Solutions - Homework 2

(Due date: October 3rd @ 9:30 am)
Presentation and clarity are very important!

PROBLEM 1 (10 PTS)

- Complete the following table. Use the fewest number of bits for every case. Show your procedure.

Decimal	Sign-and-magnitude	1's complement	2's complement
-257	1100000001	1011111110	1011111111
128	010000000	010000000	010000000
-120	11111000	10000111	10001000
-61	1111101	1000010	1000011
-1022	11111111110	10000000001	10000000010
511	0111111111	0111111111	0111111111

PROBLEM 2 (15 PTS)

- We need to perform the following operations, where numbers are represented in 2's complement:
  - 73 + 45
  - -35 + 64
  - -129 + 128
  - -255 - 230
  - 490 + 47
  - 990 + 113
- For each case:
  - ✓ Determine the minimum number of bits required to represent both summands. You might need to sign-extend one of the summands, since for proper summation, both summands must have the same number of bits.
  - ✓ Perform the binary addition in 2's complement arithmetic. The result must have the same number of bits as the summands.
  - ✓ Determine whether there is overflow by:
    - i. Using c_n, c_{n-1} (carries).
    - ii. Performing the operation in the decimal system and checking whether the result is within the allowed range for n bits, where n is the minimum number of bits for the summands.
  - ✓ If we want to avoid overflow, what is the minimum number of bits required to represent the summands and the result?

n = 8 bits

c_8=0 c_7=0

+73 = 01001001 +
+45 = 00101101

+118 = 01110110

overflow = $c_8 \oplus c_7$=0 -> no overflow

+118 ∈ $[-2^7, 2^7-1]$ -> no overflow

n = 8 bits

c_8=1 c_7=1

-35 = 11011101 +
+64 = 01000000

+29 = 00011101

overflow = $c_8 \oplus c_7$=0 -> no overflow

+29 ∈ $[-2^7, 2^7-1]$ -> no overflow

n = 9 bits

c_9=0 c_8=0

-129 = 101111111 +
+128 = 010000000

-1 = 111111111

overflow = $c_9 \oplus c_8$=0 -> no overflow

-1 ∈ $[-2^8, 2^8-1]$ -> no overflow

n = 9 bits

$c_9=1$ $c_8=0$

```
−255 = 100000001 +
−230 = 100011010
-------------------
       000011011
```

overflow = $c_9 \oplus c_8 = 1$ -> overflow!

$-255-230=-485 \notin [-2^8, 2^8-1]$ -> overflow!

To avoid overflow:
n = 10 bits (sign-extension):

$c_{10}=1$ $c_9=1$

```
−255 = 1100000001 +
−230 = 1100011010
-------------------
−485 = 1000011011
```

overflow = $c_{10} \oplus c_9 = 0$ -> no overflow

$-485 \in [-2^9, 2^9-1]$ -> no overflow

n = 10 bits

$c_{10}=0$ $c_9=1$

```
+490 = 0111101010 +
 +47 = 0000101111
-------------------
      1000011001
```

overflow = $c_{10} \oplus c_9 = 1$ -> overflow!

$490+47=537 \notin [-2^9, 2^9-1]$ -> overflow!

To avoid overflow:
n = 11 bits (sign-extension):

$c_{11}=0$ $c_{10}=0$

```
+490 = 00111101010 +
 +47 = 00000101111
-------------------
+537 = 01000011001
```

overflow = $c_{11} \oplus c_{10} = 0$ -> no overflow

$+537 \in [-2^{10}, 2^{10}-1]$ -> no overflow

n = 11 bits

$c_{11}=0$ $c_{10}=1$

```
+990 = 01111011110 +
+113 = 00001110001
-------------------
      10001001111
```

overflow = $c_{11} \oplus c_{10} = 1$ -> overflow!

$990+113=1103 \notin [-2^{10}, 2^{10}-1]$ -> overflow!

To avoid overflow:
n = 12 bits (sign-extension):

$c_{12}=0$ $c_{11}=0$

```
+990 = 001111011110 +
+113 = 000001110001
-------------------
+1103= 010001001111
```

overflow = $c_{12} \oplus c_{11} = 0$ -> no overflow

$+1103 \in [-2^{11}, 2^{11}-1]$ -> no overflow

PROBLEM 3 (15 PTS)

- Sign-extension in 2's complement: Whenever we need to increase the number of bits for representing a number, we append the MSB to the left as many times as needed:

$$b_{n-1}b_{n-2}\dots b_0 \equiv b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0$$

Examples: $00101_2 = 00000101_2 = 2^2 + 2^0 = 5$

$10101_2 = 11110101_2 = -2^4 + 2^2 + 2^0 = -2^6 + 2^5 + 2^4 + 2^2 + 2^0 = -11$

We can think of the sign-extended number as an m-bit number, where $m > n$:

$$b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0 = k_{m-1}\dots k_n k_{n-1}k_{n-2}\dots k_0$$

- Demonstrate that $b_{n-1}b_{n-2}\dots b_0$ represents the same decimal number as $b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0$, i.e., demonstrate that sign-extension is correct for any $m > n$.

Useful formula: $\sum_{i=k}^{l} r^i = \frac{r^k - r^{l+1}}{1-r}, r \neq 1$

We need to demonstrate that: $b_{m-1}\dots b_n b_{n-1}b_{n-2}\dots b_0 = b_{n-1}\dots b_{n-1}b_{n-1}b_{n-2}\dots b_0 = b_{n-1}b_{n-2}\dots b_0$, where: $b_i = b_{n-1}, i = n, n+1, \dots, m-1$

Using the formula for 2's complement numbers:

$$-2^{m-1}b_{m-1} + \sum_{i=0}^{m-2} 2^i b_i = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

$$-2^{m-1}b_{m-1}+\sum_{i=n-1}^{m-2}2^i b_i+\sum_{i=0}^{n-2}2^i b_i=-2^{n-1}b_{n-1}+\sum_{i=0}^{n-2}2^i b_i \Rightarrow -2^{m-1}b_{m-1}+\sum_{i=n-1}^{m-2}2^i b_i=-2^{n-1}b_{n-1}$$

$$-2^{m-1}b_{n-1}+b_{n-1}\sum_{i=n-1}^{m-2}2^i=-2^{n-1}b_{n-1},\ \sum_{i=k}^{l}2^i=\frac{2^k-2^{l+1}}{1-2}=2^{l+1}-2^k$$

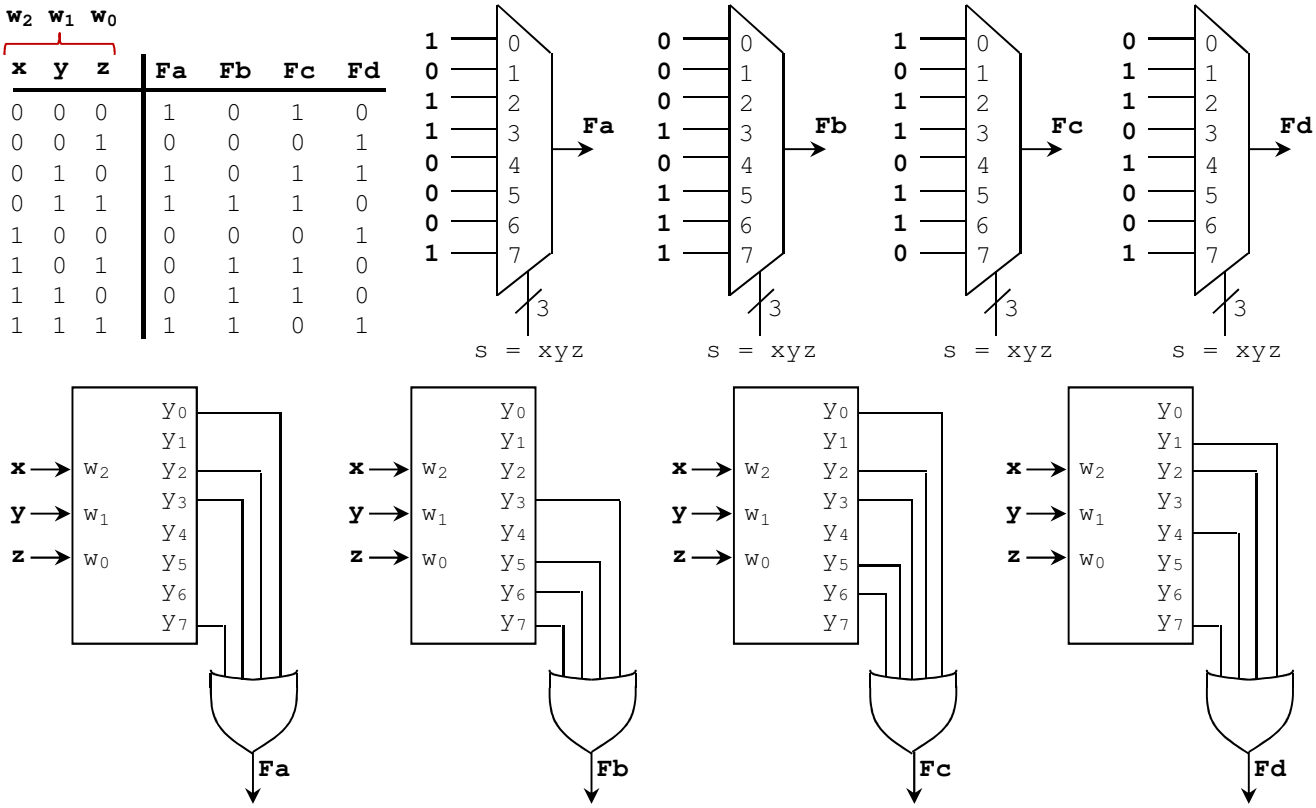
$$-2^{m-1}b_{n-1}+b_{n-1}(2^{m-1}-2^{n-1})=-2^{n-1}b_{n-1}$$

$$-2^{m-1}b_{n-1}+2^{m-1}b_{n-1}-2^{n-1}b_{n-1}=-2^{n-1}b_{n-1}\ \therefore -2^{n-1}b_{n-1}=-2^{n-1}b_{n-1}$$

PROBLEM 4 (15 PTS)

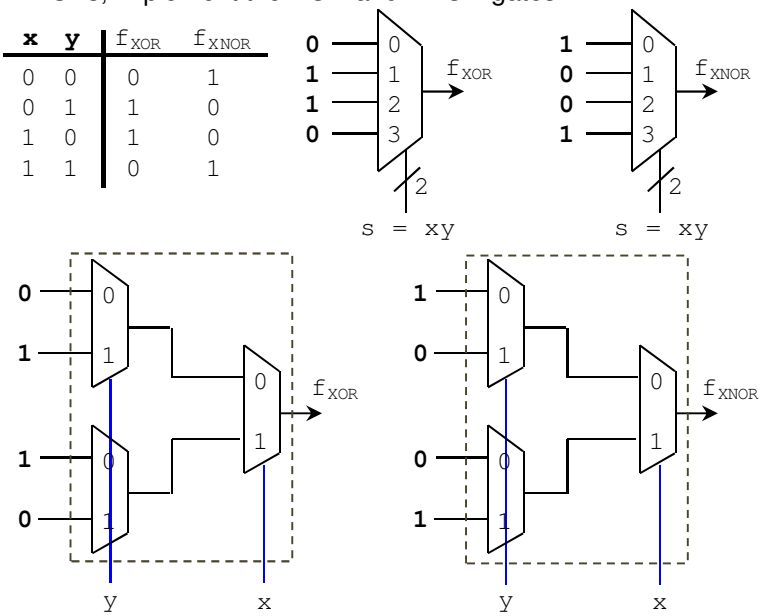
- Implement the following functions using i) decoders and ii) multiplexers:
  - ✓ $F_a=\overline{X+Z}+ZY$
  - ✓ $F_b=XY+YZ+XZ$
  - ✓ $F_c(X,Y,Z)=\prod(M_1,M_4,M_7)$
  - ✓ $F_d=X\oplus Y\oplus Z$

w2 w1 w0						
x	y	z	Fa	Fb	Fc	Fd
0	0	0	1	0	1	0
0	0	1	0	0	0	1
0	1	0	1	0	1	1
0	1	1	1	1	1	0
1	0	0	0	0	0	1
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	0	1



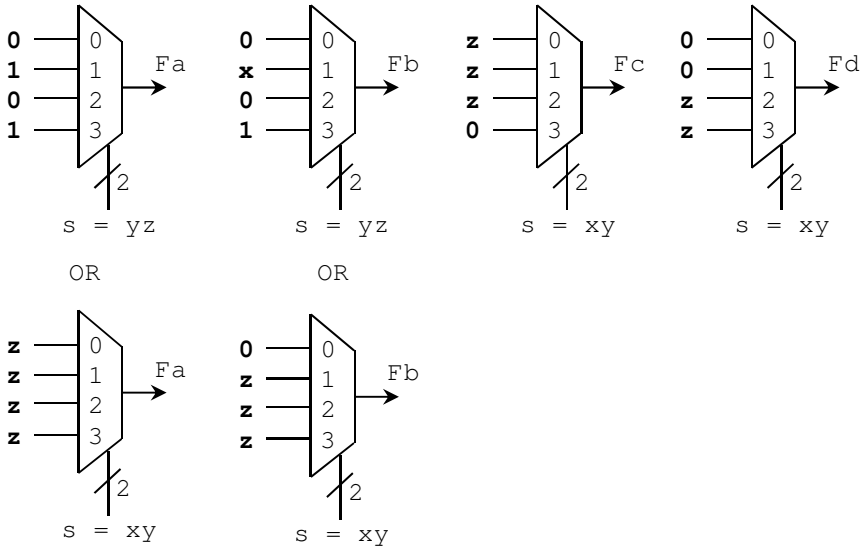
- Using only 2-to-1 MUXs, implement the XOR and XNOR gates.

x	y	f_{XOR}	f_{XNOR}
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1



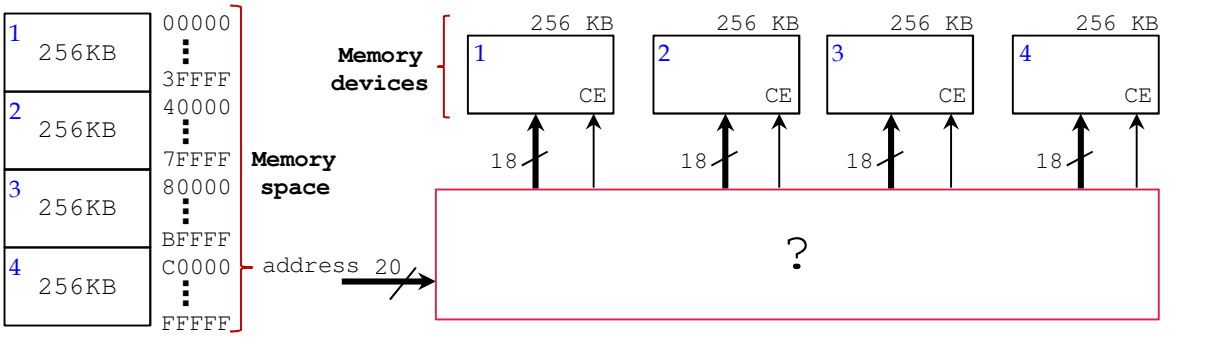
- Using only a 4-to-1 MUX, implement the following functions.
  - $F_a(X,Y,Z) = \sum(m_1, m_3, m_5, m_7)$
  - $F_b(X,Y,Z) = \sum(m_3, m_5, m_7)$
  - $F_c(X,Y,Z) = \sum(m_1, m_3, m_5)$
  - $F_d(X,Y,Z) = \sum(m_5, m_7)$

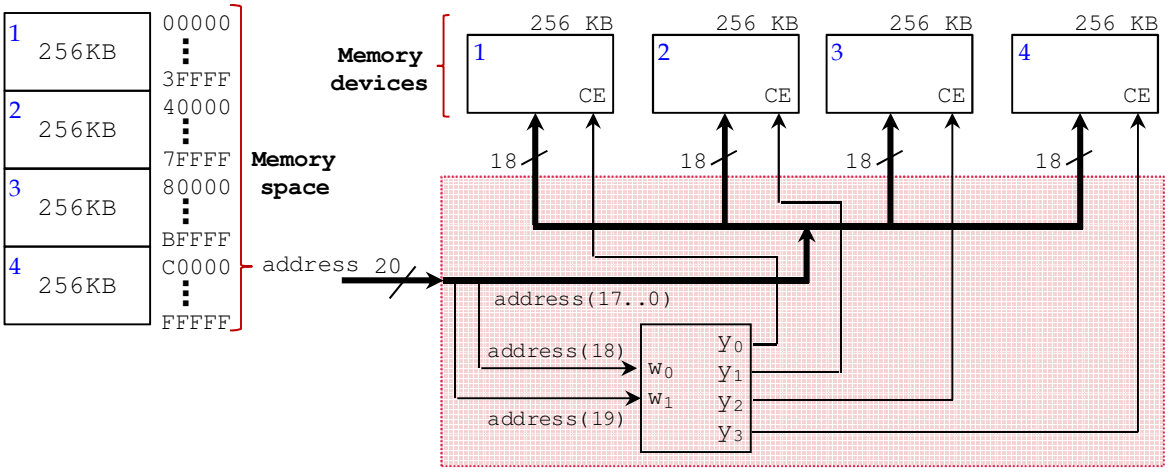
x	y	z	Fa	Fb	Fc	Fd
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	1	1	1	0	1



PROBLEM 5 (10 PTS)

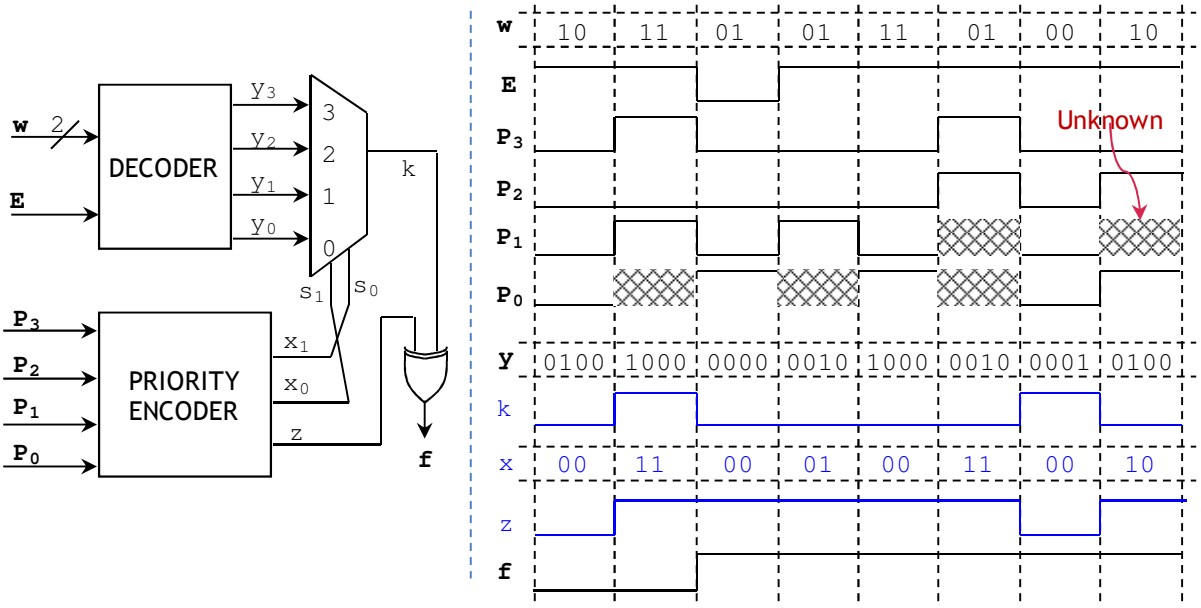
- A 20-bit address line in a µprocessor handles up to $2^{20} = 1\,MB$ of addresses, each address containing one-byte of information. We want to connect four 256KB memory chips to the µprocessor.
- Sketch the circuit that: i) addresses the memory chips, and ii) enables only one memory chip (via CE: chip enable) when the address falls in the corresponding range. Example: if $address = 0x5FFFF$, → only memory chip 2 is enabled (CE=1). If $address = 0xD0123$, → only memory chip 4 is enabled.



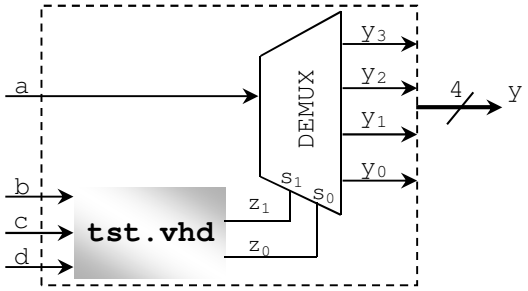


PROBLEM 6 (15 PTS)

- Complete the timing diagram of the circuit shown below:

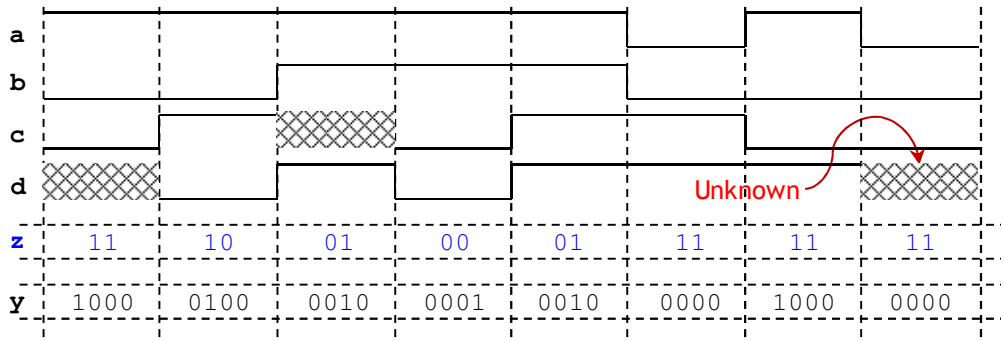


- The following VHDL code corresponds to the shaded circuit. Complete the timing diagram:



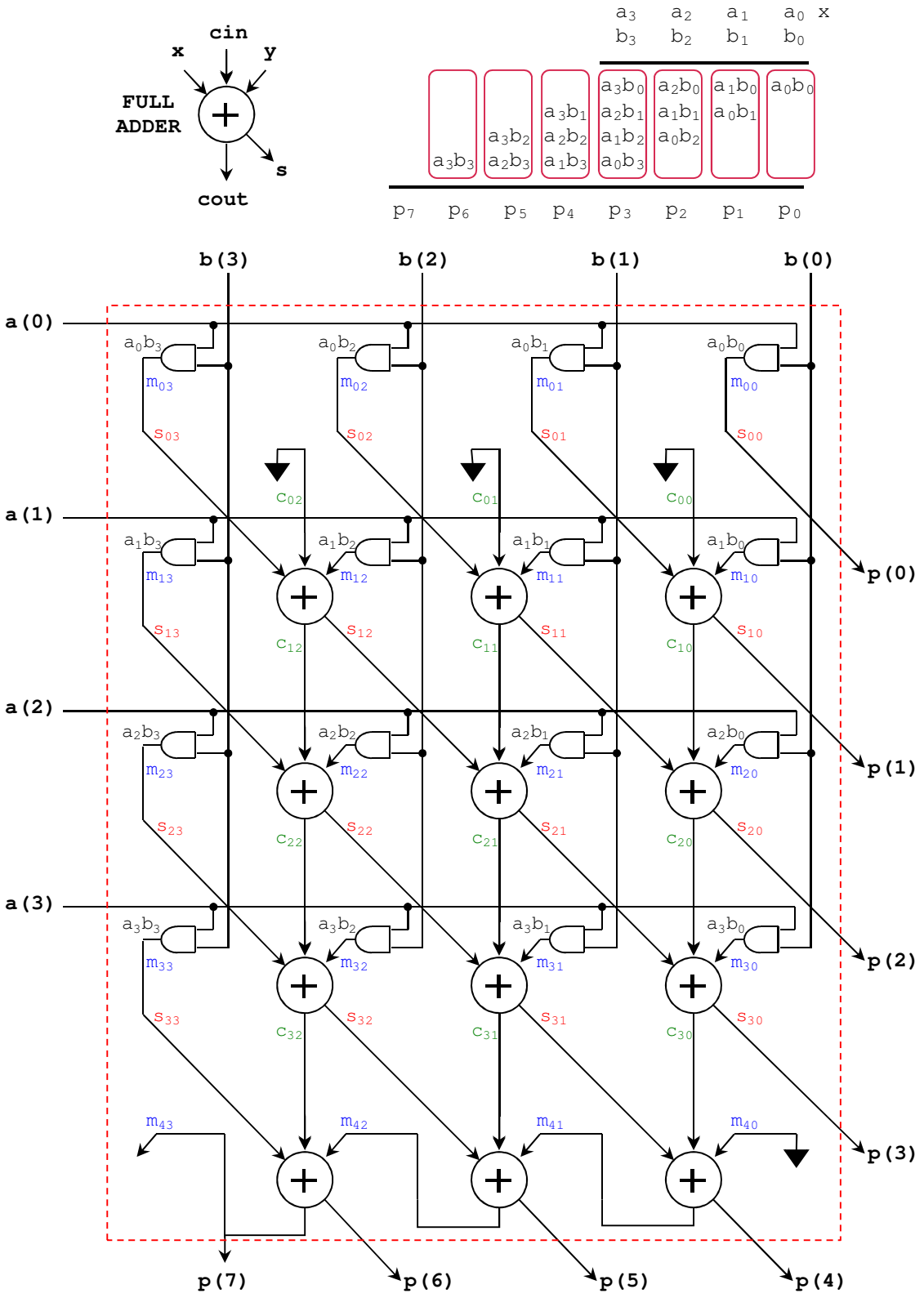

```
library ieee;
use ieee.std_logic_1164.all;

entity tst is
  port (b,c,d: in std_logic;
        z: out std_logic_vector(1 downto 0));
end tst;
```

```
architecture bhv of circ is
  signal x, y: std_logic;
begin
  process (b,c,d)
  begin
    z <= c & d;
    if b = '1' then
       case d is
          when '1' => z <= "01";
          when others => z <= "00";
       end case;
    else
      if c = '0' then z <= "11";
      end if;
    end if;
  end process;
end bhv;
```



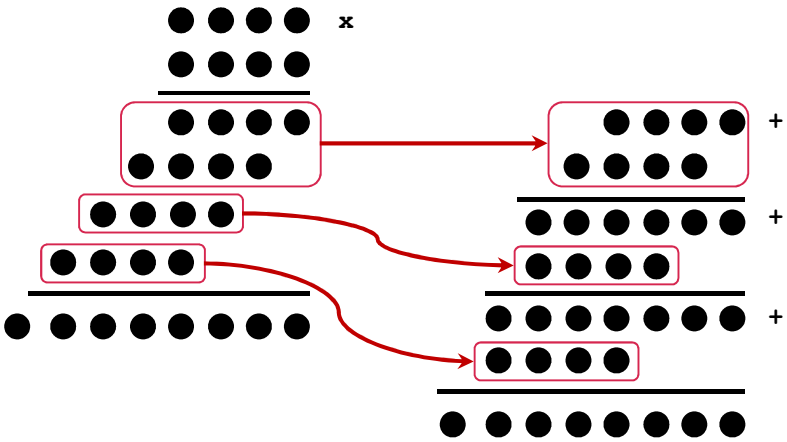
PROBLEM 7 (20 PTS)

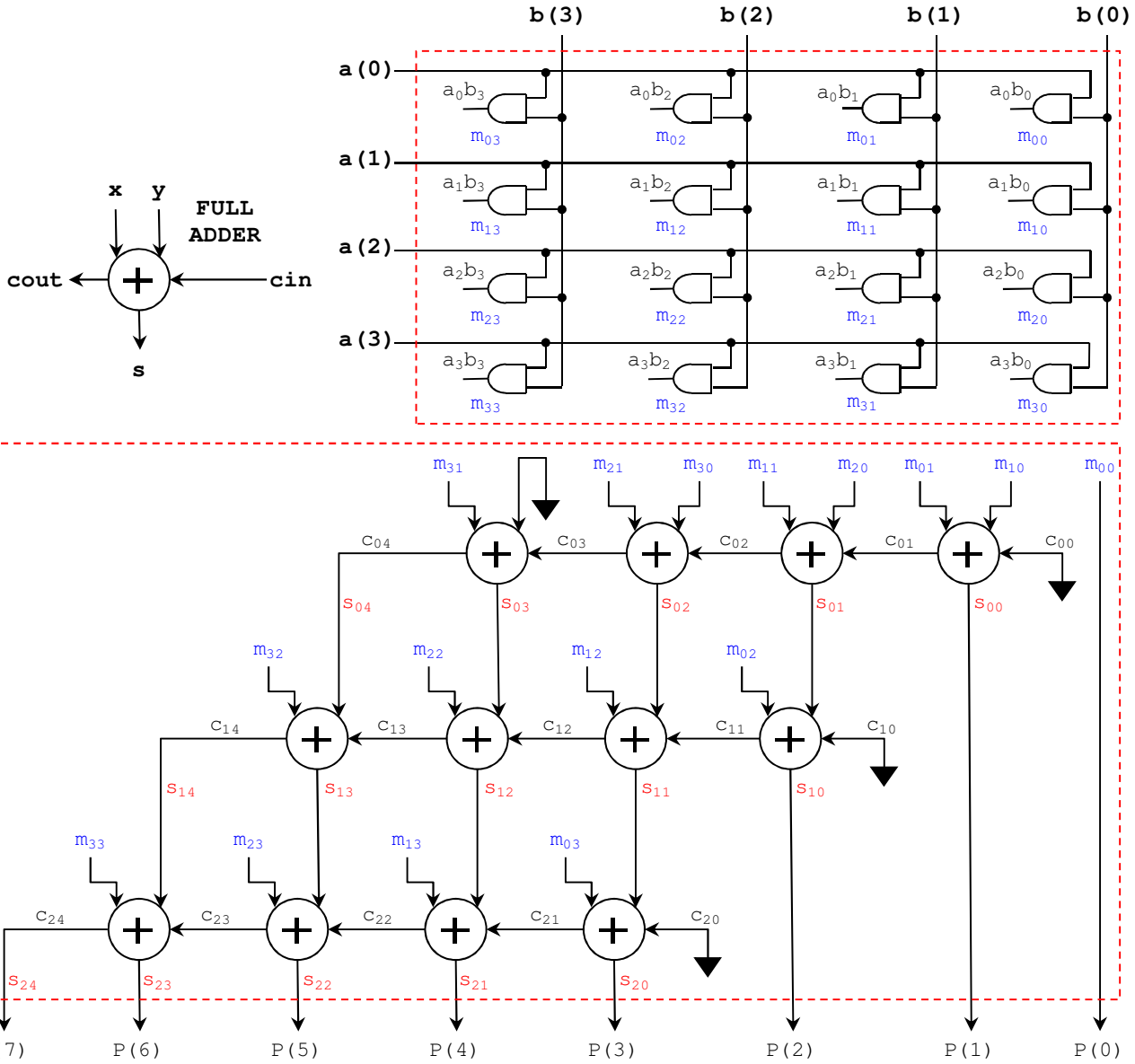
- A straightforward implementation of the multiplication operation (for positive or unsigned numbers) is called ***Array Multiplier***: the partial products are added up at every column. The figure depicts the hardware implementation for multiplying two unsigned number of 4 bits.



- One of the drawbacks of this approach is the long delay between the input and the output. In the figure, the bits p_7, p_6, p_4, p_4 require the inputs to pass through 4 adders as well as AND gates.

- An alternative implementation, called ***Wallace Multiplier***, features a shorter delay between the input and the output. The hardware implementation is more complicated though.
- The figure shows a Wallace multiplier implementation for two unsigned numbers of 4 bits. Note how the addition operation of the 4 rows has been rearranged so that only 2 rows are added up at every stage.





- ✓ Write the VHDL implementation for the multiplication operation of two unsigned numbers of 4 bits using: i) Array Multiplier, and ii) Wallace multiplier. Use the ***Structural Description***: create a separate VHDL file for the Full Adder circuit.
- ✓ Simulate both circuits and make sure that they are performing the correct operation.
- ✓ Attach your VHDL code, testbench code, and simulation results.

ARRAY MULTIPLIER:

VHDL Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity my_mult is
   generic (N: INTEGER:= 4);
        port (A,B: in std_logic_vector (N-1 downto 0);
              P: out std_logic_vector (2*N-1 downto 0));
end my_mult;

architecture structure of my_mult is
        component fulladd
               port( cin, x, y : in std_logic;
                              s, cout   : out std_logic);
        end component;

        type my_array is array (natural range <>, natural range <>) of std_logic;
        signal m: my_array(N downto 0, N-1 downto 0);
        signal s: my_array(N-1 downto 0, N-1 downto 0);
        signal c: my_array(N-1 downto 0, N-2 downto 0);

begin

-- Array of N * (N-1) full adders:
   gi: for i in 0 to N-1 generate -- along rows
              gj: for j in 0 to N-1 generate -- along columns
                     m(i,j) <= A(i) and B(j);
                     fa: if i /= N-1 and j /= N-1 generate
                            fij: fulladd port map (cin => c(i,j), x => s(i,j+1) , y => m(i+1,j),
                                                   s => s(i+1,j),  cout => c(i+1,j));
                         end generate;
                     fb: if i = 0 generate
                            s(i,j) <= m(i,j);
                         end generate;
                  end generate;
        end generate;

   m(N,0) <= '0';
   glj: for j in 0 to N-2 generate
           c(0,j) <= '0';
           s(j+1,N-1) <= m(j+1,N-1); -- Column 3 (from rows 1 to N-1)
           P(j+1) <= s(j+1,0);
           flj: fulladd port map (cin => c(N-1,j), x => s(N-1,j+1), y => m(N,j), s => p(N+j),
                                  cout => m(N,j+1));
         end generate;

   P(0) <= m(0,0);
   P(2*N-1) <= m(N,N-1);

end structure;
```

Full Adder VHDL Code:

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

entity fulladd is
        port( cin, x, y : in std_logic;
          s, cout   : out std_logic);
end fulladd;

architecture structure of fulladd is

begin
        s <= x xor y xor cin;
        cout <= (x and y) or (x and cin) or (y and cin);
end structure;
```

Testbench Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.all;

ENTITY tb_my_mult IS
       generic (N: INTEGER:= 4);
END tb_my_mult;

ARCHITECTURE behavior OF tb_my_mult IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT my_mult
    PORT(
         A : IN  std_logic_vector(N-1 downto 0);
         B : IN  std_logic_vector(N-1 downto 0);
         P : OUT  std_logic_vector(2*N-1 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic_vector(N-1 downto 0) := (others => '0');
   signal B : std_logic_vector(N-1 downto 0) := (others => '0');

   --Outputs
   signal P : std_logic_vector(2*N-1 downto 0);

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: my_mult PORT MAP (
          A => A,
          B => B,
          P => P
        );

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      -- insert stimulus here
      A <= conv_std_logic_vector(3,N);  B <= conv_std_logic_vector (5,N); wait for 20 ns;
      A <= conv_std_logic_vector(12,N); B <= conv_std_logic_vector (11,N); wait for 20 ns;
      A <= conv_std_logic_vector(10,N); B <= conv_std_logic_vector (13,N); wait for 20 ns;
      A <= conv_std_logic_vector(8,N);  B <= conv_std_logic_vector (14,N); wait for 20 ns;
      A <= conv_std_logic_vector(10,N); B <= conv_std_logic_vector (8,N); wait for 20 ns;
      A <= conv_std_logic_vector(2**N-1,N); B <=conv_std_logic_vector (2**N-1,N); wait for 20 ns;
      A <= conv_std_logic_vector(0,N); B <= conv_std_logic_vector (0,N); wait for 20 ns;

      wait;
   end process;

END;
```

WALLACE MULTIPLIER (4-BIT):

VHDL Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity my_wallace_4bit is
       port (A, B: in std_logic_vector (3 downto 0);
             P : out std_logic_vector (7 downto 0));
end my_wallace_4bit;

architecture structure of my_wallace_4bit is

       component fulladd
              port( cin, x, y : in std_logic;
                           s, cout   : out std_logic);
       end component;

       type my_array is array (natural range <>, natural range <>) of std_logic;
       signal m: my_array(3 downto 0, 3 downto 0);
       signal c: my_array(2 downto 0, 4 downto 0);
       signal s: my_array(2 downto 0, 4 downto 0);

begin

   gi: for i in 0 to 3 generate -- along rows
          gj: for j in 0 to 3 generate -- along columns
                 m(i,j) <= A(i) and B(j);
              end generate;
         end generate;

   c(0,0) <= '0'; c(1,0) <= '0'; c(2,0) <= '0';
   s(0,4) <= c(0,4); s(1,4) <= c(1,4); s(2,4) <= c(2,4);

-- First Layer
f00: fulladd port map (cin => c(0,0), x => m(0,1), y => m(1,0), s => s(0,0), cout => c(0,1));
f01: fulladd port map (cin => c(0,1), x => m(1,1), y => m(2,0), s => s(0,1), cout => c(0,2));
f02: fulladd port map (cin => c(0,2), x => m(2,1), y => m(3,0), s => s(0,2), cout => c(0,3));
f03: fulladd port map (cin => c(0,3), x => m(3,1), y => '0', s => s(0,3), cout => c(0,4));

-- Second Layer
f10: fulladd port map (cin => c(1,0), x => m(0,2), y => s(0,1), s => s(1,0), cout => c(1,1));
f11: fulladd port map (cin => c(1,1), x => m(1,2), y => s(0,2), s => s(1,1), cout => c(1,2));
f12: fulladd port map (cin => c(1,2), x => m(2,2), y => s(0,3), s => s(1,2), cout => c(1,3));
f13: fulladd port map (cin => c(1,3), x => m(3,2), y => s(0,4), s => s(1,3), cout => c(1,4));

-- Third Layer
f20: fulladd port map (cin => c(2,0), x => m(0,3), y => s(1,1), s => s(2,0), cout => c(2,1));
f21: fulladd port map (cin => c(2,1), x => m(1,3), y => s(1,2), s => s(2,1), cout => c(2,2));
f22: fulladd port map (cin => c(2,2), x => m(2,3), y => s(1,3), s => s(2,2), cout => c(2,3));
f23: fulladd port map (cin => c(2,3), x => m(3,3), y => s(1,4), s => s(2,3), cout => c(2,4));

       P(0) <= m(0,0);
       P(1) <= s(0,0);
       P(2) <= s(1,0);
       P(3) <= s(2,0);
       P(4) <= s(2,1);
       P(5) <= s(2,2);
       P(6) <= s(2,3);
       P(7) <= s(2,4);

end structure;
```

Testbench Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.all;

ENTITY tb_my_wallace_4bit IS
END tb_my_wallace_4bit;

ARCHITECTURE behavior OF tb_my_wallace_4bit IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT my_wallace_4bit
    PORT(
         A : IN  std_logic_vector(3 downto 0);
         B : IN  std_logic_vector(3 downto 0);
         P : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic_vector(3 downto 0) := (others => '0');
   signal B : std_logic_vector(3 downto 0) := (others => '0');

   --Outputs
   signal P : std_logic_vector(7 downto 0);

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: my_wallace_4bit PORT MAP (
          A => A,
          B => B,
          P => P
        );

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
      -- insert stimulus here
               li: for i in 0 to 15 loop
                       A <= conv_std_logic_vector(i,4);
                       lj: for j in 0 to 15 loop
                              B <= conv_std_logic_vector (j,4); wait for 20 ns;
                           end loop;
                       end loop;
               A <= x"0"; B <= x"0"; wait for 20 ns;

      wait;
   end process;

END;
```