Эта глава посвящена вызову функций. Речь идёт об императивных языках, именно они являются основой индустрии программирования. Функциональное или декларативное программирование, разумеется, важны, но это совсем другая история.

С точки зрения теории вычислимости именно способность вызывать функции делает языки программирования полными по Тьюрингу. Можно сказать, пусть и с некоторой натяжкой, что без них, всё что происходит в программе (вычисление выражений, ветвление (if), арифметические циклы (с известным заранее числом итераций)) относится к классу т.н. примитивно - рекурсивных функций и имеет лишь ограниченное применение.

Полнота по Тьюрингу в рамках одной функции всё таки достижима с помощью циклов с заранее неизвестным числом итераций, но это крайне неудобно с практической точки зрения. Именно так были устроены программы первых компьютеров, когда, например, чтобы организовать цикл в Mark-I, перфолента с программой склеивалась в кольцо. Циклы в программировании - весьма полезная и необходимая конструкция, но именно функции позволяют разделить задачу на более простые подзадачи, действовать в рамках политики “разделяй и властвуй”, …

По мере усложнения и компьютеров и программ для них, стиль программирования в духе “кто во что горазд” или “спагетти-кода” зашел в тупик т.к. начиная с некоторого уровня сложности, программы стало невозможно развивать и поддерживать.

В 1960-х возникла парадигма т.н. структурного программирования, одним из признаков которой является отказ (по возможности) от использования оператора безусловного перехода (goto). Результатом стало появление массы языков программирования: FORTRAN, COBOL, ALGOL, SNOBOL, C, PL/1, Pascal, FORTH, …

В конце 1970-х по мере усложнения задач и структур данных возникла идея объектно-ориентированного программирования и появились такие промышленные языки, как Smalltalk, C++, Ruby, Python, Java, …

В дальнейшем была выдвинута концепция обобщенного программирования, позволяющая отделить отделить алгоритмическую часть задачи от описания данных. Не все в восторге от этой парадигмы, есть мнение, что она не только решает проблемы, но и создаёт новые.

Пожалуй, парадигмы программирования будут возникать и дальше в ответ на усложняющиеся задачи. Тем не менее, если во времена FORTRAN-а и PL/1 глубина вложенности функций редко достигала десяти, то сейчас уровнем в десятки вложенных вызовов никого не удивишь. Более того, уровень вложенности в процессе работы легко может меняться на десятки единиц.

Это приводит к тому, что затраты на вызов/возврат из функций зачастую становятся узким местом в работе программ. Возник этот эффект, конечно, не сегодня, но он постепенно становится всё более значимым.

Вызов функций (регистровые машины)

Проблема с вызовом функций заключается в следующем - после возврата из функции мы ожидаем и возврата содержимого регистров в то состояние, которое было до вызова.

Существует два крайних варианта - сохранением регистров занимается либо вызывающая сторона, либо вызываемая. Оба варианта по своему логичны.

Вызывающая сторона знает, какие регистры заняты чем-то полезным, а какие временно свободны и может сэкономить на сохранении только нужных данных.

Вызываемая сторона в свою очередь располагает информацией о том, какие регистры ей понадобятся и сохраняет только их. Может, и потребуется то всего парочка, зачем же выгружать все 32.

В IBM S/360 (1964 г), 16 целочисленных регистров (32-х разрядных) общего назначения и 4 с плавающей точкой, но нет аппаратного стека. Перед вызовом функции все регистры сохраняются в специальной области (контекст вызова). Компилятор в теле программы отводит специальное место для сохранения регистров для каждой функции, поэтому рекурсия для обычных функций невозможна. Для рекурсивного вызова надо динамически выпрашивать память для контекста вызова у ОС, что очень недёшево.

Параметры передаются в регистре R1 в виде указателя на список указателей на значения параметров (если функция принимает больше одного параметра).
В контексте сохраняются все 16 регистров плюс ссылка на контекст вызывающей стороны и резерв, всего 18 слов. Благодаря ссылке на контекст вызывающей стороны, можно проследить стек вызовов, что очень удобно при отладке.

В PDP-11 8 регистров, но последние два используются как указатель на вершину аппаратного стека и счетчик команд. Т.е. регистров общего назначения 6.
Для этой архитектуры существовало два варианта вызова функций -

- с передачей параметров через связующий регистр, который указывал на область памяти с параметрами
- с передачей параметров через стек. Собственно, так и появился “C calling convention” (cdecl) т.к. именно для здесь выкристаллизовался язык C.

В обоих случаях о сохранности регистров заботится вызываемая сторона. Т.е. если функция собирается использовать регистр R1, она обязана сохранить его в стеке, а перед возвращением вернуть в исходное состояние.

В более современных архитектурах (регистровые машины) используется смешанная стратегия. При вызове функции ответственность за сохранение содержимого части

регистров лежит на вызывающей стороне и это так называемые **volatile** регистры. О содержимом остальных регистров заботится вызываемая сторона и это **non-volatile** регистры.

Кто может определить, к какому классу относится тот или иной регистр? Сам компилятор, к архитектуре это имеет косвенное отношение. Вот что писал в 2003г по этому поводу один из разработчиков GCC:

> *Решение к какой категории отнести конкретный регистр было непростым. В AMD64 15 регистров общего назначения (*%rsp не в счёт, его в любом случае спасать*), причем использование 8 из них (так называемые расширенные регистры) в инструкции требует наличия префикса REX, что увеличивает её размер. Кроме того, регистры %rax, %rdx, %rcx, %rsi и %rdi неявно используются в некоторых инструкциях IA-32. Мы решили сделать эти регистры volatile чтобы избежать ограничений на использование инструкций. Таким образом, мы можем сделать non-volatile только %rbx, %rbp и расширенные регистры. Ряд тестов показал, что наименьший код получается в том случае, когда non-volatile регистрами назначены (%rbx, %rbp, %r12-%r15).*
>
> *Изначально мы хотели сделать volatile 6 регистров SSE. Однако возникли затруднения - эти регистры 128-битные и только 64 бита используются для хранения данных, так что сохранять их для вызывающей стороны дороже чем для стороны вызываемой.*
>
> *Проводились разные эксперименты и выводу в том, что самый компактный и быстрый код получается в случае, когда все SSE регистры объявлены как volatile.*

Именно так дела обстоят и до сих пор, см. AMD64 ABI spec, стр 21.
Тот же ABI поддерживается и в B OS X.

В Microsoft сочли иначе и их деление таково:

- volatile: RAX, RCX, RDX, R8:R11, XMM0:XMM5, YMM0:YMM5
- non-volatile: RSI, RDI, RBX, RBP, RSP, R12:R15, XMM6:XMM15, YMM6:YMM15

А как насчет более регистро-богатых архитектур? Вот как обстоят дела с 64-битным компилятором OS X для PowerPC, где по 32 целочисленных регистров и оных с плавающей точкой:

- volatile: GPR0, GPR2:GPR10, GPR12, FPR0:FPR13, **всего 11 + 14**
- non-volatile: GPR1, GPR11(non-volatile для листовых функций, из которых нет других вызовов), GPR13:GPR31, FPR14:FPR31, **всего 21 + 18**

Итого: разделение регистров на два класса реализует универсальную оптимизацию вызова функций:

- часть регистров используется для передачи аргументов, это быстрее чем работать через стек (плюс часть регистров уходит на служебные нужды)
- число этих регистров определяют разработчики компиляторов на основании статистики и своих представлений о типичном коде
- содержимое остальных регистров спасают только по необходимости, так в случае небольших и не-жадных функций может и спасать то ничего не придётся

- а при вызове полноценной функции содержимое всех регистров будет сохранено, но это ничто по сравнению со временем работы тела этой функции

Berkeley RISC (регистровые окна)

В 1978 г. DARPA инициировала проект VLSI (Very Large Scale Integration), предназначенный для продвижения передовых идей, в том числе, в области архитектур микропроцессоров. Результатом были две ветки перспективных архитектур.

Одну из них развивали в Университете Беркли под названием RISC (Reduced Instruction Set Computer, термин, введённый Дэвидом Паттерсоном (David Patterson)). Проект развивался с 1980 по 1984 гг.

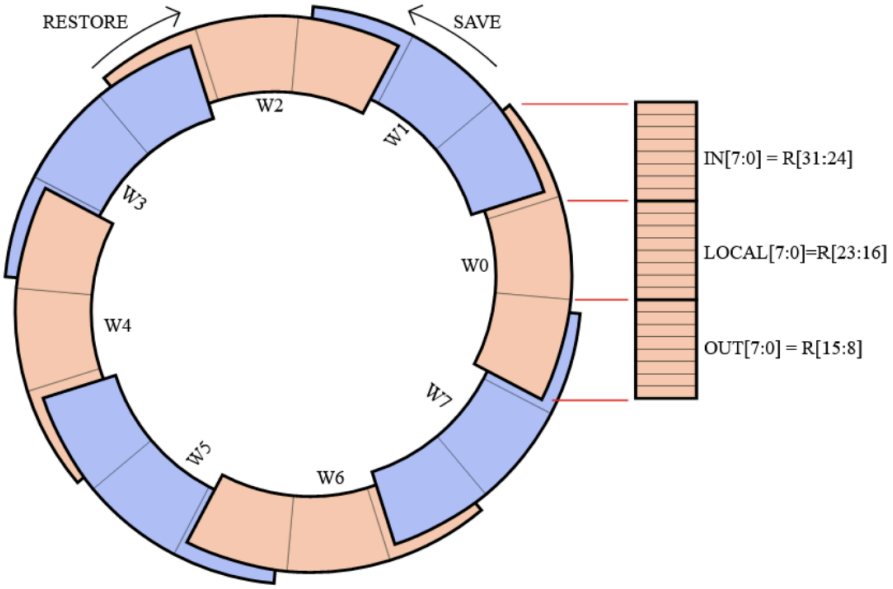
Второй веткой была архитектура MIPS (Microprocessor without Interlocked Pipeline Stages), развивавшаяся в Стэнфордском Университете с 1981 по 1984 гг. Парадоксальным образом, в дальнейшем, MIPS дала старт целому семейству архитектур с групповым названием RISC. Именно развитие этой ветви мы рассматривали в главе CISC vs RISC, её же разбирали выше в этой главе.

А RISC превратился в Berkeley RISC.

Разница между ними как раз заключается в вызове функций. Если в просто RISC стек используется для того, чтобы сохранять значения регистров при вызове функций, то в Berkeley RISC пул регистров фактически является верхней частью стека.

Технически, пул регистров - это кольцевой буфер, кэш для доступа к верхней части стека. Разберём на примере архитектуры SPARC.

SPARC (1986)



Фиг.1 Кольцевой буфер регистров SPARC V8 (отсюда)

- типичный процессор имеет 128 регистров общего назначения,
- из которых одномоментно видны только 24 - которые образуют т.н. окно.
- еще доступны 8 глобальных регистров, итого - 32
- окно состоит из 8 входных (аргументы вызова текущей функции), 8 локальных и 8 выходных (для следующего вызова, если он будет, до того их можно использовать под локальные нужды).
- регистры образуют кольцевой буфер, при вызове функции окно сдвигается на 16 регистров. При этом 8 выходных регистров для вызванной функции становятся входными.
- нумерация регистров каждой функции одинаковая, это R8..R31 (вне зависимости от того, каким физическим регистрам соответствуют эти номера), первые 8 - глобальные
- если необходимые для нового вызова 16 регистров заняты (вершина кольцевого буфера наползает на свой хвост), их содержимое выталкивается в стек.Технически это устроено так - если в процессе вызова функции нет свободных регистров, происходит аппаратное исключение (trap), в процессе обработки которого данные из последнего окна выталкиваются в память и освобождается новое окно.
- аналогично устроена обработка недополнения - столкновение головы и хвоста при выходе из функции приводит к аппаратному исключению и загрузке окна (или окон) в пул регистров.

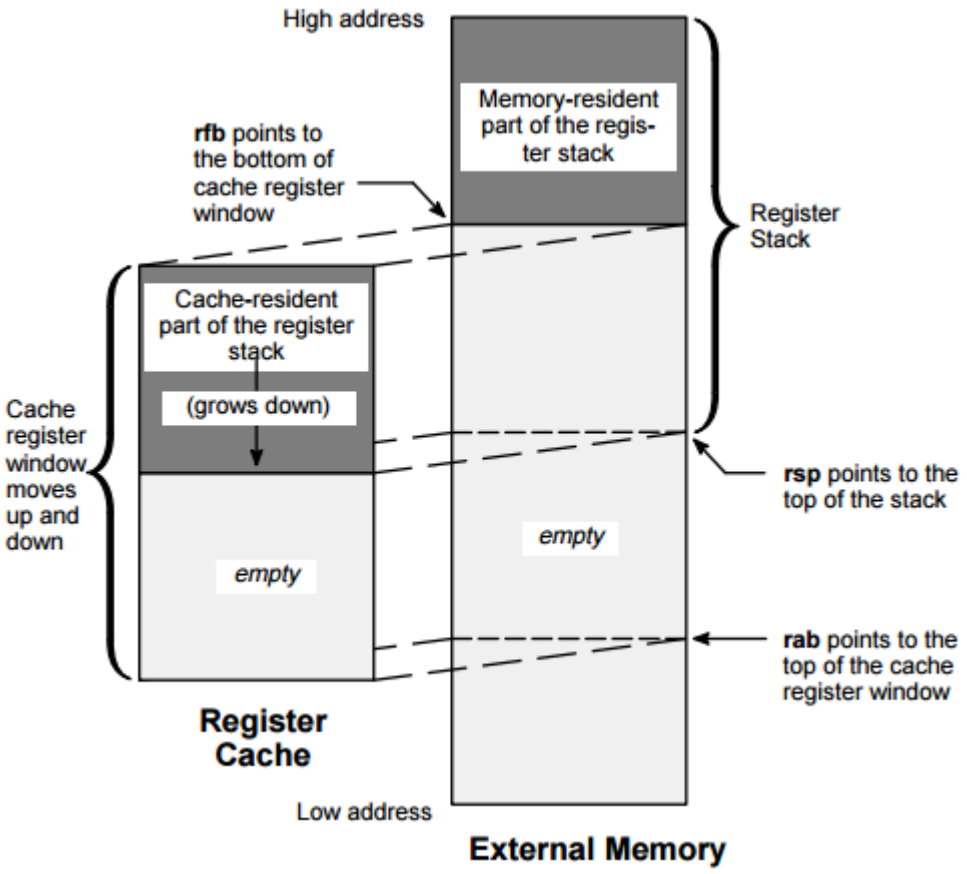
Количество окон (и число регистров) в пуле регистров могло меняться, предполагалось, что это даст архитектуре масштабируемость (Scalability, S - первая буква в SPARC). Для дешевых процессоров предполагалось мало окон, для более дорогих и производительных - больше. Типичное значение - 8 окон, 128 регистров.

AMD 29K (1988 г.)

32-разрядный процессор со 192 (sic!) регистрами. Про него мало кто помнит, хотя в какой-то момент это был самый распространённый 32-разрядный микропроцессор. Он заслуживает отдельных добрых слов.

В нём два аппаратных стека. Несколько стеков в архитектуре не новы, это было еще на Burroughs B5000, советских (да и нынешних) Эльбрусах. Но там второй стек предназначен для хранения адресов возврата из процедур. Здесь же они оба используются для хранения данных:

- memory stack - используется для хранения больших локальных переменных (структуры и массивы) также как и хвоста параметров, если их больше 16. Регистр gr125(глобальный регистр №125, у него есть собственное имя - msp, memory stack pointer) является указателем на вершину этого стека.
- register stack - в наличии 128 локальных регистров, которые образуют вершину стека
    - регистровый стек служит для быстрого доступа к вершине стека в памяти (отличного от вышеописанного memory stack, конечно)
    - глобальные регистры gr126(rab) и gr127(rfb) определяют верх и низ стека, gr1(rsp) хранит указатель на его вершину.



Фиг.2 Стек AMD29K (отсюда)

    - в одном цикле возможны два чтения и одна запись в/из памяти

- в нём нет явных стековых операций таких как push & pop, вместо них при вызове функции для нее освобождается определенное компилятором количество регистров (activation record, так здесь называется call frame)
- доступ к данным из activation record идет через **регистры, которые для каждой функции нумеруются от lr0**
- lr0 и lr1 зарезервированы, в первом адрес возврата, во втором - activation record вызывающей функции
- activation record'ы вызывающей и вызываемой функций пересекаются параметрами аналогично SPARC
- если для вызова новой функции не хватает свободных регистров, происходит trap SPILL, обработчик которого выталкивает часть значений регистров в память, освобождая их
- наоборот, когда свободных регистров становится слишком много, срабатывает FILL.
- чтобы это происходило, компилятор вставляет инструкции

```
        sub gr1,gr1,16          ;function prologue, lr0+lr1+2 local variables
        asgeu SPILL,gr1,rab     ;compare with top of window
. . .                           ;function body
        jmpi lr0                ;return
        asleu FILL,lr1,rfb      ;compare with bottom of window gr127
```

Какие интересные идеи следует здесь отметить?

1. Нумерация регистров для каждой функции своя, это особенность Berkeley RISC
2. А вот расщепление стека - особенность именно этой архитектуры. В SPARC'е регистровые окна сохраняются в тот же самый стек, где лежат и обычные (не быстрые) переменные. И fill/spill делаются с разрывами - каждое окно из своего фрейма.
3. В отличие от SPARC, размеры части входных/выходных параметров и локальных регистров не фиксированы (по 8 регистров), что даёт компилятору гибкость их использования.

Вот это разделение стека на "большой, но медленный" и "маленький, но быстрый" очень интересно. Жаль, что компания AMD в какой-то момент не потянула одновременно две ветки процессоров - x86-32 и AMD29K. Выбор был сделан в пользу x86, хотя есть предположение, что ядро AMD29K (или его отдельные блоки) было помещено внутрь линейки процессоров K5 (аналог Intel Pentium) и продолжило там своё развитие.

Intel i960 (1988).

32-разрядный процессор с 16 локальными и 16 глобальными регистрами общего назначения. Параметры передаются через глобальные регистры, при вызове функции все локальные регистры сохраняются специальной инструкцией. Фактически все локальные регистры - non-volatile, ~~но~~ спасаются принудительно в надежде, что аппаратная поддержка придаст этому какое-то ускорение.Впрочем, по нынешним временам это всего-лишь одна линия кэша.

Процессор был предназначен для встраиваемых систем и не имел MMU (Memory Management Unit, виртуальной памяти). Некоторые поздние варианты процессора стали суперскалярными, так что это единственный суперскалярный процессор без MMU.

В середине 1990 гг. Intel перестала развивать эту ветку процессоров, хотя, производство продолжалось до 2007г.

Intel Itanium (2001)

Архитектуру этого процессора мы разбирали в главе 2.3.Superscalar vs VLIW, здесь же коротко коснёмся его реализации регистровых окон.

- всего 128 целочисленных общего назначения регистров общего назначения (64-разрядных) и столько же плавающих
- 32 из них считаются глобальными
- 96 - локальными и они образуют верхушку стека регистров
- процессор сам заботится об их загрузке и выгрузке, создавая иллюзию бесконечного стека регистров (RSE, Register Stack Engine)
- при вызове функции, для нее специальной инструкцией alloc создается регистровое окно, при этом компилятор должен явно задать его размеры
- окно устроено аналогично SPARC, но размеры его частей гибкие и также задаются компилятором, общий размер не больше 96 регистров
    - in часть предназначена для входных параметров функции, не больше 8
    - local часть для локальных данных
    - out - предназначена для параметров функций, которые будут вызываться из этой, не больше 8
- при вызове функции из функции регистровое окно сдвигается и лёгким движением out часть превращается в in
- обычный стек также присутствует, в него компилятор кладет всё, что не удалось разместить в стеке локальных регистров

Вцелом, работа регистровых окон Itanium-а очень похожа на оную в AMD29K. Основное отличие в том, что процессор берёт на себя всю работу по сдвигу окон при пере(недо)-полнении пула регистров. Это важно т.к. реализация механизма через аппаратные прерывания (AMD29K) не очень то дешёвая.