Michelle Yang
A20298610

ECE 485
Final project

**Abstract**
The objective of this project is to build a multicycle processor that can support a subset of the MIPS instruction set. This is intended as an educational exercise to understand the benefits of the RISC architecture over the CISC architecture in terms of consistency and optimization.

**Introduction**
The MIPS architecture is an example of a Reduced Instruction Set Computer, or RISC architecture. The primary advantage of RISC over CISC is instruction standardization. In MIPS, all instructions are 32-bits long, which allows for fixed instruction processing time and shared hardware.

Two processor designs exist for MIPS:
- Single cycle: each instruction takes a single clock cycle to complete
- Multicycle: each instruction takes several cycles to complete

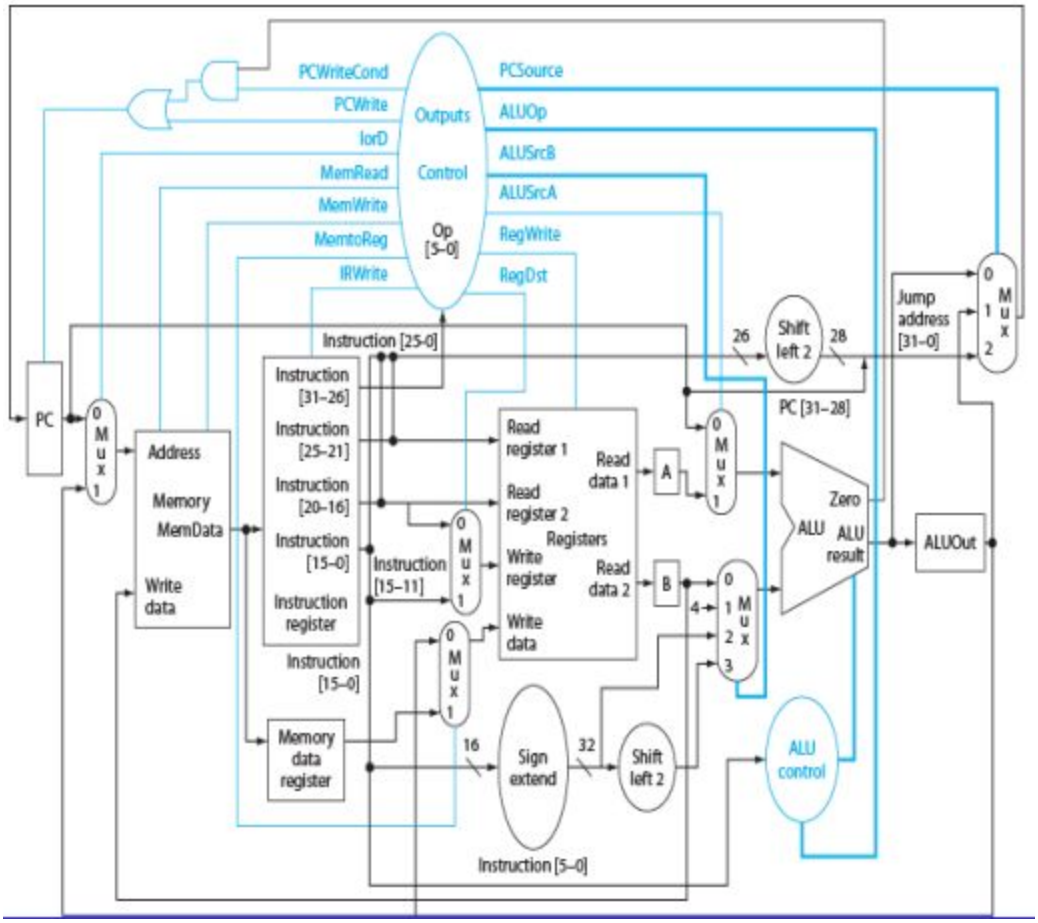
The first is easier to control but requires more hardware. The second one requires more complex control but allows shared hardware. However, the multicycle design does not allow pipelining. Pipelining is outside the scope of this project.

**Design**
The objective of this project is to build a multicycle MIPS processor that can support the following subset of instructions:

| Op Code [31:26] | Function [5:0] | Instruction | Operation | Machine code |
|---|---|---|---|---|
| 100011 | - | lw | lw $t3, 300($s2) | 8E4B 012C |
| 101011 | - | sw | sw $t6, 400($s7 | AEEE 0190 |
| 000000 | 100000 | add | add $t5, $t3, $s1 | 0171 6820 |
| 000100 | - | bne | beq $s6, $t5, 200 | 11B6 00C8 |
| (not actual MIPS) 000000 | - | subi | subi $t1,$t2, 100 | 0149 0064 |
| 001101 | - | ori | ori $t1, $t2, 100 | 3549 0064 |

Note that subi in the above table is not an actual MIPS instruction. As the control path is not implemented, any opcode will be valid, thus 000000 is chosen as an arbitrary opcode.

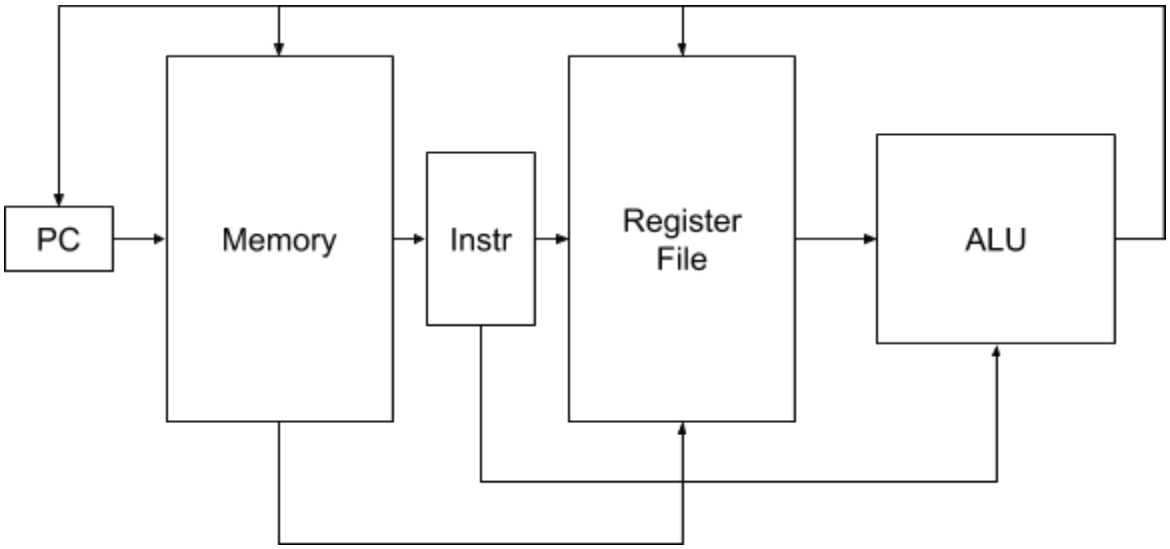
The design follows the multicycle pattern given by Patterson & Hennessy:




In a multicycle datapath, components can be reused. Thus the following cycles are required:

- Instruction fetch (IF):
  - Memory is accessed at location PC and value is brought to instruction register
  - PC is incremented to next instruction
- Instruction decode/register read (ID/RR):
  - Instruction is split into pieces and sent to the control unit for processing
  - Register values are read from register file
  - Branch target address is calculated
- Execution (EX):
  - ALU calculation
- Memory access (MEM):
  - Data is read from or written to memory
- Writeback (WB):
  - ALU calculated output or memory value is written to the register file

On a high level, the processor looks like this:




Discrete blocks were built for the following:

- MUX
- Adder
- 1-bit ALU
- ALU
- Memory
- Register
- Register file
- Shift
- Sign extend

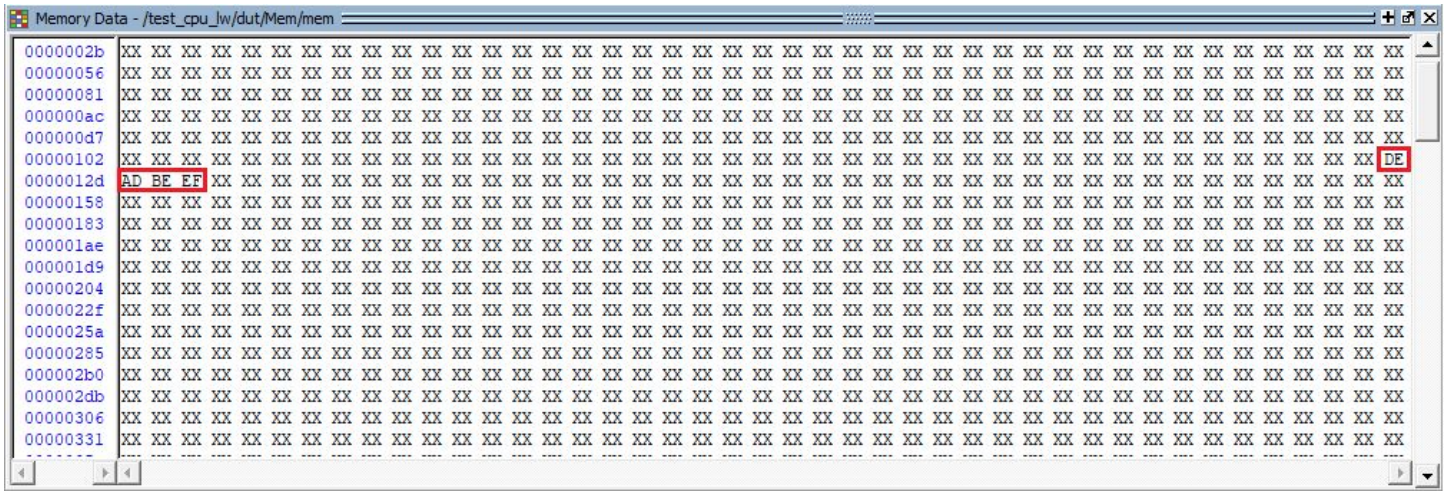
Each of the pieces had an associated testbench to verify the correctness. The structural VHDL approach was favored in putting the pieces together into the complete CPU. Component code can be seen in appendix A, testbenches can be seen in appendix B.

**Test procedure**

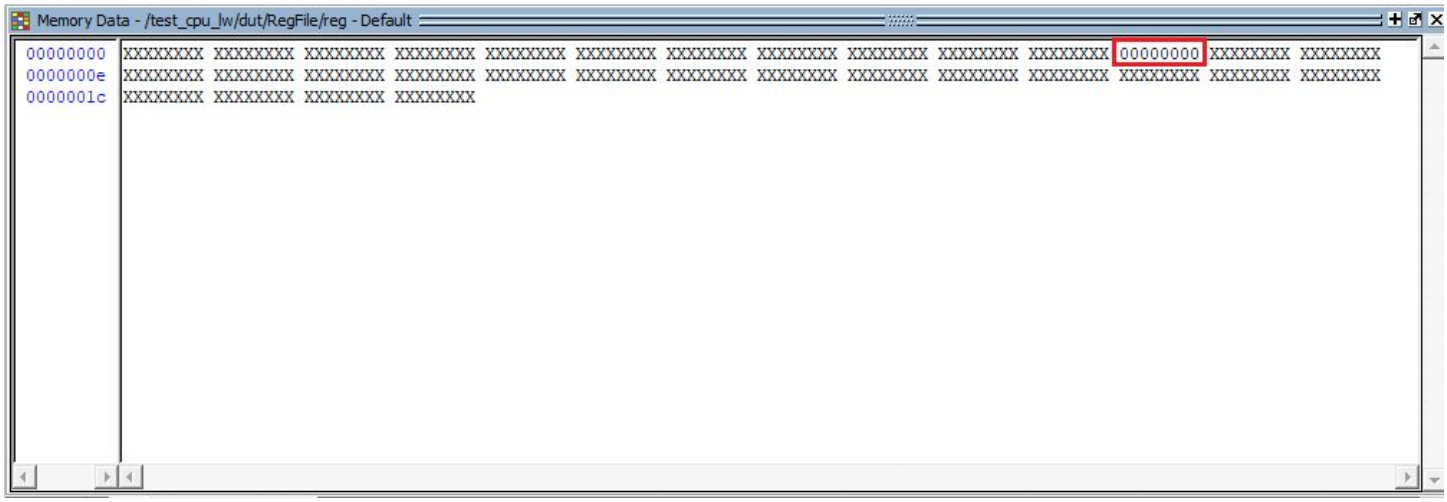
A test bench was written for each instruction that were run independently. Test procedure was as follows:

1. Compile all files by clicking “Compile All” in ModelSim.
2. Select “Simulation > Start Simulation” from the top menu bar and select the relevant simulation. This will launch the simulation view.
3. Go to “Memory List” tab on the left panel and double click “test_cpu\dut\mem” to launch the editor. Write the instruction and relevant data into the correct memory location.
4. Double click “test_cpu\dut\reg” on the Memory List tab to launch the editor. Write the relevant register data into the correct location.
5. Add waveforms and run the testbench.
6. If nothing is displayed in the output panel on the bottom of the window, the test has passed.

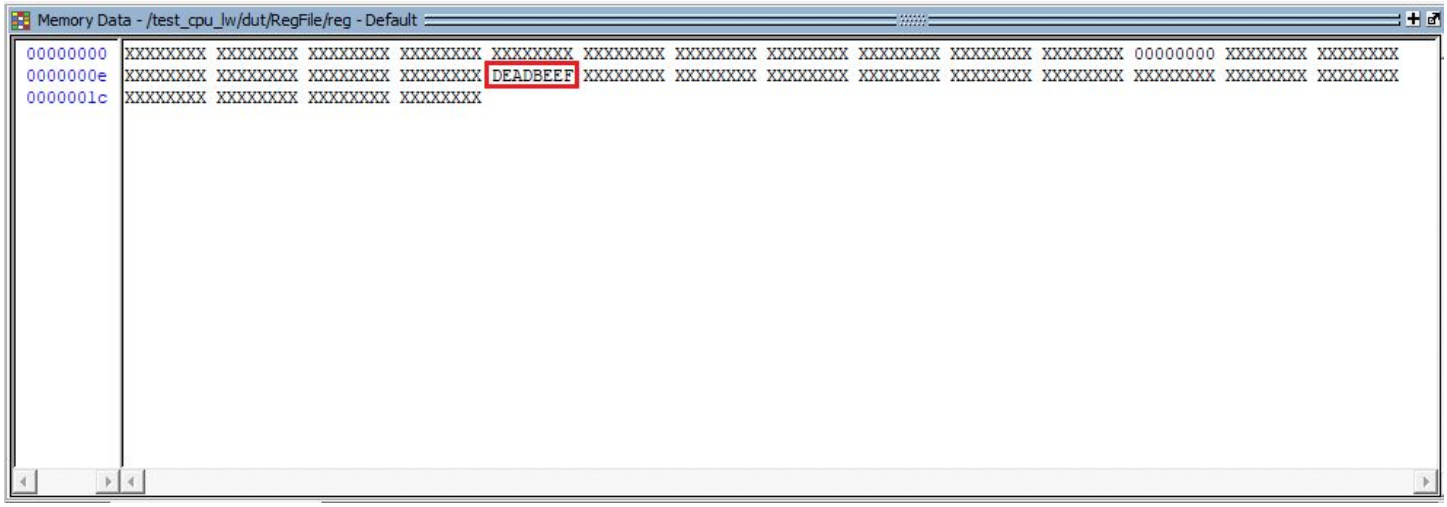
For example, for the lw $t3, 300($s2) instruction, memory is modified using the memory pane:




Registers are modified similarly:




The final result:



As the correct memory value was loaded from memory to the register file, this test was successful.
All test benches were ran and passed successfully in a similar manner.

**Discussion**

Some of the improvements that could be made are the following:

- The ALU is built from 32 1-bit ALUs, which means when an add (or subtract) operation is performer, it acts as a ripple carry adder. In order to speed up the time required for addition, a carry lookahead adder could be implemented. The downside of this approach is additional complexity caused by a lot of extra hardware.
- Certain registers should not be modified, for example $zero. Additional checks can be put in place to enforce this rule.
- The memory block was restricted to a size of roughly 4KB. It could be expanded to about 2GB by changing the array size, but cannot reach the full 4GB address space by doing so. The problem lies in VHDL's representation of an integer (which must be used for array indices): it is always a signed integer, and therefore only goes up to a maximum of 2147483648. In order to access the full 32-bit address space, a second array will need to be defined. Each array can be indexed by the bottom 31 bits, and a mux used to choose between the two based on the most significant bit.

**Conclusion**

In conclusion, the objective of the project was met. A MIPS processor was designed and tested using ModelSim, during the process learning more about the architecture and design.

**Appendix A: component code**

2-input MUX:

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

entity mux2 is
   generic(
        K           : integer  := 32
   );
   port(
        a, b        : in STD_LOGIC_VECTOR(K-1 downto 0);
        sel         : in STD_LOGIC;
        output      : out STD_LOGIC_VECTOR(K-1 downto 0)
   );
end;

architecture behavior of mux2 is
begin
   output <=   a when sel = '0' else b;
```

```
end;
```

4-input MUX:

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

entity mux4 is
   generic(
       K           : integer  := 32
   );
   port(
       a, b, c, d  : in STD_LOGIC_VECTOR(K-1 downto 0);
       sel         : in STD_LOGIC_VECTOR(1 downto 0);
       output      : out STD_LOGIC_VECTOR(K-1 downto 0)
   );
end;

architecture struct of mux4 is
   component mux2 is
       generic(
           K: integer  := 32
       );
       port(
           a, b        : in STD_LOGIC_VECTOR(K-1 downto 0);
           sel         : in STD_LOGIC;
           output      : out STD_LOGIC_VECTOR(K-1 downto 0)
       );
   end component;

   signal tmp_1, tmp_2 : STD_LOGIC_VECTOR(K-1 downto 0);
begin
   mux_1: mux2 port map(a, b, sel(0), tmp_1);
   mux_2: mux2 port map(c, d, sel(0), tmp_2);
   mux_3: mux2 port map(tmp_1, tmp_2, sel(1), output);
end;
```

Adder:

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

-- half adder
```

```
entity half_adder is
   port(
       a, b        : in STD_LOGIC;
       sum, carry  : out STD_LOGIC
   );
end;

architecture behavior of half_adder is
begin
   sum     <= a XOR b after 1 ns;
   carry   <= a AND b after 1 ns;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- full adder
entity adder is
   port(
       a, b, cin   : in STD_LOGIC;
       sum, carry  : out STD_LOGIC
   );
end;

architecture struct of adder is
   component half_adder
       port(
           a, b        : in STD_LOGIC;
           sum, carry  : out STD_LOGIC
       );
   end component;
   signal sum_part, carry_1, carry_2
                   : STD_LOGIC;
begin
   half_1: half_adder port map(a, b, sum_part, carry_1);
   half_2: half_adder port map(sum_part, cin, sum, carry_2);
   carry   <= carry_1 OR carry_2;
end;
```

1-bit ALU:

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_MISC.all;

-- one bit ALU
-- supports add, neg, or
-- op, neg, and Cin are control signals
entity alu1 is
   port(
       a, b, op, neg, Cin      : in STD_LOGIC;
       output, Cout            : out STD_LOGIC
   );
end;

architecture behavior of alu1 is
   component adder
       port(
           a, b, cin       : in STD_LOGIC;
           sum, carry      : out STD_LOGIC
       );
   end component;
   signal not_b, input_b, out_or, out_add
                           : STD_LOGIC;
begin
   not_b       <= NOT b;
   input_b     <= fun_MUX2x1(b, not_b, neg);
   out_or      <= a OR b;
   add: adder port map(a, input_b, Cin, out_add, Cout);
   output      <= fun_MUX2x1(out_or, out_add, op);
end;
```

Full-ALU:

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

-- full 32-bit ALU
-- note this uses ripple carry adder
-- op is the control signal
-- upper bit represents op, lower represents neg
entity alu is
```

```
    port(
        a, b        : in STD_LOGIC_VECTOR(31 downto 0);
        ALUOp       : in STD_LOGIC_VECTOR(1 downto 0);
        output      : out STD_LOGIC_VECTOR(31 downto 0);
        zero        : out STD_LOGIC
    );
end;

-- make a 32-bit ALU by stacking 32 1-bit ALUs
architecture struct of alu is
    component alu1
        port(
            a, b, op, neg, Cin  : in STD_LOGIC;
            output, Cout        : out STD_LOGIC
        );
    end component;
    signal carry, alu_out   : STD_LOGIC_VECTOR(31 downto 0);
    signal op, neg          : STD_LOGIC;
begin
    op  <= ALUOp(1);
    neg <= ALUOp(0);

    alu_0:  alu1 port map(a(0), b(0), op, neg, neg, alu_out(0), carry(0));
    alu_1:  alu1 port map(a(1), b(1), op, neg, carry(0), alu_out(1), carry(1));
    alu_2:  alu1 port map(a(2), b(2), op, neg, carry(1), alu_out(2), carry(2));
    alu_3:  alu1 port map(a(3), b(3), op, neg, carry(2), alu_out(3), carry(3));
    alu_4:  alu1 port map(a(4), b(4), op, neg, carry(3), alu_out(4), carry(4));
    alu_5:  alu1 port map(a(5), b(5), op, neg, carry(4), alu_out(5), carry(5));
    alu_6:  alu1 port map(a(6), b(6), op, neg, carry(5), alu_out(6), carry(6));
    alu_7:  alu1 port map(a(7), b(7), op, neg, carry(6), alu_out(7), carry(7));
    alu_8:  alu1 port map(a(8), b(8), op, neg, carry(7), alu_out(8), carry(8));
    alu_9:  alu1 port map(a(9), b(9), op, neg, carry(8), alu_out(9), carry(9));
    alu_10: alu1 port map(a(10), b(10), op, neg, carry(9), alu_out(10), carry(10));
    alu_11: alu1 port map(a(11), b(11), op, neg, carry(10), alu_out(11), carry(11));
    alu_12: alu1 port map(a(12), b(12), op, neg, carry(11), alu_out(12), carry(12));
    alu_13: alu1 port map(a(13), b(13), op, neg, carry(12), alu_out(13), carry(13));
    alu_14: alu1 port map(a(14), b(14), op, neg, carry(13), alu_out(14), carry(14));
    alu_15: alu1 port map(a(15), b(15), op, neg, carry(14), alu_out(15), carry(15));
    alu_16: alu1 port map(a(16), b(16), op, neg, carry(15), alu_out(16), carry(16));
    alu_17: alu1 port map(a(17), b(17), op, neg, carry(16), alu_out(17), carry(17));
    alu_18: alu1 port map(a(18), b(18), op, neg, carry(17), alu_out(18), carry(18));
```

```
    alu_19: alu1 port map(a(19), b(19), op, neg, carry(18), alu_out(19), carry(19));
    alu_20: alu1 port map(a(20), b(20), op, neg, carry(19), alu_out(20), carry(20));
    alu_21: alu1 port map(a(21), b(21), op, neg, carry(20), alu_out(21), carry(21));
    alu_22: alu1 port map(a(22), b(22), op, neg, carry(21), alu_out(22), carry(22));
    alu_23: alu1 port map(a(23), b(23), op, neg, carry(22), alu_out(23), carry(23));
    alu_24: alu1 port map(a(24), b(24), op, neg, carry(23), alu_out(24), carry(24));
    alu_25: alu1 port map(a(25), b(25), op, neg, carry(24), alu_out(25), carry(25));
    alu_26: alu1 port map(a(26), b(26), op, neg, carry(25), alu_out(26), carry(26));
    alu_27: alu1 port map(a(27), b(27), op, neg, carry(26), alu_out(27), carry(27));
    alu_28: alu1 port map(a(28), b(28), op, neg, carry(27), alu_out(28), carry(28));
    alu_29: alu1 port map(a(29), b(29), op, neg, carry(28), alu_out(29), carry(29));
    alu_30: alu1 port map(a(30), b(30), op, neg, carry(29), alu_out(30), carry(30));
    alu_31: alu1 port map(a(31), b(31), op, neg, carry(30), alu_out(31), carry(31));

    output <= alu_out;
    zero <= nor_reduce(alu_out);
end;
```

Memory:

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

-- memory
entity memory is
    port(
        MemRead, MemWrite   : in STD_LOGIC;
        Address, WriteData  : in STD_LOGIC_VECTOR(31 downto 0);
        ReadData            : out STD_LOGIC_VECTOR(31 downto 0)
    );
end;

architecture behavior of memory is
    -- size of memory may vary
    -- here it is assumed to be 4K
    -- memory is byte addressible
    -- assume memory accesses are word size
    -- vhdl integer size only goes up to 2**31 - 1
    -- if full address space is required, need 2
    -- arrays and mux them together
    type t_Memory is array (0 to 4095) of STD_LOGIC_VECTOR(7 downto 0);
```

```
    signal mem      : t_Memory;
    signal addr     : integer;
begin
    process (MemWrite, WriteData, Address)
        variable addr: natural range 0 to 4095;
    begin(Benenson, Fraichard, & Parent, 2008)
        -- address needs to be truncated
        -- if higher address range is required, second
        -- array is needed and muxed based on first bit
        addr := to_integer(unsigned(Address));

        if (MemWrite = '1') then
            mem(addr)       <= WriteData(31 downto 24) after 1 ns;
            mem(addr + 1)   <= WriteData(23 downto 16) after 1 ns;
            mem(addr + 2)   <= WriteData(15 downto 8)  after 1 ns;
            mem(addr + 3)   <= WriteData(7 downto 0)   after 1 ns;
        end if;

        ReadData(31 downto 24)  <= mem(addr)     after 1 ns;
        ReadData(23 downto 16)  <= mem(addr + 1) after 1 ns;
        ReadData(15 downto 8)   <= mem(addr + 2) after 1 ns;
        ReadData(7 downto 0)    <= mem(addr + 3) after 1 ns;
    end process;
end;
```

Register file:

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

entity register_file is
    port(
        ReadReg_A, ReadReg_B, WriteReg  : in STD_LOGIC_VECTOR(4 downto 0);
        WriteData                       : in STD_LOGIC_VECTOR(31 downto 0);
        RegWrite                        : in STD_LOGIC;
        ReadData_A, ReadData_B          : out STD_LOGIC_VECTOR(31 downto 0)
    );
end;

architecture behavior of register_file is
    type t_RegFile  is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
```

```
    signal reg      : t_RegFile;
    signal addr     : integer;
begin
    process (ReadReg_A, ReadReg_B, WriteReg, WriteData)
        variable ReadAddr_A, ReadAddr_B, WriteAddr : natural range 0 to 31;
    begin
        ReadAddr_A := to_integer(unsigned(ReadReg_A));
        ReadAddr_B := to_integer(unsigned(ReadReg_B));
        WriteAddr  := to_integer(unsigned(WriteReg));

        if (RegWrite = '1') then
            reg(WriteAddr) <= WriteData;
        end if;

        ReadData_A  <= reg(ReadAddr_A);
        ReadData_B  <= reg(ReadAddr_B);
    end process;
end;
```

Sign extender:

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

entity sign_extend is
    port(
        input       : in STD_LOGIC_VECTOR(15 downto 0);
        output      : out STD_LOGIC_VECTOR(31 downto 0)
    );
end;

architecture behavior of sign_extend is
begin
    output <= std_logic_vector(resize(signed(input), 32));
end;
```

Shift left:

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

```
entity shift_left is
   generic(
       K           : integer := 32;
       S           : integer := 2
   );
   port(
       input       : in STD_LOGIC_VECTOR(K-1 downto 0);
       output      : out STD_LOGIC_VECTOR(K-1 downto 0)
   );
end;

architecture behavior of shift_left is
begin
   output(K-1 downto S) <= input(K-S-1 downto 0);
   output(1 downto 0) <= "00";
end;
```

CPU:

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

-- 32-bit cpu
-- inputs are control signals
entity cpu is
   port(
       PCSource, ALUOp, ALUSrcB    : in STD_LOGIC_VECTOR(1 downto 0);
       PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemToReg,
           IRWrite, RegDst, RegWrite, ALUSrcA
                                   : in STD_LOGIC
   );
end;

architecture struct of cpu is
   -- 2 input mux
   component mux2 is
       generic(
           K           : integer  := 32
       );
       port(
           a, b        : in STD_LOGIC_VECTOR(K-1 downto 0);
```

```
        sel         : in STD_LOGIC;
        output      : out STD_LOGIC_VECTOR(K-1 downto 0)
    );
end component;

-- 4 input mux
component mux4 is
    generic(
        K: integer  := 32
    );
    port(
        a, b, c, d  : in STD_LOGIC_VECTOR(31 downto 0);
        sel         : in STD_LOGIC_VECTOR(1 downto 0);
        output      : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

-- alu
component alu is
    port(
        a, b        : in STD_LOGIC_VECTOR(31 downto 0);
        ALUOp       : in STD_LOGIC_VECTOR(1 downto 0);
        output      : out STD_LOGIC_VECTOR(31 downto 0);
        zero        : out STD_LOGIC
    );
end component;

-- memory for instructions and data
component memory is
    port(
        MemRead, MemWrite   : in STD_LOGIC;
        Address, WriteData  : in STD_LOGIC_VECTOR(31 downto 0);
        ReadData            : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

-- shift left by 2 bits (sll)
component shift_left is
    generic(
        K           : integer := 32;
        S           : integer := 2
```

```
        );
        port(
            input       : in STD_LOGIC_VECTOR(K-1 downto 0);
            output      : out STD_LOGIC_VECTOR(K-1 downto 0)
        );
    end component;

    -- sign extension
    component sign_extend is
        port(
            input       : in STD_LOGIC_VECTOR(15 downto 0);
            output      : out STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;

    -- register file
    component register_file is
        port(
            ReadReg_A, ReadReg_B, WriteReg  : in STD_LOGIC_VECTOR(4 downto 0);
            WriteData                       : in STD_LOGIC_VECTOR(31 downto 0);
            RegWrite                        : in STD_LOGIC;
            ReadData_A, ReadData_B          : out STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;

    -- registers
    signal PC, instruction, memData, A, B, ALUOut
                            : STD_LOGIC_VECTOR(31 downto 0);
    -- intermediate signals
    signal addr, WriteData, ALU_A, ALU_B, ALU_OUT, extended, offset, jump_addr, PC_in
                            : STD_LOGIC_VECTOR(31 downto 0);
    signal WriteReg         : STD_LOGIC_VECTOR(4 downto 0);
    signal zero, ChangePC   : STD_LOGIC;
begin
    IorD_Mux:       mux2 port map(PC, ALUOut, IorD, addr);
    RegDst_Mux:     mux2
                        generic map(5)
                        port map(instruction(20 downto 16), instruction(15 downto 11),
RegDst, WriteReg);
    MemToReg_Mux:   mux2 port map(ALUOut, memData, MemToReg, WriteData);
    ALUSrcA_Mux:    mux2 port map(PC, A, ALUSrcA, ALU_A);
```

```
   ALUSrcB_Mux:    mux4 port map(B, x"00000004", extended, offset, ALUSrcB, ALU_B);
   PCSource_Mux:   mux4 port map(ALU_OUT, ALUOut, jump_addr, jump_addr, PCSource,
PC_in);

   Extend:         sign_extend port map(instruction(15 downto 0), extended);
   Shift_1:        shift_left port map(extended, offset);
   Shift_2:        shift_left
                       generic map(26)
                       port map(instruction(25 downto 0), jump_addr);

   Mem:            memory port map(MemRead, MemWrite, addr, WriteData, instruction);
   Reg:            register_file port map(instruction(25 downto 21), instruction(20
downto 16),
                                          WriteReg, WriteData, RegWrite, A, B);
   ALU_1:          alu port map(ALU_A, ALU_B, ALUOp, ALU_OUT, zero);
   PC              <= PC_in when ChangePC = '1' else PC;
   ChangePC        <= PCWrite OR (PCWriteCond AND zero);
end;
```

**Appendix B: testbenches**

Adder test:

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

-- test program for adder
entity test_adder is
end;

architecture sim of test_adder is
   component adder is
       port(
           a, b, cin   : in STD_LOGIC;
           sum, carry  : out STD_LOGIC
       );
   end component;
   signal a, b, cin, sum, carry    : STD_LOGIC;
begin
   dut: adder port map(a, b, cin, sum, carry);

   process begin
       a   <= '0';
```

```
b   <= '0';
cin <= '0';
wait for 10 ns;
assert sum = '0' report "0 + 0 + 0 failed";
assert carry = '0' report "0 + 0 + 0 failed";

a   <= '0';
b   <= '0';
cin <= '1';
wait for 10 ns;
assert sum = '1' report "0 + 0 + 1 failed";
assert carry = '0' report "0 + 0 + 1 failed";

a   <= '0';
b   <= '1';
cin <= '0';
wait for 10 ns;
assert sum = '1' report "0 + 1 + 0 failed";
assert carry = '0' report "0 + 1 + 0 failed";

a   <= '0';
b   <= '1';
cin <= '1';
wait for 10 ns;
assert sum = '0' report "0 + 1 + 1 failed";
assert carry = '1' report "0 + 1 + 1 failed";

a   <= '1';
b   <= '0';
cin <= '0';
wait for 10 ns;
assert sum = '1' report "1 + 0 + 0 failed";
assert carry = '0' report "0 + 0 + 0 failed";

a   <= '1';
b   <= '0';
cin <= '1';
wait for 10 ns;
assert sum = '0' report "1 + 0 + 1 failed";
assert carry = '1' report "1 + 0 + 1 failed";
```

```
        a   <= '1';
        b   <= '1';
        cin <= '0';
        wait for 10 ns;
        assert sum = '0' report "1 + 1 + 0 failed";
        assert carry = '1' report "1 + 1 + 0 failed";

        a   <= '1';
        b   <= '1';
        cin <= '1';
        wait for 10 ns;
        assert sum = '1' report "1 + 1 + 1 failed";
        assert carry = '1' report "1 + 1 + 1 failed";

        wait;
    end process;
end;
```

Test 1-bit ALU:

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

-- test addition for 1-bit ALU
entity test_alu1_add is
end;

architecture sim of test_alu1_add is
    component alu1 is
        port(
            a, b, op, neg, Cin    : in STD_LOGIC;
            output, Cout            : out STD_LOGIC
        );
    end component;
    signal a, b, op, neg, Cin, output, Cout  : STD_LOGIC;
begin
    dut: alu1 port map(a, b, op, neg, Cin, output, Cout);

    -- set control to addition
    op      <= '1';
    neg     <= '0';
    Cin     <= '0';
```

```
    process begin
        a   <= '0';
        b   <= '0';
        wait for 10 ns;
        assert output = '0' report "0 + 0 failed";
        assert Cout = '0' report "0 + 0 failed";

        a   <= '0';
        b   <= '1';
        wait for 10 ns;
        assert output = '1' report "0 + 1 failed";
        assert Cout = '0' report "0 + 1 failed";

        a   <= '1';
        b   <= '0';
        wait for 10 ns;
        assert output = '1' report "1 + 0 failed";
        assert Cout = '0' report "1 + 0 failed";

        a   <= '1';
        b   <= '1';
        wait for 10 ns;
        assert output = '0' report "1 + 1 failed";
        assert Cout = '1' report "1 + 1 failed";

        wait;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- test subtraction for 1-bit ALU
entity test_alu1_sub is
end;

architecture sim of test_alu1_sub is
    component alu1 is
        port(
            a, b, op, neg, Cin   : in STD_LOGIC;
```

```
            output, Cout            : out STD_LOGIC
        );
    end component;
    signal a, b, op, neg, Cin, output, Cout  : STD_LOGIC;
begin
    dut: alu1 port map(a, b, op, neg, Cin, output, Cout);

    -- set control to subtraction
    op   <= '1';
    neg     <= '1';
    Cin     <= '1';

    process begin
        a   <= '0';
        b   <= '0';
        wait for 10 ns;
        assert output = '0' report "0 - 0 failed";
        assert Cout = '1' report "0 - 0 failed";

        a   <= '0';
        b   <= '1';
        wait for 10 ns;
        assert output = '1' report "0 - 1 failed";
        assert Cout = '0' report "0 - 1 failed";

        a   <= '1';
        b   <= '0';
        wait for 10 ns;
        assert output = '1' report "1 - 0 failed";
        assert Cout = '1' report "1 - 0 failed";

        a   <= '1';
        b   <= '1';
        wait for 10 ns;
        assert output = '0' report "1 - 1 failed";
        assert Cout = '1' report "1 - 1 failed";

        wait;
    end process;
end;
```

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

-- test or for 1-bit ALU
-- note that Cout is meaningless in this test
entity test_alu1_or is
end;

architecture sim of test_alu1_or is
   component alu1 is
       port(
           a, b, op, neg, Cin    : in STD_LOGIC;
           output, Cout            : out STD_LOGIC
       );
   end component;
       signal a, b, op, neg, Cin, output, Cout  : STD_LOGIC;
begin
   dut: alu1 port map(a, b, op, neg, Cin, output, Cout);

   -- set control to or
   op   <= '0';
   neg     <= '0';
   Cin     <= '0';

   process begin
       a   <= '0';
       b   <= '0';
       wait for 10 ns;
       assert output = '0' report "0 or 0 failed";

       a   <= '0';
       b   <= '1';
       wait for 10 ns;
       assert output = '1' report "0 or 1 failed";

       a   <= '1';
       b   <= '0';
       wait for 10 ns;
       assert output = '1' report "1 or 0 failed";

       a   <= '1';
```

```
        b   <= '1';
        wait for 10 ns;
        assert output = '1' report "1 or 1 failed";

        wait;
    end process;
end;
```

Test ALU:

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

-- test addition for 1-bit ALU
entity test_alu_add is
end;

architecture sim of test_alu_add is
    component alu is
        port(
            a, b        : in STD_LOGIC_VECTOR(31 downto 0);
            ALUOp       : in STD_LOGIC_VECTOR(1 downto 0);
            output      : out STD_LOGIC_VECTOR(31 downto 0);
            zero        : out STD_LOGIC
        );
    end component;
    signal a, b, output : STD_LOGIC_VECTOR(31 downto 0);
    signal zero         : STD_LOGIC;
    signal ALUOp        : STD_LOGIC_VECTOR(1 downto 0);
begin
    dut: alu port map(a, b, ALUOp, output, zero);

    -- set control to addition
    ALUOp   <= "10";

    process begin
        a   <= "00000000000000000000000000000000";
        b   <= "00000000000000000000000000000000";
        wait for 150 ns;
        assert output = "00000000000000000000000000000000" report "0 + 0 failed";
        assert zero = '1' report "0 + 0 zero detection failed";
```

```
        a   <= "00000000000000000000000000000001";
        b   <= "00000000000000000000000000000001";
        wait for 150 ns;
        assert output = "00000000000000000000000000000010" report "1 + 1 failed";
        assert zero = '0' report "1 + 1 zero detection failed";

        a   <= "10101010101010101010101010101010";
        b   <= "01010101010101010101010101010101";
        wait for 150 ns;
        assert output = "11111111111111111111111111111111" report "0xAAAA AAAA +
0x5555 5555 failed";
        assert zero = '0' report "0xAAAA AAAA + 0x5555 5555 zero detection failed";

        a   <= "01010101010101010101010101010101";
        b   <= "10101010101010101010101010101010";
        wait for 150 ns;
        assert output = "11111111111111111111111111111111" report "0x5555 5555 +
0xAAAA AAAA failed";
        assert zero = '0' report "0x5555 5555 + 0xAAAA AAAA zero detection failed";

        a   <= "11111111111111111111111111111111";
        b   <= "11111111111111111111111111111111";
        wait for 150 ns;
        assert output = "11111111111111111111111111111110" report "0xFFFF FFFF +
0xFFFF FFFF failed";
        assert zero = '0' report "0xFFFF FFFF + 0xFFFF FFFF zero detection failed";

        wait;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- test subtraction for 1-bit ALU
entity test_alu_sub is
end;

architecture sim of test_alu_sub is
    component alu is
        port(
```

```
            a, b        : in STD_LOGIC_VECTOR(31 downto 0);
            ALUOp       : in STD_LOGIC_VECTOR(1 downto 0);
            output      : out STD_LOGIC_VECTOR(31 downto 0);
            zero        : out STD_LOGIC
        );
    end component;
    signal a, b, output : STD_LOGIC_VECTOR(31 downto 0);
    signal zero         : STD_LOGIC;
    signal ALUOp        : STD_LOGIC_VECTOR(1 downto 0);
begin
    dut: alu port map(a, b, ALUOp, output, zero);

    -- set control to subtraction
    ALUOp   <= "11";

    process begin
        a   <= "00000000000000000000000000000000";
        b   <= "00000000000000000000000000000000";
        wait for 150 ns;
        assert output = "00000000000000000000000000000000" report "0 - 0 failed";
        assert zero = '1' report "0 - 0 zero detection failed";

        a   <= "00000000000000000000000000000010";
        b   <= "00000000000000000000000000000001";
        wait for 150 ns;
        assert output = "00000000000000000000000000000001" report "2 - 1 failed";
        assert zero = '0' report "2 - 1 zero detection failed";

        a   <= "00000000000000000000000000000001";
        b   <= "00000000000000000000000000000010";
        wait for 150 ns;
        assert output = "11111111111111111111111111111111" report "1 - 2 failed";
        assert zero = '0' report "1 - 2 zero detection failed";

        a   <= "10101010101010101010101010101010";
        b   <= "01010101010101010101010101010101";
        wait for 150 ns;
        assert output = "01010101010101010101010101010101" report "0xAAAA AAAA -
0x5555 5555 failed";
        assert zero = '0' report "0xAAAA AAAA - 0x5555 5555 zero detection failed";
```

```
        a   <= "01010101010101010101010101010101";
        b   <= "10101010101010101010101010101010";
        wait for 150 ns;
        assert output = "10101010101010101010101010101011" report "0x5555 5555 -
0xAAAA AAAA failed";
        assert zero = '0' report "0x5555 5555 - 0xAAAA AAAA zero detection failed";

        wait;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- test or for 1-bit ALU
entity test_alu_or is
end;

architecture sim of test_alu_or is
    component alu is
        port(
            a, b        : in STD_LOGIC_VECTOR(31 downto 0);
            ALUOp       : in STD_LOGIC_VECTOR(1 downto 0);
            output      : out STD_LOGIC_VECTOR(31 downto 0);
            zero        : out STD_LOGIC
        );
    end component;
    signal a, b, output : STD_LOGIC_VECTOR(31 downto 0);
    signal zero         : STD_LOGIC;
    signal ALUOp        : STD_LOGIC_VECTOR(1 downto 0);
begin
    dut: alu port map(a, b, ALUOp, output, zero);

    -- set control to or
    ALUOp   <= "00";

    process begin
        a   <= "00000000000000000000000000000000";
        b   <= "00000000000000000000000000000000";
        wait for 150 ns;
        assert output = "00000000000000000000000000000000" report "0 or 0 failed";
```

```
        assert zero = '1' report "0 or 0 zero detection failed";

        a   <= "00000000000000000000000000000010";
        b   <= "00000000000000000000000000000001";
        wait for 150 ns;
        assert output = "00000000000000000000000000000011" report "2 or 1 failed";
        assert zero = '0' report "2 or 1 zero detection failed";

        a   <= "10101010101010101010101010101010";
        b   <= "01010101010101010101010101010101";
        wait for 150 ns;
        assert output = "11111111111111111111111111111111" report "0xAAAA AAAA or
0x5555 5555 failed";
        assert zero = '0' report "0xAAAA AAAA or 0x5555 5555 zero detection failed";

        a   <= "01010101010101010101010101010101";
        b   <= "10101010101010101010101010101010";
        wait for 150 ns;
        assert output = "11111111111111111111111111111111" report "0x5555 5555 or
0xAAAA AAAA failed";
        assert zero = '0' report "0x5555 5555 or 0xAAAA AAAA zero detection failed";

        wait;
    end process;
end;
```

Test memory:

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

entity test_memory is
end;

architecture sim of test_memory is
    component memory is
        port(
            MemRead, MemWrite       : in STD_LOGIC;
            Address, WriteData      : in STD_LOGIC_VECTOR(31 downto 0);
            ReadData                : out STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;
```

```
    signal MemRead, MemWrite                : STD_LOGIC;
    signal Address, WriteData, ReadData     : STD_LOGIC_VECTOR(31 downto 0);
begin
    dut: memory port map(MemRead, MemWrite, Address, WriteData, ReadData);

    process begin
        -- set default memory address
        Address     <= x"00000120";

        wait for 50 ns;

        MemRead     <= '0';
        MemWrite    <= '1';
        WriteData   <= x"12345678";
        wait for 50 ns;

        MemRead     <= '1';
        MemWrite    <= '0';
        wait for 50 ns;

        assert ReadData = x"12345678";

        Address     <= x"00000030";

        wait for 50 ns;

        MemRead     <= '0';
        MemWrite    <= '1';
        WriteData   <= x"DEADBEEF";
        wait for 50 ns;

        Address     <= x"00000120";

        MemRead     <= '1';
        MemWrite    <= '0';
        wait for 50 ns;

        assert ReadData = x"12345678";

        Address     <= x"00000030";
```

```
        MemRead     <= '1';
        MemWrite    <= '0';
        wait for 50 ns;

        assert ReadData = x"DEADBEEF";

        wait;
    end process;
end;
```

Test CPU

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

entity test_cpu_lw is
end;

architecture sim of test_cpu_lw is
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    dut: cpu port map(ALUOp, ALUSrcB, PCSource, PCWriteCond, PCWrite,
                      IorD, MemRead, MemWrite, MemToReg, IRWrite, RegDst,
                      RegWrite, ALUSrcA, clk);

    clk             <= not clk after 25 ns;

    process begin
        -- setup
```

```
-- write instruction into memory location 0
-- 8E4B 012C
-- write data into register $s2
-- R18 (x12): 0000 0000
-- write data into memory
-- x12C: DEAD BEEF

-- cycle 1: IF
PCWrite     <= '1';
PCWriteCond <= '0';
IorD        <= '1';
MemRead     <= '1';
MemWrite    <= '0';
MemToReg    <= '0';
IRWrite     <= '1';
PCSource    <= '0';
ALUOp       <= "10";
ALUSrcA     <= '0';
ALUSrcB     <= "01";
RegWrite    <= '0';
RegDst      <= '0';

wait for 50 ns;

-- cycle 2: ID/RR
PCWrite     <= '0';
IRWrite     <= '0';
ALUSrcB     <= "11";
RegDst      <= '0';

wait for 50 ns;

-- cycle 3: EX
ALUSrcA     <= '1';
ALUSrcB     <= "10";

wait for 50 ns;

-- cycle 4: MEM
IorD        <= '1';
```

```
        wait for 50 ns;

        -- cycle 5: WB
        MemToReg    <= '1';
        RegWrite    <= '1';

        wait for 50 ns;

        -- check register file entry 11 (xB)
        -- value should be DEAD BEEF
        wait;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test_cpu_sw is
end;

architecture sim of test_cpu_sw is
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    dut: cpu port map(ALUOp, ALUSrcB, PCSource, PCWriteCond, PCWrite,
                      IorD, MemRead, MemWrite, MemToReg, IRWrite, RegDst,
                      RegWrite, ALUSrcA, clk);

    clk             <= not clk after 25 ns;
```

```
process begin
    -- setup
    -- write instruction into memory location 0
    -- AEEE 0190
    -- write data into register $t6 and $s7
    -- R14 (xE): DEAD BEEF
    -- R23 (x17): 0000 0000

    -- cycle 1: IF
    PCWrite     <= '1';
    PCWriteCond <= '0';
    IorD        <= '1';
    MemRead     <= '1';
    MemWrite    <= '0';
    MemToReg    <= '0';
    IRWrite     <= '1';
    PCSource    <= '0';
    ALUOp       <= "10";
    ALUSrcA     <= '0';
    ALUSrcB     <= "01";
    RegWrite    <= '0';
    RegDst      <= '0';

    wait for 50 ns;

    -- cycle 2: ID/RR
    PCWrite     <= '0';
    IRWrite     <= '0';
    ALUSrcB     <= "11";
    RegDst      <= '0';

    -- cycle 3: EX
    ALUSrcA     <= '1';
    ALUSrcB     <= "10";

    wait for 50 ns;

    -- cycle 4: MEM
    IorD        <= '1';
    MemWrite    <= '1';
```

```
        wait for 50 ns;

        -- check memory location x190
        -- value should be DEAD BEEF
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test_cpu_add is
end;

architecture sim of test_cpu_add is
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    process begin
        -- setup
        -- write instruction into memory location 0
        -- 0170 0190
        -- write data into register $t3, $s1
        -- R11 (xB): 0000 0001
        -- R17 (x11): 0000 0002

        -- cycle 1: IF
        PCWrite     <= '1';
        PCWriteCond <= '0';
        IorD        <= '1';
```

```
        MemRead     <= '1';
        MemWrite    <= '0';
        MemToReg    <= '0';
        IRWrite     <= '1';
        PCSource    <= '0';
        ALUOp       <= "10";
        ALUSrcA     <= '0';
        ALUSrcB     <= "01";
        RegWrite    <= '0';
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 2: ID/RR
        PCWrite     <= '0';
        IRWrite     <= '0';
        ALUSrcB     <= "11";
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 3: EX
        ALUSrcA     <= '1';
        ALUSrcB     <= "00";

        -- cycle 4: WB
        MemToReg    <= '0';
        RegWrite    <= '1';

        -- check register $t5
        -- R13 (xD): 0000 0003
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test_cpu_beq is
end;

architecture sim of test_cpu_beq is
```

```
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    dut: cpu port map(ALUOp, ALUSrcB, PCSource, PCWriteCond, PCWrite,
                      IorD, MemRead, MemWrite, MemToReg, IRWrite, RegDst,
                      RegWrite, ALUSrcA, clk);

    clk             <= not clk after 25 ns;
    process begin
        -- setup
        -- write instruction into memory location 0
        -- 11B6 00C8
        -- write data into register $s6, $t5
        -- R22 (x16): 1111 1111
        -- R13 (xD): 1111 1111

        -- cycle 1: IF
        PCWrite     <= '1';
        PCWriteCond <= '0';
        IorD        <= '1';
        MemRead     <= '1';
        MemWrite    <= '0';
        MemToReg    <= '0';
        IRWrite     <= '1';
        PCSource    <= '0';
        ALUOp       <= "10";
        ALUSrcA     <= '0';
        ALUSrcB     <= "01";
        RegWrite    <= '0';
```

```
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 2: ID/RR
        PCWrite     <= '0';
        IRWrite     <= '0';
        ALUSrcB     <= "11";
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 3: EX
        ALUOp       <= "11";
        ALUSrcA     <= '1';
        ALUSrcB     <= "00";
        PCWriteCond <= '1';

        -- check PC: value should be branch address
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test_cpu_subi is
end;

architecture sim of test_cpu_subi is
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
```

```
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    dut: cpu port map(ALUOp, ALUSrcB, PCSource, PCWriteCond, PCWrite,
                      IorD, MemRead, MemWrite, MemToReg, IRWrite, RegDst,
                      RegWrite, ALUSrcA, clk);

    clk             <= not clk after 25 ns;
    process begin
        -- setup

        -- cycle 1: IF
        PCWrite     <= '1';
        PCWriteCond <= '0';
        IorD        <= '1';
        MemRead     <= '1';
        MemWrite    <= '0';
        MemToReg    <= '0';
        IRWrite     <= '1';
        PCSource    <= '0';
        ALUOp       <= "10";
        ALUSrcA     <= '0';
        ALUSrcB     <= "01";
        RegWrite    <= '0';
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 2: ID/RR
        PCWrite     <= '0';
        IRWrite     <= '0';
        ALUSrcB     <= "11";
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 3: EX
        ALUOp       <= "11";
        ALUSrcA     <= '1';
        ALUSrcB     <= "10";
```

```
        -- cycle 4: WB
        MemToReg    <= '0';
        RegWrite    <= '1';

        -- check
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity test_cpu_ori is
end;

architecture sim of test_cpu_ori is
    component cpu is
        port(
            ALUOp, ALUSrcB              : in STD_LOGIC_VECTOR(1 downto 0);
            PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
                MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA, clk
                                        : in STD_LOGIC
        );
    end component;

    signal ALUOp, ALUSrcB               : STD_LOGIC_VECTOR(1 downto 0);
    signal PCSource, PCWriteCond, PCWrite, IorD, MemRead, MemWrite,
           MemToReg, IRWrite, RegDst, RegWrite, ALUSrcA
                                        : STD_LOGIC;
    signal clk                          : STD_LOGIC := '0';
begin
    dut: cpu port map(ALUOp, ALUSrcB, PCSource, PCWriteCond, PCWrite,
                      IorD, MemRead, MemWrite, MemToReg, IRWrite, RegDst,
                      RegWrite, ALUSrcA, clk);

    clk             <= not clk after 25 ns;
    process begin
        -- setup

        -- cycle 1: IF
        PCWrite     <= '1';
        PCWriteCond <= '0';
```

```
        IorD        <= '1';
        MemRead     <= '1';
        MemWrite    <= '0';
        MemToReg    <= '0';
        IRWrite     <= '1';
        PCSource    <= '0';
        ALUOp       <= "10";
        ALUSrcA     <= '0';
        ALUSrcB     <= "01";
        RegWrite    <= '0';
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 2: ID/RR
        PCWrite     <= '0';
        IRWrite     <= '0';
        ALUSrcB     <= "11";
        RegDst      <= '0';

        wait for 50 ns;

        -- cycle 3: EX
        ALUOp       <= "00";
        ALUSrcA     <= '1';
        ALUSrcB     <= "00";

        -- cycle 4: WB
        MemToReg    <= '0';
        RegWrite    <= '1';

        -- check
    end process;
end;
```