Lab 1

Jason Reynolds
Vinay Vazir
CPEG 324 Lab 1

Abstract:

The goal for this project was to develop a basic ISA and simulator to introduce skills that are necessary to complete the task. Over the course of this lab, the C programming language was used to successfully create an ISA simulator and compiler. In accomplishing this, a greater understanding of how ISA's function was achieved specifically with regards to the relationships between binary and assembly language.

Division of Labor:

The lab was broken down into two distinct components, the compiler and the simulator. The compiler would take in a text file, read it, remove comments, and then convert the assembly instructions into a binary code and write it to an output file. The simulator would take in this binary code from a file, strip out the instructions and decode them and then call the corresponding function associated with each opcode.

For the compiler, Vinay worked on reading in the correct file and parsing it for all the correct instructions and putting them in a list. Jason would then take these instructions and convert them to binary before returning them to Vinay, who would then write all the code out to a text file with the extension ".jv".

For the simulator, Vinay again worked on reading in the correct file and parsing it for all the binary code. Error checking is done here to make sure that the file is the correct length and does not contain invalid characters. Vinay then decoded the binary instructions into the corresponding opcodes. Jason created functions that corresponded with these opcodes that would take in the binary instruction and determine what to do with the rest of the binary code.

For both parts of the project, Vinay wrote the code that correspond with the verbose output to provided details on what is going on with the program. Jason worked on documentation for all the code and the ISA/usage of the code. All work was managed using GitHub for version control.

Strategy:

The first part of our strategy was to work out the ISA we were planning on using with pencil and paper. The next step we worked on was the compiler, as we decided that while we could write out all the test files with binary, having a compiler would allow us to very easily create complicated test cases faster and with more certainty than doing it by hand. (See Appendix II)

For the compiler, we decided to have the code split into two chunks. The first chunk was the main code, it would read in the assembly file and call a function that would encode each instruction and produce a binary string which was then written out to another text file. The second chunk of code was the actual encoder that took in a string and produced a binary string. The purpose of splitting the code this way was because by doing so we were able to easily split the workload between the two of us in a way that allowed both of us to work on the code without needing the others code until the very end, at which point we could just put the two chunks together. (See calcC.c for more details. The two main chunks are main() and encode())

For the simulator, again we split the code into chunks. The first chunk was again the main function that would read in a file and fetch all the instructions using the decode function. The other chunks were the individual instructions such as load, add, sub, display, and compare. Again, this was done because the workload could easily be split between us, with Vinay working on the main function and the decoder, and Jason working on the individual functions.

Results:

We were successfully able to produce a compiler and simulator for the ISA that we designed. It functions as specified and can be used to perform simple calculations, compare numbers, and display them.

Conclusion:

Our project went very well, as we had a lot of time to work on it and add extra features that made it more usable, such as the verbose output. The main weakness of the program is in the compiler, as it did very little error checking when compiling code. This was because the compiler was only an extra feature, not a required component of the lab, so we decided to spend more time on having the simulator be better able to reject invalid binary strings. If we had more time we would likely go back to the compiler and fix some of the issues it has and have it be able to properly compile code and give warning and error messages like a real compiler.

The area that gave us the most trouble was the fact that we worked with strings a lot during the project, and we had a lot of trouble with the null terminator. All of our char arrays that we thought would have worked without issue had to be extended by one to accommodate the null terminator. We also had the same issue during comparison statements, as if we did not include the null terminator the comparison failed.

Appendix I:

Name	Hours spent
Jason Reynolds	10
Vinay Vazir	12

Appendix II:
All code located here:
https://github.com/Syncla/CPEG324/tree/master/lab1
Compiler:

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

// Buffer to store filename
char filename[200];
// Buffer to store output filename
char outputFile[200];

// Buffer to store contents of input file
//char  lines[1000][100];

// Temp buffer to store binary conversion
char bin[9];
// Debug flag
int debug = 0;

// Input file handle
FILE * in;
// Output file handle
FILE * out;

// Arrays to hold stripped instructions
char* strpIns;
char* strpArgs[4];


//Function declerations
int fileLen(FILE * fp);
const char *getFilenameExt(const char *filename);
void getOutputFile(const char *filename,char* outputFilename);
void encode(char *args[],char[9]);
void stripInstruction(char** arg,char* inputStr);

void main(int argc, char *argv[]) {
      if (argc != 2 && argc !=3 ) {
            printf("Invalid number of arguments\n");
            printf("Proper usage:\n");
            printf("calcC <filename>\n [-v]");
```

```
            printf("Will output a file with the same filename and extension
jv\n");
            printf("add the flag -v for verbose output \n");
            return;
      }
      else {
            if (argc == 3) {
                  // Verbose output
                  if (!strcmp(argv[2], "-v")) {
                        debug = 1;
                  }
            }
            // Check for proper file extension
            if (strcmp("txt", getFilenameExt(argv[1]))){
                  printf("%s\n", argv[1]);
                  printf("Invalid file extension %s\n",
getFilenameExt(argv[1]));
                  printf("All files must have file extension of .txt\n");
                  return;
            }
            // Get the filename for the output file
            getOutputFile(argv[1], outputFile);
            // Open the input file for reading
            in = fopen(argv[1], "r");
            int len = fileLen(in);
            // Close the file so we can reopen it later to read it properly
            fclose(in);

            char lines[len][200];

            printf("Compiling %d lines\n", len);

            // Open input and output file
            in = fopen(argv[1], "r");
            out = fopen(outputFile, "w");
            for (int i = 0; i < len; i++) {

                  // Read a line from the input file
                  fgets(lines[i], sizeof(lines[0]), in);
                  // Strip the newline character
                  //strtok(lines[i], "\n");
```

```
            // If it is not an empty string
            if (strlen(lines[i]) > 2) {

                  // Debugging info
                  if (debug) {
                        printf("Line %3d : ", i + 1);
                  }
                  // Strip any comments from the code
                  for (int c = 0; c < sizeof(lines[i]); c++) {
                        if (lines[i][c] == '#'|| lines[i][c]=='\r') {
                              lines[i][c] = '\0';
                        }
                        if (lines[i][c] == '\t') {
                              lines[i][c] = ' ';
                        }
                  }
                  if (debug) {
                        printf("%s\t\t", lines[i]);
                        if (lines[i][0]=='d'||lines[i][0]=='c')
                              printf("\t");
                  }
                  // Parse string and seperate instructions
                  stripInstruction(strpArgs,lines[i]);
                  // More debugging info
                  // Convert to binary format
                  encode(strpArgs,bin);
                  // Write to output file
                  fprintf(out, bin,9);
                  // Uncomment this line for human read-ability
                  //fprintf(out, "\n");
                  // Print out encoded line
                  if (debug) {
                        //printf("\n");
                  }
            }
      }


      printf("Compiled successfully ");
      printf("output is located in %s\n", outputFile);
      // Close files
      fclose(out);
```

```
            fclose(in);
            return;
      }
      // The code should never reach here, but just incase, say something
went wrong
      printf("Error while compiling\n");
}
void encode(char *args[], char out[9]){

            //Set up variables
            char cmdi[4];     //command in
            char cmdo[2];     //command out

            char rg0i[3];     //destination register in
            char rg0o[3];     //destination register out

            char rg1i[3];     //source register in
            char rg1o[3];     //source register out

            char rg2i[3];     //target register in
            char rg2o[3];     //target register out

            char imm[5];      //immediate value
            strcpy(imm,"none"); //initialize to "none"

            char jmpi[2];     //jump amount in
            char jmpo[3];     //jump amount out

            char instructionOut[9];       //binary instruction out
            instructionOut[0] = '\0';     //start with null

            strcpy(cmdi, args[0]);        //pull instruction into command in
            cmdi[3] = '\0';

            strcpy(rg0i, args[1]);        //pull destination register into
destination register in
            rg0i[2] = '\0';

            //Identify the instruction, determine the binary code for it,
and store that in command out.
            if ((strcmp(cmdi, "lod") == 0)){
                  strcpy(cmdo,"00");
```

```
            }//if

            else if((strcmp(cmdi, "add") == 0)){
                  strcpy(cmdo,"01");
            }//else if

            else if((strcmp(cmdi, "sub") == 0)){
                  strcpy(cmdo,"10");
            }//else if

            else if((strcmp(cmdi, "dsp") == 0)){
                  strcpy(cmdo,"11");
            }//else if

            else if((strcmp(cmdi, "cmp") == 0)){
                  strcpy(cmdo,"11");
            }//else if

            //Identify destination register and determine the binary for it
            if((strcmp(rg0i,"r0") == 0)){
                  strcpy(rg0o,"00");
            }//if

            else if((strcmp(rg0i,"r1") == 0)){
                  strcpy(rg0o,"01");
            }//if

            else if((strcmp(rg0i,"r2") == 0)){
                  strcpy(rg0o,"10");
            }//if

            else if((strcmp(rg0i,"r3") == 0)){
                  strcpy(rg0o,"11");
            }//if


            //From here on out we are decoding/encoding the instruction
based on which one it is.

            //From here on out we are decoding/encoding the instruction
based on which one it is.
```

```
            //From here on out we are decoding/encoding the instruction
based on which one it is.


            //Starting with the load immediate.
            if ((strcmp(cmdi,"lod") == 0)){                //copy the
binary immediate from the input string to the immediate variable.
                  strcpy(imm, args[2]);
                  imm[5] = '\0';

                  strcat(instructionOut,cmdo);        //cat the command out
binary to the instruction out.
                  strcat(instructionOut,rg0o);        //cat the destination
register to the instruction out.
                  strcat(instructionOut,imm);              //cat the
immediate value to the instruction out.

                  instructionOut[8] = '\0';
            }//if load case

            //Now we handle the add and sub instructions.
            else if ((strcmp(cmdi, "add") == 0) || (strcmp(cmdi, "sub") ==
0)){  //add or sub

                  strcpy(rg1i, args[2]);        //pull source register info
from input string to source register in.
                  rg1i[2] = '\0';

                  strcpy(rg2i, args[3]);        //pull target register info
from input string to source register in.
                  rg2i[2] = '\0';

                  //Identify source register and store binary
representation in source register out.
                  if((strcmp(rg1i,"r0") == 0)){
                        strcpy(rg1o,"00");
                  }//if

                  else if((strcmp(rg1i,"r1") == 0)){
                        strcpy(rg1o,"01");
```

```
                }//if

                else if((strcmp(rg1i,"r2") == 0)){
                      strcpy(rg1o,"10");
                }//if

                else if((strcmp(rg1i,"r3") == 0)){
                      strcpy(rg1o,"11");
                }//if

                //Identify target register and store binary
representation in target register out.

                if((strcmp(rg2i,"r0") == 0)){
                      strcpy(rg2o,"00");
                }//if

                else if((strcmp(rg2i,"r1") == 0)){
                      strcpy(rg2o,"01");
                }//else if

                else if((strcmp(rg2i,"r2") == 0)){
                      strcpy(rg2o,"10");
                }//else if

                else if((strcmp(rg2i,"r3") == 0)){
                      strcpy(rg2o,"11");
                }//else if

                strcat(instructionOut,cmdo);
                strcat(instructionOut,rg0o);
                strcat(instructionOut,rg1o);
                strcat(instructionOut,rg2o);

          }//else if add or sub

          else if ((strcmp(cmdi, "cmp") == 0)){     //cmp
                strcpy(rg1i, args[2]);
                rg1i[2] = '\0';

                //read in 2nd register
```

```
            if((strcmp(rg1i,"r0") == 0)){
                  strcpy(rg1o,"00");
            }//if

            else if((strcmp(rg1i,"r1") == 0)){
                  strcpy(rg1o,"01");
            }//if

            else if((strcmp(rg1i,"r2") == 0)){
                  strcpy(rg1o,"10");
            }//if

            else if((strcmp(rg1i,"r3") == 0)){
                  strcpy(rg1o,"11");
            }//if


            strcpy(jmpi, args[3]);
            jmpi[1] = '\0';

            //checking for number of lines to skip
            if ((strcmp(jmpi,"1") == 0)){
                  strcpy(jmpo,"01");
            }//if

            else if((strcmp(jmpi,"2") == 0)){
                  strcpy(jmpo,"10");
            }//else if

            strcat(instructionOut,cmdo);
            strcat(instructionOut,rg0o);
            strcat(instructionOut,rg1o);
            strcat(instructionOut,jmpo);

      }//else if cmp


      else if ((strcmp(cmdi, "dsp") == 0)){     //dsp
            strcat(instructionOut,cmdo);
            strcat(instructionOut,rg0o);
            strcat(instructionOut,"0000");
```

```
            }//else if dsp

            /*
            printf("Command in: %s\n",cmdi);
            printf("Register in: %s\n\n", rg0i);

            printf("Command out: %s\n",cmdo);
            printf("Register out: %s\n\n", rg0o);

            printf("imm is: %s\n\n", imm);
            */
            for (int i =0;i<9;i++){
                  out[i]=instructionOut[i];
            }
            if (debug){
                  printf("Final binary is: %s\n",instructionOut);
            }

}//decode()


void stripInstruction(char** arg, char* inputStr){
              char * instr;
              int index = 0;

              instr = strtok (inputStr," ,.-");

              while (instr != NULL){
                  arg[index] = instr;
                  instr = strtok (NULL, " ,.-");
                  index++;
              }//while

              while (index < 4){
                        arg[index] = " ";
                        index++;
              }//while

}//strip
```

```
const char *getFilenameExt(const char *filename) {
	/*
		This function consumes a file name and extracts the file
extension
		parameters:
			filename - a char array containing the filename
		returns:
			ext - a char array containing the file extension
	*/
	const char *ext = strrchr(filename, '.');
	// If there is no file extension, return an empty string
	if (!ext || ext == filename) {
		return "";
	}
	// Return the file exension
	return ext + 1;
}
void getOutputFile(const char *filename,char *outputFilename) {
	/*
		This function consumes a filename and produces a new filename
with the extension
		changed to .jv
		parameters:
			filename - a char array containing the filename
			outputFilename - the filename with the changed extension
		returns:
	*/
	// If it is an empty filename, exit
	if (strlen(filename) == 0) {
		return;
	}
	// Find the last file extension, and replace it with .jv
	for (int i = strlen(filename)-1; i >=0; i--) {
		if (filename[i] == '.') {
			for (int c = 0; c <= i; c++) {
				outputFilename[c] = filename[c];
			}
			outputFilename[i + 1] = 'j';
			outputFilename[i + 2] = 'v';
			break;
		}
	}
```

```
	return;
}


int fileLen(FILE * fp) {
	/*
		Given a file fp, find the number of lines in the file
		Parameters:
			fp - a file handle
		returns:
			lines - number of lines in the file
	*/
	int lines = 0;
	char ch;
	// If there is an invalid file handle, return 0
	if (fp == NULL) {
		return 0;
	}
	lines++;
	while (!feof(fp))
	{
		ch = fgetc(fp);
		if (ch == '\n')
		{
			lines++;
		}
	}
	return lines;
}
```

Simulator:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
// Data type
struct Ins {
	char op[4];
	char r1[3];
	char r2[3];
	char r3[3];
	char imm[5];
```

```
      char jmp[2];
};

// Input file handle
FILE * in;

// Debug flag
int debug = 0;

//Function declerations
int fileLen(char* filename);
const char *getFilenameExt(const char *filename);
int readFile(char* filename, char [][8], int len);
int decode(char[][8], struct Ins[],int len);
int getReg(char[2]);

void lod(struct Ins ins);
void add(struct Ins ins);
void sub(struct Ins ins);
void dsp(struct Ins ins);
int ipow(int base, int exp);
int cmp(struct Ins ins);


#define KNRM  "\x1B[0m"
#define KRED  "\x1B[31m"
#define KGRN  "\x1B[32m"
#define KYEL  "\x1B[33m"
#define KBLU  "\x1B[34m"
#define KMAG  "\x1B[35m"
#define KCYN  "\x1B[36m"
#define KWHT  "\x1B[37m"

#define eof '\0'
int8_t r0=0;
int8_t r1=0;
int8_t r2=0;
int8_t r3=0;
int pc=0;


void main(int argc, char *argv[]) {
```

```
	if (argc != 2 && argc !=3 ) {

		printf("%sInvalid number of arguments%s\n",KRED,KWHT);
		printf("Use the [-h] argument for help on proper usage\n");
		return;
	}
	else {
		if (argc == 2) {
			if (!strcmp(argv[1], "-h")) {
				printf("Proper usage:\n");
				printf("calc <filename> [-v]\n");
				printf("Will read in a binary file and currently do
nothing, use the verbose command for cool stuff\n");
				printf("add the flag -v for verbose output \n");
				return;
			}
		}
		if (argc == 3) {
			// Verbose output
			if (!strcmp(argv[2], "-v")) {
				debug = 1;
			}
			else if (!strcmp(argv[2], "-h")) {
				printf("Proper usage:\n");
				printf("calc <filename> [-v]\n");
				printf("Will read in a binary file and currently do
nothing, use the verbose command for cool stuff\n");
				printf("add the flag -v for verbose output \n");
				return;
			}
		}
		// Check for valid file name
		char * filename = argv[1];
		if (debug){
			printf("Opening file: %s\n",filename);
		}
		in = fopen(filename,"r");
		if (in==NULL){
			printf("%sInvalid file name %s%s\n",KRED,filename,KNRM);
			return;
		}
		fclose(in);
```

```
            // Check for valid extension
            const char * ext = getFilenameExt(filename);
            if (strcmp(ext,"jv")){
                  printf("%sInvalid file extension %s%s\n",KRED,ext,KNRM);
                  return;
            }
            // Get number of instructions
            int instructions = fileLen(filename);
            if (instructions<0){
                  printf("%sError please use the -v option for verbose
logging%s\n",KRED,KNRM);
                  return;
            }
            if (debug){
                  printf("There are %s%d%s instructions in the file
%s%s%s\n",KGRN,instructions,KWHT,KCYN,filename, KNRM);
            }
            char bitList[instructions][8];
            struct Ins insList[instructions];
            // Get instructions
            if (debug){
                  printf("Reading instructions from file\n\n");
            }
            if (readFile(filename, bitList, instructions))
                  return;
            // Decode instructions
            if (decode(bitList,insList,instructions)){
                  printf("Error invalid instructions in file\n");
                  printf("Read error log for what caused the error\n");
                  printf("Run the calc program with the -v option for
verbose logging\n");

printf("-----------------------------------------------\n");
            }
            // If debugging, print out instructions read
            if (debug){
                  printf("line#:\t\t|\t:Binary:\t|\top  r1 r2 r3 imm
extra\n");

printf("---------------+----------------------+-------------------------
-\n");
                  for (int i = 0;i<instructions;i++){
```

```
                        printf("ins %s%3d%s:\t|\t",KGRN,i, KNRM);
                        for(int c=0;c<8;c++){
                              printf("%s%c%s",KGRN,bitList[i][c], KNRM);
                        }
                        printf("\t|\t%s%s%s",KMAG,insList[i].op, KNRM);
                        printf(" %s%s", KYEL,insList[i].r1);
                        printf(" %s", insList[i].r2);
                        printf(" %s%s", insList[i].r3, KNRM);
                        printf(" %s%s", KBLU,insList[i].imm);
                        printf(" %s%s", KNRM,insList[i].jmp);


                        printf("\n");
                  }
                  printf("\n==============Now running
instructions==============%s\n\n",KNRM);
            }
            int nextPC = 1;
            while (pc < instructions) {
                  nextPC = 1;
                  if (debug) {
                        printf("PC:%s%3d%s ins: %s%s%s ", KGRN,pc,
KNRM,KMAG,insList[pc].op, KNRM);
                        printf(" %s%s",KYEL, insList[pc].r1);
                        printf(" %s", insList[pc].r2);
                        printf(" %s%s", insList[pc].r3,KBLU);
                        printf(" %s%s", insList[pc].imm, KNRM);
                        printf(" %s", insList[pc].jmp);
                        printf(" | REG: %sr0%s=%s%3d%s | %sr1%s=%s%3d%s |
%sr2%s=%s%3d%s | %sr3%s=%s%3d%s\n",KYEL,KWHT,KGRN,r0,KWHT, KYEL, KWHT,
KGRN, r1, KWHT, KYEL, KWHT, KGRN, r2, KWHT, KYEL, KWHT, KGRN, r3, KNRM);
                  }
                  if (!strcmp(insList[pc].op, "lod\0")) {
                        lod(insList[pc]);
                  }
                  else if (!strcmp(insList[pc].op, "add\0")) {
                        add(insList[pc]);
                  }
                  else if (!strcmp(insList[pc].op, "sub\0")) {
                        sub(insList[pc]);
                  }
                  else if (strcmp(insList[pc].op, "dsp\0")==0) {
```

```
                        dsp(insList[pc]);
                }
                else if (!strcmp(insList[pc].op, "cmp\0")) {
                        int jump = cmp(insList[pc]);
                        if (debug)
                                printf("PC += %d\n", jump+1);
                        nextPC += jump;
                }
                else {
                        printf("%sUnrecognized command %s on line %d
,terminating program%s\n",KRED,insList[pc].op,pc, KNRM);
                        return;
                }
                pc+=nextPC;
        }
    }
}
int decode(char bitList[][8],struct Ins insList[],int len){

    const char space[2] = " \0";
    for (int ins=0;ins<len;ins++){
        char opCode[3] = "  \0";
        char op[4] = "   \0";
        char reg1[3] = "  \0";
        char reg2[3] = "  \0";
        char reg3[3] = "  \0";
        char imm[5] = "    \0";
        char extra[3] = "  \0";
        char jmp[2] = "0\0";
        strncpy(jmp," ",1);
        opCode[0] = bitList[ins][0];
        opCode[1] = bitList[ins][1];

        reg1[0] = bitList[ins][2];
        reg1[1] = bitList[ins][3];

        reg2[0] = bitList[ins][4];
        reg2[1] = bitList[ins][5];

        reg3[0] = bitList[ins][6];
        reg3[1] = bitList[ins][7];
```

```
            imm[0] = bitList[ins][4];
            imm[1] = bitList[ins][5];
            imm[2] = bitList[ins][6];
            imm[3] = bitList[ins][7];

            extra[0] = bitList[ins][6];
            extra[1] = bitList[ins][7];

            if (getReg(reg1)||getReg(reg2)||getReg(reg3)){
                        return 1;
            }
            if (!strcmp(opCode,"00")){
                  strncpy(op,"lod",3);

            }
            else if (!strcmp(opCode,"01")){
                  strncpy(op,"add",3);
            }
            else if (!strcmp(opCode,"10")){
                  strncpy(op,"sub",3);
            }
            else if (!strcmp(opCode,"11")){
                  if (!strcmp(extra,"00")){
                        strncpy(op,"dsp",3);
                        strncpy(jmp, "0", 1);
                  }
                  else if (!strcmp(extra,"01")){
                        strncpy(op,"cmp",3);
                        strncpy(jmp,"1",1);
                  }
                  else if (!strcmp(extra,"10")){
                        strncpy(op,"cmp",3);
                        strncpy(jmp,"2",1);
                  }
                  else {
                        printf("%sInvalid Command found %s with extra flag
%s%s\n", KRED, opCode,extra, KWHT);
                        return -1;
                  }
            }
            strncpy(insList[ins].op,op,4);
            strncpy(insList[ins].r1, reg1, 3);
```

```
            strncpy(insList[ins].r2, reg2, 3);
            strncpy(insList[ins].r3, reg3, 3);
            strncpy(insList[ins].imm, imm, 5);
            strncpy(insList[ins].jmp,jmp, 2);
            //strncpy(insList[ins],out,16);
      }
      return 0;
}
int getReg(char reg[2]){
            if (!strcmp(reg,"00")){
                  strncpy(reg,"r0",2);
            }
            else if (!strcmp(reg,"01")){
                  strncpy(reg,"r1",2);
            }
            else if (!strcmp(reg,"10")){
                  strncpy(reg,"r2",2);
            }
            else if (!strcmp(reg,"11")){
                  strncpy(reg,"r3",2);
            }
            else{
                  printf("Invalid reg number %c%c\n",reg[0],reg[1]);
                  return 1;
            }
            return 0;
}
int readFile(char* filename, char bitList[][8],int len){
      FILE * fp = fopen(filename,"r");
      char bit;
      for (int ins = 0;ins<len;ins++){
            for (int pos = 0;pos<8;pos++){
                  bit = fgetc(fp);
                  if (bit != '\0')
                  {
                        bitList[ins][pos]=bit;
                  }
                  if (!(bit == '0' || bit == '1') && bit != '\0') {
                        printf("%sInvalid character found in binary file:
%c%s\n", KRED, bit, KNRM);
                        return -1;
                  }
```

```
            }
      }
      fclose(fp);
      return 0 ;
}

int fileLen(char* filename) {
      /*
            Given a file, find the length of the file in 8 bit segments
            Parameters:
                  char* - a file name
            returns:
                  ins - number of instructions in the file
      */
      int ins = 0;
      char ch;
      // If there is an invalid file handle, return 0
      FILE * fp = fopen(filename, "r");
      if (fp == NULL) {
            return 0;
      }
      int c = 0;
      while (!feof(fp))
      {
            ch = fgetc(fp);
            if (ch != '\0')
            {
                  c++;
            }
            if (c==8){
                  ins++;
                  c=0;
            }
            if (!(ch == '0' || ch == '1') && !feof(fp)) {
                  printf("%sInvalid character found in binary file:
%c%s\n", KRED, ch, KNRM);
                  return -2;
            }
      }
      if (c!=1){
            printf("%sMissing binary characters, should have 8 digits,
instead you have %d on line:%d%s\n",KRED,c-1,ins+1,KWHT);
```

```
            return -1;
      }
      fclose(fp);
      return ins;
}

const char *getFilenameExt(const char *filename) {
      /*
            This function consumes a file name and extracts the file
extension
            parameters:
                  filename - a char array containing the filename
            returns:
                  ext - a char array containing the file extension
      */
      const char *ext = strrchr(filename, '.');
      // If there is no file extension, return an empty string
      if (!ext || ext == filename) {
            return "";
      }
      // Return the file exension
      return ext + 1;
}

void binary(int8_t x,char bin[9]) {
      bin[8] = '\0';
      if (x < 0) {
            bin[0] = '1';
            x = x * -1;
      }
      else {
            bin[0] = '0';
      }
      int i = 7;
      while (i>0) {
            //printf("%d\n",i);
            bin[i] = (x % 2) + '0';
            x = x / 2;
            i--;
      }
      if (bin[0] == '1') {
            for (i = 1; i < 8; i++) {
```

```
                  bin[i] = (bin[i] == '0') ? '1' : '0';
            }
            int carry = 0;
            if (bin[7] == '0') {
                  bin[7] = '1';
                  return;
            }
            else {
                  bin[7] = '0';
                  carry = 1;
            }
            for (i = 6; i > 0; i--) {
                  if (carry) {
                        if (bin[i] == '0') {
                              bin[i] = '1';
                              return;
                        }
                        else {
                              bin[i] = '0';
                        }
                  }
            }
      }
}

void lod(struct Ins ins) {
      int immInt = 0;
      int isNeg = 0;
      int carry = 0;

      //check for negative
      if (ins.imm[0] == '1') {
            isNeg = 1;
      }//if

       //two's compliment time
      if (isNeg == 1) {

            //First we flip
            for (int i = 3; i >= 0; i--) {
                  if (ins.imm[i] == '1') {
                        ins.imm[i] = '0';
```

```
                }//if

                else if (ins.imm[i] == '0') {
                        ins.imm[i] = '1';
                }//else

        }//for

         //now add one
         //LSB == 0 case first
        if (ins.imm[3] == '0') {
                ins.imm[3] = '1';
        }//if

         //Now if the LSB == 1
        else if (ins.imm[3] == '1') {
                ins.imm[3] = '0';
                carry = 1;
                int i = 2;

                //changing the bits as long as the carry == 1
                while ((carry == 1) && (i >= 0)) {

                        if (ins.imm[i] == '1') {
                                ins.imm[i] = '0';
                                carry = 1;
                        }//if

                        else if (ins.imm[i] == '0') {
                                ins.imm[i] = '1';
                                carry = 0;

                        }//else if
                        i--;

                }//while

        }//else if

}//if

 //converting to decimal
```

```
	for (int i = 3; i >0; i--) {

		if (ins.imm[i] == '1') {
			immInt += ipow(2, 3 - i);
		}//if

	}//for

	 //handling MSB and negativity
	if (isNeg == 1) {
		if (strcmp(ins.imm, "1000") == 0) {
			immInt = 8;
		}//if
		immInt = 0 - immInt;
	}//if


	 //storing in appropriate register
	if (strcmp(ins.r1, "r0") == 0) {
		r0 = immInt;
	}//if r0

	else if (strcmp(ins.r1, "r1") == 0) {
		r1 = immInt;
	}//else if r1

	else if (strcmp(ins.r1, "r2") == 0) {
		r2 = immInt;
	}//else if r2

	else if (strcmp(ins.r1, "r3") == 0) {
		r3 = immInt;
	}//else if r3

}//lod
void add(struct Ins ins) {
	int reg2;
	int reg3;
	int sum;

	if (strcmp(ins.r2, "r0") == 0) {
		reg2 = r0;
```

```
	}//if r0

	else if (strcmp(ins.r2, "r1") == 0) {
		reg2 = r1;
	}//else if r1

	else if (strcmp(ins.r2, "r2") == 0) {
		reg2 = r2;
	}//else if r2

	else if (strcmp(ins.r2, "r3") == 0) {
		reg2 = r3;
	}//else if r3

	if (strcmp(ins.r3, "r0") == 0) {
		reg3 = r0;
	}//if r0

	else if (strcmp(ins.r3, "r1") == 0) {
		reg3 = r1;
	}//else if r1

	else if (strcmp(ins.r3, "r2") == 0) {
		reg3 = r2;
	}//else if r2

	else if (strcmp(ins.r3, "r3") == 0) {
		reg3 = r3;
	}//else if r3

	sum = reg2 + reg3;

	if (strcmp(ins.r1, "r0") == 0) {
		r0 = sum;
	}//if r0

	else if (strcmp(ins.r1, "r1") == 0) {
		r1 = sum;
	}//else if r1

	else if (strcmp(ins.r1, "r2") == 0) {
		r2 = sum;
```

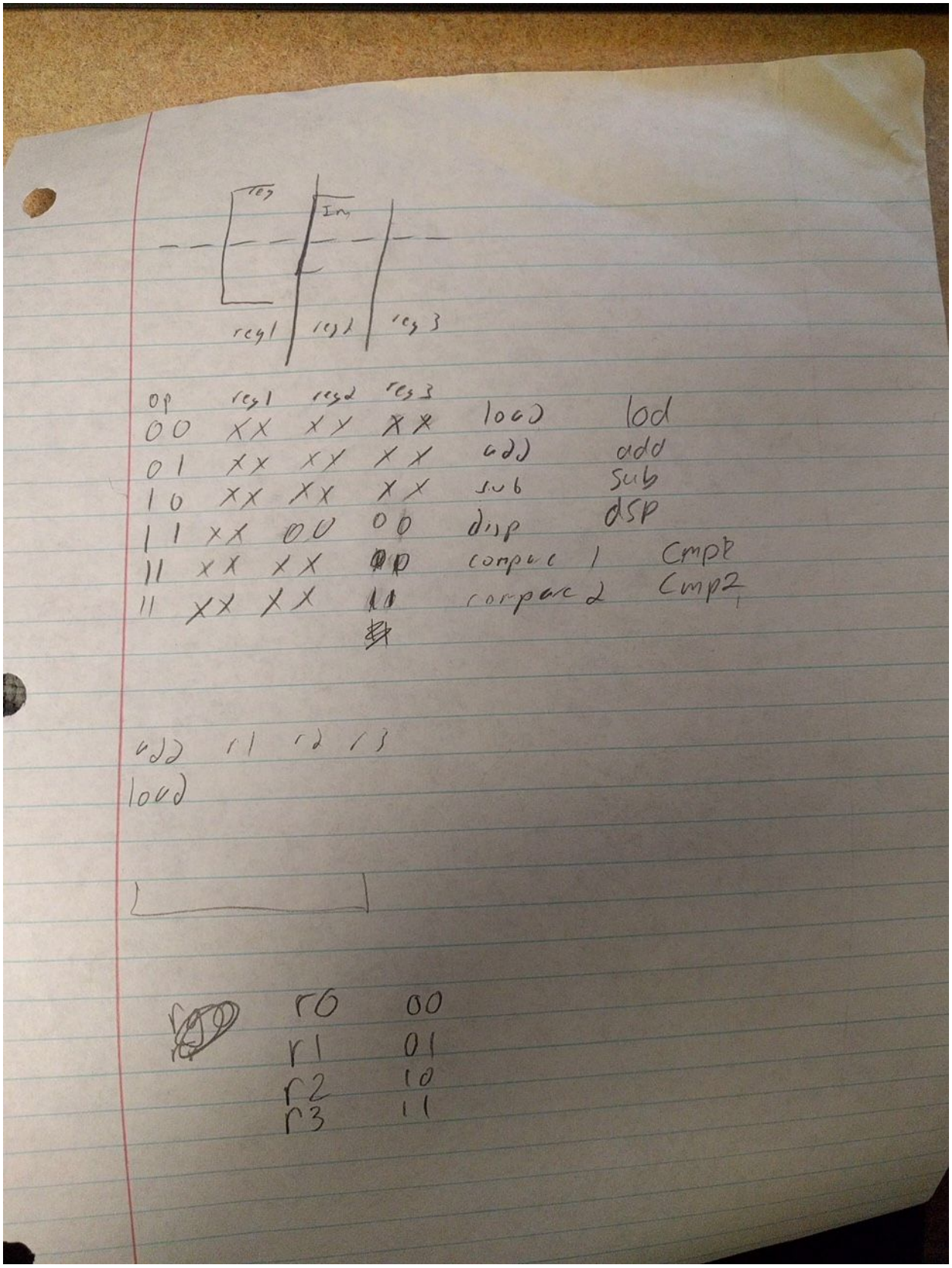
```
	}//else if r2

	else if (strcmp(ins.r1, "r3") == 0) {
		r3 = sum;
	}//else if r3


}//add
void sub(struct Ins ins) {
	int reg2;
	int reg3;
	int dif;

	if (strcmp(ins.r2, "r0") == 0) {
		reg2 = r0;
	}//if r0

	else if (strcmp(ins.r2, "r1") == 0) {
		reg2 = r1;
	}//else if r1

	else if (strcmp(ins.r2, "r2") == 0) {
		reg2 = r2;
	}//else if r2

	else if (strcmp(ins.r2, "r3") == 0) {
		reg2 = r3;
	}//else if r3

	if (strcmp(ins.r3, "r0") == 0) {
		reg3 = r0;
	}//if r0

	else if (strcmp(ins.r3, "r1") == 0) {
		reg3 = r1;
	}//else if r1

	else if (strcmp(ins.r3, "r2") == 0) {
		reg3 = r2;
	}//else if r2

	else if (strcmp(ins.r3, "r3") == 0) {
```

```
            reg3 = r3;
      }//else if r3

      dif = reg2 - reg3;

      if (strcmp(ins.r1, "r0") == 0) {
            r0 = dif;
      }//if r0

      else if (strcmp(ins.r1, "r1") == 0) {
            r1 = dif;
      }//else if r1

      else if (strcmp(ins.r1, "r2") == 0) {
            r2 = dif;
      }//else if r2

      else if (strcmp(ins.r1, "r3") == 0) {
            r3 = dif;
      }//else if r3


}//dif
void dsp(struct Ins ins) {
      char bin[9];
      if (strcmp(ins.r1, "r0") == 0) {
            binary(r0, bin);
            printf("%4d : %s\n", r0,bin);
      }//if r0

      else if (strcmp(ins.r1, "r1") == 0) {
            binary(r1, bin);
            printf("%4d : %s\n", r1,bin);
      }//else if r1

      else if (strcmp(ins.r1, "r2") == 0) {
            binary(r2, bin);
            printf("%4d : %s\n", r2, bin);
      }//else if r2

      else if (strcmp(ins.r1, "r3") == 0) {
            binary(r3, bin);
```

```
            printf("%4d : %s\n", r3, bin);
      }//else if r3
}//dsp
int cmp(struct Ins ins) {
      int reg1;
      int reg2;
      int jump;

      if (!strcmp(ins.jmp, "1")) {
            jump = 1;
      }
      else if (!strcmp(ins.jmp, "2")) {
            jump = 2;
      }//else if

       //set reg1 val
      if (strcmp(ins.r1, "r0") == 0) {
            reg1 = r0;
      }//if r0

      else if (strcmp(ins.r1, "r1") == 0) {
            reg1 = r1;
      }//else if r1

      else if (strcmp(ins.r1, "r2") == 0) {
            reg1 = r2;
      }//else if r2

      else if (strcmp(ins.r1, "r3") == 0) {
            reg1 = r3;
      }//else if r3

       //set reg2 val
      if (strcmp(ins.r2, "r0") == 0) {
            reg2 = r0;
      }//if r0

      else if (strcmp(ins.r2, "r1") == 0) {
            reg2 = r1;
      }//else if r1

      else if (strcmp(ins.r2, "r2") == 0) {
```

```
            reg2 = r2;
      }//else if r2

      else if (strcmp(ins.r2, "r3") == 0) {
            reg2 = r3;
      }//else if r3

       //comparison time
      if (reg1 != reg2) {
            return 0;
      }//if

      else if (reg1 == reg2) {
            return jump;
      }//if

}//cmp
int ipow(int base, int exp) {
      int result = 1;
      while (exp) {
            if (exp & 1)
                  result *= base;
            exp >>= 1;
            base *= base;
      }//while

      return result;
}//ipow
```

Notes:


reg
Im
reg1
reg2
reg3
op reg1 reg2 reg3
00 XX XX XX load lod
01 XX XX XX add add
10 XX XX XX sub sub
11 XX 00 00 disp dsp
11 XX XX 01 compare 1 Cmp1
11 XX XX 10 compare 2 Cmp2
add r1 r2 r3
load
r0 00
r1 01
r2 10
r3 11

Appendix III:
All test files can be found here:
https://github.com/Syncla/CPEG324/tree/master/lab1
The .txt files are the assembly code files, while the .jv files are the binary strings