Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Архитектура вычислительных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

Добавление новых ассемблерных команд в компилятор GCC для RISC-V

Студент: гр. 853504
Шепшук Н. В.
Руководитель: Леченко А. В.

Минск 2020

Содержание

Архитектура RISC-V

Расширение набора команд, увеличение числа способов адресации, введение сложных команд сопровождаются увеличением длины кода команды, в первую очередь кода операции, что может приводить к использованию “расширяющегося кода операции”, увеличению числа форматов команд. Это вызывает усложнение и замедления процесса дешифрации кода операции и других процедур обработки команд. Возрастающая сложность процедур обработки команд заставляет прибегать к микропрограммному управлению устройствами с управляющей памятью вместо более быстродействующего устройства управления с “жесткой” логикой.

Сложный процессор с микропрограммным управлении трудно реализовать на одном кристалле, а это ведет к увеличению длины межмодульных связей. Таким образом устроены большинство процессоров с полным набором команд.

Напротив, при сокращении количества команд до некоторого оптимального значения, можно сократить длину команд и упростить управляющее устройство микропроцессора. Поэтому при проектировании структуры микропроцессора выделилось два направления в отношении набора системы команд:

- RISC – Reduced Instruction Set Computer.
- CISC – Complicated Instruction Set Computer.

Характерные особенности RISC-процессоров:

1. Одинаковая длина команд;
2. Использование большого количество регистров;
3. Сокращенный набор команд – 50-100 команд;
4. Простые способы адресации памяти;
5. Отсутствие совмещенной операции чтения/записи с обработкой данных;
6. Необходимость соответствующей компиляции программ для повышения эффективности.

Назначение новой архитектуры

Основополагающим принципом при разработке RISC-V стало создание такой архитектуры команд, которая будет подходить практически для любого вычислительного устройства. Для достижения этой задачи требуется выполнение следующих условий:

1. Архитектура RISC-V не должна быть “пере-оптимизирована” для работы с конкретной микроархитектурой или устройством.
2. Архитектура должна быть открытой и свободной для имплементации (с открытым исходным кодом). Хоть преимущества открытого исходного кода и неоднозначны, однако это позволит сообществу вносить существенный вклад в разработку инструментов (компиляторы, ассемблеры и т.д.) для различных платформ.

Разработчики архитектуры RISC-V ставили перед собой следующие задачи:

- Разделить набор команд на основные команды и дополнительные расширения;
- Поддержка как 32-битных, так и 64-битных адресных пространств;
- Облегчить разработку сторонних расширений набора команд;
- Обеспечить эффективную поддержку аппаратной части, соответствующую современным стандартам.

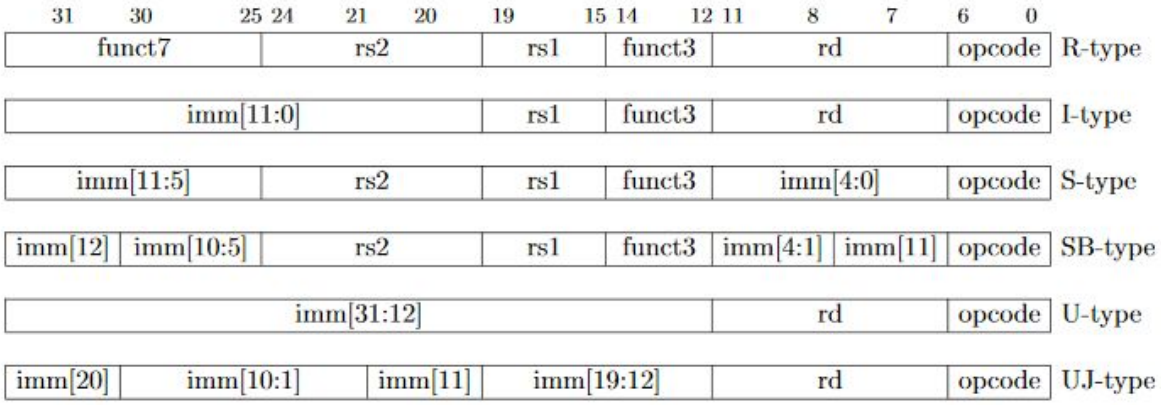
Для RISC-V было разработано 3 основных набора команд: RV32I, RV32E и RV64I. Они являются отдельными сущностями, однако их внутреннее устройство очень схоже. RV32I и RV64I в основном отличаются размером регистров и адресного пространства. RV32E – это вариация набора команд RV32I с меньшим числом регистров, которая предназначена для внедрения в малопроизводительные системы[1].

[1] “systems where every transistor counts”

Подробности реализации архитектуры RV32I

RV32I – это целочисленная 32-битная архитектура с небольшим набором команд, состоящим из 47 инструкций, чего однако вполне хватает для удовлетворения основных потребностей современных операционных систем и устройств. Архитектура RV32I состоит из 31 регистра общего назначения (`x1-x31`) размерностью 32 бита каждый. Регистр `x0` является обозначением нуля; также он может быть использован для удаления результата операции.

Инструкции архитектуры RV32I (также 32-битные) делятся на 6 форматов: 4 основных – R, I, S и U; а также 2 варианта – SB и UJ – которые идентичны форматам S и U за исключением кодировки непосредственного литерала[2]. Спецификации всех форматов инструкций представлены на иллюстрации:



Для кода операции[3] выделено 7 бит, однако базовый набор команд использует только 5 из них (два младших разряда кода инструкции из базового набора команд всегда равны двум единицам). Оставшиеся ¾ пространства кодов зарезервированы для расширений набора команд, которые значительно увеличивают число инструкций.

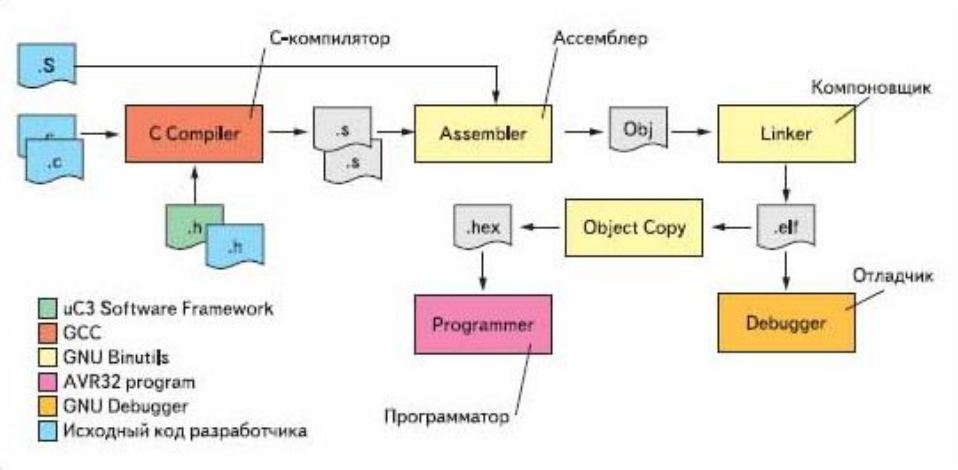
[2] immediate

[3] opcode

GNU Toolchain

Проект GNU — проект по разработке свободного программного обеспечения, является результатом сотрудничества множества отдельных проектов. Изначальной целью проекта было «разработать достаточно свободного программного обеспечения, чтобы можно было обойтись без программного обеспечения, которое не является свободным». Чтобы этого достичь, проект в 1984 году приступил к разработке операционной системы GNU. Эта цель была достигнута в 1992 году, когда последний пробел в ОС GNU — ядро системы — был заполнен сторонней разработкой, ядром Linux, которое было выпущено как свободное программное обеспечение в соответствии с лицензией GNU GPL v2.

GNU Toolchain – набор созданных в рамках проекта GNU пакетов программ, необходимых для компиляции и генерации исполняемого кода из исходных текстов.



GNU Compiler Collection

GCC – GNU Compiler Collection – набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU, является свободным программным обеспечением.

Фронтенд компилятора

Каждый фронтенд использует синтаксический анализатор[4] для создания абстрактного синтаксического дерева из исходного кода. Абстрагирование синтаксического дерева от языка исходного кода позволяет использовать один и тот же бэкенд для разных языков программирования. Изначально в GCC механизм синтаксического анализатора генерировался с помощью Bison. Теперь же все фронтенды оснащены рукописными синтаксическими анализаторами с рекурсивным спуском[5].

Бэкенд компилятора

Устройство бэкенда GCC частично обуславливается макросами препроцессора и особенностями целевой архитектуры: порядок байтов, размер слова, конвенции вызова и т.д. Внешняя часть бэкенда учитывает эти особенности при генерации RTL, так что хотя RTL и является платформо-независимым, однако сама последовательность абстрактных инструкций адаптирована под машинное описание целевой платформы.

RTL (Register Transfer Language) – это язык промежуточного представления программы, очень близкого к ассемблерному коду. Он используется для описания потока информации на уровне регистров. В GCC используется Lisp-подобная форма записи выражений RTL:

```
(set (reg:SI 140)
     (plus:SI (reg:SI 138)
              (reg:SI 139)))
```

Такое выражение означает: “просуммировать содержимое регистров 138 и 139 и сохранить результат в регистр 140”.

Машинное описание[6] бэкенда содержит RTL паттерны, ограничения для операндов инструкций и фрагменты кода для вывода скомпилированного кода. Ограничения указывают, что конкретный RTL паттерн может быть применен только (например) для определенных регистров. Чтобы бэкенд использовал конкретный фрагмент RTL, он должен подходить под один

[4] parser

[5] hand-written recursive-descent parsers

[6] machine description

(или более) RTL паттерн и удовлетворять ограничениям этого паттерна. Иначе будет невозможно конвертировать полученный RTL в машинный код.

GNU Binutils

GNU Binary Utilities, или `binutils` – это набор инструментов для создания и управления бинарными программами, объектными файлами, библиотеками или ассемблерным кодом. Включает в себя ассемблер, линкер, профилировщик и другие инструменты. GNU Assembler (`gas` или `as`) – это стандартный бэкенд для GCC.

Практическая часть

Цель работы - добавить в компилятор для RISC-V команды из расширения Crypto. В момент написания работы это расширение находится на стадии стандартизации, т.е. происходит процесс утверждения набора команд, их назначения и операционных кодов. Актуальный список инструкций с опкодами выглядит следующим образом:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

bs	00100	rs2	rs1	011	rd	0101011	sm4ed
bs	00101	rs2	rs1	011	rd	0101011	sm4ks
bs	00000	rs2	rs1	010	rd	0101011	aes32esmi
bs	00001	rs2	rs1	010	rd	0101011	aes32esi
bs	00010	rs2	rs1	010	rd	0101011	aes32dsmi
bs	00011	rs2	rs1	010	rd	0101011	aes32dsi
0000100	0	rcon	rs1	010	rd	0101011	aes64ks1i
0000101		rs2	rs1	010	rd	0101011	aes64ks2
0000110		00001	rs1	010	rd	0101011	aes64im
0000111		rs2	rs1	010	rd	0101011	aes64esm
0001000		rs2	rs1	010	rd	0101011	aes64es
0001001		rs2	rs1	010	rd	0101011	aes64dsm
0001010		rs2	rs1	010	rd	0101011	aes64ds
0000111		00000	rs1	111	rd	0101011	sha256sig0
0000111		00001	rs1	111	rd	0101011	sha256sig1
0000111		00010	rs1	111	rd	0101011	sha256sum0
0000111		00011	rs1	111	rd	0101011	sha256sum1
0000111		01000	rs1	111	rd	0101011	sm3p0
0000111		01001	rs1	111	rd	0101011	sm3p1
0001000		rs2	rs1	111	rd	0101011	sha512sig0l
0001001		rs2	rs1	111	rd	0101011	sha512sig0h
0001010		rs2	rs1	111	rd	0101011	sha512sig1l
0001011		rs2	rs1	111	rd	0101011	sha512sig1h
0001100		rs2	rs1	111	rd	0101011	sha512sum0r
0001101		rs2	rs1	111	rd	0101011	sha512sum1r
0000111		00100	rs1	111	rd	0101011	sha512sig0
0000111		00101	rs1	111	rd	0101011	sha512sig1
0000111		00110	rs1	111	rd	0101011	sha512sum0
0000111		00111	rs1	111	rd	0101011	sha512sum1
0001111		00000	01010	111	rd	0101011	pollentropy

Для работы были выбраны следующие инструкции:

```
sha256sig0
sha256sig1
sha256sum0
sha256sum1
```

Инструкции будут добавляться в открытый проект `riscv-gnu-toolchain`, который содержит в себе все компоненты тулчейна: `gcc`, `binutils` и другие. Помимо самого тулчейна нам понадобится репозиторий, содержащий опкоды инструкций `riscv-opcodes`, а также скрипт для генерации необходимых впоследствии значений.

Добавление инструкций

Работа по добавлению инструкции в компилятор может быть разделена на два основных шага:

1. Добавление инструкции непосредственно в gcc (с целью получения ассемблерного кода, включающего новую инструкцию, из языка высокого уровня).
2. Добавление инструкции в binutils, что позволит создавать исполняемый бинарный файл из ассемблерного кода, который будет содержать в себе код новой инструкции.

Первый шаг был опущен вследствие неопределенности процесса тестирования. Ограничивающим фактором здесь является момент трансляции исходного кода в промежуточные форматы, так как неизвестно, какой набор команд на языке C подойдет под RTL-паттерн новой инструкции, и следовательно спровоцирует компилятор подставить новую инструкцию в результирующий ассемблерный код.

Исходя из приведенных ограничений сразу перейдем ко второму шагу.

Первым действием здесь станет генерация значений MATCH и MASK, которые описывают операционный код инструкции. Для этого в проекте `riscv-opcodes` создадим новый файл и запишем в нем описание формата новых инструкций, что в итоге выглядит следующим образом:

```
sha256sig0  rd rs1      31..25=7 24..20=0 14..12=7 6..0=0x2B
sha256sig1  rd rs1      31..25=7 24..20=1 14..12=7 6..0=0x2B
sha256sum0  rd rs1      31..25=7 24..20=2 14..12=7 6..0=0x2B
sha256sum1  rd rs1      31..25=7 24..20=3 14..12=7 6..0=0x2B
```

Теперь передадим все файлы с опкодами в скрипт, генерирующий значения MATCH и MASK.

```
../riscv-opcodes$ cat opcodes* | ./parse-opcodes -c > ~/temp.h
```

В сгенерированном файле найдем следующие значения:

```
#define MATCH_SHA256SIG0 0xe00702b
#define MASK_SHA256SIG0  0xfff0707f
#define MATCH_SHA256SIG1 0xe10702b
#define MASK_SHA256SIG1  0xfff0707f
#define MATCH_SHA256SUM0 0xe20702b
#define MASK_SHA256SUM0  0xfff0707f
#define MATCH_SHA256SUM1 0xe30702b
#define MASK_SHA256SUM1  0xfff0707f

DECLARE_INSN(sha256sig0, MATCH_SHA256SIG0, MASK_SHA256SIG0)
DECLARE_INSN(sha256sig1, MATCH_SHA256SIG1, MASK_SHA256SIG1)
DECLARE_INSN(sha256sum0, MATCH_SHA256SUM0, MASK_SHA256SUM0)
DECLARE_INSN(sha256sum1, MATCH_SHA256SUM1, MASK_SHA256SUM1)
```

Эти значения, а также вызовы операции `DECLARE_INSN` добавим в файл `riscv-binutils/include/opcode/riscv-opc.h`, содержащий сгенерированные таким же методом значения для других инструкций.

Следующим шагом нужно добавить общее описание инструкций в массив структур `riscv-opcode riscv-opcodes[]` (файл `riscv-binutils/opcodes/riscv-opc.c`).

```
const struct riscv_opcode riscv_opcodes[] =
{
/* name,     xlen,   isa,    operands, match,           mask,
match_func, pinfo.  */
{"sha256sig0", 0, INSN_CLASS_C, "d,s", MATCH_SHA256SIG0, MASK_SHA256SIG0,
match_opcode, 0},
{"sha256sig1", 0, INSN_CLASS_C, "d,s", MATCH_SHA256SIG1, MASK_SHA256SIG1,
match_opcode, 0},
{"sha256sum0", 0, INSN_CLASS_C, "d,s", MATCH_SHA256SUM0, MASK_SHA256SUM0,
match_opcode, 0},
{"sha256sum1", 0, INSN_CLASS_C, "d,s", MATCH_SHA256SUM1, MASK_SHA256SUM1,
match_opcode, 0},
... other instructions ...
}
```

После выполнения приведенных действий можно приступить к сборке и установке gnu-toolchain. Для этого следует перейти в директорию и применить команды:

```
../riscv-gnu-toolchain$  ./configure --prefix=/opt/riscv
../riscv-gnu-toolchain$  sudo make
```

Теперь для удобства добавим директорию с исполняемыми файлами компилятора в переменную PATH:

```
$  export PATH=”/opt/riscv/bin:$PATH”
```

Проверка работоспособности метода

Для проверки работоспособности компилятора напишем простую программу на языке C с использованием новых ассемблерных инструкций через `asm volatile`.

```
#include <stdio.h>

int main() {
   int a, b;
   a = 1;

   asm volatile (
       "sha256sig0 %[z], %[x]\n\t"
       : [z] "=r" (b)
       : [x] "r"  (a)
   );

   asm volatile (
       "sha256sig1 %[z], %[x]\n\t"
       : [z] "=r" (b)
       : [x] "r"  (a)
   );

   asm volatile (
       "sha256sum0 %[z], %[x]\n\t"
       : [z] "=r" (b)
       : [x] "r"  (a)
   );
```

```
    asm volatile (
        "sha256sum1 %[z], %[x]\n\t"
        : [z] "=r" (b)
        : [x] "r"  (a)
    );
}
```

Скомпилируем исходный код программы в бинарный файл:

```
$ riscv64-unknown-elf-gcc -o test1 test1.c
```

Завершенный без ошибок процесс компиляции программы означает, что новые инструкции действительно добавились в компилятор. Проверить наличие новых инструкций в машинном коде программы можно с помощью утилиты `objdump`, которая устанавливается вместе с другими компонентами `riscv-gnu-toolchain`.

```
$ riscv64-unknown-elf-objdump -dC test1 | grep sha
```

```
(base) shepshook@shepshook-asus:~/term5/avs$ riscv64-unknown-elf-objdump -dC test1 | grep sha
   1015a:       0e07f7ab                sha256sig0      a5,a5
   10166:       0e17f7ab                sha256sig1      a5,a5
   10172:       0e27f7ab                sha256sum0      a5,a5
   1017e:       0e37f7ab                sha256sum1      a5,a5
```

Исходя из этого можно утверждать, что инструкции были добавлены в компилятор.

Вывод

В ходе курсовой работы было изучено много информации о внутреннем устройстве компиляторов в общем и gcc в частности. Первая версия gcc была выпущена в 1987 году, однако он не утратил своей актуальности. Открытый исходный код gcc и его огромное сообщество позволяет ему развиваться и по сей день. Так, например, в него постоянно добавляются новые архитектуры и расширения. В данный момент ведется стандартизация расширения Crypto для платформы RISC-V, обновляются и другие расширения.

Данная курсовая работа является входной точкой в разносторонний мир open source. В большой степени ценность этой работы заключается в полученном опыте взаимодействия с крупными проектами, над которыми трудилось целое сообщество. Внесенные в проект небольшие изменения могут стать базой для более глубокого внедрения расширения в компилятор.

Использованная литература

Waterman, Andrew. “Design of the RISC-V Instruction Set Architecture”, University of California, Berkeley, Technical Report No. UCB/EECS-2016-1. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.pdf

Asanovic, Krste; Waterman, Andrew. “The RISC-V Instruction Set Manual Volume I: User-Level ISA”, University of California, Berkeley. https://riscv.org//wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

Intel, “Intel 64 and IA-32 Architectures. Software Developer’s Manual”, Order Number: 253666-060US, September 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf

GNU Org. “GCC Documentation”. https://gcc.gnu.org/onlinedocs/gccint/Patterns.html