**Group Project Assignment**

**STIK 1014 - Computer System Organization**

**Semester A172**

**Task**

You will write a simple assembler program in whatever high level language you know ( mostly will use java). The assembler will translate assembly source code written in SI86 assembly language code to machine language file.

The translation will be done in a two pass process explained below.

This is a group project (in 2 or 3 person in a group). you will send in the program and further do a presentation for the project to be arranged with your lecturer.

This project will contribute 20% of your final mark, so please do it properly and give due effort to complete it.

You will divide job properly between group members. Submission will be done in three phases as follows:

**Phase one** (**5 marks**): submit a working code for reading the source code and successfully identifying the different part of the source code consisting of the opcode, label, and operands. You will also generate the symbol table listing properly by this time. Basically, the first pass of the 2 pass assembler is complete.

**Phase two ( 5 marks):** submit a working code for the generation of the machine code properly in a proper object file ( .obj)

**Phase three (10 marks):** submit a complete working program of the simple assembler with addition consisting of code for generating the listing file (.lst) and proper error checking of the source code program. You will also have prepare for the presentation needs.

The phases due date will be decided by the lecturer. Please observe the submission time properly.

**Introduction**

An assembler is a program that reads an assembly source code and translate the code into a machine language code (target code = object file) for a specific processor of a given computer. An assembler basically works in the way explained in diagram 1.

Diagram 1:

Source code

Listing

file

SymTab

file

object code

**Pass 1**

**Pass 2**

Intermediate file

Symbol table

opcodeTable

Symbol Table

**Processor architecture**

The target machine processor is a simple machine with a processor known as the SI86 processor. This machine is an experimental machine, thus it is made very simple. SI86 will have 2 registers only known as R1 and R2. both of the registers are 8 bits register. The memory of the machine is 24 bytes = 16 bytes. Each memory space is 8 bits (1 byte).

The memory of the computer is a small memory with address ranging from 00h to FFh - the address is 8 bits with every address represents an 8 bits(1 byte) space.

**The assembly language**

The assembly language you will be working with have a very simple instruction set consisting of 10 instructions. The instructions along with its needed specification are listed in table 1 below. The instruction set is basically all using only register direct and immediate addressing modes only (no direct data movement to memory address is allowed). It can have jump and branch instructions to certain address which means that generic name identifier can be used as label. The instruction specification is described in the table below:

|  |  |  |
| --- | --- | --- |
| **Instruction opcode** | **operands** | **description** |
| add | x1,20  x1,x2 | add 61 to x1 and load result to x1  add x2 to x1 and load result to x1 |
| sub | x1,20  x1,x2 | Subtract 61 from x1 and load result to x1  Subtract x2 from x1 and load result to x1 |
| ld | 61 | move data(61) into reg X1 |
| mov | x1,x2  x1,20 | Move data from register to register  Move data (61) into register x1 |
| sto | num | move content of R1 to memory address (num) |
| jmp | lbl | Jump to a label lbl |
| brneg | lbl | Branch if negative to label lbl |
| brpos | lbl | Branch if positive to label lbl |
| end |  | End of code |

The assembler will generate the machine code as the target in a process known as a two pass assembler steps.

**Instruction translation**

The general format of SI86 machine is is detailed in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **opcode** | **Opcode Machine translation** | **operand** | **Instruction size** |
| **mov** x1,x2  (reg mode) | 0000 | 0001 for x1,x2  0010 for x2,x1 | 1 byte |
| **mov** x1,20  (immediate mode) | 0001 | 0001 for x1 + data (8 bits)  0010 for x2 + data (8 bits) | 2 bytes |
| **add** x1,x2  (reg mode) | 1000 | 0001 for x1,x2  0010 for x2,x1 | 1 byte |
| **add** x1,20  (immediate mode) | 1001 | 0001 for x1 + data (8 bits)  0010 for x2 + data (8 bits) | 2 bytes |
| **sub** x1,x2  (reg mode) | 1100 | 0001 for x1,x2  0010 for x2,x1 | 1 byte |
| **sub** x1,20  (immediate mode) | 1101 | 0001 for x1 + data (8 bits)  0010 for x2 + data (8 bits) | 2 bytes |
| **load** x1,20 | 0010 | 0001 for x1 + data (8 bits)  0010 for x2 + data (8 bits) | 2 bytes |
| **sto spc** | 0011 | Address | 1 bytes |
| **jmp** lbl | 0100 | Address of lbl | 1 byte |
| **brneg** lbl | 0101 | Address of lbl | 1 byte |
| **brpos** lbl | 0111 | Address of lbl | 1 byte |
| **end** | 1111 | 0000 | 1 byte |

**Translation process**

An assembler is a translator, that translates an assembler program into a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created.

**Pass 1**: Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, name of data fields and programs labels and their relative location (offset) within the segment. Pass 1 will determines the amount of code to be generated for each instruction.

The symbol table general structure is as specified in table 1.

|  |  |  |
| --- | --- | --- |
| **identifier** | **Address (offset loc in the code)** | value |
|  |  |  |

**Pass 2**: The assembler uses the symbol table that it constructed in Pass 1. Now it knows the length and relative of each data field and instruction, it can complete the object code for each instruction. It produces .OBJ (Object file), .LST (list file).

**Example**

**A. Source code file**

**; .code is assembler directive for code segment**

**.code**

**mov R1,num1**

**add R1,num2**

**brneg lbl1**

**sto spc2**

**lbl1 sto spc1**

**jmp fin**

**fin end**

**; .data is assembler directive for data segment**

**.data**

**num1 .eq 04**

**num2 .eq -05**

**spc1 .eq ??**

**spc2 .eq ??**

During the first pass, the symbol table contents will be:

|  |  |  |
| --- | --- | --- |
| **identifier** | **relative address** | **value** |
| num1 | 1001 | 00000100 |
| num2 | 1010 | 11111011 |
| lb1 | 0110 |  |
| fin | 1000 |  |
| spc1 | 1011 | ?? |
| spc2 | 1100 | ?? |

After the second pass, the **listing file ( .lst)** generated will be:

0000 00010001 00000100 11 14 **mov R1,num1**

0001 1001000111111011 91 FB **add R1,num2**

0010 0101011056 **brneg lbl1**

0011 001111003C **sto spc2**

0101 00111011 3B **lbl1 sto spc1**

0100 0100100048 **jmp fin**

0110 11110000F0 **fin end**

The above listing description is as follows:

Column 1 - line address

Column 2 and 3 - machine code translation in binary

Column 4 and 5- machine code translation in hexadecimal

Column 6,7 and 8 - assembly listing.

Listing file (.lst) will include error message, in case there is error (mainly syntax error). If there is error, no obj file will be generated. The error message will be printed printed in the .lst file instead of the program listing as above. The error message will be listed in the format of:

<Line number> <error description>

Besides the listing file, the **symbol table file (symtab)** generated will be as follows:

**N a m e Type Value Attr**

num1 . . . . . . . . . . . . . . Byte 1001 \_DATA

num2 . . . . . . . . . . . . . . Byte 1010 \_DATA

lbl . . . . . . . . . . . . . . Label 0110 \_TEXT

fin . . . . . . . . . . . . . . Label 1000 \_TEXT

spc1 . . . . . . . . . . . . . . Byte 1011 \_DATA

spc2 . . . . . . . . . . . . . . Byte 1100 \_DATA

**Furthermore, the object file (.obj) produced will be:**

00010001 00000100

1001000111111011

01010110

00111100

01001000

00111011

11110000

**GOOD LUCK!**