```
-- Arithmetic Logic Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- 8 bit operands and output

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;


-- decode op code, perform operation,
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;
```

```
-- Control Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ControlUnit is

port (
      --Op code used for instructions (NOT the ALU Op)
      OpCode : in std_logic_vector(2 downto 0);
      --Clock Signal
      clk : in std_logic;
      --Load bits to basically turn components on and off at a given
state
      ToALoad : out std_logic;
      ToMarLoad : out std_logic;
      ToIrLoad : out std_logic;
      ToMdriLoad : out std_logic;
      ToMdroLoad : out std_logic;
      ToPcIncrement : out std_logic := '0';
      ToMarMux : out std_logic;
      ToRamWriteEnable : out std_logic;
      --This is the ALU op code, look inside the ALU code to set this
      ToAluOp : out std_logic_vector (2 downto 0)
);

end;

architecture behavior of ControlUnit is
--Custom Data Type to Define Each State
type cu_state_type is (load_mar, read_mem, load_mdri, load_ir, decode,
                                 ldaa_load_mar, ldaa_read_mem,
ldaa_load_mdri, ldaa_load_a,
                                 adaa_load_mar, adaa_read_mem,
adaa_load_mdri, adaa_store_load_a,
                                 staa_load_mdro, staa_write_mem,
                                 increment_pc);

--Signal to hold current state
signal current_state : cu_state_type;

begin
--Defines the transitions in our state machine
```

```
process(clk)
begin

if (clk'event and clk = '1') then
     case current_state is
           --Increment the pc and fetch the instruction, then load the
IR with the fetched instruction
           --Decode the instruction, use the diagram in the handout to
determine the next states
           when increment_pc =>
                current_state <= load_mar;
           when load_mar =>
           --INSERT CODE HERE




           --Decode Opcode to determine Instruction
           --Assign current state based on the opCode
           when decode =>
           --INSERT CODE HERE


           --Instructions, need to determine the next state to
implement each instruction
           --Follow the path to perform each instruction as described
in the handout, and determine
           --Where the state machine needs to go to implement the
instruction
           ---Load instruction
           when ldaa_load_mar =>
                current_state <= ldaa_read_mem;
                --INSERT CODE HERE

           --Add Instruction
           when adaa_load_mar =>
                current_state <= adaa_read_mem;
                --INSERT CODE HERE

           --Store Instruction
           when staa_load_mdro =>
                --INSERT CODE HERE

     end case;
end if;
end process;
-- Defines what happens at each state, set to '1' if we want that
component to be on
-- Set Op Code accordingly based on ALU, different from the
instruction op code, look at the actual ALU code
-- Keep in mind when ToMarMux = 0 , MAR is loaded from PC address,
when ToMarMux = 1, MAR is loaded with IR address

process(current_state)
begin

      ToALoad <= '0';
      ToMdroLoad <= '0';
      ToAluOp <= "000";

      case current_state is
            --Turns on the increment pc bit
            when increment_pc =>
                  ToALoad <= '0';
                  ToPcIncrement <= '1';
                  ToMarMux <= '0';
                  ToMarLoad <= '0';
                  ToRamWriteEnable <= '0';
                  ToMdriLoad <= '0';
                  ToIrLoad <= '0';
                  ToMdroLoad <= '0';
                  ToAluOp <= "000";
            --Loads MAR with address from program counter
            when load_mar =>
            --INSERT CODE HERE

            --Reads Address located in MAR
            when read_mem =>
            --INSERT CODE HERE

            --Load Memory Data Register Input
            when load_mdri =>
            --INSERT CODE HERE

            --Loads the Instruction Register with instruction fetched
from Memory
            when load_ir =>
            --INSERT CODE HERE
            --Decodes The current instruction (everything should be off
for this)
            when decode =>
            --INSERT CODE HERE

            --Loads the MAR with address stored in IR
            when ldaa_load_mar =>
            --INSERT CODE HERE

            --Reads Data in memory retrieved from Address in MAR
            when ldaa_read_mem =>
            --INSERT CODE HERE

            --Loads the Memory data Register Input with data read from
memory
            when ldaa_load_mdri =>
            --INSERT CODE HERE

            --Loads the accumulator with data held in MDRI
            when ldaa_load_a =>
            --INSERT CODE HERE

            --Loads the MAR with address held in IR
            when adaa_load_mar =>
            --INSERT CODE HERE

            --Reads Memory based on address in MAR
            when adaa_read_mem =>
            --INSERT CODE HERE

            --Loads MDRI with data just read from memory
            when adaa_load_mdri =>
            --INSERT CODE HERE

            --Loads accumulator with data in MDRI
            when adaa_store_load_a =>
            --INSERT CODE HERE

            --Loads MDRO with data to be written to memory (this data
comes from the accumulator)
            when staa_load_mdro =>
            --INSERT CODE HERE
```

```
        --Writes to memory the data stored in MDRO
        when staa_write_mem =>
        --INSERT CODE HERE

    end case;
end process;
end behavior;
```

```
-- 8 By 32 Memory Array
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity memory_8_by_32 is

Port(
     clk:       in std_logic;
     Write_Enable: in std_logic;
     Read_Addr: in std_logic_vector   (4 downto 0);
     Data_in:   in std_logic_vector   (7 downto 0);
     Data_out:  out std_logic_vector(7 downto 0));
     end memory_8_by_32;

     architecture behavior of memory_8_by_32 is
     type ram_type is array(0 to 31) of std_logic_vector(7 downto 0);
     --instructions / data go into memory here
     signal Z:
ram_type:=("00000101","00100011","01000111","00000111","00101000","000
00110","00010100","00001101","00000001","10110100","10001010","1010101
0","10101001","00000000","10100101","01010101","10101110","10110100","
10001010","10101010","10101001","00000000","10100101","01010101","1010
1110","10110100","10001010","10101010","10101001","00000000","10100101
","01010101");
     Begin
     Process(clk,Read_Addr, Data_in, Write_Enable)
     Begin
     --Read from memory
     if(clk'event and clk='1' and Write_Enable='0') then
     Data_out<=Z(conv_integer(Read_Addr));
     --Write to Memory
     elsif(clk'event and clk='1' and Write_Enable='1') then
     Z(conv_integer(Read_Addr))<=Data_in;
     end if;
     end process;
     end;
```

```
--Program Counter Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--Increments the program counter by 1 if there is a positive edge
clock and increment =1
entity ProgramCounter is
port (
     output : out std_logic_vector(7 downto 0);
     clk : in std_logic;
     increment : in std_logic
);
end;

architecture behavior of ProgramCounter is
begin

process(clk,increment)
--Define a counter variable as an integer and initialize it to 0 (use
variable counter: integer:=) and fill in the value
--INSERT CODE HERE

begin
     --Create an if statement to check for the condition of a positive
edge clock and increment =1
     if (clk'event and clk = '1' and increment = '1') then
          --Increment counter variable by 1
          --INSERT CODE HERE

          --Output the counter variable as a std logic vector of 8
bits,
          --Use function conv_std_logic_vector(counter,8)
          --INSERT CODE HERE
     end if;
end process;
end behavior;
```

```
--Register component for CPU
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

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

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;
```

```
--Seven Segment Display, keep in mind the output here is 8 bits to
match
--The CPU component outputs, when connecting the pins, ignore the MSB
of o or o(7)
library ieee;
use ieee.std_logic_1164.all;

entity sevenseg is
port(
     i : in std_logic_vector(3 downto 0);
     o : out std_logic_vector(7 downto 0)
);
end sevenseg;

architecture logic of sevenseg is
begin
o <= "00000001" when i="0000" else
     "01001111" when i="0001" else
     "00010010" when i="0010" else
     "00000110" when i="0011" else
     "01001100" when i="0100" else
     "00100100" when i="0101" else
     "00100000" when i="0110" else
     "00001111" when i="0111" else
     "00000000" when i="1000" else
     "00000100" when i="1001" else
     "00001000" when i="1010" else
     "01100000" when i="1011" else
     "00110001" when i="1100" else
     "01000010" when i="1101" else
     "00110000" when i="1110" else
     "00111000" when i="1111";
end;
```

```
--Simple CPU template, This is the top level entity in your project
library ieee;
use ieee.std_logic_1164.all;

entity SimpleCPU_Template is
--These are the Outputs that can be displayed on the FPGA, More port
statements may be necessary,
--Depending on how you want to display each signal to the FPGA
port (
     clk : in std_logic;
     pcOut : out std_logic_vector(7 downto 0);
     marOut : out std_logic_vector (7 downto 0);
     irOutput : out std_logic_vector (7 downto 0);
     mdriOutput : out std_logic_vector (7 downto 0);
     mdroOutput : out std_logic_vector (7 downto 0);
     aOut : out std_logic_vector (7 downto 0);
     incrementOut : out std_logic
);

end;

architecture behavior of SimpleCPU_Template is
--Initialize our memory component
component memory_8_by_32
port( clk:       in std_logic;
     Write_Enable: in std_logic;
     Read_Addr: in std_logic_vector   (4 downto 0);
     Data_in:   in std_logic_vector   (7 downto 0);
     Data_out:  out std_logic_vector(7 downto 0)
);
end component;
--initialize the alu
component aluport (

     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 component;
--initialize the registers
component reg
port (
     input : in std_logic_vector      (7 downto 0);
     output : out std_logic_vector    (7 downto 0);
     clk : in std_logic;
     load : in std_logic
);
end component;
--initialize the program counter
component ProgramCounter
port (
     increment : in std_logic;
     clk : in std_logic;
     output : out std_logic_vector    (7 downto 0)
);
end component;
--initialize the mux
component TwoToOneMux
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 component;
--initialize the seven segment decoder
component sevenseg
port(
     i : in std_logic_vector(3 downto 0);
     o : out std_logic_vector(7 downto 0)
);
end component;

-- initialize control unit
component ControlUnit
port (
     OpCode : in std_logic_vector(2 downto 0);
     clk : in std_logic;
     ToALoad : out std_logic;
     ToMarLoad : out std_logic;
     ToIrLoad : out std_logic;
     ToMdriLoad : out std_logic;
     ToMdroLoad : out std_logic;
     ToPcIncrement : out std_logic;
     ToMarMux : out std_logic;
     ToRamWriteEnable : out std_logic;
     ToAluOp : out std_logic_vector (2 downto 0)
```

```vhdl
);
end component;

--The following signals will be used in your port map statements,
don't use the port variables in your port maps

-- Connections : Need to be sorted
signal ramDataOutToMdri : std_logic_vector (7 downto 0);

-- MAR Multiplexer connections
signal pcToMarMux : std_logic_vector(7 downto 0);
signal muxToMar : std_logic_vector    (7 downto 0);

-- RAM connections
signal marToRamReadAddr : std_logic_vector  (4 downto 0);
signal mdroToRamDataIn : std_logic_vector (7 downto 0);

-- MDRI connections
signal mdriOut : std_logic_vector     (7 downto 0);

-- IR connection
signal irOut : std_logic_vector       (7 downto 0);

-- ALU / Accumulator connections
signal aluOut: std_logic_vector  (7 downto 0);
signal aToAluB : std_logic_vector     (7 downto 0);

-- Control Unit connections
signal cuToALoad : std_logic;
signal cuToMarLoad : std_logic;
signal cuToIrLoad : std_logic;
signal cuToMdriLoad : std_logic;
signal cuToMdroLoad : std_logic;
signal cuToPcIncrement : std_logic;
signal cuToMarMux : std_logic;
signal cuToRamWriteEnable : std_logic;
signal cuToAluOp : std_logic_vector (2 downto 0);
begin
```

```
--PORT MAP STATEMENTS GO HERE
-- Create port map statements for each component in the CPU and map
them to the appropriate signal defined above
-- RAM
--INSERT CODE HERE

-- Accumulator
--INSERT CODE HERE

-- ALU
--INSERT CODE HERE

-- Program Counter
--INSERT CODE HERE

-- Instruction Register
--INSERT CODE HERE

-- MAR mux
--INSERT CODE HERE


-- Memory Access Register
--INSERT CODE HERE

-- Memory Data Register Input
--INSERT CODE HERE

-- Memory Data Register Output
--INSERT CODE HERE

-- Control Unit
--INSERT CODE HERE

--REMAINING CODE GOES HERE
--Here is where you connect the port statement to the matching signal
to display it on the FPGA
--If you want to display the signal on LED's, just set it to the port
statement port<=signal;
--If you want to send the signal to the seven segment display,
initialize an instance of the sevenseg
--Then map i=>signal, o=>port , keep in mind i needs to be 4 bits and
o 8 bits
--pcOut <= pcToMarMux;
```

```
end behavior;
```

```vhdl
--Mux used to create a shared connection between PC and IR to the MAR
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;

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;
```