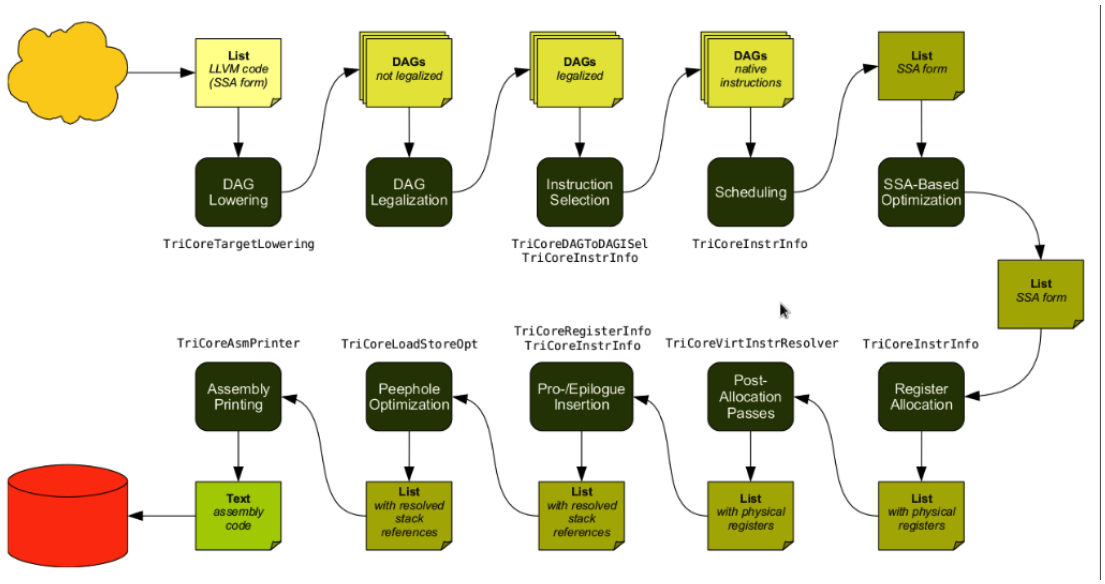
Для того, чтобы понять устройство и происхождение стековых и регистровых процессоров, придётся окунуться во внутренний мир компиляторов. Делать это будем очень упрощенно, просто для понимания.

Компиляция.

Задача компилятора - преобразовать представленную в текстовом виде программу на некотором языке программирования в исполняемый код для заданной архитектуры. Компиляторы прошли длительный путь развития и в современном виде весьма сложно устроены. К счастью, сложные задачи разбиваются на более простые подзадачи, более доступные для понимания. Чем мы и займёмся.



Фиг.1 стадии компиляции (LLVM) (отсюда).

Лексический анализ.
Его задача - превратить текстовый файл в поток так называемых лексем. Лексема в лингвистике - слово, элементарная часть естественного языка. В языках программирования - совокупность информации, описывающей элементарные объекты этого языка. Лексемой может быть идентификатор ключевого слова этого языка (BEGIN, END, …). Или литерал - строчная или числовая константа. Или оператор языка (+, -, *, /, …). Или описатель переменной, включающий её имя.

Итак, лексический анализатор:
- просматривает входной текст, находит последовательность символов, определяющих лексему
- получает всю информацию о лексеме
- сдвигает текущее положение во входном потоке
- возвращает тип лексемы (ключевое слово, литерал, оператор, переменная, …) и сопутствующую информацию, если есть (значение для литерала, имя переменной, ...)

Самый низкий уровень работы компилятора - вызывать лексический анализатор до тех пор, пока не закончится входной текст.

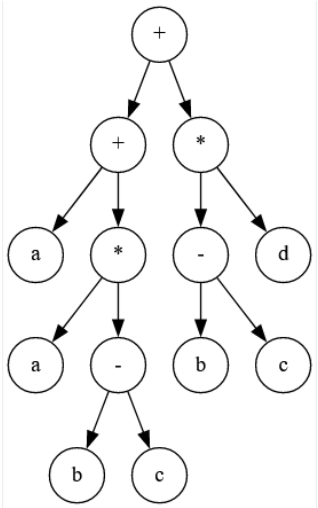
Синтаксический анализ.

Языки программирования похожи на естественные, они также состоят из высказываний (предложений). Типичное предложение естественного языка состоит из подлежащего, сказуемого и вспомогательных слов. Высказывания языков программирования более строги, их состав определяется строгим набором правил, так называемой грамматики языка. Например,

выражение это:
число
или *переменная*
или *переменная = выражение*
или *выражение + выражение*
или *выражение - выражение*
или *выражение * выражение*
или *выражение / выражение*
или *- выражение*
или *‘(‘ выражение ‘)’*

Предположим в нашем языке программирования каждая строка текста содержит одно выражение. Как работает компилятор ? Синтаксический анализатор это машина с набором состояний (автомат). Он вызывает лексический анализатор и получает первую считанную лексему. Разберём на примере.
Пусть разбираемое выражение “*a+a*(b-c)+(b-c)*d*”
Изначальное состояние - *выражение*.
Первая лексема - переменная a, это соответствует состоянию выражение.
Вторая лексема - оператор +, это соответствует правилу *выражение + выражение.*
Третья лексема - опять переменная a
Оператор умножения - …
Скобка ‘(‘, согласно грамматике ожидаем дальше выражение, после чего закрывающую скобку.
…

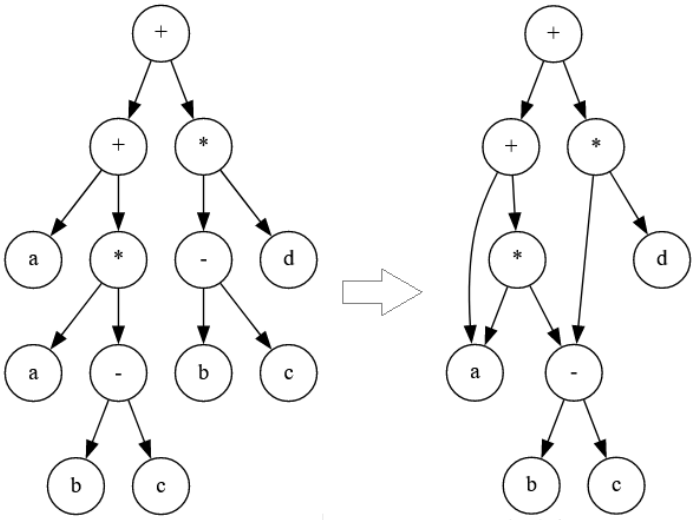
В процессе работы компилятор переходит из состояния в состояние, в процессе чего строит дерево синтаксического разбора (AST, Abstract Syntax Tree), которое можно рассматривать как историю переходов между состояниями. Например, для примера выше оно будет таким:



Фиг.2 Дерево синтаксического разбора (отсюда)

Упрощение дерева разбора.

Иногда, дерево разбора можно упростить, например, исключая дублирующиеся ветви. Так, вышеприведенное дерево превращается из дерева в ориентированный ациклический граф (DAG, Directed Acyclic Graph)



Фиг.3 Построение DAG. (отсюда)

Генерация промежуточного кода.

Наличия DAG уже достаточно для того, чтобы начать генерировать исполняемый код. Во времена, когда под каждую архитектуру писался свой компилятор, так и было. Однако, создание компилятора - дело непростое, отнимающее массу времени и сил. Тем более что в каждом компиляторе есть и общая для всех компиляторов языка часть и специфичная для целевой архитектуры.

Общая практика - генерация универсального промежуточного кода и его оптимизация. Далее следует преобразование промежуточного кода в целевой и окончательная оптимизация целевого кода (очень грубо).

Универсальный промежуточный код трёхадресный, а вот что это такое и почему именно трёхадресный, заслуживает отдельного внимания.

Почему промежуточный код трёхадресный.

Первые компьютеры были устроены весьма примитивно, в то время было не до архитектурных изысков и оптимизаций, достаточно того, что всё это хоть как-то работало. В аналитической машине Бэббиджа данные хранились в хранилище (сам он это называл склад, store). Чтобы произвести вычисления, данные со склада переносились в арифметическое устройство (мельница, mill). После выполнения арифметической операции данные можно было выгрузить обратно на склад. В современных терминах, это больше всего похоже на архитектуру с регистром - аккумулятором. Такая архитектура (с одноадресными инструкциями, т.е. в инструкции может быть не более одного номера регистра) не отличается скоростью работы, но весьма проста в реализации. Некоторые современные микроконтроллеры (PIC, 8051) имеют подобное устройство.

Регистр - место хранения информации (слова), которая может быть немедленно использована в вычислениях. Эти он отличается от кэш-памяти, которую для использования требуется скопировать в регистр и потратить на это пару тактов. Даже если физически они устроены одинаково и работают на одной частоте.

Неудивительно, что развитие пошло в сторону увеличения числа регистров, в пределах целесообразности, конечно. Помимо всего прочего, это еще и удобно. Для первых компьютеров не существовало ни языков программирования, ни компиляторов, программы писались вручную. И иметь под рукой небольшое количество регистров интуитивно понятно и очень удобно. Мы уже разбирали эту ситуацию в главе CISC против RISC. Все подобные архитектуры можно объединить под общим названием *регистровые машины*.

Системы команд таких компьютеров состояли из одно-, двух-, а позднее и трёхадресных инструкций (RISC). Например, трёхадресная инструкция сложения для архитектуры MIPS выглядит так:

add $t2, $t0, $t1

где *$t0* и *$t1* - регистры операнды, а *$t2* - регистр куда будет помещен результат. Для архитектуры x86 это выглядит немного иначе

add eax,dword ptr [u]

здесь складывается содержимое регистра *eax* со значением (в данном варианте инструкции) из памяти (переменная u), результат помещается в *eax*.

Четырёх-адресных инструкций не бывает. Операции имеют не более двух аргументов и результат. Единственно, где теоретически могло бы пригодиться более трёх адресов регистров в инструкции - целочисленное деление т.к. оно порождает два результата -

частное и остаток от деления. Но разработчики процессоров предпочитают помещать результаты целочисленного деления в специально отведенные для этого места, например, инструкция MIPS

```
div     $s0, $s1
```

помещает частное и остаток в специальные регистры, соответственно, *lo* и *hi.*
В x86 инструкции DIV и IDIV(знаковая) помещают результат в регистры AX (частное) и DX (остаток).

Таким образом можно считать, что трёхадресный код перекрывает возможности любой архитектуры. А, следовательно, универсальный трёхадресный код можно преобразовать без потерь в сколько-угодно-адресный код любой целевой регистровой архитектуры.

Да, но каково число регистров в архитектуре универсального кода? Бесконечное количество, вот сколько нужно, столько и регистров. Можно каждому узлу DAG присвоить свой регистр, почему нет?

Распределение регистров (registers allocation).

Роспись (распределение) регистров происходит при конвертации промежуточного кода в код целевой архитектуры. В целевой архитектуре количество регистров ограничено, кроме того, на них иногда наложены ограничения, например, выше мы видели как инструкция x86 DIV выдаёт результат сразу в два регистра.

Задача в том, чтобы использовать регистры максимально эффективно. Для архитектур с единственным регистром - аккумулятором всё просто - аргументы инструкции должны быть загружены из памяти, результат выгружен обратно в память. Но это слишком дорого, шина памяти неизбежно становится узким местом в производительности системы.

Если в процессоре несколько регистров, возникает возможность часть данных хранить в регистрах и выгружать в память только при необходимости. Однако, число возможных комбинаций с размером DAG-а растёт экспоненциально - это означает что перебрать все варианты для сколь-нибудь нетривиальной задачи становится невозможно (NP-полная задача).

В главе 2.2. CISC vs RISC уже упоминался алгоритм Грегори Хайтина (Gregory Chaitin). Строго говоря, это даже не алгоритм, а эвристика - набор приёмов, которые позволяют найти приемлемое по стоимости решение задачи, честное решение которой слишком дорого. Хайтин свёл распределение регистров к задаче раскраски графа в N цветов. Про эту задачу известно, что она тоже NP-полная, но эвристика быстро даёт решение не сильно хуже оптимального.

Оптимизация, кодогенерация.

После того, как код для целевой архитектуры сформирован, остаётся еще возможность для оптимизации этого кода с учетом особенностей этой архитектуры. Например:

- пул регистров тоже имеет ограничения. Предположим допускается два одновременных чтения и одна запись за такт. Поэтому есть смысл по возможности так размещать инструкции, чтобы не возникали заведомые простои (глава 2.3. суперскалярные или VLIW архитектуры). Для этого используется перестановка взаимозависимых инструкций.
- код находится в памяти и подвержен кэшированию, есть смысл так располагать его в памяти, чтобы он занимал минимальное число линий кэша
- перестановками независимых инструкций можно добиться более оптимальной загрузки конвейеров функциональных устройств (сумматоров, умножителей, делителей, памяти, …)
- так называемая peephole оптимизация, когда вместо некоторых последовательностей инструкций подставляются другие, которые делают то же самое, но более эффективно. Например, вместо целочисленного умножения на 8 можно сдвинуть число влево на три разряда.
- …

“Блеск и нищета” суперскалярных архитектур.

Набор регистров и операции с ними образуют архитектуру процессора и в основном сложились еще в то время, когда суперскалярных процессоров не существовало, а именами регистров обозначали регистры вполне физические.

Концепция регистров в суперскалярных процессорах претерпела существенные изменения. Ведь этих регистров в реальности даже не существует, это фикция, которую использует суперскалярный процессор для обозначения связи между инструкциями.

Архитектура сейчас - это скорее интерфейс между компилятором и процессором. Компилятор не знает как на самом деле устроен процессор, и может выдавать код для целой кучи совместимых устройств. А разработчики процессоров могут сосредоточиться на полезных и важных делах, создавая внешне совместимые изделия. И то и другое - великолепно.

Но какова плата за подобную унификацию?

1. С точки зрения суперскалярного процессора. Вот, например, статья от Intel, дословно: “*Суперскаляр на ходу распараллеливает последовательный код. Но этот процесс распараллеливания слишком трудоемок даже для нынешних процессорных мощностей, и именно он ограничивает производительность машины. Делать это преобразование быстрее определенного количества команд за такт невозможно. Можно сделать больше, но при этом упадет тактовая частота – такой подход, очевидно, бессмыслен. Разумный предел количества команд в настоящее время изучен со всех сторон и пересмотру не подлежит.*”. Вот еще из любимого: “*Беда не в самом суперскаляре, а в представлении программ. Программы представлены последовательно, и их нужно во время исполнения преобразовывать в параллельное выполнение. Главная проблема суперскаляра – в неприспособленности входного кода к его нуждам.*

Имеется параллельный алгоритм работы ядра и параллельно организованная аппаратная часть. Однако между ними, по середине, находится некая бюрократическая организация – последовательная система команд".

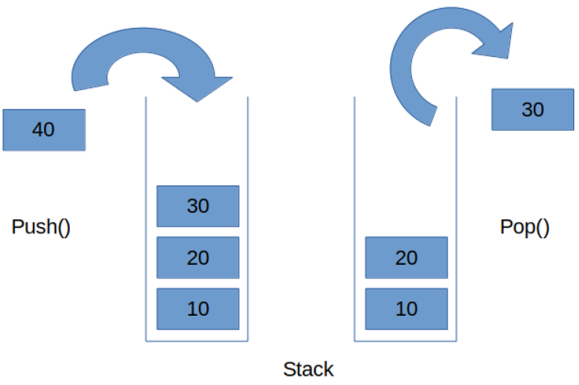
2. С точки зрения компилятора. Оттуда же: "*Компилятор преобразует программу в последовательную систему команд; от той последовательности, в какой он это сделает, будет зависеть общая скорость процесса. Но компилятор не знает в точности, как работает машина. Поэтому, вообще говоря, работа компилятора сегодня – это шаманство*". Компилятор прилагает титанические усилия чтобы втиснуть программу в заданное число регистров, а это число фиктивно. Весь выявленный в программе параллелизм компилятор пытается (как канат в игольное ушко) протащить сквозь ограничения архитектуры. В значительной мере безуспешно.

Похоже, регистровые машины по мере своей эволюции дошли до пределов развития.
А есть ли какая-то альтернатива регистровым машинам?
Да, стековые (безадресные) машины.

Стековые (безадресные) машины.

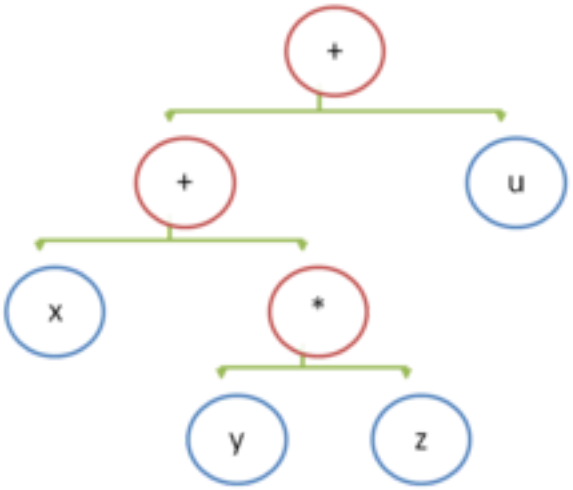
Стеком называют способ хранения и манипулирования данными по принципу LIFO (Last In First Out, последним пришел, первым уйдёшь). Бытовой аналог - магазин с патронами. Для стека существуют две операции

- Push: помещает элемент данных на вершину стека, содержимое стека сдвигается вниз.
- Pop: удаляет элемент с вершины стека, остальное содержимое сдвигается вверх



Фиг.4 Работа стека (отсюда)

Рассмотрим простой пример, вычисление выражения *x+y*z+u*.
При построении дерева разбора компилятором оно выглядит так:



Фиг.5 дерево разбора выражения

Обойдём это дерево в глубину, т.е. попав в вершину дерева, сначала спускаемся вниз к её потомкам, а уже потом обрабатываем саму вершину. Каждый раз, когда обрабатывается вершина, указывающая на переменную, будем выполнять операцию Push со значением этой переменной, встретив бинарную операцию, достанем с помощью Pop оба аргумента, выполним над ними действие, после чего с помощью Push внесём в стек результат. В результате имеем код

```
push x
push y
push z
multiply
add
push u
add
```

результатом которого является значение выражения на вершине стека. Максимально задействованная глубина стека - 3. Так и работает стековая машина, она очень проста, нетребовательна к железу, компилятор для неё лишен всех ужасов распределения регистров …

Удачными образцами компьютеров со стековой архитектурой можно назвать Burroughs B5000 (1961...1986) и HP3000 (1974...2006). Не забудем и про советские Эльбрусы.

Тот же пример для Эльбруса выглядел бы так:

```
ВЗ x
ДВЗ y
ВЗ z
“*”
“+”
ДВЗ u
“+”
```

Где ВЗ означает “вызвать значение”, ДВЗ - “длинная ВЗ”, из предположения что переменные x и z являются локальными для текущей функции, а y и u - глобальными.

В Эльбрусе стек называется СтОп (стек операндов) и состоит из 32 регистров и одного 32-разрядного регистра - маски занятости регистров. При выдаче инструкции в исполнительное устройство процессора формируются внутренние адреса операндов и результата операции - номера регистров из СтОп.

При декодирования бинарной команды, например, суммирования, считается, что номера операндов уже получены ранее, для результата можно назначать любой свободный регистр из СтОп. После того как команда исполнилась, операнды освобождаются.

Что следует из такой организации стека?

- Эльбрусы по праву считаются суперскалярными системами. Как только операнды инструкции вычислены, она может быть исполнена вне зависимости от положения в изначальной последовательности инструкций.. Вполне себе OoO (Out Of Order, внеочередное исполнение).
- 32 регистров достаточно для вычисления выражения любой сложности. А даже если бы и не хватило, компилятор может разбить выражение на подвыражения и сохранить промежуточные значения во временных переменных.
- с другой стороны, перед вызовом вложенной функции стек должен быть пуст, ведь механизма сохранения содержимого стека нет, после возвращения из функции восстановить данные будет невозможно. Рассчитывать на то, что 32 регистра хватит и текущей функции и вложенной нельзя, ведь эта функция может вызвать еще одну функцию, та еще одну …

Получается, что немаленький пул в 32 регистра будет постоянно недозаполнен, ведь вызов функций, да еще и в цикле - вполне обыденная практика.
Алгоритм росписи регистров для регистровой машины распорядился бы таким пулом более эффективно. Впрочем, алгоритм этот появился в 1981 году, когда Эльбрус-1 уже пошел в серию.

Неэффективность использования пула регистров, по видимому и явилась основной причиной, по которой стековые архитектуры уступили место регистровым машинам. Были, конечно, и более поздние попытки спрятать регистровую машину внутри стековой обертки, например, Ignite или ICL2900, но коммерческого успеха они не

имели, т.к. фактически, только лишь ради удобства компиляции, пользователи должны были нести постоянные издержки при исполнении кода.
Однако, если задуматься, код стековой машины - это всего лишь другое (но эквивалентное) представление промежуточного трёхадресного кода. И оба они - варианты записи оптимизированного синтаксического дерева (DAG).

Выше, когда речь шла о суперскалярных регистровых машинах, мы видели сколько сил нужно компилятору, чтобы пропихнуть код сквозь абстракцию архитектуры. Далее декодер процессора тратит силы на то, чтобы понять что именно хотел сказать программист до того как с его кодом поработал компилятор.

Безадресный (стековый) код - уникальная возможность донести до декодера без искажений именно то, что изначально было задумано. Дело за малым - научиться эффективно пользоваться пулом внутренних регистров, для этого требуется эффективно сохранять/восстанавливать его состояние при вызове функций. Справедливости ради, для регистровых машин это тоже весьма нетривиальная задача.