Tricore Port for GCC - An Analysis

CS384 Course Project Report

by

Lakulish Antani
Hidayath Ansari
Aditya Parameswaran

under the guidance of

Prof. Uday Khedker

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

Abstract

This document deals with the changes that the existing Tricore port has made to the GCC source. We deal mainly with respect to bit variables, and at what stage in the build process the bit variable code is included. We supplement this with our experiments with bit variables on Tricore. We also include our general studies on Tricore and GCC that we had done prior to doing this.

1 Introduction

GCC or the GNU Compiler Collection, being *portable*, provides a method for specifying the processor architecture in order to create a cross-compiler specific to that architecture. This is via Machine Description files (which are written in an RTL-like syntax) and associated C source code which are placed in the `config` folder in the GCC source directory. More details on the structure of GCC are in Stallman's GCC Internals [1].

Our original project was to do with porting the GCC 3.3 compiler for the Tricore Architecture to one for GCC 4.0. So, our first task was to understand the build process for GCC 3.3 and to gain an overview of Machine Description files. Our studies on the build process are documented in Section 3.

We then were expected to provide some analysis on how GCC 3.3 for Tricore differed from the pristine GCC 3.3 version, so that we could mirror the same (or similar changes) in GCC 4.0 in order to make it "hospitable" for Tricore. We expected to find a modularized set of changes within the GCC structure which dealt with Tricore, but realized that it was not so. We then set ourselves the revised goal of seeing if there was any possibility of cleaning up the GCC port for Tricore to localize the changes made by HighTec to the `config` folder in the source (as an ideal port (Section 4.1) should be)

We then discovered that the Tricore GCC port went much deeper that we had originally thought. It introduces a new construct called Bit Variables (via `_bit`), and makes other changes to core GCC files. We performed extensive analysis and experiments to find out where exactly in the build process `_bit` has been introduced and the true depth to which the effect of these bit variables are felt. These are documented in Section 5.2.3. We also analyzed the similarity of bit variables with another construct that we discovered closely resembled bit variables, i.e. `_Bool`. These experiments can be found in Section 5.3.

2 Structure of GCC

The GCC compiler building framework can be separated into three kinds of files: *generic, generator* and *generated*. The *generic* files are a standard part of the compiler, to be used without any modifications. They mainly correspond to the front end of the compiler.

Like the *generic* component, the *generator* files are also a part of the source tree and are tasked with mining the target machine description. They extract information related to the instructions, constants, codes, RTL information and components of the compiler which use this information. This information is used to create components that form the *generated* segment. The *generic* and *generated* segments are then linked to give the final compiler executable. The information flow begins at the AST to RTL conversion stage, where the AST is matched against the RTL templates of the MD. The MD is also used to generate assembly code after the RTL undergoes several optimization passes. This information from the MD resides in the *generated* files. A graphical view of this structure can be found in Fig. 1.

An instance of a generator file using the MD file to create a generated file (from the build log file):

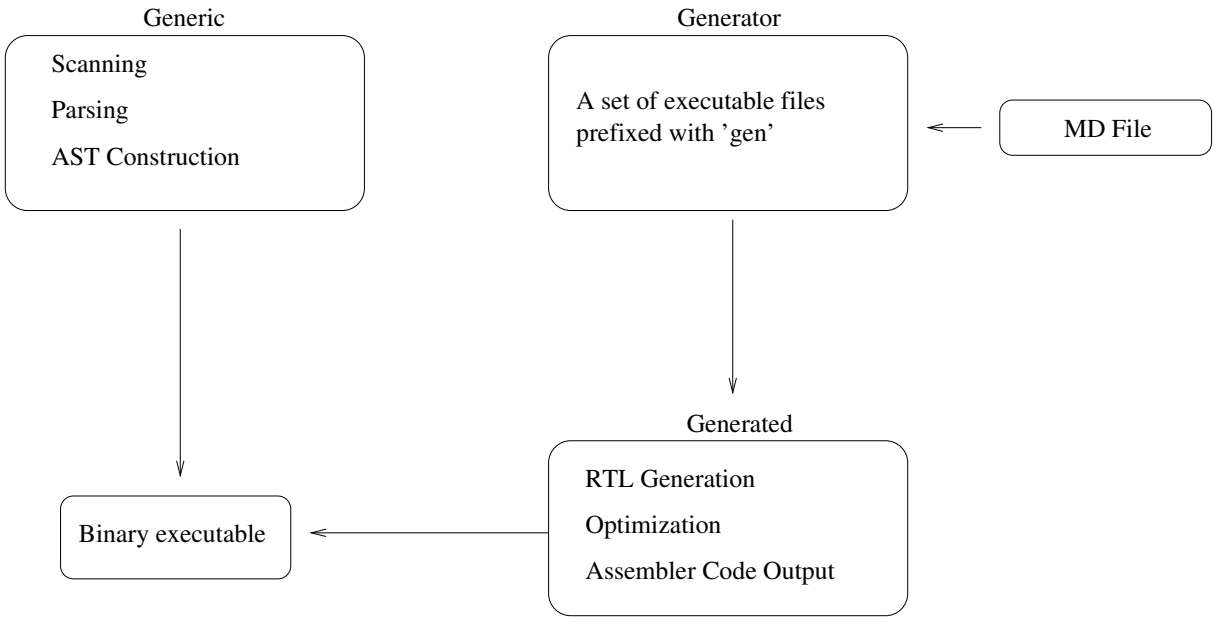

Figure 1: GCC Structure

```
gcc33 -c <other options, sources> ../../sources/gcc-3.3/gcc/genpeep.c
                                                        -o genpeep.o
gcc33  -g -O2 <other options, sources> -DGENERATOR_FILE  -o genpeep
                                                    genpeep.o rtl.o ...

./genpeep ../../sources/gcc-3.3/gcc/config/tricore/tricore.md > tmp-peep.c
/bin/sh ../../sources/gcc-3.3/gcc/move-if-change tmp-peep.c insn-peep.c
```

`genpeep.c` and its binary file `genpeep`, both of which are part of the *generator* set, take the MD file as input and produces `insn-peep.c`. `insn-peep.c` (*generated*) here contains information related to peephole optimization.

In the next section we explore the steps in the build process for a GCC compiler.

3 The Build Process

The build process involves three systems, which may potentially all be the same.

- Build machine: The system whose compiler we are using to build.
- Host machine: The system on which the compiler we produce is to run.
- Target machine: The system for which the generated compiler produces binary code.

A native compiler is one that has all three systems of identical architectures. A cross-compiler is one in which the build and host are the same, but target is different. Tricore falls in this category, since we use GCC on i386 machines to generate compilers that run on i386 machines which produce binary for Tricore processors.

There are basically two components to the build process. For a complete installation of the Tricore cross-compiler, the binutils package has to be compiled as well. The Binutils package includes the associated binaries with the core compiler, such as linker, loader and assembler.

For a full installation, these come into play after the `xgcc` is built and are required for a clean finish of the make. We used binutils-2.13. The directory structure is as follows:

- `sources` {SOURCEDIR}
  - `binutils-2.13`
  - `gcc-3.3`
  - `newlib`
- `builds`
  - `binutils`
  - `gcc`
- `installs` {INSTALLDIR}

3.1 Compiling binutils

The following commands are best issued from the `builds/binutils` directory. Firstly, the following command needs to be executed:

```
{SOURCEDIR}/binutils-2.13/configure
     --target=tricore
     --prefix={INSTALLDIR}
     --program-prefix=tricore- -v
```

The switches given to configure are self-explanatory. Since we are compiling the GCC 3.3 cross-compiler, we need to make sure that we compile it with a lesser or equal GCC. This may require going into the generated Makefile and changing the `CC` variable to the appropriate compiler binary (line 112 of Makefile). We changed

```
CC=gcc
```

to

```
CC=gcc33
```

assuming that the compiler binary is `gcc33`. Then the following commands are executed.

```
make all
make install
```

This completes the `binutils` installation.

3.2 Compiling C Cross-compiler

This build is best executed from the `builds/gcc` directory. The configure command is as follows:

```
{SOURCEDIR}/gcc-3.3/configure
    --target=tricore
    --enable-languages=c
    --prefix={INSTALLDIR}
    --program-prefix=tricore-
    --with-local-prefix={INSTALLDIR}
    --with-newlib
    --with-included-gettext
    --with-catgets -v
```

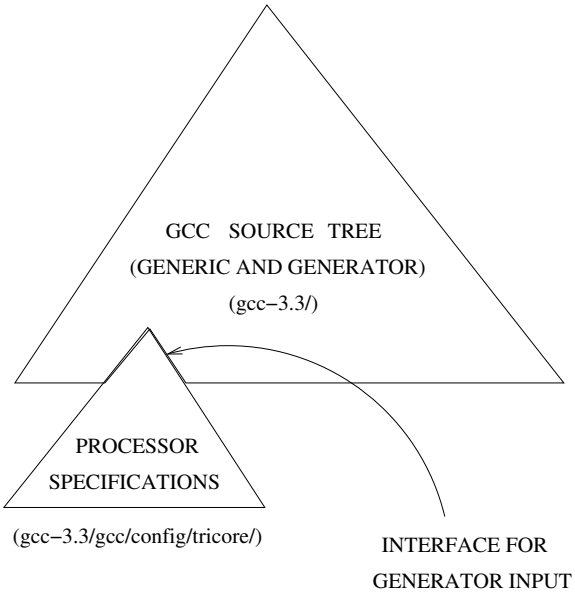

Figure 2: Ideal Port

The compiler can be compiled for C++ as well, by changing the `--enable-languages` flag to `c++`. The `CC` variable in the `Makefile` needs to be checked and changed appropriately, for a similar reason as above. After that, the following commands need to be issued (from the `builds/gcc` directory):

```
ln -s ./gcc/tricore.specs ./gcc/tricore-elf.specs
mkdir {INSTALLDIR}/tricore
cp -rf {SOURCEDIR}/newlib/newlib/libc/include/
       {INSTALLDIR}/tricore/include
export PATH={INSTALLDIR}/tricore/bin:{INSTALLDIR}/bin\:\$PATH
```

And the make command is as follows:

```
make all-gcc LANGUAGES=c
```

(`c++` instead of `c` if one is compiling for C++) This completes the build process. The executable is to be found in `INSTALLDIR/bin/tricore-gcc`.

4 Overview of the Tricore Port

4.1 An Ideal Port

For constructing a port of GCC for an architecture, the specifications of the processor are used to implement certain features in the instruction set generated. An ideal port of GCC for the Tricore Architecture should localize related architecture-specific information to one directory in the source tree i.e. (`gcc/config/tricore`). Our ideal port would be like Fig. 2. The associated C functions and constants required to model the processor should all exist within that directory. However, we find that this is not the case with the current Tricore port for GCC. Here are a few files which ideally should not have Tricore specific code, but end up "polluting" the pristine GCC source:

- gcc.c
- tree.c
- stor-layout.c
- rtl.c
- emit-rtl.c

Since these are files with code that is central to the GCC build process, it is rather counter-intuitive that a port would need to make changes in them. Just as language dependencies in the front end of the processor can be eliminated to give a intermediate language (for example, GIMPLE) irrespective of the input language, we would like our processor-based dependences not to affect other parts of the compiler construction suite, and these dependencies be introduced in an entirely streamlined manner.

However, as our investigations into the Tricore port show, the clean separation of processor-related information (and optimizations) and the generic GCC compiler construction framework has not taken place. At several places in the *generic* (Section 4.3) part of GCC, we find that Tricore has introduced additions (mainly) and modifications (rarely). This makes the task of porting Tricore to GCC 4.0 using the GCC 3.3 version all the more difficult.

4.2 Preprocessor Flags

We note that the Tricore port for GCC uses a number of preprocessor flags.

- `__BIT_VARIABLES__`
- `__TRICORE__`
- `__HIGHTEC__`

These flags are used to modify the operation of GCC at various stages of the compilation process, and at times are used to optimize and include additional features to GCC.

Most of the code in the TriCore port for GCC that deals with bit variables are within an `#ifdef __BIT_VARIABLES__`

4.3 A Possible Classification Of Changes

A Classification of the changes that Tricore does to GCC, and the files associated: (All files are in the `gcc` directory unless otherwise specified.) Please note that the sentences beside each file describe the changes that were introduced by Tricore, and not the primary function of the files.

4.3.1 Installation or Configuration Changes

The Tricore machine name and other details are added to `config.sub`, `configure.in`, `config.gcc` (these 3 files in base source directory), `gcc/configure`, `gcc/configure.in`, `gcc/Makefile.in`.

4.3.2 `__BIT_VARIABLES__` Changes

This includes `OWN_BIT_TYPE`, and other `#ifdefs`. We went through the changes added by `__BIT_VARIABLES__` in files that are changed by Tricore. A brief summary of some of the files is presented below.

One change common to many files was that in some instances, the function `GET_MODE_SIZE` (which is the number of bytes used by a mode) was replaced by `GET_MODE_BITSIZE` (which is the

number of bits used by a mode). This change was not global, but only in a few select places. Also, within __BIT_VARIABLES__ #ifdef and #endif, there were occurrences of GET_MODE_SIZE as well. This is used, for example, in the following case. BIMode has a size of 1 and a bitsize of 1, whereas QIMode has a size of 1 and a bitsize of 8, so a decision as to whether widening has to be done in a typecast requires the bitsizes to be compared as opposed to the sizes. This happens in the files combine.c, cse.c, function.c, expr.c, reload1.c, reload.c and simplify-rtx.c in the gcc directory. In one file, a nearby comment elucidated this change by mentioning that bitsize needs to be tested in some cases because of BIMode which has the same unit size as QIMode but has to be extended.

The function gcc_init_libintl undergoes a change in its prototype from:

```
gcc_init_libintl((void))
```

to

```
gcc_init_libintl((char *))
```

This happens in the file gcc/intl.c, and a number of other files call this function in its new form. We are not clear with how this change is necessary. In some cases, an append of share/locale is done whereas for others it is not performed.

Here are some other changes::

- c-common.c : The change occurs in the c_common_type_for_mode which returns appropriate tree nodes for machine modes. The change adds support for BIMode nodes. Similar code is compiled if BIT_MODE is defined, but the type of node returned in this case is different. This suggests that __BIT_VARIABLES__ code can be separated from the other added code.
- c-common.h : Defines tree node types for Bit Variables.
- c-decl.c : Adds AST node-building functions to handle Bit Variables and sets up virtual typedefs (Section 5.2.3).
- c-typeck.c : Supplementary AST-traversing code, and actual bit operations are augmented here. Error reporting is also done.
- tree.c, tree.h : A type check is added.
- Other files that are changed: combine.c, cse.c, c-typeck.c, expmed.c, expr.c, function.c, reload1.c, reload.c, simplify-rtx.c, stor-layout.c, regrename.c

4.3.3 __HIGHTEC__ and __TRICORE__ Changes

- combine.c : Defines the mode_dependent_mem_address_p function. This doesnt change the mode of the address if it changes the meaning of the address.
- cpplib.c: Adds an error-handling condition dealing with preprocessing directives in the _cpp_handle_directive function.
- emit-rtl.c : Tricore-specific optimizations for generated RTL.

- `expmed.c, toplev.c, varasm.c`: Tricore-specific Abstract Syntax Tree manipulation in the `extract_bit_field` function.
- `final.c` : Statistics generation for Tricore insns.
- `gcc.c`: Handling Tricore-specific command-line options in the `process_command` function. The changes in `gcc.c` also deal with path information and related manipulation. Two new functions `target_add_version_info` and `target_add_options_translations` are also added in the Tricore version.
- `stor-layout.c`: Contains code to widen the access mode in case it's a `BIMode`. Some code is enclosed in a `TRICORE #ifdef` , previously unconditional code.

4.3.4 Other Changes

- `TARGET_BRANCH_PRAGMAS` a target hook defined in `tricore.h` and called from `c-semantics.c` and `tree.c`
- `cppmacro.c` : Augments error-handling procedures with more checks. Seems to be fixing holes in C++.
- Other trivial clerical changes contained in other files: `config.if, cppmacro.c, cpplib.c, cpperror.c, diagnostic.c, version.c`

A complete list of changed files is at `ourStudies/completelisting.txt`.

5 Bit Variables

5.1 Introduction to Bit Variables

We noticed that the Tricore port gives an additional data type, that of _bit. The _bit code can be used to define a simple boolean data type for Tricore. We also checked that programs using this data type do not compile with unmodified GCC 3.3 source. (henceforth referred to as "Original GCC").

For example, consider the program below, which compiles with Tricore GCC, but not with ordinary GCC.

```
main()
{
        _bit myint;
        myint = 1;
}
```

This compiled for Tricore GCC but not for Ordinary GCC. In a later section (5.3) we will explore the facility that ordinary GCC has for bit variables (`_Bool`)

5.2 Bit Variables in the AST

5.2.1 A Brief Introduction to AST Nodes

The AST, or Abstract Syntax Tree allows information from high level programming language constructs to be expressed in a tree format. The tree stores information on the following:

- Declaration of procedures, variables, types etc.
- Expressions
- Loops, Conditionals etc.

The AST has nodes corresponding to each instruction in the code file, plus for items in this line of code, like variable names, initial values etc. It represents each instruction in a hierarchical fashion. The nodes also have attributes which point to other nodes or values. The AST is converted later on to RTL.

These are the fields common to all the nodes of the AST. We list them here to provide a more complete picture of the AST phase of the compilation process, which is relevant to bit variables.

- tree_code: gives the type of the node. These types are defined in tree.def
- node_type: gives a pointer to the base type of the node.
- node_chain: gives a pointer to another node (as in a linked list)

5.2.2 Declarations and the Declaration Stack

Types are defined by creating such `TYPE_DECL` nodes in the initial AST. When a `typedef` is encountered during parsing an input program, a similar `TYPE_DECL` node is created.

The following attributes are set for a `TYPE_DECL` node:

- `TREE_CODE`: Set to `TYPE_DECL`.
- `TREE_TYPE`: Contains the type information.
- `DECL_NAME`: The name of the type.
- `DECL_CONTEXT`: Pointer to a `FUNCTION_DECL` node in which the type is declared, or `NULL_TREE` if the type is declared in global scope.

This type declaration node is then pushed on the *declaration stack*. The declaration stack is the manner in which lexical scope rules are implemented in GCC. As each declaration is made, it is pushed onto the declaration stack, so if the same name is redeclared (where the name can belong to a type, variable or function) then the new copy will override the old copy by virtue of being closer to the top of the stack. In the above code snippet the call to `pushdecl` is how the type declaration node for `_bit` is created and pushed on the declaration stack.

`tree pushdecl(tree decl)` inserts the declaration node `decl` into the symbol table, and returns a tree node back. The returned node need not necessarily be the same as the argument. In case an equivalent declaration is already in the symbol table, this is returned. In this manner, redundant declarations can be easily caught. The declarations are thus placed at the front of the declaration list, so that the list is inverted with respect to the insertion order.

5.2.3 How Bit Variables Are Added

The `BIT_VARIABLES` flag compiles support for the `_bit` basic data type, which represents a single bit. This data type is 1 byte wide (as shown by the definition of `c_bit_type_node`).

The following code initializes the attributes of `c_bit_type_node`:

```
c_bit_type_node = make_unsigned_type (BIT_TYPE_SIZE);
TREE_SET_CODE (c_bit_type_node, BOOLEAN_TYPE);
TYPE_MAX_VALUE (c_bit_type_node) = build_int_2 (1, 0);
TREE_TYPE (TYPE_MAX_VALUE (c_bit_type_node)) = c_bit_type_node;
TYPE_PRECISION (c_bit_type_node) = 1;
pushdecl (build_decl (TYPE_DECL, get_identifier ("_bit"),
                      c_bit_type_node));
(*targetm.set_default_type_attributes) (c_bit_type_node);
```

The following fields of AST nodes are used in the above code snippet:

- `TREE_CODE`: Constant indicating what type of node this is. Possible values that this field may have include `TYPE_DECL`, `VAR_DECL` and `FUNCTION_DECL`.
- `TYPE_MAX_VALUE`: The maximum value that the type can represent (1 here).
- `TYPE_PRECISION`: The number of bits used by the type (1 here).

`c_bit_type_node` is an AST node representing the `_bit` type. This problem is resolved using the following method. In the function `c_init_decl_processing`, which sets up the initial state of the AST, a `TYPE_DECL` tree node is created, which defines `_bit`.

5.2.4 Why Do It This Way?

A look at the YACC script that GCC uses as part of the C language parser (`c-parse.y`) shows that `_bit` is not a reserved keyword. However, note that `_Bool` is a reserved keyword, and its definition shows that it is identical (in the case of Tricore) to `_bit`. `_bit` cannot be defined as a reserved keyword in the YACC script. This is because unlike C/C++ (which supports conditional compilation via preprocessor macros like `#ifdef`), YACC (or Bison, on a GNU/Linux host machine) does not allow conditional compilation of the YACC script i.e. YACC doesn't allow parts of the `.y` file to be included or excluded based on the presence or absence of certain flags.

5.3 Connection of `_bit` with `_Bool`

In this section we talk of the facility provided by original GCC to deal with boolean variables and a few experiments that we did to find out the relationship between `_bit` and `_Bool`. We talk of the introduction of `_Bool` in GCC 3.3 in Section 5.5.

The crux is: `_Bool` is a *builtin*. Here is the code from `c-parse.y` which sets `_Bool` as a reserved keyword.

```
static const struct resword reswords[] =
{
  { "_Bool",            RID_BOOL,       0 },

⋮

  { "while",            RID_WHILE,      0 },
};
```

But, at the same time, it is defined along with `_bit` in `c-decl.c`.

```
TYPE_PRECISION (c_bool_type_node) = 1;
pushdecl (build_decl (TYPE_DECL, get_identifier ("_Bool"),
                      c_bool_type_node));
```

where pushdecl, build_decl, c_bool_type_node have all been defined in previous sections. Thus, like `_bit _Bool` is also a typedef defined by GCC to deal with boolean variables. However on the other hand, it is also a reserved keyword, which is why it cannot be used as a variable name. This is the reason why the following code doesn't compile,

```
main()
{
        _Bool b1, b2;
        int _Bool;
        b1 = 0;
        b2 = !b1;
}
```

but the corresponding `_bit` variable code compiles.

```
main()
{
        _bit b1, b2;
        int _bit;
        b1 = 0;
        b2 = !b1;
}
```

Here are some more differences between `_Bool` and `_bit`. Consider these two bits of code.

```
main()
{
        _bit b1, b2;
        b1 = 0;
        b2 = !b1;
}
```

and

```
main()
{
        _Bool b1, b2;
        b1 = 0;
        b2 = !b1;
}
```

The difference in the assembly output of these two bits of code is as follows: (rest of the code is same) for `_Bool`

```
ld.bu   %d15, [%a10] 7
```

and for `_bit`

```
ld.b    %d15, [%a10] 7
extr.u  %d15, %d15, 0, 1
```

The difference in this assembly output is dealt with in the next section by stepping through the entire compilation process.

5.4 The Stages of Translation

In this subsection, we provide a line by line analysis of how code using `_bit` and `_Bool` is modified along the build process from the C source code to the assembly code.

Firstly we consider the following code:

```
_Bool b1, b2;
b1 = 0;
b2 = !b1;
```

Here is the relevant sections from the RTL for the translation of the first line i.e. `b1 = 0;`

```
(insn 10 7 11 (nil) (set (reg:QI 38)
        (const_int 0 [0x0])) -1 (nil)
    (nil))

(insn 11 10 12 (nil) (set (mem/f:QI (plus:PSI (reg/f:PSI 33 virtual-stack-vars)
                (const_int -1 [0xffffffff])) [0 b1+0 S1 A8])
        (reg:QI 38)) -1 (nil)
    (nil))
```

The first such insn in the RTL sets a register value to 0, and the second moves the value stored in the same register to a QI memory location. Here is how it looks in the Assembly Code

```
mov     %d15, 0          ; the first insn
st.b    [%a10] 7, %d15   ; the second insn
```

The code from the Machine Description file used to generate these two lines are instances of `movqi_internal`

```
(define_insn "movqi_internal"
  [(set (match_operand: QI 0 "nonimmediate_operand" "=d,*d,a,*a,m,d,d,a")
        (match_operand: QI 1 "general_operand"       "d,*a,a,*d,d,m,i,i"))]
  "register_operand(operands[0],QImode) ||
register_operand(operands[1],QImode)"
  {
        output_move_qi (operands, &insn);
        return "";
  }
  ...
)
```

The instances used are moving of integer to register, and from register to memory.

Now we analyse how the code for the other line `b2 = !b1;` is generated. The RTL code for this is

```
(insn 13 11 14 (nil) (set (reg:SI 39)
        (zero_extend:SI (mem/f:QI (plus:PSI (reg/f:PSI 33 virtual-stack-vars)
                    (const_int -1 [0xffffffff])) [0 b1+0 S1 A8]))) -1 (nil)
    (nil))
(insn 14 13 15 (nil) (set (reg:SI 40)
        (eq:SI (reg:SI 39)
            (const_int 0 [0x0]))) -1 (nil)
    (nil))
(insn 15 14 16 (nil) (set (reg:QI 41)
        (subreg:QI (reg:SI 40) 0)) -1 (nil)
    (nil))
(insn 16 15 17 (nil) (set (mem/f:QI (plus:PSI (reg/f:PSI 33
virtual-stack-vars)
                (const_int -2 [0xfffffffe])) [0 b2+0 S1 A8])
        (reg:QI 41)) -1 (nil)
    (nil))
```

This is what each of these lines do:

1. Extends the QI register by padding it with zeroes and saves it in a SI Mode register.
2. Sets the value of a new register equal to the value obtained on comparing the first register with 0.
3. Takes the subregister of the SI Mode register to give a QI Mode value once again.
4. Sets the value of the memory location to be the value in the register.

Here is the assembly code for the same.

```
        ld.bu   %d15, [%a10] 7
        eq      %d15, %d15, 0
        st.b    [%a10] 6, %d15
```

The first two instructions directly correspond to the first two insns in the RTL code. By comparing the RTL dumps just after AST to RTL conversion and just before assembly code generation, we see that the third and fourth insns are combined during the processing of the RTL IR. The final (after combining) MD insn used is a `movqi_internal2`.

Here are the MD lines corresponding to the above:

```
(define_insn "zero_extendqisi2"
  [(set (match_operand:SI 0 "register_operand"                    "=d,d")
        (zero_extend:SI (match_operand:QI 1 "nonimmediate_operand" "d,QS")))]
  ""
  "@
  extr.u\t%0, %1, 0, 8
  ld.bu\t%0, %1"
  [
   (set_attr_alternative "pipe" [
                                 (const_string "ip")
                                 (const_string "lsp")
                                ])
  ]
 )
```

```
(define_insn "seq_internal1"
  [(set (match_operand:SI 0 "data_register_operand"         "=d,d,d")
        (eq:SI (match_operand:SI 1 "register_operand" "%d,a,a")
               (match_operand:SI 2 "register_or_shortint_operand"
"dI,a,O")))]
  ""
  "@
   eq\t%0, %1, %2
   eq.a\t%0, %1, %2
   eqz.a\t%0, %1"
  [(set_attr_alternative "pipe" [(const_string "ip")
                                 (const_string "dual")
                                 (const_string "dual")])])
```

The code for the other line in the assembly code `movqi_internal` as is already given above. The instances used in the first MD insn is that of `d` and `QS`, while the arguments for the second match are `d, d, dI`.

We now explore the corresponding code for `_bit`. The first line of `b1 = 0` matches the following RTL code

```
(insn 27 5 10 0 (nil) (set (reg:BI 15 d15)
        (const_int 0 [0x0])) 25 {movbi_internal2} (nil)
    (nil))

(insn 10 27 12 0 0xf6dd2294 (set (mem/f:BI (plus:PSI (reg/f:PSI 26 a10)
                (const_int 7 [0x7])) [0 b1+0 S1])
        (reg:BI 15 d15)) 25 {movbi_internal2} (nil)
    (nil))
```

The assembly code is identical to that obtained for `_Bool`. The MD file function used for this conversion is `movbi_internal2`:

```
(define_insn "movbi_internal2"
  [(set (match_operand:BI 0 "register_or_memory_operand" "=d,d,d,d,d,B,B,Z,m,a,a")
        (match_operand:BI 1 "general_operand" " B,m,M,d,a,d,M,M,d,d,a"))
  ]
  ""
  {
     switch (which_alternative) {
       case 2:
                tric_set_symbol_at_reg(operands [0],0);
            return "mov\t%0,%1";
      case 8:
                if (tric_get_symbol_at_reg(operands[1]) != 0) {
                        output_asm_insn("extr.u\t%1, %1, %b1, 1",operands);
                        tric_set_symbol_at_reg(operands[1],0);
                }
                return ("st.b\t%0,%1");
     }
    return "";
```

```
  }
  ...
)
```

The cases that execute correspond to moving `M` (constant integer in the range 0-32) → `d` (for the `mov`) and moving `d` → `m` for the `st.b`.

Now for the second line of code, we have the following RTL insns:

```
(insn 12 10 13 0 0xf6dd2294 (set (reg:SI 15 d15 [38])
        (zero_extend:SI (mem/f:BI (plus:PSI (reg/f:PSI 26 a10)
                    (const_int 7 [0x7])) [0 b1+0 S1]))) 28 {zero_extendbisi2} (nil)
    (nil))

(insn 13 12 14 0 0xf6dd2294 (set (reg:SI 15 d15 [39])
        (eq:SI (reg:SI 15 d15 [38])
            (const_int 0 [0x0]))) 43 {seq_internal1} (nil)
    (nil))

(insn 14 13 16 0 0xf6dd2294 (set (mem/f:BI (plus:PSI (reg/f:PSI 26 a10)
                (const_int 6 [0x6])) [0 b2+0 S1])
        (reg:BI 15 d15 [39])) 25 {movbi_internal2} (nil)
    (nil))
```

while the assembly code is

```
        ld.b    %d15, [%a10] 7
        extr.u  %d15, %d15, 0, 1
        eq      %d15, %d15, 0
        st.b    [%a10] 6,%d15
```

Note that the number of RTL insns generated initially is 3, so no combining of insns occurs.

The MD insns corresponding to these are:

```
(define_insn "zero_extendbisi2"
  [(set (match_operand:SI 0 "register_operand"                      "=d,d,d")
        (zero_extend:SI (match_operand:BI 1 "nonimmediate_operand" " d,B,m")))]
  ""
   {
        switch (which_alternative) {
        case 0:
                output_asm_insn("extr.u %0, %1, %b1, 1",operands);
                tric_set_symbol_at_reg(operands[0],0);
                return "";
        case 1:
                tric_set_symbol_at_reg(operands[0],0);
                return("ld.b\t%0, %1\;extr.u\t%0, %0, bpos:%1, 1");
        case 2:
                tric_set_symbol_at_reg(operands[0],0);
                return("ld.b\t%0, %1\;extr.u\t%0, %0, 0, 1");
        }
```

```
        return "";
   }
   ...
)

(define_insn "seq_internal1"
  [(set (match_operand:SI 0 "data_register_operand"         "=d,d,d")
        (eq:SI (match_operand:SI 1 "register_operand" "%d,a,a")
               (match_operand:SI 2 "register_or_shortint_operand"  "dI,a,O")))]
  ""
  "@
   eq\t%0, %1, %2
   eq.a\t%0, %1, %2
   eqz.a\t%0, %1"
  ...
)
```

The `zero_extendbisi2` insn executes case 2, which corresponds to moving `m` → `d`, and outputs the first two assembly instructions, i.e. `ld.b` and `extr.u`. The `seq_internal1` insn outputs `eq` since the operands match the case of moving `m` → `d`. The `st.b` is output by `movbi_internal2` in a manner identical to the previous `st.b`.

5.5 `_Bool` in GCC

GCC supports the following C standards (while also allowing GNU extensions to these standards):

- ISO C90 (also C90 with Amendment 1)
- ISO C99 (not fully supported)

C99 supports the _Bool keyword [4], which declares a two-valued integer type (capable of representing the values true and false). It also provides a standard `<stdbool.h>` header that contains definitions for the following macros:

- `bool` is equivalent to `_Bool`.
- `false` is equivalent to `(_Bool) 0`.
- `true` is equivalent to `(_Bool) 1`.

When compiling C source with GCC, the `-ansi` option (or equivalently `-std=c89`) makes GCC use the C90 standard. Even when this option is not specified, it is still possible to use some of the features of newer standards as far as they do not conflict with previous standards. The GCC man pages [3], and other documents [2] state that

> The `-ansi` option does not cause non-ISO programs to be rejected gratuitously. For that, `-pedantic` is required in addition to `-ansi`.

However, we attempted to compile the following program with the `-ansi -pedantic` options on both Tricore GCC and GCC 4.0 for i386:

```
main()
{
 _Bool b;
 b = 0;
}
```

In both cases, the program compiled successfully. Further investigation is required to ascertain the reason for the successful compilation of the above program given the `-ansi -pedantic` option.

5.6 Miscellaneous Experiments

In this section, we present code for various experiments carried out by us to ascertain the true depth to which this port of Tricore happens with respect to bit variables.

5.6.1 Differences between a typedef and a `_bit`

In this experiment, we try to find the difference between a typedef to a char and a `_bit`. We find that the assembly code for the first case (i.e typedef of `_fun` to char, and using variables of `_fun`) and the same case (i.e. use of an `_bit` variable) is the same, indicating that `_bit` is effectively the same as typedef to a char.

```
main()
{
        _bit myint; // initializing a variable as one of _bit
        myint = 1;
}
```

```
   Assembly
   --------
   sub.a   %SP, 8          # allocate space for locals
   call    __main
   mov     %d15,1
   st.b    [%a10] 7,%d15
   mov     %d2, %d15
   ret
```

```
main()
{
        typedef char _fun;
        _fun varmax; // varmax is a typedef-ed char
        varmax = 1;
}
```

```
   Assembly
   --------
   sub.a   %SP, 8          # allocate space for locals
   call    __main
   mov     %d15, 1
   st.b    [%a10] 7, %d15
```

```
mov     %d2, %d15
ret
```

Note that this means that there are basically no bit-level assembly instructions in the Tricore instruction set (atleast for this simple sample program). Instead bit operations are performed by generating `extr` instructions to extract specific bits from bytes in registers.

5.6.2 `signed` and `unsigned` Qualifiers

The following program does not compile with Tricore GCC, indicating that `_bit` cannot be qualified with the `signed` or `unsigned` qualifiers:

```
main()
{
  signed _bit bs;
  unsigned _bit us;
  bs = -1;
  us = 1;
  us = (unsigned _bit) bs;
  return;
}
```

6 Other Changes Made To GCC For Porting Tricore

6.1 Optimizations Made By HighTec

Here is an example of how a few changes are made to provide certain optimizations to how code is generated. Here is a snippet of code from `stor-layout.c`.

⋮

```
#ifndef __TRICORE__
  if (SLOW_BYTE_ACCESS && ! volatilep)
#endif
    {
      enum machine_mode wide_mode = VOIDmode, tmode;
      ...
      return wide_mode;
    }
  return mode;
```

⋮

This code is interesting because, as a nearby comment indicates, the `if` condition is there to check for slow byte access (i.e. memory access to individual bytes are slower than accesses to individual machine words), and return a larger operation mode which would be more suited for such an architecture. But Tricore conditionally doesn't compile this check and instead forces the block of code to be executed always. This is because Tricore always takes the largest mode since it is better at accessing words than bytes.

7 Some Interesting Observations

7.1 From `stdbool.h`

`stdbool.h` is a file required by the ISO C99 standard. Among other things, it defines `bool` as equivalent to `_Bool`. The snippet below shows how this definition is performed:

⋮

```
#ifndef __cplusplus

#define bool    _Bool
#define true    1
#define false   0

#else /* __cplusplus */

/* Supporting <stdbool.h> in C++ is a GCC extension.  */
#define _Bool   bool
#define bool    bool
#define false   false
#define true    true

#endif /* __cplusplus */
```

⋮

The above code also includes some `#define` statements that are processed for C++ programs (in which case the constant `__cplusplus` is defined). Since in C++, `bool` is a reserved keyword (and an atomic data type), the above code defines `_Bool` as equivalent to `bool`. However, the utility of the next three `#define` statements is not entirely clear, as they seem to be redundant.

8 Conclusions and Future Work

We have noticed the following while working on this project:

- The porting for Tricore goes deeper than we had initially expected and changes a number of files in the *generic* component (which, according to us should not be changed)
- The effect of these additional bits of code, plus additional macros may not be achievable by restricting the changes made by the port to one single directory (especially the situations when certain optimizations are used). In other words, the goal of the ideal port may not be feasible.
- The facility of bit variables is a not-so-clean way (in other words, a hack) to introduce a new construct while the parser file is fixed, and to overcome the fact that bison does not support conditional compilation.
- However it may be possible to create a patch to introduce the bit variable changes to pristine GCC in order to give this version suitable for Tricore.
- The `-ansi -pedantic` option in GCC seems to be flouting the rules of the C90 standard, which seems rather odd.

- A classification of changes made by the port to the main GCC source tree is possible, however the outlook for separating these changes (and optimizations) from the main source tree remains bleak.
- Porting GCC in general is a hard task because it requires intricate knowledge of the GCC source internals and the target machine architecture.
- Even porting Tricore GCC from 3.3 to 4.0 is not easy because there are a large number of changes in the GCC structure which would need to be addressed in the port.
- It is not entirely clear as to what more features `_bit` provides over `_Bool`. We believe that any optimizations made to the main GCC sources for `_bit` may be applied to the `_Bool` case. This way, any target architecture requiring support for bit-level manipulations can rely on an efficient standard data type.

Here are the possible areas of future work:

- A patch for bit variables could be attempted to clean up the port of Tricore.
- Now that we have identified the various classes of changes made by Tricore while adapting it for GCC 3.3, a port to GCC 4.0 may be attempted.
- It may be possible to construct a tool to automate the migration of machine descriptions and machine-specific macros from the GCC 3.3.3 specifications to the GCC 4.0.0 specifications. More investigation in this direction will be required.
- A tool for building a tool chain comprising of `binutils`, `gcc`, and `glibc` for Tricore that allow seamless specification of target, host, languages, paths and the other compile options.
- This may be a bit optimistic, but there should be a general way (by making changes to the original GCC source) to introduce these optimizations when there is a possibility that they work. These optimizations could be selectively enabled or disabled via macros/target hooks in the Machine Description file. This basically means moving common machine-specific changes to the main GCC sources and introducing macros/target hooks to be specified in the target-specific header files to selectively enable these optimizations. Note that this would mean that a study of the optimizations generally employed in various architectures would have to be collated and flags to enable or disable them would need to be specified in the *generic* gcc source.
- Further investigation into why the `-pedantic` option in GCC does not force adherence to the C90 standard, as seen in Section 5.5, and discovery of other instances where the C90 standard is not enforced in `-pedantic`.
- Further studies on whether `_bit` can be made unneccessary by increasing the functionality of `_Bool` may be attempted. This could be done by checking whether the assembly code generated for various programs highlighting the use of bit operations has any items that is more efficient than the corresponding `_Bool` version using one byte.

References

[1] Stallman, R. M. : GNU Compiler Collection Internals for GCC 3.3. Published by the *Free Software Foundation*.

[2] Stallman, R. M. : Using the GNU Compiler Collection (GCC 3.3). Published by the *Free Software Foundation*.

[3] gcc, g++ - GNU project C and C++ Compiler (gcc-3.3): Man Pages for GCC.

[4] Tribble, D. R.: Incompatibilities between ISO C and ISO C++. At *http://david.tribble.com/text/cdiffs.htm*.