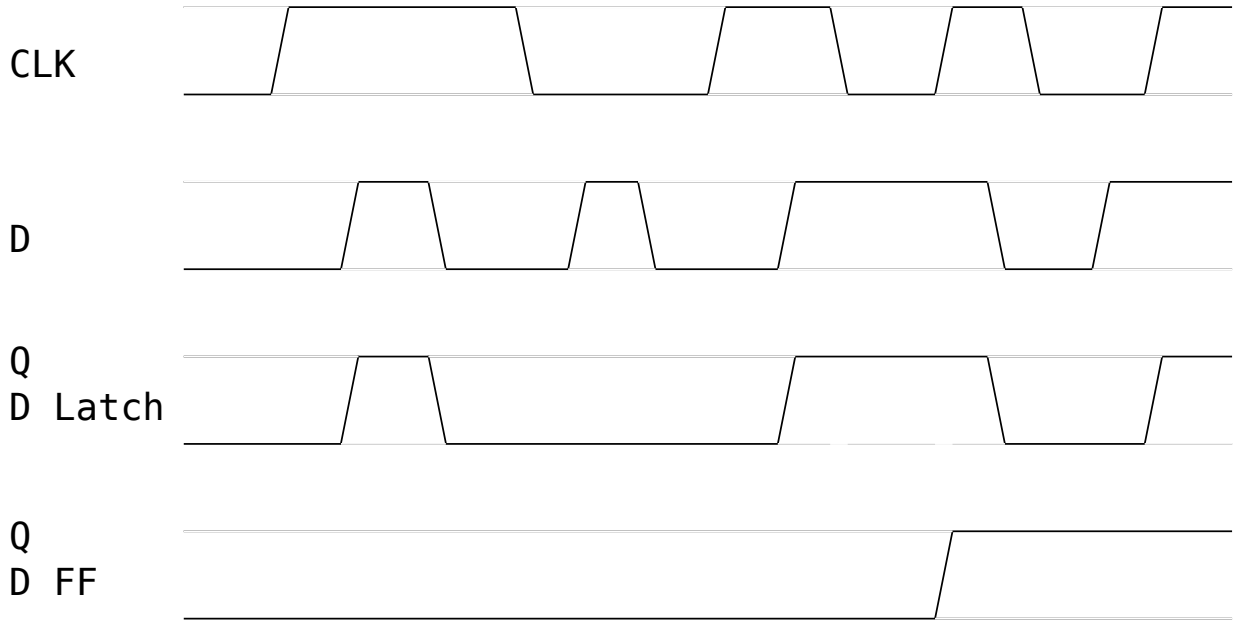
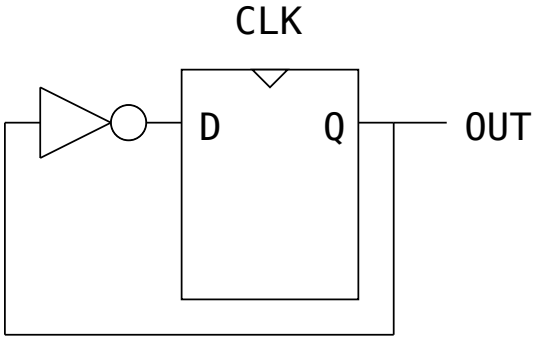
2 Homework 2

2.1 DDCA 3.2 and DDCA 3.3



2.2 DDCA 3.6



This toggle flip-flop will alternate between 1'b1 and 1'b0 every clock cycle provided that the D flip-flop has an initial value.

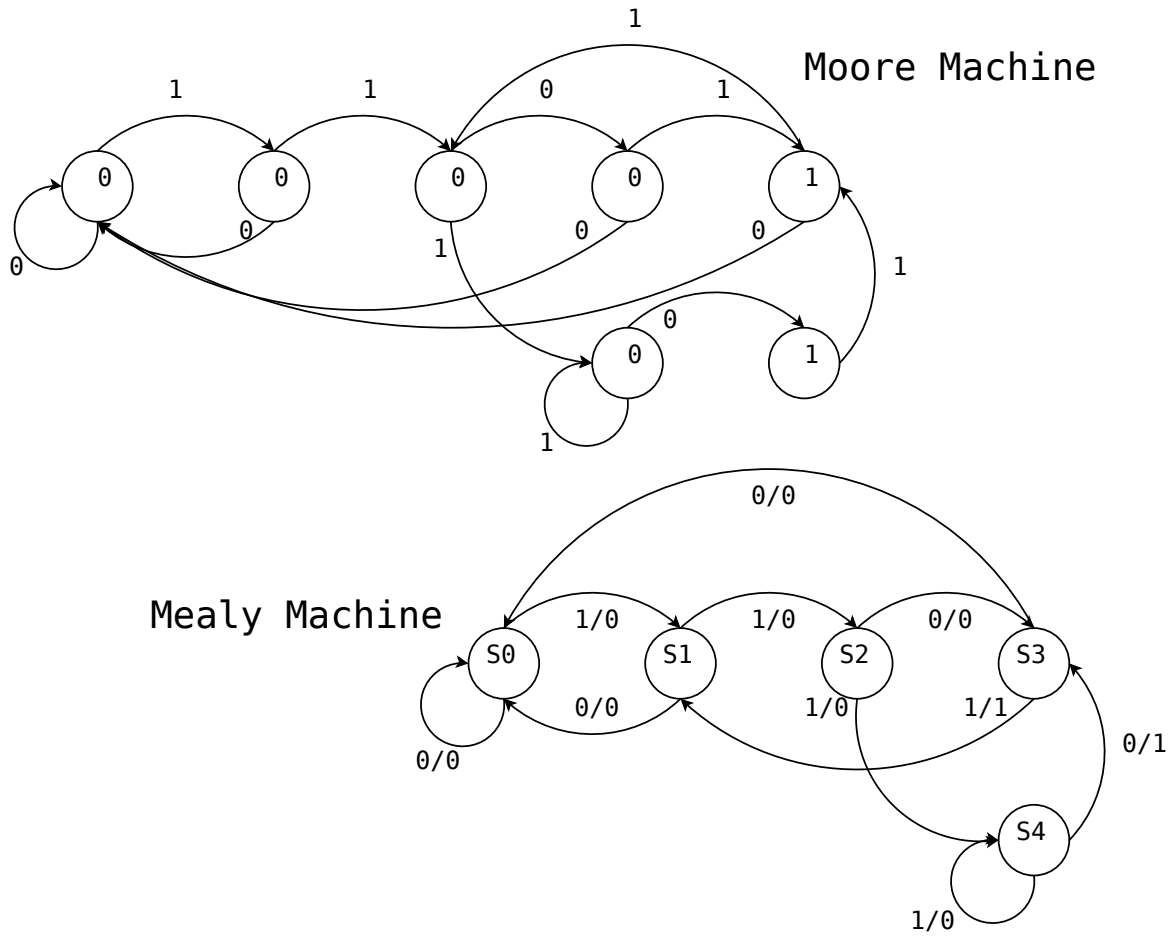
2.3 DDCA 3.16

There are 25 floors numbered as $\{1 \ldots 12, 14 \ldots 26\}$. There are a total of 25 possible states which can be represented by 5 bits : $2^5 > 25$.

2.4 DDCA 3.17

There are 4 students and each student has 5 moods (HAPPY, SAD, BUSY, CLUELESS, ASLEEP). There are a total of $4 \times 5 = 20$ possible states which can be represented by 5 bits : $2^5 > 20$.

2.5 DDCA 3.22



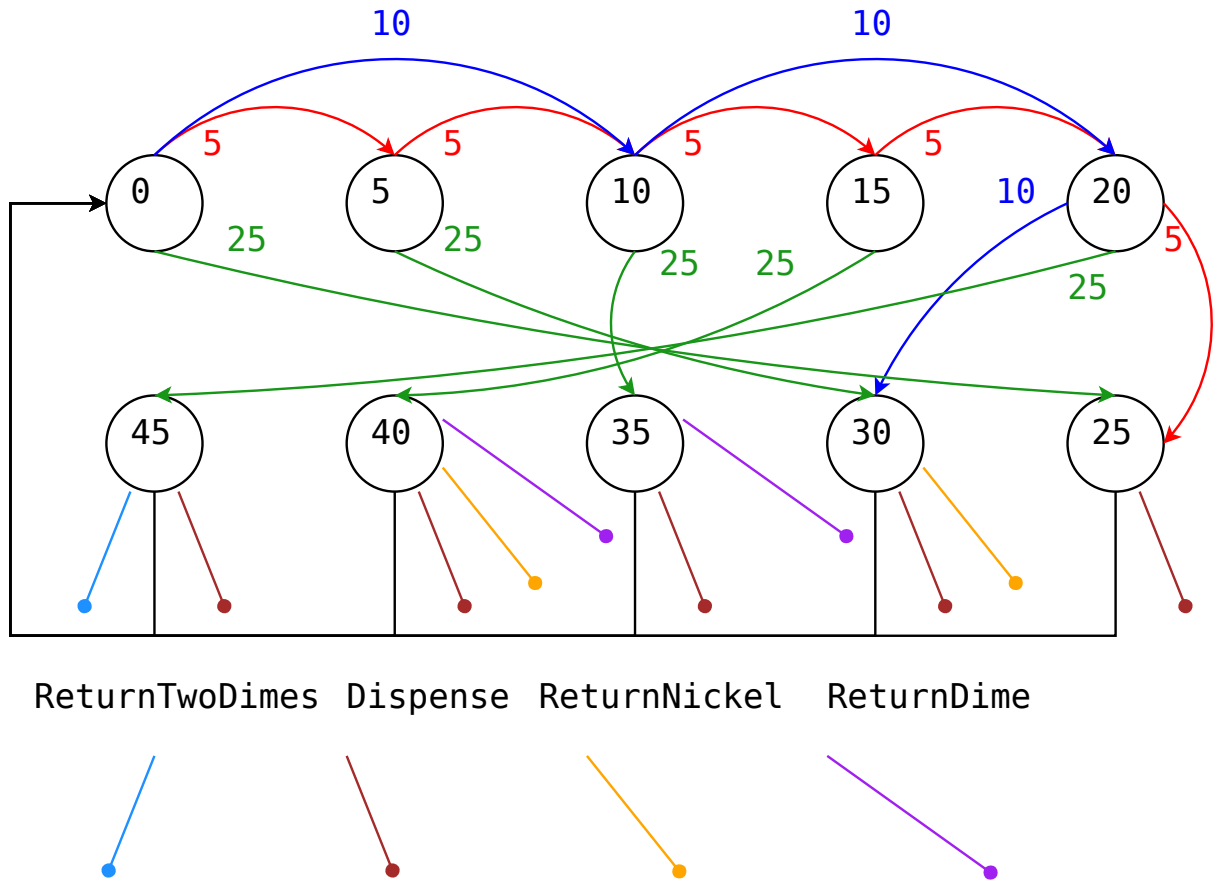
Here is the combined state transition and output table of the **Mealy Machine** representation of the 1101 and 1110 detector. The FSM schematic can be easily realized by implementing the next state boolean equations. The Moore Machine state transition diagram is provided as a reference.

State	Current State			Input	Output	Next State			Next State
State	S_2	S_1	S_0	Input	Output	S'_2	S'_1	S'_0	Encoding
S0	0	0	0	0	0	0	0	0	S0
S0	0	0	0	1	0	0	0	1	S1
S1	0	0	1	0	0	0	0	0	S0
S1	0	0	1	1	0	0	1	0	S2
S2	0	1	0	0	0	0	1	1	S3
S2	0	1	0	1	0	1	0	0	S4
S3	0	1	1	0	0	0	0	0	S0
S3	0	1	1	1	1	0	0	1	S1
S4	1	0	0	0	1	0	1	1	S3
S4	1	0	0	1	0	1	0	0	S4

$$
\begin{aligned}
Output &= \bar{S}_2 S_1 S_0 Input + S_2 \bar{S}_1 \bar{S}_0 \bar{Input} \\
S'_2 &= \bar{S}_2 S_1 \bar{S}_0 Input + S_2 \bar{S}_1 \bar{S}_0 Input \\
&= \bar{S}_0 Input(\bar{S}_2 S_1 + S_2 \bar{S}_1) \\
S'_1 &= \bar{S}_2 \bar{S}_1 S_0 Input + \bar{S}_2 S_1 \bar{S}_0 \bar{Input} + S_2 \bar{S}_1 \bar{S}_0 \bar{Input} \\
&= \bar{S}_2 \bar{S}_1 S_0 Input + \bar{S}_0 \bar{Input}(\bar{S}_2 S_1 + S_2 \bar{S}_1) \\
S'_0 &= \bar{S}_2 \bar{S}_1 \bar{S}_0 Input + \bar{S}_2 S_1 \bar{S}_0 \bar{Input} + \ \bar{S}_2 S_1 S_0 Input + S_2 \bar{S}_1 \bar{S}_0 \bar{Input} \\
&= \bar{S}_2 \bar{S}_1 \bar{S}_0 Input + \bar{S}_2 S_1 (\bar{S}_0 \bar{Input} + S_0 Input) + \ \bar{S}_0 \bar{Input}(\bar{S}_2 S_1 + S_2 \bar{S}_1)
\end{aligned}
$$

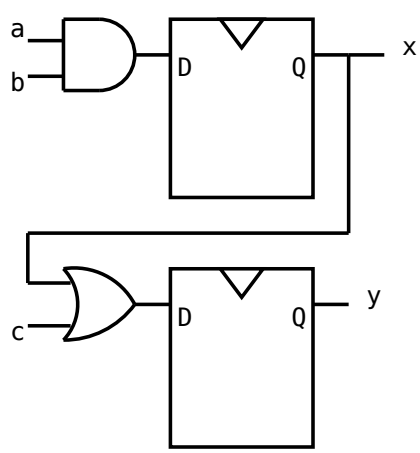
2.6 DDCA 3.23

With the possible inputs being *Nickel(5)*, *Dime(10)*, and *Quarter(25)*, there are 10 possible states that the soda machine could be in: {0, 5, 10, 15, 20, 25, 30(5 + 25, 20 + 10), 35(10 + 25), 40(15 + 25), 45(20 + 25)}. With this information, we know that 4 bits are needed to encode all 10 states : $2^4 < 10$. The possible output are *Dispense*, *ReturnNickel*, *ReturnDime*, and *ReturnTwoDimes*. Here is the state transition diagram.



2.7 DDCA 4.46

The two Verilog modules behave exactly alike and produce the same hardware because they both utilize non-blocking statements which are performed in parallel.



Listing 1: DDCA 4.46

```
module code1 (  input        clk, a, b, c,
                output reg   y);
    reg x;

    always @ (posedge clk) begin
        x <= a & b;
        y <= x | c;
    end
endmodule

module code2 (  input        a, b, c, clk,
                output reg   y);
    reg x;

    always @ (posedge clk) begin
        y <= x | c;
        x <= a & b;
    end
endmodule
```

2.8 DDCA 4.48

Listing 2: DDCA 4.48.a

```
module latch (  input                clk,
                input        [3:0]   d,
                output reg   [3:0]   q);

//  always @ (clk) -> ERROR
    // This is an error because the latch will
    // not be transparent. For example, if d changes
    // after clk has already been high, then q will not
    // also change to the new value of d.
    always @ (clk or d)
    //always @ (*) // Better to use (*) as the sensitivity list
        if (clk) q <= d;

endmodule
```

Listing 3: DDCA 4.48.b

```
module gates (  input        [3:0] a, b,
                output reg   [3:0] y1, y2, y3, y4, y5);

//  always @ (a) -> ERROR
    // The sensitivity list must include ALL inputs.
    // The safest way to go would be to use (*) as the
    // sensitivity list and let the tools adjust accordingly.
    // also correct : always @ (a or b)
    always @ (*)
        begin
            y1 = a & b;
            y2 = a | b;
            y3 = a ^ b;
```

```
            y4 =  ~( a & b );
            y5 = ~(a | b);
        end
endmodule
```

Listing 4: DDCA 4.48.c

```
module mux2 (    input        [3:0]    d0, d1,
                 input                 s,
                 output reg   [3:0]    y);

//  always @ (posedge s) -> ERROR
    // This is not quite a true mux because the output
    // will only change on the pisitive edge of s.
    // The mux output, y, must change whenever s, d0, or d1 changes.
    always @ (*)
        if (s) y <= d1;
        else y <= d0;
    // You could also use the C-style ?: notation.
    // assign y = (s) ? d1 : d0;
    // The output y must be declared as a wire in this case.
endmodule
```

Listing 5: DDCA 4.48.d

```
module twoflops (    input        clk,
                     input        d0, d1,
                     output reg   q0, q1);

    always @ (posedge clk)
    // Error 1 : no begin/end
    // Error 2 : non-blocking("<=") assignments should
    // be used for sequential logic
        begin
//          q1 = d1;
//          q0 = d0;
            q1 <= d1;
            q0 <= d0;
        end
endmodule
```

Listing 6: DDCA 4.48.e

```
module FSM (input        clk,
            input        a,
            output reg   out1, out2);
    reg state;

    // next state logic and register (sequential)
    always @ (posedge clk)
    if (state == 0) begin
        if (a) state <= 1;
    end else begin
        if (~a) state <= 0;
    end
```

```
    // In an "always @ (*)" block, if anything
    // is ever assigned, it must be assigned
    // in all possible cases. If this is not done,
    // a latch is generated.
    always @ (*) // output logic (combinational)
//      if (state == 0) out1 = 1; -> ERROR : out2 would be latched
//      else            out2 =1; -> ERROR : out1 would be latched
        if (state == 0)
            begin
                out1 = 1;
                out2 = 0;
            end
        else
            begin
                out1 = 0;
                out2 = 1;
            end
endmodule
```

Listing 7: DDCA 4.48.f

```
module priority (   input       [3:0] a,
                    output reg  [3:0] y);

    // In an "always @ (*)" block, if anything
    // is ever assigned, it must be assigned
    // in all possible cases. If this is not done,
    // a latch is generated.
    always @ (*)
        if      (a[3])  y = 4'b1000;
        else if (a[2])  y = 4'b0100;
        else if (a[1])  y = 4'b0010;
        else if (a[0])  y = 4'b0001;
        else            y = 4'bXXXX; // ADDED
endmodule
```

Listing 8: DDCA 4.48.g

```
module divideby3FSM (   input clk,
                        input reset,
                        output out);

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;

    // State Register
    always @ (posedge clk, posedge reset)
    if (reset)  state <= S0;
    else        state = nextstate;

    // Next State Logic
```

```
    always @ (state)
        case (state)
            S0: nextstate = S1;
            S1: nextstate = S2;
//          2:  nextstate = S0; // ERROR
            // The parameter S2 should be used just in case
            // the actual representation of the states changes
            // in the future.
            S2 : nextstate = S0;
            default : nextstate = 2'bXX; // ADDED
            // There should be a default statement in most if not
            // all case statements to prevent a latch from being generated.
        endcase

        // Output Logic
    assign out = (state == S2);
endmodule
```

Listing 9: DDCA 4.48.h

```
module mux2tri (input [3:0]     d0, d1,
                input           s,
                output [3:0]    y);

//  tristate t0 (d0, s, y); -> ERROR
    tristate t0 (d0, ~s, y);
    // It must be guaranteed that y is not driven by multiple inputs.
    // This is done by making sure that only ONE tristate drives y
    // at any point in time.
    tristate t1 (d1, s, y);
endmodule
```

Listing 10: DDCA 4.48.i

```
module floprsen (input                clk,
                 input                reset,
                 input                set,
                 input        [3:0]   d,
                 output reg   [3:0]   q);

    // q cannot be assigned in 2 separate always blocks.
    // If reset and set are high, which takes precedence?
    // In this module, both reset and set are asynchronous as
    // in they change q regardless of the rising edge of the clk.

    // If this were to have a synchronous reset and set, the
    // always block would only be sensitive to posedge clk.

//  always @ (posedge clk, posedge reset)
    always @ (posedge clk, reset, set)
    // posedge reset is unecessary because q is only changed
    // when reset if 1 which is checked in the always block
        if (reset)  q <= 0;
        else if (set) q <= 1;
        else        q <= d;
```

```
//  always @ (set)
//      if (set) q <= 1;
endmodule
```

Listing 11: DDCA 4.48.j

```
module and3 (   input a, b, c,
                output reg y);

    reg tmp;
//  always @ (a or b or c)
    // It is a good habit to use "always @ (*)" instead
    // of explicitly stating the sensitivity list.
    always @ (*)
        begin
//          tmp <= a & b; -> ERROR
//          y <= tmp & c; -> ERROR
            // Non-blocking statements should not be used
            // for combinational logic.
            tmp = a & b;
            y = tmp & c;

        end
endmodule
```