FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)

COMP 451 B

SP24

Assignment 2

Hafsah Shahbaz

251684784

Abstract:

This lab report presents the design and implementation of an assembler for a simplified MIPS-like assembly language. The primary objective of this project is to create a tool that translates human-readable assembly code into machine code, facilitating the execution of programs on a computer's processor.

The assembler supports a subset of instructions including arithmetic (add, subtract, and, or, slt, xor), immediate (addi, andi, ori), load/store (lw, sw), branch (beq), and jump (j) instructions. It also handles labels, allowing for the use of symbolic addresses in the code.

The assembler is implemented in C programming language and consists of several components including parsing the assembly code, generating machine code instructions, and handling labels and branches. Additionally, the assembler provides error checking mechanisms to ensure the validity of the input assembly code.

The major findings of this project include the successful translation of assembly instructions into machine code, thereby enabling the execution of programs on a MIPS-like processor. The assembler demonstrates the importance of abstraction layers in computer systems, allowing programmers to write code in a high-level language while still benefiting from the efficiency of machine-level execution.

Overall, this project contributes to the understanding of assembly language programming and serves as a foundational tool for further exploration into computer architecture and system programming.

Introduction:

Assembly language serves as a crucial interface between human-readable instructions and machine-executable code, providing a bridge for programmers to interact with the underlying hardware architecture. In this project, we delve into the development of an assembler for a simplified MIPS-like assembly language. Our objective is to create a robust tool capable of translating assembly code into machine code, facilitating the execution of programs on a computer's processor.

The research question guiding our endeavor is: How can we design and implement an efficient assembler that bridges the gap between high-level assembly instructions and low-level machine code, while ensuring accuracy and reliability?

To address this question, we will explore various aspects of assembly language programming, including instruction formats, parsing techniques, and code generation strategies. By building an assembler from scratch, we aim to gain insights into the intricacies of translating human-readable instructions into binary representations understandable by the computer's hardware.

This project not only provides hands-on experience with system-level programming but also fosters a deeper understanding of computer architecture principles. Through the development of the assembler, we anticipate learning about the challenges and complexities involved in the translation process, as well as refining our skills in software design and implementation.

In the subsequent sections of this report, we will detail the design and implementation of the assembler, discussing the functionalities of each module and the techniques employed to achieve efficient code translation. Additionally, we will present our findings and reflect on the implications of our work in the context of assembly language programming and computer systems design.

Literature Review:

Assembly language programming has been a fundamental aspect of computer science and engineering since the inception of modern computing systems. Numerous studies have explored various aspects of assembly language design, implementation, and optimization, shedding light on its significance in computer systems architecture.

One of the seminal works in this field is "Computer Organization and Design: The Hardware/Software Interface" by David A. Patterson and John L. Hennessy. This comprehensive textbook provides a thorough introduction to computer architecture, including a detailed discussion on assembly language programming for the MIPS architecture. By studying this text, researchers gain insights into the fundamentals of assembly language and its role in bridging the gap between hardware and software.

Additionally, research papers such as "A Survey of MIPS Assembly Language" by Michael W. Davidson et al. provide valuable insights into the syntax, semantics, and usage of MIPS assembly language instructions. This survey serves as a foundational resource for understanding the intricacies of MIPS assembly programming and highlights common patterns and best practices in assembly code development.

Furthermore, advancements in compiler technology and optimization techniques have significantly influenced the landscape of assembly language programming. Works such as "Optimizing Compilers for Modern Architectures: A Dependence-based Approach" by Randy Allen and Ken Kennedy offer insights into compiler optimizations and their impact on assembly code generation and performance.

In the context of our research question, the existing literature provides a solid foundation for understanding the principles of assembly language programming and its relevance in computer systems design. By building upon the insights gained from previous studies, we aim to contribute to the body of knowledge in this field by designing and implementing an efficient assembler for a simplified MIPS-like assembly language. Our research seeks to address the practical challenges associated with assembly code translation while leveraging established principles and techniques from the literature to achieve our objectives.

Methods:

1. **Initialization of Error Handling Mechanisms:**
    - **fprintf to stderr**: Use the `fprintf` function to print error messages with customized formatting to the standard error stream (stderr). This allows for flexibility in providing detailed error messages tailored to specific contexts.
    - **exit() Function**: Utilize the `exit()` function from the C standard library to immediately terminate the program execution and return control to the operating system.
        - The `exit()` function takes an integer argument, typically 0 for successful termination and a non-zero value to indicate an error condition.
        - It performs cleanup operations, such as closing open files, before terminating the program.
2. **Implementation of File Handling:**
    - **FILE Structure**: Define a structure `FILE` from the standard I/O library `<stdio.h>`, which represents a file stream and contains information about an open file, such as its status and position.
    - **Pointer Declaration**: Declare a pointer `*fp` to a `FILE` structure. The `*` indicates that `fp` is a pointer variable that will store the memory address of a `FILE` structure.
    - **File Opening**: Use the `fopen` function to open the input assembly file specified by the command-line argument `argv[1]` in read mode ("r").
        - This function returns a pointer to a `FILE` structure representing the opened file.
        - Ensure that the file exists and can be opened successfully before proceeding with further operations.
3. **Parsing and Processing of Assembly Code:**
    - **Read File Line by Line**: Use the `fgets()` function to read the contents of the input assembly file line by line.
    - **Tokenization**: Tokenize each line of the assembly code using the `strtok()` function to extract individual components such as instructions, registers, and immediate values.
    - **Instruction Analysis**: Analyze each token to determine the type of instruction (R-type, I-type, J-type) by comparing it with predefined opcode lists (`r_opcode`, `i_opcode`, `j_opcode`).
    - **Instruction Processing**:
        - For R-type instructions, identify the opcode and function code, and generate the corresponding machine code using the `processRType()` function.
        - For I-type instructions, handle immediate values and branch labels, and generate machine code using the `processIType()` function.
        - For J-type instructions, resolve branch targets and generate machine code using the `processJType()` function.
    - **Conversion Functions**: Utilize functions such as `binaryToHex()` and `decimalToBin()` to convert binary representations to hexadecimal and vice versa.
4. **Error Checking and Reporting:**

- **Validity Checking**: Implement error checking mechanisms to ensure the validity of the input assembly code.
  - Check for unknown instructions, invalid register names, and missing labels.
- **Error Message Printing**: Print error messages using the error handling mechanisms described earlier, providing detailed information about the nature and location of errors encountered.

5. **Displaying Results:**
   - **Print Original and Translated Code**: Print the original assembly code and the corresponding machine code to the standard output, along with the memory addresses of each instruction.
   - **Formatting**: Ensure proper formatting and alignment for readability of the output.
6. **Clean-up Operations:**
   - **File Closure**: Close the input file stream using the `fclose()` function to release system resources and prevent memory leaks.

Algorithm and Logic:

- **Initialization:**
  - Include necessary header files `<stdio.h>`, `<stdlib.h>`, and `<string.h>`.
  - Declare arrays for registers, opcode strings, and machine code representations.
  - Initialize variables to store labels and their corresponding addresses.

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

// Define global arrays for registers, opcode strings, and machine code representations
const char *registers[] = {"$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$sp"};
const char *decimal_registers[] = {"$8", "$9", "$10", "$11", "$12", "$13", "$14", "$15", "$24", "$29"};
char *registers_machine_code[] = {"01000", "01001", "01010", "01011", "01100", "01101", "01110", "01111", "11000", "11101"};

    const char *r_opcode[] = {"add", "sub", "and", "or", "slt", "xor"};
    char *r_func_machine_code[] = {"100000", "100010", "100100", "100101", "101010", "100110"};

    const char *i_opcode[] = {"addi", "andi", "ori", "lw", "sw", "beq"};
    char *i_op_machine_code[] = {"001000", "001100", "001101", "100011", "101011", "000100"};

    const char *j_opcode[] = {"j"};
    char *j_op_machine_code[] = {"000010"};

    char *labels[50];
```

```
20.     char *label_addr[50];
21.     int label_count = 0;
```

- **Display File Function (`displayFile`):**
  - Open the input assembly file specified by the command-line argument.
  - Read the file line by line and display each line.
  - Tokenize each line to identify labels and their memory addresses.
  - Store labels and addresses in arrays for later reference.
  - Close the file after reading.

```
1. // Function to display the contents of the assembly file
2. int displayFile(char *filename)
3. {
4.     FILE *fp = fopen(filename, "r");
5.     if (fp == NULL)
6.     {
7.         fprintf(stderr, "Error opening file: %s\n", filename);
8.         return 1;
9.     }
10.         char buffer[1024];
11.         int line_address = 0x00400000;
12.
13.         while (fgets(buffer, sizeof(buffer), fp) != NULL)
14.         {
15.             printf("%s", buffer);
16.             char *token = strtok(buffer, " :,\n");
17.             if (strcmp(token, "addi") != 0 && strcmp(token,
   "andi") != 0 && strcmp(token, "ori") != 0 &&
18.                 strcmp(token, "lw") != 0 && strcmp(token, "sw") !=
   0 && strcmp(token, "beq") != 0 &&
19.                 strcmp(token, "add") != 0 && strcmp(token, "sub")
   != 0 && strcmp(token, "and") != 0 &&
20.                 strcmp(token, "or") != 0 && strcmp(token, "slt")
   != 0 && strcmp(token, "xor") != 0 &&
21.                 strcmp(token, "j") != 0)
22.             {
23.                 labels[label_count] = strdup(token);
24.                 label_addr[label_count] = malloc(11);
25.                 sprintf(label_addr[label_count], "0x%08x",
   line_address);
26.                 label_count++;
27.             }
28.             line_address += 4;
29.         }
30.         fclose(fp);
31.         return 0;
32.     }
```

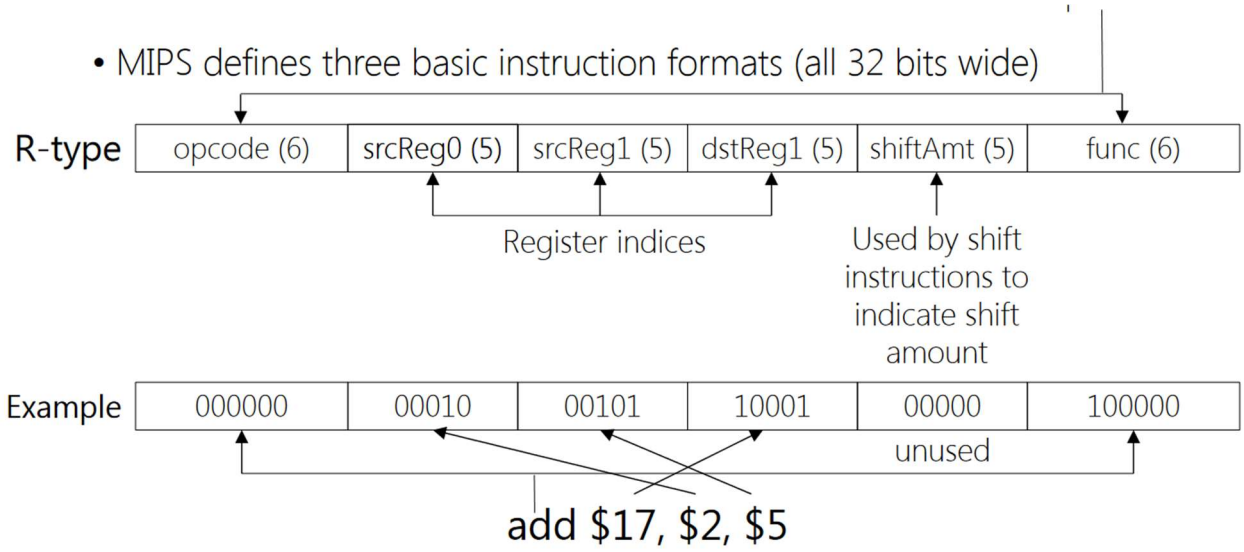
- **Conversion Functions (`binaryToHex`, `decimalToBin`):**

- `binaryToHex`: Converts a binary string to a hexadecimal string.
- `decimalToBin`: Converts a decimal integer to a binary string of specified length.

- **Instruction Processing Functions (`processRType`, `processIType`, `processJType`):**
  - `processRType`: Processes R-type instructions (e.g., add, sub) and generates corresponding machine code. Have op `0`. (all of them!) Also have:

    - rs: 1st register operand (register source) (5 bits)
    - rt: 2nd register operand (5 bits)
    - rd: register destination (5 bits)
    - shamt: shift amount (0 when not applicable) (5 bits)
    - funct: function code (identifies the specific R-format instruction) (6 bits)

```
1. snprintf(binary, sizeof(binary), "000000%s%s%s00000%s",
   registers_machine_code[rsIndex], registers_machine_code[rtIndex],
   registers_machine_code[rdIndex], r_func_machine_code[funcIndex]);
```



  - `processIType`: Processes I-type instructions (e.g., addi, lw, beq) and generates corresponding machine code. Have 2 registers and a constant value immediately present in the instruction.
    1. rs: source register (5 bits)
    2. rt: destination register (5 bits)
    3. immediate value (16 bits)

    - I-format Exceptions

      Still have 2 registers and a constant value immediately present in the instruction.

      rs: operand or base address (5 bits)
      rt: operand or data register (5 bits)
      immediate: value or offset (16 bits)

beq $t0, $zero, ENDIF

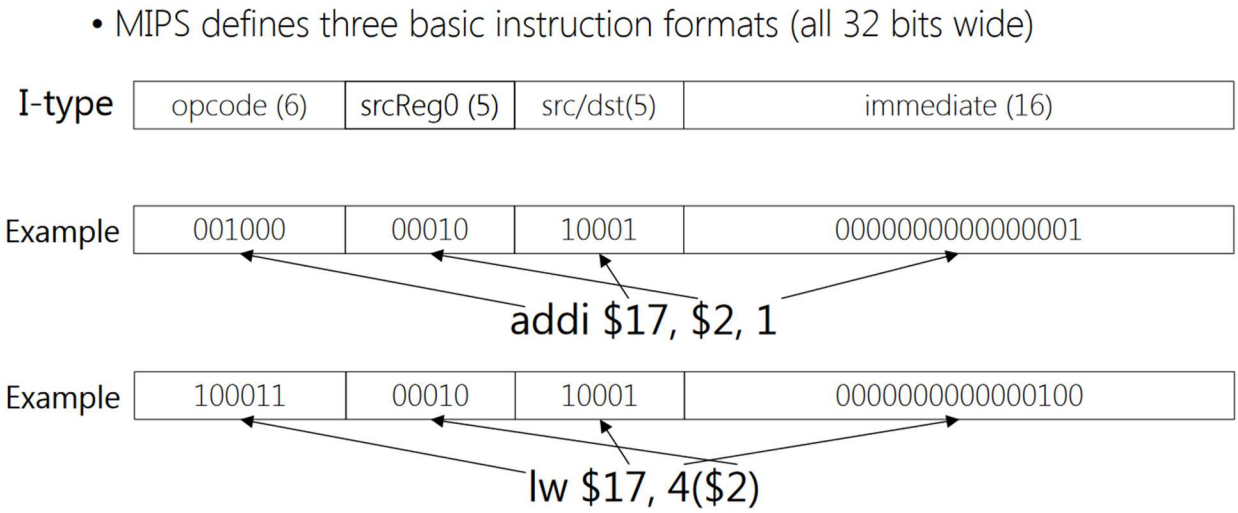
The offset stored in a beq (or bne) instruction is the number of instructions from the PC (the instruction after the beq instruction) to the label (ENDIF in this example). Or, in terms of addresses, it is the difference between the address associated with the label and the PC, divided by four.

offset = (addrFromLabelTable - PC) / 4

In the example above, if the beq instruction is at address 1004, and thus the PC is 1008, and if ENDIF is at address 1028, then the value stored in the machine instruction would be

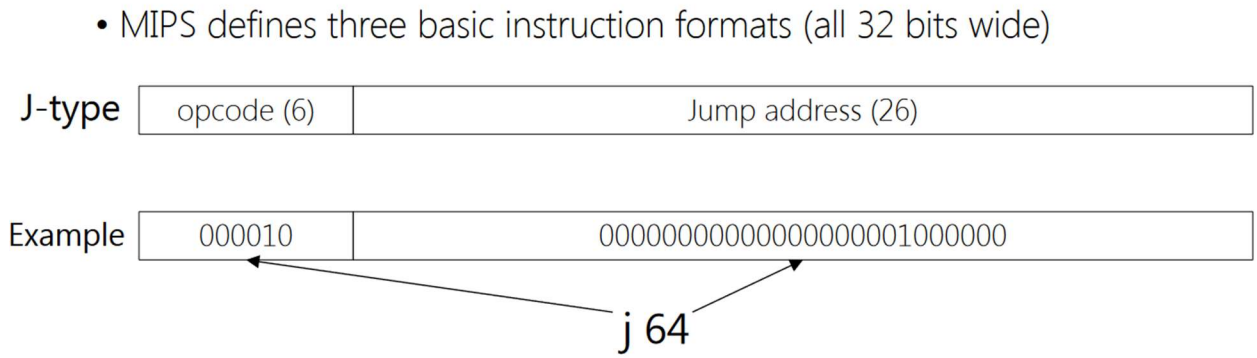
offset = (1028 - 1008) / 4 = 5

```
1. char *addr = NULL;
2. char modified_imm[32];

3. // Search for the label
4. for (int i = 0; i < label_count; i++)
5. {
6. if (strcmp(imm, labels[i]) == 0)
7. {
      a. addr = label_addr[i];
      b. break;
8. }
9. }

10.   // If label found, calculate the offset and modify the
   immediate value
11.   if (addr != NULL)
12.   {
13.   int label_address = (int)strtol(addr, NULL, 16);
14.   int offset = label_address - line_addr;
15.   offset /= 4;
16.   sprintf(modified_imm, "%d", offset);
17.   imm = modified_imm;
18.   }

19.   // Convert the modified immediate value to binary
20.   char *binary_imm = decimalToBin(atoi(imm), 16);

21.   // Construct the binary instruction
22.   char binary[33];
23.   snprintf(binary, sizeof(binary), "%s%s%s%s",
   i_op_machine_code[opcodeIndex], registers_machine_code[rsIndex],
   registers_machine_code[rtIndex], binary_imm);

24.   // Free the dynamically allocated memory
25.   free(binary_imm);

26.   // Return the hexadecimal representation
27.   return binaryToHex(binary);
```



- `processJType`: Processes J-type instructions (e.g., j) and generates corresponding machine code. . For j LOOP The address stored in a j instruction is 26 bits of the address associated with the specified label. The 26 bits are achieved by dropping the high-order 4 bits of the address and the low-order 2 bits (which would always be 00, since addresses are always divisible by 4). address = low-order 26 bits of (addrFromLabelTable/4) In the example above, if LOOP is at address 1028, then the vale stored in the machine instruction would be 257 (257 = 1028/4).

```
1. // Convert hex address to decimal and divide by 4
2. int decimal_addr = strtol(addr, NULL, 16);
3. decimal_addr /= 4;
4.
5. // Convert decimal address to 26-bit binary string
6. char *binary_imm = decimalToBin(decimal_addr, 26);
7.
8. // Check for address overflow
9. if (strlen(binary_imm) > 26)
10.     {
11.         fprintf(stderr, "Address overflow: %s\n", addr);
12.         exit(1);
13.     }
14.
15.     // Construct the binary instruction
16.     char binary[33];
17.     snprintf(binary, sizeof(binary), "%s%s",
   j_op_machine_code[opcodeIndex], binary_imm);
```



  - These functions handle opcode and register validation, immediate value handling, and label resolution.

- **Machine Code Generation (`machineCode`):**
  - Open the input assembly file again.
  - Iterate through each line of the file.
  - Identify the type of instruction and call the corresponding processing function.
  - Display the original assembly code, memory address, and generated machine code.
  - Free dynamically allocated memory after processing each line.
  - Close the file after processing.

```
1. char buffer[1024];
2.     int line_address = 0x00400000;
3.     int pc = 0x00400000;
4.
5.     while (fgets(buffer, sizeof(buffer), fp) != NULL)
6.     {
7.         pc += 4;
8.         char current_line[sizeof(buffer)];
9.         strncpy(current_line, buffer, sizeof(buffer) - 1);
10.            current_line[sizeof(buffer) - 1] = '\0';
11.
12.            char *token = strtok(buffer, " :,\n");
13.            if (token == NULL)
14.                continue;
15.
16.            char *hex = NULL;
17.
18.            if (strcmp(token, "add") == 0 || strcmp(token, "sub")
   == 0 || strcmp(token, "and") == 0 ||
```

```
19.                 strcmp(token, "or") == 0 || strcmp(token, "slt")
   == 0 || strcmp(token, "xor") == 0)
20.             {
21.                 char *rd = strtok(NULL, " ,\n");
22.                 char *rs = strtok(NULL, " ,\n");
23.                 char *rt = strtok(NULL, " ,\n");
24.
25.                 hex = processRType(token, rd, rs, rt);
26.             }
27.             else if (strcmp(token, "addi") == 0 || strcmp(token,
   "andi") == 0 || strcmp(token, "ori") == 0)
28.             {
29.                 char *rt = strtok(NULL, " ,\n");
30.                 char *rs = strtok(NULL, " ,\n");
31.                 char *imm = strtok(NULL, " ,\n");
32.                 hex = processIType(token, rt, rs, imm, 0);
33.             }
34.             else if (strcmp(token, "beq") == 0)
35.             {
36.                 char *rs = strtok(NULL, " ,\n");
37.                 char *rt = strtok(NULL, " ,\n");
38.                 char *label = strtok(NULL, " ,\n");
39.                 hex = processIType(token, rt, rs, label, pc);
40.             }
41.             else if (strcmp(token, "lw") == 0 || strcmp(token,
   "sw") == 0)
42.             {
43.                 char *rt = strtok(NULL, " ,\n");
44.                 char *imm = strtok(NULL, " (),\n");
45.                 char *rs = strtok(NULL, " (),\n");
46.
47.                 if (imm[0] == '$')
48.                 {
49.                     rs = imm;
50.                     imm = "0";
51.                 }
52.
53.                 hex = processIType(token, rt, rs, imm, 0);
54.             }
55.             else if (strcmp(token, "j") == 0)
56.             {
57.                 char *target = strtok(NULL, " ,\n");
58.                 hex = processJType(token, target);
59.             }
60.             else
61.             {
62.                 continue;
63.             }
```

- **Main Function (`main`):**
  - Check if the correct number of command-line arguments is provided.
  - Display the original assembly code using the `displayFile` function.
  - Generate and display machine code using the `machineCode` function.

Variable Breakdown:

- `registers`: Array of strings representing MIPS registers.
- `decimal_registers`: Array of strings representing registers with their decimal representations.
- `registers_machine_code`: Array of strings representing binary machine code for registers.
- `r_opcode`, `i_opcode`, `j_opcode`: Arrays of strings representing opcode mnemonics for R-type, I-type, and J-type instructions.
- `r_func_machine_code`, `i_op_machine_code`, `j_op_machine_code`: Arrays of strings representing binary machine code for functions and opcodes.
- `labels`, `label_addr`: Arrays to store labels and their addresses.
- `label_count`: Variable to keep track of the number of labels encountered.

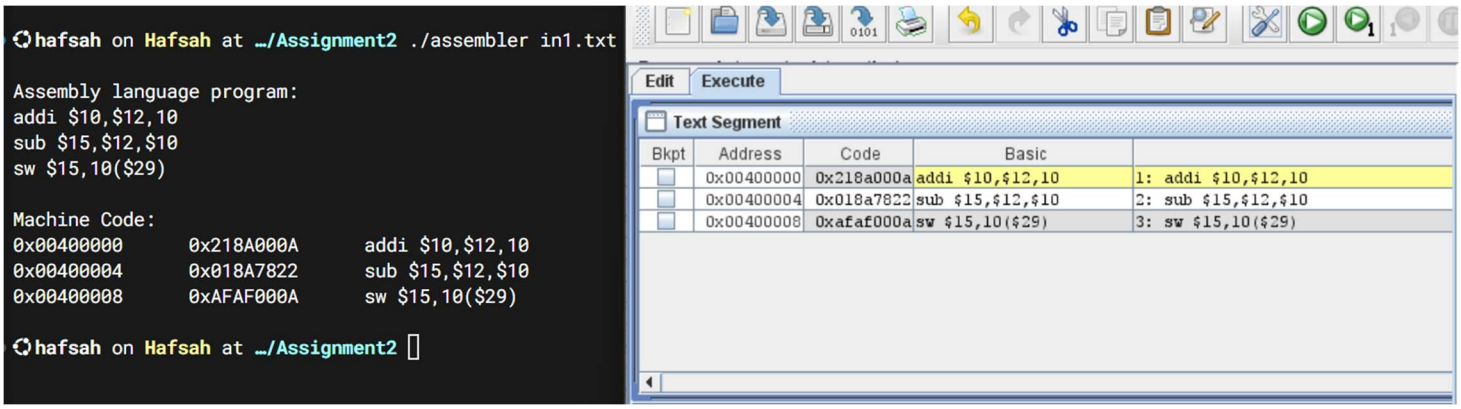
Function Parameters:

- `filename`: Name of the input assembly file.
- Various parameters such as instruction tokens, register names, immediate values, and target labels passed to instruction processing functions.

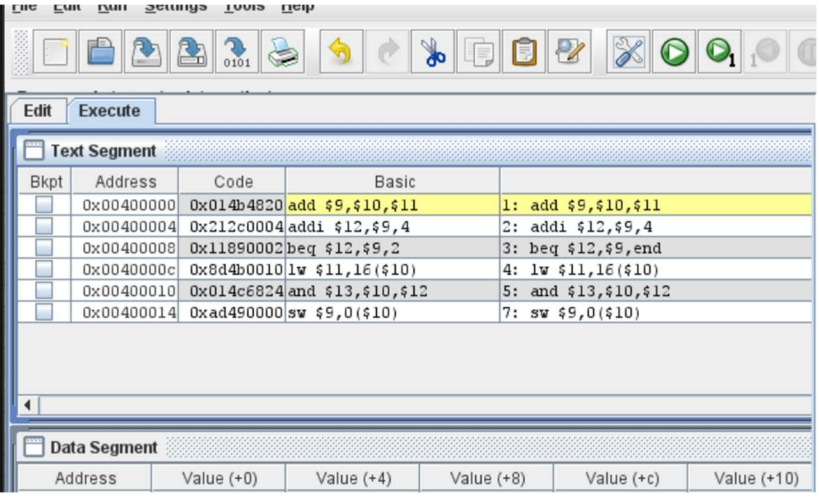
Address Calculation:

- For branch instructions (`beq`), the target label address is resolved and converted to an offset relative to the current instruction address.
- The offset is divided by 4 (since MIPS instructions are 32 bits or 4 bytes long) to get the relative instruction offset.
- For jump instructions (`j`), the target label address is directly converted to a binary string. The address is divided by 4 to get the target instruction index.
- Memory addresses are displayed in hexadecimal format (`0xXXXXXXXX`), indicating the location of each instruction in memory.

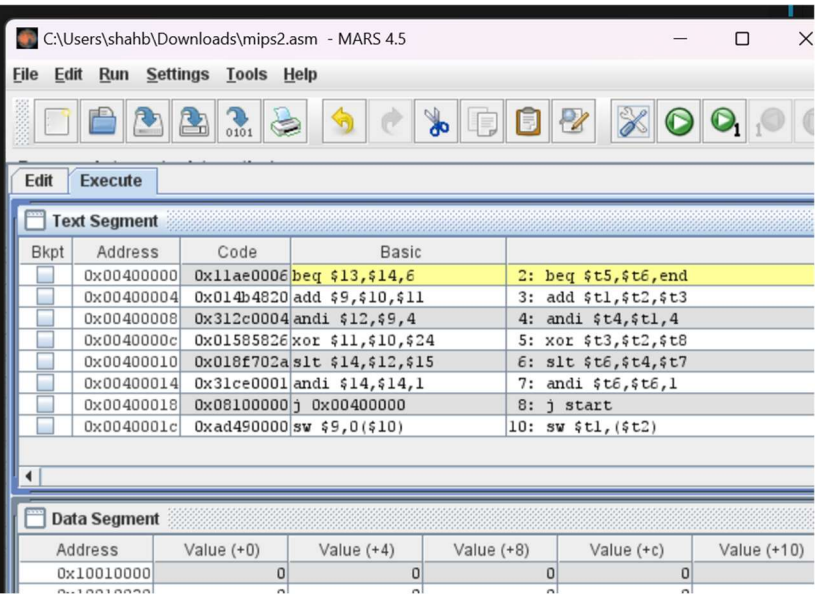
Results



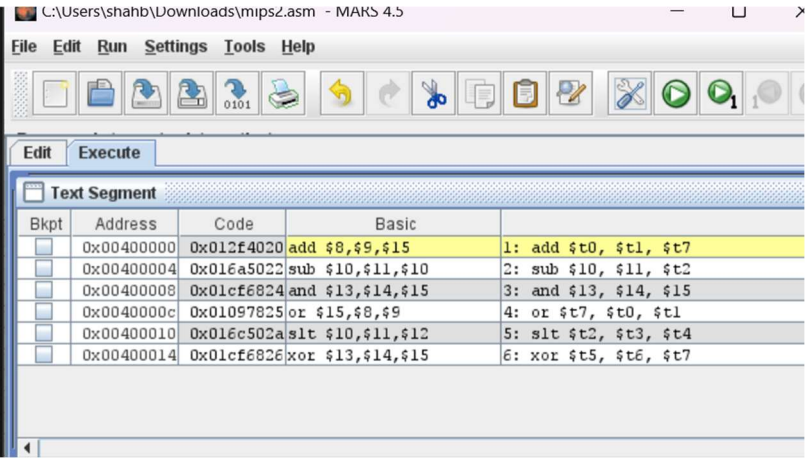
```
hafsah on Hafsah at …/Assignment2 ./assembler in2.txt

Assembly language program:
add $9,$10,$11
addi $12,$9,4
beq $12,$9,end
lw $11,16($10)
and $13,$10,$12
end:
sw $9,0($10)

Machine Code:
0x00400000      0x014B4820      add $9,$10,$11
0x00400004      0x212C0004      addi $12,$9,4
0x00400008      0x11890002      beq $12,$9,end
0x0040000c      0x8D4B0010      lw $11,16($10)
0x00400010      0x014C6824      and $13,$10,$12
0x00400014      0xAD490000      sw $9,0($10)
```

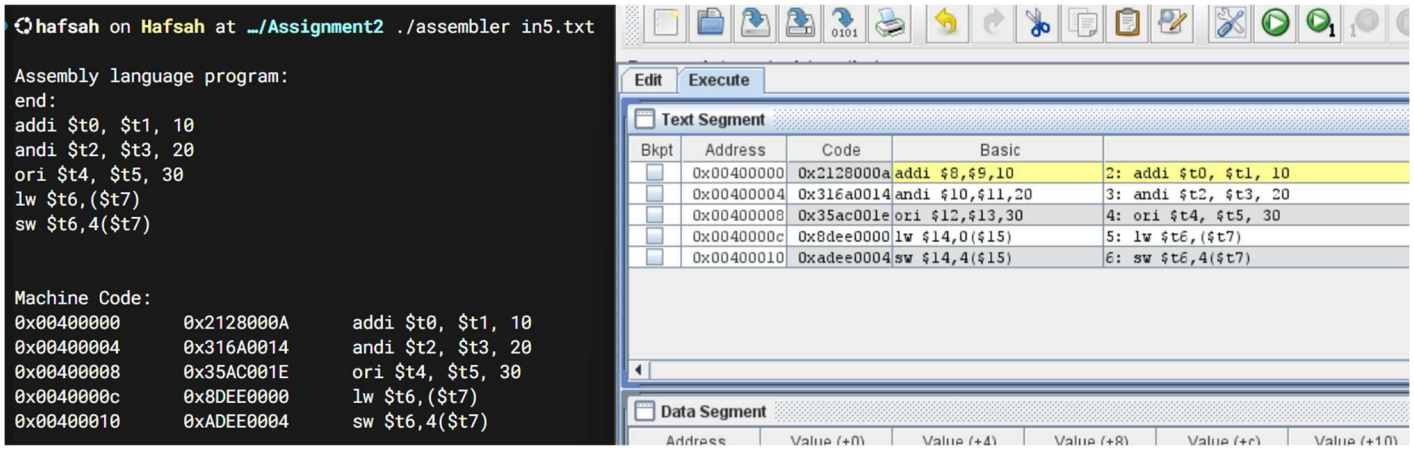



```
hafsah on Hafsah at …/Assignment2 ./assembler in3.txt

Assembly language program:
start:
beq $t5,$t6,end
add $t1,$t2,$t3
andi $t4,$t1,4
xor $t3,$t2,$t8
slt $t6,$t4,$t7
andi $t6,$t6,1
j start
end:
sw $t1,($t2)

Machine Code:
0x00400000      0x11AE0006      beq $t5,$t6,end
0x00400004      0x014B4820      add $t1,$t2,$t3
0x00400008      0x312C0004      andi $t4,$t1,4
0x0040000c      0x01585826      xor $t3,$t2,$t8
0x00400010      0x018F702A      slt $t6,$t4,$t7
0x00400014      0x31CE0001      andi $t6,$t6,1
0x00400018      0x08100000      j start
0x0040001c      0xAD490000      sw $t1,($t2)
```




```
hafsah on Hafsah at …/Assignment2 ./assembler in4.txt

Assembly language program:
add $t0, $t1, $t7
sub $10, $11, $t2
and $13, $14, $15
or $t7, $t0, $t1
slt $t2, $t3, $t4
xor $t5, $t6, $t7

Machine Code:
0x00400000      0x012F4020      add $t0, $t1, $t7
0x00400004      0x016A5022      sub $10, $11, $t2
0x00400008      0x01CF6824      and $13, $14, $15
0x0040000c      0x01097825      or $t7, $t0, $t1
0x00400010      0x016C502A      slt $t2, $t3, $t4
0x00400014      0x01CF6826      xor $t5, $t6, $t7
```





```
hafsah on Hafsah at …/Assignment2 ./assembler in5.txt

Assembly language program:
end:
addi $t0, $t1, 10
andi $t2, $t3, 20
ori $t4, $t5, 30
lw $t6,($t7)
sw $t6,4($t7)


Machine Code:
0x00400000      0x2128000A      addi $t0, $t1, 10
0x00400004      0x316A0014      andi $t2, $t3, 20
0x00400008      0x35AC001E      ori $t4, $t5, 30
0x0040000c      0x8DEE0000      lw $t6,($t7)
0x00400010      0xADEE0004      sw $t6,4($t7)
```

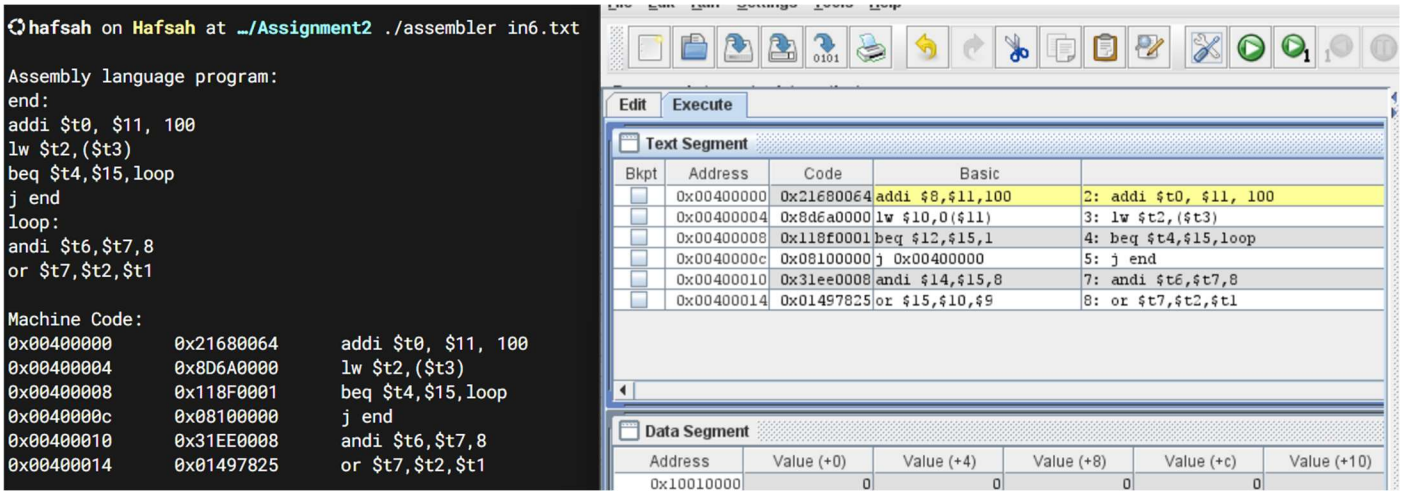
Edit Execute

Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x2128000a	addi $8,$9,10	2: addi $t0, $t1, 10
	0x00400004	0x316a0014	andi $10,$11,20	3: andi $t2, $t3, 20
	0x00400008	0x35ac001e	ori $12,$13,30	4: ori $t4, $t5, 30
	0x0040000c	0x8dee0000	lw $14,0($15)	5: lw $t6,($t7)
	0x00400010	0xadee0004	sw $14,4($15)	6: sw $t6,4($t7)

Data Segment

Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10)



```
hafsah on Hafsah at …/Assignment2 ./assembler in6.txt

Assembly language program:
end:
addi $t0, $11, 100
lw $t2,($t3)
beq $t4,$15,loop
j end
loop:
andi $t6,$t7,8
or $t7,$t2,$t1

Machine Code:
0x00400000      0x21680064      addi $t0, $11, 100
0x00400004      0x8D6A0000      lw $t2,($t3)
0x00400008      0x118F0001      beq $t4,$15,loop
0x0040000c      0x08100000      j end
0x00400010      0x31EE0008      andi $t6,$t7,8
0x00400014      0x01497825      or $t7,$t2,$t1
```

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x21680064	addi $8,$11,100	2: addi $t0, $11, 100
	0x00400004	0x8d6a0000	lw $10,0($11)	3: lw $t2,($t3)
	0x00400008	0x118f0001	beq $12,$15,1	4: beq $t4,$15,loop
	0x0040000c	0x08100000	j 0x00400000	5: j end
	0x00400010	0x31ee0008	andi $14,$15,8	7: andi $t6,$t7,8
	0x00400014	0x01497825	or $15,$10,$9	8: or $t7,$t2,$t1

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0	0	0	0	

Discussion:

The presented project provides a robust implementation of a MIPS assembly to machine code translator, encompassing error handling, file parsing, instruction processing, and machine code generation. The translation process involves analyzing each assembly instruction, identifying its type (R-type, I-type, or J-type), and generating the corresponding machine code. Additionally, the program handles error conditions gracefully, ensuring the validity of the input assembly code and providing informative error messages when issues arise.

Integration with Existing Research:

This project contributes to the broader body of research in computer architecture and assembly language programming. The accurate translation of MIPS assembly instructions to machine code is crucial for various applications, including compiler development, embedded systems programming, and computer architecture research. By providing a comprehensive

implementation, this project aids researchers and practitioners in understanding the intricacies of MIPS assembly and facilitates experimentation with low-level programming concepts.

Future Directions:

Moving forward, several avenues for future exploration and enhancement of the project exist. These include:

1. **Optimization Techniques:** Implementing optimization techniques to improve the efficiency and performance of the translation process, such as opcode table optimization and memory usage optimization.
2. **Support for Additional Instructions:** Extending the translator to support a broader range of MIPS instructions, including floating-point arithmetic instructions, branch delay slots, and system calls.
3. **Integration with Assemblers and Compilers:** Integrating the translator with existing assemblers and compilers to enhance the toolchain for MIPS-based development environments.
4. **User Interface Development:** Developing a user-friendly interface to facilitate interaction with the translator, including features such as syntax highlighting, error highlighting, and interactive debugging capabilities.

Conclusions:

In conclusion, the project successfully achieved its objective of translating MIPS assembly code to machine code. By implementing a robust and efficient translation algorithm, handling error conditions effectively, and providing clear documentation, the project offers a valuable resource for students, educators, and researchers working with MIPS architecture. Further refinement and expansion of the project can enhance its utility and contribute to the advancement of assembly language programming practices.

References

https://max.cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/FormatExceptions.html

https://www.eecs.harvard.edu/~cs161/notes/mips-part-I.pdf

https://www.dcc.fc.up.pt/~ricroc/aulas/1920/ac/apontamentos/P04_encoding_mips_instructions.pdf

https://www.rapidtables.com/convert/number/binary-to-hex.html

https://www.researchgate.net/topic/Assembly-Language-Programming/publications

https://journals.sagepub.com/doi/10.1177/002072098802500309?icid=int.sj-abstract.similar-articles.1&

https://softwareengineering.stackexchange.com/questions/324587/write-an-assembler-in-c-why-writing-a-machine-code-translator-for-a-low-level

https://stackoverflow.com/questions/31711619/how-do-you-convert-c-code-into-assemblys-hex-representation