



Project 1: Rock Paper Scissors

The deadline for this project is **Monday, 14th of May**, end of day, anywhere on earth (AoE)¹. You can find more information about the project submission in appendix A.

1 Overview

In this project, you will implement the game rock paper scissors. Your implementation differs from the traditional two player game in the fact that the computer will play against itself. To do this, you will randomly generate moves for both players, compare their selections and announce the winner!

The following sections structure the project logically and their associated tasks are roughly in increasing difficulty. Section 2 documents how your program should generate rock, paper and scissors at random using MIPS's random number generation syscalls. Section 3 summarizes how a single round of rock paper scissors is played and how the winner is determined. In section 4, we replace MIPS's random number generator by a cellular automaton.

2 Generating Random Numbers

Modern computers are deterministic machines, strictly following the instructions given. However, many applications, such as random simulations, ask for probabilistic behaviour of the machine. Due to computers deterministic fashion, obtaining true randomness is a very difficult task.

Instead of true randomness, computers provide means of generating pseudorandom numbers, that is numbers that are indistinguishable from truly random numbers. To generate pseudorandom numbers, the computer starts with an initial configuration, called the seed, algorithmically generating a chaotic sequence of numbers. Due to the deterministic nature of the random number generator, given the same seed twice, the algorithm will produce the same sequence of numbers twice. We will use known seed values to make our random programs reproducible.

Configuration

Your MIPS program's behaviour will be determined by the program configuration, which is a sequence of values in memory. Starting from some address `conf`, the relevant values in memory are always at fixed locations relative to `conf`. The following table uses the `offset(address)` notation which specifies the memory address `offset` bytes after `address`.

	Configuration					
Name	eca	tape	tape_len	rule	skip	column
Size	(4 bytes)	(4 bytes)	(1 byte)	(1 byte)	(1 byte)	(1 byte)
Memory Address	conf	4(conf)	8(conf)	9(conf)	10(conf)	11(conf)

For tasks 1–3, the value `eca` will always be 0 and `tape` contains the seed that your random number generator will use. Section 4 documents the other values in the case that `eca` \neq 0.

Random Numbers in MIPS

The MARS simulator supports several syscalls related to random number generation. Each syscall related to random number generators receives as first argument a number uniquely determining the random number generator to use. Throughout this project, will always use the random number generator with i.d. 0. Figure 1 shows an excerpt of the MARS documentation (accessed by pressing F1 in the simulator).

Your task is to implement two functions `gen_bit` and `gen_byte`. Assume in both of these functions that the random number generator's seed has already been set, e.g. by the `.main` program.

Assignment 1: `gen_bit` (1 Points)

Implement the function `gen_bit` in `random.s`. When `gen_bit` is called, it should query the random number generator for the next number and return its least significant bit.



¹https://en.wikipedia.org/wiki/Anywhere_on_Earth

Service	Code in \$v0	Arguments	Result
set seed	40	\$a0 = i.d. of pseudorandom number generator (any int). \$a1 = seed for corresponding pseudorandom number generator.	No values are returned. Sets the seed of the corresponding underlying Java pseudorandom number generator (java.util.Random). See note below table
random int	41	\$a0 = i.d. of pseudorandom number generator (any int).	\$a0 contains the next pseudorandom, uniformly distributed int value from this random number generator's sequence. See note below table

Figure 1: MARS syscall documentation for random number generation

Generating Rock Paper Scissors Moves

Using the bits produced by `gen_bit`, you will generate the rock paper scissors moves played by the computer player. It is clear that one bit is not sufficient to choose between rock, paper and scissors with equal probability of $\frac{1}{3}$. Generating a second bit will give us four possible outcomes (00, 01, 10, 11), but assigning them to rock, paper and scissors will not result in equal probabilities for each move.

Figure 2 illustrates a procedure that uses only random *bits* to generate three equally likely outcomes 00, 01 and 10. The arrows of the graph indicate the results of the function `gen_bit`, the leaf nodes at the bottom illustrate the final results. Essentially, we keep throwing away 11 results, until a permitted result comes up. If we *would* end with 11, we start over and generate (at least) two more bits. Implement this procedure in the function `gen_byte`.

Your implementation must only make calls to `gen_bit` when necessary. Calling the function more often will result in the random sequence deviating from what we expect and your program will fail to produce the expected outputs.

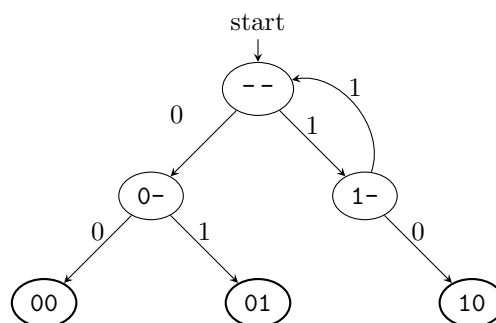


Figure 2: Uniformly sampling the three results 00, 01 and 10, only using bits of randomness

Assignment 2: `gen_byte` (1 Points)

Implement the function `gen_byte` in `random.s`.



3 Simulating the Game

In the function `play_game_once` you will simulate a single round of rock paper scissors. First, you will call `gen_byte` twice, generating the moves for player one and one for player two, respectively. The bytes generated by `gen_byte` correspond to the moves of the game as follows:

Byte	Move
00	Rock
01	Paper
10	Scissors

Rock wins against scissors, but loses against paper, paper wins against rock, but loses against scissors, and scissors wins against paper but loses to rock. All other combinations are ties.

Your task is to announce whether player *one* won, lost or tied the game. In each case, correspondingly print W, L, or T to the console.

Assignment 3: play_game_once (2 Points)

Implement the function `play_game_once` in `rps.s`.



4 Cellular Automata

Cellular automata, such as Conway's Game of Life, are computational models consisting of a collection of *cells* and a *transition rule*. Each cell can be in one of several *states* and its state can be updated by applying the transition rule simultaneously to all cells. We call the state of all cells a *generation* and we transform one generation of cells to the next by applying the rule to all cells.

For example, in Conway's Game of Life, the cells are arranged on an infinite 2-dimensional board. Each cell can be either *live* or *dead*. The transition rule describes whether a cell will be live or dead in the next generation, depending on the number of live cells bordering any of the four sides of the cell, and whether the cell in question started live or dead.

Elementary Cellular Automaton

Our concern lies on Elementary Cellular Automata (ECAs), a variant of cellular automata with a 1-dimensional, finite tape, in which cells can be *live* or *dead*. The right side of the tape connects to the left side of the tape, hence there is no "end". Whether or not a cell is alive or dead in the next generation solely depends on the state of the cell itself and its left and right neighbors.

Figure 3 illustrates an example. White cells are considered dead and black cells are considered live. Starting in the first line, we use a simple rule as example: A cell is live in the next generation if it was dead and both of its neighbors were live in the current generation.

The first line, generation 1, shows the initial configuration of the tape. It contains four live cells, each separated by one dead cell, and one group of two dead cells. In generations 2, 3, and 4, dead cells surrounded by live cells will turn live themselves, and the live cells become dead. Starting from generation 4, no two live cells exist, resulting in no new live cells being created. Thus, from generation 5, the tape consists of only dead cells.

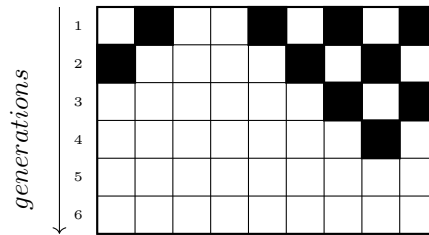


Figure 3: Six generations of Rule 32, starting on tape $\langle 149 \rangle_{10}$

Tape Encoding We can elegantly describe an ECA's initial tape as a binary number. Dead cells correspond to 0-bits and live cells to 1-bits. The initial tape shown in the first row of fig. 3 translates to the binary number $\langle 010010101 \rangle_2$, which equals to $\langle 149 \rangle_{10}$.

Rule Encoding Similar to how we encoded the tape, we can elegantly encode an ECA's transition rule by a single number. Notice how there are only $2^3 = 8$ possible 3-cell neighborhoods for a cell. For each such neighborhood, our transition rule must decide whether or not the cell of the next generation will be dead or alive.

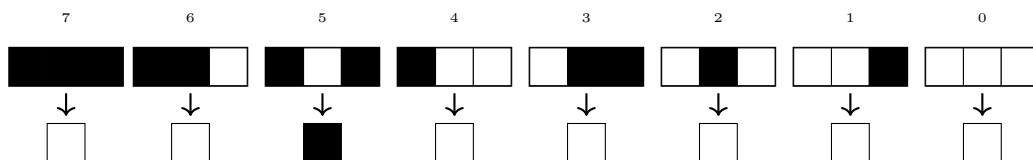


Figure 4: A graphical description of "Rule 32"

To encode the transition rule, we list all eight such neighborhoods from $\langle 111 \rangle_2$ to $\langle 000 \rangle_2$ and denote whether they should produce a dead or a live cell, encoded as 0 and 1, respectively. Recall that the left neighbour of the left most cell is the right most cell and the right neighbour of the right most cell is the left most cell. Then, we can understand this sequence of 0s and 1s as a binary number that uniquely determines the transition rule of our ECA. Figure 4 illustrates the encoding of Rule 32 which was used in fig. 6. In this example, a cell only produced a live cell, if it was initially dead and both of its neighbors were alive. This behaviour is captured by

the rule in which only the neighborhood $\langle 101 \rangle_2$ produces a live cell. We encode the transition rule as the 8-bit number that has ones corresponding to each neighborhood that produces a live cell. The rule depicted in fig. 4 is $\langle 00100000 \rangle_2 = \langle 32 \rangle_{10}$.

You can find more visualizations on https://en.wikipedia.org/wiki/Elementary_cellular_automaton.

Configuration


The upcoming tasks use the previously not discussed values of the `configuration`, elaborated below.

	Configuration					
Name	eca	tape	tape_len	rule	skip	column
Size	(4 bytes)	(4 bytes)	(1 byte)	(1 byte)	(1 byte)	(1 byte)


The value `tape` describes the initial configuration of the tape, `tape_len` describes the length of the tape before repeating ($1 \leq \text{tape_len} \leq 30$) and `rule` contains the transition rule of the automaton, encoded as described in “Rule Encoding”.

When using the ECA to generate pseudorandom numbers, `tape` again plays the role of the seed of our random number generator.


Assignment 4: `print_tape` (2 Points)

Implement the function `print_tape` in `automaton.s`. When called, it should print the ECA’s current tape to the console. Print an `X` in place of an alive cell, print an underscore (`_`) for a dead cell. Make sure to only print `tape_len` characters. Do not forget to print a newline character (`\n`) to end the line. 

Assignment 5: `simulate_automaton` (3 Points)

Implement the function `simulate_automaton` in `automaton.s`. Calling `simulate_automaton` should simulate one generation of the ECA and update the tape in the configuration struct accordingly. 

Assignment 6: `gen_bit` (1 Point)

Update your implementation of `gen_bit` in `random.s`. If `eca` is non-zero, you should use the ECA to generate a random bit. To do this, read the values `skip` and `column` from the configuration struct, simulate the automaton `skip` times and return the bit in the `column`th column of the tape. 

A Infrastructure / How to pass

Using dGit

Use git to clone your personal project repository from dGit, for example by entering

```
1 git clone git@dgkit.cs.uni-saarland.de:2222/prog2/2024/students/project-1-{your matriculation number}.git
```

Submit your project using `git push` until Monday, 14th of May, end of day, anywhere on earth (AoE). If you need a refresher on how to use git, consult the materials from the Programming 2 Pre-Course or additional material online².

Grading

Similar to the way you obtain your project, you will submit the project through git. The last commit pushed to our server before the deadline will be considered for grading. Please ensure that the last commit actually represents the version of your project that you wish to hand in.

Once you `git pushed` your project to our server, we will automatically grade your implementation on our server and give you feedback about its correctness. You can view your latest test results by following the link to the Prog2 Leaderboard in dCMS Personal Status page. We use three types of tests: **public** tests, **regular** tests and **eval** tests. The public tests are available to you from the beginning of the project in your project repository. You should use them to test your implementation locally on your system. Once you *pass all public tests*, we will run the regular tests for your implementation, disclosing to you the names of the regular tests and whether or not you successfully pass them.

Eval tests are similar to regular tests, but run only once after the project deadline. They directly determine the number of points your implementation scored.

Hints

- (a) From your submission, we will only grade the files in the `src` directory which were already shipped with your git repository
- (b) Adhere to the MIPS calling conventions!
- (c) You must assume that we test files of your implementation individually: We will for example check, if your `random.s` can cooperate with our `automaton.s`.
- (d) Note that we will use the MARS version uploaded in dCMS to grade your project. Your project will only score points if it shows the expected behaviour on our server, not necessarily your machine.
- (e) Document your assembly code with comments! It makes your code more readable and easier to debug.

²Such as <https://git-scm.com/book/en/v2>

B Writing, Testing, and Debugging MIPS in MARS

First, clone your repository and download the MARS executable from the dCMS Materials page. Then, place `Mars4_5.jar` into the root directory of your repository, e.g. `project-1/`. We recommend that you start MARS from the commandline within your project directory, e.g. by navigating your commandline to the directory and executing `java -jar Mars4_5.jar`.

When developing in MARS, it is imperative that you first tick the boxes *Assemble all files in directory* and *Initialize Program Counter to global 'main' if defined* under *Settings* to initialize your program at the `.main` label in `src/main.s`.

Testing

You can test the correctness of your implementation by running the provided `run_tests` script. The script will then run your implementation against the tests contained in the directory `tests/`. Each test consists of a pair of two files: `{testname}.s` and `{testname}.ref`. The `.s` file will take the role of the `main.s` file during execution of the test. `{testname}.ref` is a text file that contains the *reference output*, e.g. the text that a correct implementation would print during execution.

Debugging By default, our provided script `run_tests` prints a summary of which tests produced the correct output, and which did not. Calling it with the extra *verbose* flag `./run_tests -v` gives you more explicit information, in particular the text printed by your program. In order to investigate existing bugs deeper, it can be advisable to manually step through a test scenario in MARS. Executing `./run_tests {path_to_test.s} -debug` creates a directory named `debugbox/` containing all files that constitute the test, allowing you to easily open and debug the relevant files in MIPS.

Custom Tests To grow confident that your implementation is actually correct and does not just coincidentally pass all tests, we recommend you create additional tests yourself. We recommend creating a directory `tests/custom/` that holds your custom `.s` and `.ref` files. When you fail a custom test, you know that (a) your program incorrectly produced the wrong output, or (b) the output specified in the `.ref` file is incorrect and you need to improve your understanding of the test case and correct it.

C ECA Simulation Example

For better understanding of the bonus task, we will explain a complete example, giving more insight into how `play_game_once` works. Consider the following example configuration struct at memory address `conf`:

Name	eca	tape	tape_len	rule	skip	column
Size	(4 bytes)	(4 bytes)	(1 byte)	(1 byte)	(1 byte)	(1 byte)
Memory Address	conf	4(conf)	8(conf)	9(conf)	10(conf)	11(conf)
Value	3	252	8	106	1	5

As `eca` equals three—which is non-zero—we should use an ECA to generate random numbers for the rock paper scissors simulation. The initial tape configuration is 252 and the tape length is 8, hence we can depict the tape as the 8 bit binary number $\langle 252 \rangle_{10} = \langle 11111100 \rangle_2$, depicted as generations 1 of fig. 6. Encoded in binary, the rule reads $\langle 106 \rangle_{10} = \langle 01101010 \rangle_2$. The eight patterns, produce a live cell, if and only if their corresponding bit in the rule is 1.

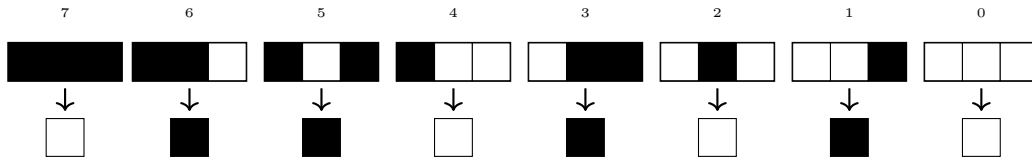


Figure 5: A graphical description of “Rule 106”

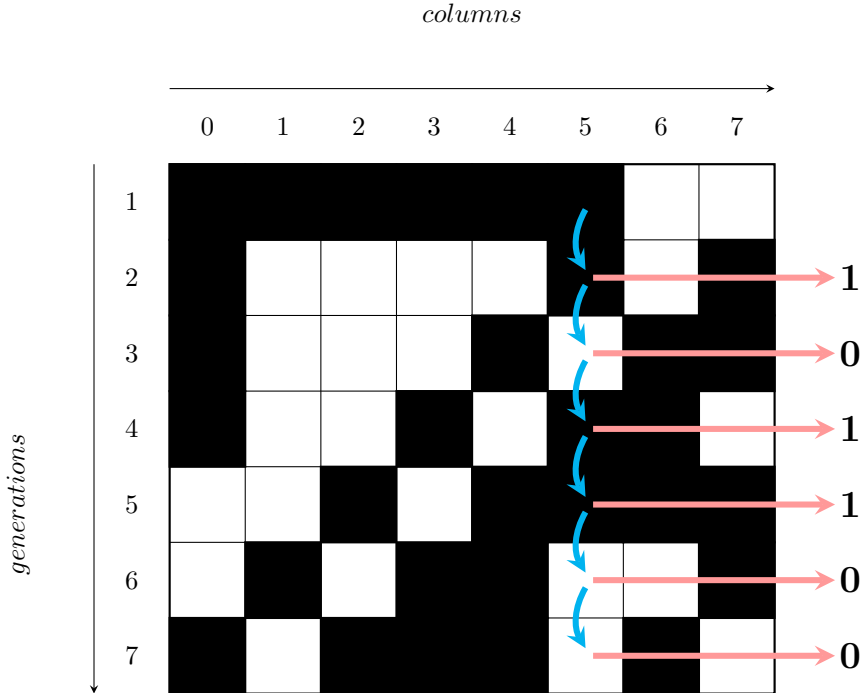


Figure 6: Six generations of Rule 106, starting on tape $\langle 252 \rangle_{10}$

The parameters `skip` and `column` instruct us how to generate the next bit. First, we skip `skip=1` generation and output the bit of the column with index `column`.

To simulate the first round of the game, we start by generating a move for the first player. Calling `gen_byte` calls `gen_bit` twice. `gen_bit` will instruct the automaton to skip one generation and output bit **1**. The next call to `gen_bit` skips to the next generation and produces **0**. Thus, our call to `gen_byte` returns **10**, which we interpret as player one playing scissors.

We repeat the same procedure for player two. Calling `gen_byte` produces **11**, which we must discard, as it does not correspond to a legal move. The subsequent call to `gen_byte` produces **00**, which we interpret as player two playing rock.

To conclude, player one loses to player two, requiring the implementation to output L.