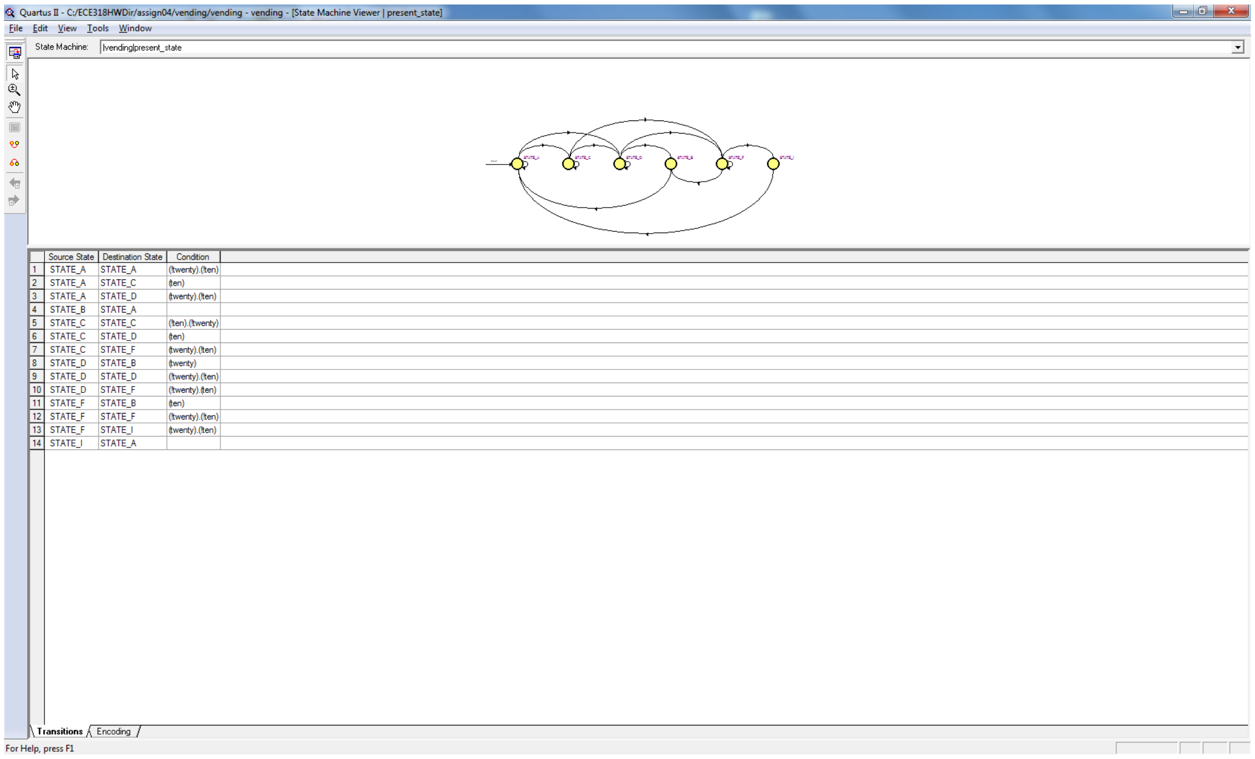
ECE318
Assignment 4 Solution

Q1.

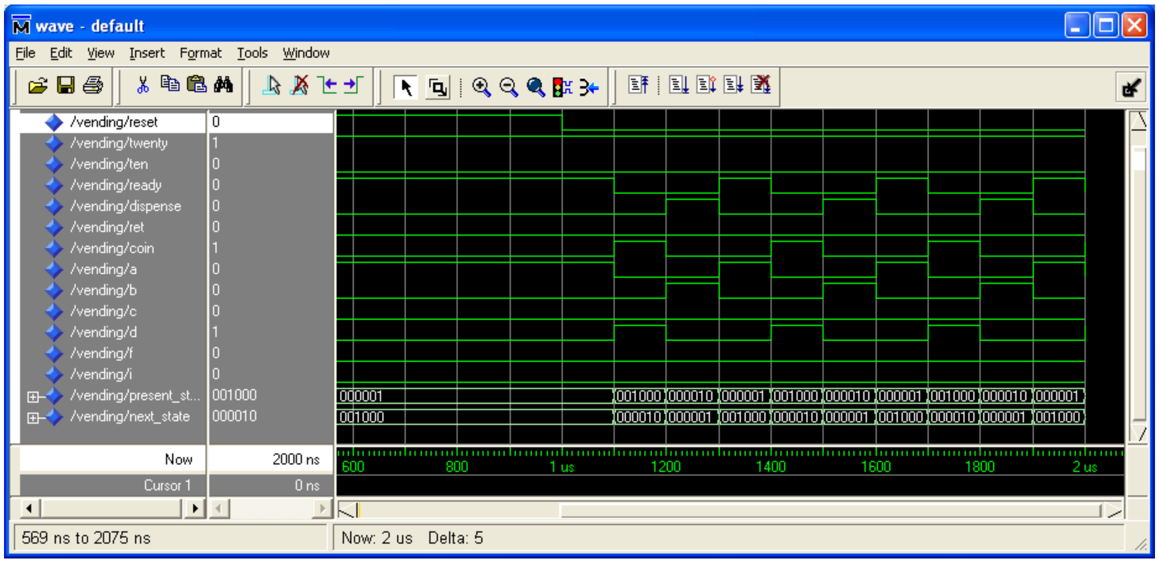
Code Listing:

```
-- New code that follows the Altera guidelines for state machines:
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
entity vending is
port (clock, reset, twenty, ten : in std_logic;
      A, B, C, D, F, I : buffer std_logic;
      ready, dispense, ret, coin : out std_logic);
end entity vending;


architecture asm of vending is
   TYPE STATE_TYPE IS (STATE_A, STATE_B, STATE_C, STATE_D, STATE_F,
STATE_I);
   signal A_next, B_next, C_next, D_next, F_next, I_next : std_logic;
   signal present_state, next_state : STATE_TYPE;
begin

   A <= '1' when present_state = STATE_A else '0';
   B <= '1' when present_state = STATE_B else '0';
   C <= '1' when present_state = STATE_C else '0';
   D <= '1' when present_state = STATE_D else '0';
   F <= '1' when present_state = STATE_F else '0';
   I <= '1' when present_state = STATE_I else '0';

-- Use concurrent statements to describe the state
-- transitions
   A_next <= (B or I) or (not twenty and not ten and A);
   B_next <= (D and twenty) or (F and ten);
   C_next <= (A and ten) or (not twenty and not ten and C);
   D_next <= (A and twenty) or (C and ten) or (not twenty and not ten and D);
   F_next <= (C and twenty) or (D and ten) or (not twenty and not ten and F);
   I_next <= F and twenty;

   next_state <= STATE_A when A_next = '1' else
   STATE_B when B_next = '1' else
   STATE_C when C_next = '1' else
   STATE_D when D_next = '1' else
   STATE_F when F_next = '1' else
   STATE_I;
-- Use concurrent statements to describe the
-- outputs
   ready <= A;
   dispense <= B;
   ret <= I;
   coin <= C or D or F;

   seq: process(clock,reset) is
   begin
     if (reset = '1') then
       present_state <= STATE_A;
     elsif (rising_edge(clock)) then
       present_state <= next_state;
    end if;
end process seq;

end architecture asm;
```


Quartus II - C:/ECE318HWDir/assign04/vending/vending - vending - [State Machine Viewer | present_state]
File Edit View Tools Window
State Machine: |vending|present_state
Transitions
Encoding
For Help, press F1

	Source State	Destination State	Condition
1	STATE_A	STATE_A	(!twenty).(!ten)
2	STATE_A	STATE_C	(ten)
3	STATE_A	STATE_D	(twenty).(!ten)
4	STATE_B	STATE_A	
5	STATE_C	STATE_C	(!ten).(!twenty)
6	STATE_C	STATE_D	(ten)
7	STATE_C	STATE_F	(twenty).(!ten)
8	STATE_D	STATE_B	(twenty)
9	STATE_D	STATE_D	(!twenty).(!ten)
10	STATE_D	STATE_F	(!twenty).(ten)
11	STATE_F	STATE_B	(ten)
12	STATE_F	STATE_F	(!twenty).(!ten)
13	STATE_F	STATE_I	(twenty).(!ten)
14	STATE_I	STATE_A	

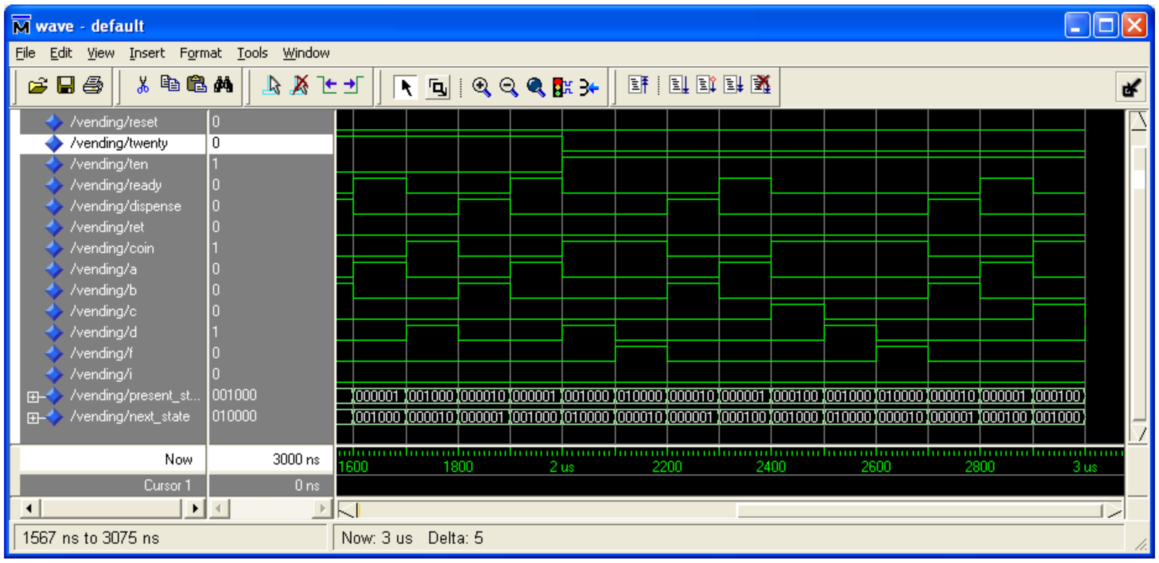
Simulation Results:



```
/vending/reset          0
/vending/twenty         1
/vending/ten            0
/vending/ready          0
/vending/dispense       0
/vending/ret            0
/vending/coin           1
/vending/a              0
/vending/b              0
/vending/c              0
/vending/d              1
/vending/f              0
/vending/i              0
/vending/present_st...  001000   000001 001000 000010 000001 001000 000010 000001 001000 000010 000001
/vending/next_state     000010   001000 000010 000001 001000 000010 000001 001000 000010 000001 001000
Now                     2000 ns
Cursor 1                0 ns
569 ns to 2075 ns       Now: 2 us  Delta: 5
```

Figure 1

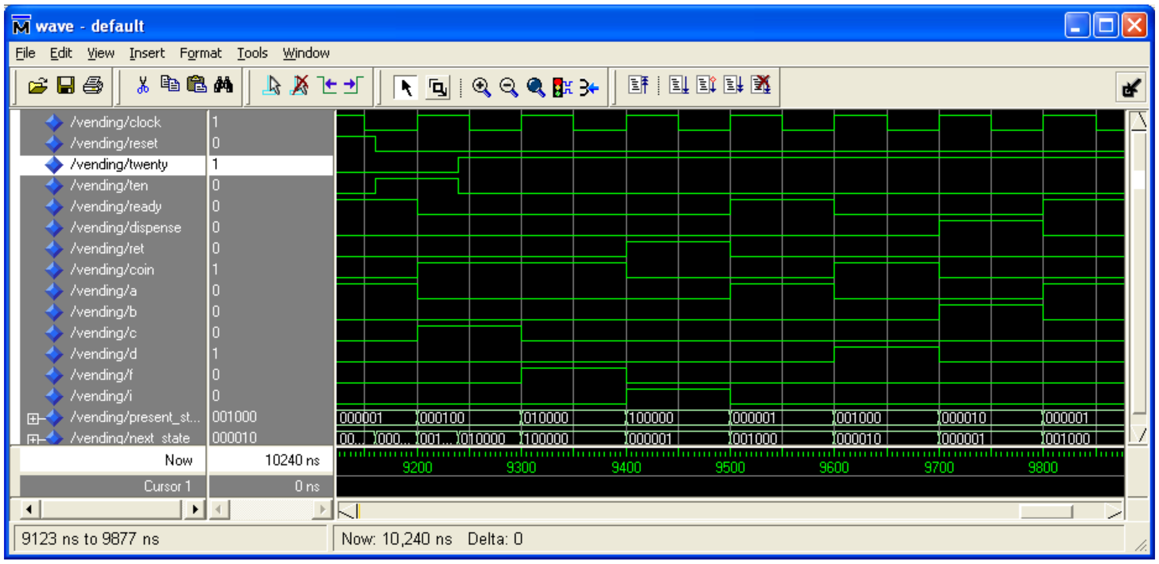
Recall that the states have been coded in the order “IFDCBA”. In Figure 1, the machine is put into the reset state which is state A, coded as “000001”. Then, the twenty line is set to high and it is observed that the machine goes through the correct sequence of states.



```
/vending/reset          0
/vending/twenty         0
/vending/ten            1
/vending/ready          0
/vending/dispense       0
/vending/ret            0
/vending/coin           1
/vending/a              0
/vending/b              0
/vending/c              0
/vending/d              1
/vending/f              0
/vending/i              0
/vending/present_st...  001000   000001 001000 000010 000001 001000 010000 000010 000001 000100 001000 010000 000010 000001 000100
/vending/next_state     010000   001000 000010 000001 001000 010000 000010 000001 000100 001000 010000 000010 000001 000100 001000
Now                     3000 ns
Cursor 1                0 ns
1567 ns to 3075 ns      Now: 3 us  Delta: 5
```

Figure 2

In Figure 2, there is a transition from inserting twenty cent coins to inserting ten cent coins and again, it is observed that the machine functions correctly.



/vending/clock	1
/vending/reset	0
/vending/twenty	1
/vending/ten	0
/vending/ready	0
/vending/dispense	0
/vending/ret	0
/vending/coin	1
/vending/a	0
/vending/b	0
/vending/c	0
/vending/d	1
/vending/f	0
/vending/i	0
/vending/present_st...	001000
/vending/next_state	000010
Now	10240 ns
Cursor 1	0 ns

9123 ns to 9877 ns

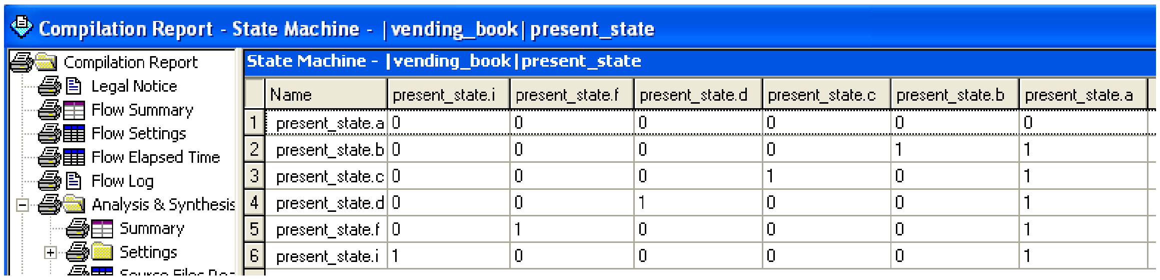
Now: 10,240 ns Delta: 0

Figure 3

Finally, I wanted to see that the machine could be put in the state "100000", i.e., the state I is reached. This took a while to get but it can be seen that the machine does get into this state as shown in Figure 3. Can you figure out how?

Quartus Simulation:

By default the coding in Quartus is one-hot for this state machine. The encoding of the states is given in Figure 4.

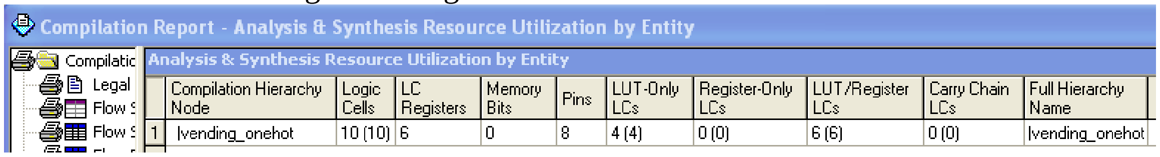


State Machine - |vending_book|present_state

	Name	present_state.i	present_state.f	present_state.d	present_state.c	present_state.b	present_state.a
1	present_state.a	0	0	0	0	0	0
2	present_state.b	0	0	0	0	1	1
3	present_state.c	0	0	0	1	0	1
4	present_state.d	0	0	1	0	0	1
5	present_state.f	0	1	0	0	0	1
6	present_state.i	1	0	0	0	0	1

Figure 4

The resources used are given in Figure 5.

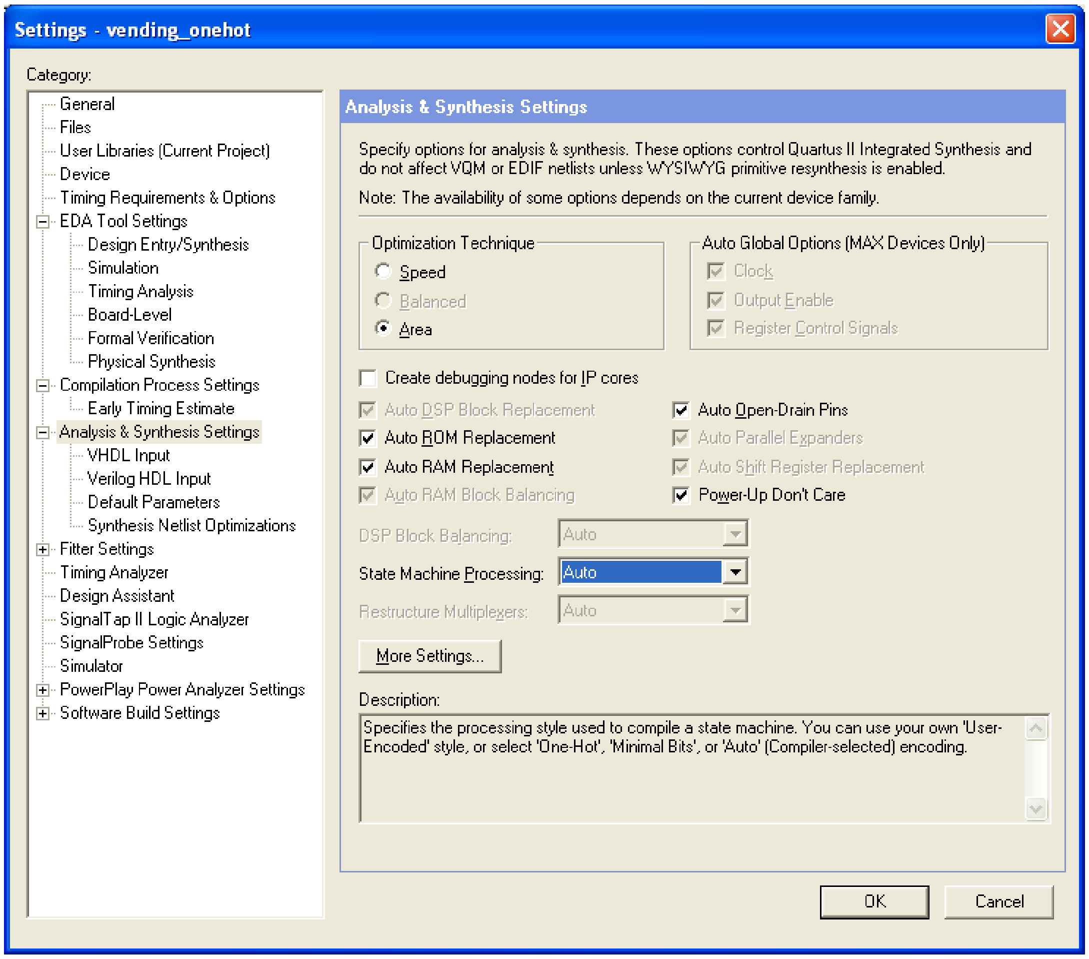

Compilation Report - Analysis & Synthesis Resource Utilization by Entity
Compilatic
Legal
Flow S
Flow S

Analysis & Synthesis Resource Utilization by Entity

	Compilation Hierarchy Node	Logic Cells	LC Registers	Memory Bits	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs	Carry Chain LCs	Full Hierarchy Name
1	|vending_onehot	10 (10)	6	0	8	4 (4)	0 (0)	6 (6)	0 (0)	|vending_onehot

Figure 5

The state encoding can be changed in Quartus by going to the Assignments-Settings and the “Analysis by Synthesis” tab and then choosing a different option than “Auto” as shown in Figure 6.


Settings - vending_onehot

Category:

- General
- Files
- User Libraries (Current Project)
- Device
- Timing Requirements & Options
- EDA Tool Settings
  - Design Entry/Synthesis
  - Simulation
  - Timing Analysis
  - Board-Level
  - Formal Verification
  - Physical Synthesis
- Compilation Process Settings
  - Early Timing Estimate
- Analysis & Synthesis Settings
  - VHDL Input
  - Verilog HDL Input
  - Default Parameters
  - Synthesis Netlist Optimizations
- Fitter Settings
- Timing Analyzer
- Design Assistant
- SignalTap II Logic Analyzer
- SignalProbe Settings
- Simulator
- PowerPlay Power Analyzer Settings
- Software Build Settings

Analysis & Synthesis Settings

Specify options for analysis & synthesis. These options control Quartus II Integrated Synthesis and do not affect VQM or EDIF netlists unless WYSIWYG primitive resynthesis is enabled.

Note: The availability of some options depends on the current device family.

Optimization Technique: Speed [✓ Area] Balanced

Auto Global Options (MAX Devices Only): [✓] Clock [✓] Output Enable [✓] Register Control Signals

[ ] Create debugging nodes for IP cores

[✓] Auto DSP Block Replacement

[✓] Auto ROM Replacement

[✓] Auto RAM Replacement

[✓] Auto RAM Block Balancing

[✓] Auto Open-Drain Pins

[✓] Auto Parallel Expanders

[✓] Auto Shift Register Replacement

[✓] Power-Up Don't Care

DSP Block Balancing: Auto

State Machine Processing: Auto

Restructure Multiplexers: Auto

More Settings...

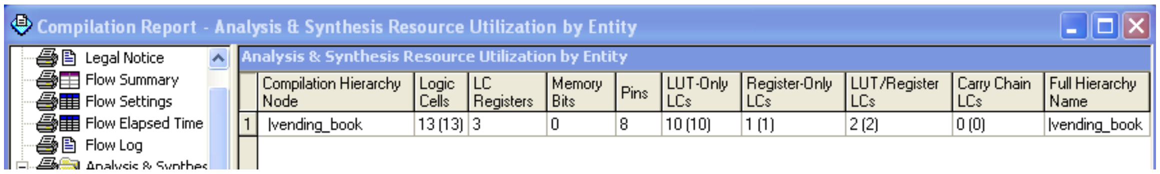
Description:

Specifies the processing style used to compile a state machine. You can use your own 'User-Encoded' style, or select 'One-Hot', 'Minimal Bits', or 'Auto' (Compiler-selected) encoding.

OK Cancel

Figure 6

Here are the new state assignments and resources with the new encoding. As expected, the number of registers required is now reduced to 3. However, the total number of cells required to implement the state machine has increased from 10 to 13.

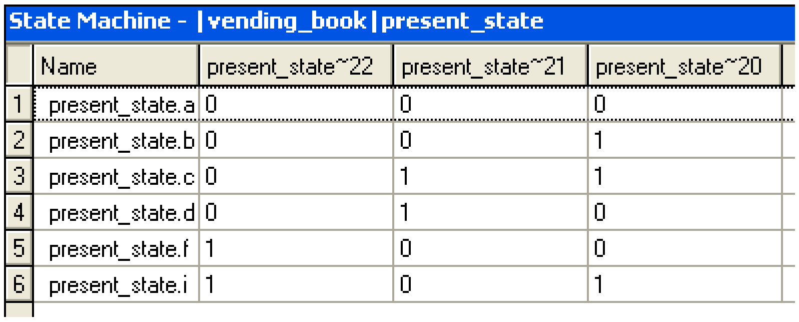

Compilation Report - Analysis & Synthesis Resource Utilization by Entity

- Legal Notice
- Flow Summary
- Flow Settings
- Flow Elapsed Time
- Flow Log
- Analysis & Synthes

Analysis & Synthesis Resource Utilization by Entity

	Compilation Hierarchy Node	Logic Cells	LC Registers	Memory Bits	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs	Carry Chain LCs	Full Hierarchy Name
1	|vending_book	13 (13)	3	0	8	10 (10)	1 (1)	2 (2)	0 (0)	|vending_book

Figure 7

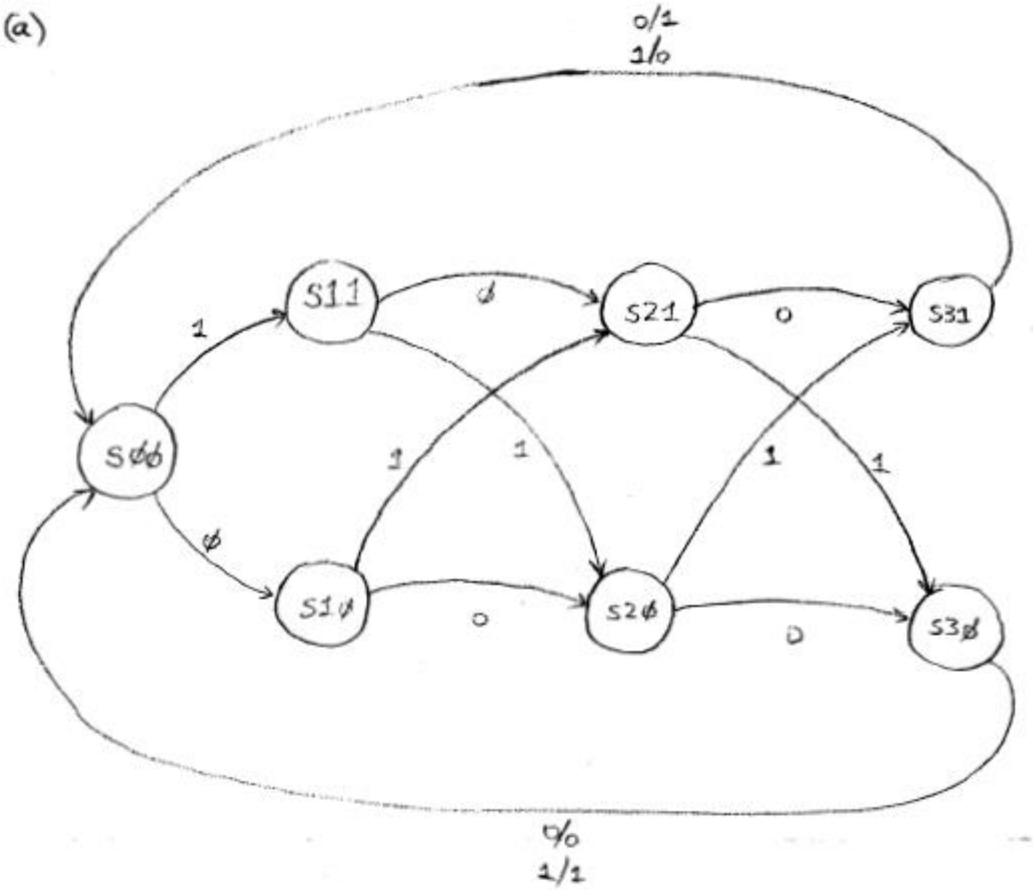


State Machine - |vending_book|present_state

	Name	present_state~22	present_state~21	present_state~20
1	present_state.a	0	0	0
2	present_state.b	0	0	1
3	present_state.c	0	1	1
4	present_state.d	0	1	0
5	present_state.f	1	0	0
6	present_state.i	1	0	1

Figure 8

Q2(a)



Notation :-

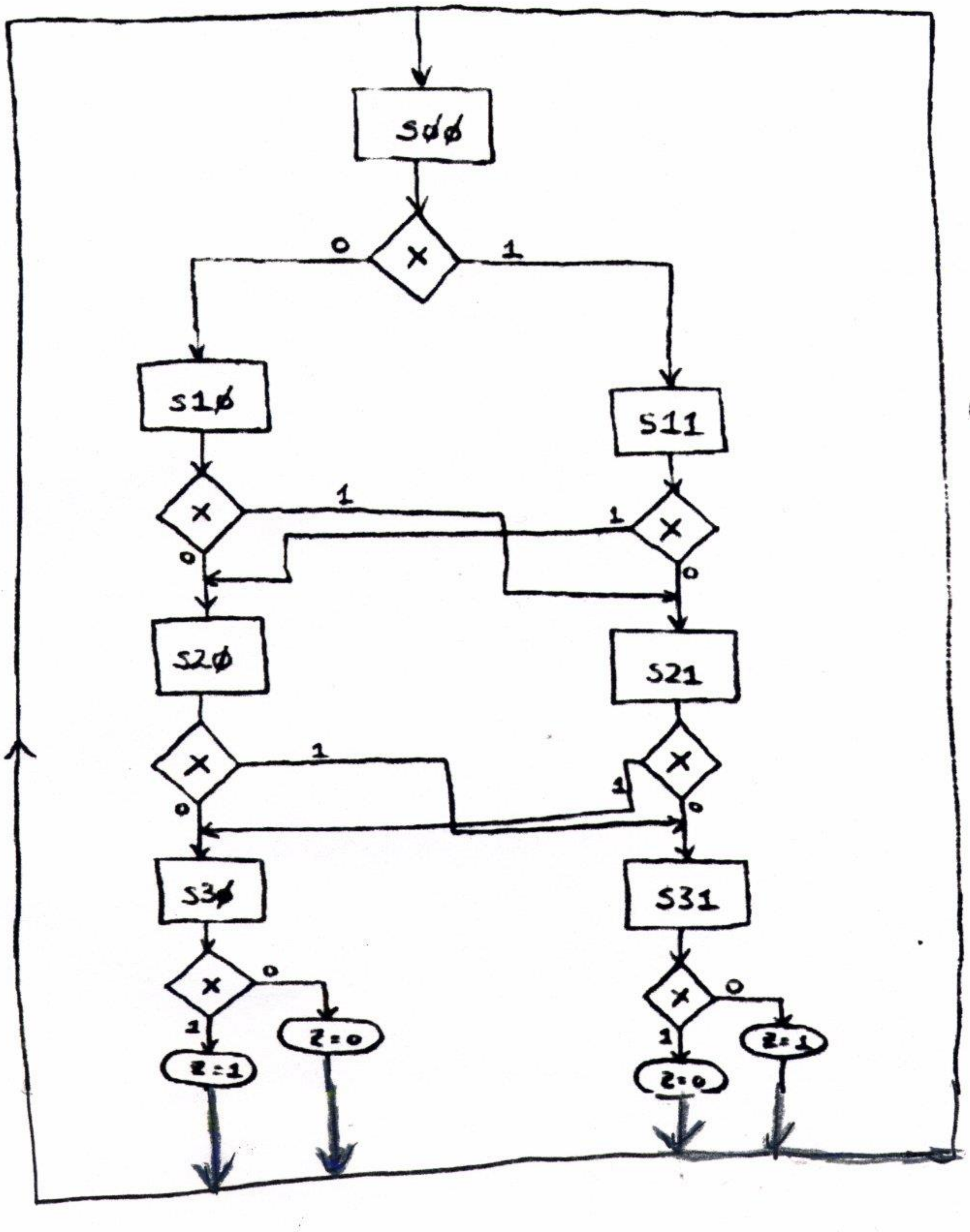
State $s[k][m]$

bits received

current parity → odd/even

Input X

Output Z (Error)


S∅∅
X
0
1
S1∅
S11
X
1
1
X
0
0
S2∅
S21
X
1
X
1
0
0
S3∅
S31
X
0
1
Z=0
Z=1
X
0
1
Z=1
Z=0

Q3.

The answer to this can be found in the textbook.

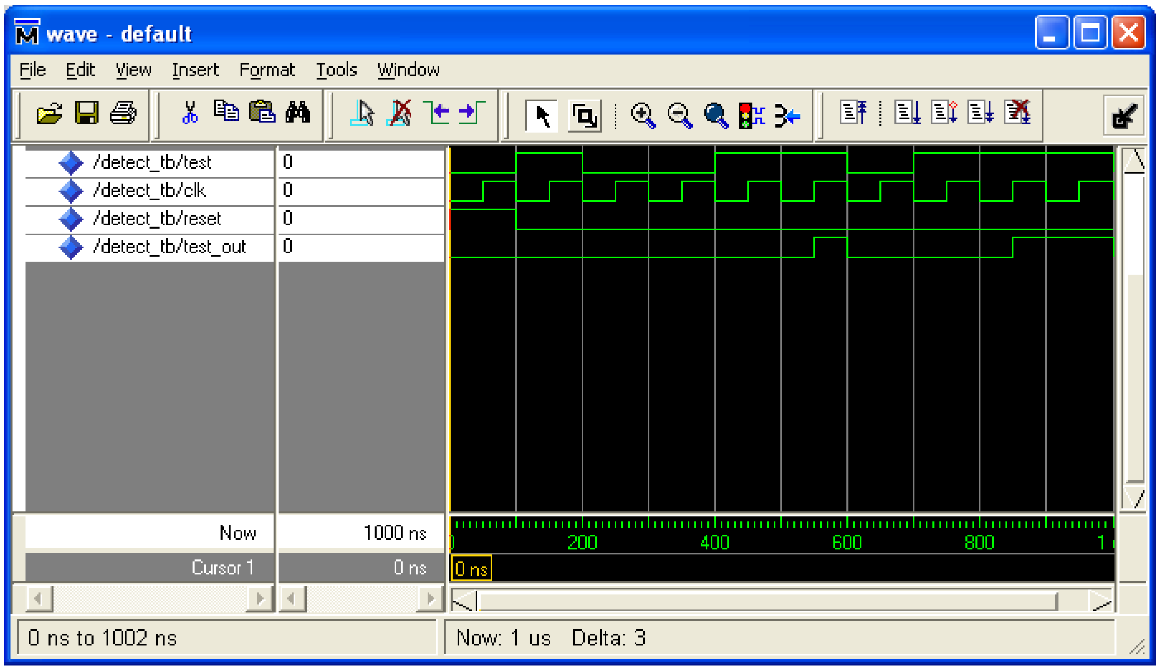
Q4.

Code Listing: (Testbench)

```
entity detect_tb is
end entity detect_tb;

architecture tb of detect_tb is

-- Component declaration
component  seq_detect is
    port(x, clock, reset : in BIT;
        z: out BIT);
end component seq_detect;

-- Signal declaration
signal test, clk, reset, test_out : bit;

begin
-- Device under test

DUT: seq_detect port map ( test, clk, reset, test_out);

tim: process is
    begin
        clk <= '0';
    wait for 50 NS;
       clk <= '1';
    wait for 50 NS;
end process tim;

rst: process is
begin
    reset <= '1';
    wait for 100 NS;
    reset <= '0';
    wait;
end process rst;

put_in: process is
    begin
       test <= '0';
       wait for 100 NS;
       test <= '1';
       wait for 100 NS;
       test <= '0';
       wait for 100 NS;
       test <= '0';
       wait for 100 NS;
       test <= '1';
```

```
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '0';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '0';
        wait for 100 NS;
        test <= '0';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '1';
        wait for 100 NS;
        test <= '0';
        wait;
end process put_in;

end architecture tb;
```

Simulation Results:



Q5(a)

Code $S\phi$ as 00
S1 as 01
S2 as 11

$\rightarrow$ this is just one possibility

$\Rightarrow$ Require two flip-flops Y_0 & Y_1

Present	State	Input	Next	State	Outpu
Y_1	Y_0	X	Y_1^+	Y_0^+	Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	1	0
1	1	0	0	0	0
1	1	1	1	1	1

$Z = X . Y_0 . Y_1$

Y_1^+ :

$X \backslash Y_1Y_0$	00	01	11	10
0	0	0	0	x
1	0	1	1	x

$Y_1^+ = X . Y_0$

Y_0^+ :

$X \backslash Y_1Y_0$	00	01	11	10
0	0	0	0	x
1	1	1	1	x

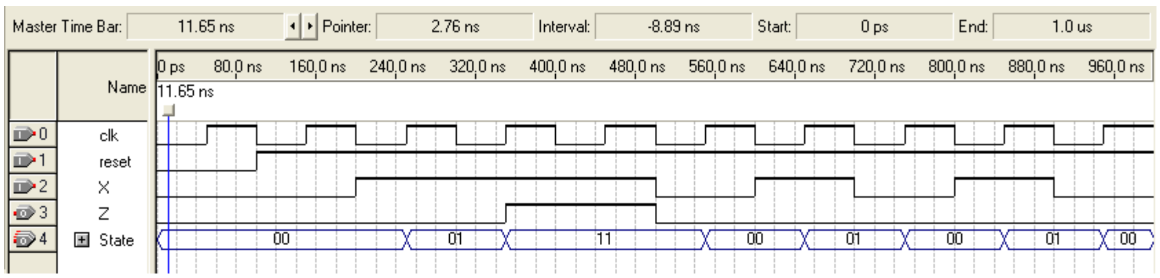
$Y_0^+ = X$

these equations define the next state logic used as i/p's to th flip-flops.

Q5 (b)

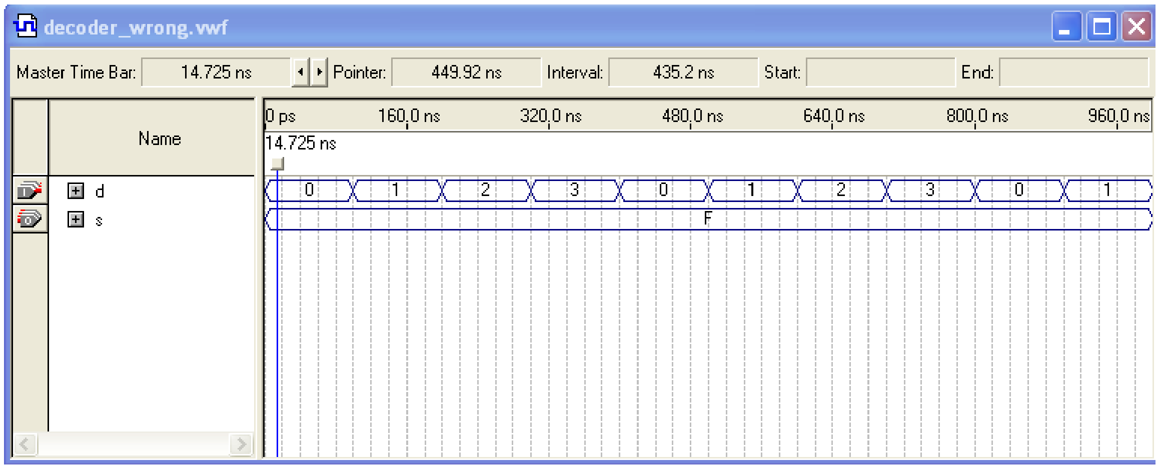
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY fsm_simple IS
        PORT
        (
                X, clk, reset           : IN      STD_LOGIC;
                Z                       : OUT  STD_LOGIC;
                y0, y1                  : BUFFER STD_LOGIC  -- to show the state
        );
END fsm_simple;

ARCHITECTURE dataflow OF fsm_simple IS
        SIGNAL y0_next, y1_next : STD_LOGIC;
BEGIN

trans:
PROCESS(clk,reset)
BEGIN
        IF reset = '0' THEN  -- assume active low reset
                y0 <= '0'; y1 <= '0';
        ELSIF rising_edge(clk) THEN
                y0 <= y0_next; y1 <= y1_next;
        END IF;
END PROCESS trans;

comb:
PROCESS(y0,y1,X)
BEGIN
        y0_next <= X;
        y1_next <= X and y0;
        Z <= X and y0 and y1;
END PROCESS comb;

END dataflow;
```

Simulation

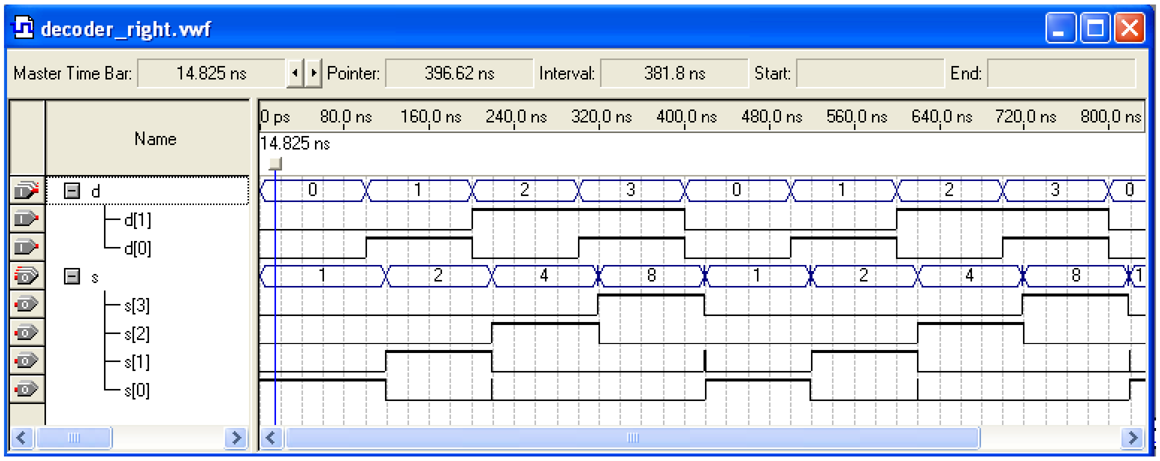


Q6.

Simulation Results (Wrong Version):



Simulation Results (Right Version):



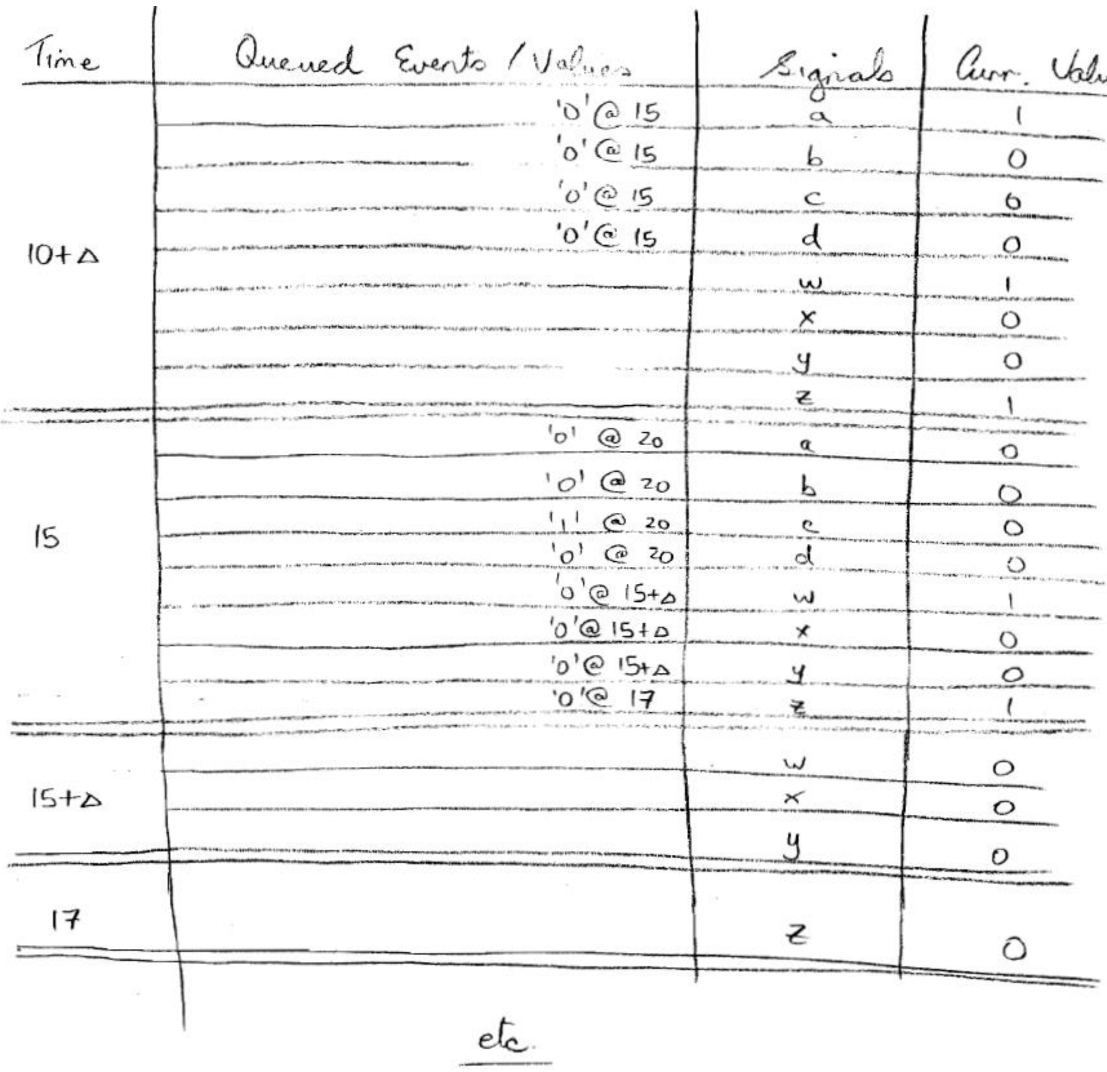
Code Listing:

```
library ieee;
USE ieee.std_logic_1164.all;

entity decoder_right is
port (d: in std_logic_vector(1 downto 0);
       s: out std_logic_vector(3 downto 0));
end decoder_right;

architecture alright of decoder_right is
begin

process(d)
begin
     S <= "0000";
     case d is
          when "00" =>
               S(0) <= '1';
          when "01" =>
               S(1) <= '1';
          when "10" =>
               S(2) <= '1';
          when "11" =>
               S(3) <= '1';
          when others =>
               null;
     end case;

end process;

end alright;
```

Q7.

Initially, all values will have the value 'U'
First transition occurs at time 0 and the process is executed, so well as the assignment statement

Time	Queued Events/Values	Signals	Current Value
	'1' @ 10	a	1
	'0' @ 10	b	0
0	'0' @ 10	c	0
	'0' @ 10	d	1
	'0' @ Δ	w	U
	'0' @ Δ	x	U
	'1' @ Δ	y	U
	'1' @ 2ns	z	U
	'1' @ 10	a	1
	'0' @ 10	b	0
	'0' @ 10	c	0
Δ	'0' @ 10	d	1
		w	0
		x	0
		y	1
	'1' @ 2	z	U
2	only change is in z	z	1
	'0' @ 15	a	1
	'0' @ 15	b	0
	'0' @ 15	c	0
10	'0' @ 15	d	0
	'1' @ 10+Δ	w	0
	'0' @ 10+Δ	x	0
	'0' @ 10+Δ	y	1
	'1' @ 12ns	z	1



Time	Queued Events / Values	Signals	Curr. Value
10+Δ	'0'@ 15	a	1
	'0'@ 15	b	0
	'0'@ 15	c	0
	'0'@ 15	d	0
		w	1
		x	0
		y	0
		z	1
15	'0' @ 20	a	0
	'0' @ 20	b	0
	'1' @ 20	c	0
	'0' @ 20	d	0
	'0'@ 15+Δ	w	1
	'0'@ 15+Δ	x	0
	'0'@ 15+Δ	y	0
	'0'@ 17	z	1
15+Δ		w	0
		x	0
		y	0
17		z	0

etc.