VHDL MIPS Processor Components Analysis Report

Register File Component

The register file implementation provides 32 registers, each 32 bits wide, with asynchronous read and synchronous write capabilities. The design includes two read ports and one write port, with registers initialized to specific values for testing purposes. Register $a0 is set to 6, $a1 to 8, and $t0 to 9. The implementation contains a syntax error where "Read_data_1" is incorrectly written as "Read_data_l" in the output assignment. The write operation is properly synchronized to the clock rising edge and gated by the write signal.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
Entity reg_file is
port(
reg_read_1,reg_read_2,reg_write: in std_logic_vector(4 downto 0);
Reg_write_signal,clk: in std_logic;
write_data: in std_logic_vector(31 downto 0);
Read_data_1,Read_data_2: out std_logic_vector(31 downto 0));

end reg_file;

architecture reg of reg_file is
type Regarray is array(0 to 31) of std_logic_vector(31 downto 0);
signal Regfile: Regarray := (
    0  => x"00000000", 1  => x"00000000", 2  => x"00000000", 3  => x"00000000", 4  => x"00000006", -- $a0 = 6
    5  => x"00000008", -- $a1 = 8
    6  => x"00000000",
    7  => x"00000000",
    8  => x"00000000", -- $t0
    9  => x"00000000",10 => x"00000000", 11 => x"00000000",12 => x"00000000",13 => x"00000000",
    14 => x"00000000",
    15 => x"00000000",
    16 => x"00000000", -- $s0
    17 => x"00000000", -- $s1
    others => x"00000000"
  );
begin
Read_data_1<=Regfile(to_integer(unsigned(reg_read_1)));
Read_data_2<=Regfile(to_integer(unsigned(reg_read_2)));
process(clk)
begin
        if rising_edge(clk) and reg_write_signal='1' then

        regfile(to_integer(unsigned(reg_write)))<=write_data;
end if;
end process;
end reg;
```

Program Counter Component

The pc_counter entity implements a basic program counter with synchronous reset functionality. When reset is active high, the counter clears to zero. On rising clock edges when reset is inactive, the counter loads the input value. The output reflects the current counter value through a direct signal assignment.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pc_counter is
    port (
        clk, reset : in std_logic;
        pc_in : in std_logic_vector(31 downto 0);
        pc_out : out std_logic_vector(31 downto 0)
    );
end entity;

architecture behavior of pc_counter is
    signal pc_reg : std_logic_vector(31 downto 0) := (others => '0');
begin
    process(clk, reset)
    begin
        if reset = '1' then
            pc_reg <= (others => '0');
        elsif rising_edge(clk) then
            pc_reg <= pc_in;
        end if;
    end process;

    pc_out <= pc_reg;
end architecture;
```

Multiplexer Components

Two multiplexer implementations are present: a 5-bit version (mux_5bit) and a 32-bit version (mux2_1). Both follow the same design pattern, selecting between two inputs based on a control signal. The 5-bit mux handles register addresses while the 32-bit mux manages data paths. The implementations are purely combinational and correctly use a process sensitive to all inputs.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux_5bit is
port(
        inl:in std_logic_vector (4 downto 0);
        in2: in std_logic_vector (4 downto 0);
        sel:in std_logic;
        output: out std_logic_vector (4 downto 0));
end mux_5bit;

architecture mux of mux_5bit is
begin
process(sel,inl,in2)
begin
IF sel='1' then
output<=in2;
else
output<=inl;
end if;
end process;
end mux;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity mux2_1 is
    port(
        inl   : in  std_logic_vector(31 downto 0);
        in2   : in  std_logic_vector(31 downto 0);
        sel   : in  std_logic;
        mux_out : out std_logic_vector(31 downto 0)
    );
end mux2_1;

architecture mux of mux2_1 is
begin
    process(sel,inl,in2)
    begin
        if sel = '1' then
            mux_out <= in2;
        else
            mux_out <= inl;
        end if;
    end process;
end mux;
```

Memory Components

Data Memory

The data memory (mem) is byte-addressable with 36 bytes of storage initialized with test patterns. The memory operations are properly synchronized to the clock, with separate paths for read and write operations using big-endian byte ordering.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity mem is
port(Memread, memwrite,clk:in std_logic;
Address,writedata: in std_logic_vector(31 downto 0);
Readdata:out std_logic_vector(31 downto 0));
end mem;

architecture unit of mem is
type memarray is array(0 to 35) of std_logic_vector (7 downto 0);
 signal memfile: memarray := (
    0  => x"AB", 1  => x"CD", 2  => x"EF", 3  => x"00",   -- Memory[0]
    4  => x"75", 5  => x"74", 6  => x"65", 7  => x"72",   -- Memory[1]
    8  => x"20", 9  => x"41", 10 => x"72", 11 => x"63",   -- Memory[2]
    12 => x"68", 13 => x"69", 14 => x"74", 15 => x"65",   -- Memory[3]
    16 => x"12", 17 => x"34", 18 => x"56", 19 => x"78",   -- Memory[4]
    20 => x"7F", 21 => x"7F", 22 => x"6D", 23 => x"6D",   -- Memory[5]
    24 => x"00", 25 => x"00", 26 => x"00", 27 => x"00",   -- Memory[6]
    28 => x"78", 29 => x"78", 30 => x"6A", 31 => x"6A",   -- Memory[7]
    32 => x"00", 33 => x"00", 34 => x"00", 35 => x"01"    -- Memory[8]
  );
```

```
  );
begin
process(clk)
    variable addr : integer;
begin
    if rising_edge(clk) then
        addr := to_integer(unsigned(Address));

        if memwrite = '1' and memread = '0' then
            memfile(addr)     <= writedata(31 downto 24);
            memfile(addr + 1) <= writedata(23 downto 16);
            memfile(addr + 2) <= writedata(15 downto 8);
            memfile(addr + 3) <= writedata(7 downto 0);

        elsif memread = '1' then
            Readdata(31 downto 24) <= memfile(addr);
            Readdata(23 downto 16) <= memfile(addr + 1);
            Readdata(15 downto 8)  <= memfile(addr + 2);
            Readdata(7 downto 0)   <= memfile(addr + 3);
        end if;
    end if;
end process;
end unit;
```

Instruction Memory

The instruction memory (inst_memory) stores MIPS instructions in big-endian format across 24 bytes (6 instructions). The address calculation correctly implements word-aligned access (PC/4) .The stored instructions represent valid MIPS operations including add, store word, load word, branch equal, set less than, and subtract.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity inst_memory is
port(
        clk:in std_logic;
        Readdata:out std_logic_vector(31 downto 0);
        Address: in std_logic_vector(31 downto 0)

);
end inst_memory;

architecture memory of inst_memory is
    type memarray is array (0 to 23) of std_logic_vector (7 downto 0);
    signal instfile : memarray := (
        -- add $v0, $a0, $a1 => 0x00851020
        0  => x"00", 1 => x"85", 2 => x"10", 3 => x"20",
        -- sw $v0, 8($zero) => 0xAC020008
        4  => x"AC", 5 => x"02", 6 => x"00", 7 => x"08",
        -- lw $a2, 8($zero) => 0x8C060008
        8  => x"8C", 9 => x"06", 10 => x"00", 11 => x"08",
        -- beq $v0, $a2, 1 => 0x10460001
        12 => x"10", 13 => x"46", 14 => x"00", 15 => x"01",
        -- slt $s1, $v0, $a2 => 0x0046202A
        16 => x"00", 17 => x"46", 18 => x"20", 19 => x"2A",
        -- sub $s1, $a1, $a0 => 0x00A48022
        20 => x"00", 21 => x"A4", 22 => x"80", 23 => x"22"
    );

    signal word_addr : integer;
begin
```

```
    signal word_addr : integer;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            word_addr <= to_integer(unsigned(Address(9 downto 2)));  -- >> PC / 4
            Readdata(31 downto 24) <= instfile(word_addr * 4);
            Readdata(23 downto 16) <= instfile(word_addr * 4 + 1);
            Readdata(15 downto 8)  <= instfile(word_addr * 4 + 2);
            Readdata(7 downto 0)   <= instfile(word_addr * 4 + 3);
        end if;
    end process;
end memory;
```

Control Unit Components

Main Control

The control unit decodes the 6-bit opcode to generate 9 control signals for pipeline management. It handles R-type instructions (opcode "000000"), load word ("100011"), store word ("101011"), and branch equal ("000100").

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity control is
port( op : in std_logic_vector(5 downto 0);
clk:in std_logic;
RegDst,AluSrc,MemtoReg,Regwrite,MemRead,Memwrite,Branch,Aluop1,Aluop0:out std_logic);
end control;

architecture unit of control is
begin
process(clk,op)
begin
if op="000000" then
        RegDst<='1';
        AluSrc<='0';
        MemtoReg<='0';
        Regwrite<='1';
        MemRead<='0';
        Memwrite<='0';
        Branch<='0';
        Aluop1<='1';
        Aluop0<='0';
elsif op="100011" then
        RegDst<='0';
        AluSrc<='1';
        MemtoReg<='1';
        Regwrite<='1';
        MemRead<='1';
        Memwrite<='0';
        Branch<='0';
        Aluop1<='0';
        Aluop0<='0';
```

```
elsif op="101011" then
        AluSrc<='1';
        Regwrite<='0';
        MemRead<='0';
        Memwrite<='1';
        Branch<='0';
        Aluop1<='0';
        Aluop0<='0';
elsif op="000100" then

        AluSrc<='0';
        Regwrite<='0';
        MemRead<='0';
        Memwrite<='0';
        Branch<='1';
        Aluop1<='0';
        Aluop0<='1';
end if;
end process;
end unit;
```

ALU Control

The ALU control unit maps function fields and ALUop signals to 4-bit operation codes.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_Arith.all;
use ieee.std_logic_unsigned.all;
entity alucontrol is
port( funct_field:in std_logic_vector (5 downto 0);
        Aluop:in std_logic_vector (1 downto 0);
        op:out std_logic_vector (3 downto 0));

end alucontrol;

architecture unit of alucontrol is
begin
process(aluop,funct_field)
begin
if Aluop="00" and funct_field="------" then op<="0010";
elsif Aluop="01" and funct_field="------" then op<="0110";
elsif Aluop="10" and funct_field="100000" then op<="0010";-- add
elsif Aluop="10" and funct_field="100010" then op<="0110"; --sub
elsif Aluop="10" and funct_field="100100" then op<="0000"; --and
elsif Aluop="10" and funct_field="100101" then op<="0001"; -- or
elsif Aluop="10" and funct_field="101010" then op<="0111"; -- set

end if;
end process;
end unit;
```

Arithmetic Logic Unit

The ALU performs operations based on 4-bit control codes, supporting AND, OR, addition, subtraction, set less than, and NOR operations. The zero flag is set when inputs are equal.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_Arith.all;
use ieee.std_logic_unsigned.all;
entity alu is
port(
control: in std_logic_vector (3 downto 0);
input_1: in std_logic_vector (31 downto 0);
input_2: in std_logic_vector (31 downto 0);
zero:  out std_logic;
result: out std_logic_vector(31 downto 0));
end alu;

architecture au of alu is
begin
process(control,input_1,input_2)
begin
case control is
when "0000" =>
 result <= input_1 and input_2;
when "0001" =>
        result <= input_1 or input_2;
when "0010" =>
result<= input_1 + input_2;
when "0110" =>
result<= input_1-input_2;
when "0111"=>
if input_1 < input_2 then result<=(others=>'0');
result(0)<='1';
else
result<=(others=>'0');
end if;
when "1100" =>
result <= input_1 nor input_2;
when others =>
end case;
```

```
end case;
if input_1 = input_2 then zero <='1';
else
zero<='0';
end if;
end process;
end au;
```

Adder Component

The 32-bit adder implementation correctly handles carry propagation using a 33-bit temporary sum. It extends inputs by one bit to properly capture the carry-out value. The design is combinational and properly assigns both the sum result and carry flag.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_Arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
    port(
        A, B  : in  std_logic_vector(31 downto 0);
        Sum   : out std_logic_vector(31 downto 0);
        Carry : out std_logic
    );
end adder;

architecture adder32 of adder is
 signal tmp_sum : std_logic_vector(32 downto 0); -- 1 bit wider for carry

begin
    process(A, B)

    begin

        tmp_sum <= ('0' &A) + ('0' & B); -- extend to 33 bits

    end process;
 Sum   <= tmp_sum(31 downto 0); -- sum result
        Carry <= tmp_sum(32); -- carry-out bit
end adder32;
```

Sign Extend:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sign is
    port(
        A : in  std_logic_vector(15 downto 0);
        B : out std_logic_vector(31 downto 0)
    );
end sign;

architecture extend of sign is
begin
    process(A)
        variable A_signed : signed(15 downto 0);
        variable B_signed : signed(31 downto 0);
    begin
        A_signed := signed(A);           -- Convert std_logic_vector to signed
        B_signed := resize(A_signed, 32); -- Sign-extend to 32 bits
        B <= std_logic_vector(B_signed);  -- Convert back to std_logic_vector
    end process;
end extend;
```

Top-Level MIPS Processor

The Mips entity instantiates all major processor components including program counter, register file, memories, ALU, control units, and multiplexers. The architecture shows proper component declarations.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Mips is
    port(
        clk   : in std_logic;
        reset : in std_logic
    );
end Mips;

architecture single_cycle of Mips is

    component mux2_1 is
        port(
            in1, in2 : in std_logic_vector(31 downto 0);
            sel      : in std_logic;
            mux_out   : out std_logic_vector(31 downto 0)
        );
    end component;

    component reg_file is
        port(
            reg_read_1, reg_read_2, reg_write : in std_logic_vector(4 downto 0);
            Reg_write_signal, clk             : in std_logic;
            write_data                         : in std_logic_vector(31 downto 0);
            Read_data_1, Read_data_2           : out std_logic_vector(31 downto 0)
        );
    end component;

    component mem is
        port(
            Memread, memwrite, clk : in std_logic;
            Address, writedata     : in std_logic_vector(31 downto 0);
            Readdata               : out std_logic_vector(31 downto 0)
        );
    end component;
```

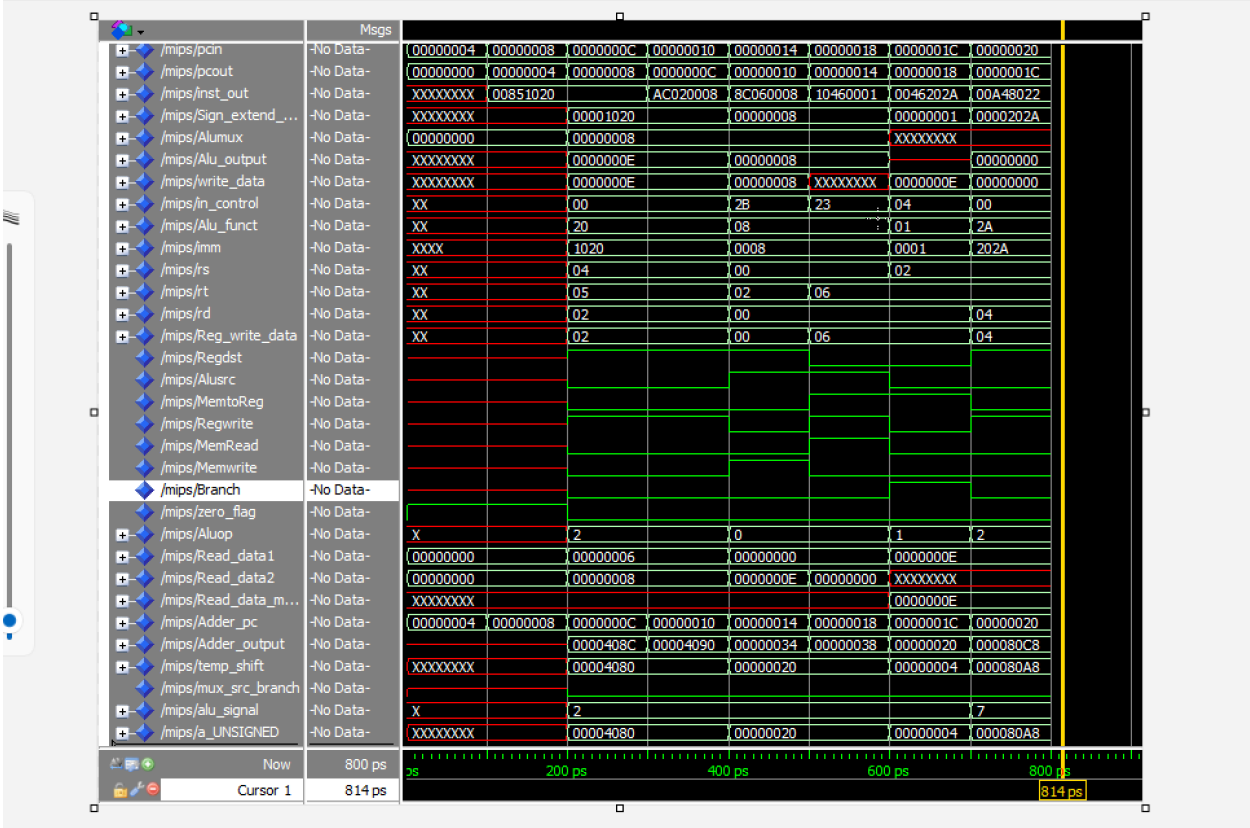
```
    component mem is
        port(
            Memread, memwrite, clk : in std_logic;
            Address, writedata     : in std_logic_vector(31 downto 0);
            Readdata               : out std_logic_vector(31 downto 0)
        );
    end component;

    component sign is
        port(
            A : in  std_logic_vector(15 downto 0);
            B : out std_logic_vector(31 downto 0)
        );
    end component;

    component pc_counter is
        port(
            clk, reset : in std_logic;
            pc_in      : in std_logic_vector(31 downto 0);
            pc_out     : out std_logic_vector(31 downto 0)
        );
    end component;

    component control is
        port(
            op                         : in std_logic_vector(5 downto 0);
            clk                        : in std_logic;
            RegDst, AluSrc, MemtoReg : out std_logic;
            Regwrite, MemRead, Memwrite, Branch : out std_logic;
            Aluop1, Aluop0           : out std_logic
        );
    end component;
```

```
    component alucontrol is
        port(
            funct_field : in std_logic_vector(5 downto 0);
            Aluop       : in std_logic_vector(1 downto 0);
            op          : out std_logic_vector(3 downto 0)
        );
    end component;

    component alu is
        port(
            control       : in std_logic_vector(3 downto 0);
            input_1, input_2 : in std_logic_vector(31 downto 0);
            zero          : out std_logic;
            result        : out std_logic_vector(31 downto 0)
        );
    end component;

    component adder is
        port(
            A, B  : in std_logic_vector(31 downto 0);
            Sum   : out std_logic_vector(31 downto 0);
            Carry : out std_logic
        );
    end component;

    component inst_memory is
        port(
            clk       : in std_logic;
            Readdata  : out std_logic_vector(31 downto 0);
            Address   : in std_logic_vector(31 downto 0)
        );
    end component;
```

```
    component mux_5bit is
        port(
            in1, in2 : in std_logic_vector(4 downto 0);
            sel      : in std_logic;
            output   : out std_logic_vector(4 downto 0)
        );
    end component;

    signal pcin, pcout, inst_out, Sign_extend_out, Alumux, Alu_output, write_data : std_logic_vector(31 downto 0) := (others => '0')
    signal in_control, Alu_funct : std_logic_vector(5 downto 0);
    signal imm : std_logic_vector(15 downto 0);
    signal rs, rt, rd, Reg_write_data : std_logic_vector(4 downto 0);
    signal Regdst, Alusrc, MemtoReg, Regwrite, MemRead, Memwrite, Branch, zero_flag : std_logic := '0';
    signal Aluop : std_logic_vector(1 downto 0);
    signal Read_data1, Read_data2, Read_data_mem : std_logic_vector(31 downto 0);
    signal Adder_pc, Adder_output, temp_shift : std_logic_vector(31 downto 0);
    signal mux_src_branch : std_logic;
    signal alu_signal : std_logic_vector(3 downto 0);
    signal a_UNSIGNED : unsigned(31 downto 0);

egin
    adder_pc_counter : adder port map(pcout, X"00000004", Adder_pc, open);
    a_UNSIGNED <= shift_left(unsigned(Sign_extend_out), 2);
    temp_shift <= std_logic_vector(a_UNSIGNED);
    adder_branch : adder port map(Adder_pc, temp_shift, Adder_output, open);
    mux_src_branch <= Branch and zero_flag;
    mux_3 : mux2_1 port map(Adder_pc, Adder_output, mux_src_branch, pcin);
        pc_inst : pc_counter port map(clk, reset, pcin, pcout);

    process(inst_out,clk)
    begin
        if rising_edge(clk) then
        in_control <= inst_out(31 downto 26);
        rs <= inst_out(25 downto 21);
        rt <= inst_out(20 downto 16);
        rd <= inst_out(15 downto 11);
        imm <= inst_out(15 downto 0);
        Alu_funct <= inst_out(5 downto 0);

end if;
    end process;


    inst_mem : inst_memory port map(clk, inst_out, pcout);
    contorl_unit : control port map(in_control, clk, Regdst, Alusrc, MemtoReg, Regwrite, MemRead, Memwrite, Branch, Aluop(1), Aluop(0));
    mux_1 : mux_5bit port map(rt, rd, Regdst, Reg_write_data);
        mux_write_data_reg : mux2_1 port map(Alu_output, Read_data_mem, MemtoReg, write_data);
    register_file : reg_file port map(rs, rt, Reg_write_data, Regwrite, clk, write_data, Read_data1, Read_data2);
    sign_extend : sign port map(imm, Sign_extend_out);
    mux_2 : mux2_1 port map(Read_data2, Sign_extend_out, Alusrc, Alumux);
    Alu_con : alucontrol port map(Alu_funct, Aluop, alu_signal);
    Alu_unit : alu port map(alu_signal, Read_data1, Alumux, zero_flag, Alu_output);
    mem_unit : mem port map(MemRead, Memwrite, clk, Alu_output, Read_data2, Read_data_mem);


end single_cycle;
```

Results:

Assembly	Hex
`add $v0, $a0, $a1`	`0x00851020`
`sw $v0, 8($zero)`	`0xAC020008`
`lw $a2, 8($zero)`	`0x8C060008`
`beq $v0, $a2` *(offset=0)*	`0x10460000`
`slt $s1, $v0, $a2`	`0x0046882A`
`sub $s1, $a1, $a0`	`0x00A48822`



General Observations

The code collection represents a nearly complete single-cycle MIPS processor implementation in VHDL. Several components contain minor syntax errors and typographical mistakes that would prevent successful compilation. The design follows conventional MIPS architecture with separate instruction and data memories, register file, and control units. Memory implementations use big-endian byte ordering consistent with MIPS conventions. Initial values in registers and memory suggest the design was prepared for testing specific instruction sequences.