Workshop on Essential Abstractions in GCC

Spim Machine Descriptions: Levels 0 and 1

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

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

2 July 2011
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
## In Search of Modularity in Retargetable Compilation

### Phase 1

<table>
<thead>
<tr>
<th>Source Features</th>
<th>Phases of Compilation</th>
<th>Target Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Phase 1</td>
<td>Phase n</td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

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

Phases of Compilation

Source Features

Phase 1

Phases of Compilation

Target Features

Phase n

Feature 1

Feature m
In Search of Modularity in Retargetable Compilation
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 `$\texttt{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} is 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` `${target}` for

```bash
# 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

```
.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 :-)
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;
}
```

- Level 0.2: Incorporates complete activation record structure
  Required for Level 1

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

#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

- All Registers
  - Available to Compiler
    - General
      - GPRs (address + data)
        - Caller-saved
        - Callee-saved
    - Floating Point
      - Fixed
        - Address
        - Data
  - Not Available to Compiler
# Registers in Spim

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Size</th>
<th>Use</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>$s0</td>
<td>temporary</td>
<td>32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s1</td>
<td>temporary</td>
<td>32</td>
<td>callee</td>
</tr>
<tr>
<td>$s2</td>
<td>result</td>
<td>32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s3</td>
<td>result</td>
<td>32</td>
<td>callee</td>
</tr>
<tr>
<td>$s4</td>
<td>temporary</td>
<td>32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s5</td>
<td>temporary</td>
<td>32</td>
<td>callee</td>
</tr>
<tr>
<td>$s6</td>
<td>temporary</td>
<td>32,64</td>
<td>callee</td>
</tr>
<tr>
<td>$s7</td>
<td>temporary</td>
<td>32</td>
<td>callee</td>
</tr>
<tr>
<td>$t8</td>
<td>temporary</td>
<td>32,64</td>
<td>caller</td>
</tr>
<tr>
<td>$t9</td>
<td>temporary</td>
<td>32</td>
<td>caller</td>
</tr>
<tr>
<td>$k0</td>
<td>NA</td>
<td>32,64</td>
<td></td>
</tr>
<tr>
<td>$k1</td>
<td>NA</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td>global pointer</td>
<td>32,64</td>
<td>address</td>
</tr>
<tr>
<td>$sp</td>
<td>stack pointer</td>
<td>32</td>
<td>address</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
<td>32,64</td>
<td>address</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
<td>32</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) 

$zero,$at 
$k0,$k1 
$gp,$sp,$fp,$ra 
$s0$ to $s7$ 

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Register Classes in Level 0.2

```c
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_CONTENTS
/* Register numbers */
{
    0x00000000,0xffffffff,
    0x00ff0000,0xf0000000,
    0xcfffff3,0xffffffff
}
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Function Calling Conventions

Pass arguments on stack. Return values goes in register $v0$ (in level 1).

```c
#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

<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>

Argument Pointer
Activation Record Structure in Spim

- Caller’s Activation Record
  - Parameter $n$
  - Parameter $n-1$
  - ... (ellipses)
  - 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
Activation Record Structure in Spim

- Caller’s Responsibility:
  - Parameter $n$
  - Parameter $n-1$
  - ... 
  - Parameter 1

- Callee’s Responsibility:
  - Return Address
  - Caller’s FPR (Control Link)

Argument Pointer
Activation Record Structure in Spim

- **Caller's Activation Record**
  - Parameter $n$
  - Parameter $n-1$
  - ... (continues)
  - Parameter 1
  - Return Address
  - Caller's FPR (Control Link)
  - Caller's SPR

- **Callee's Responsibility**
  - Argument Pointer

- **Caller's Responsibility**
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

Caller’s Responsibility

Argument Pointer

Callee’s Responsibility

Size is known only after register allocation
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

**Caller’s Responsibility**
- Argument Pointer
- Call Stack Pointer (initial frame pointer)

**Callee’s Responsibility**
- Size is known only after register allocation
- Essential Abstractions in GCC

[Image: Diagram showing activation record structure with attributes and responsibilities of caller and callee]
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

**Caller’s Responsibility**
- Parameter
- Call

**Callee’s Responsibility**
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
- Local Variable
- Initial Frame Pointer
- Argument Pointer

Size is known only after register allocation
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
- ...

Caller’s Responsibility

Callee’s Responsibility

Size is known only after register allocation

Argument Pointer

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 |
| ... |
| Local Variable n |

Argument Pointer

Size is known only after register allocation

Initial Frame Pointer

Stack Pointer

Caller’s Responsibility

Callee’s Responsibility
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
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"
  )
```

---

**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>

### 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

```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.

```lisp
(define_expand "movsi"
  [(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>

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

```c
void spim_epilogue()
{
    emitInsn(gen_IITB_return());
}
```

```c
spim0.2.c
```

(spim0.2.md)

---

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>partial</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
    - `$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>( \text{Dest} \leftarrow \text{Src} )</td>
<td>( R_i \leftarrow R_j )</td>
<td>move ( r_j, r_i )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R \leftarrow M_{\text{global}} )</td>
<td>( \text{lw } r, m )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R \leftarrow M_{\text{local}} )</td>
<td>( \text{lw } r, c($fp$) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R \leftarrow C )</td>
<td>( \text{li } r, c )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( M \leftarrow R )</td>
<td>( \text{sw } r, m )</td>
<td></td>
</tr>
<tr>
<td>RETURN ( \text{Src} )</td>
<td>RETURN ( \text{Src} )</td>
<td>( $v0 \leftarrow \text{Src} )</td>
<td>level 0</td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 + \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j + R_k )</td>
<td>add ( r_i, r_j, r_k )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R_i \leftarrow R_j + C )</td>
<td>addi ( r_i, r_j, c )</td>
<td></td>
</tr>
</tbody>
</table>

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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`

```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 &&
  (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 [0xfffffffc])) [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 `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(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 [0xfffffffff])) [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 [0xfffffffff8])) [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"
Move Operations in spim1.md

- **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

\[
\begin{align*}
\text{(insn 6 5 7 3 t.c:25 (set (reg:SI 38) }
\text{ (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars) }
\text{ (const_int -4 [0xffffffffc])) [0 b+0 S4 A32])) -1} \\
\text{(insn 7 6 8 3 t.c:25 (set (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars) }
\text{ (const_int -8 [0xfffffffff8])) [0 a+0 S4 A32])}
\text{ (reg:SI 38)) -1 (nil))}
\end{align*}
\]

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

- RTL statements

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

- Generated Code

\[
lw \$v0, -16(\$fp)
\]
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 (reg:SI 38))
    (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)
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 $\text{zero}$ for constant 0

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

```c
if(GET_CODE(operands[1])==CONST_INT && INTVAL(operands[1])==0)
{
    emit_insn(gen_IITB_move_zero(operands[0],
                                   gen_rtx_REG(SI_mode,0)));
    DONE;
}
else /* Usual processing */
```

- DONE says do not generate the RTL template associated with "movsi"

- required template is generated by

```c
emit_insn(gen_IITB_move_zero(operands[0],
                            gen_rtx_REG(SI_mode,0)));
```
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_expanded` whereas `addsi3` uses combination of operands
- Why not use constraints for `movsi` too?
Comparing movsi and addsi3

- movsi uses define_expanded 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.
- `define_insn` and `define_expand` are the main constructs used on machine descriptions.