Code generation for floatting-point arithmetic in architecture MIPS

1st Ivan Arkhipov
faculty of mathematics and mechanics
St Petersburg University
St Petersburg, Russia
arkhipov.iv99@mail.ru

***Abstract*—This article is related to code generation for floatting-point arithmetics in the MIPS architecture. This work is a part of the "RuC" project. It is specialized only in code generation for operations with floatting-point numbers. This paper does not consider lexical, syntactic, and species-specific analyses.**

***Index Terms*—code generation, translator, floatting-point arithmetics, MIPS**

I. Introduction

RISC and CISC architectures, unlike stack architectures and virtual machines systems, have different ways to express high-level language features. There are many registers for working with data, which creates a large variability in optimal code generation. Therefore, code generation in these architectures is quite a difficult task.

For work with such architectures a technique of request and response [1] have been developed at the mathematics and mechanics faculty Leningrad State University: from the top of the constructed parse tree the requests for values are received, and from below the answers – form submission parse (register, memory, constant). In addition, there are certain relationship agreements. For example, in the MIPS32 architecture, function parameter values must be in some specific registers, and function values must be in other specific registers. There are stored registers that must be preserved when calling functions, and there are non-stored (unsafe) registers. For example, the left operand of a binary formula must be represented in a stored register if the right operand has calls or slicing that apply the same rules as functions, otherwise the left operand can also be represented in an unsaved register. Determining the complexity of the right operand is the task of optimizing parse.

This work has practical application: the MIPS32 architecture is the basic architecture of one of the russian computers Baikal-T1 [2].

The goal of this work is to generate codes for floating-point arithmetic in MIPS32 codes using the request and response technique.

II. Motivation

The development of a domestic translator is an actual task, since many industries require domestic software to avoid "back doors" in foreign software. Writing your own translator is a difficult task. This work is part of the "RuC" project [3] and is specialized only in code generation of operations with floatting-point numbers. At the moment, this project has a customer, which is an additional evidence of the relevance of this work.

III. Problem Statement

To achieve the goal of this work, the following tasks were set:

- Implement code generation for operations with floatting-point numbers in RuC using the query and response technique
- Implement printing floatting-point numbers to the console
- Prepare tests and test the implemented code generation

The results can be considered successful if the assembler code received during code generation is executed on the Baikal-T1 machine and displays the correct result in the console.

IV. Overview

Since this work is part of the RuC project, the same ideas as in the RuC are used to achieve the set goals. The code generator will view the parsing tree of the program and generate code based on the lexemes located in it.

It is necessary to describe the principles of RuC in general. The RuC translator has a traditional two-view structure. On the first view a scanner (lexical analyzer), a view-independent analyzer (parser) and a view-dependent analyzer work. The result of the first view is a parsing tree. This tree is input to the second code generation view, which outputs the result in MIPS32 architecture codes. This work is a part of second code generation view module, that implements operations with floatting-point numbers. More information about RuC may be found in section wiki of 'RuC' project github [9] and in the following article [10].

It is also worth mentioning a few general decisions made during the work:

- It was decided to generate commands for working with single-precision floatting-point numbers. This is due to two reasons. Firstly, at the moment there is no need for double-precision calculations on the Baikal-T1 computer. Secondly, the computer Baikal-T1 (another name BE-T1000) has 2 32-bit p5600 processor cores of the MIPS32

r5 architecture, which makes it unsuitable for double-precision computing. For example, because of the 32-bit version, you will need two commands to load a double-precision number from memory, not one.
- Implementation of using registers manually without using LLVM [8], firstly, to support RuC, and secondly, to guarantee the absence of malicious code, since due to the huge amount of code in LLVM, it is difficult to check, for example, the absence of "back doors".
- Processing requests of the register-to-register type only (more on this later), since there are no commands with a direct operand for floatting-point values, and working with memory is represented by only two commands: load and store.

RuC has its own virtual machine, so assembly code could be generated like this: first code in virtual machine codes is generated, and then each virtual machine instruction is translated to MIPS32 assembly code. It was decided to abandon this approach because it generated large code that is difficult to optimize in the future.

V. Related Work

In the process, we also looked at the code generated by the gcc compiler [7] and compared it with our own. Of course, the gcc compiler has already implemented many optimizations, which makes the code generated by it better. At the moment, RuC does not have any optimizations related to arithmetic operations. Optimization is the next stage of RuC development and a topic for future work.

If you compare the code generated by RuC with the non-optimized code generated by gcc, you can see that RuC uses more temporary registers than gcc for intermediate calculations. This approach is closer to a relationship agreement in mips architecture.

RuC has its own virtual machine, so it was possible to generate assembly code like this: firstly generate code in virtual machine codes, and then translate each virtual machine instruction into mips assembly code. We abandoned this approach because it generated a large code that is difficult to optimize in the future.

As an alternative approach to code generation, generation to LLVM [8] codes can be also offered. But, as it was written above, you can not guarantee that there are no "back doors" in LLVM.

As for the application, after further improvements, RuC can be used in areas where a security guarantee is required, which is why it is not possible to use foreign software products. This is the novelty of RuC - it is the first Russian translator that modifies the C language.

VI. Implementation

A. Parse tree lexemes

As described above, the code generator views lexemes from the parse tree. We are only interested in lexemes that describe operations with floatting-point numbers, namely the following:

- TConstf – floatting-point constant
- TIdenttovalf – take the value of an identificator
- "Unary" arithmetic operation lexemes:
  - ASSR – =
  - PLUSASSR – +=
  - MINUSASSR – -=
  - MULTASSR – *=
  - DIVASSR – /=
  - INCR – increment
  - POSTINCR – postincrement
  - DECR – decrement
  - POSTDECR – postdecrement
- "Binary" arithmetic operation lexemes:
  - LPLUSR – +
  - LMINUSR – -
  - LMULTR – *
  - LDIVR – /
- Logic operation lexemes:
  - EQEQR – ==
  - NOTEQR – !=
  - LLTR – <=
  - LGTR – >=
  - LLER – <
  - LGER – >

The processing of each type of lexemes will be shown below.

B. A technique of request and response

Before describing the processing of lexemes, it is necessary to describe the technique of requests and responses. There are several types of requests, we are interested in the following:

- BREG – load the result in a register "breg". "breg" is a global variable in the translator, that contains the register's number.
- BREGF – request on the left operand, you can get an answer. Answers will be shown below.
- BF – free request on the right operand.

Type of request is contained in global variable "mbox". The types of responses are:

- AREG – the result in a register "areg". "areg" is a global variable in the translator, that contains the register's number.
- AMEM – the result in memory. Global variable "adispl" contains displacement and global variable "areg" contains register.
- CONST – result is a constant.

Type of request is contained in global variable "manst".

C. TConstf

This lexeme means that a constant request was received. After this lexeme in the tree there is a constant value. Depending on the request type, we can get a register to put the constant value in ("breg"). If we don't get a register, the constant value is put in a temporary register $f4 with the pseudo instruction li.s. It is described about floating point registers in [4]. The type of the response is AREG.

D. TIdenttovalf

This lexeme means that the value of variable must be put in register by identificator. If this is register variable, it is necessary to move it to the register "breg" when request BREG or BREGF is received. Otherwise we must put the value of this variable from memory in register "breg" or $f4 with the instruction lwc1 [5].

E. "Unary" arithmetic operation lexemes

These operations are called "unary" operations because when when processing them, it is necessary to request the right operand, and the left operand is already known. The left operand may be already in register if it is register variable or in memory. If it is in memory it is necessary to put it in register. Only a register request must be issued for the right operand since there are not operations addition, subtraction, multiplication and division for floating point numbers with register and number operands. So, left and right operands must be in registers.

After this it is necessary to execute the instruction (addition, subtraction, multiplication or division). Then if variable is in memory new value of variable is saved in memory. In "areg" register of left operand is put. The type of the response is AREG.

It is worth noting that the division operation is performed like the rest with a single command, in contrast to the similar operation with integers.

F. "Binary" arithmetic operation lexemes

"Binary" operations differ from "unary" operations in that both the left and right operands must be requested before operation is executed. In contrast to the similar operation with integers for executing operations with floating point numbers left and right operands must be in registers. That's why only a register request must be issued for the left and right operands.

After getting values of left and rights operands in registers the instruction may be executed. This stage is performed as for unary operations. The type of the response is AREG.

G. Logic operation lexemes

Just like in "unary" and "binary" operations both operands must be in registers. That's why only a register request must be issued for the left and right operands.

Unlike in similar operations with integers floating point operations change flag FP. Based on the logic operation, conditional transition commands are generated. If the conditional expression is complex (contains operations "and" or "or"), it is divided into simple logical expressions, the result of which is stored in the global variable in translator. When processing subsequent conditional expressions, the value of this global variable is also taken into account.

H. Printf

To see the results of code generation printf function must be implemented. Firstly, string in data segment is generated. String is given in parse tree after TString lexeme. Then text segment begins again. Address of string is put in register $a0. After this a register request for the second operand is created. If this operand is integer or char value of this operand is put in register $a1 and printf is executed. If this operand is float pointing due to mips agreements we must convert the single precision floating point number to a double precision number. After these operations printf is executed.

If printf has more than one arguments string is divided into several parts and for each part printf is executed.

VII. Evaluation

After implementing code generation for operations with floatting-point numbers and printing floatting-point numbers tests were prepared. Tests have been prepared that demonstrate the code generation for each operation separately and for complex expressions with floatting-point operations. For example, 'RuC' translates a program in application 1 to the assembly code in application 2.

It is important to note that the goal is considered achieved only when the generated code is assembled successfully and is executed on the Baikal-T1. This is significant since we can think that code generation is correct but in fact it does not work. Also in such way successfulness of this work can be demonstrated.

For this purpose:

- emulator qemu [6] was installed;
- Baikal-T1 was bought;
- Baikal-T1 was connected to a laptop.

After this the prepared tests for arithmetic operations were first tested on the emulator, and then on the Baikal-T1. The tests were successful, so we can assume that the goal was achieve. You can find tests in [3] in branch mips.

VIII. Conclusion

This work solves the problem of code generation in MIPS32 codes of arithmetic operations with floating point numbers. Various approaches to code generation have been considered and one of them has been implemented – direct code generation.

The novelty of this work is that this work is part of the RuC project, the first Russian translator to modify the C language in favor of programming security.

In the course of the work important results were obtained showing the applicability of the results of this work in practice. Direct code generation was implemented MIPS32 codes for floating point arithmetic operations. The generated code was successfully run on Baikal-T1.

This work has many opportunities for further research. The RuC project is not yet complete, and some C language structures are not yet implemented. Optimization of generated code is also a big area of research. Also it is necessary to implement a linker. As you can see, there are still many sources for research.

References

[1] A. N. Baluev, I. L. Bratchikov, I. B. Gindysh, N. A. Krupko, G. S. Tseytin, A. N. Terekhov, (total, 12 people), “ALGOL 68. Method of implementing.” Saint Petersburg University press, 1976.

[2] Baikal-T1 specifications – URL: http://www.baikalelectronics.ru/products/35/ (accessed: 15.05.2020)

[3] ’RuC’ project github – URL: https://github.com/andrey-terekhov/RuC (accessed: 15.05.2020)

[4] SYSTEM V APPLICATION BINARY INTERFACE MIPS RISC Processor, 3rd Edition

[5] MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual

[6] QEMU official site – URL: https://www.qemu.org/ (accessed: 15.05.2020)

[7] GCC official site – URL: https://gcc.gnu.org/ (accessed: 15.05.2020)

[8] LLVM official site – URL: https://llvm.org/ (accessed: 15.05.2020)

[9] ’RuC’ project github, section wiki – URL: https://github.com/andrey-terekhov/RuC/wiki (accessed: 15.05.2020)

[10] A. N. Terekhov, M. A. Terekhov, “RuC project for education and reliable software systems development” ISSN 0321-2653 IZVESTIYA VUZOV. SEVERO-KAVKAZSKIYREGION. TECHNICAL SCIENCE, 2017

Application 1

```
void main()
{
    float a = 5.1, b = 6.3, c = 2.3;
    if (c > a && b < 5.3 || 5.2 >= a)
        c += (a + b) * 3.2 - 6.7 / c;
    printf("%f\n", c);
}
```

Application 2

```
        .file 1 "tests/mips/float.c"
        .section .mdebug.abi32
        .previous
        .nan    legacy
        .module fp=xx
        .module nooddspreg
        .abicalls
        .option pic0
        .text
        .align 2

        .globl  main
        .ent    main
        .type   main, @function
main:
        move $fp, $sp
        addi $fp, $fp, -4
        sw $ra, 0($fp)
        li $t0, 268500992
        sw $t0, -8060($gp)
        j NEXT2
        nop
FUNC2:
        addi $fp, $fp, -96
        sw $sp, 20($fp)
        move $sp, $fp
        sw $ra, 16($sp)
        li.s $f4, 5.100000
        swc1 $f4, 80($sp)
        li.s $f4, 6.300000
        swc1 $f4, 84($sp)
        li.s $f4, 2.300000
        swc1 $f4, 88($sp)
        lwc1 $f20, 88($sp)
        lwc1 $f4, 80($sp)
        c.le.s $f20, $f4
        bc1t ELSE4
        lwc1 $f20, 84($sp)
        li.s $f4, 5.300000
        c.lt.s $f20, $f4
        bc1t ELSE3
ELSE4:
        li.s $f20, 5.200000
        lwc1 $f4, 80($sp)
        c.lt.s $f20, $f4
        bc1t ELSE1
ELSE3:
        lwc1 $f20, 80($sp)
        lwc1 $f4, 84($sp)
        add.s $f20, $f20, $f4
        li.s $f4, 3.200000
        mul.s $f20, $f20, $f4
        li.s $f22, 6.700000
        lwc1 $f4, 88($sp)
        div.s $f22, $f22, $f4
        sub.s $f20, $f20, $f22
        lwc1 $f6, 88($sp)
        add.s $f4, $f6, $f20
        swc1 $f4, 88($sp)
ELSE1:
        .rdata
        .align 2
STRING1:
        .ascii "%f\n\0"
        .text
        .align 2
        lwc1 $f4, 88($sp)
        cvt.d.s $f4,$f4
        mfc1    $5,$f4
        mfhc1   $6,$f4
        lui $t1, %hi(STRING1)
        addiu $a0, $t1, %lo(STRING1)
        jal printf
        nop
        j FUNCEND2
        nop

FUNCEND2:
        lw $ra, 16($sp)
        addi $fp, $sp, 96
        lw $sp, 20($sp)
        jr $ra
        nop
```

```
NEXT2:
        jal FUNC2
        nop
        lw $ra, -4($sp)
        jr $ra
        nop
        .end    main
        .size   main, .-main
```