Verilog

- Guidelines
- Lab Exercise 1
- Verilog Basics

- Part 5 and Part 6 suggestions
- Quartus warnings guidelines (on Moodle)
- Verilog style guidelines (on Moodle)

Thursday

Laboratory Exercise 1

Switches, Lights, and Multiplexers

The purpose of this exercise is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices. We will use the switches SW_{17-0} on the DE2 board as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

The DE2 board provides 18 toggle switches, called SW_{17-0}, that can be used as inputs to a circuit, and 18 red lights, called $LEDR_{17-0}$, that can be used to display output values. Figure 1 shows a simple Verilog module that uses these switches and shows their states on the LEDs. Since there are 18 switches and lights it is convenient to represent them as vectors in the Verilog code, as shown. We have used a single assignment statement for all 18 *LEDR* outputs, which is equivalent to the individual assignments

```
assign LEDR[17] = SW[17];
assign LEDR[16] = SW[16];
...
assign LEDR[0] = SW[0];
```

The DE2 board has hardwired connections between its FPGA chip and the switches and lights. To use SW_{17-0} and $LEDR_{17-0}$ it is necessary to include in your Quartus II project the correct pin assignments, which are given in the *DE2 User Manual*. For example, the manual specifies that SW_0 is connected to the FPGA pin *N25* and $LEDR_0$ is connected to pin *AE23*. A good way to make the required pin assignments is to import into the Quartus II software the file called *DE2_pin_assignments.csv*, which is provided on the *DE2 System CD* and in the University Program section of Altera's web site. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction using Verilog Design*, which is also available from Altera. Page 22 of AlteraQuartusII_Introduction.pdf

It is important to realize that the pin assignments in the *DE2_pin_assignments.csv* file are useful only if the pin names given in the file are exactly the same as the port names used in your Verilog module. The file uses the names *SW*[0] ... *SW*[17] and *LEDR*[0] ... *LEDR*[17] for the switches and lights, which is the reason we used these names in Figure 1.

```
// Simple module that connects the SW switches to the LEDR lights
module  part1 (SW, LEDR);
    input [17:0] SW;        // toggle switches
    output [17:0] LEDR;     // red LEDs

    assign LEDR = SW;
endmodule
```

Figure 1. Verilog code that uses the DE2 board switches and lights.

Perform the following steps to implement a circuit corresponding to the code in Figure 1 on the DE2 board.

1. Create a new Quartus II project for your circuit. Select Cyclone II EP2C35F672C6 as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a Verilog module for the code in Figure 1 and include it in your project.
3. Include in your project the required pin assignments for the DE2 board, as discussed above. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Lab 1, Part 1, cont.

Pin Assignments

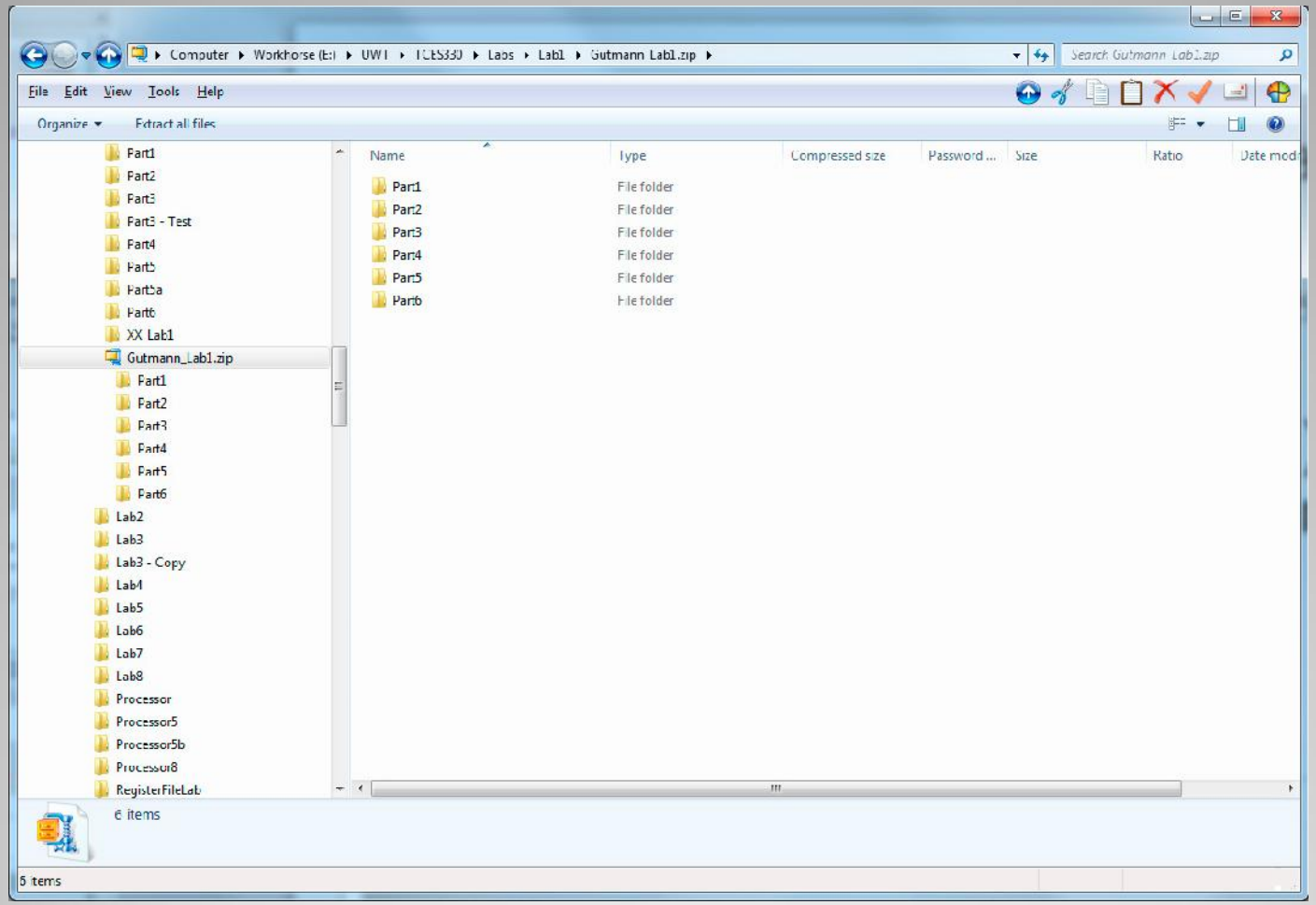
- Could use the Assignment Editor (like we did in Lab 0)
- Easier to import pin assignments specific to the DE2 board
- Demo

Turn In for Homework 1

- Six Verilog projects called Part1, Part2,..., Part6
- Zip all six projects together and turn in to Moodle.
  - All parts should be at the top level
  - It's a good idea to put your name and 'HW1' somewhere in the zip file name.

Homework Conventions

- Compress with zip not rar or anything else.
- Use .v as the extension for Verilog files and not anything else
- When you turn in Homework1 (Altera's Lab 1) make sure your zip file looks like the example (next chart).
- Also you can try out the script HW1 script.



Zip File Format

- HW1 Help File for Lab 1
- Simple Quartus Project Setup Video

Help File for Assignment 1 (Lab 1)

Verilog Basics

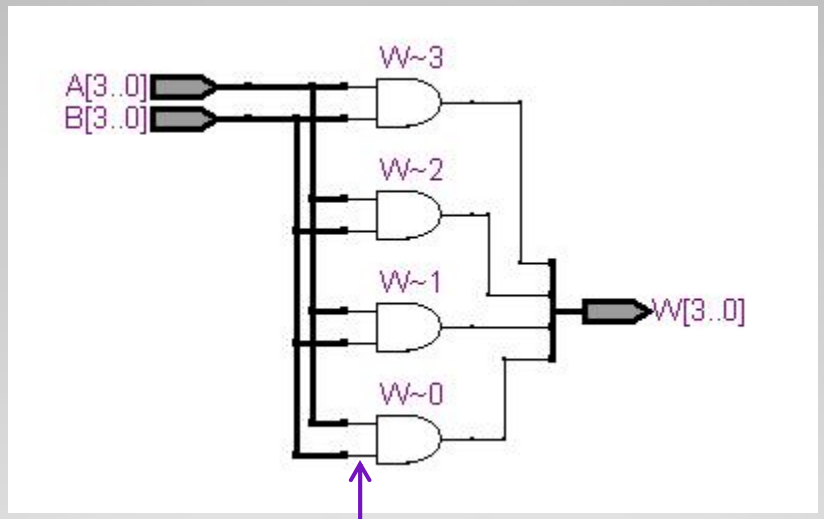
Vectors
Operators
Constants

Bit-wise Operators
(Between Two Vectors)

- AND &
- OR |
- XOR ^
- XNOR ^~ or ~^
- If vectors are different sizes, the result becomes the larger of the two vectors

```
module AND4by4( A, B, W );
  input [3:0] A;
  input [3:0] B;
  output [3:0] W;
  assign W = A & B;
endmodule
```

Bitwise AND Example

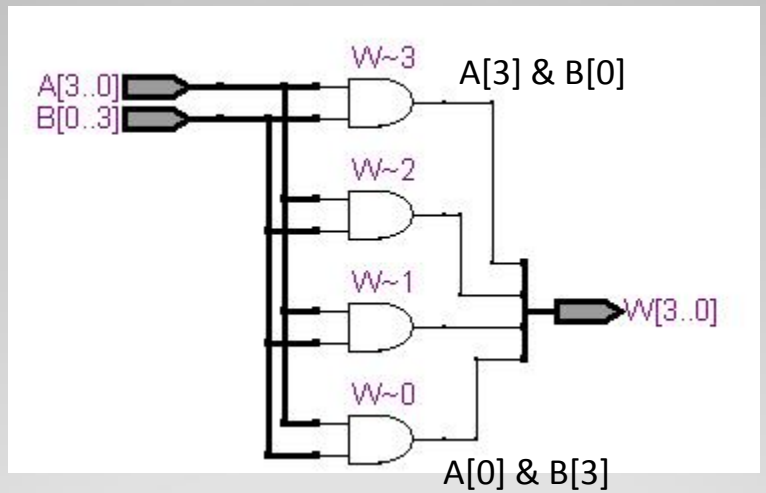


In RTL View put your cursor here to display the variable name

AND4by4

```
module AND4by4( A, B, W );
      input [3:0] A;
      input [0:3] B;  // big endian
      output [3:0] W;
      assign W = A & B;
endmodule
```

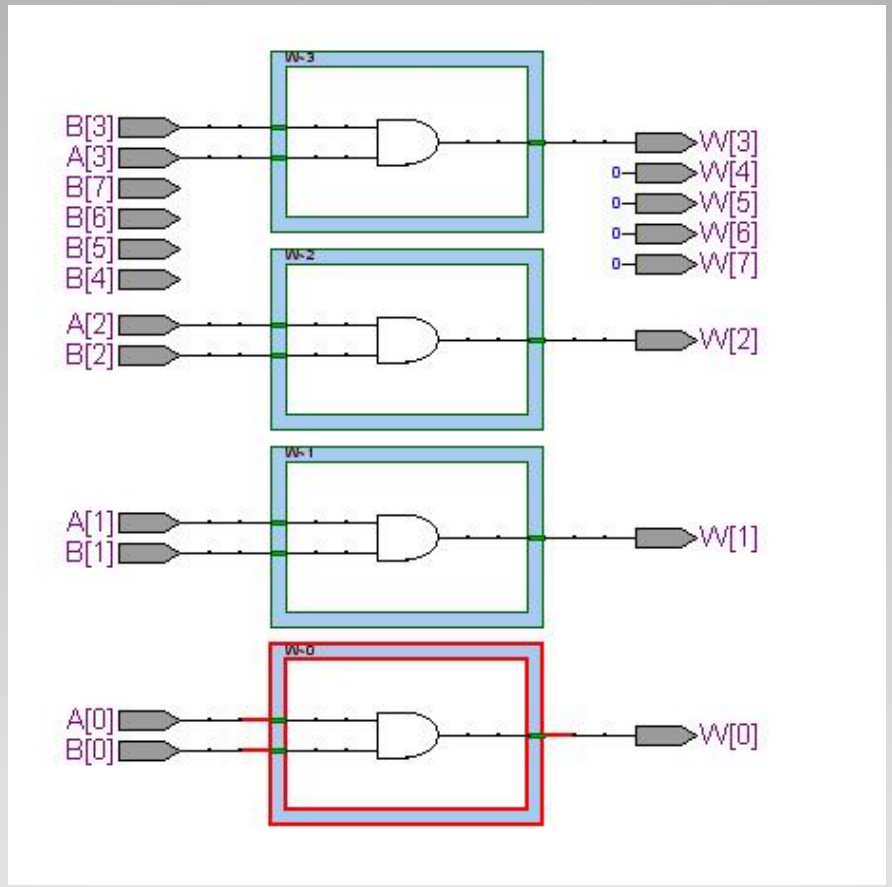
A Slight Twist



AND4by4, Reversed B

```
module OpTest( A, B, W );
  input [3:0] A;
  input [7:0] B;
  output [7:0] W;

  assign W = A & B;

endmodule
```

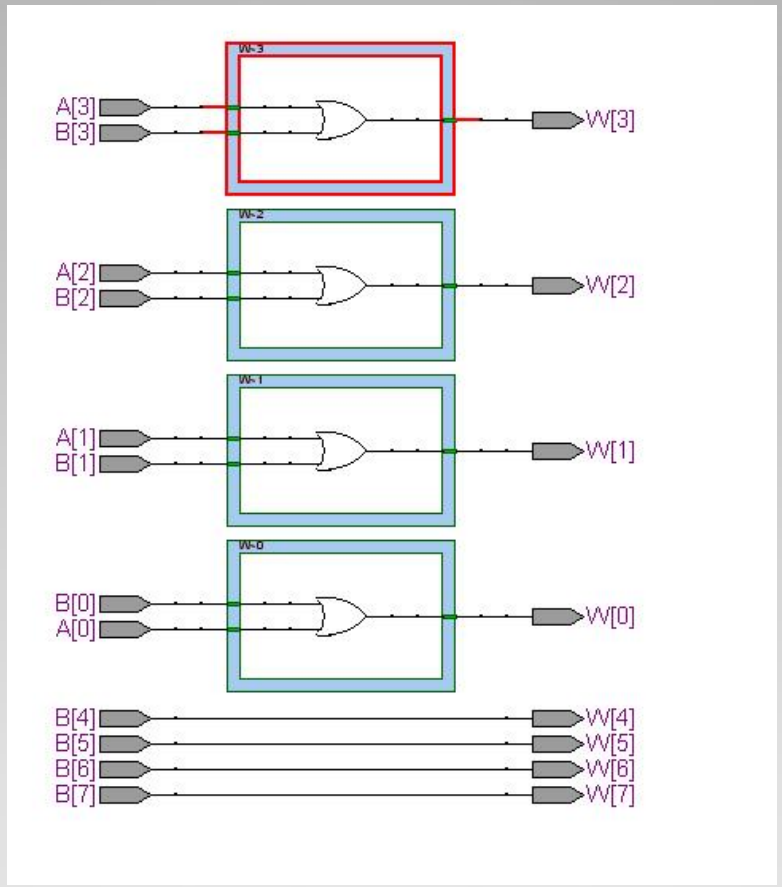
Different Size Vectors



AND-ing Different Size Vectors

```
module OpTest( A, B, W );
  input [3:0] A;
  input [7:0] B;
  output [7:0] W;

  assign W = A | B;

endmodule
```

Another Example



OR-ing Different Size Vectors

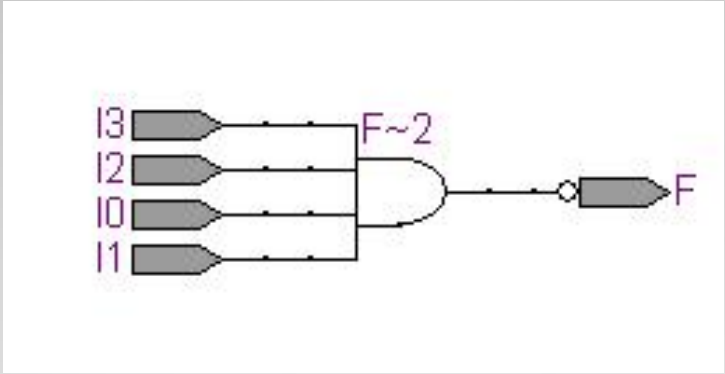
Reduction Operators

(One Vector -> One Bit)

- AND &
- NAND ~&
- OR |
- NOR ~|
- XOR ^
- XNOR ^~ or ~^
- Performs operation on all bits
- One-bit result

```
module NAND4( I, W );
   input [3:0] I;
   output W = ~&I;
  // same as
  // assign W = ~(I[0] & I[1] & I[2] & I[3]);
endmodule
```

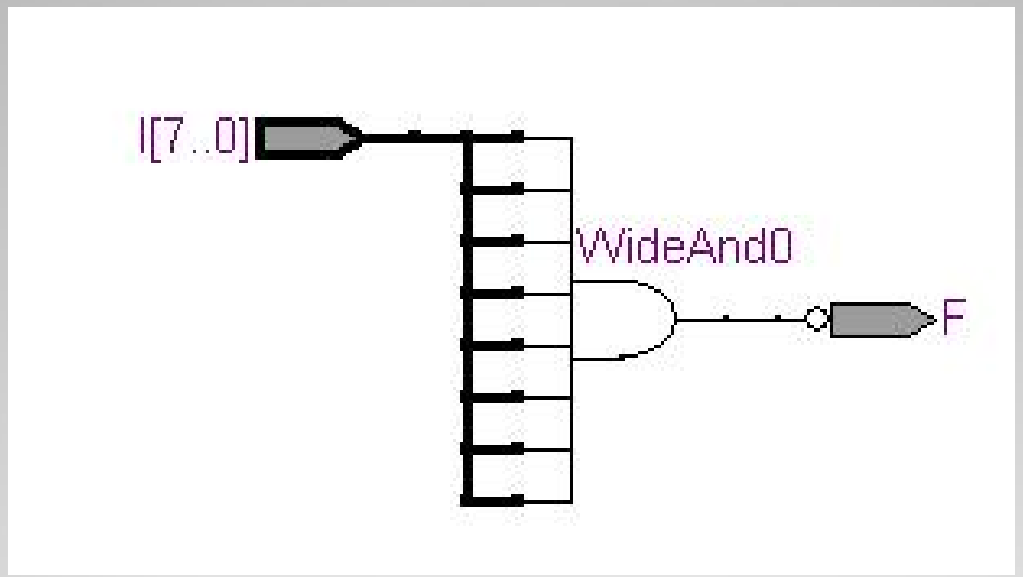
NAND4 Verilog



NAND4 Netlist Viewer

```
module NAND8( I, W );
    input [7:0] I;
    output W = ~&I;
endmodule
```

8-Wide NAND



8-Wide NAND

```
module NANDBunch( I0, J0, K0, Iout, Jout, Kout );
  input [3:0]  I0;
  input [7:0]  J0;
  input [11:0] K0;

  output Iout;
  output Jout;
  output Kout;

  // Requires 3 different NAND modules?
  NAND4 U0(  I0, Iout );
  NAND8 U1(  J0, Jout );
  NAND12 U2( K0, Kout );

endmodule
```

Do I have to write and include three different modules?

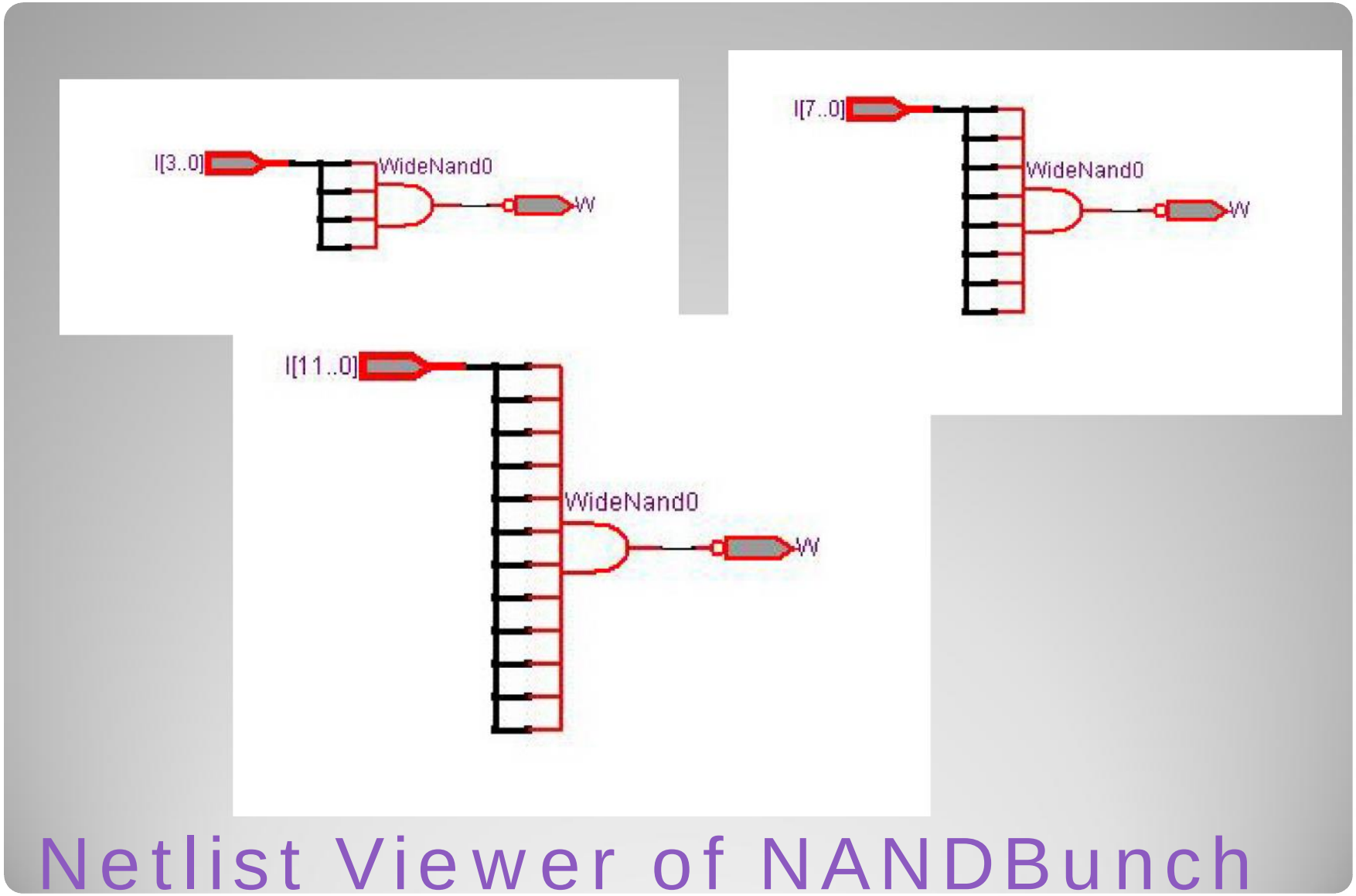
What If

```
module NANDn( I, W );
  parameter WIDTH = 8;  //default value
  input [WIDTH-1:0] I;
  output W = ~&I;
endmodule
```

```
module NANDBunch( I0, J0, K0, Iout, Jout, Kout );
  input [3:0]  I0;
  input [7:0]  J0;
  input [11:0] K0;

  output Iout;
  output Jout;
  output Kout;

  NANDn #(.WIDTH(4)) N0( I0, Iout );    // NAND4
  NANDn N1( J0, Jout );  // use default WIDTH: NAND8
  NANDn #(.WIDTH(12)) N12( K0, Kout );  // NAND12

endmodule
```

Modifies WIDTH at compile time

Verilog Has An Answer: Parameters



Netlist Viewer of NANDBunch

Vector Part Selection

- Select a range of bits from a vector
- Assume vector Areg[7:0], then
  - Areg[5:3] selects bits 5, 4, and 3
  - Areg[5-:4] selects bits 5, 4, 3, and 2
  - Areg[2+:4] selects bits 5, 4, 3, and 2

Vector Concatenation Operator

- Used to combine vectors and scalars into a larger vector
- Assume [3:0]A and [5:0]B, then
- {A, B} is 10 bits wide
- {A[3], B[2:1]} is 3 bits wide

Repetition Operator

- Assume
  - A has the value 1101
  - AA has the value 001001
  - AAA has the value 11
- Then { AA, {2{A}}, AAA } has the value 001001110111011

Verilog Constants

- Example: 4’b0000 is a four-bit binary number of all zeros.
- 4’b0 is also a four-bit binary number (all zeros).
- 4’B0101 is binary 0101 (= decimal 5)
- Other bases are decimal (d or D), octal (o or 0), hexadecimal (h or H).
- 12’hFA2 is 12-bit hex number (0xFA2)
- The size part (in front) is always in bits
- You can omit the size part for any base and/or the d (or D) if the number is decimal (but don’t!)

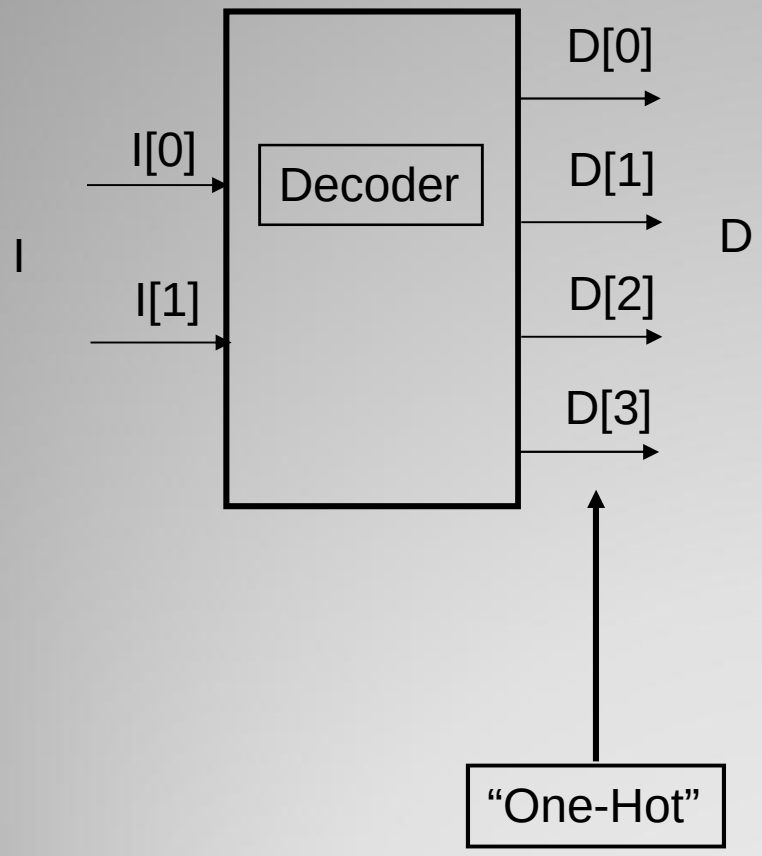
Using C-Type Constructs

- We can use many of the C or Java control constructs in Verilog (if/else, for-loops, etc.).
- We need to put them in an always @(...) procedure.
- Variables changed inside of a procedure have to be declared reg.
- Use either = (blocking) or <= (non-blocking) assignments.
- Examples: if, if/else, switch, for-loops
- Caution: these can generate a lot of hardware!

```
module Mux2to1D( A, B, S, W )
  input A, B, S;
  output reg W;
  // using a procedure
  always @( A, B, S ) begin
    if ( S ) W = B;
    else W = A;
  end
endmodule
```

Mux2to1D.v

Decoder Example




```
module TwoDecoder( I, D );
  input [1:0] I;
  output reg [3:0]D;

  always @( I ) begin
    if ( I == 2'd0 ) begin
      D = 4'b0001;
    end  // I == 0
    else if ( I == 2'd1 ) begin
      D = 4'b0010;
    end  // I == 1
    else if ( I == 2'd2 ) begin
      D = 4'b0100;
    end  // I == 2
    else begin
      D = 4'b1000;
    end  // I == 3 (default)
  end

endmodule
```

(To do: add Enable)

```
module NWideMux( I0, I1, S, W );
  parameter Width = 8;
  input[ Width-1:0] I0, I1;
  input S;
  output reg [Width-1:0] W;

  always @ * begin
    if ( S == 1'b1 ) begin
      W = I1;
    end
    else begin
      W = I0;
    end
  end // always
endmodule
```

N-Wide MUX

```
module MuxTest( J0, J1, S, X );
  input [3:0] J0, J1;
  input S;
  output [3:0] X;

  // change the MUX Width to 4:
  NWideMux #(.Width(4)) M1( J0, J1, S, X );
endmodule
```

Using the N-Wide MUX