Victoria University of Wellington
School of Engineering and Computer Science

SWEN326: Safety-Critical Systems

Assignment 2

Due: Monday 25th May @ 23:59

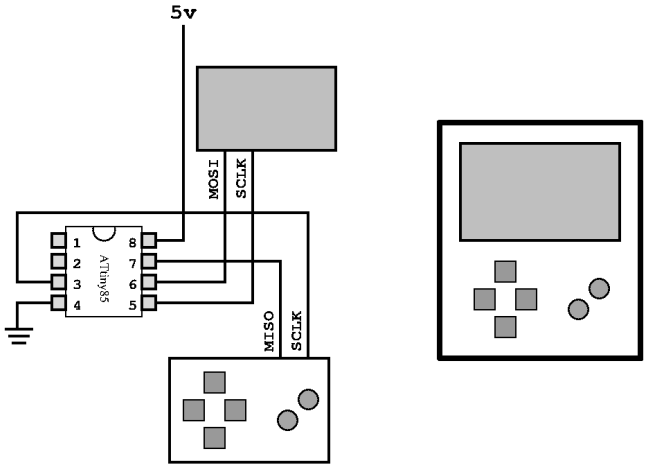
NOTE: For this assignment, you must use the giblab repository created specifically for you: `http://gitlab.ecs.vuw.ac.nz/course-work/SWEN326/2020/<username>`

Overview

The purpose of the assignment is to gain experience testing a realistic embedded system. Specifically, you will implement a tool which generates test cases for firmware running on an AVR ATtiny85.

TinyBoy

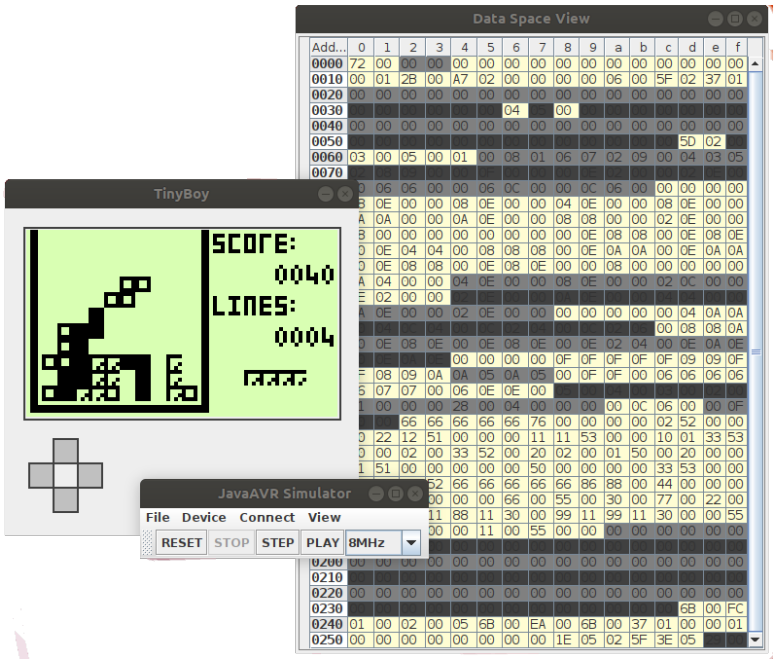
The *TinyBoy* system provides a simple embedded environment that roughly resembles a handle-held games console like the *Game Boy* or, perhaps more closely, the *Arduaboy*. Specifically, it provides simple monochrome dot-matrix display along with directional buttons for controlling games. A schematic is given as follows:



The system consists of the ATtiny85 AVR microcontroller, along with a *dot matrix display (64 x 64 pixels)* and a *control pad*. The two peripherals communicate with the AVR using the *Serial Peripheral Interface (SPI)*.

Emulator

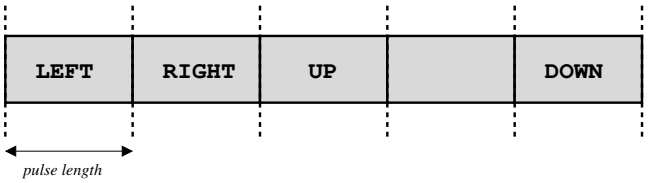
For the purposes of this assignment you will not be working physical hardware. Instead, you will be working with an *emulator* for the TinyBoy system:



The emulator emulates the underlying AVR microcontroller and the two peripherals. The advantage of using an emulator is that it makes development of embedded quicker and more easy to debug. From our perspective here, it also means we can instrument the AVR microcontroller as it is running to see exactly what it's doing. This means, for example, we can monitor memory usage and which instructions have been executed.

Fuzzing. In this assignment you will complete the components of a simple *feedback-directed fuzz tester* for TinyBoy programs. The goal is to automatically generate user inputs which force as many paths in the program to be executed as possible. Success will be measured by the amount of *instruction* and *branch coverage* obtained. To do this, You will use a mixture of approaches to input generation, adopting *exhaustive* generation as the primary approach whilst retaining scope to employ *randomness* and other techniques to help with *pruning*.

Input Sequence. This assignment is concerned with testing of TinyBoy programs. As such, the concept of an *input sequence* is used. That is a sequence of button pushes which are fed into the emulator. The sequence is divided up into a given number of *pulses*, where each pulse is a particular button being held down for a fixed *pulse length*. The following illustrates:



Here, we see a sequence of five pulses with the corresponding buttons being indicated. Note, to reduce the space of input sequences, the possibility of *multiple buttons* being held down at the same time is discounted. However, the possibility of *no buttons* being held down is permitted.

Input sequences are represented using the class `TinyBoyInputSequence`. A textual representation is also used for simplicity to aid debugging. For example, the string `"LRU_D"` represents the above sequence.

Getting Started. To get started, download the accompanying software `tinyboy-coverage.tgz`, and import the Java code into Eclipse as appropriate. This includes the the libraries `javaavr-1.x.jar` and `tinyboy-1.y.jar` which you will need to add as external archives. You should spend some time familiarising yourself with how the emulator works. In completing this assignment, you should extend the implementation found in `tinyboycov.core.TinyBoyInputGenerator`. *At this stage, relatively little functionality is provided and you will need to extend this considerably.*

Generating Input Sequences Suppose we want to generate an input sequence of length n. For each point in that sequence, there are *five* possible pulses (i.e. *no buttons pressed*, *left button pressed*, *right button pressed*, etc). Therefore, there are 5^n possible inputs we can generate. The following table illustrates the space for different values of n:

n	Size
1	5
2	25
3	125
4	625
5	3125
6	15625
7	78125
8	390625
9	1,953125
10	9,765625

As we can see, the number of possible inputs goes up *very rapidly indeed*! From this, it seems completely implausible that we could possibly generate input sequences with 10 or more pulses, for example. However, with careful pruning, we can indeed generate relatively large input sequences to test our programs!

Part 1 — Single Input Programs (worth 5%)

You should find a selection of tests in the class `Part1_Tests` to get started with. These are very simple tests indeed, which simply loop until a single input is given. Thus, to get a sufficiently high coverage, we must find the correct input. To do this, *we need only consider all possible sequences of length one.* **For this part, the coverage target is 100%.**

Part 2 — Simple Sequence Programs (worth 15%)

You should find a slightly more complex selection of tests in the class `Part2_Tests`. These require specific input sequences of length at most four to unlock. This time, to get a sufficiently high coverage, we must find the correct *sequence*. However, this is only a marginally harder task than before. **For this part, the coverage target is 100%.**

HINT: The key challenge at this point is to begin thinking about how to programmatically generate *all input sequences* of a given length. Writing a function to do this will prove extremely beneficial later on.

Part 3 — TinyBoy Programs (worth 20%)

At this point, the challenge starts to increase considerably as we use more realistic programs for testing the TinyBoy emulator. These are compiled from C source files (located in the `tests/` directory) and, in many cases, require more complex input sequences than before. **For this part, the coverage target is 95%.**

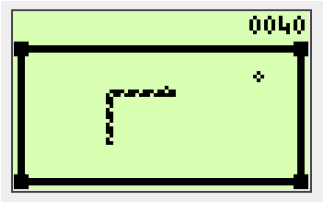
HINT: At this point, it starts to become less feasible that the problems can be solved simply by enumerating all inputs of a given length. Instead, we need to make use of the `record()` feature which provides feedback. For example, we might generate a batch of inputs and then, once they are all completed, examine their feedback and pick the best n (e.g. with respect to coverage).

HINT: You can modify the test programs if you wish by editing their C source files and recompiling with `avr-gcc`. A `makefile` is provided to help with this process.

HINT: You can speed the testing process through parallelism by adjusting the `NTHREADS` configuration parameter. Most likely, you will need to disable the GUI to see the real benefits of multiple threads.

HINT: The automated marking system used for this assignment will use a hard deadline of `300s` (5mins) for each test case. We can assume at least four threads will be available for this. The configuration parameter `HARD_TIMEOUT` should be enabled to check whether your program is likely to run into problems with timeouts. Unfortunately, this setting makes debugging harder as standard output is only printed afterwards.

Part 4 — TinyBoy Games (worth 30%)



Finally, we consider yet more challenging examples, including complete implementations of the well-known games *Tetris* and *Snake*. These require a sophisticated tuned approach to manage the space of inputs. In particular, we must reasonable amount of initial seeds, and then prune them based on feedback obtained before using them as seeds for the next generation. Some strategies for pruning are discussed below. **For this part, the coverage target is 85%.**

HINT: The Tetris and Snake games have been modified from their original form to help automated testing complete more quickly. These modified versions (found in `tests/`) are essentially unplayable. However, in the `roms/` directory you can find the original versions which you can run using the `Main` class provided.

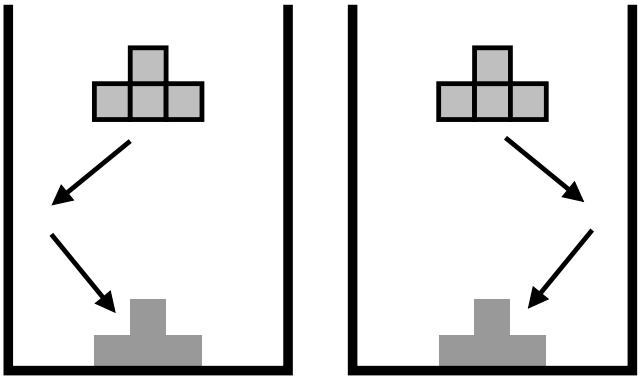
HINT: Whilst some realistic programs have been provided, keep in mind that the automated marking system will employ *hidden tests* for marking. **Therefore, you should design your fuzz test generator to work as well as possible on arbitrary programs**. Using the instructions given above you can create your own firmware programs very easily.

HINT: If you haven't explored the parallel processing feature of the testing harness yet, you should do so now. This can save a *considerable* amount of time on a typical multi-core machine.

Pruning Strategies

Making progress on the larger programs is challenging and we must find ways to reduce the number of inputs for each generation using the feedback obtained from testing.

(reduncancy) We can observe that many inputs lead to the same place. For example, in Tetris we can move a piece in several ways and arrive at the same state:



The `record()` method makes it relatively easy to spot when two inputs lead to an *identical state*. This is because it provides a complete dump of memory (i.e. SRAM) as it was when the test completed stored in a **`byte`**`[]` array. For inputs which lead to identical states, these arrays will be the same.

(representatives) Another useful approach to pruning is to retain a handful of representative elements from the different *classes* of input encountered, rather than *all* such inputs. A key question is how to divide up inputs into different classes, and there are at least two options here. We can consider inputs which have differing levels of coverage to be in different classes. Likewise, we can consider inputs which lead to different program states to be in different classes.

(subsumption) Finally, we can also consider inputs which are *subsumed*. For example, if every branch covered in an input sequence s_1 is also covered in another sequence s_2, then we say s_2 subsumes s_1, denoted $s_1 \subseteq s_2$. We can consider *strict* subsumption here as well.

HINT: An interesting question is whether one should consider the full range of input sequences. Specifically, whether or not there is value in considering the case when *no* button is pressed.

Marking

This assignment will be marked out of **100**, with marks based on four distinct areas:

- **Coverage Goals (70 marks)**. The correctness of your implementation will be assessed based on the number of passing (hidden) tests *as determined by the automated marking system*.
- **Code Quality**. The overall quality of your code will be assessed manually (by tutors). The marks for this section are broken down as follows:
  - **JavaDoc (5 marks)**. The quality of the JavaDoc comments you have written will be assessed. Comments are expected to be both *concise*, and to provide *accurate* summaries of the methods and/or fields in question.
  - **Decomposition (5 marks)**. The manner in which your code decomposes the problem will also be assessed. This means that methods should provide *logical* and *coherent* units which can be *easily understood*.
- **Coding Standard**. The degree to which your source code adheres to the required Java Coding Standard will be assessed using a combination of manual inspection (by tutors) and automated checking (e.g. by running the Eclipse compiler). Since this assignment is not concerned with developing safety-critical code, only Rule 9 *"Code is Free of Warnings"* and Rule 10 *"Checkstyle is Activated"* apply. The marks for this section are broken down as follows:
  - **Eclipse Warnings (5 marks)**. The degree to which your code meets the requirements of Rule 9 *"Code is Free of Warnings"* will be assessed.
  - **Checkstyle (5 marks)**. The degree to which your code meets the requirements of Rule 10 *"Checkstyle is Activated"* will be assessed.
- **Version Control (10)**. The manner in which you have used version control will also be assessed. Specifically, it is expected that all commits have sensible and coherent commit messages. As such, you are recommended to use a sensible style for your commit messages.[1]

NOTE: the test cases used for automated marking will differ from those in the supplementary code provided for this assignment. In particular, they may constitute a more comprehensive set of test cases than given in the supplementary code.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. The following file is required for submission:

`tinyboycov/core/TinyBoyInputGenerator.java`

You must ensure your submission is accepted by the submission system without error. This means it must: (a) successfully compile against `javaavr-1.x.jar` and `tinyboy-1.x.jar`; and, (b) pass at least one hidden test case. **A hard timeout of five minutes per test will be strictly enforced to ensure submissions can be marked in reasonable time**.

[1] See "How to Write a Git Commit Message", `https://chris.beams.io/posts/git-commit/`.