West Virginia University - Statler College of Engineering and Mineral Resources
Lane Department of Computer Science and Electrical Engineering

Final Project
Design of a Simple CPU

CpE 272 Digital Logic Laboratory
Fall 2022

Author: Azain Uqaily
Co-Author: Dhyan Patel

Instructor : Ivy Yousuf Moutushi

06/12/2022

Table Of Contents

1. Introduction

After a semester of being introduced to digital logic and design, this complex final project allows students to apply their gained knowledge. This project tasked us with building a simple Central Processing Unit(CPU).

This CPU has nine total components listed as follows: Accumulator Register (A), instruction register (IR) , control unit (CU), program counter (PC) Arithmetic Logic Unit(ALU), Memory Address Register (MAR), Memory Data Register Input (MDRI), Memory Data Register output (MDRO) and Random Access Memory (RAM).

Ultimately, this CPU is driven by the clock. It reads from the first address (8 bits) and decodes it. The first 3 bits point to the operation being done and the following 5 are a memory location. The three operations are LOADA (000) ADDA(001) and STOREA (010). The load operation loads the value at the memory address into the accumulator. The add operation adds the memory value to the accumulator. The store operation stores the value of the accumulator to the specified memory location. Since the RAM is hard coded with a few addresses meant to be these three commands this CPU pretty much runs itself traversing through each address instruction.

2. Hardware Description

The hardware used in this class and the final project is the DE10 Lite Board. This is an FPGA board which acts as a reprogrammable chip, allowing us to implement various functionalities through software. The DE10 lite board is a great board due to its versatile functionality. It has been used throughout the semester to implement various software such as a 7 segment display unit, adders, arithmetic logic units and much more.

3. Software Description

The software used in this class and the final project is Intel's Quartus Prime Lite Software. It allowed us to make schematic and VHDL files which are needed to implement our labs.Quartus prime made it very easy for us to implement our code on the FPGA board. By compiling our code, by which Quartus checks for errors and through correct pin assignment we can program our boards. Quartus prime offers many benefits such as waveforms. Waveforms allow the user to test outputs without having to connect to a physical board.

4. Description of each Vhdl File

The project is basically divided into different vhdl files containing codes of each different component of the Control Processing Unit such as the Control Unit file , Program Counter , most important is Arithmetic & Logic Unit, Memory files for addressing and registering the input and output. To begin with the **Program Counter** code it holds the 8 bit memory address so that the instructions provided in the code are executable, ,but the most important part is the memory component holds only a 5 bit address so it is allowed to take only that amount of memory instructions. In the code it gives output when positive edge -1 and clock event =1).
providing the counter increases by 1.

Additionally, in the **ALU** code it operates the 8 inputs and gives us the 8 bit output results. It uses operations including addition, subtraction, bitwise AND, bitwise OR, output A, and output B. It uses Input B from the Accumulator and Input A from the MDRI. Most of the code in the ALU file uses if (sequential statements).

Also, the **accumulator** is the component that provides the 8 bit output from the ALU. It basically acts as loop - being each one of the ALU and accumulator acting as input and output and vice- versa, this only happens when the clk and load =1 giving the provided input to its output.

Moreover, **MDR**I is a 8-bit register that holds a value read from memory, and passes it to the component when it needs to be implemented and executed.

Port-mapping (port-map statements) go in each of the component files as that is the part of the code that is going to give the results in general.

Furthermore in the **Control Unit** code, it is as important as the Program counter code and other vhdl files. In the Control Unit it acts as a leader of all components where it coordinates the operations of CPU components including incrementing the PC, accessing memory/registers, and ALU operations. The control unit models as a state machine that has instructions representing a state. The CU determines which register is active and what the next state is going to be. Therefore, for this signals are used in the code so that all information and instructions are done in the right order.

The **Reg** (Register) file is the code for each of the register components. Components including the accumulator, IR, MAR, MDRI, MDRO use the register code for their port maps.

The **TwoToOneMux** file is placed before the MAR as it only receives one out of two inputs which is controlled by the control unit.

The **SevenSeg** file converts binary outputs to be able to be shown on a seven segment display unit. The boolean expressions for these were taken from lab 2. Accumulator and PC are displayed.

5. Problems occurred and solutions

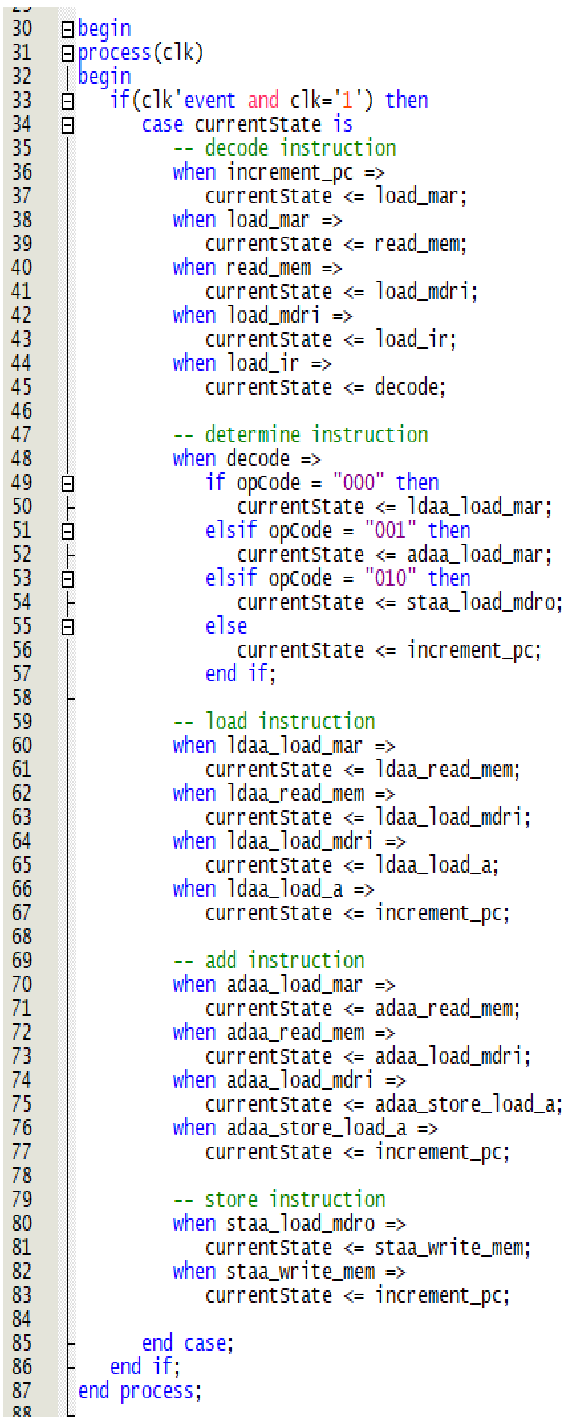
One of the problems faced was the connections made to each element. Due to the complexity of the components and trying to understand the flow of the CPU it helped to read the in-depth provided notes. What took a while to understand was that the full output of the internal register goes to the MAR but must pass through the multiplexer first.

Another big issue was file path and file name errors when the waveform was being simulated. This issue was solved by copy pasting each separate VHDL file into a new version and all the files were neatly placed in respected folders. Another sub issue to this was a bit technical for which the intel support page was used. For some reason the output waveform file was being saved in the Output files as it should instead be placed with the other VHDL files.

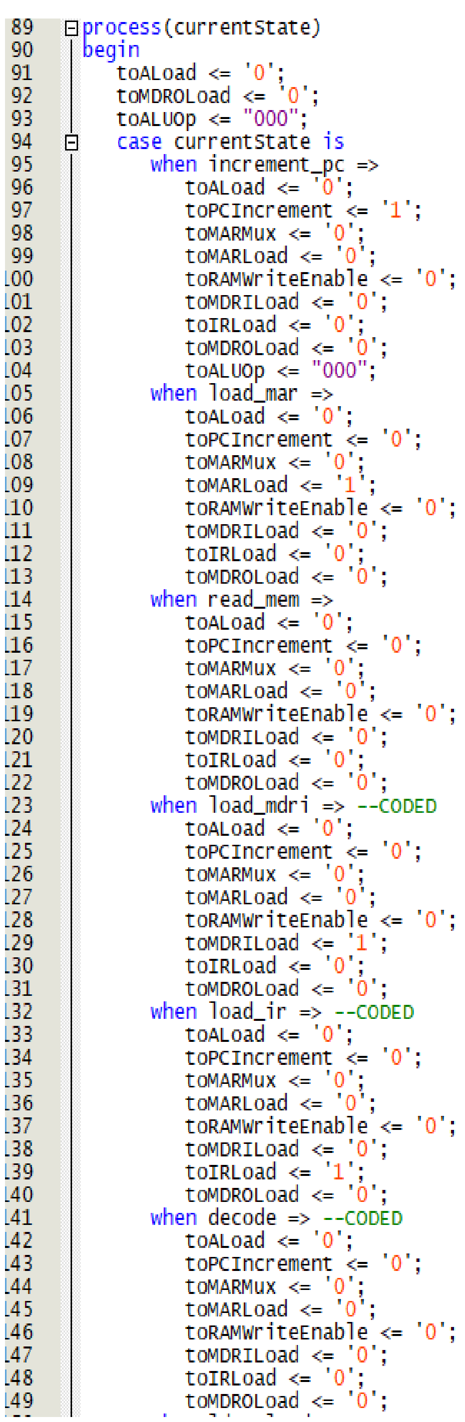
6. Completed Code

(Rest of code files are inserted in appendix) These two were significant as they were not entirely provided.

Control Unit Code



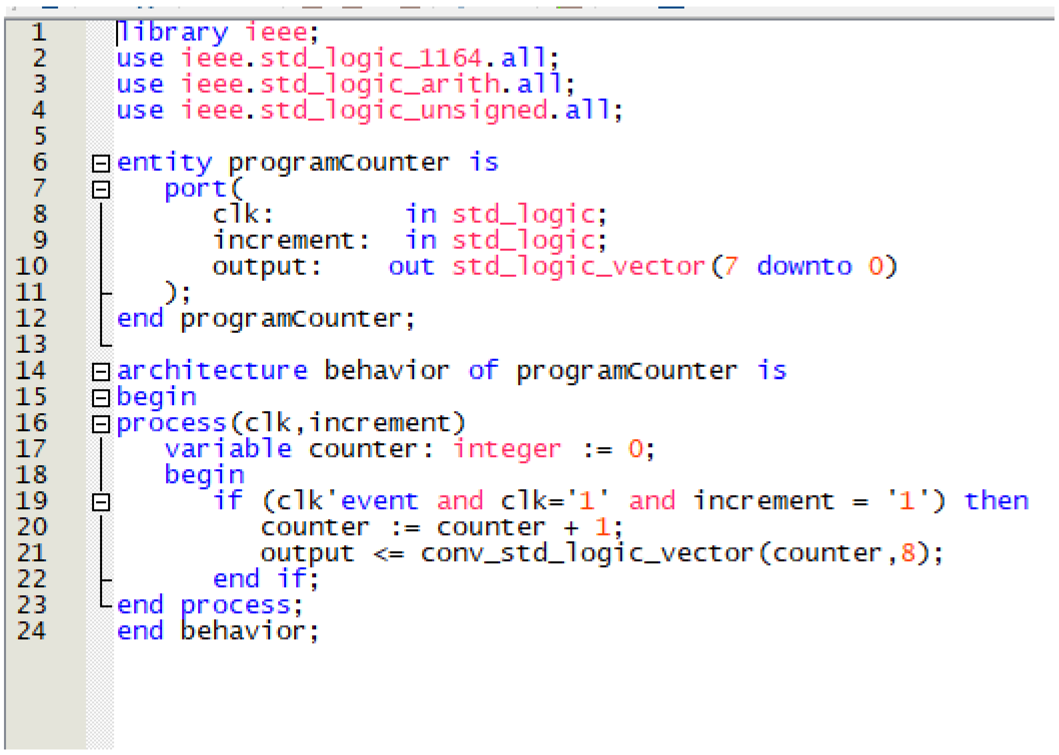
```
begin
process(clk)
begin
   if(clk'event and clk='1') then
      case currentState is
         -- decode instruction
         when increment_pc =>
            currentState <= load_mar;
         when load_mar =>
            currentState <= read_mem;
         when read_mem =>
            currentState <= load_mdri;
         when load_mdri =>
            currentState <= load_ir;
         when load_ir =>
            currentState <= decode;

         -- determine instruction
         when decode =>
            if opCode = "000" then
               currentState <= ldaa_load_mar;
            elsif opCode = "001" then
               currentState <= adaa_load_mar;
            elsif opCode = "010" then
               currentState <= staa_load_mdro;
            else
               currentState <= increment_pc;
            end if;

         -- load instruction
         when ldaa_load_mar =>
            currentState <= ldaa_read_mem;
         when ldaa_read_mem =>
            currentState <= ldaa_load_mdri;
         when ldaa_load_mdri =>
            currentState <= ldaa_load_a;
         when ldaa_load_a =>
            currentState <= increment_pc;

         -- add instruction
         when adaa_load_mar =>
            currentState <= adaa_read_mem;
         when adaa_read_mem =>
            currentState <= adaa_load_mdri;
         when adaa_load_mdri =>
            currentState <= adaa_store_load_a;
         when adaa_store_load_a =>
            currentState <= increment_pc;

         -- store instruction
         when staa_load_mdro =>
            currentState <= staa_write_mem;
         when staa_write_mem =>
            currentState <= increment_pc;

      end case;
   end if;
end process;
```



```
process(currentState)
begin
   toALoad <= '0';
   toMDROLoad <= '0';
   toALUOp <= "000";
   case currentState is
      when increment_pc =>
         toALoad <= '0';
         toPCIncrement <= '1';
         toMARMux <= '0';
         toMARLoad <= '0';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '0';
         toIRLoad <= '0';
         toMDROLoad <= '0';
         toALUOp <= "000";
      when load_mar =>
         toALoad <= '0';
         toPCIncrement <= '0';
         toMARMux <= '0';
         toMARLoad <= '1';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '0';
         toIRLoad <= '0';
         toMDROLoad <= '0';
      when read_mem =>
         toALoad <= '0';
         toPCIncrement <= '0';
         toMARMux <= '0';
         toMARLoad <= '0';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '0';
         toIRLoad <= '0';
         toMDROLoad <= '0';
      when load_mdri => --CODED
         toALoad <= '0';
         toPCIncrement <= '0';
         toMARMux <= '0';
         toMARLoad <= '0';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '1';
         toIRLoad <= '0';
         toMDROLoad <= '0';
      when load_ir => --CODED
         toALoad <= '0';
         toPCIncrement <= '0';
         toMARMux <= '0';
         toMARLoad <= '0';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '0';
         toIRLoad <= '1';
         toMDROLoad <= '0';
      when decode => --CODED
         toALoad <= '0';
         toPCIncrement <= '0';
         toMARMux <= '0';
         toMARLoad <= '0';
         toRAMWriteEnable <= '0';
         toMDRILoad <= '0';
         toIRLoad <= '0';
         toMDROLoad <= '0';
```

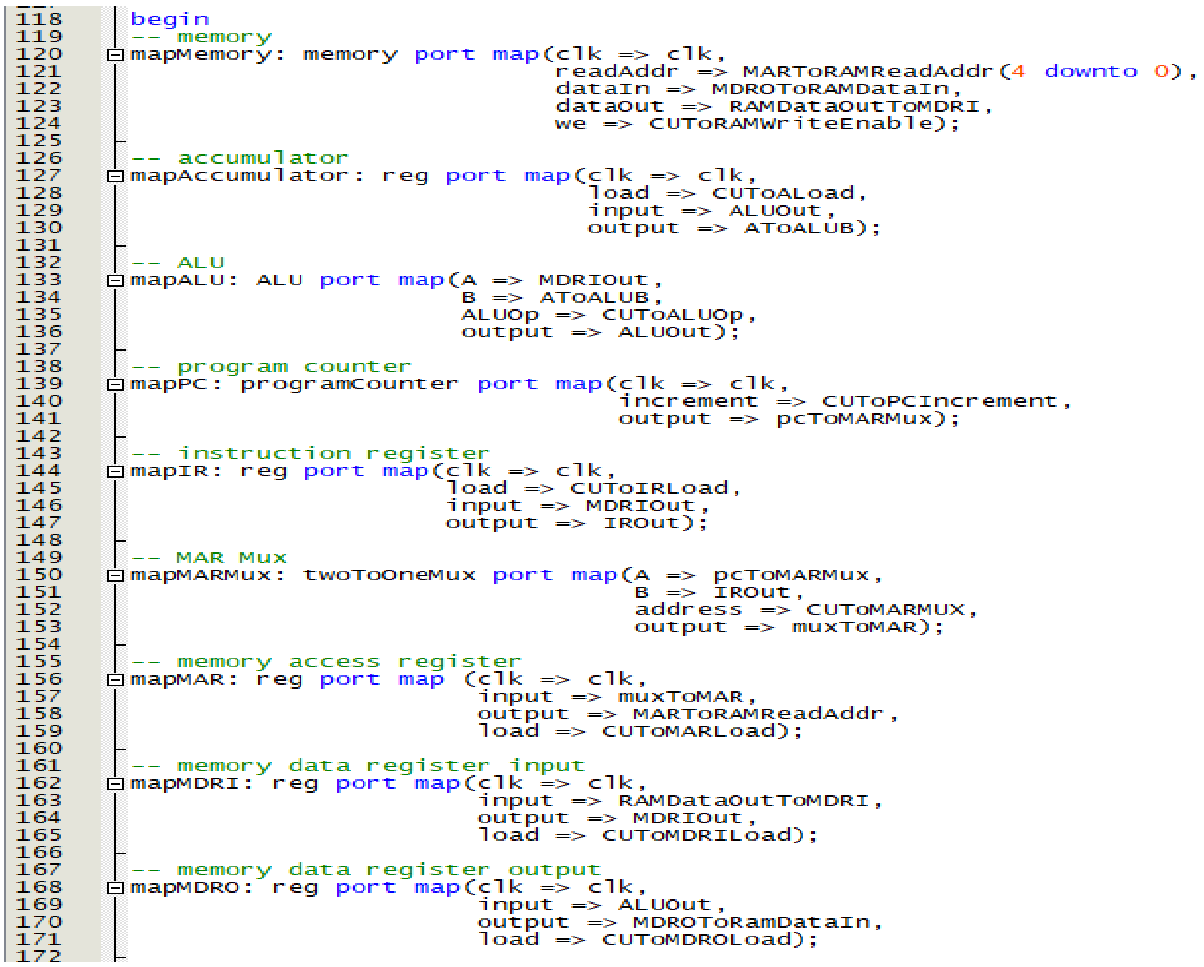
```
when ldaa_load_mar =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '1';
    toMARLoad <= '1';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
    toALUOp <= "101";
when ldaa_read_mem =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
when ldaa_load_mdri =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '1';
    toIRLoad <= '0';
    toMDROLoad <= '0';
    toALUOp <= "101";
when ldaa_load_a =>
    toALoad <= '1';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
    toALUOp <= "101";
when adaa_load_mar =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '1';
    toMARLoad <= '1';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
    toALUOp <= "000";
when adaa_read_mem =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
when adaa_load_mdri =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '1';
    toIRLoad <= '0';
    toMDROLoad <= '0';
when adaa_store_load_a =>
    toALoad <= '1';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
    toALUOp <= "000";
when staa_load_mdro =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '1';
    toMARLoad <= '1';
    toRAMWriteEnable <= '0';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '1';
    toALUOp <= "100";
when staa_write_mem =>
    toALoad <= '0';
    toPCIncrement <= '0';
    toMARMux <= '0';
    toMARLoad <= '0';
    toRAMWriteEnable <= '1';
    toMDRILoad <= '0';
    toIRLoad <= '0';
    toMDROLoad <= '0';
  end case;
end process;
end behavior;
```

Program Counter Code

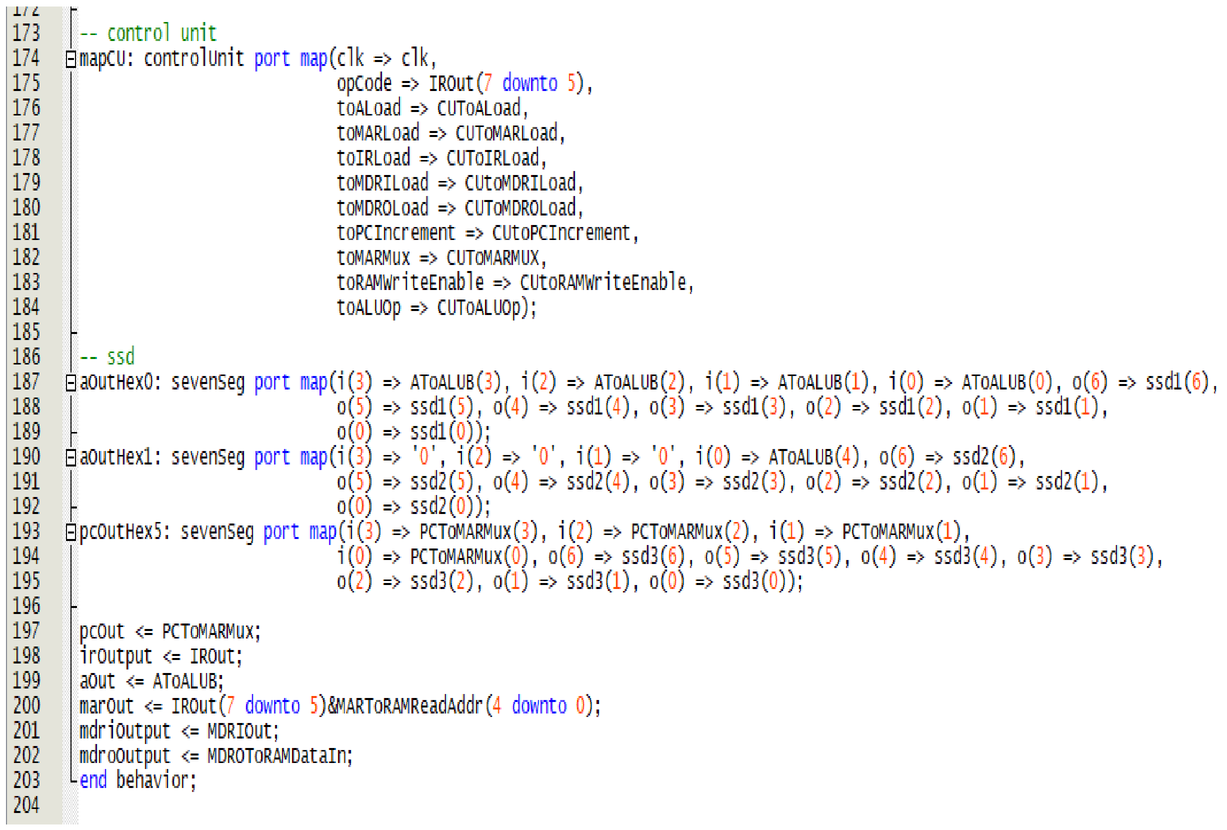


```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity programCounter is
   port(
      clk:         in std_logic;
      increment:   in std_logic;
      output:     out std_logic_vector(7 downto 0)
   );
end programCounter;

architecture behavior of programCounter is
begin
process(clk,increment)
   variable counter: integer := 0;
   begin
      if (clk'event and clk='1' and increment = '1') then
         counter := counter + 1;
         output <= conv_std_logic_vector(counter,8);
      end if;
end process;
end behavior;
```

CPU Code (port map statements)

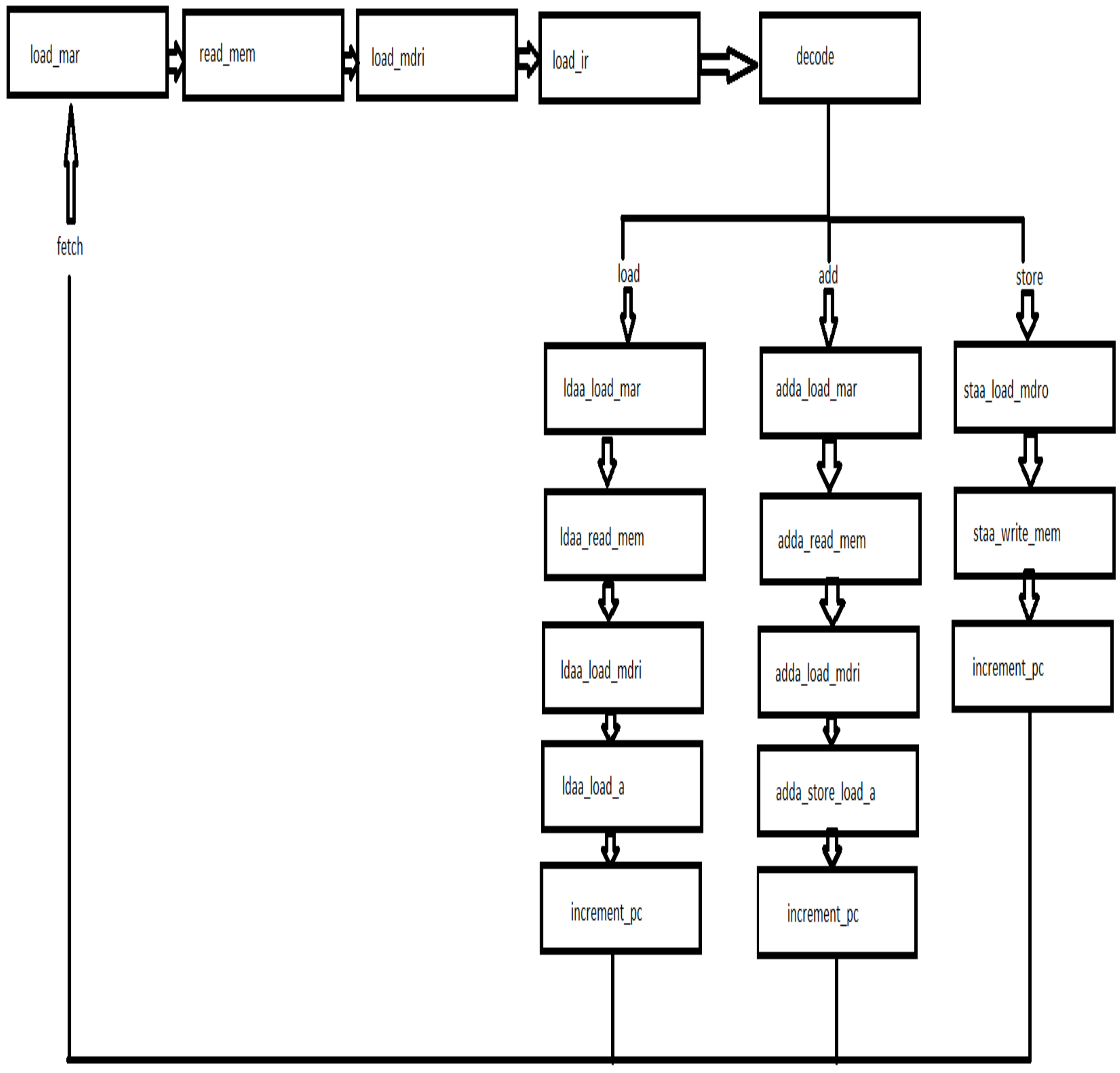


```vhdl
begin
-- memory
mapMemory: memory port map(clk => clk,
                           readAddr => MARToRAMReadAddr(4 downto 0),
                           dataIn => MDROToRAMDataIn,
                           dataOut => RAMDataOutToMDRI,
                           we => CUToRAMWriteEnable);

-- accumulator
mapAccumulator: reg port map(clk => clk,
                             load => CUToALoad,
                             input => ALUOut,
                             output => AToALUB);

-- ALU
mapALU: ALU port map(A => MDRIOut,
                     B => AToALUB,
                     ALUOp => CUToALUOp,
                     output => ALUOut);

-- program counter
mapPC: programCounter port map(clk => clk,
                               increment => CUToPCIncrement,
                               output => pcToMARMux);

-- instruction register
mapIR: reg port map(clk => clk,
                    load => CUToIRLoad,
                    input => MDRIOut,
                    output => IROut);

-- MAR Mux
mapMARMux: twoToOneMux port map(A => pcToMARMux,
                                B => IROut,
                                address => CUToMARMUX,
                                output => muxToMAR);

-- memory access register
mapMAR: reg port map (clk => clk,
                      input => muxToMAR,
                      output => MARToRAMReadAddr,
                      load => CUToMARLoad);

-- memory data register input
mapMDRI: reg port map(clk => clk,
                      input => RAMDataOutToMDRI,
                      output => MDRIOut,
                      load => CUToMDRILoad);

-- memory data register output
mapMDRO: reg port map(clk => clk,
                      input => ALUOut,
                      output => MDROToRamDataIn,
                      load => CUToMDROLoad);

```

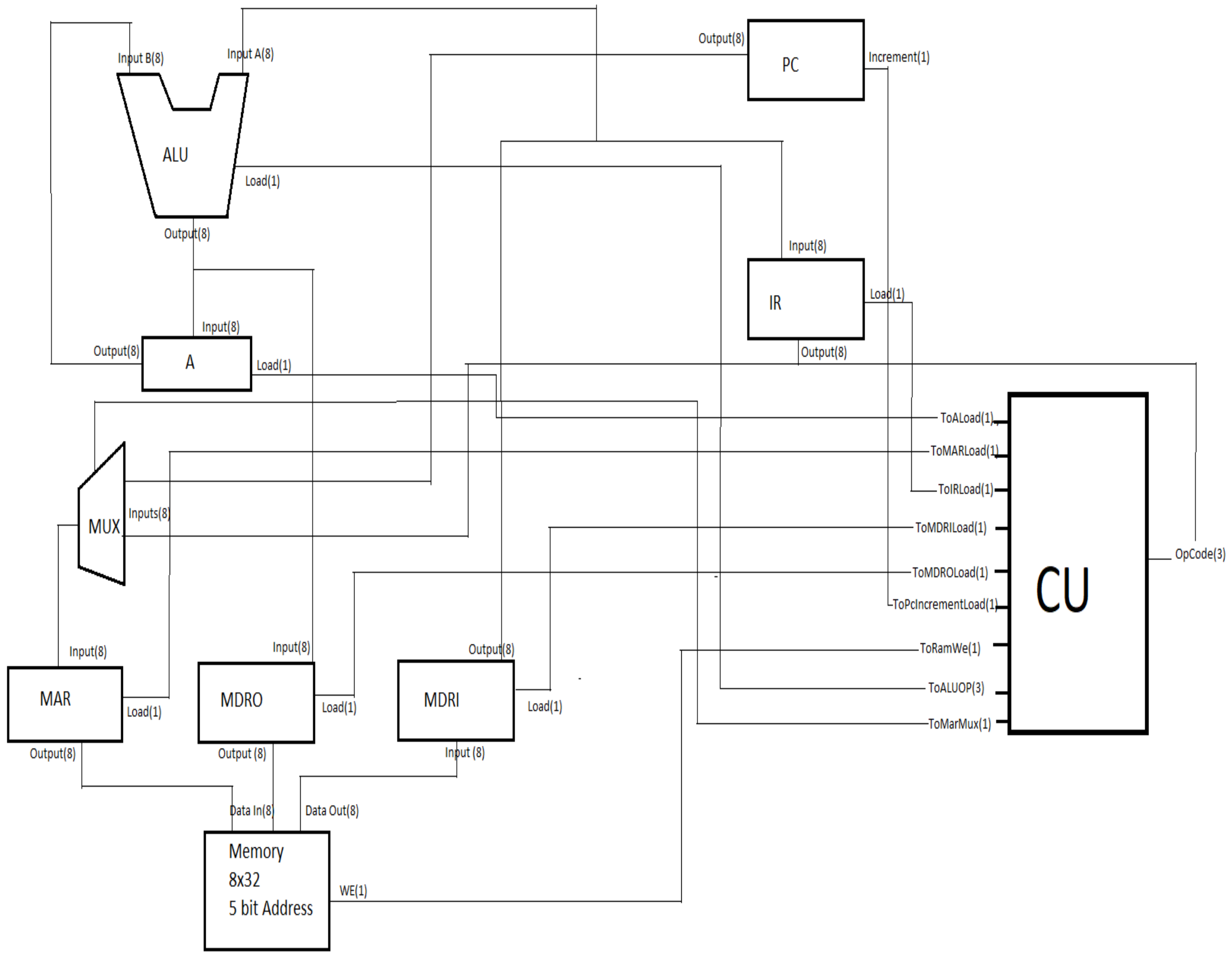


```vhdl

-- control unit
mapCU: controlUnit port map(clk => clk,
                            opCode => IROut(7 downto 5),
                            toALoad => CUToALoad,
                            toMARLoad => CUToMARLoad,
                            toIRLoad => CUToIRLoad,
                            toMDRILoad => CUToMDRILoad,
                            toMDROLoad => CUToMDROLoad,
                            toPCIncrement => CUtoPCIncrement,
                            toMARMux => CUToMARMUX,
                            toRAMWriteEnable => CUtoRAMWriteEnable,
                            toALUOp => CUToALUOp);

-- ssd
aOutHex0: sevenSeg port map(i(3) => AToALUB(3), i(2) => AToALUB(2), i(1) => AToALUB(1), i(0) => AToALUB(0), o(6) => ssd1(6),
                            o(5) => ssd1(5), o(4) => ssd1(4), o(3) => ssd1(3), o(2) => ssd1(2), o(1) => ssd1(1),
                            o(0) => ssd1(0));
aOutHex1: sevenSeg port map(i(3) => '0', i(2) => '0', i(1) => '0', i(0) => AToALUB(4), o(6) => ssd2(6),
                            o(5) => ssd2(5), o(4) => ssd2(4), o(3) => ssd2(3), o(2) => ssd2(2), o(1) => ssd2(1),
                            o(0) => ssd2(0));
pcOutHex5: sevenSeg port map(i(3) => PCToMARMux(3), i(2) => PCToMARMux(2), i(1) => PCToMARMux(1),
                            i(0) => PCToMARMux(0), o(6) => ssd3(6), o(5) => ssd3(5), o(4) => ssd3(4), o(3) => ssd3(3),
                            o(2) => ssd3(2), o(1) => ssd3(1), o(0) => ssd3(0));

pcOut <= PCToMARMux;
irOutput <= IROut;
aOut <= AToALUB;
marOut <= IROut(7 downto 5)&MARToRAMReadAddr(4 downto 0);
mdriOutput <= MDRIOut;
mdroOutput <= MDROToRAMDataIn;
end behavior;

```

7. Finite State Machine Diagram

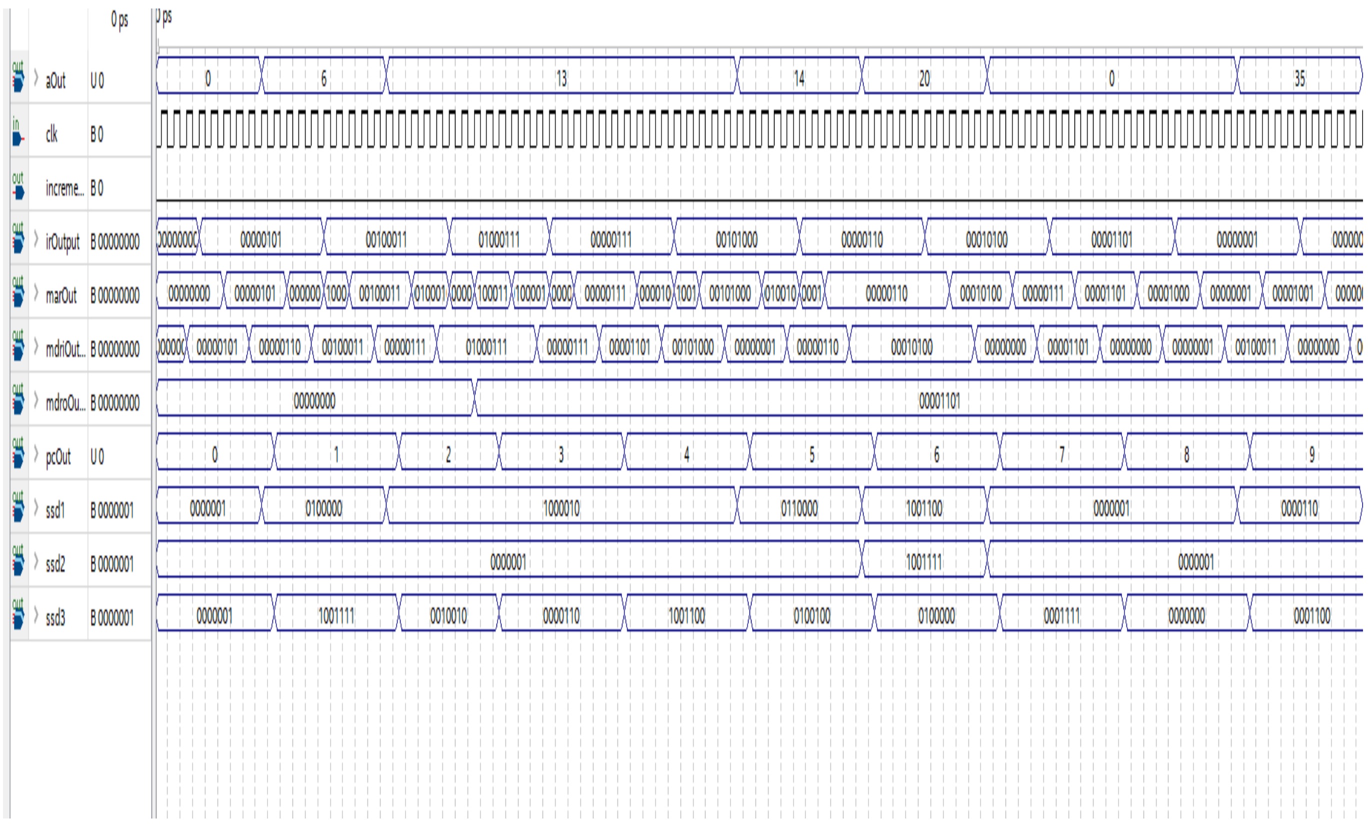


8. Block Diagram with components and Connections



9. Results

Memory Data					
RAM	**Binary**	**OpCode(3 MSB from binary)**	**Address(5 LSB from binary)**	**Value**	**Accumulator**
0	**00000101**	**000–loadA**	**00101**	**6**	**6**
1	**00100011**	**001–addA**	**00011**	**7**	**13**
2	**01000111**	**010–storeA**	**00111**	**13**	**13**
3	**00000111**	**000–loadA**	**00111**	**13**	**13**
4	**00101000**	**001–addA**	**01000**	**1**	**14**
5	**00000110**	**010–storeA**	**00110**	**20**	**20**
6	**00010100**	**010–storeA**	**10100**	**0**	**0**
7	**00001101**	**010–storeA**	**01101**	**0**	**0**
8	**00000001**	**010–storeA**	**00001**	**35**	**35**



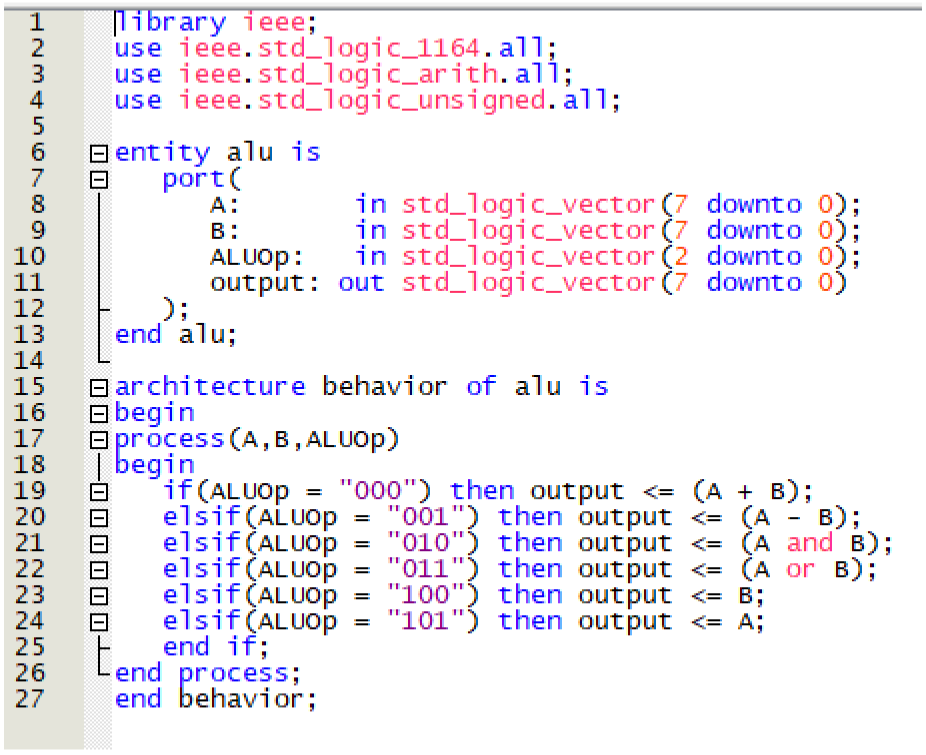
As we can see in this output waveform the value in the accumulator changes as the CPU traverses the hard coded RAM. The first address is 00000101 so it will load (000) the value of address 00101 (5) which is 6 into the accumulator. Then on the next clock cycle it will add the value of address 3 to the accumulator which is 7. So now 7 + 6 is 13 which is shown in the accumulator at this point in time.

10. Conclusion

This project does a good job of integrating key ideas learned throughout the semester. Main concepts such as ALU, memory and finite state machines are used to build this “simple” CPU. Even after completion of this project it still feels very daunting and complex for an introduction to digital logic design course. I think what would make this CPU better would be if we could be able to perform more operations or some method of user interface to perform desired tasks.

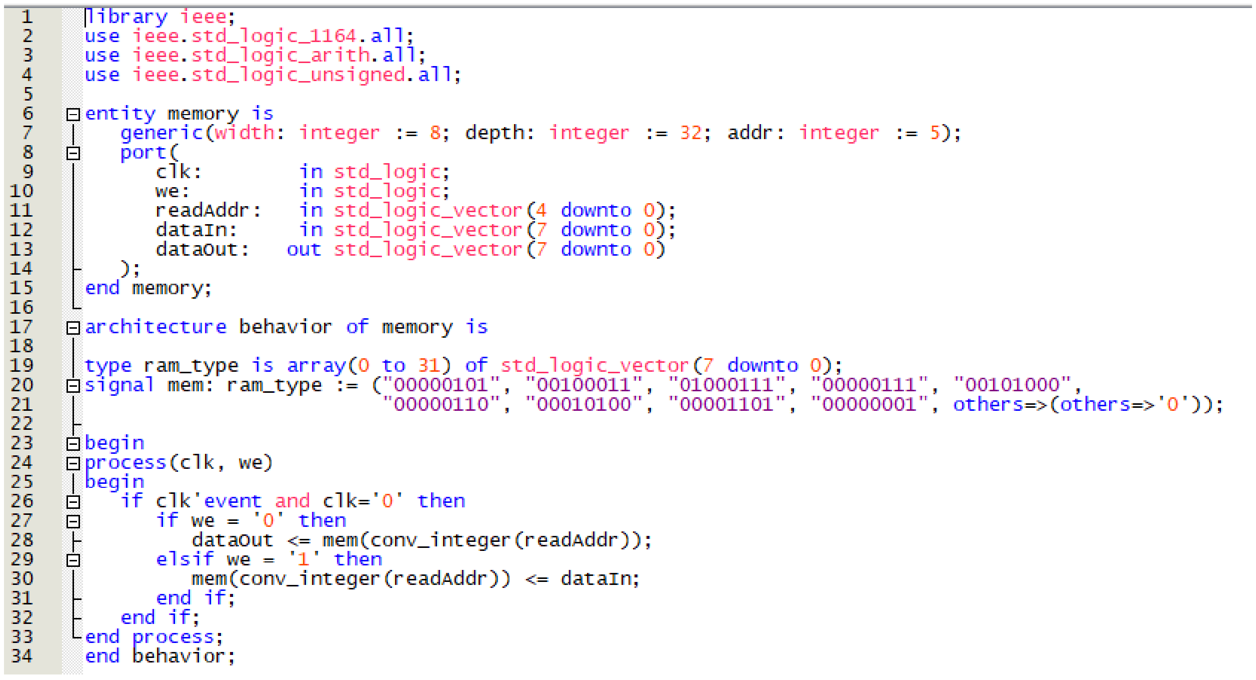
11. Appendix

ALU Code



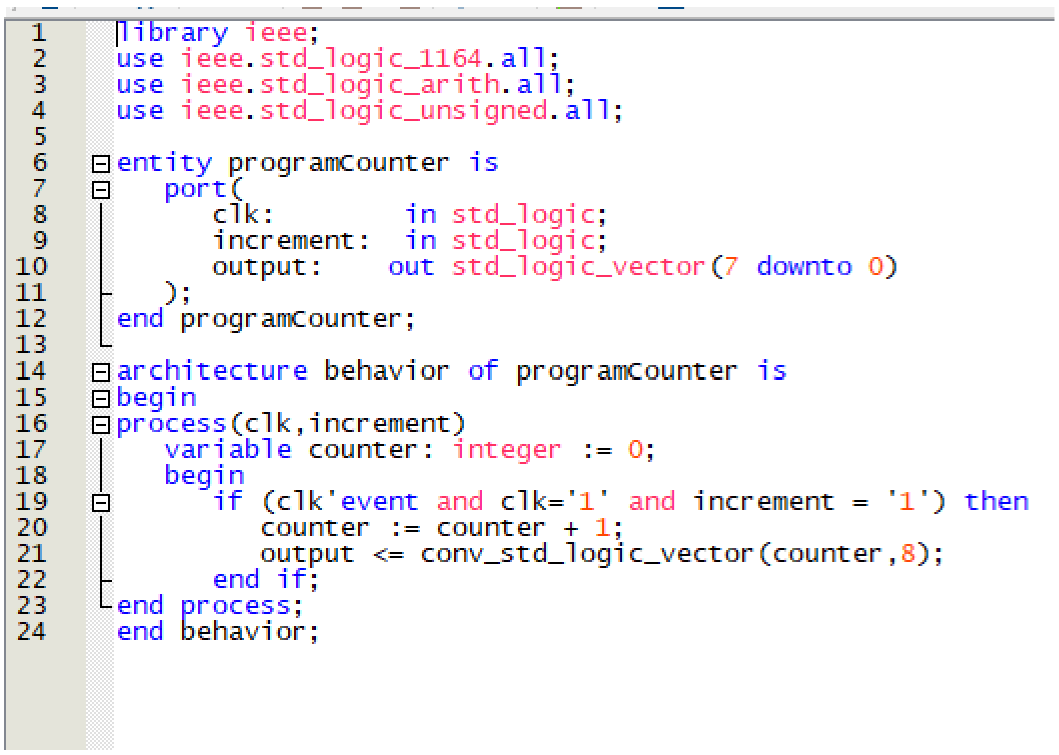
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu is
    port(
        A:        in std_logic_vector(7 downto 0);
        B:        in std_logic_vector(7 downto 0);
        ALUOp:    in std_logic_vector(2 downto 0);
        output:  out std_logic_vector(7 downto 0)
    );
end alu;

architecture behavior of alu is
begin
process(A,B,ALUOp)
begin
    if(ALUOp = "000") then output <= (A + B);
    elsif(ALUOp = "001") then output <= (A - B);
    elsif(ALUOp = "010") then output <= (A and B);
    elsif(ALUOp = "011") then output <= (A or B);
    elsif(ALUOp = "100") then output <= B;
    elsif(ALUOp = "101") then output <= A;
    end if;
end process;
end behavior;
```

Memory Code



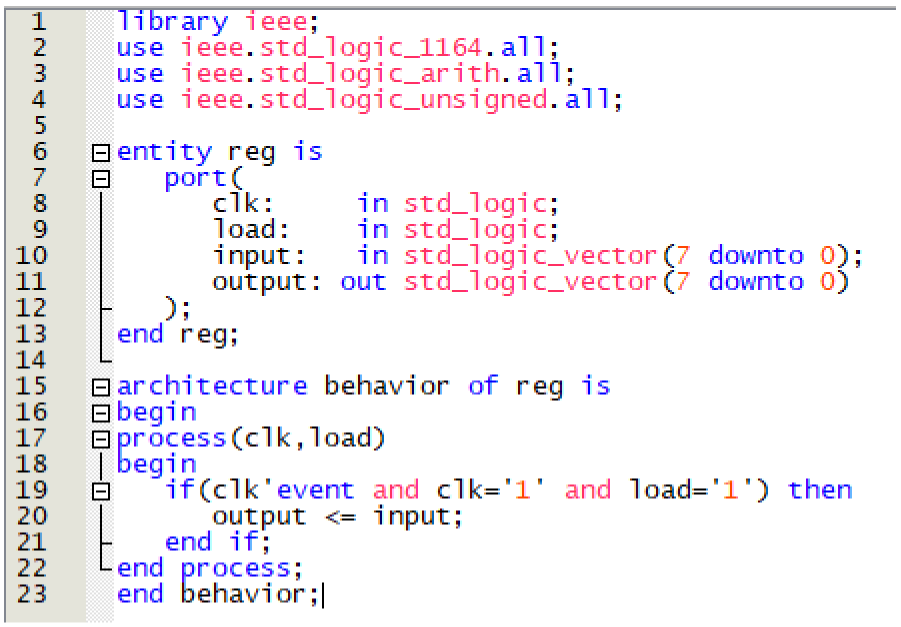
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity memory is
    generic(width: integer := 8; depth: integer := 32; addr: integer := 5);
    port(
        clk:        in std_logic;
        we:         in std_logic;
        readAddr:   in std_logic_vector(4 downto 0);
        dataIn:     in std_logic_vector(7 downto 0);
        dataOut:   out std_logic_vector(7 downto 0)
    );
end memory;

architecture behavior of memory is

type ram_type is array(0 to 31) of std_logic_vector(7 downto 0);
signal mem: ram_type := ("00000101", "00100011", "01000111", "00000111", "00101000",
                         "00000110", "00010100", "00001101", "00000001", others=>(others=>'0'));

begin
process(clk, we)
begin
    if clk'event and clk='0' then
        if we = '0' then
            dataOut <= mem(conv_integer(readAddr));
        elsif we = '1' then
            mem(conv_integer(readAddr)) <= dataIn;
        end if;
    end if;
end process;
end behavior;
```

Program Counter Code



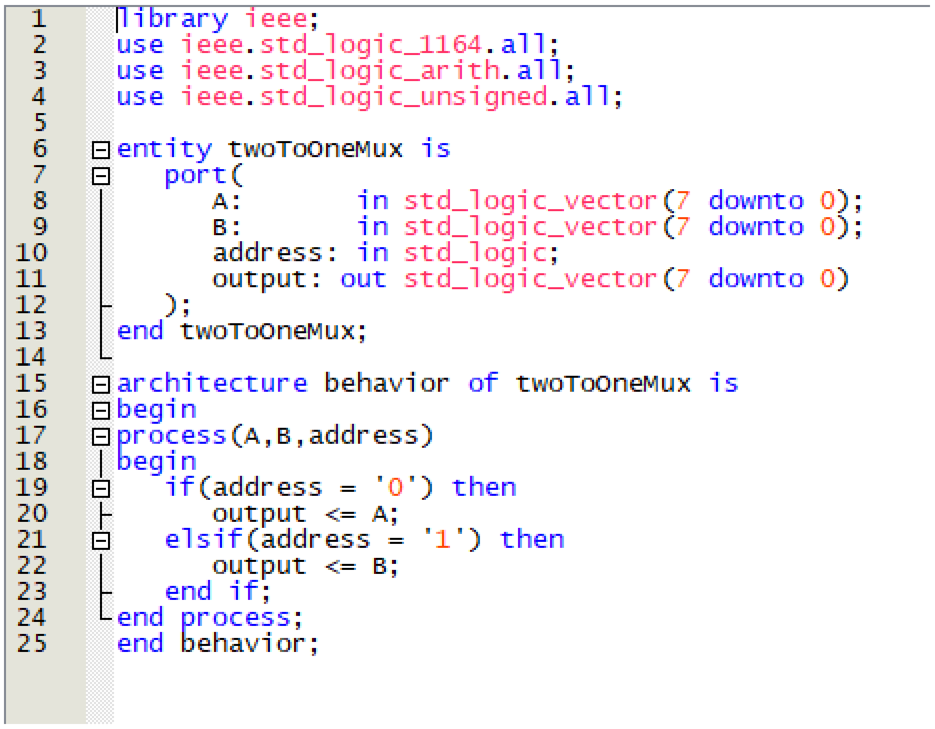
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity programCounter is
    port(
        clk:        in std_logic;
        increment:  in std_logic;
        output:    out std_logic_vector(7 downto 0)
    );
end programCounter;

architecture behavior of programCounter is
begin
process(clk,increment)
    variable counter: integer := 0;
    begin
        if (clk'event and clk='1' and increment = '1') then
            counter := counter + 1;
            output <= conv_std_logic_vector(counter,8);
        end if;
end process;
end behavior;
```

Registers Code



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

entity reg is
    port(
        clk:     in std_logic;
        load:    in std_logic;
        input:   in std_logic_vector(7 downto 0);
        output: out std_logic_vector(7 downto 0)
    );
end reg;

architecture behavior of reg is
begin
process(clk,load)
begin
    if(clk'event and clk='1' and load='1') then
        output <= input;
    end if;
end process;
end behavior;
```

TwoToOneMuxCode



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

entity twoToOneMux is
   port(
      A:        in std_logic_vector(7 downto 0);
      B:        in std_logic_vector(7 downto 0);
      address: in std_logic;
      output: out std_logic_vector(7 downto 0)
   );
end twoToOneMux;

architecture behavior of twoToOneMux is
begin
process(A,B,address)
begin
   if(address = '0') then
      output <= A;
   elsif(address = '1') then
      output <= B;
   end if;
end process;
end behavior;
```