Lab 4 Final Report

1. Team Members

Team member: Dusting Trimmer
Team member: Julion Rowland

2. Implementation Diagram

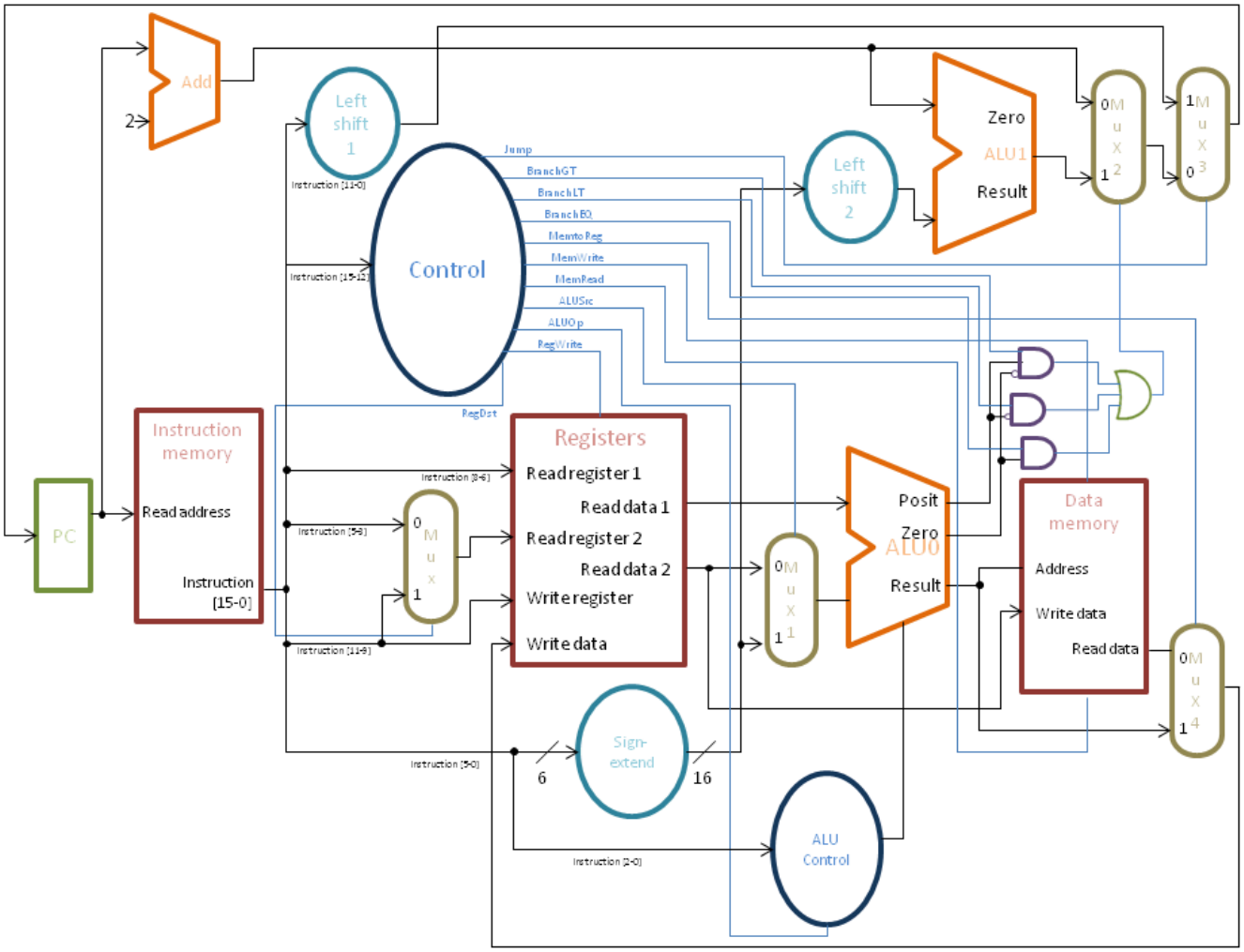


Figure 1 CPU block diagram representing top level architecture

3. Datapath

The CPU has two main datapaths: the path for calculating the result of the instruction operation, and the path for incrementing the program counter. The program counter takes its input and outputs it on a resign edge. This is the component that allows the CPU to iterate through different instructions.

The first step in performing the instruction arithmetic is the instruction memory taking in the output of the program counter. It uses that input as an index for the instruction memory and outputs the proper instruction. The instruction is then split into its opcode, registers, and immediate value. The control unit takes the opcode and outputs the corresponding signals to control the operations of the future components. The register file takes the appropriate register inputs and outputs the values stored in them to the read data outputs. Then the ALU takes either two register inputs for an R-type instruction, or one register and one immediate value for an I-type. The ALU operation is controlled by the ALU control unit, which takes an ALUOp from the control unit and the function bits from the instruction. The ALU output is either used as an address for a data memory operation or as a value to be stored back to a register. The final mux switches between a data memory output and the ALU output to write back to the register file.

The program counter data path is usually quite simple; the current program count gets incremented by two to advance to the next instruction, unless the instruction is a jump or branch.

On a jump instruction, the address stored in the instruction is shifted left once to multiply by two, doubling the range of the jump instruction. The control unit raises the jump signal, which switches a mux to pass the jump address as the next program counter input.

Branches are more complicated because they are conditional. Since the program doesn't always branch on a branch instruction, the ALU needs to do a subtraction operation to identify the appropriate times. On a branch equal, the branch is successful when the ALU zero flag is raised, on a less than, the ALU result needs to be negative, and greater than needs a non-zero value, and a non-negative value. If a branch is successful, a mux will take the output of the second ALU as the input to the program counter. The ALU is fixed to exclusively add, and it adds the current program counter value and the offset stored in the branch instruction.

4. ALU Control

The ALU control took a 4-bit ALU op signal and the function code and output the ALU control signal. The first two values of the ALU op indicate an R-type instruction, then the ALU control unit uses a case statement to pick an ALU operation. The remaining inputs directly correspond to an ALU operation. The values worked out so the output was two less than the input signal.

5. Control Unit

The architecture of the control unit is a case statement. On a change in the opcode input, all of the outputs are reset to zero, and signals are raised based on the opcode value. The ALU op defaults to "1111", which is an unused operation in the ALU control unit, so it will default to an add operation. The first two opcodes, "0000" and "0001", are for R-type instructions, so they raise regDst, regWrite, and memToReg, in addition to their specific ALU op signals. The control unit also specifies all of the signals for controlling our instruction set, not just those for arithmetic. For example, it also signals branches, jumps, and memory access.

6. Simulation Results Memory and Register File at clock cycle 0

You need to print out the data and instructions in your memory and register file throughout the simulation as follows.

6.1. Memory and register file contents at clock cycle 0

Instruction Memory (data + instruction):

Address	Hex Value	Binary Value
0x000	0x6000	0110 0000 0000 0000
0x002	0x2001	0010 0000 0000 0001
0x004	0xE006	1110 0000 0000 0110
0x006	0x6240	0110 0010 0100 0000
0x008	0x2241	0010 0010 0100 0001
0x00A	0xE248	1110 0010 0100 1000
0x00C	0x2241	0010 0010 0100 0001
0x00E	0xE244	1110 0010 0100 0100
0x010	0x6480	0110 0100 1000 0000
0x012	0x248F	0010 0100 1000 1111
0x014	0x66C0	0110 0110 1100 0000
0x016	0x26CF	0010 0110 1100 1111
0x018	0xE6C4	1110 0110 1100 0100
0x01A	0x6900	0110 1001 0000 0000
0x01C	0x6B40	0110 1011 0100 0000
0x01E	0x2B41	0010 1011 0100 0001
0x020	0xEB44	1110 1011 0100 0100
0x022	0x6D80	0110 1101 1000 0000
0x024	0x2D86	0010 1101 1000 0110
0x026	0x3D81	0011 1101 1000 0001
0x028	0x8940	1000 1001 0100 0000

0x02A	0x6FC0	0110 1111 1100 0000
0x02C	0x2FC1	0010 1111 1100 0001
0x02E	0xEFC8	1110 1111 1100 1000
0x030	0xC9CC	1100 1001 1100 1100
0x032	0x4484	0100 0100 1000 0100
0x034	0x06D6	0000 0110 1101 0110
0x036	0x6FC0	0110 1111 1100 0000
0x038	0x2FCF	0010 1111 1100 1111
0x03A	0xEFC4	1110 1111 1100 0100
0x03C	0x7FCF	0111 111 1110 01111
0x03E	0x9F40	1001 1111 0100 0000
0x040	0x2B42	0010 1011 0100 0010
0x042	0x6FC0	0110 1111 1100 0000
0x044	0x2FC1	0010 1111 1100 0001
0x046	0xBDCA	1011 1101 1100 1010
0x048	0xD013	1101 0000 0001 0011
0x04A	0x5008	0101 0000 0000 1000
0x04C	0x0245	0000 0010 0100 0101
0x04E	0x6FC0	0110 1111 1100 0000
0x050	0x2FCF	0010 1111 1100 1111
0x052	0xEFC4	1110 1111 1100 0100
0x054	0x7FCF	0111 1111 1100 1111
0x056	0xEFC8	1110 1111 1100 1000
0x058	0x9F40	1001 1111 0100 0000

0x05A	0xD040	1101 0000 0100 0000

Register file (data):

Register #	Hex Value	Binary Value
0	0x0040	0000 0000 0100 0000
1	0x1010	0001 0000 0001 0000
2	0x000F	0000 0000 0000 1111
3	0x00F0	0000 0000 1111 0000
4	0x0000	0000 0000 0000 0000
5	0x0010	0000 0000 0001 0000
6	0x0006	0000 0000 0000 0110
7	0x0000	0000 0000 0000 0000

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0x0101	0000 0001 0000 0001
0x0012	0x0110	0000 0001 0001 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111

6.2. Memory and register file contents for each loop:

You need to print out the data (not instructions) in your memory and register file after each loop. L, M, N should be the clock cycle numbers after each loop (e.g., L = 55 cycles after the first loop, N = 110 cycles after the second loop, etc). Your program should count the number of clock cycles after executing instructions.

After the first loop, clock cycle: 33 cycles

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0x0110	0000 0001 0001 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111

Register file (data):

Register #	Hex Value	Binary Value
0	0x0008	0000 0000 0000 1000
1	0x1018	0001 0000 0001 1000
2	0x000F	0000 0000 0000 1111
3	0x00F0	0000 0000 1111 0000
4	0x0101	0000 0001 0000 0001
5	0x0010	0000 0000 0001 0000
6	0x0005	0000 0000 0000 0101
7	0xFF00	1111 1111 0000 0000

After the second loop, clock cycle: 53 cycles

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000

0x0018	0x00FF	0000 0000 1111 1111

Register file (data):

Register #	Hex Value	Binary Value
0	0x0001	0000 0000 0000 0001
1	0x1019	0001 0000 0001 1001
2	0x000F	0000 0000 0000 1111
3	0x00F0	0000 0000 1111 0000
4	0x0110	0000 0001 0001 0000
5	0x0012	0000 0000 0001 0010
6	0x0004	0000 0000 0000 0100
7	0xFF00	1111 1111 0000 0000

After the 3rd loop, clock cycle: 72 cycles

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111

Register file (data):

Register #	Hex Value	Binary Value
0	0x0001	0000 0000 0000 0001

1	0x1019	0001 0000 0001 1001
2	0x003C	0000 0000 0011 1010
3	0x00CC	0000 0000 1010 1010
4	0x0011	0000 0000 0001 0001
5	0x0014	0000 0000 0001 0100
6	0x0003	0000 0000 0000 0011
7	0x00FF	0000 0000 1111 1111

After the 4th loop, clock cycle: 90 cycles

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00FF	0000 0000 1111 1111
0x0018	0x00FF	0000 0000 1111 1111

Register file (data):

Register #	Hex Value	Binary Value
0	0x0001	0000 0000 0000 0001
1	0x1019	0001 0000 0001 1001
2	0x00F0	0000 0000 1111 0000
3	0x003C	0000 0000 0011 1010
4	0x00F0	0000 0000 1111 0000
5	0x0016	0000 0000 0001 0110

6	0x0002	0000 0000 0000 0010
7	0x00FF	0000 0000 1111 1111

After the 5th loop, clock cycle: 111cycles

Memory (data + instruction):

Address	Hex Value	Binary Value
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00FF	0000 0000 1111 1111
0x0018	0x00FF	0000 0000 1111 1111

Register file (data):

Register #	Hex Value	Binary Value
0	0x0001	0000 0000 0000 0001
1	0x1019	0001 0000 0001 1001
2	0x03C0	0000 0011 1010 0000
3	0x03FC	0000 0011 1111 1010
4	0x00FF	0000 0000 1111 1111
5	0x0018	0000 0000 0001 1000
6	0x0001	0000 0000 0000 0001
7	0x00FF	0000 0000 1111 1111

7. Integrated output data from the simulation results

Summarize the output **data (not instructions)** for the memory and register file for n loops of the test program.

Memory (data):

Address	Hex value after each loop					
	Initial	1st	2nd	3rd	4th	5th
0x0010	0x0101	0xFF00	0xFF00	0xFF00	0xFF00	0xFF00
0x0012	0x0110	0x0110	0xFF00	0xFF00	0xFF00	0xFF00
0x0014	0x0011	0x0011	0x0011	0x00FF	0x00FF	0x00FF
0x0016	0x00F0	0x00F0	0x00F0	0x00F0	0x00FF	0x00FF
0x0018	0x00FF	0x00FF	0x00FF	0x00FF	0x00FF	0x00FF

Register file (data):

Address	Hex value after each loop					
	Initial	1st	2nd	3rd	4th	5th
0	0x0040	0x0008	0x0001	0x0001	0x0001	0x0001
1	0x1010	0x1018	0x1019	0x1019	0x1019	0x1019
2	0x000F	0x000F	0x000F	0x003C	0x00F0	0x03C0
3	0x00F0	0x00F0	0x00F0	0x00CC	0x003C	0x03FC
4	0x0000	0x0101	0x0110	0x0011	0x00F0	0x00FF
5	0x0010	0x0010	0x0012	0x0014	0x0016	0x0018
6	0x0006	0x0005	0x0004	0x0003	0x0002	0x0001
7	0x0000	0xFF00	0xFF00	0x00FF	0x00FF	0x00FF

8. Bonus Materials

We implemented 7 instructions not found in the standard MIPS. There are two additional R-types, nand and xnor. The other instructions, subi, muli, divi, slli, and srli, don't usually have immediate versions, but were added for use in the test program.

9. Discussion

There was no need to alter the test program significantly. We were able to fit all the necessary instructions in instruction memory, and the branch and jump destinations were close enough to be referenced in the allotted space in each instruction. The only logical change was to increase the value $a1 was initialized to from 5 to 6. This was to keep all five loop iterations working when the decrement was moved in the assembly code.

Each component has been individually testbenched, and the test program runs completely and correctly. As far as we can tell, there are no problems or non-functional aspects.

One major change we made was reordering the rs, rt, and rd registers in the machine code. In traditional MIPS, the last register is the destination register. Ours is reversed, with the registers listed in machine code in the same order as in assembly code. This made more sense for translating and double-checking, but did affect the register file inputs. Instead of ra and rb being the first two registers, and rw being the result of a mux between the second and third registers, rw is the first register, ra is the second, and rb is the result of a mux between the first and third registers. This didn't greatly affect the architecture or functionality of the CPU, but was important to remember when troubleshooting.

10. Final Version of Simulator Code (Attached)

Top Level

```
None
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Top_Level is
    Port (
        clk       : in  STD_LOGIC;
        Reset     : in STD_LOGIC;
        Result    : out unsigned(15 downto 0)
    );
end Top_Level;

architecture Behavioral of Top_Level is
-- signals and components are arranged from left to right and top to bottom
-- using the CPU block diagram
    signal PCIn, PCOut : unsigned(15 downto 0) := X"0000";
    signal adderOut : unsigned(15 downto 0);
    signal instrOut : unsigned(15 downto 0);
    signal mux0Out : unsigned(2 downto 0);
    signal shift0Out : unsigned(15 downto 0) := X"0000";
    signal regDst, ALUSrc, memtoReg, regWrite, memRead, memWrite, brancheq,
branchgt, branchlt, jump : std_logic;
    signal ALUOp : unsigned(3 downto 0);
```

```
signal readData1, readData2 : unsigned(15 downto 0);
signal signExtendOut : unsigned(15 downto 0);
signal mux1Out : unsigned(15 downto 0);
signal ALUControlOut : unsigned(3 downto 0);
signal shift1Out : unsigned(15 downto 0);
signal Zero, Posit : std_logic;
signal ALU0Out : unsigned(15 downto 0);
signal ALU1Out : unsigned(15 downto 0);
signal andOut : std_logic;
signal dataMemOut : unsigned(15 downto 0);
signal mux2Out : unsigned(15 downto 0);
signal mux4Out : unsigned(15 downto 0);-- := RegWr;

component ProgramCounter -- Component: Program Counter
    generic ( N : integer := 16 );
    port (
        clk     : in  std_logic;
        rst     : in  std_logic;
        pc_in   : in  unsigned(N-1 downto 0);
        pc_out  : out unsigned(N-1 downto 0)
    );
end component;

component PCAdder -- Component: Program Adder
        port (
            pc_in  : in  unsigned(15 downto 0);
            pc_out : out unsigned(15 downto 0)
        );
end component;

component InstructionMemory -- Component: Instruction Memory
        port (
            clk   : in  std_logic;
            addr  : in  unsigned(15 downto 0);
            instr : out unsigned(15 downto 0)
        );
end component;

component ControlUnit -- Component: Instruction Memory
    port (
        instr     : in  unsigned(15 downto 12);
        RegDst    : out std_logic;
        ALUSrc    : out std_logic;
```

```
            MemToReg  : out std_logic;
            RegWrite  : out std_logic;
            MemRead   : out std_logic;
            MemWrite  : out std_logic;
            BranchEQ    : out std_logic;
            BranchGT    : out std_logic;
            BranchLT    : out std_logic;
            Jump      : out std_logic;
            ALUOp     : out unsigned(3 downto 0)
        );
    end component;

    component MUX2to1 -- Component: MUX
        generic(N : integer := 16);
        port (
            A   : in  unsigned(N-1 downto 0);
            B   : in  unsigned(N-1 downto 0);
            Sel : in  std_logic;
            Y   : out unsigned(N-1 downto 0)
        );
    end component;

    component Registers -- Component: Registers
        Port (
            clk    : in  STD_LOGIC;
            RegWr  : in  STD_LOGIC;
            Ra, Rb, Rw : in  unsigned(2 downto 0);
            busW   : in  unsigned(15 downto 0);
            busA, busB : out unsigned(15 downto 0)
        );
    end component;

    component SignExtend -- Component: Sign Extender
            port (
                In6  : in  unsigned(5 downto 0);
                Out16 : out unsigned(15 downto 0)
            );
    end component;

    component ALUControl -- Component: Sign Extender
            port (
                ALUOp  : in  unsigned(3 downto 0); ---<<--- might need to be
larger?
```

```
            func   : in unsigned(2 downto 0);
            ALUctr : out unsigned(3 downto 0)
        );
    end component;

    component ShiftLeft -- Component: Left shift by 2
        generic (WIDTH_IN  : integer := 16;
                 WIDTH_OUT : integer := 16;
                 SHIFT_SIZE : integer := 2);
        port (
            data_in  : in  unsigned(WIDTH_IN-1 downto 0);
            data_out : out unsigned(WIDTH_OUT-1 downto 0)
        );
    end component;

    component ALU -- Component: ALU
        generic ( N : integer := 16 );
        Port (
            A, B    : in  unsigned(N-1 downto 0);
            ALUctr  : in  unsigned(3 downto 0);
            Result  : out unsigned(N-1 downto 0);
            Zero, Overflow, Carryout, Posit : out STD_LOGIC
        );
    end component;

    component DataMemory -- Component: Data Memory
        port (
            clk      : in  std_logic;
            MemRead  : in  std_logic;
            MemWrite : in  std_logic;
            Address  : in  unsigned(15 downto 0);
            WriteData: in  unsigned(15 downto 0);
            ReadData : out unsigned(15 downto 0)
        );
    end component;

begin
    PC: ProgramCounter
        port map(
            clk => clk,
            rst => Reset,
            pc_in => PCIn,
            pc_out => PCOut
```

```
    );

PCADDER0 : PCAdder
    port map(
        pc_in => PCOut,
        pc_out => adderOut
    );

INSTRUCTION_MEMORY : InstructionMemory
    port map(
        clk => clk,
        addr => PCOut,
        instr => instrOut
    );

LSHIFT0 : ShiftLeft
    generic map(WIDTH_IN => 12,
             WIDTH_OUT => 12,
             SHIFT_SIZE=> 2)
    port map(
        data_in => instrout(11 downto 0),
        data_out => shift0Out(11 downto 0)
    );

Control : ControlUnit
    port map(
        instr => instrOut(15 downto 12),
        RegDst => regDst,
        ALUSrc => ALUSrc,
        MemToReg => memToReg,
        RegWrite => regWrite,
        MemRead => memRead,
        MemWrite => memWrite,
        BranchEQ => brancheq,
        BranchGT => branchgt,
        BranchLT => branchlt,
        Jump => jump,
        ALUOp => ALUoP
    );

MUX0 : Mux2to1-- MUX0 RESERVED FOR register input
    generic map(N => 3)
    port map(
```

```
        A => instrOut(11 downto 9),
        B => instrOut(5 downto 3),
        Sel => regDst,
        Y => mux0Out
    );

RF: Registers --My Register File
    port map (
        clk   => clk,
        RegWr => regWrite,
        Ra    => instrOut(8 downto 6),
        Rb    => mux0Out,
        Rw    => instrOut(11 downto 9),
        busW  => mux4Out,
        busA  => readData1,
        busB  => readData2
    );

SIGN_XTDR : SignExtend
    port map(
        In6 => instrOut(5 downto 0),
        out16 => signExtendOut
    );

MUX1 : Mux2to1
    port map(
        A => readData2,
        B => signExtendOut,
        Sel => ALUSrc,
        Y => mux1Out
    );

ALUControl0 : ALUControl
    port map(
        ALUOp => ALUOp,
        func => instrOut(2 downto 0),
        ALUctr => ALUControlOut
    );

LSHIFT1 : ShiftLeft
  generic map(WIDTH_IN => 16,
              WIDTH_OUT => 16,
              SHIFT_SIZE=> 1)
```

```
        port map(
            data_in => signExtendOut,
            data_out => shift1Out
        );

    ALU0: ALU --My Alu
        generic map ( N => 16 )
        port map (
            A        => readData1,
            B        => mux1Out,
            ALUctr   => ALUControlOut,
            Result   => ALU0Out,
            Zero     => Zero,
            Posit => Posit,
            Overflow => open,
            Carryout => open
        );

    ALU1: ALU --My Alu
        generic map ( N => 16 )
        port map (
            A        => adderOut,
            B        => shift1Out,
            ALUctr   => "0000",
            Result   => ALU1Out,
            Zero     => open,
            Overflow => open,
            Carryout => open
        );

    andOut <= (brancheq and Zero) or (branchlt and NOT Posit) or (branchgt and
(NOT Zero and Posit));

    DATA_MEMORY : dataMemory
        port map(
            clk => clk,
            memRead => memRead,
            memWrite => memWrite,
            Address => ALU0Out,
            WriteData => readData2,
            ReadData => dataMemOut
        );
```

```
     MUX2 : Mux2to1
        port map(
            A => adderOut,
            B => ALU1Out,
            Sel => andOut,
            Y => mux2Out
        );

    MUX3 : Mux2to1
        port map(
            A => mux2Out,
            B => shift0Out,
            Sel => jump,
            Y => PCIn
        );

    MUX4 : Mux2to1
        port map(
            A => dataMemOut,
            B => ALU0Out,
            Sel => MemtoReg,
            Y => mux4Out
        );

    Result <= mux4Out; --Final Result

end Behavioral;
```

Program Counter

None

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

entity ProgramCounter is
    generic (
        N : integer := 16
    );
    port (
        clk     : in  std_logic;
        rst     : in  std_logic;
```

```
        pc_in   : in  unsigned(N-1 downto 0);
        pc_out  : out unsigned(N-1 downto 0)
    );
end entity ProgramCounter;

architecture Behavioral of ProgramCounter is
    signal pc_reg : unsigned(N-1 downto 0);
    signal count : integer := 0;
begin
    process(clk, rst)
    begin
        if rst = '1' then
            pc_reg <= (others => '0');
        elsif rising_edge(clk) then
            pc_reg <= pc_in;
            count <= count + 1;
        end if;
    end process;

    pc_out <= pc_reg;
end architecture Behavioral;


    process(clk, rst)
    begin
        if rst = '1' then
            pc_reg <= (others => '0');
        elsif rising_edge(clk) then
            pc_reg <= pc_in;
             count <= count + 1;
        end if;
    end process;

    pc_out <= pc_reg;
end architecture Behavioral;
```

Program Adder

None

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

```
entity PCAdder is
    port (
        pc_in  : in  unsigned(15 downto 0);
        pc_out : out unsigned(15 downto 0)
    );
end entity PCAdder;

architecture Behavioral of PCAdder is
begin
    pc_out <= pc_in + 2;
end architecture Behavioral;
```

Instruction Memory

None

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

entity InstructionMemory is
    generic (
        ADDR_WIDTH : integer := 16
    );
    port (
        clk   : in  std_logic;
        addr  : in  unsigned(15 downto 0);
        instr : out unsigned(15 downto 0)
    );
end entity InstructionMemory;

architecture Behavioral of InstructionMemory is
    constant DEPTH : integer := 256;

    type rom_type is array (0 to DEPTH - 1) of unsigned(15 downto 0);

    signal ROM : rom_type := (
        0 => "0110000000000000",
        1 => "0000000000000000",
        2 => "0010000000000001",
        3 => "0000000000000000",
        4 => "1110000000000110",
```

```
5 => "0000000000000000",
6 => "0110001001000000",
7 => "0000000000000000",
8 => "0010001001000001",
9 => "0000000000000000",
10 => "1110001001001000",
11 => "0000000000000000",
12 => "0010001001000001",
13 => "0000000000000000",
14 => "1110001001000100",
15 => "0000000000000000",
16 => "0110010010000000",
17 => "0000000000000000",
18 => "0010010010001111",
19 => "0000000000000000",
20 => "0110011011000000",
21 => "0000000000000000",
22 => "0010011011001111",
23 => "0000000000000000",
24 => "1110011011000100",
25 => "0000000000000000",
26 => "0110100100000000",
27 => "0000000000000000",
28 => "0110101101000000",
29 => "0000000000000000",
30 => "0010101101000001",
31 => "0000000000000000",
32 => "1110101101000100",
33 => "0000000000000000",
34 => "0110110110000000",
35 => "0000000000000000",
36 => "0010110110000110",
37 => "0000000000000000",
38 => "0011110110000001",
39 => "0000000000000000",
40 => "1000100101000000",
41 => "0000000000000000",
42 => "0110111111000000",
43 => "0000000000000000",
44 => "0010111111000001",
45 => "0000000000000000",
46 => "1110111111001000",
47 => "0000000000000000",
```

```
48 => "1100100111001100",
49 => "0000000000000000",
50 => "0100010010000100",
51 => "0000000000000000",
52 => "0000011011010110",
53 => "0000000000000000",
54 => "0110111111000000",
55 => "0000000000000000",
56 => "0010111111001111",
57 => "0000000000000000",
58 => "1110111111000100",
59 => "0000000000000000",
60 => "0111111111001111",
61 => "0000000000000000",
62 => "1001111101000000",
63 => "0000000000000000",
64 => "0010101101000010",
65 => "0000000000000000",
66 => "0110111111000000",
67 => "0000000000000000",
68 => "0010111111000001",
69 => "0000000000000000",
70 => "1011110111001010",
71 => "0000000000000000",
72 => "1101000000010011",
73 => "0000000000000000",
74 => "0101000000001000",
75 => "0000000000000000",
76 => "0000001001000101",
77 => "0000000000000000",
78 => "0110111111000000",
79 => "0000000000000000",
80 => "0010111111001111",
81 => "0000000000000000",
82 => "1110111111000100",
83 => "0000000000000000",
84 => "0111111111001111",
85 => "0000000000000000",
86 => "1110111111001000",
87 => "0000000000000000",
88 => "1001111101000000",
89 => "0000000000000000",
90 => "1101000000100000",
```

```
        others => (others => '0')
    );

begin
    process(addr)
        variable index : integer;
    begin
--        if rising_edge(clk) then
            index := to_integer(addr);
            if to_integer(addr) >= 0 and to_integer(addr) < DEPTH then
                instr <= ROM(to_integer(addr));
            else
                instr <= (others => '0');
            end if;
--        end if;
    end process;

end architecture Behavioral;
```

Left Shift

None

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

entity ShiftLeft is
    generic (
        WIDTH_IN   : integer := 16;
        WIDTH_OUT  : integer := 16;
        SHIFT_SIZE : integer := 1
    );
    port (
        data_in  : in  unsigned(WIDTH_IN-1 downto 0);
        data_out : out unsigned(WIDTH_OUT-1 downto 0)
    );
end entity ShiftLeft;

architecture Behavioral of ShiftLeft is
begin
    data_out <= shift_left(data_in, 1);
end architecture Behavioral;
```

Control Unit

```
None
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ControlUnit is
    port (
        instr     : in  unsigned(3 downto 0);
        RegDst    : out std_logic;
        ALUSrc    : out std_logic;
        MemToReg  : out std_logic;
        RegWrite  : out std_logic;
        MemRead   : out std_logic;
        MemWrite  : out std_logic;
        BranchEQ    : out std_logic;
        BranchGT    : out std_logic;
        BranchLT    : out std_logic;
        Jump      : out std_logic;
        ALUOp     : out unsigned(3 downto 0)
    );
end entity ControlUnit;

architecture Behavioral of ControlUnit is

begin

    process(instr)
    begin
        RegDst   <= '0';
        ALUSrc   <= '0';
        MemToReg <= '0';
        RegWrite <= '0';
        MemRead  <= '0';
        MemWrite <= '0';
        BranchEQ   <= '0';
        BranchGT   <= '0';
        BranchLT   <= '0';
        Jump     <= '0';
        ALUOp    <= "1111"; ---1111 for no alu op

        case instr is
```

```
        -- R-type
        when "0000" =>
            RegDst   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0000";  -- R-type: defer to func bits
        when "0001" =>
            RegDst   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0001";  -- R-type: defer to func bits
        -- I-type arithmetic
        when "0010" =>  -- ADDI
            ALUSrc   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0010";  -- add

        when "0011" =>  -- SUBI
            ALUSrc   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0011";  -- sub
        when "0100" =>  -- MulI
            ALUSrc   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0100";  -- mul

        when "0101" =>  -- DivI
            ALUSrc   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0101";  -- div

        when "0110" =>  -- ANDI
            ALUSrc   <= '1';
            RegWrite <= '1';
            MemToReg <= '1';
            ALUOp    <= "0110";  -- and

        when "0111" =>  -- ORI
            ALUSrc   <= '1';
```

```
        RegWrite <= '1';
        MemToReg <= '1';
        ALUOp    <= "0111";  -- or

    when "1000" =>  -- LW
        ALUSrc   <= '1';
        RegWrite <= '1';
        MemRead  <= '1';
        ALUOp    <= "0010";  -- add for address calc

    when "1001" =>  -- SW
        ALUSrc   <= '1';
        MemWrite <= '1';
        ALUOp    <= "0010";  -- add for address calc

    when "1010" =>  -- BEQ
        BranchEQ   <= '1';
        ALUOp    <= "0011";  -- sub for equality check

    when "1011" =>  -- BEQlt
        BranchLT   <= '1';
        ALUOp    <= "0011";  -- sub for equality check

    when "1100" =>  -- BEQgt
        BranchGT   <= '1';
        ALUOp    <= "0011";  -- sub for equality check

    when "1101" =>  -- JUMP
        Jump <= '1';

    when "1110" =>  -- Shift Left Logical Immediate
        ALUSrc   <= '1';
        RegWrite <= '1';
        MemToReg <= '1';
        ALUOp    <= "1000";

    when "1111" =>  -- Shift Right Logical Immediate
        ALUSrc   <= '1';
        RegWrite <= '1';
        MemToReg <= '1';
        ALUOp    <= "1001";

    when others =>
```

```
                null;
        end case;
    end process;
end architecture Behavioral;
```

Mux

```
None
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MUX2to1 is
    generic (
        N : integer := 16
    );
    port (
        A   : in  unsigned(N-1 downto 0);
        B   : in  unsigned(N-1 downto 0);
        Sel : in  std_logic;
        Y   : out unsigned(N-1 downto 0)
    );
end entity;

architecture Behavioral of MUX2to1 is
begin
    Y <= A when Sel = '0' else B;
end architecture;
```

Register File

```
None
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Registers is
    Port (
        clk    : in  STD_LOGIC;
        RegWr  : in std_logic;
        Ra, Rb, Rw : in  unsigned(2 downto 0);
```

```
        busW   : in  unsigned(15 downto 0);
        busA, busB : out unsigned(15 downto 0)
    );
end Registers;

architecture Behavioral of Registers is
    type reg_array is array (7 downto 0) of unsigned(15 downto 0);
    signal regs : reg_array := (others => (others => '0'));
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            case RegWr is
                when '1' =>
                    regs(to_integer(Rw)) <= busW;
                when others =>
            end case;
        end if;
    end process;
    busA <= regs(to_integer(Ra));
    busB <= regs(to_integer(Rb));
end Behavioral;
```

Sign Extender

None

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

entity SignExtend is
    port (
        In6  : in  unsigned(5 downto 0);
        Out16 : out unsigned(15 downto 0)
    );
end entity;

architecture Behavioral of SignExtend is
begin
    Out16 <= resize(In6, 16);
end architecture Behavioral;
```

ALU Control

None

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

entity ALUControl is
    port (
        ALUOp  : in  unsigned(3 downto 0);
        func   : in  unsigned(2 downto 0);
        ALUctr : out unsigned(3 downto 0)
    );
end entity ALUControl;

architecture Behavioral of ALUControl is
begin
    process(ALUOp, func)
    begin
        case ALUOp is

            -- R-type: decode func bits
            when "0000" =>
                case func is
                    when "000" => ALUctr <= "0000"; -- Add
                    when "001" => ALUctr <= "0001"; -- Sub
                    when "010" => ALUctr <= "1000"; -- Mul
                    when "011" => ALUctr <= "1001"; -- Div
                    when "100" => ALUctr <= "0010"; -- AND
                    when "101" => ALUctr <= "0011"; -- OR
                    when "110" => ALUctr <= "1010"; -- XOR
                    when others => ALUctr <= "0000";
                end case;
                        -- R-type: decode func bits
            when "0001" =>
                case func is
                    when "000" => ALUctr <= "0100"; -- SLL
                    when "001" => ALUctr <= "0101"; -- SRL
                    when "010" => ALUctr <= "0110"; -- SLA
                    when "011" => ALUctr <= "0111"; -- SRA
                    when "100" => ALUctr <= "1011"; -- NOR
                    when "101" => ALUctr <= "1100"; -- NAND
                    when "110" => ALUctr <= "1101"; -- XNOR
                    when "111" => ALUctr <= "1110"; -- HALUT
```

```
                    when others => ALUctr <= "0000";
                end case;
            -- Direct ALU operations (I-type)
            when "0010" => ALUctr <= "0000"; -- Add
            when "0011" => ALUctr <= "0001"; -- Sub
            when "0100" => ALUctr <= "1000"; -- Mul
            when "0101" => ALUctr <= "1001"; -- Div
            when "0110" => ALUctr <= "0010"; -- AND
            when "0111" => ALUctr <= "0011"; -- OR
            when "1000" => ALUctr <= "0100"; -- SLL
            when "1001" => ALUctr <= "0101"; -- SRL
            when others =>
                ALUctr <= "0000";
        end case;
    end process;
end architecture Behavioral;
```

ALU

None

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

entity ALU is
    generic ( N : integer := 16 );
    Port (
        A, B     : in  unsigned(N-1 downto 0);
        ALUctr   : in  unsigned(3 downto 0);
        Result   : out unsigned(N-1 downto 0);
        Zero, Overflow, Carryout, Posit  : out STD_LOGIC
    );
end ALU;

architecture Behavioral of ALU is

    component nBitAdder
        generic ( N : integer := 16 );
        port (
            A, B  : in  unsigned(N-1 downto 0);
            Cin   : in  STD_LOGIC;
            Sum   : out unsigned(N-1 downto 0);
```

```
            Cout  : out STD_LOGIC
        );
    end component;

    -- Internal Signals
    signal add_b, add_sum, alu_result : unsigned(N-1 downto 0);
    signal add_cout, add_cin : STD_LOGIC;
    signal sign_a, sign_b, sign_r : STD_LOGIC;
begin

    -- Select inputs for adder depending on ALUctr
    process(ALUctr, A, B)
    begin
        case ALUctr is
            when "0000" =>  -- ADD
                add_b   <= B;
                add_cin <= '0';
            when "0001" =>  -- SUB
                add_b   <= not B;
                add_cin <= '1';
            when others =>
                add_b   <= (others => '0');
                add_cin <= '0';
        end case;
    end process;

    -- Instantiate n-bit adder
    Adder_Unit: nBitAdder
        generic map (N => N)
        port map (
            A    => A,
            B    => add_b,
            Cin  => add_cin,
            Sum  => add_sum,
            Cout => add_cout
        );

    -- Main ALU Logic
    process(ALUctr, A, B, add_sum, add_cout)
        variable temp_product : unsigned((2*N)-1 downto 0);
        variable dividend   : unsigned(N-1 downto 0);
        variable divisor    : unsigned(N-1 downto 0);
        variable quotient   : unsigned(N-1 downto 0);
```

```
    variable remainder  : unsigned(N downto 0);
    variable temp       : unsigned(N downto 0);
    variable i : integer;
begin
    alu_result <= (others => '0');
    Carryout   <= '0';
    Overflow   <= '0';

    case ALUctr is
        when "0000" =>  -- ADD
            alu_result <= add_sum;
            Carryout   <= add_cout;

        when "0001" =>  -- SUB
            alu_result <= add_sum;
            Carryout   <= not add_cout;

        when "0010" =>  -- AND
            alu_result <= A and B;
            Carryout   <= '0';

        when "0011" =>  -- OR
            alu_result <= A or B;
            Carryout   <= '0';

        when "0100" =>  -- Logical Left Shift
            alu_result <= shift_left(A,to_integer(unsigned(B)));
            Carryout   <= '0';

        when "0101" =>  -- Logical Right Shift
            alu_result <= shift_right(A, to_integer(unsigned(B)));
            Carryout   <= '0';

        when "0110" =>  -- Arithmetic Left Shift
            alu_result <= shift_left(A, 1);
            Carryout   <= '0';

        when "0111" =>  -- Arithmetic Right Shift
            alu_result <= shift_right(A, 1);
            alu_result(N-1) <= A(N-1);  -- preserve sign
            Carryout   <= '0';

        when "1000" =>  -- MULTIPLICATION
```

```
                temp_product := (others => '0');
                temp_product(N-1 downto 0) := B;     -- lower bits = multiplier

                for i in 0 to N-1 loop
                    if temp_product(0) = '1' then
                        temp_product((2*N)-1 downto N) :=
                            temp_product((2*N)-1 downto N) + A;
                    end if;
                    -- logical right shift
                    temp_product := '0' & temp_product((2*N)-1 downto 1);
                end loop;

                alu_result <= temp_product(N-1 downto 0);
                Carryout   <= temp_product(N);  -- upper bit carry indicator

            when "1001" =>  -- DIVISION
                if B = 0 then
                    alu_result <= (others => '0');  -- divide by zero
                    Carryout   <= '0';
                else
                    dividend  := A;
                    divisor   := B;
                    remainder := (others => '0');
                    quotient  := (others => '0');

                    for i in N-1 downto 0 loop
                        -- Shift left remainder and bring down next dividend
bit
                        remainder := shift_left(remainder, 1);
                        remainder(0) := dividend(i);

                        -- Try subtraction
                        temp := remainder - ("0" & divisor);  -- extend divisor
by 1 bit

                        if temp(N) = '0' then  -- no borrow => remainder >=
divisor
                            remainder := temp;
                            quotient(i) := '1';
                        else
                            null;
                        end if;
                    end loop;
```

```
                alu_result <= quotient;  -- output quotient
                Carryout   <= '0';
                Overflow   <= '0';
            end if;
        when "1010" =>  -- XOR
            alu_result <= A xor B;
            Carryout   <= '0';
        when "1011" =>  -- NOR
            alu_result <= A xor B;
            Carryout   <= '0';
        when "1100" =>  -- NAND
            alu_result <= A xor B;
            Carryout   <= '0';
        when "1101" =>  -- XNOR
            alu_result <= A xor B;
            Carryout   <= '0';
        when "1110" =>  -- HAULT
            alu_result <= A xor B;
            Carryout   <= '0';
        when others =>
            alu_result <= (others => '0');
            Carryout   <= '0';
    end case;
end process;

-- Overflow Detection
sign_a <= A(N-1);
sign_b <= B(N-1);
sign_r <= alu_result(N-1);

process(ALUctr, sign_a, sign_b, sign_r)
begin
    if ALUctr = "0000" then  -- ADD
        Overflow <= (sign_a and sign_b and not sign_r) or
                    (not sign_a and not sign_b and sign_r);
    elsif ALUctr = "0001" then  -- SUB
        Overflow <= (sign_a and not sign_b and not sign_r) or
                    (not sign_a and sign_b and sign_r);
    elsif (ALUctr = "1001" and B = 0) then
        Overflow <= '1';
    else
        Overflow <= '0';
    end if;
```

```
    end process;

    -- Zero flag
    process(alu_result)
    begin
        if alu_result = (B"0000_0000_0000_0000_0000_0000_0000_0000") then
            Zero <= '1';
        else
            Zero <= '0';
        end if;
    end process;

    -- Positive flag
    process(ALUctr, alu_result)
    begin
        if ALUctr = "0001" then  -- SUB
            Posit <= alu_result(N-1);  -- Sign bit of subtraction result
        else
            Posit <= '0';
        end if;
    end process;
    Result <= alu_result;

end Behavioral;
```

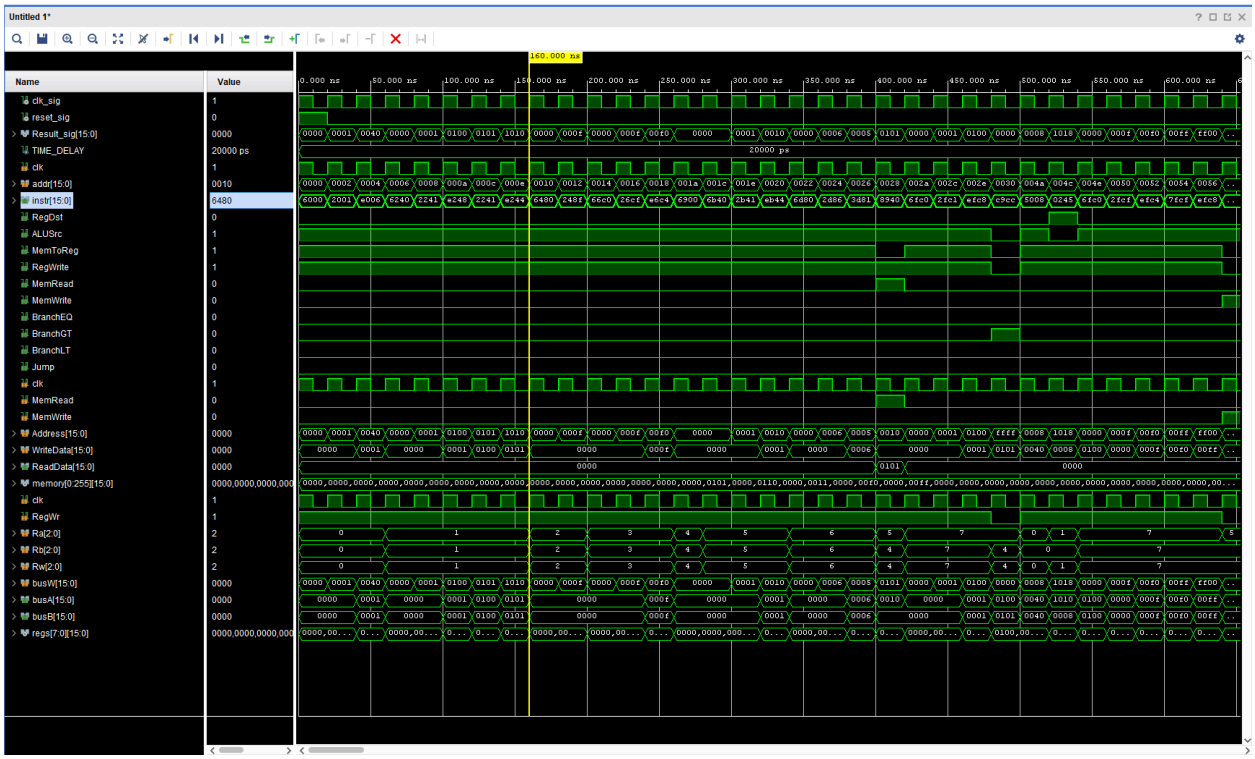
Data Memory

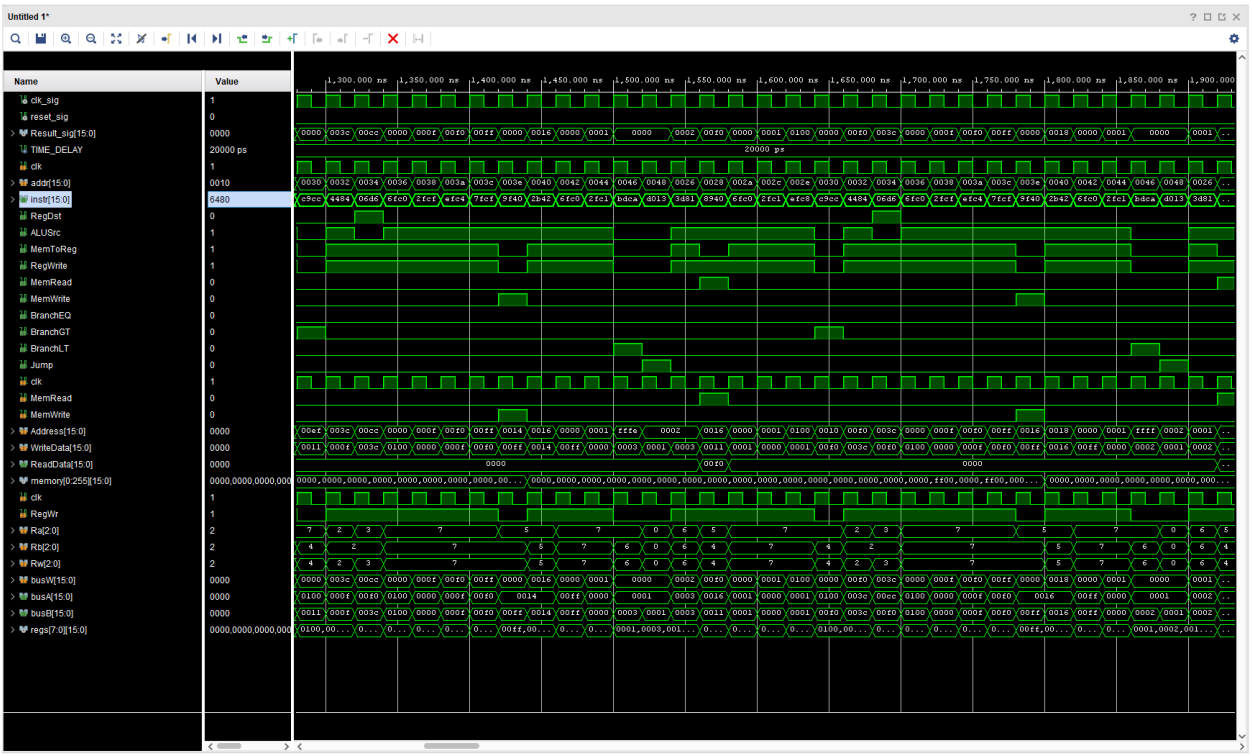
None

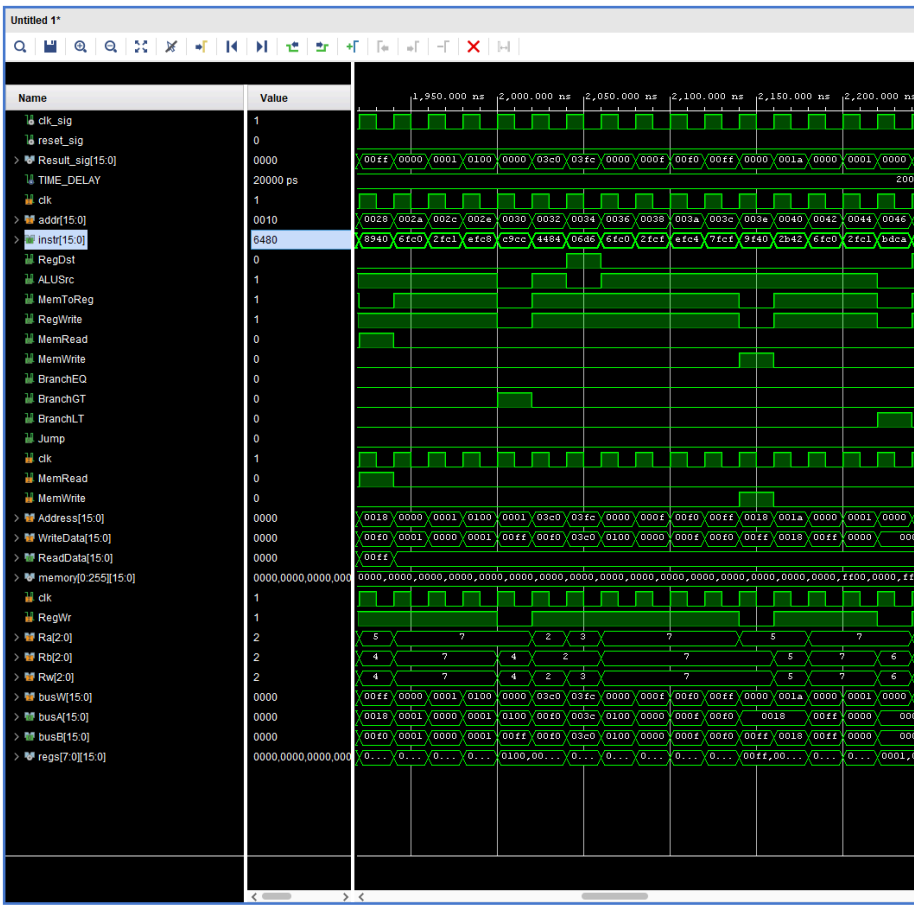
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DataMemory is
    generic (
        MEM_SIZE : integer := 256
    );
    port (
        clk      : in  std_logic;
        MemRead  : in  std_logic;
        MemWrite : in  std_logic;
        Address  : in  unsigned(15 downto 0);
        WriteData: in  unsigned(15 downto 0);
```

```
        ReadData : out unsigned(15 downto 0)
    );
end entity;

architecture Behavioral of DataMemory is
    type mem_array is array (0 to MEM_SIZE-1) of unsigned(15 downto 0);
    signal memory : mem_array := (
        16 => "0000000100000001",
        18 => "0000000100010000",
        20 => "0000000000010001",
        22 => "0000000011110000",
        24 => "0000000011111111",
        others => (others => '0')
        );
    signal addr_index : integer range 0 to MEM_SIZE-1;
begin
    addr_index <= to_integer(unsigned(Address(7 downto 0)));

    process(clk)
    begin
        if rising_edge(clk) then
            if MemWrite = '1' then
                memory(addr_index) <= WriteData;
            end if;
        end if;
    end process;

    ReadData <= memory(addr_index) when MemRead = '1' else (others => '0');
end architecture;
```

11. Final Testbench




Untitled 1*
Name
Value
clk_sig
reset_sig
Result_sig[15:0]
TIME_DELAY
clk
addr[15:0]
instr[15:0]
RegDst
ALUSrc
MemToReg
RegWrite
MemRead
MemWrite
BranchEQ
BranchGT
BranchLT
Jump
clk
MemRead
MemWrite
Address[15:0]
WriteData[15:0]
ReadData[15:0]
memory[0:255][15:0]
clk
RegWr
Ra[2:0]
Rb[2:0]
Rw[2:0]
busW[15:0]
busA[15:0]
busB[15:0]
regs[7:0][15:0]


Untitled 1*
Name
Value
clk_sig
reset_sig
Result_sig[15:0]
TIME_DELAY
clk
addr[15:0]
instr[15:0]
RegDst
ALUSrc
MemToReg
RegWrite
MemRead
MemWrite
BranchEQ
BranchGT
BranchLT
Jump
clk
MemRead
MemWrite
Address[15:0]
WriteData[15:0]
ReadData[15:0]
memory[0:255][15:0]
clk
RegWr
Ra[2:0]
Rb[2:0]
Rw[2:0]
busW[15:0]
busA[15:0]
busB[15:0]
regs[7:0][15:0]