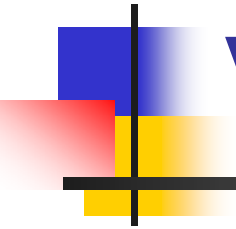
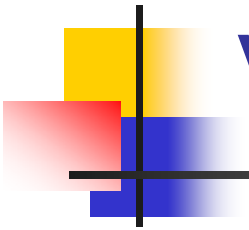


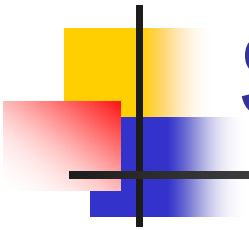
VHDL in 1h

Martin Schöberl



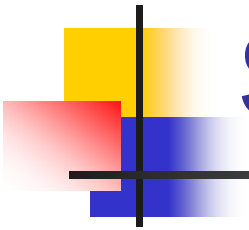
VHDL /= C, Java,...

- Think in hardware
  - All constructs run concurrent
- Different from software programming
- Forget the simulation explanation
- VHDL is complex
  - We use only a small subset



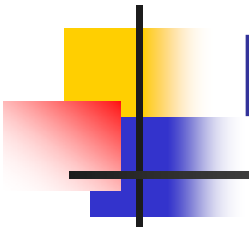
Signals are Wires

- Use
  - `std_logic`
  - `std_logic_vector (n downto 0)`
  - `unsigned`
- Variables
  - Elegant for some constructs
  - Easy to produce too much logic
  - Avoid them



Synchronous Design

- Register
  - Clock, reset
- Combinatorial logic
  - And, or,...
  - +, -
  - =, /=, >,...



Register

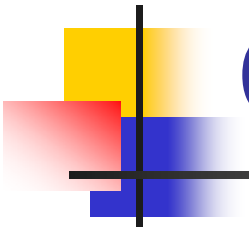
- Changes the value on the clock edge

```
process(clk)
begin

    if rising_edge(clk) then
        reg <= data;
    end if;

end process;
```

Register with Reset

- Usually has an asynchronous reset

```
process(clk, reset)
begin

    if (reset='1') then
        reg <= "00000000";
    elsif rising_edge(clk) then
        reg <= data;
    end if;

end process;
```



Combinational Logic

- Simple expressions as signal assignment
- Concurrent statement

```
a <= b;
c <= a and b;
```

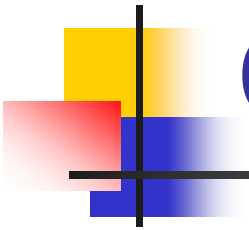
Combinational Logic

- Complex expressions in a process
- Sequential statement
- Input signals in the sensitivity list
- Assign a value to the output for each case

```
process(a, b)
begin

    if a=b then
        equal <= '1';
        gt <= '0';
    else
        equal <= '0';
        if a>b then
            gt <= '1';
        else
            gt <= '0';
        end if;
    end if;

end process;
```

Defaults for Combinational

```
process(a, b)
begin

    equal <= '0';
    gt <= '0';

    if a=b then
        equal <= '1';
    end if;
    if a>b then
        gt <= '1';
    end if;

end process;
```



Constants

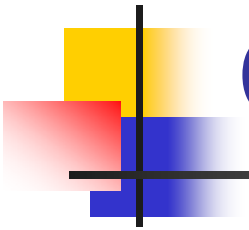
- Single bit `'0'` and `'1'`
  - Use for `std_logic`
- Bit arrays `"0011"`
  - Use for `std_logic_vector` and `unsigned`
- Integer: 1, 2, 3
  - Use for `integer`, `unsigned`

Example: Counter

```
    signal reg      : std_logic_vector(7 downto 0);
    signal count    : std_logic_vector(7 downto 0);

begin

    -- the adder process
    process(reg) begin

        count <= std_logic_vector(unsigned(reg) + 1);

    end process;

    -- the register process
    process(clk, reset) begin

        if reset='1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            reg <= count;
        end if;

    end process;

    -- assign the counter to the out port
    dout <= reg;
```

Counter in a Single Process

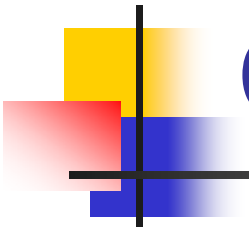
```
    -- we now use unsigned for the counter
    signal reg      : unsigned(7 downto 0);

begin

    --  a single process:
    process(clk, reset) begin

        if reset='1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            reg <= reg + 1;
        end if;

    end process;

    -- assign the counter to the out port
    dout <= std_logic_vector(reg);
```



Quiz 1

```
process(a, b, sel) begin

   if sel='0' then
       data <= a;
   else
       data <= b;
   end if;

end process;
```

- A 2:1 Multiplexer



Quiz 2

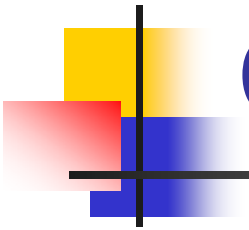
```
process(sel, a, b, c, d) begin

    case sel is
        when "00" =>
            data <= a;
        when "01" =>
            data <= b;
        when "10" =>
            data <= c;
        when others =>
            data <= d;
    end case;

end process;
```

- 4:1 Multiplexer (Mux)

Quiz 3

```
process(sel) begin

    case sel is
        when "00" =>
            z <= "0001";
        when "01" =>
            z <= "0010";
        when "10" =>
            z <= "0100";
        when "11" =>
            z <= "1000";
        when others =>
            z <= "XXXX";
    end case;

end process;
```

- 2 to 4 decoder



Quiz 4

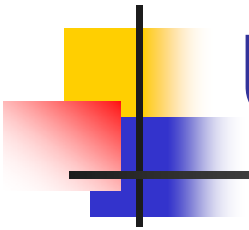
```
process(clk, reset) begin

    if reset='1' then
        reg <= (others => '0');
    elsif rising_edge(clk) then
        if en='1' then
            reg <= data;
        end if;
    end if;

end process;
```

- Register with enable

Quiz 5

```
process(data, en) begin

    if en='1' then
        reg <= data;
    end if;

end process;
```

- A Latch!
- VERY bad...



User defined types

- State machine states
- Operation type

```
    type rgb_type   is (red, green, blue);
    signal color    : rgb_type;

begin

    color <= red;
```

A simple ALU

```
    type op_type   is (op_add, op_sub, op_or, op_and);
    signal op      : op_type;

begin

process(op, a, b) begin

    case op is
        when op_add =>
            result <= a + b;
        when op_sub =>
            result <= a - b;
        when op_or =>
            result <= a or b;
        when others =>
            result <= a and b;
    end case;

end process;
```

File structure

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu is
port (
    clk, reset   : in std_logic;
    a_in, b_in   : in std_logic_vector(7 downto 0);
    dout         : out std_logic_vector(7 downto 0)
);
end alu;

architecture rtl of alu is
    signal a, b  : unsigned(7 downto 0);
    ...
begin
    a <= unsigned(a_in);
    dout <= std_logic_vector(result);

    process(op, a, b) begin
        ...
    end process;
end rtl;
```

Adder as Component

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity add is
port (
    a, b    : in std_logic_vector(7 downto 0);
    dout    : out std_logic_vector(7 downto 0)
);
end add;

architecture rtl of add is

    signal result   : unsigned(7 downto 0);
    signal au, bu   : unsigned(7 downto 0);

begin
    au <= unsigned(a);
    bu <= unsigned(b);
    result <= au + bu;
    dout <= std_logic_vector(result);
end rtl;
```

Component Use

- Declaration

```
component add is
    port (
        a    : in std_logic_vector(7 downto 0);
        b    : in std_logic_vector(7 downto 0);
        dout : out std_logic_vector(7 downto 0)
    );
end component add;
```

- Instantiation

```
cmp_add: add port map (a_in, b_in, sum);
```

Component

```
library ieee;
…

entity alu is
    …
end alu;

architecture rtl of alu is
    component add is
        port (
            a, b    : in std_logic_vector(7 downto 0);
            dout    : out std_logic_vector(7 downto 0)
        );
    end component add;

    signal a, b  : unsigned(7 downto 0);
    signal sum   : std_logic_vector(7 downto 0);

begin
    a <= unsigned(a_in);
    cmp_add : add port map (a_in, b_in, sum);
    ...
end rtl;
```

State Machine

```
architecture rtl of sm1 is

    type state_type     is (green, orange, red);
    signal state_reg    : state_type;
    signal next_state   : state_type;

begin

-- state register
process(clk, reset)
begin

    if reset='1' then
        state_reg <= green;
    elsif rising_edge(clk) then
        state_reg <= next_state;
    end if;

end process;

-- output of state machine
process(state_reg) begin

    if state_reg=red then
        ring_bell <= '1';
    else
        ring_bell <= '0';
    end if;

end process;
```

```
-- next state logic
process(state_reg, bad_event, clear) begin

    next_state <= state_reg;

    case state_reg is

        when green =>
            if bad_event = '1' then
                next_state <= orange;
            end if;
        when orange =>
            if bad_event = '1' then
                next_state <= red;
            end if;
        when red =>
            if clear = '1' then
                next_state <= green;
            end if;
    end case;

end process;

end rtl;
```

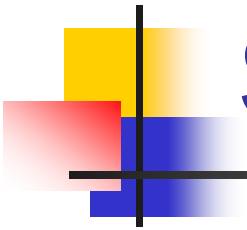
State Machine

```
architecture rtl of sm2 is

    type state_type     is (green, orange,
                            red);
    signal state_reg    : state_type;
    signal next_state   : state_type;

begin

-- state register
process(clk, reset)
begin

    if reset='1' then
        state_reg <= green;
    elsif rising_edge(clk) then
        state_reg <= next_state;
    end if;

end process;
```

```
-- next state logic and output
process(state_reg, bad_event, clear) begin

    next_state <= state_reg;
    ring_bell <= '0';

    case state_reg is

        when green =>
            if bad_event = '1' then
                next_state <= orange;
            end if;
        when orange =>
            if bad_event = '1' then
                next_state <= red;
            end if;
        when red =>
            ring_bell <= '1';
            if clear = '1' then
                next_state <= green;
            end if;
    end case;

end process;

end rtl;
```

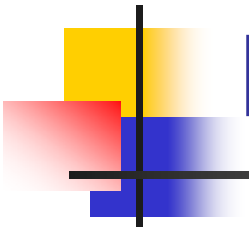
State Machine in one Process

```
architecture rtl of sm3 is

    type state_type     is (green, orange,
     red);
    signal state        : state_type;

begin

-- single process
process(clk, reset)
begin

    if reset='1' then
        state <= green;
        ring_bell <= '0';
    elsif rising_edge(clk) then
        case state is
            when green =>
                if bad_event = '1' then
                    state <= orange;
                end if;
            when orange =>
                if bad_event = '1' then
                    state <= red;
                    ring_bell <= '1';
                end if;
            when red =>
                ring_bell <= '1';
                if clear = '1' then
                    state <= green;
                    ring_bell <= ‘0';
                end if;
        end case;

    end if;

end process;

end rtl;
```



Summary

- Think in hardware!
- Beware of unassigned pathes (latch!)
- Use simple types
- Use simple statements



More Information

- FAQ comp.lang.vhdl
- Mark Zwolinski, Digital System Design with VHDL.