Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и среды (ОСИС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему

«Эмулятор ARM»

Выполнил: студент группы 353501,

Ярош Г. И.

Руководитель: ассистент кафедры

информатики, Асташкевич М.Г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При программировании на различных мобильных платформах программист часто сталкивается с трудностями при решении поставленных. Это связано с ограниченными средствами отладки программ на большинстве мобильных устройств. С такой проблемой можно столкнуться программируя и устройства построенные на архитектуре ARM.

Все ограничения можно в один момент устранить, если предоставить возможность программисту запускать и отлаживать код программ на ПК. Однако, так как большинство компьютеров построены на архитектуре отличной от ARM, приходиться прибегать к виртуализации и эмуляции ARM-устройств.

Для ряда специфических задач приходится эмулировать работу конкретного устройства со всеми нужными модулями. Но для большинства общих проблем достаточно только базовая эмуляция архитектуры с возможностью выполнения минимального набора инструкций. Для этого создаются небольшие приложения, которые созданы для эмулирования работы процессора и выполнения.

Таким образом, целью данной курсовой работы являлось: создать приложение, эмулирующего работу реального процессора построенного на архитектуре ARM и позволяющего выполнять базовые инструкции ARM-процессоров.

1. Анализ предметной области

1.1 Архитектура ARM

Архитектура ARM (Advanced RISC Machine, Acorn RISC Machine, усовершенствованная RISC-машина) — семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер разработки компании ARM Limited.

Уже давно существует справочное руководство по архитектуре ARM, которое разграничивает все типы интерфейсов, которые поддерживает ARM, так как детали реализации каждого типа процессора могут отличаться. Архитектура развивалась с течением времени, и начиная с ARMv7 были определены 3 профиля:

* A(application) — для устройств, требующих высокой производительности (смартфоны, планшеты)
* R(real time) — для приложений, работающих в реальном времени,
* M(microcontroller) — для микроконтроллеров и недорогих встраиваемых устройств.

Профили могут поддерживать меньшее количество команд (команды определенного типа). Процессор может находиться в одном из следующих операционных режимов:

* User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.
* Fast Interrupt (FIQ) — режим быстрого прерывания (меньшее время срабатывания).
* Interrupt (IRQ) — основной режим прерывания.
* System mode — защищённый режим для использования операционной системой.
* Abort mode — режим, в который процессор переходит при возникновении ошибки доступа к памяти (доступ к данным или к инструкции на этапе prefetch конвейера).
* Supervisor mode — привилегированный пользовательский режим.
* Undefined mode — режим, в который процессор входит при попытке выполнить неизвестную ему инструкцию.

Переключение режима процессора происходит при возникновении соответствующего исключения, или же модификацией регистра статуса.

1.2 RISC процессоры

Архитектура ARM обладает следующими особенностями RISC:

* Архитектура загрузки/хранения
* Нет поддержки нелинейного (не выровненного по словам) доступа к памяти (теперь поддерживается в процессорах ARMv6, за некоторыми исключениями, и полностью в ARMv7)
* Равномерный 16х32-битный регистровый файл
* Фиксированная длина команд (32 бит) для упрощения декодирования за счет снижения плотности кода. Позднее режим Thumb повысил плотность кода.
* Одноцикловое исполнение

Чтобы компенсировать простой дизайн, в сравнении с современными процессорами вроде Intel 80286 или Motorola 68020 были использованы некоторые особенности дизайна:

* Арифметические инструкции заменяют условные коды, только когда это необходимо
* 32-битное многорегистровое циклическое сдвиговое устройство, которое может быть использовано без потерь производительности в большинстве арифметических инструкций и адресных расчетов.
* Мощные индексированные адресные режимы
* Регистр ссылок для быстрого вызова функций листьев
* Простые, но быстрые, с двумя уровнями приоритетов подсистемы прерываний с включенными банками регистров.

Одним из существенных отличий архитектуры ARM (изначальная архитектура) от других архитектур ЦПУ является так называемая *предикация* — возможность условного исполнения команд. Под «условным исполнением» здесь понимается то, что команда будет выполнена или проигнорирована в зависимости от текущего состояния флагов состояния процессора. В Thumb и Arm 64 предикация не используется — в первом режиме для неё нет места в команде (всего 16 бит), а во втором предикация бессмысленна и сложна для реализации на суперскалярных архитектурах.

В то время как для других архитектур таким свойством, как правило, обладают только команды условных переходов, в архитектуру ARM была заложена возможность условного исполнения практически любой команды. Это было достигнуто добавлением в коды их инструкций особого 4-битового поля (предиката). Одно из его значений зарезервировано на то, что инструкция должна быть выполнена безусловно, а остальные кодируют то или иное сочетание условий (флагов). С одной стороны, с учётом ограниченности общей длины инструкции, это сократило число битов, доступных для кодирования смещения в командах обращения к памяти, но с другой — позволило избавляться от инструкций ветвления при генерации кода для небольших if-блоков.

ARM7 и более ранние версии имеют трехступенчатый конвейер. Это ступени переноса, декодирования и исполнения. Более производительные архитектуры, типа ARM9, имеют более сложные конвейеры. Cortex-a8 имеет 13-ступенчатый конвейер.

1.3 Сопроцессоры ARM

Архитектура предоставляет способ расширения набора команд, используя сопроцессоры, которые могут быть адресованы, используя MCR, MRC, MRRC, MCRR и похожие команды. Пространство сопроцессора логически разбито на 16 сопроцессоров с номерами от 0 до 15, причем 15-й зарезервирован для некоторых типичных функций управления, типа управления кэш-памятью и операции блока управления памятью (на процессорах, в которых они есть).

В машинах на основе ARM периферийные устройства обычно подсоединяются к процессору путем сопоставления их физических регистров в памяти ARM или в памяти сопроцессора, или путем присоединения к шинам, которые, в свою очередь, подсоединяются к процессору. Доступ к сопроцессорам имеет большее время ожидания, поэтому некоторые периферийные устройства проектируются для доступа в обоих направлениях. В остальных случаях разработчики чипов лишь пользуются механизмом интеграции сопроцессора. Например, движок обработки изображений должен состоять из малого ядра ARM7TDMI, совмещенного с сопроцессором, который поддерживает примитивные операции по обработке элементарных кодировок HDTV.

Расширение усовершенствованного SIMD, также называемое технологией NEON — это комбинированный 64- и 128-битный набор команд SIMD (single instruction multiple data), который обеспечивает стандартизованное ускорение для медиаприложений и приложений обработки сигнала. NEON может выполнять декодирование аудиоформата mp3 на частоте процессора в 10 МГц, и может работать с речевым кодеком GSM AMR (adaptive multi-rate) на частоте более 13МГц. Он обладает внушительным набором команд, отдельными регистровыми файлами, и независимой системой исполнения на аппаратном уровне. NEON поддерживает 8-, 16-, 32-, 64-битную информацию целого типа, одинарной точности и с плавающей запятой, и работает в операциях SIMD по обработке аудио и видео (графика и игры). В NEON SIMD поддерживает до 16 операций единовременно.

Технология VFP (Vector Floating Point, вектора чисел с плавающей запятой) — расширение сопроцессора в архитектуре ARM. Она производит низкозатратные вычисления над числами с плавающей запятой одинарной/двойной точности, в полной мере соответствующие стандарту ANSI/IEEE Std 754—1985 Standard for Binary Floating-Point Arithmetic. VFP производит вычисления с плавающей запятой, подходящие для широкого спектра приложений — например, для КПК, смартфонов, сжатие звука, трёхмерной графики и цифрового звука, а также принтеров и телеприставок. Архитектура VFP также поддерживает исполнение коротких векторных команд. Но, поскольку процессор выполняет операции последовательно над каждым элементом вектора, то VFP нельзя назвать истинным SIMD-набором инструкций. Этот режим может быть полезен в графике и приложениях обработки сигнала, так как он позволяет уменьшить размер кода и выработку команд.

Другие сопроцессоры с плавающей запятой и/или SIMD, находящиеся в ARM-процессорах, включают в себя FPA, FPE, iwMMXt. Они обеспечивают ту же функциональность, что и VFP, но не совместимы с ним на уровне опкодов.

1.4 Регистры ARM процессоров

ARM предоставляет 31 регистр общего назначения разрядностью 32 бит. В зависимости от режима и состояния процессора пользователь имеет доступ только к строго определённому набору регистров. В ARM state разработчику постоянно доступны 17 регистров:

* 13 регистров общего назначения (r0..r12).
* Stack Pointer (r13) — содержит указатель стека выполняемой программы.
* Link register (r14) — содержит адрес возврата в инструкциях ветвления.
* Program Counter (r15) — биты [31:1] содержат адрес выполняемой инструкции.
* Current Program Status Register (CPSR) — содержит флаги, описывающие текущее состояние процессора. Модифицируется при выполнении многих инструкций: логических, арифметических, и др.

Во всех режимах, кроме User mode и System mode, доступен также Saved Program Status Register (SPSR). После возникновения исключения регистр CPSR сохраняется в SPSR. Тем самым фиксируется состояние процессора (режим, состояние; флаги арифметических, логических операций, разрешения прерываний) на момент непосредственно перед прерыванием

1.5 Обзор аналогов

В настоящее время существует огромное множество инструментов для эмуляции процессоров построенных на разнообразных архитектурах. Для архитектуры ARM широко используются следующие эмуляторы: ARMware, QEMU, SkyEye.

Эти приложения предоставляют возможность выполнять образ программы на не ARM-платформах. Все имеют графический интерфейс, возможность запуска в режиме отладки, дизассеблинга исходного кода.

ARMware — эмулятор ARM-архитектуры работающий на платформе x86. Может эмулировать процессоры StrongARM SA-110 и Intel Xscale.

QEMU — свободная программа с открытым исходным кодом для эмуляции аппаратного обеспечения различных платформ. Эмулятор для архитектур множества архитектур, включая x86, ARM, SPARC and PowerPC. Поддерживает работу на множестве платфрм. Имеет удобных графический интерфейс.

SkyEye — быстрый и легкий эмулятор платформ ARM, Coldfine, Mips, PowerPC, Sparc, x86. Позволяет эмулировать многоядерные и многокомпьютерные системы

Общим недостатком явсляется медленная скорость выполнения кода, сложный процесс запуска образов программ,

1.6 Постановка задачи

Приведенные аналоги эмуляторов архитектуры излишне сложны в использовании, настройке. Они обладают сложным интерфейсом, излишним функционалом для решения простых задач по отладке ARM-кода

В данной курсовой работе будет реализовано приложение для эмулирования архитектуры процессора ARM с возможностью выполнения, отладки, дизассемблинга кода.

Основными задачами программного обеспечения являются:

* Эмуляция работы ARM процессора и его основных элементов
* Эмуляция работы и взаимосвязи памяти и процессора.
* Реализация возможности выполнения программ для ARM-платформ с помощью эмулируемого процессора и памяти
* Реализация возможности отладки и дизассемблинга кода

Разрабатываемый продукт будет являться простым средством для работы с ARM кодом на других платформах, дополнит уже существубшие аналоги

2. Используемые технологии

2.1 Python

Python — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций.

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях, включая проприетарные. Есть реализации интерпретаторов для JVM (с возможностью компиляции), MSIL (с возможностью компиляции), LLVM и других. Проект PyPy предлагает реализацию Python на самом Python, что уменьшает затраты на изменения языка и постановку экспериментов над новыми возможностями.

Python — активно развивающийся язык программирования, новые версии (с добавлением/изменением языковых свойств) выходят примерно раз в два с половиной года. Вследствие этого и некоторых других причин на Python отсутствуют стандарт ANSI, ISO или другие официальные стандарты, их роль выполняет CPython.

2.1. JetBrains PyCharm

PyCharm — интегрированная среда разработки для языка программирования Python. Предоставляет средства для анализа кода, графический отладчик, инструмент для запуска юнит-тестов и поддерживает веб-разработку на Django. PyCharm разработана компанией JetBrains[3] на основе IntelliJ IDEA.

Возможности:

* Статический анализ кода, подсветка синтаксиса и ошибок.
* Навигация по проекту и исходному коду: отображение файловой структуры проекта, быстрый переход между файлами, классами, методами и использованиями методов.
* Рефакторинг: переименование, извлечение метода, введение переменной, введение константы, подъём и спуск метода и т. д.
* Инструменты для веб-разработки с использованием фреймворка Django
* Встроенный отладчик для Python
* Встроенные инструменты для юнит-тестирования
* Разработка с использованием Google App Engine
* Поддержка систем контроля версий: общий пользовательский интерфейс для Mercurial, Git, Subversion, Perforce и CVS с поддержкой списков изменений и слияния

3. Программная реализация

В данном разделе будет рассмотрена архитектура приложения, а также будут приведены примеры некоторых пользовательских функций и хранимых процедур.

3.1 Обработка инструкций

Инструкции архитектуры ARM имеют фиксированую длинну — 4 байта. Это очень положительно сказывается, на удобстве работы с образом программы, т. к. воозможно однозначно прочитать и выполнить все инструкции программы без необхоидмости сложной их обработки.

В загрузки образа программы и отдельных инструкций из файла на диске были написаны функции read\_instructions\_from\_file и read\_instructions. В первую функцию аргументом указывается имя файла с образом программы и она возаращает прочитанный образ, находящийся в памяти. Вторая разбивает образ на куски по 4 байта и кажлый декодирует как инструкцию.

Также в программу добавлены средства для компиляции исходного ассемблерного кода с помощью компилятора fasmarm. В модуле emu.utils.compile содержаться функции для компиляции отдельного файла и для компиляции ARM-кода из строки напрямую.

**3.2. Эмуляция оперативной памяти**

Классы RandomAccessMemory и RandomAccessMemoryManager реализованы для создания эмулируемой оперативной памяти, поведение которой максимально подобно на настоящую.

Класс RandomAccecssMemory и представляет собой оперативную память. Хранение данных реализовано на основе массива, в котором по адресу записывается либо читаются значения.

Класс RandomAccessMemoryManager нужен для удобного управдения памятью изнутри приложения. Он содержит в себе операции по записи, чтению из базы данных по произвольному адресу, чтение блока данных по произвольному адресу, позволяет загружать образы программ в эмулируемую память

3.3 Эмуляция процессора

Для эмуляции процессора был создан класс Procssor, который является корневым и агрегирует все остальные сущности. Методы load\_image и run\_image служат для загруски и начала выполнения образа программы соответственно.

При создании объекта процессора вместе с ним создаются объекты классов RegisterManager, Pipeline, RandomAccessMemoryManager, StateManager

Класс RegisterManager служит для удобного управления регистрами процессора. При его создании можно указать количество регистров общего доступа, которые будут доступны инструкциям. Дополнительно создаются три регистра: PC — счетчик команд(указывает на текущую выполняемую команду), SP — указатель стека, LC — содержит адресс возврата в функциях с ветвлением.

StateManager — класс, который занимается обновлением флагов состояния процессора. Поддерживаются четыре флага: C — флаг переноса, Z — флаг нуля, N — флаг отрицательноси, V — флаг переполнения. Запрещено прямое изменение флагов на любое значение. Это сследует делать через специально реализованные функции get\_flag(), set\_flag(), clear\_flag()

Класс Pipeline является программной реализацией трехступенчатого так конвеера. Каждая инструкция обрабаотывается конвеером в три этапа: получение инструкции, екодирование инструкции, выполнение инструкции. На этапе получения инструкции из оперативной памяти запрашиваются 4 байта по адресу, который храниться в регистре PC процессора. Затем команда проходит этап декодирования, где определяется тип инструкции, операнды, дополнительная информация. После инструкция выполняется и регистр PC увеличивается на 4 — становиться указателем на следующую команду.

3.4 Декодирование команд

Для декодирования команд был создан класс InstructionDecoder. Он анализирует код инструкции, прочитанной из памяти. Код инструкции состоит из секций битов, декодируя которые можно получить полную информацию о инструкции.

Декодер вначале пытается определить операнды инструкции. Первый операнд задается либо регистром, либо для команд пересылки данных может быть константой. Второй операнд может задаваться со смещением, поэтому декодированием и вычислением значения второго операнда занимается отдельный декодер ShiftDecoder. Он проверяет значение сдвига, определяет тип сдвига и сразу выполняет его.

После декодирования операндов происходит декодирование самой операции. Оно частично проходит в декодере, где определяются количество операндов, процесс выполнения. Затем код операции вместе с декодированными операндами и условием выполнения передается в класс обработчика инструкции.

3.5 Обработка команд

InstructionHandler занимается самим выполнением инструкции. В самом начале находится обработчик операции по ее коду из заранее созданных. В обработчик входят код операции, ее ассемблерное имя и функция, котороя выполняет эту операцию. Т.к. количество различных операций вместе с их кодами определено, то обработчики создаются один раз и одаются обработчику инструкции, когда нужно.

Обработчик инструкции знает операнды инструкции, регистр приемник и поэтому сам выполняет все нужные операции не затрагивая процессор(получение значений, вычисление, запись значения в приемник, обновление флагов состояния)

3.6 Отладчик

Отладчик занимается выполнением ARM-программы в консоли по шагам конвейера. На каждом шаге выводится выполненная инструкция, произошедшие изменения. Также возможно непрерывное выполнение программы

4. Описание применения

Приложение содержит ряд окон для предоставления возможности предоставления информации о системе. Рассмотрим работу с каждым из них.

4.1 Запуск отладчика в терминале Linux

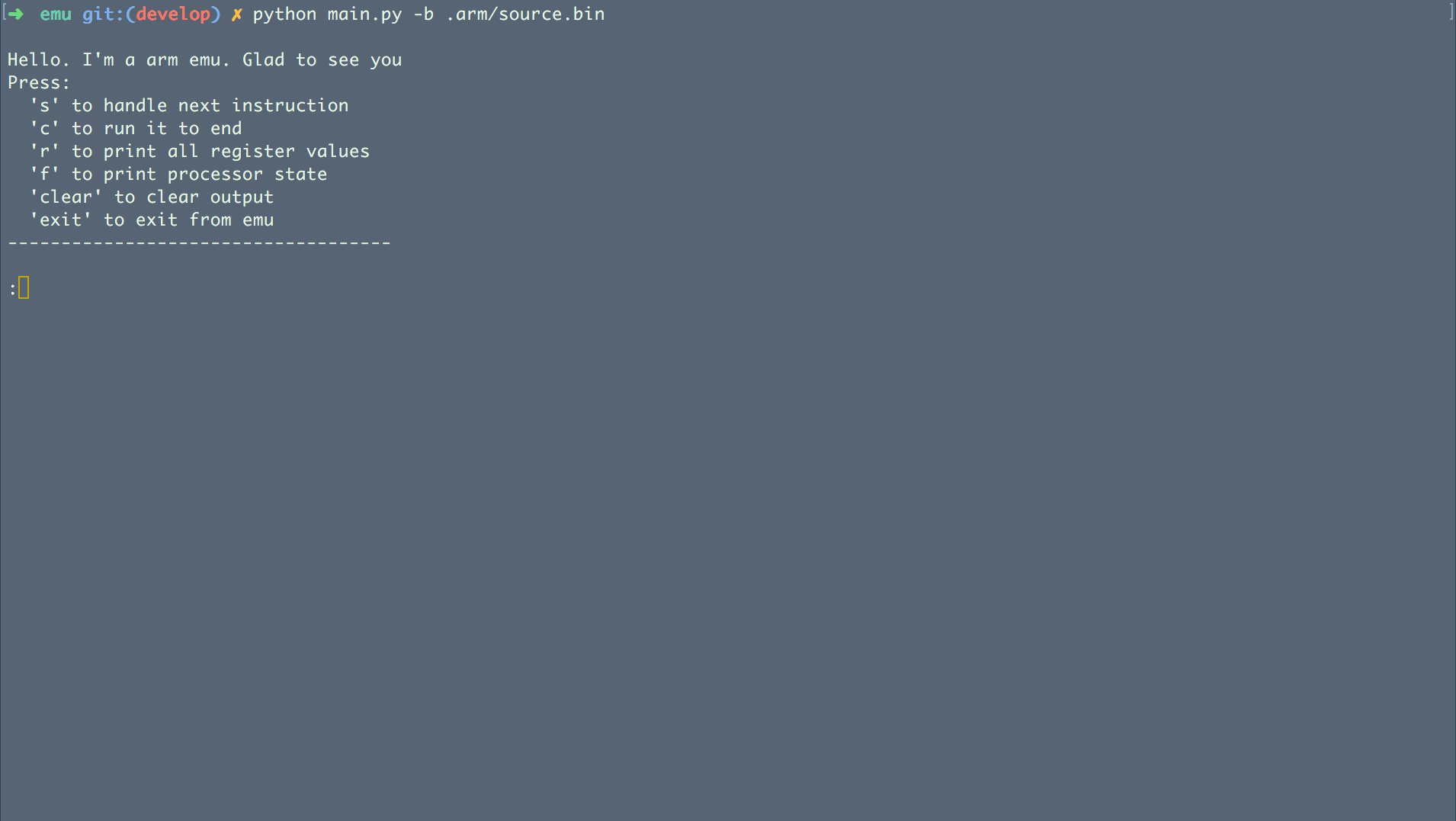
Для запуска отладчика нужно в коммандной строке вызывать скрипт main.py и в параметры передать путь к файлу с образом программы в параметре -b (--binary).

Рис. 1. Вызов отладчика из командной строки

После выполнения команды будет выведено сообщение о приветствии и неболшая подсказка, как нужно пользоваться. После этого отладчик ждет команды от пользователя. Нужно нажать s или c и выполнить перевод строки. Команда c (continue) запустит программу на непрерывное выполнение до конца образа. Команда s (step) выполнит только следующую инструкцию программы. Команда r (registers) выводит список текущих значений регистров. Команда f (flags) выводит состояние флагов регистра состояния процессора. Команда clear очищает экран терминала, а команда exit завершает работу эмулятора. Выполняя инструкции одну за другой мы можем так же выполнить программу полностью.

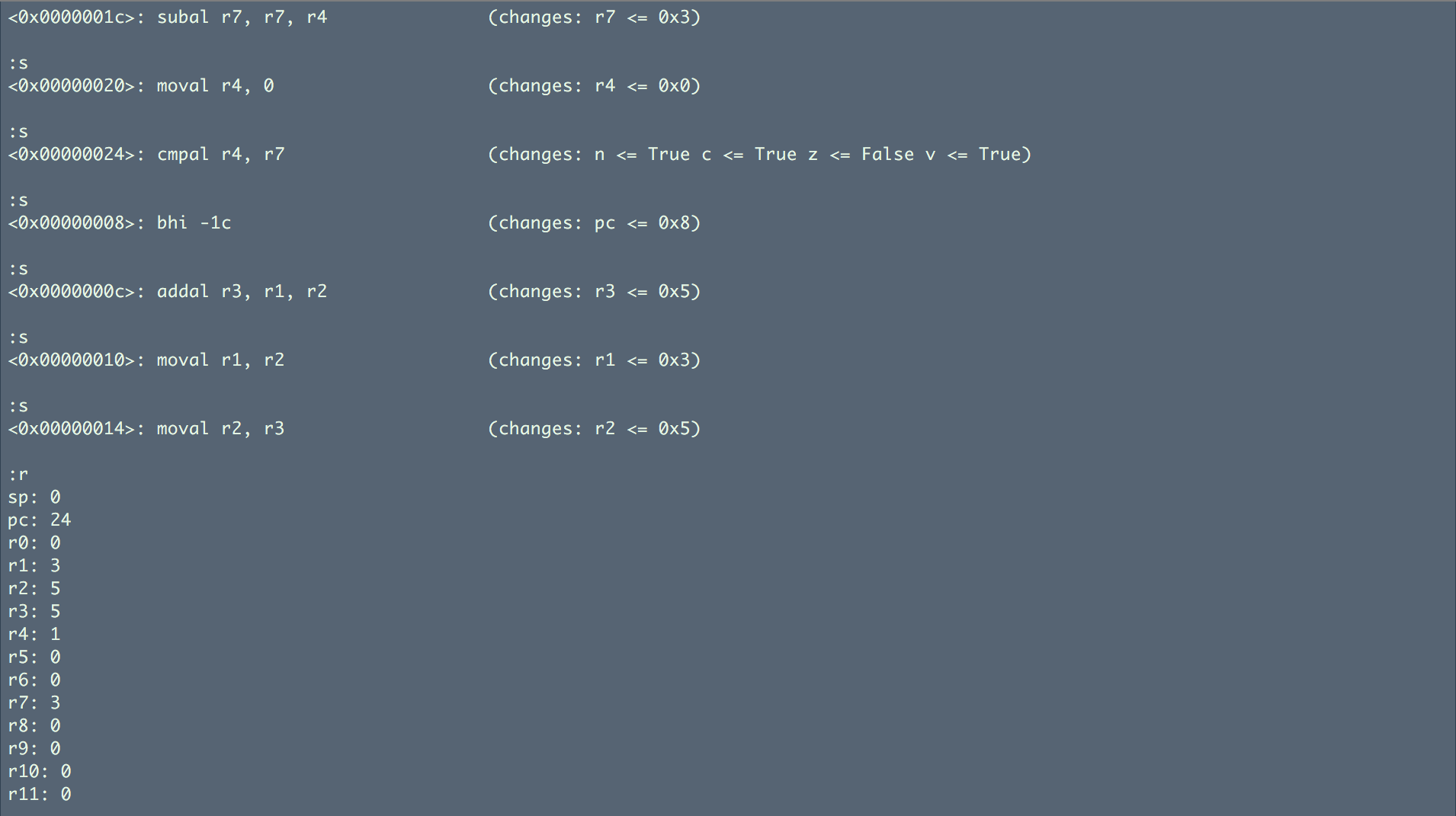
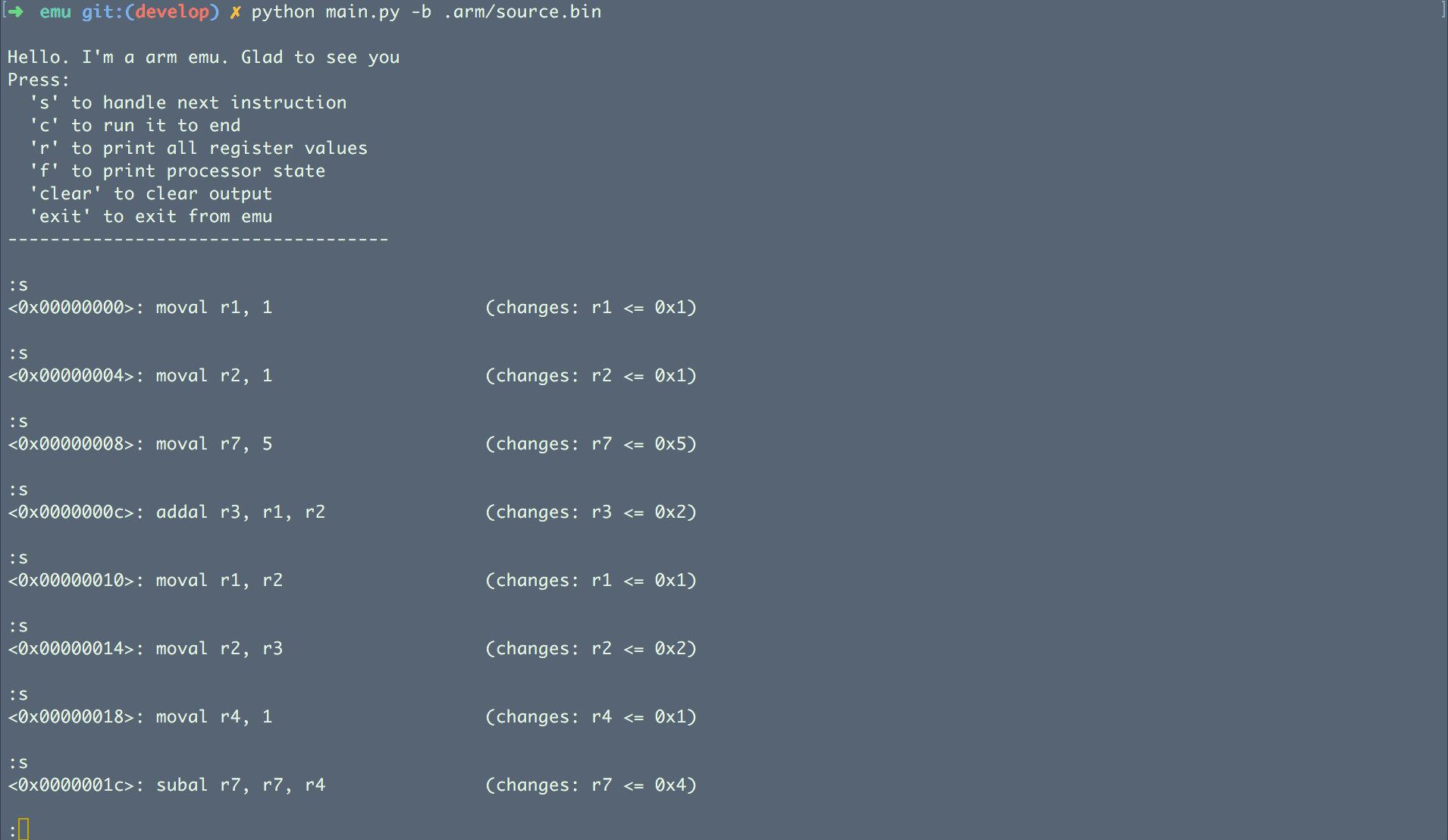
Рис. 2. Выполнение программы последовательно

Рис. 3. Вывод содержимого регистров

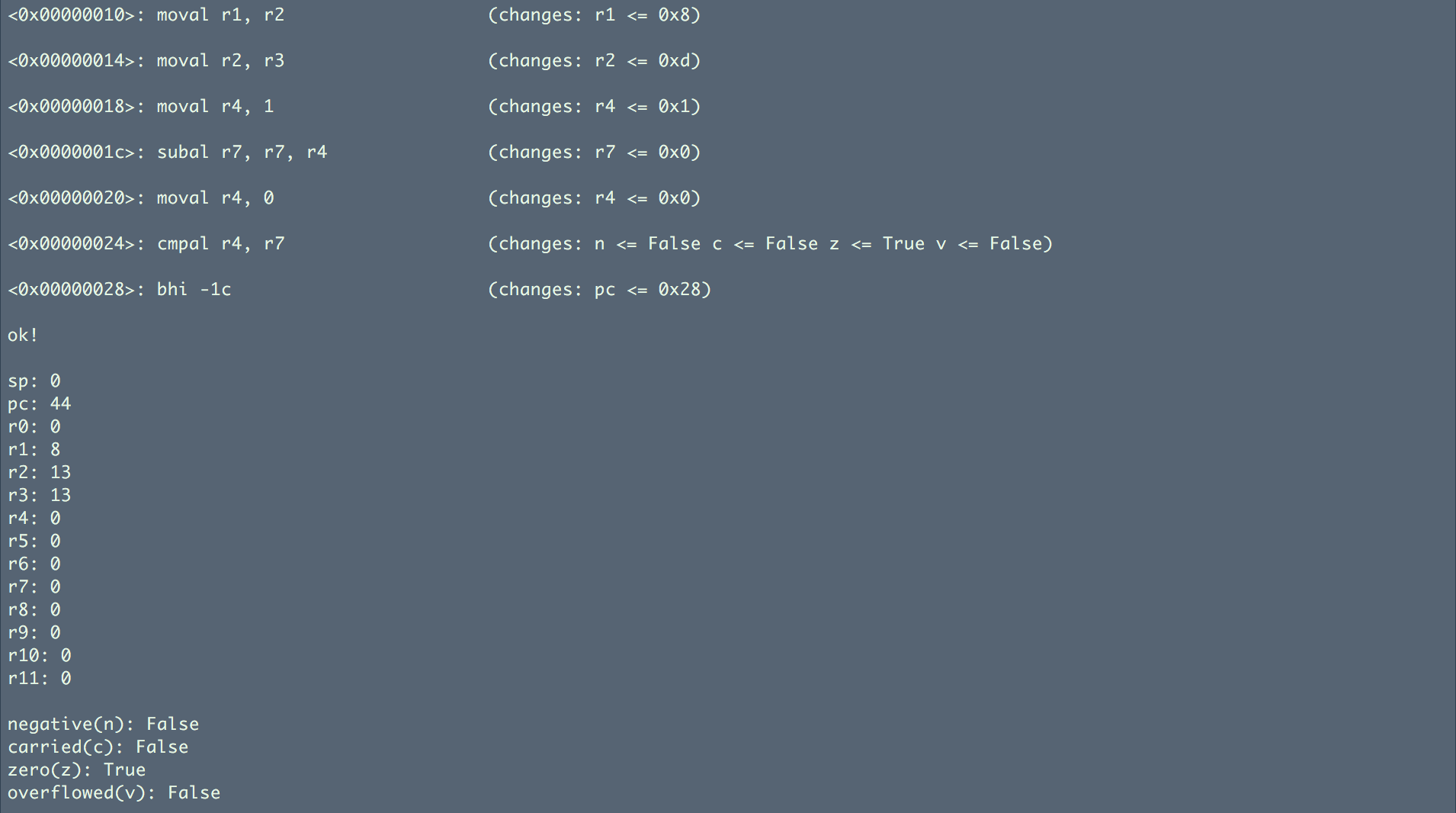


Рис. 4. Вывод флагов состояния процессора

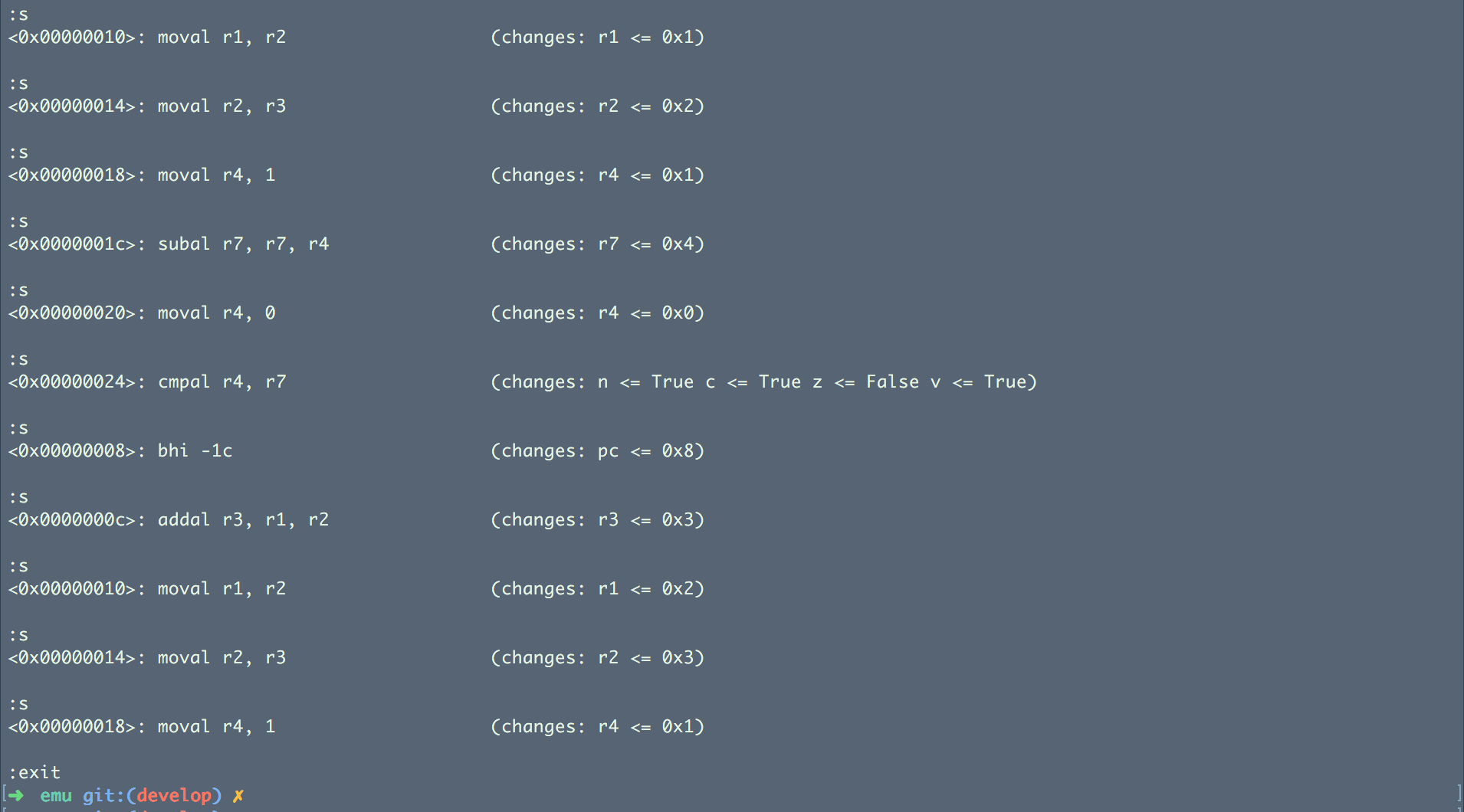
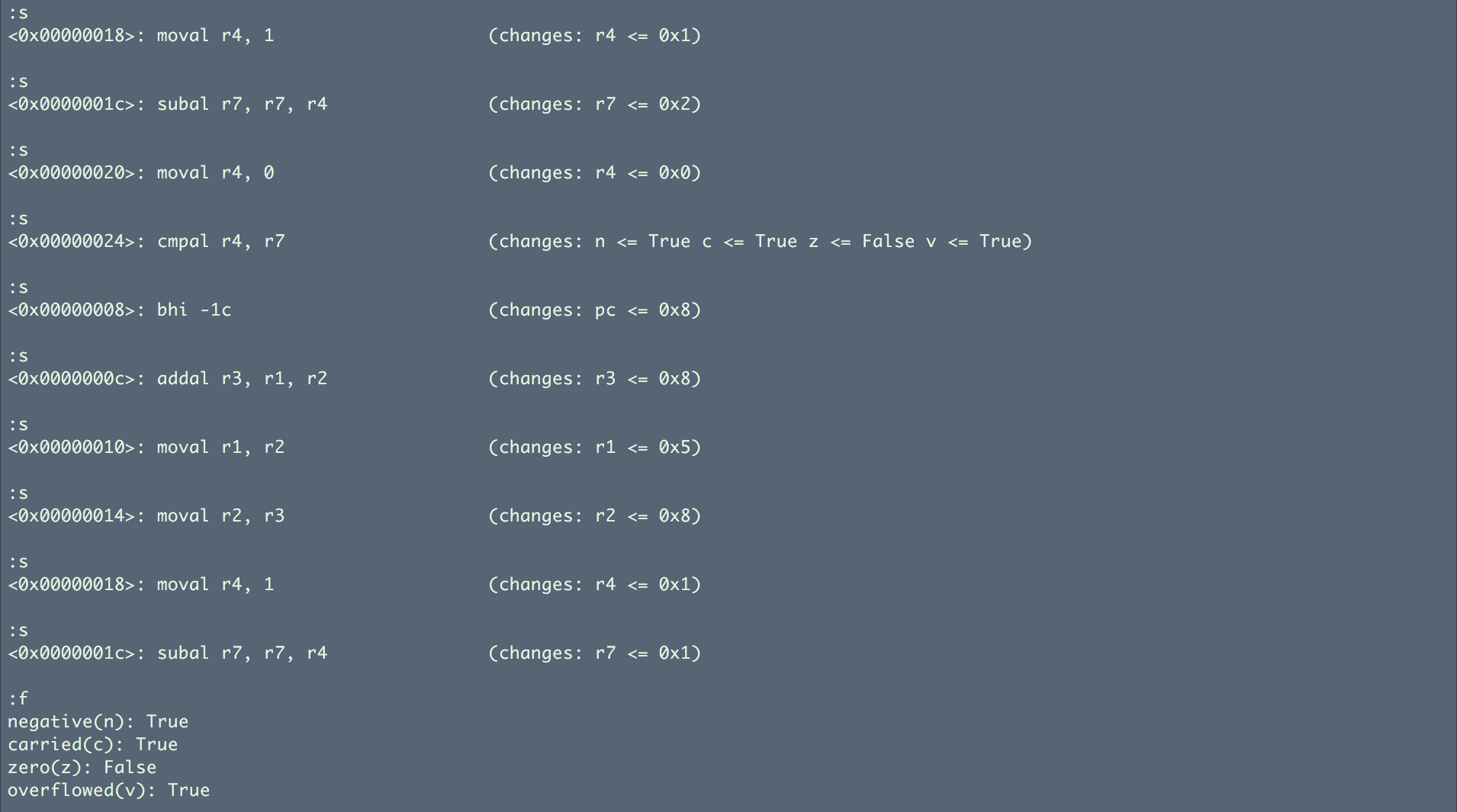


Рис.5. Завершение работы эмулятора

Рис. 6. Завершение выполнения программы

**ЗАКЛЮЧЕНИЕ**

В результате работы над курсовым проектом был разработан программный продукт для эмуляции процессора построенного на архитектуре ARM.

В процессе написания работы были закреплены навыки работы с языком Python, архитектурами компьютеров, низкоуровневом программировани на ассемблере.

Разработанное программное средство представляет собой законченный программный продукт, готовый к использованию. Но при желании приложение можно доработать: расширить функциональность программного продукта, предоставить больше возможностей для управления компонентами устройств, добавить поддержку других платформ и процессоров.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. М.Лутц. Изучаем Python, 4-ое издание / М. Лутц. - Символ-Плюс, 2010, – 1280 с.
2. ARM documentation [Электронный ресурс]. – Электронные данные. – Режим доступа

ПРИЛОЖЕНИЯ

Приложение А. Класс декодера инструкции

class InstructionDecoder(Decoder):

def \_\_init\_\_(self, processor):

self.\_processor = processor

self.str\_shifter\_operand = ''

def get\_destination\_register\_number(self, code):

dest\_reg\_number = Decoder.get\_field(code, offset=12, length=4)

return dest\_reg\_number

def get\_source\_operand\_name(self, code):

register\_number = self.get\_field(code, offset=16, length=4)

register\_name = 'r{}'.format(register\_number)

return register\_name

def get\_source\_operand\_value(self, code):

register\_name = self.get\_source\_operand\_name(code)

return self.\_processor.registers[register\_name]

def get\_shifter\_operand\_value(self, code):

is\_shift\_not\_specified = Decoder.get\_field(code, offset=25, length=1)

if is\_shift\_not\_specified:

immediate = ImmediateDecoder(self.\_processor).decode(code)

self.str\_shifter\_operand = str(immediate)

return immediate

else:

decoder = ShiftDecoder(self.\_processor)

self.str\_shifter\_operand = decoder.to\_str(code)

shift\_handler = ShiftDecoder(self.\_processor).decode(code)

self.shift\_handler = shift\_handler

shifted\_value = shift\_handler.handle()

return shifted\_value

def get\_condition\_handler(self, code):

condition\_code = Decoder.get\_field(code, offset=28, length=4)

return ConditionHandler.get\_handler(self.\_processor, condition\_code)

def to\_str(self, code):

handler = self.decode(code)

name = handler.get\_name()

if name == 'b':

return "{}{} <0x{:08x}>".format(

name, self.get\_condition\_handler(code).get\_name(),

self.\_processor.registers['pc'],

)

if name == 'mov':

return "{}{} {}, {}".format(

name, self.get\_condition\_handler(code).get\_name(),

'r{}'.format(self.get\_destination\_register\_number(code)),

self.str\_shifter\_operand

)

if name in ['tst', 'teq', 'cmp', 'cmn']:

return "{}{} {}, {}".format(

name, self.get\_condition\_handler(code).get\_name(),

self.get\_source\_operand\_name(code),

self.str\_shifter\_operand

)

else:

return "{}{} {}, {}, {}".format(

handler.get\_name(), self.get\_condition\_handler(code).get\_name(),

'r{}'.format(self.get\_destination\_register\_number(code)),

self.get\_source\_operand\_name(code),

self.str\_shifter\_operand

)

def decode(self, code):

instruction\_type = Decoder.get\_field(code, offset=26, length=2)

if instruction\_type == 0b0:

dest\_reg\_number = self.get\_destination\_register\_number(code)

source\_operand = self.get\_source\_operand\_value(code)

shifter\_operand = self.get\_shifter\_operand\_value(code)

condition\_handler = self.get\_condition\_handler(code)

instruction\_code = Decoder.get\_field(code, offset=21, length=4)

is\_state\_changing = bool(Decoder.get\_field(code, offset=20, length=1))

return InstructionHandler.get\_handler(

self.\_processor,

instruction\_code,

is\_state\_changing,

condition\_handler,

source\_operand,

shifter\_operand,

dest\_reg\_number

)

if instruction\_type == 0b10:

offset = Decoder.get\_field(code, offset=0, length=24)

condition = self.get\_condition\_handler(code)

return BranchHandler(self.\_processor, offset, condition)

Приложение Б. Класс обработчика инструкции

class InstructionHandler(object):

instructions = {

0b0000: Instruction(0b0000, 'and', lambda a, b, c: a & b),

0b0001: Instruction(0b0001, 'eor', lambda a, b, c: a ^ b),

0b0010: Instruction(0b0010, 'sub', lambda a, b, c: a - b),

0b0011: Instruction(0b0011, 'rsb', lambda a, b, c: b - a),

0b0100: Instruction(0b0100, 'add', lambda a, b, c: a + b),

0b0101: Instruction(0b0101, 'adc', lambda a, b, c: a + b + inverse(c)),

0b0110: Instruction(0b0110, 'sbc', lambda a, b, c: a - b - inverse(c)),

0b0111: Instruction(0b0111, 'rsc', lambda a, b, c: b - a - inverse(c)),

0b1000: Instruction(0b1000, 'tst', lambda a, b, c: a & b),

0b1001: Instruction(0b1001, 'teq', lambda a, b, c: a ^ b),

0b1010: Instruction(0b1010, 'cmp', lambda a, b, c: a - b),

0b1011: Instruction(0b1011, 'cmn', lambda a, b, c: a + b),

0b1100: Instruction(0b1100, 'orr', lambda a, b, c: a | b),

0b1101: Instruction(0b1101, 'mov', lambda a, b, c: b),

0b1110: Instruction(0b1110, 'bic', lambda a, b, c: a & invert(b)),

0b1111: Instruction(0b1111, 'mvn', lambda a, b, c: invert(a)),

}

@classmethod

def get\_handler(cls, processor, code, is\_state\_changing, condition\_handler,

source\_value, shifter\_value, dst\_reg\_number):

try:

handler = InstructionHandler(processor, cls.instructions[code])

handler.set\_operands(is\_state\_changing, source\_value, condition\_handler,

shifter\_value, dst\_reg\_number)

return handler

except KeyError as e:

raise ValueError('Unknown instruction code {}'.format(code))

def \_\_init\_\_(self, processor, instruction):

self.\_processor = processor

self.\_instruction = instruction

def get\_name(self):

return self.\_instruction.name

def get\_code(self):

return self.\_instruction.code

def get\_dst\_reg\_name(self):

return 'r{}'.format(self.\_dst\_reg\_number)

def set\_operands(self, is\_state\_changing, source\_value, condition\_handler,

shifter\_value, dst\_reg\_number):

self.\_source\_value = source\_value

self.\_shifter\_value = shifter\_value

self.\_dst\_reg\_number = dst\_reg\_number

self.\_is\_state\_changing = is\_state\_changing

self.\_condition\_handler = condition\_handler

def handle(self):

if self.\_condition\_handler.handle():

result\_value = self.\_instruction.callback(

self.\_source\_value, self.\_shifter\_value, 0)

register\_name = 'r{}'.format(self.\_dst\_reg\_number)

if self.\_is\_state\_changing:

self.\_processor.state.update\_flag('z', is\_zero(result\_value))

self.\_processor.state.update\_flag('n', is\_negative(result\_value))

self.\_processor.state.update\_flag('c', is\_carried(result\_value))

self.\_processor.state.update\_flag('v',

is\_overflowed(

result\_value,

self.\_source\_value,

self.\_shifter\_value

)

)

if self.get\_name() not in ['tst', 'teq', 'cmp', 'cmn']:

self.\_processor.registers[register\_name] = result\_value & (2 \*\* 32 - 1)

Приложение В. Класс конвейера

class Pipeline(object):

def \_\_init\_\_(self, processor):

self.decoder = InstructionDecoder(processor)

self.processor = processor

def retrieve\_instruction(self):

memory = self.processor.memory

pointer = self.processor.registers['pc']

instruction\_bytes = memory.read\_bytes(pointer, 4)

instruction\_code = encode\_instruction(instruction\_bytes)

return instruction\_code

def decode\_instruction(self, code):

return self.decoder.decode(code)

def handle\_instruction(self, instruction\_handler):

instruction\_handler.handle()

def itersteps(self):

try:

image\_size = self.processor.memory.get\_image\_len()

while self.processor.registers['pc'] < image\_size:

instructon\_code = self.retrieve\_instruction()

instruction\_handler = self.decode\_instruction(instructon\_code)

self.handle\_instruction(instruction\_handler)

self.processor.registers['pc'] += 4

yield {

'instruction\_code': instructon\_code,

'instruction\_handler': instruction\_handler,

}

except IndexError as e:

print e

except ValueError as e:

print e

def launch(self):

return [step for step in self.itersteps()]

Приложение Г. Классы для эмуляции оперативной памяти

class RandomAccessMemory(object):

def \_\_init\_\_(self, memory\_size, initial\_value=0):

if not 0 <= initial\_value <= 256:

raise ValueError('Default value must be in range [0..256]')

self.\_array = bytearray([initial\_value] \* memory\_size)

def \_\_len\_\_(self):

return len(self.\_array)

def \_\_iter\_\_(self):

return self.\_array.\_\_iter\_\_()

def \_\_getitem\_\_(self, address):

if isinstance(address, slice):

if len(self) <= address.stop:

raise IndexError('address is out of memory size')

return self.\_array[address.start:address.stop:address.step]

else:

if len(self) <= address:

raise IndexError('address is out of memory size')

return self.\_array[address]

def \_\_setitem\_\_(self, address, value):

if not 0 <= value <= 256:

raise ValueError('Default value must be in range [0..256]')

if (len(self) <= address):

raise IndexError('address is out of memory size')

self.\_array[address] = value

class RandomAccessMemoryManager(object):

def \_\_init\_\_(self, memory\_size, initial\_value=0):

self.\_memory = RandomAccessMemory(memory\_size, initial\_value)

self.\_image\_len = memory\_size

def \_\_iter\_\_(self):

return self.\_memory.\_\_iter\_\_()

def size(self):

return len(self.\_memory)

def read\_bytes(self, start\_address, bytes\_count):

return self.\_memory[start\_address:start\_address + bytes\_count]

def read\_byte(self, address):

return self.\_memory[address]

def write\_byte(self, address, value):

self.\_memory[address] = value

def get\_image\_len(self):

return self.\_image\_len

def load\_image(self, binary\_image):

if (len(binary\_image) > len(self.\_memory)):

raise IndexError('Size of the image must be less than memory size')

self.\_image\_len = len(binary\_image)

for address in xrange(len(binary\_image)):

self.\_memory[address] = binary\_image[address]