Compiler Construction

MIPS overview

MIPS

A simple RISC mircroprocessor

- Nintendo 64 game console
- PlayStation
- Cisco Router
- ...

MIPS Registers and Use Convention

Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0–v1	2–3	Expression evaluation and results of a function
a0–a3	4–7	Function argument 1–4
t0–t7	8–15	Temporary (not preserved across call)
s0–s7	16–23	Saved temporary (preserved across call)
t8–t9	24–25	Temporary (not preserved across call)
k0–k1	26–27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Typical RISC Instructions

The following slides are based on.

- The assembler translates pseudo-instructions (marked with † below).
- In all instructions below, `Src2` can be
  - a register
  - an immediate value (a 16 bit integer).
- The immediate forms are included for reference.
- The assembler translates the general form (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is constant.

Table of Contents

Arithmetic: Addition/Subtraction

`add` `Rdest, Rsrc1, Src2` — *Addition (with overflow)*
`addi` `Rdest, Rsrc1, Imm` — *Addition Immediate (with overflow)*
`addu` `Rdest, Rsrc1, Src2` — *Addition (without overflow)*
`addiu` `Rdest, Rsrc1, Imm` — *Addition Immediate (without overflow)*

Put the sum of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`sub` `Rdest, Rsrc1, Src2` — *Subtract (with overflow)*
`subu` `Rdest, Rsrc1, Src2` — *Subtract (without overflow)*

Put the difference of the integers from `Rsrc1` and `Src2` into `Rdest`.

Arithmetic: Division

If an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`div Rsrc1, Rsrc2` *Divide (signed)*
`divu Rsrc1, Rsrc2` *Divide (unsigned)*

Divide the contents of the two registers. Leave the quotient in register `lo` and the remainder in register `hi`.

`div Rdest, Rsrc1, Src2` *Divide (signed, with overflow)* †
`divu Rdest, Rsrc1, Src2` *Divide (unsigned, without overflow)* †

Put the quotient of the integers from `Rsrc1` and `Src2` into `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †

Likewise for the the remainder of the division.

Arithmetic: Multiplication

`mul` `Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo` `Rdest, Rsrc1, Src2` *Multiply (with overflow)* †
`mulou` `Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †

Put the product of the integers from `Rsrc1` and `Src2` into `Rdest`.

`mult` `Rsrc1, Rsrc2` *Multiply*
`multu` `Rsrc1, Rsrc2` *Unsigned Multiply*

Multiply the contents of the two registers. Leave the low-order word of the product in register `lo` and the high-word in register `hi`.

Arithmetic Instructions

`abs Rdest, Rsrc` — *Absolute Value* †

Put the absolute value of the integer from `Rsrc` in `Rdest`.

`neg Rdest, Rsrc` — *Negate Value (with overflow)* †

`negu Rdest, Rsrc` — *Negate Value (without overflow)* †

Put the negative of the integer from `Rsrc` into `Rdest`.

Table of Contents

Logical Instructions

`and` `Rdest, Rsrc1, Src2` *AND*

`andi` `Rdest, Rsrc1, Imm` *AND Immediate*

Put the logical AND of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`not` `Rdest, Rsrc` *NOT* †

Put the bitwise logical negation of the integer from `Rsrc` into `Rdest`.

Logical Instructions

`nor Rdest, Rsrc1, Src2` *NOR*

Put the logical NOR of the integers from `Rsrc1` and `Src2` into `Rdest`.

`or Rdest, Rsrc1, Src2` *OR*

`ori Rdest, Rsrc1, Imm` *OR Immediate*

Put the logical OR of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`xor Rdest, Rsrc1, Src2` *XOR*

`xori Rdest, Rsrc1, Imm` *XOR Immediate*

Put the logical XOR of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

Logical Instructions

`rol` `Rdest, Rsrc1, Src2` — *Rotate Left* †
`ror` `Rdest, Rsrc1, Src2` — *Rotate Right* †

Rotate the contents of `Rsrc1` left (right) by the distance indicated by `Src2` and put the result in `Rdest`.

`sll` `Rdest, Rsrc1, Src2` — *Shift Left Logical*
`sllv` `Rdest, Rsrc1, Rsrc2` — *Shift Left Logical Variable*
`sra` `Rdest, Rsrc1, Src2` — *Shift Right Arithmetic*
`srav` `Rdest, Rsrc1, Rsrc2` — *Shift Right Arithmetic Variable*
`srl` `Rdest, Rsrc1, Src2` — *Shift Right Logical*
`srlv` `Rdest, Rsrc1, Rsrc2` — *Shift Right Logical Variable*

Shift the contents of `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in `Rdest`.

Table of Contents

Comparison Instructions

`seq` `Rdest, Rsrc1, Src2` *Set Equal* †

Set `Rdest` to 1 if `Rsrc1` equals `Src2`, otherwise to 0.

`sne` `Rdest, Rsrc1, Src2` *Set Not Equal* †

Set `Rdest` to 1 if `Rsrc1` is not equal to `Src2`, otherwise to 0.

Comparison Instructions

`sge` `Rdest, Rsrc1, Src2` — *Set Greater Than Equal* †
`sgeu` `Rdest, Rsrc1, Src2` — *Set Greater Than Equal Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $\geq$ `Src2`, otherwise to 0.

`sgt` `Rdest, Rsrc1, Src2` — *Set Greater Than* †
`sgtu` `Rdest, Rsrc1, Src2` — *Set Greater Than Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $>$ `Src2`, otherwise to 0.

`sle` `Rdest, Rsrc1, Src2` — *Set Less Than Equal* †
`sleu` `Rdest, Rsrc1, Src2` — *Set Less Than Equal Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $\leq$ `Src2`, otherwise to 0.

`slt` `Rdest, Rsrc1, Src2` — *Set Less Than*
`slti` `Rdest, Rsrc1, Imm` — *Set Less Than Immediate*
`sltu` `Rdest, Rsrc1, Src2` — *Set Less Than Unsigned*
`sltiu` `Rdest, Rsrc1, Imm` — *Set Less Than Unsigned Immediate*
Set `Rdest` to 1 if `Rsrc1` $<$ `Src2` (or `Imm`), otherwise to 0.

Branch and Jump Instructions

Branch instructions use a signed 16-bit offset field: jump from -2^{15} to $+2^{15}-1$) *instructions* (not bytes). The *jump* instruction contains a 26 bit address field.

`b label` *Branch instruction* †

Unconditionally branch to *label*.

`j label` *Jump*

Unconditionally jump to *label*.

`jal label` *Jump and Link*
`jalr Rsrc` *Jump and Link Register*

Unconditionally jump to *label* or whose address is in `Rsrc`. Save the address of the next instruction in register 31.

`jr Rsrc` *Jump Register*

Unconditionally jump to the instruction whose address is in register `Rsrc`.

Branch and Jump Instructions

`bczt label` *Branch Coprocessor z True*

`bczf label` *Branch Coprocessor z False*

Conditionally branch to *label* if coprocessor *z*'s condition flag is true (false).

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` $*$ `Src2`.

`beq Rsrc1, Src2, label`	*Branch on Equal*
`bne Rsrc1, Src2, label`	*Branch on Not Equal*
`beqz Rsrc, label`	*Branch on Equal Zero* †
`bnez Rsrc, label`	*Branch on Not Equal Zero* †

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` $*$ `Src2`.

`bge` `Rsrc1, Src2, label` — *Branch on Greater Than Equal* †
`bgeu` `Rsrc1, Src2, label` — *Branch on GTE Unsigned* †
`bgez` `Rsrc, label` — *Branch on Greater Than Equal Zero*
`bgezal` `Rsrc, label` — *Branch on Greater Than Equal Zero And Link*

Conditionally branch to *label* if the contents of `Rsrc` are greater than or equal to 0. Save the address of the next instruction in register 31.

`bgt` `Rsrc1, Src2, label` — *Branch on Greater Than* †
`bgtu` `Rsrc1, Src2, label` — *Branch on Greater Than Unsigned* †
`bgtz` `Rsrc, label` — *Branch on Greater Than Zero*

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` are $*$ to `Src2`.

`ble Rsrc1, Src2, label` — *Branch on Less Than Equal* †

`bleu Rsrc1, Src2, label` — *Branch on LTE Unsigned* †

`blez Rsrc, label` — *Branch on Less Than Equal Zero*

`bgezal Rsrc, label` — *Branch on Greater Than Equal Zero And Link*

`bltzal Rsrc, label` — *Branch on Less Than And Link*

Conditionally branch to *label* if the contents of `Rsrc` are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

`blt Rsrc1, Src2, label` — *Branch on Less Than* †

`bltu Rsrc1, Src2, label` — *Branch on Less Than Unsigned* †

`bltz Rsrc, label` — *Branch on Less Than Zero*

Exception and Trap Instructions

`rfe` *Return From Exception*

Restore the Status register.

`syscall` *System Call*

Register `$v0` contains the number of the system call provided by SPIM.

`break` n *Break*

Cause exception *n*. Exception 1 is reserved for the debugger.

`nop` *No operation*

Do nothing.

Table of Contents

Constant-Manipulating Instructions

`li Rdest, imm` *Load Immediate* †

Move the immediate `imm` into `Rdest`.

`lui Rdest, imm` *Load Upper Immediate*

Load the lower halfword of the immediate `imm` into the upper halfword of `Rdest`. The lower bits of the register are set to 0.

Load: Byte & Halfword

`lb Rdest, address` *Load Byte*

`lbu Rdest, address` *Load Unsigned Byte*

Load the byte at *address* into `Rdest`. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`lh Rdest, address` *Load Halfword*

`lhu Rdest, address` *Load Unsigned Halfword*

Load the 16-bit quantity (halfword) at *address* into register `Rdest`. The halfword is sign-extended by the `lh`, but not the **lhu**, instruction

Load: Word

`lw Rdest, address` *Load Word*

Load the 32-bit quantity (word) at *address* into `Rdest`.

`lwcz Rdest, address` *Load Word Coprocessor*

Load the word at *address* into `Rdest` of coprocessor *z* (0–3).

`lwl Rdest, address` *Load Word Left*

`lwr Rdest, address` *Load Word Right*

Load the left (right) bytes from the word at the possibly-unaligned *address* into `Rdest`.

`ulh Rdest, address` *Unaligned Load Halfword* †

`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †

Load the 16-bit quantity (halfword) at the possibly-unaligned *address* into `Rdest`. The halfword is sign-extended by the `ulh`, but not the **ulhu**, instruction

`ulw Rdest, address` *Unaligned Load Word* †

Load the 32-bit quantity (word) at the possibly-unaligned *address* into `Rdest`.

Load Instructions

`la Rdest, address` *Load Address* †

Load computed *address*, not the contents of the location, into `Rdest`.

`ld Rdest, address` *Load Double-Word* †

Load the 64-bit quantity at *address* into `Rdest` and `Rdest + 1`.

Store: Byte & Halfword

`sb Rsrc, address` *Store Byte*

Store the low byte from `Rsrc` at *address*.

`sh Rsrc, address` *Store Halfword*

Store the low halfword from `Rsrc` at *address*.

Store: Word

`sw Rsrc, address` *Store Word*

Store the word from `Rsrc` at *address*.

`swcz Rsrc, address` *Store Word Coprocessor*

Store the word from `Rsrc` of coprocessor *z* at *address*.

`swl Rsrc, address` *Store Word Left*
`swr Rsrc, address` *Store Word Right*

Store the left (right) bytes from `Rsrc` at the possibly-unaligned *address*.

`ush Rsrc, address` *Unaligned Store Halfword* †

Store the low halfword from `Rsrc` at the possibly-unaligned *address*.

`usw Rsrc, address` *Unaligned Store Word* †

Store the word from `Rsrc` at the possibly-unaligned *address*.

Store: Double Word

`sd Rsrc, address` *Store Double-Word* †

Store the 64-bit quantity in `Rsrc` and `Rsrc + 1` at *address*.

Data Movement Instructions

`move` `Rdest, Rsrc` *Move* †

Move the contents of `Rsrc` to `Rdest`.

The multiply and divide unit produces its result in two additional registers, `hi` and `lo` (e.g., `mul Rdest, Rsrc1, Src2`).

`mfhi` `Rdest` *Move From hi*

`mflo` `Rdest` *Move From lo*

Move the contents of the `hi` (`lo`) register to `Rdest`.

`mthi` `Rdest` *Move To hi*

`mtlo` `Rdest` *Move To lo*

Move the contents `Rdest` to the `hi` (`lo`) register.

Data Movement Instructions

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

`mfcz Rdest, CPsrc` *Move From Coprocessor z*

Move the contents of coprocessor *z*'s register `CPsrc` to CPU `Rdest`.

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* †

Move the contents of floating point registers `FRsrc1` and `FRsrc1 + 1` to CPU registers `Rdest` and `Rdest + 1`.

`mtcz Rsrc, CPdest` *Move To Coprocessor z*

Move the contents of CPU `Rsrc` to coprocessor *z*'s register `CPdest`.

Table of Contents

MIPS Floating Point Instructions

- Floating point coprocessor 1 operates on single (32-bit) and double precision (64-bit) FP numbers.
- 32 32-bit registers `$f0`–`$f31`.
- Two FP registers to hold doubles.
- FP operations only use even-numbered registers
  including instructions that operate on single floats.
- Values are moved one word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudo-instructions.
- The flag set by FP comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions.

Floating Point: Arithmetics

Compute the $*$ of the floating float doubles (singles) in `FRsrc1` and `FRsrc2` and put it in `FRdest`.

`add.d FRdest, FRsrc1, FRsrc2`	*Floating Point Addition Double*
`add.s FRdest, FRsrc1, FRsrc2`	*Floating Point Addition Single*
`div.d FRdest, FRsrc1, FRsrc2`	*Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2`	*Floating Point Divide Single*
`mul.d FRdest, FRsrc1, FRsrc2`	*Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2`	*Floating Point Multiply Single*
`sub.d FRdest, FRsrc1, FRsrc2`	*Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2`	*Floating Point Subtract Single*
`abs.d FRdest, FRsrc`	*Floating Point Absolute Value Double*
`abs.s FRdest, FRsrc`	*Floating Point Absolute Value Single*
`neg.d FRdest, FRsrc`	*Negate Double*
`neg.s FRdest, FRsrc`	*Negate Single*

Floating Point: Comparison

Compare the floating point double in `FRsrc1` against the one in `FRsrc2` and set the floating point condition flag true if they are $*$.

`c.eq.d FRsrc1, FRsrc2`	*Compare Equal Double*
`c.eq.s FRsrc1, FRsrc2`	*Compare Equal Single*
`c.le.d FRsrc1, FRsrc2`	*Compare Less Than Equal Double*
`c.le.s FRsrc1, FRsrc2`	*Compare Less Than Equal Single*
`c.lt.d FRsrc1, FRsrc2`	*Compare Less Than Double*
`c.lt.s FRsrc1, FRsrc2`	*Compare Less Than Single*

Floating Point: Conversions

Convert between (i) single, (ii) double precision floating point number or (iii) integer in `FRsrc` to `FRdest`.

`cvt.d.s FRdest, FRsrc`	*Convert Single to Double*
`cvt.d.w FRdest, FRsrc`	*Convert Integer to Double*
`cvt.s.d FRdest, FRsrc`	*Convert Double to Single*
`cvt.s.w FRdest, FRsrc`	*Convert Integer to Single*
`cvt.w.d FRdest, FRsrc`	*Convert Double to Integer*
`cvt.w.s FRdest, FRsrc`	*Convert Single to Integer*

Floating Point: Moves

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †

Load the floating float double (single) at `address` into register `FRdest`.

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*

Move the floating float double (single) from `FRsrc` to `FRdest`.

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †

Store the floating float double (single) in `FRdest` at `address`.

A Sample: fact

```
/* Define a recursive function.  */
let
  /* Calculate n! */
  function fact (n : int) : int =
    if n = 0
      then 1
      else n * fact (n - 1)
in
  print_int (fact (10));
  print ("\n")
end
```

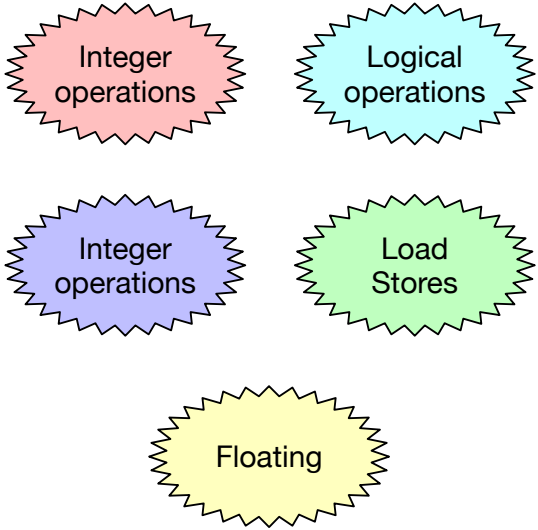
```
# Routine: fact
l0:     sw      $fp, -8 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 16
        sw      $ra, -12 ($fp)
        sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
l5:     lw      $t0, -4 ($fp)
        beq     $t0, 0, l1
l2:     lw      $a0, ($fp)
        lw      $t0, -4 ($fp)
        sub     $a1, $t0, 1
        jal     l0
        lw      $t0, -4 ($fp)
        mul     $t0, $t0, $v0
l3:     move    $v0, $t0
        j       l6
l1:     li      $t0, 1
        j       l3
l6:     lw      $ra, -12 ($fp)
        move    $sp, $fp
        lw      $fp, -8 ($fp)
```

```
.data
l4:
        .word 1
        .asciiz "\n"
.text
# Routine: Main
t_main: sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l7:     move    $a0, $fp
        li      $a1, 10
        jal     l0
        move    $a0, $v0
        jal     print_int
        la      $a0, l4
        jal     print
l8:     lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Nolimips (formerly Mipsy)

- Another MIPS emulator
- Interactive loop
- Unlimited number of $x42 registers!

Summary


Integer operations
Logical operations
Integer operations
Load Stores
Floating