VHDL GUIDELINES FOR SYNTHESIS

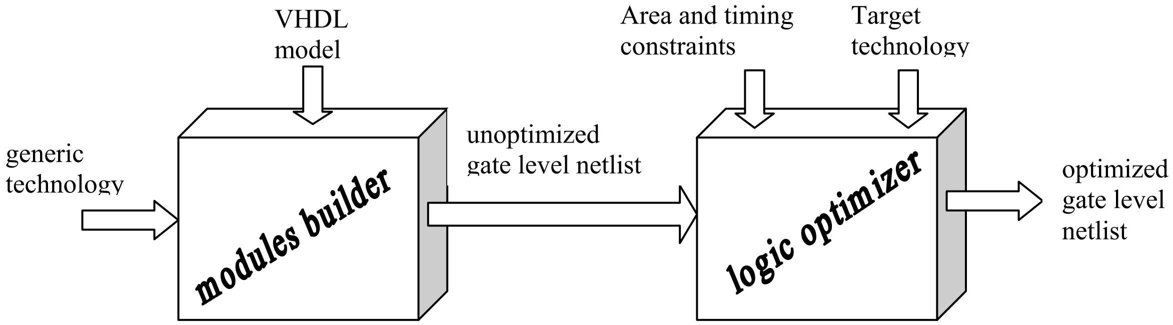
BASICS

VHDL

VHDL (**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage) is a hardware description language that allows a designer to model a circuit at different levels of abstraction, ranging from the gate level, RTL (**R**egister **T**ransfer **L**evel) level, behavioral level to the algorithmic level.
Thus a circuit can de described in many different ways, not all of which may be synthesizable.

SYNTHESIS

Synthesis is the process of constructing a gate level netlist from a model of a circuit described in VHDL.


VHDL model
Area and timing constraints
Target technology
generic technology
modules builder
unoptimized gate level netlist
logic optimizer
optimized gate level netlist

LATCH VS. FLIP-FLOP

A latch is a storage element level triggered while a flip flop is a storage element edge triggered.
If not absolutely necessary avoid the use of latches.

MAPPING STATEMENTS TO GATES

BASIC STATEMENTS

- **if statement**

```
if a > b then
z := a;
else
  z  := b;
end if;
```

If a signal or a variable is not assigned a value in all possible branches of an if statement, a latch is inferred. If the intention is not to infer a latch, then the signal or variable must be assigned a value explicitly in all branches of the statement.

- **case statement**

```
case opc is
  when add =>
    z <= a + b;
when sub =>
    z <= a - b;
when mul =>
    z <= a * b;
when div =>
    z <= a / b;
when others =>
    null;
end case;
```

If a signal or a variable is not assigned a value in all possible branches of a case statement, a latch is inferred. If the intention is not to infer a latch, then the signal or variable must be assigned a value explicitly in all branches of the statement.

- **Null statements**

A null statement means that no action is required.

- **Loop statement**

There are three kinds of loop statement in VHDL:

- while-loop
- for-loop
- loop

The only loop supported for synthesis is the *for-loop*.

- **Wait statement**

There are three forms of the wait statement:

wait for *time;*
wait until *condition;*
wait on *signal-list;*

The *wait-until* form is the only one supported for synthesis. If used the *wait* statement must be the first statement and the only wait statement present in the process. Furthermore the condition in the wait statement must be a clock expressions that indicates a falling or a rising clock edge.

wait until clock_name = clock_value;
wait until clock_name = clock_value **and** clock_name'event;
wait until clock_name = clock_value **and not** clock_name'stable;

Statements that follow a *wait* statement refer to a statement that executes synchronously with the clock edge. A variable or a signal assigned a value following the *wait* statement is synthesized as a flip-flop.

```
architecture rtl of incr is
begin
  process
  begin
      wait until clk = '1';
        count <= count + 1;
  end process;
end rtl;
```

Recommendation: avoid the use of wait statements for synthesis (see next section)

MODELING FLIP-FLOPS

We saw that it is possible to infer flip-flops from signals and variables when they are assigned values within a process that contains a wait statement as the first statement. A better way is to use a special if statement in a process. The syntax is of the form:

```
if clock expression then
  sequential-statements
end if;
```

where a clock-expression is one of the following:

clock_name = clock_value **and** clock_name'event;
clock_name = clock_value **and not** clock_name'stable;

Example:

```
library ieee
use ieee.std_logic_1164.all

entity cnt4 is
  port (clk, load, updown : in std_logic;              -- clock
           value: in std_logic_vector(1 downto 0);  -- value to be loaded
           dout: out std_logic_vector(1 downto 0))  -- data out
end cnt4;

architecture rtl of cnt4 is
  signal cnt: unsigned (1 downto 0);
begin
  process (clk,load,updown,value)
  begin
     if (clk = ‘1’ and clk’event) then
       if load = ‘1’ then
          cnt <= value;
       else
          if updown = ‘1’ then
            cnt  <= cnt + 1;
          else
            cnt  <= cnt –1;
          end if;
       end if;
    end if;
    dout <= cnt;
  end process;
end rtl;
```

These signals in sensitivity list will slow down the simulation ..

GET RID OF THEM !!

The difference between the if statement style and the wait statement style is that in the if statement style more than one clock can be modeled in a single process. More importantly, the description of combinational logic and sequential logic can be lumped into one process, and the code is more readable.

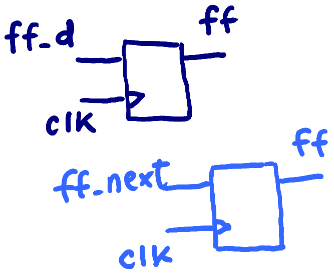
Recommendation: DO NOT USE the *wait* statement in code that is going to be synthesized.

- **Flip-Flops with asynchronous preset and reset.**

```
if condition-1 then
  asynchronous-logic-1
elsif condition-2 then
  asynchronous-logic-2
– any number of elsif
…
elsif clock-expression then
.. synchronous-logic
end if;
```

Example:

```
…
async_registers: process (clk, preset, reset)
begin – async_registers
  if (preset = ‘1’) then
    ff <= ‘1’;
  elsif (reset = ‘1’) then
    ff <= ‘0’;
  elsif (clk=’1’ and clk’event) then
    ff <= ff_d;
  endif;
end process async_registers;
…
```



- **Flip-Flops with synchronous preset and reset.**

```
…
sync_registers: process (clk)
begin – sync_registers
  if (clk=’1’ and clk’event) then
    if (preset = ‘1’) then
      ff <= ‘1’;
    elsif (reset = ‘1’) then
      ff <= ‘0’;
    else
      ff <= ff_d;
    endif;
  endif;
end process sync_registers;
…
```

SEQUENTIAL SIGNAL ASSIGNMENT STATEMENT

Functions
A function call represents combinational logic.

Procedures
A procedure call can represent either combinational logic or sequential logic depending on the context under which the procedure call occurs.

Generics
Generics provide a mechanism in VHDL to model parameterized designs.

```
library ieee
use ieee.std_logic-1164.all

entity generic_and is
  generic (size: integer);
  port (a: in std_logic_vector(0 to size-1); z: out std_logic);
end generic_and;
```

```
architecture example of generic_and is
begin
  process (a)
    variable res: std_logic;
  begin
    res := ‘1’;

    for k in a’range loop
      res := res and a(k);
    end loop;

    z <= res;
  end process;
end example;
```

The entity *generic_and* cannot be synthesized by itself since the value of *size* is not yet specified. Such an entity is synthesized when it is instantiated within other entities.

Using predefined blocks

Component instantiation statements are used when a designer is not satisfied with the quality of circuits produced by a synthesis tool.

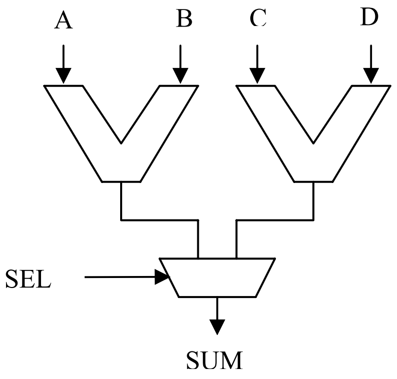
MODEL OPTIMIZATIONS

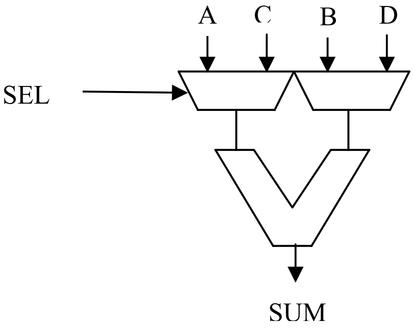
Synthesize consist in minimize certain cost functions. The initial point of the optimization process is the vhdl code. To get good quality designs (meet the expected timing and area goals) it is very important how the code is written. If the quality of the code is poor the optimization process won’t converge towards the expected results. In most cases if the synthesis tool has a hard time to converge on the desired result, rather than increasing the tool optimization effort (with considerable increase of the run time), it is better to rewrite the code.

RESOURCE ALLOCATION

Resource allocation refers to the process of sharing an operator under mutually exclusive conditions.

```
if sel = ‘1’ then
  sum := a +b;
else
  sum := c+d;
end if
```





In timing critical designs, it may be better if no resource sharing is performed. Operators that are usually shared are:

- relational operator (comparator)
- addition
- subtraction
- multiplication
- division

CONVERSION FUNCTIONS

Since conversion functions do not represent hardware, it is important to find out the built-in conversion functions provided by a synthesis tool and only use these where necessary. In such cases, no extra logic is synthesized for the conversion functions.

TYPE INTEGER

The VHDL predefined type INTEGER represent a minimum of 32bits in hardware (since the minimum defined range of type integer is $-(2^{31} -1\ )$ to $+(2^{31}- 1)$. In many modeling situations, it is not necessary to model an integer as 32 bits.
The recommendation is to use the unbounded type integer only where necessary. In most of the cases, it would be better to specify a range constraint with the type integer.

COMMON SUB-EXPRESSIONS

It's very useful in practice to identify common sub-expressions and to reuse computed values where possible.

```
…
run  <= r1 + r2;
…
stop <= r3 – (r1 + r2);
…
```

If the synthesis tool doesn't identify common sub-expression, two adders would be generated, each computing the same result, that of r1 + r2.
Therefore it is useful to identify common sub-expression and to reuse the computed values.

```
…
tmp := r1 + r2;
…
run  <= tmp;
…
stop <= r3 – tmp;
```

MOVING CODE

Typically a synthesis tool handles a for-loop by unrolling the loop the specified number of times. In such a case, redundant code is introduced for the expression whose value is independent of the loop index.

```
hot :=
…
for count in 1 to 5 loop
…
   tip := hot – 6;
   --  assumption: hot is not assigned a new value within the loop
…

end loop;
```

The best way to handle this case is to move the loop independent expression out of the loop. This also improves simulation efficiency.

```
hot := …
…
tmp := hot – 6; -- a temporary variable is introduced
for count in 1 to 5 loop
…
  tip := tmp;
  --  assumption: hot is not assigned a new value within the loop
…

end loop;
```

COMMON FACTORING

Common factoring is the extraction of common sub-expressions in mutually exclusive branches of an if or a case statement.
By performing this common factoring, less logic is synthesized so that a logic optimizer may concentrate on optimizing more critical areas.

COMMUTATIVITY AND ASSOCIATIVITY

In certain cases, it might necessary to perform commutative and/or associative operations.

```
val_a := a + b+ c;
val_b := c +a –b;
```

A better way to code may be:

```
tmp     := c+a;
val_a  := tmp +b;
val_b  := tmp –b;
```

OTHER OPTIMIZATIONS

- Dead code elimination
- Constant folding

Dead code elimination deletes code that never gets executed.
Constant folding implies to compute constant expressions during compile time as opposed to implementing logic and then letting a logic optimizer try getting rid of the logic.

```
constant fact: integer := 4;
…
z := 2** fact;
```

constant folding computes the value of the right-hand side expression during compile time and assign the value to Z. No hardware need to be generated. This leads to savings in logic optimization time.

```
z := 2**4  -- worse solution than defining a constant.
```

DESIGN SIZE

Small designs synthesize faster. Synthesis run times are exponential with design size. A reasonable block size is between 2000 and 5000 gates.

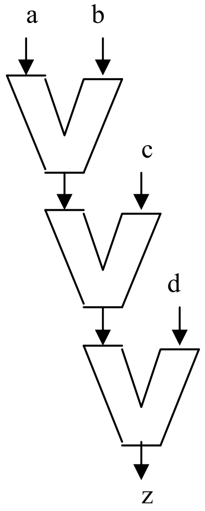
MACROS AS STRUCTURE

Synthesis is not the right mechanism to build a memory such as a ROM or a RAM. Those are usually predefined in a technology library. When a component such as a RAM or a ROM is required, it is better to treat this as a component, instantiate this in the code, and then synthesize.
Similar actions may be necessary if a designer has a statement of the form:
z := x*y; -- 16 bit arguments
and expect the synthesis tool to implement an efficient multiplier. The designer may have a better designed multiplier. So, again in this case it is better to instantiate a multiplier as a component, rather than expressing the multiplication operator.

USING PARENTHESIS

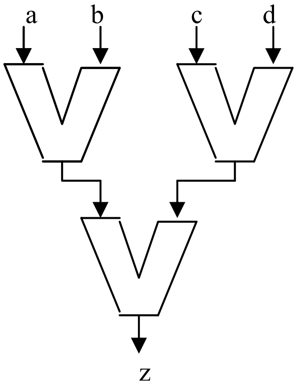
When writing VHDL the designer must be aware of the logic structure being generated. One important point is the use of parenthesis. Here is an example:

z := a + b – c – d



A better alternative is to use parenthesis:

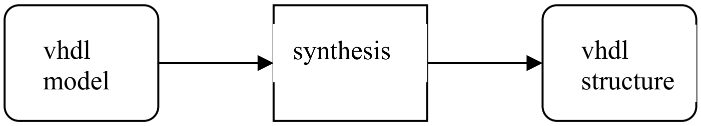
z:= (a + b) – (c+d)



Recommendation: use parenthesis to control the structure of the synthesized logic.

VERIFICATION

Having synthesized a VHDL model into a netlist, it is important to verify the functionality of the synthesized netlist to ensure it still matches the intended functionality.



A test bench is a model written in VHDL that applies stimulus, compares the output responses, and reports any simulation mismatches.

SIGNALS VS. VARIABLES

If a signal is used instead of a variable to hold a temporary value in a set of sequential assignment (within a process), simulation mismatches can occur.

```
process (clk)
begin
  if (clk = ‘1’ and clk’event) then
    zt <= a;
  end if;
  z <= zt
end process;
```

A mismatch will occur since changes to signal zt do not propagate to z at the same time.

Recommendation: check to see if there are any signals that are being assigned a value and then later on read in to the process. To prevent mismatches, it is better to model such temporaries as variables.

DELAYS

Delays are ignored by synthesis tools. This may cause a mismatch in simulation results.

```
…
z <= ‘1’ after 3ns;
if cond then
  z <= ‘0’ after 5ns;
…
```

Recommendation: avoid inserting delays into a model that is to be synthesized.

RESOLUTION FUNCTIONS

A synthesis tool may treat a resolution function in a different way than expected.
Recommendation: avoid having two or more drivers for a signal, thus removing the need for a resolution function.

SENSITIVITY LIST

A process with an incomplete sensitivity list causes simulation mismatches.
Recommendation: include all signals read in the process in the sensitivity list of the process.

INITIALIZATION

Synthesis tools ignore initial values specified for a variable or a signal in its declaration. This can cause a mismatch in simulation results.

VHDL STYLE GUIDELINES

SIMPLIFIED OVERVIEW OF THE DESIGN METHODOLOGY:

- Specify system requirements in a document
- Design top-down using block capture tools
- Fill out and simulate the leaf VHDL modules using RTL code
- Start validating larger and larger chunks, bottom-up style
- Synthesize in parallel with validation
- Use synthesizer's timing and gate count reports to judge synthesis results
- If needed, re-code VHDL
- Work on synthesis constraint files
- Simulate chips and system
- Run static timing analyzer on post-layout net-list
- Run system simulation regression on post layout net-list

PARTITIONING AN ASIC FOR SYNTHESIS: 2 GUIDING RULES

There are two important rules that a designer must keep in mind when blocking out a design for synthesis:

- Keep the leaf modules reasonably sized
- Limit the number of separate modules that a timing path can traverse

By limiting module size, a designer gains in both the synthesizer runtimes and the quality of the synthesis results.

By limiting the number of separate modules that a timing path can traverse, a designer can improve synthesis results in both timing and area. The synthesizer does much better on logic that is contained within one entity. If a certain timing path travels though 4 separate modules, the synthesizer will make its best effort within each of the modules, but the overall results won't be as good as the case where the whole path is in a single module.

Notice that there is a trade off between the two rules of partitioning. The synthesizer always does better on smaller sized modules, however it also does better when all of a path's logic is contained within one module.

THINKING IN HARDWARE

Writing synthesizable RTL is not like writing software. RTL describes real hardware. RTL means "register transfer level" – if registers aren't clearly inferred, the code is not RTL.

The main idea of "thinking in hardware" is to always have a good idea of what the lines of code really represent in terms of hardware. You must know it when you lay down an adder. You must be aware of the cost of a magnitude comparator. You must be conscious of the relative timing of different signals with respect to the clock edge. After writing a sequence of RTL, one should be able to sketch out a quick block diagram of what the code physically represents.

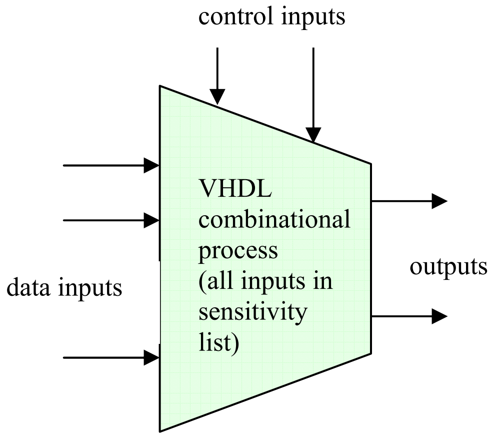
Many traditional software design rules still apply to VHDL coding. Modularity, good style, good comments, good test plan, and good design specs are all needed to successfully write a hardware description.
Writing effective VHDL RTL is easy. Only a small subset of VHDL is really needed to build a chip.

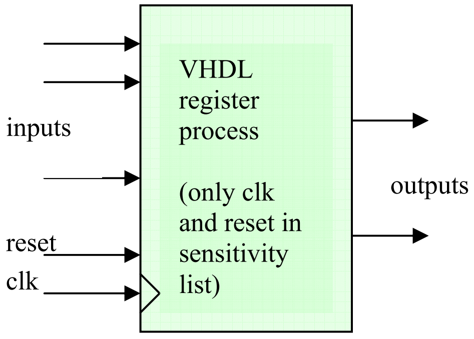
RELATIVE SIGNAL TIMING

The most important part of thinking in hardware is perhaps to consider the concept of relative signal timing. Writing RTL that does not take a signal's timing into consideration is not useful. Signals may stabilize at any point in the clock cycle. Different signals may have different maturity times depending on how much combinational logic precedes them in the timing path.
VHDL take cares of gate-level complexity. It does not take care of gate level timing. It is especially easy to make mistakes with signals arriving late from other VHDL modules. The designer must develop an "intuition" that cannot do much with such a signal in the current clock cycle. Late arriving signals should be run into a minimum of logic before a register stage.

DESCRIBING HARDWARE WITH VHDL PROCESS.

The fundamental unit of an RTL description is the VHDL "process", When thinking in hardware, there are two kinds of process: a combinational process and a register –instantiating process.
The combinational processes have sensitivity list that include all signals used in right-hand-side operations, while registers-instantiating processes only have the clock and/or reset signal in their sensitivity list.





Note that the register processes may include some combinational logic, but all effects of that logic must end up in the D-input of a register instantiated in the process. The inputs should not feed non-registered output logic. If it did, the output logic wouldn't simulate correctly, since its inputs are not in the process's sensitivity list.

There are only 5 fundamental VHDL building blocks that need to be used inside processes:

1. If statement
2. For statement
3. Case statement
4. Signal assignment precedence
5. Use of variables for structuring

"if", "for" and "case"
The VHDL "if" and "case" statements are used to build control logic for "*multiplexer-like*" structures. The assignment operators (<=, :=) specify data inputs to the mux structures. The combinational process "control inputs" are all built from the condition logic in VHDL if and case statement, while the data inputs are derived from the right hand side of assignment operators. A very simple illustration of this idea is the following 2 input mux statement.

```
if (sel = '0') then
  z<= a;
else
  z<= b;
end if;
```

Applying this case to the previous fig., sel is the only control input, and a and b are the data inputs while z is the output.

SIGNAL ASSIGNMENT PRECEDENCE

VHDL signal assignment precedence, is useful for improving code readability and simplicity. The precedence rule says "the last assignment made to a signal during process execution will override all previous assignments".
As an example, consider a controller:

```
case (state) is
when idle =>
   if (cond_a) then
     next_state <= state_a;
    end if;
when state_a =>
   if (cond_a) then
     do_1 <= di_1;
   elsif (cond_b) then
     do_2 <= di_2;
   end if;
   …
   next_state <= state_b;
   …
end case;
```

and suppose that the system has a shutdown signal that says to terminate all processing. It is easy to add one to the if statement at the bottom of the process that accomplish the shutdown.

```
if (cond_shutdown) then
  do_1 <= '0';
  do_2 <= '0';
  next_state <= idle;
end if;
```

The alternative to this strategy would be to include “condition strategy” in many of the conditions of the state machine’s case and if statements. VHDL signal assignment precedence provides a simpler solution, and it usually makes the code easier to read.

USE OF VARIABLES FOR STRUCTURING

The last of the five important VHDL building blocks is the idea of using variables to help structure the code. The synthesizer does what the code tells it to do. If the designer uses the expression (a+b) three different places in the code, the synthesizer will allocate three adders when mapping the logic. The designer can help the synthesizer map logic with the use of variables. To do this, a variable is usually assigned near the top of a process, then it is used throughout the rest of the process whenever the variable’s result is needed. Structuring can be done for all kind of resource-consuming logic: adders, comparators, complex logical equations, etc.

BIBLIOGRAPHY

Bhasker, J. - A VHDL synthesis primer - Star Galaxy Publishing - 1996
Perry D.L. - VHDL - McGraw Hill - 1993 - 2nd edition
Smith D.J. - HDL chip design - Doone Publications - 1996
Tanenbaum A.S. - Structured Computer organization - Prentice Hall - 1991 - 3rd edition.
Synopsys - RTL re-coding tricks for high performance design - 1997 -7th annual SNUG conference
Synopsys - VHDL coding guidelines for high performance - 1997
Warmke D.D.- Four white papers on VHDL design - 1993 - Integrated system design