18-360
Intro to Computer-Aided Digital Design
ECE Department

Carnegie Mellon

With corrections

Out: 1/29/04 | Homework 2 | Due: 2/10/04

We'll discuss the solutions to these problems at the beginning of class on the due date. Bring your solutions then. Solutions will be handed out, so we can't accept late assignments.

If a problem doesn't specifically state "simulate this," you do not need to turn in a simulation. However you may want to run one to be sure of your answer. If a simulation is requested, you should turn in your source description and a printout of your results showing that your description was successfully executed (i.e., time/date, "no errors", results…). It's best to use the Cadence simulator.

Symplify can be found at: /afs/ece/support/synplicity/sun4_55/image/usr/local/synplcty/. You might also be able to sneak into the 240 lab during the day and do it there. But don't go near it during a scheduled lab — you'll find some nasty Sophomores with pitchforks.

1. Making latches (15%)

In 18-240 we hit you over the head to make sure that you either synthesize combinational logic or edge-triggered logic. And when there's a warning that says "latches inferred," that's bad. Well, not always! Write a synthesizable behavioral description with the following module header. When set is one, latchOut will become one. When clear is one, ***each bit of*** latchOut will become zero. When Gate is one, the output follows the input; when it is zero, the current value is held. (current being the value just before the Gate became zero.) If either set or clear are asserted along with Gate, the set or clear will override the Gate. If both set and clear are asserted, the latch will clear. Simulate and synthesize (see above). Turn in the gate level drawing from Symplify.

```
module latchInferred
      #(parameter W= 4),
      (input         gate, set, clear,
       input   [W-1:0]in,
       output  [W-1:0]latchOut);

// always always always always …
endmodule
```

Correction

2. A D-FF with force/release (10%)

This will be ugly, and it won't be synthesizable. Fill in and simulate the following module. It should be a positive edge D flip-flop but reset_L should be implemented with force/release statements. (It should work just like the regular D-FF you know.)

```
module theyForcedMeToWriteThisUglyDff (q, d, ck, reset_L);
      input    d, ck, reset_L;
      output   q;
```

3. Hierarchical Names (10%)

Here's a description with some hierarchy in it. Answer the questions in the comments. Fill out the description enough so that you can run it through the Verilog parser to show there are no errors. (It doesn't have to do anything other than parse — no functionality!)

```
module a ();
      always
               begin: d
                     begin: e
                        //Call xyzzy from here, pass s (3 lines below) to it
                     end
                     begin: f
                        reg   q, r, s;
                     end
               end
endmodule

module b ();
      reg q, u, v;
                     // Show a $display statement here that prints all
                     // the q's in module a's instantiations

      a        a0 ();
      a        a1 ();
      a        a2 ();

      function xyzzy ();
      …
      endfunction
endmodule

module top();
      b        design ();
      tb       t ();    // From inside tb, $display register q in a1
                        // From inside tb, call xyzzy, pass s in a0 to it
endmodule
```

4. Wire You Asking This? (15%)

We're often sloppy with the term wire in Verilog. Actually Verilog has several types of *nets*, a *wire* is one of them. You can actually do logic on certain types of nets. Write a description of a half-fadder that *only* uses buffers (buf) and invertors (not) — these are gate primitives — and does all of the logic on the nets. Use the module header below. Look up "wor" and "wand" in the book.

```
module wiredAdd
      (output  sum, carryout,
       input   A, B);
```

a. Write the description as described above. Simulate and show your results.

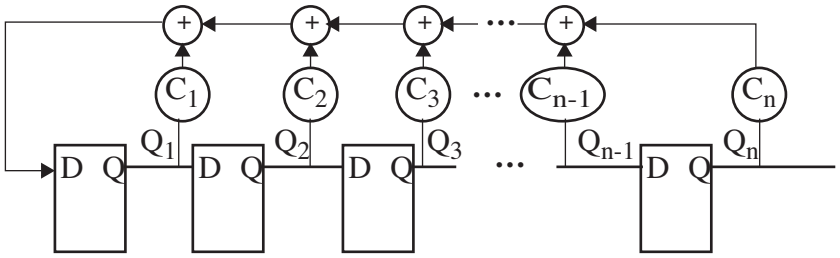
b. Change all net types to "wire" but keep everything else the same. Show the simulation results and explain the differences.

c. Will a synthesis tool crash and burn? Try synthesizing the description of part a.

5. Loopy Synthesis (20%)

In class, we discussed using a for loop when specifying circuits for synthesis. You are to write a synthesizable description for the combinational logic in a linear feedback shift register (LFSR). An LFSR has the following general form. C is an n-bit constant. If C_i=0, there is no adder in that

position. If C_i=1, then one input to the adder is Q_i. When implemented, not all adders are in the circuit, only ones where C_i is one.



Write a synthesizable description for a parameterizable version of an LFSR. Use a looping construct in the synthesizable always statement. The module will take n and C[n] as parameters, Q[n] as a module input, and produce D (the input to Q1) as its output.

This is not a sequential algorithm. Rather, the combinational logic must be specified using a looping construct. Simulate this model using the generic parameter values (n=6 and C=6'h16). Have an input that is used to initialize the value in the LFSR (it needs to be nonzero!). Define C as C[1:n] so they are number left to right as in the above diagram.

a. Write a synthesizable description using a looping construct. Simulate this model, using the generic parameter values, with a few initial values to demonstrate correctness.

b. Given a fixed n of 6, write a non-loop version of the same. i.e., the module isn't parameterized. Simulate this model with several input test vectors to demonstrate correctness

c. Synthesize and compare the resulting designs (using identical n). Pick two criteria such as the number of gates, or delay through the logic (estimated for comparison).

6. Assert Module (15%)

Write an assert module, in the style shown in lecture. There are three events, A, B, and C. Your module is to check that A and B happen (become TRUE for one clock cycle) before C. A and B can happen in any order, but they can each only happen once before C. The check will only be made at clock edges. If there are more than 20 clock edges (a parameter to them module) between the first of A or B occurring and C, report that there is a timeout error and begin looking again for A or B. Have the module print from a set of parameterized error messages if there is an error. Provide a simulation to illustrate its usage. Reset the module to looking for A or B with a reset input. You don't need to check that A, B, and C are TRUE for only one clock cycle.

```
module assert_AorBthenC
      #(// parameters…),
      (input   A, B, C, clock, reset);
…
```

7. Half-Empty Case (15%)

Write a synthesizable description of the given FSM using a one-hot encoding scheme (given). Synthesize

- using both full/parallel case attributes
- using no case attributes

Synthesize and compare/contrast the results in terms of the number of logic elements used by an Altera part.

Inputs are x1 and x0. Outputs are y1 and y0. State encoding is A: 0001, B: 0010, C: 0100, and D:1000.