Chapter-6

Samir Palnitkar
**Dataflow modeling**

- Gate level modeling works well for small circuits.
- As gate densities on chip increasing rapidly dataflow modeling has become very important.
- Dataflow: In terms of the data flow between registers and how a designing processes data rather than instantiation of individual gates.
Continuous assignment

- Most basic statement in dataflow modeling
- Drives a value onto a net
- Nets: connections between hardware elements
  - Declared with wire
  - Default value z
  - Get output of drivers (no value = z)
  - Class net
    - Wire, wand, wor, tri, triand, trior, trireg
- Continuous assignment replaces gates
Exercise 6.1 : examples of continuous assignment

//Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

//Continuous assign for vector nets. addr is a 16-bit net
//addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

//Concatenation. Left-hand side is a concatenation of a scalar net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
characteristics

- Left hand side must always be a scalar, vector, net or a concatenation of scalar and vector nets
- It can not be scalar or vector register
- Always active
- Evaluated as soon as a right hand side value changes
- RHS can be registers, nets, or function calls
- Delay is used to decide when to assign the evaluated value to LHS
Implicit continuous assignment

wire out;
assign out = in1 & in2;

Can be replaced by…..

wire out = in1 & in2;
Delays

► Regular assignment delay
assign #10 out = in1 & in2

// any change in values of in1 & in2 will result in a delay of 10 time units before re-computation. If values change before 10 time units, the values of in1 and in2 at time of re-computation are considered.
A pulse of width less than the specified assignment delay is not propagated
Delays (continue..)

► Implicit continuous assignment delay

wire out;
Assign #10 out = in1 & in2;

Can be replaced by..

wire #10 out = in1 & in2;
Delays (continue..)

►► Net declaration delay

wire out;
Assign #10 out = in1 & in2;

Can be replaced by..

Wire #10 out;
Assign out = in1 & in2;
Expressions, Operators and Operands

- Operands: Nets, Registers, Vectors, Integer, Real and Time register data types

- Value set: 0, 1, x, z
- **Registers**: data storage
- **Nets**: connection between hardware elements
- **Vectors**: e.g., wire [7:0] bus;
  
  reg [0:4] data_reg;
- **Time**: e.g., time save_sim_time;
  
  save_sim_time = $time;
- **Arrays**: reg, integer, time
  
  e.g., integer count [0:7];
  
  reg bool [31:0]
  
  reg [4:0] port_id [0:7];
  
  e.g., bool[5] \( \rightarrow \) 5th bit of count
  
  port_id[3] \( \rightarrow \) 3rd element of port_id
- **Memories**: array of registers
  
  e.g., Reg [7:0] membyte [0:1023]
  
  //1K 8-bit words
- **Parameter**: port_id = 5
Operator types

- Arithmetic
- Logical
- Relational
- Equality
- Bitwise
- reduction
- Shift
- Concatenation
- Replication
- conditional
Arithmetic

► Binary: * , / , + , - , %
  - Result is “x” if any of the two operands contains an x

► Unary operators:
  - + (positive) or – (negative) sign
  - Higher precedence

► Negative numbers are represented in 2’s complement internally in Verilog
  - Use negative numbers of type integer/real in expressions

► -10/5 , -d’10/5 = (2^32-10)/5
Logical

- **Binary**: `&&, ||`
- **Unary**: `!`
- Evaluates to a 1-bit (0, 1 or x)
  - Operand = 0 → logical 0
  - Operand ≠ 0 → logical 1
  - Operand with x’s or z’s → x (false)
Relational

►► >, <, <=, >=

►► Expression returns 0 or 1 or x

►► x’s or z’s in the operand → result unknown
Equality

- `==`, `!=` → 0, 1, x
  - Return x if any of the two operands include x’s or z’s

- `===`, `!==` → 0, 1
  - Can compare x and z values
Bitwise operators

- Yield a bit by bit value
- ~, |, &, ^(xor), ^~ or ~^(xnor)
- Extends the shorter operand with 0’s
- Z is treated as an x
Reduction

- &, ~&, |, ~|
- ^, ~^,

- Take on eoperand

- 1_bit result
Shift operator

►► ►► shift right

►► << => shift left

►► Digits are filled with 0’s
Concatenation operator

- {...}

- Operands must be sized

- \{op1, op2, op3, ..., opN}\}
  - Where opi : scalar nets or registers, vector nets or registers, bit_select, part_select, or sized constants
Replication

- $A = 1'b1$
- $\{4\{A\}\} \rightarrow 4'b1111;$
Conditional operator

- `<Condition_expr> ? True_expr : False_expr;`

- Used in data flow to model conditional statements

- e.g.
  - assign addr_bus = drive_enable ? Addr_out : 32’bz;
  - assign out = control ? in1 : in0;

Nested conditional operator

- assign out = (a==3) ? (control ? x : y) : (control ? m : n);
Example 6.2 : 4-to-1 Multiplexer, Using Logic Equations

// 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

assign out = (~s1 & ~s0 & i0) |
    (~s1 & s0 & i1) |
    (s1 & ~s0 & i2) |
    (s1 & s0 & i3) ;

endmodule
Example 6.2 : 4-to-1 Multiplexer, Using Logic Equations Continue……

module stimulus; // Define the stimulus module (no ports)
reg IN0, IN1, IN2, IN3; // Declare variables to be connected to inputs
reg S1, S0; // Declare output wire
wire OUTPUT; // Declare output wire

// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);

// Stimulate the inputs
initial
begin
    IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0; // set input lines
        #1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n", IN0, IN1, IN2, IN3);
    S1 = 0; S0 = 0; // choose IN0
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
    S1 = 0; S0 = 1; // choose IN1
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
    S1 = 1; S0 = 0; // choose IN2
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
    S1 = 1; S0 = 1; // choose IN3
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end
endmodule
Example 6-3  4-to-1 Multiplexer, Using Conditional Operators

// 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator
assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);

endmodule
Example 6-4  4-bit Full Adder, Using Dataflow Operators

// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;

endmodule
Example 6-4  4-bit Full Adder, Using Dataflow Operators Cont

// Define the stimulus (top level module)
module stimulus;

// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);

// Setup the monitoring for the signal values
initial begin
    $monitor($time, " A= %b, B=%b, C_IN= %b,, C_OUT= %b, SUM= %b\n", A, B, C_IN, C_OUT, SUM);
end

// Stimulate inputs
initial begin
    A = 4'd0; B = 4'd0; C_IN = 1'b0;
    #5 A = 4'd3; B = 4'd4;
    #5 A = 4'd2; B = 4'd5;
    #5 A = 4'd9; B = 4'd9;
    #5 A = 4'd10; B = 4'd15;
    #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end
endmodule
Example 6-5  4-bit Full Adder With Carry Lookahead

module fulladd4(sum, c_out, a, b, c_in);

    // Inputs and outputs
    output [3:0] sum;
    output c_out;
    input [3:0] a,b;
    input c_in;

    // Internal wires
    wire p0,g0, p1,g1, p2,g2, p3,g3;
    wire c4, c3, c2, c1;

    // compute the p for each stage
    assign p0 = a[0] ^ b[0],
            p1 = a[1] ^ b[1],
            p2 = a[2] ^ b[2],
            p3 = a[3] ^ b[3];

    // compute the g for each stage
    assign g0 = a[0] & b[0],
            g1 = a[1] & b[1],
            g2 = a[2] & b[2],
            g3 = a[3] & b[3];

continue…….
Example 6-5 4-bit Full Adder With Carry Lookahead (continue..)

// compute the carry for each stage
// Note that c_in is equivalent to c0 in the arithmetic equation for carry lookahead computation
assign c1 = g0 | (p0 & c_in),
    c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
    c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
    c4 = g3 | (p3 & g2) | (p3 & p2 & g1)
    | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c_in);

// Compute Sum
assign sum[0] = p0 ^ c_in,
    sum[1] = p1 ^ c1,
    sum[2] = p2 ^ c2,
    sum[3] = p3 ^ c3;

// Assign carry output
assign c_out = c4;
endmodule
Example 6-6 Verilog Code for Ripple Counter

// Ripple counter
module counter(Q, clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_ff tff0(Q[0], clock, clear);
T_ff tff1(Q[1], Q[0], clear);
T_ff tff2(Q[2], Q[1], clear);
T_ff tff3(Q[3], Q[2], clear);

endmodule
Example 6-7 Verilog Code for T-flipflop

// Edge triggered T-flipflop. Toggles every clock cycle.
module T_ff(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Empty port.
edge_dff ff1(q, ~q, clk, clear);

endmodule
Example 6-8 Verilog Code for Edge-Triggered D-flipflop

// Edge triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q, qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar, cbar;

// Data flow statements
// Create a complement of signal clear
assign cbar = ~clear;

// Input latches
assign sbar = ~(rbar & s),
    s = ~(sbar & cbar & ~clk),
    r = ~(rbar & ~clk & s),
    rbar = ~(r & cbar & d);

// Output latch
assign q = ~(s & qbar),
    qbar = ~(q & r & cbar);
endmodule
Example 6.9 Stimulus Module for Ripple Counter

```verilog
module stimulus; // Top level stimulus module

reg CLOCK, CLEAR; // Declare variables for stimulating input
wire [3:0] Q;

initial
    $monitor($time, "Count Q = %b Clear = %b", Q[3:0], CLEAR);

initial
    $gr_waves("clk", CLOCK, "Clear", CLEAR,"Q", Q[3:0],"Q0", Q[0],"Q1", Q[1],"Q2", Q[2],Q3", Q[3]);

counter c1(Q, CLOCK, CLEAR); // Instantiate the design block counter

Initial begin
    CLEAR = 1'b1;
    #34 CLEAR = 1'b0;
    #200 CLEAR = 1'b1;
    #50 CLEAR = 1'b0;
end

initial begin // Stimulate the Clear Signal
    CLOCK = 1'b0;
    forever #10 CLOCK = ~CLOCK;
end

Initial begin // Setup the clock to toggle every 10 time units
    #400 $finish;
end

endmodule
```
// Setup the clock to toggle every 10 time units
initial
begin
    CLOCK = 1'b0;
    forever #10 CLOCK = ~CLOCK;
end

// Finish the simulation at time 200
initial
begin
    #400 $finish;
end
endmodule