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 курсовому проекту
на тему

«Реализация Docker контейнера с RISCV тулчейном »

Студент гр. 853506
Акулича К.И.
Руководитель
Ассистент кафедры информатики
Леченко А. В.

Минск 2020

Содержание

Введение

Контейнеры коренным образом изменяют способ разработки, распространения и функционирования программного обеспечения. Разработчики могут создавать программное обеспечение на локальной системе, точно зная, что оно будет работать одинаково в любой операционной среде – в программном комплексе ИТ-отдела, на ноутбуке пользователя или в облачном кластере. Инженеры по эксплуатации могут сосредоточиться на поддержке работы в сети, на предоставлении ресурсов и на обеспечении бесперебойной работы и тратить меньше времени на конфигурирование окружения и на «борьбу» с системными зависимостями. Масштабы перехода к практическому применению контейнеров стремительно растут во всей промышленности информационных технологий, от небольших стартапов до крупных предприятий. Разработчики и инженеры по эксплуатации должны понимать, что необходимость постоянного

использования контейнеров будет возрастать в течение нескольких следующих лет.

Контейнеры (containers) представляют собой средства инкапсуляции приложения вместе с его зависимостями. На первый взгляд контейнеры могут показаться всего лишь упрощенной формой виртуальных машин (virtual machines – VM) –как и виртуальная машина, контейнер содержит изолированный экземпляр операционной системы (ОС), который можно использовать для запуска приложений.

Анализ предметной области
Docker контейнер

Несмотря на то что контейнеры и виртуальные машины на первый взгляд кажутся похожими, между ними существуют важные различия, которые проще всего продемонстрировать на графических схемах.

На рис. 1.1 показаны три приложения, работающих в отдельных виртуальных машинах на одном хосте. Здесь требуется гипервизор1 для создания и запуска виртуальных машин, управляющий доступом к нижележащей ОС и к аппаратуре,а также при необходимости интерпретирующий системные вызовы. Для каждой виртуальной машины необходимы полная копия ОС, запускаемое приложение и все библиотеки поддержки.

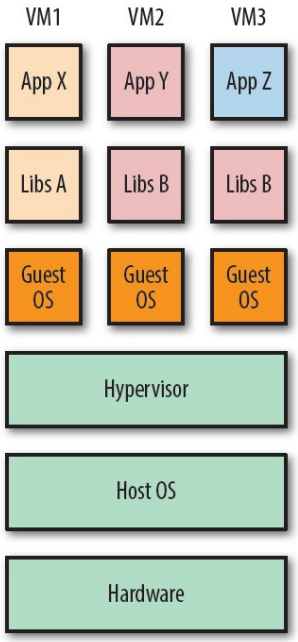


Рис. 1.1. Три виртуальные машины, работающие на одном хосте

В противоположность описанной схеме на рис. 1.2 показано, как те же самые три приложения могут работать в системе с контейнерами. В отличие от виртуальных машин, ядро2 хоста совместно используется (разделяется) работающими контейнерами. Это означает, что контейнеры всегда ограничиваются использованием того же ядра, которое функционирует на хосте. Приложения Y и Z пользуются одними и теми же библиотеками и могут совместно работать с этими данными, не создавая избыточных копий.

Внутренний механизм контейнера отвечает за пуск и остановку контейнеров так же, как гипервизор в виртуальной машине. Тем не менее процессы внутри контейнеров равнозначны собственным процессам ОС хоста и не влекут за собой дополнительных накладных расходов, связанных с выполнением гипервизора.

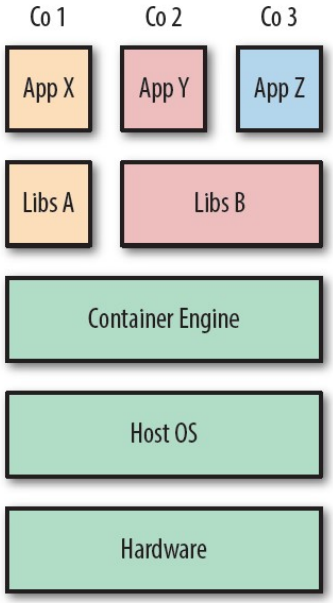


Рис. 1.2. Три контейнера, работающих на одном хосте

Dockerfile

Dockerfile – это обычный текстовый файл, содержащий набор операций, которые могут быть использованы для создания Docker-образа.

Иструкции dockerfile

CMD

Запускает заданную инструкцию во время инициализации контейнера. Если была определена инструкция ENTRYPOINT, то заданная здесь инструкция будет интерпретироваться как аргумент для ENTRY POINT (в этом случае необходимо использовать формат exec). Инструкция CMD замещается любыми аргументами, указанными в команде docker run после имени образа. В действительности выполняется только самая последняя инструкция CMD, а все предыдущие инструкции CMD будут отменены (в том числе и содержащиеся в основных образах).

COPY

Используется для копирования файлов из контекста создания в образ. Имеет два формата: COPY источник цель и COPY ["источник", "цель"] – оба копируют файл или каталог из *«источник»* в контексте создания в *«цель»* внутри контейнера. Формат JSON-массива обязателен, если путь содержит пробелы. Можно использовать шаблонные символы для определения нескольких файлов или каталогов. Следует обратить особое внимание на невозможность указания путей *«источника»*, расположенных вне пределов контекста создания (например, нельзя указать для копирования файл *../another_dir/myfile*).

ENTRYPOINT

Определяет выполняемый файл (программу) (и аргументы по умолчанию), запускаемый при инициализации контейнера. В эту выполняемую программу передаются как аргументы любые инструкции CMD или аргументы команды docker run, записанные после имени образа. Инструкции ENTRYPOINT часто используются для организации скриптов запуска, которые инициализируют переменные и сервисы перед обработкой всех передаваемых в образ аргументов.

FROM

Определяет основной образ для файла Dockerfile. Все последующие инструкции выполняют операции создания поверх заданного образа. Основной образ определяется в форме IMAGE:TAG (например, debian:wheezy). При отсутствии тега по умолчанию полагается latest, но я настоятельно рекомендую всегда явно указывать тег конкретной версии, чтобы избежать неприятных неожиданностей. Эта инструкция обязательно должна быть самой первой в Dockerfile.

RUN

Запускает заданную инструкцию внутри контейнера и сохраняет результат.

VOLUME

Объявляет заданный файл или каталог как том. Если такой файл или каталог уже существует в образе, то он копируется в том при запуске контейнера. Если задано несколько аргументов, то они интерпретируются как определение нескольких томов. Из соображений обеспечения безопасности и сохранения переносимости нельзя определить каталог хоста как том внутри файла Dockerfile. Более подробно об этом см. раздел «Управление данными с помощью томов и контейнеров данных» ниже.

Расширения RISCV

Для поддержки более общего разрабатываемого ПО,стандартные расширения определены для обеспечения умножения\деления,атомарных операций,арифметики с одинарной и двойной точностью.Стандартные челочисленный ISA имеет навание I(начинается с RV32 или RV64)и содержит целочисленные вычислительные инструкции,целочисленное чтения\запись,также операции управления потоком программы.Стандартные целочисленные умножения и деления имеют название «M»,и добавляет инструкции для умножения и деления челых чисел,хранящихся в регистрах.Стандартное расширение для атомарных операций ,имеет название «A»,и добавляет инструкции для атомарное чтения\записи\изменения для синхранизации.Стандартное расширение для чисел с плавующей точкой имеет название «F» и добавляет дробные регистры,дробные арфиметические инструкции инструкции и фунции дробного чтения\записи.Стандратное расширения для чисел с двойной точностью,имеющее название «D»,расширяет регистры с плавающей точкой и добавляет операции чтения\записи для чисел с двойной точностью.

«M» стандратное расширение для деления\умножения

Умножение

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

MUL выполняет умножение как XLEN-bit×XLEN-bit умножение rs1 на rs2 и помещает младшие XLEN бит в rd.MULH, MULHU и MULHSU выполняют то же умножение,только возвращают верхние XLEN бит от полного 2×XLEN умножения для signed×signed, unsigned×unsigned,and signed rs1×unsigned rs2 умножения,соответственно.Если необходим верхнийй и нижний результат ,тогда необходимо выполнить следующую последовательность команд MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2

Деление

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32

DIV и DIVU выполняют XLEN на XLEN битовое signed и unsigned битовое деление rs1 на rs2 .REM и REMU вычисляют остаток от деления.Для REM знак завасит от знака делителя.
Если необходимо частное и остаток от деления,то необходимо выполнить следующую цепочку команд DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2.
DIVW и DIVUW это RV64 инструкции ,которые делят младшие 32 бита rs1 на младшие 32 бита rs2.

Семантика деления на 0 и переполнения при делении приведена в следующей таблице.

Condition	Dividend	Divisor	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	x	0	$2^L - 1$	x	-1	x
Overflow (signed only)	-2^{L-1}	-1	–	–	-2^{L-1}	0

«F» стандартное расширение

F расширение добавляет 32 floating-point регистра,каждый длиной 32 бита ,а также fcsr регистр для статуса и контроля.Floating-point команды чтения и записи перемещают дробные числа между регистрами и памятью.Инструкции для перемещения значений в и из целочисленных регистров также доступны.

Floating-Point Control and Status регистр

31 8	7 5	4	3	2	1	0
Reserved	Rounding Mode (`frm`)	Accrued Exceptions (`fflags`)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

fcsr регистр может быть прочитан и записан с помощью FRCSR и FSCSR инструкций.FRCSR читает fcsr копированием его в регистр rd.FSCSR меняет значение в fcsr копированием его в регистр rd с последующей записью в регистр fcsr значения из rs1.

Операции с точкой имеют различные метод округления.

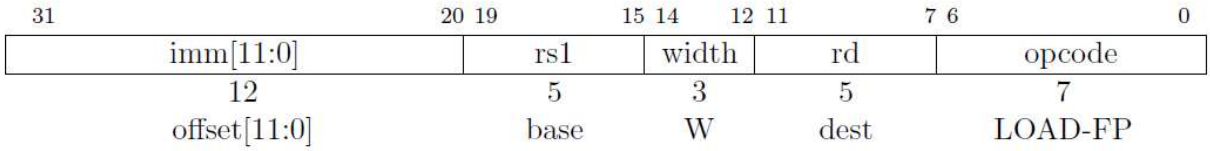
Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

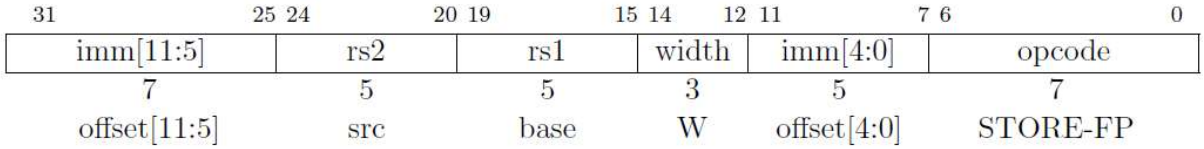
Флаги исключений устанавливаются в зависимости от появившихся исключений

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Операции чтения\записи single-precition

Операции чтения\записи дробных чисел используют base+offset адресный режим как для целых чисел,с базов в rs1 и 12 битовых сдвиг.FLW читает значение из памяти в регистр rs1.FSW записывает значение из регистра rs2 в память.





Single-Precision Floating-Point арифметические операции

Арифметические команды с дробным числами с одним или двумя операндами используют R-type команд.FADD.S и FMUL.S выполняют сложение и вычитание,соответственно между rs1 и rs2.FSUB.S выполняет вычитание между rs2 и rs1.FDIV.S выполняет целение rs1 на rs2.FSQRT.S извлекает кореь из rs1.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP

«D» стандартное расширение

D регистр

D регистр расширяет 32-битные регистры для floating-point до 64 бит.

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	width	rd	opcode	
12	5	3	5	7	
offset[11:0]	base	D	dest	LOAD-FP	

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode	
7	5	5	3	5	7	
offset[11:5]	src	base	D	offset[4:0]	STORE-FP	

FLD загружает double-precision floating-point значение из памяти в регистр rd.FSD записывает double-precision значение из регистра в память.

Double-Precision Floating-Point иструкции вычислений.
Данные операции аналогичны single-precision операциям

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	D	src2	src1	RM	dest	OP-FP
FMUL/FDIV	D	src2	src1	RM	dest	OP-FP
FMIN-MAX	D	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	D	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	D	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

«A» стандартное расширение

Стандартный RISV-V ISA имеет relaxed модель памяти,с барьерными(FENCE) инструкциями для навязывания дополнительных

упорядывающих ограничений.Адресное пространство разделено средой выполнения на 2 поля memort I/0,барьерные инструкции добавляют функцианальность для упоядования доступа к одному ии обоим одресным пространствам

Для обеспечения эффектиной поддержки release поседовательности каждая атомарная инструкция имеет 2 бита aq rl использующихся для задания доплнительного упорядовачиния памяти.Биты доступны одномуу или одному или обоим адресныи пространствам ,в зависимости в какой области памяти доступная атомарная операция.Никаких упорядывающих ограничений не применяется для доступа к другой области памяти и барьер памяти должне быть использован для упорядовачиния для 2 областей.

Если 2 бита не установлены,то никах ограничей на атомарные операции не налаживается.Если установлен только бит aq,то атомарная операция рассматривается как acquire access,т.е никакие следующие операции в данном RIST-V hart не могут выполняться восле asquer operation.Если только rl бит установлен,тогда атомарная операция рассматриваетя как release access,те никакие операции записи не могу выполнятся после любых следующих операций в этом RISV-V hart.Если оба бита установлены,то атомарная операция рассматривается как sequentially consistent.

Load-Reserved/Store-Conditional операции

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

Составные атомарные операции над одним или двумя словами выполняются с помощью oad-reserved (LR) и store-conditional (SC) инструкций.LR.W загружает слово по адресу rs1 и загружает его в rd,регистрирует reservation набор -набор бит ,которые вкл.чает в себя адресованое слово.SC.W ,соответственно,записывает слово в rs2 по адресу rs1 :SC.W выполняется успешно,если резервирование доступно и резервированный набор содержит биты,которые были в него записаны ранее.Если операция SC.W успешно,тогда в регистр rs2 записывется необходимое слово,а в регистр rd записвается значение 0.Если операция SC.W закончилась неудачей,то она не изменяет память и записывает значение 0 в rd.

Атомарные операции с памятью

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	aq rl	rs2	rs1	funct3	rd	opcode
5	1 1	5	5	3	5	7
AMOSWAP.W/D	ordering	src	addr	width	dest	AMO
AMOADD.W/D	ordering	src	addr	width	dest	AMO
AMOAND.W/D	ordering	src	addr	width	dest	AMO
AMOOR.W/D	ordering	src	addr	width	dest	AMO
AMOXOR.W/D	ordering	src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering	src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering	src	addr	width	dest	AMO

Атомарные операции выполняют read-modify-write операции для синхранизации и закодирование с помощью R-type формата команд.

31 30 25	24 21	20 19 15	14 12	11 8	7 6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

Эти атомарные операции загружают данные по адресу rs1 ,занят слово в регист rd ,применяют битовый оператор с загруженному значению и значению в регистре rs2 ,затем сохраняют результат по адресу rs1.

Поддреживаемы операции- обмен,сумма целых чисел,побитовае И,побитовое ИЛИ,побитовае XOR,знаковый или беззнаковый миниум.

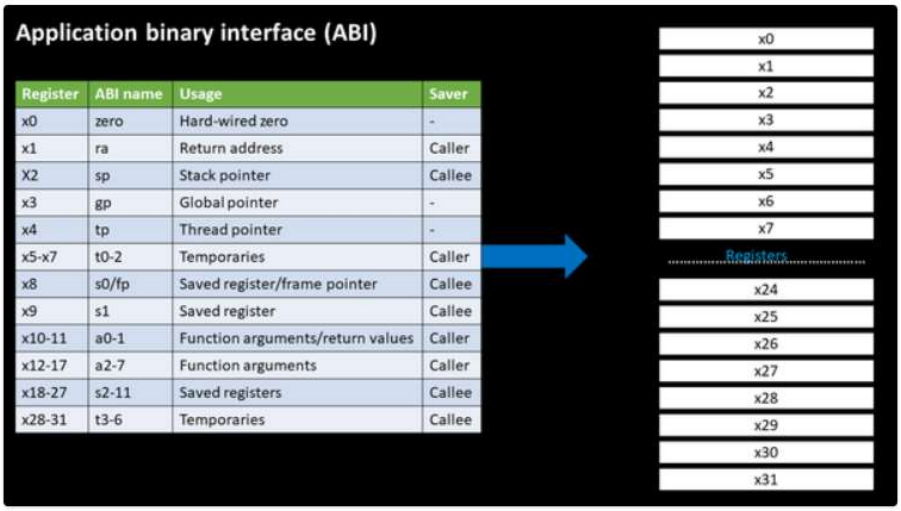
Для поддрежания синхранизации AMO команды используют логику,представленную выше.

```
    li          t0, 1        # Initialize swap value.
again:
    lw          t1, (a0)     # Check if lock is held.
    bnez        t1, again    # Retry if held.
    amoswap.w.aq t1, t0, (a0) # Attempt to acquire lock.
    bnez        t1, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Application binary interface(ABI)

Судя по названию ,это интерфейс,который помогает программам получить доступ к железу и сервисам.

Данный интерфейс состоит из 2 частей.Первая часть-польовательские инструкции,вторая-уровень операционной системы.



RISC-V архитектура имеет 32 регистра.прикладной программист имеет доступ к каждому их них,через ABI имена.Например,если необходимо узнать значение указателя стека или переместить указатель стека ,нужно выполнить следующую команду “addi sp, sp, -16”,где sp это ABI имя указателя стека.Для целочисленных abi имеем следубщие именованования ilp32 or lp64,дабвляя одну букву к стандартным именамподключам float и double регистры.

ilp32: int, long, длиной 32 бита. long long 64 битовый тип, char 8 битовый, и short 16 битовый.
lp64: long 64 длиной , int is a 32 бита длиной. Другие типы совпадают с ilp32.
floating-point ABI RISCV дополннения.
-"" (пустая строка): Никакие floating-point агрументы не передаеются в регистры.
f: 32-bit и меньше floating-point аргументы передаются в регистры. Необходимо F расширение
d: 64-bit и меньше floating-point in аргументы передаются в регистры.Необходимо D расшерение.
Приведем некоторые комбинации ABI и ISA
-arch=rv32imafd -abi=ilp32d: Инструкции для дробных операций генерируются и дробные аргкменты передаются в регистры.
-arch=rv32imac-abi=ilp32: Никакие дробные инструкции не могут генерироваться и никакие дробные агрументы не передаются в регистры.
-arch=rv32imafdc -abi=ilp32: Дробные инструкции могут быть сгенерированы,но никакие дробные агрументы не передаются в регистры.
-arch=rv32imac -abi=ilp32d: Ошибка ,т.к ABI необходимо размещать дробные аргументы,но ISA не опредилл никаких дробных регистров.
Перейдем к кнокреному примеру.Предположим,что имеется простоая C-функция

```
double dmul(double a, double b) { return b * a; }
riscv64-unknown-elf-gcc test.c -arch=rv32imac -abi=ilp32
   dmul:
     mv      a4,a2
     mv      a5,a3
     add     sp,sp,-16
     mv      a2,a0
     mv      a3,a1
     mv      a0,a4
     mv      a1,a5
     sw      ra,12(sp)
     call    __muldf3
     lw      ra,12(sp)
     add     sp,sp,16
     jr      ra

riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32d -o- -S -O3
   dmul:
     fmul.d  fa0,fa1,fa0
     ret

$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32
   dmul:
     add     sp,sp,-16
     sw      a0,8(sp)
     sw      a1,12(sp)
     fld     fa5,8(sp)
     sw      a2,8(sp)
     sw      a3,12(sp)
     fld     fa4,8(sp)
     fmul.d  fa5,fa5,fa4
     fsd     fa5,8(sp)
     lw      a0,8(sp)
     lw      a1,12(sp)
     add     sp,sp,16
     jr      ra

$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32d
   cc1: error: requested ABI requires -march to subsume the 'D' extension
```

Описание структуры программы

Для начала напишем код для создание контейнера,который будет общим для всех остальных контейнеров

```
FROM ubuntu:18.04
ENV RISCV=/opt/riscv32
RUN apt-get update
RUN apt-get install -y git
RUN apt install tree
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  git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
RUN mkdir /riscv-gnu-toolchain/build
WORKDIR  /riscv-gnu-toolchain/build
ARG with_arch
ARG with_abi=abi
RUN ../configure --prefix=/opt/riscv --with-arch=${arch} --with-abi=${with_abi}
RUN make
ENV export PATH=/opt/riscv/bin:$PATH

ARG xlen=32
WORKDIR  /riscv-pk
RUN mkdir build
RUN cd build
RUN ../configure --prefix=/opt/riscv/ --host=riscv${xlen}-unknown-elf
RUN make && make install

RUN cd /riscv-isa-sim
RUN mkdir build
RUN cd build
RUN ../configure --prefix=/opt/riscv --enable-histogram
RUN make && make install
```

В данном случае в аргменте FROM опредляем ОС нашего образа,в данном случае это ubuntu18:04
Переменные ENV необходимы для параметризации образа.
Для инициализации используется configure.sh скрипт

В конце устанавливаем и настраиваем spike симмулятор

Проверка установки

```
$ tree -L 3 -d
.
├── bin
├── include
│   └── gdb
├── lib
│   └── gcc
│       └── riscv32-unknown-elf
├── libexec
│   └── gcc
│       └── riscv32-unknown-elf
├── riscv32-unknown-elf
│   ├── bin
│   ├── include
│   │   ├── bits
│   │   ├── c++
│   │   ├── machine
│   │   ├── newlib-nano
│   │   ├── rpc
│   │   ├── ssp
│   │   └── sys
│   └── lib
│       └── ldscripts
└── share
    ├── gcc-10.1.0
    │   └── python
    ├── gdb
    │   ├── python
    │   ├── syscalls
    │   └── system-gdbinit
    ├── info
    ├── locale
    │   ├── bg
    │   ├── ca
    │   ├── da
    │   ├── de
    ... ...
    │   ├── vi
    │   ├── zh_CN
    │   └── zh_TW
    └── man
```

Тестовый пример

Напишем простую программу и обратим внимание на то,как скомпилируется функция умножения

```
int mult() {
	int a=1000,b=3;
	return a*b;
}

int main() {
	mult();
}
```

export/riscv32/bin/:$PATH
$ riscv64-unknown-elf-gcc -g tst.c -o tst
$ file tst
riscv64-unknown-elf-objdump -d tst

...
00010150 <mult>:
  10150:	fe010113		addi	sp,sp,-32
  10154:	00812e23		sw	s0,28(sp)
  10158:	02010413		addi	s0,sp,32

```
1015c:      3e800793            li      a5,1000
10160:      fef42623            sw      a5,-20(s0)
10164:      00300793            li      a5,3
10168:      fef42423            sw      a5,-24(s0)
1016c:      fec42703            lw      a4,-20(s0)
10170:      fe842783            lw      a5,-24(s0)
10174:      02f707b3            mul     a5,a4,a5
10178:      00078513            mv      a0,a5
1017c:      01c12403            lw      s0,28(sp)
10180:      02010113            addi    sp,sp,32
10184:      00008067            ret
```

Вывод

В результате выполнения проекта была разработан Docker image с риск 5 тулчейном.

Спсиок использованной литературы

1.The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified”, Editors Andrew Waterman and Krste Asanovi´c, RISC-V Foundation, March 2019.
2. https://github.com/riscv/riscv-gnu-toolchain
3.Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов. – М.: ДМК Пресс, 2017. – 354 с.: ил.