The University of Melbourne

SWEN90010: High Integrity Software Engineering

Assignment 3

Due Date: 11:59pm, Friday 31 May, 2019

1 Introduction

The assignment is worth 20% of your total mark and is done in pairs (the same pairs as assignments 1 and 2).

The subject of this assignment is the tiny assembly language Virtual Machine (VM) used throughout *SWEN90006 2018*. Details of that VM are repeated in Section 2.

The VM's behaviour has subtly changed. tl;dr: do read Section 2.1 at least.

The aim of this assignment is to use the SPARK Ada toolset to fix and verify the safety of an (somewhat buggy) Ada implementation of the VM. Additionally, to deepen your proficiency with SPARK Ada, you will design and implement an *analysis* routine for assembly programs to determine whether they contain certain kinds of faults.

As usual, get started early and use your pair to maximum advantage.

Download, install and check you can run the GNAT tools (see Section 3.1) ASAP!

2 The VM

The VM executes programs in a simple assembly language.

The Assembly Language

Programs written in the assembly language of the VM are a series of assembly instructions. Two example programs are shown in Figure 1 and Figure 2. In the figure, each instruction is on a separate line. ';' begins a single line comment, while blank lines and those that contain only comments represent instructions that do nothing (aka "nop" instructions).

The assembly language supports 10 instructions, each of which is a member of the following grammar, written here in *Backus-Naur Form* (BNF) style notation.[1]

[1] https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

```
;; factorial program, to calculate N!

;; global constants:
;;    R3 holds 'N', the thing we are computing factorial of
;;    R2 holds 1, the increment value used below
MOV R3 12                 ; N = 12
MOV R2 1                  ;

;; local variables
;;    R1 holds 'i', which is a counter from 0 .. N
;;    R0 holds 'n', which is always equal to i!
MOV R1 0                  ; i = 0;
MOV R0 1                  ; n = 1;

;;  program body
;;  loop invariant (see SWEN90010 next semester): n = i!
SUB R4 R3 R1              ; while(i != N)
JZ  R4 4                  ; {
ADD R1 R1 R2              ;    i = i + 1;
MUL R0 R0 R1              ;    n = n * i;
JMP −4                    ; }
RET R0                    ; return n;
```

Figure 1: An example assembly program for computing factorials.

```
;; array program. Fills an array with the values 0 ... N-1
;; and then iterates through the array, summing its elements

;; global constants:
;;   R3 holds 'N', the length of the array
;;   R2 holds 1, the increment value used below
MOV R3 10              ; N = 10
MOV R2 1               ;

;;  create the array
;; local variables
;;   R1 holds 'i', which is a counter from 0 .. N-1
;;   R0 holds 'p', the address of the array's ith element
MOV R1 0               ; i = 0;
MOV R0 100

SUB R4 R3 R1           ; while(i != N)
JZ  R4 5               ; {
STR R0 0  R1           ;    *p = i;
ADD R1 R1 R2           ;    i = i + 1;
ADD R0 R0 R2           ;    p = p + 1;
JMP -5                 ; }

;;  sum up the array
;; local variables
;;   R1 holds 'i', which is a counter from 0 .. N-1
;;   R0 holds 'p', the address of the array's ith element
;;   R5 holds 'sum', which always holds the sum of the array's first i e
MOV R1 0               ; i = 0;
MOV R0 100
MOV R5 0               ; sum = 0;

SUB R4 R3 R1           ; while(i != N)
JZ  R4 6               ; {
LDR R4 R0 0            ;
ADD R5 R5 R4           ;    sum = sum + *p;
ADD R0 R0 R2           ;    p = p + 1;
ADD R1 R1 R2           ;    i = i + 1;
JMP -6                 ; }
RET R5                 ; return sum;
```

Figure 2: Summing integers in an array.

```
<INSTRUCTION> ::=
                        "ADD" <REGISTER> <REGISTER> <REGISTER>
                      | "SUB" <REGISTER> <REGISTER> <REGISTER>
                      | "MUL" <REGISTER> <REGISTER> <REGISTER>
                      | "DIV" <REGISTER> <REGISTER> <REGISTER>
                      | "RET" <REGISTER>
                      | "MOV" <REGISTER> <VALUE>
                      | "LDR" <REGISTER> <REGISTER> <VALUE>
                      | "STR" <REGISTER> <VALUE> <REGISTER>
                      | "JMP" <VALUE>
                      | "JZ" <REGISTER> <VALUE>
```

Here "<REGISTER>" denotes a register name. The machine has 32 registers, and the valid register names are R0, R1, R2, . . . , R31. Each "<VALUE>" a decimal integer value (e.g. 0, 325, -1027 etc.) in the range $-65535 \dots 65535$ inclusive.

The assembly language includes instructions like "LDR" and "STR" for reading and writing to memory. The virtual machine has a memory of size 65536 words, where each word is a 32-bit signed integer.

The virtual machine has a special piece of state called the *program counter* (pc) which tracks which instruction (i.e. which line of the assembly program) it is currently executing. Certain instructions like "JMP" and "JZ" can be used to jump to different parts in the program, and so explicitly modify the pc. Otherwise, each instruction executes in order (i.e. after executing any non-"JMP" and non-"JZ" instruction, the pc is always incremented by 1, so that the subsequent instruction is executed).

If *Ra*, *Rb* and *Rc* are register names and *val* is an integer value, then the behaviour of each instruction is as follows:

Instruction	Description
ADD *Ra Rb Rc*	Adds the values in registers *Rb* and *Rc* and puts the result into *Ra*. We abbreviate this behaviour using the notation: $Ra = Rb + Rc$
SUB *Ra Rb Rc*	$Ra = Rb - Rc$
MUL *Ra Rb Rc*	$Ra = Rb * Rc$
DIV *Ra Rb Rc*	$Ra = Rb \,/\, Rc$
RET *Ra*	Causes the program to finish, returning whatever value is in *Ra*.
MOV *Ra val*	Puts the value *val* into register *Ra*.
LDR *Ra Rb val*	If *Rb* holds some value b, then first compute $b + val$. Then read from memory the value at address $b + val$ and put this into *Ra*.
STR *Rb val Ra*	If *Rb* holds some value b, then first compute $b + val$. Then store the value in register *Ra* into memory at the address $b + val$.
JMP *val*	Alter the pc by adding *val* to it. So "JMP 0" is an infinite loop, and "JMP 1" does nothing (i.e. is a nop).
JZ *Ra val*	If register *Ra* holds the value zero, alter the pc by adding *val* to it. (Otherwise, increment the pc by 1.)

The LDR and STR instructions are used to read from and write to memory respectively. The example program in Figure 2 shows these instructions in action. It creates an array with the integers $0 \dots 9$ and then iterates over the array, reading each of its elements and summing them together to produce the final result 45.

2.1 Invalid Behaviours

In this assignment, Ada's type system is used to ensure that all programs are syntactically correct. However, a program can still behave invalidly. This occurs when the last instruction executed by the program is not a RET instruction, since such programs do not return a value.

There are various behaviours that can cause a program to finish executing early, without returning a value (and so constitute invalid behaviours).

Recall that valid memory addresses are in the range $0 \ldots 65535$ inclusive. If the program tries to compute an address (i.e. during a LDR or STR instruction) that is out of range (i.e. < 0 or > 65535), then execution of the program terminates immediately without returning a value.

Recall that the word size of the machine is 32-bits and that each word is a signed integer. Therefore the range of valid values that can be held in a machine register, or in a memory location, is $-2^{31} \ldots 2^{31} - 1$ inclusive. If the program tries to compute a value in a register that is out of range (i.e. $< -2^{31}$ or $> 2^{31} - 1$) then this also causes execution of the program to terminate immediately without returning a value.

Similarly, if the program attempts to divide by zero (i.e. using a "DIV" instruction), then its execution should be terminated immediately without returning a value.

For this assignment, each program contains *exactly* 65536 instructions. The valid values of the program counter (pc) are 1..65536 (i.e. the first instruction in the program is at position 1). Executing a jump instruction (i.e. "JMP" or "JZ") that causes the pc to become < 1 or > 65536 causes the program to finish early without returning a value.

Return Codes The VM, when executing a program, may return one of three *return codes*:

- `Success`: this code should be returned when the program executes successfully. In this case, the VM also returns an (Ada) `Integer` value, which is the result returned by the executed assembly program (i.e. the result returned by the first "RET" instruction that was executed).
- `CyclesExhausted`: when given a program to execute, the VM is also given a fixed number of *cycles* for which it is allowed to run. On each cycle, one instruction is executed. Therefore, the *cycles* value specifies the maximum number of instructions that the VM will execute. If it executes this many instructions without encountering a "RET" instruction and without encountering invalid behaviour as described above, then `CyclesExhausted` will be returned.
- `IllegalProgram`: this code should be returned when the program executes an invalid behaviour as described above.

Exceptions The VM should *never* raise an exception or error. If it executes an invalid behaviour, it should return the return code `IllegalProgram`.

3 Your tasks

Get started early. Don't feel that you need to complete these tasks exactly in the order below.

3.1 Download and Install GNAT Community Edition

If you are working on your own machine, download and install GNAT Community Edition 2018 from: `https://www.adacore.com/download`.

Ensure that the `bin/` directory is in your `PATH` so that you can run the Ada tools directly. If your setup is correct, you should be able to run commands like `gnatmake` and `gnatprove` and see output like the following (noting that in this case the commands were run on MacOS):

```
$ gnatmake --version
GNATMAKE Community 2018 (20180523-73)
Copyright (C) 1995-2018, Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR \
PURPOSE.

$ gnatprove --version
2018 (20180523)
Why3 for gnatprove version 0.88+git
/Users/toby/opt/GNAT/2018/libexec/spark/bin/alt-ergo: Alt-Ergo version 1.30
/Users/toby/opt/GNAT/2018/libexec/spark/bin/cvc4: CVC4 version 1.6-prerelea\
se
/Users/toby/opt/GNAT/2018/libexec/spark/bin/z3: Z3 version 4.6.0 - 64 bit
```

3.2 Downloading and Building the VM Implementation

Download the ZIP file containing the VM implementation from the LMS. It contains a small number of Ada packages:

- `Driver`: the top level driver program that generates a random program and calls the VM to execute it.
- `Machine`: the VM implementation for executing and analysing programs.
- `Instruction`: definition of the VM instructions and debug routines for printing them out, etc.
- `Debug`: debug printing functions.

After unpacking the ZIP file, it will create the directory `assignment3` in which the Ada code is placed. You can build the code by running '`make`' in that directory.

```
$ make
gnatmake -f driver
gcc -c driver.adb
gcc -c debug.adb
gcc -c instruction.adb
gcc -c machine.adb
gnatbind -x driver.ali
gnatlink driver.ali
```

Note: if you are building the code on MacOS, you might get the warning message:

```
ld: warning: URGENT: building for OSX, but linking against dylib
(/usr/lib/libS ystem.dylib) built for (unknown). Note: This will be
an error in the future
```

This warning can be safely ignored.

Building the code should produce a binary `driver` that you can then run.

3.3 Overview of the Driver

Unlike the VM used in *SWEN90006 2018*, the Ada VM does not read its input programs from files. Instead, the driver generates random, syntactically correct, programs and calls the VM to execute them. Ada's strong, static type system is used to ensure only syntactically correct programs are generated.

However, the random programs that are generated will almost always contain invalid behaviours (as defined in Section 2.1 above).[2]

The VM is supposed to never raise an exception, even when encountering invalid behaviours. However, the implementation you are provided with is faulty. Among its faults, it is prone to raising various exceptions[3], particularly when executing invalid behaviours.

Before running the generated program on the VM, the driver first calls an (unimplemented) function of the `Machine` package whose job it is to *analyse* the program to detect whether it might encounter invalid behaviour (when executing the program for some given number of instructions). The purpose of this analysis function is to be *conservative*: it is allowed to falsely claim that a program might encounter invalid behaviour. However it should never fail to detect invalid behaviour that the program might encounter (when executed for the given number of instructions). The implementation of this function you are provided with detects invalid behaviour in *every* program (and so is *maximally* conservative).

For example, running the `driver` binary might produce the following:

```
$ ./driver
Generating Random Program...
Analysing Program for Invalid Behaviour...
Analysis Result: TRUE
Executing program...
1: STR R15 3873 R0
2: DIV R1 R14 R14

raised CONSTRAINT_ERROR : machine.adb:57 divide by zero
```

Here we see that the program was analysed and judged as possibly having invalid behaviour, before it was then executed on the VM. The VM prints out the pc and each instruction as it executes it. Here we can see that it executes a "STR" instruction followed by a "DIV" instruction.

[2]This should not be surprising, given what you already know about random fuzzing.

[3]It also has (at least one) subtle semantic bug. However, that bug becomes easier to spot once you have fixed the more egregious bugs that cause exceptions.

The implementation of the "DIV" instruction, however, is faulty and causes a division by zero exception to be raised (whereas the VM should instead have halted the program and given the return code `IllegalProgram`).

Consider a second example execution that does not raise an exception:

```
$ ./driver
Generating Random Program...
Analysing Program for Invalid Behaviour...
Analysis Result: TRUE
Executing program...
1:      SUB     R5      R17     R13
2:      NOP
3:      NOP
4:      SUB     R22     R20     R27
5:      ADD     R28     R9      R26
6:      ADD     R30     R19     R4
7:      ADD     R29     R6      R23
8:      JMP     61310
61318:  RET     R12
Return Code: SUCCESS Result: 0
```

Here we see that the analysis judged this program as possibly containing invalid behaviour. When executed on the VM, however, the return code `Success` is printed, along with the result 0 returned by this random program.

3.4 Implementing a More Precise Analysis

One task of this assignment is for you to modify the maximally conservative analysis function to implement a more precise analysis.

The function in question has the following signature:

```
function DetectInvalidBehaviour(Prog : in Program;
                                Cycles : in Integer) return Boolean;
```

Its job is to analyse the program, assuming the program's execution starts from an *arbitrary* initial state in which the contents of the registers and memory are *unknown*. `DetectInvalidBehaviour` should return `True` if the program might encounter invalid behaviour while executing for `Cycles` instructions, and should return `False` only when the program is guaranteed not to encounter invalid behaviour while executing for `Cycles` instructions *from an arbitrary initial state.*

Your goal is to have your analysis be as precise as possible, by which we mean that you should attempt to minimise the number of *false positives*: i.e. the number of times that your analysis wrongly returns `True`.

We will run your analysis function on various test cases to judge how precise it is, and to check whether it correctly detects invalid behaviours. Your analysis function will also be judged according to how elegant is its design and implementation.

You should comment your analysis function to describe its overall design, and any interesting parts of its implementation.

Rules You are free to implement whatever kind of analysis you like for this task, including any combination of static and dynamic analysis that you see fit.

(By "static" analysis, we mean an analysis that does not involve running the program or simulating its execution, but instead just examines its code. "Dynamic" analysis is the opposite of static analysis, and involves executing the program or simulating its execution, usually to detect certain kinds of errors.)

However you are *not* allowed to change the specification of any of the functions or procedures defined in `machine.ads`. Your implementation should work against the provided versions of the other packages.

3.5 Understanding the VM's Faults

Your next task is to use the SPARK tools to help diagnose as many of the faults in the provided VM implementation as possible.

Running `make prove` will run the SPARK Prover over the VM implementation.

For this task, you are required to document each of the faults that you diagnose. You should explain each fault, what behaviour causes the fault, and what happens when the fault is triggered.

You must also document for each fault which of the warning/error messages given by the SPARK Prover are evidence of that fault. For instance, when running `make prove` on the provided VM implementation, amongst the many warning and error messages it produces is the following which is evidence for the division by zero fault discussed above:

```
machine.adb:60:29: medium: divide by zero might fail, in call inlined at
machine.adb:122 (e.g. when Regs = (others => 0))
```

3.6 Fixing and Proving the VM and Analysis Safe

Your final task is to fix both the VM implementation and the implementation of your analysis function so that the SPARK prover issues no warnings and no error messages when `make prove` is run.

If you are successful, running the SPARK prover via `make prove` should look something like this:

```
$ make prove
gnatprove -P default.gpr -f
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /Users/toby/assignment3/gnatprove/gnatprove.out
```

Hint: your analysis function will almost certainly involve a loop. In order for GNAT Prover to determine that your analysis function does not raise any exceptions, you might need to state suitable *invariants* for each of its loops. See e.g.: `https://docs.adacore.com/spark2014-docs/html/ug/gnatprove_by_example/loop.html` and other similar resources therein and online.

Rules As above, you are *not* allowed to change the specification of any of the functions or procedures defined in `machine.ads`. Your implementation should work against the provided versions of the other packages.

4 Criteria

Criterion	Description	Marks
Analysis	Analysis never fails to detect invalid behaviour. Analysis precision (i.e. low false positive rate). Elegance and clarity of design and implementation. Your analysis function is free of faults. GNAT Prover judges your analysis implementation as free of faults.	10 marks
Fault diagnosis	All faults in provided VM are documented. Fault causes and consequences are clearly explained. SPARK warning/error messages for each fault are provided.	5 marks
VM Fixed	Faults in provided VM implementation fixed. GNAT Prover judges your fixed VM as free of faults.	5 marks
Total		20 marks

5 Submission

The submission has two parts.

Ada Code You will submit your `machine.adb` file that contains both your analysis implementation and the fixed VM implementation. This file should compile and run against the provided versions of the other package files, since we will be running automated tests on your code using those files.

Go to the SWEN90010 LMS page, select *Assignments* from the subject menu, and then select *View/Complete* from the *Assignment 3 Ada Code Submission* item. Upload your `machine.adb` file, ensuring that it has this file name and that the comments in the file clearly identify *both* authors in your pair.

Fault Report You will also submit a report describing the faults you diagnosed in the provided VM implementation.

Go to the SWEN90010 LMS page, select *Assignments* from the subject menu, and then select *View/Complete* from the *Assignment 3 Fault Report Submission* item. Upload your report as a PDF file, ensuring that it clearly identifies *both* authors in your pair.

Late submissions Late submissions will attract a penalty of 2 marks for every day that they are late. If you have a reason that you require an extension, email Toby *well before the due date* to discuss this.

Please note that having assignments due around the same date for other subjects is not sufficient grounds to grant an extension. It is the responsibility of individual students to ensure that, if they have a cluster of assignments due at the same time, they start some of them early to avoid a bottleneck around the due date. The content required for this assignment was presented before the assignment was released, so an early start is possible (and encouraged).

6 Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the pair's understanding of the topic.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.