Incremental Machine Descriptions for GCC

Sameera Deshpande Uday P. Khedker

Indian Institute of Technology, Bombay

{sameera,uday}@cse.iitb.ac.in

Abstract

The mechanism of providing machine descriptions to the GCC framework has been quite successful as demonstrated by a wide variety of the targets for which a GCC port exists. However, this mechanism is quite ad hoc and the machine descriptions are difficult to construct, understand, maintain, and enhance because of the verbosity, the amount of details, and the repetitiveness. The publicly available material fails to bring out the exact abstractions captured by the machine descriptions. There is no systematic way of constructing machine descriptions and there are no clear guidelines on where to begin developing machine description and how to construct them systematically. This paper proposes a methodology based on incremental construction of machine descriptions starting from a well defined minimal machine description. We illustrate the process by constructing machine descriptions with the `spim` simulator for MIPS architecture as the target.

1. Introduction

The GCC framework generates a compiler for a given architecture by reading the machine descriptions for that architecture. Although the mechanism of GCC machine descriptions seems to be practically useful, it is less than satisfactory primarily because it is quite ad hoc. As a consequence, the machine descriptions are difficult to construct, understand, maintain, and enhance because of the verbosity, the amount of details, and the repetitiveness. The sheer size of machine descriptions is formidable. For example, the directory `gcc/config/i386` in `gcc-4.1.1` contains 63103 lines. Further, a simple comparison of macros defined in various machine descriptions reveals the fact that among the descriptions available in any distribution, there are more variations than similarities (63 macros are common to all machine descriptions in `gcc-4.1.1` whereas, together, they define close to 769 distinct macros).

Unfortunately, not much discussion seems to be centered around these concerns. Most explanations of GCC which are publicly available (including those presented through special workshops and tutorials) describe the build and install process of GCC, the GCC front end, the IRs used by GCC and their manipulations by the optimization phases in GCC, and the structure of machine descriptions required by GCC. Although several dozen actual machine descriptions are readily available, one does not come across much information on the insights behind machine descriptions. As a consequence, rather than developing a new port from scratch, in practice, a new port is started with an existing machine description for an architecture of a machine that is close to the new target. This is a tedious and error-prone process resulting in machine descriptions are not easy to understand or modify.

Thus it become important to address the following questions:

- Is there a systematic way of developing GCC machine descriptions? Can one define the notion of the *minimal* machine descriptions for a given target to which features can be added in small well-defined steps?
- Can one create more easily understandable abstractions of GCC machine descriptions?
- Can the GCC machine descriptions be any simpler?

As a part of our long term investigations aimed at answering the above questions, this paper attempts to answer the first question above. At the moment, the proposed methodology does not address the issue of the quality of generated code and restricts itself to understanding the abstractions in the machine descriptions. More details of the proposed methodology are available the web page of the GCC workshop [1].

For simplicity, we restrict the notion of compilation to the process of generating assembly code and ignore the subsequent steps.

2. Incremental Construction of Machine Descriptions

The process of compilation and its implications on retargetability are influenced by the following three factors:

1. The phase and pass structure of a compiler.

   This comprises of various transformations that the source program undergoes as illustrated in standard textbooks [2]. These transformations can be either

   (a) transformations within the same intermediate representation (IR), or

   (b) transformations to convert the program from one IR to another.
2. The features of the source language.

   These include the primitive operators supported by language, control structures, calling conventions, scope rules, data types supported by the language etc.
3. The features of the target architecture.

   These include the instruction set, registers, addressing modes, data and memory layout etc.

These factors are heavily dependent on each other and their influence on machine descriptions cannot be studied in isolation from each other. Even if other factors are abstracted out, the influence of a given factor is not easy to understand because:

- Compilation phases may be independent in principle, but in practice, they are related by IRs which are quite complex and heavily dependent on the needs of different phases.
- Translation of many source features depend upon other source features.
- Many target architecture features are dependent on each other.

As a consequence, it does not seem possible to systematically classify the information present in machine descriptions in order to discover the abstractions present in them.

Instead of the "declarative" approach of explicitly defining abstractions present in machine descriptions, we present an "operational" approach of systematically constructing machine descriptions such that the process uncovers the abstractions. This approach is based on the following two crucial observations:

- Unless optimization is the main concern, source language features influence the machine descriptions more than the target language features or the phase structure of a compiler.
- It is not the partitioning of the source language features but their incremental accumulation which influences the machine descriptions systematically. In particular, if the increments are identified properly, the corresponding increments in machine description are monotonic in that no feature described earlier needs to change (unless a dummy value had to be defined for it.)

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

The subsequent levels are defined as follows and have been illustrated in Figure 1:

- Level 1: Assignment statements involving on integer constants and variables.
- Level 2: Arithmetic operations on integer data type.
- Level 3: Function handling and calling conventions.
- Level 4: Control structures.

We illustrate this process for the `spim` simulator of MIPS [3].

We identify the minimum information required in machine descriptions to support each level. The machine description for a given level is said to support that level if

- The compiler for that level gets built successfully.
- Any program written using source language features supported by corresponding level can be successfully compiled.
- Generated assembly program is executed correctly in `spim`.

Once the basic machine description with all language features is written, the advanced target features can be added on top of the machine description incrementally.

3. `spim` Machine Description for Level 0

The goal of level 0 is to build the minimal machine description successfully. We further divide the level 0 machine descriptions such that

- Level 0.0 merely builds GCC successfully for `spim`. The `xgcc` built does not compile even a single program.

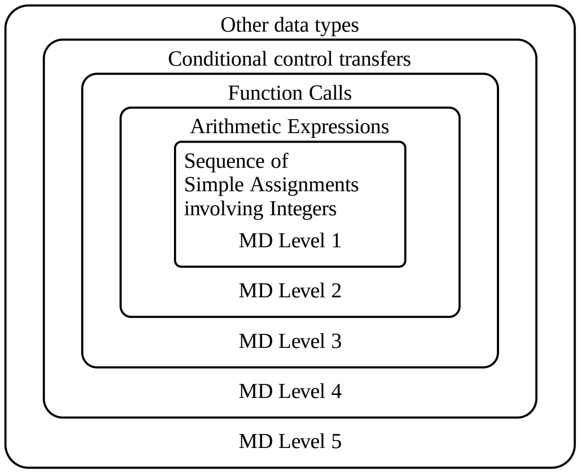

Figure 1. Systematic Development of Machine Description

- Level 0.1 adds small increment to level 0.0 so as to build GCC successfully such that the generated `xgcc` compiles empty void functions of the form:

```
void fun(int param1, int param2,...,int paramN)
{
 int var1,var2,..,varM;
}
```

  As long as parameters and local variables are not referred, the `xgcc` built for level 0.1 can compile the program successfully. As level 0 does not support any high level language statement, the program containing statement list are bound to fail for level 0.1. When return type is changed from void to int, GCC tries to generate RTL to move dummy return value into return value register `$v0`, for which assignment statement is required, which is not supported in this level. Hence, only empty void functions can be compiled.
- Level 0.2 machine description incorporates complete activation record structure. Although it is irrelevant for compiling empty void functions, it sets stage for level 1 compiler for which knowledge of location of variables being used in assignment operation is necessary, to generate semantically correct code.

Although level 0 supports minimal functions, the retargetability mechanism in GCC still requires plenty of details to be supplied to the compiler generation framework. This information is divided in the following categories:

1. Memory Layout
2. Supported Instructions
3. Registers
4. Addressing Modes
5. Activation Record Conventions

We describe them in the following subsections.

3.1 Memory layout issues

In memory layout related issues, following details are needed:

Bit, byte and word endianness: The `spim` simulator assumes the endianness of the underlying architecture on which the simulator is being executed. For our experiments, the underlying architecture is `i*86` and all entities are ordered in little endian manner.

Alignment boundaries: Alignment is not enforced strictly by the `spim` simulator. We have chosen to align stack data at 64 bits whereas words are aligned to 32 bits.

3.2 Supported Instructions

Level 0.0 does not support any high level language statement. Yet some details are required because of the following two reasons:

- Since no operation needs to be supported, in principle, the `.md` can be empty. However, this results in the declaration of initialized empty array `insn_conditions`[1] in the generated source. As a result, the build for level 0.0 compiler crashes due to `-pedantic` option supplied by the build process to the native C compiler. We overcome this problem by including the following dummy pattern in the `.md` file.

  ```
  (define_insn "dummy_pattern"
      [(reg:SI 0)]
      "1"
      "This stmnt should not be emitted!"
  )
  ```

  There is nothing else in the `.md` file.

- Since the compiler adds a common exit to each function, it expects the presence of direct and indirect jump instructions. Since our programs are empty programs, the jump to common exit gets optimized away in later RTL passes. Therefore, jump and indirect jump patterns need not exist in the `.md` file. However, we have to make GCC believe that the jump instructions have been provided. We do so by providing dummy definitions of macros and functions for which build process crashes. Though the macro `CODE_FOR_indirect_jump` and functions `gen_jump` and `gen_indirect_jump` have their own semantics in GCC, for level 0.0 we write dummy definitions for the macro `CODE_FOR_indirect_jump` as

  ```
  #define CODE_FOR_indirect_jump 8
  ```

  GCC generates functions `gen_jump` and `gen_indirect_jump` from the patterns in the `.md` file associated with the standard pattern names `indirect_jump` and `jump`. Since we do not have these patterns, we include dummy definitions shown below in the `.c` file.

  ```
  rtx gen_jump
  (rtx operand0 ATTRIBUTE_UNUSED)
  {
     return 0;
  }

  rtx gen_indirect_jump
  (rtx operand0 ATTRIBUTE_UNUSED)
  {
     return 0;
  }
  ```

  Our `.md` file continues to have only the dummy pattern in level 0.0.

[1] This array holds condition codes for various patterns defined in the `.md` file.

For level 0.1, the compiler requires following additional information in order to compile an empty function to a valid assembly program accepted by `spim` simulator:

- Assembly formats
- Jump instructions.

  The dummy definitions for macro `CODE_FOR_indirect_jump` and functions `gen_jump` and `gen_indirect_jump` provided in level 0.0 are not sufficient for compiling void empty programs and generating the corresponding assembly program. Hence, we define the patterns for `jump` and `indirect_jump` in the `.md` file as shown below:

```
(define_insn "jump"
        [(set (pc)
              (label_ref
                (match_operand 0 "" "")))]
        ""
        "j \\t%l0"
)

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

- Return instruction.

  The GCC standard pattern `return` is to be used only if the target has a single assembly instruction that is sufficient for all the work of returning from a function. In particular, the instruction must destroy the activation record. However, in `spim`, the return is effected through an indirect jump to the return address register `$ra`. This does not destroy the activation record. Therefore, instead of using the GCC standard pattern `return`, we use the `epilogue` standard pattern which emits the `spim` indirect jump instruction as a part of dismantling the activation and returning to the caller.

  The full epilogue is required when function calls are fully supported. Until that stage, the epilogue is gradually built across levels.

  For the present level, we generate the return through standard pattern `epilogue` as shown below.

```
(define_expand "epilogue"
        [(clobber (const_int 0))]
        ""
        {
                spim_epilogue();
                DONE;
        }
)

void spim_epilogue()
{
        emit_jump_insn(gen_IITB_return());
}

(define_insn "IITB_return"
        [(return)]
        ""
        "jr \\t\\$ra"
)
```

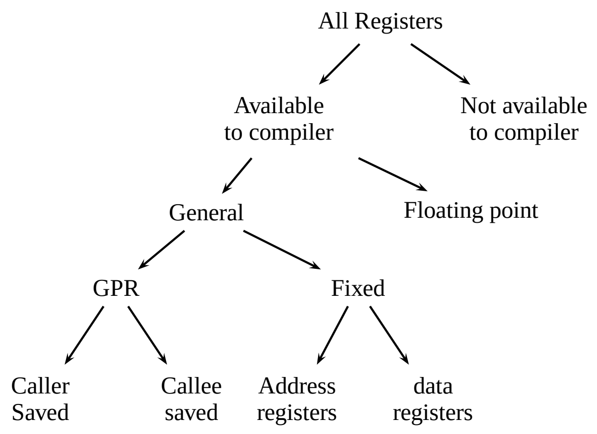

Figure 2. Register Class Hierarchy

Note that in this level, there is no activation record to be destroyed.

As a side effect of inclusion of jump instruction, the compiler so generated can also compile the program

```
void foo()
{
   L: goto L;
}
```

Level 0.2 does not support any additional instruction.

3.3 Addressing Mode Issues

In level 0, only `jump` and `indirect_jump` have been supported. Hence, the only addressing modes supported in level 0 are

- For address of data: absolute addressing.
- For address of code: absolute and register indirect addressing as shown below.
  - Register indirect addressing, e.g. `jr $ra`
  - Absolute addressing, e.g. `j L2`

Macro `GO_IF_LEGITIMATE_ADDRESS` is used for data addresses and is defined in such a way that no address other than constant addresses is considered legitimate.

3.4 Register Specific Information

`spim` contains 32 32-bit general purpose registers[4].

- Register 0 contains value 0.
- Registers `$at` and `$k0`, `$k1` are reserved as assembler and kernel registers respectively, and hence are not available to compiler. We can either remove them completely from specification or mark them as `FIXED_REGISTERS`.
- Registers `$a0` to `$a3` an be used to pass arguments to the function. This is irrelevant till level 0.1 as activation record related issues are not handled in these levels. For level 0.2, as all arguments are passed on stack, these registers are used as other general purpose registers only.
- Registers `$t0` to `$t9` are caller saved registers. i.e. it is caller's responsibility to save these registers, and callee can use these registers freely. Hence, these registers must be marked as call clobbered.

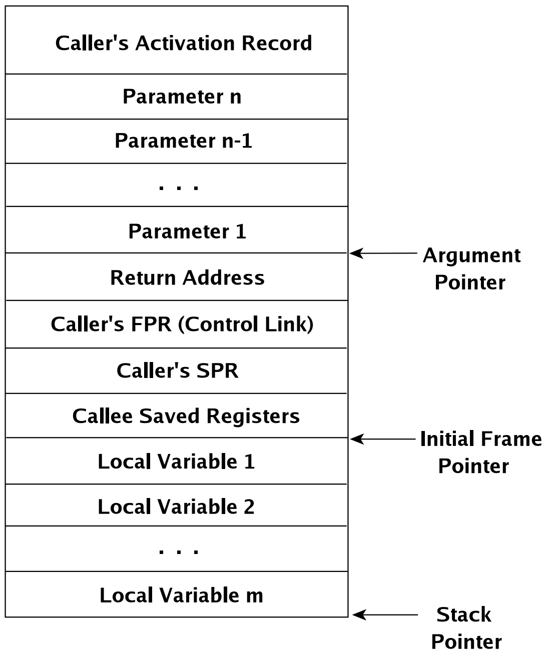

Figure 3. Activation Record Design for `spim`

- Registers `$s0` to `$s7` are callee saved registers, and they must be saved by callee's prologue, and restored back by callee's epilogue. As in level 0, no register will be used, prologue and epilogue need not store any of the registers.
- Registers `$gp`, `$sp` and `$fp` are global pointer registers, stack pointer and frame pointer respectively.
- `$ra` is return address register, which is used in function epilogue, to return to caller.

These registers can be classified in different groups as per requirement. The classification of registers is shown in Figure 2.

3.5 Activation Record Specific Information

Though activation record is not required in level 0, activation record design is completely orthogonal issue. Hence, we can design activation record in level 0.2 which can be used in level 1. The activation record designed for `spim` is shown in Figure 3. While describing activation record to GCC, following information must be provided through macro definitions:

- Direction of growth of stack frame: pushing a word onto the stack moves the stack pointer to a smaller address.
- Direction of growth of local variable frame: Local variable frame grows in same direction of stack.
- Direction of growth of Parameter frame: Parameter frame grows in opposite direction of growth of stack.
- Position where current stack pointer `$sp` points: It is assumed in `spim` that the stack pointer always points to first empty slot on the stack.
- Position where current frame pointer `$fp` points: Because of design of activation record, the callee saved registers lie between parameters and local variable frame. As argument pointer and frame pointer in `spim` are same, the offset calculation of both, parameters and local variables is done with reference to the frame pointer register. However, the variables as well as parameters are referred well before register allocation phase. Hence

the computed offsets before register allocation might be different from actual offsets after register allocation. In such situations, we have to make use of dummy frame pointer initially, which gets eliminated to final frame pointer after register allocation is done. As shown in Figure 3, the dummy frame pointer points to the location where first local variable is saved.

- Position where current arguments pointer AP points: The argument pointer points to the location from where first parameter can be obtained.
- Relative offsets of fields from address registers in activation record:
  - Parameters passed to the function: The first argument's location from argument pointer is specified using the macro `FIRST_PARM_OFFSET`. As shown in the figure, this offset is 0 for `spim`.

    ```
    #define FIRST_PARM_OFFSET(FUN) 0
    ```

    The offset at which outgoing parameters are to be placed is given by macro `STACK_POINTER_OFFSET` which gives offset of first parameter to be passed from stack pointer.

    ```
    #define STACK_POINTER_OFFSET 0
    ```
  - Callee saved registers: This is callee's responsibility, and is handled by function prologue. Even though, this information will not be needed until function calls are incorporated in the language features (which is actually done in level 3), the number of registers stored by callee is required for proper offset computation of other fields. Hence total number of registers stored is computed as follows:

    ```
    int
    registers_to_be_saved
    (void)
    {
      int i,num;
      for(i=0,num=0;
          i<FIRST_PSEUDO_REGISTER;
          i++)
      {
        if(regs_ever_live[i]
           && !call_used_regs[i]
           && !fixed_regs[i])
        num++;
      }
      return num;
    }
    ```
  - Return address: In `spim`, return address is passed in register `$ra`. Hence it becomes callee's responsibility to store return address register on stack, if required. However, for accurate offset computations, knowing number of words reserved for return address is necessary.
  - Previous activation's pointer: Previous activation record's pointers can be obtained from stack pointer, frame pointer and argument pointer registers. In `spim`, frame pointer and argument pointer registers are same. Hence, we store stack pointer and frame pointer in activation record. However, since it is callee's responsibility, we just compute the number of bytes to be allocated on stack for proper offset computation of other fields.
  - Return value: The register in which the value to be returned is stored is given by macro `FUNCTION_VALUE`. In `spim` register number 2 serves this purpose. Hence, this can be specified as:

    ```
    #define FUNCTION_VALUE(valtype, func)\
    function_value()
    rtx
    function_value
    (void)
    {
      /* Return register is register 2
       * when value is of type SImode.*/
      return (gen_rtx_REG(SImode,2));
    }
    ```
  - Local frame: The location of first local variable allocated on stack from dummy frame pointer can be specified by macro `STARTING_FRAME_OFFSET`.

    ```
    #define STARTING_FRAME_OFFSET \
    starting_frame_offset ()

    int
    starting_frame_offset
    (void)
    {
      return 0;
    }
    ```
- Relative offsets of frame pointer and argument pointer registers from stack pointer: This information can be specified by defining the macros `ELIMINABLE_REGS, CAN_ELIMINATE` and `INITIAL_ELIMINATION_OFFSET` as follows:

```
#define ELIMINABLE_REGS\
{\
{FRAME_POINTER_REGNUM,
   STACK_POINTER_REGNUM},\
{FRAME_POINTER_REGNUM,
   HARD_FRAME_POINTER_REGNUM},\
{ARG_POINTER_REGNUM,
   STACK_POINTER_REGNUM},\
{HARD_FRAME_POINTER_REGNUM,
   STACK_POINTER_REGNUM}\
}

#define CAN_ELIMINATE(FROM, TO) \
((FROM == FRAME_POINTER_REGNUM
  && (TO == STACK_POINTER_REGNUM
      || TO == HARD_FRAME_POINTER_REGNUM)) \
|| (FROM == ARG_POINTER_REGNUM
    && TO == STACK_POINTER_REGNUM) \
|| (FROM == HARD_FRAME_POINTER_REGNUM
    && TO == STACK_POINTER_REGNUM))

#define INITIAL_ELIMINATION_OFFSET(FROM,TO,V)\
(V) = initial_elimination_offset(FROM,TO)
```

4. Machine Description for Level 1 of `spim`

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

- Some dummy definitions of macros are replaced by meaningful definitions. These include the macros related to addressing modes and assembly format.
- RTL patterns for newly added operations are included in the `.md` file.

HLL Operation	Primitive Variants	Implementation
Dest $\leftarrow$ *Src*	$R_i \leftarrow R_j$	`move rj, ri`
	$R \leftarrow M$	`lw r, m`
	$R \leftarrow C$	`li r, c`
	$M \leftarrow R$	`sw r, m`
RETURN *Src*	RETURN	`$v0` $\leftarrow$ *Src* `j $ra`
Dest $\leftarrow$ *Src*$_1$ + *Src*$_2$	$R_i \leftarrow R_j + R_k$	`add ri, rj, rk`
	$R_i \leftarrow R_j + C$	`addi ri, rj, c`

Table 1. Instructions supported in Level 1.

4.1 Supported Instructions

Level 1 primarily supports assignment operations whose destination can be a register or a memory location; the source can be a register, a memory location or a constant value. Depending upon whether the variable is defined globally or locally, addressing mode for the memory operand varies. The memory location can be addressed using symbol associated with name of variable or location of the variable in activation record. The instructions supported in level 1 are listed in Table 1.

Inclusion of RETURN instruction is to ensure correctness of the assembly code generated. The implementation of return instruction as shown in Table 1 says that while executing the high language return instruction, compiler first moves the value to be returned into return value register `$v0`, using one of the move instructions given in same table, followed by jump to return address, given in register `$ra`.

The inclusion of addition operation is necessitated for emitting prologue as explained in the following. Though we have designed activation record in level 0, we haven't constructed the activation record in that level as it was not needed. In level 1, we construct activation record as shown in Figure 3 by defining RTL patterns for function prologue and epilogue.

```
(define_expand "prologue"
        [(clobber (const_int 0))]
        ""
        {
                spim_prologue();
                DONE;
        }
)
void
spim_prologue(void)
{
 int i,j;
 emit_move_insn(gen_rtx_MEM(SImode,
     plus_constant(stack_pointer_rtx,0)),
   return_addr_rtx);
 emit_move_insn(gen_rtx_MEM(SImode,
     plus_constant(stack_pointer_rtx,-4)),
   stack_pointer_rtx);
 emit_move_insn(gen_rtx_MEM(SImode,
     plus_constant(stack_pointer_rtx,-8)),
   hard_frame_pointer_rtx);
 emit_move_insn(hard_frame_pointer_rtx,
     stack_pointer_rtx);
 for(i=0,j=3;i<FIRST_PSEUDO_REGISTER;i++){
  if(regs_ever_live[i]
      && !call_used_regs[i]
      && !fixed_regs[i]){
    emit_move_insn(gen_rtx_MEM(SImode,
        plus_constant(hard_frame_pointer_rtx,
            -4*j)),
      gen_rtx_REG(SImode,i));
    j++;
  }}
 emit_move_insn(stack_pointer_rtx,
   plus_constant(hard_frame_pointer_rtx,
      -((3+j)*4+get_frame_size())));
}
(define_expand "epilogue"
        [(clobber (const_int 0))]
        ""
        {
                spim_epilogue();
                DONE;
        }
)
void
spim_epilogue(void)
{
 int i,j;
 for(i=0,j=3;i<FIRST_PSEUDO_REGISTER;i++){
  if(regs_ever_live[i]
      && !call_used_regs[i]
      && !fixed_regs[i]){
    emit_move_insn(gen_rtx_REG(SImode,i),
    gen_rtx_MEM(SImode,
       plus_constant(hard_frame_pointer_rtx,
           -4*j)));
    j++;
  }}
 emit_move_insn(stack_pointer_rtx,
   hard_frame_pointer_rtx);
 emit_move_insn(hard_frame_pointer_rtx,
   gen_rtx_MEM(SImode,
      plus_constant(stack_pointer_rtx,-8)));
 emit_move_insn(return_addr_rtx,
   gen_rtx_MEM(SImode,
      plus_constant(stack_pointer_rtx,0)));
 emit_jump_insn(gen_IITB_return());
}
```

It can be seen from the code snippet given above, that to move stack pointer in activation record for memory allocation, addition operation is required. Hence,as a side-effect of prologue and epilogue definition, add instruction is added in level 1.

4.2 Addressing Mode Issues

Though only assignment operation is supported in level 1, it is evident from Table 1 that the operands of move instruction can be registers, constants along with local and global memory. Hence, we must support all addressing modes in this level. The addressing modes that are supported in `spim` are as follows:

- Absolute addressing:
  - Labels which give absolute address of code memory, for example `jal _fun`.
  - Symbols which represent global data memory, for example `lw $v0, _var`.

  Note that the symbol names are prefixed by an underscore (_). It is because assembler may give error because symbol/label name and instruction mnemonics for `spim` might be the same.
- Register indirect addressing: The address of memory location is stored in register, for example `jr $ra`.

HLL Operation	Primitive Variants	Implementation
Dest $\leftarrow$ *Src*$_1$ $-$ *Src*$_2$	$R_i \leftarrow R_j - R_k$	`sub ri, rj, rk`
Dest $\leftarrow$ $-$*Src*	$R_i \leftarrow -R_j$	`neg ri, rj`
Dest $\leftarrow$ *Src*$_1$/*Src*$_2$	$R_i \leftarrow R_j/R_k$	`div rj, rk` `mflo ri`
Dest $\leftarrow$ *Src*$_1$%*Src*$_2$	$R_i \leftarrow R_j\%R_k$	`div rj, rk` `mfhi ri`
Dest $\leftarrow$ *Src*$_1$ $*$ *Src*$_2$	$R_i \leftarrow R_j * R_k$	`mul ri, rj, rk`
Dest $\leftarrow$ *Src*$_1$ $\ll$ *Src*$_2$	$R_i \leftarrow R_j \ll R_k$	`sllv ri, rj, rk`
	$R_i \leftarrow R_j \ll C_5$	`sll ri, rj, c`
Dest $\leftarrow$ *Src*$_1$ $\gg$ *Src*$_2$	$R_i \leftarrow R_j \gg R_k$	`srav ri, rj, rk`
	$R_i \leftarrow R_j \gg C_5$	`sra ri, rj, c`
Dest $\leftarrow$ *Src*$_1$&*Src*$_2$	$R_i \leftarrow R_j \& R_k$	`and ri, rj, rk`
	$R_i \leftarrow R_j \& C$	`andi ri, rj, c`
Dest $\leftarrow$ *Src*$_1$|*Src*$_2$	$R_i \leftarrow R_j \| R_k$	`or ri, rj, rk`
	$R_i \leftarrow R_j \| C$	`ori ri, rj, c`
Dest $\leftarrow$ *Src*$_1$ ˆ *Src*$_2$	$R_i \leftarrow R_j \hat{} R_k$	`xor ri, rj, rk`
	$R_i \leftarrow R_j \hat{} C$	`xori ri, rj, c`
Dest $\leftarrow\sim$ *Src*	$R_i \leftarrow\sim R_j$	`not ri, rj`

Table 2. Instructions supported in Level 2.

- Base offset addressing: The effective address of memory location is offset plus contents of base register, for example `sw $v0, -20($fp)`.

5. Machine Description for Level 2 of `spim`

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

5.1 Supported Instructions

The instructions supported in level 2 are given in Table 2. Out of these operations, the division ('/') and modulo ('%') require a special treatment because of the implementation of `div` instruction in `spim`. This instruction internally stores the quotient of division in register `lo` and remainder in register `hi`. Hence it becomes the compiler's responsibility to ensure that the result is explicitly moved to the destination registers by using special instructions `mfhi` and `mflo`. The former moves a value from the `hi` register to the specified destination while the latter moves a value from the `lo` register to the specified destination. Thus division and modulo operations in high level language can be seen as compound statements consisting of div instruction followed by move instruction in assembly language. This compound operation can be supported by the following pattern:

```
(define_insn "divsi3"
[(set (match_operand:SI 0 "register_operand" "=r")
   (div:SI
     (match_operand:SI 1 "register_operand" "r")
     (match_operand:SI 2 "register_operand" "r"))
)]
""
"div \\t%1, %2\\n\\tmflo \\t%0"
)
(define_insn "modsi3"
[(set (match_operand:SI 0 "register_operand" "=r")
   (mod:SI
     (match_operand:SI 1 "register_operand" "r")
     (match_operand:SI 2 "register_operand" "r"))
)]
""
"div \\t%1, %2\\n\\tmfhi \\t%0"
)
```

Though it is very simple to add RTL pattern using `define_insn`, and primitive language features are represented as single pattern in the `.md` file, the drawback of this pattern definition mechanism is that it suppresses instruction scheduling: Since the division and modulo operations are expressed by single patterns in all RTL passes and are split directly at assembly level, both `div` and `mflo` or `mfhi` instructions which are actually independent of each other, are treated atomically thereby prohibiting the possibility of being scheduled independently.

In order to enable this optimization, we split these instructions into two independent instructions right from the first RTL IR in the expander pass. The `divsi3` pattern can be defined as follows, which needs definition of named RTL patterns `IITB_divide` and `IITB_move_from_lo` which are used at the time of assembly generation.

```
(define_expand "divsi3"
[(parallel[(set
  (match_operand:SI 0 "register_operand" "")
  (div:SI
    (match_operand:SI 1 "register_operand" "")
    (match_operand:SI 2 "register_operand" "")))
 (clobber (reg:SI 26))
 (clobber (reg:SI 27))])]
""
{
 emit_insn(gen_IITB_divide(gen_rtx_REG(SImode,26),
             operands[1], operands[2]));
 emit_insn(gen_IITB_move_from_lo(operands[0],
             gen_rtx_REG(SImode,26)));
DONE;
}
)

(define_insn "IITB_divide"
[(parallel[(set
  (match_operand:SI 0 "LO_register_operand" "=q")
  (div:SI
    (match_operand:SI 1 "register_operand" "r")
    (match_operand:SI 2 "register_operand" "r")))
(clobber (reg:SI 27))])]
""
"div \\t%1, %2"
)

(define_insn "IITB_move_from_lo"
[(set
  (match_operand:SI 0 "register_operand" "=r")
  (match_operand:SI 1 "LO_register_operand" "q")
)]
""
"mflo \\t%0"
)
```

Similarly, `modsi3` pattern can be defined as follows which makes use of named RTL patterns `IITB_mod` and `IITB_move_from_hi` for expanding.

```
(define_expand "modsi3"
[(parallel[
  (set (match_operand:SI 0 "register_operand" "")
  (mod:SI
    (match_operand:SI 1 "register_operand" "")
    (match_operand:SI 2 "register_operand" "")))
 (clobber (reg:SI 26))
 (clobber (reg:SI 27))])]
""
{
 emit_insn(gen_IITB_mod(gen_rtx_REG(SImode,27),
             operands[1], operands[2]));
 emit_insn(gen_IITB_move_from_hi(operands[0],
            gen_rtx_REG(SImode,27)));
 DONE;
}
)
```

Although this enables instruction scheduling, this method has following drawbacks:

- C interface is needed in `.md` file.
- Compilation becomes slower and requires more space.

Hence, we use `define_split` construct for these instructions.

```
(define_split
[(parallel
  [(set
     (match_operand:SI 0 "register_operand" "")
     (div:SI
       (match_operand:SI 1 "register_operand" "")
       (match_operand:SI 2 "register_operand" ""))
   )
   (clobber (reg:SI 26))
   (clobber (reg:SI 27))])]
 ""
 [(parallel [(set (match_dup 3)
    (div:SI (match_dup 1)
      (match_dup 2)))
  (clobber (reg:SI 27))])
  (set (match_dup 0)
       (match_dup 3))
 ]
 "{operands[3]=gen_rtx_REG(SImode,26); }"
)

(define_split
[(parallel
  [(set
     (match_operand:SI 0 "register_operand" "")
     (mod:SI
       (match_operand:SI 1 "register_operand" "")
       (match_operand:SI 2 "register_operand" ""))
   )
   (clobber (reg:SI 26))
   (clobber (reg:SI 27))])]
 ""
 [(parallel [(set (match_dup 3)
    (mod:SI (match_dup 1)
      (match_dup 2)))
  (clobber (reg:SI 26))])
  (set (match_dup 0)
       (match_dup 3))
 ]
 "{operands[3]=gen_rtx_REG(SImode,27); }"
)
```

HLL Operation	Primitive Variants	Implementation

Dest $\leftarrow$ *fun*$(P_1,\ldots,P_n)$	`call` L_{fun}, n	`lw` r_i, `[SP]`
		`sw` r_i, `[SP]`
		:
		`lw` r_i, `[SP-n*4]`
		`sw` r_i, `[SP-n*4]`

		`jal L`
		Dest $\leftarrow \$v0$

$fun(P_1,P_2,\ldots,P_n)$	`call` L_{fun}, n	`lw` r_i, `[SP]`
		`sw` r_i, `[SP]`
		:
		`lw` r_i, `[SP-n*4]`
		`sw` r_i, `[SP-n*4]`

		`jal L`

Table 3. Instructions supported in Level 3

5.2 Register Specific Information

The registers `lo` and `hi` are internal registers in `spim` which are not introduced to GCC as they were not being used in previous levels. However, as '/' and '%' operations modify contents of these registers by moving quotient in register `lo` and remainder in register `hi`, we introduce these registers to GCC by announcing them in register set. Special register class is created for them as they are not general purpose registers, and are used by specific instructions only.

```
enum reg_class \
{\
    NO_REGS, ZERO_REGS,\
    CALLER_SAVED_REGS,\
    CALLEE_SAVED_REGS,\
    BASE_REGS, HI_REGS,\
    LO_REGS, GENERAL_REGS,\
    ALL_REGS, LIM_REG_CLASSES \
};
```

6. Machine Description for Level 3 of `spim`

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

6.1 Supported Instructions

The sequence of operations performed when a function is called is as follows:

- Operations performed by the caller
  - Push parameters on the stack.
  - Load the return address in the return address register.
  - Transfer control to the callee.
- Operations performed by callee
  - Push the return address register on the stack.
  - Push the caller's frame pointer register on the stack.
  - Push the caller's stack pointer on the stack.
  - Save the callee saved registers, if used by callee on the stack.
  - Create local variable frame on the stack.
  - Start callee body execution.

Operation	Primitive Variants	Implementation
Src$_1 <$ *Src*$_2$? `goto` L : PC	$CC \leftarrow R_i < R_j$ CC $<$ 0 ? goto L : PC	`blt` r_i, r_j,L
Src$_1 >$ *Src*$_2$? `goto` L : PC	$CC \leftarrow R_i > R_j$ CC $>$ 0 ? goto L : PC	`bgt` r_i, r_j,L
Src$_1 \leq$ *Src*$_2$? `goto` L : PC	$CC \leftarrow R_i \leq R_j$ CC $\leq$ 0 ? goto L : PC	`ble` r_i, r_j,L
Src$_1 \geq$ *Src*$_2$? `goto` L : PC	$CC \leftarrow R_i \geq R_j$ CC $\geq$ 0 ? goto L : PC	`bge` r_i, r_j,L

Table 4. Instructions supported in Level 4

Out of these, the operations performed by callee are taken care of by the prologue of the callee which has already been defined. Hence, in this level, we cover tasks performed by the caller in a call instruction. Depending upon whether the callee returns a value or not, there are two variants of a call as shown in Table 3.

The standard pattern named `call` is used to define pattern for subroutine call instruction which does not return a value. It is defined in `.md` file as follows:

```
(define_insn "call"
 [(call (match_operand:SI 0 "memory_operand" "m")
   (match_operand:SI 1 "immediate_operand" "i"))
  (clobber (reg:SI 31))
 ]
 ""
 "*
    return emit_asm_call(operands,0);
 "
)
```

The call instruction returning a value is defined using named pattern `call_value` as follows:

```
(define_insn "call_value"
 [(set (match_operand:SI 0 "register_operand" "=r")
    (call (match_operand:SI 1 "memory_operand" "m")
      (match_operand:SI 2 "immediate_operand" "i"))
  )
  (clobber (reg:SI 31))
 ]
 ""
 "*
    return emit_asm_call(operands,1);
 "
)
```

7. Machine Description for Level 4 of `spim`

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

7.1 Supported Instructions

The conditional constructs supported in this level are as given in Table 4

We can define `if-then-else` construct as a combination of compare and conditional branch instructions. In this, `cond` pattern sets conditional code, which is used by branch if condition pattern that in turn checks the conditional code set by `cond`, and goto label if condition is satisfied. There is single standard named pattern for each conditional code to represent the branch condition. Hence, instead of defining each pattern separately, we make use of code macro and implement the conditional branch as follows:

```
(define_code_macro cond_code
    [lt ltu eq ge geu gt gtu le leu ne])

(define_expand "cmpsi"
 [(set (cc0)
    (compare
      (match_operand:SI 0 "register_operand" "")
      (match_operand:SI 1 "nonmemory_operand" "")
  ))]
 ""
 {
        compare_op0=operands[0];
        compare_op1=operands[1];
        DONE;
 }
)

(define_expand "b<code>"
 [(set (pc)
    (if_then_else
       (cond_code:SI (match_dup 1)
            (match_dup 2))
       (label_ref (match_operand 0 "" ""))
       (pc)))]
 ""
 {
  operands[1]=compare_op0;
  if(immediate_operand(compare_op1,SImode))
  {
     operands[2]=force_reg(SImode,compare_op1);
  }
  else
  {
     operands[2]=compare_op1;
  }
 }
)
```

Same effect of above define patterns can be obtained by defining a single pattern `cbranchsi4`, as shown below:

```
(define_insn "cbranchsi4"
 [(set (pc)
    (if_then_else
      (match_operator:SI 0 "comparison_operator"
        [(match_operand:SI 1 "register_operand" "")
         (match_operand:SI 2 "register_operand" "")])
         (label_ref (match_operand 3 "" ""))
         (pc)))]
 ""
 "*
   return conditional_insn(GET_CODE(operands[0]),
                           operands,0);
  "
)
```

8. Conclusions and Future Work

Until now the techniques of writing GCC machine descriptions have been ad hoc. We demonstrate that it is possible to construct them systematically by identifying suitable language increments. This allows us to define the minimal machine descriptions which can then be systematically enhanced. The significance of this approach lies in the fact that it allows us to ask meaningful questions on the need of various constructs in machine descriptions. We have demonstrated this approach by generating compilers for different levels of C for the `spim` simulator. The effectiveness of this approach became evident in the workshop we had conducted: Even novices could start writing meaningful machine descriptions in a span of just 3 days! The complete descriptions corresponding to each level are available on the workshop page [1].

For the future work, we would like to extend the proposed machine descriptions to include various data types. Once a base machine description for the full language is ready, subsequent increments in the machine descriptions should be sought for improving the quality of generated code.

Acknowledgement

We would like to thank Ramana Radhakrishnan and Abhijat Vichare for their valuable suggestions from time to time.

References

[1] Workshop on GCC Internals. Centre for Formal Design and Verification of Software and Dept. of Computer Science & Engg., IIT Bombay., June 2007. `http://www.cse.iitb.ac.in/uday/gcc-workshop`.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[3] James Larus. Spim: A mips32 simulator. `http://pages.cs.wisc.edu/~larus/spim.html`.

[4] David A. Patterson and John L. Hennessy. Appendix A. In *Computer Organization and Design: The Hardware/Software Interface, Third Edition*. Morgan Kaufmann, August 2004.