Logic Synthesis with VHDL
Sequential Circuits

Bob Reese
Electrical Engineering Department
Mississippi State University
Logic which contains both combinational logic and storage elements form sequential circuits. All sequential circuits can be divided into a combinational block and a storage element block.

The above diagram shows a single-phase sequential system. In a single-phase system the storage elements are edge-triggered devices (flip-flops).

- Moore–type outputs are a combinatorial function of PresentState signals.
- Moore–type outputs are a combinatorial function of both PresentState and external input signals.

Multiple-phase design is also supported since latches can be synthesized as the storage elements.
Sequential Template

library declarations

entity model_name is
  port
  (    
    list of inputs and outputs
  );
  end model_name;

architecture behavior of model_name is

internal signal declarations

begin
  — the state process defines the storage elements
  state: process (sensitivity list — clock, reset, next_state inputs)
  begin
    vhdl statements for state elements
  end process state;

  — the comb process defines the combinational logic
  comb: process (sensitivity list — usually includes all inputs)
  begin
    vhdl statements which specify combinational logic
  end process comb;

end behavior;
library ieee;
use ieee.std_logic_1164.all;

entity reg8bit is port (
    signal clk, reset, load: in  std_logic;
    signal din: in  std_logic_vector(7 downto 0);
    signal dout: out  std_logic_vector(7 downto 0)
);
end reg8bit;

architecture behavior of reg8bit is
signal n_state, p_state : std_logic_vector(7 downto 0);
begin
    dout <= p_state;

    comb: process(p_state, load, din)
    begin
        n_state <= p_state;
        if (load='1') then  n_state <= din;  end if;
    end process comb;

    state: process(clk, reset)
    begin
        if (reset = '0') then  p_state <= (others => '0');
        elsif (clk'event and clk = '1') then
            p_state <= n_state;
        end if;
    end process state;

end behavior;
reg8bit State Process

state: process(clk, reset)
begin
if (reset = '0') then  p_state <= (others => '0');
elsif  (clk'event and clk = '1') then
    p_state <= n_state;
end if;
end process state;

⇒ The state process defines a storage element which is 8–bits wide, rising edge triggered, and had a low true asynchronous reset.

⇒ The output of this process is the p_state signal.

⇒ Note that the reset input has precedence over the clock in order to define the asynchronous operation.

⇒ The 'event attribute is used to detect a change in the clock signal; comparing the current clock value against ’1’ implies that p_state gets the n_state value on a 0 to 1 transition (rising edge).

⇒ The state holding action of the process arises from the fact that p_state is not assigned a value is reset is not asserted and a rising clock edge does not occur.
**wait Statement**

⇒ An alternative method of specifying the storage elements is shown below:

```vhdl
state: process
begin
wait until ((clk'event and clk = '1') or (reset = '0'));
if (reset = '0') then  p_state <= (others => '0');
else
  p_state <= n_state;
end if;
end process state;
```

⇒ The *wait* statement is a sequential statement.

⇒ The *wait* statement causes a suspension of a process or procedure until the condition clause is satisfied.

⇒ The signals used in the condition clause form an implicit sensitivity list for the *wait* statement.

  ⇒ Can use 'wait on *sig1, sig2, ..sigN* until *condition_clause*' to explicitly specify the sensitivity list.

  ⇒ Note that the process has no sensitivity list.

⇒ 'if' statements used with processes generally give more flexibility and control than 'wait' statements.
Finite State Machine Example

Upon RESET enter state HGC

Traffic Light Controller

- **HG**: HL:GRN, FL:RED
- **HGC**: HL:GRN, FL:RED
- **HY**: HL:YEL, FL:RED

Start timer with short value

- T→START_SHORT_TIMER

Start timer with long value

- T→START_LONG_TIMER

- **FG**: HL:RED, FL:GRN
- **FY**: HL:RED, FL:YEL

- ALARM + CAR
- ALARM

- T→START_SHORT_TIMER
- T→START_LONG_TIMER

- HL:GRN
- HL:RED
- HL:YEL
- HL:GRN
- HL:RED

- F
- T
- F
- T
Traffic Light Controller Block Diagram

- **Glue**
  - select
  - load
  - 0/1
    - en
    - long load
    - short load

- **TLC Control**
  - car
  - start_short_timer
  - start_long_timer
  - timer
  - fl
  - hl
  - 2
  - count

- **Timer**
  - start_A
  - start_B
  - alarm
  - count
  - Port B
  - Port A
  - N
  - N
  - count
library ieee;
use ieee.std_logic_1164.all;
— vhdl model for the Traffic Light Control, sync reset, encoded states
entity tlc_enc is port(
    signal reset, car, timer, clk: in std_logic;
    signal stateout: out std_logic_vector(2 downto 0);
    signal highway_light, farm_light: out std_logic_vector(1 downto 0);
    signal start_short_timer, start_long_timer: out std_logic);
end tlc_enc;
architecture behavior of tlc_enc is

constant HGC: std_logic_vector(2 downto 0) := "000";
constant HY: std_logic_vector(2 downto 0) := "001";
constant FG: std_logic_vector(2 downto 0) := "010";
constant FY: std_logic_vector(2 downto 0) := "011";
constant HG: std_logic_vector(2 downto 0) := "100";

constant GREEN: std_logic_vector(1 downto 0) := "00";
constant YELLOW: std_logic_vector(1 downto 0) := "01";
constant RED: std_logic_vector(1 downto 0) := "11";

signal p_state, n_state : std_logic_vector(2 downto 0);

begin
stateout <= p_state;
statereg: process(clk, reset)
    if (reset = '0') then  p_state <= HGC;
    elsif (clk'event and clk = '1') then  p_state <= n_state; end if;
end process statereg;

VHDL For Traffic Light FSM (cont)

```vhdl
comb:process(car, timer, p_state)
begin
    — default assignments — VERY IMPORTANT
    start_short_timer <= '0';
    start_long_timer <= '0';
    — by default, stay in same state!!!
    n_state <= p_state;
    highway_light <= GREEN; farm_light <= RED;

    if p_state = HG then
        highway_light <= GREEN; farm_light <= RED;
        if (timer = '1') then n_state <= HGC; end if;
    end if;

    if p_state = HGC then
        highway_light <= GREEN; farm_light <= RED;
        if car = '1' then
            n_state <= HY; start_short_timer <= '1'; end if;
    end if;

    if p_state = HY then
        highway_light <= YELLOW; farm_light <= RED;
        if timer = '1' then
            n_state <= FG; start_long_timer <= '1'; end if;
    end if;
```

All outputs should be assigned default values!! If you do not assign default values then outputs may get synthesized with output latches!

Use 'if' statements to enumerate states.

Start timer with short timeout value (yellow light).

Start timer with long timeout value.
if p_state = FG then
    highway_light <= RED;  farm_light <= GREEN;
    if timer = '1' or car = '0' then
        n_state <= FY;   start_short_timer <= '1';  end if;
    end if;
end if;

if p_state = FY then
    highway_light <= RED;  farm_light <= YELLOW;
    if timer = '1' then
        n_state <= HG;   start_long_timer <= '1';  end if;
    end if;
end if;

end process comb;

end behavior;
One–Hot Encoding for FSMs

⇒ One–Hot encoding of FSMs uses one flip–flop per state.

→ Only one flip–flop is allowed ’on’ at anytime.

→ E.G., states are "00001", "00010", "00100", "01000", "10000" for a five state FSM. All other states are illegal.

⇒ One–Hot encoding trades combinational logic for flip–flops.

→ Good for ‘flip–flop’ rich implementation technologies.

→ Because the combinational logic is reduced, the length of the critical path can be reduced resulting in a faster FSM. Speed increase is more significant for larger finite state machines.
library IEEE; use IEEE.std_logic_1164.all;

entity tlc_onehot is port (  
signal  reset, car, timer, clk: in std_logic;  
signal  stateout: out std_logic_vector(4 downto 0);  
signal  highway_light, farm_light: out std_logic_vector(1 downto 0);  
signal  start_long_timer, start_short_timer: out std_logic  
); end tlc_onehot;

architecture behavior of tlc_onehot is

constant HG: integer := 0;
constant HGC: integer := 1;
constant HY: integer := 2;
constant FG: integer := 3;
constant FY: integer := 4;

constant GREEN: std_logic_vector(1 downto 0) := "00";
constant YELLOW: std_logic_vector(1 downto 0) := "01";
constant RED: std_logic_vector(1 downto 0) := "11";

signal p_state, n_state : std_logic_vector(4 downto 0);

begin
  stateout <= p_state;

  state: process(clk, reset)
  begin
    if (reset = '0') then p_state <= (HGC => '1', others => '0');
    elsif (clk'event and clk = '1') then
      p_state <= n_state;
    end if;
  end process state;

State assignments now specify bit positions in the state FFs

Initial state is '00010'

One Hot Encoding for TLC
One Hot Encoding for TLC

```
com:process(car, timer, p_state)
begin
    — default assignments — VERY IMPORTANT
    start_long_timer <= '0'; start_short_timer <= '0'; start <= '0';
    n_state <= p_state;
    highway_light <= GREEN; farm_light <= RED;
    if p_state(HG) = '1' then
        highway_light <= GREEN; farm_light <= RED;
        if (timer = '1') then
            n_state(HG) <= '0'; n_state(HGC) <= '1';
        end if;
    end if;
    if p_state(HGC) = '1' then
        highway_light <= GREEN; farm_light <= RED;
        if car = '1' then
            n_state(HGC) <= '0'; n_state(HY) <= '1';
            start_short_timer <= '1';
        end if;
    end if;
    if p_state(HY) = '1' then
        highway_light <= YELLOW; farm_light <= RED;
        if timer = '1' then
            n_state(HY) <= '0'; n_state(FG) <= '1';
            start_long_timer <= '1';
        end if;
    end if;
When changing states you must turn off current state FF and turn on next state FF.
```
One Hot Encoding for TLC

if p_state(FG) = ’1’ then
    highway_light <= RED; farm_light <= GREEN;
    if timer = ’1’ or car = ’0’ then
        n_state(FG) <= ’0’; n_state(FY) <= ’1’;
        start_short_timer <= ’1’;
    end if;
end if;

if p_state(FY) = ’1’ then
    highway_light <= RED; farm_light <= YELLOW;
    if timer = ’1’ then
        n_state(FY) <= ’0’; n_state(HG) <= ’1’;
        start_long_timer <= ’1’;
    end if;
end if;

end process comb;

end behavior;
library IEEE; use IEEE.std_logic_1164.all;

entity shift4 is port(
    signal clk, reset: in std_logic;
    signal din:               in std_logic;
    signal dout:        out std_logic_vector(3 downto 0)
); end shift4;

architecture behavior of shift4 is
signal n_state, p_state : std_logic_vector(3 downto 0);

begin
    dout <= p_state;
    state: process(clk, reset)
    begin
        if (reset = '0') then p_state <= (others => '0');
        elsif  (clk'event and clk = '1') then
            p_state <= n_state;
        end if;
    end process state;

    comb:process (p_state,din)
    begin
        n_state(0) <= din;
        for i in 3 downto 1 loop
            n_state(i) <= p_state(i - 1);
        end loop;
    end process comb;
end behavior;
Loop function for Shift Register

```
comb: process (p_state, din)
begin
    n_state(0) <= din;
    for i in 3 downto 1 loop
        n_state(i) <= p_state(i - 1);
    end loop;
end process comb;
```

Left Shift Operation (LSB to MSB)

DIN

\[
\text{left shift}
\]

\[
\text{n_state(0)} \quad \text{n_state(1)} \quad \text{n_state(2)}
\]

\[
\text{p_state(0)} \quad \text{p_state(1)} \quad \text{p_state(2)}
\]

i=0

i=1

i=2

n_state(i) \rightarrow D

n_state(i) \rightarrow Q

p_state(i-1) \rightarrow D

p_state(i) \rightarrow Q

i’th stage
Scan Path Synthesis

The ‘for–loop’ VHDL construct can be used to create a scan–path in your design. A scan path is a design technique used for improving the testability of a design.

→ A scan path requires three extra pins on the design: ‘scan’, ‘scan_in’, and ‘scan_out’.

→ When ‘scan’ is asserted, all flip–flops in the design act like a serial shift register; the ‘scan_in’ pin is the serial input and the ‘scan_out’ pin the serial output. When ‘scan’ is negated the design functions normally.

→ Because all flip–flops in the design are on the scan path the circuit can be placed in any desired state.

→ To enter a test vector via the scan path do:

   → Assert ‘scan’.

   → Apply the test vector serially to the ‘scan_in’ input; this requires N clocks if N flip–flops are on the scan path.

   → Negate ‘scan’, clock the circuit once. This will allow the circuit to operate normally for one clock cycle; the result of the test vector will be loaded into the flip–flops.

   → Assert ‘scan’; clock N times to clock out the test vector result and to clock in the next test vector. Thus, each test vector requires N+1 clocks.
4–bit Register with Scan Path

entity scanreg4 is port (  
     signal  clk, reset_b, load:       in  std_logic;  
     signal  scan, scan_in:          in  std_logic;  
     signal  din:               in  std_logic_vector(3 downto 0);  
     signal  dout:           out  std_logic_vector(3 downto 0)  
   ); end scanreg4;

architecture behavior of scanreg4 is  
signal n_state, p_state : std_logic_vector(3 downto 0);
begin  
dout <= p_state;
state: process(clk, reset)  
begin  
if (reset = '0') then p_state <= (others => '0');  
elsif  (clk'event and clk = '1') then  
p_state <= n_state;
end if;  
end process state;

process (scan,scan_in,load,p_state,din)  
begin  
n_state <= p_state;  
if (scan = '1') then
    n_state(0) <= scan_in;
    for i in 3 downto 1 loop
        n_state(i) <= p_state(i – 1);
    end loop;
elseif (load = '1') then
    n_state <= din;
end if;
end process;
end behavior;

'scan', 'scan_in' signals

'scan_out' will be MSB of 'dout'; don't need an extra pin for 'scan_out'.

When 'scan' is asserted the scan path is active.

Register functions normally when 'scan' is negated.
Adding Scan to *tlc_onehot.vhd*

⇒ Add ‗scan‘, ‗scan_in‘ to port list. ‗scan_out‘ will be MSB of port ‗stateout‘.

```vhdl
entity tlc_onehot_scan is port (
  signal reset, car, timer, clk:        in std_logic;
  signal scan, scan_in:       in  std_logic;
  signal stateout:                           out std_logic_vector(4 downto 0);
  signal highway_light, farm_light: out std_logic_vector(1 downto 0);
  signal start_long_timer, start_short_timer:    out std_logic
); end tlc_onehot_scan;
```

⇒ Add ‗scan‘, ‗scan_in‘ to sensitivity list of process: state_machine.

```vhdl
state_machine:process(scan, scan_in, reset, car, timer, p_state)
```

⇒ Add scan path in Architecture body:

```vhdl
if (scan = ’1’) then
  n_state(0) <= scan_in;
  for i in 4 downto 1 loop
    n_state(i) <= p_state(i – 1);
  end loop;
else
  if p_state(HG) = ’1’ then
    highway_light <= GREEN; farm_light <= RED;
    .... etc...
```

Bob Reese 5/95  SeqSyn–24 Sequential Circuits
library IEEE; use IEEE.std_logic_1164.all;

entity tsreg8bit is port (  signal  clk, reset, load, en: in  std_logic;
    signal  din:               in  std_logic_vector(7 downto 0);
    signal  dout:           out  std_logic_vector(7 downto 0)
);
end tsreg8bit;

architecture behavior of tsreg8bit is
signal n_state, p_state : std_logic_vector(7 downto 0);
begin

  dout <= p_state when (en = '1')
    else "ZZZZZZZZ";

  state: process(clk, reset)
  begin
    if (reset = '0') then p_state <= (others => '0');
    elsif  (clk'event and clk = '1') then
      p_state <= n_state;
    end if;
  end process state;

  comb: process (p_state, load, din)
  begin
    n_state <= p_state;
    if (load = '1') then n_state <= din;
  end if;
  end process comb;
end behavior;

Make Z assignment to specify tri-state capability.
Mapped to ITD stdcell library because Actel ACT1 does not have tristate capability.