**Вопросы создания системного или инструментального программного обеспечения для софт-ядер архитектуры RISC-V**

Интеграция с FPGA проект софт-процессора с архитектурой RISC-V позволяет использовать при разработке программного обеспечения для него любой удобный инструментарий, особенно тот, в котором есть возможность указать, какие из наборов команд должны быть учтены при компиляции.

При работе с ядрами отвечающими стандартным популярным расширениям (все или любое из набора G) особых проблем наблюдаться не должно. Основное что необходимо соблюдать это распределение памяти, адресные пространства(как минимум адрес начала исполнимого кода – где-то адресация ведется с нуля, где-то с 0x40000000 и пр.).

Проблемы начинаются при модификации системы команд. Компилятор должен как-то узнать, что он система команд изменилась, а в случае с языками высокого уровня еще и знать когда и куда именно вставлять нестандартные инструкции.

Путей, как обычно, несколько – модификация имеющихся инструментов, написание своего инструментария.

Прямым способом является расширения существующих инструментов, добавляя новые инструкции напрямую. Например, можно модифицировать инструменты из набора GNU. Данный подход позволяет использовать довольно мощный инструментарий, проверенный годами и разрабатываемый большим количество разработчиков, но требующий большого количества знаний спецификации содержимого кода и большого количества времени на компиляцию данного инструментария из-за огромных габаритов.

Альтернативным вариантом является разработка собственного ассемблера или компилятора высокоуровнего языка, способного создавать поддерживаемые процессором объектные файлы. Данный подход является наиболее гибким, элегантным и небольшим, но трудозатратным в реализации (отладка, тестирование, необходимость развития). С вариантом собственной реализации ассемблера все достаточно просто (относительно, конечно) – уже есть ряд проектов с открытым кодом – в частности на языках Lisp, Python.

Преимущество интерпретируемых языков в данном случае в том, что инструментарий написанный на них будет работать на любой платформе на которую портирован сам интерпретатор, плюс это достаточно легое для модификаций решение с «низким порогом вхождения» (в отличие от С-инструментария с их make- и прочими специфическими зависимостями).

**Путь модификации GNU-инструментария**

В Сети относительно подробно раскрывается только методика модификации RISC-V GNU/GCC toolchain[1] только под Linux-системами[2-3]. На вид выглядит шаги простые, но отзывы на их эффективность несколько разные.

Надо клонировать модификации RISC-V GNU/GCC toolchain с субмодулями (из первых минусов – размер скачанного будет приличным – более 7 ГБ.

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

$ git submodule update --init –recursive

Установить необходимые компоненты для набора инструментов RISC-V GNU/GCC (Ubuntu) :

$ 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

Собрать кросс-компилятор RISC-V для GNU/GCC Newlib:

./configure --prefix=<installation\_path>

Make

Клонируются оригинальные опкоды RISC-V:

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

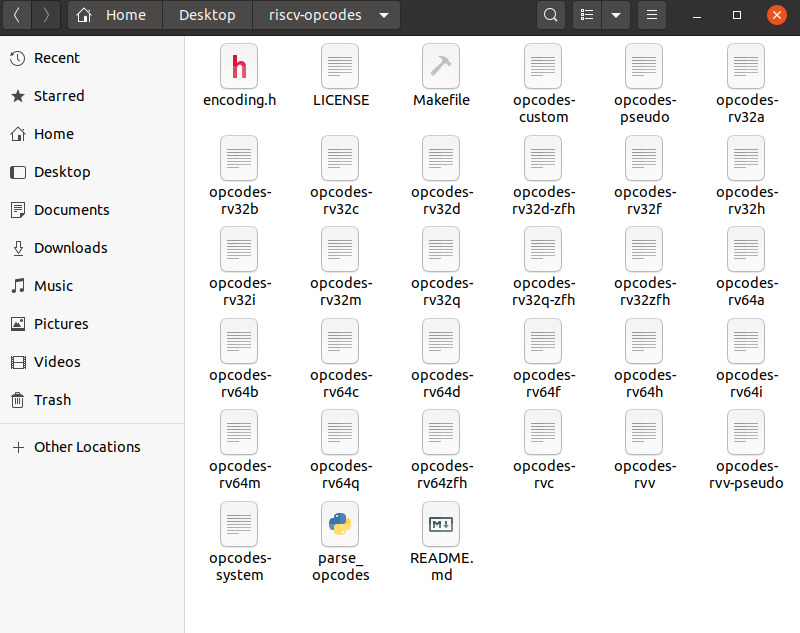
Потом надо добавить свою инструкцию – в примере – вычисление остатка от деления.

mod r1, r2, r3

Semantics:

R[r1] = R[r2] % R[r3]

В корневой папке инструмента RISC-V Opcodes находятся различные файлы, каждый из которых относится к определенному набору RISC-V.



(<https://hsandid.github.io/images/posts/risc-v-custom/risc-v-custom-6.jpeg>)

Необходимо добавить свою инструкцию в один из наборов (в один из файлов opcodes-xxx).:

mod rd rs1 rs2 31..25=1 14..12=0 6..2=0x1A 1..0=3

Потом надо запустить следующую команду для генерации констант MATCH и MASK:

cat <opcodes-file-you-modified> | ./parse-opcodes -c > instructionInfo.h

В файле  instructionInfo.h должна появиться запись:

#define MATCH\_MOD 0x200006b

#define MASK\_MOD 0xfe00707f

...

DECLARE\_INSN(mod, MATCH\_MOD, MASK\_MOD)

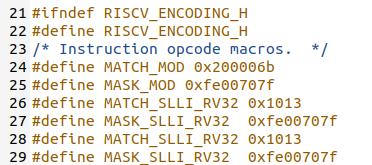
Шестнадцатеричное значение MASK является 32-разрядным шестнадцатеричным значением, которое помещает "маску", указывающую, где находятся все поля, связанные с типом инструкции. Например, mod - это инструкция типа R со следующей шестнадцатеричной маской : 0xfe00707f Соответствующее шестнадцатеричное значение - это, по сути, 32-разрядный шестнадцатеричный код, в котором значения кодов операций/funct3/funct7/.

Например, mod имеет шестнадцатеричное соответствие 0x200006b, что означает, что в двоичном формате funct7 = 0b0010000; funct3 = 0b000 и код операции = 0b1101011.

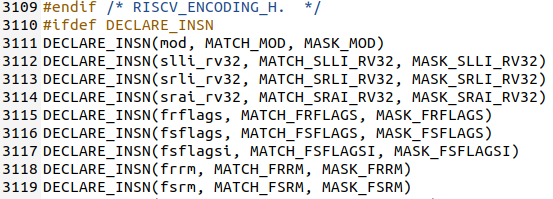
Поскольку модификация самого компилятора GCC является отнюдь не тривиальной задачей, изменению подвергнется только RISC-V GNU/GCC Binutils, чтобы добавить дополнительную инструкцию.

В файле riscv-gnu-toolchain/riscv-binutils-gdb/include/opcode/riscv-opc.h надо добавить элементы #define и DECLARE\_INSN(), которые были сгенерированы ранее для новой инструкции в соответствующие блоки.

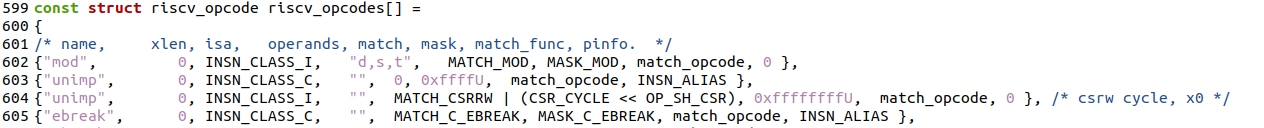
#define



DECLARE\_INSN() :



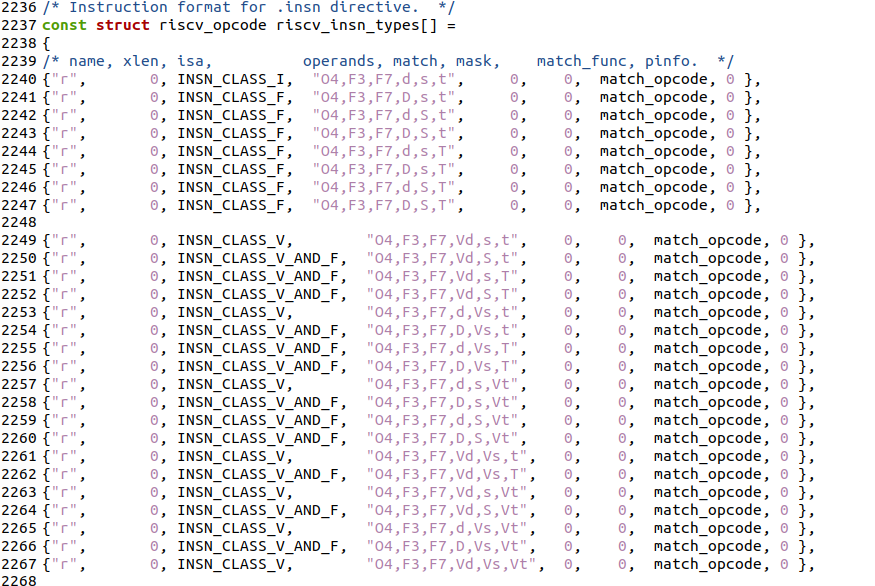
Далее в файле riscv-gnu-toolchain/riscv-binutils-gdb/opcodes/riscv-opc.c нужно объявить новую инструкцию, используя определенный набор параметров



name: название желаемой инструкции

xlen: ширина целочисленного регистра в битах (0,32,64)

ISA : тип используемой инструкции. Вы можете найти более подробную информацию о различных директивах инструкции в том же файле, если прокрутите страницу вниз.



operands: Функция, обрабатывающая операнды, доступна в riscv-binutils/gas/config/tc-recv.c. Ознакомьтесь с ней, чтобы узнать, какие символы должны использоваться для представления ваших операндов.



Это основной оператор switch функции синтаксического анализа riscv\_ip(), который обрабатывает различные операнды. Ориентировочное положение в файле – в районе строки с номером 2000.

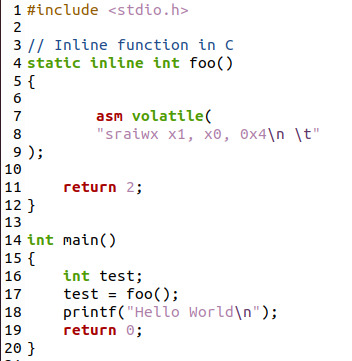
**match:** полученное ранее значение MATCH

**mask:** аналогично для MASK

**match\_opcode:** указатель на функцию, которую вы хотели бы использовать, чтобы "определить", какой код операции вы используете

**pinfo:** Используется для указания некоторого специального поведения. В основном используется с инструкциями перехода и сжатыми инструкциями RISC-V. Значение равно 0, если нет специального поведения.

Аналогично первых шагам процесса необходимо пересобрать RISC-V GNU/GCC Compiler Toolchain



Теперь можно использовать свои пользовательские инструкции в своих программах. Чтобы иметь возможность вызывать свои пользовательские инструкции, следует использовать ассемблерные вставки.

Наконец, инструкции RISC-V собираются в машинный код. Можно использовать obj-dump для проверки сгенерированных двоичных файлов RISC-V и проверить, где вызывается новая инструкция.

В общем – такой путь модификации инструментария не радует – 7 гигабайт трафика, около часа процессорного времени на перекомпиляцию за сомнительное удовольствие работать с ассемблерными вставками.

Конечно, на самом деле настроить компилятор языка высокого уровня на работу с «крафтовыми» командами достаточно комплексная задача, особенно если требуется оптимизированный код.

**Open source ассемблеры**

Ситуация с ассемблерами с открытым исходным кодом несколько проще, также как и в случае с gcc-risc-v имеются исходные тексты компиляторов. Случае с исходниками на Python разобраться с исходными текстами и правилами модификации исходной версии ассемблерного компилятора относительно не сложно.

Относительно простой переход к языкам высокого уровня в проектах FPGA с софт-процессорами возможен для языков с т.н. «виртуальными машинами» (ВМ) - наподобие упомянутого уже LISP, экзотического в современном мире Forth или Java. Связано это с тем, что реализуется на низком уровне ядро ВМ (возможно даже с учетом принятого в RISC-V ABI), а далее просто идет ее развитие в требуемом направлении.

Для относительно широкого применения, снижения «входного порога», а также для повторного использования кода и применения наработок кода, целесообразнее перейти на ЯВУ, отличный от Форта (отчасти это связано с суевериями и заблуждениями майн-стрим программистов относительно сложностей данного языка и читабельности его кода (к слову, один из авторов данной работы аналогичного мнения о С-подобных языках)).

Интересен вариант с адаптацией некоторого ограниченного подмножества языка Python к RISC-V. Python - высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, поддерживающий несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное, плюс – «его преподают даже в детском садике».

Некоторое время назад был анонсирован компилятор Uzh - небольшой компилятор для программного процессора FPGA Zmey. Uzh – это также статически скомпилированное подмножество Python, основывается на перспективном инструментарии raddsl (набор инструментов для быстрого создания прототипов DSL-компиляторов).

Компилятор Uzh принимает код на языке Python и формирует на выходе загрузочный поток для инициации памяти программ и памяти данных процессора (да, практически любой язык программирования можно использовать и в варианте интерпретатора, и в варианте компилятора с той или иной степенью комфорта), ключевым моментом является то, что на этапе компиляции доступен весь функционал языка.

Для установки компилятора Uzh достаточно скачать его архив и распаковать в любую удобную папку (лучше придерживаться общих рекомендаций для специализированного программного обеспечения – избегать путей, содержащих кириллицу и пробелы). Также необходимо скачать и распаковать в основную папку компилятора инструментарий raddsl.

Папка test компилятора содержит примеры программ для софт-процессора, папка src – исходные тексты элементов компилятора. Для удобства работы лучше создать небольшой командный файл (расширение .cmd) с содержимым: c.py C:\D\My\_Docs\Documents\uzh-master\tests\abc.py , где abc.py – имя файла с программой для софт-процессора.

Для адаптации компилятора Uzh-а к конкретному процессору потребуются некоторые изменения в его исходном коде. К счастью, мест, подлежащих корректировке в компиляторе не много. Основные «аппаратно-зависимые» файлы:

- asm.py – ассемблер и формирование чисел (литералов);

- gen.py – низкоуровневые правила формирования кода (функции, переменные, переходы и условия);

- stream.py – формирование загрузочного потока;

- macro.py – макроопределения, по факту – расширения базового языка аппаратно-специфичными функциями.

Следующие изменения в компиляторе коснутся модуля asm.py в котором описывается система команд процессора (прописываются мнемоники команд и опкоды команд) и способ представления/компиляции числовых значений – литералов.

Команды упаковываются в словарь, а за литералы отвечает функция lit(). Если с системой команд все просто – просто меняется список мнемоник и соответствующих им опкодов, то с литералами дело обстоит немного иначе.

Основные и самые ответственные изменения/определения – в модуле gen.py. Данный модуль определяет основную логику работы/исполнения высокоуровневого кода на уровне ассемблера:

- условные и безусловные переходы;

- вызов функций и передача им аргументов;

- возврат из функций и возвращение результатов;

- подстройки под размеры памяти программ, памяти данных и стеков;

- последовательность действий при старте процессора.

Для поддержки ЯВУ процессор должен уметь достаточно произвольно работать с памятью и указателями и иметь область памяти для хранения локальных переменных функций – здесь вполне логично распределить функциональность регистров в соответствии с принятым в RISC-V ABI.

Теперь можно приступить к модификации кода модуля gen.py.

Переменные \*\_SIZE в комментариях не нуждаются и требуют только подстановки значений, заданных в проекте процессорного ядра.

Список STUB – временная заглушка для формирования места для адресов переходов с последующим их заполнением компилятором (текущие значения соответствуют 24-битному адресному пространству памяти кода).

Список STARTUP – задает последовательность действий, выполняемых ядром после сброса – будет задан начальный адрес памяти локальных переменных – 900, и переход на точку старта.

Функция func() прописывает действия, производимые при вызове функции, а именно – перенос аргументов функции в область локальных переменных, выделение памяти для собственных локальных переменных функции.

Epilog() определяет действия при возвращении из функции – освобождение памяти временных переменных, возврат на точку вызова.

Работа с переменными идет посредством их адресов, ключевое определение для этого – push\_local(), возвращающее адрес «высокоуровневой» переменной.

Следующие ключевые моменты – это условный и безусловный переходы. И основное определение компилятора на низком уровне – набор правил для операций языка и работы с памятью.

Модуль macro.py позволяет несколько «расширить» словарь целевого языка за счет макроопределений на ассемблере целевого процессора. Для компилятора языка высокого уровня определения в macro.py не будут отличаться от «родных» операторов и функций языка.

Путь создания своего компилятора, несмотря на возможно чувствительный по времени цикл разработки, представляется перспективным. Плюсы – возможность быстрой и простой адаптации инструмента под текущий проект.

1. <https://github.com/riscv/riscv-gnu-toolchain>
2. Adding custom instructions compilation support, to RISCV toolchain. – <https://medium.com/@viveksgt/adding-custom-instructions-compilation-support-to-riscv-toolchain-78ce1b6efcf4>
3. Adding Custom Instructions to the RISC-V GNU-GCC toolchain – <https://hsandid.github.io/posts/risc-v-custom-instruction/>
4. GNU toolchain for RISC-V, including GCC – <https://github.com/riscvcollab/riscv-gnu-toolchain>
5. <https://github.com/celebi-pkg/riscv-assembler>
6. Lisp RISC-V Assembler – <https://gitmemories.com/technoblogy/lisp-riscvassembler>
7. GitHub - true-grue\_uzh\_ Uzh compiler // <https://github.com/true-grue/uzh>
8. GitHub - true-grue\_raddsl\_ Tools for rapid prototyping of DSL compilers // <https://github.com/true-grue/raddsl>
9. <http://sovietov.com/txt/dsl_python_conf.pdf>