Hardware Modeling [VU] (191.011)
– WS25 –
VHDL Basics

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

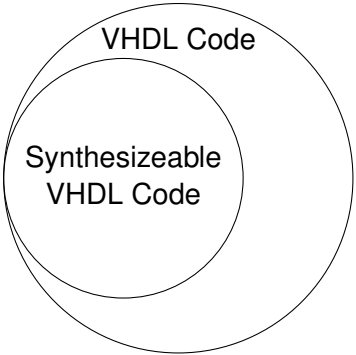
Modified: 2025-12-16, 16:09 (f8a58e9)

Introduction

- VHDL (Very High Speed Integrated Circuit Hardware Description Language)
  - Widely used in industry and academia
  - Alternatives: Verilog, SystemC, System Verilog, ...
- Lots of online resources available
  - Tutorials, books, tools, ...
- Developed in the 80's for U.S. Department of Defense
  - Based on Ada (strongly typed concept)
  - Revisions 1987, 93, 2000 and 2002

Introduction (cont'd)

- Initially solely used to document hardware
  - Later extended by synthesis tools
  - Only subset of commands can be transferred to hardware (= synthesizable VHDL)
  - All VHDL code is simulatable
- VHDL Standard 2008 taught in this lecture
  - Attention: not all 2008 features are supported by EDA tools
  - Has to be explicitly selected in tools (not the default)



Language Properties

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`
- Commands terminated by ';'
- Comments
  - '- -' single line comment
  - Since 2008 multi-line comments possible ('/*' and '*/')
- Format used in lecture
  - `keywords`
  - `datatype`
  - `comment`
  - `CONSTANT`
  - `everything else`

Basic Identifiers


```
basic_identifier ::= letter{[underline]letter_or_digit}
```

- First character must be a letter
- No underscore at the end
- No two consecutive underscores

valid	invalid
left	_left
left1	0left
left_0	left_
left2_0	left__0

Extended Identifiers

- Extended identifier syntax:

```
extended_identifier ::=
    \graphic_character{graphic_character}\
```

- Special identifier enclosed by backslashes
- `graphic_character` can contain:
  - Upper-/lower-case letters (including language specific letters like ä, å, â)
  - Digits
  - Special characters (”, #, &, ¾, etc.)
  - Space characters
- Examples:
  - `\best’VAR’ev@r\`
  - `\# of bits\`
  - `\this const represents m in ˜inch\`
  - `VHDL`, `\VHDL\`, `\vhdl\` - three different identifiers

Entity

- Start by characterizing a hardware `entity`
  - Defines a module's interface
  - Specifies name, inputs, and outputs
  - Encapsulates internal details
  - Blackbox definition: no knowledge of inner workings is needed

- Example `entity` with no I/O:

```
1 entity ENTITY_NAME is
2   -- I/O definitions
3 end entity;
```

Architecture

- Describes the internal structure via `architecture`
- Defines how the entity's functionality is implemented
  - Knowledge of inner workings is needed
  - How does the entity behave?
- It is possible to define multiple architectures for a single entity
  - Typical names for architectures: `beh`/`behavioral` or `struct`/`structural`

- Example: empty `architecture`

```
1 architecture ARCH_NAME of ENTITY_NAME is
2   -- constants, etc.
3 begin
4   -- description of inner workings of ENTITY_NAME
5 end architecture;
```

Process

- Defines sequential execution within an architecture
- `wait` at the end signals process termination
- Example `process`:

```
1  architecture ARCH_NAME of ENTITY_NAME is
2    -- constants, etc.
3  begin
4    process
5      -- constants, variables
6    begin
7      -- sequential statements
8      wait;
9    end process;
10 end architecture;
```

Variables & Constants

- Used within processes for temporary storage
- Declared after process definition and before `begin`
- Local to the process
- Variables:
  - Like variables in other programming languages
  - Optional default value on declaration
- Constants: Read-only and requires value on declaration
- Example declarations of one constant and two variables:

```
1 constant BYTE_WIDTH : integer := 8;
2 variable x,y : integer := 0;
```

Packages

- Define reusable code modules via packages
- Contain common declarations: constants, types, functions, and procedures
  - Promotes modularity and code reuse
  - Provides a central place to manage shared definitions

- Example `package` declaration:

```
1 package screenInfo is
2   -- constant declarations
3   constant SCREEN_WIDTH : integer := 720;
4   constant SCREEN_HEIGHT : integer := 480;
5 end package;
```

Using Packages

- Import packages into your VHDL code with the `use` clause
- Provides access to types, constants, functions, and procedures defined in the package
- Use the following syntax:
  - `use` *library_name.package_name.*`all`*;*
  - `use` *library_name.package_name.element_name;*
  - Default library ***work***: current project working library

- Example of using a `package`:

```
1 -- make SCREEN_WIDTH, SCREEN_HEIGHT available to this file
2 use work.screenInfo.all;
3
4 -- entity declaration
5
6 -- entity architecture
```

Basic Operators

- Assignment Operator: `:=`
- Logical Operators (*logop*): `and`, `or`, `nand`, `nor`, `xor`, `xnor`
- Relational Operators (*relop*): `=`, `/=`, `<`, `<=`, `>`, `>=`
  - Used for comparing values, returns a boolean result
- Arithmetic Operators (*addop*): `+`, `-`, `&` (concatenation)
- Multiplication Operators (*mulop*): `*`, `/`, `mod`, `rem`
- Miscellaneous Operators (*miscop*): `**` (exponentiation), `abs` (absolute value), `not` (logical negation)

- Important: VHDL is a strongly typed language
  - Types of both operands must match
  - Result type on the left side must match operands on the right side of assignments

Basic Expression Elements

- `prim ::= lit | const`
  `5, '1', true, clk_freq`

- `factor ::= (prim [** prim]) | (abs prim) | (not prim)`
  `abs -3, not true, 5 ** 2`

- `term ::= factor [mulop factor]`
  `5 * 2, 10 / 2, 7 mod 3`

Combining Elements into Expressions

- `sexpr ::= [+/-] term [addop term]`
  `3 + 2, -1 + 7, result - 5`

- `relation ::= sexpr [relop sexpr]`
  `5 > 3, a <= b, x = y`

- `expr ::= relation [logop relation]`
  `(a = b) or (c > d)`

Basic Sequential Statements

- *literal* := *sexpr*;
  - Assigns the value of an expression to a variable (literal)
  - Strongly typed: the types on the left and right sides must match
- `report` *string*;
  - Outputs the given string during simulation
  - In VHDL 2008: use `to_string(var)` to convert variables to strings
  - Use `&` operator to concatenate strings
  - Example:

```
report "Current screen width: " & to_string(SCREEN_WIDTH);
```

- `null`;
  - No operation; useful as a placeholder

Control Flow: Branching

- if/else:

```
if expr then
    {sequential statement}
[{elsif expr then
    {sequential statement}}]
[else
    {sequential statement}]
end if;
```

- select:

```
case expr is
{when choice [{|choice}] =>
    {sequential statement}}
end case;
```

- *choice* ::= *sexpr* | `others`
- All possible choices must be covered

Control Flow: Basic Loops

- Basic loop:

```
loop
    {sequential statements}
end loop;
```

- `next` [`when` *expr*];
  - Skips the rest of the current iteration (like `continue` in Java/C)
- `exit` [`when` *expr*];
  - Exits the loop entirely (like `break` in Java/C)

Control Flow: Typed Loops

- `while` loop:

  `while` *expr* `loop`
  {sequential statements}
  `end loop`;

  - Continues as long as the condition is true

- `for` loop:

  `for` *literal* `in` *range* `loop`
  {sequential statements}
  `end loop`;

  - *range* ::= prim `to` prim
    | prim `downto` prim
  - Iterates over a specific range

Miscellaneous: Block Identifiers & Labels

- Every VHDL code scope can have an identifier
  - Required in certain scopes (e.g., `architecture`, `entity`)
  - Identifiers help to clearly define and manage the scope's purpose
- Scopes are closed with the `end` keyword
  - Identifier can be included in the `end` line, e.g., '`end entity myEntity;`'
  - We advise against this practice
- Loops, `if` statements, and `process` blocks can have optional labels
  - C-like syntax: e.g., '`LABELID: process`'
  - Labels can be used with `next` and `exit` to target specific loops, improving control flow within nested structures

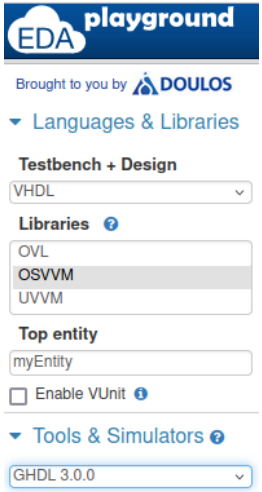
Example Process: Fibonacci


```
1  main: process
2    constant UPPER_BOUND : integer := 20;
3    variable current, previous, temp : integer := 0;
4  begin
5    previous := 0;
6    current := 1;
7    for i in 0 to UPPER_BOUND loop
8      temp := current;
9      current := previous + current;
10     previous := temp;
11     report "Fibonacci(" & to_string(i) & ") = "
12                  & to_string(current);
13   end loop;
14   wait;
15 end process;
```

Testing: EDA Playground Overview

- **Web-based platform** for simulating and sharing HDL code
  - `https://www.edaplayground.com/`
- Supports multiple languages:
  - VHDL, Verilog, SystemVerilog, and more
- Features:
  - Access to various simulators (e.g., GHDL, ModelSim (QuestaSim))
  - Collaborative coding with sharing links
  - No installation needed, runs in the browser
- Ideal for:
  - Learning and practicing HDL coding
  - Testing and debugging small code snippets
  - Demonstrating concepts in a classroom setting

EDA Playground Settings



- Set Testbench + Design language to VHDL
- Set Libraries to `OSVVM`
  - Open Source VHDL Verification Methodology
- Specify ”Top entity”
  - ”main” entity: `myEntity` in this example (see testbench.vhd on next slide)
- Select `GHDL 3.0.0` as Simulator
  - Open Source VHDL Simulator

EDA Playground Code

testbench.vhd

```
1  entity myEntity is
2  end entity;
3
4  architecture beh of myEntity is
5  begin
6
7    main: process
8      constant UPPER_BOUND : integer := 20;
9      variable current, previous, temp : integer := 0;
10   begin
11     previous := 0;
12     current := 1;
13     for i in 0 to UPPER_BOUND loop
14     temp := current;
15     current := previous + current;
16     previous := temp;
17     report "Fibonacci(" & to_string(i) & ") = "
18                 & to_string(current);
19     end loop;
20   wait;
21   end process;
22 end architecture;
```

design.vhd

```
1  entity design is
2
3  end entity;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

- Put your code (entity + architecture) into the testbench file on the left
- GHDL requires a non-empty design file on the right

EDA Playground Output

Log Share

```
[2024-08-14 14:15:08 UTC] ghdl -i  --std=08 design.vhd testbench.vhd
analyze testbench.vhd
elaborate myentity
testbench.vhd:16:11:@0ms:(report note): Fibonacci(0) = 1
testbench.vhd:16:11:@0ms:(report note): Fibonacci(1) = 2
testbench.vhd:16:11:@0ms:(report note): Fibonacci(2) = 3
testbench.vhd:16:11:@0ms:(report note): Fibonacci(3) = 5
testbench.vhd:16:11:@0ms:(report note): Fibonacci(4) = 8
testbench.vhd:16:11:@0ms:(report note): Fibonacci(5) = 13
testbench.vhd:16:11:@0ms:(report note): Fibonacci(6) = 21
testbench.vhd:16:11:@0ms:(report note): Fibonacci(7) = 34
testbench.vhd:16:11:@0ms:(report note): Fibonacci(8) = 55
testbench.vhd:16:11:@0ms:(report note): Fibonacci(9) = 89
testbench.vhd:16:11:@0ms:(report note): Fibonacci(10) = 144
testbench.vhd:16:11:@0ms:(report note): Fibonacci(11) = 233
testbench.vhd:16:11:@0ms:(report note): Fibonacci(12) = 377
testbench.vhd:16:11:@0ms:(report note): Fibonacci(13) = 610
testbench.vhd:16:11:@0ms:(report note): Fibonacci(14) = 987
testbench.vhd:16:11:@0ms:(report note): Fibonacci(15) = 1597
testbench.vhd:16:11:@0ms:(report note): Fibonacci(16) = 2584
testbench.vhd:16:11:@0ms:(report note): Fibonacci(17) = 4181
testbench.vhd:16:11:@0ms:(report note): Fibonacci(18) = 6765
testbench.vhd:16:11:@0ms:(report note): Fibonacci(19) = 10946
testbench.vhd:16:11:@0ms:(report note): Fibonacci(20) = 17711
Done
```

- Outputs Fibonacci sequence
- Caution if you try higher *UPPER_BOUND*: `integer` is 32-bit wide

Lecture Complete!