Introduction to Verilog

Modules
Input/Output Ports
Basic Gates
Shorthand (Operators)

Verilog Module Specifications

```
module <module-name>( portlist );
   Declarations
   ...
   Functional specification of module
   ...
endmodule
```

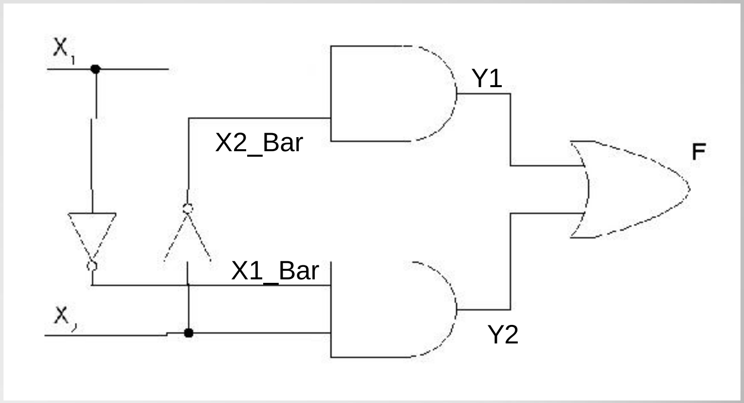
```
module light( x1, x2, f );
    input x1, x2;
    output f;
    // Note: ~ means NOT
    assign f = ( x1 & ~x2) | ( ~x1 & x2 );
endmodule
```

Behavioral Verilog

Verilog Code for Module light

```
module Light( X1, X2, F );
    input X1, X2;
    output F;

    wire X1_Bar, X2_Bar, Y1, Y2;

    // use built-in primitive gates (structural Verilog)
    // note: the order of these statements is not important
   or( F, Y1, Y2 );
   not( X1_Bar, X1 ),
      ( X2_Bar, X2 );
   and( Y1, X1, X2_Bar ),
      ( Y2, X1_Bar, X2 );

endmodule
```

Structural Verilog

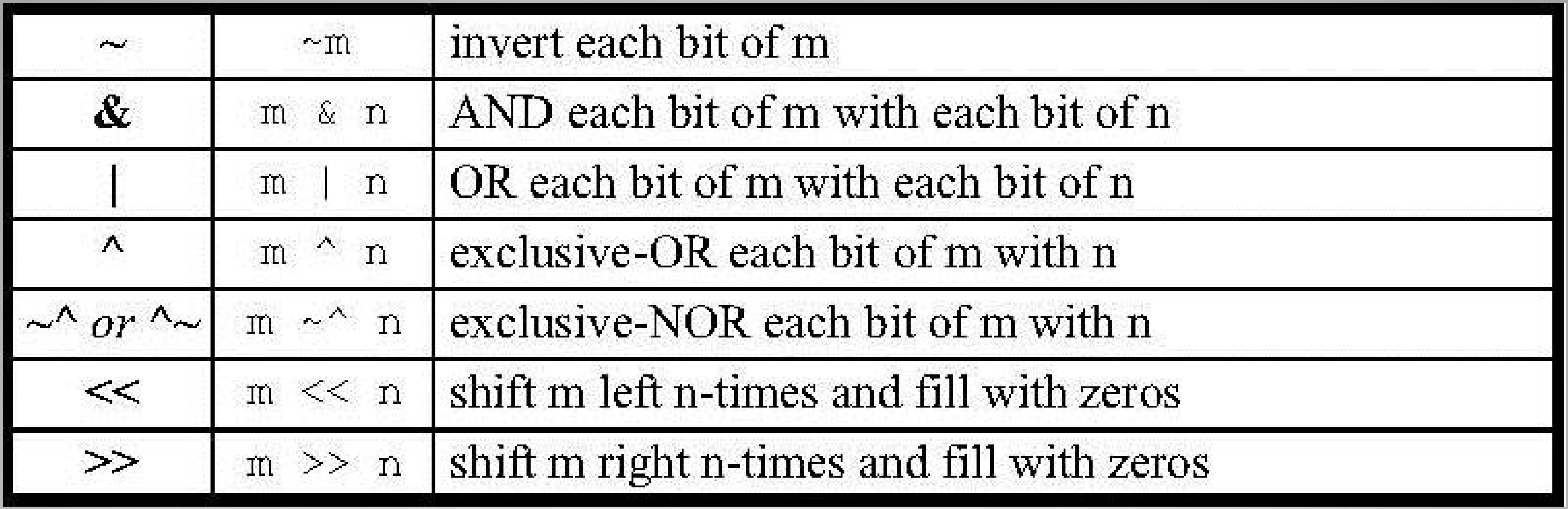


Light Circuit Using Primitive Gates

Name	Usage	Name	Usage
and	and(f,a,b,…)	not	not(f,a)
nand	nand(f,a,b,…)	buf	buf(f,a)
or	or(f,a,b,…)	notif0	notif0(f,a,e) †
nor	nor(f,a,b,…)	notif1	notif1(f,a,e) †
xor	xor(f,a,b,…)	bufif0	bufif0(f,a,e) †
xnor	xnor(f,a,b,…)	bufif1	bufif1(f,a,e) †

† 'e' means enable; outputs not enabled are high-z

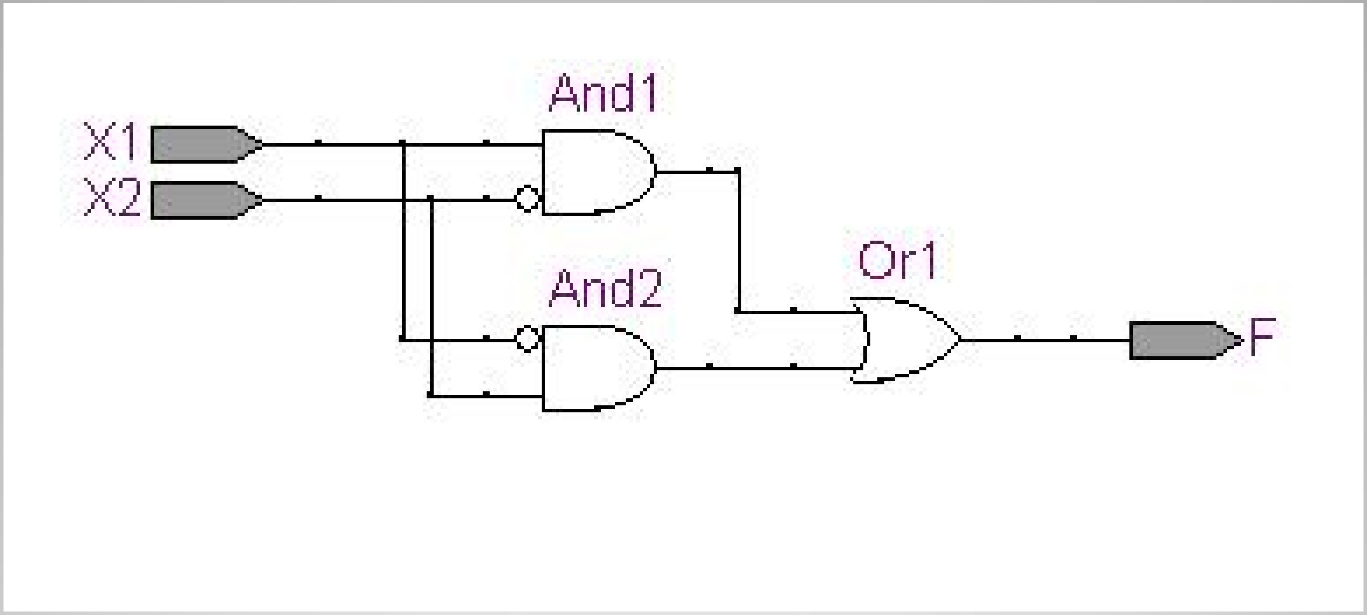
Verilog Primitive Gates (Subset)



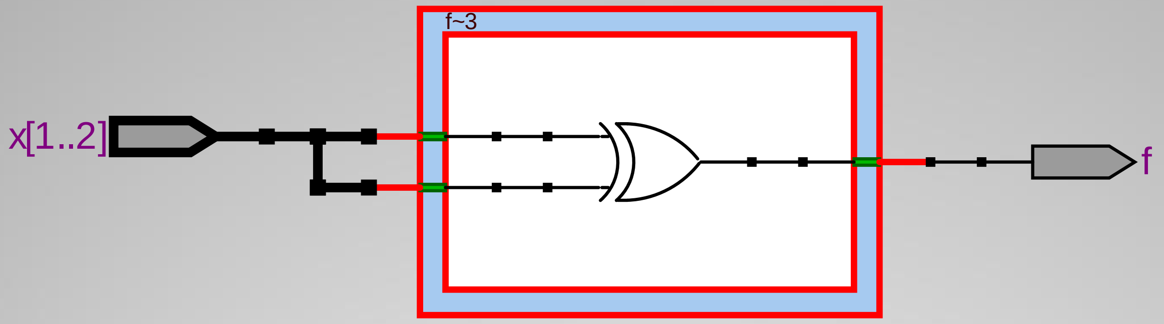
~	~m	invert each bit of m
&	m & n	AND each bit of m with each bit of n
|	m | n	OR each bit of m with each bit of n
^	m ^ n	exclusive-OR each bit of m with n
~^ *or* ^~	m ~^ n	exclusive-NOR each bit of m with n
<<	m << n	shift m left n-times and fill with zeros
>>	m >> n	shift m right n-times and fill with zeros

Note: m and n can be vectors

Bitwise Verilog Operators



Quartus II RTL Viewer



Quartus Technology Map Viewer: What Actually Gets Loaded in the FPGA

Comments and White Space in Verilog

- Comments:

```
// this is a comment
/* this is a comment – can span multiple
  lines */
```

- White space is ignored
  - Spaces
  - Tabs
  - new lines

- Verilog IS case sensitive: foo and Foo and fOO are all different.
- Any letter or digit may be used, as well as _ and $
- An identifier must not begin with a digit.
- An identifier must not be a Verilog keyword (refer to the Verilog HDL Quick Reference Guide section on Reserved Keywords – which are all lower case).
- Legal identifiers : f, x1, X1, x_y
- Illegal identifiers : 1x, +y, 258j
- We will want our identifiers to start with capital letters! This is to distinguish them from keywords (at a glance).

Identifier Names

Each signal in any of our circuits can take on one of four possible values:

- 0 = logic value 0 (false)
- 1 = logic value 1 (true)
- z = tri-state (high impedance)
- x = unknown value

Signal Values

Vectors

We’ll talk about this in more depth later,
but variables in Verilog can be scalars (one bit) or vectors.

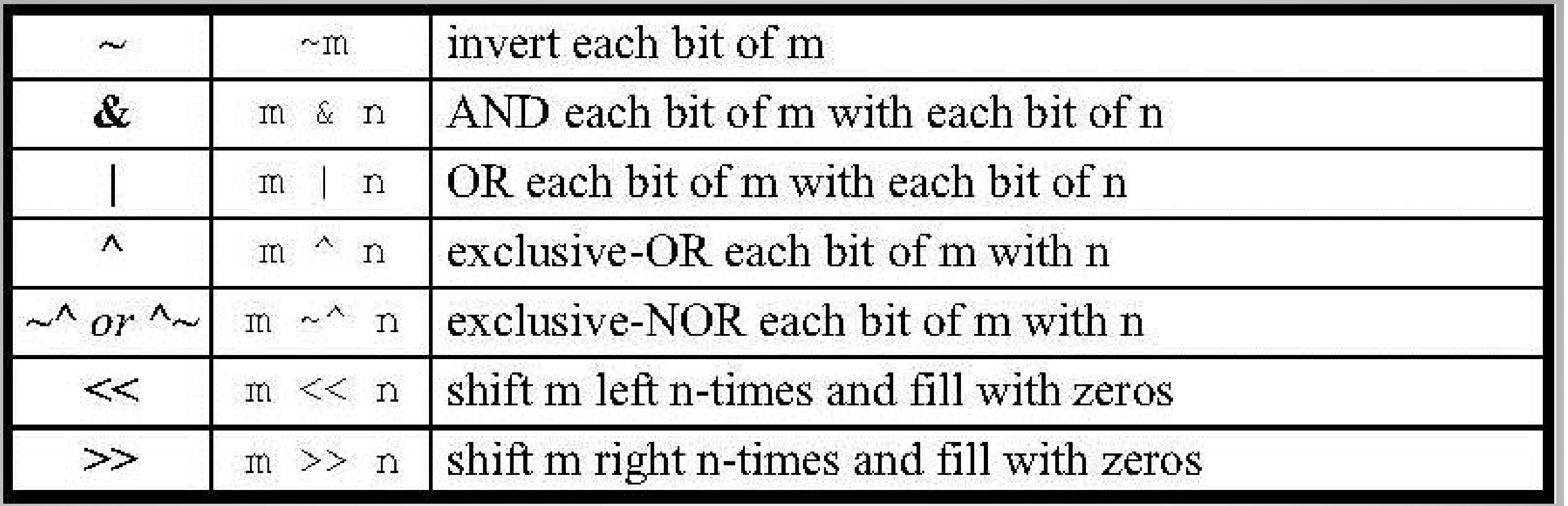
Scalar: $\boxed{1}$ or $\boxed{0}$ or $\boxed{z}$ or $\boxed{x}$ Example: `wire S1;`

Vector: $\boxed{1}\boxed{1}\boxed{0}\boxed{1}$ (four-bit vector) Example: `wire[3:0] V1;`

Vectors

- A vector data type defines a collection of bits.
- The range specification defines the numbering and order of the bits [msb:lsb].
- [3:0] A$\rightarrow$ Little endian convention
- [0:3] A$\rightarrow$ Big endian convention
- [1:4] A$\rightarrow$ legal, but unconventional
- Use [3:0] A unless there's a good reason to do otherwise.
- Use A[2] (for example) to select individual bits.
- Use A[2:1] to select bits 2 and 1.

Bitwise Verilog Operators

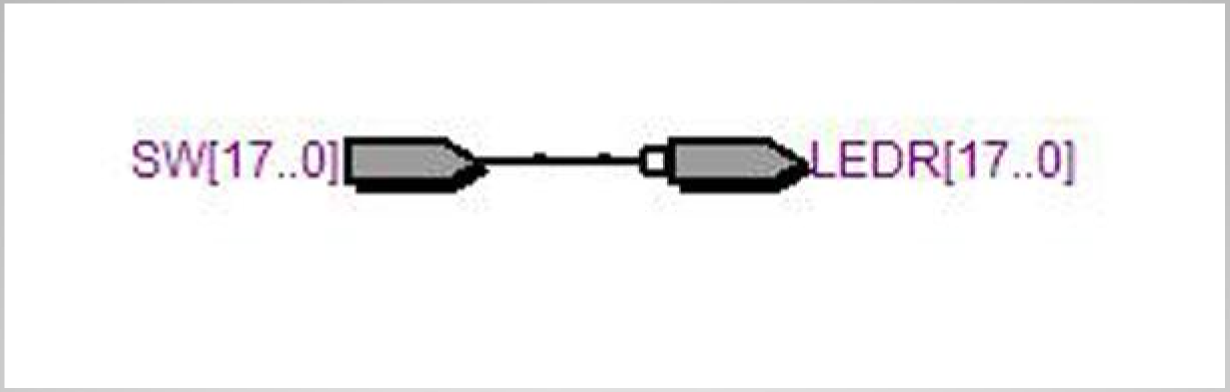


~	~m	invert each bit of m
&	m & n	AND each bit of m with each bit of n
|	m | n	OR each bit of m with each bit of n
^	m ^ n	exclusive-OR each bit of m with n
~^ *or* ^~	m ~^ n	exclusive-NOR each bit of m with n
<<	m << n	shift m left n-times and fill with zeros
>>	m >> n	shift m right n-times and fill with zeros

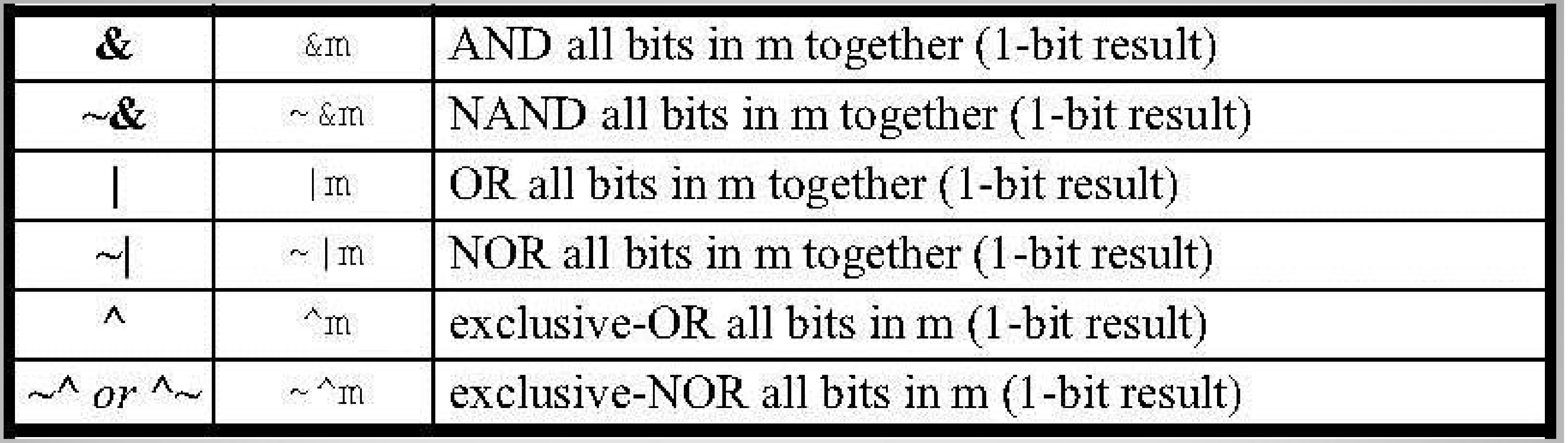
Note: m and n can be vectors

```
// TCES 330, Spring 2011
// R. Gutmann
// This module illustrates a simple vector operation

module SimpleVector( SW, LEDR );
  input [17:0] SW;      // vector input
  output [17:0] LEDR;   // vector output

  wire [17:0] Temp;     // temporary variable

  assign Temp = ~SW;

  assign LEDR = Temp;

  // Note: have been assign LEDR = ~SW;

endmodule
```

Simple Vector Example



RTL View of SimpleVector



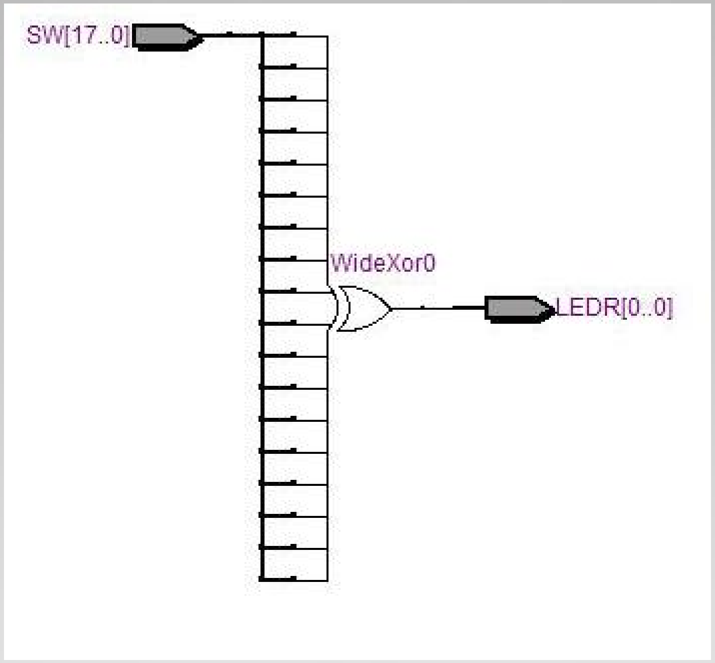
&	&m	AND all bits in m together (1-bit result)
~&	~ &m	NAND all bits in m together (1-bit result)
|	|m	OR all bits in m together (1-bit result)
~|	~ |m	NOR all bits in m together (1-bit result)
^	^m	exclusive-OR all bits in m (1-bit result)
~^ *or* **^~**	~^m	exclusive-NOR all bits in m (1-bit result)

Note: m is a vector

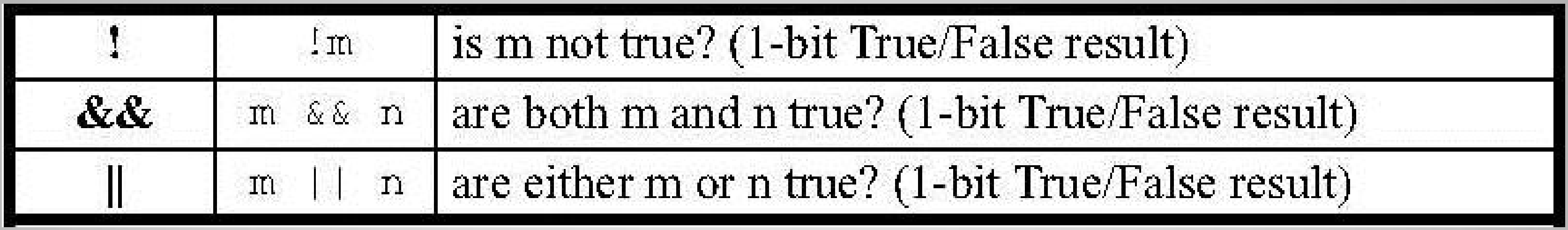
Verilog Reduction Operators

```
// TCES 330, Spring 2011
// R. Gutmann
// This module illustrates a simple vector reduction operation
//
module SimpleVector( SW, LEDR );
  input [17:0] SW;   // vector input

  output [0:0] LEDR; // scalar output

  assign LEDR[0] = ^SW;

endmodule
```

Vector Reduction Example

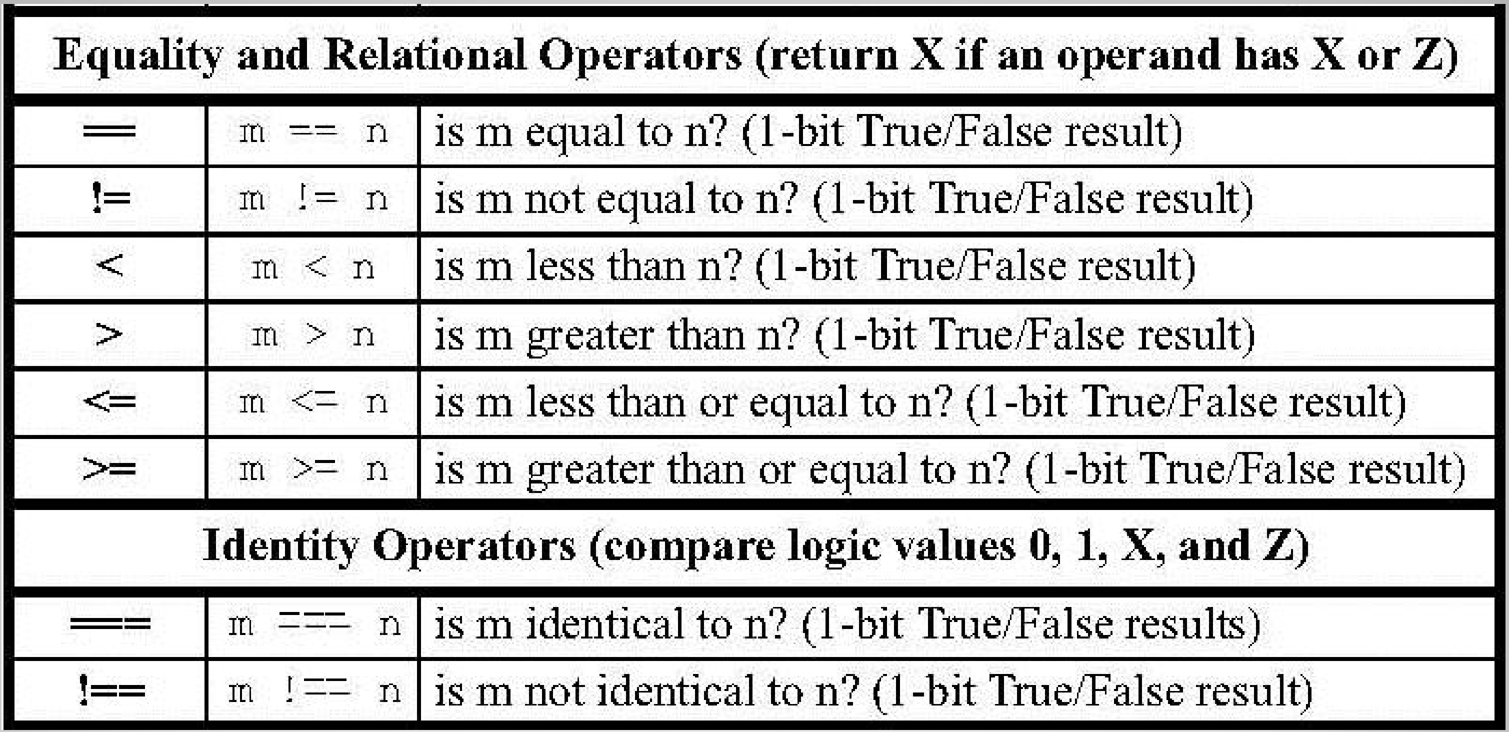


RTL View of VectorReduction



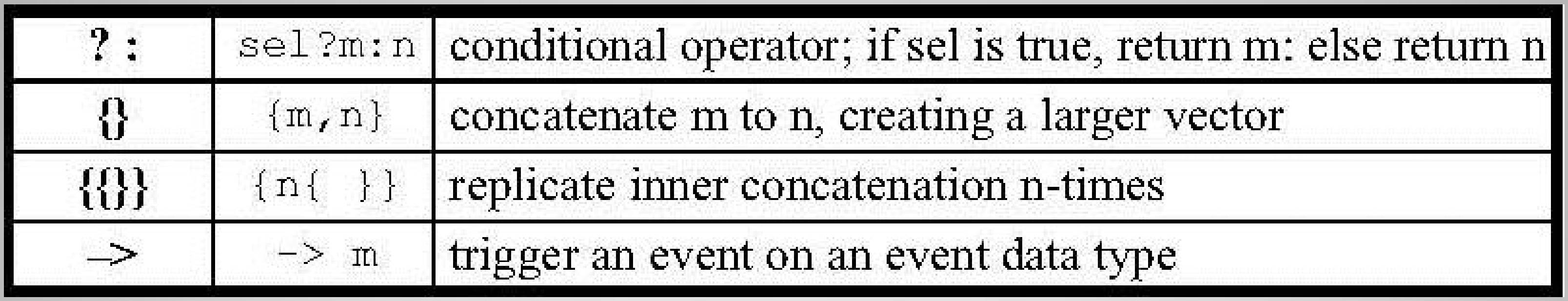
!	`!m`	is m not true? (1-bit True/False result)
&&	`m && n`	are both m and n true? (1-bit True/False result)
||	`m || n`	are either m or n true? (1-bit True/False result)

Verilog Logical Operators



Equality and Relational Operators (return X if an operand has X or Z)		
==	`m == n`	is m equal to n? (1-bit True/False result)
!=	`m != n`	is m not equal to n? (1-bit True/False result)
<	`m < n`	is m less than n? (1-bit True/False result)
>	`m > n`	is m greater than n? (1-bit True/False result)
<=	`m <= n`	is m less than or equal to n? (1-bit True/False result)
>=	`m >= n`	is m greater than or equal to n? (1-bit True/False result)
Identity Operators (compare logic values 0, 1, X, and Z)		
===	`m === n`	is m identical to n? (1-bit True/False results)
!==	`m !== n`	is m not identical to n? (1-bit True/False result)

Verilog Relational Operators

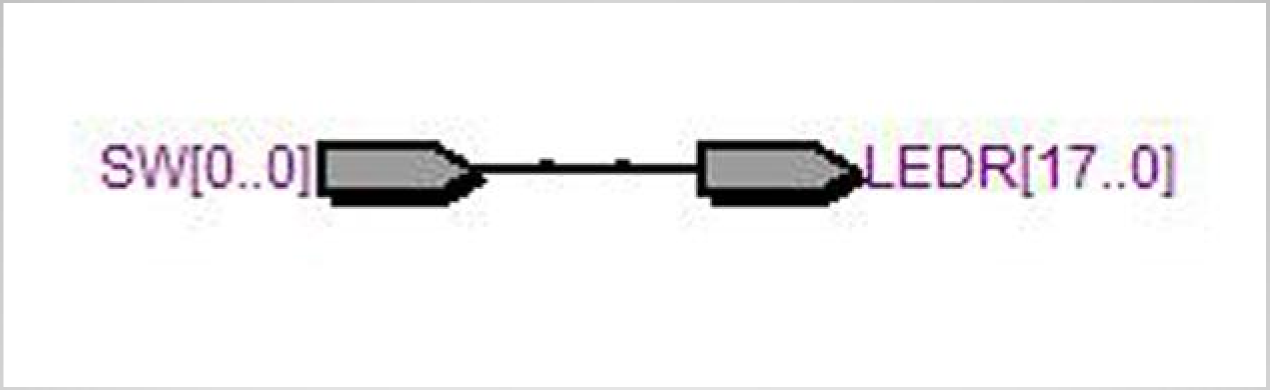


? :	sel?m:n	conditional operator; if sel is true, return m: else return n
{}	{m,n}	concatenate m to n, creating a larger vector
{{}}	{n{ }}	replicate inner concatenation n-times
->	-> m	trigger an event on an event data type

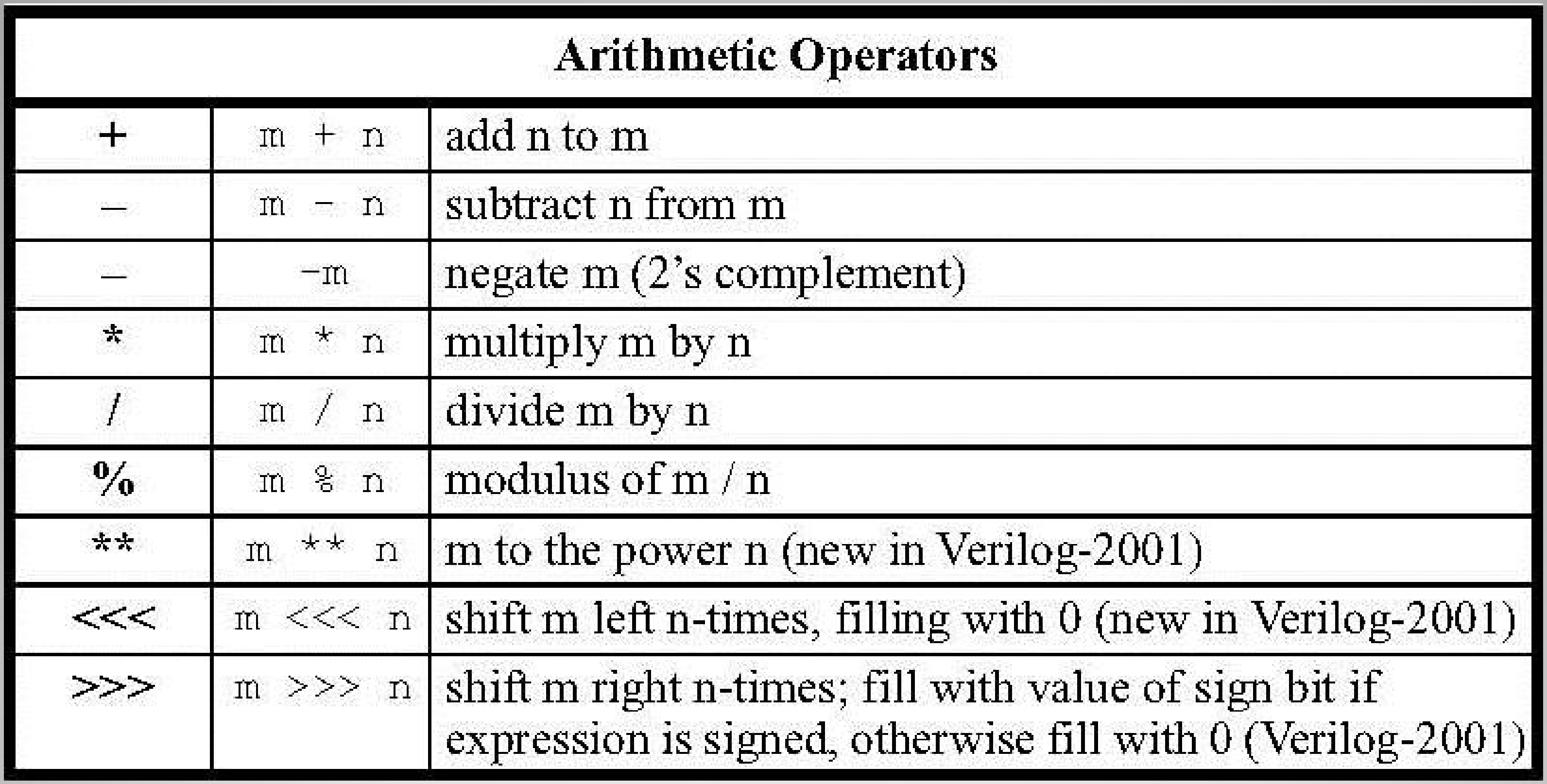
Miscellaneous Verilog Operators

```
// TCES 330, Spring 2011
// R. Gutmann
// This module illustrates the replication operator
//
module MiscOperator( SW, LEDR );
  input [0:0] SW;     // scalar input

  output [17:0] LEDR; // vector output

  assign LEDR = {18{SW[0]}};  // replicate SW[0] 18 times

endmodule
```

Miscellaneous Operators Example

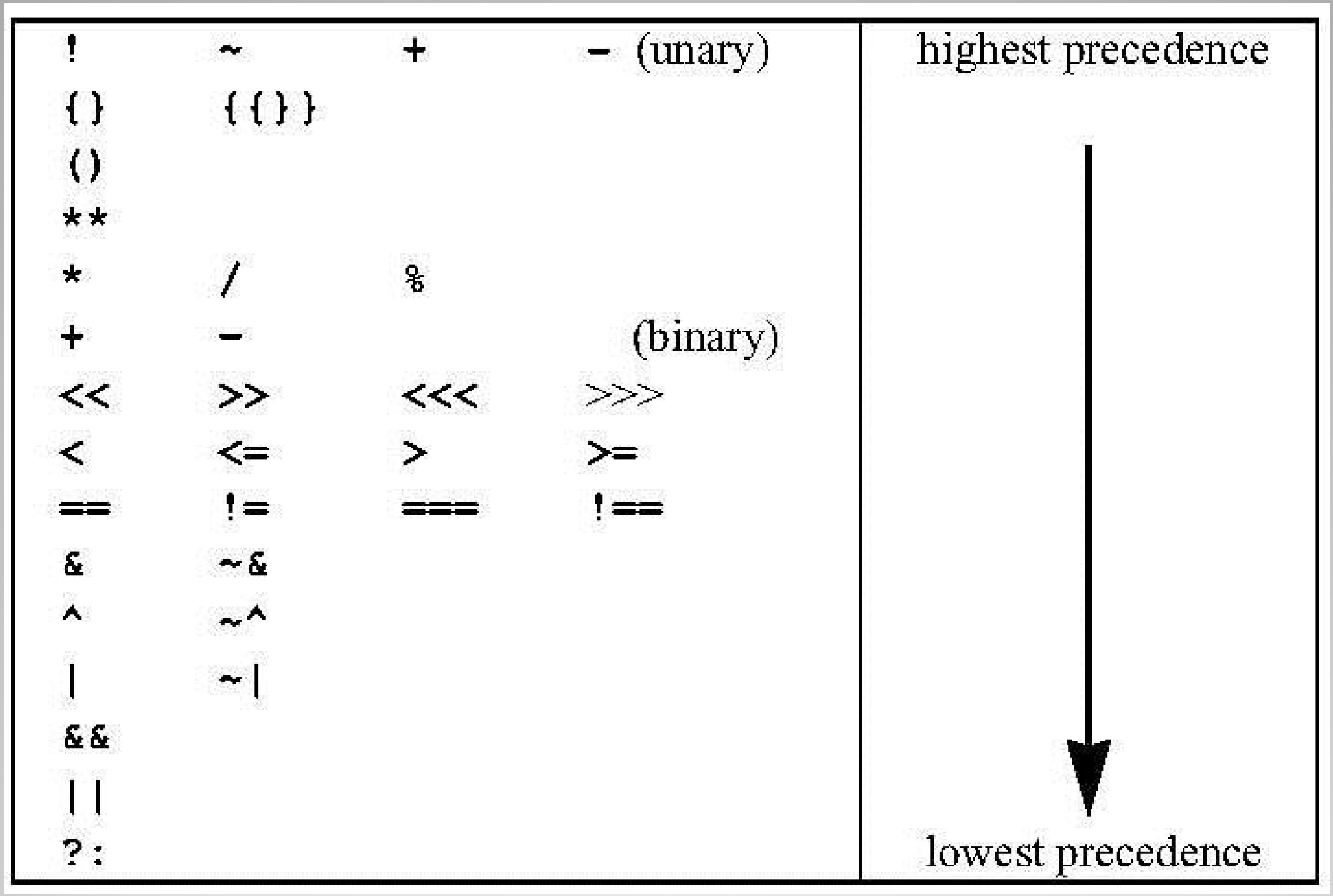


RTL View of MiscOperators



Arithmetic Operators		
+	m + n	add n to m
–	m - n	subtract n from m
–	-m	negate m (2's complement)
*	m * n	multiply m by n
/	m / n	divide m by n
%	m % n	modulus of m / n
**	m ** n	m to the power n (new in Verilog-2001)
<<<	m <<< n	shift m left n-times, filling with 0 (new in Verilog-2001)
>>>	m >>> n	shift m right n-times; fill with value of sign bit if expression is signed, otherwise fill with 0 (Verilog-2001)

Verilog Arithmetic Operators

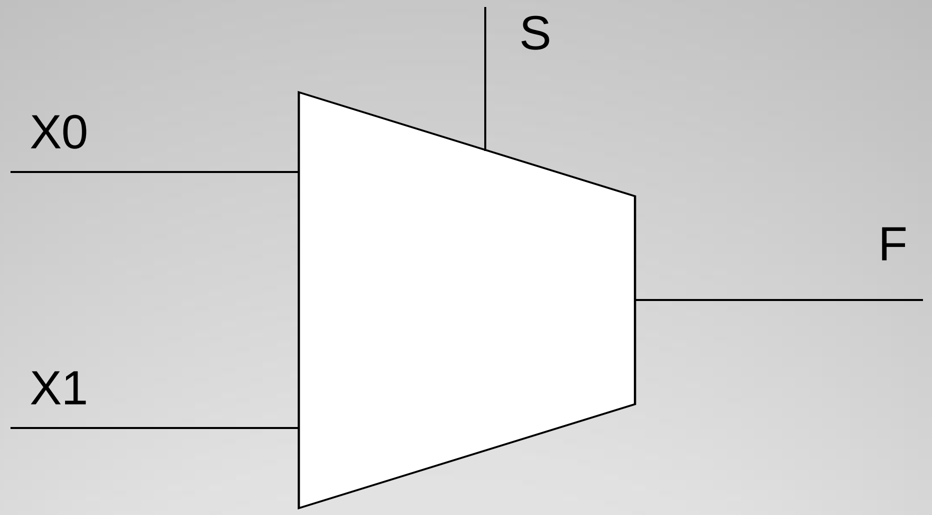


!	~	+	-	(unary)	highest precedence
{}	{{}}				
()					
**					
*	/	%			
+	-			(binary)	
<<	>>	<<<	>>>		
<	<=	>	>=		
==	!=	===	!==		
&	~&				
^	~^				
|	~|				
&&					
||					
?:					lowest precedence

Operator Precedence

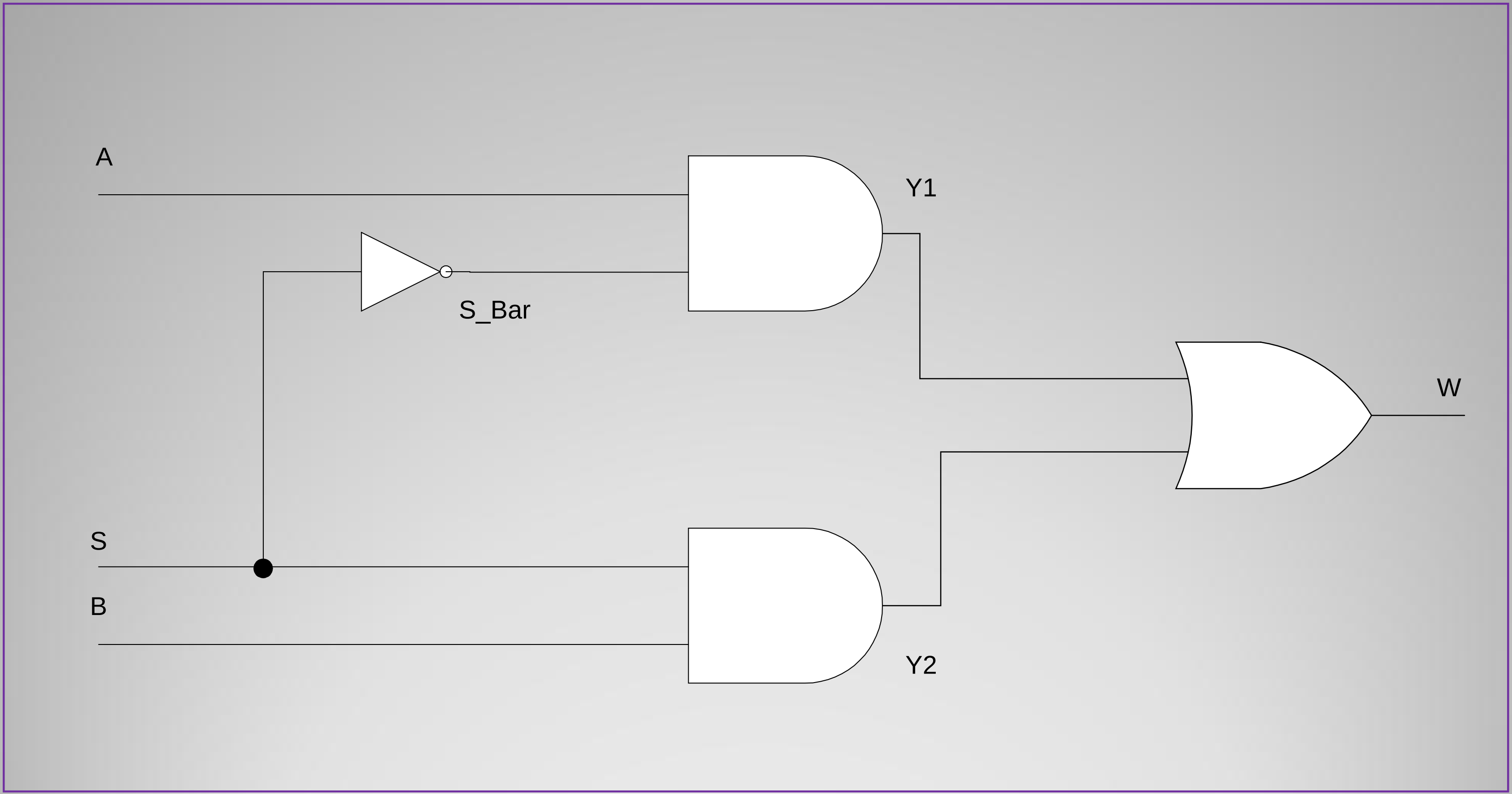
Pretty much as you would expect, but use parenthesizes to make it clear!

2-to-1 Mux

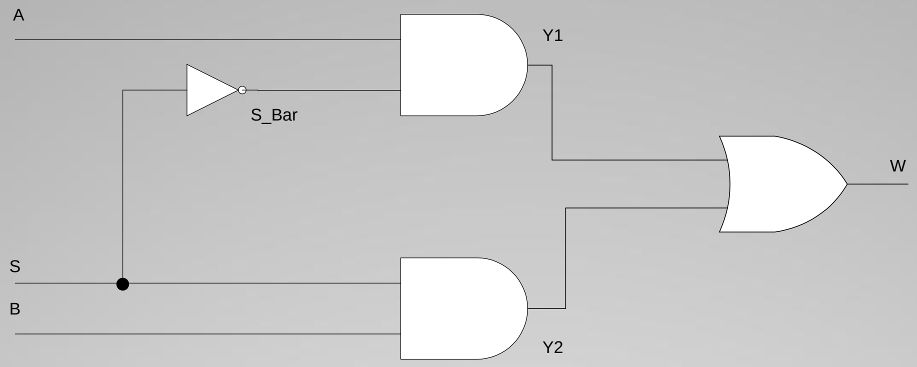


X0 and X1 are the inputs
S is a ‘select’ bit
F is the output
F = X0 if S==0; F = X1 if S==1

2-to-1 Multiplexer (Mux)

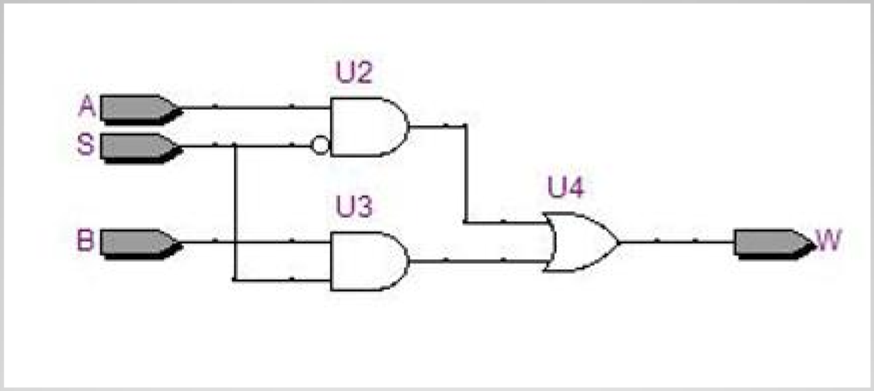


2-to-1 Multiplexer

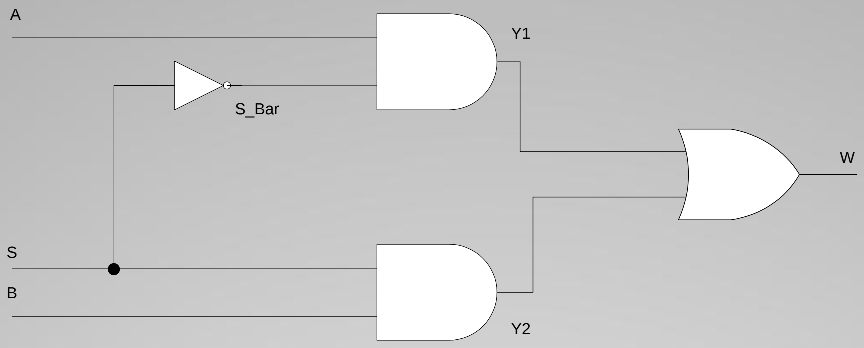


```
 module Mux2to1A( A, B, S, W );
      input A, B, S;
      output W;

      wire S_Bar, Y1, Y2;

      // use built-in primitive gates
      not U1( S_Bar, S );
      and U2( Y1, A, S_Bar );
      and U3( Y2, B, S );
      or  U4( W, Y1, Y2 );
endmodule
```

Mux2to1A.v



RTL View of Mux2to1

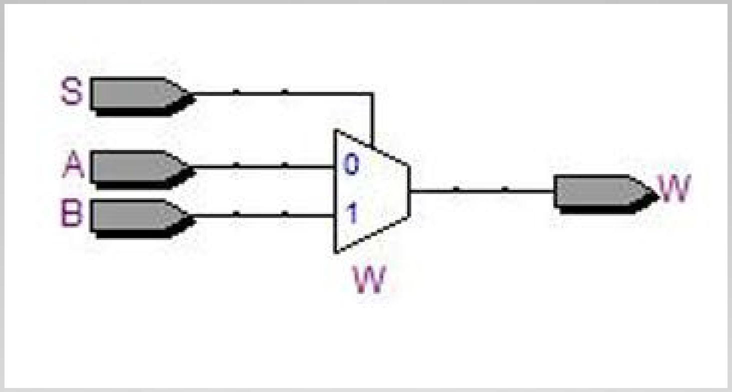


```
 module Mux2to1B( A, B, S, W );
     input A, B, S;
     output W;
     // using assign and boolean operators
     assign W = ( A & ~S) | ( B & S );
endmodule
```

Mux2to1B.v

```
 module Mux2to1C( A, B, S, W );
     input A, B, S;
     output W;

     // using assign and conditional operator
     assign W = S ? B : A;
endmodule
```

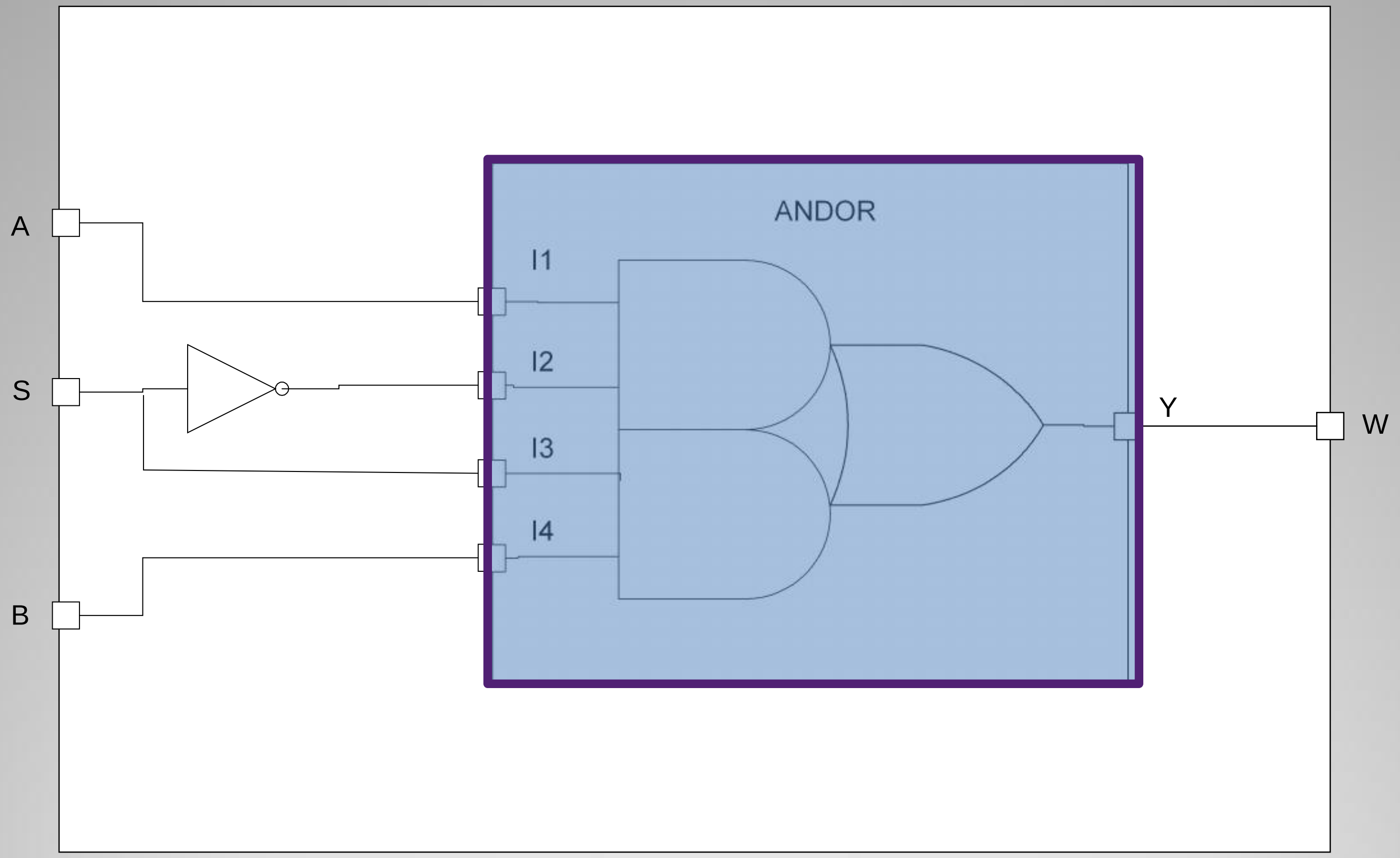
Mux2to1C.v



RTL View of Mux2to1C

```
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



2-to-1 Mux Using ANDOR Module

```
 module ANDOR( input I1, I2, I3, I4, output Y );
    assign Y = ( I1 & I2 ) | ( I3 & I4 );
endmodule
```

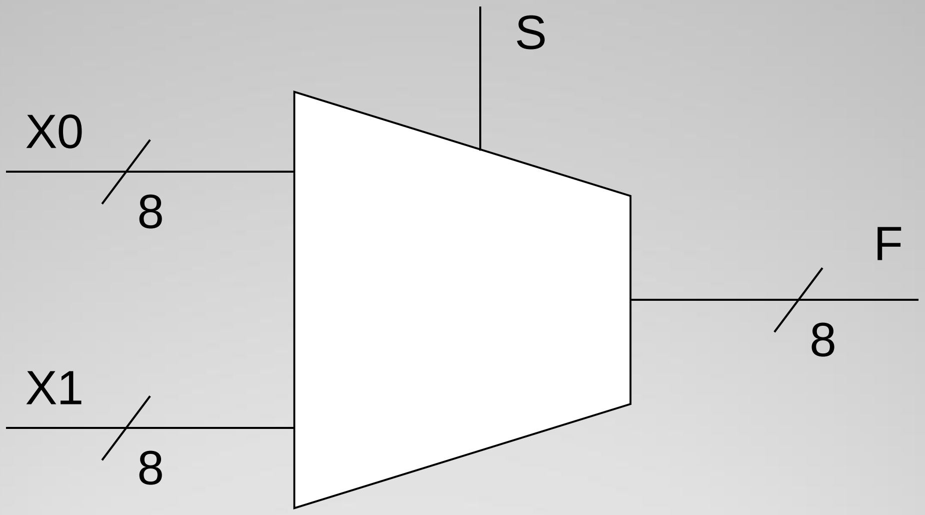
Note the “input” and “output” designation in the module’s port list. This is an optional way of specifying the inputs and outputs.

ANDOR.v

```
 module Mux2to1E( A, B, S, W );
     input A, B, S;
     output W;

     wire S_Bar = ~S;

     // make an instance of ANDOR:
     ANDOR U1( A, S_Bar, S, B, W );
endmodule
```

Mux2to1E.v

Vector inputs and outputs: 8-Wide 2-to-1 Mux



X0 and X1 are the inputs
S is a ‘select’ bit
F is the output
F = X0 if S==0; F = X1 if S==1

2-to-1 Multiplexer (Mux)

$A = X3(X1 + X2')$
$B = X1(X2' + X3')$
$C = X2(X1 + X3')$

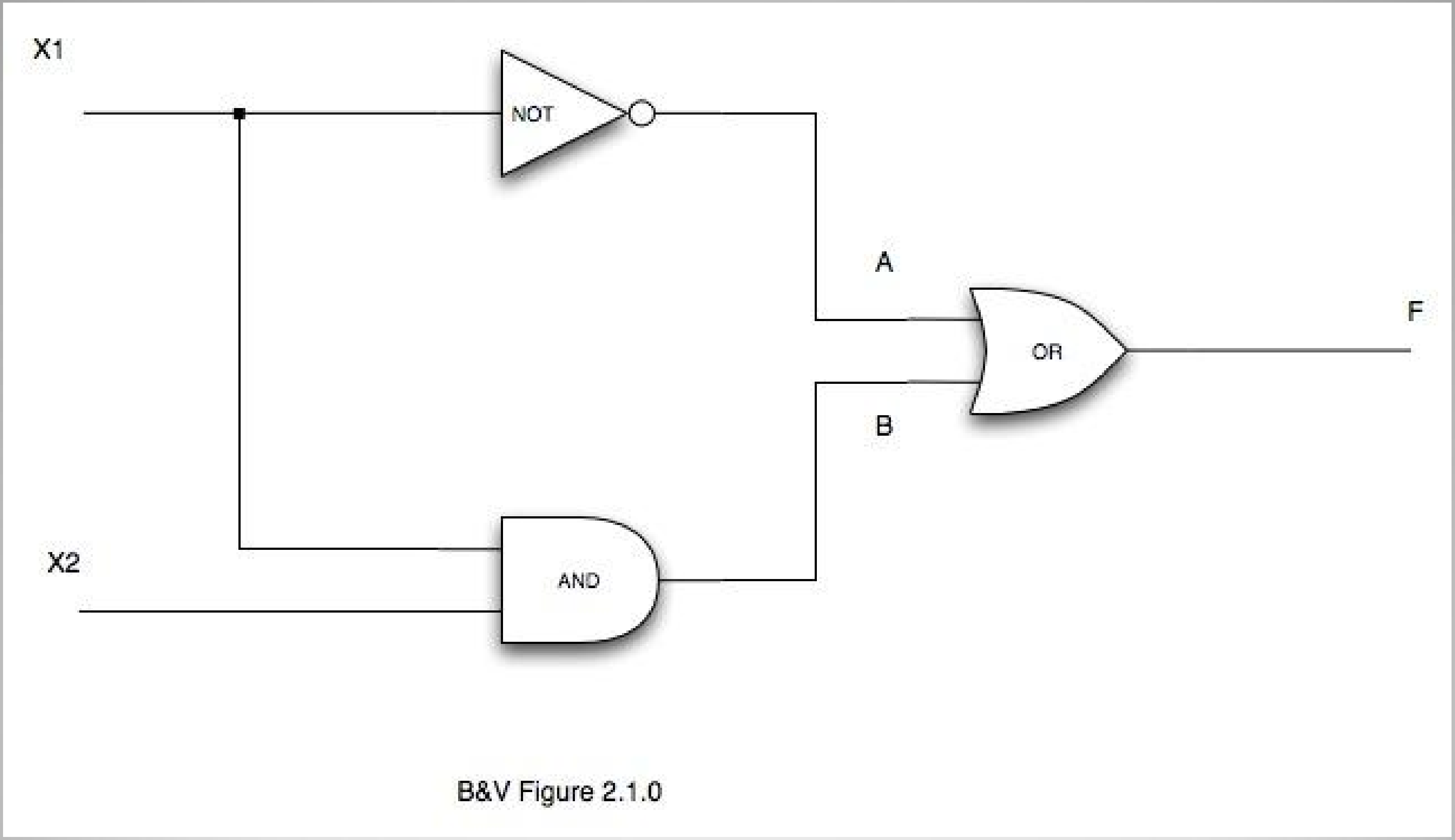
$F = AB + AC + BC$

```
// compute A, B, C
 assign A = X3 & ( X1 | ~X2 );
 assign B = X1 & (~X2 | ~X3 );
 assign C = X2 & ( X1 | ~X3 );

 // compute F
 assign F = A&B | A&C | B&C;
```

after some algebra…
$F = X1(X3 + X2)$

Quartus II Project

Example 2.10
(B&V Page 67)



B&V Figure 2.1.0

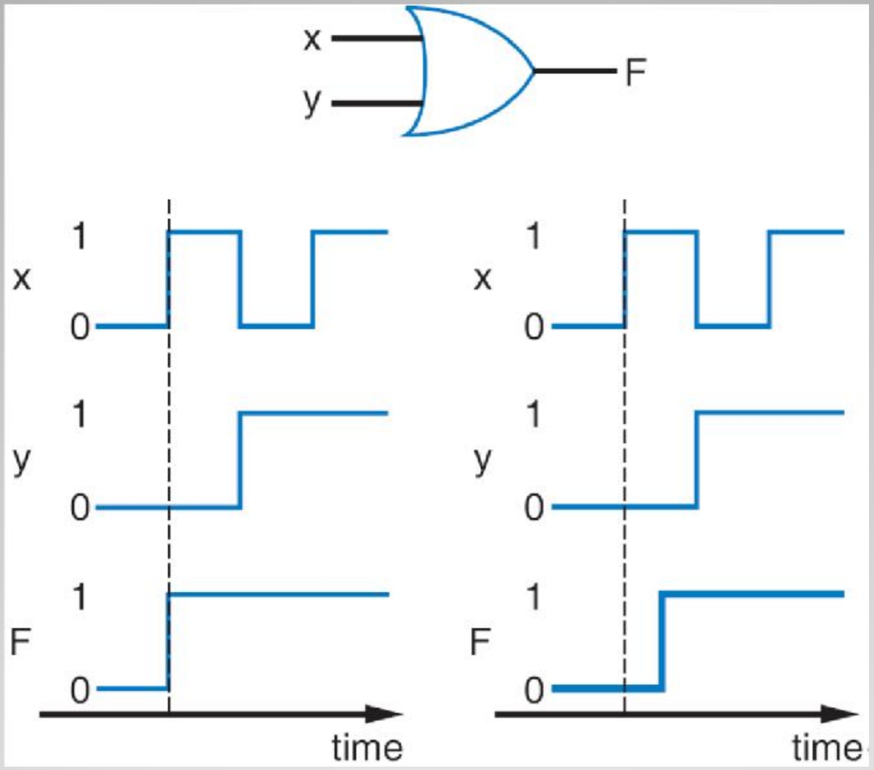
B&V Figure 2.10a

- The Lab0 tutorial tells you how to import pin assignments. Look on page 22 at the sentence that begins “You can import a pin assignment…”
- The file you want to import is DE2_pin_assignments.csv. This file is up on the Moodle site. And you can copy it to whatever folder you are using for Lab 1, Part 1.
- The Lab 1 Part 1 write-up also talks about using DE2_pin_assignments.csv
- After you import this file and recompile, you will get lots of warnings about pins you’ve assigned but not used. We will deal with this issue later.

Pin Assignments

Additional Considerations

Non-Ideal Gate Behavior -- Delay



- Real gates have some delay
  - Outputs don’t change immediately after inputs change