Dataflow SystemVerilog

- Program functions as high level expressions
  - Program logical and arithmetic expressions that describe what functions to implement
  - Wide selection of available operators (logical, arithmetic, shift, etc.)

- Synthesize a circuit from the expressions
  - Convert (synthesize) arithmetic expressions to logic equations
  - Convert (synthesize) logic equations to gates
  - Let the synthesizer minimize logic

- Significantly reduces time to design logic circuits
  - Focus on what the high level logic and arithmetic that you need, and not how it is implemented.
  - Design productivity is much higher than designing with gates
  - Less risk of errors and less debugging

© 2020 BYU

Dataflow "assign" statement

- assign <wire/port> = <dataflow expression>;
- SystemVerilog module can contain a mix of structural SystemVerilog statements *and* dataflow "assign" statements
- "assign" statement operates concurrently with the other statements
  - Demand driven semantics: the ordering of the assign statements *does not matter*

© 2020 BYU

Dataflow "assign" statement

- Wide variety of operators that can be used within an "assign" statement
  - Constant Assignment
  - Bitwise logic
  - Logic reduction
  - Arithmetic
  - Shift
  - Concatenate
  - Relational
  - Logical
  - Conditional operations

© 2020 BYU

Operator Type	Operator Symbol	Operation Performed	# of Operands	Comments
Arithmetic	*, /, +, -	As expected	2	* and / take LOTS of hardware
	%	Modulo	2	
Logical	!	Logic NOT	1	As in C
	&&	Logic AND	2	As in C
	||	Logic OR	2	As in C
Bitwise	~	Bitwise NOT	1	As in C
	&	Bitwise AND	2	As in C
	|	Bitwise OR	2	As in C
	^	Bitwise XOR	2	As in C
	~^	Bitwise XNOR	2	
Relational	<, >, <=, >=	As expected	2	As in C
Equality	==, !=	As expected	2	As in C
Reduction	&	Red. AND	1	Multi-bit input
	~&	Red. NAND	1	Multi-bit input
	|	Red. OR	1	Multi-bit input
	~|	Red. NOR	1	Multi-bit input
	^	Red. XOR	1	Multi-bit input
	~^	Red. XNOR	1	Multi-bit input
Shift	<<	Left shift	2	Fill with 0's
	>>	Right shift	2	Fill with 0's
Concat	{ }	Concatenate	Any number	
Replicate	{ { } }	Replicate	Any number	
Cond	?:	As expected	3	As in C

Higher precedence
Unary -, unary +, !, ~
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&, ~&
^, ~^
|, ~|
&&
||
?:
Lower precedence

© 2020 BYU

Constant Assignment

- Assign a constant value to a wire

- Example:

```
logic [7:0] Q;

assign Q = 8'h3F;
```


© 2020 BYU

SystemVerilog Bitwise Logical Operators

- Perform Boolean operation on single- or multi-bit signals

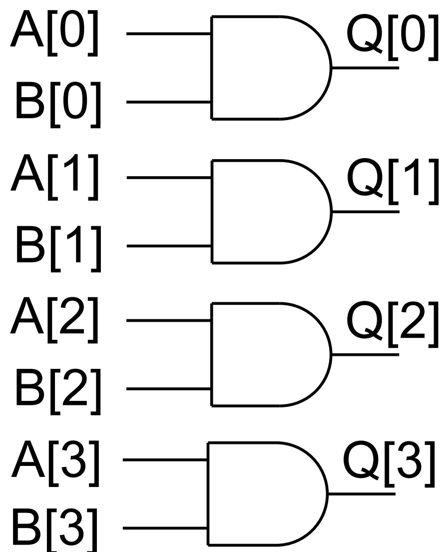
Operator	Function	# Operands
~	Bitwise NOT	1
&	Bitwise AND	2
|	Bitwise OR	2
^	Bitwise XOR	2
~^	Bitwise XNOR	2

© 2020 BYU

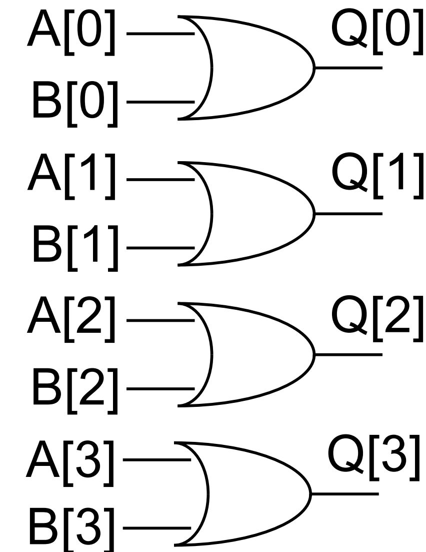
Multi-Bit Logical Operators

```
logic [3:0] A,B;
assign A = 4'b0101;
assign B = 4'b1101;
```

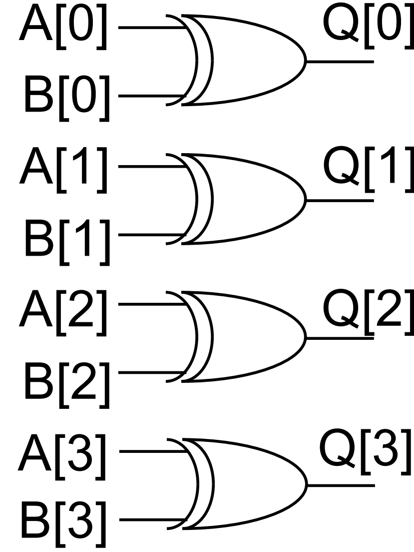
`assign Q = A&B;`



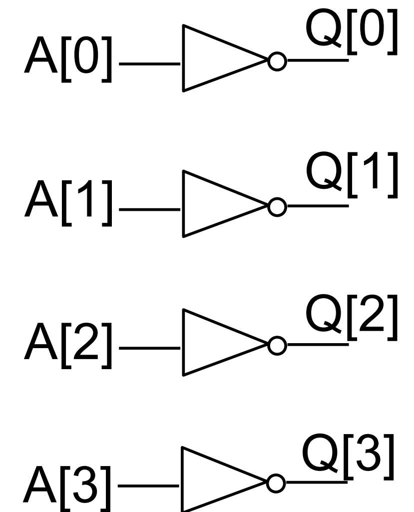
`assign Q = A|B;`



`assign Q = A^B;`



`assign Q = ~A;`

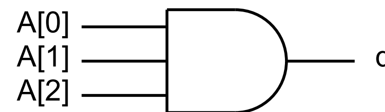


© 2020 BYU

Reduction Operators

- Single (unary) operand operators that operate on multi-bit signals
  - Reduces a multi-bit wire to a single-bit result

Operator	Function	# Operands
&	Red. AND	1
~&	Red. NAND	1
|	Red. OR	1
~|	Red. NOR	1
^	Red. XOR	1
~^	Red. XNOR	1



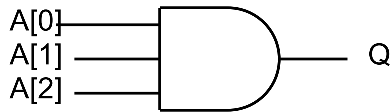
© 2020 BYU

Reduction Operators

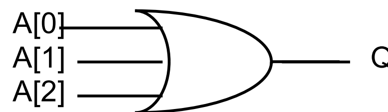
Operator	Function	# Operands
&	Red. AND	1
~&	Red. NAND	1
|	Red. OR	1
~|	Red. NOR	1
^	Red. XOR	1
~^	Red. XNOR	1

Q is one bit

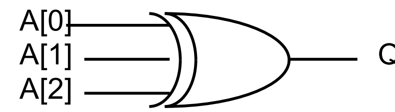
`assign Q = &A;`



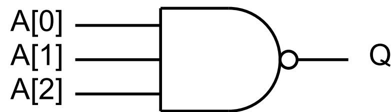
`assign Q = |A;`



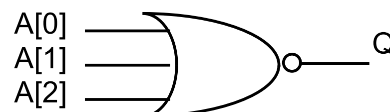
`assign Q = ^A;`



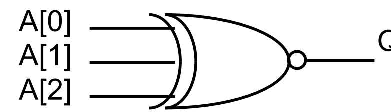
`assign Q = ~&A;`



`assign Q = ~|A;`



`assign Q = ~^A;`



© 2020 BYU

Arithmetic Operators

- Performs 2's complement arithmetic
  - Generates a multi-bit result

Operator	Function	# Operands	
-	Negate	1	
+	Addition	2	
-	Subtraction	2	
*	Multiplication	2	
/	Division	2	
%	Modulo	2	remainder

```
logic [7:0] A,B,O;
assign O = A+B;
assign O = A%3;
```

Remainder of A divided by 3

Generates all logic necessary to implement 2's complement arithmetic. Avoid using '*' and '/' (may not synthesize)

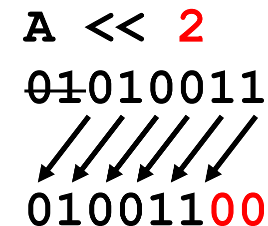
© 2020 BYU

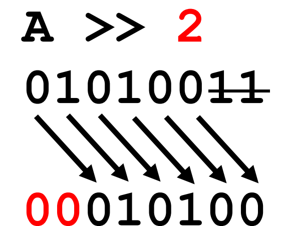
Shift Operators

- Shift signals

Operator	Function	# Operands
<<	Left Shift	2
>>	Right Shift	2

  - Fills vacated bits with zeros






```
logic [7:0] A,F,G;
assign A = 8'b01010011;

assign F = A<<2; → assign F = {A[5:0], 2'b00};
assign G = A>>2; → assign G = {2'b00, A[7:2]};
```


© 2020 BYU

Shift Operators

Shift to the left : multiply by 2 per digit shifted
0010 0001 = 33
0100 0010 = 66 = 33x2
1000 0100 = 132 == 66x2

Shift to the right : divide by 2 per digit shifted
0010 0001 = 33
0001 0000 = 16 = 33/2
0000 1000 = 8 == 33/4

© 2020 BYU

Concatenate Operators

Operator	Function
{x, y}	Concatenate x and y
N {x}	Replicate ‘x’ N times

© 2020 BYU

Concatenation and Replication Operators

`{x, y}`	*Concatenate x to y*
`N{x}`	*Replicate x N-times*

```
logic[3:0] x, y;
logic[7:0] z, q, w, t;
logic[31:0] m, n;

assign x = 4'b1100;
assign y = 4'b0101;
assign z = {x, x};   // z is 8'b11001100
assign q = {x, y};   // q is 8'b11000101
assign w = {4'b1101, y};   // w is 8'b11010101
assign t = {2{x}};   // same as {x, x}
assign m = {{4{x}}, {2{q}}};
           // m is 32'b11001100110011001100010111000101
```


© 2020 BYU

Relational Operators

- Comparing two operands against each other
  - Returns a logical '1' or '0' (representing T and F)

Operator	Function	# Operands
==	Equal	2
!=	Not Equal	2
<	Less Than	2
<=	Less Than or equal	2
>	Greater Than	2
>=	Greater Than or equal	2

```
always_comb
  If (w <= 4'b1001)
    A = 0;
  else
    A = 1;
```

```
always_comb
  If (w == 4'd9)
    A = 0;
  else
    A = 1;
```

```
always_comb
  If (w == 9)
    A = 0;
  else
    A = 1;
```

```
always_comb
  If (X&Y != Z)
    A = 0;
  else
    A = 1;
```


© 2020 BYU

Logical Operators

- Operate on logical (rather than bit) values
  - Returns a logical '1' or '0' (representing T and F)

Operator	Function	# Operands
!	Not	1
&&	And	2
||	Or	2

```
always_comb
    If (w <= 4'b1001 && X&Y != Z)
        A = 0;
    else
        A = 1;
```

```
assign A = (w == 4'b1001 || X&Y != Z) ? 0 : 1;
```


© 2020 BYU

Conditional Operator

- Performs like a 2-to-1 multiplexor
  - "If-then-else" operator

Operator	Function	# Operands
? :	Conditional	3

<logical expression> ? <result if true> : <result if false>

```
assign o = A>B ? C : 4'hF;
```

```
O = C if A > B
O = 4'hF if A <= B
```

© 2020 BYU

Operator Precedence

- Similar to C

Higher precedence
Unary -, unary +, !, ~
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&, ~&
^, ~^
|, ~|
&&
||
?:
Lower precedence

© 2020 BYU

A Dataflow MUX - Version 1

```
module mux21(
      input wire logic sel, a, b,
      output logic q);

      assign q = (~sel & a) | (sel & b);
endmodule
```

Much simpler, less typing, familiar C-like syntax.
Synthesizer turns it into optimized gate-level design.

© 2020 BYU

A Dataflow MUX - Version 2

```
module mux21(
     input wire logic sel, a, b,
     output logic q);

     assign q = sel ? b : a;
endmodule
```

Even simpler, uses C-like ternary (?:) construct

© 2020 BYU

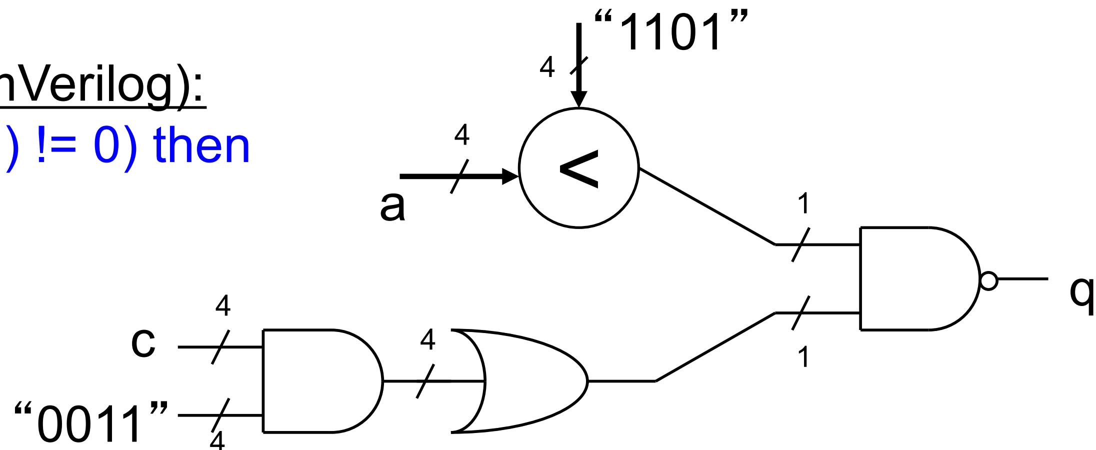
Bitwise vs. Logic Operators

- Similar to C

```
assign q = ((a<4'b1101) && ((c&4'b0011)!=0)) ? 1'b0:1'b1;
```

<u>Pseudo-code (not real SystemVerilog):</u>

```
if (a<4'b1101 && (c&4'b0011) != 0) then
  q <= '0';
else
  q <= '1';
```



Use &&, ||, ! for 1-bit quantities (results of comparisons)

Use &, |, ~ for bit-by-bit logical operations

© 2020 BYU

A Note on Matching Widths

- This is a valid 2:1 MUX statement:

```
logic a, b, sel, q;
assign q = (~sel & a) | (sel & b);
```

- But the following is not:

```
logic[3:0] a, b, q;
logic sel;
assign q = (~sel & a) | (sel & b);
```

Why?

© 2020 BYU

More On Matching Wire Widths

- This is an acceptable substitute:

```
logic[3:0] a, b, q;
logic sel;
assign q = ({4{~sel}}&a)|({4{sel}}&b);
```

- It turns the **`sel`** and **`~sel`** values into 4-bit versions for AND-ing and OR-ing

- A better version:

```
logic[3:0] a, b, q;
logic sel;
assign q = sel ? b: a;
```


© 2020 BYU

Design Example: A 2:4 Decoder

```
module decode24(
  output logic[3:0] q,
  input wire logic[1:0] a);

  assign q = (4'b0001) << a;
endmodule
```

Can you see how to make a 3:8 or 4:16 decoder in the same fashion?

a is a two-bit number from 0~3.

If a = 0, what is q ?
If a = 1, what is q ?
If a = 2, what is q ?
If a = 3, what is q ?

© 2020 BYU

Four Functions

```
module FourFunctions(
  input wire logic A,B,C,
  output logic O1,O2,O3,O4);

  assign O1 = (A&C)|(~A&B);         // AC+A’B
  assign O2 = (A|~C)&B&C;           // (A+C’)BC
  assign O3 = (A&~B)+C;             // AB’+C
  assign O4 = ~(A&B)+~(~C&~B);      // (AB)’+(B’C’)’

endmodule
```

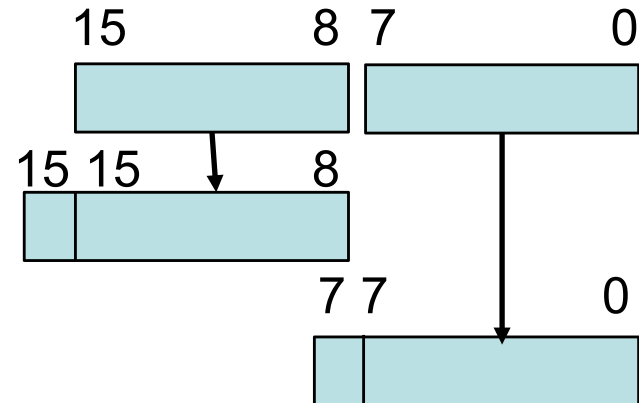
Lab 4 - Arithmetic

```
module arithmetic(
  input wire logic[15:0] sw,
  input wire logic btnl, btnr,
  output logic[8:0] led);

  logic[8:0] a,b;

  assign b = {sw[7],sw[7:0]};
  assign a = (btnl == 1’b1) ? 9’d0 : {sw[15],sw[15:8]});
  assign led = (btnr == 1’b1) ? a-b : a+b;

endmodule
```



sw	Input	16	Switches (sw[15:8] = A input, sw[7:0] = B)
btnl	Input	1	Left Button (Zero A operand)
btnr	Input	1	Right Button (negation of B operand)
led	Output	9	LED signals (result)

Lab 5 - Seven Segment Decoder

```
module seven_segment(
  input wire logic[3:0] data,
  output logic[6:0] segment);
  assign segment =
        (data == 0) ? 7'b1000000 :
        (data == 1) ? 7'b1111001 :
        (data == 2) ? 7'b0100100 :
        (data == 3) ? 7'b0110000 :
        (data == 4) ? 7'b0011001 :
        (data == 5) ? 7'b0010010 :
        (data == 6) ? 7'b0000010 :
        (data == 7) ? 7'b1111000 :
        (data == 8) ? 7'b0000000 :
        (data == 9) ? 7'b0010000 :
        (data == 10) ? 7'b0001000 :
        (data == 11) ? 7'b0000011 :
        (data == 12) ? 7'b1000110 :
        (data == 13) ? 7'b0100001 :
        (data == 14) ? 7'b0000110 :
        7'b0001110;
endmodule
```

© 2020 BYU

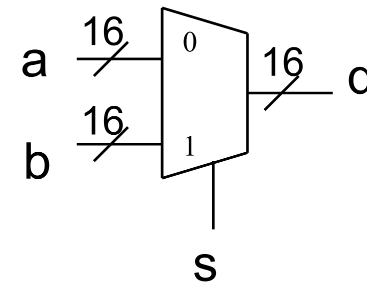
Multi-bit Design and Parameterization

© 2020 BYU

A Dataflow MUX - Multi-Bit

```
module mux21(
     input logic sel,
     input logic[15:0] a, b,
     output logic[15:0] q);

     assign q = sel ? b : a;
endmodule
```



Key Ideas:

- The predicate must evaluate to true or false (1 or 0)
- The parts getting assigned must be all same widths.

© 2020 BYU

A Dataflow MUX - Parameterized Width

```
module mux21n #(parameter WID = 16)(
      input wire logic sel,
      input wire logic[WID-1:0] a, b,
      output logic[WID-1:0] q);

      assign q = sel ? b : a;
endmodule
```

- By default, this is now a 16-bit wide MUX.
- When instantiating, the default value of 16 can be overridden:

```
mux21n M1(q, sel, a, b);          // Instance a 16-bit version
mux21n M0 #(4)(q, sel, a, b);     // Instance a 4-bit version
```

Does this work for a 1-bit MUX?

© 2020 BYU

4:1 MUX - Method 1

```
module mux41n #(parameter WID=16)(
        input wire logic[1:0] sel,
        input wire logic[WID-1:0] a, b, c, d,
        output logic[WID-1:0] q
        );

        logic[WID-1:0] tmp1, tmp2;

        mux21n M0 #(WID)(tmp1, sel[0], a, b);
        mux21n M1 #(WID)(tmp2, sel[0], c, d);
        mux21n M2 #(WID)(q, sel[1], tmp1, tmp2);
endmodule
```

If the mux21n cells are parameterizable for bit-width this works…
If not, it doesn't work…

© 2020 BYU

4:1 MUX - Method 2

```
module mux41 #(WID=16)(
        input logic[1:0] sel,
        input logic[WID-1:0] a, b, c, d,
        output logic[WID-1:0] q
        );

        assign q = (sel == 2'b00) ? a:
                   (sel == 1)     ? b:
                   (sel == 2'b10) ? c:
                  d;
endmodule
```

Cascaded ?: operators form an if-then-else structure

Note how **`sel`** can be compared to bit patterns (**`2'b00`**) or to decimal numbers (**`1`**)

© 2020 BYU