# Software Testing and Reliability

Semester 2, 2018

## Contents

# 1 Test Template Tree

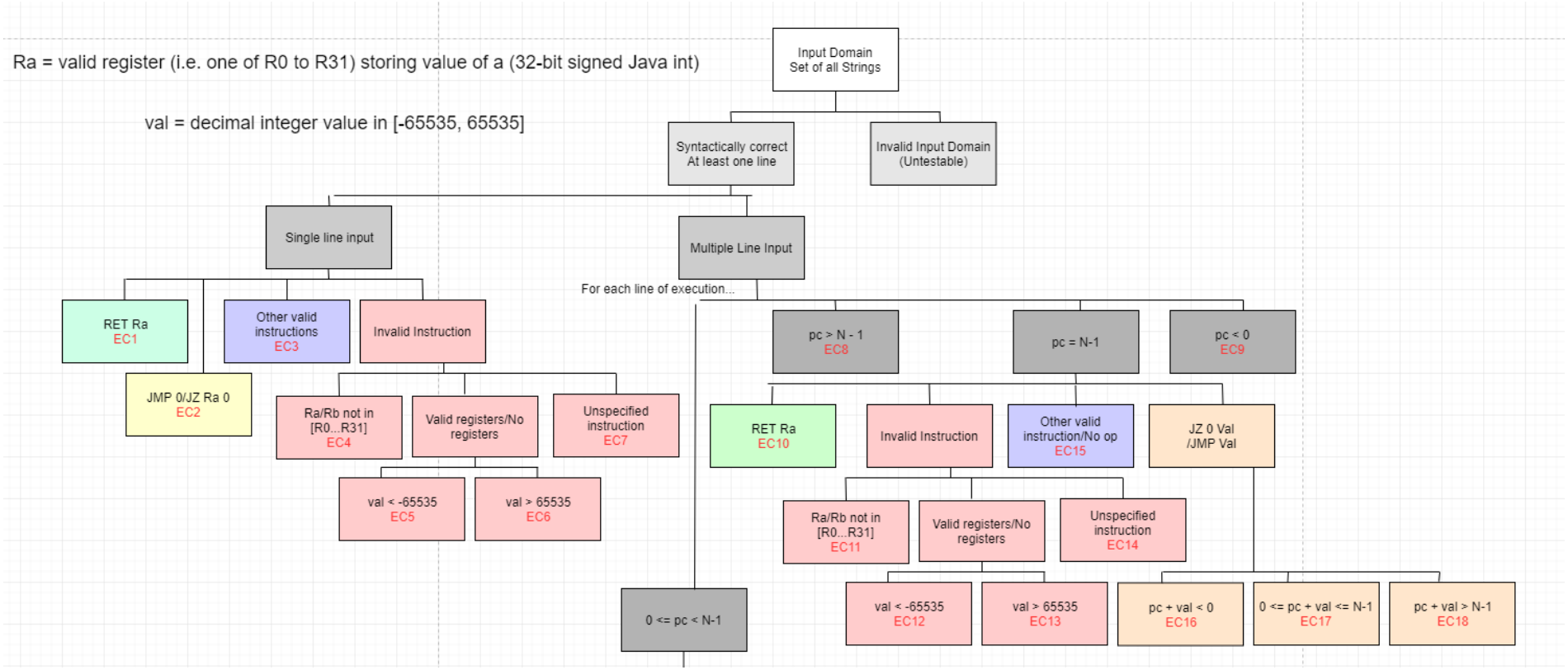


Figure 1: Test Template Tree Part 1

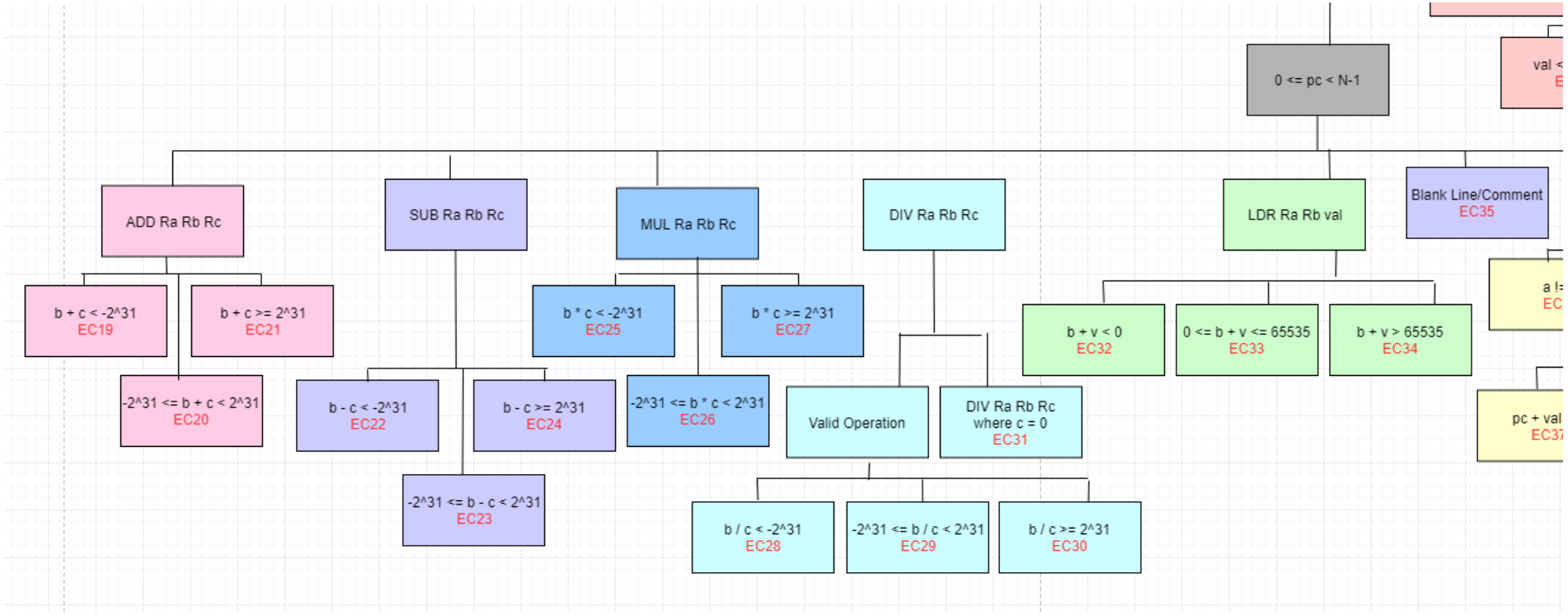


Figure 2: Test Template Tree Part 2

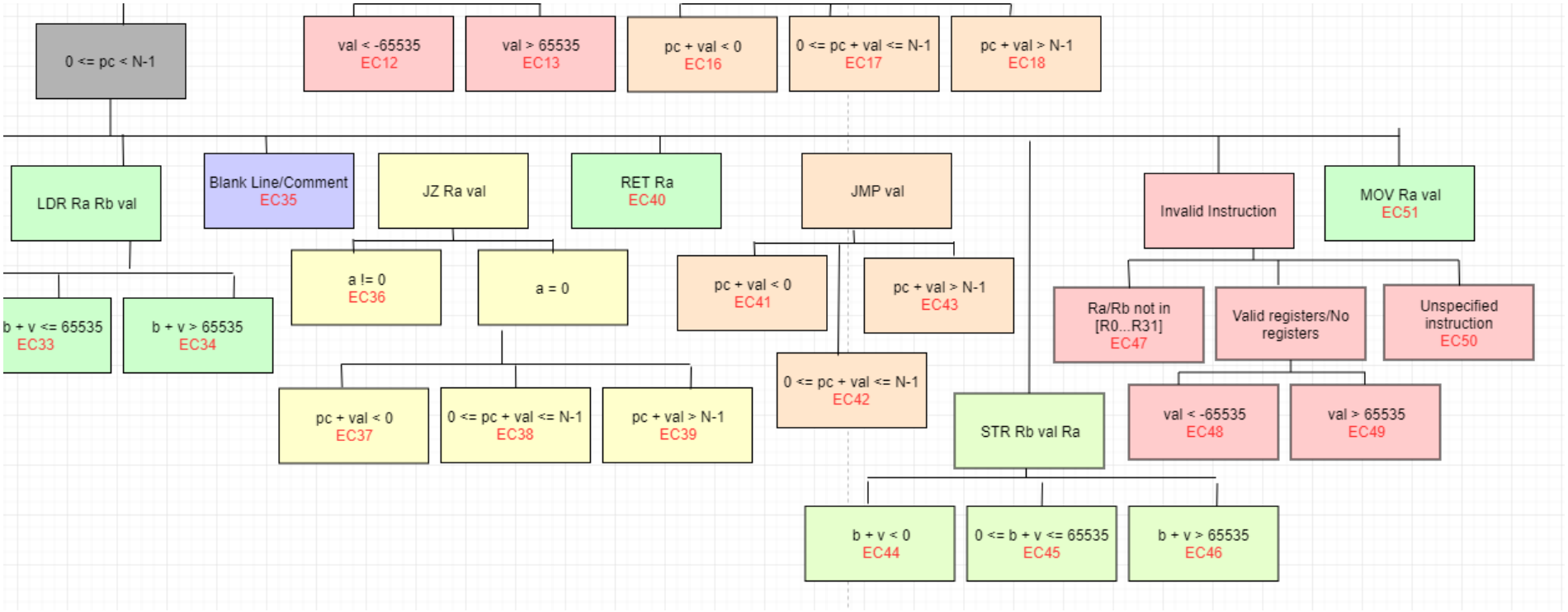


Figure 3: Test Template Tree Part 3

# 2 Task One - Equivalence Classes

As it is safely assumed that input programs will contain at least 1 line, we can decompose the valid input domain into single and multiple line inputs. These new nodes are further decomposed in a similar fashion while taking into consideration the possible values that may be taken and following the provided guidelines for equivalence class partitioning. Consequently, the resulting set of equivalence classes cover the input domain.

# 3 Task Two - Equivalence Class Test Cases

The Equivalence Partitioning tests and their corresponding Test IDs are listed below, together with their associated equivalence classes.

| Test ID | Test Name | Equivalence Classes |
|---|---|---|
| 1 | singleLineValidInstruction | EC3 |
| 2 | overflowAdd2 | EC19 |
| 3 | validAdd | EC20 |
| 4 | overflowAdd1 | EC21 |
| 5 | overflowSub1 | EC24 |
| 6 | validSub | EC23 |
| 7 | overflowSub2 | EC22 |
| 8 | overflowMul1 | EC27 |
| 9 | validMul | EC26 |
| 10 | overflowMul2 | EC25 |
| 11 | validDiv | EC29 |
| 12 | invalidDiv | EC31 |
| 13 | blankLineComment | EC35 |
| 14 | earlyReturn | EC40 |
| 15 | validSTRLDR | EC33, EC45 |
| 16 | invalidSTRLDR1 | EC34, EC46 |
| 17 | invalidSTRLDR2 | EC32, EC44 |
| 18 | validMOV | EC51 |
| 19 | validJZ0 | EC38 |
| 20 | validJZ1 | EC36 |
| 21 | validJMP | EC42 |
| 22 | invalidJMP1 | EC43 |
| 23 | invalidJMP2 | EC41 |
| 24 | invalidJZ01 | EC37 |
| 25 | invalidJZ02 | EC39 |
| 26 | invalidRegister | EC47 |
| 27 | invalidVal1 | EC49 |
| 28 | invalidVal2 | EC48 |
| 29 | invalidInstruction | EC50 |
| 30 | noReturn | EC15 |

Table 1: Equivalence Class Partitioning Test Suite

# 4 Task Three - Boundary Value Analysis

## 4.1 Number of Line Inputs

As the number of line inputs is defined by the input file, we can use the zero-one-many rule to decompose the input domain. As mentioned above, it is assumed that the number of line inputs is greater than 0, so we can consider empty input files as invalid, leaving input sizes of 1 and many remaining. We can determine that, from the perspective of the single line input equivalence class, an input size of 1 is considered on point, and 2 is off point.

No test cases have been directly derived from this as the test suite derived from other classes and boundaries already contain tests that consider these on and off points for number of input lines.

## 4.2 Valid Registers and Valid Values

If the valid registers are considered to be a range from R0..R31, 'R31' and 'R0' would be on point, while 'R32' and 'R-1' are off point.

If a val is involved in the instruction, its valid domain is [-65535, 65535]. Thus, -65536 and 65536 are off point, and -65535 and 65535 are on point.

The assumption has also been made that the validity of registers and values will be done in the same process in both single and multiple line inputs, and as a result some of the derived test cases will only test the validity within single line inputs.

The test cases derived from these will contain:

**Derived Test Cases**
1. RET R-1
2. RET R0
3. RET R31
4. RET R32
5. MOV R31 65535
6. MOV R31 65536
7. MOV R0 65535
8. MOV R0 -65535
9. MOV R32 65536
10. MOV R31 -65535
11. MOV R31 -65536
12. MOV R32 -65536

## 4.3 JMP/JZ and Program Counter

The program counter is restricted between 0 and N-1, where N is the number of lines of input. Thus -1 and N are off point, and 0 and N-1 are on point.

For the instruction JZ to change the program counter, it requires a val of 0. As val can take values between [-65535, 65535], 0 is considered on point, and -1 and 1 are considered off point.

The test cases derived from these will be:

**Derived Test Cases**
1. JMP to pc = 0
2. JMP to pc = -1
3. JMP to pc = N-1
4. JMP to pc = N
5. JZ 0 to pc = 0
6. JZ 0 to pc = -1
7. JZ 0 to pc = N-1
8. JZ 0 to pc = N
9. JZ -1 to pc = N-1
10. JZ 1 to pc = N-1

## 4.4 Mathematical Operations and Register Values

A register stores a 32-bit signed Java integer, meaning it can hold the value within $[-2^{31}, 2^{31})$. Thus, $-2^{31}$ and $2^{31} - 1$ are on point, while $-2^{31} - 1$ and $2^{31}$ are off point.

EC31, which handles divide by zero invalid operations, requires Rc to hold a value of 0. Thus, -1 and 1 are off point, and 0 is on point for c.

The test cases derived from these will be:

**Derived Test Cases**
1. DIV Ra Rb Rc where c = 0
2. DIV Ra Rb Rc where c = 1
3. DIV Ra Rb Rc where c = -1
4. ADD Ra Rb Rc where b = 2147483646 and c= 1
5. ADD Ra Rb Rc where b = 2147483646 and c = 2
6. ADD Ra Rb Rc where b = -2147483646 and c = -2
7. ADD Ra Rb Rc where b = -2147483646 and c = -3
8. SUB Ra Rb Rc where b = 2147483646 and c = -1
9. SUB Ra Rb Rc where b = 2147483646 and c = -2
10. SUB Ra Rb Rc where b = -2147483646 and c = 2
11. SUB Ra Rb Rc where b = -2147483646 and c = 3
12. MUL Ra Rb Rc where b = 1073741823 and c = 2
12. MUL Ra Rb Rc where b = 1073741824 and c = 2
13. MUL Ra Rb Rc where b = -1073741824 and c = 2
14. MUL Ra Rb Rc where b = -1073741825 and c = 2

## 4.5 Reading and Writing

The address space is limited to values within [0, 65535]. Thus, 0 and 65535 are on point, and -1 and 65536 are off point. The test cases derived from these will involve:

**Derived Test Cases**
1. STR Rb val Ra where b + v = -1 && LDR Ra Rb val where b + v = -1 (b = -1 val = 0)
2. STR Rb val Ra where b + v = -1 && LDR Ra Rb val where b + v = -1 (b = 0 val = -1)
3. STR Rb val Ra where b + v = 0 && LDR Ra Rb val where b + v = 0 (b = 0 val = 0)
4. STR Rb val Ra where b + v = 65535 && LDR Ra Rb val where b + v = 65535 (b = 65535 val = 0)
5. STR Rb val Ra where b + v = 65535 && LDR Ra Rb val where b + v = 65535 (b = 0 val = 65535)
6. STR Rb val Ra where b + v = 65536 && LDR Ra Rb val where b + v = 65536 (b = 65535 val = 1)
7. STR Rb val Ra where b + v = 65536 && LDR Ra Rb val where b + v = 65536 (b = 1 val = 65535)

## 4.6 Discrete Equivalence Classes

In addition to the specified on and off points above concerning ranges and other values, discrete equivalent classes will be tested as well.

The test cases derived from these will involve:

**Derived Test Cases**
1. Empty lines and lines with only comments
2. Executing RET before final line in program
3. No RET in program
4. Invalid instruction (RAT R0)

# 5 Task Four - Boundary Value Analysis Test Suite

| Test ID | Tests | Boundaries/Classes |
|---|---|---|
| 1 | invalidLowerRegister | EC4, EC11, EC47 |
| 2 | validRegister | EC4, EC11, EC47 |
| 3 | invalidHigherRegister | EC4, EC11, EC47 |
| 4 | validMov | EC51, EC47, EC48, EC49 |
| 5 | invalidMov1 | EC51, EC49 |
| 6 | invalidMov2 | EC51, EC48 |
| 7 | invalidMov3 | EC51, EC49, EC47 |
| 8 | invalidMov4 | EC51, EC48, EC47 |
| 9 | invalidInstruction | EC50 |
| 10 | validJMP | EC42, EC41, EC43 |
| 11 | invalidJMP1 | EC42, EC41 |
| 12 | invalidJMP2 | EC42, EC43 |
| 13 | validJZ | EC36, EC38, EC37, EC39 |
| 14 | invalidJZ1 | EC38, EC37 |
| 15 | invalidJZ2 | EC38, EC39 |
| 16 | validDiv | EC29, EC31 |
| 17 | validAdd | EC20, EC19, EC21 |
| 18 | overflowAdd1 | EC20, EC21 |
| 19 | overflowAdd2 | EC20, EC19 |
| 20 | validSub | EC23, EC22, EC24 |
| 21 | overflowSub1 | EC23, EC24 |
| 22 | overflowSub2 | EC23, EC22 |
| 23 | validMul | EC26, EC25, EC27 |
| 24 | overflowMul1 | EC26, EC27 |
| 25 | overflowMul2 | EC26, EC25 |
| 26 | validLDRSTR | EC33, EC45, EC32, EC44, EC34, EC46 |
| 27 | uppLDRSTR | EC33, EC45, EC34, EC46 |
| 28 | negLDRSTR | EC33, EC45, EC32, EC44 |
| 29 | earlyReturn | EC40 |
| 30 | blankLineComment | EC35 |
| 31 | noReturn | EC15 |

Table 2: Boundary Value Test Suite

# 6 Task Five - Multiple Condition Coverage

The execute method contains:

- 1 if statement with 2 conditions
- 12 if statements with 1 condition
- 1 if-else chain with 11 different cases

From this we determine:

- $2 + 12 + 1 = 15$ conditions
- $2^2 + 12 * 2 + 11 = 39$ condition permutations

| Condition | Branch Code | Permutations |
|---|---|---|
| **C1 \|\| C2** | pc <0 \|\| pc >= progLength | {00, 01, 10, 11} |
| **C3** | if (inst.equals(“”)) | {0, 1} |
| **C4** | if (toks.length <2) | {0, 1} |
| **C5** | If else chain beginning with:<br>if (toks[0].equals(INSTRUCTION_ADD)) | {ADD, SUB, MUL, DIV, RET, LDR, STR, MOV, JMP, JZ, default} |
| **C6** | (ADD) if (toks.length != 4) | {0, 1} |
| **C7** | (SUB) if (toks.length != 4) | {0, 1} |
| **C8** | (MUL) if (toks.length != 4) | {0, 1} |
| **C9** | (DIV) if (toks.length != 4) | {0, 1} |
| **C10** | (LDR) if (toks.length != 4) | {0, 1} |
| **C11** | (STR) if (toks.length != 4) | {0, 1} |
| **C12** | (MOV) if (toks.length != 3) | {0, 1} |
| **C13** | (JMP) if (toks.length != 2) | {0, 1} |
| **C14** | (JZ) if (toks.length != 3) | {0, 1} |
| **C15** | (JZ) if (regs[ra] == 0) | {0, 1} |

Table 3: Multiple Condition Coverage Table for execute method

## 6.1 Equivalence Partitioning

As shown in Table 4 and 5, 11 out of 39 objectives were unmet. One of the objectives {C1,C2} = {11} is impossible to meet - the program counter cannot be simultaneously larger than or equal to progLength (number of lines in the program), and be less than 0, as progLength cannot take negative values.

The remaining unmet objectives relate to the syntax of the grammar of the provided instructions. As it was assumed that the provided programs would be syntactically correct, the chosen test cases do not account for these token length conditions.

The multiple condition coverage can be calculated with:

$$\frac{Number\ of\ Unmet\ Objectives}{Total\ Number\ of\ Objectives} = \frac{28}{39} = 72\%$$

| Test ID | {C1, C2} | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | {00}{01} | 0 | 0 | MOV | | | | | | | 0 | | | |
| **2** | {00} | 0 | 0 | MOV, ADD, MUL, RET | 0 | | 0 | | | | 0 | | | |
| **3** | {00} | 0 | 0 | MOV, ADD, RET | 0 | | | | | | 0 | | | |
| **4** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **5** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **6** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **7** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **8** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **9** | {00} | 0 | 0 | MOV, MUL, RET | | | 0 | | | | 0 | | | |
| **10** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **11** | {00} | 0 | 0 | MOV, DIV, RET | | | | 0 | | | 0 | | | |
| **12** | {00} | 0, 1 | 0 | MOV, DIV, RET | | | | 0 | | | 0 | | | |
| **13** | {00} | 1 | 0 | MOV, RET | | | | | | | 0 | | | |
| **14** | {00} | 0 | 0 | MOV, ADD, RET | 0 | | | | | | 0 | | | |
| **15** | {00} | 0, 1 | 0 | MOV, SUB, JZ, STR, ADD, JMP, LDR, RET | 0 | 0 | | | 0 | 0 | 0 | 0 | 0 | 0,1 |
| **16** | {00} | 0 | 0 | MOV, STR, LDR, RET | | | | | 0 | 0 | 0 | | | |
| **17** | {00} | 0 | 0 | MOV, STR, LDR, RET | | | | | 0 | 0 | 0 | | | |
| **18** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **19** | {00} | 0 | 0 | MOV, JZ, RET | | | | | | | 0 | | 0 | 1 |
| **20** | {00} | 0 | 0 | MOV, JZ, RET | | | | | | | 0 | | 0 | 0 |
| **21** | {00} | 0 | 0 | MOV, JMP, RET | | | | | | | 0 | 0 | | |
| **22** | {00}{01} | 0 | 0 | MOV, JMP, RET | | | | | | | 0 | 0 | | |
| **23** | {00}{10} | 0 | 0 | MOV, JMP, RET | | | | | | | 0 | 0 | | |
| **24** | {00}{10} | 0 | 0 | MOV, JZ, RET | | | | | | | 0 | | 0 | 1 |
| **25** | {00}{01} | 0 | 0 | MOV, JZ, RET | | | | | | | 0 | | 0 | 1 |
| **26** | {00} | 0 | 0 | MOV | | | | | | | 0 | | | |
| **27** | {00} | 0 | 0 | MOV | | | | | | | 0 | | | |
| **28** | {00} | 0 | 0 | MOV | | | | | | | 0 | | | |
| **29** | {00} | 0 | 0 | MOV, default | | | | | | | 0 | | | |
| **30** | {00} {01} | 0 | 0 | MOV, ADD | 0 | | | | | | 0 | | | |

Table 4: Equivalence Partitioning Multiple Condition Coverage

| Condition | {C1C2} | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Seen** | {00}, {01}, {10} | 0, 1 | 0 | MOV, ADD, SUB, MUL, DIV, LDR, STR, RET, JMP, JZ, default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0,1 |
| **Missing** | {11} | | 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

Table 5: Equivalence Partitioning Multiple Condition Coverage Objectives

## 6.2 Boundary Value Analysis

As demonstrated in Table 6 and 7, 11 out of 39 objectives were unmet. One of the objectives {C1,C2} = {11} is impossible to meet - the program counter cannot be simultaneously larger than or equal to progLength (number of lines in the program), and be less than 0, as progLength cannot take negative values.

The remaining unmet objectives relate to the syntax of the grammar of the provided instructions. As it was assumed that the provided programs would be syntactically correct, the chosen test cases do not account for these token length conditions.

The multiple condition coverage can be calculated with:

$$\frac{Number\ of\ Unmet\ Objectives}{Total\ Number\ of\ Objectives} = \frac{28}{39} = 72\%$$

| Test ID | {C1, C2} | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | {00} | 0 | 0 | RET | | | | | | | | | | |
| **2** | {00} | 0 | 0 | RET | | | | | | | | | | |
| **3** | {00} | 0 | 0 | RET | | | | | | | | | | |
| **4** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **5** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **6** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **7** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **8** | {00} | 0 | 0 | MOV, RET | | | | | | | 0 | | | |
| **9** | {00} | 0 | 0 | default | | | | | | | | | | |
| **10** | {00} | 0 | 0 | MOV, JZ,JMP, RET | | | | | | | 0 | 0 | 0 | 0, 1 |
| **11** | {00}{10} | 0 | 0 | MOV, JZ, JMP | | | | | | | 0 | 0 | 0 | 1 |
| **12** | {00}{01} | 0 | 0 | JMP | | | | | | | | 0 | | |
| **13** | {00} | 0 | 0 | JZ, MOV, RET | | | | | | | 0 | | 0 | 0, 1 |
| **14** | {00}{10} | 0 | 0 | JZ, MOV | | | | | | | 0 | | 0 | 1 |
| **15** | {00}{01} | 0 | 0 | JZ | | | | | | | | | 0 | 1 |
| **16** | {00} | 0 | 0 | MOV, DIV, RET | | | | 0 | | | 0 | | | |
| **17** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **18** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **19** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **20** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **21** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **22** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **23** | {00} | 0 | 0 | MOV, MUL, RET | | | 0 | | | | 0 | | | |
| **24** | {00} | 0 | 0 | MOV, MUL, ADD, RET | 0 | | 0 | | | | 0 | | | |
| **25** | {00} | 0 | 0 | MOV, MUL, SUB, RET | | 0 | 0 | | | | 0 | | | |
| **26** | {00} | 0 | 0 | MOV, STR, LDR, RET | | | | | 0 | 0 | 0 | | | |
| **27** | {00} | 0 | 0 | MOV, ADD, STR, LDR, RET | 0 | | | | 0 | 0 | 0 | | | |
| **28** | {00} | 0 | 0 | MOV, STR, LDR, RET | | | | | 0 | 0 | 0 | | | |
| **29** | {00} | 0 | 0 | MOV, ADD, RET | 0 | | | | | | 0 | | | |
| **30** | {00} | 0, 1 | 0 | MOV, RET | | | | | | | 0 | | | |
| **31** | {00}{01} | 0 | 0 | MOV, ADD | 0 | | | | | | 0 | | | |

Table 6: Boundary Value Analysis Multiple Condition Coverage

| Condition | {C1C2} | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Seen** | {00}, {01}, {10} | 0, 1 | 0 | MOV, ADD, SUB, MUL, DIV, LDR, STR, RET, JMP, JZ, default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0,1 |
| **Missing** | {11} | | 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

Table 7: Boundary Value Analysis Multiple Condition Coverage Objectives

# 7 Task Seven - Comparison

## 7.1 Input/Output Domain Coverage

As both equivalence partitioning and boundary value analysis are derived from equivalence classes, both techniques represent the input domain in a very similar fashion. As long as the derived equivalence classes are disjoint and span the input domain, the resulting input domain coverage for the two sets of test cases should be adequate.

## 7.2 Multiple Condition Coverage

The resulting multiple condition coverage for both sets of test cases were the same - 72%. As the remaining unmet objectives were either impossible, or not considered within the scope of our assumed inputs, we can observe that both equivalence partitioning and boundary value analysis performed quite well under this particular metric.

## 7.3 Mutants

Due to the nature of how mutants are generated, boundary value analysis outperforms equivalence partitioning in this regard. Mutations which modify constants or inequalities by small amounts are much more likely to be caught through boundary value analysis, rather than equivalence partitioning, which treats all inputs in a particular equivalence class as equally as valuable. These types of mutations are also representative of some of the more common errors that programmers can find themselves making.

Additionally, as the technique of utilising on and off points to derive test cases are still based upon the original equivalence classes, the benefits of using equivalence partitioning are not lost from using boundary value analysis.

## 7.4 Summary

Based on these evaluation metrics and results, it was determined that boundary value analysis was more valuable when compared to equivalence partitioning. Rather than two entirely separate techniques of deriving a set of test cases, boundary value analysis can be considered to be an extension of equivalence partitioning. Although the resulting tests may be more complicated to create, based on its performance on my own generated mutants, boundary value analysis provides a more sophisticated set of test cases.