TABLE OF CONTENTS:

LIST OF FIGURES:

1. Software Project Synopsis

1.1 Context

Current microprocessors have a very complex architecture which is due to many advanced facilities provided by them and large number of instructions supported. This also increases costs. For application specific needs, customer purchasing a microprocessor needs very few facilities or instructions for the intended use. So, it may happen that a customer pays for the facilities that are not needed.

Also Compilers for such architectures are complex and large. Assemblies generated by such compilers can't be fully optimized and they will have to select instructions very precisely as there can be number of instructions to do the same thing.

If a processor is built as per customer requirements, it will reduce cost and complexity. A new custom compiler will be needed for such a custom built processor. This compiler will be less complex and will generate more efficient assemblies. So there is need for compiler generation system to generate a 'C' compiler for custom built processor.

1.2 Problem

Customer should provide the instruction set architecture in XML file format. System should process it and modify the GCC Source code accordingly. On compiling that modified GCC source, a target cross compiler should be generated. If customer provides 'C' source file, cross compiler should generate binaries for custom-built architecture.

1.3 Solution

There will be a TCL script which should take input from customer in XML format. This XML file will contain the instructions that are needed to be supported on the custom built processor. This script then should generate .md, .c and .h files specific to that processor.

Then these three files will be fed to GCC Source Code directories ($SOURCE\GCC\Config). The Custom architecture will be registered in config.sub file in $SOURCE\GCC directory.

This modified GCC source code will be compiled using a standard C compiler on Linux Environment. After successful compilation, there will be an executable GCC generated which will then compile 'C' programs for target custom built processor.

1.4 Context Diagram

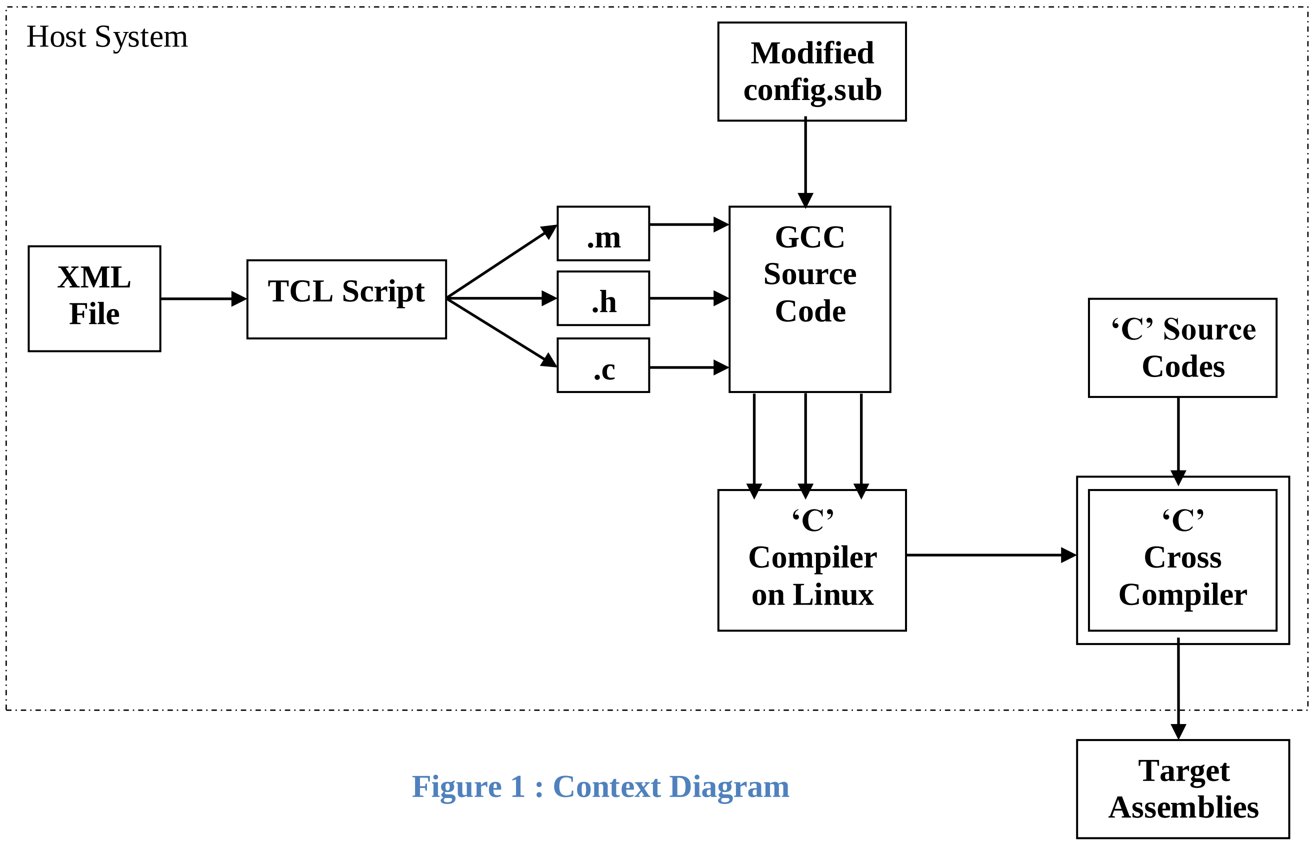


Figure 1 : Context Diagram

Automated Cross Compiler generation

2. Feasibility Study Report

2.1 INTRODUCTION

Even though GCC can be cross compiled for number of different processor architecture, it will be very cost efficient solution if there exists a custom built processor architecture which is built as per customer requirements and is cost effective. There should be an application which generates and builds ‘C’ compiler for this architecture when provided with customer requirements.

2.1.1 PURPOSE

The purpose of feasibility study is to check whether the proposed system –
- Can be implemented or not.
- Can produce a feasible solution.

2.1.2METHODOLOGY

Two varied methodologies are used to prepare this document
1. Literature Survey
2. Group Discussions

2.1.3 REFERENCES

1. http://gcc.gnu.org/onlinedocs/gccint/
2. http://www.cse.iitb.ac.in/grc

2.2 GENERAL INFORMATION

Sample customer requirements have to be provided in XML file format. This has to be provided to our application which will then build cross compiler capable of dealing with requirements provided in input XML file and generating binaries for target architecture.

2.2.1 CURRENT SYSTEMS AND PROCESSES

Full functional compilers are build which support all the instructions that are supported by the architecture for which they are built, though target user will use very few of them. This approach is not cost efficient for both architecture and compiler considering cost, speed and complexity.

2.2.2 System Objectives

The major goal of the system is to accept customer requirements as input and modify GCC source code as per these requirements. This modified source will be compiled and will build target ‘C’ compiler. This final compiler will in turn compile ‘C’ programs given by customer (user). Such compiler will be very small, less complex, fast and efficient.

2.2.3 Issues

- The Operating System (Linux) used to implement the system.
- The type of interface that displays the output.

2.2.4 Assumptions and Constraints

Assumptions:

- Customer is familiar with Linux environment.
- Customer has all the packages installed required compiling the GCC source.

Constraints:

- System is Platform dependent.
- Time required building the final compiler from modified source code.

2.3 Alternatives

Currently there is no other application/system which allows user to build compiler automatically as per his/her required instructions. Customer has to use full compiler though most of the part of compiler remain unused.

2.4 Recommendations and Conclusion

Recommendation:

In order to make project more successful, we need to minimize the wastage that occur during implementation. For achieving this, we are minimizing the risk factors involved during implementation.

The technology that we are going to use will minimize the risk factors discussed above. Tools that we are using like TCL, VIM are easily and freely available. So it will put break on economical constraints. The system can be mapped and planned for given academic year and can be completed successfully.

Conclusion:

We are implementing a system which will allow and give room to the customer to easily get compiler only supporting their requirements and the custom built architecture they are using. Customer need not to know compiler internals.

3. Software Project Plan

3.1 Overview

The purpose of this document is to collect, analyze high-level needs and features of the Automated Cross Compiler Generation. It focuses on the estimation of feasibility of the system. Such systems will give flexibility to the end user to build a compiler which will support the only instructions that he/she needs for their custom built architecture.

The system consists of a script which will accept requirements from user in XML file format. This file will be processed by script and the GCC source code will be modified. This modified source will be then built using standard ISO C compiler in Linux.

This method will be completely new and will not be extended from any existing system. The project will not cost much as everything will be done on computer and operation like data collection, its manipulation, the work done on existing methods will be done on the machine itself.

This project will estimate take a time of about 7-8 months.

3.2 Goals and Scope

3.2.1 Project Goals

Project Goal	Priority	Comment/Description/Reference
Functional Goals:		
Requirement specification	1	User will have to provide his/her requirements in the form of XML file.
UI	2	The UI will be a simple command line interface in Linux.
Product functions	3	This includes accepting requirements file from user, processing it and generating machine descriptors. Also, gcc source configuration has to be modified; this includes making entry of new architecture in source. Finally modified source will be compiled using standard ISO C compiler of Linux.
Technological Goals:		
Help System	1	A complete detailed Help system made available to the end-user of the system will contain all the support information the end-user requires in case of some difficulties in using the system arises. Help allows the user to walk through pictorial presentation of all the functionalities of the system.
Use of simple technology	2	Simple TCL scripting language will be used for coding.

Project Goal	Priority	Comment/Description/Reference
Quality Goals:		
Requirement reliability entry	1	Requirement Entry deals with reliable storing, retrieving, parsing of requirements and extracting of information required for future phases.
Data Commonality	2	The use of standard data structure and type throughout the implementation.
Portability	3	Help in implementing the system without worrying about the internal hardware by using open source software.
Constraints:		
Compilation Time	1	The time required to build compiler from modified source code is large.

3.2.2 Project Scope

1. System will accept XML file from user which will contain compiler requirements.
2. TCL script will read this file.
3. Script will then process this file and generate three output files .c , .h and .md.
4. Script will also modify configuration files of GCC source.
5. Script will then start compilation of modified source code.
6. After compilation, custom built 'C' compiler will be ready for custom built architecture.

3.2.3 Included

This project will provide facility to user to build compiler which will support instructions that he/she needs for their custom built processor architecture.

This project also describes the **requirements, constraints and system interfaces** of the system. It gives an idea to the developers and its user on what the system will shape up like and the functions the system will perform. This document also introduces all the functionalities that the system can perform and its effect on the users and the stakeholders.

This project also includes **Help System**. A complete detailed Help system made available to the end-user of the system will contain all the support information the end-user requires in case of some difficulties in using the system arises. Help allows the user to walk through pictorial presentation of all the functionalities of the system.

3.2.4 Excluded

This project will exclude the guarantee that the compiled programs by generated compiler will be fully optimized.

3.3 Schedule and Milestones

Milestones	Description	Milestone Criteria	Planned Date
M0	Start Project		10-09-2010
	Idea about project is understood.	Concise view of the project, understanding of the merits and demerits is done.	
M1	Start Planning		25-09-2010
	Objectives and strategy defined and feasibility studied	Scope, Goals and Concepts are described.	
M2	Start Execution		22-11-2010
	Complete project plan defined and reusable modules identified	Requirements agreed, project plan reviewed and resources committed.	
M3	Confirm Execution		30-12-2010
	First phase completed	Basic functionalities tested, Architecture reviewed.	
M4	Start Introduction		30-01-2011
	Screens of the product are designed.	Coding of New functionalities finished Draft documentation.	
M5	Release Product		10-03-2011
	Final version completed with enhancements	Product system tested, documentation reviewed.	
M6	Close Project		15-03-2011

4. Software Requirements Specification

4.1 Introduction

The name of system is Automated Cross Compiler Generation. This system will generate files required to build cross compiler for custom built architecture.

4.1.1 Purpose

- The purpose of the Software Requirements Specification (SRS) is to give the customer a clear and precise description of the functionality of the software to be developed and to eliminate ambiguities and misunderstandings that may exist. For the customer, the SRS will explain all functions that the software should perform. For the developer, it will be a reference point during software design, implementation and maintenance. To clarify keywords used throughout the document, a set of definitions, acronyms and abbreviations is provided in section.

4.1.2 Scope

The system will accept parameters in a specified format e.g. in XML format and will generate files required for building cross compiler for the specifications of the architecture specified.

The system will not address following issues:

- It will not provide complete functionalities of the compiler as compared to the system developed if files were written manually.

4.1.3 Definitions, Acronyms, and Abbreviations

Term or Acronym	**Definition**
GCC	GNU Compiler Collection.

4.1.4 References

1. www.gcc.gnu.org
2. http://www.cse.iitb.ac.in/grc

4.2 Overview

- The next chapter, the Overall Description section, of this document gives an overview of the functionality of the product. It describes the informal requirements and is used to establish a context for the technical requirements specification in the next chapter.

- The third chapter, Requirements Specification section, of this document is written primarily for the developers and describes in technical terms the details of the functionality of the product.

4.3 Overall Description

All the processor architectures available today were developed in 80's and early nineties. The new processors being developed today are based on those architectures only. The customer has to shape his needs around the processor architecture. But with the induction of modular processor design, the customer will be able to have a custom built processor architecture to meet his requirements which would be requiring a compiler for the same.

With the increase in number purpose specific processors, it will be more feasible to generate the files for building compiler automatically as the purpose specific architecture will not be requiring all the functionalities that previously had to be written manually.

This section will create requirements background for non-technical audience.

4.4 Problem Statement

The Problem Of	1. Generating compiler files manually. 2. Time and man hours wasted in writing these files manually. 3. Flexibility for customers.
Affects	1. Customer 2. Company's productivity
The impact of which is	1. Wastage of time. 2. Lack of productivity. 3. Human resource wastage.
A successful solution would	1. Save time of both company and consumer. 2. Will increase efficiency of customer. 3. Will provide flexibility to customer.

4.5 Product Perspective

- The processor architecture specifications will be provided to the system in an specified format.
- The system will read the specifications provided.
- It compiler will then generate the files required to build the cross compiler.

4.5.1 Product Position Statement

For	Customers in requirement of custom architecture processor.
Who	Wants the architecture to be designed around very specific needs.
That	Will provide the customer the flexibility of increasing the efficiency without increasing cost of system.
Our product	Increases focus on task completion rather than provide a generalized solution.

4.5.2 System Interfaces

There will be no graphical user interface as such because the main task of the system is to generate cross compiler files for the input provided in a specified format to the system. The basic user interface will be command line interface as provided by Linux.

4.6 Product Functions

The System will basically be atomizing the generation of cross compiler files needed for the architecture.

4.7 User Characteristics

The input provided to the system will be provided by technical experts after taking sufficient input regarding the architecture form the customer.

4.8 Constraints

Since this is the compiler for the specific processor architecture, the input provided to the system must be technically feasible.

4.9 Assumptions

- The processor architecture exists for the input being provided.

4.10 Specific Requirements-

This specific requirement document section contains all the software requirements at a level of detail sufficient to enable designers to design a system to satisfy those requirements.

4.11 Function

The compiler is developed to generate the files required to build cross compiler. The system takes processor architecture specifications as input and generates the machine description files as output.

4.12 Logical Database Requirements

Our system will not have any logical database requirement. The requirement will be the availability of GCC with the required files necessary for the compilation process of the system to complete successfully.

4.13 Software System Attributes

4.13.1 Reliability

The reliability of the system is achieved using following:

- The system will run on Ubuntu 10.04 or higher versions.
- The automatic generation of machine description files will reduce element of human error.

4.13.2 Availability

As automated Cross Compiler Generation needs GNU compiler collection for generation, it is easily available as an open source on internet. GCC has by far the most number of functionalities available and supports most programming languages thus providing flexibility to the developer in developing the system.

4.13.3 SECURITY

As the system is being developed in linux, the security issues drop to the minimum as this system is best known for its security. Once the machine description files are generated, these files will constitute the inner level of the system.

4.13.4 PORTABILITY

The System will run on various linux platforms such as Ubuntu, Open Suse, Fedora, Red Hat etc.

4.14 GCC – An Introduction

4.14.1 Overview of GCC

The GNU Compiler Collection (GCC) is a compiler system introduced by the GNU Project supporting various programming languages. It has been adopted as the standard compiler by most modern Unix-like operating systems such as Linux. It is possible to support GCC ports for various target platforms. The target processor families supported by GCC include MIPS, I386, and MMIX etc. GCC is retargetable. One can use it as a cross compiler component of a complete cross development system. For e.g. one can generate a cross compiler for MIPS architecture on i386 machine. Compilers are fairly complex systems that take a program written in one language and transform it into an equivalent program in another language.GCC consists of about 2.1 million lines of code that has been in development for over 15 years. GCC is open source. The availability of its source code allows one to add new features to the compiler. When a new functionality is implemented, the source code of GCC gets modified directly.

GCC gets most of the information about the target from machine descriptions
which give a specification for each of the machine's instructions. Our project deals with
the machine descriptions which are machine dependent.

4.14.2 GCC Compilation Process

There are 3 components of the pipeline (source code to object code translation process). They are Front end, Middle end and Back end.

Front End: It reads and validates the syntax of the input program. It takes away all the spacing, comments and other formatting and converts the original program into a more concise form called intermediate representation (IR). These representations are internal data structures that are much easier to manipulate than the original stream of text. The front end parses the input program and builds a data structure called Abstract Syntax Trees (AST), which represents each statement in a hierarchical structure.

Middle End: With the intermediate representation built by the front end, the middle end proceeds to analyze and transform the program into a more efficient form. All the transformations done here and in the back end usually have two goals:

• Make the object code run as fast as possible (performance optimizations).

• Make the object code take as little space as possible (space optimizations).

These optimizations are typically machine and target independent. Also Data Flow and Control Flow Analysis are required for making optimized decisions. The following optimization techniques are used in standard optimizing compilers.

• **Algebraic Simplifications:** Expressions are simplified using algebraic properties of their operators and operands. For instance, i+1-i is converted to 1. Other properties like

associativity, commutativity, and distributivity are also used to simplify expressions.

• **Constant Folding:** Expressions for which all operands are constant can be evaluated at compile time and replaced with their values. For instance, the expression a = 4 + 3 - 8 can be replaced with a = -1. This optimization yields best results when combined with constant propagation.

• **Redundancy Elimination:** There are several techniques that deal with the elimination of redundant computations. Some of the more common ones include:

o **Loop-Invariant Code Motion:** Computations inside loops that produce the same result for every iteration are moved outside the loop.

o **Common Sub-Expression Elimination:** If an expression is computed more than once on a specific execution path and its operands are never modified, the repeated computations are replaced with the result computed in the first one.

o **Partial Redundancy Elimination:** A computation is partially redundant if some execution path computes the expression more than once. This optimization adds and removes computations from execution paths to minimize the number of redundant computations in the program. It encompasses the effects of loop-invariant code motion and common sub-expression elimination.

Back End: It performs the final mapping to object code (machine code for target architecture). At this stage, the compiler needs to have a very detailed knowledge about the hardware where the program will run. This means that the intermediate representation of the program is usually converted into a form resembling machine language. In this form, we can apply transformations that take advantage of the target architecture.

• **Register Allocation:** Tries to maximize the amount of program variables that are assigned to hardware registers instead of memory (registers are orders of magnitude faster than main memory).

• **Code Scheduling:** Takes advantage of the super-scalar features of modern processors to re-arrange instructions so that multiple instructions are in different stages of execution simultaneously.

Even after final code generation, the resulting code is typically not ready to run as it is. Some compilers like GCC generate assembly code, which is then fed into the assembler for object code generation. After object code has been generated, the linker is responsible for collecting all the different object files and libraries needed to build the final executable.

4.14.3 INTERNAL ORGANIZATION OF GCC

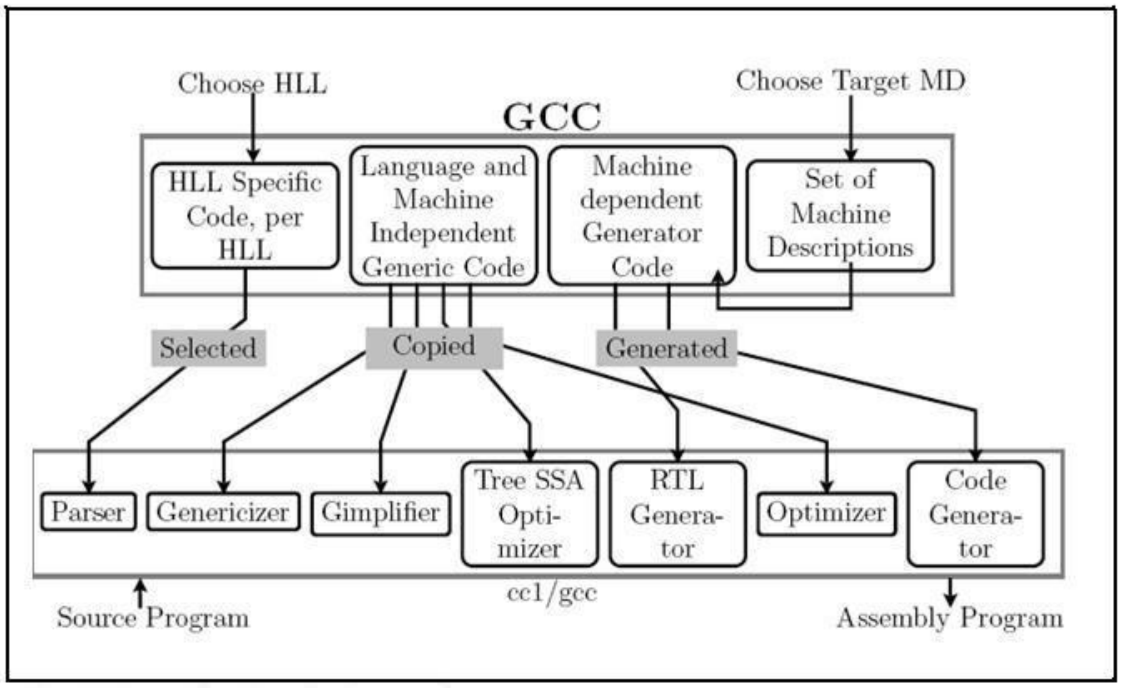


Figure 2 : Internal Organization Of GCC[Ref: http://gcc.gnu.org]

While compiling any high level language, the input program is scanned and parsed to generate abstract syntax tree (AST) using lang.hooks.parse file. The existing compilation process generates a separate abstract syntax tree for each front end. The C parser generates C trees; the C++ parser generates C++ trees, etc. The analysis and optimizations on each version requires N different implementations, which is not feasible.

RTL is a low level representation in the form of assembly language of the target machine. It works well for the optimizations that are close to the target (for e.g. register allocation, delay slot optimizations, peepholes, etc.) RTL is designed in a way that provides ease of portability.

Some analysis and transformations need higher level information about the program, which is very difficult to obtain from RTL, since too many target features are exposed.

To overcome the above problem, a new framework was generated based on Static Single Assignment (SSA). Since it was implemented on the top of the tree data structure and uses SSA, it was named as Tree SSA.

GCC parse trees provide a great detail of information about the original program. But it is not suitable for optimization for two main reasons. Firstly, there is no single tree representation shared by all the front ends. Secondly, these trees are quite complex since

they are the exact representation of the source program. To overcome these limitations, two new tree-based representations were introduced called GENERIC and GIMPLE. GENERIC trees are a superset of all the existing tree representations in GCC. It removes the language dependencies from the program representation. GIMPLE is lexically identical to GENERIC but it imposes structural restrictions on statements. For the languages like C, middle end invokes the function gimplify_function_tree in gimplify.c to flatten the generated AST to GIMPLE code.

The Static Single Assignment form is a representation laid on top of GIMPLE that explicitly exposes the flow of data within the program. The central idea is to create a new

version of each variable, and using that version the next time. The compiler looks up the latest version and uses it when required.

The middle end performs various optimizations on GIMPLE code and finally converts the gimplified function into RTL form using the function expand_gimple_basic_block from the file cfgexpand.c.

The back end then performs various optimizations on the RTL code, and finally generates assembly code using machine descriptions for the desired underlying architecture, for which the compiler is built. The machine descriptions are specific to architecture and contain the information about the target machine. It has three parts i.e. **.md** file, **.h** file and **.c** file.

4.15 MACHINE DESCRIPTIONS

4.15.1 OVERVIEW

Machine descriptions refer to a high level description of the CPU. It is the specification of all the required values of the target system for which we have to build a cross compiler. It includes a description of the target CPU, the target architecture relevant for compilation (e.g. the word size), the target system software (e.g. the OS), and the required particulars of the software development tool chain (e.g. the layout of the assembler file).

The GCC Machine Description, MD for short, is mainly composed of a set of C Preprocessor macros and a description of the target CPU in RTL. Some additional C files are also included depending on the requirements of the implementation. Referring to the root of the GCC source tree as $GCCHOME, all the MD files are found in $GCCHOME/gcc/config/<target>/ directory.

The MD files are located during the configure process via the -target= switch (if not specified, the target is assumed to be the machine on which configure is running).

The GCC build first uses these files to fill up the target dependent parts of the compiler and conceptually generates the sources for the requested target. It then proceeds to compile the compiler for the target. For example, during the generation, the GCC build system makes a note of the instructions that the target supports (as specified in <target>**.md**). This information is used during the conversion of AST to IR, and during

the conversion of IR to target ASM. The <target>**.md** file, therefore, specifies two main

pieces of information: the structure of the IR for a given AST node type, and the target

ASM instruction sequence. To facilitate lookup, a “key” is also required.

The GCC framework generates a compiler for a given architecture by reading the machine descriptions for it. These machine descriptions are difficult to construct, understand, maintain, and enhance because of the verbosity, the amount of details, and

the repetitiveness. There are several machine descriptions available for different target architectures. All of them have a common format. However there are more differences

than similarities. The size of these machine descriptions is quite large. They are difficult
to understand and maintain as well. The detailed explanation regarding machine descriptions is explained below.

To add new port for some target machine in GCC, the information about the target architecture must be provided to GCC, through machine description files. If the basic features of the target are not given to the compiler, it fails to build. The basic features include

• Register set.

• Storage layout.

• Activation record design.

• Addressing Modes.

• Assembly file output format along with some miscellaneous
parameters.

The information supplied by the machine descriptions are as follows:

• The target instructions – as ASM strings.

• A description of the semantics of each.

• A description of the features of each like:

  o Data size limits

  o One of the operands must be a register

  o Implicit operands

  o Register restrictions

4.15.2 Classification of Machine Descriptions

The information that is needed by GCC to build a compiler for any given processor can be broadly divided into two groups:

4.15.3 Machine Dependent Information: It is a detailed description of computational, communication and data storage elements (hardware) of a computer system, how these components interact (machine organization), and how they are controlled (Instruction set).

• **Instruction Set Architecture:** It describes the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. It mainly deals with design of instruction set. The attributes needed for designing the instruction set are also included in this class.

• **Micro-Architecture:** It is the set of processor design techniques used to implement the instruction set. It consists of low level details of a machine such as pipelining, memory management systems, specialized instructions processors, buses for communication within and between system and the design and layout of a microprocessor.

4.15.4 MACHINE INDEPENDENT INFORMATION: This information does not directly depend upon the target architecture as it holds information needed by GCC depending on the selection of a target machine. This information can be further divided as:

• GCC specific information

• Function calling convention

• Assembly file output format.

• Layout of source language data-types.

• Runtime target specifications.

• Tool chain specific information.

Classifying the information required by GCC in this manner helps us in organizing the architectural details of target machine in a better way.

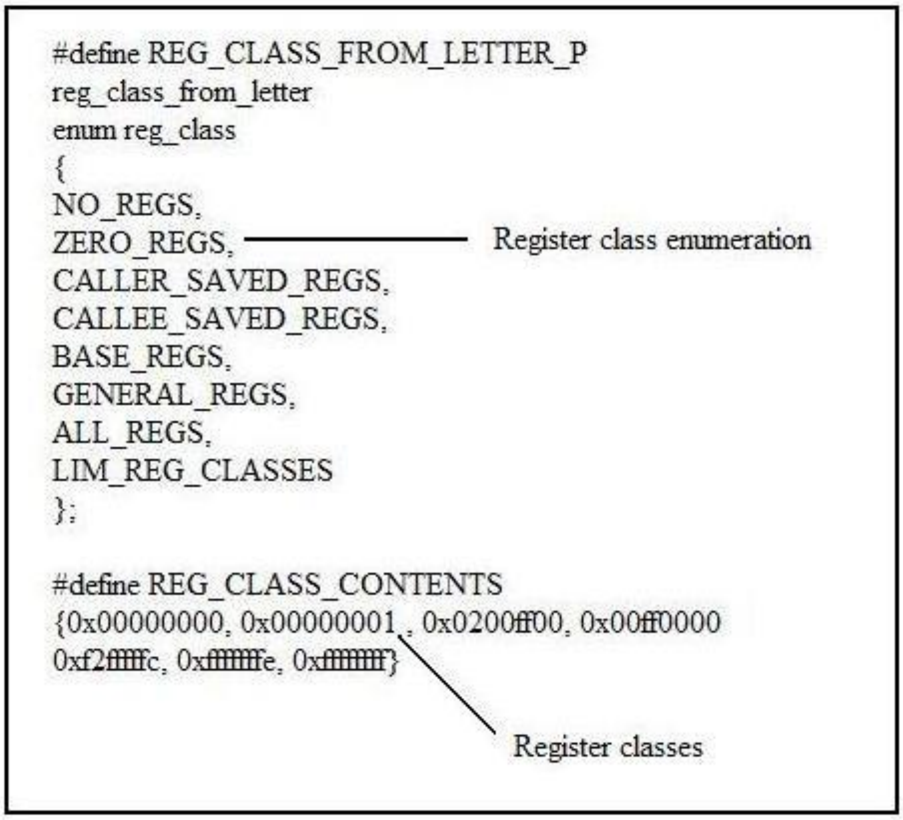
4.15.5 ORGANIZATION OF MACHINE DESCRIPTIONS

The GCC machine descriptions are made up of several files, each containing some part of the description. They are comprised of the following files:

The <target.h> file

The macros file are referred to as the “MD header” file, and are denoted by <target>.h, where <target> will be replaced by the actual target name if necessary(for example SPIM.h).

As an example of “target-cpu-independent” information that GCC MD files contain, we refer to the **TARGET_ASM_FUNCTION_PROLOGUE** (and the complementary **TARGET_ASM_FUNCTION_EPILOGUE**) macro found in the **<target>.h** file. This macro is responsible for defining the structure of the activation record implementation for the target. There are macros that specify the syntactic headers required by the target assembler in its input. The following is an example of a macro taken from SPIM.h



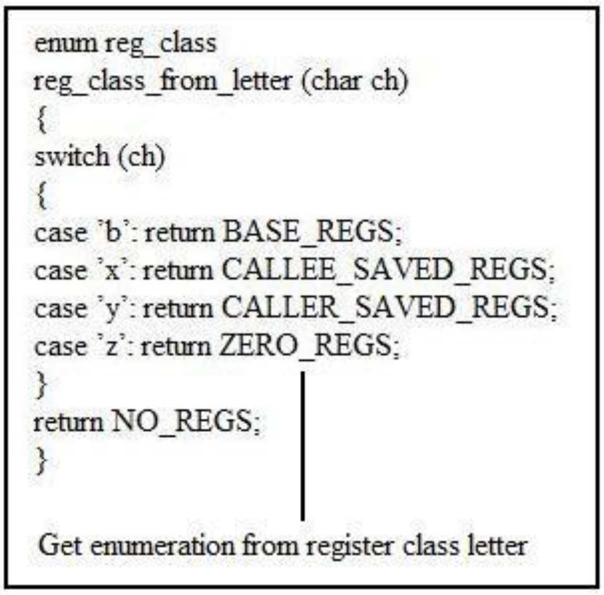
```
#define REG_CLASS_FROM_LETTER_P
reg_class_from_letter
enum reg_class
{
NO_REGS,
ZERO_REGS,
CALLER_SAVED_REGS,
CALLEE_SAVED_REGS,
BASE_REGS,
GENERAL_REGS,
ALL_REGS,
LIM_REG_CLASSES
};

#define REG_CLASS_CONTENTS
{0x00000000, 0x00000001 , 0x0200ff00, 0x00ff0000
0xf2fffffc, 0xfffffffe, 0xffffffff}
```

Register class enumeration
Register classes

Register Classes. [Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

The <target.c> file

The machine descriptions are made of preprocessor macros. Some of these macros are, or may be implemented as, C functions. These supporting C functions reside in <target>.c file. The following is an example of a macro taken from SPIM.c.



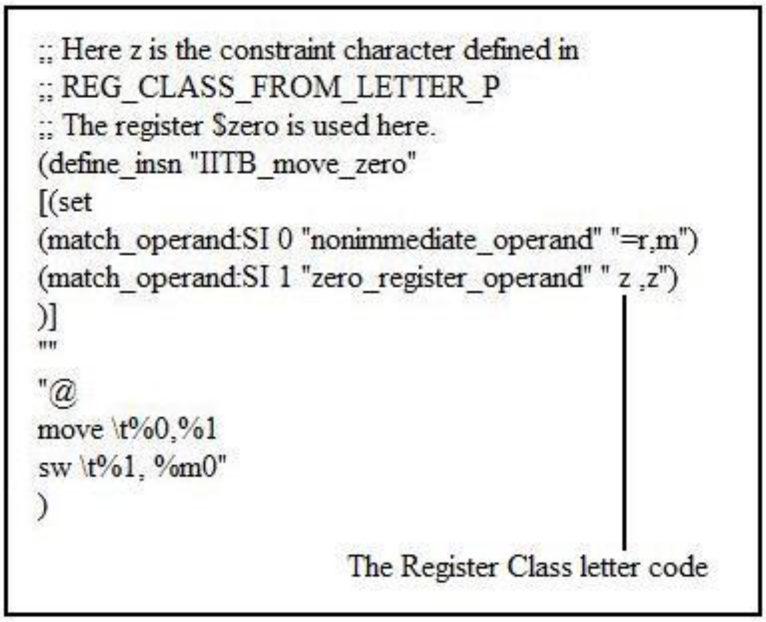
```
enum reg_class
reg_class_from_letter (char ch)
{
switch (ch)
{
case 'b': return BASE_REGS;
case 'x': return CALLEE_SAVED_REGS;
case 'y': return CALLER_SAVED_REGS;
case 'z': return ZERO_REGS;
}
return NO_REGS;
}
```

Get enumeration from register class letter

[Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

The <target.md> file

The target description in RTL is found in <target>**.md**. The following is an example of a macro taken from SPIM.c



```
;; Here z is the constraint character defined in
;; REG_CLASS_FROM_LETTER_P
;; The register $zero is used here.
(define_insn "IITB_move_zero"
[(set
(match_operand:SI 0 "nonimmediate_operand" "=r,m")
(match_operand:SI 1 "zero_register_operand" " z ,z")
)]
""
"@
move \t%0,%1
sw \t%1, %m0"
)
```

The Register Class letter code

[Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

4.15.6 The GCC Phase Sequence

The machine description is used at three levels in the generated compiler:

• To generate RTL code from gimple.

• To perform RTL to RTL transformations, during optimization passes.

• To emit assembly code corresponding to matched RTL pattern.

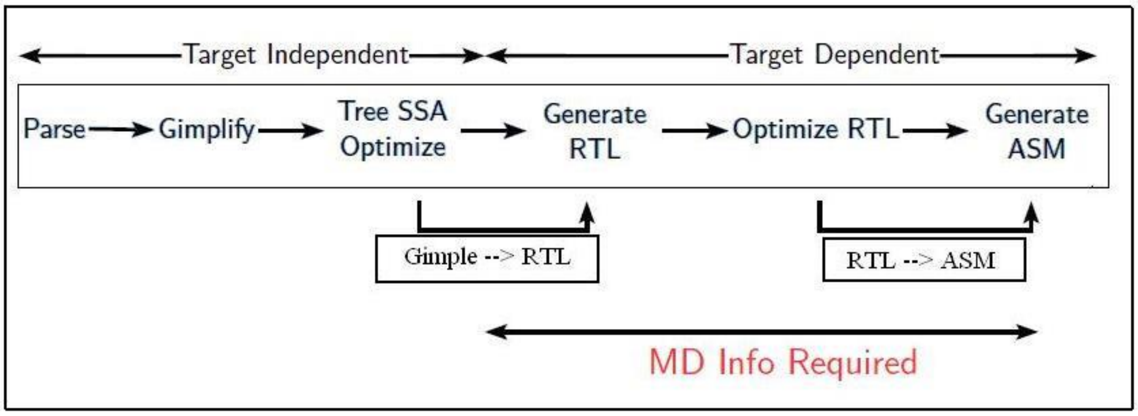
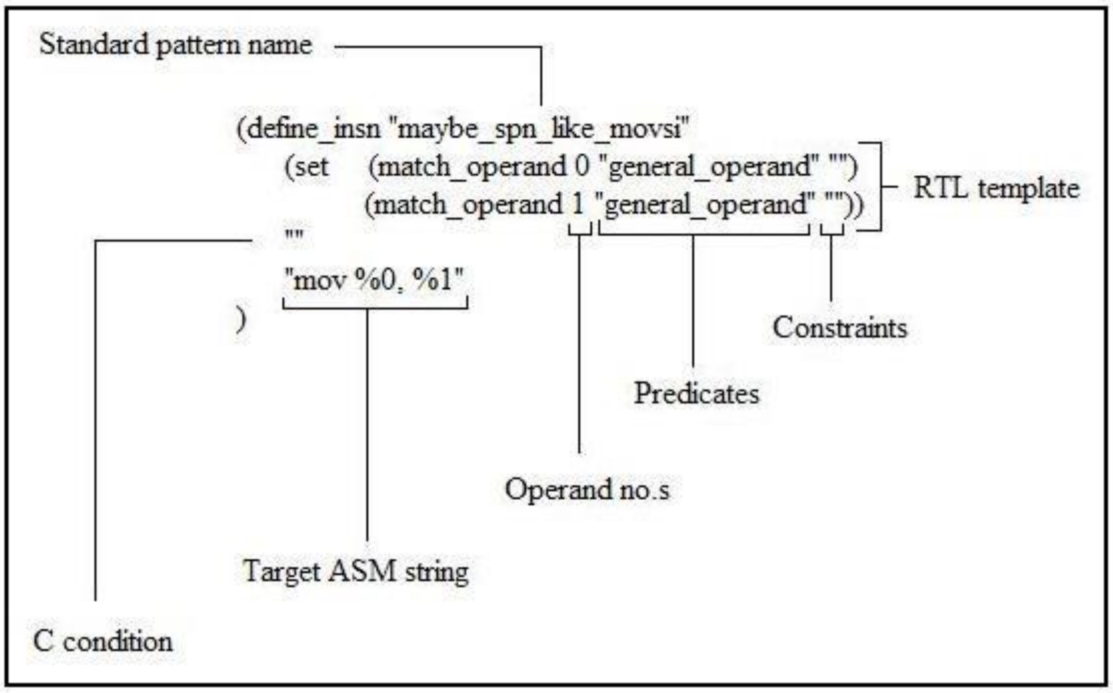


Figure 3 : GCC Phase Sequence[Ref: http://gcc.gnu.org]

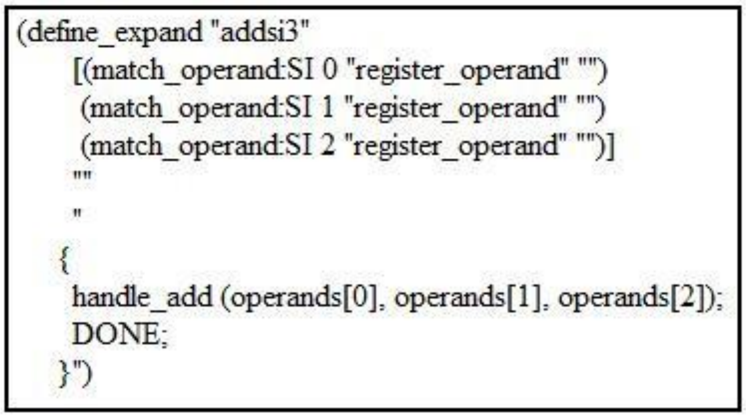
For the generate pass, only the names of the insn matter, from either a named define_insn or a define_expand. The compiler will choose the pattern with the right name and apply the operands according to the documentation later in this chapter, without regard for the RTL template or operand constraints. Note that the names the compiler looks for are hard-coded, it will ignore unnamed patterns and patterns with names it doesn't know about, but if you don't provide a named pattern it needs, it will abort. If a define_insn is used, the template given is inserted into the insn list. If a define_expand is used, one of three things happens, based on the condition logic. The condition logic may manually create new insns for the insn list, via emit_insn(), and invoke DONE. For certain named patterns, it may invoke FAIL to tell the compiler to use an alternate way of performing that task. If it invokes neither DONE nor FAIL, the template given in the pattern is inserted, as if the define_expand were a define_insn. Once the insn list is generated, various optimization passes convert, replace, and rearrange the insns in the insn list. This is where the define_split and define_peephole patterns get used.

Finally, the insn list's RTL is matched up with the RTL templates in the define_insn patterns, and those patterns are used to emit the final assembly code. For this purpose, each named define_insn acts like it is unnamed, since the names are ignored.



```
(define_insn "maybe_spn_like_movsi"
    (set    (match_operand 0 "general_operand" "")
            (match_operand 1 "general_operand" ""))
    ""
    "mov %0, %1"
)
```

Fig. : The define_insn construct [Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]



```
(define_expand "addsi3"
      [(match_operand:SI 0 "register_operand" "")
       (match_operand:SI 1 "register_operand" "")
       (match_operand:SI 2 "register_operand" "")]
      ""
      "
    {
      handle_add (operands[0], operands[1], operands[2]);
      DONE;
    }")
```

Fig.: The define_expand construct [Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

On some target machines, some standard pattern names for RTL generation cannot be handled with single insn, but a sequence of RTL insns can represent them. For these target machines, you can write a define_expand to specify how to generate the sequence of RTL.

A define_expand is an RTL expression that looks almost like a define_insn; but, unlike the latter, a define_expand is used only for RTL generation and it can produce more than one RTL insn. Every RTL insn emitted by a define_expand must match some define_insn in the machine description. Otherwise, the compiler will crash when trying to generate code for the insn or trying to optimize it. The RTL template, in addition to controlling generation of RTL insns, also describes the operands that need to be specified when this pattern is used. In particular, it gives a predicate for each operand.

4.16 GCC INSTALLATION

4.16.1 GCC-4.4.2

The first and foremost task of the project was to install the open source software GCC. It is used to build and install a cross compiler for a specific target, which for our project is SPIM. Once the systematic construction of the machine descriptions was completed, the different levels of SPIM were added to GCC source and then a cross compiler was built specific to the SPIM level. However before this process was started, GCC compiler had to be downloaded and installed.

4.16.2 GCC 4.4.2 INSTALLATION

• **Source:** We downloaded GCC from gcc.gnu.org. An alternative source is the GNU FTP server and its mirror sites.

• **Installation:** The following is the procedure to install GCC.

• Create directories build and install in a tree not rooted at
$GCCHOME.

• $GCCHOME/configure--prefix=$ABSOLUTEPATH/install--
target=SPIM<n> --enable-languages=c --no-libgcc

• $make

• $make install

4.17 Cross Compilation for SPIM Levels

4.17.1 Incremental Machine Description

The GCC framework generates a compiler for a given architecture by reading the machine description for it. In practice, rather than developing a new port from scratch, we develop it using an existing machine description for the architecture of a machine that is

close to the new target. Another issue is that the machine descriptions may be ad hoc. As a consequence, the machine descriptions become difficult to construct, understand, maintain and enhance because of the verbosity, the amount of details, and the repetitiveness. The publicly available material fails to bring out the exact abstractions captured by the machine descriptions. There is no systematic way of constructing machine descriptions and there are no clear guidelines on where to begin developing machine description and how to construct them systematically.

Initially a compiler must be built for a small subset of C language, and then gradually the scope of language supported by the port, must be increased until all the language features are supported. The advanced target features can be added on top of the machine descriptions incrementally. For this purpose, we have presently divided this small subset of C language into five levels. The purpose behind using incremental strategy in building compiler a is

• The information required to build a complete port for a target machine is huge, and managing it at a stretch becomes difficult. Divide and conquer strategy always helps in managing large tasks. So, the machine descriptions are systematically divided, which enables us to concentrate on a smaller part at a time.

• To know the information that is absolutely necessary to support some higher level language constructs for specific target machine. Because of incremental structure, each construct can be handled and observed carefully against variations in the target code.

• To extract underlying abstract machine assumed by GCC.

4.17.2 Detailed Description of Levels

We first identify the minimal machine description, as the specification of target architecture features that are absolutely necessary to build the compiler. The compiler built might not compile programs, but the executable xgcc is generated. It is minimal because no redundant or extra information is provided to GCC in this level and even if a single macro definition or RTL pattern is removed from the description, the compiler fails to build. We call this minimal machine description as Level 0 machine description.

Level 1: Assignment statements involving on integer constants

Level 1 supports assignment operation involving integer constants or integer variables. Level 1 machine descriptions are built on top of level 0.2 machine description; hence no macro is required to be added in level 1.

Level 2: Arithmetic operations on integer data type.

Level 2 of machine description covers arithmetic and bitwise operations in the source language. The macro definitions remain the same as in level 1. New RTL patterns are added in .md file corresponding to additional operations supported in this level. The register class information is modified in order to let the compiler know about internal registers of SPIM.

Level 3: Function handling and calling conventions.

Level 3 of machine description adds function handling and calling conventions. No new macro is defined in this level. New instructions are added in the .md file corresponding to instructions added in this level. The activation record definition remains same as in level 0.2.

Level 4: Control structures.

Level 4 includes conditional control transfers and control structures in higher level languages. The looping constructs like while, for, etc. can be transformed into a simple structure which is a combination of sequential operational instructions, branch and goto statements. Hence, the only high level language construct that is additionally supported in this level is if-then-else.

The objective is to further optimize each level. This can be done by finding a way of selecting an efficient combination of instructions and add them to the machine descriptions in systematic way. This will ultimately reduce the complexity. However before this the remaining features to support floating point instructions must be added.

4.17.3 Cross Compilation Process

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. The SPIM machine description is built into 4 levels. We built a cross compiler for each level and ran test cases for each. After making the 5th level to incorporate additional features, we repeated the same process.

4.17.4 Steps to Build Cross Compiler for each Level of SPIM

1. Create directories BUILD and INSTALL outside $GCCHOME directory.
2. Go to BUILD directory and execute the following configure command:

```
[gcc_project@localhost]$ $GCCHOME/configure
--prefix=$ABSOLUTEPATH/install --target=SPIM<n> --enable -languages=c --no-
libgcc
```

• $GCCHOME is the home directory of GCC. Here, it is gcc- 4.0.2.

• --prefix is used to specify absolute path of INSTALL directory, which is given by $ABSOLUTEPATH
(e.g./home/gcc_project/install).

• --target option is used to specify the target machine. 'n' in SPIM<n> corresponds to the level of SPIM for which we intend to build the cross compiler (e.g. SPIM0, SPIM2….
SPIM5).

• --enable-languages specifies that the cross compiler is being built only for the source language C.

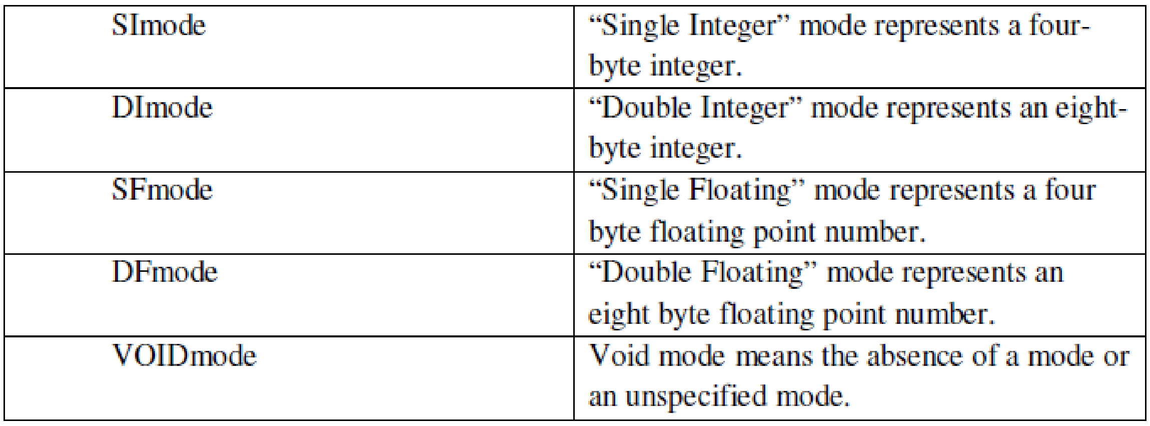
• --no-libgcc option is used to disable the local library of GCC.

3. Execute make command [gcc_project@localhost]$ make
4. Execute make install command [gcc_project@localhost]$ make install

Initially we had used xgcc to run the programs which gave the output in the form executable file. However, using xgcc was error prone. The reason that we figured out was
that GCC could not find xgcc in any of the PATHs. Later on, we found that xgcc resides
in the folder build/gcc. So we could run .c files successfully. xgcc generated an executable file i.e. a.out file. However, we needed to generate .s file from .c file in order to run .s file in SPIM simulator. So, instead of using xgcc we used cc1.

4.18 MACHINE MODES

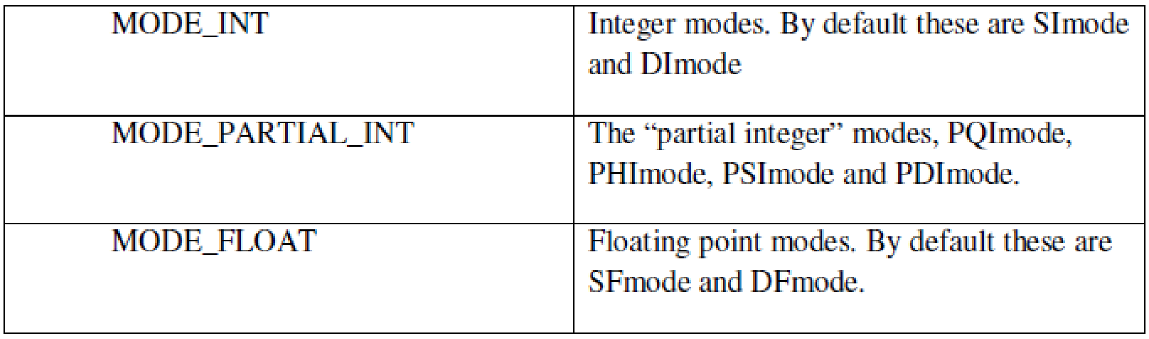
A machine mode describes a size of data object and the representation used for it. In the C code, machine modes are represented by an enumeration type, enum machine_mode, defined in machmode.def. Each RTL expression has room for a machine mode and so do certain kinds of tree expressions (declarations and types, to be precise). In debugging dumps and machine descriptions, the machine mode of a RTL expression is written after the expression code with a colon to separate them. The word `mode' which appears at the end of each machine mode name is optional. For example (reg: SI 38) is a reg expression with machine mode SImode.



SImode	"Single Integer" mode represents a four-byte integer.
DImode	"Double Integer" mode represents an eight-byte integer.
SFmode	"Single Floating" mode represents a four byte floating point number.
DFmode	"Double Floating" mode represents an eight byte floating point number.
VOIDmode	Void mode means the absence of a mode or an unspecified mode.

Table : Machine Modes[Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

The explicit references to machine modes that remain in the compiler will soon be removed. Instead, the machine modes are divided into mode classes. These are represented by the enumeration type enum mode_class defined in machmode.h. The possible mode classes are:



MODE_INT	Integer modes. By default these are SImode and DImode
MODE_PARTIAL_INT	The "partial integer" modes, PQImode, PHImode, PSImode and PDImode.
MODE_FLOAT	Floating point modes. By default these are SFmode and DFmode.

Table No: Mode Classes [Ref: Workshop on Essential Abstractions in GCC, GRC, IITB]

4.18.1 Standard Pattern Names

Following are some of the standard pattern names that are used in machine descriptions. The same patterns are written for all the modes, some of which are described above. In each of the pattern given below, *m* stands for a two-letter machine mode name, in lowercase. Whenever the pattern is written, m is replaced by mode i.e. SI, SF, DF, etc. There are many such patterns, each having a different functionality. Only few are described below.

• mov*m*

This instruction pattern moves data with 'm' machine mode from operand 1 to operand 0. For example, `movsi' moves full-word data. If operand 0 is a subreg with mode *m* of a register whose own mode is wider than *m*, the effect of this instruction is to store the specified value in the part of the register that corresponds to mode *m*. Bits outside of *m*, but which are within the same target word as the subreg are undefined. Bits which are outside the target word are left unchanged.

• abs*m*2

Store the absolute value of operand 1 into operand 0.

• neg*m*2

Negate operand 1 and store the result in operand 0.

• cmp*m*

Compare operand 0 and operand 1, and set the condition codes. The RTL pattern should look like this:

(set (cc0) (compare (match_operand:*m* 0 ...)

(match_operand: *m* 1 ...)))

• call_value

Subroutine calls instruction returning a value. Operand 0 is the hard register in which the value is returned. There are three more operands, the same as the three operands of the `call' instruction (but with numbers increased by one).

• add*m*3

Add operand 2 and operand 1, storing the result in operand 0. All operands must have mode *m*.

• sub*m*3

Subtract operand 2 from operand 1, storing the result in operand 0. All operands must have mode *m*.

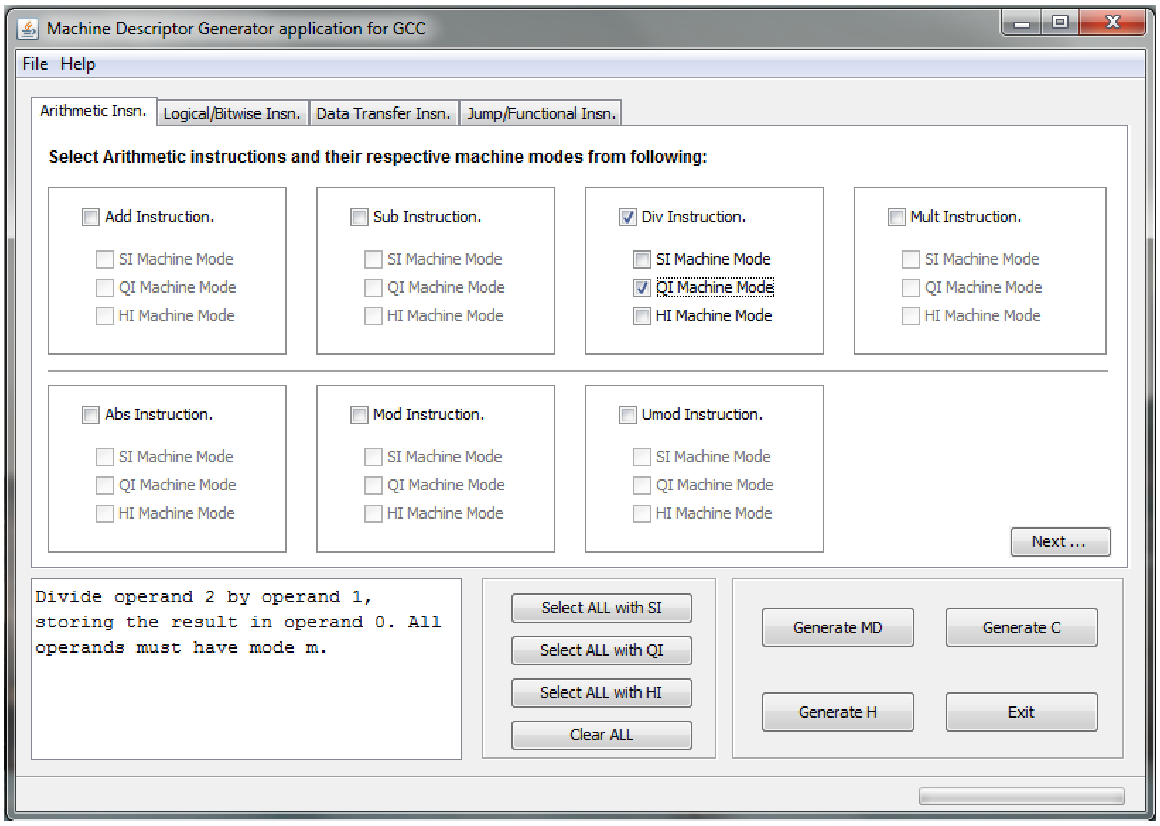
• mul*m*3

Multiply operand 1 by operand 2, storing the result in operand 0. All operands must have mode *m*.

• sub*m*3

Divide operand 1 by operand 2, storing the result in operand 0. All operands must have mode *m*.

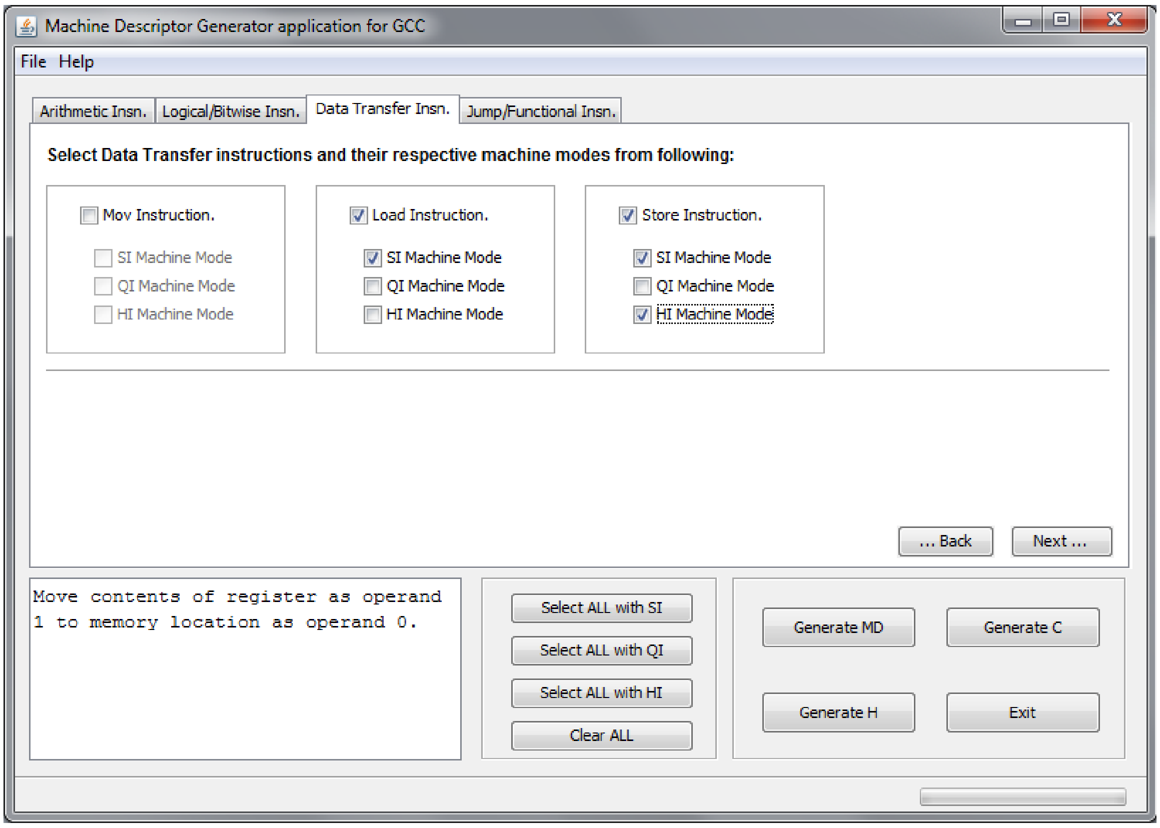
4.19 SNAPSHOTS



Div Instruction: ✓; QI Machine Mode: ✓

```
Divide operand 2 by operand 1,
storing the result in operand 0. All
operands must have mode m.
```

Figure 4 : Action: Adding Modes to Arithmetic Insn

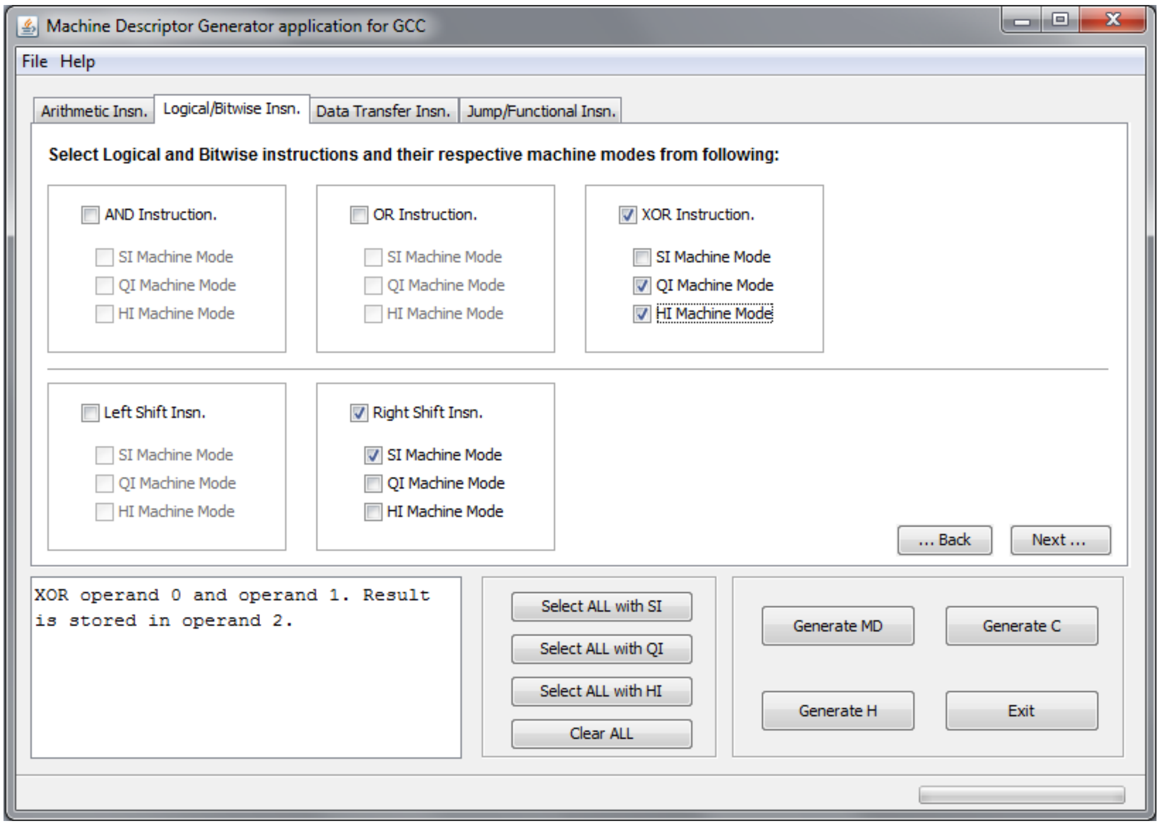


Select Data Transfer instructions and their respective machine modes from following:

- [ ] Mov Instruction.
  - [ ] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] Load Instruction.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] Store Instruction.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [✓] HI Machine Mode

```
Move contents of register as operand
1 to memory location as operand 0.
```

Figure 5 : Action: Adding modes to Data Transfer Insn

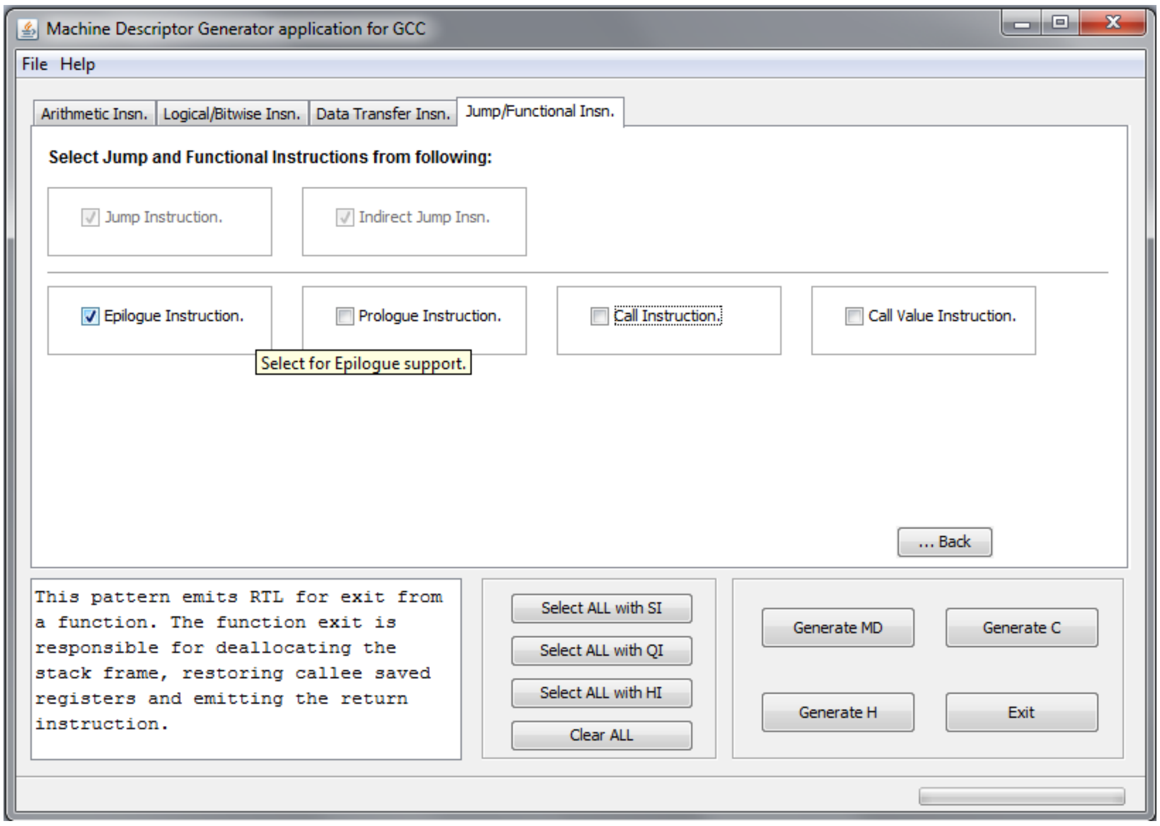


Select Logical and Bitwise instructions and their respective machine modes from following:

- [ ] AND Instruction.
  - [ ] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [ ] OR Instruction.
  - [ ] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] XOR Instruction.
  - [ ] SI Machine Mode
  - [✓] QI Machine Mode
  - [✓] HI Machine Mode
- [ ] Left Shift Insn.
  - [ ] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] Right Shift Insn.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode

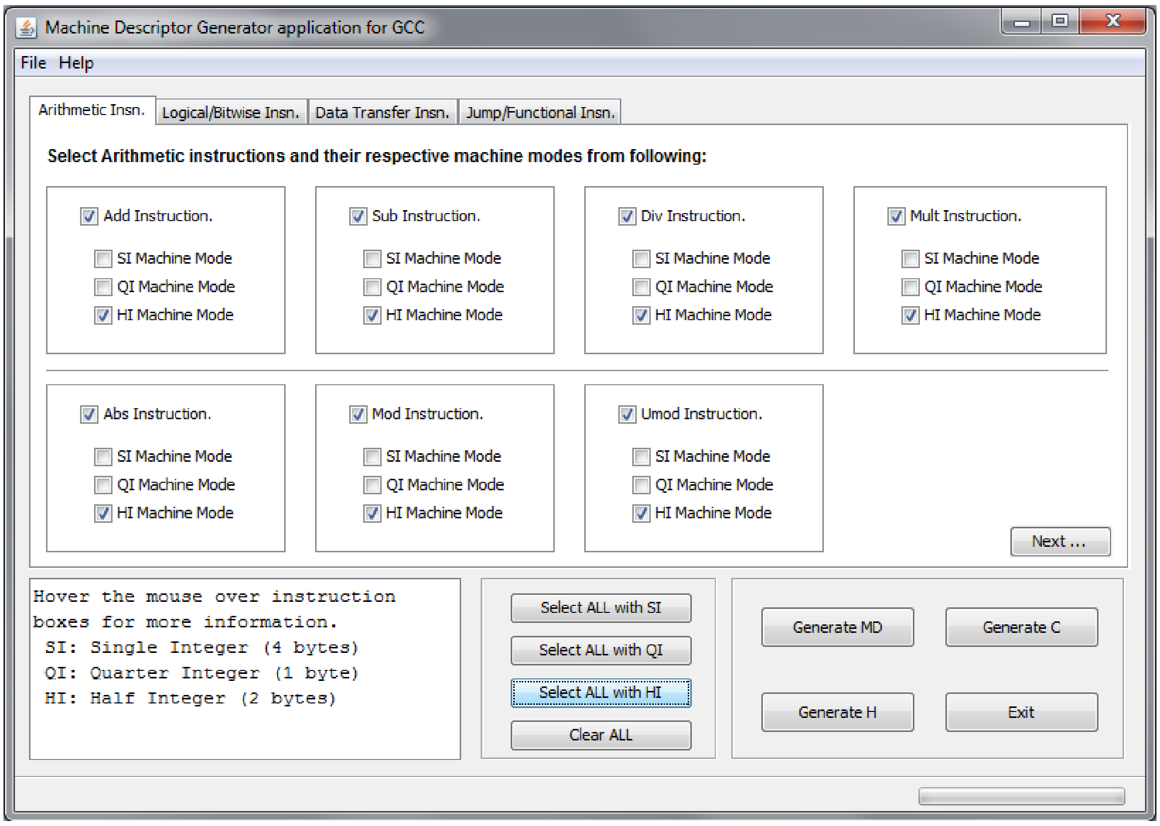
```
XOR operand 0 and operand 1. Result
is stored in operand 2.
```

Figure 6 : Action: Adding modes to logical Insn



```
This pattern emits RTL for exit from
a function. The function exit is
responsible for deallocating the
stack frame, restoring callee saved
registers and emitting the return
instruction.
```

Figure 7 : Action: Adding modes to Jump Insns

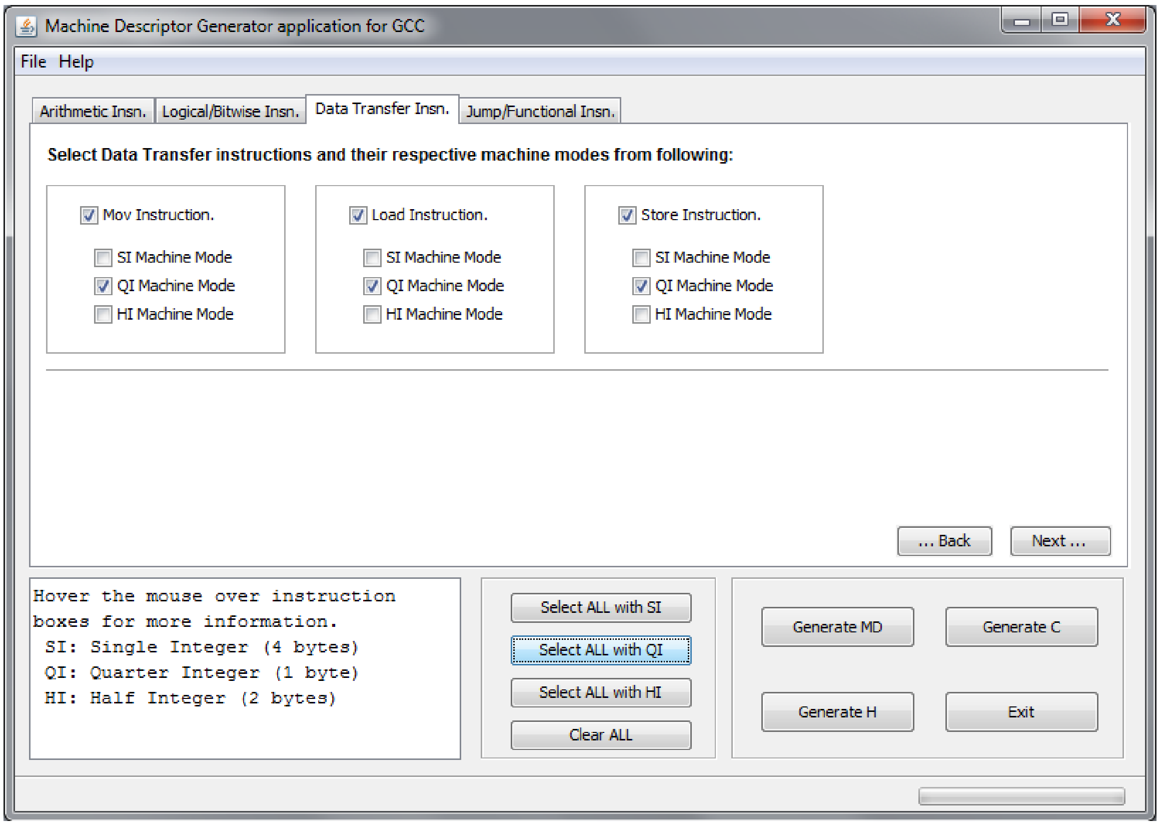


Select Arithmetic instructions and their respective machine modes from following:

✓ Add Instruction.	✓ Sub Instruction.	✓ Div Instruction.	✓ Mult Instruction.
SI Machine Mode	SI Machine Mode	SI Machine Mode	SI Machine Mode
QI Machine Mode	QI Machine Mode	QI Machine Mode	QI Machine Mode
✓ HI Machine Mode	✓ HI Machine Mode	✓ HI Machine Mode	✓ HI Machine Mode
✓ Abs Instruction.	✓ Mod Instruction.	✓ Umod Instruction.	
SI Machine Mode	SI Machine Mode	SI Machine Mode	
QI Machine Mode	QI Machine Mode	QI Machine Mode	
✓ HI Machine Mode	✓ HI Machine Mode	✓ HI Machine Mode	

```
Hover the mouse over instruction
boxes for more information.
 SI: Single Integer (4 bytes)
 QI: Quarter Integer (1 byte)
 HI: Half Integer (2 bytes)
```

Figure 8 : Action: Selecting all with HI



Select Data Transfer instructions and their respective machine modes from following:

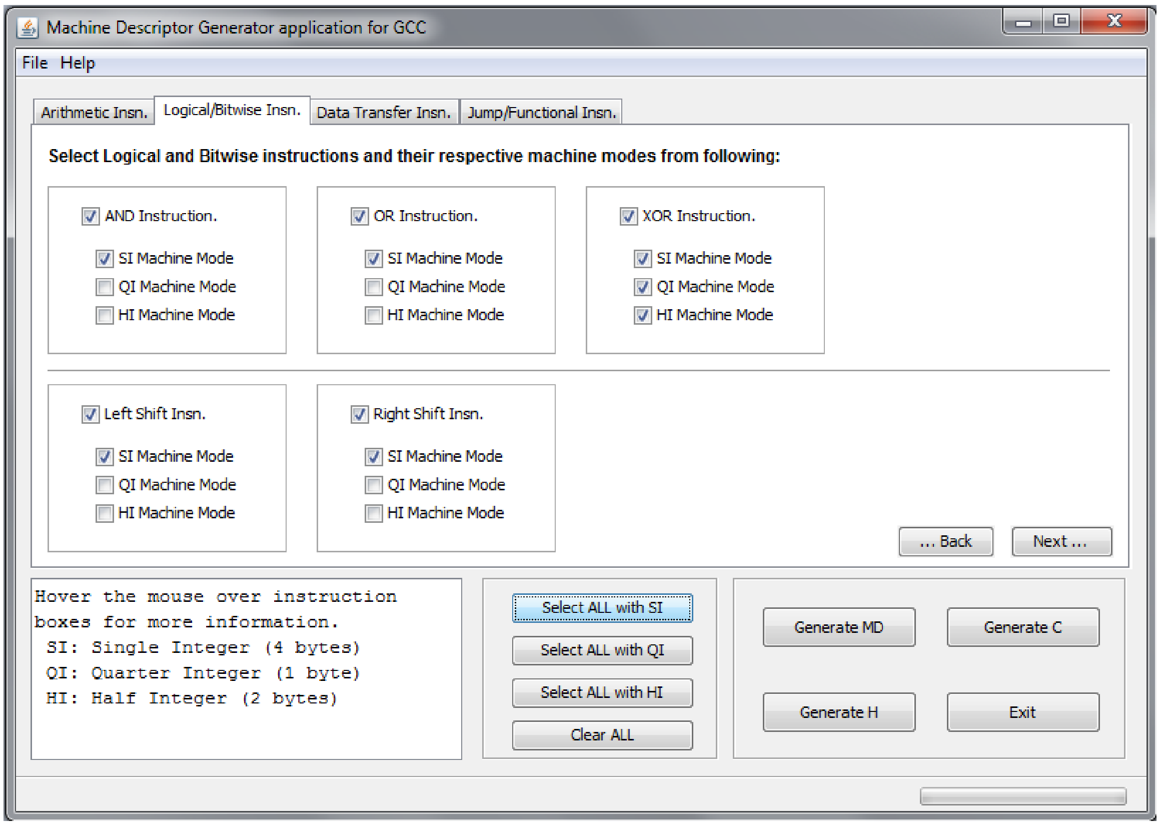
[✓] Mov Instruction.
[ ] SI Machine Mode
[✓] QI Machine Mode
[ ] HI Machine Mode

[✓] Load Instruction.
[ ] SI Machine Mode
[✓] QI Machine Mode
[ ] HI Machine Mode

[✓] Store Instruction.
[ ] SI Machine Mode
[✓] QI Machine Mode
[ ] HI Machine Mode

```
Hover the mouse over instruction
boxes for more information.
 SI: Single Integer (4 bytes)
 QI: Quarter Integer (1 byte)
 HI: Half Integer (2 bytes)
```

Figure 9 : Action: Selecting all with QI

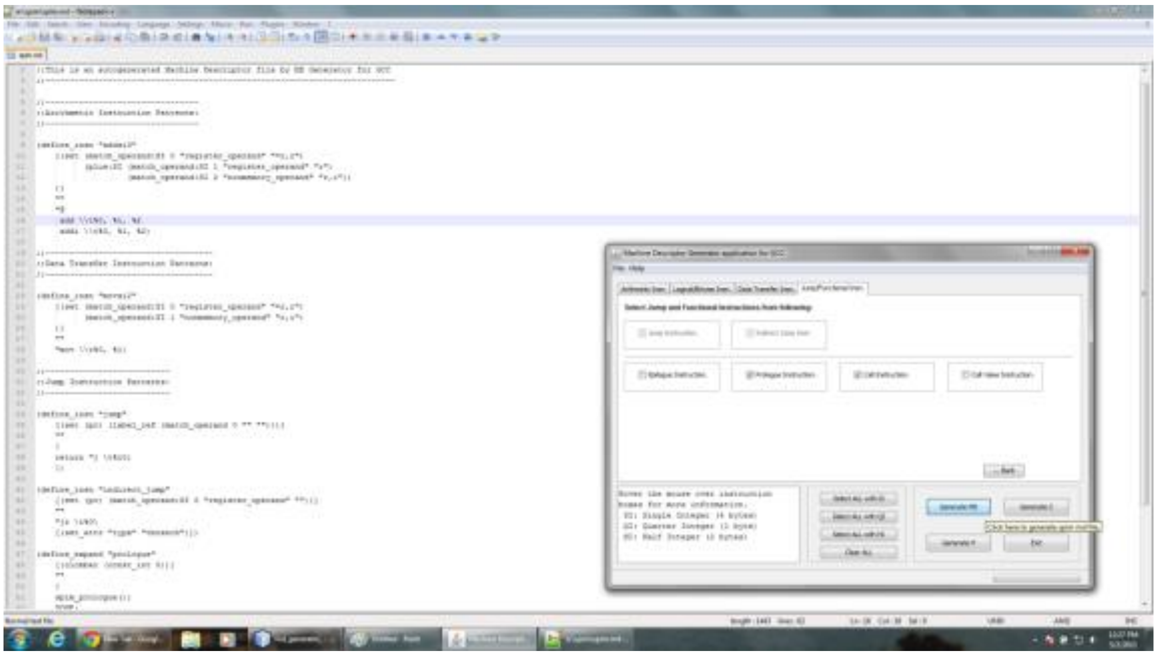


Select Logical and Bitwise instructions and their respective machine modes from following:

- [✓] AND Instruction.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] OR Instruction.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] XOR Instruction.
  - [✓] SI Machine Mode
  - [✓] QI Machine Mode
  - [✓] HI Machine Mode
- [✓] Left Shift Insn.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode
- [✓] Right Shift Insn.
  - [✓] SI Machine Mode
  - [ ] QI Machine Mode
  - [ ] HI Machine Mode

```
Hover the mouse over instruction
boxes for more information.
 SI: Single Integer (4 bytes)
 QI: Quarter Integer (1 byte)
 HI: Half Integer (2 bytes)
```

Figure 10 : Action: Selecting all with SI


Machine Descriptor Generator application for GCC
File Help
Arithmetic Insn. | Logical/Bitwise Insn. | Data Transfer Insn. | Jump/Functional Insn.
Select Jump and Functional Instructions from following:
Jump Instruction
Indirect Jump Insn
Epilogue Instruction
Prologue Instruction
Call Instruction
Call Value Instruction
Back
Select ALL with SI
Select ALL with QI
Select ALL with HI
Clear ALL
Generate MD
Generate C
Generate H
Exit
Click here to generate spim.md file

```
;;This is an autogenerated Machine Descriptor file by MD Generator for GCC
;;--------------------------------------------------------------------------


;;----------------------------------
;;Arithmetic Instruction Patterns:
;;----------------------------------

(define_insn "addsi3"
    [(set (match_operand:SI 0 "register_operand" "=r,r")
          (plus:SI (match_operand:SI 1 "register_operand" "r")
                   (match_operand:SI 2 "nonmemory_operand" "r,i")))
    ]
    ""
    "@
     add \\t%0, %1, %2
     addi \\t%0, %1, %2)

;;------------------------------------
;;Data Transfer Instruction Patterns:
;;------------------------------------

(define_insn "movsi2"
    [(set (match_operand:SI 0 "register_operand" "=r,r")
          (match_operand:SI 1 "nonmemory_operand" "r,i"))
    ]
    ""
    "mov \\t%0, %1)

;;-----------------------------
;;Jump Instruction Patterns:
;;-----------------------------

(define_insn "jump"
    [(set (pc) (label_ref (match_operand 0 "" "")))]
    ""
    {
    return "j \\t%l0";
    })

(define_insn "indirect_jump"
    [(set (pc) (match_operand:SI 0 "register_operand" ""))]
    ""
    "jr \\t%0"
    [(set_attr "type" "unknown")])

(define_expand "prologue"
    [(clobber (const_int 0))]
    ""
    {
    spim_prologue();
    DONE;
```

```
Hover the mouse over instruction
boxes for more information.
 SI: Single Integer (4 bytes)
 QI: Quarter Integer (1 byte)
 HI: Half Integer (2 bytes)
```

Figure 11 : Action: Generating .MD file

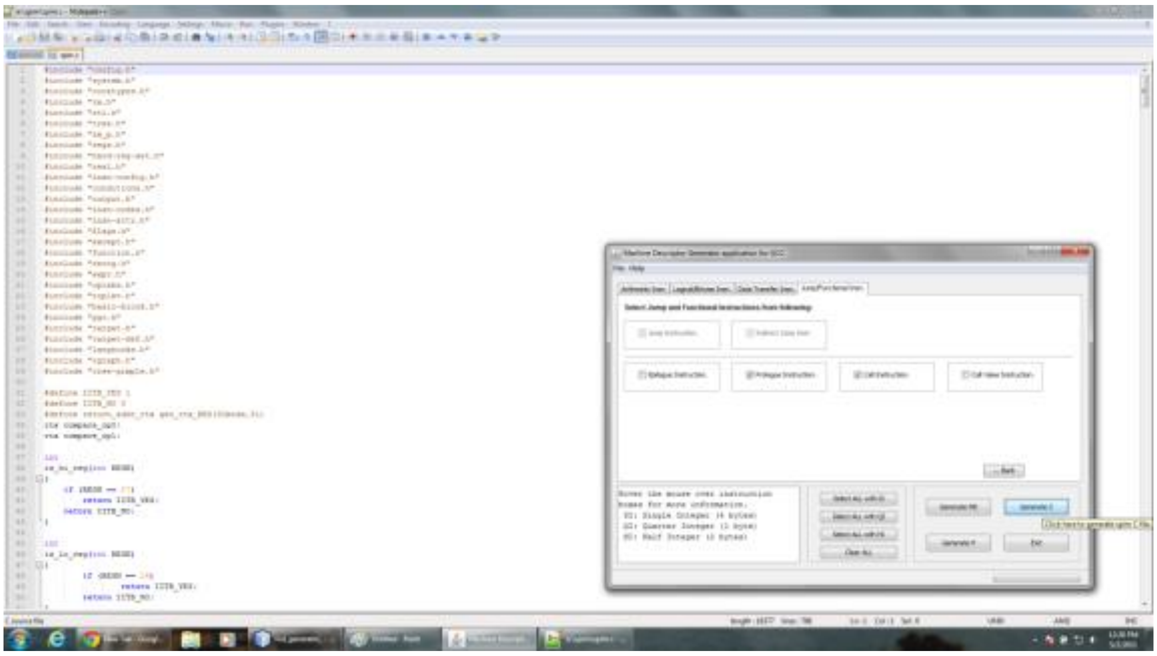


Figure 12 : Action: Generating .C file

5. Conclusion

5.1 Conclusion

The files required to cross compile GCC i.e. .md, .c, .h files are being automatically generated so as to provide limited functionality to a custom made architecture. Currently, we have simulated custom made architecture by SPIM which is a custom version of MIPS architecture which can further be expanded. Small modifications in the code will allow it to support any custom made architecture.

6. References

Bibliography

1) Work done on incremental machine descriptions by Prof. Uday P. Khedker, Indian Institute of Technology, Bombay
2) GCC Internals : http://gcc.gnu.org/
3) www.cse.iitb.ac.in/grc/
4) Work done previous project group in incremental spim levels.
5) Workshop on GCC internals by GCC Recourse Center, Dept of Computer Science and Engineering, IIT Bombay.
6) www.redhat.com