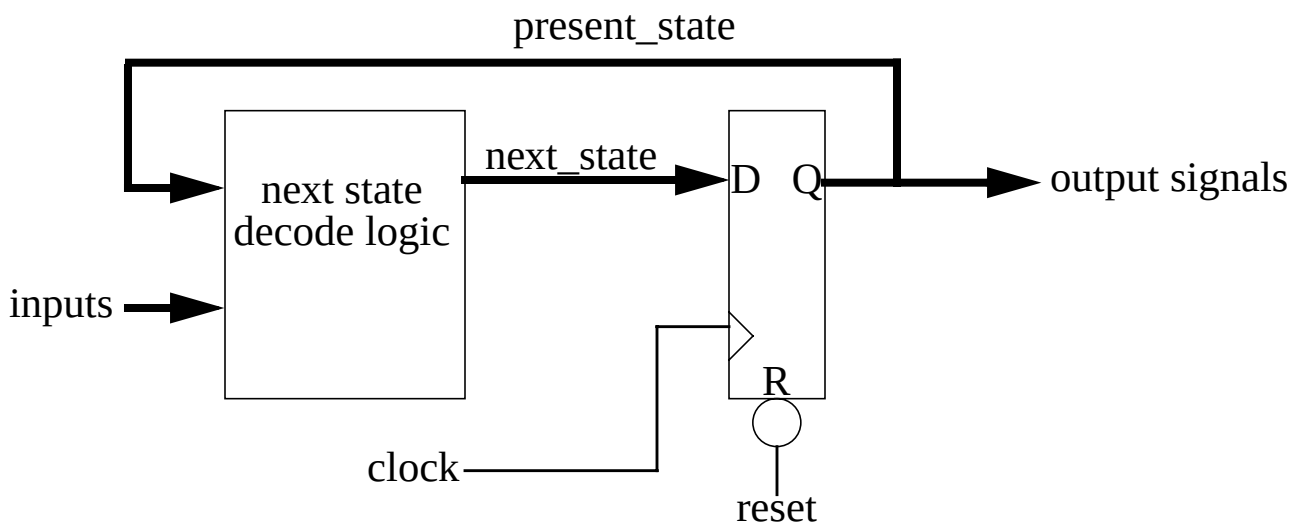
State Machines in VHDL

Implementing state machines in VHDL is fun and easy provided you stick to some fairly well established forms. These styles for state machine coding given here is not intended to be especially clever. They are intended to be portable, easily understandable, clean, and give consistent results with almost any synthesis tool.

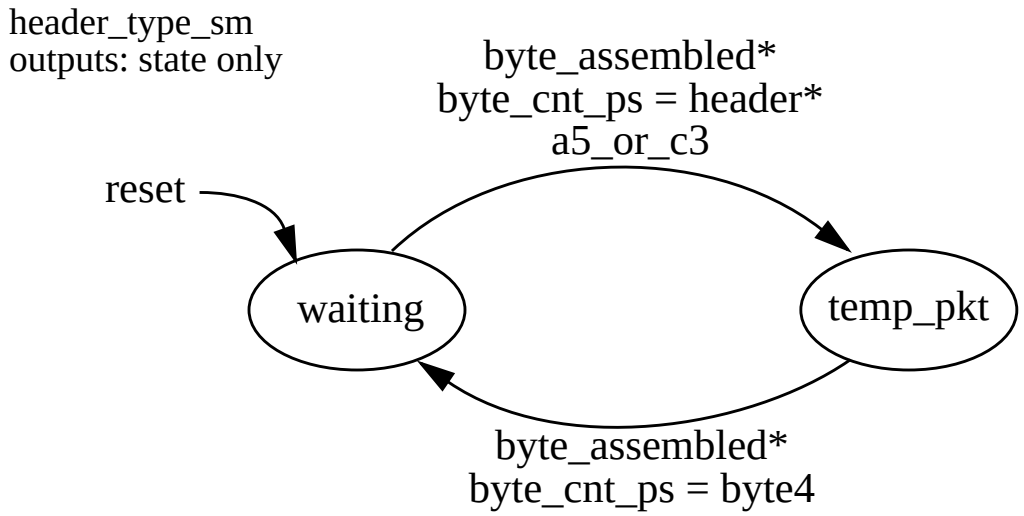
The format for coding state machines follows the general structure for a state machine. Lets look at the basic Moore machine structure.



The Moore state machine consists of two basic blocks, next state decode (or steering) logic, and some state storage usually (always for our case) D-type flip flops. Inputs are applied to the next state decode block along with the present state to create the next state output. The flip flops simply hold the value of the present state. In the example above, the only output signals are the outputs of the state flip flops. Alternatively, the flip flop outputs could be decoded to create the output signals.

For a first example we will look at the state machine in TAS which holds the state of what type of header is being received, *waiting* or *temp_pkt*. First we look at the state diagram.

State Diagram for header_type_sm



All your state machines should be documented in roughly this fashion. The name of the process holding the code for the state machine is the name of the state machine. In this case it is *header_type_sm.*

Every state machine has an arc from “reset”. This indicates what state the state machine goes to when a reset is applied. The diagram is worthless without knowing what the initial state is.

Each state in this example is given a name. In this case we are using a type for the states that is an enumerated state type. We will see what this means to the code later. For now, it provides a easy way to understand and to talk about what and how the state machine works.

Each possible transition between states is shown via an arc with the condition for the transition to occur shown. The condition need not be in VHDL syntax but should be understandable to the reader. Typically (highly recommended) logic expressions are given with active high assertion assumed.

It should be understood that all transitions occur on the clock edge.

Outputs from the state machine should be listed. The only outputs from this state machine are its present state. Most likely, some other state machine is watching this one’s state to determine its next state.

State Machines (cont.)

To use the enumerated state types in our example, we need to declare what they are. This would be done in the declarative area of the architecture as shown.

```
ARCHITECTURE beh OF ctrl_blk_50m IS
--declare signals and enumerated types for state machines

--further down we see.......

TYPE header_type_type IS (waiting, temp_pkt);
SIGNAL header_type_ps, header_type_ns: header_type_type;

--bla, bla, bla.....

BEGIN
```

The **TYPE** declaration states that we have a type called *header_type_type* and that the two only states for this type are *waiting* and *temp_pkt*. Having done this we can declare two signals for our present state and next state vectors called *header_type_ps* and *header_type_ns*. Note that the vectors get their names from the state machine they are apart of plus the *_ps* or *_ns* to distinguish present or next state vectors.

This style of state machine state coding is called enumerated state encoding. It is flexible in the sense that the synthesis tool is left to make the decision about how to assign a bit pattern to each state. More about this later.

Now using these state declarations, lets make the process that creates the state machine.

State Machine Process Body

Below we see the body of the process that creates the state machine.

```
header_type_sm:
 PROCESS (clk_50, reset_n, a5_or_c3, byte_assembled, byte_cnt_ps,
header_type_ps, header_type_ns)
 BEGIN
   --clocked part
   IF (reset_n = '0') THEN
     header_type_ps <= waiting;
   ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
     header_type_ps <= header_type_ns;
   END IF;

   --combinatorial part
   CASE header_type_ps IS
     WHEN waiting =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = header) AND
(a5_or_c3 = '1') THEN

        header_type_ns <= temp_pkt;
      ELSE
        header_type_ns <= waiting;
      END IF ;
     WHEN temp_pkt =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
        header_type_ns <= waiting;
      ELSE
        header_type_ns <= temp_pkt;
      END IF ;
   END CASE;
 END PROCESS header_type_sm;
```

First we see that the process label is consistent with the documentation and the signal names we have assigned. Also all the signals that may be read are listed in the process sensitivity list for the process.

```
header_type_sm:
 PROCESS (clk_50, reset_n, a5_or_c3, byte_assembled, byte_cnt_ps,
header_type_ps, header_type_ns)
```

Next the flip flops are created to hold the present state. This is what is commonly called the clocked or synchronous part since it is controlled by the clock.

State Machine Process Body (synchronous part)

```
--clocked part
IF (reset_n = '0') THEN
  header_type_ps <= waiting;
ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
  header_type_ps <= header_type_ns;
END IF;
```

Here we see an active low asynchronous reset that forces the present state to become the state called *waiting*. It does so without regard to the clock. That is why it is called an asynchronous reset.

Following the reset clause, the "clock tick event" clause identifies the following statements to be generating flip flops. The flip flops it creates are rising edge sensitive and cause the signal *header_type_ps* to take on the value of *header_type_ns* at the rising clock edge.

This concludes the clocked part of the process. We have created the necessary state flip flops and connected the D inputs to header_type_ns and the Q outputs to header_type_ps.

Now we will create the next state steering logic. It consists only of gates, i.e.; combinatorial logic. This part of the process is thus commonly called the combinatorial part of the process.

State Machine Process Body (combinatorial part)

```
 --combinatorial part
   CASE header_type_ps IS
     WHEN waiting =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = header) AND
         (a5_or_c3 = '1') THEN
        header_type_ns <= temp_pkt;
      ELSE
        header_type_ns <= waiting;
      END IF ;
     WHEN temp_pkt =>
      IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
        header_type_ns <= waiting;
      ELSE
        header_type_ns <= temp_pkt;
      END IF ;
   END CASE;
```

To clearly make the next state logic, a structure is created where **IF** statements are tucked under each distinct **CASE** state possibility. Each **CASE** possibility is a state in the state machine. Given a present state the **IF** statements determine from the input conditions what the next state is to be.

To further illustrate this:

The **CASE** statement enumerates each possible present state:

```
CASE header_type_ps  IS
     WHEN waiting =>
     --bla, bla, bla
     WHEN temp_pkt =>
     --bla, bla, bla
END CASE
```

In any given state the IF determines the input conditions to steer the machine to the next state. For example:

```
WHEN temp_pkt =>
   IF (byte_assembled = '1') AND (byte_cnt_ps = byte4) THEN
       header_type_ns <= waiting;  --go to waiting if IF is true
   ELSE
       header_type_ns <= temp_pkt; --else, stay put
  END IF ;
```

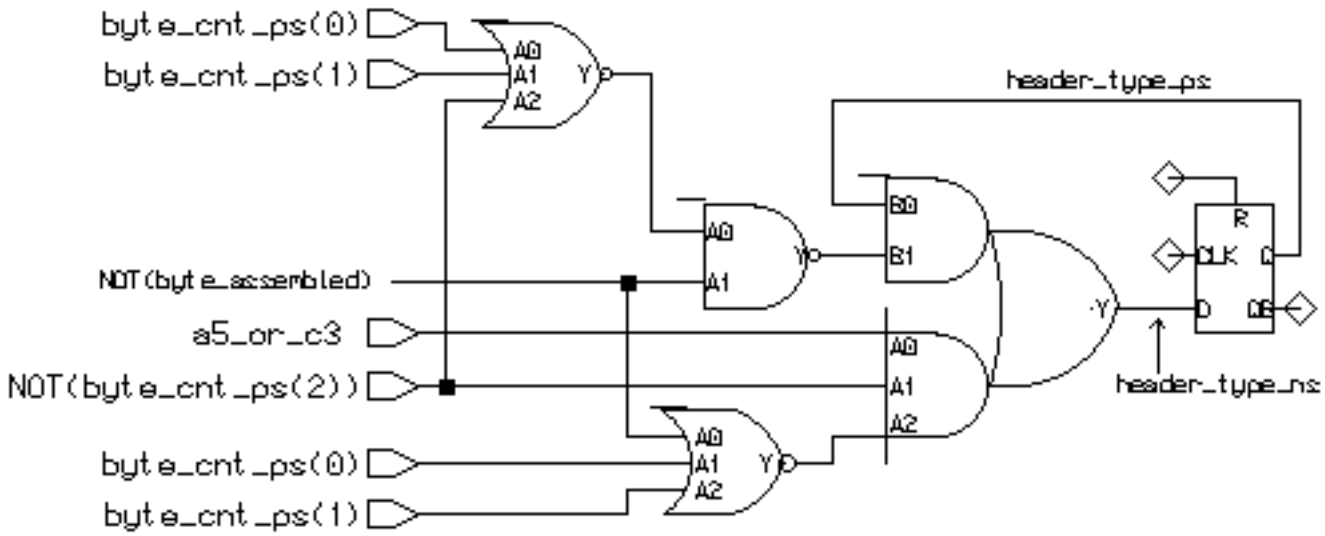
State Machine Synthesis

If we synthesize the state machine we see in the transcript:

```
"/nfs/guille/u1/t/traylor/ece574/src/header.vhd",line 28: Info,
Enumerated type header_type_type with 2 elements encoded as binary.

Encodings for header_type_type values
   value    header_type_type[0]
==============================
   waiting          0
   temp_pkt         1
```

This tells us that the synthesis tool selected the value of ‘0’ to represent the state *waiting*, and the value ‘1’ to represent the state *temp_pkt*. This makes since because we have only two states, thus 0 and 1 can represent them. We would furthermore expect only one flip flop to be needed. So the schematic looks like this: (clock and reset wiring omitted)



You might say, “That’s not the way I would do it.” But for the circuit this state machine was taken from, this was what it considered an optimal realization. Study the tas design file *ctrl_50m.vhd* and you can probably figure out some of the how and why the circuit was built this way.

The next state steering logic can be clearly seen to the left of the state storage (flip flop).

Enumerated Encoding

Using enumerated state encoding allows the synthesis tool to determine the optimal encoding for the state machine. If speed is the primary concern, a state machine could be created using one hot encoding. If minimum gate count is the most important criterion, a binary encoding might be best. For minimum noise or to minimize decoding glitching in outputs, grey coding might be best. Four different ways to encode a 2 bit state machine might be like this:

binary	00, 01, 10, 11
one hot	0001, 0010, 0100, 1000
grey	00, 01, 11, 10
random	01, 00, 11,10

While enumerated encoding is the most flexible and readable, there are cases where we want to create output signals that have no possibility of output glitches. Asynchronous FIFOs and DRAMs in particular.

As an example of a glitching state machine, lets build a two bit counter that has an output which is asserted in states “01” or “10” and is deasserted for states “00” and “11”. We will allow binary encoding of the counter.

Counter State Machine with Decoded Output

The code looks like this:

```
ARCHITECTURE beh OF sm1 IS
TYPE byte_cnt_type IS (cnt1, cnt2, cnt3, cnt4);
SIGNAL byte_cnt_ps, byte_cnt_ns:byte_cnt_type;
BEGIN
byte_cntr:
PROCESS (clk_50, reset_n, enable, byte_cnt_ps, byte_cnt_ns)
BEGIN
  --clocked part
  IF (reset_n = '0') THEN
    byte_cnt_ps <= cnt1;
  ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
    byte_cnt_ps <= byte_cnt_ns;
  END IF;
  --combinatorial part
  decode_out <= '0';  --output signal
  CASE byte_cnt_ps IS
    WHEN cnt1 =>  --output signal takes default value
    IF (enable = '1') THEN
       byte_cnt_ns <= cnt2;
     ELSE
       byte_cnt_ns <= cnt1;
     END IF ;
    WHEN cnt2 =>
    decode_out <= '1'; --output signal assigned
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt3;
     ELSE
       byte_cnt_ns <= cnt2;
     END IF ;
    WHEN cnt3 =>
     decode_out <= '1'; --output signal assigned
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt4;
     ELSE
       byte_cnt_ns <= cnt3;
     END IF ;
    WHEN cnt4 =>  --output signal takes default value
     IF (enable = '1') THEN
       byte_cnt_ns <= cnt1;
     ELSE
       byte_cnt_ns <= cnt4;
     END IF ;
  END CASE;
END PROCESS byte_cntr;
```

Specifying Outputs with Enumerated States

In the proceeding code, we see one way an output other than the present state may be created. At the beginning of the combinatorial part, before the CASE statement, the default assignment for *decode_out* is given.

```
--combinatorial part
decode_out <= '0';  --output signal
CASE byte_cnt_ps IS
```

In this example, the default value for the output *decode_out* is logic zero. The synthesis tool sees that *decode_out* is to be logic zero unless it is redefined to be logic one. In state cnt1, no value is assigned to decode_out.

```
WHEN cnt1 =>  --output signal takes default value
IF (enable = '1') THEN
   byte_cnt_ns <= cnt2;
 ELSE
   byte_cnt_ns <= cnt1;
 END IF ;
```

Thus if the present state is cnt1, decode_out remains zero. However, if the present state is cnt2, the value of decode_out is redefined to be logic one.

```
WHEN cnt2 =>
decode_out <= '1'; --output signal assigned
 IF (enable = '1') THEN
   byte_cnt_ns <= cnt3;
 ELSE
   byte_cnt_ns <= cnt2;
 END IF ;
WHEN cnt3 =>
```

We could have omitted the default assignment before the **CASE** statement and specified the value of *decode_out* in each state. But for state machines with many outputs, this becomes cumbersome and more difficult to see what is going on.

Specifying Outputs with Enumerated States

What happens if we have a state machine with an output, yet do not specify the outputs value in each state. This is similar to the situation of IF without ELSE. Latches are created on the output signal.

If we specify a “off” or default value for each output signal prior to the case statement we will never have the possibility for a latch

State Machine with Decoded Outputs

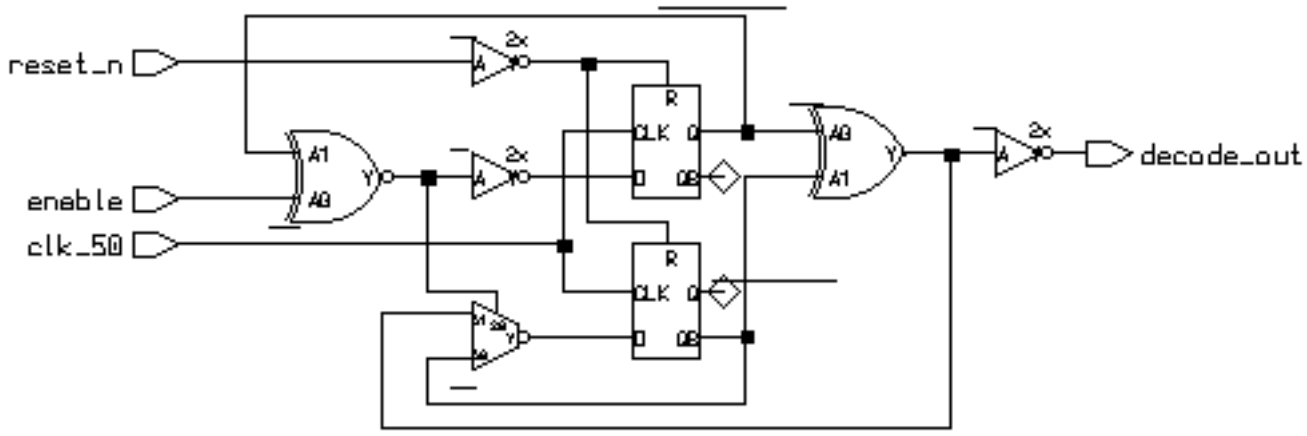
Note the declaration of the enumerated states:

```
TYPE byte_cnt_type IS (cnt1, cnt2, cnt3, cnt4);
SIGNAL byte_cnt_ps, byte_cnt_ns:byte_cnt_type;
```

Typically, (i.e., may change with tool and/or vendor) with binary encoding, the state assignments will occur following a binary counting sequence in the order in which the states are named. i.e., cnt1 = “00”, cnt2 = “01”, etc. Surely enough when the synthesis is done, we see the transcript say:

```
-- Loading entity sm1 into library work
"/nfs/guille/u1/t/traylor/ece574/src/sm1.vhd",line 23: Info,
Enumerated type byte_cnt_type with 4 elements encoded as binary.
Encodings for byte_cnt_type values
    value    byte_cnt_type[1-0]
==================================
     cnt1            00
     cnt2            01
     cnt3            10
     cnt4            11
```

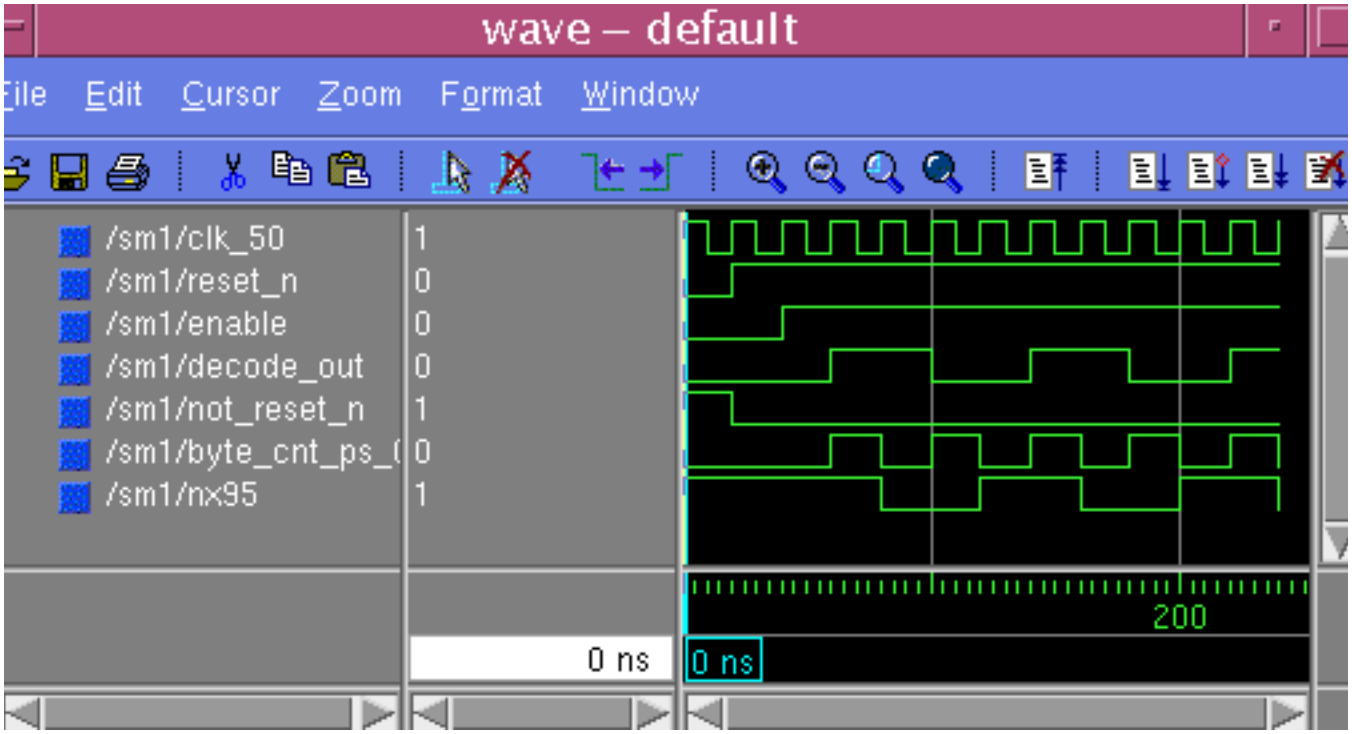
The circuit we get is shown below:



Note the output decoding created by the XOR and NOT gate. If there are unequal delays from the flip flop outputs to the XOR gate, glitches will result at the XOR output. In a “delayless” simulation this would not be seen. In a fabricated chip however, the glitches are almost a certainty.

Glitches

If we simulate this circuit without delays, we see the following. Note that the signal *decode_out* has no glitches.



When we are first creating a circuit, we usually simulate without delays. However, when we synthesize, we can specify that a "sdf" file be created. The sdf (standard delay format) file contains the delay information for the synthesized circuit.

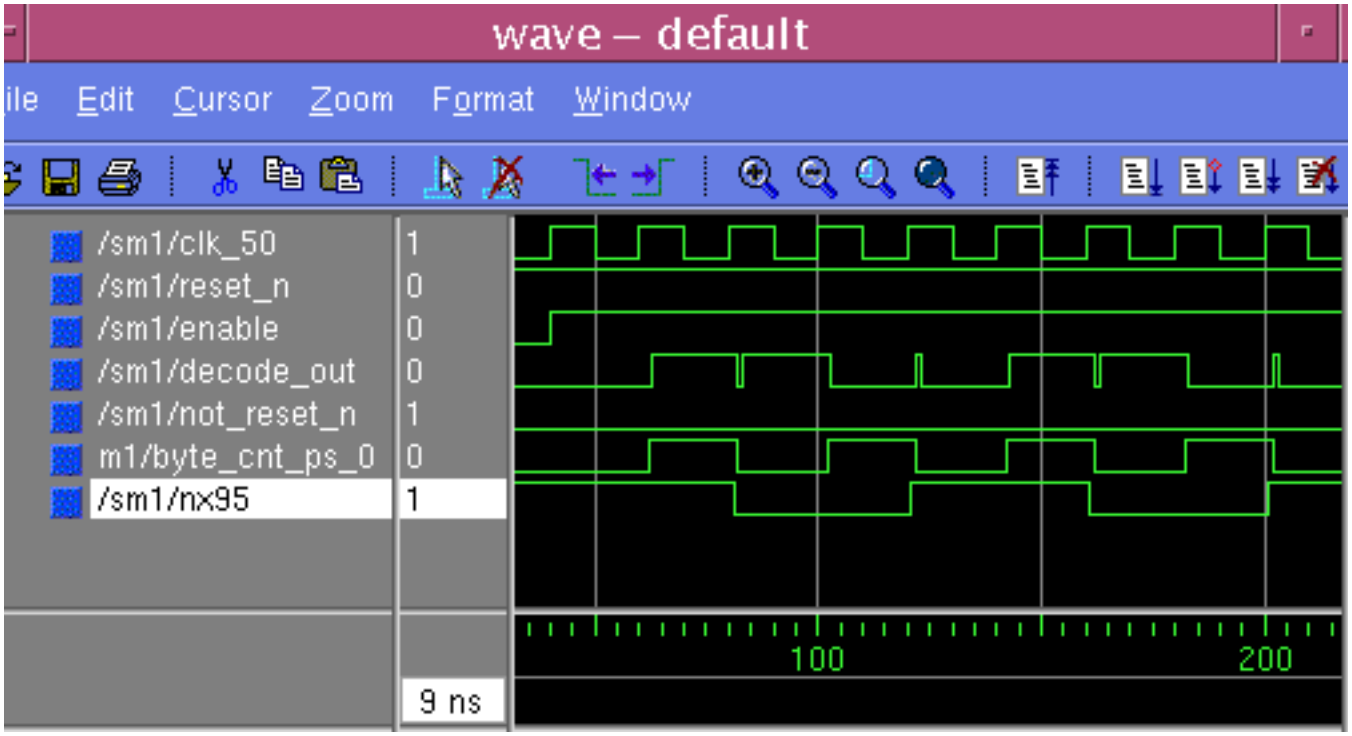
Once the synthesized circuit is created, we can invoke the vsim simulator with the sdf switch used to apply the delays to the circuit. More about this process in the latter part of the class.

So, when we invoke vsim like this:

```
vsim -sdfmax ./sdfout/sm1.sdf sm1
```

delays are added to the circuit. To make the glitch clearer, I added an additional 1ns delay to one of the flip flops by editing the sdf file. The simulation output now looks like this:

State machine output with glitches



So using enumerated types when coding state machines is a clear and flexible coding practice. However,.....the synthesizer may eat your lunch in certain situations! As folks often say, “It depends.”. If you state machine outputs go to another state machine as inputs, the glitches won’t make a bit of difference. The glitches will come only after the clock edge and will be ignored by the flip flop. But, if the outputs go to edge sensitive devices, BEWARE.

So, lets see how we can make outputs that are always clean, without glitches for those special cases.

grey coding, choosing states wisely, following flip flops, explicit states

State Encoding for Glitchless Outputs

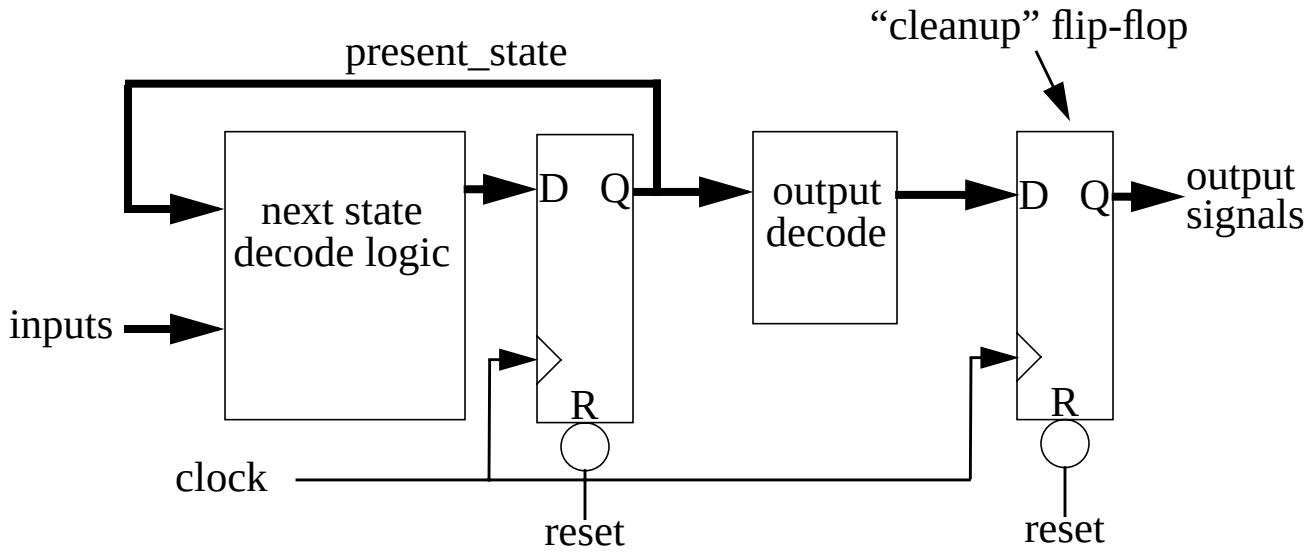
One solution to creating glitchless outputs is to strategically select the state encoding such that glitches cannot occur. This comes down to selecting states such that as the state machine goes through its sequence, the encoded states are adjacent. You can think of this as the situation in a Karnough map where two cells are directly next to each other. One of the easiest way of doing this is by using Grey coding.

If the counter advanced in the sequence “00”, “01”, “11”, “10”, no glitch would be created between states “01” and “11” as these two states are adjacent. Most synthesis tools will do grey encoding for state machines simply by setting a switch in the synthesis script.

Using grey coding in a counter makes an easy way to prevent output decoder glitches. However, in a more general state machine where many sequences may be possible and arcs extend from most states to other states, it becomes very difficult to make sure you have correctly encoded all the states to avoid a glitch in every sequence. In this situation, we can employ a more “bombproof” methodology.

Glitchless Output State Machine Methodology

The key to making glitchless outputs from our state machines is to make sure that all outputs come directly from a flip flop output. In the previous example we could accomplish this by adding an additional flip flop to the circuit.



The "cleanup" flip flop effectively removes the glitches created by the output decoder logic by "re-registering" the glitchy output signal. This is an effective solution but it delays the output signal by one clock cycle. It would be better to place the flip flop so that somehow the next state logic could create the output signal one cycle early. This is analogous to placing the output decode inside the next state decoder and adding one state bit.

The final solution alluded to above is to add one state bit that does not represent the present state but is representative only of the output signal. For the counter example we would encode the state bits like this: "000", "101", "110", "011". Thus the present state is actually broken into two parts, one representing the state (two LSB's) and the other part representing the output signal (the MSB). This will guarantee that the output signal must come from a flip flop. An another advantage is that the output signal becomes available in the same cycle as the state becomes active and with no decoder delay

Lets see how we can code this style of state machine so that the synthesis tool gives us what we want

Coding the Glitchless State Machine

Using our two-bit counter as an example here is how we could force an encoding that would allocate one flip flop output as our glitchless output.

First of all we decide up on our encoding.

present state vector consists of

"output state" bit	"present state" bits
0	00
1	01
1	10
0	11

determines the value of decode_out

keeps track of what state the counter is in in

The present state vector we will declare as STD_LOGIC_VECTOR actually consists of a one bit vector that represents the value that *decode_out* should have in the state we are in, plus two bits representing the present count state

Now, to create the present and next state vectors and assign their values as we have just stated, we do the following in the declaration area of our architecture.

```
--declare the vectors
SIGNAL byte_cnt_ns  :  STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL byte_cnt_ps  :  STD_LOGIC_VECTOR(2 DOWNTO 0);

--state encodings
CONSTANT cnt1 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "000";
CONSTANT cnt2 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "101";
CONSTANT cnt3 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "110";
CONSTANT cnt4 : STD_LOGIC_VECTOR(4 DOWNTO 0) := "011";
```

The use of CONSTANT here allows us to use the names instead of bit vectors like our enumerated example and not be guessing what state "110" is. This becomes far more important in more complex state machines.

Coding the Glitchless State Machine

The rest of our state machine is coded identically to the enumerated example given previously with two exceptions.

Remember that the output signal is now really a part of the vector *byte_cnt_ps*. How do we associate this bit of the bit vector with the output signal? We simply rip the bus. For this example:

```
decode_out <= byte_cnt_ps(2); --attach decode_out to bit 2 of _ps
```

This piece of code can be placed in a concurrent area preferably adjacent to the process containing this state machine.

Also, since we are not covering every possibility in our **CASE** statement with our four **CONSTANT** definitions, we must take care of this. To do so we utilize the **OTHERS** statement as follows:

```
 WHEN OTHERS =>
      byte_cnt_ns <= (OTHERS => '-');
END CASE; (OTHERS => '-')
```

This code segment implies when no other case matches, the next state vector may be assigned to any value. As we do not expect (outside of catastrophic circuit failure) any other state to be entered, this is of no concern. By allowing assignment of the next state vector to any value the synthesis tool can use the assigned “don’t cares” to minimize the logic.

The mysterious portion of the above: `(OTHERS => '-')`

Really just says that for every how many bits are in the vector (all the others) assign a don’t care value. Its a handy trick that allows you to modify your code later and add or subtract bits in a vector but never have to change the **OTHERS** case in your **CASE** statement.

Lets look at the new and improved state machine code and the synthesized output.

Code for the Glitchless Output State Machine

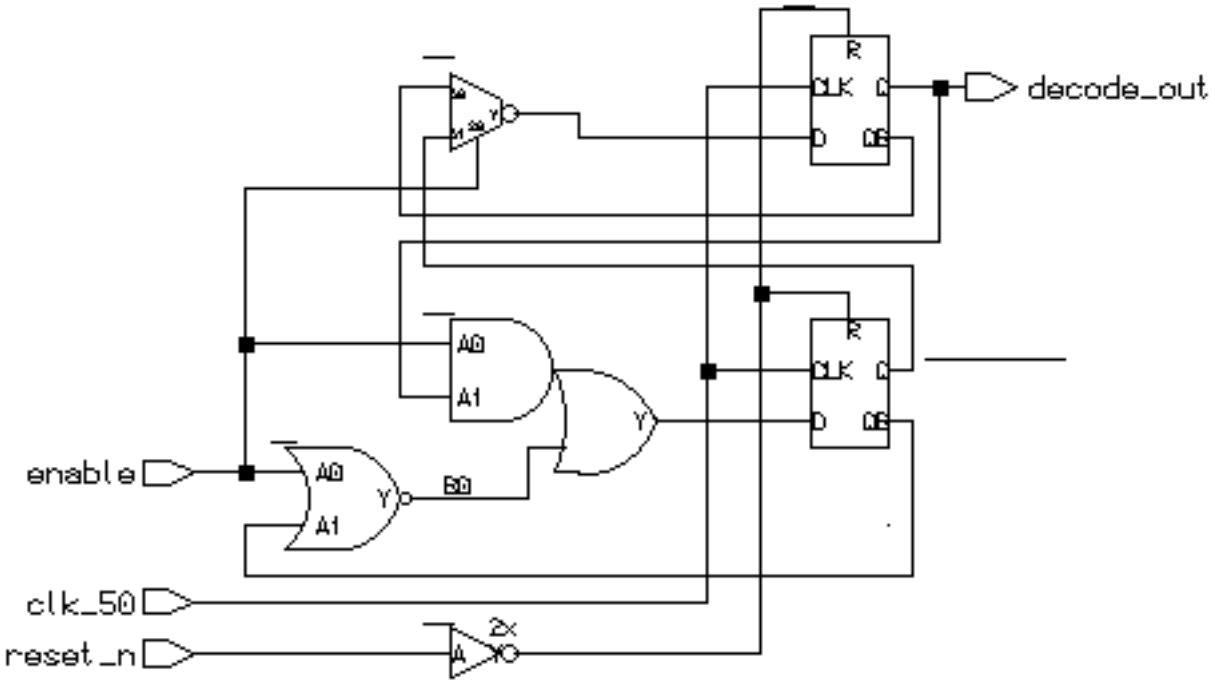
```
ARCHITECTURE beh OF sm1 IS
--declare the vectors and state encodings
SIGNAL byte_cnt_ns  :  STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL byte_cnt_ps  :  STD_LOGIC_VECTOR(2 DOWNTO 0);

CONSTANT cnt1 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
CONSTANT cnt2 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "101";
CONSTANT cnt3 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "110";
CONSTANT cnt4 : STD_LOGIC_VECTOR(2 DOWNTO 0) := "011";

 BEGIN
 byte_cntr:
 PROCESS (clk_50, reset_n, enable, byte_cnt_ps, byte_cnt_ns)
 BEGIN
   --clocked part
   IF (reset_n = '0') THEN
     byte_cnt_ps <= cnt1;
   ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
     byte_cnt_ps <= byte_cnt_ns;
   END IF;
   --combinatorial part
     CASE byte_cnt_ps IS
     WHEN cnt1 =>
     IF (enable = '1') THEN
        byte_cnt_ns <= cnt2;
      ELSE
        byte_cnt_ns <= cnt1;
      END IF ;
     WHEN cnt2 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt3;
      ELSE
        byte_cnt_ns <= cnt2;
      END IF ;
     WHEN cnt3 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt4;
      ELSE
        byte_cnt_ns <= cnt3;
      END IF ;
     WHEN cnt4 =>
      IF (enable = '1') THEN
        byte_cnt_ns <= cnt1;
      ELSE
        byte_cnt_ns <= cnt4;
      END IF ;
     WHEN OTHERS =>
        byte_cnt_ns <= (OTHERS => '-');
   END CASE;
 END PROCESS byte_cntr;
 decode_out <= byte_cnt_ps(2);  --output signal assignment
```

Synthesized Glitchless Output State Machine

Here is the synthesized output for our glitchless output state machine:



Whoa, you say. That's not what I expected. Here is a case where the synthesis tool did what you "meant" but not what "you said". We sure enough got an output from a flip flop that is glitchless but the circuit still only has two flip flops. What the synthesis tool did what to rearrange the state encoding such that the bit that *decode_out* is tied to is one in states cnt2 and cnt3. In other words, it Grey coded the states to avoid the extra flip flop.

Other tools may or may not behave in the same way. Once again, it pays to checkout the transcript, take a look at the gates used and take a peek at the schematic. The synthesis transcript did mention what was done in a vague sort of way:

```
-- Compiling root entity sm1(beh)
"/nfs/guille/u1/t/traylor/ece574/src/sm1.vhd", line 27:
Info, D-Flipflop reg_byte_cnt_ps(0) is unused, optimizing...
```

The moral of the story... read the transcripts. Don't trust any tool completely. Double check everything. "The paranoid survive."..... Andy Grove

State Machines in VHDL - General Form

As seen in the previous example, most state machines in VHDL assume the following form:

```
process_name:
PROCESS(sensitivity_list)
BEGIN
--synchronous portion
  IF (reset = ‘1’) THEN
    present_state <= reset_conditon;
  ELSIF (clk’EVENT AND clk = ‘1’) THEN
    present_state <= next_state;
  END IF;
--combinatorial part
  CASE present_state IS
     WHEN state1 =>
        next_state <= state_value;
        other_statements;
     WHEN state2 =>
        next_state <= state_value;
        other _statements;
                    *
                    *
     WHEN OTHERS => next_state <= reset_state;
   END CASE;
END PROCESS;
```