W2. Introduction to Verilog language

Modeling of combinational circuits using hardware description languages

1. Objectives

O1. Learning the basics of Verilog, a hardware description language

O2. Implementation of combinational logic circuits using Verilog HDL

O3. Familiarization with the Verilog operators and constants

2. Theoretical background

2.1 Fundamentals of modeling in Verilog

Verilog Hardware Description Language (HDL) provides a modular and portable, textual representation of system architecture. A digital system can be implemented at the level of several abstraction layers, starting from the transistor level, technology-dependent to the logic gate level, independent of technology, transfer register level (RTL), an algorithm (or behavioral level). This laboratory guide focuses on the description of RTL digital systems because it provides an effective and condensed illustration while remaining strongly correlated with the physical implementation of the system. The exemplified Verilog code can be channeled directly to the configurable logic blocks (CLBs) of a Field Programmable Gate Array (FPGA) device or even an Application Specific Integrated Circuit (ASIC).

2.1.1 Verilog modules

The basic unit of the Verilog language is the **module**, which contains the description of the interface and behavior of an electronic circuit. A module is the closest to the concept of "black box" which consists of knowing the inputs and outputs, without knowing the details of implementation and how it works.

Observation:
In other programming languages, the basic unit is the function. A module, once instantiated, can be seen as a "function" that inherits a series of properties.
We cannot "call" a module to execute an action. For example, we cannot call an adder module to realize the sum of numbers. But what we can do is to instantiate an adder, to link the two numbers as inputs and we will know that at the output of the adder we will have their sum.

The declaration of a module is done as follows:

```
module <module_name> (
    <port_1> <port_name_1>,
    <port_2> <port_name_2>,
    ...
    <port_n> <port_name_n>
    );
// Module implementation.
endmodule
```

This statement consists of:

- The keyword **module**
- **Name of module**
- **List of ports:** the elements declared here are used to connect external signals to the module and can be
    - **inputs**
    - **outputs**
    - **inouts**
- **; (dot and comma)**
- Module implementation
- Keyword **endmodule**

The interface of a module consists of:

- Module name: can contain letters, numbers, $ and _. The first character must be a letter or _.
- List of ports: zero or more ports, each with a direction (input, output or inout) and a name. We can only declare the names of the ports in this list, but in this case, the direction of the ports must be declared in the body of the module.

```
module moduleA ();

// Module implementation.

endmodule



module moduleB (a, b, c, d);

input a;

input b;

output c;

inout d;

// Module implementation.

endmodule



module modulC (input a, output b);

// Module implementation.

endmodule
```

The implementation of the mode consists in describing its operation using:

- Declarations
  - **wire**: represents physical connections between components. They are used for signal transmission (only in the description of combinational logic) and do not have the ability to retain information (they do not have a state)
  - **reg**: are used to store data, which persists even if the registry is disconnected. The register is the equivalent of an internal variable in a programming language. It holds a value and can be assigned a value
- Constructions
  - Module instantiations
  - **assign** can only be used on wire and implies that this wire is the output of a combination of logic gates
  - **initial:** they allow us to define an initial state. This block will run only once when the module is initialized.
  - **always:** contain actions that will be performed periodically.

The inputs, outputs, and bidirectional ports of a module are by default considered to have the wire type. We can specify if we want, that the outputs have the ***reg*** type.

All ***wire*** and ***reg*** variables (including input, output, and inout variables) have, by default, 1-bit bandwidth. We can specify, if we want, a width greater than that using the construction [i: j]. Attention, i and j cannot take the value 0 simultaneously and i> = j or j> = i.

```
input a;            // 1-bit input wire variable.

output b;           // 1-bit output variable.

output reg c;       // 1-bit output register variable.

inout d;             // 1-bit bidirectional wire variable.

wire e;              // 1-bit local wire variable.

reg f;                // 1-bit local register variable.

input [7:0] g;        // Variable input wire, on 8 bits ordered 7 → 0

wire [0:7] h;         // Local wire variable, on 8 bits ordered 0 → 7.

reg [2:6] i;          // Local register variable, on 5 bits ordered 2 → 6.
```

The Verilog language allows the use of one module in describing another module by instantiating it. The modules are instantiated as follows:

```
module_name <instance_name> (

    .port_name_1(<signal_name_1>),

    .port_name_2(<signal_name_2>),

    ...

    .port_name_n(<signal_name_n>),

    );
```

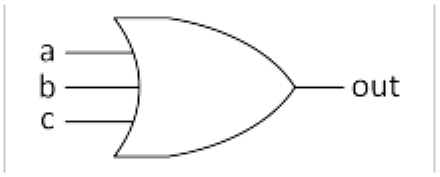
Below is an example of declaring and instantiating the module:

```
module moduleA(input a, input b, output c, output d);

    // Module implementation.

endmodule

module moduleB(input in1, input in2, output out);

    reg out1, out2, out3, out4, out5;

    moduleA a1(in1, in2, out1, out2);

// I wrote the arguments in the order of the ports.

    moduleA a2(.b(in2), .a(in1), .c(out3), .d(out4));

// I wrote the arguments in a random order, but for each, I specified the name of
the port to which it should be linked.

    moduleA a3(in1, in2, out5);       // I didn't link port "d". This is not a
mistake.

// Module implementation.

endmodule
```

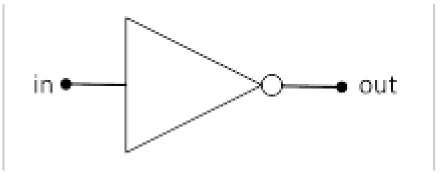
Observation:
It is not necessary to link all ports to a module when instantiating it. For example, an adder can have a data output (the sum of the two inputs) and a transport output (carry). If the transport output is not useful, it is usual not to connect it to any variable.

To describe circuits using Verilog, we also have several primitives that are associated with the basic logic gates. Each primitive has associated ports through which it connects to the outside. Thus, there are predefined primitives that offer the possibility to connect several inputs (and, or, nor, nand, xor, xnor), or several outputs (buf, not).

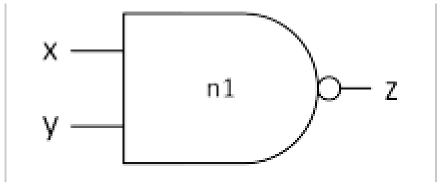
The use of a primitive is done by instantiating with the list of signals that will be connected to its ports.



```
or(out, a, b, c);
```



```
not(out, in);
```



```
nand(z, x, y);
```

Observation:
In the case of primitives, it is mandatory to declare at the beginning the output signals, these being followed by the inputs.
For predefined primitives the instance name is optional.
As the previous statement suggested, in Verilog you can also define UDPs (User Defined Primitives), through the truth table.

Verilog descriptions of a digital system consisting of modules that, at the RTL, illustrate the behavior of a system component. The module starts with the description of inputs and outputs. Globally, the inputs and outputs of a module are known as ports. At the level of transfer registers, a module can contain the following types of instructions:

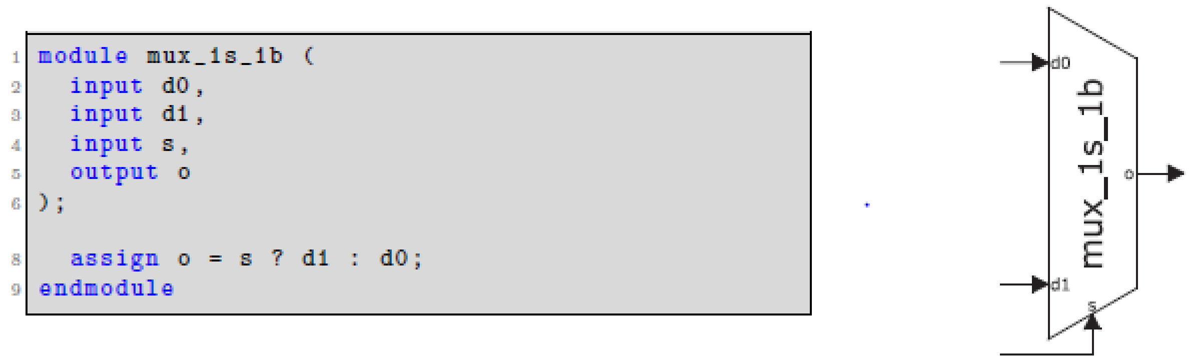
a) assign
b) always
c) instances of other modules

Assign instructions are called continuous commands precisely because they describe the behavior of the combinational design. On the other hand, always-type instructions can be used to describe both combinational and sequential circuits, depending on how the command is used. Improper use can lead to undesirable effects when it comes to defining a combinatorial description instead of a sequential one and vice versa.

The Verilog module in Fig. 1 implements a 2-to-1 multiplexer with input data d0 and d1, selection line s, and output o. A Verilog module starts with the module interface that includes the module name and its ports, enclosed in parentheses. The implementation of the module contains a line of code where the output will be defined by the value given by the ***conditional operator "?"***

```
condition_expression ? expression_true : expression_false
```

which evaluates either expression true or expression false depending on the value given by condition_expression (1 or 0).



```
module mux_1s_1b (
   input d0,
   input d1,
   input s,
   output o
);

   assign o = s ? d1 : d0;
endmodule
```

Fig. 1. Verilog description of a 2-by-1 multiplexer on 1 bit

An important aspect to understand when writing assign instructions is that “assign instructions are not imperative in nature” [Stro05], an aspect that differentiates Verilog from other imperative programming languages. The line of code in Fig.1 does not assign a value to the variable o! Instead, it connects the right side and the left side of the instruction via a physical link [Stro05]. Also, the module itself defines the physical connection of the model inputs to its outputs. All signal names become threads in a physical implementation. Comments in Verilog have the same format as in Java and C. For high visibility and accessible reading, appropriate indentation is encouraged.

2.1.2 Bus lines

Consider that the 2-to-1 multiplexer, defined above, needs to be modified in such a way that it can operate with 32-bit data. To eliminate the need to work with 65 input lines (32 lines for input d0, 32 lines for input d1, and a selection line) and to avoid writing 32 similar assign commands, Verilog incorporates the bus concept as a collection of threads. A bus can be assimilated with a vector and is defined by specifying the upper and lower rank of a bit in square brackets, near the bus. The 32-bit multiplexer, type 2-to-1 is described in Fig.2.

It is worth mentioning that the buses defined in lines of code 2,3 and 5 in Fig.2 have ranged from 31 to 0. The assign instruction in the code in Fig.1 remains unchanged.

```verilog
module mux_1s_32b (
  input [31:0] d0,
  input [31:0] d1,
  input s,
  output [31:0] o
);

  assign o = s ? d1 : d0;
endmodule
```

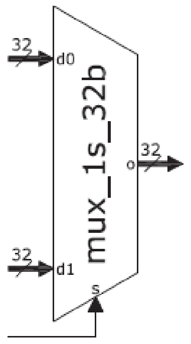


Fig. 2. Verilog description of a 2-by-1 multiplexer on 32 bits

The Verilog language allows you to select a set of lines from a bus. Consider, for example, the case of building a multiplexer that takes a 64-bit input, and depending on the selection line, will have either the more significant or less significant half. This can be achieved using the part-select facility as described in Fig. 3.

```verilog
1 module mux_half_64b (
2   input [63:0] d,
3   input s,
4   output [31:0] o
5 );

7   assign o = s ? d[63:32] : d[31:0];
8 endmodule
```

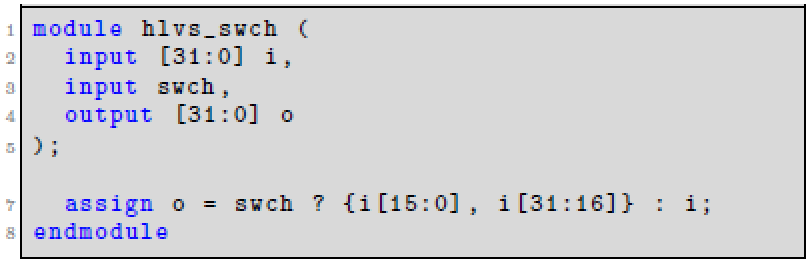
Fig. 3. Un multiplexor 2 la 1 cu facilitatea part-select pentru o intrare pe 64 biți

If the selection line s is active, the more significant half of input d is generated at the output, otherwise, the less significant half is carried to the output. The true expression of the conditional operator, in line 7, is a 32-bit vector that includes ranks 63 to 32 of input d. Similarly, the expression false represents the less significant half of input d.

Observation:
When you want to deliver a data package that represents half of the initially declared bit range, it is important to make sure that the widths of the true and false expressions of the conditional operator are equal!

Buses can also be generated in Verilog through dedicated builders. The first Verilog operator to build buses is the concatenation operator. Concatenation is constructed as a comma-separated list of signals, enclosed in curly brackets. The concatenated signals are attached, the first signal occupying the most significant positions, followed by the second concatenated signal, and so on. As an example, consider a module that, depending on the value of the control line, called an swch, switches the two halves of input to 32 bits. The implementation of the mode is illustrated in Figs. 4.

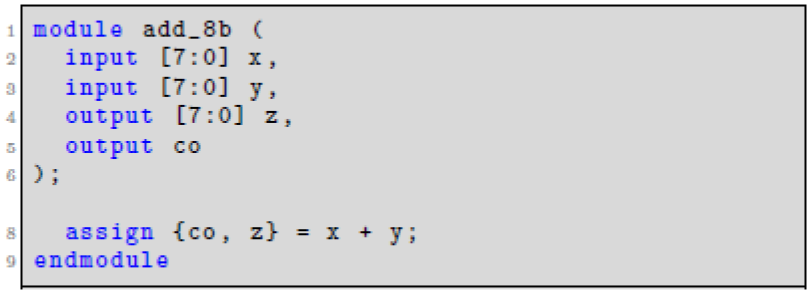
The switching of the halves is performed by placing the less significant half of i (position 15 to 0) in front of the most significant half within the concatenation operator. If swch has the value 0, the output will have the vector ordered by bits as stated at the input.



```
module hlvs_swch (
  input [31:0] i,
  input swch,
  output [31:0] o
);

  assign o = swch ? {i[15:0], i[31:16]} : i;
endmodule
```

Fig. 4. Verilog mode for switching the halves of a 32-bit input

The concatenation operator can also be used as a factor to the left of an expression. Such a construction is used to assign values to several signals in a single sentence. The widths of the left side and the right side need to be equal. The behavioral implementation of a binary adder is a typical example of this. Consider an 8-bit binary adder whose code is described in Fig.5.



```
1 module add_8b (
2   input [7:0] x,
3   input [7:0] y,
4   output [7:0] z,
5   output co
6 );
7
8   assign {co, z} = x + y;
9 endmodule
```

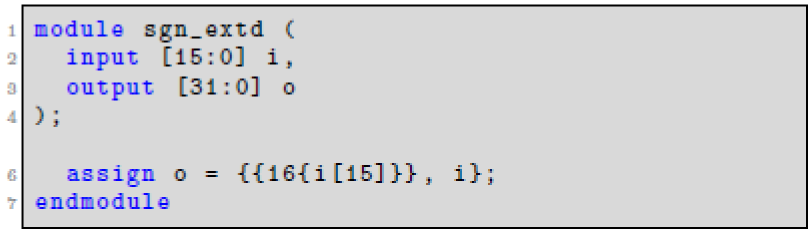
Fig. 5. 8-bit adder with output carry

The adder interface includes 8-bit x and y operands, z output, also 8-bit, for sum, and a 1-bit transport output. Adding two numbers on n-bits will result in a sum of n + 1 bits. Consequently, the left member of the expression in the line of code 8 (see Fig.5) has 9 bits, the transport bit being on the most significant position. A remark should be made in connection with modulo 2^n type operands. The result of this assembly is only n bits (without transport). In this case, the left hand-side of the expression adopts an n-bit bandwidth and the result is generated as follows: assign z = a + b. The synthesis simulator or tool will select either a typical n-bit adder or a modulo 2 adder to the power of n based on the width of the left-hand-side of the expression.

The second facility for building buses is the replication operator, which, as the name suggests, duplicates an expression several times, concatenating all copies of that expression into a vector. The replication operator can be built in the following format:

```
{<number of replications>{<value to be replicated>}}
```

For example, consider designing a Verilog module to extend a 16-bit signed number to a 32-bit one. Negative numbers have the value 1 as the most significant bits (MSB), while positive positions assign the value 0. The broad representation of the dominant signature in digital systems is the Two's Complement (C2) [Vlad12] and the extension of the signs is achieved by adding the signal bit in the most significant positions.



```
module sgn_extd (
  input [15:0] i,
  output [31:0] o
);

  assign o = {{16{i[15]}}, i};
endmodule
```

Fig. 6. Verilog mode for sign extension

The code fragment of Fig. 6 implements the extension of the sign by duplicating the inputs 16 times in the most significant positions and then switching to concatenating the input number.

2.1.3 Types of data

Verilog operates with two types of data::

- physical
- abstract

Physical data types have an exact match in the hardware used in RTL modeling. Only physical data types can form buses. The most common types of physical data are::

a) **wires -** describe the connections between the internal components of a module. The threads are used at any level of Verilog abstraction. A thread carries a value from one place to another without storing that value. Wire values are set either by allocation declarations or by module outputs.

b) **reg** - describes the deposits that retain the last value assigned to them. Consequently, reg signals must not be conducted continuously, unlike wires. Both internal signals or modules can be of the reg type. They are used at RTL or algorithm level as well as in test benches. Their value is set in an always or an initial block.

The types of abstract data, although they have a hardware correspondence to a certain extent, depend on the simulation/synthesis tool used. They are used in Verilog behavioral implementations, and test benches, facilitating the testing of other Verilog modules. Abstract types include:

a) ***integer***
b) ***time***

c) ***real***
d) ***parameter***

2.1.4 Operators

The conditional operator, presented above, is the only ternary Verilog operator. Bitwise operators have vectors as arguments. Their symbols, functions, and the number of operands that an operator can take over are shown in the table below:

Table 1 - Verilog bitwise operators

Symbol	Function	Arity
∼	bitwise complementation	unary
&	bitwise AND	binary
—	bitwise OR	binary
^	bitwise EXOR	binary
^∼	bitwise XNOR	binary

The bitwise NAND operator is obtained by denying the result of a bitwise AND such as (a | b). Similarly, the NOR operator is defined. Using bitwise operators, the 32-bit multiplexer 2-to-1, implemented in Fig.2 can be rewritten as in Fig.7.

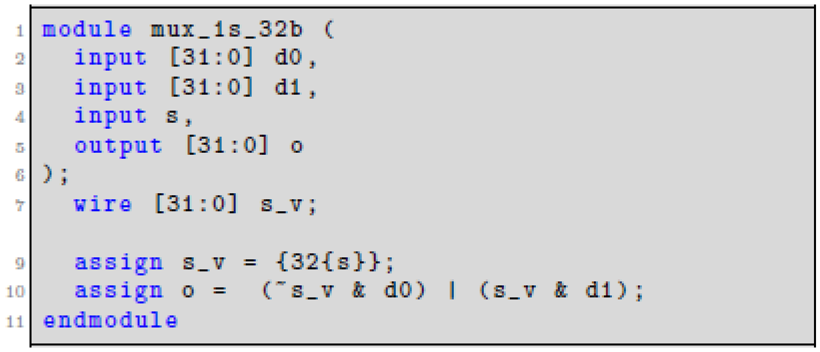

```
module mux_1s_32b (
  input [31:0] d0,
  input [31:0] d1,
  input s,
  output [31:0] o
);
  wire [31:0] s_v;

  assign s_v = {32{s}};
  assign o =  (~s_v & d0) | (s_v & d1);
endmodule
```

Fig. 7. 32-bit multiplexer implemented with bitwise operands

Reduction operators are unary type operators taking a vector input and generating an output obtained by applying that operator overall lines of the input vector. As an example, testing a vector a that is zero can be done by the following statement assign is_0 = ~ (| a);. The reduction operator applies an OR over all bits of the vector to obtain a result on a single bit that is denied that is_0 to be 1 when a is 0 ... 00.

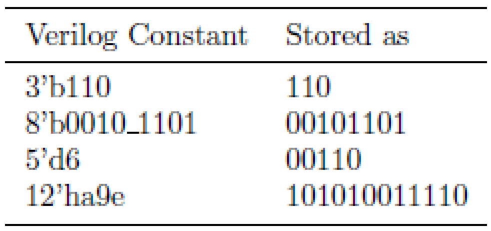
Verilog also provides arithmetic, relational, displacement, and logic numbers such as +, -, *, /,%, <,>, <=,> =, ==,! =, <<, >>,!, &&, ||. Logical and relational operators return either the Boolean logical value 1 if an expression is true, or 0 for a false expression.

2.1.5 Constants

Verilog constants have the following format `<bit width>'<radix specifier><value>` for which:

a) **bit_width** is a decimal number representing the number of bits allocated to the constant. Although optional, it is good practice to specify a constant bit length.
b) **radix_specifier** it can be b for binary, o for octal, d for decimal, and h for hexadecimal. This area is optional, and the decimal radix is the default.
c) **value** is the value of the constant mentioned in the specified radix.

Tabel 2 – Constante Verilog



Verilog Constant	Stored as
3'b110	110
8'b0010_1101	00101101
5'd6	00110
12'ha9e	101010011110

3. References

[Ashe07] P. J. Ashenden, *Digital Design (Verilog): An Embedded Systems Approach Using Verilog.* Morgan Kaufmann, 2007.
[Nava05] Z. Navabi, *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification*, 2nd ed. McGraw-Hill Professional, 2005.
[Paln03] S. Palnitkar, *Verilog Hdl: A Guide to Digital Design and Synthesis*, 2nd ed. Prentice Hall, 2003.
[Vlad12] M. Vlăduțiu, *Computer Arithmetic: Algorithms and Hardware Implementations*, 2012th ed. Springer, 2012.
[Stro05] L. Strozek. Verilog Tutorial - Edited for CS141. [Online]. Available: Verilog Tutorial