EE 3755

Verilog handout #2

```
/// LSU EE 3755 -- Fall 2001 -- Computer Organization
//
/// Verilog Notes 2 -- Expressions

/// Contents

// Continuous Assignment
// Operators:  Bitwise Logical
// Vectors, Bit Selects, and Part Select
// Sized Numbers and Constants (Parameters)
// Operators: Bitwise with Vectors
// Operators: Arithmetic
// Operators: Logical AND, etc
// Operators: Equality, Comparison
// Operator:  Conditional
// Operators: Shift, Cat
// Operator Precedence and Association
// ALU Example

/// References

// :P:   Palnitkar, "Verilog HDL"
// :Q:   Qualis, "Verilog HDL Quick Reference Card Revision 1.0"
// :H:   Hyde, "Handbook on Verilog HDL"
// :LRM: IEEE, Verilog Language Reference Manual (Hawaii Section Numbering)
// :PH:  Patterson & Hennessy, "Computer Organization & Design"



////////////////////////////////////////////////////////////////////////////////
/// Continuous Assignment

// :P: 6.1     Good description.
// :H: 2.7.2   Material covered out of order.
// :LRM: 6.1   Advanced

// Continuous assignment specifies a value for a wire.
//
// Continuous assignments can be made using the /assign/ keyword and in a
// wire [or other net] declaration.

// :Keyword: assign
//
// :Sample:  assign foo = bar;
//

// :Example:
//
// A simple use of the assign keyword.  Output x is given
// the value of "a".  Whenever "a" changes "x" also changes.
// Don't forget that: Whenever "a" changes "x" also changes.
//
// The module below passes the signal through unchanged, not very
// useful other than as a simple example of assign.

module really_simple(x,a);
   input a;
   output x;

   assign x = a;

endmodule


// :Example:
//
// The module below also passes a signal through the module unchanged,
// but this time it goes through an extra wire, w.  Whenever "a"
```

```
// changes, w will change and that will change x.

module something_similar(x,a);
   input a;
   output x;

   wire   w;

   assign w = a;
   assign x = w;

endmodule


// :Example:
//
// The module below is like the one above except that
// continuous assignment to w is done in the wire declaration
// rather than with a separate assign statement.

module something_else_similar(x,a);
   input a;
   output x;

   wire   w = a;
   assign x = w;

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Operators:  Bitwise Logical

// :P: 6.3,6.4 Overview of expressions and operators.
// :P: 6.4.5   Bitwise logical operators.
// :H: 2.5.5   Bitwise logical operators.
// :LRM: 4.1   Includes all operators.

// Operators for performing bitwise logical operations.
//
// (The meaning of bitwise and logical will be explained in the Vector
// section below.)
//
//
// Operators:
//  &  bitwise and
//  |  bitwise or
//  ~  bitwise not
//  ^  bitwise exclusive or


// :Example:
//
// A module implementing: x = ab + c;

module my_umm_lost_count_module_redux(x,a,b,c);
   input a, b, c;
   output x;

   assign x = a & b | c;

endmodule


// :Example:
//
// Simple uses of the &, |, and ~ operators along with truth tables.
```

```
module operator_demo(x1,x2,x3,x4,a,b);
   input a, b;
   output x1, x2, x3, x4;

   // Bitwise AND.  (There is also a logical AND, discussed later.)
   //
   assign x1 = a & b;
   //
   // a  b  |  a & b
   // """""""""""""""
   // 0  0  |    0
   // 0  1  |    0
   // 1  0  |    0
   // 1  1  |    1

   // Bitwise OR.  (There is also a logical OR, discussed later.)
   //
   assign x2 = a | b;
   //
   // a  b  |  a | b
   // """""""""""""""
   // 0  0  |    0
   // 0  1  |    1
   // 1  0  |    1
   // 1  1  |    1

   // Bitwise XOR.
   //
   assign x3 = a ^ b;
   //
   // a  b  |  a ^ b
   // """""""""""""""
   // 0  0  |    0
   // 0  1  |    1
   // 1  0  |    1
   // 1  1  |    0

   // Bitwise NOT;
   //
   assign x4 = ~a;
   //
   // a   |  ~a
   // """""""""""
   // 0   |   1
   // 1   |   0

endmodule


// :Example:
//
// A binary-full adder using operators.  Some consider this more
// readable than the earlier version.  Note how spaces and returns
// (whitespace) are used to make the assignment to sum readable.

module bfa_implicit(sum,cout,a,b,cin);
   input a,b,cin;
   output sum,cout;

   assign sum =
          ~a & ~b &  cin |
          ~a &  b & ~cin |
           a & ~b & ~cin |
           a &  b &  cin;

   assign cout = a & b | b & cin | a & cin;
```

```
endmodule


////////////////////////////////////////////////////////////////////////////////
/// Vectors, Bit Selects, and Part Select

// :P: 3.2.4   Vectors, bit and part selects
// :H: 2.4.1   Includes other material
// :LRM: 3.3, 4.2

 /// Vectors
//
// So far only 1-bit wires have been shown.
//
// A /vector/ is a wire [or reg] that is more than on bit.
//
// The size and the bit numbering are specified in the declaration.
// (See examples.)

// :Sample:  wire [10:0] foo;
// :Sample:  output [10:0] bar;
//
// Declares output foo and bar with least-significant bit 0 and
// most-significant bit 10 (for a total of 11 bits).

 /// Bit and Part Selects
//
// A bit select is used to extract a single bit from a wire [or reg].
// :Sample:  assign x = foo[1]; // Extract bit 1.
//
// A part select is used to extract several consecutive bits from a
// wire [or reg].
// :Sample:  assign y = foo[5:3]; // Set y to 3 bits of foo, bits 5 to 3.


// :Example:

// This example does not use vectors. The author, who knew nothing
// about vectors, wanted a 4-bit input "a" and a 4-bit output x,
// plus a one-bit output msb.

module without_vectors(msb,x3,x2,x1,x0,a3,a2,a1,a0);
   output msb;
   output x3, x2, x1, x0;
   input  a3, a2, a1, a0;

   assign msb = a3;
   assign x3 = a3;
   assign x2 = a2;
   assign x1 = a1;
   assign x0 = a0;

endmodule


// :Example:
//
// This is the way the module above should be done.  Here input "a"
// and output "x" are declared as 4-bit vectors.  A bit select
// is used in the assignment to msb.

module with_vectors(msb,x,a);
   output msb;
   // Declare x as a 4-bit input, bit 3 is the most-significant bit (MSB).
   output [3:0] x;
   input  [3:0] a;
```

```
   assign       msb = a[3];  // Extract bit 3 from a, which is the MSB.
   assign       x = a;

endmodule


// :Example:
//
// This module does the same thing as the one above.  It is written
// differently: 0 is used to indicate the MSB (rather than 3).  Some
// prefer that the MSB be the highest-numbered bit, some like it to be
// the lowest-numbered bit.

module more_vectors(msb,x,a);
   output msb;
   output [0:3] x; // Here MSB is 0 and LSB is 3.
   input  [0:3] a;

   assign       msb = a[0];
   assign       x = a;

endmodule


// :Example:
//
// Here the vector declaration is used in a wire, rather than
// a port (input or output).

module and_more_vectors(x,a);
   input [15:0] a;
   output [15:0] x;

   wire [15:0]   w;

   assign        w = a;
   assign        x = w;

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Sized Numbers and Constants (Parameters)

// :P: 3.1.4, 3.2.8  Sized Numbers, Parameters
// :H: 2.2, 2.7.1    Sized Numbers, Parameters
// :LRM: 2.5, 3.10   Sized Numbers, Parameters

 /// Sized Numbers
// In Verilog numbers can be written with a specific size and using
// one of four radices (bases).

// The number consists of a size (in bits) followed by an apostrophe,
// followed by a b,o,d, or h (for binary, octal, decimal,
// hexadecimal), followed by the number.

 /// Constants
// Constants are specified using the parameter keyword.
// :Sample:  parameter limit = 5;
// This specifies a parameter named limit with a value of 5.
// It can be used in expressions but it cannot be assigned.
// For more information see parameter_examples module below.

// :Keyword:
// parameter

// :Example:
//
```

```
// The number one and the number ten are specified in a variety of
// ways.  Note the size of the wires and that each wire is assigned
// multiple times. (In this class, one should not assign the
// same wire multiple times. [Elsewhere, it might be done if
// the wire is a bus and is driven by tri-state devices.])

module fixed(one,ten);

   output [7:0] one;
   output [15:0] ten;

   // All assignments below do the same thing.  Use the one that's
   // easiest to read.
   //
   assign        one = 1;     // No size specified. Truncated from integer.
   assign        one = 8'b1;  // One specified in binary.
   assign        one = 8'b00000001;  // One specified in binary.
   assign        one = 4'b1;  // Sloppy, but works: Specified wrong size.
   assign        one = 8'd1;  // One specified in decimal.
   assign        one = 8'h1;  // One specified in hexadecimal;

   // All assignments below do the same thing.  Use the one that's
   // easiest to read.
   //
   assign        ten = 10;
   assign        ten = 16'b1010;
   assign        ten = 16'b0000000000001010;
   assign        ten = 16'd10;
   assign        ten = 16'ha;
   assign        ten = 16'h000a;

endmodule


// :Example:
//
// A sized number is used to specify the carry in of one of the bfas
// in a two-bit ripple adder.  It's important to use a sized number
// (rather than a regular number) for connections to modules.

module ripple_2(sum,cout,a,b);
   input [1:0] a, b;
   output [1:0] sum;
   output       cout;

   wire         c0;

   bfa_implicit bfa0(sum[0],c0,a[0],b[0],1'b0);
   bfa_implicit bfa1(sum[1],cout,a[1],b[1],c0);

endmodule


// :Example:
//
// A module illustrating the use of parameters (as constants).

module parameter_examples(an_input);
   input [7:0] an_input;

   parameter one = 1;

   // Parameters can be sized numbers.
   parameter two = 10'd2;

   // The value of seconds_per_day is computed once, at analysis
   // (usually compile) time.
   parameter seconds_per_day = 60 * 60 * 24;
```

```
   parameter width = 12;
   parameter height = 21;
   // The value of area is computed once, at analysis
   // (usually compile) time.
   parameter area = width * height;

   // The line below is an error because "an_input" is not a constant.
   parameter double_input = an_input * two;

endmodule


 /// More Information
//
//   Yes, "parameter" is a funny name for a constant. Though
//   parameters can be used as constants they can also be used
//   as parameters to a module, with values usually specified
//   in the instantiation.  This use of parameter will
//   not be covered in this course.


////////////////////////////////////////////////////////////////////////////////
/// Operators:  Bitwise with Vectors

// The same references as the earlier bitwise logical section.
// :P: 6.3,6.4 Overview of expressions and operators.
// :P: 6.4.5   Bitwise logical operators.
// :H: 2.5.5   Bitwise logical operators.
// :LRM: 4.1   Includes all operators.


// Operators &, |, ~, ^ are called /bitwise/ because when
// the operands are vectors the operation is done on each
// bit of the vector.

// :Example:
//
// An appropriate use of a bitwise operator, followed by
// code that does the same thing the hard way.

module and_again(x1,a,b);
   input [2:0] a, b;   // Both a and b are 3 bits.
   output [2:0] x1;

   // An appropriate use.
   //
   // Bitwise AND, but this time operands are 3-bit vectors.
   //
   assign x1 = a & b;  // ANDs three pairs of bits.

   // The same thing the hard way:
   //
   assign x1[0] = a[0] & b[0];
   assign x1[1] = a[1] & b[1];
   assign x1[2] = a[2] & b[2];
   //
   // Examples:
   //
   // a    b    |  a & b
   // """""""""""""""""
   // 000  000  |  000
   // 000  001  |  000
   // 010  111  |  010
   // 100  110  |  100

endmodule
```

```
}

////////////////////////////////////////////////////////////////////////////////
/// Operators: Arithmetic

// :P: 6.4.1
// :H: 2.5.1, 2.5.2
// :LRM: 4.1   Includes all operators.

// The arithmetic operators are:
//   + (addition), - (subtraction), * (multiplication), / (division),
//   and % (remainder).

// They operate on wires and data types to be covered later.


// :Example:
//
// A simple demonstration of arithmetic operators.

module arith_op(sum,diff,prod,quot,rem,a,b);
   input [15:0] a, b;
   output [15:0] sum, diff, prod, quot, rem;

   // For these operators vectors are interpreted as integers.

   // Addition
   assign sum = a + b;

   // Subtraction
   assign diff = a - b;

   // Multiplication
   assign prod = a * b;

   // Division
   assign quot = a / b;

   // Remainder (Modulo)
   assign rem = a % b;
   //
   // Examples:
   //
   // a    b    |  a % b
   // """""""""""""""""""
   // 3    5    |  3
   // 5    3    |  2
   // 7    7    |  0

endmodule


// :Example:
//
// The input to the module below is a complex number (specified
// with a real and imaginary part); the output is the square
// of that number.

module complex_square(sr,si,ar,ai);
   input [31:0] ar, ai;
   output [31:0] sr, si;

   assign       sr = ar * ar - ai * ai;
   assign       si = 2 * ar * ai;

   // Notes on code above:
   //
   // Multiplication has higher precedence than subtraction and so
   // subtraction done last.
```

```
   //
   // A constant is used in the si expression.

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Operators: Logical AND, etc

// :P: 6.4.2
// :H: 2.5.4
// :LRM: 4.1   Includes all operators.

// As in C, there is a logical AND operator, &&, that is different than
// the bitwise AND, &.
// Likewise there is a logical OR, ||, and a logical NOT, !.

// See the example.


// :Example:
//
// Illustration of logical operators, and how they are different than
// the bitwise operators.

module and_again_again(x1,x2,x3,x4,a,b);
   input [2:0] a, b;   // Both a and b are 3 bits.
   output [2:0] x1;
   output       x2, x3, x4;

   // Bitwise AND, but this time operands are 3-bit vectors.
   //
   assign x1 = a & b;  // ANDs three pairs of bits.
   //
   // The same thing the hard way:
   //
   assign x1[0] = a[0] & b[0];
   assign x1[1] = a[1] & b[1];
   assign x1[2] = a[2] & b[2];
   //
   // Examples:
   //
   // a    b    |  a & b
   // """""""""""""""""""
   // 000  000  |  000
   // 000  001  |  000
   // 010  111  |  010
   // 100  110  |  100

   // Logical AND. (Not to be confused with bitwise AND!)
   //
   assign x2 = a && b;
   //
   // Here both a and b are 3 bits, but x2 is a single bit.
   // Wire a is treated as logical true if any bit is 1;
   // it is treated as logical false if all bits are 0.
   //
   // Therefore x2 is set to 1 if a and b are true and to zero
   // if a or b is false.
   //
   // Examples:
   //
   // a    b    |  a && b
   // """""""""""""""""""
   // 000  000  |  0
   // 000  001  |  0
   // 010  101  |  1
```

```
    // 100  110  |  1

    // Logical OR. (That's right, not to be confused with bitwise OR.)
    //
    assign x3 = a || b;
    //
    // Operands are treated as logical values as described for &&.
    //
    // Wire x3 is set to 1 if a or b is true and to zero
    // if a and b are false.
    //
    // Examples:
    //
    // a    b    |  a || b
    // """""""""""""""""""
    // 000  000  |  0
    // 000  001  |  1
    // 010  101  |  1
    // 100  110  |  1

    // Logical NOT.
    //
    assign x4 = !a;
    //
    // Wire x4 is set to zero if any bit in a is 1,
    // x4 is set to one if all bits in a are zero.

endmodule


////////////////////////////////////////////////////////////////////////////////
// Operators:   Equality, Comparison

// :P: 6.4.3, 6.4.4
// :H: 2.5.3, 2.5.7
// :LRM: 4.1   Includes all operators.

// There are two equality operators, == and ===, and two inequality
// operators, != and !==.
//
// There are also comparison operators.
//
// See the example.


// :Example:
//
// Examples and description of the equality operators.

module equality(x1,x2,x3,x4,a,b);
   input [2:0] a, b;   // Both a and b are 3 bits.
   output x1, x2, x3, x4;

   // Logical Equality
   //
   assign       x1 = a == b;
   //
   // If a and b consist only of 0's and 1's:
   //   If a and b are identical, x1 is set to 1,
   //   if a and b differ, x1 is set to 0.
   // If a or b has an x or z, x1 is set to x.
   //
   // Examples:
   //
   // a    b    |  a == b
   // """""""""""""""""""
   // 000  000  |  1
```

```
   // 000  001  |  0
   // 001  001  |  1
   // x01  001  |  x
   // x01  x01  |  x       // Note: x is not a don't care (wildcard).

   // Logical Inequality
   //
   assign       x2 = a != b;
   //
   // Analogous to logical equality.
   //
   // Examples:
   //
   // a    b    |  a != b
   // """""""""""""""""""
   // 000  000  |  0
   // 000  001  |  1
   // 001  001  |  0
   // x01  001  |  x
   // x01  x01  |  x

   // Case Equality
   //
   assign       x3 = a === b;
   //
   // If a and b are identical (including x's and z's) x3 is 1 otherwise
   // x3 is set to zero.
   //
   // Examples:
   //
   // a    b    |  a === b
   // """""""""""""""""""""
   // 000  000  |  1
   // 000  001  |  0
   // 001  001  |  1
   // x01  001  |  0
   // x01  x01  |  1

   // Case Inequality
   //
   assign       x4 = a !== b;
   //
   // If a and b are identical (including x's and z's) x4 is set to 0 otherwise
   // x4 is set to one.
   //
   // Examples:
   //
   // a    b    |  a !== b
   // """""""""""""""""""""
   // 000  000  |  0
   // 000  001  |  1
   // 001  001  |  0
   // x01  001  |  1
   // x01  x01  |  0

endmodule


// :Example:
//
// Illustration and explanation of the comparison operators.

module compare(x1,x2,x3,x4,a,b);
   input [7:0] a, b;   // Both a and b are 8 bits.
   output x1, x2, x3, x4;

   // Take note: Comparison (and other) operators interpret contents of a
   // vector (e.g., a and b) as unsigned.  You've been warned.
```

```
// Greater Than
//
assign       x1 = a > b;
//
//
// Examples:
//
// a    b    |  a > b
// """"""""""""""""""""
// 3    1    |  1
// 1    3    |  0
// 4    4    |  0
// x    4    |  x
// -1   3    |  1   See note below.
//
// The 8-bit 2's complement representation of -1 is 8'b11111111
// which is identical to the 8-bit representation of 255, 8'b11111111.
// The comparison operand will always interpret contents of a vector
// as an unsigned integer.

// Greater Than or Equal To
//
assign       x2 = a >= b;
//
//
// Examples:
//
// a    b    |  a >= b
// """"""""""""""""""""
// 3    1    |  1
// 1    3    |  0
// 4    4    |  1
// x    4    |  x
// -1   3    |  1   See note above.

// Less Than
//
assign       x3 = a < b;
//
//
// Examples:
//
// a    b    |  a < b
// """"""""""""""""""""
// 3    1    |  0
// 1    3    |  1
// 4    4    |  0
// x    4    |  x
// -1   3    |  0   See note above


// Less Than or Equal To
//
assign       x4 = a <= b;
//
//
// Examples:
//
// a    b    |  a <= b
// """"""""""""""""""""
// 3    1    |  0
// 1    3    |  1
// 4    4    |  1
// x    4    |  x
// -1   3    |  0   See note above.

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
/// Operator: Conditional

// :P: 6.4.10
// :H: 2.5.7
// :LRM: 4.1   Includes all operators.

// Those who understand C's conditional operator ?:, can quickly scan this.
 /// OTHERS SHOULD PAY CLOSE ATTENTION.
//
// :Sample: assign v =  x ? y : z
// If x is 1 assign y to v, if x is 0 assign z to v.
// :Sample: assign v =  x < 3 ? y + 1 : z - x;
// If x < 3 assign y+1 to v; if x >= 3 assign z-x to v.


// :Example:
//
// A simple illustration of the conditional operator.

module conditional(x,a,b,s);
   input [31:0] a, b;
   input        s;
   output [31:0] x;

   // The Conditional Operator
   //
   // Don't skip this one.
   //
   assign        x = s ? a : b;
   //
   // Examples:
   //
   // a  b  s  | s ? a : b
   // ##################
   // 11 22 0  | 22   (b selected if s == 0)
   // 11 22 1  | 11   (b selected if s != 0 and not unknown)
   // 11 22 x  | x    (It's a bit more complicated.)

endmodule


// :Example:
//
// Use of the conditional operator to implement a four-input
// multiplexor.  Make sure this example is understood.  This is no
// time to be shy about asking for help.

module mux4(x,a,b,c,d,select);
   input [31:0] a, b, c, d;
   input [1:0]  select;
   output [31:0] x;

   assign        x =
                 select == 0 ? a :
                 select == 1 ? b :
                 select == 2 ? c : d;

   // Notes on example above:
   //
   // "select == 0" evaluates to 0 (if select isn't 0) or to 1 (if select is 0).
   //
   // The conditional operator associates right to left, unlike the
   // other operators.

endmodule
```

```
// :Example:
//
// Another use of the conditional operator.  In this one parenthesis
// are necessary.  Take some time to figure out what would happen if
// they were missing.

module add_and_maybe_scale(x,a,b,adjust_a,adjust_b);
   output [15:0] x;
   input [15:0]  a, b;
   input         adjust_b, adjust_a;

   assign        x = ( adjust_a ? a + 5 : a ) + ( adjust_b ? b + 5 : b );

   // In example above parentheses are needed, otherwise the middle "+"
   // would grab the "a" away from the first conditional.

endmodule



////////////////////////////////////////////////////////////////////////////////
/// Operators: Shift, Concatenate

// :P: 6.4.7, 6.4.8
// :H: 2.5.7
// :LRM: 4.1   Includes all operators.

// The shift operators, << and >>, perform left and right shifts.
// The concatenation operator, {}, makes a large vector out of
// its arguments.
//
// See examples for further description.


// :Example:
//
// Illustration and description of shift operators.

module shifts(x1,x2,a,s);
   input [15:0] a;
   input [2:0]  s;
   output [15:0] x1, x2;

   // The Left Shift Operator
   //
   assign        x1 = a << s;
   //
   // Examples:
   //
   // a    s    |  a << s
   // """"""""""""""""""""
   // 01011 0   | 01011    // Note: shown in binary
   // 01011 1   | 10110
   // 01011 2   | 01100
   // 01011 7   | 00000

   // The Right Shift Operator
   //
   assign        x2 = a >> s;
   //
   // Examples:
   //
   // a    s    |  a >> s
   // """"""""""""""""""""
   // 11011 0   | 11011    // Note: shown in binary
   // 11011 1   | 01101
```

```
   // 01011 2    | 00010
   // 01011 7    | 00000


endmodule


// :Example:
//
// Use of the shift operator to multiply by four.
// The alert students ... (continued below)

module a_times_4_plus_b(x,a,b);
   input [7:0] a, b;
   output [10:0] x;

   assign        x = ( a << 2 ) + b;

endmodule

// ...will have realized that two input bits to this module
// can be omitted, in which case the shift wouldn't be necessary
// either.


// :Example:
//
// Illustration and description of the concatenate operator.

module cat(x,a,b);
   input [7:0] a, b;
   output [15:0] x;

   // The Concatenation Operator
   //
   assign      x = { a, b };
   //
   // Examples:
   //
   // a    b    |  { a, b }
   // """""""""""""""""""""
   // 0000 1111 | 00001111
   // 1010 1111 | 10101111

   // Note: Constants within the concatenation operator must be sized,
   // for example, 5'd3 is okay but 3 is not.

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Operator Precedence and Association

// :P: 6.4.11
// :H: 6.4.8
// :LRM: 4.1 Table 4-4.

// Precedence and association specify for which operator an operand is used.
// See Examples.

///  Operator Precedence
//  + - ! ~ (unary)            highest precedence (Strongest Binding)
//  *  /  %
//  +  - (binary)
//  <<  >>
//  <  <=  >  >=
//  ==  !=  ===  !==
```

```
//  &  ~&
//  ^  ^~
//  |  ~|
//  &&
//  ||
//  ?: (conditional operator)  lowest precedence  (Weakest Binding)

//  Association is from left to right except conditional (?:).

// :Example:
//
// Examples illustrating precedence and association.  This example
// uses behavioral code which has not been covered yet but should
// be easy enough to figure out. (The example was borrowed from EE 4702
// notes.)

module precedence_and_association_examples();

   integer x, a, b, c, d, e, temp;
   integer i_am_a_condition_for_operator_to_the_right;
   integer i_am_the_else_part_of_the_operator_to_the_left;

   initial begin

      /// Precedence Examples

      // Multiplication has higher precedence than addition so multiply first.
      x = a + b * c;  // Multiply then add.
      x = ( a + b ) * c;  // Add then multiply.

      // Comparison (<,>) has higher precedence than logical or (||),
      // so compare first, then or the results.
      if( a < b || c > d ) $display("Condition true.");

      // Logical equality (==) has higher precedence than bitwise xor (^).
      if( a ^ b == c ) $display("This is probably an error.");
      if( ( a ^ b ) == c ) $display("This is probably okay.");

      // Unary negation (!) has higher precedence than logical or (||).
      if( !a || b ) $display("(Not a) or b.");
      if( !(a || b) ) $display("Not (a or b)");

      // The conditional operator has the lowest precedence, lower than addition.
      // if ( a < 10 ) x = b; else x = c + 1;
      x = a < 10 ? b : c + 1;
      // if ( a < 10 ) x = b + 1; else x = c + 1;
      x = ( a < 10 ? b : c ) + 1;

      /// Association Examples

      // All operators associate left-to-right except ?:.

      // Addition and subtraction have the same precedence, so the
      // leftmost operation is performed first.
      x = a + b - c; // Add then subtract.
      x = a - b + c; // Subtract then add.

      // The conditional operator has right-to-left association. (All
      // other operators associate left to right.)
      x =   a ? b : i_am_a_condition_for_operator_to_the_right       ? d : e;
      x = ( a ? b : i_am_the_else_part_of_the_operator_to_the_left ) ? d : e;

      // The three lines below do the same things.
      x = a < 10 ? 10 : a < 20 ? 20 : 30;
      x = a < 10 ? 10 : ( a < 20 ? 20 : 30 );
      if( a < 10 ) x = 10; else begin if ( a < 20 ) x = 20; else x = 30; end
```

```
      // The first line below and the next two lines do the same
      // probably-not-too-useful thing. (Which is why conditionals associate
      // right to left.)
      x = ( a < 10 ? 10 : a < 20 ) ? 20 : 30;
      if( a < 10 ) temp = 10; else temp = a < 20;
      if ( temp ) x = 20; else x = 30;

   end

endmodule // precedence_examples

 /// Note on Style
//
// Avoid unnecessary parenthesis for common operators.
// For example use:
//   x = a + b * c;
// Do not use:
//   x = a + ( b * c );
// The second assignment does exactly the same thing as the first but
// the parenthesis add clutter which can become cumbersome in complex
// expressions.
//
// Note that the Palnitkar text and Hyde notes specify the opposite style,
// and so would prefer the second assignment.



////////////////////////////////////////////////////////////////////////////////
/// ALU Example

// :PH: 4.5  An example of a different ALU.  Covered in a later set.

// An example of a CPU arithmetic and logical unit (ALU).
// ALUs will be discussed in future lectures.

// An ALU typically has two operand inputs (a and b below),
// a control input (op below), and a result output.  It
// can perform several arithmetic and logical operations,
// the operation to perform is specified by the op input.

// There are two ALU modules below, one uses the conditional
// operator, the other uses a multiplexor module, which
// is also defined.

// :Example:
//
// ALU using the conditional operator.

module alu(result,a,b,op);
   input [31:0] a, b;
   input [3:0]  op;
   output [31:0] result;

   parameter     op_add = 0;
   parameter     op_sub = 1;
   parameter     op_or  = 2;
   parameter     op_a   = 3;  // Result is a (unmodified).
   parameter     op_b   = 4;  // Result is b (unmodified).
   parameter     op_srl = 5;  // Shift a right logical (no sign extension).
   parameter     op_sra = 6;  // Shift a right arithmetic (sign extension).
   parameter     op_rr  = 7;  // Rotate a right.

   assign result =
          op == op_add ? a + b          :
          op == op_sub ? a - b          :
          op == op_or  ? a | b          :
          op == op_a   ? a              :
          op == op_b   ? b              :
```

```
          op == op_srl ? a >> 1           :
          //  op == op_sra ? {a[31], a[31:1] } : // Equivalent
          op == op_sra ? {a[31], a >> 1 } :
          op == op_rr  ? {a[0], a >> 1 }  : 0;

endmodule


// :Example:
//
// A multiplexor with eight 32-bit inputs.  To be used in the
// ALU below.

module mux8x32(o,s,i0,i1,i2,i3,i4,i5,i6,i7);
   input [31:0] i0,i1,i2,i3,i4,i5,i6,i7;
   input [2:0]  s;
   output [31:0] o;

   assign o =
          s == 0 ? i0 :
          s == 1 ? i1 :
          s == 2 ? i2 :
          s == 3 ? i3 :
          s == 4 ? i4 :
          s == 5 ? i5 :
          s == 6 ? i6 : i7;

endmodule


// :Example:
//
// An ALU using the multiplexor.  Note that the constants are not used
// and that it is much harder to tell if a particular operation is in
// the correct multiplexor input.

module alu2(result,a,b,op);
   input [31:0] a, b;
   input [2:0]  op;
   output [31:0] result;

   // Parameters not used here.
   parameter     op_add = 0;
   parameter     op_sub = 1;
   parameter     op_or  = 2;
   parameter     op_a   = 3;  // Result is a (unmodified).
   parameter     op_b   = 4;  // Result is b (unmodified).
   parameter     op_srl = 5;  // Shift a right logical (no sign extension).
   parameter     op_sra = 6;  // Shift a right arithmetic (sign extension).
   parameter     op_rr  = 7;  // Rotate a right.

   mux8x32 m1(result, op,
              a + b,
              a - b,
              a | b,
              a,
              b,
              a >> 1,
              {a[31], a >> 1 },
              {a[0], a >> 1 }
              );

endmodule
```