/// LSU EE 3755 -- -- Computer Organization

//

/// Verilog Notes 2 -- Expressions

/// Contents



/// References

// :P: Palnitkar, "Verilog HDL"

// :Q: Qualis, "Verilog HDL Quick Reference Card Revision 1.0"

// :PH: Patterson & Hennessy, "Computer Organization & Design"

////////////////////////////////////////////////////////////////////////

/// Continuous Assignment

```
// :P: 6.1
// Continuous assignment specifies a value for a wire using an
// expression.
//
//
// Continuous assignments can be made using the /assign/ keyword and in a
// wire [or other net] declaration.


// :Keyword: assign
//
// :Sample:  assign a = b;
//
// Here "b" is the expression.
// The left hand side of an assignment always should be a wire[net].
// :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.


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

```
  output x;

  assign x = a;

endmodule


// :Example:
//
// The module below is functionally equivalent to the one above.

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

  and a1(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
// :P: 6.4.5
// 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_abc_module(x,a,b,c);
  input a, b, c;
  output x;

  assign x = a & b | c;

endmodule


// :Example:
//
// The module below is functionally equivalent but less readable than
// the one above.

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

  wire  ab;
```

```
  and a1(ab,a,b);
  or o1(x,ab,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 &in |
```

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

// :Sample:  wire [10:0] a;
```

```
// :Sample:  output [10:0] b;
//
// Declares wire a and output b 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 = a[1]; // Extract bit 1.
//
// A part select is used to extract several consecutive bits from a
// wire [or reg].
// :Sample:  assign y = a[5:3]; // Set y to 3 bits of a, 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
//
```

```
// 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.
//
// :Sample: 16'd5
//
// The number five using 16 bits written in decimal.
//
// :Sample: 16'b101
//
// The number five represented using 16 bits written in binary.

/// Constants
//
// Constants are specified using the /parameter/ keyword.
//
// :Sample:  parameter limit = 5;
//
// This specifies a parameter named limit with a value of 5.
// Identifier limit can be used in expressions but 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.
  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 bfa's
// 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

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

// The same references as the earlier bitwise logical section.
//
```

```
// :P: 6.3,6.4
// :P: 6.4.5


// 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
//

// 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
// 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. (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
```