Basic example

We will use the std_ulogic and std_ulogic_vector types defined in the ieee.std_logic_1164 package. You have access to basic logic operators.

library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port(a, b, cin: in std_ulogic;
       sum, cout: out std_ulogic);
end full_adder;
architecture dataflow of full_adder is
begin
  sum <= (a xor b) xor c;
  carry <= (a and b) or (a and c) or (b and c);
end dataflow;

Other concurrent VHDL statements

when . . . else is a useful concurrent VHDL statement.

library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_4_1 is
  port(in0, in1, in2, in3: in std_ulogic_vector(0 to 15);
       s0, s1: in std_ulogic;
       z: out std_ulogic_vector(0 to 15));
end multiplexer_4_1;
architecture dataflow of multiplexer_4_1 is
begin
  z <= in0 when (s0 = '0' and s1 = '0') else
       in1 when (s0 = '1' and s1 = '0') else
       in2 when (s0 = '0' and s1 = '1') else
       in3 when (s0 = '1' and s1 = '1') else
     "xxxxxxxxxxxxxxxxxx";
end dataflow;
Here's another way of doing the multiplexer with . . . select is also useful.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_4_1 is
  port(in0, in1, in2, in3: in std_logic_vector(0 to 15);
       s0, s1: in std_logic;
       z: out std_logic_vector(0 to 15));
end multiplexer_4_1;
architecture dataflow of multiplexer_4_1 is
signal sels: std_logic_vector(0 to 1);
begin
  sels <= s0 & s1;
  with sels select
    z <= in0 when "00",
        in1 when "01",
        in2 when "10",
        in3 when "11",
        "XXXXXXXXXXXXXXXX" when others;
end dataflow;
```

But maybe the selects cannot both be 1 or 0

Use can use VHDL assert statements to flag invalid conditions.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_2_1 is
  port(in0, in1: in std_logic_vector(0 to 15);
       s0, s1: in std_logic;
       z: out std_logic_vector(0 to 15));
end multiplexer_2_1;
architecture dataflow of multiplexer_2_1 is
signal sels: std_logic_vector(0 to 1);
begin
  sels <= s0 & s1;
  with sels select
    z <= in0 when "00",
        in1 when "01",
        in2 when "10",
        in3 when "11",
        "XXXXXXXXXXXXXXXX" when others;
end dataflow;
```
\[
\begin{align*}
z & \leq \text{in0 when "10"}, \\
& \quad \text{in1 when "01"}, \\
& \quad "\text{XXXXXXXXXXXXXXX}" \text{ when others}; \\
\text{assert not}(\text{sels = "00" or sels = "11"}) \\
\text{report } \text{"The select signals must be orthogonal!"} \\
\text{severity ERROR;} \\
\end{align*}
\]

Tristate drivers

Note the use of the resolved \texttt{std_logic} type on signal \texttt{z}.

\begin{verbatim}
library ieee;
use ieee.std_logic_1164.all;
entity part1 is
  port(a, sel: in std_ulogic;
       z: out std_logic);
end part1;
architecture dataflow of part1 is
begin
  z <= a when (sel = '1') else 'Z';
end dataflow;
\end{verbatim}

4-bit adder with concurrent VHDL

\begin{verbatim}
library ieee;
use ieee.std_logic_1164.all;
entity adder is
  port(a, b: in std_ulogic_vector(0 to 3);
       cin: in std_ulogic;
       sum: out std_ulogic_vector(0 to 3);
       cout: out std_ulogic);
end adder;
architecture dataflow of adder is
signal car: std_ulogic_vector(0 to 4);
begin
  car(0) <= cin;
  G1: for m in 3 downto 0 generate
\end{verbatim}
sum(m) <= a(m) xor b(m) xor car(m);
car(m+1) <= (a(m) and b(m)) or (b(m) and car(m)) or (a(m) and car(m));
end generate G1;
cout <= car(4);
end dataflow;

16-bit carry-lookahead adder
Coding at this level of detail ensures that the adder is logically structured exactly the way you want it.

library ieee;
use ieee.std_logic_1164.all;
entity adder_16 is
  port(a, b: in std_ulogic_vector(0 to 15);
    sum: out std_ulogic_vector(0 to 15);
    cout: out std_ulogic);
end adder_16;
architecture dataflow of adder_16 is
signal car: std_ulogic_vector(0 to 16);
signal pg, gg: std_ulogic_vector(0 to 3);
beg
-- full adder blocks
G1: for m in 0 to 15 generate
  sum(m) <= p(m) xor g(m) xor car(m);
  g(m) <= a(m) and b(m);
  p(m) <= a(m) or b(m);
end generate G1;
G2: for m in 0 to 3 generate
  gg(m) <= g(4*m+3) or (p(4*m+3) and g(4*m+2)) or
           (p(4*m+3) and p(4*m+2) and g(4*m+1)) or
           (p(4*m+3) and p(4*m+2) and p(4*m+1) and g(4*m));
  pg(m) <= p(4*m+3) and p(4*m+2) and p(4*m+1) and p(4*m));
  car(4*m+1) <= g(4*m) or (p(4*m) and car(4*m));
  car(4*m+2) <= g(4*m+1) or (p(4*m+1) and g(4*m)) or
                 (p(4*m+1) and p(4*m) and car(4*m));
  car(4*m+3) <= g(4*m+2) or (p(4*m+2) and g(4*m+1)) or
                 (p(4*m+2) and p(4*m+1) and g(4*m)) or


(p(4*m+2) and p(4*m+1) and p(4*m) and car(4*m));

end generate G2;
car(0) <= cin;
car(4) <= gg(0) or (pg(0) and car(0));
car(8) <= gg(1) or (pg(1) and gg(0)) or
        (pg(1) and pg(0) and car(0));
car(12) <= gg(2) or (pg(2) and gg(1)) or
        (pg(2) and pg(1) and gg(0)) or
        (pg(2) and pg(1) and pg(0) and car(0));
car(16) <= gg(3) or (pg(3) and gg(2)) or
        (pg(3) and pg(2) and gg(1)) or
        (pg(3) and pg(2) and pg(1) and gg(0)) or
        (pg(3) and pg(2) and pg(1) and pg(0) and car(0));
cout <= car(16);
end dataflow;

4-bit adder using the std_logic_arith package

In this case, we use a “higher-level” arithmetic operator. In general, you must be very careful when you do this because you lose control over the detailed logic implementation of the adder (in this case). The synthesis tool will implement this a certain way (for example, a ripple-carry adder) which might not be acceptable for a particular application.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_
use ieee.std_logic_arith.all;
entity adder is
    port(a, b: in std_ulogic_vector(0 to 3);
        cin: in std_ulogic;
        c: out std_ulogic_vector(0 to 3);
        cout: out std_ulogic);
end adder;
architecture dataflow of adder is
signal temp_sum: std_ulogic_vector(0 to 4);
begin
    temp_sum <= to_stdulogicvector(unsigned(a) + unsigned(b))
+ convUnsigned(cin,1));
cout <= temp_sum(0);
c <= temp_sum(1 to 4);
end dataflow;

Positive edge-trigger 32-bit register

VHDL process statements are the preferred mechanism for defining flip-flops and registers.

library ieee;
use ieee.std_logic_1164.all;
entity register_32 is
  port(clk, write_enable: in std_ulogic;
       data_in: in std_logic_vector(0 to 31);
       data_out: out std_logic_vector(0 to 31));
end register_32;
architecture dataflow of register_32 is
begin
  process(clk)
  begin
    if ((clk = '1') and not(clk'stable) and (write_enable = '1')) then
      data_out <= data_in;
    end if;
  end process;
end dataflow;

8-bit right shifter

Note the use of the concatenation operator & to form the shifter.

library ieee;
use ieee.std_logic_1164.all;
entity eight_bit_shifter is
  port(shift_in: in std_logic_vector(0 to 7);
       shift_out: out std_logic_vector(0 to 7);
       shift_control: in std_logic_vector(0 to 2));
end eight_bit_shifter;
architecture dataflow of eight_bit_shifter is
signal first_stage: std_ulogic_vector(0 to 7);
signal second_stage: std_ulogic_vector(0 to 7);
begin
  first_stage <= shift_in when (shift_control(0) = '0') else
                   ("0" & shift_in(0 to 6));
  second_stage <= first_stage when (shift_control(0) = '0') else
                   ("00" & first_stage(0 to 5));
  shift_out <= second_stage when (shift_control(2) = '0') else
                   ("0000" & second_stage(0 to 4));
end dataflow;

Dataflow example. Counter in traffic light problem.
Note how this matches precisely the “block-diagram” structure of the
design that we discussed in class. VHDL should be used to capture logic
design; it should not be used as a programming language.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY counter IS
  PORT(
    long : OUT std_ulogic;
    short : OUT std_ulogic;
    start_timer : IN std_ulogic;
    clock : IN std_ulogic
  );
END counter;
architecture dataflow of counter is
signal latch_in, latch_out, incr_out: std_ulogic_vector(0 to 6);
signal short_int, long_int, short_temp, long_temp: std_ulogic;
beginsync: process(clock)
  begin
    if (clock = '1' and not(clock’stable)) then
      latch_out <= latch_in;
      short_temp <= not(start_timer) and (short_int or short_temp);
end process;
long_temp <= not(start_timer) and (long_int or long_temp);
end if;
end process;
latch_in <= "0000000" when (start_timer = '1') else incr_out;
incr_out <= to_stdlogicvector(unsigned(latch_out) + '1');
short_int <= '1' when (incr_out = "0100000") else '0';
long_int <= '1' when (incr_out = "1000000") else '0';

short <= short_temp;
long <= long_temp;
end dataflow;

ASM translation to VHDL. Traffic light controller.
ASM’s can be translated to VHDL in a completely formulaic manner.
One process is used for the state register. Another process can be used for
the combinational logic of the state machine. This second process must be
combinational. To ensure this, two conditions must be satisfied:

- The process must be activated by all inputs of the combinational logic
block.

- All “cases” must be covered in the logic.

Note the use of VHDL constants to define the states. This makes the
code easier to read and makes the state encodings easy to change.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY controller IS
  PORT(
    h11 : OUT std_ulogic;
    h10 : OUT std_ulogic;
    f11 : OUT std_ulogic;
    f10 : OUT std_ulogic;
    start_timer : OUT std_ulogic;
    clock : IN std_ulogic;
  );
reset : IN std_ulogic;
cars : IN std_ulogic;
short : IN std_ulogic;
long : IN std_ulogic
);
END controller;
architecture behavior of controller is

signal current_state, next_state: std_ulogic_vector(0 to 1);
constant HG: std_ulogic_vector := "00";
constant HY: std_ulogic_vector := "01";
constant FY: std_ulogic_vector := "10";
constant FG: std_ulogic_vector := "11";

begin
  
  sync: process(clock)
  begin
    if ((clock = '1') and not(clock'stable)) then
      current_state <= next_state;
    end if;
  end process sync;

  fsm_comb: process(current_state, short, long, cars)
  begin
    if (reset = '1') then -- synchronous reset
      next_state <= HG;
      start_timer <= '1';
    else
      case current_state is
        when HG =>
          -- set lights
          h11 <= '0';
          h10 <= '0';
          f11 <= '0';
          f10 <= '1';
          if (cars and long) = '1' then

        when HY =>
          -- set lights
          h11 <= '0';
          h10 <= '0';
          f11 <= '0';
          f10 <= '0';

        when FY =>
          -- set lights
          h11 <= '0';
          h10 <= '0';
          f11 <= '0';
          f10 <= '1';

        when FG =>
          -- set lights
          h11 <= '0';
          h10 <= '0';
          f11 <= '0';
          f10 <= '1';

        end case;
      end if;
  end process fsm_comb;

end behavior;
next_state <= HY;
    start_timer <= '1';
else
    next_state <= HG;
    start_timer <= '0';
end if;
when HY =>
    -- set lights
    h1 <= '1';
    h0 <= '0';
    f1 <= '0';
    f0 <= '1';
    if short = '1' then
        next_state <= FG;
        start_timer <= '1';
    else
        next_state <= HY;
        start_timer <= '0';
    end if;
when FG =>
    h1 <= '0';
    h0 <= '1';
    f1 <= '0';
    f0 <= '0';
    if (not(cars) or long) = '1' then
        next_state <= FY;
        start_timer <= '1';
    else
        next_state <= FG;
        start_timer <= '0';
    end if;
when FY =>
    h1 <= '0';
    h0 <= '1';
    f1 <= '1';
    f0 <= '0';
    if (short = '1') then
next_state <= HG;
    start_timer <= '1';
else
    next_state <= FY;
    start_timer <= '0';
end if;
when others =>
    next_state <= "XX";
    start_timer <= 'X';
h10 <= 'X';
h11 <= 'X';
f10 <= 'X';
f11 <= 'X';
end case;
end if;
end process;
end behavior;

You can make a latch you don’t intend if you aren’t careful with processes

In this case, I violated the second rule for a “combinational” process and
did not cover all cases.

library ieee;
use ieee.std_logic_1164.all;
entity is_latch is
    port(a: in std_ulogic;
         b: out std_ulogic);
end is_latch;
architecture dataflow of is_latch is
begin
    process(a)
    begin
        if (a = '0') then
            b <= '1';
        end if;
    end process;
end dataflow;
Modelling an SRAM. Use of variables.

Memory modelling is really the only legitimate use of VHDL variables in RTL design.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity sram_2168 is
  port( io: inout std_logic_vector(0 to 3);
        addr: in std_ulogic_vector(0 to 11);
        ce_n: in std_ulogic;
        we_n: in std_ulogic);
end sram_2168;
architecture dataflow of sram_2168 is
begin
  memory: process(addr, ce_n, we_n)
  type sram_array_word is std_ulogic_vector(0 to 3);
  variable sram_array: array(0 to 4096) of sram_array_word;
  begin
    if (ce_n = '0') then
      if (we_n = '1') then -- read the memory
        io <= sram_array(to_integer(addr));
      else
        sram_array(to_integer(addr)) := io;
      end if;
    else
      io <= "ZZZZ";
    end if;
  end process;
end dataflow;
```

There is a difference between signals and variables in processes

These examples indicate the subtle difference between signals and variables in processes.

```vhdl
entity what_is_b is
end what_is_b;
architecture dataflow of what_is_b is
```
signal a, b: std_ulogic;
begin
process
begin
a <= '1';
if (a = '1') then
b <= '1';
else
b <= '0';
end if;
end process;
end dataflow;

entity what_is_b is
end what_is_b;
architecture dataflow of what_is_b is
signal b: std_ulogic;
begin
process
variable a: std_ulogic;
begin
a := '1';
if (a = '1') then
b <= '1';
else
b <= '0';
end if;
end process;
end dataflow;

Testbench example
Testbenches are how you supply a stimulus to your design.1

ENTITY test IS
END test;
LIBRARY ieee, light;
USE ieee.std_logic_1164.all;
ARCHITECTURE stimulus OF test IS

COMPONENT traffic_light

PORT(
    h10 : OUT std_ulogic;
    h11 : OUT std_ulogic;
    f10 : OUT std_ulogic;
    f11 : OUT std_ulogic;
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    cars : IN std_ulogic
);
END COMPONENT;

-- Fill in values for each generic

-- Fill in values for each signal
SIGNAL h10 : std_ulogic;
SIGNAL h11 : std_ulogic;
SIGNAL f10 : std_ulogic;
SIGNAL f11 : std_ulogic;
SIGNAL clock : std_ulogic := '0';
SIGNAL reset : std_ulogic := '1';
SIGNAL cars : std_ulogic;

FOR ALL: traffic_light USE ENTITY light.traffic_light(schematic);

BEGIN

dut : traffic_light

    PORT MAP (h10, h11, f10, f11, clock, reset, cars);

reset <= '1', '0' after 1 ms;
clock <= not(clock) after 50 ms;
cars <= '0', '1' after 200 ms, '0' after 700 ms,
    '1' after 1200 ms, '0' after 7200 ms, '1' after 7300 ms,
'0' after 17300 ms;
END stimulus;

Testbench example using a process for waveform generation

ENTITY test IS
END test;
LIBRARY ieee, light;
USE ieee.std_logic_1164.all;
ARCHITECTURE stimulus OF test IS

COMPONENT traffic_light
  PORT(
    h10 : OUT std_ulogic;
    h11 : OUT std_ulogic;
    f10 : OUT std_ulogic;
    f11 : OUT std_ulogic;
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    cars : IN std_ulogic
  );
END COMPONENT;

-- Fill in values for each generic

-- Fill in values for each signal
SIGNAL h10 : std_ulogic;
SIGNAL h11 : std_ulogic;
SIGNAL f10 : std_ulogic;
SIGNAL f11 : std_ulogic;
SIGNAL clock : std_ulogic := '0';
SIGNAL reset : std_ulogic := '1';
SIGNAL cars : std_ulogic;

FOR ALL: traffic_light USE ENTITY light.traffic_light(schematic);

BEGIN
dut : traffic_light

    PORT MAP (h10, h11, f10, f11, clock, reset, cars);

reset <= '1', '0' after 1 ms;
clock <= not(clock) after 50 ms;

process
begin
    cars <= '0';

    wait for 200 ms;
cars <= '1';

    wait for 500 ms;
cars <= '0';

    wait for 500 ms;
cars <= '1';

    wait for 5000 ms;
cars <= '0';

    wait for 100 ms;
cars <= '1';

    wait for 10000 ms;
cars <= '0';
end process;

END stimulus;