FORMAN CHRISTIAN COLLEGE
(A CHARTERED UNIVERSITY)

COMP 451 COMPILER CONSTRUCTION B

SPRING 2025

Assignment 2

C Assembler

Muhammad Zeeshan 261940660

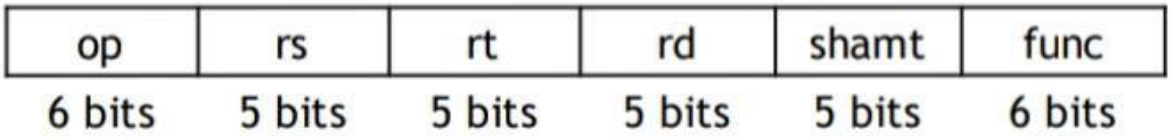
Muhammad Zeeshan Mufti 251694851

Introduction:

In this assignment, we are given instructions to create a simulation of a simple assembler that will deduce a file containing mips instructions which we have limited to ourselves in order to make it simpler to understand.

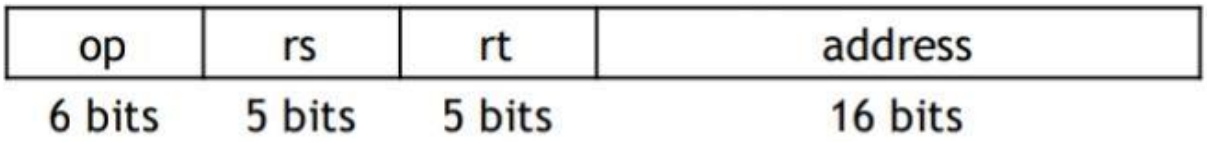
To understand how mips instructions work we should look at the format for the 3 types of mips instructions

R Type



The machine code of all instructions assigns certains bits out of 32 (because mips is 32 bit) to registers, the funct code, shift amount, opcode (in r type it specifies that it is rtype instruction when first 6 bits are 0)

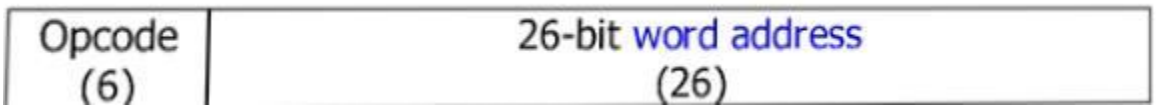
I Type



I type instruction never make use of three registers in one statement and are detected by the opcode field varying for different I type instruction. If opcode isnt just 0 we can say its an I type instruction. The rest will contains binary form of all components of the instruction no matter how its formatted

(addi $t1,$t1,4 , lw $t1,4($s1))

J Type



The j type instruction has a specific opcode that differentiate it from r type and I type and confirms that it's a j type instruction. 26 bits are stated for the address to which a jump should be made. This address is line number of the label to which it is trying to route the control of the program towards.

Now we know how these work lets build our logic.

- We shall traverse a mips file containing simple instructions.
- Store opcode of instructions in our program, registers in the order that yhey are in mips, functions thay we are focusing on.
- Separating handling for r and I type instructions and the varying I type instruction that have different placements
  Some I type have
  Instr rs,rt,address
  Instr rs,address(rt)
- Handle conversion of binary.
- Give accurate string of binary for each instruction
- Display our mips file and follow it by our converted machine code of each line

Code Explanation:

Let us take a look at the modules of our code

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Macros for opcodes
#define OPCODE_R    "000000"
#define OPCODE_ADDI "001000"
#define OPCODE_ORI "001101"
#define OPCODE_LW   "100011"
#define OPCODE_SW   "101011"
#define OPCODE_BEQ  "000100"
#define OPCODE_J    "000010"
#define FUNCT_ADD   "100000"
#define FUNCT_AND   "100100"
#define FUNCT_SLT   "101010"
#define FUNCT_OR   "100101"



// Instruction categories
const char *r_instructions[] = {"add", "and", "slt","or"};
const char *i_instructions[] = {"addi", "lw", "sw", "beq","ori"};
const char *j_instructions[] = {"j"};

//Ordering register based on MIPS
```

- Including base header files. Using string.h for the conversion of str to integer when traversing the mips file we will encounter $n registers where n is any digit.
- Defining macros for the opcode instructions we will focus on
- Making array for the instructions and puttinf them in their respective type array.

```c
//Ordering register based on MIPS
char *reg_names[] = {
    "$zero","$at","$v0","$v1","$a0","$a1","$a2","$a3",
    "$t0","$t1","$t2","$t3","$t4","$t5","$t6","$t7",
    "$s0","$s1","$s2","$s3","$s4","$s5","$s6","$s7",
    "$t8","$t9","$k0","$k1","$gp","$sp","$fp","$ra"
};

// Label table to store label addresses for j and beq instr
typedef struct {
    char label[20];
    int address;
} Label;

//List of label in code
Label labels[100];
//No of labels
int label_count = 0;
```

- Making an array for register names that could be in the mips programs.
- We have ordered this based on their order in MARS.
- Defining a struct called Label which will just have a name of label and its address (line number)
- Array consisting label type structs
- A counter for labels indicates , number of labels and length of the labels array

Main:

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./assembler file.asm\n");
        return 1;
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        perror("File open failed");
        return 1;
    }
```

- Making main function that command line arguments
- It will assume the second argument is filename and try open it. This jis asm file

```
//Collecting labels
char line[100];
int address = 0;
while (fgets(line, sizeof(line), fp)) {
    char *colon = strchr(line, ':');
    if (colon) {
        *colon = '\0';
        strcpy(labels[label_count].label, line);
        labels[label_count].address = address;
        label_count++;
    } else {
        address++;
    }
}
```

- Creating a buffer called line to store lines when using fgets
- Address specifitying the order in which labels came/ line number at which they came.
- Using fgets to traverse file line by line
- Checking for occurrence of colon in the line. This colon is the statements that have labels

- If it exists turn colon to a null terminator to make it a string
- Now we are left with the label name.
- Copying line to the name of a label stored in labels array, making that labels address the current address
- Increasing label count that indicates number of labels in code
- If no colon came that means its an instruction statement so skip it and increase the address

```
    rewind(fp); //rewinding file traversal so
    printf("Assembly language program\n");
    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), fp))
    fputs(buffer,stdout);
}

    printf("Tentative Machine Code:\n");

    rewind(fp);
    int line_num = 0;
    while (fgets(line, sizeof(line), fp)) {
        if (strchr(line, ':')) continue; //ignc
        process_line(line, line_num); //applyir
        line_num++; //collecting line number
    }

    fclose(fp); //closing file
    return 0;
}
```

- Rewind file traversal as we already traversed it once using fgets.
- Printing the output in the manner described in handout.
- First printing assembly file contents
- Rewinding again
- Checking for occurrence of colon (**we will not deduce labels**) continue to the next.

- Processing line based on our function which we will discuss, processline also print out the machine code string

```
int get_register_code(char *reg) {
    for (int i = 0; i < 32; i++) {
        if (strcmp(reg, reg_names[i]) == 0)
            return i;
    }
    if (reg[0] == '$' && !(reg[1] == 't' || reg[1] == 's' || reg[1] == 'v' ||
                           reg[1] == 'a' || reg[1] == 'k' || reg[1] == 'z' ||
                           reg[1] == 'r' || reg[1] == 'g' || reg[1] == 'f' ||
                           reg[1] == 'p'))
        return atoi(&reg[1]);
    return -1;
}

void to_binary(int num, int bits, char *out) {
    out[bits] = '\0';
    for (int i = bits - 1; i >= 0; i--) {
        out[i] = (num & 1) + '0';
        num >>= 1;
    }
```

Logic:

- This getregistercode function simply accepts a register passed from our logic in process line function.
- It will traverse are existing register and get its code by comparing the register we have with the incoming argument, when comparison is 0 returns the address of that register specified by a register.
- Handling the digit type tregisters
- First detecing that it is a register by confirming it has $ at the start.
- Confirming that the second occurrence isnt the valid register letters.
- Converting it to integer and returning it.

Binary:

- Binary conversion functions accepts a number, bits we need the binary in, the output array where binary is to be stored
- Terminating out array at the nnumbers of bits specified in order to use the space required. We do this by null character.
- Based on number of bits traverse the loop and fill the out array from left to right.

- On the left most unoccupied index, placing the binary of the least significatant bit of our number by AND with 1
- We need to plus with 0 to make it string.
- Shifting our num by 1 to the right. This handles the binary of our num by checking the bits of our number and repeatedly handling least significant bit everytime after shifting

Type Checking Functions

```c
//Traversing through r type instr array and comparing this lines instr with the ar
array, same for i and j type
int is_r_type(const char *instr) {
    for (int i = 0; i < sizeof(r_instructions)/sizeof(r_instructions[0]); i++) {
        if (strcmp(instr, r_instructions[i]) == 0)
            return 1;
    }
    return 0;
}

int is_i_type(const char *instr) {
    for (int i = 0; i < sizeof(i_instructions)/sizeof(i_instructions[0]); i++) {
        if (strcmp(instr, i_instructions[i]) == 0)
            return 1;
    }
    return 0;
}

int is_j_type(const char *instr) {
    for (int i = 0; i < sizeof(j_instructions)/sizeof(j_instructions[0]); i++) {
        if (strcmp(instr, j_instructions[i]) == 0)
            return 1;
    }
    return 0;
}
```

- These functions simply check and compare with our predefined arrays of instructions.

- If they exist in their respective arrays it returns 1 for yes and 0 for no

- Again we do this by comparing the instruction with all instructions in the array

Label Address Determination

```c
//Comparing label with the label array table to det
int get_label_address(const char *label) {
    for (int i = 0; i < label_count; i++) {
        if (strcmp(labels[i].label, label) == 0)
            return labels[i].address;
    }
    return -1;
}
```

- We only call this function when handling branch or j instructions
- Basically since all labels are already stored in labels array, we traverse through and find the label matching the one in our j or beq statements
- The same logic we have been doing a lot, where we traverse the array and compare with each element in the array.
- We will compare incoming label with labels attribute label and return the corresponding labels address which is also an attribute.

```
void process_line(char *line, int line_num) {
    char instr[10], arg1[20], arg2[20], arg3[20]; //scanning line till first space
are registers
    int matched = sscanf(line, "%s %[^,\n], %[^,\n], %[^,\n]", instr, arg1, arg2,

    if (is_r_type(instr)) { //matching register accordingly to mips
        int rd = get_register_code(arg1); //getting registers code/number in the a
as int
        int rs = get_register_code(arg2);
        int rt = get_register_code(arg3);
        char rs_bin[6], rt_bin[6], rd_bin[6], shamt[6] = "00000", funct[7];

        to_binary(rs, 5, rs_bin); //convert to binary
        to_binary(rt, 5, rt_bin);
        to_binary(rd, 5, rd_bin);

        if (strcmp(instr, "add") == 0) strcpy(funct, FUNCT_ADD); //putting funct m
        else if (strcmp(instr, "and") == 0) strcpy(funct, FUNCT_AND);
        else if (strcmp(instr, "slt") == 0) strcpy(funct, FUNCT_SLT);
        else if (strcmp(instr, "or") == 0) strcpy(funct, FUNCT_OR);

        printf("%s%s%s%s%s%s\n", OPCODE_R, rs_bin, rt_bin, rd_bin, shamt, funct);

    }
```

Process Line:

- Accepts a line and line number
- Creating bufffers for the components of the instructions
- Using scanf to give our buffers the right inputs
- Till first space we have instruction, after space and before newline or commas we have first register, after comma and till next comma or new line we have next register, after comma and till newline we have third register.

R Type Checking

- Checking if its r type
- Putting registers in rd rs rt based on our format and understanding of r type instructions
- We put register code in these variables which is a number

- Creating buffer for the binary version of these registers, shamt always 5 bits of 0, funct specifiers the functions already define in macros
- Converting register code to binary using our binary function that also is passed with our binary register buffer so that it can store our binary converted string in it
- If statement for filling up funct bits based on the actual instr we found in the line
- Putting all strings together.

```
else if (is_i_type(instr)) {
    if (strcmp(instr, "lw") == 0 || strcmp(instr, "sw") == 0) {
        sscanf(line, "%s %[^,], %s", instr, arg1, arg2); //scan line first is
        char offset_str[10], rs[10];
        sscanf(arg2, "%[^($](%[^)])", offset_str, rs); //arg2 has further divi
                                                       //whatever is before parenthes
        int rs_code = get_register_code(rs);
        int rt_code = get_register_code(arg1);
        int offset = atoi(offset_str);  //converting offset to int (string.h)

        char rs_bin[6], rt_bin[6], offset_bin[17];
        to_binary(rs_code, 5, rs_bin); //storing it in buffers based on their
        to_binary(rt_code, 5, rt_bin); //converting to binary
        to_binary(offset, 16, offset_bin);

        const char *opcode = strcmp(instr, "lw") == 0 ? OPCODE_LW : OPCODE_SW;
erwise sw
        printf("%s%s%s%s\n", opcode, rs_bin, rt_bin, offset_bin);
    }
```

I Type Checking:

Load and Store:

- First of all handling if instr is lw or sw
- Scanning 2nd argument as it is:

Offset(register)

- Scan line differently first. Character till first space is instruction, after space till comma is register, the rest is arg2. This is the format of lw, sw
- Arg2 is further scanned to separate offset and register
- Before parenthesis is the offset, inparethesis is register. This handles conditions like
  ($s2)
  $s2
  Where offset isnt specified. So it terminates as empty string at parethesis or $.
  Offset is 0 in this situation
- Storing offset and register codes in their respective buffers
- Making buffers for binary version of register and offset
- Passing our values based on the bits the require in the format to binaryfunction with their binary buffers to store the binary string we need
- If instr was lw or sw use the respective opcode
- Print based on format

```
else if (strcmp(instr, "beq") == 0) { //a different i type instr
    int rs = get_register_code(arg1);
    int rt = get_register_code(arg2); //only uses two register the
    int target_line = get_label_address(arg3); //getting labels ac


    if (target_line == -1) {
        printf("Label not found: %s\n", arg3);
        return;
    }

    int offset = target_line - (line_num + 1);
    char rs_bin[6], rt_bin[6], offset_bin[17];
    to_binary(rs, 5, rs_bin);
    to_binary(rt, 5, rt_bin);
    to_binary((offset & 0xFFFF), 16, offset_bin); // 2's complemer
our beq statement and the address of label

    printf("%s%s%s%s\n", OPCODE_BEQ, rs_bin, rt_bin, offset_bin);
```

Branch Equal:

- Storing arguments in respective variables to make it easier for us to map with format.
  Beq $t1,$t2,loop

Where loop is a label

- Get the line number of label by using our labels array and finding address of our particular label (already handled in get_label_address)
- If label not found, print
- Calculating distance between the line we are at which is linenum to the address of label which is the linr num of label
- This is our offset address
- Make buffers for arguments binary form
- Pass to binary converter
- 2s complement of offset to get the distance between our branch statement and label is also passed
- Prinitng it all based on format

Immediate Instruction

```
else { // addi or ori
    int rt = get_register_code(arg1);
    int rs = get_register_code(arg2);
    int immediate = atoi(arg3);

    char rs_bin[6], rt_bin[6], imm_bin[17];
    to_binary(rs, 5, rs_bin);
    to_binary(rt, 5, rt_bin);
    to_binary(immediate & 0xFFFF, 16, imm_bin);

    if (strcmp(instr, "ori") == 0)
    printf("%s%s%s%s\n", OPCODE_ORI, rs_bin, rt_bin, imm_bin);
    else
    printf("%s%s%s%s\n", OPCODE_ADDI, rs_bin, rt_bin, imm_bin);
}
```

- Getting arguments.
- Third argument is an immediate value
- Storing in respective buffers
- Pass to binary converted
- 2s complement of immediate to pass actual value of immediate

- If its ori, andi or addi handle it appropriately by changing opcode in the output.

Jump Instruction:

```
else if (is_j_type(instr)) {
    int target = get_label_address(arg1);
    if (target == -1) {
        printf("Label not found: %s\n", arg1);
        return;
    }
    char addr_bin[27];
    to_binary(target, 26, addr_bin); //just convert ine no of label to
    printf("%s%s\n", OPCODE_J, addr_bin);
}

else {
    printf("Unknown instruction: %s\n", instr);
}
```

- Simply consists of j opcode and label address (not distance between j and label)
- Creating address binary buffer
- Passing to binary converter.
- Printing in format
- Otherwise after covering all instructions, the instruction given by user isnt identified.

Outputs:

1:

The hex code given by mars for this code is

```
add $9,$10,$11
add $12,$9,$10
and $13,$10,$12
```

Code	Basic
0x014b4820	add $9,$10,$11
0x012a6020	add $12,$9,$10
0x014c6824	and $13,$10,$12

Converting hex code to binary:

00000001010010110100100000100000

00000001001010100110000000100000

00000001010011000110100000100100

Comparing with our program output:

```
Zeeshan@Ubuntu:~/Desktop/a2$ ./a2 mips2
Assembly language program
add $9,$10,$11
add $12,$9,$10
and $13,$10,$12
Tentative Machine Code:
00000001010010110100100000100000
00000001001010100110000000100000
00000001010011000110100000100100
```

2:

```
add $t1,$t2,$t3
sw $t1,($t2)
addi $t4,$t1,4
lw $t3,16($t2)
and $t5,$t2,$t4
add $9,$10,$11
add $12,$9,$10
and $13,$10,$12
add $s4,$t1,$t2
beq $at,$0,L
L:
add $v1,$v0,$v0
j L
```

The hex code:

Code	Basic
0x014b4820	add $9,$10,$11
0xad490000	sw $9,0($10)
0x212c0004	addi $12,$9,4
0x8d4b0010	lw $11,16($10)
0x014c6824	and $13,$10,$12
0x014b4820	add $9,$10,$11
0x012a6020	add $12,$9,$10
0x014c6824	and $13,$10,$12
0x012aa020	add $20,$9,$10
0x10200000	beq $1,$0,0
0x00421820	add $3,$2,$2

Converting to binary:

00000001010010110100100000100000

10101101010010010000000000000000

00100001001011000000000000000100

10001101010010110000000000010000

00000001010011000110100000100100

00000001010010110100100000100000

00000001001010100110000000100000

00000001010011000110100000100100

00000001001010101010000000100000

00010000001000000000000000000000

00000000010000100001100000100000

00001000000000000000000000000001010

Comparing with our program output:

```
Zeeshan@Ubuntu:~/Desktop/a2$ ./a2 mips.asm
Assembly language program
add $t1,$t2,$t3
sw $t1,($t2)
addi $t4,$t1,4
lw $t3,16($t2)
and $t5,$t2,$t4
add $9,$10,$11
add $12,$9,$10
and $13,$10,$12
add $s4,$t1,$t2
beq $at,$0,L
L:
add $v1,$v0,$v0
j L
Tentative Machine Code:
00000001010010110100100000100000
10101111111010010000011111101001
00100001001011000000000000000100
10001101010010110000000000010000
00000001010011000110100000100100
00000001010010110100100000100000
00000001001010100110000000100000
00000001010011000110100000100100
00000001001010101010000000100000
00010000001000000000000000000000
00000000010000100001100000100000
00001000000000000000000000001010
```

3:

```
slt $t0, $s1, $s2
beq $t0, $zero, skip1
addi $t1,$t1,22
sw $t1,15($t9)
skip1:
lw $t2,($s1)
ori $t2,$t1,5
or $t2,$t3,$t4
```

Code	Basic
0x0232402a	slt $8,$17,$18
0x11000002	beq $8,$0,2
0x21290016	addi $9,$9,22
0xaf29000f	sw $9,15($25)
0x8e2a0000	lw $10,0($17)
0x352a0005	ori $10,$9,5
0x016c5025	or $10,$11,$12

Converting to binary

00000010001100100100000000101010

00010001000000000000000000000010

00100001001010010000000000010110

10101111001010010000000000001111

10001110001010100000000000000000

00110101001010100000000000000101

00000001011011000101000000100101

Comparing with our program:

```
Zeeshan@Ubuntu:~/Desktop/a2$ ./a2 mips3.asm
Assembly language program
slt $t0, $s1, $s2
beq $t0, $zero, skip1
addi $t1,$t1,22
sw $t1,15($t9)
skip1:
lw $t2,($s1)
ori $t2,$t1,5
or $t2,$t3,$t4
Tentative Machine Code:
00000010001111110100000000101010
00010001000000000000000000000010
00100001001010010000000000010110
10101111001010010000000000001111
10001111001010100000000000001111
00110101001010100000000000000101
00000001011011000101000000100101
```