

# CPE 300L

## DIGITAL SYSTEM ARCHITECTURE AND DESIGN LABORATORY

### LABORATORY 2

#### QUARTUS, MODEL SIM AND VERILOG REVIEW

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING  
UNIVERSITY OF NEVADA, LAS VEGAS

### OBJECTIVE

Get familiar with Quartus 7.1, ModelSim and review the Verilog HDL.

### INTRODUCTION

The Verilog Hardware Definition Language (Verilog HDL) was originally developed as a tool for the simulation and testing of digital systems. At the time digital design was moving to systems having 100,000+ gates and the existing methods for such design were becoming impractical. Verilog was quickly adopted by many designers and soon used also for synthesis.

Verilog was based on the syntax of the C programming language and shares many of its operators and constructions. If you are familiar with C, learning Verilog should be pretty easy. The most significant difference is that C is a procedural language used to implement sequential algorithms while Verilog is used to model hardware which operates concurrently.

### THEORY OF OPERATION

#### 1. Continuous assignments

Combinational logic uses the current value of the inputs to determine the value of the outputs. Unlike sequential logic, combinational logic lacks memory. Combinational logic can be modeled in Verilog using gate primitives, Boolean expressions, or procedural descriptions. These correspond to structural, dataflow, and behavioral descriptions.

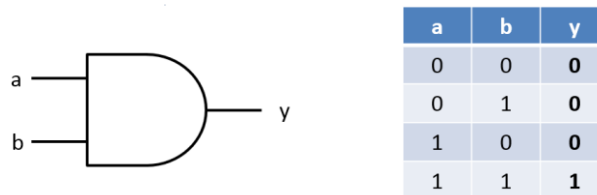


Fig. 1. AND gate

In Verilog a Boolean equation describing the relationship between the inputs and output can be written as a *continuous assignment statement*

Continuous assignment statement for AND gate:

```
assign c = a&b;
```

Here *a*, *b*, and *c* are Verilog nets. In Verilog the net data type is used to model a logic signal much like the voltage on a physical wire. A net has the value with which it is driven, if a net is undriven it floats (like a floating wire in the circuit). In Verilog a scalar net is the default data type.

The **&** operator is used here and it represents the bitwise AND operation. See the rest of operators in tables in *Operators* section. Bit value can always be complemented with **~** symbol. Keyword **assign** indicates that this is a continuous assignment statement. Unlike a procedural language such as C, this expression is evaluated continuously in simulation time. If there are any other continuous assignment statements in the model they will also be continuously evaluated at the same time, or concurrently. This concurrency is a basic difference between Verilog and procedural programming languages.

However, just the statement above is not sufficient to run the Verilog program. It must be put inside of a Verilog *module*. You can think of a Verilog module as the package that contains the logic circuit, in this case for the AND gate. Therefore, the complete code for implementing the AND gate in Verilog is:

```
module and2 (c, b, a);
    output c;           // module outputs
    input a,b;         // module inputs
    assign c = a&b;    // continuous assignment
endmodule
```

The module begins with the module keyword followed by the name of the module, *and2*, and then the port list. The nets in the port list connect the module to the rest of the world much like the “wires” on the *AND* gate in a schematic or the pins of an integrated circuit. Inside the module we need to declare them as input or output using the corresponding Verilog keywords. The continuous assignment statement is then used to implement the *AND* function in this module and we use the *endmodule* keyword to indicate the end of the module. Comments are placed, as in C, between */\** and *\*/* or after *//*. Like C, Verilog is case sensitive so that *And2* and *AND2* are not the same as *and2*.

Another style of code representation is called ANSI-C style, and looks as follows for the AND gate:

```
module and2 (output c, input a,b); // ANSI-C style port list
    assign c = a&b;
endmodule
```

## 2. Primitives

The AND operation is used frequently enough that Verilog includes an AND gate primitive as part of its built in component library. Instead of having to define the module `and2` as shown above we could use for the 2-input AND gate

```
and (c, a, b) ;
```

The AND gate primitive, like all Verilog primitives, lists the scalar output on the left of the port list and the scalar inputs on the right. It also allows for an arbitrary number of inputs which we could add to the right side of the port list. The Verilog primitives, `or`, `xor`, `nand`, and `nor` work similarly. The primitive `not` is used for an inverter, although in this case there is a single input on the right and an arbitrary number of outputs on the left, `buf` works similarly but does not invert the input. This output and input ordering is easy to remember since assignment statements also have inputs on the right and output on the left. (Note that port lists for modules in these notes use the same convention, outputs on the left of the port list and inputs on the right.)

## 3. Conditional operators in Verilog

The example of conditional operator can be written as:

```
assign c=s?a:b;
```

If the conditional variable `s = 1` then the statement will assign to `c` the value of the first variable while if the conditional value `s = 0` then the statement will assign to `c` the value of the second variable `b`.

## 4. Delays

The following construction:

```
assign #5 y = a & b;
```

or

```
and #5 G1(y,a,b) ;
```

models AND gate with a propagation delay of 5 ns. If you simulate and use input signal change more often than 5ns, then these changes are lost.

Regarding the ns time unit in the above example, ``timescale` compiler directive to indicates that in simulation the time unit and rounding. For example:

```
`timescale 1 ns / 100 ps
```

indicates that the time unit is 1 ns and any delays will be rounded to the nearest 100 ps.

## 5. Verilog Value types

The following value types exist in Verilog:

0, 1	Logical Zero, Logical One
Z	<ul style="list-style-type: none"> <li>Output of an undriven tri-state driver – high-impedance value</li> <li>Models case where nothing is setting a wire's value</li> </ul>
X	Models when the simulator can't decide the value – uninitialized or unknown logic value <ul style="list-style-type: none"> <li>Initial state of registers</li> <li>When a wire is being driven to 0 and 1 simultaneously</li> <li>Output of a gate with z inputs</li> </ul>

### Nets

- Nets represent connections between things
- Do not hold their value
- Take their value from a driver such as a gate or other module
- Cannot be assigned in an initial or always block

### Regs

- Regs represent data storage
- Behave exactly like memory in a computer
- Hold their value until explicitly assigned in an initial or always block
- Never connected to something
- Shared variables with all their attendant problems

## 6. Number representation

Format of the number in Verilog:

**<size>'<base\_format><number>**  
**4'b0110**

- **<size>** - decimal specification of number of bits
  - default is unsized and machine-dependent but at least 32 bits
- **<base\_format>** - ' followed by arithmetic base of number
  - <d> <D> - decimal - default base if no <base\_format> given
  - <h> <H> - hexadecimal
  - <o> <O> - octal
  - <b> <B> - binary
- **<number>** - value given in base of <base\_format>
  - \_ can be used for reading clarity
  - If first character of sized, binary number 0, 1, x or z, will extend 0, 1, x or z

Examples of number representations:

```

6'b010_111      gives 010111
8'b0110         gives 00000110
4'bx01          gives xx01
16'H3AB         gives 0000001110101011
24              gives 0...0011000
5'O36           gives 11100
16'Hx           gives xxxxxxxxxxxxxxxxx
8'hz            gives zzzzzzzzz

```

## 7. Vectors

Frequently we would like to use multibit values, for example, a byte in memory of 8-bits or an instruction of 32-bits. In Verilog 8-bit signals, a and b, might be defined as

```
wire [7:0] a, b;
```

Here [7:0] indicates a vector that consists of 8-bits, indexed from 7 to 0. Indexing them 7 to 0 is arbitrary, we could just as well have specified them as [0:7] or [3:10].

## 8. Verilog Operators

Operator Symbol	Functionality	Examples ain = 5 ; bin = 10 ; cin = 2'b01 ; din = 2'b0z
+	Add, Positive	bin + cin ⇒ 11    +bin ⇒ 10    ain + din ⇒ x
-	Subtract, Negate	bin - cin ⇒ 9    -bin ⇒ -10    ain - din ⇒ x
*	Multiply	ain * bin ⇒ 50
/	Divide	bin / ain ⇒ 2
%	Modulus	bin % ain ⇒ 0
**	Exponent*	ain ** 2 ⇒ 25

- Treats vectors as a whole value
- Results unknown if any operand is Z or X
- Carry bit(s) handled automatically if result wider than operands
- Carry bit lost if operands and results are same size

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b000 ; cin = 3'b01x		
!	Expression not true	!ain $\Rightarrow$ 1'b0	!bin $\Rightarrow$ 1'b1	!cin $\Rightarrow$ 1'bx
&&	AND of two expressions	ain && bin $\Rightarrow$ 1'b0	bin && cin $\Rightarrow$ 1'bx	
	OR of two expressions	ain    bin $\Rightarrow$ 1'b1	bin    cin $\Rightarrow$ 1'bx	

- Used to evaluate single expression or compare multiple expressions
  - Each operand is considered a single expression
  - Expressions with a zero value are viewed as false (0)
  - Expressions with a non-zero value are viewed as true (1)
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
~	Invert each bit	~ain $\Rightarrow$ 3b'010	~cin $\Rightarrow$ 3'b10x
&	AND each bit	ain & bin $\Rightarrow$ 3'b100	bin & cin $\Rightarrow$ 3'b010
	OR each bit	ain   bin $\Rightarrow$ 3'b111	bin   cin $\Rightarrow$ 3'b11x
^	XOR each bit	ain ^ bin $\Rightarrow$ 3'b011	bin ^ cin $\Rightarrow$ 3'b10x
^~ or ~^	XNOR each bit	ain ^~ bin $\Rightarrow$ 3'b100	bin ^~ cin $\Rightarrow$ 3'b01x

- Operates on each bit or bit pairing of the operand(s)
- Result is the size of the largest operand
- X or Z are both considered unknown in operands, but result maybe a known value
- Operands are left-extended if sizes are different

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b01x	
<<	Logical shift left	ain << 2 $\Rightarrow$ 3'b100	bin << 2 $\Rightarrow$ 3'bx00
>>	Logical shift right	ain >> 2 $\Rightarrow$ 3'b001	bin >> 2 $\Rightarrow$ 3'b000
<<<	Arithmetic shift left	ain <<< 2 $\Rightarrow$ 3'b100	bin <<< 2 $\Rightarrow$ 3'bx00
>>>	Arithmetic shift right	ain >>> 2 $\Rightarrow$ 3'b111 (signed)	bin >>> 2 $\Rightarrow$ 3'b000 (signed)

- Shifts a vector left or right some defined number of bits
- Left shifts (logical or arithmetic): Vacated positions always filled with zero
- Right shifts
  - Logical: Vacated positions always filled with zero
  - Arithmetic (unsigned): Vacated positions filled with zero
  - Arithmetic (signed): Vacated position filled with sign bit value (MSB value)
- Shifted bits are lost
- Shifts by values of X or Z (right operand) return unknown



Operator Symbol	Functionality	Examples ain = 4'b1010 ; bin = 4'b10xz ; cin = 4'b111z		
&	AND all bits	&ain ⇒ 1'b0	&bin ⇒ 1'b0	&cin ⇒ 1'bx
~&	NAND all bits	~&ain ⇒ 1'b1	~&bin ⇒ 1'b1	~&cin ⇒ 1'bx
	OR all bits	ain ⇒ 1'b1	bin ⇒ 1'b1	cin ⇒ 1'b1
~	NOR all bits	~ ain ⇒ 1'b0	~ bin ⇒ 1'b0	~ cin ⇒ 1'b0
^	XOR all bits	^ain ⇒ 1'b0	^bin ⇒ 1'bx	^cin ⇒ 1'bx
^~ or ~^	XNOR all bits	^~ain ⇒ 1'b1	~^bin ⇒ 1'bx	~^cin ⇒ 1'bx

- Reduces a vector to a single bit value
- X or Z are both considered unknown in operands, but result maybe a known value

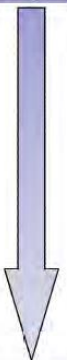
Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
>	Greater than	ain > bin ⇒ 1'b0	bin > cin ⇒ 1'bx
<	Less than	ain < bin ⇒ 1'b1	bin < cin ⇒ 1'bx
>=	Greater than or equal to	ain >= bin ⇒ 1'b0	bin >= cin ⇒ 1'bx
<=	Less than or equal to	ain <= bin ⇒ 1'b1	bin <= cin ⇒ 1'bx

- Used to compare values
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
==	Equality	ain == bin ⇒ 1'b0	cin == cin ⇒ 1'bx
!=	Inequality	ain != bin ⇒ 1'b1	cin != cin ⇒ 1'bx
===	Case equality	ain === bin ⇒ 1'b0	cin === cin ⇒ 1'b1
!==	Case inequality	ain !== bin ⇒ 1'b1	cin !== cin ⇒ 1'b0

- Used to compare values
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- For equality/inequality, X or Z are both considered unknown in operands and result is always unknown
- For case equality/case inequality, X or Z are both considered distinct values and operands must match completely

Operator Symbol	Functionality	Format & Examples
<b>?:</b>	Conditional test	(condition) ? true_value : false_value sig_out = (sel == 2'b01) ? a : b
<b>{}</b>	Concatenate	ain = 3'b010 ; bin = 3'b110 {ain,bin} ⇒ 6'b010110
<b>{{}}</b>	Replicate	{3 {3'b101}} ⇒ 9'b101101101

Operator(s)	Priority
+ - ! ~ & ~& etc. (unary* operators)	<div>High</div>  <div>Low</div>
**	
* / %	
+ - (binary operators)	
<< >> <<< >>>	
< > <= >=	
== != === !==	
& (binary operator)	
^ ~^ ^~ (binary operators)	
(binary operator)	
&&	
?:	
{ } {{ }}	

- ( ) used to override default and provide clarity



## LAB DELIVERIES

### PRELAB

#### 1. Install Quartus on your machine

In order to install Quartus on your personal machine, follow the steps:

1. Download the Quartus 7.1 from [http://eelabs.faculty.unlv.edu/soft/71\\_quartus\\_free.exe](http://eelabs.faculty.unlv.edu/soft/71_quartus_free.exe)
2. Go to <https://www.intel.com/content/www/us/en/forms/basic-intel-registration.html>
3. Register new basic Intel account, if you don't have one.
4. Log in and go to page:  
<https://www.intel.com/content/www/us/en/programmable/support/support-resources/download/licensing/lic-q2we-success.html>
5. Choose: *Request a license for Quartus II Web Edition (legacy versions prior to 8.1)*
6. Follow the instruction to download the license file.
7. After Quartus is installed, specify the location of license file.

#### 2. Drawing a circuit from code

Having the following code:

```
module f1(x, y, a, b, c);
    input a, b, c; output x, y;
    wire t1, t2;
    xor x1(t1,a,b);
    not n1(x,t1);
    and a1(t2,x,c);
    or o1(y,t2,b);
endmodule
```

Draw the schematic corresponding to the above code. Mark all the signals and identifiers.

#### 3. Write a Verilog code for a XNOR gate

Implement the module for 2-input XNOR gate

- a) Normal Verilog style
- b) ANSI-C style

#### 4. Write a Verilog code for a four input AND gate

### PRELAB DELIVERIES:

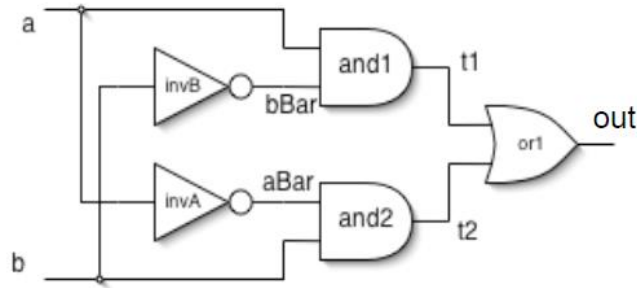
1. Not required
2. Schematic of all the signals marked
3. Verilog code for a) and b)
4. Verilog code

## LAB EXPERIMENTS

1. Compile and obtain waveforms for both a) and b) styles from prelab 3 and implement any one of the styles in a DE2 board.

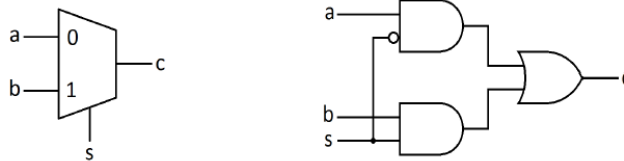
### 2. Using primitives

Write a Verilog code using primitives, for the following schematic:



Simulate and obtain the waveform. Verify the operation of the circuit, by comparing truth table with the DE2 implementation.

### 3. Multiplexer



Model the 2-to-1 MUX. Write the code

- Using continuous assignments
- Using the conditional statement

Simulate and obtain the waveform in ModelSim. Verify the operation of the circuit, by comparing truth table and the waveforms.

### 4. Delays

Model the OR gate having the 5ms delay.

- simulate the gate for gate input state changing every 10ms. Obtain waveform.
- simulate the gate for gate input state changing every 2ms. Obtain waveform.

Analyze the differences between these two waveforms.

## 5. Vectors

Write the Verilog code implementing the 8-bit adder. Simulate and get the waveform in ModelSim.

## POSTLAB REPORT:

Include the following elements in your postlab report:

Section	Element	
1	Theory of operation <i>Include a brief description of every element and phenomenon that appears during the experiments.</i> a. Describe what the primitive is in Verilog b. Define the structural model in Verilog c. Define the dataflow in Verilog	
2	Prelab report	
3	Results of the experiments	
	Experiment	Experiment Results
	1	a. Waveform for a and b. b. Picture of implementation in DE2
	2	a. Verilog code and waveform b. Picture of implementation in DE2
	3	a. The waveform (screenshot from ModelSim) b. Code for a and b
	4	a. Verilog code for a), waveforms b. Verilog code for b), waveforms
	5	a. Verilog code for the 8-bit adder b. The waveform (screenshot from ModelSim)
	Answer the questions	
4	Question no.	Question
	1	What is the Verilog preprocessor directive?
	2	What is the difference between & and && operators?
	3	Is Verilog case sensitive? If yes, what does it mean?
	4	List types of nets in Verilog.
	5	Label the structural and behavioral model in experiment 3
5	Conclusions <i>Write down your conclusions, things learned, problems encountered during the lab and how they were solved, etc.</i>	