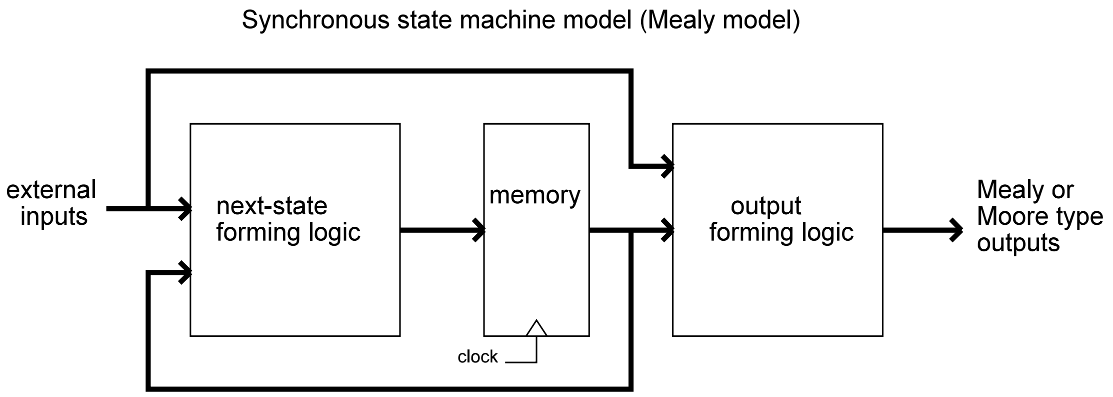
How to Create a State Machine Design Using VHDL

To possibly restate what you know, a state machine is typically composed of three main blocks as shown in figure 1, namely Next State Forming logic, Memory, and Output Forming logic.



Synchronous state machine model (Mealy model)
external inputs
next-state forming logic
memory
clock
output forming logic
Mealy or Moore type outputs

Figure 1 - State machine model

Design of a synchronous state machine that will be implemented with VHDL should follow this outline:

A) Start with a clear definition of the required functionality
B) Create a state diagram showing the needed states, branching conditions, and outputs.
C) Write VHDL statements using a multi-segment style description of the state machine.
  a) State machine memory should be defined with a process dedicated to it
  b) Next state forming logic can be defined with concurrent signal assignment statements (such as conditional signal assignment statements or selected signal assignment statements) or with sequential style statements that are defined within a process (such as if-elsif-else statements or a case statement). A case statement format lends itself to clearly showing the states and the actions to be taken in each state.
  c) Output logic may be defined separate from the next state forming logic (creating a so-called three segment style) or combined with the next state logic (resulting in a two segment style). Use a style that makes it easiest to understand your design.

As an example, consider the design of a state machine whose state diagram is shown in figure 2 (steps A and B in the outline above have already been done):

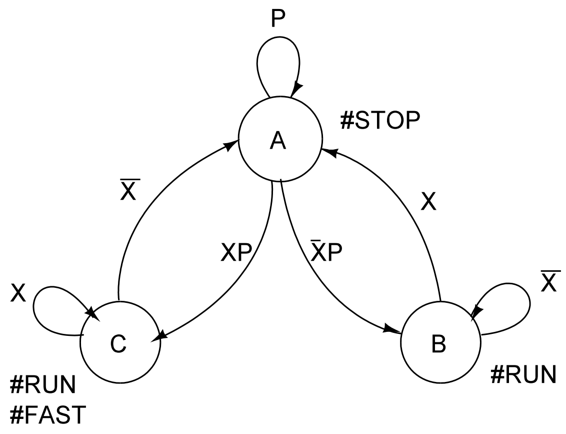
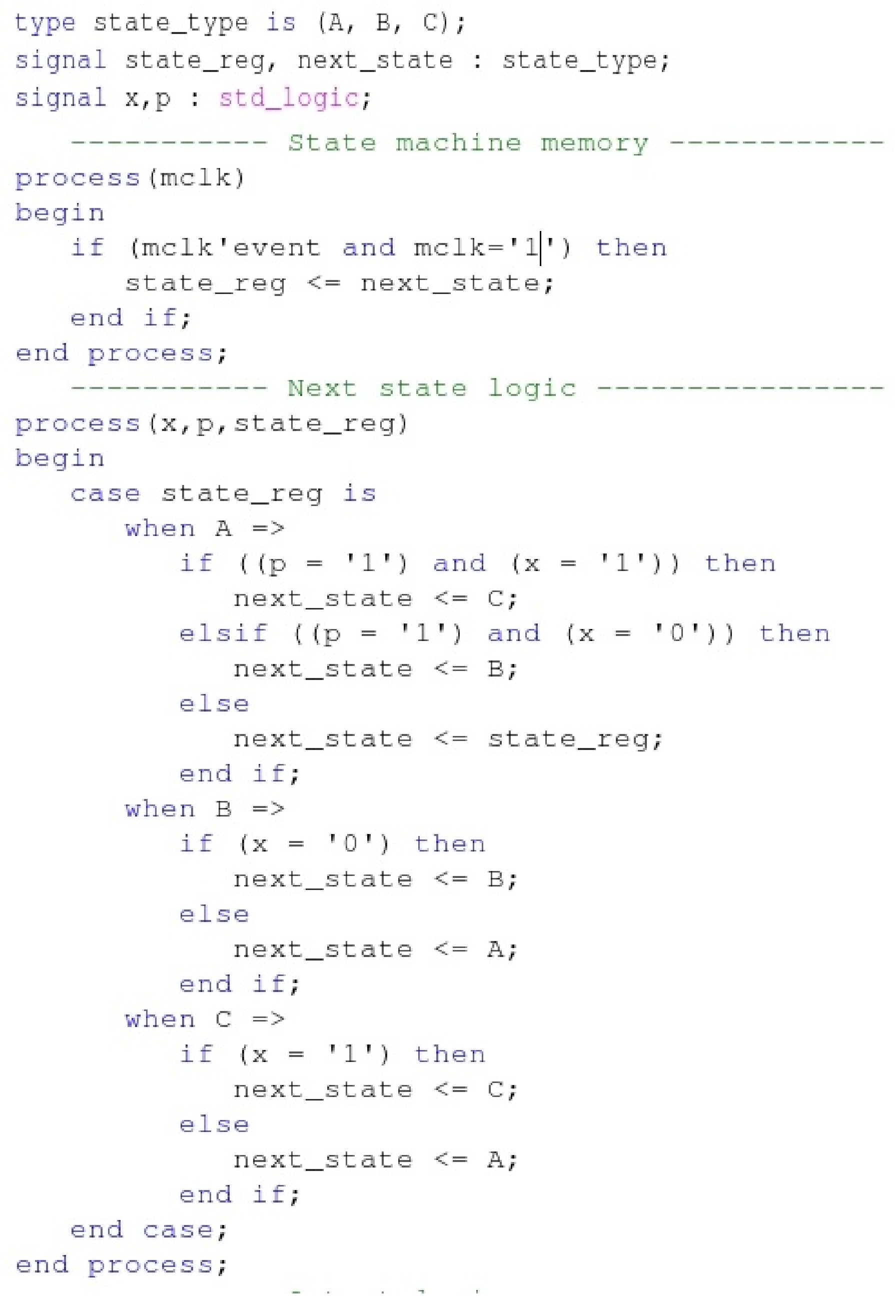


Figure 2 - State diagram for the state machine to be implemented

A 3-segment style VHDL state machine description for figure 2 is shown below.

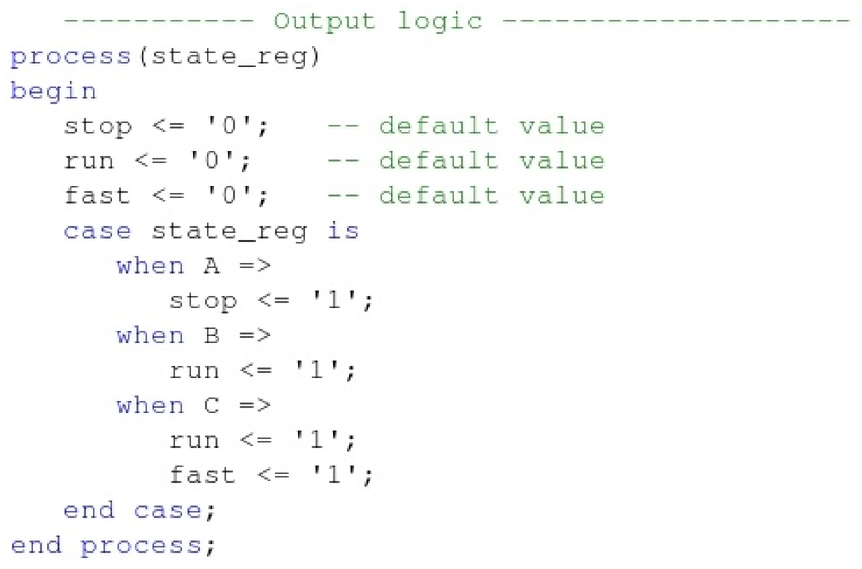
While state numbers can be assigned explicitly, using symbolic names may be more descriptive and the synthesizer is capable of assigning binary values to the symbolic names. By default, the style of encoding (binary or One-hot encoding) used by the synthesizer is set to automatic meaning the synthesizer has some algorithm for deciding the style to use. While it is possible to force a particular style, for our class work we will allow the synthesizer to make the choice.

Signals used to create the state machine need to be defined in the architecture prior to the architecture begin statement. Note the creation of a data type called state_type (that is not a key word, but rather a name chosen for this example) and its subsequent use in defining signals for the state machine memory and next_state.



```
type state_type is (A, B, C);
signal state_reg, next_state : state_type;
signal x,p : std_logic;
   ----------- State machine memory ------------
process(mclk)
begin
   if (mclk'event and mclk='1') then
      state_reg <= next_state;
   end if;
end process;
   ----------- Next state logic ----------------
process(x,p,state_reg)
begin
   case state_reg is
      when A =>
         if ((p = '1') and (x = '1')) then
            next_state <= C;
         elsif ((p = '1') and (x = '0')) then
            next_state <= B;
         else
            next_state <= state_reg;
         end if;
      when B =>
         if (x = '0') then
            next_state <= B;
         else
            next_state <= A;
         end if;
      when C =>
         if (x = '1') then
            next_state <= C;
         else
            next_state <= A;
         end if;
   end case;
end process;
```

As mentioned above, a 3-segment style state machine is illustrated here meaning that the output logic is written separate from the next-state logic. I recommend using the 3-segment style for your first designs that include a state machine. And for this class, either 2-segment or 3-segment style is required for all labs.



```
   ----------- Output logic --------------------
process(state_reg)
begin
   stop <= '0';   -- default value
   run <= '0';    -- default value
   fast <= '0';   -- default value
   case state_reg is
      when A =>
         stop <= '1';
      when B =>
         run <= '1';
      when C =>
         run <= '1';
         fast <= '1';
   end case;
end process;
```

Continue to keep in mind that the sequential style statements are evaluated by the synthesizer to create circuits that operate concurrently.