Synchronous Digital Design

Combinational Logic

Sequential Logic

Summary of Modeling Styles
Why HDLs?

1970s: SPICE transistor-level netlists

An XOR built from four NAND gates

```
.MODEL P PMOS
.MODEL N NMOS

.SUBCKT NAND A B Y Vdd Vss
M1 Y A Vdd Vdd P
M2 Y B Vdd Vdd P
M3 Y A X Vss N
M4 X B Vss Vss N
.ENDS

X1 A B I1 Vdd 0 NAND
X2 A I1 I2 Vdd 0 NAND
X3 B I1 I3 Vdd 0 NAND
X4 I2 I3 Y Vdd 0 NAND
```
Why HDLs?

1980s: Graphical schematic capture programs
Why HDLs?

1990s: HDLs and Logic Synthesis

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
  port(A: in unsigned(1 downto 0);
       B: in unsigned(1 downto 0);
       Sel: in unsigned(1 downto 0);
       Res: out unsigned(1 downto 0));
end ALU;

architecture behv of ALU is begin
  process (A,B,Sel) begin
    case Sel is
      when "00" => Res <= A + B;
      when "01" => Res <= A + (not B) + 1;
      when "10" => Res <= A and B;
      when "11" => Res <= A or B;
      when others => Res <= "XX";
    end case;
  end process;
end behv;
Separate but Equal: Verilog and VHDL

Verilog: More succinct, really messy
VHDL: Verbose, overly flexible, fairly messy
Part of languages people actually use identical
Every synthesis system supports both
SystemVerilog a newer version. Supports many more features.
Synchronous Digital Design
The Synchronous Digital Logic Paradigm

Gates and D flip-flops only

No level-sensitive latches

All flip-flops driven by the same clock

No other clock signals

Every cyclic path contains at least one flip-flop

No combinational loops
$t_c$: Clock period. E.g., 10 ns for a 100 MHz clock
Timing in Synchronous Circuits

Sufficient Hold Time?

\[ t_p(\text{min}, \text{FF}) \leq t_p(\text{min}, \text{CL}) \]

Hold time constraint: how soon after the clock edge can D start changing? Min. FF delay + min. logic delay
Timing in Synchronous Circuits

Setup time constraint: when before the clock edge is D guaranteed stable?
Max. FF delay + max. logic delay
Combinational Logic
Systems are built from modules.

```
// Full adder
module full_adder(input logic a, b, c, output logic sum, carry);

assign sum = a ^ b ^ c;
assign carry = a & b | a & c | b & c;
endmodule
```

“Continuous assignment” expresses combinational logic.

![Diagram of a full adder circuit]
module gates(input logic [3:0] a, b, 
               output logic [3:0] y1, y2, y3, 
                               y4, y5);

    /* Five groups of two-input logic gates   
       acting on 4-bit busses */
    assign y1 = a & b;       // AND
    assign y2 = a | b;       // OR
    assign y3 = a ^ b;       // XOR
    assign y4 = ~(a & b);    // NAND
    assign y5 = ~(a | b);    // NOR

endmodule
Reduction AND Operator

```verilog
module and8(input logic [7:0] a, 
            output logic y);

assign y = &a; // Reduction AND

// Equivalent to

// Also ~|a NAND
// |a OR
// ~|a NOR
// ^a XOR
// ~^a XNOR

endmodule
```
module mux2(input logic [3:0] d0, d1, 
            input logic s, 
            output logic [3:0] y);

    // Array of two-input muxes

    assign y = s ? d1 : d0;
endmodule
Operators in Precedence Order

- `!c` - `c` - reduction AND, NAND
- `|c` - `^c` - OR, NOR, XOR, XNOR

- `a * b` - `a / b` - `a % b` - Multiply, Divide, Modulus
- `a + b` - `a - b` - Add, Subtract
- `a << b` - `a >> b` - Logical Shift
- `a <<< b` - `a >>> b` - Arithmetic Shift
- `a<b` - `a<=b` - `a>b` - `a>=b` - Relational
- `a == b` - `a != b` - Equality
- `a & b` - `a ^& b` - AND
- `a ^ b` - `a ~^ b` - XOR, XNOR
- `a | b` - OR
- `a ? b : c` - Conditional
- `{a,b,c,d,r{e}}` - Concatenation and Replication
An XOR Built Hierarchically

```verilog
module mynand2(input logic a, b, output logic y);
    assign y = ~(a & b);
endmodule

module myxor2(input logic a, b, output logic y);
    logic abn, aa, bb;
    mynand2 n1(a, b, abn), n2(a, abn, aa), n3(abn, b, bb), n4(aa, bb, y);
endmodule
```

---

- Declare internal wires
- `n1`: A `mynand2` connected to `a`, `b`, and `abn`
Verilog Numbers

16’h8_0F

Number of Bits
Base: b, o, d, or h
Value:
- are ignored
Zero-padded
Can include X and Z

4’b1010 = 4’o12 = 4’d10 = 4’ha
16’h4840 = 16’b 100_1000_0100_0000
A Decimal-to-Seven-Segment Decoder

```verilog
module dec7seg(input logic [3:0] a, output logic [6:0] y);

always_comb
    case (a)
        4'd0: y = 7'b111_1110;
        4'd1: y = 7'b011_0000;
        4'd2: y = 7'b110_1101;
        4'd3: y = 7'b111_1001;
        4'd4: y = 7'b011_0011;
        4'd5: y = 7'b101_1011;
        4'd6: y = 7'b101_1111;
        4'd7: y = 7'b111_0000;
        4'd8: y = 7'b111_1111;
        4'd9: y = 7'b111_0011;
        default: y = 7'b000_0000;
    endcase
endmodule
```
Imperative Combinational Logic

module comb1(
    input logic [3:0] a, b,
    input logic s,
    output logic [3:0] y);

    always_comb
        if (s)
            y = a + b;
        else
            y = a & b;
endmodule

Both a + b and a & b computed, mux selects the result.
Imperative Combinational Logic

```verilog
module comb2(
    input logic [3:0] a, b,
    input logic s, t,
    output logic [3:0] y);

    always_comb
        if (s)
            y = a + b;
        else if (t)
            y = a & b;
        else
            y = a | b;

endmodule
```

All three expressions computed in parallel. Cascaded muxes implement priority (s over t).

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>a + b</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
</tbody>
</table>
Imperative Combinational Logic

module comb3(
    input logic [3:0] a, b,
    input logic s, t,
    output logic [3:0] y, z);

    always_comb begin
        z = 4'b0;
        if (s) begin
            y = a + b;
            z = a - b;
        end else if (t) begin
            y = a & b;
            z = a + b;
        end else
            y = a | b;
    end
endmodule

Separate mux cascades for y and z.
One copy of a + b.
An Address Decoder

```verilog
module adecode(input logic [15:0] address,
               output logic RAM, ROM,
               output logic VIDEO, IO);

always_comb begin
    {RAM, ROM, VIDEO, IO} = 4'b 0;
    if (address[15])
        RAM = 1;
    else if (address[14:13] == 2'b 00)
        VIDEO = 1;
    else if (address[14:12] == 3'b 101)
        IO = 1;
    else if (address[14:13] == 2'b 11)
        ROM = 1;
end
endmodule
```

Omitting defaults for RAM, etc. will give “construct does not infer purely combinational logic.”
Sequential Logic
A D-Flip-Flop

module mydff(input logic clk, input logic d, output logic q);

always_ff @ (posedge clk)
q <= d;
endmodule

always_ff introduces sequential logic

Copy d to q

Non-blocking assignment:
“just after” the rising edge

Triggered by the rising edge of clk
A Four-Bit Binary Counter

```verilog
module count4(input logic clk, output logic [3:0] count);
always_ff @(posedge clk)
  count <= count + 4'd 1;
endmodule
```

Width optional but good style
A Decimal Counter with Reset, Hold, and Load

module dec_counter(input logic clk, 
                   input logic reset, hold, load, 
                   input logic [3:0] d, 
                   output logic [3:0] count);

always_ff @(posedge clk)
  if (reset) count <= 4'd 0;
  else if (load) count <= d;
  else if (~hold)
    if (count == 4'd 9) count <= 4'd 0;
    else count <= count + 4'd 1;
endmodule
The Moore Form:

Outputs are a function of *only* the current state.
The Mealy Form:

Outputs may be a function of both the current state and the inputs.

A mnemonic: *Moore* machines often need *more* states.
module moore_tlc(input logic clk, reset,
                   input logic advance,
                   output logic red, yellow, green);

enum logic [2:0] {R, Y, G} state; // Symbolic state names

always_ff @(posedge clk) // Moore-style next-state logic
  if (reset) state <= R;
  else case (state)
    R: if (advance) state <= G;
    G: if (advance) state <= Y;
    Y: if (advance) state <= R;
    default: state <= R;
  endcase

assign red = state == R; // Combinational output logic
assign yellow = state == Y; // separated from next-state logic
assign green = state == G;

endmodule
module mealy_tlc( 
    input logic clk, reset, 
    input logic advance, 
    output logic red, yellow, green); 

typedef enum logic [2:0] {R, Y, G} state_t; 
state_t state, next_state; 
always_ff @(posedge clk) 
    state <= next_state; 
always_comb begin // Mealy-style next state and output logic 
    {red, yellow, green} = 3'b0; // Default: all off and 
    next_state = state; // hold state 
    if (reset) next_state = R; 
    else case (state) 
        R: begin red = 1; if (advance) next_state = G; end 
        G: begin green = 1; if (advance) next_state = Y; end 
        Y: begin yellow = 1; if (advance) next_state = R; end 
        default: next_state = R; 
    endcase 
end 
endmodule
Blocking vs. Nonblocking assignment

module nonblock(input clk, input logic a, output logic d);
logic b, c;
always_ff @(posedge clk)
begin
  Nonblocking assignment:
  b <= a;
  c <= b;
  d <= c;
end
endmodule

module blocking(input clk, input logic a, output logic d);
logic b, c;
always_ff @(posedge clk)
begin
  Blocking assignment:
  b = a;
  c = b;
  d = c;
end
endmodule
Summary of Modeling Styles
module styles_tlc(input logic clk, reset,
            input logic advance,
            output logic red, yellow, green);

enum logic [2:0] {R, Y, G} state;

always_ff @ (posedge clk) // Imperative sequential
    if (reset) state <= R; // Non-blocking assignment
    else case (state) // Case
        R: if (advance) state <= G; // If-else
        G: if (advance) state <= Y;
        Y: if (advance) state <= R;
        default: state <= R;
    endcase

always_comb begin // Imperative combinational
    {red, yellow} = 2'b 0; // Blocking assignment
    if (state == R) red = 1; // If-else
    case (state) // Case
        Y: yellow = 1;
        default: ;
    endcase;
end

assign green = state == G; // Cont. assign. (comb)
endmodule