Spim Machine Descriptions

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

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

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

Phase 1

Phase n

Target Features

Source Features

Phases of Compilation

Phase n
In Search of Modularity in Retargetable Compilation

Source Features

Phases of Compilation

Target Features

Phase 1

Phase n

Feature m

Feature k

Feature 1

Essential Abstractions in GCC

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

Phases of Compilation

Level 1

Level p

Source Features (Cumulative)

Minimal Target Features (Cumulative)

Notes

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

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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.

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_DIR}/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.

• Step 2b. In the file $(SOURCE_D)/gcc/config.gcc

Add the following in the case ${target} for

# Support site-specific machine types.

spim*-*-*
    gas=no
gnu_ld=no
    file_base="\'echo ${target} | sed s/-.*$/\'"'
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

```
.PHONY: cc1
ccl:
    make all-gcc TARGET-gcc=cc1$(exeext)
```

- Create directories `${BUILD_DIR}` and in a tree not rooted at `${SOURCE_DIR}`.
- Change the directory to `${BUILD_DIR}` and execute the commands

```
$ cd ${BUILD_DIR}
$ ${SOURCE_DIR}/configure --target=spim<n>
$ make cc1
```
- Pray for 10 minutes :-)
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 BIGGEST_ALIGNMENT 64
#define BYTES_BIG_ENDIAN 0
#define STRICT_ALIGNMENT 0
#define WORDS_BIG_ENDIAN 0
#define MOVE_MAX 4
#define UNITS_PER_WORD 4
#define PARM_BOUNDARY 32
#define FUNCTION_MODE SImode
#define FUNCTION_BOUNDARY 32
#define SLOW_BYTE_ACCESS 0
#define CASE_VECTOR_MODE SImode
```
Register Categories for Spim

All Registers
- Available to Compiler
- Not Available to Compiler

General Floating Point
- GPRs (address + data)
- Address
- Data
- Caller-saved
- Callee-saved

Notes

Registers in Spim

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>00</td>
<td>32</td>
<td>constant</td>
<td>data</td>
</tr>
<tr>
<td>$at</td>
<td>01</td>
<td>32</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>$v0</td>
<td>02</td>
<td>32,64</td>
<td>result</td>
<td>caller</td>
</tr>
<tr>
<td>$v1</td>
<td>03</td>
<td>32</td>
<td>result</td>
<td>caller</td>
</tr>
<tr>
<td>$a0</td>
<td>04</td>
<td>32,64</td>
<td>argument</td>
<td>caller</td>
</tr>
<tr>
<td>$a1</td>
<td>05</td>
<td>32</td>
<td>argument</td>
<td>caller</td>
</tr>
<tr>
<td>$a2</td>
<td>06</td>
<td>32,64</td>
<td>argument</td>
<td>caller</td>
</tr>
<tr>
<td>$a3</td>
<td>07</td>
<td>32</td>
<td>argument</td>
<td>caller</td>
</tr>
<tr>
<td>$t0</td>
<td>08</td>
<td>32,64</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t1</td>
<td>09</td>
<td>32</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t2</td>
<td>10</td>
<td>32,64</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t3</td>
<td>11</td>
<td>32</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t4</td>
<td>12</td>
<td>32,64</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t5</td>
<td>13</td>
<td>32</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t6</td>
<td>14</td>
<td>32,64</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$t7</td>
<td>15</td>
<td>32</td>
<td>temporary</td>
<td>caller</td>
</tr>
<tr>
<td>$a0</td>
<td>16</td>
<td>32,64</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a1</td>
<td>17</td>
<td>32</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a2</td>
<td>18</td>
<td>32,64</td>
<td>result</td>
<td>callee</td>
</tr>
<tr>
<td>$a3</td>
<td>19</td>
<td>32</td>
<td>result</td>
<td>callee</td>
</tr>
<tr>
<td>$a4</td>
<td>20</td>
<td>32,64</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a5</td>
<td>21</td>
<td>32</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a6</td>
<td>22</td>
<td>32,64</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a7</td>
<td>23</td>
<td>32</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a8</td>
<td>24</td>
<td>32,64</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$a9</td>
<td>25</td>
<td>32</td>
<td>temporary</td>
<td>callee</td>
</tr>
<tr>
<td>$t0</td>
<td>26</td>
<td>32,64</td>
<td>temporary</td>
<td>NA</td>
</tr>
<tr>
<td>$t1</td>
<td>27</td>
<td>32</td>
<td>temporary</td>
<td>NA</td>
</tr>
<tr>
<td>$t2</td>
<td>28</td>
<td>32,64</td>
<td>temporary</td>
<td>global pointer</td>
</tr>
<tr>
<td>$t3</td>
<td>29</td>
<td>32</td>
<td>temporary</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$t4</td>
<td>30</td>
<td>32,64</td>
<td>temporary</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$t5</td>
<td>31</td>
<td>32</td>
<td>temporary</td>
<td>return address</td>
</tr>
<tr>
<td>$t6</td>
<td>32</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t7</td>
<td>33</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t8</td>
<td>34</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t9</td>
<td>35</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t10</td>
<td>36</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t11</td>
<td>37</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t12</td>
<td>38</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t13</td>
<td>39</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t14</td>
<td>40</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t15</td>
<td>41</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t16</td>
<td>42</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t17</td>
<td>43</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t18</td>
<td>44</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t19</td>
<td>45</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t20</td>
<td>46</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t21</td>
<td>47</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t22</td>
<td>48</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t23</td>
<td>49</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t24</td>
<td>50</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t25</td>
<td>51</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t26</td>
<td>52</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t27</td>
<td>53</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t28</td>
<td>54</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t29</td>
<td>55</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t30</td>
<td>56</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
<tr>
<td>$t31</td>
<td>57</td>
<td>32</td>
<td>temporary</td>
<td>address</td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Register Information in Level 0.2

#define FIRST_PSEUDO_REGISTER 32
#define FIXED_REGISTERS
// not for global */
/* register allocation */
{ 1,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0,
  0,0,1,1, 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)

Register Classes in Level 0.2

enum reg_class
{ NO_REGS, CALLER_SAVED_REGS,
  CALLEE_SAVED_REGS, BASE_REGS,
  GENERAL_REGS, ALL_REGS,
  LIM_REGS_CLASSES \ 
};
#define N_REG_CLASSES \ 
LIM_REGS_CLASSES
#define REG_CLASS_NAMES \ 
{ "NO_REGS","CALLER_SAVED_REGS","\n  CALLEE_SAVED_REGS","\n  BASE_REGS","GEN_REGS","\n  ALL_REGS" \ 
}
#define REGCLASS_CONTENTS \ 
/* Register numbers */
{ 0x00000000,0xffff0000,
  0x0f000000,0x00000000,
  0xc0000000,0xff000000 }
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

<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>
<tr>
<td>Local Variable n</td>
</tr>
</tbody>
</table>

Notes

Size is known only after register allocation

Initial Frame Pointer
Stack Pointer

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Minimizing Registers for Accessing Activation Records

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

#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

#### Empty :-)

### Operations in Level 0

<table>
<thead>
<tr>
<th>Operations</th>
<th>Level 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;
}
```

**spim0.2.md**

```markdown
(define_insn "jump"
  [(set (pc)
        (label ref (match_operand 0 "" ""))]
  "\"j %l0\""
)
```
### 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.

```c
#include <stdint.h>

#define expand "movsi"
#define insn "IITB_return"

void spim_epilogue()
{
    emit_insn(gen_IITB_return);
}
```
### 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></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"
)
```
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
    - $zero now belongs to new class Assembly output needs to change
  - Some function bodies expanded
  - New operations included in the .md file
    - `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>Dest ← Src</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>RETURN Src</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>Dest ← Src₁ + Src₂</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`

```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]); }
}
```

(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 [0xffffffff]) [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 [0xffffffff]) [0 a+0 S4 A32])
   (reg:SI 38)) -1 (nil))
```
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"
  )
  ```

- Register Move \( R_i \leftarrow R_j \)

  ```
  (define_insn "*move_regs"
    [(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 (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))
  ```

- **Generated Code**

  ```
  lw $v0, -16($fp)
  sw $v0, -20($fp)
  ```

**Notes**

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 $zero for constant 0

- Use `define` to 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(...))`

Notes

Supporting Addition in Level 1

```
(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?
- 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
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
- `define_insn` and `define_expand` are the main constructs used on machine descriptions
### 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>Dest ← Src₁ − Src₂</td>
<td>Rₗ ← Rᵢ − Rₖ</td>
<td>sub ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>Dest ← − Src</td>
<td>Rₗ ← −Rᵢ</td>
<td>neg ri, rj</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src₁ / Src₂</td>
<td>Rₗ ← Rᵢ / Rₖ</td>
<td>div ri, rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td>Dest ← − Src₁ % Src₂</td>
<td>Rₗ ← Rᵢ % Rₖ</td>
<td>rem ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src₁ * Src₂</td>
<td>Rₗ ← Rᵢ * Rₖ</td>
<td>mul ri, rj, rk</td>
<td></td>
</tr>
</tbody>
</table>

### 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 ← Src₁ ≪ Src₂</td>
<td>Rₗ ← Rᵢ ≪ C₅</td>
<td>sllv ri, rj, c</td>
<td>level 2</td>
</tr>
<tr>
<td>Dest ← Src₁ ≫ Src₂</td>
<td>Rₗ ← Rᵢ ≫ C₅</td>
<td>sra ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src₁ &amp; Src₂</td>
<td>Rₗ ← Rᵢ &amp; Rₖ</td>
<td>and ri, rj, rₖ</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src₁</td>
<td>Rₗ ← Rᵢ</td>
<td>or ri, rj, rₖ</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src₁ ^ Src₂</td>
<td>Rₗ ← Rᵢ ^ Rₖ</td>
<td>xor ri, rj, rₖ</td>
<td></td>
</tr>
<tr>
<td>Dest ← ~ Src</td>
<td>Rₗ ← ~Rᵢ</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.
- Two ASM instructions are emitted using single RTL 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-mflo\t%0"
)
```

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.

```lisp
(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_RTX_DIVIDE(SImode, 26),
  operands[1], operands[2]));
emit_insn(gen_RTX_MOVE_FROM_REG(operands[0],
  gen_RTX_REG(SImode, 26)));
DONE;
```

Notes

- 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"))
    ])
  ]"

"div t%1, %2"
)```
Divide Operation in `spim2.md` using `define_expanded`

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

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

(define_insn "IITE_move_to_lo"
 [(set (match_operand:SI 0 "LO_register_operand" "+q")
   (match_operand:SI 1 "register_operand" "+r"))]
 "mtlo \t%1"
 )
```

- Divide pattern equivalent to div instruction in architecture.

```
(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, expand 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

Notes
## Operations Required in Level 3

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
</table>
| $Dest \leftarrow \text{fun}(P_1, \ldots , P_n)$| $\text{call } L_{\text{fun}}, n$ | $\text{lw } r_i, \ [\text{SP}+c1]$  
$\text{sw } r_i, \ [\text{SP}]$  
:  
$\text{lw } r_i, \ [\text{SP}+c2]$  
$\text{sw } r_i, \ [\text{SP}-n \times 4]$  
$\text{jal } L$  
$Dest \leftarrow $ \$v0$ | Level 1  
New  
level 1 |
| fun($P_1, P_2, \ldots , P_n$) | $\text{call } L_{\text{fun}}, n$ | $\text{lw } r_i, \ [\text{SP}+c1]$  
$\text{sw } r_i, \ [\text{SP}]$  
:  
$\text{lw } r_i, \ [\text{SP}+c2]$  
$\text{sw } r_i, \ [\text{SP}-n \times 4]$  
$\text{jal } L$ | Level 1  
New |
(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
  - 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.

---

Prologue in `spim3.md`

```prolog
(define expand "prologue"
 [(clobber (const int 0))
  
  
  
  (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

(define \texttt{expand} "epilogue"
  [(clobber (const int 0))
   ""
   spim\textunderscore epilogue();
   DONE;
   )

(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))])

Essential Abstractions in GCC

Notes

Essential Abstractions in GCC

Part 8

Constructs Supported in Level 4

Notes
## Operations Required in Level 4

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Src_1 &lt; Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← 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><code>Src_1 &gt; Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← 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><code>Src_1 ≤ Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← R_i ≤ R_j&lt;br&gt;CC ≤ 0 ? goto L : PC</td>
<td>ble r_i, r_j, L</td>
<td></td>
</tr>
<tr>
<td><code>Src_1 ≥ Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← R_i ≥ R_j&lt;br&gt;CC ≥ 0 ? goto L : PC</td>
<td>bge r_i, r_j, L</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Src_1 == Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← R_i == R_j&lt;br&gt;CC == 0 ? goto L : PC</td>
<td>beq r_i, r_j, L</td>
<td></td>
</tr>
<tr>
<td><code>Src_1 ≠ Src_2</code></td>
<td>goto L : PC&lt;br&gt;CC ← R_i ≠ R_j&lt;br&gt;CC ≠ 0 ? goto L : PC</td>
<td>bne r_i, r_j, L</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Branch Instruction in spim4.md

(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);"
  )]

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 LE:return "ble %1, %2, %l3";
    case LT:return "blt %1, %2, %l3";
    case GEU:return "bgeu %1, %2, %l3";
    case GTU:return "bgtu %1, %2, %l3";
    case LEU:return "bleu %1, %2, %l3";
    case LTU:return "bltu %1, %2, %l3";
    default: /* Error. Issue ICE */
  }
}
(define_code_iterator cond_code
    [lt ltu eq geu gt 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;
        operands[2]=compare_op1;
        if(immediate_operand(operands[2],SImode))
        {
        }
    })
(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);"
  ""
)