Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к курсовому проекту по дисциплине «Конструирование компиляторов»

Портирование компилятора BeRo TinyPascal на платформу Linux x64

Руководитель курсового проекта: _____ (А. В. Коновалов)
(подпись, дата)

Исполнитель курсового проекта,
студент группы ИУ9-72: _____ (А. В. Беляев)
(подпись, дата)

Москва, 2017

Содержание

1 АННОТАЦИЯ

Основная цель данной работы — портирование самоприменимого компилятора языка Pascal, BeRo TinyPascal под ОС Linux, используемого на кафедре ИУ9 для проведения лабораторных работ по курсу «Конструирование компиляторов». Исходный компилятор работает на платформе Windows 32-х битной разрядности. В процессе портирования необходимо увеличить разрядность до 64 бит. Таким образом будет возможно «полноценно» использовать компилятор, не имея 32-х разрядной ОС Windows.

В ходе работы будет исследовано устройство исполняемых файлов ОС Windows, исходной платформы компилятора, а также ОС Linux, целевой платформы и произведено их сравнение. Будет исследована архитектура существующего компилятора и особенности его реализации в ОС Windows. Затем будет произведена попытка обеспечить подобную архитектуру в ОС Linux.

Архитектура исходного компилятора имеет множество особенностей и значительно отличается от «классического» устройства компилятора.

Далее будет произведено непосредственно портирование компилятора, изучены последствия переноса, возможность самоприменения и будет произведено его тестирование на новой платформе.

2 ОБЗОР ФОРМАТОВ ИСПОЛНЯЕМЫХ ФАЙЛОВ

Прежде всего стоит рассмотреть устройство исполняемых файлов. Исполняемый файл(Executable file) представляет из себя программу в таком виде, в котором она может быть загружена в память и после этого исполнена. Перед исполнением могут быть также выполнены некоторые подготовительные операции, например, загрузка динамических библиотек и настройка окружения. Программа несет в себе решение определенной задачи, что влияет на ее внутреннее устройство.

Внутри файла данные хранятся в определенном формате, который разделяет их на несколько составляющих частей. Общими для большинства форматов частями являются *заголовки, инструкции* и *метаданные*.

Заголовки представляют из себя структуры, определяющие, что именно находится в определенной части файла, как данные из этой части должны быть интерпретированы, как они должны быть исполнены и так далее. Более формально, в заголовках могут быть указаны параметры окружения, исполнители инструкций, настройки этих исполнителей, а также формат инструкций и данных. Исполнителями в данном случае могут являться конкретные процессоры конкретной архитектуры (например, Intel 80386, i486 от Intel или Am386, Am486 от AMD архитектуры x86), микроконтроллеры, интерпретаторы, как программные, так и аппаратные. Также известными исполнителями являются всевозможные виртуальные машины, начиная от JVM, CPython, .NET и вплоть до VirtualBox и VMware. В рамках данной работы целевыми исполнителями будут процессоры архитектуры x86_64.

Часть исполняемого файла с инструкциями может содержать как машинные инструкции, так и исходный код, или же байт-код виртуальной машины.

Существуют и другие части файла, разнящиеся от формата к формату. Так, кроме заголовков и инструкций в файле могут присутствовать данные для загрузчика файла в память, данные для отладки, таблицы симоволов, константы, описание окружения, на котором подразумевается исполнение файла, а также всяческие изображения, тексты, архивы, иконки ярлыков и любые другие данные.

Так как описанные выше части файла и их наполнение рознятся от формата к формату, существует привязка файлов к конкретным исполнителям, например файлы только для виртуальной машины Java.

Среди наиболее популярных форматов файлов, которые будучи загруженными соответствующим загрузчиком, могут быть непосредственно выполнены CPU, а не интерпретироваться специальным ПО, можно выделить — PE (на Microsoft Windows), ELF (на Unix-подобных ОС), Mach-O (на OS X и iOS) и устаревший формат MZ (на DOS).

Исходным форматом в рамках данной работы является PE формат (его 32-х битная версия, PE32). Целевым форматом является ELF. Далее они будут рассмотрены подробнее, так как значительная часть работы связана с обеспечением исходной (в рамках PE32) архитектуры Bero TinyPascal в рамках формата ELF.

2.1 Формат PE32

После того, как для ОС Windows написан код, подключены библиотеки и загружены ресурсы, все компонуется в единственный исполняемый файл формата .exe

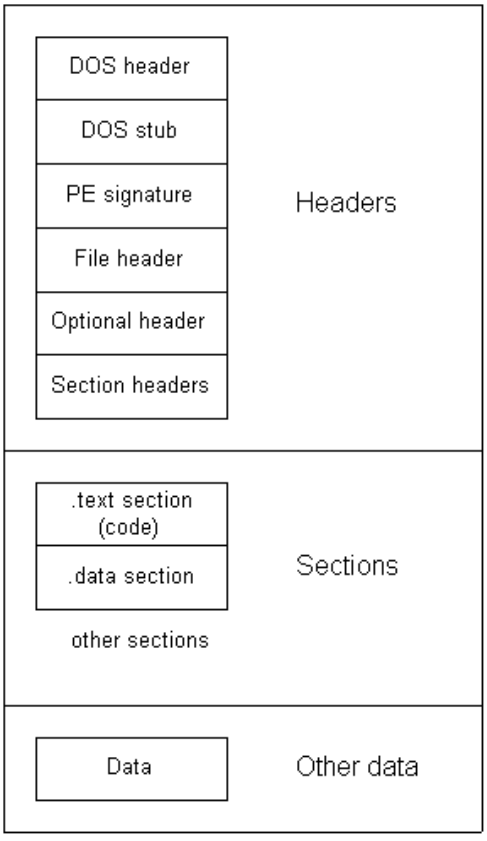
(для исполняемых файлов типов DLL и SYS также используется формат PE). Он содержит в себе всю информацию, необходимую PE-загрузчику для отображения файла в память.

PE (Portable Executable, также известный, как PE/COFF) представляет из себя модифицированную под ОС Windows версию COFF формата Unix (не только Windows, но и ReactOS использует PE). С появлением Windows NT 3.1 в 1993 году Microsoft перешла на новый формат. Однако, для обратной совместимости с DOS-системами, формат сохранил ограниченную поддержку популярного на тот момент MZ. Так, формат PE на данный момент содержит в себе «заглушку» размером в 256 байт, в которую может быть записана минимизированная версия программы размером в 192 байта и заголовок в 64 байта. Очевидно, что в настоящее время программ, «ужимаемых» до таких размеров практически не существует. Формат MZ на данный момент устарел и большинство программ содержат заглушку в виде вывода единственного сообщения – «`This program cannot be run in DOS mode`».

Однако, формат PE32 уже начинает устаревать и появляются его расширения, например PE32+ (PE+) для x64, PE.NET.

Перейдем непосредственно к техническим деталям формата. [1]. Для этого рассмотрим приблизительную схему его устройства.

Рис. 1: Схематичное устройство PE



Рассмотрение начнем сверху-вниз. Начинается файл, согласно Рисунку 1, с заголовков. Первым заголовком является упомянутый выше DOS заголовок (DOS header) длиной в 64 байта. Останавливаться на нем не имеет смысла, так как интерес в нем

представляют лишь первое и последнее поле - e_magic и e_lfanew. Первое представляет из себя 2 байта сигнатуры родительского формата – 0x4D 0x5A – «MZ». Последнее поле – по смещению 0x3C – содержит адрес PE-заголовка (PE header), который начинается в свою очередь с сигнатуры 0x50 0x45 – «PE». Если данные сигнатуры не соответствуют описанным, файл не загрузится, либо не будет исполняемым.

Далее следует сама «DOS-заглушка».

Затем начинается заголовок файла (File header, или же COFF header). Формально, file header расположен внутри PE header'a, вместе с сигнатурой и опциональным заголовком. File header представлен структурой на Листинге 1.

Листинг 1: Структура File header'a

```
typedef struct _IMAGE_FILE_HEADER {
  WORD          Machine;
  WORD          NumberOfSections;
  DWORD         TimeDateStamp;
  DWORD         PointerToSymbolTable;
  DWORD         NumberOfSymbols;
  WORD          SizeOfOptionalHeader;
  WORD          Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Здесь содержится набор полей, описывающий базовые характеристики файла, такие, как архитектура процессора (`Machine`), количество секций в файле (ограниченное числом 96), дата и время создания файла, указатель на таблицу символов и размер таблицы символов (`NumberOfSymbols`), размер опционального заголовка и характеристики файла.

Далее следует опциональный заголовок (Optional header). Он содержит стандартные COFF поля, Windows-специфичные поля и директории данных. Представленные здесь поля – RVA (Relative Virtual Address) адрес точки входа в программу, RVA начала секции кода (`BaseOfCode`), RVA начала секции данных (`BaseOfData`) и так далее.

Листинг 2: Структура элемента таблицы директорий, Data Directory

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD         VirtualAddress;
  DWORD         Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Сразу за рассмотренным выше массивом идут друг за другом заголовки секций (Section headers). Для дальнейшей работы важно остановиться на данной структуре. Заголовок секции представлен на Листинге 3.

Листинг 3: Структура заголовка секции

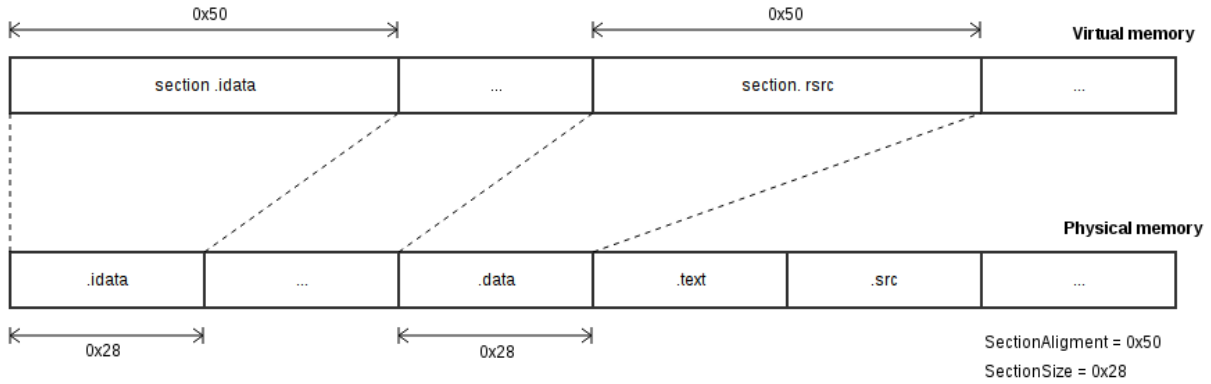
```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE          Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
     DWORD      PhysicalAddress;
```

```
    DWORD       VirtualSize;
  } Misc;
  DWORD         VirtualAddress;
  DWORD         SizeOfRawData;
  DWORD         PointerToRawData;
  DWORD         Characteristics;
  ...
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Характеристики секции представлены большим числом полей – название, размер в виртуальной памяти (`Virtual Size`), размер в файле (`SizeOfRawData`), RVA адрес секции (`VirualAddress`), RAW смещение (смещение относительно начала файла) до начала секции, кратное FileAlignment, атрибуты секции (`Characteristics`) – атрибуты содержимого (код, данные, неинициализированные данные), атрибуты доступа (чтение, запись, выполнение). Как видно из описанной выше структуры, секции в PE формате имеют довольно много характеристик.

За заголовками, как следует из Рисунка 1, расположен следующий большой раздел файла – секции. При загрузке в виртуальную память их размер увеличивается до размера, кратного выравниванию. Более наглядно это представлено на Рисунке 2. [5]. «Дополнительное» пространство, образовавшееся при этом в секции – обнуляется.

Рис. 2: Выгрузка секций в память



Исполняемый образ состоит из нескольких секций, каждая из которых требует различных прав доступа к памяти, а значит, начало секции должно быть выровнено по границе страницы. Чтобы не тратить место на диске, секции на нем не выровнены, а значит это дополнительная работа для загрузчика.

Примерами секций могут служить таблицы импорта и экспорта. Таблица импорта (IAT, Import Address Table) соотносит вызовы функций динамических библиотек по имени или ординалу (порядковому номеру) с соответствующими адресами. Таблица экспорта (EAT – Export Address Table) в свою очередь предоставляет адреса функций другим модулям, обращающимся к EAT за импортом. Это характерно для DLL библиотек. Еще один пример – таблица релокаций, исполь-

зуемая для вычисления адреса загрузки в случае, если предпочтительный адрес (`IMAGE_OPTIONAL_HEADER.ImageBase`) оказался недоступен. Каждая секция представлена своей специфической структурой, которая в свою очередь представляет из себя набор структур (например, «путь» до имен функций в таблице импорта или экспорта).

Представленной выше информации достаточно для общего понимания внутреннего устройства файлов PE формата.

Процесс загрузки можно вкратце описать так:

- Считывание и валидация сигнатур и заголовков.
- Выделение виртуальной памяти под приложение и попытка выгрузить его по предпочтительному адресу.
- Вычисление адреса и размера секций в виртуальной памяти, в том числе вычисление смещений и установка атрибутов страниц согласно атрибутам секций, обнуление секций до нужной длины.
- Анализ таблицы импорта и загрузка соответствующих DLL, связывание импортируемых симоволов.

На данный момент уже можно представить, что происходит с исполняемым файлом в исходной системе. Получив базовые сведения об устройстве PE, необходимо изучить и ELF, так как он является целевым форматом.

2.2 Формат ELF

ELF (Executable and Linkable Format) – формат исполняемых файлов во многих Unix-подобных системах, разработанный USL (Unix System Laboratories) в качестве переносимого для ОС на базе Intel x86. Затем, будучи доработанным компанией HP до стандарта ELF-64, распространился и для 64-разрядных систем.

На первый взгляд схема ELF'а напоминает немного перекомпанованный PE файл. Рассмотрим ее устройство на Рисунке 3, и проверим, насколько верно это утверждение.

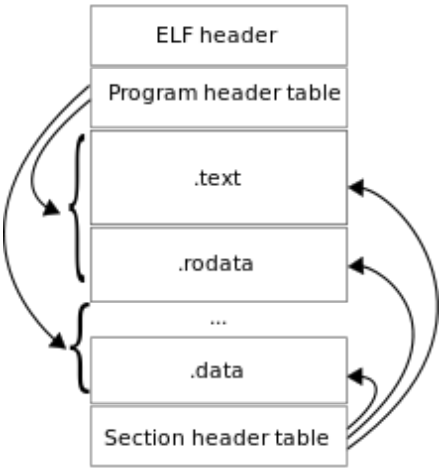
ELF-файл состоит из заголовка файла, таблицы программных заголовков, секций и таблицы заголовков секций в конце файла.

Заголовок файла (ELF header) расположен в самом начале файла, как это следует из Рисунка 3 и содержит общее описание структуры файла и его основные характеристики, такие, как тип, версия, архитектура, точка входа. Опустим детальное рассмотрение этой структуры.

Таблица заголовков программы (Program header table), также известная как таблица заголовков сегментов (Segment table) идет следом за ELF header'ом. Каждый элемент этой таблицы содержит в себе тип сегмента (p_type), расположение сегмента, точку входа, размер(p_filesz, p_memsz) и флаги доступа(p_flags), согласно Листингу 4.

Сегменты необходимы для загрузки файла в память и исполнения. Только та часть данных, которую они охватывают нужна для работы с файлом. Остальное, с позиции загрузчика – отладочная информация, или же – метаданные.

Рис. 3: Устройство ELF



Листинг 4: Программный заголовок (заголовок сегмента)

```
typedef struct _ELF64_PHDR{
    Elf64_Word          p_type;
    Elf64_Word          p_flags;
    Elf64_Xword         p_filesz;
    Elf64_Xword         p_memsz;
    ...
} Elf64_Phdr;
```

Далее следуют непосредственно секции. Секции необходимы при линковке (компоновке) файла. Не стоит путать секции с сегментами ELF'а. Это разные вещи. Как упомяналось ранее, сегменты являются выгружаемыми в память частями файла и эти части представлены в том числе и секциями. Так, одному сегменту могут соответствовать несколько секций, при этом некоторые могут не отображаться в сегменты (память) вообще. Соответствие одних другим описывается при компоновке файла и будет подробнее рассмотрено позднее в данной работе. В том числе будут продемонстрированы и секции, не отображаемые в память вовсе.

За секциями вновь идет таблица – таблица заголовков секций (Section header table). Изучение ее стркутуры может быть опущено.

На этом описание заголовков можно закончить. Процесс выгрузки ELF файла в память во многом схож с таковым для PE файлов. Следующим пунктом этого раздела будет краткое сравнение форматов.

2.3 Сравнение форматов

На основании сказанного выше, можно отметить следующее:

- PE формат представлен гораздо бОльшим числом структур и всевозможных подструктур.
- Поля PE формата предоставляют загрузчику исчерпывающую информацию для выгрузки в память, делая код позиционно-зависимым.

- Размещение структур в PE файле задано более жестко.
- В PE имеется лишняя неиспользуемая «заглушка времен DOS-овского прошлого» системы.
- ELF использует полностью позиционно-независимый код и глобальную таблицу смещений, которая жертвует временем выполнения в пользу расходования памяти.
- В случае отсутствия необходимости в перебазировании адресов, PE-файлы имеют преимущество очень эффективного кода, но при наличии перебазирования издержки в использовании памяти могут быть значительными.

Исходя из всего этого, можно предположить, что работа с ELF файлом проще, чем с PE. Однако, опровергнуть или доказать это утверждение удастся только при непосредственной работе по устройству архитектуры. На данный момент имеется достаточно информации об устройстве исходного и целевого форматов. Следует приступить к изучению самого компилятора.

3 ОБЗОР КОМПИЛЯТОРА BeRo TinyPascal

Рассматриваемым в данной работе компилятором является компилятор языка Pascal, *Bero TinyPascal*. Он был разработан в 2006 году, тогда немецким программистом Бенджамином «BeRo» Россо (Benjamin Rosseaux) и на данный момент распространяется под лицензией zlib (совместимой с GNU GPU) из его репозитория на GitHub (github.com/BeRo1985/berotinypascal).

Этот компилятор используется на кафедре ИУ-9 МГТУ им. Баумана для проведения лабораторных работ по изучению раскрутки (bootstrapping) и самоприменения в рамках курса «Конструирование компиляторов».

Компилятор является самоприменимым и преобразует исходный код из подмножества языка Pascal в двоичный код платформы Windows x86.

Условно его можно разделить на 2 части – «верхнюю», btpc.dpr, сам *компилятор BTPC*, реализующий фазы анализа и синтеза, и «нижнюю», rtl.asm, *библиотеку RTL* (RunTime Library), содержащую низкоуровневые реализации функций BTPC.

3.1 Библиотека времени выполнения, RTL

Библиотека времени выполнения (runtime) RTL отвечает за низкоуровневое устройство BTPC, а именно – распределение памяти и запись выходного ассемблерного кода в выходной поток (файл). Непосредственно генерация кода на целевом языке происходит на «верхнем» уровне, в самом BTPC.

Библиотека предоставляет 9 основных функций:

1. RTLHalt — остановка программы.
2. RTLWriteChar — запись char'a на stdout.
3. RTLWriteInteger — запись целого на stdout, принимает два параметра: число и ширину вывода.

4. RTLWriteLn — выводит на stdout символ новой строки.

5. RTLReadChar — считывает символ из stdin, результат кладёт в `EAX`.

6. RTLReadInteger — считывает целое из stdin, результат кладёт в `EAX`.

7. RTLReadLn — пропускает стандартный ввод до конца файла или ближайшего перевода строки.

8. RTLEOF — возвращает в `EAX` число больше 1, если достигнут конец файла (следующий символ прочитать невозможно) или 0 в противном случае.

9. RTLEOLN — устанавливает в `DL` 1, если следующий символ \n, 0 — в противном случае.

Также имеется импорт утилитарных функций из библиотеки kernel32.dll. Среди них HeapAlloc для работы с кучей, выделяющая 4 Мбайта памяти при инициализации, устанавливающая на них указатель стека и HeapFree, в конце работы программы освобождающая эту память. Импортируются также функции обработки ввода данных – GetStdHandle, SetConsole, ReadConsoleInputA.

Все взаимодействие с библиотекой осуществляется посредством таблицы встроенных функций (RTLFunctionTable). В таблице, в описанном выше порядке находятся указатели на соответствующие функции, при этом таблица в течение работы всегда доступна в регистре `ESI`. Особенностью работы рантайма является неизменяемость этого регистра.

Примечательной особенностью, сбивающей с толку является взаимодействие RTL и BTPC. В этих файлах нет ни одной ссылки друг на друга, а в RTL точка входа в программу (`ProgramEntryPoint`)является последней строкой файла и указывает «в никуда». То есть стандартным поведением RTL, как самостоятельного процесса будет ошибка сегментирования (Segmentation fault), см Листинг 5 (оригинальная нотация соблюдена).

Листинг 5: Метки StubEntryPoint и ProgramEntryPoint, получающие управление

```
StubEntryPoint:
  INVOKE GetStdHandle,BYTE -10
  MOV DWORD PTR StdHandleInput,EAX
  INVOKE SetConsoleMode,EAX,1+2
  INVOKE HeapAlloc,EAX,HEAP_GENERATE_EXCEPTIONS+HEAP_ZERO_MEMORY
                      +HEAP_CREATE_ALIGN_16,4194332
  ...
 .LIBRARY "kernel32.dll"
 IMPORT WriteFile "WriteFile"
 IMPORT HeapFree "HeapFree"
ProgramEntryPoint:
```

Это первая встретившаяся «особенность» реализации. Идея BeRo будет понятна при рассмотрении самого BTPC.

3.2 Компилятор BTPC

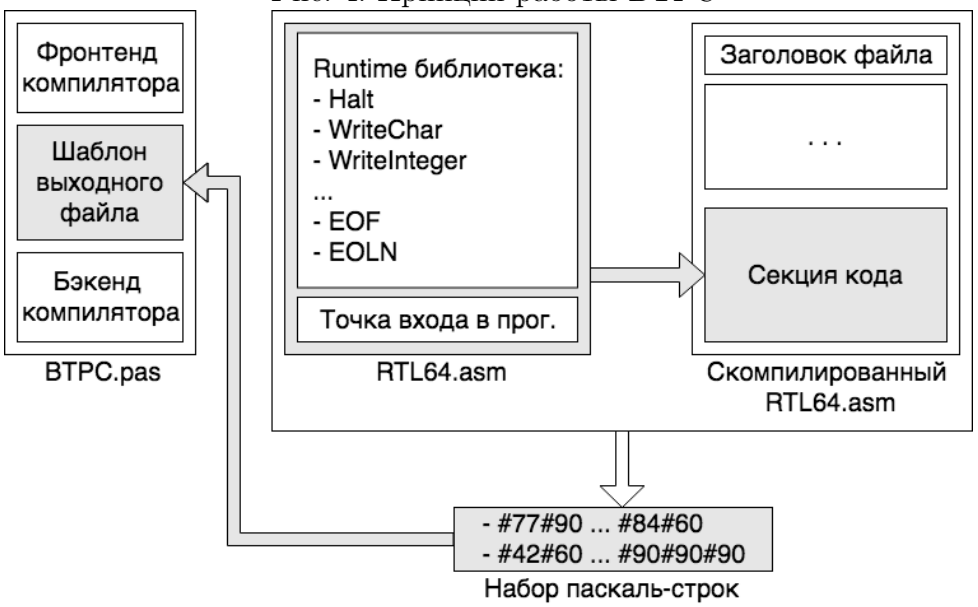
Компилятор BTPC (BeRo TinyPascal Comiler), представленный исходным кодом btpc.dpr и является «верхней» и главной частью системы. В нем происходит как *фаза анализа* – чтение входного потока, лексический анализ, синтаксический анализ, генерация промежуточного представления, так и *фаза синтеза* – генерация кода на целевом языке. Оптимизация, как стадия компиляции, отсутствует.

3.2.1 Общий принцип работы

Устройство компилятора нетривиально и существенно отличается от «привычного» представления. «Исходными файлами компилятора» являются рассмотренная ранее библиотека времени выполнения RTL, написанная на языке ассемблера и единственный файл компилятора на языке Pascal, устроенный по принципу «все в одном» (фронтенд, бэкенд) – BTPC.pas.

Далее следует рассмотреть Рис. 4., чтобы понять принцип работы. Файл библиотеки RTL.asm, содержащий таблицу функций и имеющий точку входа, указывающую в конец файла компилируется компилятором Excagena, производя на выходе минималистичный выходной файл, в секции которого расположен код библиотеки RTL.asm и секция кода при этом расположена в «конце» файла. Если этот файл будет запущен, то он вызовет ошибку Segmentation Fault.

Рис. 4: Принцип работы BTPC



Затем этот файл побайтно разбирается на паскаль-строки. Эти паскаль строки вставляются в исходный код BTPC.pas. Назовем их шаблонами выходного файла. Затем файл BTPC.pas, компилируется в компилятор BTPC.

При компиляции любой программы с помощью BTPC происходит следующее: фронтенд производит необходимые стадии анализа и генерирует промежуточное представление – байт-код. Этот байт-код преобразуется следующим образом: каждой инструкции (или нескольким инструкциям) байт-кода ставится в соответствие набор

ассемблерных инструкций. Эти инструкции, как и ранее, преобразуются в строковое представление и дописываются в конец шаблона (полученное ранее текстовое представление скомпилированного файла библиотеки RTL.asm).

Таким образом выходной файл, генерируемый компилятором представляет из себя конкатенацию кода библиотеки и кода компилируемой программы.

Затем происходит патчевание получившегося файла (все еще представленного в виде строк), которое можно сравнить с хирургической операцией: по заранее определенным местам, в которых лежат значения, например, сдвигов секций, размеров секций и прочих необходимых для функционирования значений записываются вычисленные самостоятельно значения с учетом кода, дописанного в конец файла. Очевидно, что неверное значение или неверный адрес записи с большой вероятностью вызовет повреждение программы.

Пример патчевания: запись о размере секции кода начинается 156 байтом в файле и занимает 4 байта. При этом во время работы компилятора было сгенерировано X байтов кода. Тогда необходимо извлечь 4 байта, начиная со 156го, прибавить к извлеченному числу значение X и записать обратно. И подобным образом необходимо поступить со всеми значениями, которые могли потерять актуальность после дописывания кода в конец файла.

Такой подход к архитектуре весьма необычен и нетривиален из-за тонкостей, связанных с генерацией двух частей программы (очевидно, что если дописать код в конец случайной программы, то это либо не приведет ни к чему, либо вызовет поломку) и представляет отдельный интерес при работе.

3.2.2 Детальное описание устройства

Код, как в промежуточном представлении, так и целевой, генерируется по входному потоку «as is», как есть, то есть без каких-либо оптимизаций и изменений. Входным потоком на первом этапе служит поданный на stdin код. Входным потоком на втором этапе служит сгенерированный на первом этапе промежуточный код. Он представляет из себя обычный массив (`Code`), каждый i-й элемент которого является идентификатором операции (которых 45), а $(i+1)$-й — значением, связанными с операцией (`Value`). С некоторой долей условности это может быть названо байт-кодом.

После порождения промежуточного кода запускается процедура `AssembleAndLink`, в которой с обходом массива промежуточного кода генерируется двоичный код в виде отдельных кодов инструкций в hex формате. Это также происходит в 2 этапа:

На первом этапе библиотека RTL из пункта выше вручную компилируется в исполняемый файл rtl.exe. Для этого используется ассемблер Excagena Assembler за авторством самого BeRo, со специальной конфигурацией, настроенной на получение минималистичного выходного файла. Этот файл преобразуется специальной программой, написанной BeRo также на Pascal (rtl2pas.dpr) в pascal-строки вида `#77#90 ... #$90#$90`, где после символов `#` идут десятичные представления байтов программы, поданной на вход, а дополнение в конец строки до длины в 255 элементов происходит путем записи недостающего количества байтов `$90`, являющихся кодами (opcode) ассемблерной инструкции `NOP`. В примере выше, элементы `#77#90` при переводе обратно в hex представление становятся байтами `4D 5A`, или строкой «MZ», и являются сигнатурой PE файла, описанной ранее в данной работе. Очевидно, что,

например, файл размером в 1000 байт будет записан 4 подобными строками, последняя из которых дополнена до длины 255 инструкциями `NOP` (No operation). Размер файла в строковом представлении при этом будет искусственно увеличен на 20 байт (20 инструкций `NOP`). Полученные таким образом строки записываются в «итоговую» последовательность кода (массив `OutputCodeData`).

На данном этапе внутри BTPC имеется полноценный работоспособный (инициализирующий библиотеку и намеренно затем «сваливающийся» в Seg fault) файл, записанный в массив.

Листинг 6: Генерация инструкции байтами кода операции

```
procedure OCMovEAXDWordPtrESP;
begin
  EmitByte($8b); EmitByte($04); EmitByte($24);
  LastOutputCodeValue:=locMovEAXDWordPtrESP;
end;
```

Далее процедурами `EmitChar(char)`, `EmitByte(integer)`, `EmitInt16(integer)`, `EmitInt32(integer)` и более частными их случаями (например, процедура на Листинге 6, отвечающая за инструкцию `MOV EAX,DWORD PTR [ESP]`) в выходной массив `OutputCodeData`, поверх инструкций `NOP` записываются порожденные из промежуточного кода ассемблерные инструкции, также в виде кодов операций, но уже с префиксом `$` – в hex формате.

Однако, не все инструкции получили свои процедуры и бОльшая часть кода генерируется прямо в `AssembleAndLink` блоками `switch`-оператора, подобными представленному на Листинге 7 блоку, эквивалентному инструкциям на Листинге 8.

Описанным выше образом на месте «падения» из метки `ProgramEntryPoint` появляется новый, «скомпилированный» код, являющийся программой, поданной на вход компилятору.

Листинг 7: Генерация инструкций (opcode) из промежуточной инструкции OPStl

```
OPStL:begin
  OCPopEAX;
  Value:=Value-4;
  if Value=0 then begin
   EmitByte($89); EmitByte($04); EmitByte($24);
  end else if (Value>=-128) and (Value<=127) then begin
   EmitByte($89); EmitByte($44); EmitByte($24); EmitByte(Value);
  end else begin
   EmitByte($89); EmitByte($84); EmitByte($24); EmitInt32(Value);
  end;
  LastOutputCodeValue:=locNone;
  PC:=PC+1;
 end;
```

Листинг 8: Ассемблерные инструкции блока OPStl

```
POP EAX
```

```
MOV DWORD PTR [ESP],EAX
MOV DWORD PTR [ESP+BYTE Value],EAX
MOV EAX,DWORD PTR [ESP+DWORD Value]
```

Pascal, который обрабатывает BTPC, является подмножеством обычного языка Pascal, в нем запрещены дженерики, шаблоны, перегрузка операторов и прочие «новые» возможности, которых не было в Delphi 7 и в самом BTPC на момент создания (2006 год), но при этом присутствуют и «незадекларированные» возможности, связанные с недоработкой логики компилятора (об этом позднее).

В самом конце процедуры `AssembleAndLink` находится еще одна архитектурная особенность BTPC – редактирование (патчевание, жарг.) выходного файла. Фрагмент этой части представлен на Листинге 9.

Листинг 9: Редактирование выходного файла

```
{ Size Of Code }
 PEEXECodeSize:=OutputCodeGetInt32($29)+(OutputCodeDataSize-CodeStart);
 OutputCodePutInt32($29,PEEXECodeSize);
{ Get section alignment }
 PEEXESectionAlignment:=OutputCodeGetInt32($45);
 ...
{ Calculate and patch image size }
 OutputCodePutInt32($5d,PEEXESectionVirtualSize+OutputCodeGetInt32($39));
{ Patch section raw size }
 OutputCodePutInt32($115,OutputCodeGetInt32($115)+(OutputCodeDataSize-CodeStart));
```

На Литинге 10 представлен дамп памяти, соответствующий началу файла BTPC в PE32 формате. Как и упомяналось ранее, выходной файл скомипилированной RTL библиотеки очень мнималистичен. В частности на месте DOS-«заглушки» находится авторская сигнатура, хотя сигнатуры, необходимые PE-загрузчику тоже присутствуют («MZ», «PE»).

Листинг 10: Дамп памяти BTPC

```
0000:  4d 5a 52 c3-42 65 52 6f-5e 66 72 00-50 45 00 00  MZR.BeRo ^fr.PE..
0010:  4c 01 01 00-00 00 00 00-00 00 00 00-00 00 00 00  L....... ........
0020:  e0 00 0f 03-0b 01 00 00-3f cf 00 00-00 00 00 00  ........ ?.......
0030:  00 00 00 00-c4 10 00 00-00 10 00 00-0c 00 00 00  ........ ........
0040:  00 00 40 00-00 10 00 00-00 02 00 00-04 00 00 00  ..@..... ........
```

Исследованного ранее устройства формата PE32 и комментариев BeRo достаточно для понимания описанных в Листинге 9 действий. Происходит следующее:

- Корректировка ближних переходов (`JMP xx, CALL xx`) — это связано с особенностью кодогенерации меток и переходов по ним.
- Корректировка размера кода (рассмотренное ранее поле SizeOfCode Optional header'a).

- Получение имеющегося выравнивания и корректировка с помощью него виртуальных размеров секций (SectionHeader.VirtualSize). Это необходимо при загрузке файла на исполнение, см Рисунок 2.
- Корректировка `ImageSize`'а, размера образа.
- Обновление RAW размера единственной секции (размер в файле).

Почти ко всем полям добавляется размер инъектированного кода входной программы.

Компилятор работает, а значит, редактирование происходит успешно и нескольких подкорректированных полей достаточно, чтобы PE загрузчик обработал файл корректно.

Однако, становится заметна еще одна «неприятная особенность» кода — наличие «магических» констант, к большинству которых отсутствуют поясняющие комментарий. В том числе и по этой причине в начале работы изучалось устройство PE.

Оптимальность порожденного кода также оставляет желать лучшего, хотя данная проблема и находится вне контекста работы.

На этом изучения входных данных и теоретической подготовки достаточно для перехода к портированию.

4 ПОРТИРОВАНИЕ

Первым делом необходимо разобраться с внутренним устройством библиотеки RTL и портировать ее. Затем изменить кодогенерацию на порождение инструкций набора x64. Далее – отредактировать выходной ELF аналогичным описанному ранее способом и убедиться в работоспособности полученного ELF файла.

4.1 Библиотека RTL

Верхнеуровневое описание библиотеки приводилось ранее. Теперь необходимо портировать ее на платформу amd64. Для этого нужно заменить библиотечные вызовы к WinAPI на аналогичные в Linux. Исходные системные вызовы BeRo реализовал с помощью макросов своей среды разработки.

На платформе же amd64 системные вызовы осуществляются подготовкой аргументов вызываемой функции с помощью регистров и затем вызовом инструкции `syscall` (`int 0x80` для x32). Более детально распределение регистров на архитектурах i386 и x86_64 представлено в Таблице 1.

Так как BeRo работал с макросами, использующими стек для обращения к WinAPI, в то время, как системные вызовы `syscall` используют регистры, необходимо постараться обеспечить аналогичную «чистоту» функций RTL, чтобы конечный результат функционировал так же, как и оригинал.

Портируем простую функцию – RTLWriteChar. Ее назначение понятно из названия. На Листинге 11 представлена оригинальная функция, а на Листинге 12 – ее портированная версия.

Вызовом `syscall` обратимся к функции `Write()`. Ее подробную спецификацию

Таблица 1: Интерфейс 32 и 64-битных системных вызовов

архитектура	инструкция	syscall #	ret value	arg1	arg2	arg3	arg4	arg5	arg6
i386	int $0x80	eax	eax	ebx	ecx	edx	esi	edi	ebp
x86_64	syscall	rax	rax	rdi	rsi	rdx	r10	r8	r9

можно получить командой «`man 2 write`» в терминале. Распределим ее аргументы по соответствующим регистрам.

Листинг 11: Оригинал RTLWriteChar

```
RTLWriteCharBytesWritten: DB 0x90, 0x8D, 0x40, 0x00
RTLWriteChar:
  push esi
  lea esi, [esp+8]
  pushad
  invoke WriteFile, dword ptr StdHandleOutput,  esi, byte 1,
                    offset RTLWriteCharBytesWritten, byte 0
  popad
  pop esi
  ret 4
```

С портированием даже такой простой функции выявляется еще одна «особенность» — данные внутри кода. На Листинге 11 количество байт, отправленное на stdout, хранится по метке `RTLWriteCharBytesWritten`, которая расположена не в секции данных, а в секции кода, в связи с чем секция кода является единственной (!) секцией в файле и отмечена как *исполняемая, перезаписываемая*, что строго говоря является нарушением многих норм.

Это может привести к непредвиденным последствиям в работе, начиная от Seg fault'а и заканчивая внедрением стороннего кода. Если не учесть всех деталей (что спустя 10 лет может быть довольно затруднительно даже для автора), можно столкнуться с большими трудностями, особенно с учетом не просто портирования, но и увеличнения разрядности инструкций.

Обезопасить себя можно, отрефакторив код параллельно с портированием. Для инициализированных данных в новой версии введена секция `.data`, для неинициализированных – секция `.bss`.

В связи с введением дополнительных регистров в x64 и меньшей надобности сохранять регистры в стеке при вызовах, из набора существующих инструкций исчезли аналоги инструкций `pusha, pushad` и `popa, popad` для 64-битных регистров. Но, поскольку так или иначе, при `syscall`'ах регистры меняются, стоит самостоятельно ввести аналогичные инструкции в виде макросов – `pushall` и `popall`, запушивающих регистры от `rdi` до `rax` и возвращающих их из стека в обратном порядке. Эти и другие отладочные макросы добавлены в секцию `.bss`.

Для отладочных целей и для соблюдения best practices введем также и создание стекового фрейма. [4]. Стоит заметить, что в портированной версии используется более привычный для Unix-подобных систем AT&T синтаксис ассемблерных инструкций (см. Листинг 12).

Листинг 12: Портированная версия RTLWriteChar

```
.section .data
RTLWriteIntegerBuffer: .byte 0x3c, 0x57, 0x72, 0x69, 0x74, 0x69, ...

.section .text
RTLWriteChar:
    pushall
    movq    %rsp,   %rbp    #form stack frame

    movq    $1,     %rax    #syscall #1 = Write()
    movq    $1,     %rdi    #param1 = write_to = 1 = stdout
    movq    %rbp,   %rsi    #p2 = write_from = %rbp = stack
    addq    $72,    %rsi    #reach arg on top of stack
    movq    $1,     %rdx    #p3 = count = single_byte
    syscall

    popall
    ret     $8              #remove arg from stack
```

Портируем следующую функцию, RTLWriteInteger. Ее листинг может быть опущен. Стоит лишь заметить, что эта функция используется для вывода целых чисел на stdout и она обращается к RTLWriteChar. Отдельно скомпилируем и соберем с помощью ld имеющийся рантайм:

```
gcc -c rtl64.s
ld rtl64.o -g -o rtl64
```

Протестируем так все имеющиеся функции. На этом этапе появилась еще одна типичная для работы с legacy кодом проблема — функция RTLReadInteger оказалась нерабочей потому, что прочитанное из stdin значение сохранялось в `EAX`, а затем просто затиралось при восстановлении регистров из стека. Судя по состоянию репозитория и отсутствию коммитов, исправляющих баг, функция оставалась нерабочей на протяжении 10(!) лет. После уведомления об этом автора, функция спустя какое-то время была исправлена (значение регистра в стеке теперь обновляется). Однако, после исправления ошибки, автором не было произведено должного тестирования и обнаружилась более «скрытая» проблема. Так, во время тестирования оригинала, была обнаружена невозможность многократного вызова функции `ReadLn`. Значение второго и последующих вызовов не доходит до «адресата». Данный баг был зафиксирован в issue к проекту на GitHub и на момент написания данного материала (февраль, 2017) исправлен не был. Это еще раз поднимает вопрос эффективности работы с legacy кодом.

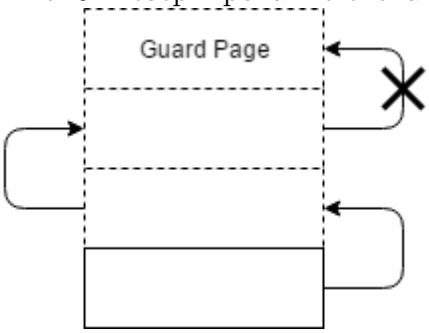
Портированная функция была приведена в соответствие с оригиналом.

На данном этапе новые функции работают корректно и ведут себя предсказуемо. Портируем аналогично все остальные функции рантайма.

Теперь, когда имеется полный набор RTL функций, необходимо подготовить окружение — зарезервировать стек так, как это делает BeRo. Чтобы инициализировать достаточный для работы программы стек можно пройти по нему, записывая какие-

либо значения через некоторые промежутки, пока не встретится Guard Page, ограничительная страница — см Рисунок 5.

Рис. 5: Резервирование стека



При более детальном изучении проблемы выяснилось, что для каждого запущенного процесса Linux запоминает список регионов виртуальной памяти и в случае ошибки обращения к странице, проверяет, не произошел ли Seg fault. Если не произошел, значит было обращение к неинициализированной странице, а значит необходимо лишь выделить еще одну страницу виртуальной памяти и добавить к региону. Столкнуться с Guard Page можно при этом только при использовании некоторых функций дебаггинга, содержащих `malloc`. Значит, нет необходимости в аналогичных действиях в новой библиотеке.

Осталось лишь скомпилировать библиотеку в «заглушку» для вставки ее в BTPC. Так как программа для перевода написана на Delphi Pascal'e, необходимо найти все необходимые инструменты для ее запуска, что в 2017 году – не сама простая задача. Но, поскольку цель подобной программы предельно ясна, было принято решение переписать эту программу на более удобном для парсинга файлов языке – *C++*. С помощью новой программы rtl64topas переводим скомпилированную библиотеку в строковый формат и вставляем ее на место заглушки BTPC – в функцию `EmitStubCode`.

На этом можно считать, что библиотека RTL портирована и на новой платформе имеется набор функций, аналогичных оригинальной RTL.

4.2 Генерация кода

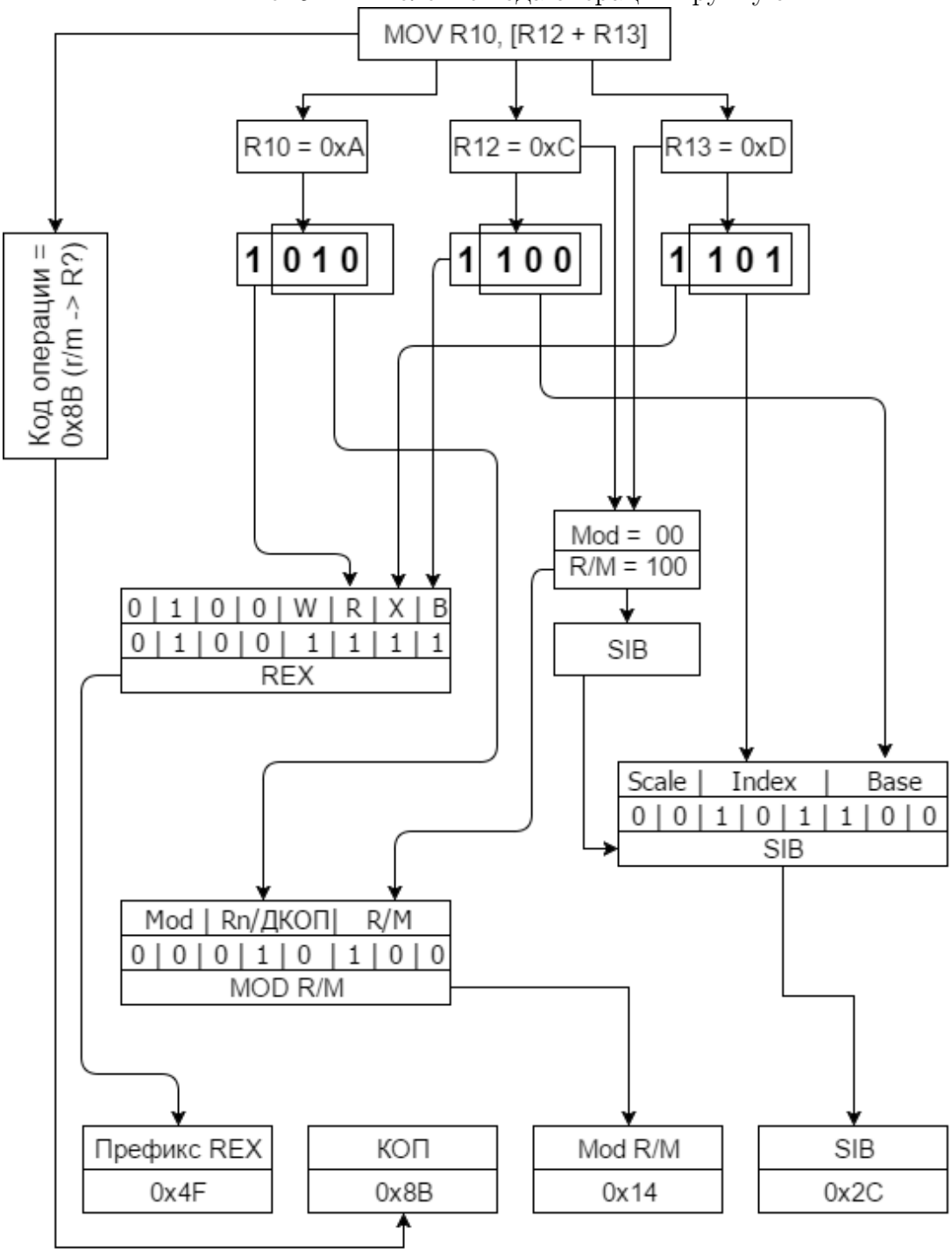
Следующим этапом является изменение генерации кода, порождаемого на целевой платформе. Необходим набор инструкций x64.

Переведем инструкцию «`MOV R10, (R12 + R13)`» (в AT&T-синтаксисе соответственно: `mov (%r12, %r13), %r10`) вручную. Для этого понадобятся справочные материалы курса «Низкоуровневое программирование» или же руководство разработчика для архитектуры Intel [3].

1. Переводим номера операндов в двоичный вид. Регистр-приемник – `R10` подходит под формат «R?».

2. Находим инструкцию `MOV` в таблице кодов инструкций i8086+. Нужен формат «r/m →R?». Итого, КОП инструкции = 0x8B.

Рис. 6: Вычисление кода операции вручную



3. `R10` в двоичном представлении имеет длину 4 бита, а значит не входит в отведенные 3 бита Rn байта ModR/M.

4. Для представления подобных «больших» регистров понадобится байт «Префикс REX».

5. Старший бит `R10` уходит в бит R префикса REX и расширяет Rn в байте ModR/M.

6. Согласно таблице ModR/M для 32-разрядных инструкций, биты R/M байта ModR/M = 100, а биты Mod = 00. Это дает дополнительный байт SIB.

7. Старший бит регистра `R12`, равный 1, взводит бит X в префиксе REX и расширяет номер индекса в SIB.

8. Байт SIB выглядит, как `[#Base + #Index*`2^{Scale}`]`. Базу образует регистр `R12`. 3 младших его бита = 100 и устанавливаются на место 3х битов Base в SIB.

9. Индекс(Index) в SIB составляют 3 младших бита регистра `R13` = 101.

10. Так как в инструкции простое сложение (1 регистр + 1 регистр), биты Scale в байте SIB = 00 и дают $2^{Scale} = 2^0 = 1$.

11. Старший бит регистра `R13` уходит в бит B префикса REX. Это расширяет набор базового регистра в байте SIB.

12. Сконкатенируем байт префикса REX: «0100» + «W:1» + «R:1» + «X:1» + «B:1» = 01001111, что в hex записи дает 0x4F. Префикс REX найден.

13. Сконкатенируем байт ModR/M: «Mod:00» + «Rn:010» + «R/M:100» = 00010100, что дает байт 0x14.

14. Сконкатенируем наконец байт SIB: «Scale:00» + «Index:101» + «Base:100» = 00101100, что дает байт 0x2C.

Путем конкатенации полученных байтов, имеем инструкцию «`MOV R10, (R12 + R13)`» в виде 0x4F 0x8B 0x14 0x2C. На Рисунке 6 описанный выше процесс представлен более наглядно.

Подобных инструкций, требующих перевода в BTPC около 70, без повторений. Очевидно, что этот способ совсем не оптимален и очень затратен. Один упущенный бит меняет инструкцию или создает несуществующую. В связи с этим и после того, как процесс перевода стал понятен, можем воспользоваться более быстрым путем — **выписать необходимые инструкции** в формате x64, **скомпилировать** файл а затем **дизассемблировать** его. На выходе получим все, что необходимо, чтобы заменить генерируемые байты в процедурах, подобных Листингам 6 и 7.

После этого кодогенерацию можно также считать портированной, так как теперь инструкции порождаются в новом, 64-битном формате.

4.3 Формирование ELF файла

Теперь, когда имеется портированная библиотека в функции `EmitStubCode`, а новый код генерируется в формате x64, остается лишь пропатчевать ELF файл, собранный в заглушке, подобно патчеванию PE файла у BeRo.

При компиляции и обычной линковке командами

```
gcc -c rtl64.s
ld rtl64.o -g -o rtl64
```

в файл может попасть множество «мусора» – функции библиотек, таблицы символов, лишние секции и прочие метаданные, в которых нет надобности при загрузке и выполнении файла. Мусором в данном случае считается все, чего нет в «образцовом» и минималистичном исходном PE файле. На Листинге 13 представлены секции полученного «обычной» линковкой файла, найденные утилитой readelf:

```
readelf -S rtl64
```

Особенностью ELF формата является наличие пустой секции. Она всегда имеет нулевой номер. Далее идет секция кода `.text`, затем данных `.data`. После чего следуют отладочные секции – секция с именами секций `.shstrtab` (Section header string table), секция с таблицей символов `.symtab` и наконец секция строк- меток в ассемблерном коде `.strtab`.

Листинг 13: Секции файла rtl64

```
[#]  Name              Type              Address           Offset
     Size              Size.Ent          Flags    Link Info  Align
[ 0]                   NULL              0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
[ 1] .text             PROGBITS          00000000004000b0  000000b0
     0000000000000339  0000000000000000  AX       0     0     1
[ 2] .data             PROGBITS          00000000006003e9  000003e9
     0000000000000067  0000000000000000  WA       0     0     1
[ 3] .shstrtab         STRTAB            0000000000000000  00000450
     0000000000000027  0000000000000000           0     0     1
[ 4] .symtab           SYMTAB            0000000000000000  000005f8
     0000000000000420  0000000000000018           5    40     8
[ 5] .strtab           STRTAB            0000000000000000  00000a18
     000000000000024e  0000000000000000           0     0     1
```

На первом же шаге выявляется проблема: **секция кода, к которой должен был конкатенироваться код, находится в самом начале файла**.

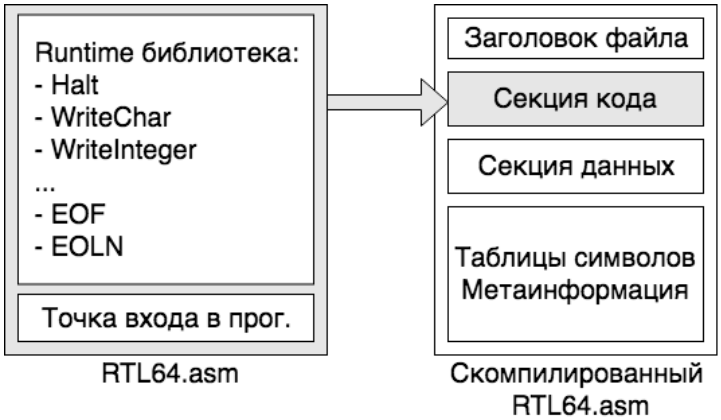


Рис. 7: Сгенерированный файл с секцией кода в начале фала и метаинформацией

Схематично получившийся файл изображен на Рисунке 7. Таким образом, дописывание кода после метки `ProgramEntryPoint` в RTL сделает файл неработоспособным. В эталонном файле присутствует лишь одна секция 4. Постараемся добиться максимально приближения к этому, но с учетом введенных при рефакторинге дополнительных секций (секций инициализированных и неинициализированных данных). То есть, необходимо оставить лишь 2 секции (кода и данных), при этом приблизив секцию кода к концу файла.

Лишние секции никак не заданы в исходном коде RTL64, а значит попадают в файл на стадии линковки. Это заметно по размеру файла после этих действий. **Зададим линкеру не генерировать мусор в виде функций библиотеки stdlib и секций с таблицами**:

```
ld rtl64.o -g -o rtl64 -nostdlib -s
```

При повторном чтении ELF'а оказывается, что секция `shstrtab` осталась и все так же хранит имена других секций. Нулвая секция тоже на месте, но это несущественно.

В комплекте binutils есть утилита для редактирования в том числе и исполняемых файлов. С помощью этой утилиты, strip, **«вырезаем» оставшуюся секцию** `shstrtab`:

```
strip -R shstrtab rtl64
```

Однако, вновь прочитав файл на предмет секций, обнаруживаем там `shstrtab`. Так как секция хранит имена других секций, она *всегда* **генерируется при линковке**. Однако, для выполнения, согласно спецификации, она не нужна. Так что можно будет очистить ее при генерации кода. Остается проблема с тем, что секция кода все еще в глубине файла и «прикрыта» секцией данных.

Для того, чтобы поменять местами секции, необходимо использовать специальный линкер-скрипт для ld. Это файл, определяющий какие данные куда отобразятся при линковке. В том числе здесь задается и порядок вхождения секций. Выполним линковку согласно скрипту на Листинге 14, заменив им стандартный скрипт (команда «`ld --verbose`»). Для этого запустим линкер с новой опцией:

```
ld rtl64.o -g -o rtl64 -T linkerScript.ld -nostdlib -s
```

В представленном скрипте указана точка входа –`_start`, адреса размещения секций данных и кода, и выделяемая память на стеке и в куче.

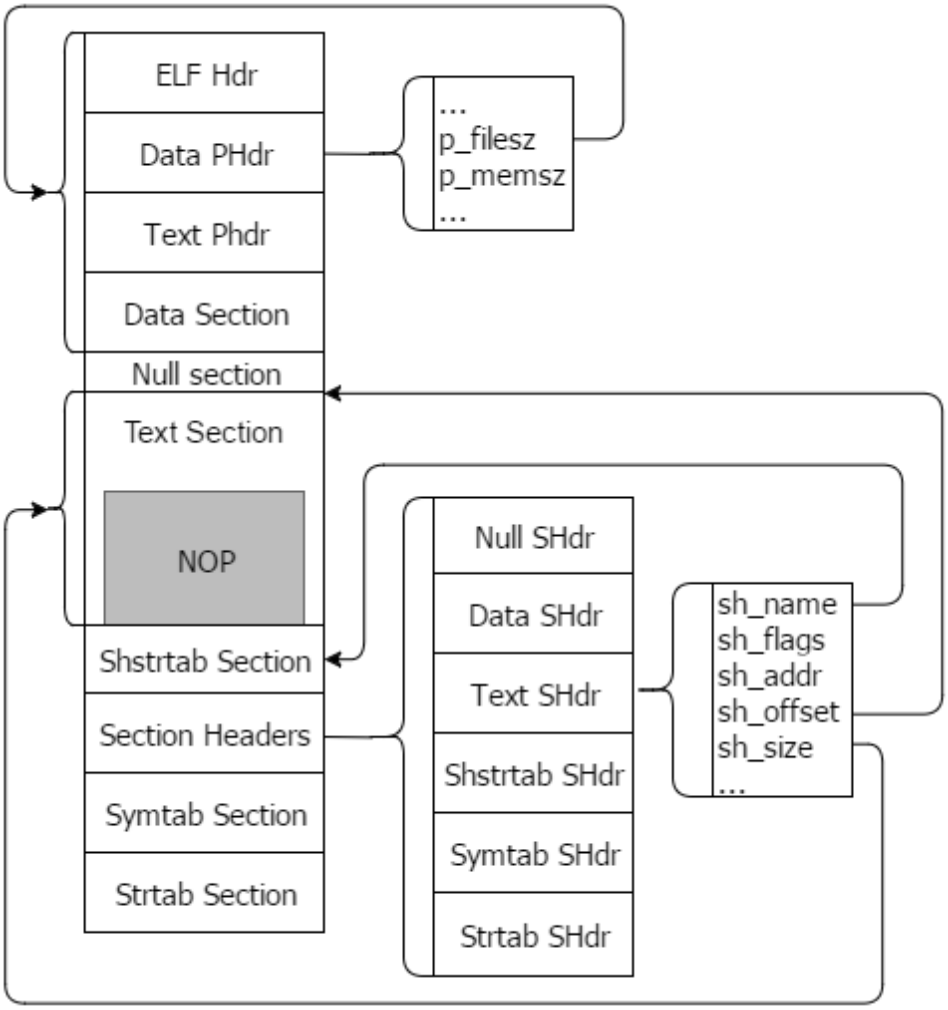
Листинг 14: Скрипт линкера ld

```
stack_size = 4194332;
heap_size = 256;
ENTRY(_start)
SECTIONS
{
        . = 0x4000b0;
        .data : { *(.data) }
        .bss :  { *(.bss)  *(COMMON) }
        . = 0x6000d3;
        .text : { *(.text) }
}
```

После линковки по этому скрипту, секции кода и данных поменялись местами. Секция `shstrtab` все также замыкает файл. Но из-за минималистичности скрипта, линкер перестал делать какие-либо оптимизации и вычисления, полностью положившись на файл, в котором ничего кроме порядка секций не задано. После некоторых манипуляций приводим текущий скрипт к виду стандартного скрипта (`verbose`), но с описанными в Листинге 14 изменениями. Получаем оптимизированный скомпонованный файл. Обновляем заглушку `EmitStubCode` в BTPC.

Пробуем собрать компилятор, отключив генерацию кода. На данном этапе компилятор лишь выводит заглушку. Выводит успешно. Файл, наполненный `NOP`'ами для выравнивания под размер в 255 может быть запущен и также предсказуемо «сваливается» в Seg fault.

Рис. 8: Устройство конечного ELF файла



Однако, при тестировании есть проблема – файл собран слишком минималистично и в нем отсутствует какая-либо полезная для отладки информация. Несмотря на то, что код RTL64 был тщательно изучен, отлаживать и тестировать ассемблерный код без таблицы символов и без единой именованной метки очень затруднительно. В

связи с этим было принято решение вернуть на место секции `symtab` и `strtab`, убрав соответствующие опции линкера.

На данный момент RTL64 может быть протестирован и отлажен, так как внутри устроен абсолютно корректно с точки зрения отладки. Это одно из преимуществ новой библиотеки перед оригиналом, где отсутствуют какие-либо метки, а многие команды распознаются неправильно из-за наличия байтов данных в коде, которые дизассемблером путаются с реальными командами.

Теперь необходимо отправить на выполнение весь сгенерированный код, содержащий логику компилируемой программы. Для этого необходимо отредактировать поля файла согласно внесенным изменениям.

Поскольку секция кода, хоть и была перенесена в конец, все равно осталась скрытой за отладочными секциями, необходимо **«разрезать» файл надвое, вставить туда код и «склеить» обратно**. На данный момент представление файла полностью соответствует изображенному на Рисунке 8.

Кав видно из Рисунка 8, файл выстроен в соответствии описанными ранее требованиями и допусками: секция текста идет после секции данных, за ней следуют отладочные секции, среди которых имеется и таблица заголовков секций. На этой схеме также показаны поля ELF, рассматривавшиеся в самом начале работы, а именно – поля программных заголовков (в данном случае Data program header) и поля заголовков секций, такие, как *имя* (sh_name), содержащее смещение внутри секции `shstrtab` по которому лежит название этой секции, *смещение* (sh_offset), указывающее на начало секции кода в файле и *размер секции* (sh_size).

Код компилируемой программы будет порождаться в секции кода поверх операций `NOP`, положение которых отмечено серым цветом.

Таблица 2: Редактируемые поля

Название поля	Смещение до поля	Значение
Elf_hdr.e_shoff	0x28	+= injSize
Text_phdr.p_filesz	s(Elf_hdr)+s(p_hdr)+0x20	+= injSize
Text_phdr.p_memsz	s(Elf_hdr)+s(p_hdr)+0x28	+= injSize
Text_shdr.sh_size	Elf_hdr.e_shoff+injSize+2*s(s_hdr)+0x20	+= injSize
Shstrtab_shdr.sh_offs	Elf_hdr.e_shoff+injSize+3*s(s_hdr)+0x18	+= injSize
Symtab_shdr.sh_offs	Elf_hdr.e_shoff+injSize+4*s(s_hdr)+0x18	+= injSize
Strtab_shdr.sh_offs	Elf_hdr.e_shoff+injSize+5*s(s_hdr)+0x18	+= injSize

Итого, чтобы добиться работоспособности подобного файла, необходимо исправить как минимум поля, указанные в Таблице 2.

Значения всех представленных полей должны быть увеличены на размер сгенерированного кода (injSize). В столбце «Смещение до поля» функция s(X) обозначает размер соответствующей структуры X, и далее прибавлено смещение конкретного поля внутри структуры. Значение Elf_hdr.e_shoff — значение этого поля *после* редактирования этого поля в первой строке таблицы. Поля указаны в том порядке, в котором они должны быть отредактированы, так как среди значений, как следует из Таблицы 2, наблюдается зависимость.

Все смещения и значения могут быть с легкостью вычислены вручную, однако это не самый эффективный подход, так как высока вероятность ошибки в таком случае. С целью вычисления полей была доработана программа rtl64topas.cpp. Она не только разбивает файл на 2 набора строк (начало файла до секции кода включительно и конец файла), но также вычисляет смещения и значения полей, после чего с помощью функций `OutputCodePutInt32(offset, value)` значения записываются в итоговый набор опкодов инструкций.

Листинг 15: Редактирование полей ELF файла

```
const EndStubSize =        $8f7;
      ElfHdrShoff_0 =      $488;
      TextPhdrFilesz_0 =   $349;
      OffsElfHdrShoff =    $28;
      OffsTextPHdrFilesz = $98;
  ...
  InjSize := OutputCodeDataSize - EndStubSize - PECodeStart;
{ElfHdr.e_shoff, 8b}
  OutputCodePutInt32(OffsElfHdrShoff + $1,      ElfHdrShoff_0 + InjSize);
  OutputCodePutInt32(OffsElfHdrShoff + $1 + $4, 0);
{TextPhdr.p_filesz, 4b}
  OutputCodePutInt32(OffsTextPHdrFilesz + $1, TextPhdrFilesz_0 + InjSize);
```

Так как байты представлены в формате Little Endian (по старшему адресу – старшие байты) и с допущением, что генерируемый промежуточный код имеет 32-х разрядный формат, а значит не может превзойти или «добраться» по размеру до x64 кода, старшие 4 байта (расположенные по старшему адресу) 8-байтных полей могут быть обнулены. Пример такого редактирования поля – в Листинге 15. К смещению добавляется один байт из-за устройства процедуры `OutputCodePutInt32`, которая редактирует байт по текущему адресу, а не со следующего байта. Суффикс `_0` означает исходное значение поля и формируется при создании заглушки.

Еще одной доработкой BTPC является двукратное увеличение значения `Value` (данных для операции, представленной предыдущим значением) там, где Value является *смещением*. Например, в операциях `subq Value, %rsp` (формально, Value в подобной инструкции не является смещением, однако вычитание у BeRo реализовано прибавлением отрицательного значения, а сами инструкции вычитания используются для аллокации переменных в стеке, в связи с чем значение должно быть увеличено в 2 раза) или же `movq Value(%rsp), %rax`.

На этом основную часть портирования можно считать завершенной.

Данный проект (A Bauman BTPC 64, *AB BTPC*) в завершенном состоянии доступен из репозитория на GitHub по адресу github.com/avbelyaev/A-Bauman-BTPC-64

5 ТЕСТИРОВАНИЕ И ОТЛАДКА

Далее следует отладка в стиле TDD разработки (Test Driven Development), т.е. необходимо получить эталонное поведение, сравнивая поведение оригинала и новой

реализации на тестах. В случае «падения» необходимо дизассемблировать и изучить причину подробнее.

Далее следует *простой* пример одной итерации такого TDD подхода к отладке (при проведении полноценной раскрутки на вход компилятору поддается компилятор и «отловить» ошибку среди кода, сгенерированного из 3000 строк исходного кода компилятора весьма трудно. Для этого сначала производится отладка наиболее простых частей и затем идет усложнение подаваемых на вход программ. Последней программой, поданной на вход, будет сам компилятор) На Рисунке 9, в левом столбце содержится код, сгенерированный новым компилятором по программе из Листинга 16, в правом столбце — код, порожденный исходным компилятором.

Листинг 16: Тестирование работы функции, обращающейся к 4 функциям RTL

```
program RTLFucnTest;
var x:integer;
procedure simpleProc(s:integer);
begin
  Write('tst');
  WriteLn(s + 3);
end;

begin
  x := 5;
  simpleProc(x);
end.
```

Как видно из результата сравнения дизассемблирования, ассемблерная функция (метка) порождается на том же месте, что и процедура в верхнеуровневом коде. Далее «пушится» значение аргумента и следует переход на метку функции. Можно также заметить, что печать строки это на самом деле печать одного символа $n = strlen$ раз (функцией RTLWriteChar по смещению 0x8), а вся арифметика происходит прямо в стеке. Далее следуют еще 2 RTL функции – RTLWriteInteger и RTLWriteLn по смещениям 0x10 и 0x18 соответственно. Заканчивается программа переходом на функцию RTLHalt по смещению 0.

В данном случае имеется проблема с аллокацией места в стеке под переменные – инструкция `sub #0x4, %rsp` сдвигает стек на 4 байта, что приводит к коллизии с остальными 8-байтовыми значениями в нем. Если теперь изменить функцию, порождающую эту инструкцию, то оба файла отработают корректно. Первый – под Linux x64, второй – под Windows 32.

Если же проблема на уровне заглушки и требует пересборки библиотеки, то необходимо **отредактировать код, скомпилировать и слинковать** его, затем **преобразовать в набор строк, вставить выходной результат программы rtl64topas на соответствующее место в BTPC**. Затем вновь идут итерации тестирования.

Таким образом мы получаем полностью работоспособный компилятор, который порождает ELF файлы для использования под Linux, но все еще не реализует главную особенность BTPC — самоприменимость. Прежде, чем приступить к отладке

Рис. 9: Сравнение сгенерированных частей

e937000000	jmpq loc_00a0049d	e931000000	jmp loc_004013c5
	unknown **func_00a00466**		*unknown* **func_00401394**
4883ec08	sub $0x4,%rsp	83ec04	sub $0x4,%esp
6a65	pushq $0x65	6a65	push $0x65
ff5608	callq *0x8(%rsi)	ff5604	call *0x4(%esi)
6a78	pushq $0x78	6a78	push $0x78
ff5608	callq *0x8(%rsi)	ff5604	call *0x4(%esi)
6a6d	pushq $0x6d	6a6d	push $0x6d
ff5608	callq *0x8(%rsi)	ff5604	call *0x4(%esi)
488b442410	mov 0x10(%rsp),%rax	8b442408	mov 0x8(%esp),%eax
50	push %rax	50	push %eax
4883042403	addq $0x3,(%rsp)	83042403	addl $0x3,(%esp)
58	pop %rax	58	pop %eax
48890424	mov %rax,(%rsp)	890424	mov %eax,(%esp)
488b0424	mov (%rsp),%rax	8b0424	mov (%esp),%eax
50	push %rax	50	push %eax
6a01	pushq $0x1	6a01	push $0x1
ff5610	callq *0x10(%rsi)	ff5608	call *0x8(%esi)
ff5618	callq *0x18(%rsi)	ff560c	call *0xc(%esi)
4883c408	add $0x8,%rsp	83c404	add $0x4,%esp
c20800	retq $0x8	c20400	ret $0x4
loc_00a0049d:		loc_004013c5:	
4883c408	add $0x8,%rsp	83c404	add $0x4,%esp
6a05	pushq $0x5	6a05	push $0x5
e8beffffff	callq func_00a00466	e8c5ffffff	call func_00401394
ff6600	jmpq *0x0(%rsi)	ff6600	jmp *0x0(%esi)

самоприменимости, отметим препятствующие факторы.

5.1 Особенности исходной реализации

Чтобы в дальнейшем избежать ошибок, стоит отметить имеющиеся проблемы, которые появлялись по мере работы по портированию:

- В RTL нет ни одного комментария на несколько сотен строк ассемблерного кода притом, что в некоторых функциях используются не самые очевидные решения.
- В RTL данные хранятся внутри кода, в нарушение практик по безопасному программированию, что еще и приводит к невозможности отладить работу, дизассемблировав файл (можно рассматривать это, как своеобразный прием антиотладки – техники злоумышленников по скрытию реальных действий программы), так как данные принимаются за инструкции.
- Отсутствуют фреймы функций, что серьезно затрудняет отладку.
- Содержимое регистров, которое можно принять за «мусор» после операций, спустя некоторое количество инструкций будет переиспользовано и дальнейший код ожидает увидеть там определенное значение. В других местах мусор перезаписывается.

- Наличие ошибок в коде, одна из которых просуществовала 10 лет.
- В BTPC присутствует несколько десятков комментариев (многие из которых дублируют названия функций, в которых эти комментарии указаны) на почти 3000 строк кода собственного множества Pascal'a. Какая-либо иная документация отсутствует.
- Для формирования «заглушки» используется *самописное* ПО, найти которое *невозможно*.
- Множество никак не аннотированных «магических констант» в коде.

Описанные выше проблемы действительно способны вызвать значительные затруднения в работе с legacy кодом даже для автора кода спустя большое количество времени, так что в новом проекте была предпринята попытка избавиться хотя бы от части проблем.

5.2 Раскрутка компилятора

Раскрутка (bootstrapping) — это итерационный процесс создания самоприменимого компилятора, на каждой итерации которого получается компилятор для все большего подмножества языка. BTPC является самоприменимым и в рамках данной работы рассматривается лишь перенос текущей функциональности. Для начала подадим на вход компилятору исходный код себя же и, получив отрицательный результат, перейдем к итерационной отладке.

Рассмотрим `main`-функцию BTPC: она реализует фазы компиляции в виде нескольких функций на Листинге 17.

Листинг 17: Фазы компиляции в BTPC

```
  ReadChar;
  GetSymbol;
  Write('ReadChar and GetSymbol succeeded');
{ Expect(SymPROGRAM);
  ...
  EmitOpcode2(OPHalt);
  Check(TokPeriod);
  AssembleAndLink; }
```

Отключим все функции и будем подключать их одну за другой, «отлавливая» ошибки – локализуя, сверяясь с оригиналом и исправляя соответствующим образом.

После того, как все функции отлажены, выполним самоприменение, получив кросскомпилятор:

```
btpc.exe < btpc64.pas > btpcCrossWin.exe
```

Сделаем еще шаг, получив его Linux–версию:

```
btpcCrossWin.exe < btpc64.pas > btpc64Linux
```

Еще один, «контрольный» шаг, уже на Linux:

```
btpc64Linux < btpc64.pas > btpc64Check
```

Проверив командой «`diff btpc64Linux btpc64Check`» отличия этих двух версий и убедившись в «нулевом выводе» (файлы полностью идентичны), считаем работу завершенной.

Компилятор портирован и самоприменим.

6 ЗАКЛЮЧЕНИЕ

В результате данной работы был портирован самоприменимый компилятор, написанный более 10 лет назад.

В ходе работы было изучено внутреннее устройство исполняемых файлов разных форматов для разных операционных систем, был изучен процесс компиляции, кросскомпиляции, линковки и сборки выходного файла, а также был установлен контроль над данным процессом, что привело к желаемому результату. Была изучена генерация кодов операций из инструкций, а затем и упаковка полученных кодов.

Во время работы было встречено и преодолено множество проблем, характерных для работы с legacy кодом.

Данная работа, несмотря на наличие работоспособного портированного компилятора, может быть названа выполненной лишь отчасти, так как, в оригинальном коде присутствует еще множество ошибок и, вероятно, кто-нибудь с ними столкнется, после чего придется вновь углубляться в изучение кода, так как большая часть кода нового компилятора досталась ему по наследству от старого.

Задачи, поставленные в рамках данной работы, тем не менее, выполнены.

Список литературы

[1] Peering Inside the PE: A Tour of the Win32 Portable Executable File Format [Электронный ресурс] — Режим доступа: https://msdn.microsoft.com/en-us/library/ms809762.aspx

[2] Executable and Linkable Format (ELF) Specification [Электронный ресурс] — Режим доступа: http://www.skyfree.org/linux/references/ELF_Format.pdf

[3] Intel 64 and IA-32 Architectures Developer's Manual [Электронный ресурс] — Режим доступа: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[4] С. Зубков, Assembler для DOS, Windows и Unix —МСК: ДМК Пресс, 2015. — 206 с.

[5] PE (Portable Executable): На странных берегах [Электронный ресурс] — Режим доступа: https://habrahabr.ru/post/266831/