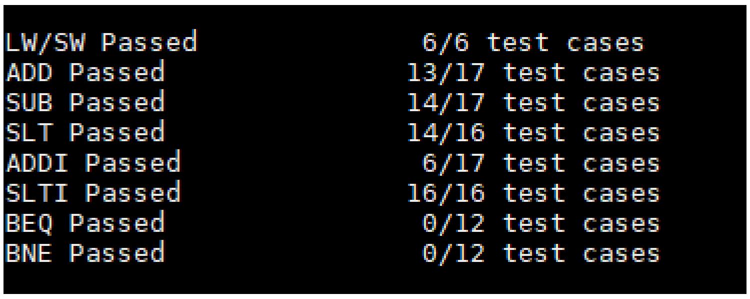
CoE 113 ME3

1 Instruction set test results

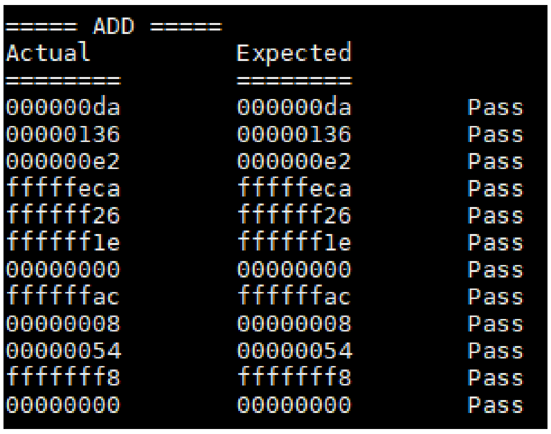
The results of the test are shown below.



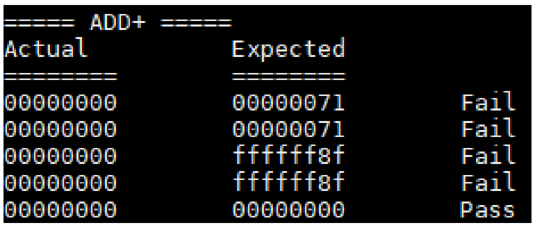
```
LW/SW Passed                6/6 test cases
ADD Passed                 13/17 test cases
SUB Passed                 14/17 test cases
SLT Passed                 14/16 test cases
ADDI Passed                 6/17 test cases
SLTI Passed                16/16 test cases
BEQ Passed                  0/12 test cases
BNE Passed                  0/12 test cases
```

2 Erroneous intructions

For the ADD instruction (ADD+ in this case just means additional test cases that were added later), the first 12 cases had only non-zero inputs, while the five additional cases had at least one zero input. For the five additional cases, all the outputs were zero, which means that whenever at least one input is zero, ADD immediately outputs a zero.



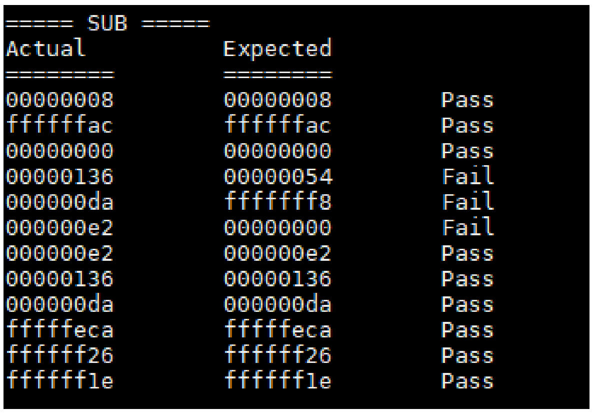
```
===== ADD =====
Actual          Expected
========        ========
000000da        000000da        Pass
00000136        00000136        Pass
000000e2        000000e2        Pass
fffffeca        fffffeca        Pass
ffffff26        ffffff26        Pass
ffffff1e        ffffff1e        Pass
00000000        00000000        Pass
ffffffac        ffffffac        Pass
00000008        00000008        Pass
00000054        00000054        Pass
fffffff8        fffffff8        Pass
00000000        00000000        Pass
```



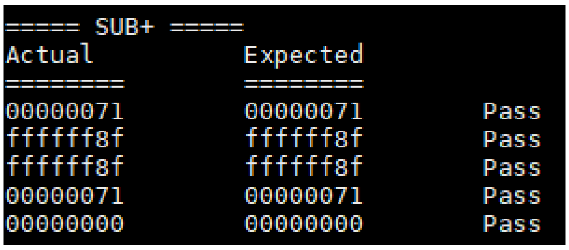
```
===== ADD+ =====
Actual          Expected
========        ========
00000000        00000071        Fail
00000000        00000071        Fail
00000000        ffffff8f        Fail
00000000        ffffff8f        Fail
00000000        00000000        Pass
```

For the SUB instruction, it gave a wrong output for the test cases where both inputs were negative. For example, the first failed case was -113 –(-197) = 84 or 0x54. However, the output was 0x136 or 310, which is the result of 113 + 197. The same thing happened with the next two cases.

This error may be caused by the instruction being implemented to check if both inputs are negative. If they are, the first input is turned positive, then subtracted with the 2nd input. So instead of the instruction doing $-X - (-Y)$, it does $X - (-Y)$. For the other test cases, this did not happen.

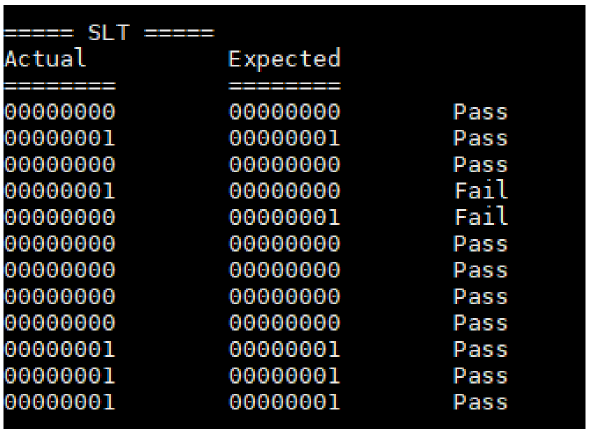


```
===== SUB =====
Actual          Expected
========        ========
00000008        00000008        Pass
ffffffac        ffffffac        Pass
00000000        00000000        Pass
00000136        00000054        Fail
000000da        fffffff8        Fail
000000e2        00000000        Fail
000000e2        000000e2        Pass
00000136        00000136        Pass
000000da        000000da        Pass
fffffeca        fffffeca        Pass
ffffff26        ffffff26        Pass
ffffff1e        ffffff1e        Pass
```

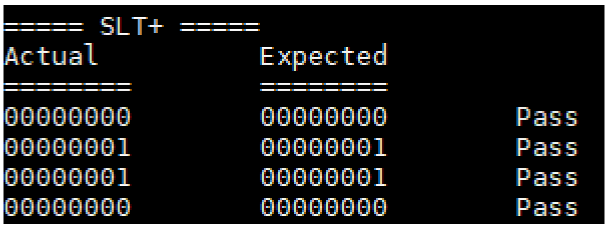


```
===== SUB+ =====
Actual          Expected
========        ========
00000071        00000071        Pass
ffffff8f        ffffff8f        Pass
ffffff8f        ffffff8f        Pass
00000071        00000071        Pass
00000000        00000000        Pass
```

For the SLT instruction, errors happened with the test cases where both inputs were negative and not equal. The errors were similar to the errors in SUB. When both inputs are negative, they are first turned positive before being compared, which results in the output being the opposite of what is expected. Note that when the inputs are both negative and equal, the output is still correct.

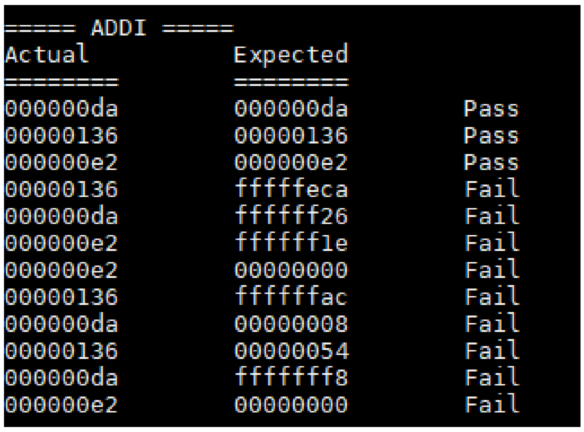


```
===== SLT =====
Actual          Expected
========        ========
00000000        00000000        Pass
00000001        00000001        Pass
00000000        00000000        Pass
00000001        00000000        Fail
00000000        00000001        Fail
00000000        00000000        Pass
00000000        00000000        Pass
00000000        00000000        Pass
00000000        00000000        Pass
00000001        00000001        Pass
00000001        00000001        Pass
00000001        00000001        Pass
```

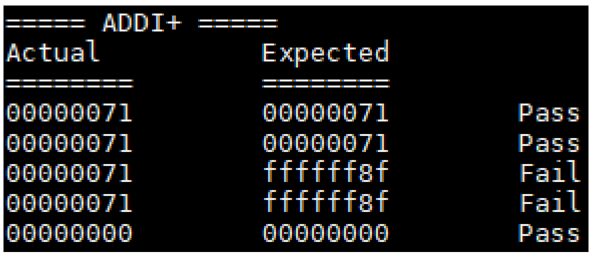


```
===== SLT+ =====
Actual          Expected
========        ========
00000000        00000000        Pass
00000001        00000001        Pass
00000001        00000001        Pass
00000000        00000000        Pass
```

For ADDI, the errors happened when one of the inputs were negative. For the first three cases, both inputs are positive, so the outputs are as expected. For the next 9 cases, however, if an input is negative, it is turned positive first before being added. So, if both inputs are negative, both of them are turned positive before being added. The same thing happened in the five added cases under ADDI+. For the two failed cases, one of the inputs were negative, which resulted in it being turned positive first before being added.



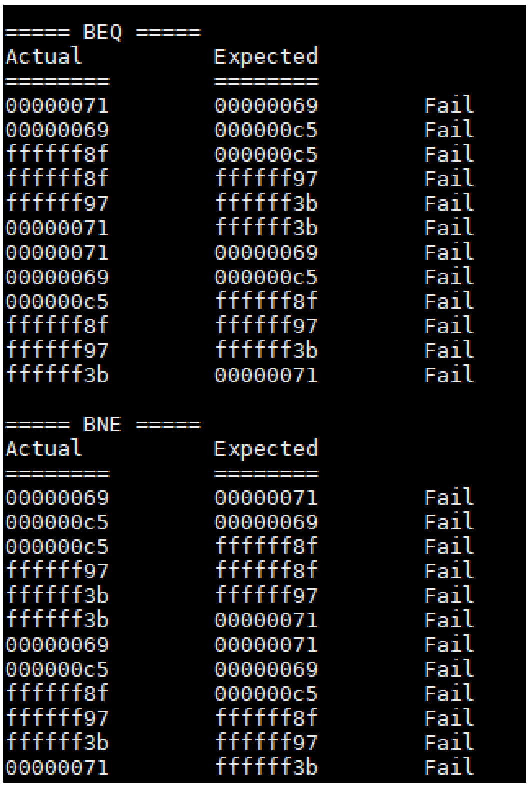
```
===== ADDI =====
Actual          Expected
========        ========
000000da        000000da        Pass
00000136        00000136        Pass
000000e2        000000e2        Pass
00000136        fffffeca        Fail
000000da        ffffff26        Fail
000000e2        ffffff1e        Fail
000000e2        00000000        Fail
00000136        ffffffac        Fail
000000da        00000008        Fail
00000136        00000054        Fail
000000da        fffffff8        Fail
000000e2        00000000        Fail
```



```
===== ADDI+ =====
Actual          Expected
========        ========
00000071        00000071        Pass
00000071        00000071        Pass
00000071        ffffff8f        Fail
00000071        ffffff8f        Fail
00000000        00000000        Pass
```

For BEQ and BNE, the errors happened on all test cases. In the test assembly code, a value was first stored into data memory, then BEQ/BNE was called, then another value was stored on the same data memory address as the first, which would overwrite the first data stored if the branch does not succeed. This means that for BEQ, if the inputs are equal, the 2nd SW call should be skipped. Similarly, for BNE, if the inputs aren't equal, the 2nd SW call is skipped.

For the first BEQ test case, the inputs aren't equal, so the 2nd SW call should be executed, which stores 0x69 in memory. For the third test case, the inputs are equal, so the 2nd SW call should be skipped, which means that 0xffffff8f isn't overwritten in memory. Following this, it is clear that the opposite of what is expected is happening for all test cases, which means that the implementations of BEQ and BNE were swapped.



```
===== BEQ =====
Actual          Expected
========        ========
00000071        00000069        Fail
00000069        000000c5        Fail
ffffff8f        000000c5        Fail
ffffff8f        ffffff97        Fail
ffffff97        ffffff3b        Fail
00000071        ffffff3b        Fail
00000071        00000069        Fail
00000069        000000c5        Fail
000000c5        ffffff8f        Fail
ffffff8f        ffffff97        Fail
ffffff97        ffffff3b        Fail
ffffff3b        00000071        Fail

===== BNE =====
Actual          Expected
========        ========
00000069        00000071        Fail
000000c5        00000069        Fail
000000c5        ffffff8f        Fail
ffffff97        ffffff8f        Fail
ffffff3b        ffffff97        Fail
ffffff3b        00000071        Fail
00000069        00000071        Fail
000000c5        00000069        Fail
ffffff8f        000000c5        Fail
ffffff97        ffffff8f        Fail
ffffff3b        ffffff97        Fail
00000071        ffffff3b        Fail
```