



# State Machines in VHDL

# Announcements

- Homework #10 due Wednesday
- Exam #3 – 1 week from today
  - Material through Wednesday
  - Study Jam over the weekend (details to follow)
- Read Chapter 11 Sections 1-4

# Enumerated Types

- The **Enumeration** type is a type whose values are defined by listing (enumerating) them explicitly
- It is a user-named and user-defined type
- Syntax:
  - **Type identifier is** (enumeration\_literal {, enumeration\_literal})
  - There must be at least one literal in the list

# Enumerated Type Examples

- Predefined
  - `TYPE BIT IS ('0', '1');`
  - `TYPE BOOLEAN IS (FALSE, TRUE);`
  - `TYPE STD_LOGIC IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X');`
- User Defined
  - `TYPE STATE_TYPE IS (IDLE, QUARTER, NICKEL, DIME);`
- Using enumerated type
  - `SIGNAL current_state, next_state : state_type;`
    - Current\_state and next\_state are two signals of type state\_type
    - They can only ever be assigned one of the four values defined in the type declaration.
    - Assign by name. ex: `current_state <= nickel;`

# Enumerated Type for State Machines

- **TYPE** state\_type **IS** (wait1, nickel, dime, quarter, enough, excess, vend, change);
  - The designer picks the names for the states
  - The synthesizer assigns the encoding
  - For one-hot encoding, 8 states would be encoded with 8 bits
- **SIGNAL** current\_state, next\_state : state\_type;
  - Current\_state and next\_state are two signals of type state\_type
  - They can only ever be assigned one of the eight values defined in the type declaration.
  - Assign by name. ex: current\_state <= nickel;

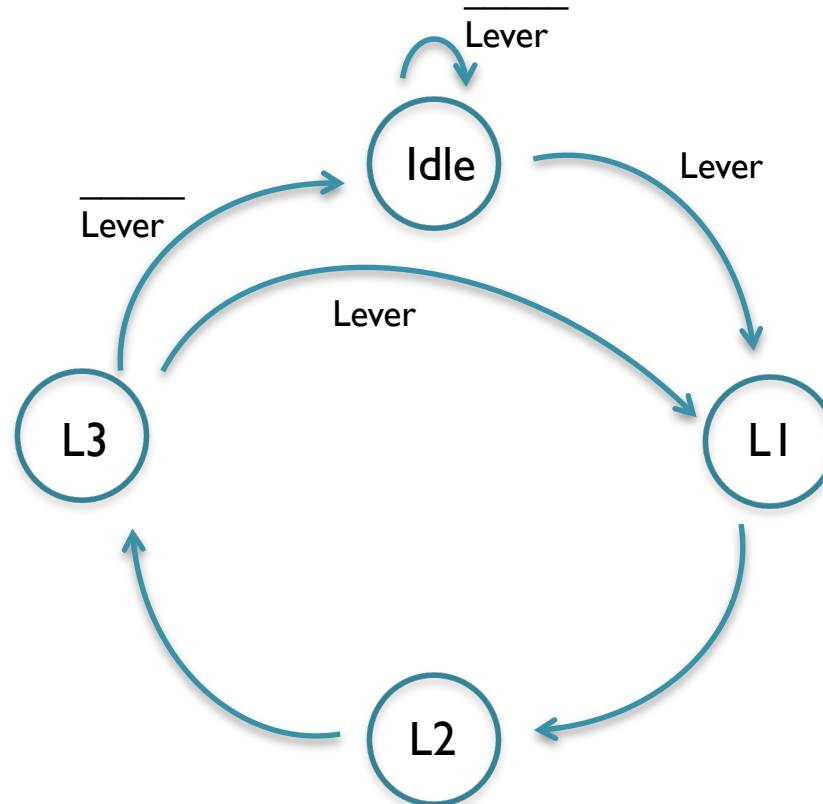
# Problem statement

Design a state machine to control a 3 bulb right turn signal on a Mustang. The sequence of lights is as follows:

ooo                    \* = lit   o = off  
\*oo  
\*\*o  
\*\*\*  
\*oo  
\*\*o  
repeat

When the directional lever on the steering column is pushed up, the sequence begins. It continues until the lever returns to the neutral position. Once the lever returns to the neutral position, the controller will not return to the all off state until the sequence finishes (i.e. all bulbs are on). In other words, the sequence will always finish.

# State Transition Diagram



# VHDL

```
ENTITY turn_signal IS
PORT( reset_n, clk, lever :IN STD_LOGIC;
      bulb :OUT STD_LOGIC_VECTOR(2 downto 0));
END turn_signal;

ARCHITECTURE behave OF turn_signal IS

TYPE state_type IS (IDLE,L1,L2,L3); --enumerated type for the 4 possible states
SIGNAL current_state, next_state : state_type;
```



# VHDL (con't)

```
BEGIN
```

```
-- The following process clocks the state flip flops. Its purpose is to move the state  
-- machine to the next state. Nothing else belongs here
```

```
sync: PROCESS(clk, reset_n)
```

```
BEGIN
```

```
  if (reset_n = '0') then
```

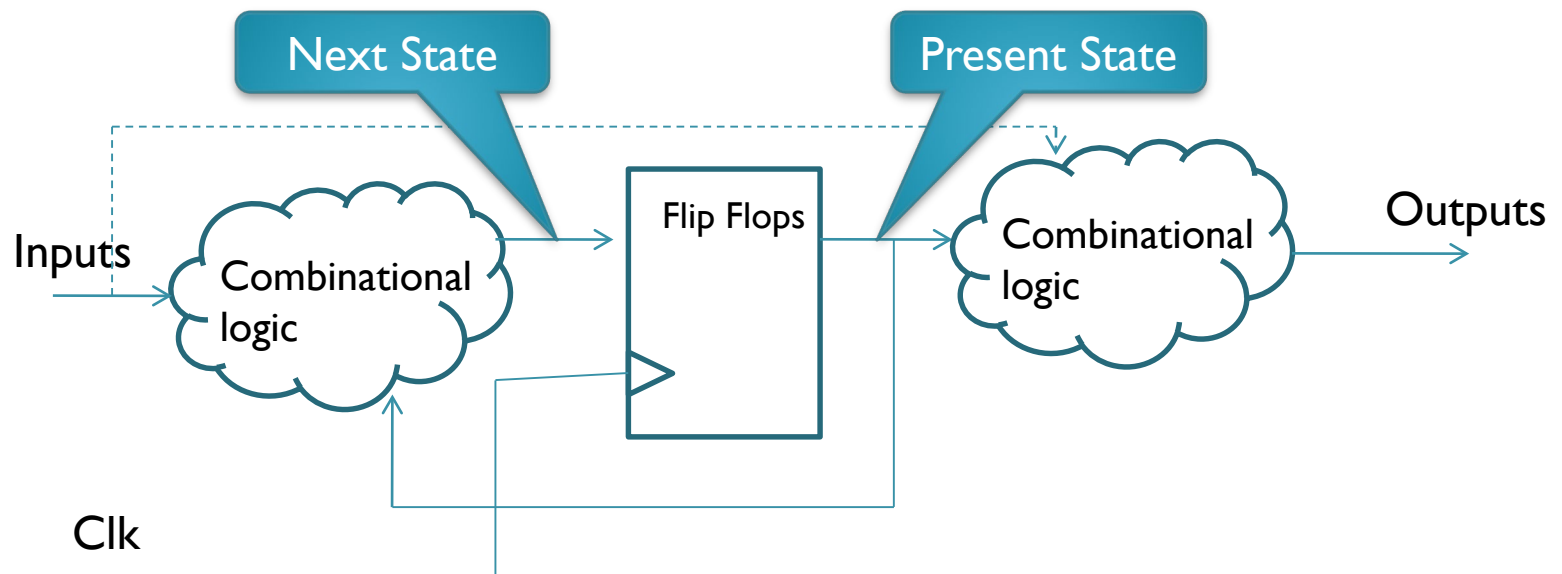
```
    current_state <= IDLE;  --OFF is the reset or default state
```

```
  elsif (clk'event and clk = '1') then
```

```
    current_state <= next_state;  --advance the state machine
```

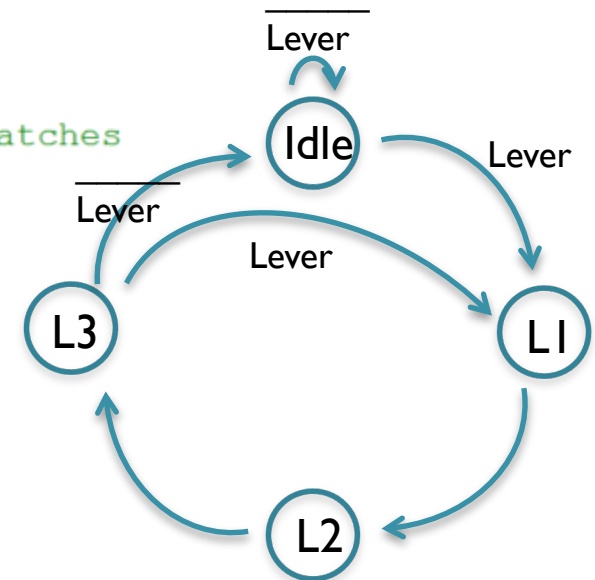
```
  end if;
```

```
END PROCESS;
```



--The following process sets up the conditions to transition to the next state.  
--The next state is determined by the current state and the inputs.  
--This process follows the state transition diagram exactly.  
--No other outputs belong here.

```
comb: PROCESS(current_state, lever)
BEGIN
CASE (current_state) IS
WHEN IDLE =>
    IF (LEVER = '0') THEN --remain off until lever is active
        next_state <= IDLE;
    ELSE --lever = 1
        next_state <= L1;
    END IF;
WHEN L1 => --only one path out of L1
    next_state <= L2;
WHEN L2 => --one path out
    next_state <= L3;
WHEN L3 => --if lever is down, return to idle, else keep blinking
    IF (LEVER = '0') THEN
        next_state <= IDLE;
    ELSE --lever = 1
        next_state <= L1;
    END IF;
WHEN OTHERS => --keep the compiler happy and avoid latches
    next_state <= IDLE;
END CASE;
END PROCESS;
```



# VHDL (con't)

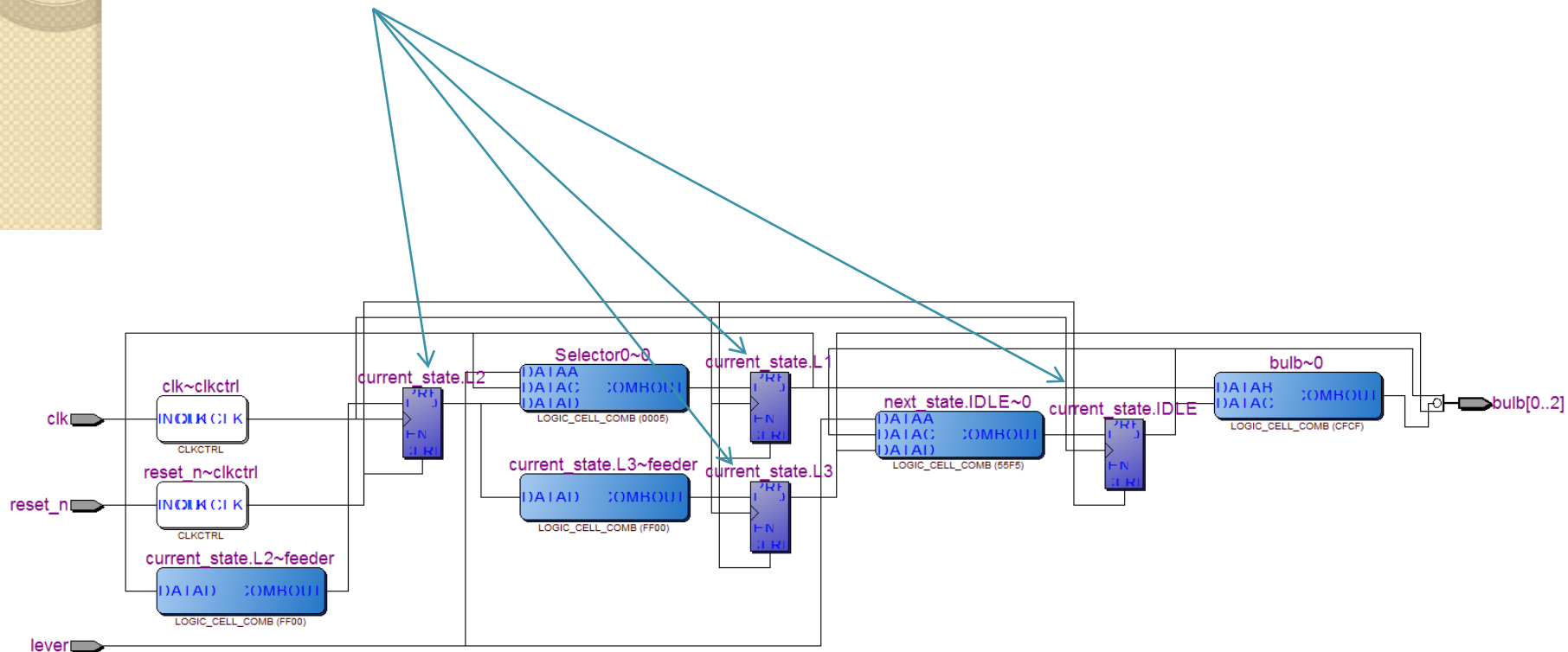
```
outputs: PROCESS(current_state)
--because this is a MOORE state machine, outputs depend on current state only
BEGIN
    CASE(current_state) IS
        WHEN IDLE =>
            bulb <= "000";
        WHEN L1 =>
            bulb <= "100";
        WHEN L2 =>
            bulb <= "110";
        WHEN L3 =>
            bulb <= "111";
        WHEN OTHERS => --keep the compiler happy and avoid latches
            bulb <= "000";
    END CASE;
END PROCESS;
```

# Default Compilation Report

Table of Contents		Flow Summary	
Flow Summary		Flow Status	Successful - Sun Apr 03 13:34:26 2016
Flow Settings		Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Flow Non-Default Global Settings		Revision Name	turn_signal
Flow Elapsed Time		Top-level Entity Name	turn_signal
Flow OS Summary		Family	Cyclone II
Flow Log		Device	EP2C35F672C6
Analysis & Synthesis		Timing Models	Final
Fitter		Total logic elements	5 / 33,216 ( < 1 % )
Flow Messages		Total combinational functions	3 / 33,216 ( < 1 % )
Flow Suppressed Messages		Dedicated logic registers	4 / 33,216 ( < 1 % )
Assembler		Total registers	4 ←
TimeQuest Timing Analyzer		Total pins	6 / 475 ( 1 % )
		Total virtual pins	0
		Total memory bits	0 / 483,840 ( 0 % )
		Embedded Multiplier 9-bit elements	0 / 70 ( 0 % )
		Total PLLs	0 / 4 ( 0 % )

# Default synthesis

Each state has its own ff



# Evil Toaster VHDL

```
--This VHDL module is a state machine that controls the evil toaster.
--The toaster sits in the idle state when not in use
--When the user pushes down the lever, the toaster goes to the toasting state
--When in the toasting state:
-----If timer is done and the user is not Prof. Cliver, the toaster returns to idle state
-----If timer is done and the user is Prof. Cliver, the toaster enters the burning state
--When in the burning state if timer is done, the toaster returns to idle
--Inputs:
--    User (Cliver = 1, not Cliver = 0)
--    Lever (up = 1, down = 0)
--    Timer (done = 1, not done = 0)
--Outputs
--    Coil (on = 1, off = 0)
--    Flame thrower (on = 1, off = 0)
--States
--    Idle
--    Toasting
--    burning

ENTITY evil_toaster IS
    PORT( reset_n, clk, lever, Cliver, timer    :IN STD_LOGIC;
          coil, flame_thrower                  :OUT STD_LOGIC);
END evil_toaster;
```

# Evil Toaster VHDL

```
ARCHITECTURE behave OF evil_toaster IS
```

```
    TYPE state_type IS (IDLE, toasting, burning); --enumerated type for the 3 possible states
    SIGNAL current_state, next_state : state_type;
    BEGIN
```

```
-- The following process clocks the state flip flops. Its purpose is to move the state
-- machine to the next state. Nothing else belongs here
```

```
sync: PROCESS(clk, reset_n)
```

```
    BEGIN
```

```
        if (reset_n = '0') then
```

```
            current_state <= IDLE;    --OFF is the reset or default state
```

```
        elsif (clk'event and clk = '1') then
```

```
            current_state <= next_state; --advance the state machine
```

```
        end if;
```

```
    END PROCESS;
```

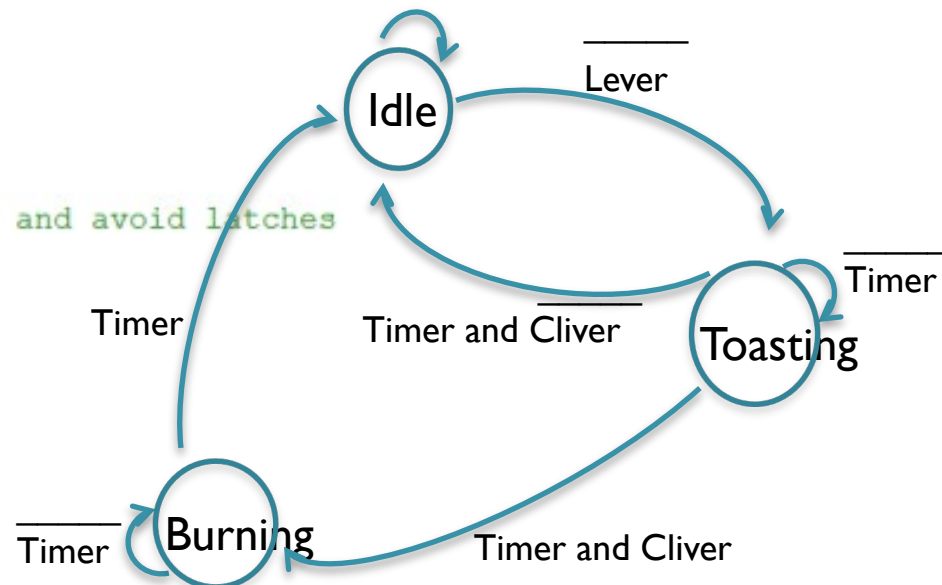


- The following process sets up the conditions to transition to the next state.
- The next state is determined by the current state and the inputs.
- This process follows the state transition diagram exactly.
- No other outputs belong here.

```

comb: PROCESS(current_state, lever, Cliver, timer)
BEGIN
  CASE (current_state) IS
    WHEN IDLE =>
      IF (LEVER = '1') THEN --remain off until lever is active
        next_state <= IDLE;
      ELSE --lever = 0
        next_state <= toasting;
      END IF;
    WHEN toasting =>
      IF (timer = '0') THEN
        next_state <= toasting;
      ELSIF (cliver = '1') THEN --don't have to check timer again
        next_state <= burning;
      ELSE --time is up and user is not cliver
        next_state <= IDLE;
      END IF;
    WHEN burning =>
      IF (timer = '0') THEN
        next_state <= burning;
      ELSE
        next_state <= IDLE;
      END IF;
    WHEN OTHERS => --keep the compiler happy and avoid latches
      next_state <= IDLE;
  END CASE;
END PROCESS;

```



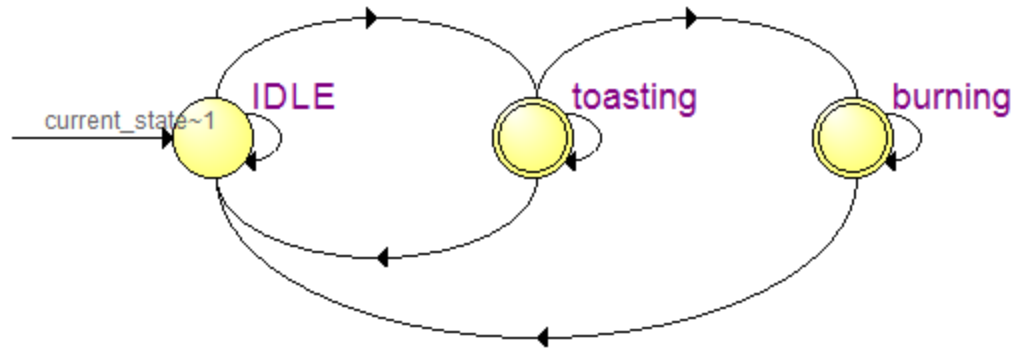
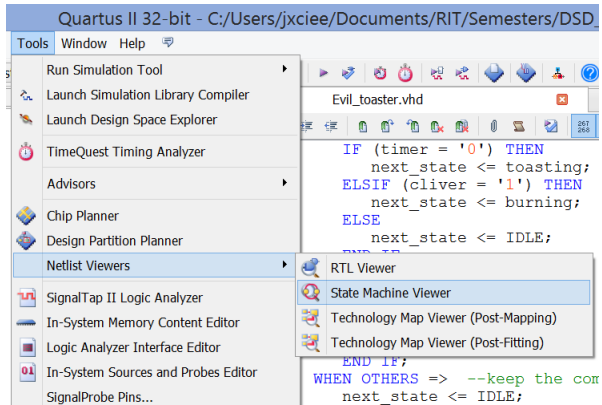


# Evil Toaster VHDL

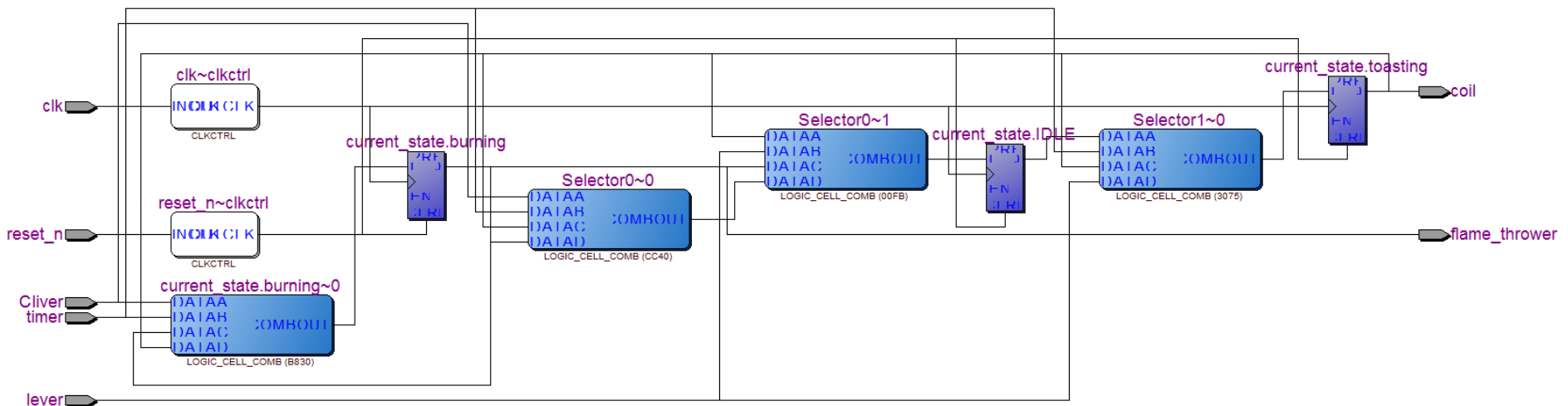
```
coil_out: PROCESS(current_state)
|--because this is a MOORE state machine, outputs depend on current state only
BEGIN
    CASE(current_state) IS
        WHEN toasting =>
            coil <= '1';
        WHEN OTHERS =>
            coil <= '0';
    END CASE;
END PROCESS;

flame_out: PROCESS(current_state)
    CASE(current_state) IS
        WHEN burning =>
            flame_thrower <= '1';
        WHEN OTHERS =>
            flame_thrower <= '0';
    END CASE;
END PROCESS;
```

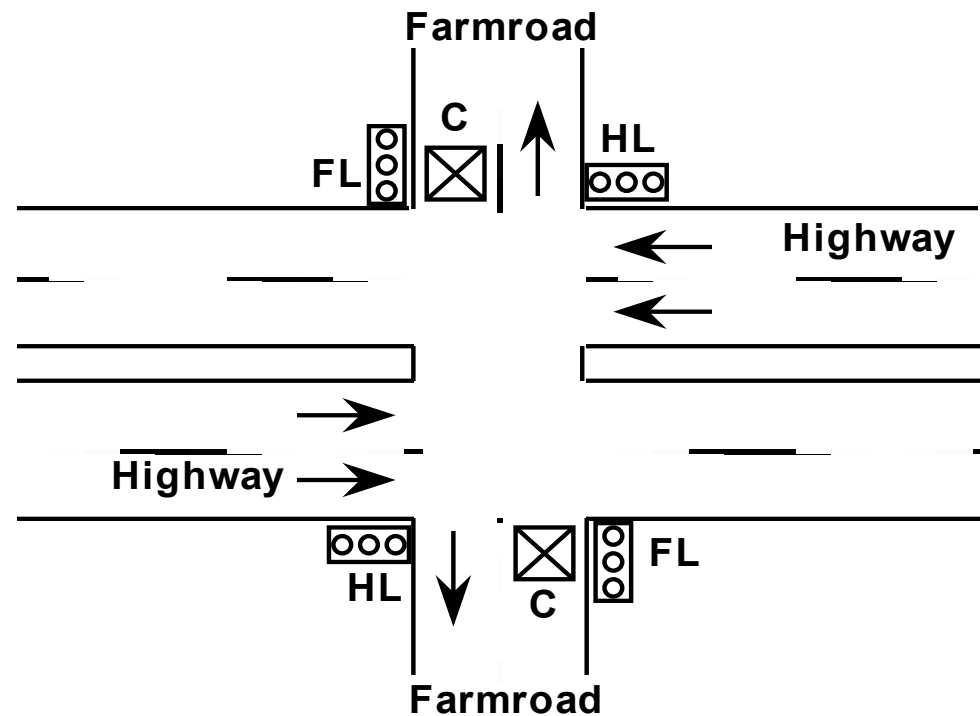
# After synthesis




# After Synthesis



# Class Activity





Consider the controller of a traffic light. This traffic light is at the intersection of a little used farm road and a busy highway. Detector C detects a car on the farm road. The light remains green for the highway until a car is detected on the farm road.

- Once a car is detected on the farm road, the highway light goes from green to yellow for 5 seconds and then from yellow to red.

- Both lights will be red for 5 seconds and then the farm road light will go green

The farm road light stays green until no more cars are detected, or 40 seconds has passed

- When the farm road light transitions, it goes to yellow for 5 seconds and then to red

- Both lights will be red for 5 seconds and then the highway light will go green

Once the highway goes green, it stays there at least 40 seconds, even if another car is detected on the farm road

# Determine the following

- Inputs
- States
- Outputs
- Create state transition diagram and output table