

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 RISC-V тулчейном»

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

Минск 2020

Содержание

Введение	3
Анализ предметной области.....	4
Описание структуры программы	14
Спсиок использованной литературы.....	18

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.

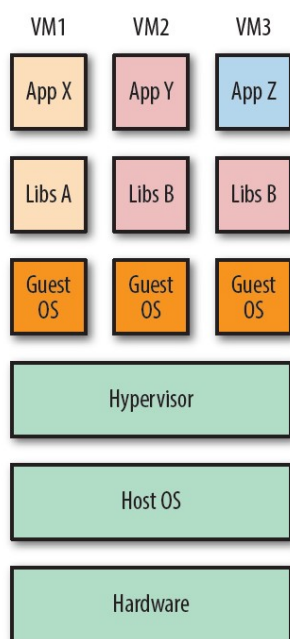


Рис. 1.1. Три виртуальные машины, работающие на одном хосте

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.

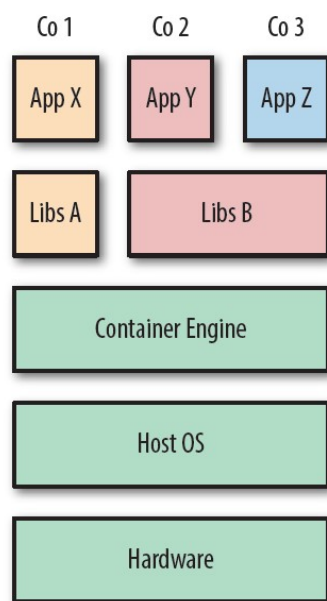


Рис. 1.2. Три контейнера, работающих на одном хосте

Dockerfile

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

Инструкции 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. Основным образом 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 RISC-V

Стандартные расширения `riscv` определены для умножения\деления, атомарных операций, арифметики с одинарной и двойной точностью. Стандартный целочисленный ISA имеет название `I` (начинается с `RV32` или `RV64`) и содержит целочисленные вычислительные инструкции, целочисленные операции чтения\записи, также операции управления потоком программы. Стандартные целочисленные умножения и деления имеют название «`M`», и добавляет инструкции для умножения и деления целых чисел, хранящихся в регистрах. Стандартное расширение для атомарных операций имеет название «`A`», и добавляет инструкции для атомарного чтения\записи\изменения с синхронизацией. Стандартное расширение для чисел с плавающей точкой имеет название «`F`» и добавляет дробные регистры, дробные арифметические инструкции и функции дробного чтения\записи. Стандартное расширение для чисел с двойной точностью, имеющее название «`D`», расширяет регистры с плавающей точкой и добавляет операции чтения\записи для чисел с двойной точностью. Стандартное расширение «`G`» можно описать следующим образом «`G=M+A+F+D`».

«`M`» стандартное 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 верхний 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 зависит 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
								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 <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, <i>Invalid</i> .

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-precision

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

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	W	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	W	offset[4:0]	STORE-FP	

Single-Precision Floating-Point арифметические операции

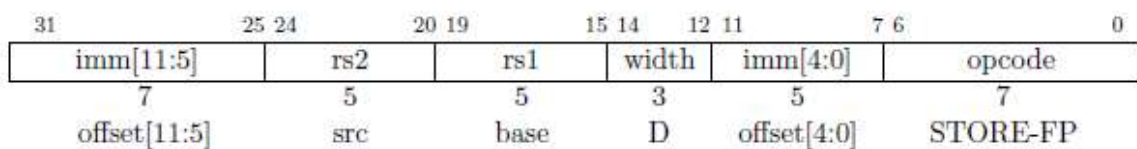
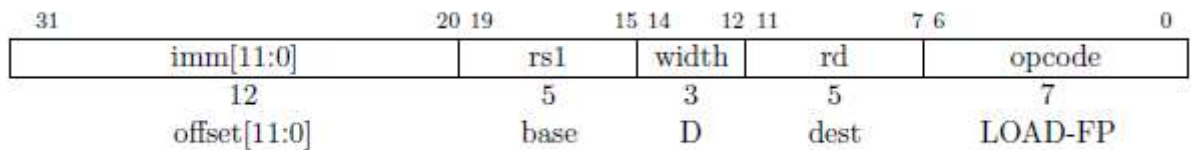
Арифметические команды с дробными числами с одним или двумя операндами используют R-type команд. FADD.S и FMUL.S выполняют сложение и вычитание, соответственно между rs1 и rs2. FSUB.S выполняет вычитание между rs2 и rs1. FDIV.S выполняет деление rs1 на rs2. FSQRT.S извлекает корень из rs1.

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fnt	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

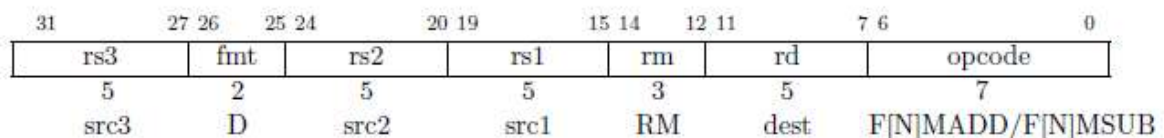
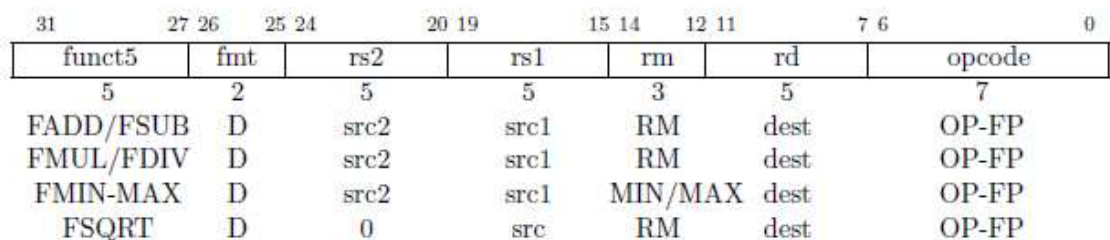
D регистр расширяет 32-битные регистры для floating-point до 64 бит.



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 инструкции вычислений.

Данные операции аналогичны single-precision операциям



«A» стандартное расширение

Стандартный RISV-V ISA имеет relaxed модель памяти, с

барьерными(FENCE) инструкциями для навязывания дополнительных упорядочивающих ограничений. Адресное пространство разделено средой выполнения на 2 поля memory, I/O, барьерные инструкции добавляют функциональность для упорядочивания доступа к одному и обоим адресным пространствам

Для обеспечения эффективной поддержки последовательности каждая атомарная инструкция имеет 2 бита aq rl использующихся для задания дополнительного упорядочивания памяти. Биты доступны одному или обоим

адресным пространствам, в зависимости в какой области памяти доступна атомарная операция.

Если 2 бита не установлены, то никаких ограничений на атомарные операции не накладывается. Если установлен только бит *aq*, то атомарная операция рассматривается как *acquire access*, т.е. никакие следующие операции в данном RISV-V hart не могут выполняться после *asquer operation*. Если только *rl* бит установлен, тогда атомарная операция рассматривается как *release access*, т.е. никакие операции записи не могут выполняться 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

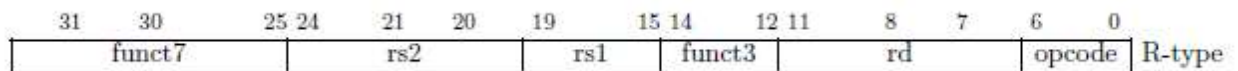
31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		aq	rl	rs2		rs1		funct3		rd		opcode	
5		1	1	5		5		3		5		7	
LR.W/D		ordering		0		addr		width		dest		AMO	
SC.W/D		ordering		src		addr		width		dest		AMO	

Составные атомарные операции над одним или двумя словами выполняются с помощью *load-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 включает в себя адресованное 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* записывается необходимое слово, а в регистр *rd* записывается 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	
AMOODR.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 синхронизации и кодирования с помощью R-типе формата команд.



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

Поддерживаемы операции- обмен,сумма целых чисел,побитовое И,побитовое ИЛИ,побитовое XOR,знаковый или беззнаковый минимум.

Для поддрежания синхронизации АМО команды используют логику,представленную выше.

```

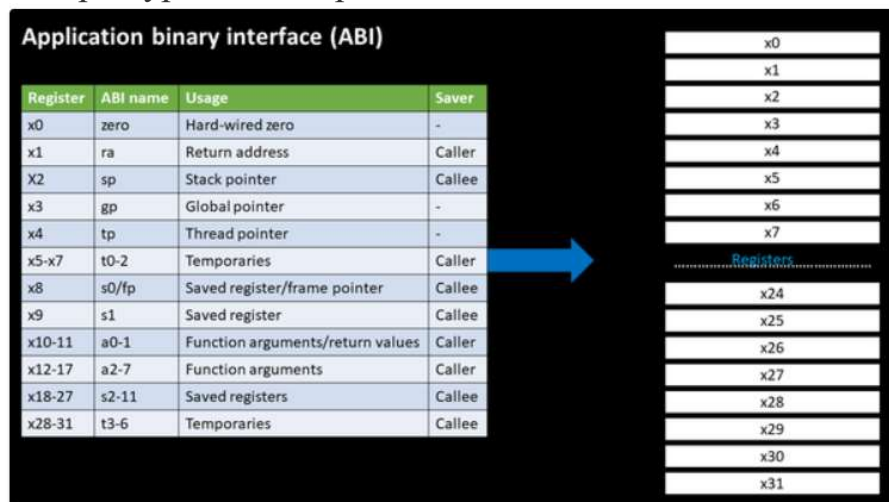
li          t0, 1          # Initialize swap value.
again:
lw          t1, (a0)        # Check if lock is held.
bnez        t1, again       # Retry if held.
amoswap.w.aq t1, t0, (a0)   # Attempt to acquire lock.
bnez        t1, again       # Retry if held.
# ...
# Critical section.
# ...
amoswap.w.rl 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-пользовательские инструкции,вторая-уровень операционной системы.



RISC-V архитектура имеет 32 регистра.прикладной программист имеет доступ к каждому из 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 следующие 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 RISC-V дополнения.

-"" (пустая строка): Никакие floating-point аргументы не передаются 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 расширение.

Приведем некоторые комбинации ABI и ISA

-arch=rv32imafd -abi=ilp32d: Инструкции для дробных операций генерируются и дробные аргументы передаются в регистры.

-arch=rv32imac -abi=ilp32: Никакие дробные инструкции не могут генерироваться и никакие дробные аргументы не передаются в регистры.

-arch=rv32imafdc -abi=ilp32: Дробные инструкции могут быть сгенерированы, но никакие дробные аргументы не передаются в регистры.

-arch=rv32imac -abi=ilp32d: Ошибка, т.к. ABI необходимо размещать дробные аргументы, но ISA не определил никаких дробных регистров.

Перейдем к конкретному примеру. Предположим, что имеется простая 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 Контейнер с тулчейном

FROM ubuntu:18.04

ENV RISCVC=/opt/riscv

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

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

RUN mkdir build


```

RUN cd build
RUN ../configure --prefix=/opt/riscv --with-arch=$1 --with-abi=$2
RUN make && make install
WORKDIR /
VOLUME [ "/tests" ]
RUN clone https://github.com/riscv/riscv-tests
RUN cd riscv-tests
RUN git submodule update --init --recursive
RUN autoconf
RUN ../configure --prefix=/opt/riscv/target
COPY patch.txt /riscv-tests/
RUN git apply patch.txt
RUN make
RUN make install
RUN cp -avf /riscv-tests/benchmarks /tests/
RUN cp -avf /riscv-tests/isa /tests/

```

Код для контейнера с симулятором

```

FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y git
RUN git clone --recursive https://github.com/MIPT-ILab/mipt-mips.git
RUN apt-get install ninja-build
RUN mkdir build && cd build

RUN sudo apt install build-essential
RUN apt-get install -y cmake
RUN apt-get install -y libboost-all-dev
RUN apt-get install -y gcc-10 && apt-get install g++-10
RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-10 100 --
slave /usr/bin/g++ g++ /usr/bin/g++-10 --slave /usr/bin/gcov gcov /usr/bin/gcov-10

RUN cmake /mipt-mips/simulator -G "Ninja"
RUN ninja

COPY "run_tests.sh" "/"
ENTRYPOINT [ "run_tests.sh" ]

```

Проверка установки

Запустим тесты и приведем вывод некоторых из них

```

*** rsort.riscv ***

*****
instrs:      107894
cycles:      137660
IPC:         0.783772
sim freq:    871.925 kHz
sim IPS:     683.39 kips
instr size:  256 bytes
mispredict:  detected on decode stage - 0.0170016%
              detected on branch stage - 0.0136017%
*****
*** towers.riscv ***

*****
instrs:      91093
cycles:      116276
IPC:         0.78342
sim freq:    856.286 kHz
sim IPS:     670.832 kips
instr size:  256 bytes
mispredict:  detected on decode stage - 0.0201394%
              detected on branch stage - 0.0161115%
*****
*** vvadd.riscv ***

*****
instrs:      93529
cycles:      119377
IPC:         0.783476
sim freq:    879.407 kHz
sim IPS:     688.994 kips
instr size:  256 bytes
mispredict:  detected on decode stage - 0.019614%
              detected on branch stage - 0.0156912%
*****

```

Образы на DOCKER HUB

К готовому образу с riscv toolchain arch=64g abi=ilp64d можно получить доступ с помощью

```
docker {params} kirylakulich/rv64g_lp64
```

К готовому образцу с симулятором можно получить с помощью команды

```
docker {params} --volumes-from {toolchain-container} kirylakulich/mips-simulator
```


Вывод

В результате выполнения проекта была разработан Docker image с riscv тулчейном.

Список использованной литературы

1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified”, Editors Andrew Waterman and Krste Asanović, 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.