Конспект лекций

По средствам разработки

Автор: Владимиров К. И.

Страница с последней редакцией:
sourceforge.net/p/cpp-lects-rus
Email автора: konstantin.vladimirov@gmail.com
30 марта 2016 г. LaTeX

Содержание

1 Toolchain

Общеупотребительным англицизмом toolchain (можно перевести как "набор средств разработки" или как "система компиляции", можно оставить просто слово "тулчейн") обозначают набор средств разработки, совместно используемых для создания программного обеспечения. Обычно это компилятор, ассемблер, линкер, а также набор бинарных утилит. Иногда в состав toolchain включают средства отладки и верификации программ, профилировщики и анализаторы покрытия кода.

1.1 Обзор средств разработки

Обычно весь набор средств разработки скрыт от пользователя за единой программой-драйвером (как в случае GNU Toolchain, когда роль такой программы играет GCC) или за ширмой средств IDE (как в случае Microsoft Toolchain, когда компиляция, ассемблирование и сборка происходят по нажатию кнопки F7 в Visual Studio).

Кроме GNU и Microsoft, известны также LLVM toolchain, Intel toolchain, ARM toolchain, а также пакеты средств разработки Borland и Comeau. Для всех примеров ниже будет использоваться GNU Toolchain, но иногда будут приводиться примеры из других средств разработки, там, где есть некие концептуальные отличия. Точно так же ниже будут описаны в основном примеры компиляции программ на языке C, разве что с некоторыми экскурсиями в C++ и Fortran.

Эта лекция ставит перед собой следующие цели:

- Сделать ясной структуру и последовательность работы системы компиляции в целом
- Осветить роль каждого из инструментов в отдельности, его применимость и решаемые им задачи
- Дать базу для самостоятельных экспериментов (на примере GNU Toolchain)
- Изложить основные этапы портирования toolchain на новую архитектуру

Первый пункт можно переформулировать как простой и конкретный вопрос: что происходит при исполнении следующей строчки?

```
$ gcc program.c
```

Правильный ответ: происходит запуск трёх основных программ, как это показано на (рис. 1).

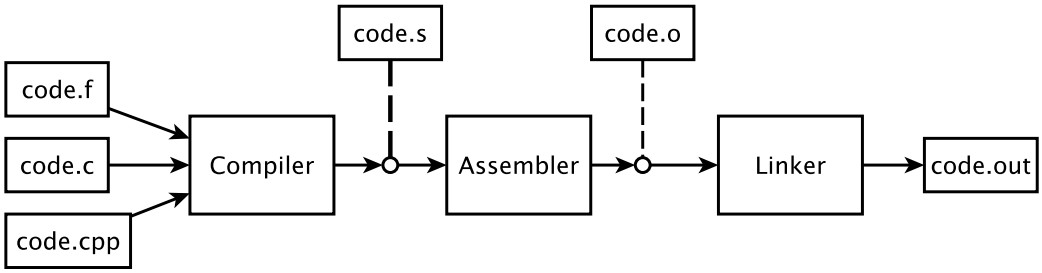

Рис. 1: Упрощенная схема toolchain

1. **Компилятор** – в случае GNU toolchain это cc1 или cc1plus для кода на C или C++, в случае MSVS это cl.exe. Переводит текст на языке высокого уровня в язык ассемблера целевой архитектуры

2. **Ассемблер** – в случае GNU toolchain это as, в случае MSVS это ml.exe. Кодирует ассемблер целевой архитектуры и порождает объектный код

3. **Линкер** – в случае GNU toolchain это ld или gold, в случае MSVS это link.exe. Собирает несколько объектных модулей в исполняемый файл

Toolchain не исчерпывается этой упрощенной схемой, но для начала рассмотрение трёх основных компонент позволяет создать общую картину происходящего.

1.2 Компилятор

Компилятор является главным и наиболее известным компонентом средств разработки (автору не раз доводилось слышать, как “компилятором” называют и драйвер, и весь toolchain). Тем не менее, происходящее внутри компилятора часто так же остается неизвестным разработчику, как и происходящее внутри системы компиляции в целом.

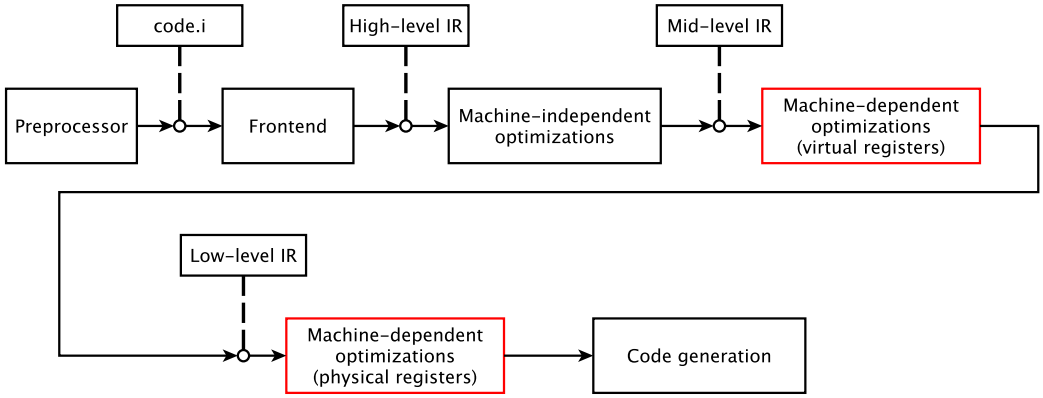

Рис. 2: Компилятор изнутри

На (рис. 2) изображена схема работы компилятора, состоящая из следующих элементов:

1. **Препроцессор** – занимается текстовой обработкой исходного текста и подготовкой к компиляции

2. **Фронтенд** – переводит язык высокого уровня в промежуточное представление

3. **Бэкенд** – состоит из оптимизаций на разных уровнях промежуточного представления и кодогенерации в ассемблер целевой архитектуры. Иногда его делят на middle-end, где выполняются машинно-независимые оптимизации и backend (выделен красным цветом), где происходят машинно-зависимые преобразования.

В LLVM toolchain препроцессор и фронтенд языков C и C++ вынесены в отдельную программу (clang), которая порождает машинно-независимое промежуточное представление (intermediate representation, или для краткости IR), над которым оптимизатор (подключаемый к clang как библиотека) производит преобразования до ассемблера.

Препроцессор часто поставляется отдельно в наборе бинарных утилит, например в составе GNU Toolchain программа, производящая препроцессинг, называется **cpp**. Она, тем не менее, не запускается в нормальной циклограмме работы, так как препроцессор является частью компилятора **cc1**, а cpp это просто отдельно лежащий препроцессор, отпиленный для удобства.

При достаточной универсальности IR, бэкенд тоже может быть вынесен как отдельная программа, таков например бэкенд llc, производящий трансформации LLVM IR и кодогенерацию. И llc и clang подключают одну и ту же библиотеку для использования в качестве бэкенда. Таким образом все три основные части тулчейна могут быть как совмещены в один исполняемый файл, так и разнесены по отдельным запускам в любых комбинациях.

1.2.1 Препроцессор

Работа препроцессора происходит в основном над текстом и в случае языка C регламентирована стандартом языка (5.1.1.2 в C11). В случае gcc, для препроцессирования файла имеется опция -E.

Препроцессор, как это изображено на (рис. 3), выполняет действия в следующем порядке:

- Замена триграфов

  Наиболее эзотерическая часть работы. Триграфы это специальные последовательности символов, ориентированные на консоли, где не было возможности ввести и обработать некоторые спецсимволы языка. Код с использованием триграфов выглядит действительно необычно. Например попробуйте понять, что происходит на этой строчке:

```
1 !ErrorOccured() ??!??! HandleError();
```

  Правильный ответ – здесь вместо вертикальной черты использован триграф `??!`, так что непонятная мешанина знаков в середине это просто длинное или. Ещё пример:

```
1 // WTF????/
2 a += 1;
```

  Здесь использован триграф `??/`, обозначающий обратный слеш. Что приводит к следующему пункту.

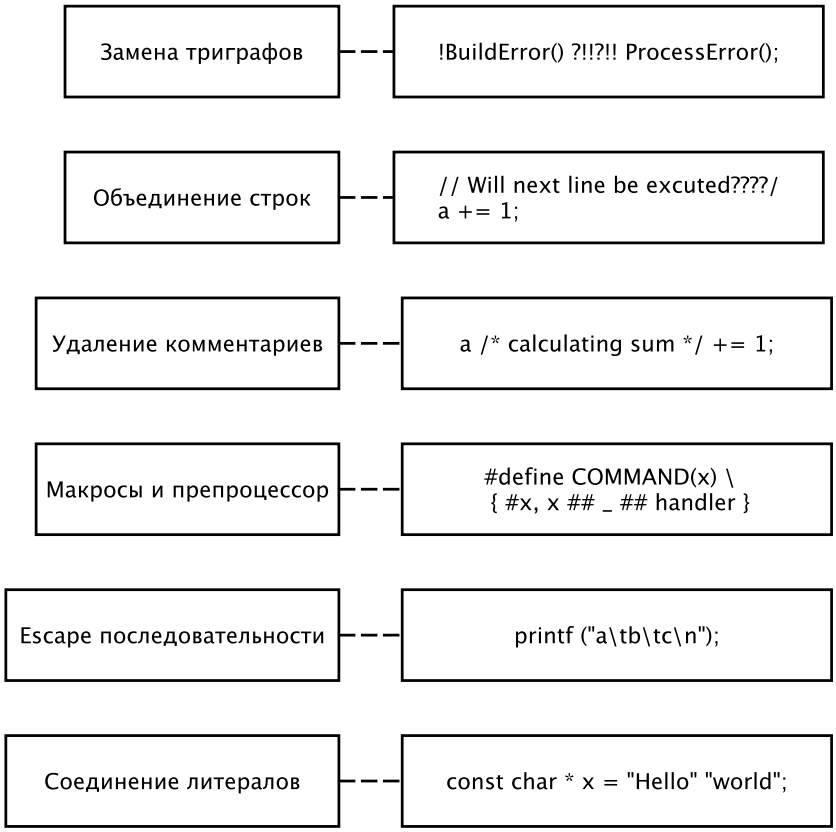

Рис. 3: Схема препроцессора

- Объединение строк по обратному слешу

  В примере выше после раскрытия триграфа остаётся:

```
1 // WTF??\
2 a += 1;
```

  И на этом этапе эти строки будут объединены в:

```
1 // WTF??a += 1;
```

- Разбиение на токены препроцессирования и замена комментариев пробелами

  Здесь будут вычеркнуты все комментарии и заменены одним пробельным символом.

- Раскрытие макросов и обработка директив препроцессора

На этом этапе раскрываются все макро-определения, как явно введенные через директиву `#define`, так и неявно заданные как билтины компилятора (скажем `__cplusplus` позволяющий отличить компиляцию на C и C++). В случае GNU Toolchain, посмотреть встроенные макроопределения можно с помощью опций -E -dM.

- Замена escape-последовательностей

  Я полагаю, все встречались с escape-последовательностями `\n` (перенос строки) или `\t` (табуляция) при использовании функции printf и ей подобных. На самом деле их гораздо больше, например `\0` можно использовать для обозначения завершающего нуля в строковом литерале. Полный список всегда можно посмотреть в стандарте языка.

- Соединение стоящих рядом строковых литералов

  Это позволяет писать строчки вида:

```
1 #define HELLO "Hello"
2 printf (HELLO ", world!\n");
```

  После макроподстановки получившийся составной литерал `"Hello "", world\n"` пройдет конкатенацию строк и превратится в одну строку.

Файл, получившийся после препроцессирования, называется единицей трансляции (translation unit) и обрабатывается компилятором. В некоторых языках (например Fortran) несколько единиц трансляции могут располагаться в одном файле.

1.2.2 Фронтенд

Фронтенд компилятора работает с препроцессированным файлом и выполняет тяжелую задачу разбора языковых конструкций и перевода программы с конкретного языка программирования (например C++) в высокоуровневое промежуточное представление, удобное для машинно-независимых оптимизаций. Обычно фронтенд осуществляет:

- Лексический анализ: выделение ключевых слов языка, имён переменных и символов операций Пример: `int z = x+++y;` будет распознано как `int z = (x++)+y;` (слева направо), но при этом `int z = x = y;` будет распознано как `int z = (x = y);` (справа налево)

- Синтаксический анализ и построение дерева синтаксического разбора (см. пример на рис. 4)
- Анализ и вывод типов выражений

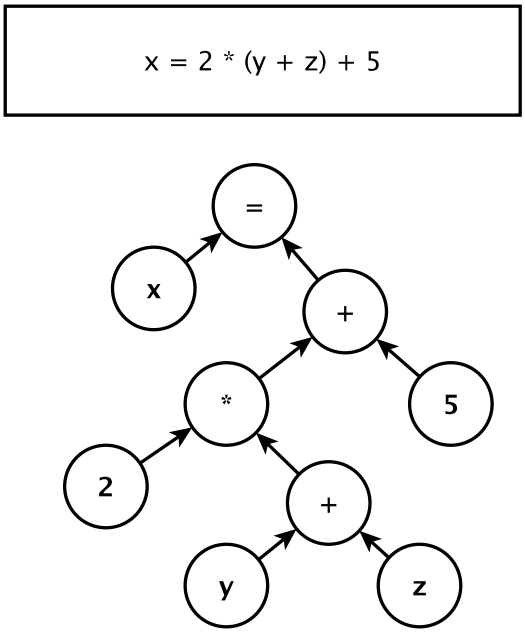

Рис. 4: Дерево синтаксического разбора

Иногда фронтенду приходится решать довольно сложные задачи, связанные с тем, что современные языки высокого уровня (такие как C++) могут иметь неочевидный синтаксис и семантику.

Классический пример:

```
ifstream datafile ("ins.dat");
list<int> data (istream_iterator<int>(dataFile),
                istream_iterator<int>());
```

Может показаться удивительным, но вторая строчка содержит объявление функции, а вовсе не заполнение контейнера из файла. Спасает простановка скобок:

```
ifstream datafile ("ins.dat");
list<int> data ((istream_iterator<int>(dataFile)),
                istream_iterator<int>());
```

Ещё один пример:

```
template <class T>
foo (T x)
{
  T::iterator *y;
  /* .... */
}
```

Увы, без дополнительных подсказок, синтаксический разбор определяет это как умножение некоего статического члена `T::iterator` на некоторую глобальную переменную `y`. Чтобы помочь фронтенду, здесь уместно добавить ключевое слово `typename`:

```
template <class T>
foo (T x)
{
  typename T::iterator *y;
  /* .... */
}
```

Как правило, результатом работы фронтенда является промежуточное представление (Gimple в случе GCC, LLVM IR в случае LLVM и так далее).

1.2.3 SSA представление

SSA это аббревиатура для static single assignment. Программа считается представленной в SSA-форме, если каждой именованной переменной в программе значение может быть присвоено только один раз. Одним из важных следствий этого определения является referencial transparency – то свойство программы, когда значение переменной независимо от её положения в программе.

```
x = 1;
y = x + 1;
x = 2;
z = x + 1;
```

Эта программа является корректным кодом на C, но значение `x` зависит от анализируемой строчки, в результате чего `y = x + 1` и `z = x + 1` имея идентичную форму определения, имеют разные значения. В случае SSA-формы этот код может быть переписан:

```
x1 = 1;
```

```
y = x1 + 1;
x2 = 2;
z = x2 + 1;
```

Здесь очевидно, что разные значения соответствуют разной форме определения. Но для линейных участков всё просто. Сложнее преобразовать в SSA форму ветвления. Например:

```
x = input ();
if (x > 42)
  y = 1;
else
  y = x - 2;
use (y);
```

Здесь для того, чтобы избавиться от двух определений `y`, нужно ввести ϕ-функцию которая будет "выбирать" верное значение `y` из двух пришедших к ней y_1 и y_2, как показано на рисунке:

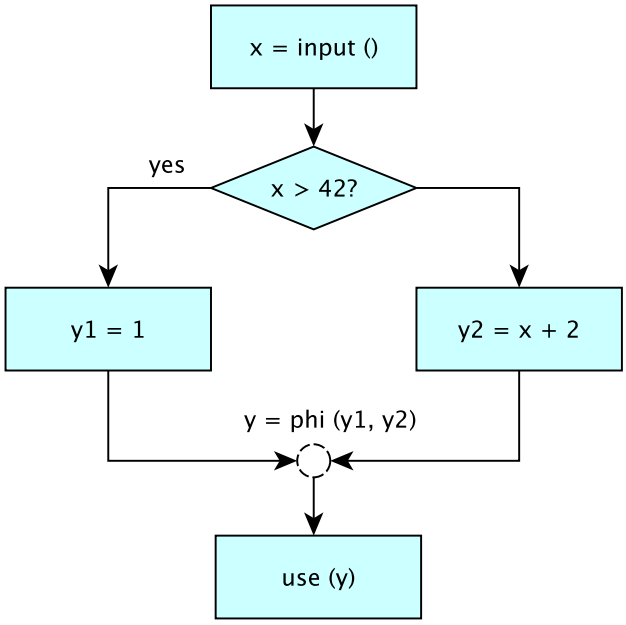

Рис. 5: $y = \phi(y_1, y_2)$

Изначально такие функции получили своё название от phony-functions (фальшивые функции), но название прижилось и сократилось. Обозначения `phi(y1, y2)` и $\phi(y_1, y_2)$ далее будут достаточно свободно взаимозаменяемы.

К сожалению, их никак нельзя переписать на C, но в GIMPLE дампах они выглядят так:

```
  <bb 3>:
  # n_5 = PHI <5(2), _6(4)>
  # mult_acc_7 = PHI <1(2), mult_acc_8(4)>
...
```

Эта запись означает, что есть два пути, по которым управление сходится в `bb3` из блоков `bb2` (там где в скобках стоит 2) и `bb4` для `n_5`, причём внутри `bb2` определение этой переменной имеет имя `5` (по правилам GIMPLE это просто константа 5), а в `bb4` имеет имя `_6` (искуственно созданные временные переменные в GIMPLE часто имеют безликие имена `_1`, `_2` и так далее).

1.2.4 Машинно-независимые преобразования

Для примера работы промежуточного представления можно рассмотреть функцию, вычисляющую (с первого взгляда – крайне неоптимально) факториал числа, переданного в качестве аргумента.

```
unsigned
fact (unsigned x)
{
  if (x < 2)
    return 1;

  return x * fact (x-1);
}
```

Трансформации сильно зависят от опций конкретного компилятора, рассмотрим этот пример как скомпилированный с `-O2` в 64-битном режиме (обычно чем выше уровень, тем сильнее оптимизация, уровень 2 означает применять большинство полезных оптимизаций и оптимизировать код по скорости исполнения).

После работы фронтенда GCC эта функция представлена в виде машинно-независимого языка промежуточного представления GIMPLE. В виде графа её можно изобразить как показано на рис. 6.

Для получения дампа можно воспользоваться опцией `-fdump-tree-ssa`, а в общем случае опцией `-fdump-tree-all`, позволяющей посмотреть все результаты работы всех фаз на уровне GIMPLE.

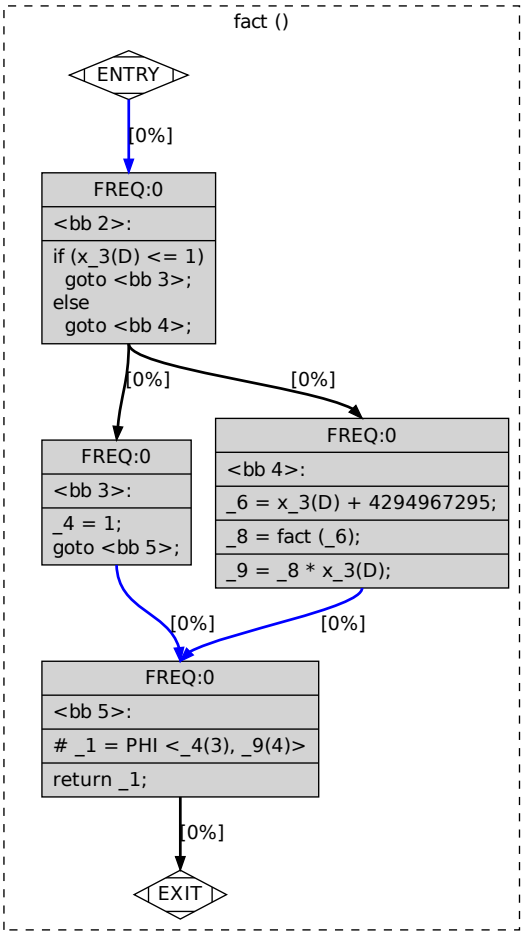

Рис. 6: GIMPLE на начальных фазах

Программа на рис. 6 практически повторяет структуру исходного кода на C, с учётом перевода в SSA и расстановки ϕ-функций. Далее за дело берутся машинно-независимые оптимизации, которых в GCC 5.2 насчитывается более двухсот. Уже через несколько итераций происходит подстановка хвостового вызова, и код начинает выглядет гораздо интереснее, как показано на (рис. 7).

Пунктирной линией показана обратная дуга цикла, который получился в результате раскрытия хвостовой рекурсии. Кроме того, следует обратить внимание на расставленные вероятности переходов – здесь они расставлены компилятором априорно. Их можно сделать гораздо более точными, если использовать при компиляции профиль реального запуска программы.

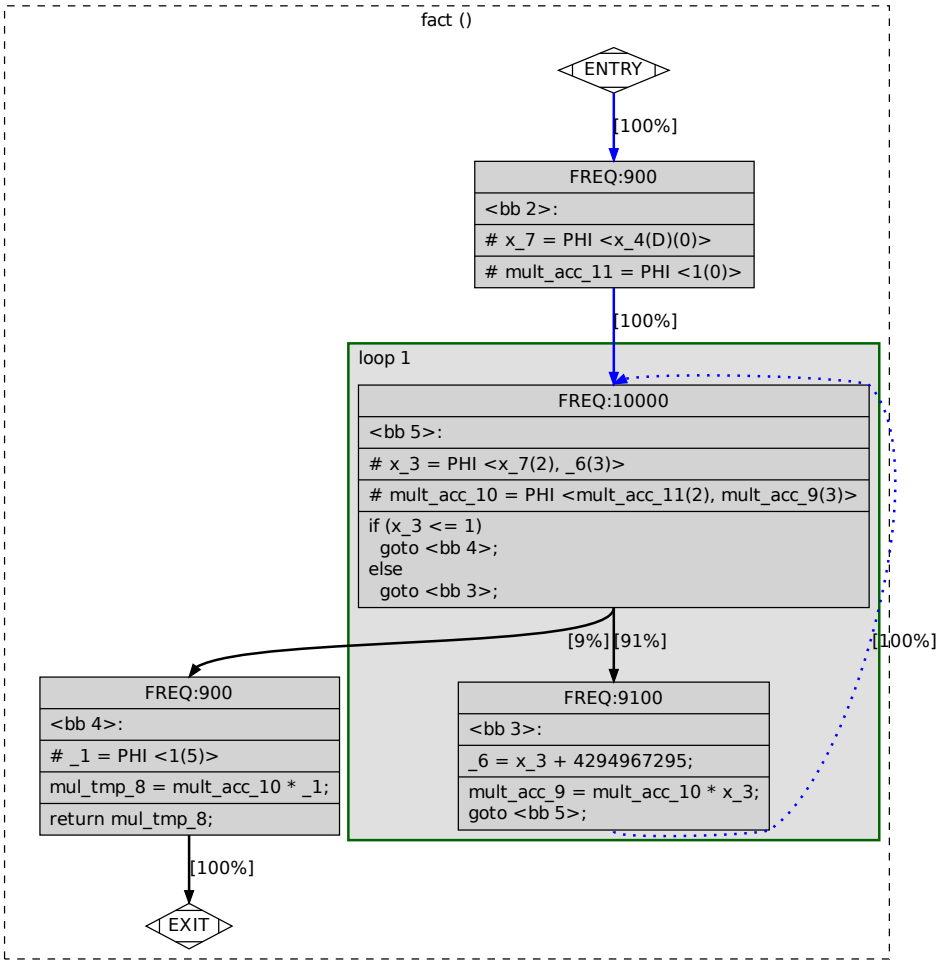

Рис. 7: GIMPLE после tail-call оптимизаций

1.2.5 Машинно-зависимые преобразования

После окончания машинно-независимых преобразований код переводится в машинно-зависимую форму, называемую в случае GCC register transfer language или RTL. Интересно, что этот перевод называется поразному в разных компиляторах. В мире LLVM говорят “lowering”, выражая этим понижение уровня представления. В мире GCC говорят “expand” – раскрытие машинно-независимых конструкций в машинно-зависимый формат.

Все инструкции в машинно-зависимом формате зависят от описания конкретной машины в бэкенде компилятора и на ранних фазах RTL оперируют с виртуальными регистрами. Виртуальные регистры это не SSA

представление, это абстрактная регистровая машина, в которой количество регистров не ограничено.

Получить дампы RTL в GCC можно с помощью опции `-fdump-rtl-all`, где, как обычно, вместо all можно подать символическое имя фазы (`-fdump-rtl-expand` для кода сразу после раскрытия и так далее).

Чтение дампов RTL осложняется тем фактом, что они представлены в сложном LISP-подобном формате:

```
(insn 15 14 16 5 (parallel [
            (set (reg:SI 90 [ D.1850 ])
                (mult:SI (reg:SI 90 [ D.1850 ])
                    (reg/v:SI 92 [ x ])))
            (clobber (reg:CC 17 flags))
        ]) -1
     (nil))
```

На самом деле выше происходит следующее: в виртуальный регистр с номером 90 записывается произведение виртуальных регистров 90 и 92, параллельно с чем непредсказуемо изменяется флаговый регистр. Для простоты это можно записать как:

```
1 v90 = v90 * v92;
2 clobber (v17);
```

Здесь буква v подчеркивает, что регистры виртуальные. На рис. 8 изображен код факториала сразу после раскрытия в машинно-зависимый формат.

Можно заметить, что каждому регистру сопоставлен режим (register mode), в котором он используется. Некоторые из этих режимов перечисленны ниже:

1. SI – одно машинное слово (single integer mode). Обычно соответствует 32 битам.
2. DI, HI, QI – двойное слово, половинное и четверть слова (double, half, quad)
3. CC, CCZ – регистры флагов

Использование виртуального регистра в каком-то режиме не отменяет того, что после окончания жизни этой переменной он не будет переиспользован в каком-то другом режиме.

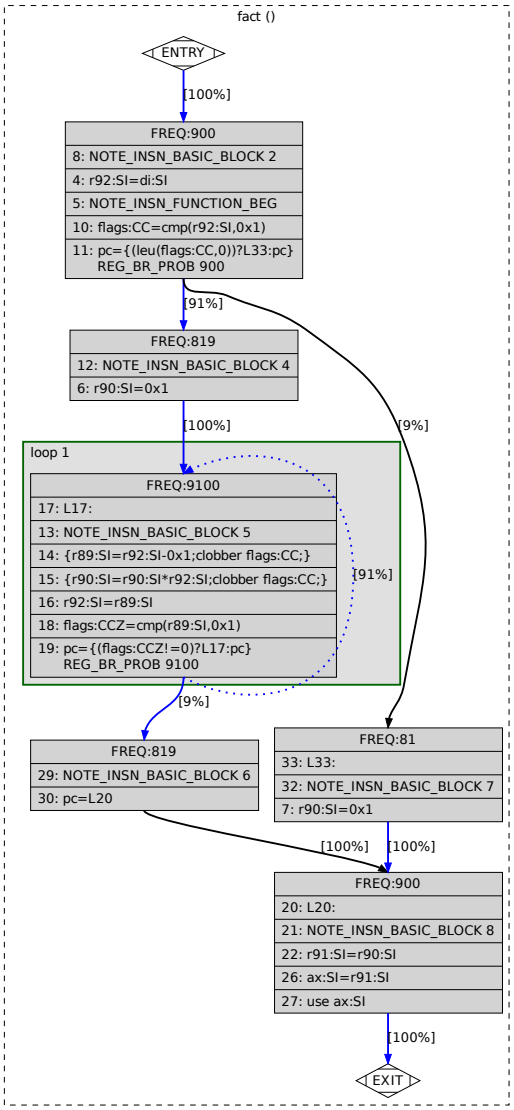

Рис. 8: RTL с виртуальными регистрами

1.2.6 Распределение регистров и кодогенерация

После выполнения преобразований на виртуальных регистрах, должен отработать распределитель регистров, который отображает виртуальные регистры в физические, заданные в описании целевой архитектуры. Для x86 это ax, bx, cx, dx и так далее. Пример RTL с физическими регистрами приведен на рис 9

Виртуальных регистров обычно больше чем физических, поэтому рас-

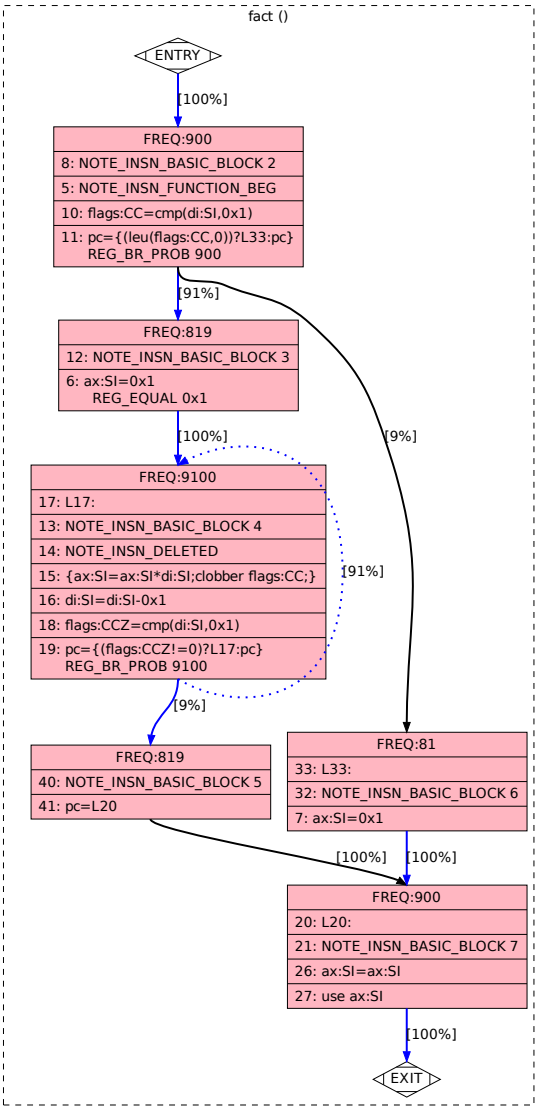

Рис. 9: RTL с физическими регистрами

пределение регистров не может быть проведено один к одному, а должно использовать временные слоты в памяти для материализации (spilling) и последующего восстановления (filling) регистров.

Может показаться удивительным, но многие преобразования требуют представляения на физических регистрах, например расписание (scheduling) инструкций требует информации о конкретных задержках и модели конвейера и таким образом должно проводится только когда уже известно окончательное распределение переменных по регистрам и памяти в оп-

тимизированном коде. Точно так же переименование регистров с целью облегчить регистровое давление бесполезно проводить на этапе виртуальных регистров, потому что там сам этот термин не имеет смысла.

После отработки оптимизаций на физических регистрах (if-conversion, переупорядочение базовых блоков, расписание инструкций), кодогенератор в GCC порождает ассемблер целевой архитектуры (в заданном синтаксисе).

В других компиляторах детали происходящего могут отличаться: в LLVM машинно-независимый IR очень похож на машинно-зависимый, в ICC промежуточное представление скрыто от пользователя и так далее. Но общая схема везде одна и та же: выскоуровневый IR (обычно машинно-независимый) преобразуется в машинно-зависимый IR среднего уровня, который после распределения регистров преобразуется в низкоуровневый IR. Который, в свою очередь, уже и используется для генерации кода.

1.3 Ассемблер

Важно различать язык ассемблера как язык и ассемблер как программу.

- **Язык ассемблера** это удобная символьная форма записи (мнемокод) команд и их аргументов
- **Ассемблер** это программа, переводящая язык ассемблера в объектный код

Язык ассемблера целевой архитектуры задается документом, который называется Instruction Set Architecture (ISA) и является своим для каждой архитектуры. Более того, даже в пределах одной архитектуры могут существовать несколько синтаксисов языка ассемблера (пример – AT&T и Intel синтаксисы ассемблера x86).

1.3.1 Ассемблирование

Ассемблер как программа обрабатывает текстовый файл на языке ассемблера конкретной архитектуры и порождает объектный код для последующей линковки и исполнения.

В качестве примера, ассемблер (x86, синтаксис AT&T), порождённый из функции факториал может выглядеть следующим образом:

```
  .file "fact.c"
  .section  .text.unlikely,"ax",@progbits
.LCOLDB0:
  .text
.LHOTB0:
  .p2align 4,,15
  .globl  fact
  .type fact, @function
fact:
.LFB0:
  .cfi_startproc
  cmpl  $1, %edi
  movl  $1, %eax
  jbe .L4
```

```
  .p2align 4,,10
  .p2align 3
.L3:
  imull %edi, %eax
  subl  $1, %edi
  cmpl  $1, %edi
  jne .L3
  rep; ret
.L4:
  rep; ret
  .cfi_endproc
.LFE0:
  .size fact, .-fact
```

В нём можно выделить:

- Инструкции x86: `cmpl`, `movl`, `jbe`, `imull`, `subl` и прочие
- Метки: `fact`, `LFB0`, `L3`, `L4`, `LFE0` Можно заметить, что одна из этих меток – `fact` объявлена глобальной, то есть попадёт в таблицу экспортируемых из этого модуля функций
- Директивы: `globl`, `type`, `size`, `p2align`
- Отладочную информацию: `cfi_startproc`, `cfi_endproc`
- Секции `text`, `text.unlikely`

Ассемблер выполняет следующую работу:

1. Кодирование инструкций
2. Подстановка абсолютных значений переходов (где они известны) вместо символьных меток
3. Ассемблирование секций (кода, данных, стека)
4. Исполнение директив ассемблера (выравнивание, отладочная информация, прочее)
5. Формирование таблиц экспорта

1.3.2 Макроассемблер

Почти любой ассемблер (включая ml.exe от Microsoft, GNU AS, а также отдельные распространённые ассемблеры, такие как wasm и nasm) поддерживает не только свои основные функции по обработке выдачи компилятора, но также предоставляет определенные средства для разработки собственно на языке ассемблера. В современном мире чистая разработка на ассемблере это экзотика, но она может быть нужна для работы с системными регистрами, использовании специфичных возможностей архитектуры и так далее – в общем для всего, чего нельзя написать на C.

Препроцессор ассемблера обычно позволяет:

- Объявлять и использовать макросы. Например приведенный ниже макрос для GNU AS:

```
.macro  sum from=0, to=5
  .long   \from
  .if     \to-\from
  sum     "(\from+1)",\to
  .endif
.endm
```

  При вызове `SUM 0,5` раскроется в:

```
  .long   0
  .long   1
  .long   2
  .long   3
  .long   4
  .long   5
```

- Включать ассемблерный код из других файлов (директива `.include "file"` в GNU as или аналогичные ей).
- Писать комментарии. Стили комментирования очень разнообразны и отличаются от ассемблера к ассемблеру. Большинство ассемблеров также поддерживают комментарии в стиле C.

- Также поддерживаются дополнительные (иногда – удивительно разнообразные) возможности. Так например в GNU as символ точка обозначает адрес текущей ассемблируемой инструкции. Поэтому запись:

  ```
  melvin: .long .
  ```

  Обозначает, что метка melvin содержит собственный адрес.

Кроме общих для всех архитектур возможностей ассемблера в рамках его синтаксиса, каждая архитектура может определять набор своих средств для задания адресов в памяти, обращения к регистрам и так далее.

1.3.3 Дизассемблирование

Результатом работы ассемблера является объектный файл, который, в свою очередь, может быть дизассемблирован, то есть переведен обратно в ассемблерное представление. В случае GNU toolchain соответствующая утилита называется objdump.

```
$ as fact.s -o fact.o
$ objdump -d fact.o
```

Ниже рассмотрен вывод ассемблированного и сразу же дизассемблированного кода:

```
 0: 83 ff 01             cmp    $0x1,%edi
 3: b8 01 00 00 00       mov    $0x1,%eax
 8: 76 13                jbe    1d <fact+0x1d>
 a: 66 0f 1f 44 00 00    nopw   0x0(%rax,%rax,1)
10: 0f af c7             imul   %edi,%eax
13: 83 ef 01             sub    $0x1,%edi
16: 83 ff 01             cmp    $0x1,%edi
19: 75 f5                jne    10 <fact+0x10>
1b: f3 c3                repz retq
1d: f3 c3                repz retq
```

Дизассемблер аннотирован кодировкой (ниже будут рассмотрены способы аннотировать дизассемблер многим другим, в том числе номерами строк и даже самими строчками исходного кода программы).

Кроме того видно, что две p2align директивы породили выравнивающий nop.

1.4 Линкер

Линкер это программа, предназначенная для связывания (linking) нескольких объектных файлов в один исполняемый. Линкер входит во все известные наборы средств разработки – в случае GCC он называется ld (или gold, это улучшенная версия ld от Google), в случае Microsoft это link.exe, есть и другие варианты линкеров. В случае LLVM, конкретная архитектура может реализовать свой линкер как библиотеку или использовать из-под драйвера вызов любого существующего линкера.

1.4.1 Объявления и определения

Прежде чем переходить к рассмотрению линкера, необходимо расширить тестовую задачу. Пусть функция `fact` физически находится в файле `fact.c` и кроме него есть файл `main.c`, задача которого – распечатать факториал пяти, выполнив для этого строчку вроде следующей:

```
printf ("%d\n", fact(5));
```

Но откуда вообще компилятор при обработке файла `main.c` узнает что такое `fact` и что такое `printf`? Для того, чтобы какая-то функция могла быть использована на этапе компиляции, она должна быть **объявлена** – то есть указано её имя, типы её аргументов и тип возвращаемого значения. Обычно объявления собираются вместе в заголовочные файлы. Есть стандартные заголовочные файлы, такие, как `stdio.h`, содержащий объявление функции `printf`. Для факториала, заголовочный файл `fact.h` может выглядеть как-то так:

```
#ifndef FACT_GUARD_
#define FACT_GUARD_

extern unsigned fact (unsigned);

#endif
```

Теперь программа, распечатывающая факториал может быть написана как-то так:

```
#include <stdio.h>
#include "fact.h"

int
main (void)
```

```
6 {
7   printf ("%d\n", fact(5));
8   return 0;
9 }
```

Этот файл использует функцию, которая **определена** в другом месте – в данном случае в файле `fact.c`. Если попытаться сделать исполняемый файл из одного `main.c`, это завершится ошибкой:

```
$ gcc main.c
/tmp/ccunVXEF.o: In function ‘main’:
main.c:(.text+0xa): undefined reference to ‘fact’
collect2: ld returned 1 exit status
```

Компилятор видит тип функции `fact` и знает как её вызвать, но линкер не может разрешить этот вызов. Можно сказать, что в коде на C, объявления пишутся для компилятора (достаточны чтобы сгенерировать в ассемблере вызов на внешнюю метку), а определения для линкера (достаточны для разрешения этой метки в программе, состоящей из множества единиц трансляции).

Простейший способ успешно скомпилировать искомую программу успешно это подать оба файла драйверу:

```
$ gcc main.c fact.c
```

Но что и в каком порядке при этом будет сделано?

- Будут скомпилированы отдельно `main.c` и `fact.c` до ассемблера (временные файлы будут скорее всего сохранены куда-то в недоступное для глаз пользователя место).
- Оба получившихся ассемблерных файла будут сассемблированы до объектного кода (скорее всего будет размещен там же)
- Получившийся объектный код через специальный скрипт `collect2` будет подан линкеру (в случае GNU Toolchain это `ld` или `gold`).

Как обычно опция `-save-temps` позволяет сохранить промежуточные результаты работы.

1.4.2 Статическая линковка

Существует два вида линковки – статическая и динамическая. Дефакто стандартным умолчанием в большинстве средств разработки является динамическая линковка (и её преимущества, благодаря которым это так, тоже будут рассмотрены). Но статическая линковка гораздо проще для объяснения, поэтому следует начать с неё. Статически слинкованный пример с сохранёнными временными файлами можно получить следующим образом.

```
$ gcc main.c fact.c -static --save-temps
```

Или с явным вызовом `ld` и сохранением порядка секций:

```
$ gcc main.c -c
$ gcc fact.c -c
$ ld -static main.o fact.o
```

Для исследования происходящего (под Linux) удобна утилита `readelf` из набора бинарных утилит. Анализироваться ниже будет её вывод с опцией `readelf -s`. Можно увидеть, что для объектного файла `main.o` список функций выглядит как-то так:

```
     9: 0000000000000000    43 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND fact
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Здесь одна глобальная метка `main` и две не разрешенных `fact` и `printf`. Теперь для объектного файла `fact.o`:

```
     8: 0000000000000000    43 FUNC    GLOBAL DEFAULT    1 fact
```

можно увидеть, что определение этой функции в нём есть. При статической линковке, линкер собирает все объектные файлы и разрешает их метки относительно друг друга в адресах получившегося общего файла. В итоге для результата `a.out` картина выглядит так:

```
   901: 0000000000401740   157 FUNC    GLOBAL DEFAULT    6 printf
  1545: 0000000000400434    43 FUNC    GLOBAL DEFAULT    6 main
  1833: 0000000000400460    43 FUNC    GLOBAL DEFAULT    6 fact
```

Здесь во второй колонке указаны адреса в объектном файле и те же адреса можно увидеть в дизассемблере (всё так же `objdump -d` для GNU Toolchain ну или ваш любимый дизассемблер в вашей системе).

```
0000000000400434 <main>:
  400434: 55                    push   %rbp
  400435: 48 89 e5              mov    %rsp,%rbp
  400438: bf 05 00 00 00        mov    $0x5,%edi
  40043d: e8 1e 00 00 00        callq  400460 <fact>
  400442: 89 c2                 mov    %eax,%edx
....
0000000000400460 <fact>:
  400460: 55                    push   %rbp
  400461: 48 89 e5              mov    %rsp,%rbp
....
```

Можно заметить, что в кодировку инструкции `call` (в x86 ассемблере это `e8`) подставлено относительное смещение `1e`, и действительно (дисплейсмент вызова считается относительно начала следующей инструкции) получается, что `400442 + 1e = 400460`.

В этом суть статической линковки: объектный код объединяется, разрешаются ссылки и ссылки подставляются всюду, в том числе и в ассемблерные инструкции такие как `call`. Если посмотреть дизассемблер объектного (ещё не не слинкованного) файла, можно увидеть, что вызов там не разрешён. В конкретный адрес, вместо этого подставлены нули:

```
0000000000000000 <main>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: bf 05 00 00 00        mov    $0x5,%edi
   9: e8 00 00 00 00        callq  e <main+0xe>
   e: 89 c2                 mov    %eax,%edx
```

Это интерпретируется как вызов с передачей управления на следующую инструкцию. Но это конечно не так, речь просто о не подставленной на этом этапе **релокации**. Релокацией (relocation information или попросту релоком) называется метаинформация в объектном файле обо всех местах, куда линкер должен подставить адреса при линковке.

Информацию о релокациях можно считать из объектного файла:

```
readelf -r main.o
00000000000a  000a00000002 R_X86_64_PC32  00000000 fact - 4
000000000011  00050000000a R_X86_64_32    00000000 .rodata + 0
000000000020  000b00000002 R_X86_64_PC32  00000000 printf - 4
```

Первая в этом списке (по адресу `0xa`) это и есть та самая, которая должна быть подставлена в `call`, а конкретно `fact` минус размер инструкции вызова.

Хорошо, допустим с `fact` всё понятно. Но где посмотреть объектник, который используется при линковке с функцией `printf`? Здесь не всё так очевидно, потому что эта функция берётся из **библиотеки**.

1.4.3 Статические библиотеки

Статическая библиотека (static library) это способ организации объектного кода, при котором большое число раздельно сассемблированных файлов заменяется одним, объединяющим их все и также содержащим некую метаинформацию.

В MSVS собрать статическую библиотеку можно настроив опции линковщика, в то время, как в GNU Toolchain, для этого есть отдельная утилита, называющаяся `ar`. На примере уже рассмотренного факториала, можно собрать статическую библиотеку из одного модуля:

```
ar -cr libfact.a fact.o
```

В конце этой строки можно указать сколько угодно объектных файлов и все они будут объединены (также говорят – упакованы) в одну библиотеку. Ей можно пользоваться непосредственно:

```
gcc -static main.c libfact.a
```

Но общепринятый способ использования – указание пути и сокращенного имени, которые достаточны драйверу, чтобы сформировать для линкера полное имя:

```
gcc -static -L. main.c -lfact
```

Также конвенциальным является сокращение `libxxx.a` в записи аргументов командной строки до `-lxxx`. Например `libm.a` записывается как `-lm`, а `clib.a` как `-cl`. Именно таким образом на сборку подаются все стандартные библиотеки, причем даже если их явно не указать. Так например запустив указанную выше строчку с дополнительной опцией `verbose`, можно увидеть окончательную строчку вызова линкера вида:

```
collect2 -m elf_x86_64 -static crtbeginT.o -L.
-L<other paths here> main.o -lfact --start-group
-lgcc -lgcc_eh -lc --end-group crtend.o crtn.o
```

Видно, что драйвер тихо добавил в опции линкера несколько стандартных библиотек и объектных файлов, в частности libc, откуда и берется определение для функции `printf`.

Ту же утилиту, которая отвечает за создание библиотек, можно использовать и для исследования библиотек. Например вот так можно распечатать все объектные файлы из которых собрана `libc`:

```
ar -vt /usr/lib/libc.a
```

1.4.4 Динамическая линковка

Статические библиотеки хороши и просты, но у них есть недостатки:

- Большой размер файла на диске
- Тратится много памяти при исполнении нескольких файлов, использующих одну и ту же библиотеку

Чтобы преодолеть эти проблемы, используется динамическая линковка. В этом случае, код из библиотеки связывается с кодом использующей её программы только после того, как программа загружена.

Всё, что будет сказано ниже актуально для Unix-подобных систем. В случае Windows, динамические библиотеки (так называемые DLL) работают несколько иначе, например там нет настоящего позиционно-независимого кода и всего такого.

Сборка в динамическую библиотеку с помощью GNU Toolchain под OS Linux проходит в два этапа:

```
gcc -c -fPIC -O2 fact.c
gcc -shared fact.o -o libfact.so
```

Опция `-fPIC` определяет необходимость сассемблировать код так, чтобы он был positional-independent, то есть не содержал прямых вызовов функций. На примере `main.c` можно посмотреть типичный positional-independent ассемблер:

```
main:
  subq  $8, %rsp
  movl  $5, %edi
  call  fact@PLT
...
```

Отметка `@PLT` здесь сообщает о необходимости сгенерировать вызов fact относительно PLT (procedure linkage table). На (рис. 10) показана общая схема работы.

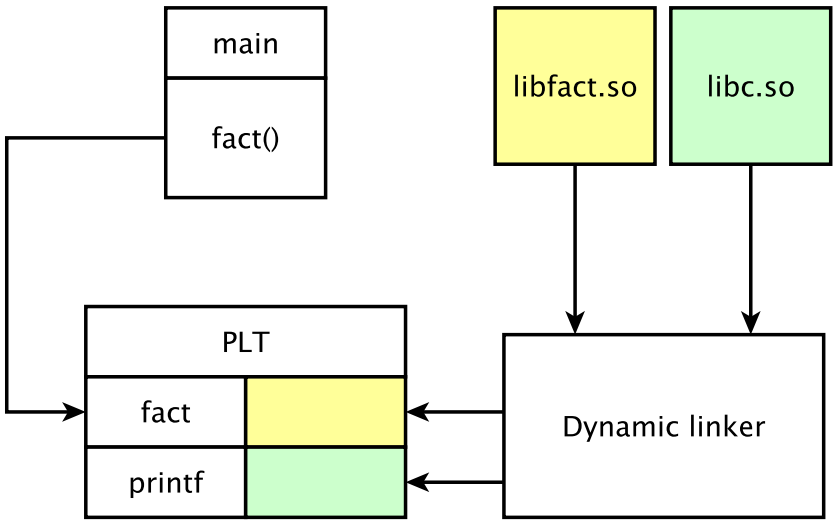

Рис. 10: Динамическая линковка

Окончательное связывание вызываемой функции с её точой входа в таблице PLT осуществляется сравнительно поздно специальной программой, известной как динамический линкер (dynamic linker). Часто на него ссылаются по его типичному имени `ld.so`.

Для данных, прямые ссылки на которых тоже неизвестны на этапе линковки, существует GOT – global offset table, которая тоже заполняется динамическим линкером уже после загрузки программы.

1.4.5 Динамические и смешанные библиотеки

После того, как `libfact.so` построена, использовать её для сборки кода с ней не сложнее, чем в случае статических библиотек.

```
gcc -fPIC main.c -L. -lfact -Wl,-rpath,.
```

Драйвер достаточно умён, чтобы в случае динамической сборки заменить короткое имя `-lxxx` на `libxxx.so`, как в случае с `-lfact` выше.

Поскольку динамические библиотеки не объединяются с кодом исполняемого файла, они должны лежать отдельно (и лежать по правильным путям). Выше опция `-rpath` указывает начинать поиск зависимостей с текущей папки. Она подаётся под специальным флажком `Wl`, который прокидывает все свои опции непосредственно линкеру, минуя обычную обработку драйвером. Аналогичный флажок `Wa` существует и для ассемблера.

В слинкованную программу, строчки поиска динамических библиотек записываются как метаинфрмация (зависимости). Чтобы посмотреть все зависимости того или иного файла можно использовать специальную утилиту ldd:

```
$ ldd a.out
linux-vdso.so.1 =>  (0x00007fff6956b000)
libfact.so => ./libfact.so (0x00007f86adf83000)
libc.so.6 => /lib/libc.so.6 (0x00007f86adbd2000)
/lib64/ld-linux-x86-64.so.2 (0x00007f86ae186000)
```

Видны зависимости от динамического линкера, libc, и libfact, а также зависимость от vdso – это системный механизм Linux.

Динамические библиотеки имеют свои недостатки:

- Каждый вызов идёт по косвенности, что заставляет платить за него дополнительный переход
- Программа может не заработать если библиотеки будут не найдены или лежать по неправильным путям

В случае крупных блиблиотек, таких как libc, достоинства с лихвой перекрывают недостатки. Но в случае факториала, хорошо бы прилинковать его статически, оставив libc по возможности динамической. Такая

возможность есть почти во всех тулчейнах, в GNU Toolchain она реализуется логическими скобками Bstatic/Bdynamic.

```
gcc test.c -Wl,-Bstatic -lfact -Wl,-Bdynamic -L.
```

Здесь факториал будет прилинкован статически, а всё остальное, что линкер подставит по умолчанию будет линковаться динамически.

1.5 Уточнения

Прежде чем двигаться дальше, полезно ещё раз окинуть взглядом ту общую картину которая пока что получилась.

На (рис. 11) представлена схема системы компиляции, более подробная, по сравнению с (рис. 1) и гораздо более реалистичная.

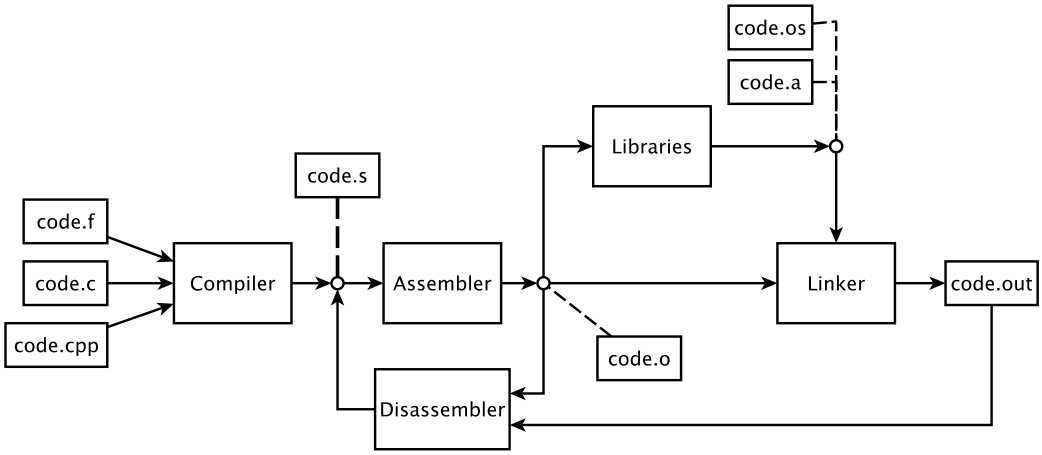

Рис. 11: Уточненная схема toolchain

На этой схеме всё ещё не отображены:

- Большинство бинарных утилит
- Стандартные библиотеки
- Средства отладки и профилировки

Далее все эти дополнительные средства будут разобраны подробно.

1.5.1 Форматы исполняемых файлов

Финальным результатом работы системы компиляции являются, как видно из (11) исполняемые или объектные файлы. На самом деле форматов этих файлов тоже бывает достаточно много и хороший линкер может порождать файлы разных форматов в зависимости от требований пользователя.

Наиболее известные форматы файлов:

- a.out – древний, но всё ещё популярный за свою минималистичность формат (в качестве исполняе)
- ELF – де-факто стандарт для исполняемого и объектного кода в Unix-подобных системах
- PE – де-факто стандарт (включая свою доработку PE32+) в исполняемого кода Windows-подобных системах
- COFF – популярен (как и его доработка ECOFF) для объектных файлов

Далее будет предполагаться что везде речь идёт именно об ELF файлах.

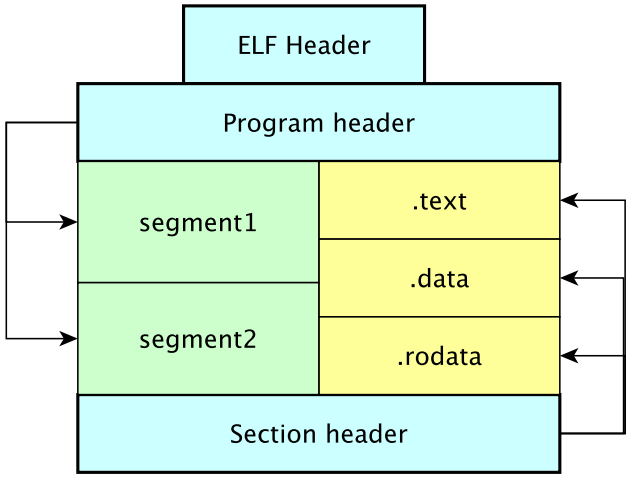

Рис. 12: Формат ELF файла

ELF это аббревиатура от "Executable and Linkable Format изначально он был разработан и опубликован компанией USL как часть двоичного интерфейса приложений операционной системы UNIX System V. Каждый такой файл имеет два заголовка: program header, описывающий сегменты файла и section header, описывающий секции файла.

Сегменты являются информацией для загрузчика, в то время как секции являются информацией для линкера. Сегменты и секции могут пересекаться (и обычно пересекаются, как это изображено на рис. 12), но в файле могут быть также и байты не относящиеся ни к одной секции.

2 Вокруг toolchain

Системы компиляции не существуют в одиночестве. Программу мало скомпилировать, сассемблировать и слинковать, жизненный цикл современного ПО требует отладки, профилировки, поддержки со стороны стандартных библиотек.

2.1 Стандартная библиотека

Стандарты большинства современных языков программирования регламентируют поставку вместе с компилятором языка так называемой "стандартной библиотеки" – набора часто используемых подпрограмм с хорошо документированным поведением.

Классикой жанра является стандартная библиотека языка C. Можно перечислить несколько разновидностей:

- GNU C Library – самая распространенная реализация, используемая в Linux
- Microsoft C Run-time Library
- Dietlibc – альтернативная небольшая реализация Стандартной библиотеки Си
- uClibc – Стандартная библиотека Си для встраиваемых систем на базе Linux
- Newlib – Стандартная библиотека языка Си для встраиваемых систем
- Klibc – применяется главным образом для загрузки Linux-систем
- Eglibc – разновидность glibc для встраиваемых систем
- bionic – реализация стандартной библиотеки в Android

Обычные требования к стандартной библиотеке: стабильность и качество реализации библиотечных подпрограмм, максимальное использование специфики архитектуры.

Хороший пример: функция `strcpy`. Её настоящая реализация в той же glibc для x86 очень существенно отличается от наивного подхода изложенного в K&R. Обычно каждая функция в стандартной библиотеке для каждой архитектуры это результат многолетнего и кропотливого труда программистов и аналитиков.

2.2 Бинарные утилиты

Многие бинарные утилиты уже были рассмотрены выше, но здесь можно привести полный список бинарных утилит GNU Toolchain (он почти не отличается от такого же списка для LLVM toolchain и имеет некоторые, но не концептуальные отличия от тех утилит, которые доступны под Windows).

Некоторые бинарные утилиты уже рассматривались выше:

- as – ассемблер, рассматривался в (1.3.1)
- objdump – дизассемблер, рассматривался в (1.3.3)
- ld – линкер, рассматривался в (1.4.2)
- readelf – упоминался в (1.4.2)
- ar – упаковщик статических библиотек, упоминался в (1.4.2)

Другие пока не так хорошо знакомы и будут рассмотрены далее

2.2.1 addr2line

Очень часто при запуске плохо отлаженной программы, можно получить segfault, неприятного вида. Синтетический пример:

```
1 int main()
2 {
3   int *p = (int*)0xDEADBEEF;
4   *p = 5; /* boom */
5 }
```

При запуске даёт:

```
$ gcc segf.c
$ ./a.out
Segmentation fault
$ dmesg | tail -1
[ 4051.415509] a.out[3346]: segfault at deadbeef
ip 00000000004004da sp 00007fffdd0a97c0 error 6 in a.out[400000+1000]
```

Если программа большая, может быть не вполне ясно чему соответствует адрес $4004da$. В этом случае можно собрать ту же программу с отладочной информацией и узнать номер строки по адресу:

```
$ gcc -g segf.c
$ addr2line -e a.out 0x00000000004004da
segf.c:4
```

Что сразу даёт информацию о том, что ошибка в четвертой строке и может сделать отладку гораздо проще.

2.2.2 c++filt

Деманглер иногда оказывается крайне полезен. Для того, чтобы сделать возможными функции-члены в структурах, перегрузку функций и пространства имён, C++ искажает имена функций, в результате чего в ассемблере целевой архитекутры они мало читаемы:

```
_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
```

Всегда страшно встретить нечто такое. Но простое использование здесь деманглера позволяет получить (расшифровать?) точное имя этой метки в C++ коде:

```
1 std::basic_ostream<char, std::char_traits<char>>&
2 std::endl<char, std::char_traits<char>> (
3   std::basic_ostream<char, std::char_traits<char>>&)
```

Откуда видно, что это просто `std::endl` – стандартный способ указать конце строки при выводе в C++.

К сожалению, правила манглирования имён не регламентируются никакими стандратными документами, поэтому деманглер одной архитектуры скорее всего не подойдет для любой другой.

2.2.3 elfedit и objcopy

Утилита elfedit используется для редактирования метаданных в ELF-файлах. Во многих отношениях она дополняет readelf, который позволяет только анализировать файлы, возможностями редактирования.

Формат ELF играет громадную роль в GNU Toolchain и в Linux, но его использование за их пределами ограничено, поэтому нет смысла здесь углубляться в детали.

Гораздо более распространена утилита objcopy, которая позволяет делать удивительные вещи: например создавать объектный файл, готовый для линковки с кодом на C практически из любого бинарного мусора

```
$ dd if=/dev/urandom of=blob.bin bs=1 count=16
$ objcopy -I binary -O elf64-x86-64 -B i386 blob.bin blob.o
$ objdump -t blob.o
blob.o:     file format elf64-x86-64
SYMBOL TABLE:
00000000 l    d  .data 00000000 .data
00000000 g       .data 00000000 _binary_blob_bin_start
00000010 g       .data 00000000 _binary_blob_bin_end
00000010 g       *ABS* 00000000 _binary_blob_bin_size
```

Теперь в C коде можно использовать эти метки как нормальные объявления

```
1 extern unsigned char _binary_blob_bin_start;
2 extern unsigned char _binary_blob_bin_end;
3 extern unsigned char _binary_blob_bin_size;
4 /* .... */
5 unsigned char *pblob = &_binary_blob_bin_start;
```

Разумеется мусор тут взят только для примера. Линковка в программу любых ресурсов, в том числе самых причудливых, средствами objcopy не сложнее.

2.2.4 nm и objdump

Утилита nm является упрощением objdump для posix-подсистем и всё, что умеет nm можно сделать через objdump. До сих пор objdump рассматривался только как дизассемблер, но эта пара умеет гораздо больше. Так например стандартная фича nm (она же используется через objdump -t) это просмотр таблицы имён.

```
$ nm test.o
                 U fact
```

```
0000000000000000 T main
                 U printf
```

Здесь видны два неопределенных имени (fact и printf). Objdump покажет больше информации:

```
$ objdump -t test.o

test.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .text 0000000000000000 .text
0000000000000000 l    d  .data 0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .rodata 0000000000000000 .rodata
....
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g     F .text 0000000000000026 main
0000000000000000         *UND* 0000000000000000 fact
0000000000000000         *UND* 0000000000000000 printf
```

Видны также все секции, каждую из которых можно посмотреть отдельно.

2.2.5 strip

Полезная утилита strip используется для вырезания символьной и отладочной информации и оптимизации размера файла:

```
$ du -b fact.o
1232 fact.o
$ strip fact.o -o fact-s.o
$ du -b fact-s.o
832 fact-s.o
```

Как видно для простого объектного файла, содержащего только функцию факториал, его размер был уменьшен в полтора раза. Увы, цена этого уменьшения велика. Например теперь оттуда ничего нельзя прчоитать командой nm:

```
$ nm fact-s.o
nm: fact-s.o: no symbols
```

Поэтому имеет смысл использовать strip с осторожностью только для уже отлаженного и гарантированно работающего кода.

2.3 Отладчики и отладка

Важным этапом жизненного цикла программы является её отладка, когда возможен контроль её состояния в пошаговом режиме, включая как низкоуровневый контроль (считать регистры, память, исполнить один ассемблерный шаг), так и довольно высокоуровневый, включающий построчное выполнение кода и просмотр состояния переменных.

Большинство современных процессоров поддерживают возможности для отладки для аппаратном уровне, например возможность пошагового исполнения часто реализуется как trap flag, аппаратная поддержка точек останова и наблюдения через специальные прерывания, поддержка специальных отладочных протоколов самого процессора, таких так JTAG, etc.

2.3.1 Обзор отладчиков

Отладчики делятся по уровню работы на:

- Отладчики микроархитектуры, такие как OpenOCD (on-chip debugger), способные работать непосредственно с процессором, с полным доступом к его системным регистрам и памяти, исполняются обычно вне отлаживаемого устройства, подключаясь к нему по протоколу, такому как JTAG
- Отладчики уровня ядра, такие как KGDB или WinDbg, позволяют отлаживать операционную систему: имеют доступ к системным регистрам и физической памяти, исполняются как привилегированный код, но чаще уже на самом устройстве.
- Отладчики пользовательского уровня, такие как GDB, LLDB, IDB, MSVS Debugger и многие другие: исполняются с правами обычного пользователя и имеют непривилегированный доступ к виртуальной памяти отлаживаемого процесса.

Отладчики первых двух типов бывают очень интересны, но обычно обладают урезанными возможностями и являются инструментами профессионалов со всеми присущими ограничениями. Далее речь пойдет в основном про GDB и более простую отладку уровня пользователя.

2.3.2 Работа с GDB

Проще всего научится отлаживать программы отлаживая программы.

Пусть дан на первый взгляд обычный вариант пузырьковой сортировки.

```
1  /* Simple but buggy bubble sort *
2   * Can you find the bugs?     */
3  void
4  sort(item *a, int n)
5  {
6    int i = 0, j = 0;
7    int s;
8
9    for (; i < n && s != 0; i++)
10     {
11       s = 0;
12       for (j = 0; j < n; j++)
13         if (a[j].key > a[j+1].key)
14           {
15             item t = a[j];
16             a[j] = a[j+1];
17             a[j+1] = t;
18             s++;
19           }
20       n--;
21     }
22 }
```

Сколько проблем в таком коде можно найти пристально вглядываясь в него? Вряд ли много. Чтобы начать отладку, программу следует откомпилировать с отладочной информацией и запустить под отладчиком:

```
$ gcc buggy-sort.c -g -o buggy
```

Здесь хорошо выучить несколько наиболее распространенных команд gdb:

```
help - list commands
run - start your program [with arglist if needed]
```

```
list - show lines of source
break - set a break point in your code
next - execute next line, stepping over function calls
step - execute next line, stepping into function calls
print - print the contents of variables and data structures
frame - select frame number, point to the stack you are interested in
backtrace - display program stack and control flow
command - execute GDB command-list every time breakpoint n is reached
cont - continue the execution of the code
quit - quit the debugger
```

Далее лучше делать с проектором вместе со студентами:

```
(gdb) break sort
Breakpoint 1 at 0x40056f: file buggy-sort.c, line 23.
(gdb) run
Breakpoint 1, sort (a=0x601040, n=5) at buggy-sort.c:23
23   int i = 0, j = 0;
(gdb) next
26   for (; i < n && s != 0; i++)
(gdb) print s
$1 = 0
```

Видно, что переменная s должна быть инициализирована во что-то ненулевое. Но после инициализации переменной s, программа сортирует явно не слишком хорошо.

Тут надо в аудитории продолжить разбор и найти все ошибки. Читающему это как текстовый файл это может быть оставлено как домашнее задание.

2.3.3 Отладочная информация

Отладочная информация, которая позволяет делать все вышеперечисленное.

Существует много форматов отладочной информации:

- OMF – ранее популярный, но устаревший формат для ранних DOS-подобных систем

- DWARF – де-факто стандарт отладочной информации в Unix-подобных системах
- PE/COFF – де-факто стандарт отладочной информации в Windows-подобных системах

Далее будет рассматриваться формат DWARF, названный так в пару к формату ELF исполняемых файлов.

Часть отладочной информации можно видеть в ассемблере:

```
fact:
.LFB0:
  .file 1 "fact.c"
  .loc 1 3 0
  .cfi_startproc
  pushq %rbp
.LCFI0:
  .cfi_def_cfa_offset 16
  movq  %rsp, %rbp
  .cfi_offset 6, -16
.LCFI1:
  .cfi_def_cfa_register 6
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  .loc 1 4 0
  cmpl  $1, -4(%rbp)
  ja  .L2
  .loc 1 5 0
  movl  $1, %eax
  jmp .L3
```

Здесь специальные метки loc отмечают границы строк кода.

```
1 unsigned
2 fact (unsigned x)
3 {
4   if (x < 2)
5     return 1;
6
7   return x * fact (x-8, 1);
8 }
```

Можно обратить внимание, что строка 3 это всего лишь открывающая фигурная скобка, но как много всего происходит между ней и строчкой 4!

Этот фрагмент кода называется **пролог** и можно ещё раз его повторить ниже:

```
  .loc 1 3 0
  .cfi_startproc
  pushq %rbp
.LCFI0:
  .cfi_def_cfa_offset 16
  movq  %rsp, %rbp
  .cfi_offset 6, -16
.LCFI1:
  .cfi_def_cfa_register 6
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  .loc 1 4 0
```

В прологе происходит:

- Сохранение фрейм пойнтера (в данном случае регистр rbp) в стек.
- Заполнение фрейм пойнтера стек пойнтером для того, чтобы отметить начало текущего фрейма в стеке.
- Сдвиг вниз стек пойнтера чтобы дать место для локальных переменных

Отладочная информация зашифрованная в CFI (call frame information) метках, легко считывается визуально:

- **cfi_startproc**

  Обозначает начало текущей функции, точку входа, на которой остановится отладчик.

- **cfi_def_cfa_offset**

  Устанавливает смещение CFA (canonical frame address) в это функции, отмечая место в котором на стеке сохранено значение указателя стека вызывающей процедуры.

- **cfi_offset**

  Устанавливает, что предыдущее значение регистра 6 (то есть rsp) сохранено на указанном в offset (второй параметр) расстоянии от CFA.

- **cfi_def_cfa_register**

  Модифицирует правило для вычисления CFA. Начиная от этой директивы для этого используется указанный как параметр регистр

Эта отладочная информация зависит от архитектуры по сути но не по формату. Тот же пролог для ARM, а не для x86 выглядит так:

```
.loc 1 3 0
.cfi_startproc
stmfd sp!, {fp, lr}
.cfi_def_cfa_offset 8
.cfi_offset 11, -8
.cfi_offset 14, -4
add fp, sp, #4
.cfi_def_cfa 11, 4
sub sp, sp, #8
str r0, [fp, #-8]
.loc 1 4 0
```

По сути он совсем другой, поскольку в мире ARM действуют иные соглашения о вызове и раскладке стека, но все CFI метки все так же отлично считываемы и ясны.

Кроме того, много отладочной информации зашифровано в специальных секциях. Её можно исследовать с помощью **objdump** с опцией dwarf и указанием того, какую именно отладлочную информацию хочется получить.

Благодаря сохранению в отладочной информации определенной избыточности, можно откомпилировать файл с отладочной информации, а потом дизассемблировать, аннторировав ассемблер строчками исходного кода

```
gcc -c -g -O0 fact.c
objdump -gdS fact.o
```

Выдает результат

```
unsigned
fact (unsigned x)
{
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 83 ec 10           sub    $0x10,%rsp
   8: 89 7d fc              mov    %edi,-0x4(%rbp)
  if (x < 2)
   b: 83 7d fc 01           cmpl   $0x1,-0x4(%rbp)
   f: 77 07                 ja     18 <fact+0x18>
    return 1;
  11: b8 01 00 00 00        mov    $0x1,%eax
  16: eb 11                 jmp    29 <fact+0x29>

  return x * fact (x-1);
  18: 8b 45 fc              mov    -0x4(%rbp),%eax
  1b: 83 e8 01              sub    $0x1,%eax
  1e: 89 c7                 mov    %eax,%edi
  20: e8 00 00 00 00        callq  25 <fact+0x25>
  25: 0f af 45 fc           imul   -0x4(%rbp),%eax
}
  29: c9                    leaveq
  2a: c3                    retq
```

Отладочная информация может использоваться не только отладчиком: профилировщики и анализаторы кода так же любят возможность показать на строчку исходного кода содержащую ошибку.

2.4 Профилировка и анализ кода

Стандартными средствами анализа программ, часто упоминающимися в составе GNU Toolchain (если понимать слово тулчейн в широком смысле) являются профилировщик gprof, анализатор покрытия тестами gcov и тот или иной санитайзер памяти (иногда использую valgrind, иногда один из встроенных санитайзеров в компиляторе). В этом разделе будут рассмотрены все три вида утилит.

2.4.1 Инструментирование кода

Прежде чем профилировать код, его следует инструментировать. Для этого в gcc используется специальная опция `-pg`. Например, можно откомпилировать с этой опцией факториал, чтобы посмотреть на то, как изменится код:

```
$ gcc -pg fact.c -S -O2
```

Теперь при анализе получившегося ассемблера визуально заметно отличие:

```
fact:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  movq  %rsp, %rbp
  .cfi_offset 6, -16
  .cfi_def_cfa_register 6
  call  mcount
  movl  $1, %eax
  cmpl  $1, %edi
```

Добавился вызов некоей странной функции `mcount`. Это и есть инструментировка кода. В процессе работы, инкрементируется значение счетчиков при каждом вызове каждой функции, благодаря чему профилировщик имеет исчерпывающую информацию. Разумеется, инструментирование ассемблера это просто развлечение, для настоящей профилировки, инструментировать нужно бинарный файл и лучше выбрать для этого приложение посложнее, чем факториал.

2.4.2 Бенчмарки

Бенчмарком (англицизм от benchmark) или тестом производительности называется контрольная задача, необходимая для определения сравнительных характеристик производительности компьютерной системы. Обычно это некоторые "усредненные" компьютерные программы, позволяющие проверить насколько хорошо микроархитектура справляется с циклами, переходами и всем таким.

Популярными open-source бенчмарками являются:

- Dhrystone – производительность целочисленной арифметики
- Whetstone – производительность арифметики с плавающей точкой
- Coremark – пакет тестов производительности, покрывающих недостатки Dhrystone
- NAS parallel benchmarks – производительность параллельных вычислений

Наиболее известным платным пакетом являются SPEC benchmarks, широко использующиеся для официальных замеров относительной производительности микропроцессоров.

Разумеется существуют бенчмарки не только на производительность архитекутры, но и на браузеры, операционные системы, компиляторы и прочее.

Дальнейшее исследование средств профилировки будет проводится на примере dhrystone на x86 под linux с использованием запуска для 50 миллионов циклов.

2.4.3 Профилировка и чтение профиля

После скачивания исходников dhrystone (откуда угодно из интернета), необходимо собрать их с отключенным инлайном, и инструментировать:

```
$ gcc -fno-inline -O dhry.c -pg
$ ./dhry
```

После прогона инструментированного кода остаётся файл gmon.out Далее можно запустить gprof с этим результирующим файлом и исходным бинарником для анализа кода:

```
$ gprof dhry gmon.out > analysis.txt
```

Получившаяся в итоге таблица выдаёт тотальную раскладку по времени, проведенному в каждой из процедур:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 23.34      0.53     0.53        1     0.53     2.28  Proc0
 17.61      0.93     0.40 50000000     0.00     0.00  Proc1
 15.63      1.29     0.36 50000000     0.00     0.00  Proc8
  9.03      1.50     0.21 150000000     0.00     0.00  Func1
  7.05      1.66     0.16 50000000     0.00     0.00  Func2
  6.61      1.81     0.15 50000000     0.00     0.00  Proc6
  5.28      1.93     0.12 50000000     0.00     0.00  Proc3
  4.62      2.03     0.11 150000000     0.00     0.00  Proc7
  3.96      2.12     0.09 50000000     0.00     0.00  Proc2
  2.86      2.19     0.07 50000000     0.00     0.00  Proc4
  2.20      2.24     0.05 50000000     0.00     0.00  Proc5
  1.98      2.28     0.05 50000000     0.00     0.00  Func3
  0.22      2.29     0.01                             frame_dummy
```

Также в выходном файле имеется call graph программы, позволяющий определить кто кого откуда вызывал.

2.4.4 Покрытие кода

Другая полезная утилита нужна для работы во время отладки для проверки того, что при тестировании программа зашла в каждую возможную ветку исполнения. Такой анализ называется анализом покрытия.

Код (на этот раз отладочный) снова нужно инструментировать, на этот раз с использованием опций profile-arcs и test-coverage. Первая опция приводит к сохранению статистики исполнения строк исходного файла, а вторая — к записи статистики условных переходов (ветвлений). Сборка проходит без оптимизаций (которые в этом случае будут только мешать) и с включением отладочной информации:

```
$ gcc -fprofile-arcs -ftest-coverage fact.c -S -O0 -g
```

В ассемблере можно видеть строчку:

```
  call  __gcov_init
```

и некоторые счетчики в глобальных переменных, но все функции не инструментируются.

После получения бинарного файла, он запускается на исполнение и генерирует несколько файлов со стандартными именами в папке, откуда был запущен:

```
$ gcc -fprofile-arcs -ftest-coverage fact.c main.c -O0 -g -o fact
$ ./fact
```

В данном случае будут сгенерированы fact.gcda, fact.gcno, main.gcda, main.gcno. Далее они используются утилитой gcov для анализа:

```
$ gcov fact.c
File 'fact.c'
Lines executed:100.00% of 4
fact.c:creating 'fact.c.gcov'
```

Теперь строчки программы аннторированы числом исполнений каждой из них.

```
        -:    1:unsigned
        5:    2:fact (unsigned x)
        -:    3:{
        5:    4:  if (x < 2)
        1:    5:    return 1;
        -:    6:
        4:    7:  return x * fact (x-1);
        -:    8:}
```

Инструментирование, производимое gcov гораздо мягче, чем gprof, потому что счетчики вставляются даже не на всех дугах, а только в тех местах, где они реально нужны.

```
+ static ulong counts[numbbs];
+ static struct bbobj =
+       { numbbs, &counts‘‘,file1.’’c};
+ static void _GLOBAL_.I.fooBarGCOV()
+       { __bb_init_func(&bbobj); }

 void fooBar (void)
 {
+  counts[i]++;
   <bb-i>
   if (condition) {
+    counts[j]++;
     <bb-j>
   } else {
     <bb-k>
   }
 }
```

Здесь знаками + отмечено возможное инструментирование условной функции fooBar, содержащей некоторый нетривиальный код внутри линейных участ bb-i, bb-j и bb-k.

2.4.5 Проверки времени исполнения

Очень часто самое пристальное вчитывание в код не позвоялет исключить возможных случаев неопределенного поведения:

```
int foo(int* a, int len)
{
  assert ((a != NULL) && (len > 1));
  return a[len/2];  /* ORLY? */
}
```

В этом куске кода никак нельзя прикрыться от ситуации выхода за границы массива.

Некоторые компиляторы (такие как GCC) поддерживают инструментирование кода в местах, где возможно неопределенное поведение. Для GCC доступен вызов с опцией `-fsanitize=undefined`, которая включает UndefinedBehaviorSanitizer – open-source средство определения неопределенного поведения во время исполнения. Он проверяет почти все случаи возможного UB, но существенно замедляет программу, так как по сути

генерирует вызов проверяющей функции на каждое знаковое суммирование, доступ по указателю, сдвиг и прочее.

Для того, чтобы проверить конкретный случай UB а не все их сразу, можно подать вместо undefined, одну из её подопций:

```
-fsanitize=shift
-fsanitize=integer-divide-by-zero
-fsanitize=unreachable
-fsanitize=vla-bound
-fsanitize=null
-fsanitize=return
-fsanitize=signed-integer-overflow
-fsanitize=bounds
-fsanitize=bounds-strict
-fsanitize=alignment
-fsanitize=object-size
-fsanitize=float-divide-by-zero
-fsanitize=float-cast-overflow
-fsanitize=nonnull-attribute
-fsanitize=returns-nonnull-attribute
-fsanitize=bool
-fsanitize=enum
-fsanitize=vptr
```

Сама библиотека UBsan, поддерживающая всю эту функциональность перекочевала в GCC из LLVM/Clang поэтому доступна и там.

Очень полезно прогонять программы под санитайзерами на предмет поиска сложных случаев UB которые не всегда легко проверить глазами.

Ещё полезнее использовать специальные средства, такие как valgrind, который позволяет кроме обычных случаев UB ловить, например, такие коварные вещи как data races в многопоточных программах:

```
#include <thread>
#include <iostream>

int x = 0;

void
foo ()
{
  std::thread([&] {
```

```
10      ++x;
11    }).detach();
12    ++x;
13  }
14
15  int
16  main ()
17  {
18    foo ();
19    std::cout << x << std::endl;
20    return 0;
21  }
```

Можно скомпилировать, запустить и не заметить подвоха:

```
$ g++ race.cpp -pthread
$ ./a.out
1
```

Но valgrind исправно отрапортует о возможной проблеме:

```
$ valgrind --tool=helgrind ./a.out
==4257== Possible data race during write
         of size 4 at 0x602548 by thread #1
==4257==    at 0x400DEF: foo()
==4257==    by 0x400E20: main
==4257==  This conflicts with a previous write
         of size 4 by thread #2
```

Различные tools, которые можно указывать для valgrind включают в себя:

- Memcheck – анализирует утечки памяти, неинициализированные переменные, выход за границы массивов
- Cachegrind – профилировщик кэша, указывает источники промахов мимо кэша в вашем коде (что может быть важно для критчиных по производительности частей)
- Callgrind – расширение cachegrind с дополнительным профилем вызовов функций (плюс позволяет их отличную визуализацию с помощью KCachegrind)

- Massif – профилировщик кучи, способен составлять график использования кучи от времени исполнения, находить места наиболее частых выделений памяти
- Helgrind – иллюстрировавшийся выше анализатор многопоточного кода

Но они этим не ограничиваются, на самом деле valgrind это просто океан.

3 Портирование toolchain

При портировании системы компиляции основная работа это портирование компилятора и бинарных утилит.

Портирование отладчика обычно проще, так как там многое построено на платформенно-независимой DWARF информации, а профилировщик часто достается вообще бесплатно, если компилятор верно аннотировал функции счетчиками для профилировки. Ещё проще поддержка в библиотеках, она часто напрямую завязана на портирование операционной системы и конкретные реализации системных вызовов в аритектуре. Конечно для совсем embedded без нормальных системных вызовов эта часть может быть не вполне тривиальной, но её сложность сравнима со сложностью портирования отладчика.

В портировании бинарных утилит в свою очередь самое сложное это прописывание особенностей форматов бинарных команд для архитектуры, мест, где нужны релокации, типов этих релокаций и так далее. Это иногда сложная, но, в общем, техническая работа.

Основная творческая часть портирования toolchain это портирование компилятора.

Порядок портирования компилятора может быть примерно следующим:

- Определить набор инструкций
- Определить ограничения на параметры
- Написать эффективный перевод из промежуточного в машинно-зависимое представление
- Разработать и реализовать ABI (включая конвенции вызовов и выравнивание)
- Определить простые преобразования инструкций

3.1 GCC machine description

Портирование компилятора требует некоторого навыка работы с его структурой папок. Внутри исходников GCC самый верхний уровень занимают папки основных утилит и библиотек, включая папки собственно для gcc, libgcc, libstdc++ и прочих:

```
> ls -d */
boehm-gc/
config/
contrib/
fixincludes/
gcc/
gnattools/
....
```

Второй уровень (внутри папки gcc первого уровня) занимают файлы компилятора и подпапки фронтендов (таких как Ada, C и Fortran) а также специальная папка config:

```
gcc> ls -ld */
ada/
c/
c-family/
common/
config/
cp/
doc/
....
```

И наконец в самой папке config (третий уровень) расположены бэкенды для разных архитектур.

```
gcc/config> ls -ld */
aarch64/
alpha/
arc/
arm/
avr/
bfin/
```

Обычная задача портирования это задача добавления такой папки для своего бэкенда. Для примеров в этом разделе будет использована папка mips, содержащая бэкенд для mips. Основные его файлы: mips.h, mips.c и mips.md

3.1.1 Описание инструкций

Простой паттерн для инструкции add архитектуры mips сначала может выглядеть пугающе:

```
(define_insn "*addsi3_extended"
  [(set (match_operand:DI 0 "register_operand" "=d,d")
        (sign_extend:DI
             (plus:SI (match_operand:SI 1 "register_operand" "d,d")
                      (match_operand:SI 2 "arith_operand" "d,Q"))))]
  "TARGET_64BIT && !TARGET_MIPS16"
  "@
    addu\t%0,%1,%2
    addiu\t%0,%1,%2"
  [(set_attr "alu_type" "add")
   (set_attr "mode" "SI")])
```

Он написан в скобочной нотации, похожей на язык LISP и состоит из следующих основных частей:

- Названия и заголовка

  ```
  (define_insn "*addsi3_extended"
  ....
  )
  ```

- Собственно паттерна

  ```
  ....
  [(set (match_operand:DI 0 "register_operand" "=d,d")
        (sign_extend:DI
             (plus:SI (match_operand:SI 1 "register_operand" "d,d")
                      (match_operand:SI 2 "arith_operand" "d,Q"))))]
  ....
  ```

Это тот самый паттерн, который порождает машинно-зависимую инструкцию, которая потом попадает в дампы RTL, которые уже разбирались выше (см. 1.2.5). Здесь описано сложение со знаковым расширением.

- Условия применимости

```
....
  "TARGET_64BIT && !TARGET_MIPS16"
....
```

- Строчек генерируемого ассемблера

```
....
  "@
    addu\t%0,%1,%2
    addiu\t%0,%1,%2"
....
```

- Атрибутов

```
....
  [(set_attr "alu_type" "add")
   (set_attr "mode" "SI")]
```

3.1.2 Предикаты и констрейнты

В собственно паттерне можно различить предикаты: register_operand, arith_operand и констрейнты, записанные сразу справа от предикатов строчками букв (в данном случае d и Q).

register_operand это стандартный предикат, означающий, что при переводе в машинно-зависимое представление там должен встретиться регистр.

arith_operand это пользовательский предикат, определенный в predicates.md

```
(define_predicate "const_arith_operand"
  (and (match_code "const_int")
       (match_test "SMALL_OPERAND (INTVAL (op))")))
```

```
(define_predicate "arith_operand"
  (ior (match_operand 0 "const_arith_operand")
       (match_operand 0 "register_operand")))
```

Здесь макрос определен в mips.h:

```
#define SMALL_OPERAND(VALUE) \
  ((unsigned HOST_WIDE_INT) (VALUE) + 0x8000 < 0x10000)
```

В отличии от предикатов, которые используются для поддержания корректности паттерна в процессе трансформаций, констрейнты являются указаниями распределителю регистров и так же бывают стандартные (например r означает регистр) и пользовательские, как d и Q в данном случае.

Констрейнт d является регистровым констрейнтом и определен в constraints.md

```
(define_register_constraint "d" "TARGET_MIPS16 ? M16_REGS : GR_REGS"
  "An address register.  This is equivalent to @code{r} unless
   generating MIPS16 code.")
```

Констрейнт Q определен там же так, чтобы соответствовать предикату

```
(define_constraint "Q"
  "@internal"
  (match_operand 0 "const_arith_operand"))
```

3.1.3 Перевод в машинно-зависимое представление

В паттернах, переводящих в машинно-зависимое представление критчино важно их имя, так как именно оно матчится на этапе экспанда.

```
(define_expand "add<mode>3"
  [(set (match_operand:GPR 0 "register_operand")
        (plus:GPR (match_operand:GPR 1 "register_operand")
                  (match_operand:GPR 2 "arith_operand")))]
  "")
```

Итератор mode является стандартным итератором, в то время как составной режим GPR определен в том же файле чуть выше как:

```
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
```

Это позволяет при обработке такого паттерна сгенерировать из него сразу два: для 32-битного и 64-битного сложения.

3.1.4 Преобразования инструкций

```
(define_split
  [(set (match_operand:DI 0 "register_operand")
        (and:DI (match_operand:DI 1 "register_operand")
                (const_int 4294967295)))]
  "TARGET_64BIT && !ISA_HAS_EXT_INS && reload_completed"
  [(set (match_dup 0)
        (ashift:DI (match_dup 1) (const_int 32)))
   (set (match_dup 0)
        (lshiftrt:DI (match_dup 0) (const_int 32)))])
```

TODO: больше про split

3.2 Calling conventions

При вызове функции существует много вариантов передачи параметров (для параметров можно использовать какие-то из регистров и стек) а так же много вариантов возврата значений из функции.

На (рис. 13) изображен пример передачи параметров в некоей условной архитектуре.

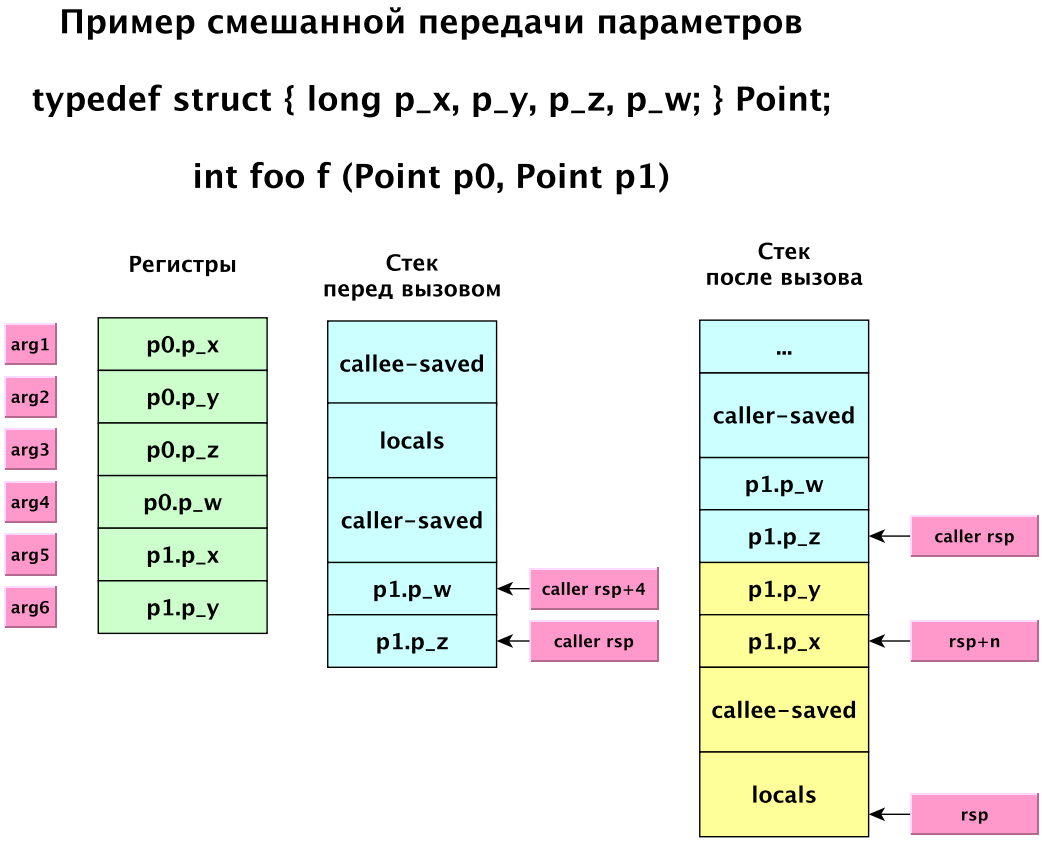

Рис. 13: Пример передачи параметров

TODO: здесь какой-нибудь разбор этой картинки

Некоторые традиционные конвенции вызова имеют свои устоявшиеся имена:

- cdecl – через стек, справа налево, обратный сдвиг – caller
- pascal – через стек, слева направо, сдвиг – callee
- fastcall – на регистрах, сдвиг – callee

- stdcall – через стек, справа налево, сдвиг – callee
- tailcall – вызов непосредственно перед возвратом, можно не двигать стек