Question 1

a

- Change state boxes to D flip flops. This controls which state the binary multiplier is in and outputs a 1.
- Decision boxes are demultiplexers. The decision is made with the decision signal and outputs the state of the state box. If it is the current state, a one is outputted.
- Junctions become OR gates. If there is a result to be passed into the state box, this will allow it.
- Add output signals. This performs the operations.

b

Address	NXTADD1	NXTADD0	SEL	DATAPATH
IDLE	INIT	IDLE	G	N/A
INIT	N/A	MUL0	NXT	IT,CC
MUL0	ADD	MUL1	Q	N/A
ADD	N/A	MUL1	NXT	LD
MUL1	IDLE	MUL0	Z	CC,SD

Address	NXTADD1	NXTADD0	SEL	DATAPATH
000	001	000	01	0000
001	000	010	00	1010
010	011	100	10	0000
011	000	100	00	0010
100	000	010	11	1100

- IDLE: 000001000010000
- INIT: 001000010001010
- MUL0: 010011100100000
- ADD: 011000100000010
- MUL1: 100000010111100

Question 2

a

Memory

Memory is accessed by an address passed in from the program counter or loaded from bus A. It accesses instructions stored there, or can be written to with data passed in from bus B. Instructions are stored like:

Opcode	DA	SA	SB
7-bit	3-bit	3-bit	3-bit

Control Memory

Every instruction needs various control signals to be executed, and some can take several clock cycles to be executed. This corresponds to each instruction's microprogram. Microprograms consist of one or more control words which sets the state of the processor for that particular clock cycle. Using the opcode that is loaded from memory, this corresponds to the memory address of the first part of that instruction's microprogram in Control Memory.

Registers

Program Counter

This contains the address of main memory that is currently being accessed. The PC is calculated as follows:

State	PL	PI
Unchanged	0	0
Increment	0	1
Offset	1	0

Instruction Register

This holds the current instruction being performed.

State	IL
Unchanged	0
Loaded	1

Control Address Register

This holds the address of the control word in control memory currently being performed.

State	Mux S Out
Increment	0
Load from Mux C	1

Processor Operation

Fetch

This microprogram will fetch the next instruction from memory.

1. Increment the PC and load the instructions from memory.
2. Pass into the IR
3. Write opcode from the instruction into the CAR

This would be stored in memory like:

Address
0x10
0x11
0x12

- This could be used as an entire instruction in itself, as in, in memory the opcode part of the instruction is: 0010000.
- It makes more sense to fetch the next instruction after the end of the previous instruction. This can then be put in the control signal Next Address (NA) in the control word.

Branching

In the case of a branch, the PC is offset by the value that is the concatenation of DR and SB, and PL would be set. In the case of a conditional branch, the flag is checked by loading the value in from Mux S into the CAR. If the value from the flag is 0, the CAR in incremented, otherwise it loads the unconditional branch microprogram.

b

1. The instruction is fetched.
    1. The PC is incremented.

    2. This is passed into the IR.
    3. The opcode is passed to the CAR and the three registers (desination, register A and register B) are selected.
2. The microprogram to execute the subtraction instruction begins.
    1. The values of registers A and B are loaded into the functional unit.
    2. FS is 00100 which corresponds to A+(¬B)==A-B and to set the multiplexer that selects the Arithmetic Circuit in the ALU.
    3. Save the result specified in DR into the register and fetch next instruction.
3. When the next instruction is fetched (BNE), Mux S is set to 0101 and te N flag is loaded into the CAR.
    1. If N=0 then increment the CAR to continue to the next instruction of the microprogram, ie. fetch next instruction.
    2. If N=1 then go to the memory address that stores the microprogram for unconditional branching would be loaded.
        1. The PL flag would be set and the next memory address to fetch from would be determined by DR and SB.
        2. The fetch next instruction microprogram would begin.

Question 3

b

3-line multiplexer

- 1-bit inputs: Values(3), Selects(2)
- 1-bit output: Result

```
entity mux3 is
    Port(data0, data1, data2: in std_logic;
         sel0, sel1: in std_logic;
         result: out std_logic
        );
end mux3;

architecture Behavioral of mux3 is

begin
result <= data0 after 5 ns when sel0='0' and sel1='0' else
          data1 after 5 ns when sel0='1' and sel1='0' else
          data2 after 5 ns when sel0='0' and sel1='1' else
          '0' after 5 ns;
end Behavioral;
```

Barrel Shifter

- 4-bit input: Value
- 2-bit input: SL or SR
- 4-bit output: Result

```
entity barrel_shifter is
    Port(data_in: in std_logic_vector(3 downto 0);
         shift: in std_logic_vector(2 downto 0);       -- 0=sr, 1=sl
         data_out: out std_logic_vector(3 downto 0)
        );
end barrel_shifter;

architecture Behavioral of barrel_shifter is

component mux3
    Port(data0, data1, data2: in std_logic;
         sel0, sel1: in std_logic;
         result: out std_logic
        );
end component;

begin

    mux01: mux3 PORT MAP(data0 => data_in(0),
                         data1 => data_in(1),
                         data2 => '0',
                         sel0 => shift(0),
                         sel1 => shift(1),
                         result => data_out(0)
                        );

    mux02: mux3 PORT MAP(data0 => data_in(1),
                         data1 => data_in(2),
                         data3 => data_in(3),
                         sel0 => shift(0),
                         sel1 => shift(1),
                         result => data_out(1)
                        );

    mux03: mux3 PORT MAP(data0 => data_in(2),
                         data1 => data_in(3),
                         data2 => data_in(1),
                         sel0 => shift(0),
                         sel1 => shift(1),
                         result => data_out(2)
```

```
                        );

    mux04: mux3 PORT MAP(data0 => data_in(3),
                         data1 => '0',
                         data2 => data_in(2),
                         sel0 => shift(0),
                         sel1 => shift(1),
                         result => data_out(3)
                        );
end Behavioral;
```

Question 4

a

2-line Multiplexer

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

architecture Behavioural of mux2 is
begin
z <= ln0 after 5 ns when s='0' else
     ln1 after 5 ns when s='1' else
     "0000" after 5 ns;
end Behavioral;
```

4-line Multiplexer

```
entity mux4 is
    Port(s0, s1: in std_logic;
         ln0, ln1, ln2, ln3: in std_logic_vector(3 downto 0);
         z: out std_logic_vector(3 downto 0)
        );
end mux4;

architecture Behavioral of mux4 is
begin
z <= ln0 after 5 ns when s0='0' and s1='0' else
```

```
     ln1 after 5 ns when s0='1' and s1='0' else
     ln2 after 5 ns when s0='0' and s1='1' else
     ln3 after 5 ns when s0='1' and s1='1' else
     "0000" after 5 ns;
end Behavioral;
```

2-to-4 Decoder

```
entity decoder2to4 is
    Port(a0, a1: in std_logic;
         q0, q1, q2, q3: out std_logic
        );
end decoder2to4;

architecture Behavioral of decoder2to4 is
begin
q0 <= (not a0) and (not a1) after 5 ns;
q1 <= a0 and (not a1) after 5 ns;
q2 <= (not a0) and a1 after 5 ns;
q3 <= a0 and a1 after 5 ns;
end Behavioural;
```

Register

```
entity register is
    Port(load, clk: in std_logic;
         reg_in: in std_logic_vector(3 downto 0);
         reg_out: out std_logic_vector(3 downto 0);
        );
end register;

architecture Behavioral of register is
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            if load='1' then
                reg_out <= reg_in 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: in std_logic;
         clock, data_src: in std_logic;
         data: in std_logic_vector(3 downto 0);
         reg_out0, reg_out1: out std_logic_vector(3 downto 0);
         reg_out2, reg_out3: out std_logic_vector(3 downto 0)
        );
end register_file;

architecture Behavioral of register_file is

component mux2
    Port(s: in std_logic;
         ln0, ln1: in std_logic_vector(3 downto 0);
         z: out std_logic_vector(3 downto 0)
        );
end component;

component mux4
    Port(s0, s1: in std_logic;
         ln0, ln1, ln2, ln3: in std_logic_vector(3 downto 0);
         z: out std_logic_vector(3 downto 0)
        );
end component;

component decoder2to4
    Port(a0, a1: in std_logic;
         q0, q1, q2, q3: out std_logic
        );
end component;

component register
    Port(load, clk: in std_logic;
         reg_in: in std_logic_vector(3 downto 0);
         reg_out: out std_logic_vector(3 downto 0);
        );
end component;

signal dec0, dec1, dec2, dec3: std_logic;
signal mux2_out, mux4_out: std_logic_vector(3 downto 0);
signal reg_sig0, reg_sig1, reg_sig2, reg_sig3: std_logic_vector(3 downto 0);

begin
```

```
muxA: mux2 PORT MAP(s => data_src,
                    ln0 => data,
                    ln1 => mux4_out,
                    z => mux2_out
                   );

muxB: mux4 PORT MAP(s0 => src_s0,
                    s1 => src_s1,
                    ln0 => reg_sig0,
                    ln1 => reg_sig1,
                    ln2 => reg_sig2,
                    ln3 => reg_sig3,
                    z => mux4_out
                   );

dec: decoder2to4 PORT MAP(a0 => des_a0,
                          a1 => des_a1,
                          q0 => dec0,
                          q1 => dec1,
                          q2 => dec2,
                          q3 => dec3
                         );

reg0: register PORT MAP(load => dec0,
                        clk => clock,
                        reg_in => mux2_out,
                        reg_out => reg_sig0
                       );

reg1: register PORT MAP(load => dec1,
                        clk => clock,
                        reg_in => mux2_out,
                        reg_out => reg_sig1
                       );

reg2: register PORT MAP(load => dec2,
                        clk => clock,
                        reg_in => mux2_out,
                        reg_out => reg_sig2
                       );

reg3: register PORT MAP(load => dec3,
                        clk => clock,
                        reg_in => mux2_out,
                        reg_out => reg_sig3
```

```
                    );

reg_out0 <= reg_sig0;
reg_out1 <= reg_sig1;
reg_out2 <= reg_sig2;
reg_out3 <= reg_sig3;

end Behavioral;
```

b

```
entity test_register_file is
end test_register_file;

architecture Behavioral of test_register_file is

component register_file
    Port(src_s0, src_s1, des_a0, des_a1: in std_logic;
         clock, data_src: in std_logic;
         data: in std_logic_vector(3 downto 0);
         reg_out0, reg_out1: out std_logic_vector(3 downto 0);
         reg_out2, reg_out3: out std_logic_vector(3 downto 0)
        );
end component;

signal src_s0, src_s1, clock: std_logic := '0';
signal des_a0, des_a1: std_logic := '0';
signal data: std_logic_vector(3 downto 0) := (others => '0');
signal reg_out0, reg_out1: std_logic_vector(3 downto 0) := (others => '0');
signal reg_out2, reg_out3: std_logic_vector(3 downto 0) := (others => '0');

constant clock_period: time := 10 ns;

begin
    uut: register_file PORT MAP(clock => clk,
                                src_s0 => src_s0,
                                src_s1 => src_s1,
                                des_a0 => des_a0,
                                des_a1 => des_a1,
                                data => data,
                                reg_out0 => reg_out0,
                                reg_out1 => reg_out1,
                                reg_out2 => reg_out2,
                                reg_out3 => reg_out3
                               );
```

```
    clk_process: process
    begin
        clk <= '0';
        wait for clock_period/2;
        clk <= '1';
        wait for clock_period/2;
    end process;

    reg_process: process
    begin
        src_s0 <= '0';
        src_s1 <= '0';
        des_a0 <= '0';
        des_a1 <= '0';
        data_src <= '0';
        data <= "0001";
        wait for clock_period;    -- reg0=0001
        des_a0 <= '1';
        data_src <= '1';
        wait for clock_period;    -- reg1=reg0=0001
        data_src <= '0';
        data <= "1100";
        wait for clock_period;    -- reg1=1100
        src_s0 <= '1';
        src_s1 <= '1';
        wait for clock_period;    -- mux4=0000
        data_src <= '1';
        des_a0 <= '0';
        des_a1 <= '1';
        wait for clock_period;    -- reg0=reg4=0000
    end process;

end Behavioral;
```