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

Организация докер-контейнера с RISC-V тулчейном

Студент: гр. 853504
Гончарик Г. С.

Руководитель: ассистент
кафедры
информатики Леченко А. В.

Минск 2020

Оглавление

1. Введение

При разработке часто встает вопрос настройки и установки тулчейна для компиляции кода. И, конечно, в процессе часто возникает много независимых друг от друга проблем, на решение которых требуется немало времени. Причем никто не гарантирует, что при повторной установке или обновлении вашего набора инструментов не возникнет новых ошибок. Поэтому, в теории, было бы хорошо настроить и протестировать среду для программирования один раз и использовать ее повторно на хост-системах с разным аппаратным и программным обеспечением. Этого возможно достичь с помощью виртуализации.

Виртуализация представляет собой процесс запуска виртуальной компьютерной системы на уровне, абстрагированном от реального аппаратного обеспечения. Чаще всего она относится к запуску нескольких операционных систем на одном компьютере одновременно. Для приложений, запущенных поверх виртуализированной машины, может показаться, что они находятся на собственной выделенной машине, где операционная система, библиотеки и другие программы уникальны для гостевой виртуализированной системы и не связаны с операционной системой хоста, находящейся под ней.

Существует много причин, по которым люди используют виртуализацию в вычислениях. Для пользователей настольных компьютеров наиболее распространенным является возможность запуска приложений, предназначенных для другой операционной системы, без необходимости переключать компьютеры или перезагружать их в другую систему. Для администраторов серверов виртуализация также предлагает возможность запускать различные операционные системы, но, возможно, что более важно, она предлагает способ сегментировать большую систему на множество более мелких частей, позволяя серверу более эффективно использоваться несколькими разными пользователями или приложениями с разными потребностями. Она также позволяет изолировать, сохраняя программы, запущенные внутри виртуальной машины, от процессов, происходящих на другой виртуальной машине на том же самом хосте.

2. Постановка задачи

Необходимо организовать докер-контейнер с RISC-V тулчейном. Контейнер должен быть собран с предустановленным тулчейном собранным из исходников и параметризован на этапе сборки с возможностью выбрать любую из доступных опций: 32 или 64 битная архитектура(rv32i или rv64i), стандартные наборы расширений((a)tomics, (m)ultiplication and division, (f)loat, (d)ouble, or (g)eneral), а также двоичный интерфейс приложений или IBA(ilp32, ilp32d, ilp32f, lp64, lp64d, lp64f). Также в репозитории должна быть возможность задать коммит или ветку, которую надо собрать.

3. Теория

Контейнер или VM

Виртуальная машина является эмулируемым эквивалентом компьютерной системы, которая работает поверх другой системы. Виртуальные машины могут иметь доступ к любому количеству ресурсов: вычислительной мощности, через аппаратный, но ограниченный доступ к процессору и памяти хост-машины; одному или нескольким физическим или виртуальным дисковым устройствам для хранения данных; виртуальному или реальному сетевому интерфейсу; а также любым устройствам, таким как видеокарты, USB-устройства или другое аппаратное обеспечение, которое совместно используется с виртуальной машиной. Если виртуальная машина хранится на виртуальном диске, ее часто называют образом диска. Образ диска может содержать файлы для загрузки виртуальной машины, или может содержать любые другие специфические требования к хранилищу.

Возможно, вы слышали о контейнерах Linux, которые концептуально схожи с виртуальными машинами, но функционируют несколько иначе. В то время как и контейнеры, и виртуальные машины позволяют запускать приложения в изолированной среде, позволяя Вам складывать многие из них в стек на одной машине, как если бы они были отдельными компьютерами, контейнеры не являются полными, независимыми машинами. На самом деле контейнер - это просто изолированный процесс, который использует то же ядро Linux, что и операционная система хоста, а также библиотеки и другие файлы, необходимые для выполнения программы, запущенной внутри контейнера, часто с сетевым интерфейсом, таким образом, что контейнер может быть открыт миру так же, как и виртуальная машина. Как правило, контейнеры предназначены для выполнения одной программы и определенного стека программ. Одним из программных решений для автоматического развертывания и управления приложениями в средах с поддержкой контейнеризации является Docker.

Virtualization	Docker
Hypervisor	Libcontainer
Virtual Machine	Container
Template	Image
Linked VMs	Repository
Catalog	Registry
Boots in minutes	Boots in seconds
Can run variety of guest OS	Limited to Linux
VMs are dependent on hypervisor	Containers are portable
VMs run into a few GBs	Containers are lightweight

Рисунок 1. Сравнение виртуализации и Docker

Docker

Docker является инструментом, предназначенным для облегчения создания, развертывания и запуска приложений с помощью контейнеров. Контейнеры позволяют разработчику упаковать приложение со всеми необходимыми ему компонентами, такими как библиотеки и другие зависимости, и развернуть его как один пакет. Таким образом, благодаря контейнеру, разработчик может быть уверен, что приложение будет запущено на любой другой машине с Linux, независимо от любых настроек, которые могут отличаться от настроек машины, используемой для написания и тестирования кода.

В некотором смысле Docker немного похож на виртуальную машину. Но в отличие от виртуальной машины, вместо того, чтобы создавать целую виртуальную операционную систему, Docker позволяет приложениям использовать то же ядро Linux, что и система, на которой они запущены, и требует, чтобы приложения поставлялись только с компонентами, которые еще не запущены на хост-компьютере. Это дает значительный прирост производительности и уменьшает размер приложения.

И что немаловажно, Docker является open source (с открытым исходным кодом) проектом. Это означает, что любой может внести свой вклад в Docker и расширить его для удовлетворения своих собственных потребностей, если ему нужны дополнительные функции, которые не доступны из коробки.

Docker разработан в интересах как разработчиков, так и системных администраторов, что делает его частью многих тулчейнов. Для разработчиков это означает, что они могут сосредоточиться на

написании кода, не беспокоясь о системе, на которой их тулчейн или программа в конечном итоге будет работать. Это также позволяет им получить преимущество, используя одну из тысяч программ, уже разработанных для запуска в контейнере Docker в качестве части своего приложения.

RISC-V

Сегодня, если вы хотите построить высокопроизводительное вычислительное устройство, вы почти наверняка найдете все необходимые вам программы в свободной и открытой форме. Правда, это не относится к процессорным чипам, на которых работают эти свободно распространяемые программы - что бы вы ни выбрали, часть ваших расходов уйдет на лицензии проприетарного оборудования Intel, ARM или их друзьям.

RISC-V является новой архитектурой, которая доступна под открытыми, свободными и неограниченными лицензиями. Она имеет широкую поддержку в отрасли от производителей чипов и устройств, и предназначена для свободного расширения и настройки под любую рыночную нишу. Однако, чтобы быть успешной, она должна работать как технически, так и экономически эффективно, чтобы проектировать, программировать и тестировать.

История

Одна из основополагающих истин вычислений, впервые открытая Аланом Тьюрингом, заключается в том, что любой компьютер теоретически может решить любую проблему. Другая истина заключается в том, что если вы можете делать это на аппаратном обеспечении, то вы можете делать это на программном обеспечении, и наоборот. Реальные системы, однако, ограничены скоростью, эффективностью и ресурсами. Различные архитектуры процессоров допускают различные компромиссы.

В 1980-х годах произошла битва между микросхемами с несколькими специальными аппаратными блоками, чтобы справиться с конкретными ситуациями - CISC, или Complex Instruction Set Computing - и теми, которые сохраняли аппаратное обеспечение простым, быстрым и универсальным, и оставили сложности программному обеспечению.

Этот подход, названный Reduced ISC или RISC, поначалу казался проигрышным, так как Intel поднялся до статуса абсолютного лидера со своими x86 CISC чипами. Чипы RISC, такие как Sun Microsystem SPARC и IBM PowerPC, были неплохи, но никогда не преодолевали Intel - которая была симбиотичной с сильно ориентированной на Intel Windows от Microsoft. Однако под капотом, микросхемы Intel были удивительно похожи на RISC, а возможности CISC были переведены на RISC инструкции внутри компании. Мобильный рынок, который развивался независимо от Windows, вскоре остановился на ARM-чипах, которые использовали RISC-дизайн, чтобы быть гораздо более энергоэффективными. Можно сказать, что RISC выиграл дело повсеместно.

Для разработчиков аппаратного обеспечения 21 века, однако, и x86- и ARM-чипы имеют основной недостаток: стоимость. Это стоимость не только кремния, но и интеллектуальной собственности, в первую очередь патентов, которыми владеют крупные компании и за которые приходится платить - ARM через лицензирование тысяч партнеров, Intel путем существования в непростой дуополии с AMD.

RISC-V стремится развалить проприетарное владение процессорами точно так же, как открытое программное обеспечение высвободило огромные массивы индустрии.

Техническая часть

В основе RISC-V лежит массив из 32 регистров, содержащих состояние процессора, данные, которые сразу же обрабатываются, и информацию о системе управления. Этот массив достаточно большой, чтобы свести к минимуму необходимость доступа к внешней памяти для выполнения большого количества основных задач процессора, что снижает энергопотребление и увеличивает скорость работы. RISC-V поставляется в 32-битном и 64-битном вариантах, при этом размер регистра меняется в соответствии с ним. Ведется работа над 128-битной версией.

В ее основе лежит массив из 32 регистров, содержащих состояние процессора, данные, которые сразу же обрабатываются, и информацию о системе управления. Этот массив достаточно большой, чтобы свести к минимуму необходимость доступа к внешней памяти для выполнения большого количества основных задач процессора, что снижает

энергопотребление и увеличивает скорость работы. RISC-V поставляется в 32-битном и 64-битном вариантах, при этом размер регистра меняется в соответствии с ним. Ведется работа над 128-битной версией.

Набор команд - низкоуровневые команды, которые процессор может напрямую интерпретировать - очень прост и модулен. Ядро RISC-V может быть собрано с помощью простых целочисленных инструкций, даже без умножения или деления. Или же оно может иметь добавленные, расширения с плавающей точкой (еще с 32 регистрами), и сжатые инструкции.

Сжатие является ключевой частью RISC-V. Первоначальная концепция RISC достигает скорости, благодаря тому, что его инструкции кодируются в форме, очень простой и быстрой для аппаратного обеспечения, чтобы декодировать и выполнять - без особых случаев, все вписывается в единый фреймворк. Это, однако, может быть очень неэффективно при использовании памяти, так как шаблон инструкции должен учитывать все возможности независимо от того, нужны ли они той или иной операции или нет.

Сжатие нарушает это правило и позволяет компилятору помещать гораздо больше инструкций в заданную область памяти. Инструкции имеют переменную длину, как и в x86 CISC. В отличие от x86, которая создавалась по частям в течение многих лет, спецификации длины переменных RISC-V спроектированы с самого начала, чтобы дать аппаратуре процессора как можно больше возможностей знать их длину и позволить ему быстро расшифровывать их, управляя своими внутренними очередями инструкций. Управление очередями - это то место, где большинство современных процессорных архитектур достигают своей скорости.

Основная спецификация RISC-V сертифицирована как свободная от патентных ограничений и лицензирована в соответствии с Creative Commons CC BY 4.0. Это не требует и не означает, что расширения должны быть одинаково свободными и открытыми - при желании разработчики могут включать лицензионные аспекты в свои дополнения. Ключевым моментом является то, что архитектура расширяема без потери эффективности.

4. Ход работы

В ходе работы мы напишем ряд Dockerfile-ов для нескольких вариантов рабочих процессов, которые будут включать в себя целевой тулчейн и симулятор для исполнения RISC-V программ, протестируем сделанное с помощью тестов, которые сами соберем из исходников, скомпилируем и запустим простую программу вида “Hello, World!”, а также загрузим все сделанное на Github и сгруппируем для удобства в виде веток.

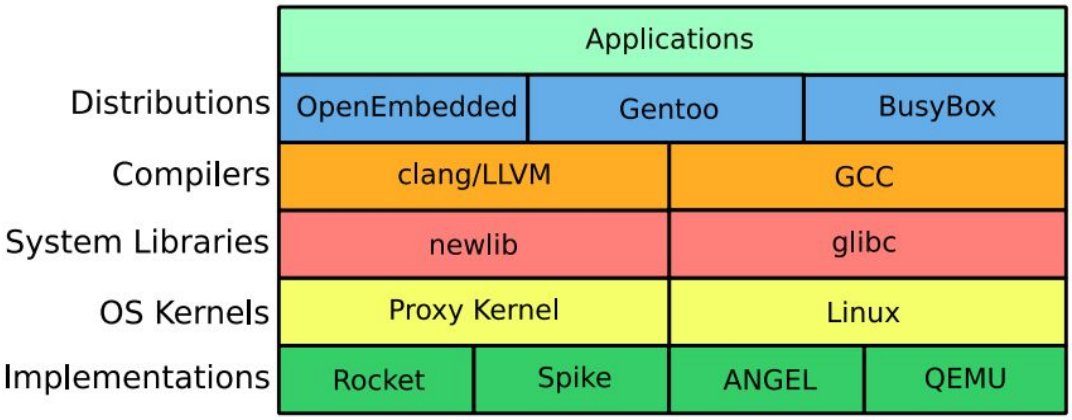

Рисунок 3. Программный стек RISC-V

Предусловия

В данной работе не будут рассматриваться процесс установки и настройки Docker. А также автор полагает, что читатель уже знаком с процессом работы в Docker и командным интерпретатором в юниксоподобных системах. Все тесты будут рассмотрены отдельно, и не будут включены в готовые сборки. В качестве хост-машины будет использоваться x86-устройство с установленной ОС Ubuntu 20.04.

Параметризованный контейнер

Начнем с параметризованного контейнера, в котором выполним все требования поставленной задачи: тулчейн из исходников, выбор разрядности архитектуры, а также стандартных наборов расширений и двоичного интерфейса приложения. В качестве тулчейна будет использован riscv-gnu-toolchain, исходный код которого можно найти на GitHub. Параметризация будет обеспечиваться проксированием аргументов Docker, переданных на этапе сборки образа контейнера, в конфигурационный файл, поставляемый с исходными файлами тулчейна.

Начнем рассматривать файл Dockerfile для параметризованного контейнера.

Примечание. Полный код всех рассмотренных файлов можно найти в разделе Приложение 1, где находится и ссылка на репозиторий.

Первым делом указываем из какого образа будем строить наш образ. Я использую базовый образ Ubuntu 20.04.

```
FROM ubuntu:20.04
```

Далее указываем метаданные об образе, где отмечаем автора, назначение и версию образа.

```
LABEL maintainer="georgii.goncharik@gmail.com"
LABEL version="0.1"
LABEL description="docker-container with RISC-V GNU
Compiler Toolchain, parameterized at build stage."
```

Объявляем переменные, чтобы упростить работу и повысить читаемость команд.

```
ENV RISCV=/opt/riscv
ENV SRCDIR=/tmp/riscv-tools
ENV PROJDIR=/riscv-projects
ENV PATH=$RISCV/bin:$PATH
```

Теперь объявим аргументы, которые будем передавать при сборке нашего образа: ARCH – архитектура, ABI – двоичный интерфейс приложения, LINUX – опциональный аргумент для компиляции Linux кросс-компилятора.

```
ARG ARCH
ARG ABI
ARG LINUX
```

А также один специфический аргумент.

```
ARG DEBIAN_FRONTEND=noninteractive
```

С помощью строки выше включается режим, когда требуется нулевое взаимодействие при установке или обновлении системы через пакетный менеджер apt. Он принимает ответ по умолчанию на все вопросы и может отправить сообщение об ошибке корневому пользователю – идеальный режим для автоматической установки.

Обновляем базы данных пакетов и устанавливаем необходимые нам.

```
RUN apt-get update
RUN apt-get install -y 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
RUN apt-get install -y git
```

Переходим в заранее созданную папку, на которую ссылается переменная окружения SRCDIR, и клонируем в нее репозиторий riscv-gnu-toolchain и все подмодули с исходными кодами тулчейна.

```
WORKDIR $SRCDIR
RUN git clone https://github.com/riscv/riscv-gnu-toolchain
WORKDIR $SRCDIR/riscv-gnu-toolchain
RUN git submodule update --init --recursive
```

Конфигурируем сборку тулчейна, передавая аргументы командной строки, полученные при запуске сборки образа Docker.

```
RUN ./configure --prefix=${RISCV} --with-arch=${ARCH}
--with-abi=${ABI} && make ${LINUX}
```

Удаляем исходный код тулчейна.

```
WORKDIR $SRCDIR
RUN rm -r -f riscv-gnu-toolchain
```

И наконец, переходим в рабочую директорию, созданную с целью стандартизировать корневую папку для проектов, которая будет открываться по умолчанию при запуске контейнера.

```
WORKDIR $PROJDIR
```

Дополнительные контейнеры

Также рассмотрим несколько Dockerfile-ов, которые не будут параметризованные, но зато предоставят готовые к работе контейнеры с тулчейном, без необходимости разбираться в передаваемых аргументах на этапе сборки.

Linux

В репозитории на Github имеется две ветви linux и linux_32 с 64 и 32-битным тулчейном соответственно. По факту, в коде минимальное количество изменений, что, с другой стороны, подтверждает универсальность параметризованного контейнера.

Программы можно запускать под установленной системой Linux. Программы, запущенные под Linux, имеют полную системную поддержку таких стандартов, как pthread, time и т.д.

По умолчанию сборка тулчейна ориентирована на RV64GC (64 бит), даже в 32-битной сборочной среде. Для того, чтобы собрать фактически 32-битный тулчейн, передадим конфигурационный скрипт следующий аргумент:

```
--with-xlen=32.
```

В результате, чтобы собрать 32-битный тулчейн, необходимо сконфигурировать тулчейн следующим образом (dockerfile).

```
RUN ./configure --prefix=${RISCV} --with-xlen=32
--with-arch=rv32gc
```

Proxy Kernel + Spike

Теперь рассмотрим ветку pk на Github, которая содержит 2 dockerfile-а: с тулчейном и симулятором Spike.

Тесты и системы, не требующие доступа к полнофункциональной ОС, могут быть скомпилированы для использования с прокси-ядром (PK) и newlib. Прокси-ядро просто проксирует syscalls на хост через целевой

интерфейс хоста (HTIF). Это обеспечивает поддержку часто используемых функций, таких как printf().

Прокси-ядро RISC-V, pk, представляет собой легковесную среду выполнения приложений, в которой могут размещаться статически скомпонованные двоичные ELF-файлы RISC-V. Оно разработано для поддержки реализаций RISC-V с ограниченными возможностями ввода/вывода и, таким образом, обрабатывает системные вызовы ввода/вывода, проксируя их на хост-компьютер.

Spike является золотым эталоном функционального симулятора программного обеспечения RISC-V ISA C++. Он обеспечивает полную эмуляцию системы или прокси-серверную эмуляцию с помощью HTIF/FESVR. Он служит отправной точкой для запуска RISC-V программного обеспечения.

Компилятор GCC riscv64-unknown-elf-gcc производит код для использования с PK и newlib.

Процесс сборки тулчейна мало отличается от того, что мы рассматривали ранее, поэтому не будем на этом останавливаться. При желании код доступен в разделе Приложение 1 и в репозитории на Github. Лучше мы рассмотрим dockerfile контейнера, содержащий симулятор Spike. Приступим.

Контейнер с симулятором базируется на образе с newlib тулчейном, поэтому компилятор и все остальные утилиты остаются доступными.

```
FROM newlib:64
```

Объявим необходимые переменные.

```
ENV SIMSRC=/tmp/sim
ENV PATH=$SIMSRC/riscv-isa-sim/build/:$PATH
```

Переходим в директорию для исходных кодов и копируем репозитории с прокси-ядром и симулятором Spike.

```
WORKDIR ${SIMSRC}
```

```
RUN git clone https://github.com/riscv/riscv-pk
RUN git clone https://github.com/riscv/riscv-isa-sim
```

Конфигурируем и собираем прокси-ядро.

```
WORKDIR ${SIMSRC}/riscv-pk
RUN mkdir build
WORKDIR ${SIMSRC}/riscv-pk/build
RUN ../configure --prefix=$RISCV
--host=riscv64-unknown-elf
RUN make && make install
```

Теперь то же самое с симулятор Spike.

```
WORKDIR ${SIMSRC}/riscv-isa-sim
RUN mkdir build
WORKDIR ${SIMSRC}/riscv-isa-sim/build
RUN ../configure --prefix=$RISCV --enable-histogram
RUN make && make install
```

Переходим в рабочую директорию.

```
WORKDIR $PROJDIR
```

Использование

В этом и следующем разделе будем использовать контейнер из ветки pk, так как только она включает в себя симулятор Spike. Важно уточнить, что Spike может исполнять только rv64* bare-metal/elf файлы. Linux тулчейны не поставляются с функциональными симуляторами, поскольку это выходит за границы данной работы. Spike был добавлен, чтобы вы “из коробки” могли выполнить то, что будет написано далее.

Целью поставим написать, скомпилировать и запустить простую программу типа “Hello, World!”. Но для начала соберем образ. Так как образ с симулятором базируется на образе с тулчейном, то сначала собираем его. Для этого переходим в папку /toolchain и выполняем следующую команду.

```
$ [sudo] docker build . -t newlib:64
```

После сборки переходим в директорию с dockerfile-ом для Spike и делаем то же самое.

```
$ [sudo] docker build . -t spike:64
```

Теперь мы готовы двигаться дальше. Запускаем контейнер из образа spike:64 и не забываем, что он включает в себя и тулчейн.

```
$ [sudo] docker run -it [-v SomeVolume:/riscv-projects] spike:64 /bin/bash
```

Мы “внутри” контейнера с доступом к bash. Давайте создадим файл и напишем простую команду на языке C.

```
$ cat > hello.c << 'EOF'
> #include<stdio.h>
> int main(void){
>    printf("Hello, RISC-V!\n");
>    return 0;
> }
> EOF
```

Скомпилируем программу следующим образом.

```
$ riscv64-unknown-elf-gcc -O2 -o hello hello.c
```

Чтобы просмотреть полученный бинарный файл, можно выполнить эти команды.

```
$ riscv64-unknown-elf-readelf -a hello | less
$ riscv64-unknown-elf-objdump -d hello | less
```

Обратите внимание, что newlib поддерживает только статическое связывание.

Образы с тулчейном и симулятором собраны, контейнер запущен, программа написана и скомпилирована, а значит мы можем ее наконец запустить.

```
$ spike pk hello
```

Тесты

Тесты не поставляются ни с одним из образов, поэтому их возьмем из репозитория, указанного в разделе Приложение 1, и будем выполнять на Spike-контейнере. Считаем, что контейнер уже запущен и клонируем из репозиторий.

```
$ git clone https://github.com/riscv/riscv-tests
$ cd riscv-tests
$ git submodule update --init --recursive
```

Осталось собрать.

```
$ autoconf
$ ./configure --prefix=$RISCV/target
$ make
$ make install
```

Мы имеем собранные из исходников тесты. Перейдем в директорию isa и запустим тесты.

```
$ make run
```

Тесты пройдены успешно.

5. Вывод

Итак, мы рассмотрели тему виртуализации и контейнеризации, взглянули на одну из реализаций виртуализации на уровне операционной системы в лице Docker, разобрались с целью разработки архитектуры RISC-V и немного с ее реализацией, а, главное, разработали docker образы с RISC-V GNU тулчейном для разных целевых платформ. На одной из версий контейнеров была написана, скомпилирована и запущена простая программа с выводом в стандартный поток ввода-вывода, а также были собраны и запущены юнит-тесты. В ходе работы были изучены основы работы с Docker, а также с RISC-V GNU тулчейном и симулятором Spike. Следующим этапом в изучении может стать разработка более функциональных программ под RISC-V или, например, установка и настройка Linux на RISC-V QEMU. В любом случае, в этой работе было затронуто немало важных и актуальных тем, каждая из которых заслуживает написания не одной работы наподобие этой.

Литература

1. https://inst.eecs.berkeley.edu/~cs250/fa10/handouts/tut3-riscv.pdf
2. https://medium.com/@tonistiigi/early-look-at-docker-containers-on-risc-v-40ed43b16b09
3. https://github.com/riscv/riscv-gnu-toolchain
4. https://www.sifive.com/blog/all-aboard-part-1-compiler-args
5. https://github.com/Charles-J97/run-linux-with-riscv-emulators
6. https://www.howtoforge.com/tutorial/how-to-create-docker-images-with-dockerfile/
7. https://riscv.org/wp-content/uploads/2015/02/riscv-software-stack-tutorial-hpca2015.pdf
8. https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html
9. https://www.lowrisc.org/docs
10. https://prydt.xyz/b/2020/06/16/installing-spike.html
11. https://docs.docker.com/

Приложение 1

Ссылка на репозиторий Github:

https://github.com/GeorgeiGoncharik/AVS-Coursework-5

Ссылка на репозиторий Github с тестами:

https://github.com/riscv/riscv-tests

Ветка: main

./dockerfile

```
FROM ubuntu:20.04

LABEL maintainer="georgii.goncharik@gmail.com"
LABEL version="0.1"
LABEL description="docker-container with RISC-V GNU
Compiler Toolchain, parameterized at build stage."


ENV RISCV=/opt/riscv
ENV SRCDIR=/tmp/riscv-tools
ENV PROJDIR=/riscv-projects
ENV PATH=$RISCV/bin:$PATH

ARG DEBIAN_FRONTEND=noninteractive

ARG ARCH
ARG ABI

# if you want to build for linux elf pass 'linux', in case
of newlib do nothing
ARG LINUX

RUN apt-get update
RUN apt-get install -y 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
RUN apt-get install -y git

WORKDIR $SRCDIR
RUN git clone https://github.com/riscv/riscv-gnu-toolchain
WORKDIR $SRCDIR/riscv-gnu-toolchain
RUN git submodule update --init --recursive

RUN ./configure --prefix=${RISCV} --with-arch=${ARCH}
--with-abi=${ABI} && make ${LINUX}

WORKDIR $SRCDIR
RUN rm -r -f riscv-gnu-toolchain

WORKDIR $PROJDIR
```

Ветка: newlib

./toolchain/dockerfile

```
FROM ubuntu:20.04

LABEL maintainer="georgii.goncharik@gmail.com"
LABEL version="0.1"
LABEL description="docker-container with RISC-V GNU
Compiler Toolchain (newlib)"

ENV RISCV=/opt/riscv
ENV SRCDIR=/tmp/riscv-tools
ENV PROJDIR=/riscv-projects
ENV PATH=$RISCV/bin:$PATH

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update
RUN apt-get install -y 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
RUN apt-get install -y git

WORKDIR $SRCDIR
RUN git clone https://github.com/riscv/riscv-gnu-toolchain
WORKDIR $SRCDIR/riscv-gnu-toolchain
RUN git -c submodule."qemu".update=none submodule update
--init --recursive

RUN ./configure --prefix=${RISCV} && make

WORKDIR $SRCDIR
RUN rm -r -f riscv-gnu-toolchain

WORKDIR $PROJDIR
```

./simulator/dockerfile

```
FROM newlib:64

RUN apt-get update
RUN apt-get install -y device-tree-compiler

ENV SIMSRC=/tmp/sim
ENV PATH=$SIMSRC/riscv-isa-sim/build/:$PATH
WORKDIR ${SIMSRC}

RUN git clone https://github.com/riscv/riscv-pk
RUN git clone https://github.com/riscv/riscv-isa-sim

WORKDIR ${SIMSRC}/riscv-pk
RUN mkdir build
WORKDIR ${SIMSRC}/riscv-pk/build
RUN ../configure --prefix=$RISCV
--host=riscv64-unknown-elf
RUN make && make install

WORKDIR ${SIMSRC}/riscv-isa-sim
RUN mkdir build
WORKDIR ${SIMSRC}/riscv-isa-sim/build
```

```
RUN ../configure --prefix=$RISCV --enable-histogram
RUN make && make install

WORKDIR $PROJDIR
```

Ветка: qemu_linux

./dockerfile

```
FROM ubuntu:20.04

LABEL maintainer="georgii.goncharik@gmail.com"
LABEL version="0.1"
LABEL description="docker-container with RISC-V GNU
Compiler Toolchain (Linux 64-bit)"

ENV RISCV=/opt/riscv
ENV SRCDIR=/tmp/riscv-tools
ENV PROJDIR=/riscv-projects
ENV PATH=$RISCV/bin:$PATH

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update
RUN apt-get install -y 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
RUN apt-get install -y git

WORKDIR $SRCDIR
RUN git clone https://github.com/riscv/riscv-gnu-toolchain
WORKDIR $SRCDIR/riscv-gnu-toolchain
RUN git -c submodule."qemu".update=none -c
submodule."riscv-newlib".update=none submodule update
--init --recursive

RUN ./configure --prefix=${RISCV} && make linux
```

```
WORKDIR $SRCDIR
RUN rm -r -f riscv-gnu-toolchain

WORKDIR $PROJDIR
```

Ветка: qemu_linux_32

./dockerfile

```
FROM ubuntu:20.04

LABEL maintainer="georgii.goncharik@gmail.com"
LABEL version="0.1"
LABEL description="docker-container with RISC-V GNU
Compiler Toolchain (Linux 32-bit)"

ENV RISCV=/opt/riscv
ENV SRCDIR=/tmp/riscv-tools
ENV PROJDIR=/riscv-projects
ENV PATH=$RISCV/bin:$PATH

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update
RUN apt-get install -y 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
RUN apt-get install -y git

WORKDIR $SRCDIR
RUN git clone https://github.com/riscv/riscv-gnu-toolchain
WORKDIR $SRCDIR/riscv-gnu-toolchain
RUN git -c submodule."qemu".update=none -c
submodule."riscv-newlib".update=none submodule update
--init --recursive

RUN ./configure --prefix=${RISCV} --with-xlen=32
--with-arch=rv32gc && make linux
```

```
WORKDIR $SRCDIR
RUN rm -r -f riscv-gnu-toolchain

WORKDIR $PROJDIR
```