Circuit Timing Behavior

- Simple model: gates react after fixed delay

![Circuit Diagram]

1 GATE
DELAY ~1ns

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hazards/Glitches

- Circuit can temporarily go to incorrect states

Circuit Diagram:

- Copilot Autopilot Request
- Pilot in Charge?
- Pilot Autopilot Request

Timing Chart:

- CAR
- PIC
- PAR
- A
- B
- C
- AE

Note: Diagram includes logic gates and timing delays.
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

FPGA CAD Tools

Bitstream

Simulation
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
module AOI (F, A, B, C, D);
  output F;
  input A, B, C, D;
  assign F = ~((A & B) | (C & D));
endmodule

// end of Verilog code
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
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
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 (.F(VB), .A(I), .B(SEL), .C(SELB), .D(J));
    not G3 (V, VB);
endmodule

G2 (VB, I, SEL, SELB, J)
module MUX2TEST;    // No ports!
    reg SEL, I, J;    // Remembers value -
    reg
    wire V;        // EXECUTED ONCE

    initial    // Stimulus
    begin
        SEL = 1; I = 0; J = 0;
        #10 I = 1;
        #10 SEL = 0;
        #10 J = 1;
    end

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

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

- Values on a wire
  - 0, 1, \( x \) (unknown or conflict), \( z \) (tri-state or unconnected)

- Vectors
    - Unsigned integer value
    - Indices must be constants

- Concatenating bits/vectors (curly brackets on left or right side)
  - e.g. sign-extend

- 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 readability)
- 3’ h046
  - 3-digit (12-bit) hexadecimal number
- Verilog values are unsigned
    - if A = 0110 (6) and B = 1010(–6), then C = 10000 (not 00000)
    - B is zero-padded, not sign-extended
## Operators

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

**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.)

A
B 3-Parity Out
C

A
B
Out

C

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!**