Lecture 13: Finite State Machines II

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

Some slides are courtesy of Prof. Lin or Prof. Karna ECE111 courses

Important announcements (Details in Canvas)

- **Prof office hours next week:** Tues 11/12 from 12-2pm or by email appointment
- **TA office hours:** are now MWF 9-11am for Fall’24
  - Zoom Meeting ID: 948 6397 0932; Passcode 004453
  - https://ucsd.zoom.us/j/94863970932?pwd=iXcQXbLYjOaAQTdOJlrmglCYMSyeir.1
- **TA Discussion session:** Wed 4-4:50PM (in person) Location: CNTRE 214
- **Nov 5:** Homework 6 posted on Canvas
  - Due on Wed Nov **11/13/24**
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 6 overview

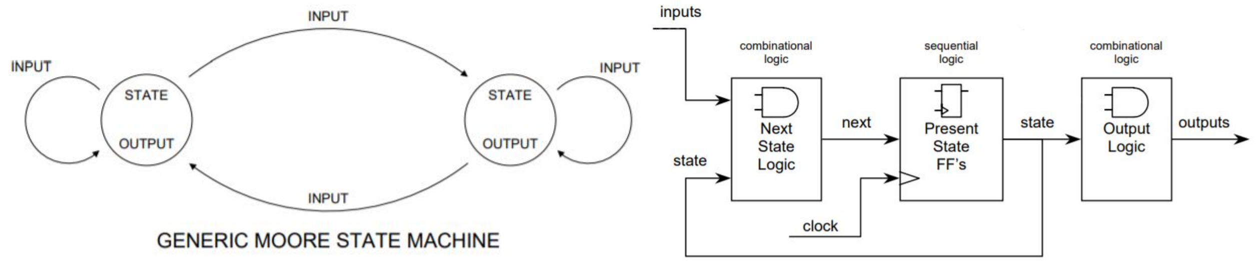
- Design of a synthesizable SystemVerilog code for a Vending Machine (Moore and Mealy), and Synchronous FIFO, and design a system for UART communication protocol by implementing Rx-Tx communication system and a UART controller.
- You will learn how to:
  - Create synthesizable SystemVerilog code
  - Better learn how to use testbenches
  - Design functional SystemVerilog code that can compile post synthesis.
- There will be three parts for this homework:
  - **Homework-6a**: Developing a Synthesizable SystemVerilog model for a of a Vending Machine (Moore and Mealy FSM).
  - **Homework-6b**: Developing a synthesizable SystemVerilog model of a synchronous FIFO.
  - **Homework-6c:** Developing a synthesizable SystemVerilog model of a UART communication system with a Receiver, Transmitter, and a UART controller.

Recap

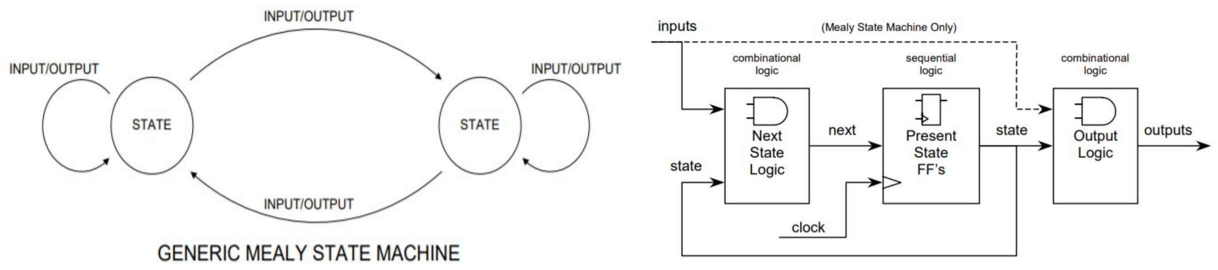
Finite State Machines (FSMs)

- Moore FSM:
  - 2 always block
  - 3 always block
  - 1 always block (delayed output)
- Mealy FSM:
  - 2 always block approach

Moore



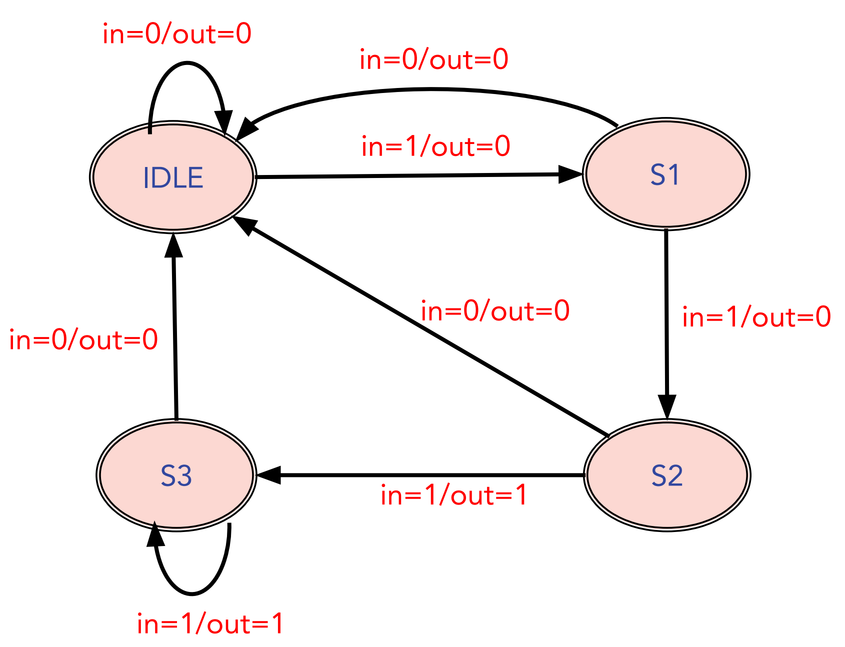
Mealy

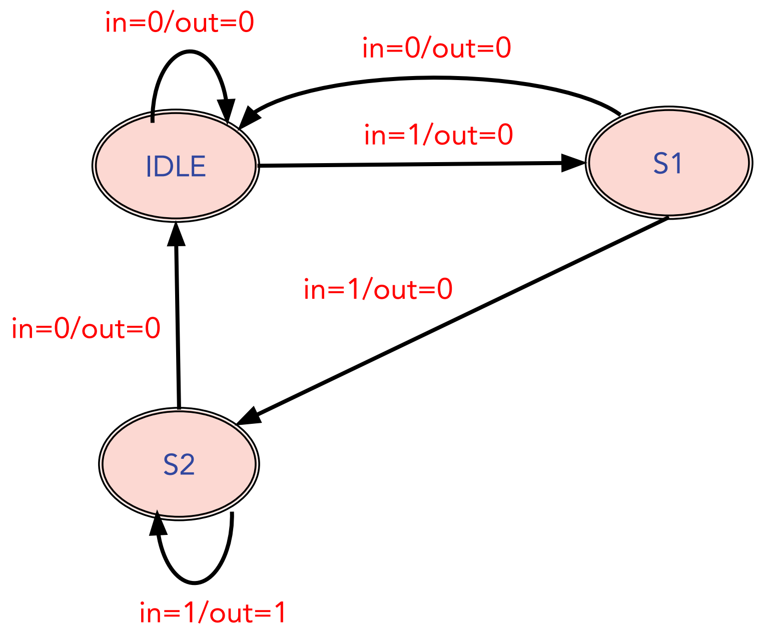


True or False?

The FSMs corresponding to the following two state diagrams are equivalent

- ❑ True
- ❑ False





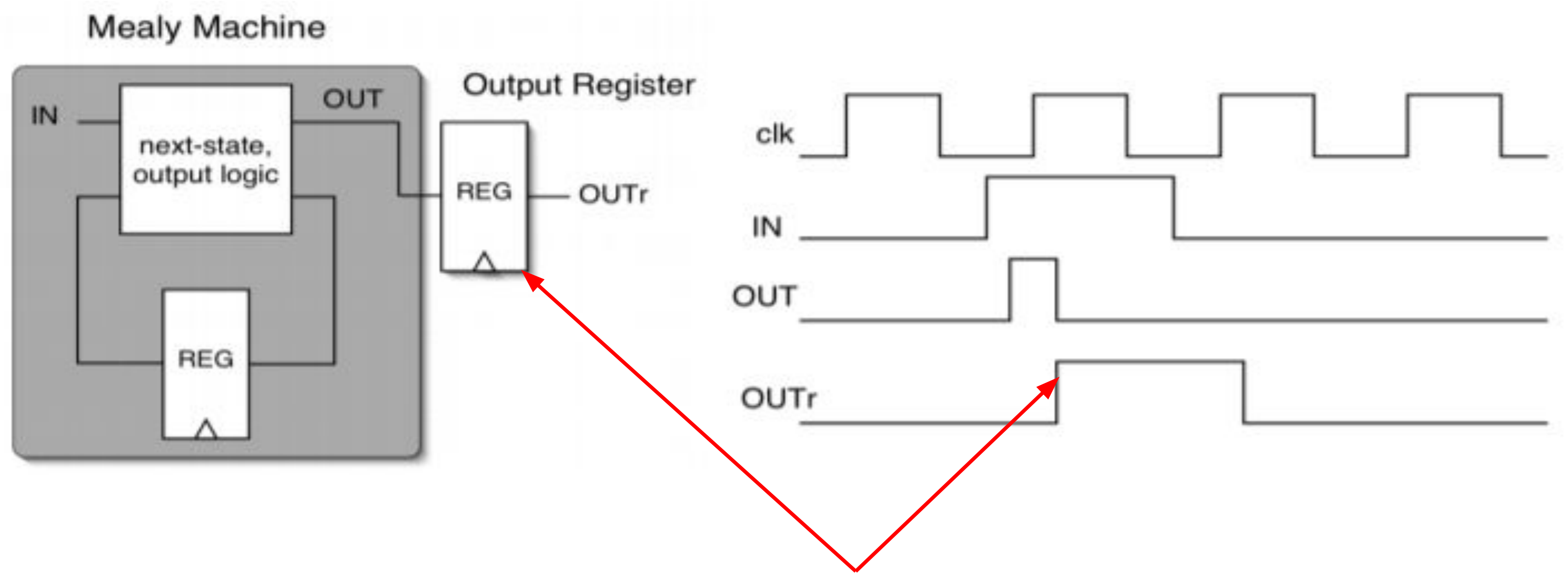
Multiple Choice Questions

Select which of the following mentioned statements are true for Moore and Mealy finite state machines. Select **all** statements which are true :

- ☐ Output from Moore FSM depends only on current state
- ☐ Moore FSM output reacts to input immediately
- ☐ Registering Mealy FSM output is equivalent to Moore FSM output
- ☐ Output from Mealy FSM depends on both present state and inputs

Registering Mealy Machine = Moore Machine!

- The output timing behavior of the Moore machine can be achieved in a Mealy machine by "**registering**" the Mealy output values i.e., by having a **flipflop** at the output signal



Registered output of Mealy matches the output timing behavior of Moore machine

State Encoding

State Encoding

- States in FSM are represented by **encoded value**. There are many choices:
    1. Binary encoding
    2. One-hot encoding
    3. Gray encoding
    4. Johnson encoding
    5. Etc.

$\left.\begin{array}{l}1.\ \text{Binary encoding}\\ 2.\ \text{One-hot encoding}\end{array}\right\}$ Most common!

Binary and One-Hot Encoding

Binary encoding

- For **N** states, use **ceil($\log_2$N)** bits to encode the states, with each state represented by a unique combination of the bits

- Example (with 4 states):
  - **enum** logic [1:0] {**RESET** = 2'b00, **WAIT** = 2'b01, **LOAD** = 2'b10, **DONE** = 2'b11} next_state

- Tradeoffs:
  - Most **efficient use of state registers** (less number of Flipflops are required)
  - **Slower** and **more complicated** combinational logic to detect when in a particular state

One-hot encoding

- For **N** states, **N** bits to encode the states, with bit corresponding to the current state set 1 and all the other bits set to 0

- Example (with 4 states):
  - **localparam RESET** = 4'b0001, **WAIT** = 4'b0010, **LOAD** = 4'b0100, **DONE** = 4'b1000;

- Tradeoffs:
  - Usually leads to **simpler** and **faster** designs as there is much less combinational logic for state decoding (generally a good choice to optimize for speed and power)
  - **Costly** in terms of the number for Flipflops required for a large number of states => **more area**!

State Encoding Constants and Variable Declaration

1. Using enumeration

```
enum logic[1:0] {WAIT=2'b00, EDGE=2'b01}
present_state, next_state;
```

2. Using parameter

```
parameter  logic [1:0]  WAIT = 2’b00, EDGE =
2’b01;
logic [1:0] present_state, next_state;
```

3. Using local parameter

```
localparam  logic [1:0]  WAIT = 2’b00, EDGE =
2’b01;
logic [1:0] present_state, next_state;
```

4. Using typedef (SystemVerilog only)

```
typedef enum logic [1:0] {WAIT=2'b00,
EDGE=2'b01} e_states;
e_states present_state, next_state
```

One-Hot FSM Implementation Example

```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state       <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE]  :  if (req)        next[BBUSY] = 1'b1;
                      else            next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)      next[BBUSY] = 1'b1;
                      else if ( dly)  next[BWAIT] = 1'b1;
                      else            next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)       next[BFREE] = 1'b1;
                      else            next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)        next[BBUSY] = 1'b1;
                      else            next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Index into the state register, not state encodings

Onehot requires larger declarations

Reset modification

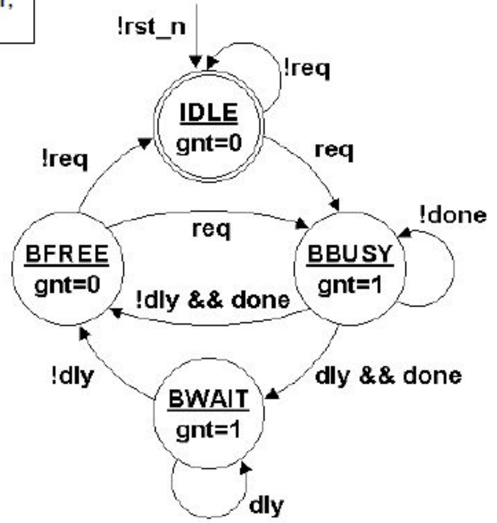
Must make all-0's assignment

case (1'b1)

Add "full" & "parallel" case

state[*current_state*] case items

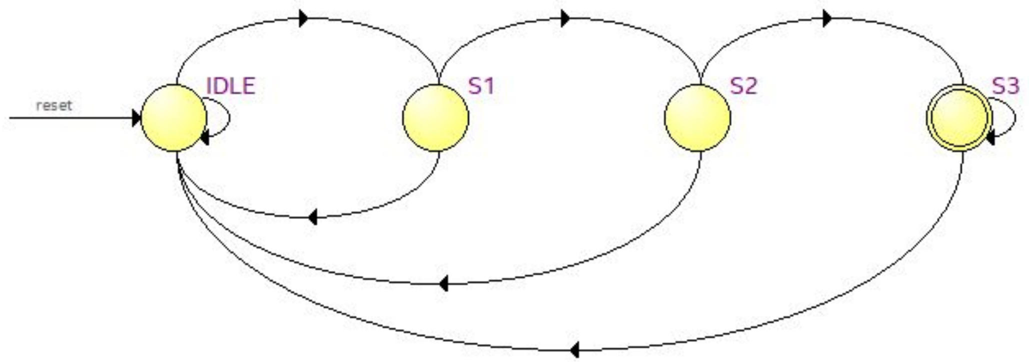
Only update the next[*next state*] bit



One-Hot State Encoding

- In case of **one-hot encoding style** FSM, Quartus Prime **will not auto-generate FSM state diagrams**!
  - **Example:** if sequence detector state machine state encoding is changed from binary counting to one hot encoding style as described below, no state machine diagram generated however it will still work as an FSM

```
parameter[3:0]  IDLE=4’b0001, S1=4’b0010,  S2=4’b0100, S3=1000;
logic[3:0] present_state, next_state;
```



Sequence Detector State machine diagram will not be generated by Quartus Synthesizer

Functionally logic will still behave as a correct FSM !

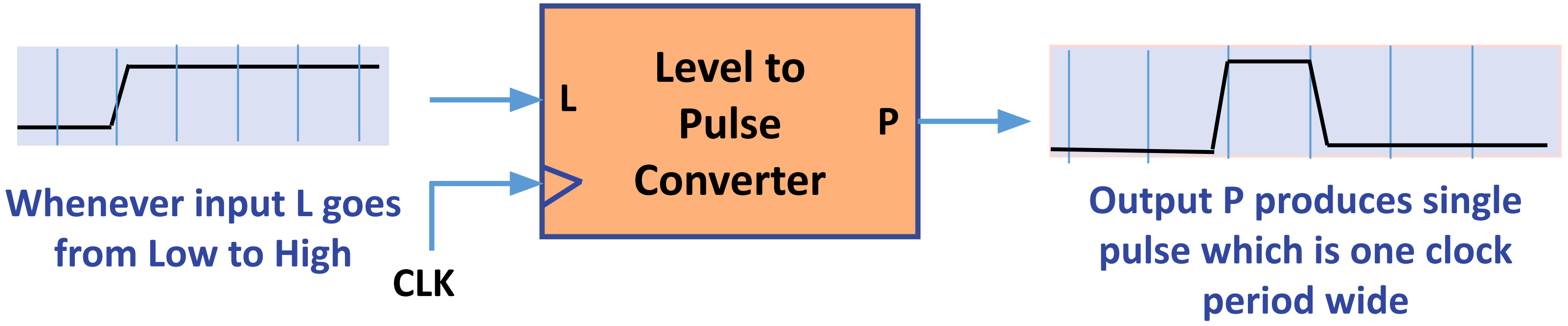
FSM Design Steps

FSM Design Steps Using SystemVerilog

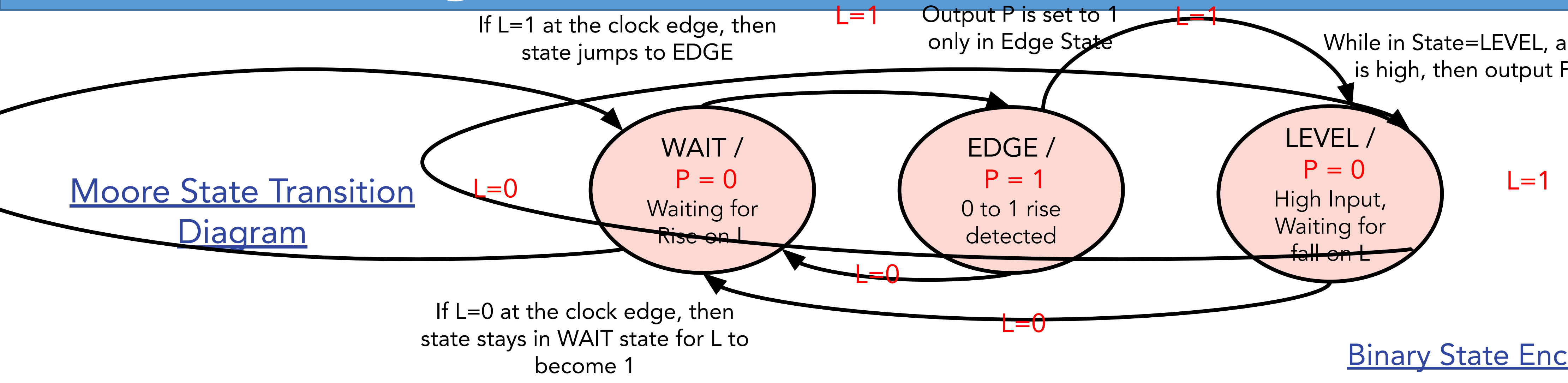
1. Specify circuit **function**
2. Draw **state transition** diagram
3. **Minimize** number of states (later lecture!)
4. Derive **state transition** table
5. Determine **next state** and **output function**
   - For Moore style machine make outputs dependent only on state not dependent on inputs
6. Assign **encodings** (bit patterns) to symbolic states
7. **Implement** State Machine using SystemVerilog :
   - Use either of the **parameters/enum/localparam/typedef** to represent encoded states.
   - Use **separate always blocks** for next state register assignment and combinational logic blocks
   - Use **always_comb** for all combinational block specification
   - Use **case statements** within combinational block (including **default case** item expression to avoid latches) to assign all outputs and next state value based on inputs

Level to Pulse Converter

- **A level-to-pulse converter produces a single cycle pulse each time its input goes high**
  - It is also known as **synchronous rising edge detector**
  - Example use-case: pushing a button of a signal traffic light controller at pedestrian crossing; the pedestrian pressing the button for ab arbitrary period of time should generate a single-cycle enable signal for counters in traffic light controller system

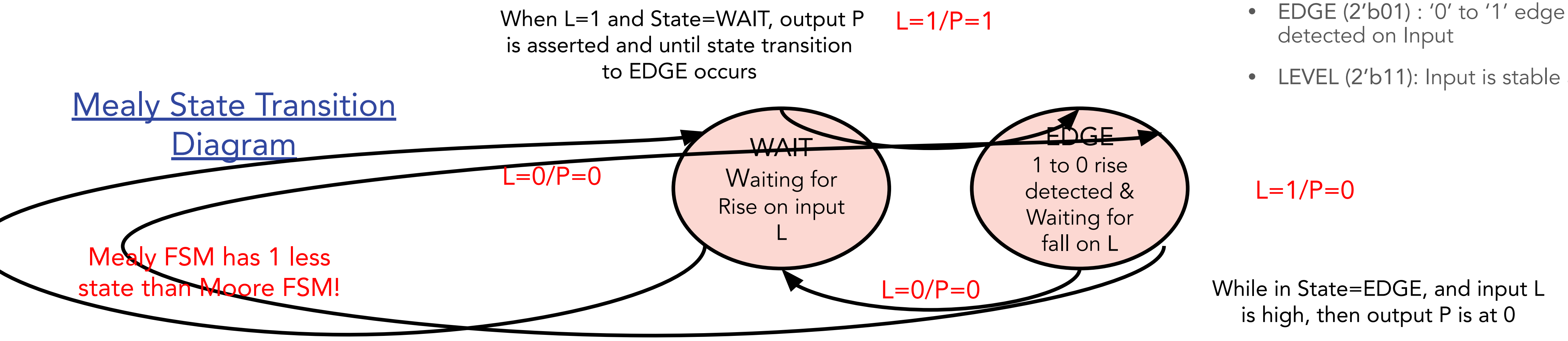

Whenever input L goes from Low to High
CLK
L
Level to Pulse Converter
P
Output P produces single pulse which is one clock period wide

Level to Pulse Converter : Moore and Mealy State Diagrams


If L=1 at the clock edge, then state jumps to EDGE
L=1
Output P is set to 1 only in Edge State
L=1
While in State=LEVEL, and input L is high, then output P is at 0
Moore State Transition Diagram
L=0
WAIT / P = 0 Waiting for Rise on L
EDGE / P = 1 0 to 1 rise detected
LEVEL / P = 0 High Input, Waiting for fall on L
L=1
L=0
L=0
If L=0 at the clock edge, then state stays in WAIT state for L to become 1

Binary State Encoding

- WAIT (2'b00) : Input is '0', wait for '1'
- EDGE (2'b01) : '0' to '1' edge detected on Input
- LEVEL (2'b11): Input is stable at '1'


When L=1 and State=WAIT, output P is asserted and until state transition to EDGE occurs
L=1/P=1
Mealy State Transition Diagram
L=0/P=0
WAIT Waiting for Rise on input L
EDGE 1 to 0 rise detected & Waiting for fall on L
L=1/P=0
Mealy FSM has 1 less state than Moore FSM!
L=0/P=0
While in State=EDGE, and input L is high, then output P is at 0

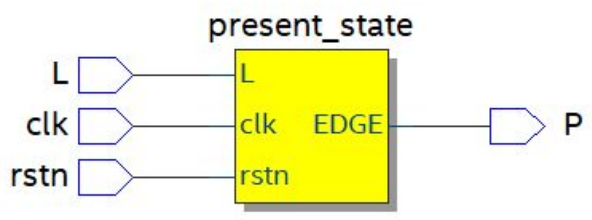
Level to Pulse Converter (Moore FSM)

```
module level_to_pulse_converter_moore(
 input logic clk, rstn,
 input logic L,
 output logic P);

 // FSM state encodings and state registers declaration
enum logic[1:0] {WAIT=2'b00,
                 EDGE=2'b01,
                 LEVEL=2'b11} present_state, next_state;

 // Sequential Logic for present state
always_ff@(posedge clk) begin
  if(!rstn)
    present_state <= WAIT;
  else
   present_state <= next_state;
 end

 (continued….)
```

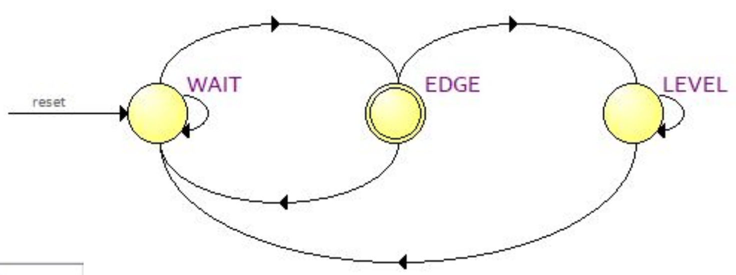
```
 // Combination Logic for Next State and Output
always_comb begin
 case(present_state)
  WAIT: begin
    P = 0;
    if(L==1) next_state = EDGE;
    else next_state = WAIT;
  end
  EDGE: begin
   P = 1;
   if(L==1) next_state = LEVEL;
   else next_state = WAIT;
  end
  LEVEL: begin
   P = 0;
   if(L==1) next_state = LEVEL;
   else next_state = WAIT;
  end
  default: begin
   P = 0;  next_state = WAIT;
  end
 endcase
end
endmodule: level_to_pulse_converter_moore
```

Level to Pulse Converter Simulation and Synthesis Results (Moore FSM)

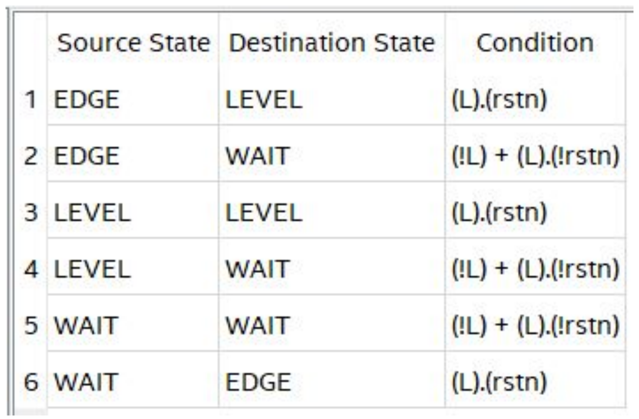
Post Synthesis RTL Netlist Schematic



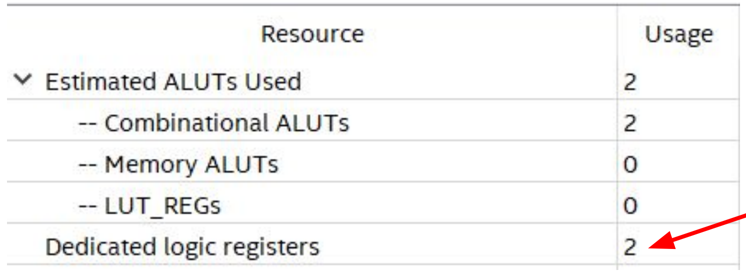
Moore State Diagram



Mealy State Transition Table



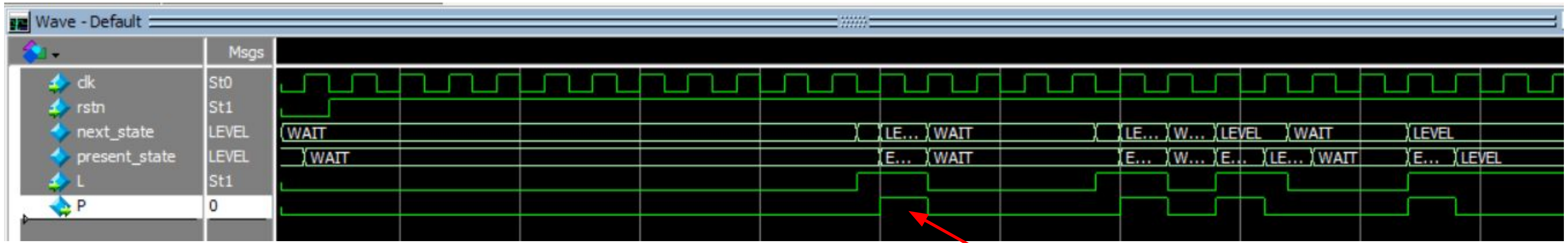
	Source State	Destination State	Condition
1	EDGE	LEVEL	(L).(rstn)
2	EDGE	WAIT	(!L) + (L).(!rstn)
3	LEVEL	LEVEL	(L).(rstn)
4	LEVEL	WAIT	(!L) + (L).(!rstn)
5	WAIT	WAIT	(!L) + (L).(!rstn)
6	WAIT	EDGE	(L).(rstn)



Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	2

2 Flipflops to represent three states WAIT EDGE, LEVEL

Simulation Waveform



Output P is a single cycle pulse, available sometime after there is a rising edge on input L

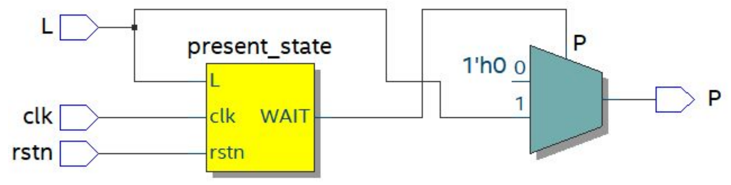
Level to Pulse Converter (Mealy FSM)

```
module level_to_pulse_converter_mealy(
 input logic clk, rstn,
 input logic L,
 output logic P);

 // FSM state encodings and state registers declaration
 enum logic[1:0] WAIT=2'b00,
                  EDGE=2'b01} present_state,
next_state;


 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn)
    present_state <= WAIT;
  else
   present_state <= next_state;
 end

 (continued....)
```

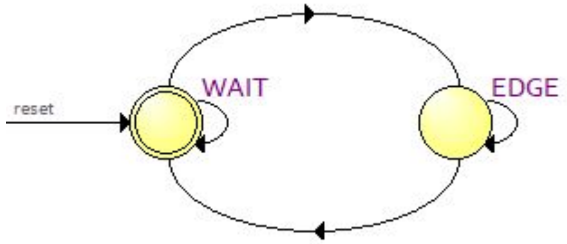
```
 // Combination Logic for Next State and Output
 always_comb begin
 case(present_state)
   WAIT: begin
     if(L==1) begin
       next_state = EDGE; P = 1;
     end
     else begin
       next_state = WAIT;  P = 0;
     end
   end
   EDGE: begin
     if(L==1) begin
       next_state = EDGE; P = 0;
     end
     else begin
       next_state = WAIT; P = 0;
     end
  end
  default: begin P = 0; next_state = WAIT; end
 endcase
end
endmodule: level_to_pulse_converter_mealy
```

Level to Pulse Converter Simulation and Synthesis Results (Mealy FSM)

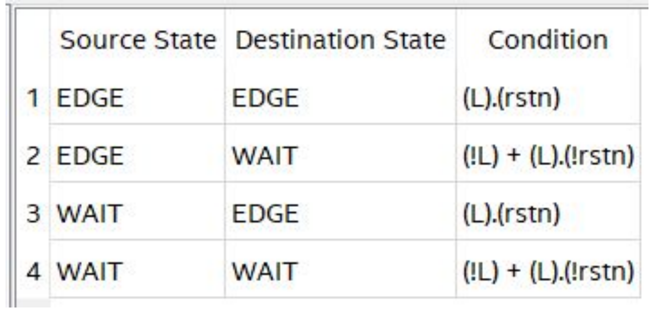
Post Synthesis RTL Netlist Schematic



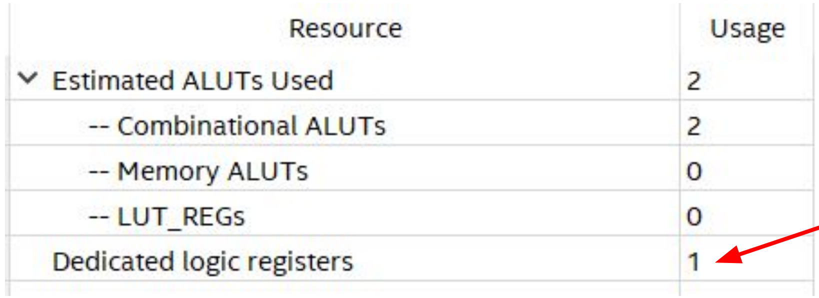
Moore State Diagram



Mealy State Transition Table



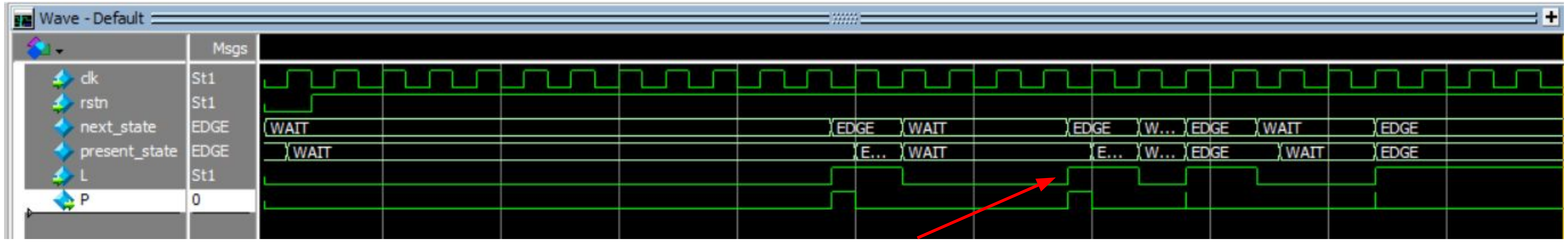
	Source State	Destination State	Condition
1	EDGE	EDGE	(L).(rstn)
2	EDGE	WAIT	(!L) + (L).(!rstn)
3	WAIT	EDGE	(L).(rstn)
4	WAIT	WAIT	(!L) + (L).(!rstn)



Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	1

1 Flipflop to represent two states WAIT and EDGE.
1 less D-flipflop required in Mealy compared to Moore FSM

Simulation Waveform



Output P is available as soon as input L changed from ‘0’ to ‘1’. However output P pulse is not stable for 1 cycle. Which does not meet design requirement for P to be 1 cycle pulse!

Level to Pulse Converter (Mealy FSM + Registered output)

```
module level_to_pulse_converter_mealy(
 input logic clk, rstn,
 input logic L,
 output logic P);

 // FSM state encodings and state registers declaration
 enum logic[1:0] {WAIT=2'b00,
                  EDGE=2'b01} present_state, next_state;

 logic r_P; // local variable declaration

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn) begin
    present_state <= WAIT;
    P <= 0;
  end
  else begin
   present_state <= next_state;
   P <= r_P;
  end
end
(continued....)
```

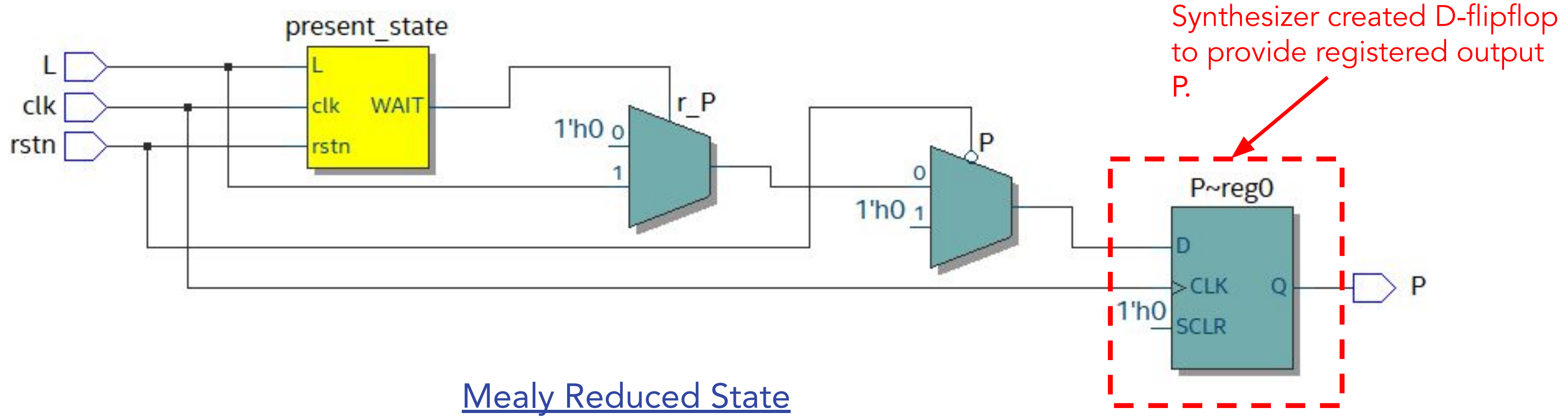
Synthesizer will create D-flipflop to provide registered output P. Output P is registered to ensure P is at least 1 cycle pulse

```
// Combination Logic for Next State and Output
 always_comb begin
 case(present_state)
   WAIT: begin
    if(L==1) begin
      next_state = EDGE; r_P = 1;
    end
    else begin
      next_state = WAIT;  r_P = 0;
    end
   end
   EDGE: begin
    if(L==1) begin
      next_state = EDGE; r_P = 0;
    end
    else begin
      next_state = WAIT; r_P = 0;
    end
  end
  default: begin r_P = 0; next_state = WAIT; end
 endcase
end
endmodule: level_to_pulse_converter_mealy
```

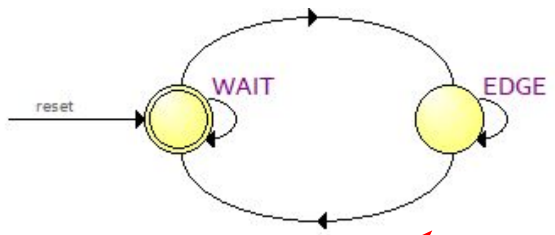
Output is assigned to local variable r_P

Level to Pulse Converter Simulation and Synthesis Results (Moore FSM)

Post Synthesis RTL Netlist Schematic

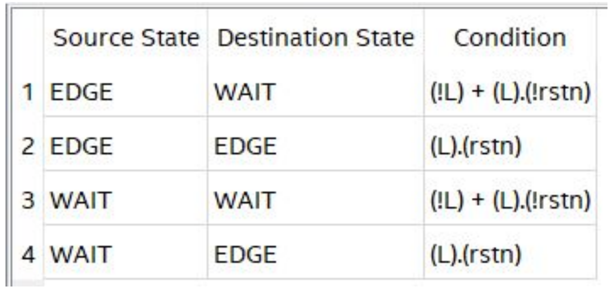


Mealy Reduced State Diagram



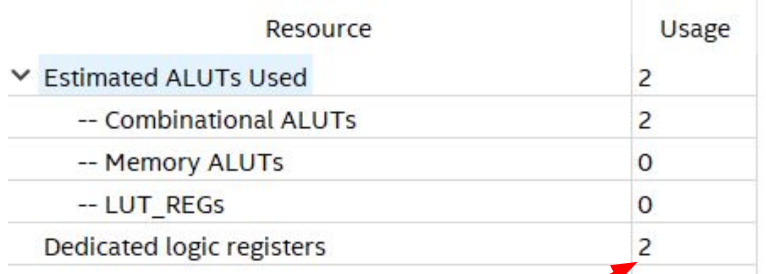
Only 2 states in Mealy FSM compared to Moore which as 3 States

Mealy Reduced State Transition Table



	Source State	Destination State	Condition
1	EDGE	WAIT	(!L) + (L).(!rstn)
2	EDGE	EDGE	(L).(rstn)
3	WAIT	WAIT	(!L) + (L).(!rstn)
4	WAIT	EDGE	(L).(rstn)

Post Synthesis Resource Utilization

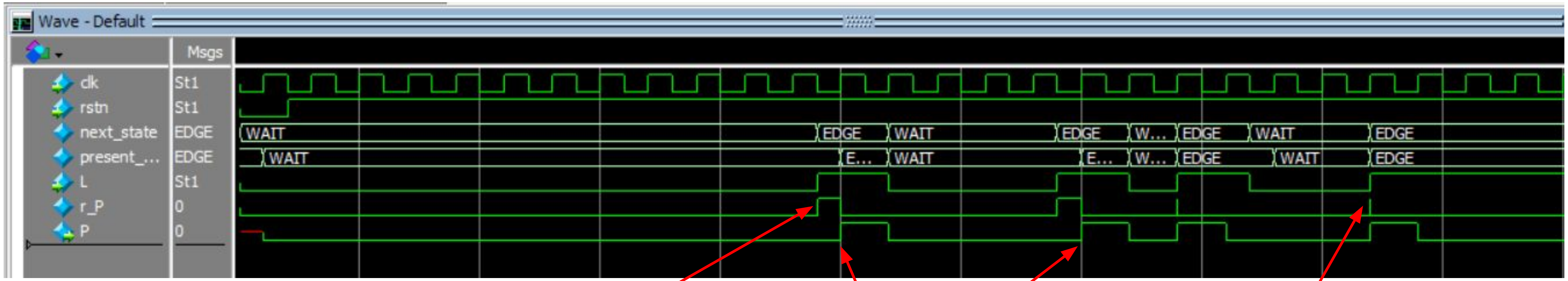


Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	2

1 Flipflop for registering output P and 1 Flipflop for representing two states WAIT and EDGE

Level to Pulse Converter Simulation Result(Mealy FSM) : With Registered Output

Post Synthesis RTL Netlist Schematic



Unregistered output r_P is available as soon as input L changed from '0' to '1'

Registered Output P which is single cycle pulse when there is rising edge on L detected.

Glitches in simulation on unregistered output r_P

True or False?

Two or three always block approaches for Moore FSMs are equivalent

- ☐ True
- ☐ False

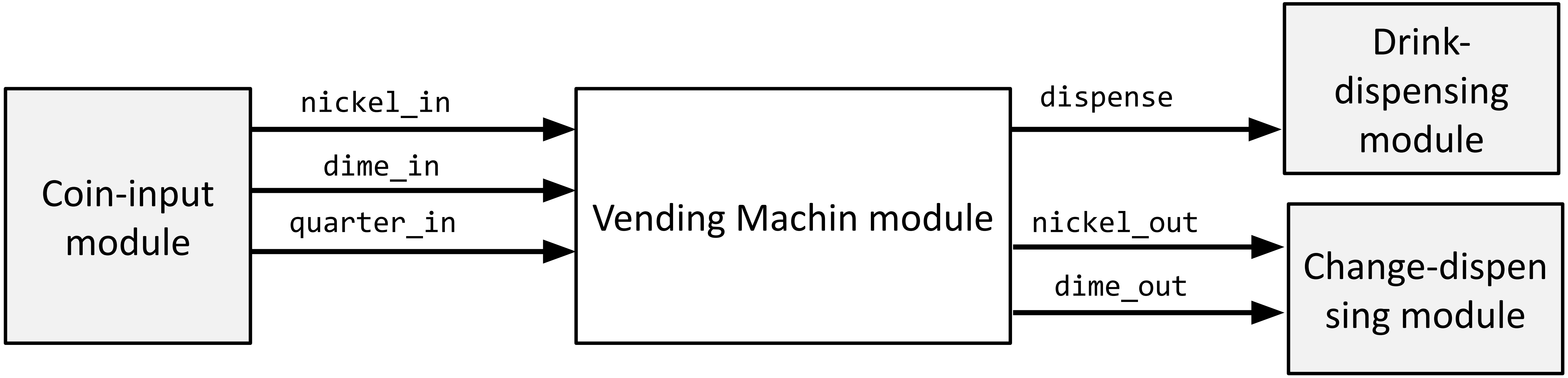
Example 1: Vending Machine

- Vending Machine: A soft drink dispenser
- Accepts Nickel (¢5), Dime (¢10), and Quarter (¢25)
- Dispenses a drink when receives ¢35
- Returns change

Ref: http://www.ece.uc.edu/~wjone/Digital/Drinking_machine.pdf

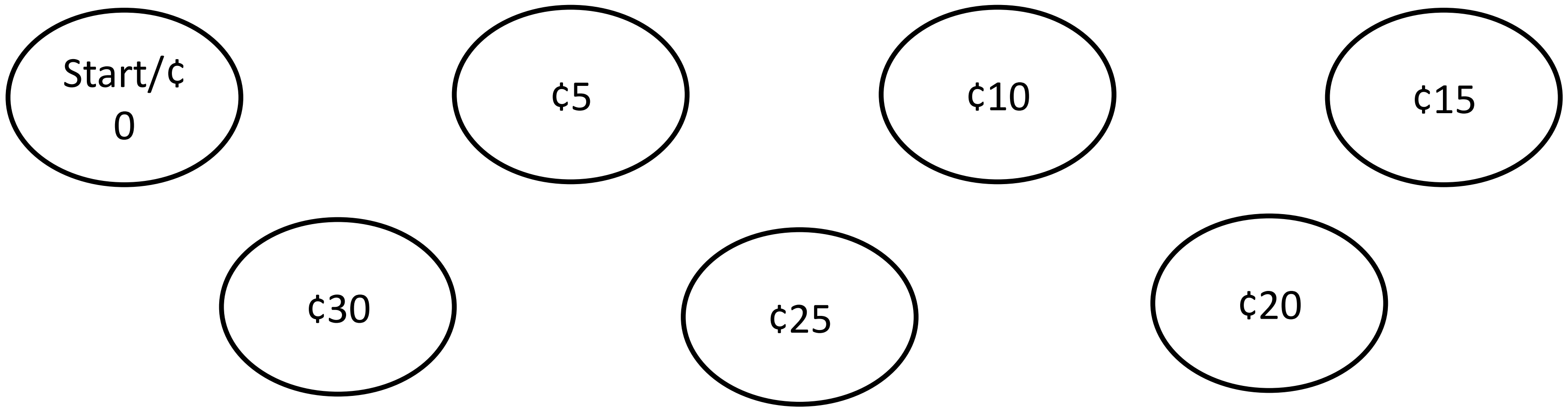
Diagram

- Inputs and outputs
- Other units

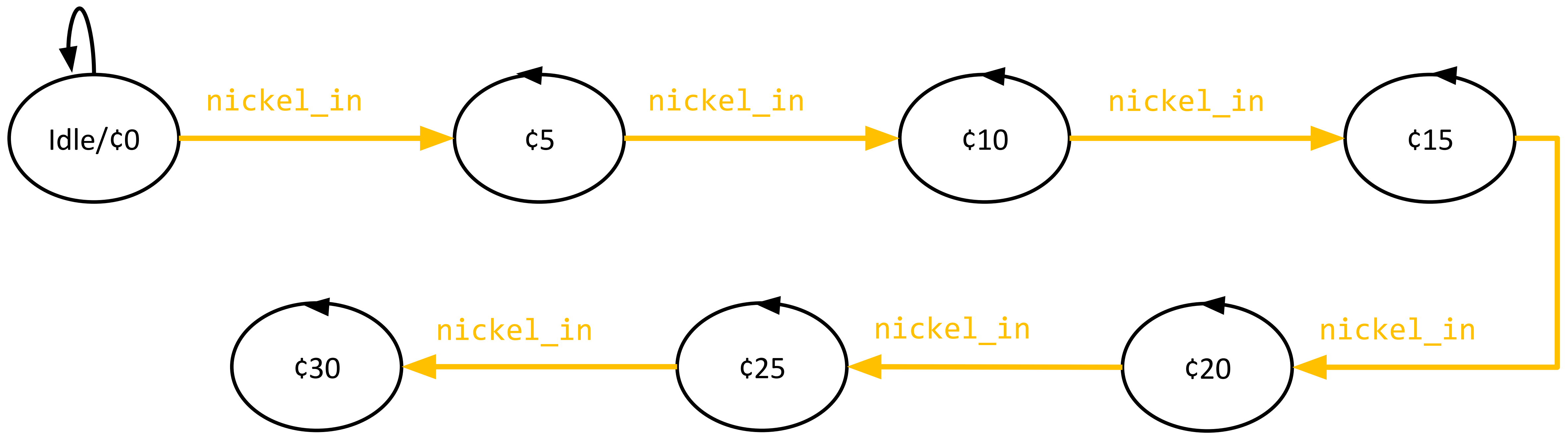


Vending Machine

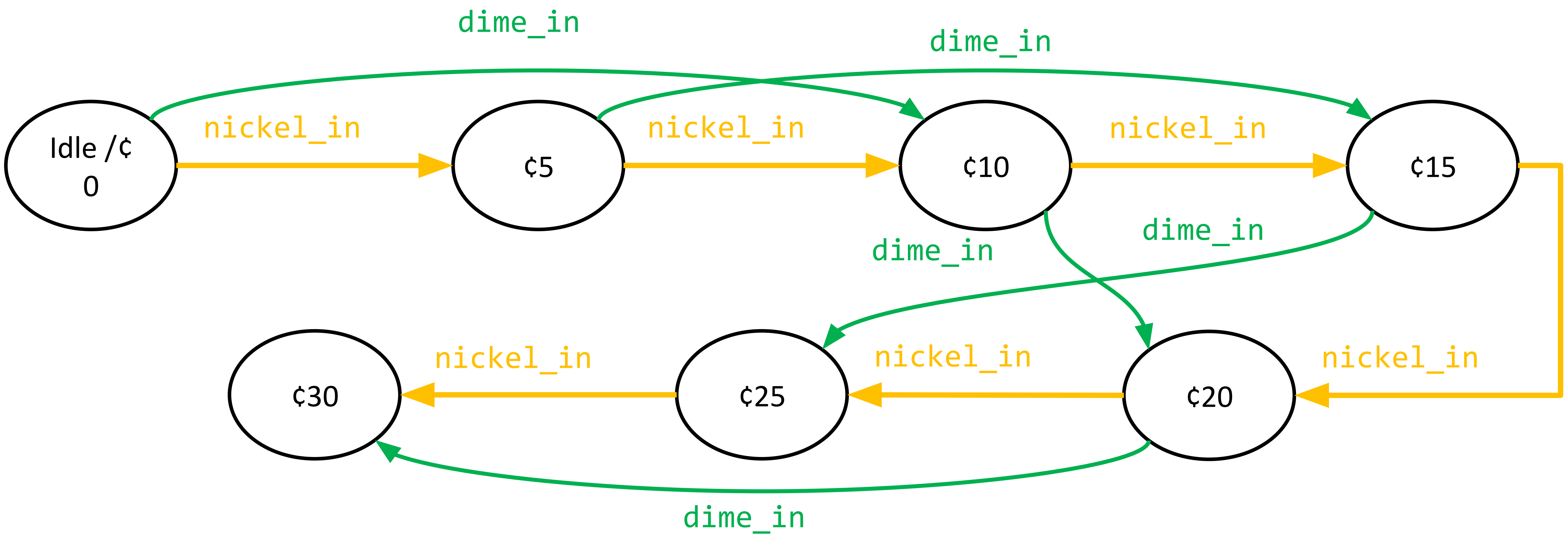
- Let’s draw a Mealy FSM
- The states represent sum of the inserted coins (do we need more?).



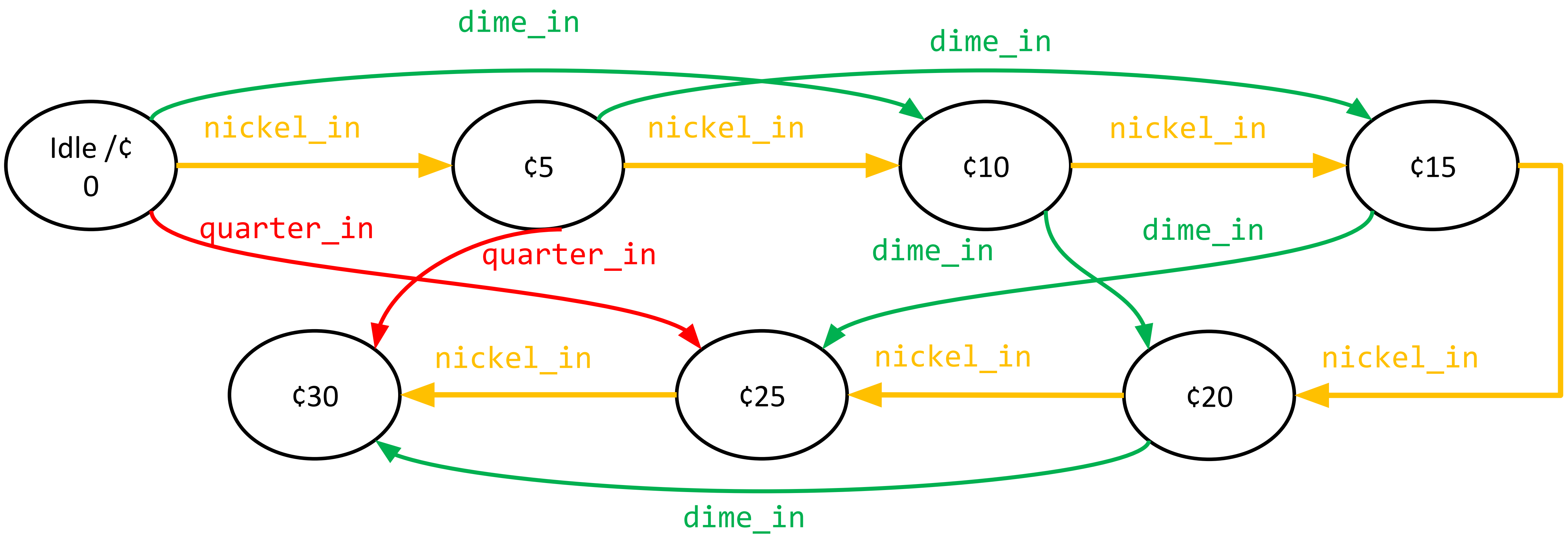
nickel_in


Idle/¢0
nickel_in
¢5
nickel_in
¢10
nickel_in
¢15
nickel_in
¢20
nickel_in
¢25
nickel_in
¢30

dime_in


dime_in
dime_in
Idle /¢
0
nickel_in
¢5
nickel_in
¢10
nickel_in
¢15
dime_in
dime_in
¢30
nickel_in
¢25
nickel_in
¢20
nickel_in
dime_in

quarter_in

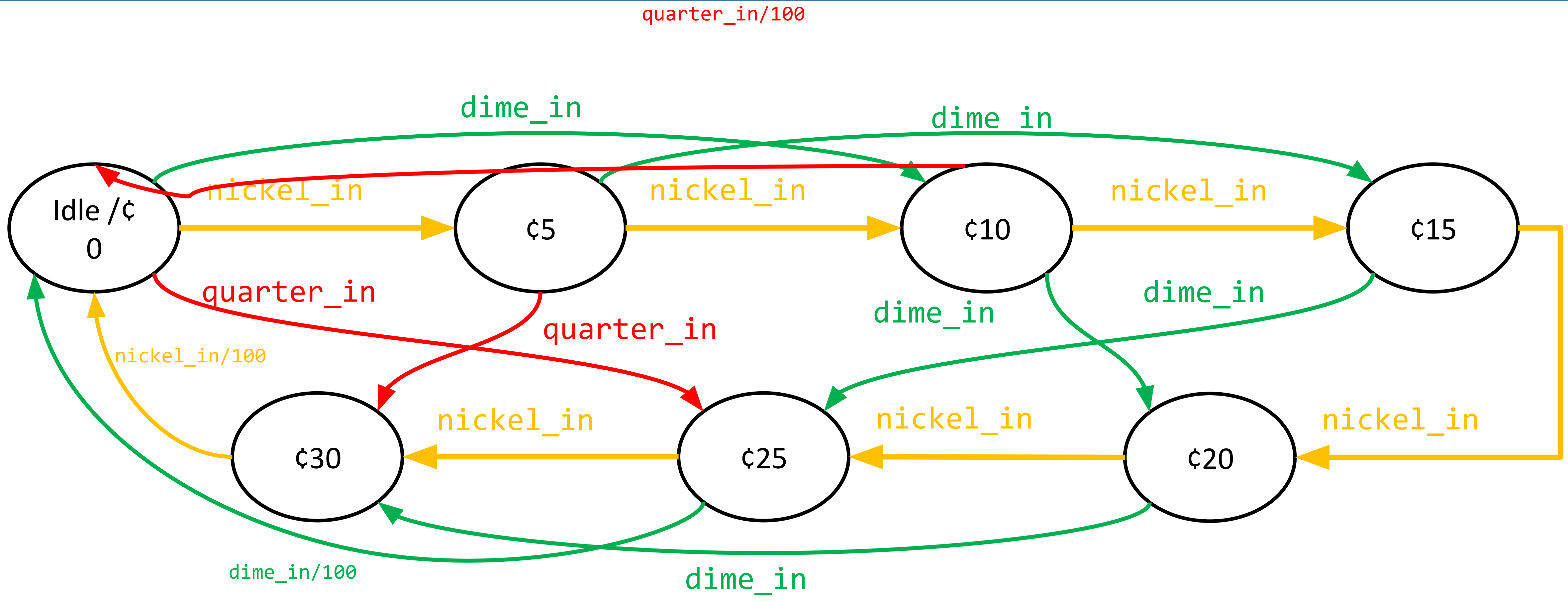


dime_in
dime_in
Idle /¢ 0
nickel_in
¢5
nickel_in
¢10
nickel_in
¢15
quarter_in
quarter_in
dime_in
dime_in
¢30
nickel_in
¢25
nickel_in
¢20
nickel_in
dime_in

Output

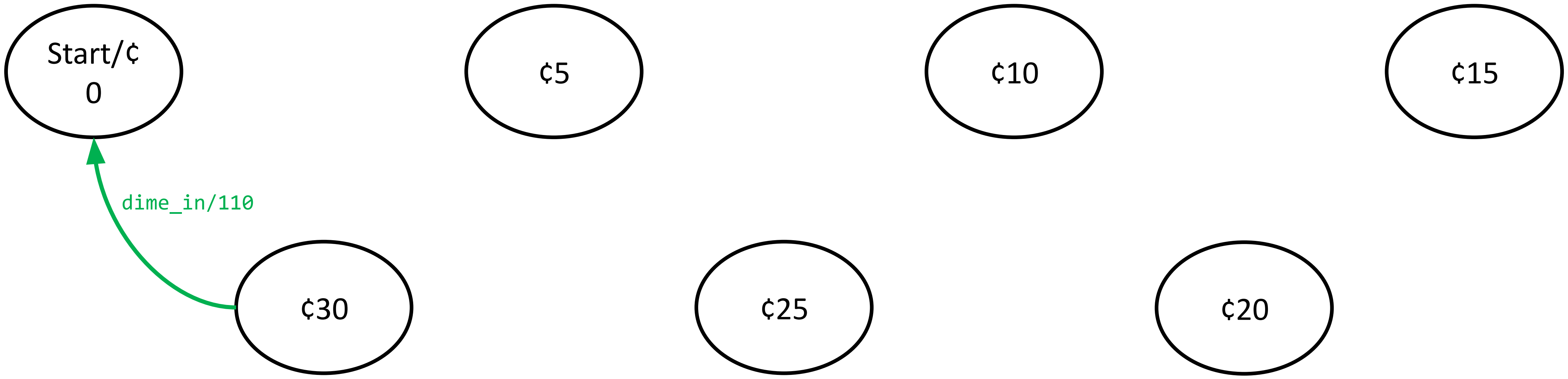
- Output:
  - `{dispense, nickel_out, dime_out}`
  - E.g, output = 3’b110 (/110) means dispense and return a nickel.

Output/

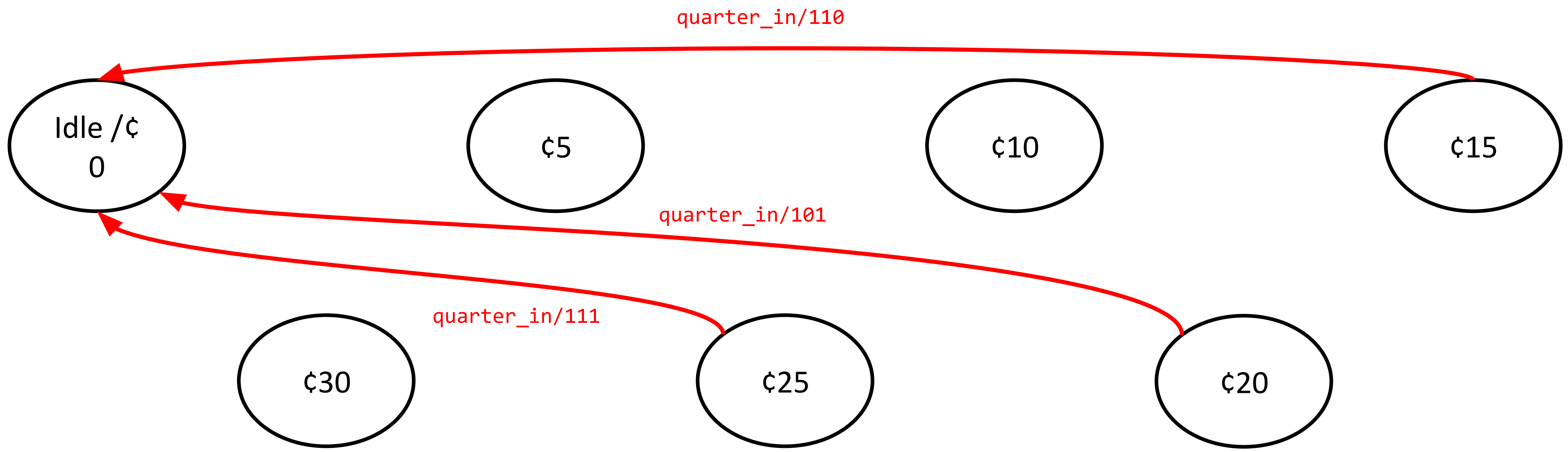


quarter_in/100
dime_in
dime_in
Idle /¢ 0
nickel_in
¢5
nickel_in
¢10
nickel_in
¢15
quarter_in
quarter_in
dime_in
dime_in
nickel_in/100
¢30
nickel_in
¢25
nickel_in
¢20
nickel_in
dime_in/100
dime_in

Change Dime

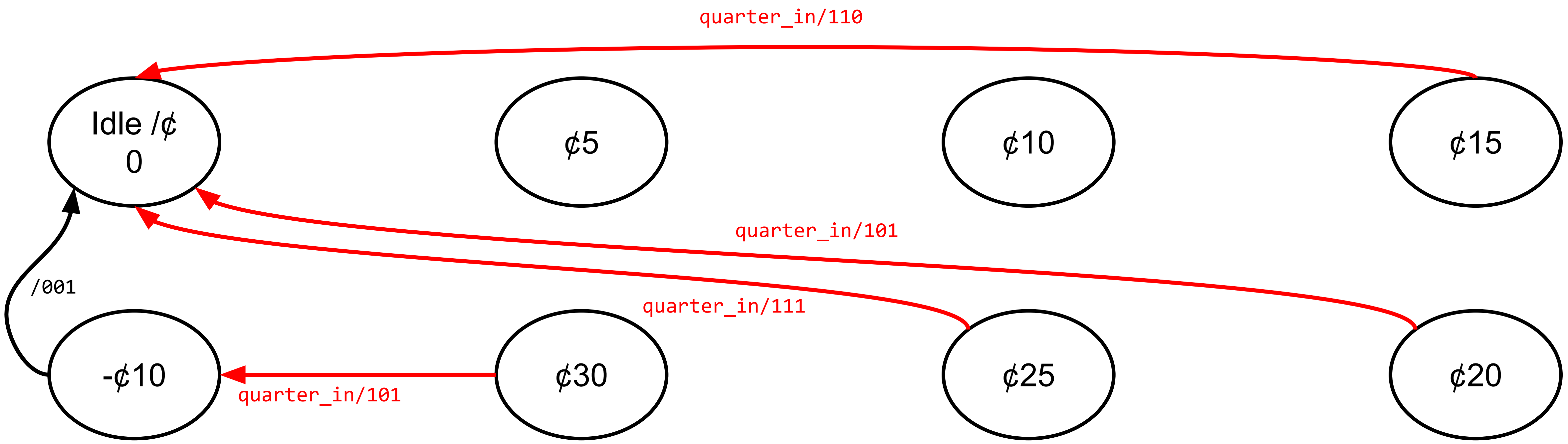


Change Quarter



Ops! what about ¢30 with a quarter? 30+25 = 35 + 10 + 10
We need one more state.

Owe ¢10



In -¢10, we owe the user ¢10.