VHDL

- History
  - VHDL = VHSIC Hardware Description Language (VHSIC = Very High Speed Integrated Circuit)
  - Ordered by US defence department
  - Developed by Intermetrics, IBM and Texas Instruments (1983)
  - Standardized by IEEE (1987, 1993, and 2001) and ANSI (1988)
- Original goals
  - Modelling and simulation
  - Executable and unambiguous specifications
  - Standardized way of describing circuits
- Also used for
  - Synthesis

Entities and Architectures

- Designs are described using a number of modules, consisting of *entities* and *architectures*.
- An **entity** describes the *interface* of a module: names and types of input ports, output ports, generic parameters, etc.
- An **architecture** describes how the module works internally. Corresponds to the implementation or function body in other programming languages.
- Each module has exactly one entity, but may have several architectures (for instance one behavioral and one RTL implementation).

Examples of Entities

```
entity half_adder is
    port ( c_in, a : in bit;
           s, c_out : out bit );
end entity half_adder;
```

```
entity d_flip_flop is
    port ( i, clk : in bit;
           o : out bit );
end entity d_flip_flop;
```

An Entity with an Architecture

```
entity d_flip_flop is
    port ( i, clk : in bit;
           o : out bit );
end entity d_flip_flop;

architecture behavioral of d_flip_flop is
begin
    dff_behavior : process is
    begin
        if clk'event and clk='1' then
            o <= i;
        end if;
        wait on clk;
    end process dff_behavior;
end architecture behavioral;
```

Three ways to write code in architectures

1. **Structurally:** Instantiate other modules and connect them with signals (≈wires)
2. **Concurrently:** Simple statements (e.g. assignments) that are reexecuted every time signals they depend on change
3. **Sequentially:** Sequential code (possibly in several parallel processes), with loops etc

Higher abstraction levels usually contain more sequential code, while lower levels contain more structural code.

It is possible to mix different kinds of code in the same architecture.

Architecture with Concurrent Code

```
entity half_adder is
    port ( c_in, a : in bit;
                s, c_out : out bit );
end entity half_adder;

architecture behavioral of half_adder is
begin
    s <= a xor c_in;
    c_out <= a and c_in;
end architecture behavioral;
```

Structural architectures: example 1

Assuming that and_gate and xor_gate are implemented:

```
entity half_adder is
    port ( c_in, a : in bit;
           s, c_out : out bit );
end entity half_adder;

architecture structural of half_adder is
begin
    xor1 : entity work.xor_gate(behavioral)
           port map( c_in, a, s );
    and1 : entity work.and_gate(behavioral)
           port map( c_in, a, c_out );
end architecture structural;
```

Structural architectures: example 2

```
entity full_adder is
    port ( c_in, a, b : in bit;
                s, c_out : out bit );
end entity full_adder;

architecture structural of full_adder is
    signal c_1, c_2, s_1 : bit;
begin
    ha1 : entity work.half_adder(structural)
            port map( a, b, s_1, c_1 );
    ha2 : entity work.half_adder(structural)
            port map( c_in, s_1, s, c_2 );
    or1 : entity work.or_gate(behavioral)
           port map( c_1, c_2, c_out );
end architecture structural;
```

Architecture with Sequential Code

```
entity max3 is
    port(i1, i2, i3 : in integer; o : out integer);
end entity max3;

architecture behavioral of max3 is
begin
    p : process is
    begin
        if(i1 >= i2 and i1 >= i3) then
            o <= i1;
        elsif(i2 >= i1 and i2 >= i3) then
            o <= i2;
        else
            o <= i3;
        end if;
        wait on i1, i2, i3;
    end process p;
end architecture behavioral;
```

More on Sequential VHDL

- Sequential code is always written in *processes*
- An architecture may contain several processes
- Processes are implicit loops: when the last statement in a process has been executed, the process is restarted
- Processes communicate using *signals*
- From a language point of view, signals resemble variables (although important differences exist)
- From a hardware point of view, signals resemble wires

Sensitivity lists

```
entity half_adder is
    port ( c_in, a : in bit;
           s, c_out : out bit );
end entity half_adder;

architecture behavioral of half_adder is
begin
    half_add_behavior : process(a, c_in) is
    begin
        s <= a xor c_in;
        c_out <= a and c_in;
    end process half_add_behavior;
end architecture behavioral;
```

Sensitivity lists may not be combined with **wait** statements.

A simple clock generator

```
entity clock1MHz is
    port ( clk : out bit );
end entity clock1MHz;

architecture behavioral of clock1MHz is
begin
    c : process is
    begin
        clk <= '0';
        wait for 500 ns;
        clk <= '1';
        wait for 500 ns;
    end process c;
end architecture behavioral;
```

Time in VHDL

- VHDL has a notion of time
- Simulation time $\neq$ actual running time
- Certain events are scheduled to happen at specific points in time
- Time advances in discrete steps - when nothing more is scheduled to happen at the current point in time, time is advanced to the next point in time where there is a scheduled event.
- Time has two components:
  - a number of (nano/micro/milli)seconds
  - a number of *deltas* (δ)
- The minimal delay for a signal assignment is 1δ.

A small example

```
proc : process is
begin
    a <= 0;
    b <= 0;
    wait for 0 ns;
    a <= a+1;
    b <= a;
    wait for 0 ns;
    - - Here, a=1, b=0 !
    wait;
end process proc;
```

Example: Counting Bits, Attempt 1

```
entity count_ones is
    port( arr : in bit_vector; result : out integer );
end entity count_ones;

architecture attempt_1 of count_ones is
begin
    p1 : process(arr) is
    begin
        result <= 0;
        for i in arr'range loop
            if(arr(i) = '1') then
                result <= result+1;
            end if;
        end loop;
    end process p1;
end architecture attempt_1;
```

Compilation error: *"Cannot read output: result"*

Example: Counting Bits, Attempt 2

```
architecture attempt_2 of count_ones is
    signal tempSum : integer;
begin
    p1 : process(arr) is
    begin
        tempSum <= 0;
        for i in arr'range loop
            if(arr(i) = '1') then
                tempSum <= tempSum+1;
            end if;
        end loop;
        result <= tempSum;
    end process p1;
end architecture attempt_2;
```

Testing reveals strange values on result.

Example: Counting Bits, Correct Version

```
architecture correct of count_ones is
begin
    p1 : process(arr) is
        variable tempSum : integer;
    begin
        tempSum := 0;
        for i in arr'range loop
            if(arr(i) = '1') then
                tempSum := tempSum+1;
            end if;
        end loop;
        result <= tempSum;
    end process p1;
end architecture correct;
```

Simulation cycle

1. Simulation time is advanced until the next scheduled event (can be a signal assignment, or a wait).
2. Scheduled signal assignments are carried out.
3. Processes resume execution if they
   - are sensitive to signals that was affected, or
   - are scheduled to wait until the current time point
4. The processes continue to run until they all reach wait statements. New events that the processes create are put in the event queue. Events to happen after a delay of length 0 are scheduled to happen in the next delta cycle. Events to happen after a positive time are scheduled at that time $+0\delta$.

Variables vs. signals

Variables

Used in algorithms

Immediate assignments (**:=**)

Belong to a process

Used only within a process

Signals

Represent physical wires

Delayed assignments (**<=**)

Belong to an architecture

Communicate betw procs/entts

Different attributes (’**event** etc)

Sensitivity lists and **wait on**

Example: Variables

```
entity count_pos_edges is
   port (i : in bit; c : out integer);
end entity count_pos_edges;

architecture behavioral of count_pos_edges is
begin
   p1 : process(i) is
      variable counter : integer := 0;
   begin
      if i'event and i='1' then
         counter := counter + 1;
         c <= counter;
      end if;
   end process p1;
end architecture behavioral;
```

Delayed Signal Assignments

- Signal assignments can be delayed:
  - o <= i **after** 2 ns;
  - test_input <= 0,
    1 **after** 1 ms,
    2 **after** 2 ms,
    3 **after** 3 ms;

- The smallest possible clock generator:
  - clk <= not clk **after** 50 us;

  (concurrent assignment)
- The value is evaluated upon execution of assignment statement, not when the assignment is carried out.

Assert statements

- **assert** e;
  Gives a warning if e evaluates to false.
- **assert** e **report** str;
  Prints the message str if e evaluates to false.
- **assert** e **report** str **severity** sev;
- Sev can be one of: note, warning, error, failure.
  Simulator can be set to stop simulation at given severities
- An assert statement in an architecture is concurrent: property should always hold
- An assert statement in a process is sequential: property should hold when the assert is executed

More wait statements

- **wait**; (halts process)
- **wait for** 100 ms;
- **wait on** in1, in2, in3; (wait for *events* on signals)
- **wait until** a='1';
- **wait on** clk **until** reset = '0';
- **wait until** a = '1' **for** 150 ns;

A wait statement *always* causes a process to wait. If we write **wait until** a='1', and a='1', then the process will wait until an event that changes a's value to '1' happens. **Wait for** 0 ns; will cause the process to wait one δ.

More signal attributes

A *transaction* is when a value is assigned to a signal. An *event* is when a transaction causes a signal to change value.

- **'event**: event has occured last (simulation) cycle
- **'active**: transaction has occured last cycle
- **'stable**(t): no event has occured for t time units
- **'quiet**(t): no transaction has occured for t time units
- **'last_value**: value before last event
- **'last_event**: time since last event
- **'delayed**(t): implicit signal, delays the original signal with t time units.

Declaring Your Own Types

- Types are usually declared in packages:

```
package my_types is
    type int_array is array (integer range<>) of integer;
    type int_pair is record
        first : integer;
        second : integer;
    end record;
end package my_types;
```

- Packages must be imported before *all* entities that use them:

```
use work.my_types.all;
```

Guidelines for writing architectures

Processes that are sensitive to the clock signal implicitly introduces flipflops. Architectures usually becomes easiest to write and maintain if they are divided into two parts:

1. One single process which is sensitive only to the clock signal (and possibly an asynchronous reset signal)
2. One or more processes that are sensitive to any input signals or internal signals, except the clock

The only thing the first process does is to update the internal state (i.e. the flipflops). The other parts includes all logic.

More about this in Jiri Gaisler's notes (see Literature page)