Lab 2 - DLX Assembler

Nate Herbst
A02307138
Nathan Walker
A02364124

Introduction

This project required making a program that can take .dlx files in and output .mif files. This required the use of the C programming language to parse the input file, interpret the text found within, and write it to an output file with the correct instruction set. An optional task was to also have comments present in the .mif files from the assembly file.

Procedure

Data Structures:

- Making efficient data structures to hold all the different types of op-codes and ways to hold labels for when the program would jump. This lead to using the C struct and pointers to dynamically allocate memory for how many labels were used in the file and link them to the correct op-code jumps.

Parsing the file

- The C language has the ability to grab lines out of a file as a char*. Then using the strtok function to allow the program to grab “tokens” from the line (i.e. “R0”, ADD, .text). Once the program was able to grab lines and grab tokens, it was a matter of understanding what the tokens meant and outputting it to the .mif files.

Challenges and Solutions:

- The initial challenge in the lab was finding an efficient way that made sense to parse the file. After looking at different methods, grabbing the lines in the file and then the tokens seemed the simplest way to grab the information. The second challenge of the lab was ensuring correct formatting to the output file. This was tricky to ensure that the .mif files matched the format from the assignment description. However, learning some neat C fprintf formats this turned to be not too bad.

Results

The final implementation allows us to read in .dlx files and output .mif files that can eventually be read in by a DLX processor and ran. The .mif files also include helpful comments so we can look back at the memory files and understand how the op-codes translate to instructions.

Conclusion

This lab provided valuable insights into how to build an assembler. This was a fun exercise of abilities to make something custom to the course. Building a custom assembler for our DLX processor and being able to add or remove op-codes to the assembler is a cool feature that we can further explore.

Appendix

Assembler.c (top level file)

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

/**
* @brief Converts a register string (e.g., "R1", "r2") into its integer
representation.
*
* @param token The string token to parse.
* @return The register number (0-31) if valid, or -1 if the token is not a
valid register.
*/
int get_register(char* token) {
   if (token == NULL) return -1;
   // Registers are expected to start with 'R' or 'r'.
   if (token[0] == 'R' || token[0] == 'r') {
       // atoi converts the numeric part of the string to an integer.
       int reg = atoi(token + 1);
       if (reg >= 0 && reg <= 31) return reg;
   }
   return -1;
}

/**
* @brief Looks up the address of a given label in an array of labels.
*
* @param token The name of the label to find.
* @param labels An array of `label` structs to search within.
* @param count The number of labels in the array.
* @return The address of the label if found, otherwise -1.
*/
int get_label_address(char* token, label* labels, int count) {
   if (token == NULL) return -1;
   for (int i = 0; i < count; i++) {
```

```
        if (strcmp(token, labels[i].label) == 0) {
            return labels[i].address;
        }
    }
    return -1;
}

/**
* @brief Gets a numeric value from a string token.
*
* This function is used for instruction operands that can be either an
immediate value
* or a label. It first tries to resolve the token as a label, and if that
fails,
* it parses it as a number (decimal or hexadecimal).
*
* @param token The string token to parse.
* @param labels The array of known labels.
* @param count The number of labels in the array.
* @return The resolved numeric value.
*/
int get_value(char* token, label* labels, int count) {
    if (token == NULL) return 0;
    // First, check if the token is a known label.
    int addr = get_label_address(token, labels, count);
    if (addr != -1) return addr;

    // If not a label, parse it as an integer. strtol with base 0
    // automatically handles decimal and hexadecimal (if "0x" prefix is
present).
    return (int)strtol(token, NULL, 0);
}

/**
* @brief Prints the standard header for an Intel MIF (Memory
Initialization File).
*
* @param fo A file pointer to the output MIF file.
*/
void print_header(FILE* fo){
```

```
fprintf(fo, "DEPTH = 1024;\n");
fprintf(fo, "WIDTH = 32;\n");
fprintf(fo, "ADDRESS_RADIX = HEX;\n");
fprintf(fo, "DATA_RADIX = HEX;\n");
fprintf(fo, "CONTENT\n");
fprintf(fo, "BEGIN\n\n");
}

/**
* @brief Prints the standard footer for an Intel MIF file.
*
* @param fo A file pointer to the output MIF file.
*/
void print_footer(FILE* fo){
fprintf(fo, "\nEND;\n");
}

int main(int argc, char* argv[]) {
   // The assembler requires three command-line arguments.
   if (argc != 4) {
       printf("Usage: ./dlx_asm <source_file>.dlx <data_file>.mif
<code_file>.mif\n");
       return 1;
   }

   char* source_file = argv[1];
   char* data_file = argv[2];
   char* code_file = argv[3];

   // === PASS 1: Find all labels in the .text segment ===
   // This is done first so that forward jumps and branches can be
resolved.
   int text_label_count = 0;
   label* text_labels = find_labels(source_file, &text_label_count);

   // Open the source file for reading.
   FILE* fp = fopen(source_file, "r");
   if(fp == NULL){
       printf("failed to open file %s\n", source_file);
       return 1;
```

```
    }

    // === DATA SEGMENT PROCESSING ===
    // Open the output data MIF file for writing.
    FILE* fo_data = fopen(data_file, "w");
    if(fo_data == NULL){
        printf("failed to open file %s\n", data_file);
        return 1;
    }

    int data_flag = 0;
    char line[256];

    // We will have a single list of labels for both data and text.
    label* all_labels = text_labels;
    int total_label_count = text_label_count;

    print_header(fo_data);

    int data_addr = 0; // Start data addresses at 0.

    // Go back to the beginning of the source file to parse the data
segment.
    rewind(fp);

    while (fgets(line, sizeof(line), fp)) {
        char buffer[256];
        strcpy(buffer, line);
        char *comment = strchr(buffer, ';');
        if (comment) *comment = '\0'; // Ignore comments.

        char *token = strtok(buffer, " ,\t\n\r");
        if (token == NULL) continue; // Skip empty lines.

        // Check for segment directives.
        if (strcmp(token, ".data") == 0) {
            data_flag = 1;
            continue;
        }
        if (strcmp(token, ".text") == 0) {
```

```
            // Once we hit the .text segment, we're done with data.
            break;
        }

        if (data_flag) {
            // Data format is: <label_name> <size> <value1> <value2> ...
            char* label_name = strdup(token);

            // Add this data label to our list of all labels.
            all_labels = (label*)realloc(all_labels, sizeof(label) *
(total_label_count + 1));
            all_labels[total_label_count].label = label_name;
            all_labels[total_label_count].address = data_addr;
            total_label_count++;

            // Parse the size of the data array.
            token = strtok(NULL, " ,\t\n\r");
            if (token == NULL) continue;
            int size = (int)strtol(token, NULL, 0);

            // Parse each initial value and write it to the data MIF file.
            for(int i = 0; i < size; i++) {
                token = strtok(NULL, " ,\t\n\r");
                long val = 0;
                if (token) val = strtol(token, NULL, 0);

                fprintf(fo_data, "%03X : %08X; --%s[%d]\n", data_addr++,
(unsigned int)val, label_name, i);
            }
        }
    }

    print_footer(fo_data);
    fclose(fo_data);

    // === PASS 2: CODE GENERATION ===
    // Open the output code MIF file for writing.
    FILE* fo_code = fopen(code_file, "w");
    if(fo_code == NULL){
        printf("failed to open file %s\n", code_file);
```

```
        return 1;
    }
    print_header(fo_code);

    // Go back to the beginning of the source file to parse the code
segment.
    rewind(fp);
    int code_flag = 0;
    int code_addr = 0;

    while (fgets(line, sizeof(line), fp)) {
        // Keep a copy of the original line to print as a comment in the
MIF file.
        char original_line[256];
        strcpy(original_line, line);
        if (original_line[strlen(original_line)-1] == '\n')
original_line[strlen(original_line)-1] = 0;

        char *comment = strchr(line, ';');
        if (comment) *comment = '\0';

        // Add '(' and ')' to delimiters to handle LW/SW syntax like
"offset(base)".
        char *token = strtok(line, " ,\t\n\r()");

        // Skip lines until we are in the .text segment.
        if (token == NULL) continue;
        if (strcmp(token, ".data") == 0) {
            code_flag = 0;
            continue;
        }
        if (strcmp(token, ".text") == 0) {
            code_flag = 1;
            continue;
        }
        if (!code_flag) continue;

        // Find the opcode for the current instruction.
        int opcode_idx = -1;
        for(int i = 0; i < 49; i++) {
```

```
            if(strcmp(token, opcodes[i].name) == 0) {
                opcode_idx = i;
                break;
            }
        }

        // If the first token is not an opcode, it must be a label.
        // In that case, the opcode is the *next* token.
        if (opcode_idx == -1) {
            token = strtok(NULL, " ,\t\n\r()");
            if (token == NULL) continue;

            for(int i = 0; i < 49; i++) {
                if(strcmp(token, opcodes[i].name) == 0) {
                    opcode_idx = i;
                    break;
                }
            }
        }

        // If we found a valid opcode, we can assemble the instruction.
        if (opcode_idx != -1) {
            int opcode_hex = opcodes[opcode_idx].hex;
            unsigned int instruction = 0;
            // The 6-bit opcode always goes in the most significant bits
(31-26).
            instruction |= (opcode_hex & 0x3F) << 26;

            const char* op_name = opcodes[opcode_idx].name;

            // --- INSTRUCTION FORMAT PARSING ---
            // The following blocks handle the different instruction
formats (I-Type, R-Type, J-Type, etc.).
            // Each block parses its specific operands and assembles the
32-bit instruction word
            // by shifting the operand values into their correct bit
positions.

            // Branch Instructions (e.g., BEQZ rs1, label)
```

```
            // Format: Op[31:26] | Rs1[25:21] | Unused[20:16] |
Immediate[15:0]
            if (strcmp(op_name, "BEQZ") == 0 || strcmp(op_name, "BNEZ") ==
0) {
                char* rs1_str = strtok(NULL, " ,\t\n\r()");
                char* label_str = strtok(NULL, " ,\t\n\r()");

                int rs1 = get_register(rs1_str);
                int addr = get_value(label_str, all_labels,
total_label_count);

                instruction |= (rs1 & 0x1F) << 21;
                instruction |= (addr & 0xFFFF);
            }
            // Jump Instructions (e.g., J label)
            // Format: Op[31:26] | Immediate[25:0]
            else if (strcmp(op_name, "J") == 0 || strcmp(op_name, "JAL") ==
0) {
                char* label_str = strtok(NULL, " ,\t\n\r()");
                int addr = get_value(label_str, all_labels,
total_label_count);
                instruction |= (addr & 0x3FFFFFF);
            }
            // Jump Register Instructions (e.g., JR rs1)
            // Format (deduced from examples): Op[31:26] | Unused[25:5] |
Rs1[4:0]
            else if (strcmp(op_name, "JR") == 0 || strcmp(op_name, "JALR")
== 0) {
                char* rs1_str = strtok(NULL, " ,\t\n\r()");
                int rs1 = get_register(rs1_str);
                instruction |= (rs1 & 0x1F);
            }
            // Load/Store Instructions (e.g., LW rd, offset(rs1))
            // Format: Op[31:26] | Rd[25:21] | Rs1[20:16] | Immediate[15:0]
            else if (strcmp(op_name, "LW") == 0) {
                char* rd_str = strtok(NULL, " ,\t\n\r()");
                char* off_str = strtok(NULL, " ,\t\n\r()");
                char* rs1_str = strtok(NULL, " ,\t\n\r()");

                int rd = get_register(rd_str);
```

```
                int rs1 = get_register(rs1_str);
                int off = get_value(off_str, all_labels,
total_label_count);

                instruction |= (rd & 0x1F) << 21;
                instruction |= (rs1 & 0x1F) << 16;
                instruction |= (off & 0xFFFF);
            }
            // Note: SW has a different operand order in the assembly
syntax.
            else if (strcmp(op_name, "SW") == 0) {
                // SW offset(rs1), rd
                char* off_str = strtok(NULL, " ,\t\n\r()");
                char* rs1_str = strtok(NULL, " ,\t\n\r()");
                char* rd_str = strtok(NULL, " ,\t\n\r()"); // This is the
source register.

                int off = get_value(off_str, all_labels,
total_label_count);
                int rs1 = get_register(rs1_str);
                int rd = get_register(rd_str);

                // The encoding is the same as other I-types, but 'rd'
holds the source register.
                instruction |= (rd & 0x1F) << 21;
                instruction |= (rs1 & 0x1F) << 16;
                instruction |= (off & 0xFFFF);
            }
            // Immediate Instructions (e.g., ADDI rd, rs1, imm)
            // Format: Op[31:26] | Rd[25:21] | Rs1[20:16] | Immediate[15:0]
            else if (op_name[strlen(op_name)-1] == 'I') {
                char* rd_str = strtok(NULL, " ,\t\n\r()");
                char* rs1_str = strtok(NULL, " ,\t\n\r()");
                char* imm_str = strtok(NULL, " ,\t\n\r()");

                int rd = get_register(rd_str);
                int rs1 = get_register(rs1_str);
                int imm = get_value(imm_str, all_labels,
total_label_count);
```

```
                instruction |= (rd & 0x1F) << 21;
                instruction |= (rs1 & 0x1F) << 16;
                instruction |= (imm & 0xFFFF);
            }
            // Register Instructions (e.g., ADD rd, rs1, rs2)
            // Format: Op[31:26] | Rd[25:21] | Rs1[20:16] | Rs2[15:11] |
Unused[10:0]
            else {
                char* rd_str = strtok(NULL, " ,\t\n\r()");
                char* rs1_str = strtok(NULL, " ,\t\n\r()");
                char* rs2_str = strtok(NULL, " ,\t\n\r()");

                int rd = get_register(rd_str);
                int rs1 = get_register(rs1_str);
                int rs2 = get_register(rs2_str);

                instruction |= (rd & 0x1F) << 21;
                instruction |= (rs1 & 0x1F) << 16;
                instruction |= (rs2 & 0x1F) << 11;
            }

            // Write the assembled instruction to the code MIF file.
            fprintf(fo_code, "%03X : %08X; --%s\n", code_addr++,
instruction,  original_line);
        }
    }

    print_footer(fo_code);

    // Clean up by closing files and freeing allocated memory.
    fclose(fo_code);
    fclose(fp);

    // Free the memory allocated for labels.
    for (int i = 0; i < total_label_count; i++) {
        free(all_labels[i].label);
    }
    free(all_labels);

    return 0;
```

```
}
```

find_lables.c

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

// This array defines all the DLX instructions that the assembler
recognizes.
// Each entry contains the instruction's mnemonic (e.g., "ADD") and its
corresponding 6-bit opcode in hexadecimal.
opcode opcodes[] = {
{"NOP", 0x00},
{"LW", 0x01},
{"SW", 0x02},
{"ADD", 0x03},
{"ADDI", 0x04},
{"ADDU", 0x05},
{"ADDUI", 0x06},
{"SUB", 0x07},
{"SUBI", 0x08},
{"SUBU", 0x09},
{"SUBUI", 0x0A},
{"AND", 0x0B},
{"ANDI", 0x0C},
{"OR", 0x0D},
{"ORI", 0x0E},
{"XOR", 0x0F},
{"XORI", 0x10},
{"SLL", 0x11},
{"SLLI", 0x12},
{"SRL", 0x13},
{"SRLI", 0x14},
{"SRA", 0x15},
{"SRAI", 0x16},
```

```
{"SLT", 0x17},
{"SLTI", 0x18},
{"SLTU", 0x19},
{"SLTUI", 0x1A},
{"SGT", 0x1B},
{"SGTI", 0x1C},
{"SGTU", 0x1D},
{"SGTUI", 0x1E},
{"SLE", 0x1F},
{"SLEI", 0x20},
{"SLEU", 0x21},
{"SLEUI", 0x22},
{"SGE", 0x23},
{"SGEI", 0x24},
{"SGEU", 0x25},
{"SGEUI", 0x26},
{"SEQ", 0x27},
{"SEQI", 0x28},
{"SNE", 0x29},
{"SNEI", 0x2A},
{"BEQZ", 0x2B},
{"BNEZ", 0x2C},
{"J", 0x2D},
{"JR", 0x2E},
{"JAL", 0x2F},
{"JALR", 0x30}
};

/**
* @brief This function performs the first pass of the assembler.
*
* Its primary purpose is to find all the labels within the .text (code)
segment
* of the DLX assembly file and record their addresses. These addresses are
crucial
* for resolving jumps and branches in the second pass.
*
* @param dlx_file The path to the input DLX assembly file.
* @param label_count A pointer to an integer that will be updated with the
number of labels found.
```

```
* @return A dynamically allocated array of `label` structs, each
containing a label name and its address.
*/
label* find_labels(char* dlx_file, int* label_count) {
   label* labels = NULL;
   *label_count = 0;

   FILE* fp = fopen(dlx_file, "r");
   if(fp == NULL){
       printf("failed to open file %s\n", dlx_file);
       exit(1);
   }

   char line[256];
   int in_text_segment = 0;

   // First, we need to find the beginning of the .text segment.
   // We'll read the file line by line until we find the ".text"
directive.
   while(fgets(line, sizeof(line), fp)) {
       // Remove comments from the line to avoid parsing them.
       char *comment = strchr(line, ';');
       if (comment) *comment = '\0';

       // Tokenize the line to find the .text directive.
       char* token = strtok(line, " ,\t\n\r");
       while(token != NULL) {
           if(strcmp(token, ".text") == 0) {
               in_text_segment = 1;
               break;
           }
           token = strtok(NULL, " ,\t\n\r");
       }
       if(in_text_segment) break; // Exit the loop once .text is found.
   }

   if (!in_text_segment) {
       printf("No .text segment found\n");
       // It's valid for a file to have no code, so we just return.
       fclose(fp);
```

```
        return NULL;
    }

    // Now that we are in the .text segment, we can start counting
addresses and finding labels.
    int addr = 0;
    while(fgets(line, sizeof(line), fp)) {
        char *comment = strchr(line, ';');
        if (comment) *comment = '\0'; // Truncate at comment

        char* token = strtok(line, " ,\t\n\r");
        if (token == NULL) continue; // Skip empty lines.

        // A line in the .text segment can contain a label, an instruction,
or both.
        // We process tokens from left to right.
        while (token != NULL) {
            int is_opcode = 0;
            // Check if the token is a recognized instruction mnemonic.
            for(int i = 0; i < 49; i++) {
                if(strcmp(token, opcodes[i].name) == 0) {
                    is_opcode = 1;
                    break;
                }
            }

            if (is_opcode) {
                // If we found an opcode, this line contains an
instruction.
                // We increment the address counter and stop parsing this
line,
                // as anything after the opcode is an operand.
                addr++;
                break;
            } else {
                // If the token is not an opcode, it must be a label.
                // We allocate memory for the new label and store its name
and current address.
                labels = (label*)realloc(labels, sizeof(label) *
(*label_count + 1));
```

```
                if (labels == NULL) {
                    printf("Memory allocation failed\n");
                    exit(1);
                }
                labels[*label_count].label = strdup(token);
                labels[*label_count].address = addr;
                (*label_count)++;
                // A label can be on the same line as an instruction, so we
continue to the next token.
            }
            token = strtok(NULL, " ,\t\n\r");
        }
    }

    fclose(fp);
    return labels; // Return the array of found labels.
}
```

structs.h

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

typedef struct{
    const char *name;
    int hex;
}opcode;

typedef struct{
    char *label;
    int address;
}label;

extern opcode opcodes[];

label* find_labels(char* dlx_file, int* label_count);
```

factorial.dlx

```
.data

; f stores result
f   1 0

; n stores original number
n   1 5

.text

setup:
    ; R5 = n
    LW   R5, n(R0)

    ; f = 1 (initialize factorial result)
    ADDI R4, R0, 1
    SW   R4, f(R0)

loop:
    ; if n == 1, done
    ADDI R7, R5, -1
    BEQZ R7, done

    ; call factorial multiply step
    JAL  factorial

    ; decrement n
    SUBI R5, R5, 1

    ; loop again
    J    loop


; ------------------------------------------------
; factorial:
;   f = f * n
;   Uses repeated addition
; ------------------------------------------------
```

```
factorial:
    ; load f into R6
    LW   R6, f(R0)

    ; R8 = loop counter = n
    ADD  R8, R0, R5

    ; R3 = accumulator (product)
    ADDI R3, R0, 0

multiply_loop:
    ; R3 += f
    ADD  R3, R3, R6

    ; decrement counter
    SUBI R8, R8, 1

    ; if counter != 0, loop
    BNEZ R8, multiply_loop

    ; store result back into f
    SW   R3, f(R0)

    ; return
    JR   R31


done:
    ; program end (halt or infinite loop)
    J done
```

factorial_code_passOff.mif

```
DEPTH = 1024;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
```

```
BEGIN

000 : 04A00001; --    LW   R5, n(R0)
001 : 10800001; --    ADDI R4, R0, 1
002 : 081F0000; --    SW   R4, f(R0)
003 : 10E5FFFF; --    ADDI R7, R5, -1
004 : ACE00000; --    BEQZ R7, done
005 : BC000000; --    JAL  factorial
006 : 20A50001; --    SUBI R5, R5, 1
007 : B4000000; --    J    loop
008 : 04C00000; --    LW   R6, f(R0)
009 : 0D002800; --    ADD  R8, R0, R5
00A : 10600000; --    ADDI R3, R0, 0
00B : 0C633000; --    ADD  R3, R3, R6
00C : 21080001; --    SUBI R8, R8, 1
00D : B1000000; --    BNEZ R8, multiply_loop
00E : 081F0000; --    SW   R3, f(R0)
00F : B800001F; --    JR   R31
010 : B4000000; --    J done

END;
```

factorial_data_passOff.mif

```
DEPTH = 1024;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN

000 : 00000000; --f[0]
001 : 00000005; --n[0]

END;
```