CS226 Practice Problem Set 3 (Spring 2016) Solutions

1. $F = \mathsf{ite}(x_5, f_5', f_5)$, where $f_5 = (x_2 \oplus x_3).((x_1 \oplus x_3') + x_4)$.
   In turn, $f_5 = \mathsf{ite}(x_4, (x_2 \oplus x_3), (x_2 \oplus x_3).(x_1 \oplus x_3'))$
   Furthermore, $x_2 \oplus x_3 = \mathsf{ite}(x_2, x_3', x_3)$ and
   $(x_2 \oplus x_3).(x_1 \oplus x_3') = \mathsf{ite}(x_2, x_3'.(x_1 \oplus x_3'), x_3.(x_1 \oplus x_3'))$.
   The latter can be re-written as: $\mathsf{ite}(x_2, \mathsf{ite}(x_3, 0, x_1'), \mathsf{ite}(x_3, x_1, 0))$.
   To get the ite expression for f_5', we simply use the recursive formulation used for complementing an ROBDD. Thus, $f_5' = \mathsf{ite}(x_4, (x_2 \oplus x_3)', ((x_2 \oplus x_3).(x_1 \oplus x_3'))')$. Applying this recursively, we get $(x_2 \oplus x_3)' = \mathsf{ite}(x_2, x_3, x_3')$ and
   $((x_2 \oplus x_3).(x_1 \oplus x_3'))' = \mathsf{ite}(x_2, \mathsf{ite}(x_3, 1, x_1), \mathsf{ite}(x_3, x_1', 1))$.
   The ROBDD can now be constructed as shown in Fig. 1.

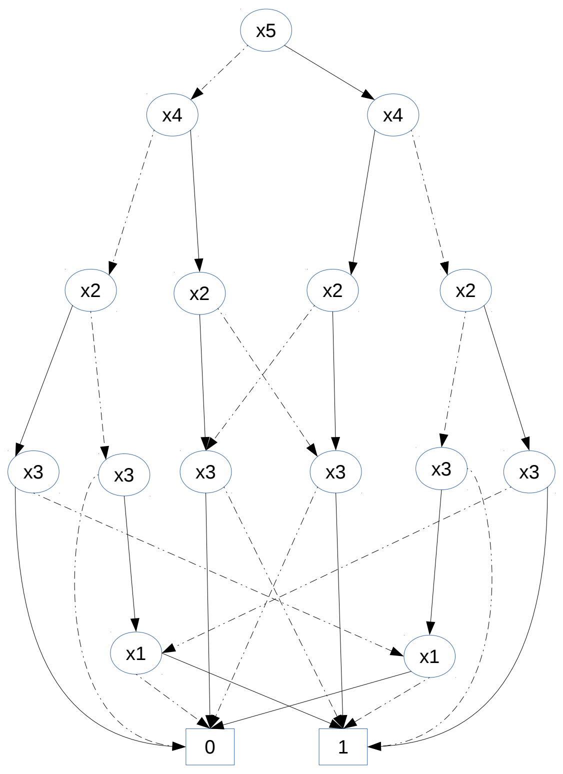
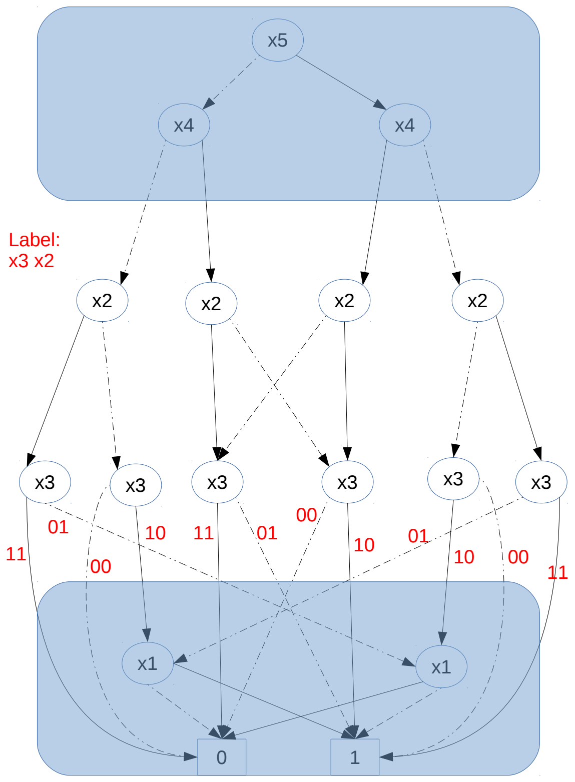


Figure 1: ROBDD for 1(a)

The following figures show how the variable ordering of x_2 and x_3 can be swapped in the ROBDD. The red labels in Fig. 2 give the labels of the outgoing edges as values of x_3x_2. We need to make sure that outgoing edges with these labels are also available after swapping the order of x_2 and x_3. The blue shaded regions must stay unchanged. The result is shown in Fig. 3.



(a)

Figure 2: ROBDD from 1(a) with labels of outgoing edges (in red)

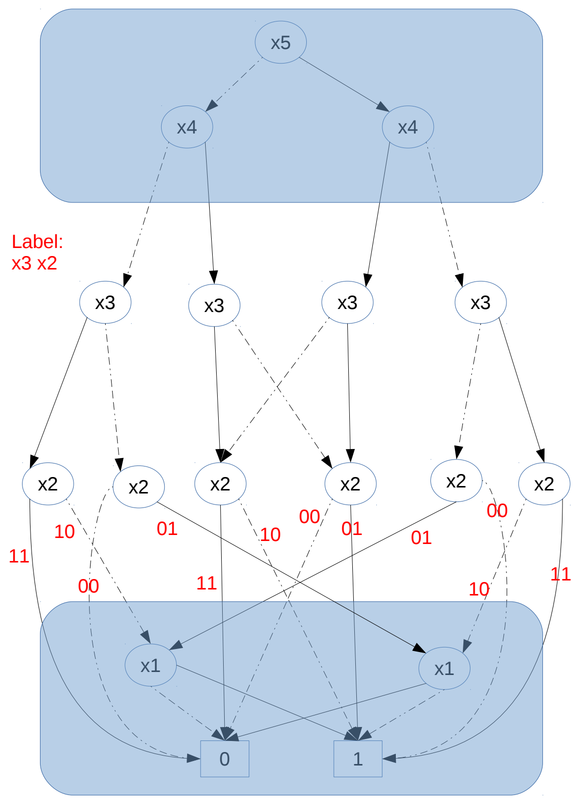


Figure 3: ROBDD solution for 1(b).

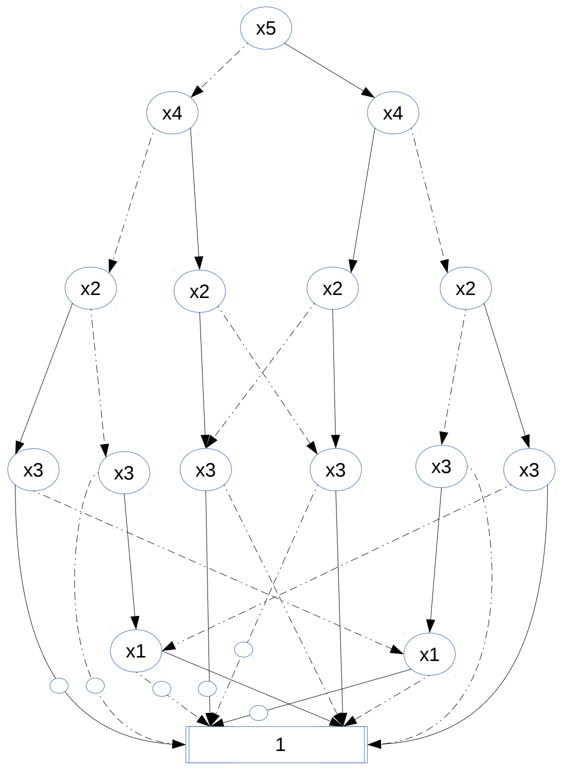


Figure 4: Inserting bubbles on edges to 0-leaf

(b) This requires a sequence of transformations.

   i. Insert bubbles on all edges leading to the 0-leaf. This is shown in Fig. 4.

   ii. Now shift bubbles out from solid edges, going from lower levels of the BDD upwards. You should continue this until no solid edges are left with bubbles. Note that, in general, this can result in an edge above the root node with a bubble on it (in this example, this doesn't happen though). This is shown in Fig. 5, Fig. 6, and Fig. 7.

   iii. Next, cancel even numbers of bubbles on all dotted edges. This is shown in Fig. 8.

   The end-result of the sequence of transformations is shown in Fig. 9.

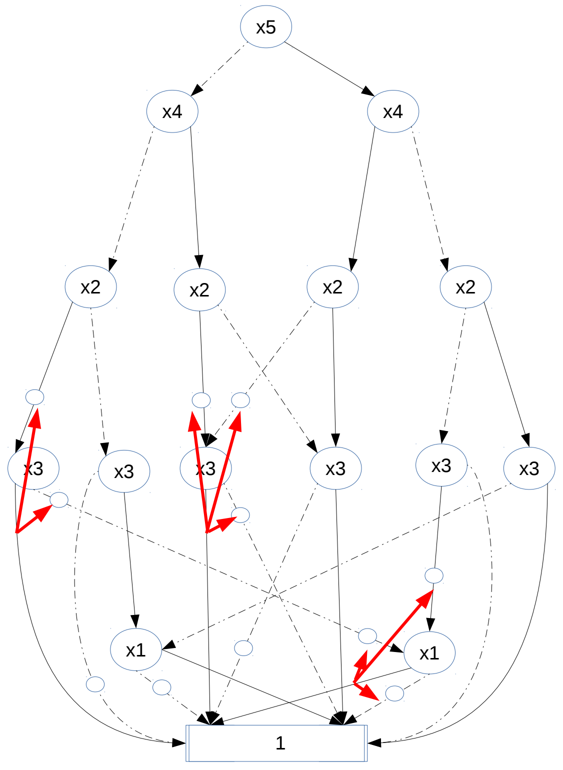


Figure 5: Moving bubbles out of solid edges (step 1).

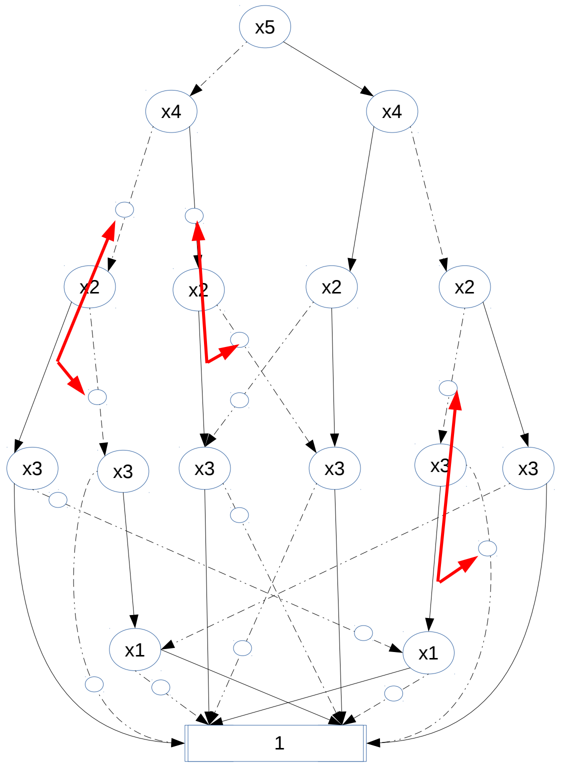


Figure 6: Moving bubbles out of solid edges (step 2).

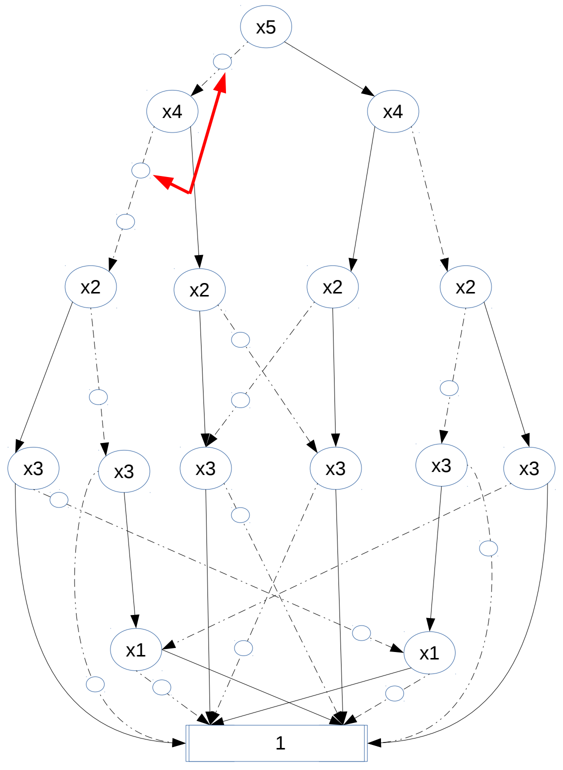


Figure 7: Moving bubbles out of solid edges (step 3).

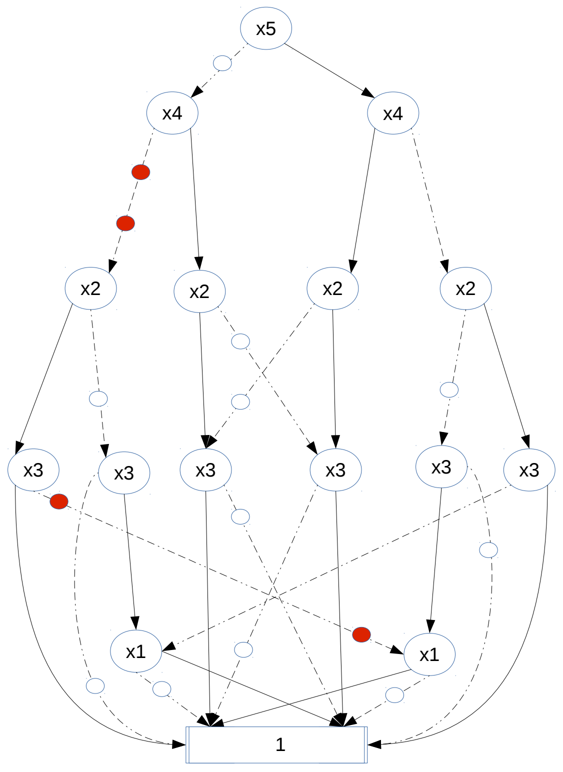


Figure 8: Cancelling even numbers of bubbles.

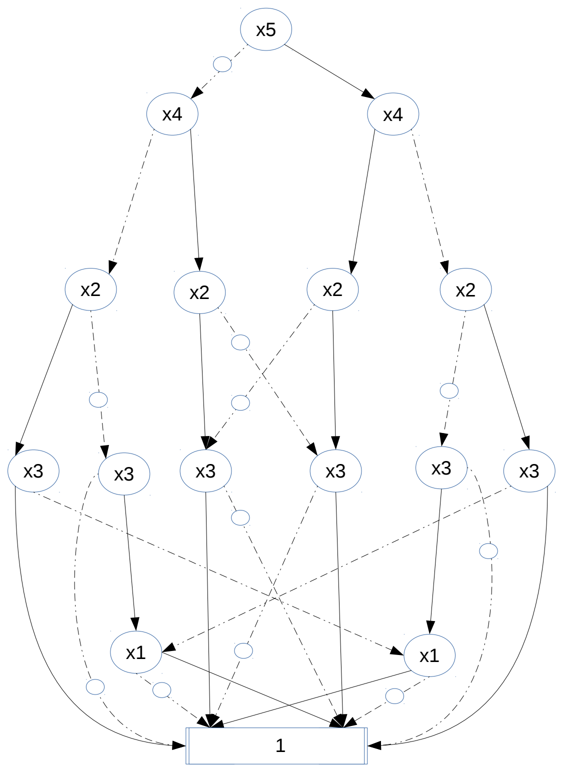


Figure 9: The end result

(c) Fig. 10 shows nodes with the same functionality coloured with the same colour. On merging these nodes, we get the ROBDD with complement edges shown in Fig. 11.

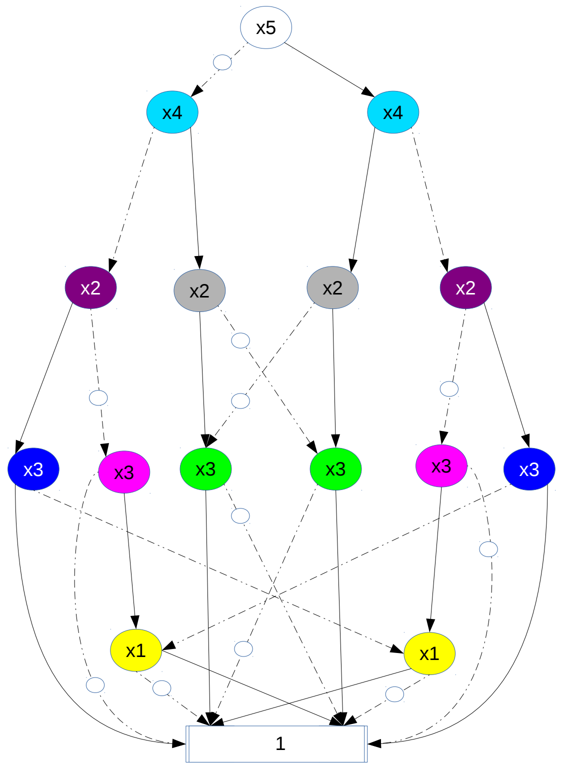


Figure 10: Identifying functionally equivalent nodes.

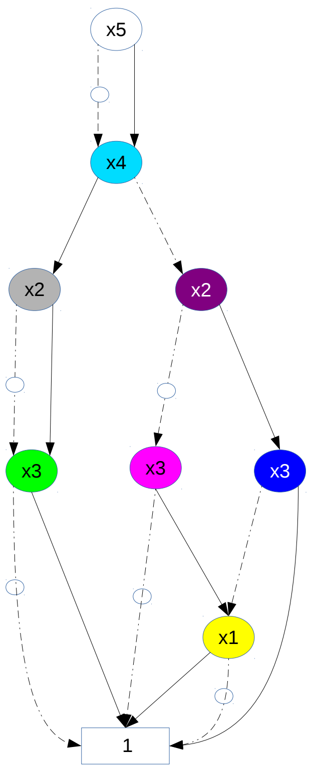


Figure 11: ROBDD with complement edges

2. Observe from the given ROBDD (shown in Fig. 12) that when $a = 1$, then regardless of the value of b, if $c = 0$, the same node (coloured node labeled d) is reached in the ROBDD.

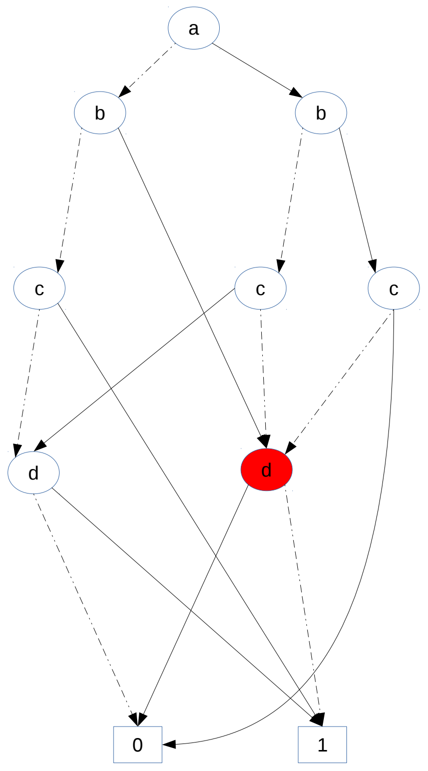


Figure 12: Original ROBDD

This suggests that when $a = 1$, if we split on c first, then the 0-edge from the node labeled c can directly reach the coloured node labeled d without splitting on b. On the other hand, when $a = 1$, and $c = 0$, then depending on whether $b = 0$ or $b = 1$, the appropriate co-factors have to be chosen. This suggests that instead of having one node to split on b, and two nodes to split on c (as in Fig. 12), we can have only one node to split on c and one node to split on b. The resulting BDD is shown in Fig. 13.

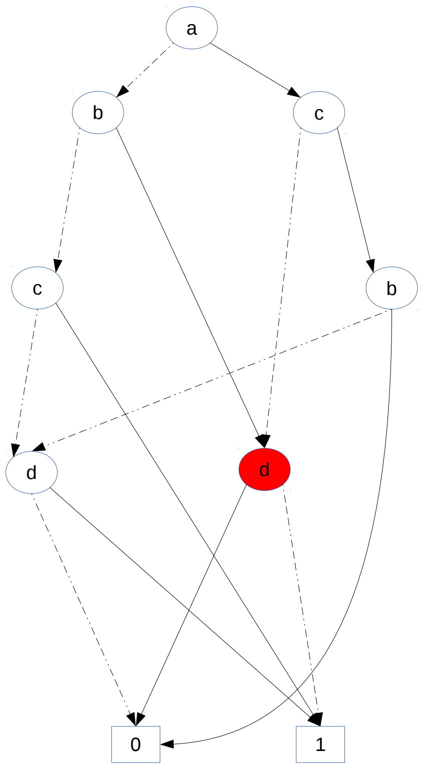


Figure 13: BDD with different orders on different paths

3. Recall that we discussed in class how the ordering of variables affects the size of the ROBDD. In particular, if the variable order is such that you can't determine whether the function will evaluate to 1 or 0 even after knowing the values of the first few variables in the order, then the ROBDD must split into sufficiently many parts after reading these variables, effectively remembering something about the values of the variables read so far. This is then used to determine the value of the function after the remaining variables's values are read.

   The example we saw in class (and given on the slides on the course webpage) is one such function: $x_1.x_2 + x_3.x_4 + x_5.x_6 + \cdots x_{2n-1}.x_{2n}$, where the variable order $x_1 < x_3 < \cdots x_{2n-1} < x_2 < x_4 < \cdots x_{2n}$ requires that you remember the 2^n combinations of values of $x_1, x_3, \ldots x_{2n-1}$, before you can decide the value of the function on reading the values of $x_2, x_4, \ldots$.

   XOR functions are notoriously good (or bad, as you see it) in this respect: you cannot determine the value of the function until you have read in all the variables. However, XORs are not the only functions with such a property. In this problem, however, using an XOR function is sufficient. In particular, $(x_1 \oplus x_4).(x_2 \oplus x_3)$ requires 11 nodes (including the leaves) with the order $x_1 < x_2 < x_3 < x_4$, while each of $(x_1 \oplus x_4)$ and $(x_2 \oplus x_3)$ requires 5 nodes (including the leaves).

   Draw the ROBDDs yourself to convince yourself of the above.

4. For algorithm 1, we can follow the sequence of steps given below. Here, we have chosen to store X in register U, Z in register V and Y in register W.

CurrSt	A_lt_B	NextSt	M_u	M_v	M_w	M_a	M_b	M_q	Reset	doAdd	Comment
S_1	-	S_2	01	0	01	-	-	-	1	-	Reset regs (0 in all regs), feed X,Y to regs U,W
S_2	-	S_3	00	0	00	00	01	0	0	1	X in U; Y in W; 0 in V and P feed U,W to adder
S_3	1	S_4	00	1	00	00	01	0	0	0	$U+W$ in P; U,V,W unchanged $U < W$: entering while loop feed P to V, feed U,W to subtractor
S_3	0	S_7	00	0	00	11	00	0	0	1	$U+W$ in P; U,V,W unchanged $U \geq W$: exiting while loop feed V,V to adder
S_4	-	S_5	11	0	00	01	01	0	0	1	$U+W$ in V; V,W unchanged; $U-W$ in P Inside while loop feed P to U feed W,W to adder
S_5	-	S_6	00	0	11	-	-	0	0	-	$2W$ in P; $U_{prev} - W$ in U; V,W unchanged Inside while loop feed P to W
S_6	-	S_3	00	0	00	00	01	0	0	1	$2W$ in W; U,V unchanged; garbage in PP Inside while loop feed U, W to adder (like S_2)
S_7	-	S_8	00	0	00	11	00	1	0	1	$2V$ in P; U,V,W unchanged Out of while loop: feed P to Q, and V,V to adder
S_8	-	S_8	00	0	00	11	00	1	0	1	$2V$ in P and in Q; U,V,W unchanged Keep looping: feed P to Q, and V,V to adder

   Given that there are 8 states, three state bits should suffice. You could encode S_1 as 000, S_2 as 001 or use your favourite encoding. By replacing each state S_i with its encoding in the above table, and by removing the "Comment" column, we get the state transition table of the controller required to implement algorithm 1 with the given datapath.

To implement algorithm 2, we can use the following sequence of steps. Once again, we have chosen to store X in register U, Z in register V and Y in register W.

CurrSt	A_lt_B	NextSt	M_u	M_v	M_w	M_a	M_b	M_q	Reset	doAdd	Comment
S_1	-	S_2	01	0	01	-	-	-	1	-	Reset regs (0 in all regs), feed X,Y to regs U,W
S_2	-	S_3	00	0	00	00	11	0	0	1	X in U; Y in W; 0 in V and P In do-while loop feed U,U to adder
S_3	-	S_4	11	0	00	-	-	0	0	-	$2U$ in P; U, V, W unchanged In do-while loop feed P to U
S_4	-	S_5	00	0	00	00	01	0	0	0	$2U_{prev}$ in U; V, W unchanged; garbage in P In do-while loop feed U,W to subtractor
S_5	-	S_6	00	1	00	-	-	0	0	-	$U - W$ in P; U, V, W unchanged Inside do-while loop feed P to V
S_6	-	S_7	00	0	00	11	00	0	0	1	$U - W$ in V; U, W unchanged; garbage in P Inside do-while loop feed V, V to adder
S_7	-	S_8	00	0	11	11	11	0	0	-	$2V$ in P; U, V, W unchanged Inside do-while loop feed P to W, and V, U to adder/sub
S_8	1	S_3	00	0	00	00	11	0	0	1	$2V$ in W; U, V unchanged; garbage in P Inside do-while loop: feed U, U to adder (as in S_2)
S_8	0	S_9	00	0	00	11	00	0	0	1	$2V$ in W; U, V unchanged; garbage in P Out of do-while loop: feed V, V to adder
S_9	-	S_{10}	00	0	00	11	00	1	0	1	$2V$ in P; U, V, W unchanged Out of do-while loop: feed P to Q, and V, V to adder
S_{10}	-	S_{10}	00	0	00	11	00	1	0	1	$2V(= W)$ in P and Q; U, V, W unchanged Keep looping: feed P to Q, and V, V to adder

There are 10 states now and four state bits should suffice. You can choose your encoding of states to complete the controller's state transition table.