Sergio Garcia Tapia
Digital Design and Computer Architecture: RISC-V, by Sarah and David Harris
Chapter 4: Hardware Description Languages
February 23, 2024

The following exercises may be done using your favorite HDL. If you have a simulator available, test your design. Print the waveforms and explain how they prove that it works. If you have a synthesizer available, synthesize your code. Print the generated circuit diagram, and explain why it matches your expectations.

Exercise 4.1. Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in STD_LOGIC;
       y, z:   out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or (a and not b and c);
  z <= (a and b) or (not a and not b);
end;
```

Solution: The circuit has three inputs, A, B, C, and two outputs, Y, Z. Each output is defined by

$$Y = ABC + AB\bar{C} + A\bar{B}C$$
$$Z = AB + \bar{A}\bar{B}$$

We can simply Y to:

$$\begin{aligned} Y &= AB\bar{C} + ABC + A\bar{B}C \\ &= AB\bar{C} + ABC + ABC + A\bar{B}C \\ &= AB + AC \\ &= A(B + C) \end{aligned}$$

The sketch of the circuit is in Figure 1. The circuit synthesized by Quartus II is in Figure 2.

Exercise 4.2. Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

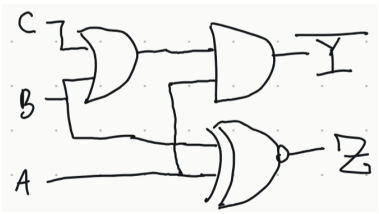

Figure 1: Schematic sketch for Exercise 4-1

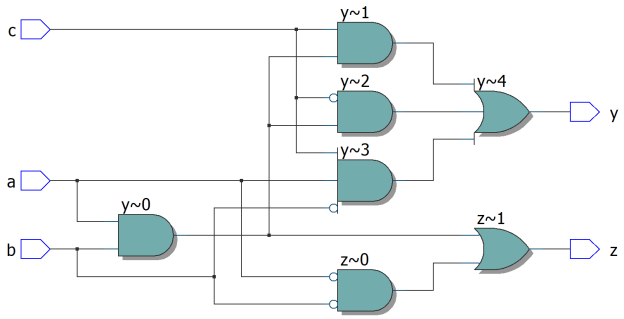

Figure 2: Synthesized Circuit by Quartus II for Exercise 4-1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(all) begin
    if     a(0) then y <= "11";
    elsif  a(1) then y <= "10";
    elsif  a(2) then y <= "01";
    elsif  a(3) then y <= "00";
    else             y <= a(1 downto 0);
    end if;
  end process;
end;
```

Solution: Note that the input is $A_{3:0}$ and the output is $Y_{1:0}$. Then $Y_1 = A_0 + A_1$ and $Y_0 = A_0 + A_2\bar{A}_1$. The sketch of the circuit is in Figure 3. The circuit synthesized by Quartus II is in Figure 2.

Exercise 4.3. Write an HDL module that computes a four-input XOR function. The input is $a_{3:0}$ and the output is y.

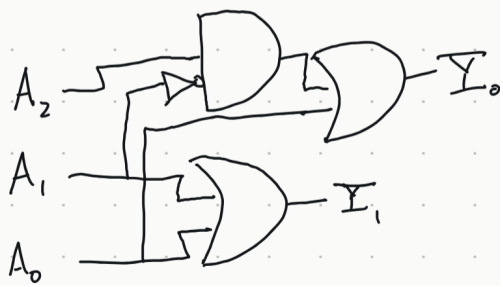

Figure 3: Schematic sketch for Exercise 4-2

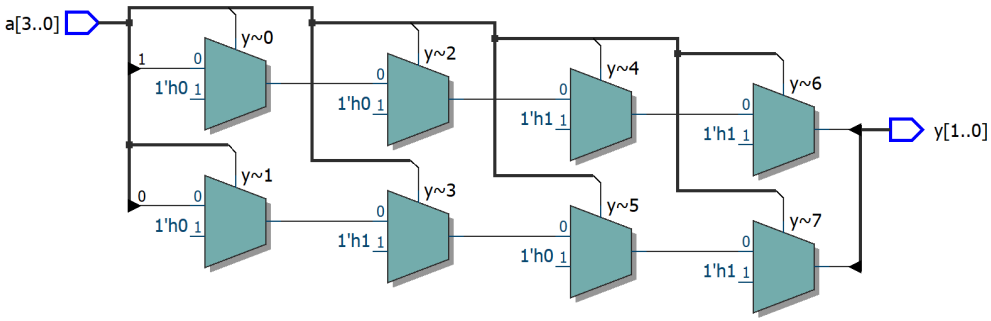

Figure 4: Synthesized Circuit by Quartus II for Exercise 4-2.

Solution: See the following listing, which can be found at `./hdl/03-04-xor4/xor4.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- Exercise 4-3: Four-input XOR
entity xor4 is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of xor4 is
begin
  y <= a(3) xor a(2) xor a(1) xor a(0);
end;
```

Exercise 4.4. Write a self-checking testbench for Exercise 4.3. Create a test vector file containing all 16 cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Solution: The 16 cases are given in `./hdl/03-04-xor4/testvectors.txt`, available in the following listing:

```
0000_0
```

```
0001_1
0010_1
0011_0
0100_1
0101_0
0110_0
0111_1
1000_1
1001_0
1010_0
1011_1
1100_0
1101_1
1110_1
1111_0
```

The self-checking testbench is available in `./hdl/03_04/tb_xor4.vhd.`, which is a light modification of HDL Example 4.39. Specifically, it changes the file path to `../../testvectors.txt`, and the component used from `sillyfunction` to `xor4`:

```
-- 4.39: testbench example 3, adapted for Exercise 4-4

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity tb_xor4 is -- no inputs or outputs
end;

architecture sim of tb_xor4 is
  component xor4
    port(a: in STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC);
  end component;
  signal a: STD_LOGIC_VECTOR(3 downto 0);
  signal y: STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- instantiate device under test
  dut: xor4 port map(a, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
```

```
process begin
  reset <= '1'; wait for 22 ns; reset <= '0';
  wait;
end process;

-- run tests
process is
  file tv: text;
  variable L: line;
  variable vector_in: std_logic_vector(3 downto 0);
  variable dummy: character;
  variable vector_out: std_logic;
  variable vectornum: integer := 0;
  variable errors: integer := 0;
begin
  FILE_OPEN(tv, "../../testvectors.txt", READ_MODE);
  while not endfile(tv) loop

    -- change vectors on rising edge
    wait until rising_edge(clk);

    -- read the next line of testvectors and split into pieces
    readline(tv, L);
    read(L, vector_in);
    read(L, dummy); -- skip over underscore
    read(L, vector_out);
    a <= vector_in(3 downto 0) after 1 ns;
    y_expected <= vector_out after 1 ns;

    -- check results on falling edge
    wait until falling_edge(clk);

      report "just loaded stuff";
    if y /= y_expected then
      report "Error: y = " & std_logic'image(y);
    errors := errors + 1;
  end if;

    vectornum := vectornum + 1;
  end loop;

  -- summarize results at end of simulation
  if (errors = 0) then
    report "NO ERRORS -- " &
        integer'image(vectornum) &
        " tests completed successfully."
      severity failure;
```

```
    else
      report integer'image(vectornum) &
           " tests completed, errors = " &
           integer'image(errors)
           severity failure;
    end if;
  end process;
end;
```

By modifying the last entry in the `./hdl/03-04-xor4/testvectors.txt` so that it says `1111_1` instead, an error is reported, since the 4-input XOR gate should output 0 if there is an even number of 1s or all 0s, but the modified vector expects a 1.

Exercise 4.5. Write an HDL module called `minority`. It receives three inputs `a`, `b`, and `c`. It produces one output, `y`, that is TRUE if at least two of the inputs are FALSE.

Solution: The truth table for the Boolean function is below:

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

From the table we are able to identify the minterms and write

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$$

Though we could reduce the equation via Karnaugh maps, we let the HDL program handle the reduction. The listing is below:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity minority is
  port(a, b, c: in STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of minority is
begin
  y <= (not a and not b and not c) or
       (not a and not b and c) or
         (not a and b and not c) or
         (a and not b and not c);
end;
```

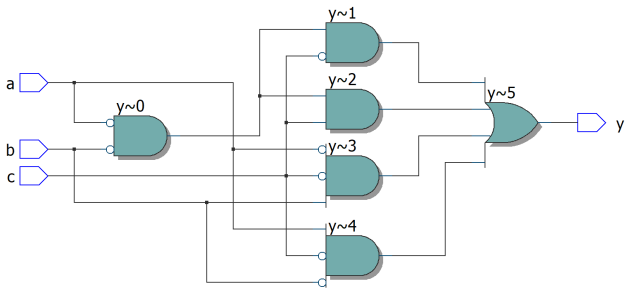

Figure 5: Synthesized circuit for `minority` module in Exercise 4.5

The synthesized circuit is given in Figure 5 The truth table also forms the basis for the test vectors used in the test bench:

```
000_1
001_1
010_1
011_0
100_1
101_0
110_0
111_0
```

Using a self-checking testbench adapted from HDL Example 4.39, the simulation is successful, so the module correctly models the boolean function.

Exercise 4.6. Write an HDL module for a hexadecimal seven-segment display decoder. The decoder should handle the digits A, B, C, D, E and F, as well as 0-9.

Solution: Figure 6 and Figure 7 show the images from Chapter 2 for this circuit. Figure 8 shows the letters used in hexadecimal using seven segments. Notice that it uses lowercase *b* and *d*, since their uppercase versions would be confused with 8 and 0, respectively. The truth table is

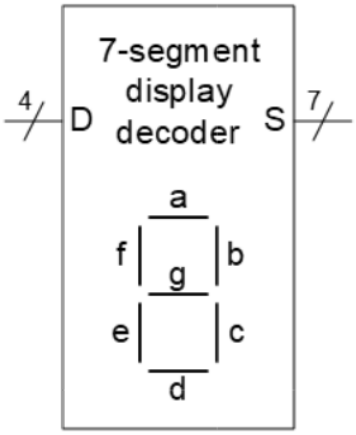

Figure 6: Seven Segment Display Decoder Circuit

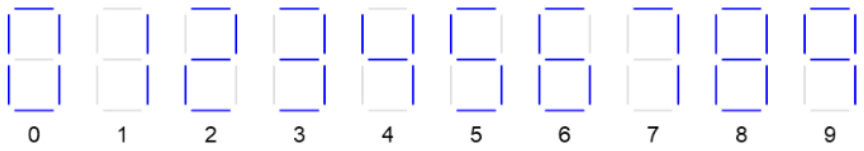

Figure 7: Seven Segment Display Digits

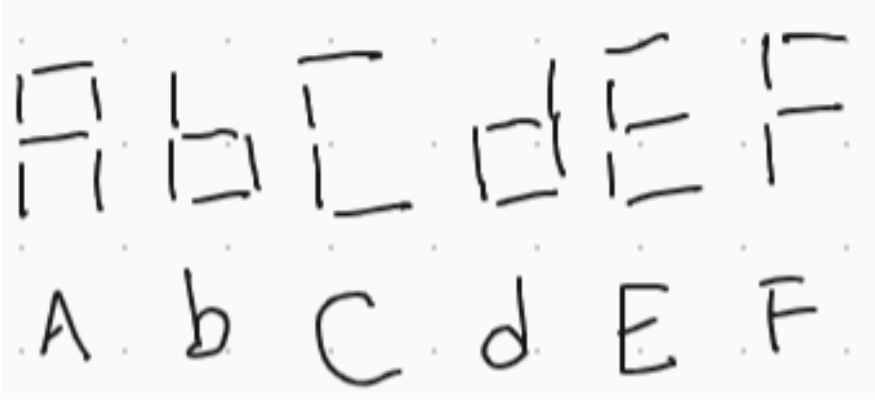

Figure 8: Seven Segment Display Hex Letters

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
1010	1	1	1	0	1	1	1
1011	0	0	1	1	1	1	1
1100	1	0	0	1	1	1	0
1101	0	1	1	1	1	0	1
1110	1	0	0	1	1	1	1
1111	1	0	0	0	1	1	1

It is not necessary to create Boolean equations and write out the minterms. The following code listing shows the implementation:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity seven_segment_hex_decoder is
  port(data:    in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_segment_hex_decoder is
begin
  process(all) begin
    case data is --           abcdefg
      when X"0"  => segments <= "1111110";
       when X"1"  => segments <= "0110000";
       when X"2"  => segments <= "1101101";
       when X"3"  => segments <= "1111001";
       when X"4"  => segments <= "0110011";
       when X"5"  => segments <= "1011011";
       when X"6"  => segments <= "1011111";
       when X"7"  => segments <= "1110000";
       when X"8"  => segments <= "1111111";
       when X"9"  => segments <= "1110011";
       when X"A"  => segments <= "1110111";
       when X"b"  => segments <= "0011111";
       when X"C"  => segments <= "1001110";
       when X"d"  => segments <= "0111101";
```

```
      when X"E"  => segments <= "1001111";
      when X"F"  => segments <= "1000111";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

Exercise 4.7. Write a self-checking testbench for Exercise 4.6. Create a test vector file containing all 16 cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Solution: We can use the following vector file, `testvectors.txt`:

```
0000_1111110
0001_0110000
0010_1101101
0011_1111001
0100_0110011
0101_1011011
0110_1011111
0111_1110000
1000_1111111
1001_1110011
1010_1110111
1011_0011111
1100_1001110
1101_0111101
1110_1001111
1111_1000111
```

The code listing below, `06-seven-segment-hex-decoder/tb_seven_segment_hex_decoder.vhd` is the testbench module:

```
-- 4.39: testbench example 3, adapted for Exercise 4-4

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity tb_seven_segment_hex_decoder is -- no inputs or outputs
end;

architecture sim of tb_seven_segment_hex_decoder is
  component seven_segment_hex_decoder
    port(data:    in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
  end component;
  signal data: STD_LOGIC_VECTOR(3 downto 0);
```

```
  signal segments: STD_LOGIC_VECTOR(6 downto 0);
  signal y_expected: STD_LOGIC_VECTOR(6 downto 0);
  signal match: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- instantiate device under test
  dut: seven_segment_hex_decoder port map(data, segments);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 22 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(3 downto 0);
    variable dummy: character;
    variable vector_out: std_logic_vector(6 downto 0);
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "../../testvectors.txt", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
      read(L, vector_out);
      data <= vector_in(3 downto 0) after 1 ns;
      y_expected <= vector_out(6 downto 0) after 1 ns;

      -- check results on falling edge
      wait until falling_edge(clk);
      if (segments /= vector_out) then
```

```
        report "Error: data = ";
          for i in 0 to data'length - 1 loop
            report std_logic'image(segments(i));
          end loop;
         errors := errors + 1;
       end if;

      vectornum := vectornum + 1;
    end loop;

    -- summarize results at end of simulation
    if (errors = 0) then
      report "NO ERRORS -- " &
          integer'image(vectornum) &
          " tests completed successfully."
        severity failure;
    else
      report integer'image(vectornum) &
          " tests completed, errors = " &
          integer'image(errors)
          severity failure;
    end if;
  end process;
end;
```

Exercise 4.8. Write an 8:1 multiplexer module called `mux8` with inputs $s_{2:0}$, `d0`, `d1`,`d2`,`d3`,`d4`,`d5`,`d6`,`d7`, and output `y`.

Solution: The following code listing implements `mux8` with generics, where the inputs and outputs are vectors of any given length.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux8 is
  generic (N: integer := 2);
  port(d0, d1, d2, d3, d4, d5, d6, d7: in STD_LOGIC_VECTOR(N-1 downto 0);
       s: in STD_LOGIC_VECTOR(2 downto 0);
         y: out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of mux8 is
begin
  process(all) begin
    case s is
       when "000" => y <= d0;
        when "001" => y <= d1;
```

```
      when "010" => y <= d2;
      when "011" => y <= d3;
      when "100" => y <= d4;
      when "101" => y <= d5;
      when "110" => y <= d6;
      when "111" => y <= d7;
      when others => y <= (others => '0');
    end case;
  end process;
end;
```

Exercise 4.9. Write a structural module to compute the logic function $Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$ using multiplexer logic. Use the 8:1 multiplexer from Exercise 4.8.

Solution: Recall, as discussed in Section 2.8.1, that any 2^N-input multiplexer can be programmed to perform any N-input multiplexer. Since Y has 3 inputs, we can use an 8-input multiplexer, which is precisely `mux8` from the previous exercise. We begin by creating the boolean table:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

It serves as the basis for the test vector file, as well as the HDL module, listed in `./hdl/09-structmodule/st`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity structmodule is
  port (a, b, c: in STD_LOGIC;
        y:       out STD_LOGIC);
end;

architecture struct of structmodule is
  component mux8
    generic(N: integer := 2);
    port(d0, d1, d2, d3, d4, d5, d6, d7: in STD_LOGIC_VECTOR(N-1 downto 0);
         s: in STD_LOGIC_VECTOR(2 downto 0);
           y: out STD_LOGIC_VECTOR(N-1 downto 0));
  end component;
  signal mux_out: STD_LOGIC_VECTOR(0 downto 0);
```

```
begin
  mux8_1: mux8 generic map(1)
              port map("1", "0", "0", "1", "1", "1", "0", "0", (a, b, c),
                  mux_out);
  y <= mux_out(0);
end;
```

Exercise 4.10. Repeat Exercise 4.9 using a 4:1 multiplexer and as many NOT gates as you need.

Solution: Section 2.8.1 mentions its possible to represent any N-input logic function with a 2^{N-1} input multiplexer. We eliminate C and express the output of $Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$ in terms of C:

A	B	Y
0	0	$\overline{C}$
0	1	C
1	0	1
1	1	0

The mux4 is given below in `./hdl/10-structmodulemux4/mux4.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  generic (N: integer := 2);
  port(d0, d1, d2, d3: in STD_LOGIC_VECTOR(N-1 downto 0);
       s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of mux4 is
begin
  process(all) begin
    case s is
       when "00"  => y <= d0;
        when "01"  => y <= d1;
        when "10"  => y <= d2;
        when "11"  => y <= d3;
        when others => y <= (others => '0');
     end case;
  end process;
end;
```

The boolean function is implemented in the module in `./hdl/10-structmodulemux4/structmodule.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity structmodule is
  port(a, b, c: in STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of structmodule is
  component mux4
    generic(N: integer := 2);
     port (d0, d1, d2, d3: in STD_LOGIC_VECTOR(N-1 downto 0);
           s:              in  STD_LOGIC_VECTOR(1 downto 0);
             y:              out STD_LOGIC_VECTOR(N-1 downto 0));
    end component;
    signal mux_out: STD_LOGIC_VECTOR(0 downto 0);
begin
  mux4_1: mux4 generic map(1)
              port map((0 => not C), (0 => C), "1", "0", (a, b), mux_out);
  y <= mux_out(0);
end;
```

Exercise 4.12. Write an HDL module for an eight-input priority circuit.

Solution: See the code listing for `./hdl/12-priority8/priority8.vhd` below:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity priority8 is
  port(a: in STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of priority8 is
begin
  process(all) begin
    case? a is
       when "1-------" => y <= (0 => '1', others => '0');
        when "01------" => y <= (1 => '0', others => '0');
        when "001-----" => y <= (2 => '0', others => '0');
        when "0001----" => y <= (3 => '0', others => '0');
        when "00001---" => y <= (4 => '0', others => '0');
        when "000001--" => y <= (5 => '0', others => '0');
        when "0000001-" => y <= (6 => '0', others => '0');
        when "00000001" => y <= (7 => '0', others => '0');
        when others     => y <= (others => '0');
```

```
    end case?;
  end process;
end;
```

Exercise 4.13. Write an HDL module for a 2:4 decoder.

Solution: A 2-input 4-output decoder chooses one of its 4 outputs according to the given input combination. See the table below:

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

See the code listing for `./hdl/13-decoder2_4/decoder2_4.vhd`.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder2_4 is
  port(a: in STD_LOGIC_VECTOR(1 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of decoder2_4 is
begin
  process(all) begin
    case a is
      when "00"  => y <= "0001";
      when "01"  => y <= "0010";
       when "10"  => y <= "0100";
       when "11"  => y <= "1000";
       when others => y <= "XXXX";
     end case;
  end process;
end;
```

Exercise 4.14. Write an HDL module for a 6:64 decoder using three instances of the 2:4 decoders from Exercise 4.13 and a bunch of three-input AND gates.

Solution: Let the input be $D_{5:0}$, and split them into three 2-bit groups: $D_{5:4}$, $D_{3:2}$, and $D_{1:0}$. Each group will be fed into a separate decoder, each producing outputs $A_{3:0}$, $B_{3:0}$, and $C_{3:0}$ as follows:

$$D_{5:4} \rightarrow A_{3:0}$$
$$D_{3:2} \rightarrow B_{3:0}$$
$$D_{1:0} \rightarrow C_{3:0}$$

Then, we'll have 64 different three-input AND gates. Letting $0 \leq i, j, k \leq 3$, we'll have AND gate for all 64 combinations of $Y_m = A_i B_j C_k$, where $m = 16i + 4j + k$. For example, if $D_{5:0}$ is all zeroes, then $D_{5:4}, D_{3:2}, D_{1:0}$ are also all zeroes, so that $A_0 = B_0 = C_0 = 1$. Since $Y_0 = A_0 B_0 C_0$, it follows that Y_0, as required. If, on the other hand, any bit is 1, then one of A_0, B_0, or C_0 will be 0, so the output Y_0 becomes 0, as required. All other cases are justified similarly.

The following code listing, `./hdl/14-decoder6_64/decoder6_64.vhd`, uses `generate` to loop and create all combinations.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder6_64 is
  port(d: in STD_LOGIC_VECTOR(5 downto 0);
       y: out STD_LOGIC_VECTOR(63 downto 0));
end;

architecture synth of decoder6_64 is
  component decoder2_4
    port(a: in STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal a: STD_LOGIC_VECTOR(3 downto 0);
  signal b: STD_LOGIC_VECTOR(3 downto 0);
  signal c: STD_LOGIC_VECTOR(3 downto 0);
begin
  dec2_4a: decoder2_4 port map(d(5 downto 4), a);
  dec2_4b: decoder2_4 port map(d(3 downto 2), b);
  dec2_4c: decoder2_4 port map(d(1 downto 0), c);
  gen: for i in 0 to 3 generate
    gen: for j in 0 to 3 generate
       gen: for k in 0 to 3 generate
          y((16 * i) + (4 * j) + k) <= a(i) and b(j) and c(k);
        end generate;
     end generate;
  end generate;
end;
```

The following code listing, `./hdl/14-decoder6_64/tb_decoder6_64.vhd` is a self-checking test bench that verifies it works

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity tb_decoder6_64 is -- no inputs
end;
```

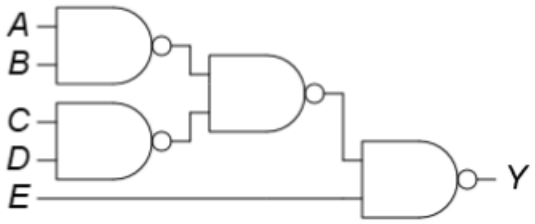

Figure 9: Circuit from Exercise 2.26 for Exercise 4.16

```
architecture sim of tb_decoder6_64 is
  component decoder6_64
    port(d: in STD_LOGIC_VECTOR(5 downto 0);
         y: out STD_LOGIC_VECTOR(63 downto 0));
    end component;
    signal d: STD_LOGIC_VECTOR(5 downto 0);
    signal y, y_expected: STD_LOGIC_VECTOR(63 downto 0);
begin
  dut: decoder6_64 port map(d, y);
  process begin
    for i in 0 to 63 loop
       y_expected <= (i => '1', others => '0');
       d <= STD_LOGIC_VECTOR(to_unsigned(i, d'length)); wait for 5 ns;
        if y /= y_expected then
          report "Assertion error";
        end if;
     end loop;
    wait; -- wait forever;
  end process;
end;
```

Exercise 4.16. Write an HDL module that implements the circuit from Exercise 2.26. See Figure 04-16-circuit.

Solution: See code listing `./hdl/16-exercise2_26/exercise2_26.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity exercise2_26 is
  port(a, b, c, d, e: in STD_LOGIC;
       y           : out STD_LOGIC);
end;

architecture synth of exercise2_26 is
begin
  y <= not (
```

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 10: Truth table for Exercise 4.18

```
        not (
             (not (a and b))
              and (not (c and d)))
        and e
        );
end;
```

Exercise 4.18. Write an HDL module that implements the logic from Exercise 2.28. Pay careful attention to how you handle don't cares. See Figure 10.

Solution: We can make it so that the input combinations that lead to don't cares yield 0. See code listing `./hdl/18-exercise2_28/exercise2_26.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;


entity exercise2_28 is
  port(v: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of exercise2_28 is
begin
  process(all) begin
    case v is
```

```
      when "1000" => y <= '1';
       when "1011" => y <= '1';
       when "1100" => y <= '1';
       when "1101" => y <= '1';
       when "1111" => y <= '1';
       when others => y <= '0';
    end case;
  end process;
end;
```

Exercise 4.19. Write an HDL module that implements the functions from Exercise 2.35.

Solution: The specification is that we have an input $A_{3:0}$, representing the numbers 0 through 15. There are two outputs: P is TRUE if the number is prime, and D is true if the number is divisible by 3. The truth table below shows the function inputs and outputs:

A_3	A_2	A_1	A_0	P	D
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	1

See code listing `./hdl/19-exercise2_35/exercise2_35.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity exercise2_35 is
  port(a:   in  STD_LOGIC_VECTOR(3 downto 0);
       p, d: out STD_LOGIC);
end;

architecture synth of exercise2_35 is
begin
  process(all) begin
```

```
    case a is
       when 4D"0" => p <= '0'; d <= '1';
        when 4D"1" => p <= '0'; d <= '0';
        when 4D"2" => p <= '1'; d <= '0';
        when 4D"3" => p <= '1'; d <= '1';
        when 4D"4" => p <= '0'; d <= '0';
        when 4D"5" => p <= '1'; d <= '0';
        when 4D"6" => p <= '0'; d <= '1';
        when 4D"7" => p <= '1'; d <= '0';
        when 4D"8" => p <= '0'; d <= '0';
        when 4D"9" => p <= '0'; d <= '1';
        when 4D"10" => p <= '0'; d <= '0';
        when 4D"11" => p <= '1'; d <= '0';
        when 4D"12" => p <= '0'; d <= '1';
        when 4D"13" => p <= '1'; d <= '0';
        when 4D"14" => p <= '0'; d <= '0';
        when 4D"15" => p <= '0'; d <= '1';
     end case;
  end process;
end;
```

Exercise 4.24. Sketch the state transition diagram for the FSM described by the following HDL code.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if (a xor b) then
                   nextstate <= S1;
```

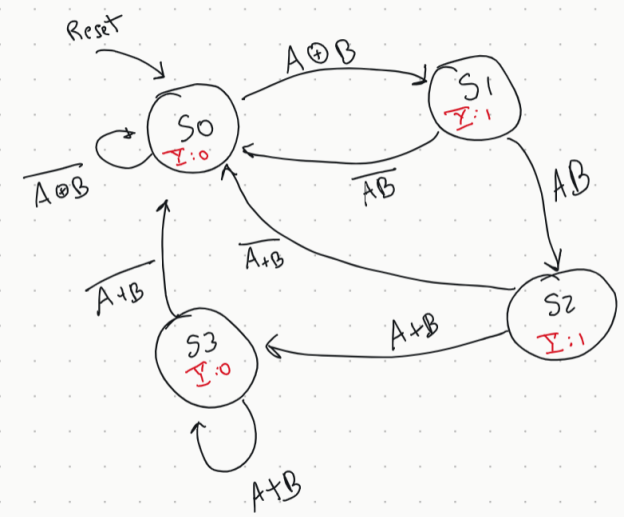

Figure 11: State Transition Diagram for Exercise 4.24

```
                else nextstate <= S0;
                end if;
      when S1 => if (a and b) then
                      nextstate <= S2;
                else nextstate <= S0;
                end if;
      when S2 => if (a or b) then
                      nextstate <= S3;
                else nextstate <= S0;
                end if;
      when S3 => if (a or b) then
                      nextstate <= S3;
                else nextstate <= S0;
                end if;
  end process;

  y <= '1' when((state = S1) or (state = S2))
       else '0';
end;
```

Solution: This is a Moore state machine. See Figure 11.

Exercise 4.25. Sketch the state transition diagram for the FSM described by the following HDL code. An FSM of this nature is used in a branch predictor on some microprocessors.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset: in STD_LOGIC;
       taken, back: in STD_LOGIC;
```

```
      predicttaken: in STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if taken then
                    nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken then
                    nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if taken then
                    nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken then
                    nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken then
                    nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others => nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                  ((state = S4) or (state = S3) or
                  (state = S2 and back = '1'))
end;
```

Solution: This is a Mealy state machine. See Figure 12.

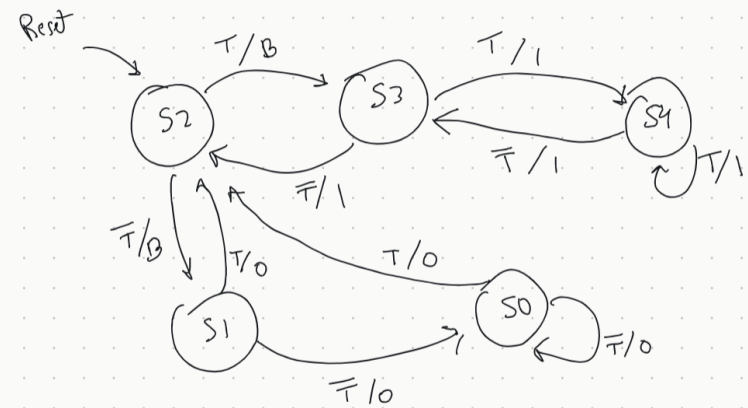

Figure 12: State Transition Diagram for Exercise 4.25

Exercise 4.26. Write an HDL module for an SR latch.

Solution: See code listing `./hdl/26-sr_latch/sr_latch.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity sr_latch is
  port(s, r: in STD_LOGIC;
       q, notq:   out STD_LOGIC);
end;

architecture synth of sr_latch is
  signal qn, notqn: STD_LOGIC;
begin
  -- output
  q <= qn;
  notq <= notqn;
  -- inner nodes
  process(all) begin
    if    (s = '0' and r = '1') then qn <= '0'; notqn <= '1';
     elsif (s = '1' and r = '0') then qn <= '1'; notqn <= '0';
     elsif (s = '1' and r = '1') then qn <= '0'; notqn <= '0';
     end if;
  end process;
end;
```

Exercise 4.27. Write an HDL module for a *JK flip-flop*. The flip-flop has inputs CLK, J, and K, and output Q. On the rising edge of the clock, Q keeps its old value if $J = K = 0$. It sets Q to 1 if $J = 1$, resets Q to 0 if $K = 1$, and inverts Q if $J = K = 1$.

Solution: See code listing `./hdl/27-jk_ff/jk_ff.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity jk_ff is
  port(j, k: in  STD_LOGIC;
       clk:  in  STD_LOGIC;
         q:     out STD_LOGIC);
end;

architecture synth of jk_ff is
begin
  process(clk) begin
    if rising_edge(clk) then
       if    j = '1' and k = '1' then q <= not q;
        elsif j = '1' and k = '0' then q <= '1';
        elsif j = '0' and k = '1' then q <= '0';
        end if;
     end if;
  end process;
end;
```

Exercise 4.29. Write an HDL module for the traffic light controller from Section 3.4.1.

Solution: See code listing `./hdl/29-traffic_light/traffic_light.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity traffic_light is
  port(ta, tb:   in  STD_LOGIC;
       clk, reset: in STD_LOGIC;
       la, lb:    out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of traffic_light is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
        state <= nextstate;
     end if;
  end process;

  -- green: 00, yellow: 01, red: 10
  process(all) begin
```

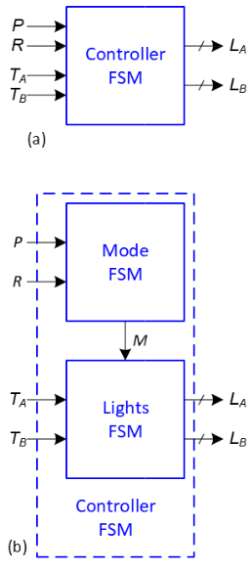

Figure 13: Factored Traffic Light State Machine Design for Exercise 4.30

```
    case state is
       when S0 => if ta then nextstate <= S0;
                  else       nextstate <= S1;
                     end if;
        when S1 => nextstate <= S2;
        when S2 => if tb then nextstate <= S2;
                  else       nextstate <= S3;
                     end if;
        when S3 => nextstate <= S0;
        when others => nextstate <= S0;
     end case;
  end process;

  process(all) begin
    case state is
         when S0 => la <= "00"; lb <= "10";
         when S1 => la <= "01"; lb <= "00";
         when S2 => la <= "10"; lb <= "00";
         when S3 => la <= "10"; lb <= "01";
     end case;
  end process;
end;
```

Exercise 4.30. Write HDL modules for the factored parade mode traffic light controller from Example 3.8. The modules should be called `controller`, `mode`, and `lights`, and they should have the inputs and outputs shown in Figure 13. See the state transition diagram in Figure 14.

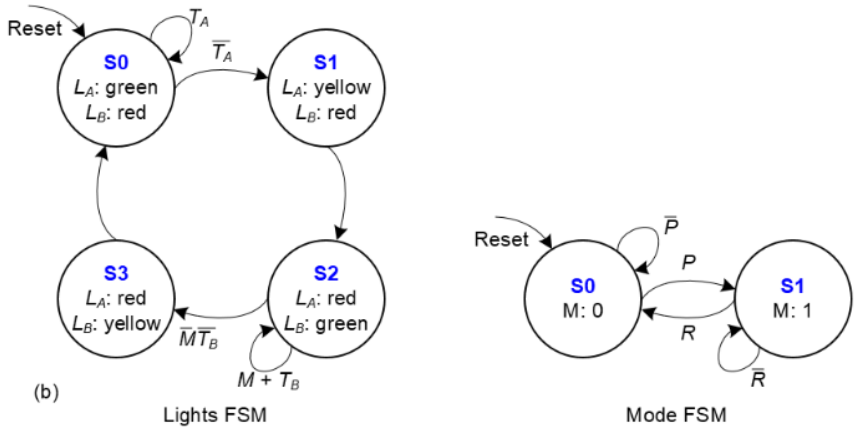

Figure 14: Factored Traffic Light State Transition Diagram for Exercise 4.30

Solution: The `mode` module is in listing `./hdl/30-traffic_fsm_factored/mode.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mode is
  port(p, r:      in  STD_LOGIC;
       clk, reset: in STD_LOGIC;
       m:          out STD_LOGIC);
end;

architecture synth of mode is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
       state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
       when S0 => if p then nextstate <= S1;
                  else nextstate <= S0;
                     end if;
       when S1 => if r then nextstate <= S0;
                  else nextstate <= S1;
                     end if;
    end case;
  end process;
```

```
  m <= '1' when (state = S1) else '0';
end;
```

:

The `lights` module is in listing `./hdl/30-traffic_fsm_factored/lights.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity lights is
    port(ta, tb, m: in STD_LOGIC;
         clk, reset: in STD_LOGIC;
           la, lb:    out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of lights is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
       state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
       when S0 => if ta then nextstate <= S0;
                  else      nextstate <= S1;
                     end if;
       when S1 => nextstate <= S2;
       when S2 => if (m or tb) then nextstate <= S2;
                  else              nextstate <= S3;
                    end if;
       when S3 => nextstate <= S0;
    end case;
  end process;

  process(all) begin
    case state is -- green="00", yellow="01", red="10"
       when S0 => la <= "00"; lb <= "10";
       when S1 => la <= "01"; lb <= "10";
       when S2 => la <= "10"; lb <= "00";
       when S3 => la <= "10"; lb <= "01";
    end case;
  end process;
```

```
end;
```

:

The `controller` module is in listing `./hdl/30-traffic_fsm_factored/controller.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity controller is
  port(p, r, ta, tb: in STD_LOGIC;
       clk, reset:   in  STD_LOGIC;
       la, lb:       out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of controller is
  component lights
    port(ta, tb, m: in STD_LOGIC;
         clk, reset: in STD_LOGIC;
           la, lb:    out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component mode
    port(p, r:      in  STD_LOGIC;
         clk, reset: in STD_LOGIC;
         m:         out STD_LOGIC);
  end component;
  signal m: STD_LOGIC;
begin
  mode_fsm: mode port map(p, r, clk, reset, m);
  lights_fsm: lights port map(ta, tb, clk, reset, m);
end;
```

:

Exercise 4.31. Write an HDL describing the circuit in Figure **??**.

Solution: See code listing `./hdl/31-exercise31/exercise31.vhd`:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity exercise31 is
  port(a, b, c, d: in STD_LOGIC;
       clk:       in  STD_LOGIC;
         x, y:      out STD_LOGIC);
end;

architecture synth of exercise31 is
  signal aa, bb, cc, dd, n2: STD_LOGIC;
begin
```

```
  process(clk) begin
    if rising_edge(clk) then
       aa <= a;
        bb <= b;
        cc <= c;
        dd <= d;
        x <= n2;
        y <= not (n2 or dd);
     end if;
  end process;
  n2 <= (aa and bb) or cc;
end;
```