Project Report

On

SIC/XE Assembler (Program Blocks)

CSN 252

System Software

Under

Prof. Manoj Misra

Name: Nayan Ratan Kakade

Enrollment No: 22114060

Batch: O3

Date: 11/04/2024

SIC/XE Assembler

I have implemented 2 pass SIC/XE Assembler supporting Program Block in C++ language. This Assembler converts assembly code written in SIC/XE into object program.

Input File (Pass 1)

1. **input.txt**: Contains the assembly code in SIC/XE language.

Output File (Pass 1)

1. **intermediate.txt**: Contains useful information about instruction, its corresponding program block.

2. **symtab.txt**: Symbol Table - Contains mapping of labels, literals and their location counter, program block, and its relative/absolute status.

3. **error_pass1.txt**: Contains errors and location of errors that are found while performing pass1.

Input File (Pass 2)

1. **intermediate.txt**: Generated from pass 1.
2. **symtab.txt**: Generated from pass 1.

Output File (Pass 2)

1. **listingFile.txt**: Contains instruction, its line number, memory address after the assembly, program block, and assembled object code of every instruction. It also contains comments.

2. **objectFile.txt**: Contains output generated by the assembler after successfully assembling the source code. It also contains the machine code instructions and data, organized in a format that can be directly loaded and executed by the target machine (in this case, a SIC/XE computer).
   It is organized in the following manner.
   a) Header Record: Contains information about name of program, starting address, and program length.
   b) Text Record: Contains actual object code generated by the actual program. It also contains the starting address of the current text record and its length.
   c) Modification Record: Contains information about the instructions that require modifications at specific locations.
   d) End Record: Contains information about the address of the first executable instruction.

3. **error_pass2.txt**: Contains errors and location of errors that are found while performing pass 2.

CPP code files

1. **main.cpp**: It calls two functions pass1() and pass2() present in pass1.cpp and pass2.cpp respectively.

2. **pass1.cpp**: Contains the necessary code for performing pass 1 of the assembly.

3. **pass2.cpp:** Contains the necessary code for performing pass 2 of the assembly.

4. **pass1.hpp**: Contains the shared variable (program length) between both pass1.cpp and pass2.cpp.

5. **optab.cpp**: Contains the opcode mapping and register mapping.

Supporting Features

1. Literals
2. Symbol Defining Statements
3. Expressions
4. Program Block

Working

Pass 1

In pass 1 of the assembly, I have traversed line by line of input.txt.

For each line, if it is a comment, then added to intermediate.txt file. Else, I have parsed the line using break_line() function and extracted label, opcode and operand from it.

If opcode is START, I have assigned starting address and initialized location counter of default block and then added the line to intermediate.txt file.

If opcode is USE, then I have checked for name of block. If it is new block then initialize new location counter for that block, else continue the location counter used previously for that block.

If operand[0] is '=', it means it is literal, then it is inserted into literal tab.

If opcode is LTORG, then I have inserted all literals from the literal tab after the current line, and emptied the literal tab.

If opcode is EQU, and if operand is *, then I have assigned location counter to the label and inserted into symbol table, else if operand is expression, then evaluate the expression and insert the evaluated value in the symbol table. I have also inserted the status of the label whether it is absolute or relative.

Then perform opcode search using the search_optab() function. If it is found then depending on the instruction type, location counter is added. If it is not found then I have checked for other opcode type such as RESB, RESW, WORD, BYTE. If not found in any of these, the error is reported.

Pass 2

Firstly, the contents of symtab.txt containing the symbol table is read and stored in the map named symbolTable in the format:- symbolTable[symbol]={rel,{value,block_number}};

In pass 2 of the assembly, I have traversed line by line of intermediate.txt.

For each line, if it is a comment, then added to intermediate.txt file. Else, I have parsed the line using break_intermediateline() function and extracted program_block,label, opcode and operand from it.

If opcode is START, then I have initialized location counter and starting address for default block and initialized the header record.

If opcode is BASE, then I have initialized the base value from the symbol table and base block number.

If opcode is RESB/RESW, then location counter is increased as per size of operand.

If opcode is of format 1, object code is initialized by accessing value of the opcode from optab.

If opcode is of format 2, then depending on whether there are 1 or 2 operands in the operand field, the object code is initialized.

If opcode is of format 3, then difference is calculated between the current program counter and location counter of the symbol. Also the starting address of their respective block is added to their counter. If absolute value of difference is less than 2048, then pc relative addressing is used. Else if base is initialized then base relative addressing is used, Else error is reported.

If opcode is of format 4, then e value is set to 1. Using the address of symbol, the object code is initialized.

For each operand, the search is performed by removing the first character if it is @ or #, and corresponding n and i value is set.

If opcode is BYTE, then object code is initialized using the ASCII values of the operand.

If opcode is WORD, then object code is initialized using the operand valxue directly.

For each object code, it is added to the text record if enough space is left. Maximum space allowed for one text record is 30 bytes. Else if space not left, new text record initialized. Also if opcode is USE, new text record is initialized.

If opcode is END, then all the modification records are inserted and finally the end record is inserted in the objectFile.

Input File

```
COPY START 0
FIRST STL RETADR
CLOOP JSUB RDREC
 LDA LENGTH
 COMP #0
 JEQ ENDFIL
 JSUB WRREC
 J CLOOP
ENDFIL LDA =C'EOF'
 STA BUFFER
 LDA #3
 STA LENGTH
 JSUB WRREC
 J @RETADR
 USE CDATA
RETADR RESW 1
LENGTH RESW 1
 USE CBLKS
BUFFER RESB 4096
BUFFEND EQU *
MAXLEN EQU BUFFEND-BUFFER
.
. READ RECORD FROM BUFFER
.
 USE
RDREC CLEAR X
 CLEAR A
 CLEAR S
 +LDT #MAXLEN
RLOOP TD INPUT
 JEQ RLOOP
 RD INPUT
 COMPR A,S
 JEQ EXIT
```

```
 STCH BUFFER,X
 TIXR T
 JLT RLOOP
EXIT STX LENGTH
 RSUB
 USE CDATA
INPUT BYTE X'F1'
.
. WRITE RECORD INTO BUFFER
.
 USE
WRREC CLEAR X
 LDT LENGTH
WLOOP TD =X'05'
 JEQ WLOOP
 LDCH BUFFER,X
 WD =X'05'
 TIXR T
 JLT WLOOP
 RSUB
 USE CDATA
 LTORG
 END FIRST
```

How to Compile and Run

Name of text file: input.txt

Insert single space between two fields. Example(CLOOP JSUB RDREC)

If starting field empty, then insert single space at the start.

Example - `CLOOP JSUB RDREC` `LDA LENGTH` instruction with two and three fields.

Run the main.cpp file in the following manner

```
PS C:\Users\dell\Desktop\Assembler> g++ main.cpp -o a
PS C:\Users\dell\Desktop\Assembler> ./a
Source code input is taken from input.txt file in the same folder

Performing Pass 1 on the input source code

Inside pass1 function

Pass1 is performed successfully and intermediate file is generated in the same folder
```

Folder Structure

Name	Date modified	Type	Size
.vscode	09-04-2024 18:11	File folder	
a	11-04-2024 23:20	Application	607 KB
error_pass1	11-04-2024 23:20	Text Document	0 KB
error_pass2	11-04-2024 23:20	Text Document	0 KB
input	11-04-2024 18:23	Text Document	1 KB
intermediate	11-04-2024 23:20	Text Document	1 KB
listingFile	11-04-2024 23:20	Text Document	3 KB
main	11-04-2024 18:05	C++ Source File	1 KB
main	11-04-2024 23:15	Application	607 KB
objectFile	11-04-2024 23:20	Text Document	1 KB
optab	10-04-2024 00:21	C++ Source File	3 KB
pass1	11-04-2024 22:07	C++ Source File	14 KB
pass1	10-04-2024 00:21	C++ Header Sourc...	1 KB
pass2	11-04-2024 23:10	C++ Source File	28 KB
symtab	11-04-2024 23:20	Text Document	1 KB

Code Snippets

```
0000 1 COPY   START  0
0000 1 FIRST  STL    RETADR     172063
0003 1 CLOOP  JSUB   RDREC      4B2021
0006 1        LDA    LENGTH     032060
0009 1        COMP   #0         290000
000C 1        JEQ    ENDFIL     332006
000F 1        JSUB   WRREC      4B203B
0012 1        J      CLOOP      3F2FEE
0015 1 ENDFIL LDA    =C'EOF'    032055
0018 1        STA    BUFFER     0F2056
001B 1        LDA    #3         010003
001E 1        STA    LENGTH     0F2048
0021 1        JSUB   WRREC      4B2029
0024 1        J      @RETADR    3E203F
0000 2        USE    CDATA
0000 2 RETADR RESW   1
0003 2 LENGTH RESW   1
0000 3        USE    CBLKS
0000 3 BUFFER RESB   4096
1000 3 BUFFENDEQU    *
1000   MAXLEN EQU    BUFFEND-BUFFER
       .
       . READ RECORD FROM BUFFER
       .
0027 1        USE
0027 1 RDREC  CLEAR  X          B410
0029 1        CLEAR  A          B400
002B 1        CLEAR  S          B440
002D 1        +LDT   #MAXLEN    75100000
0031 1 RLOOP  TD     INPUT      E32038
0034 1        JEQ    RLOOP      332FFA
0037 1        RD     INPUT      DB2032
003A 1        COMPR  A,S        A004
003C 1        JEQ    EXIT       332008
003F 1        STCH   BUFFER,X   57A02F
```

Listing File

```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE^032055^0F2056^010003^
T^00001E^09^0F2048^4B2029^3E203F^
T^000027^1D^B410^B400^B440^75100000^E32038^332FFA^DB2032^A004^332008^57A02F^B850^
T^000044^09^3B2FEA^13201F^4F0000^
T^00006C^01^F1^
T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850^3B2FEF^4F0000^
T^00006D^04^454F46^05^
E^000000
```

Object File

```
SYMTAB
---------------------
=C'EOF' R 7 2
=X'05' R 10 2
BUFFEND R 4096 3
BUFFER R 0 3
CLOOP R 3 1
ENDFIL R 21 1
EXIT R 71 1
FIRST R 0 1
INPUT R 6 2
LENGTH R 3 2
MAXLEN A 4096 -1
RDREC R 39 1
RETADR R 0 2
RLOOP R 49 1
WLOOP R 82 1
WRREC R 77 1
```

Symbol Table

```
1 COPY START 0
1 FIRST STL RETADR
1 CLOOP JSUB RDREC
1  LDA LENGTH
1  COMP #0
1  JEQ ENDFIL
1  JSUB WRREC
1  J CLOOP
1 ENDFIL LDA =C'EOF'
1  STA BUFFER
1  LDA #3
1  STA LENGTH
1  JSUB WRREC
1  J @RETADR
2  USE CDATA
2 RETADR RESW 1
2 LENGTH RESW 1
3  USE CBLKS
3 BUFFER RESB 4096
3 BUFFEND EQU *
- MAXLEN EQU BUFFEND-BUFFER
```

Intermediate file

```
map<string, pair<char, pair<int, int>>> read_symtab_pass2_file()
{
    map<string, pair<char, pair<int, int>>> symbolTable;
    ifstream symtab_pass2File("symtab.txt");
    // Check if the symtab_pass2 file is opened successfully
    if (!symtab_pass2File)
    {
        cout << "Error: Unable to open the symtab_pass2 file!" << endl;
    }
    string line;
    while (getline(symtab_pass2File, line))
    {
        istringstream iss(line);
        string symbol;
        char rel;
        int value;
        int blk_no;

        // Read symbol and value from the line
        if (!(iss >> symbol >> rel >> value >> blk_no))
        {
            // cout << "Error: Invalid input line: " << line << endl;
            continue;
        }

        // Insert into the map
        symbolTable[symbol] = {rel, {value, blk_no}};
    }
    return symbolTable;
}
```

Read Symbol Table

```
vector<string> break_intermediateline(string line)
{
    vector<string> parse_line;
    // cout<<"Line: "<<line<<endl;
    // cout<<"LIne[0]:"<<line[0]<<":"<<endl;
    if (line[0] == ' ')
        parse_line.push_back(" ");

    int i;
    if (line[0] == ' ')
        i = 1;
    else
        i = 0;

    string part = "";
    while (i < line.length())
    {
        if (line[i] == ' ')
        {
            parse_line.push_back(part);
            part = "";
        }
        else
            part += (line[i]);
        i++;
    }
    parse_line.push_back(part);

    return parse_line;
}
```

Break line into program block, label, opcode, operand

```
string intTo6Hex(int value)
{
    if (value < 0)
    {
        value = (1 << 24) + value;
    }

    std::stringstream stream;
    stream << uppercase << hex << setw(6) << setfill('0') << value;
    return stream.str();
```

Convert decimal to hexadecimal

```
if (opcode == "START")
{
    prog_blk_num_pass2["default"] = 1;
    string header_record = "H^";
    header_record += (pad_string(label) + "^");
    header_record += (intTo6Hex(stoi(operand)) + "^");
    starting_address = stoi(operand);
    loc_ctr_pass2 = stoi(operand);
    header_record += intTo6Hex(program_length);
    objectProgram.push_back(header_record);
    list_line.replace(0, 4, intToHex(loc_ctr_pass2, 4));
    list_line.replace(5, 1, to_string(prog_blk_num[block_name]));
    list_line.replace(7, label.size(), label);
    list_line.replace(14, opcode.length(), opcode);
    list_line.replace(21, operand.length(), operand);
    listingFile << list_line << endl;
    continue;
}

if (opcode == "END")
{
    int text_rec_length = 60 - left;
    string text_rec_len_str = intToHex(text_rec_length / 2, 2);
    text_record[9] = text_rec_len_str[0];
    text_record[10] = text_rec_len_str[1];
    objectProgram.push_back(text_record);

    list_line.replace(14, opcode.length(), opcode);
    list_line.replace(21, operand.length(), operand);
    listingFile << list_line << endl;

    string end_record = "E^";
    end_record += intTo6Hex((symtab[operand].second.first));
    // cout<<end_record<<endl;
    objectProgram.push_back(end_record);
```

Handling start and end

```
            int symbol_address = symtab_pass2[operand].second.first + prog_blk_start_addr[symtab_pass2[operand].second.second
            char symbol_type = symtab_pass2[operand].first;
            // cout<<"sym addr: "<<symbol_address<<endl;
            pc = loc_ctr_pass2 + 3;
            // cout<<"pc: "<<pc<<endl;
            if (symbol_type == 'R')
            {
                int difference = symbol_address - (pc + prog_blk_start_addr[prog_blk_num[block_name]]);
                // cout<<symbol_address<<(pc+prog_blk_start_addr[prog_blk_num[block_name]])<<endl;
                if (difference >= -2048 && difference < 2047)
                {
                    xbpe[2] = '1';
                }
                else if(difference<=4095 && difference>=0)
                {
                    if(symtab_pass2.find("BASE")==symtab_pass2.end()){
                        errorFile<<"Error: Assembler directive BASE not found for handling large displacement"<<endl;
                    }
                    xbpe[1] = '1';
                    difference = symbol_address - (base + prog_blk_start_addr[base_blk]);
                }
                else{
                    errorFile<<"Error: For relative addressing using BASE, the displacement is out of bounds"<<endl;
                }

                string hex_xbpe = binary_to_hex(xbpe);

                obj_code = intToHex(hexStringToInt(optab_pass2[opcode].second) + ni, 2) + hex_xbpe + intToHex(difference, 3);
            }
            else
            {
                string hex_xbpe = binary_to_hex(xbpe);
                obj_code = intToHex(hexStringToInt(optab_pass2[opcode].second) + ni, 2) + hex_xbpe + intToHex(symbol_address,
            }
        }
```

Handling format 3 instruction

```
// cout<<left<<" "<<obj_code<<endl;
if (obj_code.length() > left)
{

    int text_rec_length = 60 - left;
    string text_rec_len_str = intToHex(text_rec_length / 2, 2);
    text_record[9] = text_rec_len_str[0];
    text_record[10] = text_rec_len_str[1];
    objectProgram.push_back(text_record);
    if (search == "1")
    {
        text_record = begin_text_record(loc_ctr_pass2 - 1 + prog_blk_start_addr[prog_blk_num_pass2[block_name]]);
        // text_record=begin_text_record(loc_ctr_pass2-1);
    }
    else if (search == "2")
    {
        text_record = begin_text_record(loc_ctr_pass2 - 2 + prog_blk_start_addr[prog_blk_num_pass2[block_name]]);
        // text_record=begin_text_record(loc_ctr_pass2-2);
    }
    else
    {
        if (opcode[0] == '+')
        {
            text_record = begin_text_record(loc_ctr_pass2 - 4 + prog_blk_start_addr[prog_blk_num_pass2[block_name]]);
            // text_record=begin_text_record(loc_ctr_pass2-4);
        }
        else
        {
            text_record = begin_text_record(loc_ctr_pass2 - 3 + prog_blk_start_addr[prog_blk_num_pass2[block_name]]);
            // text_record=begin_text_record(loc_ctr_pass2-3);
        }
    }

    text_record += (obj_code + "^");
    left = 60 - obj_code.length();
}
else
{
    if (left == 60)
        text_record = begin_text_record(prev_loc_ctr + prog_blk_start_addr[prog_blk_num_pass2[block_name]]);
    text_record += (obj_code + "^");
    left -= obj_code.length();
}
```

Inserting object code in text record

```
map<string, pair<string, string>> opcode_map;
opcode_map["ADD"] = {"3/4", "18"};
opcode_map["ADDF"] = {"3/4", "58"};
opcode_map["ADDR"] = {"2", "90"};
opcode_map["AND"] = {"3/4", "40"};
opcode_map["CLEAR"] = {"2", "B4"};
opcode_map["COMP"] = {"3/4", "28"};
opcode_map["COMPF"] = {"3/4", "88"};
opcode_map["COMPR"] = {"2", "A0"};
opcode_map["DIV"] = {"3/4", "24"};
opcode_map["DIVF"] = {"3/4", "64"};
opcode_map["DIVR"] = {"2", "9C"};
opcode_map["FIX"] = {"1", "C4"};
opcode_map["FLOAT"] = {"1", "C0"};
opcode_map["HIO"] = {"1", "F4"};
opcode_map["J"] = {"3/4", "3C"};
opcode_map["JEQ"] = {"3/4", "30"};
opcode_map["JGT"] = {"3/4", "34"};
opcode_map["JLT"] = {"3/4", "38"};
opcode_map["JSUB"] = {"3/4", "48"};
opcode_map["LDA"] = {"3/4", "00"};
opcode_map["LDB"] = {"3/4", "68"};
opcode_map["LDCH"] = {"3/4", "50"};
opcode_map["LDF"] = {"3/4", "70"};
opcode_map["LDL"] = {"3/4", "08"};
opcode_map["LDS"] = {"3/4", "6C"};
opcode_map["LDT"] = {"3/4", "74"};
opcode_map["LDX"] = {"3/4", "04"};
opcode_map["LPS"] = {"3/4", "D0"};
opcode_map["MUL"] = {"3/4", "20"};
opcode_map["MULF"] = {"3/4", "60"};
opcode_map["MULR"] = {"2", "98"};
opcode_map["NORM"] = {"1", "C8"};
opcode_map["OR"] = {"3/4", "44"};
```

OPTAB

CONCLUSION

It was a great experience implementing the SIC/XE Assembler. I learnt how to implement literals, expressions, symbol defining statements using Program Blocks which help me to improve my concepts.