AN ABSTRACT OF THE THESIS OF

Savithri H. Venkatachalapathy for the degree of
Master of Science in
Electrical and Computer Engineering presented
on April 9, 2004.
Title: Porting GCC to X32V Architecture.

Abstract approved: Signature redacted for privacy.

Ben Lee

This thesis work evaluates the need for a re-configurable cross compiler for the X32V processor architecture and discusses the process of developing a cross compiler for X32V. X32V is a new processor intended at the embedded applications domain whose instruction set is designed based on the widely used MIPS processor. A Cross compiler for X32V was required for performing statistical analysis of the Instruction set and facilitates further profiling and recommendations for the architecture. GCC was chosen as the base compiler as it is a stable and reliable compilers with front-end support for multiple languages and back-end support for most of the architectures, contemporary and older as well. Porting GCC to X32V involved identifying architecture similar to X32V and obtaining the port for the same. Further, relevant modifications are made to the GCC back-end including the machine descriptions, the calling conventions and macros. Porting process requires detailed understanding of the target architecture mainly its instruction set, register file and define the ABI (Application Binary Interface) for the target. ABI is the method used to pass arguments, register usage conventions and layout of stack frame allocation. The coding is done in RTL (Register-transfer-language), which is a lisp like language and has its own in-built expressions and constructs. Also the floating point operations are supported using off-the-shelf specific macro libraries. The concept of re-configurability was kept in mind while developing a working X32V cross compiler. New instructions when

required can be added with ease by editing the machine descriptions. This compiler produces assembly code for X32V which is further assembled to produce binaries using a third party assembler. The binaries are run on a simulator designed for this purpose. Thus we have set up basic infrastructure (cross compiler, simulator, and assembler) which would help in continuing this computer architectural research to new heights.

Porting GCC to X32V Architecture

by

Savithri H. Venkatachalapathy

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented April 9, 2004

Commencement June 2004

ACKNOWLEDGEMENTS

The author expresses sincere appreciation and regards to Dr. Ben Lee for the timely advice and guidance provided during this research work. Regarded highly is Young Soo Kim from ETRI for providing the necessary resources and promptly responding to any changes requested. Also thanking Dr. Wen-Tsong Shiue, Dr. Bella Bose and Dr. Mike J. Pavol for consenting to be on the committee. And acknowledge the consistent support of the team members David Zier, Jumnit Hong and Jerrard Nelson, whose help plays a vital role in the realization of this thesis work.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

Porting GCC to X32V Architecture

1 Introduction

In collaboration with the Electronics and Telecommunications Research Institute of Korea, we developed an embedded processor core named X32V. The design principles mainly focused on extensibility and reconfiguration. After having the Instruction Set Architecture in place, we developed a simulator which simulates the functional behavior of X32V. It is a functional simulator and not cycle accurate. The simulator is based on the PISA Simple scalar version, with deliberate and relevant modifications. Noting that X32V architecture is similar to MIPS, we wrote a script which would translate the MIPS assembly to X32V assembly, using the one-to-one correspondence of the instructions of MIPS and X32V. But as there are certain instructions of the X32V ISA that are not compatible with MIPS, we were not able to successfully port those instructions. Owing to these disparities, we had to restrict ourselves to simple programs which were, otherwise, sufficient to test the functionality of the simulator. This was proving to be a handicap for further research towards ISA exploration, ISA comparison, and performance ratio.

Thus, these discrepancies led us to develop a C cross compiler, which generates the assembly code for the X32V processor. This thesis work focuses on achieving the goal of developing a working cross compiler for X32V running on an Intel Pentium III, with a Linux Operating System. This C compiler translates from C language into assembler code for X32V, which is further assembled by assembler to produce binaries for X32V. Cross compilers are also important for design space exploration in cases where the target processor architecture is not stable in advance.

We have chosen to port GCC as it is one of the most widely used compilers. It is freely available and has back-end support for most of the contemporary machines along with front-end support for several languages like C, C++, Java, etc. The porting process involved identifying a machine with a similar instruction set to that of X32V and obtaining the GCC compiler for that machine. And then relevant changes then had to be made to the GCC back-end.

This report is organized into chapters which discuss the compiler fundamentals and X32V port. Chapter 2 is an introduction to the fundamentals of Compilers, the general porting process and introduction to the X32V port. Chapter 3 discusses the X32V architecture, the instruction set and the simulation environment before we start discussing the porting process in detail. Chapter 4 is all about the GCC compiler system, different phases of the compiler and the back-end structure. Chapter 5 talks in length about the register-transfer language constructs and the machine descriptions, while chapter 6 deals with the porting process, specific to the X32V architecture. It focuses on the X32V Application Binary Interface, the problems encountered while porting GCC, the existing limitations and the steps to build the compiler. In chapter 7, we talk about the test benchmarks we ran on the X32V cross compiler and its performance. This is followed by conclusion remarks and a documented list of references. Also provided is an appendix A for the X32V Instruction Set and appendix B for X32V assembly output.

2 Compiler concepts

A compiler is a program that reads a program written in one language (source language) and translates it into a program in another language (target language). The source languages could be FORTRAN, Pascal, C, C++, ADA or any other programming language. Similarly the target language may be another programming language, machine language for a particular processor, or an intermediate assembly language which has instructions represented as mnemonics.

Normally the compiler back-end is specific for the target CPU and system on which the compiled code will run. But most parts of the compiler code are not target machine dependent, like the front-end which can be re-used for developing compilers for various targets. Writing a compiler from scratch for a new target is a humungous and time consuming task, involving limitations and is usually error-prone. Instead, source code for compiler like GCC which is available freely could be reused code and rewritten for new targets. This is the approach we are taking and porting C compiler of GCC suite to X32V target.

Many times, while developing new processor architectures, the performance evaluation can be effectively done if there exists a way to simulate the behavior of the machine. That is, having binaries instructions for that architecture as early as the conception of the processor's instruction set. This can be achieved through cross compilers.

A cross compiler is a compiler which runs on one platform and produces code for another, as opposed to a native code compiler which produces code for the platform on which it runs. GCC can work as a cross compiler if correctly set up. We are harnessing this feature of GCC in order to port it to X32V.

2.1 Porting GCC

We have chosen GCC as the base compiler and ported it to X32V. Porting GCC involves two phases. In the first phase, it is configured for the target machine with support for specific front end language mentioned. In the phase two it is built with the help of make files.

Porting GCC is a major undertaking. GCC is made of multiple front-ends, each of them being used for a different language. The front-ends are completely independent of the machine they are generating the code for. Porting GCC as a cross compiler to a new target means adding code in the configuration of GCC so that it knows about the architecture of the new processor (for optimizations) and also about the target assembler syntax. The porting mechanism is outlined in the subsequent chapters.

2.2 X32V Port

In our research we are developing a cross compiler by writing the back-end for X32V architecture. It is built on Intel processor and operating system used is Linux. It emits the assembly code for X32V, whose architecture is described in chapter 3. Our study focuses mostly on the compiler, while the assembler is provided by a third party. At this moment we are restricting ourselves to the 32-bit instruction set of X32V. Also, floating point instructions are not supported by the back-end implementation, but by the fp-bit library which is an off-the-shelf commercially available library for floating point operations. The runtime support is provided by the newlib library. For testing the port, we have chosen a few test benches. And despite certain limitations, we have successfully compiled those test benches and the results are discussed in a later chapter.

3 X32V Architecture

X32V architecture was designed keeping in mind embedded applications. X32V is a 32-bit Big-Endian, RISC based microprocessor. It is built around a 32-bit data-path, five-stage pipeline. The core ISA is based on 32-bit instructions; however, it has support for lighter instruction format modes (32/16, and 32/24/16). It is sculptured to be extensible with provisions for inclusion of floating-point, multimedia and DSP instructions.

For the current research, we are restricting ourselves to the 32 bit ISA, which is mentioned below. All the 32 bit instructions are defined in the back-end of the port.

3.1 Registers

X32V has 36 registers, each 32bits wide and are categorized as: General Purpose, Floating Point, and Reserved Usage.

There are 16, 32-bit, General Purpose Registers (R0 – R15). None of these GPRs have reserved usage. Registers can be loaded with byte (8 bits), half-word (16 bits), or word (32 bits) values. Byte values are stored in byte 0. Half-word values are stored in byte 1 and byte 0. Word values use all four bytes. X32V ISA also has 16, 32-bit, Floating Point Registers.

Like many other RISC machines, X32V has several registers reserved for special purposes: Program Counter, Link Register, Stack Pointer, and Status Register. To be noted is that X32V has a designated Stack Pointer, which point to the top of the

stack area. Also, the Status Register serves the purpose of Condition Code Register (cc0).

3.2 Addressing Modes

X32V has load/store architecture. It only allows loads and stores to access data memory. As such, only immediate and displacement addressing modes are supported. Immediate addressing mode allows a 16-bit immediate value as an operand. Displacement addressing uses a GPR in conjunction with a signed immediate value to provide the effective address. The different addressing modes that can be deduced from these are explained in the appendix A.

3.3 Instruction Set

As mentioned earlier X32V supports signed and unsigned, 32-bit, 24-bit and 16-bit instruction formats, even though we are currently working on the default mode of 32-bit instructions. The instructions can be brought under the dimensions of their purposes and categorized as: Load/Store, Immediate, Branch, Register, and Jump/Call. The instruction set architecture is given in detail in the appendix A.

Arithmetic and logical instructions have both register and immediate formats. The register format instructions have three operands with each of them being a general purpose register. The source registers are represented as rs1, rs2 while the destination register is rd. The immediate versions look the same, but with the second source operand being a 16-bit immediate value embedded into the instruction.

Branch instructions are architected with respect to the condition code register. There are two formats, with the former taking two register operands followed by a label, while the latter has one register operand and a label.

Move instructions which are categorized as load/ store, can have special registers as either source or destination. And the same applies to "add", "push" and "pop" instructions. The "syscal" instruction is used for trap to system calls. The run time system library used is newlib. Also the "nop" instruction which inserts a delay is defined.

3.4 Compilation Environment

We have set up infrastructure necessary for performing simulations and profiling of X32V. The minimum hardware requirements for the X32V compiler include:

- PC with LINUX based Operating System.
- Built with egcs version 1.1.2
- Script file to build and configure the compiler.

The assembly output generated by the compiler is assembled using the third party assembler or a specific perl file which reads the format of the assembly file and translates it into X32V machine instructions. The simulator takes in a binary X32V file and an optional configuration file. Execution of the simulator runs to completion upon entering zero at the prompt.

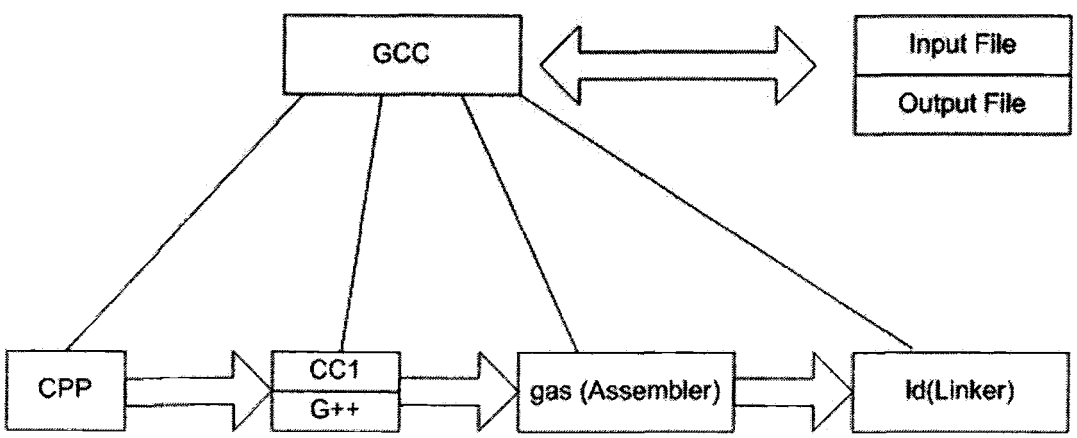
4 GNU Compiler System

For our purpose, we are using the GNU Cross Compiler (GCC) suite. GCC is re-targetable and it is available from the Free Software Foundation, Boston, Massachusetts. It is copyrighted, licensed and has front-end and back-end support for many popular processors. Also the code quality of GCC is better than some known compilers and is very close to DEC's compiler for Alpha. Some of the most stable and popular versions are: GCC 2.6.3, 2.7.2.3, egcs-1.1.2, GCC 2.95.3.

Front Ends supported are C, C++, Objective C, ADA, Java (supported from GCC 3.0), Pascal, Fortran 77, and FORTRAN 95. The back-end machines supported are Acorn, Advanced RISC Machine(ARM), Intel x86 Families, i860, i960, Motorola 680x0, 68C11, DSP56000, Hitachi SH, H8300, MIPS, IBM PowerPC, HP PA-RISC, SUN SPARC, Alpha (DEC) and new targets supported frequently.

The GCC system invokes different programs that are internally, while generating the binaries for a target. The execution model is shown in Figure 1.

Figure1: GCC System



Components shown in Figure 1 are as follows:

cpp: The C preprocessor. It takes care of preprocessor directives, such as include files and macro expansion. It also removes the comments. The result is a file with the C code, lots of white space and some line-numbering directives that the compiler core can use in warning and error messages. This program is a part of GCC.

cc1,g++: The compiler core, the actual compiler. It includes both the language front end and the target machine back end. This program is a part of GCC.

gas: The assembler.

ld: The linker.

These programs (gas and ld) are called by these names and perform the functions of assembler and linker/loader. They are invoked automatically unless overridden by certain specific options given below.

When GCC is used, it is commonly called as in one of the following examples:

The following command compiles and links a C program try.c to generate an executable:

GCC try.c

This command generates a file called 'a.out' which is an executable file.

The following only compiles the program file to generate an object file:

GCC -c try.c

This generates a file 'try.o' which is the object file corresponding to the program file 'try.c'. This object file can then be linked by the *c++* command to get the executable file 'a.out'.

The default executable produced by GCC is 'a.out'. But the executable name can be changed by using the *–o* flag with *c++* as shown below.

GCC try.c -o try.exe

This command produces try.exe as the executable file. More alternatives exist, and are described in next section.

4.1 Compiler Options

The above section discusses only a few relevant compiler options. Here we present a description of all the possible options that control the output, as mentioned by Richard Stallman [3].

For any given input file, the file name suffix determines what kind of compilation is done:
file.c - C source code which must be preprocessed.
file.i - C source code which should not be preprocessed.
file.ii - C++ source code which should not be preprocessed.
file.m - Objective-C source code. Note that you must link with the library 'libobjc.a' to make an Objective-C program work.
file.h - C header file (not to be compiled or linked).
file.cc, file.cxx, file.cpp, file.c - C++ source code which must be preprocessed. Note that in '.cxx', the last two letters must both be literally 'x'. Likewise, '.C' refers to a literal capital C.
file.s - Assembler code.
file.S - Assembler code which must be preprocessed.
other - An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.
You can specify the input language explicitly with the '-x' option:

-x none
Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if '-x' has not been used at all). If you only want some of the stages of compilation, you can use '-x' (or filename suffixes) to tell GCC where to start, and one of the options '-c', '-S', or '-E' to say

where GCC is to stop. Note that some combinations (for example, '-x cpp-output -E') instruct GCC to do nothing at all.

-c

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the suffix '.c', '.i', '.s', etc., with '.o'. Unrecognized input files, not requiring compilation or assembly, are ignored.

-S

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. By default, the assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'. Input files that don't require compilation are ignored.

-E

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files which don't require preprocessing are ignored.

-o file

Place output in file. This applies regardless to whatever sort of output is being produced, whether it is an executable file, an object file, an assembler file or preprocessed C code. Since only one output file can be specified, it does not make sense to use '-o' when compiling more than one input file, unless you are producing an executable file as output.

If '-o' is not specified, the default is to put an executable file in 'a.out', the object file for 'source.suffix' in 'source.o', its assembler file in 'source.s', and all preprocessed C source on standard output.

-v

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-pipe

Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

--help

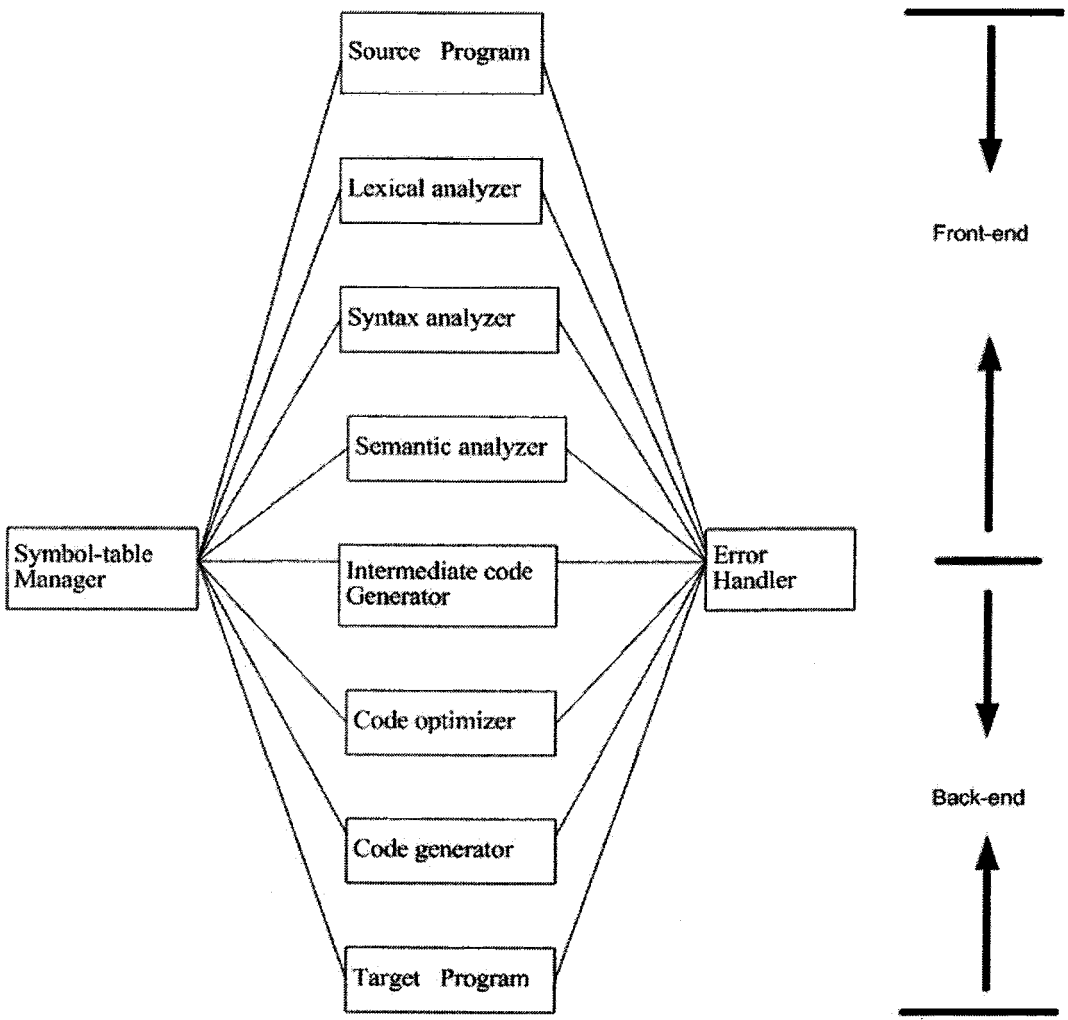
Print (on the standard output) a description of the command line options understood by GCC. If the -v option is also specified then --help will also be passed on to the various processes invoked by GCC, so that they can display the command line options they accept. If the -W option is also specified then command line options which have no documentation associated with them will also be displayed.

4.2 Phases of GCC

Conceptually, a compiler has several phases [4], as shown in Figure 2. Each of these phases transforms the source program into another representation. The characters of the source program are grouped into tokens in the lexical analysis phase. Syntax analysis or parsing involves grouping the tokens of source programs into grammatical phrases, represented as parse trees that are used by the compiler to synthesize output. Semantic analysis performs certain checks to ensure that the components of a program fit together meaningfully and gathers information for subsequent code generation. It also performs type-checking and reports any errors. The Symbol table keeps record of each identifier with fields for

attributes such as type, scope for each of them. Then the compiler generates an intermediate representation, RTL (Register Transfer Language) of the source program for the target machine.

Figure 2: Phases of compiler



Finally the target code is generated, consisting of re-locatable machine code or assembly code. The intermediate RTL instructions are translated into machine instructions. Often the lexical, syntax and semantics analysis phases are collected into front-end and the last three phases into back-end. The back-end depends on the target machine and the target specific intermediate RTL code. But it is independent of the source language.

5 Porting Process

GNU Compiler Collection is mainly concentrated on support for CPUs with several 32-bit general registers and byte-addressable memory. One can even port GCC to a target with 16-bit registers. Also, the size of the memory must not be bigger than what can be addressed from a register.

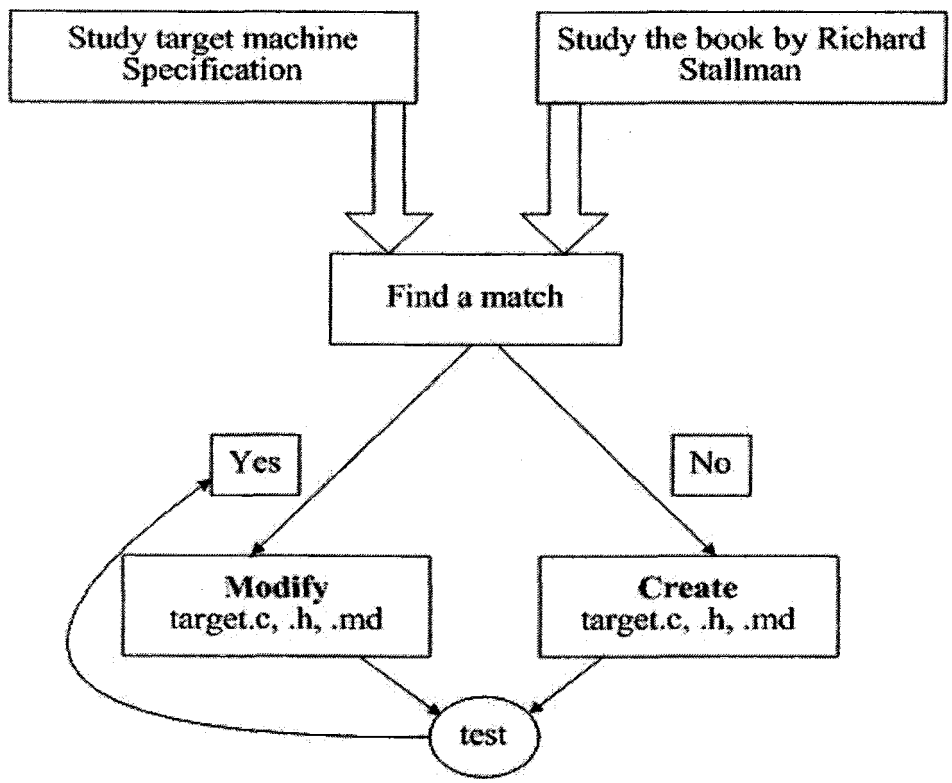
Building a cross compiler can be a tricky procedure and a little hard to understand for the first-time builder. First, the host is the machine on which the compiler is being built and the target is the machine for which the compiler will produce executables. Since support for a wide gamut of target processors already exists, we have decided to port GCC rather than building a cross compiler from the scratch.

While porting GCC, one of the approaches is to look for target architecture with a similar instruction set and for which a GCC cross compiler has already been ported. Also the book "Using and Porting GNU" by Richard Stallman [3] helps a great deal. It is the only standard book on porting GCC. Apart from this, there are various mailing lists on the internet with a detailed FAQ collection [5] which come handy.

For porting GCC to a new target, most of the work is concentrated in 3 files which make up the back-end description of the target. They are the target.c, target.h and target.md. The source files .c and .h have the functions and macros describing the application binary interface for the target. The .md file is the machine description file and it defines the register transfer language (RTL) formats for each of the instruction in the instruction set. Each one of these files is discussed in greater detail below.

In a nutshell the porting process is as described by the flowchart given below. The target architecture has to be studied extensively. If there is architecture who's ISA is similar to that of the target to be ported and a cross compiler for that architecture exists, then go ahead and modify the 3 files mentioned above. Otherwise those files have to be created.

Figure 3: Flowchart of porting process



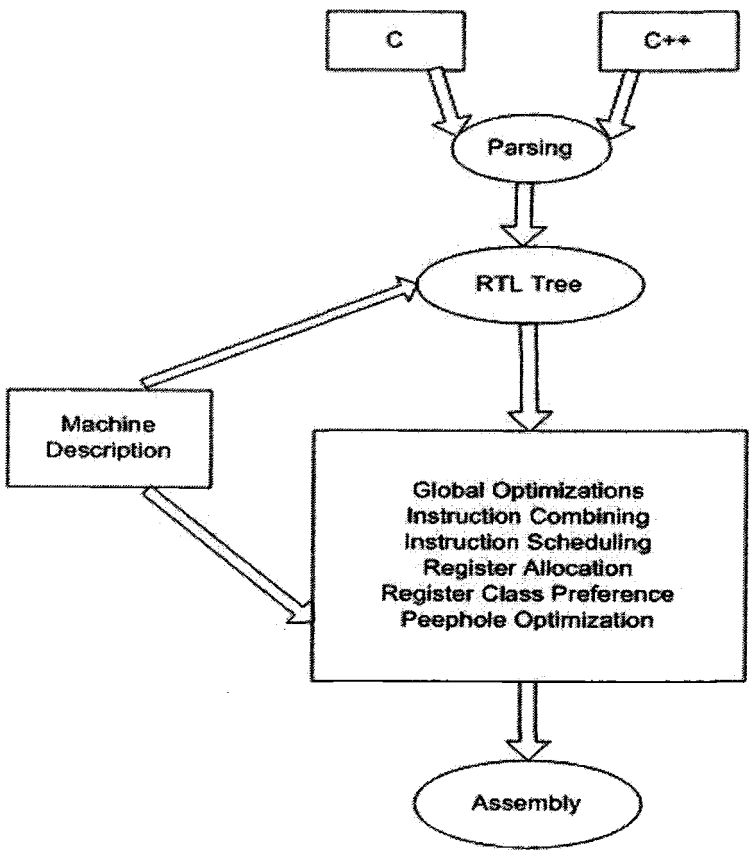
5.1 Machine Description

The machine description is written in a machine description format called RTL, a lisp-like language, which is closely related to the internal data representation, RTX. The architecture for the RTL machine is an abstraction of actual processor architecture. It therefore reflects the technical concepts for the targeted processor like storage, processor registers, address types, jump command etc. A code

generator for a target processor is built through a language which allows one to describe the conversion of patterns of RTL into an assembler language.

The structure of a C compiler is shown in Figure 4. When a C source program is input to the compiler, it scans the program, statement by statement and ensures that the syntax and semantics are correct. Also the type checking is done during this phase. After this a high level tree representation is generated from the input, one function at a time. This is the syntax tree, shown as Parsing phase in the Figure 4 .

Figure 4: Structure of C Compiler



The syntax tree is stored in 'tree' nodes, which are created via function calls defined in files 'tree.def', 'tree.h' and 'tree.c'. These files are part of the GCC distribution. This tree code is then transformed into RTL intermediate code, and

processed. The files involved in transforming the trees into RTL are 'expr.c', 'expmed.c', and 'stmt.c'. The order in which trees are processed is not necessarily the same order that they are generated from the input, due to deferred in-lining and other considerations.

Therefore, on compilation, each statement in the source code is converted into a RTL tree. Subsequently these RTL codes are matched with the patterns in the .md file to generate assembly code.

The .md file contains instruction template patterns usable for both instruction generation and instruction matching. They are instruction definitions, attribute definitions, instruction-splitting descriptions and peephole optimization definitions. The intermediate RTL format is matched with the description in the .md file to generate the assembly output. During compilation, in the first pass a basic sequence of instructions is generated and later passes combines and splits these instructions, trying to match a faster sequence.

The following sections describe the generic RTL format of instructions described by Richard Stallman [3] that go in the .md file, followed by examples of X32V instructions in the subsequent chapter.

5.1.1 Instruction definition patterns

An instruction has the generic pattern definition given below:

(define pattern-type "(optional) pattern name"
[(set (target)
(the operand))
optionally more setting-effects]
"optional pattern-condition"
"output template"
(optionally: [attribute settings]))

The semantics and meaning of each part is given below:

pattern name: The pattern does not have to have a name. It can even be an empty string, which makes it an anonymous pattern. Other than that, names have to be unique. There are some names that are reserved for common and mandatory instruction patterns with a predefined behavior and usage. For example "addsi3" is the standard name for addition involving 3 operands of word size. The three operands are the two source operands and the result. Similarly one of the pattern names for move is "movqi" which denotes the operation of moving a byte value between registers. There are pre-defined standard names for branch and procedure call operations also. If the name does not exist for certain operations then a library routine is called for it.

pattern-type: There are two types of pattern descriptions:

define_insn: It is valid both as a generator and matching pattern and it is the most common pattern type.

define_expand: Some standard-named patterns are preferably translated into a couple of other instructions rather than a library call. This is called pattern expansion. The expansion definition patterns for standard names are used only when an operation related to that standard name is called for. They are not recognized when synthesizing complex instructions from simpler ones.

The pattern name from a define_insn or define_expand is turned into a generated function called gen_name (). The functions generated from standard-named patterns are called explicitly from within GCC, when needed and defined. GCC knows by itself how to synthesize missing instructions for simple logical and arithmetic operations, using existing instruction patterns of larger or smaller sizes. For example, two word-sized "and" instructions can be used on a dword-sized operand, if the machine has no dword-sized "and" instruction.

operand: It is a group of operations that need to be performed. It could be either simple or very complex involving several side-effect operations. For an addition, this looks like (plus:M (operand1) (operand2)), where M denotes the machine mode, and the operands of the operation can in turn be composed of other operations. The machine mode can be either byte (qi), half-word (hi) or word (si). Also there is a fixed rule governing the appearance and placement of operands. Therefore define expand does not specify the original pattern, just the resulting patterns. The generic form is (match_operand: M operand-number "predicate" "constraints"). The operands and operators of the pattern are normally numbered by increasing numbers from zero, with the first destination operand as zero.

The matching expressions can be of four different types:

match_operand: The main case: (match_operand: M operand-number "predicate" "constraints"). The predicate defines what general kinds of operand are allowed. The constraints further specify what combinations of operands are allowed.

match_operator: To match a set of operators, like "plus", "and", "or", you can specify a predicate for that set, just as for different operand types.

match_dup: This is used to match the current operand with n^{th} operand. For this case, just the operand-number part is present; the predicate and constraint are determined to be same as that of the matching operand. For example: (match_dup 1).

match_op_dup: The same as the operand-number n^{th} operator, as with match_dup. The predicate is the name of a C function returning an integer, zero or one, for a match of the operand (or operator) and mode in its fixed-type arguments.

There are several pre-defined predicate functions:

register_operand: An operand that is a register.

address_operand: An operand that is an address, as matched by GO_IFLEGITIMATE ADDRESS ().

immediate_operand: An operand that is a constant (maybe a constant address).

const_int_operand: An immediate operand, but for integer values only.
const_double_operand: An immediate operand, but for floating-point values only.
nonimmediate_operand: An operand that does not match immediate operand.
memory_operand: An operand that is in memory.
general_operand: Just any valid operand; register, constant or memory.
indirect_operand: A memory operand whose address is a general operand.
comparison_operator: An operator that is a comparison (equal, greater, less-than etc.)

The constraints are either empty strings or strings of letters and symbols, one for each operand. During instruction matching, the predicates are matched first and only if it is a successful match, the constraints which serve as second level of matching are checked. Predicates impose stricter guidelines while matching. For example if the predicate is nonmemory _operand, then both integers and register operands gets matched. Then the constraints are checked, if it is string "r" or "i" which leads to matching registers and integers respectively. Operators do not have constraints, but their operands often have. There can be more than one constraint, separated by a comma. For architectures with different classes of registers, it is common to define a letter for each register class. Certain symbols and letters have predefined meanings. Some of the special symbols relate to that alternative only, others mark a property for the entire operand. The following list covers most of the simpler constraints:
0 . . . 9: The operand for this alternative has to be the same as the operand with the specified number.
r: This alternative is a register that is in GENERAL REGS.
g: Any operand that fits the general operand predicate matches this alternative.
m: Any memory operand matches this alternative.
=: The operand is assigned to, and the original value is lost. This is in effect for all alternatives of this operand.

+: This operand is only partly modified and is not completely determined by the assignment of the operand. It marks this effect for all alternatives.

&: This symbol marks that this constraint-alternative is partly written before the other input operands are finished reading in, so they can not be the same as this operand.

%: This operand can be exchanged for the operand with the next number, for all alternatives. This normally happens for commutative operations, such as plus and logical and.

i . . .p: As an example of constraints defined in the target description, these letters are reserved for ranges of constants, defined through the macro CONST_OK_FOR_LETTER().

target: This represents the output of the instruction. In case of branch instructions this is simply cc0 implying condition code register. Otherwise it is the destination operand. The predicates and constraints are to be defined for the target operand.

more setting-expressions: These are the parallel RTL expressions which perform more than one operation simultaneously. But care should be taken while the output has to be exchanged between these RTL expressions. The keyword "parallel" is used to denote parallel expressions.

pattern-condition: An optional (default true) condition for when the pattern applies. This is used to check TARGET_FLAGS which might be defined if there are several architectures for which the compilation could be done.

output_template: It specifies the assembler output. It could be a string corresponding to the instruction to be outputted or it is possible to have a C function which dictates how the operands need to be outputted. For define_insn, the assembler output can be generated in a number of ways given below:

1. If the output template is a piece of C-code then it can return a string and call output_asm_insn (const char *, const rtx []). The first character after " must then be *.
2. Just the assembler instruction, as a string. No format specification is needed.
3. A list of assembler instruction strings, divided by a new-line with surrounding white-space. Then the first character after " has to be @. The number of the matching constraint-alternative decides which list element is used.

If the assembler output string has a percent-sign (%) followed by a number, then it denotes the place of that operand number. The operand is output by PRINT_OPERAND () or PRINT_OPERAND_ADDRESS (), which is defined in the x32v.c file.

attribute_settings: If the instruction has any specific attributes then it is set here.

5.1.2 Instruction splitting

Sometimes a complex instruction can be re-written as a set of simpler instructions which, when executed in the correct order, increases the performance of the code. If the target machine requires delay slots, then the compiler must be able to move instructions appropriately into these delay slots. The disadvantage of doing so is that it will cause the compilation to be slower and require more space. If the resulting instructions are too complex, it may also suppress some optimizations.

Also there are situations where a complex instruction formed by merging three or more simple instructions does not have a matching define_insn pattern. In that case, the compiler splits the complex instruction into simple patterns that can be recognized. However, in some other cases, the way to split instructions is machine-dependent.

The generic format of a "split"-definition description is:

define_split

[insn-pattern]
"condition"
"new-insn-pattern-1
new-insn-pattern-2
...]
preparation *statements")*

insn-pattern: A pattern that needs to be split and **condition** is the final condition to be tested, as in a define_insn. If for an instruction the pattern is matched and the condition satisfied, then that instruction is replaced by the instructions from the new patterns given in the replacement list.

preparation statements: It is similar to those statements that are specified for define_expand and are executed before the new RTL is generated to prepare for the generated code or emit some instructions whose pattern is not fixed.

5.1.3 Peephole optimizations

Certain instruction sequences are not easily optimized from analysis of the data flow. For example, sometimes two consecutive instructions might be writing a value to the same register. If the value computed by the first instruction is not being used, but is overwritten by the second instruction then this can be detected by a machine specific peephole optimizer. It is taken care of by the define_peephole pattern, usually written towards the end of the machine descriptions. The generic format is the same as for define_expand and define_insn, but without a name. For example:

(define_peephole
[insn-pattern-1
insn-pattern-2
....]

"condition"

"template"

"optional attribute statements"*)*

insn-pattern-1, insn-pattern-2: These are patterns to be matched with consecutive instructions. Only when the pattern1 is matched, the optimizer continues matching the rest of the patterns. When all the patterns are matched and a corresponding define_insn pattern also exists then the optimized instruction patterns will be outputted. But if a corresponding define_insn pattern does not exist then the compiler might crash.

Peepholes are checked only at the last stage just before code generation, and only optionally. The peephole optimizations are machine dependent and only meant for cases where no other possibility of optimization exists. GCC makes no attempt to rearrange instructions to match the peephole optimizations.

5.1.4 Attribute definitions

In the machine description we can also define attributes for each instruction. For example whether the instruction sets the condition code register or not, whether it results in an overflow or not, etc. These attributes can further be used by NOTICE_UPDATE_CC to track the condition codes. Also it is possible to group certain attributes together like in MIPS depending on logical combining of instructions. The define_attr expression is used to define each attribute required by the target machine. The generic pattern looks like:

(define attr name list-of-values default)

name: Is a string specifying the name of the attribute being defined.

list-of-values: is either a string that specifies a comma-separated list of values that can be assigned to the attribute, or a null string to indicate that the attribute takes numeric values.

default: Is an attribute expression that gives the value of this attribute for instructions that match patterns whose definition does not include an explicit value for this attribute.

For each defined attribute, a number of definitions are written to the 'insn-attr.h' file. For cases where an explicit set of values is specified for an attribute, the following are defined: A '#define' is written for the symbol 'HAVE_ATTR_name'. An enumerable class is defined for 'attr_name' with elements of the form 'upper-name_upper-value' where the attribute name and value are first converted to upper case. A function 'get_attr_name' is defined that takes an instruction and returns the attribute value for that instruction.

6 X32V port

Working with the GNU toolset on the Intel Pentium III and the Mandrake Linux Operating System, the Enhanced GNU Compiler System (egcs version 1.1.2) was ported for X32V architecture.

The nearest port to X32V was determined to be ae32k, which is a 32-bit RISC based processor core, big-Endian and has load/store architecture. Hence we have modified the machine description and macro definition files of ae32k, while retaining its calling conventions.

6.1 X32V ABI

The X32V ABI (Application Binary Interface) defines the calling conventions, how functions are handled, which registers need to be saved and restored, how arguments are passed and which registers hold the value returned from a function. The functions prologue () and epilogue () which define the function entry and exit are defined in the x32v.c file.

prologue (file, size): C compound statement that outputs the assembler code for entry to a function. The prologue is responsible for setting up the activation frame, initializing the frame pointer register, saving registers that must be saved, and allocating *size* additional bytes of storage for the local variables. *Size* is an integer, and *file* is a standard I/O stream to which the assembler code should be output.
epilogue (file, size): C compound statement that outputs the assembler code for exit from a function. The epilogue is responsible for restoring the saved registers and stack pointer to their values when the function was called, and returning control to the caller. This macro takes the same arguments as the macro

FUNCTION_PROLOGUE, and the registers to restore are determined from regs_ever_live and CALL_USED_REGISTERS in the same way.
The "push" and "pop" instructions are used to store and restore the required register values. In X32V calling convention, register r8 and r9 are used to pass the arguments to a function and the result is returned in r9 register. As of now, passing variable arguments to functions is not implemented.

The predicates, constraints and any other functions which are called in the .md file are defined in the x32v.h or x32v.c files. The .h file has macro definitions, global variable declarations, while any other code which cannot be expressed as a macro is placed in the x32v.c file.

Some of the macros and functions defined in these files are given below along with their intended purpose.

```
/* Define this if most significant bit is lowest numbered in instructions that operate
on numbered bit-fields. */
#define BITS_BIG_ENDIAN 1

/* Define this if most significant byte of a word is the lowest numbered. */
#define BYTES_BIG_ENDIAN 1

/* Define this if most significant word of a multiword number is lowest numbered.
*/
#define WORDS_BIG_ENDIAN 1

/* Number of bits in an addressable storage unit. */
#define BITS_PER_UNIT 8

/* Width in bits of a "word", which is the contents of a machine register. */
```

```
#define BITS_PER_WORD          32

/* Width of a word, in units (bytes). */
#define UNITS_PER_WORD         4

/* Width in bits of a pointer. See also the macro `Pmode' defined below. */
#define POINTER_SIZE           32

/* Allocation boundary (in *bits*) for storing arguments in argument list. */
#define PARM_BOUNDARY          32

/* The stack goes in 32 bit lumps. */
#define STACK_BOUNDARY         32

/* Allocation boundary (in bits) for the code of a function.  8 is the minimum
boundary*/
#define FUNCTION_BOUNDARY 16

/* No data type wants to be aligned rounder than this. */
#define BIGGEST_ALIGNMENT   32

/* Alignment of field after 'int : 0' in a structure. */
#define EMPTY_FIELD_BOUNDARY   32

/* Define this if move instructions actually fail to work when given unaligned data.
*/
#define STRICT_ALIGNMENT 1

/* Standard register usage.  Number of actual hardware registers.  The hardware
registers are assigned numbers for the compiler from 0 to just below
```

```
FIRST_PSEUDO_REGISTER.  All registers that the compiler knows about must
be given numbers, even those that are not normally considered general registers.
*/
#define FIRST_PSEUDO_REGISTER 22

/* 1 for registers that have definitive standard uses and are not available for the
register allocation.  */
#define FIXED_REGISTERS \
    { 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1 }
/*    0  1  2  3  4  5  6  7  8  9  10  11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 */
 /* 5 is Frame Pointer, 6 is Index Register and 15 to 21 are the special purpose
registers.  1 for registers not available across function calls.  These must include
the FIXED_REGISTERS and also any registers that can be used without being
saved.  The latter must include the registers where values are returned and the
register where structure-value addresses are passed.  */

#define CALL_USED_REGISTERS \
   { 0, 0, 0, 0, 0, 1, 1, 0, 1, 1,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1 }
/*    0  1  2  3  4  5  6  7  8  9  10  11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 */

#define REG_ALLOC_ORDER \
 {8, 9,                                  /* return/argument registers */ \
  0, 1, 2, 3, 4, 7, 10, 11, 12, 13, 14, 15,  /* general registers       */ \
  5, 6, 16, 17, 18, 19, 20, 21           /* fixed registers         */ \
 }

/* Stack layout; function entry, exit and calling.  */

/* Define this if pushing a word on the stack makes stack pointer a smaller address.
*/
```

```
#define STACK_GROWS_DOWNWARD

/* Define this if the nominal address of the stack frame is at the high-address end
of the local variables; that is, each additional local variable allocated goes at a
more negative offset in the frame. */

#define FRAME_GROWS_DOWNWARD

/* Offset from the frame pointer to the first local variable slot to be allocated.
Offset within stack frame to start allocating local variables at. If
FRAME_GROWS_DOWNWARD, this is the offset to the END of the first local
allocated. Otherwise, it is the offset to the BEGINNING of the first local allocated.
*/

#define STARTING_FRAME_OFFSET 0

/* These are the various functions used in the .md file whose definitions appear in
the x32v.c file. */
extern void asm_file_start ();
extern int const_costs ();
extern void print_operand ();
extern void print_operand_address ();
extern void expand_prologue ();
extern void expand_epilogue ();
extern void notice_update_cc ();
extern int call_address_operand ();
extern int impossible_plus_operand ();
extern enum reg_class secondary_reload_class ();
extern void override_options ();
extern int initial_offset ();
```

```
extern char *output_tst ();
int symbolic_operand ();
```

6.2 RTL examples of X32V instructions

While developing the machine descriptions for X32V, we broke it down into different sets based on the categories of instructions. The arithmetic/logical instructions like SUB, LG_AND, LG_OR etc were grouped together as they had similar operand constraints and assembly output notations. But the ADD instruction had a different treatment as it can have special registers as operands, hence imposing more severe constraints.

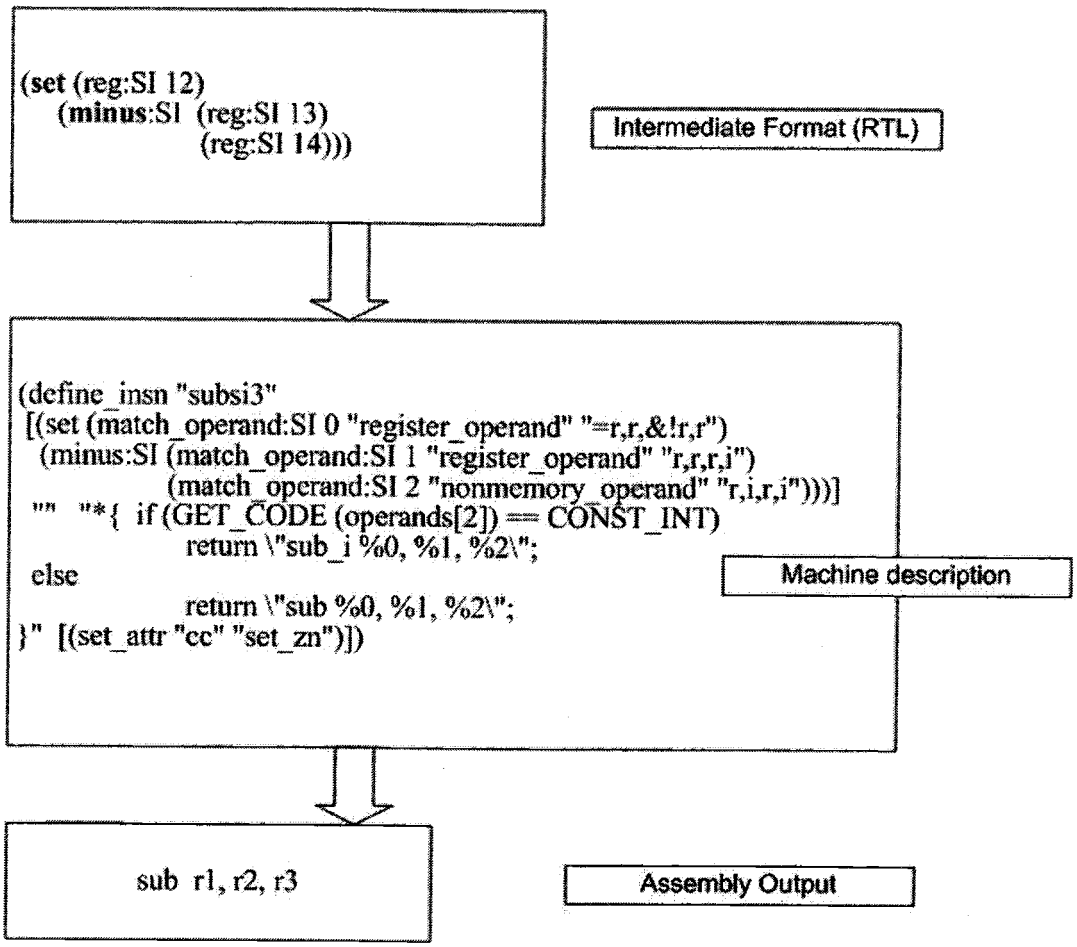
In this section, we describe a few x32v instructions. We start with a simple subtraction instruction and then delve into more intricate and complex notations of add, shift and branch instructions. Also a brief discussion on load/store instructions is given.

The "sub" instruction definition is given below followed by the pictorial description of the RTL to Assembly process in Figure 5.

```
(define_insn "subsi3"
 [(set (match_operand:SI 0 "register_operand" "=r,r,&!r,r")
       (minus:SI (match_operand:SI 1 "register_operand" "r,r,r,r")
                 (match_operand:SI 2 "nonmemory_operand" "r,i,r,i")))]
 ""
 "*
{
 if (GET_CODE (operands[2]) == CONST_INT)
                 return \"sub_i %0, %1, %2\";
 else
```

```
                return \"sub %0, %1, %2\";
}"
 [(set_attr "cc" "set_zn")])
```

Figure 5: RTL to Assembly



This is a very simple example. It defines a standard-named pattern for the "sub" instruction. The mode of the operands is the 32 bits of the register (SImode). It takes 3 operands numbered 0, 1, and 2. Hence the pattern name is subsi3. The destination operand is number 0 and it has to be a register, hence its predicate is defined to be "register_operand", which is also imposed on operand 1. Operand 2 could either be a register or an immediate value; hence the predicate

"nonmemory_operand" is used. Furthermore, there is the constraint "r" on destination operand, implying that it can be a general-purpose register. The string "x" which implies a special register is not used for the constraint. The X32V sub instruction does not support subtraction from the stack register.

Since the sub instruction has both register and immediate formats, there is an if-else condition which is used to determine the correct format of the instruction depending on whether operand 2 is a register or an immediate value.

The assembler output is simple; just the text "sub_i" or "sub" followed by the three operands, where the destination register is the first one. The return statement is included satisfying the * requirement. The instruction has an attribute that is not covered by the default; the attribute sets the condition code register and set_zn, hence these two values are specified here.

The addition instruction is more complex as it can have general purpose registers or immediate value or special purpose register as its operands. The RTL format for the ADD instruction is explained as below.

```
(define_expand "addsi3"
 [(set (match_operand:SI 0 "register_operand" "")
       (plus:SI (match_operand:SI 1 "register_operand" "")
                (match_operand:SI 2 "nonmemory_operand" "")))]
 ""
 "
{
 /* We can't add a variable amount directly to the stack pointer. Hence do it via a
temporary register. */
 if (operands[0] == stack_pointer_rtx
    && GET_CODE (operands[1]) != CONST_INT
```

```
    && GET_CODE (operands[2]) != CONST_INT)
  {
   rtx temp = gen_reg_rtx (SImode);
   emit_move_insn (temp, gen_rtx (PLUS, SImode, operands [1],
   operands [2]));
   emit_move_insn (operands[0], temp);
   DONE;
  }
}")
```

In this define_expand pattern, we establish the base conditions which specify how the add instruction has to be dealt with. We check for the destination operand: if it is a stack pointer register then we go ahead and determine whether the operands 1 and 2 are constant values or registers. In the former case, it simplifies the process. We go ahead and output the immediate assembly instruction format. But if the latter case is true, then the contents of the registers are added and moved to a temporary register. Finally this value is added to the stack pointer register. This complicated implementation is because of the fact that general purpose registers cannot be added directly to the stack pointer register.

The various RTL constructs used in this implementation are explained below.

gen_reg_rtx (SImode): This is a generator construct used to generate a general purpose register. The width of the register would be same as that defined in the x32v.c file. The parameter passed in this case is SImode which states that the register created would hold a SImode value(32 bit).

gen_rtx (PLUS, SImode, operands[1], operands[2]): This is also a generator construct and it returns an RTX expression, which holds the value computed by performing the operation specified in the parameter list on the rest of the operands supplied as parameters.

GET_CODE: This returns the RTX code of the operand passed. The result could be compared against **CONST_INT** or **CONST_REG**.

The **emit_move_insn** is an inbuilt construct which moves values form the source register to the destination register mentioned by the parameter list by generating RTL code for a move instruction.

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r,r,&!r,x")
        (plus:SI (match_operand:SI 1 "register_operand" "r,r,r,r")
                 (match_operand:SI 2 "nonmemory_operand" "r,i,r,i")))]
  ""
  "*
{
  if (GET_CODE (operands[2]) == CONST_INT)
                return \"add_i %0, %1, %2\";
  else
                return \"add %0, %1, %2\";
}"
[(set_attr "cc" "set_zn")])
```

The define_insn pattern for the add instruction is straight-forward. The constraint "r" on destination operand implies that it can be a general-purpose register while "x" implies it could be a special register. The X32V add instruction supports the addition of stack register and an immediate value. Depending on the type of operand 2 one of the two formats of the add instruction gets outputted.

The move instructions are categorized based on the size of the operands. Since the X32V architecture supports moving of 8, 16, 32 bit signed and unsigned values between registers, we have **moveqi**, **movehi** and **movesi** patterns explicitly defined in the .md file. A detailed explanation of **movsi** follows.

```
(define_expand "movsi"
 [(set (match_operand:SI 0 "general_operand" "")
       (match_operand:SI 1 "general_operand" ""))]
 ""
 "
{
 /* One of the operands has to be in a register */
 if (!register_operand (operand1, SImode)
   && !register_operand (operand0, SImode))
       operands[1] = copy_to_mode_reg (SImode, operand1);
}")
```

To be noted is that the "move" instruction takes only two operands and moves the values between them. Hence the set expression has only two predicates and constraints. At least one of the operands has to be a registers; if this is not the case, then one of the operands is forced into a register (of **SImode** as this is the pattern definition for **movsi)** using the **copy_to_mode_reg (SImode, operand1)** construct. Thus the expand definition imposes and establishes the necessary conditions for the following define_insn pattern to generate appropriate assembly.

```
(define_insn "movsi_insn"
[(set (match_operand:SI 0 "general_operand" "=r,*r,*r,*r,m")
       (match_operand:SI 1 "general_operand" "0,i,*r,m,*r"))]
 "register_operand (operands [0], SImode)
 || register_operand (operands [1], SImode)"
 "*
 {
  long val;
  switch (which_alternative)
  {
   case 0:
```

```
    return \"nop\";
  case 1:
      val = INTVAL(operands[1]);
    return \"mov_upi %0, %1\";
  case 2:
    return \"mov %0, %1\";
  case 3:
   if (REGNO (operands[0])
       == STACK_POINTER_REGNUM)
            return \"l_w %%r6, %1\;mov %0, %%r6\";
      else
            return \"l_w %0, %1\";
  case 4:
   if (REGNO (operands[1])
       == STACK_POINTER_REGNUM)
            return \"mov %%r6, %1\;s_w %0, %%r6\";
      else
            return \"s_w %0, %1\";
  }
}"
[(set_attr "cc" "none,none_0hit,none_0hit,none_0hit,none_0hit")])
```

The define_insn pattern requires some detailed exploration as it introduces the concept of switch constructs relative to RTL language. The load and store instructions are also treated as a type of move instruction. The variable **which_alternative** is an in-built variable used with switch statements. The constraints of operands 0 and 1 are used to determine the value of the expression to the switch statement. As per the constraints, if both the operands are registers then which_alternative evaluates to 2, thus emitting the assembly format for moving registers. But if operand 0 (destination) is a register while the other operand is a

memory location, then the which_alternative evaluates to 3, as this condition is listed as 3rd in the constraint list. This implies moving a value from a memory location to a register, similar to a load instruction. Hence the load assembly pattern gets outputted in switch case 3.

Supposing the destination operand is a memory location and operand 1 is a register then case 4 is evaluated and output for a store assembly instruction is generated. Also the last two cases for load and store reiterate that the moves between registers and stack pointers cannot be performed directly. The index register R6 is used as a temporary register for achieving this. This pattern described above is for **SIword** values. Similar patterns for **HIword** and **QIword** which deal with half-word and byte values' move, load and store instructions respectively are implemented in the machine description file. The signed load instructions (both half-word and byte) are dealt with slightly differently and explained in the next section.

6.3 Problems Encountered

As mentioned earlier, porting GCC is a daunting task, which the lack of documentation or support makes even more difficult. Further, there was an imminent time deadline which had to be met. Also other constraints, like relying on mailing lists for any technical support, proved to be a factor which made this porting process a tough ride. In this section I have listed some of the technical problems encountered and the ways I overcame them.

Conflicting prototypes of branch instructions in ae32k and X32V was one of the problems. One of the interesting instructions of any ISA is the branch instruction. After considerate deliberation, we decided to have two versions of branch instructions which take 3 and 2 operands respectively, along with a provision for

immediate branch instruction. Also there are no compare or test instructions defined for X32V, as the branch instructions handle all the possibilities.

The files of the compiler core of ae32K, defining the RTL protocols for branch generation expected only two parameters (operands) for branch instructions, while the X32V branch patterns had both 2 and 3 operands. This posed a severe problem while porting. We did consider changing the branch pattern but ruled it out as that would impair our instruction set, which lacks compare and test instructions. And the base compiler could not have been changed as several months of work had already been invested in porting. So after studying several ports and trying several alternatives to implement branch instructions, we came up with a scheme which proved successful.

We defined a **tstsi** instruction pattern which takes one operand passed as a parameter, while the other operand is a constant zero. It handles branches with 2 operands and performs the "compare with zero" operation. It writes the result into the condition code register, which is used by branch patterns to determine the correct value for the program counter. We define two static variables for storing the operands and a flag which determines the type of branch: whether immediate or register format. The motive behind this is to distinguish the different branch patterns and output the correct assembly format at the later stage. We store the operands in the static variables and set the imm_branch flag to 1. This pattern does not output any assembly instructions, but performs the ground work for immediate branch patterns by returning a null pattern. The tstsi definition is shown below.

```
(define_insn "tstsi"
[(set (cc0)
      (compare (match_operand:SI 0 "register_operand" "!*r,r,r,r")
               (const_int 0)))]
```

```
""
"*
{
   branch_cmp[0] = operands[0];
   branch_cmp[1] = const0_rtx;
   imm_branch = 1;
   return \"\";
}"
[(set_attr "cc" "set_znv")])
```

While the **tstsi** took care of immediate zero branch instructions, we had to define the **cmpsi** pattern to handle the other branch format. Here a subtle trick is used to generate correct branch patterns. In the case of an immediate compare, we generate a sub instruction which stores the result of the subtraction of operand 1 from 0 in the register number matching operand 0. Also the imm_branch flag is set to 1. This would ensure that the outcome of subtraction is available to the branch instruction, which is explained later. In the case of a register compare, a null pattern is outputted and the imm_branch flag is set to 0 indicating that it is a register compare. The operands are copied into the static variables. The cmpsi pattern is described below.

```
(define_insn "cmpsi"
[(set (cc0)
        (compare (match_operand:SI 0 "register_operand" "!*r,r,r,r")
                 (match_operand:SI 1 "nonmemory_operand" "!*0,0,i,r"))))]
""
"*
{
   if (GET_CODE (operands[1]) == CONST_INT )
    {
```

```
    imm_branch = 1;
    branch_cmp[0] = operands[0];
    branch_cmp[1] = operands[1];
    return \"sub_i %0 %0 %1 \";
   }
   else
   {
    imm_branch = 0;
    branch_cmp[0] = operands[0];
    branch_cmp[1] = operands[1];
    return \"\";
   }
}"
[(set_attr "cc" "invert,compare,compare,compare")])
```

This is a generic branch pattern. Depending on the type of branch (greater than, lesser than, equal to, etc) different patterns say **bge, ble, beq, bne, bgt, blt** are defined individually. The expand definition compares the condition code register with constant zero. The comparison operator used is corresponding to the branch type. For a "greater than" branch, the operator used is **gt** and the branch pattern name is **bgt**. An if-else construct is used to determine the correct value of the program counter. If the condition code register is zero then the label passed with the branch instruction is used to set the value of the program counter, otherwise the program counter is incremented by 4 automatically by the compiler. We need not write it explicitly.

```
(define_expand "bxx"
[(set (pc)
      (if_then_else (xx (cc0)
                        (const_int 0))
```

```
                (label_ref (match_operand 0 "" ""))
                (pc)))]
""
"")
```

The define_insn pattern for branch is explained below. The temporary character buffer is used to output the correct branch pattern. The branch instructions have the r8 and r9 registers fixed as operands. The type of comparison operator is determined by checking the rtx code for match operand 1 against all possible combinations and it is copied to the buffer. Then depending on the type of branch, checking if the imm_branch flag is set to 0 or 1, the appropriate branch assembly instruction is outputted. Although using temporary buffers is not a very good design idea, we are sticking to it due to the constraints briefly described earlier. The branch definition is given below.

```
(define_insn ""
[(set (pc)
        (if_then_else (match_operator 1 "comparison_operator"
                                    [(cc0) (const_int 0)])
                (label_ref (match_operand 0 "" ""))
                (pc)))]
""
"*
{
  char buffer[80] =\"\" ;
  rtx xoperands[3];
  xoperands[0] = operands[0];

 if ((cc_status.flags & CC_OVERFLOW_UNUSABLE) != 0
    && (GET_CODE (operands[1]) == GT
```

```
        || GET_CODE (operands[1]) == GE
        || GET_CODE (operands[1]) == LE
        || GET_CODE (operands[1]) == LT))
    return 0;

  strcat(buffer,\"%!b_%b1\");

  /* Only if imm_branch is 0 append the reg2 to the instruction. */
  if(imm_branch == 0)
  {
    strcat(buffer,\" %%r8 %%r9 %0\");
  }
  else
    strcat(buffer,\" %%r8 %0\");

  output_asm_insn (buffer, operands);
  return \"\";
}"
[(set_attr "cc" "none")])
```

Another interesting implementation is of the shift instructions. There are both arithmetic and logical shift instructions defined in the X32V instruction set. The logical and arithmetic shifts left are treated as the same in the GNU compiler set. Hence one instruction definition would suffice. The RTL built-in constructs **ashift** (arithmetic left shift), **ashiftrt** (arithmetic right shift) and **lshiftrt** (logical shift right) are used. The define_expand pattern checks if one of the operands, sparing the destination, is an immediate value in the permitted range. In that case, it goes ahead and generates the instruction pattern for the immediate format with the call for the **gen_ashlsi3_const** function. The definition of ashlsi3 follows below.

```
(define_expand "ashlsi3"
[(set (match_operand:SI 0 "register_operand" "")
        (ashift:SI
         (match_operand:SI 1 "register_operand" "")
         (match_operand:SI 2 "nonmemory_operand" "")))]
""
"
{
 if (GET_CODE (operands[2]) == CONST_INT &&
    (!(CONST_OK_FOR_J(INTVAL(operands[2]))) &&
     !(CONST_OK_FOR_K(INTVAL(operands[2]))) &&
     !(CONST_OK_FOR_L(INTVAL(operands[2]))) &&
     !(CONST_OK_FOR_M(INTVAL(operands[2])))))
  {
    emit_insn (gen_ashlsi3_const (operands[0], operands[1], operands[2]));
    DONE;
  }
}")
```

The **CONST_OK_FOR_J()** functions are defined in the x32v.c file. They define the set of immediate values that are allowed to appear in an instruction pattern. The **INTVAL(operands[])** is an in-built function which returns the integer value of the operand passed as the parameter. The define_insn pattern is simple and outputs the register format of the shift instruction.

```
(define_insn "ashlsi3_insn"
[(set (match_operand:SI 0 "register_operand" "=r,y")
        (ashift:SI
     (match_operand:SI 1 "register_operand" "r,r")
     (match_operand:SI 2 "nonmemory_operand" "r,z")))]
""
```

```
"*
{
 return \"sft_ll %0 %1 %2\";
}"
[(set_attr "cc" "set_zn")])
```

The ashlsi3_const pattern defines how the constant value needs to be shifted. The X32V shift instruction shifts only the least significant 5 bits of the immediate value. A temporary variable of RTX type is declared and the immediate value is assigned to it. Further on performing a logical "AND" operation with constant value 31, we extract the least 5 bits of the immediate value. This is copied into the operand 2 and used to output the assembly format. The actual definition follows.

```
(define_insn "ashlsi3_const"
 [(set (match_operand:SI 0 "register_operand" "=y")
       (ashift:SI
    (match_operand:SI 1 "register_operand" "r")
    (match_operand:SI 2 "const_int_operand" "i")))
  (clobber (match_scratch:SI 3 "=&z"))]
 ""
 "*
{
 rtx xoperands[2];
 xoperands[0] = operands[2];
 xoperands[0] = GEN_INT(INTVAL(xoperands[0]) & 31);
 operands[2] = xoperands[0];
 return \"sft_lli %0 %1 %S2\";
}"
[(set_attr "cc" "clobber")])
```

The move and load instructions sometimes have registers holding byte or half-word values to be moved into word length registers. This requires zero extending or sign extending those registers to word length depending on whether it is a signed or unsigned load. Hence the zero_extendqisi2 and sign_extendqisi2 patterns are defined. The **zero_extend()** function which takes a byte value register and extends it to the word value is used. The following is the definition of zero_extendqisi2 which zero extends the byte value to full word.

```
define_insn "zero_extendqisi2"
[(set (match_operand:SI 0 "general_operand" "=r,r,r")
        (zero_extend:SI
         (match_operand:QI 1 "general_operand" "0,r,m")))]
""
"*
{
 long val;
 rtx xoperands[3];
 switch (which_alternative)
 {
  case 0:
   return \"nop\";
  case 1:
   return \"mov %0, %1\";
  case 2:
   return \"l_bu %0, %1\";
  }
}"
[(set_attr "cc" "set_zn")])
```

Similarly, on the same lines we can have zero extension patterns for half-word to word instructions. Just like zero extensions, there are situations where a sign extension of byte and half-word to word is required. Hence the extendqisi2 pattern is defined which uses the in-built **sign_extend()** function..

```
(define_insn "extendqisi2"
[(set (match_operand:SI 0 "general_operand" "r,r")
        (sign_extend:SI
         (match_operand:QI 1 "general_operand" "r,m")))]
""
"@
  mov %0,%1;
  l_b %0,%1"
[(set_attr "cc" "set_zn")])
```

The peephole optimization tracks subsequent updates to the same register and combines them and outputs a single assembly instruction.

6.4 Limitations

One of the conspicuous limitations of the thesis work is the lack of support for floating point instructions and EM3 modules. The floating point operations are supported through specific macros, which are supplied by the fp-bit floating point library. They are not hand-coded in the back-end machine description module of the compiler.

EM3 is the multi-media extension to the X32V. Supporting EM3 requires writing certain routines in C or X32V assembly which mimics the behavior of the EM3 instructions. This library then has to be integrated to the compiler such that calls to appropriate functions are made when the corresponding instruction occurs. In

its intensity this would demand a work of three to four months. Due to time constraints and limitations in resources we have set this as future enhancement to the cross compiler.

Furthermore, at this time only the 32 bit instruction set is supported while provisions to support 16 and 24 bit instructions are reserved for future work.

6.5 Porting Guidelines

One of the salient features of GCC which makes it the foremost choice for porting is the detailed bifurcation of machine dependent code and machine independent code. It can be ported for any 32 bit new target architectures easily despite lack of documentation. Listed below are some of the guidelines followed during the porting process.

The instruction patterns for matching and generating the assembly instructions for a target are clustered in the .md file. GCC has a detailed list of standard names for each of the operations. For Ex: If there is an instruction which adds two numbers of 32 bits width and stores the result of 32bit width in a register, then the corresponding name to look for is "addsi3". The key is to correlate the operation each instruction performs with one of the standard names defined by GCC.

Once the standard name is identified, the next task is to determine if the instruction is straight forward and could be matched with RTL tree and generates the assembly instruction on the fly. If this is the case then the define_insn pattern for that particular name is to be written which outputs the assembly. But if the instruction internally needs some ground work to be performed before generating the assembly, then a define_expand pattern is defined in addition to the define_insn pattern. The define_expand pattern does not generate assembly

instructions but it imposes the predicates and constraints on the operands. This discussion highlighted the steps to add a new arithmetic or logical instruction or modify an existing one in the machine description file.

The function calls, procedures are handled by call_insn patterns. Normally the calling conventions need not be altered unless there are target specific registers to be used for passing parameters unless the Application Binary Interface of the target being ported is very different form that of the base architecture. The functions epilogue, prologue defined in the .c files need to be altered to change the ABI.

The branch instructions can be a potential bottleneck. It needs to be checked if the target has condition code register which holds the results of branch operations. The branch address to jump to would be computed based on the condition code register cc0. It would save time if the branch formats of the target and the base architecture match in terms of the number of operands. The placement of the operands in the instruction does not matter. But the destination operand is numbered zero operand always.

Also GCC has collection of arithmetic and logical operations like plus, minus, compare defined. There are several RTL constructs to create a register of a specified width, or to move value between temp registers or store the operands. These operations and constructs could be used while defining the instruction patterns.

The source code for the cross compiler is under the /egcs-1.1.2/GCC/config/x32v directory. After defining the patterns, macros in x32v.c, x32v.h and x32v.md, the full fledged building process starts. There are changes to several other files, with xm-x32v.h being noticeably important. It defines the general attributes of the target.

The compiler is configured in the egcs-1.1.2/GCC directory, with target specified as x32v, include library as newlib and source language as 'C', before being made using the GCC makefile.

./configure --target=x32v --with-newlib --languages=c

make –k

make –k install

A script was written which performs the steps listed above. This simplifies the building process and frees the user from the hassles of the building process. The script is run from the egcs-1.1.2/GCC directory and it produces the executable "cross compiler" which is further used to compile test benches. To run a different test program, edit the script file and replace the old test file with the new test file.

The option –k is used as the assembler for X32V is not integrated with the compiler and it is provided by a third party.

7 Results

At the time of this writing we did not have an assembler yet. Hence the compiler was tested by compiling individual source files from the ADPCM, G711 benchmarks.

ADPCM: Adaptive differential pulse code modulation. It is a family of Audio compression and decompression algorithms. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The files for the adpcm program namely adpcm.c, rawdaudio.c, rawcaudio.c, timing.c are compiled.

G711: It is collection of the audio format encoding and decoding algorithms used in Voice over IP (VoIP). It is the most widely used compression algorithms for world-wide telecommunications. It compresses linear PCM sample to an 8-bit logarithmic representation. The G711 program including files decode.c, encode.c, g711.c is compiled to generate assembly for X32V.

The X32V assembly output for these files is given in the appendix B.

8 Conclusion

In order to facilitate the simulation and profiling of X32V instructions, a compiler was deemed to be inevitable. Although the initial intent was to have a rudimentary cross compiler outputting X32V binary, the current trend for reconfigurable compilers influenced our approach. Working with GNU toolset on Intel Pentium III and Mandrake Linux Operating System, Enhanced GNU Compiler System (egcs version 1.1.2) is ported for X32V architecture. The time frame from the conception of requirement for cross compiler to having a working cross compiler was about nine months. At this time only 32 bit instruction set is supported while provisions to support 16 and 24 bit instructions are reserved for future work. Also support for EM3 modules is delegated as future work. The floating point operations are handled by macro modules in the fp-bit floating point library while newlib library provides the run-time support.

Porting process involved identifying architecture similar to X32V and making relevant modifications to the machine descriptions and calling conventions. The ABI for X32V is defined with registers R16 designated as Stack Pointer while R5 and R6 are the Index and Frame registers. The parameter passing is done through registers R8 and R9, with the latter holding the return value.

The associated makefile defines certain flags for the target X32V, which when set appropriately compiles the binaries for regular set of instructions. At the instance when compiler comes across an unsupported instruction "nop" is inserted. This technique prevents the compiler form crashing and continues outputting binaries. Addition of new instructions can be done with ease as the entire machine dependent code is isolated to specific files.

The script file provided with the source code is used to configure and build the compiler, thereby keeping the build process transparent to the user. Also the script file can be edited to compile different test programs. The x32v assembler assembles the compiled code to generate x32v assembly. And further, results of running few media benchmarks have been discussed in this report.

The future work would be directed towards developing a wrapper around the existing compiler which takes as input, in addition to the source program to be translated, a machine description of the processor core and generates the configurable compiler which scans the source program and generates the assembly for that particular target as output. Almost no processor specific source code is to be included in the compiler but the target specific characteristics are captured in explicit target description files. It will reduce the time develop compilers for new architectures, further reducing the time to market the embedded systems.

9 Bibliography

Hennessy, J.L., and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, 2003, pp. A-1 - A-56.

Hennessy, J.L., and D.A. Patterson, *Computer Organization and Design*: *The Hardware /Software Interface*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, 1998.

Stallman, R., *Using and Porting GCC*, Free Software Foundation, Boston, 1990.

Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley Publications, Boston, 1986.

www.gnu.org

APPENDIX A

X32V Instruction Set

1 Architecture

1.1 Introduction

The purpose of this manual is to give the reader an understanding of the X32V architecture and instruction set. X32V is a RISC based microprocessor. It is built around a 32-bit data-path, five-stage pipeline. The core ISA is based on 32-bit instructions; however, later sections describe the multiple instruction format modes (32, 32/16, 32/24/16) that X32V operates in.

1.2 Addressing Modes

Keeping in line with a strict RISC architecture, X32V only allows loads and stores to access data memory. As such, only immediate and displacement addressing modes are supported. Immediate addressing mode allows a 16-bit immediate value as an operand. Displacement addressing uses a GPR in conjunction with a signed immediate value to provide the effective address.

Indirect addressing is possible if the immediate value is set to zero. Likewise, absolute addressing is accomplished by specifying a 16-bit immediate value and a GPR, which is set to zero. Byte, half-word, and word addressing are all available. However, word addressing is word aligned, half-word addressing is half-word aligned, and byte addressing is byte aligned.

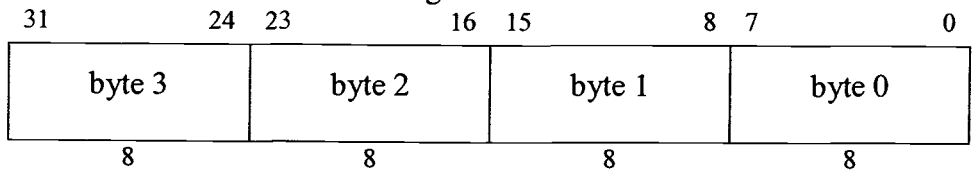
1.3 Registers

Registers are broken up into three different categories: General Purpose, Floating Point, and Reserved Usage.

1.3.1 General Purpose Registers (GPRs)

X32V has 16, 32-bit, GPRs (R0 – R15). None of these GPRs have reserved usage. Registers can be loaded with byte (8-bits), half-word (16-bits), or word (32-bits) values.

Table 1: Big Endian Word



Byte values are stored in byte 0. Half-word values are stored in byte 1 and byte 0. Word values use all four bytes. The X32V architecture is Big Endian architecture, thus, the most significant bit in a word or half-word is always in the highest byte. Words are ordered so the most significant bit (MSB) is bit 31, and the least significant bit (LSB) is bit 0.

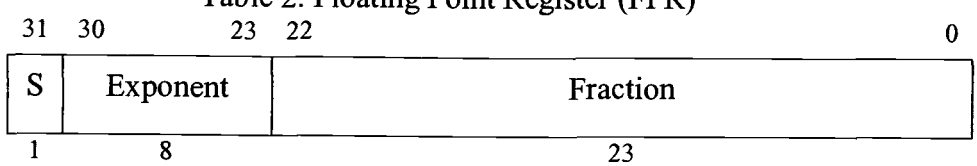
If a value is a signed integer, the MSB is sign extended throughout the rest of the word. Example: see L_H (Load Half-word Signed). If the half-word loaded is 0x82, the value would be sign extended to 0xFF82 and placed in the destination register.

Instructions such as Move Upper Immediate (MOV_Ui) load a 16-bit immediate value into the upper half-word (byte 3 and byte 2). The 'SET' instructions (SET_LT, SET_LTi, etc) place the result of their operation in the LSB of the register (bit 0).

1.3.2 Floating Point Registers (FPRs)

X32V ISA also has 16, 32-bit, FPRs. Floating-point values are stored in the following format.

Table 2: Floating Point Register (FPR)



FPRs can also be used to store double precision numbers (64-bit) by allotting two consecutive registers for each FP number. Registers are then accessed on an even basis (FPR0, FPR2, FPR4, etc). For more information on double precision operations, see the double precision floating point instructions.

1.3.3 Reserved Usage Registers (RURs)

Like many other RISC machines, X32V has several registers reserved for special purposes. The operation of the Program Counter, Link Register, Stack Pointer, and Status Register are outlined below. All RURs are 32-bit registers.

Program Counter: The PC contains the address of the current instruction being fetched from memory. The PC is always incremented by four to retrieve the next instruction word in memory. If the current instruction is a jump or a taken branch, the relative address is given assuming the PC has been pre-incremented.

Link Register: The LR is used to temporarily hold the return address during a J_P or J_PR instruction. Upon return from the procedure call, the return address is loaded into the PC with the return (RET) instruction.

Stack Pointer Register: The SPR holds the memory address of the stack.

Status Register: The SR holds key information about the state of the processor.

Table 3: Byte 0 of SR

7	6	5	4	3	2	1	0
CF	ZF	SF	VF	FMT1	FMT0	PM	IE
		4				4	

IE: The Interrupt Enable bit.

0: External interrupts have no effect on the processor.

1: External interrupts are enabled.

PM: Processor Mode Bit. This bit allows protected modes of operation.

0: User Mode

1: Kernel (Supervisor) Mode

FMT1 & FMT0: Instruction Format Bits. Details the mode the current executing instruction is in.

00: Ultra Light

01: Light

10: Default

11: Unused

VF: Overflow Flag. When an overflow condition is computed by the ALU then this bit is set.

SF: Sign Flag. If the ALU output is negative then this bit is set.

ZF: Zero Flag. Set when a divide by zero condition occurs.

CF: is the Carry bit. This bit is modified by the ALU upon execution of an instruction. The carry bit is set during a carry out, borrow in, or an overflow. If an instruction modifies the carry bit, the condition is outlined in the instruction description.

2 Instruction Set

This section describes the different modes of operation for X32V, and the instruction formats for each of those modes.

2.1 Modes of Operation

X32V supports three different modes of operation.

Mode 1 (Default): 32-bit instruction length

Mode 2 (Light): 32-bit / 16-bit instruction length

Mode 3 (Ultra Light): 32-bit / 24-bit / 16-bit instruction length

All core instructions are available in the default 32-bit mode. However, wherever possible, 32-bit instructions can be condensed into their 24-bit or 16-bit equivalents in modes 2 and 3. This allows minimal code size with little loss in performance. Each instruction has the different formats listed in their detailed description below.

2.2 Instruction Formats

Instruction formats are subdivided into five instruction categories; Load/Store, Immediate, Branch, Register, and Jump/Call.

Table below shows all possible instruction formats for 32-bit, 24-bit, and 16-bit instructions.

Table 4: Instruction Formats

32-bit Instruction Format

	4	4	4	4	16
Load / Store	0000	op1	rd	rs1	disp
Immediate	0001	op1	rd	rs1	imm
Branch	0010	op1	rd	rs1	Label

	4	4	4	4	4	4	4	4
Register	0011	op1	rd	rs1	rs2	op2	op3	op4

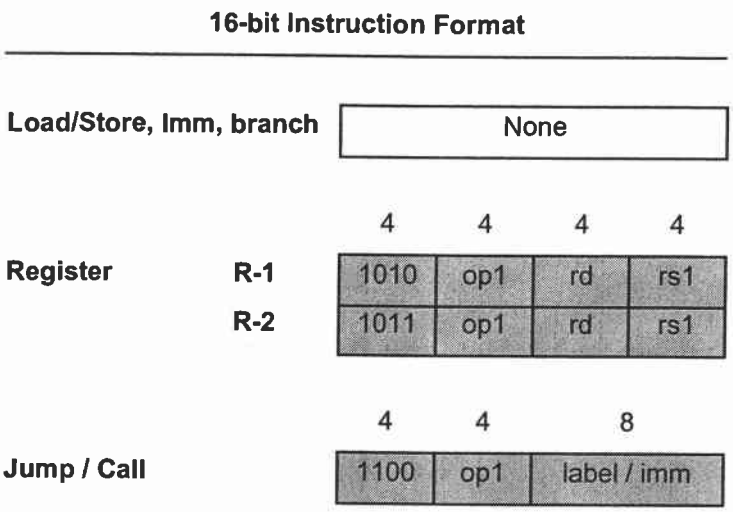
	4	4	24
Jump / Call	0100	op1	label / imm

24-bit Instruction Format

	4	4	4	4	8
Load / Store	0101	op1	rd	rs1	disp
SR Load Store	0101	op1	op2	rs1	disp
Immediate	0110	op1	rd	rs1	imm
Branch	0111	op1	rd	rs1	label

	4	4	4	4	4	4
Register	1000	op1	rd	rs1	rs2	op2

	4	4	16
Jump / Call	1001	op1	label / imm



2.2.1 Load / Store (L/S)

Load and Store instructions operate on two registers and an immediate value. The registers are labeled source (rs) and destination (rd). The immediate value is labeled as a displacement. All L/S instructions undergo the same operation.

1. The displacement (disp) value is sign extended and added to the contents of the source register.
2. The result is used as an effective address to either retrieve from memory (load) or write to memory (store). In the case of a load, data is written to rd, in the case of a store, data is read from rd.

In the 32-bit format, the immediate value is 16-bits. In the 24-bit format, the immediate value is 8-bits. The 16-bit format does not support load/store instructions. The rd value for Status Register load/store instructions are disregarded because it is implicitly known from the instruction.

Table 5: Load and Store Instructions

L_W	Load Word
L_B	Load Byte
L_BU	Load Unsigned Byte
L_H	Load Half-word
L_HU	Load Unsigned Half-word

S_W	Store Word
S_B	Store Byte
S_H	Store Half-word
L_SR	Load Status Register
S_SR	Store Status Register

2.2.2 Immediate (Imm)

Immediate instructions operate on a source register (rs1), a destination register (rd), and an immediate value. Depending upon the instruction, the immediate value is either sign extended (arithmetic operations) or zero padded (logical operations). The immediate value is then combined with the source register contents to produce a result, which is placed in the destination register.

Like the load and store instructions, immediate instructions only come in 32-bit and 24-bit formats shown below.

Table 6: Immediate Instructions

LG_ANDi	Logical AND – Immediate
LG_ORi	Logical OR – Immediate
LG_XORi	Logical Exclusive OR (XOR) - Immediate
ADDi	Signed Integer Addition - Immediate
ADD_Ui	Unsigned Integer Addition - Immediate
SUBi	Signed Integer Subtraction - Immediate
SUB_Ui	Unsigned Integer Subtraction - Immediate
SFT_LLi	Shift Left Logical – Immediate
SFT_RLi	Shift Right Logical – Immediate
SFT_RAi	Shift Right Arithmetic - Immediate
SET_LTi	Set on Less Than – Immediate
SET_LTUi	Unsigned Set on Less Than - Immediate
MOV_UPi	Move Upper – Immediate

2.2.3 Branch (B)

Branch instructions compare either two source registers (rs1, rs2) or a source register (rs1) and zero. The immediate value is sign extended and added to the pre-incremented PC. If the comparison is true, the branch is taken and the PC is updated with the new value. If the comparison is false, the branch is not taken, and the next instruction is executed.

Branches, like Immediate and Load/Store, only come in 32-bit and 24-bit formats.

Table 7: Branch Instructions

B_EQ	Branch if Equal
B_NE	Branch if Not Equal
B_EQZ	Branch if Equal to Zero
B_NEZ	Branch if Not Equal to Zero
B_LTZ	Branch if Less Than Zero
B_GTZ	Branch if Greater Than Zero
B_LTEZ	Branch if Less Than or Equal to Zero
B_GTEZ	Branch if Greater Than or Equal to Zero

2.2.4 Register (R)

Register instructions operate on two source registers (rs1, rs2) and place their result in a destination register (rd). The format is the same for both GPRs and FPRs, but the op-codes differ. Instructions J_AR and J_PR are considered register instructions (as opposed to jump instructions) because of their similarity in instruction format to a register instruction.

Table 8: Register Instructions

LG_AND	Logical AND
LG_OR	Logical OR
LG_XOR	Logical Exclusive OR (XOR)
ADD	Signed Integer Addition
ADD_U	Unsigned Integer Addition
SUB	Signed Integer Subtraction
SUB_U	Unsigned Integer Subtraction

SFT_LL	Shift Left Logical
SFT_RL	Shift Right Logical
SFT_RA	Shift Right Arithmetic
SET_LT	Set on Less Than
SET_LTU	Unsigned Set on Less Than
MUL	Signed Integer Multiplication
MUL_U	Unsigned Integer Multiplication
DIV	Signed Integer Division
DIV_U	Unsigned Integer Division
J_AR	Absolute Jump Register
J_PR	Procedural Jump Register
MOV_	Move
PUSH	Push a register onto stack
POP	Pop value from stack

2.2.5 Jump / Call (J/C)

The Jump and Call instructions are lumped together because of their similarity in format, though dissimilar in operation. All of the various Jumps and Calls behave somewhat uniquely. As such, one can gain an understanding by reading the detailed instruction descriptions in Section 3.

Table 9: Jump and Call instructions

J_A	Absolute Jump
J_P	Procedural Jump

3 Instructions

3.1 Introduction

The following pages outline each instruction in the X32V ISA. Detailed information about instruction type, format, usage, and encoding are given.

Table 10: Symbol Definition

=	Substitute left side of operator with right side
+	Addition
-	Subtraction
*	Multiplication
/	Division
==	Test equality
!=	Test inequality
>	Greater Than
<	Less Than
&	Bit wise Logical AND
|	Bit wise Logical OR
^	Bit wise Logical XOR
||	Join or Concatenate
<<	Bit wise Shift Left
>>	Bit wise Shift Right
rs1	Source register one
rs2	Source register two
rd	Destination register
MEM(0x2a)	Value at main memory address 0x2a
$'0'^{8}$	Zero extended 8 places
$'imm_{15}'^{16}$	15^{th} bit of immediate value, sign extended 16 places

Logical AND (LG_AND):

Description:

A bit wise logical AND is performed on the contents of GPR rs1 and GPR rs2. The result of the operation is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

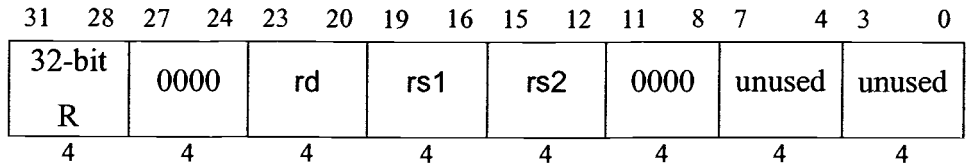
Format:

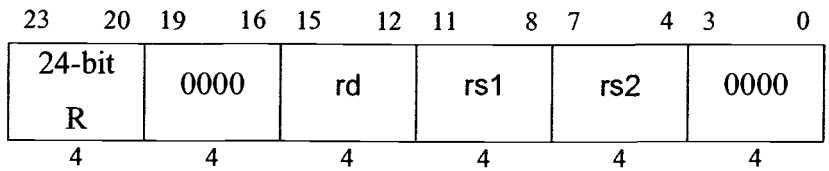
32-bit / 24-bit	*16-bit*
LG_AND rd, rs1, rs2	LG_AND rd, rs

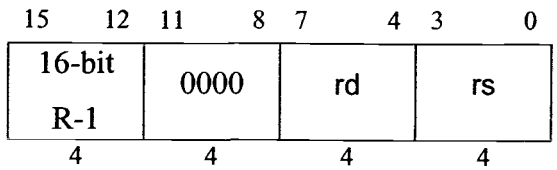
Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 & rs2	rd = rd & rs

Encoding:







Logical OR (LG_OR):

Description:

A bit wise logical OR is performed on the contents of GPR rs1 and GPR rs2. The result of the operation is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

Format:

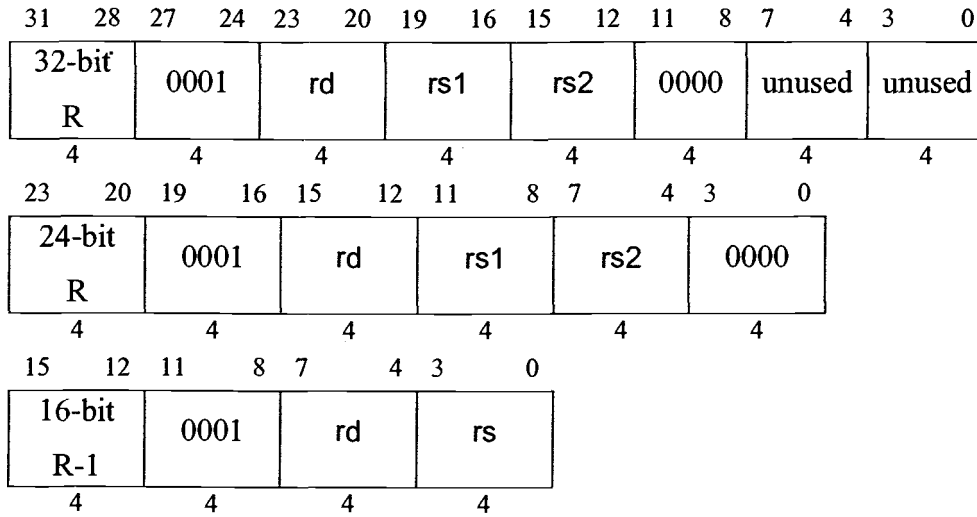
32-bit / 24-bit	*16-bit*
LG_OR rd, rs1, rs2	LG_OR rd, rs

Operation:

32-bit / 24-bit	*16-bit*

rd = rs1 | rs2 rd = rd | rs

Encoding:



Logical Exclusive OR (LG_XOR):

Description:

A bit wise logical XOR is performed on the contents of GPR rs1 and GPR rs2. The result of the operation is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

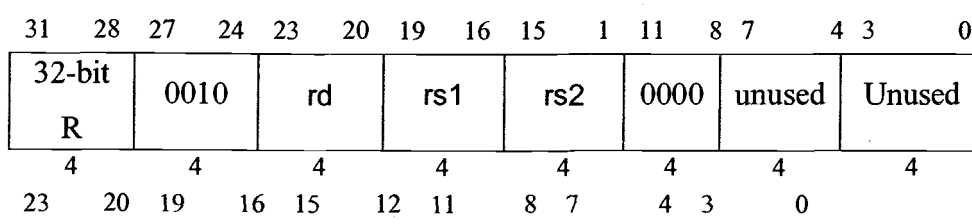
Format:

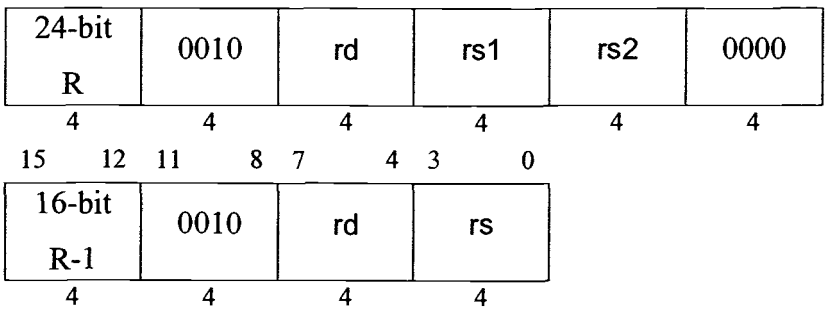
32-bit / 24-bit	*16-bit*
LG_XOR rd, rs1, rs2	LG_XOR rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 ^ rs2	rd = rd ^ rs

Encoding:





Signed Integer Addition (ADD):

Description:

The contents of GPR rs1 and GPR rs2 are arithmetically added. The result of the operation (in two's complement format) is placed in GPR rd. An overflow exception occurs if the result of the operation is greater than $2^{32} - 1$. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

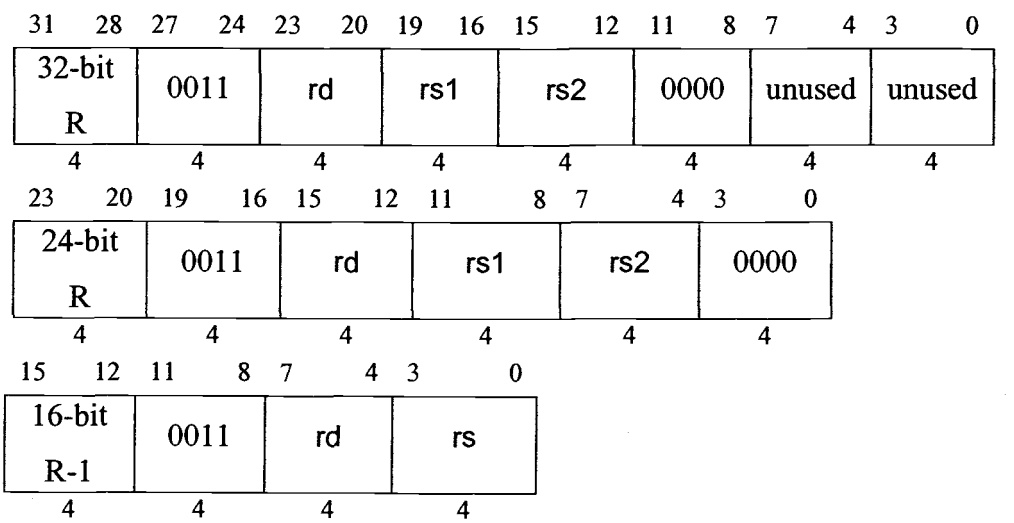
Format:

32-bit / 24-bit	*16-bit*
ADD rd, rs1, rs2	ADD rd, rs1

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 + rs2	rd = rd + rs

Encoding:



Unsigned Integer Addition (ADD_U):

Description:

The contents of GPR rs1 and GPR rs2 are arithmetically added. The result of the operation is placed in GPR rd. No overflow exception will occur with this instruction. (See the ADD instruction for signed addition) When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

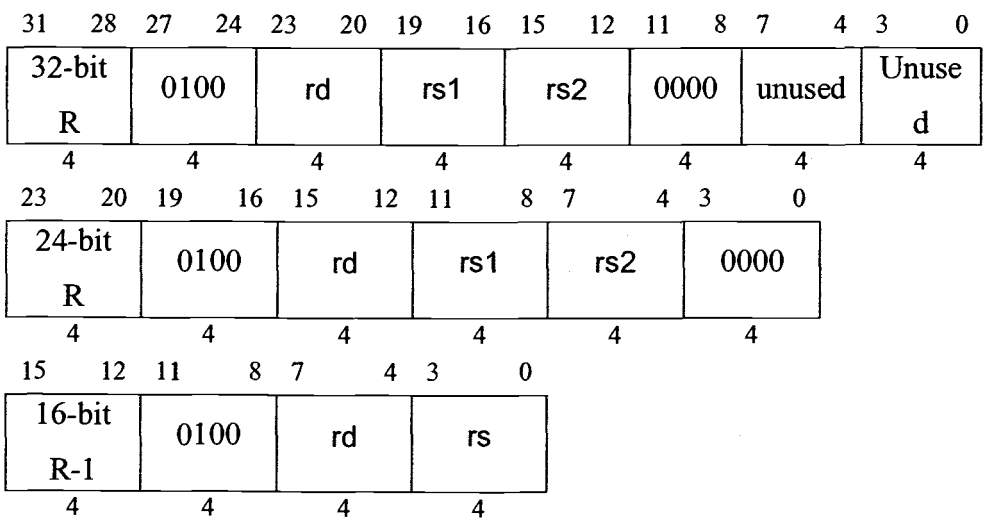
Format:

32-bit / 24-bit	*16-bit*
ADD_U rd, rs1, rs2	ADD_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 + rs2	rd = rd + rs

Encoding:



Signed Integer Subtraction (SUB):

Description:

The contents of GPR rs1 and GPR rs2 are arithmetically subtracted. The result of the operation (in two's complement format) is placed in GPR rd. An overflow exception occurs if the result of the operation is less than -2^{31}. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

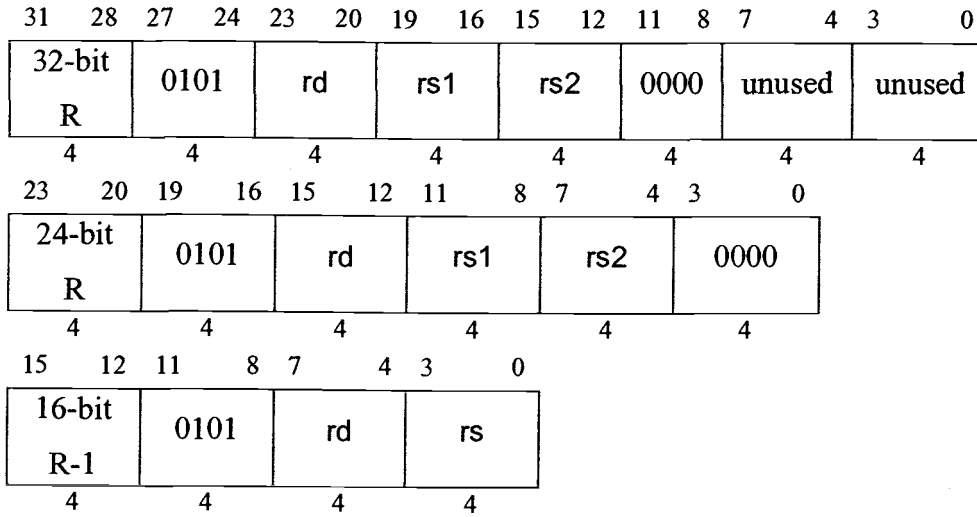
Format:

32-bit / 24-bit	*16-bit*
SUB rd, rs1, rs2	SUB rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 - rs2	rd = rd - rs

Encoding:



Unsigned Integer Subtraction (SUB_U):

Description:

The contents of GPR rs1 and GPR rs2 are arithmetically subtracted. The result of the operation is placed in GPR rd. No overflow exception will occur with this instruction. (See the SUB instruction for signed subtraction) When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

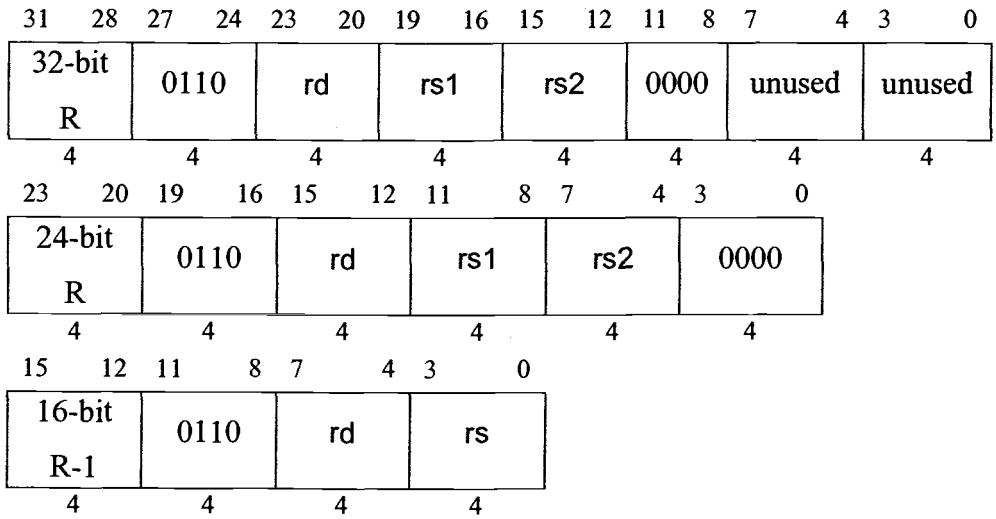
Format:

32-bit / 24-bit	*16-bit*
SUB_U rd, rs1, rs2	SUB_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 - rs2	rd = rd - rs

Encoding:



Shift Left Logical (SFT_LL):

Description:

The contents of GPR rs1 are shifted left a variable amount corresponding to the least significant five bits in GPR rs2. Zeros are inserted into the shifted locations (least significant bits), and the result is stored in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

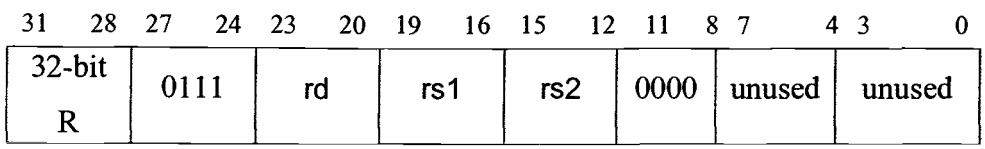
Format:

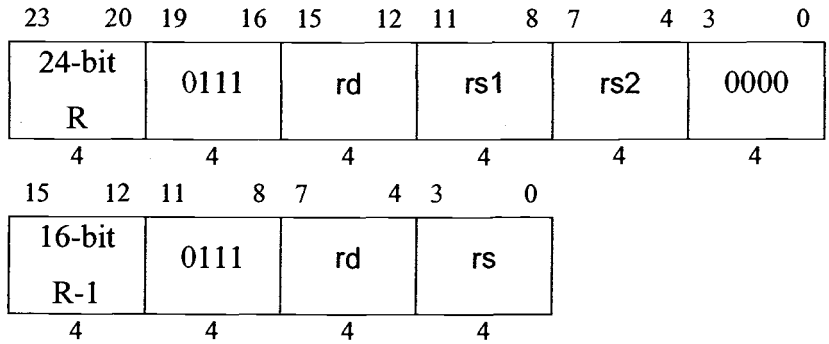
32-bit / 24-bit	*16-bit*
SFT_LL rd, rs1, rs2	SFT_LL rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 << rs2	rd = rd << rs

Encoding:





Shift Right Logical (SFT_RL):

Description:

The contents of GPR rs1 are shifted right a variable amount corresponding to the least significant five bits in GPR rs2. Zeros are inserted into the shifted locations (most significant bits), and the result is stored in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

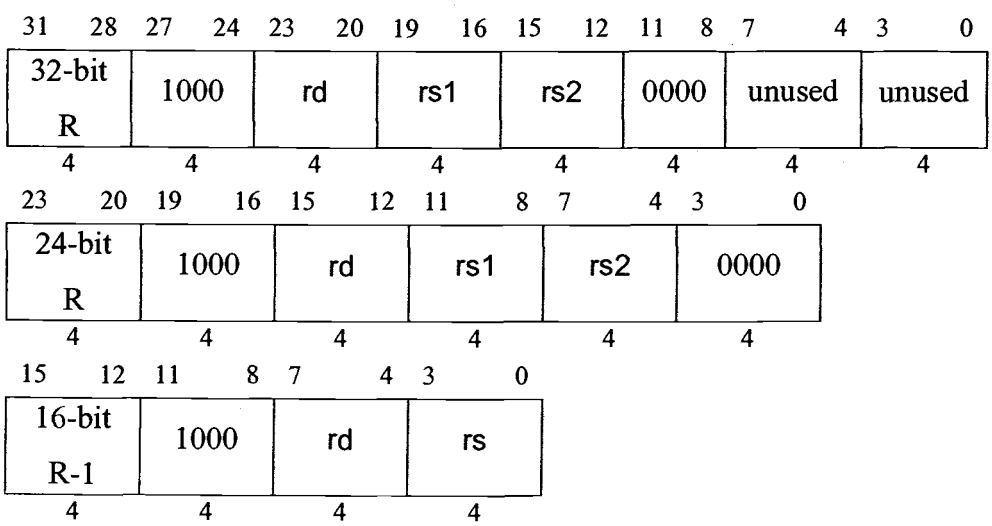
Format:

32-bit / 24-bit	*16-bit*
SFT_RL rd, rs1, rs2	SFT_RL rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 >> rs2	rd = rd >> rs

Encoding:



Shift Right Arithmetic (SFT_RA):

Description:

The contents of GPR rs1 are shifted right a variable amount corresponding to the least significant five bits in GPR rs2. The most significant bits are then sign extended (rather than padded with zeros, see SFT_RL), and the result is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

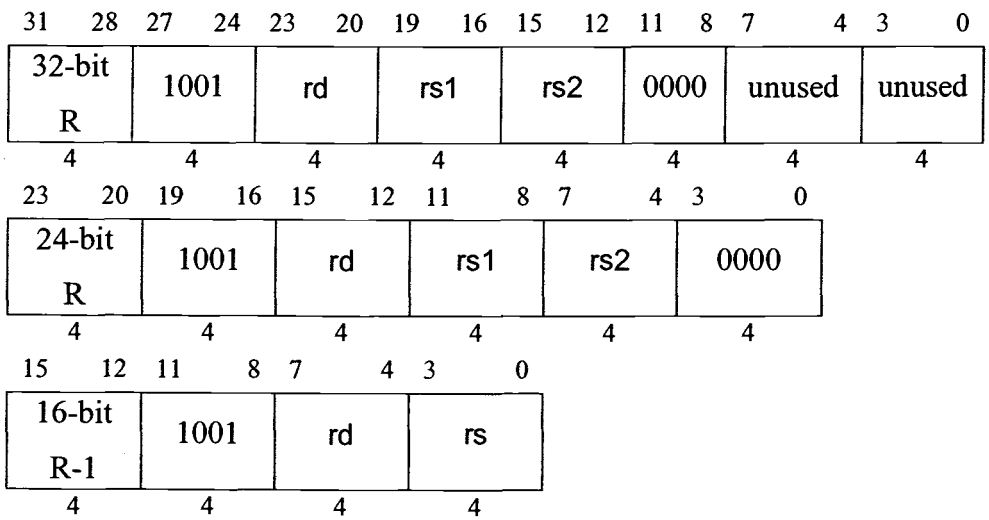
Format:

32-bit / 24-bit	*16-bit*
SFT_RA rd, rs1, rs2	SFT_RA rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 >> rs2	rd = rd >> rs

Encoding:



Set on Less Than (SET_LT):

Description:

The contents of GPR rs1 and GPR rs2 are compared as two's complement integers. If GPR rs1 is less than GPR rs2, then GPR rd is set to '1'. Otherwise,

GPR rd is set to '0'. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

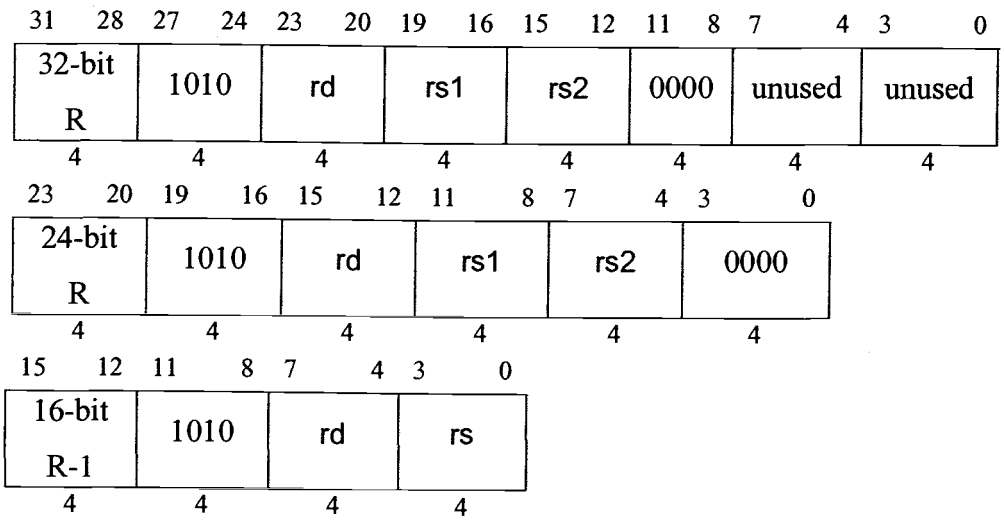
Format:

32-bit / 24-bit	*16-bit*
SET_LT rd, rs1, rs2	SET_LT rd, rs

Operation:

32-bit / 24-bit	*16-bit*
if (rs1 < rs2)	if (rd < rs)
rd = 1	rd = 1
else	else
rd = 0	rd = 0

Encoding:



Unsigned Set on Less Than (SET_LTU):

Description:

The contents of GPR rs1 and GPR rs2 are compared as positive integers. If GPR rs1 is less than GPR rs2, then GPR rd is set to '1'. Otherwise, GPR rd is set to '0'. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

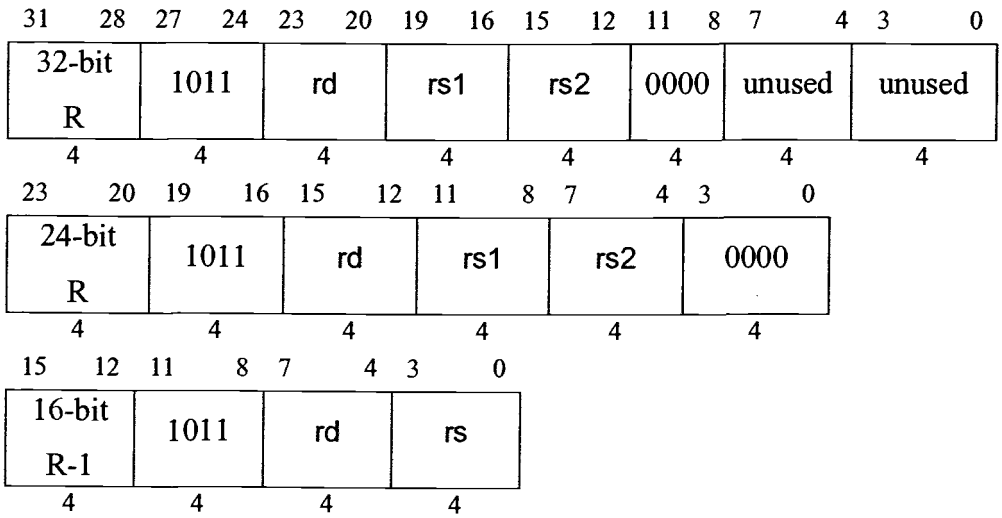
Format:

32-bit / 24-bit	*16-bit*
SET_LTU rd, rs1, rs2	SET_LTU rd, rs

Operation:

32-bit / 24-bit	*16-bit*
if (rs1 < rs2)	if (rd < rs)
rd = 1	rd = 1
else	else
rd = 0	rd = 0

Encoding:



Signed Integer Multiplication (MUL):

Description:

The contents of GPR rs1 are multiplied by the contents of GPR rs2 using two's complement format. The least significant word of the result is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

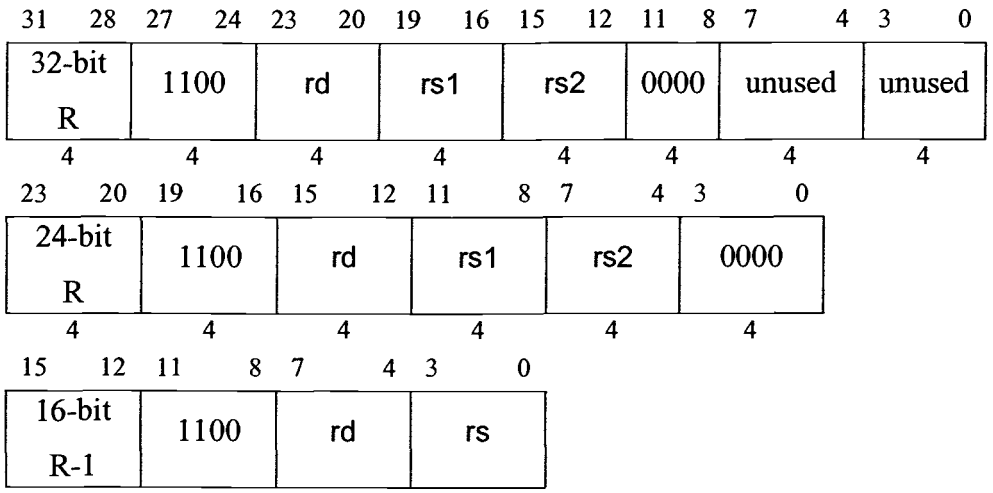
Format:

32-bit / 24-bit	*16-bit*
MUL rd, rs1, rs2	MUL rd, rs

Operation:

32-bit / 24-bit	*16-bit*
$rd = (rs1 * rs2)_{31-0}$	$rd = (rs1 * rs2)_{31-0}$

Encoding:



Unsigned Integer Multiplication (MUL_U):

Description:

The contents of GPR rs1 are multiplied by the contents of GPR rs2. Register contents are all positive values. The least significant word of the result is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

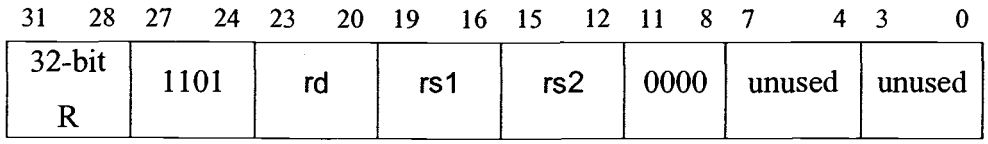
Format:

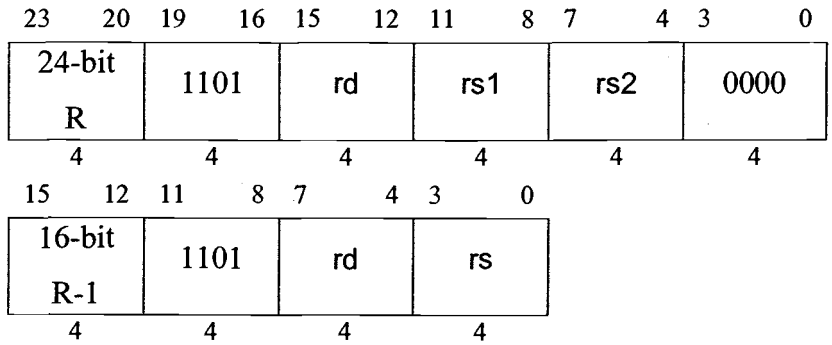
32-bit / 24-bit	*16-bit*
MUL_U rd, rs1, rs2	MUL_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 * rs2	rd = rd * rs

Encoding:





Signed Integer Division (DIV):

Description:

The contents of GPR rs1 are divided by the contents of GPR rs2 using two's complement format. The quotient is rounded toward zero and placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

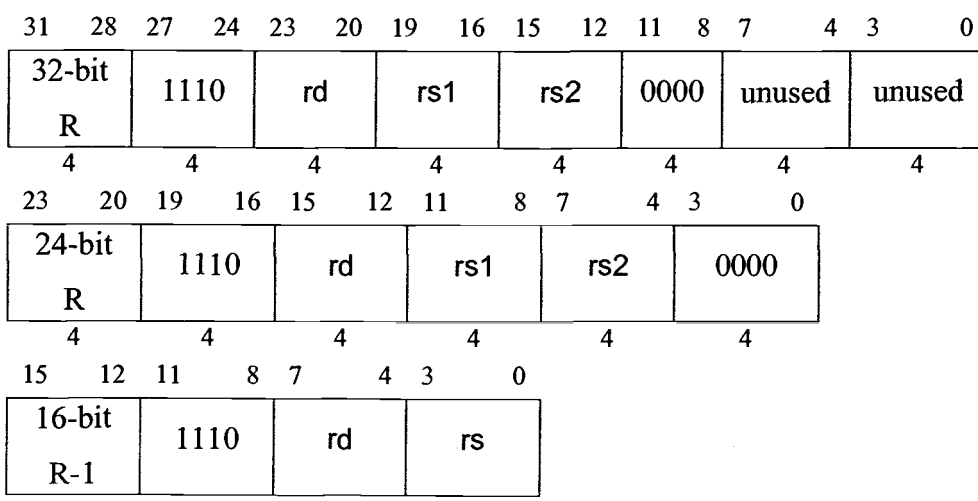
Format:

32-bit / 24-bit	*16-bit*
DIV rd, rs1, rs2	DIV rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 / rs2	rd = rd / rs

Encoding:



Unsigned Integer Division (DIV_U):

Description:

The contents of GPR rs1 are divided by the contents of GPR rs2. Register contents are all treated as positive integers. The quotient is rounded toward zero and placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type: Register

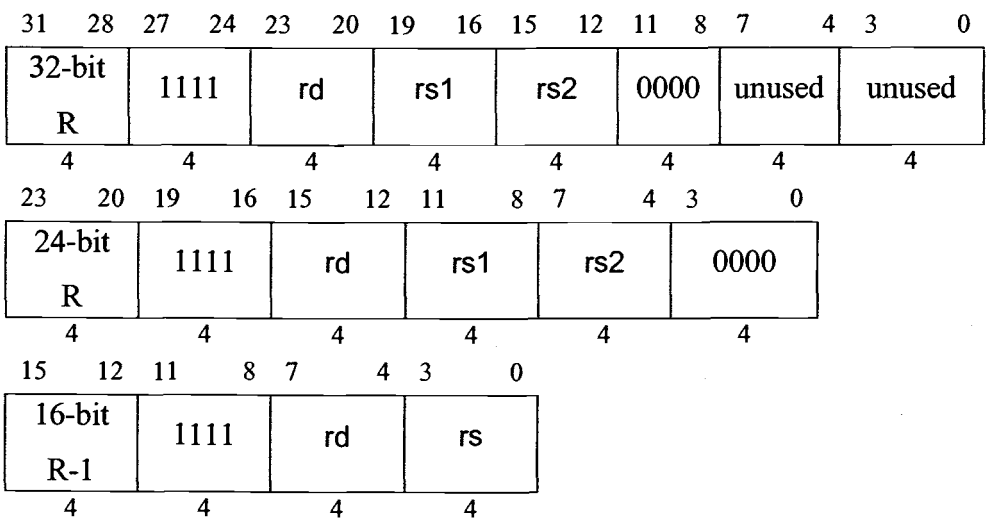
Format:

32-bit / 24-bit	*16-bit*
DIV_U rd, rs1, rs2	DIV_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 / rs2	rd = rd / rs

Encoding:



Move (MOV):

Description:

The contents of GPR rs are copied into GPR rd. The contents of GPR rs do not change.

Type: Register

Format:

32-bit / 24-bit / 16-bit

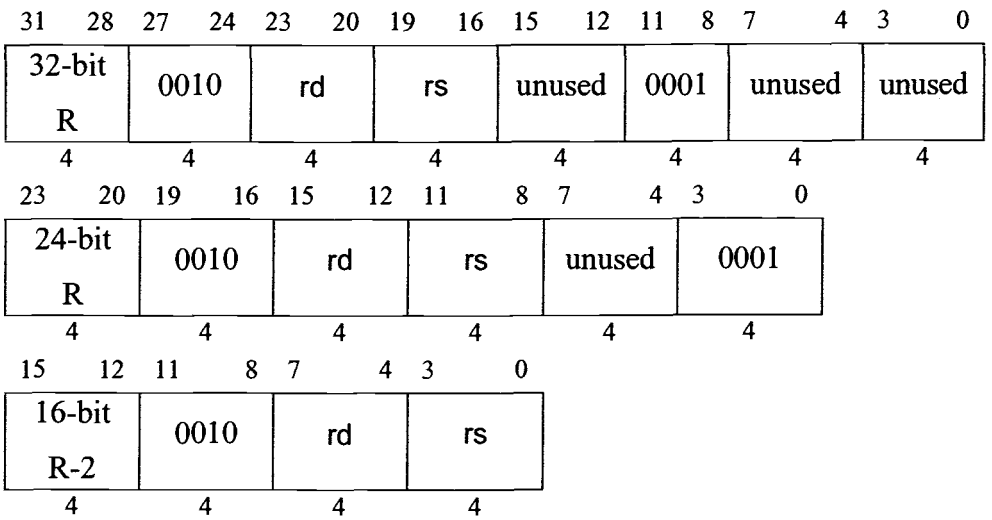
MOV rd, rs

Operation:

32-bit / 24-bit / 16-bit

rd = rs

Encoding:



Load Word (L_W):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The word in memory at this address is copied into GPR rd.

Type: Load / Store

Format:

32-bit / 24-bit

L_W rd, disp(rs1)

Operation:

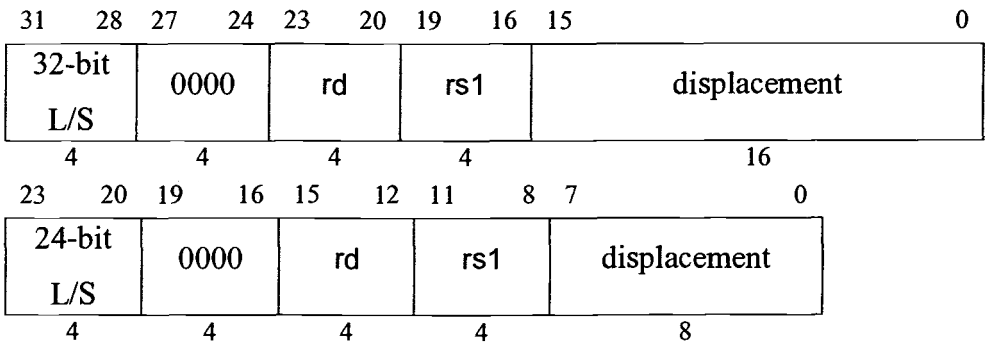
32-bit

rd = MEM (rs1 + ('disp$_{15}$' ^{16} || disp))

24-bit

rd = MEM (rs1 + ('disp$_{7}$' ^{24} || disp))

Encoding:



Load Byte (L_B):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The byte in memory at this address is sign extended and copied into GPR rd.

Type: Load / Store

Format:

32-bit / 24-bit

L_B rd, disp(rs1)

Operation:

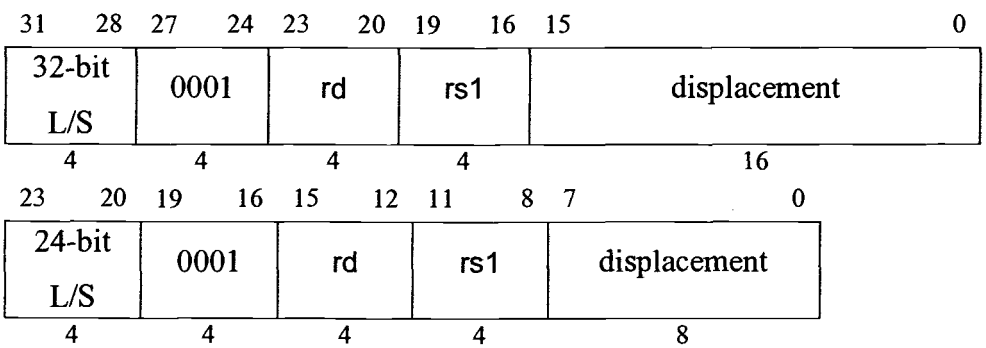
32-bit

rd = MEM (rs1 + ('disp$_{15}$'^{16} || disp))

24-bit

rd = MEM (rs1 + ('disp$_{7}$'^{24} || disp))

Encoding:



Load Unsigned Byte (L_BU):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The byte in memory at this address is copied into the least significant byte of GPR rd. The upper 3-bytes of GPR rd are padded with zeros.

Type: Load / Store

Format:

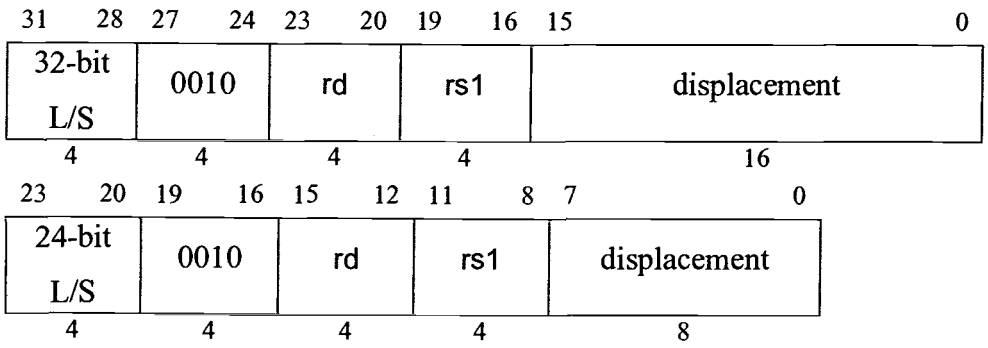
32-bit / 24-bit

L_BU rd, disp(rs1)

Operation:

32-bit	*24-bit*
rd = MEM (rs1 + ('$disp_{15}$' ^{16} || disp))	rd = MEM (rs1 + ('$disp_7$' ^{24} || disp))

Encoding:



Load Half-word (L_H):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The half-word in memory at this address is sign extended and copied into GPR rd.

Type: Load / Store

Format:

32-bit / 24-bit

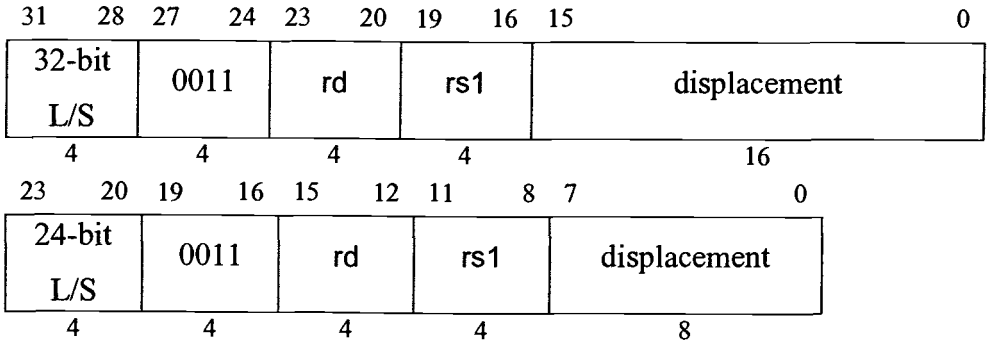
L_H rd, disp(rs1)

Operation:

32-bit *24-bit*

rd = MEM (rs1 + ('disp$_{15}$'^{16} || disp)) rd = MEM (rs1 + ('disp$_7$'^{24} || disp))

Encoding:



Load Unsigned Half-word (L_HU):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The half-word in memory at this address is copied into the lower half of GPR rd. The upper half of GPR rd is padded with zeros.

Type: Load / Store

Format:

32-bit / 24-bit

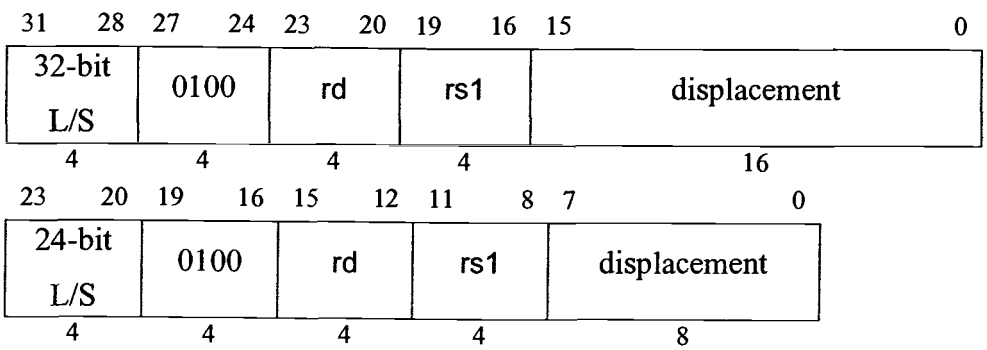
L_HU rd, disp(rs1)

Operation:

32-bit *24-bit*

rd = MEM (rs1 + ('disp$_{15}$'^{16} || disp)) rd = MEM (rs1 + ('disp$_7$'^{24} || disp))

Encoding:



Store Word (S_W):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The word in GPR rd is then copied to the memory address.

Type: Load / Store

Format:

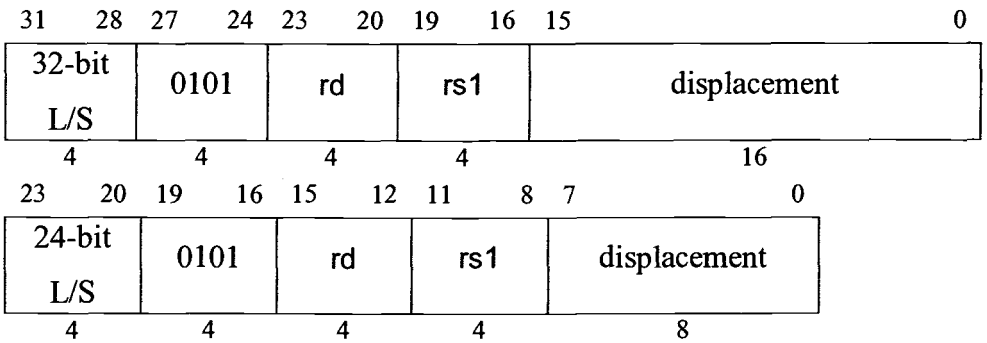
32-bit / 24-bit

S_W disp(rs1), rd

Operation:

32-bit	*24-bit*
MEM (rs1 + ('$disp_{15}$'^{16} || disp)) = rd	MEM (rs1 + ('$disp_7$'^{24} || disp)) = rd

Encoding:



Store Byte (S_B):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The lowest byte (byte 0) in GPR rd is then copied to the memory address.

Type: Load / Store

Format:

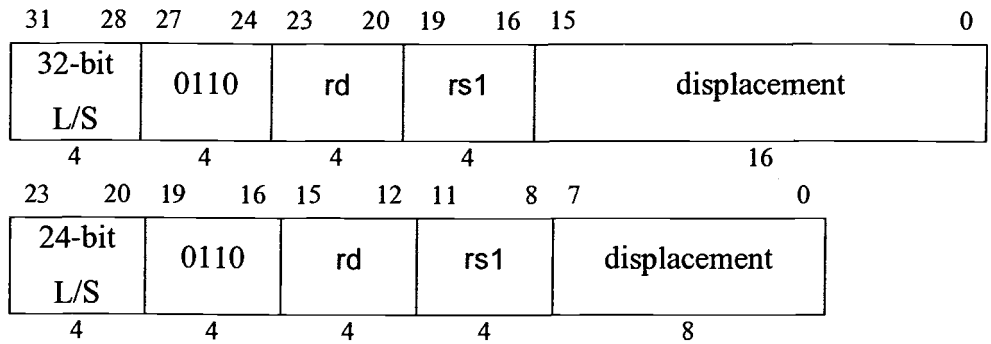
32-bit / 24-bit

S_B disp(rs1), rd

Operation:

32-bit	*24-bit*
MEM (rs1 + ('$disp_{15}$'^{16} || disp)) = rd	MEM (rs1 + ('$disp_7$'^{24} || disp)) = rd

Encoding:



Store Half-word (S_H):

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The lower half-word (byte 1 - byte 0) in GPR rd is then copied to the memory address.

Type: Load / Store

Format:

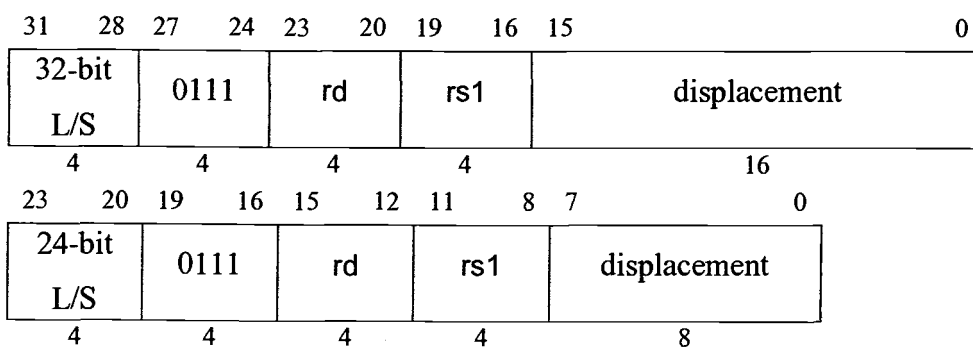
32-bit / 24-bit

S_H disp(rs1), rd

Operation:

32-bit	*24-bit*
MEM (rs1 + ('$disp_{15}$'^{16} || disp)) = rd	MEM (rs1 + ('$disp_7$'^{24} || disp)) = rd

Encoding:



Logical AND – Immediate (LG_ANDi):

Description:

The immediate value is zero extended and a bit wise logical AND is performed on the contents of GPR rs1 and the extended immediate. The result of the operation is placed in GPR rd.

Type: Immediate

Format:

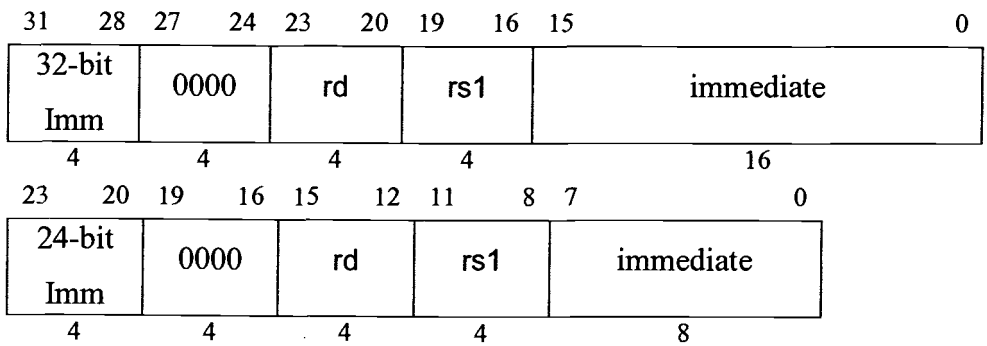
32-bit / 24-bit

LG_ANDi rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 & ('0'^{16} || imm)	rd = rs1 & ('0'^{24} || imm)

Encoding:



Logical OR – Immediate (LG_ORi):

Description:

The immediate value is zero extended and a bit wise logical OR is performed on the contents of GPR rs1 and the extended immediate. The result of the operation is placed in GPR rd.

Type: Immediate

Format:

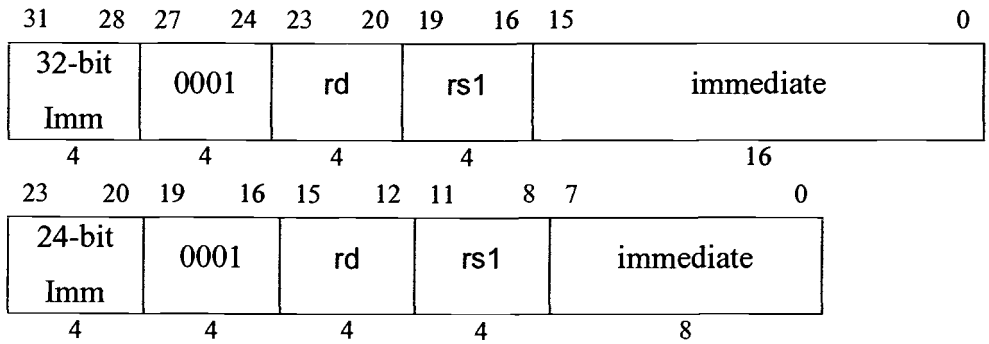
32-bit / 24-bit

LG_ORi rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 | ('0'^{16} || imm)	rd = rs1 | ('0'^{24} || imm)

Encoding:



Logical XOR – Immediate (LG_XORi):

Description:

The immediate value is zero extended and a bit wise logical XOR is performed on the contents of GPR rs1 and the extended immediate. The result of the operation is placed in GPR rd.

Type: Immediate

Format:

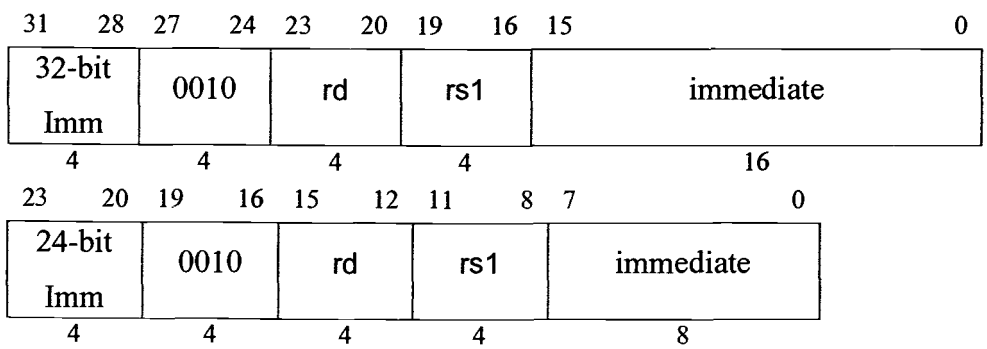
32-bit / 24-bit

LG_XORi rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 ^ ('0'^{16} || imm)	rd = rs1 ^ ('0'^{24} || imm)

Encoding:



Signed Integer Addition – Immediate (ADDi):

Description:

The immediate value is sign extended and added to GPR rs1. The result of the operation (in two's complement format) is placed in GPR rd. An overflow exception occurs if the result of the operation is greater than $2^{32} - 1$.

Type: Immediate

Format:

32-bit / 24-bit

ADDi rd, rs1, imm

Operation:

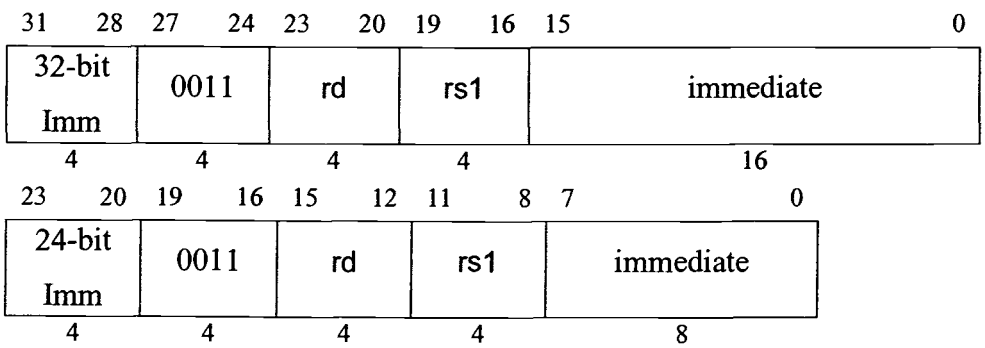
32-bit

rd = rs1 + ('imm$_{15}$'^{16} || imm)

24-bit

rd = rs1 + ('imm$_{7}$'^{24} || imm)

Encoding:



Unsigned Integer Addition – Immediate (ADD_Ui):

Description:

The immediate value is padded with zeros and added to GPR rs1. The result of the operation is placed in GPR rd. No overflow exception will occur with this instruction (see ADDi).

Type: Immediate

Format:

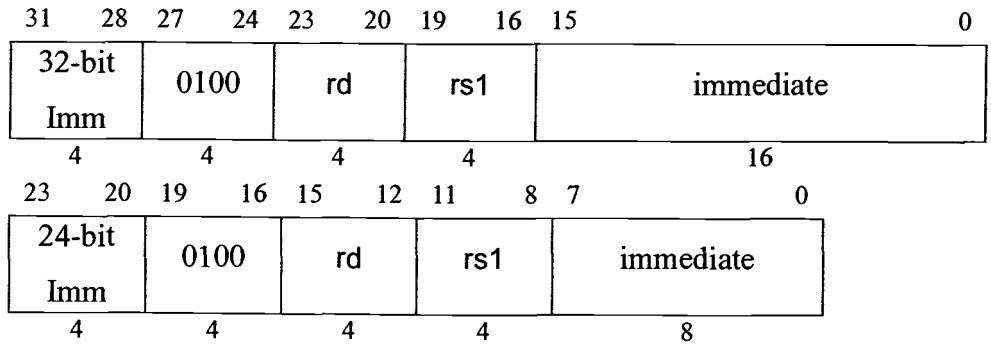
32-bit / 24-bit

ADD_Ui rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 + ('0'^{16} || imm)	rd = rs1 + ('0'^{24} || imm)

Encoding:



Signed Integer Subtraction – Immediate (SUBi):

Description:

The immediate value is sign extended and subtracted from GPR rs1. The result of the operation (in two's complement format) is placed in GPR rd. An overflow exception occurs if the result of the operation is greater than 2^{31}.

Type: Immediate

Format:

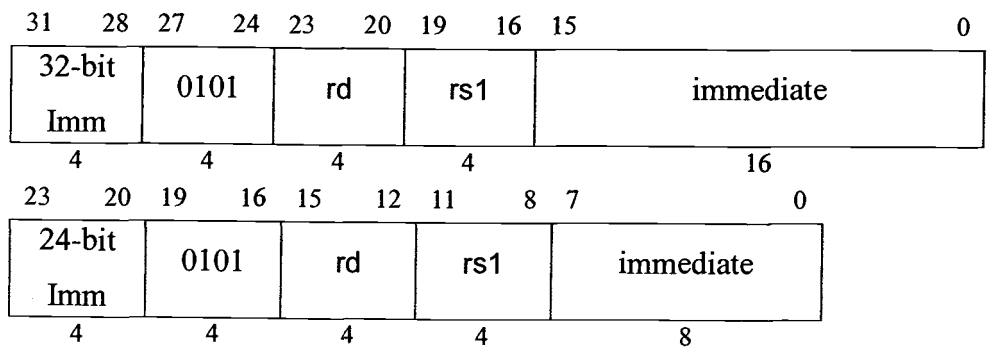
32-bit / 24-bit

SUBi rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 - ('imm$_{15}$'^{16} || imm)	rd = rs1 - ('imm$_7$'^{24} || imm)

Encoding:



Unsigned Integer Subtraction – Immediate (SUB_Ui):

Description:

The immediate value is padded with zeros and subtracted from GPR rs1. The result of the operation is placed in GPR rd. No overflow exception will occur with this instruction (see SUBi).

Type: Immediate

Format:

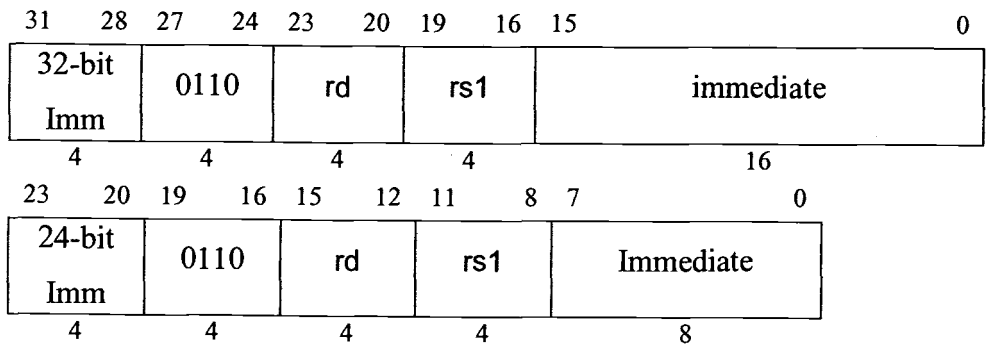
32-bit / 24-bit

SUB_Ui rd, rs1, imm

Operation:

32-bit	*24-bit*
rd = rs1 - ('0'^{16} || imm)	rd = rs1 - ('0'^{24} || imm)

Encoding:



Shift Left Logical – Immediate (SFT_LLi):

Description:

The contents of GPR rs1 are shifted left a variable amount corresponding to the least significant five bits of the immediate value. Zeros are inserted into the shifted locations (least significant bits), and the result is stored in GPR rd.

Type: Immediate

Format:

32-bit / 24-bit

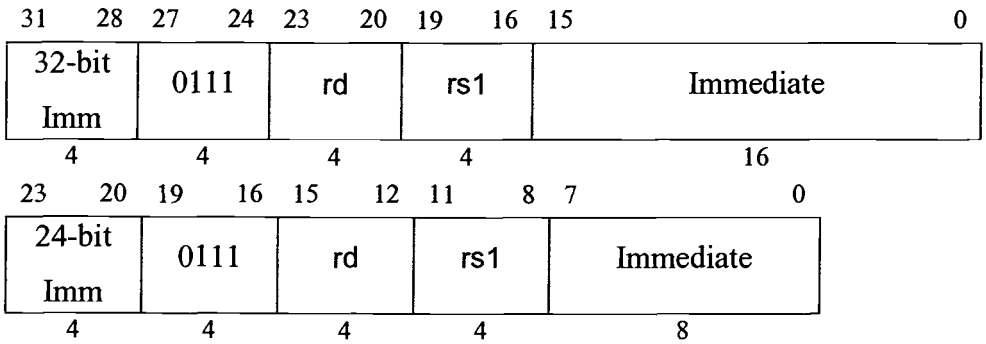
SFT_LLi rd, rs1, imm

Operation:

32-bit / 24-bit

rd = rs1 << imm_{4-0}

Encoding:



Shift Right Logical – Immediate (SFT_RLi):

Description:

The contents of GPR rs1 are shifted right a variable amount corresponding to the least significant five bits of the immediate value. Zeros are inserted into the shifted locations (most significant bits), and the result is stored in GPR rd.

Type: Immediate

Format:

32-bit / 24-bit

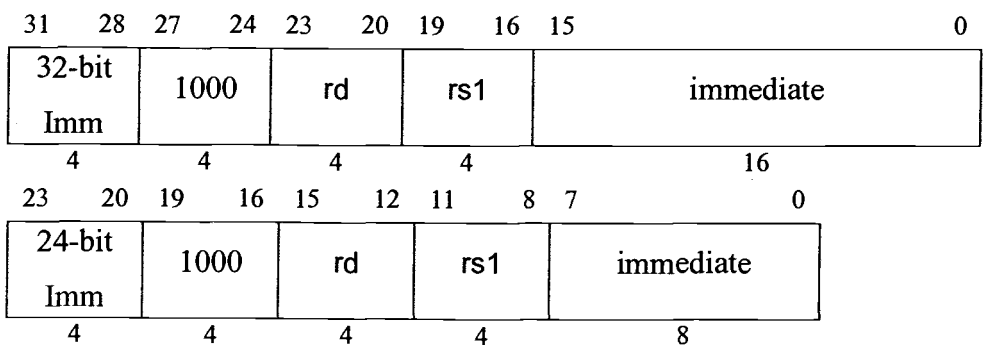
SFT_RLi rd, rs1, imm

Operation:

32-bit / 24-bit

rd = rs1 >> imm_{4-0}

Encoding:



Shift Right Arithmetic – Immediate (SFT_RAi):

Description:

The contents of GPR rs1 are shifted right a variable amount corresponding to the least significant five bits of the immediate value. The most significant bits of GPR rs1 are then sign extended (rather than padded with zeros, see SFT_RLi), and the result is placed in GPR rd.

Type: Immediate

Format:

32-bit / 24-bit

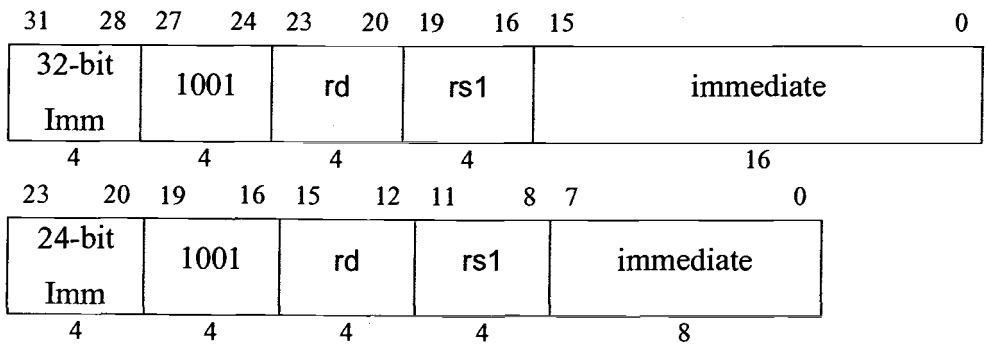
SFT_RAi rd, rs1, imm

Operation:

32-bit / 24-bit

$rd = rs1 >> imm_{4-0}$

Encoding:



Set on Less Than – Immediate (SET_LTi):

Description:

The contents of GPR rs1 are compared to the sign extended immediate value as two's complement integers. If GPR rs1 is less than immediate value, then GPR rd is set to '1'. Otherwise, GPR rd is set to '0'.

Type: Immediate

Format:

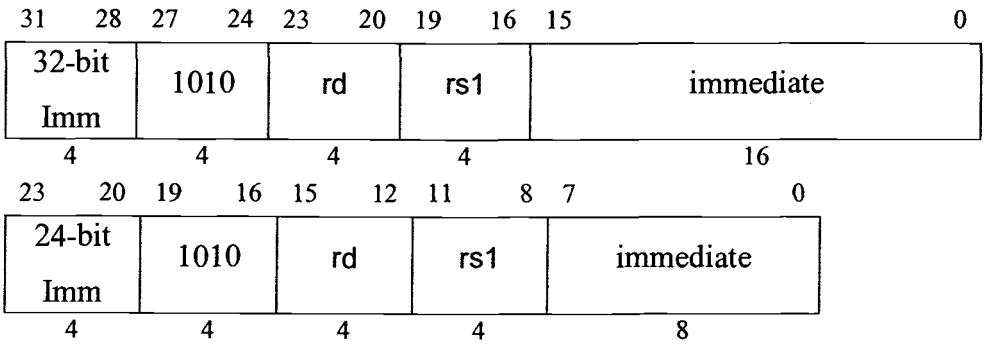
32-bit / 24-bit

SET_LTi rd, rs1, imm

Operation:

32-bit	*24-bit*
if (rs1 < ('imm$_{15}$'^{16} || imm))	if (rs1 < ('imm$_{7}$'^{24} || imm))
rd = 1	rd = 1
else	else
rd = 0	rd = 0

Encoding:



Unsigned Set on Less Than – Immediate (SET_LTUi):

Description:

The contents of GPR rs1 are compared to the immediate value (MSBs padded with zeros) as positive integers. If GPR rs1 is less than immediate value, then GPR rd is set to '1'. Otherwise, GPR rd is set to '0'.

Type: Immediate

Format:

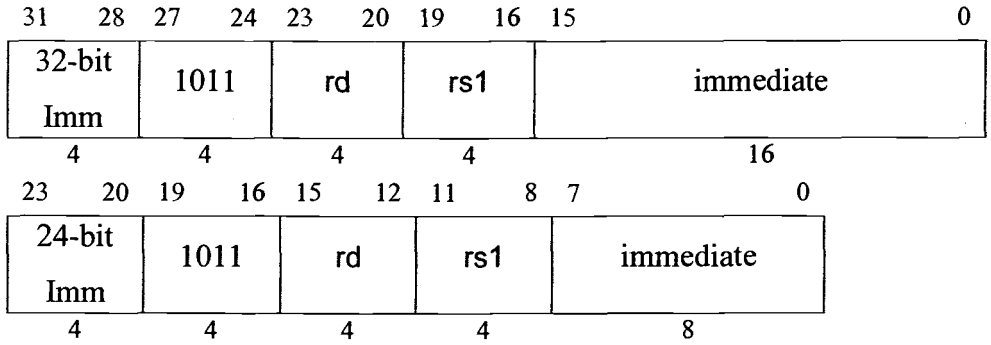
32-bit / 24-bit

SET_LTUi rd, rs1, imm

Operation:

32-bit	*24-bit*
if (rs1 < ('0'^{16} || imm))	if (rs1 < ('0'^{24} || imm))
rd = 1	rd = 1
else	else
rd = 0	rd = 0

Encoding:



Move Upper – Immediate (MOV_UPi):

Description:

The immediate value is stored in GPR rd with the least significant 16-bits padded with zero. If a 24-bit format is used, the least significant 24-bits are padded with zero.

Type: Immediate

Format:

32-bit / 24-bit

MOV_UPi rd, imm

Operation:

32-bit

rd = imm || '0'^{16}

24-bit

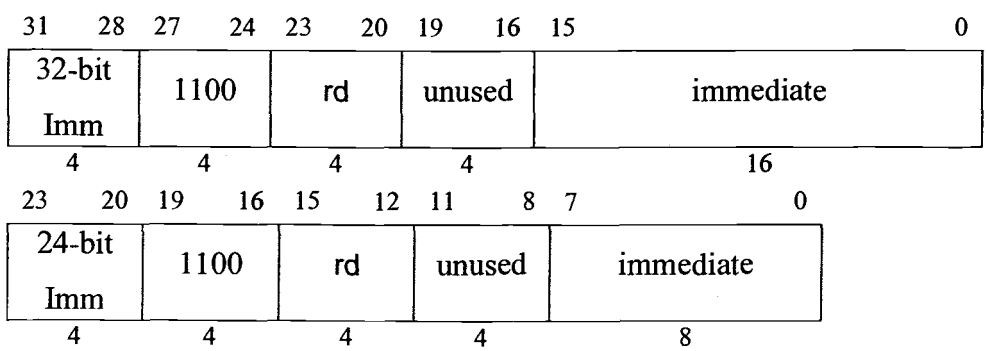
rd = imm || '0'^{24}

MOV_UPi rd, imm

Operation:

rd = imm || '0'^{16}

Encoding:



Branch if Equal (B_EQ):

Description:

The contents of GPR rs1 and GPR rs2 are compared, if they are *not* equal, the PC is incremented by 4 and the next instruction is fetched. If they *are* equal, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address.

Type: Branch

Format:

32-bit / 24-bit

B_EQ rs2, rs1, label

Operation:

32-bit

if (rs1 == rs2)

  PC = PC + 4 + '$label_{15}$'^{16} || label

else

  PC = PC + 4

24-bit

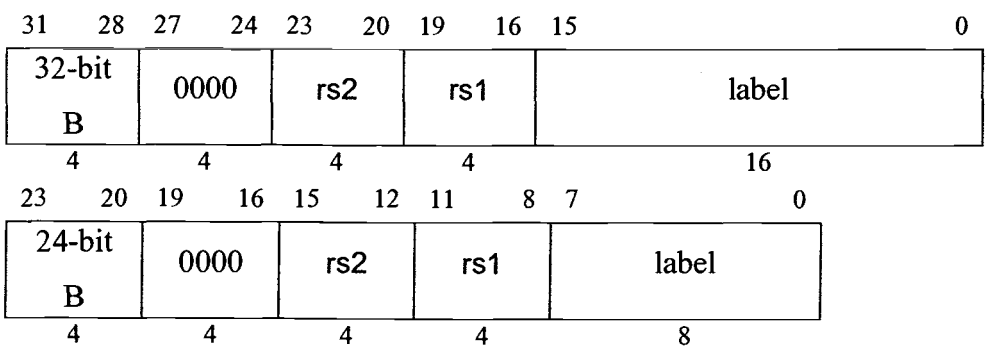
if (rs1 == rs2)

  PC = PC + 4 + '$label_{7}$'^{24} || label

else

  PC = PC + 4

Encoding:



Branch if Not Equal (B_NE):

Description:

The contents of GPR rs1 and GPR rs2 are compared. If they are *not* equal, the PC is incremented by 4 and added to the sign extended immediate value. The next

instruction is then fetched from the newly calculated address. If they *are* equal, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_NE rs2, rs1, label

Operation:

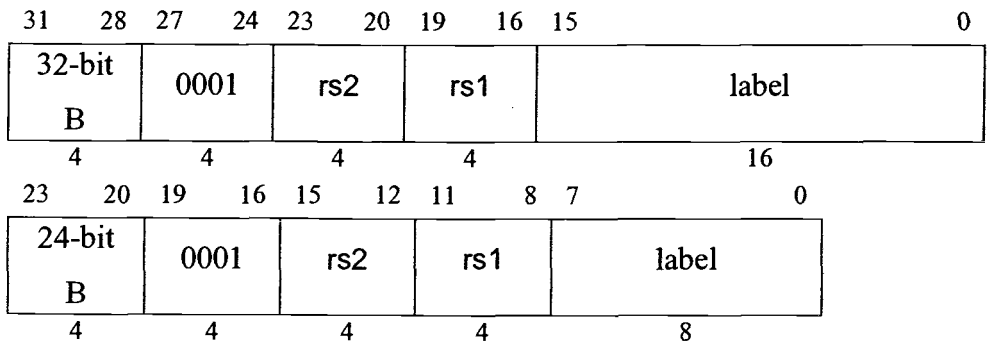
32-bit

if (rs1 != rs2)
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 != rs2)
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:



Branch if Equal to Zero (B_EQZ):

Description:

The contents of GPR rs1 are compared to zero, if they *are* equal, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If they are *not* equal, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_EQZ rs1, label

Operation:

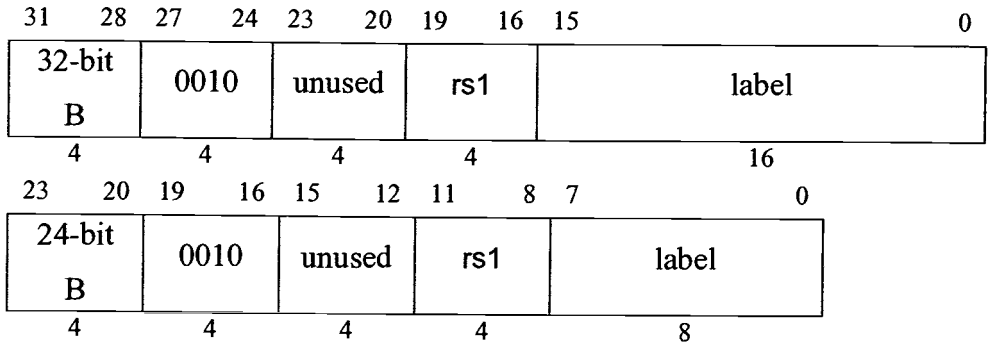
32-bit

if (rs1 == '0'^{32})
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 == '0'^{32})
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:



Branch if Not Equal to Zero (B_NEZ):

Description:

The contents of GPR rs1 are compared to zero, if they are *not* equal, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If they *are* equal, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_NEZ rs1, label

Operation:

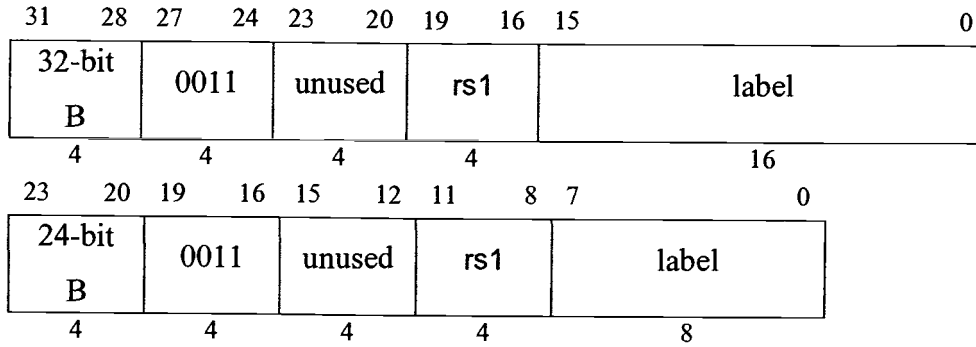
32-bit

if (rs1 != '0'^{32})
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 != '0'^{32})
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:



Branch if Less Than Zero (B_LTZ):

Description:

The contents of GPR rs1 are compared to zero, if the result is less than zero, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If the result is greater than or equal to zero, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_LTZ rs1, label

Operation:

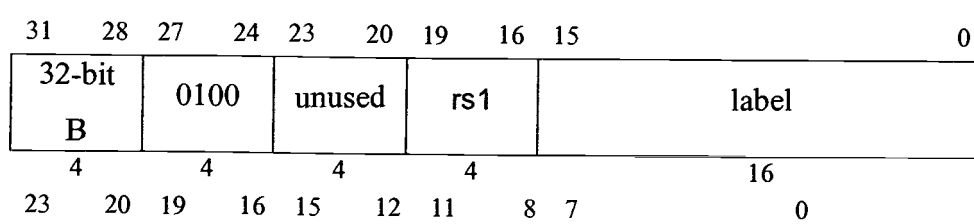
32-bit

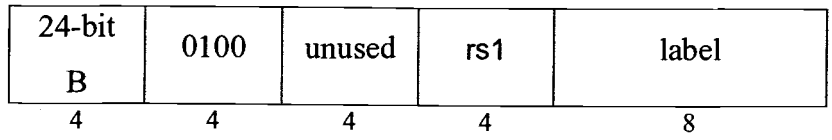
if (rs1 < '0'^{32})
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 < '0'^{32})
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:





Branch if Greater Than Zero (B_GTZ):

Description:

The contents of GPR rs1 are compared to zero, if the result is greater than zero, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If the result is less than or equal to zero, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_GTZ rs1, label

Operation:

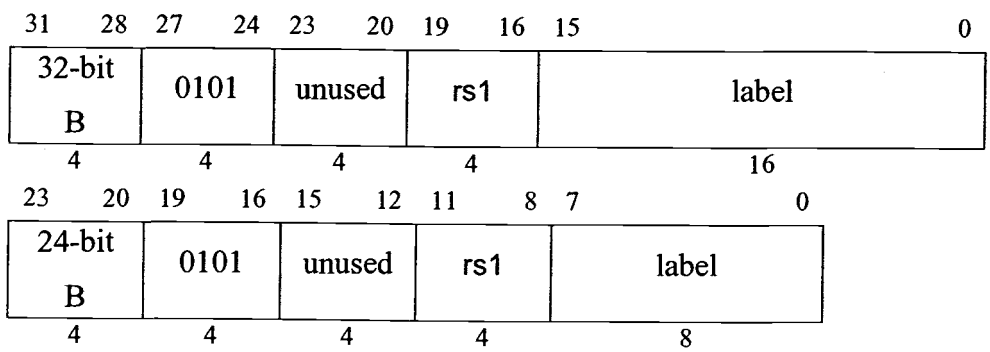
32-bit

if (rs1 > '0'^{32})
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 > '0'^{32})
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:



Branch if Less Than or Equal to Zero (B_LTEZ):

Description:

The contents of GPR rs1 are compared to zero, if the result is less than or equal to zero, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If the result is greater than zero, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_LTEZ rs1, label

Operation:

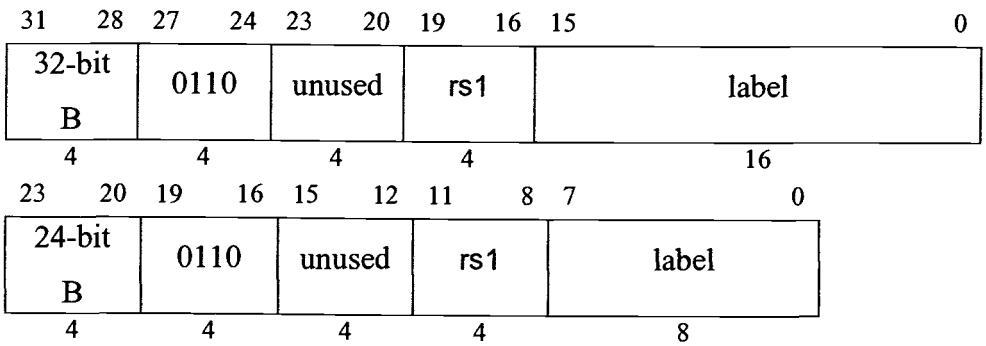
32-bit

if (rs1 <= '0'^{32})
  PC = PC + 4 + 'label$_{15}$'^{16} || label
else
  PC = PC + 4

24-bit

if (rs1 <= '0'^{32})
  PC = PC + 4 + 'label$_{7}$'^{24} || label
else
  PC = PC + 4

Encoding:



Branch if Greater Than or Equal to Zero (B_GTEZ):

Description:

The contents of GPR rs1 are compared to zero, if the result is greater than or equal to zero, the PC is incremented by 4 and added to the sign extended immediate value. The next instruction is then fetched from the newly calculated address. If the result is less than zero, the PC is incremented by 4 and the next instruction is fetched.

Type: Branch

Format:

32-bit / 24-bit

B_GTEZ rs1, label

Operation:

32-bit

if (rs1 >= '0'^{32})

  PC = PC + 4 + 'label$_{15}$'^{16} || label

else

  PC = PC + 4

24-bit

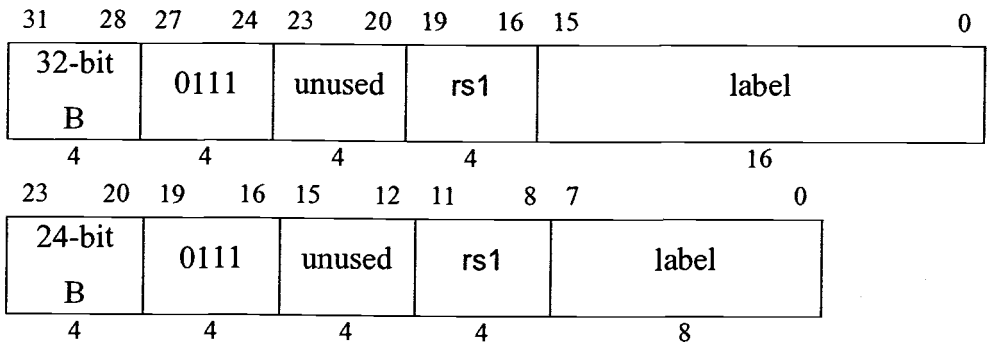
if (rs1 >= '0'^{32})

  PC = PC + 4 + 'label$_{7}$'^{24} || label

else

  PC = PC + 4

Encoding:



Absolute Jump (J_A):

Description:

The immediate value is sign extended and added to the pre incremented PC. This target address is then unconditionally placed in the PC.

Type: Jump / Call

Format:

32-bit / 24-bit

J_A label

Operation:

32-bit

PC = PC + 4 + 'label$_{23}$'^{8} || label

24-bit

PC = PC + 4 + 'label$_{15}$'^{16} || label

Encoding:

Procedural Jump (J_P):

Description:

The immediate value is sign extended and added to the pre incremented PC. This target address is then unconditionally placed in the PC. The address of the next instruction is placed in the link register (LR).

Type: Jump / Call

Format:

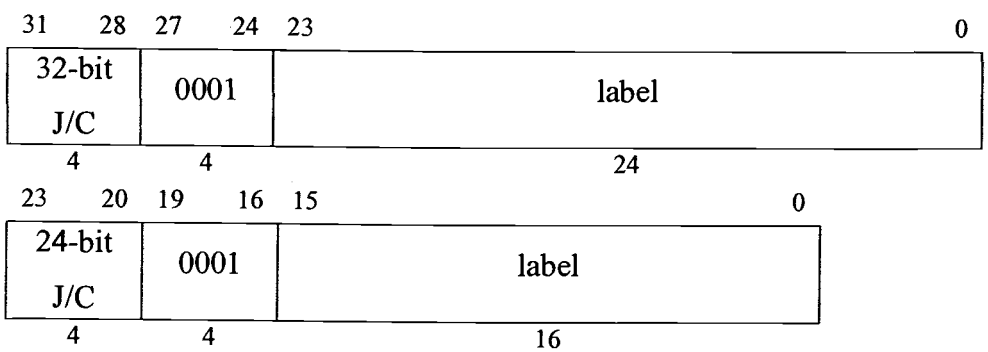
32-bit / 24-bit

J_P label

Operation:

32-bit	*24-bit*
PC = PC + 4 + '$label_{23}$'^{8} || label	PC = PC + 4 + '$label_{15}$'^{16} || label
LR = PC + 4	LR = PC + 4

Encoding:



Absolute Jump Register (J_AR):

Description:

The value in GPR rs is unconditionally placed in the PC.

Type: Register

Format:

32-bit / 24-bit / 16-bit

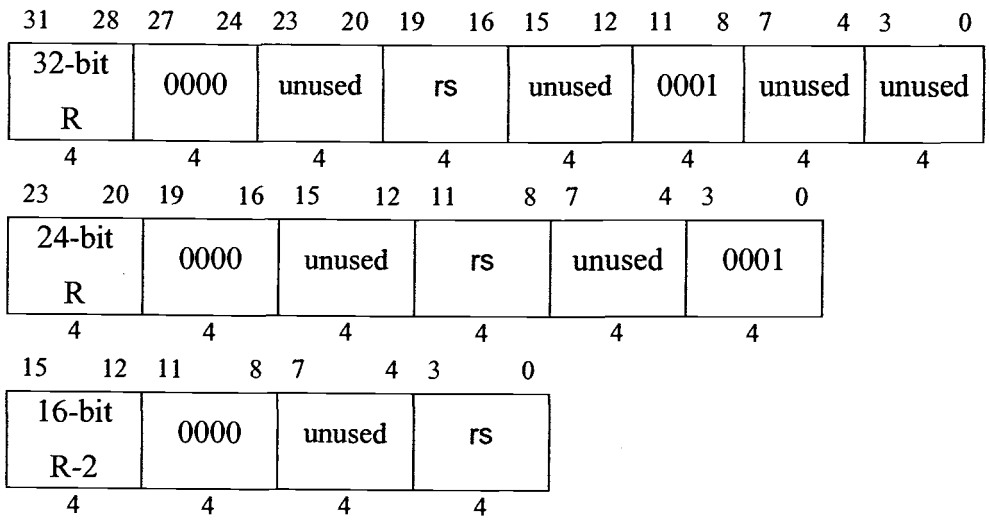
J_AR rs

Operation:

32-bit / 24-bit / 16-bit

PC = rs

Encoding:



Procedural Jump Register (J_PR):

Description:

The value in GPR rs is unconditionally placed in the PC. The address of the next instruction is placed in the link register (LR).

Type: Register

Format:

32-bit / 24-bit / 16-bit

J_PR rs

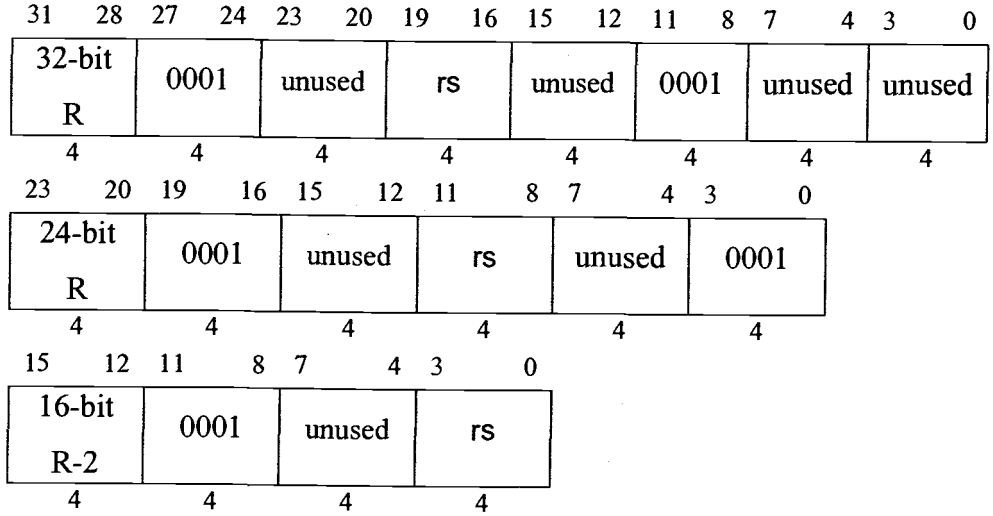
Operation:

32-bit / 24-bit / 16-bit

PC = rs

LR = PC + 4

Encoding:



Jump to Link Register (J_LR):

Description:

The contents of LR are unconditionally placed in the program counter.

Type: Register

Format:

32-bit / 24-bit / 16-bit

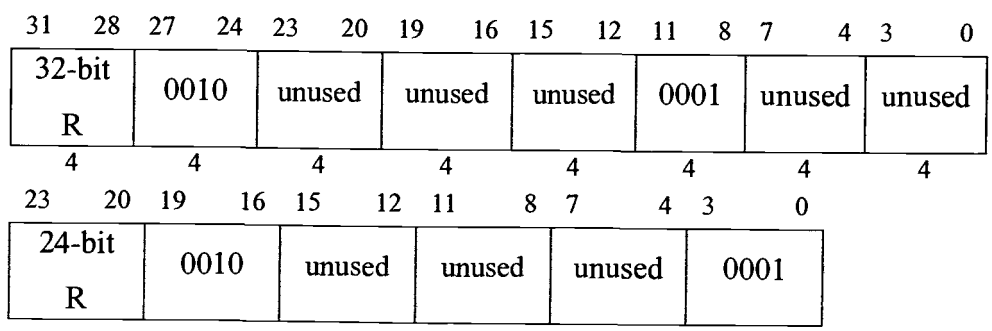
RET

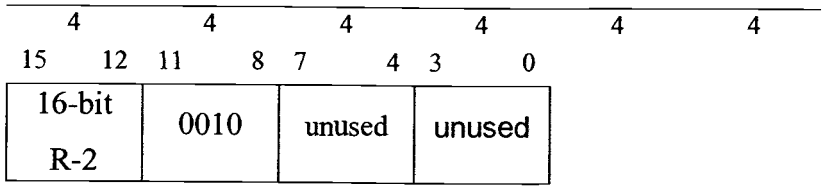
Operation:

32-bit / 24-bit / 16-bit

PC = LR

Encoding:





Push From Stack (PUSH):

Description:

The register value rd is placed onto the next open position in the stack. Then the stack pointer is decremented by 4. The stack pointer initially begins at 0xFFFFFFF8.

Type: Register

Format:

32-bit / 24-bit / 16-bit

PUSH rd

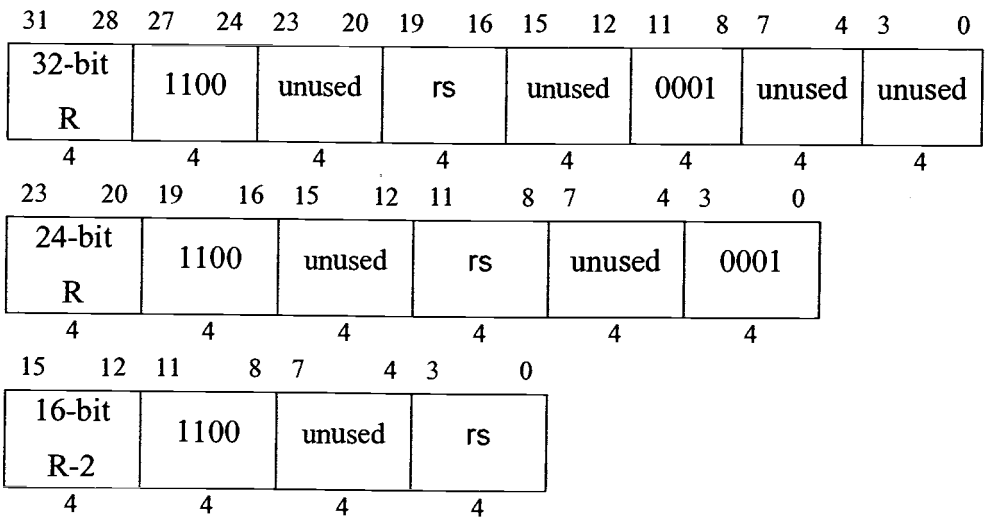
Operation:

32-bit / 24-bit / 16-bit

MEM[SP] = rd

SP = SP - 4

Encoding:



Pop From Stack (POP):

Description:

First the stack pointer is incremented by 4. Then register value rd is placed onto the next open position in the stack. The stack pointer initially begins at 0xFFFFFFF8.

Type: Register

Format:

32-bit / 24-bit / 16-bit

POP rd

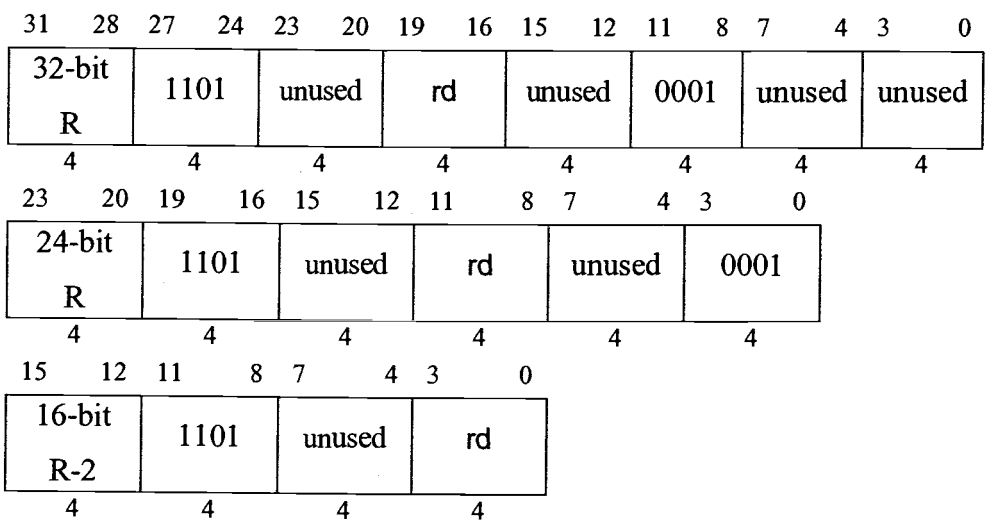
Operation:

32-bit / 24-bit / 16-bit

rd = MEM[SP]

SP = SP + 4

Encoding:



No Operation (NOP):

Description:

NOP does nothing. It is just space filler.

Type: Jump / Call

Format:

32-bit / 24-bit / 16-bit

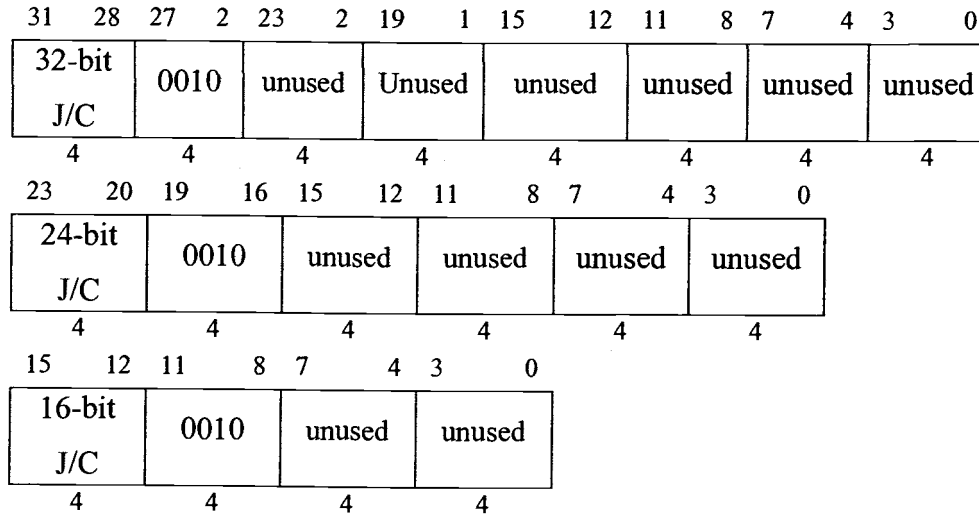
NOP

Operation:

32-bit / 24-bit / 16-bit

NOP

Encoding:



APPENDIX B

X32V Assembly Output

Results

Assembly output for the adpcm.c file

```
#       GCC For the X32V
.file    "adpcm.c" // Source file name
gcc2_compiled.:
.section .data    // The data section
.align 2 // Aligned to the power of 2
.type    _indexTable,@object
        .size    _indexTable,64
// Global data and functions available to
all files in the project.
.global _adpcm_coder
        .type
_adpcm_coder,@function
_adpcm_coder:
// Push the registers to be saved
        push %R0
        push %R1
        push %R2
        push %R5
        push %R6
// Create the Activation Frame
        mov %r6, %sp
        add_i %sp, %sp, -48
        s_w (%r6,24), %r8
        s_w (%r6,28), %r9
        l_w %r8, (%r6,28)
        s_w (%r6,-8), %r8
        l_w %r8, (%r6,24)
        s_w (%r6,-4), %r8
        l_w %r8, (%r6,36)
        l_w %r2, (%r6,-32)
        l_h  %r2,(%r8);
        l_w %r8, (%r6,36)
        l_w %r2, (%r6,-40)
        l_b %r2,(%r8,2)
        l_w %r9, (%r6,-40)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9, _stepsizeTable
        add %r8, %r9, %r8
        l_w %r9, (%r8)
        s_w (%r6,-28), %r9
        mov_upi %r8, 1
        s_w (%r6,-48), %r8
.L2:
        l_w %r8, (%r6,32)
        b_gtz %r8 .L5
        j_a .L3
.L5:
        add_i %r8, %r6, -4
        l_w %r9, (%r8)
        l_w %r2, (%r6,-12)
        l_h  %r2,(%r9);
        add_i %r9, %r9, 2
        s_w (%r8), %r9
        l_w %r8, (%r6,-12)
        l_w %r9, (%r6,-32)
        sub %r8, %r8, %r9
        s_w (%r6,-24), %r8
        l_w %r8, (%r6,-24)
        b_gtez %r8 .L6
        mov_upi %r8, 8
        j_a .L7
.L6:
        mov_upi %r8, 0
.L7:
        s_w (%r6,-16), %r8
        l_w %r8, (%r6,-16)
        b_eqz %r8 .L8
        l_w %r9, (%r6,-24)
        mov_upi %r8, 0
        sub %r8, %r8, %r9
        mov %r9, %r8
        s_w (%r6,-24), %r9
.L8:
        mov_upi %r8, 0
        s_w (%r6,-20), %r8
        l_w %r8, (%r6,-28)
        mov_upi %r2, 3
        sft_ra %r9 %r8 %r2
```

```
        s_w (%r6,-36), %r9
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        b_ltz %r8 %r9 .L9
        mov_upi %r8, 4
        s_w (%r6,-20), %r8
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        sub %r8, %r8, %r9
        s_w (%r6,-24), %r8
        l_w %r8, (%r6,-36)
        l_w %r9, (%r6,-28)
        add %r8, %r8, %r9
        s_w (%r6,-36), %r8
.L9:    // Label for this address
        l_w %r8, (%r6,-28)
        mov_upi %r2, 1
        sft_ra %r9 %r8 %r2
        s_w (%r6,-28), %r9
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        b_ltz %r8 %r9 .L10
        l_w %r8, (%r6,-20)
        lg_ori %r9, %r8, 2
        s_w (%r6,-20), %r9
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        sub %r8, %r8, %r9
        s_w (%r6,-24), %r8
        l_w %r8, (%r6,-36)
        l_w %r9, (%r6,-28)
        add %r8, %r8, %r9
        s_w (%r6,-36), %r8
.L10:
        l_w %r8, (%r6,-28)
        mov_upi %r2, 1
        sft_ra %r9 %r8 %r2
        s_w (%r6,-28), %r9
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        b_ltz %r8 %r9 .L11
        l_w %r8, (%r6,-20)
        lg_ori %r9, %r8, 1
        s_w (%r6,-20), %r9
        l_w %r8, (%r6,-36)
        l_w %r9, (%r6,-28)
        add %r8, %r8, %r9
        s_w (%r6,-36), %r8
.L11:
        l_w %r8, (%r6,-16)
        b_eqz %r8 .L12
        l_w %r8, (%r6,-32)
        l_w %r9, (%r6,-36)
        sub %r8, %r8, %r9
        s_w (%r6,-32), %r8
        j_a .L13
.L12:
        l_w %r8, (%r6,-32)
        l_w %r9, (%r6,-36)
        add %r8, %r8, %r9
        s_w (%r6,-32), %r8
.L13:
        l_w %r8, (%r6,-32)
        sub_i %r8 %r8 32767
        b_ltez %r8 .L14
        mov_upi %r8, 32767
        s_w (%r6,-32), %r8
        j_a .L15
.L14:
        l_w %r8, (%r6,-32)
        sub_i %r8 %r8 -32768
        b_gtez %r8 .L15
        mov_upi %r8, -32768
        s_w (%r6,-32), %r8
.L16:
.L15:
        l_w %r8, (%r6,-20)
        l_w %r9, (%r6,-16)
        lg_or %r8, %r8, %r9
        s_w (%r6,-20), %r8
        l_w %r9, (%r6,-20)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9, _indexTable
        add %r8, %r9, %r8
        l_w %r9, (%r6,-40)
        l_w %r8, (%r8)
        add %r9, %r9, %r8
```

```
        s_w (%r6,-40), %r9
        l_w %r8, (%r6,-40)
        b_gtez %r8 .L17
        mov_upi %r8, 0
        s_w (%r6,-40), %r8
.L17:
        l_w %r8, (%r6,-40)
        sub_i %r8 %r8 88
        b_ltez %r8 .L18
        mov_upi %r8, 88
        s_w (%r6,-40), %r8
.L18:
        l_w %r9, (%r6,-40)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9, _stepsizeTable
        add %r8, %r9, %r8
        l_w %r9, (%r8)
        s_w (%r6,-28), %r9
        l_w %r8, (%r6,-48)
        b_eqz %r8 .L19
        l_w %r9, (%r6,-20)
        mov_upi %r2, 4
        sft_ll %r8 %r9 %r2
        lg_andi %r9, %r8, 240
        s_w (%r6,-44), %r9
        j_a .L20
.L19:
        add_i %r8, %r6, -8
        l_w %r9, (%r8)
        l_b %r0, (%r6,-17)
        lg_andi %r1, %r0, 15
        mov %r0, %r1
        l_b %r1, (%r6,-41)
        lg_or %r0, %r0, %r1
        s_b (%r9), %r0
        add_i %r9, %r9, 1
        s_w (%r8), %r9
.L20:
        mov_upi %r8, 0
        l_w %r9, (%r6,-48)
        b_nez %r8 .L21
        mov_upi %r8, 1
.L21:
        s_w (%r6,-48), %r8
.L4:
        l_w %r8, (%r6,32)
        add_i %r9, %r8, -1
        s_w (%r6,32), %r9
        j_a .L2
.L3:
        l_w %r8, (%r6,-48)
        b_nez %r8 .L22
        add_i %r8, %r6, -8
        l_w %r9, (%r8)
        l_b %r0, (%r6,-41)
        s_b (%r9), %r0
        add_i %r9, %r9, 1
        s_w (%r8), %r9
.L22:
        l_w %r8, (%r6,36)
        l_h %r9, (%r6,-30)
        s_h (%r8), %r9
        l_w %r8, (%r6,36)
        l_b %r9, (%r6,-37)
        s_b (%r8,2), %r9
.L1:
        mov %sp, %r6
        pop %R6
        pop %R5
        pop %R2
        pop %R1
        pop %R0

        ret
.Lfe1:
        .size
_adpcm_coder,.Lfe1-
_adpcm_coder
        .align 1
        .global _adpcm_decoder
        .type
_adpcm_decoder,@function
_adpcm_decoder:
        push %R0
        push %R1
        push %R5
        push %R6
```

```
        mov %r6, %sp
        add_i %sp, %sp, -40
        s_w (%r6,20), %r8
        s_w (%r6,24), %r9
        l_w %r8, (%r6,24)
        s_w (%r6,-8), %r8
        l_w %r8, (%r6,20)
        s_w (%r6,-4), %r8
        l_w %r8, (%r6,32)
        l_w %r1, (%r6,-24)
        l_h  %r1,(%r8);
        l_w %r8, (%r6,32)
        l_w %r1, (%r6,-32)
        l_b %r1,(%r8,2)
        l_w %r9, (%r6,-32)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9, _stepsizeTable
        add %r8, %r9, %r8
        l_w %r9, (%r8)
        s_w (%r6,-20), %r9
        mov_upi %r8, 0
        s_w (%r6,-40), %r8
.L24:
        l_w %r8, (%r6,28)
        b_gtz %r8 .L27
        j_a .L25
.L27:
        l_w %r8, (%r6,-40)
        b_eqz %r8 .L28
        l_w %r8, (%r6,-36)
        lg_andi %r9, %r8, 15
        s_w (%r6,-16), %r9
        j_a .L29
.L28:
        add_i %r8, %r6, -4
        l_w %r9, (%r8)
        l_w %r1, (%r6,-36)
        l_b %r1,(%r9)
        add_i %r9, %r9, 1
        s_w (%r8), %r9
        l_w %r9, (%r6,-36)
        mov_upi %r1, 4
        sft_ra %r8 %r9 %r1
        lg_andi %r9, %r8, 15
        s_w (%r6,-16), %r9
.L29:
        mov_upi %r8, 0
        l_w %r9, (%r6,-40)
        b_nez %r8 .L30
        mov_upi %r8, 1
.L30:
        s_w (%r6,-40), %r8
        l_w %r9, (%r6,-16)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9, _indexTable
        add %r8, %r9, %r8
        l_w %r9, (%r6,-32)
        l_w %r8, (%r8)
        add %r9, %r9, %r8
        s_w (%r6,-32), %r9
        l_w %r8, (%r6,-32)
        b_gtez %r8 .L31
        mov_upi %r8, 0
        s_w (%r6,-32), %r8
.L31:
        l_w %r8, (%r6,-32)
        sub_i %r8 %r8 88
        b_ltez %r8 .L32
        mov_upi %r8, 88
        s_w (%r6,-32), %r8
.L32:
        l_w %r8, (%r6,-16)
        lg_andi %r9, %r8, 8
        s_w (%r6,-12), %r9
        l_w %r8, (%r6,-16)
        lg_andi %r9, %r8, 7
        s_w (%r6,-16), %r9
        l_w %r8, (%r6,-20)
        mov_upi %r1, 3
        sft_ra %r9 %r8 %r1
        s_w (%r6,-28), %r9
        l_w %r9, (%r6,-16)
        lg_andi %r8, %r9, 4
```

```
        b_eqz %r8 .L33
        l_w %r8, (%r6,-28)
        l_w %r9, (%r6,-20)
        add %r8, %r8, %r9
        s_w (%r6,-28), %r8
.L33:
        l_w %r9, (%r6,-16)
        lg_andi %r8, %r9, 2
        b_eqz %r8 .L34
        l_w %r9, (%r6,-20)
        mov_upi %r1, 1
        sft_ra %r8 %r9 %r1
        l_w %r9, (%r6,-28)
        add %r8, %r9, %r8
        s_w (%r6,-28), %r8
.L34:
        l_w %r9, (%r6,-16)
        lg_andi %r8, %r9, 1
        b_eqz %r8 .L35
        l_w %r9, (%r6,-20)
        mov_upi %r1, 2
        sft_ra %r8 %r9 %r1
        l_w %r9, (%r6,-28)
        add %r8, %r9, %r8
        s_w (%r6,-28), %r8
.L35:
        l_w %r8, (%r6,-12)
        b_eqz %r8 .L36
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        sub %r8, %r8, %r9
        s_w (%r6,-24), %r8
        j_a .L37
.L36:
        l_w %r8, (%r6,-24)
        l_w %r9, (%r6,-28)
        add %r8, %r8, %r9
        s_w (%r6,-24), %r8
.L37:
        l_w %r8, (%r6,-24)
        sub_i %r8 %r8 32767
        b_ltez %r8 .L38
        mov_upi %r8, 32767
        s_w (%r6,-24), %r8
        j_a .L39
.L38:
        l_w %r8, (%r6,-24)
        sub_i %r8 %r8 -32768
        b_gtez %r8 .L39
        mov_upi %r8, -32768
        s_w (%r6,-24), %r8
.L40:
.L39:
        l_w %r9, (%r6,-32)
        mov %r8, %r9
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        mov_upi %r9,
_stepsizeTable
        add %r8, %r9, %r8
        l_w %r9, (%r8)
        s_w (%r6,-20), %r9
        add_i %r8, %r6, -8
        l_w %r9, (%r8)
        l_h %r0, (%r6,-22)
        s_h (%r9), %r0
        add_i %r9, %r9, 2
        s_w (%r8), %r9
.L26:
        l_w %r8, (%r6,28)
        add_i %r9, %r8, -1
        s_w (%r6,28), %r9
        j_a .L24
.L25:
        l_w %r8, (%r6,32)
        l_h %r9, (%r6,-22)
        s_h (%r8), %r9
        l_w %r8, (%r6,32)
        l_b %r9, (%r6,-29)
        s_b (%r8,2), %r9
.L23:
        mov %sp, %r6
        pop %R6
        pop %R5
        pop %R1
        pop %R0
        ret
.Lfe2:
```

```
        .size    _adpcm_decoder,.Lfe2-
_adpcm_decoder
        .ident   "GCC: (GNU)
egcs-2.91.66 19990314 (egcs-1.1.2
release)"
```

Assembly output for the G711 decode.c program

```
#       GCC For the  X32V
.file   "decode.c"
gcc2_compiled.:
        .section .data
        .align 2
        .type    _in_buffer___2,@object
        .size    _in_buffer___2,4
_in_buffer___2:
        .long 0
        .align 2
        .type    _in_bits___3,@object
        .size    _in_bits___3,4
_in_bits___3:
        .long 0
        .section .text
        .align 1
        .global _unpack_input
        .type
_unpack_input,@function
_unpack_input:
        push %lr
        push %R0
        push %R5
        push %R6
        mov %r6, %sp
        add_i %sp, %sp, -24
        s_w (%r6,20), %r8
        s_w (%r6,24), %r9
        l_w %r8, (_in_bits___3)
        l_w %r9, (%r6,24)
        b_gtez %r8 %r9 .L2
        add_i %r8, %r6, -1
        mov_upi %r9, 1
        s_w (%sp,12), %r9
        l_w %r9, (_fp_in)
        s_w (%sp,16), %r9
        mov_upi %r9, 1
        j_p _fread
        sub_i %r8 %r8 1
        b_eqz %r8 .L3
        l_w %r8, (%r6,20)
        mov_upi %r9, 0
        s_b (%r8), %r9
        mov_upi %r8, -1
        j_a .L1
.L3:
        l_bu %r8, (%r6,-1)
        l_w %r9, (_in_bits___3)
        sft_ll %r8 %r8 %r9
        l_w %r9, (_in_buffer___2)
        lg_or %r8, %r9, %r8
        s_w (_in_buffer___2), %r8
        l_w %r8, (_in_bits___3)
        add_i %r9, %r8, 8
        s_w (_in_bits___3), %r9
.L2:
        l_w %r8, (%r6,20)
        mov_upi %r9, 1
        l_w %r0, (%r6,24)
        sft_ll %r9 %r9 %r0
        mov %r0, %r9
        add_i %r9, %r0, -1
        l_b %r0,
(_in_buffer___2+3)
        nop
        lg_and %r0, %r0, %r9
        s_b (%r8), %r0
        l_w %r8, (_in_buffer___2)
        l_w %r9, (%r6,24)
        sft_rl %r8 %r8 %r9
        s_w (_in_buffer___2), %r8
        l_w %r8, (_in_bits___3)
        l_w %r9, (%r6,24)
        sub %r8, %r8, %r9
        s_w (_in_bits___3), %r8
        mov_upi %r8, 0
```

```
        l_w %r9, (_in_bits___3)
        b_ltez %r8 .L4
        mov_upi %r8, 1
.L4:
        j_a .L1
.L1:
        mov %sp, %r6
        pop %R6
        pop %R5
        pop %R0

        pop %pc
.Lfe1:
        .size    _unpack_input,.Lfe1-
_unpack_input
.section        .rodata
.LC0:
        .string "r"
.LC1:
        .string "%s\n"
.LC2:
        .string "file error"
.LC3:
        .string ".pcm"
.LC4:
        .string "w"
.LC5:
        .string "CCITT ADPCM
Decoder -- usage:\n"
.LC6:
        .string "\tdecode [-3|4|5] [-a|u|l]
-f infile\n"
.LC7:
        .string "where:\n"
.LC8:
        .string "\t-3\tProcess G.723
24kbps (3-bit) input data\n"
.LC9:
        .string "\t-4\tProcess G.721
32kbps (4-bit) input data [default]\n"
.LC10:
        .string "\t-5\tProcess G.723
40kbps (5-bit) input data\n"
.LC11:
        .string "\t-a\tGenerate 8-bit
A-law data\n"
.LC12:
        .string "\t-u\tGenerate 8-bit
u-law data [default]\n"
.LC13:
        .string "\t-l\tGenerate 16-
bit linear PCM data\n"
        .section .text
        .align 1
        .global _main
        .type    _main,@function
_main:
        push %lr
        push %R0
        push %R5
        push %R6
        mov %r6, %sp
        add_i %sp, %sp, -176
        s_w (%r6,20), %r8
        s_w (%r6,24), %r9
        add_i %r9, %r6, -60
        mov %r8, %r9
        j_p _g72x_init_state
        mov_upi %r8, 1
        s_w (%r6,-64), %r8
        mov_upi %r8, 1
        s_w (%r6,-68), %r8
        mov_upi %r8,
_g721_decoder
        s_w (%r6,-72), %r8
        mov_upi %r8, 4
        s_w (%r6,-76), %r8
.L6:
        l_w %r8, (%r6,20)
        sub_i %r8 %r8 1
        b_ltez %r8 .L9
        l_w %r9, (%r6,24)
        add_i %r8, %r9, 4
        l_w %r9, (%r8)
        l_b %r8,(%r9)
        sub_i %r8 %r8 45
        b_eqz %r8 .L8
        j_a .L9
.L9:
```

```
        j_a .L7
.L8:
        l_w %r9, (%r6,24)
        add_i %r8, %r9, 4
        l_w %r9, (%r8)
        add_i %r8, %r9, 1
        l_b %r9, (%r8)
        add_i %r0, %r9, -51
        mov %r8,%r0;
        sub_i %r8 %r8 66
        b_gtz %r8 .L20
        mov %lo, %r8
        mul  %lo, 4 , %lo
        mov %hi, %hi
        add %r9, %r8, .L21
        l_w %r8, (%r9)
        j_ar  %r8
.section        .rodata
        .align 2
        .align 2
.L21:
        .long .L20
        .long .L20
        .long .L14
        .section .text
        j_a .L20
.L11:
        mov_upi %r8,
_g723_24_decoder
        s_w (%r6,-72), %r8
        mov_upi %r8, 3
        s_w (%r6,-76), %r8
        j_a .L10
.L12:
        mov_upi %r8, _g721_decoder
        s_w (%r6,-72), %r8
        mov_upi %r8, 4
        s_w (%r6,-76), %r8
        j_a .L10
.L13:
        mov_upi %r8,
_g723_40_decoder
        s_w (%r6,-72), %r8
        mov_upi %r8, 5
        s_w (%r6,-76), %r8
        j_a .L10
.L14:
        mov_upi %r8, 1
        s_w (%r6,-64), %r8
        mov_upi %r8, 1
        s_w (%r6,-68), %r8
        j_a .L10
.L15:
        mov_upi %r8, 2
        s_w (%r6,-64), %r8
        mov_upi %r8, 1
        s_w (%r6,-68), %r8
        j_a .L10
.L16:
        mov_upi %r8, 3
        s_w (%r6,-64), %r8
        mov_upi %r8, 2
        s_w (%r6,-68), %r8
        j_a .L10
.L17:
        l_w %r8, (%r6,20)
        add_i %r9, %r8, -1
        s_w (%r6,20), %r9
        l_w %r8, (%r6,24)
        add_i %r9, %r8, 4
        s_w (%r6,24), %r9
        l_w %r8, (%r6,24)
        add_i %r9, %r8, 4
        s_w (%r6,24), %r9
        l_w %r8, (%r6,24)
        l_w %r9, (%r8)
        mov_upi %r0, .LC0
        mov %r8, %r9
        mov %r9, %r0
        j_p _fopen
        s_w (_fp_in), %r8
        l_w %r8, (_fp_in)
        b_nez %r8 .L18
        l_w %r9, (__impure_ptr)
        l_w %r8, (%r9,12)
        mov_upi %r9, .LC1
        mov_upi %r0, .LC2
        s_w (%sp,12), %r0
        j_p _fprintf
        mov_upi %r8, 1
```

```
        j_p _exit
.L18:
        add_i %r8, %r6, -156
        l_w %r9, (%r6,24)
        l_w %r0, (%r9)
        mov %r9, %r0
        j_p _strcpy
        add_i %r8, %r6, -156
        mov_upi %r9, .LC3
        j_p _strcat
        add_i %r8, %r6, -156
        mov_upi %r9, .LC4
        j_p _fopen
        s_w (_fp_out), %r8
        l_w %r8, (_fp_out)
        b_nez %r8 .L19
        l_w %r9, (__impure_ptr)
        l_w %r8, (%r9,12)
        mov_upi %r9, .LC1
        mov_upi %r0, .LC2
        s_w (%sp,12), %r0
        j_p _fprintf
        mov_upi %r8, 1
        j_p _exit
.L19:
        j_a .L10
.L20:
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC5
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC6
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC7
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC8
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC9
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC10
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC11
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC12
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        l_w %r8, (__impure_ptr)
        l_w %r9, (%r8,12)
        mov_upi %r0, .LC13
        mov %r8, %r9
        mov %r9, %r0
        j_p _fprintf
        mov_upi %r8, 1
        j_p _exit
.L10:
        l_w %r8, (%r6,20)
        add_i %r9, %r8, -1
        s_w (%r6,20), %r9
        l_w %r8, (%r6,24)
        add_i %r9, %r8, 4
        s_w (%r6,24), %r9
```

```
        j_a .L6
.L7:
        nop
.L22:
        add_i %r8, %r6, -3
        l_w %r9, (%r6,-76)
        j_p _unpack_input
        b_gtez %r8 .L24
        j_a .L23
.L24:
        l_bu %r8, (%r6,-3)
        l_w %r9, (%r6,-64)
        add_i %r0, %r6, -60
        s_w (%sp,12), %r0
        l_w %r0, (%r6,-72)
        j_pr %r0
        s_h (%r6,-2), %r8
        l_w %r8, (%r6,-68)
        sub_i %r8 %r8 2
        b_nez %r8 .L25
        add_i %r8, %r6, -2
        l_w %r9, (%r6,-68)
        mov_upi %r0, 1
        s_w (%sp,12), %r0
        l_w %r0, (_fp_out)
        s_w (%sp,16), %r0
        j_p _fwrite
        j_a .L26
.L25:
        l_b %r8, (%r6,-1)
        s_b (%r6,-3), %r8
        add_i %r8, %r6, -3
        l_w %r9, (%r6,-68)
        mov_upi %r0, 1
        s_w (%sp,12), %r0
        l_w %r0, (_fp_out)
        s_w (%sp,16), %r0
        j_p _fwrite
.L26:
        j_a .L22
.L23:
        l_w %r9, (_fp_out)
        mov %r8, %r9
        j_p _fclose
.L5:
        mov %sp, %r6
        pop %R6
        pop %R5
        pop %R0
        pop %pc
.Lfe2:
        .size    _main,.Lfe2-_main
        .local   _fp_in
        .comm _fp_in,4,4
        .local   _fp_out
        .comm _fp_out,4,4
        .ident   "GCC: (GNU)
egcs-2.91.66 19990314 (egcs-1.1.2
release)
```