**Project report**

**Abstract or title :** implementation of a single pass assembler using SIC/XE machines.

**Introduction :**

**Assembler** – the purpose of an assembler is to convert an assembly program to machine level language or to an object code. The object code generated is loaded into the memory using a loader.

**Single pass assembler :** an assembler that generates the object code by scanning the assembly program only once.

**SIC machines** – This machine is used to illustrate the most commonly used hardware features and concepts , while avoiding most of the complexities encountered in real machines.

**SIC/XE machines** – It is an extended version of the SIC machines which supports extra instruction formats and addressing modes. Here, XE stands for extra equipment. The machine is designed to be upward compatible with the SIC machines.

**Problem description :**

The problem is to write a code for a single pass assembler that generates the object code of an SIC/XE assembly program. The assembler should support all the four formats of SIC/XE machines and the all the six different addressing modes.

The main problem is to assemble a program in a single pass which involves forward reference problem. Forward reference problem occurs when the assembler encounters a symbol which has not yet been defined, that is, a symbol whose memory address is not known to the assembler.

**Problem solution and approach :**

To resolve the forward reference problem one way is to place all the storage reservation instructions in the beginning of the assembly program, so that most of the symbols are defined before the assembler scans the actual program.

But, there can also be some symbols/operands in the actual program which are not defined. To deal with it, what the assembler does is that while scanning the program, when it sees an unknown operand it records the symbol in a table called SYMTAB and mark it as undefined. It also inserts the forward reference address of the symbol in a list associated with that symbol entry. When the object code generated for this instruction, then 0’s are put in place of it’s address.

When the assembler encounters the definition of the symbol, that is, when the symbol is assigned a memory address, then a new text record is generated which tells the loader that forward reference address has to be replaced with the actual address of the symbol.

Example : T^1017^202E - it indicates that the object code at the memory address 1017 is to be replaced by 202E

**Algorithm implementation approach :**

* First approach is to create the tables : **OPTAB , SYMTAB , DIRECT and REGISTER**

All of the tables are implemented using the ‘map’ container which is provided by the standard template library of C++. The map container facilitates searching in O(nlogn) time. It also helps in assigning a key to a value(or group of values) though which that particular value can easily be searched.

**OPTAB –** It stores the mnemonic and the corresponding opcode.

Map<mnem , opcode> , where mnem is the name of the instruction and opcode is a structure which contains the corresponding opcode.

**SYMTAB –** It stores the symbols, their address values, and the forward references of each of the corresponding symbols.

Map<symbol, label> , where symbol is the name of the symbol and label is a structure containing the memory address of the symbol and a list containing the forward references for a particular symbol.

**DIRECT –** It stores the SIC/XE assembler directives.

Map<directive, existence> , where directive is the name of the directive and existence is a Boolean value to check whether the directive exists or not.

**REGISTER –** It stores the name of the registers and their corresponding numbers.

Map<reg\_name, reg\_num> , where reg\_name is the name of the register and reg\_num is the corresponding register number.

* Now we need functions to facilitate the conversion of a decimal number to its corresponding hexadecimal form and vice versa, because these functions will be needed very often in the implementation of the algorithm.

We also need a function to find the 2’s complement of a number and a function which pad 0’s at the front of a hexadecimal number.

* A function to increment the location counter :

**void incrmTheCntr(string opcode, string operand)**

This function increments the location counter based on the format of the opcode.

If opcode has a ‘+’ sign in front of it then it is format 4, so increment the counter by 4.

If opcode is RESB, then increment the counter by the value specified in the operand field.

If opcode is RESW, then increment the counter by three times the value specified in the operand fiels because a WORD is of 3 bytes.

If opcode is WORD, then increment the counter by 3.

If opcode is BYTE, then there are two cases –

1. If the operand is a string of characters then increment the counter by the length of the string.
2. If the operand is a hexadecimal number(usually denoted by a ‘X’ at the front), then increment the counter by half the length of the hex number, because one hex digit is of half bytes.

* A function to generate the object code of an instruction :

**string generateCode(string opcode, string operand)**

three string variables code1, code2 and code3 are taken which is used to store the object code.

Code1 contains the first 8 bits of the object code of which the first 6 bits is the opcode of the instruction and the next 2 bits represents indirect and immediate flags respectively .

Code2 contains next 4 bits which represents the remaining four flags, which are x, b, p and e. But in case of format 2 it is used to store the register numbers which is of 4 bits each.

Code3 contains the displacement in hex.

If the opcode is ‘**RESW**’ or ‘**RESB**’ then return an empty string.

If the opcode is ‘**WORD**’ then return the hexadecimal equivalent of the operand

If the opcode is ‘**BYTE**’ then there are two cases –

1. If the operand is a hexadecimal number then return the number itself
2. If the operand is a string of characters then covert the string to their equivalent ASCII value and then return its hexadecimal form.

Check the formats of the opcode. If an opcode has a ‘+’ sign in front of it then it is of format 4. Depending on the format of the opcode, different procedures are required to compute the object code.

For format 1 :

Simply return the opcode of the corresponding instruction.

For format 2 :

Store the opcode of the instruction in code1.

There are two cases for operands –

1. If the operand consists of two registers separated by a “,” then store their respective register numbers in code2.
2. If the operand consists of only a single register the store its register number in code2 and pad it with a ‘0’ which indicates last 4 bits of are zeroes .

Now, return (code1 + code2)

For format 3 :

Store the opcode of the instruction in code1. Assign a zero value to code2.

If the operand has a ‘#’ sign in front of it then it indicates that it is an immediate addressing

mode. Add 1 to code1 to set the i flag. If the operand is a value then convert the operand to its

equivalent hexadecimal form and store it in code3. Return (code1 + code2 + code3).

If the operand has a ‘@’ sign in front of it then it indicates that it is an indirect addressing mode.

Add 2 to code1 to set the n flag in code1.

If there is no sign in front of the operand then it is a direct addressing mode. Add 3 to code1 to

set the n and i flags.

If the operand has a ‘,X’ sign at its end then it indicates that it is an index addressing mode.

Add 8 to code2 to set the x flag.

Now, check whether the operand exist in SYMTAB or not.

If not, then store it in the SYMTAB and put a ‘\*’ mark in its address field to indicate that it is not

yet defined. Also push the forward reference address in a list associated with it. Store zeroes in

code3 which indicates that the operand is not yet defined. Return (code1 + code2 + code3).

If yes, then store its actual address in its address field. Then check whether it supports PC

relative or base relative addressing mode.

If it supports the PC relative mode then calculate the displacement value and store it in code3.

Add 2 to code2 to set the p flag. Return (code1 + code2 + code3).

If it supports the base relative mode then calculate the displacement value and store it in code3.

Add 4 to code2 to set the b flag. Return (code1 + code2 + code3).

If it does not support any of these modes then show a memory overflow error.

For format 4 :

The procedure is almost same as that of format 3. The difference in this case is that code3 will

store 20 bit address because the displacement value is of 20 bits in extended format. The b and

p flags will always be zero and e flag is always 1. So code2 is assigned the value 1 to

set the e flag. For each format 4 instruction a modification record is generated.

**ALGORITHM :**

prog\_counter 🡨 0

start 🡨 0

location\_counter 🡨 0

Base 🡨 0

Read a line from the input file

Parse the line into tokens

If (the first token is START) then

start 🡨 second token

Location\_counter 🡨 start

Prog\_counter 🡨 start

Else if( the second token is START) then

start 🡨 third token

location\_counter 🡨 start

prog\_counter 🡨 start

While( the END directive does not occur )

Read a line from the input file

Parse the line into tokens

location\_counter 🡨 prog\_counter

If (the second token is a directive) then

If (the first token does not exist in SYMTAB) then

Insert the token in the SYMTAB

Store its address

Increment the location\_counter

Generate the object code and add it to BLOCK.object

BLOCK.length 🡨 BLOCK.lenght + length of BLOCK.object

Else

Show a duplicate symbol error

Else

Break the loop

Write the BLOCK.start, BLOCK.length and BLOCK.object in the INTERMED file

prog\_start 🡨 prog\_counter

BLOCK.start 🡨 prog\_counter

BLOCK.object 🡨 NULL

BLOCK.length 🡨 0

While(the END directive does not occur)

Read a line from the INPUT file

Parse the line into tokens

location\_counter 🡨 prog\_counter

if(the first token is an instruction)

increment the location\_counter

generate the object code and add it to BLOCK.object

BLOCK.length 🡨 BLOCK.lenght + length of BLOCK.object

else

if(the first token is a label and it does not exist in SYMTAB)

insert the label in SYMTAB

add its address value

increment the location counter

generate the object code and add it to BLOCK.object

BLOCK.length 🡨 BLOCK.lenght + length of BLOCK.object

else if(the label exists in SYMTAB and it is undefined) then

define the label by assigning its address value to it

write the BLOCK.object, BLOCK.length and BLOCK.object in INTERMED file

BLOCK.object 🡨 NULL

BLOCK.start 🡨 prog\_counter

BLOCK.length 🡨 0

Increment the location\_counter

Generate the object code and add it to BLOCK.object

BLOCK.length 🡨 BLOCK.lenght + length of BLOCK.object

If (the instruction is of format 4) then

Write the modification record to the MODIFY file

Else

Show a duplicate symbol error

Break the loop and abort the program

If(the SYMTAB contains an undefined label) then

Show an undefined symbol error

Abort the program

Write BLOCK.start, BLOCK.length and BLOCK.object in INTERMED file

Calculate the length of the program

Write the header record to OBJECT file

Read every line of the INTERMED file and write it to OBJECT file

Read every line of the MODIFY file and write it to OBJECT file

Write the end record to the OBJECT file