**ASM2MIF Compiler Design**

ScomP Microprocessor

FINAL REPORT

*EEL5930 Final Project Repot, December 6th, 2015*  
*Sourindu Chatterjee, Electrical & Computer Engineering Florida State University*

**Abstract:- This archive plots the knowledge the creator has picked up from the semester project of EEL 5930, "Embedded Micro System Design”, at Florida State University under professor Dr. Uwe Meyer-Baese. The work explores insightful approach to design and develop a c compiler for the SCOMP microprocessor assembly, using tools like FLEX – Fast Lexical Analyzer. It would convert the SCOMP assembly instruction set to MIF – Machine Instruction File, with Opcode and Initial Data Values.  
This document is outlined as follows: Section I Introduction; Section II Documentation and Theory; Section III continues on the documentation part extending it to the code used in the development of the compiler; Section IV Aspects of Designed Compiler; Section V Conclusion and Future Research Perspective; Section VI References; Section VII Appendix consisting of the .l file, .asm & .MIf files.**

1. **Introduction**

The SCOMP instruction set for the µP3[1] microprocessors set has been in development in recent years, thus a custom instruction set consisting both the operation specific instructions and data or branching specific instruction, have been proposed for the study and analysis of simple computers like µP3. This job is classically accomplished by compilers and assemblers like GCC but in this special custom microprocessor the use of a compiler to translate the symbolic assembly instruction to Opcode[2] instruction was not profound. The purpose of this project is to design a c compiler for mapping the SCOMP assembly instruction to the MIF instruction code directly readable by the microprocessor. Now in order to accomplish that use of FLEX is profound as it provides the necessary basis for the pattern analysis and recognition in text. Well it uses the standard c code to provide the action description for the specified patterns.

1. **Documentation**

This part primarily deals with the theory of the FLEX tool and SCOMP instruction set. As discussed previously the FLEX generates a scanner program for analyzing specified patterns in input text and then matches the same with specified patterns for executing specific tasks or action. The description is defined as sets of normal expression and c code defined as rules[3]. FLEX produces an output file yy.lex.c if otherwise not specified, which if compiled with gcc or cc on Windows system produces a executable file which runs on windows and if that is compiled on Unix system produces a executable file capable of running on the Unix system. This executable file is then used with a specified input file in this case “example.asm” to get the desired output “example.MIF”.

Thus the format of the input flex “.l” file can be divided into three primary sections: first the **Description**, second the **Rules** and third the **User Code**. These sections are separated by %% in between them. Along with these an additional section can also be defined namely **Declarations**, which contains the c header files and all the global variables, arrays, and c structures thus making those accessible throughout the code. In this project the Opcode structure is defined in the declaration part. Also any commented text for programmers readability can also be included in this section with “/\* \*/” like traditional c code. A typical example of the declarations used in this project in the “asm2mifv32.l” is provided in Figure 1a & Figure 1b.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Figure 1a.** |  | **Figure 1b.** |

The description for patterns as used in these compiler design along with actions specifies for each rules are discussed later in section III, which will further enhance the understanding of the application of FLEX in the development of a custom instruction compiler.

MIF or Machine Instruction File is a unique approach to send information to the microprocessor in this case a SCOMP[4] microprocessor which as previously discussed consists of a 16 bit information word length composed of 8 bit Opcode and 8 bit memory address or branching label address whereas in some unique cases the latter half of the instruction code consists of specific values to be used in processing operation. The typical example of the Opcode table used in this compiler design project is given in Figure 2.

|  |
| --- |
|  |
| **Table 1 Opcode Instruction Table[4]** |

In this SCOMP assembly instruction set there are certain special instructions such as “ADDI” and “SUBI” with opcode “0B” & “0C” respectively which puts the last 8 bit of the instruction directly to the Accumulator for adding or subtracting from its magnitude. Thought this does not directly affect the flow of the compiler while executing the instruction but it does performs a role, which is essential in segregating between the data addresses in the SCOMP assembly, the data values for ADDI or SUBI instruction and branching values for particular labels.

The JPOS, JZERO and JUMP instructions again deals with branching of the flow control with respect to the execution of the .MIF file but does play a pivotal role in building the compiler from the c executable custom compiler. These values needs to be mapped to the instruction number so that the specific instruction value could be put after the opcode of the mentioned three branching instructions.

1. **Code Documentation**

The finished code is written in most simple way without much fancy way to express instruction, the action for each compiler pattern is kept exclusive from each other as much as possible so that further development on the basis of understanding the previous code can be carried on. The patterns used for various purposes declared under the definitions category are of special interest and betting of discussion. The Table 1 below provides the set of patterns used in the development of the compiler.

|  |  |  |
| --- | --- | --- |
| No. | Pattern | Analyzes |
| 1. | [\r\n]+|"\r\n"+ | New Line |
| 2. | "-"?[0-9]+ | Digits |
| 3. | [Ll][0-9]+[:] | Labels |
| 4. | [a-zA-Z][a-z0-9\_]\* | Variable name |
| 5. | ("0x"|"0X")[ABCDEF0-9]{1,4} | Hexadecimal Values |
| 6. | (?i:variable) | Keyword Variable |
| 7. | [ \t]\*":"[ \t]\* | Colon |
| 8. | "--"\* | Comment |
| 9. | [" "\t]+ | White Space |
| 10. | JUMP|JPOS|JZERO|JNEG | Branching & Switching Instructions |
| 11. | ADDI|SUBI | Instruction ADDI or SUBI |
| 12. | STORE|ADD|LOAD|XOR|OR|SUBT | Memory Address Specific Instructions |

|  |
| --- |
| **Table 2 Pattern Declarations** |

A few patterns from the table given are of special interest as those a are epitome of FLEX tools of skills. The new line patterns (1) in the table is very important because is bridges the difference between the end line sequence difference between Windows and Unix system, where Windows operating system provides the end line pattern of “\r\n” and Unix operating system files uses “\n”. The above difference also exists between the use of various software like “notepad++” and “notepad” or other text editor.

Again the Hexadecimal value character (5) provides a special to disintegrate itself or the lexical text it analyzes, from the ordinary variable values, it also provides a way to segregate it by limiting the number of characters that can be possibly used to 4, which is maximum hexadecimal value that can be used by SCOMP microprocessor or the MIF instruction file.

The code also specifies certain key actions for each pattern that not only controls the flow of the program but sets the execution of the program as efficiently as possible. The compiler executes a two phase lexical analysis of the input file. It does so because on the first pass it lists all the variables with values and then passes on to number of ALU instructions and the number of Branching statements. On the second pass the compiler maps one to one the instruction serial number then the instructions Opcode for each instruction followed by the memory address value or the instruction serial number value or any decimal or hexadecimal value depending the type of instruction that is being mapped on. The overview of the input file and output is provided in the Section IV. The use of flag variables like “iflag”[6], “vvflag” & “ccflag” are of immense importance as they channelizes the compilers flow to the next succeeding 8 bit value, as perhaps is known that if the instruction is any of the ALU instructions specified in the Table 1 then the value following that would be a address value of the corresponding variable, where the data value of that variable will be listed on the specified address location whereas if the instruction is any of the Jump instructions then the succeeding value would be a address number or in this case serial number of where the specified instruction is located in the MIF file. More over if the instruction if any of the two Immediate instruction [9] then the latter half of the instruction code would be a decimal or hexadecimal value directly mapped to the output MIF file.

The Compiler does provide for specifying the initial data value of certain variables used in the program. This system is aided by use of specified pattern “variable”[10] which sets up a flagged execution where the variable name and data value both are added to the array and also printed out during the first execution cycle. The data value is again mapped depending on whether or not it is a hexadecimal or decimal value. Also during the execution of the ADDI or SUBI instruction the compiler sets up a flag to limit the execution to only converting and adding the value in case of a hexadecimal or decimal value.

IV. **Aspect Of Designed Compiler**

In this section a brief overview of the designed compiler is provided along with brief summary of the generated output. The Figure 2a below shows the inputted Assembly instruction file and Figure 2b shows another Assembly instruction file & Figure 3a and 3b provides the generated output from the Unix console for Scomp.asm and Figure 4a provides the glimpse of the output generated in Unix system and Figure 4b provides the snap of original Scomp4.MIF file of the assembly instruction file Scomp4.asm. In Section VI later the Machine Instruction File Output is provided.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Figure 2a.** Scomp.asm |  | **Figure 2B.** Scomp2.asm |

|  |
| --- |
|  |
| **FIGURE 3A.** Scomp2.asm Output First Part |

|  |
| --- |
|  |
| **FIGURE 3A.** Scomp2.asm Output Second Part |

|  |
| --- |
|  |
| **FIGURE 4A.** Scomp4.asm Output First Part |

|  |
| --- |
|  |
| **FIGURE 4B.** Scomp.MIF Output File |

V. **Conclusion & Future Research**

The project has been successfully completed and the design of the Compiler for converting SCOMP assembly instruction set to Machine Instruction File is established. This has been a stable design where every aspect of the input file has been considered and necessary inclusions in the code has been made accordingly to deal with exceptional situations. However there are limitations to the program, such that the number of instructions cannot be 30 more than the number of variables used. In such a scenario the instruction serial number or the “memory address” value to put it technically, would overlap the “memory address”[f] value of the variables and output .MIF file would end up being unstable. This situation can be dealt with but in order to do so the Compiler execution time has to be made three time instead to two times which is currently happening, which would make the Compiler inefficient as there will be 50% increase in the execution time. Thus a more refined algorithm has to be developed to pass around this issue.

Further development can be made in this field as there is a lot of scope for development of more efficient algorithm. Furthermore a reverse Compiler can be designed which would provide the Scomp Assembly Instruction file from the Machine Instruction file. Lastly there can a lot of error detection and error correction can be provided to the existing Compiler design.

**Acknowledgement**