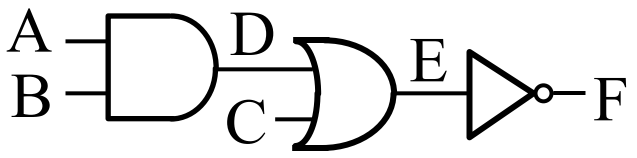
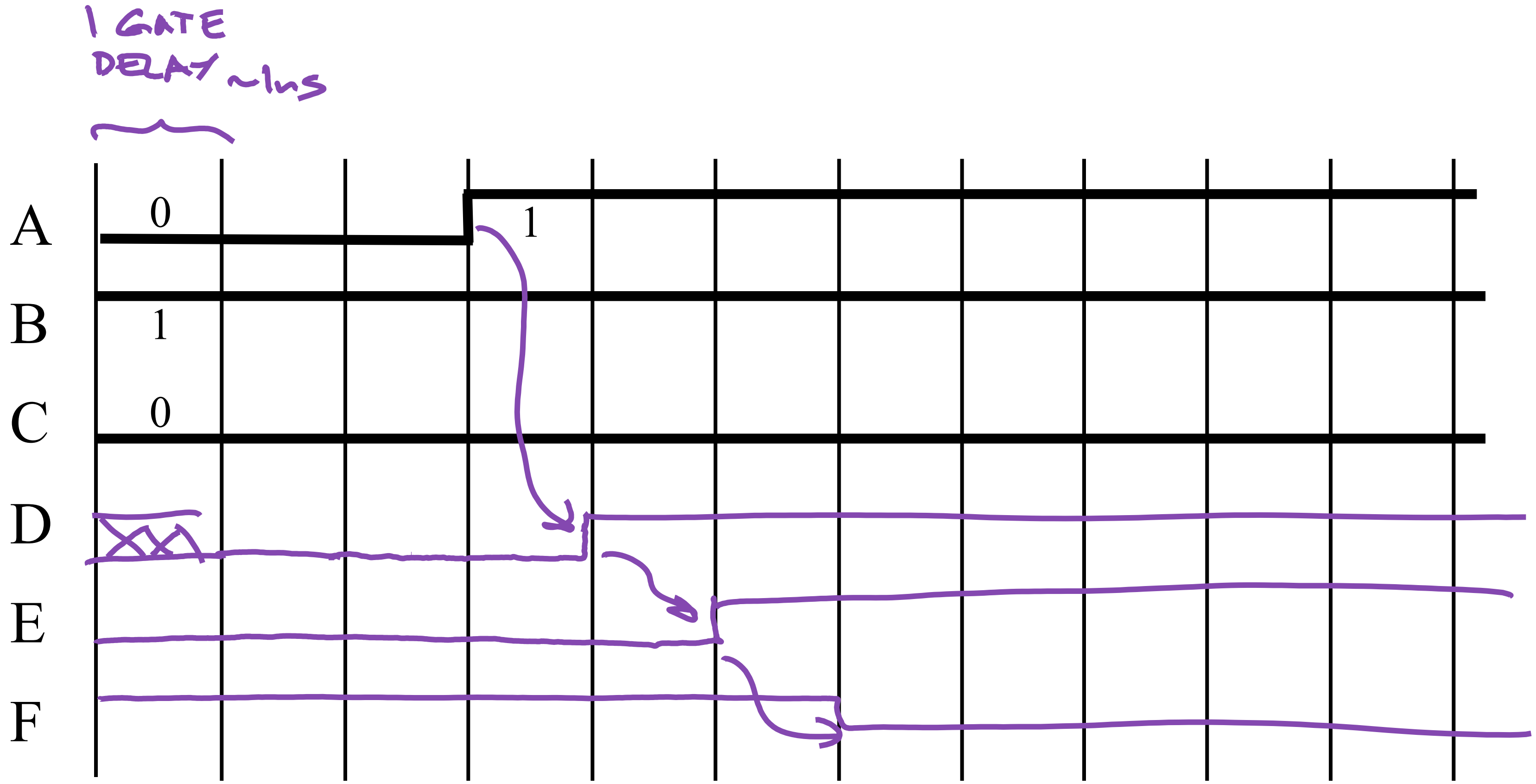
Circuit Timing Behavior

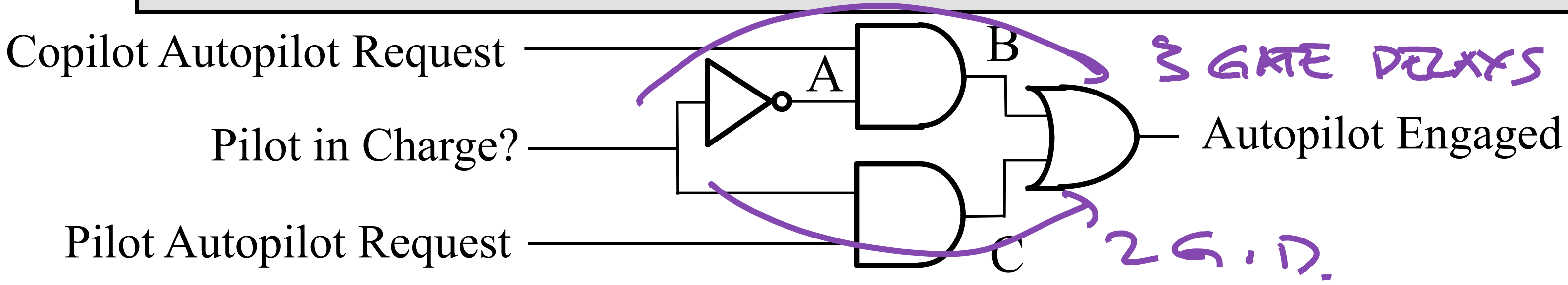
- Simple model: gates react after fixed delay

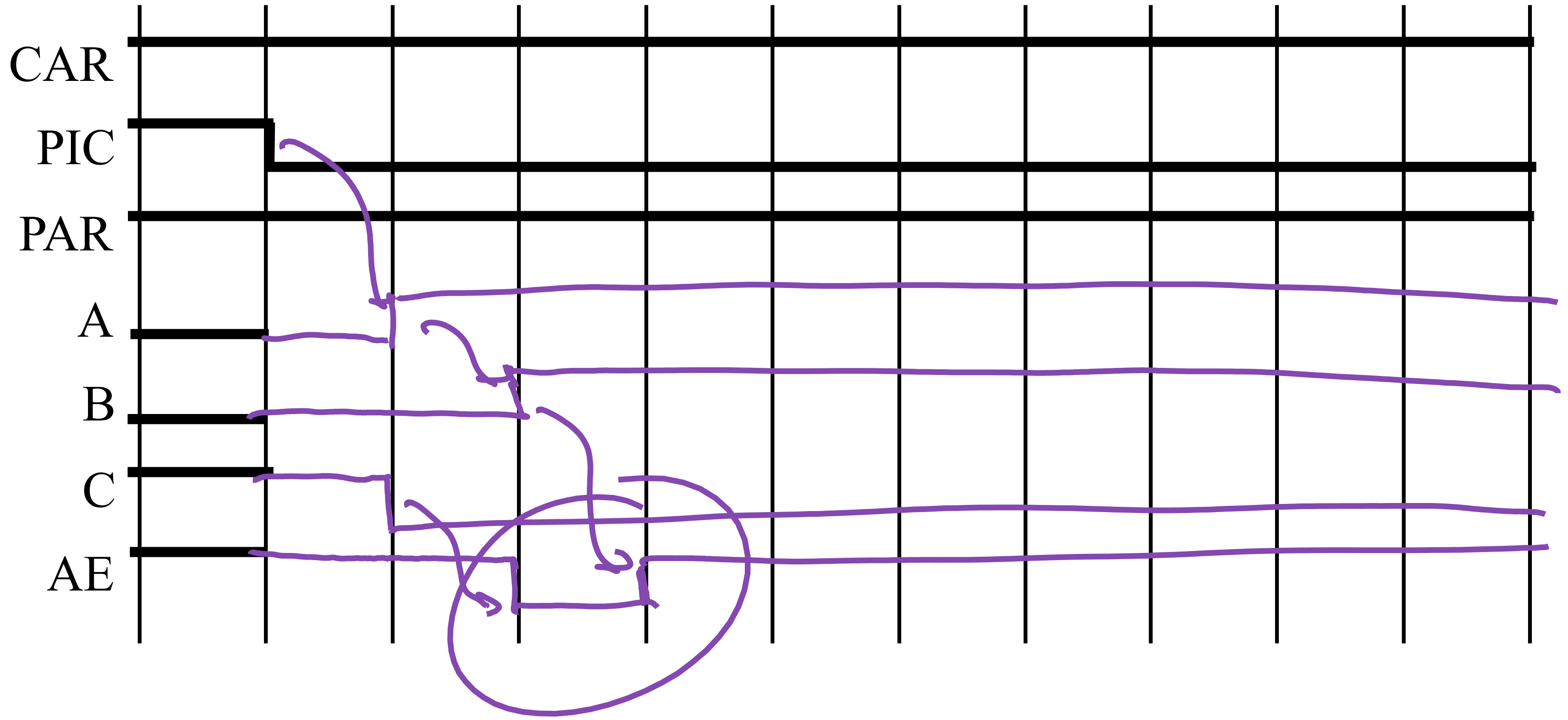




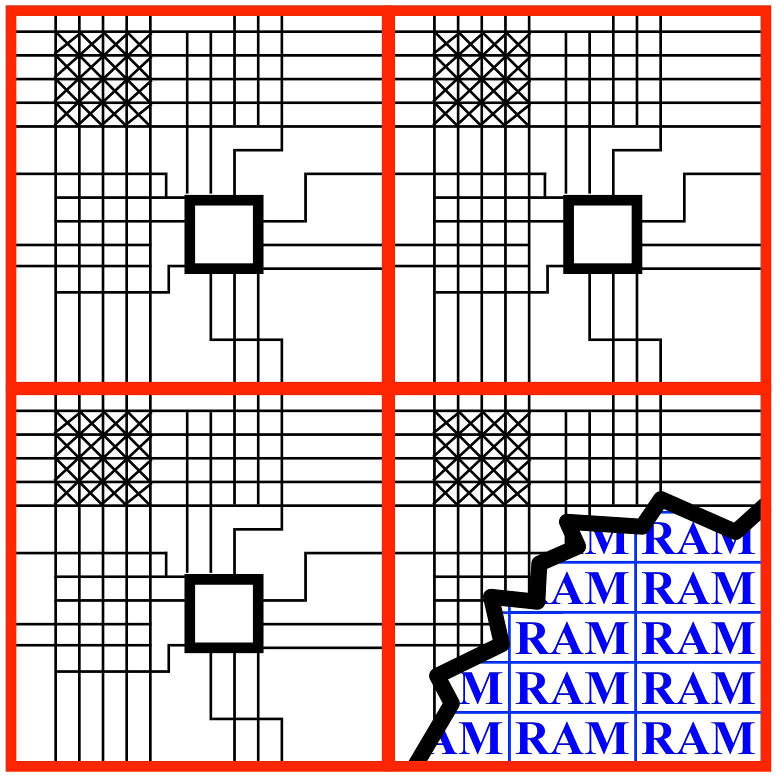
Hazards/Glitches

- Circuit can temporarily go to incorrect states

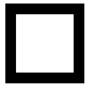




Field Programmable Gate Arrays (FPGAs)



Logic cells imbedded in a general routing structure

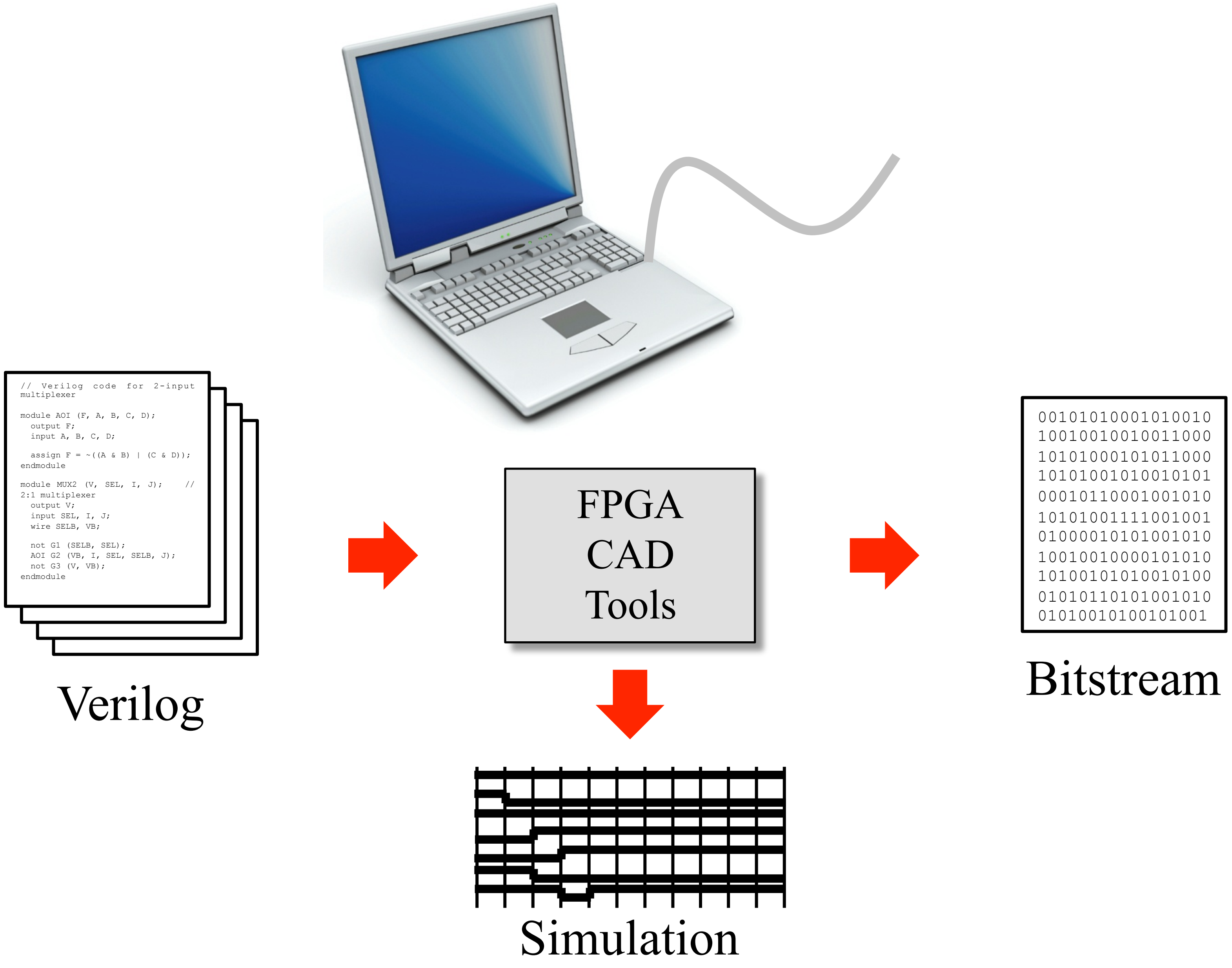


Logic cells usually contain:

- **6-input Boolean function calculator**
- **Flip-flop (1-bit memory)**

All features electronically (re)programmable

Using an FPGA



```
// Verilog code for 2-input
multiplexer

module AOI (F, A, B, C, D);
  output F;
  input A, B, C, D;

  assign F = ~((A & B) | (C & D));
endmodule

module MUX2 (V, SEL, I, J);    //
2:1 multiplexer
  output V;
  input SEL, I, J;
  wire SELB, VB;

  not G1 (SELB, SEL);
  AOI G2 (VB, I, SEL, SELB, J);
  not G3 (V, VB);
endmodule
```

```
00101010001010010
10010010010011000
10101000101011000
10101001010010101
00010110001001010
10101001111001001
01000010101001010
10010010000101010
10100101010010100
01010110101001010
0101001010010100 1
```

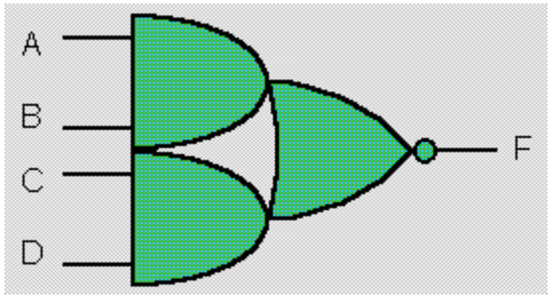
Verilog

- Programming language for describing hardware
  - Simulate behavior before (wasting time) implementing
  - Find bugs early
  - Enable tools to automatically create implementation
- Similar to C/C++/Java
  - VHDL similar to ADA
- Modern version is “System Verilog”
  - Superset of previous; cleaner and more efficient

Structural vs. Behavioral

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

Structural Verilog



COMMENT

"FUNCTION"

```
// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
   output F;
   input A, B, C, D;
```

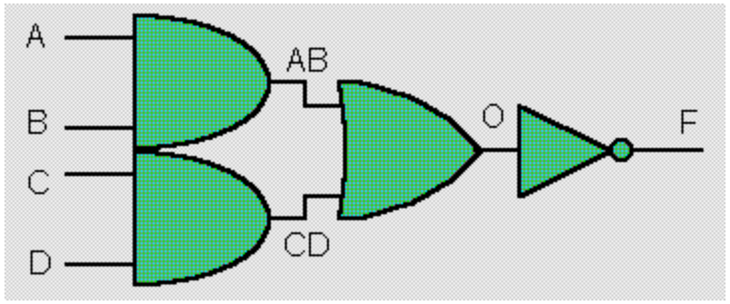
CONTINUOUS ASSIGNMENT : F CHANGES WHEN INPUTS CHANGE

```
   assign F = ~((A & B) | (C & D));
endmodule

// end of Verilog code
```

NOT AND OR

Verilog Wires/Variables



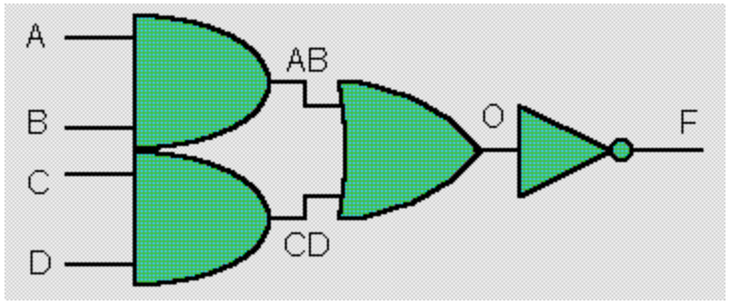
```
// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
  output F;
  input A, B, C, D;
  wire AB, CD, O;  // necessary

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
endmodule
```

LOCAL VARIABLE IN MODULE; ACTUAL WIRE

Verilog Gate Level



```
// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
  output F;
  input A, B, C, D;
  wire AB, CD, O;  // necessary

  and a1(AB, A, B);
  and a2(CD, C, D);
  or o1(O, AB, CD);
  not n1(F, O);
endmodule
```

OTHER KEYWORDS: NAND, NOR, XOR, XNOR, BUF

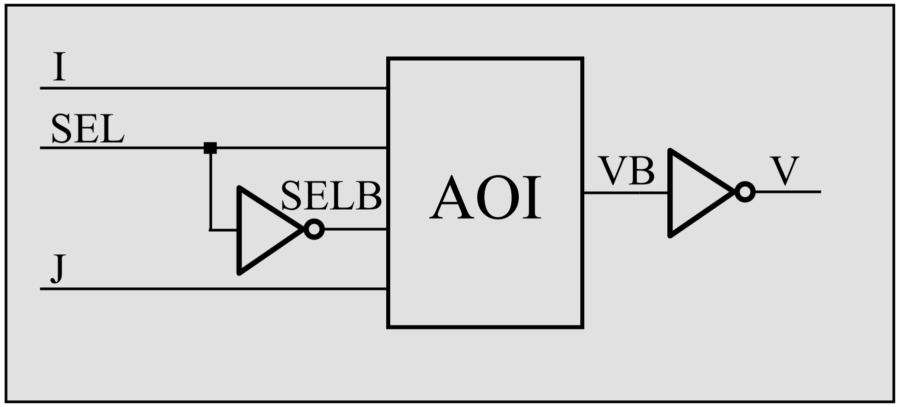
Verilog Hierarchy

```
// Verilog code for 2-input multiplexer

module AOI (F, A, B, C, D);
  output F;
  input A, B, C, D;

  assign F = ~((A & B) | (C & D));
endmodule
```

2-input Mux



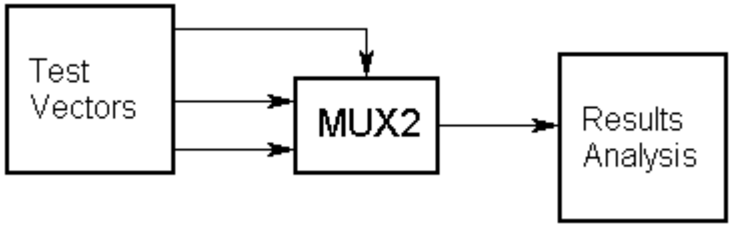
```
module MUX2 (V, SEL, I, J);   // 2:1 multiplexer
  output V;
  input SEL, I, J;
  wire SELB, VB;

  not G1 (SELB, SEL);
  AOI G2 (.F(VB), .A(I), .B(SEL), .C(SELB), .D(J));
  not G3 (V, VB);
endmodule
```

G2 (VB, I, SEL, SELB, J)

Verilog Testbenches

REG: SIMULATION REGISTER - KEEPS TRACK OF SIGNAL VALUE
(USE REG IN "INITIAL" AND "ALWAYS" BLOCK, ELSE WIRE)



```
module MUX2TEST;   // No ports!
    reg SEL, I, J;    // Remembers value - 
reg
   wire V;    EXECUTED ONCE

   initial  // Stimulus
TIME DELAY begin
     SEL = 1; I = 0; J = 0;
     #10 I = 1;
     #10 SEL = 0;
     #10 J = 1;  .V(V)
   end

   MUX2 M (.V, .SEL, .I, .J);   PRINT

   initial  // Response
       $monitor($time, , SEL, I, J, , V);
```

TIME ↓ SEL I J V

```
 0 100 0
10 110 1
20 010 0
30 011 1
```

Data types

- Values on a wire
  - 0, 1, ***x*** (unknown or conflict), ***z*** (tri-state or unconnected)
- Vectors
  - A[3:0]   vector of 4 bits: A[3], A[2], A[1], A[0]
    - Unsigned integer value
    - Indices must be constants
  - Concatenating bits/vectors (curly brackets on left or right side)
    - e.g. sign-extend
      - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
      - {4{A[3]}, A[3:0]} = B[7:0];
  - Style: Use   a[7:0] = b[7:0] + c;
    Not   a = b + c;
  - Bad style but legal syntax: C = &A[6:7]; // *and* of bits 6 and 7 of A

Data types that do not exist

- Structures
- Pointers
- Objects
- Recursive types
- (Remember, Verilog is not C or Java or Lisp or …!)

Numbers

- Format: <sign><size><base format><number>
- 14
  - Decimal number
- –4’b11
  - 4-bit 2’s complement binary of 0011 (is 1101)
- 12’b0000_0100_0110
  - 12-bit binary number (_ is ignored, just used for readibility)
- 3’h046
  - 3-digit (12-bit) hexadecimal number
- Verilog values are unsigned
  - C[4:0] = A[3:0] + B[3:0];
    - if A = 0110 (6) and B = 1010(–6), then C = 10000 (*not* 00000)
    - B is zero-padded, *not* sign-extended

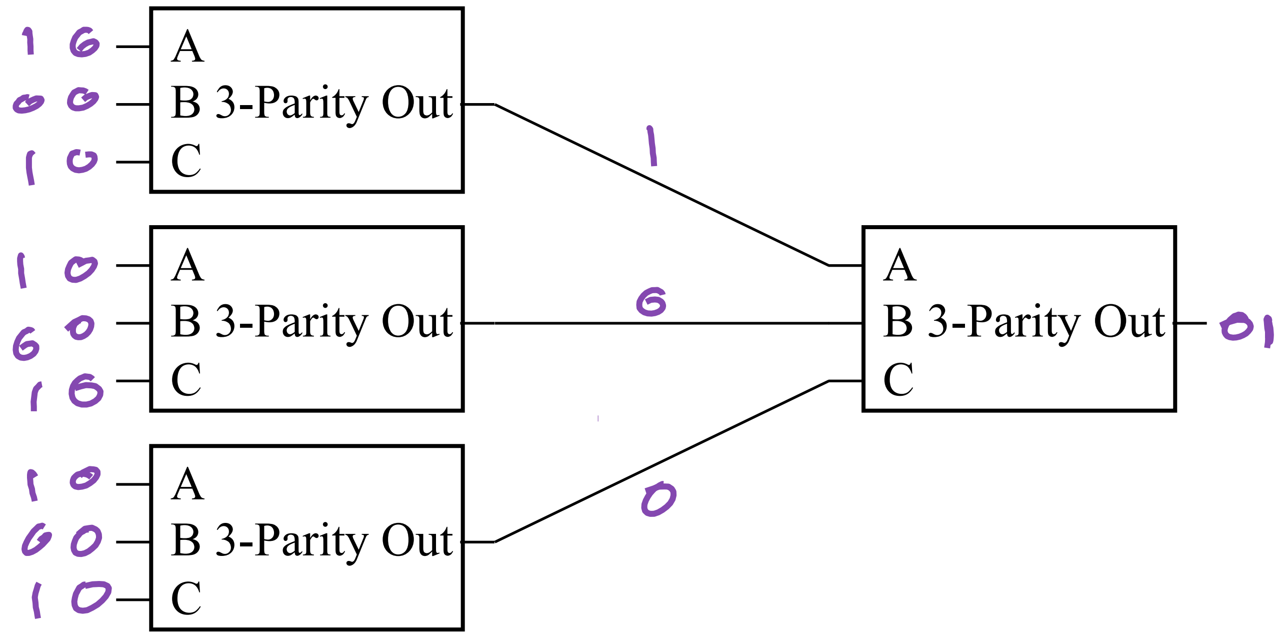
Operators

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
|	reduction OR	Reduction
~&	reduction NAND	Reduction
~|	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift
>	greater than	Relational
>=	greater than or equal to	Relational
<	less than	Relational
<=	less than or equal to	Relational
==	logical equality	Equality
!=	logical inequality	Equality
===	case equality	Equality
!==	case inequality	Equality
&	bit-wise AND	Bit-wise
^	bit-wise XOR	Bit-wise
^~ or ~^	bit-wise XNOR	Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

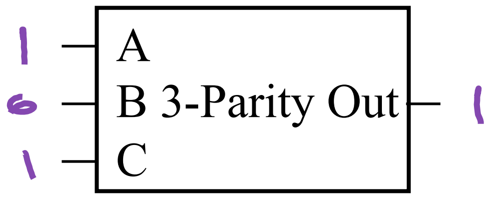
Similar to C operators

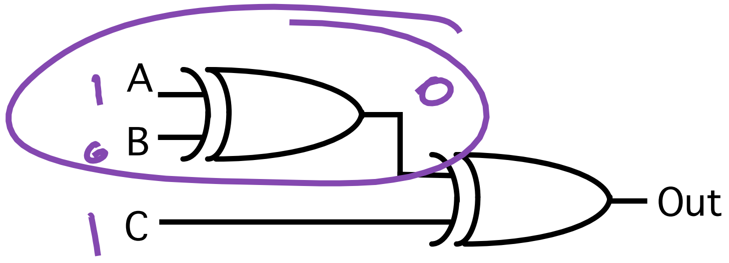
Debugging Complex Circuits

- Complex circuits require careful debugging
  - Rip up and retry?
- Ex. Debug a 9-input odd parity circuit
  - True if an odd number of inputs are true



Debugging Complex Circuits (cont.)





DON'T RIP UP, DEBUG!

Debugging Approach

- Test all behaviors.
  - All combinations of inputs for small circuits, subcircuits.
- Identify any incorrect behaviors.
- Examine inputs and outputs to find earliest place where value is wrong.
  - Typically, trace backwards from bad outputs, forward from inputs.
  - Look at values at intermediate points in circuit.
- DO NOT RIP UP, DEBUG!