Combinational Logic Design using Verilog

Verilog

- Defined in 1985 at Gateway Design Automation Inc., which was then acquired by Cadence Design Systems
- C-like syntax
- Initially a proprietary language, but became open standard in early 1990s, then IEEE standard ("1364") in 1995, revised in 2002, and again in 2005
- VHDL
  - VHDL Hardware Description Language / defined in 1980s / U.S. Dept. of Defense project / Ada-like syntax / IEEE standard ("1076") in 1987
- VHDL & Verilog very similar in capabilities, differ mostly in syntax
- SystemC
  - Defined in 2000s by several companies / C++ libraries and macro routines / IEEE standard ("1666") in 2005
  - Excels for system-level; cumbersome for logic level
- SystemVerilog
  - System-level modeling extensions to Verilog / IEEE Standard ("1800") in 2005

Verilog Modules and Ports

- module DoorOpener(C,H,P,F);
  - input C, H, P;
  - output F;
  - reg F;
- always @(C,H,P)
  - begin
    F <= (~C) & (H | P);
  - end
- endmodule

- ENTITY DoorOpener IS
  - PORT (c, h, p: IN std_logic;
  - f: OUT std_logic);
- ARCHITECTURE Beh OF DoorOpener IS
  - BEGIN
    PROCESS(c, h, p)
    BEGIN
      f <= NOT(c) AND (h OR p);
    END PROCESS;
  END Beh;

AND/OR/NOT Gates

- module And2(X, Y, F);
  - input X, Y;
  - output F;
- module Or2(X, Y, F);
  - input X, Y;
  - output F;

- module Inv(X, F);
  - input X;
  - output F;

Digital Systems and HDLs

- Typical digital components per IC
  - 1960s/1970s: 10-1,000
  - 1980s: 1,000-100,000
  - 1990s: Millions
  - 2000s: Billions

- 1970s
  - IC behavior documented using combination of schematics, diagrams, and natural language (e.g., English)
- 1980s
  - Simulating circuits becoming more important
    - Schematics commonplace
    - Simulating schematic helped ensure circuit was correct before costly implementation

- 1990s
  - Millions
- 2000s
  - Billions

- 1970s
  - IC behavior documented using combination of schematics, diagrams, and natural language (e.g., English)
- 1980s
  - Simulating circuits becoming more important
    - Schematics commonplace
    - Simulating schematic helped ensure circuit was correct before costly implementation

- 1990s
  - Millions
- 2000s
  - Billions

- Other HDLs
  - VHDL
    - VHDL Hardware Description Language / defined in 1980s / U.S. Dept. of Defense project / Ada-like syntax / IEEE standard ("1076") in 1987
  - VHDL & Verilog very similar in capabilities, differ mostly in syntax
  - SystemC
    - Defined in 2000s by several companies / C++ libraries and macro routines / IEEE standard ("1666") in 2005
    - Excels for system-level; cumbersome for logic level
  - SystemVerilog
    - System-level modeling extensions to Verilog / IEEE Standard ("1800") in 2005
AND/OR/NOT Gates

Modules and Ports

Verilog has several dozen keywords:
- User cannot use keywords when naming items like modules or ports (e.g., module, input, and output are keywords above).
- Keywords must be lowercase, not UPPER CASE or a mixture thereof.

User-defined names — Identifiers:
- Begin with letter or underscore (_), optionally followed by any sequence of letters, digits, underscores, and dollar signs ($).

Valid identifiers:
- A, X, Hello, JYYZ, B14, Sig432, Wave_23_F, FS2_Inp_5_5, Input
- Note: underscore and "Input" are valid, but unwise.

Invalid identifiers:
- input (keyword), $ab (doesn't start with letter or underscore), 2A (doesn't start with letter or underscore).

Note: Verilog is case sensitive. Sig432 differs from SIG432 and sig432.

We'll initially capitalize identifiers (e.g., Sig432) to distinguish from keywords.

Y X F
module And2(X, Y, F);
  input X, Y;
  output F;
  reg F;
  always @(X, Y) begin
    F <= X & Y;
  end
endmodule

Y X F
module Or2(X, Y, F);
  input X, Y;
  output F;
  reg F;
  always @(X, Y) begin
    F <= X | Y;
  end
endmodule

Y X F
module Inv(X, F);
  input X;
  output F;
  reg F;
  always @(X) begin
    F <= ~X;
  end
endmodule

Q: Begin a module definition for a 4x1 multiplexor.
- Inputs: i3, i2, i1, i0, s1, s0. Outputs: d.

Y X F
module Mux4(I3, I2, I1, I0, S1, S0, D);
  input I3, I2, I1, I0;
  input S1, S0;
  output D;
  ...

Note that input ports above are separated into two declarations for clarity.

Q: Given that "|" and "~" are built-in operators for OR and NOT, complete the modules for a 2-input OR gate and a NOT gate.

Y X F
module Or2(X, Y, F);
  input X, Y;
  output F;
  reg F;
  always @(X, Y) begin
    F <= X | Y;
  end
endmodule

Y X F
module Inv(X, F);
  input X;
  output F;
  reg F;
  always @(X) begin
    F <= ~X;
  end
endmodule
AND/OR/NOT Gates
Simulation and Testbenches — A First Look

- How does our new module behave?
  - **Simulation**
    - User provides input values, simulator generates output values
    - **Test vectors** — sequence of input values
    - **Waveform** — graphical depiction of sequence

  User provides test vectors
  Simulator generates output values based on HDL description

  Timescale directive is for simulation. More later.

```
timescale 1 ns/1 ns
module And2(X, Y, F);
  input X, Y;
  output F;
  reg F;
  always @(X, Y) begin
    F <= X & Y;
  end
endmodule
```

Instead of drawing test vectors, user can describe them with HDL

```
timescale 1 ns/1 ns
module Testbench();
  reg X_s, Y_s;
  wire F_s;
  And2 CompToTest(X_s, Y_s, F_s);
  initial begin
    // Test all possible input combinations
    #10 Y_s <= 0; X_s <= 0;
    #10 Y_s <= 0; X_s <= 1;
    #10 Y_s <= 1; X_s <= 0;
    #10 Y_s <= 1; X_s <= 1;
    end
endmodule
```

Idea: Create new "Testbench" module that provides test vectors to component's inputs

```
timescale 1 ns/1 ns
module Testbench();
  wire X, Y;
  CompToTest(X, Y, F);
  initial begin
    // Test all possible input combinations
    Y <= 0; X <= 0;
    Y <= 0; X <= 1;
    Y <= 1; X <= 0;
    Y <= 1; X <= 1;
    end
endmodule
```

Note: `CompToTest` short for Component To Test
AND/OR/NOT Gates
Simulation and Testbenches

- Provide testbench file to simulator
  - Simulator generates waveforms
  - We can then check if behavior looks correct

Simulation

timescale 1 ns/1 ns
module Testbench();
  reg X_s, Y_s;
  wire F_s;
  And2 CompToTest(X_s, Y_s, F_s);
  initial begin
    // Test all possible input combinations
    $display($time, "Y_s <= 0; X_s <= 0;");
    $display($time, "Y_s <= 0; X_s <= 1;");
    $display($time, "Y_s <= 1; X_s <= 0;");
    $display($time, "Y_s <= 1; X_s <= 1;");
  end
endmodule

Combinational Circuits
Component Instantiations

- Circuit – A connection of modules
  - Also known as structure
  - A circuit is a second way to describe a module
    - vs. using an always procedure, as earlier
- Instance – An occurrence of a module in a circuit
  - May be multiple instances of a module
    - e.g., Car's modules: tires, engine, windows, etc., with 4 tire instances, 1 engine instance, 6 window instances, etc.

Module instances

Y X F
0 0 0
1 1 1

Modules to be used

BeltWarn

Combinational Circuits
Module Instantiations

- Creating a circuit
  1. Start definition of a new module
  2. Declare nets for connecting module instances
    - N1, N2
      - Note: W is also a declared as a net. By defaults outputs are considered wire nets unless explicitly declared as a reg variable
  3. Create module instances, create connections

Module instantiation statement

And2 And2_1(K, P, N1);
And2 And2_2(N1, N2, W);
Inv Inv_1(S, N2);
And2 And2_3(K, P, S);
Inv Inv_2(K, N1);
And2 And2_4(N1, N2, W);
BeltWarn

BeltWarn example: Turn on warning light (w=1) if car key is in ignition (k=1), person is seated (p=1), and seatbelt is not fastened (s=0)

Combination Circuits
Module Instantiations

- Module instantiation
  - Name of module to instantiate
  - Name of new module instance
    - Must be distinct; hence And2_1 and And2_2
  - Connects instantiated module's ports to nets and variables

Name of new module instance
Name of module to instantiate

And2 And2_1(K, P, N1);
And2 And2_2(N1, N2, W);
Inv Inv_1(S, N2);
Inv Inv_2(K, N1);
And2 And2_3(K, P, S);
And2 And2_4(N1, N2, W);
BeltWarn
Combinational Circuits
Module Instantiations

Q: Complete the 2x1 mux circuit’s module instantiations
1. Start definition of a new module (done)
   (Draw desired circuit, if not already done)
2. Declare nets for internal wires
3. Create module instances and connect ports

```
module Mux2(I1, I0, S0, D);
input I1, I0;
input S0;
output D;

Inv Inv_1 (S0, N1);
And2 And2_1 (I0, N1, N2);
And2 And2_2 (I1, S0, N3);
Or2 Or2_1 (N2, N3, D);
endmodule
```

Combinational Circuit Structure
Simulating the Circuit

Same testbench format for BeltWarn module as for earlier And2 module
```
module Testbench();
reg X_s, Y_s;
wire F_s;
And2 CompToTest(X_s, Y_s, F_s);
initial begin
  // Test all possible input combinations
  Y_s <= 0; X_s <= 0;#10 Y_s <= 0; X_s <= 1;#10
  Y_s <= 1; X_s <= 0;#10 Y_s <= 1; X_s <= 1;#10
  end
endmodule
```

Simulate testbench file to obtain waveforms
```
module Testbench();
reg K_s, P_s, S_s;
wire W_s;
BeltWarn CompToTest(K_s, P_s, S_s, W_s);
initial begin
  K_s <= 0; P_s <= 0; S_s <= 0;#10
  K_s <= 0; P_s <= 1; S_s <= 0;#10
  K_s <= 1; P_s <= 1; S_s <= 0;#10
  K_s <= 1; P_s <= 1; S_s <= 1;
  end
endmodule
```

Combinational Circuit Structure
Simulating the Circuit

More on testbenches
- Note that a single module instantiation statement used
- use reg and wire declarations (K_s, P_s, S_s, W_s) used because procedure cannot access instantiated module's ports directly
- Inputs declared as regs so can assign values (which are held between assignments)
- Note module instantiation statement and procedure can both appear in one module
**Top-Down Design – Combinational Behavior to Structure**

- Designer may initially know system behavior, but not structure
  - BeltWarn: \( W = KPS' \)
- Top-down design
  - Capture behavior, and simulate
  - Capture structure (circuit), simulate again
  - Gets behavior right first, unfettered by complexity of creating structure

**Procedures with Assignment Statements**

- Procedural assignment statement
  - Assigns value to variable
  - Right side may be expression of operators
    - Built-in bit operators include: & (AND), | (OR), ~ (NOT)
    - ^ (XOR), ~^ (XNOR)
- Q: Create an always procedure to compute: \( F = CH + CH' \)

**Example Procedures**

```verilog
'timescale 1 ns/1 ns
module BeltWarn(K, P, S, W);
input K, P, S;
output W;
reg W;
always @(K, P, S) begin
    W <= K & P & ~S;
end
endmodule
```

**Example Two-Output Ex**

```verilog
'timescale 1 ns/1 ns
module TwoOutputEx(A, B, C, F, G);
input A, B, C;
output F, G;
reg F, G;
always @(A, B, C) begin
    F <= (B & B) | ~C;
    G <= (A & B) | (B & C);
end
endmodule
```
Process may use if-else statements (a.k.a. conditional statements)
- If expression is true (evaluates to nonzero value), execute corresponding statement(s)
- If false (evaluates to 0), execute else's statement (else part is optional)
- Example shows use of operator == (logical equality, returns true/false actually, returns 1 or 0)
- True is nonzero value, false is zero

Example shows use of operator == (logical equality, returns 1 or 0)

```
if ((K & P & ~S) == 1)
W <= 1;
else
W <= 0;
```

More than two possibilities
- Handled by stringing if-else statements together
- Known as if-else-if construct
- Example: 4x1 mux behavior
  - If S1S0 change to 01
    - if's expression is false
      - Else's statement executes, which is an if statement whose expression is true

```
module Mux4(I3, I2, I1, I0, S1, S0, D);
input I3, I2, I1, I0;
input S1, S0;
output D;
reg D;
always @(I3, I2, I1, I0, S1, S0)
begin
  if (S1==0 && S0==0)
    D <= I0;
  else if (S1==0 && S0==1)
    D <= I1;
  else if (S1==1 && S0==0)
    D <= I2;
  else
    D <= I3;
end
endmodule
```

Order of assignment statements does not matter.
Placing two statements on one line does not matter.
To execute multiple statements if expression is true, enclose them between "begin" and "end"

```
if (I1==0 && I0==0)
begin
  D3 <= 0; D2 <= 0;
  D1 <= 0; D0 <= 1;
end
else if (I1==0 && I0==1)
begin
  D3 <= 0; D2 <= 0; D1 <= 1; D0 <= 0;
end
else if (I1==1 && I0==0)
begin
  D3 <= 0; D2 <= 1;
  D1 <= 0; D0 <= 0;
end
else
begin
  D3 <= 1; D2 <= 0;
  D1 <= 0; D0 <= 0;
end
```

Top-down design
- Capture behavior, and simulate
- Capture structure using a second module, and simulate

```
module BeltWarn(K, P, S, W);
input K, P, S;
output W;
wire N1, N2;
And2 And2_1(K, P, N1);
Inv  Inv_1(S, N2);
And2 And2_2(N1, N2, W);
endmodule
```
Discuss how last else could have been "else if (I1==1 && I0==1)"?
**Top-Down Design – Combinational Behavior to Structure**

**Common Pitfall – Missing Inputs from Event Control Expression**

- **Pitfall** - Missing inputs from event control's sensitivity list when describing combinational behavior
  - Results in sequential behavior
  - Wrong 4x1 mux example
    - Has memory
    - No compiler error
    - Still not a true

```
module Mux4(I3, I2, I1, I0, S1, S0, D);
input I3, I2, I1, I0;
input S1, S0;
output D;
reg D;
always @(S1, S0)
begin
    if (S1==0 && S0==0)
        D <= I0;
    else if (S1==0 && S0==1)
        D <= I1;
    else if (S1==1 && S0==0)
        D <= I2;
    else
        D <= I3;
end
endmodule
```

- **Reminder**
  - Combinational behavior: Output value is purely a function of the present input values
  - Sequential behavior: Output value is a function of present and past input values, i.e., the system has memory

Verilog provides mechanism to help avoid this pitfall

- @* – implicit event control expression
  - Automatically adds all nets and variables that are read by the controlled statement or statement group
  - Thus, @* in example is equivalent to @(S1,S0,I0,I1,I2,I3)
  - @(*) also equivalent

```
module Mux4(I3, I2, I1, I0, S1, S0, D);
input I3, I2, I1, I0;
input S1, S0;
output D;
reg D;
always @*
begin
    if (S1==0 && S0==0)
        D <= I0;
    else if (S1==0 && S0==1)
        D <= I1;
    else if (S1==1 && S0==0)
        D <= I2;
    else
        D <= I3;
end
endmodule
```

**Top-Down Design – Combinational Behavior to Structure**

**Common Pitfall – Output not Assigned on Every Pass**

- **Pitfall** - Failing to assign every output on every pass through the procedure for combinational behavior
  - Results in sequential behavior
  - Wrong 2x4 decoder example
    - Has memory
    - No compiler error
    - Just not a decoder

```
`timescale 1 ns/1 ns
module Dcd2x4(I1, I0, D3, D2, D1, D0);
input I1, I0;
output D3, D2, D1, D0;
reg D3, D2, D1, D0;
always @(I1, I0)
begin
    if (I1==0 && I0==0)
        begin
            D3 <= 0;
            D2 <= 0;
            D1 <= 0;
            D0 <= 1;
        end
    else if (I1==0 && I0==1)
        begin
            D3 <= 0;
            D2 <= 0;
            D1 <= 1;
            D0 <= 0;
        end
    else if (I1==1 && I0==0)
        begin
            D3 <= 0;
            D2 <= 1;
            D1 <= 0;
            D0 <= 0;
        end
    else if (I1==1 && I0==1)
    begin
        D3 <= 1;
    end
    // Note: missing assignments
    // to every output in last "else"
    endmodule
```

- **Reminder**
  - Same pitfall often occurs due to not considering all possible input combinations

```
if (I1==0 && I0==0)
begin
    D3 <= 0;
    D2 <= 0;
    D1 <= 0;
    D0 <= 1;
end
else if (I1==0 && I0==1)
begin
    D3 <= 0;
    D2 <= 0;
    D1 <= 1;
    D0 <= 0;
end
else if (I1==1 && I0==0)
begin
    D3 <= 0;
    D2 <= 1;
    D1 <= 0;
    D0 <= 0;
end
else if (I1==1 && I0==1)
begin
    D3 <= 1;
end
// Note: missing assignments
// to every output in last "else"
endmodule
```
Hierarchical Circuits
Using Modules Instances in Another Module

- Module can be used as instance in a new module
  - As seen earlier: And2 module used as instance in BeltWarn module
  - Can continue: BeltWarn module can be used as instance in another module
    - And so on
  - Hierarchy powerful mechanism for managing complexity

BeltWarn
WindowLock
Display

Hierarchical Circuits
Using Module Instances in Another Module

4-bit 2x1 mux example

- 4-bit 2x1 mux example
  - Create four Mux2 instances
  - Can then use Mux2_4b in another module's circuit, and so on ...

Hierarchical Circuits
Using Module Instances in Another Module

4-bit 2x1 mux example

- 4-bit 2x1 mux example
  - `timescale 1 ns/1 ns
  - module Mux2(I1, I0, S0, D);
    - input I1, I0;
    - input S0;
    - output D;
    - wire N1, N2, N3;
    - Inv Inv_1 (S0, N1);
    - And2 And2_1 (I0, N1, N2);
    - And2 And2_2 (I1, S0, N3);
    - Or2 Or2_1 (N2, N3, D);
  - endmodule

Built-In Gates

- We previously defined AND, OR, and NOT gates
- Verilog has several built-in gates that can be instantiated
  - and, or, and, nor, xor, xnor
  - One output, one or more inputs
  - The output is always the first in the list of port connections
- Example of 4-input AND:
  - and4 (out, in1, in2, in3, in4);
- not is another built-in gate
- Earlier BeltWarn example using built-in gates
- Note that gate size is automatically determined by the port connection list