Arkhipov I.S. Code generation for floating-point arithmetic in architecture MIPS. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 3, 2020, pp. 49-56.

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

Keywords: code generation; translator; floating-point arithmetic; MIPS

For citation: Arkhipov I.S. Code generation for floating-point arithmetic in architecture MIPS. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 3, 2020, pp. 49-56. DOI: 10.15514/ISPRAS–2020–32(3)–4

Генерация кодов для вещественной арифметики в архитектуре MIPS

И.С. Архипов, ORCID: 0000-0002-8566-1654 <arkhipov.iv99@mail.ru>
Санкт-Петербургский государственный университет, Россия, 199034, Санкт-Петербург, Университетская наб., д. 7–9

Abstract. Эта статья посвящена генерации кода для вещественной арифметики в архитектуре MIPS. Эта работа является частью проекта «RuC». В ней рассматривается толька генерация кодов для операций с числами с плавающей запятой. В статье не рассматриваются лексический, синтаксический и видозависимый анализ

Ключевые слова: кодогенерация; транслятор; арифметика чисел с плавающей запятой; MIPS

Для цитирования: Архипов И.С. Генерация кодов для вещественной арифметики в архитектуре MIPS. Труды ИСП РАН, том 32, вып. 3, 2020 г., стр. 49-56 (на английском языке). DOI: 10.15514/ISPRAS–2020–32(3)–4

1. 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 of the 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.

2. 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 floating-point numbers. At the moment, this project has a customer, which is an additional evidence of the relevance of this work.

3. Problem statement

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

• to implement code generation for operations with floating-point numbers in RuC using the query and response technique;
• to implement printing floating-point numbers to the console;
• to 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.

4. 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 floating-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 floating-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 floating-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.

5. Related Work
In the process, we also looked at the code generated by the gcc compiler [7] and compared it with our own code. 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 MIPS32 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.

6. Implementation
6.1 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 floating-point numbers, namely the following:

- TConstf – floating-point constant;
- TIdenttovalf – take the value of an identifier;
- «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 – /;

51

52
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.

6.5 «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.

6.6 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.

6.7 Print
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.

7. Evaluation
After implementing code generation for operations with floating-point numbers and printing floating-point numbers tests were prepared. Tests have been prepared that demonstrate the code generation for each operation separately and for complex expressions with floating-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 usefulness of this work can be demonstrated.

For this purpose:
- emulator qemu [6] was installed;
- Baikal-T1 was bought;

...
Arkhipov I.S. Code generation for floating-point arithmetic in architecture MIPS. *Trudy ISP RAN/Proc. ISP RAS* vol. 32, issue 3, 2020, pp. 49-56.

Information about authors / Информация об авторах
Ivan Sergeevich ARKHIPOV – undergraduate student in the Department of System Programming. Research interests: MIPS architecture, compilers, code generation.

Иван Сергеевич Архипов – студент бакалавриата кафедры системного программирования. Научные интересы: архитектура MIPS, компиляторы, генерация кода.