Experiment 4
Verilog & Adders and Subtractors

1. Objectives

- To familiarize students with Verilog HDL for combinational logic circuits using Verilog language and Cloud V online digital design platform
- To familiarize students with simulation using HDL
- Introduce students to modular hardware design
- Design and implement different types of adders
- Compare the time delay and area consumption of different types of adders
- Design a subtractor using an adder

1. Material

- Cloud V: https://cloudv.io

2. Introduction to Verilog

As digital designs are becoming complex, the conventional way of designing digital circuits starting from schematic became unsuitable. That led to the introduction of Hardware Description Languages (HDL) and Electronic Design Automation (EDA) tools. A digital circuit can be modeled (described) using an HDL then synthesized and mapped to any of the possible implementation technologies (e.g., FPGA, CPLD, ASIC, etc..) as shown in Figure 1. Also, HDLs can be used to verify digital circuits through simulation.

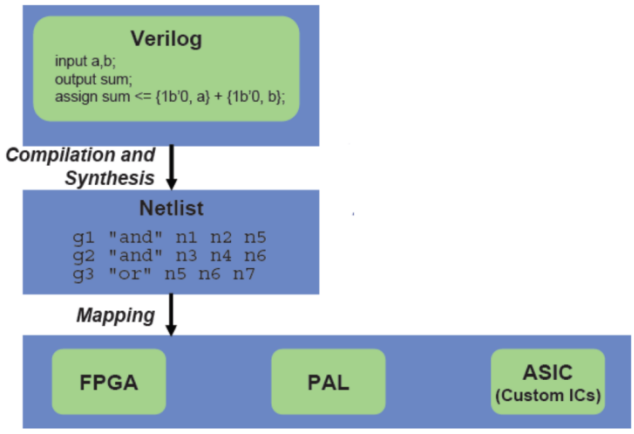

Figure 1. Verilog based Digital Systems Design Flow (Synthesis and Technology Mapping)

Verilog is the most used HDL to design digital systems. Verilog allows the designer to represent digital circuits in many ways. However, designers usually use Verilog to describe digital circuits in two fundamentally different ways: Structural and Behavioral (see Figure 2). One possibility is to use Verilog constructs that represent simple circuit elements such as logic gates. A larger circuit is defined by writing a code that connects such elements together. This is referred to as the structural representation of logic circuits. The second possibility is to describe a circuit using logic expressions

and programming constructs that define the behavior of the circuit but not its actual structure in terms of gates. This is called the behavioral representation.

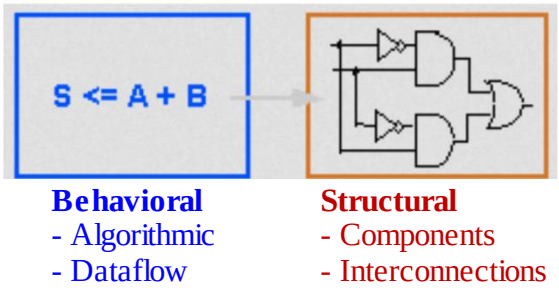

Figure 2. Structural vs. Behavioral Description

a. Verilog Modules

A module is the building block in Verilog. It is declared by `module` and is terminated by `endmodule`. Inputs and outputs connections of the circuit are declared with the keywords `input` and `output` respectively. The following is a Verilog code example that describes 2 modules. A Verilog module is a block of hardware.

Example code 1: Verilog Module

```
module circuit_one(A,B,C,D,E);
      input A,B,C;
      output D,E;
      // logic (circuit) description
endmodule

module circuit_2(input A,B,C, output D,E);
      // logic (circuit) description
endmodule
```

Models are the basic building blocks (similar to functions in C programming) of hardware description to represent your circuit. Please note the following:

- The first line of each module is named the module declaration. It has the `module` keyword, the module name, and the list of the module input and output ports. It should be terminated with semicolon.
- All statements in Verilog must be terminated with a semicolon. The line having `endmodule` is not terminated by a semicolon because `endmodule` is not a statement, it is a keyword.

i. Modules instantiation

In C programing language, we cannot define a function inside another one. However, we can call a function inside the body of another function. Similarly, one module definition cannot contain another module definition within its `module` and `endmodule` statements. Instead, a module definition can incorporate copies of other modules by instantiating them. For example as shown in Figure 3, to design a “full adder” circuit, 2 identical circuits (“half-adders") are needed.

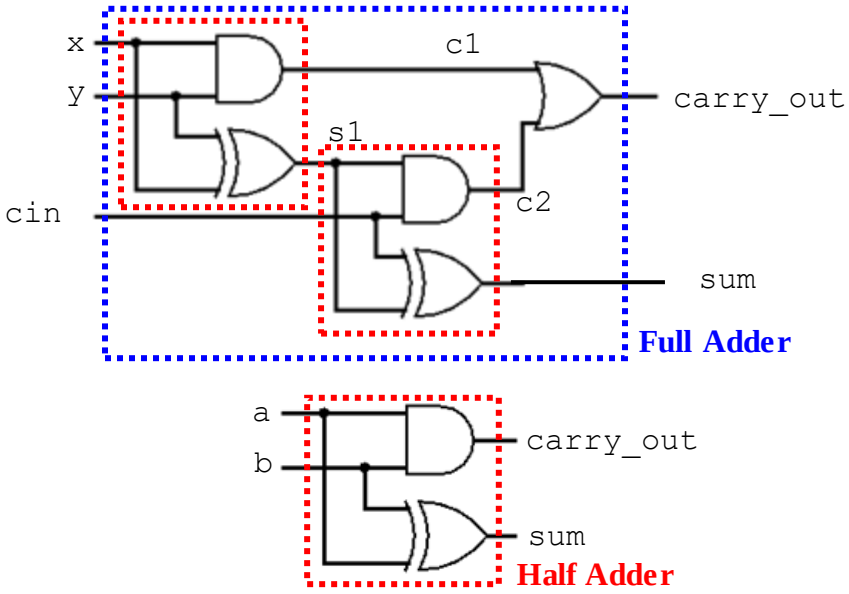

Figure 3. Full adder circuit which include 2 half adder circuits

The following code example 2 shows the implementation of a half adder circuit module. And it also has an implementation of a full adder circuit which instantiates 2 half adders.

Example code 2: A full adder module which instantiates 2 half adder modules

```
module HalfAdder(input a,b, output sum, carry_out);
      assign sum = a^b; // sum = a xor b
      assign carry_out = a&b; // carry = a and b
endmodule

module FullAdder (input x,y,cin, output sum, carry_out);
      wire s1,c1,c2;
      HalfAdder HA1 (.a(x),.b(y),.sum(s1),.carry_out (c1));
      HalfAdder HA2 (.a(cin),.b(s1),.sum(sum),.carry_out (c2));
      assign carry_out = c1|c2;
endmodule
```

To declare an instance of a module inside another module, you write the name of your module (like `HalfAdder` in the example), give it a unique name (like `HA1`, `HA2` in the example) and connect the ports. In the example code, the line:

```
HalfAdder HA1 (.a(x),.b(y),.sum(s1),.carry(c1));
```

tells the synthesizer (like a compiler in C programming language) to create an instance of the half adder module called `HA1`, connect the inputs `x` and `y` of the full adder to the ports `a` and `b` of the instance and to connect `sum` and the `carry_out` of the half adder to the internal signals (wires) `s1` and `c1`. The other line:

```
HalfAdder HA2 (.a(cin),.b(s1),.sum(sum),.carry(c2));
```

is generating another instance called `HA2`, where signal `s1` is connected to port `b` of the half adder, the input `cin` is connected to port `a`, the `carry_out` port is connected to an internal signal `c2`, and the `sum` port is connected to the output `sum`. The keyword `wire` is used for internal signal, which are neither inputs nor outputs. "`|`" indicates a bit-wise "or"ing.

b. Vectors

Until now, we have only considered signals with only one bit. However, there are times when we need to use a bus which is a group of bits. For this, Verilog allows us to use vectors. To declare a bus signal, this is the syntax:

<signal type> [MSB:LSB] <signal_name>;

Example code 3 shows some examples for different buses' declarations and discusses the differences between them, and example code 4 shows a module that computes a four-input XOR function.

Example code 3: Examples for different buses' declarations and discusses the differences between them

```
input [3:0] a; // Defines an input port bus named a of 4 bits

output [5:0] sum; // Defines an output port bus named sum of 6 bits

wire [7:1] carry; // Defines an internal bus carry of 7 bits

wire [3:0] x = 4'b0001; // Defines an internal bus x of 4 bits where
x[0]=1, x[1]=0 , x[2]=0,  x[3]=0. The MSB is x[3] and the LSB is x[0].
b refers to binary representation of constant.

wire [0:3] y = 4'b0001; // Defines an internal bus y of 4 bits where
y[0]=0, y[1]=0, y[2]=0, y[3]=1. The MSB is y[0] and the LSB is y[3]

wire [3:0] z = 4'd7; // Defines an internal bus z of 4 bits where z
holds the binary representation of the decimal value 7 (because of d).
```

Example code 4: HDL module that computes a four-input XOR function, the input is a3:0, and the output is y

```
module myXOR(input [3:0]A, output Y);
      assign Y = ^A; // Y = A[0] ^ A[1] ^ A[2] ^ A[3]
endmodule
```

c. Structural Specification of Logic Circuits

Verilog includes a set of gate level primitives that correspond to commonly used logic gates. In Verilog, a logic gate is represented by indicating its functional name, output and inputs. For example:

```
and g1(y,x1,x2); //y is the output, x1 and x2 are the inputs
```

Verilog supports basic logic gates as primitives: `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`, `buf`. The connections between hardware elements are declared with the keyword `wire`. Figure 3 shows an example of simple circuit and code example 5 shows the Verilog structural specification of such circuit.

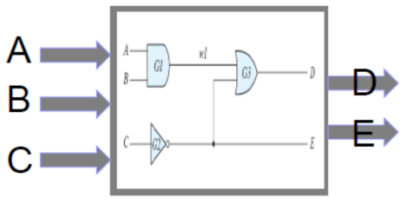

Figure 4. Simple design circuit example

Example code 5: Simple circuit (Figure 3) structural specification/description

```
module simple_circuit(A,B,C,D,E);
      input A, B, C;
      output D, E;
      wire w1;

      and g1(w1, A, B);
      not g2(E, C);
      or g3(D, w1, E);
endmodule
```

Inputs and outputs are wires by default.
So, `input A, B` is equivalent to `input wire A, B`.

Please note that structural modeling means, also, that your circuit (module) can be described by showing its structure (which is made of instances of other modules). Verilog supports more level-gate primitives as shown in code example 6, other primitives include `nand`, `nor`, `xnor`, and more.

Example code 6: Examples for supported level-gate primitives in Verilog

```
and g1(c,a,b)    // Sytax: and <name>(output, input 1, ..., input N)

or  g2(d,a,b,c) // Sytax: or <name>(output, input 1, ..., input N)

xor x(c,a,b)     // Sytax: xor <name>(output, input 1, ..., input N)

not N(b,c,a)     // Sytax: not <name>(output 1, ..., output N, input)
```

d. Behavioral Specification of Logic Circuits

Using gate level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a digital circuit. Code example 7 shows the behavioral Verilog model that describes the simple circuit in Figure 3.

Example code 7: Simple circuit (Figure 3) behavioral specification/description

```
module simple_circuit(A,B,C,D,E);
      input A, B, C;
      output D, E;
      wire w1;

      assign D = ~C|(A&B);
      assign E = ~C;
endmodule
```

Behavioral Verilog modeling (description) of combinational logic can be done using continuous assignment of asynchronous procedural blocks.

i. Continuous Assignment (using assign)

It defines the circuit using logic expressions. Figure 3 shows how the circuit can be defined using the Boolean expressions:

$$D = AB + C'$$

$$E = C'$$

Whenever any signal on the right-hand side changes its state, the value of **D** or **E** will be re-evaluated. The keyword for continuous assignment is `assign`. The following is a subset of the operators that can be using with continuous assignment:

- Boolean logic: `~` (not), `&` (and), `|` (or), `^` (xor)
- Arithmetic: `+`, `-`, `*`, `/`, `%` (modulus), `**` (power)
- Conditional assignment:
- Verilog does not allow the use of `if`/`else` statements in the normal syntax. You need to have what is called an `always` block which will be discussed later in Lab 6. However, you can use still can describe conditional combinational logic using the conditional assignment. Which have the following syntax:

  (<conditional_expression>)? (<value-if-true>) : (<value-if-false>);

  The following code example 8 shows some example.

Example code 8: Conditional Assignment Examples

```
assign d = c ? a : b; // if c is not zero, then d = a else d = b
assign d = (c==10) ? a : ( (c==5)?b:f ); // if c = 10 is true,
then d = a, else if c = 5 then d = b else d = f
```

ii. Procedural Assignment

To be discussed later in Lab 6.

e. Test Bench

A test-bench is a Verilog module that represents a circuit involving the circuit to be tested; the circuit to be tested is usually called Design Under Test (DUT). This code will apply different inputs to the DUT, monitor the outputs and display them to check their correction. The following code example 8 shows a Verilog testbench for the `simple_circuit` module given in Figure 3.

Example code 8: A Testbench for the Simple circuit (Figure 3)

```
module stimulus_simpleCircuit;
   reg A, B, C; // a reg for each input to the simple circuit
   wire x, y; // a wire for each ouptut to the simple circuit

   simple_circuit simp_ct(A, B, C, x, y); // DUT module instantiation

   initial begin // Initial block executes only once at start simulation
      A = 1'b0; B = 1'b0; C = 1'b0;
      #100 // delay of 100 nanoseconds
      A = 1'b1; B = 1'b1; C = 1'b1;
   end
endmodule
```

The `initial` block is used similarly to an always block except that the code in the block will only run once, at the start of the simulation. Test benches can be self-checking as in the following code example. Also, note how for loops are used to generate test cases.

Example code 9: A Self-checking Testbench

```
`timescale 1ns/1ns

module adder_tb ();
  parameter N = 2;

  reg [N-1:0] a = 0;
  reg [N-1:0] b = 0;
  wire [N:0] out;

  ripple_carry_adder #(.N(N)) uut(.A(a), .B(b), .O(out)); // Unit under test

  // "tasks" allow for reusable and easier to read code, like "functions"
but allowing time delay if needed inside
  task check; // Variables declared inside a task are local to that task
    input i, j, out;
    reg [N:0] out;
    integer i, j, golden_out;
    begin
      golden_out = i + j;
      if (out !== golden_out) begin // For the == and != operators, the
result is x, if either operand contains an x or a z
        $display($time, " Error: i=%d, j=%d, expected %d (%16b), got %d
(%16b)", i, j, golden_out, golden_out, out, out);
        // $stop; // suspends the simulation if needed
      end
    end
 endtask

  initial
    begin : TB // name should be given to declare variables
      integer i, j;
      for (i=0; i<=2**N-1; i=i+1)
        for (j=0; j<=2**N-1; j=j+1) begin // Don't forget begin
          a = i;
          b = j;
          #10
          check(i, j, out);
        end
      $stop; // To stop early if needed
    end
endmodule
```

f. Procedural Assignment

Previously in lab 4 we discussed two types of logic circuits specification using Verilog: Structural and Behavioral. Also, for the behavioral specification of logic circuits we have discussed one type of assignment which is the Contentious Assignment, and mentioned that there is another type which is the Procedural Assignment. Procedural assignment allows an alternative higher-level behavioral description of combinational logic. It supports C-like control structures such as `if`, `for`, `while` and `case`. Procedural statements have to be contained in an `always` block as in example code 1. The symbols in parentheses after the `@` symbol is the sensitivity list, the statements inside the `always`

block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. The statements inside the `always` block are evaluated in the order given in the code. Coming up with the incomplete sensitivity list is one of the common bugs in Verilog code. This can be solved by replacing the sensitivity list by "`*`" (`always @ *)`. Please note that anything assigned in an always block must also be declared as type `reg` (shorthand for a register). It's a register because once any item in the sensitivity list changes its value, the assignment is reevaluated and the register value is changed, and it keeps (stores) its changed value until any item in the sensitivity list changes its value in the future.

Example code 1: Procedural Assignment Behavioral Specification

```
module circuit(x1, x2, x3, f);
      input x1, x2, x3;
      output f;
      reg f;

      always @(x1 or x2 or x3) begin
            if (x2 == 1) begin
                  f = x1;
            end
            else begin
                  f = x3;
            end
      end
endmodule
```

- The two lines `output f; reg f;` are equivalent to a single line `output reg f;`
- The sensitivity list `(x1 or x2 or x3)` is not equivalent to `(x1 | x2 | x3)`. The first one has 3 items, while the second one has a single item representing an expression. For example, the first list is considered changing whenever x1 goes from 0 to 1, while the second list is not necessarily changing at this case because the whole expression value may be constant (equal 1) before and after x1 changes its value (for example x2 was equal to 1 when x1 was equal to 0).
- The sensitivity list items can be separated by comma's (`,`) or `or`'s.

g. Case Statements

The case statement performs different actions depending on the value of its input. A case statement implies combinational logic if all possible input combinations are defined. Example code 2 shows an example of 2x1 multiplexer implemented using a case statement.

Example code 2: 2x1 multiplexer implemented using a case statement

```
module circuit_example(input[1:0] In, sel, output reg
out);
      always @(In,sel)begin
            case(sel)
                  1'b0: out = In[0];
                  1'b1: out = In[1];
            endcase
      end
endmodule
```

h. Parameterized Modules

Verilog provides constant values that are fixed within the scope of a module instance and can be changed from an instance to another. They provide more flexibility to the module and more readability of the code. In Verilog, the parameter can be declared using the parameter keyword.

Code Example 1: A Parametrized Module

```
module parameterized_module #(parameter width = 3) (a,b);
       // recall this declaration from lab 4
       input [width-1:0]a; //  = input [2:0] a
       output b;
       assign b = &a;
       // recall the reduction operator from lab 4
       // b = a[2]&a[1]&a[0]
endmodule

module top_module (<some_inputs_and_outputs>);
       // some code
       parameterized_module #(5)  e1(a,b); // width = 5,
a[4:0]
       parameterized_module #(10) e2(a,b); // width = 10,
a[9:0]
endmodule
```

We can use more than one parameter inside a module as follows. Then, when creating an instance of the module, the parameters have to be called in order.

Code Example 2: More than one parameter

```
module parameterized_module #(parameter width = 3, n = 4'b0000) (a,b);
      //some code
endmodule

module top (<some_inputs_and_outputs>);
      parameterized_module #(4,10) m1 (a,b); // width = 4, n = 4'b1010
      parameterized_module #(4)    m2 (a,b); // width = 4, n = 4'b0000 (default)
      parameterized_module          m3 (a,b); // width = 3 (default), n = 4'b0000
(default)
endmodule
```

i. Generate block

Suppose you need to implement a 6-input AND gate as shown in Figure 2, a possible implementation can be as shown in Example.1.

Figure 1. module AND

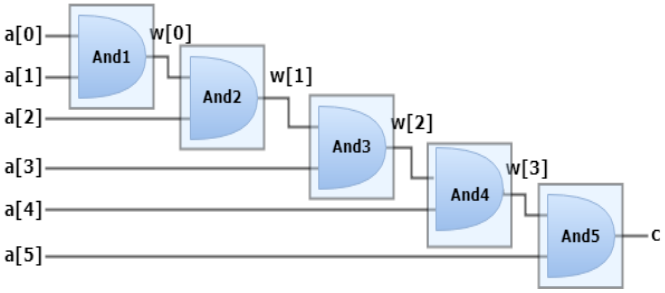

Figure 2. module AND6

Example code 1: 6-input AND gate

```
module AND(input a, b, output c);
      assign c = a&b;
endmodule

module AND6 (input[5:0] a, output c);
      wire [3:0] w;
      AND And1(a[0],a[1],w[0]);
      AND And2(a[2],w[0],w[1]);
      AND And3(a[3],w[1],w[2]);
      AND And4(a[4],w[2],w[3]);
      AND And5(a[5],w[3],c);
endmodule
```

The previous code is correct from a syntax point of view, however, it is not very clear. It can also be more tedious with an increasing number of instances (imagine a 32-input AND gate). This is why Verilog provides `generate` statements. The following example shows you how to generate 6-input AND gate using a `generate` statement. The following Example code 2 shows how to achieve that.

Example code 2: 6-input AND gate using generate statement

```
module AND_usingGenerate(input[5:0] a, output c);
      wire[3:0]w;
      genvar i;
      AND and1(a[0],a[1],w[0]);

      generate
            for (i=1;i<6;i=i+1) begin: myblockname
            AND myAND(a[i+1],w[i-1],w[i]);
            end
      endgenerate
endmodule
```

From the second to the fifth AND gates the inputs are: (1) one of the 6 inputs and the output of the previous AND gate, except for the first AND gate where the two inputs are two of the 6 inputs. That's why the first AND gate is defined separately before the generate statement.

- Generate regions can only occur directly within a module, and they cannot nest.
- Whilst `if/else`, `for` and `case` statements cannot be normally used in Verilog, they can be used inside the generate block.
- In general, `generate` blocks are used for either:
  - Allowing the instantiation of Verilog modules multiple times (using `for` loop).
    - The index of the for loop must be defined as a `genvar` variable (`genvar i` in the previous example)
    - The for loop must have `begin/end` keywords.
  - Selecting one Verilog module from multiple modules (using `if/else` or `case` statements)
- A generate block is declared by `generate` and `endgenerate` keywords. These two keywords are optional like `begin/end`. However, it is a very good practice to include them for readability.
- A generate block should have a name (e.g. myblockname in the previous example).

3. Pre-lab Questions

1. Draw the black box diagram of a full adder (FA) showing the inputs and the outputs

2. Design a 1-bit full adder circuit
   a. Write the truth table

b. Get the simplified function of the outputs using k-maps.

c. Draw the circuit

3. Draw the block diagram of a 4-bit ripple carry adder using full adders. Use the black box of the full adder in question 1.

4. Draw the block diagram of a 4-bit adder/subtractor circuit. The circuit should have an input (add/sub) that determines the operation (0: add, 1: subtract)

5. Draw the block diagram of an 8-bit ripple carry adder using 4-bit ripple carry adders.

6. Draw the block diagram of an 8-bit carry select adder using 4-bit ripple carry adders.

4. Experiments

All the following development should be on a private project on Cloud V platform.

a. Full Adder

Develop a Verilog module that implements the full adder in the second pre-lab question.

b. 4-bit ripple carry adder (4-RCA)

1. Develop a Verilog module that implements the 4-bit ripple carry adder in the pre-lab question 3 using the full adder module from question a.
2. Develop a self-checking testbench module to test your module from (a) and simulate it.

c. 4-bit adder/subtractor

Develop a Verilog module that implements the adder/ subtractor from pre-lab question 4 using the ripple carry adder module developed in the previous question.

d. 8-bit ripple carry adder (8-RCA)

1. Develop a Verilog module that implements 8-bit ripple carry adder using the full adder module from the first question and generate blocks.
2. Develop a self-checking testbench to test your module and simulate it.
3. Record the critical path delay and the chip area of the 8-bit RCA.

e. 8-bit carry select adder (8-CSA)

1. Develop a Verilog module that implements 8-bit carry select adder using the 4-bit RCA from question b(1).
2. Develop a self-checking testbench to test your module and simulate it.
3. Record the critical path delay and the chip area of 8-bit CSA.

5. Lab Report

1. [2 Pts] Report the values that you got in experiment questions d(3) and e(3) in the following table.
   [2 Pt] Comment on the results.

	8-bit RCA	8-bit CSA
Delay (ns)		
Chip Area		

2. [6 Pts]Repeat experiment question c using a generate block with if statements.