UNIVERSIDADE DE ÉVORA

Escola de Ciências e Tecnologias
Mestrado em Engenharia Informática

Tópicos Avançados de Compilação
2015/2016

Assignment 2: MIPS Assembly Code Generation

Autor:
Ricardo Fusco

Professor:
Vasco Pedro

Évora, 22 de Janeiro de 2016

1 Introduction

This assignment consists in making a MIPS Assembly code generator, given the IR representation generated in the first assignment, for the TACL language created by the professor for this curricular unit, a language a bit similar to C or Java in some aspects but a simpler one. The input form chosen for this assignment was the standard format in Prolog in which each function prolog term holds 2 arguments, the function identifier as an atom and the list of IR instruction for that function in which each IR instruction is also represented by a prolog term. The code produced is meant to be runnable on the MARS MIPS simulator which simulates a 32bit MIPS system.

2 Tools used

Prolog language was used (Swi-Prolog) to make the MIPS generator due to the fact that the IR is given either in 3 ways, a standard form like its seen in the slides and which was used in class, or in the form of prolog terms, either in Prolog standard format or in Prolog IR trees. Given those Prolog terms it becomes much easier to generate the MIPS assembly code using Prolog taking away the need to have a parser for the IR previously generated. It interprets each instruction, or block of instructions, by having a prolog predicate that matches the terms for each of those instructions.

3 Code Developed and MIPS Assembly Code

In order to generate the assembly code there were some first issues that had to do with how to properly "parse"through the list of instruction within the function blocks in order to correctly print the assembly instructions for each IR instruction and manage the problems regarding the use of the several MIPS registers and treating the cases where functions are called within other functions by using the call stack in order to hold each variable instantiation. Managing the registers and replacing the temporaries by the adequate register was also an issue to start with.

Every instruction used by the IR instructions was implemented, and the code generation process seems to be correct. The MIPS assembly code was generated with registers instead of only temporaries. The code was split into 4 different files. One is the main file which holds the predicates to run the generator for global declarations and function predicates, to process the list of instructions for each function and helper predicates in order to make the sign extension when needed.

This sign extension is needed for when we need to load immediate values lower than -32768 and greater than 65535 creating the need to divide the i_value instruction into

two different MIPS instructions, one for loading the upper 16 bits and another one to concatenate the lower 16 bits for the whole 32 bit immediate.

In the Instruction list processing algorithm there is a slight code optimization bit for immediate values with only 16 bits which checks each time there is an i_value instruction followed by an i_sub or i_add which uses the register where the immediate value was placed and they are replaced with a single addiu instruction when generating the code, for example in this case:

```
i_value t1, 5
i_add t0, t1, t2
```

these instructions are replaced by this MIPS instruction:

```
addiu t0, t2, 5
```

The other 3 files are the instructions file, prints file and stack_operations file. In the stack operations file there is all the code related to the runtime stack, stack frames, symbol table to map the registers and temporaries and everything related to the registers. In the print file there is all of the code for prints, for instance, for global variables, for functions, function prologue and function epilogue prints, macros and stack pushes and pops. Lastly, the instructions file contains all of the predicates matching the IR instructions terms and the MIPS assembly instructions predicates where the correct instruction are printed, all of the instructions (including the pseudo-instructions referred by the teacher) are implemented and present in this file.

Regarding the register management, taking into account that the IR can use any number of temporaries as it needs to, a solution had to be found to solve this problem due to the limitations in the number of registers available in MIPS. During the code generation the registers are discarded every time it is used as an operator in an instruction, the register is then reallocated back into the registers stack to be sorted afterwards. For all of this to work, as mentioned before, a registers stack and a symbol table were created in order correctly map every temporary to a register making it easier to replace the temporaries throught the code by actual registers. In function calls the registers being used are stored in the stack, and then place the arguments in the stack. When the function comes to an end the values are then loaded to the respective registers.

Regarding the macros for i_print, b_print and r_print they can be found in the prints file and these macros are printed in every generated MIPS assembly code.

4 Instructions

The main file for running the program is the main.pl one and the program can be run in two ways, one can open Swi-Prolog (Swi-Prolog was the environment used for this) and consult the main.pl file by running "['main.pl']. "or "consult('main.pl'). "and then pasting the IR instructions prolog terms for the TACL code in the terminal and the program will output the MIPS assembly code, or a command can be run while in the terminal redirecting the file to the standard input -¿ "swipl main.pl ¡ example.pl "and the IR will be printed to the standard output, if a file with the output is wanted one can run the same command with one more thing -¿ "swipl main.pl ¡ example.pl ¿ output.asm". There is also a bash file in the assignment folder called "run_tests.sh"with a small script to runs all the IR example prolog terms in .pl files within the inputs folder and outputs the IR to the outputs folder to .asm files in order to run on MARS.