CSE 378: Machine Organization and Assembly Language

Assignment #3: One-Dimensional Cellular Automata

(Assigned October 13, 2000, Due October 20, 2000 by 5:00)

1 Cellular Automata

You may be familiar with cellular automata from the Game of Life, created by mathematician John Conway. Conway's game is played on a square grid of cells, each of which is either empty or contains a live creature. Each cell has eight neighboring cells (the grid is assumed to wrap around on the top, bottom and sides). At time zero the grid is randomly filled with live creatures. At each time after that a new generation of creatures is created based on a number of simple rules. The particular rules for Conway's game are as follows:

1) *Loneliness*: if a creature has zero or one live neighbors it dies of loneliness.
2) *Overcrowding*: if a creature has four or more live neighbors it dies of overcrowding.
3) *Life*: if a creature has two or three neighbors it stays alive.
4) *Birth*: if an empty cell has exactly three neighbors, a new creature is born there.

From these simple rules an amazing variety of patterns and shapes can be created. Enthusiasts and researchers have named hundreds of these patterns such as "gliders", "glider guns", "pumps", "exploders", etc. If you are unfamiliar with this game (which is a very common screensaver) do a quick search on the Internet. There are dozens of web sites with interactive Java applets that let you play with Conway's game.

This version of the game can be considered a "two-dimensional" cellular automaton, as it is played on a two-dimensional grid. SPIM lacks the facilities to display such a grid on the console window (there is no way to move the cursor programmatically). So, rather than writing a program to simulate Conway's game, we will be writing a program to simulate one-dimensional cellular automata.

2 One-Dimensional Cellular Automata

One-dimensional cellular automata consist of a single row of cells arranged in a circle (so that the first and last cell in the row are neighbors). Rather than each cell having eight neighbors, as in Conway's game, they have just two (one on either side). This simplifies things considerably. You will model a one-dimensional automaton using an array with 80 elements (as there are 80 columns of text in the XSpim console). Remember that the first and last elements of the array are also neighbors, so that every cell has exactly two neighbors. You should create a function called 'simulate' that takes the following parameters:

$a0 – pointer to an array of size 80 that contains the first generation of the automata.

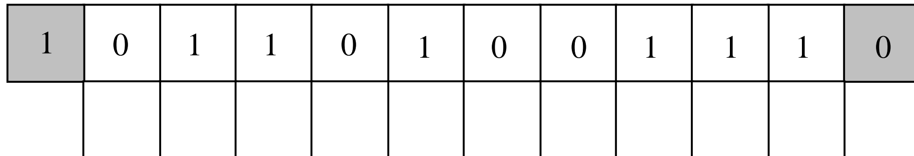
$a1 – the number of generations to simulate.
$a2 – the rule set to use (explained below in section 3).

You will simulate the specified number of generations, printing each one out to the console on its own line. When you print the array simply print a space for empty cells and an X for live cells. When the procedure is done you will have a sort of 'timeline' for the automata on the SPIM console.

3 Rules for Creating New Generations

The most important – and difficult – part of this assignment is the specification of rules that govern the creation of a new generation of cells. In Conway's game there were four basic rules that explicitly determined what the new generation looked like given the old generation. Note that the set of rules given in the first part is one of possibly millions of sets of rules that could be used (take a second to think about the various possibilities, there are literally 2^{2^9} of them). Luckily for us, in the one-dimensional case there are only 256 possible sets of rules. Rather than just using one of these rule sets, we can allow the user to specify which set to use by passing a number between 0 and 255 to our simulate function. The way this works is explained in detail below.

At each step in the simulate function you are going to have a current generation and a new, empty generation. Using the rule set specified by the user (in argument register $a2) and the current generation your function will fill in the new generation. Then you will make the new generation the current one and start over. To help illustrate this process, figure 3.1 shows a current generation filled in with some live cells and an empty, new generation. The gray cells on the far left and right of the current generation are the wrap-around values from the other side of the array (remember that each cell must have exactly two neighbors).



1	0	1	1	0	1	0	0	1	1	1	0

$a2 - 00000000 00000000 00000000 01010101

Figure 3.1: sample set of generations

Here is how the process works. Start with the first cell of the new generation and look at its ancestor – the first cell in the current generation. In this case the ancestor's value is zero and its two neighbors are both one. These three cells determine the value of the first cell in the new generation. Rather than looking at them as separate cells, treat them as a single number, in this case 101 (or 5 in decimal). Now suppose that the user specified that rule-set number 85 be used. Eighty-five is 01010101 in binary and the fifth bit (going from least significant to most) is zero. So the value of the new cell is zero. This may seem a little confusing, so allow me to clarify. The first step is to look at the

three cells above the cell you are trying to fill in. You convert the value of these three cells to a decimal number and use it as an index into the bits of the rule set number. The value of the bit at that index is the value of the cell in the new generation. Just to make sure that this is clear a few more examples are in order. For the second cell in the new generation of figure 3.1, the three cells above it are 011 (i.e. the cell directly above it is a one and that cells neighbors are both zeroes). That is three in decimal, so we look at the third bit of the rule set number, which is a zero (the bits in the rule-set number are numbered 0-7 from the right to the left, so the least significant bit is the zero'th bit and the most significant bit is the seventh). Now just set the value of the second cell in the new generation to zero. The third cell in the new generation has 110 above it. This is six in decimal, so we look at the sixth bit of the rule-set number. The sixth bit is a one, so set the value of the third cell in the new generation to one. Figure 3.2 shows the entire new generation from figure 3.1 filled in with values. You should take time to verify all of these values before starting the program, as this is the most conceptually difficult part of the assignment.

1	0	1	1	0	1	0	0	1	1	1	0
	0	0	1	0	1	1	0	0	0	1	

$a2 - 00000000 00000000 00000000 01010101

Figure 3.2: sample set of generations with answers

4 Final Remarks

There has been a good deal of research done on cellular automata. If you are interested in reading more about one-dimensional cellular automata, Steve Wolfram (creator of *Mathematica*) has written a number of good (although somewhat technical) papers on the subject, which are online. Once you have the simulate function completed you should spend some time experimenting with different rule sets and starting generations. Some rule sets produce quite boring results while others produce interesting and even remarkable behavior. If you run into a particularly interesting variation send me email and I will post a screenshot of it on the web for everyone to enjoy. The main function that I will use to test your function will be posted on the web page later this week.

For this assignment we will be taking points off for sloppy, uncommented code. So make sure to use comments and variable names well. Also, you should put your name, student ID and section in a comment at the top of the file. As in the last assignment, you will only be turning in a file with your function. The file you turn in should NOT contain a main function. If it does, you will lose points. Finally, the file you turn in should be named 'hw3.s'.