CS226 Mid-Semester Examination (Spring 2016)

Max marks: 60 **Time: 120 mins**

- ***The exam is open book and notes brought to the exam hall.***
- *Be brief, complete and stick to what has been asked.*
- *Unless asked for explicitly, you may cite results/proofs covered in class without reproducing them.*
- *If you need to make any assumptions, state them clearly.*
- ***Please start writing your answer to each sub-question on a fresh page. DO NOT write answers to multiple sub-questions on the same page.***
- ***The use of internet enabled devices is strictly prohibited. You will be debarred from taking the examination if you are found accessing the internet during the examination. All IIT Bombay rules apply in this respect.***
- ***Please do not engage in unfair or dishonest practices during the examination. Anybody found indulging in such practices will be referred to the D-ADAC.***

1. (a) *[3+3 marks]* In Boolean algebra parlance, a *literal* is either a variable or its complement. For example, if x_1 and x_3 are Boolean variables, then x_1 and x_3' are literals. A *minterm* is either a single literal or an AND of more literals. Similarly, a *maxterm* is either a single literal or an OR of more literals. For example, $x_1.x_3'$ is a minterm, while $x_1 + x_3'$ is a maxterm. A *sum-of-products* (SOP) representation of a Boolean function is the OR of one or more *minterms*. Similarly, a *product-of-sums* (POS) representation is the AND of one or more maxterms. For example, $x_1.x_2' + x_1.x_3'$ is in SOP form, while $x_1.(x_3' + x_4')$ is in POS form.

   Let $G(x_1, x_2, x_3, x_4)$ and $F(x_1, x_2, x_3)$ be Boolean functions given by the K-maps shown below. Give a *minimal SOP representation* of F and a *minimal POS representation* of G. A minimal representation uses as few minterms/maxterms as possible, and each minterm/maxterm uses as few literals as possible.

G:

$\frac{x_1,x_2\rightarrow}{x_3,x_4\downarrow}$	**00**	**01**	**11**	**10**
00	1	0	1	1
01	0	1	1	1
11	0	1	0	1
10	1	1	0	1

F:

$\frac{x_1,x_2\rightarrow}{x_3\downarrow}$	**00**	**01**	**11**	**10**
0	1	1	0	0
1	0	1	1	1

Answer:
$F = x_1'.x_3' + x_1.x_3 + x_3.x_2$ or $F = x_1'.x_3' + x_1.x_3 + x_1'.x_2$
$G = (x_4' + x_1 + x_2).(x_3' + x_1' + x_2').(x_1 + x_2' + x_3 + x_4)$

(b) *[5 marks]* Give a K-map of the function $H(x_1, x_2, x_3, x_4) = F(x_1, G(x_1, x_2, x_3, x_4), x_3)$, where F and G are as in sub-question (a) above.

Answer:

$\frac{x_1,x_2\rightarrow}{x_3,x_4\downarrow}$	**00**	**01**	**11**	**10**
00	1	1	0	0
01	1	1	0	0
11	0	1	1	1
10	1	1	1	1

(c) *[4 marks]* The above operation is also called *function composition* and is usually denoted as $\texttt{compose}(F, x_2, G)$, i.e. replace all occurrences of x_2 in F by G. In general, function compositions can be chained in a natural way, e.g. we can talk of $\texttt{compose}(\texttt{compose}(F, x_2, G), x_1, F)$ as being the result of $\texttt{compose}(H, x_1, F)$, where $H = \texttt{compose}(F, x_2, G)$.

Show by means of an example that if F, G and H are three Boolean functions (not necessarily the ones in the previous subquestions) that depend on variables x_1 and x_2, then $\texttt{compose}(\texttt{compose}(F, x_1, G), x_2, H)$ is not necessarily the same (in terms of truth table or K-map) as $\texttt{compose}(\texttt{compose}(F, x_2, H), x_1, G)$. You are free to chooose functions F, G and H different from those in the previous sub-questions.

Answer:

There are several possible examples that work. Here is a simple one:
$F = x_1 + x_2$, $G = x_1' + x_2'$, $H = x_1.x_2$.
Then $\texttt{compose}(\texttt{compose}\ (F, x_1, G), x_2, H) = \texttt{compose}(x_1 + x_1' + x_2', x_2, H) = \texttt{compose}(1, x_2, H) = 1$.
However, $\texttt{compose}(\texttt{compose}\ (F, x_2, H), x_1, G) = \texttt{compose}(x_1 + x_1.x_2, x_1, G) = \texttt{compose}(x_1, x_1, G) = x_1' + x_2'$.

---

2. In environments where data is likely to be corrupted, *dual-rail encoding* is often used to represent binary values. In dual-rail encoding, two Boolean variables, say $v_1 v_0$, are used to encode a Boolean value v or an invalid value, as shown in the table below.

v_1	v_0	Boolean value represented
1	0	1
0	1	0
0	0	None (invalid)
1	1	None (invalid)

As you can see, if any bit in a dual-rail encoded Boolean value is flipped (or corrupted), we can immediately detect the corruption.

You are required to design the following datapath elements where the inputs and outputs are dual-rail encoded.

(a) *[3 × 4 marks]* A full adder that takes six inputs, a_1 a_0 b_1 b_0 cin_1 cin_0 and generates four outputs sum_1 sum_0 $cout_1$ $cout_0$. Note that it is not the case that one of the inputs being invalid renders all outputs invalid. For example, if a_1 a_0 b_1 b_0 cin_1 $cin_0 = 010111$, then we have a case where the inputs a and b are both 0 and the carry in (cin) is invalid. In this case, the sum is indeed invalid;

hence sum_1 and sum_0 should be either 11 or 00. However, carry out is certainly 0, and hence $cout_1$ $cout_0$ should be 01. Your design must take all of these cases into account.

Answer:

Let $\overline{\oplus}$ denote the xnor function. Clearly, an input v is invalid if $v_1 \overline{\oplus} v_0$ is 1.

In a normal (not dual-rail-encoded) full-adder, the *sum* and *cout* are given by:
$sum = a \oplus (b \oplus cin)$, and $cout = a.b + b.cin + a.cin$.

If none of the inputs are invalid, then simply looking at the values of a_1, b_1 and cin_1 gives us the Boolean values of a, b and cin, respectively. Similarly, looking at the values of a'_0, b'_0 and cin'_0 also gives us the Boolean values of a, b and cin, respectively. Therefore, if none of the inputs are invalid, we have:
$sum_1 = a_1 \oplus (b_1 \oplus cin_1)$ and $cout_1 = a_1.b_1 + b_1.cin_1 + a_1.cin_1$.
Similarly, $sum'_0 = a'_0 \oplus (b'_0 \oplus cin'_0)$ and $cout'_0 = a'_0.b'_0 + b'_0.cin'_0 + a'_0.cin'_0$.

Note that the above Boolean functions for $sum_1, sum_0, cout_1, cout_0$ also work correctly even if at most one input is invalid. Can you convince yourself of this?

In fact, there is something more general that we can state. Let $F(x, y, \ldots z)$ be a Boolean function (not using dual-rail encoding). Now, if we use dual-rail encoding of the inputs and output, then $f_1 = F(x_1, y_1, \ldots z_1)$ and $f'_0 = F(x'_0, y'_0, \ldots z'_0)$ works correctly (gives dual-rail encoded output of the Boolean function) if at most one input is invalid. Try to convince yourself why. However, this may not work correctly if two or more inputs are invalid. So in our case too, we'll need to take care of these cases separately.

Clearly, if two or more of a, b or cin are invalid, then *sum* must be invalid, since the output of an xor gate depends on the values of all its inputs. Similarly, if two or more of a, b or cin are invalid, then *cout* must also be invalid (easy to see from the expression $a.b + a.cin + b.cin$).

This suggests the following Boolean functions for sum_1, sum_0, $cout_1$ and $cout_0$.

$$\begin{aligned}
a_{invalid} &= (a_1 \overline{\oplus} a_0) \\
b_{invalid} &= (b_1 \overline{\oplus} b_0) \\
cin_{invalid} &= (cinv_1 \overline{\oplus} cinv_0) \\
t &= a_{invalid}.b_{invalid} + a_{invalid}.cin_{invalid} + b_{invalid}.cin_{invalid} \qquad (1) \\
sum_1 &= t'.(a_1 \oplus b_1 \oplus cin_1) \\
sum_0 &= t'.(a'_0 \oplus b'_0 \oplus cin'_0)' \\
cout_1 &= t'.(a_1.b_1 + a_1.cin_1 + b_1.cin_1) \\
cout_0 &= t'.(a'_0.b'_0 + a'_0.cin'_0 + b'_0.cin'_0)'
\end{aligned} \qquad (2)$$

In the above, t becomes 1 iff two or more inputs are invalid. Note the way in which t is used in the above expressions for $sum_1, sum_0, cout_1$ and $cout_0$. Effectively, whenever t is 0, we let the expressions obtained earlier (that work correctly when 1 or fewer inputs are invalid) determine the outputs. Otherwise, we set all the outputs to 0. We could use t in an alternative way too – where whenever two or more inputs are invalid, we set all the outputs to 1. This gives.

$$\begin{aligned}
sum_1 &= t + (a_1 \oplus b_1 \oplus cin_1) \\
sum_0 &= t + (a'_0 \oplus b'_0 \oplus cin'_0)' \\
cout_1 &= t + (a_1.b_1 + a_1.cin_1 + b_1.cin_1) \\
cout_0 &= t + (a'_0.b'_0 + a'_0.cin'_0 + b'_0.cin'_0)'
\end{aligned} \qquad (3)$$

(b) *[3 × 2 marks]* A 2-to-1 multiplexer that takes six inputs $d0_1$ $d0_0$ $d1_1$ $d1_0$ c_1 c_0 and generates two outputs m_1 m_0, where $d0$ represents the input that should normally be copied to the output if the control c is 0, and $d1$ represents the input that should be normally be copied to the output if the control c is 1. In this case too, you should be careful **not** to assume that any one input being invalid renders the output invalid.

In each case, give Boolean functions for each of the dual-rail encoded outputs of the datapath element in terms of the dual-rail encoded inputs. You can use the flexibility of encoding an invalid value (there are two ways to encode an invalid value) to simplify your functions.

Answer:

The logic to be followed in this sub-question is similar to that followed in the previous sub-question. The normal (non dual-rail encoded) function for the output of a 2-to-1 multiplexer is:
$m = c.d1 + c'.d0$

Therefore, we can start with:
$m_1 = c_1.d1_1 + c_1'.d0_1$ and $m_0 = (c_0'.d1_0' + c_0.d0_0')'$.

The above works fine even if one of the inputs is invalid (recall the discussion in the solution to the previous sub-question). To take care of the case where more than one input is invalid, note that if c is invalid, the only case where the output should be valid is when both $d1$ and $d0$ are valid and equal. Also, if c is invalid, then c_1 and c_0 have the same value. Depending on whether $(c_1, c_0) = 11$ or 00, the above expressions for m_1 and m_0 give either $(m_1, m_0) = (d1_1, d0_0)$ or $(m_1, m_0) = (d0_1, d1_0)$. Therefore, the output can (erroneously) have a valid value when c is invalid in some cases when one of $d1$ and $d0$ is also invalid: e.g. if $(c_1, c_0) = 11$, $(d1_1, d1_0) = 10$ and $(d0_1, d0_0) = 00$. To prevent such cases, we wish to detect if c is invalid, and if one of $d1$ and $d0$ is also invalid, and set the output to invalid in such cases.

This gives rise to the following equations:

$$\begin{aligned} d1_{invalid} &= d1_1 \overline{\oplus} d1_0 \\ d0_{invalid} &= d0_1 \overline{\oplus} d0_0 \\ c_{invalid} &= c_1 \overline{\oplus} c_0 \\ t &= c_{invalid}.(d1_{invalid} + d0_{invalid}) \\ m_1 &= t'.(c_1.d1_1 + c_1'.d0_1) \\ m_0 &= t'.(c_0'.d1_0' + c_0.d0_0')' \end{aligned} \tag{4}$$

Like in the previous subquestion, we could also write the last two equations above as:

$$\begin{aligned} m_1 &= t + (c_1.d1_1 + c_1'.d0_1) \\ m_0 &= t + (c_0'.d1_0' + c_0.d0_0')' \end{aligned} \tag{5}$$

---

3. *[10 marks]* Consider the datapath shown in Fig. 1. For those of you who have tried to solve the practice questions, this is exactly the same datapath as in Practice Problem Set 3.

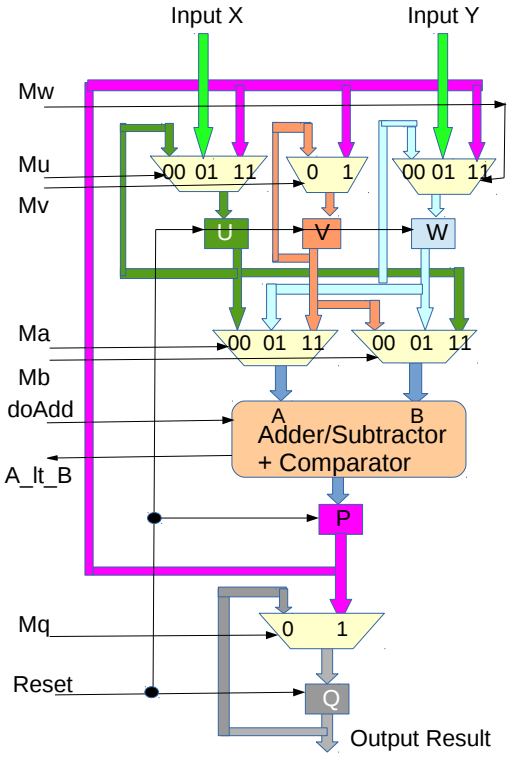

Figure 1: Datapath from Practice Problem Set 3

The datapath has the following components: five registers named U, V, W, P, Q of appropriate bit-widths, six multiplexors with control signals $M_w, M_u, M_v.M_a, M_b, M_q$, an adder/subtractor block that takes two inputs A and B and computes either $A+B$ or $A-B$ depending on the value of *doAdd* (adds if $doAdd = 1$, subtracts otherwise). The adder/subtractor also always gives a one-bit output A_lt_B that is set to 1 iff $A < B$ (treated as signed integers). The datapath has an input *Reset* that is used to reset all the registers, i.e when $Reset = 1$, the values stored in each of U, V, W, P, Q becomes 0. It is also assumed that all registers in the datapath are clocked by the same clock signal as the controller. There are two inputs X and Y to the datapath, and the output of register Q is assumed to be the result.

We wish to implement a variant of Euclid's greatest common divisor algorithm (shown below) using this datapath and a controller that you must design.

```
read X and Y; // signed integers
if ((X <= 0) OR (Y <= 0)) { return 0; }
else {  while (X != Y) {
            if (X > Y) {X = X - Y;}
            else {Y = Y - X;}
     }
  return X;
}
```

Please indicate clearly the registers where you are storing the values of X, Y and any intermediate results in your computation. You are free to add intermediate steps of computation

in the algorithm above, if it helps you solve the problem. However, all such steps must be clearly indicated.

For the controller, you must give the state transition table in the format given below. Note that the controller also generates an output signal named *Done* that becomes 1 only when the entire computation is over. You may assume that the values of X and Y do not change until the entire computation is over (i.e. until *Done* becomes 1).

You MUST indicate through brief comments what each row of the controller table achieves, e.g. resets registers, or adds U and V, etc. Answers without comments will get 0.

Answer:

We will store X in register U, Y in register W and 0 (a constant) in register V. Note that `(X <= 0) OR (Y <= 0)` can be written as `NOT ((X > 0) AND (Y > 0))`. Similarly, `(X != Y)` can be written as `(X > Y) OR (Y > X)`. These observations are useful since the datapath only provides a signal that allows us to determine if `A < B`, wher `A` and `B` are the inputs of the adder/subtractor/comparator. Thus, we have the following table.

CurSt	A_lt_B	NxtSt	M_u	M_v	M_w	M_a	M_b	M_q	Reset	doAdd	Done	Comment
S_0	-	S_1	01	0	01	-	-	0	1	-	0	Reset regs (0 in U, V, W, P, Q) Feed X to U; Y to W;
S_1	1	S_2	00	0	00	11	11	0	0	-	0	X in U; Y in W; 0 in V, Q Feed $V(=0), U$ to comparator Note: A_lt_B is $0 < U$ If $(0 < U)$ go to S_2
S_1	0	S_3	00	0	00	11	11	0	0	-	0	X in U; Y in W; 0 in V, Q Feed $V(=0), U$ to comparator Note: A_lt_B is $0 < U$ If $(U \leq 0)$ return 0 (from S_3)
S_2	1	S_4	00	0	00	11	01	0	0	-	0	X in U; Y in W; 0 in V, Q Feed $V(=0), W$ to comparator Note: A_lt_B is $0 < W$ If $(0 < W)$ try to enter loop (at S_4)
S_2	0	S_3	00	0	00	11	01	0	0	-	0	X in U; Y in W; 0 in V, Q Feed $V(=0), W$ to comparator Note: A_lt_B is $0 < W$ If $(W \leq 0)$ return 0 (from S_3)
S_3 (return 0)	-	S_3	-	-	-	-	-	-	1	-	1	Reset all regs (in particular Q) Set *Done* to 1 and loop
S_4 (loop head)	1	S_5	00	0	00	01	11	0	0	-	0	U, V, W, Q contain X, 0, Y, 0 Feed W, U to comparator Note: A_lt_B is $W < U$ If $(W < U)$ go to S_5 (for `X = X-Y`)
S_4	0	S_6	00	0	00	01	11	0	0	-	0	U, V, W, Q contain X, 0, Y, 0 Feed W, U to comparator Note: A_lt_B is $W < U$ If $(W \geq U)$ go to S_6 (to check if $U \geq W$)
S_6	1	S_7	00	0	00	00	01	0	0	-	0	U, V, W, Q contain X, 0, Y, 0 Feed U, W to comparator Note: A_lt_B is $U < W$ If $(U < W)$ go to S_7 (for `Y = Y-X`)
S_6	0	S_8	00	0	00	00	00	0	0	0	0	U, V, W, Q contain X, 0, Y, 0 $(U \geq W)$ and $(W \geq U)$, i.e. $U = W$ Come out of while loop Feed $U, V(=0)$ to subtractor, $U - 0$ to P
S_8	-	S_9	00	0	00	00	00	1	0	0	0	U, V, W, Q unchanged, U in P Out of while loop Feed $U, V(=0)$ to subtractor, $U - 0$ to P Feed P to Q
S_9 (return X)	-	S_9	00	0	00	00	00	1	0	0	1	U, V, W unchanged, U in P, Q Out of while loop Feed $U, V(=0)$ to subtractor, $U - 0$ to P Feed P to Q and loop
S_5	-	S_{10}	00	0	00	00	01	0	0	0	0	U, V, W, Q contain X, 0, Y, 0 Feed U, W to subtractor, $U - W$ to P
S_{10}	-	S_4	11	0	00	-	-	0	0	-	0	U, V, W, Q unchanged, $U - W$ in P Feed P to U Go back to loop head (S_4)
S_7	-	S_{11}	00	0	00	01	11	0	0	0	0	U, V, W, Q contain X, 0, Y, 0 Feed W, U to subtractor, $W - U$ to P
S_{11}	-	S_4	00	0	11	-	-	0	0	-	0	U, V, W, Q unchanged, $W - U$ in P Feed P to W Go back to loop head (S_4)

4. Consider the Boolean function $F(x_1, x_2, x_3, x_4, x_5) = ((x_1 + x_2').(x_3 + x_4')) \oplus (x_1 \oplus x_5)$

   (a) *[5 marks]* Construct an ROBDD for F using the variable order $x_1 < x_2 < x_3 < x_4 < x_5$.
   No partial marks will be given for incorrect ROBDDs or for graphs that are not ROBDDs.

Answer:

We use the fact that $1 \oplus v = v'$ and $0 \oplus v = v$ for any Boolean variable v. Therefore, using Shannon decomposition, we get $F = \mathsf{ite}(x_1, F_1, F_0)$, where
$F_1 = (x_3 + x_4') \oplus x_5'$ and $F_0 = (x_2'.(x_3 + x_4')) \oplus x_5$.

In turn, $F_1 = \mathsf{ite}(x_3, x_5, x_4' \oplus x_5')$, and $F_0 = \mathsf{ite}(x_2, x_5, F_{00})$, where $F_{00} = (x_3 + x_4') \oplus x_5$.
Continuing Shannon decomposition, $F_{00} = \mathsf{ite}(x_3, x_5', x_4' \oplus x_5)$.
Finally, $x_4' \oplus x_5' = \mathsf{ite}(x_4, x_5', x_5)$ and $x_4' \oplus x_5 = \mathsf{ite}(x_4, x_5, x_5')$.
This gives the ROBDD shown in Fig. 2.

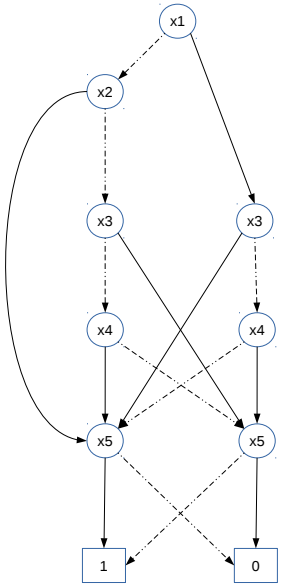

Figure 2: ROBDD for 4(a)

(b) *[5 marks]* Determine if it is possible to reduce the number of nodes in the ROBDD obtained above using complement edges such that

- **there is no 1-leaf in the BDD** *(note that this is different from examples seen in class and in the practice questions)*, and
- **no solid edge (or 1-edge) is complemented**.

"Yes/No" answers without adequate justification will fetch 0 marks.

Answer:

We start by replacing the 1-leaf with a 0-leaf and inserting bubbles on edges that originally led to the 1-leaf. This is shown in Fig. 3.

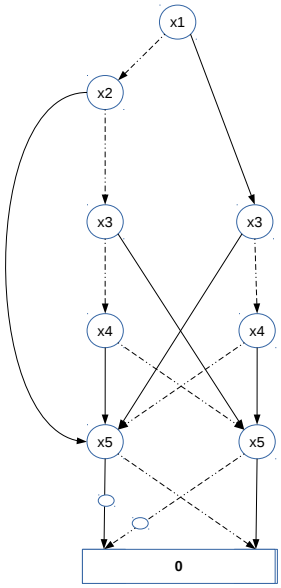

Figure 3: Initial complement bubbles for 4(b)

Next, we successively remove bubbles from the solid edges, as shown in Figs. 4, 5, 6.

Notice that the above transformations result in an ROBDD *with a bubble on the root edge*. The question did not state anything about whether a bubble is allowed on the root edge. So there are two possible answers to this question. In each case, you should specify whether you are assuming bubbles on root edge are allowed or not.

- Assuming the root edge is treated like a solid edge, and hence bubbles are disallowed on it, the above transformations cannot be done, and hence no reduction in the number of nodes is allowed.
- Assuming bubbles are allowed on the root edge, which is treated differently from a solid edge, the above transformations are allowed, and we get the diagram in Fig. 6. By merging nodes *with the same functionality* (only such nodes can be merged in an ROBDD) in this figure, we finally get the ROBDD in Fig. 7. Clearly, this has fewer nodes than the original ROBDD in Fig. 3.

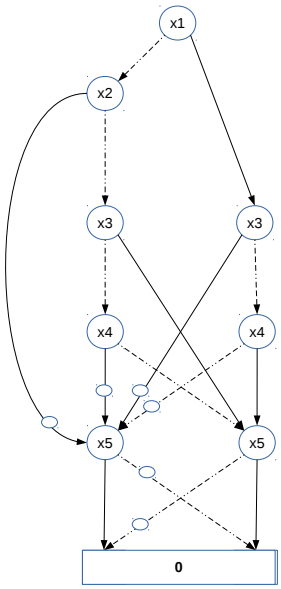

Figure 4: Moving bubbles out of solid edges (step 1)

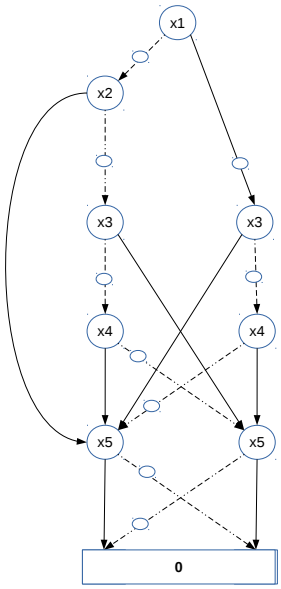

Figure 5: Moving bubbles out of solid edges (step 2)

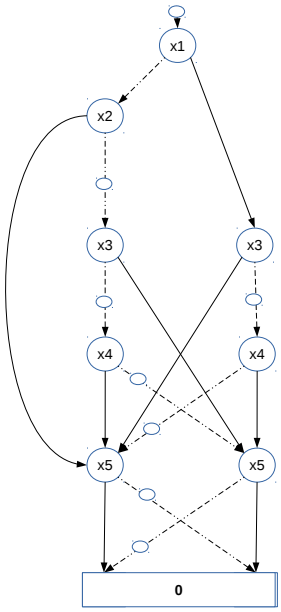

Figure 6: Moving bubbles out of solid edges (step 3)

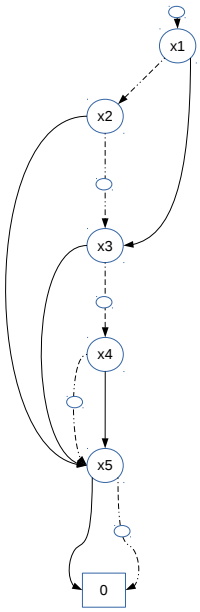

Figure 7: ROBDD with complement edges and complement root edge

(c) *[7 marks]* Now consider the Boolean function $G(x_1, x_5) = (x_1 \oplus x_5)$. Suppose we are interested in the values of $F(x_1, x_2, x_3, x_4, x_5)$ only when $G(x_1, x_5)$ evaluates to 1. In other words, the values of F when G evaluates to 0 may be treated as don't care. Simplify the ROBDD for F obtained in part (a) considering these don't cares.

You must use Shannon decomposition and suitable termination cases, and systematically show how you obtain the simplification. Simplify producing a simplified ROBDD without any steps will fetch 0 marks.

[Hint: Think about how we recursively applied different operations on ROBDDs using the `ite` *formulation]*

Answer:

Note that this problem is related to but slightly different from what was discussed in class just prior to mid-sem. In the problem discussed in class prior to mid-sem, we were given a function f and a function g, and we wanted to simplify f by treating the value of the simplified function as don't-care when g evaluated to 1 and f evaluated to 0. In contrast, in the current problem, we want to simplify f by treating the value of the simplified function as don't care whenever g evaluates to 0.

The recursive formulation is as follows (similar to that discussed in class prior to mid-sem):
Suppose $f = \mathsf{ite}(v, f_1, f_0)$ and $g = \mathsf{ite}(v, g_1, g_0)$.
Then $\mathsf{MySimplify}(f, g) = \mathsf{ite}(v, \mathsf{MySimplify}(f_1, g_1), \mathsf{MySimplify}(f_0, g_0))$.

The termination cases (not the same as that discussed in class prior to mid-sem) are:

- $\mathsf{MySimplify}(f, f) = 1$
- $\mathsf{MySimplify}(f, f') = 0$
- $\mathsf{MySimplify}(f, 1) = f$
- $\mathsf{MySimplify}(f, 0) =$ don't care — this case shouldn't arise if you have simplified properly.
- $\mathsf{MySimplify}(1, f) = 1$
- $\mathsf{MySimplify}(0, f) = 1$.

Using the above, and using the notation used in the solution for 4(a), we get:
$\mathsf{MySimplify}(F, x_1 \oplus x_5) = \mathsf{ite}(x_1, \mathsf{MySimplify}(F_1, x_5'), \mathsf{MySimplify}(F_0, x_5))$.

Using the expression for F_1 from the solution for 4(a), we get
$\mathsf{MySimplify}(F_1, x_5') = \mathsf{ite}(x_3, \mathsf{MySimplify}(x_5, x_5'), \mathsf{MySimplify}(x_4' \oplus x_5', x_5'))$
$= \mathsf{ite}(x_3, 0, \mathsf{ite}(x_4, \mathsf{MySimplify}(x_5', x_5'), \mathsf{MySimplify}(x_5, x_5')) = \mathsf{ite}(x_3, 0, \mathsf{ite}(x_4, 1, 0)) = \mathsf{ite}(x_3, 0, x_4)$.

Similarly, using the expression for F_0 and F_{00} from the solution for 4(a), we get
$\mathsf{MySimplify}(F_0, x_5) = \mathsf{ite}(x_2, \mathsf{MySimplify}(x_5, x_5), \mathsf{MySimplify}(F_{00}, x_5))$
$= \mathsf{ite}(x_2, 1, \mathsf{ite}(x_3, \mathsf{MySimplify}(x_5', x_5), \mathsf{MySimplify}(x_4' \oplus x_5, x_5)))$
$= \mathsf{ite}(x_2, 1, \mathsf{ite}(x_3, 0, \mathsf{ite}(x_4, \mathsf{MySimplify}(x_5, x_5), \mathsf{MySimplify}(x_5', x_5))))$
$= \mathsf{ite}(x_2, 1, \mathsf{ite}(x_3, 0, \mathsf{ite}(x_4, 1, 0))) = \mathsf{ite}(x_2, 1, \mathsf{ite}(x_3, 0, x_4)$.

Using the above Shannon decomposition for $\mathsf{MySimplify}(F, G)$, we get the ROBDD as shown in Fig. 8. Note that we **MUST** use the same variable order as in part (a) in order to simplify the ROBDD in part (a).

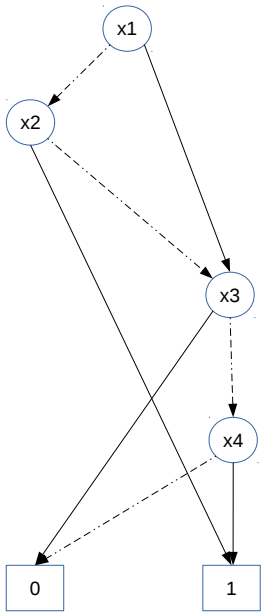

Figure 8: ROBDD for MySimplify(F, G)