UNIVERSIDADE DE ÉVORA

Escola de Ciências e Tecnologias
Mestrado em Engenharia Informática

Tópicos Avançados de Compilação
2016/2017

Assignment 2: MIPS Assembly Code Generation

Author:
Ricardo Fusco

Professor:
Vasco Pedro

Évora, January 14, 2017

Contents

1 Introduction

The second assignment consists in building a MIPS Assembly code generator, given the AST obtained after parsing the Intermediate Representation generated from the TACL language created by the professor for this curricular unit. In this assignment only the part of the IR that does not deal with reals is considered.

2 Tools used

The tools used for this assignment were JFlex and Cup for building compilers using Java. Regarding JFlex there was a slight problem in this assignment with its versions that was solved after some research. The problem was with the generated Yylex.java file and its constructors.

In the first assignment **JFlex 1.4.3** was used and in the cup file's main method a new Yylex object is created passing as argument the standard input (*System.in*) which is an *InputStream*. The generated Yylex.java had 2 constructors in that version of JFlex, a constructor that received an *InputStream* and another that received a Reader and at that point passing the System.in input stream as argument to Yylex was enough.

After the first assigment I updated **JFlex** to version **1.6.1** and got errors while compiling, after researching what the error was I realized that at some point between versions **1.4.3** and **1.6.1** one of the two constructors for the generated Yylex file was eliminated, namely the constructor that received an input stream so I had to create a new *InputStreamReader* object that received the input stream so that the Yylex constructor receives a Reader instead of the raw input stream.

3 MIPS Assembly code generation

3.1 Issues

The first minor issue in the beginning was coming up with a grammar to parse the IR but it was much easier than it was for the first assignment because it is a simpler grammar and an instruction by instruction approach. One aspect that was, not really an issue, forgetting to include in the first assignment negative numbers in the regular expression for the numbers in the lexer.

Then the next issue was coming up with the direct translations for each IR instruction and how to deal with the arguments and locals for each function and the offset taking these into account. It was easily solved by holding all the locals and arguments for each

function in a vector declared as a class variable for the class where the code is generated and having a class variable to hold the offset for the prologue, epilogue and the data transfer instructions.

Another minor issue was figuring out how to structure the grammar and the AST node visitors in one of two ways, either have a simpler grammar and a more complex and dense instruction treatment in just one node visitor, or make the grammar a little bit more complex and divide instructions in several parts and nodes. I went with the second option in order to have everything more organized and structured.

3.2 Implementation details

The implementation of the AST processing methods was based on the first assignment which is an implementation that uses the Visitor pattern in which an interpretation is implemented by a visitor.

Each class for each element meant to be interpreted in the abstract syntax tree contains an accept method which is called by the visitor and its purpose is to pass control and be handled by its corresponding method inside the visitor. The class for the visitor is the class where all the visit methods corresponding each of the AST node classes are. This control is passed constantly between the visitor and the AST nodes classes.

The visitor basically calls the accept method to know which is the class it is visiting and calls the appropriate visit method from the visitor. In Java this can be done by having an interface for the visitor with all the visit methods for each of the AST node that is meant to be interpreted. Then in each AST class there will be the accept method which calls the corresponding visit method. And finally there is the class which I called *Assembly_Gen* that implements the visitor interface and implements all the visit methods that will generate the MIPS Assembly code for each of the AST nodes.

Regarding the grammar the instructions part of the grammar was divided into several parts, in three major instruction groups (two operand, one operand and data transfer instructions), calls, prints, jumps, cjumps, labels and returns. The grammar became a bit bigger but it simplified and gave a better structure to the instructions' processing in the visitor class.

In this assignment the chosen method for processing the IR was the simple **instruction-by-instruction instruction select** with no register allocation or delayed branching.

In the IR node visitor the first thing to check is if there are global variable declarations in order to know when to print the *.data* section. Afterwards every global symbol

declaration will be visited by processing the preamble in a cycle and for each function definition in the IR preamble the locals and formal arguments information will be saved in the *functions* vector. After processing the preamble, all of the functions and their instructions will be processed, checking for each function if the number of formals and locals in order to print the prologue and epilogue after processing the instructions list for that function. After processing each function the data from the preamble will be removed from the *functions* vector.

3.3 Instruction selection and code generation

3.3.1 Prologue, Epilogue, and Labels

Regarding the prologue and epilogue the calling convention followed was the same taught in classes and used in the examples provided. In the prologue we save the caller frame pointer, set the callee frame pointer, save return address and allocate space for the local variables for the funtion:

```
sw $fp, -4($sp) // save caller frame pointer
addiu $fp, $sp, -4 // set callee frame pointer
sw $ra, -4($fp) //save return address
addiu $sp, $fp, offset // allocate space for locals
```

The offset will be the number of local variables plus the return address times -4 due to the immediate being a 16 bit integer (4 bytes) and the fact that locals are in lower memory addresses so in this case: offset = (locals.size()+1)*(-4) .

In the epilogue we copy the result to $v0 (function result when not a procedure), restore the return address, restore the caller stack pointer, restore caller frame pointer and return from function by jumping to the return address:

```
lw $ra, -4($fp) // restore return address
addiu $sp, $fp, offset  // restore caller stack pointer
lw $fp, 0($fp) // restore caller frame pointer
jr $ra // return from function
```

The offset here will be the number of formal arguments plus the return address times 4 due to the immediate being a 16 bit integer (4 bytes) and the fact that arguments are in higher memory addresses so in this case: offset = (args.size()+1)*4 .

Regarding the labels in the label visitor the label is simply printed with the dollar sign ($) included.

3.3.2 Two operand instructions

The translations for the two operand instructions were pretty straightforward and were decided based on the slides and examples from the teacher.

- `t1 <- i_add t2, t3:`

```
addu t1, t2, t3
```

- `t1 <- i_sub t2, t3:`

```
subu t1, t2, t3;
```

- `t1 <- i_mul t2, t3:`

```
mult t2, t3
mflo t1
```

- `t1 <- i_div t2, t3:`

```
div t2, t3
mflo t1
```

- `t1 <- mod t2, t3:`

```
div t2, t3
mfhi t1
```

- `t1 <- i_eq t2, t3:`

```
subu t1, t2, t3
sltiu t1, t1, 1
```

- `t1 <- i_lt t2, t3:`

```
slt t1, t2, t3
```

- `t1 <- i_ne t2, t3:`

```
subu t1, t2, t3
sltu t1, $0, t1
```

- `t1 <- i_le, t2, t3:`

```
slt t1, t2, t3
xori t1, t1, 1
```

3.3.3 One operand instructions

The one operand instructions were also pretty straightforward based also on the slides and the examples provided.

- `t1 <- i_inv t2:`

  ```
  subu t1, $0, t2
  ```

- `t1 <- not t2:`

  ```
  xori t1, t2, 1
  ```

- `t1 <- i_copy t2:`

  ```
  addu t1, $0, t2
  ```

3.3.4 Data transfer instructions

Regarding the data transfer instructions there were some things to consider like the calculation of the offset in the loads and stores for locals and arguments and for the *i_value* instruction the cases where the binary representation of the integer value in the immediate field exceeds 16 bits which leads to a need to divide the operation in two instructions.

- `t1 <- i_value value:`

  1. If value >= 65536 or value < -32768

     ```
     lui t1, (value >> 16)
     ori t1, t1, (value & 65536)
     ```

  2. If value < 65536 and value >= -32768

     ```
     ori t1, $0, value
     ```

- `t1 <- i_aload @id:`

  ```
  lw t1, offset($fp)
  ```

  The offset here will be the number of arguments minus the index of the argument that is being loaded (already accounts for the return address and accounts for the arguments' disposition from argument n to 1, meaning it will fetch the arguments index counting from the end) times 4 : offset = (args.size() - args.indexOf(id))*4 .

- `t1 <- i_lload @id:`

  ```
  lw t1, offset($fp)
  ```

  The offset here will be the index of the local that is being loaded plus two (plus two to get the correct position in the stack and to account for the return address) times -4 : offset = (locals.indexOf(n.id)+2)*(-4) .

- `t1 <- i_gload @id:`

  ```
  la t1, id
  lw t1, 0(t1)
  ```

- `@id <- i_astore t1:`

  ```
  sw t1, offset($fp)
  ```

  The offset here will be the number of arguments minus the index of the argument that is being stored (already accounts for the return address and accounts for the arguments disposition from argument n to 1, meaning it will fetch the arguments' index counting from the end) times 4 : offset = (args.size() - args.indexOf(id))*4 .

- `@id <- i_lstore t1:`

  ```
  sw t1, offset($fp)
  ```

  The offset here will be the index of the local that is being stored plus two (plus two to get the correct position in the stack and to account for the return address) times -4 : offset = (locals.indexOf(n.id)+2)*(-4) .

- `@id <- i_gstore t1:`

  ```
  la $at, id
  sw t1, 0($at)
  ```

3.3.5 Call instruction

For the call instruction the first thing to do is push all the functions' arguments to the stack from the last to the first, and then call the function, and gets the calls' return value if it is not a procedure.

```
// t0 <- i_call @id, [t1, t2]
    addiu $sp, $sp, -4
    sw t2, 0($sp)
    jal id
    or t0, $0, $v0
```

3.3.6 Print instruction

Regarding the print instruction it just prints the print with a dollar sign at the end.

```
i_print$ t0
```

3.3.7 Jump instruction

```
j label
```

3.3.8 Cjump instruction

```
// cjump t0, l1, l2
beq t0, $0, l$2
j l$1
```

3.3.9 Return

In the return part it just sets the return value for the function, if it is a procedure does nothing.

```
// i_return t0
or $v0, $0, t0
```

4 Instructions on how to run the assignment

There is a Makefile included in the assignments' folder. So to compile the program just run **make** and to run all the examples in the examples folder just run **make run-all** and the files with the MIPS Assembly code with temporaries will be generated in the same folder.