

APPENDIX B

VHDL DESCRIPTION AND SYNTHESIS OF MIPS R2000 COMPLETE DATAPATH

This appendix presents the development of the Register Transfer Level (RTL) description of the complete datapath (without the control unit yet) of the MIPS R2000 microprocessor. The datapath concepts are first reviewed and then followed by my own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of this complete datapath 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 C builds upon to create the VHDL description and synthesis of the finalized full MIPS R2000 microprocessor in chapter 6.

B.1 Introduction

This appendix presents the VHDL description, synthesis and simulation of the MIPS R2000 complete datapath, which is the hardware implementation of the MIPS instruction subset presented in chapter five. This hardware implementation of the MIPS complete datapath is built from combining together the basic building blocks and datapath functional components (described in detail in Appendix A) to first build the larger datapath sections. Then, these datapath sections will be combined together along with the Digital Clock Manager (described in Appendix A) to build the final MIPS complete datapath. This complete datapath does not include the control unit; the latter will be discussed in Appendix C.

The format for presentation of the material in this appendix is the same as that in Appendix A where I take each datapath section and briefly review its RTL description as described in [47] and [48], then follow it with my own work implementing this unit in VHDL, along with its synthesis and simulation. This process follows the design cycle (described in chapter 4) and comprised of the following steps: RTL Description, Design Entry and Synthesis, Synthesis Results, FPGA Device Synthesis Summary, Place-and-Route onto the FPGA, and Simulation Results. Also, the logic conventions and clocking methodology followed in this appendix are detailed in Appendix A (section A.2).

This chapter starts with an overview of the MIPS hardware implementation in section B.2, thereby setting the scene for the material to follow, which is covered in sections B.3, B.4 and B.5. Section B.3 builds upon the material detailed in Appendix A and shows how the larger datapath sections are constructed. Then, section B.4 puts these datapath sections together. Section B.5 finishes this off by putting it all together to build the final complete datapath by combining the datapath sections and the DCM (Digital Clock Manager). Section B.6 concludes this appendix with a summary.

B.2 An Overview of the MIPS Hardware Implementation

For the MIPS instruction subset (reviewed in chapter 5) to be implemented in hardware, much of what needs to be done is similar, regardless of the actual instruction class [47].

For all MIPS instructions, the first two steps of execution are identical [47]:

- ❑ The program counter (PC) sends the instruction address to the instruction memory that contains the code (instructions) and, as a result, the required instruction is fetched from that memory location specified by the PC [47].
- ❑ Referring to the fields of the fetched instruction in order to select which registers (inside the Register File) to read, one or two registers (depending on the instruction class) are read [47].

“After these two steps, the actions required to complete the instruction execution depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the instruction opcode” [47, p.339].

There exist some similarities even across the different instruction classes [47]. For example, all instruction classes utilize the Arithmetic Logical Unit (ALU) after reading the registers [47]. After using the ALU, the operations needed to complete executing the different instruction classes vary significantly [47]. More elaboration on this matter is found on pages 339 and 340 of [47]. Figure B.1 below shows the high-level abstraction view of a MIPS hardware implementation.

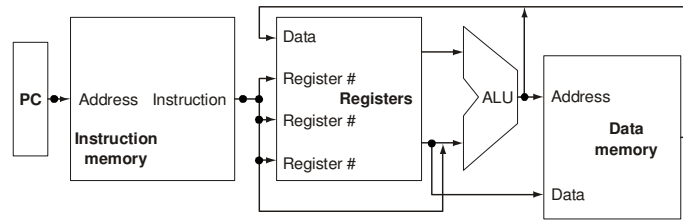


Figure B.1 Abstract view of the hardware implementation of the MIPS instruction subset showing the major functional units and the major connections between them [47, p.340].

In section B.3 that follows, the abstract view in figure B.1 is refined to fill in more and more details, leading up to the MIPS complete datapath being put together.

It is worth noting that the MIPS hardware implementation in this research is based on the simple implementation detailed in [47] that uses one single clock cycle for the execution of each instruction. This means that each instruction begins execution on one rising clock edge and completes execution before the next rising clock edge [47]. However, in the context of this research, instruction execution is spread over a few clock cycles due to the read/write nature of the memory elements (register file, instruction memory, data memory) implemented on the FPGA chip.

B.3 Combining The Basic Building Blocks: The Datapath Sections

This section puts together the datapath basic building blocks, which are discussed in detail in Appendix A, to form the different larger datapath sections. Each of these datapath sections serves a specific yet different purpose. The first datapath section to be covered here is the one dealing with *Instruction Fetch (IF)*. This is followed by the datapath for R-type instructions, the datapath for Load/Store instructions, then the datapath for the Branch instructions, and finally the datapath for the Jump instruction.

B.3.1 The Datapath Section for Instruction Fetch (IF)

➤ RTL Description

The starting point for executing any instruction is to first fetch it from memory using the address value in the program counter. Then, the program counter is incremented so that it points at the next instruction to prepare for its execution [47]. The datapath section for instruction fetch is shown in figure B.2 below.

It is important to note that in figure B.2 the PC is incremented by 1 since my hardware implementation is such that $PC_{next} = PC_{current} + 1$, being word-addressable directly. This is to simplify implementation on the FPGA chip.

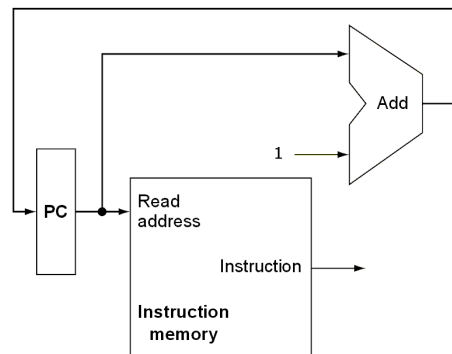


Figure B.2 The datapath portion used for fetching instructions and incrementing the program counter.

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the instruction fetch section of the datapath shown in figure B.2. Figure B.3 shows the final schematic diagram in which there are significant additions to what is illustrated in figure B.2. These necessary additions are:

- ❑ Input signal *Value_of_0*: This is a 1-bit signal connected to the *CarryIn (CI)* input of the 8-bit adder (ADD8). This signal is always assigned the value of 0 (hence the name).
- ❑ Component **mux8b_2to1**: This is an 8-bit/32-bit 2-to-1 multiplexer. As shown in figure B.3, it has been designed specifically for the context of this research as it truncates internally its second input *din1(31:0)* from 32 bits down to 8 bits. Through the use of its control signal *Sel_Addr_Src*, this multiplexer chooses between two sources for the 8-bit address input *a(7:0)* to the instruction RAM. These two sources are either the program counter (an internal source) or an externally applied 32-bit signal called *Write_Addr(31:0)* which is used to preload the instruction memory with the set of instructions (code) necessary for running the simulation for the whole datapath.
- ❑ Input signal *Inst_In(31:0)*: This is the 32-bit instruction input to be loaded into the instruction RAM.
- ❑ Output signals *Ignore_1* and *Ignore_2*: These are 1-bit signals mapped respectively to the *Overflow (OFL)* and *CarryOut (CO)* output signals from the 8-bit adder (ADD8). These signals are of no concern to the context of this research and are therefore ignored.

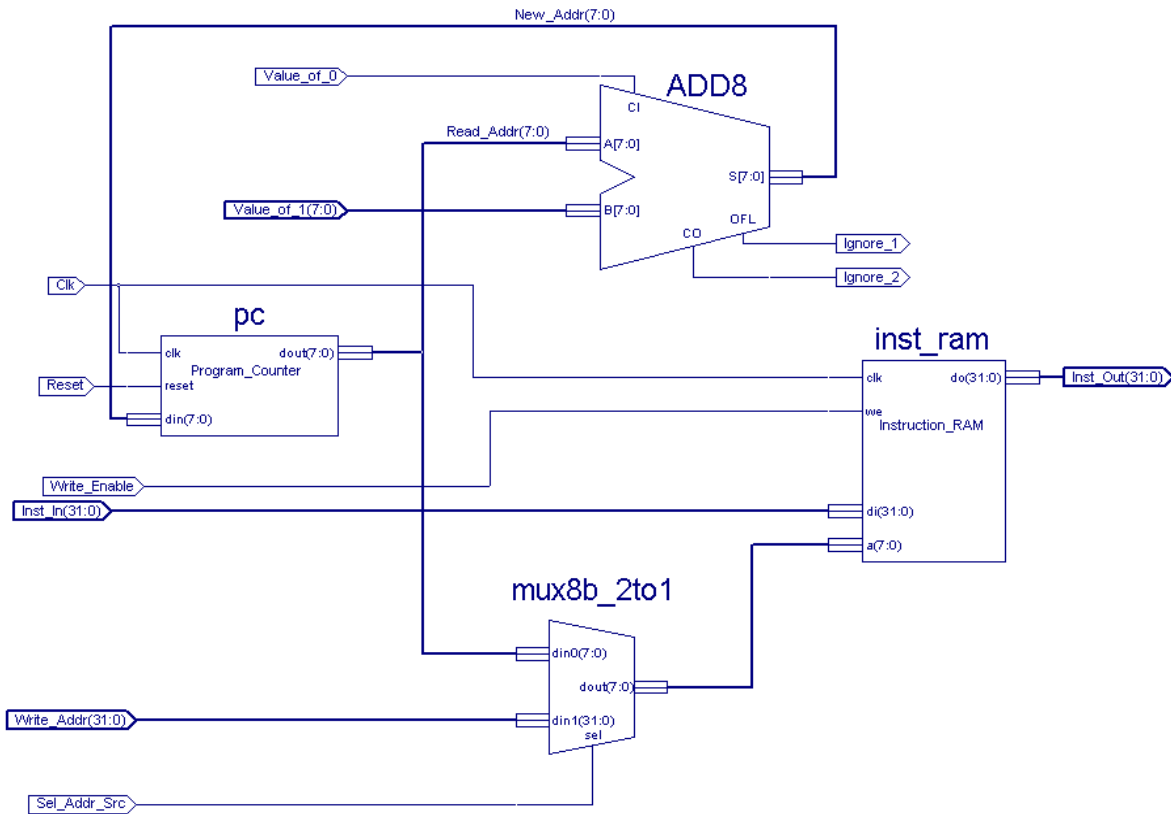


Figure B.3 Schematic diagram design entry in Schematic Editor for the instruction fetch section of the datapath.

After synthesis of the schematic diagram in figure B.3 using XST, the following VHDL code was generated:

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\add8.sch - Fri Jan
16 12:35:02 2004

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 ADD8_MXILINX_figure_5_5 IS
    PORT ( A      : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          B      : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          CI      : IN      STD_LOGIC;
          CO      : OUT     STD_LOGIC;
          OFL     : OUT     STD_LOGIC;
          S      : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));

end ADD8_MXILINX_figure_5_5;

ARCHITECTURE SCHEMATIC OF ADD8_MXILINX_figure_5_5 IS
    SIGNAL C0      : STD_LOGIC;
    SIGNAL C1      : STD_LOGIC;
    SIGNAL C2      : STD_LOGIC;
    SIGNAL C3      : STD_LOGIC;
    SIGNAL C4      : STD_LOGIC;
    SIGNAL C5      : STD_LOGIC;
    SIGNAL C6      : STD_LOGIC;
    SIGNAL C6O     : STD_LOGIC;
    SIGNAL CO_DUMMY : STD_LOGIC;
    SIGNAL I0      : STD_LOGIC;
    SIGNAL I1      : STD_LOGIC;
    SIGNAL I2      : STD_LOGIC;
    SIGNAL I3      : STD_LOGIC;
    SIGNAL I4      : STD_LOGIC;
    SIGNAL I5      : STD_LOGIC;
    SIGNAL I6      : STD_LOGIC;
    SIGNAL I7      : STD_LOGIC;
    SIGNAL dummy   : STD_LOGIC;

    ATTRIBUTE BOX_TYPE      : STRING;
    ATTRIBUTE RLOC          : STRING;
    ATTRIBUTE RLOC OF I_36_16 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_17 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_23 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_22 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_18 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_19 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_20 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_21 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_64 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_107 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_110 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_63 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_58 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_111 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_55 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_62 : LABEL IS "X0Y1";

    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 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_D
  PORT ( CI      :      IN      STD_LOGIC;
        DI      :      IN      STD_LOGIC;
        S       :      IN      STD_LOGIC;
        LO      :      OUT     STD_LOGIC;
        O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY_D : 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 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";
COMPONENT XORCY
  PORT ( CI      :      IN      STD_LOGIC;
        LI      :      IN      STD_LOGIC;
        O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XORCY : COMPONENT IS "BLACK_BOX";
BEGIN
CO <= CO_DUMMY;

I_36_16 : FMAP
  PORT MAP (I1=>A(0), I2=>B(0), I3=>dummy, I4=>dummy, O=>I0);

I_36_17 : FMAP
  PORT MAP (I1=>A(1), I2=>B(1), I3=>dummy, I4=>dummy, O=>I1);

I_36_23 : FMAP
  PORT MAP (I1=>A(7), I2=>B(7), I3=>dummy, I4=>dummy, O=>I7);

I_36_22 : FMAP
  PORT MAP (I1=>A(6), I2=>B(6), I3=>dummy, I4=>dummy, O=>I6);

I_36_18 : FMAP
  PORT MAP (I1=>A(2), I2=>B(2), I3=>dummy, I4=>dummy, O=>I2);

I_36_19 : FMAP
  PORT MAP (I1=>A(3), I2=>B(3), I3=>dummy, I4=>dummy, O=>I3);

I_36_20 : FMAP
  PORT MAP (I1=>A(4), I2=>B(4), I3=>dummy, I4=>dummy, O=>I4);

I_36_21 : FMAP
  PORT MAP (I1=>A(5), I2=>B(5), I3=>dummy, I4=>dummy, O=>I5);

I_36_64 : MUXCY
  PORT MAP (CI=>C6, DI=>A(7), S=>I7, O=>CO_DUMMY);

I_36_107 : MUXCY_D
  PORT MAP (CI=>C5, DI=>A(6), S=>I6, LO=>C6, O=>C6O);

```

```

I_36_110 : MUXCY_L
  PORT MAP (CI=>C4, DI=>A(5), S=>I5, LO=>C5);

I_36_63 : MUXCY_L
  PORT MAP (CI=>C3, DI=>A(4), S=>I4, LO=>C4);

I_36_58 : MUXCY_L
  PORT MAP (CI=>C2, DI=>A(3), S=>I3, LO=>C3);

I_36_111 : MUXCY_L
  PORT MAP (CI=>CI, DI=>A(0), S=>I0, LO=>C0);

I_36_55 : MUXCY_L
  PORT MAP (CI=>C0, DI=>A(1), S=>I1, LO=>C1);

I_36_62 : MUXCY_L
  PORT MAP (CI=>C1, DI=>A(2), S=>I2, LO=>C2);

I_36_239 : XOR2
  PORT MAP (I0=>C60, I1=>CO_DUMMY, O=>OFL);

I_36_230 : XOR2
  PORT MAP (I0=>A(2), I1=>B(2), O=>I2);

I_36_229 : XOR2
  PORT MAP (I0=>A(1), I1=>B(1), O=>I1);

I_36_228 : XOR2
  PORT MAP (I0=>A(0), I1=>B(0), O=>I0);

I_36_224 : XOR2
  PORT MAP (I0=>A(4), I1=>B(4), O=>I4);

I_36_223 : XOR2
  PORT MAP (I0=>A(5), I1=>B(5), O=>I5);

I_36_222 : XOR2
  PORT MAP (I0=>A(6), I1=>B(6), O=>I6);

I_36_225 : XOR2
  PORT MAP (I0=>A(3), I1=>B(3), O=>I3);

I_36_221 : XOR2
  PORT MAP (I0=>A(7), I1=>B(7), O=>I7);

I_36_80 : XORCY
  PORT MAP (CI=>C6, LI=>I7, O=>S(7));

I_36_73 : XORCY
  PORT MAP (CI=>CI, LI=>I0, O=>S(0));

I_36_74 : XORCY
  PORT MAP (CI=>C0, LI=>I1, O=>S(1));

I_36_76 : XORCY
  PORT MAP (CI=>C1, LI=>I2, O=>S(2));

I_36_75 : XORCY
  PORT MAP (CI=>C2, LI=>I3, O=>S(3));

I_36_78 : XORCY
  PORT MAP (CI=>C3, LI=>I4, O=>S(4));

I_36_77 : XORCY
  PORT MAP (CI=>C4, LI=>I5, O=>S(5));

I_36_81 : XORCY
  PORT MAP (CI=>C5, LI=>I6, O=>S(6));

END SCHEMATIC;

-- Vhdl model created from schematic figure_5_5.sch - Fri Jan 16 12:35:02 2004

```

```

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 figure_5_5 IS
    PORT ( Clk          : IN          STD_LOGIC;
           Inst_In      : IN          STD_LOGIC_VECTOR (31 DOWNTO 0);
           Reset        : IN          STD_LOGIC;
           Sel_Addr_Src : IN          STD_LOGIC;
           Value_of_0    : IN          STD_LOGIC;
           Value_of_1    : IN          STD_LOGIC_VECTOR (7 DOWNTO 0);
           Write_Addr    : IN          STD_LOGIC_VECTOR (31 DOWNTO 0);
           Write_Enable  : IN          STD_LOGIC;
           Ignore_1     : OUT         STD_LOGIC;
           Ignore_2     : OUT         STD_LOGIC;
           Inst_Out      : OUT         STD_LOGIC_VECTOR (31 DOWNTO 0));

END figure_5_5;

ARCHITECTURE SCHEMATIC OF figure_5_5 IS
    SIGNAL New_Addr : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL Read_Addr : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL XLXN_28 : STD_LOGIC_VECTOR (7 DOWNTO 0);

    ATTRIBUTE BOX_TYPE : STRING;
    ATTRIBUTE U_SET : STRING ;
    ATTRIBUTE U_SET OF Add8 : LABEL IS "Add8_0";

    COMPONENT ADD8_MXILINK_figure_5_5
        PORT ( A : IN          STD_LOGIC_VECTOR (7 DOWNTO 0);
              B : IN          STD_LOGIC_VECTOR (7 DOWNTO 0);
              CI : IN          STD_LOGIC;
              CO : OUT         STD_LOGIC;
              OFL : OUT        STD_LOGIC;
              S : OUT         STD_LOGIC_VECTOR (7 DOWNTO 0));
    END COMPONENT;

    COMPONENT inst_ram
        PORT ( clk : IN          STD_LOGIC;
              we : IN          STD_LOGIC;
              a : IN          STD_LOGIC_VECTOR (7 DOWNTO 0);
              di : IN          STD_LOGIC_VECTOR (31 DOWNTO 0);
              do : OUT         STD_LOGIC_VECTOR (31 DOWNTO 0));
    END COMPONENT;

    COMPONENT mux8b_2to1
        PORT ( din0 : IN          STD_LOGIC_VECTOR (7 DOWNTO 0);
              din1 : IN          STD_LOGIC_VECTOR (31 DOWNTO 0);
              dout : OUT         STD_LOGIC_VECTOR (7 DOWNTO 0);
              sel : IN          STD_LOGIC);
    END COMPONENT;

    COMPONENT pc
        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 COMPONENT;

BEGIN

    Add8 : ADD8_MXILINK_figure_5_5
        PORT MAP (A(7)=>Read_Addr(7), A(6)=>Read_Addr(6), A(5)=>Read_Addr(5),
                  A(4)=>Read_Addr(4), A(3)=>Read_Addr(3), A(2)=>Read_Addr(2),
                  A(1)=>Read_Addr(1), A(0)=>Read_Addr(0), B(7)=>Value_of_1(7),
                  B(6)=>Value_of_1(6), B(5)=>Value_of_1(5), B(4)=>Value_of_1(4),
                  B(3)=>Value_of_1(3), B(2)=>Value_of_1(2), B(1)=>Value_of_1(1),
                  B(0)=>Value_of_1(0), CI=>Value_of_0, CO=>Ignore_2, OFL=>Ignore_1,
                  S(7)=>New_Addr(7), S(6)=>New_Addr(6), S(5)=>New_Addr(5),

```

```

S(4)=>New_Addr(4), S(3)=>New_Addr(3), S(2)=>New_Addr(2),
S(1)=>New_Addr(1), S(0)=>New_Addr(0));

Instruction_RAM : inst_ram
PORT MAP (clk=>Clk, we=>Write_Enable, a(7)=>XLXN_28(7), a(6)=>XLXN_28(6),
a(5)=>XLXN_28(5), a(4)=>XLXN_28(4), a(3)=>XLXN_28(3), a(2)=>XLXN_28(2),
a(1)=>XLXN_28(1), a(0)=>XLXN_28(0), di(31)=>Inst_In(31),
di(30)=>Inst_In(30), di(29)=>Inst_In(29), di(28)=>Inst_In(28),
di(27)=>Inst_In(27), di(26)=>Inst_In(26), di(25)=>Inst_In(25),
di(24)=>Inst_In(24), di(23)=>Inst_In(23), di(22)=>Inst_In(22),
di(21)=>Inst_In(21), di(20)=>Inst_In(20), di(19)=>Inst_In(19),
di(18)=>Inst_In(18), di(17)=>Inst_In(17), di(16)=>Inst_In(16),
di(15)=>Inst_In(15), di(14)=>Inst_In(14), di(13)=>Inst_In(13),
di(12)=>Inst_In(12), di(11)=>Inst_In(11), di(10)=>Inst_In(10),
di(9)=>Inst_In(9), di(8)=>Inst_In(8), di(7)=>Inst_In(7),
di(6)=>Inst_In(6), di(5)=>Inst_In(5), di(4)=>Inst_In(4),
di(3)=>Inst_In(3), di(2)=>Inst_In(2), di(1)=>Inst_In(1),
di(0)=>Inst_In(0), do(31)=>Inst_Out(31), do(30)=>Inst_Out(30),
do(29)=>Inst_Out(29), do(28)=>Inst_Out(28), do(27)=>Inst_Out(27),
do(26)=>Inst_Out(26), do(25)=>Inst_Out(25), do(24)=>Inst_Out(24),
do(23)=>Inst_Out(23), do(22)=>Inst_Out(22), do(21)=>Inst_Out(21),
do(20)=>Inst_Out(20), do(19)=>Inst_Out(19), do(18)=>Inst_Out(18),
do(17)=>Inst_Out(17), do(16)=>Inst_Out(16), do(15)=>Inst_Out(15),
do(14)=>Inst_Out(14), do(13)=>Inst_Out(13), do(12)=>Inst_Out(12),
do(11)=>Inst_Out(11), do(10)=>Inst_Out(10), do(9)=>Inst_Out(9),
do(8)=>Inst_Out(8), do(7)=>Inst_Out(7), do(6)=>Inst_Out(6),
do(5)=>Inst_Out(5), do(4)=>Inst_Out(4), do(3)=>Inst_Out(3),
do(2)=>Inst_Out(2), do(1)=>Inst_Out(1), do(0)=>Inst_Out(0));

XLXI_5 : mux8b_2to1
PORT MAP (din0(7)=>Read_Addr(7), din0(6)=>Read_Addr(6),
din0(5)=>Read_Addr(5), din0(4)=>Read_Addr(4), din0(3)=>Read_Addr(3),
din0(2)=>Read_Addr(2), din0(1)=>Read_Addr(1), din0(0)=>Read_Addr(0),
din1(31)=>Write_Addr(31), din1(30)=>Write_Addr(30),
din1(29)=>Write_Addr(29), din1(28)=>Write_Addr(28),
din1(27)=>Write_Addr(27), din1(26)=>Write_Addr(26),
din1(25)=>Write_Addr(25), din1(24)=>Write_Addr(24),
din1(23)=>Write_Addr(23), din1(22)=>Write_Addr(22),
din1(21)=>Write_Addr(21), din1(20)=>Write_Addr(20),
din1(19)=>Write_Addr(19), din1(18)=>Write_Addr(18),
din1(17)=>Write_Addr(17), din1(16)=>Write_Addr(16),
din1(15)=>Write_Addr(15), din1(14)=>Write_Addr(14),
din1(13)=>Write_Addr(13), din1(12)=>Write_Addr(12),
din1(11)=>Write_Addr(11), din1(10)=>Write_Addr(10),
din1(9)=>Write_Addr(9), din1(8)=>Write_Addr(8), din1(7)=>Write_Addr(7),
din1(6)=>Write_Addr(6), din1(5)=>Write_Addr(5), din1(4)=>Write_Addr(4),
din1(3)=>Write_Addr(3), din1(2)=>Write_Addr(2), din1(1)=>Write_Addr(1),
din1(0)=>Write_Addr(0), dout(7)=>XLXN_28(7), dout(6)=>XLXN_28(6),
dout(5)=>XLXN_28(5), dout(4)=>XLXN_28(4), dout(3)=>XLXN_28(3),
dout(2)=>XLXN_28(2), dout(1)=>XLXN_28(1), dout(0)=>XLXN_28(0),
sel=>Sel_Addr_Src);

Program_Counter : pc
PORT MAP (clk=>Clk, reset=>Reset, din(7)=>New_Addr(7),
din(6)=>New_Addr(6), din(5)=>New_Addr(5), din(4)=>New_Addr(4),
din(3)=>New_Addr(3), din(2)=>New_Addr(2), din(1)=>New_Addr(1),
din(0)=>New_Addr(0), dout(7)=>Read_Addr(7), dout(6)=>Read_Addr(6),
dout(5)=>Read_Addr(5), dout(4)=>Read_Addr(4), dout(3)=>Read_Addr(3),
dout(2)=>Read_Addr(2), dout(1)=>Read_Addr(1), dout(0)=>Read_Addr(0));

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the instruction fetch section of the datapath, was generated. Figure B.4 shows the resulting top level RTL symbol while figure B.5 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are already covered in detail in Appendix A.

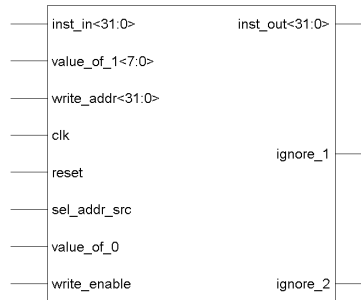


Figure B.4 Resulting top level RTL symbol for the instruction fetch section of the datapath.

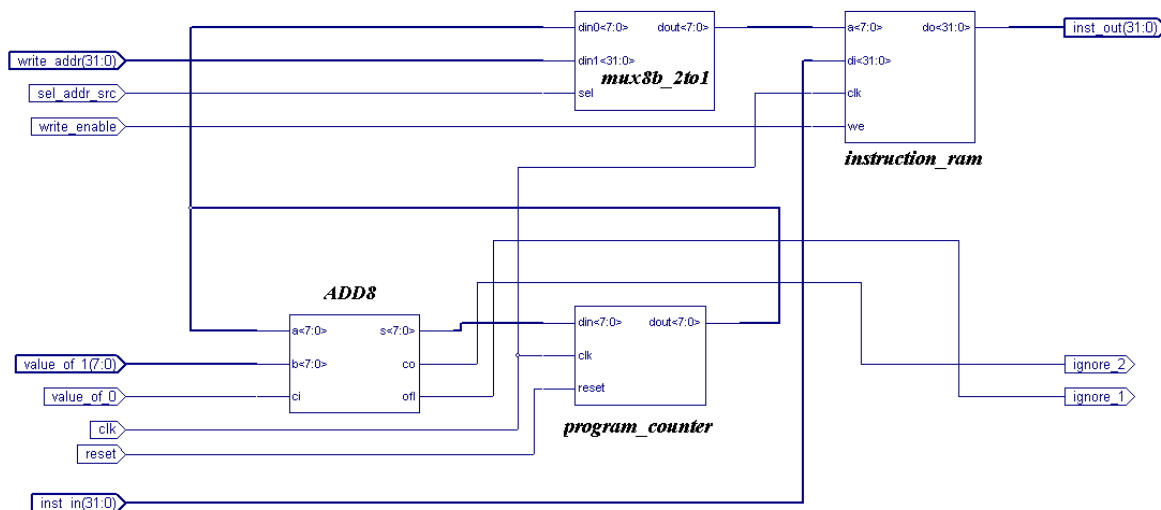


Figure B.5 Resulting top level RTL Schematic for the instruction fetch section of the datapath.

➤ FPGA Device Synthesis Summary

After the hardware implementation for the instruction fetch section of the datapath, 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                      : 111

Macro Statistics:
# RAM                      : 1
# 256x32-bit single-port block RAM: 1
# Registers                : 1
# 8-bit register           : 1
# Tristates                : 2
# 8-bit tristate buffer    : 2

Cell Usage:
# BELS                     : 29
# GND                      : 2
# LUT1                     : 1
# muxcy                    : 1
# muxcy_d                  : 1
# muxcy_l                  : 6
# VCC                      : 1
# xor2                     : 9
```

```

#      xorcy                      : 8
# FlipFlops/Latches              : 8
#      FDC                       : 8
# RAMS                          : 1
#      RAMB16_S36                : 1
# Tri-States                    : 16
#      BUFT                      : 16
# Clock Buffers                 : 1
#      BUFGP                     : 1
# IO Buffers                    : 86
#      IBUF                     : 52
#      OBUF                     : 34
# Others                        : 8
#      fmap                      : 8

Device utilization summary:

Number of Slices:                10 out of 46592    0%
Number of Slice Flip Flops:      8 out of 93184    0%
Number of 4 input LUTs:         1 out of 93184    0%
Number of bonded IOBs:          86 out of 1108     7%
Number of TBUFs:                16 out of 23296    0%
Number of BRAMs:                1 out of 168       0%
Number of GCLKs:                1 out of 16        6%

Timing Summary:

Minimum period: 4.873ns (Maximum Frequency: 205.212MHz)
Minimum input arrival time before clock: 5.169ns
Maximum output required time after clock: 10.618ns
Maximum combinational path delay: 10.493ns

```

➤ Place-and-Route onto the FPGA

In figure B.6, FPGA Editor shows the synthesized instruction fetch section of the datapath 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.

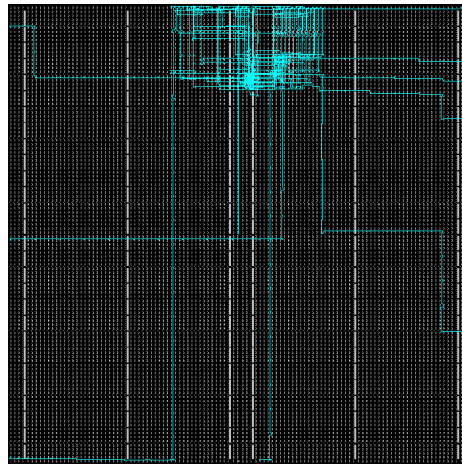


Figure B.6 FPGA Editor showing the synthesized instruction fetch section of the datapath after place-and-route onto the target Virtex-II FPGA chip.

➤ Simulation Results

Figure B.7 shows the waveform results of simulating the instruction fetch VHDL behavioural model in Mentor Graphics ModelSim. In this figure, the input signals *clk*, *reset*, *sel_addr_src*, *value_of_0*, *value_of_1*, and *write_enable* are all in binary format while the input signals *inst_in*, and *write_addr*

and the output signal *inst_out* are in decimal format. Both signals *inst_in* and *inst_out* have been presented here in decimal format for clarity and testing purposes only. However, for the correct operation of the MIPS datapath, they are required to be presented in 32-bit format, as will be seen later on.

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

❑ **During part 1 of the waveform (clock cycles 1 to 8):**

- This is the pre-loading phase: Before executing any code, its instructions (32 bits wide each) must be pre-loaded into Inst_RAM.
- In this case, memory locations 0 to 7 are being loaded with eight instructions, in sequence.
- This is accomplished by asserting high both control signals *write_enable* and *sel_addr_src*. This allows the address input of the instruction memory to be assigned by an external input signal *write_addr*, which is driven by a VHDL testbench host environment in the context of this research. This is covered in detail in upcoming chapters.
- Also, the control signal *reset* is asserted high. This resets the contents of the program counter register.
- During this preloading phase, reading the output from the instruction memory for any address location returns the un-initialized value of **xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**.

❑ **During part 2 of the waveform (clock cycles 9 onwards):**

- This is the phase after the preloading (cycles 1 to 8) has completed and when the actual simulation starts at the 9th clock cycle.
- In this case, memory locations 0 to 7 are being read out in sequence. This is because the program counter and the adder are working in tandem to increment the address input of the instruction memory one consecutive location at a time during each consecutive clock cycle. This in turn is returning the value (instruction) stored in that address location.
- Also, all three control signals *reset*, *sel_addr_src* and *write_enable* are de-asserted.
- Note that beyond the 16th clock cycle, there are no more instructions stored in the instruction memory after address location 7. Therefore, when address location 8 is being read during the 17th clock cycle, it is returning the uninitialized value of **xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**.

It is clear that the resulting synthesized hardware functions according to the specified behaviour of instruction fetch section of the datapath. This concludes the design cycle for this datapath section.

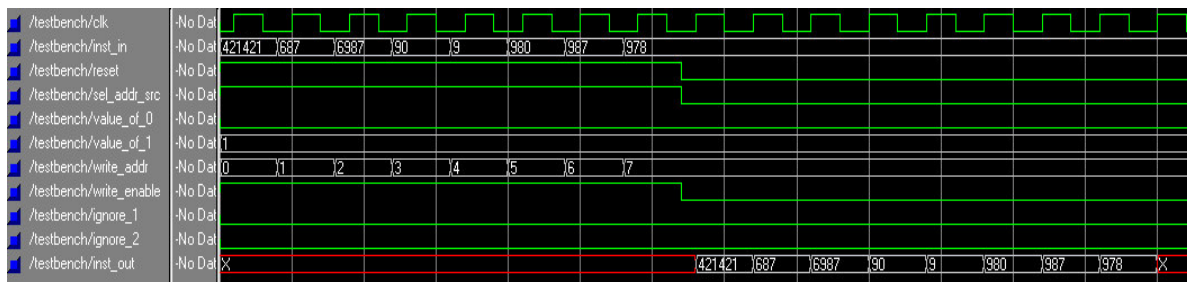


Figure B.7 Results of simulating the synthesized instruction fetch section of the datapath using ModelSim.

B.3.2 The Datapath Section for R-Type Instructions

➤ RTL Description

The datapath section for the R-type instructions (arithmetic-logical instructions) is discussed in detail on pages 344 to 347 of [47]. Figure B.8 below shows that this datapath section is comprised of the register file and the ALU. The ALU is 32-bits wide and is controlled by a 3-bit *ALU operation* control signal. For the register file, since all three 5-bit register numbers (register specifiers) *Read register 1*, *Read register 2*, and *Write register* come from fields in the supplied instruction, figure B.8 shows that this instruction, which is supplied by the instruction fetch stage shown in figure B.2 earlier, is connected to the register number inputs of the register file [47].

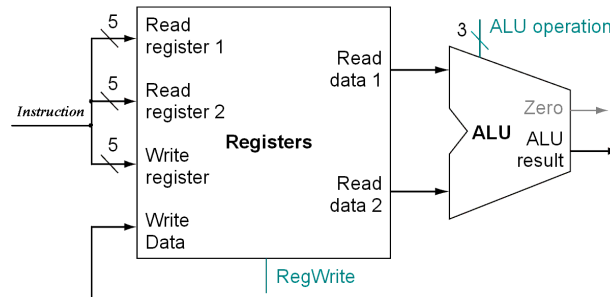


Figure B.8 The datapath section for R-type instructions [47, p.347].

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the datapath section for R-type instructions shown in figure B.8. Figure B.9 shows the final schematic diagram. It is clear from figure B.9 that there are significant additions to what is illustrated in figure B.8. These necessary additions are:

❑ Components:

- Component **mux5b_2to1**: This is a 5-bit 2-to-1 multiplexor. Through the use of its control signal *Mux_Sel_RF_Num_Preload*, this multiplexor chooses between two sources for the 5-bit *write_reg(4:0)* input port of the register file. These two sources are either the destination register field *rd(4:0)* from the instruction itself (fed through the instruction splitter) or an externally applied 32-bit signal called *Reg_Write_Num_Preload(4:0)*, which is used to select the destination register number for preloading the register file with the set of register operands necessary for running the simulation for the whole datapath.
- Component **mux32b_2to1**: This is a 32-bit 2-to-1 multiplexor. Through the use of its control signal *Mux_Sel_RF_Din_Preload*, this multiplexor chooses between two sources for the 32-bit *write_data(31:0)* input port of the register file. These two sources are either the ALU output *Result(31:0)* or an externally applied 32-bit signal called *Reg_Write_Din_Preload(31:0)*, which is used to preload the register file with the set of register operands necessary for running the simulation for the whole datapath.
- Component **instruction_splitter**: This component is made up of bus taps to extract different fields from the 32-bit instruction input *Instruction(31:0)* and feed them into the different sections of the datapath.

□ **Input Signals:**

- Input signal *Value_of_Zero*: This is connected to the *less_zero* input port of the ALU and is part of the implementation for the SLT instruction (as detailed in Appendix A). This is not the *Value_of_0* input signal we've seen earlier in figure B.3, even though both these signals are 1-bit and always assigned the value of 0 (hence the names).

□ **Control Signals:**

- Control signal *ALU_Mux_En*: This is a 1-bit signal connected to the *mux_enable* input port of the ALU and is part of the Xilinx library implementation for a multiplexor.
- Control signal *ALU_Operation(2:0)*: This is the 3-bit control signal for the ALU as described in detail in Appendix A.
- Control signal *RF_En_Read_Write*: This is a 1-bit signal connected to the *enable* input port of the register file and is part of the Xilinx library implementation for the BlockRAM from which the register file is synthesized. This signal is the implementation of the *RegWrite* signal shown in figure B.8. The register file implementation for this research is synchronous read and synchronous write (refer to Appendix A for details) and, therefore, this control signal must be asserted high whether the register file is read and/or written.

□ **Output Probe Signals:**

These output ports are test probes extracted from internal signals for debugging purposes. As the design hierarchy gets more complicated, these probes become increasingly necessary as they make the testing and debugging process more manageable. From this point onwards in the research, these probe signals will be implemented more often as the design gets bigger and larger.

The output probe signals added to figure B.9 are:

- Probe signal *rd(4:0)*: This is the 5-bit destination register specifier extracted from the input 32-bit instruction and fed through the instruction splitter.
- Probe signal *rs(4:0)*: This is the first 5-bit source register specifier extracted from the input 32-bit instruction and fed through the instruction splitter.
- Probe signal *rt(4:0)*: This is the second 5-bit source register specifier extracted from the input 32-bit instruction and fed through the instruction splitter.
- Probe signal *A_in(31:0)*: This is the 32-bit value read out from the RF first data output port *read_data_1(31:0)* and fed into the ALU first data input port *A(31:0)*.
- Probe signal *B_in(31:0)*: This is the 32-bit value read out from the RF second data output port *read_data_2(31:0)* and fed into the ALU second data input port *B(31:0)*.
- Probe signal *WriteData(31:0)*: This is the 32-bit value to be written into the *write_data(31:0)* input port of the register file.
- Probe signal *WriteReg(4:0)*: This signal is connected to the *write_reg(4:0)* input port of the register file and represents the 5-bit register specifier for selecting the destination register to be written to.
- Probe signal *ALU_to_RF(31:0)*: This is the 32-bit output result *Result(31:0)* generated by the ALU and fed back into the RF.

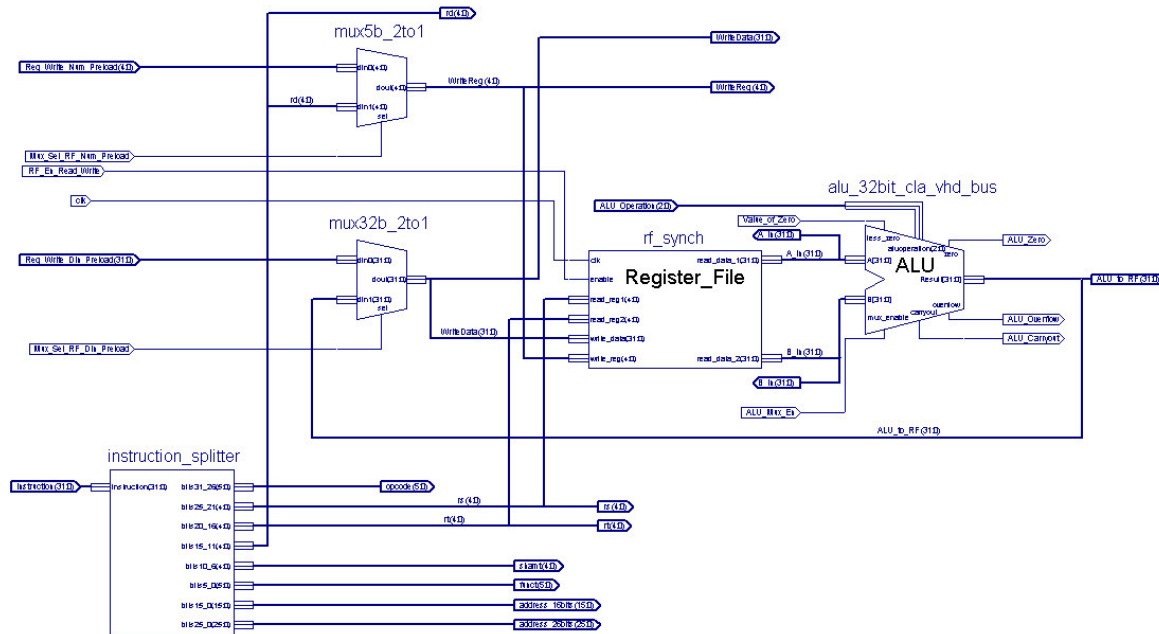


Figure B.9 Schematic diagram design entry in Schematic Editor for the datapath section for R-type instructions.

After synthesis of the schematic diagram in figure B.9 using XST, the following VHDL code was generated:

- Vhdl model created from schematic figure_5_7.sch - Wed Jun 21 22:11:52 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 figure_5_7 IS
    PORT (
        ALU_Mux_En          : IN        STD_LOGIC;
        ALU_Operation       : IN        STD_LOGIC_VECTOR (2 DOWNTO 0);
        Instruction          : IN        STD_LOGIC_VECTOR (31 DOWNTO 0);
        Mux_Sel_RF_Din_Preload : IN        STD_LOGIC;
        Mux_Sel_RF_Num_Preload : IN        STD_LOGIC;
        RF_En_Read_Write    : IN        STD_LOGIC;
        Reg_Write_Din_Preload : IN        STD_LOGIC_VECTOR (31 DOWNTO 0);
        Reg_Write_Num_Preload : IN        STD_LOGIC_VECTOR (4 DOWNTO 0);
        Value_of_Zero       : IN        STD_LOGIC;
        clk                  : IN        STD_LOGIC;
        ALU_Carryout        : OUT        STD_LOGIC;
        ALU_Overflow        : OUT        STD_LOGIC;
        ALU_Zero            : OUT        STD_LOGIC;
        ALU_to_RF          : OUT        STD_LOGIC_VECTOR (31 DOWNTO 0);
        A_in                : OUT        STD_LOGIC_VECTOR (31 DOWNTO 0);
        B_in                : OUT        STD_LOGIC_VECTOR (31 DOWNTO 0);
        WriteData           : OUT        STD_LOGIC_VECTOR (31 DOWNTO 0);
        WriteReg            : OUT        STD_LOGIC_VECTOR (4 DOWNTO 0);
        address_16bits      : OUT        STD_LOGIC_VECTOR (15 DOWNTO 0);
        address_26bits      : OUT        STD_LOGIC_VECTOR (25 DOWNTO 0);
        funct               : OUT        STD_LOGIC_VECTOR (5 DOWNTO 0);
        opcode              : OUT        STD_LOGIC_VECTOR (5 DOWNTO 0);
        rd                  : OUT        STD_LOGIC_VECTOR (4 DOWNTO 0);
        rs                  : OUT        STD_LOGIC_VECTOR (4 DOWNTO 0);
        rt                  : OUT        STD_LOGIC_VECTOR (4 DOWNTO 0);
        shamt               : OUT        STD_LOGIC_VECTOR (4 DOWNTO 0));

```

```

end figure_5_7;

ARCHITECTURE SCHEMATIC OF figure_5_7 IS
    SIGNAL ALU_to_RF_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL A_in_DUMMY      : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL B_in_DUMMY      : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL WriteData_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL WriteReg_DUMMY  : STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL rd_DUMMY        : STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL rs_DUMMY        : STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL rt_DUMMY        : STD_LOGIC_VECTOR (4 DOWNTO 0);

    ATTRIBUTE BOX_TYPE : STRING;

    COMPONENT alu_32bit_cla_vhd_bus
        PORT ( A      : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              B      : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              less_zero : IN      STD_LOGIC;
              carryout : OUT     STD_LOGIC;
              overflow  : OUT     STD_LOGIC;
              Result    : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
              zero      : OUT     STD_LOGIC;
              mux_enable : IN      STD_LOGIC;
              aluoperation : IN    STD_LOGIC_VECTOR (2 DOWNTO 0));
    END COMPONENT;

    COMPONENT instruction_splitter
        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 COMPONENT;

    COMPONENT mux32b_2to1
        PORT ( din0 : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              din1 : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              dout  : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
              sel   : IN      STD_LOGIC);
    END COMPONENT;

    COMPONENT mux5b_2to1
        PORT ( sel : IN      STD_LOGIC;
              din0 : IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
              din1 : IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
              dout  : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0));
    END COMPONENT;

    COMPONENT rf_synch
        PORT ( clk      : IN      STD_LOGIC;
              enable    : IN      STD_LOGIC;
              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);
              read_data_1 : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
              read_data_2 : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
    END COMPONENT;

BEGIN
    ALU_to_RF <= ALU_to_RF_DUMMY;
    A_in      <= A_in_DUMMY;
    B_in      <= B_in_DUMMY;
    WriteData <= WriteData_DUMMY;
    WriteReg  <= WriteReg_DUMMY;
    rd        <= rd_DUMMY;
    rs        <= rs_DUMMY;
    rt        <= rt_DUMMY;

```

```

ALU : alu_32bit_cla_vhd_bus
    PORT MAP (A(31)=>A_in_DUMMY(31), A(30)=>A_in_DUMMY(30),
              A(29)=>A_in_DUMMY(29), A(28)=>A_in_DUMMY(28), A(27)=>A_in_DUMMY(27),
              A(26)=>A_in_DUMMY(26), A(25)=>A_in_DUMMY(25), A(24)=>A_in_DUMMY(24),
              A(23)=>A_in_DUMMY(23), A(22)=>A_in_DUMMY(22), A(21)=>A_in_DUMMY(21),
              A(20)=>A_in_DUMMY(20), A(19)=>A_in_DUMMY(19), A(18)=>A_in_DUMMY(18),
              A(17)=>A_in_DUMMY(17), A(16)=>A_in_DUMMY(16), A(15)=>A_in_DUMMY(15),
              A(14)=>A_in_DUMMY(14), A(13)=>A_in_DUMMY(13), A(12)=>A_in_DUMMY(12),
              A(11)=>A_in_DUMMY(11), A(10)=>A_in_DUMMY(10), A(9)=>A_in_DUMMY(9),
              A(8)=>A_in_DUMMY(8), A(7)=>A_in_DUMMY(7), A(6)=>A_in_DUMMY(6),
              A(5)=>A_in_DUMMY(5), A(4)=>A_in_DUMMY(4), A(3)=>A_in_DUMMY(3),
              A(2)=>A_in_DUMMY(2), A(1)=>A_in_DUMMY(1), A(0)=>A_in_DUMMY(0),
              B(31)=>B_in_DUMMY(31), B(30)=>B_in_DUMMY(30), B(29)=>B_in_DUMMY(29),
              B(28)=>B_in_DUMMY(28), B(27)=>B_in_DUMMY(27), B(26)=>B_in_DUMMY(26),
              B(25)=>B_in_DUMMY(25), B(24)=>B_in_DUMMY(24), B(23)=>B_in_DUMMY(23),
              B(22)=>B_in_DUMMY(22), B(21)=>B_in_DUMMY(21), B(20)=>B_in_DUMMY(20),
              B(19)=>B_in_DUMMY(19), B(18)=>B_in_DUMMY(18), B(17)=>B_in_DUMMY(17),
              B(16)=>B_in_DUMMY(16), B(15)=>B_in_DUMMY(15), B(14)=>B_in_DUMMY(14),
              B(13)=>B_in_DUMMY(13), B(12)=>B_in_DUMMY(12), B(11)=>B_in_DUMMY(11),
              B(10)=>B_in_DUMMY(10), B(9)=>B_in_DUMMY(9), B(8)=>B_in_DUMMY(8),
              B(7)=>B_in_DUMMY(7), B(6)=>B_in_DUMMY(6), B(5)=>B_in_DUMMY(5),
              B(4)=>B_in_DUMMY(4), B(3)=>B_in_DUMMY(3), B(2)=>B_in_DUMMY(2),
              B(1)=>B_in_DUMMY(1), B(0)=>B_in_DUMMY(0), less_zero=>Value_of_Zero,
              carryout=>ALU_Carryout, overflow=>ALU_Overflow,
              Result(31)=>ALU_to_RF_DUMMY(31), Result(30)=>ALU_to_RF_DUMMY(30),
              Result(29)=>ALU_to_RF_DUMMY(29), Result(28)=>ALU_to_RF_DUMMY(28),
              Result(27)=>ALU_to_RF_DUMMY(27), Result(26)=>ALU_to_RF_DUMMY(26),
              Result(25)=>ALU_to_RF_DUMMY(25), Result(24)=>ALU_to_RF_DUMMY(24),
              Result(23)=>ALU_to_RF_DUMMY(23), Result(22)=>ALU_to_RF_DUMMY(22),
              Result(21)=>ALU_to_RF_DUMMY(21), Result(20)=>ALU_to_RF_DUMMY(20),
              Result(19)=>ALU_to_RF_DUMMY(19), Result(18)=>ALU_to_RF_DUMMY(18),
              Result(17)=>ALU_to_RF_DUMMY(17), Result(16)=>ALU_to_RF_DUMMY(16),
              Result(15)=>ALU_to_RF_DUMMY(15), Result(14)=>ALU_to_RF_DUMMY(14),
              Result(13)=>ALU_to_RF_DUMMY(13), Result(12)=>ALU_to_RF_DUMMY(12),
              Result(11)=>ALU_to_RF_DUMMY(11), Result(10)=>ALU_to_RF_DUMMY(10),
              Result(9)=>ALU_to_RF_DUMMY(9), Result(8)=>ALU_to_RF_DUMMY(8),
              Result(7)=>ALU_to_RF_DUMMY(7), Result(6)=>ALU_to_RF_DUMMY(6),
              Result(5)=>ALU_to_RF_DUMMY(5), Result(4)=>ALU_to_RF_DUMMY(4),
              Result(3)=>ALU_to_RF_DUMMY(3), Result(2)=>ALU_to_RF_DUMMY(2),
              Result(1)=>ALU_to_RF_DUMMY(1), Result(0)=>ALU_to_RF_DUMMY(0),
              zero=>ALU_Zero, mux_enable=>ALU_Mux_En,
              aluoperation(2)=>ALU_Operation(2), aluoperation(1)=>ALU_Operation(1),
              aluoperation(0)=>ALU_Operation(0));

XLXI_4 : instruction_splitter
    PORT MAP (instruction(31)=>Instruction(31),
              instruction(30)=>Instruction(30), instruction(29)=>Instruction(29),
              instruction(28)=>Instruction(28), instruction(27)=>Instruction(27),
              instruction(26)=>Instruction(26), instruction(25)=>Instruction(25),
              instruction(24)=>Instruction(24), instruction(23)=>Instruction(23),
              instruction(22)=>Instruction(22), instruction(21)=>Instruction(21),
              instruction(20)=>Instruction(20), instruction(19)=>Instruction(19),
              instruction(18)=>Instruction(18), instruction(17)=>Instruction(17),
              instruction(16)=>Instruction(16), instruction(15)=>Instruction(15),
              instruction(14)=>Instruction(14), instruction(13)=>Instruction(13),
              instruction(12)=>Instruction(12), instruction(11)=>Instruction(11),
              instruction(10)=>Instruction(10), instruction(9)=>Instruction(9),
              instruction(8)=>Instruction(8), instruction(7)=>Instruction(7),
              instruction(6)=>Instruction(6), instruction(5)=>Instruction(5),
              instruction(4)=>Instruction(4), instruction(3)=>Instruction(3),
              instruction(2)=>Instruction(2), instruction(1)=>Instruction(1),
              instruction(0)=>Instruction(0), bits31_26(5)=>opcode(5),
              bits31_26(4)=>opcode(4), bits31_26(3)=>opcode(3),
              bits31_26(2)=>opcode(2), bits31_26(1)=>opcode(1),
              bits31_26(0)=>opcode(0), bits25_21(4)=>rs_DUMMY(4),
              bits25_21(3)=>rs_DUMMY(3), bits25_21(2)=>rs_DUMMY(2),
              bits25_21(1)=>rs_DUMMY(1), bits25_21(0)=>rs_DUMMY(0),
              bits20_16(4)=>rt_DUMMY(4), bits20_16(3)=>rt_DUMMY(3),
              bits20_16(2)=>rt_DUMMY(2), bits20_16(1)=>rt_DUMMY(1),
              bits20_16(0)=>rt_DUMMY(0), bits15_11(4)=>rd_DUMMY(4),
              bits15_11(3)=>rd_DUMMY(3), bits15_11(2)=>rd_DUMMY(2),
              bits15_11(1)=>rd_DUMMY(1), bits15_11(0)=>rd_DUMMY(0),
              bits10_6(4)=>shamt(4), bits10_6(3)=>shamt(3), bits10_6(2)=>shamt(2),

```

```

bits10_6(1)=>shamt(1), bits10_6(0)=>shamt(0), bits5_0(5)=>funct(5),
bits5_0(4)=>funct(4), bits5_0(3)=>funct(3), bits5_0(2)=>funct(2),
bits5_0(1)=>funct(1), bits5_0(0)=>funct(0),
bits15_0(15)=>address_16bits(15), bits15_0(14)=>address_16bits(14),
bits15_0(13)=>address_16bits(13), bits15_0(12)=>address_16bits(12),
bits15_0(11)=>address_16bits(11), bits15_0(10)=>address_16bits(10),
bits15_0(9)=>address_16bits(9), bits15_0(8)=>address_16bits(8),
bits15_0(7)=>address_16bits(7), bits15_0(6)=>address_16bits(6),
bits15_0(5)=>address_16bits(5), bits15_0(4)=>address_16bits(4),
bits15_0(3)=>address_16bits(3), bits15_0(2)=>address_16bits(2),
bits15_0(1)=>address_16bits(1), bits15_0(0)=>address_16bits(0),
bits25_0(25)=>address_26bits(25), bits25_0(24)=>address_26bits(24),
bits25_0(23)=>address_26bits(23), bits25_0(22)=>address_26bits(22),
bits25_0(21)=>address_26bits(21), bits25_0(20)=>address_26bits(20),
bits25_0(19)=>address_26bits(19), bits25_0(18)=>address_26bits(18),
bits25_0(17)=>address_26bits(17), bits25_0(16)=>address_26bits(16),
bits25_0(15)=>address_26bits(15), bits25_0(14)=>address_26bits(14),
bits25_0(13)=>address_26bits(13), bits25_0(12)=>address_26bits(12),
bits25_0(11)=>address_26bits(11), bits25_0(10)=>address_26bits(10),
bits25_0(9)=>address_26bits(9), bits25_0(8)=>address_26bits(8),
bits25_0(7)=>address_26bits(7), bits25_0(6)=>address_26bits(6),
bits25_0(5)=>address_26bits(5), bits25_0(4)=>address_26bits(4),
bits25_0(3)=>address_26bits(3), bits25_0(2)=>address_26bits(2),
bits25_0(1)=>address_26bits(1), bits25_0(0)=>address_26bits(0));

XLXI_5 : mux32b_2to1
PORT MAP (din0(31)=>Reg_Write_Din_Preload(31),
din0(30)=>Reg_Write_Din_Preload(30), din0(29)=>Reg_Write_Din_Preload(29),
din0(28)=>Reg_Write_Din_Preload(28), din0(27)=>Reg_Write_Din_Preload(27),
din0(26)=>Reg_Write_Din_Preload(26), din0(25)=>Reg_Write_Din_Preload(25),
din0(24)=>Reg_Write_Din_Preload(24), din0(23)=>Reg_Write_Din_Preload(23),
din0(22)=>Reg_Write_Din_Preload(22), din0(21)=>Reg_Write_Din_Preload(21),
din0(20)=>Reg_Write_Din_Preload(20), din0(19)=>Reg_Write_Din_Preload(19),
din0(18)=>Reg_Write_Din_Preload(18), din0(17)=>Reg_Write_Din_Preload(17),
din0(16)=>Reg_Write_Din_Preload(16), din0(15)=>Reg_Write_Din_Preload(15),
din0(14)=>Reg_Write_Din_Preload(14), din0(13)=>Reg_Write_Din_Preload(13),
din0(12)=>Reg_Write_Din_Preload(12), din0(11)=>Reg_Write_Din_Preload(11),
din0(10)=>Reg_Write_Din_Preload(10), din0(9)=>Reg_Write_Din_Preload(9),
din0(8)=>Reg_Write_Din_Preload(8), din0(7)=>Reg_Write_Din_Preload(7),
din0(6)=>Reg_Write_Din_Preload(6), din0(5)=>Reg_Write_Din_Preload(5),
din0(4)=>Reg_Write_Din_Preload(4), din0(3)=>Reg_Write_Din_Preload(3),
din0(2)=>Reg_Write_Din_Preload(2), din0(1)=>Reg_Write_Din_Preload(1),
din0(0)=>Reg_Write_Din_Preload(0), din1(31)=>ALU_to_RF_DUMMY(31),
din1(30)=>ALU_to_RF_DUMMY(30), din1(29)=>ALU_to_RF_DUMMY(29),
din1(28)=>ALU_to_RF_DUMMY(28), din1(27)=>ALU_to_RF_DUMMY(27),
din1(26)=>ALU_to_RF_DUMMY(26), din1(25)=>ALU_to_RF_DUMMY(25),
din1(24)=>ALU_to_RF_DUMMY(24), din1(23)=>ALU_to_RF_DUMMY(23),
din1(22)=>ALU_to_RF_DUMMY(22), din1(21)=>ALU_to_RF_DUMMY(21),
din1(20)=>ALU_to_RF_DUMMY(20), din1(19)=>ALU_to_RF_DUMMY(19),
din1(18)=>ALU_to_RF_DUMMY(18), din1(17)=>ALU_to_RF_DUMMY(17),
din1(16)=>ALU_to_RF_DUMMY(16), din1(15)=>ALU_to_RF_DUMMY(15),
din1(14)=>ALU_to_RF_DUMMY(14), din1(13)=>ALU_to_RF_DUMMY(13),
din1(12)=>ALU_to_RF_DUMMY(12), din1(11)=>ALU_to_RF_DUMMY(11),
din1(10)=>ALU_to_RF_DUMMY(10), din1(9)=>ALU_to_RF_DUMMY(9),
din1(8)=>ALU_to_RF_DUMMY(8), din1(7)=>ALU_to_RF_DUMMY(7),
din1(6)=>ALU_to_RF_DUMMY(6), din1(5)=>ALU_to_RF_DUMMY(5),
din1(4)=>ALU_to_RF_DUMMY(4), din1(3)=>ALU_to_RF_DUMMY(3),
din1(2)=>ALU_to_RF_DUMMY(2), din1(1)=>ALU_to_RF_DUMMY(1),
din1(0)=>ALU_to_RF_DUMMY(0), dout(31)=>WriteData_DUMMY(31),
dout(30)=>WriteData_DUMMY(30), dout(29)=>WriteData_DUMMY(29),
dout(28)=>WriteData_DUMMY(28), dout(27)=>WriteData_DUMMY(27),
dout(26)=>WriteData_DUMMY(26), dout(25)=>WriteData_DUMMY(25),
dout(24)=>WriteData_DUMMY(24), dout(23)=>WriteData_DUMMY(23),
dout(22)=>WriteData_DUMMY(22), dout(21)=>WriteData_DUMMY(21),
dout(20)=>WriteData_DUMMY(20), dout(19)=>WriteData_DUMMY(19),
dout(18)=>WriteData_DUMMY(18), dout(17)=>WriteData_DUMMY(17),
dout(16)=>WriteData_DUMMY(16), dout(15)=>WriteData_DUMMY(15),
dout(14)=>WriteData_DUMMY(14), dout(13)=>WriteData_DUMMY(13),
dout(12)=>WriteData_DUMMY(12), dout(11)=>WriteData_DUMMY(11),
dout(10)=>WriteData_DUMMY(10), dout(9)=>WriteData_DUMMY(9),
dout(8)=>WriteData_DUMMY(8), dout(7)=>WriteData_DUMMY(7),
dout(6)=>WriteData_DUMMY(6), dout(5)=>WriteData_DUMMY(5),
dout(4)=>WriteData_DUMMY(4), dout(3)=>WriteData_DUMMY(3),

```

```

dout(2)=>WriteData_DUMMY(2), dout(1)=>WriteData_DUMMY(1),
dout(0)=>WriteData_DUMMY(0), sel=>Mux_Sel_RF_Din_Preload);

XLXI_3 : mux5b_2to1
PORT MAP (sel=>Mux_Sel_RF_Num_Preload, din0(4)=>Reg_Write_Num_Preload(4),
din0(3)=>Reg_Write_Num_Preload(3), din0(2)=>Reg_Write_Num_Preload(2),
din0(1)=>Reg_Write_Num_Preload(1), din0(0)=>Reg_Write_Num_Preload(0),
din1(4)=>rd_DUMMY(4), din1(3)=>rd_DUMMY(3), din1(2)=>rd_DUMMY(2),
din1(1)=>rd_DUMMY(1), din1(0)=>rd_DUMMY(0), dout(4)=>WriteReg_DUMMY(4),
dout(3)=>WriteReg_DUMMY(3), dout(2)=>WriteReg_DUMMY(2),
dout(1)=>WriteReg_DUMMY(1), dout(0)=>WriteReg_DUMMY(0));

XLXI_6 : rf_synch
PORT MAP (clk=>clk, enable=>RF_En_Read_Write, read_reg1(4)=>rs_DUMMY(4),
read_reg1(3)=>rs_DUMMY(3), read_reg1(2)=>rs_DUMMY(2),
read_reg1(1)=>rs_DUMMY(1), read_reg1(0)=>rs_DUMMY(0),
read_reg2(4)=>rt_DUMMY(4), read_reg2(3)=>rt_DUMMY(3),
read_reg2(2)=>rt_DUMMY(2), read_reg2(1)=>rt_DUMMY(1),
read_reg2(0)=>rt_DUMMY(0), write_data(31)=>WriteData_DUMMY(31),
write_data(30)=>WriteData_DUMMY(30), write_data(29)=>WriteData_DUMMY(29),
write_data(28)=>WriteData_DUMMY(28), write_data(27)=>WriteData_DUMMY(27),
write_data(26)=>WriteData_DUMMY(26), write_data(25)=>WriteData_DUMMY(25),
write_data(24)=>WriteData_DUMMY(24), write_data(23)=>WriteData_DUMMY(23),
write_data(22)=>WriteData_DUMMY(22), write_data(21)=>WriteData_DUMMY(21),
write_data(20)=>WriteData_DUMMY(20), write_data(19)=>WriteData_DUMMY(19),
write_data(18)=>WriteData_DUMMY(18), write_data(17)=>WriteData_DUMMY(17),
write_data(16)=>WriteData_DUMMY(16), write_data(15)=>WriteData_DUMMY(15),
write_data(14)=>WriteData_DUMMY(14), write_data(13)=>WriteData_DUMMY(13),
write_data(12)=>WriteData_DUMMY(12), write_data(11)=>WriteData_DUMMY(11),
write_data(10)=>WriteData_DUMMY(10), write_data(9)=>WriteData_DUMMY(9),
write_data(8)=>WriteData_DUMMY(8), write_data(7)=>WriteData_DUMMY(7),
write_data(6)=>WriteData_DUMMY(6), write_data(5)=>WriteData_DUMMY(5),
write_data(4)=>WriteData_DUMMY(4), write_data(3)=>WriteData_DUMMY(3),
write_data(2)=>WriteData_DUMMY(2), write_data(1)=>WriteData_DUMMY(1),
write_data(0)=>WriteData_DUMMY(0), write_reg(4)=>WriteReg_DUMMY(4),
write_reg(3)=>WriteReg_DUMMY(3), write_reg(2)=>WriteReg_DUMMY(2),
write_reg(1)=>WriteReg_DUMMY(1), write_reg(0)=>WriteReg_DUMMY(0),
read_data_1(31)=>A_in_DUMMY(31), read_data_1(30)=>A_in_DUMMY(30),
read_data_1(29)=>A_in_DUMMY(29), read_data_1(28)=>A_in_DUMMY(28),
read_data_1(27)=>A_in_DUMMY(27), read_data_1(26)=>A_in_DUMMY(26),
read_data_1(25)=>A_in_DUMMY(25), read_data_1(24)=>A_in_DUMMY(24),
read_data_1(23)=>A_in_DUMMY(23), read_data_1(22)=>A_in_DUMMY(22),
read_data_1(21)=>A_in_DUMMY(21), read_data_1(20)=>A_in_DUMMY(20),
read_data_1(19)=>A_in_DUMMY(19), read_data_1(18)=>A_in_DUMMY(18),
read_data_1(17)=>A_in_DUMMY(17), read_data_1(16)=>A_in_DUMMY(16),
read_data_1(15)=>A_in_DUMMY(15), read_data_1(14)=>A_in_DUMMY(14),
read_data_1(13)=>A_in_DUMMY(13), read_data_1(12)=>A_in_DUMMY(12),
read_data_1(11)=>A_in_DUMMY(11), read_data_1(10)=>A_in_DUMMY(10),
read_data_1(9)=>A_in_DUMMY(9), read_data_1(8)=>A_in_DUMMY(8),
read_data_1(7)=>A_in_DUMMY(7), read_data_1(6)=>A_in_DUMMY(6),
read_data_1(5)=>A_in_DUMMY(5), read_data_1(4)=>A_in_DUMMY(4),
read_data_1(3)=>A_in_DUMMY(3), read_data_1(2)=>A_in_DUMMY(2),
read_data_1(1)=>A_in_DUMMY(1), read_data_1(0)=>A_in_DUMMY(0),
read_data_2(31)=>B_in_DUMMY(31), read_data_2(30)=>B_in_DUMMY(30),
read_data_2(29)=>B_in_DUMMY(29), read_data_2(28)=>B_in_DUMMY(28),
read_data_2(27)=>B_in_DUMMY(27), read_data_2(26)=>B_in_DUMMY(26),
read_data_2(25)=>B_in_DUMMY(25), read_data_2(24)=>B_in_DUMMY(24),
read_data_2(23)=>B_in_DUMMY(23), read_data_2(22)=>B_in_DUMMY(22),
read_data_2(21)=>B_in_DUMMY(21), read_data_2(20)=>B_in_DUMMY(20),
read_data_2(19)=>B_in_DUMMY(19), read_data_2(18)=>B_in_DUMMY(18),
read_data_2(17)=>B_in_DUMMY(17), read_data_2(16)=>B_in_DUMMY(16),
read_data_2(15)=>B_in_DUMMY(15), read_data_2(14)=>B_in_DUMMY(14),
read_data_2(13)=>B_in_DUMMY(13), read_data_2(12)=>B_in_DUMMY(12),
read_data_2(11)=>B_in_DUMMY(11), read_data_2(10)=>B_in_DUMMY(10),
read_data_2(9)=>B_in_DUMMY(9), read_data_2(8)=>B_in_DUMMY(8),
read_data_2(7)=>B_in_DUMMY(7), read_data_2(6)=>B_in_DUMMY(6),
read_data_2(5)=>B_in_DUMMY(5), read_data_2(4)=>B_in_DUMMY(4),
read_data_2(3)=>B_in_DUMMY(3), read_data_2(2)=>B_in_DUMMY(2),
read_data_2(1)=>B_in_DUMMY(1), read_data_2(0)=>B_in_DUMMY(0));

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the datapath section for R-type instructions, was generated. Figure B.10 shows the resulting top level RTL symbol while figure B.11 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are covered in detail in Appendix A.

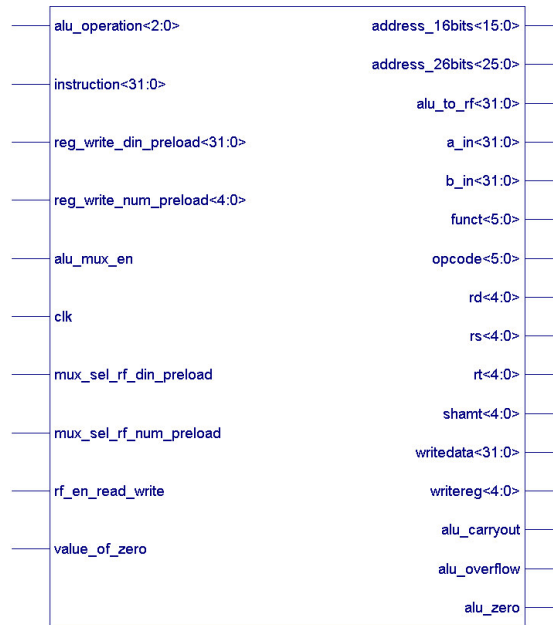


Figure B.10 Resulting top level RTL symbol for the datapath section for R-type instructions.

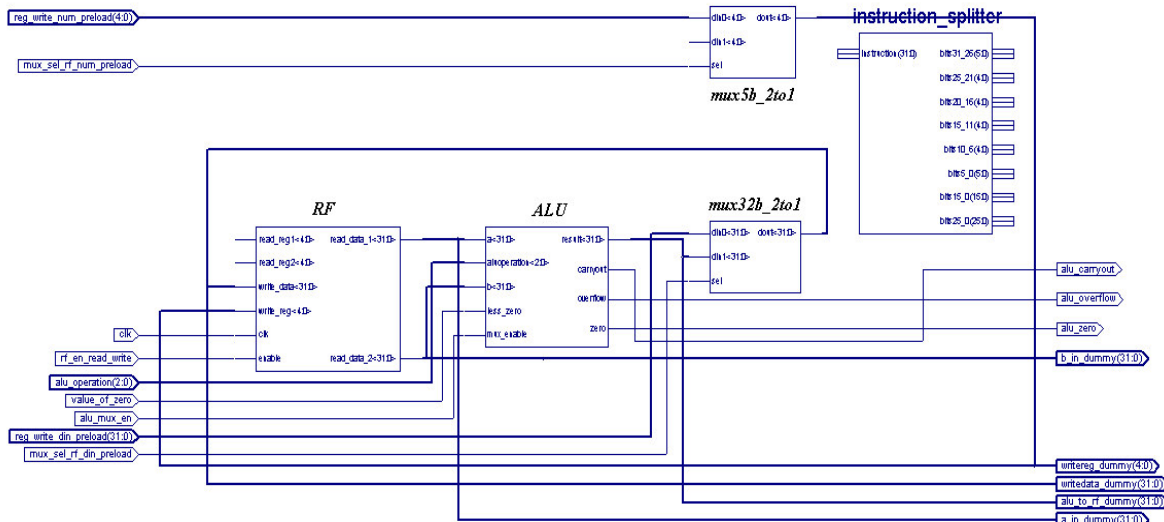


Figure B.11 Resulting top level RTL schematic for the datapath section for R-type instructions.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for the datapath section for R-type instructions, 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                                : 288

Macro Statistics:
# RAM                                  : 2
#      32x32-bit dual-port block RAM: 2
# Tristates                           : 4
#      32-bit tristate buffer         : 2
#      5-bit tristate buffer          : 2

Cell Usage:
# BELS                                : 719
#      and2                           : 96
#      and2b1                         : 32
#      and3                           : 64
#      and3b1                        : 64
#      GND                            : 3
#      inv                            : 33
#      LUT1                           : 34
#      LUT1_L                         : 32
#      LUT2_L                         : 1
#      LUT3                           : 50
#      LUT3_L                        : 11
#      LUT4                           : 21
#      LUT4_D                        : 10
#      LUT4_L                        : 30
#      muxcy                          : 2
#      muxcy_1                       : 6
#      muxf5                          : 32
#      or2                            : 161
#      VCC                            : 3
#      xor2                           : 2
#      xor3                           : 32
# RAMS                                : 2
#      RAMB16_S36_S36                : 2
# Tri-States                         : 74
#      BUFT                           : 74
# Clock Buffers                      : 1
#      BUFGP                         : 1
# IO Buffers                         : 287
#      IBUF                          : 77
#      OBUF                          : 210
# Logical                            : 8
#      nor4                           : 8
# Others                             : 8
#      fmap                           : 8

Device utilization summary:

Number of Slices:                      93 out of 46592    0%
Number of 4 input LUTs:                189 out of 93184    0%
Number of bonded IOBs:                287 out of 1108    25%
Number of TBUFs:                      74 out of 23296    0%
Number of BRAMs:                      2 out of 168        1%
Number of GCLKs:                      1 out of 16          6%

Timing Summary:

Minimum period: 22.856ns (Maximum Frequency: 43.752MHz)
Minimum input arrival time before clock: 21.533ns
Maximum output required time after clock: 30.323ns
Maximum combinational path delay: 29.000ns

```

➤ *Place-and-Route onto the FPGA*

In figure B.12, FPGA Editor shows the synthesized datapath section for R-type instructions after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the upper section of the FPGA chip.

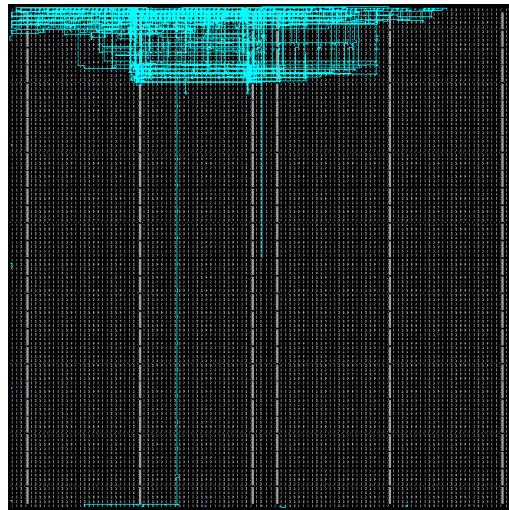


Figure B.12 *FPGA Editor showing the synthesized datapath section for R-type instructions after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figures B.13 to B.23 show the waveform results of simulating in ModelSim the VHDL behavioural model for the synthesized datapath section for R-type instructions.

Before elaborating and analysing these simulation waveforms, a few key points must first be highlighted which also pertain to the schematic diagrams of figures B.13 to B.23. These key points are:

- ❑ Although the whole 32-bit instruction input is supplied, yet the 3-bit *ALU_Operation(2:0)* control signal must yet be supplied independently during these simulations. This is acceptable only for now as this datapath section is being tested in isolation from the final complete MIPS processor in which the Control Unit supplies the *ALU_Operation(2:0)* control signal by extracting it from both the *op* and *funct* fields within the supplied instruction. This is discussed in detail in Appendix C and Chapter 6.
- ❑ In figures B.13 to B.23, the R-format instructions simulated are ADD, SUB, AND, OR, and SLT respectively.
- ❑ The simulation for all of these instructions follows the same testing sequence of operations:
 - First, the RF is preloaded with the data values for *rs* and *rt* source register operands.
 - Then, the ALU calculates the result value *Result(31:0)*.
 - This is followed by the result value *Result(31:0)* being written into the destination register operand *rd* in the RF.
 - Finally, a read cycle for the RF is performed to read *rd* to make sure it has properly written the result value.

- In figures B.13 to B.23, some of the signals are in binary format while others are in decimal (for ease of debugging).

Following is the detailed elaboration and analysis for these simulation waveforms:

➤ **ALU Operation = ADD**

Figure B.13 shows the simulation waveforms for ALU Operation = ADD = $(010)_{\text{binary}} = (2)_{\text{decimal}}$. The waveform is divided into 4 parts to facilitate analysis and discussion:

□ **During part 1 of the waveform (clock cycles 1 and 2):**

- Preload the register file with the register operands:

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (16)_{\text{decimal}}$

- Test Instruction:

ADD	\$R7 ,	\$R5 ,	\$R6
-----	-----	-----	-----
rd	rs	rt	

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

000000	00101	00110	00111	00000	100000
-----	-----	-----	-----	-----	-----
$op=0$	$rs=\$R5$	$rt=\$R6$	$rd=\$R7$	$shamt$	$funct=32$

□ **During part 2 of the waveform (clock cycle 3):**

- The ALU calculates the result value:

$$\begin{aligned}
 result(31:0) \Rightarrow alu_to_rf(31:0) \Rightarrow writedata(31:0) &= A_in(31:0) + B_in(31:0) \\
 &= [\$R5] + [\$R6] \\
 &= (15)_{\text{decimal}} + (16)_{\text{decimal}} \\
 &= (31)_{\text{decimal}}
 \end{aligned}$$

- Timing Information:

- ♦ Delay between application of input signals and clock rising edge = 1 ns.
- ♦ On this clock rising edge, both source register operands represented by the intermediate signals a_in and b_in are read out from the RF and fed into the ALU inputs.
- ♦ Delay between application of input signals a_in and b_in to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.7 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

□ **During part 3 of the waveform (clock cycle 4):**

- The ALU result value is written back into the RF (on rising clock edge):

$rd = \$R7$, $[\$R7] = (31)_{\text{decimal}}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

❑ **During part 4 of the waveform (clock cycle 5):**

- The ALU result value is read out from the RF (on the rising clock edge):

$rd = \$R7$, $[\$R7] = (31)_{\text{decimal}}$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) $\$R7$:

```
111111 00111 00111 00111 11111 111111
-----
op      rs=$R7 rt=$R7 rd=$R7 shamt funct
```

- The ALU calculates the next result value:

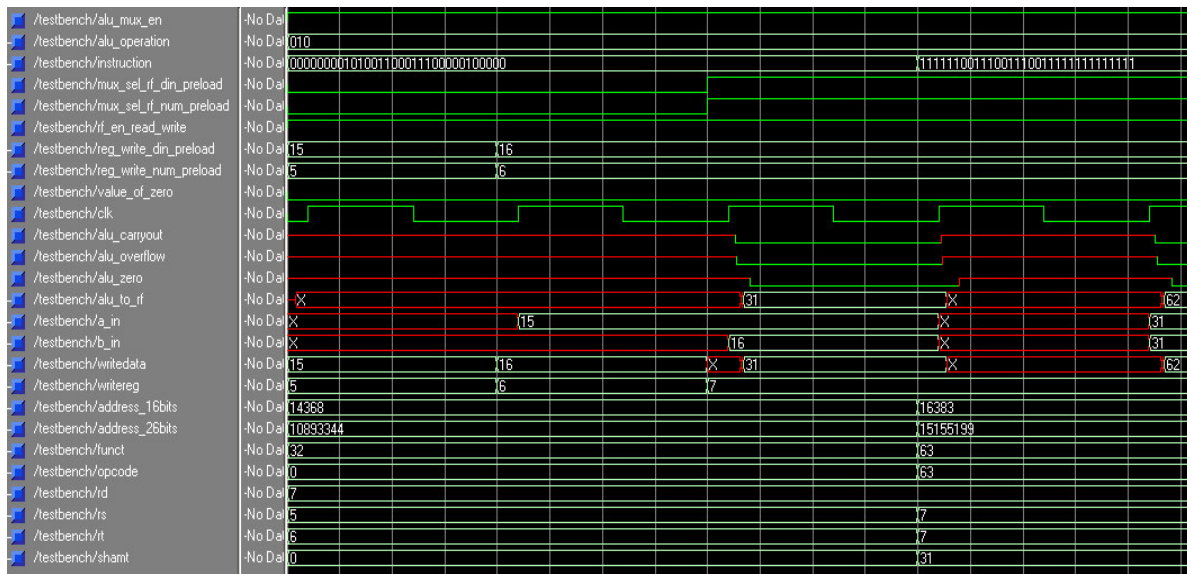
$$\begin{aligned} \text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) &= A_in(31:0) + B_in(31:0) \\ &= [\$R7] + [\$R7] \\ &= (31)_{\text{decimal}} + (31)_{\text{decimal}} \\ &= (62)_{\text{decimal}} \end{aligned}$$


Figure B.13 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = ADD.

➤ **ALU Operation = SUB**

Figure B.14 shows the simulation waveforms for ALU Operation = SUB = $(110)_{\text{binary}} = (6)_{\text{decimal}}$. The waveform is divided into 4 parts to facilitate analysis and discussion:

❑ **During part 1 of the waveform (clock cycles 1 and 2):**

- Preload the register file with the register operands:

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (16)_{\text{decimal}}$

- Test Instruction:

SUB	$\$R7$,	$\$R5$,	$\$R6$
	-----	-----	-----
	rd	rs	rt

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

000000	00101	00110	00111	00000	100010
-----	-----	-----	-----	-----	-----
$op=0$	$rs=\$R5$	$rt=\$R6$	$rd=\$R7$	$shamt$	$funct=34$

❑ **During part 2 of the waveform (clock cycle 3):**

- The ALU calculates the result value:

$$\begin{aligned}
 result(31:0) => alu_to_rf(31:0) => writedata(31:0) &= A_in(31:0) - B_in(31:0) \\
 &= [\$R5] - [\$R6] \\
 &= (15)_{\text{decimal}} - (16)_{\text{decimal}} \\
 &= (-1)_{\text{decimal}}
 \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals a_in and b_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals a_in and b_in to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.8 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

❑ **During part 3 of the waveform (clock cycle 4):**

- The ALU result value is written back into the RF (on rising clock edge):

$rd = \$R7$, $[\$R7] = (-1)_{\text{decimal}}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

❑ **During part 4 of the waveform (clock cycle 5):**

- The ALU result value is read out from the RF (on the rising clock edge):

$rd = \$R7$, $[\$R7] = (-1)_{\text{decimal}}$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) $\$R7$:

111111	00111	00111	00111	11111	111111
-----	-----	-----	-----	-----	-----
op	$rs=\$R7$	$rt=\$R7$	$rd=\$R7$	$shamt$	$funct$

- The ALU calculates the next result value:

$$\begin{aligned}
 \text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) &= A_in(31:0) - B_in(31:0) \\
 &= [\$R7] - [\$R7] \\
 &= (-1)_{\text{decimal}} - (-1)_{\text{decimal}} \\
 &= (0)_{\text{decimal}}
 \end{aligned}$$

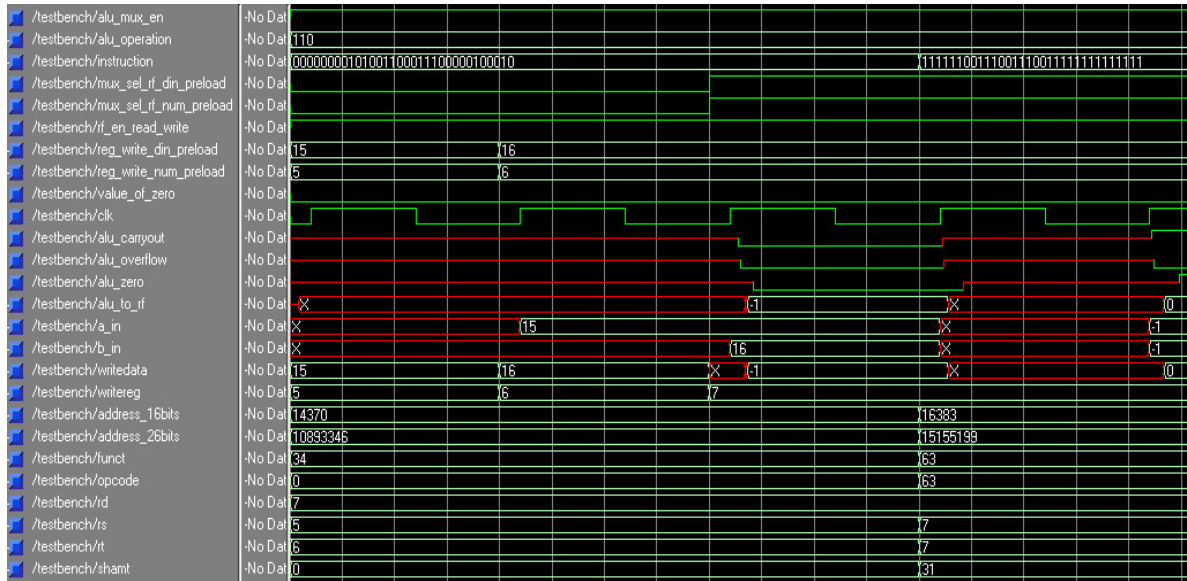


Figure B.14 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = SUB.

➤ ALU Operation = AND

Figures B.15 to B.17 show the simulation waveforms for ALU Operation = AND = (000)_{binary} = (0)_{decimal}. The waveform is divided into 4 parts across three figures to facilitate analysis and discussion:

❑ During part 1 of the waveform (clock cycles 1 and 2) – Figure B.15:

- Preload the register file with the register operands:

$$rs = \$R5, \quad [\$R5] = (00101101001001001111110110011001)_{\text{binary}}$$

$$rt = \$R6, \quad [\$R6] = (11110110011001001011010010010011)_{\text{binary}}$$

- Test Instruction:

$$\begin{array}{ccc}
 \text{AND} & \$R7, & \$R5, & \$R6 \\
 \text{-----} & \text{-----} & \text{-----} & \\
 rd & rs & rt &
 \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccccc}
 000000 & 00101 & 00110 & 00111 & 00000 & 100100 \\
 \text{-----} & \text{-----} & \text{-----} & \text{-----} & \text{-----} & \text{-----} \\
 op=0 & rs=\$R5 & rt=\$R6 & rd=\$R7 & shamt & funct=36
 \end{array}$$

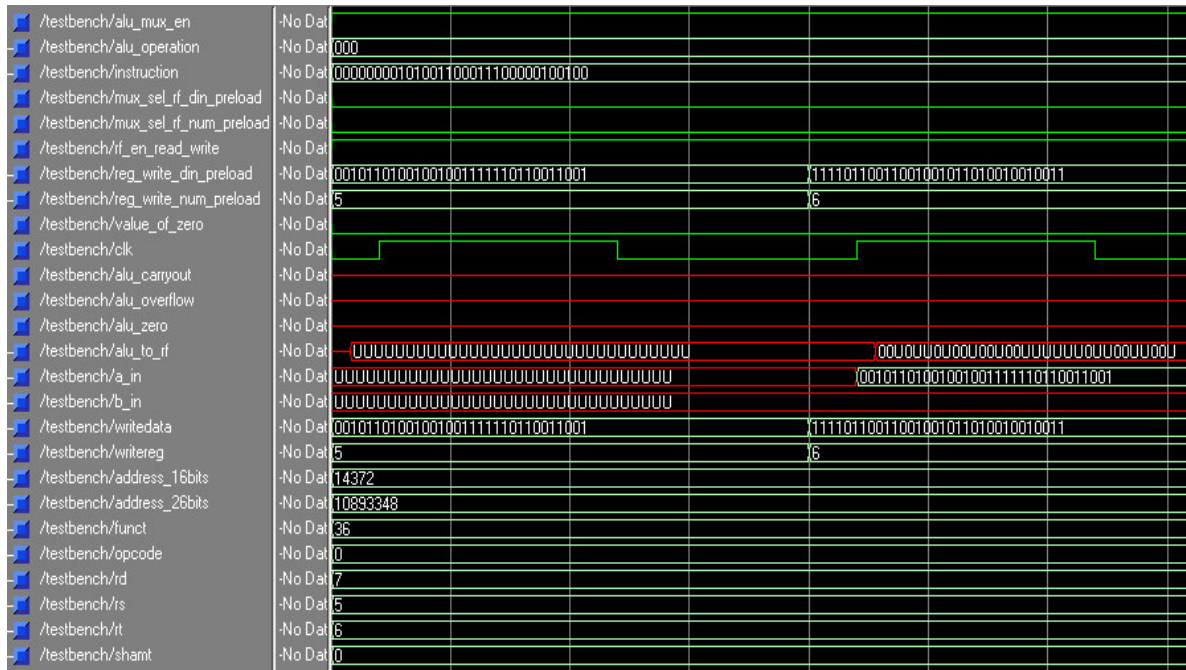


Figure B.15 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = AND. This is part 1 of the waveform during clock cycles 1 and 2.

□ During part 2 of the waveform (clock cycle 3) – Figure B.16:

- The ALU calculates the result value:

$$\begin{aligned} result(31:0) => alu_to_rf(31:0) => writedata(31:0) &= A_in(31:0) \text{ AND } B_in(31:0) \\ &= [\$R5] \quad \text{AND} \quad [\$R6] \end{aligned}$$

$$\begin{aligned} &= (00101101001001001111110110011001)_{\text{binary}} \\ &\quad \text{AND} \\ &\quad (11110110011001001011010010010011)_{\text{binary}} \\ &= (00100100001001001011010010010001)_{\text{binary}} \end{aligned}$$

- Timing Information:

- ♦ Delay between application of input signals and clock rising edge = 1 ns.
- ♦ On this clock rising edge, both source register operands represented by the intermediate signals *a_in* and *b_in* are read out from the RF and fed into the ALU inputs.
- ♦ Delay between application of input signals *a_in* and *b_in* to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.6 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ During part 3 of the waveform (clock cycle 4) – Figure B.16:

- The ALU result value is written back into the RF (on rising clock edge):

$rd = \$R7$, $[\$R7] = (00100100001001001011010010010001)_{\text{binary}}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

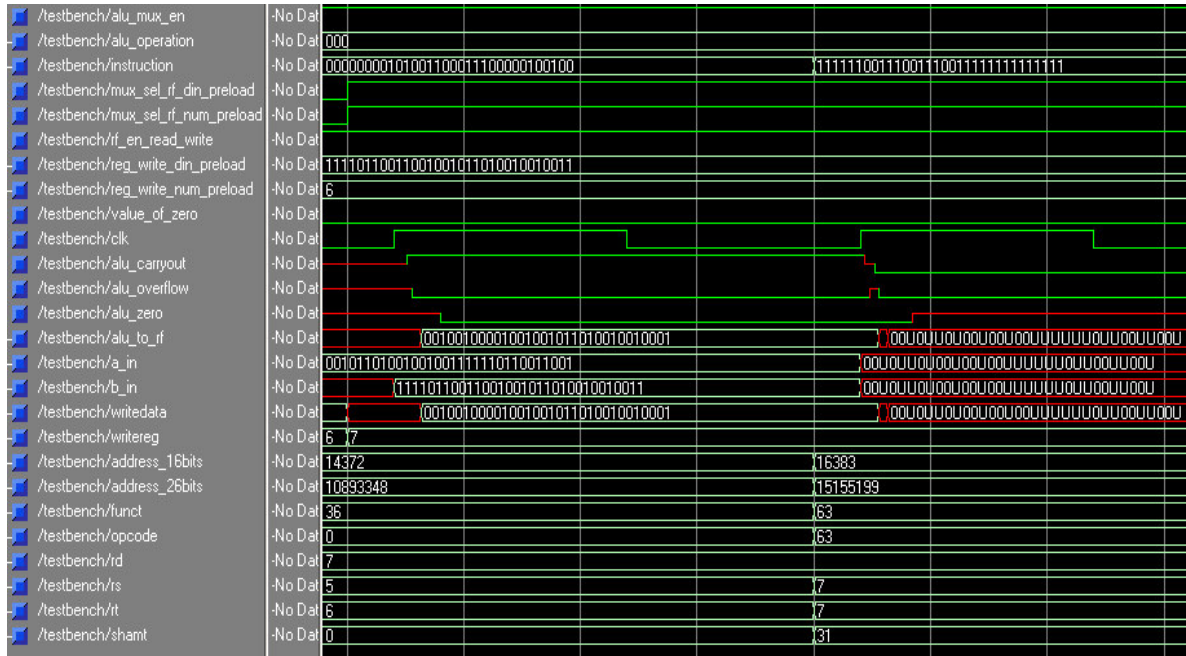


Figure B.16 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = AND. This is parts 2 and 3 of the waveform during clock cycles 3 and 4, respectively.

□ During part 4 of the waveform (clock cycle 5) – Figure B.17:

- The ALU result value is read out from the RF (on the rising clock edge):

$rd = \$R7$, $[\$R7] = (00100100001001001011010010010001)_{\text{binary}}$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) \$R7:

```

111111 00111 00111 00111 11111 111111
-----
op      rs=$R7 rt=$R7 rd=$R7 shamt funct

```

- The ALU calculates the next result value:

$$\begin{aligned}
 \text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) &= A_in(31:0) + B_in(31:0) \\
 &= [\$R7] + [\$R7] \\
 &= (00100100001001001011010010010001)_{\text{binary}} \\
 &\quad \text{AND} \\
 &\quad (00100100001001001011010010010001)_{\text{binary}} \\
 &= (00100100001001001011010010010001)_{\text{binary}}
 \end{aligned}$$

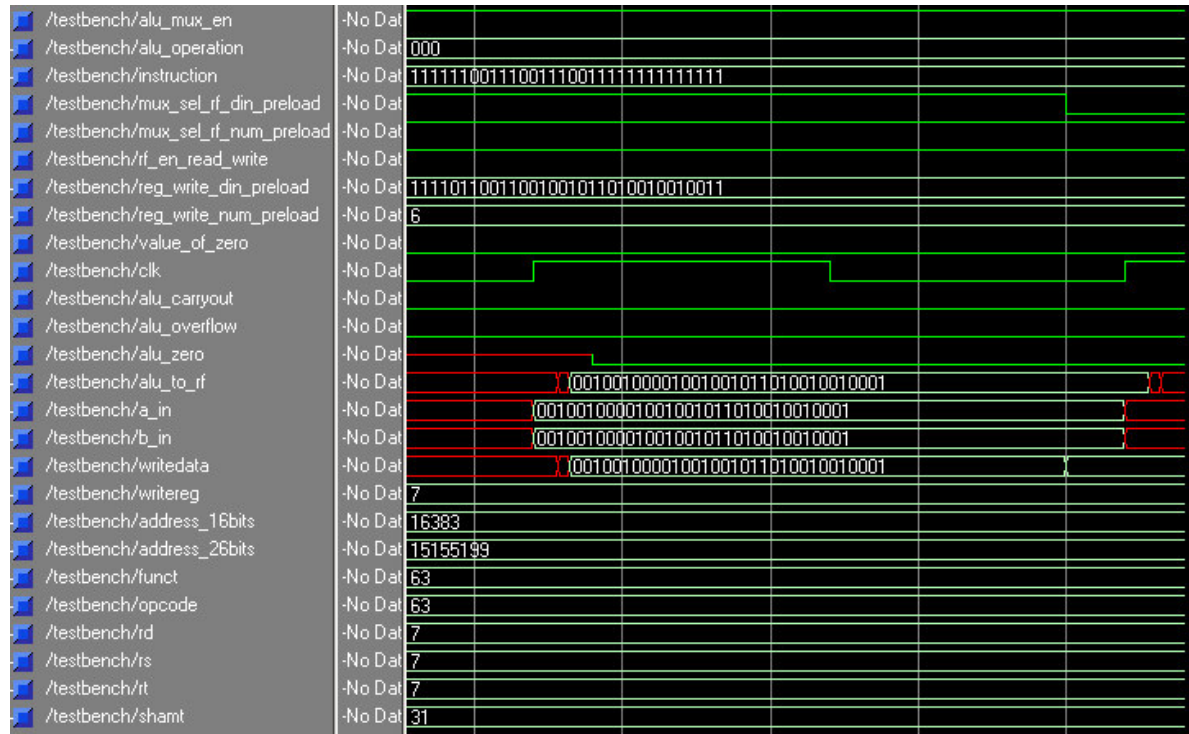


Figure B.17 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = AND. This is part 4 of the waveform during clock cycle 5.

➤ ALU Operation = OR

Figures B.18 to B.20 show the simulation waveforms for ALU Operation = OR = $(001)_{\text{binary}} = (1)_{\text{decimal}}$. The waveform is divided into 4 parts across three figures to facilitate analysis and discussion:

❑ During part 1 of the waveform (clock cycles 1 and 2) – Figure B.18:

- Preload the register file with the register operands:

$rs = \$R5$, $[\$R5] = (00101101001001001111110110011001)_{\text{binary}}$

$rt = \$R6$, $[\$R6] = (11110110011001001011010010010011)_{\text{binary}}$

- Test Instruction:

OR	$\$R7$,	$\$R5$,	$\$R6$
	-----	-----	-----
	rd	rs	rt

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

000000	00101	00110	00111	00000	100101
-----	-----	-----	-----	-----	-----
$op=0$	$rs=\$R5$	$rt=\$R6$	$rd=\$R7$	$shamt$	$funct=37$

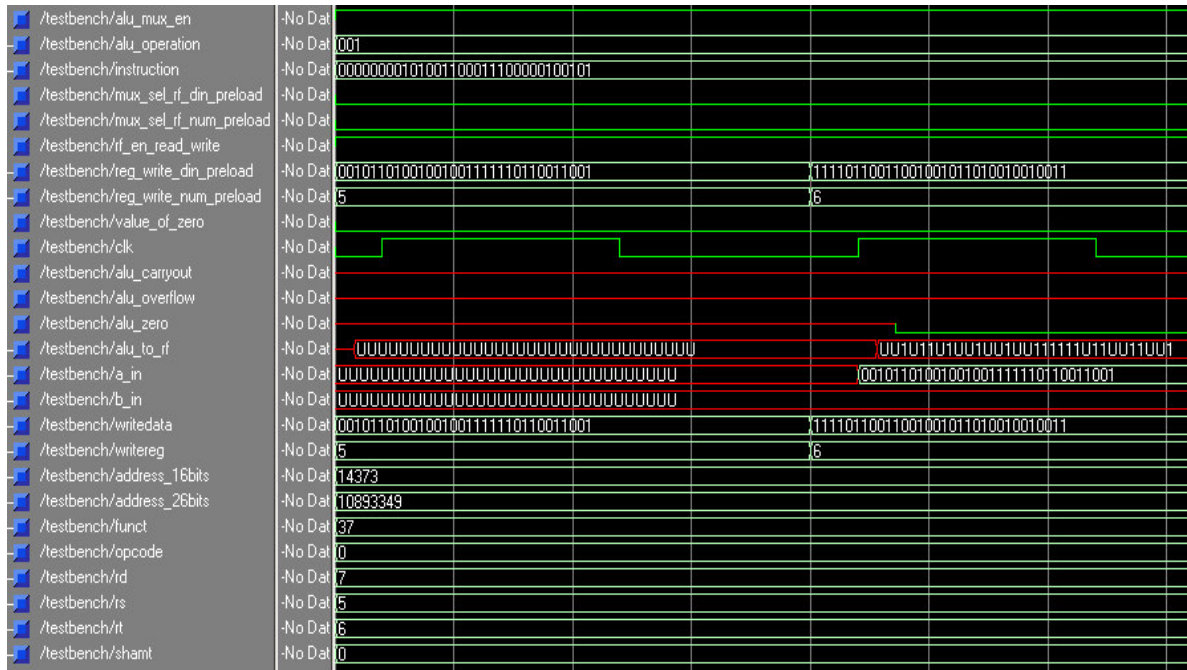


Figure B.18 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = OR. This is part 1 of the waveform during clock cycles 1 and 2.

❑ **During part 2 of the waveform (clock cycle 3) – Figure B.19:**

- The ALU calculates the result value:

$$\begin{aligned} result(31:0) \Rightarrow alu_to_rf(31:0) \Rightarrow writedata(31:0) &= A_in(31:0) \text{ OR } B_in(31:0) \\ &= [\$R5] \quad \text{OR} \quad [\$R6] \end{aligned}$$

$$\begin{aligned} &= (00101101001001001111110110011001)_{\text{binary}} \\ &\quad \text{OR} \\ &= (11110110011001001011010010010011)_{\text{binary}} \\ &= (11111111011001001111110110011011)_{\text{binary}} \end{aligned}$$

- Timing Information:

- ♦ Delay between application of input signals and clock rising edge = 1 ns.
- ♦ On this clock rising edge, both source register operands represented by the intermediate signals a_in and b_in are read out from the RF and fed into the ALU inputs.
- ♦ Delay between application of input signals a_in and b_in to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.6 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

❑ **During part 3 of the waveform (clock cycle 4) – Figure B.19:**

- The ALU result value is written back into the RF (on rising clock edge):

$rd = \$R7$, $[\$R7] = (11111111011001001111110110011011)_{\text{binary}}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

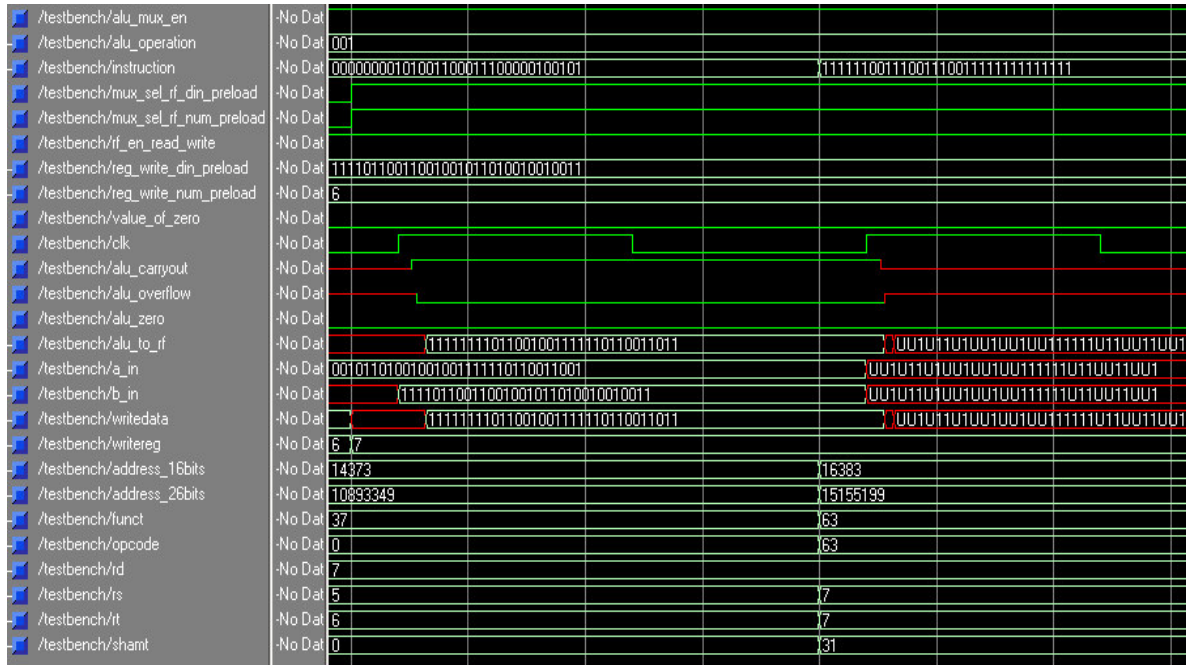


Figure B.19 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = OR. This is parts 2 and 3 of the waveform during clock cycles 3 and 4, respectively.

□ During part 4 of the waveform (clock cycle 5) – Figure B.20:

- The ALU result value is read out from the RF (on the rising clock edge):

$rd = \$R7$, $[\$R7] = (11111111011001001111110110011011)_{\text{binary}}$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) \$R7:

```

111111 00111 00111 00111 11111 111111
-----
op      rs=$R7 rt=$R7 rd=$R7 shamt funct

```

- The ALU calculates the next result value:

$$\begin{aligned}
 \text{result}(31:0) &\Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) = A_in(31:0) + B_in(31:0) \\
 &= [\$R7] + [\$R7] \\
 &= (11111111011001001111110110011011)_{\text{binary}} \\
 &\quad \text{OR} \\
 &\quad (11111111011001001111110110011011)_{\text{binary}} \\
 &= (11111111011001001111110110011011)_{\text{binary}}
 \end{aligned}$$

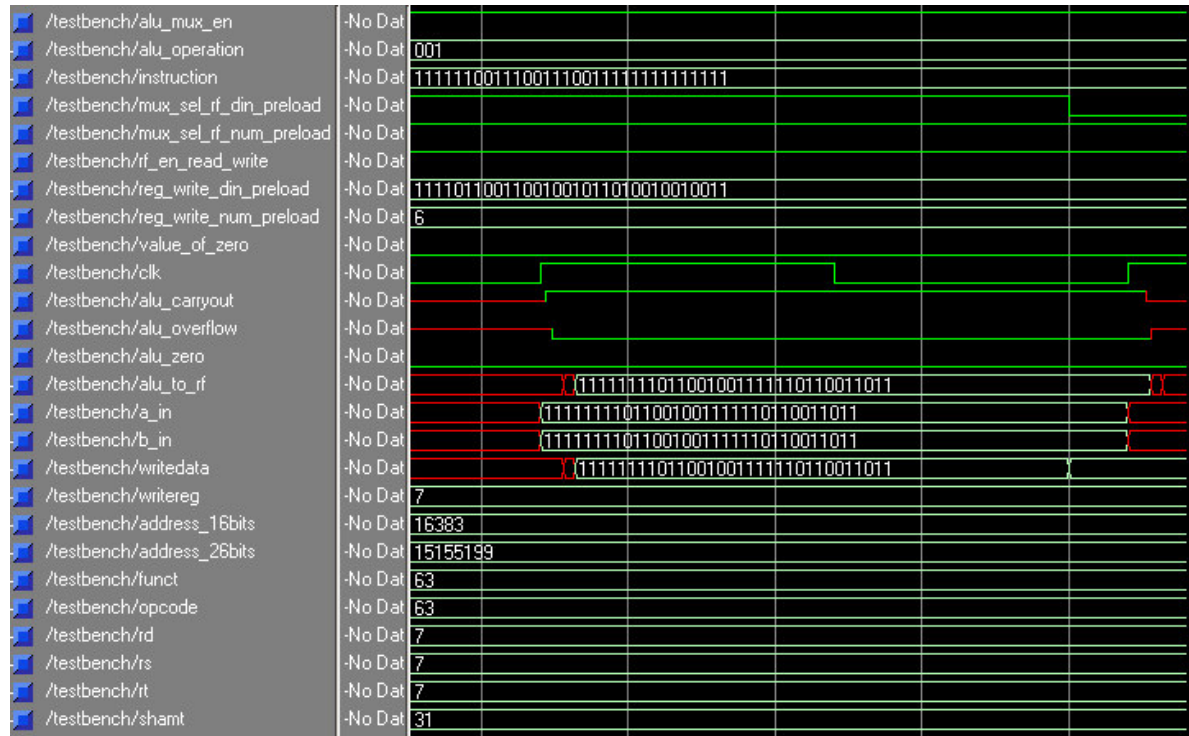


Figure B.20 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = OR. This is part 4 of the waveform during clock cycle 5.

➤ ALU Operation = SLT

Figures B.21 to B.23 show the simulation waveforms for ALU Operation = SLT = $(111)_{\text{binary}} = (7)_{\text{decimal}}$ for the three conditions $[rs] < [rt]$, $[rs] = [rt]$, and $[rs] > [rt]$, respectively. The waveform for each condition is divided into 4 parts to facilitate analysis and discussion:

❖ Condition 1 (Figure B.21): When $[rs] < [rt]$

□ During part 1 of the waveform (clock cycles 1 and 2):

- Preload the register file with the register operands:

$$rs = \$R5, \quad [\$R5] = (15)_{\text{decimal}}$$

$$rt = \$R6, \quad [\$R6] = (16)_{\text{decimal}}$$

- Test Instruction:

$$\begin{array}{ccc} \text{SLT} & \$R7, & \$R5, \quad \$R6 \\ \hline & rd & rs \quad rt \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccccc} 000000 & 00101 & 00110 & 00111 & 00000 & 101010 \\ \hline op=0 & rs=\$R5 & rt=\$R6 & rd=\$R7 & shamt & funct=42 \end{array}$$

□ **During part 2 of the waveform (clock cycle 3):**

- The ALU calculates the result value:

Since

$$(15)_{\text{decimal}} < (16)_{\text{decimal}}$$

i.e.

$$[\$R5] < [\$R6]$$

Then

$$\text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) = (1)_{\text{decimal}}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals *a_in* and *b_in* are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals *a_in* and *b_in* to the ALU and the ALU result ($\text{result}(31:0) = \text{alu_to_rf}(31:0) = \text{writedata}(31:0)$) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During part 3 of the waveform (clock cycle 4):**

- The ALU result value is written back into the RF (on rising clock edge):

$$rd = \$R7, \quad [\$R7] = (1)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During part 4 of the waveform (clock cycle 5):**

- The ALU result value is read out from the RF (on the rising clock edge):

$$rd = \$R7, \quad [\$R7] = (1)_{\text{decimal}}$$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) \$R7:

111111	00111	00111	00111	11111	111111
-----	-----	-----	-----	-----	-----
<i>op</i>	<i>rs</i> =\$R7	<i>rt</i> =\$R7	<i>rd</i> =\$R7	<i>shamt</i>	<i>funct</i>

- The ALU calculates the next result value:

Since

$$(1)_{\text{decimal}} = (1)_{\text{decimal}}$$

i.e.

$$[\$R7] = [\$R7]$$

Then

$$\text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) = (0)_{\text{decimal}}$$

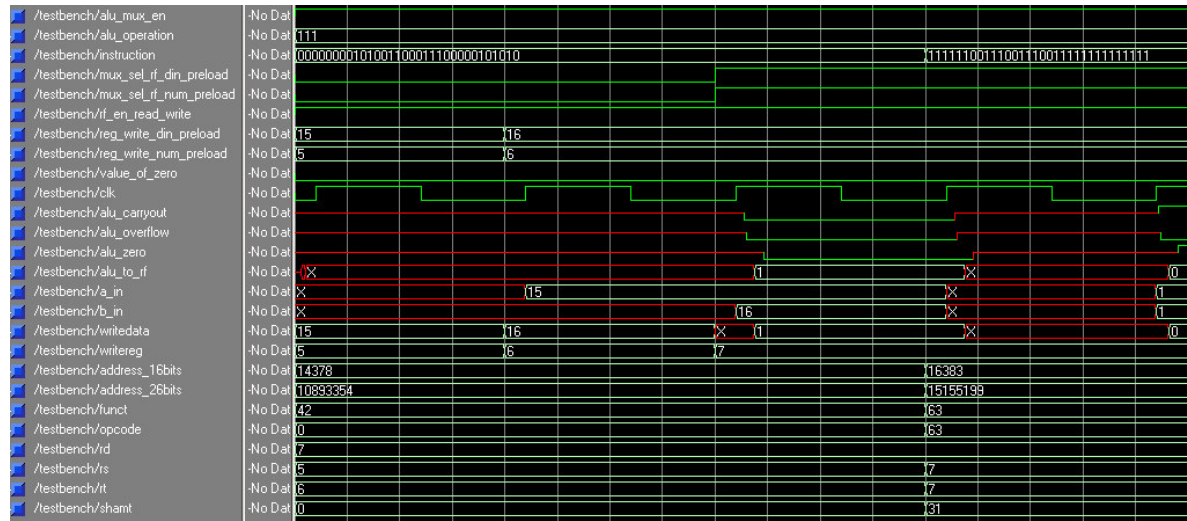


Figure B.21 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = SLT for $[rs] < [rt]$.

❖ **Condition 2 (Figure B.22): When $[rs] = [rt]$**

□ **During part 1 of the waveform (clock cycles 1 and 2):**

- Preload the register file with the register operands:

$$rs = \$R5, \quad [\$R5] = (15)_{\text{decimal}}$$

$$rt = \$R6, \quad [\$R6] = (15)_{\text{decimal}}$$

- Test Instruction:

$$\begin{array}{ccc} \text{SLT} & \$R7, & \$R5, \quad \$R6 \\ \text{-----} & \text{-----} & \text{-----} \\ & rd & rs \quad rt \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccccc} 000000 & 00101 & 00110 & 00111 & 00000 & 101010 \\ \text{-----} & \text{-----} & \text{-----} & \text{-----} & \text{-----} & \text{-----} \\ op=0 & rs=\$R5 & rt=\$R6 & rd=\$R7 & shamt & funct=42 \end{array}$$

□ **During part 2 of the waveform (clock cycle 3):**

- The ALU calculates the result value:

Since

$$(15)_{\text{decimal}} = (16)_{\text{decimal}} \\ \text{i.e.}$$

$$[\$R5] = [\$R6]$$

Then

$$\text{result}(31:0) \Rightarrow \text{alu_to_rf}(31:0) \Rightarrow \text{writedata}(31:0) = (0)_{\text{decimal}}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals a_in and b_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals a_in and b_in to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During part 3 of the waveform (clock cycle 4):**

- The ALU result value is written back into the RF (on rising clock edge):

$$rd = \$R7, \quad [\$R7] = (0)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During part 4 of the waveform (clock cycle 5):**

- The ALU result value is read out from the RF (on the rising clock edge):

$$rd = \$R7, \quad [\$R7] = (0)_{\text{decimal}}$$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) \$R7:

111111	001111	001111	001111	111111	111111
-----	-----	-----	-----	-----	-----
<i>op</i>	<i>rs=\$R7</i>	<i>rt=\$R7</i>	<i>rd=\$R7</i>	<i>shamt</i>	<i>funct</i>

- The ALU calculates the next result value:

Since

$$(0)_{\text{decimal}} = (0)_{\text{decimal}}$$

i.e.

$$[\$R7] = [\$R7]$$

Then

$$result(31:0) \Rightarrow alu_to_rf(31:0) \Rightarrow writedata(31:0) = (0)_{\text{decimal}}$$

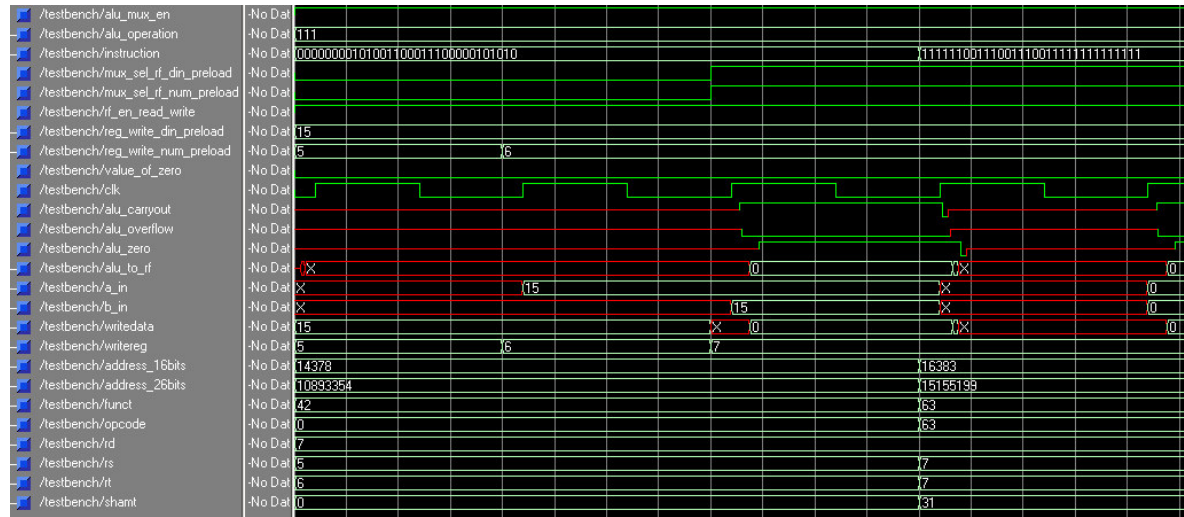


Figure B.22 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = SLT for $[rs] = [rt]$.

❖ **Condition 3 (Figure B.23): When $[rs] > [rt]$**

□ **During part 1 of the waveform (clock cycles 1 and 2):**

- Preload the register file with the register operands:

$$rs = \$R5, \quad [\$R5] = (16)_{\text{decimal}}$$

$$rt = \$R6, \quad [\$R6] = (15)_{\text{decimal}}$$

- Test Instruction:

$$\begin{array}{ccc} \text{SLT} & \$R7, & \$R5, \quad \$R6 \\ \hline & rd & rs \quad rt \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccccc} 000000 & 00101 & 00110 & 00111 & 00000 & 101010 \\ \hline op=0 & rs=\$R5 & rt=\$R6 & rd=\$R7 & shamt & funct=42 \end{array}$$

□ **During part 2 of the waveform (clock cycle 3):**

- The ALU calculates the result value:

Since

$$\begin{array}{c} (16)_{\text{decimal}} > (15)_{\text{decimal}} \\ \text{i.e.} \\ [\$R5] > [\$R6] \end{array}$$

Then

$$result(31:0) \Rightarrow alu_to_rf(31:0) \Rightarrow writedata(31:0) = (0)_{\text{decimal}}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.

- ◆ On this clock rising edge, both source register operands represented by the intermediate signals a_in and b_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals a_in and b_in to the ALU and the ALU result ($result(31:0) = alu_to_rf(31:0) = writedata(31:0)$) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During part 3 of the waveform (clock cycle 4):**

- The ALU result value is written back into the RF (on rising clock edge):

$$rd = \$R7, \quad [\$R7] = (0)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During part 4 of the waveform (clock cycle 5):**

- The ALU result value is read out from the RF (on the rising clock edge):

$$rd = \$R7, \quad [\$R7] = (0)_{\text{decimal}}$$

- This is achieved with the following bogus instruction (for testing and debugging purposes only) which basically sets the RF to read from (and write to) \$R7:

111111	00111	00111	00111	11111	111111
-----	-----	-----	-----	-----	-----
<i>op</i>	<i>rs=\$R7</i>	<i>rt=\$R7</i>	<i>rd=\$R7</i>	<i>shamt</i>	<i>funct</i>

- The ALU calculates the next result value:

Since

$$(0)_{\text{decimal}} = (0)_{\text{decimal}}$$

i.e.

$$[\$R7] = [\$R7]$$

Then

$$result(31:0) \Rightarrow alu_to_rf(31:0) \Rightarrow writedata(31:0) = (0)_{\text{decimal}}$$

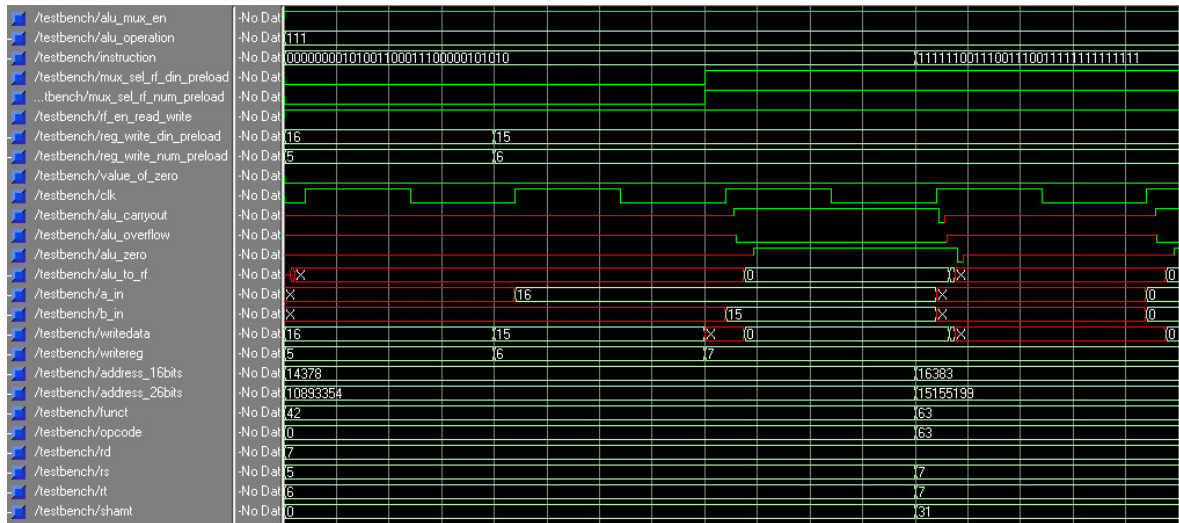


Figure B.23 Results of simulating the synthesized datapath section for R-type instructions, using ModelSim, with ALU Operation = SLT for [rs] > [rt].

As a final conclusion, it is clear that the resulting synthesized hardware functions according to the specified behaviour of datapath section for R-type instructions. This concludes the design cycle for this datapath section.

B.3.3 The Datapath Section for Load/Store Instructions

➤ RTL Description

The datapath section for the LW and SW instructions (I-format memory-reference instructions) is discussed in detail on pages 346 to 348 of [47]. Figure B.24 below shows that this datapath section is comprised of the register file, a sign extension unit, the ALU and the data memory. All these components are discussed in detail in Appendix A.

Figure B.24 illustrates that this datapath section performs a register access first, followed by a memory address calculation (base register operand plus the sign-extended offset supplied in the instruction) using the ALU, then a read from (if LW) or write to (if SW) the data memory, and finally a write into (if LW) or read from (if SW) the register file [47].

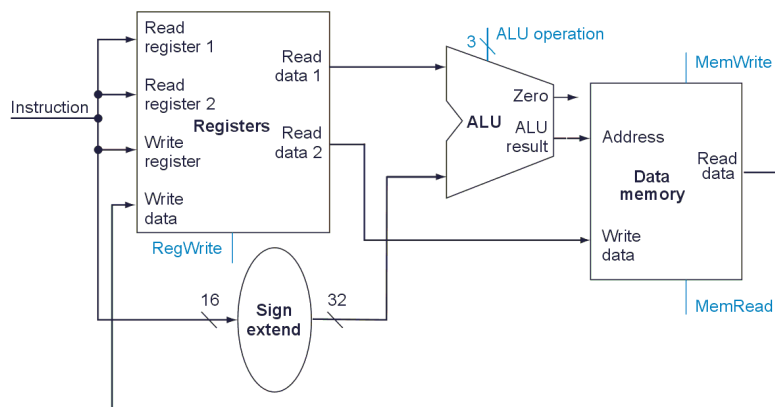


Figure B.24 The datapath section for LW and SW instructions [1, p.348].

➤ *Design Entry and Synthesis*

Schematic Editor was used to create the design entry for the datapath section for LW and SW instructions shown in figure B.24. Figure B.25 shows the final schematic diagram where there are significant additions to what is illustrated in figure B.24. These additional signals and components are the following:

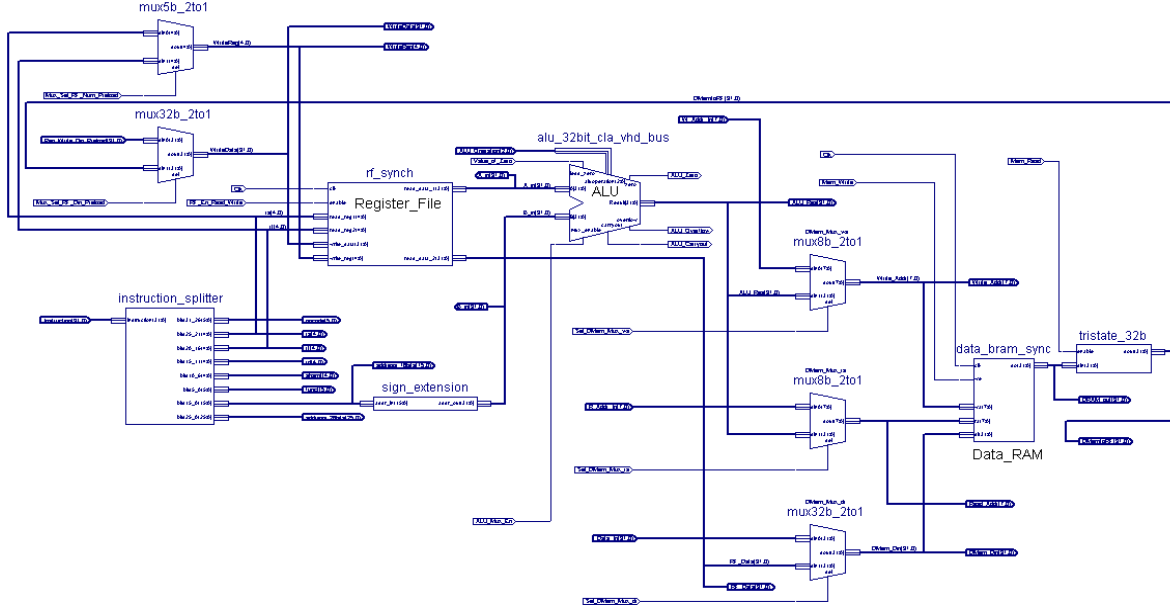


Figure B.25 Schematic diagram design entry in Schematic Editor for the datapath section for LW and SW instructions.

□ *Components:*

- **mux5b_2to1:** This is the same as discussed in section B.3.2 previously.
- **mux32b_2to1:** This is also the same as discussed in section B.3.2 previously.
- **instruction_splitter:** This is also the same as discussed in section B.3.2 previously.
- **mux8b_2to1 (DMem_Mux_wa):** This is an 8-bit/32-bit 2-to-1 multiplexor. As shown in figure B.25, it has been designed specifically for the context of this research as it truncates internally its second input *din1(31:0)* from 32 bits down to 8 bits. Through the use of its control signal *Sel_DMem_Mux_wa*, this multiplexer chooses between two sources for the 8-bit *wa(7:0)* input port of the Data_RAM. These two sources are either the 32-bit address *ALU_Res(31:0)* calculated by the ALU or an externally applied 8-bit signal called *W_Addr_In(7:0)*, which is used to select the destination memory location for preloading the data memory with the set of data values necessary for running the simulation for the whole datapath.
- **mux8b_2to1 (DMem_Mux_ra):** This is exactly similar to the above multiplexer **DMem_Mux_wa**. Through the use of its control signal *Sel_DMem_Mux_ra*, this multiplexer chooses between two sources for the 8-bit *ra(7:0)* input port of the Data_RAM. These two sources are either the 32-bit address *ALU_Res(31:0)* calculated by the ALU or an externally applied 8-bit signal called *R_Addr_In(7:0)*, which is used to select a specific memory location for reading the data value stored in there. This is necessary for debugging the simulation for

the whole datapath as it allows probing into any location in memory to check for the validity of the data stored in that location.

- **tristate_32b:** This is a 32-bit tri-state buffer. When the control signal *Mem_Read* (connected to the *enable* input port) is asserted high, the input *din(31:0)* is available at the output port *dout(31:0)* instantly. Otherwise, when the control signal *Mem_Read* is de-asserted, the output port *dout(31:0)* is in high impedance mode [47]. This combination of the tri-state buffer and its *Mem_Read* control signal is my way of implementing the *MemRead* control signal shown in figure B.24. Tri-states are discussed in detail in Appendix A as part of implementing a multiplexer with tri-state buffers.

□ **Input Signals:**

- *Value_of_Zero:* This is the same as discussed in section B.3.2 previously.

□ **Control Signals:**

- *ALU_Mux_En:* This is the same as discussed in section B.3.2 previously.
- *ALU_Operation(2:0):* This is the same as discussed in section B.3.2 previously.
- *RF_En_Read_Write:* This is the same as discussed in section B.3.2 previously.

□ **Output Probe Signals:**

These output ports are test probes extracted from internal signals for debugging purposes. As the design hierarchy gets more complicated, these probes become increasingly necessary as they make the testing and debugging process more manageable. These probe signals will be implemented more often as the design gets bigger and larger.

The output probe signals added are:

- *rd(4:0):* This is the same as discussed in section B.3.2 previously. Actually, this signal is of no importance in the context of this datapath section.
- *rs(4:0):* This is the same as discussed in section B.3.2 previously.
- *rt(4:0):* This is the same as discussed in section B.3.2 previously.
- *A_in(31:0):* This is the same as discussed in section B.3.2 previously.
- *B_in(31:0):* This is the 32-bit address output from the sign extension unit output port *addr_out(31:0)* and fed into the ALU second data input port *B(31:0)*.
- *WriteData(31:0):* This is the same as discussed in section B.3.2 previously.
- *WriteReg(4:0):* This is the same as discussed in section B.3.2 previously.
- *ALU_Res(31:0):* This is the 32-bit output result *Result(31:0)* generated by the ALU and fed into the data memory via two multiplexers..
- *DMemtoRF(31:0):* This is the 32-bit data output from the data memory fed through the 32-bit tri-state then a multiplexer into the register file.

After synthesis of the schematic diagram in figure B.25 using XST, the following VHDL code was generated:

```
-- Vhdl model created from schematic figure_5_9.sch - Sat Aug 12 10:57:07 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 figure_5_9 IS
    PORT (
        ALU_Mux_En      : IN      STD_LOGIC;
        ALU_Operation   : IN      STD_LOGIC_VECTOR (2 DOWNTO 0);
        Clk              : IN      STD_LOGIC;
        Data_In          : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        Instruction       : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        Mem_Read         : IN      STD_LOGIC;
        Mem_Write        : IN      STD_LOGIC;
        Mux_Sel_RF_Din_Preload : IN  STD_LOGIC;
        Mux_Sel_RF_Num_Preload : IN  STD_LOGIC;
        RF_En_Read_Write : IN      STD_LOGIC;
        R_Addr_In        : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        Reg_Write_Din_Preload : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
        Sel_DMem_Mux_di  : IN      STD_LOGIC;
        Sel_DMem_Mux_ra  : IN      STD_LOGIC;
        Sel_DMem_Mux_wa  : IN      STD_LOGIC;
        Value_of_Zero    : IN      STD_LOGIC;
        W_Addr_In        : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        ALU_Carryout     : OUT     STD_LOGIC;
        ALU_Overflow     : OUT     STD_LOGIC;
        ALU_Res          : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        ALU_Zero         : OUT     STD_LOGIC;
        A_in             : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        B_in             : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        DMem_Din         : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        DRAM_out         : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        RF_Data          : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        Read_Addr        : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
        WriteData        : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        WriteReg         : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
        Write_Addr       : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
        address_16bits   : OUT     STD_LOGIC_VECTOR (15 DOWNTO 0);
        address_26bits   : OUT     STD_LOGIC_VECTOR (25 DOWNTO 0);
        funct            : OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
        opcode           : OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
        rd               : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
        rs               : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
        rt               : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
        shamt            : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
    );

end figure_5_9;

ARCHITECTURE SCHEMATIC OF figure_5_9 IS
    SIGNAL ALU_Res_DUMMY      : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL A_in_DUMMY         : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL B_in_DUMMY         : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL DMem_Din_DUMMY     : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL DMemtoRF           : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL DRAM_out_DUMMY     : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL RF_Data_DUMMY      : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL Read_Addr_DUMMY    : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL WriteData_DUMMY    : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL WriteReg_DUMMY     : STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL Write_Addr_DUMMY   : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL address_16bits_DUMMY : STD_LOGIC_VECTOR (15 DOWNTO 0);
    SIGNAL rs_DUMMY           : STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL rt_DUMMY           : STD_LOGIC_VECTOR (4 DOWNTO 0);

    ATTRIBUTE BOX_TYPE : STRING;
```

```

COMPONENT alu_32bit_cla_vhd_bus
  PORT ( A      : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        B      : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        less_zero : IN      STD_LOGIC;
        carryout  : OUT     STD_LOGIC;
        overflow  : OUT     STD_LOGIC;
        Result    : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        zero      : OUT     STD_LOGIC;
        mux_enable : IN      STD_LOGIC;
        aluoperation : IN     STD_LOGIC_VECTOR (2 DOWNTO 0));
END COMPONENT;

COMPONENT data_bram_sync
  PORT ( clk      : IN      STD_LOGIC;
        we      : IN      STD_LOGIC;
        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 COMPONENT;

COMPONENT instruction_splitter
  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 COMPONENT;

COMPONENT mux32b_2to1
  PORT ( din0 : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        din1 : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout  : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        sel   : IN      STD_LOGIC);
END COMPONENT;

COMPONENT mux5b_2to1
  PORT ( sel : IN      STD_LOGIC;
        din0 : IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
        din1 : IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
        dout : OUT     STD_LOGIC_VECTOR (4 DOWNTO 0));
END COMPONENT;

COMPONENT mux8b_2to1
  PORT ( din0 : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        din1 : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout  : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
        sel   : IN      STD_LOGIC);
END COMPONENT;

COMPONENT rf_synch
  PORT ( clk      : IN      STD_LOGIC;
        enable    : IN      STD_LOGIC;
        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);
        read_data_1 : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        read_data_2 : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT sign_extension
  PORT ( addr_in : IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
        addr_out : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT tristate_32b
  PORT ( enable : IN      STD_LOGIC;

```

```

        din      :      IN      STD_LOGIC_VECTOR (31 DOWNT0 0);
        dout     :      OUT     STD_LOGIC_VECTOR (31 DOWNT0 0);
END COMPONENT;

BEGIN
    ALU_Res <= ALU_Res_DUMMY;
    A_in <= A_in_DUMMY;
    B_in <= B_in_DUMMY;
    DMem_Din <= DMem_Din_DUMMY;
    DRAM_out <= DRAM_out_DUMMY;
    RF_Data <= RF_Data_DUMMY;
    Read_Addr <= Read_Addr_DUMMY;
    WriteData <= WriteData_DUMMY;
    WriteReg <= WriteReg_DUMMY;
    Write_Addr <= Write_Addr_DUMMY;
    address_16bits <= address_16bits_DUMMY;
    rs <= rs_DUMMY;
    rt <= rt_DUMMY;

    ALU : alu_32bit_cla_vhd_bus
        PORT MAP (A(31)=>A_in_DUMMY(31), A(30)=>A_in_DUMMY(30),
            A(29)=>A_in_DUMMY(29), A(28)=>A_in_DUMMY(28), A(27)=>A_in_DUMMY(27),
            A(26)=>A_in_DUMMY(26), A(25)=>A_in_DUMMY(25), A(24)=>A_in_DUMMY(24),
            A(23)=>A_in_DUMMY(23), A(22)=>A_in_DUMMY(22), A(21)=>A_in_DUMMY(21),
            A(20)=>A_in_DUMMY(20), A(19)=>A_in_DUMMY(19), A(18)=>A_in_DUMMY(18),
            A(17)=>A_in_DUMMY(17), A(16)=>A_in_DUMMY(16), A(15)=>A_in_DUMMY(15),
            A(14)=>A_in_DUMMY(14), A(13)=>A_in_DUMMY(13), A(12)=>A_in_DUMMY(12),
            A(11)=>A_in_DUMMY(11), A(10)=>A_in_DUMMY(10), A(9)=>A_in_DUMMY(9),
            A(8)=>A_in_DUMMY(8), A(7)=>A_in_DUMMY(7), A(6)=>A_in_DUMMY(6),
            A(5)=>A_in_DUMMY(5), A(4)=>A_in_DUMMY(4), A(3)=>A_in_DUMMY(3),
            A(2)=>A_in_DUMMY(2), A(1)=>A_in_DUMMY(1), A(0)=>A_in_DUMMY(0),
            B(31)=>B_in_DUMMY(31), B(30)=>B_in_DUMMY(30), B(29)=>B_in_DUMMY(29),
            B(28)=>B_in_DUMMY(28), B(27)=>B_in_DUMMY(27), B(26)=>B_in_DUMMY(26),
            B(25)=>B_in_DUMMY(25), B(24)=>B_in_DUMMY(24), B(23)=>B_in_DUMMY(23),
            B(22)=>B_in_DUMMY(22), B(21)=>B_in_DUMMY(21), B(20)=>B_in_DUMMY(20),
            B(19)=>B_in_DUMMY(19), B(18)=>B_in_DUMMY(18), B(17)=>B_in_DUMMY(17),
            B(16)=>B_in_DUMMY(16), B(15)=>B_in_DUMMY(15), B(14)=>B_in_DUMMY(14),
            B(13)=>B_in_DUMMY(13), B(12)=>B_in_DUMMY(12), B(11)=>B_in_DUMMY(11),
            B(10)=>B_in_DUMMY(10), B(9)=>B_in_DUMMY(9), B(8)=>B_in_DUMMY(8),
            B(7)=>B_in_DUMMY(7), B(6)=>B_in_DUMMY(6), B(5)=>B_in_DUMMY(5),
            B(4)=>B_in_DUMMY(4), B(3)=>B_in_DUMMY(3), B(2)=>B_in_DUMMY(2),
            B(1)=>B_in_DUMMY(1), B(0)=>B_in_DUMMY(0), less_zero=>Value_of_Zero,
            carryout=>ALU_Carryout, overflow=>ALU_Overflow,
            Result(31)=>ALU_Res_DUMMY(31), Result(30)=>ALU_Res_DUMMY(30),
            Result(29)=>ALU_Res_DUMMY(29), Result(28)=>ALU_Res_DUMMY(28),
            Result(27)=>ALU_Res_DUMMY(27), Result(26)=>ALU_Res_DUMMY(26),
            Result(25)=>ALU_Res_DUMMY(25), Result(24)=>ALU_Res_DUMMY(24),
            Result(23)=>ALU_Res_DUMMY(23), Result(22)=>ALU_Res_DUMMY(22),
            Result(21)=>ALU_Res_DUMMY(21), Result(20)=>ALU_Res_DUMMY(20),
            Result(19)=>ALU_Res_DUMMY(19), Result(18)=>ALU_Res_DUMMY(18),
            Result(17)=>ALU_Res_DUMMY(17), Result(16)=>ALU_Res_DUMMY(16),
            Result(15)=>ALU_Res_DUMMY(15), Result(14)=>ALU_Res_DUMMY(14),
            Result(13)=>ALU_Res_DUMMY(13), Result(12)=>ALU_Res_DUMMY(12),
            Result(11)=>ALU_Res_DUMMY(11), Result(10)=>ALU_Res_DUMMY(10),
            Result(9)=>ALU_Res_DUMMY(9), Result(8)=>ALU_Res_DUMMY(8),
            Result(7)=>ALU_Res_DUMMY(7), Result(6)=>ALU_Res_DUMMY(6),
            Result(5)=>ALU_Res_DUMMY(5), Result(4)=>ALU_Res_DUMMY(4),
            Result(3)=>ALU_Res_DUMMY(3), Result(2)=>ALU_Res_DUMMY(2),
            Result(1)=>ALU_Res_DUMMY(1), Result(0)=>ALU_Res_DUMMY(0), zero=>ALU_Zero,
            mux_enable=>ALU_Mux_En, aluoperation(2)=>ALU_Operation(2),
            aluoperation(1)=>ALU_Operation(1), aluoperation(0)=>ALU_Operation(0));

    XLXI_5 : data_bram_sync
        PORT MAP (clk=>Clk, we=>Mem_Write, wa(7)=>Write_Addr_DUMMY(7),
            wa(6)=>Write_Addr_DUMMY(6), wa(5)=>Write_Addr_DUMMY(5),
            wa(4)=>Write_Addr_DUMMY(4), wa(3)=>Write_Addr_DUMMY(3),
            wa(2)=>Write_Addr_DUMMY(2), wa(1)=>Write_Addr_DUMMY(1),
            wa(0)=>Write_Addr_DUMMY(0), ra(7)=>Read_Addr_DUMMY(7),
            ra(6)=>Read_Addr_DUMMY(6), ra(5)=>Read_Addr_DUMMY(5),
            ra(4)=>Read_Addr_DUMMY(4), ra(3)=>Read_Addr_DUMMY(3),
            ra(2)=>Read_Addr_DUMMY(2), ra(1)=>Read_Addr_DUMMY(1),
            ra(0)=>Read_Addr_DUMMY(0), di(31)=>DMem_Din_DUMMY(31),
            di(30)=>DMem_Din_DUMMY(30), di(29)=>DMem_Din_DUMMY(29),

```

```

di(28)=>DMem_Din_DUMMY(28), di(27)=>DMem_Din_DUMMY(27),
di(26)=>DMem_Din_DUMMY(26), di(25)=>DMem_Din_DUMMY(25),
di(24)=>DMem_Din_DUMMY(24), di(23)=>DMem_Din_DUMMY(23),
di(22)=>DMem_Din_DUMMY(22), di(21)=>DMem_Din_DUMMY(21),
di(20)=>DMem_Din_DUMMY(20), di(19)=>DMem_Din_DUMMY(19),
di(18)=>DMem_Din_DUMMY(18), di(17)=>DMem_Din_DUMMY(17),
di(16)=>DMem_Din_DUMMY(16), di(15)=>DMem_Din_DUMMY(15),
di(14)=>DMem_Din_DUMMY(14), di(13)=>DMem_Din_DUMMY(13),
di(12)=>DMem_Din_DUMMY(12), di(11)=>DMem_Din_DUMMY(11),
di(10)=>DMem_Din_DUMMY(10), di(9)=>DMem_Din_DUMMY(9),
di(8)=>DMem_Din_DUMMY(8), di(7)=>DMem_Din_DUMMY(7),
di(6)=>DMem_Din_DUMMY(6), di(5)=>DMem_Din_DUMMY(5),
di(4)=>DMem_Din_DUMMY(4), di(3)=>DMem_Din_DUMMY(3),
di(2)=>DMem_Din_DUMMY(2), di(1)=>DMem_Din_DUMMY(1),
di(0)=>DMem_Din_DUMMY(0), do(31)=>DRAM_out_DUMMY(31),
do(30)=>DRAM_out_DUMMY(30), do(29)=>DRAM_out_DUMMY(29),
do(28)=>DRAM_out_DUMMY(28), do(27)=>DRAM_out_DUMMY(27),
do(26)=>DRAM_out_DUMMY(26), do(25)=>DRAM_out_DUMMY(25),
do(24)=>DRAM_out_DUMMY(24), do(23)=>DRAM_out_DUMMY(23),
do(22)=>DRAM_out_DUMMY(22), do(21)=>DRAM_out_DUMMY(21),
do(20)=>DRAM_out_DUMMY(20), do(19)=>DRAM_out_DUMMY(19),
do(18)=>DRAM_out_DUMMY(18), do(17)=>DRAM_out_DUMMY(17),
do(16)=>DRAM_out_DUMMY(16), do(15)=>DRAM_out_DUMMY(15),
do(14)=>DRAM_out_DUMMY(14), do(13)=>DRAM_out_DUMMY(13),
do(12)=>DRAM_out_DUMMY(12), do(11)=>DRAM_out_DUMMY(11),
do(10)=>DRAM_out_DUMMY(10), do(9)=>DRAM_out_DUMMY(9),
do(8)=>DRAM_out_DUMMY(8), do(7)=>DRAM_out_DUMMY(7),
do(6)=>DRAM_out_DUMMY(6), do(5)=>DRAM_out_DUMMY(5),
do(4)=>DRAM_out_DUMMY(4), do(3)=>DRAM_out_DUMMY(3),
do(2)=>DRAM_out_DUMMY(2), do(1)=>DRAM_out_DUMMY(1),
do(0)=>DRAM_out_DUMMY(0);

Inst_Split : instruction_splitter
PORT MAP (instruction(31)=>Instruction(31),
instruction(30)=>Instruction(30), instruction(29)=>Instruction(29),
instruction(28)=>Instruction(28), instruction(27)=>Instruction(27),
instruction(26)=>Instruction(26), instruction(25)=>Instruction(25),
instruction(24)=>Instruction(24), instruction(23)=>Instruction(23),
instruction(22)=>Instruction(22), instruction(21)=>Instruction(21),
instruction(20)=>Instruction(20), instruction(19)=>Instruction(19),
instruction(18)=>Instruction(18), instruction(17)=>Instruction(17),
instruction(16)=>Instruction(16), instruction(15)=>Instruction(15),
instruction(14)=>Instruction(14), instruction(13)=>Instruction(13),
instruction(12)=>Instruction(12), instruction(11)=>Instruction(11),
instruction(10)=>Instruction(10), instruction(9)=>Instruction(9),
instruction(8)=>Instruction(8), instruction(7)=>Instruction(7),
instruction(6)=>Instruction(6), instruction(5)=>Instruction(5),
instruction(4)=>Instruction(4), instruction(3)=>Instruction(3),
instruction(2)=>Instruction(2), instruction(1)=>Instruction(1),
instruction(0)=>Instruction(0), bits31_26(5)=>opcode(5),
bits31_26(4)=>opcode(4), bits31_26(3)=>opcode(3),
bits31_26(2)=>opcode(2), bits31_26(1)=>opcode(1),
bits31_26(0)=>opcode(0), bits25_21(4)=>rs_DUMMY(4),
bits25_21(3)=>rs_DUMMY(3), bits25_21(2)=>rs_DUMMY(2),
bits25_21(1)=>rs_DUMMY(1), bits25_21(0)=>rs_DUMMY(0),
bits20_16(4)=>rt_DUMMY(4), bits20_16(3)=>rt_DUMMY(3),
bits20_16(2)=>rt_DUMMY(2), bits20_16(1)=>rt_DUMMY(1),
bits20_16(0)=>rt_DUMMY(0), bits15_11(4)=>rd(4), bits15_11(3)=>rd(3),
bits15_11(2)=>rd(2), bits15_11(1)=>rd(1), bits15_11(0)=>rd(0),
bits10_6(4)=>shamt(4), bits10_6(3)=>shamt(3), bits10_6(2)=>shamt(2),
bits10_6(1)=>shamt(1), bits10_6(0)=>shamt(0), bits5_0(5)=>funct(5),
bits5_0(4)=>funct(4), bits5_0(3)=>funct(3), bits5_0(2)=>funct(2),
bits5_0(1)=>funct(1), bits5_0(0)=>funct(0),
bits15_0(15)=>address_16bits_DUMMY(15),
bits15_0(14)=>address_16bits_DUMMY(14),
bits15_0(13)=>address_16bits_DUMMY(13),
bits15_0(12)=>address_16bits_DUMMY(12),
bits15_0(11)=>address_16bits_DUMMY(11),
bits15_0(10)=>address_16bits_DUMMY(10),
bits15_0(9)=>address_16bits_DUMMY(9),
bits15_0(8)=>address_16bits_DUMMY(8),
bits15_0(7)=>address_16bits_DUMMY(7),
bits15_0(6)=>address_16bits_DUMMY(6),

```



```

bits15_0(5)=>address_16bits_DUMMY(5),
bits15_0(4)=>address_16bits_DUMMY(4),
bits15_0(3)=>address_16bits_DUMMY(3),
bits15_0(2)=>address_16bits_DUMMY(2),
bits15_0(1)=>address_16bits_DUMMY(1),
bits15_0(0)=>address_16bits_DUMMY(0), bits25_0(25)=>address_26bits(25),
bits25_0(24)=>address_26bits(24), bits25_0(23)=>address_26bits(23),
bits25_0(22)=>address_26bits(22), bits25_0(21)=>address_26bits(21),
bits25_0(20)=>address_26bits(20), bits25_0(19)=>address_26bits(19),
bits25_0(18)=>address_26bits(18), bits25_0(17)=>address_26bits(17),
bits25_0(16)=>address_26bits(16), bits25_0(15)=>address_26bits(15),
bits25_0(14)=>address_26bits(14), bits25_0(13)=>address_26bits(13),
bits25_0(12)=>address_26bits(12), bits25_0(11)=>address_26bits(11),
bits25_0(10)=>address_26bits(10), bits25_0(9)=>address_26bits(9),
bits25_0(8)=>address_26bits(8), bits25_0(7)=>address_26bits(7),
bits25_0(6)=>address_26bits(6), bits25_0(5)=>address_26bits(5),
bits25_0(4)=>address_26bits(4), bits25_0(3)=>address_26bits(3),
bits25_0(2)=>address_26bits(2), bits25_0(1)=>address_26bits(1),
bits25_0(0)=>address_26bits(0));

DMem_Mux_di : mux32b_2to1
PORT MAP (din0(31)=>Data_In(31), din0(30)=>Data_In(30),
din0(29)=>Data_In(29), din0(28)=>Data_In(28), din0(27)=>Data_In(27),
din0(26)=>Data_In(26), din0(25)=>Data_In(25), din0(24)=>Data_In(24),
din0(23)=>Data_In(23), din0(22)=>Data_In(22), din0(21)=>Data_In(21),
din0(20)=>Data_In(20), din0(19)=>Data_In(19), din0(18)=>Data_In(18),
din0(17)=>Data_In(17), din0(16)=>Data_In(16), din0(15)=>Data_In(15),
din0(14)=>Data_In(14), din0(13)=>Data_In(13), din0(12)=>Data_In(12),
din0(11)=>Data_In(11), din0(10)=>Data_In(10), din0(9)=>Data_In(9),
din0(8)=>Data_In(8), din0(7)=>Data_In(7), din0(6)=>Data_In(6),
din0(5)=>Data_In(5), din0(4)=>Data_In(4), din0(3)=>Data_In(3),
din0(2)=>Data_In(2), din0(1)=>Data_In(1), din0(0)=>Data_In(0),
din1(31)=>RF_Data_DUMMY(31), din1(30)=>RF_Data_DUMMY(30),
din1(29)=>RF_Data_DUMMY(29), din1(28)=>RF_Data_DUMMY(28),
din1(27)=>RF_Data_DUMMY(27), din1(26)=>RF_Data_DUMMY(26),
din1(25)=>RF_Data_DUMMY(25), din1(24)=>RF_Data_DUMMY(24),
din1(23)=>RF_Data_DUMMY(23), din1(22)=>RF_Data_DUMMY(22),
din1(21)=>RF_Data_DUMMY(21), din1(20)=>RF_Data_DUMMY(20),
din1(19)=>RF_Data_DUMMY(19), din1(18)=>RF_Data_DUMMY(18),
din1(17)=>RF_Data_DUMMY(17), din1(16)=>RF_Data_DUMMY(16),
din1(15)=>RF_Data_DUMMY(15), din1(14)=>RF_Data_DUMMY(14),
din1(13)=>RF_Data_DUMMY(13), din1(12)=>RF_Data_DUMMY(12),
din1(11)=>RF_Data_DUMMY(11), din1(10)=>RF_Data_DUMMY(10),
din1(9)=>RF_Data_DUMMY(9), din1(8)=>RF_Data_DUMMY(8),
din1(7)=>RF_Data_DUMMY(7), din1(6)=>RF_Data_DUMMY(6),
din1(5)=>RF_Data_DUMMY(5), din1(4)=>RF_Data_DUMMY(4),
din1(3)=>RF_Data_DUMMY(3), din1(2)=>RF_Data_DUMMY(2),
din1(1)=>RF_Data_DUMMY(1), din1(0)=>RF_Data_DUMMY(0),
dout(31)=>DMem_Din_DUMMY(31), dout(30)=>DMem_Din_DUMMY(30),
dout(29)=>DMem_Din_DUMMY(29), dout(28)=>DMem_Din_DUMMY(28),
dout(27)=>DMem_Din_DUMMY(27), dout(26)=>DMem_Din_DUMMY(26),
dout(25)=>DMem_Din_DUMMY(25), dout(24)=>DMem_Din_DUMMY(24),
dout(23)=>DMem_Din_DUMMY(23), dout(22)=>DMem_Din_DUMMY(22),
dout(21)=>DMem_Din_DUMMY(21), dout(20)=>DMem_Din_DUMMY(20),
dout(19)=>DMem_Din_DUMMY(19), dout(18)=>DMem_Din_DUMMY(18),
dout(17)=>DMem_Din_DUMMY(17), dout(16)=>DMem_Din_DUMMY(16),
dout(15)=>DMem_Din_DUMMY(15), dout(14)=>DMem_Din_DUMMY(14),
dout(13)=>DMem_Din_DUMMY(13), dout(12)=>DMem_Din_DUMMY(12),
dout(11)=>DMem_Din_DUMMY(11), dout(10)=>DMem_Din_DUMMY(10),
dout(9)=>DMem_Din_DUMMY(9), dout(8)=>DMem_Din_DUMMY(8),
dout(7)=>DMem_Din_DUMMY(7), dout(6)=>DMem_Din_DUMMY(6),
dout(5)=>DMem_Din_DUMMY(5), dout(4)=>DMem_Din_DUMMY(4),
dout(3)=>DMem_Din_DUMMY(3), dout(2)=>DMem_Din_DUMMY(2),
dout(1)=>DMem_Din_DUMMY(1), dout(0)=>DMem_Din_DUMMY(0),
sel=>Sel_DMem_Mux_di);

XLXI_7 : mux32b_2to1
PORT MAP (din0(31)=>Reg_Write_Din_Preload(31),
din0(30)=>Reg_Write_Din_Preload(30), din0(29)=>Reg_Write_Din_Preload(29),
din0(28)=>Reg_Write_Din_Preload(28), din0(27)=>Reg_Write_Din_Preload(27),
din0(26)=>Reg_Write_Din_Preload(26), din0(25)=>Reg_Write_Din_Preload(25),
din0(24)=>Reg_Write_Din_Preload(24), din0(23)=>Reg_Write_Din_Preload(23),
din0(22)=>Reg_Write_Din_Preload(22), din0(21)=>Reg_Write_Din_Preload(21),

```

```

din0(20)=>Reg_Write_Din_Preload(20), din0(19)=>Reg_Write_Din_Preload(19),
din0(18)=>Reg_Write_Din_Preload(18), din0(17)=>Reg_Write_Din_Preload(17),
din0(16)=>Reg_Write_Din_Preload(16), din0(15)=>Reg_Write_Din_Preload(15),
din0(14)=>Reg_Write_Din_Preload(14), din0(13)=>Reg_Write_Din_Preload(13),
din0(12)=>Reg_Write_Din_Preload(12), din0(11)=>Reg_Write_Din_Preload(11),
din0(10)=>Reg_Write_Din_Preload(10), din0(9)=>Reg_Write_Din_Preload(9),
din0(8)=>Reg_Write_Din_Preload(8), din0(7)=>Reg_Write_Din_Preload(7),
din0(6)=>Reg_Write_Din_Preload(6), din0(5)=>Reg_Write_Din_Preload(5),
din0(4)=>Reg_Write_Din_Preload(4), din0(3)=>Reg_Write_Din_Preload(3),
din0(2)=>Reg_Write_Din_Preload(2), din0(1)=>Reg_Write_Din_Preload(1),
din0(0)=>Reg_Write_Din_Preload(0), din1(31)=>DMemtoRF(31),
din1(30)=>DMemtoRF(30), din1(29)=>DMemtoRF(29), din1(28)=>DMemtoRF(28),
din1(27)=>DMemtoRF(27), din1(26)=>DMemtoRF(26), din1(25)=>DMemtoRF(25),
din1(24)=>DMemtoRF(24), din1(23)=>DMemtoRF(23), din1(22)=>DMemtoRF(22),
din1(21)=>DMemtoRF(21), din1(20)=>DMemtoRF(20), din1(19)=>DMemtoRF(19),
din1(18)=>DMemtoRF(18), din1(17)=>DMemtoRF(17), din1(16)=>DMemtoRF(16),
din1(15)=>DMemtoRF(15), din1(14)=>DMemtoRF(14), din1(13)=>DMemtoRF(13),
din1(12)=>DMemtoRF(12), din1(11)=>DMemtoRF(11), din1(10)=>DMemtoRF(10),
din1(9)=>DMemtoRF(9), din1(8)=>DMemtoRF(8), din1(7)=>DMemtoRF(7),
din1(6)=>DMemtoRF(6), din1(5)=>DMemtoRF(5), din1(4)=>DMemtoRF(4),
din1(3)=>DMemtoRF(3), din1(2)=>DMemtoRF(2), din1(1)=>DMemtoRF(1),
din1(0)=>DMemtoRF(0), dout(31)=>WriteData_DUMMY(31),
dout(30)=>WriteData_DUMMY(30), dout(29)=>WriteData_DUMMY(29),
dout(28)=>WriteData_DUMMY(28), dout(27)=>WriteData_DUMMY(27),
dout(26)=>WriteData_DUMMY(26), dout(25)=>WriteData_DUMMY(25),
dout(24)=>WriteData_DUMMY(24), dout(23)=>WriteData_DUMMY(23),
dout(22)=>WriteData_DUMMY(22), dout(21)=>WriteData_DUMMY(21),
dout(20)=>WriteData_DUMMY(20), dout(19)=>WriteData_DUMMY(19),
dout(18)=>WriteData_DUMMY(18), dout(17)=>WriteData_DUMMY(17),
dout(16)=>WriteData_DUMMY(16), dout(15)=>WriteData_DUMMY(15),
dout(14)=>WriteData_DUMMY(14), dout(13)=>WriteData_DUMMY(13),
dout(12)=>WriteData_DUMMY(12), dout(11)=>WriteData_DUMMY(11),
dout(10)=>WriteData_DUMMY(10), dout(9)=>WriteData_DUMMY(9),
dout(8)=>WriteData_DUMMY(8), dout(7)=>WriteData_DUMMY(7),
dout(6)=>WriteData_DUMMY(6), dout(5)=>WriteData_DUMMY(5),
dout(4)=>WriteData_DUMMY(4), dout(3)=>WriteData_DUMMY(3),
dout(2)=>WriteData_DUMMY(2), dout(1)=>WriteData_DUMMY(1),
dout(0)=>WriteData_DUMMY(0), sel=>Mux_Sel_RF_Din_Preload);

XLXI_6 : mux5b_2to1
PORT MAP (sel=>Mux_Sel_RF_Num_Preload, din0(4)=>rs_DUMMY(4),
din0(3)=>rs_DUMMY(3), din0(2)=>rs_DUMMY(2), din0(1)=>rs_DUMMY(1),
din0(0)=>rs_DUMMY(0), din1(4)=>rt_DUMMY(4), din1(3)=>rt_DUMMY(3),
din1(2)=>rt_DUMMY(2), din1(1)=>rt_DUMMY(1), din1(0)=>rt_DUMMY(0),
dout(4)=>WriteReg_DUMMY(4), dout(3)=>WriteReg_DUMMY(3),
dout(2)=>WriteReg_DUMMY(2), dout(1)=>WriteReg_DUMMY(1),
dout(0)=>WriteReg_DUMMY(0));

DMem_Mux_wa : mux8b_2to1
PORT MAP (din0(7)=>W_Addr_In(7), din0(6)=>W_Addr_In(6),
din0(5)=>W_Addr_In(5), din0(4)=>W_Addr_In(4), din0(3)=>W_Addr_In(3),
din0(2)=>W_Addr_In(2), din0(1)=>W_Addr_In(1), din0(0)=>W_Addr_In(0),
din1(31)=>ALU_Res_DUMMY(31), din1(30)=>ALU_Res_DUMMY(30),
din1(29)=>ALU_Res_DUMMY(29), din1(28)=>ALU_Res_DUMMY(28),
din1(27)=>ALU_Res_DUMMY(27), din1(26)=>ALU_Res_DUMMY(26),
din1(25)=>ALU_Res_DUMMY(25), din1(24)=>ALU_Res_DUMMY(24),
din1(23)=>ALU_Res_DUMMY(23), din1(22)=>ALU_Res_DUMMY(22),
din1(21)=>ALU_Res_DUMMY(21), din1(20)=>ALU_Res_DUMMY(20),
din1(19)=>ALU_Res_DUMMY(19), din1(18)=>ALU_Res_DUMMY(18),
din1(17)=>ALU_Res_DUMMY(17), din1(16)=>ALU_Res_DUMMY(16),
din1(15)=>ALU_Res_DUMMY(15), din1(14)=>ALU_Res_DUMMY(14),
din1(13)=>ALU_Res_DUMMY(13), din1(12)=>ALU_Res_DUMMY(12),
din1(11)=>ALU_Res_DUMMY(11), din1(10)=>ALU_Res_DUMMY(10),
din1(9)=>ALU_Res_DUMMY(9), din1(8)=>ALU_Res_DUMMY(8),
din1(7)=>ALU_Res_DUMMY(7), din1(6)=>ALU_Res_DUMMY(6),
din1(5)=>ALU_Res_DUMMY(5), din1(4)=>ALU_Res_DUMMY(4),
din1(3)=>ALU_Res_DUMMY(3), din1(2)=>ALU_Res_DUMMY(2),
din1(1)=>ALU_Res_DUMMY(1), din1(0)=>ALU_Res_DUMMY(0),
dout(7)=>Write_Addr_DUMMY(7), dout(6)=>Write_Addr_DUMMY(6),
dout(5)=>Write_Addr_DUMMY(5), dout(4)=>Write_Addr_DUMMY(4),
dout(3)=>Write_Addr_DUMMY(3), dout(2)=>Write_Addr_DUMMY(2),
dout(1)=>Write_Addr_DUMMY(1), dout(0)=>Write_Addr_DUMMY(0),
sel=>Sel_DMem_Mux_wa);

```

```

DMem_Mux_ra : mux8b_2to1
PORT MAP (din0(7)=>R_Addr_In(7), din0(6)=>R_Addr_In(6),
din0(5)=>R_Addr_In(5), din0(4)=>R_Addr_In(4), din0(3)=>R_Addr_In(3),
din0(2)=>R_Addr_In(2), din0(1)=>R_Addr_In(1), din0(0)=>R_Addr_In(0),
din1(31)=>ALU_Res_DUMMY(31), din1(30)=>ALU_Res_DUMMY(30),
din1(29)=>ALU_Res_DUMMY(29), din1(28)=>ALU_Res_DUMMY(28),
din1(27)=>ALU_Res_DUMMY(27), din1(26)=>ALU_Res_DUMMY(26),
din1(25)=>ALU_Res_DUMMY(25), din1(24)=>ALU_Res_DUMMY(24),
din1(23)=>ALU_Res_DUMMY(23), din1(22)=>ALU_Res_DUMMY(22),
din1(21)=>ALU_Res_DUMMY(21), din1(20)=>ALU_Res_DUMMY(20),
din1(19)=>ALU_Res_DUMMY(19), din1(18)=>ALU_Res_DUMMY(18),
din1(17)=>ALU_Res_DUMMY(17), din1(16)=>ALU_Res_DUMMY(16),
din1(15)=>ALU_Res_DUMMY(15), din1(14)=>ALU_Res_DUMMY(14),
din1(13)=>ALU_Res_DUMMY(13), din1(12)=>ALU_Res_DUMMY(12),
din1(11)=>ALU_Res_DUMMY(11), din1(10)=>ALU_Res_DUMMY(10),
din1(9)=>ALU_Res_DUMMY(9), din1(8)=>ALU_Res_DUMMY(8),
din1(7)=>ALU_Res_DUMMY(7), din1(6)=>ALU_Res_DUMMY(6),
din1(5)=>ALU_Res_DUMMY(5), din1(4)=>ALU_Res_DUMMY(4),
din1(3)=>ALU_Res_DUMMY(3), din1(2)=>ALU_Res_DUMMY(2),
din1(1)=>ALU_Res_DUMMY(1), din1(0)=>ALU_Res_DUMMY(0),
dout(7)=>Read_Addr_DUMMY(7), dout(6)=>Read_Addr_DUMMY(6),
dout(5)=>Read_Addr_DUMMY(5), dout(4)=>Read_Addr_DUMMY(4),
dout(3)=>Read_Addr_DUMMY(3), dout(2)=>Read_Addr_DUMMY(2),
dout(1)=>Read_Addr_DUMMY(1), dout(0)=>Read_Addr_DUMMY(0),
sel=>Sel_DMem_Mux_ra);

XLXI_8 : rf_synch
PORT MAP (clk=>Clk, enable=>RF_En_Read_Write, read_reg1(4)=>rs_DUMMY(4),
read_reg1(3)=>rs_DUMMY(3), read_reg1(2)=>rs_DUMMY(2),
read_reg1(1)=>rs_DUMMY(1), read_reg1(0)=>rs_DUMMY(0),
read_reg2(4)=>rt_DUMMY(4), read_reg2(3)=>rt_DUMMY(3),
read_reg2(2)=>rt_DUMMY(2), read_reg2(1)=>rt_DUMMY(1),
read_reg2(0)=>rt_DUMMY(0), write_data(31)=>WriteData_DUMMY(31),
write_data(30)=>WriteData_DUMMY(30), write_data(29)=>WriteData_DUMMY(29),
write_data(28)=>WriteData_DUMMY(28), write_data(27)=>WriteData_DUMMY(27),
write_data(26)=>WriteData_DUMMY(26), write_data(25)=>WriteData_DUMMY(25),
write_data(24)=>WriteData_DUMMY(24), write_data(23)=>WriteData_DUMMY(23),
write_data(22)=>WriteData_DUMMY(22), write_data(21)=>WriteData_DUMMY(21),
write_data(20)=>WriteData_DUMMY(20), write_data(19)=>WriteData_DUMMY(19),
write_data(18)=>WriteData_DUMMY(18), write_data(17)=>WriteData_DUMMY(17),
write_data(16)=>WriteData_DUMMY(16), write_data(15)=>WriteData_DUMMY(15),
write_data(14)=>WriteData_DUMMY(14), write_data(13)=>WriteData_DUMMY(13),
write_data(12)=>WriteData_DUMMY(12), write_data(11)=>WriteData_DUMMY(11),
write_data(10)=>WriteData_DUMMY(10), write_data(9)=>WriteData_DUMMY(9),
write_data(8)=>WriteData_DUMMY(8), write_data(7)=>WriteData_DUMMY(7),
write_data(6)=>WriteData_DUMMY(6), write_data(5)=>WriteData_DUMMY(5),
write_data(4)=>WriteData_DUMMY(4), write_data(3)=>WriteData_DUMMY(3),
write_data(2)=>WriteData_DUMMY(2), write_data(1)=>WriteData_DUMMY(1),
write_data(0)=>WriteData_DUMMY(0), write_reg(4)=>WriteReg_DUMMY(4),
write_reg(3)=>WriteReg_DUMMY(3), write_reg(2)=>WriteReg_DUMMY(2),
write_reg(1)=>WriteReg_DUMMY(1), write_reg(0)=>WriteReg_DUMMY(0),
read_data_1(31)=>A_in_DUMMY(31), read_data_1(30)=>A_in_DUMMY(30),
read_data_1(29)=>A_in_DUMMY(29), read_data_1(28)=>A_in_DUMMY(28),
read_data_1(27)=>A_in_DUMMY(27), read_data_1(26)=>A_in_DUMMY(26),
read_data_1(25)=>A_in_DUMMY(25), read_data_1(24)=>A_in_DUMMY(24),
read_data_1(23)=>A_in_DUMMY(23), read_data_1(22)=>A_in_DUMMY(22),
read_data_1(21)=>A_in_DUMMY(21), read_data_1(20)=>A_in_DUMMY(20),
read_data_1(19)=>A_in_DUMMY(19), read_data_1(18)=>A_in_DUMMY(18),
read_data_1(17)=>A_in_DUMMY(17), read_data_1(16)=>A_in_DUMMY(16),
read_data_1(15)=>A_in_DUMMY(15), read_data_1(14)=>A_in_DUMMY(14),
read_data_1(13)=>A_in_DUMMY(13), read_data_1(12)=>A_in_DUMMY(12),
read_data_1(11)=>A_in_DUMMY(11), read_data_1(10)=>A_in_DUMMY(10),
read_data_1(9)=>A_in_DUMMY(9), read_data_1(8)=>A_in_DUMMY(8),
read_data_1(7)=>A_in_DUMMY(7), read_data_1(6)=>A_in_DUMMY(6),
read_data_1(5)=>A_in_DUMMY(5), read_data_1(4)=>A_in_DUMMY(4),
read_data_1(3)=>A_in_DUMMY(3), read_data_1(2)=>A_in_DUMMY(2),
read_data_1(1)=>A_in_DUMMY(1), read_data_1(0)=>A_in_DUMMY(0),
read_data_2(31)=>RF_Data_DUMMY(31), read_data_2(30)=>RF_Data_DUMMY(30),
read_data_2(29)=>RF_Data_DUMMY(29), read_data_2(28)=>RF_Data_DUMMY(28),
read_data_2(27)=>RF_Data_DUMMY(27), read_data_2(26)=>RF_Data_DUMMY(26),
read_data_2(25)=>RF_Data_DUMMY(25), read_data_2(24)=>RF_Data_DUMMY(24),
read_data_2(23)=>RF_Data_DUMMY(23), read_data_2(22)=>RF_Data_DUMMY(22),

```

```

read_data_2(21)=>RF_Data_DUMMY(21), read_data_2(20)=>RF_Data_DUMMY(20),
read_data_2(19)=>RF_Data_DUMMY(19), read_data_2(18)=>RF_Data_DUMMY(18),
read_data_2(17)=>RF_Data_DUMMY(17), read_data_2(16)=>RF_Data_DUMMY(16),
read_data_2(15)=>RF_Data_DUMMY(15), read_data_2(14)=>RF_Data_DUMMY(14),
read_data_2(13)=>RF_Data_DUMMY(13), read_data_2(12)=>RF_Data_DUMMY(12),
read_data_2(11)=>RF_Data_DUMMY(11), read_data_2(10)=>RF_Data_DUMMY(10),
read_data_2(9)=>RF_Data_DUMMY(9), read_data_2(8)=>RF_Data_DUMMY(8),
read_data_2(7)=>RF_Data_DUMMY(7), read_data_2(6)=>RF_Data_DUMMY(6),
read_data_2(5)=>RF_Data_DUMMY(5), read_data_2(4)=>RF_Data_DUMMY(4),
read_data_2(3)=>RF_Data_DUMMY(3), read_data_2(2)=>RF_Data_DUMMY(2),
read_data_2(1)=>RF_Data_DUMMY(1), read_data_2(0)=>RF_Data_DUMMY(0));

Sign_Extender : sign_extension
PORT MAP (addr_in(15)=>address_16bits_DUMMY(15),
addr_in(14)=>address_16bits_DUMMY(14),
addr_in(13)=>address_16bits_DUMMY(13),
addr_in(12)=>address_16bits_DUMMY(12),
addr_in(11)=>address_16bits_DUMMY(11),
addr_in(10)=>address_16bits_DUMMY(10),
addr_in(9)=>address_16bits_DUMMY(9), addr_in(8)=>address_16bits_DUMMY(8),
addr_in(7)=>address_16bits_DUMMY(7), addr_in(6)=>address_16bits_DUMMY(6),
addr_in(5)=>address_16bits_DUMMY(5), addr_in(4)=>address_16bits_DUMMY(4),
addr_in(3)=>address_16bits_DUMMY(3), addr_in(2)=>address_16bits_DUMMY(2),
addr_in(1)=>address_16bits_DUMMY(1), addr_in(0)=>address_16bits_DUMMY(0),
addr_out(31)=>B_in_DUMMY(31), addr_out(30)=>B_in_DUMMY(30),
addr_out(29)=>B_in_DUMMY(29), addr_out(28)=>B_in_DUMMY(28),
addr_out(27)=>B_in_DUMMY(27), addr_out(26)=>B_in_DUMMY(26),
addr_out(25)=>B_in_DUMMY(25), addr_out(24)=>B_in_DUMMY(24),
addr_out(23)=>B_in_DUMMY(23), addr_out(22)=>B_in_DUMMY(22),
addr_out(21)=>B_in_DUMMY(21), addr_out(20)=>B_in_DUMMY(20),
addr_out(19)=>B_in_DUMMY(19), addr_out(18)=>B_in_DUMMY(18),
addr_out(17)=>B_in_DUMMY(17), addr_out(16)=>B_in_DUMMY(16),
addr_out(15)=>B_in_DUMMY(15), addr_out(14)=>B_in_DUMMY(14),
addr_out(13)=>B_in_DUMMY(13), addr_out(12)=>B_in_DUMMY(12),
addr_out(11)=>B_in_DUMMY(11), addr_out(10)=>B_in_DUMMY(10),
addr_out(9)=>B_in_DUMMY(9), addr_out(8)=>B_in_DUMMY(8),
addr_out(7)=>B_in_DUMMY(7), addr_out(6)=>B_in_DUMMY(6),
addr_out(5)=>B_in_DUMMY(5), addr_out(4)=>B_in_DUMMY(4),
addr_out(3)=>B_in_DUMMY(3), addr_out(2)=>B_in_DUMMY(2),
addr_out(1)=>B_in_DUMMY(1), addr_out(0)=>B_in_DUMMY(0));

XLXI_3 : tristate_32b
PORT MAP (enable=>Mem_Read, din(31)=>DRAM_out_DUMMY(31),
din(30)=>DRAM_out_DUMMY(30), din(29)=>DRAM_out_DUMMY(29),
din(28)=>DRAM_out_DUMMY(28), din(27)=>DRAM_out_DUMMY(27),
din(26)=>DRAM_out_DUMMY(26), din(25)=>DRAM_out_DUMMY(25),
din(24)=>DRAM_out_DUMMY(24), din(23)=>DRAM_out_DUMMY(23),
din(22)=>DRAM_out_DUMMY(22), din(21)=>DRAM_out_DUMMY(21),
din(20)=>DRAM_out_DUMMY(20), din(19)=>DRAM_out_DUMMY(19),
din(18)=>DRAM_out_DUMMY(18), din(17)=>DRAM_out_DUMMY(17),
din(16)=>DRAM_out_DUMMY(16), din(15)=>DRAM_out_DUMMY(15),
din(14)=>DRAM_out_DUMMY(14), din(13)=>DRAM_out_DUMMY(13),
din(12)=>DRAM_out_DUMMY(12), din(11)=>DRAM_out_DUMMY(11),
din(10)=>DRAM_out_DUMMY(10), din(9)=>DRAM_out_DUMMY(9),
din(8)=>DRAM_out_DUMMY(8), din(7)=>DRAM_out_DUMMY(7),
din(6)=>DRAM_out_DUMMY(6), din(5)=>DRAM_out_DUMMY(5),
din(4)=>DRAM_out_DUMMY(4), din(3)=>DRAM_out_DUMMY(3),
din(2)=>DRAM_out_DUMMY(2), din(1)=>DRAM_out_DUMMY(1),
din(0)=>DRAM_out_DUMMY(0), dout(31)=>DMemtoRF(31),
dout(30)=>DMemtoRF(30), dout(29)=>DMemtoRF(29), dout(28)=>DMemtoRF(28),
dout(27)=>DMemtoRF(27), dout(26)=>DMemtoRF(26), dout(25)=>DMemtoRF(25),
dout(24)=>DMemtoRF(24), dout(23)=>DMemtoRF(23), dout(22)=>DMemtoRF(22),
dout(21)=>DMemtoRF(21), dout(20)=>DMemtoRF(20), dout(19)=>DMemtoRF(19),
dout(18)=>DMemtoRF(18), dout(17)=>DMemtoRF(17), dout(16)=>DMemtoRF(16),
dout(15)=>DMemtoRF(15), dout(14)=>DMemtoRF(14), dout(13)=>DMemtoRF(13),
dout(12)=>DMemtoRF(12), dout(11)=>DMemtoRF(11), dout(10)=>DMemtoRF(10),
dout(9)=>DMemtoRF(9), dout(8)=>DMemtoRF(8), dout(7)=>DMemtoRF(7),
dout(6)=>DMemtoRF(6), dout(5)=>DMemtoRF(5), dout(4)=>DMemtoRF(4),
dout(3)=>DMemtoRF(3), dout(2)=>DMemtoRF(2), dout(1)=>DMemtoRF(1),
dout(0)=>DMemtoRF(0));

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the datapath section for the load and store instructions, was generated. Figure B.26 shows the resulting top level RTL symbol while figure B.27 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are covered in detail in Appendix A.



Figure B.26 Resulting top level RTL symbol for the datapath section for LW and SW instructions.

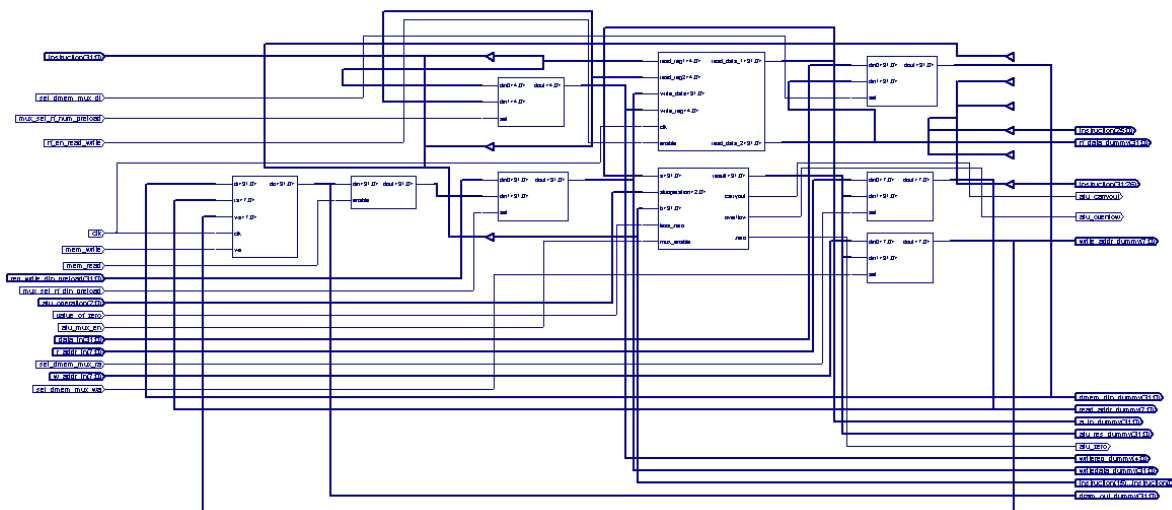


Figure B.27 Resulting top level RTL schematic for the datapath section for LW and SW instructions.

➤ *FPGA Device Synthesis Summary*

After the hardware implementation for this datapath section 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 : 448

Macro Statistics:
RAM : 3
256x32-bit dual-port block RAM: 1
32x32-bit dual-port block RAM: 2
Tristates : 11
32-bit tristate buffer : 5
5-bit tristate buffer : 2
8-bit tristate buffer : 4

Cell Usage:
BELS : 684
and2 : 96
and2b1 : 32
and3 : 64
and3b1 : 64
gnd : 3
inv : 33
LUT1 : 70
LUT2 : 1
LUT2_L : 1
LUT3 : 34
LUT3_D : 1
LUT3_L : 2
LUT4 : 24
LUT4_D : 10
LUT4_L : 9
muxcy : 2
muxcy_1 : 6
muxf5 : 34
or2 : 161
vcc : 3
xor2 : 2
xor3 : 32
RAMS : 3
RAMB16_S36_S36 : 3
Tri-States : 202
BUFT : 202
Clock Buffers : 1
BUFGRP : 1
IO Buffers : 447
IBUF : 125
OBUF : 322
Logical : 8
nor4 : 8
Others : 8
fmap : 8

Device utilization summary:

Number of Slices:	83	out of	46592	0%
Number of 4 input LUTs:	152	out of	93184	0%
Number of bonded IOBs:	447	out of	1108	40%
Number of TBUFs:	202	out of	23296	0%
Number of BRAMs:	3	out of	168	1%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

Minimum period: 18.331ns (Maximum Frequency: 54.552MHz)
Minimum input arrival time before clock: 20.522ns
Maximum output required time after clock: 28.065ns
Maximum combinational path delay: 29.804ns
Maximum combinational path delay: 29.804ns

➤ *Place-and-Route onto the FPGA*

In figure B.28, FPGA Editor shows the synthesized datapath section for LW and SW instructions after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the upper section of the FPGA chip.

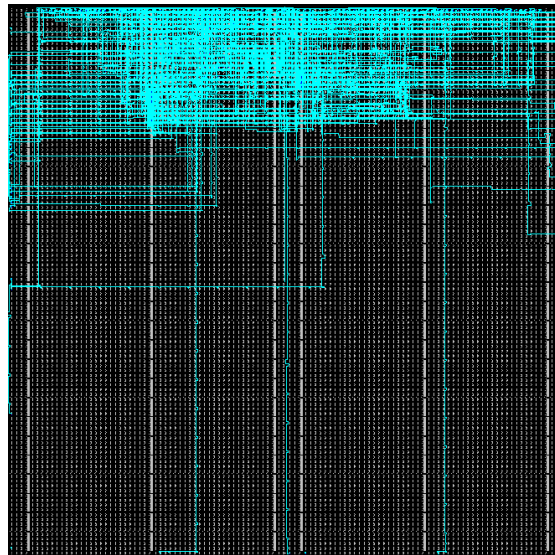


Figure B.28 *FPGA Editor showing the synthesized datapath section for LW and SW instructions after place-and-route onto the target Virtex-II FPGA chip.*

➤ *Simulation Results*

Figures B.29 and B.30 show the waveform results of simulating in ModelSim the VHDL behavioural model for the synthesized datapath section for LW and SW instructions.

Before elaborating and analysing these simulation waveforms, a few key points must first be highlighted which also pertain to the schematic diagram of figure B.25. These key points are:

- ❑ Although the whole 32-bit instruction input is supplied, yet the 3-bit *ALU_Operation(2:0)* control signal must yet be supplied independently during these simulations. This is acceptable only for now as this datapath section is being tested in isolation from the complete datapath in which the Control Unit supplies the *ALU_Operation(2:0)* control signal by extracting it from both the *op* and *funct* fields within the supplied instruction. This is discussed in detail in Appendix C and Chapter 6.
- ❑
- ❑ In figures B.29 and B.30, the I-format instructions simulated are LW and SW, respectively.
- ❑ The terminology that follows is based on the detailed elaboration on the syntax for these instructions, which is found in Section 5.4.2 in Chapter 5.
- ❑ In simulating both these instructions, the testing sequence of operations varies depending on the instruction:
 - ❖ For LW (figure B.29):
 - First, the RF is preloaded with the register operand for *rs*, which would be later used as the base register for calculating the target memory address. Also, this target memory location in Data_RAM is preloaded with the data.

- Then, the ALU adds the value stored in register *rs* in the register file to the 32-bit sign extended address to calculate the result value *Result(31:0)* which is the target memory address to be read.
 - The ALU result value *Result(31:0)* is truncated inside the multiplexer mux32b_2to1 from 32 bits down to 8 bits and then fed into the read address *ra(7:0)* input port of the Data_RAM.
 - Finally, the output *do(31:0)* from the Data_RAM passes through the tri-state buffer then mux32b_2to1 before it is written into the destination register *rt* inside the register file.
- ❖ For SW (figure B.30):
- First, the RF is preloaded with the register operand for *rs*, which would be later used as the base register for calculating the memory address. Also, the source register *rt* inside the register file is preloaded with the data, which would be later stored in (written to) Data_RAM.
 - Then, the ALU adds the value stored in register *rs* in the register file to the 32-bit sign extended address to calculate the result value *Result(31:0)* which is the target memory address to be written.
 - The ALU result value *Result(31:0)* is truncated inside the multiplexer mux32b_2to1 from 32 bits down to 8 bits and then fed into the write address *wa(7:0)* input port of the Data_RAM.
 - Finally, the output *read_data_2(31:0)* from the register file passes through the multiplexer mux32b_2to1 before it is written into the destination (target) memory address inside the Data_RAM.
- ❑ In figures B.29 and B.30, some of the signals are in binary format while others are in decimal (for ease of debugging).

Following is the detailed elaboration and analysis for these simulation waveforms:

➤ *Simulation for LW*

Figure B.29 shows the simulation waveforms for LW by selecting ALU Operation = ADD = $(010)_{\text{binary}} = (2)_{\text{decimal}}$. There is no ALU Operation specifically for LW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address from which to load the data.

The simulation waveform is divided into 5 parts to facilitate analysis and discussion:

❑ **During part 1 of the waveform (clock cycle 1):**

- Preload the register file with the register operand for the base register:

$rs \Rightarrow write_reg = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

- Preload the data memory with the data value:

$Target\ Memory\ Location = 25$, $[Memory[25]] = (5678)_{\text{decimal}}$

- Test Instruction:

LW \$R6 , 10 (\$R5)

 rt *offset* (*rs*)

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

100011 00101 00110 000000000001010

op=35 *rs*=\$R5 *rt*=\$R6 *offset*=10

❑ **During part 2 of the waveform (clock cycle 2):**

- The ALU calculates the result value = target memory address:

$$\begin{aligned}
 result(31:0) => alu_res(31:0) &= A_in(31:0) && + B_in(31:0) \\
 &= [read_reg1(4:0)] && + addr_out(31:0) \\
 &= [$R5] && + addr_out(31:0) \\
 &= (15)_{decimal} && + (10)_{decimal} \\
 &= (25)_{decimal}
 \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, the first source register operand represented by the intermediate signals *a_in* is read out from the RF and fed into the A input of the ALU.
- ◆ Delay between application of the signal *a_in* to the ALU and the ALU result (*result(31:0) = alu_res(31:0) = read_addr(7:0)*) stabilizing to the correct value = 0.5 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

❑ **During part 3 of the waveform (clock cycle 3):**

- The ALU result value *result(31:0) => alu_res(31:0)* which is the target memory address is passed through the multiplexer mux8b_2to1 (Dmem_Mux_ra) and gets truncated from 32 bits down to 8 bits in order to drive the read address *ra(7:0)* input of the data memory which then reads out the value from the target memory address of 25:

$$alu_res(31:0) => ra(7:0) = (25)_{decimal}$$

- This yields the data output *do(31:0)* from the Data_RAM which passes through the tri-state buffer then mux32b_2to1 before it is written into the destination register *rt* inside the register file:

❑ **During part 4 of the waveform (clock cycle 4):**

- The destination register *rt* inside the register file is loaded (written) with the value just read from memory in cycle 3:

$$rt => write_reg = \$R6 \quad , \quad [\$R6] = (5678)_{decimal}$$

- Important Note: The register file could not be written in clock cycle 3 since the data to be written was not available until exactly the rising clock edge. This condition did not allow the register file the required and sufficient setup time of 1 ns prior to the rising clock edge,

and therefore had to wait until the next rising clock edge to perform the write into the register file.

❑ **During part 5 of the waveform (clock cycle 5):**

- The signal *rf_data* is inspected for debugging purposes, which confirms that the data value just stored in (loaded into) destination register *rt* is read out from the RF (on the rising clock edge):

rt => *read_reg2* = \$R6 , [\$R6] = (5678)_{decimal}

- This is achieved with the same instruction as above:

```
100011 00101 00110 0000000000001010
-----
op=35  rs=$R5  rt=$R6  offset=10
```

❑ **Conclusions:**

- The execution of the LW instruction takes exactly 3 clock cycles. These are clock cycles no. 2, 3, and 4 in Figure B.29.
- Actually, total elapsed execution time is 4 clock cycles when including clock cycle no. 1, which is for the data pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.

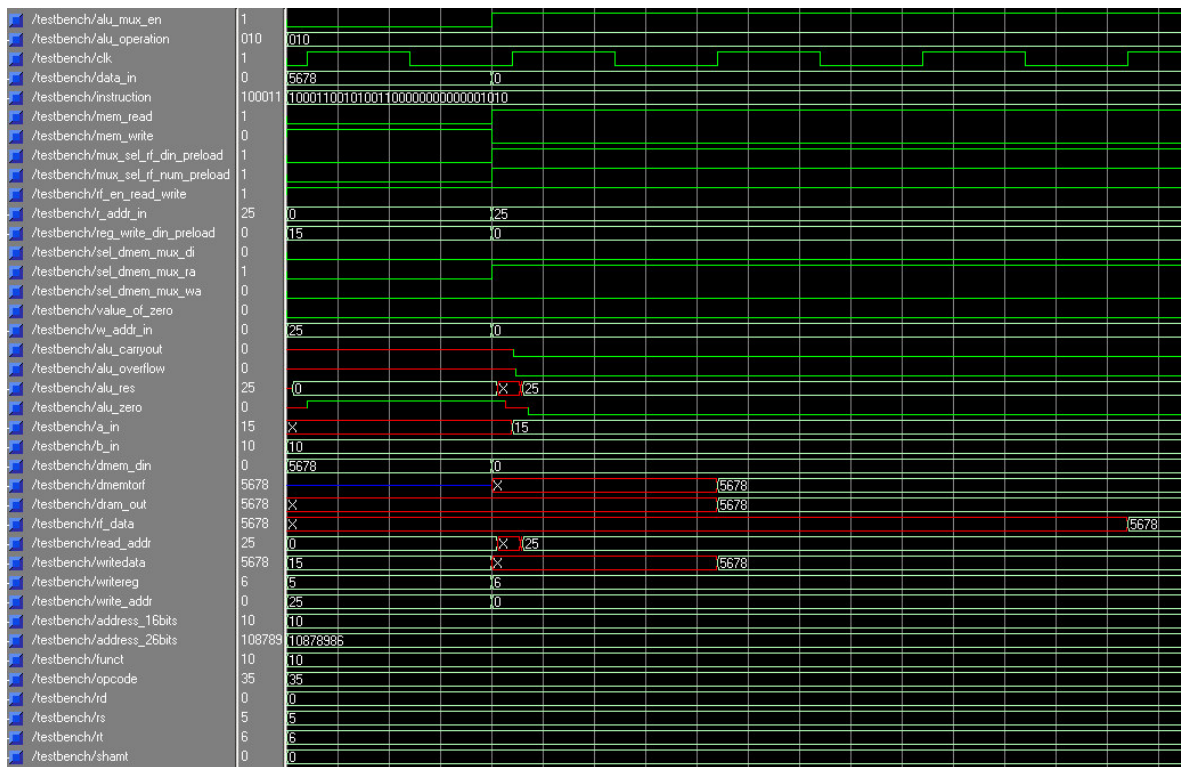


Figure B.29 Results of simulating the synthesized datapath section for LW instruction, using ModelSim, with ALU Operation = ADD.

➤ **Simulation for SW**

Figure B.30 shows the simulation waveforms for SW by selecting ALU Operation = ADD = (010)_{binary} = (2)_{decimal}. Similar to LW, there is no ALU Operation specifically for SW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address for storing the data.

The simulation waveform is divided into 5 parts to facilitate analysis and discussion:

❑ **During part 1 of the waveform (clock cycle 1):**

- Preload the register file with the register operand for the base register:

$rs \Rightarrow write_reg = \$R5$, $[\$R5] = (15)_{decimal}$

- Test Instruction:

SW	\$R6 ,	10	(\$R5)
	-----	-----	
	<i>rt</i>	<i>offset</i>	<i>(rs)</i>

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

101011	00101	00110	0000000000001010
-----	-----	-----	-----
<i>op=43</i>	<i>rs=\$R5</i>	<i>rt=\$R6</i>	<i>offset=10</i>

❑ **During part 2 of the waveform (clock cycle 2):**

- Preload the register file with the data value which is to be stored into the memory later:

$rt \Rightarrow write_reg = \$R6$, $[\$R6] = (5678)_{decimal}$

- Simultaneously, the ALU calculates the result value = target memory address:

$result(31:0) \Rightarrow alu_res(31:0)$	$= A_in(31:0)$	$+ B_in(31:0)$
	$= [read_reg1(4:0)]$	$+ addr_out(31:0)$
	$= [\$R5]$	$+ addr_out(31:0)$
	$= (15)_{decimal}$	$+ (10)_{decimal}$
	$= (25)_{decimal}$	

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, the first source register operand represented by the intermediate signals a_in is read out from the RF and fed into the A input of the ALU.
- ◆ Delay between application of the signal a_in to the ALU and the ALU result ($result(31:0) = alu_res(31:0) = write_addr(7:0)$) stabilizing to the correct value = 0.5 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

❑ **During part 3 of the waveform (clock cycle 3):**

- The ALU result value $result(31:0) \Rightarrow alu_res(31:0)$ which is the target memory address is passed through the multiplexer $mux8b_2to1$ ($Dmem_Mux_wa$) and gets truncated from 32 bits down to 8 bits in order to drive the write address $wa(7:0)$ input of the data memory:

$alu_res(31:0) \Rightarrow ra(7:0) = (25)_{decimal}$

- Simultaneously, the data value (register operand) in source register rt is read out from the register file and is available and ready to be written (stored) into the target memory address 25 in the data memory:

$rt \Rightarrow read_reg2 = \$R6$, $[\$R6] = (5678)_{decimal}$

$read_data2 \Rightarrow RF_Data \Rightarrow Dmem_Din \Rightarrow di = (5678)_{decimal}$

❑ **During part 4 of the waveform (clock cycle 4):**

- The data value (register operand) in source register rt just read out from the register file (in the previous clock cycle) is now written (stored) into the target memory address 25 in the data memory:

$Target\ Memory\ Location = 25$, $[Memory[25]] = (5678)_{decimal}$

❑ **During part 5 of the waveform (clock cycle 5):**

- The signal $dram_out$ is inspected for debugging purposes, which confirms that the data value just stored in (loaded into) the target memory address 25 is read out from the data memory (on the rising clock edge):

$Target\ Memory\ Location = 25$, $[Memory[25]] = (5678)_{decimal}$

- This is achieved with the same instruction as above:

101011 00101 00110 000000000001010

 $op=43$ $rs=\$R5$ $rt=\$R6$ $offset=10$

❑ **Conclusions:**

- Similar to LW, the execution of the SW instruction takes exactly 3 clock cycles. These are clock cycles no. 2, 3, and 4 in Figure B.30.
- Actually, total elapsed execution time is 4 clock cycles when including clock cycle no. 1, which is for the first part of the 2-clock-cycle data pre-loading phase.
- Therefore, it is worth noting that in the case of SW, the end of the data pre-loading phase and the start of the instruction execution over-lap in clock cycle no. 2. This is shown in Figure B.30 where the data pre-loading phase is during the first 2 clock cycles (consecutively pre-load rs then rt into RF as shown in Figure B.30), while the instruction execution starts at clock cycle no. 2 during which time the ALU already calculates and generates the output result alu_res .
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.

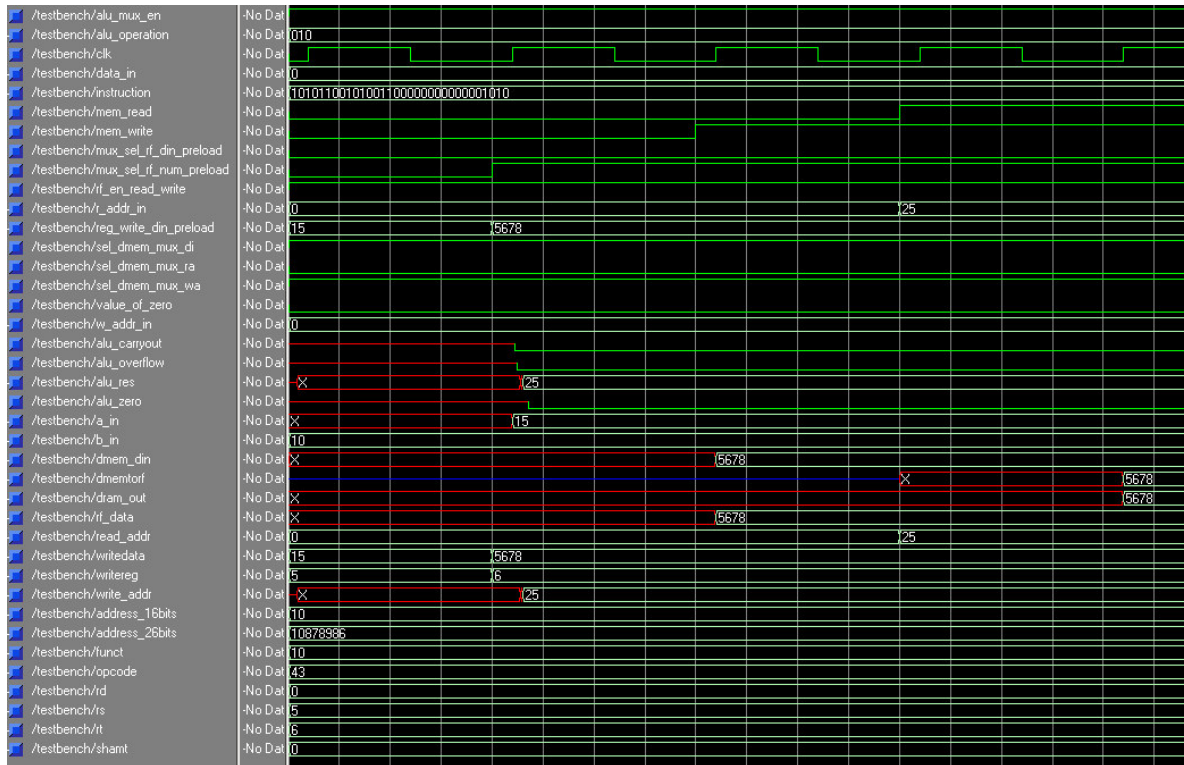


Figure B.30 Results of simulating the synthesized datapath section for SW instruction, using ModelSim, with ALU Operation = ADD.

As a final conclusion, it is clear that the resulting synthesized hardware functions according to the specified behaviour of datapath section for the LW and SW instructions. This concludes the design cycle for this datapath section.

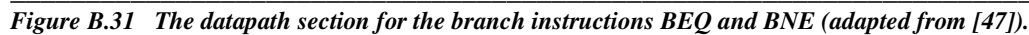
B.3.4 The Datapath Section for Branch Instructions

➤ RTL Description

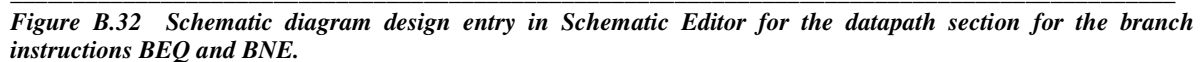
The datapath section for both branch instructions BEQ and BNE (I-format branching instructions) is discussed in detail on pages 347 to 349 of [47]. Figure B.31 below shows that this datapath section is comprised of the register file, an address truncator unit, the ALU and an adder. All these components are discussed in detail in Appendix A.

Figure B.31 illustrates that this datapath section performs a register access to read the value of both source register operands (*[rs]* and *[rt]*), then performs a SUB operation by the ALU to test for equality [47]. At the same time, the 16-bit offset supplied by the instruction is truncated down to 8 bits and added to the incremented program counter (PC+1) using the adder, to calculate the branch target address [47].

It is worth noting here that this datapath section for the BEQ and BNE instructions ties in very closely with the instruction fetch unit to decide which address to fetch the next instruction from (depending on the outcome of the register operands comparison). This will be covered in section B.4 next when putting together the complete datapath. Also as will be seen later, it is a matter of asserting and de-asserting the proper control signals in this complete datapath to choose the desired functionality of BEQ or BNE.



Schematic Editor was used to create the design entry for the datapath section for BEQ and BNE instructions shown in figure B.31. Figure B.32 shows the final schematic diagram.



After synthesis of the schematic diagram in figure B.32 using XST, the following VHDL code was generated:

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\add8.sch - Sat Aug
12 11:10:34 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 ADD8_MXILINX_figure_5_10 IS
    PORT ( A      : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          B      : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          CI      : IN      STD_LOGIC;
          CO      : OUT     STD_LOGIC;
          OFL     : OUT     STD_LOGIC;
          S      : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));

end ADD8_MXILINX_figure_5_10;

ARCHITECTURE SCHEMATIC OF ADD8_MXILINX_figure_5_10 IS
    SIGNAL C0      : STD_LOGIC;
    SIGNAL C1      : STD_LOGIC;
    SIGNAL C2      : STD_LOGIC;
    SIGNAL C3      : STD_LOGIC;
    SIGNAL C4      : STD_LOGIC;
    SIGNAL C5      : STD_LOGIC;
    SIGNAL C6      : STD_LOGIC;
    SIGNAL C6O     : STD_LOGIC;
    SIGNAL CO_DUMMY : STD_LOGIC;
    SIGNAL I0      : STD_LOGIC;
    SIGNAL I1      : STD_LOGIC;
    SIGNAL I2      : STD_LOGIC;
    SIGNAL I3      : STD_LOGIC;
    SIGNAL I4      : STD_LOGIC;
    SIGNAL I5      : STD_LOGIC;
    SIGNAL I6      : STD_LOGIC;
    SIGNAL I7      : STD_LOGIC;
    SIGNAL dummy   : STD_LOGIC;

    ATTRIBUTE BOX_TYPE : STRING;
    ATTRIBUTE RLOC : STRING ;
    ATTRIBUTE RLOC OF I_36_16 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_17 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_23 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_22 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_18 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_19 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_20 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_21 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_64 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_107 : LABEL IS "X0Y3";
    ATTRIBUTE RLOC OF I_36_110 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_63 : LABEL IS "X0Y2";
    ATTRIBUTE RLOC OF I_36_58 : LABEL IS "X0Y1";
    ATTRIBUTE RLOC OF I_36_111 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_55 : LABEL IS "X0Y0";
    ATTRIBUTE RLOC OF I_36_62 : LABEL IS "X0Y1";

    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 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_D
  PORT ( CI : IN STD_LOGIC;
        DI : IN STD_LOGIC;
        S : IN STD_LOGIC;
        LO : OUT STD_LOGIC;
        O : OUT STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY_D : 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 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";
COMPONENT XORCY
  PORT ( CI : IN STD_LOGIC;
        LI : IN STD_LOGIC;
        O : OUT STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XORCY : COMPONENT IS "BLACK_BOX";
BEGIN
  CO <= CO_DUMMY;
  I_36_16 : FMAP
    PORT MAP (I1=>A(0), I2=>B(0), I3=>dummy, I4=>dummy, O=>I0);
  I_36_17 : FMAP
    PORT MAP (I1=>A(1), I2=>B(1), I3=>dummy, I4=>dummy, O=>I1);
  I_36_23 : FMAP
    PORT MAP (I1=>A(7), I2=>B(7), I3=>dummy, I4=>dummy, O=>I7);
  I_36_22 : FMAP
    PORT MAP (I1=>A(6), I2=>B(6), I3=>dummy, I4=>dummy, O=>I6);
  I_36_18 : FMAP
    PORT MAP (I1=>A(2), I2=>B(2), I3=>dummy, I4=>dummy, O=>I2);
  I_36_19 : FMAP
    PORT MAP (I1=>A(3), I2=>B(3), I3=>dummy, I4=>dummy, O=>I3);
  I_36_20 : FMAP
    PORT MAP (I1=>A(4), I2=>B(4), I3=>dummy, I4=>dummy, O=>I4);
  I_36_21 : FMAP
    PORT MAP (I1=>A(5), I2=>B(5), I3=>dummy, I4=>dummy, O=>I5);
  I_36_64 : MUXCY
    PORT MAP (CI=>C6, DI=>A(7), S=>I7, O=>CO_DUMMY);
  I_36_107 : MUXCY_D
    PORT MAP (CI=>C5, DI=>A(6), S=>I6, LO=>C6, O=>C6O);
  I_36_110 : MUXCY_L
    PORT MAP (CI=>C4, DI=>A(5), S=>I5, LO=>C5);
  I_36_63 : MUXCY_L
    PORT MAP (CI=>C3, DI=>A(4), S=>I4, LO=>C4);
  I_36_58 : MUXCY_L
    PORT MAP (CI=>C2, DI=>A(3), S=>I3, LO=>C3);
  I_36_111 : MUXCY_L
    PORT MAP (CI=>CI, DI=>A(0), S=>I0, LO=>C0);
  I_36_55 : MUXCY_L
    PORT MAP (CI=>C0, DI=>A(1), S=>I1, LO=>C1);
  I_36_62 : MUXCY_L
    PORT MAP (CI=>C1, DI=>A(2), S=>I2, LO=>C2);

```



```

I_36_239 : XOR2
  PORT MAP (I0=>C60, I1=>CO_DUMMY, O=>OFL);
I_36_230 : XOR2
  PORT MAP (I0=>A(2), I1=>B(2), O=>I2);
I_36_229 : XOR2
  PORT MAP (I0=>A(1), I1=>B(1), O=>I1);
I_36_228 : XOR2
  PORT MAP (I0=>A(0), I1=>B(0), O=>I0);
I_36_224 : XOR2
  PORT MAP (I0=>A(4), I1=>B(4), O=>I4);
I_36_223 : XOR2
  PORT MAP (I0=>A(5), I1=>B(5), O=>I5);
I_36_222 : XOR2
  PORT MAP (I0=>A(6), I1=>B(6), O=>I6);
I_36_225 : XOR2
  PORT MAP (I0=>A(3), I1=>B(3), O=>I3);
I_36_221 : XOR2
  PORT MAP (I0=>A(7), I1=>B(7), O=>I7);
I_36_80 : XORCY
  PORT MAP (CI=>C6, LI=>I7, O=>S(7));
I_36_73 : XORCY
  PORT MAP (CI=>CI, LI=>I0, O=>S(0));
I_36_74 : XORCY
  PORT MAP (CI=>C0, LI=>I1, O=>S(1));
I_36_76 : XORCY
  PORT MAP (CI=>C1, LI=>I2, O=>S(2));
I_36_75 : XORCY
  PORT MAP (CI=>C2, LI=>I3, O=>S(3));
I_36_78 : XORCY
  PORT MAP (CI=>C3, LI=>I4, O=>S(4));
I_36_77 : XORCY
  PORT MAP (CI=>C4, LI=>I5, O=>S(5));
I_36_81 : XORCY
  PORT MAP (CI=>C5, LI=>I6, O=>S(6));

END SCHEMATIC;

-- Vhdl model created from schematic figure_5_10.sch - Sat Aug 12 11:10:35 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 figure_5_10 IS
  PORT (
    ALU_Mux_En      : IN      STD_LOGIC;
    ALU_Operation   : IN      STD_LOGIC_VECTOR (2 DOWNTO 0);
    Clk              : IN      STD_LOGIC;
    Instruction      : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
    Mux_Sel_RF_Num_Preload : IN  STD_LOGIC;
    PCplus1_Addr     : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
    Reg_En_Read_Write : IN      STD_LOGIC;
    Value_of_0       : IN      STD_LOGIC;
    Write_Data       : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
    ALU_Carryout     : OUT      STD_LOGIC;
    ALU_Overflow     : OUT      STD_LOGIC;
    ALU_Res          : OUT      STD_LOGIC_VECTOR (31 DOWNTO 0);
    ALU_Zero         : OUT      STD_LOGIC;
    A_in             : OUT      STD_LOGIC_VECTOR (31 DOWNTO 0);
    B_in             : OUT      STD_LOGIC_VECTOR (31 DOWNTO 0);
    Branch_Target    : OUT      STD_LOGIC_VECTOR (7 DOWNTO 0);
    Ignore_1         : OUT      STD_LOGIC;
    Ignore_2         : OUT      STD_LOGIC;
    WriteReg         : OUT      STD_LOGIC_VECTOR (4 DOWNTO 0);
    address_16bits    : OUT      STD_LOGIC_VECTOR (15 DOWNTO 0);
    address_26bits    : OUT      STD_LOGIC_VECTOR (25 DOWNTO 0);
    address_8bits     : OUT      STD_LOGIC_VECTOR (7 DOWNTO 0);
    funct            : OUT      STD_LOGIC_VECTOR (5 DOWNTO 0);
    opcode           : OUT      STD_LOGIC_VECTOR (5 DOWNTO 0);
    rd               : OUT      STD_LOGIC_VECTOR (4 DOWNTO 0);
  );

```

```

        rs          :      OUT      STD_LOGIC_VECTOR (4 DOWNTO 0);
        rt          :      OUT      STD_LOGIC_VECTOR (4 DOWNTO 0);
        shamt       :      OUT      STD_LOGIC_VECTOR (4 DOWNTO 0));

end figure_5_10;

ARCHITECTURE SCHEMATIC OF figure_5_10 IS
    SIGNAL A_in_DUMMY      :      STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL B_in_DUMMY      :      STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL WriteReg_DUMMY  :      STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL address_16bits_DUMMY : STD_LOGIC_VECTOR (15 DOWNTO 0);
    SIGNAL address_8bits_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL rs_DUMMY        :      STD_LOGIC_VECTOR (4 DOWNTO 0);
    SIGNAL rt_DUMMY        :      STD_LOGIC_VECTOR (4 DOWNTO 0);

    ATTRIBUTE BOX_TYPE : STRING;
    ATTRIBUTE U_SET : STRING ;
    ATTRIBUTE U_SET OF XLXI_8 : LABEL IS "XLXI_8_0";

    COMPONENT ADD8_MXILINX_figure_5_10
        PORT ( A      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
              B      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
              CI      :      IN      STD_LOGIC;
              CO      :      OUT     STD_LOGIC;
              OFL     :      OUT     STD_LOGIC;
              S      :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
    END COMPONENT;

    COMPONENT alu_32bit_cla_vhd_bus
        PORT ( A      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              B      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
              less_zero :      IN      STD_LOGIC;
              carryout  :      OUT     STD_LOGIC;
              overflow  :      OUT     STD_LOGIC;
              Result    :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
              zero      :      OUT     STD_LOGIC;
              mux_enable :      IN      STD_LOGIC;
              aluoperation :      IN     STD_LOGIC_VECTOR (2 DOWNTO 0));
    END COMPONENT;

    COMPONENT instruction_splitter
        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 COMPONENT;

    COMPONENT mux5b_2to1
        PORT ( sel      :      IN      STD_LOGIC;
              din0      :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
              din1      :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
              dout      :      OUT     STD_LOGIC_VECTOR (4 DOWNTO 0));
    END COMPONENT;

    COMPONENT rf_synch
        PORT ( clk      :      IN      STD_LOGIC;
              enable    :      IN      STD_LOGIC;
              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);
              read_data_1 :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
              read_data_2 :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
    END COMPONENT;

    COMPONENT sign_dextension
        PORT ( addr_in   :      IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
              addr_out   :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
    END COMPONENT;

```

```

END COMPONENT;

BEGIN
  A_in <= A_in_DUMMY;
  B_in <= B_in_DUMMY;
  WriteReg <= WriteReg_DUMMY;
  address_16bits <= address_16bits_DUMMY;
  address_8bits <= address_8bits_DUMMY;
  rs <= rs_DUMMY;
  rt <= rt_DUMMY;

  XLXI_8 : ADD8_MXILINX_figure_5_10
    PORT MAP (A(7)=>PCplus1_Addr(7), A(6)=>PCplus1_Addr(6),
      A(5)=>PCplus1_Addr(5), A(4)=>PCplus1_Addr(4), A(3)=>PCplus1_Addr(3),
      A(2)=>PCplus1_Addr(2), A(1)=>PCplus1_Addr(1), A(0)=>PCplus1_Addr(0),
      B(7)=>address_8bits_DUMMY(7), B(6)=>address_8bits_DUMMY(6),
      B(5)=>address_8bits_DUMMY(5), B(4)=>address_8bits_DUMMY(4),
      B(3)=>address_8bits_DUMMY(3), B(2)=>address_8bits_DUMMY(2),
      B(1)=>address_8bits_DUMMY(1), B(0)=>address_8bits_DUMMY(0),
      C1=>Value_of_0, CO=>Ignore_2, OFL=>Ignore_1, S(7)=>Branch_Target(7),
      S(6)=>Branch_Target(6), S(5)=>Branch_Target(5), S(4)=>Branch_Target(4),
      S(3)=>Branch_Target(3), S(2)=>Branch_Target(2), S(1)=>Branch_Target(1),
      S(0)=>Branch_Target(0));

  XLXI_3 : alu_32bit_cla_vhd_bus
    PORT MAP (A(31)=>A_in_DUMMY(31), A(30)=>A_in_DUMMY(30),
      A(29)=>A_in_DUMMY(29), A(28)=>A_in_DUMMY(28), A(27)=>A_in_DUMMY(27),
      A(26)=>A_in_DUMMY(26), A(25)=>A_in_DUMMY(25), A(24)=>A_in_DUMMY(24),
      A(23)=>A_in_DUMMY(23), A(22)=>A_in_DUMMY(22), A(21)=>A_in_DUMMY(21),
      A(20)=>A_in_DUMMY(20), A(19)=>A_in_DUMMY(19), A(18)=>A_in_DUMMY(18),
      A(17)=>A_in_DUMMY(17), A(16)=>A_in_DUMMY(16), A(15)=>A_in_DUMMY(15),
      A(14)=>A_in_DUMMY(14), A(13)=>A_in_DUMMY(13), A(12)=>A_in_DUMMY(12),
      A(11)=>A_in_DUMMY(11), A(10)=>A_in_DUMMY(10), A(9)=>A_in_DUMMY(9),
      A(8)=>A_in_DUMMY(8), A(7)=>A_in_DUMMY(7), A(6)=>A_in_DUMMY(6),
      A(5)=>A_in_DUMMY(5), A(4)=>A_in_DUMMY(4), A(3)=>A_in_DUMMY(3),
      A(2)=>A_in_DUMMY(2), A(1)=>A_in_DUMMY(1), A(0)=>A_in_DUMMY(0),
      B(31)=>B_in_DUMMY(31), B(30)=>B_in_DUMMY(30), B(29)=>B_in_DUMMY(29),
      B(28)=>B_in_DUMMY(28), B(27)=>B_in_DUMMY(27), B(26)=>B_in_DUMMY(26),
      B(25)=>B_in_DUMMY(25), B(24)=>B_in_DUMMY(24), B(23)=>B_in_DUMMY(23),
      B(22)=>B_in_DUMMY(22), B(21)=>B_in_DUMMY(21), B(20)=>B_in_DUMMY(20),
      B(19)=>B_in_DUMMY(19), B(18)=>B_in_DUMMY(18), B(17)=>B_in_DUMMY(17),
      B(16)=>B_in_DUMMY(16), B(15)=>B_in_DUMMY(15), B(14)=>B_in_DUMMY(14),
      B(13)=>B_in_DUMMY(13), B(12)=>B_in_DUMMY(12), B(11)=>B_in_DUMMY(11),
      B(10)=>B_in_DUMMY(10), B(9)=>B_in_DUMMY(9), B(8)=>B_in_DUMMY(8),
      B(7)=>B_in_DUMMY(7), B(6)=>B_in_DUMMY(6), B(5)=>B_in_DUMMY(5),
      B(4)=>B_in_DUMMY(4), B(3)=>B_in_DUMMY(3), B(2)=>B_in_DUMMY(2),
      B(1)=>B_in_DUMMY(1), B(0)=>B_in_DUMMY(0), less_zero=>Value_of_0,
      carryout=>ALU_Carryout, overflow=>ALU_Overflow, Result(31)=>ALU_Res(31),
      Result(30)=>ALU_Res(30), Result(29)=>ALU_Res(29),
      Result(28)=>ALU_Res(28), Result(27)=>ALU_Res(27),
      Result(26)=>ALU_Res(26), Result(25)=>ALU_Res(25),
      Result(24)=>ALU_Res(24), Result(23)=>ALU_Res(23),
      Result(22)=>ALU_Res(22), Result(21)=>ALU_Res(21),
      Result(20)=>ALU_Res(20), Result(19)=>ALU_Res(19),
      Result(18)=>ALU_Res(18), Result(17)=>ALU_Res(17),
      Result(16)=>ALU_Res(16), Result(15)=>ALU_Res(15),
      Result(14)=>ALU_Res(14), Result(13)=>ALU_Res(13),
      Result(12)=>ALU_Res(12), Result(11)=>ALU_Res(11),
      Result(10)=>ALU_Res(10), Result(9)=>ALU_Res(9), Result(8)=>ALU_Res(8),
      Result(7)=>ALU_Res(7), Result(6)=>ALU_Res(6), Result(5)=>ALU_Res(5),
      Result(4)=>ALU_Res(4), Result(3)=>ALU_Res(3), Result(2)=>ALU_Res(2),
      Result(1)=>ALU_Res(1), Result(0)=>ALU_Res(0), zero=>ALU_Zero,
      mux_enable=>ALU_Mux_En, aluoperation(2)=>ALU_Operation(2),
      aluoperation(1)=>ALU_Operation(1), aluoperation(0)=>ALU_Operation(0));

  Inst_Split : instruction_splitter
    PORT MAP (instruction(31)=>Instruction(31),
      instruction(30)=>Instruction(30), instruction(29)=>Instruction(29),
      instruction(28)=>Instruction(28), instruction(27)=>Instruction(27),
      instruction(26)=>Instruction(26), instruction(25)=>Instruction(25),
      instruction(24)=>Instruction(24), instruction(23)=>Instruction(23),
      instruction(22)=>Instruction(22), instruction(21)=>Instruction(21),
      instruction(20)=>Instruction(20), instruction(19)=>Instruction(19),

```

```

instruction(18)=>Instruction(18), instruction(17)=>Instruction(17),
instruction(16)=>Instruction(16), instruction(15)=>Instruction(15),
instruction(14)=>Instruction(14), instruction(13)=>Instruction(13),
instruction(12)=>Instruction(12), instruction(11)=>Instruction(11),
instruction(10)=>Instruction(10), instruction(9)=>Instruction(9),
instruction(8)=>Instruction(8), instruction(7)=>Instruction(7),
instruction(6)=>Instruction(6), instruction(5)=>Instruction(5),
instruction(4)=>Instruction(4), instruction(3)=>Instruction(3),
instruction(2)=>Instruction(2), instruction(1)=>Instruction(1),
instruction(0)=>Instruction(0), bits31_26(5)=>opcode(5),
bits31_26(4)=>opcode(4), bits31_26(3)=>opcode(3),
bits31_26(2)=>opcode(2), bits31_26(1)=>opcode(1),
bits31_26(0)=>opcode(0), bits25_21(4)=>rs_DUMMY(4),
bits25_21(3)=>rs_DUMMY(3), bits25_21(2)=>rs_DUMMY(2),
bits25_21(1)=>rs_DUMMY(1), bits25_21(0)=>rs_DUMMY(0),
bits20_16(4)=>rt_DUMMY(4), bits20_16(3)=>rt_DUMMY(3),
bits20_16(2)=>rt_DUMMY(2), bits20_16(1)=>rt_DUMMY(1),
bits20_16(0)=>rt_DUMMY(0), bits15_11(4)=>rd(4), bits15_11(3)=>rd(3),
bits15_11(2)=>rd(2), bits15_11(1)=>rd(1), bits15_11(0)=>rd(0),
bits10_6(4)=>shamt(4), bits10_6(3)=>shamt(3), bits10_6(2)=>shamt(2),
bits10_6(1)=>shamt(1), bits10_6(0)=>shamt(0), bits5_0(5)=>funct(5),
bits5_0(4)=>funct(4), bits5_0(3)=>funct(3), bits5_0(2)=>funct(2),
bits5_0(1)=>funct(1), bits5_0(0)=>funct(0),
bits15_0(15)=>address_16bits_DUMMY(15),
bits15_0(14)=>address_16bits_DUMMY(14),
bits15_0(13)=>address_16bits_DUMMY(13),
bits15_0(12)=>address_16bits_DUMMY(12),
bits15_0(11)=>address_16bits_DUMMY(11),
bits15_0(10)=>address_16bits_DUMMY(10),
bits15_0(9)=>address_16bits_DUMMY(9),
bits15_0(8)=>address_16bits_DUMMY(8),
bits15_0(7)=>address_16bits_DUMMY(7),
bits15_0(6)=>address_16bits_DUMMY(6),
bits15_0(5)=>address_16bits_DUMMY(5),
bits15_0(4)=>address_16bits_DUMMY(4),
bits15_0(3)=>address_16bits_DUMMY(3),
bits15_0(2)=>address_16bits_DUMMY(2),
bits15_0(1)=>address_16bits_DUMMY(1),
bits15_0(0)=>address_16bits_DUMMY(0), bits25_0(25)=>address_26bits(25),
bits25_0(24)=>address_26bits(24), bits25_0(23)=>address_26bits(23),
bits25_0(22)=>address_26bits(22), bits25_0(21)=>address_26bits(21),
bits25_0(20)=>address_26bits(20), bits25_0(19)=>address_26bits(19),
bits25_0(18)=>address_26bits(18), bits25_0(17)=>address_26bits(17),
bits25_0(16)=>address_26bits(16), bits25_0(15)=>address_26bits(15),
bits25_0(14)=>address_26bits(14), bits25_0(13)=>address_26bits(13),
bits25_0(12)=>address_26bits(12), bits25_0(11)=>address_26bits(11),
bits25_0(10)=>address_26bits(10), bits25_0(9)=>address_26bits(9),
bits25_0(8)=>address_26bits(8), bits25_0(7)=>address_26bits(7),
bits25_0(6)=>address_26bits(6), bits25_0(5)=>address_26bits(5),
bits25_0(4)=>address_26bits(4), bits25_0(3)=>address_26bits(3),
bits25_0(2)=>address_26bits(2), bits25_0(1)=>address_26bits(1),
bits25_0(0)=>address_26bits(0));

XLXI_6 : mux5b_2to1
PORT MAP (sel=>Mux_Sel_RF_Num_Preload, din0(4)=>rs_DUMMY(4),
din0(3)=>rs_DUMMY(3), din0(2)=>rs_DUMMY(2), din0(1)=>rs_DUMMY(1),
din0(0)=>rs_DUMMY(0), din1(4)=>rt_DUMMY(4), din1(3)=>rt_DUMMY(3),
din1(2)=>rt_DUMMY(2), din1(1)=>rt_DUMMY(1), din1(0)=>rt_DUMMY(0),
dout(4)=>WriteReg_DUMMY(4), dout(3)=>WriteReg_DUMMY(3),
dout(2)=>WriteReg_DUMMY(2), dout(1)=>WriteReg_DUMMY(1),
dout(0)=>WriteReg_DUMMY(0));

XLXI_13 : rf_synch
PORT MAP (clk=>Clk, enable=>Reg_En_Read_Write, read_reg1(4)=>rs_DUMMY(4),
read_reg1(3)=>rs_DUMMY(3), read_reg1(2)=>rs_DUMMY(2),
read_reg1(1)=>rs_DUMMY(1), read_reg1(0)=>rs_DUMMY(0),
read_reg2(4)=>rt_DUMMY(4), read_reg2(3)=>rt_DUMMY(3),
read_reg2(2)=>rt_DUMMY(2), read_reg2(1)=>rt_DUMMY(1),
read_reg2(0)=>rt_DUMMY(0), write_data(31)=>Write_Data(31),
write_data(30)=>Write_Data(30), write_data(29)=>Write_Data(29),
write_data(28)=>Write_Data(28), write_data(27)=>Write_Data(27),
write_data(26)=>Write_Data(26), write_data(25)=>Write_Data(25),
write_data(24)=>Write_Data(24), write_data(23)=>Write_Data(23),

```

```

write_data(22)=>Write_Data(22), write_data(21)=>Write_Data(21),
write_data(20)=>Write_Data(20), write_data(19)=>Write_Data(19),
write_data(18)=>Write_Data(18), write_data(17)=>Write_Data(17),
write_data(16)=>Write_Data(16), write_data(15)=>Write_Data(15),
write_data(14)=>Write_Data(14), write_data(13)=>Write_Data(13),
write_data(12)=>Write_Data(12), write_data(11)=>Write_Data(11),
write_data(10)=>Write_Data(10), write_data(9)=>Write_Data(9),
write_data(8)=>Write_Data(8), write_data(7)=>Write_Data(7),
write_data(6)=>Write_Data(6), write_data(5)=>Write_Data(5),
write_data(4)=>Write_Data(4), write_data(3)=>Write_Data(3),
write_data(2)=>Write_Data(2), write_data(1)=>Write_Data(1),
write_data(0)=>Write_Data(0), write_reg(4)=>WriteReg_DUMMY(4),
write_reg(3)=>WriteReg_DUMMY(3), write_reg(2)=>WriteReg_DUMMY(2),
write_reg(1)=>WriteReg_DUMMY(1), write_reg(0)=>WriteReg_DUMMY(0),
read_data_1(31)=>A_in_DUMMY(31), read_data_1(30)=>A_in_DUMMY(30),
read_data_1(29)=>A_in_DUMMY(29), read_data_1(28)=>A_in_DUMMY(28),
read_data_1(27)=>A_in_DUMMY(27), read_data_1(26)=>A_in_DUMMY(26),
read_data_1(25)=>A_in_DUMMY(25), read_data_1(24)=>A_in_DUMMY(24),
read_data_1(23)=>A_in_DUMMY(23), read_data_1(22)=>A_in_DUMMY(22),
read_data_1(21)=>A_in_DUMMY(21), read_data_1(20)=>A_in_DUMMY(20),
read_data_1(19)=>A_in_DUMMY(19), read_data_1(18)=>A_in_DUMMY(18),
read_data_1(17)=>A_in_DUMMY(17), read_data_1(16)=>A_in_DUMMY(16),
read_data_1(15)=>A_in_DUMMY(15), read_data_1(14)=>A_in_DUMMY(14),
read_data_1(13)=>A_in_DUMMY(13), read_data_1(12)=>A_in_DUMMY(12),
read_data_1(11)=>A_in_DUMMY(11), read_data_1(10)=>A_in_DUMMY(10),
read_data_1(9)=>A_in_DUMMY(9), read_data_1(8)=>A_in_DUMMY(8),
read_data_1(7)=>A_in_DUMMY(7), read_data_1(6)=>A_in_DUMMY(6),
read_data_1(5)=>A_in_DUMMY(5), read_data_1(4)=>A_in_DUMMY(4),
read_data_1(3)=>A_in_DUMMY(3), read_data_1(2)=>A_in_DUMMY(2),
read_data_1(1)=>A_in_DUMMY(1), read_data_1(0)=>A_in_DUMMY(0),
read_data_2(31)=>B_in_DUMMY(31), read_data_2(30)=>B_in_DUMMY(30),
read_data_2(29)=>B_in_DUMMY(29), read_data_2(28)=>B_in_DUMMY(28),
read_data_2(27)=>B_in_DUMMY(27), read_data_2(26)=>B_in_DUMMY(26),
read_data_2(25)=>B_in_DUMMY(25), read_data_2(24)=>B_in_DUMMY(24),
read_data_2(23)=>B_in_DUMMY(23), read_data_2(22)=>B_in_DUMMY(22),
read_data_2(21)=>B_in_DUMMY(21), read_data_2(20)=>B_in_DUMMY(20),
read_data_2(19)=>B_in_DUMMY(19), read_data_2(18)=>B_in_DUMMY(18),
read_data_2(17)=>B_in_DUMMY(17), read_data_2(16)=>B_in_DUMMY(16),
read_data_2(15)=>B_in_DUMMY(15), read_data_2(14)=>B_in_DUMMY(14),
read_data_2(13)=>B_in_DUMMY(13), read_data_2(12)=>B_in_DUMMY(12),
read_data_2(11)=>B_in_DUMMY(11), read_data_2(10)=>B_in_DUMMY(10),
read_data_2(9)=>B_in_DUMMY(9), read_data_2(8)=>B_in_DUMMY(8),
read_data_2(7)=>B_in_DUMMY(7), read_data_2(6)=>B_in_DUMMY(6),
read_data_2(5)=>B_in_DUMMY(5), read_data_2(4)=>B_in_DUMMY(4),
read_data_2(3)=>B_in_DUMMY(3), read_data_2(2)=>B_in_DUMMY(2),
read_data_2(1)=>B_in_DUMMY(1), read_data_2(0)=>B_in_DUMMY(0));

XLI14 : sign_dextension
PORT MAP (addr_in(15)=>address_16bits_DUMMY(15),
addr_in(14)=>address_16bits_DUMMY(14),
addr_in(13)=>address_16bits_DUMMY(13),
addr_in(12)=>address_16bits_DUMMY(12),
addr_in(11)=>address_16bits_DUMMY(11),
addr_in(10)=>address_16bits_DUMMY(10),
addr_in(9)=>address_16bits_DUMMY(9), addr_in(8)=>address_16bits_DUMMY(8),
addr_in(7)=>address_16bits_DUMMY(7), addr_in(6)=>address_16bits_DUMMY(6),
addr_in(5)=>address_16bits_DUMMY(5), addr_in(4)=>address_16bits_DUMMY(4),
addr_in(3)=>address_16bits_DUMMY(3), addr_in(2)=>address_16bits_DUMMY(2),
addr_in(1)=>address_16bits_DUMMY(1), addr_in(0)=>address_16bits_DUMMY(0),
addr_out(7)=>address_8bits_DUMMY(7), addr_out(6)=>address_8bits_DUMMY(6),
addr_out(5)=>address_8bits_DUMMY(5), addr_out(4)=>address_8bits_DUMMY(4),
addr_out(3)=>address_8bits_DUMMY(3), addr_out(2)=>address_8bits_DUMMY(2),
addr_out(1)=>address_8bits_DUMMY(1), addr_out(0)=>address_8bits_DUMMY(0));

```

END SCHEMATIC;

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the datapath section for the BEQ and BNE instructions, was generated. Figure B.33 shows the resulting top level RTL symbol while

figure B.34 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are covered in detail in Appendix A.

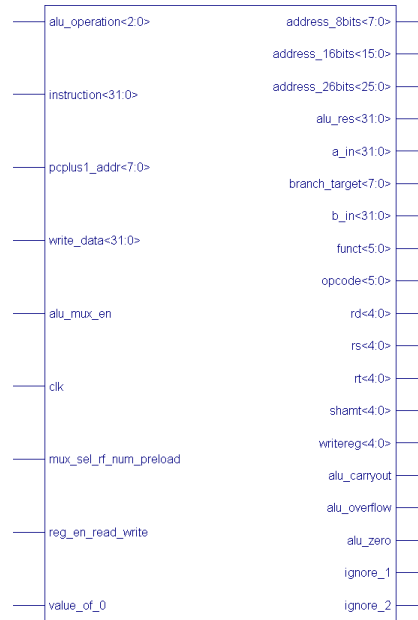


Figure B.33 Resulting top level RTL symbol for the synthesized datapath section for the branch instructions BEQ and BNE.

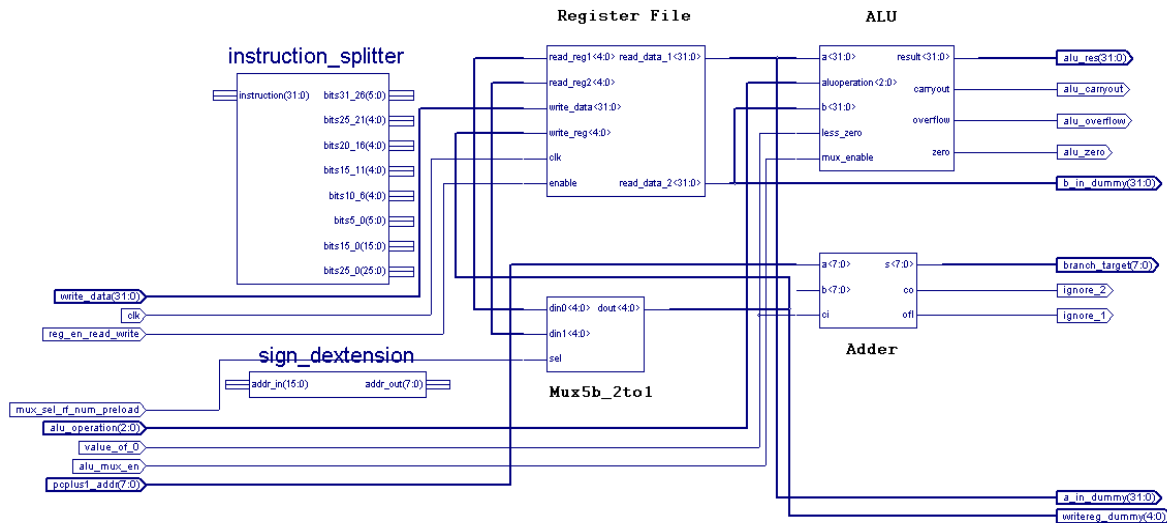


Figure B.34 Resulting top level RTL schematic for the synthesized datapath section for the branch instructions BEQ and BNE.

➤ FPGA Device Synthesis Summary

After the hardware implementation for this datapath section 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                                     : 276

Macro Statistics:
# RAM                                     : 2
#      32x32-bit dual-port block RAM: 2
# Tristates                             : 2
#      5-bit tristate buffer           : 2

Cell Usage:
# BELS                                  : 683
#      and2                             : 96
#      and2b1                           : 32
#      and3                             : 64
#      and3b1                           : 64
#      GND                              : 4
#      inv                              : 33
#      LUT1                             : 65
#      LUT3                             : 32
#      LUT4                             : 30
#      muxcy                            : 3
#      muxcy_d                          : 1
#      muxcy_l                          : 12
#      muxf5                            : 32
#      or2                              : 161
#      VCC                              : 3
#      xor2                             : 11
#      xor3                             : 32
#      xorcy                            : 8
# RAMS                                  : 2
#      RAMB16_S36_S36                  : 2
# Tri-States                           : 10
#      BUFT                             : 10
# Clock Buffers                         : 1
#      BUFGP                           : 1
# IO Buffers                            : 275
#      IBUF                             : 79
#      OBUF                             : 196
# Logical                               : 8
#      nor4                             : 8
# Others                                : 16
#      fmap                             : 16

Device utilization summary:

Number of Slices:                        72 out of 46592      0%
Number of 4 input LUTs:                  127 out of 93184     0%
Number of bonded IOBs:                   275 out of 1108     24%
Number of TBUFs:                         10 out of 23296      0%
Number of BRAMs:                         2 out of 168         1%
Number of GCLKs:                         1 out of 16          6%

Timing Summary:

Minimum period: No path found
Minimum input arrival time before clock: 5.440ns
Maximum output required time after clock: 36.075ns
Maximum combinational path delay: 34.750ns

```

➤ *Place-and-Route onto the FPGA*

In figure B.35, FPGA Editor shows the synthesized datapath section for BEQ and BNE instructions after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the upper and lower section of the FPGA chip.

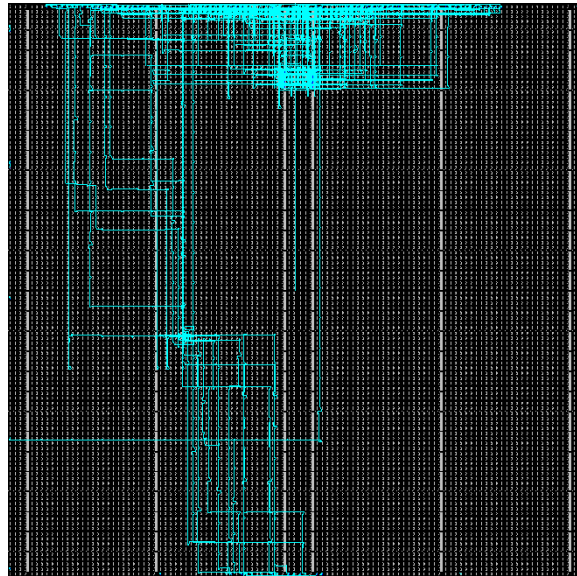


Figure B.35 FPGA Editor showing the synthesized datapath section for the branch instructions BEQ and BNE.

➤ Simulation Results

Figures B.36 and B.37 show the waveform results of simulating in ModelSim the VHDL behavioural model for the synthesized datapath section for the branch instructions BEQ and BNE by selecting ALU Operation = SUB = $(110)_{\text{binary}} = (6)_{\text{decimal}}$. There is no ALU Operation specifically for BEQ/BNE, but instead a SUB is all that is needed since the ALU performs a subtraction $([rs] - [rt])$ to compute the zero signal *ALU_Zero*.

Figures B.36 and B.37 show the simulation waveforms for the two conditions $[rs] = [rt]$ and $[rs] \neq [rt]$, respectively.

Before elaborating and analysing these simulation waveforms, a few key points must first be highlighted which also pertain to the schematic diagram of figure B.32. These key points are:

- ❑ Although the whole 32-bit instruction input is supplied, yet the 3-bit *ALU_Operation(2:0)* control signal must yet be supplied independently during these simulations. This is acceptable only for now as this datapath section is being tested in isolation from the complete datapath in which the Control Unit supplies the *ALU_Operation(2:0)* control signal by extracting it from both the *op* and *funct* fields within the supplied instruction. This is discussed in detail in Appendix C and Chapter 6.
- ❑ Figures B.36 and B.37 show the simulation for the I-format branch instructions in general, whether BEQ or BNE. This is due to the fact that the only difference in the hardware implementation for each of these instructions, is whether or not the output signal *ALU_Zero* is inverted before it is fed to the branch control logic, as will be discussed later in section B.5.
- ❑ The terminology that follows is based on the detailed elaboration on the syntax for these instructions, which is found in Section 5.4.2 in Chapter 5.
- ❑ In simulating each of these instructions (BEQ and BNE), the testing sequence of operations is the same:
 - First, the RF is preloaded with the register operands for *rs* and *rt*. This is done during the first two consecutive clock cycles.

- Then, the ALU performs a subtraction by subtracting the register operand for *rt* from the register operand for *rs*. The result from this subtraction decides the value of the output signal *ALU_Zero*:

If [*\$Rs*] = [*\$Rt*] Then *ALU_Zero* = 1

Else *ALU_Zero* = 0
 - At the same time, the branch target address *Branch_Target(7:0)* is calculated by adding the address/offset supplied by the instruction to the address value of PC+1 supplied by the instruction fetch stage.
 - However, none of the resulting generated signals mentioned above is required to be stored in any state element.
- In figures B.36 and B.37, some of the signals are in binary format while others are in decimal (for ease of debugging).

The waveform for each condition is divided into 3 parts to facilitate analysis and discussion. Following is the detailed elaboration and analysis for these simulation waveforms:

❖ **Condition 1 (Figure B.36): When [*rs*] = [*rt*]**

□ **During part 1 of the waveform (clock cycle 1):**

- Preload the register file with the register operand for *rs*:

$$rs = \$R5, \quad [\$R5] = (15)_{\text{decimal}}$$

- Test Instruction:

$$\begin{array}{ccc} \text{BEQ} & \$R5, & \$R6, \quad -10 \\ \hline & rs & rt \quad offset \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccc} 000100 & 00101 & 00110 & 111111111110110 \\ \hline op=4 & rs=\$R5 & rt=\$R6 & offset=-10 \end{array}$$

- The adder computes the branch target address *Branch_Target(7:0)* by adding the value of PC + 1 (supplied by the instruction fetch unit/stage/datapath) to the 8-bit address/offset *address_8bits(7:0)* which is supplied by the instruction itself and truncated from 16 bits down to 8 bits.

$$PCplus1_Addr = (100)_{\text{decimal}}$$

$$address_8bits = (-10)_{\text{decimal}}$$

$$\begin{aligned} Branch_Target &= PCplus1_Addr + address_8bits \\ &= (100)_{\text{decimal}} + (-10)_{\text{decimal}} \\ &= (90)_{\text{decimal}} \end{aligned}$$

□ **During part 2 of the waveform (clock cycle 2):**

- Preload the register file with the register operand for *rt*:

$$rt = \$R6, \quad [\$R6] = (15)_{\text{decimal}}$$

□ **During part 3 of the waveform (clock cycle 3):**

- The ALU calculates the value for the signal *ALU_Zero* by performing the subtraction:

Since

$$(15)_{\text{decimal}} = (15)_{\text{decimal}} \Rightarrow (15)_{\text{decimal}} - (15)_{\text{decimal}} = 0$$

i.e.

$$[\$R5] = [\$R6]$$

Then

$$ALU_Zero = 1$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals *a_in* and *b_in* are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals *a_in* and *b_in* to the ALU and the output signal *alu_zero* stabilizing to the correct value = 1.5 ns.
- ◆ Delay between application of input signals *PCplus1_Addr* and *address_8bits* to the 8-bit adder (ADD8) and the output signal *Branch_Target* stabilizing to the correct value = 0.4 ns.

□ **Conclusions:**

- The execution of either branch instruction (BEQ or BNE) takes only 1 clock cycle. This is clock cycle no. 1 in Figure B.36.
- Actually, total elapsed execution time is 3 clock cycles when including clock cycles no. 1 and 2, which are for the data pre-loading phase.

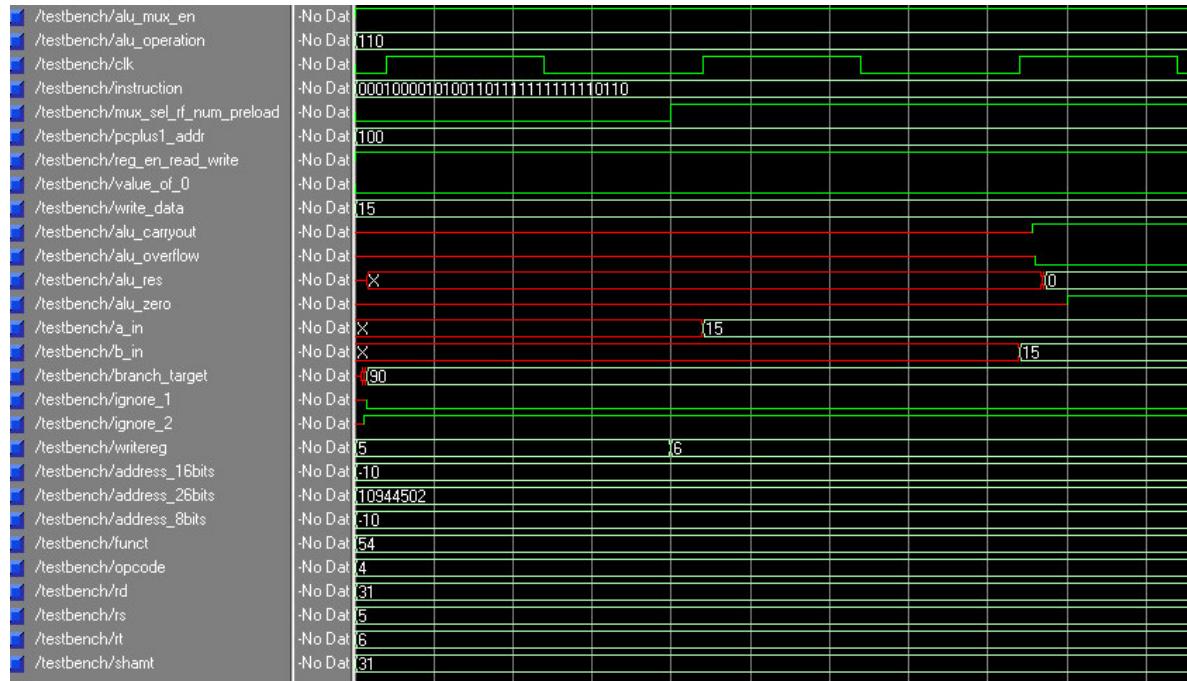


Figure B.36 Results of simulating the synthesized datapath section for the branch instructions BEQ and BNE, using ModelSim, with ALU Operation = SUB for the condition $[rs] = [rt]$.

❖ **Condition 2 (Figure B.37): When $[rs] \neq [rt]$**

□ **During part 1 of the waveform (clock cycle 1):**

- Preload the register file with the register operand for rs :

$$rs = \$R5, \quad [\$R5] = (15)_{\text{decimal}}$$

- Test Instruction:

$$\begin{array}{ccc} \text{BEQ} & \$R5, & \$R6, \quad -10 \\ \hline & rs & rt \quad \text{offset} \end{array}$$

Assembly Language 32-bit Instruction Representation (discussed in Ch.5):

$$\begin{array}{cccc} 000100 & 00101 & 00110 & 111111111110110 \\ \hline op=4 & rs=\$R5 & rt=\$R6 & \text{offset}=-10 \end{array}$$

- The adder computes the branch target address $Branch_Target(7:0)$ by adding the value of $PC + 1$ (supplied by the instruction fetch unit/stage/datapath) to the 8-bit address/offset $address_8bits(7:0)$ which is supplied by the instruction itself and truncated from 16 bits down to 8 bits.

$$PCplus1_Addr = (100)_{\text{decimal}}$$

$$address_8bits = (-10)_{\text{decimal}}$$

$$Branch_Target = PCplus1_Addr + address_8bits$$

$$\begin{aligned} &= (100)_{\text{decimal}} + (-10)_{\text{decimal}} \\ &= (90)_{\text{decimal}} \end{aligned}$$

□ **During part 2 of the waveform (clock cycle 2):**

- Preload the register file with the register operand for *rt*:

$$rt = \$R6, \quad [\$R6] = (16)_{\text{decimal}}$$

□ **During part 3 of the waveform (clock cycle 3):**

- The ALU calculates the value for the signal *ALU_Zero* by performing the subtraction:

$$\begin{aligned} &\text{Since} \\ &\quad (15)_{\text{decimal}} < (16)_{\text{decimal}} \Rightarrow (15)_{\text{decimal}} - (16)_{\text{decimal}} = -1 \\ &\quad \text{i.e.} \\ &\quad [\$R5] \neq [\$R6] \\ &\text{Then} \\ &\quad ALU_Zero = 0 \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals *a_in* and *b_in* are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals *a_in* and *b_in* to the ALU and the output signal *alu_zero* stabilizing to the correct value = 1.1 ns.
- ◆ Delay between application of input signals *PCplus1_Addr* and *address_8bits* to the 8-bit adder (ADD8) and the output signal *Branch_Target* stabilizing to the correct value = 0.4 ns.

□ **Conclusions:**

- The execution of either branch instruction (BEQ or BNE) takes only 1 clock cycle. This is clock cycle no. 3 in Figure B.37.
- Actually, total elapsed execution time is 3 clock cycles when including clock cycles no. 1 and 2, which are for the data pre-loading phase.

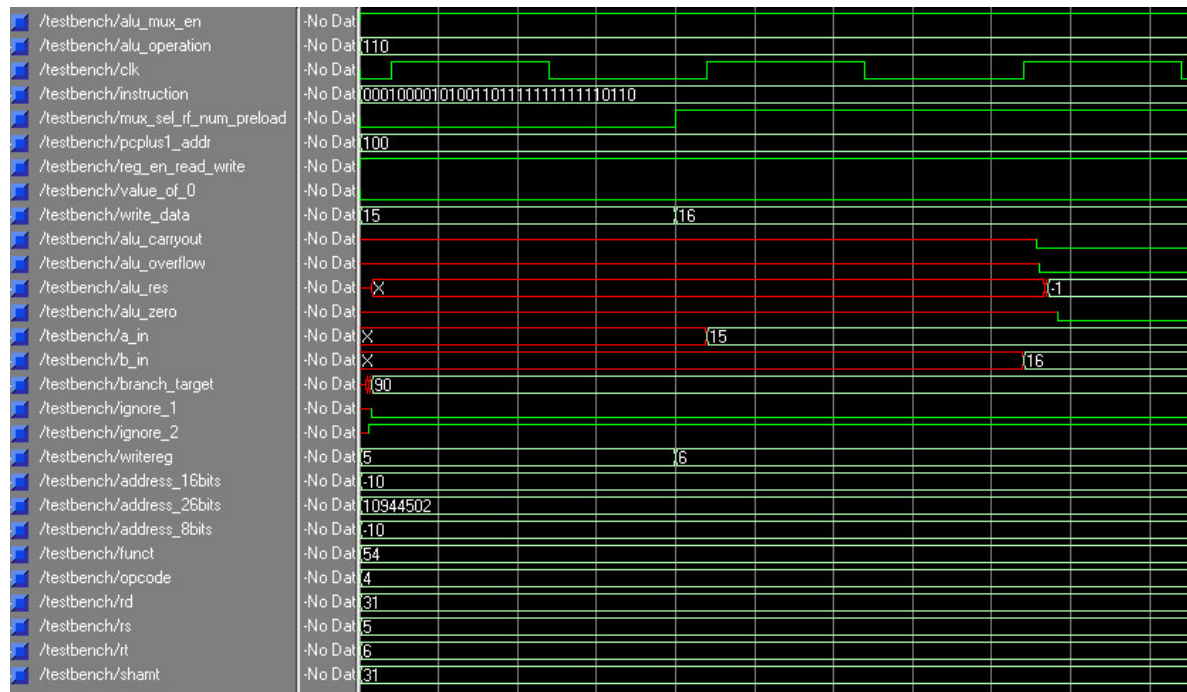


Figure B.37 Results of simulating the synthesized datapath section for the branch instructions BEQ and BNE, using ModelSim, with ALU Operation = SUB for the condition $[rs] \neq [rt]$.

As a final conclusion, it is clear that the resulting synthesized hardware functions according to the specified behaviour of datapath section for the BEQ and BNE instructions. This concludes the design cycle for this datapath section.

B.3.5 The Datapath Section for Jump Instruction

The branch instructions BEQ and BNE allow for only 16 bits for the offset supplied within the instruction, which limits the branch transfer range to only $+2^{15}$ or -2^{15} instruction words relative to the current one specified by PC+1 [47]. This is a limitation if the control transfer needed is larger than that range. This is where the Jump instruction comes in allowing unconditional jumps farther than what is allowed by the branch instructions [47].

The MIPS implementation for the jump instruction is 6 bits for the opcode and the remaining 26 bits for the address/offset, the latter is then concatenated with the upper 6 bits from the PC to generate the final target address. This requires a separate datapath section to implement [47].

However, since my implementation is a word addressable 8-bit address space only, it is not necessary to synthesize a separate datapath for performing this bit concatenation operation for implementing the jump instruction. Still, the hardware implementation for the jump instruction ties in very closely with the instruction fetch stage and the branching datapath section, and will be implemented in the next section when combining all the datapath sections together.

B.4 Combining The Datapath Sections Together

➤ RTL Description

The complete datapath is constructed by combining all the previously discussed datapath sections (section B.3) together. This is discussed in more detail in Chapter 5 in [47]. Figure B.38 below shows this complete datapath.

However, in my implementation of this complete datapath, there is a need for the addition of another unit called the DCM (Digital Clock Manager), described in Appendix A, to divide the clock and feed the resulting different clock frequencies to different sections of the datapath. The reasoning behind this will be elaborated shortly when simulating the branch instructions. The next section (B.5) will cover this finalized complete datapath with DCM.

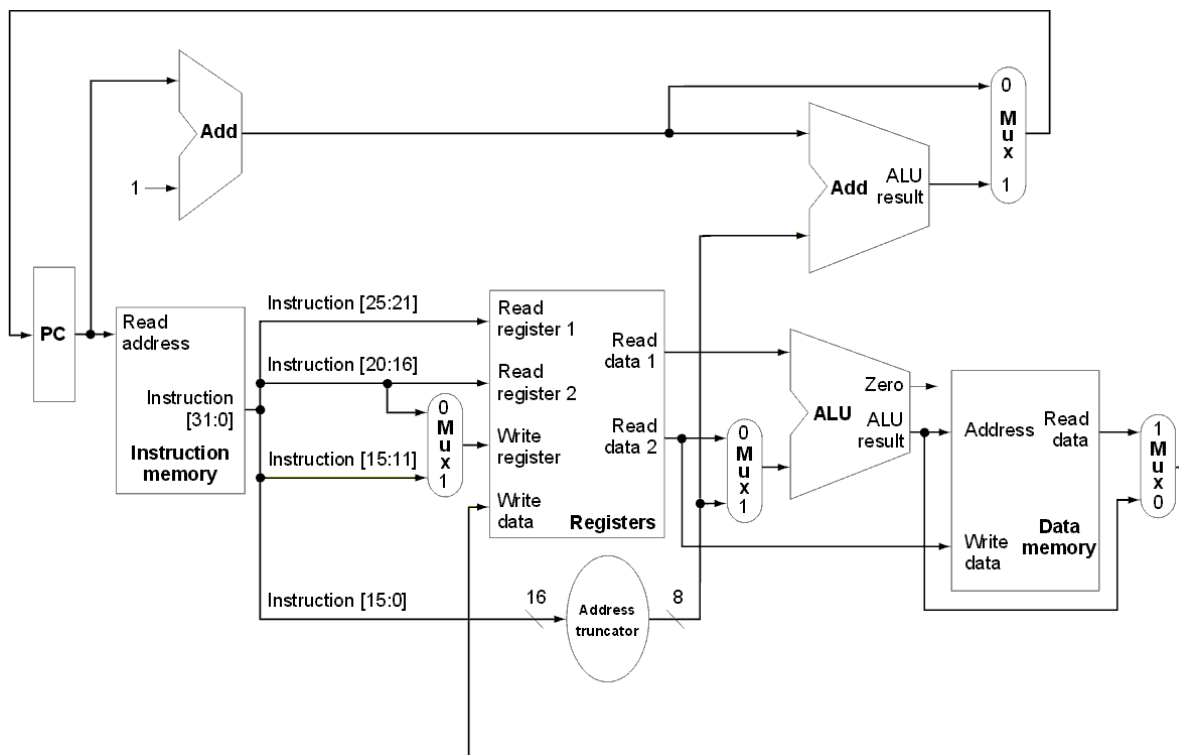


Figure B.38 The complete datapath for the MIPS architecture combines the different elements (datapath sections) required by the different instruction classes (adapted from [47]).

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the complete datapath shown in figure B.38. Figure B.39 shows the final schematic diagram.

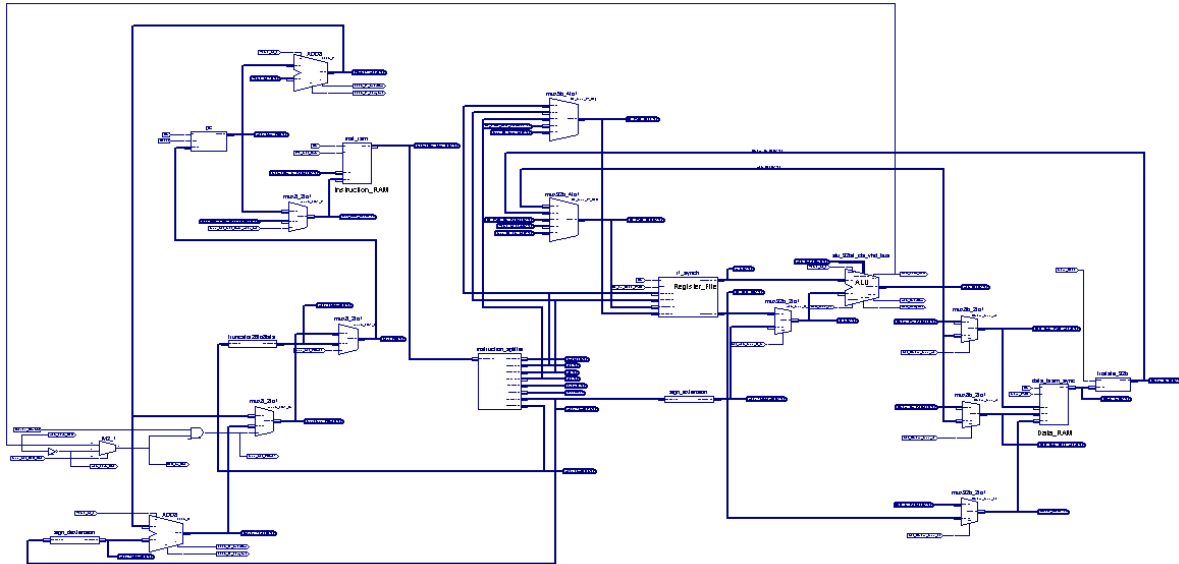


Figure B.39 Schematic diagram design entry in Schematic Editor for the complete datapath.

After synthesis of the schematic diagram in figure B.39 using XST, the following VHDL code was generated:

- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\add8.sch - Sat Sep 09 17:02:20 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 ADD8_MXILINX_complete_datapath_new_beginning IS
  PORT ( A      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        B      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        CI      : IN    STD_LOGIC;
        CO      : OUT   STD_LOGIC;
        OFL     : OUT   STD_LOGIC;
        S      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0));
```

```
end ADD8_MXILINX_complete_datapath_new_beginning;
```

```
ARCHITECTURE SCHEMATIC OF ADD8_MXILINX_complete_datapath_new_beginning IS
  SIGNAL C0      : STD_LOGIC;
  SIGNAL C1      : STD_LOGIC;
  SIGNAL C2      : STD_LOGIC;
  SIGNAL C3      : STD_LOGIC;
  SIGNAL C4      : STD_LOGIC;
  SIGNAL C5      : STD_LOGIC;
  SIGNAL C6      : STD_LOGIC;
  SIGNAL C60     : STD_LOGIC;
  SIGNAL CO_DUMMY : STD_LOGIC;
  SIGNAL I0      : STD_LOGIC;
  SIGNAL I1      : STD_LOGIC;
  SIGNAL I2      : STD_LOGIC;
  SIGNAL I3      : STD_LOGIC;
  SIGNAL I4      : STD_LOGIC;
  SIGNAL I5      : STD_LOGIC;
  SIGNAL I6      : STD_LOGIC;
  SIGNAL I7      : STD_LOGIC;
  SIGNAL dummy   : STD_LOGIC;
```

```

ATTRIBUTE BOX_TYPE : STRING;
ATTRIBUTE RLOC : STRING ;
ATTRIBUTE RLOC OF I_36_16 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_17 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_23 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_22 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_18 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_19 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_20 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_21 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_64 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_107 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_110 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_63 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_58 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_111 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_55 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_62 : LABEL IS "X0Y1";

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 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_D
  PORT ( CI      :      IN      STD_LOGIC;
         DI      :      IN      STD_LOGIC;
         S       :      IN      STD_LOGIC;
         LO      :      OUT     STD_LOGIC;
         O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY_D : 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 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";
COMPONENT XORCY
  PORT ( CI      :      IN      STD_LOGIC;
         LI      :      IN      STD_LOGIC;
         O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XORCY : COMPONENT IS "BLACK_BOX";
BEGIN
  CO <= CO_DUMMY;

  I_36_16 : FMAP
    PORT MAP (I1=>A(0), I2=>B(0), I3=>dummy, I4=>dummy, O=>I0);

```



```

I_36_17 : FMAP
  PORT MAP (I1=>A(1), I2=>B(1), I3=>dummy, I4=>dummy, O=>I1);
I_36_23 : FMAP
  PORT MAP (I1=>A(7), I2=>B(7), I3=>dummy, I4=>dummy, O=>I7);
I_36_22 : FMAP
  PORT MAP (I1=>A(6), I2=>B(6), I3=>dummy, I4=>dummy, O=>I6);
I_36_18 : FMAP
  PORT MAP (I1=>A(2), I2=>B(2), I3=>dummy, I4=>dummy, O=>I2);
I_36_19 : FMAP
  PORT MAP (I1=>A(3), I2=>B(3), I3=>dummy, I4=>dummy, O=>I3);
I_36_20 : FMAP
  PORT MAP (I1=>A(4), I2=>B(4), I3=>dummy, I4=>dummy, O=>I4);
I_36_21 : FMAP
  PORT MAP (I1=>A(5), I2=>B(5), I3=>dummy, I4=>dummy, O=>I5);
I_36_64 : MUXCY
  PORT MAP (CI=>C6, DI=>A(7), S=>I7, O=>CO_DUMMY);
I_36_107 : MUXCY_D
  PORT MAP (CI=>C5, DI=>A(6), S=>I6, LO=>C6, O=>C6O);
I_36_110 : MUXCY_L
  PORT MAP (CI=>C4, DI=>A(5), S=>I5, LO=>C5);
I_36_63 : MUXCY_L
  PORT MAP (CI=>C3, DI=>A(4), S=>I4, LO=>C4);
I_36_58 : MUXCY_L
  PORT MAP (CI=>C2, DI=>A(3), S=>I3, LO=>C3);
I_36_111 : MUXCY_L
  PORT MAP (CI=>CI, DI=>A(0), S=>I0, LO=>C0);
I_36_55 : MUXCY_L
  PORT MAP (CI=>C0, DI=>A(1), S=>I1, LO=>C1);
I_36_62 : MUXCY_L
  PORT MAP (CI=>C1, DI=>A(2), S=>I2, LO=>C2);
I_36_239 : XOR2
  PORT MAP (I0=>C6O, I1=>CO_DUMMY, O=>OFL);
I_36_230 : XOR2
  PORT MAP (I0=>A(2), I1=>B(2), O=>I2);
I_36_229 : XOR2
  PORT MAP (I0=>A(1), I1=>B(1), O=>I1);
I_36_228 : XOR2
  PORT MAP (I0=>A(0), I1=>B(0), O=>I0);
I_36_224 : XOR2
  PORT MAP (I0=>A(4), I1=>B(4), O=>I4);
I_36_223 : XOR2
  PORT MAP (I0=>A(5), I1=>B(5), O=>I5);
I_36_222 : XOR2
  PORT MAP (I0=>A(6), I1=>B(6), O=>I6);
I_36_225 : XOR2
  PORT MAP (I0=>A(3), I1=>B(3), O=>I3);
I_36_221 : XOR2
  PORT MAP (I0=>A(7), I1=>B(7), O=>I7);
I_36_80 : XORCY
  PORT MAP (CI=>C6, LI=>I7, O=>S(7));
I_36_73 : XORCY
  PORT MAP (CI=>CI, LI=>I0, O=>S(0));
I_36_74 : XORCY
  PORT MAP (CI=>C0, LI=>I1, O=>S(1));
I_36_76 : XORCY
  PORT MAP (CI=>C1, LI=>I2, O=>S(2));
I_36_75 : XORCY
  PORT MAP (CI=>C2, LI=>I3, O=>S(3));
I_36_78 : XORCY
  PORT MAP (CI=>C3, LI=>I4, O=>S(4));
I_36_77 : XORCY
  PORT MAP (CI=>C4, LI=>I5, O=>S(5));
I_36_81 : XORCY
  PORT MAP (CI=>C5, LI=>I6, O=>S(6));

END SCHEMATIC;

-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1.sch - Sat Sep
09 17:02:20 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 M2_1_MXILINX_complete_datapath_new_beginning IS
    PORT ( D0      :      IN      STD_LOGIC;
           D1      :      IN      STD_LOGIC;
           S0      :      IN      STD_LOGIC;
           O        :      OUT     STD_LOGIC);

end M2_1_MXILINX_complete_datapath_new_beginning;

ARCHITECTURE SCHEMATIC OF M2_1_MXILINX_complete_datapath_new_beginning 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;

-- Vhdl model created from schematic complete_datapath_new_beginning.sch - Sat Sep 09
17:02:38 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 complete_datapath_new_beginning IS
    PORT ( ALU_Mux_En      : IN      STD_LOGIC;
           ALU_Operation   : IN      STD_LOGIC_VECTOR (2 DOWNTO 0);
           Branch_Control  : IN      STD_LOGIC;
           Clk             : IN      STD_LOGIC;
           DMem_RA_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
           DMem_WA_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
           Data_in_Preload : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
           Inst_RAM_Write_Addr_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
           Instruction_in_Preload : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
           Mem_Read        : IN      STD_LOGIC);

```

```

Mem_Write           : IN    STD_LOGIC;
Mux_RF_Num_Sel      : IN    STD_LOGIC_VECTOR (1 DOWNTO 0);
Mux_RF_din_Sel      : IN    STD_LOGIC_VECTOR (1 DOWNTO 0);
Mux_Sel_BEQ_BNE     : IN    STD_LOGIC;
Mux_Sel_Inst_RAM_Addr_Src : IN    STD_LOGIC;
Mux_Sel_PCSrc2      : IN    STD_LOGIC;
RF_En_Read_Write    : IN    STD_LOGIC;
RF_Write_Din_Preload : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Num_Preload : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
Reset               : IN    STD_LOGIC;
Sel_ALU_Mux_B_in     : IN    STD_LOGIC;
Sel_DMem_Mux_di      : IN    STD_LOGIC;
Sel_DMem_Mux_ra      : IN    STD_LOGIC;
Sel_DMem_Mux_wa      : IN    STD_LOGIC;
Value_of_0          : IN    STD_LOGIC;
Value_of_1          : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
Value_of_Zero       : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
WE_Inst_RAM         : IN    STD_LOGIC;
ALU_Carryout        : OUT    STD_LOGIC;
ALU_Overflow        : OUT    STD_LOGIC;
ALU_Res             : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
ALU_Zero_BEQ        : OUT    STD_LOGIC;
ALU_Zero_BNE        : OUT    STD_LOGIC;
A_in                : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
Addr_16bits         : OUT    STD_LOGIC_VECTOR (15 DOWNTO 0);
Addr_16to32bits     : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
Addr_16to8bits      : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Addr_26bits         : OUT    STD_LOGIC_VECTOR (25 DOWNTO 0);
Addr_26to8bits      : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Addr_from_PC        : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
BEQ_or_BNE          : OUT    STD_LOGIC;
B_in                : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
Branch_Target       : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
DMem_Din            : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
DMem_to_RF          : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
DRAM_out            : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
Data_RAM_Read_Addr  : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Data_RAM_Write_Addr : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Inst_RAM_Addr       : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Instruction_from_IF : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
Mux2Mux_PCSrc       : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
Mux_Sel_PCSrc1      : OUT    STD_LOGIC;
Next_PC             : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
PCplus1_Addr        : OUT    STD_LOGIC_VECTOR (7 DOWNTO 0);
RF_Data_B           : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Data       : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Reg        : OUT    STD_LOGIC_VECTOR (4 DOWNTO 0);
add8_i1_carryout    : OUT    STD_LOGIC;
add8_i1_overflow    : OUT    STD_LOGIC;
add8_i2_carryout    : OUT    STD_LOGIC;
add8_i2_overflow    : OUT    STD_LOGIC;
funct               : OUT    STD_LOGIC_VECTOR (5 DOWNTO 0);
opcode              : OUT    STD_LOGIC_VECTOR (5 DOWNTO 0);
rd                  : OUT    STD_LOGIC_VECTOR (4 DOWNTO 0);
rs                  : OUT    STD_LOGIC_VECTOR (4 DOWNTO 0);
rt                  : OUT    STD_LOGIC_VECTOR (4 DOWNTO 0);
shamt               : OUT    STD_LOGIC_VECTOR (4 DOWNTO 0);

end complete_datapath_new_beginning;

ARCHITECTURE SCHEMATIC OF complete_datapath_new_beginning IS
    SIGNAL ALU_Res_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL ALU_Zero_BEQ_DUMMY : STD_LOGIC;
    SIGNAL ALU_Zero_BNE_DUMMY : STD_LOGIC;
    SIGNAL A_in_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL Addr_16bits_DUMMY : STD_LOGIC_VECTOR (15 DOWNTO 0);
    SIGNAL Addr_16to32bits_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL Addr_16to8bits_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL Addr_26bits_DUMMY : STD_LOGIC_VECTOR (25 DOWNTO 0);
    SIGNAL Addr_26to8bits_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL Addr_from_PC_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL BEQ_or_BNE_DUMMY : STD_LOGIC;
    SIGNAL B_in_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);

```

```

SIGNAL Branch_Target_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL DMem_Din_DUMMY          :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL DMem_to_RF_DUMMY        :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL DRAM_out_DUMMY          :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL Data_RAM_Read_Addr_DUMMY :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Data_RAM_Write_Addr_DUMMY :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Inst_RAM_Addr_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Instruction_from_IF_DUMMY :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL Mux2Mux_PCSrc_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Mux_Sel_PCSrc1_DUMMY     :      STD_LOGIC;
SIGNAL Next_PC_DUMMY           :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL PCplus1_Addr_DUMMY       :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL RF_Data_B_DUMMY         :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL RF_Write_Data_DUMMY     :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL RF_Write_Reg_DUMMY      :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rd_DUMMY                :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rs_DUMMY                :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rt_DUMMY                :      STD_LOGIC_VECTOR (4 DOWNTO 0);

ATTRIBUTE BOX_TYPE : STRING;
ATTRIBUTE U_SET : STRING ;
ATTRIBUTE U_SET OF add8_i2 : LABEL IS "add8_i2_2";
ATTRIBUTE U_SET OF add8_i1 : LABEL IS "add8_i1_0";
ATTRIBUTE U_SET OF m2_1_i1 : LABEL IS "m2_1_i1_1";

COMPONENT ADD8_MXILINK_complete_datapath_new_beginning
  PORT ( A      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        B      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        CI      :      IN      STD_LOGIC;
        CO      :      OUT     STD_LOGIC;
        OFL      :      OUT     STD_LOGIC;
        S      :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT alu_32bit_cla_vhd_bus
  PORT ( A      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        B      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        less_zero :      IN      STD_LOGIC;
        carryout  :      OUT     STD_LOGIC;
        overflow  :      OUT     STD_LOGIC;
        Result    :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        zero      :      OUT     STD_LOGIC;
        mux_enable :      IN      STD_LOGIC;
        aluoperation :      IN      STD_LOGIC_VECTOR (2 DOWNTO 0));
END COMPONENT;

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 data_bram_sync
  PORT ( clk      :      IN      STD_LOGIC;
        we      :      IN      STD_LOGIC;
        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 COMPONENT;

COMPONENT inst_ram
  PORT ( clk      :      IN      STD_LOGIC;
        we      :      IN      STD_LOGIC;
        a      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        di      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        do      :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT instruction_splitter
  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 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_MXILINK_complete_datapath_new_beginning
  PORT ( D0 : IN STD_LOGIC;
        D1 : IN STD_LOGIC;
        S0 : IN STD_LOGIC;
        O : OUT STD_LOGIC);
END COMPONENT;

COMPONENT mux32b_2to1
  PORT ( din0 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        din1 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        sel : IN STD_LOGIC);
END COMPONENT;

COMPONENT mux32b_4to1
  PORT ( din0 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        din1 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        din2 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        din3 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        sel : IN STD_LOGIC_VECTOR (1 DOWNTO 0));
END COMPONENT;

COMPONENT mux5b_4to1
  PORT ( din0 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        din1 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        din2 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        din3 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
        sel : IN STD_LOGIC_VECTOR (1 DOWNTO 0));
END COMPONENT;

COMPONENT mux8_2to1
  PORT ( sel : IN STD_LOGIC;
        din0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        din1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT mux8b_2to1
  PORT ( din0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        din1 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        sel : IN STD_LOGIC);
END COMPONENT;

COMPONENT pc
  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 COMPONENT;

COMPONENT rf_synch
  PORT ( clk : IN STD_LOGIC;
        enable : IN STD_LOGIC;
        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);
        read_data_1     :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        read_data_2     :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT sign_dextension
    PORT ( addr_in      :      IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
          addr_out      :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT sign_extension
    PORT ( addr_in      :      IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
          addr_out      :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT tristate_32b
    PORT ( enable       :      IN      STD_LOGIC;
          din           :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          dout          :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT truncator26to8bits
    PORT ( addr_in      :      IN      STD_LOGIC_VECTOR (25 DOWNTO 0);
          addr_out      :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

BEGIN
    ALU_Res <= ALU_Res_DUMMY;
    ALU_Zero_BEQ <= ALU_Zero_BEQ_DUMMY;
    ALU_Zero_BNE <= ALU_Zero_BNE_DUMMY;
    A_in <= A_in_DUMMY;
    Addr_16bits <= Addr_16bits_DUMMY;
    Addr_16to32bits <= Addr_16to32bits_DUMMY;
    Addr_16to8bits <= Addr_16to8bits_DUMMY;
    Addr_26bits <= Addr_26bits_DUMMY;
    Addr_26to8bits <= Addr_26to8bits_DUMMY;
    Addr_from_PC <= Addr_from_PC_DUMMY;
    BEQ_or_BNE <= BEQ_or_BNE_DUMMY;
    B_in <= B_in_DUMMY;
    Branch_Target <= Branch_Target_DUMMY;
    DMem_Din <= DMem_Din_DUMMY;
    DMem_to_RF <= DMem_to_RF_DUMMY;
    DRAM_out <= DRAM_out_DUMMY;
    Data_RAM_Read_Addr <= Data_RAM_Read_Addr_DUMMY;
    Data_RAM_Write_Addr <= Data_RAM_Write_Addr_DUMMY;
    Inst_RAM_Addr <= Inst_RAM_Addr_DUMMY;
    Instruction_from_IF <= Instruction_from_IF_DUMMY;
    Mux2Mux_PCSrc <= Mux2Mux_PCSrc_DUMMY;
    Mux_Sel_PCSrc1 <= Mux_Sel_PCSrc1_DUMMY;
    Next_PC <= Next_PC_DUMMY;
    PCplus1_Addr <= PCplus1_Addr_DUMMY;
    RF_Data_B <= RF_Data_B_DUMMY;
    RF_Write_Data <= RF_Write_Data_DUMMY;
    RF_Write_Reg <= RF_Write_Reg_DUMMY;
    rd <= rd_DUMMY;
    rs <= rs_DUMMY;
    rt <= rt_DUMMY;

    add8_i2 : ADD8_MXILINX_complete_datapath_new_beginning
        PORT MAP (A(7)=>PCplus1_Addr_DUMMY(7), A(6)=>PCplus1_Addr_DUMMY(6),
                  A(5)=>PCplus1_Addr_DUMMY(5), A(4)=>PCplus1_Addr_DUMMY(4),
                  A(3)=>PCplus1_Addr_DUMMY(3), A(2)=>PCplus1_Addr_DUMMY(2),
                  A(1)=>PCplus1_Addr_DUMMY(1), A(0)=>PCplus1_Addr_DUMMY(0),
                  B(7)=>Addr_16to8bits_DUMMY(7), B(6)=>Addr_16to8bits_DUMMY(6),
                  B(5)=>Addr_16to8bits_DUMMY(5), B(4)=>Addr_16to8bits_DUMMY(4),
                  B(3)=>Addr_16to8bits_DUMMY(3), B(2)=>Addr_16to8bits_DUMMY(2),
                  B(1)=>Addr_16to8bits_DUMMY(1), B(0)=>Addr_16to8bits_DUMMY(0),
                  CI=>Value_of_0, CO=>add8_i2_carryout, OFL=>add8_i2_overflow,
                  S(7)=>Branch_Target_DUMMY(7), S(6)=>Branch_Target_DUMMY(6),
                  S(5)=>Branch_Target_DUMMY(5), S(4)=>Branch_Target_DUMMY(4),
                  S(3)=>Branch_Target_DUMMY(3), S(2)=>Branch_Target_DUMMY(2),
                  S(1)=>Branch_Target_DUMMY(1), S(0)=>Branch_Target_DUMMY(0));

```

```

add8_i1 : ADD8_MXILINK_complete_datapath_new_beginning
PORT MAP (A(7)=>Addr_from_PC_DUMMY(7), A(6)=>Addr_from_PC_DUMMY(6),
A(5)=>Addr_from_PC_DUMMY(5), A(4)=>Addr_from_PC_DUMMY(4),
A(3)=>Addr_from_PC_DUMMY(3), A(2)=>Addr_from_PC_DUMMY(2),
A(1)=>Addr_from_PC_DUMMY(1), A(0)=>Addr_from_PC_DUMMY(0),
B(7)=>Value_of_1(7), B(6)=>Value_of_1(6), B(5)=>Value_of_1(5),
B(4)=>Value_of_1(4), B(3)=>Value_of_1(3), B(2)=>Value_of_1(2),
B(1)=>Value_of_1(1), B(0)=>Value_of_1(0), CI=>Value_of_0,
CO=>add8_i1_carryout, OFL=>add8_i1_overflow, S(7)=>PCplus1_Addr_DUMMY(7),
S(6)=>PCplus1_Addr_DUMMY(6), S(5)=>PCplus1_Addr_DUMMY(5),
S(4)=>PCplus1_Addr_DUMMY(4), S(3)=>PCplus1_Addr_DUMMY(3),
S(2)=>PCplus1_Addr_DUMMY(2), S(1)=>PCplus1_Addr_DUMMY(1),
S(0)=>PCplus1_Addr_DUMMY(0));

alu : alu_32bit_cla_vhd_bus
PORT MAP (A(31)=>A_in_DUMMY(31), A(30)=>A_in_DUMMY(30),
A(29)=>A_in_DUMMY(29), A(28)=>A_in_DUMMY(28), A(27)=>A_in_DUMMY(27),
A(26)=>A_in_DUMMY(26), A(25)=>A_in_DUMMY(25), A(24)=>A_in_DUMMY(24),
A(23)=>A_in_DUMMY(23), A(22)=>A_in_DUMMY(22), A(21)=>A_in_DUMMY(21),
A(20)=>A_in_DUMMY(20), A(19)=>A_in_DUMMY(19), A(18)=>A_in_DUMMY(18),
A(17)=>A_in_DUMMY(17), A(16)=>A_in_DUMMY(16), A(15)=>A_in_DUMMY(15),
A(14)=>A_in_DUMMY(14), A(13)=>A_in_DUMMY(13), A(12)=>A_in_DUMMY(12),
A(11)=>A_in_DUMMY(11), A(10)=>A_in_DUMMY(10), A(9)=>A_in_DUMMY(9),
A(8)=>A_in_DUMMY(8), A(7)=>A_in_DUMMY(7), A(6)=>A_in_DUMMY(6),
A(5)=>A_in_DUMMY(5), A(4)=>A_in_DUMMY(4), A(3)=>A_in_DUMMY(3),
A(2)=>A_in_DUMMY(2), A(1)=>A_in_DUMMY(1), A(0)=>A_in_DUMMY(0),
B(31)=>B_in_DUMMY(31), B(30)=>B_in_DUMMY(30), B(29)=>B_in_DUMMY(29),
B(28)=>B_in_DUMMY(28), B(27)=>B_in_DUMMY(27), B(26)=>B_in_DUMMY(26),
B(25)=>B_in_DUMMY(25), B(24)=>B_in_DUMMY(24), B(23)=>B_in_DUMMY(23),
B(22)=>B_in_DUMMY(22), B(21)=>B_in_DUMMY(21), B(20)=>B_in_DUMMY(20),
B(19)=>B_in_DUMMY(19), B(18)=>B_in_DUMMY(18), B(17)=>B_in_DUMMY(17),
B(16)=>B_in_DUMMY(16), B(15)=>B_in_DUMMY(15), B(14)=>B_in_DUMMY(14),
B(13)=>B_in_DUMMY(13), B(12)=>B_in_DUMMY(12), B(11)=>B_in_DUMMY(11),
B(10)=>B_in_DUMMY(10), B(9)=>B_in_DUMMY(9), B(8)=>B_in_DUMMY(8),
B(7)=>B_in_DUMMY(7), B(6)=>B_in_DUMMY(6), B(5)=>B_in_DUMMY(5),
B(4)=>B_in_DUMMY(4), B(3)=>B_in_DUMMY(3), B(2)=>B_in_DUMMY(2),
B(1)=>B_in_DUMMY(1), B(0)=>B_in_DUMMY(0), less_zero=>Value_of_0,
carryout=>ALU_Carryout, overflow=>ALU_Overflow,
Result(31)=>ALU_Res_DUMMY(31), Result(30)=>ALU_Res_DUMMY(30),
Result(29)=>ALU_Res_DUMMY(29), Result(28)=>ALU_Res_DUMMY(28),
Result(27)=>ALU_Res_DUMMY(27), Result(26)=>ALU_Res_DUMMY(26),
Result(25)=>ALU_Res_DUMMY(25), Result(24)=>ALU_Res_DUMMY(24),
Result(23)=>ALU_Res_DUMMY(23), Result(22)=>ALU_Res_DUMMY(22),
Result(21)=>ALU_Res_DUMMY(21), Result(20)=>ALU_Res_DUMMY(20),
Result(19)=>ALU_Res_DUMMY(19), Result(18)=>ALU_Res_DUMMY(18),
Result(17)=>ALU_Res_DUMMY(17), Result(16)=>ALU_Res_DUMMY(16),
Result(15)=>ALU_Res_DUMMY(15), Result(14)=>ALU_Res_DUMMY(14),
Result(13)=>ALU_Res_DUMMY(13), Result(12)=>ALU_Res_DUMMY(12),
Result(11)=>ALU_Res_DUMMY(11), Result(10)=>ALU_Res_DUMMY(10),
Result(9)=>ALU_Res_DUMMY(9), Result(8)=>ALU_Res_DUMMY(8),
Result(7)=>ALU_Res_DUMMY(7), Result(6)=>ALU_Res_DUMMY(6),
Result(5)=>ALU_Res_DUMMY(5), Result(4)=>ALU_Res_DUMMY(4),
Result(3)=>ALU_Res_DUMMY(3), Result(2)=>ALU_Res_DUMMY(2),
Result(1)=>ALU_Res_DUMMY(1), Result(0)=>ALU_Res_DUMMY(0),
zero=>ALU_Zero_BEQ_DUMMY, mux_enable=>ALU_Mux_En,
aluoperation(2)=>ALU_Operation(2), aluoperation(1)=>ALU_Operation(1),
aluoperation(0)=>ALU_Operation(0));

and2_i1 : AND2
PORT MAP (I0=>BEQ_or_BNE_DUMMY, I1=>Branch_Control,
O=>Mux_Sel_PCSrc1_DUMMY);

Data_RAM : data_bram_sync
PORT MAP (clk=>Clk, we=>Mem_Write, wa(7)=>Data_RAM_Write_Addr_DUMMY(7),
wa(6)=>Data_RAM_Write_Addr_DUMMY(6), wa(5)=>Data_RAM_Write_Addr_DUMMY(5),
wa(4)=>Data_RAM_Write_Addr_DUMMY(4), wa(3)=>Data_RAM_Write_Addr_DUMMY(3),
wa(2)=>Data_RAM_Write_Addr_DUMMY(2), wa(1)=>Data_RAM_Write_Addr_DUMMY(1),
wa(0)=>Data_RAM_Write_Addr_DUMMY(0), ra(7)=>Data_RAM_Read_Addr_DUMMY(7),
ra(6)=>Data_RAM_Read_Addr_DUMMY(6), ra(5)=>Data_RAM_Read_Addr_DUMMY(5),
ra(4)=>Data_RAM_Read_Addr_DUMMY(4), ra(3)=>Data_RAM_Read_Addr_DUMMY(3),
ra(2)=>Data_RAM_Read_Addr_DUMMY(2), ra(1)=>Data_RAM_Read_Addr_DUMMY(1),
ra(0)=>Data_RAM_Read_Addr_DUMMY(0), di(31)=>DMem_Din_DUMMY(31),

```

```

di(30)=>DMem_Din_DUMMY(30), di(29)=>DMem_Din_DUMMY(29),
di(28)=>DMem_Din_DUMMY(28), di(27)=>DMem_Din_DUMMY(27),
di(26)=>DMem_Din_DUMMY(26), di(25)=>DMem_Din_DUMMY(25),
di(24)=>DMem_Din_DUMMY(24), di(23)=>DMem_Din_DUMMY(23),
di(22)=>DMem_Din_DUMMY(22), di(21)=>DMem_Din_DUMMY(21),
di(20)=>DMem_Din_DUMMY(20), di(19)=>DMem_Din_DUMMY(19),
di(18)=>DMem_Din_DUMMY(18), di(17)=>DMem_Din_DUMMY(17),
di(16)=>DMem_Din_DUMMY(16), di(15)=>DMem_Din_DUMMY(15),
di(14)=>DMem_Din_DUMMY(14), di(13)=>DMem_Din_DUMMY(13),
di(12)=>DMem_Din_DUMMY(12), di(11)=>DMem_Din_DUMMY(11),
di(10)=>DMem_Din_DUMMY(10), di(9)=>DMem_Din_DUMMY(9),
di(8)=>DMem_Din_DUMMY(8), di(7)=>DMem_Din_DUMMY(7),
di(6)=>DMem_Din_DUMMY(6), di(5)=>DMem_Din_DUMMY(5),
di(4)=>DMem_Din_DUMMY(4), di(3)=>DMem_Din_DUMMY(3),
di(2)=>DMem_Din_DUMMY(2), di(1)=>DMem_Din_DUMMY(1),
di(0)=>DMem_Din_DUMMY(0), do(31)=>DRAM_out_DUMMY(31),
do(30)=>DRAM_out_DUMMY(30), do(29)=>DRAM_out_DUMMY(29),
do(28)=>DRAM_out_DUMMY(28), do(27)=>DRAM_out_DUMMY(27),
do(26)=>DRAM_out_DUMMY(26), do(25)=>DRAM_out_DUMMY(25),
do(24)=>DRAM_out_DUMMY(24), do(23)=>DRAM_out_DUMMY(23),
do(22)=>DRAM_out_DUMMY(22), do(21)=>DRAM_out_DUMMY(21),
do(20)=>DRAM_out_DUMMY(20), do(19)=>DRAM_out_DUMMY(19),
do(18)=>DRAM_out_DUMMY(18), do(17)=>DRAM_out_DUMMY(17),
do(16)=>DRAM_out_DUMMY(16), do(15)=>DRAM_out_DUMMY(15),
do(14)=>DRAM_out_DUMMY(14), do(13)=>DRAM_out_DUMMY(13),
do(12)=>DRAM_out_DUMMY(12), do(11)=>DRAM_out_DUMMY(11),
do(10)=>DRAM_out_DUMMY(10), do(9)=>DRAM_out_DUMMY(9),
do(8)=>DRAM_out_DUMMY(8), do(7)=>DRAM_out_DUMMY(7),
do(6)=>DRAM_out_DUMMY(6), do(5)=>DRAM_out_DUMMY(5),
do(4)=>DRAM_out_DUMMY(4), do(3)=>DRAM_out_DUMMY(3),
do(2)=>DRAM_out_DUMMY(2), do(1)=>DRAM_out_DUMMY(1),
do(0)=>DRAM_out_DUMMY(0));

instruction_ram : inst_ram
PORT MAP (clk=>Clk, we=>WE_Inst_RAM, a(7)=>Inst_RAM_Addr_DUMMY(7),
a(6)=>Inst_RAM_Addr_DUMMY(6), a(5)=>Inst_RAM_Addr_DUMMY(5),
a(4)=>Inst_RAM_Addr_DUMMY(4), a(3)=>Inst_RAM_Addr_DUMMY(3),
a(2)=>Inst_RAM_Addr_DUMMY(2), a(1)=>Inst_RAM_Addr_DUMMY(1),
a(0)=>Inst_RAM_Addr_DUMMY(0), di(31)=>Instruction_in_Preload(31),
di(30)=>Instruction_in_Preload(30), di(29)=>Instruction_in_Preload(29),
di(28)=>Instruction_in_Preload(28), di(27)=>Instruction_in_Preload(27),
di(26)=>Instruction_in_Preload(26), di(25)=>Instruction_in_Preload(25),
di(24)=>Instruction_in_Preload(24), di(23)=>Instruction_in_Preload(23),
di(22)=>Instruction_in_Preload(22), di(21)=>Instruction_in_Preload(21),
di(20)=>Instruction_in_Preload(20), di(19)=>Instruction_in_Preload(19),
di(18)=>Instruction_in_Preload(18), di(17)=>Instruction_in_Preload(17),
di(16)=>Instruction_in_Preload(16), di(15)=>Instruction_in_Preload(15),
di(14)=>Instruction_in_Preload(14), di(13)=>Instruction_in_Preload(13),
di(12)=>Instruction_in_Preload(12), di(11)=>Instruction_in_Preload(11),
di(10)=>Instruction_in_Preload(10), di(9)=>Instruction_in_Preload(9),
di(8)=>Instruction_in_Preload(8), di(7)=>Instruction_in_Preload(7),
di(6)=>Instruction_in_Preload(6), di(5)=>Instruction_in_Preload(5),
di(4)=>Instruction_in_Preload(4), di(3)=>Instruction_in_Preload(3),
di(2)=>Instruction_in_Preload(2), di(1)=>Instruction_in_Preload(1),
di(0)=>Instruction_in_Preload(0), do(31)=>Instruction_from_IF_DUMMY(31),
do(30)=>Instruction_from_IF_DUMMY(30),
do(29)=>Instruction_from_IF_DUMMY(29),
do(28)=>Instruction_from_IF_DUMMY(28),
do(27)=>Instruction_from_IF_DUMMY(27),
do(26)=>Instruction_from_IF_DUMMY(26),
do(25)=>Instruction_from_IF_DUMMY(25),
do(24)=>Instruction_from_IF_DUMMY(24),
do(23)=>Instruction_from_IF_DUMMY(23),
do(22)=>Instruction_from_IF_DUMMY(22),
do(21)=>Instruction_from_IF_DUMMY(21),
do(20)=>Instruction_from_IF_DUMMY(20),
do(19)=>Instruction_from_IF_DUMMY(19),
do(18)=>Instruction_from_IF_DUMMY(18),
do(17)=>Instruction_from_IF_DUMMY(17),
do(16)=>Instruction_from_IF_DUMMY(16),
do(15)=>Instruction_from_IF_DUMMY(15),
do(14)=>Instruction_from_IF_DUMMY(14),
do(13)=>Instruction_from_IF_DUMMY(13),

```



```

do(12)=>Instruction_from_IF_DUMMY(12),
do(11)=>Instruction_from_IF_DUMMY(11),
do(10)=>Instruction_from_IF_DUMMY(10),
do(9)=>Instruction_from_IF_DUMMY(9), do(8)=>Instruction_from_IF_DUMMY(8),
do(7)=>Instruction_from_IF_DUMMY(7), do(6)=>Instruction_from_IF_DUMMY(6),
do(5)=>Instruction_from_IF_DUMMY(5), do(4)=>Instruction_from_IF_DUMMY(4),
do(3)=>Instruction_from_IF_DUMMY(3), do(2)=>Instruction_from_IF_DUMMY(2),
do(1)=>Instruction_from_IF_DUMMY(1), do(0)=>Instruction_from_IF_DUMMY(0));

instruction_split : instruction_splitter
PORT MAP (instruction(31)=>Instruction_from_IF_DUMMY(31),
instruction(30)=>Instruction_from_IF_DUMMY(30),
instruction(29)=>Instruction_from_IF_DUMMY(29),
instruction(28)=>Instruction_from_IF_DUMMY(28),
instruction(27)=>Instruction_from_IF_DUMMY(27),
instruction(26)=>Instruction_from_IF_DUMMY(26),
instruction(25)=>Instruction_from_IF_DUMMY(25),
instruction(24)=>Instruction_from_IF_DUMMY(24),
instruction(23)=>Instruction_from_IF_DUMMY(23),
instruction(22)=>Instruction_from_IF_DUMMY(22),
instruction(21)=>Instruction_from_IF_DUMMY(21),
instruction(20)=>Instruction_from_IF_DUMMY(20),
instruction(19)=>Instruction_from_IF_DUMMY(19),
instruction(18)=>Instruction_from_IF_DUMMY(18),
instruction(17)=>Instruction_from_IF_DUMMY(17),
instruction(16)=>Instruction_from_IF_DUMMY(16),
instruction(15)=>Instruction_from_IF_DUMMY(15),
instruction(14)=>Instruction_from_IF_DUMMY(14),
instruction(13)=>Instruction_from_IF_DUMMY(13),
instruction(12)=>Instruction_from_IF_DUMMY(12),
instruction(11)=>Instruction_from_IF_DUMMY(11),
instruction(10)=>Instruction_from_IF_DUMMY(10),
instruction(9)=>Instruction_from_IF_DUMMY(9),
instruction(8)=>Instruction_from_IF_DUMMY(8),
instruction(7)=>Instruction_from_IF_DUMMY(7),
instruction(6)=>Instruction_from_IF_DUMMY(6),
instruction(5)=>Instruction_from_IF_DUMMY(5),
instruction(4)=>Instruction_from_IF_DUMMY(4),
instruction(3)=>Instruction_from_IF_DUMMY(3),
instruction(2)=>Instruction_from_IF_DUMMY(2),
instruction(1)=>Instruction_from_IF_DUMMY(1),
instruction(0)=>Instruction_from_IF_DUMMY(0), bits31_26(5)=>opcode(5),
bits31_26(4)=>opcode(4), bits31_26(3)=>opcode(3),
bits31_26(2)=>opcode(2), bits31_26(1)=>opcode(1),
bits31_26(0)=>opcode(0), bits25_21(4)=>rs_DUMMY(4),
bits25_21(3)=>rs_DUMMY(3), bits25_21(2)=>rs_DUMMY(2),
bits25_21(1)=>rs_DUMMY(1), bits25_21(0)=>rs_DUMMY(0),
bits20_16(4)=>rt_DUMMY(4), bits20_16(3)=>rt_DUMMY(3),
bits20_16(2)=>rt_DUMMY(2), bits20_16(1)=>rt_DUMMY(1),
bits20_16(0)=>rt_DUMMY(0), bits15_11(4)=>rd_DUMMY(4),
bits15_11(3)=>rd_DUMMY(3), bits15_11(2)=>rd_DUMMY(2),
bits15_11(1)=>rd_DUMMY(1), bits15_11(0)=>rd_DUMMY(0),
bits10_6(4)=>shamt(4), bits10_6(3)=>shamt(3), bits10_6(2)=>shamt(2),
bits10_6(1)=>shamt(1), bits10_6(0)=>shamt(0), bits5_0(5)=>funct(5),
bits5_0(4)=>funct(4), bits5_0(3)=>funct(3), bits5_0(2)=>funct(2),
bits5_0(1)=>funct(1), bits5_0(0)=>funct(0),
bits15_0(15)=>Addr_16bits_DUMMY(15), bits15_0(14)=>Addr_16bits_DUMMY(14),
bits15_0(13)=>Addr_16bits_DUMMY(13), bits15_0(12)=>Addr_16bits_DUMMY(12),
bits15_0(11)=>Addr_16bits_DUMMY(11), bits15_0(10)=>Addr_16bits_DUMMY(10),
bits15_0(9)=>Addr_16bits_DUMMY(9), bits15_0(8)=>Addr_16bits_DUMMY(8),
bits15_0(7)=>Addr_16bits_DUMMY(7), bits15_0(6)=>Addr_16bits_DUMMY(6),
bits15_0(5)=>Addr_16bits_DUMMY(5), bits15_0(4)=>Addr_16bits_DUMMY(4),
bits15_0(3)=>Addr_16bits_DUMMY(3), bits15_0(2)=>Addr_16bits_DUMMY(2),
bits15_0(1)=>Addr_16bits_DUMMY(1), bits15_0(0)=>Addr_16bits_DUMMY(0),
bits25_0(25)=>Addr_26bits_DUMMY(25), bits25_0(24)=>Addr_26bits_DUMMY(24),
bits25_0(23)=>Addr_26bits_DUMMY(23), bits25_0(22)=>Addr_26bits_DUMMY(22),
bits25_0(21)=>Addr_26bits_DUMMY(21), bits25_0(20)=>Addr_26bits_DUMMY(20),
bits25_0(19)=>Addr_26bits_DUMMY(19), bits25_0(18)=>Addr_26bits_DUMMY(18),
bits25_0(17)=>Addr_26bits_DUMMY(17), bits25_0(16)=>Addr_26bits_DUMMY(16),
bits25_0(15)=>Addr_26bits_DUMMY(15), bits25_0(14)=>Addr_26bits_DUMMY(14),
bits25_0(13)=>Addr_26bits_DUMMY(13), bits25_0(12)=>Addr_26bits_DUMMY(12),
bits25_0(11)=>Addr_26bits_DUMMY(11), bits25_0(10)=>Addr_26bits_DUMMY(10),
bits25_0(9)=>Addr_26bits_DUMMY(9), bits25_0(8)=>Addr_26bits_DUMMY(8),

```

```

bits25_0(7)=>Addr_26bits_DUMMY(7), bits25_0(6)=>Addr_26bits_DUMMY(6),
bits25_0(5)=>Addr_26bits_DUMMY(5), bits25_0(4)=>Addr_26bits_DUMMY(4),
bits25_0(3)=>Addr_26bits_DUMMY(3), bits25_0(2)=>Addr_26bits_DUMMY(2),
bits25_0(1)=>Addr_26bits_DUMMY(1), bits25_0(0)=>Addr_26bits_DUMMY(0));

inv_i1 : INV
PORT MAP (I=>ALU_Zero_BEQ_DUMMY, O=>ALU_Zero_BNE_DUMMY);

m2_1_i1 : M2_1_MXILINK_complete_datapath_new_beginning
PORT MAP (D0=>ALU_Zero_BEQ_DUMMY, D1=>ALU_Zero_BNE_DUMMY,
S0=>Mux_Sel_BEQ_BNE, O=>BEQ_or_BNE_DUMMY);

ALU_Mux_B_in : mux32b_2to1
PORT MAP (din0(31)=>RF_Data_B_DUMMY(31), din0(30)=>RF_Data_B_DUMMY(30),
din0(29)=>RF_Data_B_DUMMY(29), din0(28)=>RF_Data_B_DUMMY(28),
din0(27)=>RF_Data_B_DUMMY(27), din0(26)=>RF_Data_B_DUMMY(26),
din0(25)=>RF_Data_B_DUMMY(25), din0(24)=>RF_Data_B_DUMMY(24),
din0(23)=>RF_Data_B_DUMMY(23), din0(22)=>RF_Data_B_DUMMY(22),
din0(21)=>RF_Data_B_DUMMY(21), din0(20)=>RF_Data_B_DUMMY(20),
din0(19)=>RF_Data_B_DUMMY(19), din0(18)=>RF_Data_B_DUMMY(18),
din0(17)=>RF_Data_B_DUMMY(17), din0(16)=>RF_Data_B_DUMMY(16),
din0(15)=>RF_Data_B_DUMMY(15), din0(14)=>RF_Data_B_DUMMY(14),
din0(13)=>RF_Data_B_DUMMY(13), din0(12)=>RF_Data_B_DUMMY(12),
din0(11)=>RF_Data_B_DUMMY(11), din0(10)=>RF_Data_B_DUMMY(10),
din0(9)=>RF_Data_B_DUMMY(9), din0(8)=>RF_Data_B_DUMMY(8),
din0(7)=>RF_Data_B_DUMMY(7), din0(6)=>RF_Data_B_DUMMY(6),
din0(5)=>RF_Data_B_DUMMY(5), din0(4)=>RF_Data_B_DUMMY(4),
din0(3)=>RF_Data_B_DUMMY(3), din0(2)=>RF_Data_B_DUMMY(2),
din0(1)=>RF_Data_B_DUMMY(1), din0(0)=>RF_Data_B_DUMMY(0),
din1(31)=>Addr_16to32bits_DUMMY(31), din1(30)=>Addr_16to32bits_DUMMY(30),
din1(29)=>Addr_16to32bits_DUMMY(29), din1(28)=>Addr_16to32bits_DUMMY(28),
din1(27)=>Addr_16to32bits_DUMMY(27), din1(26)=>Addr_16to32bits_DUMMY(26),
din1(25)=>Addr_16to32bits_DUMMY(25), din1(24)=>Addr_16to32bits_DUMMY(24),
din1(23)=>Addr_16to32bits_DUMMY(23), din1(22)=>Addr_16to32bits_DUMMY(22),
din1(21)=>Addr_16to32bits_DUMMY(21), din1(20)=>Addr_16to32bits_DUMMY(20),
din1(19)=>Addr_16to32bits_DUMMY(19), din1(18)=>Addr_16to32bits_DUMMY(18),
din1(17)=>Addr_16to32bits_DUMMY(17), din1(16)=>Addr_16to32bits_DUMMY(16),
din1(15)=>Addr_16to32bits_DUMMY(15), din1(14)=>Addr_16to32bits_DUMMY(14),
din1(13)=>Addr_16to32bits_DUMMY(13), din1(12)=>Addr_16to32bits_DUMMY(12),
din1(11)=>Addr_16to32bits_DUMMY(11), din1(10)=>Addr_16to32bits_DUMMY(10),
din1(9)=>Addr_16to32bits_DUMMY(9), din1(8)=>Addr_16to32bits_DUMMY(8),
din1(7)=>Addr_16to32bits_DUMMY(7), din1(6)=>Addr_16to32bits_DUMMY(6),
din1(5)=>Addr_16to32bits_DUMMY(5), din1(4)=>Addr_16to32bits_DUMMY(4),
din1(3)=>Addr_16to32bits_DUMMY(3), din1(2)=>Addr_16to32bits_DUMMY(2),
din1(1)=>Addr_16to32bits_DUMMY(1), din1(0)=>Addr_16to32bits_DUMMY(0),
dout(31)=>B_in_DUMMY(31), dout(30)=>B_in_DUMMY(30),
dout(29)=>B_in_DUMMY(29), dout(28)=>B_in_DUMMY(28),
dout(27)=>B_in_DUMMY(27), dout(26)=>B_in_DUMMY(26),
dout(25)=>B_in_DUMMY(25), dout(24)=>B_in_DUMMY(24),
dout(23)=>B_in_DUMMY(23), dout(22)=>B_in_DUMMY(22),
dout(21)=>B_in_DUMMY(21), dout(20)=>B_in_DUMMY(20),
dout(19)=>B_in_DUMMY(19), dout(18)=>B_in_DUMMY(18),
dout(17)=>B_in_DUMMY(17), dout(16)=>B_in_DUMMY(16),
dout(15)=>B_in_DUMMY(15), dout(14)=>B_in_DUMMY(14),
dout(13)=>B_in_DUMMY(13), dout(12)=>B_in_DUMMY(12),
dout(11)=>B_in_DUMMY(11), dout(10)=>B_in_DUMMY(10),
dout(9)=>B_in_DUMMY(9), dout(8)=>B_in_DUMMY(8), dout(7)=>B_in_DUMMY(7),
dout(6)=>B_in_DUMMY(6), dout(5)=>B_in_DUMMY(5), dout(4)=>B_in_DUMMY(4),
dout(3)=>B_in_DUMMY(3), dout(2)=>B_in_DUMMY(2), dout(1)=>B_in_DUMMY(1),
dout(0)=>B_in_DUMMY(0), sel=>Sel_ALU_Mux_B_in);

DMem_Mux_di : mux32b_2to1
PORT MAP (din0(31)=>Data_in_Preload(31), din0(30)=>Data_in_Preload(30),
din0(29)=>Data_in_Preload(29), din0(28)=>Data_in_Preload(28),
din0(27)=>Data_in_Preload(27), din0(26)=>Data_in_Preload(26),
din0(25)=>Data_in_Preload(25), din0(24)=>Data_in_Preload(24),
din0(23)=>Data_in_Preload(23), din0(22)=>Data_in_Preload(22),
din0(21)=>Data_in_Preload(21), din0(20)=>Data_in_Preload(20),
din0(19)=>Data_in_Preload(19), din0(18)=>Data_in_Preload(18),
din0(17)=>Data_in_Preload(17), din0(16)=>Data_in_Preload(16),
din0(15)=>Data_in_Preload(15), din0(14)=>Data_in_Preload(14),
din0(13)=>Data_in_Preload(13), din0(12)=>Data_in_Preload(12),
din0(11)=>Data_in_Preload(11), din0(10)=>Data_in_Preload(10),

```

```

din0(9)=>Data_in_Preload(9), din0(8)=>Data_in_Preload(8),
din0(7)=>Data_in_Preload(7), din0(6)=>Data_in_Preload(6),
din0(5)=>Data_in_Preload(5), din0(4)=>Data_in_Preload(4),
din0(3)=>Data_in_Preload(3), din0(2)=>Data_in_Preload(2),
din0(1)=>Data_in_Preload(1), din0(0)=>Data_in_Preload(0),
din1(31)=>RF_Data_B_DUMMY(31), din1(30)=>RF_Data_B_DUMMY(30),
din1(29)=>RF_Data_B_DUMMY(29), din1(28)=>RF_Data_B_DUMMY(28),
din1(27)=>RF_Data_B_DUMMY(27), din1(26)=>RF_Data_B_DUMMY(26),
din1(25)=>RF_Data_B_DUMMY(25), din1(24)=>RF_Data_B_DUMMY(24),
din1(23)=>RF_Data_B_DUMMY(23), din1(22)=>RF_Data_B_DUMMY(22),
din1(21)=>RF_Data_B_DUMMY(21), din1(20)=>RF_Data_B_DUMMY(20),
din1(19)=>RF_Data_B_DUMMY(19), din1(18)=>RF_Data_B_DUMMY(18),
din1(17)=>RF_Data_B_DUMMY(17), din1(16)=>RF_Data_B_DUMMY(16),
din1(15)=>RF_Data_B_DUMMY(15), din1(14)=>RF_Data_B_DUMMY(14),
din1(13)=>RF_Data_B_DUMMY(13), din1(12)=>RF_Data_B_DUMMY(12),
din1(11)=>RF_Data_B_DUMMY(11), din1(10)=>RF_Data_B_DUMMY(10),
din1(9)=>RF_Data_B_DUMMY(9), din1(8)=>RF_Data_B_DUMMY(8),
din1(7)=>RF_Data_B_DUMMY(7), din1(6)=>RF_Data_B_DUMMY(6),
din1(5)=>RF_Data_B_DUMMY(5), din1(4)=>RF_Data_B_DUMMY(4),
din1(3)=>RF_Data_B_DUMMY(3), din1(2)=>RF_Data_B_DUMMY(2),
din1(1)=>RF_Data_B_DUMMY(1), din1(0)=>RF_Data_B_DUMMY(0),
dout(31)=>DMem_Din_DUMMY(31), dout(30)=>DMem_Din_DUMMY(30),
dout(29)=>DMem_Din_DUMMY(29), dout(28)=>DMem_Din_DUMMY(28),
dout(27)=>DMem_Din_DUMMY(27), dout(26)=>DMem_Din_DUMMY(26),
dout(25)=>DMem_Din_DUMMY(25), dout(24)=>DMem_Din_DUMMY(24),
dout(23)=>DMem_Din_DUMMY(23), dout(22)=>DMem_Din_DUMMY(22),
dout(21)=>DMem_Din_DUMMY(21), dout(20)=>DMem_Din_DUMMY(20),
dout(19)=>DMem_Din_DUMMY(19), dout(18)=>DMem_Din_DUMMY(18),
dout(17)=>DMem_Din_DUMMY(17), dout(16)=>DMem_Din_DUMMY(16),
dout(15)=>DMem_Din_DUMMY(15), dout(14)=>DMem_Din_DUMMY(14),
dout(13)=>DMem_Din_DUMMY(13), dout(12)=>DMem_Din_DUMMY(12),
dout(11)=>DMem_Din_DUMMY(11), dout(10)=>DMem_Din_DUMMY(10),
dout(9)=>DMem_Din_DUMMY(9), dout(8)=>DMem_Din_DUMMY(8),
dout(7)=>DMem_Din_DUMMY(7), dout(6)=>DMem_Din_DUMMY(6),
dout(5)=>DMem_Din_DUMMY(5), dout(4)=>DMem_Din_DUMMY(4),
dout(3)=>DMem_Din_DUMMY(3), dout(2)=>DMem_Din_DUMMY(2),
dout(1)=>DMem_Din_DUMMY(1), dout(0)=>DMem_Din_DUMMY(0),
sel=>Sel_DMem_Mux_di);

RF_Mux_W_Data : mux32b_4to1
PORT MAP (din0(31)=>ALU_Res_DUMMY(31), din0(30)=>ALU_Res_DUMMY(30),
din0(29)=>ALU_Res_DUMMY(29), din0(28)=>ALU_Res_DUMMY(28),
din0(27)=>ALU_Res_DUMMY(27), din0(26)=>ALU_Res_DUMMY(26),
din0(25)=>ALU_Res_DUMMY(25), din0(24)=>ALU_Res_DUMMY(24),
din0(23)=>ALU_Res_DUMMY(23), din0(22)=>ALU_Res_DUMMY(22),
din0(21)=>ALU_Res_DUMMY(21), din0(20)=>ALU_Res_DUMMY(20),
din0(19)=>ALU_Res_DUMMY(19), din0(18)=>ALU_Res_DUMMY(18),
din0(17)=>ALU_Res_DUMMY(17), din0(16)=>ALU_Res_DUMMY(16),
din0(15)=>ALU_Res_DUMMY(15), din0(14)=>ALU_Res_DUMMY(14),
din0(13)=>ALU_Res_DUMMY(13), din0(12)=>ALU_Res_DUMMY(12),
din0(11)=>ALU_Res_DUMMY(11), din0(10)=>ALU_Res_DUMMY(10),
din0(9)=>ALU_Res_DUMMY(9), din0(8)=>ALU_Res_DUMMY(8),
din0(7)=>ALU_Res_DUMMY(7), din0(6)=>ALU_Res_DUMMY(6),
din0(5)=>ALU_Res_DUMMY(5), din0(4)=>ALU_Res_DUMMY(4),
din0(3)=>ALU_Res_DUMMY(3), din0(2)=>ALU_Res_DUMMY(2),
din0(1)=>ALU_Res_DUMMY(1), din0(0)=>ALU_Res_DUMMY(0),
din1(31)=>DMem_to_RF_DUMMY(31), din1(30)=>DMem_to_RF_DUMMY(30),
din1(29)=>DMem_to_RF_DUMMY(29), din1(28)=>DMem_to_RF_DUMMY(28),
din1(27)=>DMem_to_RF_DUMMY(27), din1(26)=>DMem_to_RF_DUMMY(26),
din1(25)=>DMem_to_RF_DUMMY(25), din1(24)=>DMem_to_RF_DUMMY(24),
din1(23)=>DMem_to_RF_DUMMY(23), din1(22)=>DMem_to_RF_DUMMY(22),
din1(21)=>DMem_to_RF_DUMMY(21), din1(20)=>DMem_to_RF_DUMMY(20),
din1(19)=>DMem_to_RF_DUMMY(19), din1(18)=>DMem_to_RF_DUMMY(18),
din1(17)=>DMem_to_RF_DUMMY(17), din1(16)=>DMem_to_RF_DUMMY(16),
din1(15)=>DMem_to_RF_DUMMY(15), din1(14)=>DMem_to_RF_DUMMY(14),
din1(13)=>DMem_to_RF_DUMMY(13), din1(12)=>DMem_to_RF_DUMMY(12),
din1(11)=>DMem_to_RF_DUMMY(11), din1(10)=>DMem_to_RF_DUMMY(10),
din1(9)=>DMem_to_RF_DUMMY(9), din1(8)=>DMem_to_RF_DUMMY(8),
din1(7)=>DMem_to_RF_DUMMY(7), din1(6)=>DMem_to_RF_DUMMY(6),
din1(5)=>DMem_to_RF_DUMMY(5), din1(4)=>DMem_to_RF_DUMMY(4),
din1(3)=>DMem_to_RF_DUMMY(3), din1(2)=>DMem_to_RF_DUMMY(2),
din1(1)=>DMem_to_RF_DUMMY(1), din1(0)=>DMem_to_RF_DUMMY(0),
din2(31)=>RF_Write_Din_Preload(31), din2(30)=>RF_Write_Din_Preload(30),

```

```

din2(29)=>RF_Write_Din_Preload(29), din2(28)=>RF_Write_Din_Preload(28),
din2(27)=>RF_Write_Din_Preload(27), din2(26)=>RF_Write_Din_Preload(26),
din2(25)=>RF_Write_Din_Preload(25), din2(24)=>RF_Write_Din_Preload(24),
din2(23)=>RF_Write_Din_Preload(23), din2(22)=>RF_Write_Din_Preload(22),
din2(21)=>RF_Write_Din_Preload(21), din2(20)=>RF_Write_Din_Preload(20),
din2(19)=>RF_Write_Din_Preload(19), din2(18)=>RF_Write_Din_Preload(18),
din2(17)=>RF_Write_Din_Preload(17), din2(16)=>RF_Write_Din_Preload(16),
din2(15)=>RF_Write_Din_Preload(15), din2(14)=>RF_Write_Din_Preload(14),
din2(13)=>RF_Write_Din_Preload(13), din2(12)=>RF_Write_Din_Preload(12),
din2(11)=>RF_Write_Din_Preload(11), din2(10)=>RF_Write_Din_Preload(10),
din2(9)=>RF_Write_Din_Preload(9), din2(8)=>RF_Write_Din_Preload(8),
din2(7)=>RF_Write_Din_Preload(7), din2(6)=>RF_Write_Din_Preload(6),
din2(5)=>RF_Write_Din_Preload(5), din2(4)=>RF_Write_Din_Preload(4),
din2(3)=>RF_Write_Din_Preload(3), din2(2)=>RF_Write_Din_Preload(2),
din2(1)=>RF_Write_Din_Preload(1), din2(0)=>RF_Write_Din_Preload(0),
din3(31)=>Value_of_Zero(31), din3(30)=>Value_of_Zero(30),
din3(29)=>Value_of_Zero(29), din3(28)=>Value_of_Zero(28),
din3(27)=>Value_of_Zero(27), din3(26)=>Value_of_Zero(26),
din3(25)=>Value_of_Zero(25), din3(24)=>Value_of_Zero(24),
din3(23)=>Value_of_Zero(23), din3(22)=>Value_of_Zero(22),
din3(21)=>Value_of_Zero(21), din3(20)=>Value_of_Zero(20),
din3(19)=>Value_of_Zero(19), din3(18)=>Value_of_Zero(18),
din3(17)=>Value_of_Zero(17), din3(16)=>Value_of_Zero(16),
din3(15)=>Value_of_Zero(15), din3(14)=>Value_of_Zero(14),
din3(13)=>Value_of_Zero(13), din3(12)=>Value_of_Zero(12),
din3(11)=>Value_of_Zero(11), din3(10)=>Value_of_Zero(10),
din3(9)=>Value_of_Zero(9), din3(8)=>Value_of_Zero(8),
din3(7)=>Value_of_Zero(7), din3(6)=>Value_of_Zero(6),
din3(5)=>Value_of_Zero(5), din3(4)=>Value_of_Zero(4),
din3(3)=>Value_of_Zero(3), din3(2)=>Value_of_Zero(2),
din3(1)=>Value_of_Zero(1), din3(0)=>Value_of_Zero(0),
dout(31)=>RF_Write_Data_DUMMY(31), dout(30)=>RF_Write_Data_DUMMY(30),
dout(29)=>RF_Write_Data_DUMMY(29), dout(28)=>RF_Write_Data_DUMMY(28),
dout(27)=>RF_Write_Data_DUMMY(27), dout(26)=>RF_Write_Data_DUMMY(26),
dout(25)=>RF_Write_Data_DUMMY(25), dout(24)=>RF_Write_Data_DUMMY(24),
dout(23)=>RF_Write_Data_DUMMY(23), dout(22)=>RF_Write_Data_DUMMY(22),
dout(21)=>RF_Write_Data_DUMMY(21), dout(20)=>RF_Write_Data_DUMMY(20),
dout(19)=>RF_Write_Data_DUMMY(19), dout(18)=>RF_Write_Data_DUMMY(18),
dout(17)=>RF_Write_Data_DUMMY(17), dout(16)=>RF_Write_Data_DUMMY(16),
dout(15)=>RF_Write_Data_DUMMY(15), dout(14)=>RF_Write_Data_DUMMY(14),
dout(13)=>RF_Write_Data_DUMMY(13), dout(12)=>RF_Write_Data_DUMMY(12),
dout(11)=>RF_Write_Data_DUMMY(11), dout(10)=>RF_Write_Data_DUMMY(10),
dout(9)=>RF_Write_Data_DUMMY(9), dout(8)=>RF_Write_Data_DUMMY(8),
dout(7)=>RF_Write_Data_DUMMY(7), dout(6)=>RF_Write_Data_DUMMY(6),
dout(5)=>RF_Write_Data_DUMMY(5), dout(4)=>RF_Write_Data_DUMMY(4),
dout(3)=>RF_Write_Data_DUMMY(3), dout(2)=>RF_Write_Data_DUMMY(2),
dout(1)=>RF_Write_Data_DUMMY(1), dout(0)=>RF_Write_Data_DUMMY(0),
sel(1)=>Mux_RF_din_Sel(1), sel(0)=>Mux_RF_din_Sel(0));

RF_Mux_W_Reg : mux5b_4to1
PORT MAP (din0(4)=>rs_DUMMY(4), din0(3)=>rs_DUMMY(3),
din0(2)=>rs_DUMMY(2), din0(1)=>rs_DUMMY(1), din0(0)=>rs_DUMMY(0),
din1(4)=>rt_DUMMY(4), din1(3)=>rt_DUMMY(3), din1(2)=>rt_DUMMY(2),
din1(1)=>rt_DUMMY(1), din1(0)=>rt_DUMMY(0), din2(4)=>rd_DUMMY(4),
din2(3)=>rd_DUMMY(3), din2(2)=>rd_DUMMY(2), din2(1)=>rd_DUMMY(1),
din2(0)=>rd_DUMMY(0), din3(4)=>RF_Write_Num_Preload(4),
din3(3)=>RF_Write_Num_Preload(3), din3(2)=>RF_Write_Num_Preload(2),
din3(1)=>RF_Write_Num_Preload(1), din3(0)=>RF_Write_Num_Preload(0),
dout(4)=>RF_Write_Reg_DUMMY(4), dout(3)=>RF_Write_Reg_DUMMY(3),
dout(2)=>RF_Write_Reg_DUMMY(2), dout(1)=>RF_Write_Reg_DUMMY(1),
dout(0)=>RF_Write_Reg_DUMMY(0), sel(1)=>Mux_RF_Num_Sel(1),
sel(0)=>Mux_RF_Num_Sel(0));

mux8_2to1_i1 : mux8_2to1
PORT MAP (sel=>Mux_Sel_Inst_RAM_Addr_Src, din0(7)=>Addr_from_PC_DUMMY(7),
din0(6)=>Addr_from_PC_DUMMY(6), din0(5)=>Addr_from_PC_DUMMY(5),
din0(4)=>Addr_from_PC_DUMMY(4), din0(3)=>Addr_from_PC_DUMMY(3),
din0(2)=>Addr_from_PC_DUMMY(2), din0(1)=>Addr_from_PC_DUMMY(1),
din0(0)=>Addr_from_PC_DUMMY(0), din1(7)=>Inst_RAM_Write_Addr_Preload(7),
din1(6)=>Inst_RAM_Write_Addr_Preload(6),
din1(5)=>Inst_RAM_Write_Addr_Preload(5),
din1(4)=>Inst_RAM_Write_Addr_Preload(4),
din1(3)=>Inst_RAM_Write_Addr_Preload(3),

```

```

    dinl(2)=>Inst_RAM_Write_Addr_Preload(2),
    dinl(1)=>Inst_RAM_Write_Addr_Preload(1),
    dinl(0)=>Inst_RAM_Write_Addr_Preload(0), dout(7)=>Inst_RAM_Addr_DUMMY(7),
    dout(6)=>Inst_RAM_Addr_DUMMY(6), dout(5)=>Inst_RAM_Addr_DUMMY(5),
    dout(4)=>Inst_RAM_Addr_DUMMY(4), dout(3)=>Inst_RAM_Addr_DUMMY(3),
    dout(2)=>Inst_RAM_Addr_DUMMY(2), dout(1)=>Inst_RAM_Addr_DUMMY(1),
    dout(0)=>Inst_RAM_Addr_DUMMY(0));

mux8_2to1_i2 : mux8_2to1
PORT MAP (sel=>Mux_Sel_PCSrc2, din0(7)=>Mux2Mux_PCSrc_DUMMY(7),
    din0(6)=>Mux2Mux_PCSrc_DUMMY(6), din0(5)=>Mux2Mux_PCSrc_DUMMY(5),
    din0(4)=>Mux2Mux_PCSrc_DUMMY(4), din0(3)=>Mux2Mux_PCSrc_DUMMY(3),
    din0(2)=>Mux2Mux_PCSrc_DUMMY(2), din0(1)=>Mux2Mux_PCSrc_DUMMY(1),
    din0(0)=>Mux2Mux_PCSrc_DUMMY(0), dinl(7)=>Addr_26to8bits_DUMMY(7),
    dinl(6)=>Addr_26to8bits_DUMMY(6), dinl(5)=>Addr_26to8bits_DUMMY(5),
    dinl(4)=>Addr_26to8bits_DUMMY(4), dinl(3)=>Addr_26to8bits_DUMMY(3),
    dinl(2)=>Addr_26to8bits_DUMMY(2), dinl(1)=>Addr_26to8bits_DUMMY(1),
    dinl(0)=>Addr_26to8bits_DUMMY(0), dout(7)=>Next_PC_DUMMY(7),
    dout(6)=>Next_PC_DUMMY(6), dout(5)=>Next_PC_DUMMY(5),
    dout(4)=>Next_PC_DUMMY(4), dout(3)=>Next_PC_DUMMY(3),
    dout(2)=>Next_PC_DUMMY(2), dout(1)=>Next_PC_DUMMY(1),
    dout(0)=>Next_PC_DUMMY(0));

mux8_2to1_i3 : mux8_2to1
PORT MAP (sel=>Mux_Sel_PCSrc1_DUMMY, din0(7)=>PCplus1_Addr_DUMMY(7),
    din0(6)=>PCplus1_Addr_DUMMY(6), din0(5)=>PCplus1_Addr_DUMMY(5),
    din0(4)=>PCplus1_Addr_DUMMY(4), din0(3)=>PCplus1_Addr_DUMMY(3),
    din0(2)=>PCplus1_Addr_DUMMY(2), din0(1)=>PCplus1_Addr_DUMMY(1),
    din0(0)=>PCplus1_Addr_DUMMY(0), dinl(7)=>Branch_Target_DUMMY(7),
    dinl(6)=>Branch_Target_DUMMY(6), dinl(5)=>Branch_Target_DUMMY(5),
    dinl(4)=>Branch_Target_DUMMY(4), dinl(3)=>Branch_Target_DUMMY(3),
    dinl(2)=>Branch_Target_DUMMY(2), dinl(1)=>Branch_Target_DUMMY(1),
    dinl(0)=>Branch_Target_DUMMY(0), dout(7)=>Mux2Mux_PCSrc_DUMMY(7),
    dout(6)=>Mux2Mux_PCSrc_DUMMY(6), dout(5)=>Mux2Mux_PCSrc_DUMMY(5),
    dout(4)=>Mux2Mux_PCSrc_DUMMY(4), dout(3)=>Mux2Mux_PCSrc_DUMMY(3),
    dout(2)=>Mux2Mux_PCSrc_DUMMY(2), dout(1)=>Mux2Mux_PCSrc_DUMMY(1),
    dout(0)=>Mux2Mux_PCSrc_DUMMY(0));

DMem_Mux_ra : mux8b_2to1
PORT MAP (din0(7)=>DMem_RA_Preload(7), din0(6)=>DMem_RA_Preload(6),
    din0(5)=>DMem_RA_Preload(5), din0(4)=>DMem_RA_Preload(4),
    din0(3)=>DMem_RA_Preload(3), din0(2)=>DMem_RA_Preload(2),
    din0(1)=>DMem_RA_Preload(1), din0(0)=>DMem_RA_Preload(0),
    dinl(31)=>ALU_Res_DUMMY(31), dinl(30)=>ALU_Res_DUMMY(30),
    dinl(29)=>ALU_Res_DUMMY(29), dinl(28)=>ALU_Res_DUMMY(28),
    dinl(27)=>ALU_Res_DUMMY(27), dinl(26)=>ALU_Res_DUMMY(26),
    dinl(25)=>ALU_Res_DUMMY(25), dinl(24)=>ALU_Res_DUMMY(24),
    dinl(23)=>ALU_Res_DUMMY(23), dinl(22)=>ALU_Res_DUMMY(22),
    dinl(21)=>ALU_Res_DUMMY(21), dinl(20)=>ALU_Res_DUMMY(20),
    dinl(19)=>ALU_Res_DUMMY(19), dinl(18)=>ALU_Res_DUMMY(18),
    dinl(17)=>ALU_Res_DUMMY(17), dinl(16)=>ALU_Res_DUMMY(16),
    dinl(15)=>ALU_Res_DUMMY(15), dinl(14)=>ALU_Res_DUMMY(14),
    dinl(13)=>ALU_Res_DUMMY(13), dinl(12)=>ALU_Res_DUMMY(12),
    dinl(11)=>ALU_Res_DUMMY(11), dinl(10)=>ALU_Res_DUMMY(10),
    dinl(9)=>ALU_Res_DUMMY(9), dinl(8)=>ALU_Res_DUMMY(8),
    dinl(7)=>ALU_Res_DUMMY(7), dinl(6)=>ALU_Res_DUMMY(6),
    dinl(5)=>ALU_Res_DUMMY(5), dinl(4)=>ALU_Res_DUMMY(4),
    dinl(3)=>ALU_Res_DUMMY(3), dinl(2)=>ALU_Res_DUMMY(2),
    dinl(1)=>ALU_Res_DUMMY(1), dinl(0)=>ALU_Res_DUMMY(0),
    dout(7)=>Data_RAM_Read_Addr_DUMMY(7),
    dout(6)=>Data_RAM_Read_Addr_DUMMY(6),
    dout(5)=>Data_RAM_Read_Addr_DUMMY(5),
    dout(4)=>Data_RAM_Read_Addr_DUMMY(4),
    dout(3)=>Data_RAM_Read_Addr_DUMMY(3),
    dout(2)=>Data_RAM_Read_Addr_DUMMY(2),
    dout(1)=>Data_RAM_Read_Addr_DUMMY(1),
    dout(0)=>Data_RAM_Read_Addr_DUMMY(0), sel=>Sel_DMem_Mux_ra);

DMem_Mux_wa : mux8b_2to1
PORT MAP (din0(7)=>DMem_WA_Preload(7), din0(6)=>DMem_WA_Preload(6),
    din0(5)=>DMem_WA_Preload(5), din0(4)=>DMem_WA_Preload(4),
    din0(3)=>DMem_WA_Preload(3), din0(2)=>DMem_WA_Preload(2),
    din0(1)=>DMem_WA_Preload(1), din0(0)=>DMem_WA_Preload(0),

```

```

dinl(31)=>ALU_Res_DUMMY(31), dinl(30)=>ALU_Res_DUMMY(30),
dinl(29)=>ALU_Res_DUMMY(29), dinl(28)=>ALU_Res_DUMMY(28),
dinl(27)=>ALU_Res_DUMMY(27), dinl(26)=>ALU_Res_DUMMY(26),
dinl(25)=>ALU_Res_DUMMY(25), dinl(24)=>ALU_Res_DUMMY(24),
dinl(23)=>ALU_Res_DUMMY(23), dinl(22)=>ALU_Res_DUMMY(22),
dinl(21)=>ALU_Res_DUMMY(21), dinl(20)=>ALU_Res_DUMMY(20),
dinl(19)=>ALU_Res_DUMMY(19), dinl(18)=>ALU_Res_DUMMY(18),
dinl(17)=>ALU_Res_DUMMY(17), dinl(16)=>ALU_Res_DUMMY(16),
dinl(15)=>ALU_Res_DUMMY(15), dinl(14)=>ALU_Res_DUMMY(14),
dinl(13)=>ALU_Res_DUMMY(13), dinl(12)=>ALU_Res_DUMMY(12),
dinl(11)=>ALU_Res_DUMMY(11), dinl(10)=>ALU_Res_DUMMY(10),
dinl(9)=>ALU_Res_DUMMY(9), dinl(8)=>ALU_Res_DUMMY(8),
dinl(7)=>ALU_Res_DUMMY(7), dinl(6)=>ALU_Res_DUMMY(6),
dinl(5)=>ALU_Res_DUMMY(5), dinl(4)=>ALU_Res_DUMMY(4),
dinl(3)=>ALU_Res_DUMMY(3), dinl(2)=>ALU_Res_DUMMY(2),
dinl(1)=>ALU_Res_DUMMY(1), dinl(0)=>ALU_Res_DUMMY(0),
dout(7)=>Data_RAM_Write_Addr_DUMMY(7),
dout(6)=>Data_RAM_Write_Addr_DUMMY(6),
dout(5)=>Data_RAM_Write_Addr_DUMMY(5),
dout(4)=>Data_RAM_Write_Addr_DUMMY(4),
dout(3)=>Data_RAM_Write_Addr_DUMMY(3),
dout(2)=>Data_RAM_Write_Addr_DUMMY(2),
dout(1)=>Data_RAM_Write_Addr_DUMMY(1),
dout(0)=>Data_RAM_Write_Addr_DUMMY(0), sel=>Sel_DMem_Mux_wa);

Program_Counter : pc
PORT MAP (clk=>Clk, reset=>Reset, din(7)=>Next_PC_DUMMY(7),
din(6)=>Next_PC_DUMMY(6), din(5)=>Next_PC_DUMMY(5),
din(4)=>Next_PC_DUMMY(4), din(3)=>Next_PC_DUMMY(3),
din(2)=>Next_PC_DUMMY(2), din(1)=>Next_PC_DUMMY(1),
din(0)=>Next_PC_DUMMY(0), dout(7)=>Addr_from_PC_DUMMY(7),
dout(6)=>Addr_from_PC_DUMMY(6), dout(5)=>Addr_from_PC_DUMMY(5),
dout(4)=>Addr_from_PC_DUMMY(4), dout(3)=>Addr_from_PC_DUMMY(3),
dout(2)=>Addr_from_PC_DUMMY(2), dout(1)=>Addr_from_PC_DUMMY(1),
dout(0)=>Addr_from_PC_DUMMY(0));

Register_File : rf_synch
PORT MAP (clk=>Clk, enable=>RF_En_Read_Write, read_reg1(4)=>rs_DUMMY(4),
read_reg1(3)=>rs_DUMMY(3), read_reg1(2)=>rs_DUMMY(2),
read_reg1(1)=>rs_DUMMY(1), read_reg1(0)=>rs_DUMMY(0),
read_reg2(4)=>rt_DUMMY(4), read_reg2(3)=>rt_DUMMY(3),
read_reg2(2)=>rt_DUMMY(2), read_reg2(1)=>rt_DUMMY(1),
read_reg2(0)=>rt_DUMMY(0), write_data(31)=>RF_Write_Data_DUMMY(31),
write_data(30)=>RF_Write_Data_DUMMY(30),
write_data(29)=>RF_Write_Data_DUMMY(29),
write_data(28)=>RF_Write_Data_DUMMY(28),
write_data(27)=>RF_Write_Data_DUMMY(27),
write_data(26)=>RF_Write_Data_DUMMY(26),
write_data(25)=>RF_Write_Data_DUMMY(25),
write_data(24)=>RF_Write_Data_DUMMY(24),
write_data(23)=>RF_Write_Data_DUMMY(23),
write_data(22)=>RF_Write_Data_DUMMY(22),
write_data(21)=>RF_Write_Data_DUMMY(21),
write_data(20)=>RF_Write_Data_DUMMY(20),
write_data(19)=>RF_Write_Data_DUMMY(19),
write_data(18)=>RF_Write_Data_DUMMY(18),
write_data(17)=>RF_Write_Data_DUMMY(17),
write_data(16)=>RF_Write_Data_DUMMY(16),
write_data(15)=>RF_Write_Data_DUMMY(15),
write_data(14)=>RF_Write_Data_DUMMY(14),
write_data(13)=>RF_Write_Data_DUMMY(13),
write_data(12)=>RF_Write_Data_DUMMY(12),
write_data(11)=>RF_Write_Data_DUMMY(11),
write_data(10)=>RF_Write_Data_DUMMY(10),
write_data(9)=>RF_Write_Data_DUMMY(9),
write_data(8)=>RF_Write_Data_DUMMY(8),
write_data(7)=>RF_Write_Data_DUMMY(7),
write_data(6)=>RF_Write_Data_DUMMY(6),
write_data(5)=>RF_Write_Data_DUMMY(5),
write_data(4)=>RF_Write_Data_DUMMY(4),
write_data(3)=>RF_Write_Data_DUMMY(3),
write_data(2)=>RF_Write_Data_DUMMY(2),
write_data(1)=>RF_Write_Data_DUMMY(1),

```

```

write_data(0)=>RF_Write_Data_DUMMY(0),
write_reg(4)=>RF_Write_Reg_DUMMY(4), write_reg(3)=>RF_Write_Reg_DUMMY(3),
write_reg(2)=>RF_Write_Reg_DUMMY(2), write_reg(1)=>RF_Write_Reg_DUMMY(1),
write_reg(0)=>RF_Write_Reg_DUMMY(0), read_data_1(31)=>A_in_DUMMY(31),
read_data_1(30)=>A_in_DUMMY(30), read_data_1(29)=>A_in_DUMMY(29),
read_data_1(28)=>A_in_DUMMY(28), read_data_1(27)=>A_in_DUMMY(27),
read_data_1(26)=>A_in_DUMMY(26), read_data_1(25)=>A_in_DUMMY(25),
read_data_1(24)=>A_in_DUMMY(24), read_data_1(23)=>A_in_DUMMY(23),
read_data_1(22)=>A_in_DUMMY(22), read_data_1(21)=>A_in_DUMMY(21),
read_data_1(20)=>A_in_DUMMY(20), read_data_1(19)=>A_in_DUMMY(19),
read_data_1(18)=>A_in_DUMMY(18), read_data_1(17)=>A_in_DUMMY(17),
read_data_1(16)=>A_in_DUMMY(16), read_data_1(15)=>A_in_DUMMY(15),
read_data_1(14)=>A_in_DUMMY(14), read_data_1(13)=>A_in_DUMMY(13),
read_data_1(12)=>A_in_DUMMY(12), read_data_1(11)=>A_in_DUMMY(11),
read_data_1(10)=>A_in_DUMMY(10), read_data_1(9)=>A_in_DUMMY(9),
read_data_1(8)=>A_in_DUMMY(8), read_data_1(7)=>A_in_DUMMY(7),
read_data_1(6)=>A_in_DUMMY(6), read_data_1(5)=>A_in_DUMMY(5),
read_data_1(4)=>A_in_DUMMY(4), read_data_1(3)=>A_in_DUMMY(3),
read_data_1(2)=>A_in_DUMMY(2), read_data_1(1)=>A_in_DUMMY(1),
read_data_1(0)=>A_in_DUMMY(0), read_data_2(31)=>RF_Data_B_DUMMY(31),
read_data_2(30)=>RF_Data_B_DUMMY(30),
read_data_2(29)=>RF_Data_B_DUMMY(29),
read_data_2(28)=>RF_Data_B_DUMMY(28),
read_data_2(27)=>RF_Data_B_DUMMY(27),
read_data_2(26)=>RF_Data_B_DUMMY(26),
read_data_2(25)=>RF_Data_B_DUMMY(25),
read_data_2(24)=>RF_Data_B_DUMMY(24),
read_data_2(23)=>RF_Data_B_DUMMY(23),
read_data_2(22)=>RF_Data_B_DUMMY(22),
read_data_2(21)=>RF_Data_B_DUMMY(21),
read_data_2(20)=>RF_Data_B_DUMMY(20),
read_data_2(19)=>RF_Data_B_DUMMY(19),
read_data_2(18)=>RF_Data_B_DUMMY(18),
read_data_2(17)=>RF_Data_B_DUMMY(17),
read_data_2(16)=>RF_Data_B_DUMMY(16),
read_data_2(15)=>RF_Data_B_DUMMY(15),
read_data_2(14)=>RF_Data_B_DUMMY(14),
read_data_2(13)=>RF_Data_B_DUMMY(13),
read_data_2(12)=>RF_Data_B_DUMMY(12),
read_data_2(11)=>RF_Data_B_DUMMY(11),
read_data_2(10)=>RF_Data_B_DUMMY(10), read_data_2(9)=>RF_Data_B_DUMMY(9),
read_data_2(8)=>RF_Data_B_DUMMY(8), read_data_2(7)=>RF_Data_B_DUMMY(7),
read_data_2(6)=>RF_Data_B_DUMMY(6), read_data_2(5)=>RF_Data_B_DUMMY(5),
read_data_2(4)=>RF_Data_B_DUMMY(4), read_data_2(3)=>RF_Data_B_DUMMY(3),
read_data_2(2)=>RF_Data_B_DUMMY(2), read_data_2(1)=>RF_Data_B_DUMMY(1),
read_data_2(0)=>RF_Data_B_DUMMY(0));

sign_dextend : sign_dextension
PORT MAP (addr_in(15)=>Addr_16bits_DUMMY(15),
addr_in(14)=>Addr_16bits_DUMMY(14), addr_in(13)=>Addr_16bits_DUMMY(13),
addr_in(12)=>Addr_16bits_DUMMY(12), addr_in(11)=>Addr_16bits_DUMMY(11),
addr_in(10)=>Addr_16bits_DUMMY(10), addr_in(9)=>Addr_16bits_DUMMY(9),
addr_in(8)=>Addr_16bits_DUMMY(8), addr_in(7)=>Addr_16bits_DUMMY(7),
addr_in(6)=>Addr_16bits_DUMMY(6), addr_in(5)=>Addr_16bits_DUMMY(5),
addr_in(4)=>Addr_16bits_DUMMY(4), addr_in(3)=>Addr_16bits_DUMMY(3),
addr_in(2)=>Addr_16bits_DUMMY(2), addr_in(1)=>Addr_16bits_DUMMY(1),
addr_in(0)=>Addr_16bits_DUMMY(0), addr_out(7)=>Addr_16to8bits_DUMMY(7),
addr_out(6)=>Addr_16to8bits_DUMMY(6),
addr_out(5)=>Addr_16to8bits_DUMMY(5),
addr_out(4)=>Addr_16to8bits_DUMMY(4),
addr_out(3)=>Addr_16to8bits_DUMMY(3),
addr_out(2)=>Addr_16to8bits_DUMMY(2),
addr_out(1)=>Addr_16to8bits_DUMMY(1),
addr_out(0)=>Addr_16to8bits_DUMMY(0));

sign_extend : sign_extension
PORT MAP (addr_in(15)=>Addr_16bits_DUMMY(15),
addr_in(14)=>Addr_16bits_DUMMY(14), addr_in(13)=>Addr_16bits_DUMMY(13),
addr_in(12)=>Addr_16bits_DUMMY(12), addr_in(11)=>Addr_16bits_DUMMY(11),
addr_in(10)=>Addr_16bits_DUMMY(10), addr_in(9)=>Addr_16bits_DUMMY(9),
addr_in(8)=>Addr_16bits_DUMMY(8), addr_in(7)=>Addr_16bits_DUMMY(7),
addr_in(6)=>Addr_16bits_DUMMY(6), addr_in(5)=>Addr_16bits_DUMMY(5),
addr_in(4)=>Addr_16bits_DUMMY(4), addr_in(3)=>Addr_16bits_DUMMY(3),

```

```

addr_in(2)=>Addr_16bits_DUMMY(2), addr_in(1)=>Addr_16bits_DUMMY(1),
addr_in(0)=>Addr_16bits_DUMMY(0),
addr_out(31)=>Addr_16to32bits_DUMMY(31),
addr_out(30)=>Addr_16to32bits_DUMMY(30),
addr_out(29)=>Addr_16to32bits_DUMMY(29),
addr_out(28)=>Addr_16to32bits_DUMMY(28),
addr_out(27)=>Addr_16to32bits_DUMMY(27),
addr_out(26)=>Addr_16to32bits_DUMMY(26),
addr_out(25)=>Addr_16to32bits_DUMMY(25),
addr_out(24)=>Addr_16to32bits_DUMMY(24),
addr_out(23)=>Addr_16to32bits_DUMMY(23),
addr_out(22)=>Addr_16to32bits_DUMMY(22),
addr_out(21)=>Addr_16to32bits_DUMMY(21),
addr_out(20)=>Addr_16to32bits_DUMMY(20),
addr_out(19)=>Addr_16to32bits_DUMMY(19),
addr_out(18)=>Addr_16to32bits_DUMMY(18),
addr_out(17)=>Addr_16to32bits_DUMMY(17),
addr_out(16)=>Addr_16to32bits_DUMMY(16),
addr_out(15)=>Addr_16to32bits_DUMMY(15),
addr_out(14)=>Addr_16to32bits_DUMMY(14),
addr_out(13)=>Addr_16to32bits_DUMMY(13),
addr_out(12)=>Addr_16to32bits_DUMMY(12),
addr_out(11)=>Addr_16to32bits_DUMMY(11),
addr_out(10)=>Addr_16to32bits_DUMMY(10),
addr_out(9)=>Addr_16to32bits_DUMMY(9),
addr_out(8)=>Addr_16to32bits_DUMMY(8),
addr_out(7)=>Addr_16to32bits_DUMMY(7),
addr_out(6)=>Addr_16to32bits_DUMMY(6),
addr_out(5)=>Addr_16to32bits_DUMMY(5),
addr_out(4)=>Addr_16to32bits_DUMMY(4),
addr_out(3)=>Addr_16to32bits_DUMMY(3),
addr_out(2)=>Addr_16to32bits_DUMMY(2),
addr_out(1)=>Addr_16to32bits_DUMMY(1),
addr_out(0)=>Addr_16to32bits_DUMMY(0);

TBuff_32b : tristate_32b
PORT MAP (enable=>Mem_Read, din(31)=>DRAM_out_DUMMY(31),
din(30)=>DRAM_out_DUMMY(30), din(29)=>DRAM_out_DUMMY(29),
din(28)=>DRAM_out_DUMMY(28), din(27)=>DRAM_out_DUMMY(27),
din(26)=>DRAM_out_DUMMY(26), din(25)=>DRAM_out_DUMMY(25),
din(24)=>DRAM_out_DUMMY(24), din(23)=>DRAM_out_DUMMY(23),
din(22)=>DRAM_out_DUMMY(22), din(21)=>DRAM_out_DUMMY(21),
din(20)=>DRAM_out_DUMMY(20), din(19)=>DRAM_out_DUMMY(19),
din(18)=>DRAM_out_DUMMY(18), din(17)=>DRAM_out_DUMMY(17),
din(16)=>DRAM_out_DUMMY(16), din(15)=>DRAM_out_DUMMY(15),
din(14)=>DRAM_out_DUMMY(14), din(13)=>DRAM_out_DUMMY(13),
din(12)=>DRAM_out_DUMMY(12), din(11)=>DRAM_out_DUMMY(11),
din(10)=>DRAM_out_DUMMY(10), din(9)=>DRAM_out_DUMMY(9),
din(8)=>DRAM_out_DUMMY(8), din(7)=>DRAM_out_DUMMY(7),
din(6)=>DRAM_out_DUMMY(6), din(5)=>DRAM_out_DUMMY(5),
din(4)=>DRAM_out_DUMMY(4), din(3)=>DRAM_out_DUMMY(3),
din(2)=>DRAM_out_DUMMY(2), din(1)=>DRAM_out_DUMMY(1),
din(0)=>DRAM_out_DUMMY(0), dout(31)=>DMem_to_RF_DUMMY(31),
dout(30)=>DMem_to_RF_DUMMY(30), dout(29)=>DMem_to_RF_DUMMY(29),
dout(28)=>DMem_to_RF_DUMMY(28), dout(27)=>DMem_to_RF_DUMMY(27),
dout(26)=>DMem_to_RF_DUMMY(26), dout(25)=>DMem_to_RF_DUMMY(25),
dout(24)=>DMem_to_RF_DUMMY(24), dout(23)=>DMem_to_RF_DUMMY(23),
dout(22)=>DMem_to_RF_DUMMY(22), dout(21)=>DMem_to_RF_DUMMY(21),
dout(20)=>DMem_to_RF_DUMMY(20), dout(19)=>DMem_to_RF_DUMMY(19),
dout(18)=>DMem_to_RF_DUMMY(18), dout(17)=>DMem_to_RF_DUMMY(17),
dout(16)=>DMem_to_RF_DUMMY(16), dout(15)=>DMem_to_RF_DUMMY(15),
dout(14)=>DMem_to_RF_DUMMY(14), dout(13)=>DMem_to_RF_DUMMY(13),
dout(12)=>DMem_to_RF_DUMMY(12), dout(11)=>DMem_to_RF_DUMMY(11),
dout(10)=>DMem_to_RF_DUMMY(10), dout(9)=>DMem_to_RF_DUMMY(9),
dout(8)=>DMem_to_RF_DUMMY(8), dout(7)=>DMem_to_RF_DUMMY(7),
dout(6)=>DMem_to_RF_DUMMY(6), dout(5)=>DMem_to_RF_DUMMY(5),
dout(4)=>DMem_to_RF_DUMMY(4), dout(3)=>DMem_to_RF_DUMMY(3),
dout(2)=>DMem_to_RF_DUMMY(2), dout(1)=>DMem_to_RF_DUMMY(1),
dout(0)=>DMem_to_RF_DUMMY(0));

truncator26to8bit : truncator26to8bits
PORT MAP (addr_in(25)=>Addr_26bits_DUMMY(25),
addr_in(24)=>Addr_26bits_DUMMY(24), addr_in(23)=>Addr_26bits_DUMMY(23),

```



```

addr_in(22)=>Addr_26bits_DUMMY(22), addr_in(21)=>Addr_26bits_DUMMY(21),
addr_in(20)=>Addr_26bits_DUMMY(20), addr_in(19)=>Addr_26bits_DUMMY(19),
addr_in(18)=>Addr_26bits_DUMMY(18), addr_in(17)=>Addr_26bits_DUMMY(17),
addr_in(16)=>Addr_26bits_DUMMY(16), addr_in(15)=>Addr_26bits_DUMMY(15),
addr_in(14)=>Addr_26bits_DUMMY(14), addr_in(13)=>Addr_26bits_DUMMY(13),
addr_in(12)=>Addr_26bits_DUMMY(12), addr_in(11)=>Addr_26bits_DUMMY(11),
addr_in(10)=>Addr_26bits_DUMMY(10), addr_in(9)=>Addr_26bits_DUMMY(9),
addr_in(8)=>Addr_26bits_DUMMY(8), addr_in(7)=>Addr_26bits_DUMMY(7),
addr_in(6)=>Addr_26bits_DUMMY(6), addr_in(5)=>Addr_26bits_DUMMY(5),
addr_in(4)=>Addr_26bits_DUMMY(4), addr_in(3)=>Addr_26bits_DUMMY(3),
addr_in(2)=>Addr_26bits_DUMMY(2), addr_in(1)=>Addr_26bits_DUMMY(1),
addr_in(0)=>Addr_26bits_DUMMY(0), addr_out(7)=>Addr_26to8bits_DUMMY(7),
addr_out(6)=>Addr_26to8bits_DUMMY(6),
addr_out(5)=>Addr_26to8bits_DUMMY(5),
addr_out(4)=>Addr_26to8bits_DUMMY(4),
addr_out(3)=>Addr_26to8bits_DUMMY(3),
addr_out(2)=>Addr_26to8bits_DUMMY(2),
addr_out(1)=>Addr_26to8bits_DUMMY(1),
addr_out(0)=>Addr_26to8bits_DUMMY(0);

```

END SCHEMATIC;

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the complete datapath was generated. Figure B.40 shows the resulting top level RTL symbol while figure B.41 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are covered in detail in Appendix A.

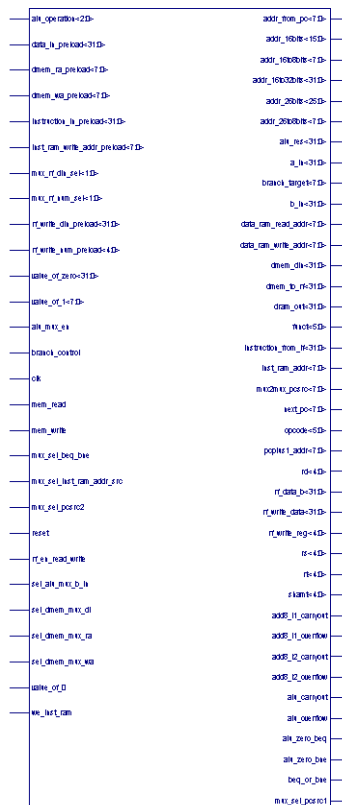


Figure B.40 Resulting top level RTL symbol for the synthesized complete datapath.

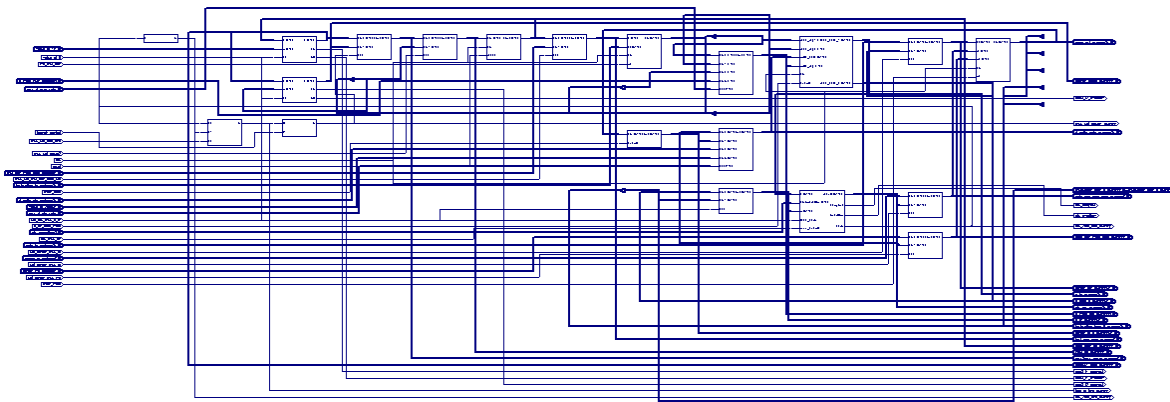


Figure B.41 Resulting top level RTL schematic for the synthesized complete datapath.

➤ FPGA Device Synthesis Summary

After the hardware implementation for this complete datapath 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                      : 677

Macro Statistics:
# RAM                      : 4
#   256x32-bit dual-port block RAM: 1
#   256x32-bit single-port block RAM: 1
#   32x32-bit dual-port block RAM: 2
# Registers                : 1
#   8-bit register         : 1
# Tristates                : 23
#   32-bit tristate buffer : 9
#   5-bit tristate buffer  : 4
#   8-bit tristate buffer  : 10

Cell Usage:
# BELS                    : 821
#   and2                  : 98
#   and2b1                : 33
#   and3                  : 64
#   and3b1                : 64
#   GND                   : 5
#   inv                   : 34
#   LUT1                  : 7
#   LUT1_L                : 65
#   LUT2                  : 10
#   LUT2_L                : 5
#   LUT3                  : 52
#   LUT3_D                : 2
#   LUT3_L                : 6
#   LUT4                  : 42
#   LUT4_D                : 20
#   LUT4_L                : 25
#   muxcy                 : 4
#   muxcy_d               : 2
#   muxcy_l               : 18
#   muxf5                 : 32
#   or2                   : 162
#   vcc                   : 3
#   xor2                  : 20
#   xor3                  : 32
#   xorcy                 : 16
# FlipFlops/Latches       : 8
```

```

#      FDC                      : 8
#      RAMS                     : 4
#      RAMB16_S36               : 1
#      RAMB16_S36_S36          : 3
#      Tri-States               : 388
#      BUFT                     : 388
#      Clock Buffers            : 1
#      BUFGP                    : 1
#      IO Buffers               : 676
#      IBUF                     : 187
#      OBUF                     : 457
#      OBUFT                    : 32
#      Logical                   : 8
#      nor4                     : 8
#      Others                   : 24
#      fmap                     : 24

```

Device utilization summary:

Number of Slices:	133	out of	46592	0%
Number of Slice Flip Flops:	8	out of	93184	0%
Number of 4 input LUTs:	234	out of	93184	0%
Number of bonded IOBs:	676	out of	1108	61%
Number of TBUFs:	388	out of	23296	1%
Number of BRAMs:	4	out of	168	2%
Number of GCLKs:	1	out of	16	6%

Timing Summary:

```

Minimum period: 35.403ns (Maximum Frequency: 28.246MHz)
Minimum input arrival time before clock: 36.644ns
Maximum output required time after clock: 41.140ns
Maximum combinational path delay: 42.381ns

```

➤ Place-and-Route onto the FPGA

In figure B.42, FPGA Editor shows the synthesized complete datapath after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the blue interconnections concentrated at the top half section of the FPGA chip.

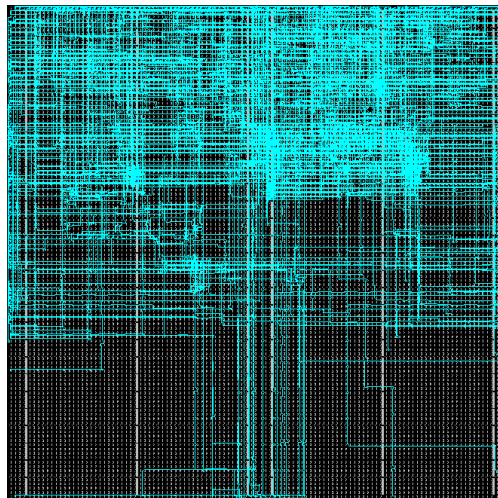


Figure B.42 FPGA Editor showing the synthesized complete datapath.

➤ *Simulation Results*

➤ *Simulation for AND*

Figure B.43 shows the simulation waveforms for ALU Operation = AND = (000)_{binary} = (0)_{decimal}. The instruction simulated is:

<u>AND</u>	\$R7 ,	\$R5 ,	\$R6
-----	-----	-----	-----
<i>rd</i>	<i>rs</i>	<i>rt</i>	

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	100100
-----	-----	-----	-----	-----	-----
<i>op=0</i>	<i>rs=\$R5</i>	<i>rt=\$R6</i>	<i>rd=\$R7</i>	<i>shamt</i>	<i>funct=36</i>

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000100100)_2 = (A63824)_{\text{hex}}$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

❑ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the first source register operand:

rs = \$R5

$[\$R5] = (0010110100100100111110110011001)_2 = (2D24FD99)_{\text{hex}}$

- Preload the instruction memory with the instruction to be simulated:

instruction_in_preload = (A63824)_{hex}

❑ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

rt = \$R6

$[\$R6] = (11110110011001001011010010010011)_2 = (F664B493)_{\text{hex}}$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

instruction_from_if = (A63824)_{hex}

❑ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

$ALU_Res(31:0) \Rightarrow RF_Write_Data(31:0) = A_in(31:0) \quad \underline{AND} \quad B_in(31:0)$

$$\begin{aligned}
 &= [\$R5] \quad \underline{\text{AND}} \quad [\$R6] \\
 &= (2D24FD99)_{\text{hex}} \underline{\text{AND}} \quad (F664B493)_{\text{hex}} \\
 &= (00101101001001001111110110011001)_2 \\
 &\quad \underline{\text{AND}} \\
 &\quad (11110110011001001011010010010011)_2 \\
 &= (00100100001001001011010010010001)_2 \\
 &= (2424B491)_{\text{hex}}
 \end{aligned}$$

▪ Timing Information:

- ♦ Delay between application of input signals and clock rising edge = 1 ns.
- ♦ On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
- ♦ Delay between application of input signals A_in and B_in to the ALU and the ALU result ($ALU_Res = RF_Write_Data$) stabilizing to the correct value = 0.6 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (00100100001001001011010010010001)_2 = (2424B491)_{\text{hex}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (00100100001001001011010010010001)_2 = (2424B491)_{\text{hex}}$$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

□ **Conclusions:**

- The execution of the AND instruction takes two clock cycles. These are clock cycles no. 2 and 3 in Figure B.43 below.
- Actually, total elapsed execution time is four clock cycles when including clock cycles no. 1 and 2, which are for the pre-loading phase.

- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.2 and prove that the complete datapath is functioning 100% as expected for an AND instruction.

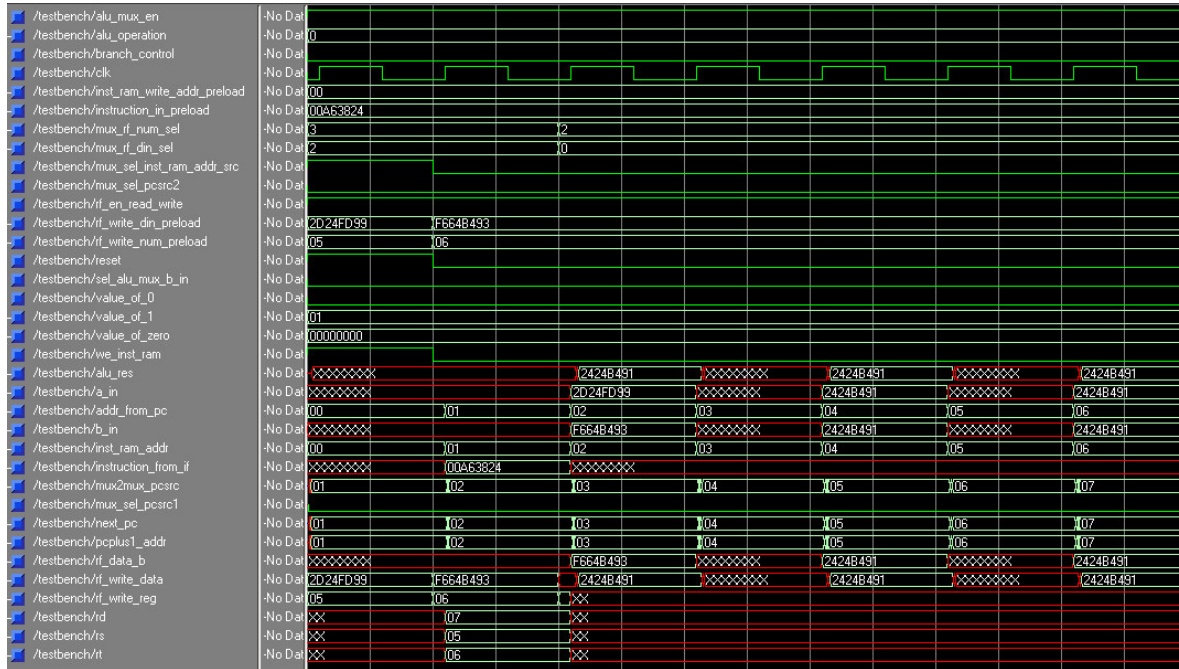


Figure B.43 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = AND.

➤ Simulation for OR

Figure B.44 shows the simulation waveforms for ALU Operation = OR = $(001)_{\text{binary}} = (1)_{\text{decimal}}$. The instruction simulated is:

<u>OR</u>	\$R7,	\$R5,	\$R6
----	----	----	----
rd	rs	rt	

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	100101
-----	-----	-----	-----	-----	-----
op=0	rs=\$R5	rt=\$R6	rd=\$R7	shamt	funct=37

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000100101)_2 = (A63825)_{\text{hex}}$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

□ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the first source register operand:

$$rs = \$R5$$

$$[\$R5] = (00101101001001001111110110011001)_2 = (2D24FD99)_{\text{hex}}$$

- Preload the instruction memory with the instruction to be simulated:

$$\text{instruction_in_preload} = (A63825)_{\text{hex}}$$

□ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (11110110011001001011010010010011)_2 = (F664B493)_{\text{hex}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$\text{instruction_from_if} = (A63825)_{\text{hex}}$$

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

$$\begin{aligned} \text{ALU_Res}(31:0) \Rightarrow \text{RF_Write_Data}(31:0) &= A_in(31:0) \quad \underline{\text{OR}} \quad B_in(31:0) \\ &= [\$R5] \quad \underline{\text{OR}} \quad [\$R6] \\ &= (2D24FD99)_{\text{hex}} \quad \underline{\text{OR}} \quad (F664B493)_{\text{hex}} \\ &= (00101101001001001111110110011001)_2 \\ &\quad \underline{\text{OR}} \\ &\quad (11110110011001001011010010010011)_2 \\ &= (11111111011001001111110110011011)_2 \\ &= (FF64FD9B)_{\text{hex}} \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals A_in and B_in to the ALU and the ALU result ($\text{ALU_Res} = \text{RF_Write_Data}$) stabilizing to the correct value = 0.6 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$rd = \$R7$

$[\$R7] = (11111111011001001111110110011011)_2 = (FF64FD9B)_{hex}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$rd = \$R7$

$[\$R7] = (11111111011001001111110110011011)_2 = (FF64FD9B)_{hex}$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

□ **Conclusions:**

- The execution of the OR instruction takes two clock cycles. These are clock cycles no. 2 and 3 in Figure B.44 below.
- Actually, total elapsed execution time is four clock cycles when including clock cycles no. 1 and 2, which are for the pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.2 and prove that the complete datapath is functioning 100% as expected for an OR instruction.

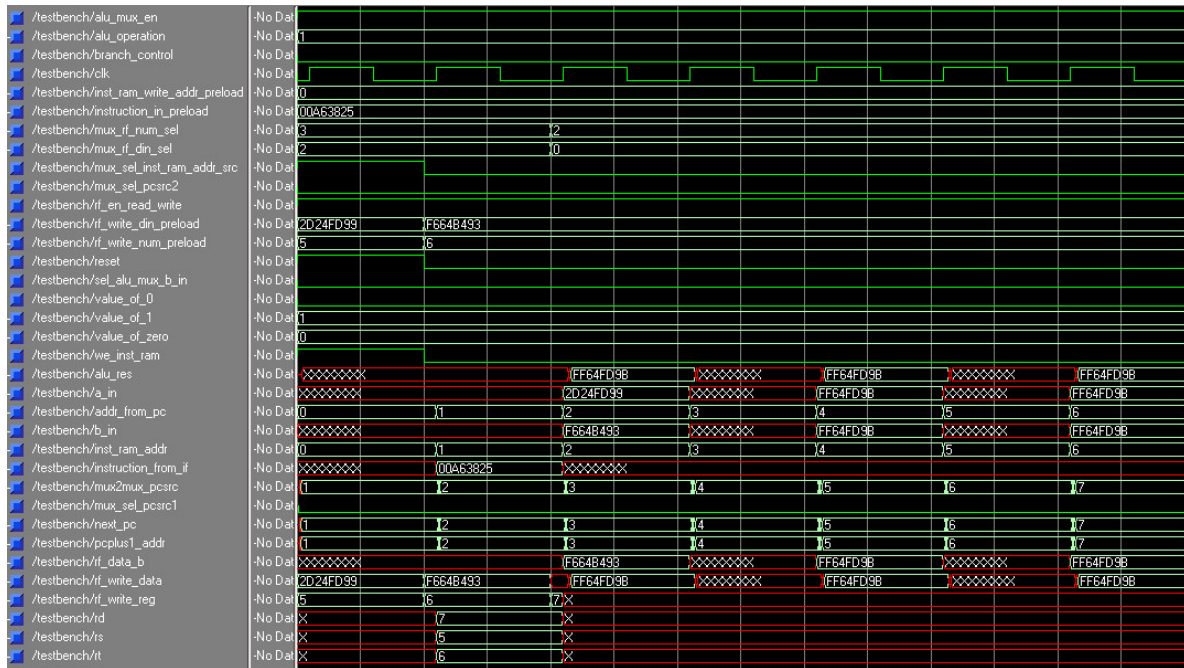


Figure B.44 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = OR.

➤ Simulation for ADD

Figure B.45 shows the simulation waveforms for ALU Operation = ADD = $(010)_{\text{binary}} = (2)_{\text{decimal}}$. The instruction simulated is:

<u>ADD</u>	$\$R7$,	$\$R5$,	$\$R6$
-----	-----	-----	-----
<i>rd</i>	<i>rs</i>	<i>rt</i>	

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	100000
-----	-----	-----	-----	-----	-----
<i>op</i> =0	<i>rs</i> =\$R5	<i>rt</i> =\$R6	<i>rd</i> =\$R7	<i>shamt</i>	<i>funct</i> =32

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000100000)_2 = (A63820)_{\text{hex}}$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

❑ During clock cycle 1 (Pre-load Phase):

- Preload the register file with the first source register operand:

rs = \$R5

$$[\$R5] = (15)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$\text{instruction_in_preload} = (\text{A63820})_{\text{hex}}$$

□ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (16)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$\text{instruction_from_if} = (\text{A63820})_{\text{hex}}$$

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

$$\begin{aligned} \text{ALU_Res}(31:0) => \text{RF_Write_Data}(31:0) &= A_in(31:0) + B_in(31:0) \\ &= [\$R5] + [\$R6] \\ &= (15)_{\text{decimal}} + (16)_{\text{decimal}} \\ &= (31)_{\text{decimal}} \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals A_in and B_in to the ALU and the ALU result ($\text{ALU_Res} = \text{RF_Write_Data}$) stabilizing to the correct value = 0.7 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (31)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$rd = \$R7$

$[\$R7] = (31)_{\text{decimal}}$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

□ Conclusions:

- The execution of the ADD instruction takes two clock cycles. These are clock cycles no. 2 and 3 in Figure B.45 below.
- Actually, total elapsed execution time is four clock cycles when including clock cycles no. 1 and 2, which are for the pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.2 and prove that the complete datapath is functioning 100% as expected for an ADD instruction.

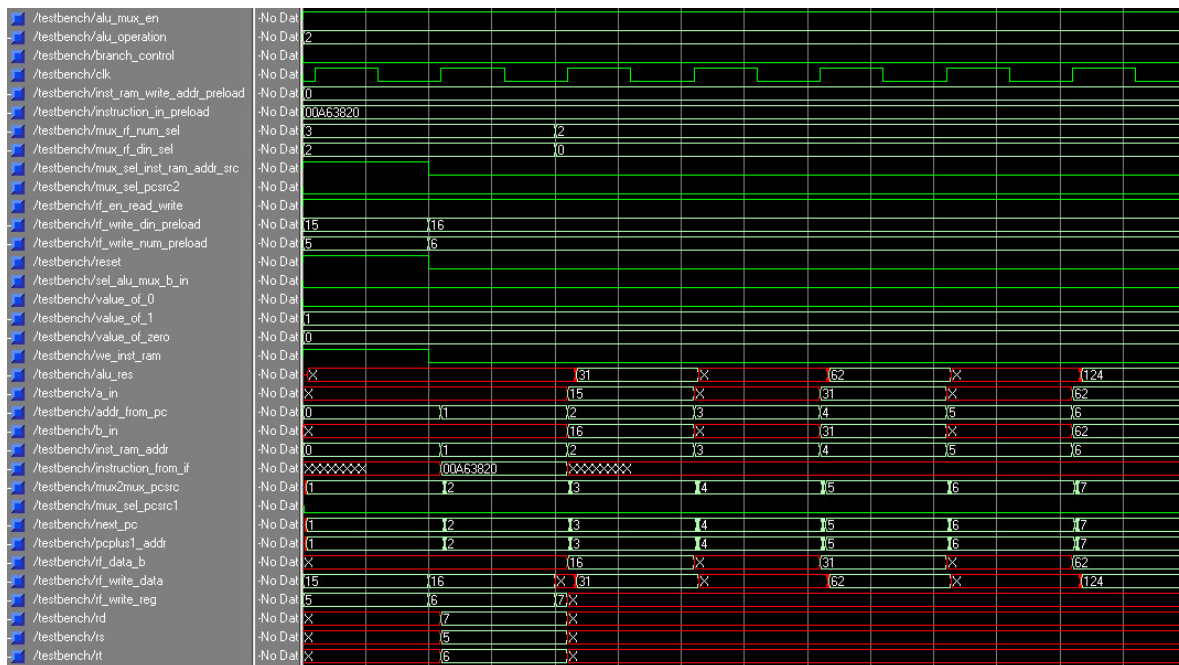


Figure B.45 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = ADD.

➤ Simulation for SUB

Figure B.46 shows the simulation waveforms for ALU Operation = SUB = $(110)_{\text{binary}} = (6)_{\text{decimal}}$. The instruction simulated is:

<u>SUB</u>	\$R7 ,	\$R5 ,	\$R6
-----	-----	-----	-----
<i>rd</i>	<i>rs</i>	<i>rt</i>	

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	100010
-----	-----	-----	-----	-----	-----
<i>op</i> =0	<i>rs</i> =\$R5	<i>rt</i> =\$R6	<i>rd</i> =\$R7	<i>shamt</i>	<i>funct</i> =34

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00000000101001100011100000100010)_2 = (A63822)_{\text{hex}}$$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

❑ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the first source register operand:

$$rs = \$R5$$

$$[\$R5] = (15)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$instruction_in_preload = (A63822)_{\text{hex}}$$

❑ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (16)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$instruction_from_if = (A63822)_{\text{hex}}$$

❑ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

$$ALU_Res(31:0) \Rightarrow RF_Write_Data(31:0) = A_in(31:0) - B_in(31:0)$$

$$= [\$R5] - [\$R6]$$

$$= (15)_{\text{decimal}} - (16)_{\text{decimal}}$$

$$= (-1)_{\text{decimal}}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
- ◆ Delay between application of input signals A_in and B_in to the ALU and the ALU result ($ALU_Res = RF_Write_Data$) stabilizing to the correct value = 0.8 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$rd = \$R7$

$[\$R7] = (-1)_{\text{decimal}}$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$rd = \$R7$

$[\$R7] = (-1)_{\text{decimal}}$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

□ **Conclusions:**

- The execution of the SUB instruction takes two clock cycles. These are clock cycles no. 2 and 3 in Figure B.46 below.
- Actually, total elapsed execution time is four clock cycles when including clock cycles no. 1 and 2, which are for the pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.2 and prove that the complete datapath is functioning 100% as expected for a SUB instruction.

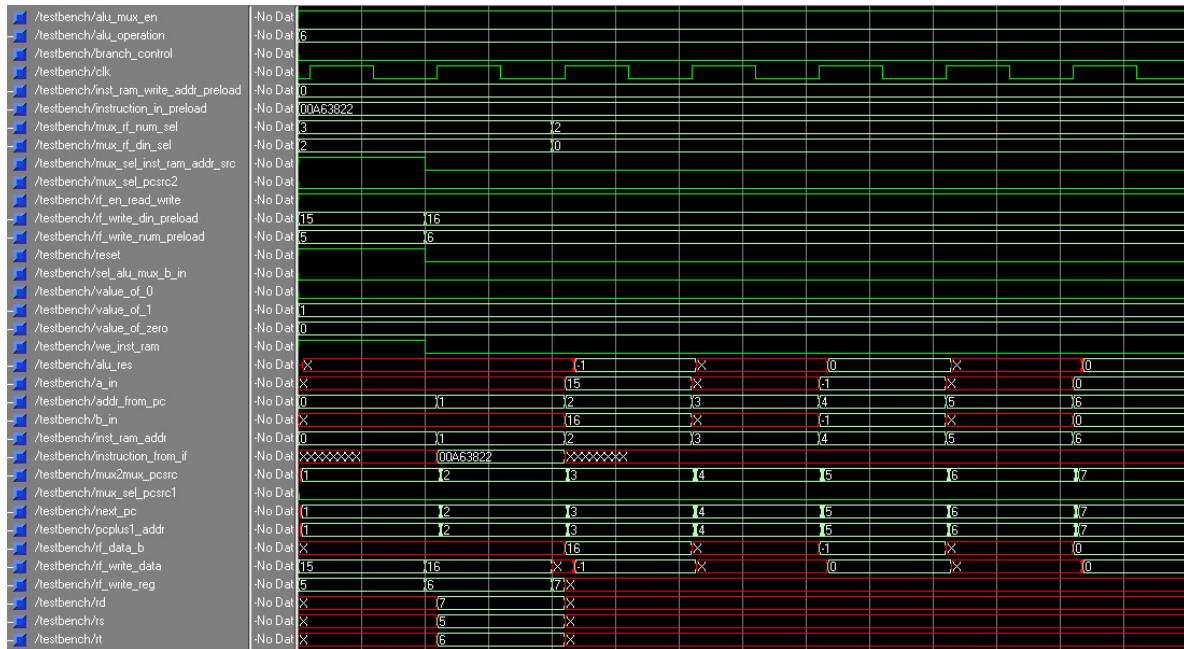


Figure B.46 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = SUB.

➤ Simulation for SLT

Figures B.47 to B.49 show the simulation waveforms for ALU Operation = SLT = $(111)_{\text{binary}} = (7)_{\text{decimal}}$ for the three conditions $[rs] < [rt]$, $[rs] = [rt]$, and $[rs] > [rt]$, respectively. The instruction simulated is:

<u>SLT</u>	\$R7 ,	\$R5 ,	\$R6
	-----	-----	-----
	rd	rs	rt

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	101010
-----	-----	-----	-----	-----	-----
op=0	rs=\$R5	rt=\$R6	rd=\$R7	shamt	funct=42

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000101010)_2 = (A6382A)_{\text{hex}}$

❖ Condition 1 (Figure B.47): When $[rs] < [rt]$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

□ During clock cycle 1 (Pre-load Phase):

- Preload the register file with the first source register operand:

$rs = \$R5$

$$[\$R5] = (15)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$\text{instruction_in_preload} = (A6382A)_{\text{hex}}$$

□ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (16)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$\text{instruction_from_if} = (A6382A)_{\text{hex}}$$

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

Since

$$(15)_{\text{decimal}} < (16)_{\text{decimal}}$$

i.e.

$$[\$R5] < [\$R6]$$

Then

$$ALU_Res(31:0) \Rightarrow RF_Write_Data(31:0) = (1)_{\text{decimal}}$$

- Timing Information:
 - Delay between application of input signals and clock rising edge = 1 ns.
 - On this clock rising edge, both source register operands represented by the intermediate signals *A_in* and *B_in* are read out from the RF and fed into the ALU inputs.
 - Delay between application of input signals *A_in* and *B_in* to the ALU and the ALU result (*ALU_Res* = *RF_Write_Data*) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (1)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ During clock cycle 5 (Debugging Phase):

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (1)_{\text{decimal}}$$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

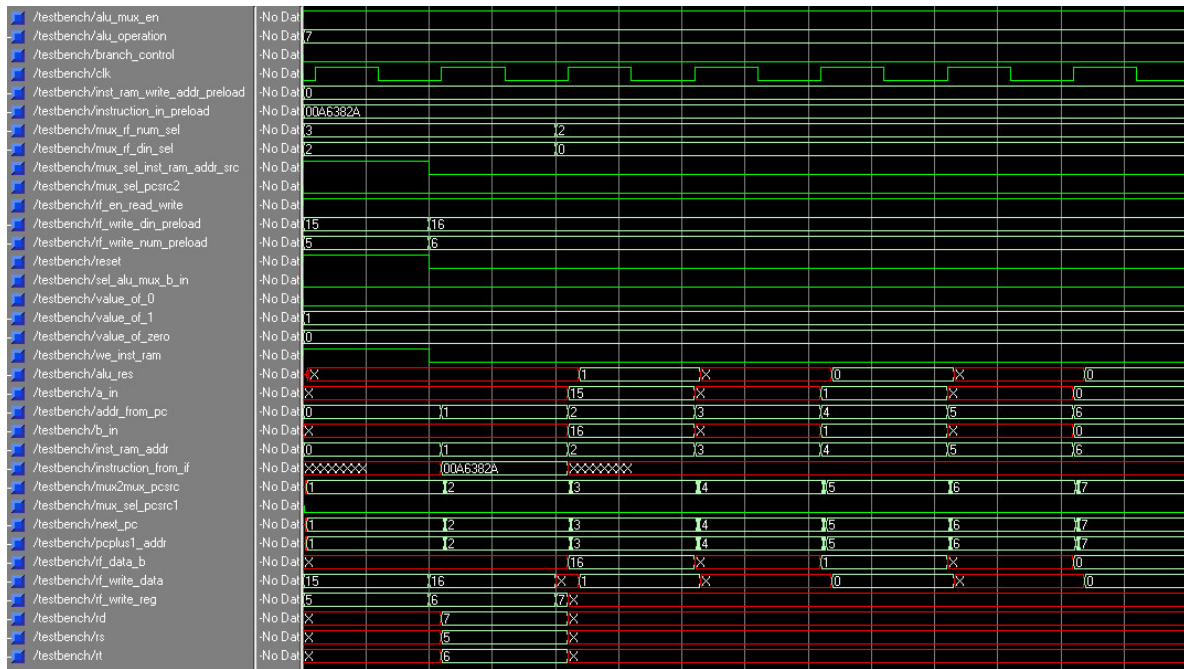


Figure B.47 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = SLT for the condition $[rs] < [rt]$.

❖ Condition 2 (Figure B.48): When $[rs] = [rt]$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

□ During clock cycle 1 (Pre-load Phase):

- Preload the register file with the first source register operand:

$$rs = \$R5$$

$$[\$R5] = (15)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$instruction_in_preload = (A6382A)_{\text{hex}}$$

□ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (15)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$instruction_from_if = (A6382A)_{\text{hex}}$$

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

Since

$$(15)_{\text{decimal}} = (15)_{\text{decimal}}$$

i.e.

$$[\$R5] = [\$R6]$$

Then

$$ALU_Res(31:0) \Rightarrow RF_Write_Data(31:0) = (0)_{\text{decimal}}$$

- Timing Information:
 - Delay between application of input signals and clock rising edge = 1 ns.
 - On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
 - Delay between application of input signals A_in and B_in to the ALU and the ALU result ($ALU_Res = RF_Write_Data$) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (0)_{\text{decimal}}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (0)_{\text{decimal}}$$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

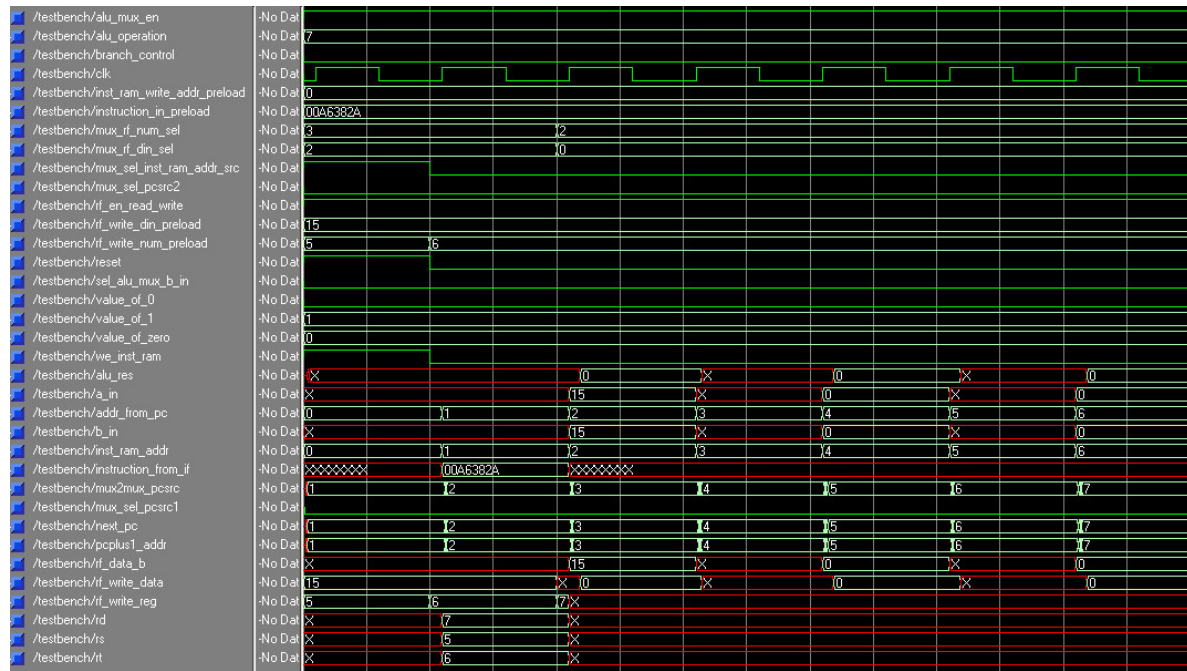


Figure B.48 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = SLT for the condition $[rs] = [rt]$.

❖ **Condition 3 (Figure B.49): When $[rs] > [rt]$**

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

□ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the first source register operand:

$$rs = \$R5$$

$$[\$R5] = (16)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$instruction_in_preload = (A6382A)_{\text{hex}}$$

□ **During clock cycle 2 (Pre-load Phase):**

- Preload the register file with the second source register operand:

$$rt = \$R6$$

$$[\$R6] = (15)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$instruction_from_if = (A6382A)_{hex}$$

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU calculates the result value:

Since

$$(16)_{decimal} > (15)_{decimal}$$

i.e.

$$[\$R5] > [\$R6]$$

Then

$$ALU_Res(31:0) => RF_Write_Data(31:0) = (0)_{decimal}$$

- Timing Information:
 - Delay between application of input signals and clock rising edge = 1 ns.
 - On this clock rising edge, both source register operands represented by the intermediate signals A_in and B_in are read out from the RF and fed into the ALU inputs.
 - Delay between application of input signals A_in and B_in to the ALU and the ALU result ($ALU_Res = RF_Write_Data$) stabilizing to the correct value = 0.9 ns. This delay is exactly in line with what is discussed in detail in Appendix A.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The ALU result value is written back into the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (0)_{decimal}$$

- This value just written cannot be read out in this clock cycle, but only in the next one, due to the synchronous read/write operation of the RF (as detailed in Appendix A).

□ **During clock cycle 5 (Debugging Phase):**

- The signals A_in and B_in are inspected for debugging purposes, which confirm that the data value just stored in (loaded into) destination register rd is read out from the RF (on the rising clock edge):

$$rd = \$R7$$

$$[\$R7] = (0)_{decimal}$$

- The ALU calculates the next result value, which will be stored in the RF during the next clock cycle, and so on.

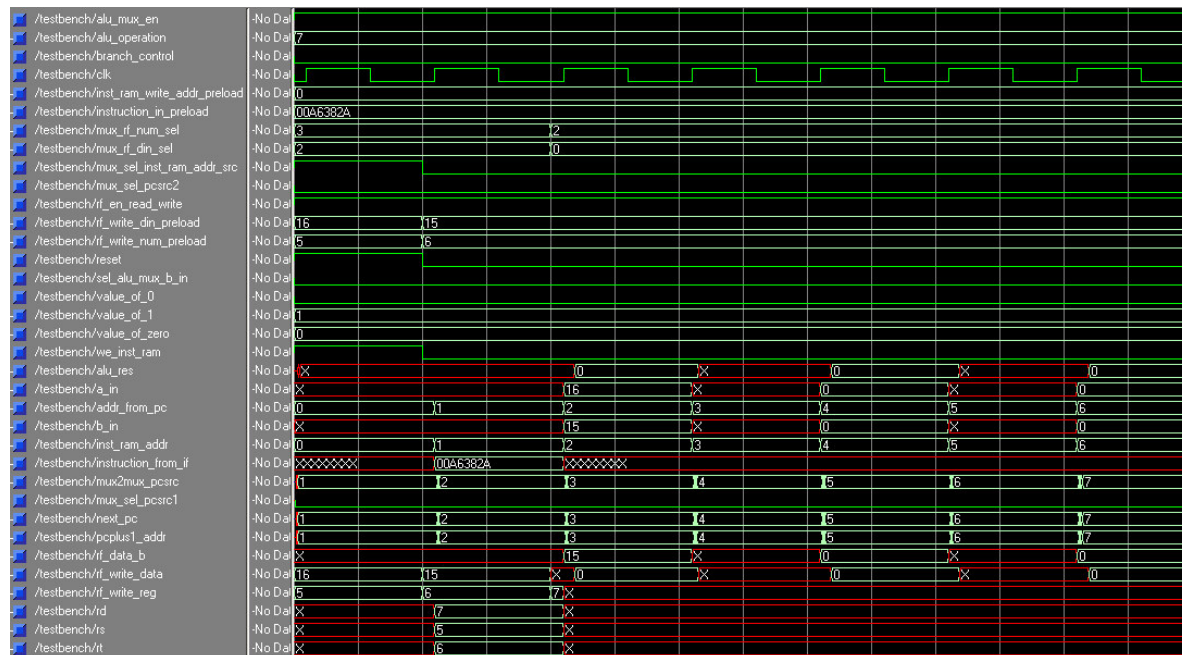


Figure B.49 Results of simulating the synthesized complete datapath, using ModelSim, with ALU Operation = SLT for the condition $[rs] > [rt]$.

❑ Conclusions:

- The execution of the SLT instruction takes two clock cycles. These are clock cycles no. 2 and 3 in Figures B.47 to B.49 below.
- Actually, total elapsed execution time is four clock cycles when including clock cycles no. 1 and 2, which are for the pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.2 and prove that the complete datapath is functioning 100% as expected for an SLT instruction.

➤ Simulation for LW

Figure B.50 shows the simulation waveforms for LW by selecting ALU Operation = ADD = $(010)_{\text{binary}} = (2)_{\text{decimal}}$. There is no ALU Operation specifically for LW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address from which to load the data.

The instruction simulated is:

<u>LW</u>	\$R6 ,	10	(\$R5)
	-----	-----	
	<i>rt</i>	<i>offset</i>	<i>(rs)</i>

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

```

100011 00101 00110 0000000000001010
-----
op=35  rs=$R5  rt=$R6  offset=10

```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101001100000000000001010)_2 = (8CA6000A)_{\text{hex}}$$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

□ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the base register operand:

$$rs \Rightarrow RF_Write_Reg = \$R5, \quad [\$R5] = (15)_{\text{decimal}}$$

- Preload the data memory with the data value:

$$Target\ Memory\ Location = 25, \quad [Memory[25]] = (5678)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$instruction_in_preload = (8CA6000A)_{\text{hex}}$$

□ **During clock cycle 2 (Instruction Execution Phase):**

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$instruction_from_if = (8CA6000A)_{\text{hex}}$$

- The ALU calculates the result value = target memory address:

$$\begin{aligned}
 Result(31:0) \Rightarrow ALU_Res(31:0) &= A_in(31:0) && + B_in(31:0) \\
 &= [read_reg1(4:0)] && + Addr_16to32bits \\
 &= [\$R5] && + Addr_16to32bits \\
 &= (15)_{\text{decimal}} && + (10)_{\text{decimal}} \\
 &= (25)_{\text{decimal}}
 \end{aligned}$$

- Timing Information:

- ◆ Delay between application of input signals and clock rising edge = 1 ns.
- ◆ On this clock rising edge, the first source register operand represented by the intermediate signals a_in is read out from the RF and fed into the A input of the ALU.
- ◆ Delay between application of the signal a_in to the ALU and the ALU result ($result(31:0) = alu_res(31:0) = Data_RAM_Read_Addr(7:0)$) stabilizing to the correct value = 0.7 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

□ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU result value $result(31:0) \Rightarrow alu_res(31:0)$ which is the target memory address is passed through the multiplexer mux8b_2to1 (Dmem_Mux_ra) and gets truncated from 32

bits down to 8 bits in order to drive the read address $ra(7:0)$ input of the data memory which then reads out the value from the target memory address of 25:

$alu_res(31:0) \Rightarrow ra(7:0) = (25)_{decimal}$

- This yields the data output $do(31:0)$ from the Data_RAM which passes through the tri-state buffer then mux32b_4to1 before it is written into the destination register rt inside the register file.

□ **During clock cycle 4 (Instruction Execution Phase):**

- The destination register rt inside the register file is loaded (written) with the value just read from memory in cycle 3:

$rt \Rightarrow RF_Write_Reg = \$R6$, $[\$R6] = (5678)_{decimal}$

- Important Note: The register file could not be written in clock cycle 3 since the data to be written was not available until exactly the rising clock edge. This condition did not allow the register file the required and sufficient setup time of 1 ns prior to the rising clock edge, and therefore had to wait until the next rising clock edge to perform the write into the register file.

□ **During clock cycle 5 (Debugging Phase):**

- The signal rf_data_b is inspected for debugging purposes, which confirms that the data value just stored in (loaded into) destination register rt is read out from the RF (on the rising clock edge):

$rt \Rightarrow read_reg2 = \$R6$, $[\$R6] = (5678)_{decimal}$

□ **Conclusions:**

- The execution of the LW instruction takes 3 clock cycles. These are clock cycles no. 2, 3, and 4 in Figure B.50
- Actually, total elapsed execution time is 4 clock cycles when including clock cycle no. 1, which is for the pre-loading phase.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.
- These results are exactly in line with those for section B.3.3 and prove that the complete datapath is functioning 100% as expected for a LW instruction.

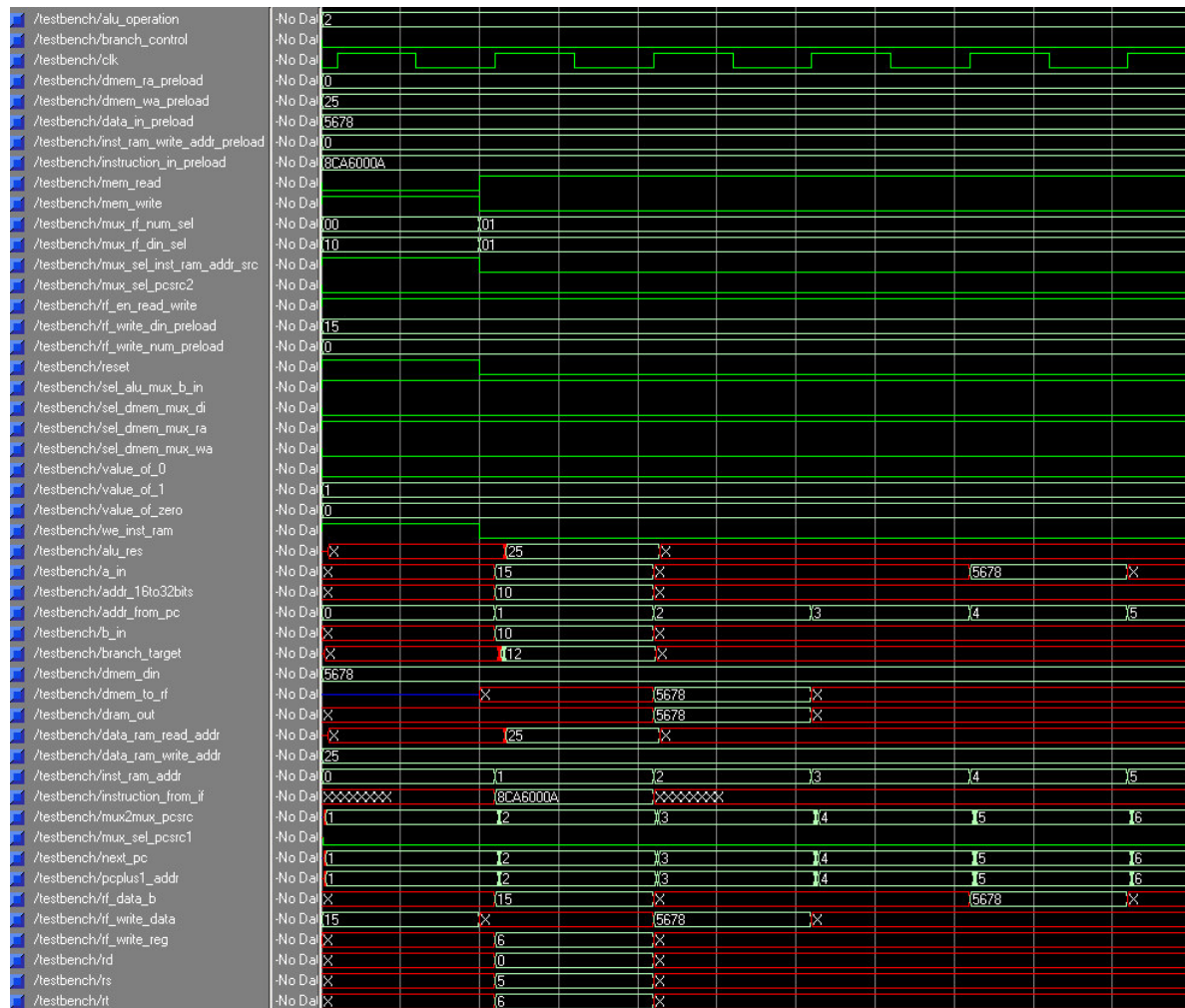


Figure B.50 Results of simulating the synthesized datapath section for LW instruction, using ModelSim, with ALU Operation = ADD.

➤ Simulation for SW

Figure B.51 shows the simulation waveforms for SW by selecting ALU Operation = ADD = (010)_{binary} = (2)_{decimal}. Similar to LW, there is no ALU Operation specifically for SW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address for storing the data.

The instruction simulated is:

<u>SW</u>	\$R6 ,	10	(\$R5)
	-----	-----	
	rt	offset	(rs)

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

101011	00101	00110	0000000000001010
-----	-----	-----	-----
op=43	rs=\$R5	rt=\$R6	offset=10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10101100101001100000000000001010)_2 = (ACA6000A)_{\text{hex}}$$

Following is the description and analysis of the operations taking place during each of the 5 clock cycles:

❑ **During clock cycle 1 (Pre-load Phase):**

- Preload the register file with the base register operand:

$$rs \Rightarrow RF_Write_Reg = \$R5 \quad , \quad [\$R5] = (15)_{\text{decimal}}$$

- Preload the instruction memory with the instruction to be simulated:

$$instruction_in_preload = (ACA6000A)_{\text{hex}}$$

❑ **During clock cycle 2 (Pre-load Phase & Instruction Execution Phase):**

- Preload the register file with the data value which is to be stored into the memory later:

$$rt \Rightarrow write_reg = \$R6 \quad , \quad [\$R6] = (5678)_{\text{decimal}}$$

- The instruction preloaded into the instruction memory in the previous clock cycle is now read out:

$$instruction_from_if = (ACA6000A)_{\text{hex}}$$

- The ALU calculates the result value = target memory address:

$$\begin{aligned} Result(31:0) \Rightarrow ALU_Res(31:0) &= A_in(31:0) && + B_in(31:0) \\ &= [read_reg1(4:0)] && + Addr_16to32bits \\ &= [\$R5] && + Addr_16to32bits \\ &= (15)_{\text{decimal}} && + (10)_{\text{decimal}} \\ &= (25)_{\text{decimal}} \end{aligned}$$

- Timing Information:

- ♦ Delay between application of input signals and clock rising edge = 1 ns.
- ♦ On this clock rising edge, the first source register operand represented by the intermediate signals a_in is read out from the RF and fed into the A input of the ALU.
- ♦ Delay between application of the signal a_in to the ALU and the ALU result ($result(31:0) = alu_res(31:0) = Data_RAM_Read_Addr(7:0)$) stabilizing to the correct value = 0.7 ns. This delay is exactly in line with what is discussed in detail in Appendix A as the ALU used here is based on carry lookahead.

❑ **During clock cycle 3 (Instruction Execution Phase):**

- The ALU result value $result(31:0) \Rightarrow alu_res(31:0)$ which is the target memory address is passed through the multiplexer mux8b_2to1 (Dmem_Mux_wa) and gets truncated from 32 bits down to 8 bits in order to drive the write address $wa(7:0)$ input of the data memory:

$$alu_res(31:0) \Rightarrow ra(7:0) = (25)_{\text{decimal}}$$

- Simultaneously, the data value (register operand) in source register *rt* is read out from the register file and is available and ready to be written (stored) into the target memory address 25 in the data memory:

$rt \Rightarrow read_reg2 = \$R6$, $[\$R6] = (5678)_{decimal}$

- $read_data2 \Rightarrow RF_Data_B \Rightarrow Dmem_Din \Rightarrow di = (5678)_{decimal}$

□ **During clock cycle 4 (Instruction Execution Phase):**

- The data value (register operand) in source register *rt* just read out from the register file (in the previous clock cycle) is now written (stored) into the target memory address 25 in the data memory:

$Target\ Memory\ Location = 25$, $[Memory[25]] = (5678)_{decimal}$

□ **During clock cycle 5 (Debugging Phase):**

- The signal *dram_out* is inspected for debugging purposes, which should confirm that the data value just stored in (loaded into) the target memory address 25 is read out from the data memory (on the rising clock edge):

$Target\ Memory\ Location = 25$, $[Memory[25]] = (5678)_{decimal}$

However, this is not the case! As seen in Figure B.51, the signal *dram_out* is not yielding the expected correct output value. By further inspecting the waveforms in Figure B.51, it is evident that the SW instruction does not execute properly on this datapath. This is because after the instruction is read out from the instruction memory, it is available only for one clock cycle, and then disappears during the next cycle (during which time the next instruction is being read out from the instruction memory). This availability during one clock cycle only is not long enough for the instruction to execute to completion. The rectification to address this problem will be discussed very shortly.

□ **Conclusions:**

- Similar to LW, the execution of the SW instruction takes exactly 3 clock cycles. These are clock cycles no. 2, 3, and 4 in Figure B.51.
- Actually, total elapsed execution time is 4 clock cycles when including clock cycle no. 1, which is for the first part of the 2-cycle data pre-loading phase.
- Therefore, it is worth noting that in the case of SW, the end of the data pre-loading phase and the start of the instruction execution over-lap in clock cycle no. 2. This is shown in Figure B.51 where the data pre-loading phase is during the first 2 clock cycles (consecutively pre-load *rs* then *rt* into RF), while the instruction execution starts at clock cycle no. 2 during which time the ALU already calculates and generates the output result *alu_res*.
- However, in either case, this is not inclusive of clock cycle no. 5, which is purely for debugging purposes to confirm that the instruction has executed correctly.

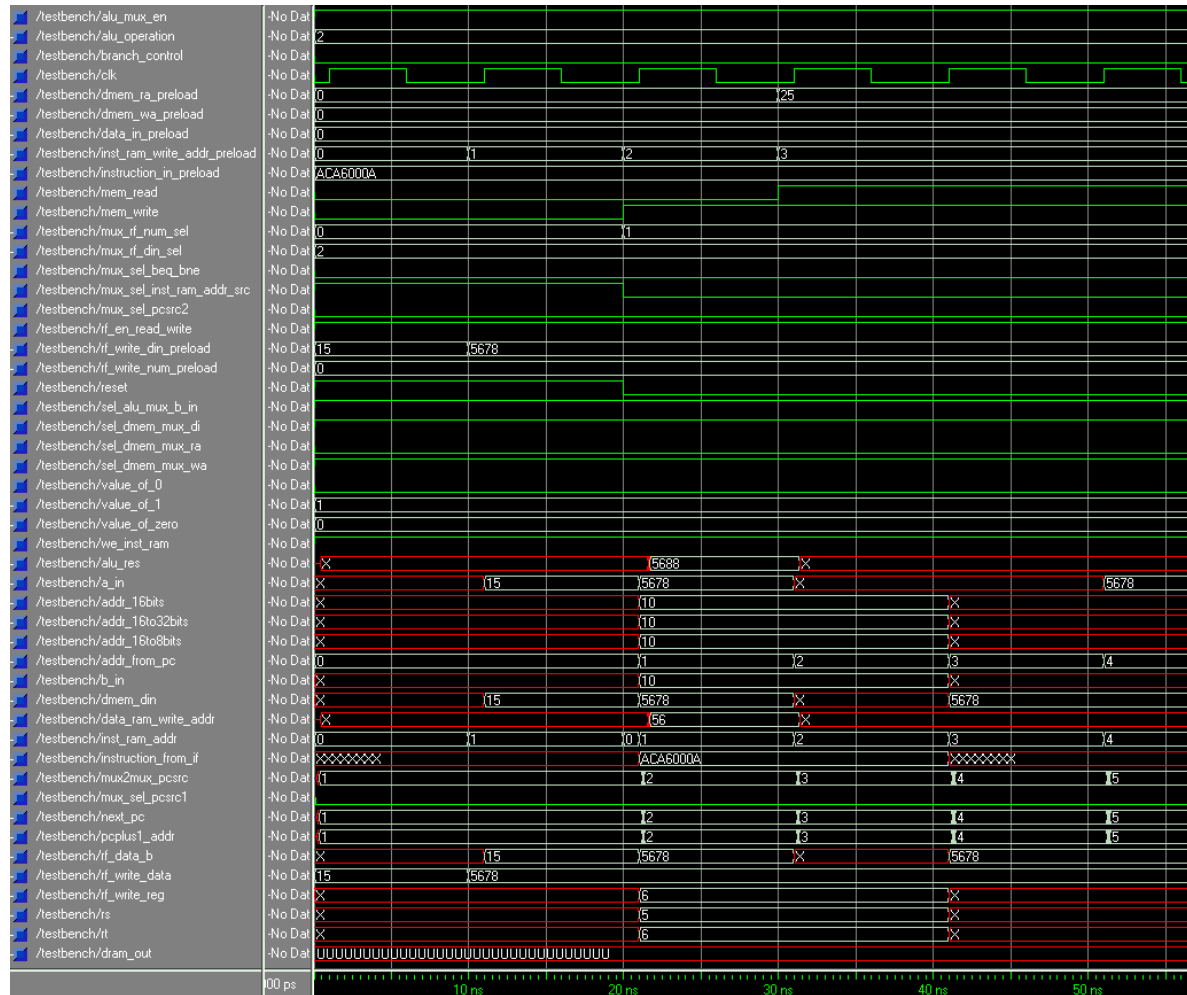


Figure B.51 Results of simulating the synthesized datapath section for SW instruction, using ModelSim, with ALU Operation = ADD.

The ramification of the fact that the SW instruction does not execute properly on this datapath, is that this datapath has to be modified in either one of these following two solutions to remedy this problem:

- **Solution One:** Inserting pipeline registers that would hold the instruction (and any accompanying control signals generated by the instruction) for a number of clock cycles sufficient enough for the SW instruction to execute correctly to completion.

OR

- **Solution Two:** Slowing down the clock which drives both the program counter and instruction memory while keeping the master (original) clock driving the register file and data memory.

Solution Two was chosen due to its relative simplicity (compared to the first one) and the fact of limited time until the deadline for the submission of this thesis. The next section elaborates on the implementation of this solution chosen.

B.5 Putting It All Together: The Complete Datapath with DCM

➤ RTL Description

The complete datapath with DCM (Digital Clock Manager) is constructed by combining the complete datapath discussed in the last section (section B.4) together with the DCM discussed in Appendix A.

As seen when simulating the SW instruction in the last section, there is a need for the inclusion of the DCM to divide the master clock and feed the resulting different clock frequencies to different sections of the datapath. As shown in Figure B.52, the implementation utilized here is to divide the master clock *Clkin* by a factor of 5 to generate the output clock *ClkDv*.

➤ Design Entry and Synthesis

Schematic Editor was used to create the design entry for the complete datapath with DCM. This is shown in Figure B.52 below.

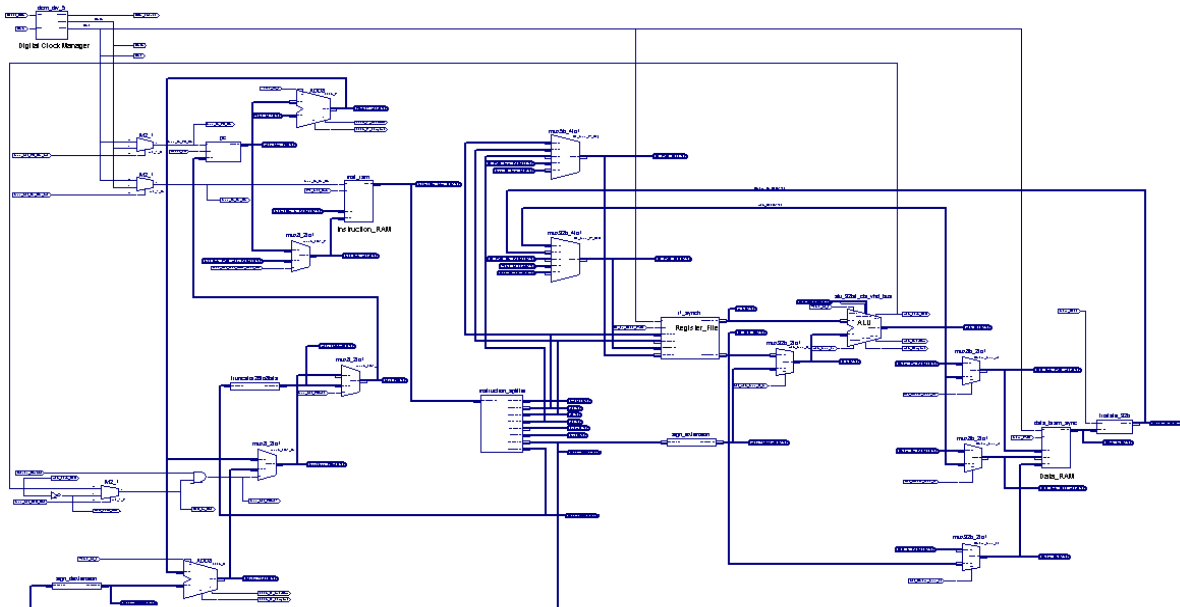


Figure B.52 Schematic diagram design entry in Schematic Editor for the complete datapath with DCM.

After synthesis of the schematic diagram in figure B.52 using XST, the following VHDL code was generated:

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1.sch - Thu Oct
19 08:17:45 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 M2_1_MXILINX_complete_datapath_w_dcm_div_5 IS
    PORT ( D0      :      IN      STD_LOGIC;
```

```

        D1      :      IN      STD_LOGIC;
        S0      :      IN      STD_LOGIC;
        O :      OUT      STD_LOGIC);

end M2_1_MXILINX_complete_datapath_w_dcm_div_5;

ARCHITECTURE SCHEMATIC OF M2_1_MXILINX_complete_datapath_w_dcm_div_5 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;

-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\add8.sch - Thu Oct
19 08:17:45 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 ADD8_MXILINX_complete_datapath_w_dcm_div_5 IS
    PORT ( A :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          B :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          CI :      IN      STD_LOGIC;
          CO :      OUT     STD_LOGIC;
          OFL :      OUT     STD_LOGIC;
          S :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));

end ADD8_MXILINX_complete_datapath_w_dcm_div_5;

ARCHITECTURE SCHEMATIC OF ADD8_MXILINX_complete_datapath_w_dcm_div_5 IS
    SIGNAL C0      :      STD_LOGIC;
    SIGNAL C1      :      STD_LOGIC;
    SIGNAL C2      :      STD_LOGIC;

```

```

SIGNAL C3      :      STD_LOGIC;
SIGNAL C4      :      STD_LOGIC;
SIGNAL C5      :      STD_LOGIC;
SIGNAL C6      :      STD_LOGIC;
SIGNAL C60     :      STD_LOGIC;
SIGNAL CO_DUMMY :      STD_LOGIC;
SIGNAL I0      :      STD_LOGIC;
SIGNAL I1      :      STD_LOGIC;
SIGNAL I2      :      STD_LOGIC;
SIGNAL I3      :      STD_LOGIC;
SIGNAL I4      :      STD_LOGIC;
SIGNAL I5      :      STD_LOGIC;
SIGNAL I6      :      STD_LOGIC;
SIGNAL I7      :      STD_LOGIC;
SIGNAL dummy   :      STD_LOGIC;

ATTRIBUTE BOX_TYPE : STRING;
ATTRIBUTE RLOC : STRING ;
ATTRIBUTE RLOC OF I_36_16 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_17 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_23 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_22 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_18 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_19 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_20 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_21 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_64 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_107 : LABEL IS "X0Y3";
ATTRIBUTE RLOC OF I_36_110 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_63 : LABEL IS "X0Y2";
ATTRIBUTE RLOC OF I_36_58 : LABEL IS "X0Y1";
ATTRIBUTE RLOC OF I_36_111 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_55 : LABEL IS "X0Y0";
ATTRIBUTE RLOC OF I_36_62 : LABEL IS "X0Y1";

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 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_D
  PORT ( CI      :      IN      STD_LOGIC;
         DI      :      IN      STD_LOGIC;
         S       :      IN      STD_LOGIC;
         LO      :      OUT     STD_LOGIC;
         O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF MUXCY_D : 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 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";
COMPONENT XORCY
  PORT ( CI      :      IN      STD_LOGIC;
        LI      :      IN      STD_LOGIC;
        O       :      OUT     STD_LOGIC);
END COMPONENT;

ATTRIBUTE BOX_TYPE OF XORCY : COMPONENT IS "BLACK_BOX";
BEGIN
  CO <= CO_DUMMY;

  I_36_16 : FMAP
    PORT MAP (I1=>A(0), I2=>B(0), I3=>dummy, I4=>dummy, O=>I0);

  I_36_17 : FMAP
    PORT MAP (I1=>A(1), I2=>B(1), I3=>dummy, I4=>dummy, O=>I1);

  I_36_23 : FMAP
    PORT MAP (I1=>A(7), I2=>B(7), I3=>dummy, I4=>dummy, O=>I7);

  I_36_22 : FMAP
    PORT MAP (I1=>A(6), I2=>B(6), I3=>dummy, I4=>dummy, O=>I6);

  I_36_18 : FMAP
    PORT MAP (I1=>A(2), I2=>B(2), I3=>dummy, I4=>dummy, O=>I2);

  I_36_19 : FMAP
    PORT MAP (I1=>A(3), I2=>B(3), I3=>dummy, I4=>dummy, O=>I3);

  I_36_20 : FMAP
    PORT MAP (I1=>A(4), I2=>B(4), I3=>dummy, I4=>dummy, O=>I4);

  I_36_21 : FMAP
    PORT MAP (I1=>A(5), I2=>B(5), I3=>dummy, I4=>dummy, O=>I5);

  I_36_64 : MUXCY
    PORT MAP (CI=>C6, DI=>A(7), S=>I7, O=>CO_DUMMY);

  I_36_107 : MUXCY_D
    PORT MAP (CI=>C5, DI=>A(6), S=>I6, LO=>C6, O=>C60);

  I_36_110 : MUXCY_L
    PORT MAP (CI=>C4, DI=>A(5), S=>I5, LO=>C5);

  I_36_63 : MUXCY_L
    PORT MAP (CI=>C3, DI=>A(4), S=>I4, LO=>C4);

  I_36_58 : MUXCY_L
    PORT MAP (CI=>C2, DI=>A(3), S=>I3, LO=>C3);

  I_36_111 : MUXCY_L
    PORT MAP (CI=>CI, DI=>A(0), S=>I0, LO=>C0);

  I_36_55 : MUXCY_L
    PORT MAP (CI=>C0, DI=>A(1), S=>I1, LO=>C1);

  I_36_62 : MUXCY_L
    PORT MAP (CI=>C1, DI=>A(2), S=>I2, LO=>C2);

  I_36_239 : XOR2
    PORT MAP (I0=>C60, I1=>CO_DUMMY, O=>OFL);

  I_36_230 : XOR2
    PORT MAP (I0=>A(2), I1=>B(2), O=>I2);

  I_36_229 : XOR2
    PORT MAP (I0=>A(1), I1=>B(1), O=>I1);

  I_36_228 : XOR2
    PORT MAP (I0=>A(0), I1=>B(0), O=>I0);

```

```

I_36_224 : XOR2
  PORT MAP (I0=>A(4), I1=>B(4), O=>I4);

I_36_223 : XOR2
  PORT MAP (I0=>A(5), I1=>B(5), O=>I5);

I_36_222 : XOR2
  PORT MAP (I0=>A(6), I1=>B(6), O=>I6);

I_36_225 : XOR2
  PORT MAP (I0=>A(3), I1=>B(3), O=>I3);

I_36_221 : XOR2
  PORT MAP (I0=>A(7), I1=>B(7), O=>I7);

I_36_80 : XORCY
  PORT MAP (CI=>C6, LI=>I7, O=>S(7));

I_36_73 : XORCY
  PORT MAP (CI=>CI, LI=>I0, O=>S(0));

I_36_74 : XORCY
  PORT MAP (CI=>C0, LI=>I1, O=>S(1));

I_36_76 : XORCY
  PORT MAP (CI=>C1, LI=>I2, O=>S(2));

I_36_75 : XORCY
  PORT MAP (CI=>C2, LI=>I3, O=>S(3));

I_36_78 : XORCY
  PORT MAP (CI=>C3, LI=>I4, O=>S(4));

I_36_77 : XORCY
  PORT MAP (CI=>C4, LI=>I5, O=>S(5));

I_36_81 : XORCY
  PORT MAP (CI=>C5, LI=>I6, O=>S(6));

END SCHEMATIC;

-- Vhdl model created from schematic complete_datapath_w_dcm_div_5.sch - Thu Oct 19
-- 08:18:01 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 complete_datapath_w_dcm_div_5 IS
  PORT (
    ALU_Mux_En      : IN      STD_LOGIC;
    ALU_Operation   : IN      STD_LOGIC_VECTOR (2 DOWNTO 0);
    Branch_Control  : IN      STD_LOGIC;
    Clkin           : IN      STD_LOGIC;
    DMem_RA_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
    DMem_WA_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
    Data_in_Preload : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
    Inst_RAM_Write_Addr_Preload : IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
    Instruction_in_Preload : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
    Mem_Read        : IN      STD_LOGIC;
    Mem_Write       : IN      STD_LOGIC;
    Mux_RF_Num_Sel  : IN      STD_LOGIC_VECTOR (1 DOWNTO 0);
    Mux_RF_din_Sel  : IN      STD_LOGIC_VECTOR (1 DOWNTO 0);
    Mux_Sel_BEQ_BNE : IN      STD_LOGIC;
    Mux_Sel_IM_Clk_Src : IN      STD_LOGIC;
    Mux_Sel_Inst_RAM_Addr_Src : IN      STD_LOGIC;
    Mux_Sel_PCSrc2  : IN      STD_LOGIC;
    Mux_Sel_PC_Clk_Src : IN      STD_LOGIC;

```

```

RF_En_Read_Write :    IN      STD_LOGIC;
RF_Write_Din_Preload :    IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Num_Preload :    IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
Reset_DCM          :    IN      STD_LOGIC;
Reset_PC          :    IN      STD_LOGIC;
Sel_ALU_Mux_B_in :    IN      STD_LOGIC;
Sel_DMem_Mux_di  :    IN      STD_LOGIC;
Sel_DMem_Mux_ra  :    IN      STD_LOGIC;
Sel_DMem_Mux_wa  :    IN      STD_LOGIC;
Value_of_0       :    IN      STD_LOGIC;
Value_of_1       :    IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
Value_of_Zero    :    IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
WE_Inst_RAM      :    IN      STD_LOGIC;
ALU_Carryout     :    OUT     STD_LOGIC;
ALU_Overflow     :    OUT     STD_LOGIC;
ALU_Res          :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
ALU_Zero_BEQ     :    OUT     STD_LOGIC;
ALU_Zero_BNE     :    OUT     STD_LOGIC;
A_in             :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
Addr_16bits      :    OUT     STD_LOGIC_VECTOR (15 DOWNTO 0);
Addr_16to32bits  :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
Addr_16to8bits   :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Addr_26bits      :    OUT     STD_LOGIC_VECTOR (25 DOWNTO 0);
Addr_26to8bits   :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Addr_from_PC     :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
BEQ_or_BNE       :    OUT     STD_LOGIC;
B_in             :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
Branch_Target    :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Clk0             :    OUT     STD_LOGIC;
ClkDv            :    OUT     STD_LOGIC;
DCM_Locked       :    OUT     STD_LOGIC;
DMem_Din         :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
DMem_to_RF       :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
DRAM_out         :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
Data_RAM_Read_Addr :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Data_RAM_Write_Addr :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Inst_RAM_Addr    :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Instruction_from_IF :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
Mux2Mux_PCSrc   :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
Mux_Sel_PCSrc1   :    OUT     STD_LOGIC;
Mux_to_IM_Clk    :    OUT     STD_LOGIC;
Mux_to_PC_Clk    :    OUT     STD_LOGIC;
Next_PC          :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
PCplus1_Addr     :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
RF_Data_B        :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Data    :    OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
RF_Write_Reg     :    OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
add8_i1_carryout :    OUT     STD_LOGIC;
add8_i1_overflow :    OUT     STD_LOGIC;
add8_i2_carryout :    OUT     STD_LOGIC;
add8_i2_overflow :    OUT     STD_LOGIC;
funct            :    OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
opcode           :    OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
rd               :    OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
rs               :    OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
rt               :    OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);
shamt            :    OUT     STD_LOGIC_VECTOR (4 DOWNTO 0);

end complete_datapath_w_dcm_div_5;

ARCHITECTURE SCHEMATIC OF complete_datapath_w_dcm_div_5 IS
    SIGNAL ALU_Res_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL ALU_Zero_BEQ_DUMMY : STD_LOGIC;
    SIGNAL ALU_Zero_BNE_DUMMY : STD_LOGIC;
    SIGNAL A_in_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL Addr_16bits_DUMMY : STD_LOGIC_VECTOR (15 DOWNTO 0);
    SIGNAL Addr_16to32bits_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL Addr_16to8bits_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL Addr_26bits_DUMMY : STD_LOGIC_VECTOR (25 DOWNTO 0);
    SIGNAL Addr_26to8bits_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL Addr_from_PC_DUMMY : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL BEQ_or_BNE_DUMMY : STD_LOGIC;
    SIGNAL B_in_DUMMY : STD_LOGIC_VECTOR (31 DOWNTO 0);

```



```

SIGNAL Branch_Target_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Clk0_DUMMY               :      STD_LOGIC;
SIGNAL ClkDv_DUMMY              :      STD_LOGIC;
SIGNAL DMem_Din_DUMMY           :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL DMem_to_RF_DUMMY         :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL DRAM_out_DUMMY           :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL Data_RAM_Read_Addr_DUMMY :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Data_RAM_Write_Addr_DUMMY :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Inst_RAM_Addr_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Instruction_from_IF_DUMMY :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL Mux2Mux_PCSrc_DUMMY      :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Mux_Sel_PCSrc1_DUMMY     :      STD_LOGIC;
SIGNAL Mux_to_IM_Clk_DUMMY      :      STD_LOGIC;
SIGNAL Mux_to_PC_Clk_DUMMY      :      STD_LOGIC;
SIGNAL Next_PC_DUMMY            :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL PCplus1_Addr_DUMMY       :      STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL RF_Data_B_DUMMY          :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL RF_Write_Data_DUMMY      :      STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL RF_Write_Reg_DUMMY       :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rd_DUMMY                 :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rs_DUMMY                 :      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL rt_DUMMY                 :      STD_LOGIC_VECTOR (4 DOWNTO 0);

ATTRIBUTE BOX_TYPE : STRING;
ATTRIBUTE U_SET : STRING ;
ATTRIBUTE U_SET OF add8_i1 : LABEL IS "add8_i1_2";
ATTRIBUTE U_SET OF add8_i2 : LABEL IS "add8_i2_0";
ATTRIBUTE U_SET OF m2_1_i3 : LABEL IS "m2_1_i3_4";
ATTRIBUTE U_SET OF m2_1_i2 : LABEL IS "m2_1_i2_3";
ATTRIBUTE U_SET OF m2_1_i1 : LABEL IS "m2_1_i1_1";

COMPONENT ADD8_MXILINK_complete_datapath_w_dcm_div_5
  PORT ( A      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        B      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        CI      :      IN      STD_LOGIC;
        CO      :      OUT     STD_LOGIC;
        OFL     :      OUT     STD_LOGIC;
        S      :      OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT alu_32bit_cla_vhd_bus
  PORT ( A      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        B      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        less_zero :      IN      STD_LOGIC;
        carryout  :      OUT     STD_LOGIC;
        overflow  :      OUT     STD_LOGIC;
        Result    :      OUT     STD_LOGIC_VECTOR (31 DOWNTO 0);
        zero      :      OUT     STD_LOGIC;
        mux_enable :      IN      STD_LOGIC;
        aluoperation :      IN      STD_LOGIC_VECTOR (2 DOWNTO 0));
END COMPONENT;

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 data_bram_sync
  PORT ( clk      :      IN      STD_LOGIC;
        we        :      IN      STD_LOGIC;
        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 COMPONENT;

COMPONENT dcm_div_5
  PORT ( rst_in   :      IN      STD_LOGIC;
        clk_in_in :      IN      STD_LOGIC;
        locked_out :      OUT     STD_LOGIC;
        clkdv_out :      OUT     STD_LOGIC);

```

```

        clk0_out      :      OUT      STD_LOGIC);
END COMPONENT;

COMPONENT inst_ram
    PORT ( clk      :      IN      STD_LOGIC;
          we      :      IN      STD_LOGIC;
          a      :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          di      :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          do      :      OUT      STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT instruction_splitter
    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 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_complete_datapath_w_dcm_div_5
    PORT ( D0      :      IN      STD_LOGIC;
          D1      :      IN      STD_LOGIC;
          S0      :      IN      STD_LOGIC;
          O      :      OUT      STD_LOGIC);
END COMPONENT;

COMPONENT mux32b_2to1
    PORT ( din0 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          din1 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          dout :      OUT      STD_LOGIC_VECTOR (31 DOWNTO 0);
          sel  :      IN      STD_LOGIC);
END COMPONENT;

COMPONENT mux32b_4to1
    PORT ( din0 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          din1 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          din2 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          din3 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          dout :      OUT      STD_LOGIC_VECTOR (31 DOWNTO 0);
          sel  :      IN      STD_LOGIC_VECTOR (1 DOWNTO 0));
END COMPONENT;

COMPONENT mux5b_4to1
    PORT ( din0 :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
          din1 :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
          din2 :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
          din3 :      IN      STD_LOGIC_VECTOR (4 DOWNTO 0);
          dout :      OUT      STD_LOGIC_VECTOR (4 DOWNTO 0);
          sel  :      IN      STD_LOGIC_VECTOR (1 DOWNTO 0));
END COMPONENT;

COMPONENT mux8_2to1
    PORT ( sel :      IN      STD_LOGIC;
          din0 :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          din1 :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          dout :      OUT      STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT mux8b_2to1
    PORT ( din0 :      IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          din1 :      IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
          dout :      OUT      STD_LOGIC_VECTOR (7 DOWNTO 0);
          sel  :      IN      STD_LOGIC);

```

```

END COMPONENT;

COMPONENT pc
  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 COMPONENT;

COMPONENT rf_synch
  PORT ( clk      : IN      STD_LOGIC;
        enable    : IN      STD_LOGIC;
        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);
        read_data_1 : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0);
        read_data_2 : OUT    STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT sign_dextension
  PORT ( addr_in   : IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
        addr_out  : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

COMPONENT sign_extension
  PORT ( addr_in   : IN      STD_LOGIC_VECTOR (15 DOWNTO 0);
        addr_out  : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT tristate_32b
  PORT ( enable    : IN      STD_LOGIC;
        din       : IN      STD_LOGIC_VECTOR (31 DOWNTO 0);
        dout      : OUT     STD_LOGIC_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT truncator26to8bits
  PORT ( addr_in   : IN      STD_LOGIC_VECTOR (25 DOWNTO 0);
        addr_out  : OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

BEGIN
  ALU_Res <= ALU_Res_DUMMY;
  ALU_Zero_BEQ <= ALU_Zero_BEQ_DUMMY;
  ALU_Zero_BNE <= ALU_Zero_BNE_DUMMY;
  A_in <= A_in_DUMMY;
  Addr_16bits <= Addr_16bits_DUMMY;
  Addr_16to32bits <= Addr_16to32bits_DUMMY;
  Addr_16to8bits <= Addr_16to8bits_DUMMY;
  Addr_26bits <= Addr_26bits_DUMMY;
  Addr_26to8bits <= Addr_26to8bits_DUMMY;
  Addr_from_PC <= Addr_from_PC_DUMMY;
  BEQ_or_BNE <= BEQ_or_BNE_DUMMY;
  B_in <= B_in_DUMMY;
  Branch_Target <= Branch_Target_DUMMY;
  Clk0 <= Clk0_DUMMY;
  ClkDv <= ClkDv_DUMMY;
  DMem_Din <= DMem_Din_DUMMY;
  DMem_to_RF <= DMem_to_RF_DUMMY;
  DRAM_out <= DRAM_out_DUMMY;
  Data_RAM_Read_Addr <= Data_RAM_Read_Addr_DUMMY;
  Data_RAM_Write_Addr <= Data_RAM_Write_Addr_DUMMY;
  Inst_RAM_Addr <= Inst_RAM_Addr_DUMMY;
  Instruction_from_IF <= Instruction_from_IF_DUMMY;
  Mux2Mux_PCSrc <= Mux2Mux_PCSrc_DUMMY;
  Mux_Sel_PCSrc1 <= Mux_Sel_PCSrc1_DUMMY;
  Mux_to_IM_Clk <= Mux_to_IM_Clk_DUMMY;
  Mux_to_PC_Clk <= Mux_to_PC_Clk_DUMMY;
  Next_PC <= Next_PC_DUMMY;
  PCplus1_Addr <= PCplus1_Addr_DUMMY;
  RF_Data_B <= RF_Data_B_DUMMY;
  RF_Write_Data <= RF_Write_Data_DUMMY;
  RF_Write_Reg <= RF_Write_Reg_DUMMY;

```

```

rd <= rd_DUMMY;
rs <= rs_DUMMY;
rt <= rt_DUMMY;

add8_i1 : ADD8_MXILINX_complete_datapath_w_dcm_div_5
PORT MAP (A(7)=>Addr_from_PC_DUMMY(7), A(6)=>Addr_from_PC_DUMMY(6),
A(5)=>Addr_from_PC_DUMMY(5), A(4)=>Addr_from_PC_DUMMY(4),
A(3)=>Addr_from_PC_DUMMY(3), A(2)=>Addr_from_PC_DUMMY(2),
A(1)=>Addr_from_PC_DUMMY(1), A(0)=>Addr_from_PC_DUMMY(0),
B(7)=>Value_of_1(7), B(6)=>Value_of_1(6), B(5)=>Value_of_1(5),
B(4)=>Value_of_1(4), B(3)=>Value_of_1(3), B(2)=>Value_of_1(2),
B(1)=>Value_of_1(1), B(0)=>Value_of_1(0), CI=>Value_of_0,
CO=>add8_i1_carryout, OFL=>add8_i1_overflow, S(7)=>PCplus1_Addr_DUMMY(7),
S(6)=>PCplus1_Addr_DUMMY(6), S(5)=>PCplus1_Addr_DUMMY(5),
S(4)=>PCplus1_Addr_DUMMY(4), S(3)=>PCplus1_Addr_DUMMY(3),
S(2)=>PCplus1_Addr_DUMMY(2), S(1)=>PCplus1_Addr_DUMMY(1),
S(0)=>PCplus1_Addr_DUMMY(0));

add8_i2 : ADD8_MXILINX_complete_datapath_w_dcm_div_5
PORT MAP (A(7)=>PCplus1_Addr_DUMMY(7), A(6)=>PCplus1_Addr_DUMMY(6),
A(5)=>PCplus1_Addr_DUMMY(5), A(4)=>PCplus1_Addr_DUMMY(4),
A(3)=>PCplus1_Addr_DUMMY(3), A(2)=>PCplus1_Addr_DUMMY(2),
A(1)=>PCplus1_Addr_DUMMY(1), A(0)=>PCplus1_Addr_DUMMY(0),
B(7)=>Addr_16to8bits_DUMMY(7), B(6)=>Addr_16to8bits_DUMMY(6),
B(5)=>Addr_16to8bits_DUMMY(5), B(4)=>Addr_16to8bits_DUMMY(4),
B(3)=>Addr_16to8bits_DUMMY(3), B(2)=>Addr_16to8bits_DUMMY(2),
B(1)=>Addr_16to8bits_DUMMY(1), B(0)=>Addr_16to8bits_DUMMY(0),
CI=>Value_of_0, CO=>add8_i2_carryout, OFL=>add8_i2_overflow,
S(7)=>Branch_Target_DUMMY(7), S(6)=>Branch_Target_DUMMY(6),
S(5)=>Branch_Target_DUMMY(5), S(4)=>Branch_Target_DUMMY(4),
S(3)=>Branch_Target_DUMMY(3), S(2)=>Branch_Target_DUMMY(2),
S(1)=>Branch_Target_DUMMY(1), S(0)=>Branch_Target_DUMMY(0));

alu : alu_32bit_cla_vhd_bus
PORT MAP (A(31)=>A_in_DUMMY(31), A(30)=>A_in_DUMMY(30),
A(29)=>A_in_DUMMY(29), A(28)=>A_in_DUMMY(28), A(27)=>A_in_DUMMY(27),
A(26)=>A_in_DUMMY(26), A(25)=>A_in_DUMMY(25), A(24)=>A_in_DUMMY(24),
A(23)=>A_in_DUMMY(23), A(22)=>A_in_DUMMY(22), A(21)=>A_in_DUMMY(21),
A(20)=>A_in_DUMMY(20), A(19)=>A_in_DUMMY(19), A(18)=>A_in_DUMMY(18),
A(17)=>A_in_DUMMY(17), A(16)=>A_in_DUMMY(16), A(15)=>A_in_DUMMY(15),
A(14)=>A_in_DUMMY(14), A(13)=>A_in_DUMMY(13), A(12)=>A_in_DUMMY(12),
A(11)=>A_in_DUMMY(11), A(10)=>A_in_DUMMY(10), A(9)=>A_in_DUMMY(9),
A(8)=>A_in_DUMMY(8), A(7)=>A_in_DUMMY(7), A(6)=>A_in_DUMMY(6),
A(5)=>A_in_DUMMY(5), A(4)=>A_in_DUMMY(4), A(3)=>A_in_DUMMY(3),
A(2)=>A_in_DUMMY(2), A(1)=>A_in_DUMMY(1), A(0)=>A_in_DUMMY(0),
B(31)=>B_in_DUMMY(31), B(30)=>B_in_DUMMY(30), B(29)=>B_in_DUMMY(29),
B(28)=>B_in_DUMMY(28), B(27)=>B_in_DUMMY(27), B(26)=>B_in_DUMMY(26),
B(25)=>B_in_DUMMY(25), B(24)=>B_in_DUMMY(24), B(23)=>B_in_DUMMY(23),
B(22)=>B_in_DUMMY(22), B(21)=>B_in_DUMMY(21), B(20)=>B_in_DUMMY(20),
B(19)=>B_in_DUMMY(19), B(18)=>B_in_DUMMY(18), B(17)=>B_in_DUMMY(17),
B(16)=>B_in_DUMMY(16), B(15)=>B_in_DUMMY(15), B(14)=>B_in_DUMMY(14),
B(13)=>B_in_DUMMY(13), B(12)=>B_in_DUMMY(12), B(11)=>B_in_DUMMY(11),
B(10)=>B_in_DUMMY(10), B(9)=>B_in_DUMMY(9), B(8)=>B_in_DUMMY(8),
B(7)=>B_in_DUMMY(7), B(6)=>B_in_DUMMY(6), B(5)=>B_in_DUMMY(5),
B(4)=>B_in_DUMMY(4), B(3)=>B_in_DUMMY(3), B(2)=>B_in_DUMMY(2),
B(1)=>B_in_DUMMY(1), B(0)=>B_in_DUMMY(0), less_zero=>Value_of_0,
carryout=>ALU_Carryout, overflow=>ALU_Overflow,
Result(31)=>ALU_Res_DUMMY(31), Result(30)=>ALU_Res_DUMMY(30),
Result(29)=>ALU_Res_DUMMY(29), Result(28)=>ALU_Res_DUMMY(28),
Result(27)=>ALU_Res_DUMMY(27), Result(26)=>ALU_Res_DUMMY(26),
Result(25)=>ALU_Res_DUMMY(25), Result(24)=>ALU_Res_DUMMY(24),
Result(23)=>ALU_Res_DUMMY(23), Result(22)=>ALU_Res_DUMMY(22),
Result(21)=>ALU_Res_DUMMY(21), Result(20)=>ALU_Res_DUMMY(20),
Result(19)=>ALU_Res_DUMMY(19), Result(18)=>ALU_Res_DUMMY(18),
Result(17)=>ALU_Res_DUMMY(17), Result(16)=>ALU_Res_DUMMY(16),
Result(15)=>ALU_Res_DUMMY(15), Result(14)=>ALU_Res_DUMMY(14),
Result(13)=>ALU_Res_DUMMY(13), Result(12)=>ALU_Res_DUMMY(12),
Result(11)=>ALU_Res_DUMMY(11), Result(10)=>ALU_Res_DUMMY(10),
Result(9)=>ALU_Res_DUMMY(9), Result(8)=>ALU_Res_DUMMY(8),
Result(7)=>ALU_Res_DUMMY(7), Result(6)=>ALU_Res_DUMMY(6),
Result(5)=>ALU_Res_DUMMY(5), Result(4)=>ALU_Res_DUMMY(4),
Result(3)=>ALU_Res_DUMMY(3), Result(2)=>ALU_Res_DUMMY(2),
Result(1)=>ALU_Res_DUMMY(1), Result(0)=>ALU_Res_DUMMY(0),

```

```

zero=>ALU_Zero_BEQ_DUMMY, mux_enable=>ALU_Mux_En,
aluoperation(2)=>ALU_Operation(2), aluoperation(1)=>ALU_Operation(1),
aluoperation(0)=>ALU_Operation(0));

and2_i1 : AND2
  PORT MAP (I0=>BEQ_or_BNE_DUMMY, I1=>Branch_Control,
    O=>Mux_Sel_PCSrc1_DUMMY);

Data_RAM : data_bram_sync
  PORT MAP (clk=>Clk0_DUMMY, we=>Mem_Write,
    wa(7)=>Data_RAM_Write_Addr_DUMMY(7), wa(6)=>Data_RAM_Write_Addr_DUMMY(6),
    wa(5)=>Data_RAM_Write_Addr_DUMMY(5), wa(4)=>Data_RAM_Write_Addr_DUMMY(4),
    wa(3)=>Data_RAM_Write_Addr_DUMMY(3), wa(2)=>Data_RAM_Write_Addr_DUMMY(2),
    wa(1)=>Data_RAM_Write_Addr_DUMMY(1), wa(0)=>Data_RAM_Write_Addr_DUMMY(0),
    ra(7)=>Data_RAM_Read_Addr_DUMMY(7), ra(6)=>Data_RAM_Read_Addr_DUMMY(6),
    ra(5)=>Data_RAM_Read_Addr_DUMMY(5), ra(4)=>Data_RAM_Read_Addr_DUMMY(4),
    ra(3)=>Data_RAM_Read_Addr_DUMMY(3), ra(2)=>Data_RAM_Read_Addr_DUMMY(2),
    ra(1)=>Data_RAM_Read_Addr_DUMMY(1), ra(0)=>Data_RAM_Read_Addr_DUMMY(0),
    di(31)=>DMem_Din_DUMMY(31), di(30)=>DMem_Din_DUMMY(30),
    di(29)=>DMem_Din_DUMMY(29), di(28)=>DMem_Din_DUMMY(28),
    di(27)=>DMem_Din_DUMMY(27), di(26)=>DMem_Din_DUMMY(26),
    di(25)=>DMem_Din_DUMMY(25), di(24)=>DMem_Din_DUMMY(24),
    di(23)=>DMem_Din_DUMMY(23), di(22)=>DMem_Din_DUMMY(22),
    di(21)=>DMem_Din_DUMMY(21), di(20)=>DMem_Din_DUMMY(20),
    di(19)=>DMem_Din_DUMMY(19), di(18)=>DMem_Din_DUMMY(18),
    di(17)=>DMem_Din_DUMMY(17), di(16)=>DMem_Din_DUMMY(16),
    di(15)=>DMem_Din_DUMMY(15), di(14)=>DMem_Din_DUMMY(14),
    di(13)=>DMem_Din_DUMMY(13), di(12)=>DMem_Din_DUMMY(12),
    di(11)=>DMem_Din_DUMMY(11), di(10)=>DMem_Din_DUMMY(10),
    di(9)=>DMem_Din_DUMMY(9), di(8)=>DMem_Din_DUMMY(8),
    di(7)=>DMem_Din_DUMMY(7), di(6)=>DMem_Din_DUMMY(6),
    di(5)=>DMem_Din_DUMMY(5), di(4)=>DMem_Din_DUMMY(4),
    di(3)=>DMem_Din_DUMMY(3), di(2)=>DMem_Din_DUMMY(2),
    di(1)=>DMem_Din_DUMMY(1), di(0)=>DMem_Din_DUMMY(0),
    do(31)=>DRAM_out_DUMMY(31), do(30)=>DRAM_out_DUMMY(30),
    do(29)=>DRAM_out_DUMMY(29), do(28)=>DRAM_out_DUMMY(28),
    do(27)=>DRAM_out_DUMMY(27), do(26)=>DRAM_out_DUMMY(26),
    do(25)=>DRAM_out_DUMMY(25), do(24)=>DRAM_out_DUMMY(24),
    do(23)=>DRAM_out_DUMMY(23), do(22)=>DRAM_out_DUMMY(22),
    do(21)=>DRAM_out_DUMMY(21), do(20)=>DRAM_out_DUMMY(20),
    do(19)=>DRAM_out_DUMMY(19), do(18)=>DRAM_out_DUMMY(18),
    do(17)=>DRAM_out_DUMMY(17), do(16)=>DRAM_out_DUMMY(16),
    do(15)=>DRAM_out_DUMMY(15), do(14)=>DRAM_out_DUMMY(14),
    do(13)=>DRAM_out_DUMMY(13), do(12)=>DRAM_out_DUMMY(12),
    do(11)=>DRAM_out_DUMMY(11), do(10)=>DRAM_out_DUMMY(10),
    do(9)=>DRAM_out_DUMMY(9), do(8)=>DRAM_out_DUMMY(8),
    do(7)=>DRAM_out_DUMMY(7), do(6)=>DRAM_out_DUMMY(6),
    do(5)=>DRAM_out_DUMMY(5), do(4)=>DRAM_out_DUMMY(4),
    do(3)=>DRAM_out_DUMMY(3), do(2)=>DRAM_out_DUMMY(2),
    do(1)=>DRAM_out_DUMMY(1), do(0)=>DRAM_out_DUMMY(0));

DCM_i1 : dcm_div_5
  PORT MAP (rst_in=>Reset_DCM, clk_in=>Clkin, locked_out=>DCM_Locked,
    clkdv_out=>ClkDv_DUMMY, clk0_out=>Clk0_DUMMY);

instruction_ram : inst_ram
  PORT MAP (clk=>Mux_to_IM_Clk_DUMMY, we=>WE_Inst_RAM,
    a(7)=>Inst_RAM_Addr_DUMMY(7), a(6)=>Inst_RAM_Addr_DUMMY(6),
    a(5)=>Inst_RAM_Addr_DUMMY(5), a(4)=>Inst_RAM_Addr_DUMMY(4),
    a(3)=>Inst_RAM_Addr_DUMMY(3), a(2)=>Inst_RAM_Addr_DUMMY(2),
    a(1)=>Inst_RAM_Addr_DUMMY(1), a(0)=>Inst_RAM_Addr_DUMMY(0),
    di(31)=>Instruction_in_Preload(31), di(30)=>Instruction_in_Preload(30),
    di(29)=>Instruction_in_Preload(29), di(28)=>Instruction_in_Preload(28),
    di(27)=>Instruction_in_Preload(27), di(26)=>Instruction_in_Preload(26),
    di(25)=>Instruction_in_Preload(25), di(24)=>Instruction_in_Preload(24),
    di(23)=>Instruction_in_Preload(23), di(22)=>Instruction_in_Preload(22),
    di(21)=>Instruction_in_Preload(21), di(20)=>Instruction_in_Preload(20),
    di(19)=>Instruction_in_Preload(19), di(18)=>Instruction_in_Preload(18),
    di(17)=>Instruction_in_Preload(17), di(16)=>Instruction_in_Preload(16),
    di(15)=>Instruction_in_Preload(15), di(14)=>Instruction_in_Preload(14),
    di(13)=>Instruction_in_Preload(13), di(12)=>Instruction_in_Preload(12),
    di(11)=>Instruction_in_Preload(11), di(10)=>Instruction_in_Preload(10),
    di(9)=>Instruction_in_Preload(9), di(8)=>Instruction_in_Preload(8),

```

```

di(7)=>Instruction_in_Preload(7), di(6)=>Instruction_in_Preload(6),
di(5)=>Instruction_in_Preload(5), di(4)=>Instruction_in_Preload(4),
di(3)=>Instruction_in_Preload(3), di(2)=>Instruction_in_Preload(2),
di(1)=>Instruction_in_Preload(1), di(0)=>Instruction_in_Preload(0),
do(31)=>Instruction_from_IF_DUMMY(31),
do(30)=>Instruction_from_IF_DUMMY(30),
do(29)=>Instruction_from_IF_DUMMY(29),
do(28)=>Instruction_from_IF_DUMMY(28),
do(27)=>Instruction_from_IF_DUMMY(27),
do(26)=>Instruction_from_IF_DUMMY(26),
do(25)=>Instruction_from_IF_DUMMY(25),
do(24)=>Instruction_from_IF_DUMMY(24),
do(23)=>Instruction_from_IF_DUMMY(23),
do(22)=>Instruction_from_IF_DUMMY(22),
do(21)=>Instruction_from_IF_DUMMY(21),
do(20)=>Instruction_from_IF_DUMMY(20),
do(19)=>Instruction_from_IF_DUMMY(19),
do(18)=>Instruction_from_IF_DUMMY(18),
do(17)=>Instruction_from_IF_DUMMY(17),
do(16)=>Instruction_from_IF_DUMMY(16),
do(15)=>Instruction_from_IF_DUMMY(15),
do(14)=>Instruction_from_IF_DUMMY(14),
do(13)=>Instruction_from_IF_DUMMY(13),
do(12)=>Instruction_from_IF_DUMMY(12),
do(11)=>Instruction_from_IF_DUMMY(11),
do(10)=>Instruction_from_IF_DUMMY(10),
do(9)=>Instruction_from_IF_DUMMY(9), do(8)=>Instruction_from_IF_DUMMY(8),
do(7)=>Instruction_from_IF_DUMMY(7), do(6)=>Instruction_from_IF_DUMMY(6),
do(5)=>Instruction_from_IF_DUMMY(5), do(4)=>Instruction_from_IF_DUMMY(4),
do(3)=>Instruction_from_IF_DUMMY(3), do(2)=>Instruction_from_IF_DUMMY(2),
do(1)=>Instruction_from_IF_DUMMY(1), do(0)=>Instruction_from_IF_DUMMY(0));

instruction_split : instruction_splitter
PORT MAP (instruction(31)=>Instruction_from_IF_DUMMY(31),
instruction(30)=>Instruction_from_IF_DUMMY(30),
instruction(29)=>Instruction_from_IF_DUMMY(29),
instruction(28)=>Instruction_from_IF_DUMMY(28),
instruction(27)=>Instruction_from_IF_DUMMY(27),
instruction(26)=>Instruction_from_IF_DUMMY(26),
instruction(25)=>Instruction_from_IF_DUMMY(25),
instruction(24)=>Instruction_from_IF_DUMMY(24),
instruction(23)=>Instruction_from_IF_DUMMY(23),
instruction(22)=>Instruction_from_IF_DUMMY(22),
instruction(21)=>Instruction_from_IF_DUMMY(21),
instruction(20)=>Instruction_from_IF_DUMMY(20),
instruction(19)=>Instruction_from_IF_DUMMY(19),
instruction(18)=>Instruction_from_IF_DUMMY(18),
instruction(17)=>Instruction_from_IF_DUMMY(17),
instruction(16)=>Instruction_from_IF_DUMMY(16),
instruction(15)=>Instruction_from_IF_DUMMY(15),
instruction(14)=>Instruction_from_IF_DUMMY(14),
instruction(13)=>Instruction_from_IF_DUMMY(13),
instruction(12)=>Instruction_from_IF_DUMMY(12),
instruction(11)=>Instruction_from_IF_DUMMY(11),
instruction(10)=>Instruction_from_IF_DUMMY(10),
instruction(9)=>Instruction_from_IF_DUMMY(9),
instruction(8)=>Instruction_from_IF_DUMMY(8),
instruction(7)=>Instruction_from_IF_DUMMY(7),
instruction(6)=>Instruction_from_IF_DUMMY(6),
instruction(5)=>Instruction_from_IF_DUMMY(5),
instruction(4)=>Instruction_from_IF_DUMMY(4),
instruction(3)=>Instruction_from_IF_DUMMY(3),
instruction(2)=>Instruction_from_IF_DUMMY(2),
instruction(1)=>Instruction_from_IF_DUMMY(1),
instruction(0)=>Instruction_from_IF_DUMMY(0), bits31_26(5)=>opcode(5),
bits31_26(4)=>opcode(4), bits31_26(3)=>opcode(3),
bits31_26(2)=>opcode(2), bits31_26(1)=>opcode(1),
bits31_26(0)=>opcode(0), bits25_21(4)=>rs_DUMMY(4),
bits25_21(3)=>rs_DUMMY(3), bits25_21(2)=>rs_DUMMY(2),
bits25_21(1)=>rs_DUMMY(1), bits25_21(0)=>rs_DUMMY(0),
bits20_16(4)=>rt_DUMMY(4), bits20_16(3)=>rt_DUMMY(3),
bits20_16(2)=>rt_DUMMY(2), bits20_16(1)=>rt_DUMMY(1),
bits20_16(0)=>rt_DUMMY(0), bits15_11(4)=>rd_DUMMY(4),

```

```

bits15_11(3)=>rd_DUMMY(3), bits15_11(2)=>rd_DUMMY(2),
bits15_11(1)=>rd_DUMMY(1), bits15_11(0)=>rd_DUMMY(0),
bits10_6(4)=>shamt(4), bits10_6(3)=>shamt(3), bits10_6(2)=>shamt(2),
bits10_6(1)=>shamt(1), bits10_6(0)=>shamt(0), bits5_0(5)=>funct(5),
bits5_0(4)=>funct(4), bits5_0(3)=>funct(3), bits5_0(2)=>funct(2),
bits5_0(1)=>funct(1), bits5_0(0)=>funct(0),
bits15_0(15)=>Addr_16bits_DUMMY(15), bits15_0(14)=>Addr_16bits_DUMMY(14),
bits15_0(13)=>Addr_16bits_DUMMY(13), bits15_0(12)=>Addr_16bits_DUMMY(12),
bits15_0(11)=>Addr_16bits_DUMMY(11), bits15_0(10)=>Addr_16bits_DUMMY(10),
bits15_0(9)=>Addr_16bits_DUMMY(9), bits15_0(8)=>Addr_16bits_DUMMY(8),
bits15_0(7)=>Addr_16bits_DUMMY(7), bits15_0(6)=>Addr_16bits_DUMMY(6),
bits15_0(5)=>Addr_16bits_DUMMY(5), bits15_0(4)=>Addr_16bits_DUMMY(4),
bits15_0(3)=>Addr_16bits_DUMMY(3), bits15_0(2)=>Addr_16bits_DUMMY(2),
bits15_0(1)=>Addr_16bits_DUMMY(1), bits15_0(0)=>Addr_16bits_DUMMY(0),
bits25_0(25)=>Addr_26bits_DUMMY(25), bits25_0(24)=>Addr_26bits_DUMMY(24),
bits25_0(23)=>Addr_26bits_DUMMY(23), bits25_0(22)=>Addr_26bits_DUMMY(22),
bits25_0(21)=>Addr_26bits_DUMMY(21), bits25_0(20)=>Addr_26bits_DUMMY(20),
bits25_0(19)=>Addr_26bits_DUMMY(19), bits25_0(18)=>Addr_26bits_DUMMY(18),
bits25_0(17)=>Addr_26bits_DUMMY(17), bits25_0(16)=>Addr_26bits_DUMMY(16),
bits25_0(15)=>Addr_26bits_DUMMY(15), bits25_0(14)=>Addr_26bits_DUMMY(14),
bits25_0(13)=>Addr_26bits_DUMMY(13), bits25_0(12)=>Addr_26bits_DUMMY(12),
bits25_0(11)=>Addr_26bits_DUMMY(11), bits25_0(10)=>Addr_26bits_DUMMY(10),
bits25_0(9)=>Addr_26bits_DUMMY(9), bits25_0(8)=>Addr_26bits_DUMMY(8),
bits25_0(7)=>Addr_26bits_DUMMY(7), bits25_0(6)=>Addr_26bits_DUMMY(6),
bits25_0(5)=>Addr_26bits_DUMMY(5), bits25_0(4)=>Addr_26bits_DUMMY(4),
bits25_0(3)=>Addr_26bits_DUMMY(3), bits25_0(2)=>Addr_26bits_DUMMY(2),
bits25_0(1)=>Addr_26bits_DUMMY(1), bits25_0(0)=>Addr_26bits_DUMMY(0));

inv_i1 : INV
PORT MAP (I=>ALU_Zero_BEQ_DUMMY, O=>ALU_Zero_BNE_DUMMY);

m2_1_i3 : M2_1_MXILINX_complete_datapath_w_dcm_div_5
PORT MAP (D0=>Clk0_DUMMY, D1=>ClkDv_DUMMY, S0=>Mux_Sel_IM_Clk_Src,
O=>Mux_to_IM_Clk_DUMMY);

m2_1_i2 : M2_1_MXILINX_complete_datapath_w_dcm_div_5
PORT MAP (D0=>Clk0_DUMMY, D1=>ClkDv_DUMMY, S0=>Mux_Sel_PC_Clk_Src,
O=>Mux_to_PC_Clk_DUMMY);

m2_1_i1 : M2_1_MXILINX_complete_datapath_w_dcm_div_5
PORT MAP (D0=>ALU_Zero_BEQ_DUMMY, D1=>ALU_Zero_BNE_DUMMY,
S0=>Mux_Sel_BEQ_BNE, O=>BEQ_or_BNE_DUMMY);

DMem_Mux_di : mux32b_2to1
PORT MAP (din0(31)=>Data_in_Preload(31), din0(30)=>Data_in_Preload(30),
din0(29)=>Data_in_Preload(29), din0(28)=>Data_in_Preload(28),
din0(27)=>Data_in_Preload(27), din0(26)=>Data_in_Preload(26),
din0(25)=>Data_in_Preload(25), din0(24)=>Data_in_Preload(24),
din0(23)=>Data_in_Preload(23), din0(22)=>Data_in_Preload(22),
din0(21)=>Data_in_Preload(21), din0(20)=>Data_in_Preload(20),
din0(19)=>Data_in_Preload(19), din0(18)=>Data_in_Preload(18),
din0(17)=>Data_in_Preload(17), din0(16)=>Data_in_Preload(16),
din0(15)=>Data_in_Preload(15), din0(14)=>Data_in_Preload(14),
din0(13)=>Data_in_Preload(13), din0(12)=>Data_in_Preload(12),
din0(11)=>Data_in_Preload(11), din0(10)=>Data_in_Preload(10),
din0(9)=>Data_in_Preload(9), din0(8)=>Data_in_Preload(8),
din0(7)=>Data_in_Preload(7), din0(6)=>Data_in_Preload(6),
din0(5)=>Data_in_Preload(5), din0(4)=>Data_in_Preload(4),
din0(3)=>Data_in_Preload(3), din0(2)=>Data_in_Preload(2),
din0(1)=>Data_in_Preload(1), din0(0)=>Data_in_Preload(0),
din1(31)=>RF_Data_B_DUMMY(31), din1(30)=>RF_Data_B_DUMMY(30),
din1(29)=>RF_Data_B_DUMMY(29), din1(28)=>RF_Data_B_DUMMY(28),
din1(27)=>RF_Data_B_DUMMY(27), din1(26)=>RF_Data_B_DUMMY(26),
din1(25)=>RF_Data_B_DUMMY(25), din1(24)=>RF_Data_B_DUMMY(24),
din1(23)=>RF_Data_B_DUMMY(23), din1(22)=>RF_Data_B_DUMMY(22),
din1(21)=>RF_Data_B_DUMMY(21), din1(20)=>RF_Data_B_DUMMY(20),
din1(19)=>RF_Data_B_DUMMY(19), din1(18)=>RF_Data_B_DUMMY(18),
din1(17)=>RF_Data_B_DUMMY(17), din1(16)=>RF_Data_B_DUMMY(16),
din1(15)=>RF_Data_B_DUMMY(15), din1(14)=>RF_Data_B_DUMMY(14),
din1(13)=>RF_Data_B_DUMMY(13), din1(12)=>RF_Data_B_DUMMY(12),
din1(11)=>RF_Data_B_DUMMY(11), din1(10)=>RF_Data_B_DUMMY(10),
din1(9)=>RF_Data_B_DUMMY(9), din1(8)=>RF_Data_B_DUMMY(8),
din1(7)=>RF_Data_B_DUMMY(7), din1(6)=>RF_Data_B_DUMMY(6),

```

```

dinl(5)=>RF_Data_B_DUMMY(5), dinl(4)=>RF_Data_B_DUMMY(4),
dinl(3)=>RF_Data_B_DUMMY(3), dinl(2)=>RF_Data_B_DUMMY(2),
dinl(1)=>RF_Data_B_DUMMY(1), dinl(0)=>RF_Data_B_DUMMY(0),
dout(31)=>DMem_Din_DUMMY(31), dout(30)=>DMem_Din_DUMMY(30),
dout(29)=>DMem_Din_DUMMY(29), dout(28)=>DMem_Din_DUMMY(28),
dout(27)=>DMem_Din_DUMMY(27), dout(26)=>DMem_Din_DUMMY(26),
dout(25)=>DMem_Din_DUMMY(25), dout(24)=>DMem_Din_DUMMY(24),
dout(23)=>DMem_Din_DUMMY(23), dout(22)=>DMem_Din_DUMMY(22),
dout(21)=>DMem_Din_DUMMY(21), dout(20)=>DMem_Din_DUMMY(20),
dout(19)=>DMem_Din_DUMMY(19), dout(18)=>DMem_Din_DUMMY(18),
dout(17)=>DMem_Din_DUMMY(17), dout(16)=>DMem_Din_DUMMY(16),
dout(15)=>DMem_Din_DUMMY(15), dout(14)=>DMem_Din_DUMMY(14),
dout(13)=>DMem_Din_DUMMY(13), dout(12)=>DMem_Din_DUMMY(12),
dout(11)=>DMem_Din_DUMMY(11), dout(10)=>DMem_Din_DUMMY(10),
dout(9)=>DMem_Din_DUMMY(9), dout(8)=>DMem_Din_DUMMY(8),
dout(7)=>DMem_Din_DUMMY(7), dout(6)=>DMem_Din_DUMMY(6),
dout(5)=>DMem_Din_DUMMY(5), dout(4)=>DMem_Din_DUMMY(4),
dout(3)=>DMem_Din_DUMMY(3), dout(2)=>DMem_Din_DUMMY(2),
dout(1)=>DMem_Din_DUMMY(1), dout(0)=>DMem_Din_DUMMY(0),
sel=>Sel_DMem_Mux_di);

ALU_Mux_B_in : mux32b_2to1
PORT MAP (din0(31)=>RF_Data_B_DUMMY(31), din0(30)=>RF_Data_B_DUMMY(30),
din0(29)=>RF_Data_B_DUMMY(29), din0(28)=>RF_Data_B_DUMMY(28),
din0(27)=>RF_Data_B_DUMMY(27), din0(26)=>RF_Data_B_DUMMY(26),
din0(25)=>RF_Data_B_DUMMY(25), din0(24)=>RF_Data_B_DUMMY(24),
din0(23)=>RF_Data_B_DUMMY(23), din0(22)=>RF_Data_B_DUMMY(22),
din0(21)=>RF_Data_B_DUMMY(21), din0(20)=>RF_Data_B_DUMMY(20),
din0(19)=>RF_Data_B_DUMMY(19), din0(18)=>RF_Data_B_DUMMY(18),
din0(17)=>RF_Data_B_DUMMY(17), din0(16)=>RF_Data_B_DUMMY(16),
din0(15)=>RF_Data_B_DUMMY(15), din0(14)=>RF_Data_B_DUMMY(14),
din0(13)=>RF_Data_B_DUMMY(13), din0(12)=>RF_Data_B_DUMMY(12),
din0(11)=>RF_Data_B_DUMMY(11), din0(10)=>RF_Data_B_DUMMY(10),
din0(9)=>RF_Data_B_DUMMY(9), din0(8)=>RF_Data_B_DUMMY(8),
din0(7)=>RF_Data_B_DUMMY(7), din0(6)=>RF_Data_B_DUMMY(6),
din0(5)=>RF_Data_B_DUMMY(5), din0(4)=>RF_Data_B_DUMMY(4),
din0(3)=>RF_Data_B_DUMMY(3), din0(2)=>RF_Data_B_DUMMY(2),
din0(1)=>RF_Data_B_DUMMY(1), din0(0)=>RF_Data_B_DUMMY(0),
dinl(31)=>Addr_16to32bits_DUMMY(31), dinl(30)=>Addr_16to32bits_DUMMY(30),
dinl(29)=>Addr_16to32bits_DUMMY(29), dinl(28)=>Addr_16to32bits_DUMMY(28),
dinl(27)=>Addr_16to32bits_DUMMY(27), dinl(26)=>Addr_16to32bits_DUMMY(26),
dinl(25)=>Addr_16to32bits_DUMMY(25), dinl(24)=>Addr_16to32bits_DUMMY(24),
dinl(23)=>Addr_16to32bits_DUMMY(23), dinl(22)=>Addr_16to32bits_DUMMY(22),
dinl(21)=>Addr_16to32bits_DUMMY(21), dinl(20)=>Addr_16to32bits_DUMMY(20),
dinl(19)=>Addr_16to32bits_DUMMY(19), dinl(18)=>Addr_16to32bits_DUMMY(18),
dinl(17)=>Addr_16to32bits_DUMMY(17), dinl(16)=>Addr_16to32bits_DUMMY(16),
dinl(15)=>Addr_16to32bits_DUMMY(15), dinl(14)=>Addr_16to32bits_DUMMY(14),
dinl(13)=>Addr_16to32bits_DUMMY(13), dinl(12)=>Addr_16to32bits_DUMMY(12),
dinl(11)=>Addr_16to32bits_DUMMY(11), dinl(10)=>Addr_16to32bits_DUMMY(10),
dinl(9)=>Addr_16to32bits_DUMMY(9), dinl(8)=>Addr_16to32bits_DUMMY(8),
dinl(7)=>Addr_16to32bits_DUMMY(7), dinl(6)=>Addr_16to32bits_DUMMY(6),
dinl(5)=>Addr_16to32bits_DUMMY(5), dinl(4)=>Addr_16to32bits_DUMMY(4),
dinl(3)=>Addr_16to32bits_DUMMY(3), dinl(2)=>Addr_16to32bits_DUMMY(2),
dinl(1)=>Addr_16to32bits_DUMMY(1), dinl(0)=>Addr_16to32bits_DUMMY(0),
dout(31)=>B_in_DUMMY(31), dout(30)=>B_in_DUMMY(30),
dout(29)=>B_in_DUMMY(29), dout(28)=>B_in_DUMMY(28),
dout(27)=>B_in_DUMMY(27), dout(26)=>B_in_DUMMY(26),
dout(25)=>B_in_DUMMY(25), dout(24)=>B_in_DUMMY(24),
dout(23)=>B_in_DUMMY(23), dout(22)=>B_in_DUMMY(22),
dout(21)=>B_in_DUMMY(21), dout(20)=>B_in_DUMMY(20),
dout(19)=>B_in_DUMMY(19), dout(18)=>B_in_DUMMY(18),
dout(17)=>B_in_DUMMY(17), dout(16)=>B_in_DUMMY(16),
dout(15)=>B_in_DUMMY(15), dout(14)=>B_in_DUMMY(14),
dout(13)=>B_in_DUMMY(13), dout(12)=>B_in_DUMMY(12),
dout(11)=>B_in_DUMMY(11), dout(10)=>B_in_DUMMY(10),
dout(9)=>B_in_DUMMY(9), dout(8)=>B_in_DUMMY(8), dout(7)=>B_in_DUMMY(7),
dout(6)=>B_in_DUMMY(6), dout(5)=>B_in_DUMMY(5), dout(4)=>B_in_DUMMY(4),
dout(3)=>B_in_DUMMY(3), dout(2)=>B_in_DUMMY(2), dout(1)=>B_in_DUMMY(1),
dout(0)=>B_in_DUMMY(0), sel=>Sel_ALU_Mux_B_in);

RF_Mux_W_Data : mux32b_4to1
PORT MAP (din0(31)=>ALU_Res_DUMMY(31), din0(30)=>ALU_Res_DUMMY(30),
din0(29)=>ALU_Res_DUMMY(29), din0(28)=>ALU_Res_DUMMY(28),

```



```

din0(27)=>ALU_Res_DUMMY(27), din0(26)=>ALU_Res_DUMMY(26),
din0(25)=>ALU_Res_DUMMY(25), din0(24)=>ALU_Res_DUMMY(24),
din0(23)=>ALU_Res_DUMMY(23), din0(22)=>ALU_Res_DUMMY(22),
din0(21)=>ALU_Res_DUMMY(21), din0(20)=>ALU_Res_DUMMY(20),
din0(19)=>ALU_Res_DUMMY(19), din0(18)=>ALU_Res_DUMMY(18),
din0(17)=>ALU_Res_DUMMY(17), din0(16)=>ALU_Res_DUMMY(16),
din0(15)=>ALU_Res_DUMMY(15), din0(14)=>ALU_Res_DUMMY(14),
din0(13)=>ALU_Res_DUMMY(13), din0(12)=>ALU_Res_DUMMY(12),
din0(11)=>ALU_Res_DUMMY(11), din0(10)=>ALU_Res_DUMMY(10),
din0(9)=>ALU_Res_DUMMY(9), din0(8)=>ALU_Res_DUMMY(8),
din0(7)=>ALU_Res_DUMMY(7), din0(6)=>ALU_Res_DUMMY(6),
din0(5)=>ALU_Res_DUMMY(5), din0(4)=>ALU_Res_DUMMY(4),
din0(3)=>ALU_Res_DUMMY(3), din0(2)=>ALU_Res_DUMMY(2),
din0(1)=>ALU_Res_DUMMY(1), din0(0)=>ALU_Res_DUMMY(0),
din1(31)=>DMem_to_RF_DUMMY(31), din1(30)=>DMem_to_RF_DUMMY(30),
din1(29)=>DMem_to_RF_DUMMY(29), din1(28)=>DMem_to_RF_DUMMY(28),
din1(27)=>DMem_to_RF_DUMMY(27), din1(26)=>DMem_to_RF_DUMMY(26),
din1(25)=>DMem_to_RF_DUMMY(25), din1(24)=>DMem_to_RF_DUMMY(24),
din1(23)=>DMem_to_RF_DUMMY(23), din1(22)=>DMem_to_RF_DUMMY(22),
din1(21)=>DMem_to_RF_DUMMY(21), din1(20)=>DMem_to_RF_DUMMY(20),
din1(19)=>DMem_to_RF_DUMMY(19), din1(18)=>DMem_to_RF_DUMMY(18),
din1(17)=>DMem_to_RF_DUMMY(17), din1(16)=>DMem_to_RF_DUMMY(16),
din1(15)=>DMem_to_RF_DUMMY(15), din1(14)=>DMem_to_RF_DUMMY(14),
din1(13)=>DMem_to_RF_DUMMY(13), din1(12)=>DMem_to_RF_DUMMY(12),
din1(11)=>DMem_to_RF_DUMMY(11), din1(10)=>DMem_to_RF_DUMMY(10),
din1(9)=>DMem_to_RF_DUMMY(9), din1(8)=>DMem_to_RF_DUMMY(8),
din1(7)=>DMem_to_RF_DUMMY(7), din1(6)=>DMem_to_RF_DUMMY(6),
din1(5)=>DMem_to_RF_DUMMY(5), din1(4)=>DMem_to_RF_DUMMY(4),
din1(3)=>DMem_to_RF_DUMMY(3), din1(2)=>DMem_to_RF_DUMMY(2),
din1(1)=>DMem_to_RF_DUMMY(1), din1(0)=>DMem_to_RF_DUMMY(0),
din2(31)=>RF_Write_Din_Preload(31), din2(30)=>RF_Write_Din_Preload(30),
din2(29)=>RF_Write_Din_Preload(29), din2(28)=>RF_Write_Din_Preload(28),
din2(27)=>RF_Write_Din_Preload(27), din2(26)=>RF_Write_Din_Preload(26),
din2(25)=>RF_Write_Din_Preload(25), din2(24)=>RF_Write_Din_Preload(24),
din2(23)=>RF_Write_Din_Preload(23), din2(22)=>RF_Write_Din_Preload(22),
din2(21)=>RF_Write_Din_Preload(21), din2(20)=>RF_Write_Din_Preload(20),
din2(19)=>RF_Write_Din_Preload(19), din2(18)=>RF_Write_Din_Preload(18),
din2(17)=>RF_Write_Din_Preload(17), din2(16)=>RF_Write_Din_Preload(16),
din2(15)=>RF_Write_Din_Preload(15), din2(14)=>RF_Write_Din_Preload(14),
din2(13)=>RF_Write_Din_Preload(13), din2(12)=>RF_Write_Din_Preload(12),
din2(11)=>RF_Write_Din_Preload(11), din2(10)=>RF_Write_Din_Preload(10),
din2(9)=>RF_Write_Din_Preload(9), din2(8)=>RF_Write_Din_Preload(8),
din2(7)=>RF_Write_Din_Preload(7), din2(6)=>RF_Write_Din_Preload(6),
din2(5)=>RF_Write_Din_Preload(5), din2(4)=>RF_Write_Din_Preload(4),
din2(3)=>RF_Write_Din_Preload(3), din2(2)=>RF_Write_Din_Preload(2),
din2(1)=>RF_Write_Din_Preload(1), din2(0)=>RF_Write_Din_Preload(0),
din3(31)=>Value_of_Zero(31), din3(30)=>Value_of_Zero(30),
din3(29)=>Value_of_Zero(29), din3(28)=>Value_of_Zero(28),
din3(27)=>Value_of_Zero(27), din3(26)=>Value_of_Zero(26),
din3(25)=>Value_of_Zero(25), din3(24)=>Value_of_Zero(24),
din3(23)=>Value_of_Zero(23), din3(22)=>Value_of_Zero(22),
din3(21)=>Value_of_Zero(21), din3(20)=>Value_of_Zero(20),
din3(19)=>Value_of_Zero(19), din3(18)=>Value_of_Zero(18),
din3(17)=>Value_of_Zero(17), din3(16)=>Value_of_Zero(16),
din3(15)=>Value_of_Zero(15), din3(14)=>Value_of_Zero(14),
din3(13)=>Value_of_Zero(13), din3(12)=>Value_of_Zero(12),
din3(11)=>Value_of_Zero(11), din3(10)=>Value_of_Zero(10),
din3(9)=>Value_of_Zero(9), din3(8)=>Value_of_Zero(8),
din3(7)=>Value_of_Zero(7), din3(6)=>Value_of_Zero(6),
din3(5)=>Value_of_Zero(5), din3(4)=>Value_of_Zero(4),
din3(3)=>Value_of_Zero(3), din3(2)=>Value_of_Zero(2),
din3(1)=>Value_of_Zero(1), din3(0)=>Value_of_Zero(0),
dout(31)=>RF_Write_Data_DUMMY(31), dout(30)=>RF_Write_Data_DUMMY(30),
dout(29)=>RF_Write_Data_DUMMY(29), dout(28)=>RF_Write_Data_DUMMY(28),
dout(27)=>RF_Write_Data_DUMMY(27), dout(26)=>RF_Write_Data_DUMMY(26),
dout(25)=>RF_Write_Data_DUMMY(25), dout(24)=>RF_Write_Data_DUMMY(24),
dout(23)=>RF_Write_Data_DUMMY(23), dout(22)=>RF_Write_Data_DUMMY(22),
dout(21)=>RF_Write_Data_DUMMY(21), dout(20)=>RF_Write_Data_DUMMY(20),
dout(19)=>RF_Write_Data_DUMMY(19), dout(18)=>RF_Write_Data_DUMMY(18),
dout(17)=>RF_Write_Data_DUMMY(17), dout(16)=>RF_Write_Data_DUMMY(16),
dout(15)=>RF_Write_Data_DUMMY(15), dout(14)=>RF_Write_Data_DUMMY(14),
dout(13)=>RF_Write_Data_DUMMY(13), dout(12)=>RF_Write_Data_DUMMY(12),
dout(11)=>RF_Write_Data_DUMMY(11), dout(10)=>RF_Write_Data_DUMMY(10),

```

```

dout(9)=>RF_Write_Data_DUMMY(9), dout(8)=>RF_Write_Data_DUMMY(8),
dout(7)=>RF_Write_Data_DUMMY(7), dout(6)=>RF_Write_Data_DUMMY(6),
dout(5)=>RF_Write_Data_DUMMY(5), dout(4)=>RF_Write_Data_DUMMY(4),
dout(3)=>RF_Write_Data_DUMMY(3), dout(2)=>RF_Write_Data_DUMMY(2),
dout(1)=>RF_Write_Data_DUMMY(1), dout(0)=>RF_Write_Data_DUMMY(0),
sel(1)=>Mux_RF_din_Sel(1), sel(0)=>Mux_RF_din_Sel(0));

RF_Mux_W_Reg : mux5b_4to1
PORT MAP (din0(4)=>rs_DUMMY(4), din0(3)=>rs_DUMMY(3),
din0(2)=>rs_DUMMY(2), din0(1)=>rs_DUMMY(1), din0(0)=>rs_DUMMY(0),
din1(4)=>rt_DUMMY(4), din1(3)=>rt_DUMMY(3), din1(2)=>rt_DUMMY(2),
din1(1)=>rt_DUMMY(1), din1(0)=>rt_DUMMY(0), din2(4)=>rd_DUMMY(4),
din2(3)=>rd_DUMMY(3), din2(2)=>rd_DUMMY(2), din2(1)=>rd_DUMMY(1),
din2(0)=>rd_DUMMY(0), din3(4)=>RF_Write_Num_Preload(4),
din3(3)=>RF_Write_Num_Preload(3), din3(2)=>RF_Write_Num_Preload(2),
din3(1)=>RF_Write_Num_Preload(1), din3(0)=>RF_Write_Num_Preload(0),
dout(4)=>RF_Write_Reg_DUMMY(4), dout(3)=>RF_Write_Reg_DUMMY(3),
dout(2)=>RF_Write_Reg_DUMMY(2), dout(1)=>RF_Write_Reg_DUMMY(1),
dout(0)=>RF_Write_Reg_DUMMY(0), sel(1)=>Mux_RF_Num_Sel(1),
sel(0)=>Mux_RF_Num_Sel(0));

mux8_2to1_i1 : mux8_2to1
PORT MAP (sel=>Mux_Sel_Inst_RAM_Addr_Src, din0(7)=>Addr_from_PC_DUMMY(7),
din0(6)=>Addr_from_PC_DUMMY(6), din0(5)=>Addr_from_PC_DUMMY(5),
din0(4)=>Addr_from_PC_DUMMY(4), din0(3)=>Addr_from_PC_DUMMY(3),
din0(2)=>Addr_from_PC_DUMMY(2), din0(1)=>Addr_from_PC_DUMMY(1),
din0(0)=>Addr_from_PC_DUMMY(0), din1(7)=>Inst_RAM_Write_Addr_Preload(7),
din1(6)=>Inst_RAM_Write_Addr_Preload(6),
din1(5)=>Inst_RAM_Write_Addr_Preload(5),
din1(4)=>Inst_RAM_Write_Addr_Preload(4),
din1(3)=>Inst_RAM_Write_Addr_Preload(3),
din1(2)=>Inst_RAM_Write_Addr_Preload(2),
din1(1)=>Inst_RAM_Write_Addr_Preload(1),
din1(0)=>Inst_RAM_Write_Addr_Preload(0), dout(7)=>Inst_RAM_Addr_DUMMY(7),
dout(6)=>Inst_RAM_Addr_DUMMY(6), dout(5)=>Inst_RAM_Addr_DUMMY(5),
dout(4)=>Inst_RAM_Addr_DUMMY(4), dout(3)=>Inst_RAM_Addr_DUMMY(3),
dout(2)=>Inst_RAM_Addr_DUMMY(2), dout(1)=>Inst_RAM_Addr_DUMMY(1),
dout(0)=>Inst_RAM_Addr_DUMMY(0));

mux8_2to1_i3 : mux8_2to1
PORT MAP (sel=>Mux_Sel_PCsrc1_DUMMY, din0(7)=>PCplus1_Addr_DUMMY(7),
din0(6)=>PCplus1_Addr_DUMMY(6), din0(5)=>PCplus1_Addr_DUMMY(5),
din0(4)=>PCplus1_Addr_DUMMY(4), din0(3)=>PCplus1_Addr_DUMMY(3),
din0(2)=>PCplus1_Addr_DUMMY(2), din0(1)=>PCplus1_Addr_DUMMY(1),
din0(0)=>PCplus1_Addr_DUMMY(0), din1(7)=>Branch_Target_DUMMY(7),
din1(6)=>Branch_Target_DUMMY(6), din1(5)=>Branch_Target_DUMMY(5),
din1(4)=>Branch_Target_DUMMY(4), din1(3)=>Branch_Target_DUMMY(3),
din1(2)=>Branch_Target_DUMMY(2), din1(1)=>Branch_Target_DUMMY(1),
din1(0)=>Branch_Target_DUMMY(0), dout(7)=>Mux2Mux_PCsrc_DUMMY(7),
dout(6)=>Mux2Mux_PCsrc_DUMMY(6), dout(5)=>Mux2Mux_PCsrc_DUMMY(5),
dout(4)=>Mux2Mux_PCsrc_DUMMY(4), dout(3)=>Mux2Mux_PCsrc_DUMMY(3),
dout(2)=>Mux2Mux_PCsrc_DUMMY(2), dout(1)=>Mux2Mux_PCsrc_DUMMY(1),
dout(0)=>Mux2Mux_PCsrc_DUMMY(0));

mux8_2to1_i2 : mux8_2to1
PORT MAP (sel=>Mux_Sel_PCsrc2, din0(7)=>Mux2Mux_PCsrc_DUMMY(7),
din0(6)=>Mux2Mux_PCsrc_DUMMY(6), din0(5)=>Mux2Mux_PCsrc_DUMMY(5),
din0(4)=>Mux2Mux_PCsrc_DUMMY(4), din0(3)=>Mux2Mux_PCsrc_DUMMY(3),
din0(2)=>Mux2Mux_PCsrc_DUMMY(2), din0(1)=>Mux2Mux_PCsrc_DUMMY(1),
din0(0)=>Mux2Mux_PCsrc_DUMMY(0), din1(7)=>Addr_26to8bits_DUMMY(7),
din1(6)=>Addr_26to8bits_DUMMY(6), din1(5)=>Addr_26to8bits_DUMMY(5),
din1(4)=>Addr_26to8bits_DUMMY(4), din1(3)=>Addr_26to8bits_DUMMY(3),
din1(2)=>Addr_26to8bits_DUMMY(2), din1(1)=>Addr_26to8bits_DUMMY(1),
din1(0)=>Addr_26to8bits_DUMMY(0), dout(7)=>Next_PC_DUMMY(7),
dout(6)=>Next_PC_DUMMY(6), dout(5)=>Next_PC_DUMMY(5),
dout(4)=>Next_PC_DUMMY(4), dout(3)=>Next_PC_DUMMY(3),
dout(2)=>Next_PC_DUMMY(2), dout(1)=>Next_PC_DUMMY(1),
dout(0)=>Next_PC_DUMMY(0));

DMem_Mux_ra : mux8b_2to1
PORT MAP (din0(7)=>DMem_RA_Preload(7), din0(6)=>DMem_RA_Preload(6),
din0(5)=>DMem_RA_Preload(5), din0(4)=>DMem_RA_Preload(4),
din0(3)=>DMem_RA_Preload(3), din0(2)=>DMem_RA_Preload(2),

```

```

din0(1)=>DMem_RA_Preload(1), din0(0)=>DMem_RA_Preload(0),
din1(31)=>ALU_Res_DUMMY(31), din1(30)=>ALU_Res_DUMMY(30),
din1(29)=>ALU_Res_DUMMY(29), din1(28)=>ALU_Res_DUMMY(28),
din1(27)=>ALU_Res_DUMMY(27), din1(26)=>ALU_Res_DUMMY(26),
din1(25)=>ALU_Res_DUMMY(25), din1(24)=>ALU_Res_DUMMY(24),
din1(23)=>ALU_Res_DUMMY(23), din1(22)=>ALU_Res_DUMMY(22),
din1(21)=>ALU_Res_DUMMY(21), din1(20)=>ALU_Res_DUMMY(20),
din1(19)=>ALU_Res_DUMMY(19), din1(18)=>ALU_Res_DUMMY(18),
din1(17)=>ALU_Res_DUMMY(17), din1(16)=>ALU_Res_DUMMY(16),
din1(15)=>ALU_Res_DUMMY(15), din1(14)=>ALU_Res_DUMMY(14),
din1(13)=>ALU_Res_DUMMY(13), din1(12)=>ALU_Res_DUMMY(12),
din1(11)=>ALU_Res_DUMMY(11), din1(10)=>ALU_Res_DUMMY(10),
din1(9)=>ALU_Res_DUMMY(9), din1(8)=>ALU_Res_DUMMY(8),
din1(7)=>ALU_Res_DUMMY(7), din1(6)=>ALU_Res_DUMMY(6),
din1(5)=>ALU_Res_DUMMY(5), din1(4)=>ALU_Res_DUMMY(4),
din1(3)=>ALU_Res_DUMMY(3), din1(2)=>ALU_Res_DUMMY(2),
din1(1)=>ALU_Res_DUMMY(1), din1(0)=>ALU_Res_DUMMY(0),
dout(7)=>Data_RAM_Read_Addr_DUMMY(7),
dout(6)=>Data_RAM_Read_Addr_DUMMY(6),
dout(5)=>Data_RAM_Read_Addr_DUMMY(5),
dout(4)=>Data_RAM_Read_Addr_DUMMY(4),
dout(3)=>Data_RAM_Read_Addr_DUMMY(3),
dout(2)=>Data_RAM_Read_Addr_DUMMY(2),
dout(1)=>Data_RAM_Read_Addr_DUMMY(1),
dout(0)=>Data_RAM_Read_Addr_DUMMY(0), sel=>Sel_DMem_Mux_ra);

DMem_Mux_wa : mux8b_2to1
PORT MAP (din0(7)=>DMem_WA_Preload(7), din0(6)=>DMem_WA_Preload(6),
din0(5)=>DMem_WA_Preload(5), din0(4)=>DMem_WA_Preload(4),
din0(3)=>DMem_WA_Preload(3), din0(2)=>DMem_WA_Preload(2),
din0(1)=>DMem_WA_Preload(1), din0(0)=>DMem_WA_Preload(0),
din1(31)=>ALU_Res_DUMMY(31), din1(30)=>ALU_Res_DUMMY(30),
din1(29)=>ALU_Res_DUMMY(29), din1(28)=>ALU_Res_DUMMY(28),
din1(27)=>ALU_Res_DUMMY(27), din1(26)=>ALU_Res_DUMMY(26),
din1(25)=>ALU_Res_DUMMY(25), din1(24)=>ALU_Res_DUMMY(24),
din1(23)=>ALU_Res_DUMMY(23), din1(22)=>ALU_Res_DUMMY(22),
din1(21)=>ALU_Res_DUMMY(21), din1(20)=>ALU_Res_DUMMY(20),
din1(19)=>ALU_Res_DUMMY(19), din1(18)=>ALU_Res_DUMMY(18),
din1(17)=>ALU_Res_DUMMY(17), din1(16)=>ALU_Res_DUMMY(16),
din1(15)=>ALU_Res_DUMMY(15), din1(14)=>ALU_Res_DUMMY(14),
din1(13)=>ALU_Res_DUMMY(13), din1(12)=>ALU_Res_DUMMY(12),
din1(11)=>ALU_Res_DUMMY(11), din1(10)=>ALU_Res_DUMMY(10),
din1(9)=>ALU_Res_DUMMY(9), din1(8)=>ALU_Res_DUMMY(8),
din1(7)=>ALU_Res_DUMMY(7), din1(6)=>ALU_Res_DUMMY(6),
din1(5)=>ALU_Res_DUMMY(5), din1(4)=>ALU_Res_DUMMY(4),
din1(3)=>ALU_Res_DUMMY(3), din1(2)=>ALU_Res_DUMMY(2),
din1(1)=>ALU_Res_DUMMY(1), din1(0)=>ALU_Res_DUMMY(0),
dout(7)=>Data_RAM_Write_Addr_DUMMY(7),
dout(6)=>Data_RAM_Write_Addr_DUMMY(6),
dout(5)=>Data_RAM_Write_Addr_DUMMY(5),
dout(4)=>Data_RAM_Write_Addr_DUMMY(4),
dout(3)=>Data_RAM_Write_Addr_DUMMY(3),
dout(2)=>Data_RAM_Write_Addr_DUMMY(2),
dout(1)=>Data_RAM_Write_Addr_DUMMY(1),
dout(0)=>Data_RAM_Write_Addr_DUMMY(0), sel=>Sel_DMem_Mux_wa);

Program_Counter : pc
PORT MAP (clk=>Mux_to_PC_Clk_DUMMY, reset=>Reset_PC,
din(7)=>Next_PC_DUMMY(7), din(6)=>Next_PC_DUMMY(6),
din(5)=>Next_PC_DUMMY(5), din(4)=>Next_PC_DUMMY(4),
din(3)=>Next_PC_DUMMY(3), din(2)=>Next_PC_DUMMY(2),
din(1)=>Next_PC_DUMMY(1), din(0)=>Next_PC_DUMMY(0),
dout(7)=>Addr_from_PC_DUMMY(7), dout(6)=>Addr_from_PC_DUMMY(6),
dout(5)=>Addr_from_PC_DUMMY(5), dout(4)=>Addr_from_PC_DUMMY(4),
dout(3)=>Addr_from_PC_DUMMY(3), dout(2)=>Addr_from_PC_DUMMY(2),
dout(1)=>Addr_from_PC_DUMMY(1), dout(0)=>Addr_from_PC_DUMMY(0));

Register_File : rf_synch
PORT MAP (clk=>Clk0_DUMMY, enable=>RF_En_Read_Write,
read_reg1(4)=>rs_DUMMY(4), read_reg1(3)=>rs_DUMMY(3),
read_reg1(2)=>rs_DUMMY(2), read_reg1(1)=>rs_DUMMY(1),
read_reg1(0)=>rs_DUMMY(0), read_reg2(4)=>rt_DUMMY(4),
read_reg2(3)=>rt_DUMMY(3), read_reg2(2)=>rt_DUMMY(2),

```

```

read_reg2(1)=>rt_DUMMY(1), read_reg2(0)=>rt_DUMMY(0),
write_data(31)=>RF_Write_Data_DUMMY(31),
write_data(30)=>RF_Write_Data_DUMMY(30),
write_data(29)=>RF_Write_Data_DUMMY(29),
write_data(28)=>RF_Write_Data_DUMMY(28),
write_data(27)=>RF_Write_Data_DUMMY(27),
write_data(26)=>RF_Write_Data_DUMMY(26),
write_data(25)=>RF_Write_Data_DUMMY(25),
write_data(24)=>RF_Write_Data_DUMMY(24),
write_data(23)=>RF_Write_Data_DUMMY(23),
write_data(22)=>RF_Write_Data_DUMMY(22),
write_data(21)=>RF_Write_Data_DUMMY(21),
write_data(20)=>RF_Write_Data_DUMMY(20),
write_data(19)=>RF_Write_Data_DUMMY(19),
write_data(18)=>RF_Write_Data_DUMMY(18),
write_data(17)=>RF_Write_Data_DUMMY(17),
write_data(16)=>RF_Write_Data_DUMMY(16),
write_data(15)=>RF_Write_Data_DUMMY(15),
write_data(14)=>RF_Write_Data_DUMMY(14),
write_data(13)=>RF_Write_Data_DUMMY(13),
write_data(12)=>RF_Write_Data_DUMMY(12),
write_data(11)=>RF_Write_Data_DUMMY(11),
write_data(10)=>RF_Write_Data_DUMMY(10),
write_data(9)=>RF_Write_Data_DUMMY(9),
write_data(8)=>RF_Write_Data_DUMMY(8),
write_data(7)=>RF_Write_Data_DUMMY(7),
write_data(6)=>RF_Write_Data_DUMMY(6),
write_data(5)=>RF_Write_Data_DUMMY(5),
write_data(4)=>RF_Write_Data_DUMMY(4),
write_data(3)=>RF_Write_Data_DUMMY(3),
write_data(2)=>RF_Write_Data_DUMMY(2),
write_data(1)=>RF_Write_Data_DUMMY(1),
write_data(0)=>RF_Write_Data_DUMMY(0),
write_reg(4)=>RF_Write_Reg_DUMMY(4), write_reg(3)=>RF_Write_Reg_DUMMY(3),
write_reg(2)=>RF_Write_Reg_DUMMY(2), write_reg(1)=>RF_Write_Reg_DUMMY(1),
write_reg(0)=>RF_Write_Reg_DUMMY(0), read_data_1(31)=>A_in_DUMMY(31),
read_data_1(30)=>A_in_DUMMY(30), read_data_1(29)=>A_in_DUMMY(29),
read_data_1(28)=>A_in_DUMMY(28), read_data_1(27)=>A_in_DUMMY(27),
read_data_1(26)=>A_in_DUMMY(26), read_data_1(25)=>A_in_DUMMY(25),
read_data_1(24)=>A_in_DUMMY(24), read_data_1(23)=>A_in_DUMMY(23),
read_data_1(22)=>A_in_DUMMY(22), read_data_1(21)=>A_in_DUMMY(21),
read_data_1(20)=>A_in_DUMMY(20), read_data_1(19)=>A_in_DUMMY(19),
read_data_1(18)=>A_in_DUMMY(18), read_data_1(17)=>A_in_DUMMY(17),
read_data_1(16)=>A_in_DUMMY(16), read_data_1(15)=>A_in_DUMMY(15),
read_data_1(14)=>A_in_DUMMY(14), read_data_1(13)=>A_in_DUMMY(13),
read_data_1(12)=>A_in_DUMMY(12), read_data_1(11)=>A_in_DUMMY(11),
read_data_1(10)=>A_in_DUMMY(10), read_data_1(9)=>A_in_DUMMY(9),
read_data_1(8)=>A_in_DUMMY(8), read_data_1(7)=>A_in_DUMMY(7),
read_data_1(6)=>A_in_DUMMY(6), read_data_1(5)=>A_in_DUMMY(5),
read_data_1(4)=>A_in_DUMMY(4), read_data_1(3)=>A_in_DUMMY(3),
read_data_1(2)=>A_in_DUMMY(2), read_data_1(1)=>A_in_DUMMY(1),
read_data_1(0)=>A_in_DUMMY(0), read_data_2(31)=>RF_Data_B_DUMMY(31),
read_data_2(30)=>RF_Data_B_DUMMY(30),
read_data_2(29)=>RF_Data_B_DUMMY(29),
read_data_2(28)=>RF_Data_B_DUMMY(28),
read_data_2(27)=>RF_Data_B_DUMMY(27),
read_data_2(26)=>RF_Data_B_DUMMY(26),
read_data_2(25)=>RF_Data_B_DUMMY(25),
read_data_2(24)=>RF_Data_B_DUMMY(24),
read_data_2(23)=>RF_Data_B_DUMMY(23),
read_data_2(22)=>RF_Data_B_DUMMY(22),
read_data_2(21)=>RF_Data_B_DUMMY(21),
read_data_2(20)=>RF_Data_B_DUMMY(20),
read_data_2(19)=>RF_Data_B_DUMMY(19),
read_data_2(18)=>RF_Data_B_DUMMY(18),
read_data_2(17)=>RF_Data_B_DUMMY(17),
read_data_2(16)=>RF_Data_B_DUMMY(16),
read_data_2(15)=>RF_Data_B_DUMMY(15),
read_data_2(14)=>RF_Data_B_DUMMY(14),
read_data_2(13)=>RF_Data_B_DUMMY(13),
read_data_2(12)=>RF_Data_B_DUMMY(12),
read_data_2(11)=>RF_Data_B_DUMMY(11),
read_data_2(10)=>RF_Data_B_DUMMY(10), read_data_2(9)=>RF_Data_B_DUMMY(9),

```

```

    read_data_2(8)=>RF_Data_B_DUMMY(8), read_data_2(7)=>RF_Data_B_DUMMY(7),
    read_data_2(6)=>RF_Data_B_DUMMY(6), read_data_2(5)=>RF_Data_B_DUMMY(5),
    read_data_2(4)=>RF_Data_B_DUMMY(4), read_data_2(3)=>RF_Data_B_DUMMY(3),
    read_data_2(2)=>RF_Data_B_DUMMY(2), read_data_2(1)=>RF_Data_B_DUMMY(1),
    read_data_2(0)=>RF_Data_B_DUMMY(0));

sign_dextend : sign_dextension
PORT MAP (addr_in(15)=>Addr_16bits_DUMMY(15),
addr_in(14)=>Addr_16bits_DUMMY(14), addr_in(13)=>Addr_16bits_DUMMY(13),
addr_in(12)=>Addr_16bits_DUMMY(12), addr_in(11)=>Addr_16bits_DUMMY(11),
addr_in(10)=>Addr_16bits_DUMMY(10), addr_in(9)=>Addr_16bits_DUMMY(9),
addr_in(8)=>Addr_16bits_DUMMY(8), addr_in(7)=>Addr_16bits_DUMMY(7),
addr_in(6)=>Addr_16bits_DUMMY(6), addr_in(5)=>Addr_16bits_DUMMY(5),
addr_in(4)=>Addr_16bits_DUMMY(4), addr_in(3)=>Addr_16bits_DUMMY(3),
addr_in(2)=>Addr_16bits_DUMMY(2), addr_in(1)=>Addr_16bits_DUMMY(1),
addr_in(0)=>Addr_16bits_DUMMY(0), addr_out(7)=>Addr_16to8bits_DUMMY(7),
addr_out(6)=>Addr_16to8bits_DUMMY(6),
addr_out(5)=>Addr_16to8bits_DUMMY(5),
addr_out(4)=>Addr_16to8bits_DUMMY(4),
addr_out(3)=>Addr_16to8bits_DUMMY(3),
addr_out(2)=>Addr_16to8bits_DUMMY(2),
addr_out(1)=>Addr_16to8bits_DUMMY(1),
addr_out(0)=>Addr_16to8bits_DUMMY(0));

sign_extend : sign_extension
PORT MAP (addr_in(15)=>Addr_16bits_DUMMY(15),
addr_in(14)=>Addr_16bits_DUMMY(14), addr_in(13)=>Addr_16bits_DUMMY(13),
addr_in(12)=>Addr_16bits_DUMMY(12), addr_in(11)=>Addr_16bits_DUMMY(11),
addr_in(10)=>Addr_16bits_DUMMY(10), addr_in(9)=>Addr_16bits_DUMMY(9),
addr_in(8)=>Addr_16bits_DUMMY(8), addr_in(7)=>Addr_16bits_DUMMY(7),
addr_in(6)=>Addr_16bits_DUMMY(6), addr_in(5)=>Addr_16bits_DUMMY(5),
addr_in(4)=>Addr_16bits_DUMMY(4), addr_in(3)=>Addr_16bits_DUMMY(3),
addr_in(2)=>Addr_16bits_DUMMY(2), addr_in(1)=>Addr_16bits_DUMMY(1),
addr_in(0)=>Addr_16bits_DUMMY(0),
addr_out(31)=>Addr_16to32bits_DUMMY(31),
addr_out(30)=>Addr_16to32bits_DUMMY(30),
addr_out(29)=>Addr_16to32bits_DUMMY(29),
addr_out(28)=>Addr_16to32bits_DUMMY(28),
addr_out(27)=>Addr_16to32bits_DUMMY(27),
addr_out(26)=>Addr_16to32bits_DUMMY(26),
addr_out(25)=>Addr_16to32bits_DUMMY(25),
addr_out(24)=>Addr_16to32bits_DUMMY(24),
addr_out(23)=>Addr_16to32bits_DUMMY(23),
addr_out(22)=>Addr_16to32bits_DUMMY(22),
addr_out(21)=>Addr_16to32bits_DUMMY(21),
addr_out(20)=>Addr_16to32bits_DUMMY(20),
addr_out(19)=>Addr_16to32bits_DUMMY(19),
addr_out(18)=>Addr_16to32bits_DUMMY(18),
addr_out(17)=>Addr_16to32bits_DUMMY(17),
addr_out(16)=>Addr_16to32bits_DUMMY(16),
addr_out(15)=>Addr_16to32bits_DUMMY(15),
addr_out(14)=>Addr_16to32bits_DUMMY(14),
addr_out(13)=>Addr_16to32bits_DUMMY(13),
addr_out(12)=>Addr_16to32bits_DUMMY(12),
addr_out(11)=>Addr_16to32bits_DUMMY(11),
addr_out(10)=>Addr_16to32bits_DUMMY(10),
addr_out(9)=>Addr_16to32bits_DUMMY(9),
addr_out(8)=>Addr_16to32bits_DUMMY(8),
addr_out(7)=>Addr_16to32bits_DUMMY(7),
addr_out(6)=>Addr_16to32bits_DUMMY(6),
addr_out(5)=>Addr_16to32bits_DUMMY(5),
addr_out(4)=>Addr_16to32bits_DUMMY(4),
addr_out(3)=>Addr_16to32bits_DUMMY(3),
addr_out(2)=>Addr_16to32bits_DUMMY(2),
addr_out(1)=>Addr_16to32bits_DUMMY(1),
addr_out(0)=>Addr_16to32bits_DUMMY(0));

TBuf_32b : tristate_32b
PORT MAP (enable=>Mem_Read, din(31)=>DRAM_out_DUMMY(31),
din(30)=>DRAM_out_DUMMY(30), din(29)=>DRAM_out_DUMMY(29),
din(28)=>DRAM_out_DUMMY(28), din(27)=>DRAM_out_DUMMY(27),
din(26)=>DRAM_out_DUMMY(26), din(25)=>DRAM_out_DUMMY(25),
din(24)=>DRAM_out_DUMMY(24), din(23)=>DRAM_out_DUMMY(23),

```

```

din(22)=>DRAM_out_DUMMY(22), din(21)=>DRAM_out_DUMMY(21),
din(20)=>DRAM_out_DUMMY(20), din(19)=>DRAM_out_DUMMY(19),
din(18)=>DRAM_out_DUMMY(18), din(17)=>DRAM_out_DUMMY(17),
din(16)=>DRAM_out_DUMMY(16), din(15)=>DRAM_out_DUMMY(15),
din(14)=>DRAM_out_DUMMY(14), din(13)=>DRAM_out_DUMMY(13),
din(12)=>DRAM_out_DUMMY(12), din(11)=>DRAM_out_DUMMY(11),
din(10)=>DRAM_out_DUMMY(10), din(9)=>DRAM_out_DUMMY(9),
din(8)=>DRAM_out_DUMMY(8), din(7)=>DRAM_out_DUMMY(7),
din(6)=>DRAM_out_DUMMY(6), din(5)=>DRAM_out_DUMMY(5),
din(4)=>DRAM_out_DUMMY(4), din(3)=>DRAM_out_DUMMY(3),
din(2)=>DRAM_out_DUMMY(2), din(1)=>DRAM_out_DUMMY(1),
din(0)=>DRAM_out_DUMMY(0), dout(31)=>DMem_to_RF_DUMMY(31),
dout(30)=>DMem_to_RF_DUMMY(30), dout(29)=>DMem_to_RF_DUMMY(29),
dout(28)=>DMem_to_RF_DUMMY(28), dout(27)=>DMem_to_RF_DUMMY(27),
dout(26)=>DMem_to_RF_DUMMY(26), dout(25)=>DMem_to_RF_DUMMY(25),
dout(24)=>DMem_to_RF_DUMMY(24), dout(23)=>DMem_to_RF_DUMMY(23),
dout(22)=>DMem_to_RF_DUMMY(22), dout(21)=>DMem_to_RF_DUMMY(21),
dout(20)=>DMem_to_RF_DUMMY(20), dout(19)=>DMem_to_RF_DUMMY(19),
dout(18)=>DMem_to_RF_DUMMY(18), dout(17)=>DMem_to_RF_DUMMY(17),
dout(16)=>DMem_to_RF_DUMMY(16), dout(15)=>DMem_to_RF_DUMMY(15),
dout(14)=>DMem_to_RF_DUMMY(14), dout(13)=>DMem_to_RF_DUMMY(13),
dout(12)=>DMem_to_RF_DUMMY(12), dout(11)=>DMem_to_RF_DUMMY(11),
dout(10)=>DMem_to_RF_DUMMY(10), dout(9)=>DMem_to_RF_DUMMY(9),
dout(8)=>DMem_to_RF_DUMMY(8), dout(7)=>DMem_to_RF_DUMMY(7),
dout(6)=>DMem_to_RF_DUMMY(6), dout(5)=>DMem_to_RF_DUMMY(5),
dout(4)=>DMem_to_RF_DUMMY(4), dout(3)=>DMem_to_RF_DUMMY(3),
dout(2)=>DMem_to_RF_DUMMY(2), dout(1)=>DMem_to_RF_DUMMY(1),
dout(0)=>DMem_to_RF_DUMMY(0));

truncator26to8bit : truncator26to8bits
PORT MAP (addr_in(25)=>Addr_26bits_DUMMY(25),
addr_in(24)=>Addr_26bits_DUMMY(24), addr_in(23)=>Addr_26bits_DUMMY(23),
addr_in(22)=>Addr_26bits_DUMMY(22), addr_in(21)=>Addr_26bits_DUMMY(21),
addr_in(20)=>Addr_26bits_DUMMY(20), addr_in(19)=>Addr_26bits_DUMMY(19),
addr_in(18)=>Addr_26bits_DUMMY(18), addr_in(17)=>Addr_26bits_DUMMY(17),
addr_in(16)=>Addr_26bits_DUMMY(16), addr_in(15)=>Addr_26bits_DUMMY(15),
addr_in(14)=>Addr_26bits_DUMMY(14), addr_in(13)=>Addr_26bits_DUMMY(13),
addr_in(12)=>Addr_26bits_DUMMY(12), addr_in(11)=>Addr_26bits_DUMMY(11),
addr_in(10)=>Addr_26bits_DUMMY(10), addr_in(9)=>Addr_26bits_DUMMY(9),
addr_in(8)=>Addr_26bits_DUMMY(8), addr_in(7)=>Addr_26bits_DUMMY(7),
addr_in(6)=>Addr_26bits_DUMMY(6), addr_in(5)=>Addr_26bits_DUMMY(5),
addr_in(4)=>Addr_26bits_DUMMY(4), addr_in(3)=>Addr_26bits_DUMMY(3),
addr_in(2)=>Addr_26bits_DUMMY(2), addr_in(1)=>Addr_26bits_DUMMY(1),
addr_in(0)=>Addr_26bits_DUMMY(0), addr_out(7)=>Addr_26to8bits_DUMMY(7),
addr_out(6)=>Addr_26to8bits_DUMMY(6),
addr_out(5)=>Addr_26to8bits_DUMMY(5),
addr_out(4)=>Addr_26to8bits_DUMMY(4),
addr_out(3)=>Addr_26to8bits_DUMMY(3),
addr_out(2)=>Addr_26to8bits_DUMMY(2),
addr_out(1)=>Addr_26to8bits_DUMMY(1),
addr_out(0)=>Addr_26to8bits_DUMMY(0));

END SCHEMATIC;

```

➤ Synthesis Results

Using the Xilinx ISE synthesis tools, the hardware implementation for the complete datapath with DCM was generated. Figure B.53 shows the resulting top level RTL symbol while figure B.54 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are covered in detail in Appendix A and section B.4 previously.

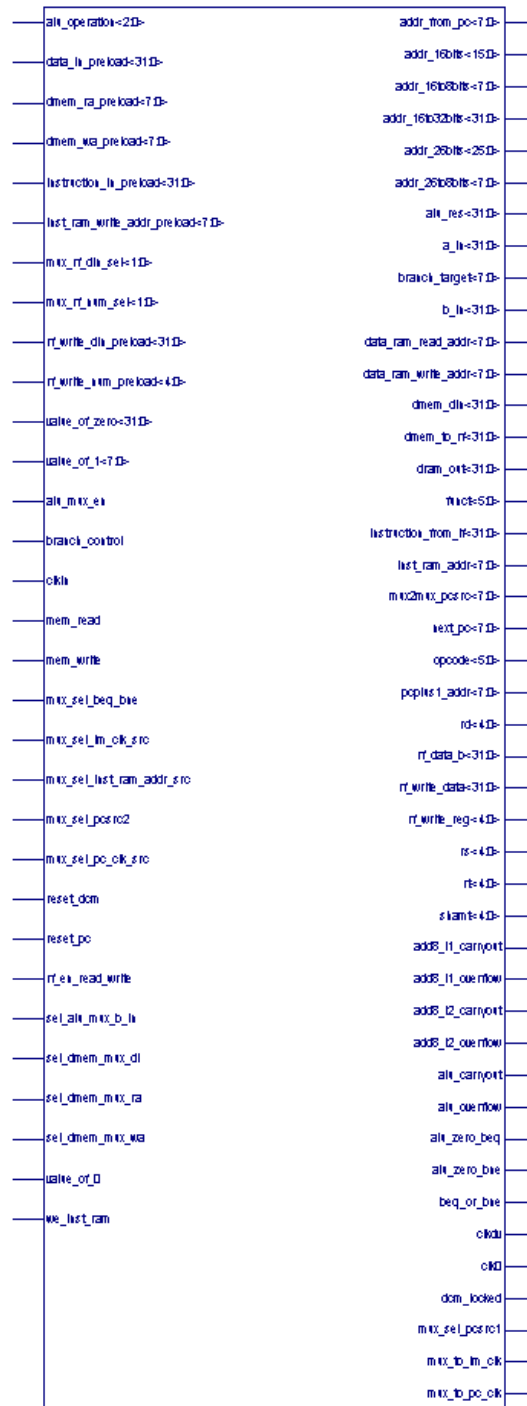


Figure B.53 Resulting top level RTL symbol for the synthesized complete datapath with DCM.

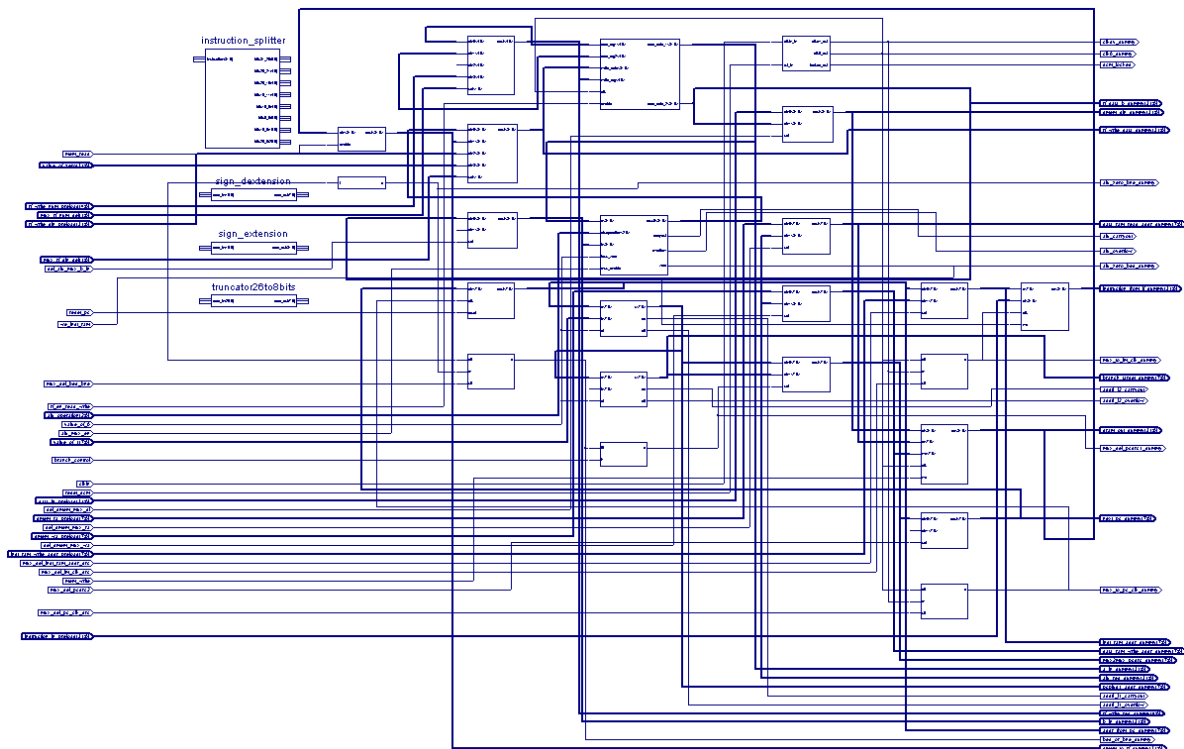


Figure B.54 Resulting top level RTL schematic for the synthesized complete datapath with DCM.

➤ FPGA Device Synthesis Summary

After the hardware implementation for this complete datapath with 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                      : 685

Macro Statistics:
# RAM                      : 4
#   256x32-bit dual-port block RAM: 1
#   256x32-bit single-port block RAM: 1
#   32x32-bit dual-port block RAM: 2
# Registers                : 1
#   8-bit register         : 1
# Tristates                : 23
#   32-bit tristate buffer : 9
#   5-bit tristate buffer  : 4
#   8-bit tristate buffer  : 10

Cell Usage:
# BELS                     : 801
#   and2                   : 100
#   and2b1                 : 35
#   and3                   : 64
#   and3b1                 : 64
#   GND                    : 8
#   inv                    : 34
#   LUT1                   : 40
#   LUT1_L                 : 32
#   LUT2                   : 8
#   LUT2_L                 : 1
#   LUT3                   : 50
```



```

#      LUT3_L           : 11
#      LUT4             : 21
#      LUT4_D           : 10
#      LUT4_L           : 30
#      muxcy            : 4
#      muxcy_d          : 2
#      muxcy_l          : 18
#      muxf5            : 32
#      or2              : 164
#      VCC              : 5
#      xor2             : 20
#      xor3             : 32
#      xorcy            : 16
# FlipFlops/Latches    : 8
#      FDC              : 8
# RAMS                 : 4
#      RAMB16_S36       : 1
#      RAMB16_S36_S36   : 3
# Tri-States           : 356
#      BUFT             : 356
# Clock Buffers        : 2
#      bufg             : 2
# IO Buffers           : 685
#      IBUF             : 190
#      ibufg            : 1
#      OBUF             : 462
#      OBUFT            : 32
# DCMs                 : 1
#      dcm              : 1
# Logical              : 8
#      nor4             : 8
# Others               : 24
#      fmap             : 24

```

Device utilization summary:

Number of Slices:	116	out of	46592	0%
Number of Slice Flip Flops:	8	out of	93184	0%
Number of 4 input LUTs:	203	out of	93184	0%
Number of bonded IOBs:	685	out of	1108	61%
Number of TBUFs:	356	out of	23296	1%
Number of BRAMs:	4	out of	168	2%
Number of GCLKs:	2	out of	16	12%
Number of DCMs:	1	out of	12	8%

Timing Summary:

Minimum period: 24.522ns (Maximum Frequency: 40.780MHz)
 Minimum input arrival time before clock: 38.489ns
 Maximum output required time after clock: 42.695ns
 Maximum combinational path delay: 44.226ns

➤ Simulation Results

➤ Simulation for AND

Figure B.55 shows the simulation waveforms for ALU Operation = AND = $(000)_{\text{binary}} = (0)_{\text{decimal}}$. The instruction simulated is:

<u>AND</u>	\$R7,	\$R5,	\$R6
-----	-----	-----	
<i>rd</i>	<i>rs</i>	<i>rt</i>	

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	100100
-----	-----	-----	-----	-----	-----
<i>op</i> =0	<i>rs</i> =\$R5	<i>rt</i> =\$R6	<i>rd</i> =\$R7	<i>shamt</i>	<i>funct</i> =36

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00000000101001100011100000100100)_2 = (A63824)_{\text{hex}}$$

□ Conclusions:

These resulting waveforms are in line with those for section B.3 and B.4 and prove that this complete datapath with DCM is functioning as expected for an AND instruction.

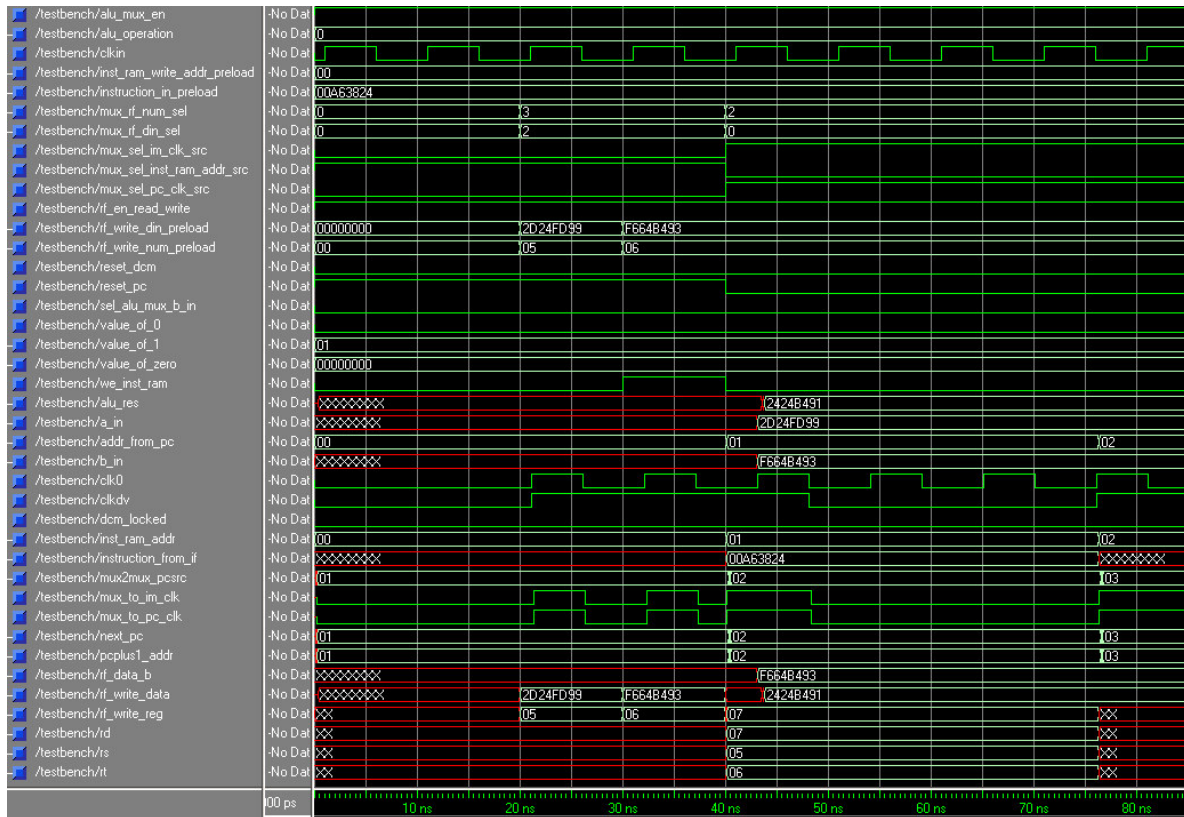


Figure B.55 Results of simulating the synthesized complete datapath with DCM, using ModelSim, for ALU Operation = AND.

➤ Simulation for LW

Figure B.56 shows the simulation waveforms for LW by selecting ALU Operation = ADD = $(010)_{\text{binary}} = (2)_{\text{decimal}}$. There is no ALU Operation specifically for LW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address from which to load the data.

The instruction simulated is:

<u>LW</u>	\$R6 ,	10	(\$R5)
	-----	-----	
	rt	offset	(rs)

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

```

100011 00101 00110 0000000000001010
-----
op=35  rs=$R5  rt=$R6      offset=10

```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101001100000000000001010)_2 = (8CA6000A)_{\text{hex}}$$

□ Conclusions:

These resulting waveforms are in line with those for section B.3 and B.4 and prove that this complete datapath with DCM is functioning as expected for a LW instruction.

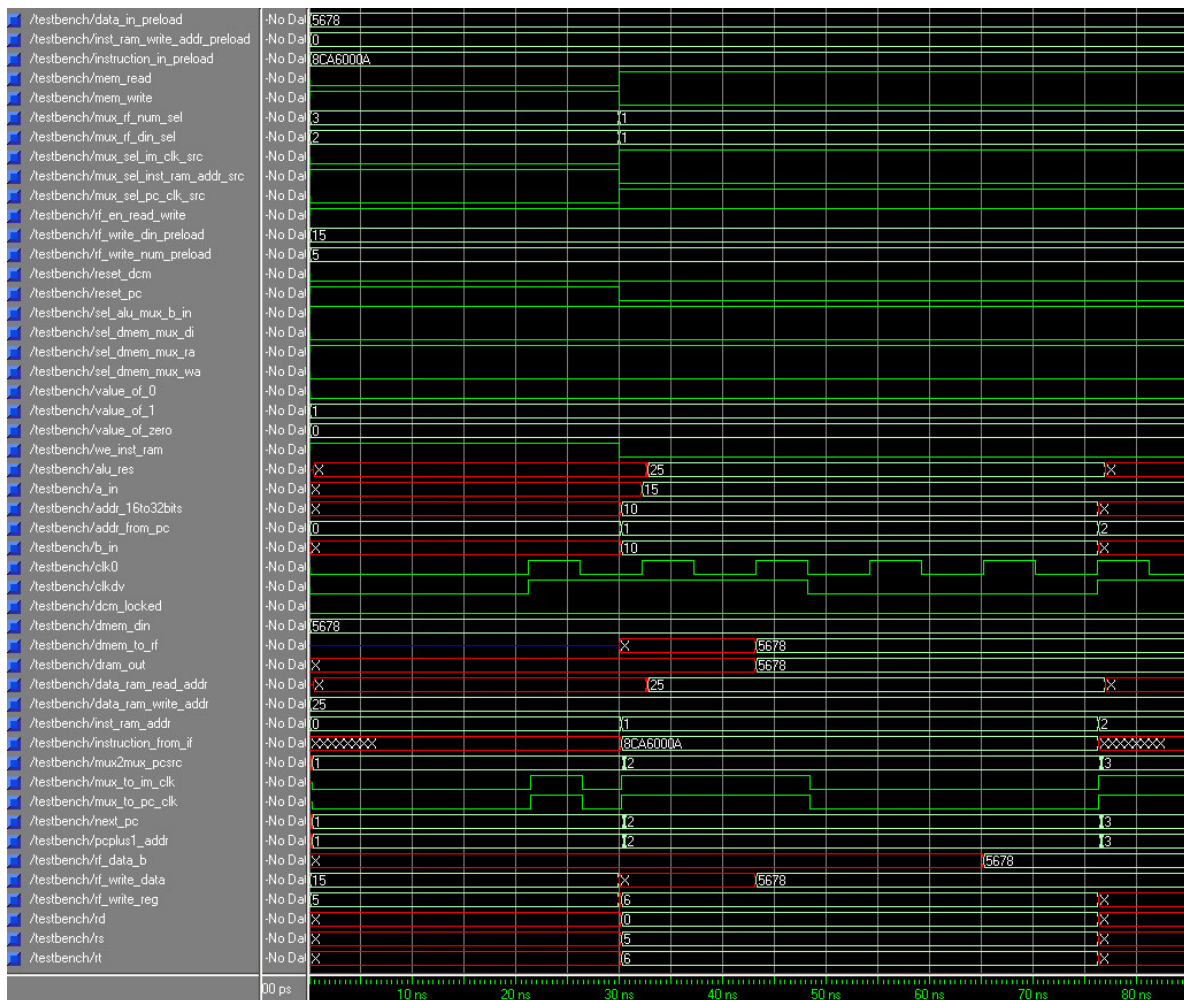


Figure B.56 Results of simulating the synthesized complete datapath with DCM for LW instruction, using ModelSim, with ALU Operation = ADD.

➤ **Simulation for SW**

Figure B.57 shows the simulation waveforms for SW by selecting ALU Operation = ADD = (010)_{binary} = (2)_{decimal}. Similar to LW, there is no ALU Operation specifically for SW, but instead an ADD is all what is needed since the ALU performs an addition (base register + constant/offset supplied in the instruction) to calculate the target memory address for storing the data.

The instruction simulated is:

<u>SW</u>	\$R6 ,	10	(\$R5)
	-----	-----	
	<i>rt</i>	<i>offset</i>	<i>(rs)</i>

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

101011	00101	00110	0000000000001010
-----	-----	-----	-----
<i>op=43</i>	<i>rs=\$R5</i>	<i>rt=\$R6</i>	<i>offset=10</i>

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(10101100101001100000000000001010)_2 = (ACA6000A)_{\text{hex}}$

❑ **Conclusions:**

These resulting waveforms are in line with those for section B.3 and prove that this complete datapath with DCM is functioning as expected for a SW instruction.

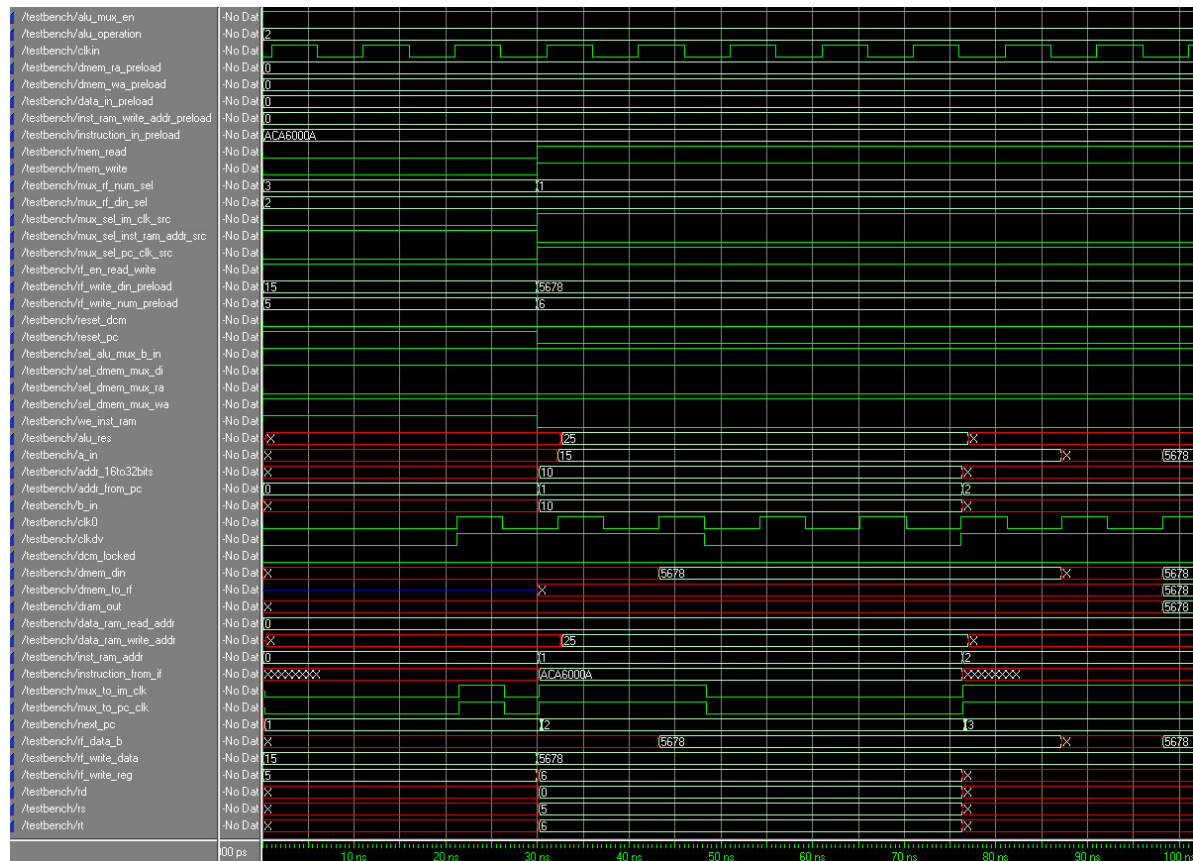


Figure B.57 Results of simulating the synthesized complete datapath with DCM for SW instruction, using ModelSim, with ALU Operation = ADD.

➤ **Simulation for Sample Code No. 1 (SLT, SW, then LW)**

Figure B.58 shows the simulation waveforms for Sample Code No.1. This code is as follows:

Instruction Memory Location No.	Instruction Loaded			Comments
0	SLT	\$R7 ,	\$R5 , \$R6	$[\$R5] < [\$R6] \Rightarrow [\$R7] = 1$
1	SW	\$R7 ,	10 (\$R5)	
2	LW	\$R8 ,	10 (\$R5)	

This is assembled as follows:

□	<u>SLT</u>	\$R7,	\$R5,	\$R6
		-----	-----	-----
		<i>rd</i>	<i>rs</i>	<i>rt</i>

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	101010
-----	-----	-----	-----	-----	-----
<i>op</i> =0	<i>rs</i> =\$R5	<i>rt</i> =\$R6	<i>rd</i> =\$R7	<i>shamt</i>	<i>funct</i> =42

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00000000101001100011100000101010)_2 = (A6382A)_{\text{hex}}$$

$$\square \quad \underline{\text{SW}} \quad \begin{array}{ccc} \$R7, & 10 & (\$R5) \\ \hline & & \\ & rt & \text{offset } (rs) \end{array}$$

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

$$\begin{array}{cccc} 101011 & 00101 & 00111 & 0000000000001010 \\ \hline & & & \\ op=43 & rs=\$R5 & rt=\$R7 & offset=10 \end{array}$$

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10101100101001110000000000001010)_2 = (ACA7000A)_{\text{hex}}$$

$$\square \quad \underline{\text{LW}} \quad \begin{array}{ccc} \$R8, & 10 & (\$R5) \\ \hline & & \\ & rt & \text{offset } (rs) \end{array}$$

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

$$\begin{array}{cccc} 100011 & 00101 & 01000 & 0000000000001010 \\ \hline & & & \\ op=35 & rs=\$R5 & rt=\$R6 & offset=10 \end{array}$$

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101010000000000000001010)_2 = (8CA8000A)_{\text{hex}}$$

□ **Conclusions:**

- Prior to running this sample code, the data is preloaded sequentially into the RF while the instructions are also preloaded sequentially into the instruction memory. The instruction memory simulates main memory without any cache memory between it and the datapath. Therefore, when execution of the code starts, instructions are fetched sequentially from the instruction memory.
- It is worth noting here that data preloading is happening in parallel with instruction preloading, prior to running the code.
- These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for this sample code.

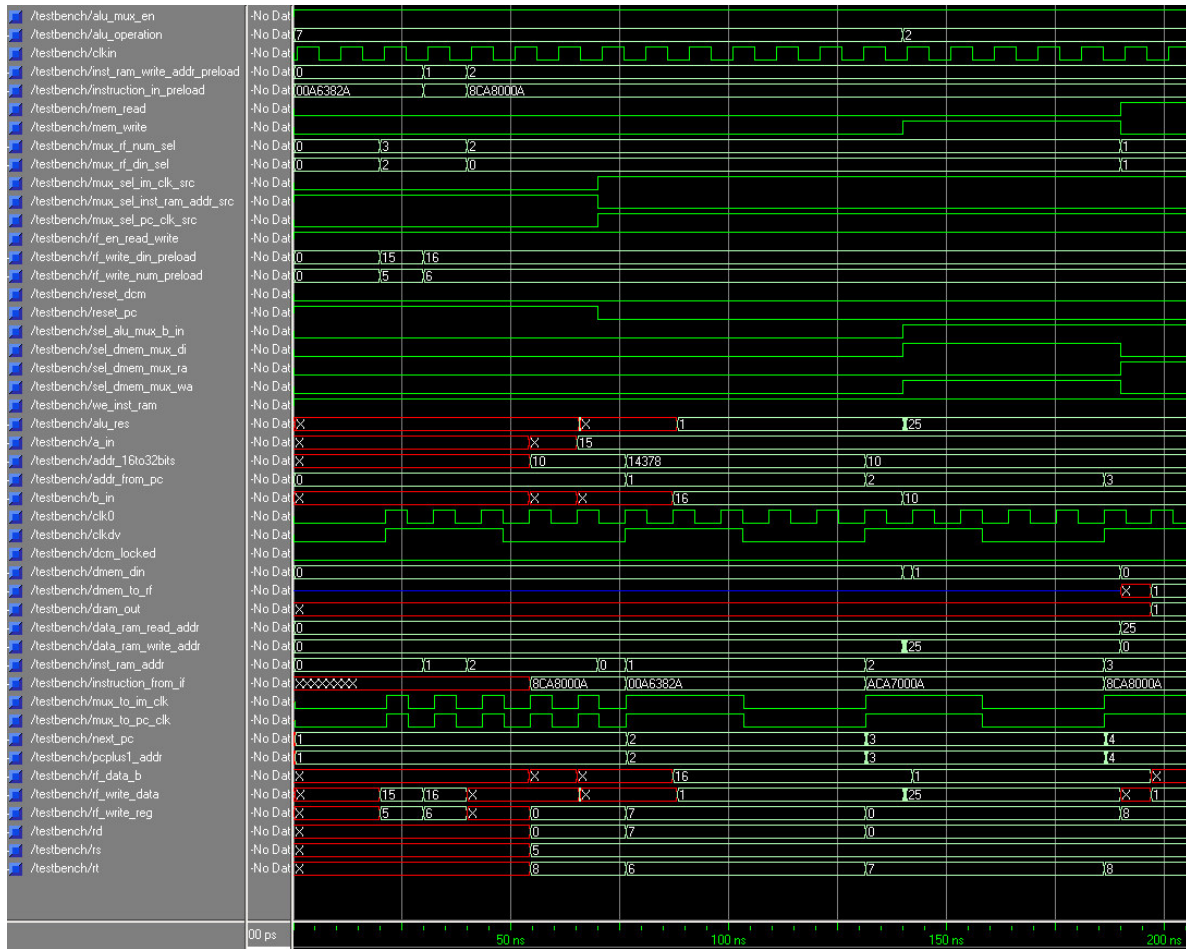


Figure B.58 Results of simulating the synthesized complete datapath with DCM for Sample Code No.1 (SLT, SW, then LW)

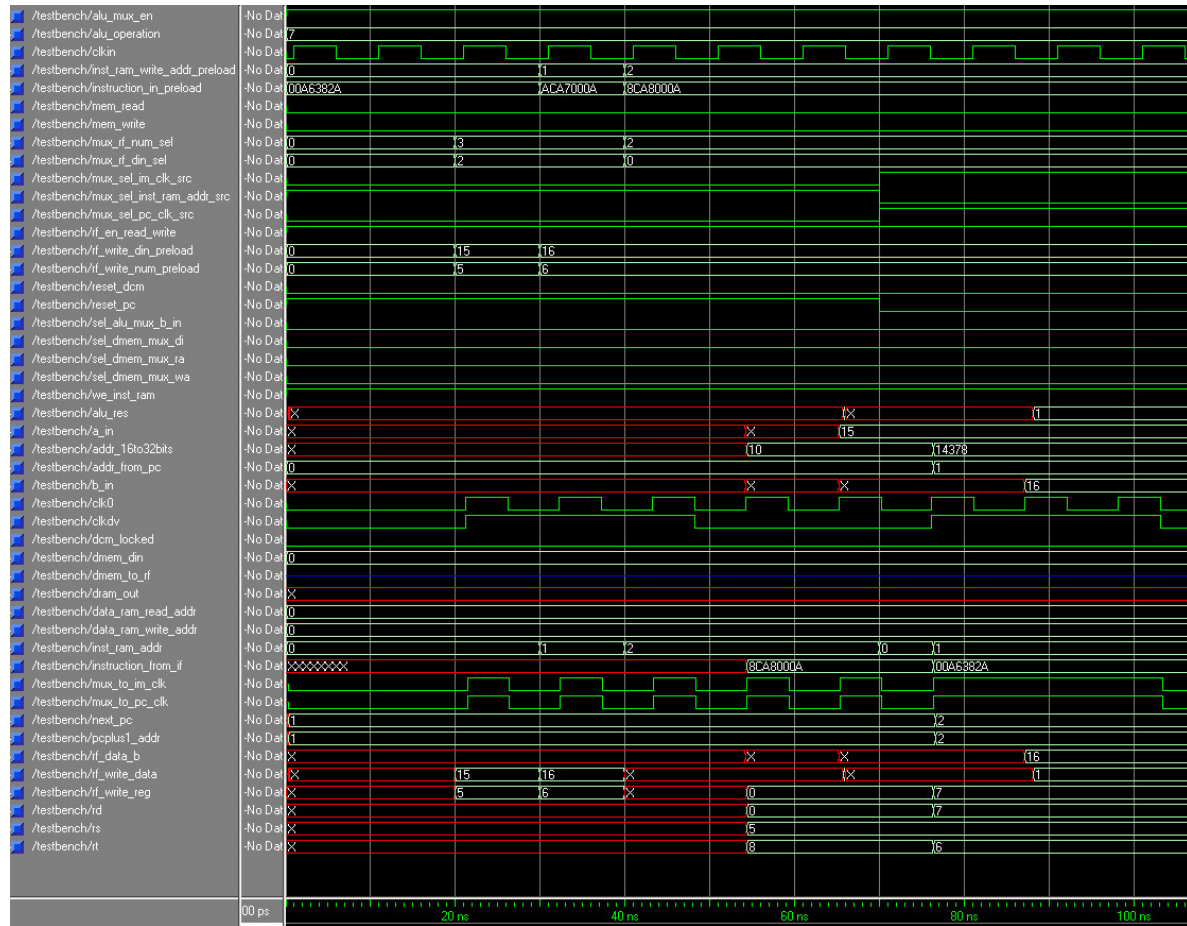


Figure B.58A Results of simulating the synthesized complete datapath with DCM for Sample Code No.1 (SLT, SW, then LW) – This is the first half.

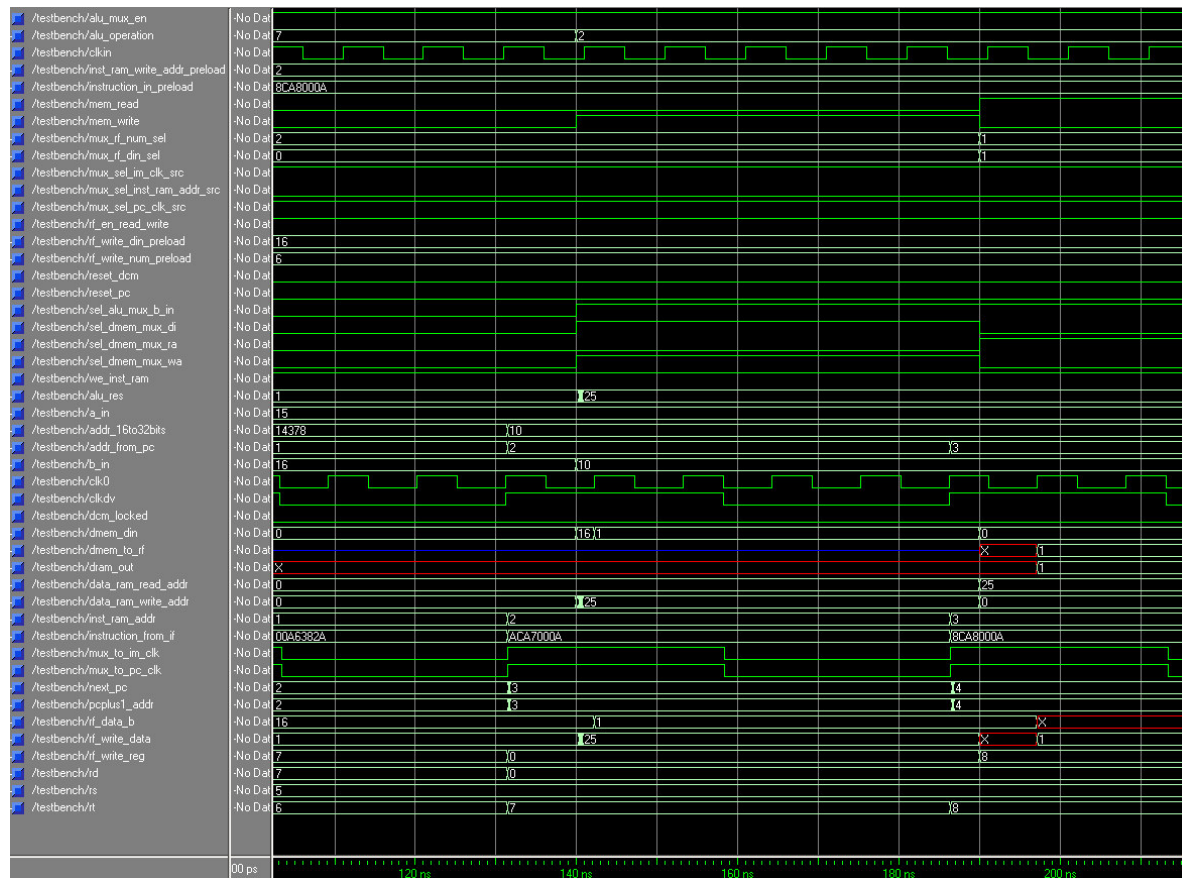


Figure B.58B Results of simulating the synthesized complete datapath with DCM for Sample Code No.1 (SLT, SW, then LW) – This is the second half.

➤ *Simulation for J*

Figure B.59 shows the simulation waveforms for J (Jump). Because the Jump instruction is involved only with control transfer, it does not involve the ALU at all and, therefore, has no ALU Operation.

The instruction simulated is:

J 100

address

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

```
000010 000000000000000000001100100
-----
op=2  address=100
```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

[illegible]

Conclusions:

These resulting waveforms are in line with the functionality expected and prove that this complete datapath with DCM is functioning as expected for a J instruction.

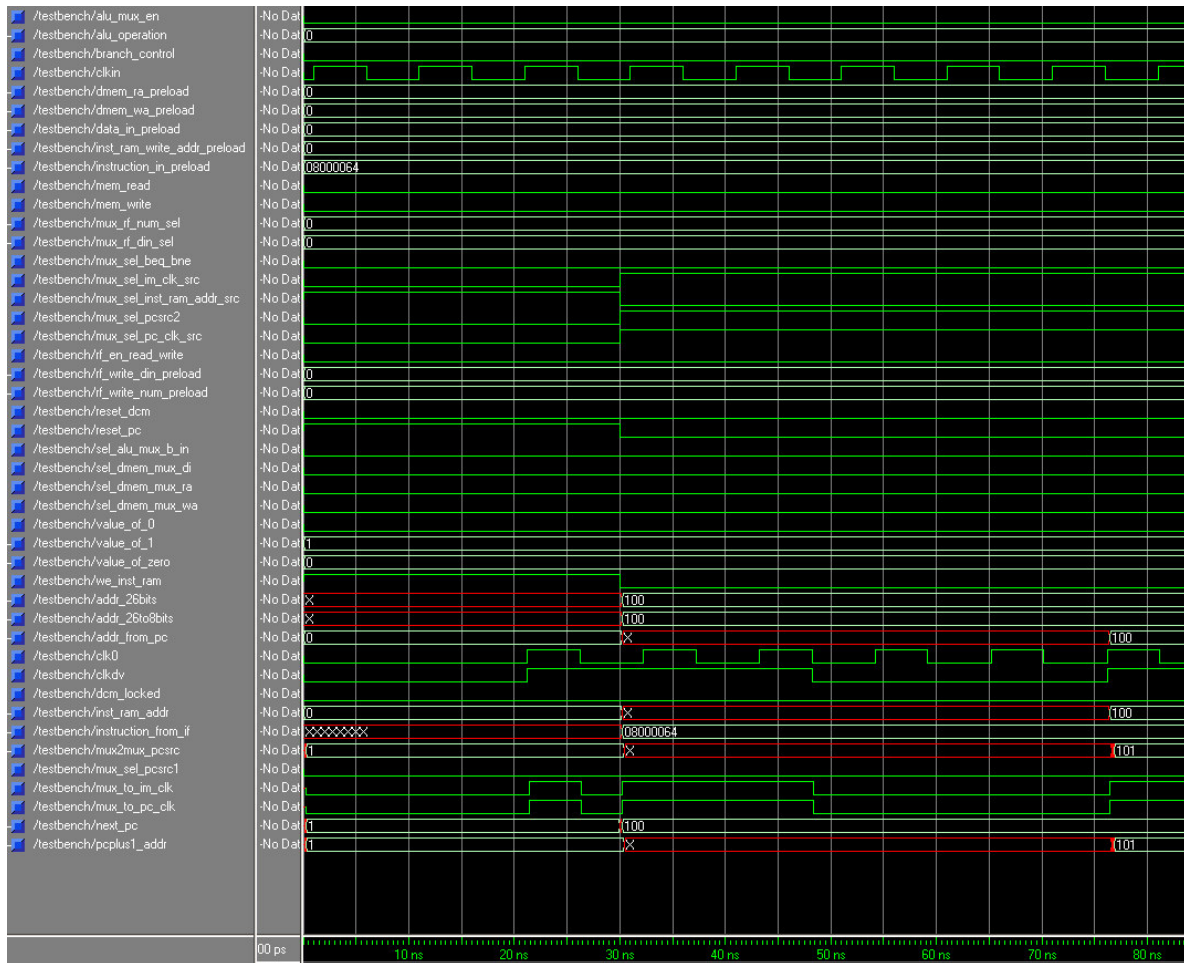


Figure B.59 Results of simulating the synthesized complete datapath with DCM for J instruction, using ModelSim. There is no ALU Operation for this instruction.

Simulation for Sample Code No. 2 (J then LW)

Figure B.60 shows the simulation waveforms for Sample Code No.2. This code is as follows:

Instruction Memory Location No.	Instruction Loaded			Comments
0	<u>J</u>	100		
100	<u>LW</u>	\$R8 , 10 (\$R5)		

This is assembled as follows:

J 100

address

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

```
000010 000000000000000000001100100
-----
op=2  address=100
```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(0000100000000000000000001100100)_2 = (8000064)_{\text{hex}}$$

□	<u>LW</u>	\$R8,	10	(\$R5)
		-----	-----	
		<i>rt</i>	<i>offset</i>	<i>(rs)</i>

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

```
100011 00101 01000 0000000000001010
-----
op=35  rs=$R5  rt=$R6      offset=10
```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101010000000000000001010)_2 = (8CA8000A)_{\text{hex}}$$

□ **Conclusions:**

These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for this sample code.

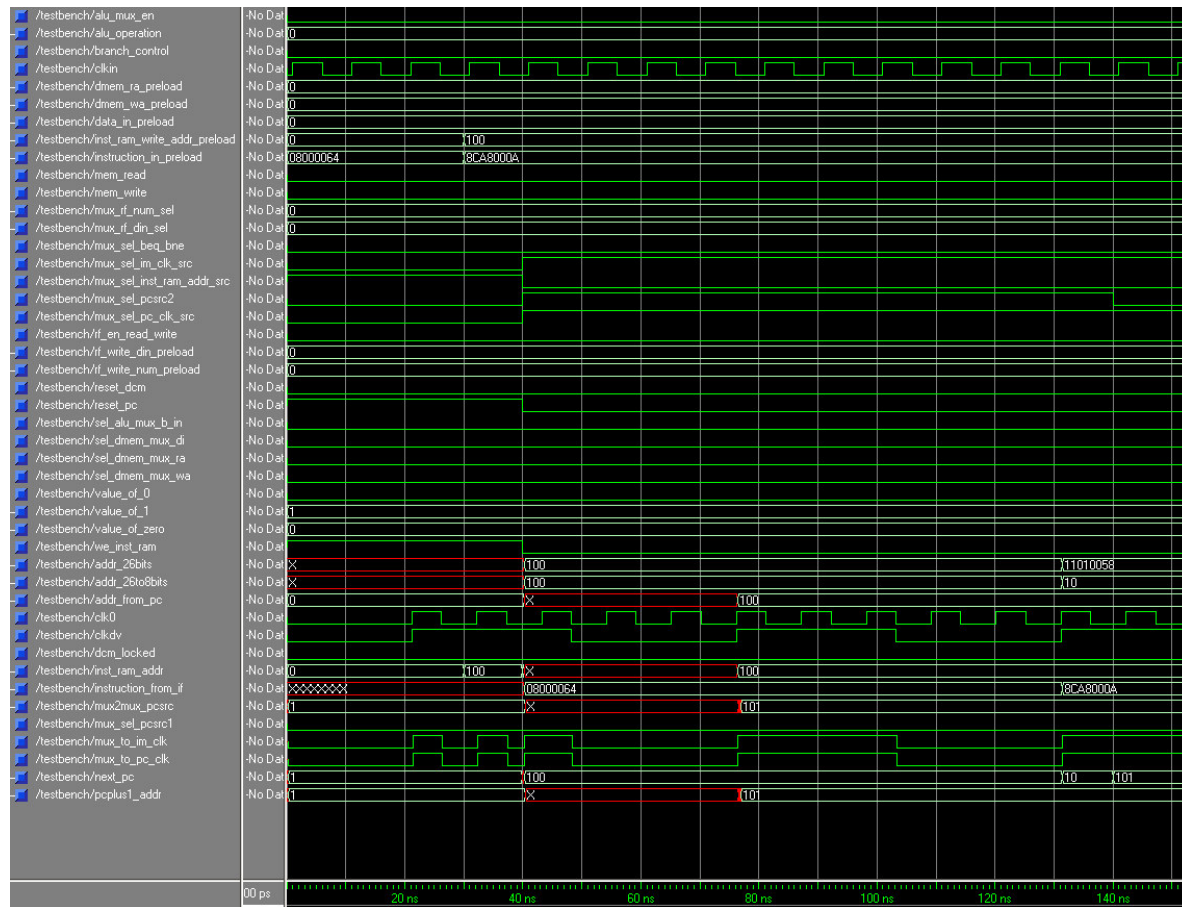


Figure B.60 Results of simulating the synthesized complete datapath with DCM for Sample Code No.2 (J, LW)

➤ Simulation for BEQ

Figures B.61 and B.62 show the waveform results of simulating the branch instruction BEQ by selecting ALU Operation = SUB = $(110)_{\text{binary}} = (6)_{\text{decimal}}$. There is no ALU Operation specifically for BEQ/BNE, but instead a SUB is all what is needed since the ALU performs a subtraction $([rs] - [rt])$ to compute the zero signal *ALU_Zero*.

The instruction simulated is:

<u>BEQ</u>	\$R5 ,	\$R6 ,	10
	-----	-----	-----
	<i>rs</i>	<i>rt</i>	<i>offset</i>

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000100	00101	00110	0000000000001010
-----	-----	-----	-----
<i>op</i> =4	<i>rs</i> =\$R5	<i>rt</i> =\$R6	<i>offset</i> = 10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00010000101001100000000000001010)_2 = (10A6000A)_{\text{hex}}$

❖ **Condition 1 (Figure B.61): When $[rs] = [rt]$**

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (15)_{\text{decimal}}$

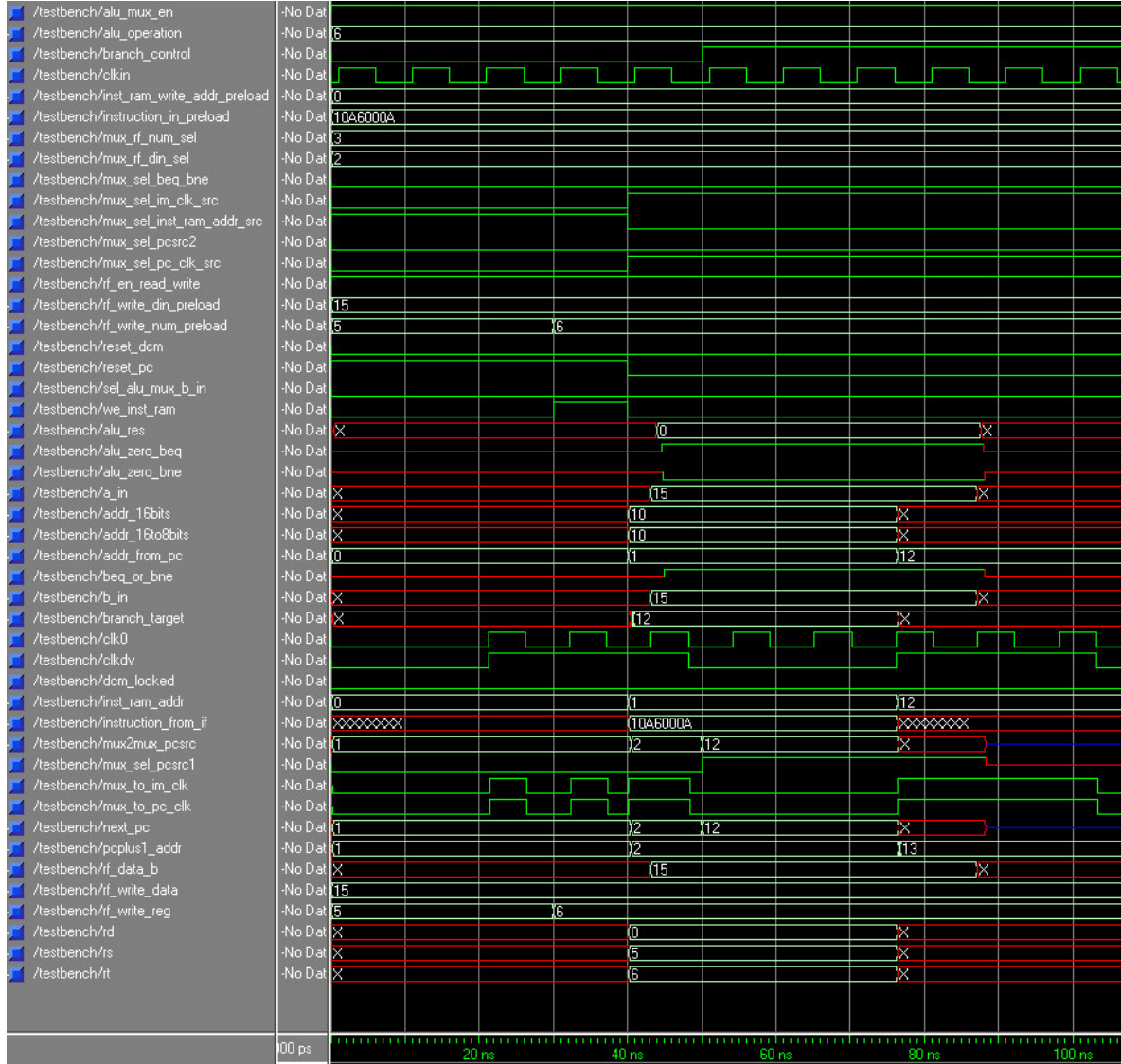


Figure B.61 Results of simulating the synthesized complete datapath with DCM for the branch instruction BEQ, using ModelSim, with ALU Operation = SUB for the condition $[rs] = [rt]$.

❖ **Condition 2 (Figure B.62): When $[rs] \neq [rt]$**

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (16)_{\text{decimal}}$

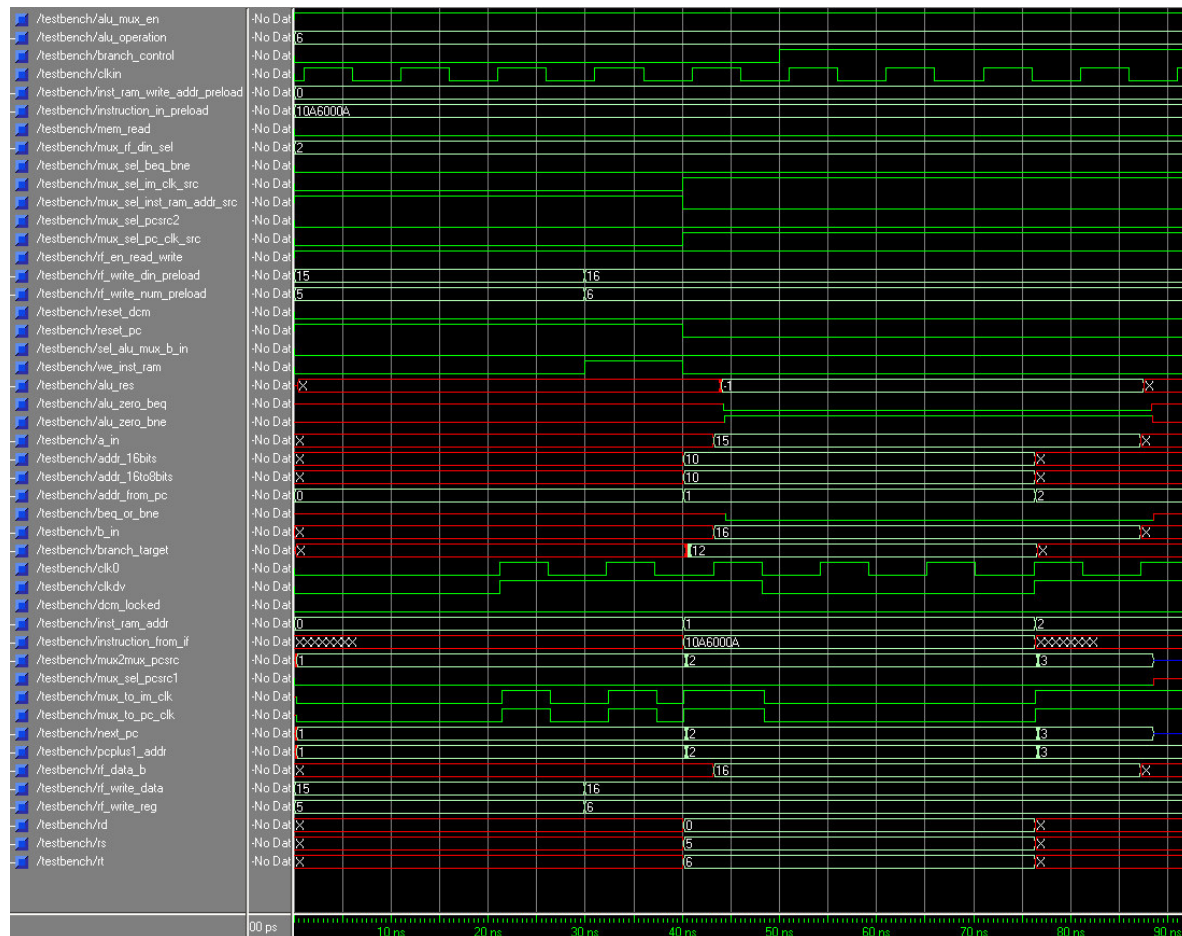


Figure B.62 Results of simulating the synthesized complete datapath with DCM for the branch instruction BEQ, using ModelSim, with ALU Operation = SUB for the condition $[rs] \neq [rt]$.

❑ Conclusions:

These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for the branch instruction BEQ.

➤ Simulation for BNE

Figures B.63 and B.64 show the waveform results of simulating the branch instruction BNE by selecting ALU Operation = SUB = $(110)_{\text{binary}} = (6)_{\text{decimal}}$. There is no ALU Operation specifically for BEQ/BNE, but instead a SUB is all what is needed since the ALU performs a subtraction $([rs] - [rt])$ to compute the zero signal ALU_Zero .

The instruction simulated is:

<u>BNE</u>	\$R5 ,	\$R6 ,	10
	-----	-----	-----
	rs	rt	offset

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:


```

000101 00101 00110 0000000000001010
-----
op=5    rs=$R5  rt=$R6    offset= 10

```

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00010100101001100000000000001010)_2 = (14A6000A)_{\text{hex}}$

❖ **Condition 1 (Figure B.63): When $[rs] \neq [rt]$**

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (16)_{\text{decimal}}$

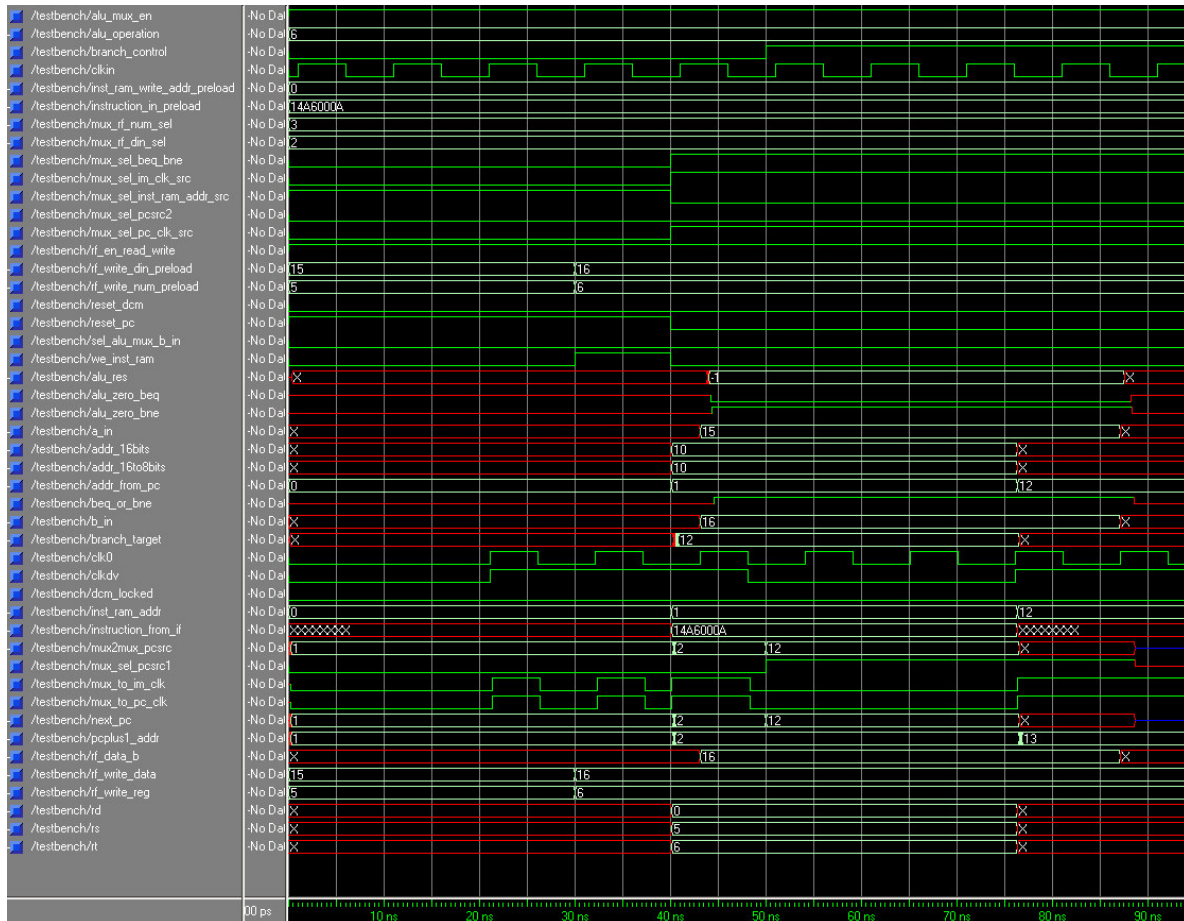


Figure B.63 Results of simulating the synthesized complete datapath with DCM for the branch instruction BNE, using ModelSim, with ALU Operation = SUB for the condition $[rs] \neq [rt]$.

❖ **Condition 2 (Figure B.64): When $[rs] = [rt]$**

$rs = \$R5$, $[\$R5] = (15)_{\text{decimal}}$

$rt = \$R6$, $[\$R6] = (15)_{\text{decimal}}$

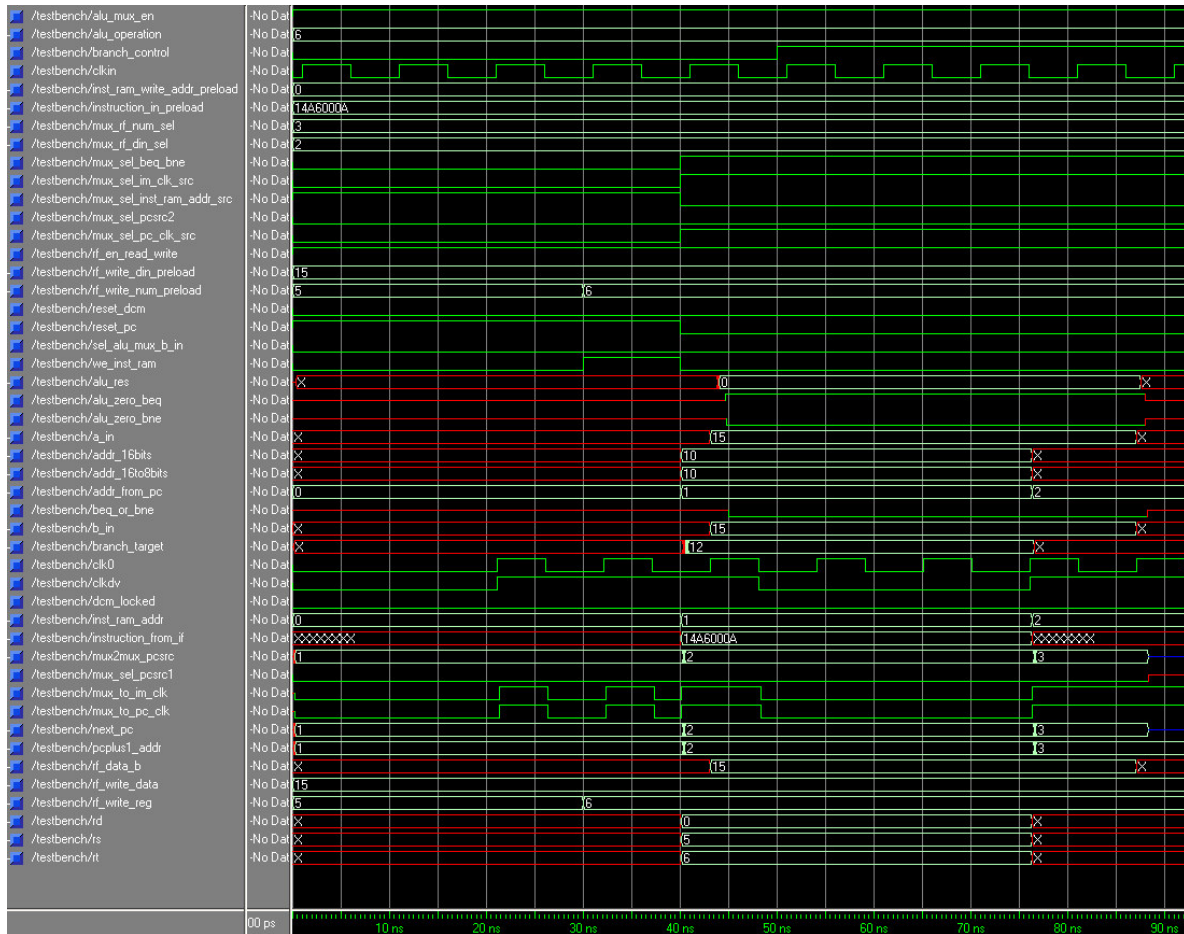


Figure B.64 Results of simulating the synthesized complete datapath with DCM for the branch instruction BNE, using ModelSim, with ALU Operation = SUB for the condition $[rs] = [rt]$.

❑ Conclusions:

These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for the branch instruction BNE.

➤ Simulation for Sample Code No. 3 (BEQ then LW)

Figure B.65 shows the simulation waveforms for Sample Code No.3. This code is as follows:

Instruction Memory Location No.	Instruction Loaded	Comments
0	<u>BEQ</u> \$R5, \$R6, 10	$[\$R5] = [\$R6] = (15)_{decimal}$ \Rightarrow Branch Taken Next PC = Current PC + 1 + 10 $= 1 + 1 + 10 = 12$
12	<u>LW</u> \$R8, 10 (\$R5)	

This is assembled as follows:

□ BEQ \$R5 , \$R6 , 10
 ----- -----
 rs *rt* *offset*

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000100 00101 00110 0000000000001010

 $op=4$ $rs=\$R5$ $rt=\$R6$ $offset=10$

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00010000101001100000000000001010)_2 = (10A6000A)_{\text{hex}}$

□ LW \$R8 , 10 (\$R5)
 ----- -----
 rt *offset* (*rs*)

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

100011 00101 01000 0000000000001010

 $op=35$ $rs=\$R5$ $rt=\$R6$ $offset=10$

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(10001100101010000000000000001010)_2 = (8CA8000A)_{\text{hex}}$

In the figure, the mux control signals are not set manually correctly. This is just to test that the preceeding BEQ instruction is functioning properly.

□ **Conclusions:**

These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for this sample code.

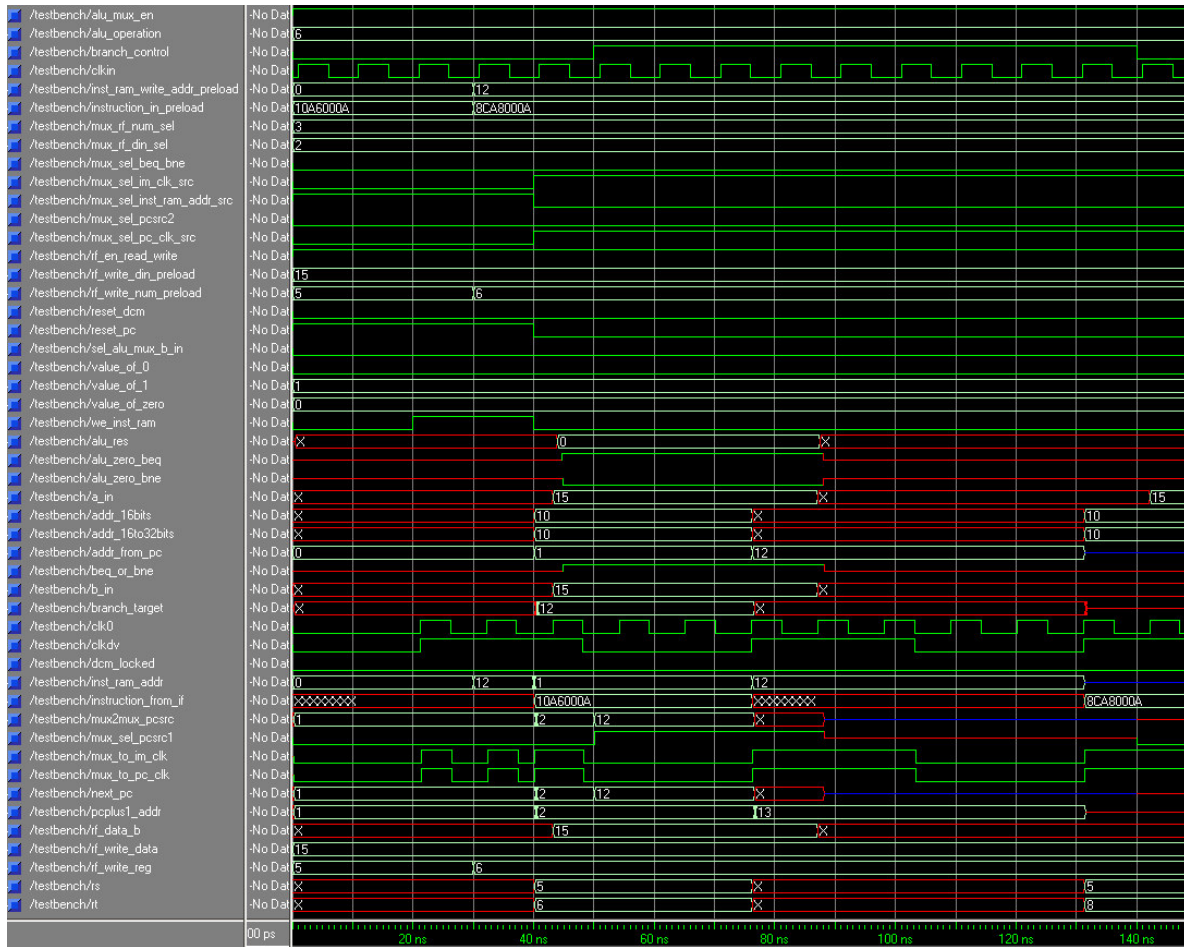


Figure B.65 Results of simulating the synthesized complete datapath with DCM for Sample Code No.3 (BEQ, then LW)

➤ **Simulation for Sample Code No. 4 (SLT, SW, J, BNE then LW)**

Figure B.66 shows the simulation waveforms for Sample Code No.4. This code is as follows:

Instruction Memory Location No.	Instruction Loaded	Comments
0	SLT \$R7, \$R5, \$R6	$[\$R5] = (15)_{10}$ $[\$R6] = (16)_{10}$ $[\$R5] < [\$R6] \Rightarrow [\$R7] = 1$
	Assembles to (A6382A)_{hex}	
1	SW \$R7, 10 (\$R5)	$[Memory[25]] = [\$R7] = 1$
	Assembles to (ACA7000A)_{hex}	
2	J 100	
	Assembles to (8000064)_{hex}	
100	BNE \$R5, \$R6, 10	$[\$R5] = [\$R6] = (15)_{10}$ \Rightarrow Branch Taken
	Assembles to (14A6000A)_{hex}	Next PC = Current PC + 1 + 10 = 101 + 1 + 10 = 112
112	LW \$R8, 10 (\$R5)	$[\$R8] = [Memory[25]] = 1$
	Assembles to (8CA8000A)_{hex}	

This is assembled as follows:

□ **SLT** \$R7, \$R5, \$R6

 rd rs rt

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000 00101 00110 00111 00000 101010

op=0 rs=\$R5 rt=\$R6 rd=\$R7 shamt funct=42

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000101010)_2 = (A6382A)_{hex}$

□ **SW** \$R7, 10 (\$R5)

 rt offset (rs)

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

101011 00101 00111 0000000000001010

op=43 rs=\$R5 rt=\$R7 offset=10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10101100101001110000000000001010)_2 = (\text{ACA7000A})_{\text{hex}}$$

□ I 100

 address

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000010 000000000000000000001100100

op=2 address=100

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(000010000000000000000000001100100)_2 = (\text{8000064})_{\text{hex}}$$

□ BNE \$R5 , \$R6 , 10
----- ----- -----
 rs rt offset

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000101 00101 00110 0000000000001010
----- ----- -----
op=5 rs=\$R5 rt=\$R6 offset= 10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00010100101001100000000000001010)_2 = (\text{14A6000A})_{\text{hex}}$$

□ LW \$R8 , 10 (\$R5)
----- -----
 rt offset (rs)

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

100011 00101 01000 0000000000001010
----- ----- -----
op=35 rs=\$R5 rt=\$R6 offset=10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101010000000000000001010)_2 = (\text{8CA8000A})_{\text{hex}}$$

Note: In HDL Bench, I couldn't assign the control/mux signals for LW because HDL Bench hangs every time I try changing a signal that is after 300 ns!!!! (is this a software glitch obviously). This is not a problem eventually as then it will be the control unit that will take over and generate these signals internally and, therefore, bypassing, this problem beyond its 300 ns limit.

However, currently the LW is still being read out from the instruction memory as a result of the branching (preceding BNE instruction), which means the datapath is functioning according to specifications.

Conclusions:

These resulting waveforms are in line with those shown in previous sections and prove that this complete datapath with DCM is functioning as expected for this sample code.

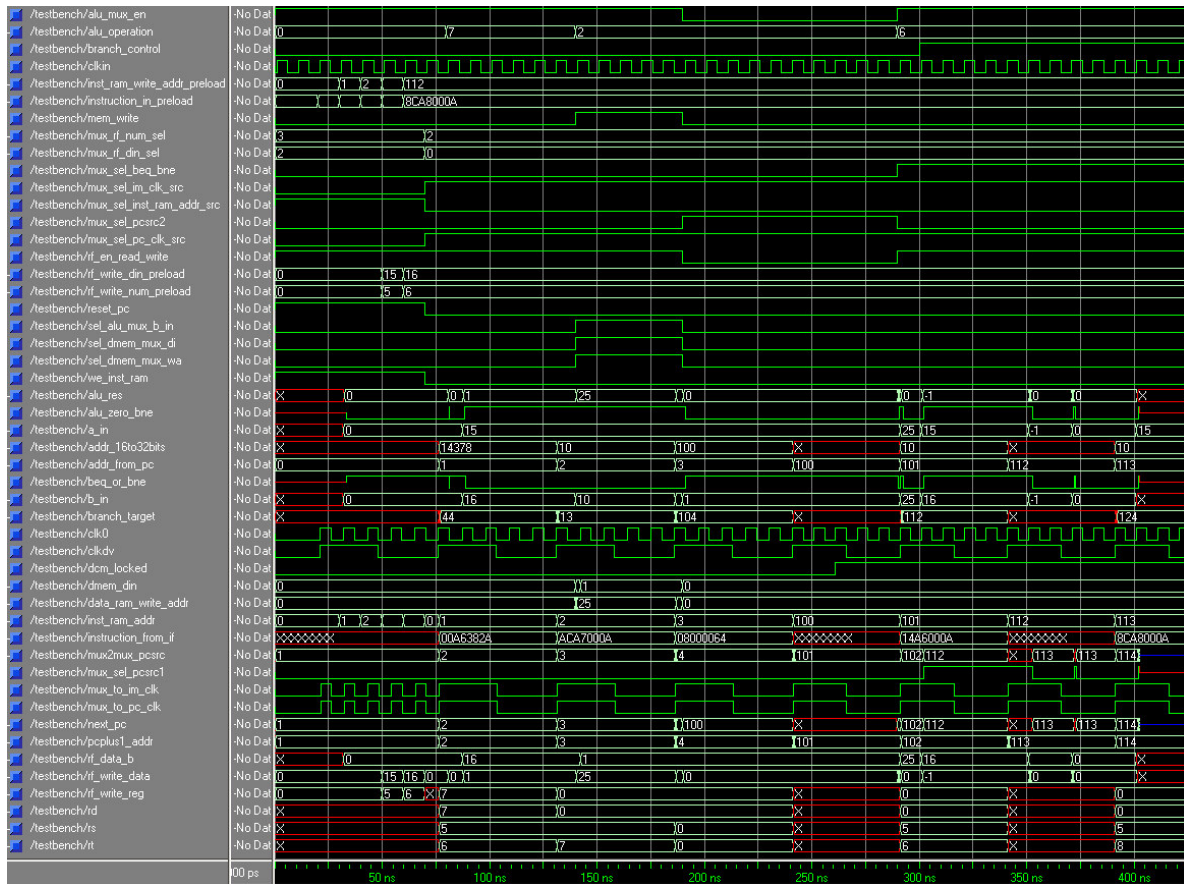


Figure B.66 Results of simulating the synthesized complete datapath with DCM for Sample Code No.4 (SLT, SW, J, BNE then LW)

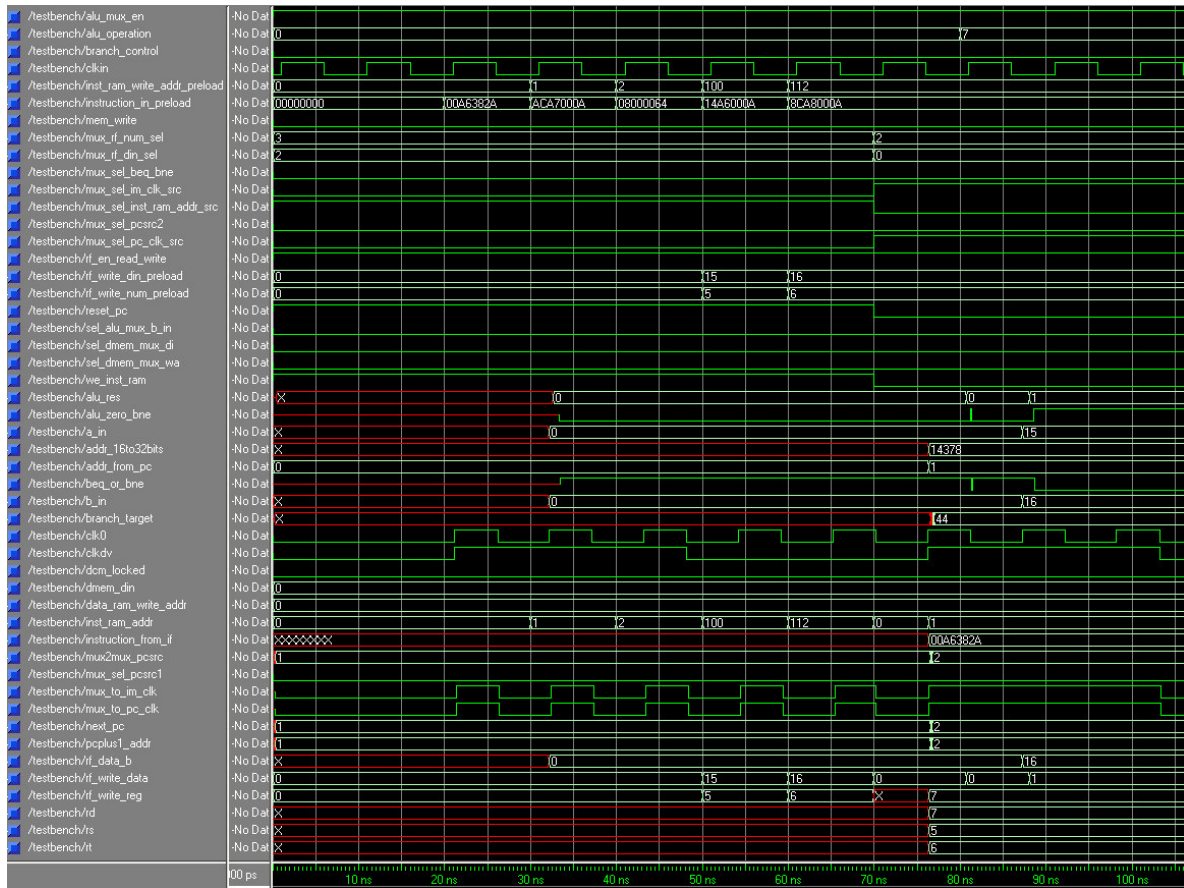


Figure B.66A Results of simulating the synthesized complete datapath with DCM for Sample Code No.4 (SLT, SW, J, BNE then LW) – Part 1 of 4.

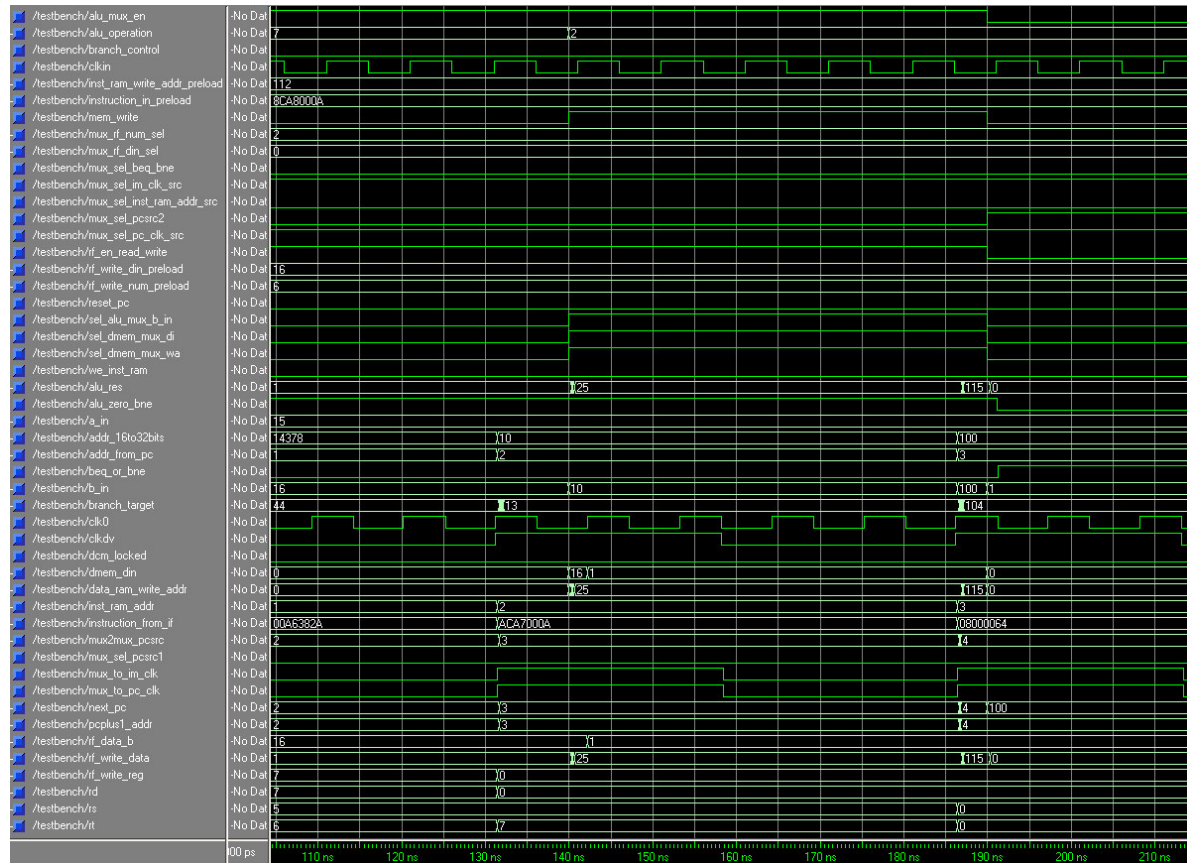
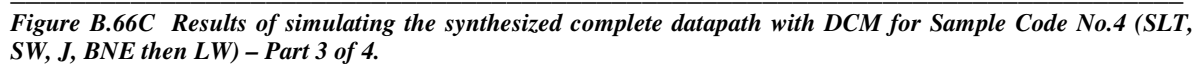


Figure B.66B Results of simulating the synthesized complete datapath with DCM for Sample Code No.4 (SLT, SW, J, BNE then LW) – Part 2 of 4.



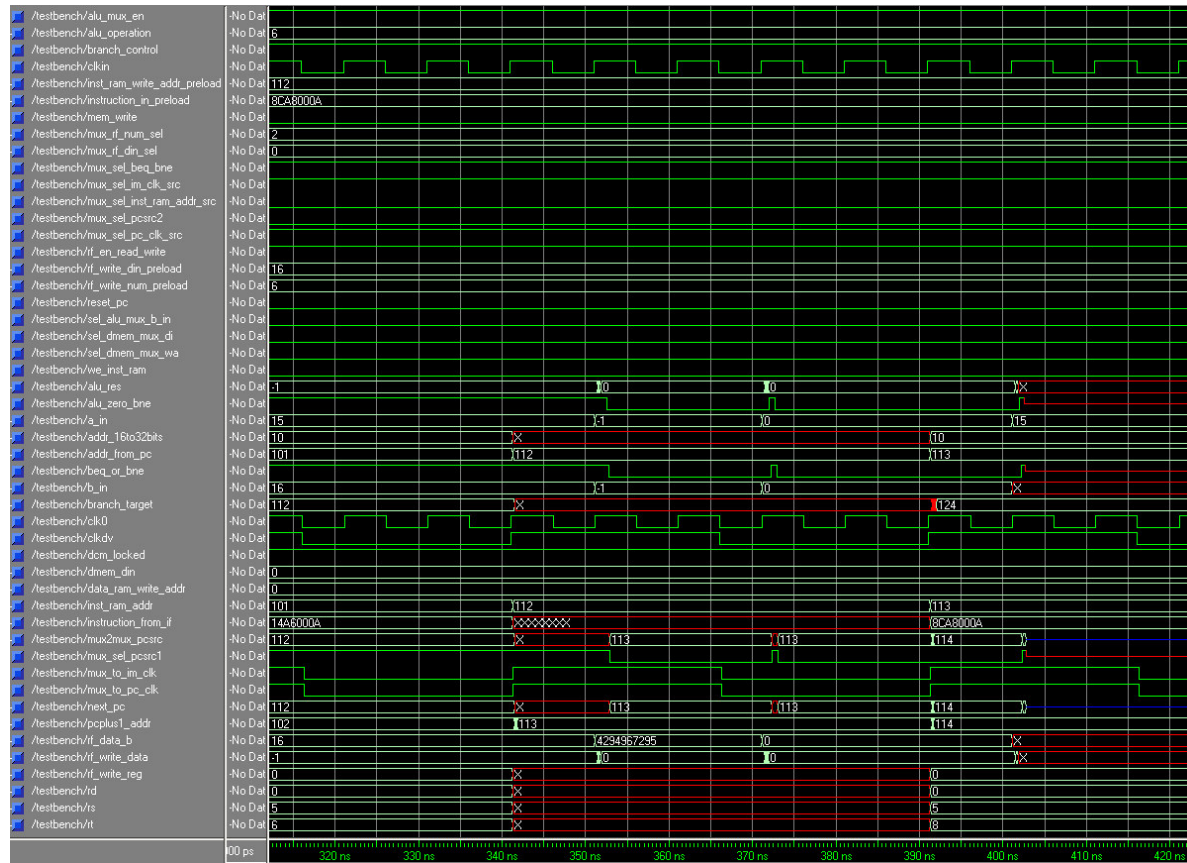


Figure B.66D Results of simulating the synthesized complete datapath with DCM for Sample Code No.4 (SLT, SW, J, BNE then LW) – Part 4 of 4.

B.6 Summary and Conclusions

This appendix discussed in detail the construction of the various datapath sections from the basic building blocks covered in the previous appendix. Also, this appendix elaborated on how these datapath sections are combined together along with the DCM to construct the final complete datapath.

The next appendix covers the design of the control unit, which ties into this complete datapath.