CME 341: Verilog Numbers and Operators

Brian Berscheid

Department of Electrical and Computer Engineering
University of Saskatchewan

Today's agenda

1. Assignment Operator (=)
2. Number Formats
3. Operators

Introduction + Motivation

- We saw previously that there are three main ways of constructing logic in Verilog:
  - ▷ Explicit structural verilog
  - ▷ **Implicit structural verilog**
  - ▷ **Behavioral verilog**

- Implicit and behavioral methods generally involve performing some operations to modify a number or signal, then assigning it to a variable
  - ▷ The power of these methods comes from the flexibility provided by the operators and number systems built into verilog
  - ▷ In order to be able to code efficiently and avoid bugs, we need to understand clearly how the operators and number formats work

    Sometimes they are a bit counter-intuitive!

Assignment Operator (=)

Assignment operator (=)

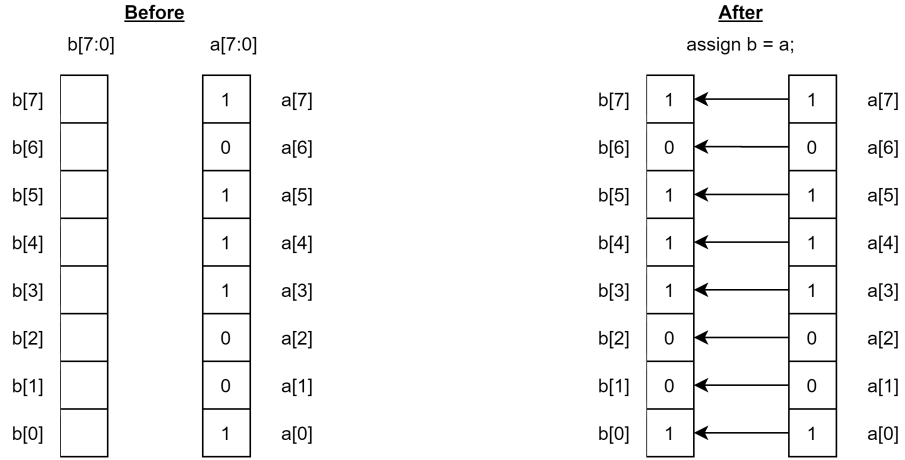
- A fundamental operation in verilog is assigning a value to a variable:

```
assign target = source;   (implicit structural code) or
       target = source;   (behavioral code)
```

- In general, both `target` and `source` may be multi-bit signals (vectors)
- If the bit lengths are the same, the functionality of this operator is obvious: the individual bits are copied from target to value
- What if the lengths are not the same?
  - ▷ Potential for confusion;
    misunderstanding the following rules is the source of many bugs!
- *Aside: The = operator is also called the 'blocking assignment operator'. There is also a 'non-blocking assignment operator' <= which obeys the same rules except when used in a block of code. It will be studied in Chapter 3.*

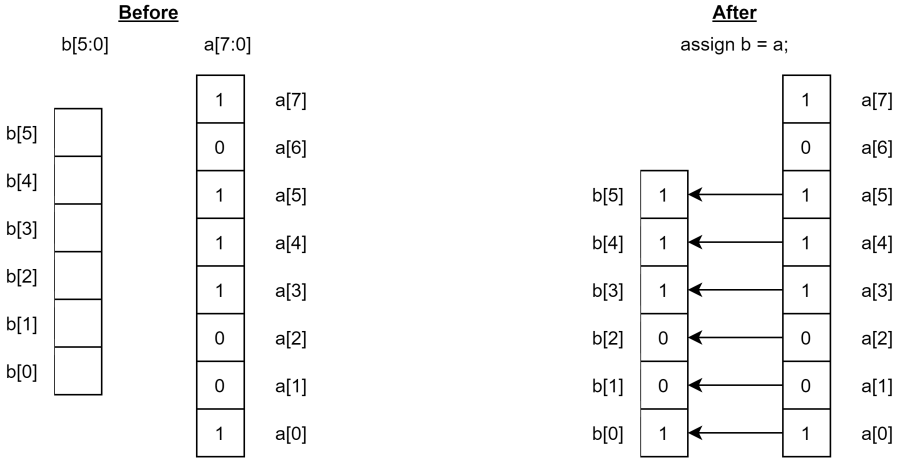
Assignment operator example 1

Target and source have same length



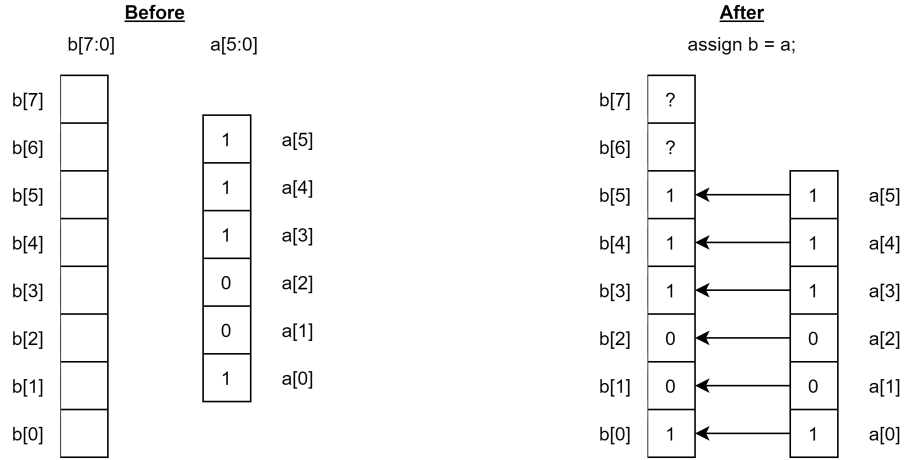
Assignment operator example 2

Source longer than target ... assignment starts by matching LSBs



Assignment operator example 3

Target longer than source ... what happens to MSBs? Need to understand number formats!



Number Formats

Overview of number formats

- Verilog allows the user to specify numbers in a variety of formats (binary, decimal, hexadecimal, octal)
- Numbers and variables can also be defined as signed or unsigned (default)
- Numbers are specified in verilog as follows: `<# bits>'<format specifier><value>`
- Valid format specifiers:
  - ▷ `b`: binary
  - ▷ `d`: decimal
  - ▷ `h`: hexadecimal
  - ▷ `o`: octal

Example number declarations

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)

Number storage

- Ultimately, all numbers are stored in binary format (digital logic); other formats are provided for convenience only

- Example: all wires below get the exact same values

```
wire [5:0] r1, r2, r3;
assign r1 = 6'b010101;
assign r2 = 6'd21;
assign r3 = 6'h15;
```

Signed numbers

- A number can be specified as signed by adding s to its format specifier
- Again, this does not change the bit values that are stored
- Reminder from EE 232: signed binary numbers use 2's complement format:
  - ▷ MSB has negative worth of 2^{N-1} for a length-N vector
  - ▷ All other bits have positive worth

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

Signed variables

- Variables can be defined as signed by using the `signed` keyword in the declaration

- Otherwise, variables will be unsigned by default:

```
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)
```

Major differences between unsigned and signed operations

- MSB extension when one operand is longer than another
  - ▷ Zero-extension used for unsigned operations
  - ▷ Sign-extension used for signed operations

```
wire [3:0] x1, x2;
assign x1 = 2'b10;  // 10 --> 0010 ;  2 -->  2 (zero extension)
assign x2 = 2'sb10; // 10 --> 1110 ; -2 --> -2 (sign extension)
```

- Multiplication and division require different circuits for signed and unsigned operation
- Comparison circuits ($<$, $>$, etc) depend on whether the number is signed or unsigned
- In CME 341, we will mainly be using unsigned numbers. Signed numbers will be important in the future if you are in the DSP stream!

Signed vs unsigned numbers (continued)

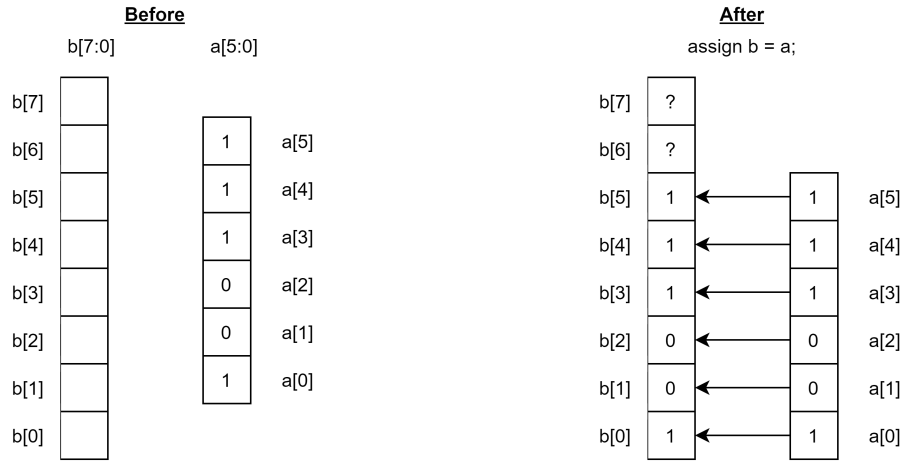
- For operations with two operands (addition, comparisons, etc.), both operands need to be signed in order to build signed hardware... otherwise unsigned hardware will be built

- When performing an assignment, verilog looks at the RHS (source) to determine whether an operation is signed or unsigned, not the LHS.

```
wire signed [3:0] x1, x2, x3;
assign x1 = 2'b10 + 2'b11;   // both operands unsigned: zero extension
                             // 0010 + 0011 = 0101
                             // 2    +  3   = 5
assign x2 = 2'sb10 + 2'b11;  // one operand unsigned: zero extension
                             // 0010 + 0011 = 0101
assign x3 = 2'sb10 + 2'sb11; // both operands signed: sign extension
                             // 1110 + 1111 =   11101 --> 1101
                             // -2   + -1   =   -3
```

Assignment operator example 3 revisited

Target longer than source ... what happens to MSBs? What information is needed to answer?



Operators

Verilog operators

- Verilog provides many operators that can be used to manipulate signals and variables
- Many of these are similar in form and function to conventional programming languages
  - ▷ To obtain the full power of Verilog, we must understand these operators
- Major categories of operators:
  - ▷ Bitwise
  - ▷ Reduction
  - ▷ Relational
  - ▷ Logical
  - ▷ Concatenation
  - ▷ Arithmetic
  - ▷ Shift

Bitwise operators

- Process two equal-length[1] vectors, generate an output vector of same length
  - ▷ Exception is negation operator ($\sim$) : one input, one output of same length
- Examples:

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

Operator		Example	Result
$\sim$	Negation	assign y = $\sim$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
$\sim$^	XNOR	assign y = a$\sim$^b;	y = 4'b0111

[1] If one input is shorter than the other, extension will be performed as discussed in the number formats section.

Reduction operators

- Single multi-bit vector input → single bit output
- Operation performed between all bits of input vector

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

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

Relational operators

- Accept two vector inputs, perform standard mathematical comparisons between numbers
  - ▷ Output will be either true (0) or false (1)
- Remember: behavior of operators depends on whether operands are signed or unsigned
- List of relational operators:
  - ▷ $<$ less than
  - ▷ $<=$ less than or equal to
  - ▷ $>$ greater than
  - ▷ $>=$ greater than or equal to
  - ▷ $==$ equal to
  - ▷ $!=$ not equal to

Relational operators: examples

Try to determine the answers yourself

- Assume `x, y, w` are 4-bit unsigned variables, and `a` is a 4-bit signed variable, assigned the following values:

```
assign x = 4'b1001;
assign y = 4'b0011;
assign w = -4'b0001;
assign a = 4'sb1001;
```

- Evaluate the following relational expressions:
  1. `(x < y)`
  2. `(x < w)`
  3. `(x < 3'b111)`
  4. `(x < -3'b111)`
  5. `(a < 3'b111)`
  6. `(a < 3'sb111)`
  7. `(x < 3'sb111)`

Relational operators: examples

Answers

- Assume `x, y, w` are 4-bit unsigned variables, and `a` is a 4-bit signed variable, assigned the following values:

```
assign x = 4'b1001;
assign y = 4'b0011;
assign w = -4'b0001;
assign a = 4'sb1001;
```

- Evaluate the following relational expressions:

```
1.  (x < y)          ----> 4'b1001  < 4'b0011  --> FALSE
2.  (x < w)          ----> 4'b1001  < 4'b1111  --> TRUE
3.  (x < 3'b111)     ----> 4'b1001  < 4'b0111  --> FALSE
4.  (x < -3'b111)    ----> 4'b1001  < 4'b0001  --> FALSE
5.  (a < 3'b111)     ----> 4'b1001  < 4'b0111  --> FALSE
6.  (a < 3'sb111)    ----> 4'sb1001 < 4'sb1111 --> TRUE
7.  (x < 3'sb111)    ----> 4'b1001  < 4'b0111  --> FALSE
```

Logical and conditional operators

- Logical operators accept boolean / single-bit inputs and produce a boolean / single-bit output
  - ▷ `!` logical negation operator (inverts a single boolean input)
  - ▷ `&&` logical AND (ANDs two boolean inputs together)
  - ▷ `||` logical OR (ORs two boolean inputs together)

  Example : `((x > 4'b0111) && (y == 4'd5))`

- Conditional operator accepts a boolean / single-bit input and outputs one of two values, depending on whether the boolean is true

  Syntax: `(boolean expression) ? value_if_true : value_if_false`
  Example: `assign y = (x < 4'b0111) ? 2'b00 : 2'b10;`

Arithmetic operators

- These perform basic arithmetic operations, and are largely self-explanatory
- Remember that they will perform unsigned operations unless both operands are signed
- List of arithmetic operators:
  - ▷ $+$ addition
  - ▷ $-$ subtraction
  - ▷ $*$ multiplication
  - ▷ $/$ division (may not work in synthesizable code)
  - ▷ % modulus (may not work in synthesizable code)

Concatenation operator

- Concatenation operator creates vectors by packing other signals / vectors together
- Surround expression by { } and separate entries with commas
- Examples:

```
wire [3:0] y1;
wire [5:0] y2;
wire [1:0] x1, x2;
assign x1 = 2'b10;
assign x2 = 2'b11;

assign y1 = {x1, x2};       // y1 gets 4'b1011;
assign y2 = {x2, x1, x2};   // y2 gets 6'b111011;
```

Replication operator

- Replication operator repeats a variable within a concatenation multiple times:
  Syntax: `{number{variable}}`
- Examples:

```
wire [3:0] y1;
wire [5:0] y2;
wire [9:0] y3;
wire [1:0] x1, x2;
assign x1 = 2'b10;
assign x2 = 2'b11;

assign y1 = {2{x1}};         // y1 gets 4'b1010;
assign y2 = {x2, {2{x1}}};   // y2 gets 6'b11_1010;
assign y3 = {5{x1}};         // y3 gets 10'b10_1010_1010
```

Shift operators

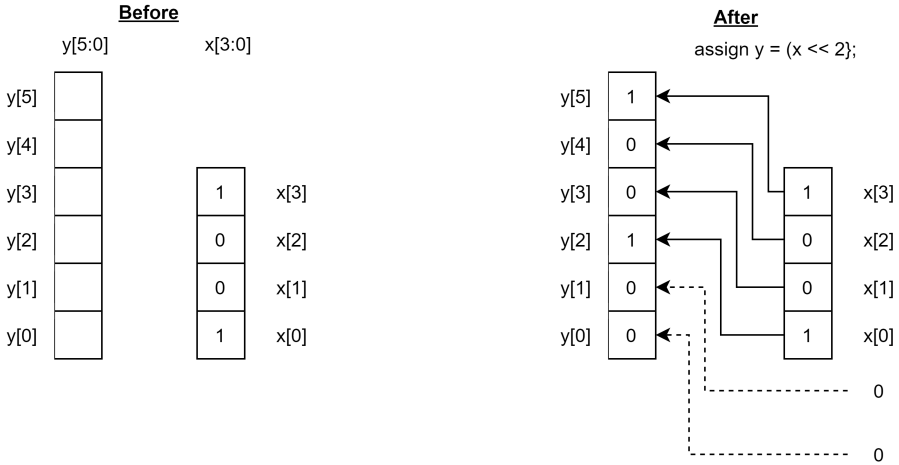
- Shift operators apply bit shifts to an input vector
- There are four main shift operators:
  - ▷ $<<$ logical shift left
  - ▷ $>>$ logical shift right
  - ▷ $<<<$ arithmetic shift left
  - ▷ $>>>$ arithmetic shift right
- When shifting left, the unpaired LSBs are connected to 0.
- When shifting right, some LSBs are lost. Any unpaired MSBs are:
  - ▷ Filled with 0's if a logical shift was used
  - ▷ Filled with sign bits if an arithmetic shift was used

Shift operators: examples

```
wire [3:0] x;
wire [5:0] y, y1, y2, y3;
assign x = 4'b1001;
assign y = {x << 2};    // y  gets 6'b100100
assign y1 = {x >> 2};   // y1 gets 6'b000010
assign y2 = {x <<< 2};  // y2 gets 6'b100100
assign y3 = {x >>> 2};  // y3 gets 6'b111110
```

Shift operators: implementation

- The shift operator is implemented by cross-wiring signals. No logic resources are consumed.



One final tip

- It is easy to create bugs in your code by forgetting the details of how an operator works
  - ▷ This happens to even experienced Verilog coders
- If you are not sure about how an operator will behave, a good strategy is to construct a testbench and run a Modelsim simulation to check the values generated by an operation.
  - ▷ Perform the operation in question inside the testbench and assign it to a wire(s).
  - ▷ No DUT is needed (no need to compile in Quartus), just compile the testbench in Modelsim and view the result in the Modesim wave window.
  - ▷ You can set up a general purpose testbench and Modelsim project for this type of syntax checking and reuse it throughout the term.

Thank you!
Have a great day!