CPE 310 Project Report

The Ohm Squad

Nathan Kirby, Tucker Wilson, Austin Driggs

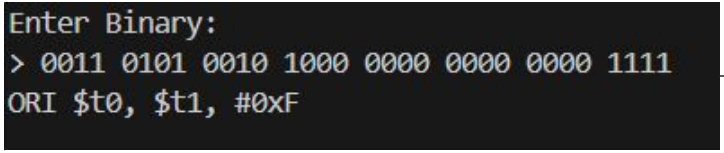
Debugging Process for MIPS Translatron 3000

- The generic procedure for the code was to first scan through the functions of the code for any immediate flags that could result in the code not working, this included checking:
    - Bits of the opcode
    - Parameter values and Restrictions
    - Missing lines of instructions
- For the instructions that needed completely entered into the directory, which were slti, ori, and mult, were entered from scratch into the files. Gold standards of completed code were used as reference

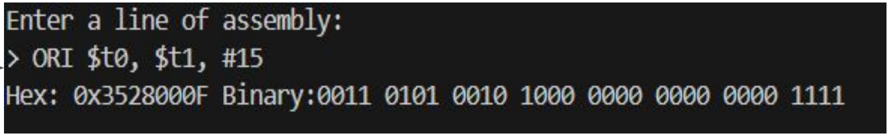
Nate’s slides are next

Debugging Process for MIPS Translatron 3000

- Next, we used VSCode to type an instruction line for the specific instruction we are working on to get the assembly code from the MIPS line.
- This assembly code was copied and pasted in VScode to go back from machine code to MIPS to verify we got the same answer we submitted.

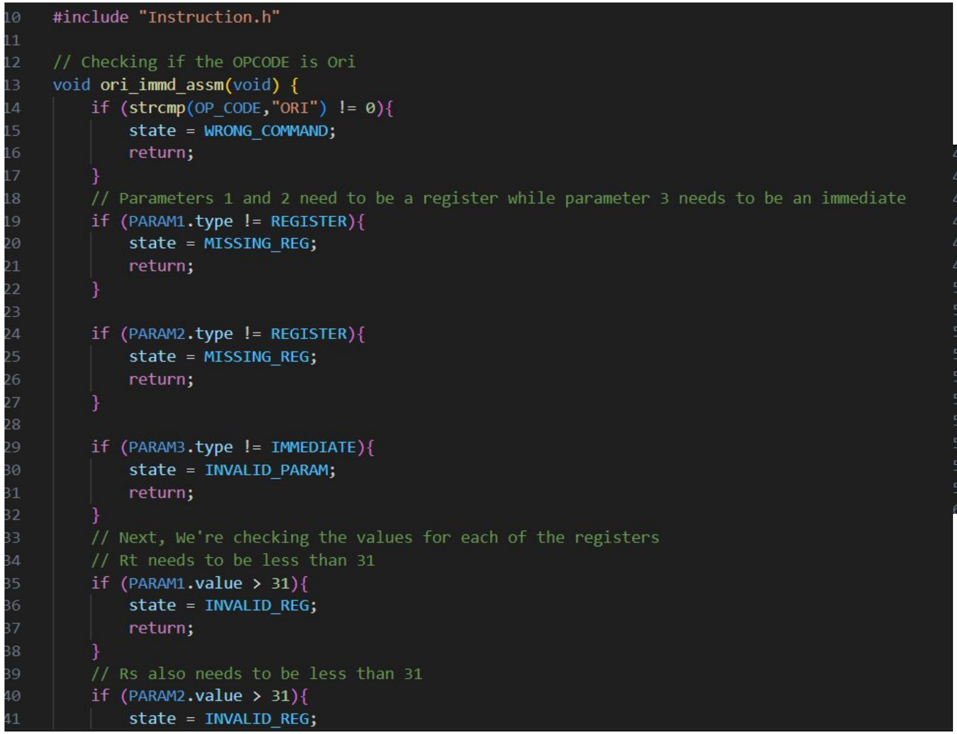


```
Enter Binary:
> 0011 0101 0010 1000 0000 0000 0000 1111
ORI $t0, $t1, #0xF
```

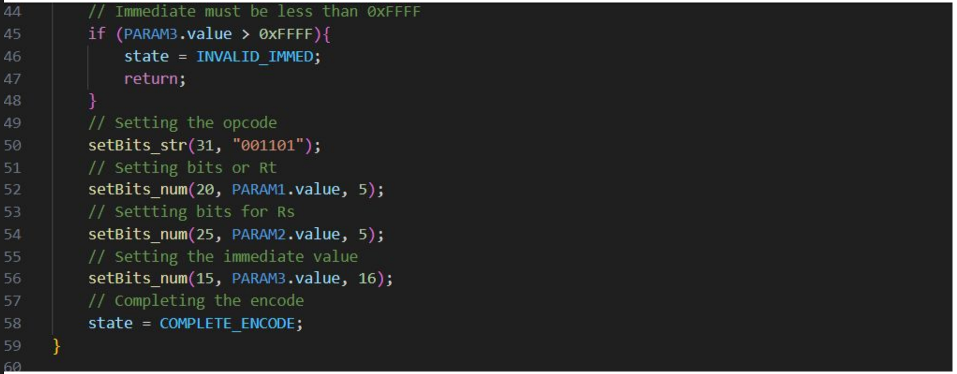


```
Enter a line of assembly:
> ORI $t0, $t1, #15
Hex: 0x3528000F Binary:0011 0101 0010 1000 0000 0000 0000 1111
```

Complete Instruction Script Implementation: ORI

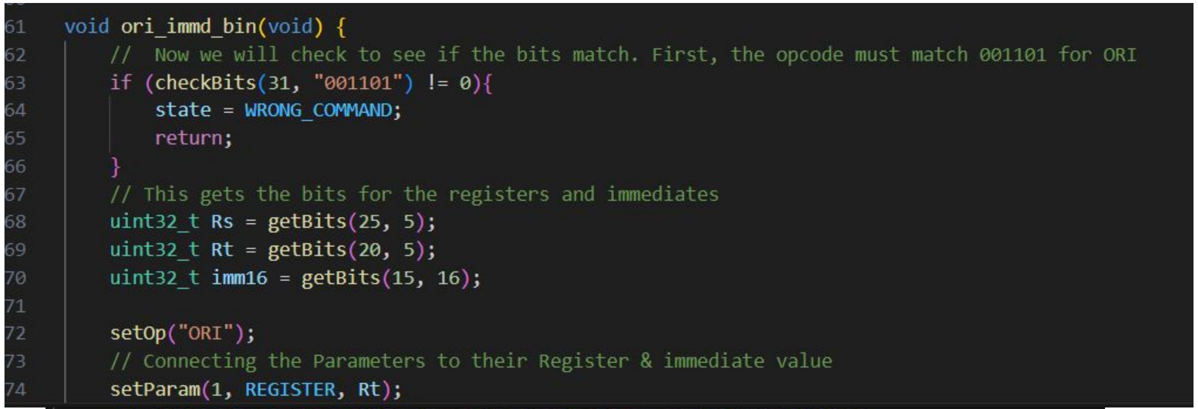


```c
#include "Instruction.h"

// Checking if the OPCODE is Ori
void ori_immd_assm(void) {
    if (strcmp(OP_CODE,"ORI") != 0){
        state = WRONG_COMMAND;
        return;
    }
    // Parameters 1 and 2 need to be a register while parameter 3 needs to be an immediate
    if (PARAM1.type != REGISTER){
        state = MISSING_REG;
        return;
    }

    if (PARAM2.type != REGISTER){
        state = MISSING_REG;
        return;
    }

    if (PARAM3.type != IMMEDIATE){
        state = INVALID_PARAM;
        return;
    }
    // Next, We're checking the values for each of the registers
    // Rt needs to be less than 31
    if (PARAM1.value > 31){
        state = INVALID_REG;
        return;
    }
    // Rs also needs to be less than 31
    if (PARAM2.value > 31){
        state = INVALID_REG;
```

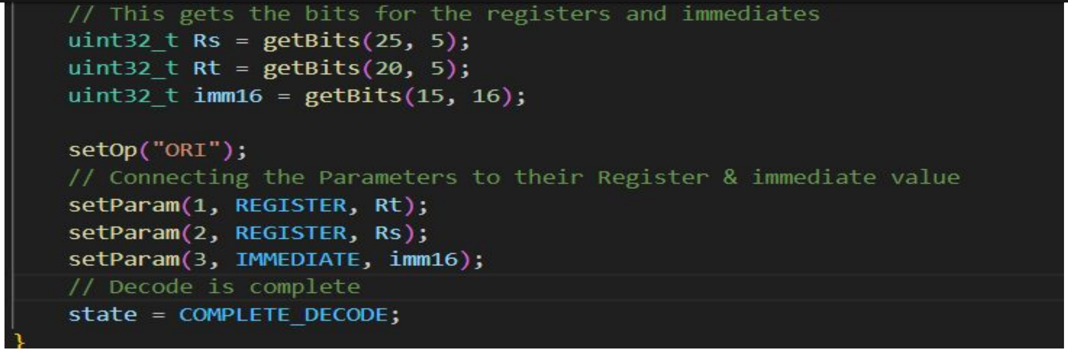


```c
    // Immediate must be less than 0xFFFF
    if (PARAM3.value > 0xFFFF){
        state = INVALID_IMMED;
        return;
    }
    // Setting the opcode
    setBits_str(31, "001101");
    // Setting bits or Rt
    setBits_num(20, PARAM1.value, 5);
    // Settting bits for Rs
    setBits_num(25, PARAM2.value, 5);
    // Setting the immediate value
    setBits_num(15, PARAM3.value, 16);
    // Completing the encode
    state = COMPLETE_ENCODE;
}
```

Complete Instruction Script Implementation: ORI



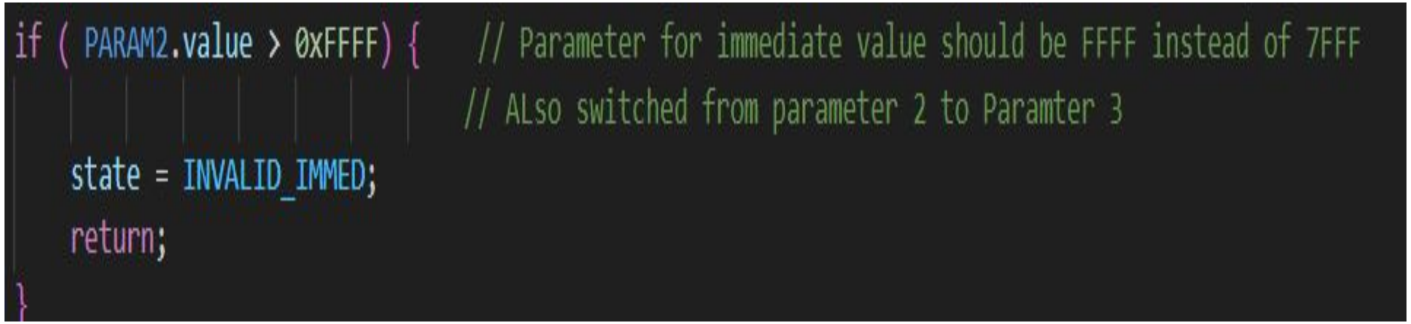
```c
void ori_immd_bin(void) {
    //  Now we will check to see if the bits match. First, the opcode must match 001101 for ORI
    if (checkBits(31, "001101") != 0){
        state = WRONG_COMMAND;
        return;
    }
    // This gets the bits for the registers and immediates
    uint32_t Rs = getBits(25, 5);
    uint32_t Rt = getBits(20, 5);
    uint32_t imm16 = getBits(15, 16);

    setOp("ORI");
    // Connecting the Parameters to their Register & immediate value
    setParam(1, REGISTER, Rt);
```



```c
    // This gets the bits for the registers and immediates
    uint32_t Rs = getBits(25, 5);
    uint32_t Rt = getBits(20, 5);
    uint32_t imm16 = getBits(15, 16);

    setOp("ORI");
    // Connecting the Parameters to their Register & immediate value
    setParam(1, REGISTER, Rt);
    setParam(2, REGISTER, Rs);
    setParam(3, IMMEDIATE, imm16);
    // Decode is complete
    state = COMPLETE_DECODE;
}
```

Additional Simple Corrections through Debugging

LW



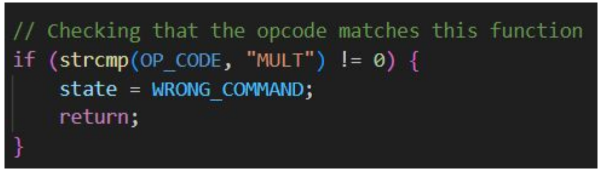
```
if ( PARAM2.value > 0xFFFF) {    // Parameter for immediate value should be FFFF instead of 7FFF
                                // ALso switched from parameter 2 to Paramter 3
    state = INVALID_IMMED;
    return;
}
```

Austins slides are next

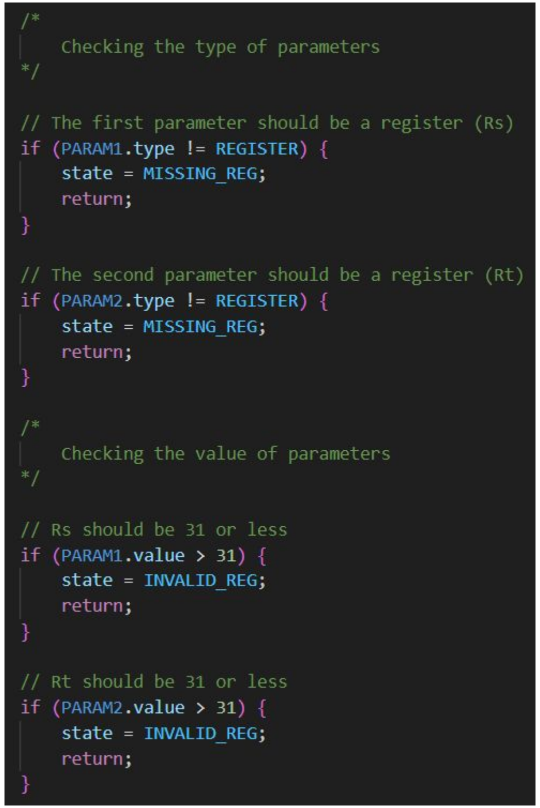
MULT Implementation - Assembly to Binary

General Procedure:

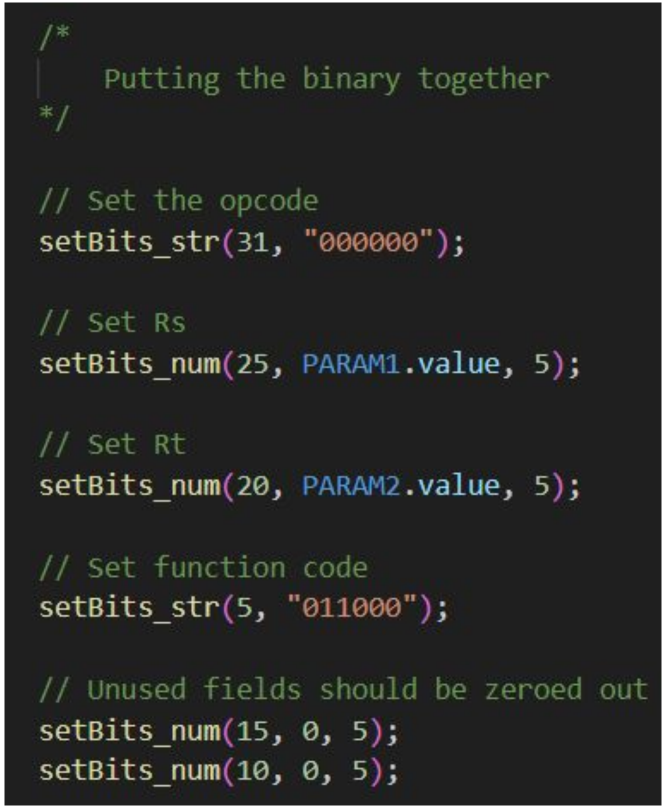
- Check opcode
- Check parameter types
- Check parameter values
- Set opcode
- Set parameter values
- Set function code
- Set unused to 0’s



```c
// Checking that the opcode matches this function
if (strcmp(OP_CODE, "MULT") != 0) {
    state = WRONG_COMMAND;
    return;
}
```

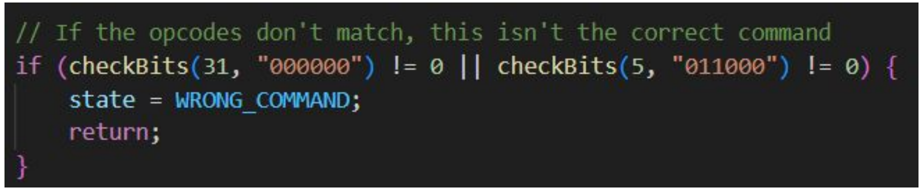


```c
/*
    Checking the type of parameters
*/

// The first parameter should be a register (Rs)
if (PARAM1.type != REGISTER) {
    state = MISSING_REG;
    return;
}

// The second parameter should be a register (Rt)
if (PARAM2.type != REGISTER) {
    state = MISSING_REG;
    return;
}

/*
    Checking the value of parameters
*/

// Rs should be 31 or less
if (PARAM1.value > 31) {
    state = INVALID_REG;
    return;
}

// Rt should be 31 or less
if (PARAM2.value > 31) {
    state = INVALID_REG;
    return;
}
```

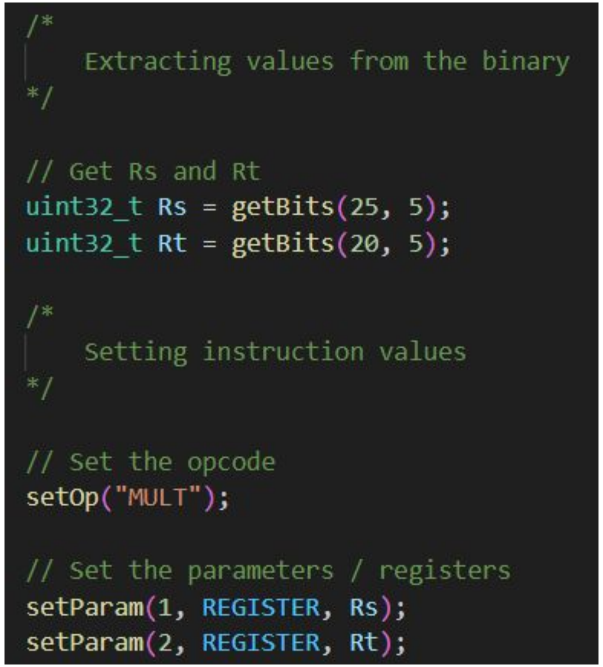


```c
/*
    Putting the binary together
*/

// Set the opcode
setBits_str(31, "000000");

// Set Rs
setBits_num(25, PARAM1.value, 5);

// Set Rt
setBits_num(20, PARAM2.value, 5);

// Set function code
setBits_str(5, "011000");

// Unused fields should be zeroed out
setBits_num(15, 0, 5);
setBits_num(10, 0, 5);
```

MULT Implementation - Binary to Assembly



```
// If the opcodes don't match, this isn't the correct command
if (checkBits(31, "000000") != 0 || checkBits(5, "011000") != 0) {
    state = WRONG_COMMAND;
    return;
}
```



```
/*
    Extracting values from the binary
*/

// Get Rs and Rt
uint32_t Rs = getBits(25, 5);
uint32_t Rt = getBits(20, 5);

/*
    Setting instruction values
*/

// Set the opcode
setOp("MULT");

// Set the parameters / registers
setParam(1, REGISTER, Rs);
setParam(2, REGISTER, Rt);
```

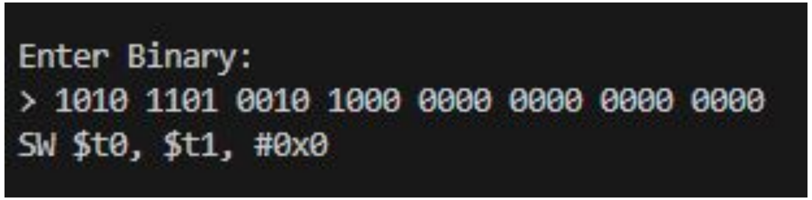
General Procedure:

- Check op and function codes
- Set parameter values
- Set opcode instruction
- Set parameter registers

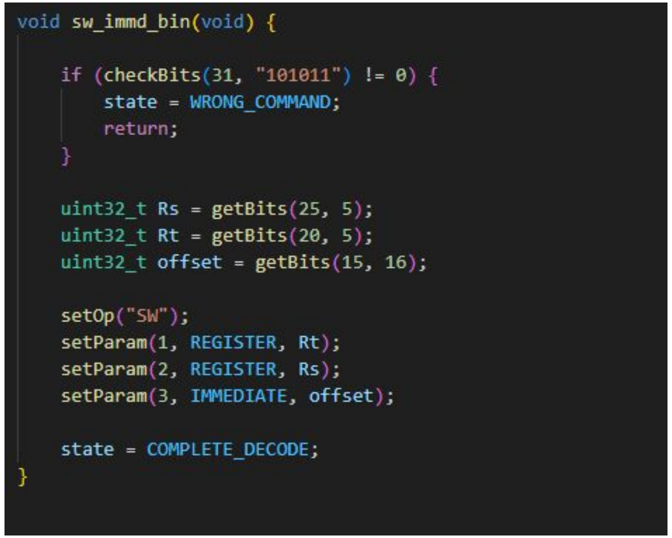
Tucker’s slides are next

SW

- The format for the binary to Assembly output format was wrong
- The incorrect code is shown on this slide



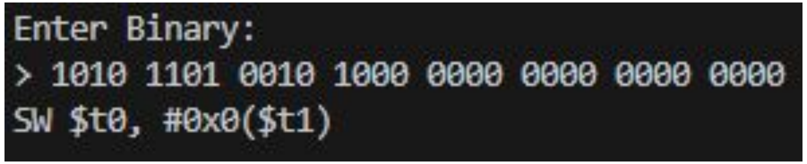
```
Enter Binary:
> 1010 1101 0010 1000 0000 0000 0000 0000
SW $t0, $t1, #0x0
```



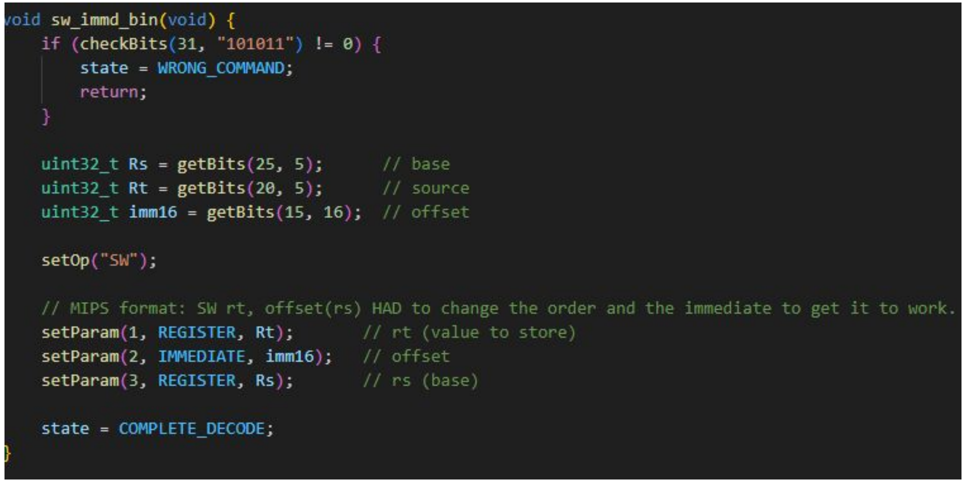
```c
void sw_immd_bin(void) {

    if (checkBits(31, "101011") != 0) {
        state = WRONG_COMMAND;
        return;
    }

    uint32_t Rs = getBits(25, 5);
    uint32_t Rt = getBits(20, 5);
    uint32_t offset = getBits(15, 16);

    setOp("SW");
    setParam(1, REGISTER, Rt);
    setParam(2, REGISTER, Rs);
    setParam(3, IMMEDIATE, offset);

    state = COMPLETE_DECODE;
}
```

SW corrected

- To fix this The lines of code in figure shown was changed
- This reordered the offset and Rs values to format the binary to assembly right.
- Also made Immediate = imm16



```
Enter Binary:
> 1010 1101 0010 1000 0000 0000 0000 0000
SW $t0, #0x0($t1)
```



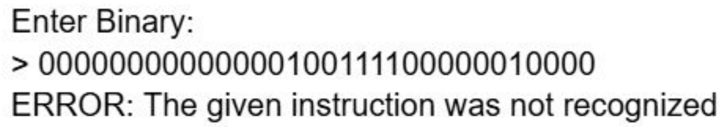
```
void sw_immd_bin(void) {
    if (checkBits(31, "101011") != 0) {
        state = WRONG_COMMAND;
        return;
    }

    uint32_t Rs = getBits(25, 5);      // base
    uint32_t Rt = getBits(20, 5);      // source
    uint32_t imm16 = getBits(15, 16);  // offset

    setOp("SW");

    // MIPS format: SW rt, offset(rs) HAD to change the order and the immediate to get it to work.
    setParam(1, REGISTER, Rt);       // rt (value to store)
    setParam(2, IMMEDIATE, imm16);   // offset
    setParam(3, REGISTER, Rs);       // rs (base)

    state = COMPLETE_DECODE;
}
```

Corrupted slides are next

I (austin) can talk about these

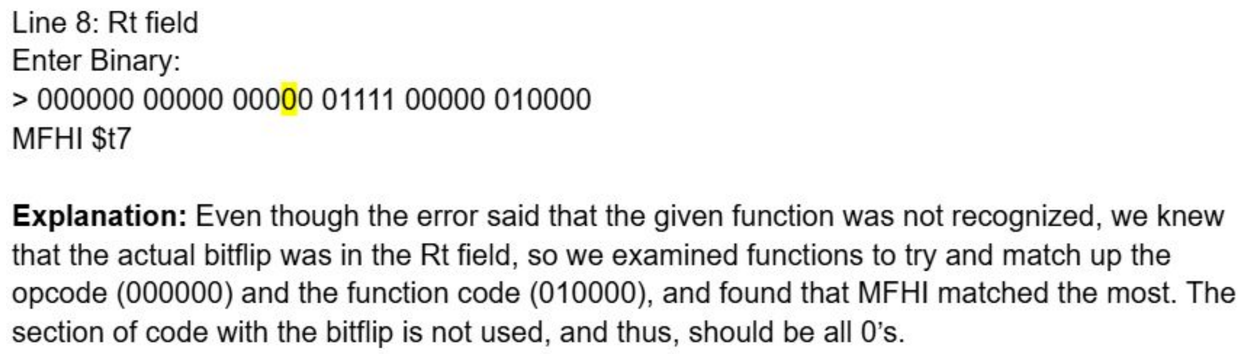
Corrupted Machine Code - Debugging

- First we started off by entering each line of the corrupted machine code into our MIPS Translatron 3000 machine. This resulted in the following outputs and errors:



```
Enter Binary:
> 00000000000000100111100000010000
ERROR: The given instruction was not recognized
```

- Then we examined each section of the 32 bit input to try and match it up with a function from our MIPSzy options, and entered the correct code back into our machine:



Line 8: Rt field

```
Enter Binary:
> 000000 00000 00000 01111 00000 010000
MFHI $t7
```

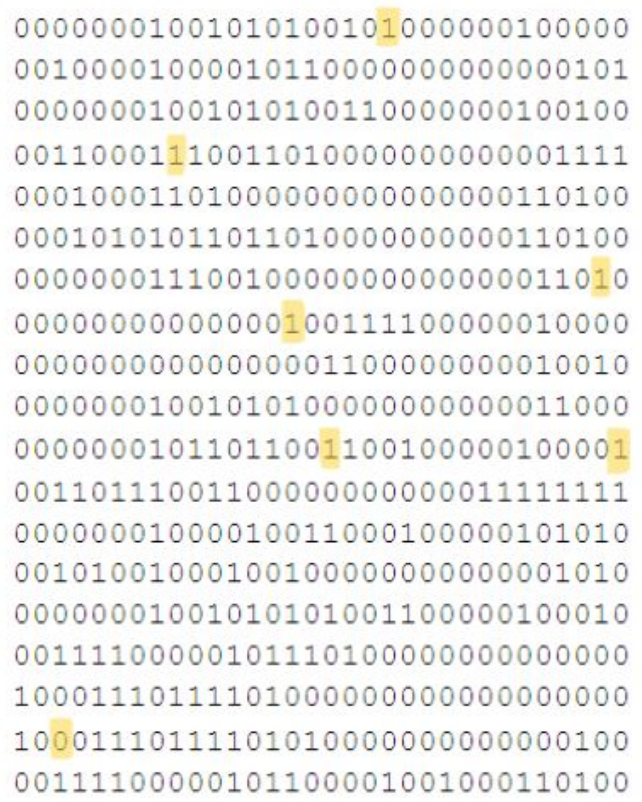
Explanation: Even though the error said that the given function was not recognized, we knew that the actual bitflip was in the Rt field, so we examined functions to try and match up the opcode (000000) and the function code (010000), and found that MFHI matched the most. The section of code with the bitflip is not used, and thus, should be all 0's.

Corrupted Machine Code - Corrections

- We followed the same process for the rest of the corrections, the above slide was just one example of a correction that we made to the corrupted machine code given to us.
- This process also proved that our MIPS Translatron 3000 machine worked as expected, giving us the correct errors when applicable, and also identifying functions to determine if they are correct or not.

Corrupted Machine Code - Summary

A summary of the results is shown to the right, with all of the incorrect bit flips highlighted:



```
00000001001010100101000000100000
00100001000010110000000000000101
00000001001010100110000000100100
00110001110011010000000000001111
00010001101000000000000000110100
00010101011011010000000000110100
00000001110010000000000000011010
00000000000000100111100000010000
00000000000000001100000000010010
00000001001010100000000000011000
00000001011011001100100000100001
00110111001100000000000011111111
00000001000010011000100000101010
00101001000100100000000000001010
00000001001010101001100000100010
00111100000101110100000000000000
10001110111101000000000000000000
10001110111101010000000000000100
00111100000101100001001000110100
```

Conclusion

In conclusion, we learned a lot about debugging, proper documentation, and fixing corrupted machine code using the MIPSzy language set.