# LECTURE 8

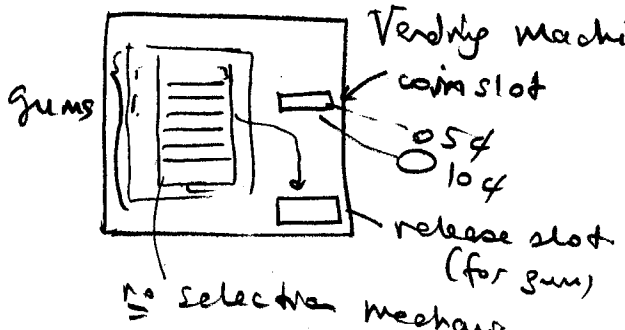
- **Emphasis**: Mapping word problems to Mealy/Moore FSMs.

## Ex1] Simple Vending Machine:

- Vending machine serves gum.
- Each gum costs 15 ¢.
- The machine accepts only nickels & dimes.
- ~~Exact change must be provided~~; The machine does not return extra money. (e.g. 10 + 10 = 20; gives 1 gum & 5 ¢ feels wasted)
- Machine should work continuously, i.e. after putting 15¢, if I put 5¢ it should start counting towards next gum.

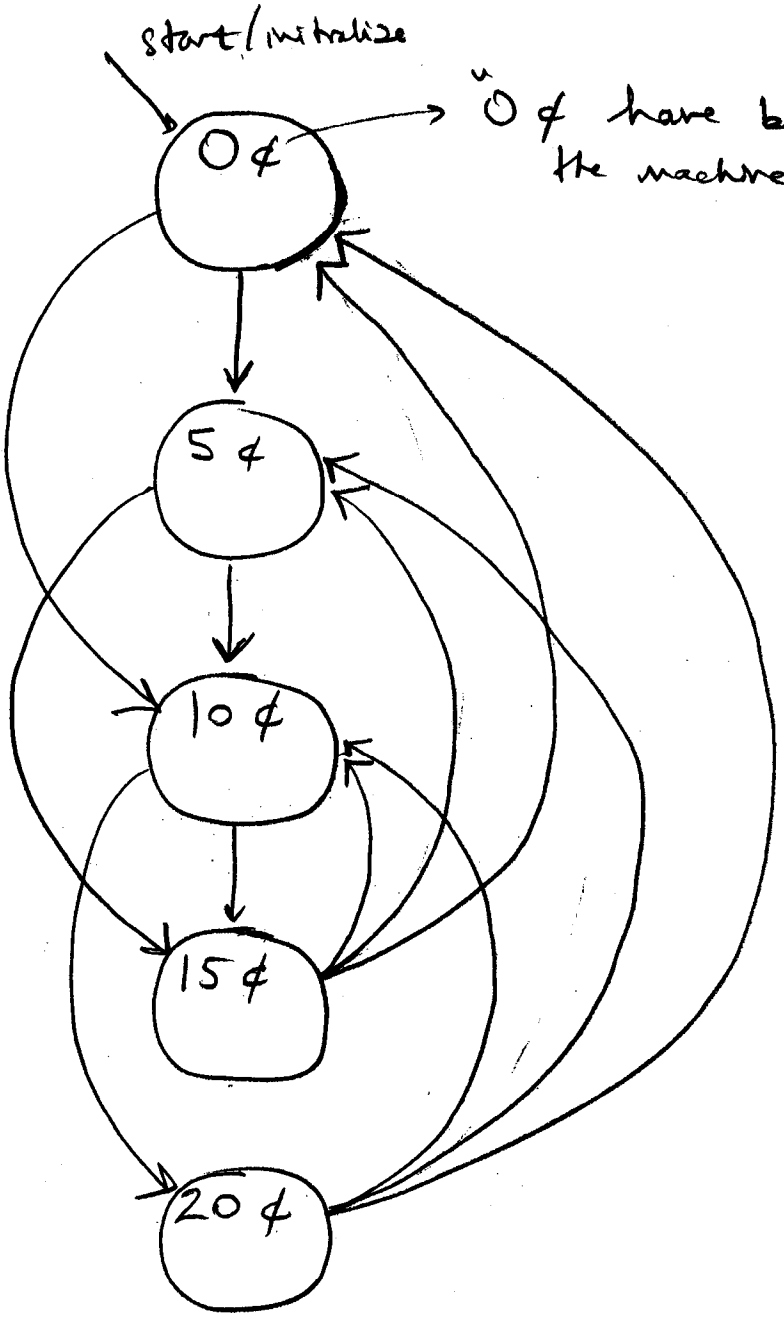
— Implement the control of this machine in Verilog.

1 — Understand the problem:




2 — Draw the state diagram: [Ask class]

See next page




- Draw the basic structure (above) to capture the essence of the solution.

- Now, define the inputs and outputs clearly.

Inputs: sensor must indicate whether 1) it's a dime 2) or nickel, ~~but not otherwise~~ 3) nothing has been deposited.

~~need~~ need 3 values: so 2 bits.

Use $N = \begin{cases} 1 & \text{nickel} \\ 0 & \text{otherwise} \end{cases}$

$$D = \begin{cases} 1 & \text{dime} \\ 0 & \text{otherwise} \end{cases}$$

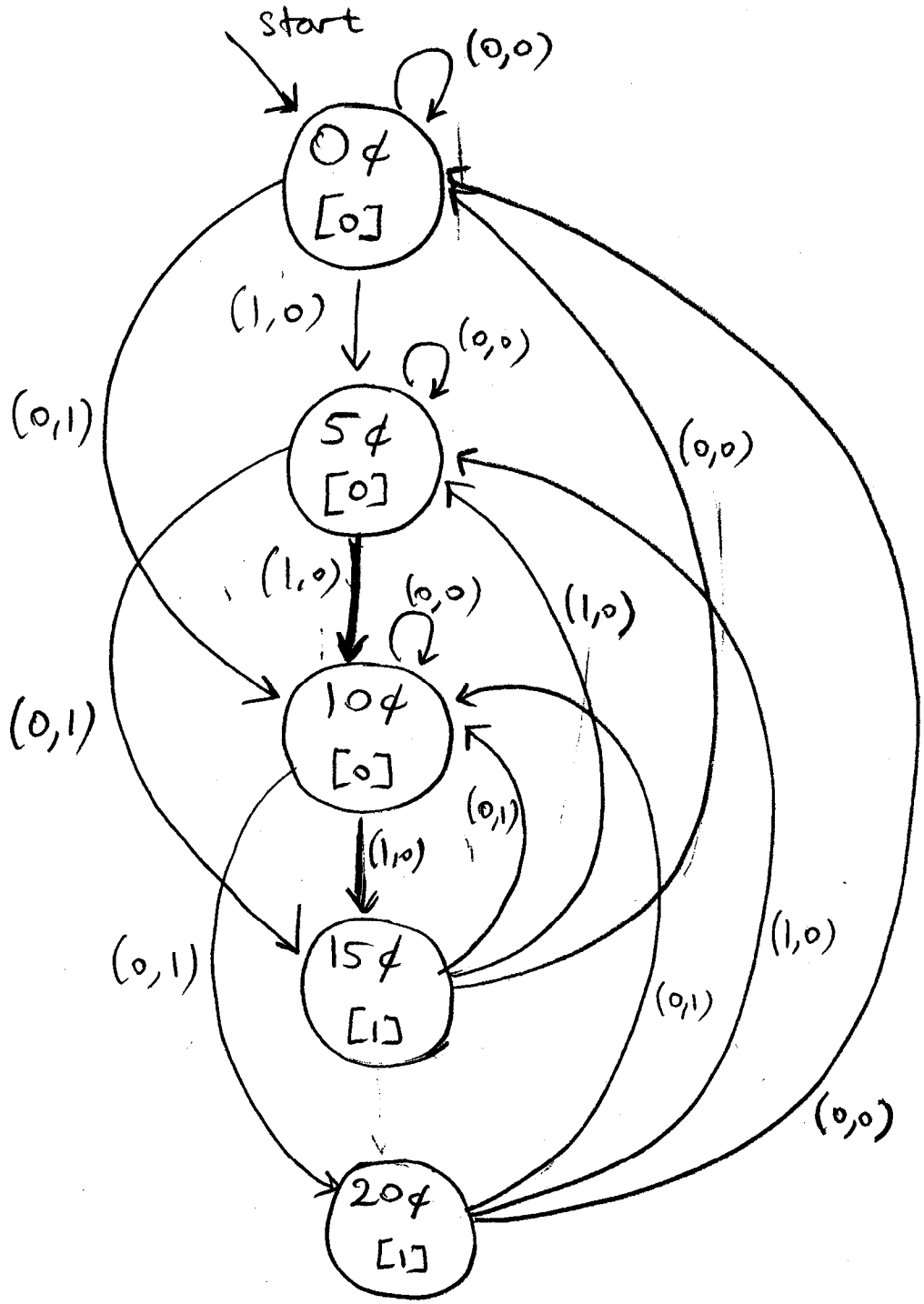
Here:

| N | D | meaning |
|---|---|---|
| 0 | 0 | nothing deposited. |
| 0 | 1 | dime |
| 1 | 0 | nickel |
| 1 | 1 | × ← never occurs: don't make care. |

<u>outputs</u>: gum releaser.

$$G = \begin{cases} 1 & \text{release gum} \\ 0 & \text{otherwise} \end{cases}$$

Now, let us add these to the state diagram.




Inputs: (N, D)

- Make sure that all input combinations are handled by the state diagram. (There should be 3 arrows exiting from each state above.)
- Write the G output on each state. (Moore machine)

4_ State minimization :

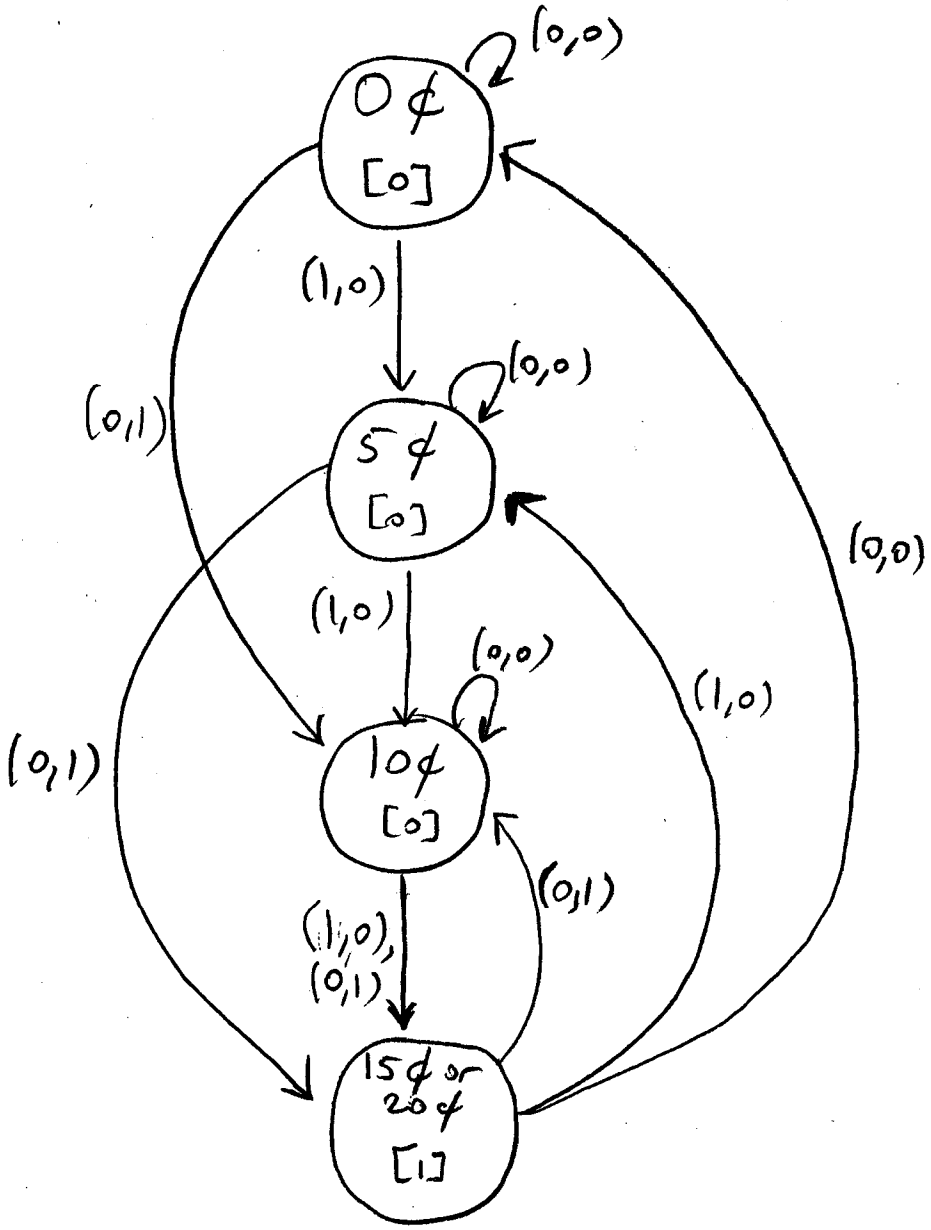
Eliminate any 'redundant' states. (This is usually difficult and we need more advanced tools)

But in this example we see that the state 20¢ is redundant : Examine (1) the output arrows of 15¢ and 20¢, and (2) the output. If sitting in any one of these states, impossible to tell which one. (A sufficient condition for equivalence)

Instead of 15¢, we can have a state meaning "15¢ or 20¢ have been deposited".

The reduced state table is : → next page

(Reduced state diagram) :




(N,D)

5_ State table:

we have 4 states: need 2 bits to represent them.

State assignment: $Q_1$ $Q_0$

0¢ : 0 0

5¢ : 0 1

10¢ : 1 0

15¢ : 1 1

State table: (state-assigned table):

| current inputs N | D | current state $Q_1$ | $Q_0$ | | next state $Q_1^+$ | $Q_0^+$ | G | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | (0¢) | 0 | 0 | (0¢) | 0 |
| 0 | 0 | 0 | 1 | (5¢) | 0 | 1 | (5¢) | 0 |
| 0 | 0 | 1 | 0 | (10¢) | 1 | 0 | (10¢) | 0 |
| 0 | 0 | 1 | 1 | (15¢) | 0 | 0 | (0¢) | 1 ← (uses current state!) |
| 0 | 1 | 0 | 0 | (0¢) | 1 | 0 | (10¢) | 0 |
| 0 | 1 | 0 | 1 | (5¢) | 1 | 1 | (15¢) | 0 |
| 0 | 1 | 1 | 0 | (10¢) | 1 | 1 | (15¢) | 0 |
| 0 | 1 | 1 | 1 | (15¢) | 1 | 0 | (10¢) | 1 |
| 1 | 0 | 0 | 0 | (0¢) | 0 | 1 | (5¢) | 0 |
| 1 | 0 | 0 | 1 | (5¢) | 1 | 0 | (10¢) | 0 |
| 1 | 0 | 1 | 0 | (10¢) | 1 | 1 | (15¢) | 0 |
| 1 | 0 | 1 | 1 | (15¢) | 0 | 1 | (5¢) | 1 |
| 1 | 1 | 0 | 0 | | × | × | | × |
| 1 | 1 | 0 | 1 | | × | × | | × |
| 1 | 1 | 1 | 0 | | × | × | | × |
| 1 | 1 | 1 | 1 | | × | × | | × |

Point out to class: (uses current state!)

# 6- ~~Methods~~

→ what we will do now

(A) By hand:
- write output and next-state equations.
- map to a flip-flop implementation.

(B) Write Verilog:
- From state diagram, write Verilog.
- Use CAD tools to minimize and to download to a target architecture (CPLD or an FPGA).

---

~~(A)~~: output equation:

We will represent it as a K-map (and include the don't cares).

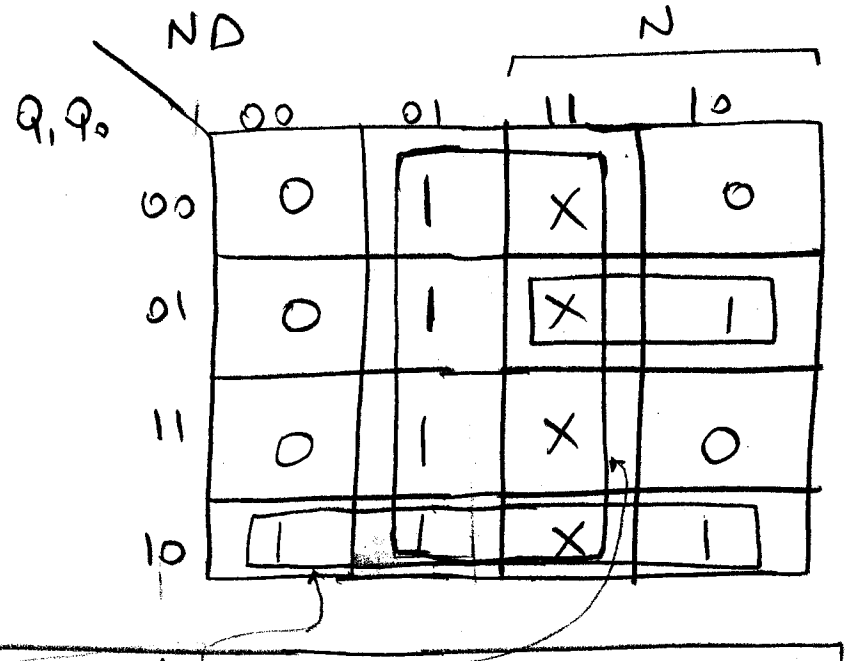
G:

| $Q_1Q_0$ \ ND | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 [0] | 0 [4] | X [12] | 0 [8] |
| 01 | 0 [1] | 0 [5] | X [13] | 0 [9] |
| 11 | 1 [3] | 1 [7] | X [15] | 1 [11] |
| 10 | 0 [2] | 0 [6] | X [14] | 0 [10] |

$G = Q_1 \cdot Q_0$ (min SOP form)

## next-state eqns:

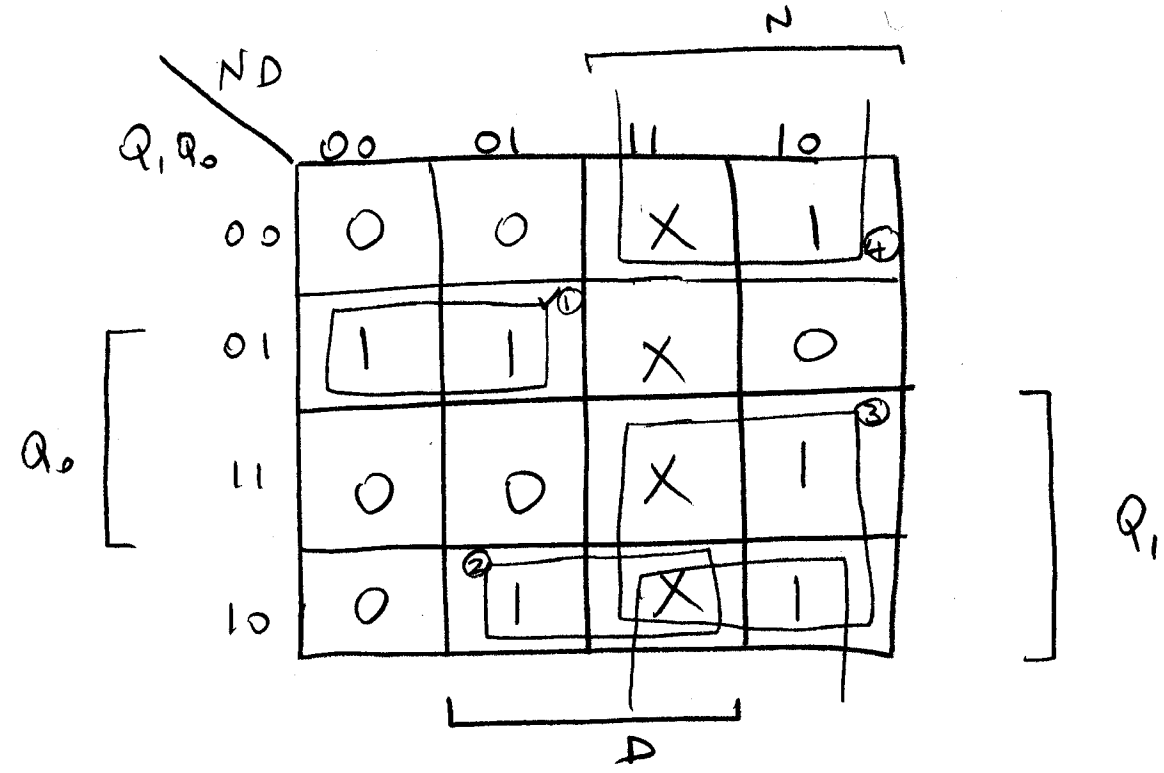
$Q_1^+$




$$Q_1^+ = Q_1 \cdot Q_0' + D + Q_1' \cdot Q_0 \cdot N$$

(min SOP form.)

$Q_0^+$

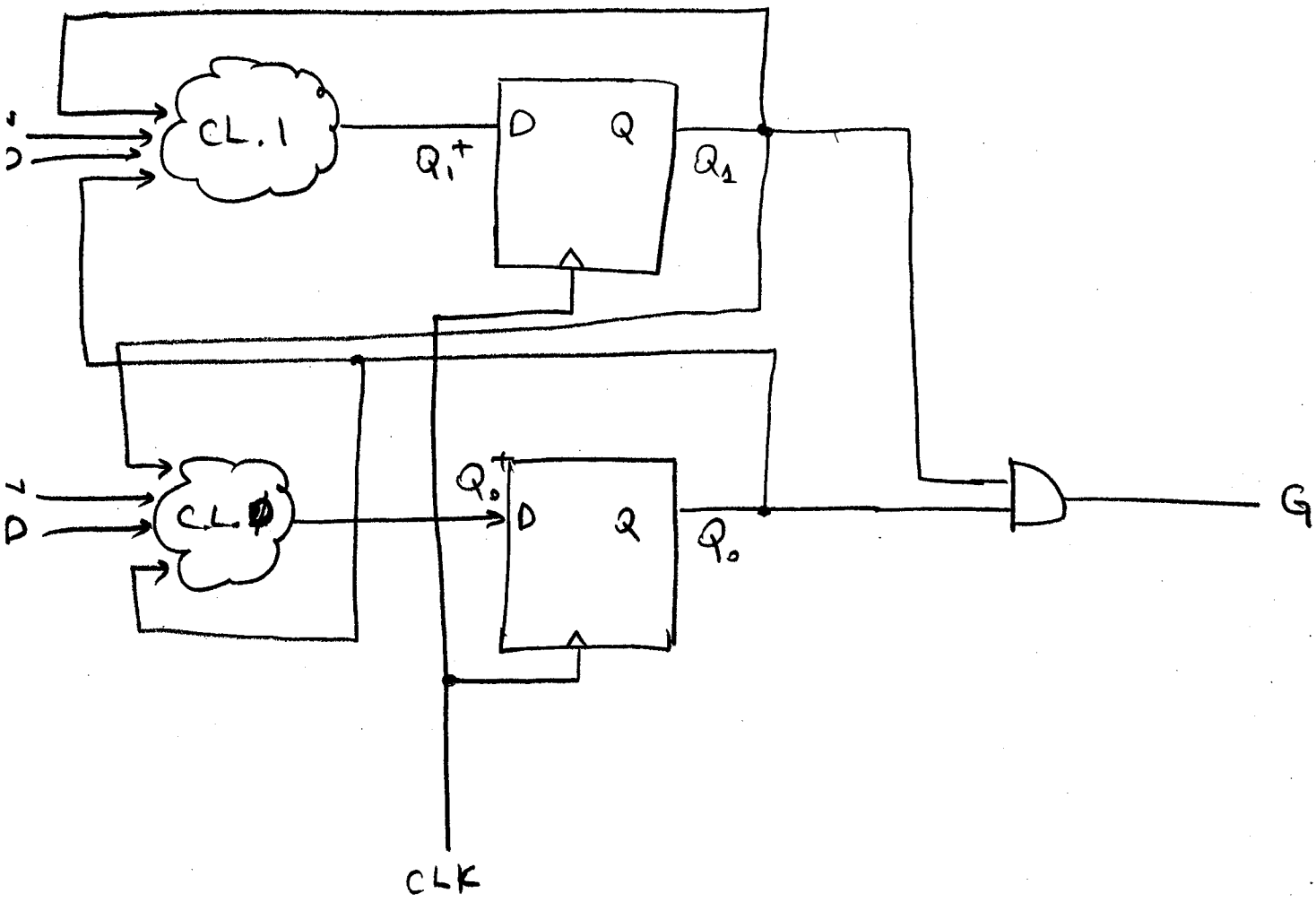



$$Q_0^+ = Q_1' \cdot Q_0 \cdot N' + Q_1 \cdot Q_0' \cdot D + Q_1 \cdot N + Q_0' \cdot N.$$

① ② ③

(min SOP).

– Map to D-flip-flops:




This implements the vending machine controller

▲

# LECTURE 9

AIM: Implement FSMs in Verilog.
(use then Verilog code for simulation & synthesis).

Two ways to write the FSM (e.g. SVM = simple vending machine).

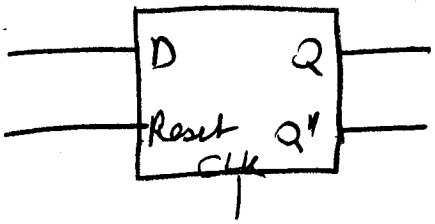
A RTL (register-transfer-level): at level of ffps & comb. logic. [from next-state & output equations.] mixes behavioral & dataflow styles.

B Behavioral level: at level of the state diagram. [from state diagram.]

---

- First we do the RTL design.
- In order to get there, first understand how to write a

Clocked D-latch:




```
module D_latch (D, Reset, CLK, Q, Q_bar);
    input D, Reset, CLK;
    output Q, Q_bar;

    reg Q, Q_bar;   //since on LHS below in
                    //procedural statement.
```

```
always @ (D or Reset or CLK)
    if (Reset)
        Q     <= 0;
        Q_bar <= 1;
    else
        begin
            if (CLK)
                Q     <= D;
                Q_bar <= ~D;
        end
endmodule
```

Remarks:

① Notice the asynchronous Reset. (This is in the nature of a latch, even clocked.) → Q will be set 0 any time Reset = 1.

② Notice the behavioral nature of the above code. It says nothing about the gate-level implementation. You could write a gate-level implementation, but the beauty of Verilog is that you can write code at different layers of abstraction. So: there we don't want to worry about the gate-level description of a latch, so we write it behaviorally.

③ What are <= assignments ?

non-blocking procedural assignments.

- The non-blocking assignments in a block transfer data from RHS → LHS in parallel.
- very useful for describing hardware.

④ Implied memory:

Note that if (CLK) ... has no else:

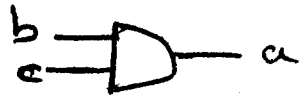
since the condition (~CLK) is not explicitly handled in code, the synthesis tools will assume that memory is implied and will synthesize a latch to produce the behavior.

When (~CLK), then hold in Q and Q_bar the values that those variables were last assigned.

# Assignments in Verilog

## Continuous

```
assign a = b & c;
```




## procedural

```
always @ (posedge CLK)
```

### blocking

```
a = b;
c = a;
```

Effect: c = b;

- Statements in same block executed sequentially (in order)

Ex: a 3, b 5, c 6

| | | | |
|---|---|---|---|
| a = b | a 5 | b 5 | c 6 |
| c = a | a 5 | b 5 | c 5 |

not good

### non-blocking

```
a <= b;
c <= a;
```

Effect: a ← b, c ← a

- Statements executed in parallel

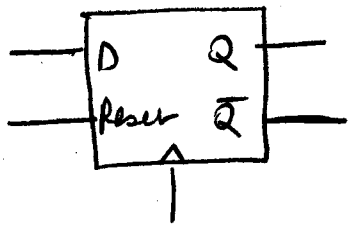
Ex: a 3, b 5, c 6

| | | | |
|---|---|---|---|
| a <= b; c <= a; | a 5 | b 5 | c 3 |

"c gets the old value of a".

good model for sequential synch. circuits

## D flip-flop:




```
module D_flipflop (D, Reset, CLK, Q, Q_bar);
    input D, Reset, CLK;
    output Q, Q_bar;

    reg Q, Q_bar;

    always @ (posedge CLK)
        if (Reset)
            Q     <= 0;
            Q_bar <= 1;
        else
            Q     <= D;
            Q_bar <= ~D;

endmodule
```

cannot mix types in sensitivity list (posedge & negedge in un-related segments)

**Remarks:**

1) **Implied memory:**

performs the body ✓ at the clock edge; at other times, it holds the value of Q (and Q_bar).

2) **Synchronous reset:** ~~At the edge, value Reset~~

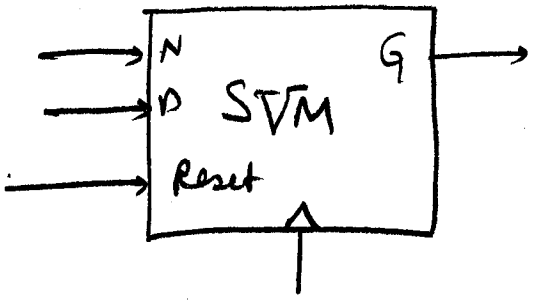
→ Reset takes effect at posedge

– Let's go back to the SVM

**A** Dataflow Level & state table

[from the next-state and output equations.]

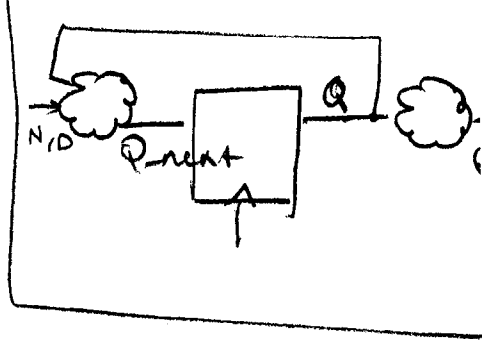
interface:




// Look at the hardware diagram at end of Lecture 8.

```
Module SimpleVendingMachine(N, D, CLK, Reset, G);
    input N, D, CLK, Reset;
    output G;

    reg[1:0] Q;        //current state
    wire[1:0] Q_next;  //next state
    // flip-flops:
    always@(posedge CLK)
        if (Reset)
            Q <= 2'b00;     // vector operation
        else
            Q <= Q_next;    //  "     "
    // combinational logic: next-state eqns
    assign Q_next[1] = Q[1] & ~Q[0]
                     | D & ~Q[1] & Q[0] & N;
```

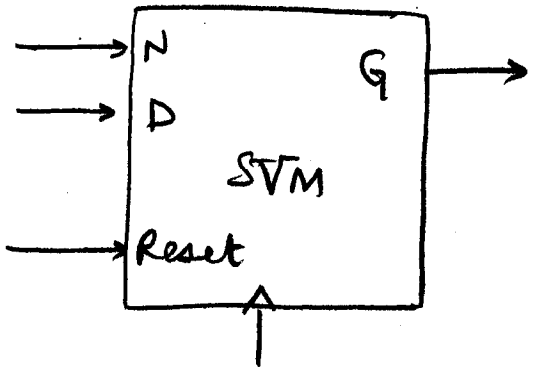
Picture in mind:



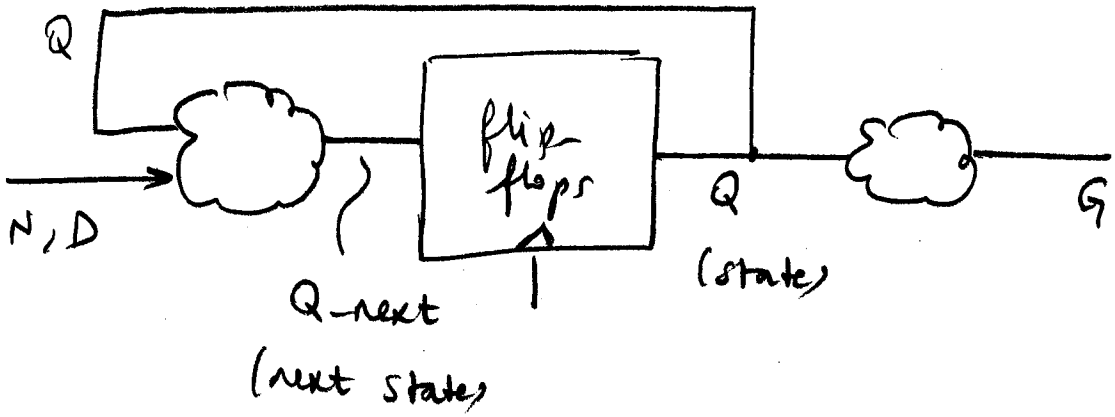
```
assign Q_next[0] = ~Q[1] & Q[0] & ~N
        | Q[1] & ~Q[0] & D | Q[1] & N | ~Q[0] & N;

// Combinational logic output eqn

assign G = Q[1] & Q[0];

endmodule.
```

## B Behavioral level : (from state diagram)

- Block diagram / Interface.




- Recall that we want to implement a Moore machine. So, hardware (after synthesis) should look like.




- Keep this model in mind when writing the behavioral level code.

```
module SimpleVendingMachine (N, D, CLK, Reset, G);
    input N, D, CLK, Reset;
    output G;

    //internal variables
    reg [1:0] Q, Q_next;

    //parameters: (short-hand; avoid errors; scope limited
    // to this module)
    parameter s0 = 2'b00;     //0¢
    parameter s5 = 2'b01;     //5¢
    parameter s10 = 2'b10;    //10¢
    parameter s15 = 2'b11;    //15¢
    //(could also write parameter s0 = 2'b00, s5 = ...;)
    //Flip-flops:
    always @ (posedge CLK)
        if (Reset)
            Q <= s0;
        else
            Q <= Q_next;
```

```
//Next-state comb. logic
always@(Q or N or D)
    case({Q, N, D})
    //s0;
    4'b0000: Q_next <= s0;
    4'b0001: Q_next <= s10;
    4'b0010: Q_next <= s5;
    //s5;
    4'b0100: Q_next <= s5;
    4'b0101: Q_next <= s15;
    4'b0110: Q_next <= s10;
    //s10:
    4'b1000: Q_next <= s10;
    4'b1001: Q_next <= s15;
    4'b1010: Q_next <= s15;
    //s15;
    4'b1100: Q_next <= s0;
    4'b1101: Q_next <= s5;
    4'b1110: Q_next <= s10;
    default: Q_next <= 2'bxx;
endcase
```

```
// output equations:
//

assign G = Q[1] & Q[0];

endmodule
```