|  |  |  |
| --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  РОССИЙСКОЙ ФЕДЕРАЦИИ  Федеральное государственное бюджетное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Кафедра математического обеспечения вычислительных систем | |
| **декомпиляция машинного кода архитектуры x86-64 на язык ПРОГРАММИРОВАНИЯ c** | | |
| *Курсовая работа* | | |
|  | | |
|  | | |
|  | | Работу выполнил студент  группы ПМИ-4 3 курса механико-математического факультета  Пухов Н.А. |
|  | | Научный руководитель:  старший преподаватель кафедры МОВС  Лядова Л.Н.  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г. |
|  | |  |
| Пермь 2021 | | |

**Аннотация**

текст.

Оглавление

Оглавление 3

Введение 5

Глава 1. Анализ существующих решений 7

Глава 2. Теоретические основы декомпиляции 11

2.1. Постановка задачи декомпиляции 11

2.2. Конвейер декомпилятора 11

2.2.1. Декодирование 12

2.2.2. Трансляция инструкций языка ассемблер в P-код 12

2.2.3. Построение графа потока управления из P-кода 13

2.2.4. Построение абстрактного синтаксического дерева 14

2.2.5. Оптимизация 15

2.2.6. Символизация 16

2.2.7. Типизация 17

2.2.8. Генерация кода 17

2.3. Стандартное преобразование типов языка C 17

2.3.1. Арифметические преобразования типов 17

2.3.2. Преобразование указателей 19

2.3.3. Преобразование чисел со знаком в числа без знака 20

2.3.4. Преобразование чисел без знака в числа со знаком 20

2.3.5. Преобразование чисел с плавающей запятой 20

2.3.6. Преобразования между целочисленным типом и типом с плавающей запятой 20

2.4. Язык ассемблера архитектуры X86-64 20

2.4.1. Размеры операндов команд 21

2.4.2. Команды общего назначения 21

2.4.3. Команды блока MMX 23

2.4.4. Команды блока XMM 24

2.5. Маски выражений 28

2.6. Адресация элементов многомерных массивов 29

2.7. Линеаризация графа потока управления 30

2.7.1. Условное выражение if 30

2.7.2. Условное выражение if-else 30

2.7.3. Цикл while 31

Глава 3. Проектирование декомпилятора 32

3.1. Этап анализа 32

3.1.1. Диаграмма прецедентов 32

3.1.2. Диаграмма активностей 33

3.2. Этап проектирования 34

3.2.1. Создание классов 35

3.2.2. Диаграмма классов 36

3.3. Этап реализации 38

3.3.1. Использование сторонних библиотек 38

3.3.2. Диаграмма компонентов 39

3.4. Реализация основных алгоритмов 39

3.4.1. Алгоритм построения графа потока управления 39

3.4.2. Итерационный алгоритм вычисления типов выражений 40

3.4.3. Алгоритм оптимизации выражений, связанных с памятью 41

Глава 4. Тестирование и сравнение 43

4.1. Тестрование декомпилятора 43

4.2. Сравнение с Ghidra Decompiler 43

4.2.1. Векторные и математические операции 43

4.2.2. Интенсивная работа со стеком 45

4.2.3. Сравнение скорости 45

Заключение 46

Библиографический список 48

Приложение A. Исходный код декомпилятора 49

Введение

На сегодняшний день существует большое количество разнообразных программ с открытым исходным кодом. Исходный код таких программ написан с использованием высокоуровневых языков программирования, среди которых C, C++, Java, Python и другие. Мотивация использования языков программирования с высоким уровнем абстракции обусловлена желанием иметь понятный, расширяемый код, который имеет возможность исполняться на разных видах операционных систем и архитектур процессора. Однако существует множество открытых программ, которые были написаны под определенную архитектуру процессора для конкретной операционной системы с использованием языка ассемблер. Среди такого рода программ можно выделить драйверы устройств или прошивки для микропроцессорной техники. Более того, программисты часто используют вставки кода на ассемблере в своих программах, написанных на высокоуровневых языках.

Использование ассемблера как низкоуровневого кода, который плохо понятен человеку, необходимо с целью повышения производительности программ. Более того, ассемблерный код имеет больше возможностей по оперированию устройством, чем код, написанный на языке высокого уровня. Но возникает большая проблема, связанная с чтением такого низкоуровневого кода другими людьми, которые могут даже не знать ассемблер. Решение данной проблемы состоит в использовании декомпилятора, который транслирует код из низкоуровневого представления в высокоуровневое. Среди существующих декомпиляторов можно выделить: IDA Hex-Rays Decompiler, Ghidra Decompiler, RetDec, Snowman, однако все они имеют определенные недостатки и не всегда генерируют высокоуровневый код в понятном человеку виде.

Объектом данного исследования является низкоуровневый код в ассемблерном или машинном представлении для архитектуры процессоров X86-64. Предмет исследования – представление такого кода в высокоуровневой форме.

Цель данной работы – декомпиляция кода на языке ассемблера архитектуры X86-64 в код высокоуровневого языка C в понятном человеку виде.

Для достижения поставленной цели должны быть решены следующие задачи:

1. Проанализировать существующие бесплатные программные реализации, которые позволяют декомпилировать код на языке ассемблера архитектуры X86-64 в высокоуровневое представление. Выявить критические проблемы этих реализаций, а именно понять, какие примеры низкоуровневого кода они не могут декомпилировать в понятном человеку виде.

2. Разработать собственные пути решения выявленных проблем. Для этого создать свой собственный программный продукт или взять готовую реализацию и исправить эти проблемы в ней.

# Анализ существующих решений

В данной главе производится критический анализ существующих программных реализаций декомпиляции кода по следующим критериям:

1. Бесплатность и открытость исходного кода.
2. Возможность встраивания в другие программы.
3. Высокая скорость работы.
4. Декомпиляция заданного участка машинного кода.
5. Использование промежуточного языка виртуальной машины (P-код).
6. Высокое качество генерируемого кода.
7. Полноценная поддержка векторных и математических операций.

На данный момент разработано несколько декомпиляторов, которые, в частности, позволяют транслировать код с языка ассемблера или непосредственно машинный код архитектуры X86-64 в высокоуровневый код языка C. Среди популярных декомпиляторов можно выделить:

1. IDA Hex-Rays Decompiler. Данный декомпилятор является платным с закрытым исходным кодом. Он является плагином к программе IDA Pro. Имеет поддержку архитектур процессора x86-64, ARM, MIPS. Отличительная особенность – наличие отладчика, благодаря которому можно в динамике увидеть работу сгенерированного кода.
2. RetDec. Данный декомпилятор был представлен компанией Avast. Он является бесплатным с открытым исходным кодом и имеет поддержку множества архитектур процессоров, среди которых x86-64, ARM, MIPS. Использует промежуточный язык виртуальной машины LLVM. Имеет возможность восстановления иерархии классов и виртуальных таблиц. Генерирует код на языках C и Python. Основной недостаток – генерация кода уступает по качеству коду, который генерирует Ghidra Decompiler. Это в первую очередь проявляется в использовании плохо читаемых предопределенных функций и макросов вместо стандартных операторов языка C. Еще одним недостатком является отсутствие возможности декомпиляции заданного участка кода, а не всего программного образа.
3. Snowman. Особенностью данного декомпилятора является возможность трансляции кода в язык C++. Также он может быть легко встроен в любую программу и может декомпилировать заданный участок машинного кода. Однако он имеет такой же недостаток, что у декомпилятора RetDec, а именно плохо читаемый код во многих случаях.
4. Ghidra Decompiler. Данный декомпилятор был представлен агентством национальной безопасности США. Использует промежуточный язык PCode абстрактной виртуальной машины. Главное его отличие от всех остальных реализаций заключается в использовании виртуальной машины Java в качестве среды для выполнения основной программы Ghidra, хотя он сам написан на языке C++. Отсюда вытекает главный недостаток – низкая скорость работы декомпилятора. Еще один недостаток – это отсутствие возможности встроить данный декомпилятор в свою программу, а также декомпилировать заданный участок машинного кода. Однако главное преимущество данного декомпилятора – генерация им хорошо читаемого кода в большинстве случаев. Это достигается за счет более совершенных эвристических алгоритмов генерации высокоуровневого кода.

В табл. 1.1 представлено соотнесение декомпиляторов изложенным выше требованиям.

Таблица .. Соотнесение декомпиляторов требованиям

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Требование** | **IDA Hex-Rays Decompiler** | **RetDec** | **Snowman** | **Ghidra Decompiler** |
| Бесплатность и открытость исходного кода | − | + | + | + |
| Возможность встраивания | − | + | + | − |
| Декомпиляция заданного участка машинного кода | − | − | + | − |
| Высокая скорость работы | + | + | + | − |
| Использование промежуточного языка виртуальной машины | − | + | − | + |
| Высокое качество генерируемого кода | + | − | − | + |
| Полноценная поддержка векторных и математических операций | − | − | − | − |

Важно отметить, что общая проблема всех перечисленных декомпиляторов – плохая работа с тем кодом, где преобладают векторные или математические операции и где наличествует пересылка данных из одного участка памяти в другой. Также большинство декомпиляторов могут анализировать и декомпилировать только цельные программные образы вместо заданных участков машинного кода.

Для решения двух названных выше основных проблем существует два варианта: улучшение готовой реализации или создание своей собственной с нуля. Рассмотрим первый вариант.

1. IDA Hex-Rays Decompiler. Данный декомпилятор не имеет открытого исходного кода, поэтому не может быть улучшен.
2. RetDec. Данный декомпилятор не подходит по многим причинам, в частности он не имеет возможности качественной декомпиляции заданного участка машинного кода. Таким образом улучшение данного декомпилятора приведёт к необходимости фундаментальных изменений, что нецелесообразно с точки зрения временных затрат.
3. Snowman. Данный декомпилятор не использует промежуточный язык виртуальной машины, что является критическим недостатком, поскольку не дает возможности виртуально вычислять те константные выражения, которые используют память компьютера. Таким образом улучшение данного декомпилятора приведёт к необходимости фундаментальных изменений, что нецелесообразно с точки зрения временных затрат.
4. Ghidra Decompiler. Данный декомпилятор не может встраиваться в другие программы, имеет привязку к виртуальной машине JVM, поэтому не может быть улучшен.

На основании вышеприведенных умозаключений по каждому декомпилятору было принято решение создать собственную программную реализацию с нуля.

# Теоретические основы декомпиляции

В данной главе описываются теоретические основы процесса декомпиляции и его основные этапы.

## Постановка задачи декомпиляции

Пусть - отображение, которое преобразует высокоуровневый код  на некотором языке программирования  в машинный код  некоторого процессора . Будем называть отображение  компилятором. Тогда отображение  будет являться идеальным декомпилятором.

Для компилятора, транслирующего код на языке C в машинный код архитектуры процессора X86-64, соответствующего идеального декомпилятора не может существовать. Это связано с тем, что в процессе компиляции безвозвратно теряется информация о названии переменных и функций, удаляются комментарии в коде. При использовании оптимизатора в компиляторе, теряется также первоначальная структура исходного кода.

Таким образом, задача сводится к построению приближенной функции , такой что . Функцию  назовем декомпилятором.

## Конвейер декомпилятора

Процесс декомпиляции можно представить в виде нескольких основных этапов преобразования данных из низкоуровневого кода в высокоуровневый код. На рис. 2.1 изображена схема конвейера декомпилятора, которая будет использована для решения поставленной задачи. Можно выделить несколько основных этапов:

1. Декодирование.
2. Трансляция в P-код.
3. Построение графа потока управления.
4. Построение абстрактного синтаксического дерева.
5. Оптимизация.
6. Символизация.
7. Типизация.
8. Генерация кода.

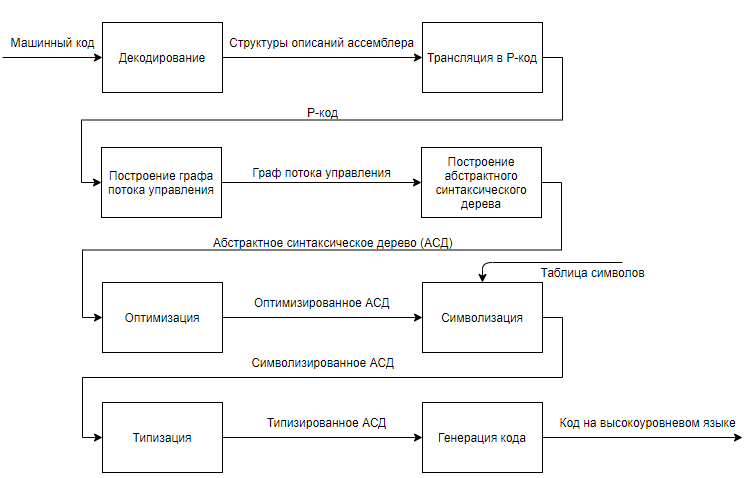


Рисунок 2.1. Конвейер декомпилятора

### Декодирование

Первым этапом является декодирование машинного кода, который представляет из себя последовательность байт, в структуры описания инструкций языка ассемблер. Структуры описания инструкций содержат информацию о типе инструкции, значениях и типах операндов, их количестве и т.д. Например, из инструкции “mov [rsp+0x10], 0x1” можно извлечь следующую информацию:

1. Тип инструкции – mov.
2. Первый операнд имеет указатель, который состоит из базового адреса, хранящегося в регистре “rsp”, и числового смещения “0x10”.
3. Второй операнд имеет числовое значение “0x1”.

### Трансляция инструкций языка ассемблер в P-код

Вторым этапом декомпиляции является преобразование структур описания инструкций языка ассемблер в аппаратно-независимый P-код. P-код – концепция аппаратно-независимого исполняемого кода в программировании, часто его определяют как «Ассемблер для гипотетического процессора», который служит необходимым промежуточным абстрактным слоем между аппаратно-зависимым языком ассемблер X86-64 и высокоуровневым языком C. P-код содержит минимальный набор команд, благодаря которому можно реализовать любой алгоритм, таким образом P-код имеет полноту по Тьюрингу. На рис. 2.2 изображен пример листинга P-кода для инструкций “AND” и “OR”.

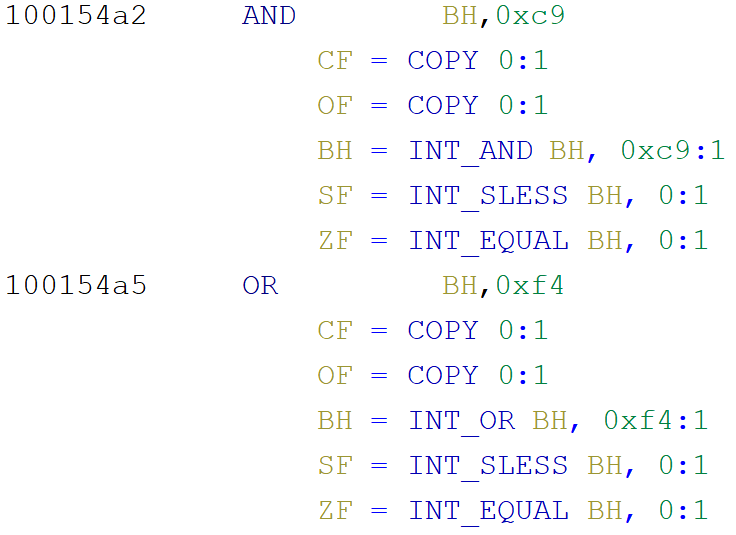


Рисунок 2.2. Пример P-кода

### Построение графа потока управления из P-кода

Третьим этапом является построение ориентированного графа потока управления из сгенерированного на предыдущем этапе листинга инструкций P-кода. В качестве вершин графа выступают блоки инструкций P-кода, которые не содержат команд условных или безусловных переходов за исключением последней команды, которая всегда осуществляет переход в другую область P-кода. В качестве рёбер ориентированного графа используются направления условных или безусловных переходов. Из одной вершины может выходить до двух ребер.

Вершина, из которого нет выходящих ребер, является конечной и всегда содержит последней инструкцией команду возврата, представляющую из себя безусловный переход в другую область программы. Конечных вершин может быть несколько. Вершина, в которую не входит ни одно ребро, является начальной вершиной в графе потока управления и всегда представлена одна. На рис. 2.3 изображен пример графа потока управления с двумя конечными вершинами. В блоках 1.1 и 1.2 содержится команда условного перехода, таким образом поток управления делится на два дочерних потока.

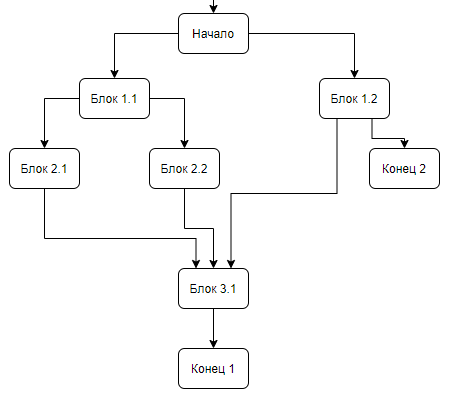


Рисунок 2.3. Пример графа потока управления

### Построение абстрактного синтаксического дерева

Четвертым этапом является построение абстрактного синтаксического дерева из построенного на предыдущем этапе графа потока управления. Абстрактное синтаксическое дерево – это промежуточное представление высокоуровневого кода в виде ориентированного дерева, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья — с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Пример абстрактного синтаксического дерева приведен на рис. 2.4.



Рисунок 2.4. Пример абстрактного синтаксического дерева

### Оптимизация

Пятый этап заключается в оптимизации полученного абстрактного синтаксического дерева, в частности сложных условных и константных выражений. Выражение – это набор численных операций, который преобразует входные числовые данные в некоторый результат, выраженный конкретным числовым значением. Оптимизация проводится с целью сделать сгенерированные выражения из P-кода более унифицированными и читаемыми для человека. Метрик читабельности выражения может быть перечислено несколько, но основная метрика – длина выражения: чем меньше длина выражения в символах, тем более читаемым оно является. Основные примеры оптимизации перечислены ниже:

1. Перемещение числовых операндов вправо: “5 + [rax]” → “[rax] + 5”.
2. Вычисление числовых операндов: “5 + 2” → “7”.
3. Раскрытие скобок: “([rax] + 5) \* 2” → “[rax] \* 2 + 10”.
4. Вычисление нулевых выражений: “[ax] & 0xFFFF0000” → “0”.

Стоит отметить, что вопреки метрике длины выражения оптимизация, заключающаяся в раскрытии скобок, часто увеличивает длину выражения. Однако, это необходимо для этапа типизации, на котором будет происходить процесс создания массивов, который требует полного раскрытия всех линейных выражений.

На этапе оптимизации также происходит оптимизация графа потока управления: несколько связанных блоков с простым условием перехода превращаются в один блок со сложным условием перехода. На рис. 2.5 представлен пример такой оптимизации.

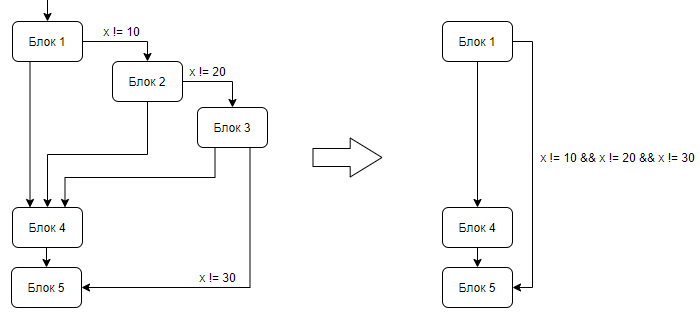


Рисунок 2.5. Пример абстрактного синтаксического дерева

Блоки 1, 2 и 3 имеют простое условие перехода. При этом данные блоки являются пустыми, они не должны содержать высокоуровневого кода. После оптимизации блоки 2 и 3 удаляются, а блок 1 приобретает новое условие, которое стало сложным.

### Символизация

Шестым этапом является процесс соотнесения узлам абстрактного синтаксического дерева символов. Символом в контексте теории компиляции является наименование и тип сущности, которой могут быть переменные или функции. Более подробно, сущностью могут являться:

1. Функции.
2. Параметры функции.
3. Локальные переменные (стековые или регистровые).
4. Глобальные переменные.

При разработке декомпилятора было принято следующее допущение: участок машинного кода будет декомпилирован полностью тогда и только тогда, когда будут определены все необходимые символы. Это означает, что для полной декомпиляции кода достаточно найти или вычислить символы для перечисленных выше сущностей в абстрактном синтаксическом дереве.

Если пользователем не были указаны символы, декомпилятор должен создать необходимые символы автоматически. Автоматически созданные символы будут иметь выбранные декомпилятором название и тип.

### Типизация

Седьмой этап заключается в итерационном прохождении по полученному на предыдущем этапе символизированному абстрактному синтаксическому дереву и вычислении для каждого узла этого дерева собственного типа. Правило вычисления типов узлов базируется на стандарте целевого высокоуровневого языка. Правила вычисления типов для языка C приведены в следующем параграфе.

### Генерация кода

Последний восьмой этап состоит в генерации высокоуровневого кода из типизированного абстрактного синтаксического дерева. Процесс генерации заключается в сопоставлении каждому узлу этого дерева некоторого текстового представления.

## Стандартное преобразование типов языка C

Этап типизации заключается в вычислении типов для каждого выражения. В данном разделе представлена информация по вычислению типов для выражений языка программирования C.

### Арифметические преобразования типов

Для арифметических выражений языка C существуют правила вычисления типов по умолчанию, которые приведены в табл. 2.1.

Таблица 2.1. Стандартные арифметические преобразования типов языка C

|  |  |
| --- | --- |
| **Выполненные условия** | **Преобразование** |
| Любой операнд имеет тип long double. | Другой операнд преобразуется в тип long double. |
| Предыдущее условие не выполнено, и любой из операндов имеет тип double. | Другой операнд преобразуется в тип double. |
| Предыдущие условия не выполнены, и любой из операндов имеет тип float. | Другой операнд преобразуется в тип float. |
| Предыдущие условия не выполнены (ни один из операндов не является операндом с плавающей запятой). | Операнды получают целочисленные продвижения следующим образом:  — Если любой из операндов имеет тип unsigned long, то другой операнд преобразуется в тип unsigned long.  — Если предыдущее условие не выполнено, и, если любой из операндов имеет тип long, а другой тип unsigned int, оба операнда преобразуются в тип unsigned long.  — Если предыдущие два условия не выполняются, и если любой из операндов имеет тип long, то другой операнд преобразуется в тип long.  — Если предыдущие три условия не выполняются, и если любой из операндов имеет тип unsigned int, то другой операнд преобразуется в тип unsigned int.  — Если ни одно из вышеперечисленных условий не выполняется, оба операнда преобразуются в тип int. |

В следующем коде, изображенном на рис. 2.5, демонстрируются правила преобразования, описанные в таблице.

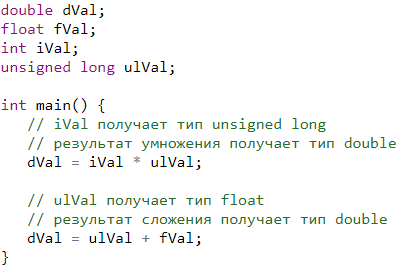


Рисунок 2.5. Код на языке C

Первая операция в примере представляет умножение двух целочисленных типов, iVal и ulVal. Условие выполнено, так как операнд не имеет типа float, а один операнд имеет тип unsigned int. Поэтому, другой операнд iVal преобразуется в тип unsigned int. Затем результат присваивается в переменную dVal. Один операнд имеет тип double, поэтому тип результата умножения становится double.

Второй оператор в предыдущем примере показывает добавление float и целочисленный тип: fVal и ulVal. Переменная ulVal преобразуется в тип float (третье условие в таблице). Результат сложения преобразуется в тип double (второе условие в таблице) и присваивается dVal.

### Преобразование указателей

Указатель на класс можно преобразовать в указатель на базовый класс в двух случаях:

1. Когда указанный базовый класс доступен и преобразование однозначно.
2. Указатель на класс можно преобразовать в указатель на базовый класс при использовании явного преобразования типов.

Указатели на тип void можно преобразовать в указатели на любой другой тип.

Указатель на функцию можно преобразовать в тип void\*, если тип void\* имеет достаточный размер для хранения указателя.

### Преобразование чисел со знаком в числа без знака

Объекты целочисленных типов со знаком можно преобразовывать в соответствующие типы без знака. При возникновении этих преобразований фактический битовый шаблон не изменяется. Однако изменяется интерпретация данных.

### Преобразование чисел без знака в числа со знаком

Объекты целочисленных типов без знака можно преобразовывать в соответствующие типы со знаком. Однако если значение без знака находится за пределами представимого диапазона типа со знаком, результат не будет иметь правильное значение.

### Преобразование чисел с плавающей запятой

Объект типа с плавающей запятой можно безопасно преобразовать в более точный тип с плавающей запятой, то есть без потери значимости. Например, преобразования из float в double double в long double являются безнадежными, а значение не изменяется.

### Преобразования между целочисленным типом и типом с плавающей запятой

Некоторые выражения могут вызывать преобразование объектов плавающего типа в целочисленные типы и наоборот. Если объект целочисленного типа преобразуется в тип с плавающей запятой, а исходное значение не может быть представлено точно, результатом является либо следующее выше, либо следующее меньшее представимое значение.

При преобразовании объекта с плавающего типа в целочисленный тип дробная часть усекается или округляется в сторону нуля. Например, число 1.4 преобразуется в число 1. Если усеченное значение выше наибольшего допустимого значения или меньше наименьшего представимого значения, результат будет неопределенным.

## Язык ассемблера архитектуры X86-64

В языке ассемблера архитектуры x86-64 команды можно разделить на следующие категории:

1. Команды общего назначения
2. Системные команды
3. Команды сопроцессора (x87 FPU)
4. Команды управления состоянием сопроцессора и SIMD
5. Команды технологии MMX
6. Команды расширения SSE
7. Команды расширения SSE2
8. Команды расширения SSE3
9. Команды расширения AVX

Наиболее часто используются команды общего назначения, команды технологии MMX, команды расширения SSE.

### Размеры операндов команд

В описаниях команд могут встречаться обозначения размеров операндов команд, которые представлены в табл. 2.2.

Таблица 2.2. Обозначения размеров операндов команд

|  |  |
| --- | --- |
| **Обозначение** | **Описание** |
| Byte | 8-битное целое (байт) |
| Word | 16-битное целое (слово) |
| DWord | 32-битное целое (двойное слово) |
| QWord | 64-битное целое (учетверенное слово) |
| Float | Вещественное число одинарной точности (32 бита) |
| Double | Вещественное число двойной точности (64 бита) |

### Команды общего назначения

Список команд общего назначения представлен в табл.

Таблица 2.3. Команды общего назначения

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| MOV | Присваивание |
| CMOVxx | Условное присваивание |
| XCHG | Обмен значений |
| BSWAP | Перестановка байтов |
| XADD | Обмен и сложение |
| CMPXCHG | Сравнение и обмен |
| CMPXCHG8B | Сравнение и обмен 8 байтов |
| PUSH | Поместить значение в стек |
| POP | Взять значение из стека |
| PUSHA/PUSHAD | Поместить значения регистров общего назначения в стек |
| POPA/POPAD | Взять значения регистров общего назначения из стека |
| IN | Прочитать значение из порта ввода/вывода |
| OUT | Записать значение в порт ввода/вывода |
| CWD | Преобразовать Word в DWord |
| CDQ | Преобразовать DWord в QWord |
| CBW | Преобразовать Byte в Word |
| CWDE | Преобразовать Word в DWord в регистре eax |
| MOVSX | Присвоить и расширить с учетом знака |
| MOVZX | Присвоить и расширить нулевым значением |
| ADD | Сложение |
| ADC | Сложение с переносом |
| SUB | Вычитание |
| SBB | Вычитание с заемом |
| IMUL | Знаковое умножение |
| MUL | Беззнаковое умножение |
| IDIV | Знаковое деление |
| DIV | Беззнаковое деление |
| INC | Инкремент |
| DEC | Декремент |
| NEG | Смена знака |
| CMP | Сравнение |
| DAA | Десятичная коррекция после сложения |
| DAC | Десятичная коррекция после вычитания |
| AAA | ASCII коррекция после сложения |
| AAS | ASCII коррекция после вычитания |
| AAM | ASCII коррекция после умножения |
| AAD | ASCII коррекция перед делением |
| AND | Побитовое логическое И |
| OR | Побитовое логическое ИЛИ |
| XOR | Побитовое логическое Исключающее ИЛИ |
| NOT | Побитовое логическое НЕ |
| SAR | Арифметический сдвиг вправо |
| SHR | Логический сдвиг вправо |
| SAL/SHL | Арифметический/логический сдвиг влево |
| SHRD | Двойной сдвиг вправо |
| SHLD | Двойной сдвиг влево |
| ROR | Вращение вправо |
| ROL | Вращение влево |
| RCR | Вращение вправо через флаг переноса |
| RCL | Вращение влево через флаг переноса |
| BT | Проверка бита |
| BTS | Проверка и установка бита |
| BTR | Проверка и сброс бита |
| BTC | Проверка и инверсия бита |
| BSF | Проверка бита в прямом направлении |
| BSR | Проверка бита в обратном направлении |
| SETxx | Установить значение байта в зависимости от флага |
| TEST | Логическое сравнение |
| JMP | Безусловный переход |
| Jxx | Условный переход |
| JCXZ/JECXZ | Переход, если cx/ecx равен 0 |
| LOOP | Цикл со счетчиком в ecx |
| LOOPZ/LOOPE | Цикл со счетчиком в ecx и выходом при нуле / равенстве |
| LOOPNZ/LOOPNE | Цикл со счетчиком в ecx и выходом при не нуле / неравенстве |
| CALL | Вызов подпрограммы |
| RET | Возврат из подпрограммы |
| IRET | Возврат из прерывания |
| INT | Вызов программного прерывания |
| INTO | Вызов прерывания по переполнению |
| BOUND | Переход при выходе значения за заданные рамки |
| ENTER | Высокоуровневый вход в процедуру |
| LEAVE | Высокоуровневый выход из процедуры |

### Команды блока MMX

Команды блока MMX представлены в табл. 2.4

Таблица 2.4. Команды блока MMX

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| EMMS | Подготовка сопроцессора к исполнению команд |
| MASKMOVQ | Запись байт в память из регистра MMX по маске (выборочная) |
| MOVD | Перемещение двойного слова |
| MOVNTQ | Запись 64 бит в память из регистра MMX (без использования кэш-памяти) |
| MOVQ | Переместить учетверенное слово (64 бит) |
| PACKSSWD | Упаковка со знаковым насыщением слов в байты |
| PACKSSDW | Упаковка со знаковым насыщением двойных слов в слова |
| PACKUSWB | Упаковка с беззнаковым насыщением слов в байты |
| PADDB | Сложение упакованных байт |
| PADDW | Сложение упакованных слов |
| PADDD | Сложение упакованных двойных слов |
| PADDSB | Сложение упакованных байт со знаковым насыщением |
| PADDSW | Сложение упакованных слов со знаковым насыщением |
| PADDUSB | Сложение упакованных байт с беззнаковым насыщением |
| PADDUSW | Сложение упакованных слов с беззнаковым насыщением |
| PAND | Упакованное логическое И |
| PANDN | Упакованное логическое И-НЕ |
| PAVGB | Упакованное среднее (байт) |
| PAVGW | Упакованное среднее (слов) |
| PCMPEQB | Сравнение на равенство упакованных байт |
| PCMPEQW | Сравнение на равенство упакованных слов |
| PCMPEQD | Сравнение на равенство упакованных двойных слов |
| PCMPGTB | Сравнение по условию "больше чем" упакованных байт |
| PCMPGTW | Сравнение по условию "больше чем" упакованных слов |
| PCMPGTD | Сравнение по условию "больше чем" упакованных двойных слов |
| PEXTRW | Извлечение 16-битного слова из регистра MMX по маске |
| PINSRW | Вставка 16-битного слова в регистр MMX |
| PMADDWD | Упакованное знаковое умножение слов операндов с последующим сложением промежуточных результатов в двойном слове |
| PMAXSW | Возврат максимальных упакованных знаковых слов |
| PMAXUB | Возврат максимальных упакованных беззнаковых байт |
| PMINSW | Возврат минимальных упакованных знаковых слов |
| PMINUB | Возврат минимальных упакованных беззнаковых байт |
| PMOVMSKB | Перемещение байтовой знаковой маски в целочисленный регистр |
| PMULHUW | Упакованное беззнаковое умножение слов с возвратом старшего слова результата |
| PMULHW | Упакованное знаковое умножение слов с возвратом старшего слова результата |
| PMULLW | Упакованное знаковое умножение слов с возвратом младшего слова результата |
| POR | Упакованное логическое ИЛИ |
| PSADBW | Суммарная разница значений пар беззнаковых упакованных байт |
| PSHUFW | Перераспределение упакованных слов |
| PSLLW | Логический сдвиг влево упакованных слов |
| PSLLD | Логический сдвиг влево упакованных двойных слов |
| PSLLQ | Логический сдвиг влево упакованных учетверенных слов |
| PSRAW | Арифметический сдвиг вправо упакованных слов |
| PSRAD | Арифметический сдвиг вправо упакованных двойных слов |
| PSRLW | Логический сдвиг вправо упакованных слов |
| PSRLD | Логический сдвиг вправо упакованных двойных слов |
| PSRLQ | Логический сдвиг вправо упакованных учетверенных слов |
| PSUBB | Вычитание упакованных байт |
| PSUBW | Вычитание упакованных слов |
| PSUBD | Вычитание упакованных двойных слов |
| PSUBSB | Вычитание упакованных байт со знаковым насыщением |
| PSUBSW | Вычитание упакованных слов со знаковым насыщением |
| PSUBUSB | Вычитание упакованных байт с беззнаковым насыщением |
| PSUBUSW | Вычитание упакованных слов с беззнаковым насыщением |
| PUNPCKHBW | Распаковка старших упакованных байт в слова |
| PUNPCKHWD | Распаковка старших упакованных слов в двойные слова |
| PUNPCKHDQ | Распаковка старших упакованных двойных слов в учетверенные слова |
| PUNPCKLBW | Распаковка младших упакованных байт в слова |
| PUNPCKLWD | Распаковка младших упакованных слов в двойные слова |
| PUNPCKLDQ | Распаковка младших упакованных двойных слов в учетверенные слова |
| PXOR | Упакованное логическое исключающее ИЛИ |

### Команды блока XMM

Команды блока XMM представлены в табл. 2.5

Таблица 2.5. Команды блока XMM

|  |  |  |
| --- | --- | --- |
| **Команда** | **Описание** | |
| ADDPS | Векторное сложение Float | |
| ADDSS | Скалярное сложение Float | |
| ANDNPS | Поразрядное логическое И-НЕ над Float | |
| ANDPS | Поразрядное логическое И над Float | |
| CMPPS | Векторное сравнение Float | |
| CMPSS | Скалярное сравнение Float | |
| COMISS | Скалярное упорядоченное сравнение с установкой EFLAGS | |
| CVTPI2PS | Преобразование двух упакованных DWord в два упакованных Float | |
| CVTPS2PI | Преобразование двух упакованных Float в два упакованных DWord | |
| CVTSI2SS | Скалярное преобразование знакового DWord в Float | |
| CVTSS2SI | Скалярное преобразование Float в знаковое DWord | |
| CVTTPS2PI | Преобразование усечением двух Float в два знаковых DWord | |
| CVTTSS2SI | Скалярное преобразование усечением Float в знаковое DWord | |
| DIVPS | Векторное деление Float | |
| DIVSS | Скалярное деление Float | |
| FXRSTOR | Восстановление без проверки наличия незамаскированных исключений с плавающей точкой состояния сопроцессора, целочисленного и потокового MMX-расширений из памяти | |
| FXSAVE | Сохранение состояния сопроцессора, целочисленного и потокового MMX-расширений в памяти | |
| LDMXCSR | Загрузка регистра состояния/управления MXCSR из памяти | |
| MAXPS | Вычисление максимальных упакованных значений Float | |
| MAXSS | Скалярное вычисление максимального значения Float | |
| MINPS | Вычисление минимальных упакованных значений Float | |
| MINSS | Скалярное вычисление минимального значения Float | |
| MOVAPS | Перемещение выровненных 128 бит | |
| MOVHLPS | Перемещение верхних упакованных значений Float в нижние | |
| MOVHPS | Перемещение верхних упакованных значений Float | |
| MOVLHPS | Перемещение нижних упакованных значений Float в верхние | |
| MOVLPS | Перемещение невыровненных нижних упакованных значений Float | |
| MOVMSKPS | Перемещение знаковой маски в целочисленный регистр | |
| MOVNTPS | Запись в память 128 бит из регистра XMM, минуя кэш | |
| MOVSS | Скалярное перемещение Float | |
| MOVUPS | Перемещение невыровненных упакованных Float | |
| MULPS | Векторное умножение Float | |
| MULSS | Скалярное умножение Float | |
| ORPS | Поразрядное логическое ИЛИ над Float | |
| RCPPS | Векторное вычисление обратных значений Float | |
| RCPSS | Скалярное вычисление обратного значения Float | |
| RSQRTPS | Векторное вычисление обратного значения квадратного корня Float | |
| RSQRTSS | Скалярное вычисление обратного значения квадратного корня Float | |
| SHUFPS | Перераспределение упакованных значений Float | |
| SQRTPS | Векторное вычисление квадратного корня Float | |
| SQRTSS | Скалярное вычисление квадратного корня Float | |
| STMXCSR | Сохранение регистра управления/состояния MXCSR в памяти | |
| SUBPS | Векторное вычитание Float | |
| SUBSS | Скалярное вычитание Float | |
| UCOMISS | Неупорядоченное скалярное сравнение с установкой флагов в EFLAGS | |
| UNPCKHPS | Перенос верхних упакованных значений Float с чередованием | |
| UNPCKLPS | Перенос нижних упакованных значений Float с чередованием | |
| XORPS | Поразрядное логическое исключающее ИЛИ над Float | |
| ADDPD | | Векторное сложение Double | |
| ADDSD | | Скалярное сложение Double | |
| ANDPD | | Поразрядное логическое И над Double | |
| ANDNPD | | Поразрядное логическое И-НЕ над Double | |
| CLFLUSH | | Сброс в память строки кэша, содержащей адрес | |
| CMPPD | | Векторное сравнение Double | |
| CMPSD | | Скалярное сравнение Double | |
| COMISD | | Скалярное сравнение упорядоченных значений и установка EFLAGS | |
| CVTDQ2PD | | Преобразование двух упакованных DWord в два Double | |
| CVTDQ2PS | | Преобразование четырех упакованных знаковых DWord в четыре упакованных Float | |
| CVTPD2DQ | | Преобразование двух упакованных Double в два упакованных DWord | |
| CVTPD2PI | | Преобразование двух упакованных Double в два упакованных DWord | |
| CVTPD2PS | | Преобразование двух упакованных Float в два упакованных Double | |
| CVTSD2SI | | Преобразование скалярного Double в DWord | |
| CVTSD2SS | | Преобразование скалярного Double в Float | |
| CVTSI2SD | | Преобразование скалярного DWord в Double | |
| CVTSS2SD | | Преобразование скалярного Float в Double | |
| CVTTPD2PI | | Преобразование усечением двух упакованных Double в два DWord | |
| CVTTPD2DQ | | Преобразование усечением двух упакованных Double в два DWord | |
| CVTTPS2DQ | | Преобразование усечением четырех упакованных Float в четыре знаковых DWord | |
| CVTTSD2SI | | Преобразование усечением скалярного Double в Float | |
| DIVPD | | Векторное деление Double | |
| DIVSD | | Скалярное деление Double | |
| LFENCE | | Упорядочить операции загрузки | |
| MASKMOVDQU | | Выборочная запись байт из источника в память | |
| MAXPD | | Вычисление максимальных упакованных значений Double | |
| MAXSD | | Скалярное вычисление максимального значения Double | |
| MFENCE | | Упорядочить операции загрузки и сохранения | |
| MINPD | | Вычисление минимальных упакованных значений Double | |
| MINSP | | Скалярное вычисление минимального значения Double | |
| MOVAPD | | Перемещение упакованных выровненных Double | |
| MOVD | | Перемещение DWord | |
| MOVDQA | | Перемещение выровненных 128 бит | |
| MOVDQU | | Перемещение невыровненных 128 бит | |
| MOVDQ2Q | | Перемещение младшего QWord | |
| MOVHPD | | Перемещение старшего упакованного Double | |
| MOVLPD | | Перемещение младшего упакованного Double | |
| MOVMSKPD | | Извлечение 2-битной знаковой маски упакованных Double | |
| MOVNTDQ | | Сохранение двух QWord в память без использования кэша | |
| MOVNTI | | Сохранение DWord из регистра общего назначения в память без использования кэша | |
| MOVNTPD | | Сохранение двух Double в память без использования кэша | |
| MOVQ | | Перемещение QWord | |
| MOVQ2DQ | | Перемещение QWord из MMX-регистра в младшее QWord XMM-регистра | |
| MOVSD | | Перемещение скалярного Double | |
| MOVUPD | | Перемещение невыровненных упакованных Double | |
| MULPD | | Векторное умножение Double | |
| MULSD | | Скалярное умножение Double | |
| ORPD | | Поразрядное логическое ИЛИ над Double | |
| PACKSSWB | | Упаковка со знаковым насыщением слов в байты | |
| PACKSSDW | | Упаковка со знаковым насыщением двойных слов в слова | |
| PACKUSWB | | Упаковка с беззнаковым насыщением слов в байты | |
| PADDB | | Сложение упакованных байт | |
| PADDW | | Сложение упакованных слов (Word) | |
| PADDD | | Сложение упакованных двойных слов (DWord) | |
| PADDQ | | Сложение учетверенных слов (QWord) | |
| PADDSB | | Сложение упакованных байт со знаковым насыщением | |
| PADDSW | | Сложение упакованных слов со знаковым насыщением | |
| PADDUSB | | Сложение упакованных байт с беззнаковым насыщением | |
| PADDUSW | | Сложение упакованных слов с беззнаковым насыщением | |
| PAND | | Упакованное логическое И | |
| PANDN | | Упакованное логическое И-НЕ | |
| PAUSE | | Улучшить выполнение цикла ожидания-занятости | |
| PAVGB | | Упакованное среднее байт | |
| PAVGW | | Упакованное среднее слов | |
| PCMPEQB | | Сравнение на равенство упакованных байт | |
| PCMPEQW | | Сравнение на равенство упакованных слов | |
| PCMPEQD | | Сравнение на равенство упакованных двойных слов | |
| PCMPGTB | | Сравнение по условию "больше чем" упакованных байт | |
| PCMPGTW | | Сравнение по условию "больше чем" упакованных слов | |
| PCMPGTD | | Сравнение по условию "больше чем" упакованных двойных слов | |
| PEXTRW | | Извлечение 16-битного слова из XMM-регистра | |
| PINSRW | | Вставка 16-битного слова в регистр XMM | |
| PMADDWD | | Упакованное знаковое умножение слов с последующим сложением промежуточных результатов в DWord | |
| PMAXSW | | Возврат максимальных упакованных знаковых слов | |
| PMAXUB | | Возврат максимальных упакованных беззнаковых байт | |
| PMINSW | | Возврат минмальных упакованных знаковых слов | |
| PMINUB | | Возврат минимальных упакованных беззнаковых байт | |
| PMOVMSKB | | Перемещение байтовой знаковой маски в целочисленный регистр | |
| PMULHUW | | Упакованное беззнаковое умножение слов с возвратом старших слов результата | |
| PMULHW | | Упакованное знаковое умножение слов с возвратом старших слов результата | |
| PMULLW | | Упакованное знаковое умножение слов с возвратом младших слов результата | |
| PMULUDQ | | Умножение DWord и сохранение результата в XMM-регистре. | |
| POR | | Упакованное логическое ИЛИ | |
| PSADBW | | Суммарная разница значений пар беззнаковых упакованных байт | |
| PSHUFD | | Перераспределение упакованных двойных слов | |
| PSHUFHW | | Перераспределение упакованных слов в старших DWord | |
| PSHUFLW | | Перераспределение упакованных слов в младших DWord | |
| PSLLDQ | | Побайтный сдвиг влево | |
| PSLLW | | Логический сдвиг слов влево | |
| PSLLD | | Логический сдвиг двойных слов влево | |
| PSLLQ | | Логический сдвиг учетверенных слов влево | |
| PSRAW | | Арифметический сдвиг слов влево | |
| PSRAD | | Арифметический сдвиг двойных слов влево | |
| PSRLDQ | | Побайтный сдвиг вправо | |
| PSRLW | | Логический сдвиг слов вправо | |
| PSRLD | | Логический сдвиг двойных слов вправо | |
| PSRLQ | | Логический сдвиг учетверенных слов вправо | |
| PSUBB | | Вычитание упакованных байт | |
| PSUBW | | Вычитание упакованных слов | |
| PSUBD | | Вычитание упакованных двойных слов | |
| PSUBQ | | Вычитание учетверенных слов | |
| PSUBSB | | Вычитание упакованных слов со знаковым насыщением | |
| PSUBSW | | Вычитание упакованных двойных слов со знаковым насыщением | |
| PSUBUSB | | Вычитание упакованных байт с беззнаковым насыщением | |
| PSUBUSW | | Вычитание упакованных слов с беззнаковым насыщением | |
| PUNPCKHBW | | Распаковка старших упакованных байт в слова | |
| PUNPCKHWD | | Распаковка старших упакованных слов в двойные слова | |
| PUNPCKHDQ | | Распаковка старших упакованных двойных слов в учетверенные слова | |
| PUNPCKHQDQ | | Распаковка старших упакованных учетверенных слов в двойные учетверенные слова | |
| PUNPCKLBW | | Распаковка младших упакованных байт в слова | |
| PUNPCKLWD | | Распаковка младших упакованных слов в двойные слова | |
| PUNPCKLDQ | | Распаковка младших упакованных двойных слов в учетверенные слова | |
| PUNPCKLQDQ | | Распаковка младших упакованных учетверенных слов в двойные учетверенные слова | |
| PXOR | | Упакованное логическое исключающее ИЛИ | |
| SHUFPD | | Перестановка упакованных значений Double | |
| SQRTPD | | Векторное вычисление квадратного корня Double | |
| SQRTSD | | Скалярное вычисление квадратного корня Double | |
| SUBPD | | Векторное вычитание Double | |
| SUBSD | | Скалярное вычитание Double | |
| UCOMISD | | Неупорядоченное скалярное сравнение Double с установкой флагов в EFLAGS | |
| UNPCKHPD | | Перенос верхних упакованных значений Double с чередованием | |
| UNPCKLPD | | Перенос нижних упакованных значений Double с чередованием | |
| XORDP | | Поразрядное логическое исключающее ИЛИ над Double | |

## Маски выражений

Каждое выражение при указании конкретных числовых входных данных выдает определенное числовое значение. Существует диапазон всех возможных значений, которые может выдать определенное выражение. Этот диапазон может быть выражен в виде битовой маски. Например, диапазон возможных значений регистра “al” в архитектуре процессора X86-64 составляет от 0 до 255, таким образом маска будет числом 0xFF. При этом выражение “al & 0xF” будет иметь диапазон значений от 0 до 16, поскольку старшие четыре бита регистра “al” будут иметь нулевое значение из-за битового умножения на число 0xF, и тогда итоговая маска данного выражения будет 0xF.

Маски выражений активно используются на этапе оптимизации выражений. Ниже приведены примеры, когда маски помогают значительно сократить длину выражения:

1. Побитовая дизъюнкция: “[al] | 0xFF” → “0xFF”
2. Побитовая конъюнкция: “[al] & 0xFF” → “[al]”

## Адресация элементов многомерных массивов

Массивы, в том числе многомерные, располагаются в памяти в виде непрерывной последовательности байт. У каждого массива есть базовый адрес, по которому располагается первый элемент массива.

Адресация любого элемента одномерного массива производится по формуле (2.1).

|  |  |
| --- | --- |
| , | (2.1) |

где A – массив, i – индекс элемента, B – базовый адрес массива, S – размер элемента массива.

Адресация любого элемента n-мерного массива производится по формуле (2.2).

|  |  |
| --- | --- |
| , | (2.2) |

где A – массив,  – индекс элемента в k-ом измерении, B – базовый адрес массива, – размер элемента массива в k-ом измерении,  - размер типа элемента массива.

Для n-мерного массива справедливо следующее соотношение (2.3).

|  |  |
| --- | --- |
|  | (2.3) |

## Линеаризация графа потока управления

Одной из основных задач декомпилятора является линеаризация графа потока управления, то есть представление графа потока управления в виде последовательности вложенных друг в друга условных выражений и циклов. Главная метрика качества алгоритма линеаризации – это количество операторов goto, которое должно быть сведено к минимуму. Другая немаловажная метрика – глубина вложенности условных выражений и циклов, и эта глубина также должна быть минимальна. Учёт двух перечисленных метрик позволит сделать структуру генерируемого кода наиболее читаемой для человека.

В следующих параграфах представлены соответствия между блоками графа потока управления и кодом на языке C.

### Условное выражение if

Условное выражение без ветки else представляется в виде трех блоков, связанных между собой так, как это показано на рис. 2.6.

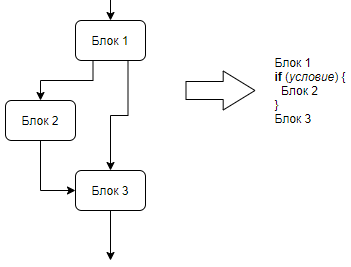


Рисунок 2.6. Конструкция if без ветки else

### Условное выражение if-else

Условное выражение с веткой else представляется в виде четырёх блоков, связанных между собой так, как это показано на рис. 2.7.

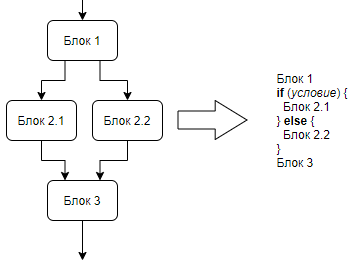


Рисунок 2.7. Конструкция if c веткой else

### Цикл while

Цикл while представляется в виде трех блоков, связанных между собой так, как это показано на рис. 2.8.

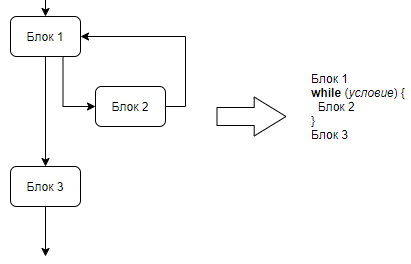


Рисунок 2.8. Конструкция цикла while

# Проектирование декомпилятора

Проектирование любой информационной системы является важной составляющей успешной разработки программного обеспечения. Этап проектирования помогает заложить правильный фундамент программы. По утверждению Стива Макконнелла в своей книге «Совершенный код» реализация этого этапа помогает избежать примерно 90% ошибок.

Было принято использовать объектный подход, поэтому будет использован язык моделирования UML.

UML – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем.

## Этап анализа

На этапе анализа необходимо выделить функциональные требования и роли для информационной системы «Декомпилятор», а также построить соответствующие диаграмму прецедентов и диаграмму активностей.

### Диаграмма прецедентов

Взаимодействовать с ИС будет пользователь. Пользователь предоставляет ИС участок машинного кода или код, написанный на языке ассемблер архитектуры X86-64, тем самым инициирует запуск конвейера декомпилятора. Конвейер декомпилятора состоит из нескольких обязательных этапов, которые были описаны в второй главе. Диаграмма прецендентов представлена на рис. 3.1.

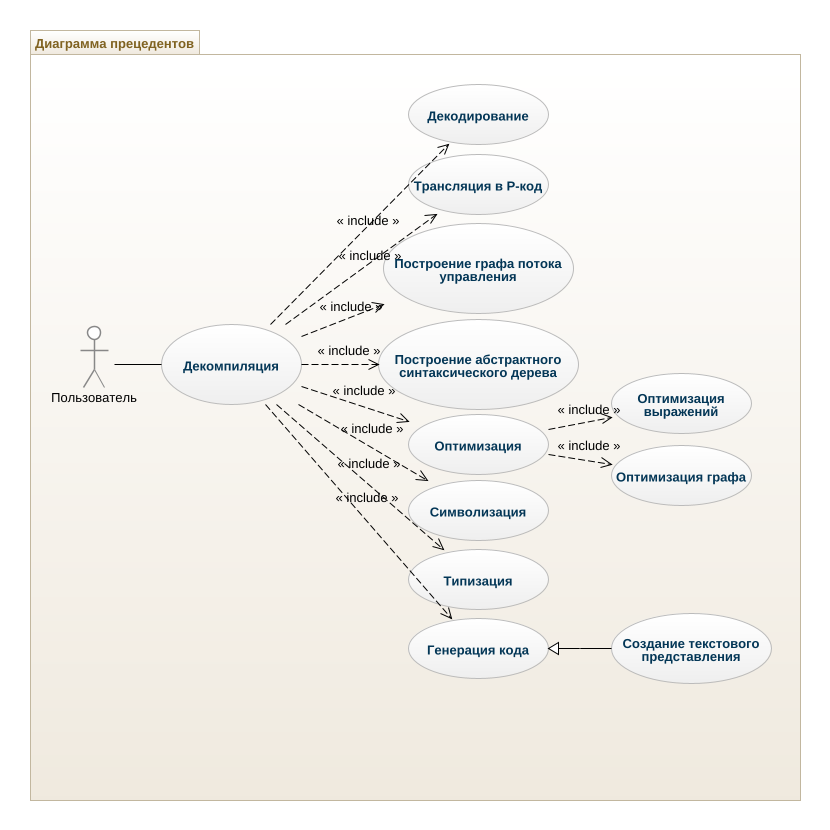


Рисунок 3.1. Диаграмма прецедентов

### Диаграмма активностей

Для понимания динамики процесса декомпиляции была использована диаграмма активностей, которая продемонстрирована на рис. 3.2. Необходимость в использовании цикла на диаграмме заключается в том, что построение абстрактного синтаксического дерева с последующими этапами его преобразования должно происходить многократно до тех пор, пока не будут распознаны сигнатуры для всех функций в виртуальных вызовах. Избежание многократности выполнения данных этапов процесса декомпиляции невозможно по причине того, что заранее невозможно идентифицировать сигнатуры функций в виртуальных вызовах, поскольку в данном случае явно не указывается адрес функции, которую необходимо вызвать.

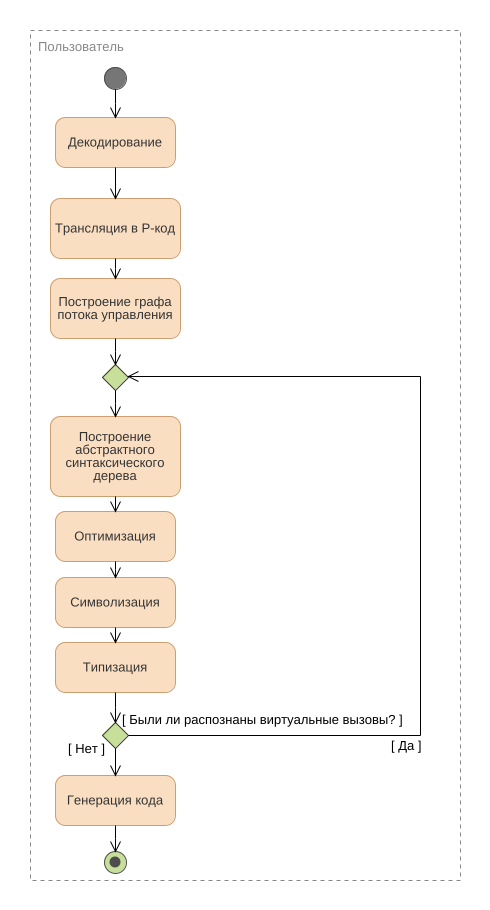


Рисунок 3.2. Диаграмма активностей

## Этап проектирования

Этап проектирования будет заключаться в создании диаграммы классов. Построенная иерархия классов должна соответствовать архитектуре конвейера декомпилятора, о котором было сказано во второй главе. Языком программирования, на котором будет написан декомпилятор, был выбран язык C++. Такой выбор обусловлен высокой скоростью и широкими возможностями данного языка.

### Создание классов

В ходе проектирования были выделены следующие классы, каждый из которых имеет своё предназначение в ИС:

1. Decompiler. Главный класс, содержащий методы, которые позволяют инициировать запуск процесса декомпиляции, а также настроить основные параметры.
2. FunctionPCodeGraph. Класс, являющиеся графом потока управления. Содержит объекты класса PCodeBlock.
3. PCodeBlock. Класс, являющиеся блоком графа потока управления, который содержит листинг инструкций P-кода. Содержит две исходящие ссылки на другие объекты класса PCodeBlock, а также список объектов класса PCodeBlock, которые ссылаются на данный блок.
4. DecoderX86. Класс, инкапсулирующий логику по декодированию инструкций машинного кода архитектуры X86-64. Данный класс распознает тип и свойства машинной инструкции, далее начинает процесс построения списка соответствующих инструкций P-кода.
5. RegisterFactoryX86. Класс, являющиеся фабрикой регистров под архитектуру X86-64.
6. BitMask64. Класс, являющиеся 64-битной маской выражения. Описывает диапазон возможных значений. Содержит операции дизъюнкции, конъюнкции, сравнения.
7. ImageAnalyzer. Класс, содержащий алгоритм построения графа потока управления по списку инструкций P-кода. Имеет возможность работать в режиме потока байт машинного кода, то есть строить граф потока управления по ходу получения машинного кода, транслируя его одновременно с использованием объекта класса DecoderX86 в P-код.
8. InstructionInterpreter. Класс, являющиеся интерпретатором P-кода. Принимая поток инструкций P-кода объект данного класса строит абстрактное синтаксическое дерево.
9. DecompiledCodeGraph. Класс, который представлен в виде графа потока управления, но который содержит выражения абстрактного синтаксического дерева.
10. GraphBlockLinker. Класс, позволяющий сделать символьное связывание блоков P-кода графа потока управления, а именно разрешить все внешние символов каждого блока, которые создаются в момент интерпретации P-кода по запросу определенного регистра.
11. ExprOptimization. Класс, позволяющий произвести оптимизацию заданного выражения по правилам, которые кратко описаны во второй главе.
12. GraphOptimization. Класс, позволяющий произвести оптимизацию структуры графа потока управления. В частности, эта оптимизация заключается в удалении и замене блоков с простым условием на блоки со сложным условием, тем самым обеспечивая минимальную вложенность условных конструкций.
13. SdaBuilding. Класс, позволяющий произвести символизацию абстрактного синтаксического дерева.
14. SdaDataTypesCalculating. Класс, позволяющий произвести типизацию абстрактного синтаксического дерева.
15. SdaGraphMemoryOptimization. Класс, позволяющий произвести оптимизацию выражений, связанных с памятью, за счет удаления лишних присваиваний и оптимизации метрики длины выражений, что делает код более читаемым.
16. LinearViewSimpleOutput. Класс, позволяющий произвести генерацию итогового высокоуровневого кода на языке C.

Также были выделены вспомогательные классы, описание которых здесь не приводится.

### Диаграмма классов

Подробное содержание каждого выделенного класса представлено на рис. 3.3 и рис. 3.4.

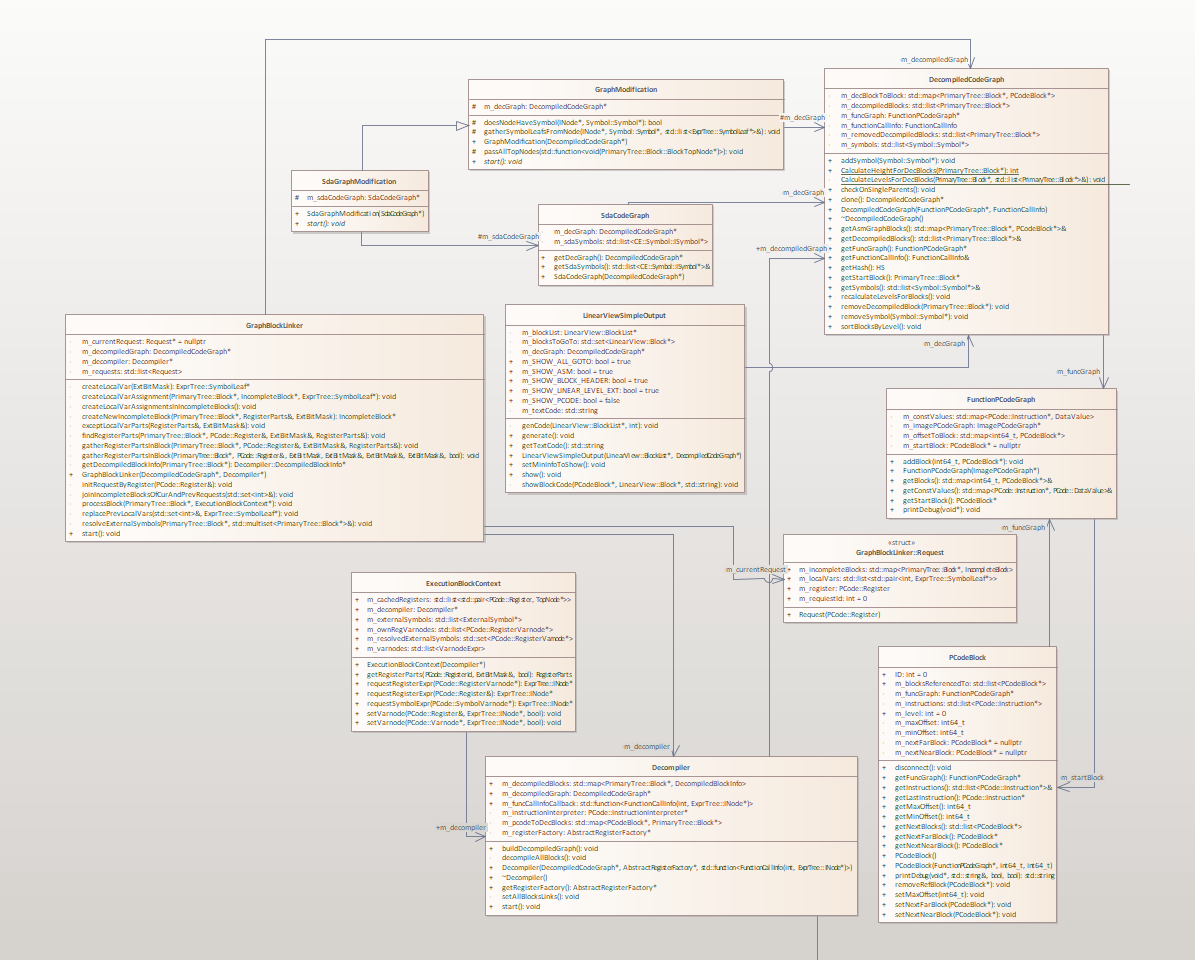


Рисунок 3.3. Диаграмма классов (первая часть)

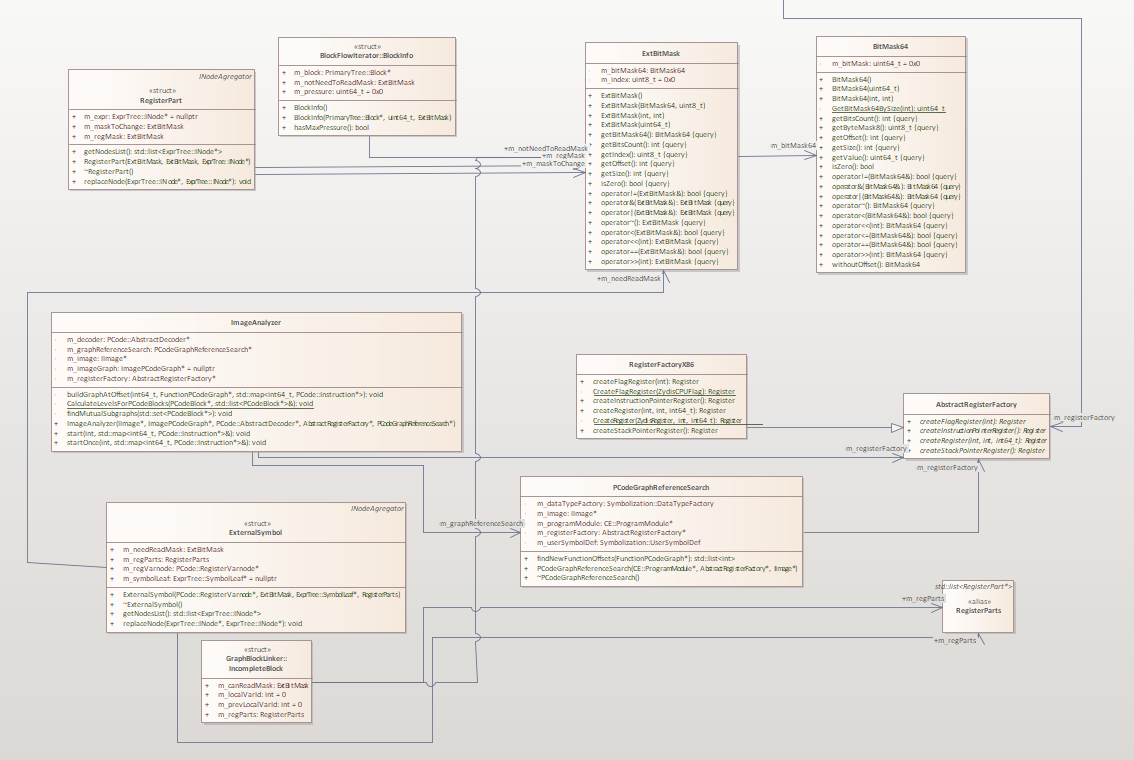


Рисунок 3.4. Диаграмма классов (вторая часть)

## Этап реализации

На этапе реализации мы должны реализовать ИС и понять, как она будет выглядеть в виде цельной системы.

### Использование сторонних библиотек

Для ускорения процесса разработки было принято решение использовать сторонние библиотеки:

1. SDA. Данная библиотека нужна для удобного манипулирования типами языка C.
2. ZydisDisassembler. Данная библиотека нужна для реализации процесса декодирования машинного кода архитектуры X86-64 в структуры описаний ассемблера.
3. Asmjit/Asmtk. Данный набор библиотек необходим для трансляции текстового кода на языке ассемблер архитектуры X86-64 в машинный код.

### Диаграмма компонентов

Диаграмма компонентов, представленная на рис. 3.5, позволяет графически изобразить структуру будущего приложения.

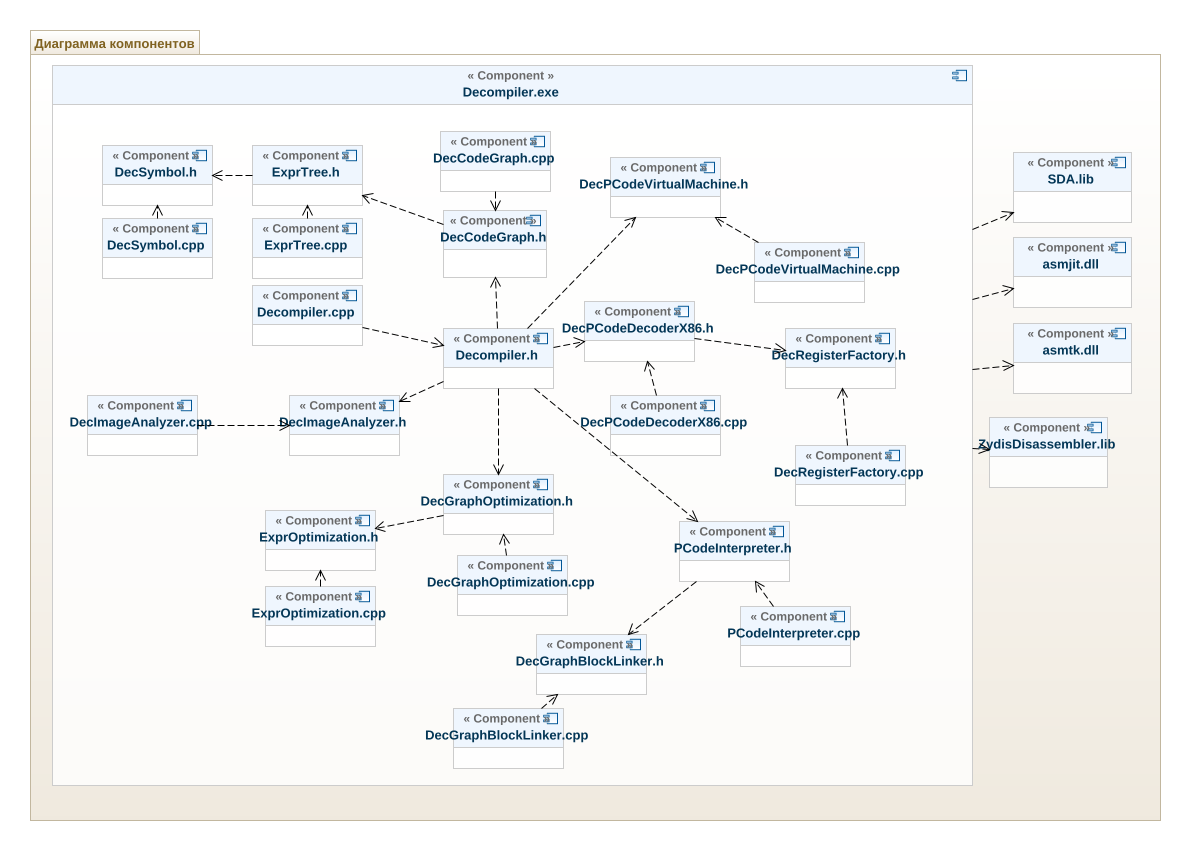


Рисунок 3.5. Диаграмма компонентов

## Реализация основных алгоритмов

В данном параграфе перечислены основные алгоритмы, которые были использованы для процесса декомпиляции.

### Алгоритм построения графа потока управления

Был самостоятельно разработан алгоритм, позволяющий на основе списка инструкций P-кода построить граф блоков, связанных между собой командами условного и безусловного перехода.

Краткое описание алгоритма:

1. Указывается текущее начальное смещение относительно базового адреса в памяти машинного кода, который необходимо декомпилировать. Также создается начальный блок P-кода, в который будут добавляться инструкции.
2. Декодируется инструкция, которая располагается по текущему смещению. Результат декодирования, который представлен в виде списка примитивных инструкций P-кода, сохраняется в карте вида “смещение – инструкция”. Если по текущему смещению невозможно декодировать инструкцию, то текущее смещение приобретает новое значение из непустого списка смещений, которые необходимо посетить. После этого, осуществляется переход на шаг 1. Если этот список оказывается пустым, алгоритм прекращает свою работу.
3. В множестве посещенных смещений сохраняется текущее смещение.
4. Осуществляет итерация по списку инструкций P-кода, полученному на шаге 2, при этом каждая инструкция добавляется в блок P-кода, который находится по текущему значению. Если встретилась инструкция безусловного перехода, то текущее смещение изменяется на то, что находится в первом операнде инструкции данного перехода. Если встретилась инструкция условного перехода, то смещение, которое находится в первом операнде инструкции, добавляется в список смещений, которые надо посетить позже.
5. По текущему смещению создается новый блок P-кода.
6. Переход на шаг 1.

Данный алгоритм реализован в классе ImageAnalyzer.

### Итерационный алгоритм вычисления типов выражений

После построения символизированного абстрактного синтаксического дерева (далее АСД), декомпилятор должен вычислить для каждого узла этого дерева тип данных языка программирования C. Для этого был самостоятельно разработан алгоритм вычисления типов.

Краткое описание алгоритма:

1. Осуществляется проход по всем узлам АСД снизу вверх.
2. Если встретилась бинарная операция, то по правилам преобразования типов данных языка C, которые описаны во второй главе, назначается определенный тип для данной операции, а также происходит явное или неявное приведение к этому типу операндов этой операции. Если среди этих операндов оказался символ, созданный автоматически, то изменяется его тип данных без привидения типа.
3. Если были сделаны новые приведения типов, то осуществляется переход на шаг 1, иначе алгоритм завершает свою работу.

Данный алгоритм реализован в классе SdaDataTypesCalculating.

Пример работы данного алгоритма изображен на рис. 3.6. В данном примере показывается вычисление типа float для нескольких переменных. После этого меняется вид кода, в частности отображение вещественных чисел.

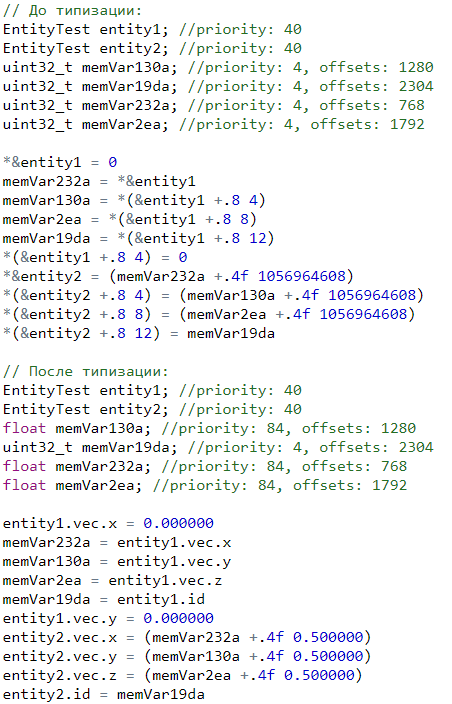


Рисунок 3.6. Пример работы алгоритма вычисления типов

### Алгоритм оптимизации выражений, связанных с памятью

Это основной самостоятельно разработанный алгоритм, позволяющий сделать код значительно более читаемым за счет удаления лишних присваиваний, связанных с пересылкой данных из одного участка памяти другой.

Краткое описание алгоритма:

1. Для каждого блока кода абстрактного синтаксического дерева осуществляется итерация по всем выражениям.
2. Если встретилось выражение присваивания, то определяется, происходит чтение из памяти или запись в память значения. Если чтение из памяти в целевую переменную, то создается снимок значения и проекция этого снимка на указанную в присваивании область памяти, а также происходит связывание этой области памяти с целевой переменной. Если запись в память, то создается в этой области памяти новое значение.
3. Если встретилось выражение, в котором происходит чтение из памяти, то просматривается область памяти, в которую были записаны снимки значений, и происходит замена выражения чтения памяти на этот снимок.

Данный алгоритм реализован в классе SdaGraphMemoryOptimization.

Пример работы данного алгоритма представлен на рис. 3.7. В результате оптимизации были удалены три операции присваивания, код стал меньше в размере и приобрел более читаемый вид.

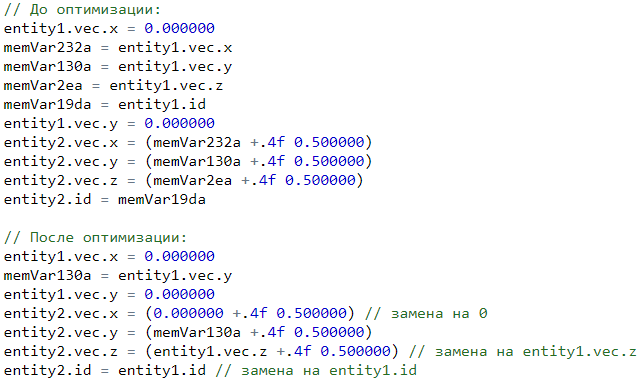


Рисунок 3.7. Пример оптимизации выражений памяти

# Тестирование и сравнение

Тестирование является очень важным этапом в разработке программного обеспечения, поскольку позволяет выявить большинство ошибок еще до того, как разработанный продукт появится в доступе у конечных пользователей.

## Тестрование декомпилятора

Тестирование разработанного декомпилятора было произведено вручную на 20 разных примерах машинного кода. Список данных примеров можно найти в файле DecSampleTests.cpp.

## Сравнение с Ghidra Decompiler

Сравнение будет произведено с декомпилятором Ghidra Decompiler в силу его большой популярности, а также потому, что он выдаёт в среднем более качественный код, чем остальные декомпиляторы, которые были рассмотрены в первой главе.

### Векторные и математические операции

Разработанный декомпилятор в отличие от Ghidra Decompiler успешно справляется с задачей декомпиляции сложных математических векторных выражений.

На рис. 4.1 показан фрагмент результата декомпиляции функции умножения двух матриц определенного вида. Разработанный декомпилятор не использует в своем выводе ничего лишнего, в то время как Ghidra Decompiler использует предопределенный макрос SUB164, операции сдвига, лишние локальные переменные, что делает код крайне нечитаемым.



Рисунок 4.1. Фрагмент результата декомпиляции умножения матриц

На рис. 4.2 приведен ещё один пример декомпиляции математической функции. Разработанный декомпилятор снова не использует в своем выводе ничего лишнего, в то время как Ghidra Decompiler использует множество предопределенных макросов и неизвестных XMM регистров.

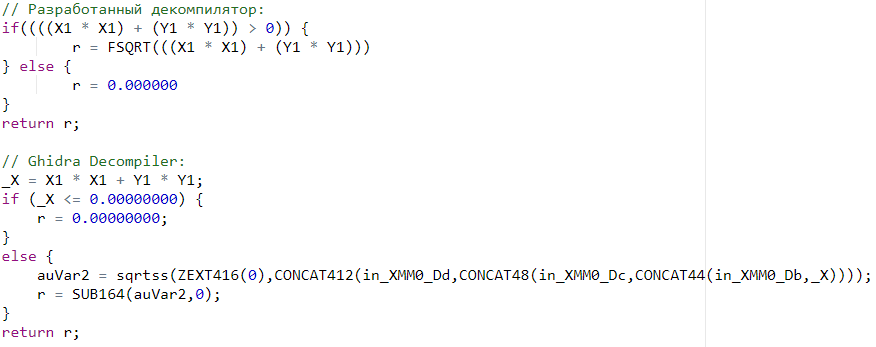


Рисунок 4.2. Фрагмент результата декомпиляции математической функции

### Интенсивная работа со стеком

Разработанный декомпилятор в отличие от Ghidra Decompiler также хорошо справляется с машинным кодом, где происходит интенсивная работа со стеком через векторные SSE инструкции.

На рис. 4.3 изображено сравнение результатов декомпиляции разработанного декомпилятора и Ghidra Decompiler для участка машинного кода, где происходят сложные векторные операции с памятью стека. В данном случае Ghidra Decompiler сделал неполный вывод всего кода, смешав его к тому же с малопонятными макросами и операциями побитового сдвига.

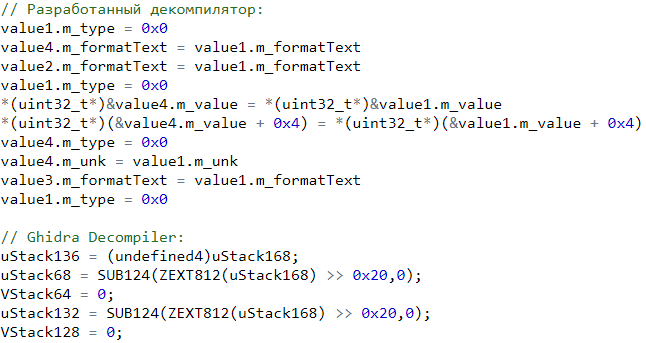


Рисунок 4.3. Фрагмент результата декомпиляции

### Сравнение скорости

На тридцати примерах машинного кода было произведено тестирование скорости декомпиляции разработанного декомпилятора и Ghidra Decompiler. Результаты представлены в табл. 4.1.

Таблица 4.1. Сравнение скорости декомпиляции

|  |  |  |
| --- | --- | --- |
| **Время декомпиляции** | **Разработанный декомпилятор** | **Ghidra Decompiler** |
| Среднее время | 5 мс | 40 мс |
| Минимальное время | 1 мс | 10 мс |
| Максимальное время | 800 мс | 5 сек |

Как итог, разработанный декомпилятор справляется с задачей значительно быстрее, чем Ghidra Decompiler.

Заключение

В ходе выполненной работы было проделано несколько важных этапов.

На первом этапе исследования были внимательно изучены и проанализированы существующие программные реализации популярных декомпиляторов: IDA Hex-Rays Decompiler, Ghidra Decompiler, RetDec, Snowman. Были выявлены сильные и слабые стороны каждого декомпилятора. В ходе анализа был сделан вывод, что ни один декомпилятор не способен генерировать понятный человеку высокоуровневый код на множестве тех примеров машинного кода, где в значительной степени преобладают векторные и математические операции, где происходит интенсивная работа с памятью компьютера. Также было выяснено, что не каждый декомпилятор предоставляет возможность интерактивной декомпиляции кода. Итогом этапа анализа существующих декомпиляторов стало принятие решение о разработке собственной программной реализации с нуля.

На втором этапе было проведено тщательное изучение теории о низкоуровневом программировании, архитектуре памяти компьютера, операционных системах, компиляторах. В силу недостатка теории непосредственно о процессе декомпиляции были изучены исходные коды существующих декомпиляторов, а также изобретены собственные методы и алгоритмы.

На третьем этапе после изучения теоретических основ процесса декомпиляции был выбран объектный подход к проектированию и реализации собственного декомпилятора, в качестве языка моделирования ИС был выбран UML. Был проведен этап анализа, проектирования и реализации. В ходе данных этапов были получены диаграмма прецедентов, диаграмма активностей, диаграммы классов и компонентов. После проектирования и создания диаграмм был реализован декомпилятор на высокопроизводительном языке C++.

На заключительном этапе было осуществлено тестирование разработанного декомпилятора на двадцати разных примерах. Было проведено сравнение данного декомпилятора с декомпилятором Ghidra Decompiler по критериям качества генерируемого кода и скорости декомпиляции.

Результатом данного исследования стало то, что разработанный декомпилятор успешно справляется с большинством примеров машинного кода архитектуры X86-64. В частности, успешно декомпилируются те примеры, где наличествуют векторные и математические операции и где производится интенсивная работа с памятью. Несмотря на успех декомпиляции в одних примерах, есть безусловные провалы на других примерах, где результат декомпиляции качественно значительно уступает такому декомпилятору, как Ghidra Decompiler.

Дальнейшие планы состоят в улучшении показателей качества генерируемого кода, увеличении числа примеров, где это качество станет приемлемым. В частности, планируется направить вектор развития в сторону внедрения в разработанный декомпилятор методов искусственного интеллекта, на основе которых станет возможно восстановление не только структуры кода, но и имен переменных и функций. Это может быть достигнуто за счет включения онтологий предметной области, в терминах которой и была разработана та или иная программа, которую необходимо декомпилировать. Также планируется реализация отладчика, который позволит интерактивно выполнять программу и смотреть за изменением её состояния по ходу отладки. Стоит отметить, что на данный момент полноценный отладчик не реализован ни в одном бесплатном декомпиляторе.

Библиографический список

1. *Брыксин Т.А*. Платформа для создания специализированных визуальных сред разработки программного обеспечения : дис. … канд. техн. наук: 05.13.11 / *Брыксин Тимофей Александрович*. СПб., 2016. – 159 с.
2. Исходный код декомпилятора

Исходный код разработанного декомпилятора располагается по ссылке: https://github.com/Fleynaro/MultiPlayer/tree/master/SDA/Module/Decompiler