Lecture 12: State Machines

• Imagine writing the logic to control a traffic light
  – Every so often the light gets a signal to change
  – But change to what? It depends on what light is illuminated:
    • If GREEN, change to YELLOW
    • If YELLOW, change to RED
    • If RED, change to GREEN
  – We can think of RED, YELLOW, and GREEN as “states” of the traffic light
  – The only logic we need is a set of rules for transitioning between the states

• Logic that can be expressed in this way is called a “state machine”
• We can visualize the state machine as follows:
• Define a bit corresponding to RED, another for YELLOW, and a third for GREEN
  – then our state machine looks like
• In a logic circuit, the state is represented by the values stored in a set of flip-flops
  – with \( N \) flip-flops, there are \( 2^N \) possible states
  – …though we may not want to use all of the states
  – for example, the state with both RED and GREEN true is not useful for a traffic light!

• The circuit that controls a traffic light might look something like this:
• The traffic light requires only a few minutes of graphical programming
  – but other state machines demand more complex logic
• Fortunately, Altera has provided an easy way for us to design any state machine we want in AHDL
• A state machine is a variable type in AHDL, with name MACHINE OF BITS
A text design for the traffic light might look like:

```
SUBDESIGN traffic2
(
    change : INPUT;
    reset : INPUT;
    red, yellow, green : OUTPUT;
)

VARIABLE
    ss : MACHINE OF BITS (red, yellow, green)
    WITH STATES ( s0 = B"100", % red state %
                   s1 = B"001", % green state %
                   s2 = B"010"); % yellow state %

BEGIN
    ss.clk = change;
    ss.reset = reset;

    % define transition rules in table %
    TABLE
        ss => ss; % means this table contains the rules for ss %
        s0 => s1; % since s0 is listed first, that's the state
                   loaded in the machine after a reset %
        s1 => s2;
        s2 => s0;
    END TABLE;
END;
```
• Some traffic lights may have more possible states, such as:

• Adding the new states is a simple modification to the existing design
• But what if we only want the turn arrows during rush hour?
  – i.e., we want our state machine’s behavior to depend on external conditions
• The code to handle such a case is on the next slide:
SUBDESIGN traffic3
(
  change : INPUT;
  reset  : INPUT;
  rush_hour : INPUT;
  red. yellow. green. green_arrow. yellow_arrow : OUTPUT;
)

VARIABLE
  ss : MACHINE OF BITS (red. yellow. green. green_arrow. yellow_arrow)
    WITH STATES ( s0 = B"10000", % red state %
                    s1 = B"10010", % red with green arrow state %
                    s2 = B"10001", % red with yellow arrow state %
                    s3 = B"00100", % green state %
                    s4 = B"01000" ); % yellow state %

BEGIN
  ss.clk = change;
  ss.reset = reset;

  % define transition rules in table %
  TABLE
    ss, rush_hour := ss; % means that ss's behavior can depend on
    the value of rush_hour %
    s0, GND => s3; % not rush hour, so skip arrow states %
    s0, UCC => s1;
    s1, X  => s2; % do this always %
    s2, X  => s3;
    s3, X  => s4;
    s4, X  => s0;
  END TABLE;
END;
Toward more complex designs

- We already found with the graphical editor that it’s often easier to break a complex design into smaller units, which can be included as symbols in the .gdf file.
- We can do a similar thing with .tdf files:
  - From the File menu, select “Create default include file”
- Let’s say we’ve done this for the traffic light state machine, and want to make this part of a large program that controls traffic flow.
- The syntax would be:
INCLUDE "traffic3.inc";

SUBDESIGN trafficflow (

% whatever inputs/outputs... %
    reset        : INPUT;
    status       : OUTPUT;
)

VARIABLE
    stoplights[127..0] : traffic3;  % note that our subcircuit
                          is a variable in the big circuit %

BEGIN

    status = VCC;
    stoplights[].rush_hour = GND;
    stoplights[].reset = reset;
    stoplights[].change = GND;
    % and whatever other logic we want...%

END;
Pre-defined functions

• Altera provides a set of predefined include files for your convenience

• Among these are circuits that perform arithmetic operations on binary numbers
  – rather than designing different circuits for every possible choice of the number of input/output bits, these functions are define with *parameters* that allow the user to choose such values

• The following slides show an example of using one of the pre-defined functions to divide two numbers
INCLUDE "lpm_divide"; % Altera's built-in divider %

SUBDESIGN my_divide (  
    numerator[3..0] : INPUT;  
    denominator[1..0] : INPUT;  
    value[3..0] : OUTPUT;  
    remainder[1..0] : OUTPUT;  
  )

VARIABLE  
    divider : lpm_divide
    WITH % following line sets parameter values %
    (LPM_WIDTHN=4, LPM_WIDTHD=2);

BEGIN  
    divider.numer[] = numerator[];
    divider.denom[] = denominator[];
    value[] = divider.quotient[];
    remainder[] = divider.remain[];

END;
Another way to do it is to use lpm_divide “in-line”:

```
INCLUDE "lpm_divide"; % Altera's built-in divider%

SUBDESIGN my_divide2(

    numerator[3..0]   : INPUT;
    denominator[1..0] : INPUT;
    value[3..0]       : OUTPUT;
    remainder[1..0]   : OUTPUT;
)

BEGIN

    (value[], remainder[]) = lpm_divide (.numer[] = numerator[],
                                 .denom[] = denominator[])

    WITH (LPM_WIDTHN=4, LPM_WIDTHD=2)
    RETURNS (.quotient[], .remain[]);

END;
```
Debugging

- What to do if the circuit you programmed into the Altera chip isn’t behaving as it’s supposed to
  - Check the outputs on a scope or multimeter
  - Check the inputs
    - are they at the levels you expect?
  - Check the power and ground connections to the chip
  - Break the circuit up into subcircuits
    - test them one at a time until you find the problem
  - Add extra “test point” outputs at key places in the circuit
  - Search for symptoms of
    - clock glitches
    - setup and hold violations