CPU Lite Design

By: Stephen Ude and Evan Mow

This lab focuses on the design and verification of a MIPS Single Cycle CPU Lite design capable of running R type, I type, and J type instructions. To make up a CPU Lite design, Figure 1 presents the necessary components needed. Due to the Instruction and Data Memory creation choice, the design shown in Figure 1 has been changed to be word accessible rather than Byte accessible. This means the PC will increment by 1 rather than 4 and the shift left by 2 components shown in Figure 1, has been removed from our design.

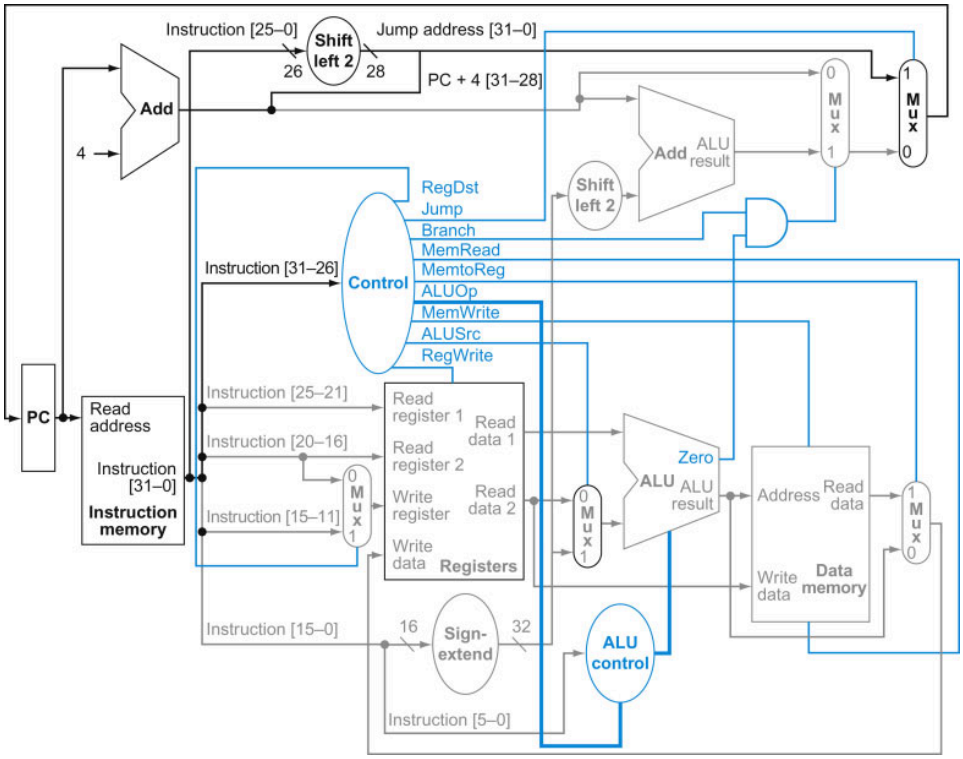

Figure 1: Single-Cycle MIPS CPU Lite Schematic

Design:

To start the design, creating the control unit was our first priority. The control unit is the brain of the CPU. It tells each component what to do in order to successfully run the command. Due to the simplicity of the design, the ALU Control and the Control Unit were merged into a single source file. The chosen instructions for this design are the following commands: ADD, SUB, ADDI, BEQ, J, SLL, LW, and SW. The I type instructions are LW, SW, ADDI, and BEQ.

The R type instructions are ADD, SUB, and SLL. Provided in Figures 2-4 is the code used to implement the control unit.

src\Control_Unit_Code.txt

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

entity control_unit is
Port(

instruction : in std_logic_vector (5 downto 0);
funct : in std_logic_vector (5 downto 0);
RegDst : out std_logic;
jump : out std_logic;
branch : out std_logic;
MemRead : out std_logic;
MemtoReg : out std_logic;
MemWrite : out std_logic;
ALUsrc : out std_logic;
RegWrite : out std_logic;
ALUop : out std_logic_vector(2 downto 0)
);
end entity;

architecture Behavioral of control_unit is

begin

process(instruction, funct) begin

RegDst   <= '0';
RegWrite <= '0';
ALUsrc   <= '0';
MemRead  <= '0';
MemWrite <= '0';
branch   <= '0';
jump     <= '0';
MemtoReg <= '0';
ALUop <= "111";

case instruction is

-- R-type instructions
    when "000000" =>
        RegDst    <= '1';
        RegWrite  <= '1';
        ALUsrc    <= '0';
        MemRead   <= '0';
        MemWrite  <= '0';
        branch    <= '0';
        jump      <= '0';
        MemtoReg  <= '0';

        case funct is
        --ADD
            when "100000" =>
```

Figure 2: First Part of Control Unit code

```
                ALUop <= "000";
                --SUB
            when "100010" =>
                ALUop <= "001";
                --Logical Shift Left
            when "000000" =>
                ALUop <= "010";

             --Null
            when others =>
                ALUop <= "111";
  end case;

-- LW
when "100011" =>
  RegDst    <= '0';
  RegWrite  <= '1';
  ALUsrc    <= '1';
  MemRead   <= '1';
  MemWrite  <= '0';
  branch    <= '0';
  jump      <= '0';
  MemtoReg  <= '1';
  ALUop <= "011";

-- SW
when "101011" =>
  RegDst    <= '0';
  RegWrite  <= '0';
  ALUsrc    <= '1';
  MemRead   <= '0';
  MemWrite  <= '1';
  branch    <= '0';
  jump      <= '0';
  MemtoReg  <= '0';
  ALUop <= "100";

-- BEQ
when "000100" =>
  RegDst    <= '0';
  RegWrite  <= '0';
  ALUsrc    <= '0';
  MemRead   <= '0';
  MemWrite  <= '0';
  branch    <= '1';
  jump      <= '0';
  MemtoReg  <= '0';
  ALUop <= "101";

 -- ADDI
when "001000" =>
  RegDst    <= '0';
  RegWrite  <= '1';
  ALUsrc    <= '1';
  MemRead   <= '0';
  MemWrite  <= '0';
```

Figure 3: Second Part of Control Unit code

```
  branch    <= '0';
  jump      <= '0';
  MemtoReg  <= '0';
  ALUop <= "110";

-- JUMP
when "000010" =>
  RegDst    <= '0';
  RegWrite  <= '0';
  ALUsrc    <= '0';
  MemRead   <= '0';
  MemWrite  <= '0';
  branch    <= '0';
  jump      <= '1';
  MemtoReg  <= '0';
  ALUop <= "111";


when others =>
null;

end case;
end process;

end Behavioral;
```

Figure 4: Third Part of Control Unit Code

The next part of the design was to create the register file. This component is responsible for assigning the correct bits derived from the instruction bit sequence, to the Rs, Rt and Rd registers. This component interprets which registers are what and what their roles play in the instruction being passed through the CPU Lite Design. Provided in Figures 5-6 is the code used to implement the register file.

src\Register_file_code.txt

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

entity reg_file is
    Port (
        Clk        : in  std_logic;
        Rst        : in  std_logic;
        RegWrite   : in  std_logic;
        WriteReg   : in  std_logic_vector(4 downto 0);
        WriteData  : in  std_logic_vector(31 downto 0);
        ReadReg1   : in  std_logic_vector(4 downto 0);
        ReadData1  : out std_logic_vector(31 downto 0);
        ReadReg2   : in  std_logic_vector(4 downto 0);
        ReadData2  : out std_logic_vector(31 downto 0)
    );
end reg_file;

architecture arch of reg_file is
    type reg_array is array (0 to 31) of std_logic_vector(31 downto 0);
    signal regs : reg_array;
begin

    -- Synchronous write and reset
    process(Clk, Rst)
    begin
        if Rst = '1' then
            regs(0)  <= x"00000000";
            regs(1)  <= x"00000000";
            regs(2)  <= x"00000000";
            regs(3)  <= x"00000000";
            regs(4)  <= x"00000000";
            regs(5)  <= x"00000000";
            regs(6)  <= x"00000000";
            regs(7)  <= x"00000000";
            regs(8)  <= x"00000000";
            regs(9)  <= x"00000000";
            regs(10) <= x"00000000";
            regs(11) <= x"00000000";
            regs(12) <= x"00000000";
            regs(13) <= x"00000000";
            regs(14) <= x"00000000";
            regs(15) <= x"00000000";
            regs(16) <= x"00000000";
            regs(17) <= x"00000000";
            regs(18) <= x"00000000";
            regs(19) <= x"00000000";
            regs(20) <= x"00000000";
            regs(21) <= x"00000000";
            regs(22) <= x"00000000";
            regs(23) <= x"00000000";
            regs(24) <= x"00000000";
            regs(25) <= x"00000000";
```

Figure 5: Part 1 of the Register File

```
            regs(26) <= x"00000000";
            regs(27) <= x"00000000";
            regs(28) <= x"00000000";
            regs(29) <= x"00000000";
            regs(30) <= x"00000000";
            regs(31) <= x"00000000";
        elsif rising_edge(Clk) then
            if RegWrite = '1' then
                    regs(to_integer(unsigned(WriteReg))) <= WriteData;
            end if;
        end if;
    end process;

    -- Async read ports
    ReadData1 <= x"00000000" when ReadReg1 = "00000" else regs(to_integer(unsigned(ReadReg1)));
    ReadData2 <= x"00000000" when ReadReg2 = "00000" else regs(to_integer(unsigned(ReadReg2)));

end arch;
```

Figure 6: Part two of the Register File Code

The next step was to implement the ALU. The ALU is responsible for which arithmetic operation is done to the Rs and Rt registers or the Rs and immediate value, depending on which instruction is called. The ALU is also responsible for outputting flags, such as the overflow, carry, negative, and zero flags, to which all have their own unique purposes in the CPU Lite design. The flags used in this design are the zero flag, responsible for whether a branch is taken or not, and the negative flag, which is used to determine whether the result is a negative number. Provided in Figure 7 is the implemented ALU.

src\ALU_Code.txt

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

entity ALU is
    Port ( Read_Data1 : in STD_LOGIC_VECTOR (31 downto 0);
           Read_Data2 : in STD_LOGIC_VECTOR (31 downto 0);
           ALUop : in STD_LOGIC_VECTOR (2 downto 0);
           flags : out STD_LOGIC_VECTOR (1 downto 0); --N, Z
           ALUout : out STD_LOGIC_VECTOR (31 downto 0));
end ALU;

architecture Behavioral of ALU is
    signal ALUout_int: std_logic_vector(31 downto 0);
    signal ALUout_int2: std_logic_vector(32 downto 0);


begin

process(Read_Data1, Read_Data2, ALUop) begin

case ALUop is
    when "000" => --ADD R type
        ALUout_int2 <= std_logic_vector(unsigned('0' & Read_Data1) + unsigned('0' & Read_Data2));
    when "001" => --SUB Rtype
        ALUout_int2 <= std_logic_vector(signed('0' & Read_Data1) - signed('0' & Read_Data2));
    when "010" => --LW
        ALUout_int2 <= std_logic_vector(unsigned('0' & Read_Data1) + unsigned('0' & Read_Data2));
    when "011" => --SW
        ALUout_int2 <= std_logic_vector(unsigned('0' & Read_Data1) + unsigned('0' & Read_Data2));
    when "100" => --BEQ
        ALUout_int2 <= std_logic_vector(signed('0' & Read_Data1) - signed('0' & Read_Data2));
    when "101" => --ADD I type
        ALUout_int2 <= std_logic_vector(unsigned('0' & Read_Data1) + unsigned('0' & Read_Data2));
    when "110" => --Logic Shift Left
        ALUout_int2 <= '0' & std_logic_vector(shift_left(unsigned(Read_Data2), to_integer(unsigned(Read_Data1(4 downto 0)))));
    when others =>
        ALUout_int2 <= (others => '0');
    end case;

end process;

ALUout_int <= ALUout_int2(31 downto 0);
ALUout <= ALUout_int;
flags(1) <= ALUout_int2(31);--negative flag
flags(0) <= '1' when ALUout_int = X"00000000" else '0'; --zero flag



end Behavioral;
```

Figure 7: Code to implement the ALU

To implement the Instruction Memory and the Data Memory, we utilized the IP catalog Block Memory Generators, and selected Single Port ROM (Read Only Memory) for the Instruction Memory and Single Port RAM (Random Access Memory) for the Data Memory.

Once all the components were created, the top layer of the CPU Lite Design was created. The previously created components are to be instantiated within this top layer such that the functionality of the components can be run through just a single source file. The rest of the logic provided in Figure 1, such as the multiplexers and additional adders, along with the signals needed to connect all the components together, was done in this source file. The result is a fully functional Single Cycle CPU Lite Design capable of running R type(Excluding the SLL instruction), I type, and J type instructions provided in the control unit. Provided in Figures 8-12 is the code used to create the CPU top layer.

src\CPU_Top_Code.txt

```
--CPU Top File
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CPU_top is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           instruction_check : out std_logic_vector(31 downto 0);
           PC_check : out std_logic_vector(31 downto 0);
           reg1_check, reg2_check, reg_write_check : out std_logic_vector(4 downto 0);
           write_data_check : out std_logic_vector(31 downto 0);
           Data_mem_output_check : out std_logic_vector(31 downto 0);
           ALU_out_check : out std_logic_vector(31 downto 0);
           PC_adder_check: out std_logic_vector(31 downto 0);
           jump_address_checker: out std_logic_vector(31 downto 0);
           Branch_add_output_checker : out std_logic_vector(31 downto 0);
           branch_mux_out_checker : out std_logic_vector(31 downto 0);
           PC_mux_out_checker : out std_logic_vector(31 downto 0);
           data_mem_write_checker : out std_logic_vector(31 downto 0);
           sign_extend_input_checker : out std_logic_vector(15 downto 0);
           Read_Data1_check, Read_Data2_check : out std_logic_vector(31 downto 0);
           control_instruction_check : out std_logic_vector(5 downto 0);
           funct_checker : out std_logic_vector(5 downto 0)
           );
end CPU_top;

architecture Behavioral of CPU_top is

--Components
component ALU is
    Port ( Read_Data1 : in STD_LOGIC_VECTOR (31 downto 0);
           Read_Data2 : in STD_LOGIC_VECTOR (31 downto 0);
           ALUop : in STD_LOGIC_VECTOR (2 downto 0);
           flags : out STD_LOGIC_VECTOR (1 downto 0); --N, Z
           ALUout : out STD_LOGIC_VECTOR (31 downto 0));
end component;

component reg_file is
    Port (
        Clk        : in  std_logic;
        Rst        : in  std_logic;
        RegWrite   : in  std_logic;
        WriteReg   : in  std_logic_vector(4 downto 0);
        WriteData  : in  std_logic_vector(31 downto 0);
        ReadReg1   : in  std_logic_vector(4 downto 0);
        ReadData1  : out std_logic_vector(31 downto 0);
        ReadReg2   : in  std_logic_vector(4 downto 0);
        ReadData2  : out std_logic_vector(31 downto 0)
    );
end component;

component control_unit is
Port(
```

Figure 8: Part one of the CPU top Layer

```
instruction : in std_logic_vector (5 downto 0);
funct : in std_logic_vector (5 downto 0);
RegDst : out std_logic;
jump : out std_logic;
branch : out std_logic;
MemRead : out std_logic;
MemtoReg : out std_logic;
MemWrite : out std_logic;
ALUsrc : out std_logic;
RegWrite : out std_logic;
ALUop : out std_logic_vector(2 downto 0)
);
end component;

COMPONENT blk_mem_gen_0
  PORT (
    clka : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

COMPONENT blk_mem_gen_1
  PORT (
    clka : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

--Signals

--ALU Port Map signals
signal ALU_out : std_logic_vector(31 downto 0);
signal ALU_flags : std_logic_vector(1 downto 0);
--ALU Logic Signals
signal ALU_mux_out : std_logic_vector(31 downto 0);
signal ALU_zero : std_logic;

--Register File Port Map Signals
signal WriteData, ReadData1, ReadData2 : std_logic_vector(31 downto 0);
signal WriteReg, ReadReg1, ReadReg2 : std_logic_vector(4 downto 0);
signal RegWrite : std_logic;
--Register File Mux signals
signal Reg_file_mux_out, reg_mux_bot : std_logic_vector(4 downto 0);

--Control Unit Port Map Signals
signal control_instruction, funct : std_logic_vector(5 downto 0);
signal RegDst, jump, branch, MemRead, MemtoReg, MemWrite, ALUsrc : std_logic;
signal ALUop : std_logic_vector(2 downto 0);
```

Figure 9: Part two of the CPU top layer

```
--Instruction Memory Port Map Signals
signal PC_out, instruction : std_logic_vector(31 downto 0);

--Data Memory Port Map Signals
signal Data_mem_mux_top : std_logic_vector(31 downto 0);
signal MemWrite1 : std_logic_vector(0 downto 0);
--Data Memory Logic Signals
signal DataMem_mux_out : std_logic_vector(31 downto 0);

--PC Logic signals
signal PC_add_out : std_logic_vector(31 downto 0);
signal PC_mux_out : std_logic_vector(31 downto 0);

--Jump Logic Signals
signal jump_instruction : std_logic_vector(25 downto 0);
signal jump_address : std_logic_vector(31 downto 0);

--Sign Extend Logic
signal sign_extend_input : std_logic_vector(15 downto 0);
signal sign_extend_output : std_logic_vector(31 downto 0);

--Branch Logic Signals
signal branch_add_out, branch_add_bot_input : std_logic_vector(31 downto 0);
signal branch_mux_out : std_logic_vector(31 downto 0);

begin
--Sign Extend Logic
sign_extend_input <= instruction(15 downto 0);
sign_extend_output <= std_logic_vector(resize(signed(sign_extend_input), 32));

--ALU Port Map
ALU_mux_out <= sign_extend_output when ALUsrc = '1' else ReadData2;
ALU_zero <= ALU_flags(0);

ALU_inst : ALU port map(
    Read_Data1 => ReadData1,
    Read_Data2 => ALU_mux_out,
    ALUop => ALUop,
    flags => ALU_flags,
    ALUout => ALU_out
);

--Register File Port Map
ReadReg1 <= instruction(25 downto 21);
ReadReg2 <= instruction(20 downto 16);
reg_mux_bot <= instruction(15 downto 11);
Reg_file_mux_out <= reg_mux_bot when RegDst = '1' else ReadReg2;
WriteReg <= Reg_file_mux_out;
reg_file_inst: reg_file port map(
    Clk => clk,
    Rst => reset,
    RegWrite => RegWrite,
    WriteReg => WriteReg,
    WriteData => WriteData,
    ReadReg1 => ReadReg1,
    ReadData1 => ReadData1,
```

Figure 10: Part three of the CPU top layer

```
    ReadReg2 => ReadReg2,
    ReadData2 => ReadData2
);

--Control Unit Port Map
control_instruction <= instruction(31 downto 26);
funct <= instruction(5 downto 0);
control_unit_Port : control_unit port map(
    instruction => control_instruction,
    funct => funct,
    RegDst => RegDst,
    jump => jump,
    branch => branch,
    MemRead => MemRead,
    MemtoReg => MemtoReg,
    MemWrite => MemWrite,
    ALUsrc => ALUsrc,
    RegWrite => RegWrite,
    ALUop => ALUop
);

--Instruction Memory Port Map
Instruction_mem : blk_mem_gen_0 port map(
    clka => clk,
    ena => '1',
    addra => PC_out(8 downto 0),
    douta => instruction
);

--Data Memory Port Map
MemWrite1(0) <= MemWrite;
DataMem_mux_out <= Data_mem_mux_top when MemtoReg = '1' else ALU_out;
WriteData <= DataMem_mux_out;
Data_Mem : blk_mem_gen_1 port map(
    clka => clk,
    ena => '1',
    wea => MemWrite1,
    addra => ALU_out(8 downto 0),
    dina => ReadData2,
    douta => Data_mem_mux_top
);

--PC Logic
PC_add_out <= std_logic_vector(unsigned(PC_out) + 1);

process(clk, reset) begin
    if reset = '1' then
        PC_out <= (others => '0');
    elsif rising_edge(clk) then
        PC_out <= PC_mux_out;
    end if;
end process;

--Jump Logic
jump_instruction <= instruction(25 downto 0);
jump_address <= PC_add_out(31 downto 26) & jump_instruction;
```

Figure 11: Part four of the CPU top layer

```
PC_mux_out <= jump_address when jump = '1' else branch_mux_out;

--Branch Logic
branch_add_bot_input <= std_logic_vector((unsigned(sign_extend_output)));
branch_add_out <= std_logic_vector(unsigned(branch_add_bot_input) + unsigned(PC_add_out));
branch_mux_out <= branch_add_out when (ALU_flags(0) = '1' and branch = '1') else PC_add_out;

--Output checks
branch_mux_out_checker <= branch_mux_out;
instruction_check <= instruction;
PC_check <= PC_out;
reg1_check <= ReadReg1;
reg2_check <= ReadReg2;
reg_write_check <= WriteReg;
write_data_check <= WriteData;
Data_mem_output_check <= Data_mem_mux_top;
ALU_out_check <= ALU_out;
PC_adder_check <= PC_add_out;
jump_address_checker <= jump_address;
Branch_add_output_checker <= branch_add_out;
PC_mux_out_checker <= PC_mux_out;
data_mem_write_checker <= ReadData2;
sign_extend_input_checker <= sign_extend_input;
Read_Data1_check <= ReadData1;
Read_Data2_check <= ReadData2;
control_instruction_check <= control_instruction;
funct_checker <= funct;

end Behavioral;
```

Figure 12: Part five of the CPU top layer

Once the CPU Top Layer was completed, the simulation file (testbench) was needed in order to test the functionality of the design. To create the testbench file, a clock is needed for the memory components to operate, i.e the instruction memory, register file, and data memory components. Provided in Figures 13-14 is the testbench used to test the CPU Lite design.

src\CPU_Top_Code_tb.txt

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

entity CPU_top_tb is
end CPU_top_tb;

architecture Behavioral of CPU_top_tb is

    -- Component Under Test
    component CPU_top is
        Port (
            clk                          : in  STD_LOGIC;
            reset                        : in  STD_LOGIC;
            instruction_check            : out STD_LOGIC_VECTOR(31 downto 0);
            PC_check                     : out STD_LOGIC_VECTOR(31 downto 0);
            reg1_check                   : out STD_LOGIC_VECTOR(4 downto 0);
            reg2_check                   : out STD_LOGIC_VECTOR(4 downto 0);
            reg_write_check              : out STD_LOGIC_VECTOR(4 downto 0);
            write_data_check             : out STD_LOGIC_VECTOR(31 downto 0);
            Data_mem_output_check        : out STD_LOGIC_VECTOR(31 downto 0);
            ALU_out_check                : out STD_LOGIC_VECTOR(31 downto 0);
            PC_adder_check               : out STD_LOGIC_VECTOR(31 downto 0);
            jump_address_checker         : out STD_LOGIC_VECTOR(31 downto 0);
            Branch_add_output_checker  : out STD_LOGIC_VECTOR(31 downto 0);
            branch_mux_out_checker       : out STD_LOGIC_VECTOR(31 downto 0);
            PC_mux_out_checker           : out STD_LOGIC_VECTOR(31 downto 0);
            data_mem_write_checker       : out STD_LOGIC_VECTOR(31 downto 0);
            sign_extend_input_checker    : out STD_LOGIC_VECTOR(15 downto 0);
            Read_Data1_check             : out STD_LOGIC_VECTOR(31 downto 0);
            Read_Data2_check             : out STD_LOGIC_VECTOR(31 downto 0);
            control_instruction_check : out std_logic_vector(5 downto 0);
            funct_checker : out std_logic_vector(5 downto 0)
        );
    end component;

    -- Signals
    signal clk                         : STD_LOGIC := '0';
    signal reset                       : STD_LOGIC := '1';
    signal instruction_check           : STD_LOGIC_VECTOR(31 downto 0);
    signal PC_check                    : STD_LOGIC_VECTOR(31 downto 0);
    signal reg1_check, reg2_check,
           reg_write_check             : STD_LOGIC_VECTOR(4 downto 0);
    signal write_data_check            : STD_LOGIC_VECTOR(31 downto 0);
    signal Data_mem_output_check       : STD_LOGIC_VECTOR(31 downto 0);
    signal ALU_out_check               : STD_LOGIC_VECTOR(31 downto 0);
    signal PC_adder_check              : STD_LOGIC_VECTOR(31 downto 0);
    signal jump_address_checker        : STD_LOGIC_VECTOR(31 downto 0);
    signal Branch_add_output_checker  : STD_LOGIC_VECTOR(31 downto 0);
    signal branch_mux_out_checker      : STD_LOGIC_VECTOR(31 downto 0);
    signal PC_mux_out_checker          : STD_LOGIC_VECTOR(31 downto 0);
    signal data_mem_write_checker      : STD_LOGIC_VECTOR(31 downto 0);
    signal sign_extend_input_checker   : STD_LOGIC_VECTOR(15 downto 0);
```

Figure 13: Part one of the CPU Lite Testbench

```
    signal Read_Data2_check            : STD_LOGIC_VECTOR(31 downto 0);
    signal control_instruction_check : std_logic_vector(5 downto 0);
    signal funct_checker : std_logic_vector(5 downto 0);
begin

    -- Instantiate the Unit Under Test (UUT)
    uut: CPU_top
        port map (
            clk                         => clk,
            reset                       => reset,
            instruction_check           => instruction_check,
            PC_check                    => PC_check,
            reg1_check                  => reg1_check,
            reg2_check                  => reg2_check,
            reg_write_check             => reg_write_check,
            write_data_check            => write_data_check,
            Data_mem_output_check       => Data_mem_output_check,
            ALU_out_check               => ALU_out_check,
            PC_adder_check              => PC_adder_check,
            jump_address_checker        => jump_address_checker,
            Branch_add_output_checker  => Branch_add_output_checker,
            branch_mux_out_checker      => branch_mux_out_checker,
            PC_mux_out_checker          => PC_mux_out_checker,
            data_mem_write_checker      => data_mem_write_checker,
            sign_extend_input_checker   => sign_extend_input_checker,
            Read_Data1_check            => Read_Data1_check,
            Read_Data2_check            => Read_Data2_check,
            control_instruction_check   => control_instruction_check,
            funct_checker               => funct_checker
        );

    -- Clock process (100 MHz)
    clk_process :process
    begin
        while now < 200 ns loop
            clk <= '0';
            wait for 5 ns;
            clk <= '1';
            wait for 5 ns;
        end loop;
        wait;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Reset again if needed
        reset <= '1';
        wait for 10 ns;
        reset <= '0';

        wait;
    end process;
end Behavioral;
```

Figure 14: Part two of the CPU Lite Testbench

Now that the testbench has been created, each instruction provided in the control unit is to be tested to see if the functionality of the design works. Provided in Figure 15 are the data memory values used to test the following instructions.

src\Data_Mem_coe_file.txt

```
memory_initialization_radix=16;
memory_initialization_vector=
00000001,
00000002,
00000003,
00000004,
00000005,
00000006,
00000007,
00000008,
00000009,
0000000a,
0000000b;
```

Figure 15: Data Memory values in the .coe file

Testing:

The first instruction tested is LW $1, (2)$2. The hex equivalent for this instruction is 0x8C410002. The expected value loaded into Register 1 is 0x3, as shown in Figure 15. Provided in Figure 16 shows the successful run of this command within one clock cycle. This instruction was the only instruction implemented within the instruction memory .coe file to obtain these results.

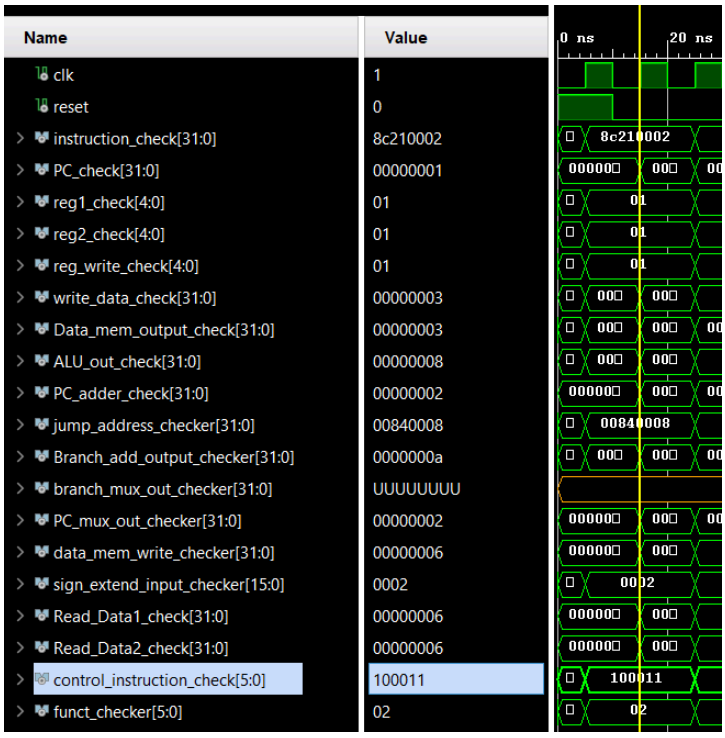

Figure 16: LW $1, (2)$2

The next instruction tested is ADD $1, $3, $4. The hex equivalent for this instruction is 0x00640820. Due to registers 3 and 4 having no data in them, the expected write back number is 0x0. Provided in Figure 17 shows the successful run of this instruction within one clock cycle. This instruction was the only instruction implemented within the instruction memory .coe file to obtain these results.

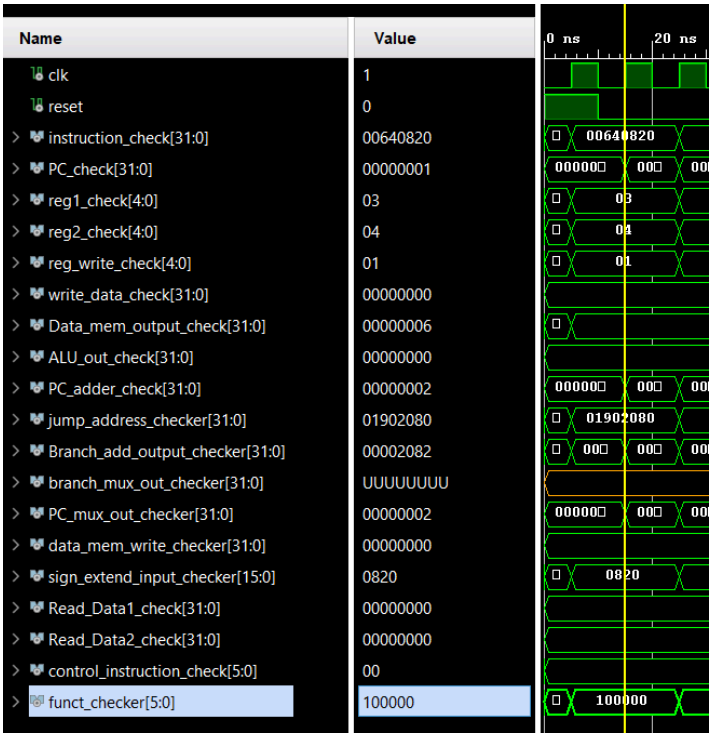

Figure 17: ADD $1, $3, $4

The next instruction tested is SUB $9, $1, $2. The hex equivalent to this instruction is 0x00224822. Due to registers 1 and 2 having no data in them, the expected write back is 0x0. Provided in Figure 18 shows the successful run of this instruction within one clock cycle. This instruction was the only instruction implemented within the instruction memory .coe file to obtain these results.

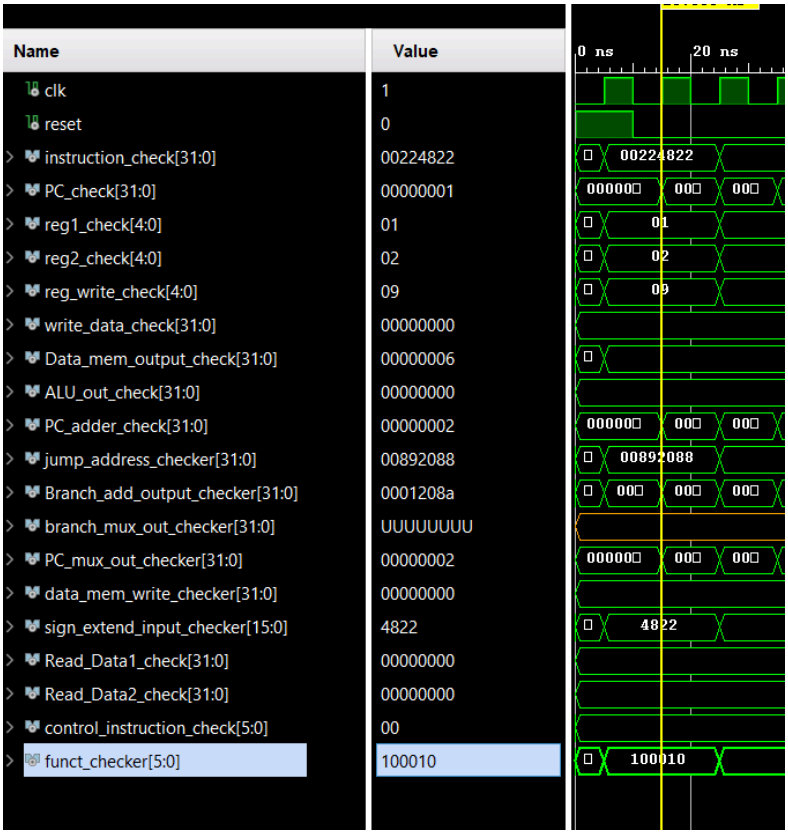

Figure 18: SUB $9, $1, $2

The next instruction tested is BEQ $4, $6, 5. The hex equivalent to this command is 0x10860005. Due to the contents within Registers 6 and 4 being zero, the branch is taken and should jump to PC 0x6, due to each PC update incrementing by 1 rather than 4 for being word accessible rather than Byte accessible. Provided in Figure 19 shows the successful run of this instruction. This instruction was the only instruction implemented within the instruction memory .coe file to obtain these results.

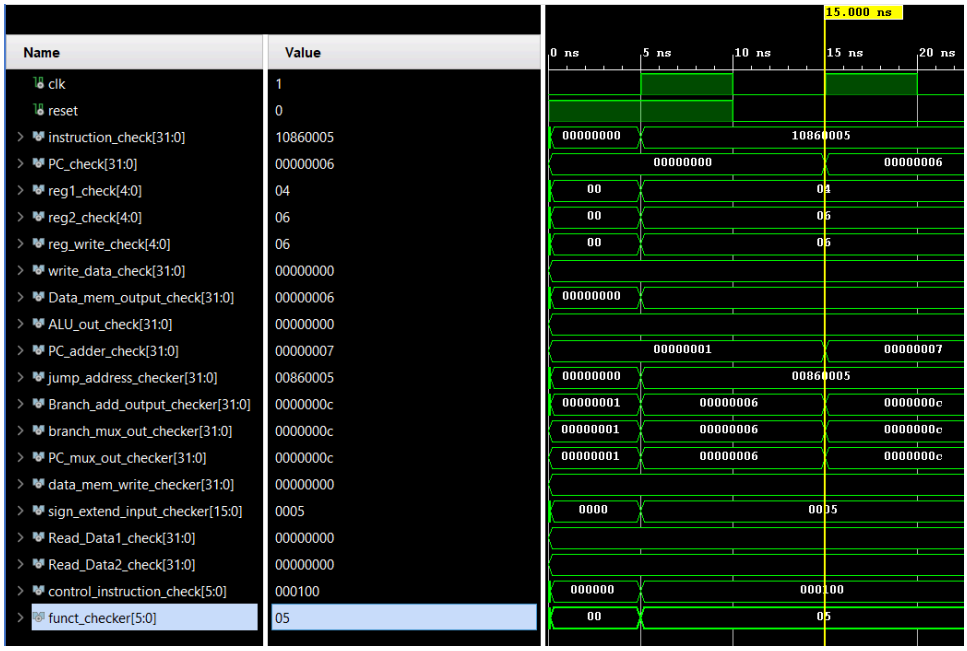

Figure 19: BEQ $4, $6, 5

The final instruction tested is J 0xC. The hex equivalent to this command is 0x08000003. The expected PC output of this command is 0xc due to this being a J-type command. Provided in Figure 20 is the successful run of this command. This instruction was the only instruction implemented within the instruction memory .coe file to obtain these results.

Due to issues discussed in the Problems selection, the SLL command provided in the control unit was unable to be successfully tested.

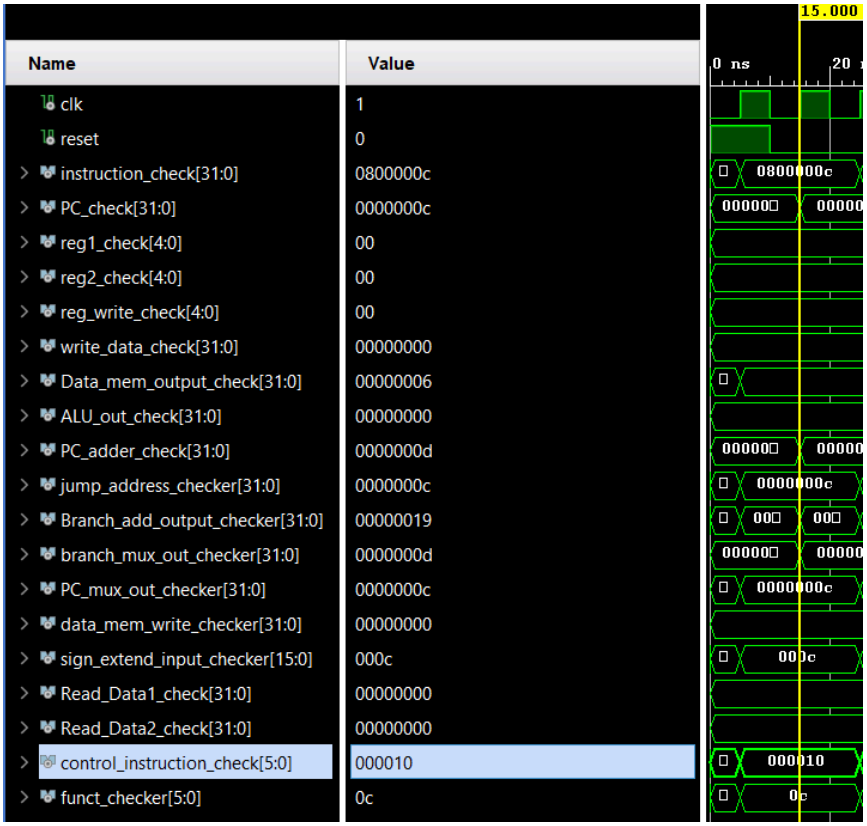

Figure 20: J 0xC

Once each instruction has been tested, running the whole script, provided in Figure 21, is done. Provided in the color green are the various instructions to be run through the CPU Lite Design. Provided in the color red are the expected values, either from the .coe data memory file or the result of the command being run, for each command.

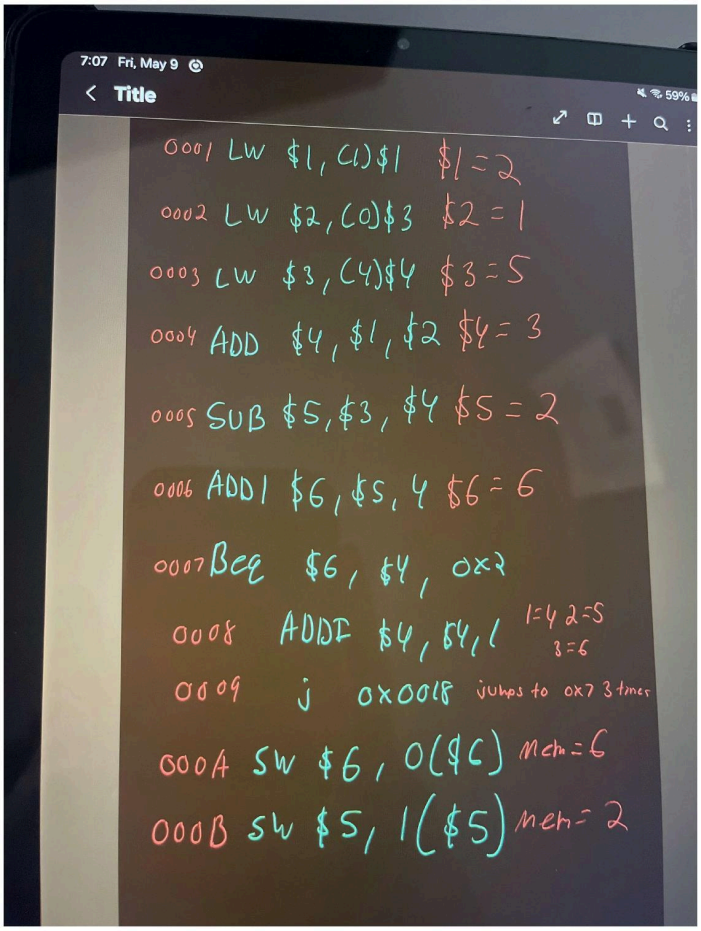

Figure 21: MIPS Script to be ran in CPU Lite Design

Provided in Figure 22 is the Script being run by the CPU Lite design.

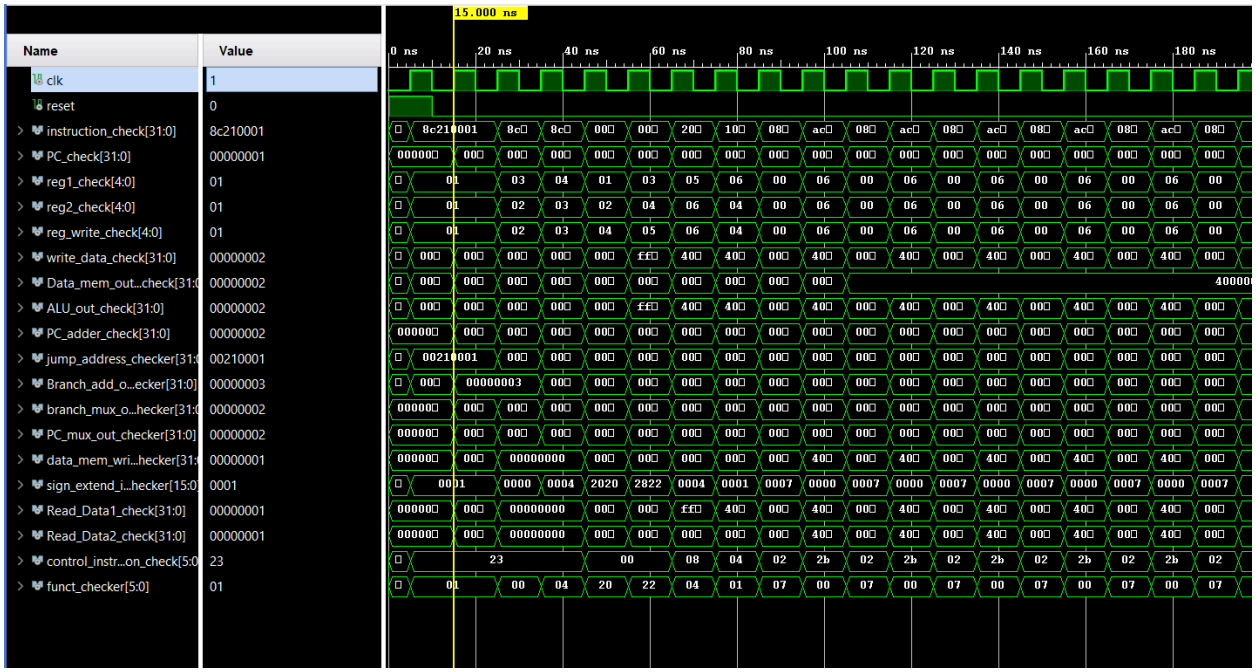

Figure 22: Whole Script run through the CPU Lite Design

Conclusion:

Provided in Figures 2-22 shows a functional Single Cycle CPU Lite Architecture if each instruction is run independently. The PC successfully increments by 1 per each instruction, each command runs in just one clock cycle and all components work together to successfully provide the correct outputs for each instruction, except SLL, provided in the Control Unit. While the design has some shortcomings, each component functions effectively.

Problems:

When running the entire script shown in Figure 21 into the CPU Lite Architecture, the Load Words after LW %1, (1)%1 do not write back the correct content within the data memory .coe file. Though the instructions after the last LW work with the values that are outputted from each Load Word, the Branch and Jump instructions are not able to function properly due to the presence of a negative value within Register 6. This resulted in the BEQ operation to never come true, making it such that the jump statement always jumps to the PC value 0x7 and not completing the entirety of the script. This CPU Lite design has complications running multiple instructions, resulting in the testing of the SLL instruction being extremely difficult. The issue with multiple instructions being run could be due to the clock cycle present within the instruction memory, data memory, and register file all needing to be synchronized so each instruction is passed through in only one clock cycle.