(CSE112) Computer Organization and Architecture

Sophomore CESS

SPRING 2022

Group ID (13)

Project Phase (2) Report

Name	ID
Ahmed Ehab Mohamed El-Baramony	21P0261
Ahmed Mohamed Mohamed	22P0283
Hesham Mohamed El-Afandy	21P0054

Table Of Figures:

Figure 1: Abstract CPU Design Diagram 3
Figure 2: SL2 RTL 7
Figure 3: MUX 2x1 RTL 7
Figure 4: Adder RTL 7
Figure 5: Datapath RTL 9
Figure 6: Controller Abstract Diagram 10
Figure 7: Main Decoder Table 10
Figure 8: Main Decoder Diagram 10
Figure 9: ALU Decoder Table 11
Figure 10: ALU Decoder Diagram 11
Figure 11: Main Decoder RTL 14
Figure 12: ALU Decoder RTL 14
Figure 13: MIPS Processor Abstract Diagram 15
Figure 14: MIPS Processor RTL 17
Figure 15: Main Module Abstract Design 18
Figure 16: Instruction Memory RTL 21
Figure 17: Data Memory RTL 22
Figure 18: Main Module RTL 22
Figure 19: MIPS Assembly Code Program 23
Figure 20: Simulation Binary Screenshot 25
Figure 21: Simulation Signed Numbers Screenshot 25
Figure 22: Simulation Console Screenshot 25

Project Overview:

A design of MIPS processor using VHDL that illustrates a basic computer system by simulating the data and control paths.

Phase (2) Requirements:

Modify the MIPS CPU as to be able to perform not only **R-instructions**, but also **I-type (lw, sw, beq)** & **J-instruction**.

1- Implement the **Control Module**: which is responsible for all the control signals
2- Implement the **MIPS Module** by connecting the **Datapath** with the **Control Module**
3- Connect **MIPS Module** with the **Instruction & Data Memory Module** together
4- Fill the **Instruction Memory Module** by a simple program
5- The CPU should be able to execute this program
6- Simulate the results and check the results.

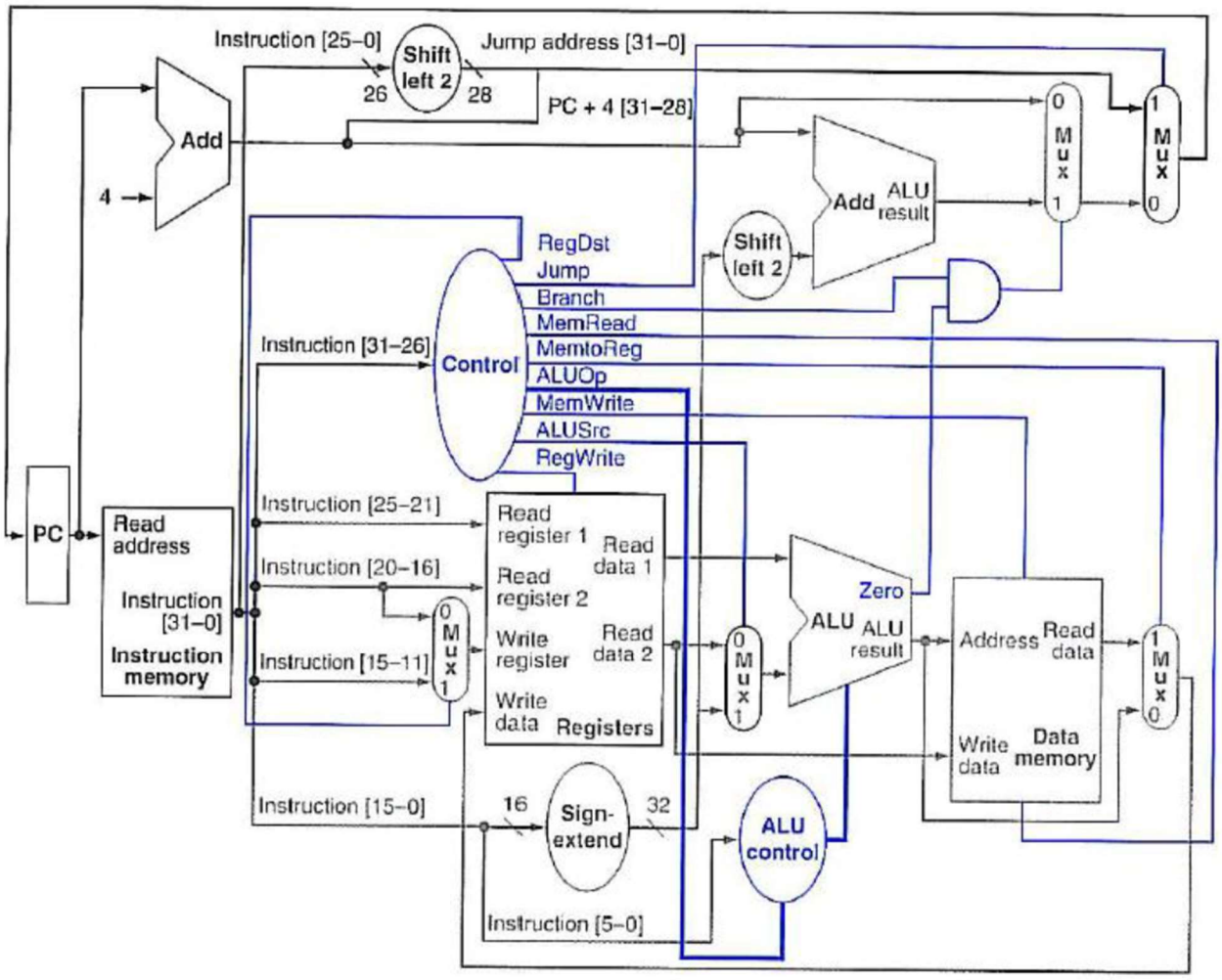

Figure 1: Abstract CPU Design Diagram

Implementation:

Part (1): Designing Components

First, we need to design some components that we need to make update our Datapath file that we implemented in Phase (1), these components are:

- Shift-Left By 2 “sl2”
- MUX 2x1
- Adder

Code:

Shift-Left By 2 “sl2”

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

entity sl2 is
     Port ( input : in  STD_LOGIC_VECTOR (31 downto 0);
            output : out  STD_LOGIC_VECTOR (31 downto 0));
end sl2;

architecture Behavioral of sl2 is

begin

output <= input (29 downto 0) & "00";

end Behavioral;
```

MUX 2x1

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

entity MUX2x1 is
    GENERIC (n: integer := 32);
     Port ( input_0 : in  STD_LOGIC_VECTOR (n-1 downto 0);
            input_1 : in  STD_LOGIC_VECTOR (n-1 downto 0);
            selector : in  STD_LOGIC;
            output : out  STD_LOGIC_VECTOR (n-1 downto 0));
end MUX2x1;

architecture Behavioral of MUX2x1 is

begin

output <=   input_0 when selector = '0' else
                input_1 when selector = '1' else
                (others =>'Z');

end Behavioral;
```

Adder

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

entity MUX2x1 is
    GENERIC (n: integer := 32);
     Port ( input_0 : in  STD_LOGIC_VECTOR (n-1 downto 0);
            input_1 : in  STD_LOGIC_VECTOR (n-1 downto 0);
            selector : in  STD_LOGIC;
            output : out  STD_LOGIC_VECTOR (n-1 downto 0));
end MUX2x1;

architecture Behavioral of MUX2x1 is

begin

output <=   input_0 when selector = '0' else
                input_1 when selector = '1' else
                (others =>'Z');

end Behavioral;
```

Package “datapath_pack”

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package datapath_pack is

component Reg
    Port ( read_sel1 : in  STD_LOGIC_VECTOR (4 downto 0);
           read_sel2 : in  STD_LOGIC_VECTOR (4 downto 0);
           write_sel : in  STD_LOGIC_VECTOR (4 downto 0);
           write_ena : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           reset: in STD_LOGIC;
           write_data : in  STD_LOGIC_VECTOR (31 downto 0);
           data1 : out  STD_LOGIC_VECTOR (31 downto 0);
           data2 : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component ALU
    Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
           data2 : in  STD_LOGIC_VECTOR (31 downto 0);
           aluop : in  STD_LOGIC_VECTOR (3 downto 0);
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
           zflag : out STD_LOGIC);
end component;

component MUX2x1
    GENERIC (n: integer);
    Port ( input_0 : in  STD_LOGIC_VECTOR (n-1 downto 0);
           input_1 : in  STD_LOGIC_VECTOR (n-1 downto 0);
           selector : in  STD_LOGIC;
           output : out  STD_LOGIC_VECTOR (n-1 downto 0));
end component;

component Adder
    GENERIC (n: integer);
    Port ( A : in  STD_LOGIC_VECTOR (n-1 downto 0);
           B : in  STD_LOGIC_VECTOR (n-1 downto 0);
           Y : out  STD_LOGIC_VECTOR (n-1 downto 0));
end component;

component sl2
    Port ( input : in  STD_LOGIC_VECTOR (31 downto 0);
           output : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component Flopr
    GENERIC(n:integer);
     Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           Load : in STD_LOGIC;
           D : in  STD_LOGIC_VECTOR (n-1 downto 0);
           Q : out  STD_LOGIC_VECTOR (n-1 downto 0));
end component;

component signext
    Port ( input : in  STD_LOGIC_VECTOR (15 downto 0);
           output : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

end datapath_pack;
```

ALU Update

Updated The ALU Designed In Phase (1). The ALU that was designed before passed the Phase (1) ALU Test, But in Phase (2) we prefered to make the design more efficent in order to maintain the whole processor performance.

```
library IEEE;

--Library Declaration (Unsigned) -> "+-...."
use IEEE.STD_LOGIC_1164.ALL;

--UNSIGNED library is used to introduce some arithmetic operation such as Addition, Subtraction, etc...
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

--Input/Output Declarations
entity ALU is
    Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
           data2 : in  STD_LOGIC_VECTOR (31 downto 0);
           aluop : in  STD_LOGIC_VECTOR (3 downto 0);
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
              zflag : out STD_LOGIC);
end ALU;

architecture Behavioral of ALU is

SIGNAL tmp_1, tmp_2  : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL tmp_3 : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL result : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL AA, BB  : STD_LOGIC_VECTOR (31 downto 0);

begin

tmp_1 <= AA + BB + aluop(2);
tmp_3 <= "00000000000000000000000000000000" & tmp_1(31);

AA <= NOT(data1) when aluop(3) = '1' else
       data1;
BB <= NOT(data2) when aluop(2) = '1' else
       data2;

result <=   (AA AND BB) when aluop(1 downto 0) = "00" else
               (AA OR BB) when aluop(1 downto 0) = "01" else
               tmp_1 when aluop(1 downto 0) = "10" else
               tmp_3 when aluop(1 downto 0) = "11" else
               (Others => 'Z');

zflag <= '1' when result = x"00000000" else '0';

dataout <= result;

end Behavioral;
```

RTL Diagrams:

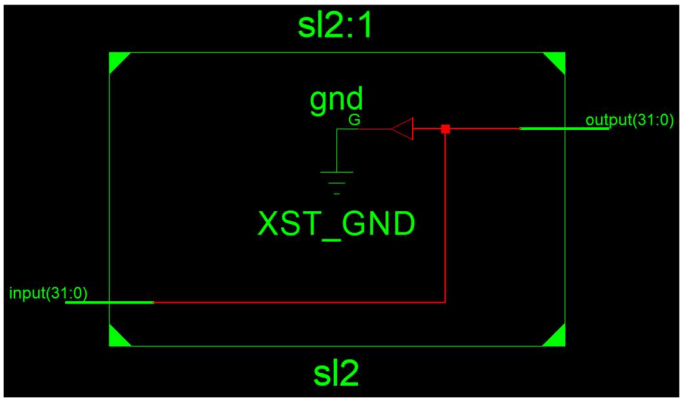

Figure 2: SL2 RTL

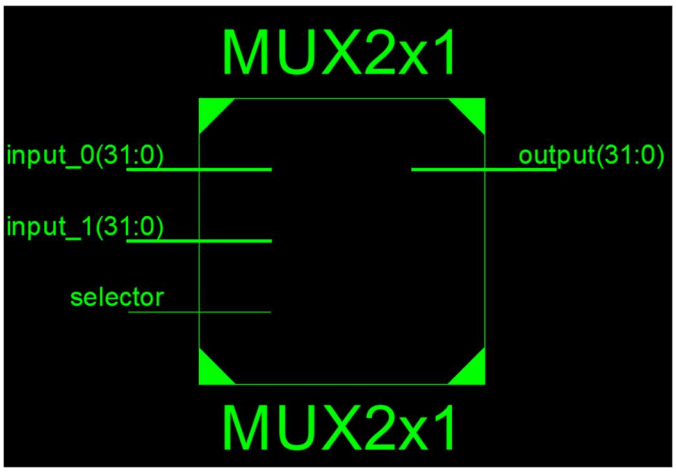

Figure 3: MUX 2x1 RTL

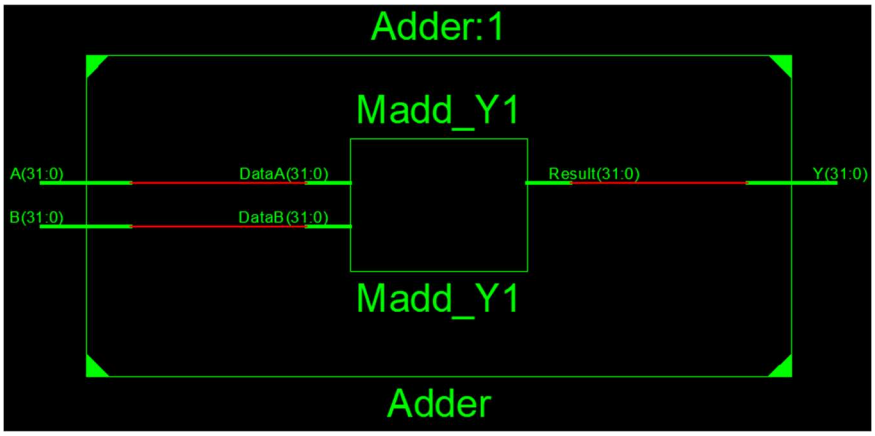

Figure 4: Adder RTL

Part (2): Updating The Datapath

After we designed the components required along with the components already designed in Phase (1) we need to begin **Updating Datapath**

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.datapath_pack.ALL;

entity Datapath is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           readdata: in STD_LOGIC_VECTOR(31 downto 0);
           instr : in  STD_LOGIC_VECTOR (31 downto 0);
           memtoreg, pcsrc,alusrc,regwrite, regdst, jump: in STD_LOGIC;
           aluoperation : in  STD_LOGIC_VECTOR (3 downto 0);
           zero : out  STD_LOGIC;
           pc,aluout, writedata : out  STD_LOGIC_VECTOR (31 downto 0));

end Datapath;

architecture Behavioral of Datapath is

signal reg_read_out_1, reg_read_out_2: STD_LOGIC_VECTOR(31 downto 0);
signal writereg: STD_LOGIC_VECTOR(4 downto 0);
signal pcjump, pcnext, pcnextbr, pcplus4,  pcbranch: STD_LOGIC_VECTOR(31 downto 0);
signal signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
signal read_mux_out, write_mux_out: STD_LOGIC_VECTOR(31 downto 0);

--Signals Instead Of Buffer
signal pc_t, aluout_t: STD_LOGIC_VECTOR(31 downto 0);

begin

--PC Instruction Path
pcjump <= pcplus4(31 downto 28) & instr (25 downto 0) & "00";

pcreg: Flopr generic map (32) port map (clk, reset, '1' ,pcnext, pc_t);
pcadd1: Adder generic map (32) port map (pc_t, X"00000004", pcplus4);
immsh: sl2 port map (signimm, signimmsh);
pcadd2: Adder generic map (32) port map (pcplus4, signimmsh, pcbranch);
pcbrmux: MUX2x1 generic map(32) port map(pcplus4, pcbranch,  pcsrc, pcnextbr);
pcmux: MUX2x1 generic map(32) port map(pcnextbr, pcjump, jump,  pcnext);

--MUX RT (Write Register) "Correct"
writeregmux: MUX2x1 generic map(5) port map(instr(20 downto 16), instr(15 downto 11),  regdst, writereg);

--Sign Extender "Correct"
sign_extender: signext port map(instr(15 downto 0),signimm);

--MUX ALU Input (2)
readregmux: MUX2x1 generic map(32) port map(reg_read_out_2, signimm,  alusrc, read_mux_out);

--Register
Regist : Reg port map (instr(25 downto 21) , instr(20 downto 16) , writereg , regwrite, clk, reset ,
write_mux_out , reg_read_out_1 , reg_read_out_2);

--ALU
ALU1 : ALU port map (reg_read_out_1 , read_mux_out , aluoperation , aluout_t , zero);

--MUX Write To Register
writeenmux: MUX2x1 generic map(32) port map(aluout_t, readdata,  memtoreg, write_mux_out);

pc <= pc_t;
writedata <= reg_read_out_2;
aluout <= aluout_t;

end Behavioral;
```

Datapath Comments:

- We used signals for **pc**, **writedata**, **aluout** as we use them as input and outputs between components instead of setting them as buffer because buffer produce error when testing, at the end we assign the values to its corresponding output signal.
- The above code is for all the datapath **EXCEPT** the **Instruction Memory** & **Data Memory** & **Controller Unit**

RTL Diagram:

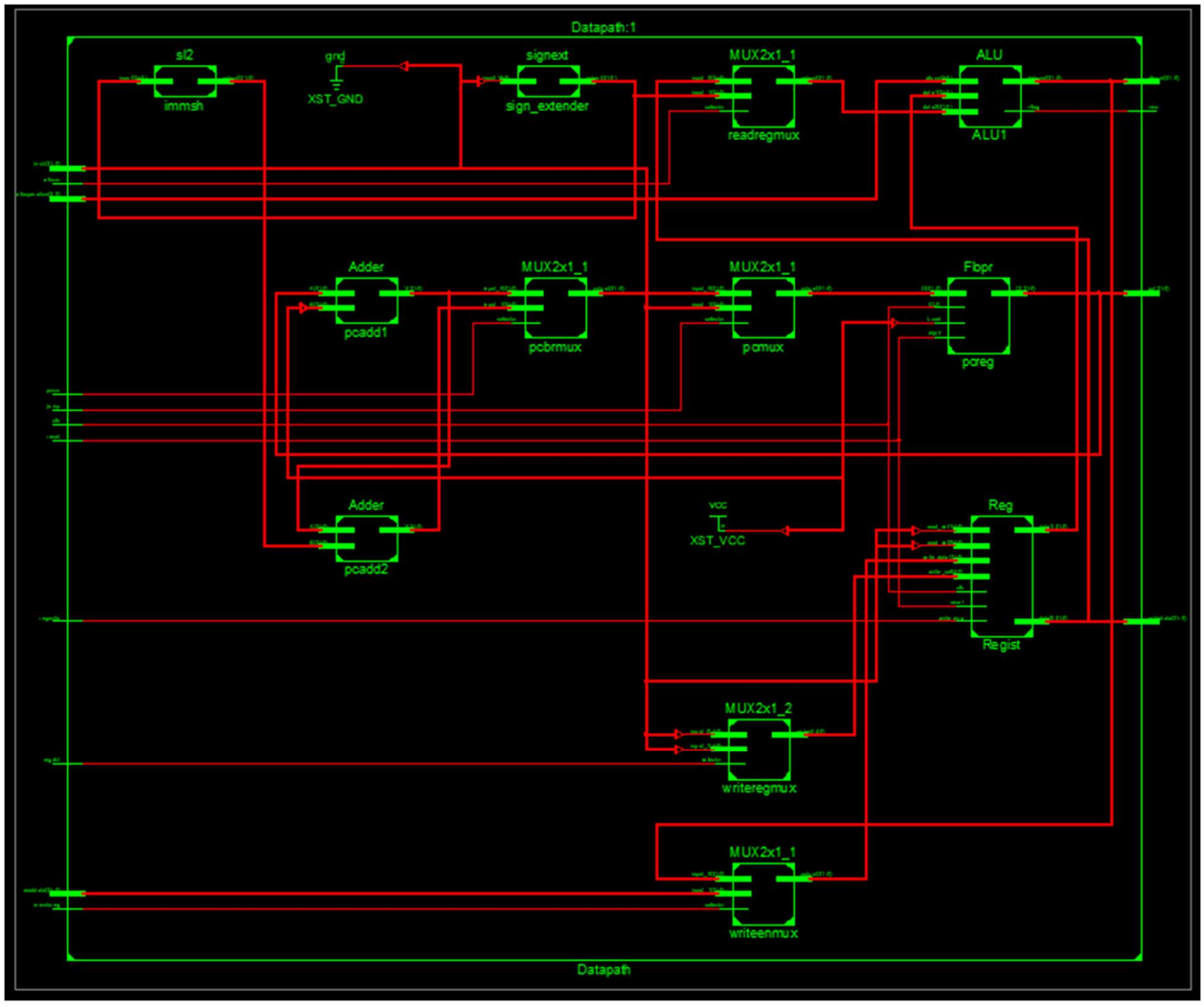

Figure 5: Datapath RTL

Part (3): Control Unit (CU)

After updating the Datapath we need the **Control Unit** which is responsible for the **Enables** & **Selectors** of the datapath components along with the **ALU Operation Selector**.

- The **Control Unit (CU)** is a component of a computer's **Central Processing Unit** (CPU) that directs the operation of the **Processor**. To design the following unit we need to do two components and then join them inside the controller file:
    1. **Main Decoder**
    2. **ALU Decoder**

Diagram:

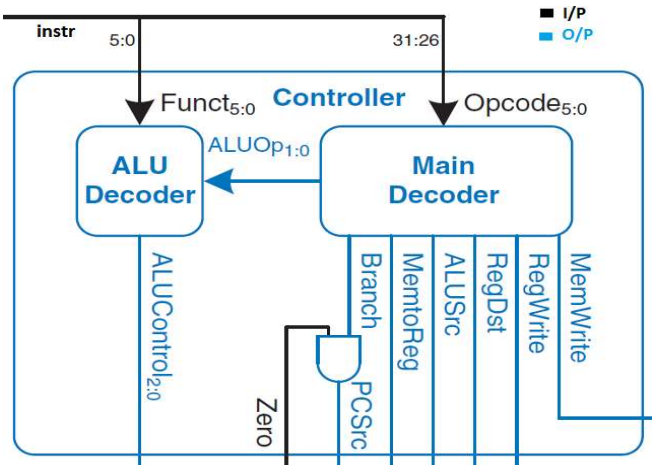

Figure 6: Controller Abstract Diagram

Instruction	op	Regwrite	Regdst	Alusrc	Branch	Mem write	Memtoreg	Jump	Aluop (1)	Aluop (0)
R-type	000000	1	1	0	0	0	0	0	1	0
Lw	100011	1	0	1	0	0	1	0	0	0
Sw	101011	0	0	1	0	1	0	0	0	0
beq	000100	0	0	0	1	0	0	0	0	1
addi	001000	1	0	1	0	0	0	0	0	0
J	000010	0	0	0	0	0	0	1	0	0

Figure 7: Main Decoder Table

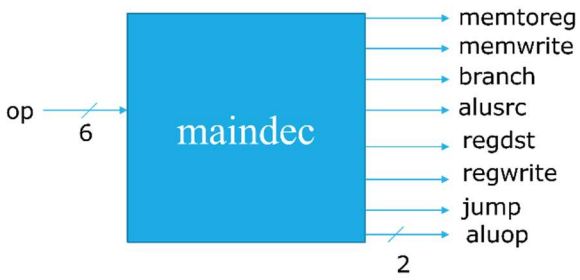

Figure 8: Main Decoder Diagram

Instruction	Aluop	funct	alucontrol
w,sw,addi	00		0010
Beq	01		0110
R-type	10	100000	0010
		100010	0110
		100100	0000
		100101	0001
		101010	0111

Figure 9: ALU Decoder Table

Figure 10: ALU Decoder Diagram

Code:

Main Decoder

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

entity control_main_decoder is
    Port ( op : in  STD_LOGIC_VECTOR (5 downto 0);
           memtoreg : out  STD_LOGIC;
           memwrite : out  STD_LOGIC;
           branch : out  STD_LOGIC;
           alusrc : out  STD_LOGIC;
           regdst : out  STD_LOGIC;
           regwrite : out  STD_LOGIC;
           jump : out  STD_LOGIC;
           aluop : out  STD_LOGIC_VECTOR (1 downto 0));
end control_main_decoder;

architecture Behavioral of control_main_decoder is

signal controls: STD_LOGIC_VECTOR(8 downto 0);

begin

process(op)
    begin
    case op is
        when "000000" => controls <= "110000010"; --RTYPE
        when "100011" => controls <= "101001000"; --LW
        when "101011" => controls <= "001010000"; --SW
        when "000100" => controls <= "000100001"; --BEQ
        when "001000" => controls <= "101000000"; --ADDI
        when "000010" => controls <= "000000100"; --J
        when others => controls <= "---------"; --Illegal OP
    end case;

end process;

(regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop(1), aluop(0)) <= controls;

end Behavioral;
```

ALU Decoder

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

entity alu_decoder is
    Port (  aluop : in  STD_LOGIC_VECTOR (1 downto 0);
                funct : in  STD_LOGIC_VECTOR (5 downto 0);
                alucontrol : out  STD_LOGIC_VECTOR (3 downto 0));
end alu_decoder;

architecture Behavioral of alu_decoder is

begin

process (aluop, funct)
begin

case aluop is
when "00" => alucontrol <= "0010"; -- Addition (LW , SW , ADDi)
when "01" => alucontrol <= "0110"; -- Subtraction (BEQ)
when others =>
case funct is
when "100000" => alucontrol <= "0010"; --Add
when "100010" => alucontrol <= "0110"; --Sub
when "100100" => alucontrol <= "0000"; --And
when "100101" => alucontrol <= "0001"; --Or
when "101010" => alucontrol <= "0111"; --Slt
when others => alucontrol <= "----";
end case;
end case;
end process;

end Behavioral;
```

Controller File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.Control_Package.ALL;

entity controller is
    Port ( op : in  STD_LOGIC_VECTOR (5 downto 0);
           funct : in  STD_LOGIC_VECTOR (5 downto 0);
           zero : in  STD_LOGIC;
           memtoreg : out  STD_LOGIC;
           memwrite : out  STD_LOGIC;
           pcsrc : out  STD_LOGIC;
           alusrc : out  STD_LOGIC;
           regdst : out  STD_LOGIC;
           regwrite : out  STD_LOGIC;
           jump : out  STD_LOGIC;
           alucontrol : out  STD_LOGIC_VECTOR (3 downto 0));
end controller;

architecture Behavioral of controller is

signal aluop: STD_LOGIC_VECTOR (1 downto 0);
signal branch: STD_LOGIC;

begin

main: control_main_decoder port map (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, aluop);
alu_dec: alu_decoder port map (aluop, funct, alucontrol);

pcsrc <= branch AND zero;

end Behavioral;
```

Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package Control_Package is

--Main Decoder
component control_main_decoder
    Port (  op : in  STD_LOGIC_VECTOR (5 downto 0);
            memtoreg : out  STD_LOGIC;
            memwrite : out  STD_LOGIC;
            branch : out  STD_LOGIC;
            alusrc : out  STD_LOGIC;
            regdst : out  STD_LOGIC;
            regwrite : out  STD_LOGIC;
            jump : out  STD_LOGIC;
            aluop : out  STD_LOGIC_VECTOR (1 downto 0));
end component;

--ALU Decoder
component alu_decoder
    Port (  aluop : in  STD_LOGIC_VECTOR (1 downto 0);
            funct : in  STD_LOGIC_VECTOR (5 downto 0);
            alucontrol : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

end Control_Package;
```

Controller Code Comments:

- The **Main Decoder** is responsible for **deciding the desired type of instruction**, it chooses from the following instruction types **(R-TYPE , LW , SW , BEQ , ADDI , J)**
- The **ALU Control** is responsible for **choosing the ALU Operation** to execute the instructions of our program.
- In Conclusion, the **Main Decoder** & **ALU Control** both build what we call **Control Unit (CU)** that takes the **Function** & **Opcode** to tell the processor how to work.

RTL Diagram:

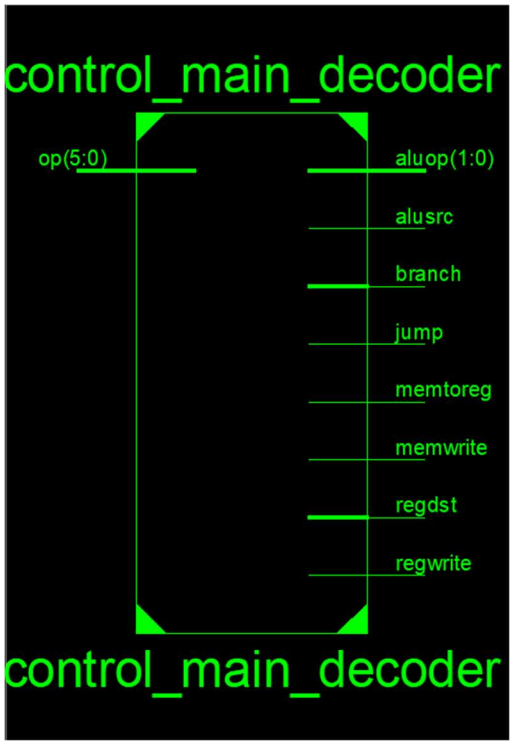

Figure 11: Main Decoder RTL

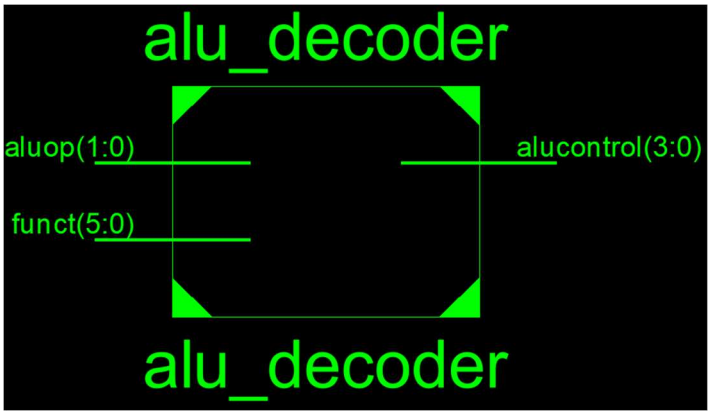

Figure 12: ALU Decoder RTL

Part (4): MIPS Processor

After designing the Control Unit (CU) & Updating the Datapath. We need to connect them both with each other in what we call MIPS Processor.

Diagram:

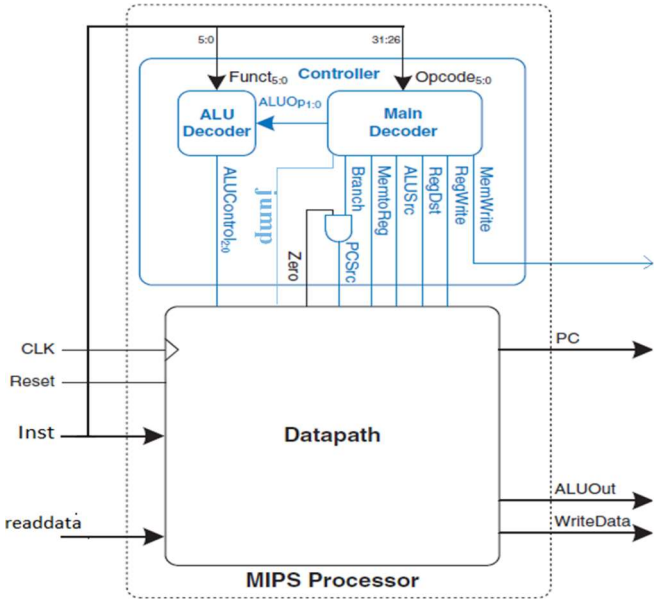

Figure 13: MIPS Processor Abstract Diagram

Code:

MIPS Processor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.MIPS_Package.ALL;


entity mips is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           pc : out  STD_LOGIC_VECTOR (31 downto 0);
           instr : in  STD_LOGIC_VECTOR (31 downto 0);
           memwrite : out  STD_LOGIC;
           aluout : out  STD_LOGIC_VECTOR (31 downto 0);
           writedata : out  STD_LOGIC_VECTOR (31 downto 0);
           readdata : in  STD_LOGIC_VECTOR (31 downto 0));
end mips;

architecture Behavioral of mips is

signal memtoreg, pcsrc, alusrc, regwrite, regdst, jump, zero: STD_LOGIC;
signal alucontrol: STD_LOGIC_VECTOR (3 downto 0);

begin

control: controller port map(instr(31 downto 26), instr(5 downto 0), zero ,memtoreg, memwrite, pcsrc, alusrc,
regdst, regwrite, jump, alucontrol);

dp: Datapath port map (clk, reset, readdata, instr, memtoreg, pcsrc, alusrc, regwrite, regdst, jump,
alucontrol, zero, pc, aluout, writedata);

end Behavioral;
```

Package “MIPS_Package”

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package MIPS_Package is

component Datapath
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           readdata: in STD_LOGIC_VECTOR(31 downto 0);
              instr : in  STD_LOGIC_VECTOR (31 downto 0);
              memtoreg, pcsrc,alusrc,regwrite, regdst, jump: in STD_LOGIC;
           aluoperation : in  STD_LOGIC_VECTOR (3 downto 0);
           zero : out  STD_LOGIC;
           pc, aluout, writedata : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component controller
    Port ( op : in  STD_LOGIC_VECTOR (5 downto 0);
           funct : in  STD_LOGIC_VECTOR (5 downto 0);
           zero : in  STD_LOGIC;
           memtoreg : out  STD_LOGIC;
           memwrite : out  STD_LOGIC;
           pcsrc : out  STD_LOGIC;
           alusrc : out  STD_LOGIC;
           regdst : out  STD_LOGIC;
           regwrite : out  STD_LOGIC;
           jump : out  STD_LOGIC;
           alucontrol : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

end MIPS_Package;
```

MIPS Code Comments:

- The MIPS File contains 2 components the Control Unit (CU) & Datapath.
- At this point we almost finished our processor design.

RTL Diagram:

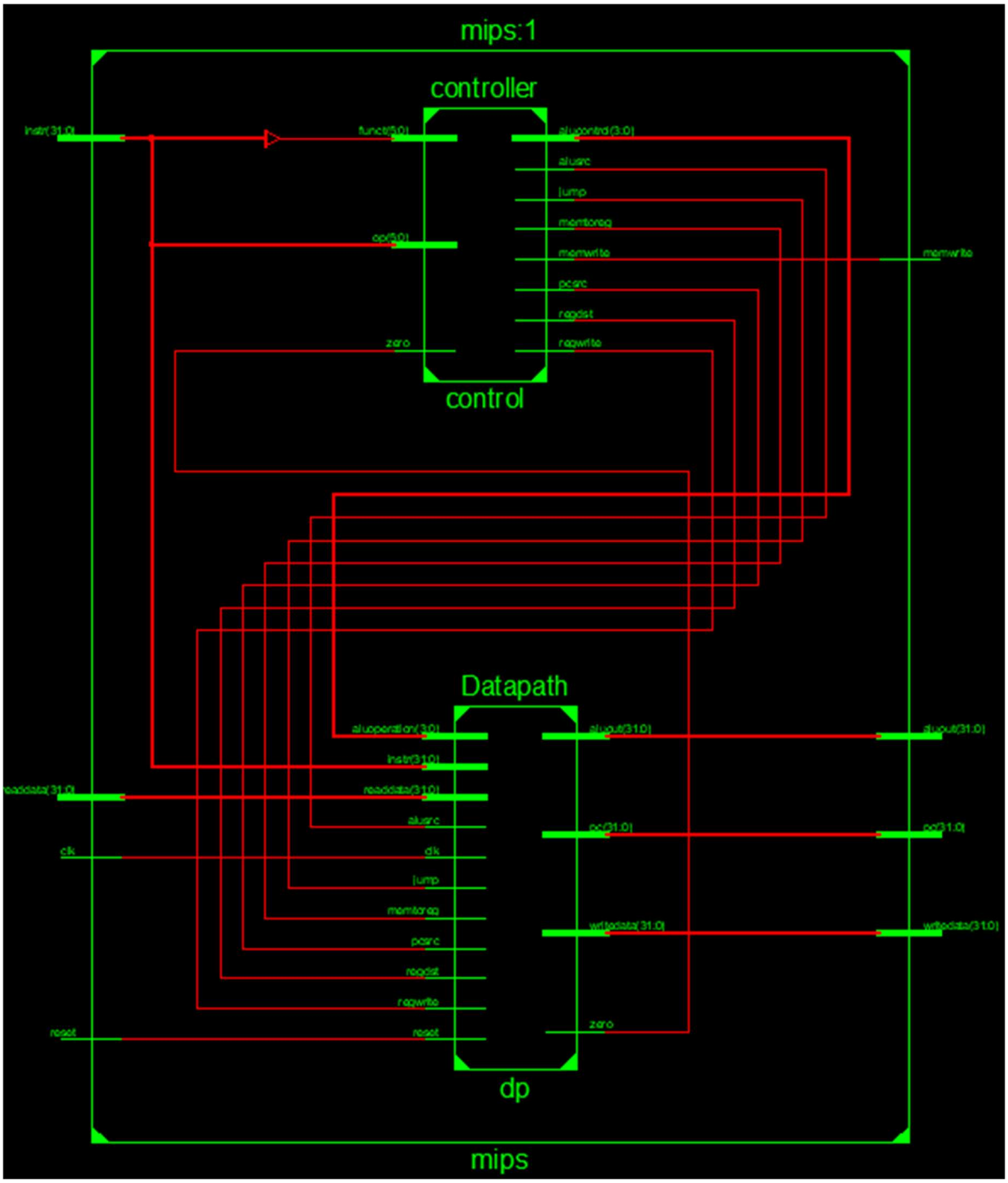

Figure 14: MIPS Processor RTL

Part (5): Main Module

Here we are at the **FINAL STEP** of our design which is connecting the **MIPS Processor** to both the **Data & Instruction Memories**.

Main Module:

For every instruction, the first two implementation steps are identical:

- Use **PC** to **fetch the instruction** from the **Instruction Memory**
- Use the **instruction fields** to **select one or two registers to read**:
  - **lw** requires reading only **one register**
  - **j** does **not require accessing any register**
  - **All other instructions** require reading **two registers**

Diagram:

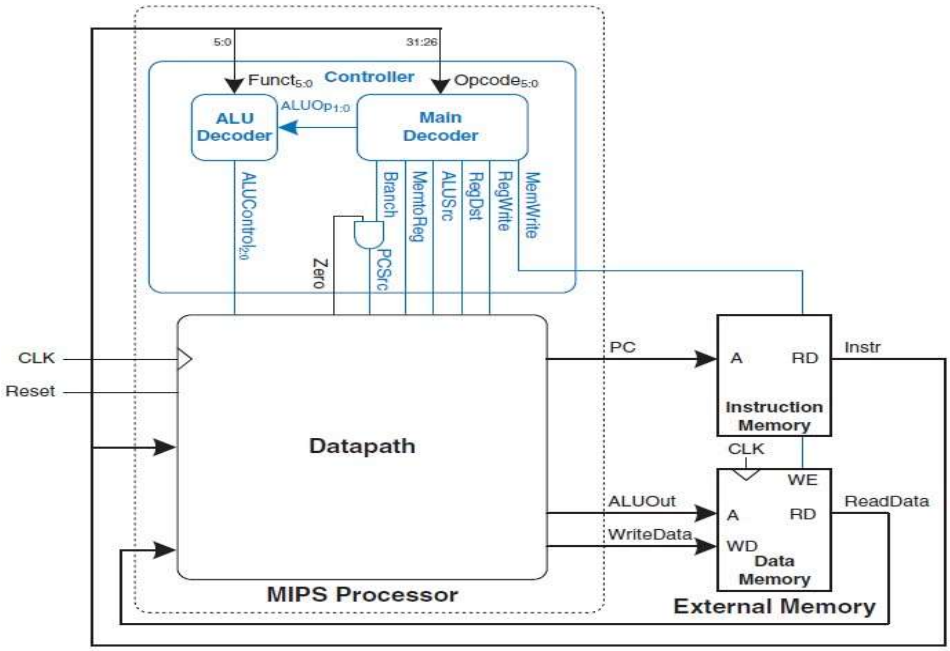

Figure 15: Main Module Abstract Design

Code:

Instruction Memory "imem" (Already Given)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
use ieee.numeric_std.all;
use IEEE.std_logic_textio.all;
library STD;
use STD.textio.all;
entity imem is -- instruction memory
port(a: in STD_LOGIC_VECTOR(5 downto 0);
rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
process is
file mem_file: TEXT;
variable L: line;
variable ch: character;
variable i, index, result: integer;
type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
variable mem: ramtype;
begin
-- initialize memory from file
for i in 0 to 63 loop -- set all contents low
mem(i) := (others => '0');
end loop;
index := 0;
FILE_OPEN (mem_file, "G:/memfile.dat", READ_MODE);
while not endfile(mem_file) loop
readline(mem_file, L);
result := 0;
for i in 1 to 8 loop
read (L, ch);
if '0' <= ch and ch <= '9' then
result := character'pos(ch) - character'pos('0');
elsif 'a' <= ch and ch <= 'f' then
result := character'pos(ch) - character'pos('a')+10;
else report "Format error on line" & integer'
image(index) severity error;
end if;
mem(index)(35-i*4 downto 32-i*4):= std_logic_vector(to_unsigned(result,4));
end loop;
index := index + 1;
end loop;
-- read memory
for i in 1 to 1000 loop
rd <= mem(CONV_INTEGER(a));
wait on a;
end loop;
end process;

end;
```

Data Memory "dmem" (Already Given)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
use ieee.numeric_std.all;

entity dmem is -- data memory
port(clk, we: in STD_LOGIC;
a, wd: in STD_LOGIC_VECTOR (31 downto 0);
rd: out STD_LOGIC_VECTOR (31 downto 0));
end;
architecture behave of dmem is
begin
process  is
type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
variable mem: ramtype;
begin
-- read or write memory
for i in 1 to 1000 loop
if rising_edge(clk) then
if (we='1') then
mem (CONV_INTEGER('0'&a(7 downto 2))):= wd;
end if;
end if;
rd <= mem (CONV_INTEGER('0'&a (7 downto 2)));
wait on clk, a;
end loop;
end process;
end;
```

Package "Main_Module_Package"

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package Main_Module_Package is

component mips
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           pc : out  STD_LOGIC_VECTOR (31 downto 0);
           instr : in  STD_LOGIC_VECTOR (31 downto 0);
           memwrite : out  STD_LOGIC;
           aluout : out  STD_LOGIC_VECTOR (31 downto 0);
           writedata : out  STD_LOGIC_VECTOR (31 downto 0);
           readdata : in  STD_LOGIC_VECTOR (31 downto 0));
end component;

component imem
    port(   a: in STD_LOGIC_VECTOR(5 downto 0);
            rd: out STD_LOGIC_VECTOR(31 downto 0));
end component;

component dmem
    port(   clk, we: in STD_LOGIC;
            a, wd: in STD_LOGIC_VECTOR (31 downto 0);
            rd: out STD_LOGIC_VECTOR (31 downto 0));
end component;

end Main_Module_Package;
```

MAIN_MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.Main_Module_Package.ALL;

entity Main_module is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           writedata : out  STD_LOGIC_VECTOR (31 downto 0);
           dataadr : out  STD_LOGIC_VECTOR (31 downto 0);
           memwrite : out  STD_LOGIC);
end Main_module;

architecture Behavioral of Main_module is

signal memwritet: STD_LOGIC;
signal pc, instr, readdata, dataadrt, writedatat: STD_LOGIC_VECTOR (31 downto 0);

begin

processor: mips port map (clk, reset, pc, instr, memwritet, dataadrt , writedatat, readdata);
instr_mem: imem port map (pc (7 downto 2), instr);
exter_mem: dmem port map (clk, memwritet , dataadrt, writedatat, readdata);

dataadr <= dataadrt;
memwrite <= memwritet;
writedata <= writedatat;

end Behavioral;
```

RTL Diagrams:

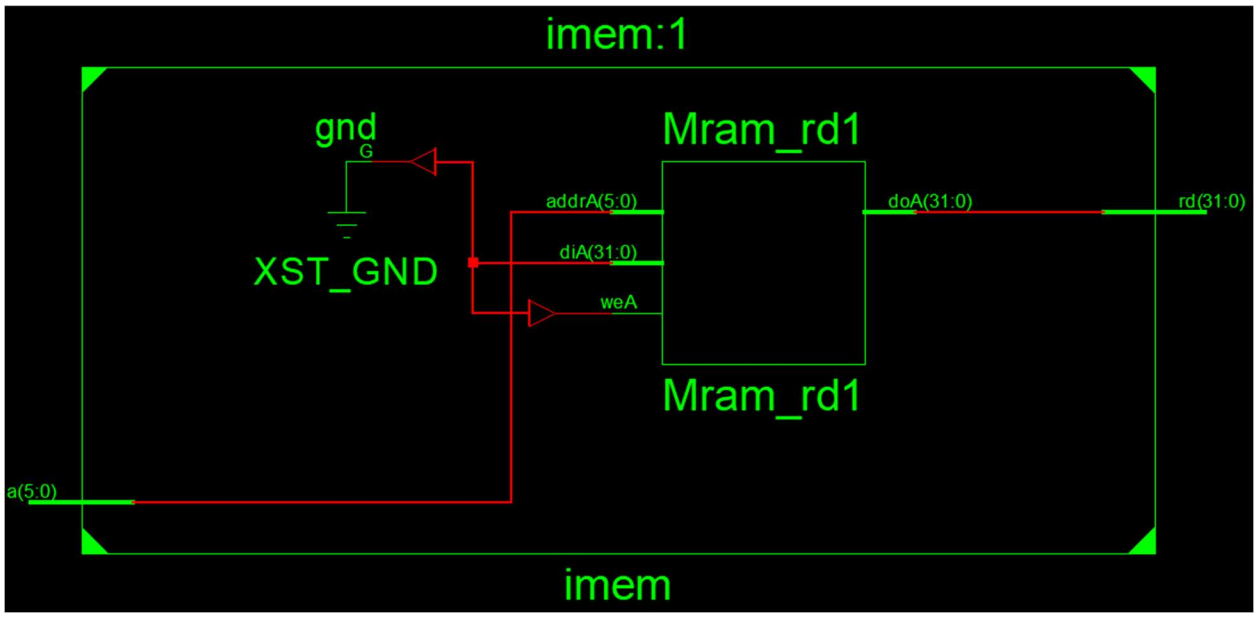

Figure 16: Instruction Memory RTL

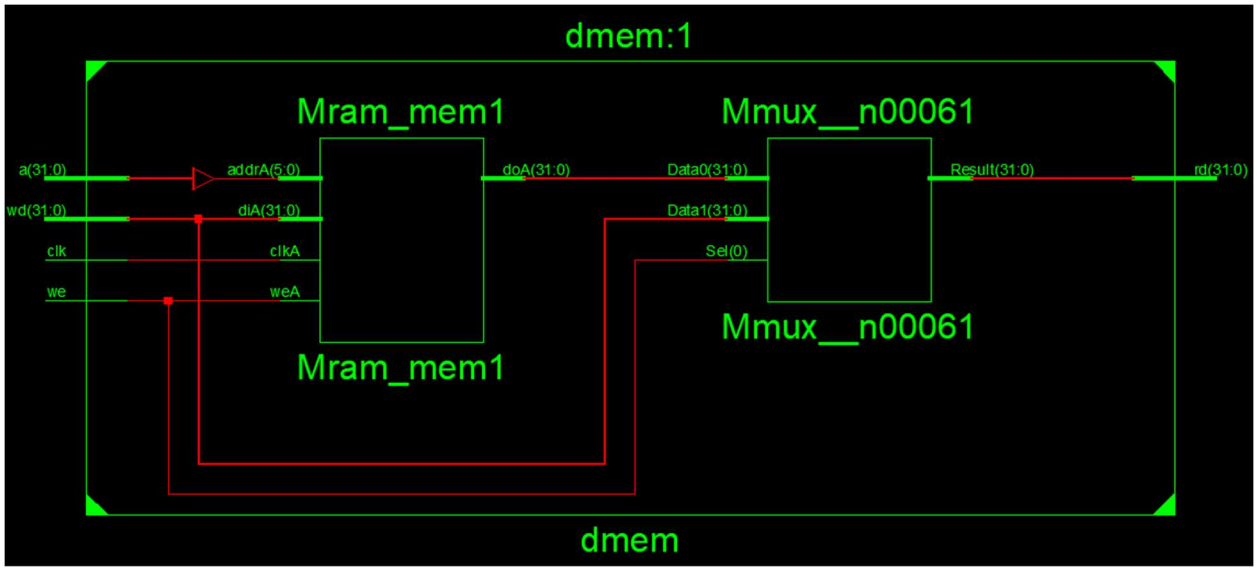

Figure 17: Data Memory RTL

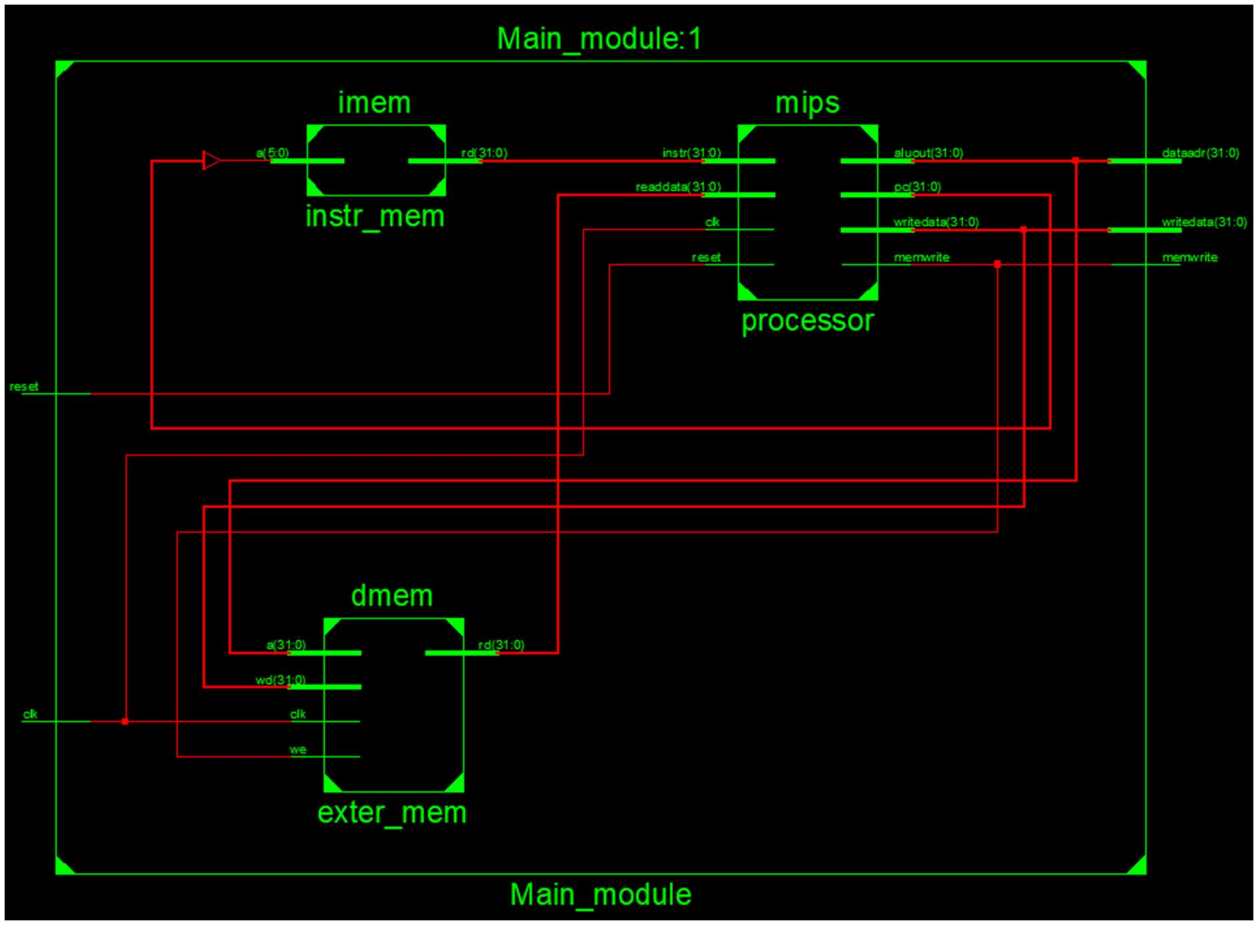

Figure 18: Main Module RTL

Part (6): Testing

After designing the Full Processor, we need to simulate it using the Test File that was handed to us. Memory has a sample program that is written in MIPS Assembly which is translated to Hexadecimal which is then converted to binary when fetching in our processor . **AND HERE WAS THE RESULT!**

Assembly Code:

```
main:   addi $2, $0, 5          # initialize $2 = 5 0 20020005
        addi $3, $0, 12         # initialize $3 = 12 4 2003000c
        addi $7, $3, -9         # initialize $7 = 3 8 2067fff7
        or $4, $7, $2           # $4 = (3 OR 5) = 7 c 00e22025
        and $5, $3, $4          # $5 = (12 AND 7) = 4 10 00642824
        add $5, $5, $4          # $5 = 4 + 7 = 11 14 00a42820
        beq $5, $7, end         # shouldn't be taken 18 10a7000a
        slt $4, $3, $4          # $4 = 12 < 7 = 0 1c 0064202a
        beq $4, $0, around      # should be taken 20 10800001
        addi $5, $0, 0          # shouldn't happen 24 20050000

around: slt $4, $7, $2          # $4 = 3 < 5 = 1 28 00e2202a
        add $7, $4, $5          # $7 = 1 + 11 = 12 2c 00853820
        sub $7, $7, $2          # $7 = 12 - 5 = 7 30 00e23822
        sw $7, 68 ($3)          # [80] = 7 34 ac670044
        lw $2, 80 ($0)          # $2 = [80] = 7 38 8c020050
        j end                   # should be taken 3c 08000011
        addi $2, $0, 1          # shouldn't happen 40 20020001

end:    sw $2, 84 ($0)          # write mem [84] = 7 44 ac020054
```

Figure 19: MIPS Assembly Code Program

Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
use ieee.numeric_std.all;
-- use IEEE.NUMERIC_STD_UNSIGNED.all;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

-- Component Declaration
COMPONENT Main_module
PORT(   clk : in  STD_LOGIC;
        reset : in  STD_LOGIC;
        writedata : out  STD_LOGIC_VECTOR (31 downto 0);
        dataadr : out  STD_LOGIC_VECTOR (31 downto 0);
        memwrite : out  STD_LOGIC);
END COMPONENT;

signal writedata, dataadr: STD_LOGIC_VECTOR(31 downto 0);
signal clk, reset, memwrite: STD_LOGIC;


BEGIN

-- Component Instantiation
dut: Main_module port map(clk, reset, writedata, dataadr, memwrite);


-- Generate clock with 10 ns period
process begin
clk <= '1';
wait for 5 ns;
clk <= '0';
wait for 5 ns;
end process;

-- Generate reset for first two clock cycles
process begin
reset <= '1';
wait for 22 ns;
reset <= '0';
wait;
end process;

-- check that 7 gets written to address 84 at end of program
process(clk) begin
if (clk'event and clk = '0' and memwrite = '1') then
if (CONV_INTEGER(dataadr) = 84 and CONV_INTEGER(writedata) = 7) then
report "NO ERRORS: Simulation succeeded" severity failure;
elsif (CONV_INTEGER(dataadr) = 84) then
report "Simulation failed" severity failure;
end if;
end if;

end process;
end;
```

Simulation Screenshots:

Figure 20: Simulation Binary Screenshot

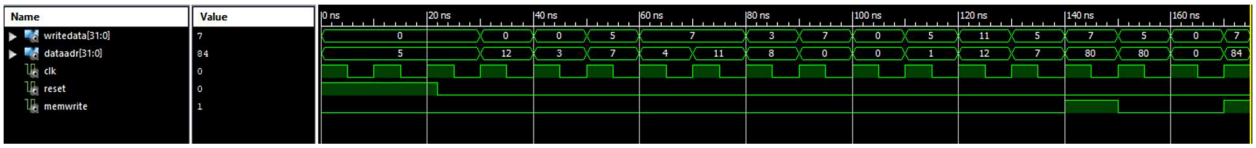

Figure 21: Simulation Signed Numbers Screenshot

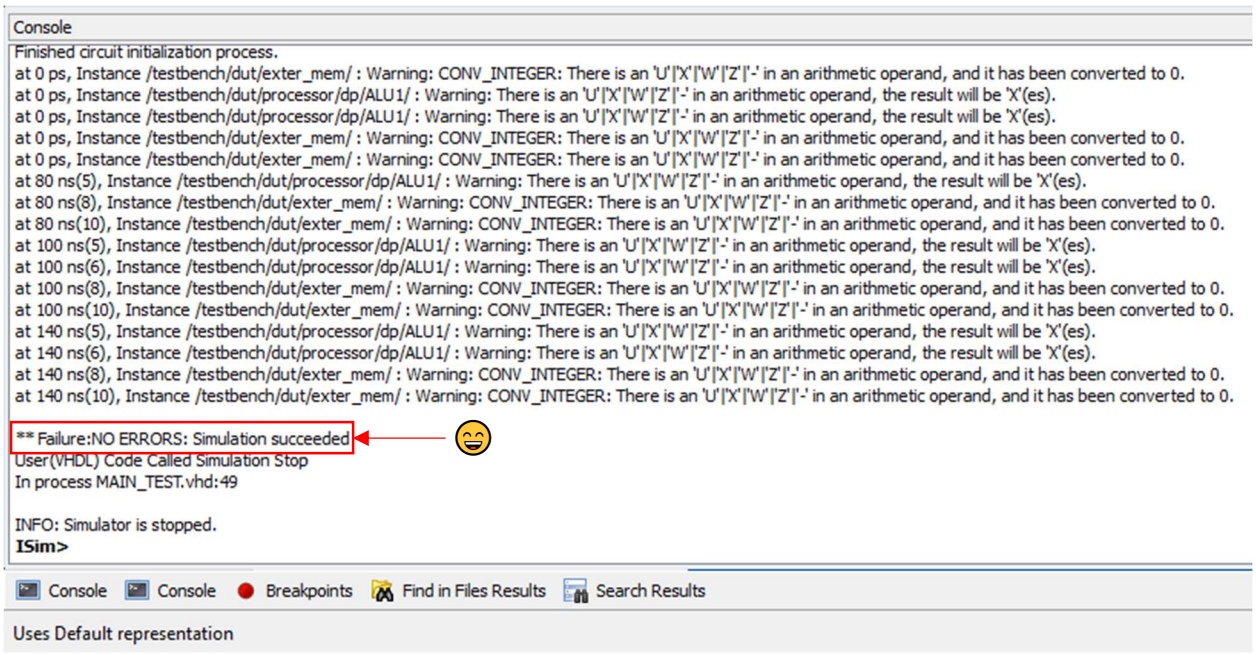

Figure 22: Simulation Console Screenshot

Voilà We Have Our Processor 🎉