Tópicos Avançados de Compilação
Assignment 2

Departamento de Informática
Universidade de Évora

2017/2018

1 Object

The assignment consists in implementing a *code generator* which, given the intermediate representation (IR) of a TACL program, selects the appropriate MIPS instructions and produces the corresponding assembly code.

The main focus of the assignment is *instruction selection*, for which it is enough to use the instruction-by-instruction approach.

Another (minor) aspect of the assignment is *register allocation*, for which you may use either the simple register allocation algorithm described in the *Instruction Selection* slides, or an approach which guarantees that registers are correctly allocated even when temporaries are used more than once or have live ranges which cross basic block boundaries.

The following sections specify the format of the IR, some constraints on the MIPS code and the assembly format, and discuss some practical issues. TACL is described in *The TACL 1.0 Language* text and in the notes for a *Crash Course em Compiladores* (CCC), which can be found in the course moodle page.

2 Intermediate representation

The intermediate representation to be used is that presented in the *IR* slides and produced in Assignment 1.

Only the part of the IR that *does not* deal with real values must be considered.

2.1 Source of the IR

The code generator may either read the IR from its standard input or be built on top of Assignment 1, in which case it will produce an internal representation of the IR, after reading the abstract syntax tree (AST) of the program, and will generate the code based on that IR.

2.2 Declarations

When the code generator reads the IR from its standard input, it does not have access to the program symbol table. To be able to generate the complete code, the input IR will include information on

- the global variables of the program, including their initial values, if defined;
- the number and types of arguments of each function and procedure; and
- the return type of every function.

2.3 Format of the IR

If you choose to read the IR from the standard input, it will be in the standard format, like it appears in the slides and which was used in class, with a preamble containing the declarations of the global symbols of the program.

Refer to Section 6 for an example.

Declarations

The declarations of the global symbols appear in the following formats:

`(id @`⟨***identifier***⟩ `var` ⟨***type***⟩`)` An uninitialised global variable.

`(id @`⟨***identifier***⟩ `var` ⟨***type***⟩ ⟨***initial-value***⟩`)` An explicitly initialised global variable.

The ⟨*initial-value*⟩ is a literal of type ⟨*type*⟩.

`(id @`⟨***identifier***⟩ `fun` ⟨***type***⟩ `[`⟨***formals***⟩`] [`⟨***locals***⟩`])` A function or procedure.

If ⟨*identifier*⟩ refers to a procedure, its ⟨*type*⟩ is `void`.

The function's formal arguments ⟨*formals*⟩ and local variables ⟨*locals*⟩ are described by a (possibly empty) space-separated list of pairs in the following format, in the order they appear in the original function definition:

`(`⟨*type*⟩ `@`⟨*identifier*⟩`)`

Function header

The IR instructions of a function (or a procedure) start on the line following a line of the form

`function @`⟨*identifier*⟩

where ⟨*identifier*⟩ is the name of the function.

The function's instructions are followed by an empty line.

3 MIPS code

The code generator must output the produced MIPS code on its standard output.

The code produced should be runnable on the MARS MIPS simulator.[1] MARS simulates a 32 bit MIPS system.

Most of the MIPS instructions needed to implement TACL may be found on the *MIPS Reference Data Card* which accompanies the 5th edition of the book *Computer Organization and Design: The Hardware/Software Interface* (COD5), by David A. Patterson and John L. Hennessy. This card may be downloaded from the course moodle page or from the book site.[2]

The MARS help window also details the instructions it supports.

3.1 Allowed instructions

Only *real* MIPS instructions may be used, with the exceptions listed next.

3.1.1 Exceptions

The following are the *only* pseudo-instructions you are allowed to use.

The `la rd, label` pseudo-instruction

The `la` (*load address*) pseudo-instruction puts a memory address in a register. The format of this pseudo-instruction is

```
la rd, address
```

where `rd` is the destination register and `address` may be a label or an integer value.

Being a pseudo-instruction, it must be translated into real MIPS instructions, which are what a MIPS processor understands. MARS translates it to

```
lui $at, 16-most-significant-bits-of-address
ori rd, $at, 16-least-significant-bits-of-address
```

where `lui` shifts its immediate field left 16 bits and loads the result into the destination register, and `ori` adds the 16 remaining bits of the address. (Register `$at`, which is the symbolic name of register `$1`, is called the *assembler temporary* register and is used by assemblers in the translations of pseudo-instructions.)

This pseudo-instruction may be used to access TACL global variables, since their address is only known when the MIPS code is assembled.

The `lw rd, label` pseudo-instruction

The `lw` (*load word*) instruction is a real MIPS instruction, which takes three arguments

```
lw rd, offset(rs)
```

[1] MARS can be downloaded from http://courses.missouristate.edu/KenVollmar/MARS/.
[2] The COD5 site is http://booksite.elsevier.com/9780124077263/.

where `rs` and `rd` are MIPS registers, and `offset` is a 16 bit signed integer value. The effect of the instruction is to load the 32 bit word at memory address `rs + offset` into register `rd`.

In its two argument form it corresponds to a pseudo-instruction, with format

```
lw rd, address
```

whose effect is to load the 32 bit word at memory address `address` into register `rd`. As for the `la` pseudo-instruction, `address` may be a label or an integer value.

As with any pseudo-instruction, it must be translated into real MIPS instructions. MARS translates it to

```
lui $at, 16-most-significant-bits-of-address
lw  rd, 16-least-significant-bits-of-address ($at)
```

Using this instruction provides an alternate way to deal with global variables. To load the value of global variable `variable` into a register, you may use either of the following idioms:

```
lw rd, variable
```

or

```
la rd, variable
lw rd', 0(rd)
```

The `sw rd, label` pseudo-instruction

The `sw` (*store word*) instruction is also a real MIPS instruction with three arguments, whose format is:

```
sw rd, offset(rs)
```

Its effect is to store the 32 bit word found in register `rd` into memory, at address `rs + offset`.

The relationship between the two argument pseudo-instruction format and the three argument real instruction format is the same as for the `lw` instruction above.

The `nop` pseudo-instruction

You may also use the `nop` (*no operation*) pseudo-instruction, if you need to use an instruction which does nothing. (This one translates to `sll $0, $0, 0`.)

This pseudo-instruction may be useful if the code you generate is meant to be executed with *delayed branching* (see the next section).

3.2 Delayed branching

You may consider that the code will be run without using *delayed branching*, *i.e.*, you do not have to worry about *delay slots*.

3.3 Function calling convention

You may use the function calling convention discussed in class, where arguments are passed in the stack frame, function results are returned in register `$v0`, and registers `$t0` to `$t9` (numbers 8 to 15, plus 24 and 25) are caller-saved registers, available to use freely inside a function. Additionally, you may also use registers `$s0` to `$s7` (numbers 16 to 23) as caller-saved registers.

4 Assembly code format

The assembly code produced will comprise *data* and *text* sections.

The data and text sections may be split across several parts, each beginning with the respective directive. Assemblers collect all these parts into a single data section and a single text section.

The relative order of the data and text sections parts is not important.

4.1 Data section

A data section begins with the assembly directive `.data` and contains the global variables of the program.

If the TACL program includes the following global declarations

```
var int ones = 1111;
var bool flag = false;
var int xpto;
```

its data section may look like

```
        .data
ones:   .word 1111
flag:   .word 0
xpto:   .space 4
```

where `.word 1111` means that `ones` is the address of a (data) memory position that contains a 32 bit integer with value 1111, `.word 0` means that the address `flag` points to a memory position with the 32 bit integer value 0 (booleans are implemented as integers), and `.space 4` means that `xpto` points to a memory area with 4 uninitialised bytes (32 bits).

4.2 Text section

The text section, which begins with the `.text` directive, contains the MIPS assembly program. Since all TACL code appears inside functions and procedures, the first instruction of their MIPS implementation will have a label equal to the name of the function or of the procedure.

If the program includes the following function

```
fun int twice(int n) = 2 * n;
```

the text section will include

```
        .text
twice:  ...
        MIPS assembly code for the twice function
        ...
```

4.3 Labels

Labels may include the dollar sign (`$`), besides letters, decimal digits and underscores.

All labels must be different from each other. To guarantee that compiler generated labels do not clash with function or (global) variable names, in the examples, all IR labels are rewritten to include a dollar sign.

4.4 Procedure or function `main`

The execution of a TACL program stars with the execution of the `main` procedure or function.

To make MARS start execution at `main`, the symbol `main` must be declared as a *global symbol*. This is accomplished by including in the assembly code the directive

```
.globl main
```

(You must also activate the MARS setting "*Initialize Program Counter to global 'main' if defined*".)

4.5 Printing

Printing of values in MARS must be accomplished through *system calls*, invoked by the assembly instruction `syscall`. This instruction reads the system-call number from register `$v0`, and the arguments it may need from either register `$a0` or `$f12`. (MARS help window details the system calls supported.)

To simplify the task of implementing the printing of a value, *assembly macros* may be used. The following is the assembly code of the `i_print$` macro (the dollar sign is a legal symbol in an assembly identifier), which prints an integer value, followed by a newline. This macro has an argument called `%int`, which will be the register where the integer to be printed is to be found.

```
# print an integer
.macro i_print$ (%int)
or $a0, $0, %int
ori $v0, $0, 1
syscall
ori $a0, $0, '\n'
ori $v0, $0, 11
syscall
.end_macro
```

Relying on this macro, if the IR contains the instruction

$$\mathsf{i\_print}\ t_8$$

and t_8 is assigned to MIPS register `$t5`, it can be translated simply to the assembly instruction

```
i_print$ $t5
```

which the assembler replaces by the code of the `i_print$` macro, with `$t5` substituted for `%int`.

Note that a macro definition must appear in the assembly code *before* any use of the macro. (On the other hand, there is no need to include the printing macros definitions in the generated code if you do not assign values to registers.)

Below are shown macros to print, respectively, a boolean value (as `true` or `false`) and, for completeness sake, a double precision real value. The argument of the `b_print$` macro will be the register containing the boolean value. (Notice the use of the `la` pseudo-instruction.)

```
        .data
true:   .asciiz "true\n"
false:  .asciiz "false\n"
bool$:  .word false true

        # print a boolean value
        .macro b_print$ (%bool)
        la    $a0, bool$
        sll   $v0, %bool, 2
        addu  $a0, $a0, $v0
        lw    $a0, 0($a0)
        ori   $v0, $0, 4
        syscall
        .end_macro
```

The `r_print$` macro will print the double precision real value contained in the (double precision) floating point register passed as its argument, followed by a newline.

```
        # print a (double precision) real
        .macro r_print$ (%real)
        mov.d $f12, %real
        ori $v0, $0, 3
        syscall
        ori $a0, $0, '\n'
        ori $v0, $0, 11
        syscall
        .end_macro
```

Note that these macros overwrite the values in registers `$a0` and `$v0` (and `$f12`).

5 Practical aspects

Your program must read the TACL code or the IR from standard input and write the generated MIPS assembly code to the standard output. You may choose any language(s) you like for the program. If the program input is the IR, the parser for reading it must be written using suitable tools (like `flex` + `bison` and JFlex + CUP).

Test examples will be provided as TACL code, its IR, and the corresponding MIPS code, both with and without assigned MIPS registers.

The output of your program does not have to be equal to the examples, namely to those shown in Section 6.

5.1 Grading

Instruction selection (9 points out of 10)

This corresponds to instruction-by-instruction instruction selection, without register allocation or delayed branching.

Simple register allocation (1 point out of 10)

This corresponds to register allocation by using the simple register allocation algorithm. You may assume that temporaries are only defined and used once, that their live ranges do not cross basic block boundaries, and that registers don't need to be spilled.

Register allocation across blocks (extra credit)

Extra credit may be obtained if your register allocation algorithm handles register allocation for temporaries with multiple definitions and uses, or with live ranges which cross basic block boundaries, and it contemplates register spilling when there are more values live simultaneously than the number of available registers.

Maximum extra credit is 1 point.

5.2 Deadline

The deadline for handing in your assignment is Friday, 5 January 2018.

Your program should be accompanied by a report describing what you have implemented and how, the function calling convention, the format of the activation records, and the choices you were faced with and the decisions you made. It should also include any comments you feel are useful to make with respect to this assignment.

The report must include instructions on how to compile and execute your program, along with the versions of the tools which are required.

Remember that the goal of a report is to help understand what you have done.

5.3 Bugs

Please contact me if you find any bugs in this text or in the examples, or if you have any questions.

6 Example

This section presents one example which includes: the TACL code; the IR in standard format; one possible MIPS implementation that keeps the IR temporaries; and one MIPS implementation after register allocation.

6.1 Function factorial

6.1.1 TACL code

```
fun int factorial(int n)
[
  var int r = 1;

  if (n > 0)
    r = n * factorial(n - 1);

  ^ r
]
```

6.1.2 IR

Standard format

```
(id @factorial fun int [(int @n)] [(int @r)])

function @factorial
        t0 <- i_value 1
        @r <- i_lstore t0
        t1 <- i_aload @n
        t2 <- i_value 0
        t3 <- i_lt t2, t1
        cjump t3, l0, l1
l0:     t4 <- i_aload @n
        t5 <- i_aload @n
        t6 <- i_value 1
        t7 <- i_sub t5, t6
        t8 <- i_call @factorial, [t7]
        t9 <- i_mul t4, t8
        @r <- i_lstore t9
l1:     t10 <- i_lload @r
        i_return t10
```

6.1.3 MIPS assembly with temporaries

```
        .text
factorial:
        sw    $fp, -4($sp)
        addiu $fp, $sp, -4
        sw    $ra, -4($fp)
        addiu $sp, $fp, -8
        ori   t0, $0, 1
        sw    t0, -8($fp)
        lw    t1, 4($fp)
```

```
        ori   t2, $0, 0
        slt   t3, t2, t1
        beq   t3, $0, l$1
        j     l$0
l$0:    lw    t4, 4($fp)
        lw    t5, 4($fp)
        ori   t6, $0, 1
        subu  t7, t5, t6
        addiu $sp, $sp, -4
        sw    t7, 0($sp)
        jal   factorial
        or    t8, $0, $v0
        mult  t4, t8
        mflo  t9
        sw    t9, -8($fp)
l$1:    lw    t10, -8($fp)
        or    $v0, $0, t10
        lw    $ra, -4($fp)
        addiu $sp, $fp, 8
        lw    $fp, 0($fp)
        jr    $ra
```

6.1.4 MIPS assembly with registers

```
        .text
factorial:
        sw    $fp, -4($sp)
        addiu $fp, $sp, -4
        sw    $ra, -4($fp)
        addiu $sp, $fp, -8
        ori   $t0, $0, 1
        sw    $t0, -8($fp)
        lw    $t0, 4($fp)
        ori   $t1, $0, 0
        slt   $t0, $t1, $t0
        beq   $t0, $0, l$1
        j     l$0
l$0:    lw    $t0, 4($fp)
        lw    $t1, 4($fp)
        ori   $t2, $0, 1
        subu  $t1, $t1, $t2
        addiu $sp, $sp, -4
        sw    $t0, 0($sp)
        addiu $sp, $sp, -4
        sw    $t1, 0($sp)
        jal   factorial
```

```
        or    $t1, $0, $v0
        lw    $t0, 0($sp)
        addiu $sp, $sp, 4
        mult  $t0, $t1
        mflo  $t0
        sw    $t0, -8($fp)
l$1:    lw    $t0, -8($fp)
        or    $v0, $0, $t0
        lw    $ra, -4($fp)
        addiu $sp, $fp, 8
        lw    $fp, 0($fp)
        jr    $ra
```