Abstract

Machines cannot understand human language, while machine language, which is all binary is unintelligible to humans. In order for things to work, an intermediate converter is needed which can take code in form of strings and deliver its machine language representation for the processor to work on. This is the need of an assembler.

This assembly code is written in an editor, which is in human understandable is converted into binary form. The code is firstly compiled, to make sure there aren't any errors in the code. It is then converted in binary by an assembler and these codes after getting lined go to the execution unit where the commands get performed and the result is ultimately obtained.

Traditionally, compiler and assembler are two separate units which are to be constructed. But when code is written in a middle level language, then in a single step, errors in code can be verified and if no flaws are found, then the assembler can do its work.

In this report, an assembler written in Verilog is proposed for a Reduced Instruction Set Computer (RISC) based MiniMIPS processor supporting 10 instructions. Assembly instructions are passed to this assembler by giving the file as input. Lexical analysis is carried out and tokenization is done to divide the instructions using whitespace and then by using case statements, each of these tokens is converted into machine language if there are no errors found. This is run on ModelSim platform. The assembler then generates an output file containing the binary equivalent of the assembly code and the program counter value for each line. The result is also displayed in the transcript window. Based on the results, the assembler has been developed successfully.

INDEX

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

R	Register
I	Immediate
L/S	Load/Store

Abbreviations

RISC	Reduced Instruction Set Computer
MIPS	Microprocessor without Interlocked Pipelined Stages

Chapter 1
Introduction

1.1 Prologue

Assembly language instructions are converted into binary by an assembler. There are several such modules that are assembled and those are liked together by the help of linker, the executable program is then loaded into the processor's memory and when time comes, it gets run and the steps listed in the program are carried out. **[1].**

Most of the assembly language instructions have one to one correspondence with machine instructions. So, each line is read one by one and according to the tokens, the binary code is generated.

1.2 Importance of the Topic

As it is difficult for humans to interpret machine language, a conversion tool becomes an absolute necessity in order to bridge the gap. This machine code is interpreted by the processor at a later stage and the execution of code can take place.

1.3 Objective of the Report

The objective of this report is to create a 32-bit MiniMIPS RISC processor using Verilog as the programming language. Any errors must be spotted, and appropriate message must be displayed to the user. Ultimately, the input assembly code is to be converted into binary form, and written into a file so that it can be executed by latter units.

1.4 Scope of the Report

An assembler has been made for 10 MiniMIPS instructions namely add, sub, addi, xor, and, ori, sw, lw, slti and sll. These are discussed in detail in the methodology section of the report. These 10 instructions belong to Register and Immediate type and this covers all of the variety of instructions except the jump type instructions.

1.5 Organization of the Rest of the Report

In chapter 2, discussion of the methodology is done wherein firstly the types of registers available have been discussed about, and the ones used for this particular project have been listed.

Then comes the explanation of the R and I type of instructions and then the opcode/opcode extension and such things which vary from instruction to instruction have been shown along with a short description of the instruction. After this, the procedure is discussed and a flowchart is also attached to give a better understanding of the workflow.

In chapter 3, the simulation results have been discussed. There are two scripts that are executed, one of them having some error, and the respective outputs are shown. A short discussion on the results is also done, talking about the outcome.

Chapter 2
Methodology

2.1 The MiniMIPS Registers & Instruction Formats

Considering the L/S architecture of MiniMIPS, registers would play a crucial role in the execution of any instruction as the processing of data would only take place after loading it into the register/s. The number of registers in a MiniMIPS processor is equal to its word-length. Hence, the 32-bit MiniMIPS processor considered for the purpose of this project contains 32 general purpose registers, each 32-bit wide.

The symbolic names of the registers corresponding to their numeric names is as shown below:

Numeric Name	Symbolic Name		Numeric Name	Symbolic Name	
$0	$zero	**Stores the value 0**	**$15**	$t7	
$1	$at	**Reserved**	**$16**	$s0	
$2	$v0	**Procedure Results**	**$17**	$s1	
$3	$v1		**$18**	$s2	
$4	$a0		**$19**	$s3	
$5	$a1	**Procedure Arguments**	**$20**	$s4	
$6	$a2		**$21**	$s5	**Operands**
$7	$a3		**$22**	$s6	
$8	$t0		**$23**	$s7	
$9	$t1	**Temporary Values**	**$24**	$t8	
$10	$t2		**$25**	$t9	**Temporary Values**
$11	$t3		**$26**	$k0	**Reserved**
$12	$t4		**$27**	$k1	
$13	$t5		**$28**	$gp	**Global Pointer**
$14	$t6		**$29**	$sp	**Stack Pointer**
			$30	$fp	**Frame Pointer**
			$31	$ra	**Return Address**

TABLE 2.1 Registers and their corresponding symbol names

The instructions supported by the assembler to be designed would only use the registers $t0-$t9 and $s0-$s7 for the required operations.

The MiniMIPS contains three instruction formats namely, R, I and of J type. These formats divide the 32-bit instructions in separate classes, each of which can be distinguished through their corresponding opcodes. Hence, uniformity is maintained throughout the instruction formats.

The instructions to be supported by the designed assembler contain R and I type instructions. Thus, discussing the formats;

R-Type: R-type instructions perform operations on two source registers (rt) and (rs) and store the consequent result in a destination register (rd). The instructions in this format contain an opcode extension (fn) along with their corresponding opcodes (op) which allows for more operations to be defined in a specific class of instructions. The (sh) field comes into the picture for instructions with a constant shift amount.

op	rs	rt	rd	sh	fn
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type: From the instructions supported, the I-type instructions can be divided into two classes. First being the instructions where an arithmetic/logical operation is performed between the source register (rs) and the immediate value, and the result stored in (rt). Second are the load/store instructions wherein the immediate operand, used as an offset is added to the base register (rs) to form a relative address which is then used to either load the data from the memory to register (rt) or store the contents of (rt) to memory.

op	rs	rt	operand/offset
6 bits	5 bits	5 bits	16 bits

The formats for all the instructions along with their opcodes would be discussed in the next section of this chapter.

2.2 Instructions

Now explaining the formats of the instructions supported along with their opcodes;

add: add $t0, $s0, $s1 (Addition of the contents in two source registers)

op	rs	rt	rd	sh	fn
000000	10000	10001	01000	00000	100000

sub: sub $t0, $s0, $s1 (Subtraction of the contents in two source registers)

op	rs	rt	rd	sh	fn
000000	10000	10001	01000	00000	100010

xor: xor $s6, $s6, $t8 (xor-ing of the contents in two source registers)

op	rs	rt	rd	sh	fn
000000	10110	11000	10110	00000	100110

sll: sll $s6, $s6, 4 (left logical shift of the contents of register $s6 stored in $s6)

op	rs	rt	rd	sh	fn
000000	00000	10110	10110	00100	000000

and: $s6, $s6, $t8 (And operation of the contents in the two source registers)

op	rs	rt	rd	sh	fn
000000	10110	11000	10110	00000	100100

addi: addi $t0, $t9, 10 (Addition operation between the immediate value and the source register.)

op	rs	rt	operand/offset
001000	01000	11001	0000000000001010

ori: ori $s7, $s5, 30 (OR operation between the immediate value and the source register.)

op	rs	rt	operand/offset
001101	10111	10101	0000000000011110

slti: slti $t1, $s7, 10 (Sets register (rt) to 1 if the contents in the source register is lesser than the immediate offset.)

op	rs	rt	operand/offset
001010	01001	10111	0000000000001010

lw: lw $t0, 40 ($s3) (loads the value at [$s3+40] into $t0)

op	rs	rt	operand/offset
100011	10011	01000	0000000000101000

sw: sw $t1, 44 ($s3) (stores the value at [$s3+44] into $t1)

op	rs	rt	operand/offset
101011	10011	01001	0000000000101100

2.3 Procedure

In order to design an assembler that supports these instructions, it would involve writing a program where on scanning a single or a multiline assembly level script containing these instructions, the following operations to be performed;

- Based on the scanned command, depending on whether if it's an R or I-type instruction, assign the corresponding opcode.

For R-Type Instructions,

- On scanning the registers used in the instruction, assign the corresponding binary values to the (rs), (rt) and (rd) fields. The binary values could be derived from converting the numeric name for each of the registers as shown in Table 1 in binary form.
- Assign the opcode extension corresponding to the instruction.
- The shift amount (sh) would be assigned based on the user entered constant if it's a shift instruction. In other cases, it would be assigned as zero.

For I-Type Instructions,

- On scanning the source/base register (rs) and the destination/data register (rt), assign the corresponding binary values to the register fields.
- Scan the 16-bit immediate value/offset which is to be entered in decimal format and assign the equivalent binary value.

Along with this, an instruction location counter would also have to be maintained which would determine the relative position of the instruction in the written assembly level script, assuming the program is loaded at the address 0 in the memory. This counter would hence, increment by four after each instruction.

The flow of this entire procedure of translating the assembly level program can be shown as follows:

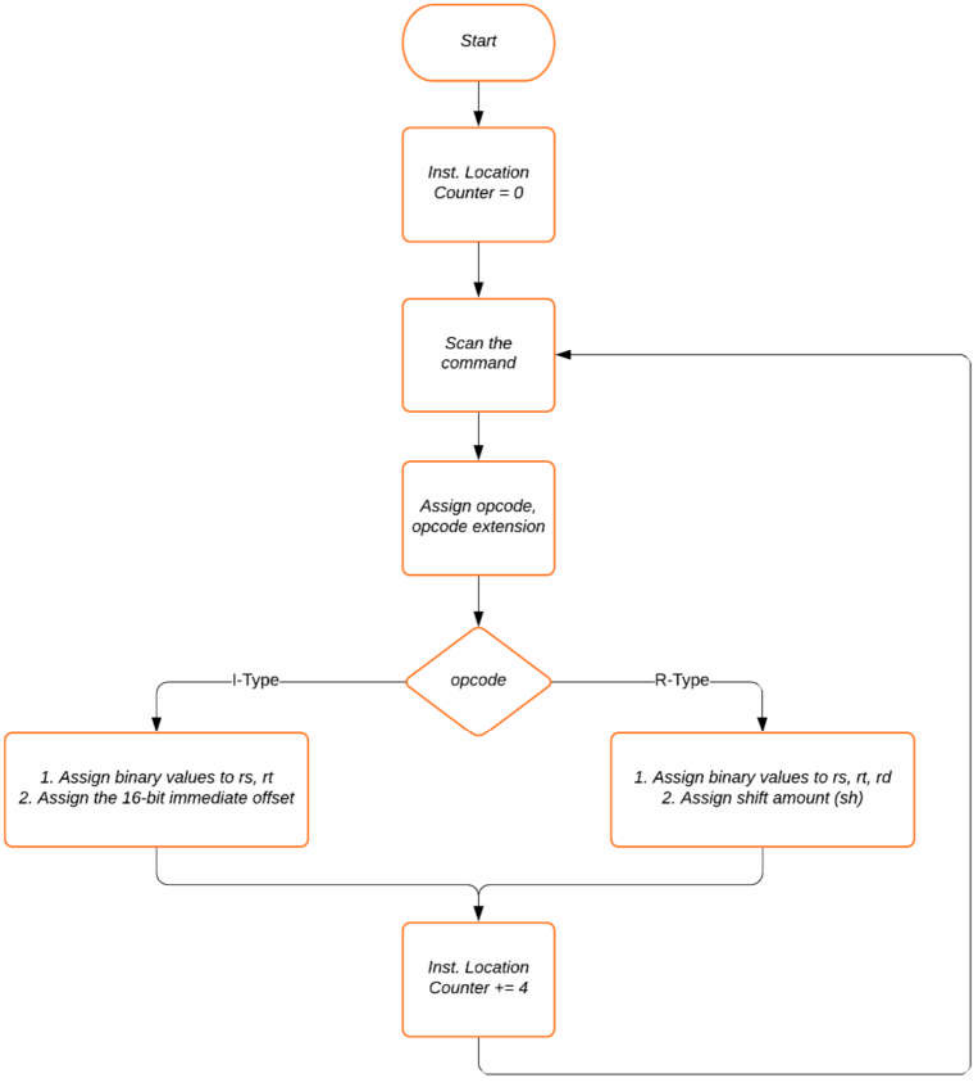

Figure 2.1: Demonstrating the Flow of the Assembler Design

Chapter 3

Implementation

3.1 Simulation Results

The implementation portion was carried out by writing a program in Verilog HDL using the ModelSim-Altera software. Following the procedure described in Section 2.3, the program was written which would scan an input assembly level file line by line and would write the corresponding machine code into an output text file, as well as display it on the ModelSim transcript. The written program uses conditional if-else and case statements in order to assign the machine code to R and I-type formats, as well as for assigning the registers.

In order to test the written program, two assembly level scripts were written which would check the desired output for all the 10 supported instructions, as well as check for how the program would respond to erroneous input. The two test scripts are as shown below;

Script 1:

```
lw $s6, 40 ($t3)
sub $s2, $t4, $s4
add $t5, $s2, $s1
sub $s6, $s6, $t5
sll $s6, $s6, 4
and $s6, $s6, $t2
xor $s6, $s6, $t8
sw $s6, 40 ($t3)
abc $s1, $s2, 4
```

Script 2:

```
lw $t0, 40 ($s3)
sll $t0, $t0, 2
addi $t0, $t9, 10
addi $s0, $s1, 56
addi $s5, $s0, -1
ori $s7, $s5, 30
slti $t1, $s7, 10
sw $t0, 40 ($s3)
sw $t1, 44 ($s3)
sw $xyz, 48 ($t8)
```

The observed output on feeding in these scripts to the program were as follows;

Output for Script 1:

```
00          100011 01011 10110 0000000000101000
04          000000 01100 10100 10010 00000 100010
08          000000 10010 10001 01101 00000 100000
0c          000000 10110 01101 10110 00000 100010
10          000000 00000 10110 10110 00100 000000
14          000000 10110 01010 10110 00000 100100
18          000000 10110 11000 10110 00000 100110
1c          101011 01011 10110 0000000000101000
```

```
# Address         Instruction
#
# 00          100011 01011 10110 0000000000101000
# 04          000000 01100 10100 10010 00000 100010
# 08          000000 10010 10001 01101 00000 100000
# 0c          000000 10110 01101 10110 00000 100010
# 10          000000 00000 10110 10110 00100 000000
# 14          000000 10110 01010 10110 00000 100100
# 18          000000 10110 11000 10110 00000 100110
# 1c          101011 01011 10110 0000000000101000
# Invalid Command: abc, exit.
```

Figure 3.1: Output Observed in the Transcript Window for Script 1

Output for Script 2:

```
00          100011 10011 01000 0000000000101000
04          000000 00000 01000 01000 00010 000000
08          001000 01000 11001 0000000000001010
0c          001000 10000 10001 0000000000111000
10          001000 10101 10000 1111111111111111
14          001101 10111 10101 0000000000011110
18          001010 01001 10111 0000000000001010
1c          101011 10011 01000 0000000000101000
20          101011 10011 01001 0000000000101100
```

```
# Address         Instruction
#
# 00          100011 10011 01000 0000000000101000
# 04          000000 00000 01000 01000 00010 000000
# 08          001000 01000 11001 0000000000001010
# 0c          001000 10000 10001 0000000000111000
# 10          001000 10101 10000 1111111111111111
# 14          001101 10111 10101 0000000000011110
# 18          001010 01001 10111 0000000000001010
# 1c          101011 10011 01000 0000000000101000
# 20          101011 10011 01001 0000000000101100
# Invalid register: $xyz, exit.
```

Figure 3.2: Output Observed in the Transcript Window for Script 2

3.2 Discussion

As shown in Figure 3.1 and Figure 3.2 and in the respective outputs of the test scripts, the corresponding machine codes for the opcodes, opcode extension, shift amount and the registers (rt), (rs) and (rd) for R-type instructions have been correctly assigned and the results can be verified from Table 2.1 and Section 2.2. Likewise, for the I-type instructions

the corresponding opcodes, registers and immediate value/offset have been correctly assigned. The instruction location counter value is also observed to get incremented by four after the execution of each instruction.

Also, on entering an invalid register or command name, an error message is displayed and the execution of the program is terminated.

Chapter 4

Conclusion

An assembler is an important part of the computer architecture, which bridges the gap between human and machine language understandability. The successful execution of an assembler with 10 instructions of R and I type was carried out in Verilog by performing lexical analysis and tokenization of input strings and its output was written in a file. The assembler was able to identify typographical errors and display an error message.

Chapter 5

Future Scope

The program can be extended to make it workable for the entire MiniMIPS instruction set. The usage of System Verilog instead of Verilog would provide a greater ease of coding as it has inbuilt file handling and manipulation functions. This would result in increased ease of parsing and tokenization of skills, and would result into a more robust assembler.

References

[1] Behrooz Parhami (2005). "Assembly Language Programs," in *Computer Architecture: From Microprocessors to Supercomputers, 1st* Indian ed. New York, NY: Oxford University Press, pp 123-130

[2] Palnitkar, S. (2020). *Verilog HDL: A guide to digital design synthesis*. Chennai: Pearson.

Appendix

```
`define NULL 0 // Defining the NULL constant
module assembler();
  reg [31:0]loc = 5'h0; // Instruction Location Counter
  reg [5:0]opcode; // Opcode
  reg [5:0]opcode_ex; // Opcode Extension
  reg [4:0]rs; // Source Register
  reg [4:0]rt; // Source Register 2/ Destination Register
  reg [4:0]rd; // Destination Register
  reg [4:0]sh; // Shift Operand
  reg [15:0]imm_off; // Immediate Operand/Offset

  // Results of scanning(strings) would be stored here
  reg [8*7:0]op_scan;
  reg [50*7:0]temp_rs, temp_rt, temp_rd, temp_sh;

  integer file, op_file, r; // For reading & writing in files

  function [4:0]assign_reg;
    // Takes registers as strings and assigns the corresponding binary
values
    input [50*7:0]reg_str;
    begin
    case(reg_str)
      "$s0,":
        assign_reg = 5'b10000;
      "$s1,":
        assign_reg = 5'b10001;
                "$s2,":
        assign_reg = 5'b10010;
      "$s3,":
        assign_reg = 5'b10011;
      "$s4,":
        assign_reg = 5'b10100;
      "$s5,":
        assign_reg = 5'b10101;
      "$s6,":
        assign_reg = 5'b10110;
      "$s7,":
        assign_reg = 5'b10111;
      "$t0,":
        assign_reg = 5'b01000;
      "$t1,":
        assign_reg = 5'b01001;
      "$t2,":
        assign_reg = 5'b01010;
      "$t3,":
        assign_reg = 5'b01011;
                "$t4,":
```

```
        assign_reg = 5'b01100;
     "$t5,":
        assign_reg = 5'b01101;
      "$t6,":
        assign_reg = 5'b01110;
      "$t7,":
        assign_reg = 5'b01111;
      "$t8,":
        assign_reg = 5'b11000;
       "$t9,":
        assign_reg = 5'b11001;
      //Error Handling
      default:
        begin
          $display("Invalid register: %0s exit.", reg_str);
          $finish;
        end
    endcase
    end
  endfunction

  function [4:0]assign_rt;
    // Takes registers as strings and assigns the corresponding binary
values
    input [50*7:0]reg_str;

    begin
    case(reg_str)
      "$s0":
        assign_rt = 5'b10000;
      "$s1":
        assign_rt = 5'b10001;
                "$s2":
        assign_rt = 5'b10010;
      "$s3":
        assign_rt = 5'b10011;
       "$s4":
        assign_rt = 5'b10100;
      "$s5":
        assign_rt = 5'b10101;
      "$s6":
        assign_rt = 5'b10110;
      "$s7":
        assign_rt = 5'b10111;
      "$t0":
        assign_rt = 5'b01000;
      "$t1":
        assign_rt = 5'b01001;
      "$t2":
        assign_rt = 5'b01010;
      "$t3":
        assign_rt = 5'b01011;
```

```
      "$t4":
        assign_rt = 5'b01100;
    "$t5":
        assign_rt = 5'b01101;
    "$t6":
        assign_rt = 5'b01110;
    "$t7":
        assign_rt = 5'b01111;
    "$t8":
        assign_rt = 5'b11000;
    "$t9":
        assign_rt = 5'b11001;
      // Error Handling
      default:
        begin
          $display("Invalid register: %0s, exit.", reg_str);
          $finish;
        end
    endcase
    end
  endfunction

  function [4:0]assign_reg_ls;
    // Takes registers as strings and assigns the corresponding binary
values
    input [50*7:0]reg_str;

    begin
    case(reg_str)
      "($s0)":
        assign_reg_ls = 5'b10000;
      "($s1)":
        assign_reg_ls = 5'b10001;
      "($s2)":
        assign_reg_ls = 5'b10010;
      "($s3)":
        assign_reg_ls = 5'b10011;
      "($s4)":
        assign_reg_ls = 5'b10100;
      "($s5)":
        assign_reg_ls = 5'b10101;
      "($s6)":
        assign_reg_ls = 5'b10110;
      "($s7)":
        assign_reg_ls = 5'b10111;
      "($t0)":
        assign_reg_ls = 5'b01000;
      "($t1)":
        assign_reg_ls = 5'b01001;
      "($t2)":
        assign_reg_ls = 5'b01010;
      "($t3)":
```

```
        assign_reg_ls = 5'b01011;
      "($t4)":
        assign_reg_ls = 5'b01100;
      "($t5)":
        assign_reg_ls = 5'b01101;
      "($t6)":
        assign_reg_ls = 5'b01110;
      "($t7)":
        assign_reg_ls = 5'b01111;
      "($t8)":
        assign_reg_ls = 5'b11000;
      "($t9)":
        assign_reg_ls = 5'b11001;
      // Error Handling
      default:
        begin
          $display("Invalid register: %0s, exit.", reg_str);
          $finish;
        end
    endcase
    end
  endfunction

  initial
    // Scanning the file
    begin : file_tread
      file = $fopen("script1.txt", "r");
      op_file = $fopen("assembler_output1.txt", "w");

      if(file == `NULL)
        disable file_tread;

      $display("Address         Instruction\n");

      while (!$feof(file))
        begin
          r = $fscanf(file, "%s \n", op_scan); // First scanning only
the operation

          // Assigning the corresponding opcodes & opcode extensions
          case(op_scan)
            "add":
                                  begin
                opcode = 6'b000000;
                                    opcode_ex = 6'b100000;
                                  end
            "sub":
                                  begin
                opcode = 6'b000000;
                                    opcode_ex = 6'b100010;
                                  end
            "addi": opcode = 6'b001000;
```

```
                                "sll":
                                  begin
                opcode = 6'b000000;
                                    opcode_ex = 6'b000000;
                                  end
              "and":
                                  begin
                opcode = 6'b000000;
                                    opcode_ex = 6'b100100;
                                  end
              "ori": opcode = 6'b001101;
                                "sw": opcode = 6'b101011;
              "lw": opcode = 6'b100011;
              "slti": opcode = 6'b001010;
                                "xor":
                                  begin
                opcode = 6'b000000;
                                    opcode_ex = 6'b100110;
                                  end
              default:
                // Error Handling
                begin
                  $display("Invalid Command: %0s, exit.", op_scan);
                  disable file_tread;
                end
            endcase

            if(opcode == 6'b000000) // For ALU type operations
                                     begin
                                               if(opcode_ex == 6'b000000)
//for sll instruction
                                                     begin
                                                       r = $fscanf(file, "
%0s %0s %d \n", temp_rd, temp_rt, sh);
                                                       rd =
assign_reg(temp_rd);
                                                       rs = 5'b00000;
                                                       rt =
assign_reg(temp_rt);
                                                       $display("%2h
%b %b %b %b %b %b", loc, opcode, rs, rt, rd, sh, opcode_ex); //
Displaying the machine code
                                                       $fwrite(op_file,
"%2h         %b %b %b %b %b %b\n", loc, opcode, rs, rt, rd, sh,
opcode_ex);
                                                     end
                                               else
                                                     begin
                                                       r = $fscanf(file, "
%0s %0s %0s \n", temp_rd, temp_rs, temp_rt);
                                                       rd =
assign_reg(temp_rd);
```

```
                                                    rs =
assign_reg(temp_rs);
                                                    rt =
assign_rt(temp_rt);
                                                    sh = 5'b00000; //
Shift amount would be zero
                                                    $display("%2h
%b %b %b %b %b %b", loc, opcode, rs, rt, rd, sh, opcode_ex); //
Displaying the machine code
                                                    $fwrite(op_file,
"%2h         %b %b %b %b %b %b\n", loc, opcode, rs, rt, rd, sh,
opcode_ex);
                                                end
                              end
                          else if(opcode == 6'b101011 || opcode ==
6'b100011) // For sw/lw instruction
                               begin
                                                r = $fscanf(file, " %0s %d
%0s \n", temp_rt, imm_off, temp_rs);
              rt = assign_reg(temp_rt);
              rs = assign_reg_ls(temp_rs);
              $display("%2h         %b %b %b %b", loc, opcode, rs, rt,
imm_off); // Displaying the machine code
              $fwrite(op_file, "%2h         %b %b %b %b\n", loc,
opcode, rs, rt, imm_off);
                               end
          else
            // For I type operations
            begin
              r = $fscanf(file, " %0s %0s %d \n", temp_rd, temp_rs,
imm_off);
              rd = assign_reg(temp_rd);
              rs = assign_reg(temp_rs);
              $display("%2h         %b %b %b %b", loc, opcode, rd, rs,
imm_off); // Displaying the machine code
              $fwrite(op_file, "%2h         %b %b %b %b\n", loc,
opcode, rd, rs, imm_off);
            end
        loc = loc + 5'h4; // Increementing the location
        end
        $fclose(file);
    end
endmodule
```