Digital Systems and HDLs

- Typical digital components per IC
  - 1960s/1970s: 10–1,000
  - 1980s: 1,000–100,000
  - 1990s: Millions
  - 2000s: Billions
- IC behavior documented using:
  - Hierarchical schematic diagrams
  - Natural language (e.g., English)
- 1980s:
  - Simulating circuits using Verilog
  - Verilog was a language for digital design
  - Defined in the 1980s at Gateway Design Automation Inc.
  - C-like syntax
  - Initially a proprietary language, but became open
  - Defined in 1985 at Gateway Design Automation Inc.,
    which was then acquired by Cadence Design Systems
  - EDA (Electronic Design Automation)
  - Hardware description languages (HDLs)
  - Machine-readable language for describing hardware
  - Test language could be more efficient means of circuit entry than graphical language

Other HDLs
- VHDL
  - Initial name was VHSIC Hardware Description Language
  - Defined in 1987 by several companies
  - Syntax close to Pascal
  - ieee standard 1076 in 1987
  - IEEE standard (“1076”) in 1993
  - Many, many more

HDLs for Simulation
- Hardware description languages (HDLs) –
  - Machine-readable textual languages for describing hardware
  - Machine-readable languages (HDLs)
    - Simulate circuit behavior
    - Should match

HDLs for Design and Synthesis
- HDLs became increasingly used for designing ICs using top-down design process
  - Convert higher-level description into lower-level one
  - Describe circuit in HDL simulate
    - Physical design tools automatically convert to low-level IC design
  - Describe circuit in HDL simulate
    - E.g., Describe adder as $A + B + C$, rather than as circuit of hundreds of logic gates
    - Compact description, designers get function right first
  - Design circuits
    - Manually, or
    - Using synthesis tools, which automatically convert HDL behavior into HDL circuit
    - Simulate circuit, should match
HDLs for Synthesis

• Use of HDLs for synthesis is growing
  – Circuits are more complex
  – Synthesis tools are maturing

• But HDLs originally defined for simulation
  – General language
  – Many constructs not suitable for synthesis
    • e.g., delays
  – Behavior description may simulate, but not synthesize, or may synthesize to incorrect or inefficient circuit

• Not necessarily synthesis tool’s fault!

Verilog for Digital Design

• This book introduces use of Verilog for design and synthesis
  – In contrast to books that introduce the general language first, and then (maybe) describe synthesis subset
  – No need to learn entire French language if your goal is just to write recipes in French

• Shows use of Verilog for increasingly complex digital systems
  – Combinational logic design
  – Sequential logic design
  – Datapath components
  – Register transfer level (RTL) design
  – Emphasizes a very disciplined use of the language for specific purposes

• Book can be used as supplement to digital design textbook
  – Specifically follows structure and examples of “Digital Design” by Frank Vahid, John Wiley and Sons, 2007
  – But can be used with other books too
  – Can also be used as standalone introduction to Verilog

AND/OR/NOT Gates

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

Likewise, consider HDL language

– General and complex; many uses
– But use for synthesizing circuits is greatly restricted
  – Synthesis tool understands: sensitivity lists, if statements, ...
  – Synthesis tool may not understand: wait statements, while loops, ...
  – HDL circuit is bad, don’t blame the tool!

– This book emphasizes use of Verilog for design and synthesis

Verilog has several dozen keywords

• User cannot use keywords when naming items like modules or ports
• module, input, output are keywords above
• Keywords must be lower case, 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 sign ($)
• Valid identifiers: s, S_4, X_2, $A, B, Right2, Wire_2, _F, _F2, _Go, _x, _y
• A “bang” (!), “and” (!), or “or” (!) are valid, but rare
• Invalid identifiers: Input (begin with keyword), !A (bang start with letter or digits), !F (bang start with letter or digit)

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

– WtF initially capitalizes identifiers (e.g., Sig132) to distinguish from keywords

HDL behavior

Synthesis

HDL circuit

Simulate

Verilog Modules and Ports

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

Q: Begin a module definition for a 4x1 multiplexer

– Inputs: I3, I2, I1, I0, SI, SO, D
– Outputs: D

module Mux4(I3, I2, I1, I0, SI, SO, D);
input I3, I2, I1, I0;
input SI, SO;
output D;

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
    - Waveform — graphical depiction of sequence
    - Test vectors — sequence of input values
  - Simulation and Testbenches
    - AND/OR/NOT Gates
      - Instantiate module, map variables to ports
    - Module with no ports
      - Procedure that executes repetitively
        - `timescale` directive is for simulation. More later.
      - Sensitivity list
        - Sometimes called "sensitivity list"
      - `always` procedure that executes repetitively
        - Assigns value to variable
        - Declares a variable data type, which holds its value between assignments
      - Needed for `F` to hold value between assignments
    - A reg variable may or may not correspond to an actual physical register. There obviously is no register inside an AND gate.

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

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.

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

```verilog
module NOT2(F);
  input F;
  output F;
  reg F;
  always @(F) begin
      F <= ~F;
  end
endmodule
```

- Waveform — graphical depiction of sequence
- Test vectors — sequence of input values
  - User provides input values, simulator generates output values
  - Procedure that executes at simulation start, but repeats
  - Delay control — number of time units to delay this statement's execution
  - Initial — procedure that executes at simulation start, but repeats
  - Valid time units — s (seconds), ms (milliseconds), us (microseconds), ns (nanoseconds)
  - Precision is for internal rounding. For our purposes, precision will be set same as time unit.

Note: CompToTest short for Component To Test

```verilog
// User-provided input values, simulator generates output values
// User provides test vectors
// Test vectors — sequence of input values
// Waveform — graphical depiction of sequence
// User provides test vectors
// Simulator generates output values based on HDL description
// Simulator

// Note: `timescale 1 ns/1 ns` is for simulation. More later.
```

```verilog
// User-provided input values, simulator generates output values
// User provides test vectors
// Test vectors — sequence of input values
// Waveform — graphical depiction of sequence
// User provides test vectors
// Simulator generates output values based on HDL description
// Simulator
```
AND/OR/NOT Gates
Simulation and Testbenches

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

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.

Creating a circuit

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

And2

3. Create module instances, create connections

Combinational Circuit Structure
Simulating the Circuit

- Same testbench format for BeltWarn module as for earlier And2 module
Combinational Circuit Structure

Simulating the Circuit

- Simulate testbench file to obtain waveforms

- `timescale 1 ns/1 ns

- 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
  - 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

Top-Down Design – Combinational Behavior to 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:
      - \( F = CY + CH' \)

  **Answer 1:**
  ```vhdl
  always @(C, H) begin
    F <= (B & B) | ~C;
  end
  ```

  **Answer 2:**
  ```vhdl
  always @(C, H)
  begin
    F <= C ^ H;
  end
  ```
Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- Process may use if-else statements (a.k.a. conditional statements)
  - if (expression)
    - If expression is true (evaluates to nonzero value), execute corresponding statement(s)
    - False (evaluates to 0), execute else’s statement (else part is optional)
  - Example shows use of operator &&
    - Logical AND, returns true/false
  - True is nonzero value, false is zero

Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- Q: Create procedure describing behavior of a 2x4 decoder using if-else construct
- Example: 4x1 mux behavior
- More than two possibilities
  - Handled by stringing if-else statements together
- Known as if-else-if construct
- Example: 4x1mux behavior
  - Suppose S100 change to 01
    - If expression is false
      - Else’s statement executes
      - This pitfall
        - Missing inputs from event control’s sensitivity list when describing combinational behavior
        - Results in sequential behavior
    - Place two statements on one line does not matter.
    - To execute multiple statements if expression is true, enclose them between "begin" and "end"

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
      - Max memory
      - No compiler error
      - And not a true

Top-Down Design – Combinational Behavior to Structure
Common Pitfall – Missing Inputs from Event Control Expression

- Verilog provides mechanism to help avoid this pitfall
Discuss how last else could have been “else if (1==1 && 10==1)”?
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
  - 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

```
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 if"
end
endmodule
```

Hierarchical Circuits
Using Module Instances in Another Module

• Module can be used as an 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

```
module BeltWarn(K, P, S, W);
input K, P, S;
output W;
wire N1, N2;
and And_1(N1, K, P);
not Inv_1(N2, S);
and And_2(W, N1, N2);
endmodule
```

Hierarchical Circuits
Using Module Instances in Another Module

• 4-bit 2x1 mux example

```
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
```

Hierarchical Circuits
Using Module Instances in Another Module

• 4-bit 2x1 mux example

```
module Mux2_4b(A3, A2, A1, A0, B3, B2, B1, B0, S0, C3, C2, C1, C0);
input A3, A2, A1, A0;
input B3, B2, B1, B0;
input S0;
output C3, C2, C1, C0;
Mux2  Mux2_3  (B3, A3, S0, C3);
Mux2  Mux2_2  (B2, A2, S0, C2);
Mux2  Mux2_1  (B1, A1, S0, C1);
Mux2  Mux2_0  (B0, A0, S0, C0);
endmodule
```

Built-In Gates

• We previously defined AND, OR, and NOT gates
• Verilog has several built-in gates that can be instantiated
  - and, or, nand, 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:
      - and (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

```
module BeltWarnAND(K, P, S, W);
input K, P, S;
output W;
wire Mux2_0;
and Mux2_0 (W, K, P);
endmodule
```

Hierarchical Circuits
Using Module Instances in Another Module

• 4-bit 2x1 mux example

```
module Mux2_4b(A3, A2, A1, A0, B3, B2, B1, B0, S0);
input A3, A2, A1, A0;
input B3, B2, B1, B0;
input S0;
output C3, C2, C1, C0;
Mux2  Mux2_3  (B3, A3, S0, C3);
Mux2  Mux2_2  (B2, A2, S0, C2);
Mux2  Mux2_1  (B1, A1, S0, C1);
Mux2  Mux2_0  (B0, A0, S0, C0);
endmodule
```