Санкт-Петербургский государственный университет

Кафедра системного программирования
Математическое обеспечение и администрирование
информационных систем

Архипов Иван Сергеевич

Генерация кодов для вещественной арифметики в архитектуре MIPS

Курсовая работа

Научный руководитель:
д. ф.-м. н. Терехов А. Н.

Санкт-Петербург 2020

Содержание

1 Введение

В RISC и CISC архитектурах, в отличие от архитектур HLL и виртуальных машин, действия языка высокого уровня можно выразить разными способами. Имеется много регистров для работы с данными, что создаёт большую вариативность оптимальной генерации кода. Поэтому кодогенерация в данных архитектурах является довольно сложной задачей.

Для работы с такими архитектурами на математико-механическом факультете СПБГУ была разработана техника запросов и ответов [1]: сверху построенного дерева разбора поступают запросы на значения, а снизу ответы – формы представления значения (регистр, память, константа). Кроме того, есть определённые соглашения о связях. Например, в архитектуре MIPS32 значения параметров функций должны находиться в одних определённых регистрах, а значения функций – в других определённых регистрах. Есть сохраняемые регистры, сохранность которых должна быть обеспечена при вызовах функций, а есть несохраняемые регистры. Например, левый операнд бинарной формулы должен быть представлен в сохраняемом регистре, если в правом операнде есть вызовы или вырезки, для которых применяются те же правила, что и для функций, иначе левый операнд может быть представлен и в несохраняемом регистре. Определить, насколько сложен правый операнд – задача оптимизирующего просмотра.

Данная работа имеет практическое применение: архитектура MIPS32 является базовой архитектурой одной из отечественных ЭВМ Байкал-Т1.

Задачей данной курсовой работы является генерация кодов для арифметики чисел с плавающей запятой в кодах MIPS32 с помощью техники запросов и ответов.

2 Актуальность

Разработка отечественного транслятора является актуальной задачей, так как многие отрасли промышленности требуют оте-

чественного программного обеспечения, чтобы избежать наличия “черных ходов” в зарубежном программном обеспечении. Написание собственного транслятора - это сложная задача. Эта работа является частью проекта RuC [3] и посвящена только генерации кода операций с вещественными числами. На данный момент у этого проекта есть заказчик, что является дополнительным доказательством актуальности данной работы.

3 Цели и задачи

3.1 Цель работы

Цель курсовой работы – реализация вещественной арифметики в трансляторе RuC техникой запросов и ответов для архитектуры MIPS32 на базе российской ЭВМ Байкал-T1.

3.2 Поставленные задачи

Для выполнения обозначенной цели были поставлены следующие задачи:

- Изучить архитектуру процессора MIPS32
- Изучить принцип работы транслятора RuC
- Реализовать кодогенерацию для операций с вещественными числами в RuC с помощью техники запросов и ответов
- Реализовать печать вещественных чисел в консоль
- Подготовить тесты и протестировать реализованную кодогенерацию
- На основании результатов тестирования при необходимости внести корректировки в кодогенерацию

4 Обзор аналогов

В процессе работы мы также изучили код, генерируемый компилятором gcc [6], и сравнили его с нашим собственным. Конечно, компилятор gcc уже реализовал множество оптимизаций, что делает сгенерированный им код лучше. На данный момент в RuC нет никаких оптимизаций, связанных с арифметическими операциями. Оптимизация – это следующий этап развития RuC и тема для дальнейших работ.

Если сравнить код, сгенерированный RuC, с неоптимизированным кодом, сгенерированным gcc, то можно увидеть, что RuC использует больше временных регистров, чем gcc, для промежуточных вычислений. Этот подход ближе к соглашению о связях в архитектуре MIPS32.

5 Подход к реализации

Поскольку эта работа является частью проекта RuC, то для достижения поставленных целей используются те же идеи, что и в RuC. Генератор кода будет просматривать дерево синтаксического разбора программы и генерировать код на основе расположенных в нем лексем.

Необходимо описать принципы работы RuC в целом. Транслятор RuC имеет двухпросмотровую структуру – на первом просмотре работает сканер (лексический анализатор), видонезависимый анализатор (парсер) и видозависимый анализатор. Результатом первого прохода является дерево синтаксического разбора. Это дерево подается на вход второму просмотру (кодогенератор), который выдает программу в кодах архитектуры MIPS32. Более подробную информацию о RuC можно найти в разделе wiki проекта 'RuC' на github [4] и в следующей статье [5].

Также стоит отметить несколько общих решений, принятых в

ходе работы:

- реализация использования регистров вручную без использования LLVM [7], во-первых, для возможности поддержки RuC, и во-вторых, для гарантии отсутсвия вредоносного кода, так как ввиду огромного количества кода в LLVM затруднительно проверить, например, отсутствие "чёрных ходов"

- реализация арифметики для чисел одинарной, а не двойной точности, так как, во-первых, на данный момент нет необходимости вычислений с двойной точностью на ЭВМ Байкал-Т1, и во-вторых, ЭВМ Байкал-Т1 (другое название BE-T1000) имеет 2 32-битных процессорных ядра P5600 архитектуры MIPS32 r5 [2], что делает её неприспособленной для вычислений с двойной точноттью. Например, из-за 32-разрядности для загрузки из памяти числа двойной точности понадобится две команды, а не одна

- обработка запросов только вида регистр-регистр, так как для вещественных значений нет команд с непосредсвенным операндом, а работа с памятью представлена только двумя командами: загрузка и выгрузка.

RuC имеет свою собственную виртуальную машину, поэтому можно было бы генерировать ассемблерный код следующим образом: сначала генерировать код в кодах виртуальной машины, а затем переводить каждую инструкцию виртуальной машины в ассемблерный код MIPS32. Было принято решение отказаться от этого подхода, потому что он генерировал большой код, который трудно оптимизировать в будущем.

6 Реализация

6.1 Лексемы дерева синтаксического разбора

Как описано выше, кодогенератор просматривает лексемы из дерева синтаксического разбора. Нас интересуют только лексемы, описывающие операции с числами плавающей точки, а именно следующие:

- TConstf – вещественная константа
- TIdenttovalf – взять значение по индентификатору
- Лексемы “унарных” арифметических операций:
  - ASSR – =
  - PLUSASSR – +=
  - MINUSASSR – -=
  - MULTASSR – *=
  - DIVASSR – /=
  - INCR – инкремент
  - POSTINCR – постинкремент
  - DECR – декремент
  - POSTDECR – постдекремент
- Лексемы “бинарных” арифметических операций:
  - LPLUSR – +
  - LMINUSR – -
  - LMULTR – *
  - LDIVR – /

Обработка каждого типа лексем будет показана ниже.

6.2 Техника запросов и ответов

Прежде чем описывать обработку лексем, необходимо описать технику запросов и ответов. Сверху построенного дерева разбора

поступают запросы на значения, а снизу ответы – формы представления значения. Существует несколько типов запросов, нас интересуют следующие:

- BREG – загрузить результат в регистр “breg”. “breg” это глобальная переменная в трансляторе, которая содержит номер регистра.
- BREGF – запрос на левый операнд, можно получить ответ. Ответы описаны ниже.
- BF – свободный запрос на правый операнд.

  Тип запроса содержится в глобальной переменной “mbox”.
  Типы ответов:

- AREG – результат в регистре “areg”. “areg” это глобальная переменная в трансляторе, которая содержит номер регистра.
- AMEM – результат в памяти. Глобальная переменная “adispl” содержит смещение и глобальная переменная “areg” содержит регистр.
- CONST – результат это константа.

  Тип ответа содержится в глобальной переменной “manst”.

6.3 Лексема TConstf

Эта лексема означает, что был получен ответ в виде константы. После этой лексемы в дереве появляется значение константы. В зависимости от типа запроса, мы можем получить регистр для ввода значения константы (“breg”). Если мы не получаем регистр, то значение константы помещается во временный регистр $f4 с помощью псевдо-инструкции li.s. О вещественных регистрах написано в [8]. Тип ответа – AREG.

6.4 Лексема TIdenttovalf

Эта лексема означает, что значение переменной должно быть загружено в регистр. Если это регистровая переменная, то при получении запроса BREG или BREGF необходимо переместить ее в регистр “breg”. В противном случае мы должны поместить значение этой переменной из памяти в регистр “breg” или в регистр $f4 инструкцией lwc1 [9].

6.5 Лексемы “унарных” арифметических операций

Эти операции называются “унарными”, потому что при их обработке необходимо сделать запрос только на правый операнд, а левый операнд уже известен. Левый операнд может быть уже в регистре, если он является регистровой переменной или в памяти. Если он есть в памяти, то его нужно поместить в регистр. Должен быть выдан только запрос на регистр на правый операнд, так как в архитектуре MIPS32 нет операций сложения, вычитания, умножения и деления для чисел с плавающей запятой с непосредственным операндом. Таким образом, левый и правый операнды должны быть в регистрах.

После этого необходимо выполнить инструкцию (сложение, вычитание, умножение или деление). Затем, если переменная находится в памяти, в памяти сохраняется новое значение переменной. В “areg” ставится регистр левого операнда. Тип ответа – AREG.

Стоит отметить, что операция деления выполняется, как и все остальные, одной командой, в отличие от аналогичной операции с целыми числами.

6.6 Лексемы “бинарных” арифметических операций

“Бинарные” операции отличаются от “унарных” тем, что перед выполнением операции необходимо выдать запрос как на левый, так и на правый операнды. В отличие от аналогичной операции с целыми числами для выполнения операций с числами с плавающей запятой левый и правый операнды должны находиться в регистрах. Поэтому для левого и правого операндов должен быть выдан только регистровый запрос.

После получения значений левого и правого операндов в регистрах исполняется инструкция. Этот этап выполняется как для унарных операций. Тип ответа – AREG.

6.7 Печать

Для просмотра результатов генерации кода необходимо реализовать вызов функции printf. Во-первых, генерируется строка в сегменте данных. Строка находится в дереве синтаксического разбора после лексемы TString. Затем снова начинается текстовый сегмент. Адрес строки помещается в регистр $a0. После этого создается регистровый запрос на второй операнд. Если этот операнд является целым числом или символом, то операнд помещается в регистр $a1, и вызывается printf. Если этот операнд является числом с плавающей запятой, то из-за соглашений mips мы должны преобразовать число с плавающей точкой одиночной точности в число двойной точности. После этого вызывается printf.

Если printf имеет более одного аргумента, то строка делится на несколько частей, и для каждой части выполняется printf.

7 Тестирование

После реализации генерации кода для операций с числами с плавающей точкой и печати чисел с плавающей точкой были подготовлены тесты. Были подготовлены тесты, демонстрирующие генерацию кода для каждой операции отдельно, и тесты для проверки сложных выражений с операциями с числами с плавающей точкой. Например, RuC транслирует программу в приложении 1 в ассемблерный код в приложении 2.

Важно отметить, что поставленная цель считается достигнутой только тогда, когда сгенерированный код успешно собран и выполнен на плате Байкале-T1. Это важно, поскольку мы можем думать, что генерация кода правильна, но на самом деле она не работает. Также таким образом можно продемонстрировать успешность этой работы.

Для этой цели:

- установлен эмулятор qemu [10];
- куплена плата Байкал-T1;
- Байкал-T1 соединён с ноутбуком.

После этого подготовленные тесты для арифметических операций сначала тестировались на эмуляторе, а затем на Байкале-T1. Тестирование прошло успешно, так что можно считать, что цель была достигнута. Тесты можно найти в [3] в ветке mips.

8 Заключение

В данной работе решается задача генерации кода в кодах MIPS32 арифметических операций с числами с плавающей запятой. Были рассмотрены различные подходы к генерации кода, и один из них был реализован – прямая генерация кода.

Новизна данной работы заключается в том, что данная работа является частью проекта RuC, первого русского транслятора,

модифицировавшего язык Си в пользу безопасности программирования.

В ходе работы были получены важные результаты, свидетельствующие о применимости результатов данной работы на практике. Прямая генерация кода была реализована в кодах MIPS32 для арифметических операций с плавающей запятой. Сгенерированный код был запущен на плате Байкал-T1.

Эта работа имеет много возможностей для дальнейших исследований. Проект RuC еще не завершен, и некоторые структуры языка Си еще не реализованы. Оптимизация генерируемого кода также является большой областью исследований. Также необходимо реализовать компоновщик. Есть еще много направлений исследований.

9 Список литературы

[1] Алгол 68. Методы реализации. Балуев А.Н.; Братчиков И.Л.; Гиндыш И.Б.; Крупко Н.А.; Цейтин Г.С.; Терехов А.Н.; (всего, 12 чел.). Издательство Санкт-Петербургского университета, 1976.

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

[3] Github проекта RuC – URL: `https://github.com/andrey-terekhov/RuC` (accessed: 15.05.2020)

[4] Github проекта RuC, раздел wiki – URL: `https://github.com/andrey-terekhov/RuC/wiki` (accessed: 15.05.2020)

[5] Проект РуСи для обучения и создания высоконадежных программных систем. Терехов А. Н.; Терехов М. А. Известия выс-

ших учебных заведений. Северо-Кавказский регион. Технические науки. 2017.

[6] Официальный сайт GCC – URL: https://gcc.gnu.org/ (accessed: 15.05.2020)

[7] Официальный сайт LLVM – URL: https://llvm.org/ (accessed: 15.05.2020)

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

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

[10] Официальный сайт QEMU – URL: https://www.qemu.org/ (accessed: 15.05.2020)

Приложение 1

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

Приложение 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 $f20, 80($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)
        .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
```