

# Graphical Behavior Capture to VHDL

Moshe S. Cohen  
i-Logix, Inc.  
22 3rd Av. Burlington, MA, 02215  
e-mail: mc@ilogix or uunet!ilogix!mc

## Abstract

This paper presents Statecharts as a graphical entry for automatic generation of synthesizable VHDL descriptions. Using a concrete example, the process of creating the Statechart model is shown, along with the VHDL description and the circuit that was automatically generated from the Statecharts.

This paper also demonstrates the superiority of Statecharts over Communicating State Machines in terms of clarity, compactness of the description and compactness of the resulting circuit.

## Introduction

Statecharts are a method to graphically describe and analyze a behavioral model. Based on the concept of Finite State Machines (FSM) and augmented by hierarchy and concurrency, they are very useful for the development and debugging of behavioral models as well as for communicating designs with others in the development team.

The similarities between the Statechart semantics and VHDL allow Statecharts to be translated into a very compact and efficient VHDL code. The generated code not only fully describes the model behavior, it also preserves the hierarchy and concurrency described in a Statechart model. This results in a more efficient circuit than if designed otherwise.

This paper consists of two parts. The first part describes the process of developing a Statechart model for the receiver portion of a Universal Asynchronous Receiver Transmitter (UART), the code generated from this model, and the schematics derived from the VHDL code. The second part describes a simple typical Statechart and compares it to the equivalent behavior described as communicating State Machines. This paper demonstrates that the use of Statecharts is far more intuitive than traditional FSM and that the resulting circuit is more compact.

## A UART Receiver:

The example discussed will be the receiver portion of a UART. The receiver reads a serial line and converts it to a parallel byte. The receiver starts when the line becomes '1'. It then reads eight consecutive bits. If these bits have the correct parity, the CPU is notified that a valid byte has been read; otherwise, the receiver enters an error mode.

The above behavior can be described as the simple state machine which is shown in Fig. 1. This state machine has four states: **wait\_for\_start**, **read\_bits**, **allow\_read** and **parity\_error**.

Upon entering the initial state, **wait\_for\_start** mode, four actions take place:

*parity\_error:=0*            Deassert the *parity\_error* signal.  
*read\_enable:=0*            Deassert the valid byte indicator.  
*parity\_bit:=parity\_type*    Set the *parity\_bit* to '1' for odd or '1' for even parity.  
*carry:=0*                    Set the carry bit to '0'.

Upon detecting a start bit, which is defined as the event of the input *line\_in* becoming '1'<sup>1</sup>, the shift register is loaded with the binary value of '1000 0000' and the receiver enters the **read\_bits** state.

While in the **read\_bits** mode, the receiver loops as long as the carry is '0'<sup>2</sup>. On every loop (that satisfies this condition) the following actions are executed (see notes at bottom Figure 1):

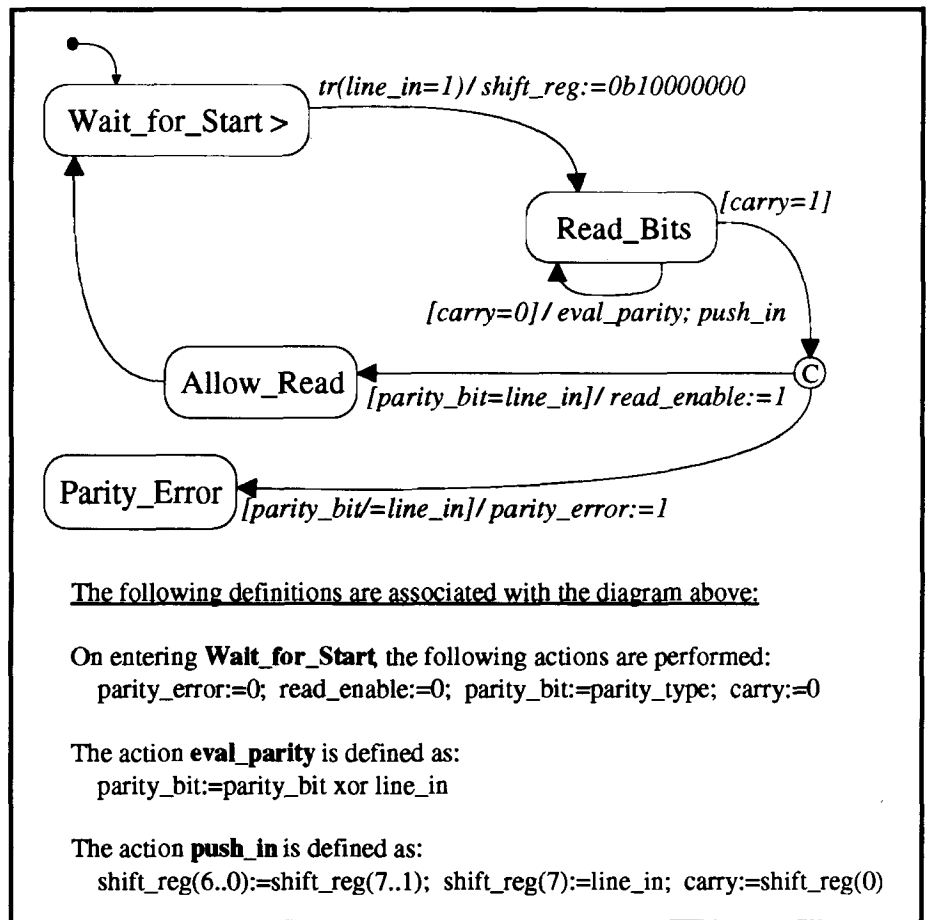
*parity\_bit:= parity\_bit xor line\_in* - evaluate the parity based on the previous parity and the current bit.

*shift\_reg(6..0):= shift\_reg(7..1)* - shift the *shift\_reg* by one bit;

*shift\_reg(7):= line\_in* - to load the *line\_in* into the shift register;

*carry:= shift\_reg(0)* - the carry will become '1' after eight shift operations.

Once the shift is completed, if the parity equals the next value on the line (which is the transmitted parity bit), the *read\_enabled* signal is made high for one clock; otherwise, a *parity\_error* signal is raised.



Using i-Logix's ExpressV-HDL or VIEWlogic's Envision Statecharts can be translated to VHDL of different flavors. For this model, VHDL code will be generated for the Synopsys synthesizer. This code is shown on the next page.

<sup>1</sup> The syntax 'tr(line\_in=1)' denotes the event of 'line\_in=1' becoming satisfied (edge sensitive).  
<sup>2</sup> The syntax '[carry=0]' denotes the condition 'carry=0' being true (level sensitive).

```

use WORK.SYNOPSIS.ALL;
entity low_receiver is
  port
    ( CLOCK:          in  bit;
      RST:            in  bit;
      gdiPARITY_TYPE: in  bit;
      gdiLINE_IN:    in  bit;
      gdiSHIFT_REG:  inout unsigned(7 downto 0);
      gdiREAD_ENABLE: out bit;
      gdiPARITY_ERROR: out bit );
end low_receiver;

architecture i_Logix of low_receiver is
  type tp_CHART_states is
    (st_WAIT_FOR_START, st_PARITY_ERROR,
     st_ALLOW_READ, st_READ_BITS);

  signal st_CHART_isin: tp_CHART_states;
  signal diCARRY:      bit;
  signal diPARITY_BIT: bit;
  signal prev_diLINE_IN: bit;

begin
  exec_all_process : process(CLOCK, RST)

    procedure exec_act_PUSH_IN is
    begin
      gdiSHIFT_REG(6 downto 0) <=
        gdiSHIFT_REG(7 downto 1);
      gdiSHIFT_REG(7) <= gdiLINE_IN;
      diCARRY <= gdiSHIFT_REG(0);
    end exec_act_PUSH_IN;

    procedure exec_st_CHART is
    begin
      case st_CHART_isin is
        when st_WAIT_FOR_START =>
          if (gdiLINE_IN = '1') and
             (gdiLINE_IN /= prev_gdiLINE_IN) then
            gdiSHIFT_REG <= "10000000";
            st_CHART_isin <= st_READ_BITS;
          end if;
        when st_ALLOW_READ =>
          gdiPARITY_ERROR <= '0';
          gdiREAD_ENABLE <= '0';
          diCARRY <= '0';
          diPARITY_BIT <= gdiPARITY_TYPE;
          st_CHART_isin <= st_WAIT_FOR_START;

          when st_READ_BITS =>
            if diCARRY = '0' then
              diPARITY_BIT <= diPARITY_BIT xor gdiLINE_IN;
              exec_act_PUSH_IN;
              st_CHART_isin <= st_READ_BITS;
            elsif diCARRY = '1' and diPARITY_BIT = gdiLINE_IN
              then
              gdiREAD_ENABLE <= '1';
              st_CHART_isin <= st_ALLOW_READ;
            elsif diCARRY = '1' and diPARITY_BIT /= gdiLINE_IN
              then
              gdiPARITY_ERROR <= '1';
              st_CHART_isin <= st_PARITY_ERROR;
            end if;
          when others =>
            null;
          end case;
        end exec_st_CHART;

    procedure exec_reset is
    begin
      diCARRY <= '0';
      diPARITY_BIT <= '0';
      gdiSHIFT_REG <= "00000000";
      gdiREAD_ENABLE <= '0';
      gdiPARITY_ERROR <= '0';
      gdiPARITY_ERROR <= '0';
      gdiREAD_ENABLE <= '0';
      diCARRY <= '0';
      diPARITY_BIT <= gdiPARITY_TYPE;
      st_CHART_isin <= st_WAIT_FOR_START;
    end exec_reset;

  begin
    if RST = '1' then
      exec_reset;
    elsif (CLOCK'event and CLOCK='1') then
      prev_gdiLINE_IN <= gdi_LINE_IN;
      exec_st_CHART;
    end if;
  end process exec_all_process;

end i_Logix;

```

Figure 2: VHDL code, automatically generated from the statechart in Fig 1.

The code shown in Figure 2 was generated automatically from the Statechart in Figure 1. The code is easy to read to the extent that the information from the Statechart can be determined directly by reading the code. Names used in the code are derived from those used in the graphical model; they are prefixed by the letters 'co' for conditions, 'di' for data items, 'st' for states and 'g' for global signals. Synthesizing this code using the LSI-10k library yields the schematics given in Appendix A.

The receiver can be made more realistic by adding the following requirements:

1. The receiver is active only when bit 0 of the Control & Status Register is high.
2. It is not realistic to require the CPU to read the byte within a single clock period after this byte has been received. The Receiver buffers the byte so that the CPU can read it while the Receiver reads a new byte.
3. The entire receiver is reset while CSR(1) is '1' and CSR(0) is '0'.

The first requirement can be addressed by adding a new state called **Rx\_Disabled** and transitions going from the four other states back to **Rx\_Disabled** as outlined in Figure 3.a.

This approach is acceptable when such a modification in the requirements results in a small number of states being augmented by one more. However, as the number of states increases, the diagram becomes less manageable as one can see from Figure 3.a.

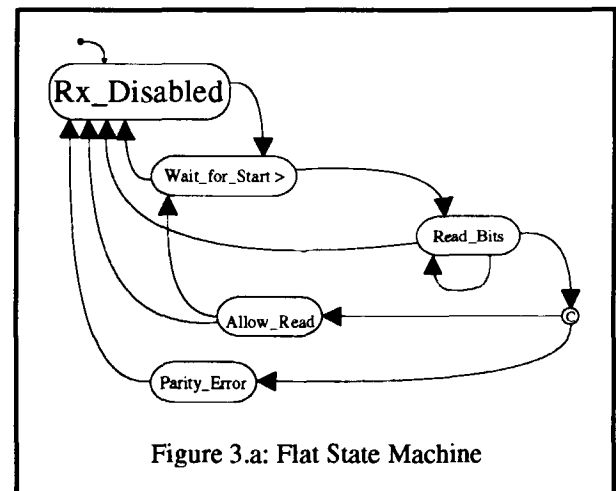


Figure 3.a: Flat State Machine

Another approach is to use **Hierarchical States** as shown in Figure 3.b. The four states are encapsulated by a higher state, **Rx\_Enabled**, and another state, called **Rx\_Disabled**, is added. In this case, only two additional transitions are required. These transitions are triggered by the value of CSR(0). Whenever CSR(0) is '0', the receiver leaves the **Rx\_Enabled** state, regardless of the specific sub-state it is in.

Note, Figures 3.a and 3.b are semantically identical, however, 3.b is simpler to read, debug and understand.

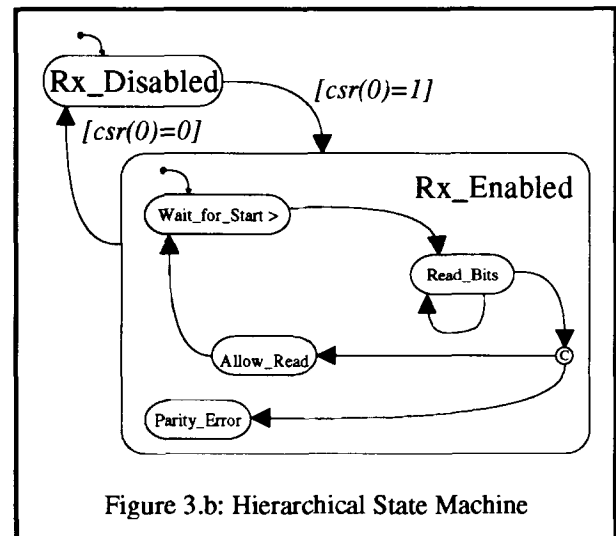


Figure 3.b: Hierarchical State Machine

Hierarchy is not sufficient in order to provide the buffering capability since the CPU must be able to read the buffer *any time during* the Receiver operation.

In Figure 4, the **Rx\_Enabled** and **Rx\_Disabled** are encapsulated into a higher state, called **Rx\_Mode**, which is concurrent to the state called **Rx\_Buffer**. The concurrency is defined using a dotted line which separates the two states, **Rx\_Mode** and **Rx\_Buffer**. Initially, the buffer is empty. If  $read\_enable = 1$ , the output  $fifo\_full$  becomes high, and the  $shift\_reg$  is loaded into the  $hold\_reg$ . If while in this mode, there is another occurrence of  $read\_enable = 1$ , an overrun error is asserted.

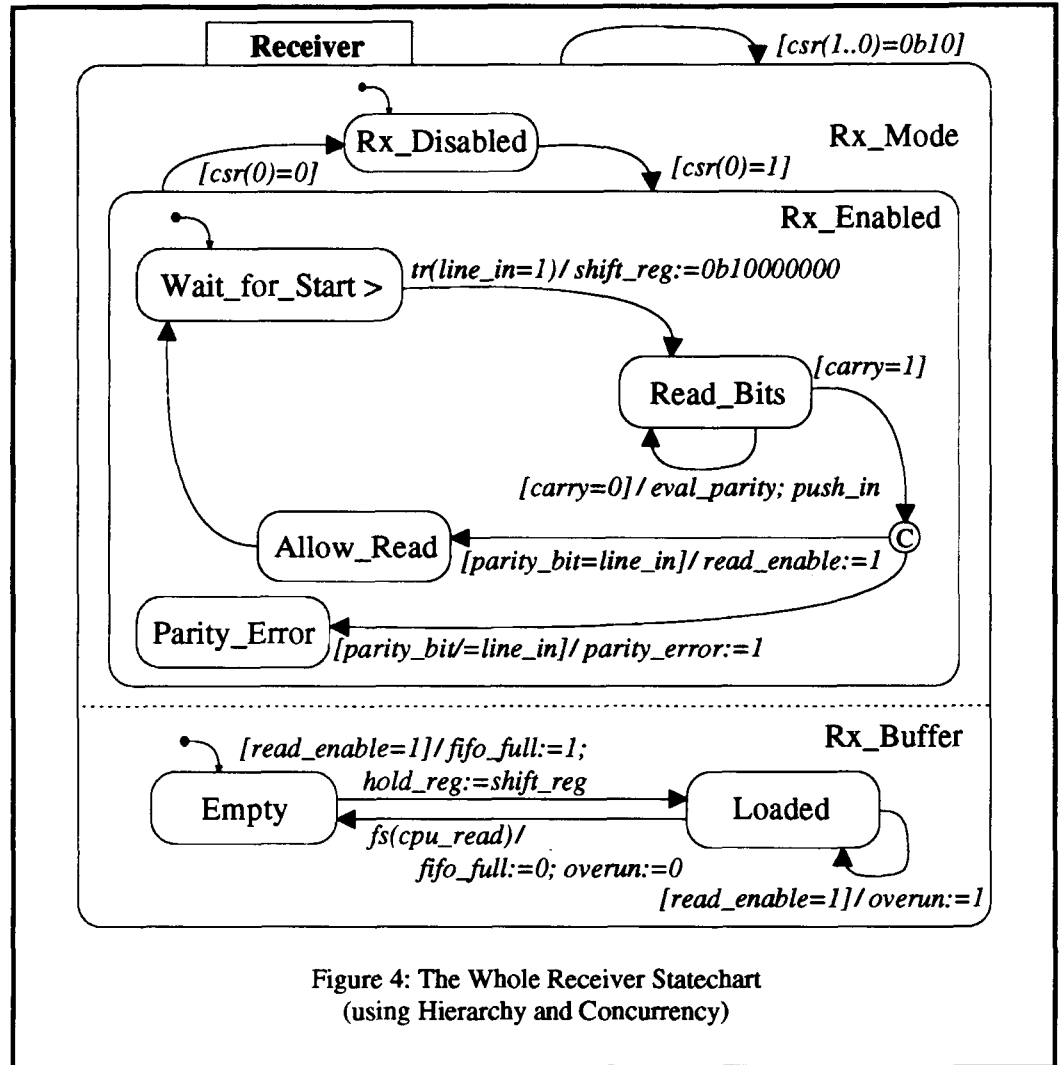


Figure 4: The Whole Receiver Statechart (using Hierarchy and Concurrency)

The third requirement is very simple to accomplish. A high level transition, labeled as  $[csr(1..0)=0b10]$  is added to the **Receiver** state. Upon taking this transition, the Receiver re-enters the **Rx\_Disabled** and **Rx\_Buffer.Empty** states.

The code generated from Figure 4 is shown on the next page. As one can see, the code remains easy to read, although it describes the more complex behavior of the modified Statechart. Notice how hierarchy and concurrency are handled in the code. These features will be further discussed. The schematics derived from this code are also given in Appendix A.

```

use WORK.SYNOPSYS.ALL;
use work.EXV_PACKAGE.all;
entity receiver is
  port
    ( CLOCK:          in  bit;
      gdiPARITY_TYPE: in  bit;
      gdiLINE_IN:     in  bit;
      gdiCSR:         in  unsigned(1 downto 0);
      gcoCPU_READ:    in  boolean;
      gdiPARITY_ERROR: out bit;
      gdiOVERUN:      out bit;
      gdiHOLD_REG:    out unsigned(7 downto 0);
      gdiFIFO_FULL:   out bit );
end receiver;

architecture i_Logix of receiver is
-- types and signals definition are ommitted

begin
  exec_all_process : process
    procedure exec_act_PUSH_IN is
    begin
      diSHIFT_REG(6 downto 0) <= diSHIFT_REG(7 downto 1);
      diSHIFT_REG(7) <= gdiLINE_IN;
      diCARRY <= diSHIFT_REG(0);
    end exec_act_PUSH_IN;

    procedure exec_st_RX_ENABLED is
    begin
      case st_RX_ENABLED_isin is
        when st_WAIT_FOR_START =>
          if (gdiLINE_IN = '1') and (gdiLINE_IN/=prev_gdiLINE_IN) then
            diSHIFT_REG <= "10000000";
            st_RX_ENABLED_isin <= st_READ_BITS;
          end if;
        when st_ALLOW_READ =>
          gdiPARITY_ERROR <= '0';
          diREAD_ENABLE <= '0';
          diCARRY <= '0';
          diPARITY_BIT <= gdiPARITY_TYPE;
          st_RX_ENABLED_isin <= st_WAIT_FOR_START;
        when st_READ_BITS =>
          if diCARRY = '0' then
            diPARITY_BIT <= diPARITY_BIT xor gdiLINE_IN;
            exec_act_PUSH_IN;
            st_RX_ENABLED_isin <= st_READ_BITS;
          elsif diCARRY = '1' and diPARITY_BIT = gdiLINE_IN then
            diREAD_ENABLE <= '1';
            st_RX_ENABLED_isin <= st_ALLOW_READ;
          elsif diCARRY = '1' and diPARITY_BIT /= gdiLINE_IN then
            gdiPARITY_ERROR <= '1';
            st_RX_ENABLED_isin <= st_PARITY_ERROR;
          end if;
        end case;
      end exec_st_RX_ENABLED;

    procedure exec_st_RX_BUFFER is
    begin
      case st_RX_BUFFER_isin is
        when st_EMPTY =>
          if diREAD_ENABLE = '1' then
            gdiFIFO_FULL <= '1';
            gdiHOLD_REG <= diSHIFT_REG;
            st_RX_BUFFER_isin <= st_LOADED;
          end if;
          when st_LOADED =>
            if ((gcoCPU_READ/=prev_gcoCPU_READ) and not gcoCPU_READ) then
              gdiFIFO_FULL <= '0';
              gdiOVERUN <= '0';
              st_RX_BUFFER_isin <= st_EMPTY;
            elsif diREAD_ENABLE = '1' then
              gdiOVERUN <= '1';
              st_RX_BUFFER_isin <= st_LOADED;
            end if;
          end case;
        end exec_st_RX_BUFFER;

    procedure exec_st_RX_MODE is
    begin
      case st_RX_MODE_isin is
        when st_RX_DISABLED =>
          if gdiCSR(0) = '1' then
            st_RX_MODE_isin <= st_RX_ENABLED;
            gdiPARITY_ERROR <= '0';
            diREAD_ENABLE <= '0';
            diCARRY <= '0';
            diPARITY_BIT <= gdiPARITY_TYPE;
            st_RX_ENABLED_isin <= st_WAIT_FOR_START;
          end if;
          when st_RX_ENABLED =>
            if gdiCSR(0) = '0' then
              st_RX_MODE_isin <= st_RX_DISABLED;
            else
              exec_st_RX_ENABLED;
            end if;
          end case;
        end exec_st_RX_MODE;

    procedure exec_st_Chart is
    begin
      case st_Chart_isin is
        when st_RECEIVER =>
          if gdiCSR(1 downto 0) = 2#10# then
            st_Chart_isin <= st_RECEIVER;
            st_RX_MODE_isin <= st_RX_DISABLED;
            st_RX_BUFFER_isin <= st_EMPTY;
          else
            exec_st_RX_MODE;
            exec_st_RX_BUFFER;
          end if;
        end case;
      end exec_st_Chart;

    begin
      wait until CLOCK'event and CLOCK='1';
      prev_gcoCPU_READ <= gcoCPU_READ;
      prev_gdiLINE_IN <= gdiLINE_IN;
      exec_st_Chart;
    end process exec_all_process;
  end i_Logix;

```

Figure 5: Code generated automatically for the Receiver described in Figure 4

Before we move on to a more theoretical discussion, I must admit that, although I am a strong advocate of top-down approaches, I find it more appropriate to describe this Receiver bottom-up. This is the writer's own preference for this particular case; Statecharts, in general, can be used equally well in either approaches.

### Statechart vs. Communicating State Machines

A very common and typical statechart is one where after some initialization, several machines are triggered simultaneously. These machines are usually not totally independent - they may affect the behavior of each other<sup>1</sup>. This section describes such a generic Statechart, and compares it to another description of the same behavior, one based on traditional communicating finite state machines.

#### Notes on the Statechart S (shown in Fig. 5):

- S begins execution in the state S0. If the condition *c1* is true, it enters S1; if *c2* is false, it assigns '1' to the output *d* and enters S2.
- Entering S2 means entering both P1 and Q1.
- If *c1* is true, P enters P2, making '[in(P2)]' true.
- [in(P2)] being true causes two things:
  - taking the transition from Q1 to Q2;
  - executing the action 'tr!(c)' (this should be read as - make the output *c* true).
- While in the state Q2, the input *c2* is monitored. Whenever it is true, Q transitions back to Q1 while making the output *c* false.
- If anytime while in any states within S2, the input *c1* becomes false, all states within S2 are exited, the machine transitions back to S0, and the output *d* is assigned the value '0'.

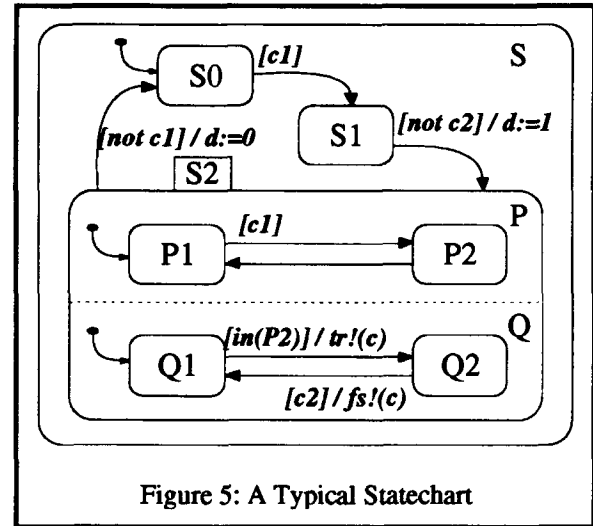


Figure 5: A Typical Statechart

The behavior described by the Statechart in Figure 5 is now described using three State Machines, given in Figures 6.a, 6.b, and 6.c.

#### Notes on the S Machine (given in Fig. 6.a):

- S describes the transitions between S0, S1 and S2.
- On entering S2, a signal called *in\_s2* is made true and it is set to false on exiting S2. This signal will be used to control two other machines - one for a P machine, another one for a Q machine.

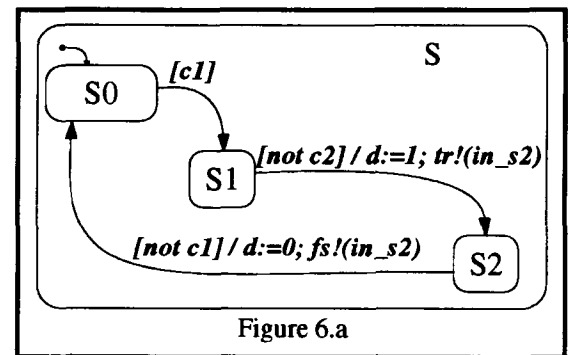
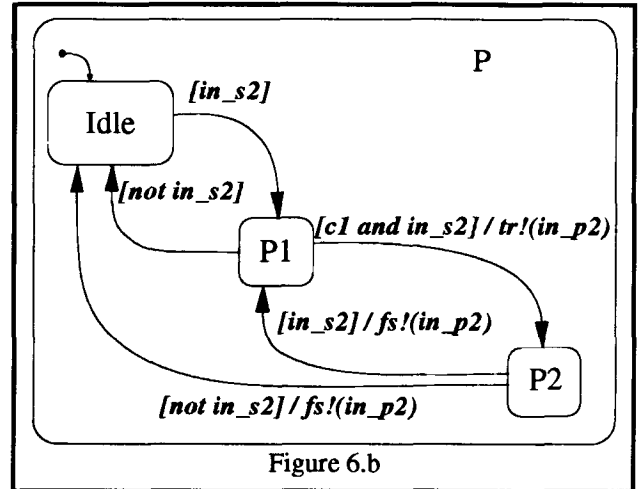


Figure 6.a

<sup>1</sup> An example for such a machine can be found in the Ethernet protocol. If the transmitter is ready to send a packet, and no other device is transmitting on the line, the Ethernet controller transmits its packet while monitoring the line to detect any possible collisions. If a collision is detected, the transmitter continues by jamming the line for a predefined number of bits.

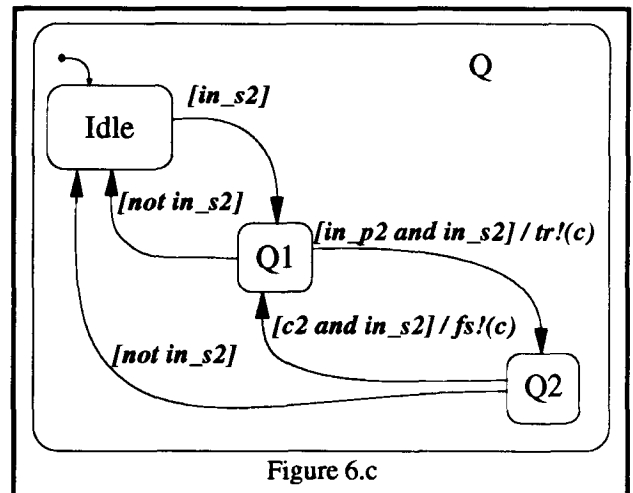
Notes on the P machine (given in Fig. 6.b):

- Initially, **P** is **Idle**. When *in\_s2* becomes high, it enters **P1**.
- While in **P1**, it may go back to **Idle** if *in\_s2* becomes false, or to **P2** if *c1* and *in\_s2* are both true.
- On entering **P2**, a signal called *in\_p2* is made true (this will be used by the **Q** machine)
- Once in **P2**, it may return back to **Idle** or to **P1** based on *in\_s2*.
- On exiting **P2**, the output *in\_p2* is made false.



Notes on the Q machine (given in Fig. 6.c):

- Initially, **Q** is **Idle**. When *in\_s2* becomes high, it enters **Q1**.
- While in **Q1**, it may go back to **Idle** if *in\_s2* becomes false, or to **Q2** if *in\_p2* and *in\_s2* are both true.
- On entering **Q2**, the output *c* is made true.
- Once in **Q2**, it may return back to **Idle** if *in\_s2* becomes false, or to **Q1** if *in\_s2* and *c2* are true.
- On exiting **Q2**, the output *c* is made false.



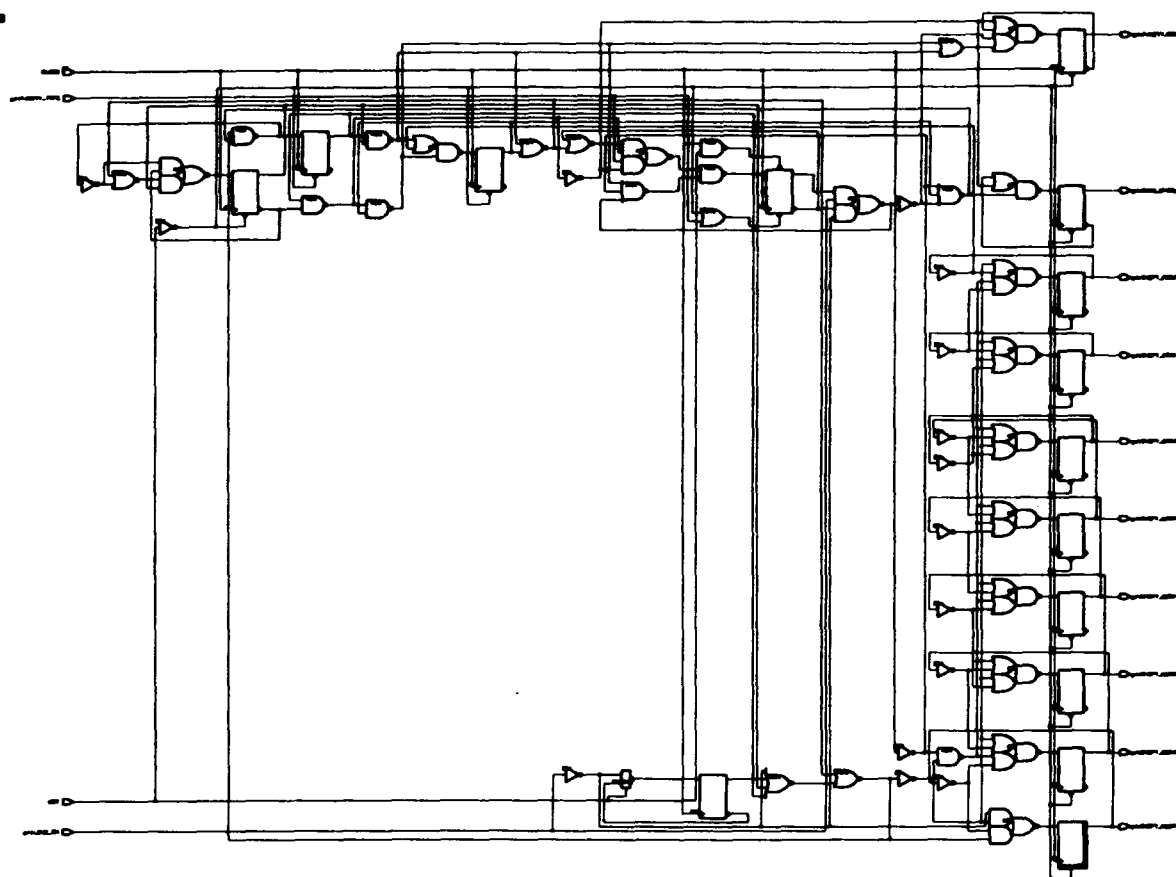
The Statechart shown in Figure 5 was mapped into schematics using the Synopsys tool, the LSI-10k library, and optimizations for smallest area. Figures 6.a, 6.b, 6.c where all mapped in the same way, and then the structure defined by the three machines was removed (ungrouped) and re-optimized for area.

The Statechart given in Figure 5 was mapped into 86 gates  
 The equivalent machines (Fig. 6.a, 6.b, 6.c) where mapped into 127 gates

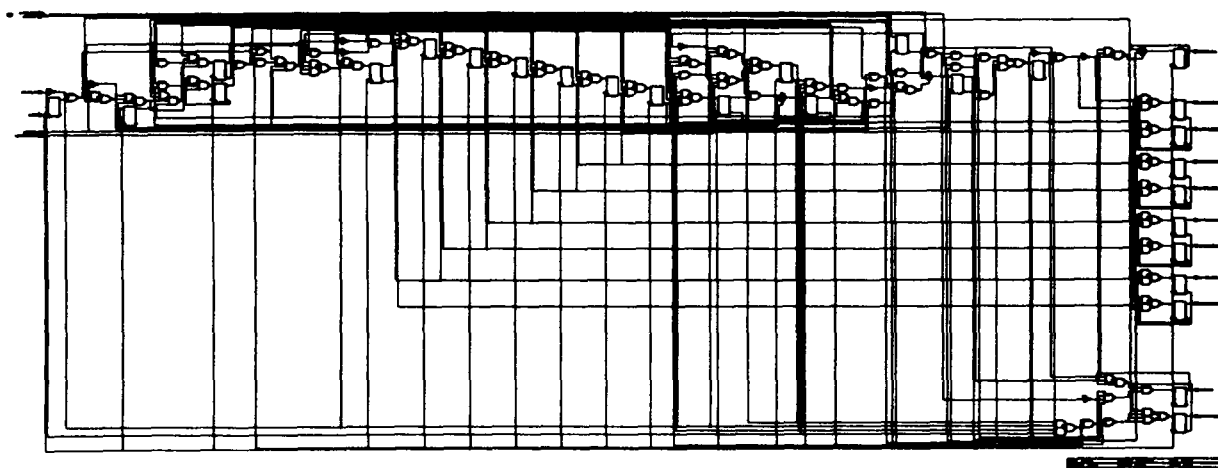
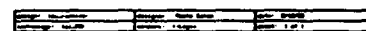
**To summarize:**

- Statecharts are very much like Finite State Machines, but are extended by hierarchy and concurrency.
- A Statechart model is easy to develop, easy to communicate and probably, easier to debug.
- There is a natural mapping between Statecharts and VHDL.
- The VHDL generated from Statecharts models is compact and efficient.
- Schematics synthesized from Statecharts models are more efficient than if designed otherwise.

## Appendix A: Schematics generated by the Synopsys tool.



Schematics for statechart in Figure 1



Schematics for statechart in Figure 4