Question 1

a

Program Counter

Counts the next address that accesses the memory. PI signal increments this value and PL concatenates the values that were in DR and SB and uses that as the next address.

Memory

This contains the addresses of instructions that are stores in the Control Memory. For example, instruction 0x002E may correspond to the memory address in Control Memory of the instruction ADI. The address accessed is determined by the program counter. If the signal MW (memory write) is set, input from data in is stored there.

Instruction Register

When an instruction is passed into here, the first 7 bits correspond to the instruction to be accessed in Control memory. The last 9 bits correspond to which registers are to be accessed - Destinarion Register, Set A and Set B.

Control Access Register

This determines what the next address the Control Memory uses to access the next instruction. In the case that its relevant to know what the control flags are, this could cause the next instruction to be executed instead.

Control Memory

The control memory accesses the instruction to be executed based on the instruction address passed in. An instruction could take more than one pass of the system (e.g. Load). Depending on the instruction, the control memory will send out signals that involve which multiplexers to be set, a possible next address for multi cycle instructions, instructions to the Function Unit, etc.

Register File

This contains an array of registers which holds data. A temporary register is also held here for multi cycle instructions so that the data the user may be storing in the registers is not disrupted.

Functional Unit

The function unit contains the ALU and a barrel shifter. The ALU performs operations such as addition, subtraction, etc. The barrel shifter can shift data left or right by any amount of bits.

b

PC

- Signals in: load, increment
- Data in: dr, sb
- Data out: address

```
entity pc is
    Port(dr: in std_logic_vector(2 downto 0);
         sb: in std_logic_vector(2 downto 0);
         load: in std_logic;
         increment: in std_logic;
         clk: in std_logic;
         pc_out: in std_logic_vector(15 downto 0)
        );
end pc;

architecture Behavioral of pc is

begin
    pc_process: process(load, increment, clk)
    variable count: std_logic_vector(15 downto 0) := X"0000";
    variable extend: std_logic_vector(5 downto 0) := "000000";

    begin
        if(rising_edge(clk)) then
            if increment='1' then
                count := std_logic_vector(unsigned(count)+1);
            elsif load='1' then
                extend := dr&sb;
                count := std_logic_vector(unsigned(count)+resize(unsigned(extend), 16));
            end if;
            pc_out <= count;
        end if;
    end process;
end Behavioral;
```

CAR

- Signal in: incr
- Data in: addr_in
- Data out: addr_out

```
entity car is
    Port(incr: in std_logic;
         addr_in: in std_logic_vector(7 downto 0);
         addr_out: out std_logic_vector(7 downto 0)
        );
end car;

architecture Behavioral of car is

begin
    addr_out <=
        std_logic_vector(unsigned(addr_in)+1) when incr='0' else
        addr_in when incr='1' else
        "00000000";
end Behavioral;
```

Question 2

a

```
      10111
    x 10011
    -------
      10111
     101110
    0000000
   00000000
+ 101110000
-----------
  110110101
```

b

```
entity binary_multiplier is
    Port(clk, reset, loadb, loadq: in std_logic;
         mult_in: in std_logic_vector(3 downto 0);
         mult_out: in std_logic_vector(3 downto 0);
        );
end binary_multiplier;

architecture Behavioral of binary_multiplier is
    type state_type is (IDLE, MUL0, MUL1);
    signal state, next_state: state_type;
    signal A, B, Q: std_logic_vector(3 downto 0);
    signal P: std_logic_vector(1 downto 0);
    signal C, Z: std_logic;

begin
    Z <= P(1) nor P(0);
    MULT_OUT <= A & Q;

    get_state: process(clk, reset)
    begin
        if(reset='1') then
            state<=IDLE;
        elsif(rising_edge(clk) then
            state<=next_state;
        end if;
    end process;

    get_next_state: process(G, Z, state)
    begin
        case state is
            when IDLE=>
                if G='1' then
                    next_state<=MUL0;
                else
                    next_state<=IDLE;
                end if;
            when MUL0=>
                next_state<=MUL1;
            when MUL1=>
                if Z='1' then
                    next_state<=IDLE;
                else
                    next_state<=MUL0;
                end if;
        end case;
    end process;

    state_process: process(clk)
    variable CA: std_logic_vector(4 downto 0);    -- Accounts for carry
    begin
        if(rising_edge(clk)) then
            if loadb='1' then
                b<=mult_in;
            end if;
            if loadq='1' then
                q<=mult_in;
            end if;

            case state is
                when IDLE=>
                    if G='1' then
                        C<='0';
                        A<="0000";
                        P<="11";
                    end if;
                when MUL0=>
                    if Q(0)='1' then
                        CA := ('0' & A) + ('0' & B);    -- Carry will be added
                    else
                        CA := C&A;
                    end if;
                    C<=CA(4);
                    A<=CA(3 downto 0);
                when MUL1=>
                    -- C & A & Q >> 1
                    C<='0';
                    A<=C & A(3 downto 1);    -- C=its state at the end of MUL0
                    Q<=A(0) & Q(3 downto 1);
                    P<=P-"01";
            end case;
        end if;
    end process;
end Behavioral;
```

Question 3

2-line Multiplexer

```
entity mux2 is
    Port(sel: in std_logic;
         ln0, ln1: in std_logic_vector(3 downto 0);
         out: out std_logic_vector(3 downto 0)
        );
end mux2;

architecture Behavioral in mux2 is

begin
    out <=
        ln0 after 5 ns when sel='0' else
        ln1 after 5 ns when sel='1' else
        "0000" after 5 ns;
end Behavioral;
```

4-line Multiplexer

```
entity mux4 is
    Port(sel0, sel1: in std_logic;
         ln0, ln1, ln2, ln3: in std_logic_vector(3 downto 0);
         out: out std_logic_vector(3 downto 0)
        );
end mux4;

architecture Behavioral in mux4 is

begin
    out <=
        ln0 after 5 ns when sel0='0' and sel1='0' else
        ln1 after 5 ns when sel0='1' and sel1='0' else
        ln2 after 5 ns when sel0='0' and sel1='1' else
        ln3 after 5 ns when sel0='1' and sel1='1' else
        "0000" after 5 ns;
end Behavioral;
```

2-to-4 Decoder

```
entity decoder2to4 is
    Port(sel0, sel1: in std_logic;
         out0, out1, out2, out3: out std_logic
        );
end decoder2to4

architecture Behavioral in decoder2to4 is

begin
    out0<=(not sel0) and (not sel1) after 5 ns;
    out1<=sel0 and (not sel1) after 5 ns;
    out2<=(not sel0) and sel1 after 5 ns;
    out3<=sel0 and sel1 after 5 ns;
end Behavioral;
```

4-bit Register

```
entity register4bit is
    Port(load, clk: in std_logic;
         data: in std_logic_vector(3 downto 0);
         store: out std_logic_vector(3 downto 0)
        );
end register4bit;

architecture Behavioral in register4bit is

begin
process(clk)
    begin
        if(rising_edge(clk)) then
            if load='1' then
                store<=data after 5 ns;
            end if;
        end if;
    end process;
end Behavioral;
```

Register File

```
entity register_file is
    Port(src_s0, src_s1, des_a0, des_a1, data_src, clk: in std_logic;
         data: in std_logic_vector(3 downto 0);
         reg0, reg1, reg2, reg3: out std_logic_vector(3 downto 0)
        );
end register_file;

architecture Behavioral in register_file is

-- components
    component mux2
        Port(sel: in std_logic;
             ln0, ln1: in std_logic_vector(3 downto 0);
             out: out std_logic_vector(3 downto 0)
            );
    end component;

    component mux4
        Port(sel0, sel1: in std_logic;
             ln0, ln1, ln2, ln3: in std_logic_vector(3 downto 0);
             out: out std_logic_vector(3 downto 0)
            );
    end component;

    component decoder2to4
        Port(load, clk: in std_logic;
             data: in std_logic_vector(3 downto 0);
             store: out std_logic_vector(3 downto 0)
            );
    end component;

    component reg4bit
        Port(load, clk: in std_logic;
             data: in std_logic_vector(3 downto 0);
             store: out std_logic_vector(3 downto 0)
            );
    end component;

-- signals
signal load_reg0, load_reg1, load_red2, load_reg3: std_logic;
signal mux2_out, mux4_out, out_reg0, out_reg1, out_reg2,
       out_reg3: std_logic_vector(3 downto 0);

begin
    mux_a: mux2 PORT MAP(sel => data_src,
                         ln0 => data,
                         ln1 => mux4_out,
                         out => mux2_out
                        );
```

```
mux_b: mux4 PORT MAP(sel0 => src_s0,
                     sel1 => src_s1,
                     ln0 => out_reg0,
                     ln1 => out_reg1,
                     ln2 => out_reg2,
                     ln3 => out_reg3,
                     out => mux4_out
                    );

des_decoder: decoder2to4 PORT MAP(sel0 => des_A0,
                                  sel1 => des_A1,
                                  out0 => load_reg0,
                                  out1 => load_reg1,
                                  out2 => load_reg2,
                                  out3 => load_reg3
                                 );

reg00: reg4bit PORT MAP(load => load_reg0,
                        clk => clk,
                        data => mux2_out,
                        store => out_reg0,
                       );

reg01: reg4bit PORT MAP(load => load_reg1,
                        clk => clk,
                        data => mux2_out,
                        store => out_reg1,
                       );

reg02: reg4bit PORT MAP(load => load_reg2,
                        clk => clk,
                        data => mux2_out,
                        store => out_reg2,
                       );

reg03: reg4bit PORT MAP(load => load_reg3,
                        clk => clk,
                        data => mux2_out,
                        store => out_reg3,
                       );

reg0 <= out_reg0;
reg1 <= out_reg1;
reg2 <= out_reg2;
reg3 <= out_reg3;
```

```
end Behavioral;
```

b

- Add an extra register to the register file as a temporary register.
- Change the 4 line multiplexer to a 5 line multiplexer.
- Change the 2-to-4 decoder to a 3-to-5 decoder.
- Connect the 2 line multiplexer to the other register.
- Add another 3-to-5 decoder.
- Each decoder outputs to bus A and bus B.