Preamble

This document was created to help ECE students at UBC with their first computing course CPEN 211. It is intended to be used as a reference and **not a tutorial.** It assumes a basic familiarity with Boolean Algebra, CMOS Logic Gates, Combinational Logic Building Blocks and Logic Optimization.

Each page explains a new piece of syntax and begins with a description of the syntax, followed by a general formula and examples that are expanded on by notes. The document is further structured so that syntax appears in the order it would likely appear in actual code.[1] Where possible, related pieces of information are grouped together, but this can't always be done while still adhering to the above principle. When such situations arise, hyperlinks have been used to bring you to the required information.

How should you use this document? It's up to you, but here are a few recommendations:

1. Armed with an understanding of basic hardware principles, you'd like to learn how to design your own hardware so you read through this document one or two times before writing your own Verilog code.

2. You've designed some hardware using Verilog but it's not working and you're getting cryptic error messages or unusual results, so you read through this document and make sure you're not violating the guidelines given in the "notes" sections.

3. You have an idea for how to achieve a certain functionality in your code but are not sure if it is syntactically valid or synthesizable so you check this document to see if your idea is "allowed".

One final note: syntax in <> is optional.

With all that out of the way...enjoy!

[1] For example, you must declare a module before making any assignments, so module declaration appears first.

Synthesizing Verilog

This is arguably the most important section of the document. Read it thoroughly and understand it well.

Listen: The Verilog code you write is not the final product that will be implemented. The code will be optimized and translated into logic gates, via an algorithm. This process is called *synthesis*. You should make a few observations about synthesis:

1. Not all Verilog code can be translated directly into logic gates. This is often formulated as: "Not all Verilog code is synthesizable".

2. It is possible to specify the exact same hardware in multiple ways.

3. The algorithms are not perfect. It is sometimes possible to "beat" the tools and come up with very clever ways to implement certain functionalities.

It's hopefully clear that you **absolutely must** have some understanding of what goes on "behind the scenes". The tools should be used to speed up a process that you could do yourself (even if it would take an extremely long time), not to complete a process that you have no understanding of. At the level of hardware, we can no longer take advantage of the abstraction from details which languages like C, Java and Python afford us. In these higher level languages, much can be accomplished with little understanding of what a compiler is or what it does. Such gaps in knowledge are not acceptable when designing hardware.

The point is not that you should be an expert in the theory before ever trying to code in Verilog. Not at all. The point is simply that complete ignorance of theory **will** lead to issues and at least a basic familiarity with logic optimization is necessary to avoid committing fatal mistakes. Not only that, but a good grasp of the theory will help you write better designs that are easier to debug and will overall save you much time and strife.

If you take away nothing else from this section, at least remember this: Verilog is a hardware **description** language. It's only capable of **describing** hardware. In spoken language, we often describe devices which could never be built(time machine, faster-than-light travel, etc.). Verilog is no different. It's easy to describe hardware that could never be physically constructed. To avoid this, there are certain guidelines which should be followed. The full extent of these guidelines is beyond the scope of this document, but many of them can be found in the "notes" section for each piece of syntax.

Define Statement

The `define statement provides a way to symbolically reference commonly used values:

```
/* Defining a symbolic constant */
`define SYMBOLIC_NAME actualValue

/* Using a symbolic constant */
`SYMBOLIC_NAME

//Example:
`define COMMON_VALUE 8'b11111111

if(exampleSignal1 == `COMMON_VALUE) begin
  exampleSignal2 = `COMMON_VALUE;
end
```

Notes:

- A symbolic constant must be defined **before** it is first used.
- Don't use a semicolon when defining a symbolic constant!
- `actualValue` is a literal.

Header Files

Header files are very useful when referencing the same values or modules across multiple Verilog documents. Things quickly get messy when symbolic constants and modules are being added, removed or edited. Using a header file will make sure that any change is applied to all parts of your code at once. There is no fancy syntax for declaring a header file. There isn't even a different file extension! Just create a regular Verilog file and write in some code. Then use the `include` statement:

```
`include "filename.v"
```

Notes:

- `filename` is the name of your header file.
- Place the header file in the **same directory** as the rest of your code.
- `include` the header file in each document that will use it.

Literals

In Verilog, constant values are called **literals** and there is special syntax for defining them:

```
Width'RadixValue

//Example 1:
1'b1; //1 in binary
2'b1; //01 in binary
3'b0; //000 in binary

//Example 2:
16'hABCD; //ABCD in hexadecimal
12'd9; //9 in decimal

//Example 3:
1'bx //x in binary
2'bx //0x in binary
3'bxxx //xxx in binary

//Example 4:
1'bz //z in binary
2'bz //0z in binary
3'bzzz //zzz in binary
```

Notes:

- `Width` specifies, **in binary**, the bit-width of the literal. Make sure your literal has enough bits to store the specified value!
- `Radix` specifies the "radix"[2] or "base" Verilog should use when interpreting the literal. Use "b" for binary, "d" for decimal and "h" for hexadecimal.
- `Value` specifies the actual value the literal will take. 1-9 and A-F are used for specifying numbers. `x` represents "don't care", which means it doesn't matter if the bit is on or off. `z` represents "high impedance", which means the literal does not pass any value through.[3]
- Any bits that are not "used" will be set to 0.

[2] A radix is just a way to represent numbers. Some common radices are binary, decimal and hexadecimal.

[3] For those familiar with circuits, high impedance means the same thing here. A wire with high impedance will not easily allow any current through it.

Module Declaration

A basic building block in Verilog is the `module`:

```
module moduleName(moduleSignals);
  /*Insert code here*/
endmodule

//Example 1:
module example1(in, out);
  input in;
  output reg out;
endmodule

//Example 2:
module example2(input [1:0] in, output reg [3:0] out);
endmodule
```

Notes:

- The module is analagous to `main` in other coding languages. An extremely important note: **a module in verilog is not like a function in C or Python** or whatever other high-level language is your favourite.
- **All** input signals and **all** output signals are specified in the module declaration. Separate signals with commas.
- There is a semicolon after the `module` declaration but not after `endmodule`.

Signal Declaration

Remember: all I/O signals should be included in the module declaration. Then, inside the module, specify which signals are `input` and which are `output`. You may also want to specify additional signals that are neither inputs nor outputs.[4] Use the following syntax with the appropriate `keywords`:

```
<I/O> <type> <[n:0]> signalName;

//Example:
input [1:0] signal1;
output reg [15:0] signal2, signal3;
reg [0:2] signal4;
wire signal5 = 1'b1;
```

Notes:

- You must define signals that appear in the module declaration as either `input` or `output`.
- `type` specifies if the signal is a `wire` or `reg`. Signals that are not assigned a type are by default a `wire`.
- `[n:0]` specifies that a signal is an (n+1)-bit wide signal.[5] This means that if you want a 16-bit signal, you should declare the bus-width as `[15:0]`. Signals that are not assigned a bus-width are by default 1-bit.
- Don't forget the semicolon!

[4] These are called **internal** signals.

[5] This will put the MSB on the left. [0:n] will put the MSB on the right.

Module Instantiation

You can use modules inside other modules. To do this, you'll **instantiate** the module:

```
moduleName instanceName (.moduleSignal(connectedSignal));

//Example 1:
example EX1(.in(num1), .out(result));

//Example 2:
example EX2
  (
  .in(in),
  .out(out)
  );
  .
```

Notes:

- **Instantiating a module is not like calling a function.** Each instantiated module is a distinct entity, a separate copy. Once instantiated, the module stays there forever. Think about writing Verilog code like building a circuit on a breadboard. After a circuit is powered, there's no adding or removing components. Every component required to achieve a certain functionality must already be implemented before connecting power.
- Do all module instantiations before messing around with signals connected to them.
- `moduleName` is the name used when declaring the module[6]. `instanceName` is what this particular copy will be referred to as.
- `connectedSignal` is the signal which will be connected to `moduleSignal`. You will need to do this for each signal in the instantiated module. Separate each connection with a comma. This process is similar to plugging wires into the different pins of a chip on a breadboard.
- Any `connectedSignal` which maps to an `output` must be declared as `wire`, even if that signal is defined as a `reg` in the instantiated module!

[6] See module declaration

Operators

There are various operators in Verilog:

- `+ - * / % == != < <= >= >` - as seen in C
  - **IMPORTANT** `<=` can also be a **non-blocking assignment**[7] if it is used for assignment rather than comparing two values.
- `a**n` raises signal `a` to the power `n`
- `^` XOR
  - When `^` is used on an operator, then it makes that operator bitwise e.g. `^ +` is bitwise addition.
- `&` bitwise AND
- `|` bitwise OR
- `~` bitwise NOT[8]
- `a<<n` shifts signal `a` to the left by `n` bits.
- `a>>n` shifts signal `a` to the right by `n` bits.[9]
- {`a, b`} concatenates signal `a` to signal `b`.
- {`a{b}`} replicates signal `b`, `a` times.
- `a ? X:Y` If `a` is 1, then `X`. Otherwise, `Y`.

[7] A non-blocking assignment doesn't stop other lines from being executed. Rather than evaluating each statement one after the other, the non-blocking assignment will calculate every assignment before updating any values. Needless to say, this only really makes a difference inside always blocks.

[8] Do not use `!` for NOT

[9] When shifting, incoming bits are 0.

Assign

The `assign` statement provides a way to continuously drive a signal:

```
assign a = statement;

//Example:
assign in = 3'b101;
assign sel = b ^ c;
assign b = 1'b1;
assign c = &(in);
assign d = sel ? 1'b1 : 1'bz;
```

Notes:

- `a` must be a `wire`!
- All assignments are performed **concurrently and continuously**. That means no signals change until every assign statement has been executed. Let's bold that and make it red: **Assignments do not occur sequentially**. The breadboard analogy is useful yet again! The signal on a physical wire will change instantly the moment any inputs to that wire change.[10] The `assign` statement is no different.
- Whenever **any** signal changes value, all assignments are re-evaluated.
- Wire declarations[11] and assignments can be combined.
- `statement` can be a constant value or it can be one or more boolean expressions.
- `assign` statements cannot be used inside `always` blocks.

[10] Provided the wire is ideal and has no inductance or capacitance.
[11] See signal declaration

Case

The case statement provides a more versatile way to drive a signal:

```
case(selector)
  caseList : <begin> statement <end>;
  <default : defaultStatement;>
endcase

//Example:
case({select1, select2})
  2'b00 : out = 1'b1;
  2'b01, 2'b10 : begin out = 1'b0 end;
  2'b11 : begin
            if(randomSignal == 1'b0) begin
              out = 1'b0;
            end else begin
              out = 1'b1;
            end
      end
  default : out = 1'bx;
endcase
```

Notes:

- case statements can only be used inside always blocks.
- caseList is a series of statements that contain constant values. Multiple values can go in one caseList statement if you separate each value by a comma.
- caseList can be as long as you'd like. The selector signal will match with the caseList that contains its current value and execute the associated statement. If the selector signal does not match with any values contained in the caseList then it will execute the default statement. If you have no default statement then nothing will happen(which is almost always a problem!).[12]
- If you want to perform more than one operation in a statement, surround your operations by begin and end, as in statement2 above.
- The casex statement has the same syntax as the case statement but "don't cares" can be included in caseList. Likewise, the casez statement allows "high impedance" to be included in caseList.

[12]Even though the default statement is optional, always include it when building combinational logic.

If and Else

`if` statements are used to check if a condition is true. Else statements specify what to do if the condition was not true:

```
if(condition) <begin>
  /*Functionality to be executed if the condition was true*/
<end> else <begin>
  /*Functionality to be executed if the condition was not true*/
<end>

//Example 1:
if(exampleSignal1 == 2'b00) out = 1'b1;
else if(exampleSignal1 == 2'b01) begin
  out = 1'b1;
end else out = 1'b0;

//Example 2:
if(exampleSignal2 == 1'b1) begin
  controlSignal1 = 1'b1;
  controlSignal2 = 1'b0;
end
else begin
  controlSignal1 = 1'b0;
  controlSignal2 = 1'b0;
end
```

Notes:

- `if` and `else` statements can only be used inside `always` blocks!
- While an `else` statement is technically optional, you should always include it. If there is no `else` associated to an `if` then when the condition isn't true, nothing will happen which is almost always a problem.
- If you're just starting out, I recommend always using `begin` and `end` statements, even though they are sometimes optional.

Always Block

The always block is a block of code where **order matters**:

```
always @(sensitivityList) begin
  /*Insert code here*/
end

//Example 1:
always @(*) begin
  case(exampleSelect)
    1'b1 : out = 1'b0;

    1'b0 : out = 1'b0;

    default : out = 1'bx;
end

//Example 2:
always @(posedge clk) begin
  register1 <= new_data;
  register2 <= load ? new_data : register2;
end
```

Notes:

- When **any** signal in `sensitivityList` changes, the always block will be executed. Signals inside `sensitivityList` are separated by commas.
  - The "wildcard" `@(*)` includes all signals
  - `posedge signal` activates the always block only when `signal` goes from low to high. Likewise, `negedge signal` activates the always block only when `signal` goes from high to low.
- Signals that are modified inside an always block **must** be defined as reg.
- Even though order matters inside an always block, its execution will not take any amount of time.[13]
- Reminder: case, casex and if statements can only be used inside always blocks. assign statements can never be used inside an always block.

[13] This is super confusing but just think about each statement taking a super tiny amount of time to execute, so that the entire block effectively takes no time to finish execution.

References

[1] T. Aamodt. (2021). Verilog for Combinational Logic. [Powerpoint Slides]
[2] T. Aamodt. (2021). Verilog for Finite State Machines. [Powerpoint Slides]