1. Mano and Kime 6-13. Specify the size of a ROM (number of words and number of
bits per word) that will accommodate the truth table for the following combinational
circuit components: (a) an 8 bit adder-subtractor with $C_{in}$ and $C_{out}$; (b) A binary
multiplier that multiplies two 8-bit numbers; (c) a code converter from a 4-digit BCD
number to a binary number.

(a) This circuit has 16 data inputs, a mode input and a carry input, so the ROM must
have $2^{18}$ words. It also has 8 data outputs and a carry output, so it must have 9 bits per
word.

(b) This circuit has two eight bit inputs and 16 outputs, so the ROM must have $2^{16}$
words with 16 bits each.

(c) This circuit has 16 inputs and 14 outputs, so the ROM must have $2^{16}$ words with
14 bits each.

2. Show how the sequential circuit shown below can be mapped onto a single
configurable logic block. Do this by specifying the functions that the LUTs implement
and showing the paths through the multiplexors that must be configured to provide the
required connections (show this by drawing heavy lines on a photocopy of page 6-21
of the notes). Also show what values must be stored in each location of the LUTs to
implement the required logic functions.

![Diagram of the sequential circuit]
For the top lookup table, the stored values are \((0,0,1,1,0,0,1,1,0,0,1,1,1,1,1,1)\) with the first value in the list going into location 0, etc. This is assuming that the top input to the LUT is the most significant address bit and that the bottom input is the least significant. The lower LUT contents are \((1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0)\). The middle LUT contents are \((0,0,0,0,0,0,1,1)\).

3. Extend the simple processor covered in section 7 of the notes to include a subtraction instruction. The instruction should be identified by an operation code of 1010 in the first four bits. The remaining 12 bits should include the address of a memory word. The value in this word is to be subtracted from the contents of the accumulator and the result saved in the accumulator.

Create a timing diagram similar to those on page 7-25 of the notes. Use this to guide you in the design process for your instruction.

You will find the VHDL source for the processor on the web site. Modify this to implement the subtraction instruction and verify that it works correctly by simulating the operation of the subtraction instruction (that is, initialize the RAM with a short program that uses the subtraction instruction and simulate the operation of the processor as it runs this program). Turn in copies of those VHDL modules you had to change and mark all the lines that were modified or added.
Note, the processor takes about 30 minutes to synthesize (results will vary depending on speed of your computer). Also, the synthesizer produces a number of warnings that you may ignore. When you simulate the circuit, you may get a message from the simulator complaining that multiple buffers are attempting to drive the same wire. You may ignore this as well.

\[
\text{subtract}
\]

\[
\begin{align*}
\text{clk} & \quad \text{mem\_END} \\
\text{mem\_rw} & \\
\text{pc\_ENA} & \\
\text{pc\_LD} & \\
\text{pc\_INC} & \\
\text{ir\_ENA} & \\
\text{ir\_END} & \\
\text{ir\_LD} & \\
\text{iar\_ENA} & \\
\text{iar\_LD} & \\
\text{acc\_END} & \\
\text{acc\_LD} & \\
\text{acc\_SA} & \\
\text{alu\_op} & 2 \times 2
\end{align*}
\]

The modified VHDL modules are shown below. The changes are shown in bold.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity instruction_register is
  port (  
    clk, en_A, en_D, ld, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0);
    load, store, add, neg, halt, branch: out STD_LOGIC;
  );
end entity instruction_register;
```
cbranch, iload, istore, dload, dadd, sub: out STD_LOGIC
end instruction_register;

architecture irArch of instruction_register is
signal irReg: STD_LOGIC_VECTOR(15 downto 0);
begin
process(clk) begin
if clk'event and clk = '0' then -- load on falling edge
if reset = '1' then
irReg <= "0000000000000000";
elsif ld = '1' then
irReg <= dBus;
end if;
end if;
end process;

aBus <= "0000" & irReg(11 downto 0) when en_A = '1' else
"ZZZZZZZZZZZZZZZ";
dBus <= "0000" & irReg(11 downto 0) when en_D = '1' else
"ZZZZZZZZZZZZZZZ";

load <= '1' when irReg(15 downto 12) = "0000" else '0';
store <= '1' when irReg(15 downto 12) = "0001" else '0';
add <= '1' when irReg(15 downto 12) = "0010" else '0';
neg <= '1' when irReg = "0011" & "000000000000" else '0';
halt <= '1' when irReg = "0011" & "000000000001" else '0';
branch <= '1' when irReg(15 downto 12) = "0100" else '0';
cbranch <= '1' when irReg(15 downto 12) = "0101" else '0';
iload <= '1' when irReg(15 downto 12) = "0110" else '0';
istore <= '1' when irReg(15 downto 12) = "0111" else '0';
dload <= '1' when irReg(15 downto 12) = "1000" else '0';
dadd <= '1' when irReg(15 downto 12) = "1001" else '0';
sub <= '1' when irReg(15 downto 12) = "1010" else '0';
end irArch;

---------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity alu is
port ( op: in STD_LOGIC_VECTOR(1 downto 0);
accD: in STD_LOGIC_VECTOR(15 downto 0);
dBus: in STD_LOGIC_VECTOR(15 downto 0);
result: out STD_LOGIC_VECTOR(15 downto 0);
accZ: out STD_LOGIC
);
end alu;

architecture aluArch of alu is
begin
result <= (not accD) + "0000000000000000" when op = "00" else
accD + dBus when op = "01" else
accD - dBus when op = "10" else
"0000000000000000";
accZ <= not (accD(0) or accD(1) or accD(2) or accD(3) or
accD(4) or accD(5) or accD(6) or accD(7) or
accD(8) or accD(9) or accD(10) or accD(11) or
accD(12) or accD(13) or accD(14) or accD(15))
end aluArch;
---------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity ram is
  port ( 
    r_w, en, reset: in STD_LOGIC;
    aBus: in STD_LOGIC_VECTOR(15 downto 0);
    dBus: inout STD_LOGIC_VECTOR(15 downto 0)
  );
end ram;

architecture ramArch of ram is
  type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
  signal ram: ram_typ;
begin
  process(en, reset, r_w, aBus, dBus) begin
    if reset = '1' then
      ram(0) <= to_stdlogicvector(x"0003");
      ram(1) <= to_stdlogicvector(x"a004");
      ram(2) <= to_stdlogicvector(x"3001");
      ram(3) <= to_stdlogicvector(x"3456");
      ram(4) <= to_stdlogicvector(x"2345");
    elsif r_w = '0' then
      ram( conv_integer(unsigned(aBus))) <= dBus;
    end if;
  end process;
  dBus <= ram( conv_integer(unsigned(aBus)))
      when reset = '0' and en = '1' and r_w = '1' else
      "ZZZZZZZZZZZZZZZZZ"
end ramArch;

library IEEE;
use IEEE.std_logic_1164.all;

entity controller is
  port ( 
    clk, reset: in  STD_LOGIC;
    mem_enD, mem_rw: out  STD_LOGIC;
    pc_enA, pc_ld, pc_inc: out  STD_LOGIC;
    ir_enA, ir_enD, ir_ld: out  STD_LOGIC;
    ir_load, ir_store, ir_add: in  STD_LOGIC;
    ir_neg, ir_halt, ir_branch: in  STD_LOGIC;
    ir_cbranch, ir_iLoad: in  STD_LOGIC;
    ir_istore, ir_dload, ir_dadd: in  STD_LOGIC;
    ir_sub: in  STD_LOGIC;
    iar_enA, iar_ld: out  STD_LOGIC;
    acc_enD, acc_ld, acc_selAlu: out  STD_LOGIC;
    alu_accZ: in  STD_LOGIC;
    alu_op: out STD_LOGIC_VECTOR(1 downto 0)
  );
end controller;

architecture controllerArch of controller is
  type state_type is ( reset_state, 
    fetch0, fetch1, 
    load0, load1, 
    store0, store1, 
    add0, add1, 
    negate0, negate1, 
    halt, 
    branch0, branch1, 
    
    -- Specific states and transitions can be defined here
  );
cbranch0, cbranch1,
iload0, iload1, iload2, iload3,
istore0, istore1, istore2, istore3,
dload0, dload1,
dadd0, dadd1,
sub0, sub1
);
signal state: state_type;

begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then state <= reset_state;
      else
        case state is
          when reset_state => state <= fetch0;
          when fetch0 => state <= fetch1;
          when fetch1 =>
            if ir_load = '1' then state <= load0;
            elsif ir_store = '1' then state <= store0;
            elsif ir_add = '1' then state <= add0;
            elsif ir_neg = '1' then state <= negate0;
            elsif ir_halt = '1' then state <= halt;
            elsif ir_branch = '1' then state <= branch0;
            elsif ir_cbranch = '1' then state <= cbranch0;
            elsif ir_iload = '1' then state <= iload0;
            elsif ir_istore = '1' then state <= istore0;
            elsif ir_dload = '1' then state <= dload0;
            elsif ir_dadd = '1' then state <= dadd0;
            elsif ir_sub = '1' then state <= sub0;
            end if;
          when load0 => state <= load1;
          when load1 => state <= fetch0;
          when store0 => state <= store1;
          when store1 => state <= fetch0;
          when add0 => state <= add1;
          when add1 => state <= fetch0;
          when negate0 => state <= negate1;
          when negate1 => state <= fetch0;
          when halt => state <= halt;
          when branch0 => state <= branch1;
          when branch1 => state <= fetch0;
          when cbranch0 => state <= cbranch1;
          when cbranch1 => state <= fetch0;
          when iload0 => state <= iload1;
          when iload1 => state <= iload2;
          when iload2 => state <= iload3;
          when iload3 => state <= fetch0;
          when istore0 => state <= istore1;
          when istore1 => state <= istore2;
          when istore2 => state <= istore3;
          when istore3 => state <= fetch0;
          when dload0 => state <= dload1;
          when dload1 => state <= fetch0;
          when dadd0 => state <= dadd1;
          when dadd1 => state <= fetch0;
        end case;
      end if;
    end if;
  end process;
end
when sub0 => state <= sub1;
when sub1 => state <= fetch0;
when others => state <= halt;
end case;

end if;
end if;
end process;

process(clk) begin -- special process for memory write timing
if clk'event and clk = '0' then
if state = store0 or state = istore2 then
mem_rw <= '0';
else
mem_rw <= '1';
end if;
end if;
end process;

mem_enD <= '1' when state = fetch0 or state = fetch1 or state = load0 or state = load1 or state = add0 or state = add1 or state = iload0 or state = iload1 or state = iload2 or state = iload3 or state = istore0 or state = istore1 or state = sub0 or state = sub1
else '0';
pc_enA <= '1' when state = fetch0 or state = fetch1 else '0';
pc_ld <= '1' when state = branch0 or (state = cbranch0 and alu_accZ = '1') else '0';
pc_inc <= '1' when state = fetch1 else '0';
ir_enA <= '1' when state = load0 or state = load1 or state = store0 or state = store1 or state = add0 or state = add1 or state = iload0 or state = iload1 or state = iload2 or state = iload3 or state = istore0 or state = istore1 or state = sub0 or state = sub1
else '0';
ir_enD <= '1' when state = branch0 or state = cbranch0 or state = dload0 or state = dadd0 or state = dadd1 else '0';
ir_ld <= '1' when state = fetch1 else '0';
iar_enA <= '1' when state = iload2 or state = iload3 or state = istore2 or state = istore3 else '0';
iar_ld <= '1' when state = iload1 or state = istore1 else '0';
acc_enD <= '1' when state = store0 or state = store1 or state = istore2 or state = istore3 else '0';
acc_ld <= '1' when state = load1 or state = add1 or state = negatel or state = iload3 or state = dload0 or state = dadd0 or state = dadd1 or state = sub1
else '0';
acc_selAlu <= '1' when state = add1 or state = negatel or state = dadd1 or state = sub1 else '0';
alu_op <= "01" when state = add0 or state = add1 or state = dadd0 or state = dadd1 else "10" when state = sub0 or state = sub1 else
"00";
end controllerArch;

library IEEE;
use IEEE.std_logic_1164.all;

datatype top_level is
  port (clk, reset: in  STD_LOGIC;
  abusX: out STD_LOGIC_VECTOR(15 downto 0);
  dbusX: out STD_LOGIC_VECTOR(15 downto 0);
  mem_enDX, mem_rwX: out STD_LOGIC;
  pc_enAX, pc_ldX, pc_incX: out STD_LOGIC;
  ir_enAX, ir_enDX, ir_ldX: out STD_LOGIC;
  iar_enAX, iar_ldX: out STD_LOGIC;
  acc_enDX, acc_ldX, acc_selAluX: out STD_LOGIC;
  acc_QX: out STD_LOGIC_VECTOR(15 downto 0);
  alu_accZX: out STD_LOGIC;
  alu_opX: out STD_LOGIC_VECTOR(1 downto 0));
end top_level;

architecture topArch of top_level is
  -- component definitions omitted for brevity
  begin
  pc: program_counter port map( clk, pc_enA, pc_ld, pc_inc, reset, abus, dbus);
  ir: instruction_register port map( clk, ir_enA, ir_enD, ir_ld, reset, abus, dbus,
  ir_load, ir_store, ir_add,
  ir_negate, ir_halt, ir_branch,
  ir_cbranch, ir_iload, ir_istore,
  ir_dload, ir_dadd, ir_sub);
  iar: indirect_addr_register port map( clk, iar_enA, iar_ld, reset, abus, dbus);
  acc: accumulator port map( clk, acc_enD, acc_ld, acc_selAlu, reset, alu_result, dbus, acc_Q);
  aluu: alu port map( alu_op, acc_Q, dbus, alu_result, alu_accZ);
  mem: ram port map( mem_rw, mem_enD, reset, abus, dbus);

  signal abus: STD_LOGIC_VECTOR(15 downto 0);
signal dbus: STD_LOGIC_VECTOR(15 downto 0);
signal mem_enD, mem_rw: STD_LOGIC;
signal pc_enA, pc_ld, pc_inc: STD_LOGIC;
signal ir_enA, ir_enD, ir_ld: STD_LOGIC;
signal ir_load, ir_store, ir_add: STD_LOGIC;
signal ir_negate, ir_halt, ir_branch: STD_LOGIC;
signal ir_cbranch, ir_iload, ir_istore: STD_LOGIC;
signal ir_dload, ir_dadd: STD_LOGIC;
signal ir_sub: STD_LOGIC;
signal iar_enA, iar_ld: STD_LOGIC;
signal acc_enD, acc_ld, acc_selAlu: STD_LOGIC;
signal acc_Q: STD_LOGIC_VECTOR(15 downto 0);
signal alu_op: STD_LOGIC_VECTOR(1 downto 0);
signal alu_accZ: STD_LOGIC;
signal alu_result: STD_LOGIC_VECTOR(15 downto 0);
ctl: controller port map(clk, reset, mem_enD, mem_rw,
    pc_enA, pc_ld, pc_inc,
    ir_enA, ir_enD, ir_ld, ir_load, ir_store, ir_add,
    ir_negate, ir_halt, ir_branch, ir_cbranch, ir_iload,
    ir_istore, ir_dload, ir_dadd, ir_sub,
    iar_enA, iar_ld, acc_enD, acc_ld, acc_selAlu,
    alu_accZ, alu_op);

abuxX <= abus;
dbusX <= dbus;
mem_enDX <= mem_enD;
mem_rwX <= mem_rw;
pc_enAX <= pc_enA;
pc_ldX <= pc_ld;
pc_incX <= pc_inc;
ir_enAX <= ir_enA;
ir_enDX <= ir_enD;
ir_ldX <= ir_ld;
 iar_enAX <= iar_enA;
 iar_ldX <= iar_ld;
 acc_enDX <= acc_enD;
 acc_ldX <= acc_ld;
 acc_selAluX <= acc_selAlu;
 acc_QX <= acc_Q;
 alu_opX <= alu_op;
 alu_accZX <= alu_accZ;
end topArch;