UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

Fundamental concepts of VHDL

A.Carini – Progettazione di sistemi elettronici

VHDL

- VHDL is an Hardware Description Language.
- It allows the description, simulation and synthesis of digital electronics systems.
- First of all, it allows the *description* of the *structure* and of the *function* of the system and of its subsystems.
- It allows *simulation* for test and verification of the design and the comparison of different design solutions. We can verify a priori the design, without the delay and cost of HW prototypes.
- It allows the *hardware synthesis* from an abstract VHDL description, drastically reducing the design time and the time-to-market.
- It transforms modelling and design of a digital system in a coding exercise, i.e., in the design, the implementation, and the verification of a SW code.

History of VHDL

- Developed by an US research project for the *Very High Speed Integrated Circuits (VHSIC).*
- VHDL means *VHSIC Hardware Description Language.*
- Further developed under the tutelage of *IEEE*.
- Standardized by IEEE in 1987 (IEEE Standard 1076) (VHDL-87).
- Novel revisions standardized in 1993 (VHDL-93), 2001 (VHDL 2002), and 2008 (VHDL 2008).
- Historically, it was first used for description and simulation of digital electronic systems.
- Only later it was also used for synthesis.

Modeling digital systems

- VHDL is for writing models of digital systems.
- The concept of digital system is extremely wide.
- With the term digital systems, we intend *all those circuits that process or store information in digital form*.
- We will consider both the digital system as a whole and the parts that compose it. Our range of digital systems goes from the basic logic gates up to the high level functional blocks.
- To manage a complex design we need a *systematic design methodology*.
- Start from the specification document, obtain an abstract description of the system, decompose the abstract system in subsystems, the subsystems in subsubsytems, etc., until a level of basic blocks.
- The result of this process is a *hieratically composed system*.

Modeling digital systems

- In a hieratically composed system, *every subsystem can be separately designed*.
- At each step, we need only the information necessary to design our subsystem and we can ignore the information necessary to design all other subsystems.
- The term ***model*** indicates a *description of the system*, which represents the relevant information and avoids any redundant information (irrelevant in that moment).
- Since different information is relevant in different contexts, *for the same system there can be different models*.
- One model could focus on representing the system function, another could represent how the system is composed of subsystems.

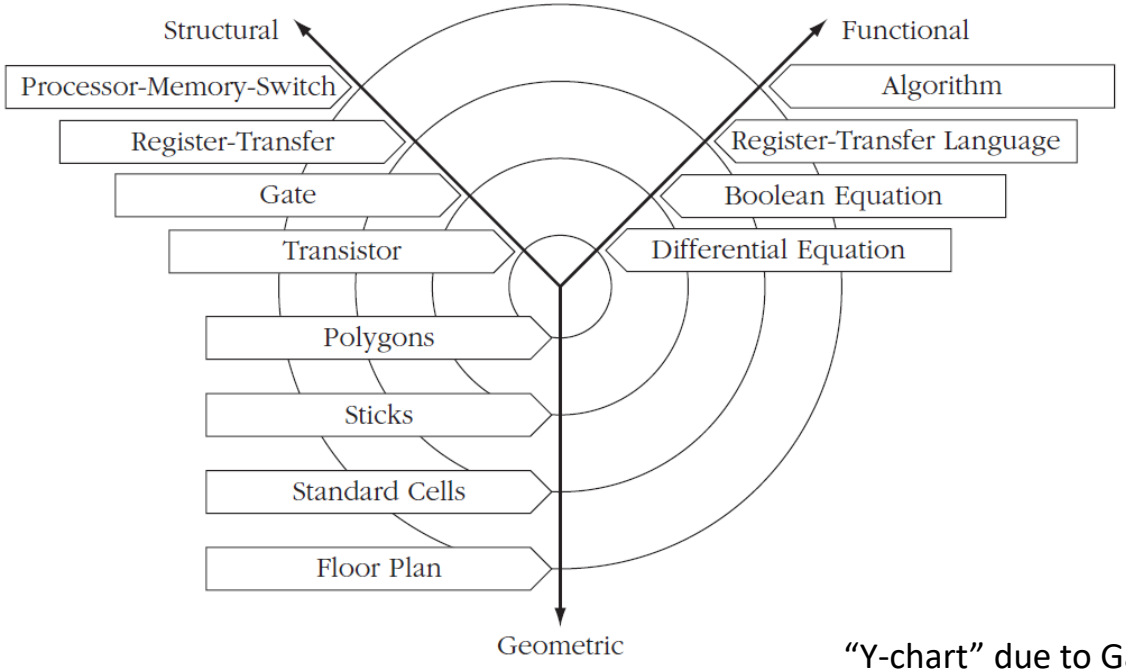
Reasons for modeling

- There are many motivations for formalizing this idea of a model:
  1. Requirements specification
  2. Documentation
  3. Testing using simulation
  4. Formal verification
  5. Synthesis
- The final goal is to achieve maximum reliability in the design process, with minimum cost and design time.
- We must guarantee that specifications are clearly provided and understood, that the subsystems work in the right way, and that designs meet the requirements.
- More importantly we want to *avoid design errors* !

Domains ad level of modeling

- There exist different models for the same system.
- We can classify these models in three domains:
  - *Functional*
    - Concerned with the operation performed by the system.
  - *Structural*
    - Concerned with how the system is composed by subsystems.
  - *Geometric*
    - Concerned with how the system is laid out in the physical space.
- Each of these domains can be divided in different *abstraction levels*.
- At the highest level we have a general description of the function, the structure, or the geometry of the model.
- Moving to lower levels we obtain a more detailed description.

Domains ad level of modeling



"Y-chart" due to Gajski & Kahn

Domains and levels of abstraction. The radial axes show the three different domains of modeling. The concentric rings show the levels of abstraction, with the more abstract levels on the outside and more detailed levels toward the center.

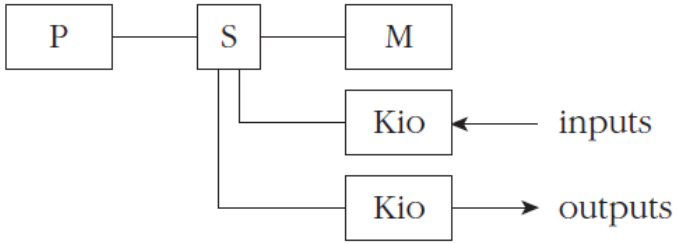
Algorithm level

- Let us consider the example of a single-chip microcontroller system used as a controller for a measurement system.

```
loop
    for each data input loop
        read the value on this input;
        scale the value using the current scale factor for this input;
        convert the scaled value to a decimal string;
        write the string to the display output corresponding to this input;
    end loop;
    wait for 10 ms;
end loop;
```

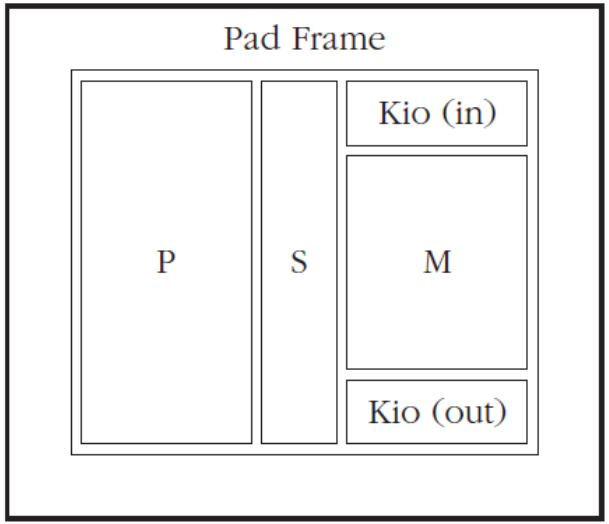
An algorithm for a measurement instrument controller.

Processor-Memory-Switch level



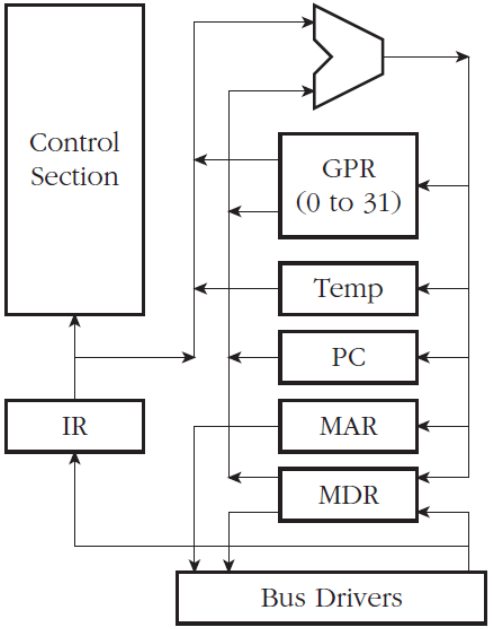
A PMS model of the controller structure. It is constructed from a processor (P), a memory (M), an interconnection switch (S) and two input/output controllers (Kio).

Floor plan level



A floor plan model of the controller geometry.

Register transfer level



A register-transfer-level structural model of the controller processor. It consists of a general-purpose register (GPR) file; registers for the program counter (PC), memory address (MAR), memory data (MDR), temporary values (Temp) and fetched instructions (IR); an arithmetic unit; bus drivers and the control section.

Register transfer language level

```
MAR ← PC,  memory_read ← 1
PC ← PC + 1
wait until ready = 1
IR ← memory_data
memory_read ← 0
```

Standard cell level

- The description at this level depends on the physical implementation.
- If a standard cell library is used to implement registers and transformation units, these entities could be placed in the areas of the floor plan.

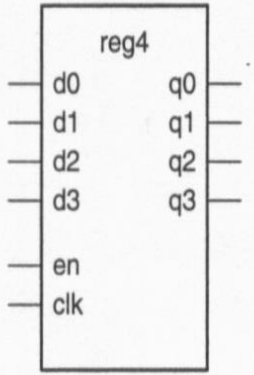
Logical and physical abstraction levels

- The next abstraction level is the *logical* one, where the structure is modelled gates interconnections, the function with Boolean equations or truth tables, and the geometry with a virtual grid, or “sticks” notation.
- The lowest abstraction level is the *physical* level. We can model the structure using transistors, the functions using the differential equations that link currents and voltages in the circuit, the geometry using the polygons of each mask layer of the integrated circuit.

VHDL model language

- It allows to describe the digital systems both from the *structural* and *behavioral* viewpoints.
- It provides also an *attribute* mechanism that can be used to provide information about the *geometry* of the model.
- It was developed for providing specifications and allowing simulation.
- It allows the automatic hardware synthesis of an RTL description.

Entity declaration



A four-bit register module. The register is named reg4 *and has six inputs,* d0, d1, d2, d3, en *and* clk, *and four outputs,* q0, q1, q2 *and* q3.

```
entity reg4 is
    port ( d0, d1, d2, d3, en, clk : in bit;
           q0, q1, q2, q3 : out  bit );
end entity reg4;
```

A VHDL entity description of a four-bit register.

Architecture body

- The architecture body describes an implementation of an entity.
- There can be several architecture body for the same entity.
- We can have :
  - *Behavioral* architecture bodies.
  - *Structural* architecture bodies.
  - *Mixed* architecture bodies.

Behavioral architecture

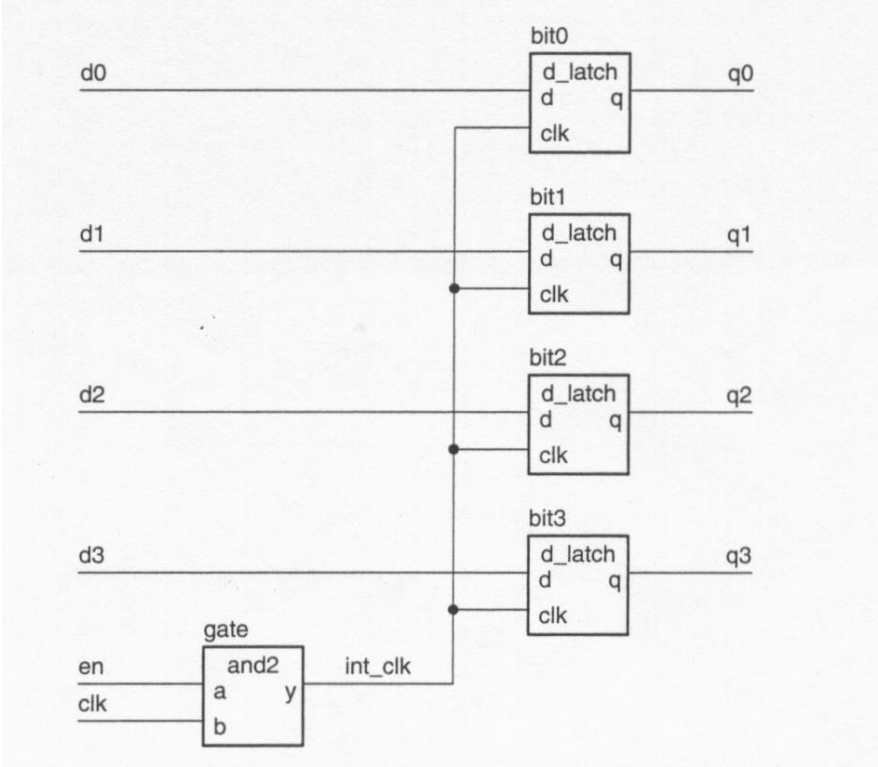
```
architecture behav of reg4 is
begin
    storage : process is
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
        if en = '1' and clk = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;
        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
        wait on d0, d1, d2, d3, en, clk;
    end process storage;
end architecture behav;
```

A behavioral architecture body of the reg4 *entity.*

Structural architecture

- It implements the module as a composition of subsystems
- It contains:
  - *signal declarations*, for internal interconnections
    - the entity ports are also treated as signals
  - *component instances*
    - Instances of previously declared entity/architecture pairs
  - *port maps in component instances*
    - connect signals to component ports.

Structural architecture



A structural composition of the reg4 *entity.*

Structural architecture

- Entity-architecture pairs of the components:

```
entity d_latch is
    port ( d, clk : in bit;  q : out bit );
end d_latch;

architecture basic of d_latch is
begin
    latch_behavior : process is
    begin
        if clk = '1' then
            q <= d after 2 ns;
        end if;
        wait on clk, d;
    end process latch_behavior;
end architecture basic;
```

```
entity and2 is
    port ( a, b : in bit;  y : out bit );
end and2;

architecture basic of and2 is
begin
    and2_behavior : process is
    begin
        y <= a and b after 2 ns;
        wait on a, b;
    end process and2_behavior;
end architecture basic;
```

Structural architecture

```
architecture struct of reg4 is
    signal int_clk : bit;
begin
    bit0 : entity work.d_latch(basic)
        port map (d0, int_clk, q0);
    bit1 : entity work.d_latch(basic)
        port map (d1, int_clk, q1);
    bit2 : entity work.d_latch(basic)
        port map (d2, int_clk, q2);
    bit3 : entity work.d_latch(basic)
        port map (d3, int_clk, q3);
    gate : entity work.and2(basic)
        port map (en, clk, int_clk);
end architecture struct;
```

A VHDL structural architecture body of the reg4 *entity.*

Mixed architecture

- The architecture can contain both behavioral and structural parts
  - process statements and component instances, collectively called *concurrent statements* since they are executed concurrently during simulation.
- Signals can be assigned to the ports of the components or can be read and written by processes.

Mixed architecture

```
entity multiplier is
    port ( clk, reset : in bit;
           multiplicand, multiplier : in integer;
           product : out integer );
end entity multiplier;
-------------------------------------------------
architecture mixed of multiplier is
    signal partial_product, full_product : integer;
    signal arith_control, result_en, mult_bit, mult_load : bit;
begin -- mixed
    arith_unit : entity work.shift_adder(behavior)
        port map ( addend => multiplicand, augend => full_product,
                   sum => partial_product,
                   add_control => arith_control);
    result : entity work.reg(behavior)
        port map ( d => partial_product, q => full_product,
                   en => result_en, reset => reset);
    multiplier_sr : entity work.shift_reg(behavior)
        port map ( d => multiplier, q => mult_bit,
                   load => mult_load, clk => clk);
    product <= full_product;
    control_section : process is
        -- variable declarations for control_section
        -- . . .
    begin -- control section
        -- sequential statements to assign values to control signals
        -- . . .
        wait on clk, reset;
    end process control_section;
end architecture mixed;
```

Test bench

- *Test benches* are used for testing a design by simulation.
- A test bench is an architecture body that
  - includes an instance of the design under test;
  - applies sequences of test values to inputs;
  - monitors values on output signals
    - either using simulator
    - or with a process that verifies correct operation.
- The entity is *self-contained*, it has no port.

Test bench

```
entity test_bench is
end entity test_bench;
-----------------------------------------
architecture test_reg4 of test_bench is
    signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
begin
    dut : entity work.reg4(behav)
        port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
    stimulus : process is
    begin
        d0 <= '1'; d1 <= '1'; d2 <= '1'; d3 <= '1';
        en <= '0'; clk <= '0';
        wait for 20 ns;
        en <= '1'; wait for 20 ns;
        clk <= '1'; wait for 20 ns;
        d0 <= '0'; d1 <= '0'; d2 <= '0'; d3 <= '0'; wait for 20 ns;
        en <= '0'; wait for 20 ns;
        . . .
        wait;
    end process stimulus;
end architecture test_reg4;
```

Design processing

- *Simulation* involves:
  - Analysis
  - Elaboration
  - Execution
- *Synthesis* involves:
  - Analysis
  - Elaboration
  - Physical synthesis

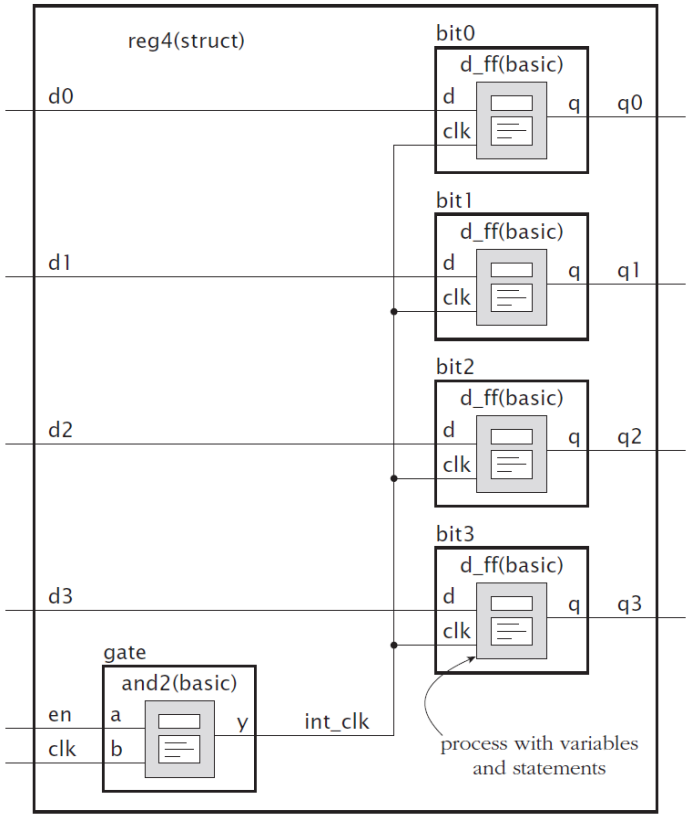
Analysis

- Checks entity-architecture pairs for syntax and semantic errors
  - syntax: grammar of the language
  - semantics: the meaning of the model
- Analyzes each *design unit* (i.e., entity/architecture pair) separately
  - Sometimes it is better to keep each design unit in a separate file
- Analyzed design units are placed in a *library*
  - They have a tool dependent internal form,
  - The current work library is called **`work`**.

Elaboration

- Originates a fully behavioral description of the entity.
- A top-down procedure:
  - Start from the top architecture.
  - Replace all components instances with the corresponding architecture bodies.
  - Repeat the procedure on these bodies, until we have only behavioral architectures.
- Result:
  - The final system is described only by signals and by processes that read and assign these signals.
  - A description that can be simulated.

Elaboration



The elaboration of the reg4 *entity using the structural architecture body. Each instance of the* d_ff *and* and2 *entities is replaced with the contents of the corresponding* basic *architecture. These each consist of a process with its variables and statements.*

Execution

- Execution/simulation of the processes in the elaborated model
- Discrete event simulation
  - time advances in discrete steps
  - when signal values change—on *events*
- Processes are sensitive to events on input signals
  - those specified in wait statements;
  - They resume and assign/schedule new values on output signals
    - they schedule *transactions;*
    - *event* on a signal if the new value is different from old one.

Execution

- Start with an initialization phase
  - each signal is given its initial value
  - simulation time set to 0
  - for each process
    - activate
    - execute until a wait statement, then suspend
      - execution usually involves scheduling transactions on signals for later times.

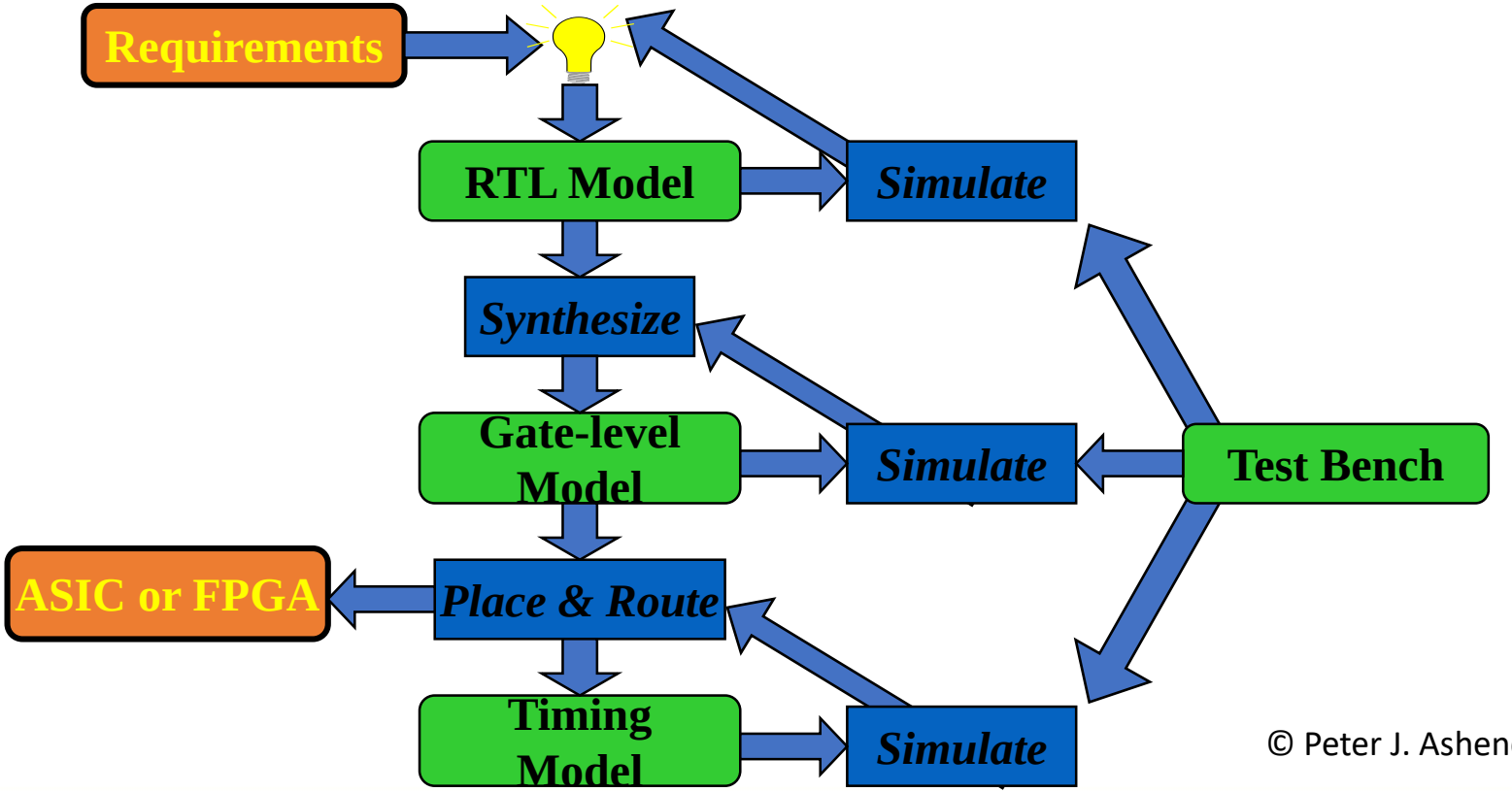
Execution

- Simulation cycle
  - simulation time advances to the time of the 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
    - resume
    - execute until a wait statement, then suspend.
- Simulation finishes when there are no further scheduled transactions.

Synthesis

- It translates a register-transfer-level (RTL) design into a gate-level netlist.
- There are restrictions on coding style for the RTL model.
- The result is tool dependant.

Basic Design Methodology



© Peter J. Ashenden

Comments

- Start with a couple of dashes, i.e. - -, and end at the end of the row.
- They are similar to // of C++.
- Comments on different lines should always start with "- -".

```
-- The following code models
-- the control section of the system
. . . some VHDL code . . .
```

- Since VHDL 2008, also `/* comment */`

```
/* This is a comment header that describes
   the purpose of the design unit. It contains
   all you ever wanted to know, plus more.
*/

entity thingumy is
  port ( clk   : in bit; -- keeps it going
         reset : in bit /* start over */
         /* other ports to be added later */ );
end entity thingumy;
```

Basic identifiers

- For naming entities, architectures, variables, signals, labels, etc.
- Can be arbitrary long.
- Can contain only letters (from ‘A’ to ‘Z’, from ‘a’ to ‘z’), decimal digits (from ‘0’ to ‘9’) and the *underscore* character (‘_’).
- Must start with a letter.
- Cannot end with *underscore.*
- Cannot include two consecutive *underscores*.
- They are NOT *case sensitive.*

Examples of basic identifiers

- Allowed:

```
A   X0   counter   Next_Value   generate_read_cycle
```

- Not allowed:

```
last@value      -- contains an illegal character for an identifier
5bit_counter    -- starts with a non-alphabetic character
_A0             -- starts with an underline
A0_             -- ends with an underline
clock__pulse    -- two successive underlines
```

Extended identifiers

- Can include any sequence of characters.
- Used for interfacing with other tools.
- Written between two *backslash*.

```
\data bus\  \global.clock\  \923\  \d#1\  \start__\
```

- The character '\' inside a string must be double:

```
\A:\\name\    -- contains a '\' between the ':' and the 'n'
```

- They are case sensitive and distinct from basic identifiers:

```
name  \name\  \Name\  \NAME\
```

Reserved words

abs	default	label	package	sla
access	disconnect	library	parameter	sll
after	downto	linkage	port	sra
alias		literal	postponed	srl
all	else	loop	procedure	strong
and	elsif		process	subtype
architecture	end	map	property	
array	entity	mod	protected	then
assert	exit		pure	to
assume		nand		transport
assume_guarantee	fairness	new	range	type
attribute	file	next	record	
	for	nor	register	unaffected
begin	force	not	reject	units
block	function	null	release	until
body			rem	use
buffer	generate	of	report	
bus	generic	on	restrict	variable
	group	open	restrict_guarantee	vmode
case	guarded	or	return	vprop
component		others	rol	vunit
configuration	if	out	ror	
constant	impure			wait
context	in		select	when
cover	inertial		sequence	while
	inout		severity	with
	is		shared	
			signal	xnor
				xor

Special symbols

- Used as operators, separators, terminators.
- Some special symbols are formed by two characters.

```
"  #  &  '  (  )  *  +  -  ,  .  /  :  ;  <  =  >  ?  @  [  ]  `  |

=>  **  :=  /=  >=  <=  <>  ??  ?=  ?/=  ?>  ?<  ?>=  ?<=  <<  >>
```

Numbers

- The numbers you write in VHDL are:
- *Integers*
  - Are always positive
  - Cannot have the decimal point
- *Reals*
  - Have always a decimal point preceded and followed by a digit.
- Example:

```
23   0   146
```

```
23.1   0.0   3.14159
```

Numbers

- An exponential notation is possible:

```
46E5   1E+12   19e00
1.234E09   98.6E+21   34.0e–08
```

- It is possible a notation in any base between 2 and 16:

```
2#11111101#   16#FD#   16#0fd#   8#0375#
```

253

```
2#0.100#   8#0.4#   12#0.6#
```

0.5

- Also with an exponential notation:

```
2#1#E10   16#4#E2   10#1024#E+00
```

Numbers

- To improve the readability of a number the underscore character ‘_’ can be used:

```
123_456   3.141_592_6   2#1111_1100_0000_0000#
```

Single characters

- Are written between single quotation marks.

```
'A'     -- uppercase letter
'z'     -- lowercase letter
','     -- the punctuation character comma
''''    -- the punctuation character single quote
' '     -- the separator character space
```

Strings

- Are written between double quotation marks.
- Can have any length but must stay on a row.
- In order to split them on two (or more) rows, the concatenation operator **&** can be used.

```
"If a string will not fit on one line, "
& "then we can break it into parts on separate lines."
```

- Any double quotation mark inside a string must be doubled:

```
"A string in a string: ""A string"". "
```

Bit strings

- Represent sequences of bits.
- Are given by a sequence of digits between double quotation marks preceded by a base specifier:
  - B (or b) for binary
  - O (or o) for octal
  - X (or x) for hexadecimal
  - D (or d) for decimal

```
B"0100011"  B"10"  b"1111_0010_0001"  B""

O"372"   -- equivalent to B"011_111_010"
o"00"    -- equivalent to B"000_000"

X"FA"    -- equivalent to B"1111_1010"
x"0d"    -- equivalent to B"0000_1101"

D"23"      -- equivalent to B"10111"
D"64"      -- equivalent to B"1000000"
D"0003"    -- equivalent to B"11"
```

Bit strings

- Since VHDL 2008, it is possible to insert in the bit-string also the characters Z, X, -, used for high-impedance state, unknown value, don't-care condition, and other similar characters.
- The character is repeated two, three, four times according to the base.

```
O"3XZ4"    -- equivalent to B"011XXXZZZ100"
X"A3--"    -- equivalent to B"10100011--------"
X"0#?F"    -- equivalent to B"0000####????1111"
B"00UU"    -- equivalent to B"00UU"
```

- It is also possible to specify if the bit-string specify a signed or unsigned by preceding the base with the S or U character, respectively.

Syntax description with EBNF notation

- The *Extended Backus-Naur Form* notation will be used to provide the syntax rules of VHDL.
- The EBNF divides the language in syntactic categories.
- A rule will be written for each syntactic category.
- Note the use of the symbol ⇐ which is read “is defined as” or “is given by”.

$$\text{variable\_assignment} \Leftarrow \text{target} := \text{expression} ;$$

Syntax description with EBNF notation

- Optional parts will be indicated with void square brackets:

function_call ⇐ name ⟦ (association_list) ⟧

- Parts that can be repeated zero, one or more times will be reported between curly brackets:

process_statement ⇐
process is
{ process_declarative_item }
begin
{ sequential_statement }
end process ;

Syntax description with EBNF notation

- To indicate that a part must be repeated one or more times:

```
case_statement ⇐
    case expression is
        case_statement_alternative
        { ... }
    end case ;
```

- and if there is a separator:

```
identifier_list ⇐ identifier { , ... }
```

Syntax description with EBNF notation

- A choice between different alternatives is written as:

mode ⇐ **in** ▯ **out** ▯ **inout**

- In case of ambiguity the void brackets will be used to group some terms:

term ⇐ factor { (* | / | **mod** | **rem**) factor }

Instead of:

term ⇐ factor { * | / | **mod** | **rem** factor }

Syntax description with EBNF notation

- To provide more information about some categories, some italic comments may be added to the category name:

function_call ⇐ *function*_name [[(*parameter*_association_list)]]

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 1