Санкт-Петербургский государственный университет
Программная инженерия

Нижарадзе Анастасия Тимуровна

Портирование ОСРВ Embox на открытую архитектуру RISC-V

Бакалаврская работа

Научный руководитель:
проф. каф. СП, д.ф.-м.н., проф. А.Н. Терехов

Научный консультант:
ассистент А. П. Козлов

Рецензент:
ведущий инженер ООО "Ланит-Терком" К. К. Смирнов

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Anastasiia Nizharadze

Porting real-time operating system Embox to open-source architecture RISC-V

Graduation Thesis

Scientific supervisor:
professor Andrey Terekhov

Scientific consultant:
assistant Anton Kozlov

Reviewer:
Senior engineer at Lanit-Tercom LLC Kirill Smirnov

Saint Petersburg
2020

Оглавление

Введение

В 80-е годы программное обеспечение (далее по тексту — ПО) с закрытым исходным кодом, распространявшееся на платных условиях, стало очень популярным. Это было связано, в основном, с принятием поправок к закону об авторском праве в США в 1974 году [1], закрепляющих за разработчиком права на интеллектуальную собственность, поставив производителей ПО в один ряд с изобретателями. Это позволило компаниям получать прибыль за разработанные ими программные продукты. Однако в скором времени стало очевидно, что такой подход имеет некоторые недостатки. Закрытость исходных кодов не позволяет пользователям оценить безопасность ПО, а также дополнить его своей функциональностью.

Вследствие указанных недостатков стал набирать популярность альтернативный подход к созданию программных продуктов, при котором сообщество программистов на распределенной основе создают ПО с открытым исходным кодом (Open-Source Software, OSS). Это позволяет снизить стоимость разработки, быстро устранять возникающие ошибки и проверять алгоритм работы ПО, что в целом повышает безопасность использования продукта. Вместе с тем появляется возможность добавления своих решений в исходные коды OSS программ [2].

Принцип открытости исходных кодов оказался применим не только к ПО. Последнее время набирает популярность принцип открытости исходных кодов аппаратного обеспечения (Open-Source Hardware, OSH [3]). До конца 90х годов все процессорные архитектуры были закрытыми. В большинстве случаев производители аппаратного обеспечения не раскрывали детали реализации. Сопроводительная документация к аппаратному обеспечению содержала только минимальную информацию, необходимую для разработки ПО. Данный подход имеет ряд недостатков. Закрытость исходных кодов, прежде всего, не позволяет узнать какие в действительности операции исполняются на процессоре. Например, на конференции Black Hat была опубликована статья, в которой было показано, что процессоры на базе архитектуры x86 исполняют недокументированные инструкции [4]. Также при таком подходе сообщество не может проверить архитектуру процессора на ошибки. Например, с 1995 года [5] многие процессоры компании Intel подвержены уязвимости Meltdown [6]. Это значительно снижает безопасность использования таких процессоров. К тому же, закрытые процессорные архитектуры в большинстве своем несвободны. Это выражается в том, что компании-производители процессоров для использования таких архитектур должны выплачивать отчисления.

OSH подход позволяет обеспечить большую безопасность использования процессоров, т.к. исходные коды процессоров проверяются сообществом. Согласно отчету Scan Report on Open Source Software 2008, в проектах с открытым исходным кодом плотность дефектов за два года может уменьшиться на 16% [2]. Также явно наблюдая исходные коды процессора, можно точно знать какие инструкции могут на нем исполняться.

Одной из таких процессорных архитектур является RISC-V. Это открытый[1] [7] и свободный проект, запущенный в 2010 году в Калифорнийском университете Беркли. Архитектура RISC-V имеет множество преимуществ. Помимо указанных выше, она, в зависимости от выбранной базовой архитектуры набора команд, 32, 64 или 128-бит, поддерживает сжатые инструкции и многое другое [8], благодаря чему архитектура стала популярной. Также архитектура RISC-V является модульной [9]. Это позволяет создавать процессоры, конфигурируя их под определенные задачи, что особенно хорошо для встраиваемых систем.

Есть и другие преимущества в использовании архитектуры RISC-V для встраиваемых систем. В частности, инструкции RISC-V имеют одинаковую длину и упрощенный формат, что делает их более простыми для декодирования и исполнения. Это способствует повышению энергоэффективности процессоров на базе этой архитектуры, а для встраиваемых систем зачастую важно иметь низкое потребление электроэнергии. К тому же процессоры на базе RISC-V в большинстве своем имеют невысокую стоимость вследствие того, что данная архитектура является свободной и не требует отчислений за ее использование, что также очень важно для большинства встраиваемых систем для сохранения невысокой стоимости конечного продукта. Благодаря этому на базе архитектуры RISC-V было выпущено большое количество микропроцессоров и микроконтроллеров [10].

Вследствие большой популярности архитектуры RISC-V, в проекте с открытым исходным кодом по разработке операционной системы реального времени для встраиваемых систем Embox появилась задача портирования ОС Embox на RISC-V.

[1] Исходный код RISC-V https://github.com/riscv

1. Постановка задачи

Цель данной работы — добавить RISC-V в список поддерживаемых архитектур ОСРВ Embox. Для её достижения были поставлены следующие задачи:

- Сделать обзор предметной области
- Реализовать архитектурно-зависимые части ядра:
    - Разметку карты памяти
    - Стартовый код
    - Функции setjmp и longjmp
    - Обработчик прерываний
    - Модуль переключения контекста
    - Функцию vfork
- Реализовать драйвера основных устройств:
    - Контроллера прерываний
    - Устройства таймера
    - Устройства сетевого контроллера
- Протестировать работоспособность полученного решения

2. Обзор

В настоящей главе описаны особенности процессорной архитектуры RISC-V, особенности и структурная схема операционной системы Embox, а также используемые в ходе работы технологии.

2.1 Процессорная архитектура RISC-V

RISC-V — это система команд и процессорная архитектура для микроконтроллеров и микропроцессоров типа RISC (Reduced Instruction Set Computer). Благодаря этому все арифметические операции имеют тип регистр-регистр, все инструкции имеют одинаковую длину, есть всего несколько схожих форматов инструкций (Таблица 1). Разные значения funct7 и funct3 обозначают разные операции. Значение первых семи битов opcode задают формат инструкции. Это позволяет быстрее декодировать и исполнять такие инструкции.

Таблица 1: Форматы инструкций архитектуры RISC-V [9]

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	Формат R
imm[11:0]		rs1	funct3	rd	opcode	Формат I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Формат S
imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1] imm[11]	opcode	Формат B
imm[31:12]				rd	opcode	Формат U
imm[20]	imm[10:1] imm[11]		imm[19:2]	rd	opcode	Формат J

Также RISC-V является модульной архитектурой. Архитектура набора команд (ISA, Instruction Set Architecture) была разделена на небольшую базовую часть и дополнительные расширения. Базовая ISA достаточно проста, чтобы ее можно было использовать для встраиваемых систем, однако она достаточно полная для современного программного стека. Расширения улучшают производительность и обеспечивают поддержку многопроцессорной обработки (Таблица 2). Есть 32, 64 и 128-битные базовые ISA.

Таблица 2: Стандартные законченные расширения [9]

Название	Описание	Версия
M	Стандартное расширение для перемножения и деления целочисленных	2.0
A	Стандартное расширение для атомарных инструкций	2.0
F	Стандартное расширение для чисел с плавающей запятой одинарной точности	2.0
D	Стандартное расширение для чисел с плавающей запятой двойной точности	2.0
Q	Стандартное расширение для чисел с плавающей запятой четырехкратной точности	2.0
C	Стандартное расширение для сжатых инструкций	2.0

Стандартное расширение «C» позволяет уменьшить статический и динамический размер кода, добавляя короткие 16-битные инструкции для общих операций. Данное расширение в среднем позволяет сжать 10% от всех инструкций и позволяет достичь степени сжатия в 5% [11].

Еще одной особенностью архитектуры RISC-V является разделение привилегий [12]. В любое время каждый аппаратный поток исполнения RISC-V (hart, hardware thread) работает с некоторым уровнем привилегий, установленным в одном или нескольких CSR (Control and Status Register, регистры управления и состояния). Всего есть три таких уровня: машинный, пользовательский и уровень супервизора. Уровни привилегий используются для обеспечения защиты между различными группами сегментов программного стека, а попытки выполнить операции, не разрешенные текущим режимом привилегий, вызовут исключение.

Также RISC-V обеспечивает поддержку косвенных переходов и вызовов процедур, что позволяет компиляторам поддерживать PIC (Position-Independent Code). Это позволяет вызывать функции и обращаться к глобальным переменным динамических библиотек через таблицу косвенного обращения GOT (Global Offset Table) и адреса текущей инструкции. Поддержка PIC важна для современных архитектур, потому что она позволяет вызывать, исполнять процедуры и получать доступ к данным, которые загружаются в системную память по адресам, не указанным программным кодом и не определяемым до тех пор, пока программный код не скомпилирован, скомпонован и загружен в системную память [13]. Без поддержки PIC для загрузки таких адресов используется подход, называемый релокацией во время загрузки (load-time relocation), при использовании которого увеличивается время загрузки программ и объем используемой памяти [14].

Немаловажным свойством рассматриваемой архитектуры также является возможность классической виртуализации без необходимости динамической двоичной трансляции. Архитектура называется классически виртуализируемой, если выполняются условия первой теоремы, сформулированной Геральдом Попеком (Gerald Popek) и Робертом Голдбергом (Robert Goldberg) в 1974 году в

работе «Formal Requirements for Virtualizable Third Generation Architectures» [15]. Теорема гласит о том, что построение эффективной виртуальной машины возможно, если множество служебных инструкций является подмножеством привилегированных. Таким образом, разработчики RISC-V гарантируют возможность создания эффективного монитора виртуальных машин для данной архитектуры.

Также архитектура RISC-V поддерживает стандарт IEEE 754-2008 в стандартных расширениях для работы с числами с плавающей запятой. Аппаратная поддержка этого стандарта позволяет уменьшить время исполнения операций над числами с плавающей запятой вследствие того, что программные реализации как правило в 100-1000 раз медленнее справляются с указанным типом задач [16].

2.2 Операционная система Embox

Embox (Essential toolbox for embedded development) — свободная операционная система реального времени, разрабатываемая для встраиваемых систем.

Исходный код ОСРВ Embox представляет собой набор модулей, что позволяет включать в сборку только те части системы, которые нацелены на решение конкретной задачи. Благодаря модульности архитектурно-зависимые части ОС явно отделены от остального ядра, что сильно упрощает процесс портирования на новые архитектуры.

Модульность и хорошее структурирование достигается за счет системы сборки Mybuild. В то же время Mybuild является языком, на основе которого происходит сборка. На нем описывается конфигурация как отдельных модулей, так и всей системы. Описания модулей, кроме прочего, обладает свойством наследования, что помогает создавать различные реализации одних и тех же интерфейсов, что упрощает процесс портирования.

Помимо этого, для разных семейств архитектур и разных процессоров существуют большое количество разных шаблонов сборки. В каждом шаблоне есть несколько конфигурационных файлов, которые помогают системе Mybuild правильно собрать разные части Embox в один объектный файл. В файле `build.conf` хранится информация о том, какой компилятор и какие флаги использовать при компиляции, это — конфигурация сборки; в файле `lds.conf` содержится информация о компоновке, разметке регионов и секций по регионам, это — конфигурация компоновщика; файл `mods.conf` хранит информацию о модулях, включенных в сборку, например модуль файловой системы или таймера, это — конфигурация модулей.

Для того, чтобы собрать проект, сначала необходимо загрузить нужный шаблон. Так можно создать минимальную сборку, необходимую для устройств с ограниченными ресурсами. А также, благодаря статической сборке системы, функциональность, не включённая в образ, не исполняется, что повышает безопасность системы.

Embox поддерживает множество архитектур таких как ARM, x86, SPARC, Microblaze, MIPS, PPC, E2k.

Также Embox является POSIX-совместимой операционной системой [17]. Это упрощает портирование приложений, опирающийся на данный стандарт.

2.3 Структурная схема Embox

Embox, как и любая современная операционная система, поддерживает вытесняющую многозадачность, управление процессами, подсистему времени, подсистему ввода/вывода. Структурную схему ОС Embox можно представить в виде UML диаграммы компонентов, представленной на Рисунке 1. Как видно из рисунка, все архитектурно-зависимые части Embox можно разбить на две компоненты: драйверы и архитектурно-зависимая часть ядра. Остальная часть ядра опирается на слой указанных аппаратных абстракций. На разных платформах этот слой предоставляет одинаковые интерфейсы, которыми пользуются остальные высокоуровневые части ОС. Чтобы портировать Embox на новую архитектуру, необходимо реализовать каждый модуль этих компонент.

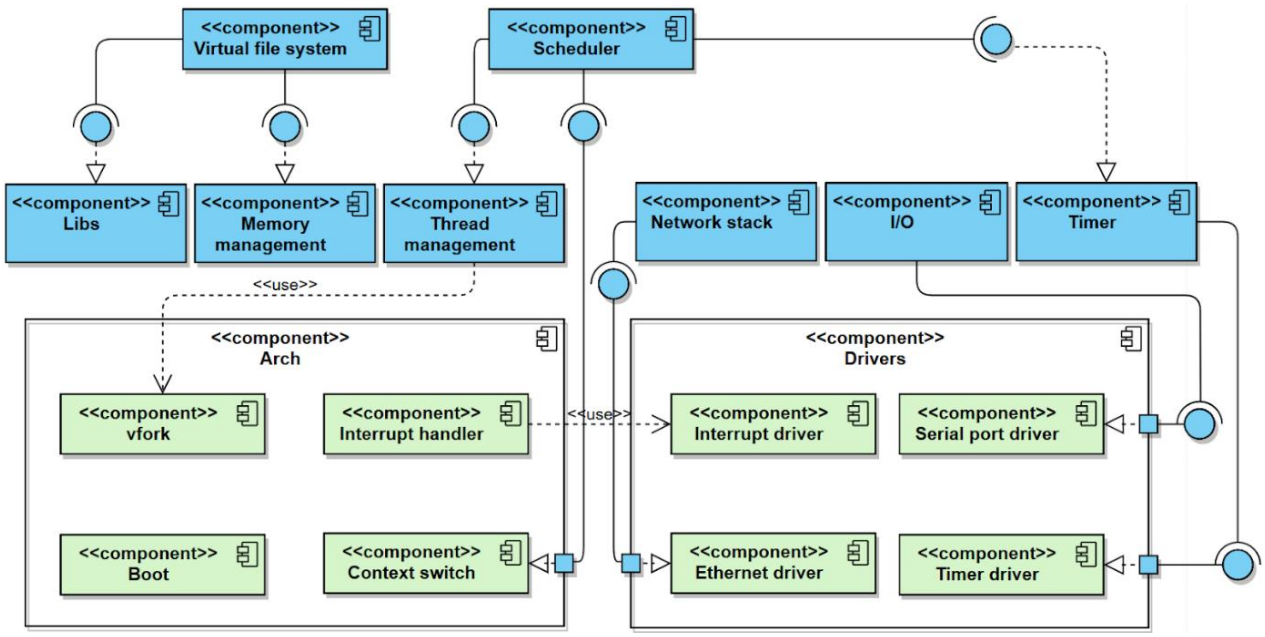


Рис. 1: Структурная схема операционной системы Embox

Помимо указанных в структурной схеме, для разных сборок ОС Embox также реализован ряд других драйверов, таких как, например, драйвер USB порта. Однако в рамках базового портирования, как указано в постановке задачи, научным руководителем было решено реализовать только основные драйвера устройств, без которых невозможно нормальное функционирование системы.

2.4 Используемые технологии

Операционная система Embox реализована в большинстве своем на языке программирования Си. Некоторые низкоуровневые модули реализованы на ассемблере. Таким образом, реализация архитектурно-зависимых частей Embox будет реализована на этих двух языках.

Портирование проводилось на эмулятор реальных плат, потому что на момент выполнения работы аппаратного обеспечения не было в наличии. В качестве эмулятора был выбран QEMU, т.к. он поддерживается ОС Embox. В качестве эмулируемой платы выбрана система на чипе HiFive Unleashed FU540-C000 SoC из-за наличия у нее сетевого устройства.

Большинство поддерживаемых ОС Embox процессоров — 32-битные. Поэтому базовое портирование было решено выполнить на 32-битную версию архитектуры RISC-V.

Для встраиваемых систем разделение исполнения на уровни привилегий не так важны, поэтому было решено реализовать поддержку только машинного режима, однако таким образом, чтобы в будущем была возможна поддержка не только машинного уровня привилегий, но и пользовательского, и уровня супервизора.

Микросхема FU540-C000 имеет 4+1 аппаратных потока исполнения (4 основных и один, который запускается первым и производит первичную настройку процессора), но для упрощения процесса портирования было решено поддерживать исполнение только на одном.

В рамках базового портирования также было решено не поддерживать стандартных расширений архитектуры RISC-V, однако в будущем имеет смысл рассмотреть вариант поддержки сжатых инструкций, т.к. это позволяет экономить объем используемой памяти, что часто оказывается важным для встраиваемых систем.

3. Реализация

Процесс портирования ОС Embox на некоторую архитектуру можно поделить на несколько основных этапов по модулям [18].

3.1 Карта памяти

Карта памяти определяет положение регионов памяти, т.е. расположение RAM, ROM, I/O и т.д. В архитектуре RISC-V нет строгого определения карты памяти, разметку регионов задает сам разработчик процессора, разработчик системного ПО может посмотреть ее в документации на конкретную платформу. Таблица 3 показывает разметку регионов на машине QEMU `sifive_u`, которая моделирует HiFive Unleashed FU540-C000 SoC. Таблица была выведена из исходных кодов эмулятора QEMU версии 4.2.0.[2].

Таблица 3: Карта памяти эмулируемой QEMU машины sifive_u

Address range	Description
00001000 - 00011fff	rom: riscv.sifive.u.mrom
02000000 - 0200ffff	i/o: riscv.sifive.clint
08000000 - 09ffffff	ram: riscv.sifive.u.l2lim
0c000000 - 0fffffff	i/o: riscv.sifive.plic
10000000 - 10000fff	i/o: riscv.sifive.u.prci
10010000 - 1001001f	i/o: riscv.sifive.uart
10011000 - 1001101f	i/o: riscv.sifive.uart
10070000 - 10070fff	i/o: riscv.sifive.u.otp
10090000 - 100907ff	i/o: enet
100a0000 - 100a0fff	i/o: riscv.sifive.u.gem-mgmt
20000000 - 2fffffff	ram: riscv.sifive.u.flash0
80000000 - 87ffffff	ram: riscv.sifive.u.ram

Информацию о RAM и ROM, а также о том, какие сегменты и в каких регионах будут располагаться, были перенесены в файл компоновщика (Linker Script File). Адреса I/O подсистем описываются в конфигурации модулей, потому что в некоторых архитектурах с подключением новых устройств адреса могут меняться.

Для разметки регионов RAM и ROM в специальный файл `lds.conf` были добавлены адреса начала регионов и их размер. Из дерева памяти видно, что тип памяти ROM — MROM (Mask ROM), то есть её содержимое программируется

[2] Исходный код машины sifive_u https://github.com/qemu/qemu/blob/master/hw/riscv/sifive_u.c

производителем интегральной схемы; поэтому было решено переместить сегменты кода и данных в RAM.

3.2 Стартовый код

Стартовый код — код, выполняющий первичную инициализацию системы, такую, как инициализация регистров, инициализация режима процессора, включение прерываний и т.д. В конце загрузчик передает управление функции инициализации ядра ОС. Стартовый код необходимо разместить по стартовому адресу. В архитектуре RISC-V стартовый адрес как таковой не определен, в машине QEMU `sifive_u` это адрес начала ROM, в которой расположена инструкция прыжка на начало RAM. Т.к. сегмент кода `.text` был размечен на RAM, загрузчик находится по адресу 0x80000000.

Реализованный в рамках данной работы загрузчик настраивает регистр global pointer, включает прерывания, обрабатывает потоки исполнения аппаратного уровня (`hart` — hardware thread), устанавливает адрес таблицы прерываний, обнуляет `.bss` и передает управление ядру.

Регистр Global Pointer (`gp`) — это решение для оптимизации доступа к памяти в пределах одной области размером 4 Кбит. Компоновщик использует адрес `gp` для сравнения адресов памяти и, если адреса находятся в пределах диапазона +/- 2 Кбит от адреса `gp`, заменяет абсолютную адресацию на относительную `gp`-адресацию, что делает код более эффективным. Этот процесс также называется relaxing. Регистр `gp` должен быть загружен во время запуска и не должен изменяться позже. Область 4 Кбит может находиться где угодно в адресуемой памяти, но для эффективности оптимизации она должна предпочтительно охватывать наиболее интенсивно используемую область RAM. Таким образом, определение значения `gp` должно быть помещено прямо перед разделом `.sdata`.

Размер области равен 4 Кбит, поскольку непосредственные значения RISC-V являются 12-битными знаковыми значениями, которые составляют +/- 2048 в десятичном виде или +/- 0x800 в шестнадцатеричном; так как значения могут быть положительными и отрицательными, `gp` должен указывать на середину региона. Размер непосредственных значений равен 12 битам, потому что RISC-V инструкции для работы с такими значениями выделяют для них 12 бит (Таблица 1).

Включение прерываний происходит за счет установки бита «interrupt enable bit» в регистре `mstatus`. Более подробное описание устройства работы прерываний приведено в главе 3.4.

Во всех версиях QEMU раньше 4.2.0. для машины `sifive_u` не было поддержки аппаратной многопоточности. Поэтому в стартовом коде изначально не было специальной обработки `hart`. С версии QEMU 4.2.0. при запуске ОС Embox каждый символ отладочного вывода дублировался. В связи с этим появилась задача выявления причины этого артефакта и его устранения. Выявление текущего аппаратного потока исполнения происходит с помощью специального регистра `mhartid`. Как уже было сказано ранее, в рамках данной задачи рассматривается исполнение на одном `hart`. Все остальные отключаются загрузчиком. После отключения всех аппаратных потоков исполнения, кроме первого, ошибка более не наблюдалась.

3.3 Отладочный вывод

Основная функция отладочного вывода — возможность посимвольного вывода на доступное для разработчика устройство вывода. Самым базовым устройством является UART (Universal Asynchronous Receiver/Transmitter).

Драйвер последовательного порта для платформы SiFive уже был реализован в Embox. Однако во всех версиях QEMU раньше 4.2.0 разметка карты памяти машины `sifive_u` была реализована отличным от документации к системе на чипе HiFive Unleashed FU540-C000 образом[3]. Поэтому с версии 4.2.0 драйвер стал работать некорректно. После исправления базового адреса стало возможно обращаться ко всем определенным регистрам, которые можно увидеть в Таблице 4. С помощью регистров `txctrl` и `rxctrl` выставляются флаги готовности принять или отправить символ, а регистры `txdata` и `rxdata` содержат сами передаваемые символы.

Таблица 4: Регистры UART

Offset	Name	Description
0x00	txdata	Transmit data register
0x04	rxdata	Receive data register
0x08	txctrl	Transmit control register
0x0c	rxctrl	Receive control register
0x10	ie	UART interrupt enable
0x14	ip	UART interrupt pending
0x18	div	Baud rate divisor

[3] Bin Meng commit “riscv: sifive_u: Update UART base addresses and IRQs” https://git.qemu.org/?p=qemu.git;a=commit;f=hw/riscv/sifive_u.c;h=4b55bc2b5f7ff065da5d2b813ee5153c598d3764

API драйвера содержит методы `setup`, `getc` и `putc`, которые делают начальную настройку, принимают символ и отправляют символ соответственно.

В процессе портирования было проведено тестирование работоспособности данного драйвера. При запуске ОС Embox в терминал выводится отладочная информация о сборке модулей и пройденных модульных тестах. После исправления базового адреса вся отладочная информация выводилась корректно, никаких ошибок выявлено не было.

Далее была написана утилита, проверяющая работоспособность драйвера в стрессовых условиях. При потоковом выводе символов через последовательный порт при неправильной реализации драйвера некоторые символы могут не успевать передаваться и, как следствие, теряться. Реализованная утилита выводит 100 000 символов от нуля до девяти в терминал, затем проверяет, что последовательность `0123456789` встречается в выводе 10 000 раз, тем самым подтверждая корректность работы драйвера.

3.4 Функции setjmp и longjmp

Функции `setjmp()` и `longjmp()` в языке Си используются для мгновенного выхода из нескольких активаций (вызовов) функций, управление передается в заранее определенное место другой функции. Функция `setjmp()` динамически устанавливает цель, в которую позже будет передаваться управление, а `longjmp()` выполняет передачу выполнения.

Реализованная в ходе работы функция `setjmp()` сохраняет различную информацию о вызываемой среде: указатель стека, указатель счетчика инструкций, маски сигналов и остальные регистры, сохраняемые вызываемой функцией (callee-saved) в специальный буфер для последующего использования `longjmp()`.

Написанная мною функция `longjmp()` использует информацию, сохраненную в буфере для передачи управления обратно к точке вызова `setjmp()` и восстанавливает состояние стека таким, какой он был на период вызова `setjmp()`. После успешного выполнения `longjmp()` исполнение продолжается, как если бы `setjmp()` вернулся во второй раз.

Данные функции используются ОС Embox, помимо прочего, для запуска модульных тестов, поэтому они были реализованы одними из первых. Для тестирования этих функций в сборку был добавлен соответствующий модуль тестов. Тесты проверяют корректный выход из нескольких рекурсивно вызванных функций. Также исполнение самих модульных тестов гарантирует работоспособность этих функций.

3.5 Базовая подсистема прерываний

Прерывание — сигнал к процессору, генерируемый на уровне аппаратного или программного обеспечения. Оно оповещает процессор о происхождении события, которое требует немедленного внимания и обработки. При этом исполнение текущего кода прерывается, сохраняется контекст потока и вызывается специальный обработчик прерываний. Обычно прерывания используются для реализации системного таймера, сигналов управления процессами операционной системы и т.д. Также прерывания необходимы для реализации передачи данных с помощью UART и Ethernet.

Базовую подсистему прерываний можно разделить на две основные части: обработчик прерываний и драйвер контроллера прерываний.

3.5.1 Обработчик прерываний

Когда происходит прерывание, процессор передает управление обработчику исключений, адрес которого он берет из вектора прерываний (Таблица 7). Обработчик определенным образом обрабатывает прерывание и затем возвращает управление обратно туда, где произошло прерывание.

В архитектуре RISC-V для срабатывания прерываний необходимо выполнение следующих условий:

1) в CSR регистре `mie` выставлен бит типа прерывания (machine interrupt enable, т.е. конкретное прерывание не маскировано),

2) прерывания включены глобально в CSR регистре `mstatus` (бит `mstatus.MIE` выставлен в 1).

В отличие от обычных регистров, взаимодействие с которыми происходит с помощью инструкций load и store, для чтения и записи в CSR регистры используются специальные инструкции `csrr` (CSR read) и `csrw` (CSR write), которые можно использовать только в привилегированных уровнях исполнения.

Когда происходит исключение, в специальном CSR регистре `mcause` размещается тип и причина исключения. Возможные типы и причины исключений, а также структура регистра `mcause` приведены в Таблице 5 и 6.

Таблица 5: Структура CSR регистра `mcause`

31	30 … 0
Interrupt	Exception Code (**запись и чтение только корректных значений**)
1	31

Таблица 6: Значения регистра `mcause` после срабатывания прерывания

Interrupt	ExceptionCode	Описание
1	1	Программное прерывание в режиме супервизора
1	3	Машинное программное прерывание
1	5	Прерывание таймера в режиме супервизора
1	7	Машинное прерывание таймера
1	9	Внешнее прерывание в режиме супервизора
1	11	Машинное внешнее прерывание
1	>= 16	*Доступно для платформенного использования*
0	0	Адрес инструкции смещен
0	1	Ошибка доступа к инструкции
0	2	Неверная инструкция
0	3	Точка останова
0	4	Адрес загрузки смещен
0	5	Ошибка доступа к загрузке
0	6	Store/AMO адрес смещен
0	7	ошибка доступа к Store/AMO
0	8	Вызов среды из U-режима
0	9	Вызов среды из S-режима
0	11	Вызов среды из M-режима
0	12	Ошибка страницы инструкции
0	13	Ошибка загрузки страницы
0	15	Ошибка страницы Store/AMO
0	24-31	*Доступно для пользовательской настройки*

Если значение поля `MODE` в регистре `mtvec` равно `Vectored`, при срабатывании исключения в регистр счетчика инструкций `pc` установится значение поля `BASE`, тогда как при срабатывании прерывании в регистр `pc` установится значение поля `BASE + (4 * ExceptionCode)`. Если `MODE = Direct`, то для любого вида исключений и прерываний адрес первичного обработчика будет одним и тем же, в регистр `pc` устанавливается значение `BASE` и управление передается по этому адресу (Таблица 7).

Таблица 7: Структура CSR регистра `mtvec`

31 … 2	1 … 0
BASE[31:2] (**запись корректных значений**)	MODE (**запись корректных значений**)
30	2

Для упрощения реализации в Embox `MODE = Direct`. Следовательно, необходимо по этому адресу разместить первичный обработчик исключений и прерываний, который будет считывать тип и код ошибки и вызывать соответствующий обработчик.

Сам первичный обработчик сначала сохраняет все регистры, чтобы не испортить состояние контекста прерванного процесса, затем в зависимости от того, пришло прерывание или исключение, вызывает обработчик прерываний

или обработчик исключений, затем восстанавливает значения всех регистров и передает управление обратно процессу, который был прерван (Листинг 1).

```
trap_handler:
   SAVE_ALL        //сохранение контекста
   csrr t6, mcause       //считывание регистра mcause
   srli t6, t6, 31       //считывание бита Interrupt
   beqz t6, exception_handler //вызов обработчика исключений
   mv   a0, sp
   jal  interrupt_handler     //вызов обработчика прерываний
   RESTORE_ALL    //восстановление контекста
   mret
```

Листинг 1: Первичный обработчик прерываний и исключений

Обработчик прерываний ставит все необходимые блокировки, выключает прерывания с помощью методов `ipl` (interrupt priority level), считывает номер прерывания и передает его архитектурно - независимому обработчику, который смотрит в таблицу прерываний и по номеру вызывает нужный обработчик.

Также в этот модуль входит и реализация `ipl`. Он уже был реализован в Embox. Содержит важную функциональность работы с прерываниями, такую как включение и выключение прерываний в зависимости от приоритета.

Для того, чтобы протестировать работоспособность базовой подсистемы прерываний, необходимо проверить корректность обработки каждого вида прерывания, т.е. внешнего, программного и прерывания таймера. Таким образом, пройденные тесты таймера и системных вызовов, а также успешный прием сетевых пакетов через Ethernet гарантируют работоспособность прерываний.

3.5.2 Драйвер контроллера прерываний

В ОС Embox драйвер контроллера прерываний маскирует прерывания по его номеру. В RISC-V, как и во многих других архитектурах, имеется регистр маскирования прерываний `mie`, который позволяет выборочно включать и отключать аппаратные прерывания. Каждый сигнал прерывания связан с битом в этом регистре (Таблица 8). Биты MEIE, MSIE и MTIE отвечают за маскирование в машинном режиме внешних, программных и прерываний таймера соответственно.

Таблица 8: Структура CSR регистра `mie`

15 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
4	1	1	1	1	1	1	1	1	1	1	1	1

API драйвера состоит из функций инициализации, включения и отключения прерывания. Соответственно, была реализована функциональность, позволяющая по номеру прерывания включать или отключать это прерывание.

3.6 Драйвер устройства таймера

Принцип работы таймера в архитектуре RISC-V схож с таковыми в других архитектурах. Есть 2 регистра для работы таймера: `mtime` и `mtimecmp`. Во 2ой устанавливается значение, а первый инкрементирует свое значение, начиная с 0, с постоянной частотой. Как только значение регистра `mtime` становится больше либо равно значению в регистре `mtimecmp`, происходит прерывание таймера. Оно сработает только в том случае, если в регистре mie установлен в единицу бит MTIE.

Таким образом, был реализован драйвер, который позволяет устанавливать временной интервал, после которого будет возникать прерывание, означающее, что установленное время прошло.

Т.к. архитектура выбрана 32-битная, а регистры `mtime` и `mtimecmp` — 64 битные, возникает особенность записи нового значения регистра `mtimecmp`. Невозможно одновременно переписать оба слова этого регистра, поэтому может возникнуть ситуация, при которой значение регистра `mtimecmp` будет меньше `mtime`, что вызовет ненужное прерывание. Чтобы прерывание не произошло раньше времени, необходимо [11]:

1) записать старшие 32 бита регистра `mtimecmp` единицами; тогда значение этого регистра будет точно не меньше, чем предыдущее значение
2) записать в младшие 32 бита регистра `mtimecmp` младшие 32 бита нового значения; тогда значение этого регистра будет точно не меньше нового значения
3) записать в старшие 32 бита регистра `mtimecmp` старшие 32 бита нового значения; тогда значение этого регистра и будет новым значением

Пример реализации данного алгоритма приведен в Листинге 2.

```
// новое значение в регистрах a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)     // Не меньше старого значения.
sw a1, 4(t1)     // Не меньше нового значения.
sw a0, 0(t1)     // Новое значение.
```

Листинг 2: Смена значения регистра `mtimecmp` [11]

API драйвера содержит методы инициализации, первичной настройки и обработчика прерывания. Реализованные метод инициализации выделяет память для таймера и добавляет номер прерывания и его обработчик в общий пул прерываний, метод первичной настройки устанавливает начальное значение регистра `mtimecmp`, обработчик прерываний меняет значение регистра и вызывает архитектурно - независимый обработчик.

Для тестирования работоспособности драйвера устройства таймера в сборку были добавлены модульные тесты, проверяющие разные сценарии срабатывания таймера:

- Запуск двух программных таймеров одновременно,
- Проверка корректности отсчета времени таймером,
- Запуск большого количества программных таймеров,
- Запуск одного и того же программного таймера большое количество раз

и другие.

3.7 Переключение контекста

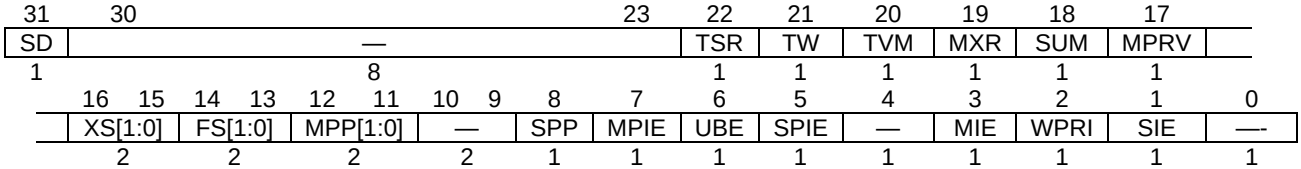
Контекст процессора — это структура данных, которая хранит состояние процессорного ядра. Сохранять контекст необходимо, если мы хотим иметь возможность исполнения на нескольких потоках. То есть, если ОС необходимо прекратить выполнение текущего потока и начать исполнять следующий, она вызывает модуль переключения контекстов вычислительного ядра. Таким образом, необходимо реализовать сохранение контекста и его переключение. Сохранение контекста — это сохранение значений всех доступных регистров в некоторой структуре, а переключение — это восстановление старого контекста другого процесса, то есть возвращение состояния регистров из значений регистров структуры контекста.

В первую очередь были определены регистры, которые необходимо сохранять как контекст процессора, то есть такие регистры, которые однозначно задают состояние процессора. Это регистры, сохраняемые вызываемой функцией (Таблица 9), а также CSR `mstatus` (Таблица 10). Остальные регистры сохраняются в вызывающей функцией (caller-saved registers). CSR регистр `mstatus` хранит информацию о том, включены ли прерывания глобально (биты `MIE` и `SIE`), какие уровни привилегий реализованы (биты `MPP` и `SPP`), привилегирован ли доступ к памяти (биты `MPRW`, `MXR` и `SUM`), какой порядок байтов (endianness) доступа к памяти (биты `MBE`, `SBE`, и `UBE`), а также поддерживается ли перехват операций управления виртуальной памятью супервизора (биты `TWM`, `TV` и `TSR`) и другое. В рамках моей работы сохранение данного регистра не обязательно, однако если в будущем будет решено в ОС Embox поддерживать уровни привилегий, значения данного регистра будет необходимо для корректной обработки потока исполнения.

Таблица 9: описание регистров в архитектуре RISC-V

Регистр	Имя, определенное в ABI	Описание	Сохраняющая функция
`x0`	`zero`	Значение этого регистра всегда ноль	—
`x1`	`ra`	Адрес возврата	Вызывающая
`x2`	`sp`	Указатель на стек	Вызываемая
`x3`	`gp`	Глобальный указатель	—
`x4`	`tp`	Указатель на поток	—
`x5-7`	`t0-2`	Временные регистры	Вызывающая
`x8`	`s0/fp`	Сохраняемые регистры / указатель на фрейм	Вызываемая
`x9`	`s1`	Сохраняемые регистры	Вызываемая
`x10-11`	`a0-1`	Аргументы функции / возвращаемые значения	Вызывающая
`x12-17`	`a2-7`	Аргументы функции	Вызывающая
`x18-27`	`s2-11`	Сохраняемые регистры	Вызываемая
`x28-31`	`t3-6`	Временные регистры	Вызывающая

Таблица 10: структура CSR регистра `mstatus`



Далее была реализована функция инициализации контекста, которая выделяет память под структуру контекста и заполняет ее базовыми значениями. Наконец, было реализовано переключение контекста. Эта функция, которой в качестве аргументов передаются указатели на структуры контекста, сохраняет

текущие значения регистров в одну структуру, и восстанавливает значения из другой.

Для проверки работы функций инициализации и смены контекста были запущены соответствующие модульные тесты, в которых инициализируется, а затем большое количество раз переключается контекст трех функций, в которых выполняются логические операции и проверяется корректность значений переменных, измененных в этих функциях. Это необходимо, чтобы удостовериться в том, что функция смены контекста не только корректно сохраняет и восстанавливает контекст, но и не перезаписывает случайным образом стековый кадр ни одной из функций.

3.8 Функция vfork

Для управления процессами необходим механизм создания дочернего процесса, т.е. функция `vfork`. Эта функция создает копию вызывающего процесса в том же адресном пространстве. Он используется для создания новых процессов без копирования таблиц страниц родительского процесса. Вызывающий поток приостанавливается до тех пор, пока дочерний процесс не завершится.

Для создания копии процесса необходимо сохранять значения всех регистров, кроме регистра возвращаемого функцией значения, и затем копировать их для дочернего процесса. В отличие от переключения контекста, в этой функции важно сохранять все регистры, потому что создается копия структуры процесса.

Это является единственной архитектурно-зависимой частью данной функции, поэтому после сохранения состояния всех регистров вызывается тело самой функции `vfork`.

Модульные тесты, запущенные для проверки работоспособности данной функции, проверяют, может ли дочерний процесс завершиться, поменять значение переменной, инициализированной в родительском процессе, вызвать функцию exec.

3.9 Драйвер устройства сетевого контроллера

Для реализации драйвера сетевого контроллера были реализованы функции инициализации, отправки пакета, установки MAC адреса, приема пакета и отправки в сетевой стек. Драйвер был реализован для сетевой карты Cadence GEMGXL Gigabit Ethernet Controller, которая используется в HiFive Unleashed FU540-C000 SoC.

Обмен пакетами сетевого устройства с драйвером происходит с помощью DMA (Direct Memory Access). Драйвер создает 2 кольцевых буфера дескрипторов на прием и отправку пакетов, в которых определено 2 поля по 32 бита. В первом хранится указатель на сам пакет, второй содержит информацию о размере пакета и конфигурационную информацию, такую, как флаг использования дескриптора, флаг последнего элемента кольца и т.д. (Рисунок 5).

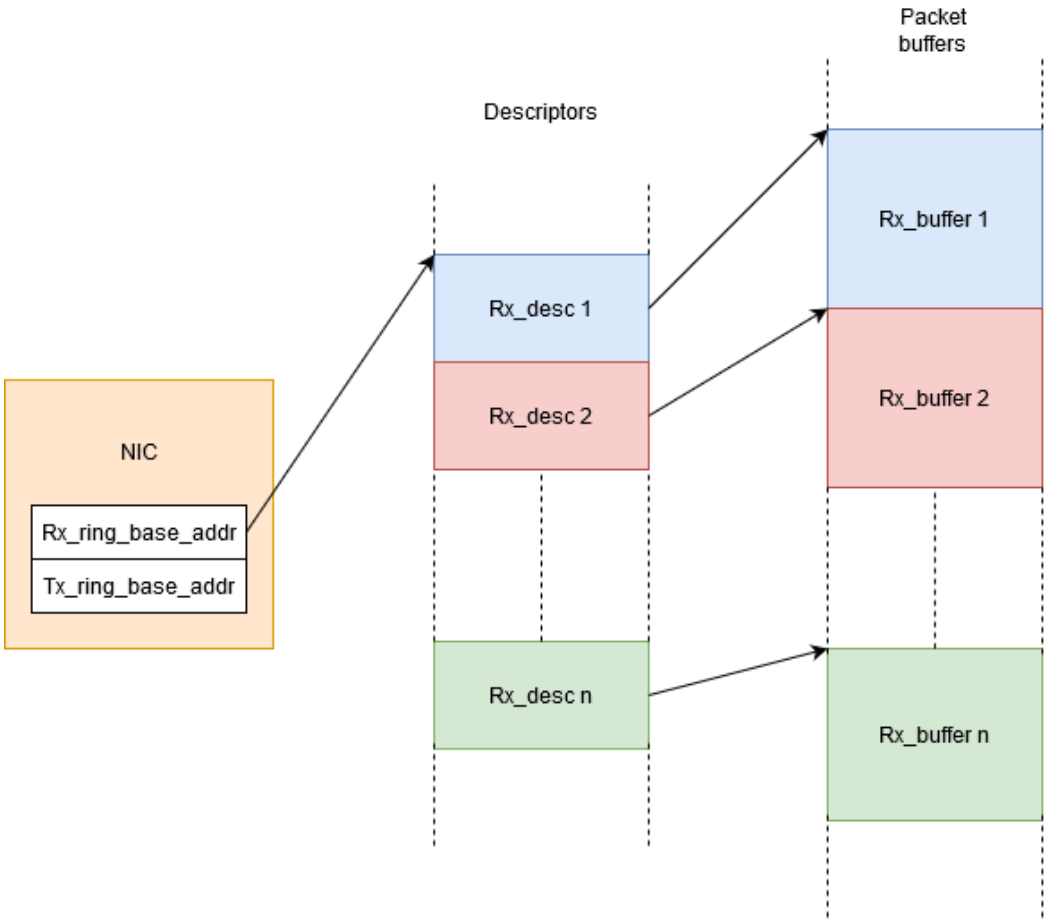


Рис. 5: Устройство DMA

При инициализации драйвера необходимо считать конфигурационные регистры сетевого устройства, чтобы узнать, используется буферизация пакетов и какое количество очередей пакетов поддерживается. Затем, необходимо создать структуру `net_device` для данного устройства и добавить обработчик прерываний в общий список всех обработчиков.

Функция `open` должна подготовить сетевое устройство к обмену пакетами. Для этого в ней создаются кольца дескрипторов, причем все дескрипторы кольца `Tx` (transmit) необходимо отметить как используемые (флаг `USED`), базовые адреса колец необходимо записать в регистры `RBQP` и `TBQP` устройства, далее в специальные регистры `SA1B` и `SA1T` необходимо

установить MAC-адрес устройства. Далее необходимо обнулить все флаги прерываний и статуса отправки и приема пакета: `TSR`, `RSR`, `IDR` и `ISR`.

Чтобы отправить пакет, необходимо заполнить свободный дескриптор. Поле адреса необходимо заполнить физическим адресом буфера, содержащего сетевой пакет; размер пакета и необходимые конфигурационные флаги необходимо записать во второе поле. Далее необходимо убрать флаг `USED`, чтобы сетевое устройство смогло иметь доступ к дескриптору, а затем в регистр `NCR` (Network Control Register) добавить флаг `TSTART`, что будет сигнализировать устройству о том, что необходимо отправить пакет.

Когда сетевое устройство принимает пакет, оно таким же образом заполняет дескриптор, а затем вызывает прерывание. В обработчике прерываний нам необходимо создать структуру `sk_buff`, проверить, не используется ли дескриптор устройством, заполнить `sk_buff` данными из дескриптора и передать далее в сетевой стек с помощью функции `netif_rx`.

4. Тестирование

По мере реализации архитектурно-зависимых модулей проводилось тестирование их работоспособности путем добавления в сборку соответствующего модуля тестов. Как видно из Рисунка 3 все разработанные модули проходят все модульные тесты.

```
        test: running embox.test.hal.context_switch_test . done
        test: running embox.test.posix.vfork_test .... done
        test: running embox.test.kernel.timer_test ..... done
        test: running embox.test.kernel.thread.thread_test .... done
        test: running embox.test.stdlib.setjmp_test . done
        unit: initializing embox.driver.net.cadence_gem: done
runlevel: init level is 2
runlevel: init level is 3
runlevel: init level is 4
Default IO device[diag]
>export PWD=/
>export HOME=/
>mkdir -v /bin
>mount -t binfs / /bin
>netmanager
>tish
root@embox:/#
```

Рис. 3: Модульное тестирование

Также для проверки работы таймера и планировщика были запущены некоторые стандартные утилиты (Рисунок 4).

```
root@embox:/#uname -a
Embox localhost 0.4.0 Apr 17 2020 22:54:16 riscv unknown  Embox-OS
root@embox:/#cat index.html
<html>
        <head>
                <title>Welcome to Embox</title>
        </head>
        <body>
                <center>
                        <h1>Welcome to Embox and have a lot of fun!</h1>
                        <h3>Embox is an open source real-time operating system designed for reso
urce constrained hardware as well as a set of tools for developing embedded applications.</h3>
                        <a href="about.html"><img src="em_big.png" alt="About Embox"></a>
                </center>
        </body>
</html>
```

Рис. 4: Запуск утилит

Все заявленные тесты используются системе непрерывной интеграции Travis, также они добавлены в отладочную сборку и исполняются при каждом запуске системы.

Заключение

В результате данной работы ОС Embox была запущена на эмулируемой QEMU плате HiFive Unleashed FU540-C000 SoC, центральный процессор которой основан на архитектуре RISC-V. В ходе данной работы были получены следующие результаты:

- Выполнен обзор операционной системы Embox и процессорной архитектуры RISC-V
- На языках программирования Си и ассемблере реализованы архитектурно-зависимые части ядра: обработчик прерываний, функции vfork и setjmp, стартовый код и модуль переключения контекстов
- На языке программирования Си реализованы драйвера основных устройств: прерываний, устройства таймера и устройства сетевого контроллера Cadence GEMGXL Gigabit Ethernet Controller
- Проведено тестирование драйвера асинхронного интерфейса UART, протестированы реализованные модули посредствам модульного тестирования ОС Embox

Исходный код для данной работы можно найти в репозитории проекта: https://github.com/embox/embox

Базовое портирование было выполнено таким образом, чтобы в будущем его можно было расширить, добавляя поддержку стандартных расширений архитектуры, драйвера других устройств.

Список литературы

[1] Lemley, Menell, Merges and Samuelson. «*Software and Internet Law*». // Wolters Kluwer, New York — 2000. — p. 34.

[2] Борисов Всеволод Васильевич, Кунявский Михаил Владимирович. «*История возникновения и вопросы распространения свободного программного обеспечения*». // Управление наукой и наукометрия — 2011.

[3] Christopher Domas «*Breaking the x86 ISA*». // Black Hat— 27 Июля 2017.

[4] OSH Association official website — Access mode: https://www.oshwa.org/research/brief-history-of-open-source-hardware-organizations-and-definitions/ (online; accessed: 22.05.2020)

[5] Which processors subject to meltdown vulnerability. — Access mode: https://meltdownattack.com/#faq-systems-meltdown (online; accessed: 29.04.2020).

[6] Moritz Lipp1, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. «*Meltdown: Reading Kernel Memory from User Space*» // Graz University of Technology, Cyberus Technology GmbH, 3G-Data Advanced Analytics, Google Project Zero, Independent (www.paulkocher.com), University of Michigan, University of Adelaide & Data, Rambus, Cryptography Research Division. — 2018.

[7] RISC-V license model. — Access mode: https://riscv.org/faq/ (online; accessed: 28.05.2020).

[8] Andrew Shell Waterman «*Design of the RISC-V Instruction Set Architecture*» // University of California, Berkeley. — 2016.

[9] Andrew Waterman, Krste Asanović «*The RISC-V Instruction Set ManualVolume I: Unprivileged ISA*» // University of California, Berkeley — 2020

[10] RISC-V cores list. — Access mode: https://github.com/riscv/riscv-cores-list (online; accessed: 12.11.2019).

[11] Peijie Li. «*Reduce Static Code Size and Improve RISC-V Compression*» // University of California at Berkeley. — 27 Июня 2019

[12] Andrew Waterman, Krste Asanović «*The RISC-V Instruction Set Manual Volume II: Privileged Architecture*» // SiFive Inc., CS Division, EECS Department, University of California, Berkeley. — 14 Ноября 2019.

[13] United States Patent № 5504901, Apr. 2, 1996. Position Independent Code Location System // R. Kim Peterson, Seattle, Wash.

[14] Position-independent code (PIC) в разделяемых библиотеках — Access mode: https://habr.com/ru/company/badoo/blog/323904/ (online; accessed: 11.05.2020).

[15] Gerald Popek, Robert Goldberg «*Formal Requirements for Virtualizable Third Generation Architectures*» // University of California, Los Angeles, Honeywell Information Systems, Harvard University. — 1974. — P. 417.

[16] Álvaro Vázquez Álvarez «*High-Performance Decimal Floating-Point Units*» // Santiago de Compostela. — January 2009. — P. 10–12.

[17] Embox key features — Access mode: https://github.com/embox/embox#key-features (online; accessed: 22.05.20).

[18] Козлов А.П. «*Портирование операционной системы с модульным HAL в пользовательский режим»* // Санкт-Петербургский государственный университет. — 2013. — P. 19–24.