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

Notes
In Search of Modularity in Retargetable Compilation

**Target Features**

**Source Features**

**Phases of Compilation**

- Phase 1
- Phase n

**Feature 1**

**Feature m**

**Feature k**

Essential Abstractions in GCC

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

Level p

Source Features (Cumulative)

Level 1

Minimal Target Features (Cumulative)

Phases of Compilation

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
• 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

- 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)/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 \( $(\text{SOURCE\_D})/gcc/config.gcc \)

In case \( \{$\text{target}\}\) used for defining \texttt{cpu\_type}, i.e. after the line

\[
\text{# Set default cpu\_type, tm\_file, tm\_p\_file and xm\_file} ... 
\]

add the following case

\[
\text{spim\-\*-\*-\*)} \\n\text{ cpu\_type=spim} \\n\text{ ;;} 
\]

This says that the machine description files are available in the directory \( $(\text{SOURCE\_D})/gcc/config/spim \).

- Step 2b. In the file \( $(\text{SOURCE\_D})/gcc/config.gcc \)

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

\[
\text{# Support site-specific machine types.} 
\]

\[
\text{spim\-\*-\*-\*)} \\n\text{ gas=no} \\n\text{ gnu\_ld=no} \\n\text{ file\_base="'echo \{$\text{target}\} | sed 's/-.*//'"} \\n\text{ tm\_file="'echo \{$\text{cpu\_type}\}/$\{file\_base\}.h"} \\n\text{ md\_file="'echo \{$\text{cpu\_type}\}/$\{file\_base\}.md"} \\n\text{ out\_file="'echo \{$\text{cpu\_type}\}/$\{file\_base\}.c"} \\n\text{ tm\_p\_file="'echo \{$\text{cpu\_type}\}/$\{file\_base\}-protos.h"} \\n\text{ echo \{$\text{target}\} } \\n\text{ ;;} 
\]
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)
```

- 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;
  }
  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 BYTE_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 MODE SImode
#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)

Fixed

Address

Data

Caller-saved

Callee-saved

Registers in Spim

$zero 00 32 constant data $a0 16 32,64 temporary callee
$at 01 32 NA $a1 17 32 temporary callee
$v0 02 32,64 result caller $a2 18 32,64 result callee
$v1 03 32 result caller $a3 19 32 result callee
$a0 04 32,64 argument caller $a4 20 32,64 temporary callee
$a1 05 32 argument caller $a5 21 32 temporary callee
$a2 06 32,64 argument caller $a6 22 32,64 temporary callee
$a3 07 32 argument caller $a7 23 32 temporary callee
$t0 08 32,64 temporary callee $t8 24 32,64 temporary caller
$t1 09 32 temporary callee $t9 25 32 temporary caller
$t2 10 32,64 temporary callee $x0 26 32,64 NA
$t3 11 32 temporary callee $x1 27 32 NA
$t4 12 32,64 temporary callee $gp 28 32,64 global pointer address
$t5 13 32 temporary callee $sp 29 32 stack pointer address
$t6 14 32,64 temporary callee $fp 30 32,64 frame pointer address
$t7 15 32 temporary callee $ra 31 32 return address address
Register Information in Level 0.2

## Notes

$\text{\textcopyright Essential Abstractions in GCC}$

GCC Resource Center, IIT Bombay
Function Calling Conventions

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

```c
#define RETURNARGS_FUN(TYPE, SIZE) 0
#define FUNCTIONARGS_FUN(TYPE, NAMED) 0
#define FUNCTIONARGS_FUN_REGNO_P 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 FUNCTIONARGS_ADVANCE(cum, mode, type, named) cum++
#define FUNCTIONARGS_VALUE(valtype, func) function_value()
#define FUNCTIONARGS_REGNO_P(REGN) ((REGN) == 2)
```

Activation Record Structure in Spim

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

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

Notes
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./

```c
#define INITIAL_ELIMINATION_OFFSET(FROM, TO, VAR)
(VAR) = initial_elimination_offset((FROM, TO))
```

Specifying Activation Record

```c
#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
```
Empty :-)

## 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.h

```c
#define CODE För间接跳转 8
```

### spim0.2.md

```yaml
(define_insn "jump"
  [(set (pc)
    (label ref (match operand 0 """")))
   **"
   "j %10"
)
```

### Notes

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

<table>
<thead>
<tr>
<th>Operation</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
(define_expansion "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`

```
void spim_epilogue()
{
    emit_insn(gen_IITB_return());
}
```

Only return. No epilogue code.
### 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>dummy</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 "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
```
diff -w shows the changes!

<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 $r_j, r_i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r ← M_{global}$</td>
<td>lw $r, m$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r ← M_{local}$</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 $r_i$</td>
<td>$v0 ← r_i$</td>
<td>level 0</td>
</tr>
<tr>
<td></td>
<td>j $ra$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest ← $src_1 + src_2$</td>
<td>$r_i ← r_j + r_k$</td>
<td>add $r_i, r_j, r_k$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r_i ← r_j + C$</td>
<td>addi $r_i, r_j, c$</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]); } }
}
```

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

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))
```

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
### 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**
  
  ```plaintext
  (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-
  (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])
  (reg:SI 38)) -1 (nil)))
  ```

- **Generated Code**

  ```plaintext
  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`
  
  ```plaintext
  (define_insn "IITB_move_zero"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,m")
  (match_operand:SI 1 "zero_register_operand" "z,z")
  )
  ""
  "@0
  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 \texttt{define\_expand} "movsi" to get \texttt{zeroregister\_operand} in an RTL

\begin{verbatim}
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 */
\end{verbatim}

- \texttt{DONE} says do not generate the RTL template associated with "movsi"
- required template is generated by

\begin{verbatim}
emit_insn(gen_IITB_move_zero(...))
\end{verbatim}

Notes

Supporting Addition in Level 1

\begin{verbatim}
(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")))
    ""
   "S"
   "add \t%0, %1, %2
addi \t%0, %1, %c2"
)
\end{verbatim}

- 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.