CSE140L: Components and Design Techniques for Digital Systems Lab

Verilog HDL

Instructor: Mohsen Imani
UC San Diego
System Tasks

• The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  – $display("The value of a is %b", a);
    • Used in procedural blocks for text output.
    • The %b is the value format (binary, in this case…)
  – $monitor
    • Similar to display, but executes every time one of its parameter changes
  – $finish;
    • Used to finish the simulation.
    • Use when your stimulus and response testing is done.
  – $stop;
    • Similar to $finish, but doesn’t exit simulation.
Blocking/Non-Blocking Assignments

- **Blocking assignments** $(X=A)$
  - completes the assignment before continuing on to next statement

- **Non-blocking assignments** $(X<=A)$
  - completes in zero time and doesn’t change the value of the target until a blocking point (delay/wait) is encountered

- **Watch out for assignments with delay!**
  - Examples follow….
Blocking Delay – Unintended Behavior

module adder_t1 (co, sum, a, b, ci);
  output [3:0] sum;
  input [3:0] a, b;
  input ci;
  reg co;
  reg [3:0] sum;

always @(a or b or ci)
  #12 {co, sum} = a + b + ci;
endmodule

Trigger the always block

Output changes only 3ns after last input change
module adder_t4 (co, sum, a, b, ci);
    output co;
    output [3:0] sum;
    input [3:0] a, b;
    input ci;

    assign #12 {co, sum} = a + b + ci;
endmodule

Assign Delay – Correct Behavior
Nonblocking delay – expected behavior

```verilog
module adder_t3 (co, sum, a, b, ci);
    output co;
    output [3:0] sum;
    input [3:0] a, b;
    input ci;
    reg co;
    reg [3:0] sum;

    always @(a or b or ci)
        {co, sum} <= #12 a + b + ci;
endmodule
```

module testbench (x, y);
  output        x, y;
  reg [1:0]      cnt;

initial begin
  cnt = 0;
  repeat (4) begin
    #10 cnt = cnt + 1;
    $display (@ time=%d, x=%b, y=%b, cnt=%b,
              $time, x, y, cnt);
    #10 $finish;
  end
end

assign x = cnt[1];
assign y = cnt[0];
endmodule

Driving a simulation through a “testbench”

2-bit vector

initial block executed only once at start of simulation

print to a console

directive to stop simulation
Delay Control

• Generation of clock and resets in testbench:

```verilog
reg rst, clk;
initial // this happens once at time zero
begin
    rst = 1'b1; // starts off as asserted at time zero
    #100; // wait for 100 time units
    rst = 1'b0; // deassert the rst signal
end
always // this repeats forever
begin
    clk = 1'b1; // starts off as high at time zero
    #25; // wait for half period
    clk = 1'b0; // clock goes low
    #25; // wait for half period
end
```
“case” Statements in Verilog

What does the following piece of code represent?

```verilog
always_comb
begin
    case ( sel_i )
        2'd0 : z_o = a_i;
        2'd1 : z_o = b_i;
        2'd2 : z_o = c_i;
        2'd3 : z_o = d_i;
        default : z_o = 1'bx;
    endcase
end
endmodule
```

always_comb implicitly assume a complete sensitivity list. It’s the same as:
always @ (*)

Useful for homework 3
Synchronous design: specify a clock

- When writing a module with a clock signal, use behavioral description with the clock in the sensitivity list.

```verilog
always @(posedge clk) begin
    // do stuff
end
```

Useful for homework 3
Generate a clock in a testbench

- When testing a module with multiple inputs, make sure you start from a known condition
- How to generate a clock signal in a testbench?

```vhdl
initial
begin
    // initialize your module inputs here
end

always
begin
    #5 clk = ! clk;
```

Useful for homework 3

\[
\frac{f}{Hz} = \frac{1}{T} = \frac{1}{1000ns} = 1000Hz
\]
CSE140L: Components and Design Techniques for Digital Systems
Circuit Delay

- Transistors have intrinsic resistance and capacitance
- Signals take time to propagate from the input to the output of a gate
- Sometimes delays are labeled as @<delay_value> in circuit drawings
1-Bit & Multi-bit Adders

**Half Adder**

\[ C_{out} = A \oplus B \]

\[ S = A \oplus B \]

**Full Adder**

\[ C_{out} = A \oplus B \oplus C_{in} \]

\[ S = A \oplus B \oplus C_{in} \]

\[ C_{out} = AB + AC_{in} + BC_{in} \]

Types of multi-bit adders

- Ripple-carry (slow)
- Carry-lookahead (faster)
- Two-level logic adder (even faster)
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow

Ripple-carry adder delay

\[ t_{\text{ripple}} = N t_{FA} \]

where \( t_{FA} \) is the delay of a full adder
Two-level Logic Adder

- No matter how many inputs you have,
  - look at the truth table,
  - convert to Kmap,
  - apply the algorithm for two-level logic minimization
- Very fast adder, but....
- Beyond 8 inputs, a shockingly large amount of gates!
  - Number of gates increases exponentially

Ripple carry adder
Carry-lookahead adder (next slide)
Two-level logic adder

FAST

COMPLEX
Carry-lookahead adders

<table>
<thead>
<tr>
<th>c4</th>
<th>c3</th>
<th>c2</th>
<th>c1</th>
<th>c0</th>
<th>Carries</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
<td></td>
<td>First operand</td>
</tr>
<tr>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
<td></td>
<td>Second operand</td>
</tr>
<tr>
<td>s3</td>
<td>s2</td>
<td>s1</td>
<td>s0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Carry-lookahead adders

- Adder with propagate (P) and generate (G) outputs:

\[
Ci+1 = Ai Bi + Ci (Ai \text{ xor} Bi)
\]

\[
Ci+1 = Gi + Ci Pi
\]

The carry at some level is equal to 1 if either the generate signal is equal to one or if the propagate and the previous carry are both 1.
**Full-Adder**

- **Two scenarios (actually three)**
  - When would we always generate a carry-out based on the $A_i$ and $B_i$ inputs alone?
  - When would we always propagate the carry?
  - When would we actually kill the carry (meaning there is guaranteed to be no carry-out)?

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_i$</th>
<th>$S$</th>
<th>$C_o$</th>
<th>Carry status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>generate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>
Carry-Lookahead Adder

- Applying these equations for a 4-bit adder we can solve for the multiple carry-in bits:

\[
\begin{align*}
C_1 &= G_0 + P_0 C_0 \\
C_2 &= G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0 \\
C_3 &= G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \\
C_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0
\end{align*}
\]
Carry-Lookahead Adder

\[ P_G = P_3 \cdot P_2 \cdot P_1 \cdot P_0 \]
\[ G_G = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 \]
\[ C_4 = G_0 + P_0 \cdot C_0 \]

Ref: Dan Earnst
Carry-Lookahead Adder

- 4 bit adders with internal carry look-ahead P and G logic
- Second level carry-lookahead unit creates the **GROUP** P and G signals
Carry-lookahead adders

Example: 4-bit CLA adder

\[ c_1 = G_0 + P_0 c_0 \]
\[ c_2 = G_1 + P_1 c_1 \]
\[ c_3 = G_2 + P_2 c_2 \]
\[ c_4 = G_3 + P_3 c_3 \]

Gi = ai bi \hspace{1cm} \text{generate} \\
Pi = ai \text{xor} bi \hspace{1cm} \text{propagate} \\

All “G” and “P” are immediately available, but “c” are not.

So you need to make substitutions:

\[ c_1 = G_0 + P_0 c_0 \]
\[ c_2 = G_1 + P_1 (G_0 + P_0 c_0) = G_1 + P_1G_0 + P_1P_0c_0 \]
\[ c_3 = G_2 + P_2 c_2 = (\text{derive at home}) \]
\[ c_4 = G_3 + P_3 c_3 = (\text{derive at home}) \]