CSE 410
Programming Project 2

Assigned: Wednesday, October 17, 2001
Due: Wednesday, October 24, 2001 before class

Introduction

In this project, you will implement a simple 1-dimensional cellular automaton. A good definition of CA as it applies to this project is given in [Sipper].

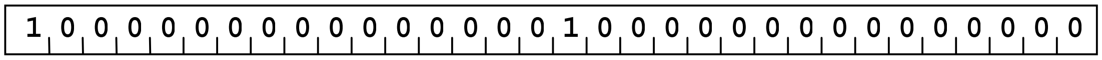
> Cellular Automata are dynamical systems in which space and time are discrete. A cellular automaton consists of a <u>regular grid of cells</u>, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps according to a local, identical <u>interaction rule</u>. The state of a cell is determined by the previous states of a <u>surrounding neighborhood of cells</u>.

Please read the above reference. It provides a concise description of 1-dimensional cellular automata.

The Algorithm

What is actually happening when this algorithm runs?

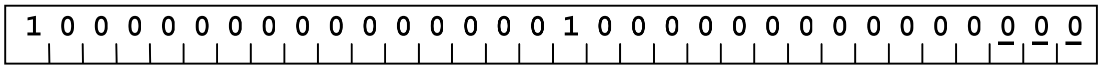
The "**regular grid of cells**" is just a long 1-dimensional row of bits. For the basic assignment, this row can be 32 bits long, and so it fits in one word. So the regular grid is a single word of 32 bits.



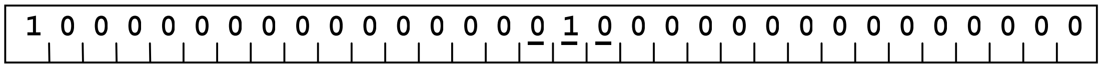
The "**surrounding neighborhood of cells**" is just whichever bit is the current bit and the bits on either side of it. This is what bitSlice3 from project 1 does - it isolates the 3-bit slice that is centered on a particular bit.

So here are some neighborhoods.

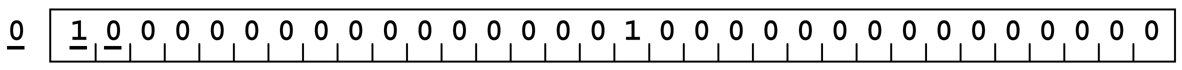
centerbit = 1, resulting bit slice = $000_2 = 0_{10}$



centerbit = 15, resulting bit slice = $010_2 = 2_{10}$

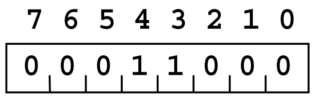


centerbit = 31, edge control = 0, resulting bit slice = $010_2 = 2_{10}$



A key point to notice is that the "resulting bit slice" shown above is a 3-bit number. We know that a 3-bit number can take on a range of values from 0 to 7. So every time we take a bit slice we get an *index value* that is in the range 0 to 7.

The last part of our algorithm is the **interaction rule**. This is the rule that tells us what the new bit values will be. This rule is specified with one 8-bit quantity. Each bit in the rule is accessed using the index value described in the previous paragraph. We use the 8-bit rule value as a very small array that has 8 single bit values that are numbered 0 to 7.



index

rule (=24_{10})

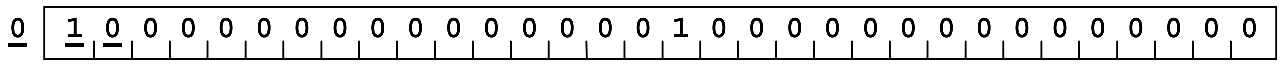
So the algorithm is as follows.

```
Start with an initial value for the <from> string as specified by the user
For the number of cycles specified by the user
      For each bit position in the <from> bit string
            Take the 3-bit neighborhood around the bit position (bitSlice3)
            Use the slice value as an index
            Look up the new bit value in the rule using the index
            Write the new bit in the <to> bit string at the bit position
      Write out the new "to" bit string
      Swap the <from> and the <to> addresses
```

The example on the following page steps through a few cycles of this process.

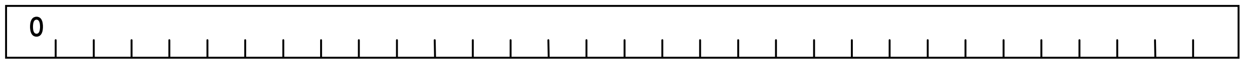
edge control = 0

center bit = 31



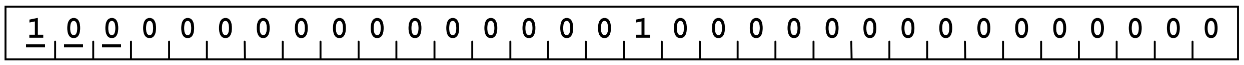
<from>

bit slice = 010_2 = 2_{10}, bit # 2 in rule 24 is 0, so the new bit 31 is 0



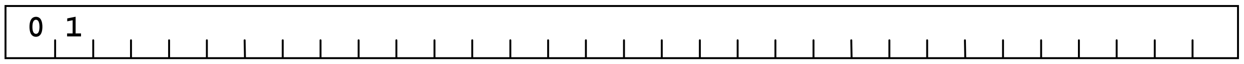
<to>

center bit = 30



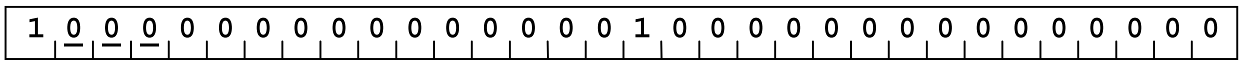
<from>

bit slice = 100_2 = 4_{10}, bit # 4 in rule 24 is 1, so the new bit 30 is 1



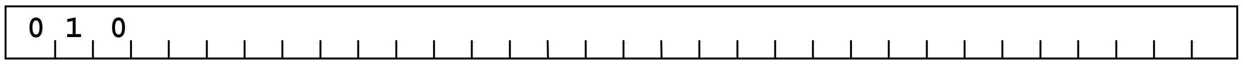
<to>

center bit = 29



<from>

bit slice = 000_2 = 0_{10}, bit # 0 in rule 24 is 0, so the new bit 29 is 0



<to>

The Program

Your task is to write a program that takes a set of input conditions (number of cycles, initial state, interaction rule, and edge control), then repeatedly updates the state of the bits according to the rule. You will use the decode_int, bitSlice3, and printBitString functions from the first project as part of this project.

1. Basic "main" Program

The "main" program takes two arguments from the command line, and then reads two more arguments from the console.

The first argument on the command line is "cycles", the number of times the user wants the program to cycle the bits in the bitstring. You need to use decode_int on this argument, just like the slicer program did with its command line arguments.

The second argument on the command line is the "seed number". This is the index of one of the hard coded initial states. There must be at least 5 initial state possibilities: 0x0, 0x1, 0xAAAAAAAA, 0x55555555, and 0x8000000. The user supplies the argument "seed number" on the command line. Your program uses decode_int to get the actual value of the argument, and then initializes the state using the specified seed from the array of possibilities.

After decoding the command line arguments, the program asks the user for a rule value, and reads in the user-supplied value using the syscall to read_int. If the rule value supplied is 0, then the program exits. Otherwise, the value must be greater than 0 and less than 256. This value is used for the <rule> when calling cycleCA (described in the next section).

The program next asks the user for an edge control value. This is either 0 or 1. This value is used for the <edge> when calling cycleCA.

Once all the values are obtained, then the main program calls procedure cycleCA for the number of cycles specified. Using printBitString, it prints out the resulting bit string after each call to cycleCA. After all the cycles are complete, it prompts again for the rule and the edge. If the rule is 0, the program exits. Otherwise, it goes through the whole process again.

2. Basic "cycleCA" procedure

The purpose of cycleCA is to derive the new set of bits (<to>), based on the old set of bits (<from>) according to the rule and the edge control. The basic procedure assumes that all bit strings are 32 bits long (ie, one word).

The basic cycleCA procedure takes four arguments.

<rule> : the 8-bit rule that governs how the new bits are generated
<from> : address(initial bit string), word aligned
<to> : address(result bit string), word aligned
<edge> : edge control, 0=> fill edge with 0, 1 => fill edge with 1

3. Extra credit

This is not part of the basic grading of the assignment.

Extend the main and cycleCA routines so that they can handle arbitrary length bit strings (up to some maximum value like 256 bits). Note that printBitString and my version of bitSlice3 already handle arbitrary length bitstrings and have an argument to specify the desired value.

a. Add an optional third argument to the command line that specifies the number of bits in the bitstring. This should default to 32.
b. Change the initial value array in main so that it contains initial values as long as the maximum bit string length you allow.
c. Add a fifth argument to cycleCA that specifies the <bit count>, ie, the number of bits in the bitstring.
d. Modify cycleCA so that it can loop through multiple words when the bit count is greater than 32.

Grading

The grading for this project is as follows.

Basic program:	7 points
Questions:	3 points
Total:	10 points * 5 = 50 points for the project
Extra credit:	Up to 5 points extra, making a maximum possible of 55 points.

Reference

[Sipper] A Brief Introduction To Cellular Automata, Moshe Sipper, Logic Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland

http://lslwww.epfl.ch/~moshes/ca.html