Masters Degree in Informatics Engineering

Instruction Selection

Assignment 2

Discipline:
Advanced Topics in Compiling
Teacher:
Vasco Pedro
Author:
João Pedro Amaral Dias
Nº 42055
Date:
January 10th 2019

Introduction

The main objective of this project is to convert Intermediate Representation code into MIPS code. This conversion will be made instruction-by-instruction, taking into account the context of each instruction.

The input will be a .ir2 file with information from the programs global variables and the arguments and return type of the program functions. The input must have this information in the beginning because the parser will not have access to the program symbol table.

Program functions will be stored in a linked list of function nodes in order to simulate a stack frame.

The final result will be a MIPS program without register allocation and optimizations.

1. Declarations

The IR program starts with global variables and functions declarations. The global variable declarations are handle by initialization of a .data space where the variables will be displayed.

- Initialized variable
  identifier: .word *value*
- Non initialized variable
  identifier: .space 4

The function declaration is handle by adding a new node into a linked list of function nodes. If is the first function being declared in the program, the list is created with only that function node.

Every node of each function has two string arrays. One to store the name of the function arguments and other to store the name of the function local variables. There is also two int values with the size of both arrays.

The list itself has a variable for the size and other to represent the current position of the function being treated.

2. Instruction Selection

The next step of the work is the direct conversion of IR instructions into MIPS instructions.
In this project MIPS instruction are divided into nine categories.

2.1. Binop Instructions

Intermidiate Representation Instruction	MIPS Instruction
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, t3, t2 xori t1, t1, 1

2.2 Unop Instructions

Intermidiate Representation Instruction	MIPS Instruction
t1 ← i_inv t2	subu t1, $0, t2
t1 ← not t2	xori t1, t2, 1
t1 ← i_copy t2	add t1, t2, $0

2.3 Value Instruction

Intermidiate Representation Instruction	MIPS Instruction
t1 ← i_value N	if N < 65536

	ori t1, $0, N else lui t1, N1 ori t1, N, N1

2.4 Load Instructions

Intermidiate Representation Instruction	MIPS Instruction
t1 ← i_gload @name	la t1, name lw t1, 0(t1)
t1 ← i_lload @name	lw t1, -N($fp)
t1 ← i_aload @name	lw t1, N($fp)

The N value is calculated by getting the current function and finding the position of either the local variable or function argument in the respective array in the function node of the list. This position is then multiplied by 4 because it occupies 32 bits.

2.5 Store Instructions

Intermidiate Representation Instruction	MIPS Instruction
@name ← i_gstore t1	la $at, name sw t1, 0($at)
@name ← i_lstore t1	sw t1, -N($fp)
@name ← i_astore t1	sw t1, N($fp)

2.6 Call Instructions

Intermidiate Representation Instruction	MIPS Instruction

t1 ← i_call @name, [t2]	addiu $sp, $sp, -4
	sw t2, 0($sp)
	jal name
	or t1, $0, $v0

call @name, [t3]	addiu $sp, $sp, -4
	sw t3, 0($sp)
	jal name

The first two instruction in the call set of instructions are executed for each argument that the function receives.

2.7 Jump Instructions

Intermidiate Representation Instruction	MIPS Instruction
jump l0	j l$0
cjump t1, l1, l2	beq t1, $0, l$2 j l$1

2.8 Return Instructions

Intermidiate Representation Instruction	MIPS Instruction
i_return t1	or $vo, $0, t1

2.9 Print Instructions

Intermidiate Representation Instruction	MIPS Instruction
i_print t1	i_print$ t1
b_print t1	b_print$ t1

3. Prologue and Epilogue

Every function when it starts must create a activation record in the stack frame and then when it ends must remove it. The prologue is a set of instructions that create the activation record when a function is called. The epilogue stores the result of the function and eliminates the activation record.

3.1 Prologue

Prologue is constituted by 4 instructions. The first one aims to save the caller's frame pointer.

- sw $fp, -4 ($sp)

The second one initializes the callee's frame pointer.

- addiu $fp, $sp, -4

Next the callee's return address is saved.

- sw $ra, -4 ($fp)

Lastly is alocated room in the stack for the local variables.

- addiu $sp, $fp, -N

3.2 Epilogue

The epilogue starts by copying the function return to $v0 register. Then the return address is restored.

- lw $ra, -4 ($fp)

The second instruction in the epilogue is the pop of the activation record in the stack frame.

- addiu $sp, $fp, N

Next the caller's frame pointer is restored.

- lw $fp, 0 ($fp)

The execution is terminated by executing a jump to the return address.

- jr $ra

4. Commands

This project is delivered with a Makefile with the following commands.

Command	Execution
make run	Run the program with a standard input typed by the user in the terminal.
make run-1	Run the example file twice.ir2
make run-2	Run the example file triangular.ir2
make run-3	Run the example file example.ir2
make run-4	Run the example file factorial.ir2
make run-5	Run the example file fig-9.ir2
make run-6	Run the example file ragbag.ir2
make dif-1	Compare the result with the example for the file twice.mips
make dif-2	Compare the result with the example for the file triangular.mips
make dif-3	Compare the result with the example for the file example.mips
make dif-4	Compare the result with the example for the file factorial.mips
make dif-5	Compare the result with the example for the file fig-9.mips
make dif-6	Compare the result with the example for the file ragbag.mips
make clean	Delete files

To run this program the tools needed are.

- Bison (GNU Bison) 3.0.x
- Flex 2.6.x