CME341 HDL Class Notes

Dr. Eric Salt

Originally created over 1 year ending September 8, 2016
Latest revision: September 3, 2021

Revised on:

October 2, 2016:	added verilog 2001 synthesis directive for full case
September 5, 2017:	Revised chapter 1 up to end of demo. Updated procedures for the new version of Quartus, Quartus Prime, and the new version of modelsim-altera, Modelsim-Altera 10.5b. Also removed references made to the old DE2 boards.
September 6, 2017:	Corrected a few typos
November 7, 2017:	Extended these typed notes to include the first part of the section on Sequential Circuits. The remainder of the notes are in the hand written version of Chapter 3
December 29, 2017:	Added instructions on saving and restoring the wave window in modelsim.
September 4+5, 2020:	Initial modifications for SystemVerilog support (Brian Berscheid)
September 21, 2020:	Fixed a few minor typos in Chapter 3. (Brian Berscheid)
September 3, 2021:	Added instructions to generate timing simulation netlist in Quartus 20. (Brian Berscheid)

Contents

Chapter 2

Numbers and Operators in Verilog HDL

2.1 Number Representation

Verilog supports numbers represented in binary, octal, decimal, or hexadecimal.

2.1.1 Unsigned Numbers

Number	#bits	Base	Decimal Equivalent	Value Stored
2'b10	2	binary	2	10
3'd5	3	decimal	5	101
3'o5	3	octal	5	101
8'o5	8	octal	5	00000101
8'ha	8	hexadecimal	10	00001010
3'b01x	3	binary	–	01x
12'hx	12	hex	–	xxxxxxxxxxxx
8'hz	8	hex	–	zzzzzzzz
8'b0000_0001	8	binary	1	00000001
8'bx01	8	binary	–	xxxxxx01
'bz	unsigned (32 bits)	binary	–	zz...zzz(32)
8'HAD	8	hex	173	10101101
1	unsigned (32 bits)	decimal	1	0..01 (32 bits)

Example:

```
wire [3:0] x, y, z;
assign x = 4'b1011;
assign y = 4'd11;
assign z = 4'hB;
```

2.1.2 Signed Numbers

Recall that, like decimal numbers, the worth of a digit in a binary number depends on its position relative to the binary point. For unsigned binary numbers, which can only represent positive numbers, the first digit to the left of the binary point is worth $2^0 = 1$, the second digit is worth $2^1 = 2$ and the k^{th} digit is worth 2^{k-1}. For example 10110. has decimal equivalent $0\times2^0+1\times2^1+1\times2^2+0\times2^3+1\times2^4 = 22$.

Signed binary numbers, which can represent both positive and negative numbers, use the same format with one exception. The most significant digit is worth the negative of that of an unsigned number. Therefore, the decimal worth of 10110. is $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 - 1 \times 2^4 = -10$.

Since the worth of the most significant bit depends on whether the number is signed or unsigned, Verilog must know which it is when performing an operation such as a comparison or multiply. To inform the Verilog HDL compiler that the number is to interpreted as a signed number an "s" in front of the base indicator. For example 3'sb101 is interpreted as a signed number and it evaluates to the decimal number -3. It will be stored as 101, but interpreted as having a decimal worth of -3.

In Verilog a minus sign placed in front of a number means "take the two's complement" whether or not that number is signed. If the number is a signed number then the $-$ sign has its usual meaning. For Example:
-4'd5 indicates the binary number of interest is the two's complement of 5, therefore $-4'\text{d}5 = -4'\text{b}0101 = 4'\text{b}1011$, which has a decimal worth of $1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 11$

Placing a negative sign in front of an unsigned number The compiler still treats it as an unsigned number, but a value that is the two's complement of what is written.

In Verilog, all values are stored in binary format. The difference between signed and unsigned numbers is in the way they are treated in Verilog operations.

Example:

Verilog	Stored Value	Decimal Equivalent
2'sb10	10	-2
2'b10	10	2
-4'sd5	1011	-5
4'sd5	0101	5
4'shB	1011	-5
-4'shB	0101	5

NOTE: The most significant bit is called the sign bit.

Example:

```
3'd7  * 3'd2  = 6'b001110
  7   *   2   =     14

3'sd7 * 3'sd2 = 6'b111110
 -1   *   2   =     -2
```

2.1.3 Variable Declaration For Signed Values

By default, declarations of type wire and reg are **unsigned**. To make the variables signed the word "signed" has to be placed after the words "wire" and "reg" as shown below.

```
reg [3:0] x;        // (4 bit unsigned)
wire [4:0] y;       // (5 bit unsigned)
reg signed [3:0] z; // (4 bit signed)
wire signed [2:0]w; // (3 bit signed)
integer i;          // (32 bits signed)
```

2.1.4 Arithmetic With Signed Values

When a Verilog operation requires an operand be extended, signed numbers are extended with the sign bit.

Example:

$$\begin{gathered} 3\text{'sd7} + 4\text{'sd2} \\ \downarrow \\ 3\text{'sb111} + 4\text{'sb0010} \\ \downarrow \\ 4\text{'sb1111} + 4\text{'sb0010} \\ \downarrow \\ -1 + 2 = 4\text{'b0001} \end{gathered}$$

2.1.5 Casting and Typing Numbers

The type of a number, i.e. signed or unsigned, can be overridden using the functions $signed(x) and $unsigned(x). This does not change any of the bits in the number so the numbers are not converted from signed to unsigned or visa versa. It simply tells the compiler what sign to use for the most significant bit.

Example:

```
input signed [3:0] x1;
input [3:0] x2; // x2 is unsigned by default
output [4:0] y;

assign y = x1 + $signed(x2); // treat x2 as signed for this
                     // operation only
```

The vector x2 is treated as a signed number, the worth of the most significant bit is to be negative, but for just the one statement in which $signed(x2) is used. If x2 is used in a subsequent statement it will be treated as an unsigned number, unless of course its type is overridden with $signed(x2).

2.2 Operators

2.2.1 Types of Verilog Operators

Bitwise Operators

Bitwise operators perform the operation bit wise on a pair of vectors. The result of a bitwise operation is a vector the same size as the operands.

Assume:

```
y is a 4 bit vector (output).
x is a 1 bit vector (output).
a = 4'b1001 (input)
b = 2'b01 (input)
```

Bitwise Logic Operators			
Operator		Example	Result
~	Negation	assign y = ~b;	y = 4'b1110
&	AND	assign y = a & b;	y = 4'b0001
|	OR	assign y = a|b;	y = 4'b1001
^	XOR	assign y = a^b;	y = 4'b1000
~^	XNOR	assign y = a~^ b;	y = 4'b0111

Rules for bitwise operators:

1. Any operands shorter than the output variable are extended to the same length of the output variable. If the operand is unsigned it is extended by placing zeros in the most significant bits. If the operand is signed, it is sign extended (operand is extended leftward by repeating its most significant bit).
2. In the event the output is shorter, only the right most bits of the result are assigned.

Reduction Operators (Unary Operators)

Reduction operators have multiple inputs, but a single bit output. If an output of a reduction operation is assigned to a vector with more than 1 bit, the result is placed in the right most bit and the other bits are stuffed with 0's.

Reduction Operators (Unary Operators)			
Operator		Example	Result
&	AND	assign x = &a;	x = 1'b0
~&	NAND	assign y = ~&a;	y = 4'b0001
|	OR	assign y = |a;	y = 4'b0001
~|	NOR	assign x = ~|a;	x = 1'b0
^	XOR	assign x = ^ a;	x = 1'b0
~^	XNOR	assign x = ~^ a;	x = 1'b1

NOTE: Result is 1 bit and assigned to the right most bit of assign variable.

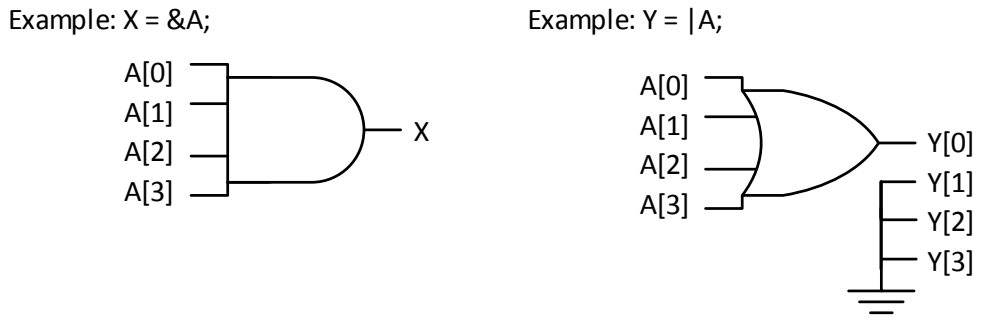

Figure 2.1: Reduction Operation Examples

Relational Operators

$<$	Less than
$<=$	Less than or equal to
$>$	greater than
$>=$	greater than or equal to
$==$	equal
$!=$	not equal

Relational operators have logic or number inputs, but produce a boolean result. A boolean result is either a "true" or a "false" as opposed to a "0" or a "1". **If the relational opertor requires numbers for its operands, like the operator >, the operands are treated as unsigned integers unless <u>BOTH</u> operands are signed.**

The effect can be illustrated using 4 variables. Suppose `x` and `y` are unsigned and have decimal worths of 11 and 4, respectively, and suppose `x1` and `y1` are signed and have decimal worths of -5 and 4, respectively, then these variables would be created in Verilog by.

```
assign x = 4'b1011;
assign y = 4'b0100;
assign x1 = 4'sb1011;
assign y1 = 4'sb0100;
```

Notice that `x` and `x1` have different decimal worths, but the same binary digits. The relational expression `(x < y)`, where both operands are unsigned, maps to the decimal comparison $(11 < 4)$ and produces a false result as expected. The comparison `(x1 < y1)`, where both operands are signed, maps to the decimal comparison $(-5 < 4)$ and produces a true result as expected. However, the comparison `(x1 < y)`, where the two operands have different signs, maps to $(11 < 4)$ and produces an unexpected result of false.

There are two other relational operators which are === and !==. They equate "x" and "z" states as well. They are not normally used in synthesis and will not be used in this class.

Examples:

Suppose x, y, and w are unsigned variables and suppose a and b are signed variables. Suppose the five variables have been assigned values:
x = 4'b1001,
y = 4'b011,
w = -4'b0001,
a = 4'sb1001, and
b = -4'sb0111.
Evaluate the expressions below:

1. (x < y) evaluates to false
2. (x < w)
   (4'b1001 < -4'b0001)
   (4'b1001 < 4'b1111) evaluates to true

3. (x < 3'b111)
   (4'b1001 < 4'b0111) evaluates to false

4. (x < -3'b111)
   (4'b1001 < 3'b001)
   (4'b1001 < 4'b0001) evaluates to false

5. (a < 3'b111)
   (4'sb1001 < 4'b0111)
   (4'b1001 < 4'b0111) evaluates to false
   Since one of the values is unsigned, all become unsigned (a is interpreted as 9 instead of -7).

6. (a < 3'sb111)
   (4'sb1001 < 4'sb1111) evaluates to true.
   Since both operands are signed, this is extended with the sign bit to become 4'b1111 and is treated as -1.

7. (x < 3'sb111)
   (4'b1001 < 4'b0111) evaluates to false.
   Since at least one operand is unsigned this is treated as 3'b111 so it is extended to 4 bits with zero padding to become 4'b0111 and is treated as 7.

Logical Operators

Logical operators have two operands, both are boolean (true or false). They also produce a boolean result.

```
!   logical negation
&&  logical and
||  logical or
```

NOTE: If a vector, say 'x', is placed where a boolean variable is expected, then Verilog interprets 'x' as the boolean value obtained from the expression (|x == 1'b1). It is better not to trust the Verilog compiler to figure out what was intended. Instead of using

```
if(x)
```

it is better to use

```
if(x != 4'h0)
```

Concatenation Operator

The concatenation operator makes a vector by stringing other vectors together. The concatenation operation is invoked by enclosing the vectors that are to be strung together in curly brackets, i.e. { }, and separating them by commas.

Example:

```
C = {A, B}
```

The length of C = length of A + length of B

if

```
A = 4'b1011
B = 2'b01
```

then

```
C = 6'b101101
```

If one variable is replicated several times within a concatenation operator the notation can be shorted using a replication factor. The expression `C = {B, B, B}` could be expressed as `C = {3{B}}`. The number 3 in front of `B` is called the replication factor. Using B from the example above has `C = {3{B}}` resulting in `C = 6'b010101`.

Example:

```
C = {A, {2{B}}} results in C = 8'b10110101
```

Conditional Operator

The conditional operator is basically the same as an if-else operator. The operator tests a boolean variable, which is usually the result of an expression involving a relational operator, and assigns one value if the boolean variable is true and another result if it not.

The conditional operator is symbolized as a question mark. It has format.

```
Y = (A <= B) ? A: B;
```

If the boolean result of the expression to the left of the question mark is true `Y` is assigned `A` otherwise `Y` is assigned `B`.

It is important to note that `Y` does not have to be and usually is not assigned a boolean value. `A`, `B` and `Y` can be vectors. Furthermore, A and B can be replaced with expressions.

2.2.2 Arithmetic Operators

+	addition
−	subtraction
∗	multiplication
/	division
%	modulus

NOTE: many compilers do not support modulus and division.

Multiplication Operator

Multiplication is indicated by placing a ∗ between numbers or variables. The number of bits in the result is the sum of the number of bits of the operands. The ∗ operator treats the inputs as integers and delivers an integer result. The result is truncated to fit the size of the output variable.

If both operands are signed then they are treated as such and the output is a signed number. If one of the operands is unsigned, then both are treated as unsigned and the result is unsigned.

2.2.3 Shift Operator

The shift operations in Verilog are done with wiring. They do not build a shift register. Furthermore, the left shift behaves differently than that of a computer. However, the right shift behaves the same. The main difference is a left increases the word size where as a right shift does not.

There are two types of shift operators: logical and arithmetic. A logical shift to the left is indicated by << and a logical shift to the right is indicated by >>. No information is lost in a logical shift to the left, but information is lost in a logical shift to the right. Some of the least significant bits are always lost in a right shift. The syntax for a shift of two bits to the left is "`assign y = { x << 2 };`". In effect this appends two "0's" to `x` adding two least significant bits and then connects the extended `x` to `y` lining up the least significant bit of the extended `x` and `y`. Clearly the length of the extended `x` is two bits longer than `x`.

The syntax for a shift of two bits to the right is `assign y = { x >> 2 };`. In effect this removes the two least significant bits of `x` and extends the truncated `x` leftwards by prepending two "0's" to make the shifted `x` the same as its original size.

Example:

```
wire [3:0] x;
wire [5:0] y, y1;
assign x = 4'b1001;
assign y = {x << 2};
assign y1 = {x >> 2};
```

The values of `y` and `y1` are `y` = 6′b100100 and `y1` = 6′b000010.

A arithmetic shift is indicated by >>> or <<<. An arithmetic left shift is exactly the same as a logical left shift. An arithmetic right is is the same except the value prepended is that of the most significant bit of `x`.

Example:

```
wire [3:0] x;
wire [5:0] y, y1;
assign x = 4'b1001;
assign y = {x <<< 2};
assign y1 = {x >>> 2};
```

The values of `y` is `y` = 6′b100100 and, because the most significant bit of `x` is 1, the value of `y1` is `y1` = 6′b111110.

2.2.4 Operator Precedence

Operators in order of precedence:

Operator	Symbols
Unary, Multiply, Divide, Modulus	$!, \sim, *, /, \%$
Add, Subtract, Shift	$+, -, <<, >>$
Relation, Equality	$<, >, <=, >=, ==, != , ===, != =$
Reduction	$\&, !\&, \hat{}, \hat{}\sim, \mid, \sim\mid$
Logic	$\&\&, \mid\mid$
Conditional	? :