Digital Circuits Review

**Combinational Circuits:**

- No memory
- Output is a function of input only

**Sequential Circuits:**

- Memory, it has a “state”
- Output is a function of input and current state
- Next state is dependent on input and current state

---

**Combinational Circuits**

\[ f(x) = f( ) \]

\[ y = f(x) \]

Different ways to describe a function

<table>
<thead>
<tr>
<th>Table</th>
<th>Curve</th>
<th>Equation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x ]</td>
<td>[ f(x) ]</td>
<td>[ f(x) = \sin(x)(x-3) ]</td>
<td>if ( x &lt; 0 ) then ( f(x) = 3 ) else ( f(x) = x )</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Truth Table* ✔️ *Timing Diagram* ✔️ *Boolean Expressions* ✔️ *Hardware Description Languages* ✔️
Hardware Description Languages

Verilog HDL is a language similar to C that can be used to describe digital circuits.

With the code you can

1. Simulate the circuit
2. Synthesize (compile) to configure hardware exactly as specified by the code

VHDL is another language (we won’t get into). It’s like ADA (similar to Pascal).

Some Verilog Components

Modules: built of

- primitive operations, e.g., logical ones
- variables, e.g., “wires”
- other modules --> hierarchical

Diagram:

- Input ports
- AND
- OR
- Module Y
- Module X
- Output ports
Example

// Module X1, which does the AND operation
module X1(i0,i1,y0); // declaring module and ports
    input i0, i1; // declaring input ports
    output y0; // declaring output ports
    wire w0, w1, w2; // declaring wire variables
    assign w0 = i0; // connecting wires to ports
    assign w1 = i1;
    assign y0 = w2;
    assign w2 = w1&w0; // connecting wires to AND
endmodule

Module Components

// Comment to the end of line
// A module declaration starts with
module NAME_OF_MODULE(port1, port2, port3, ...);
// Ports are then declared as either input or output ports. The module terminates with
dendmodule
// Notice where all the semi-colons are placed.

// The next statement is called a “continuous assignment”.
assign x = expression;
// This equation is constantly updated.
// This is useful for “connecting” the right side to the left side.

// The next statement declares “wire” variables. These variables model physical wires.
wire x, y, z;
// These variable hold values “0”, “1”, “x”, or “z”. “x” = unknown, and “z” means
// high impedance (i.e., tristate).
Building Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>~</td>
</tr>
<tr>
<td>AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
</tr>
<tr>
<td>NAND</td>
<td>~&amp;</td>
</tr>
<tr>
<td>NOR</td>
<td>~</td>
</tr>
<tr>
<td>Equivalence</td>
<td>^~</td>
</tr>
</tbody>
</table>

Parenthesis may be used for precedence to build expressions

```
assign g = ~((k&y) | (a^b));
```

Ports

Ports work like wires

```
// Module X1, which does the AND operation
module X1(i0,i1,y0); // declaring module and ports
input i0, i1; // declaring input ports
output y0; // declaring output ports
assign y0 = i1&i0;
endmodule
```
Example

Hierarchical Design
Let’s build modules to implement AND and OR

```
module AND(in0, in1, out);
    input in0, in1;
    output out;
    assign out = in0&in1;
endmodule

module OR(in0, in1, out);
    input in0, in1;
    output out;
    assign out = in0|in1;
endmodule
```
Hierarchical Design

```verilog
module ExclusiveOR(in0, in1, out);
    input in0, in1;
    output out;
    wire wire1, wire2, wire3, wire4;
    assign wire1 = ~in0;
    assign wire2 = ~in1;
    AND and1(wire1, in1, wire3);    // This is called an “instantiation” since
    // it creates an instance of the circuit AND
    AND and2(in0, wire2, wire4);
    OR or1(wire3, wire4, out);
endmodule
```

Comment: Instances of modules and devices run in parallel.

Bit Arrays

Variables and ports can be declared with different sizes.

```verilog
wire [0:3] w1;     // This corresponds to 4 bit array (w1[0], w1[1], w1[2], w1[3]).
wire [3:0] w2;         // This corresponds to 4 bit array (w2[3], w2[2], w2[1], w2[0]).
assign w1 = 0;     // This is equivalent to
    // assign w1[0] = 0;
    // assign w1[1] = 0;
    // assign w1[2] = 0;
    // assign w1[3] = 0;

// w1 and w2 are basically a 4-bit variables. We can assign them constants
// using a variety of notations
assign w2 = 5;       // decimal notation -- negative #s are okay (2s complement)
assign w2 = 'b0101;  // binary
assign w2 = 'h5;     // hexadecimal
assign w2 = 'o5;     // octal
```
Useful Operations

// Example bit-wise AND operation
wire [2:0] u, v, y;
assign y = u & v;

// Example unary reduction AND operation
wire [2:0] u;
wire y;
assign y = &u;

Other bit-wise operations
Operator | Name
----------|-----------------
~          | Bitwise negation
&          | Bitwise AND
|           | Bitwise OR
^          | Bitwise XOR
~&         | Bitwise NAND
~|         | Bitwise NOR

Other unary reduction operators
Operator | Name
----------|-----------------
&          | AND reduction
|          | OR reduction
^          | XOR reduction
~&         | NAND reduction
~|         | NOR reduction

Simple Example

module Gate(in,y,out);
input [3:0] in;
input y;
output [3:0] out;
assign out[3] = in[3]&y;
assign out[2] = in[2]&y;
assign out[1] = in[1]&y;
assign out[0] = in[0]&y;
endmodule

module Gate(in,y,out);
input [3:0] in;
input y;
output [3:0] out;
assign out = in&(y,y,y,y);
endmodule

{ } --- concatenation operation. Items to concatenate are separated by commas. The items can be of various sizes, and there can be repeats.
Sequential Circuits

Circuits with “memory”. They have “states”.

Output = function of input and current state
Next state = function of input and current state

Synchronous sequential circuits
Synchronized with respect to a “clock” signal.
Synchronized with clock edges (transitions)
We’ll assume that they’re synchronized with positive edges only (though in practice this is not necessarily true).
Then state changes only at positive clock edges.

D Flip flop

Description: Q = state
On positive clock edge, Q = D.

State

Label
(output)

Transition
Condition

E Notes G. Sasaki, Univ. Hawaii 15

E Notes G. Sasaki, Univ. Hawaii 16
Verilog D Flip Flop

```verilog
module DFF(d,q,clock);
  input d;
  input clock;
  output q;
  reg state;  // register variable
  always (@posedge clock)  state = d;  // update state
  assign q = state;      // output state
endmodule
```

Verilog variables:
- wires “wire”
- registers “reg”

Wires model physical wires
-- no memory

Register variables store their value,
so they are used to model physical
memory elements. They are more
like C variables than “wires”.

Comment for this example:
always and assign
are “running” in parallel

always

```verilog
// Continuous assignment statements are used to update wire variables
wire u, v, w;
assign u = v|w;

// always is a procedural assignment. Procedural assignment statements are used to
// model sequential circuits. Thus, they are used to update register variables
reg u;
wire v, w;
always u = v|w;

// A common form of always is
always <register variable> = <expression>

// Then the register variable is constantly being updated (i.e., always being update)
Next we'll see some other forms of always that are useful.
```
Other forms of always

// The statement is always being executed
always statement;

// The statement is updated whenever the expression changes value
always @(expression) statement;

// The statement is updated whenever the expression goes from 0 to 1
always @(posedge expression) statement;

module DFF(d, q, clock);
  input d;
  input clock;
  output q;
  reg state;
  always @(posedge clock) state = d;
  assign q = state;
endmodule

Output Ports

// You can make output ports behave like register variables
// (or reg variables are directly connected to the outputs)
module DFF(d, q, clock);
  input d;
  input clock;
  output q;
  reg q;

  always @(posedge clock) q = d;
endmodule
**T Flip Flop**

Upon positive clock edge,
- \( Q = Q \) if \( T = 0 \)
- \( Q = Q' \) if \( T = 1 \)

Hint: Use Exclusive-Or

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

```verilog
// module for a T flip flop
module TFF(T, Q, clock);

// XOR truth table
always @(posedge clock) Q = T * Q + \!T * \!Q;
endmodule
```

**Size of Register Variables**

Both register and wire variables can have different sizes.

```verilog
module Clocked_Register(d, q, clock);
    input [3:0] d;
    input clock;
    output [3:0] q;
    reg [3:0] q;
    always @(posedge clock) q = d;
endmodule
```
Sequential Circuits

module Counter(Y,clock);
  input clock;
  output [1:0] Y;
  reg [1:0] Y;
  always @(posedge clock)
  case (Y):
    0:  Y = 1;
    1:  Y = 2;
    2:  Y = 3;
    3:  Y = 0;
  endcase
endmodule

always @(posedge clock)
  procedural assignment (e.g., case-statement, y = x, etc.)
  begin
    y = x & w;
    case (y)
      0:  x = 1;
      1:  y = 1;
    endcase
    w = y ^ w;
  end

Executed in sequence, not in parallel
If-Else

```verilog
module X(A,Y,clock);
  input [1:0] A;
  input clock
  output [1:0] Y;
  reg [1:0] Y;
  always @(posedge clock)
  begin
    if (Y == 0 && ((A == 0)||(A == 3))) Y = 1;
    else if ((Y==0) && ((A==1)||(A=2)) Y = 2;
    else if (Y==1) Y = 2;
    else if (Y == 2)  Y = 3;
    else if ((Y == 3) && (A[1] == 0)) Y = 1;
    end
endmodule
```

Mealy Machine

Any Synchronous Sequential Circuit can be modeled as a Mealy Circuit

State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>A=00,11</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>A=01,10</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>A=0x</td>
<td>S3</td>
</tr>
<tr>
<td>S3</td>
<td>A=1x</td>
<td>S0</td>
</tr>
</tbody>
</table>

Truth table for combinational circuit if state register is an array of D flip flops

Can cause problems
Mealy Machine Problem

Cycle of combinational circuits
Feedback loop
Possibly unstable

Moore Machine

Special case of a Mealy Machine and safer to use

Output is dependent only on the current state, and not on the current input
Read Only Memory (ROM)

Small combinational circuits can be realized by ROM

ROM is memory
you can read
you cannot write to
you can program

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>001</td>
<td>00000010</td>
</tr>
<tr>
<td>010</td>
<td>00000100</td>
</tr>
<tr>
<td>011</td>
<td>00001000</td>
</tr>
<tr>
<td>100</td>
<td>00100000</td>
</tr>
<tr>
<td>101</td>
<td>01000000</td>
</tr>
<tr>
<td>110</td>
<td>10000000</td>
</tr>
<tr>
<td>111</td>
<td>10000000</td>
</tr>
</tbody>
</table>

ROM outputs the contents of the memory cell of address A.

This is the truth table of the ROM

Verilog Implementation

module ROM(A,D);
input [2:0] A;
output [7:0] D;
reg [7:0] D;
always @(A)
begin
  case(A)
    0: D = 'b00000001;
    1: D = 'b00000010;
    2: D = 'b00000100;
    ...
    7: D = 'b10000000;
  endcase
endmodule

Notice that in this case, “always” is used as a combinational circuit.

Whenever A changes value, the output D is updated.

Notice D is a reg. variable because it needs to hold its value when A is constant.
Example Design From State Diagram

Input/Output: A/C

State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input A</th>
<th>Next State</th>
<th>Output C</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>1</td>
</tr>
</tbody>
</table>

Example Design From State Diagram

```vhdl
// state machine
module StateMachine(A, C, clock);
    input A, clock;
    output C;
    reg C;
    reg [1:0] state;

    always @(posedge clock)
        case ({state,A}):
            0: state = 'b01;
            1: ....
            etc.
        endcase

    always @(state or A)
        case ({state,A}):
            0: C = 1;
            1: ....
            etc.
        endcase
endmodule
```