CPE 310

Assembler Project

Final Testing & Robustness Improvements

4/9/2025

Emily Allegeyer, Nicholas Bennet, Ryan Busch, Alexander White

Introduction

While preparing for its Titan-9 launch, our company ByteForge's testing division came across an important issue. There were several bit flips present in the firmware that impacted functionality. We were tasked with identifying and rectifying these bit flips to ensure the Titan-9 can laungh successfully.

Experiment

In order to fix the provided machine code, we first had to make it more easily human-readable. This meant parsing the original machine code (shown in *Figure 1*) and noting what each instruction is meant to do via the opcode (first 6 bits) and in some cases the funct value (last 6 bits). From there we split the individual instructions into their smaller parts, such as rs, rt, and immediate. The results of doing this for each instruction are shown in *Figure 2*.

After this, figuring out which bits were flipped was a relatively simple task. For instance, it was clear on the 11th line that the funct field was incorrect because it implied an operation we had no support for. It was observations like this we had to make in order to fix the machine code fully and create the intended product.

```
00000001001010100101000000100000 ->     Rd field
00100001000010110000000000000101
00000001001010100110000000100100
00110001110011010000000000001111 ->     Rs field
00010001101000000000000000110100
00010101011011010000000000110100
00000001110010000000000000011010
00000000000000010011110000010000 ->     Rt field
00000000000000000110000000010010
00000001001010100000000000011000
00000001011011001100100000100001 ->     Funct field
00110111001100000000000011111111
00000001000010011000100000101010
00101001000100100000000000001010
00000001001010101001100000100010
00111100000101110100000000000000
10001110111101000000000000000000
10001110111101010000000000000100 ->     opcode field
00111100000101100001001000110100
```

Figure 1: Original machine code

```
ADD  000000|01001|01010|01010|00000|100000  ->  Rd field
ADDI 001000|01000|01011|0000000000000101
AND  000000|01001|01010|01100|00000|100100
ANDI 001100|01110|01101|0000000000001111    ->  Rs field
BEQ  000100|01101|00000|0000000000110100
BNE  000101|01011|01101|0000000000110100
DIV  000000|01110|01000|00000|00000|011010
MFHI 000000|00000|00010|01111|00000|010000  ->  Rt field
MFLO 000000|00000|00000|11000|00000|010010
MULT 000000|01001|01010|00000|00000|011000
ADDU 000000|01011|01100|11001|00000|100001  ->  Funct field
ORI  001101|11001|10000|0000000011111111
SLT 000000|01000|01001|10001|00000|101010
SLTI 001010|01000|10010|0000000000001010
SUB  000000|01001|01010|10011|00000|100010
LUI  001111|00000|10111|0100000000000000
LW   100011|10111|10100|0000000000000000
LW   100011|10111|10101|0000000000000100    ->  opcode field
LUI  001111|00000|10110|0001001000110100
```

Figure 2: Parsed, human-readable machine code

```
ADD  000000|01001|01010|01000|00000|100000    ADD  $t0, $t1, $t2
ADDI 001000|01000|01011|0000000000000101      ADDI $t3, $t0, #0x5
AND  000000|01001|01010|01100|00000|100100    AND  $t4, $t1, $t2
ANDI 001100|01100|01101|0000000000001111      ANDI $t5, $t4, #0xF
BEQ  000100|01101|00000|0000000000110100      BEQ  $t5, $zero, #0x34
BNE  000101|01011|01101|0000000000110100      BNE  $t5, $t3, #0x34
DIV  000000|01110|01000|00000|00000|011010    DIV  $t0, $t6
MFHI 000000|00000|00000|01111|00000|010000    MFHI $t7
MFLO 000000|00000|00000|11000|00000|010010    MFLO $t8
MULT 000000|01001|01010|00000|00000|011000    MULT $t1, $t2
OR   000000|01011|01100|11001|00000|100101    OR   $t9, $t3, $t4
ORI  001101|11001|10000|0000000011111111      ORI  $s0, $t9, #0xFF
SLT  000000|01000|01001|10001|00000|101010    SLT  $s1, $t0, $t1
SLTI 001010|01000|10010|0000000000001010      SLTI $s2, $t0, #0xA
SUB  000000|01001|01010|10011|00000|100010    SUB  $s3, $t1, $t2
LUI  001111|00000|10111|0100000000000000      LUI  $s7, #0x4000
LW   100011|10111|10100|0000000000000000      LW   $s4, #0x0($s7)
SW   101011|10111|10101|0000000000000100      SW   $s5, #0x4($s7)
LUI  001111|00000|10110|0001001000110100      LUI  $s6, #0x1234
```

Figures 3 (left), 4 (right): Fixed machine code (left) and assembly instructions (right)

Results

After parsing each line and looking at the denoted fields where the bit flips happened, we believe we successfully produced the machine code that was intended by ByteForge (*Figure 3*). In the first line, the rd field was the one with the bit flip. The flip resulted in a change from storing the result in register $t0 to storing it in $t2. This was obvious both because of the pattern of registers used, and the fact that it $t0 is a source register used in the next instruction. The next bit flip to occur was in the rs field (source register) of the 4th instruction. This would create an incorrect result if left unfixed, so it needed to be changed. The process for figuring out this bit flip was similar to the previous one but reverse. You could tell which register was meant to be the source because of the destination register of the previous instruction. The 01110 had to be changed to a 01100.

The next bit flip was also easy to identify. Although it likely would not have caused any errors due to being in an unused field, it was still best to change. This bit flip was in the rt field of the 8th instruction, an MFHI instruction. As previously stated, the MFHI instruction does not use the rt field so it was a simple change from 01000 to 00000. After that was probably the hardest to fix. It was easy to identify because it was a bit flip in the funct field that implied an operation there was no support implemented for, an error that could crash any Titan-9 systems if unfixed. Figuring out what to change it to, however, was not as easy. There were two possible things the original funct field could have been, 100100 (AND instruction) or 100101 (OR instruction). Either of them would result in a valid assembly instruction, but based on the pattern of the rest of the instructions it made the most sense to change it to 100101.

The 5th and final bit flip was on line 18 and it was the opcode. The corruption turned it into a lw instruction with opcode 100011, which if unfixed would mess with register values in unintended ways. This was also a very easy instruction to fix because there was only one option that made sense: 101011, the sw operation. This made sense because it was the one we had functionality implemented for, and it was one that used an immediate field. After changing this instruction, we believe we fully fixed the corrupted machine code. The results can be seen in *Figure 3*, and were ably to be successfully run through our interpreter to get the results in *Figure 4*. This shows that

at the very least, the instructions are all syntactically correct and when reading them in their assembly format look like they create a valid instruction set.