MaskinnærProjekt

Martin Mårtensson
Daniel Styrbæk Petersen

May 11, 2020

1 Requirements

The goal of this project was to construct an assembler for a LC3's processor. A program that should be able to take a file of LC3 instructions, and convert each instruction to machine code.

Meaning that each combonent in the instruction must be translated into its machine code part (binary representation). Every instruction is made of four parts, a label, an opcode, its operands and evt. comments.
Of these four parts only the opcode and the operands are mandatory to make a succesful instruction. The opcode is the part of the instruction that tells the computer what operations to perform, while the operands are the things these operations are to be performed on.
The label part of the instruction is an optional symbolic name for an instruction. A label tells the assembler to remember the memory address of this instruction, and save it in a symbol table. This way labels can be used as repeating references to the same instruction or part of the memory.
Comments are optional parts of an instruction, that makes it more easily readable by humans reading the code.

In this project the specific requirements for the assembler was to implement 6 functions of lc3 aswell as supporting the pseudo-ops (.ORG, .BLKW, .FILL, .STRINGZ and .END) and label functionality.

2 Analysis

LC3 is a little computer which takes only 16 bit of binary instructions, so it is necessary to compile the Assembly code in a way that is compatible with this. LC3 has 8 registers lables R0 R.. R7

The Assembly language has 2 different types of "commands", instructions and directives.

Instructions are commands like ADD which can add a number to another number and store it in a register, or NOT which negates a value from a register and stores it in another register. All instructions has a binary representation (opcode) which consists of 4 bytes. Some instructions takes a destination register (DR), and one or more source registers (SR) as arguments. Some include immediate values (imm) (number values) and some include PCoffsets (links to other places in the code).

SR and DR is represented by 3 bits.
imm is represented by 5 bits
PCoffset is represented by a variable lenght either 9 or 6 bits.
BaseR is the base register and is represented by 3 bits.
nzp is conditions used for branching.
n is negative value.
z is zero.
p is positive.

In the project, the following insctuctions are included:

- **ADD**
  Adds a value to another and stores it in a destination register.
  OPCODE: 0001
  ARGS: DR, SR1, (SR2 eller imm)

- **NOT**
  Store the negated value of a source register in a destination register.
  OPCODE: 1001
  ARGS: DR, SR

- **BR(nzp)**
  Jumps to given offset address. if the given (nzp) is true.
  OPCODE: 0000
  ARGS: n, p, z, PCoffset

- **LD**
  Loads the value that is stored in the offset (memory) into the register.
  OPCODE: 0010
  ARGS: DR, PCOffset

- **LDR**
  Loads the value of the offset(memory) plus the base register into the destination or source register.
  OPCODE: 0110
  ARGS: DR, BaseR, offset.

- **ST**
  Stores the value of the register into the offset (memory).
  OPCODE: 0110
  ARGS: SR, PCoffset.

3 Design

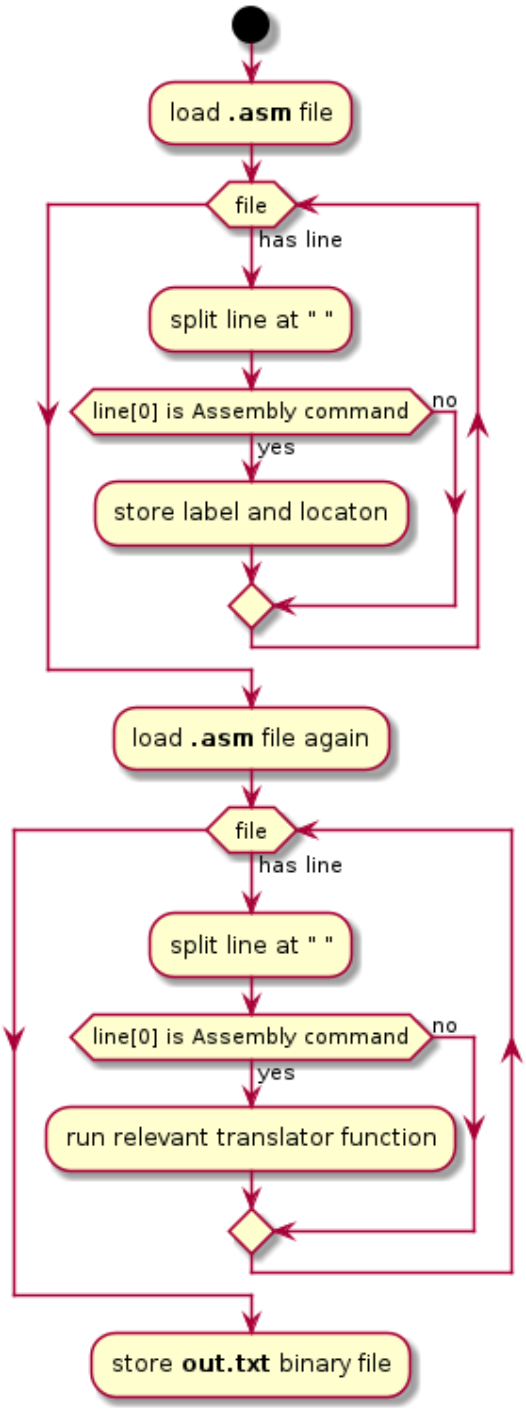

Figure 1: Overall flow of the compiler

In the figure above you see an top down view of how the whole compiler runs. First it reads the

inputfile looking for lables only. If it finds a label then it saves the label, the location and the binary code of the label to the file sym.txt for later usage. after it reads the same file again and now just to compile the code and writing it to the out.txt file. The figure abobe is an example of the use of the BR

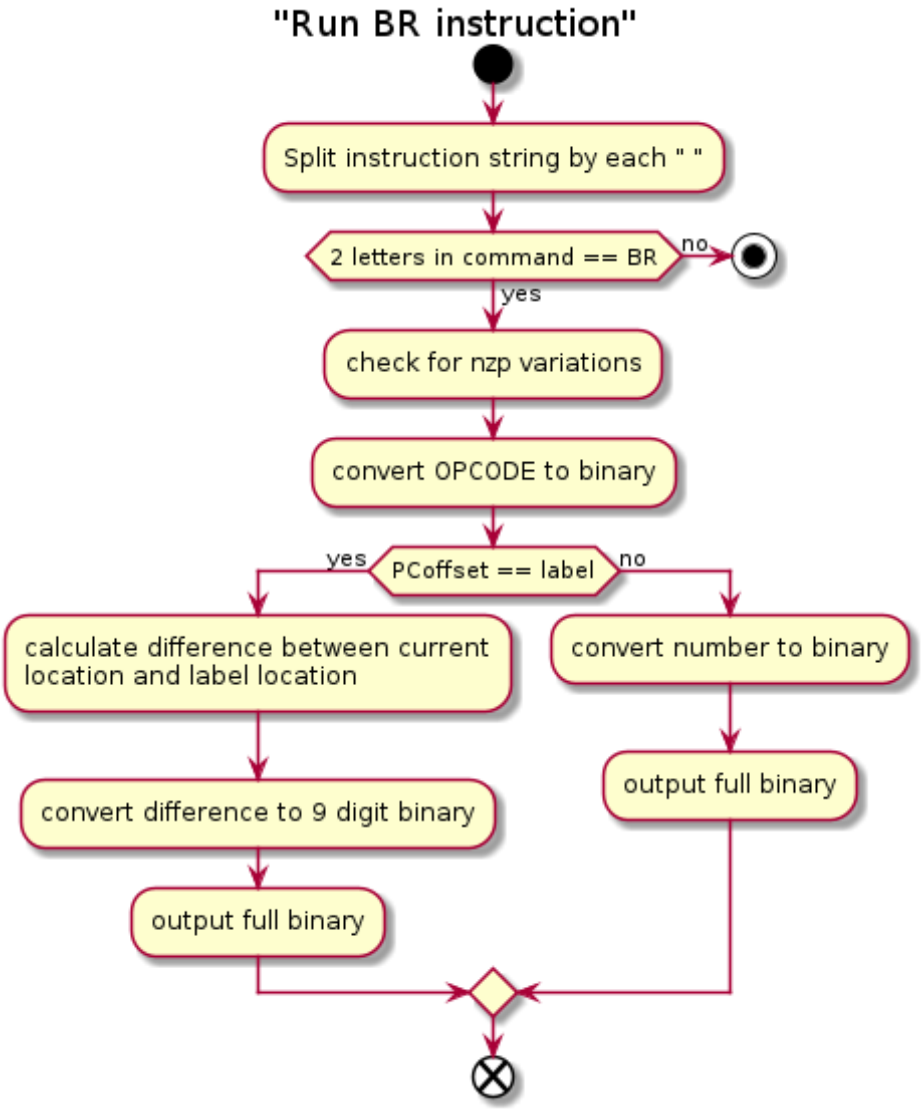

Figure 2: Example of using the BR instruction

instruction. First the string that is read from the input file is divided into each word, then it checks if the first 2 letters in the first word in the new array is BR then if it aint it will end but if it is then it will check for nzp values and convert the OPCODE based on this. There after it will check what the argument sent to the BR is. First it will check if it is a label, if it aint then it will see it as a number, if it is then it will get the location of the label and calculate the diff between the current location and the label location. thereafer no matter what it will write the fill binary code to the file out.txt

3.1 File structure

The root directory has a main.c file and 3 folders

- main.c
  This file includes everything in the project.
- /out
  here output files are stored.
  - out.txt
    all the binary code.
  - sym.txt
    the labels are saved here.
- /instructions
  here is a file for each LC3 instruction the program should be able to compile.
- /directives
  Here is a file for each LC3 directives

4 Implementation

The project structure is divided into files and folders. The root directory consists of a main.c file which includes all the headers used throughout the project.

The project includes 3 public headers from the C library. These are

- string.h
- stdlib.h
- stdio.h

```
#include <string.h>
...
int main(int argc, char *argv[])
{
    ...
    f = fopen(obj, "r");
    int i = 12288;
    while (fgets(str, MAX, f))
    {
        char *res = labelfinder(str, i);
        i++;
    }

    f = fopen(obj, "r");
    i = 12288;
    while (fgets(str, MAX, f))
    {
        char *res = split(str, i);
        i++;
    }
    return 0;
}
```

The main function takes the path of an input file as a command line argument, it then runs labelfinder() on each line of the file. i is the location of the line in the file. this location will be converted into hexidecimal and saved together with the label if a label is found. After that it calls split() on each line in the input file here it maked sure to send the location again which is used to compare with lables if a label is used later in on of the instructions.

4.1 LABELS

In order to implement the conversion of labels into binary, a function called labelfinder was implemented. As seen below (some comments were kept for clarity):

```
char *labelfinder(char str[], int line)
{

    char *p = strtok(str, " ");
    char *array[10];
    int i = 0;
    while (p != NULL)
```

```
{
    array[i++] = p;
    p = strtok(NULL, " ");
}

char *functions[30] = {
    "ADD",
    "NOT",
    "BR",
    "BRn",
    "BRz",
    "BRp",
    "BRnz",
    "BRnp",
    "BRzp",
    "BRnzp",
    "ST",
    "LD",
    "LDR",
    ".ORIG",
    ".FILL",
    ".STRINGZ",
    ".BLKW",
    ".END",
};

for (i = 0; i < 18; i++)
{
    if (strcmp(array[0], functions[i]) == 0)
    {
        return "no lable";
    }
}
int j;

char *newray[10];
for (j = 0; j < 10; j++)
{
    newray[j] = array[j + 1];
}
/*The label name is stored for reference in the symbol table */
char *key = array[0];
/*The value of the instruction following the label is converted to binary */
char *value = router(newray, line);
/*The value "line" derived from our i integer in main() is converted to hex and stor
char x[10] = "0x";
char hex[10];
sprintf(hex, "%x", line);
strcat(x, hex);
/*The table is written using the symbols function in write.c using key as "labelname
and value as binary representation of whatever instruction followed the label */
symbols(key, x, value);
```

```
    free(value);
}
```

The labelfinder function is called before the primary function in the programming that outputs the machine code to the out.txt file. The purpose of this function is to read through the entire file, and find any symbols that are not regonized by the program and denote them as labels. Because we assume correct syntax, aswell as knowledge about the limitations of the system. We can also assume that an uknown string must mean a user defined label.

The labelfinder takes a string array of elements that are generated by " " delimiting the line read from the lc3 file. It then compares the first argument of this array, with the list of accebtable opcodes and pseudo ops. If there is no match, it then does the following:
First it assigns the value of array[0] as the labelname, in this program denoted as "key". Then it takes the second arguement it received the int "line", and converts this into hexidecimal, and stores that value as address. In this program denoted as x. Finally it makes a new array, consisting of all the elements from the line in the file that was read, without the label itself. Then the values of this line are converted into binary and stored, the symbol table as value.

The symbol table generated from this function is then used, when the rest of the program translates the lc3 file into machine code. By comparing the current address of the line in the program being run, with the stored address of the label. Fx in the instruction BR LOOP.

4.2 Select implementation of a few methods

In the following section a few implementation of a select few instructions will explained in more detail.

4.2.1 ADD

The add instruction takes the value of two registers and adds their content together and stores the new value in a third register. Alternatively, it takes the value stored in a register and adds with an immediate value that stores the total in a third register. Below is the way the group has chosen to implement the ADD operation.

```
char *add(char *array[])
{
        char *binary = "";
        binary = (char *)malloc(100);
        char *end = strstr(array[3], "#");
        int i = 0;
        if (end)
        {
                for (i = 0; i < 3; ++i)
                {
                        char *ret = converter(array[i]);
                        strcat(binary, ret);
                }
                strcat(binary, "1");
                char *res = holger(array[3], 1);
                strcat(binary, res);
                return binary
        }
```

```
        else
        {
                for (i = 0; i < 3; ++i)
                {
                        char *ret = converter(array[i]);
                        strcat(binary, ret);
                }
                strcat(binary, "000");
                char *retur = converter(array[3]);
                strcat(binary, retur);
                return binary;
        }
}
```

Using

```
ADD R0, R1 #-7
```

as an example instruction. The add function takes an array as its argument, this array consists of a line read from the lc3 assembly file split into chunks delimitated by a whitespace character.

So in this case it would be { "ADD", "R0", "R1", "#-7" }. Then we create the variable binary and allocate 100 bits of memory to it. This is done so a pointer to the variable can be returned from the function and used in other parts of the program.

We then check if the fourth element of the array contains a # character. This is done because the ADD instruction has an opcode and 3 operands. The first two operands are registers, while the third can either be another registry or an immediate value, designated by a # character.

So if in this example we find that the third operand is a number, we then translate the opcode "ADD" using the converter function, which gives us "0001", then we translate operand one and two giving us "000" and "001" respectively. This means that bits [15:6] are taken care of. For bit [5] we add a 1 to denote that the third operand is a value gained from sign extending the last five bits stored in [4:0].
After the 1 is added to bit [5], we call the function to translate -7 into signed binary, but only giving us the first 5 bits of the binary. In this case -7 turns into 11001. The function is now complete and will return "0001000001111001" as the machine code for the ADD instruction given as an example.

4.2.2 BR

The BR function is lc3 branch function, which takes two operands, the first operand is the conditions that need to be made in order for the branch to execute, the three conditions are n for negative, z for zero and p for positive. If the condition are met the BR instruction will then load the memory address, gained from sign extending the second operand and adding that to the PC increment.

```
char *add(char *array[])
{
        char *binary = "";
        binary = (char *)malloc(100);
        char *end = strstr(array[3], "#");
        int i = 0;
        if (end)
        {
```

```
                for (i = 0; i < 3; ++i)
                {
                        char *ret = converter(array[i]);
                        strcat(binary, ret);
                }
                strcat(binary, "1");
                char *res = holger(array[3], 1);
                strcat(binary, res);
                return binary
        }
        else
        {
                for (i = 0; i < 3; ++i)
                {
                        char *ret = converter(array[i]);
                        strcat(binary, ret);
                }
                strcat(binary, "000");
                char *retur = converter(array[3]);
                strcat(binary, retur);
                return binary;
        }
}
```

Using

```
BRn LOOP
```

as an example, first like in our add function the br function takes an array of " " delimeted strings as an argument, unklike add br also takes an int called line as an argument, line equates to the current line in the lc3 file that program is executing.
First we allocate memory to the binary variable that will be put into the out.txt file. Then we call the convert function on the first argument of the string. In this case that would be the opcode "BRn" which would get translated to machine code as "0000100" the "0000" part signifying BR, and the "100" part signifying that we are looking for a negative number and not a zero or a positve number.

Next the program calls the function "lbl2addr" using the second argument in the array, which scans for a match between whatever is in array[1] and the generated symbol table.
If it does not find a match, that means that the PCoffset has been hardcoded by the programmer who wrote the lc3 file. And the value of the PCoffset is calculated by generating the binary representation of the offset.

In our case tho we do have a label, since this is the case that means that lbl2addr has returned the address of whatever label in the symbol table matched the second operand in our instruction.
The program then takes this address and compares it with where the PC counter is currently in the program. Lets say in the example the label LOOP has the address 0x3001, and that our branch instruction has the address 0x3007. These values are then converted into decimal numbers 12289 and 12295. These valuea are then subtracted from eachother, in this case this would give -6. That value is then converted into binary, and appended as the PCoffset. In this case -6 = 1111111111111010. Of which the first 9 bites are used to denote the offset.

5 Tests

The tests for this project were primarily done by ad hoc testing the code, as neither member of the group had any real experience with a C based testing framework, like the library cunit for unit testing in C.
As such most testing was done by making use of the C functionalities "printf()" and "puts()" which output values to the console. These outputs were then compared to the the groups knowledge of assembly and machine code, to see whether the functions behaved as intended.
Now this has limited the knowledge about the extend to which the program performs as intended. Some desirable tests to have implemented would have been regression tests, which is the practice of continuously testing both functional and non-functional code, to see if new additions to the code breaks previously implemented code.

As for crossplatform testing, the program has only been tested on the operating system linux, as this projects deal with low level programming and machine code. Differences could very well occur when the program is run on a windows or mac system.

6 Conclusion

During this project the group has acquired a more in dept knowledge of the relationship between machine code and the C programming language. The group achieved its goal of successfully translating LC3 instructions into machine given a file as user input. The overall structure of the program was implemented very well, but some solutions in the code reflects on the general inexperience of the group when handling the programming language C.

A solution that worked well, was the way the group handled the program "counter" which read what line of the input file the program was currently handling. But this could also have been implemented in a smarter way, as it stands the program counter "hardcodes" itself to start at memory location 12288 or 0x3000. While this makes no difference in the way this program was implemented, since the program only cares about the difference between start and current position. A future version of this project would ideally have been able to read whichever address that followed the ".ORIG" pseudo-op. As it stands, the program also cannot handle memory locations that are not stored in relation to the programs "beginning".