Operator Types

Verilog provides the following operator types:

1. Arithmetic
2. Logical
3. Relational
4. Equality
5. Bitwise
6. Reduction
7. Shift
8. Concatenation
9. Replication
10. Conditional
Verilog Symbols

• Verilog only uses standard ASCII symbols.
• No operators or anything else use special symbols not found on a standard keyboard.
• Easy to enter Verilog code, but
  – Leads to dual use of symbols
  – Assigns different meaning to some familiar symbols
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>**</td>
<td>exponent</td>
</tr>
</tbody>
</table>
Multiplication

• Hardware multiplication algorithms have improved, making using the multiply operator practical.

• Synthesizers will select a suitable hardware multiplier (modified Booth Array, most likely) from a “synthetic library” of scalable components.

• Thus PROD <= Multiplier * Multiplicand is now an acceptable coding style. “Beating the machine” is tough.
Division

Divide by $2^n$ (fixed divisor, shift right) has always been supported for synthesis. This will always work:

```vhdl
input [7:0] DIVIDEND;
wire [7:0] QUOTIENT;
assign QUOTIENT = DIVIDEND / 4;
/*Resolved power of two: 4 may be replaced by a parameter but not by a variable.*/
```
Integer Division

• Now integer division has some synthesis support.
• Quotient = Dividend / Divisor may work in some tools for any integers.
• Needs to be verified before use.

input [3:0] DIVISOR;
input [7:0] DIVIDEND;
reg [7:0] QUOTIENT;
always @(DIVIDEND or DIVISOR)
    QUOTIENT = DIVIDEND / DIVISOR;
More on Division

• At best, integer division will work for both 2’s compliment and unsigned numbers.

• At worst, using a variable for divisor will produce a compile error: entirely tool dependent.

• Integer and real division can be freely used in test fixtures and when using Verilog as a general-purpose programming language.
Sign Bit

Suppose the objective is to divide 10010 by 2
Shift right: 10010 → ☹ 1001
What should ☹ be? 1 or 0? Maybe Verilog x?

☺, because nearly every key on a normal keyboard means something in Verilog.
Assume 2’s Compliment

$10010_2 = -14$

Shift right one time: $01001$ or $11001$?

$01001_2 = 9$

$11001_2 = -7$

Division must **extend sign bit**.

This behavior always needs to be verified before depending on it in a circuit.
Signed Variables

• Verilog 2001 introduced signed variables and operators.
• No need to manually extend sign bits when using them.
• More on signed variables shortly.
• Do need to write code to manually extend sign bit when using normal variables.
Modulus

- Modulus operator produce the remainder from the division of two numbers. The result takes the sign of the first operand.

- Examples:
  
  \[
  14 \% 3 \quad \text{produces} \quad 2 \\
  9 \% 2 \quad \text{produces} \quad 1 \\
  -5 \% 3 \quad \text{produces} \quad -2 \\
  15 \% -2 \quad \text{produces} \quad 1 \\
  -5 \% -3 \quad \text{produces} \quad -2
  \]
Integer Division

// Not guaranteed to work for synthesis!!!

input [7:0] DATA1, DATA2;
output [7:0] QUO, REM;
reg [7:0] QUO, REM;
always @(DATA1 or DATA2) begin
    QUO = DATA1 / DATA2;
    REM = DATA1 % DATA2;
end
Floating Point

- Sorry, no such luck.
- Need to code up implement mantissa, exponent algorithms manually.
- Maybe someday.
Addition

- Can be used freely on regs and wires.
- $C <= A + B$ works fine.
- A suitable adder style will be chosen from the universe of adder styles. Synthesizer will look at design constraints and make a selection.
- Generally two’s compliment is assumed.
Subtraction

No tricks here, just like addition.
always @(A or B) C = A – B;
Exponential

- Generally *unsupported* for synthesis, though could be used in test fixtures.
- Part of Verilog 2001: not supported by all simulators, much less synthesizers.
- Example: `NUM <= BASE ** EXPON;`
- Obvious hardware limitation—numbers get large, need big registers for results.
Unary Arithmetic Operators

• The operators + and - also work as unary operators. In this case, they have higher precedence than the binary + and -.

• Examples:
  -2 + 17 produces 15
  +5 - 14 produces -9
  b + -a is legal too, though annoying and ridiculous.
Logical Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Operators

• Logical operators treat their operands as either true or false (logic 1 or logic 0).

• They return a 1-bit result
  – 1’b1, 1’b0 or 1’bx if results are ambiguous

• Thus (2’b01 && 2’b10) is true because both operands are non-zero. Logic 1 is returned.

• It is equivalent to writing
  – (2’b01 != 0) && (2’b10 != 0)
Logical Operators

• Logical operators always produce 1-bit results which could be 0 (false), 1 (true), or x (ambiguous).
• Logical operators accept variables or expressions as operands.
• If an operand is equal to zero, it is considered as logical 0 (false) otherwise it is considered as logical 1 (true).
• Logical operators are usually used to form conditions for conditional if statement or conditional assignment operator.
Logical Operand

• The operands used in a logical operation evaluate as zeros if and only if each bit is zero.
• Otherwise it is a one.
• Thus a logic AND of two operands will result in a logic 1 result if both operands have at least one bit set.
Not A Bitwise Comparison

reg [3:0] X, Y;
reg Z;
always @(X or Y) Z = X && Y;

initial begin
  X = 4'b1010; Y = 4'b0101; //Z = 1
  #10 X = 4'hf; Y = 4'h1; //Z = 1
  #10 X = 0; //Z = 0
end
Logical operators

Examples:

\[
\begin{align*}
  x &= 3 & y &= 0 & w &= 2'b0x \\
  !x & \quad \text{produces} & 0 \\
  x && y & \quad \text{produces} & 0 & (1 \text{ AND } 0) \\
  x || y & \quad \text{produces} & 1 & (1 \text{ OR } 0) \\
  x && w & \quad \text{produces} & x & (1 \text{ AND } x)
\end{align*}
\]
Logical Operator Usage

Normally used in testing true/false conditions:

if (IN1 && IN2) begin
  if (SUM > 0 || PROD < 8’h0F) begin
    if (STATE == RESET || STATE == IDLE)
      MOTOR_CONTROL <= 8'b0;
    else if (!OVERRIDE)
      MOTOR_CONTROL <= MOTOR_CONTROL + 1;
  end
end
# Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>~&amp;</td>
<td>bitwise NAND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td><del>^ or ^</del></td>
<td>bitwise XNOR</td>
</tr>
</tbody>
</table>
Bitwise Operators

• Bitwise operators take each bit in one operand and perform the operation with corresponding bit in the other operand.

• If one operand is shorter, it will be extended with zeros to match the length of the longer operand.

• In bitwise operation, a z is treated as an x.
Bitwise Operators

\[ w = 5'b11111 \; ; \; v = 4'b01x1; \; k = 3'b011 \; ; \]
\[ m = 3'b11z \; ; \]
\[ (w \& v) \; \text{evaluates to} \; 001x1 \]
\[ (w | v) \; \text{evaluates to} \; 11111 \]
\[ (v \^ w) \; \text{evaluates to} \; 110x0 \]
\[ (v \& m) \; \text{evaluates to} \; 10xx \]
\[ (~v) \; \text{evaluates to} \; 10x0 \]
Bitwise Operators

The Logical Equations:

\[ X = \overline{A} \overline{B} + A \overline{B} \]
\[ Y = A \oplus B \]

Will be represented in Verilog by:

assign \( X = \overline{A} \& B \mid A \& \overline{B} \);
assign \( Y = A \bowtie B \);
Bit-Wise Negation vs. Logical Negation

Examples:

\[ A = 5'b10101; \quad B = 4'b0; \]

\[ !A \quad \text{evaluates to} \quad 0 \]
\[ \sim A \quad \text{evaluates to} \quad 01010 \]
\[ !B \quad \text{evaluates to} \quad 1 \]
\[ \sim B \quad \text{evaluates to} \quad 1111 \]
Bitwise vs. Logical Operators

• Both perform standard Boolean operations.
  – AND, OR, etc.

• Logical Operators are used for comparisons and generate a true/false condition.

• Bitwise operators can give multi-bit answers to Boolean operations.
One-bit Operands

• Logical and bitwise operators perform identically on single bit operands.

• It’s still good policy and style to use logical operators when testing conditions and bitwise operators when performing Boolean operations.
  – if (!RST) begin //if RST == 0
  – CLK = ~CLK; //Invert value of CLK

• Some authors recommend only using bitwise operators, never logical at all.

Reduction Operators

- Yet another form of Boolean operators, this one, like Logical Operators, only producing a single bit output.
- Same symbols, different application.
- Reduction operators only operate on a single operand.
Reduction Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
</tr>
<tr>
<td>~</td>
<td>reduction OR</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
</tr>
<tr>
<td><del>^ or ^</del></td>
<td>reduction XNOR</td>
</tr>
</tbody>
</table>

Reduction operators use the same symbols as bitwise operators but are unary.
AND Reduction

reg [5:0] AND_BUS;

wire AND_RED;

assign AND_RED = &AND_BUS;
XOR Reduction

• XOR Reduction is just another way of saying positive parity generator.

\[
\text{PARITY} \leq \text{\textasciitilde DATA\_BUS};
\]
Reduction Operators

- Reduction operators perform a bitwise operation on the bits of a single operand to produce a 1-bit result.

- Examples:
  
y = 6’b101000 ; w = 6’b111000
  
  &y evaluates to 0
  |w evaluates to 1
  ^y evaluates to 0
  ^w evaluates to 1
  ~^w evaluates to 0
Homework 2

• Is due now.
Exam 1

• Midterm exam 1 will be held on Tuesday, October 6.
• Bring a blue book.
• There will be little to nothing to calculate. However, you may bring a simple calculator.
• No other electronic devices are permitted.
Exam 1

• No electronic devices
  – This explicitly includes telephones, even if used only as a watch or calculator.

• Homework/review will be held next time.

• Bring questions.

• There will be no lab exam day. Lab reports will be due the following week.
# Relational Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>
Relational Operators

- Relational operators are used in comparison expressions.
- If the relation is **true**, it evaluates to logical 1 and if it is **false** it evaluates to logical 0.
- If any of the operands contains x or z, the relation evaluates to x.
Relational Operators

Examples:

```
if x = 4; y = 8; w = 3'b11z; v = 2'b0x
```

- \( x < y \) evaluates to \( \text{Logical 1} \)
- \( y <= x \) evaluates to \( \text{Logical 0} \)
- \( x > v \) evaluates to \( x \)
- \( y >= w \) evaluates to \( x \)
Resolving Unknown Bits

- Relational operators do not resolve if either operand contains an unknown bit.
- This is inconsistent with the way equality operators work.
- Thus $1000 > 000x$ will return unknown, even if the answer appears obvious.
- Just have to live with it.
module resolve;

wire [3:0] X = 4'b1000;
wire [3:0] Y = 4'b000x;

assign A = X > Y;
assign B = X === Y;
assign C = X == Y;

initial $strobe ("A = %b B = %b C = %b", A, B, C);
endmodule
Less Than or Equal/Non-Blocking Assignment

• $\leq$ means less than or equal relationship op.
• $\leftarrow$ is the non-blocking assignment operator.
• They are exactly the same symbol.
• The compiler will sort it out.
Using Conditionals

Conditionals are used for testing.

Example:

```vhdl
if (ENABLE && STATE <= MAX_STATE) //conditional
    RUN <= `TRUE; //assignment operator
else if (ENABLE)
    RUN <= `FALSE;
else
    RUN <= RUN;
```

//VHDL designers note lack of ‘null’ keyword.
# Shift Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;&gt;</code></td>
<td>right shift</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>left shift</td>
</tr>
</tbody>
</table>

**Examples:**

```
w = 6'b111111;

v = w >> 1 will set v to 011111
v = w << 1 will set v to 111110
v = w << 3 will set v to 111000
```
Shift Operators

- Note that shifts do not wrap around or sign extend.
- Shifted bits just wind up in the bit bucket.
- Wrap or sign extend options must be added manually.
- Fills with zeros, from the right or from the left.
Shift Right
\[ v <= w >> 1 \]

Eight-bit Register

Initial value: 8’h66
Final value: 8’h33
Shift Hardware

- Shift operation implies a shift register: will thus perform multiply/divide by 2 in a single clock cycle.
- Barrel shifters can also easily be coded using shift operators, but the synthesized hardware gets factorially more complicated.
Arithmetic Shift Operators

Verilog 2001 adds Arithmetic Shift Operators:

Arithmetic Shift Right: >>>

Arithmetic Shift Left: <<<<

Arithmetic Shift preserves sign bit. It works with signed operators, which were also introduced with Verilog 2001.
module shiftest(DIST, DIN, DOUT1, DOUT2);

input signed [3:0] DIN;
input [1:0] DIST;
output signed [3:0] DOUT1, DOUT2;
reg signed [3:0] DOUT1, DOUT2;

always @(DIN or DIST) begin
    DOUT1 = DIN >> DIST;
    DOUT2 = DIN >>> DIST;
end
dendmodule
Signed Variables

- In the preceding example, inputs and outputs are signed integers.
- If they were not, the two shift operators would produce identical results: there would be no sign bit to extend.
- Since they are declared to be signed data, sign bits are preserved.
- Works with new version of simulator and synthesizer.
module shiftdemo;
  parameter WIDTH = 8;
  reg signed [WIDTH - 1 : 0] A, B, C, D, E;

  always @(E) begin
    A = E <<< 3;
    B = E << 3;
    C = E >>> 3;
    D = E >> 3;
  end

  initial begin
    E = 8'ha5;
    #0 $strobe("%d A = %h B = %h C = %h D = %h", $time, A, B, C, D);
    #1 E = 8'h5a;
    $strobe("%d A = %h B = %h C = %h D = %h", $time, A, B, C, D);
  end
endmodule
module shiftdemo;
    parameter WIDTH = 8;
    reg signed [WIDTH - 1 : 0] A, B, C, D, E;

    always @(E) begin
        A = E <<< 3;
        B = E << 3;
        C = E >>> 3;
        D = E >> 3;
    end

initial begin
    E = 8'ha5;
    #0 $strobe ("E = %b", E);
    $strobe("Hex: Time %d A = %h B = %h C = %h D = %h", $time, A, B, C, D);
    $strobe("Decimal: Time %d A = %d B = %d C = %d D = %d", $time, A, B, C, D);
    $strobe("Binary: Time %d A = %b B = %b C = %b D = %b", $time, A, B, C, D);
    #1 E = 8'h5a;
    $strobe ("E = %b", E);
    $strobe("Hex: Time %d A = %h B = %h C = %h D = %h", $time, A, B, C, D);
    $strobe("Decimal: Time %d A = %d B = %d C = %d D = %d", $time, A, B, C, D);
    $strobe("Binary: Time %d A = %b B = %b C = %b D = %b", $time, A, B, C, D);
endmodule
Simulation Results

# E = 10100101
# Hex: A = 28 B = 28 C = f4 D = 14
# Decimal: A = 40 B = 40 C = -12 D = 20
# Binary: A = 00101000 B = 00101000 C = 11110100 D = 00010100
# E = 01011010
# Hex: A = d0 B = d0 C = 0b D = 0b
# Decimal: A = -48 B = -48 C = 11 D = 11
# Binary: A = 11010000 B = 11010000 C = 00001011 D = 00001011
Conclusions

• Shift left: signed and unsigned work the same.

• Shift right: unsigned fills with zeros, signed extends sign bit.

• Negative numbers (- sign) only show up when decimal numbers are used.

• That’s just for display.
Concatenation Operator {, }

- The concatenation operator (braces) is used to append multiple operands to produce one operand.
- The operands must be sized. They can be sized constants, 4’h6, bit select, R[5], part-select, R[5:2], scalar net, scalar register, vector nets or vector registers.
Concatenation Operator \{, \}

- Examples:
  - A= 5’b10110 ;   B=3’b000 ;   C=4’b0101
  - Y= \{A , B\} will set Y to 8’b10110000
  - Y= \{A , B[2:1],C[0]\} will set Y to 8’b10110001
  - Y= \{A[4] , B[2], 3’O6\} will set Y to 5’b10110
Replication Operators \{\#\{\},\}\}

To repeat a concatenation a number of times, we precede each bracket by a replication number.

Example:

```
L=1’d1; M=1’b0; K= 3’b010; N= 4’hA ;
```

```
Y= \{ 2\{L\}, 2\{M\}, K, N\}\} \rightarrow Y=11’b1100010101010
Y= \{ 3\{M\}, L, K[1:0], N[3:1]\}\} \rightarrow Y=9’b00011010101
Replicate 0 Times

• What happens if the replication constant is zero?

• Can happen if replication constant is a parameter or variable.

• In Verilog ‘95, it was undefined. Tools generally replaced it with a one bit 0.
  – \{\text{MYVAR}, \{0\{\text{YOURVAR}\}\}\}\} becomes \{\text{MYVAR}, 1’b0\}
Replicate 0 Times, 2001

- Correct operation would be to have replicated operand disappear.
- Verilog 2001 does something different: it just declares a 0 replication constant to be illegal, a syntax error.
- Verilog 2005 specification (not yet available in tools) says it should disappear.
Conditional Operator “?:”

- The conditional operator is used in expressions and has the following syntax:
  \( Y = \text{Condition}\_\text{Expr} \ ? \ \text{Expr\_true\_cond} : \text{Expr\_false\_cond} \)

- The \text{Condition}\_\text{Expr} is first evaluated.

- If it is \textbf{true}, the \text{Expr\_true\_cond} is evaluated and its value is assigned to \( Y \).

- If the \textbf{Condition}\_\text{Expr} is \textbf{false}, the \text{Expr\_false\_cond} is evaluated and its value is assigned to \( Y \).
Conditional Operator “?:”

If the value of Condition.Expr is undetermined (x), then both Expr_true_cond and Expr_false_cond are evaluated and their values are compared bit by bit. This will return the value of the two corresponding bits if they are the same and x if they different. The final result will then be assigned to Y.
Conditional Operator “?:”

- assign out = sel ? In1 : In0;

- assign data_bus = drive_enable ? ALU_out : 16’bz;
Conditional Operator “?:”

assign out = sel1 ? (sel0 ? In3:In2) : (sel0 ? In1:In0);

For 8-to1 MUX:
assign out = sel2 ? (sel1 ? (sel0 ? In7:In6) : (sel0 ? In5:In4)):
   (sel1 ? (sel0 ? In3:In2) : (sel0 ? In1:In0)) ;
## Operator Precedence In Verilog

<table>
<thead>
<tr>
<th>Type of operators</th>
<th>Examples</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate &amp; replicate</td>
<td><code>{}</code> <code>{}</code> <code>{}</code></td>
<td>highest</td>
</tr>
<tr>
<td>Unary</td>
<td><code>!</code> <code>~</code> <code>+</code> <code>-</code></td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td><code>*</code> <code>/</code> <code>%</code></td>
<td></td>
</tr>
<tr>
<td>Logical shift</td>
<td><code>&lt;&lt;</code> <code>&gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td><code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code></td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td><code>==</code> <code>===</code> <code>!=</code> <code>!==</code></td>
<td></td>
</tr>
<tr>
<td>Binary bit-wise</td>
<td><code>&amp;</code> `</td>
<td><code> </code><del>&amp;<code> </code></del></td>
</tr>
<tr>
<td>Reduction</td>
<td><code>&amp;</code> `</td>
<td><code> </code><del>&amp;<code> </code></del></td>
</tr>
<tr>
<td>Binary logical</td>
<td><code>&amp;&amp;</code></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td><code>?:</code></td>
<td>lowest 67</td>
</tr>
</tbody>
</table>
Operator Precedence In Verilog

Example:

\[ A = 6'b110110; \ w = 6'b111111; \]

\[ v = ^ A \ & \ w >> 3; \]

// \( v = ^ {110110} \ & \ {111111 >> 3}; \)   // shift

// \( v = ^ {110110} \ & \ {000111}; \)   // bit-wise AND

// \( v = ^ {000110}; \)   // Reduction XOR

// \( v = 0; \)   // Final result
Use Parenthesis to Clarify Logic

\[
v = ^\wedge A \ & \ w >> 3; \ //all \ but \ incomprehensible
v = ^\wedge (A \ & \ (w\gg3));\ //easy \ to \ understand
\]

The meaning of the second one obvious. Without parenthesis, the logic is identical but the meaning is obscure.
Parenthesis Limitation

- Use of parenthesis can prevent the synthesizer from reordering logic for timing optimization.
  \[ \text{SUM} \leq A \cdot B + C \cdot D + E + F + G \]
  \[ \text{SUM} \leq (A \cdot B) + ((C \cdot D) + (E + F) + G) \]
- Same Boolean logic, but first one allows operator reordering to allow for different signal arrival times. Second is equally correct, but may produce worse results.