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 RISC-V ассемблерных команд в gcc**

БГУИР КР 1-40 04 01 ПЗ

Студент: гр. 853504

Грищенко М. М.

Руководитель: ассистент кафедры

информатики Леченко А. В.

Минск 2020

Оглавление

[1.Краткие теоретические сведения 3](#_Toc57299452)

[1.1 RISC-V и RISC-V ISA 3](#_Toc57299453)

[1.2 Наборы инструкций, опкоды и операнды 3](#_Toc57299454)

[1.3 Представление RTL 4](#_Toc57299455)

[1.4 Insns 4](#_Toc57299456)

[1.5 Как используется Machine Description 4](#_Toc57299457)

[1.6 Расширенный стандарт шифрования AES 5](#_Toc57299458)

[2. Добавление новых RISC-V ассемблерных команд в gcc 6](#_Toc57299459)

[2.1 RISC-V Opcodes 6](#_Toc57299460)

[2.2 RISC-V GNU Comiler Toolchain 7](#_Toc57299461)

[2.3 Тесты 8](#_Toc57299462)

[Список литературы 10](#_Toc57299463)

[Приложение 1 11](#_Toc57299464)

# Краткие теоретические сведения

## 1.1 RISC-V и RISC-V ISA

**RISC-V** - это архитектура набора команд (**ISA**)

**RISC-V ISA** определяется как базовое целое число **ISA**, которое должно присутствовать в любой реализации, плюс дополнительные расширения базового **ISA**. База тщательно ограничена минимальным набором инструкций, достаточным обеспечить разумную цель для компиляторов, ассемблеров, компоновщиков и операционных систем (с дополнительные операции на уровне супервизора), и поэтому предоставляет удобный набор инструментов **ISA** и программного обеспечения «Каркас», вокруг которого могут быть построены более настраиваемые **ISA** процессоров.

Каждый базовый набор целочисленных команд характеризуется шириной целочисленных регистров и соответствующий размер адресного пространства пользователя. Есть два основных варианта целого числа, *RV32I* и *RV64I*, которые предоставляют 32-битные или 64-битные адресные пространства на уровне пользователя соответственно. Аппаратные реализации и операционные системы могут предоставлять только один или оба *RV32I* и *RV64I* для пользовательских программ.

## 1.2 Наборы инструкций, опкоды и операнды

**Инструкции** декодируются как набор последовательных операций. Эти операции дают команду **ALU** и блоку управления внутри **ЦП**.

**Код операции** сообщает процессору о том, что необходимо выполнить. Каждая инструкция кода операции очень ограничена в том, что она может сказать процессору. Набор команд **ЦП** содержит коды операций, которые он будет принимать.

Есть два типа опкодов:

1. Которые сообщают схеме, какую операцию выполнить.
2. Которые вместе с некоторыми данными выполняются.

**Операнд** указывает данные, над которыми нужно действовать. Операнд также может относиться к месту в памяти, например, к регистру.

## 1.3 Представление RTL

Большинство работы транслятора выполняется в промежуточном представлении, называемом *register transfer language* (**RTL**). На этом языке команды, которые нужно выводить, описаны довольно подробно одна за другой в алгебраической форме, которая описывает то, что делает команда.

**RTL** построен на идеях списков *Lisp*. Он имеет как внутреннюю форму, состоящую из структур, которые указывают на другие структуры, так и текстовую форму, которая используется в машинном описании и в напечатанных дампах отладки. Текстовая форма использует вложенные круглые скобки, чтобы определить указатели во внутренней форме.

## 1.4 Insns

**RTL** представление кода функции - связанная в две стороны цепочка объектов, называемых "*insns*". Insns - выражения со специальными кодами, которые не используются ни для какой другой цели. Некоторые **insns** фактически являются командами; другие представляют таблицы управления для операторов "*switch*"; другие представляют метки перехода или различных видов описаний.

В дополнение к собственным специфическим данным, каждый **insn** должен иметь уникальный идентифицирующий номер, который отличает его от всех других **insns** в текущей функции (после отсроченного планирования перехода копии **insn** с одинаковыми идентифицирующими номерами могут присутствовать в нескольких местах в функции, но эти копии всегда будут идентичны и будут появляться только внутри "*sequence*"), и указатели на предыдущий и следующий **insn** для получения цепочки. Эти три поля занимают одну и ту же позицию в каждом insn, независимо от кода выражения **insn**.

## 1.5 Как используется Machine Description

В компиляторе происходят три основных преобразования:

1. Внешний интерфейс считывает исходный код и строит дерево синтаксического анализа.
2. Дерево синтаксического анализа используется для создания списка **RTL** insn на основе именованных шаблонов инструкций.
3. Список insn сопоставляется с шаблонами **RTL** для создания кода ассемблера.

Выражение **define\_insn** используется для определения шаблонов инструкций, которым могут соответствовать **insns**. Выражение *define\_insn* содержит неполное **RTL**-выражение с частями, которые нужно заполнить позже, ограничения операндов, которые ограничивают способ заполнения частей, и выходной шаблон или код C для генерации выходных данных ассемблера.

Для прохода генерации имеют значение только имена **insns** либо из именованного *define\_insn*, либо из *define\_expand*. Компилятор выберет шаблон с правильным именем и применит операнды. Имена, которые ищет компилятор, жестко запрограммированы в компиляторе - он будет игнорировать безымянные шаблоны и шаблоны с именами, о которых он не знает, и, если не предоставить именованный шаблон, который ему нужен, он прервется. Если используется *define\_insn*, данный шаблон вставляется в список insn.

Если используется *define\_insn*, данный шаблон вставляется в список **insn**. После создания списка **insn** различные проходы оптимизации преобразуют, заменяют и переупорядочивают **insns** в списке **insn**.

Наконец, **RTL** списка insn сопоставляется с шаблонами **RTL** в шаблонах *define\_insn*, и эти шаблоны используются для генерации окончательного кода сборки. С этой целью каждый именованный *define\_insn* действует так, как будто он не имеет имени, поскольку имена игнорируются.

## 1.6 Расширенный стандарт шифрования AES

Advanced Encryption Standard (**AES**) - это 128-битный блочный шифр с ключом 128/192/256 бит, определенный в стандарте *FIPS* *197*. **AES** является обязательным строительным блоком протокола безопасности *TLS* и широко используется для шифрования хранилища, аутентификации с общим секретом, генерации криптографических случайных чисел и во многих других приложениях.

По сравнению с более общими рабочими нагрузками криптографические алгоритмы, такие как **AES**, представляют собой серьезную проблему при реализации. Они включают в себя специализированные функции, требующие больших вычислительных ресурсов.

Хотя эффективность и является целью сама по себе, она также способствует обеспечению безопасности. Это связано с тем, что не следует ставить под угрозу безопасность ради удовлетворения требований эффективности. Следовательно, более эффективная реализация оставляет больше возможностей для защиты от атаки. **AES** представляет собой интересный пример сочетания безопасной и эффективной реализации.

# 2. Добавление новых RISC-V ассемблерных команд в gcc

В качестве примера добавим **AES: Round Instructions** для **RISC-V64**

* **aes64esm**rd rs1 rs2 *Round: ShiftRows, SubBytes, MixColumns*
* **aes64es** rd rs1 rs2 *Round: ShiftRows, SubBytes*
* **aes64dsm** rd rs1 rs2 *Round: InvShiftRows, InvSubBytes, InvMixColumn*
* **aes64ds** rd rs1 rs2 *Round: InvShiftRows, InvSubBytes*

Эти инструкции предназначены только для **RISC-V64**. Они реализуют *SubBytes*, *ShiftRows* и *MixColumns* преобразования **AES**. Каждая команда *round* принимает в качестве входных данных два 64-битных регистра, представляющих 128-битное состояние шифра **AES** и выводит один 64-битный результат, то есть половину следующего состояния *round*.

## 2.1 RISC-V Opcodes

* Для начала нам понадобится официальный git-репозиторий *riscv-opcodes*

$ git clone https://github.com/riscv/riscv-opcodes.git

* В riscv-opcodes/ создадим текстовый файл opcodes-crypto со следующими опкод спецификациями для **RISC-V**

aes64esm rd rs1 rs2 31..25=7 14..12=2 6..0=0x2B  
aes64es rd rs1 rs2 31..25=8 14..12=2 6..0=0x2B  
aes64dsm rd rs1 rs2 31..25=9 14..12=2 6..0=0x2B  
aes64ds rd rs1 rs2 31..25=10 14..12=2 6..0=0x2B

* Их можно взять с официального git-репозитория [*https://github.com/riscv/riscv-crypto/blob/master/tools/opcodes-crypto-scalar*](https://github.com/riscv/riscv-crypto/blob/master/tools/opcodes-crypto-scalar)
* Также изменим riscv-opcodes/Makefile для генерации только необходимых нам **MASK**, **MATCH** и **INSN** для наших будущих *crypto-команд на*

SHELL := /bin/sh  
CRYPTO\_H := ../encoding.h  
CRYPTO\_OPCODES := opcodes-crypto  
$(CRYPTO\_H): $(CRYPTO\_OPCODES) parse\_opcodes encoding.h  
 echo "/\*" > $@  
 echo " \* This file is auto-generated by running 'make $@' in" >> $@  
 echo " \* https://github.com/riscv/riscv-opcodes (`git log -1 --format="format:%h"`)" >> $@  
 echo " \*/" >> $@  
 echo >> $@  
 cat encoding.h >> $@  
 cat $(CRYPTO\_OPCODES) | python ./parse\_opcodes -c >> $@  
.PHONY : install

* Теперь спокойно собираем riscv-opcodes

$ sudo make

* После успешной сборки появляется ../encoding.h, содержащий необходимые **MATCH**, **MASK** и **INSN**

...  
#define MATCH\_AES64ESM 0xe00202b  
#define MASK\_AES64ESM 0xfe00707f  
#define MATCH\_AES64ES 0x1000202b  
#define MASK\_AES64ES 0xfe00707f  
#define MATCH\_AES64DSM 0x1200202b  
#define MASK\_AES64DSM 0xfe00707f  
#define MATCH\_AES64DS 0x1400202b  
#define MASK\_AES64DS 0xfe00707f  
...  
DECLARE\_INSN(aes64esm, MATCH\_AES64ESM, MASK\_AES64ESM)  
DECLARE\_INSN(aes64es, MATCH\_AES64ES, MASK\_AES64ES)  
DECLARE\_INSN(aes64dsm, MATCH\_AES64DSM, MASK\_AES64DSM)  
DECLARE\_INSN(aes64ds, MATCH\_AES64DS, MASK\_AES64DS)  
...

* Они понадобятся чуть позже...

## 2.2 RISC-V GNU Comiler Toolchain

* Теперь клонируем официальный git-репозиторий riscv-gnu-toolchain

$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

* Переходим в riscv-gnu-toolchain/riscv-binutils/include/opcode/riscv-opc.h и по аналогии добавляем **MATCH**, **MASK** и **INSN**, которые сгенерировали ранее в ../encoding.h
* Переходим в riscv-gnu-toolchain/riscv-binutils/opcodes/riscv-opc.c и по аналогии добавляем

{"aes64esm", 0, INSN\_CLASS\_C, "d,s,t", MATCH\_AES64ESM, MASK\_AES64ESM, match\_opcode, 0 },  
{"aes64es", 0, INSN\_CLASS\_C, "d,s,t", MATCH\_AES64ES, MASK\_AES64ES, match\_opcode, 0 },  
{"aes64dsm", 0, INSN\_CLASS\_C, "d,s,t", MATCH\_AES64DSM, MASK\_AES64DSM, match\_opcode, 0 },  
{"aes64ds", 0, INSN\_CLASS\_C, "d,s,t", MATCH\_AES64DS, MASK\_AES64DS, match\_opcode, 0 },

* Теперь надо собрать наш riscv-gnu-toolchain. Для этого могут понадобиться некоторые стандартные пакеты.
* Для *Ubuntu* использовал

$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev

* Для других дистрибутивов нужные пакеты можно найти на git-репозитории https://github.com/riscv/riscv-gnu-toolchain
* Укажем путь, и начнём сборку

$ ./configure --prefix=/opt/riscv  
$ sudo make

* По умолчанию используется RISC-V64 и Newlib cross-compiler

## 2.3 Тесты

* После успешной установки riscv-gnu-toolchain настроим пути

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

* Для самого простого теста создадим aes64esm-test.c

#include <stdio.h>  
int main() {  
 uint64\_t a = 1;  
 uint64\_t b = 2;  
 uint64\_t c;  
 asm volatile("aes64esm %[d], %[s], %[t]\n\t" : [d] "=r" (c) : [s] "r" (a), [t] "r" (b));  
}

* По аналогии протестируем и остальные команды: aes64ds, aes64es, aes64dsm
* Теперь попробуем скомпилировать

$ riscv64-unknown-elf-gcc -o aes64esm-obj aes64esm-test.c

* Если нет ошибки "unrecognized opcode", то riscv-gnu-toolchain был собран правильно
* Протестируем через **object dump** то, что инструкция была использована

$ riscv64-unknown-elf-objdump -dC aes64esm-obj > aes64esm.dump  
$ grep aes64esm aes64esm.dump

* 10164: oee7a7ab aes64esm a5,a5,a4

# Список литературы

1. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197, 2001.
2. RISC-V bit manipulation extension draft proposal, 2019.
3. The RISC-V instruction set manual. Technical Report Volume I: UserLevel ISA (Version 20190608-Base-Ratified), 2019.
4. The RISC-V instruction set manual. Technical Report Volume II: Privileged Architecture (Version 20190608-Priv-MSU-Ratified), 2019.
5. K. Stoffelen. Efficient cryptography on the RISC-V architecture, 2016.
6. S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In Cryptographic Hardware and Embedded Systems (CHES), LNCS 4249. Springer-Verlag, 2006.
7. A. Waterman. Design of the RISC-V Instruction Set Architecture. PhD thesis, University of California at Berkeley, 2016.
8. Markku-Juhani O. Saarinen. A Lightweight ISA Extension for AES and SM4, 2020.
9. Ben Marshall. RISC-V Cryptographic Extension Proposals, 2020.
10. Clifford Wolf. RISC-V Bitmanip Extension, 2019.

# Приложение 1

* Сам проект и инструкцию по сборке можно найти на <https://github.com/MaxGrishenko/CourseWorkABC>
* Запросить доступ на приватный репозиторий можно через почту

“max.grishenko322@gmail.com”