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

Carnegie Mellon

Out: 2/10/04 | Homework 3 | Due: 2/19/04

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 it says "simulate," … well, you know.

1. Do as soon as possible (20%)

Assume that this represents one basic block of code — a straight line segment of code without any branches into it our out of it. It represents the body of a loop.

```
begin
  x1 = (in + 3) >> 1;
  out <= y
  yold = y;
  y = x1 + x2 + x3;
  x2 = x1;
  x3 = x2;
  delta = y - yold
  delta <= >> 1;
end
```

a. Draw a dataflow graph for this code, placing operators into steps of the graph *as soon as possible*. Operators can only have two inputs. (If there is no operation, i.e., x2 = x1, this is just a renaming of a value in the graph.) The values coming in at the top are the right-hand side values. The values coming out the bottom are the outputs and/or values that will be used in the next iteration of the loop.

b. Assume that I have one ALU in my datapath, how many states in an FSM will it take to execute this code?

c. Hmm, a shifter is really just a wiring operation. Assume that you have as many adders and subtractors as you want, but that the clock period is set so that the delay through any of these operators is about 60% of the period (i.e., there's just enough time for the operator and propagation through some wires in a period). Schedule (as soon as possible) this graph into as few states as you can — what operations are in each state?

d. If I put an @(posedge clock) around the above code, what would be the physical event(s)? Given an example of a logical event in the code.

2. Slow multipliers (20%)

Use the dataflow graph for the dif-eq solver in lecture 8 for this problem.

a. Schedule this graph under the assumptions that an adder fits into one clock period, but that a multiplier takes two (i.e., a multiplier will span two levels of the graph). Redraw the graph showing the ASAP schedule.

b. How many registers are needed for this implementation? Assume that the inputs to the multiplier have to be in registers for both clock periods.

c. Answer the above two questions for the situation where the multiplier takes two clock periods, but is pipelined, i.e., it can take a new value each clock but still takes two clocks to produce the result. (Don't count any registers that would be inside the multiplier.)

3. Does anyone know what time it is? (15%)

Cycle-accurate specification means that the state of a system is accurate at the clock cycle time — a distinction being that you don't care about the accuracy of state changes (i.e., gates/wires) between the clock cycles. But are the clocked state changes really accurate? Consider the following two versions of a system.

```
module CycleAcc (InputA, …);
    input   [12:0]    InputA;

    always @(posedge clock) begin
           …
           a <= a + InputA;
           …
    end
endmodule
```

```
module GateAcc (InputA, …);
    input [12:0]      InputA;
    wire  [12:0] aluOUT, regA;

    ALU   alu(aluOUT,regA, InputA);
    RegisterA (regA, aluOUT, clock

endmodule
```

The cycle-accurate one (module CycleAcc) shows register a being loaded at the positive edge of the clock with the sum of the current register plus an input. Assume that the input (InputA) comes from outside our system. The gate-accurate version (module GateAcc) has an ALU module that has primitive gates (with #delays too!) in it and a Register module that has stuff in it like @(posedge clock) Q <= d;. This model closely captures the real gate level model. When the clock edge occurs, the register loads its input which is the output of the ALU which is the sum of the register plus InputA. So, these are meant to model the same function — just modeled at different levels.

Two state machines are equivalent if they produce the same sequence of outputs for the same series of inputs. If we clock the two systems discussed above at the same rate and connect them to the same input, will they produce the same sequence of values in register a? What situation might cause them not to have the same sequence of inputs? No simulation or synthesis needed.

4. Not the p18240 Again … (25%)

Write a cycle accurate description of this machine, including *only* the following instructions: LDA, LDI, STA, (hmm, I guess there isn't an STI) ADD, and BRN. The machine should be equivalent to the version we used in 18240 in terms of number of states to execute each instruction and what happens in each state. Simulate your description running a very simple program. It doesn't have to do anything meaningful, you only need to show that each of these instructions is used and works.

5. State transition diagram needed (20%)

Draw a state transition diagram for the following description of a FIR filter. Essentially this filter takes in a value at the first clock. Then in the next 7 clock periods it multiplies the previously inputted values times a coefficient, sums these up, and outputs the result. Then it replaces one of the previously inputted values (the next one in sequence) and does the sum again. It just keeps doing this, so that it is always looking at the last 8 inputs, calculating this sum, and outputting it.

We're not asking you to simulate this or synthesize this to hardware. The idea is to draw the state transition diagram. "Outputs" of the FSM can be statements like "index = index + 1", which means that that operation happens in a particular state or on an arc possibly with a qualifier.

```
module FIR(clock, reset, x, y);
    input           clock, reset;
    input   [7:0]   x;
    output  [7:0]   y;

    reg     [7:0]   coef_array [7:0];
    reg     [7:0]   x_array [7:0];
    reg     [7:0]   acc, y;
    reg     [2:0]   index, start_pos; // rolls over from 7 to 0

    initial
        forever @ (negedge reset) begin
            disable ffirmain;
            start_pos = 0;
            index = 0;
        end

    always begin: FIRmain
        wait (reset);
        begin: loop1
            forever begin
                @ (posedge clock);
                if (index == start_pos) begin
                    x_array[index] = x;
                    acc = x * coef_array[index];
                    index = index + 1;
                end
                else begin
                    acc = acc + x_array[index] * coef_array[index];
                    index = index + 1;
                    if (index == start_pos) disable loop1;
                end
            end
        end
        y <= acc;
        start_pos = start_pos + 1;
        index = start_pos;
    end
endmodule
```