UNIVERSIDADE DE ÉVORA

Mestrado em Engenharia Informática

TÓPICOS AVANÇADOS DE COMPILAÇÃO

Code Generator
IR to MIPS

Elaborado por:
m37280, André Figueira

1. Instruction Selection

In the following Sections it shows the **Intermediate Representation Instructions** and it's equivalent transformed to the MIPS Instructions using IR registers. Since the assignment was done using IR registers.

1.1 Arithmetics Instructions

Intermediate Representation Instructions	MIPS Instructions
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 <- i_inv t2	subu t1, $0, t2
t1 <- mod t2, t3	div t2, t3 mfhi t1
t1 <- i_copy t2	add t1, t2, $0

Table 1.0

1.2 Compare Instructions

Intermediate Representation Instructions	MIPS Instructions
t1 <- i_lt t1, t2, t3	slt t1, t2, t3
t1 <- i_eq t2, t3	subu t1, t2, t3 sltiu t1, t1, 1
t1 <- i_ne t2, t3	subu t1, t2, t3 sltiu t1, $0, t1
t1 <- i_le t2, t3	slt t1, t3, t2 xori t1, t1, 1

Table 2.0

1.3 Jump Instructions

Intermediate Representation Instructions	MIPS Instructions
cjump t1, l0, l1	beq t1, $0, l$1 j l$0
jump l0	j l$0

Table 3.0

1.3 Return Instructions

Intermediate Representation Instructions	MIPS Instructions
i_return t1	or $v0, $0, t1
return	No instruction needed

Table 4.0

1.4 Print Instructions

Intermediate Representation Instructions	MIPS Instructions
i_print t1	i_print$ t1
b_print t1	b_print$ t1

Table 5.0

1.5 Load and Store Instructions

Intermediate Representation Instructions	MIPS Instructions
t1 <- i_gload @name	la t1, name lw t1, 0($t1)
t1 <- i_aload @name	lw t1 , +X($fp)
t1 <- i_lload @name	lw t1, -X($fp)
@name <- i_gstore t1	lw $at, name sw $t1, 0($at)
@name <- i_astore t1	sw t1, +X($fp)
@name <- i_lstore t1	sw t1, -X($fp)

Table 6.0

For instructions that are not **gload** or **gstore**, the offset is an X, this because the remaining instructions are for arguments and local variables, and so their offset given the **frame pointer** on the stack frame, is different for each argument and local. The explanation for finding the X is detailed in **Section 2.2.**
As said before, for **gstore** and **gload** finding the X is not required for global variables, instead it usesthe pseudo instruction **"la"**, which stands for **load address**, gives the exact position in memory given its name.

1.6 Call Instruction

Intermediate Representation Instructions	MIPS Instructions

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

call @func, [t1]	addiu $sp, $sp, -4
	sw t1, 0($sp)
	jal func

Table 7.0

Before calling the function, that is before the **jal** instruction, it's the job of the caller to save the arguments onto the stack so when the jump is executed the stack frame for that function is correct. The arguments are saved onto the stack from last to first argument. For example:

call @func, [t1,t2], register t2 will be saved and t1 will be saved after.

1.7 Other Instructions

Intermediate Representation Instructions	MIPS Instructions
t1<- i_value X	ori t1, $0, X [X < 65536]
t1<- i_copy t2	add t1, $0, t2
t1 <- not t2	xori t1, t2, 1

Table 8.0

In regards to i_value, it may take up to two instructions, instead of one, this because, as seen in the **table 8.0**, that instruction will be enough if X is lower than 2^16 bits. If it is higher, then another instruction is needed, and that is "**lui**". So when generating the MIPS code, the following must be done, **if X >= 65536**:

lui t1, N -> N being the value of X divided by 65536
ori t1, t1, N2 -> N2 being the remainder of X divided by 65536

1.8 Prologue and Epilogue of a Function

1.8.1 Prologue

Composed of 4 instructions, which of first three are generic and the final one requires more handling since it's referring to the stack pointer which must be followed by the return address and all the Locals, **in short**, its position in the Stack must be updated, so it corresponds to the correct layout of a **Stack Frame**.
Therefore the amount required will depend on the amount of locals plus times their size and plus 4 for the return address:

```
sw $fp, -4($sp)
addiu $sp, $fp, -4
sw $ra, -4($fp)
addiu $sp, $fp, -(X)

X = return address (4) + number of Locals * 4
```

it's times 4, because in this assignment it is only considered integer values, and boolean values are treated the same as integer. Therefore it's assumed all values have size of 4 bytes.

1.8.2 Epilogue

Similar to prologue, the first three instructions are generic and the instruction that restores the value of the **stack pointer** (fourt instruction) requires more handling, but restoring the stack pointer must be done before the function ends, as in, before the jump instruction.
Therefore the amount required will depend on the amount of arguments times their size and plus 4 for the control link:

```
lw $ra, -4($fp)
addiu $fp, $sp, -4
sw $ra, -4($fp)
addiu $sp, $fp, X
jr $ra

X = control link (4) + number of Args * 4
```

2. Declarations of Global Symbols

2.1 Vars

If there are any, in the output there will be a .data section with all the global vars and their names being labels with their corresponding values, if there is no value, then it will be allocated the necessary space for it.

Example (from ragbag.ir2) :

Global Symbols	**.data section**
(id @global1 var int 32)	global1: .word 32
(id @gbool var bool true)	gbool: .word 1
(id @global2 var int)	global2: .space 4

Table 9.0

2.2 Fun

Every **Function Global Symbol** gets checked for its arguments and locals and turned into a Frame adding them to a list (in reverse order, so last is first, because each frame is added at the head of the list) . A frame contains all the arguments and locals that the function has by the following order into an array named **vars**:

frames = [frame_of_functionN, frame_of_functionN-1, .., frame_of_function1]

Table 10.0

```
#define SIZE 20

struct frame
{
    char *vars[SIZE];
    int n_fa;   // last position of formal args
    int n_lc;   // last position of locals
};
```

Figure 1.0

vars = {arg1, arg2, arg3, ... , argM, local1, local2, ... , localN}

Table 11.0

In this frame it also saves the last position of argument **argM** (last argument of the function) and local variable **localN** (last local variable of the function) as seen in Table 11.0 and figure 1.0. So when a **load or store** instruction is used it can be easily found the correct argument or local variable so this way the **code generator** knows where the **arguments end** and **local variables start and end** (so no unnecessary searches are done) and convert them to the corresponding MIPS instruction with the correct offset. That is done by accessing the correct frame (**fr** as shown in Figure 2.0) and searching through the array **vars** for the argument or local variable (for local it starts from the position of **argM + 1**).

Since a function is only treated once, every time a new function label appears, it gets removed from the List and a global var **"fr"** contents is altered so it matches the current function frame being treated.

```
list frames;            // linked list of all frames
frame fr;               // current function frame being used
int func_i;             // index of current function being used
```

Figure 2.0