


Figure A.147 Resulting top level RTL schematic for the synthesized data memory.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the data memory using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 82

Macro Statistics:

RAM : 1

256x32-bit dual-port block RAM: 1

Cell Usage:

BELS : 2

GND : 1

VCC : 1

RAMS : 1

RAMB16_S36_S36 : 1

Clock Buffers : 1

BUFGP : 1

IO Buffers : 81

IBUF : 49

OBUF : 32

Device utilization summary:

Number of bonded IOBs: 81 out of 1108 7%

Number of BRAMs: 1 out of 168 0%

Number of GCLKs: 1 out of 16 6%

➤ *Place-and-Route onto the FPGA*

In figure A.148, FPGA Editor shows the synthesized data memory after place-and-route onto the target Virtex-II FPGA chip. Notice that this is the cluster of blue interconnections at the middle of the top of the FPGA chip.

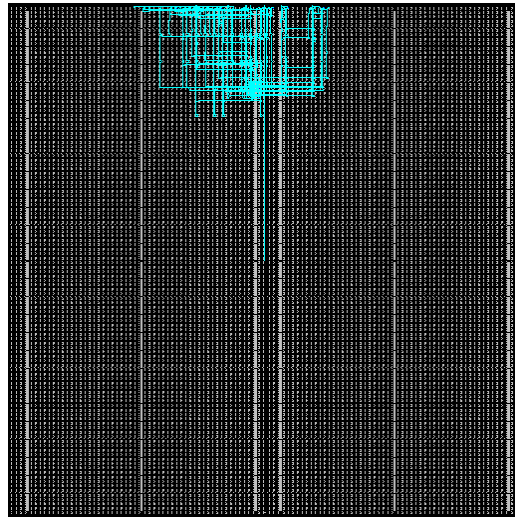


Figure A.148 FPGA Editor showing the synthesized data memory after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure A.149 shows the waveform results of simulating the synthesized data memory VHDL behavioural model in Mentor Graphics ModelSim. In this figure, the input signals *clk* and *we* are in binary format while the input signals *wa*, *ra* and *di* and the output signal *do* are in signed decimal format.

When checking these waveforms, the following points are worth noting:

➤ During part 1 of the waveform (clock cycles 1 - 4):

- The write enable signal *we* is de-asserted.
- Therefore, no writing is enabled to any location.
- At the same time, the value read out from locations 0, 1, 2, and 3 is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx because they are not initialized to any deterministic value yet.

➤ During part 2 of the waveform (clock cycles 5 - 12):

- The write enable signal *we* is asserted high.

- So memory locations 0 – 7 are being written to.
 - Also, each of these locations is also being read in the same clock cycle in which it is being written to. However, since the value being written in the current clock cycle is not available for reading until a subsequent clock cycle, the value read out in the current clock cycle is nothing but the uninitialized value of XXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
- **During part 3 of the waveform (clock cycles 13 - 20):**
- The write enable signal *we* is de-asserted.
 - So memory locations 0 - 7 are only being read. The values read out are the ones being stored in these locations during clock cycles 5 – 12.
 - No writing is enabled to any location.
- **During part 4 of the waveform (clock cycles 21 onwards):**
- The write enable signal *we* is asserted high.
 - Writing to and reading from the same memory location (location 4) at the same time:
 - Writing a value in one clock cycle.
 - Reading that same value in the next clock cycle.

It is clear that the resulting synthesized hardware functions according to the specified behaviour of the data memory. This concludes the design cycle for this component.

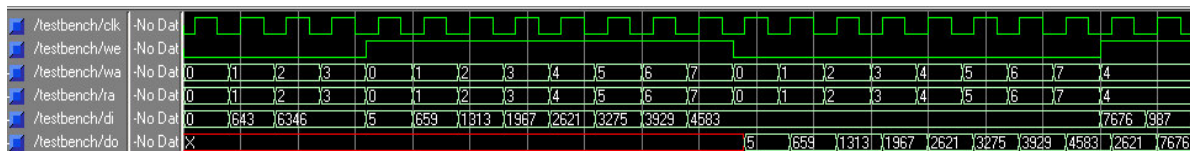


Figure A.149 Results of simulating the synthesized data memory, using ModelSim.

A.3.12 Digital Clock Manager (DCM)

➤ RTL Description

The DCM is described in detail in [69, 70]. Basically, the DCM provides a multitude of output clock frequencies, which are either in-phase or phase-shifted with regards to the input master clock *Clkin*. Also, the output clock frequency can be a division or multiplication of the input master clock *Clkin*.

➤ Design Entry and Synthesis

Figure A.150 shows how to infer and synthesize a user-customized DCM using Architecture Wizard in Xilinx ISE Project Manager.

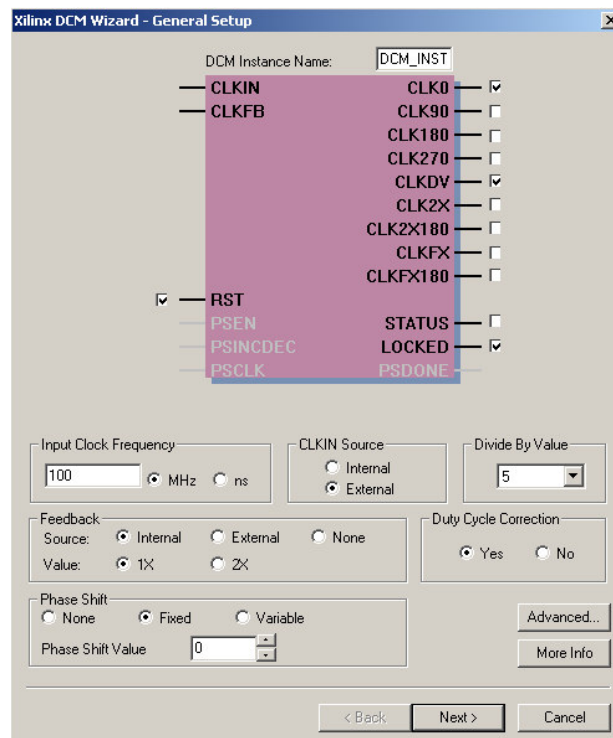


Figure A.150 Synthesizing a user-customized DCM using Architecture Wizard in Xilinx ISE Project Manager.

After synthesis of the DCM with the parameters shown in figure A.150 using Architecture Wizard, the resulting VHDL code shown below was generated.

```
-- Module DCM_Div_5
-- Generated by Xilinx Architecture Wizard
```

```
-- VHDL
-- Written for synthesis tool: XST
-- Xilinx Device: xc2v8000-4ff1517

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
-- synopsys translate_off
Library UNISIM;
use UNISIM.Vcomponents.all;
-- synopsys translate_on

entity DCM_Div_5 is
    port (
        RST_IN : in std_logic;
        CLKIN_IN : in std_logic;
        LOCKED_OUT : out std_logic;
        CLKDV_OUT : out std_logic;
        CLK0_OUT : out std_logic);
end DCM_Div_5;

architecture STRUCT of DCM_Div_5 is
    signal CLKIN_IBUFG : std_logic;
    signal CLKFB_IN : std_logic;
    signal CLK0_BUF : std_logic;
    signal CLKDV_BUF : std_logic;
    signal GND : std_logic;

    attribute CLK_FEEDBACK : string;
    attribute CLK_FEEDBACK of DCM_INST : label is "1X";
    attribute CLKDV_DIVIDE : real;
    attribute CLKDV_DIVIDE of DCM_INST : label is 5.0;
    attribute CLKFX_DIVIDE : integer;
    attribute CLKFX_DIVIDE of DCM_INST : label is 1;
    attribute CLKFX_MULTIPLY : integer;
    attribute CLKFX_MULTIPLY of DCM_INST : label is 4;
    attribute CLKIN_DIVIDE_BY_2 : boolean;
    attribute CLKIN_DIVIDE_BY_2 of DCM_INST : label is FALSE;
    attribute CLKIN_PERIOD : real;
    attribute CLKIN_PERIOD of DCM_INST : label is 10.0;
    attribute CLKOUT_PHASE_SHIFT : string;
    attribute CLKOUT_PHASE_SHIFT of DCM_INST : label is "FIXED";
    attribute DESKEW_ADJUST : string;
    attribute DESKEW_ADJUST of DCM_INST : label is "SYSTEM_SYNCHRONOUS";
    attribute DFS_FREQUENCY_MODE : string;
    attribute DFS_FREQUENCY_MODE of DCM_INST : label is "LOW";
    attribute DLL_FREQUENCY_MODE : string;
    attribute DLL_FREQUENCY_MODE of DCM_INST : label is "LOW";
    attribute DUTY_CYCLE_CORRECTION : boolean;
```



```

attribute DUTY_CYCLE_CORRECTION of DCM_INST : label is TRUE;
attribute PHASE_SHIFT : integer;
attribute PHASE_SHIFT of DCM_INST : label is 0;
attribute STARTUP_WAIT : boolean;
attribute STARTUP_WAIT of DCM_INST : label is FALSE;

component DCM
-- synopsys translate_off
generic(
    DFS_FREQUENCY_MODE : string := "LOW";
    DLL_FREQUENCY_MODE : string := "LOW";
    DUTY_CYCLE_CORRECTION : boolean := TRUE;
    CLKIN_DIVIDE_BY_2 : boolean := FALSE;
    CLK_FEEDBACK : string := "1X";
    CLKOUT_PHASE_SHIFT : string := "NONE";
    DSS_MODE : string := "NONE";
    FACTORY_JF : bit_vector := X"C080";
    STARTUP_WAIT : boolean := false;
    PHASE_SHIFT : integer := 0;
    CLKFX_MULTIPLY : integer := 4;
    CLKFX_DIVIDE : integer := 1;
    CLKDV_DIVIDE : real := 2.0;
    CLKIN_PERIOD : real := 0.0;
    DESKEW_ADJUST : string := "SYSTEM_SYNCHRONOUS"
);
-- synopsys translate_on
port (
    CLKIN : in std_logic;
    CLKFB : in std_logic;
    RST : in std_logic;
    PSEN : in std_logic;
    PSINCDEC : in std_logic;
    PSCLK : in std_logic;
    DSSEN : in std_logic;
    CLK0 : out std_logic;
    CLK90 : out std_logic;
    CLK180 : out std_logic;
    CLK270 : out std_logic;
    CLKDV : out std_logic;
    CLK2X : out std_logic;
    CLK2X180 : out std_logic;
    CLKFX : out std_logic;
    CLKFX180 : out std_logic;
    STATUS : out std_logic_vector (7 downto 0);
    LOCKED : out std_logic;
    PSDONE : out std_logic
);
end component;
component IBUFG

```

```

        port (
            I : in std_logic;
            O : out std_logic
        );
    end component;
    component BUFG
        port (
            I : in std_logic;
            O : out std_logic
        );
    end component;

begin
    DCM_INST : DCM
-- synopsys translate_off
    Generic map (
        CLK_FEEDBACK => "1X",
        CLKDV_DIVIDE => 5.0,
        CLKFX_DIVIDE => 1,
        CLKFX_MULTIPLY => 4,
        CLKIN_DIVIDE_BY_2 => FALSE,
        CLKIN_PERIOD => 10.0,
        CLKOUT_PHASE_SHIFT => "FIXED",
        DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
        DFS_FREQUENCY_MODE => "LOW",
        DLL_FREQUENCY_MODE => "LOW",
        DUTY_CYCLE_CORRECTION => TRUE,
        PHASE_SHIFT => 0,
        STARTUP_WAIT => FALSE)
-- synopsys translate_on
    port map (
        CLKIN => CLKIN_IBUFG,
        CLKFB => CLKFB_IN,
        RST => RST_IN,
        PSEN => GND,
        PSINCDEC => GND,
        PSCLK => GND,
        DSSEN => GND,
        CLK0 => CLK0_BUF,
        CLKDV => CLKDV_BUF,
        LOCKED => LOCKED_OUT);

    CLKIN_IBUFG_INST : IBUFG
        port map (
            I => CLKIN_IN,
            O => CLKIN_IBUFG);

    CLK0_BUF_INST : BUFG
        port map (

```

```

        I => CLK0_BUF,
        O => CLKFB_IN);

CLKDV_BUFG_INST : BUFG
port map (
    I => CLKDV_BUF,
    O => CLKDV_OUT);

CLK0_OUT <= CLKFB_IN;
GND <= '0';
end STRUCT;

```

➤ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the DCM, was generated. Figure A.151 shows the resulting top level RTL symbol while figure A.151 shows the resulting top level RTL schematic diagram.



Figure A.151 Resulting top level RTL symbol for the synthesized DCM.

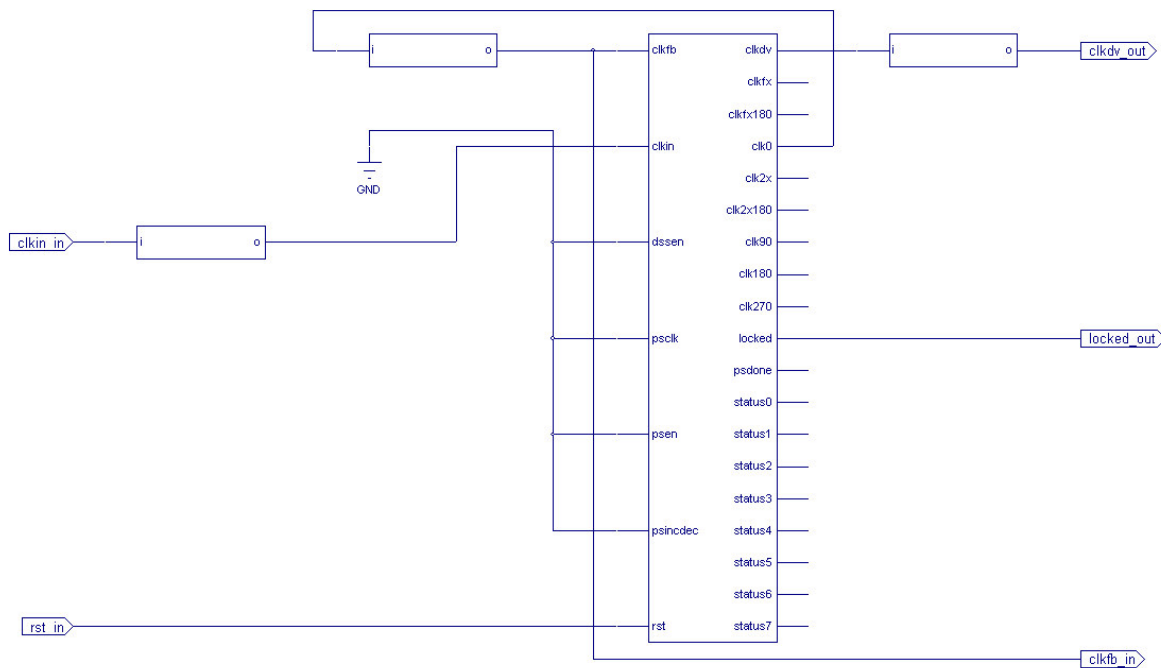


Figure A.152 Resulting top level RTL schematic for the synthesized DCM.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the DCM using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

Design Statistics:

IOs : 5

Cell Usage:

BELS : 1

GND : 1

Clock Buffers : 2

bufg : 2

IO Buffers : 5

IBUF : 1

ibufg : 1

OBUF : 3

DCMs : 1

dcm : 1

Device utilization summary:

Number of bonded IOBs: 5 out of 1108 0%

Number of GCLKs:	2 out of	16	12%
Number of DCMs:	1 out of	12	8%

➤ Place-and-Route onto the FPGA

In figure A.153, FPGA Editor shows the synthesized DCM after place-and-route onto the target Virtex-II FPGA chip. Notice that this is the blue interconnections at the middle of the top of the FPGA chip.

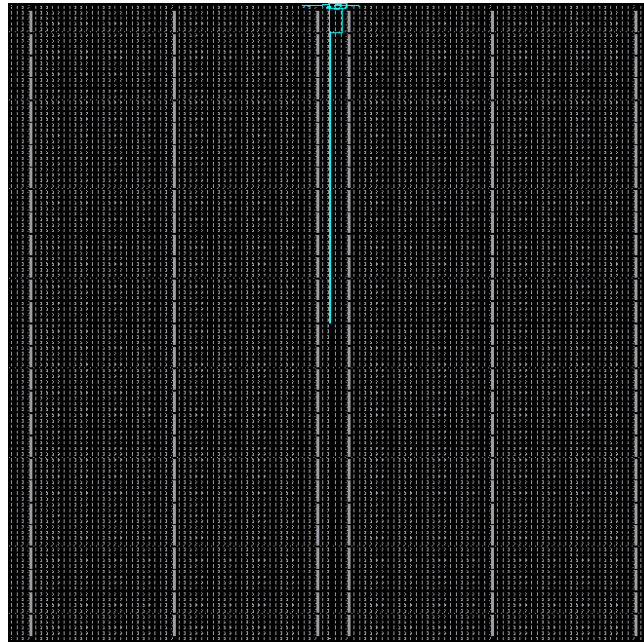


Figure A.153 FPGA Editor showing the synthesized DCM after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure A.154 shows the waveform results of simulating the synthesized DCM VHDL behavioural model in Mentor Graphics ModelSim. In this figure, all the signals are in binary format. As shown in the figure, the frequency of the output clock *Clkdv_out* is fifth that of the output clock *Clk0_out*. This confirms that this DCM is functioning as expected.

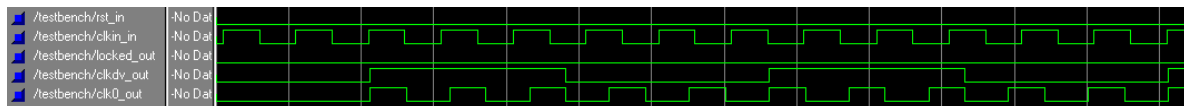


Figure A.154 Results of simulating the synthesized DCM, using ModelSim.

A.4 Summary and Conclusions

This appendix has discussed, in detail, the RTL description, design entry and synthesis, synthesis results, FPGA place-and-route, and simulation of the basic building blocks needed to build the complete datapath.

The next appendix, builds upon the material covered here, to construct the larger datapath sections, then the final complete datapath for the MIPS R2000 microprocessor.