Part II: Preparation for the DE2 lab

by

Professor Eric Salt

Department of Electrical Engineering
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

Created February 2011

```
Revised on:
October 6, 2016: revised introduction and added a
                 figure showing generic hardware
October 5, 2018: corrected a typo
October 18, 2020: major revisions
```

1 Finite State Machines

There are many ways to describe a finite state machine. One description could be:

A finite state machine is a sequential logic circuit whose input-output relationship depends on the state of the machine. The state of the machine is held in memory of some sort, usually a set of D-flip-flops. The machine sequences through different states in response to time dependent inputs and knowledge of the current state of the machine. No record of the state history is kept so only the current state can be used in the logic that computes the next state.

Finite State Machines (FSMs) are used in a wide variety of applications. The theory and operation of a FSM can, and will, be explained by working through an example where the work being done would be called parsing. The example is an electronic combination lock where the FSM has to make sure the correct combination is entered and then trip a lock.

Example

Consider a five keystroke electronic combination lock that trips a lock, i.e. allows a handle on a door to rotate. The door handle is normally pinned and not allowed to rotate. The electronic lock will momentarily pull the pin and allowed to door handle to rotate if buttons on a 7-button panel are pressed in the correct sequence. The "combination" to the lock is as follows:

1. The "clear" button must be the first button pressed.
2. The next three buttons pressed are numbers between 1 and 6 inclusive. These three numbers are the"combination" to the lock.
3. The fifth button pressed must also be the "clear" button.

After the final "clear" is pressed the pin will be removed momentarily and the door handle will be free to rotate for a moment.

If any entry in the sequence is incorrect the FSM is reset and sequence must restart from the beginning. For example, if the first four entries are correct and the fifth entry is not the "clear", then the FSM is reset.

The human interface to the electronic lock is the 7-button key pad shown in Figure 1. It is used to sequentially enter the combination of digits that open the lock.

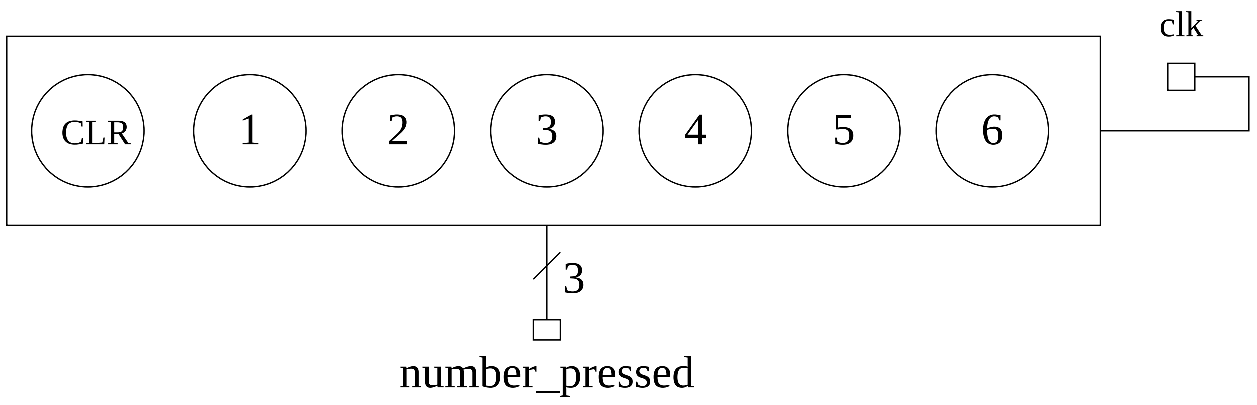


Figure 1: 7 button key pad

The key pad of Figure 1 is basically a 3-bit synchronous encoder that acts on the negative edge of the clock. It contains logic that generates a 3-bit pulse exactly one clock period in length starting and ending on a negative edge of the clock each time a button is pressed, no matter how long the button is held active. When the "CLR" button is pressed it generates a pulse with value 3'H0. The other buttons generate a 3-bit pulse that is an unsigned binary number equal to their face value. If no key is pressed then the output is a steady 3'H7. The operation of the key pad is illustrated in Figure 2

Operation of the FSM for the Electronic Lock

The user interface generates a de-bounced output one clock period long each time a button is pressed. The output is valid on the positive edge of clock. The finite state machine can only

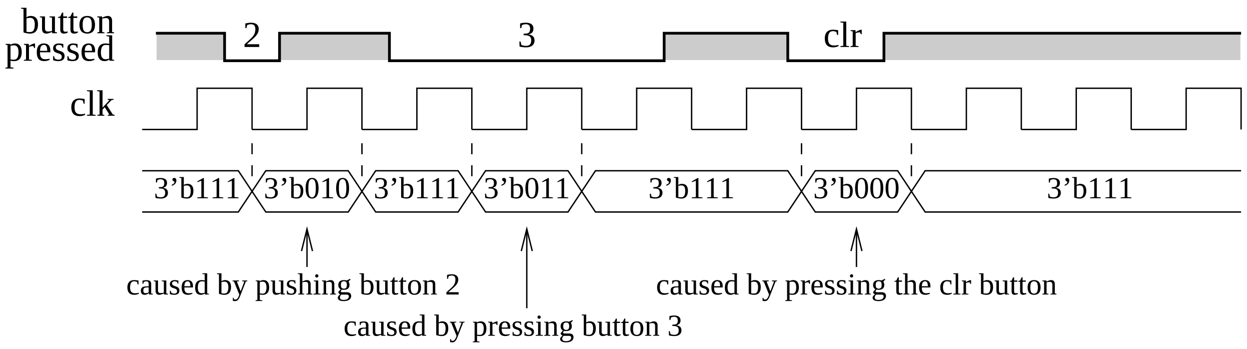

button pressed
2
3
clr
clk
3'b111
3'b010
3'b111
3'b011
3'b111
3'b000
3'b111
caused by pushing button 2
caused by pressing button 3
caused by pressing the clr button

Figure 2: Signal produced by the keypad in response to pressing buttons

change states on a positive edge of a clock.

The input/output diagram for the FSM part of the lock is shown in Figure 3. The finite

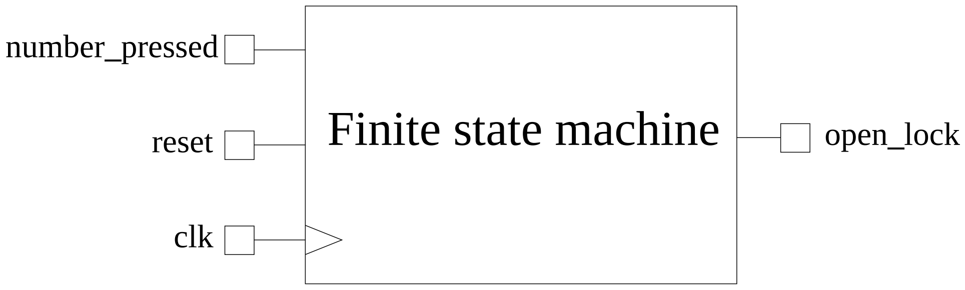


Figure 3: Finite state machine

state machine can only change states on a positive edge of a clock. The input from the keypad, which is `number_pressed`, is a pulse one clock period in length that is valid on the rising edge of the clock. The input named `reset` is an external reset.

The "state" of the machine is held in a set of flip flops. Each of the possible "states" corresponds to the flip flops having a particular/unique set of values. When one or more of the flip flops in the set changes value the machine is said to have changed states.

A diagram of circles and arrows called a state diagram is used to show how the machine moves from state to state. The state diagram for the electronic lock is shown in Figure 4. The states are represented with circles. Each of the circles represent a particular set of values for the flip flops. The arrows show how the inputs influence a change of state, i.e. a change in the values of the flip flops. The change in state happens on a positive edge of the clock. This means the machine spends all it time in one state or another and moves between states virtually instantaneously.

The rules that apply to state diagrams are:

1. With the exception of the "reset" arrow, the tail of each arrow must be connected to one and only one state and the head of each arrow must be connected to one and only one state. The exception is the tail of the reset arrow, which is not connected to anything. The rule for the head of the reset arrow is same as other arrows, it must be connected to

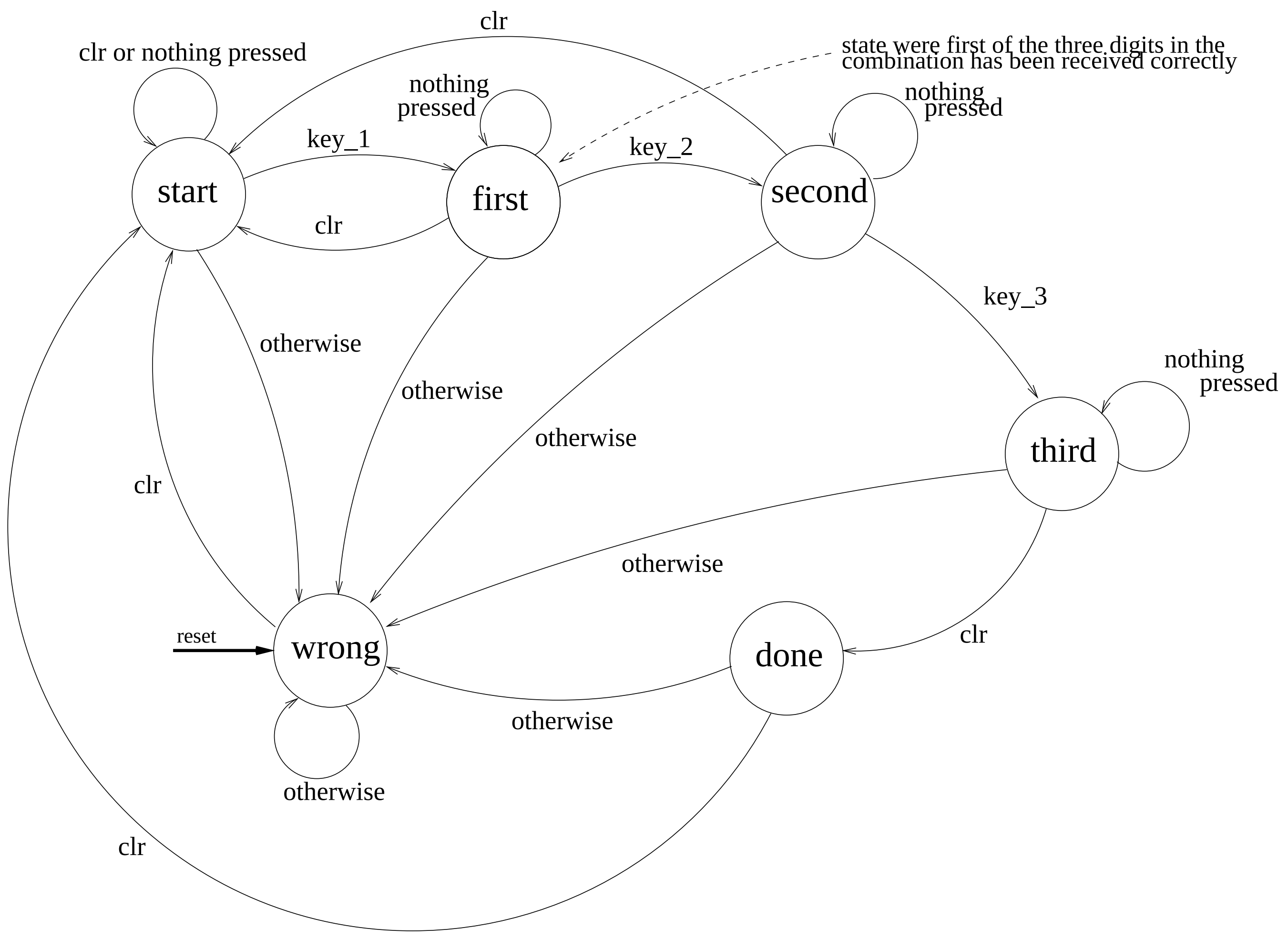


Figure 4: State diagram for the 3-digit electronic lock

one and only one state. The state the reset arrow is connected to is often referred to as the initial state.

2. With the exception of a reset arrow, an arrow is active if and only if the state connected to its tail is active and the conditions listed along side the arrow are true.

3. When the "reset" is activated, all other active arrows are ignored and the reset action is taken. The reset overrides all other active arrows. In normal steady state operation only one arrow is active at one time. If the reset arrow goes active, the one active arrow is ignored and the reset action taken.

   Noise or a power up can cause the machine to loose control. When out of control, the machine could have more than one active state and/or more than one active arrow. In this situation making "reset" active, causes the machine to take the reset action and enter the initial state..

The annotation used in the state diagram of Figure 4 is:

- clr, which is short for clear, is 3'H0. When used on the state diagram it means "number_pressed == 3'H0".
- key_1 is the first number in the combination. When used on the state diagram it means "number_pressed == key_1".
- key_2 is the second in the combination. When used on the state diagram it means "number_pressed == key_2".
- key_3 is the third number in the combination. When used on the state diagram it means "number_pressed == key_3".
- nothing_pressed is 3'H7. When used on the state diagram it means "number_pressed == 3'H7".
- Otherwise means "number_pressed == any value not specified on arrows leaving the state".

The operation of the electronic lock is illustrated by way of example using the state diagram in Figure 4. A sequence of events and the reaction by the FSM are listed below:

1. A external reset is applied. The FSM reacts by making the state named 'wrong' active.
2. The 'one', button on the keypad is pressed. The state diagram indicates 'clr' must be pressed to get out of state 'wrong'. Therefore, the reaction of the FSM is keep 'wrong' active.
3. The 'clr' button is pressed. The FSM reacts by making 'wrong' inactive and 'start' active. In other words the FSM changed states from 'wrong' to 'start'.
4. The first number in the combination is pressed, which is represented by 'key_1'. The FSM reacts by making state 'first' active.
5. A key other than the second digit of the combination is pressed. That is, some key is press and that key is not key_2. Since the only way to remain in state 'first' is to not to press any keys, pressing a key will cause the FSM to leave state 'first'. If the key pressed was 'key_2' then the FSR would react by going into state 'second' otherwise it reacts, as it does in this case, by going into state 'wrong'. (The state name 'wrong' is really short for 'wrong key pressed'.)
6. The following sequence of keys are pressed: clr, key_1, key_2, key_3, clr. The FSM reacts by changing from 'wrong' to 'start' to 'first' to 'second' to 'third' to 'done'.

   Note: The hardware is such that while 'done' is active the pin is removed and the door handle is free to rotate.
7. The FSM will remain in state 'done' for exactly 1 clock cycle. On the next clock cycle it will move to state 'wrong' unless the 'clr' button is pressed very quickly, in which case it will move to state 'start'.

Design and Implementation of the Electronic Lock

First decide how to represent the states. There are 6 states.

1. Could assign each state a binary number and store it in a 3-bit register (binary encoded states). The register would logically be called "current_state". A state mapping could be shown in Table 1.

Table 1: State mapping example

binary code	state name
3'H0	start
3'H1	first
3'H2	second
3'H3	third
3'H4	done
3'H5	wrong

2. Could use 1 flip/flop to hold the value of a state. This would require 6 flip/flops, 1 for each state. Only 1 flip/flop could be active at one time. This type of state coding is called one-hot coding as only 1 flip/flop can be "a one" at any one time. A flip/flop is said to be "hot" if its output is 1'b1. This method is preferred in industry as it is easier to design, easier to debug and easier to partition a design so several engineers can work on it at once. We will look at a one-hot design first.

One-hot design of the electronic lock FSM

The nature of one-hot design requires a *always @ (posedge clk)* procedure for each state. A circuit like the generic circuit shown in Figure 5 must be constructed for each state. For reasons

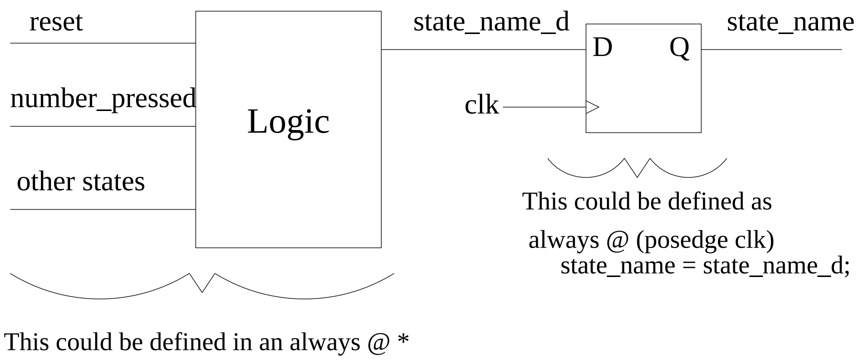


Figure 5: State holding Flip/flop and its control logic

that will be explained later it is best to implement this circuit with two *always* procedures: an *always @ ** procedure and an *always @ (posedge clk)* procedure.

The logic that controls the state flip/flop is given in the state diagram shown in Figure 4. It is simple. If an arrow entering the state is active or the arrow indicating “stay in the same state” is active, the flip/flop for that state is set to 1’b1. Otherwise the flip/flop is set to 1’b0.

The information needed to design the circuit that controls the “second” state flip/flop is extracted from the state diagram of Figure 4 and shown in Figure 6. Notice that there are only

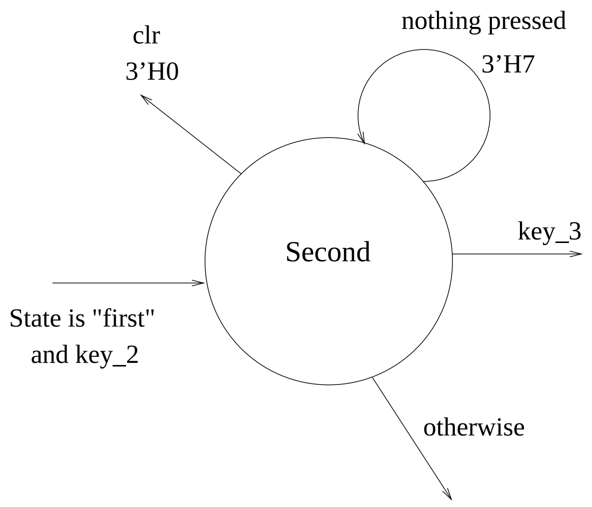


Figure 6: State called “second” of electronic lock FSM.

two cases where the “second” flip/flop will be a 1’b1.

The Verilog description that builds the flip/flop and its controlling logic for the “second” state follows:

```
always @ (posedge clk)
        second <= second_d; // build flip/flop

always @ *
   if (reset == 1'b1)            // start building the logic
        second_d <= 1'b0;
   else if ((second == 1'b1) && (number_pressed == 3'H7))
        second_d <= 1'b1;
   else if ((first == 1'b1) && (number_pressed == key_2))
        second_d <= 1'b1;
   else
        second_d <= 1'b0;
```

1.1 Generic Implementation of One-Hot coded FSMs

This section interrupts the electronic lock example to discuss one-hot coding in more generic terms. It is not needed to proceed to the next section, which continues the electronic lock example.

A one-hot FSM can be implement with a Verilog description of the logic that controls the flip/flop by only checking the arrow heads touching the state represented by the flip/flop. That being the case, this type of HDL could be referred to as a "target" based style or a "target" based architecture.

The generic Verilog HDL description of a one-hot coded state is

```
always @ (posedge clk)
        state_considered <= state_considered_d;

always @ *
   if (reset == 1'b1)
        state_considered_d <= initial_value;
   else if (arrow tip #1 active)
        state_considered_d <= 1'b1;
   else if (arrow tip #2 active)
        state_considered_d <= 1'b1;
             .
             .
             .
   else
        state_considered_d <= 1'b0;
```

It is pointed out that arrows that both start and end on the same state, as shown in Figure 7, indicate the conditions for staying in the that state. These loop back arrows must be considered

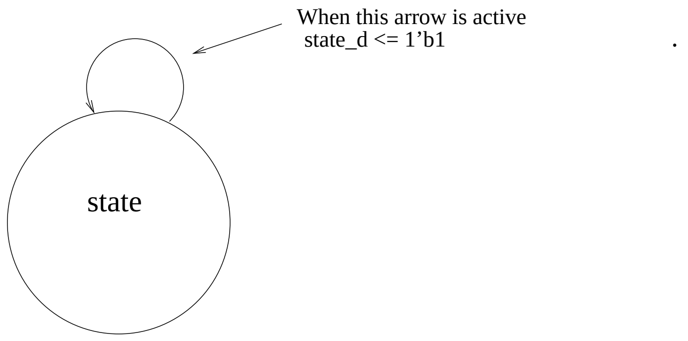


Figure 7: Loop back arrows are used to indicate when a FSM stays in a hot state, i.e. stays active

in the "if" or "else if" parts of the logic (i.e not in the final "else" part).

1.2 Electronic Lock Exercises

In class Electronic Lock exercise No. 1

Now back to the electronic lock example.

Write the Verilog code for the one-hot implementation of state “third”. The third state and the arrows that enter and exit that state have been extracted from the state diagram of Figure 4 and shown in Figure 8.

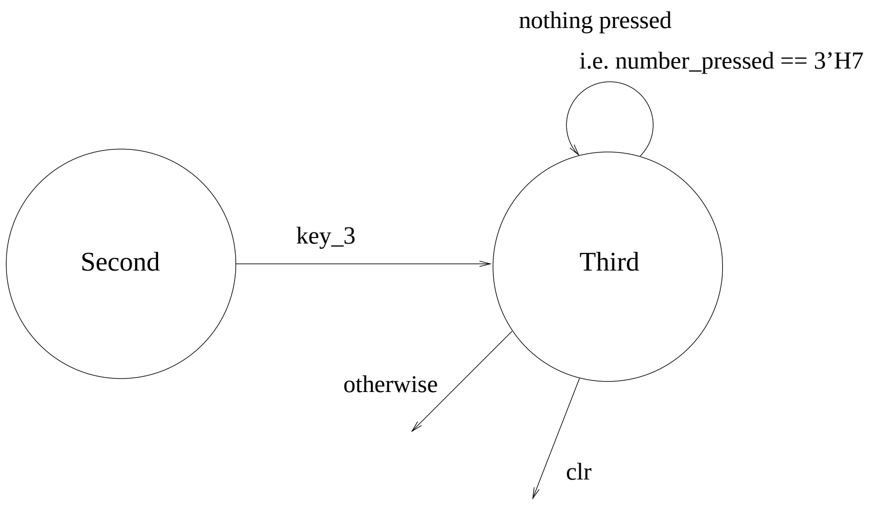


Figure 8: State “third” in electronic lock example

Solution

```
always @ (posedge clk)
        third <= third_d;

always @ *
   if (reset == 1'b1)
        third_d <= 1'b0;
   else if ((third==1'b1) && (number_pressed==3'H7))
        third_d <= 1'b1;
   else if ((second==1'b1) && (number_pressed==key_3))
        third_d <= 1'b1;
   else
        third_d <= 1'b0;
```

In class Electronic lock exercise No. 2

Design state "first" using a one-hot machine. The relevant information is given in Figure 9.

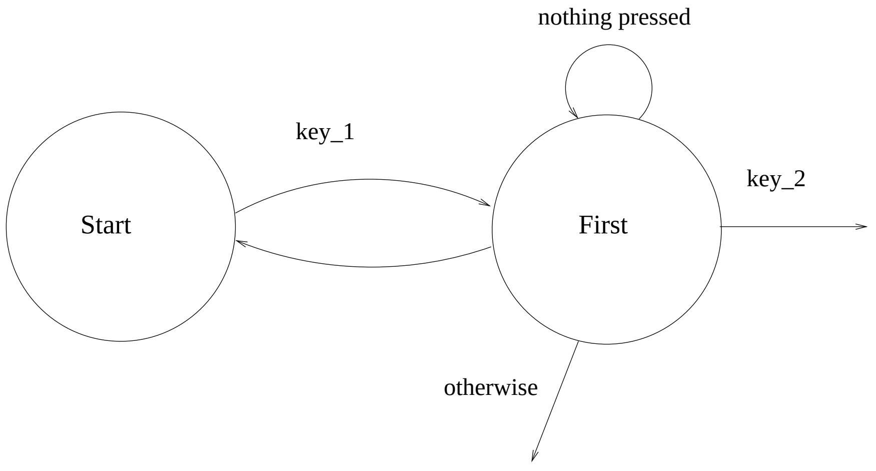


Figure 9: State "first" in the electronic lock example

The solution to this exercise is not given.

This is the end of the one-hot FSM design for the electronic lock.

1.3 Binary encoded states in FSMs

The other system of coding states in FSMs uses binary numbers held a set of flip/flops to represent the states. FSMs where each state is represented by a binary number are referred to as binary encoded (or binary coded) finite state machines. The active state, which is called the current state, is stored in a register often referred to as "state" or "active_state" or "current_state".

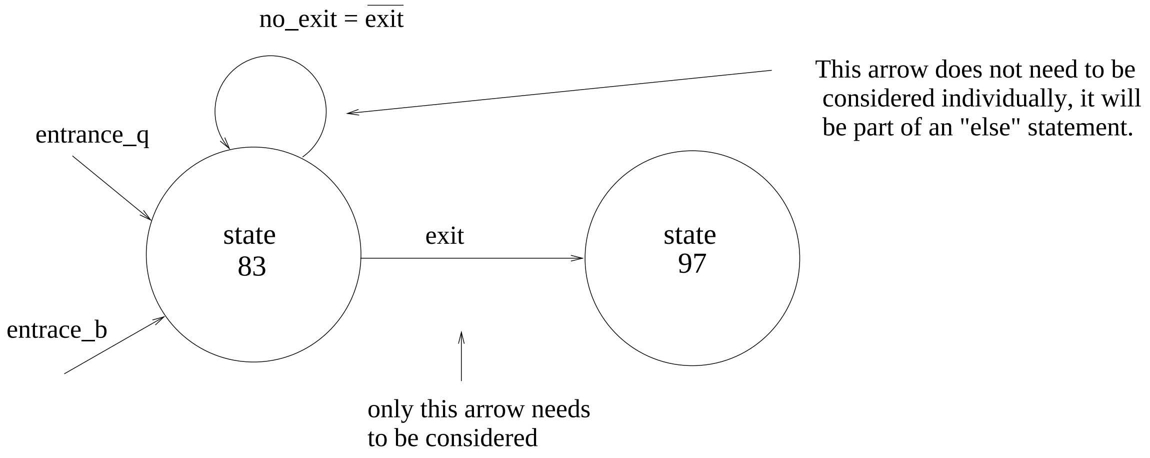
The logic for such machines are usually organized to determine the next state using the current state and the inputs. These are built as follows:

```
always @ (posedge clk)
        state <= state_d;

always @ *
   if (reset == 1'b1)
        state_d <= INITIAL_STATE;
   else
        case (state)
        state_0: state 0 logic
        state_1: state 1 logic
              .
              .
              .
        endcase
```

Binary-coded machines that are implemented with a case statement use the "current state" for the selection variable as is done in the Verilog HDL above. The Verilog HDL for the case alternatives determines the next state based on the inputs to the FSM.

Architectures that compute the next state given a current state usually, but not always, check the conditions on arrows that leave the "current state" in the "if" and "else if" parts of the "if, else if, else" logic and implement all the arrows pointing to the current state with the final "else". This is illustrated in Figure 10.



no_exit = $\overline{\text{exit}}$
This arrow does not need to be considered individually, it will be part of an "else" statement.
entrance_q
entrace_b
state 83
exit
state 97
only this arrow needs to be considered

Figure 10: Illustration of "if" and "else" clause for binary coded state

Electronic lock FSM design with binary coded states

The electronic lock example is now designed using binary coded states and next state logic based on the current state. The binary numbers used for the states are given in Table 2. The

Table 2: Representation of states with binary numbers

binary code	state
3’b000	start
3’b001	first
3’b010	second
3’b011	third
3’b100	done
3’b101	wrong

variable holding the number for the current state will be referred to as “current_state”. It may be difficult to remember the numbers assigned to a state, especially in designs with a large number of states. To avoid continually looking up numbers, the state numbers can be assigned to names using a compiler directive called `localparam`. The following Verilog HDL statements defines the association between words and the number for the state.

```
localparam START = 3'h0,
           FIRST = 3'h1,
           SECOND = 3'h2,
           THIRD = 3'h3,
           DONE = 3'h4,
           WRONG = 3'h5;
```

There are other compiler directives that can be used to associate numbers with words. The *‘define* directive could have been used instead of *localparam*, but most prefer the *localparam* directive for FSMs. There is actually one more directive that can be used. It is *parameter*. Examples of the three Verilog compiler directives that can be used to associate a word with a Verilog constant are:

```
localparam DONE = 3'h4;
parameter DONE = 3'h4;
`define DONE 3'h4
```

The *localparam* and *parameters* directives can only be used to define constants, i.e. binary numbers. The *‘define* directive instructs the compiler to make a literal replacement in the Verilog description. The syntax for the *‘define* compiler directive is
‘define “one word descriptor field”   “words to replace the word in the descriptor field”
Whenever a ‘ followed by the word in the “descriptor field” is encountered in the Verilog description it is replaced with the words in the “words to replace the word in the descriptor field”, which is all characters between the one word descriptor and the line ending enter. This means there must not be a semicolon at the end of the *‘define* directive. If a semicolon is placed after 3’h4 in the *‘define* statement, ‘DONE will be replaced with “3’h4;” and the semicolon will almost certainly cause a syntax error.

To make what was said above absolutely clear. *localparam* and *parameters* can only be used to make the character string, in this case, "DONE" equivalent to a number. The \`*define* directive replaces all occurrences of "\`DONE" with the character string that starts to the immediate right of DONE (in the \`define DONE ··· line) and ends with the character just before the line-ending "enter".

As an example, if the one-line statement
\`*define* NO_NUMBER_PRESSED number_pressed == 3'h7
is in the Verilog HDL, then all occurrences of
"\`NO_NUMBER_PRESSED" are replaced with
"number_pressed == 3'h7". Clearly, the \`*define* is more flexible and more powerful.

The directives *localparam* and *parameters* are identical, except when a number is associated with a name, i.e. a word, using *parameter* in a module that number can be changed (i.e. over ridden) when the module is instantiated using the "defparam" command.

Most choose to associate state names with numbers using *localparam* directive.

Helpful convention for literals

To make Verilog HDL descriptions more easily understood, designers often do something to distinguish the names they have associated with constants (or the literals), i.e. the names defined using \`*define* or *localparam* or *parameter*. One way to do this is to write all such names entirely with capital letters. Adopting a convention like this makes is obvious which of the variables are constants (or macros) defined by one of \`*define*, *localparam* or *parameters*.

Conditional next state implementation of electronic lock FSM

The electronic lock can be easily implemented using logic that determines the next state given the current state. Like in the one-hot implementation, the d-flip/flops holding the state are built separately from the logic that determines the next state. The Verilog HDL is as follows:

```
always @ (posedge clk)
        current_state <= current_state_d;

always @ *
if (reset == 1'b1)
        current_state_d <= WRONG;
else
   case (current_state)
        START:
              if ((button_pressed == 3'b7) || (button_pressed == 3'b0)
              current_state_d <= START;
```

```
                else if (button_pressed == key_1)
                current_state_d <= FIRST;
                else                current_state_d <= WRONG;
            FIRST: Verilog description
                .
                .
                .
            WRONG: Verilog description
            default: current_state_d <= WRONG;
        endcase
```

Binary encoded FSMs can also be implemented with a "target state" architecture. In this architecture the arrow tips are considered in the "if" and "else if" parts of the "if : else if : else" logic. Certainly "current_state_d" must be built in one *always* procedure, but in the "target state" architecture there is very large "if : else if : else" structure where a different arrow head is considered in different "else if" clauses. For example:

```
    always @ *
        if (reset == 1'b1)
            current_state_d <= WRONG;
        else if (one of the arrows tips connected to "start" is active)
            current_state_d <= START;
        else if (one of the arrows tips connected to "first" is active)
            current_state_d <= FIRST;
            .
            .
            .
        else
            current_state_d <= WRONG;
```

In my opinion, the "conditional next state" architecture is better than the "target state" architecture for binary encoded FSMs.

NOTE: In one-hot encoding each flip flop is built in a separate always statement (actually a pair of always statements, one to build the f/f, the other to make the logic). The "conditional next state" architecture can not distribute the logic for the FSM across several always procedures. The rules of Verilog prohibit defining a single variable in multiple always procedures. Since the "conditional next state" architecture holds all the state information in one variable, it must be defined in one always procedure.

This is the end of the example on binary coded FSM design of an electronic lock

1.4 Generating Signals on State Transitions

Sometimes circuits external to the FSM need to know when the FSM is about to transition from one state to another. For example, it may be necessary to reset a timer each time the FSM changes state.

There are two types of transition actions: an entrance action and an exit action. For example if a FSM resets a counter when state "I_am_mad" is entered, then the counter is reset with an entrance action. The counter is reset with the clock edge that activates state "I_am_mad".

Entrance and exit actions are easily generated if the state flip/flops and next state logic are implemented in different always statements.

In one-hot coded machines entrance, exit and state_change actions are generated as follows.

```
always @ (posedge clk)
   I_am_mad <= I_am_mad_d;

always @ *
   if-else logic to generate I_am_mad_d

always @ *
   if ((I_am_mad==1'b0) & & (I_am_mad_d==1'b1))
         entering_I_am_mad <= 1'b1;
   else
         entering_I_am_mad <= 1'b0;

always @ *
   if ((I_am_mad==1'b1) & & (I_am_mad_d==1'b0))
         exiting_I_am_mad <= 1'b1;
   else
         exiting_I_am_mad <= 1'b0;

always @ *
   changing_I_am_mad <= I_am_mad ^ I_am_mad_d;
```

1.5 Incorporating Time-Outs in the Electronic Lock

The functionality of the Electronic Lock is now modified to include a time out feature, where after entering either 'first', 'second' or 'third' the FSM will automatically reset if the correct number for the combination is not pressed within a predetermined 'time out' time. The 'time out' times are measured in clock periods.

There are several ways to the implement the new feature. Two implementations will be explored with the second being a refinement of the first.

Figure 11 shows the block diagram of the hardware (bottom of the page) and the associated state diagram (top of the page) for the first implementation of an FSM with a time out feature. The hardware external to the FSM generates 3 binary signals that are inputs to the FSM. The hardware consists of a timer, which could just as well have been called a counter, and some logic that generates the synchronous clear for the timer. The logic block issues a clear pulse one clock period wide each time the FSM is about to change states.

The timer counts up from 0 until it is cleared. The timer will be cleared when a button is pressed that causes the FSM to change states. If no button is pressed for a long time, the timer will reach its maximum value and trap there (i.e. the timer will not roll over) until a button is pushed that causes the FSM to change state and the logic block to issue a clear. While counting up the timer may reach the time-out time for state 'first'. If it does, it will assert 'time_1' for the one clock period while the timer is equal to the time-out time. The timer will similarly assert 'time_2' and 'time_3' if it reaches the time-out times for state 'second' and 'third', respectively.

The state diagram indicates that if the FSM is in state 'first' when 'time_1' is asserted, it will change to state 'wrong' on the next positive edge of the clock. Of course this change of state will cause the timer to reset. If the FSM is in state 'second' or ' third' it will exit to state 'wrong' if 'time_2' or 'time_3', respectively, is asserted.

Note: By rule no two arrows in state diagram can be active at once. Therefore, the conditions that activate the loop back arrows in states 'first', 'second' or 'third' have to be changed to deactivate the loop back arrows when time_1, time_2 and time_3, respectively, are active.

The FSM can be simplified a little by making some modifications to the timer. The modified timer is shown at the bottom of Figure 12. In this design the timer generates only one binary signal for the FSM regardless of the number of states that can time-out. In this circuit the timer is not synchronously cleared, but is synchronously preset with the time-out time for the state that is about to be entered. The timer counts down until a button is pressed that causes the FSM to change states. At which time the logic block detects the state change and signals the timer telling it what state is about to be entered.

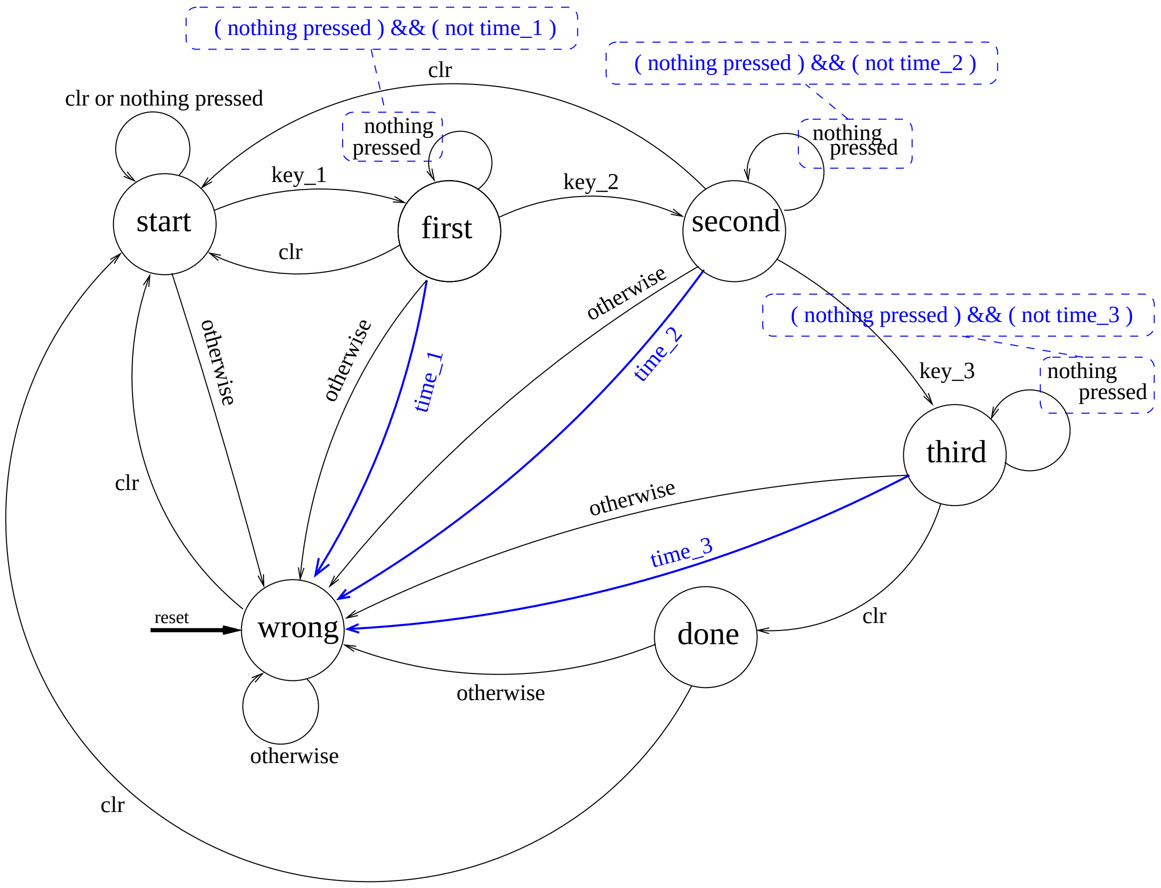
If the timer counts down to 1 before another state it entered, then it traps at one until another state is entered. If the state being entered does not have a time-out time, for example the 'wrong' state, the timer is preset to 1 and continues to trap there.

The 'time_is_up' signal is asserted while the timer is trapped at 1.

For this circuit to work the output of the logic circuit has to indicate the state that is about to be entered. The timer knows the time-out times for all states that can time out, therefore it can preset its time to the time-out time for the state that is about to be entered.

A Verliog HDL for the timer circuit is:

```
always @ (posedge clk)
   if (entering_first)
         timer <= TIME_OUT_FOR_FIRST;
   else if (entering_second)
         timer <= TIME_OUT_FOR_SECOND;
         .
         .
         .
   else if (timer == 8'b1)
         timer <= 8'b1;
```



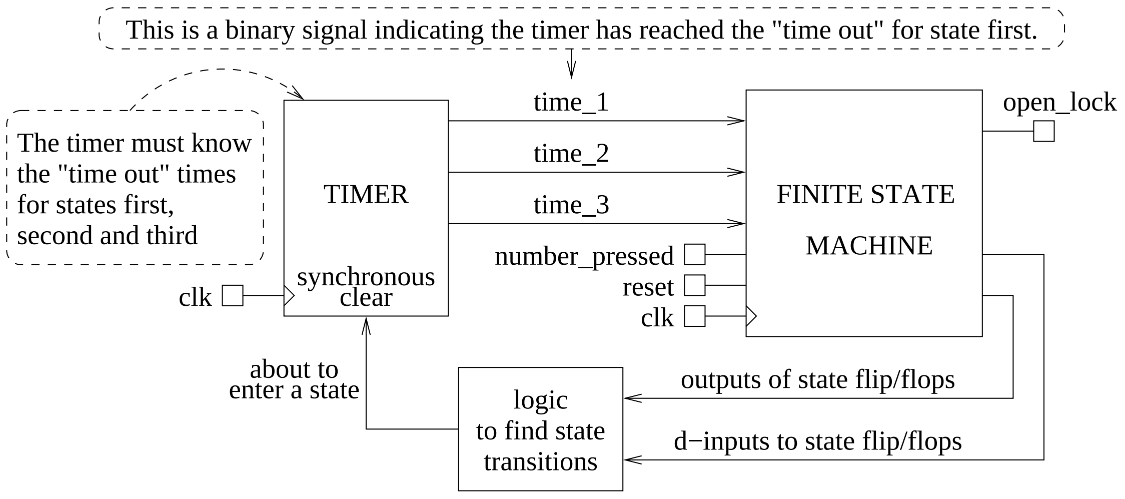

This is a binary signal indicating the timer has reached the "time out" for state first.
The timer must know the "time out" times for states first, second and third
TIMER
synchronous clear
clk
time_1
time_2
time_3
FINITE STATE MACHINE
open_lock
number_pressed
reset
clk
about to enter a state
logic to find state transitions
outputs of state flip/flops
d−inputs to state flip/flops

Figure 11: Hardware Block Diagram and State Diagram for an FSM with a Time Out feature

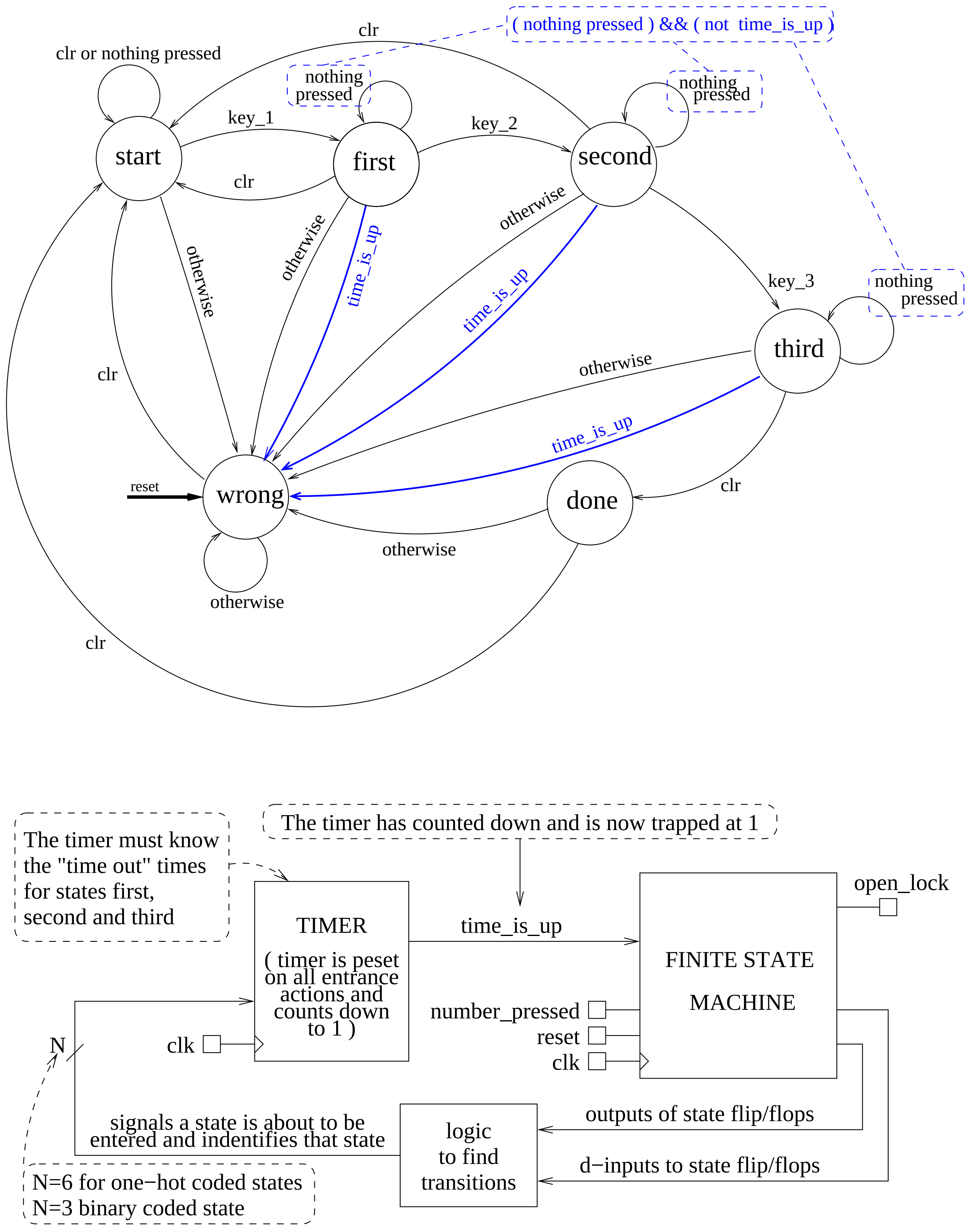

(nothing pressed) && (not time_is_up)
clr or nothing pressed
clr
nothing pressed
start
key_1
first
key_2
second
nothing pressed
clr
otherwise
otherwise
time_is_up
otherwise
time_is_up
key_3
nothing pressed
third
clr
otherwise
time_is_up
reset
wrong
done
clr
otherwise
otherwise
clr
The timer must know the "time out" times for states first, second and third
The timer has counted down and is now trapped at 1
TIMER
(timer is peset on all entrance actions and counts down to 1)
time_is_up
FINITE STATE MACHINE
open_lock
number_pressed
reset
clk
clk
N
signals a state is about to be entered and indentifies that state
logic to find transitions
outputs of state flip/flops
d−inputs to state flip/flops
N=6 for one−hot coded states
N=3 binary coded state

Figure 12: A Second Hardware Block Diagram and State Diagram for an FSM with a Time Out feature

```
    else
            timer <= timer - 8'b1;
```

```
The "time_is_up" circuit is
always @ *
    if (timer == 8'b1)
            time_is_up <= 1'b1;
    else
            time_is_up <= 1'b0;
```

Entrance and exit actions can be constructed without using a variable called "entering_state". This is done by building the logic that defined "entering_state" into the "if" statement in procedure where it is used. For example:

```
always @ (posedge clk)
    if ((first == 1'b0) & & (first_d == 1'b1))
            timer <= TIME_OUT_FOR_FIRST;
    else if ((second == 1'b0) & & (second_d == 1'b1))
            timer <= TIME_OUT_FOR_SECOND;
            .
            .
            .
```

Building the logic for an entrance or exit action into the "if condition" has its advantages and disadvantages. The advantage is that clutter associated with building a logic circuit for each "entering_state" signal is avoided. This is not a significant advantage. The disadvantage is the Verilog description is much more difficult to understand. This is a significant disadvantage. This is a situation where the compiler directive `*define* can be used to completely remove the disadvantage.

The `*define* directive sets up a literal substitution. Such definitions follow:

```
`define ENTERING_FIRST (first == 1'b0) && (first_d == 1'b1)
`define ENTERING_SECOND (second == 1'b0) && (second_d == 1'b1)
```

The timer can be built using these definitions as follows:

```
always @ (posedge clk)
    if (`ENTERING_FIRST)
            // the leading back tick is needed for words defined with `define
            timer <= TIME_OUT_FOR_FIRST;
    else if (`ENTERING_SECOND)
            timer <= TIME_OUT_FOR_SECOND;
            .
            .
```

.

Every time the compiler encounters " `ENTERING_FIRST " it substitutes "(first == 1'b0) && (first_d == 1'b1) ".

Using the `*define* directive makes the hardware description much easier to read.

An example from Section II of the Lab

The following example constructs the "state 1" flip/flop of the traffic controller that is the subject of Section II of the Lab.

This flip/flop is built in the same way as the previous example, but the check for arrow heads being active is done in two different *always @ ** procedures and then a third *always @ ** procedure combines them to make the signal for the d input. Dividing the logic for the d input into three *always @ ** procedures provides a second way to easily extract entrance and exit actions. The logic for "entering_state_1" is put in one *always @ ** procedure and the logic for "staying_in_state_1" is put in the second *always @ ** procedure and the logic for the d input of the state_1 flip/flop, which is simply
("entering_state_1" | "staying_in_state_1") ,
is put into the third.

This is illustrated below by using the state 1 flip/flop in the one-hot FSM design of the traffic light controller.

```
// building the flip flop
always @ (posedge clk or posedge reset)
   if (reset == 1'b1)
         state_1 <= 1'b1;
   else
         state_1 <= state_1_d;

// logic for entering state 1
always @ *
   if ((state_6 == 1'b1) && (time_out == 1'b1) && (walk_request == 1'b0))
         entering_state_1 <= 1'b1;
   else
         entering_state_1 <= 1'b0;

// logic for staying in state 1
always @ *
   if ((state_1 == 1'b1) && (not_time_out == 1'b1))
         //stay in state 1 on next posedge clk
         staying_state_1 <= 1'b1;
   else
```

```
            staying_state_1 <= 1'b0;

   // logic for the d-input for the state_1 flip/flop
   always @ *
      if (entering_state_1 == 1'b1)
            state_1_d <= 1'b1;
      else if (staying_in_state_1 == 1'b1)
            state_1_d <= 1'b1;
      else // not in state 1 on next posedge clk
            state_1_d <= 1'b0;
```

1.6 Entrance and Exit actions for binary coded FSMs

For binary coded FSMs entrance and exit actions are generated as follows:

```
   always @ *
      if (current_state_d != current_state)
            entering_state <= current_state_d;
      else
            entering_state <= 8'hff;
```

where entering_state holds the number of state that will be entered on the next positive edge of the clock and 8'hff is a predetermined constant (that can not equal to any of the states) that indicates the FSM is not going change states on the next positive edge of the clock.

```
   always @ *
      if (current_state_d != current_state)
            exiting_state <= current_state;
      else
            exiting_state <= 8'hff;// 8'hff can be any unique number.
```

The entrance actions can be used to reset a timer in a binary coded FSM is as follows:

```
   always @ (posedge clk)
      if (entering_state == FIRST)
            timer <= TIME_OUT_FOR_FIRST;
      else if (entering_state == SECOND)
            timer <= TIME_OUT_FOR_SECOND;
            .
            .
            .
      else if (timer == 8'b1)
            timer <= 8'b1;
      else
```

```
        timer <= timer - 8'b1;
```

Warning: Do not use asynchronous loads in the construction of the timer for the Labs.

The timer circuit that is given on the next two pages shows what can happens when asynchronous loads are used to preset the timer. In the example below the timer has an asynchronous reset to demonstrate issues that can arise when an asynchronous reset is used. (**Do not use asynchronous set, clears or loads in your lab.**)

The compiler constructs this circuit in a very predictable way, but how the compiler builds the timer depends on whether or not the outputs of the timer are sent to output pins or used internally.

Suppose an 8-bit timer is asynchronously loaded with 8'd60 which is 8'b0111_1100. The circuit built by the compiler is shown in Figure 13.

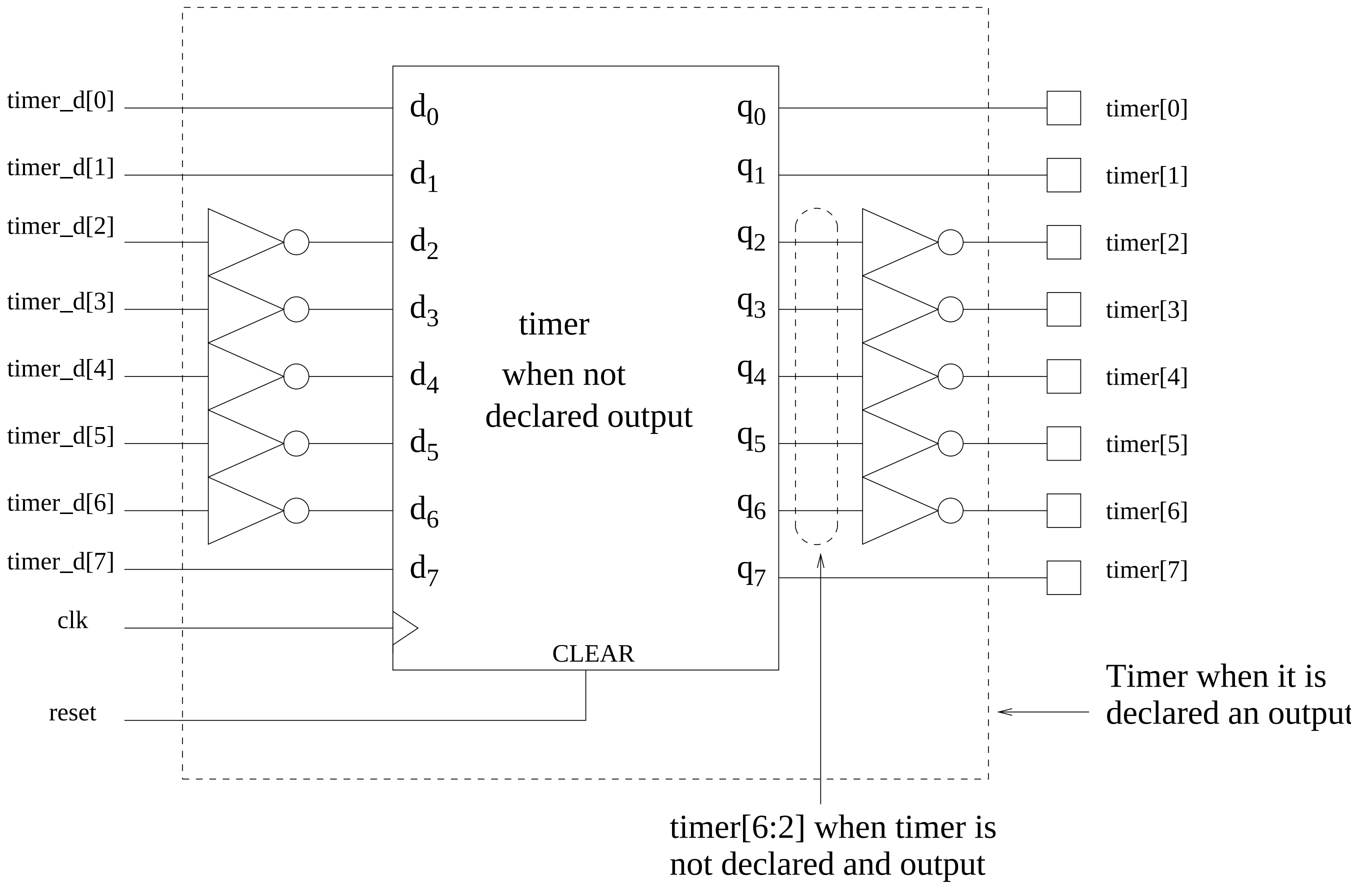

timer_d[0]
timer_d[1]
timer_d[2]
timer_d[3]
timer_d[4]
timer_d[5]
timer_d[6]
timer_d[7]
clk
reset
d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7
timer when not declared output
CLEAR
timer[0]
timer[1]
timer[2]
timer[3]
timer[4]
timer[5]
timer[6]
timer[7]
Timer when it is declared an output
timer[6:2] when timer is not declared and output

Figure 13: An 8-bit timer with an asynchronous load

NB: For FPGAs with only asynchronous clears on the flip/flops the asynchronous load of a constant (i.e. always loads the same value) is implemented with not push backs. Quartus 8.3

puts inverters in front of all flip/flops that are to be loaded with a 1'b1. and always clears the actual flip/flop. To convert the flip flop to asynchronous set the compiler should put an inverter on the output. It will only do this if the timer output is sent to an output pin. Otherwise, it handles the inversion in LUTs that follow the timer.

Confusion arises when the timer is not an output and is displayed in the simulator as a buried register. In this case the actual value of the flip/flops are displayed, which are different from what is expected. For example, if the constant 6'd60 is asynchronously loaded, the simulator display will show that it has loaded with 6'd0, because the output inverters are not there. Now suppose the timer is continually decremented after it is loaded, then the sequence produced is 6'H00, 6'H07, 6'H06, 6'H05, 6'H04, 6'H0B, 6'H0A. This can be very confusing if you are unaware of how the compiler implements asynchronous loads in FPGAs that contain flip flops with only asynchronous clears.

NB: everything is taken care of by the compiler - it just looks wrong while you are debugging.

End of asynchronous clear, set and loads warning

If the traffic light controller is designed using binary coded states, then timer could be constructed as follows:

```
always @ (posedge clk or posedge reset)
    if (reset == 1'b1)
        timer <= STATE_1_TIME;
    else if (entering_state_1 == 1'b1)
        timer <= STATE_1_TIME;
    else if (entering_state_1w == 1'b1)
        timer <= STATE_1W_TIME;
    else if (entering_state_1fd == 1'b1)
        timer <= STATE_1FD_TIME;
    else if (entering_state_1d == 1'b1)
        timer <= STATE_1D_TIME;
    else if (entering_state_2 == 1'b1)
        timer <= STATE_2_TIME;
    else if (entering_state_3 == 1'b1)
        timer <= STATE_3_TIME;
    else if (entering_state_4 == 1'b1)
        timer <= STATE_4_TIME;
    else if (entering_state_4a == 1'b1)
        timer <= STATE_4A_TIME;
    else if (entering_state_4w == 1'b1)
        timer <= STATE_4W_TIME;
    else if (entering_state_4fd == 1'b1)
        timer <= STATE_4FD_TIME;
    else if (entering_state_5 == 1'b1)
        timer <= STATE_5_TIME;
    else if (entering_state_6 == 1'b1)
        timer <= STATE_6_TIME;
    else if (timer == timed_out)
        timer <= timer;
    else
        timer <= timer - 6'd1;

always @ *
    if (timer == 6'd1)
        timed_out <= 1'b1;
    else
        timed_out <= 1'b0;

always @ *
    not_timed_out <=   timed_out;

// end building the timer
```

1.7 Comments on debugging

It would be wonderful if an engineer could write a Verilog description, compile it, remove the syntax errors, compile it again, load it into the FPGA and then give it to the customer. But that is not how it is done. An engineer must check to see if the product/device works properly.

The checking is usually broken into two parts. The first part, which is referred to as debugging, checks the basic functionality and the second part, referred to as verification and testing, checks to see that the detailed operation meets the specifications laid out by the customer under the full operating range of temperature, humidity etcetera. In this class the focus is on debugging although the students do have to check for proper operation, but do not have to worry about worrisome details like meeting timing, power consumption, electromagnetic emissions etc. Verification and testing is a very complicated process that must be approached scientifically. There is a subsequent class in the CME program dedicated to the science of verification and testing.

Debugging and testing have many similarities and the boundary between them is blurred. The main difference is the debugging process can be more intrusive since there are no concern for things like meeting timing until basic operation of the circuit is correct. This allows the engineer to add circuitry like `signal tap` to collect information, add circuitry to generate stimuli, bring in extra inputs, take out extra outputs and do other things that will help determine correct operation. For example, an internal signal may be well understood and if observed could help determine the proper operation of the circuit. If the compiler had optimized the signal away the engineer could modify the Verilog description to declare it an output. This would prevent the compiler from optimizing it out and allow it to be displayed by the simulator. Of course making the signal an output will change the way the compiler builds the circuit so this technique could not be used in verification and testing.

To debug the traffic light controller there is a useful circuit that should be added. The circuit generates a clock with a programmable frequency. The clock that drives the FSM can be changed to have a higher frequency during debugging. This will shorten the dwell times for each of the states and make it quicker to check response to the inputs e.g. the walk buttons.

The FSM machine for the traffic light controller is to be driven by a 1 Hz clock that is constructed by counting down a 27 MHz clock. For purposes of debugging it is convenient to increase the frequency of this clock. It will also reduce the time required by the examining instructor to verify the operation of the student designs. For the latter reason the students are requested to provide a debug option on the clock that changes the clock for the FSM from 1 Hz to 10 Hz.

A block digram of a circuit that will generate a clock switchable between 1 Hz and 10 Hz is given in Figure 14.

The Verilog description that makes a switchable 1 or 10 Hz clock from an input clock of frequency 27 MHz is given below:

```
reg [23:0] clock_divider_counter, clock_divider_constant;

always @ *
   if (debug == 1'd1)
         clock_divider_constant <= 24'd1_350_000; // 10 Hz
      // clock_divider_constant <= 24'd1; // uncomment to make 13.5 MHz clk
   else
         clock_divider_constant <= 24'd13_500_000; // 1 Hz

always @ (posedge clk_27)
   if (reset == 1'd1)
         clock_divider_counter <= 24'd1;
   else if (clock_divider_counter == 24'd1)
         clock_divider_counter <= clock_divider_constant;
   else
         clock_divider_counter <= clock_divider_counter - 24'd1;

always @ (posedge clk_27)
   if (clock_divider_counter == 24'd1)
         clk <= ~ clk;
   else
         clk <= clk;
```

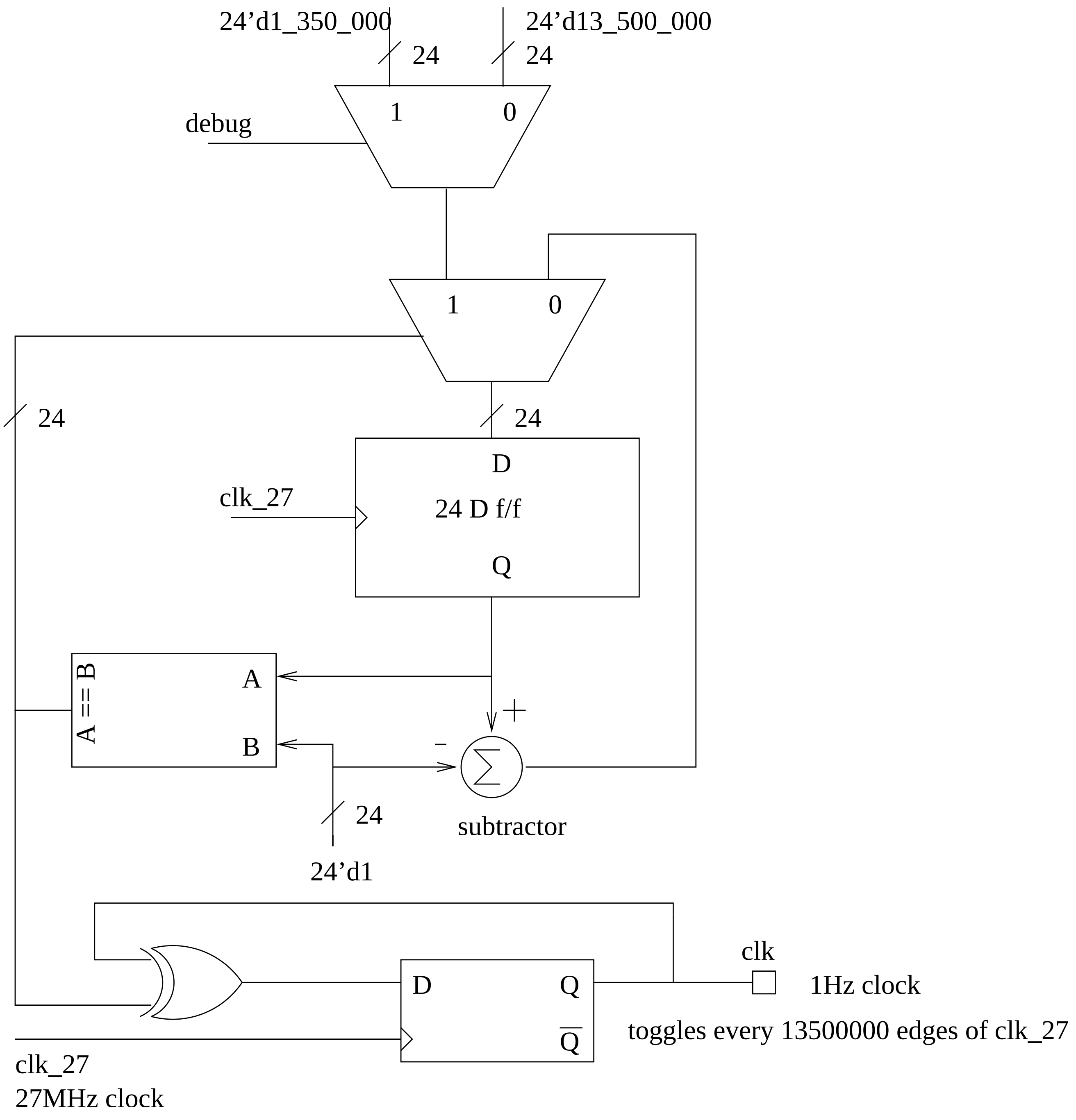


Figure 14: A switchable 1 Hz or 10 Hz clock constructed from a 27 MHz clock