VHDL language, from specification to model

Very high speed integrated circuits
Hardware Description Language

History of the VHDL Language

- VHDL (IEEE 1076-1987) was born in 1987 from the joint efforts of:
  - IEEE ->
    - Computer Society ->
      - Design Automation Technical Committee ->
        - Design Automation Standards Subcommittee ->
          - VHDL Analysis and Standardization Group
    - CAD Language Systems Inc.
- VHDL 4.0 (IEEE 1076-2008) is the most recent revision
- Some companion standards:
  - VITAL
  - Synthesis
  - AMS
  - etc.

Warning

- This course presents VHDL revision 2002
  - Some modifications in the following revisions of the standard may be contradictory with this course
  - Please read the IEEE Language Reference Manuals (LRMs) for more information
- The most important reason for this choice is that some tools still don’t implement 100% of VHDL-2008
- This course presents VHDL for synthesis. A lot of features that are very important for modeling are omitted
Agenda

- Introduction
- Principles of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

Exercise: how can we do?

- Non determinism is an issue
- The execution order of the different processes shall not impact the simulation results
  - But if the output of P1 goes to P2...
  - ... and P1 runs first, then P2...
  - ... the result will be different from running P2 first then P1

Simulating parallel systems on sequential computers

- Parallelism is needed
- Non-determinism is an issue:
  - Let's introduce a new kind of variable, dedicated to communication between sequential programs (processes): the signal
  - When executing an assignment statement the value of the signal is not affected
  - The value of the signals is modified once every process was executed (after 1 $\Delta$)
The symbolic time

- Used to specify dependencies between events, that is, to order events
- Needed to distinguish the cause from the effect
- Is the only one supported by logic synthesizers

**Wait until A = 25;**

**B <= 3; -- (after Δ)**

**Wait until B = 3;**

**C <= 12; -- (after Δ)**

The symbolic time is used to specify dependencies between events, order events, distinguish the cause from the effect, and is supported by logic synthesizers.

Simulating Parallel Systems on Sequential Computers

- Sequential programming is still needed
- Classical variables still exist inside the processes
- Variable assignment is immediate
- Processes run in a very classical way

VAR1 := 1;
VAR2 := REG;
VAR3 := TEN;

Simulating Parallel Systems on Sequential Computers

Physical time

- Physical events occur at a physical time
- We want to model physical events too (for simulation, not for logic synthesis)
- The physical time must be modeled
- Not supported by logic synthesizers

Physical events occur at a physical time, need to be modeled, and are not supported by logic synthesizers.

The signal and its driver

Current value
List of scheduled events

A <= 25 after 1 hr;

A <= 25 after 1 hr;

A <= 47 25 @ T + 1 h

if (A = 25) then

if (A = 25) then

B <= 3;
Synchronization between processes

- Delta cycles have no physical duration
- So the physical time cannot increase during simulation!

- For most processes incremental step by step execution is very inefficient
- How to run processes when and only when it is needed?

- At each simulation step the simulator resumes only the processes which inputs changed
- So it must be able to identify which signals are an input of any particular process...
- ... and decide whether they changed since the last execution of the process (or since the last simulation step)

Warning:

The simulation engine

- The time advances when the process is suspended on one of its synchronization points
- Between two synchronization points the time is constant
- This is the “Zero-Time” execution
- Signal assignment is delayed
- The process is an infinite loop
Example

entity REGS is
port (CK, DIN : in BIT; DOUT : out BIT);
end entity REGS;

architecture ARC of REGS is
signal A0, A1 : BIT;
begin
REGS_PR : process
begin
wait on CK;
if (CK = '1') then
A0 <= DIN; A1 <= A0; DOUT <= A1;
end if;
end process REGS_PR;
end architecture ARC;

Example

entity REGS is
port (CK, DIN : in BIT; DOUT : out BIT);
end entity REGS;

architecture ARC of REGS is
signal A0, A1 : BIT;
begin
REGS_PR : process
begin
if (CK = '1') then
A0 <= DIN; A1 <= A0; DOUT <= A1;
end if;
wait on CK;
end process REGS_PR;
end architecture ARC;

Event-driven simulation step by step

architecture SIM of INC is
signal CK : Bit;
signal D, Q: Natural;
begin
P1: process
begin
CK <= '0';
wait for 10 ns;
CK <= '1';
wait for 10 ns;
end process P1;
P2: process
begin
wait on Q;
D <= Q+1 after 15 ns;
end process P2;
P3: process
begin
wait on CK;
if (CK = '1') then
Q <= D;
end if;
enend process P3;
end architecture SIM;

Let’s simulate
Event-driven simulation step by step

architecture SIM of INC is
signal CK: Bit;
signal D, Q: Natural;
begin
P1: process
begin
  CK <= '0';
  wait for 10 ns;
  CK <= '0';
  wait for 10 ns;
end process P1;
P2: process
begin
  D <= Q+1 after 15 ns;
  wait on Q;
end process P2;
P3: process
begin
  if (CK = '1') then
    Q <= D;
  end if;
  wait on CK;
end process P3;
end architecture SIM;

Event-driven simulation step by step

architecture SIM of INC is
signal CK: Bit;
signal D, Q: Natural;
begin
P1: process
begin
  CK <= '0';
  wait for 10 ns;
  CK <= '0';
  wait for 10 ns;
end process P1;
P2: process
begin
  if (CK = '1') then
    Q <= D;
  end if;
  wait on CK;
end process P2;
P3: process
begin
  Q <= D;
end process P3;
end architecture SIM;

Agenda

- Introduction
- Principles of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

Structure of the language

- To simulate we need:
  - Analysis (compilation) of source files
  - Elaboration (link) of compilation results
  - We always simulate the result of an elaboration
- The result of an analysis or an elaboration is stored in a library
- The content of an existing library may be used in another program (after the proper declaration)
Structure of the language

- 5 compilation units:
  - Entity
  - Architecture
  - Package declaration
  - Package body
  - Configuration
- Only the 5 compilation units can be compiled (analyzed)
- Only the result of the compilation of an architecture or a configuration can be elaborated (linked)

Libraries

- The symbolic name WORK designates the target library of a compilation (the library in which the result of the compilation will be stored)
- To access a library it must first be declared:
  - library LIB;
    use LIB.PAQ.OBJ;
- Creation and management of the libraries are not defined in the standard, they are tool-dependant. Every environment has its own organization
- Libraries may be shared between users

Agenda

- Introduction
- Principles of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

The entity

- It’s the interface specification. It provides:
  - The module name (FA)
  - Its input-output ports:
    - Name
    - Direction (in, out, inout, ...)
    - Type (BIT, BIT_VECTOR, BOOLEAN, INTEGER, ...)
- The ports are visible and usable as signals inside the associated architecture. They must not be re-declared in the architecture
The architecture

- It’s the internal description. It’s always associated with its entity.
- A single entity may be associated with several architectures.

Structure of the language

- VHDL is a heavily declarative language:
  - Every object must be declared before usage:
    - Variable
    - Signal
    - Constant
    - Function
    - Procedure
    - Component
    - ...
  - There are dedicated declaration area
  - One cannot declare anything anywhere...

The architecture

- Declaration areas
- Body (concurrent instructions)
  - Process
  - Concurrent signal assignment
  - Entity instantiation...
- Parallel execution, file order not relevant

The architecture

- The ports of the associated entity are visible and usable as signals inside the associated architecture. They must not be re-declared in the architecture
- Input ports (in) are read-only
- Output ports (out) are write-only
The package

- It is a collection of reusable “things”
- It is made of two compilation units:
  - Package declaration
  - Package body
- The content of the package declaration is “visible” from another compilation unit if it declared its use (public part)
- The package body is “invisible” from the other compilation units (private part)

The package declaration

```
package PAQ is
  subtype WORD is BIT_VECTOR(7 downto 0);
  type ANYRAM is array(NATURAL range <>) of WORD;
  subtype TRAM is ANYRAM(0 to 1023);
  constant NBITS: POSITIVE := 8;
  constant VERSION: NATURAL := 8;
  component K
    generic(N: POSITIVE := 12);
    port(A, B: in BIT; S: out BIT);
  end component;
  function MAX(A, B, C: INTEGER) return INTEGER;
end package PAQ;
```

The package body

```
package body PAQ is
  constant NBITS: POSITIVE := 32;
  function MAX(A, B: INTEGER) return INTEGER
  is
    begin
      if (A>B) then
        return A;
      else
        return B;
      end if;
    end MAX;
  function MAX(A, B, C: INTEGER) return INTEGER
  is
    return MAX(A, MAX(B,C));
  end package body PAQ;
```

Hierarchical design

- To build a design from sub-designs
- Instantiate entity – architecture pair
- Wire them together
- It is the structural description style (vs. behavioral)

```
entity AND3 is
  port(A0, A1, A2: in BIT; Z: out BIT);
end entity AND3;
architecture STR of AND3 is
  signal TMP: BIT;
begi
  I0: entity WORK.AND2(CMP)
    port map(A => A0, B => A1, C => TMP);
  I1: entity WORK.AND2(CMP)
    port map(A => A2, B => TMP, C => Z);
end architecture STR;
```
Agenda

- Introduction
- Principles of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

The process

- A process is a sequential program
- Every object it manipulates has a type
- It manipulates objects with operators
- Its control flow is specified by control structures

Processes with sensitivity lists

- A process may have a sensitivity list
- The sensitivity list is a list of signals
- It is the only synchronization point of the process

Exercise: examples of processes

- This process models a combinatorial function of signals X and Y. What function? As soon as X or/and Y changes the function is re-executed. Write a combinatorial process implementing the majority function of 3 signals. Same question with the full adder

- This process is a synchronous one. It models the behavior of a D-flip-flop (DFF). What signal is the clock? The input? The output? Explain the behavior of this process. Write a process modeling a DFF on rising edge of its clock and with asynchronous, active low, reset.
Comments, identifiers, literals, ...

- Comments start with a double dash (--) and extend until end of line (no multi-line comment /* ... */)
- Identifiers are sequences of letters, digits and underscores (_). They must start with a letter. VHDL is case insensitive
- Literals are constant explicit values:
  - 45 and 7.89 are numeric literals
  - "this is a string of characters "
  - 'C' is a character literal
  - "000111010110" can be a string or a bit-string literal
  - B"000111010110", O"726" and X"1E6" are bit-string literals
- null is an access (pointer) literal
- Expressions are terminated by a semicolon (;)

Kinds of value containers

- Value containers can be one of three kinds:
  - Variables, very similar to variables in any other programming language, they are dedicated to classical sequential programming (inside processes)
  - Constants, similar too to what is found in other languages
  - Signals, the VHDL originality, dedicated to parallel programming and, more precisely, to the exchanges between several programs running in parallel
- In order to avoid common mistakes assignments are denoted in different ways depending on the kind of container:
  - A := 178 for variables and constants
  - S <= 178 for signals

Initialization of variables and signals

- A variable or a signal is initialized at the beginning of the simulation (time zero). Its default initialization value is the leftmost value of the declaration of its type:
  - type T is (RED, GREEN, BLUE);
  - variable V: T; -- Initialization value of V is RED
- It is possible to declare another initialization value when declaring a variable or a signal:
  - signal S: INTEGER := 0;
  - variable V: BOOLEAN := TRUE;
- Very often it’s a bad idea because it may hide real “reset” defaults

The types

- Signals, variables and constants always have a type
  - Variable V: BIT_VECTOR(1 to 10);
  - Variable M: BIT_VECTOR(7 downto 0);
The integer types

- The type Integer is built from the integer type of the host CPU. The LRM requires that its length is larger or equal to 32 bits. Subtypes may be defined.
  - type INTEGER is range CPU_DEPENDENT;
- The types NATURAL and POSITIVE are range subtypes, (range) of the same base type Integer
  - subtype NATURAL is INTEGER range 0 to INTEGER'HIGH
  - subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
- A subtype inherits the properties of its base types. Compatibility errors may occur during assignment
- The type attribute INTEGER'HIGH represents the largest element of type Integer. Its value is CPU-dependent. 'HIGH is a type attribute, as in ADA

The enumerated types

- An enumerated type is a type with an exhaustive definition by enumeration:
  - type COLORS is (RED, YELLOW, BLUE, GREEN, ORANGE);
  - type FOUR_STATES is ('X', '0', '1', 'Z');
  - type STD_ULOGIC is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
- The order in which an enumerated type is declared is meaningful. Example: a signal or a variable of enumerated type T takes T'LEFT as default initialization value

Predefined enumerated types

- type BOOLEAN is (FALSE, TRUE);
- type BIT is ('0', '1');
- type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
The array types

- Array types are collections of identical objects indexed by ranges of integer or enumerated types.
- Example:
  - `type BUS is array (0 to 31) of BIT;
  - `type RAM is array (0 to 1023, 0 to 31) of BIT;
  - `type PRICE is range 0 to INTEGER'HIGH units
    penny;
    nickel = 5 penny;
    dime = 2 nickel;
    quarter = 5 nickel;
    half = 2 quarter;
    dollar = 2 half;
    end units;
  - `type COLORS is (WHITE, BLUE, GREEN, RED, YELLOW, BLACK, RAINBOW);
  - `type PAINTINGS_PRICES is array (COLOR range WHITE to BLACK) of PRICE;

Unconstrained arrays

- An array type may be declared with an unknown range:
  - `type BIT_VECTOR is array (NATURAL range <>)
    of BIT;
- Unconstrained array types may be used to model parameters of subprograms. But of course variable or even infinite size objects cannot exist.
  - In order to create an object of type BIT_VECTOR its actual size must be declared:
    - `subtype TYPE_BUS is BIT_VECTOR(0 to 31);
    - `variable VARIABLE_BUS1: TYPE_BUS;
    - `variable VARIABLE_BUS2: BIT_VECTOR(0 to 31);

The STRING type

- VHDL defines character strings:
  - `type STRING is array (POSITIVE range <>)
    of CHARACTER;
  - "This is a string" -- STRING
- Some literals are ambiguous and cannot be typed only by evaluating the context:
  - 'l' -- BIT or CHARACTER ?
  - B"01010101" -- BIT_VECTOR in binary form
  - O"0120764" -- BIT_VECTOR in octal form
  - X"0134DF54" -- BIT_VECTOR in hexadecimal form
  - "01010101" -- BIT_VECTOR or STRING ?
- Qualification may be used to solve the ambiguity:
  - `BIT_VECTOR'("01010101")
- Warning: qualification is not a conversion

The records

- A record is an object which elements are heterogeneous.
- Example:
  - `type OPTYPE is (ADD, SUB, MFY, DIV, JMP);
  - `type INSTRUCTION is record
    OPCODE: OPTYPE;
    SRC: INTEGER;
    DST: INTEGER;
  end record;
- VHDL has no records with variants (the C unions)
The text files

- The package TEXTIO from the library STD contains subprograms and declarations for text I/O
- One file type TEXT
- Two predefined TEXT files: INPUT and OUTPUT

```vhdl
use STD.TEXTIO.all;

type TEXT is file of STRING;

file INPUT: TEXT open READ_MODE is "STD_INPUT";
file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT";
```

```vhdl
file FOO: TEXT;
FILE_OPEN(FOO, "foo.txt");

file BAR: TEXT;
FILE_OPEN(BAR, "bar.txt", WRITE_MODE);
```

The text files

- Besides the implicitly declared functions and procedures TEXT files may be accessed line by line through LINE objects
- The READLINE and WRITELINE procedures read and write one entire line of a text file
- The READ and WRITE procedures read and write inside the line; they are defined for the types BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING and TIME

```
procedure READ(L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
procedure WRITE(L: inout LINE; VALUE: in BIT; JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
```

The type attributes

- The type attributes are used to examine already declared types
  - type COLORS is (RED, YELLOW, BLUE, GREEN, ORANGE);
  - type FOUR_STATES is ('X', '0', '1', 'Z');
- Some attributes are implicitly self-declared at type declaration. Examples:
  - T'BASE - returns the base type of type T
  - COLORS'LEFT = RED
  - COLORS'RIGHT = ORANGE
  - FOUR_STATES'HIGH = 'Z'
  - FOUR_STATES'LOW = 'X'

```
type LINE is access STRING;
```

```
procedure READLINE(F: in TEXT; L: out LINE);
procedure WRITELINE(F: out TEXT; L: out LINE);
```

Files

- Examples of use: simulation environments (reading of input patterns in a file, writing of output results in another file)

```
RENDERING: process(S)
   variable L: LINE;
   file OUTPUTS: TEXT open WRITE_MODE is "out.dat";
begin
   READLINE(INPUTS, L);
   READ(L, A);
   VA <= A;
   READ(L, B);
   VB <= B;
   wait for 20 ns;
end process READING;
```

```
WRITING: process(S)
   variable L: LINE;
   file OUTPUTS: TEXT open WRITE_MODE is "out.dat";
begin
   WRITE(L, S);
   WRITE(L, STRING(" at time "));
   WRITE(L, NOW);
   WRITELINE(OUTPUTS, L);
end process WRITING;
```

```
```
Exercise: the type attributes

What is the returned value of these 5 attributes for this subtype?

```
subtype REVERSE_COLORS is COLORS range ORANGE downto RED;
```

The array types attributes

```
type T is array (0 to 3, 7 downto 0) of BIT;
variable TAB: T;
TAB'LEFT(1) -- returns 0
TAB'LEFT(2) -- returns 7
TAB'RIGHT(1) -- returns 3
TAB'RIGHT(2) -- returns 0
TAB'HIGH(1) -- returns 3
TAB'HIGH(2) -- returns 7
TAB'LOW(1) -- returns 0
TAB'LOW(2) -- returns 0
TAB'RANGE(1) -- returns 0 to 3
TAB'RANGE(2) -- returns 7 downto 0
TAB'REVERSE_RANGE(2) -- returns 0 to 7
TAB'REVERSE_RANGE(1) -- returns 3 downto 0
TAB'LENGTH(1) -- returns 4
TAB'LENGTH(2) -- returns 8
```

Aggregate

```
type OPTYPE is (ADD, SUB, MPY, DIV, JMP);
type T is array (1 to 5) of OPTYPE;
type U is record
  R1, R2, R3: INTEGER range 0 to 31;
  OP: OPTYPE;
end record;
variable A: T; variable B: U;
... A := (ADD, SUB, MPY, DIV, JMP);
A := (ADD, SUB, MPY, 5 => JMP, 4 => DIV);
A := (3 => ADD, SUB, MPY, JMP, DIV);
A := (ADD, 2 | 4 => MPY, others => DIV);
A := (SUB, 2 to 4 => DIV, 5 => JMP);
B := (0, 1, 2, ADD);
B := (OP => JMP, others => 0);
```

Operators

Logical: and, or, nand, nor, xor, not

Relational: =, /=, <, <=, >, >=

Addition: +, -, & (concaténation)

Sign: +, -

Multipliers: *, /, mod, rem

```
A = (A / B) * B + (A rem B)
sign(A rem B) = sign(A)
abs(A rem B) < abs(B)
(-A) / B = -(A / B) = A / (-B)
exists N, A = B * N + (A mod B)
sign(A mod B) = sign(B)
abs(A mod B) < abs(B)
```

Miscellaneous:** (exponentiation), abs (absolute value)
Control structures

```vhdl
if C1 = -65 then
    A := 10;
    B := '0';
elsif C1 = -64 then
    A := 20;
    B := '1';
elsif C1 >= -63 and C1 <= -60 then
    A := 20;
    B := '0';
elsif C1 = -59 or C1 = 187 then
    A := 30;
    B := '0';
else
    A := 30;
    B := '1';
end if;
```

```vhdl
case C1 is
    when -65 => A := 10;
              B := '0';
    when -64 => A := 20;
              B := '1';
    when -63 to -60 => A := 20;
                  B := '0';
    when -61 | 187 => A := 30;
                  B := '0';
    when others => A := 30;
                  B := '1';
end case;
```

The wait instruction

- A process shall contain at least one synchronization point
  - Either implicit: a list of signals, named sensitivity list, is declared in the process header. The equivalent `wait` instruction (`wait on list`) is the last instruction of the process body. It is forbidden to put other `wait` statements in the process
  - Or explicit: no signal list in the header. The process may then contain several `wait` statements

- The complete form of the `wait` instruction:
  - `wait [on S1, S2, ...] [until CONDITION] [for DURATION]`
  - `S1, S2` must be signals
  - `CONDITION` is an expression that evaluates as a boolean
  - `DURATION` is a timeout

wait: examples

- Eternal:
  - `wait;
- No condition, no timeout:
  - `wait on S1, S2;
- No list, no timeout:
  - `wait until (S1 = '0') and (S2 > ORANGE);
- Equivalent loop:
  - `loop
      wait on S1, S2;
      exit when (S1 = '0') and (S2 > ORANGE);
  end loop;
- Warning: if the condition contains no signals the `wait` becomes eternal. Classical example:
  - `wait until NOW > 10 s;`
These two processes are equivalent

```vhdl
signal X, Y, Z: BIT;
PA: process(X, Y)
begin
  if X = '1' then
    Z <= '1';
  elsif Y = '1' then
    Z <= '1';
  else
    Z <= '0';
  end if;
end process PA;
```

- **Assert**

  - Check a property and issues a message if it is not verified. Can also stop the simulation. Used to check the proper use of a model

    ```vhdl
    assert CONDITION
    [report MESSAGE]
    [severity LEVEL];
    ```

    - **CONDITION**: Boolean condition asserted true
    - **MESSAGE**: message (string) to print in case of violation
    - **LEVEL**: assertion level from predefined type: NOTE, WARNING, ERROR, FAILURE

  - An assertion which level is ERROR or FAILURE usually stops the simulation (usually a parameter in the simulator). ERROR is the default level

- **Signal assignment**

  - For synthesis, the only supported signal assignment is the zero delay one:
    
    ```vhdl
    S <= A + B;
    ```

  - Signal assignments with delay can be used only for simulation:
    
    ```vhdl
    S <= A + B after 2 ns;
    ```

  - When executing a signal assignment the simulator updates the target signal driver (simplified algorithm for zero delay assignment):
    
    1) Delete old transactions which date is equal or greater than the date of the new transaction
    2) Add new transaction at the end of the driver

- **Exercise: signal assignment (3/3)**

  - A common pitfall is the signal assignment erasure

    Look at the following combinatorial process

    ```vhdl
    process(A, B)
    variable V: integer;
    begin
      V := A + B;
      S <= 2;
      V := V + S;
      S <= V;
    end process;
    ```
Signal attributes

- If $S$ is a signal, VHDL defines several attributes to investigate the signal’s status:
  - The attribute $S’EVENT$ is a function returning a BOOLEAN. It returns TRUE if the signal changed during the current simulation step. Example: to detect the rising edge of a clock:
    ```vhdl
    if (CLK = '1') and CLK’EVENT then
    Q <= D;
    end if;
    ```
  - The attribute $S’LAST_EVENT$ is a function returning a TIME. It returns the time elapsed since the last event on signal $S$.
  - The attribute $S’LAST_VALUE$ is a function returning a $S$. It returns the value the signal $S$ had before the last event.
  - The attribute $S’STABLE(T)$ is a signal of type BOOLEAN. Its value is TRUE if there wasn’t any event of $S$ for the duration $T$.

Exercise: design of a DFF (1/2)

- Design the process(es) modeling a DFF which ports are:
  - $RST$ is an active low (‘0’) asynchronous reset
  - $CP$ is the clock; the DFF is synchronized on the rising edge of $CP$
  - $D$ is the input
  - $Q$ is the output, $QN$ is the inverted output

The subprograms

- 2 types of subprograms:
  - Functions
    - Read only parameters
    - Return a value
  - Procedures
    - Writable parameters (modes out and inout)
    - No value returned
- Both pertain to the sequential domain
- Same usage as in every programming language

Name overloading

- In VHDL, as in ADA, several subprograms may share the same name
- The compiler identifies the right subprogram depending on:
  - The name used for the call
  - The shape of the parameters:
    - Number and type of parameters
    - Type of the returned value (for functions)
- The compiler issues an error if there are more than one or zero candidates. It usually provides the list of the considered candidates
- Operators may also be overloaded by using their functional representation:
Subprogram example

function NAT2VEC(VAL: NATURAL; SIZE: POSITIVE) return BIT_VECTOR;

function NAT2VEC(VAL: NATURAL; SIZE: POSITIVE) return BIT_VECTOR is
variable RES: BIT_VECTOR(SIZE - 1 downto 0) := (others =>'0');
variable TMP: NATURAL := VAL;
begin
for I in 0 to SIZE - 1 loop
exit when TMP = 0;
if (TMP mod 2 = 1) then RES(I) =>'1'; end if;
TMP := TMP / 2;
ext loop;
assert (TMP = 0) report "NAT2VEC: overflow" severity WARNING;
return RES;
end function NAT2VEC;

Exercise: subprograms

- Imagine the function VEC2NAT
  - VEC2NAT("001011100") = 92
  - VEC2NAT("1001001") = 73

- Write the functions PP and PG
  - PP(12, 18) = 12
  - PP("001", "110") = "001"
  - PG(7, 0) = 7
  - PG("001", "110") = "110"

Combinational processes: warning

- The sensitivity list must be complete
- All the outputs must receive a value in every execution of the process
- The best logic synthesizers issue warnings
- The violation of one of these rules will probably lead to different behaviors before and after logic synthesis
- Unwanted memory units inferred by the synthesizer are usually the indicator that one of these rules is violated

Warning: the process without synchronization Point

- It’s the most frequent error. The process has no sensitivity list and no wait statements. Warning: wait statements may be present but masked by control structures (if, case, loop...) 
- Effect: the simulation time (symbolic and physical) is stuck at 0. Nothing happens, the simulator executes the same process forever and the other processes are never executed
- Solution: stop the simulation, identify what process was running at that time and fix it
Warning: the combinational loop

- It’s a process or a collection of processes equivalent to a combinational loop with a zero physical propagation time
- Effect: the symbolic time increases very fast while the physical time is stuck. Nothing happens
- Solution: stop the simulation, identify what process was running at that time and fix it
- Note: several processes may be involved in a combinational loop

Exercise: FSM

- This is the interface and state diagram of a Moore Finite State Machine (FSM). It is synchronized on the rising edge of the clock CP. RSTN is the active low asynchronous reset; it forces the state to S0. The inputs and outputs are active high ('1'). Design the process(es) needed to model this state machine

Agenda

- Introduction
- Principals of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

Concurrent VHDL

- VHDL models are made of concurrent instructions
- The 5 concurrent instructions are:
  - Processes
  - Entity (or component) instantiations
  - Concurrent procedure calls
  - Concurrent assertions
  - Concurrent signal assignments
- Concurrent instructions execute in pseudo-parallelism, their order of appearance in the file has no impact on the behavior
Concurrent VHDL

- The body of an architecture is a set of concurrent instructions.
- A process is ONE concurrent instruction. And it is made of several sequential instructions.

```
architecture ARC of FOO is
    signal I0, I1: INTEGER;
begin
    P1: process(A, B)
    begin
        I0 <= A+B;
        I1 <= A*B;
    end process P1;
    P2: process(I0, I1)
    begin
        if(I0 /= 0) then
            S <= I1/I0;
            ALARM <= FALSE;
        else
            S <= 0;
            ALARM <= TRUE;
        end if;
    end process P2;
end architecture ARC;
```

- Alternate, simpler, forms of the process are provided:
  - Concurrent procedure calls
  - Concurrent assertions
  - Concurrent signal assignments
- These concurrent instructions look like sequential instructions but they are not
- It is very important to remember that they are simplified versions of processes (short hands)

```
architecture AR of SUM is
begin
    PR: process(A, B, CI)
    begin
        if(A = '0') then
            S <= B xor CI;
        elsif (B = '0') then
            S <= not CI;
        else
            S <= CI;
        end if;
    end process PR;
end architecture AR;
```

Short hands

```
architecture AR of SUM is
begin
    PR: process(A, B, CI)
    begin
        if(A = '0') then
            S <= B xor CI;
        elsif (B = '0') then
            S <= not CI;
        else
            S <= CI;
        end if;
    end process PR;
end architecture AR;
```

```
architecture AR of SUM is
begin
    S <= B xor CI when (A = '0') else
         not CI when (B = '0') else
         CI;
end architecture AR;
```
Short hands

architecture AR of SUM is
begin
PR: process (STATE)
begin
  case STATE is
  when INIT =>
    NEXT_STATE <= RUN;
  when RUN =>
    NEXT_STATE <= WAIT;
  when WAIT =>
    NEXT_STATE <= INIT;
  when others =>
    NEXT_STATE <= INIT;
  end case;
end process PR;
end architecture AR;

architecture AR of SUM is
begin
    with STATE select
    NEXT_STATE <= RUN when INIT, WAIT when RUN, INIT when others;
end architecture AR;

Concurrent assertions

- The concurrent assertion is equivalent to a regular process containing a single sequential assertion where SIGNALS_LIST is the list of signals appearing in CONDITION.

**Short hands**

```vhdl
architecture AR of SUM is
begin
PR: process (STATE)
begin
  case STATE is
  when INIT =>
    NEXT_STATE <= RUN;
  when RUN =>
    NEXT_STATE <= WAIT;
  when WAIT =>
    NEXT_STATE <= INIT;
  when others =>
    NEXT_STATE <= INIT;
  end case;
end process PR;
end architecture AR;
```

**Concurrent assertions**

- Concurrent procedure calls are supported. They can, for instance, be used to monitor signals. It is equivalent to a regular process where SIGNALS_LIST is the list of signals in PARAMETERS_LIST that are declared in or inout.

```vhdl
architecture AR of SUM is
begin
  with STATE select
  NEXT_STATE <= RUN when INIT, WAIT when RUN, INIT when others;
end architecture AR;
```

**Concurrent procedures**

- Concurrent procedure calls are supported. They can, for instance, be used to monitor signals. It is equivalent to a regular process where SIGNALS_LIST is the list of signals in PARAMETERS_LIST that are declared in or inout.

```vhdl
LABEL: PROCEDURE_NAME(PARAMETER_LISTS);
begin
  PROCEDURE_NAME(PARAMETER_LISTS);
  wait on SIGNALS_LIST;
end processus LABEL;
```

**Structural VHDL**

- VHDL supports hierarchical descriptions. Two mechanisms can be used:
  - Entity instantiations. Simple and fast but the less flexible. Does not allow top-down design flows. Every instantiated entity must be compiled prior compilation of embedding architecture.
    ```vhdl
    ADD16: entity WORK.ADD(DTW) generic map (N => 16) port map (A, B, S);
    ```
  - Component instantiations. More complex because every component instance must be bound to an actual entity (configuration). More flexible too because components are declarations. Allows compilation of top-level first (top-down design flows).
    ```vhdl
    component ADD is generic (N: Positive);
    port (X, Y: in Unsigned(15 downto 0));
    Z: out Unsigned(15 downto 0));
    end component ADD;
    ```
    ```vhdl
    ADD16: ADD generic map (N => 16) port map (A, B, S);
    ```
Structural VHDL

- When compiling the compiler checks the compatibility between component or entity interface and port mapping to actual signals.
- If configurations are used they can be flat or hierarchical and they allow specifying actual circuits versions.
- Configurations are a powerful tool but may lead to complex descriptions.

Generic parameters

- VHDL offers several mechanisms to build generic descriptions. Generic parameters are one of them.
- Inside the associated architecture a generic parameter is considered as a constant.

```
entity ADD is
  generic(N: POSITIVE range 1 to 32 := 8);
  port(A, B: in BIT_VECTOR(N - 1 downto 0);
       C: in BIT;
       S: out BIT_VECTOR(N - 1 downto 0);
       CO: out BIT);
end entity ADD;
```

Generic parameters

- Generic parameters may have a default value.
- The actual value of a generic parameter may be given by:
  - The component or entity instantiation statement (`generic` map)
  - Default value in component declaration
  - Default value in associated entity declaration.

```
architecture ARC of ADD is
  .
end architecture ARC;
```

Generic parameters, example of use

```
architecture ARC of MUL is
  signal X1, X2, S: BIT_VECTOR(16 downto 0);
begin
  .
  I_ADD: entity WORK.ADD(ARC) generic map(N => 17)
    port map(A => X1, B => X2, S => S);
  .
end architecture ARC;
```
The generate statement

- The generate statement are another mechanism to build generic descriptions.
- They are the concurrent equivalent of the sequential for loops and if statements.

```vhdl
entity ADD1 is
    port(A, B, CI: in BIT;
          S, CO: out BIT);
end entity ADD1;
```

Resolution functions

- When a signal is driven by multiple processes it has as many drivers as source processes. In order to compute its actual value a “resolution” function is needed. This function is associated to the type of the signal which is said to be a resolved signal.

```vhdl
function WIRED_OR(VAL: BIT_VECTOR) return BIT is
begin
    if (VAL'LENGTH = 0) then
        return '0';
    end if;
    for I in VAL'RANGE loop
        if (VAL(I) = '1') then
            return '1';
        end if;
    end loop;
    return '0';
end function WIRED_OR;
```

Resolution functions: example of use

```vhdl
subtype RESOLVED_BIT is WIRED_OR BIT;
signal S1: RESOLVED_BIT;
signal S2: WIRED_OR BIT;
```

```vhdl
entity CPU is
    port(A: out STD_ULOGIC_VECTOR(7 downto 0);
         IO: inout STD_ULOGIC_VECTOR(15 downto 0);
         WEB, OEB: out STD_ULOGIC);
end entity CPU;
architecture ARC of CPU is
    . .
    IO <= "1101010011101111"; -- write
    IO <= "Z" when others;
    . .
end architecture ARC;
```
Resolved type: beware

- Never use a resolved type if it’s not needed
- Compiler and linker would not help detecting unwanted shortcuts
- Systematically using resolved types (STD_LOGIC instead of STD_ULOGIC) is thus dangerous
- Systematically using resolved types (STD_LOGIC instead of STD_ULOGIC) also slows down the simulations (resolving a conflict takes time)

Agenda

- Introduction
- Principals of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

Library STD, package STANDARD

- The STD package is a standard one that comes with any VHDL design environment
- The STD library is implicitly declared in each source file. It is never necessary to re-declare it
- The STANDARD package defines the base types:
  - Enumerated: BOOLEAN, BIT, CHARACTER, SEVERITY_LEVEL
  - Numeric: INTEGER, NATURAL, POSITIVE, REAL, TIME
  - Composite: STRING, BIT_VECTOR
- And the NOW function
- The STANDARD package is implicitly declared in each source file. It is never necessary to re-declare it

Library STD, package TEXTIO

- This package is dedicated to ASCII files I/O
- Unfortunately it is very poor
- It defines:
  - Types LINE, TEXT, SIDE and WIDTH
  - Files INPUT and OUTPUT
  - Procedures READLINE, READ, WRITELINE and WRITE
  - Function ENDLINE
- It has to be explicitly declared:
  - use STD.TEXTIO.all;
Library IEEE, package
STD_LOGIC_1164

- Defines a multi-valued logic as an enumerated type:
  - type STD_ULOGIC is (‘U’, -- Uninitialized
    ‘X’, -- Forcing Unknown
    ‘0’, -- Forcing 0
    ‘1’, -- Forcing 1
    ‘Z’, -- High Impedance
    ‘W’, -- Weak Unknown
    ‘L’, -- Weak 0
    ‘H’, -- Weak 1
    ‘-’  -- Don’t care);

- Also defines a resolution function for STD_ULOGIC and the
  associated resolved type: STD_LOGIC

- And also defines the corresponding vector types
  STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR

Library IEEE, others packages

- The IEEE library contains some other useful packages

- Two are dedicated to arithmetics on vectors:
  - NUMERIC_BIT:
    - Defines types SIGNED and UNSIGNED (arrays of BIT)
    - Overloads the arithmetic, logic and relational operators for those
types
    - Defines the conversion function from and to Integer types
    - Adds various dedicated functions (rotations, shifts, etc.)
  - NUMERIC_STD
    - Defines types SIGNED and UNSIGNED (arrays of STD_LOGIC)
    - Overloads the arithmetic, logic and relational operators for those
types
    - Defines the conversion function from and to Integer types
    - Adds various dedicated functions (rotations, shifts, etc.)

Agenda

- Introduction
- Principals of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices
Combinational logic inference

- Combinational logic is inferred from signals that are:
  - Unconditionally assigned (that is, every time the process resumes)
  - Or
  - Conditionally assigned but for every possible condition (that is, every time the process resumes)

Latches inference

- Latches are inferred from signals that are:
  - Not always assigned during process execution
  - Assigned on a particular level of a control signal or combination of signals
  - With or without initialization
    - Synchronous
    - Asynchronous
Latches inference

- Latches are inferred from variables that are:
  - Read before being written
  - Assigned on a particular level of a control signal or combination of signals
  - With or without initialization
    - Synchronous or asynchronous

```vhdl
process(data_in, enable)
variable tmp: BIT;
begin
if enable = '1' then
  tmp := data_in;
else
  tmp := '0';
end if;
data_out <= tmp;
end process;
```

Flip-flops inference

- D flip-flops are inferred from signals that are:
  - Not always assigned during process execution
  - Assigned on a particular edge of a control signal (clock)
  - With or without initialization
    - Synchronous
    - Asynchronous

```vhdl
process(clk)
begin
if clk = '1' and clk'event then
  data_out <= '1';
elsif reset_sig = '1' then
  data_out <= '0';
else
  data_out <= data_in;
end if;
end process;
```

Flip-flops inference

- D flip-flops are inferred from variables that are:
  - Read before being written
  - Assigned on a particular edge of a control signal (clock)
  - With or without initialization
    - Synchronous
    - Asynchronous

```vhdl
process(clk)
variable tmp: BIT;
begin
if clk = '1' and clk'event then
  data_out <= tmp;
  tmp := data_in;
end if;
end process;
```
Clock edge specification

- Clock edge specification supported by most synthesizers:
  - if (clk'event and clk = '1') then
  - wait until (clk'event and clk = '1');
  - if (rising_edge(clk)) then
  - wait until rising_edge(clk);
  - if (clk'event and clk = '0') then
  - wait until (clk'event and clk = '0');
  - if (falling_edge(clk)) then
  - wait until falling_edge(clk);
  - ...

Unwanted latches inference

- Avoid unwanted latches inference

```vhdl
signal curr_state, next_state, modifier:
  std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
  next_state <= "100";
  case curr_state is
    when "000"  => next_state <= "100" or modifier;
    when "001"  => next_state <= "110" or modifier;
    when "010"  => next_state <= "001" and modifier;
    when "100"  => next_state <= "101" and modifier;
    when "101"  => next_state <= "010" or modifier;
    when "110"  => next_state <= "000" and modifier;
    when others => next_state <= "100";
  end case;
end process;
```

Unwanted latches inference

- Avoid unwanted latches inference

```vhdl
signal curr_state, next_state, modifier:
  std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
  case curr_state is
    when "000" => next_state <= "100" or modifier;
    when "001" => next_state <= "110" or modifier;
    when "010" => next_state <= "001" and modifier;
    when "100" => next_state <= "101" and modifier;
    when "101" => next_state <= "010" or modifier;
    when "110" => next_state <= "000" and modifier;
    when others => null;
  end case;
end process;
```
Supported loops

- Loops
  - Can be synthesized
  - But they are unrolled first
  - Bounds of `for` loops must be static:
    - This is synthesizable: `for I in 0 to 7 loop`
    - Not this: `for I in F(X) to G(Y) loop` (except when X and Y are compile-time constants)
  - Conditions of `while` loops must be static
    - This is synthesizable: `while FALSE loop`
    - Not this: `while C(X, Y) loop` (except when X and Y are compile-time constants)
  - The `while` and infinite loops are usually not synthesizable

wait statement

- The `wait` statements are sometimes supported but with limitations. Examples of such limitations:
  - One single `wait` statement per process
  - Always as the first (or last) instruction
  - Only for clock edge specification
  - With clock as single signal in on part...

Synthesis options

- Synthesis options
  - May be special comments
  - VHDL attributes
  - Multiple usages:
    - `synthesis on/off`
    - `translate on/off`
    - Set and reset
    - Arithmetic architectures
    - Encoding of enumerated types
    - Wired or multiplexed logic (`case`)
    - Coding and optimization of state machines
    - Semantics of resolution functions
    - ...

Synthesis packages

- Some packages are dedicated to logic synthesis
  - IEEE standard:
    - `IEEE.STD_LOGIC_1164`
    - `IEEE.NUMERIC_BIT`
    - `IEEE.NUMERIC_STD`
  - Proprietary packages:
    - Attributes declarations – Synthesis options
    - Proprietary arithmetic functions
    - Macro-functions VHDL models
    - VHDL models of standard cells libraries
    - ...

**NUMERIC_BIT and NUMERIC_STD**

- Standard arithmetic on BIT- (or STD_LOGIC-) based types
  - Types SIGNED and UNSIGNED
  - Classical arithmetic operators are overloaded for SIGNED and UNSIGNED
  - SIGNED and integers or UNSIGNED and integers can be mixed in expressions
  - Integer to and from vector conversion functions are defined:
    - TO_INTEGER
    - TO_SIGNED, TO_UNSIGNED
  - These vector types are compatible one with the other; the corresponding conversion functions all have the same name as the destination type:
    - SIGNED, UNSIGNED, BIT_VECTOR for NUMERIC_BIT
    - SIGNED, UNSIGNED, STD_ULOGIC_VECTOR, STD_LOGIC_VECTOR for NUMERIC_STD

---

**Dangers**

- Beware:
  - Unwanted registers
  - Flip-flops
  - Latches
  - Incomplete sensitivity lists
  - Loops
  - Combinational loops
  - Sign in arithmetic operations
  - Partitioning
  - Portability
    - The semantics for synthesis is not standard
    - Proprietary packages are ... proprietary

---

**Agenda**

- Introduction
- Principals of Event Driven Simulation
- Practical Organization of Files and Projects
- Compilation Units
- Syntax
  - Sequential VHDL
  - Concurrent VHDL
- Standardized Packages
- Logic Synthesis
- Advices

---

**Synchronize**

```vhdl
wait for 10 * PERIODE;
...
```

```vhdl
for i in 0 to 9 loop
  wait for RISING_EDGE(CK);
end loop;
...```
For Synthesis separate synchronous and combinational

PR: process (CK)
begin
  if RISING_EDGE (CK) then
  S <= S + 1;
  end if;
end process PR;

PRS: process (CK)
begin
  if RISING_EDGE (CK) then
  S <= E;
  end if;
end process PRS;

PRC: process (S)
begin
  E <= S + 1;
end process PRC;

For Synthesis count registers

architecture ARC of REGS is
begin
  REGS_PR: process (CK) -- 3 bits registers
  variable A0, A1: BIT;
  begin
    if RISING_EDGE (CK) then
      DOUT <= A1;
      A1 := A0;
      A0 := DIN;
    end if;
  end process REGS_PR;
end ARC;

To speed up simulation, avoid signals

architecture ARC of REGS is
begin
  REGS_FR: process (CK)
  begin
    if RISING_EDGE (CK) then
      A0 <= DIN;
      A1 <= A0;
      DOUT <= A1;
    end if;
  end process REGS_FR;
end ARC;

architecture ARC of REGS is
begin
  REGS_FR: process (CK)
  variable A0, A1: BIT;
  begin
    if RISING_EDGE (CK) then
      DOUT <= A1;
      A1 := A0;
      A0 := DIN;
    end if;
  end process REGS_FR;
end ARC;

Comment a lot and don’t mix models and reality

- One line of code = 10 lines of comments
- HDL /= matériel
Master complexity

- Very frequently the problem turns out to be much more complex than initially expected. In such situations the designer progressively piles up modifications.
- Effect: if the problem is serious the code rapidly becomes a unusable piece of code, impossible to understand or maintain.
- Solution: Restart from scratch, taking into account the discovered new problems. Re-design the partitioning, the data structures, rewrite everything.

Hardware and software

- VHDL is a programming language but its main goal is to model hardware. Writing VHDL without a clear idea of the underlying hardware cannot lead to good results.
- Effects: impossibility to refine the code into a synthesizable form, different behaviors before and after synthesis.
- Solution: think hardware first. To model a hardware architecture you must have a clear idea of it.