Introduction to Verilog
Review on Digital Logic Design
Figure 2.8: Gate network corresponding to $E(x_2, x_1, x_0) = \sum m(1, 2, 6)$. 

$$z = \sum m(1,2,6)$$
Figure 8.12: D flip-flop and its state diagram.
Figure 8.12: D flip-flop and its state diagram.
Input: $x(t) \in \{a, b, c\}$  
Output: $z(t) \in \{0, 1\}$  
State: $s(t) \in \{A, B, C, D\}$  
Initial state: $s(0) = A$

Functions: The state-transition and output functions

<table>
<thead>
<tr>
<th>$PS$</th>
<th>Input $x$</th>
<th>$x = a$</th>
<th>$x = b$</th>
<th>$x = c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C,0</td>
<td>B,1</td>
<td>B,0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>D,0</td>
<td>B,0</td>
<td>A,1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A,0</td>
<td>D,1</td>
<td>D,0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>B,0</td>
<td>A,0</td>
<td>D,1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input code</th>
<th>State code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

$NS, z$
State-transition and output functions

<table>
<thead>
<tr>
<th>PS</th>
<th>$x_1x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1y_0$</td>
<td>01 10 11</td>
</tr>
<tr>
<td>00</td>
<td>01,0 10,1 10,0</td>
</tr>
<tr>
<td>10</td>
<td>11,0 10,0 00,1</td>
</tr>
<tr>
<td>01</td>
<td>00,0 11,1 11,0</td>
</tr>
<tr>
<td>11</td>
<td>10,0 00,0 11,1</td>
</tr>
</tbody>
</table>

$Y_1Y_0$, $z$

$NS$, Output

[Diagram of state transition and output functions]
- State-transition and output functions

<table>
<thead>
<tr>
<th>PS</th>
<th>$x_1x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1y_0$</td>
<td>01 10 11</td>
</tr>
<tr>
<td>00</td>
<td>01,0 10,1 10,0</td>
</tr>
<tr>
<td>10</td>
<td>11,0 10,0 00,1</td>
</tr>
<tr>
<td>01</td>
<td>00,0 11,1 11,0</td>
</tr>
<tr>
<td>11</td>
<td>10,0 00,0 11,1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Y_1Y_0$, $z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NS$, Output</td>
</tr>
</tbody>
</table>

$Y_1$: \[
\begin{array}{c|c|c|c}
\hline
 & x_0 & \quad & \quad \\
\hline
y_1 & 0 & 1 & 1 \\
\hline
x_1 & 0 & 1 & 1 \\
\hline
\end{array}
\]

$y_0$: \[
\begin{array}{c|c|c}
\hline
 & \quad & \quad \\
\hline
0 & y_0 & \quad \\
\hline
1 & \quad & \quad \\
\hline
1 & \quad & \quad \\
\hline
\end{array}
\]
- State-transition and output functions

<table>
<thead>
<tr>
<th>$PS$</th>
<th>$x_1x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1y_0$</td>
<td>01 10 11</td>
</tr>
<tr>
<td>00</td>
<td>01,0 10,1 10,0</td>
</tr>
<tr>
<td>10</td>
<td>11,0 10,0 00,1</td>
</tr>
<tr>
<td>01</td>
<td>00,0 11,1 11,0</td>
</tr>
<tr>
<td>11</td>
<td>10,0 00,0 11,1</td>
</tr>
</tbody>
</table>

$Y_1Y_0, z$
$NS, Output$

$Y_1: \begin{array}{c|ccc}
 x_0 & 0 & 1 & 1 \\
 \hline
 0 & 0 & 1 & 1 \\
 1 & 0 & 1 & 1 \\
 \end{array}$

$y_1 \\ y_0 \\
-110 \\
-101$

\[ Y_1 = y_1'x_1 + y_1'x_1 + y_0'x_0 + y_0x_1x_0 \]
- State-transition and output functions

<table>
<thead>
<tr>
<th>$PS$</th>
<th>$x_1x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1y_0$</td>
<td>01 10 11</td>
</tr>
<tr>
<td>00</td>
<td>01,0 10,1 10,0</td>
</tr>
<tr>
<td>10</td>
<td>11,0 10,0 00,1</td>
</tr>
<tr>
<td>01</td>
<td>00,0 11,1 11,0</td>
</tr>
<tr>
<td>11</td>
<td>10,0 00,0 11,1</td>
</tr>
</tbody>
</table>

$Y_1Y_0, \ z$

$NS, \ Output$

$Y_1:\ \ \ \begin{array}{c|c|c}
 x_0 & 0 & 1 & 1 \\
-011 & y_1 & y_0 \\
110 & 101 &
\end{array}$$

$Y_0:\ \ \ \begin{array}{c|c|c}
 x_0 & 0 & 1 \\
-100 & y_1 & y_0 \\
011 & 100 &
\end{array}$$

$z:\ \ \ \begin{array}{c|c|c}
 x_0 & 0 & 1 \\
-001 & y_1 & y_0 \\
010 & 100 &
\end{array}$$

\begin{align*}
Y_1 &= y'_1x_1 + y_1x'_1 + y'_0x'_0 + y_0x_1x_0 \\
Y_0 &= y'_0x'_1 + y'_1y_0x_1 + y_0x_1x_0 \\
z &= y'_1x'_0 + y_1x_1x_0
\end{align*}
Figure 8.19: Sequential network in Example 8.4.
Figure 8.19: Sequential network in Example 8.4.
Introduction to Verilog
Overview of Verilog

1. Resemblance to C: case sensitive, same style comments, operators, etc.

2. Two standards: Verilog-1995, Verilog-2001, both supported by ISE

3. VHDL is the other widely used HDL.
Behavioral vs Synthesizable code

Historically Verilog has always served two functions:

1. Describe your digital logic design in a high-level way instead of using schematics (synthesizable code).
2. Model elements that interact with your design (behavioral models/testbenches)

You should **always** have a clear conscience of whether you are writing behavioral or synthesizable code.

Better yet, know what your synthesizable code translates to in **physical hardware**.
Behavioral vs Synthesizable code

Which part corresponds to behavioral / synthesizable code?
Data types: wire, reg

Variables must be declared before used; wire/reg are the most common data types.

wire:
- models basic wire that holds transient values

reg:
- anything that stores a value
Data Values

Values: 1, 0, x/X (unknown), z/Z (high impedance)

Radices: d(decimal), h(hex), o(octal), b(binary)

Format: <size>'<radix><value>

123  // default: decimal radix, unspecified width
’d123  // ‘d = decimal radix
8’h7B  // ‘h = hex radix
‘o173  // ‘o = octal radix
‘b111_1011  // ‘b = binary radix, “_” are for readability

Signed number (rarely used):
8’shFF (8-bit twos-complement representation of -1)
Operators

Bitwise: $\sim a$, $a \& b$, $a | b$, $a^b$, $a^{\sim b}$

Reduction: bitwise operation on a **single** operand and produces one bit result.
$\& a$, $\sim \& a$, $| a$, $\sim | a$, $^a$, $~^a$
Example: $^4'b1001 = 1'b0$

Logical: output one bit result
$! a$, $a \& \& b$, $a | | b$, $==/!=, ===/!==$ (also compares x/z)

Arithmetic: $+, -, *, /, \%$ (mod, takes the sign of operand 1)
Can you tell the difference between the following expressions? What are the sizes of each expression and what do they mean?

Declaration: wire [31:0] a,b

- a & b
- a && b
- (&a) & (&b)
- (&a) && (&b)
- &(a && b)
- a && (&b)
Operators, ctd.

Shift: \(<<, >>\>\) (logical)
\(<<<<, >>>>\>\) (arithmetic)

Conditional: \(sel \, ? \, a : b\)

Concatenation: \(\{a, b\}\)

Replication: \(\{n\{m\}\}\)

Assignment: \(=\) (blocking), \(<=\) (non-blocking)
Blocking/Non-blocking assignments

Blocking assignments (=):
assignment immediate, happens first

Non-blocking assignment(<=):
assignments deferred until all right hand sides has been evaluated, closer to actual hardware register behavior.
Assignment operator guidelines

1. Sequential logic: use nonblocking (<=)
2. Pure combinational logic in “always” block: use blocking (=)
3. Do not mix blocking/non-blocking in the same “always” block
4. Both sequential and “combinational” logic in the same “always” block: use non-blocking (<=)
   
   \[ y \leftarrow a \ ^ \ b; \]

Recommendation: in complex sequential circuit, use two always blocks, one for combinational circuit, one for state update.
Basic building block: modules

1. port: input / output / inout
2. Signal declarations: wire [3:0] a;
   a. Rule-of-thumb: stick to [MSB:LSB]
3. Concurrent logic blocks, one of the following:
   a. continuous assignments
   b. always blocks (either sequential or combinatorial)
   c. initial blocks (only used in behavioral modeling)
   d. forever blocks (only used in behavioral modeling)
4. Instantiations of sub-modules

module top(a, b, ci, s, co);
   input a, b, ci;
   output s, co;
   wire s;
   reg g, p, co;
   assign s = a ^ b ^ ci;
   // combinatorial always
   // block using begin/end
   always @* begin
      g = a & b;
      p = a | b;
      co = g | (p & ci);
   end
endmodule
Basic building block: modules

1. **port**: input / output / inout

2. **Signal declarations**: 
   - `wire [3:0] a;`
   - **Rule-of-thumb**: stick to [MSB:LSB]

3. **Concurrent logic blocks**, one of the following:
   - a. **continuous assignments**
   - b. **always blocks** (either sequential or combinatorial)
   - c. **initial blocks** (only used in behavioral modeling)
   - d. **forever blocks** (only used in behavioral modeling)

4. **Instantiations of sub-modules**

```verilog
module top(a, b, ci, s, co);
  input a, b, ci;
  output s, co;
  wire s;
  reg g, p, co;
  assign s = a ^ b ^ ci;
  // combinatorial always block using begin/end
  always @* begin
    g = a & b;
    p = a | b;
    co = g | (p & ci);
  end
endmodule
```
// combinatorial always
// block using begin/end
always @* begin
  g = a & b;
  p = a | b;
  co = g | (p & ci);
end

Sensitivity list
Code that describes the real "function"
Always Block: Sensitivity list

1. Level sensitive: changes to any signals on the sensitivity list will invoke the always block. Used in combinational circuits.
   always @ (a or b) / always @* (verilog-2001, recommended)

1. Edge sensitive: invoke always block on specified signal edges. Used in sequential circuits.
   always @ (posedge clk or posedge reset)
Aways Block: case, if/else

case (sel)
  val0: statements
  val1: statements
  ...  
  default: statements
endcase

if (condition)
  statements
else if (condition)
  statements
  ...
else
  statements
module ALU(a, b, f, z);
input [3:0] a, b;
input [1:0] f;
output reg [3:0] z;

always @(*)
  z = 4’bx; (#1)
  case (f)
    0: z = a + b;
    1: z = a * b;
    2: z = a & b;
    3: z = a | b;
    default: z = 4’bx; (#2)
  endcase
endmodule

if/else or case statement fails to cover all cases → inferred latch

Example:
if (x == 2'b00)
  z = a;
else if (x == 2'b01)
  z = b;

What about other cases? The circuit will hold previous value
To avoid: add #1 or #2
Always Block: Looping (mostly for testing)

while (condition)
  statements

for (initialization; condition; increment)
  statements

repeat (n)
  statements
Data types: wire, reg

Variables must be declared before used; wire/reg are the most common data types.

wire:
- models basic wire that holds transient values
- **only wires can be used on the LHS of continuous assign statement**
- input/output port signals default to wire

reg:
- anything that stores a value
- can appear in both combinational and sequential circuits
- **only regs can be used on the LHS of non-continuous assign statement**
Wire vs. reg

When to use wire:
Left hand side of “assign”

wire s;
assign s = a ^ b ^ ci;

When to use reg:
Left hand side of “=” or “<=” in “always” blocks

reg g, p, co;
always @* begin
  g = a & b;
  p = a | b;
  co = g | (p & ci);
end
Sequential circuit example: modulo 64 counter

module counter(clk, rst, out);
  input clk, rst;
  output [5:0] out;

  reg [5:0] out;
  always @(posedge clk or posedge rst)
  begin
    if (rst)
      out <= 6'b000000;
    else
      out <= out + 1;
  end
endmodule
Module instantiation

Achieve hierarchical design by using other modules inside one module.

Recommended format:

```plaintext
type name (.port1(w1),.port2(w2)......);
```

1. Port order doesn’t matter
2. Unused output port allowed
3. Name must be unique within module
module half_adder(a, b, x, y);
    input a, b;
    output x, y;

    assign x = a & b;
    assign y = a ^ b;
endmodule

module full_adder(a, b, ci, s, co);
    input a, b, ci;
    output s, co;
    wire g, p, pc;

    half_adder h1(.a(a), .b(b),
                .x(g), .y(p));
    half_adder h2(.a(p), .b(ci),
                .x(pc), .y(s));
    assign co = pc | g;
endmodule
Testbench

Testing is important and engineers spend more time testing for complex digital circuits.

Procedures for creating testbench:
1. Instantiate unit under test (uut)
2. Provide input for uut: clk, rst, data...
3. Simulate and verify behavior
initial begin
  // Initialize Inputs
  clk = 0;
  rst = 0;
  // Wait 100 ns for global reset to finish
  #100;
  // Add stimulus here
  rst = 1;
  #20 rst = 0;
  // add verification routine here
  #1000 $finish;
end

Initial block starts execution at time 0.

#n : wait n time units
$finish: stop simulation
module ALU #(parameter W=4)(a, b, f, z);
input [W-1:0] a, b;
input [1:0] f;
output reg [W-1:0] z;

always @(*)
  case (f)
    0: z = a + b;
    1: z = a * b;
    2: z = a & b;
    3: z = a | b;
    default: z = {(W){1'bx}};
  endcase
endmodule
Instantiation of Parameterized Module:

Default parameter $W=4$. Instantiate a 16-bit ALU:

Format:

type #(params) name (ports);

Example:

```
ALU #(W(16)) alu16 (.a(a), .b(b), .f(f), .z(z));
```
Preparation Tasks

- Read *An Old Tutorial* on the course web
  - *Ignore the device parameters for Spartan-3*
  - *Ignore contents beyond Create Timing Constraints*
- Create projects, write code and simulate:
  - 2:1 mux
  - mux2( input din0, din1, sel, output out);
  - 8 bit counter
  - counter( output [7:0] out, input clk, ena, rst);
  - 16-4 encoder
  - encoder ( input [15:0] in, input enable, output [3:0] out);
References

1. MIT 6.111 course notes,
2. Verilog in one day,
http://www.asic-world.com/verilog/verilog_one_day.html
3. Wire vs. Reg,
http://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf
4. Sunburst Design (Read on assignment operators)
http://www.sunburst-design.com/papers/
5. ISE Quick Start Guide