SWEN90006 Software Testing and Reliability

2018 Semester 2

Ivan Ken Weng Chee

736901

https://gitlab.eng.unimelb.edu.au/ichee/SWEN90006-A1-2018

Task 1

Function signature for the execute method:

```
int execute(List<String> instructions) throws InvalidInstructionException, NoReturnValueException
```

Input Domain

ID = All possible lists of strings

Assumptions:

- The program is syntactically correct, but may or may not be valid (*SWEN90006: Software Testing and Reliability: Assignment 1*, 2018)
- At least one "ret" instruction is always present in the input
- The number of instructions is assumed to be greater than 0
- Comments and blank lines do not cause errors to be thrown
- Default value of register and memory is 0
- Values entered are not Floats or Doubles, and can be parsed into Integers successfully
- The input to the function is not null
- Assembly programs that result in an infinite loop is part of the valid input domain

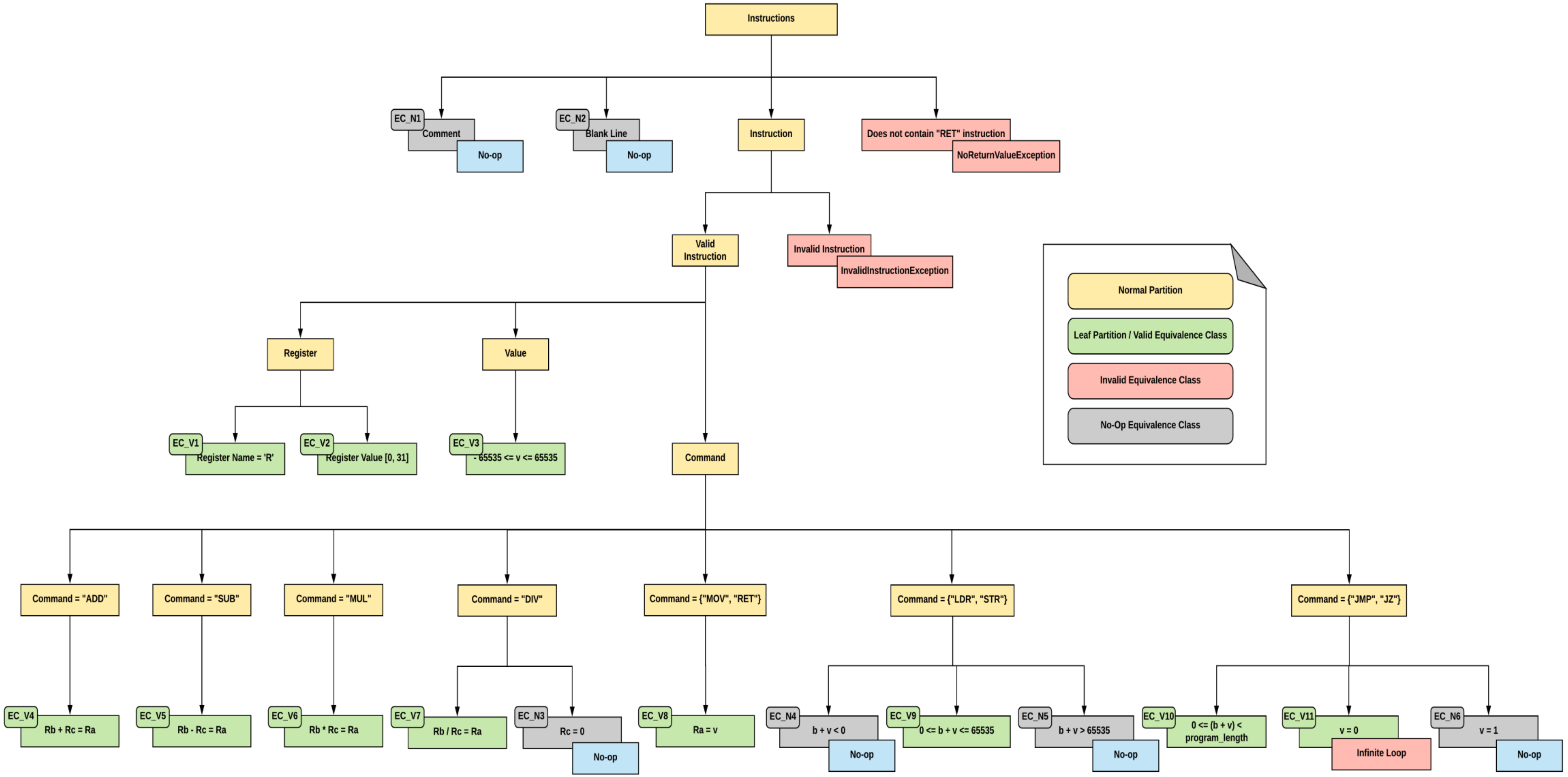
Input Conditions

The following are input conditions and naming conventions derived from the grammar written in Backus-Naur Form (BNF) ("Backus–Naur form", 2018).

- {instructions | instructions ∈ List<String> ∩ length(instructions) > 0}
- {instruction | instruction ∈ String ∩ instruction.split(" ") ∈ List<String> ∩ l length(instruction) ∈ {2, 3, 4} ∩ instruction = {command register || command value || command register value || command register register register || command register value register || command register register value}}
- {command | command ∈ String ∩ command ∈ {"add", "sub", "mul", "div", "ret", "mov", "ldr", "str", "jmp", "jz"}
- {register | register ∈ String ∩ register = {name value}}
- {name | name = 'R'}
- {value | value ∈ int ∩ value ∈ [-65535, 65535]}
- EC_V = Valid Equivalence Class
- EC_N = No-Op Valid Equivalence Class

Test Template Tree

(Miller, 2018), ("Online Diagram Software & Visual Solution | Lucidchart", 2018)


Instructions
EC_N1
Comment
No-op
EC_N2
Blank Line
No-op
Instruction
Does not contain "RET" instruction
NoReturnValueException
Valid Instruction
Invalid Instruction
InvalidInstructionException
Normal Partition
Leaf Partition / Valid Equivalence Class
Invalid Equivalence Class
No-Op Equivalence Class
Register
Value
EC_V1
Register Name = 'R'
EC_V2
Register Value [0, 31]
EC_V3
- 65535 <= v <= 65535
Command
Command = "ADD"
Command = "SUB"
Command = "MUL"
Command = "DIV"
Command = {"MOV", "RET"}
Command = {"LDR", "STR"}
Command = {"JMP", "JZ"}
EC_V4
Rb + Rc = Ra
EC_V5
Rb - Rc = Ra
EC_V6
Rb * Rc = Ra
EC_V7
Rb / Rc = Ra
EC_N3
Rc = 0
No-op
EC_V8
Ra = v
EC_N4
b + v < 0
No-op
EC_V9
0 <= b + v <= 65535
EC_N5
b + v > 65535
No-op
EC_V10
0 <= (b + v) < program_length
EC_V11
v = 0
Infinite Loop
EC_N6
v = 1
No-op

Input Space Coverage

In my opinion, my set of equivalence classes does not fully cover the input space, due to some of the assumptions made. For example, assuming the length of instructions is > 0 disregards lists of strings with length = 0. The assumption that the input program adheres to the grammar also disregards strings with an arbitrary number of space characters, as in lists having the wrong length after splitting on space (i.e. [“ADD”, “R1”]).

In terms of parsing, I have made an assumption that values entered in instructions. A null pointer or uninitialized List may also be accepted as input and may not be detected by the compiler. Derived classes inherited from List, such as ArrayList, LinkedList, and Stack may also be used as input and may cause undefined behaviour due to polymorphism.

Task 3

EC	On Point(s)	Off Point(s)

EC_V1	Rn = ‘R’	Rn = ‘Q’ (R - 1)
		Rn = ‘S’ (R + 1)
		("ASCII Table | ASCII character codes table", 2018)

EC_V2	Rv = 0
	Rv = 31
		Rv = -1
		Rv = 32

EC_V3	v = -65535
	v = 65535
		v = -65536
		v = 65536

EC_V4	(Rb,Rc,Ra) = (-65535,-65535,-131070) (Rb,Rc,Ra) = (65535,65535,131070)	(Rb,Rc,Ra) = (-65536,-65536,-131072) (Rb,Rc,Ra) = (65536,65536,131072)
EC_V5	(Rb,Rc,Ra) = (-65535,65535,-131070) (Rb,Rc,Ra) = (65535,-65535,131070)	(Rb,Rc,Ra) = (-65536,65536,-131072) (Rb,Rc,Ra) = (65536,-65536,131072)
EC_V6	(Rb,Rc,Ra) = (65535,65535,4294836225) (Rb,Rc,Ra) = (-65535,65535,-4294836225)	(Rb,Rc,Ra) = (65536,65536,4294967296) (Rb,Rc,Ra) = (-65536,65536,-4294967296)
EC_V7	(Rb,Rc,Ra) = (-65535,1,-65535) (Rb,Rc,Ra) = (65535,1,65535)	(Rb,Rc,Ra) = (65535,-1,-65535) (Rb,Rc,Ra) = (65535,1,65535)

EC_V8	Ra = 0	Ra = Ra - 1
		Ra = Ra + 1

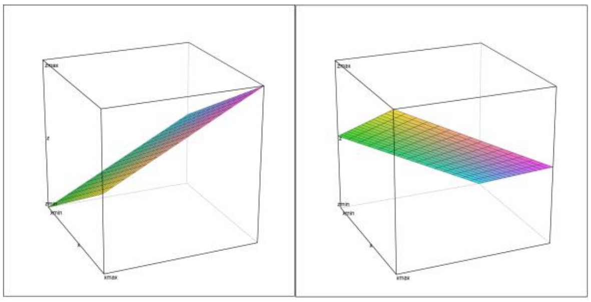
EC_V9	(b,v,b + v) = (0,0,0) (b,v,b + v) = (0,65535,65535)	(b,v,b + v) = (0,-1,-1) (b,v,b + v) = (1,65535,65536)
EC_V10	(b,v,b + v) = (0,N - 1,N - 1)	(b,v,b + v) = (0,-1,-1) (b,v,b + v) = (0,N,N)
EC_V11	v = 0	v = -1
JZ_0	(Ra, v) = (0, 2)	(Ra, v) = (1, 1) (Ra, v) = (-1, 1)

EC_N1	instruction[0] = ‘;’	instruction[0] = ‘:’
		instruction[0] = ‘<’
		("ASCII Table | ASCII character codes table", 2018)

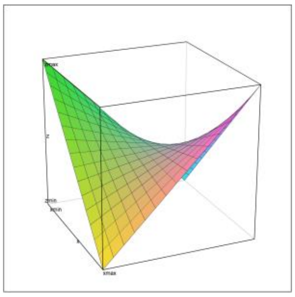
EC_N2	instruction = “”	instruction = “ ”
EC_N3	(Rb,Rc,Ra) = (65535,0,0)	(Rb,Rc,Ra) = (65535,-1,0) (Rb,Rc,Ra) = (65535,1,0)
EC_N4	(b,v,b + v) = (0,0,0)	(b,v,b + v) = (-1,0,-1)
EC_N5	(b,v,b + v) = (65535,0,65535)	(b,v,b + v) = (65535,1,65536)
EC_N6	v = 1	v = 2

EC_V4, EC_V5, EC_V6, EC_V7

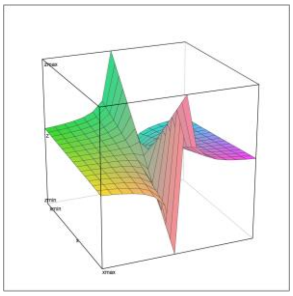
The formulas (Ra = Rb + Rc) and (Ra = Rb - Rc) implies a plane in 3 dimensions. Likewise, (Ra = Rb * Rc) and (Ra = Rb / Rc) implies a hyperbola. Choosing the maximum and minimum boundary values for Rb and Rc [-65535, 65535] gives Ra a range of [-131070, 131070] .



Ra = Rb * Rc results in a hyperbola, with Ra having a range of [-4294836225, 4294836225], which should overflow the 32 bit Java int value ("Online 3-D Function Grapher", 2018).

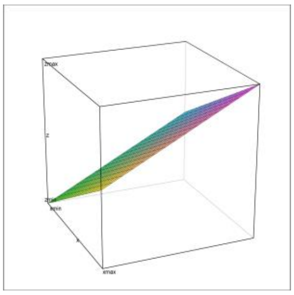


Ra = Rb / Rc results in a stranger hyperbolic shape, with Ra having a range of [-15, 15]. Division is not defined for Rc = 0, and should be treated as a no-op by the program.



EC_V9

b + v >= 0 and b + v <= 65535 are planes in 3 dimensions, with (b + v) having a range of [0, 65535]. Likewise, for EC_V10, b + v < N – 1 is a dynamic plane with respect to the length of instructions, N. b + v = 0 is not testable because jumping back to zero without encountering a return results in an infinite loop ("Online 3-D Function Grapher", 2018).



Task 5

Condition	Objectives	PartitionTests	BoundaryTests
True	2	2	2
pc < 0 || pc >= progLength	4	3	4
inst.equals("")	2	2	2
toks.length < 2	2	1	1
tok[0].equals(INSTRUCTION_ADD)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_SUBTRACT)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_MULT)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_DIVIDE)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_RETURN)	2	2	2
tok[0].equals(INSTRUCTION_LOAD)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_STORE)	2	2	2
toks.length != 4	2	1	1
tok[0].equals(INSTRUCTION_MOVE)	2	2	2
toks.length != 3	2	1	1
tok[0].equals(INSTRUCTION_JUMP)	2	2	2
toks.length != 2	2	1	1
tok[0].equals(INSTRUCTION_JZ)	2	2	2
toks.length != 3	2	1	1
regs[ra] == 0	2	0	2
Total Objectives	**49**	**37**	**40**

Test Suite	Coverage Score
Equivalence partitioning	37 / 49 = 75.51%
Boundary-value analysis	40 / 49 = 81.63%

$$Coverage\ Score = \frac{Objectives\ Met}{Total\ Objectives}$$

(*Software Testing and Reliability: Course Notes for SWEN90006*, 2018)

Question 7

Based on the multiple-condition calculations done in Task 5, and due to boundary-value analysis being applied as an additional layer on top of equivalence partitioning, it achieves a slightly higher score (81.63%) coverage versus equivalence partitioning (75.51%).

The difference can be seen in objective scores for some of the condition statements, namely (pc < 0 || pc >= progLength), or more generally conditions that include predicate operators like && and ||. The higher number of predicates contained in a condition generally results in a linear increase in equivalence classes, but an exponential increase in boundary values, which leads to the creation of more tests.

Another example would be (regs[ra] == 0). In this case, I have not determined a proper equivalence class to test this condition. However, due to the nature of integer equivalence, a boundary value is present and on and off-points can be tested.

The input domain also plays a role, as sub domains can be derived from it. From the list of strings of instructions, the splitting and tokenisation of strings into further lists and primitive values generates more and more equivalence classes and boundary values. However, equivalence classes combinations are limited by the single inheritance property, while boundary values grow exponentially with each level.

Boundaries for equivalence classes may not be easy to detect, as they may not exist or make sense. However, when they are discovered, they provide better test cases than equivalence partitioning. (1)

References

The School of Computing and Software Systems, The University of Melbourne. (2018). *Software Testing and Reliability: Course Notes for SWEN90006*. Melbourne.

The University of Melbourne. (2018). *SWEN90006: Software Testing and Reliability: Assignment 1* [Ebook]. Melbourne. Retrieved from https://app.lms.unimelb.edu.au/bbcswebdav/pid-6594864-dt-content-rid-45442915_2/courses/SWEN90006_2018_SM2/Assignment_1%2810%29.pdf

Backus–Naur form. (2018). Retrieved from https://en.wikipedia.org/wiki/Backus–Naur_form

Miller, T. (2018). Tim Miller / SWEN90006-A1-2018. Retrieved from https://gitlab.eng.unimelb.edu.au/tmiller/SWEN90006-A1-2018

ASCII Table | ASCII character codes table. (2018). Retrieved from https://www.rapidtables.com/code/text/ascii-table.html

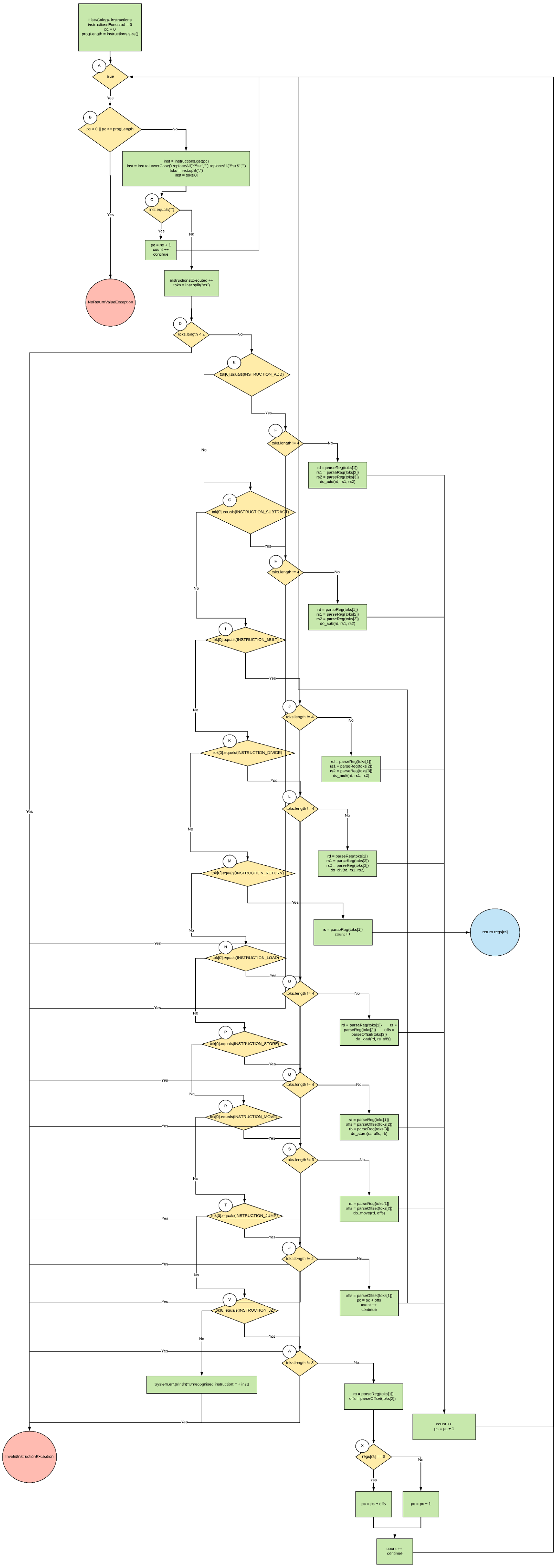
Online Diagram Software & Visual Solution | Lucidchart. (2018). Retrieved from https://www.lucidchart.com

Online 3-D Function Grapher. (2018). Retrieved from http://www.livephysics.com/tools/mathematical-tools/online-3-d-function-grapher/

Appendix

Control-flow graph for

the execute() method


List<String> instructions
instructionsExecuted = 0
pc = 0
progLength = instructions.size()
A
true
Yes
B
pc < 0 || pc >= progLength
No
inst = instructions.get(pc)
inst = inst.toLowerCase().replaceAll("^\\s+","").replaceAll("\\s+$","")
toks = inst.split(";")
inst = toks[0]
Yes
NoReturnValueException
C
inst.equals("")
Yes
No
pc = pc + 1
count ++
continue
instructionsExecuted ++
toks = inst.split("\\s")
D
toks.length < 2
No
Yes
E
tok[0].equals(INSTRUCTION_ADD)
Yes
No
F
toks.length != 4
No
rd = parseReg(toks[1])
rs1 = parseReg(toks[2])
rs2 = parseReg(toks[3])
do_add(rd, rs1, rs2)
G
tok[0].equals(INSTRUCTION_SUBTRACT)
Yes
No
H
toks.length != 4
No
rd = parseReg(toks[1])
rs1 = parseReg(toks[2])
rs2 = parseReg(toks[3])
do_sub(rd, rs1, rs2)
I
tok[0].equals(INSTRUCTION_MULT)
Yes
No
J
toks.length != 4
No
rd = parseReg(toks[1])
rs1 = parseReg(toks[2])
rs2 = parseReg(toks[3])
do_mult(rd, rs1, rs2)
K
tok[0].equals(INSTRUCTION_DIVIDE)
Yes
No
L
toks.length != 4
No
rd = parseReg(toks[1])
rs1 = parseReg(toks[2])
rs2 = parseReg(toks[3])
do_div(rd, rs1, rs2)
M
tok[0].equals(INSTRUCTION_RETURN)
Yes
No
rs = parseReg(toks[1])
count ++
return regs[rs]
N
tok[0].equals(INSTRUCTION_LOAD)
Yes
No
O
toks.length != 4
No
rd = parseReg(toks[1]) rs = parseReg(toks[2]) offs = parseOffset(toks[3])
do_load(rd, rs, offs)
P
tok[0].equals(INSTRUCTION_STORE)
Yes
No
Q
toks.length != 4
No
ra = parseReg(toks[1])
offs = parseOffset(toks[2])
rb = parseReg(toks[3])
do_store(ra, offs, rb)
R
tok[0].equals(INSTRUCTION_MOVE)
Yes
No
S
toks.length != 3
No
rd = parseReg(toks[1])
offs = parseOffset(toks[2])
do_move(rd, offs)
T
tok[0].equals(INSTRUCTION_JUMP)
Yes
No
U
toks.length != 2
No
offs = parseOffset(toks[1])
pc = pc + offs
count ++
continue
V
tok[0].equals(INSTRUCTION_JZ)
Yes
No
W
toks.length != 3
No
System.err.println("Unrecognised instruction: " + inst)
ra = parseReg(toks[1])
offs = parseOffset(toks[2])
count ++
pc = pc + 1
X
regs[ra] == 0
Yes
No
pc = pc + offs
pc = pc + 1
count ++
continue
InvalidInstructionException