Workshop on Essential Abstractions in GCC

Spim Machine Descriptions

GCC Resource Center
(www.cse.iitb.ac.in/grc)

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

2 July 2012
Outline

- Systematic construction of machine descriptions
- Retargetting GCC to spim
  - spim is mips simulator developed by James Larus
  - RISC machine
  - Assembly level simulator: No need of assembler, linkers, or libraries
- Level 0 of spim machine descriptions
- Level 1 of spim machine descriptions
Part 1

Systematic Construction of Machine Descriptions
In Search of Modularity in Retargetable Compilation

Phases of Compilation

Source Features

Target Features

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
In Search of Modularity in Retargetable Compilation
In Search of Modularity in Retargetable Compilation
In Search of Modularity in Retargetable Compilation

Phases of Compilation

Phase 1
Phase 2
Phase n

Source Features

Feature 1
Feature m

Target Features

Feature 1
Feature k

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
In Search of Modularity in Retargetable Compilation

Phases of Compilation

Level 1

Source Features (Cumulative)

Level p

Minimal Target Features (Cumulative)

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Systematic Development of Machine Descriptions

- Conditional control transfers
  - Function Calls
    - Arithmetic Expressions
      - Sequence of Simple Assignments involving integers
        - MD Level 1
        - MD Level 2
        - MD Level 3
        - MD Level 4
Systematic Development of Machine Descriptions

- Define different levels of source language
- Identify the minimal information required in the machine description to support each level
  - Successful compilation of any program, and
  - correct execution of the generated assembly program.
- Interesting observations
  - It is the increment in the source language which results in understandable increments in machine descriptions rather than the increment in the target architecture.
  - If the levels are identified properly, the increments in machine descriptions are monotonic.
Part 2

Retargeting GCC to Spim: A Recap
Retargeting GCC to Spim

- Registering spim target with GCC build process
- Making machine description files available
- Building the compiler
Registering Spim with GCC Build Process

We want to add multiple descriptions:

- Step 1. In the file $(SOURCE_D)/config.sub
  Add to the case $basic_machine
    - spim* in the part following
      # Recognize the basic CPU types without company name.
    - spim*-- in the part following
      # Recognize the basic CPU types with company name.
Registering Spim with GCC Build Process

- Step 2a. In the file $(SOURCE_D)/gcc/config.gcc

    In case ${target} used for defining cpu_type, i.e. after the line

    # Set default cpu_type, tm_file, tm_p_file and xm_file ...

    add the following case

    spim*--*--*)
        cpu_type=spim
    ;;

    This says that the machine description files are available in the directory
    $(SOURCE_D)/gcc/config/spim.
Registering Spim with GCC Build Process

- Step 2b. In the file 

\$\{(SOURCE_D)/gcc/config.gcc\}

Add the following in the case ${\text{target}}$ for

# Support site-specific machine types.

```
spim*--*--*)
  gas=no
  gnu_ld=no
  file_base="'echo ${target} | sed 's/-.*$//g'"
  tm_file="${cpu_type}/${file_base}.h"
  md_file="${cpu_type}/${file_base}.md"
  out_file="${cpu_type}/${file_base}.c"
  tm_p_file="${cpu_type}/${file_base}-protos.h"
  echo ${target}
  ;
```
Building a Cross-Compiler for Spim

- Normal cross compiler build process attempts to use the generated `cc1` to compile the emulation libraries (LIBGCC) into executables using the assembler, linker, and archiver.

- We are interested in only the `cc1` compiler.
  Add a new target in the `Makefile.in`

```plaintext
.PHONY: cc1
cc1:
    make all-gcc TARGET-gcc=cc1$(exeext)
```
Building a Cross-Compiler for Spim

- Create directories `BUILD_D` and in a tree not rooted at `SOURCE_D`.
- Change the directory to `BUILD_D` and execute the commands

  ```
  $ cd ${BUILD_D}
  $ ${SOURCE_D}/configure --target=spim
  $ make cc1
  ```

- Pray for 10 minutes :-)

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Part 3

Level 0 of Spim Machine Descriptions
Sub-levels of Level 0

Three sub-levels

- **Level 0.0**: Merely build GCC for spim simulator
  Does not compile any program (i.e. compilation aborts)

- **Level 0.1**: Compiles empty void functions

  ```c
  void fun(int p1, int p2)
  {
    int v1, v2;
  }
  ```

  ```c
  void fun()
  {
    L: goto L;
  }
  ```

- **Level 0.2**: Incorporates complete activation record structure
  Required for Level 1
## Category of Macros in Level 0

<table>
<thead>
<tr>
<th>Category</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Layout</td>
<td>complete</td>
<td>complete</td>
<td>complete</td>
</tr>
<tr>
<td>Registers</td>
<td>partial</td>
<td>partial</td>
<td>complete</td>
</tr>
<tr>
<td>Addressing Modes</td>
<td>none</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>Activation Record Conventions</td>
<td>dummy</td>
<td>dummy</td>
<td>complete</td>
</tr>
<tr>
<td>Calling Conventions</td>
<td>dummy</td>
<td>dummy</td>
<td>partial</td>
</tr>
<tr>
<td>Assembly Output Format</td>
<td>dummy</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

- Complete specification of activation record in level 0.2 is not necessary but is provided to facilitate local variables in level 1.
- Complete specification of registers in level 0.2 follows the complete specification of activation record.
Memory Layout Related Macros for Level 0

```c
#define BITS_BIG_ENDIAN 0
#define BYTES_BIG_ENDIAN 0
#define WORDS_BIG_ENDIAN 0
#define UNITS_PER_WORD 4
#define PARM_BOUNDARY 32
#define STACK_BOUNDARY 64
#define FUNCTION_BOUNDARY 32
#define BIGGEST_ALIGNMENT 64
#define STRICT_ALIGNMENT 0
#define MOVE_MAX 4
#define Pmode SImode
#define FUNCTION_MODE SImode
#define SLOW_BYTE_ACCESS 0
#define CASE VECTOR MODE SImode
```
Register Categories for Spim

- **Available to Compiler**
  - General
    - GPRs (address + data)
      - Caller-saved
      - Callee-saved
  - Floating Point
    - Fixed
      - Address
      - Data
- **Not Available to Compiler**

GCC Resource Center, IIT Bombay

Essential Abstractions in GCC
## Registers in Spim

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>constant</td>
<td>32</td>
<td>data</td>
</tr>
<tr>
<td>$at</td>
<td>NA</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>$v0</td>
<td>result</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$v1</td>
<td>result</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$a0</td>
<td>argument</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$a1</td>
<td>argument</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$a2</td>
<td>argument</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$a3</td>
<td>argument</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$t0</td>
<td>temporary</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t1</td>
<td>temporary</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$t2</td>
<td>temporary</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t3</td>
<td>temporary</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$t4</td>
<td>temporary</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t5</td>
<td>temporary</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$t6</td>
<td>temporary</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t7</td>
<td>temporary</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$s0</td>
<td>temporary</td>
<td>16,32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s1</td>
<td>temporary</td>
<td>17</td>
<td>callee</td>
</tr>
<tr>
<td>$s2</td>
<td>result</td>
<td>18,32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s3</td>
<td>result</td>
<td>19</td>
<td>callee</td>
</tr>
<tr>
<td>$s4</td>
<td>temporary</td>
<td>20,32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s5</td>
<td>temporary</td>
<td>21</td>
<td>callee</td>
</tr>
<tr>
<td>$s6</td>
<td>temporary</td>
<td>22,32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s7</td>
<td>temporary</td>
<td>23</td>
<td>callee</td>
</tr>
<tr>
<td>$t8</td>
<td>temporary</td>
<td>24,32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t9</td>
<td>temporary</td>
<td>25</td>
<td>caller</td>
</tr>
<tr>
<td>$k0</td>
<td>NA</td>
<td>26,32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$k1</td>
<td>NA</td>
<td>27</td>
<td>callee</td>
</tr>
<tr>
<td>$gp</td>
<td>global pointer</td>
<td>28,32,64</td>
<td>address</td>
</tr>
<tr>
<td>$sp</td>
<td>stack pointer</td>
<td>29</td>
<td>address</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
<td>30,32,64</td>
<td>address</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
<td>31</td>
<td>address</td>
</tr>
</tbody>
</table>
Register Information in Level 0.2

#define FIRST_PSEUDO_REGISTER 32

#define FIXED_REGISTERS
	/* not for global */
	/* register allocation */
	{
		1,1,0,0, 0,0,0,0,
		0,0,0,0, 0,0,0,0,
		0,0,0,0, 0,0,0,0,
		0,0,1,1, 1,1,1,1
	}

#define CALL_USED_REGISTERS
	/* Caller-saved registers */
	{
		1,1,1,1, 1,1,1,1,
		1,1,1,1, 1,1,1,1,
		0,0,0,0, 0,0,0,0,
		1,1,1,1, 1,1,1,1
	}

/* Register sizes */

#define HARD_REGNO_NREGS(R,M)
	((GET_MODE_SIZE (M) +
		UNITS_PER_WORD - 1) / UNITS_PER_WORD)

#define HARD_REGNO_MODE_OK(R,M)
	hard_regno_mode_ok (R, M)

#define MODES_TIEABLE_P(M1,M2)
	modes_tieable_p (M1,M2)
enum reg_class
{
  NO_REGS, CALLER_SAVED_REGS,
  CALLEE_SAVED_REGS, BASE_REGS,
  GENERAL_REGS, ALL_REGS,
  LIM_REG_CLASSES
};
#define N_REG_CLASSES 
  LIM_REG_CLASSES
#define REG_CLASS_NAMES
{
  "NO_REGS","CALLER_SAVED_REGS",
  "CALLEE_SAVED_REGS",
  "BASE_REGS", "GEN_REGS",
  "ALL_REGS"
}
#define REG_CLASS.getContent

#define REGNO_REG_CLASS_CLASS(REGNO) 
  regno_reg_class(REGNO)
#define BASE_REG_CLASS 
  BASE_REGS
#define INDEX_REG_CLASS NO_REGS
#define REG_CLASS_FROM_LETTER(c)
#define BASE_REG_CLASS
#define INDEX_REG_CLASS NO_REGS
#define REGNO_OK_FOR_BASE_P(R) 1
#define REGNO_OK_FOR_INDEX_P(R) 0
#define PREFERRED_RELOAD_CLASS(X,C)
/* Max reg required for a class */
#define CLASS_MAX_NREGS(C, M) 
  ((GET_MODE_SIZE (M) +
    UNITS_PER_WORD - 1) /
    UNITS_PER_WORD)
#define LEGITIMATE_CONSTANT_P(x)
  legitimate_constant_p(x)
function calling conventions

pass arguments on stack. return values goes in register $v0 (in level 1).

#define RETURN_POPS_ARGS(FUN, TYPE, SIZE) 0
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) 0
#define FUNCTION_ARG_REGNO_P(r) 0
  /*Data structure to record the information about args passed in
   *registers. Irrelevant in this level so a simple int will do. */
#define CUMULATIVE_ARGS int
#define INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, FNDECL, NAMED_ARGS) \
  { CUM = 0; }
#define FUNCTION_ARG_ADVANCE(cum, mode, type, named) cum++
#define FUNCTION_VALUE(valtype, func) function_value()
#define FUNCTION_VALUE_REGNO_P(REGN) ((REGN) == 2)
Activation Record Structure in Spim

Caller’s Activation Record
Activation Record Structure in Spim

Caller’s Responsibility

Caller’s Activation Record

Parameter $n$
Activation Record Structure in Spim

Caller’s Responsibility

Caller’s Activation Record

Parameter $n$

Parameter $n - 1$
Activation Record Structure in Spim

Caller’s Responsibility

Caller’s Activation Record

Parameter $n$

Parameter $n - 1$

\ldots

Argument Pointer
Activation Record Structure in Spim

- Caller’s Activation Record
  - Parameter \( n \)
  - Parameter \( n - 1 \)
  - \ldots
  - Parameter 1

Caller’s Responsibility

Argument Pointer
Activation Record Structure in Spim

Caller’s Activation Record

Parameter $n$

Parameter $n-1$

...  

Parameter 1

Return Address

Caller’s Responsibility

Callee’s Responsibility

Argument Pointer

Caller’s
Responsibility

Callee’s
Responsibility

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Activation Record Structure in Spim

- **Caller’s Activation Record**
  - Parameter $n$
  - Parameter $n - 1$
  - ... (omitted for brevity)
  - Parameter 1
  - Return Address
  - Caller’s FPR (Control Link)

**Caller’s Responsibility**
- Parameter $n$
- Parameter $n - 1$
- ... (omitted for brevity)
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)

**Callee’s Responsibility**
- Parameter $n$
- Parameter $n - 1$
- ... (omitted for brevity)
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
Activation Record Structure in Spim

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter ( n )</td>
</tr>
<tr>
<td>Parameter ( n - 1 )</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
</tbody>
</table>

Caller’s Responsibility

Callee’s Responsibility

Argument Pointer
Activation Record Structure in Spim

Caller’s Activation Record

<table>
<thead>
<tr>
<th>Parameter ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter ( n - 1 )</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>

Return Address

Caller’s FPR (Control Link)

Caller’s SPR

Callee Saved Registers

Argument Pointer

Size is known only after register allocation

Caller’s Responsibility

Callee’s Responsibility
## Activation Record Structure in Spim

### Caller’s Responsibility
- Parameter $n$
- Parameter $n - 1$
- …
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
- Local Variable 1

### Callee’s Responsibility
- Argument Pointer
- Size is known only after register allocation
- Initial Frame Pointer

---

**Essential Abstractions in GCC**

GCC Resource Center, IIT Bombay
Activation Record Structure in Spim

Caller’s Activation Record

- Parameter \( n \)
- Parameter \( n - 1 \)
- \( \ldots \)
- Parameter 1

Return Address

Caller’s FPR (Control Link)

Caller’s SPR

Callee Saved Registers

Local Variable 1

Local Variable 2

Argument Pointer

Size is known only after register allocation

Initial Frame Pointer

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Activation Record Structure in Spim

Caller’s Activation Record

- Parameter $n$
- Parameter $n-1$
- ...  
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
- Local Variable 1
- Local Variable 2
- ...
### Activation Record Structure in Spim

**Caller’s Activation Record**
- Parameter $n$
- Parameter $n-1$
- …
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
- Local Variable 1
- Local Variable 2
- …
- Local Variable $n$

**Argument Pointer**
- Size is known only after register allocation

**Initial Frame Pointer**

**Stack Pointer**
Minimizing Registers for Accessing Activation Records

Reduce four pointer registers (stack, frame, args, and hard frame) to fewer registers.

```c
#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}
}

//Recomputes new offsets, after eliminating./

#define INITIAL_ELIMINATION_OFFSET(FROM, TO, VAR)
  (VAR) = initial_elimination_offset(FROM, TO)
```
Specifying Activation Record

#define STARTING_FRAME_OFFSET starting_frame_offset ()

#define FIRST_PARM_OFFSET(FUN) 0

#define STACK_POINTER_REGNUM 29

#define FRAME_POINTER_REGNUM 1

#define HARD_FRAME_POINTER_REGNUM 30

#define ARG_POINTER_REGNUM HARD_FRAME_POINTER_REGNUM

#define FRAME_POINTER_REQUIRED 0
Level 0.0 Machine Description File
Level 0.0 Machine Description File

Empty :-)

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
## Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>
## Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

```lisp
(define_insn "jump"
  [(set (pc)
       (label_ref (match_operand 0 "" "")))
   ""
   "$j \%10"
  )
```

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
## Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

### spim0.0.c

```c
rtx gen_jump (...) {
    return 0;
}
rtx gen_indirect_jump (...) {
    return 0;
}
rtx gen_nop () {
    return 0;
}
```

### spim0.0.h

```c
#define CODE_FOR_indirect_jump 8
```

### spim0.2.md

```c
(define_insn "jump"
   [(set (pc)
       (label_ref (match_operand 0 "" ")))
    ""
   "j %10"
)
```
# Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

Only define expand. No define insn.

```lisp
(define_expand "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "))
   (match_operand:SI 1 "general_operand" "))
  ""
  {
    if(GET_CODE(operands[0])==MEM && GET_CODE(operands[1])!=REG)
    {
      if(can_create_pseudo_p())
      {
        operands[1]=force_reg(SImode,operands[1]);
      }
    }
  }
)
```

spim0.2.md
## Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

**Example Code**

```c
#define insn "IITB\_return"

(spim0.2.c)

void spim\_epilogue()
{
    emit\_insn(gen\_IITB\_return());
}

(spim0.2.md)

(define\_insn "IITB\_return"
    [(return)]
    ""
    "jr \$ra"
)

(define\_expand "epilogue"
    [(clobber (const\_int 0))]
    ""
    { spim\_epilogue();
    DONE;
    }
)
```

**Notes**

- Only return. No epilogue code.

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
## Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 0.0</th>
<th>Level 0.1</th>
<th>Level 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP direct</td>
<td>dummy</td>
<td>actual</td>
<td>actual</td>
</tr>
<tr>
<td>JUMP indirect</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>NOP</td>
<td>dummy</td>
<td>partial</td>
<td>actual</td>
</tr>
<tr>
<td>MOV</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>RETURN</td>
<td>not required</td>
<td>partial</td>
<td>partial</td>
</tr>
</tbody>
</table>

(define_insn "nop"
 [(const_int 0)]
 ""
 "nop"
 )
Part 4

Level 1 of Spim Machine Descriptions
Increments for Level 1

- Addition to the source language
  - Assignment statements involving integer constant, integer local or global variables.
  - Returning values. (No calls, though!)

- Changes in machine descriptions
  - Minor changes in macros required for level 0
    - $\texttt{zero}$ now belongs to new class Assembly output needs to change
  - Some function bodies expanded
  - New operations included in the .md file

```
$\texttt{diff -w}$ shows the changes!
```
## Operations Required in Level 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dest ← Src</strong></td>
<td>( R_i ← R_j )</td>
<td>move rj, ri</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R ← M_{\text{global}} )</td>
<td>lw r, m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R ← M_{\text{local}} )</td>
<td>lw r, c($fp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R ← C )</td>
<td>li r, c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( M ← R )</td>
<td>sw r, m</td>
<td></td>
</tr>
<tr>
<td><strong>RETURN Src</strong></td>
<td>RETURN Src</td>
<td>$v0 ← Src</td>
<td>level 0</td>
</tr>
<tr>
<td></td>
<td>j $ra</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dest ← ( Src_1 + Src_2 )</strong></td>
<td>( R_i ← R_j + R_k )</td>
<td>add ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R_i ← R_j + C )</td>
<td>addi ri, rj, c</td>
<td></td>
</tr>
</tbody>
</table>
Move Operations in `spim1.md`

- Multiple primitive variants require us to map a single operation in IR to multiple RTL patterns.
  ⇒ use `define_expand`.

- Ensure that the second operand is in a register.

```c
(define_expand "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
       (match_operand:SI 1 "general_operand" ""))
   ]"
"
  { if(GET_CODE(operands[0])==MEM &&
      GET_CODE(operands[1])!=REG &&
      (can_create_pseudo_p()) /* force conversion only */
      /* before register allocation */
      { operands[1]=force_reg(SImode,operands[1]); } }
}
```
Move Operations in spim1 Compiler for Assignment a = b

(define_expand "movsi"
   [(set (match_operand:SI 0 "nonimmediate Operand" "")
           (match_operand:SI 1 "general_operand" "")
   ]
   ""
   { if(GET_CODE(operands[0])==MEM &&
         GET_CODE(operands[1])!=REG &&
         (can_create_pseudo_p()) /* force conversion only */
         /* before register allocation */
         { operands[1]=force_reg(SImmode,operands[1]); }\n   }
)

(insn 6 5 7 3 t.c:25 (set (reg:SI 38)
                              (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
                                          (const_int -4 [0xffffffffc])) [0 b+0 S4 A32])) -1 (nil))

(insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
                                          (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
                              (reg:SI 38)) -1 (nil))
Move Operations in \texttt{spim1} Compiler for Assignment \texttt{a = b}

\begin{verbatim}
(define_expand "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
       (match_operand:SI 1 "general_operand" ""))]

  ""

  \{
    if(GET_CODE(operands[0])==MEM &&
        GET_CODE(operands[1])!=REG &&
        (can_create_pseudo_p()) /* force conversion only */
        /* before register allocation */
        \{
          operands[1]=force_reg(SImode,operands[1]);
        \}
  \})

  (insn 6 5 7 3 t.c:25 (set (reg:SI 38)
        (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
                     (const_int -4 [0xffffffffc])) [0 b+0 S4 A32])) -1 (nil))

  (insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual)
                                        (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
        (reg:SI 38)) -1 (nil))
\end{verbatim}
**Move Operations in spim1.md**

- **Load from Memory** $R \leftarrow M$

  (define_insn "*load_word"
   
   ([set (match_operand:SI 0 "register_operand" "=r")
     (match_operand:SI 1 "memory_operand" "m"))]
   ""
   "lw \t%0, %m1"
  )

- **Load Constant** $R \leftarrow C$

  (define_insn "*constant_load"
   
   ([set (match_operand:SI 0 "register_operand" "=r")
     (match_operand:SI 1 "const_int_operand" "i"))]
   ""
   "li \t%0, %c1"
  )
Move Operations in `spim1.md`

- Register Move $R_i \leftarrow R_j$

  ```
  (define_insn "*moveRegs"
    [(set (match_operand:SI 0 "register_operand" "=r")
      (match_operand:SI 1 "register_operand" "r")
    )]
    ""
    "move \t%0,%1"
  )
  ```

- Store into $M \leftarrow R$

  ```
  (define_insn "*store_word"
    [(set (match_operand:SI 0 "memory_operand" "=m")
      (match_operand:SI 1 "register_operand" "r")
    )]
    ""
    "sw \t%1, %m0"
  )
  ```
Code Generation in spim1 Compiler for Assignment \( a = b \)

- **RTL statements**

  (insn 6 5 7 3 t.c:25 (set (reg:SI 38)
   (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
     (const_int -4 [0xffffffffc])) [0 b+0 S4 A32])) -1

  (insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
   (reg:SI 38)) -1 (nil))

- **Generated Code**
Code Generation in `spim1` Compiler for Assignment `a = b`

- RTL statements

```
(insn 6 5 7 3 t.c:25 (set (reg:SI 38)
  (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
   (const_int -4 [0xffffffffc])) [0 b+0 S4 A32]))
(insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
   (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
  (reg:SI 38)) -1 (nil))
```

- Generated Code

```
lw $v0, -16($fp)
```
Code Generation in \texttt{spim1} Compiler for Assignment $a = b$

- RTL statements

\begin{verbatim}
(insn 6 5 7 3 t.c:25 (set (reg:SI 38)
   (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
      (const_int -4 [0xffffffffc])) [0 b+0 S4 A32])) -1
(insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
      (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
   (reg:SI 38)) -1 (nil))
\end{verbatim}

- Generated Code

\begin{verbatim}
lw $v0, -16($fp)
sw $v0, -20($fp)
\end{verbatim}
Using register $zero$ for constant 0

- Introduce new register class `zero_register_operand` in `spim1.h` and define `move_zero`

```
(define_insn "IITB_move_zero"
  [(set (match_operand:SI 0 "nonimmediate_operand" ";=r,m")
      (match_operand:SI 1 "zero_register_operand" ";z,z")
    ]
  ""
  "@ move \t%0,%1
  sw \t%1, %m0"
)
```

- How do we get `zero_register_operand` in an RTL?
Using register $\texttt{zero}$ for constant $0$

- Use `define_expand "movsi"` to get `zero_register_operand` in an RTL

  ```
  if(GET_CODE(operands[1])==CONST_INT && INTVAL(operands[1])==0)
  {
    emit_insn(gen_IITB_move_zero(operands[0],
                                   gen_rtx_REG(SImode,0)));
    DONE;
  }
  else /* Usual processing */
  
  DONE says do not generate the RTL template associated with "movsi"

- required template is generated by
  ```
  emit_insn(gen_IITB_move_zero(\ldots))
  ```
(define_insn "addsi3"
   [(set (match_operand:SI 0 "register_operand" "=r,r")
         (plus:SI (match_operand:SI 1 "register_operand" "r,r")
                   (match_operand:SI 2 "nonmemory_operand" "r,i")))
   ]"
"
"@
    add \t%0, %1, %2
    addi \t%0, %1, %c2"
)

- Constraints combination 1 of three operands: R, R, R
- Constraints combination 2 of three operands: R, R, C
Comparing `movsi` and `addsi3`:

- `movsi` uses `define_expand` whereas `addsi3` uses combination of operands.
- Why not use constraints for `movsi` too?
Comparing `movsi` and `addsi3`

- `movsi` uses `define_expand` whereas `addsi3` uses combination of operands.
- Why not use constraints for `movsi` too?
- Combination of operands is used during pattern matching and not during expansion.
  - We will need to support memory as both source and destination.
Comparing `movsi` and `addsi3`

- `movsi` uses `define_expand` whereas `addsi3` uses combination of operands
- Why not use constraints for `movsi` too?
- Combination of operands is used during pattern matching and not during expansion
  - We will need to support memory as both source and destination
  - Will also allow memory to memory move in RTL
    We will not know until assembly emission which one is a load instruction and which one is a store instruction
Part 5

Conclusions
Conclusions

• Incremental construction of machine description files is very instructive

• Increments in machine descriptions is governed by increments in source language

• Machine characteristics need to be specified in C macros and C functions
  ▶ Does not seem amenable to incremental construction
  ▶ Seems difficult to a novice

• Specifying instructions seems simpler and more systematic
  ▶ Is amenable to incremental construction
  ▶ The concept of minimal machine descriptions is very useful

• \texttt{define_insn} and \texttt{define_expand} are the main constructs used on machine descriptions
Part 6

Constructs Supported in Level 2
## Arithmetic Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 - \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j - R_k )</td>
<td>sub ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow -\text{Src} )</td>
<td>( R_i \leftarrow -R_j )</td>
<td>neg ri, rj</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 / \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j / R_k )</td>
<td>div rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 % \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j % R_k )</td>
<td>rem ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 \times \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j \times R_k )</td>
<td>mul ri, rj, rk</td>
<td></td>
</tr>
</tbody>
</table>
# Arithmetic Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Dest} \leftarrow \text{Src}_1 - \text{Src}_2$</td>
<td>$R_i \leftarrow R_j - R_k$</td>
<td>sub $ri$, $rj$, $rk$</td>
<td></td>
</tr>
<tr>
<td>$\text{Dest} \leftarrow -\text{Src}$</td>
<td>$R_i \leftarrow -R_j$</td>
<td>neg $ri$, $rj$</td>
<td></td>
</tr>
<tr>
<td>$\text{Dest} \leftarrow \text{Src}_1 / \text{Src}_2$</td>
<td>$R_i \leftarrow R_j / R_k$</td>
<td>div $rj$, $rk$</td>
<td>level 2</td>
</tr>
<tr>
<td>$\text{Dest} \leftarrow \text{Src}_1 % \text{Src}_2$</td>
<td>$R_i \leftarrow R_j % R_k$</td>
<td>rem $ri$, $rj$, $rk$</td>
<td></td>
</tr>
<tr>
<td>$\text{Dest} \leftarrow \text{Src}_1 \times \text{Src}_2$</td>
<td>$R_i \leftarrow R_j \times R_k$</td>
<td>mul $ri$, $rj$, $rk$</td>
<td></td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
## Bitwise Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Dest \leftarrow \ Src_1 \ll \ Src_2$</td>
<td>$R_i \leftarrow R_j \ll R_k$</td>
<td>sllv ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>$\phantom{Dest} \leftarrow \ Src_1 \gg \ Src_2$</td>
<td>$R_i \leftarrow R_j \gg R_k$</td>
<td>srav ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>$Dest \leftarrow \ Src_1 &amp; \ Src_2$</td>
<td>$R_i \leftarrow R_j &amp; R_k$</td>
<td>and ri, rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td>$\phantom{Dest} \leftarrow \ Src_1 \mid \ Src_2$</td>
<td>$R_i \leftarrow R_j \mid R_k$</td>
<td>or ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>$Dest \leftarrow \ Src_1 ^ \ Src_2$</td>
<td>$R_i \leftarrow R_j ^ R_k$</td>
<td>xor ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>$Dest \leftarrow \sim \ Src$</td>
<td>$R_i \leftarrow \sim R_j$</td>
<td>not ri, rj</td>
<td></td>
</tr>
</tbody>
</table>
Divide Operation in spim2.md using define_insn

- For division, the spim architecture imposes use of multiple asm instructions for single operation.

```scheme
(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"
)
```
Divide Operation in \texttt{spim2.md} using \texttt{define_insn}

- For division, the spim architecture imposes use of multiple asm instructions for single operation.

- Two ASM instructions are emitted using single RTL pattern

\begin{verbatim}
(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"

\end{verbatim}
Advantages/Disadvantages of using `define_insn`

- Very simple to add the pattern
- Primitive target feature represented as single insn pattern in `.md`
- Unnecessary atomic grouping of instructions
- May hamper optimizations in general, and instruction scheduling, in particular
Divide Operation in spim2.md using define_expand

- The RTL pattern can be expanded into two different RTLs.

```scheme
(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;
  }
)
```
Divide Operation in `spim2.md` using `define_expand`

- Divide pattern equivalent to div instruction in architecture.

```scheme
(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"
)
```
Divide Operation in spim2.md using define_expand

- Divide pattern equivalent to div instruction in architecture.

```lisp
(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"
)
```
Divide Operation in `spim2.md` using `define_expand`

- Moving contents of special purpose register LO to/from general purpose register

```
(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"
)
```

```
(define_insn "IITB_move_to_lo"
  [(set (match_operand:SI 0 "LO_register_operand" ";=q")
        (match_operand:SI 1 "register_operand" ";r"))]
  ""
  "mtlo \t%1"
)
```
Divide Operation in spim2.md using define_expand

- Divide pattern equivalent to div instruction in architecture.

```lisp
(define_insn "modsi3"
  [(parallel[(set (match_operand:SI 0 "register_operand" ":=r")
               (mod:SI (match_operand:SI 1 "register_operand" "r")
                       (match_operand:SI 2 "register_operand" "r"))
               (clobber (reg:SI 26))
               (clobber (reg:SI 27))])]
  ""
  "rem \t%0, %1, %2"
)"
```
Divide Operation in `spim2.md` using `define_expand`

- Divide pattern equivalent to div instruction in architecture.

```lisp
(define_insn "modsi3"
  [(parallel
    [(set (match_operand:SI 0 "register_operand" "=r")
      (mod:SI (match_operand:SI 1 "register_operand" "r")
        (match_operand:SI 2 "register_operand" "r")))
    (clobber (reg:SI 26))
    (clobber (reg:SI 27))])

""
"rem \t%0, %1, %2"
)
```
Advantages/Disadvantages of Using `define_expanded_for` for Division

- Two instructions are separated out at GIMPLE to RTL conversion phase
- Both instructions can undergo all RTL optimizations independently
- C interface is needed in md
- Compilation becomes slower and requires more space
Divide Operation in spim2.md using define_split

(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); "
)

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Divide Operation in spim2.md using define_split

(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); "
)
Divide Operation in \texttt{spim2.md} using \texttt{define\_split}

\begin{verbatim}
(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); "
\end{verbatim}
Part 7

Constructs Supported in Level 3
# Operations Required in Level 3

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Dest \leftarrow \text{fun}(P_1, \ldots, P_n)$</td>
<td></td>
<td>$\text{lw } r_i, \ [SP+c1]$ $\text{sw } r_i, \ [SP]$ $\text{call } L_{\text{fun}}, n$ $\text{sw } r_i, \ [SP-n*4]$ $\text{jal } L$</td>
<td>$\text{Level 1}$</td>
</tr>
<tr>
<td>$\text{fun}(P_1, P_2, \ldots, P_n)$</td>
<td></td>
<td>$\text{lw } r_i, \ [SP+c1]$ $\text{sw } r_i, \ [SP]$ $\text{call } L_{\text{fun}}, n$ $\text{sw } r_i, \ [SP-n*4]$ $\text{jal } L$</td>
<td>$\text{New}$</td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Call Operation in spim3.md

(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);
  ""
)
(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);
"
)
Activation Record Generation during Call

- Operations performed by caller
- Operations performed by callee
Activation Record Generation during Call

- Operations performed by caller

- Operations performed by callee

Caller’s Activation Record
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter ( n )</td>
</tr>
<tr>
<td>Parameter ( n - 1 )</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.

- Operations performed by callee

```
<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter n</td>
</tr>
<tr>
<td>Parameter n - 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>
```
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.
- Operations performed by callee
  - Push Return address stored by caller on stack.
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.

---

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
</tbody>
</table>
**Activation Record Generation during Call**

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.

---

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter n</td>
</tr>
<tr>
<td>Parameter n – 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.
- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
<tr>
<td>Local Variable 2</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
<tr>
<td>Local Variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

### Caller’s Activation Record
- Parameter $n$
- Parameter $n - 1$
- ... 
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
  - Local Variable 1
  - Local Variable 2
  - ...
  - Local Variable $n$
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.
  - Start callee body execution.

---

**Caller’s Activation Record**

<table>
<thead>
<tr>
<th>Parameter $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
<tr>
<td>Local Variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local Variable $n$</td>
</tr>
</tbody>
</table>
(define_expand "prologue"
  [(clobber (const_int 0))]
  ""
  {
    spim_prologue();
    DONE;
  })
Prologue in spim3.md

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

(set (mem:SI (reg:SI $sp))
    (reg:SI 31 $ra))

(set (mem:SI (plus:SI (reg:SI $sp)
    (const_int -4 )))
    (reg:SI $sp))

(set (mem:SI (plus:SI (reg:SI $sp)
    (const_int -8 )))
    (reg:SI $fp))

(set (reg:SI $fp)
    (reg:SI $sp))

(set (reg:SI $sp)
    (plus:SI (reg:SI $fp)
        (const_int -36)))
Epilogue in spim3.md

```scheme
(define_expand "epilogue"
  [(clobber (const_int 0))]
  ""

  spim_epilogue();
  DONE;
)
```

```scheme
(set (reg:SI $sp)
  (reg:SI $fp))

(set (reg:SI $fp)
  (mem:SI (plus:SI (reg:SI $sp)
    (const_int -8 ))))

(set (reg:SI $ra)
  (mem:SI (reg:SI $sp)))

(parallel [
  (return)
  (use (reg:SI $ra))]
```
Part 8

Constructs Supported in Level 4
<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Src_1 &lt; Src_2$ ? goto L : PC</td>
<td>$CC \leftarrow R_i &lt; R_j$&lt;br&gt;CC &lt; 0 ? goto L : PC</td>
<td>blt $r_i, r_j, L$</td>
<td></td>
</tr>
<tr>
<td>$Src_1 &gt; Src_2$ ? goto L : PC</td>
<td>$CC \leftarrow R_i &gt; R_j$&lt;br&gt;CC &gt; 0 ? goto L : PC</td>
<td>bgt $r_i, r_j, L$</td>
<td></td>
</tr>
<tr>
<td>$Src_1 \leq Src_2$ ? goto L : PC</td>
<td>$CC \leftarrow R_i \leq R_j$&lt;br&gt;CC ≤ 0 ? goto L : PC</td>
<td>ble $r_i, r_j, L$</td>
<td></td>
</tr>
<tr>
<td>$Src_1 \geq Src_2$ ? goto L : PC</td>
<td>$CC \leftarrow R_i \geq R_j$&lt;br&gt;CC ≥ 0 ? goto L : PC</td>
<td>bge $r_i, r_j, L$</td>
<td></td>
</tr>
</tbody>
</table>
# Operations Required in Level 4

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
</table>
| $Src_1 \equiv Src_2$ ? goto L : PC | $CC \leftarrow R_i \equiv R_j$
  $CC \equiv 0$ ? goto L : PC | beq $r_i, r_j, L$ |          |
| $Src_1 \neq Src_2$ ? goto L : PC | $CC \leftarrow R_i \neq R_j$
  $CC \neq 0$ ? goto L : PC | bne $r_i, r_j, L$ |          |
(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);
  "
  )
Support for Branch pattern in spim4.c

```c
char *
conditional_insn (enum rtx_code code, rtx operands[])
{
    switch (code)
    {
    case EQ: return "beq %1, %2, %l3";
    case NE: return "bne %1, %2, %l3";
    case GE: return "bge %1, %2, %l3";
    case GT: return "bgt %1, %2, %l3";
    case LT: return "blt %1, %2, %l3";
    case LE: return "ble %1, %2, %l3";
    case GEU: return "bgeu %1, %2, %l3";
    case GTU: return "bgtu %1, %2, %l3";
    case LTU: return "bltu %1, %2, %l3";
    case LEU: return "bleu %1, %2, %l3";
    default: /* Error. Issue ICE */
    }
}
```
Alternative for Branch: Conditional compare in \texttt{spim4.md}

\begin{verbatim}
(define_code_iterator 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;
    }
)
\end{verbatim}
(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;
  operands[2]=compare_op1;
  if(immediate_operand(operands[2],SImode))
  {
  }
}
)
Alternative for Branch: Branch pattern in spim4.md

(define_insn "*insn_b<code>"
 [(set (pc)
   (if_then_else
    (cond_code:SI
     (match_operand:SI 1 "register_operand" "r")
     (match_operand:SI 2 "register_operand" "r"))
     (label_ref (match_operand 0 "" ""))
     (pc)))]

"
"*

    return conditional_insn(<CODE>,operands);
"
)