How we treat VHDL in this course…

- You will not become an expert in VHDL after taking this course
- The goal is that you should learn how VHDL can be used for simulation and synthesis
  - For synthesis we must know the limitations of what constructs are supported. We **must also** have a knowledge in digital design or else we do not know what we want to model!
  - Simulation is completely different, here we can use all features of VHDL
- The course text book does not cover VHDL in detail, instead it contains a mix between VHDL modeling and digital design
- A good book on VHDL is “A Designers Guide to VHDL” by Peter Ashenden. The next few following slides are adapted from “VHDL Quick Start Guide” by Ashenden
Analysis (ncvhdI)

- Check for syntax and semantic errors
  - syntax: grammar of the language
  - semantics: the meaning of the model
- Analyze each design unit separately
  - entity declaration
  - architecture body
  - etc…
  - best if each design unit is in a separate file
- Analyzed design units are placed in a library
  - current library is called work

Elaboration (ncelab)

- Flattening the design hierarchy
  - create ports
  - create signals and processes within architecture body
  - for each component instance, copy instantiated entity and architecture body
  - repeat recursively
    - bottom out at purely behavioral architecture bodies (no component instantiations)
- Final result of elaboration
  - flat collection of signal nets and processes
Simulation (ncsim)

- Execution of the processes in the elaborated model
- Discrete event simulation
  - time advances in discrete steps
  - when signal values change – *events*
- A processes is sensitive to events on input signals
  - specified in wait statements
  - a process sensitivity list is an implicit wait statement
  - resumes and schedules new values on output signals
    - schedules *transactions* (transaction, an input change that may lead to an event)
    - event on a signal if new value different from old value

Simulation algorithm

- Initialization phase
  - each signal is given its initial value (start value from declaration or first value from type definition *type’left*)
  - simulation time set to 0
  - for each process (a concurrent signal assignment is in fact a process)
    - activate
    - execute until a wait statement, then suspend
    - since a sensitivity list is an implicit wait statement a process with a wait statement suspends at the end of the process
    - execution usually involves scheduling transactions on signals for later times
Simulation algorithm

- Simulation cycle
  - advance simulation time to time of next transaction
  - for each transaction at this time
    - update signal value
      - event if new value is different from old value
    - for each process sensitive to any of these events, or whose “wait for …” time-out has expired
      - resume
      - execute until a wait statement, then suspend
  - Simulation finishes when there are no further scheduled transactions

Delta delays

- A delta delay can be thought of as an infinitesimal unit of time “orthogonal” to simulation time
- Zero delays are modeled as delta delays, so that any events generated with zero delay are scheduled to occur one delta delay later.
- Delta delays are used to order events
Simulation - an example

architecture sim of Test is

signal A, B, C, D : std_logic := '0';
signal S1, S2, Y : std_logic;
begin

A <= '0' after 5 ns, '1' after 10 ns;
B <= '1' after 5 ns;
C <= '0' after 5 ns, '1' after 10 ns;
D <= '1' after 5 ns;
S1 <= A xor B;
S2 <= C xor D;
Y <= S1 and S2;
end Sim;

Simulation modeling issues

- A process should have a complete list of relevant signals in its sensitivity list. Do not include irrelevant signals
- Minimize the number of processes (note this is not true for synthesis)
- Understand the significant differences between signal and variable assignments
- Use variables if possible. Signals have histories and have therefore large complicated data structures
- Use the falling_edge and rising_edge functions so that only true transitions are detected (clock'event and clock = '1' is true for a 'X' → '1' transition)
- Normal programming efficiency apply – take invariant assignments out of loops
- Conditional compilations (like #ifdef in C) is provided by if ... generate
- Integers simulate faster than bit vectors (unsigned and signed)
When should we simulate?

Alternatives to event driven simulation

- Some problems with event driven simulation:
  - Slow execution
  - Finding a suitable set of test vectors
- Cycle based simulation
  - Results are only calculated at clock edges
  - Timing is ignored, but must be checked with static timing analysis tool
- Formal verification
  - “Establishing properties of hardware or software designs using logic, rather than (just) testing or informal arguments. This involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove, say, that the implementation satisfies the specification.” (source http://www.nist.gov/dads/)
Synthesis

- System synthesis
  - Processors, Memories
  - Registers, ALUs, MUXs
  - Gates, flip-flops
- Behavioral synthesis
  - Transistors
  - Transistor layouts
  - Cells
  - Chips
  - Boards, MCMs
- Logic synthesis
  - Logic Level
- Circuit synthesis
  - RT Level
- Structural view
  - System Level

VHDL support for synthesis

- The subset of VHDL supported for synthesis is tool dependent. Read your synthesis tool manual!
- The VHDL Synthesis working group (IEEE 1076.3) is working on standardizing synthesizable VHDL code
- Some constructs that are not supported
  - Variable and signal initializations are ignored by the synthesis tool. Use a reset signal to initialize registers
  - while-loop are generally not supported because the loop range must be statically determined in order to generate a fixed amount of logic
  - Delays can not be synthesized. The after and ignored and wait for statements are ignored
  - File operations
  - Pointers (access type) are not supported. The size of a dynamic data structure is not known at compile time
  - Floating point data types are not supported
library ieee;
use ieee.std_logic_1164.all;

entity Counter is
port ( 
    Clk, Reset : in std_logic;
    Pulse : out std_logic;
end Counter;

architecture RTL of Counter is
begin
    process (Clk, Reset)
    variable Counter : integer range 0 to 255;
    begin
        if Reset = '1' then
            Counter := 0;
            Pulse <= '0';
        elsif rising_edge(Clk) then
            if Counter = 23 then
                Pulse <= '1';
            else
                Pulse <= '0';
            end if;
            if Counter = 255 then
                Counter := 0;
            else
                Counter := Counter + 1;
            end if;
        end if;
    end process;
end RTL;

entity Seq is
port ( 
    A, B : in unsigned(7 downto 0);
    Opcode : in std_logic_vector(1 downto 0);
    Clk : in std_logic;
    Result : out unsigned(7 downto 0);
end Seq;

architecture Don'tCare of Seq is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            case Opcode is
                when "01" => -- Add
                    Result <= A + B;
                when "10" => -- Sub
                    Result <= A - B;
                when others =>
                    Result <= (others => 'X');
            end case;
        end if;
    end process;
end Don'tCare;

architecture DoCare of Seq is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            case Opcode is
                when "01" => -- Add
                    Result <= A + B;
                when "10" => -- Sub
                    Result <= A - B;
                when others =>
                    null;
            end case;
        end if;
    end process;
end DoCare;
Don't cares in clocked processes cont.

Don't care in others case

Null in others case

- Clock enable or feedback mux will be created (same thing actually)

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

entity Seq is
  port (A, B : in unsigned(7 downto 0);
        Opcode : in std_logic_vector(1 downto 0);
        Result : out unsigned(7 downto 0));
end Seq;

architecture DontCare of Seq is
begin
  process (A, B, Opcode)
  begin
    case Opcode is
      when "01" => -- Add
        Result <= A + B;
      when "10" => -- Sub
        Result <= A - B;
      when others =>
        Result <= (others => 'X');
    end case;
  end process;
end DontCare;

architecture NoGood of Seq is
begin
  process (A, B, Opcode)
  begin
    case Opcode is
      when "01" => -- Add
        Result <= A + B;
      when "10" => -- Sub
        Result <= A - B;
      when others =>
        null;
    end case;
  end process;
end NoGood;
```

"Combinational" process

```
“Combinational” process cont.

Don’t care in others case

Null in others case

- Latch created to “remember” last value. Not good!

To avoid having latches inferred for a signal in a process then every execution path through the process must assign a value to the signal

Priority coded logic

- Remove any unnecessary priority structures for timing-critical designs. For example, use case statements instead of nested if-then-else clauses if the logic is truly priority independent.
- Sorry, I can’t think of a good example 😊
Synthesis constraints

The perfect digital system is infinitely fast, infinitesimally small, consume no power and is totally testable.

In reality we need to determine what our objectives are and pass them as constraints to the synthesis tool. Some constraints may be:

- State encoding (see previous lecture)
- Area constraints
  - The area constraint we set in FPGA synthesis is obviously that the logic should fit in the FPGA. The goal is to fit the design in the smallest and cheapest FPGA
  - For ASIC synthesis we want to minimize the silicon area
- Resource constraints
- Timing constraints (see previous lecture)

Remember that the most important optimizations are done prior to RTL synthesis! The architecture design has a large impact on the results.

Resource sharing

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

entity ALU is
  port (A, B : in unsigned(7 downto 0);
        Opcode : in std_logic_vector(1 downto 0);
        Result : out unsigned(7 downto 0);
  end ALU;

architecture Comb of ALU is
begin
  process (Opcode, A, B)
  begin
    case Opcode is
      when "00" => -- Add
        Result <= A + B;
      when "01" => -- Inc
        Result <= A + 1;
      when "10" => -- Sub
        Result <= A - B;
      when "11" => -- Dec
        Result <= A - 1;
      when others =>
        Result <= (others => 'X');
    end case;
  end process;
end Comb;
Behavioural synthesis

- In RTL synthesis, the design is specified in terms of register operations and transformed automatically into gates and flip-flops. Before RTL synthesis the architecture of the design has been decided.
- In behavioral synthesis the input is an algorithm and the output is an RTL description of the architecture. The RTL description is then further compiled to a gate level netlist.
- The choice of an architecture for a design will have a great impact on area, speed, power consumption etc.
- We do not use a behavioural synthesis tool in this course, you derive the architecture manually.
- Behavioural synthesis will be introduced in the following slides with an example from Zwolinsky chapter 9.

package iir_defs is
  constant precision : positive := 16;
  subtype int is integer range 2**(precision - 1) to 2**(precision - 1) - 1;
  type integer_array is array (natural range <>) of int;
  constant order : positive := 5;
end package iir_defs;

use work.iir_defs.all;

entity iir is
  generic
    coeffa : integer_array (0 to order);
    coeffb : integer_array (0 to order-1);
  port
    (input : in int;
    strobe : in bit;
    output : out int);
end entity iir;

architecture behaviour of iir is
begin
  process
    variable input_sum : int;
    variable output_sum : int;
    variable delay : integer_array (0 to order) := (others => 0);
    begin
      input_sum := input;
      for j in 0 to order-1 loop
        input_sum := input_sum + (delay(j)*coeffb(j))/1024;
      end loop;
      output_sum := (input_sum*coeffa(order))/1024;
      for k in 0 to order loop
        output_sum := output_sum + (delay(k)*coeffa(k))/1024;
      end loop;
      for l in 0 to order-1 loop
        delay(l) := delay(l+1);
      end loop;
      delay(order) := input_sum;
      output <= output_sum;
      wait on strobe;
    end process;
end architecture
First order filter

First we unroll the loops (order = 1)

\[
\begin{align*}
\text{input\_sum} & := \text{input} + \text{delay}(0)\cdot\text{coeffb}(0); \\
\text{output\_sum} & := \text{input\_sum}\cdot\text{coeffa}(1); \\
\text{output\_sum} & := \text{output\_sum} + \text{delay}(0)\cdot\text{coeffa}(0); \\
\text{output} & := \text{output\_sum} + \text{delay}(1)\cdot\text{coeffa}(1);
\end{align*}
\]

Next we convert to single assignment form

\[
\begin{align*}
\text{input\_sum} & := \text{input} + \text{delay}(0)\cdot\text{coeffb}(0); \\
\text{output\_sum0} & := \text{input\_sum}\cdot\text{coeffa}(1); \\
\text{output\_sum1} & := \text{output\_sum0} + \text{delay}(0)\cdot\text{coeffa}(0); \\
\text{output} & := \text{output\_sum1} + \text{delay}(1)\cdot\text{coeffa}(1);
\end{align*}
\]

We will consider \(\text{delay}(0)\) and \(\text{delay}(1)\) to be inputs and construct a data dependency graph.

Data dependency graph

If the operations were all performed in one clock cycle, three adders and four multipliers would be needed. Assuming that each operation takes one clock cycle we can schedule the data graph. Scheduling – assigning each operation to a control stop (clock cycle)
ASAP schedule

In As Soon As Possible scheduling each operation is scheduled to be done as early as possible. We have a latency of five clock cycles.

Note that the sequence of operations is not the same as given by the original VHDL source:

\[
\begin{align*}
\text{input} & := \text{input} + \text{delay}(0) \cdot \text{coeffb}(0); \\
\text{output}_0 & := \text{input} \cdot \text{coeffa}(1); \\
\text{output}_1 & := \text{output}_0 + \text{delay}(0) \cdot \text{coeffa}(0); \\
\text{output} & := \text{output}_1 + \text{delay}(1) \cdot \text{coeffa}(1);
\end{align*}
\]

ALAP schedule

In As Late As Possible scheduling each operation is scheduled to be done as late as possible. Here we also have a latency of five clock cycles. The schedule is unconstrained, i.e. we have not set a limit on how many adders or multipliers we have available.
Comparing the ASAP and ALAP schedules

**ASAP schedule**
- Three multipliers and one adder

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Delay(0)</th>
<th>Coeffb(0)</th>
<th>Coeffa(1)</th>
<th>Coeffa(0)</th>
<th>Delay(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ALAP schedule**
- Two multipliers and one adder

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Delay(0)</th>
<th>Coeffb(0)</th>
<th>Coeffa(1)</th>
<th>Coeffa(0)</th>
<th>Delay(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another schedule...

- One multiplier and one adder with a latency of five clock cycles
Resource constrained ALAP schedule

In this ALAP schedule we are constrained to a single arithmetic unit. Hence we can only perform one operation per clock cycle. The latency increases to seven clock cycles.

Allocation

Allocation refers to the process of specifying the components in the system and the interconnect among components.

- Allocation of register or RAM storage to hold data values
- Allocation of functional units to perform specified operations
- Allocation of interconnect paths from the transmission of data among components.

Allocation and scheduling are not independent operations.
Allocating functional units

For this schedule we can allocate three functional units, two multipliers and one adder. The functional units are shared.

Allocating registers

Assumption, all inputs are stable for five clock period. Registers are inserted every time an edge crosses a clock boundary.
Register sharing

Similar to function units registers may be shared. Two or more registers whose lifetime are nonoverlapping can share the same register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Lifetime in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1-2</td>
</tr>
<tr>
<td>R2</td>
<td>1-2</td>
</tr>
<tr>
<td>R3</td>
<td>1-2</td>
</tr>
<tr>
<td>R4</td>
<td>2-3</td>
</tr>
<tr>
<td>R5</td>
<td>2-3</td>
</tr>
<tr>
<td>R6</td>
<td>2-3</td>
</tr>
<tr>
<td>R7</td>
<td>3-4</td>
</tr>
<tr>
<td>R8</td>
<td>3-4</td>
</tr>
<tr>
<td>R9</td>
<td>3-4</td>
</tr>
<tr>
<td>R10</td>
<td>4-5</td>
</tr>
<tr>
<td>R11</td>
<td>4-5</td>
</tr>
</tbody>
</table>

Data path

Now the data path can be constructed from the data flow diagram.
... and now the ASM chart

Reg1 ← input
Reg2 ← delay(0)*coeffb(0)
Reg3 ← delay(0)*coeffa(0)
Reg4 ← Reg1 + Reg2
Reg5 ← Reg3
Reg6 ← Reg5
output ← Reg4 + Reg6