The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

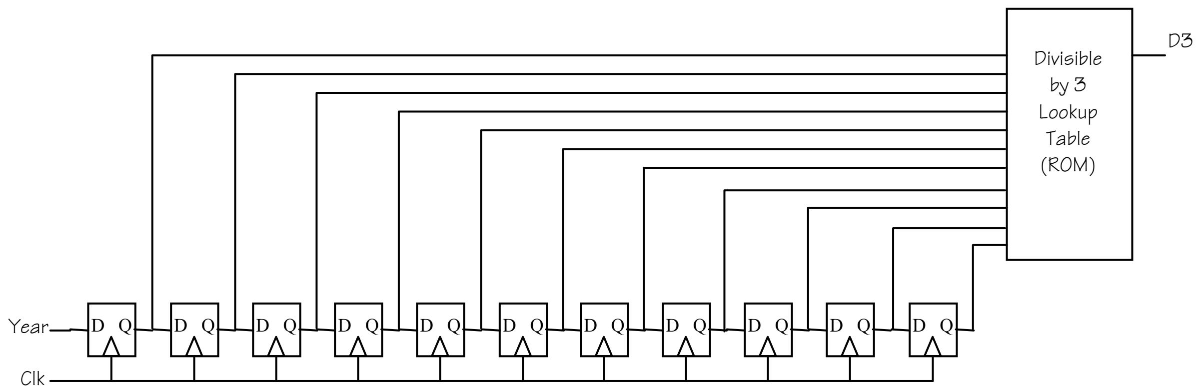
Comp 120 Computer Organization
Spring 2005

Problem Set #4
Issued Thursday, 2/17/05; Due Thursday, 2/24/05

Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

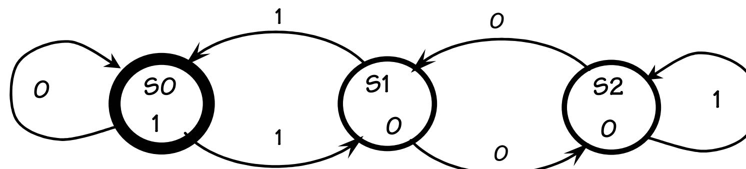
Problem 1. "Beware of the Real Y2K"

Immediately following an unusually useful Comp 120 lecture, Lee Hart races back to his part-time job at the upstart Mod-3 Calendar Corporation. Central to all products in the Mod-3's product line is a patented circuit that determines if a given calendar year (e.g. 2003) is divisible by 3. The year is entered into Mod-3's products as an unsigned integer, one bit at a time. In Mod-3's prototype system, this function was accomplished with a shift register and a look-up table using the circuit shown below.



A) Lee has been told that the contents of the look-up table have been carefully computed and programmed into a ROM. Based on the circuit shown above, how many bits are in this ROM? Is Mod-3's design a finite-state machine? **Hint:** Does the circuit's output depend solely on its input or its input and prior state? What timing characteristic must the flip flops of the shift register satisfy for correct operation?

Lee has warned the management at Mod-3 of a *real* looming Y2K problem associated with the current design, and in an effort to humor him they have assigned him the task of improving the circuit's design. Lee recalls a simple state machine specification from Comp 120. The state transition diagram of the circuit is shown below.



He recalls from lecture that this state machine also determines if a given unsigned integer input is divisible by 3, when entered *least significant bit* first.

b) Verify the operation the behavior of the state transition diagram given in class by determining its final state for each of the following input sequences following a reset
{11111010011, 11111010001, 10111010100, 11011110000, 11110101010}

Convinced that the circuit performs as specified, Lee hastily sets out to implement the state machine. He cleverly assigns the following state encodings, S0 = 10, S1 = 00, and S2 = 01. This allows the most significant bit of the state encoding to serve double duty as the divisible by three output, D3.

c) Recreate Lee's state machine using a ROM and a 2-bit state register. What is the size of the ROM. Write out a table showing the new ROM's contents.

d) Why is this new "divisible-by-3" circuit immune to the Y2K problem foreseen by Lee? Is it likely to be faster than the original design? Is it likely to be less expensive than the original (explain why)?

Problem 2. "A Next Step in Ant Evolution"

The Comp 120 ant brain discussed in lecture consisted of a simple FSM that implemented a "right antenna to the wall" algorithm for traversing a maze. This algorithm works for the simple mazes but fails if the maze contains an "island" where some inner wall of the maze is completely disconnected from the outer wall. Once the ant starts following the inner wall it will do so forever.

Your job is to help our ant step up the evolutionary ladder by designing an FSM that allows the ant to successfully navigate mazes with islands. To help in this task the ant's hardware has been upgraded; the total array of sensors and actuators at the ant's disposal now include:

L *left antenna sensor*: 1 if sensor is in contact with a wall, 0 otherwise

R *right antenna sensor*: 1 if sensor is in contact with a wall, 0 otherwise

S *crumb sensor*: 1 if the center of ant's body is over some portion of the maze that has been previously marked with a crumb left by the ant (see the M actuator).

TL *turn left actuator*: 1 causes the ant to rotate counterclockwise by 10 degrees, 0 means no left turn. Probably wouldn't be asserted in the same cycle as TR.

TR *turn right actuator*: 1 causes the ant to rotate clockwise by 10 degrees, 0 means no right turn. Probably wouldn't be asserted in the same cycle as TL.

F *step actuator*: 1 causes the ant to move forward one step, 0 means no forward motion. Can be asserted with either TL or TR, but any turns are completed before the ant moves forward. The ant will refuse to take a step if either antenna is touching a wall.

M *crumb actuator*: 1 causes the ant to mark the maze floor directly beneath the center of its body by dropping a crumb; a 0 means no crumb is dropped. The S sensor will detect these crumbs. Crumb-marking happens before any forward motion. Probably wouldn't be asserted in the same cycle as E. Multiple crumbs dropped at the same point coalesce to become one big crumb.

E *eat actuator*: 1 causes any previous crumb-marks directly beneath the center of the ant's body to be eaten; 0 means no eating takes place. Eating happens before any forward motion. Probably wouldn't be asserted in the same cycle as M.

You can attempt you design entirely on paper if you wish, but you might find that it is more fun to actually test out your design on the Roboant simulator. In order to do so, you will need to download the Roboant simulator from the links page of the course website.

http://www.unc.edu/courses/2005spring/comp/120/001/links.html

The simulator is supplied as a Java "jar" file. You will need a Java runtime system in order to use it. More than likely, you already have a Java runtime system installed on your Microsoft Windows or *nix based system. If not, Sun Microsystems provides a Java runtime environment for most common architectures. The ant simulator can be invoked by typing the following command at a shell (command window) prompt from the same directory where the jar file is located:

```
> java –cp ant.jar ant.Ant [optional filename]
```

If this does not respond by opening an ant simulator window, you need to either make sure that your Java runtime executable is configured in your current path, or you need to install a runtime system as mentioned previously.

You can supply an optional filename argument which will be loaded into the FSM editing buffer (you can also load and save FSM files from within roboant too). If no argument is supplied, the buffer is initialized with the FSM shown in lecture. After roboant starts, you'll see a window with the FSM displayed on the left and the maze displayed on the right.

The FSM display consists of the following parts:

State table. The table is organized with one state transition specified on each line. Blanks and comment lines (lines beginning with ";") are ignored. A transition specification has two parts: a pattern that will be matched against the current state and sensor readings (inputs) of the ant, and an output field that indicates the name of the ant's next state and values for the various actuator control signals that run the ant machinery. The syntax for a state transition is:

```
current_state L R S | next_state TL TR F M E
```

The states (current_state, next_state) are specified using symbolic names, which may be any sequence of letters or digits. Inputs (L, R, S) can be specified as either "0", "1" which must match exactly the value of the corresponding sensor; or "-" which indicates that any sensor value will match. Outputs (TL, TR, F, M, E) must be either "0" or "1".

"Load" button. Pressing this button will prompt for the name of a file to be read into the FSM edit buffer.

"Save" button. Pressing this button will write out the current contents of the state table to a file (roboant will prompt you for a name). You can edit this file with the editor of your choice and then reload it using the "Load" button.

The maze display consists of the following parts:

Maze select radio buttons. Select which of the five mazes the ant should try to navigate.

Maze map: Shows the current position of the ant within the maze.

Speed control. This slider controls the speed of the animation when you press the "Run" button. At the fastest speed, no status updates are performed which leads to a much faster simulation.

"Reset" button. Reset the ant to its initial state ("lost") and position.

"Step" button. Let the ant progress one state of the FSM.

"Run" button. Like "Step" except the ant will continue running the FSM until it reaches the goal square in the maze (marked with a cyan circle), the "Stop" button is pressed, or an error is detected.

"Stop" button. Stop the ant. You can proceed by pressing the "Step" or "Run" button.

"Print" button. Prints out the current state of the maze along with some statistics. Mostly useful for proving your ant has completed the course.

At the bottom of the screen is a state display showing the ant's current state and sensor readings.

The ant operates by looking for a row in the state table that matches its current state and sensor values; it will complain if no rows match or if more than one row matches. The ant's state is changed to that specified in the "next-state" field of that row and any actuator actions are performed. This sequence is repeated until the ant reaches a specially marked (with a cyan circle) goal square.

The only requirement for this problem is to devise an ant design that can solve maze "m2", and turn in a state transition table for your solution. For fun, you can try your solution on mazes "m3" or "m4". The simple marking algorithm you that you developed to solve maze "m2" may not solve these mazes!

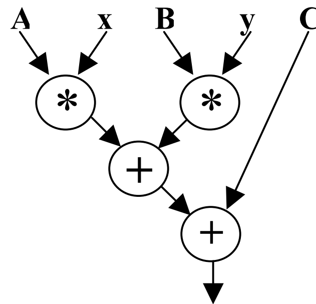
Problem #3. "To Compute a Pipelined Pixel, LEEly"

Upon completing Comp 120 Lee Hart was immediately hired by a hot new start-up company, Pixzilla, which specializes in the design of high-end computer graphics cards. On his first day of work his supervisor and company founder, Zeeb Uffer, explains to him that virtually all computer graphics functions can be computed using just one data path called a Linear Expression Evaluator, or LEE for short. A LEE simply computes the following linear function: LEE(x, y) = Ax + By + C.

The entire process of rendering a triangle can be reduced to the evaluation of a series of LEEs. For instance, whether or not a pixel (x, y) lies inside of a triangle can be determined using 3 LEEs, one for each edge. In this case, the zero-set of linear expression (where Ax + By + C = 0) describes the equation of an edge. The signs of the A, B, and C coefficients are chosen so the linear expression is positive inside of the triangle, and negative outside. Thus, a pixel inside of a triangle will give a positive result for all three edge LEEs.

Furthermore, the color of the triangle can be interpolated within the triangle using a single LEE for each primary color (i. e. red(x, y) = Ax + By + C). If the color at each vertex is known, the values of A, B and C are determined by solving a linear system using the given values of the colors and the coordinates of the three vertices. Likewise, the depth at each pixel inside a triangle can be interpolated using a LEE when the depth at each vertex is given.

Currently Pixzilla uses the following implementation of a LEE:

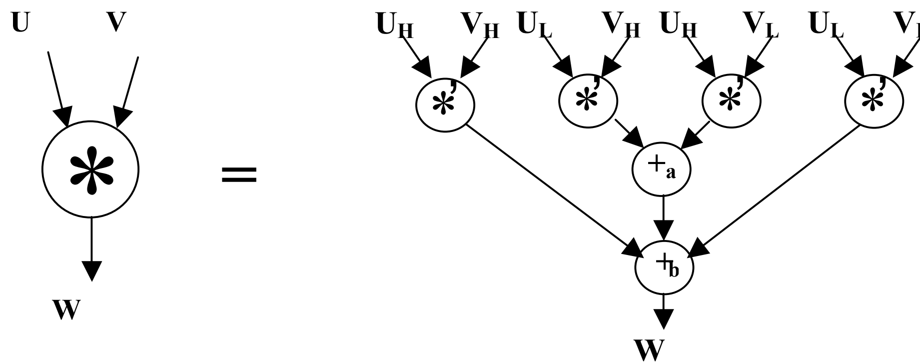


Lee's job is to improve LEE performance using the pipelining skills learned in Comp 120.

(A) Assuming that the propagation delay of the multipliers shown is 40 nS, and the propagation delay of the adders shown is 10 nS, what is the propagation delay and throughput of the initial LEE implementation?

(B) If the LEE implementation shown is pipelined using registers with 5 nS setup time and a 5 nS propagation delay, what is the maximum throughput that can be achieved? What is the minimum latency that achieves this maximum throughput?

(C) Lee thumbs through the schematic of the Pixzilla LEE implementation and realizes that the multiplier shown in the diagram above is actually implemented using four smaller multipliers as shown below:



Lee discovers that the small multipliers have a propagation delay of 25 nS, the small adder (labeled $+_a$) has a propagation delay of 5 nS, the larger adder (labeled +b) has a propagation delay of 10 nS and registers have the same timing as in part (B). With his newfound knowledge, what is the maximum throughput that can be achieved for this LEE implementation? What is the minimum latency that achieves this maximum throughput?

(D) Given that each rendered pixel requires 7 LEE evaluations, and that the average triangle turns on only 30 pixels, what is the maximum possible number of triangles per second that Pixzilla's accelerator can render?