MIPS, Register allocation, and the MARS simulator

Jost Berthold
(based on Danish version by Torben Mogensen, 2010)

November 2012

Overview

This document describes back-end tools and given SML modules for assignments in the lecture "Compilers" in the 2012 run: the MIPS instruction (sub-)set, a register allocation module, and the simulator MARS to execute MIPS assembly code.

Contents

Overview **1**

1 MIPS Instruction Set Module **1**

2 The Register Allocator **4**
2.1 Interface and Functionality 4
2.2 Using the register allocator in function translation 5
2.2.1 Function Calls 5
2.2.2 Prolog and Epilog of Functions 6

3 MARS, a MIPS simulator **7**

1 MIPS Instruction Set Module

The module `Mips.sml` defines a data structure to describe a subset of the MIPS[1] instruction set, as the type `mips`:

MIPS.sml

```
datatype mips
 = LABEL of string
 | EQU of string*string
```

[1] MIPS stands for *Microprocessor without Interlocked Pipeline Stages*, a microprocessor in RISC (reduced instruction set) architecture. The MIPS architecture and instruction set are described in extenso in Appendix A of Patterson and Hennessy [1] (available online).

```
| GLOBL of string
| TEXT of string
| DATA of string
| SPACE of string
| ASCII of string
| ASCIIZ of string
| ALIGN of string
| COMMENT of string
| LA of string*string
| LUI of string*string
| ADD of string*string*string
| ADDI of string*string*string
| SUB of string*string*string
| MUL of string*string*string (* low bytes of result in dest., no overflow *)
| DIV of string*string*string (* quotient in dest., no remainder *)
| AND of string*string*string
| ANDI of string*string*string
| OR of string*string*string
| ORI of string*string*string
| XOR of string*string*string
| XORI of string*string*string
| SLL of string*string*string
| SRA of string*string*string
| SLT of string*string*string
| SLTI of string*string*string
| BEQ of string*string*string
| BNE of string*string*string
| BGEZ of string*string
| J of string
| JAL of string * string list (* label + arg. reg.s *)
| JR of string * string list (* jump reg. + result reg.s *)
| LW of string*string*string (* lw rd,i(rs), encoded as LW (rd,rs,i) *)
| SW of string*string*string (* sw rd,i(rs) encoded as SW (rd,rs,i) *)
| LB of string*string*string (* lb rd,i(rs) encoded as LB (rd,rs,i) *)
| SB of string*string*string (* sb rd,i(rs) encoded as SB (rd,rs,i) *)
| NOP
| SYSCALL
```

We explain in brief the meaning of each instruction.

`LABEL` defines a *label* (a target for jump instructions).

`EQU` serves to define constants in assembly. For instance, `EQU ("x","-16")` stands for the assembly instruction `x = -16`. The two fields are strings because one can use hexadecimal format and other (previously defined) constants and labels as values.

`GLOBL, TEXT, DATA, SPACE, ASCII, ASCIIZ and ALIGN` stand for the assembly directives `.globl`, `.text`, `.data`, `.space`, `.ascii`, `.asciiz` and `.align`. They delimit different segments of an assembly program, and are used by the assembler which creates machine code from it.

`COMMENT` contains a comment.

`LA` is the instruction to load an address into a register. The register is given as either a number between 0 and 31 (as a string) or the name

of a variable that is held in a numerical register. *Symbolic register names* such as `v0` or `a0` *cannot be used*, because everything that is not a number will be considered as an allocated variable by the register allocator later.

`LUI` loads a value ('I' for "immediate") into the upper 16 bit of a register (given as a number between 0 and 31 or a name, as described above).

`ADD ...SLTI` are common binary arithmetic, logical and shift operations (using registers or, with 'I', values) as MIPS instructions. Detailed descriptions of these instructions can be looked up in other sources (for instance Appendix A of Patterson and Hennessy [1]). Register arguments are given as described above, numeric constants are either (signed) decimal numbers, hexadecimal numbers starting with `0x`, or symbolic constants defined by **`EQU`** before.

`BEQ...J` are conditional and unconditional jump instructions which take a label argument (and one or two register arguments in case of branch instructions). Execution continues at the instruction designated by the label argument. In case of branch instructions, the register values are checked for the indicated condition before, and continues with the next instruction if the condition is false (for instance, *branch-equal* only jumps if the values in the two argument registers are equal).

`JAL`, the MIPS *jump-and-link* instruction `jal`, stores the address of the instruction that follows `jal` into the return address register `ra` and execution continues at the instruction designated by the argument label.

In order to do register allocation on MIPS code, `JAL` should, besides the jump target label, carry a list of variables and registers that are live at the destination (typically argument registers and global variables). The `JAL` instruction in register-allocated code does not use this list and it can thus be left empty.

`JR`, the MIPS *jump-register* instruction **`jr`**, continues execution at the address stored in the register that is given as an argument. Besides the jump target register, `JR` carries a list of live variables and registers at the destination (typically return value registers and global variables). The `JR` instruction in register-allocated code does not use this list and it can thus be left empty.

`LW`, **`SW`**, **`LB`** and **`SB`** are the MIPS *load* and *store* instructions for words and bytes (*load/store-word/byte*, `lw`, `sw`, `lb`, `sb`). Please note: in contrast to usual MIPS assembly format, the *offset* from the source address stands *last* in the parameter list. For instance, `LW ("2","28","16")` encodes the instruction `lw $2,16($28)` (load word from address in register 28 plus 16 byte offset into register 2).

`NOP` is the empty instruction `nop`, *no operation*.

`SYSCALL` is the *system call* instruction `syscall`. Register $2 contains the system operation to execute, parameters are given in registers $4 and $5, a possible return value will appear in $2.

System calls interface to operating system services and will not be explained here. Interested readers are referred to appendix A.9 in [1].

Aside from these instructions, `Mips.sml` also provides two pseudo-instructions which are part of MIPS, but implemented by an *immediate-or* instruction:

`MOVE (x,y)` (MIPS `move`) is translated to `ORI (x,y,"0")`, a bitwise `OR` of 0 with the source register `y`, storing in `x`. The register allocator can remove a `move` instruction if target and source (`x` and `y`) are allocated to the same register.

`LI (x,k)` , the MIPS *load-immediate* instruction `li`, is translated to `ORI (x,"0",k)`, a bitwise `OR` of 0 with the constant `k`.

A MIPS assembly program is represented simply as a list of these instructions, i.e. `type MipsProgram = Mips.mips list`.
Finally, the `Mips` module defines functions to "pretty-print" MIPS instructions, and a function `pp_mips_list` that prints an entire program. The resulting string can then be written to a file and later read in by a MIPS assembler or simulator.

2 The Register Allocator

2.1 Interface and Functionality

Module `RegAlloc.sml` provides a register allocator, which translates symbolic register names into register numbers. The allocator works on the body of a single function at a time. Its type is

from *RegAlloc.sig*

```
val registerAlloc :
  Mips.mips list -> string list -> int -> int -> int
  -> Mips.mips list * string list * int
```

The arguments of `registerAlloc` are a list of MIPS instructions, a list of registers that are live *after* executing the instructions, and three register numbers: the lowest usable register (typically 2 for MIPS), the largest *caller-saves* register, and the maximum register overall (MIPS has 32 registers, but some of them serve special purposes. We use 25). Registers in the range from the first to the second number are *caller-saves* registers, while those

with numbers between the second and the third number are *callee-saves*. The standard conventions for MIPS are 2, 15 and 25.
`registerAlloc` returns a triple. The first, and most important, component is a modified list of instructions with symbolic register names replaced by numerical registers[2], and `move` instructions removed whenever source and target register were allocated to the same numerical register.
The other return values are: a list of variables that are live at entry of the code, and the maximum register number allocated in the instruction sequence. The live variables are only for debugging, the maximum register number is needed for generating code that protects *callee-saves* registers.
The register allocator does not *spill* registers, as this would make it considerably more complicated. If there are not enough registers available, an exception `not_colourable` is raised instead.

2.2 Using the register allocator in function translation

We use a simplified version of the MIPS calling conventions:

- Registers \$2 ... \$15 are *caller-saves* and \$16 ... \$25 are *callee-saves*.
- Parameters are passed in register \$2 ... \$15 (as necessary) and the function result is passed in register \$2. We assume that there are never more than 14 parameters, ensuring that all parameters can be passed in registers.
- Register \$29 is the stack pointer. The stack grows towards zero (in negative direction), and the stack pointer (`SP`) points to the top-most element (with the lowest address). No *frame pointer* is used.

2.2.1 Function Calls

The register allocator implements *caller-saves* for jumps, and will not allocate variables to *caller-saves* registers if these are live after the function call. Thus, a function call such as `t = f(x,y,z)` can be translated to the following instructions:

```
[Mips.MOVE ("2","x"),
 Mips.MOVE ("3","y"),
 Mips.MOVE ("4","z"),
 Mips.JAL ("f",["2","3","4"]),
 Mips.MOVE ("t","2")]
```

Please note that the list of registers used as parameters in the call is an extra argument to the `jal` instruction (as explained above). When more than 14 parameters have to be used, remaining parameters should be passed on the stack.

[2]For better readability and debugging, the instruction sequence is also commented with the original symbolic instruction.

2.2.2 Prolog and Epilog of Functions

Once we have translated the *body* of a function definition `f(x,y,z) = e` (the part that computes the value of e from `x`, `y` and `z`) to a list of MIPS instructions, this code sequence must be wrapped by additional instructions to receive the arguments in registers, and to return the result in register $2. Assuming the code sequence `body` for e stores the value in a symbolic register `result`, the entire body of `f` is the following:

```
val body_symb = [Mips.MOVE ("x","2"),
                 Mips.MOVE ("y","3"),
                 Mips.MOVE ("z","4")]
                @ body @
                [Mips.MOVE ("2","result")]
```

This symbolic code is now run through the register allocator:

```
val (f_body, _, maxReg) =
       RegAlloc.registerAlloc body_symb ["2"] 2 15 25
```

The arguments indicate that:

- register $2 is the only live register at the exit of `body_symb`. It will contain the return value.
- The minimum register is 2, the highest caller-saves register is 15, and the maximum register available is 25.

The returned `f_body` contains the body of funtion `f` with numerical registers. Many of the `move` instructions we added in the previous step are likely to be removed by the register allocator, as it will try to use the registers which are already filled with the right values.

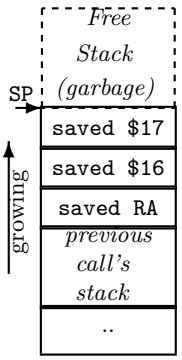
Return value `maxReg` indicates the highest register number used in `f_body`. If this number is higher than 15 (maximum caller-saves), the body uses *callee-saves registers* which need to be *saved to the stack* before executing the body code and *restored afterwards*. We also save the return address from register `RA` (register $31) to the stack, as the body might call other functions and overwrite the address otherwise.

For instance, if `maxReg` is 17, registers `RA` ($31), $16 and $17 need to be saved to the stack and restored afterwards, leading to the following code:

```
val prologue = [Mips.LABEL "f",
                Mips.SW ("31","29","-4"),  # return address (RA)
                Mips.SW ("16","29","-8"),  # 16, callee-saves 1
                Mips.SW ("17","29","-12"), # 17, callee-saves 2
                Mips.ADDI ("29","29","-12")] # adjust SP

val epilogue = [Mips.ADDI ("29","29","12"),  # adjust SP
                Mips.LW ("17","29","-12"), # 17, callee-saves 2
                Mips.LW ("16","29","-8"),  # 16, callee-saves 1
                Mips.LW ("31","29","-4"),  # return address (RA)
                Mips.JR ("31",[])]

val f_all = prologue @ f_body @ epilogue
```



The stack pointer is (by convention) always stored in register $29, and the stack grows towards smaller addresses. The picture on the right shows the stack immediately after entering the function body. The body instructions might call other functions (or `f`, recursively) and make the stack grow further. The return address register (sometimes called *link register*, used by instruction JAL) is $31. Note that this is register-allocated code, so the list of live registers for the JR instruction can be empty.

The previous example is for a specific function with 3 arguments that uses two callee-saves registers (`maxReg`). The code generator of a compiler needs to generate prolog and epilog code for a varying number of arguments and register use. Figure 1 shows a sketch of how such code might look like. If a function uses more parameters than available registers, the helper function `movePars` must be extended to pass some arguments on the stack instead.

translating a function

```
val SP = "29"
val RA = "31"

fun translateFunction(name, parameters, bodyExp, ftable) =
  let val body = translateExp(bodyExp, emptyTable, ftable, "2")
      val body_symb = movePars parameters 2 @ body
      val (f_body, _, maxReg) =
            RegAlloc.registerAlloc body_symb ["2"] 2 15 25
      val (saveCode, restoreCode, frameSize) = saveRestore maxReg
      val prologue = [Mips.LABEL name, Mips.SW (RA,SP,"-4")]
                     @ saveCode @
                     [Mips.ADDI (SP,SP,Int.toString (~frameSize))]
      val epilogue = [Mips.ADDI (SP,SP,Int.toString frameSize)]
                     @ restorecode @
                     [Mips.LW (RA,SP,"-4"), Mips.JR (RA,[])]
  in prologue @ f_body @ epilogue
  end

and movePars [] _ = []
  | movePars (par::pars) reg =
      Mips.MOVE (par,Int.toString reg) :: movePars pars (reg+1)

and saveRestore reg =
  if reg<16 then ([],[],4)
  else let val (save,restore,size) = saveRestore (reg-1)
           val newSize = size+4
           val args = (Int.toString reg, SP, int.toString (~newSize))
       in (Mips.SW args :: save, Mips.LW args :: restore, newSize)
       end
```

Figure 1: Code for translating functions

3 MARS, a MIPS simulator

MARS is an acronym for *MIPS Assembler and Runtime Simulator*, and supports a large subset of MIPS instructions. MARS is compatible with the SPIM simulator described in Appendix A.9 of Patterson/Hennessy [1]. MARS is written in Java and available under the URL http://courses.

missouristate.edu/kenvollmar/mars/. Its current version is 4.2 (appeared in August 2011). An introductory manual can be found at http://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html. MARS can be used to directly assemble and execute a MIPS assembly program, by the following command on the command line:

```
me@home$ java -jar Mars_4_2.jar program.asm
```

The executed program will use standard input and output to interact with the user.
MARS also comes with a rich GUI where the machine state during program execution can be observed and the assembly code edited. The GUI starts if MARS is started without an additional argument, like so:

```
me@home$ java -jar Mars_4_2.jar
```

The following things should be noted when using MARS:

- Registers $1, $26 and $27 are reserved for expanding pseudo-instructions (such as `la`) and for the operating system, respectively.
- The stack pointer ($29) is initialised at program start and does not need re-initialisation.
- The stack pointer points to the top element, so new stack elements need to be written to the address *4 bytes below* the current SP (offset 4 on 32-bit architectures).
- Global variables, tables, and other data created at runtime, should be placed in a data segment of the assembly, and grow upward in memory from the start of the data segment.

References

[1] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface.* Morgan Kaufmann, 1998. Appendix A is freely available at http://www.cs.wisc.edu/~larus/HP_AppA.pdf.