Содержание

1 Intro.

Напоминание: ассемблер — такой язык, который сильно отличается от остальных тем, что полностью зависит от процессора. Ассемблер — набор мнемоник для процессора. А значит ассемблеров столько же, сколько и ISA.
На нашем курсе будет изучаться x86. Помимо него можно встретить ARM (хотя это семейство ISA, они не совместимы друг с другом), MIPS (старое, встречается в роутерах и первых двух PlayStation, из роутеров замещается т.к. мертво), Power (старые маки). А ещё есть новая ISA: RISC-V. Она очень хорошо если захватит мир (т.к. это открытая ISA), но пока нет. Сейчас она очень быстро захватывает контроллеры. Например, контроллеры винчестера. Раньше там были ARM'ы. С ARM сейчас все активно убегают, потому что они отозвали лицензию Huawei, и все остальные боятся, что у них тоже отзовут. Кроме Apple, которые участвовали в создании ARM. Но это точно не всё. Потому что есть видеокарты, которые вполне можно считать специализированными устройствами со своими ISA. В линейке новых видеокарт AMD публикуют документацию к их ассемблеру. И ровно также существует тьма специализированных популярных ISA, но мы их касаться не будем.
При этом важно следующее. ISA и ассемблер сопоставляются друг другу почти один к одному. Одно только: в x86 есть два **синтаксиса** ассемблера: Intel и AT&T+GNU. Первый `ADD RAX, RBX` — прибавление `RBX` к `RAX` и запись в `RAX`. GNU же выглядит как-то так: `ADDL %RBX, %RAX`. То есть у команды уточнена битность (зачем-то), поменяны аргументы местами и стоят проценты. Мы будем использовать первый, а наш ассемблер (как программа) будет nasm.

2 История x86.

x86 странноват по историческим причинам. Сначала был 16-битный процессор i8086. К тогда мелкой кампании Intel пришла корпорация IBM и заказала процессоры. i8086 был середнячком того времени. Он уже был обременён совместимостью с 8-битным i8080. Intel позиционировала это как временную меру, но то крутое, что она хотела сделать, так и не состоялось. Что интересно про i8086 — 20-битная адресация. Как это делать с 16-битными регистрами? Сложно сказать.
В итоге i8086 попал в очень популярную линейку компов IBM, где был MS-DOS. Короче, i8086 просто повезло. Потом Intel выпустила i8088 — более дешёвую версию, у которого была 8-битная шина, поэтому он работал медленнее. За i8086 вышли его настоящие последователи: i80186, i80286, i80386, i80486... Всё это обратно совместимо баг в баг. Отсюда и название x86.
Совместимость была лютейшая, и вот промер: команда, которая считает синус, имела баг. При числе x в районе π его синус равен примерно $x-\pi$. Значит надо вставить точную константу π. Но оказалось, что константа недостаточно точная. Это поправили AMD, пользователи подняли крик, и они вернули старую неточную константу. Впрочем, практически это никого не напрягает, потому что там точность — относительное понятие (потому что совсем точно знать ответ бесполезно, потому что `double` уже скорее всего округлён). Или другой пример: до сих пор современные x86 запускается в начале именно в режиме i8086.
В i80286 добавили защищённый режим, в i80386 — 32-битный режим. В i80486 не скажу, что было, а потом был Pentium (архитектура P5), который стал первым суперскалярным процессором (что, конечно, на ISA не повлияло). Из интересного, Pentium сейчас используется для разных экспериментов.

Например, когда они показывали процессор на солнечной батарее, это был именно Pentium. Дальше был Pentium MMX, в котором добавили первое расширение на SIMD — single instruction multiple data. Предназначено для обработки мультимедиа (например, там есть встроенное сложение с насыщением), или любых других больших массивов данных. Отсюда и название, MultiMedia Extension. Параллельно с этим был Pentium Pro (архитектура P6) (out-of-order superscalar, в отличие от Pentium, который in-order superscalar), но он дорогой, а значит непопулярный.

Немного офф-топ о том, кто использует ассемблер. SIMD (и в целом модные расширения, в которые компилятор не очень умеет) — одна из основных причин писать на ассемблере, потому что писать что-то целиком на ассемблере бессмыслено — хороший компилятор вам сильно поможет. Другие причины:

- Сделать что-то, что умеет ОС, и что не умеет ваш язык.
- Системы защиты, вирусы и антивирусы. Код там, кстати, похожий, лол. Более того, они иногда лезут в машинный код, кодогенерацию на лету, более того код может даже полиморфным получаться. (Так Denuvo работает.)
- Низкоуровневые оптимизации. Если вы компилятор пишете, например. Если вы пишете код с арифметикой с насыщением. Вопрос: догадается ли компилятор, что вам надо использовать инструкцию? Можно напрямую писать на ассемблере, если вы переписываете какую-нибудь функцию, которая занимает существенное время.
  Что интересно, вы обычно пишете несколько ассемблерных вставок под одно и то же. Почему? Потому что ISA разные.
  Обычно речь идёт именно про SIMD. Не только потому, что его сложно заметить в высокоуровневом коде, но ещё и потому, что обычно SIMD происходит в числах с пониженной точностью. А значит ваш язык может просто не уметь выражать то, что вы хотите.
- Отладка программ. А вы думали иначе? Если есть исходники, то всё круто. Если нет, со знанием ассемблера вы всё равно можете. В т.ч. когда происходят баги компилятора.

Продолжаем историю. Pentium MMX был первым процессором, на котором можно было смотреть видео 320×240 30 кадров в секунду в MPEG-I без аппаратных ускорителей. После него появился Pentium 2 (который позиционировался как помесь Pentium MMX и Pentium Pro; это правда, но процессор подешевле, конечно, чем Pro) и Pentium 3. Там появился SSE — своеобразный SIMD для вещественных чисел. Потом произошла неудача с Pentium 4 и SSE2 (и SSE3). Если до этого микроархитектура была одинаковой и работающей, то Pentium 4 с новой микроархитектурой дико грелся (по тем временам) и выдавал низкую скорость (из-за извращённой архитектуры) несмотря на адские мегагерцы.
Почему нет 64-битного режима? Потому что во время Pentium 4 Intel хотели выпустить Intel Itanium, свой, новый и с лицензией только у себя. Но он по многим причинам был окрещён Intel Itanic и утонул. В итоге Intel поддержали AMD-шное расширение на 64 бита. Спойлер: оно называется amd64.
Параллельно был Pentium M — развитие Pentium 3 с оптимизацией в энергопотребление. Это для ноутбуков. Когда стало ясно, что Pentium 4 — это провал, инженеры взяли Pentium M и сделали из него Core 2, в котором добавили расширение SSSE3. Это расширение уже про целые числа, и довольно полезные. В полее поздних ревизиях было SSE 4.1. Core 2 был хорош.
Дальше была линейка i7 с микроархитектурой Nehalem (из которых уже потом были урезанные i5), и там добавили SSE4.2. Дальше серия ix-2xxx и микроархитектура Sendy Bridge, где добавили AVX., потом i7-4xxx (микроархитектура Haswell), где появился AVX2 — расширили регистры SIMD до 256 бит. А ещё там FMA — возможность сделать `d = a * b + c` с толко одним округлением в конце. Она хороша, но считается и за сложение, и за умножение, потому что увеличивает вдвое FLOPS. Потом была серия ix-6xxx (в этот момент начинают расходится цифры для десктопов и всего остального), так что будем смотреть по микроархитектуре (тут Skylake). Тут было AVX512, но было оно только у серверного Skylake. Обещали, что добавят в десктопы, когда будет новый техпроцесс (10 нм). Спойлер: нового техпроцесса не произошло.

Отсюда и до 10xxx происходит Skylake.
Что дальше? Серия 11xxx (Cypress Cove), которая быстро умерла, в которой они добавили AVX512. Но тут им дали пинка AMD, потому что они начали отгружать ядра подешёвке (и по деньгам, и по энергии). А Intel так не могли, в энергию не лезет. Поэтому в серии 12xxx надо было откуда-то кучу ядер. Поэтому туда добавили слабые ядра. Откуда их взяли? Из Atom — попытки Intel зайти на мобильный рынок (но провалились). Короче теперь в 12xxx (Golden Cove — сильные ядра + Gracemont — слабые ядра) и 13xxx с 14xxx (Raptor Cove + Gracemont) сериях крутые ядра поставляются вместе с пачкой убогих, которые не умеют AVX512. Поэтому Intel решили не поставлять AVX512 туда, где есть слабые ядра. Из интересного, когда 12xxx только выходили, люди научились в BIOS-е выключать убогие ядра и включать AVX512. Больше так нельзя, Intel теперь аппаратно отключают AVX512.

AMD. Ни одна компания не хочет завязываться на другую компанию. Поэтому когда IBM пришли к Intel, они поставили одно условие: они дадут одну лицензию компании AMD. Есть ещё одна лицензия, но она переходила по рукам, выпускала банкоматы под виндой, а сейчас её просто держат в подвале и не используют. Из интересного компания, которая имела третью лицензию, сделала себе аппаратное перепрограммирование команды, которая говорит, кто производитель процессора. Зачем им прикидываться Intel'ом или AMD? Потому что они обнаружили, что винда не грузится. А в винде было написано, что если Intel, делаем одно, если AMD — другое, а иначе нечто нерабочее. При помощи этого ещё тестировали Intel-компилятор, чтобы доказать, что он не использует расширения (даже если может) на процессорах AMD.
i8086 просто склонировали, а дальше, AMD выпускали то же самое (с названиями Am286, Am386, Am486 и Am5x86), но нестандартными расширениями. Потом был K5, K6 (Pentium 2), K6-2 (в который добавили MMX), и K6-3, в котором появилось расширение 3DNow! (теперь можно считать 2 `float` на MMX; Intel её естественно не поддержали, а вместо этого скрафтили SSE, в котором можно считать 4 `float`'а). Во время Pentium 3 появился Atlon (3DNow! Pro).
Дальше был Atlon XP, в котором наконец-то появился SSE. Дальше неинтересно до тех пор, пока не появился Atlon64 (времена Pentium 4). И он был сильно круче, чем Pentium, несмотря на низкую частоту. Да ещё и в Atlon64 был 64-битный режим (и SSE2). Зачем 64 бита? Потому что адресное пространство очень маленькое. А значит что? Что все (особенно любители серверов) покупают Atlon64. Вместо уходящего в закат и сомнительного Pentium 4 и откровенно отстойного Intel Itanium.
После этого был какой-то непопулярный кал (Phenom и линейка с названиями строительной техники), а сейчас ядро Zen (линейка Ryzen), в которым убрали всякий мусор (типа 3DNow!) и сделали сразу AVX2. Из интересного — Zen 4, который поддерживает AVX512. Ядра поменьше там тоже есть, но не в обычных процессорах, а в мобильных, причём следующего поколения. Ещё есть Zen 4c: «c» — более плотная паковка деталек Zen 4 + порезанная максимальная частота и порезанный L3 вдвое. За счёт этого туда запихали побольше ядер на кристалл. А потом начали мешать обычные и «c», что сильно круче, чем происходящее у Intel. Дальше Zen 5, которых 3 вида: обычные, «c» и мобильные, и мобильные порезаны.

3 Займёмся делом.

Как закончить программу? Одной командой: `ret`. Одна проблема: надо объяснить, что с этого надо начинать. Проще всего жить по правилам C и сказать, что это конец функции `main`. Можно иначе, но так проще и логичнее. И вот программа

```
main:
    ret
```

Разве что под Windows лучше написать `_main` (64бит такого прикола не имеет). Это уже почти работает: надо только сказать, что `main` должно быть видно снаружи:

```
    global main
main:
    ret
```

Возможно, потребуется сказать

```
        global main
        section .text
    main:
        ret
```

Если хочется код возврата дать, то так:

```
        global main
        section .text
    main:
        mov eax, 0
        ret
```

Уже хорошо. Теперь возьмём nasm и попросим его скомпилировать это. Правда, надо ему указать, под какую систему компилировать. Например, `-f win32` под Windows. У вас появится объектный файл, который можно скормить линковщику (gcc, например).

3.1 Регистры.

Первое: x86 — reg-mem-2 архитектура. И дальше мы будем говорить про регистры, про которые мы что-то можем знать. Т.е. регистры общего назначения. Они вот такие:

- `ax` — аккумулятор.
- `cx` — счётчик.
- `dx` — данные + расширение аккумулятора.
- `bx` — базовый.
- `bp` — немного по-другому базовый.
- `sp` — указатель в стек.
- `si` — source.
- `di` — destination.

При этом начиная с 32 битов эта специализация ослабла, но не исчезла. Всё описанное тут — 16-битное. Причём для обратной совместимости с 8-битным i8080 к регистрам `*x` можно обратиться не полностью, а по половинкам: `*l` — младшая, `*h` — старшая. При этом при изменении частей другая часть не меняется. Когда началось 32-битное расширение, все регистры расширили до 32 бит, написав `e` в начале имени регистра. Теперь, например, `ax` — младшие 16 бит `eax`. Потом добавили 64-битные версии, да ещё и новых регистров добавили, но о них не сейчас.
Кроме этого и совсем системных существуют два интересных: flags (eflags, rflags) и ip (eip, rip). Что такое флаги? Когда вы делаете действие, помимо нормального результата устанавливают флажки. Они представлены битами регистра FLAGS. Например: сложение может переполниться и тогда он установит CF — carry flag. Есть, например, ZF — её ставят, например, сложение и вычитание, если в результате ноль. Из ещё полезного SF — sign flag — равен старшему биту результата, а значит что он показывает знак результата. Сильно позднее нам будет интересен DF, а сейчас обсуждать его не будем.
А ip — instruction pointer. Хранит адрес следующей исполняемой инструкции. Ни к флагам, ни к ip нельзя обратиться напрямую, но некоторые команды могут с ними взаимодействовать.

3.2 Адресация.

Когда в качестве аргумента какой-то команды хочется использовать штуку, которая находится по адресу какому-то, вокруг адреса ставятся квадратные скобки. Что можно использовать в качестве адреса? В разных режимах разное. В 16-битном: сумма произвольного подмножества из: базы (регистра `bx` или `bp`), индекса (`di` или `si`) и сдвига — 16-битной константы. В 32-битном: (любой регистр) + (любой регистр, кроме `esp`) * (число из 1, 2, 4, 8) + (32-битная константа). Опять же, любая часть опускается. В некоторых ассемблерах можно менять штуки местами, в большинстве умножение на 1 можно опустить, а ещё некоторые понимают `[eax * 5]`. Как? Это равносильно `[eax + eax * 4]`. Про 64-битную адресацию не расскажем.

3.3 Инструкции.

Присваивания. Никто из инструкций этого параграфа не меняет флаги.

mov и его друзья. Присваивание.

- В нормальном синтаксисе `mov dst, src` присваивает `src` в `dst`. Один любой из аргументов может быть памятью. Например, `mov eax, [ebx]`. Хм-м, сколько данных тут будет присвоено? 4 байта т.к. `eax` имеет 32 бита. Поэтому из памяти, адресом которой является `ebx` читается 4 байта и пишется в `eax`. Каким образом? Берётся `ebx` и ещё следующие 3 байта, и читаются. В x86 раньше в памяти идут младшие байты числа. Это называется little-endian. В других системах может быть строго наоборот, тогда это называется big-endian. В упоротых системах может быть смесь. В сетевых протоколах, к сожалению, выбран big-endian.
  Кстати, на последнем месте может быть константа. Но это может привести к неясности. `mov [eax], 123` — это присваивание какой битности? Так команда (в нормальных ассемблерах, таких как nasm) некорректна, поэтому нужно уточнить: `mov byte [eax], 5` или `mov [eax], byte 5`. Принято первое. Получается, что один байт. Можно также написать `word`, `dword`, `qword` для 16, 32 и 64 бит соотвественно. Ещё перед памятью может `ptr` стоять, но это не nasm, а masm или её такой кринж.

- Ещё есть `movsx` и `movzx` — копируют меньшее в большее следующим образом: меньшая часть копируется, остаток заполняется нулём для `movzx` либо знаковым битом для `movsx`. Best practices: если нужно присвоить байт в большой регистр, применение этих команд намного лучше чем `xor` с самим собой + `mov` байта. Это потому, что присваивание регистров размера равного битности системы осуществляется быстрее.

- Ещё есть $\texttt{cmov}_{\texttt{cc}}$. Они, имеют чуть меньше вариантов откуда и куда загружать. Например, нельзя на второй аргумент написать константу. А прикол их в том, что они загружают значение при определённом условии. Например, `cmovz` — присваивает, если стоит ZF и `cmovnz` — если не стоит. Помимо таких простых штук есть ещё комбинации, которые позволяют сравнивать числа. Как сравнивать числа? По сути, вычитанием. Если аргументы равны, будет `ZF`, если первое меньше второго в беззнаковом типе, то это `CF`. А если в знакомом, то `OF`. И можно скомбинировать их в, например, `cmovbe` — меньше либо равно (комбинация с || z).
  Но у этого семейства команд есть беды. Во-первых, `cmov` умеет присваивать только регистры (т.е. не работает ни с памятью, ни с константами). Во-вторых, она не умеет в 8-битные регистры.

xchg. Обмен без дополнительной памяти. **Атомарный** swap. Если вы хотите поменять значения двух регистров, то вы делаете что-то странное, но ладно. Но если вы делаете `xchg` с памятью, то она работает невероятно долго (из-за атомарности).

bswap и movbe. `bswap`: Берёт 32-битный (в 64-битном режиме можно и 64-битный) регистр, меняет endianness на противоположный (т.е. меняет байтики местами). `movbe`: `mov`, но с попутным изменением endianness. `movbe` — относительно новая команда, а следовательно может не быть, и код может не оптимизироваться компилятором без указания процессора в коде.

lea. Имеет два аргумента: регистр и память. Она считает адрес памяти, НЕ ОБРАЩАЕТСЯ В ПАМЯТЬ и просто присваивает адрес в регистр. Типичное применение — трёхаргументное сложение, но насколько оно эффективнее — вопрос.

cwd и друзья. `cwd` — берёт старший бит `ax` и заполняем им `dx`. `cdq`, `cdo` делают то же самое, но с парами `eax`:`edx` и `rax`:`rdx`.
Ещё есть `cwde`, `cbw` и `cdqe` — соотвественно расширяют знаковым битом `al` до `ax`, `ax` до `eax` и `eax` до `rax`. Зачем — поймёте потом.

Арифметика. Важное замечание: в языках мы привыкли что типы имеют знаковость, а операции — нет. Тут ровно наоборот: регистры и память — это просто биты. А арифметика интерпретирует их как число (знаковое или беззнаковое), и совершает операцию.

Сложение и вычитание. `add` и `sub`. Делают соотвественно += и -=, в синтаксисе как `mov`. Попутно ставят флаги. Также есть `adc` — складывает два числа и прибавляет к ним CF в качестве единички — и `sbb` — вычитание, и вычесть флаг переноса.

Умножение. Беззнаковое — `mul`, знаковое — `imul`. Принимает один аргумент (регистр или память, не константу) и умножает его на `ax` (или аналог аналогичной битности). Результат имеет битность вдвое больше. Куда присваивается результат — чуть сложнее:

- `ax` = `al` * arg8.
- `dx:ax` = `ax` * arg16.
- `edx:eax` = `eax` * arg32.
- `rdx:rax` = `rax` * arg64.

Что за непонятное двоеточие? Возьмём два регистра, склеим из них одно число побольше, где первый регистр — старшая часть.
А ещё есть `imul reg, reg` — делает *=, отбрасывает старшую часть. Почему нет такого же для беззнаковых? А потому что верно следующее утверждение: младшая половина результата получается одинаковой для знакового и беззнакового умножения.
А ещё есть `imul reg, reg, const` — умножает второе на третье, присваивает в первое. Опять же, знаковость на самом деле не важна.

Деление. Есть `div` и `idiv`, и в их классической форме они похожи на правила для умножения. Например, он берёт пару `edx:eax` и делит на аргумент (всё ещё регистр или память). Частное записывается в `eax`. Что более интересно, он также считает остаток и присваивает в `edx`. Для 16 битов и 64 битов понятно, для 8 — остаток присваивается в `ah`.
Деление на ноль? Это определено как исключительная ситуация, и дальше происходит ситуация, похожая на прерывание: система останавливает выполнение, переходит в ядро к обработчику ОС и начинает делать какой-то другой код. Но это ладно. Мы делим 64-битное число на 32-битное. Оно легко может не влезть в 32-битное. И это тоже считается делением на ноль. Единственный способ заработать это на C — знаково поделить `INT_MIN` на `-1`.
Что делать, если мы хотим делить друг на друга числа одинакового размера? Ну, если числа беззнаковые, то старшую часть надо обнулить. Если знаковые, то заполнить знаковым битом, и тут мы вспоминаем `cwd` и друзей.

Инкремент и декремент. `inc` и `dec`. Делают почти всё как надо, но не меняют флаг переноса. Если вы используете этот факт, они сильно замедляются. Что в них хорошего — занимают меньше памяти, а чем меньше памяти, тем больше команд влезет в кэш.

neg. Меняет знак числа на противоположный. Флаги ставит ровно как «инвертировать все биты, добавить один».

Логика. `and`, `or`, `xor`. Побитовые логические операции. И `not` — побитовое отрицание.

xor для обнуления. Традиционно для обнуления регистра принято использовать `xor` его самого с собой. Эта инструкция занимает меньше памяти, и более того распознаётся процессорами как некоторая идиома, и если вы сделали, например `xor eax, eax`, то дальше если вы используете `eax`, то его использование не приостанавливает Superscalar.

cmp и test. `cmp` — делает то же самое, что `sub`, но не пишет результат, пишет только флаги. `test` — то же самое, но `and`, а не `sub`. Первое используется для сравнений на больше или меньше, второе — для установленных битов и сравнения с нулём т.к. `cmp` чего-то и нуля занимает много места (т.к. надо записать 32-битную константу 0). А если ваша команда занимает много места в памяти, вы меньше команд поместите в кэш.

Сдвиги. `shr`, `shl`, `sar`, `sal`.
Это всё логические или арифметические сдвиги. Сдвиги принимают два аргумента: что двигать и насколько. Сдвиг вправо двигает битики на несколько позиций в сторону меньших битов. Самый последний сдвинутый бит перемещается в CF. Но тут лучше документацию читать. Логический сдвиг заполняет освободившиеся позиции нулями (а значит полностью аналогичен делению на слепень двойки). Арифметический сдвиг заполняет их знаковым битом, что почти аналогично делению. Разница в том, что арифметический сдвиг округляет к $-\infty$. Сдвиги влево одинаковы и сдвигают в другую сторону. В CF пишется младший сдвинутый бит.
Тут есть проблема с аргументами. Первым аргументом может являться только регистр или память, а вторым — константа или `cl`. Причём из него берутся только младшие биты в необходимом для этого количестве. Например, при сдвиге 32-битных чисел берутся только 5 бит.
А ещё есть сдвиги двойной точности: `shld` и `shrd`. Принимают на вход два регистра и число (либо CL вместо числа), двигает первый регистр в нужную сторону, но пустые места заполняются частью второго аргумента. То есть для сдвига вправо пустые места заполняются младшими битами второго регистра. Для сдвига влево — заполняются старшими битами второго регистра. CF заполняется так же, как и для обычных сдвигов.
И наконец `rol` и `ror` — циклический сдвиг битов в одну из сторон. Последний выдвинутый бит всё ещё отправляется во флаг переноса. Ещё в ту же степь есть `rcr` и `rcl`. Возьмём параметр, приклеим к нему CF, сделаем циклический, сука, сдвиг полученного числа, битности на 1 больше, чем аргумент.

Работа со стеком. У вас есть стек, он растёт вниз по адресному пространству. И вы можете класть туда значения и брать их оттуда.

push/pop. Пушить на стек можно регистры, память, константы. Это по сути сдвиг `esp` выше по стеку на размер аргумента (то есть вычитание) + запись туда. `pop` принимает один аргумент (регистр или память), куда записать то, что сняли со стека. Если записывать никуда не надо, то что? Ну, можно просто прибавить значение к `esp`.
Совет: очень хорошо для производительности, если все штуки на стеке выровнены кратно режиму программы.
Ещё интересное: вы не можете класть на стек (и убирать со стека) 8-битные значения.

pusha(d)/popa(d). Здесь и далее круглые скобки означают, что это дополнительный суффикс. Т.е. тут две пары команд: `pusha`, `pushad` и `popa`, `popad`.
`pusha` пишет на стек все регистры (в порядке, указанном выше). При этом в зависимости от ассемблера, на стек пишутся либо все регистры текущей битности, либо все 16-битные регистры. С суффиксом `d` гарантированно кладёт 32-битные регистры. Ещё некоторые ассемблеры понимают `pushaw` — точно положить 16-битные регистры. `popa` читает все регистры и восстанавливает все, кроме `sp`.

Передача управления. Самый простой — `jmp` — переход по метке, регистру или значению в памяти. На практике всё это пересчитывается в смещение относительно текущей команды.

Условные переходы. `j`$_{cc}$, где `cc` — то, что мы видели выше в разделе с `cmov`. Применимо только к меткам, если хочется по регистру или памяти, делайте абсолютный переход и обходите его условным переходом.

Вызовы функций. `call` и `ret`. `call` — принимает на вход метку, регистр или адрес, записывает на стек адрес следующей команды и делает `jmp` по своему параметру.
`ret` — снимает со стека значение и переходит по нему.
`ret n` — снимает со стека адрес, увеличивает `esp` на `n`, переходит по тому, что в первый раз снял.

Прерывания. Прерывания предназначены для взаимодействия с внешними устройствами. Пример: пользователь нажимает на клавиатуру, а мы узнаём, что он там нажал. Можно периодически спрашивать клавиатуру, что там нажато, но это плохо из-за того, что если вы часто спрашиваете, это активное ожидание (значит долго), а если редко, то мы пропустим, как пользователь нажал и отпустил. Это Polling называется. Поэтому используем прерывание: устройство посылает прерывание. Процессору похрен на клавиатуру до тех пор, пока ему не придёт прерывание. Тогда процесс перейдёт на обработчик прерываний (его устанавливает ОС). Тут уже клавиатура опрашивается, данные сохраняются, и вы продолжаете исполняться как ни в чём не бывало. Но тут нужна аппаратная поддержка: PS/2 работает через прерывание, а USB клавиатура работает через Polling. Ещё из интересного: PS/2 работает тупо и всегда. А USB может перестать работать из-за проблем с дровами.
Так вот `int` — переходит на обработчик прерываний. Принимает на вход константу от 0 до 255. По сути программная эмуляция настоящих прерываний. Пример: `int 80h` зовёт ядро (на Linux). Ещё есть парная команда — `iret` т.к. прерывание сохраняет на стек побольше, чем функция (а именно адрес возврата, флаги, иногда стек). Впрочем, программные прерывания — не рекомендуемый способ позвать привелегированный режим, потому что это прерывания — это долго и неэнергоэффективно.
У `int` есть несколько специальных форм, например `int3` — отладочное прерывание. Если у вас нет отладчика, вы умираете, если есть, то он её перехватит и начнёт делать что-то своё. Из прикольного у `int3` однобайтовый код `0xCC`, а у обычных прерываний код двухбайтовый.

Вызовы ядра. `sysenter` и `sysexit`. Первое — джамп в ядро, второй — из ядра. Куда конкретно ядро? Есть машинно-специфичные регистры (MSR), и там есть в частности адрес, куда `sysenter` переходит.
В отличие от `int` эти команды не сохраняют значение на стек. Поэтому прямое использование этих команд затруднено. Обычно ОС Вам делает `call` по адресу, в котором находятся `sysenter` и `ret`. И ОС делает так, чтобы возвращаться на этот `ret`. Но это всё равно быстрее прерываний.
`syscall` и `sysret`. То же самое, что и `sysenter` и `sysret`, но сохраняет адрес возврата на стек. Проблема в том, что это придумали AMD, поэтому в 32-битном режиме этих команд может не быть. Но это не всё. Ведь 64-битный режим придумали AMD, и они объявили, что в 64-битном режиме `sysenter` и `sysexit` не поддерживаются. Intel, естественно, их поддерживает, но по умолчанию в 64-битном режиме все делают `syscall`.
Из интересного в Windows вам дают системную библиотеку, в которую вы делаете `call`, и она сама там разбирается, как трогать ядро. И вам не надо думать, какие у вас поддерживаются прерывания, вызывать ли вам `sysenter` или `syscall`. Хотите не иметь проблем — залезьте в системную библиотеку.

nop. Делает ничего. Формально эквивалентен `xchg eax, eax`. А ещё есть многобайтовые версии `nop`. Применяется, чтобы начало короткого цикла выравнять по границе кэш-линии.

ud2. Команда, которой не существует. Исполняя её, гарантированно упадёт с ошибкой «команда некорректна».

4 Calling conventions.

Calling convention — договоренность о том, как передаются параметры в функцию, куда помещается возвращаемое значение, где сохраняется адрес возврата (именно то, что отличает `call` от `jmp`), какие регистры функция обязана вернуть в исходное состояние.
Если вы хотите указать прямо около функции, какого соглашения она должна придерживаться (например, `cdecl`), в clang и MSVC это делается примерно так: `void __cdecl foobar();`, а в gcc — при помощи: `__attrubite((cdecl))__`.
Ниже будет краткое описание происходящего в 32-битных системах. Если вам этого мало, вот очень крутая ссылка:

cdecl C-шная библиотека использует конвенцию `cdecl`, которую сейчас и опишем: адрес возврата сохраняется на стеке, аргументы кладутся на стек тем ниже, чем раньше аргумент. То есть сначала кладётся последний аргумент, потом предпоследний, и так далее, а после первого кладётся адрес возврата.
Например, в функции вида `int f(int a, int b, int c)`. Тогда для вызова надо сделать так:

```
push c
push b
push a
call f
add esp, 12
```

С возвращаемым значением есть беда. Если оно влезает в регистр, его положат в соответствующую часть аккумулятора соответствующей битности. То есть если вы вернули `bool` или `char`, вам запишут результат в `al` (а в остальной части `eax` будет мусор). Если результат 16-битный — вернут в `ax`, если 32-битный — `eax`, а если 64-битный (в 32-битном режиме) — `edx:eax`. Что если больше? Тогда функция внезапно преобразовывается: нулевым её аргументом начинает идти адрес того, куда положить возвращаемое значение. Её выделяет вызывающий, потому что вызываемая функция почистить её не сможет (т.к. закончится). А выделять и освобождать в разных функциях... Не, ну, можно, конечно, если в соглашении о вызовах постулировать, как именно она выделяется, но у нас могут быть разные аллокаторы, например... Поэтому нет, кто освободил, тот и чистит.
Также конвенция вызовов заставляет вас сохранять некоторые регистры (а именно `ebx`, `ebp`, `esi` и `edi`), поэтому если вы хотите испортить их значение, положите их на стек в начале функции, а потом снимите обратно. Ещё функция с соглашением `cdecl` должна гарантировать, что `esp` останется как был до вызова (т.е. аргументы со стека снимать нельзя; но можно их менять, там уже всем будет похрен).
Из интересного если класть на стек аргумент меньше 32-х бит, он всё равно дополнится до 32-х бит. Если аргумент 64-х битный, пушится 64 бита в порядке little-endian, то есть младшие биты в младших адресах (или младшие биты раньше на стеке).

Mangling Если вы хотите вызывать функцию из C++, у вас большие проблемы, потому что, как мы помним из курса C++, там у функций могут быть одинаковые имена и разные наборы аргументов, поэтому просто по имени функцию не найти. И тут у вас есть два варианта: либо повесить на функцию модификатор `extern "C"`, что заставит её называться ровно по имени, либо узнать, как она называется на самом деле, и использовать это имя. Например, вот так под gcc 14.2 выглядит `std::vector<int>::push_back(const int&)`: `_ZNSt6vectorIiSaIiEE9push_backERKi`. А вот так под MSVC 19.40: `?push_back@?$vector@HV?$allocator@H@std@@@std@@QEAAXAEBH@Z`. Мораль проста: если вы пишете библиотеку, наружу лучше вытаскивать функции с `extern "C"`, чтобы быть кроссплатформенным.
Из интересного: декорирование имён существует не только в C++, но и в C под 32-битной Windows, где вместо `main` у вас `_main`, вместо `printf` — `_printf` и т.п.

Пример с Hello world Как положить в память строчку символов? В nasm вот так:

```
format: db "abc", 0xa, 0
```

Инструкция db кладёт в память указанный набор байт (по адресу текущей инструкции). Конкретно в данном случае в память кладётся "abc\n" (с 0-терминатором, чтобы printf смог её вывести). И устанавливается метка format:, которая указывает на данную строку.
В каком месте лучше всего написать db? Ну, эту строчку мы менять не собираемся, и для неизменяемых данных как раз есть специальная секция section .rdata.
Так, что ещё. Если мы просто сделаем call printf, нам скажут, что не нашли printf, поэтому надо его объявить командой extern printf:

```
section .text
global main
extern printf
main:
push 123
push format
call printf
pop eax ; Можно просто двигать стек,
pop eax ; если вы знаете, куда его двигать
xor eax, eax
ret

section .rdata
format:
db "%i", 0xa, 0
```

Примечание: этот код не будет работать под 32-битной Windows, даже если заменить printf на _printf (об этом было выше). Дело в том, что с некоторым шансом вам подсунут стандартную библиотеку, в которой printf определён как inline-функция прямо в stdio.h, и поэтому ни в одной статической библиотеке его не найти.

stdcall Эта конвенция очень похожа на cdecl и отличается только одним: функция сама чистит стек от параметров. Например, функция int f(int a, int b) заканчивается командой ret 8 (напоминание: это значит, что после чтения адреса возврата со стека и прыжком по нему надо увеличить esp на 8). Это немного упрощает ваш код, но такая конвенция не позволяет делать функции с переменным количеством аргументов (ведь функция тупо не знает их количество).

pascal По сути stdcall, у которого аргументы поменяли местами.
Никому нахрен не нужен. Как минимум потому, что он говно (если у тебя не vararg, то зачем тебе чистить стек самому, а если vararg, то ты фиг ты найдёшь, где там первый аргумент). А ещё потому, что Pascal мёртв, а тот, что не мёртв (Pascal ABC.Net) — это на самом деле .NET.
В 16-битном мире ещё хоть как-то использовался, а сейчас больше нет.

thiscall, методы Похоже на stdcall, но это специальная конвенция по умолчанию (в майкросовтовских компиляторах) для методов. Так вот this передаётся в ecx.
Вопрос: а как методы класса работают, если им указать другую конвенцию? Ну, this добавляется нулевым аргументом. Но тут новая проблема: а если мы решили создать метод класса, который возвращает большой объект? Кто раньше будет this, а кто возвращаемое значение? А не договорились, разные компиляторы делают это по-разному. Отсюда вы вообще не хотите, чтобы методы классов возвращали что-то большое. Если на DirectX посмотреть, то там нигде никакой метод не возвращает то, что не поместится в аккумулятор.
Из ещё более грустного, вообще не всё равно в каком порядке эти аргументы положить. Намного лучше класть сначала возврат, потом this, потому что тогда вы сможете трактовать эту функцию как нормальную функцию из C (объявив ей первый аргумент явно). А возвращаемое значение как было норм, так и остаётся. Microsoft сделали хреново, все остальные — нормально.

fastcall Тут не договорились ни о чём вообще. Какие-то аргументы передаются через регистры в каком-то порядке, сколько и как — никто не знает. Не вытаскивайте это за пределы одного компилятора.

5 Best practices.

Как правильно писать if. Вот у нас в if есть что-то типа `eax > 5`. Чтобы сформулировать, как писать условие на ассемблере, надо сначала понять, как мы сравниваем: знаково или беззнаково. если это сравнение беззнаковое, нам надо использовать прыжки с `a` и `b`. Иначе — `l` и `g`. При этом если в if несколько условий, то написание `if`'а на ассемблере отличается в зависимости от того, как связаны условия.
Например:

```
if (eax > 5 && ebx) {
    // then-branch
}
```

трансформируется в

```
    cmp eax, 5
    jbe .lend    ; полностью равносильно jna .lend
    test ebx, ebx ; так традиционно сравнивать с нулём само число
    jnz .lend
    ; then-branch
.lend:
```

Заметим, что тут хорошо работает правило из C про ленивое исполнение. Это важно. Алгоритм может зависеть от факта, что второе условие не выполнят, если не надо.
Другой пример:

```
if (eax > 5 || ebx) {
    // then-branch
}
```

трансформируется в

```
    cmp eax, 5
    ja .lthen
    test ebx, ebx
    jnz .lend
.lthen
    ; then-branch
.lend:
```

Что делать, если есть else? Да всё просто, переходы на `.lend` заменить на переходы в `else`, а после конца `if` сделать безусловный переход на метку после окончания if'а.
Вы не хотите разрывать проверку от условного перехода. Например, потому, что современные процессоры умеют в macro-fusion. Обычно одна внешняя команда превращается в одну или несколько микроопераций, в зависимости от наличия железа для выполнения команды как одной. Например, `add eax, [ecx]` свободно может превратиться в две команды: чтение памяти и сложение. Из интересного, эти действия могут происходить в разное время. Так вот macro-fusion — это фича, которая позволяет превращать несколько команд в одну. В x86 такое типично может быть для сравнения + перехода.
Мы уже говорили про то, что порядок условных переходов иногда критичен для алгоритма. Но если нет, этим можно пользоваться. Если есть условный переход, который легко предсказывается, и который позволяет обойти остальные, то его имеет смысл вставлять в начало, если можно.
И ещё best practice. Пусть мы хотим сравнить `if (x >= 3 && x < 7)`, где `x` беззнаковый. Это отлично можно сделать за один переход.

```
    mov ecx, x
    sub ecx, 3
    cmp ecx, 4
    jae ecx .lend
    ; then-branch
.lend
```

Но тут вопрос в том, насколько хорошо это предсказывается. Если значения расположены случайно, один переход лучше, чем два. Более того, такой же прикол в C работает, можно явно написать `if (x - 3 < 4)`. Компилятор может и сам догадаться, а может не.
Что ещё интереснее, то же работает для знаковых типов. Главное, чтобы сравнение (на `<` либо `<=`) точно было беззнаковое. Поэтому на ассемблерах будет работать тот же код буквально, а в C, чтобы компилятор догадался, придётся написать пару букв: `if (x - 3u < 4u)`.
Ещё из интересного, можно прямо в ассемблере писать что-то типа `cmp al, '9' - '0'`. Оно будет делать то, что ожидается. Более того, можно просить его посчитать разницу между двумя метками. Нужно это бывает для того, чтобы длину строки куда-нибудь захардкодить (для этого мы берём адрес конца её минус адрес начала).
Интересный пример с шестнадцатеричными цифрами. Кажется, тут надо делать три проверки на диапазон (`'9' - '0'`, `'F' - 'A'` и `'f' - 'a'`). Но на самом деле, диапазон больших букв от диапазона маленьких отличается битом `0x20`. И он всегда установлен для цифр. Поэтому если мы точно знаем, что на входе цифры и шестнадцатеричные цифры, можно форсированно этот бит поставить, и цифры не изменятся, а большие буквы перейдут в маленькие.

Как писать циклы.

- Начнём с цикла `do { /* X */ } while (eax < 5u);`. Как выглядит на ассемблере? Ну, очевидно:

  ```
  .loop:
      ; X
      cmp eax, 5
      jb .loop
  ```

- Теперь `while (eax < 5u) { /* X */ }`. Окей:

  ```
      cmp eax, 5
      jnb .lend
  .loop:
      ; X
      cmp eax, 5
      jb .loop
  .lend
  ```

  То есть компилятор превращает while в do-while с условием в начале. Более того, если компилятор умеет доказывать, что цикл while исполняется хотя бы раз, он может это убрать.

- И, наконец, цикл for. `for (eax = 0; eax < ebx; eax++) { /* X */ }`. От while он отличается ничем по сути, и его можно переписать так:

  ```
  eax = 0;
  if (eax < ebx) {
      do {
          // X
          eax++;
      } while (eax < ebx);
  }
  ```

Конкретно в данном случае if можно убрать.
Но это ладно. Немногие знают, что лучше работают циклы в обратном порядке: `for (unsigned i = n - 1; i !=` Почему? Потому что `dec` отлично проставляет ZF (но не CF, это важно). Более того, macro-fusion отлично работает для комбинаций `dec` + `jz` и `dec` + `jnz` (и прочих условных переходов, если они не используют CF). И также работает сложение и вычитание с переходами. Более того, компиляторы умеют переворачивать циклы:

```
    mov eax, ebx
.L1
    ; X
    dec eax
    jnz .L1
```

Но эта штука идёт от 5 до 1, а не от 4 до 0. Надо починить. Ну, всё просто

```
    lea eax, [ebx - 1]
.L1
    ; X
    sub eax, 1
    jnc .L1
```

Это правильный вариант. Но есть ещё более крутой, если `eax` не очень большой.

```
    lea eax, [ebx - 1]
.L1
    ; X
    dec eax
    jns .L1
```

Но это ладно. Что если мы хотим двигаться вперёд? Можно двигаться вперёд к нулю. Типа такого:

```
    for (eax = -ebx; eax < 0; eax++) { /* X */ }
```

Тут надо пошаманить с `X`, но если мы там к массиву хотим обратиться, то всё вообще круто. Например, `X := ecx[eax]++`, где `ecx` — массив dword'ов.. Тогда делаем так:

```
    lea ecx, [ecx + ebx * 4]
    mov eax, ebx
    neg eax
.L1
    inc dword [ecx + eax * 4]
    inc eax
    jnz .L1
```

Важное примечание: везде вышенаписанный код опирается на то, что хотя бы раз условие цикла выполнится. Иначе сверху надо бахнуть ещё одно сравнение.

Как писать switch. Для начала: аргументом switch может быть только целое число. Веткой case — только константа времени компиляции. В честь чего? В честь того, что только так это можно скомпилировать лучше, чем в кучу if'ов. Возьмём такой switch:

```
switch (eax) {
case 1:
case 2:
    // X
    break;
case 3:
```

```
        // Y
    case 5:
        // Z
    }
```

Так вот, это отлично компилируется в такое:

```
section .text
    cmp eax, 5
    ja .lend
    jmp [table + eax * 4]
.L1:
    ; X
    jmp .lend
.L2:
    ; Y
.L3:
    ; Z
.lend:

section .rdata
.table: dd .lend, .L1, .L1, .L2, .lend, .L3
```

Короче, тут безусловный переход по табличке. Если значения разнородные (типа 1, 2, 4 и 100000), компилятор их распилит. А если они совсем рандомные, то будет куча if'ов. Возможно, с двоичным поиском.

6 Время и скорость.

Начнём с простого: вот вы нашли какую-то команду. Как понять, быстро ли она работает? Для начала определимся с терминами: latency и throughput. Первая — количество тактов от начала вычисления до получения результата. Вторая — сколько команд в единицу времени может исполнять процессор. Концептуально, первое о последовательности зависимых команд, второе — об одновременном количестве исполняемых команд.
Теперь к делу. Очевидно, в документации время вам никто не напишет (потому что в документации ISA, она про то, что команда делает), а время — это микроархитектура, про которую вам никто ничего не скажет.

rdtsc Можно вызвать команду `rdtsc` и её вариации. Например, `rdtscp`. Обе пишут в `edx:eax` время с момента reset в тактах. Вторая портит `ecx`, но её менее вольно можно переставлять в Superscalar'е. При этом кратность может быть больше одного такта. А ещё команда достаточно тяжёлая (около 100 тактов). Вообще зная тактовую частоту, можно пересчитать в секунды. А ещё незадолго после того, как процессоры научились разгоняться, таймер `rdtsc` отвязали от таймера процессора. У этой штуки точность равна сотням тактов, поэтому скорость одной инструкции не измерить. Впрочем, у способов, что даёт операционная система, точность — миллисекунды, так что если вам надо именно время для блока кода измерить, то на здоровье.

Профилировщики. Если Вы измеряете скорость кода и всё, то ладно. Но если вам нужно понять, почему она такая, надо ставить профилировщик. Причём лучше всего, чтобы профилировщик был от того производителя, что и процессор. Правда, у AMD профайлер не очень. Но зато там есть норм такое API доступа ко всяким встроенным счётчикам, которые могут помочь.
А ещё чтобы что-то профилировать, по-хорошему надо стабилизировать частоту процессора. На Windows такое может AIDA64 (как профилировщик она средняя, но там можно попросить выключить турбо). На Linux нужно как-то добраться до MSR и найти, какой флаг отвечает за турбо. Правда, стабилизация частоты, вероятно, сильно снизит скорость работы. Зато `rdtsc` будет более стабильным.

Скорость работы конкретной команды. Но то было про блоки кода. А если вы хотите узнать скорость одной команды, то тут совсем мрак. Проблема в том, что ассемблерная команда, во-первых, работает очень быстро, а во-вторых, суперскаляр может ее попортить, поэтому нужно создать необходимые условия для измерения. Еще скорость команды может определяться несколькими числами. Если мы будем измерять latency, то у команды умножения есть аж три результата, получаемые с разной скоростью: младшие биты, старшие биты и флаги. Как правило, младшая половина доступна пораньше.
Но вообще люди не дураки, и создали вот такую табличку, где можно почитать числа про интересующие вас инструкции для интересующего вас процессора.

7 FPU.

Оно же имеет название x87 и «математический сопроцессор». Это модуль работы с плавающей точкой. Изначально это был отдельный набор команд и отдельные микросхемы, которые назывались так же, как и процессор, но с семёркой на конце. Именно поэтому команды такие странные, потому что изначально это был отдельный сопроцессор, который общался с CPU через память.

7.1 Регистры.

Служебные мы тут обсуждать не будем, только общего назначения. Они называются `R0`, `R1` и т.д. до `R7`, но в таком виде их не применяют. А всё потому, что эти регистры организованы в виде стека. `R0` снизу, `R7` сверху. А указатель стека находится в определённом месте, и все обращения идут относительно него. А регистры назваются `ST0`, `ST1` и так далее. Прямо под указателем стека находится `ST0`, следующий за ним — `ST1`.
Все регистры — 80-битные. И в формате extended-precision они хранят число. При этом FPU умеют автоматически конвертировать типы данных. Они поддерживают целые числа **со знаком** (16, 32 и 64 бита), плавающую точку в форматах single, double и extended (кажется, уточнение для размера — `tbyte`) и BCD80. BCD80 (Binary Code Decimal) — это про десятичные цифры в байтиках. Каждые 4 бита хранят десятичную цифру.
И это всё ладно. Но самое плохое, что ячейки помечены как занятые или свободные. И попытка прочитать не занятую или записать в занятую производит NaN. Поэтому пользоваться стеком по кругу не получится.

7.2 Инструкции пересылки.

FLD, FILD, FBLD. Загрузка на стек (сместить указатель стека вниз, записать туда, в новое `ST0`, значение). Имеют один аргумент (адрес в памяти). `FLD` ещё умеет в `ST(i)`. А отличаются они тем, что `FLD` оперирует числами с плавающей точкой, `FILD` — целыми числами, `FBLD` — BCD80.

FLDZ, FLD1, FLDPI Пушат на стек константу (0, 1 и π соотвественно). Есть ещё несколько, но это можно загуглить. Из интересного, `FLDPI` пихает на стек не просто π, а с учётом текущего режима округления.

FST(P), FIST(P), FBST(P). Версии с `I` и `B` отличаются тем же, чем отличались `FLD`, `FILD` и `FBLD`. Стоит заметить, что `FIST` округляет в соответствии с текущим режимом округления. По умолчанию он к ближайшему чётному (а не к нулю).
Если без `P` — взять значение из `ST0` и поставить в память или в указанное место на стеке. Если с `P`, то записать и после этого освободить ячейку и увеличить указатель стека (если делать `FSTP` и в качестве аргумента дать ячейку стека, вам сначала запишут, а потом сдвинут указатель).

FXCH. Меняет местами вершину стека и ещё одно значение стека. Она, как и `mov`, выполняется за 0 тактов.

FCMOV$_{cc}$. Условная загрузка. Использует флаги процессора. Что они делают, Скаков не помнит, читайте документацию.

7.3 Инструкции арифметики.

Всегда: один аргумент — вершина стека. Другой — стек или память. Если команда обращается к памяти, то память — второй аргумент. При этом вершина стека может быть любым аргументом, но по умолчанию он первый.
То есть вы можете написать команду `FADD ST2, ST0` (эта команда добавит вам `ST0` к `ST2`), а ещё можете написать либо `FADD ST0, ST2`, либо `FADD ST2` (в зависимости от ассемблера, который вы используете). Эта команда добавит `ST2` к `ST0`.
Из интересного, в GNU-синтаксисе порядок аргументов такой же. Хотя во всех остальных командах (кроме сопроцессора) он обратный.

FADD, FADDP, FIADD. `FADD` — складывает и присваивает в первый аргумент. `FADDP` — складывает и делает pop со стека. `FIADD` — складывает `ST0` и целочисленную константу из памяти.
Никаких констант нет, очень жаль. Хотите константу, помимо предложенных — разместите её в памяти, читайте оттуда.

FSUB, FSUBP, FISUB; FSUBR, FSUBRP, FISUBR. Обычное вычитание берёт первый аргумент, вычитает второй и присваивает в первый. Обратное вычитание берёт второй аргумент, вычитает первый и присваивает в первый.

FMUL, FMULP, FIMUL. Понятно.

FDIV, FDIVP, FIDIV; FDIVR, FDIVRP, FIDIVR. Тоже понятно.

FPREM(1). Вычитает `ST1` из `ST0` не более чем 64 раза до тех пор, пока модуль разности не будет меньше модуля `ST1`. Версия с единичкой отличается округлением (без неё — к нулю, с ней — к ближайшему целому).
Смысл у этого всего есть, но небольшой. Обычно используется тогда, когда вы хотите вернуть значение в диапазон от 0 до 2π.

FABS. Присваивает в `ST0` его модуль.

FCHS. Меняет знак `ST0`.

FRNDINT. Округляет `ST0` до целого в соответствии с текущим режимом округления.

FSQRT. Берёт квадратный корень `ST0`.

FSIN. Берёт синус `ST0`.

FCOS. Берёт косинус `ST0`.

FSINCOS. Берёт синус и косинус (берёт значение со стека, кладёт на стек сначала синус, потом косинус).

FPTAN. Берёт тангенс `ST0`.

FPATAN. Считает `atan2`. То есть практически арктангенс `ST1 / ST0`. Снимает со стека обоих.

FCOMI(P). Сравнивает ST0 с ST(i) и ставит флаги ZF, PF, CF. Вариация с P снимает значение со стека.
Заметьте, что они не ставят SF или OF, поэтому после сравнения вы хотите использовать ja и jb, а не jl и jg.
Есть другие команды сравнения, они ставят флаги сопроцессора, и это вы не хотите.

FINCSTP, FDECSTP. Двигает указатель стека на единичку. Не меняет занятость.

FFREE. Пометить ST(i) как свободную ячейку.

FNOP. Делает ничего. Но не совсем. Он умеет бросить возникшее, но ещё не брошенное исключение. Да, IEEE-754 умеет в исключения. И железо это умеет, но надо включить это через служебные регистры. А вообще если исключение происходит, происходит системное прерывание (и гипотетически его можно обработать).

7.4 Конвенции вызовов с FPU (32 bit).

Аргументы single и double лежат на стеке процессора как число соответствующей битности. С extended сложнее.
Возвращаемое значение лежит в ST0. Поэтому конвенциям вызовов всё равно, возвращаете вы float, double или long double: всё равно это ST0.
При вызове функции стек сопроцессора должен быть чист, при возврате, если возвращаемого значения нет, то стек должен быть чист, если есть, то чист кроме ST0, где лежит результат.

8 x86_64 (оно же amd64 и x64).

В двух про режимы словах — процессор может работать в разных режимах: 16-битный, 32-битный и 64-битный. Что такое вообще "режим"? Это то, как процессор интерпретирует ваши байтики в виде команд. Например, меняются коды команд. Обычно оно меняется не сильно, но можно найти команду, которую выбросили или команду, которая закодирована иначе. Ради чего затеяли 64-битный режим? Ради того, чтобы можно было иметь 64-битные адреса. Вообще умные операционные системы могут оперировать больше чем 4ГБ памяти, даже если они 32-битные. Но честно выдать одному процессу больше 4ГБ было нельзя. Можно было сложными системными вызовами попросить операционную систему давать вам разные адреса, но это сложно и неудобно.
Нам это всё не так важно. Нам важно, что в наше время битность процессора — это битность адреса. Их интересного, на заре вычислительной техники битностью процессора называли количество бит, которыми он оперирует за раз. Например, то, что называлось 8-битным процессором, имело 8-битные регистры, но в память обращалось по 16-битной регистровой паре. В старой терминологии наши процессоры можно было бы назвать 512-битными (потому что AVX-512 умеет обращаться с 512-битными регистрами).

Регистры. Наши регистры расширили, уау! Теперь у нас есть rax, rbx, rcx и так далее по всем регистрам общего назначения. Это 64-битные регистры, 32-битные версии которых являются младшими битами этих. То есть eax — младшие 32 бита rax.
Также добавили новых регистров: r8, r9, ..., r15. Это 64-битные регистры. Если хочется, можно и к меньшим кускам обращаться: младшие 32 бита — r8d, младшие 16 — r8w, младшие 8 — r8b.
И последнее нововведение. Теперь можно написать sil, dil, spl и bpl: младшие 8 бит соотвественно si, di, sp и bp.
В виду особенностей кодирования доступ к ah, bh, ch и dh ограничен: в одной команде нельзя использовать одновременно их и новые регистры. Из интересного, это потому, что коды ah, bh, ch и dh забрали под dil, sil, spl и bpl.
И ещё особенность. Раньше, если вы писали в ax, то старшая часть eax не меняется. Тут ситуация

иная: если вы пишете в 32-битный регистр (любой), старшая часть обнуляется. Если пишете в меньшей размерности регистр, то всё по старому. На самом деле это то, что вы хотите, чтобы не создавать зависимостей при конвейерном исполнении, и чтобы не создавать длинные константы: если вы хотите загрузить в `rax` константу `5`, то вам не надо сохранять эту константу 64-битной. Можете сделать `mov eax, 5`, и у вас и в `rax` будет пять. Правда, так только с беззнаковыми числами работает.
Мораль проста: вместо `xor rax, rax` лучше писать `xor eax, eax` (первая команда на байт длиннее, потому что для кодирования дополнительных регистров добавили специальный байт перед командой, он называется REX и хранит этот бит и флажки).
А вообще из интересного, процессорам не очень вот такое:

```
xor eax, eax
mov al, [address]
; работа с eax
```

Процессоры не очень любят мержить значения разного размера. Вместо этого лучше писать такое

```
movzx eax, byte [address]
; работа с eax
```

Ещё интересности. До 64 бит расширили и служебные регистры: например, `RFLAGS` вместо `EFLAGS` и `RIP` вместо `IP`. И если в случае флагов нам это не важно, то в случае `IP`...
Ещё добавили `xmm`-регистров, но мы до них дойдём, и порезали сегментную адресацию, но до неё тоже дойдём.

Адресация . Адресовать можно вот такую штуку: (любой регистр, включая новые) + (любой регистр, кроме `rsp`, тоже включая новые) * (число из 1, 2, 4, 8) + (32-битная знаковая константа). Очень жаль, да, 32-битная. Поэтому важно, что константа знаковая, раньше такого вопроса не возникало, потому что модулярная арифметика.
Но ещё есть RIP-относительная адресация, можно адресовать `rip` + 32-битная знаковая константа. Здесь `rip` — адрес следующей инструкции. Зачем? Можно обращаться к своим константам. Обычно вы не можете обратиться к своим константам, потому что адрес у них 64-битный, блин. Но они обычно находятся не очень далеко от вашего кода, поэтому тут 32-битного смещения должно хватить.

Константы. Константы ограничены 32 битами не только в адресации, а вообще почти везде. Есть только три исключения, в которых можно 64-битные константы: `mov reg64, const64`, `mov acc, [const64]` и `mov [const64], acc`. Здесь `acc` — это `al`, `ax`, `eax` или `rax`.

Исчезнувшие команды. Например, выкинули команду `aaa` и всех её друзей. Эти черти про двоично-десятичные числа (т.е. о том, когда 4 бита кодируют десятичную цифру). Вы могли написать умножение, после чего написать `aam`, и оно скорректировало бы результат с точки зрения того, что умножение должно было бы быть в этом двоично-десятичном представлении.
Ещё выкинули `bound` и `into` — тоже какая-то нахрен не нужная срака, например `bound` — проверка того, что значение находится в указанном диапазоне, и если оно не, то прерывание. Понятно, это штуку тоже давно никто в здравом уме не использовал.
Ну, а фиг ли. Команды никому не нужны, а коды под новые команды откуда-то взять надо.
А из команд, о которых мы что-то знаем, выкинули `pusha` и `popa`. Очень даль, больше без попы.
Ещё убрали короткие формы команд `inc` и `dec`. Что значит «короткие формы»? А вот что: у `inc` была полная форма (которая умеет во всё, что угодно), а была ещё краткая форма, которая только для регистров, и плюс её был в том, что она один байт. Теперь эту отдельную форму убрали (делать `inc` регистру всё ещё можно, просто команда не будет такой короткой), и её код переназначили под REX. Поэтому будете дизассемблировать 64-битные инструкции не с теми флагами — можете получить код с большим количеством `inc` и `dec`, хотя на самом деле там REX. То есть повторю: вы всё ещё можете делать `inc` по регистру, просто эта команда не будет занимать один байт.
И убрали `sysenter` и `sysexit`, но это мы уже знаем.

Соглашения о вызовах. Их две, и они, к сожалению, привязаны к ОС. То есть конвенций одна под каждую ОС.
Общая часть: в момент вызова функции (т.е. перед `call`) ваш стек должен быть кратен 16. Обычно он кратен 8, потому что вы пушите по 64 бита, потому что `push al` всё равно пушит 64 бита. Но перед вызовом он должен быть кратен 16 (связано это с векторизацией, это увидим). Из этого следует, что сразу в начале функции вы не можете сделать вызов другой функции.

Microsoft (fastcall64). Аргументы функции вначале передаются в регистрах `rcx`, `rdx`, `r8`, `r9` или `xmm0`, `xmm1`, `xmm2`, `xmm3`. Они в 64-битном мире используются через `XMM`. То есть если ваша функция имеет такую сигнатуру: `void foo(int, float, int*, int)`, аргументы будут в `rcx`, `xmm1`, `r8`, `r9`. Всё остальное кладётся на стек в обратном порядке, как положено.
К сохраняемым регистрам относятся `rbx`, `rbp`, `rsi`, `rdi`, `r12`, `r13`, `r14`, `r15`, `xmm6`, ..., `xmm15`. Необходимость сохранять `xmm6`, ..., `xmm15` — это мрак и тупость со стороны того, кто это проектировал. Увидим, почему так, когда дойдём на векторизации.
Возвращаемое значение: `al`, `ax`, `eax`, `rax` или `xmm0` (числа с плавающей точкой через `xmm0`, а не `st0`). Большие значения, как обычно, через первый аргумент, если там `this`, то всё как раньше в Microsoft (т.е. сначала `this`, потом возврат).
А ещё тут есть фича: между адресом возврата и пятым аргументом вам вставили 32 байта, которые называются shadow space, и в которую вы можете внутри функции использовать как угодно. Зачем? Чтобы вы могли взять ваши 4 аргумента в регистрах, засунуть их в shadow space и иметь непрерывную последовательность аргументов. А если вам не надо, используйте как хотите, это ваши 32 байта.
По-хорошему ещё вы должны внедрять в исполняемый файл специальную метаинформацию. Просто если ваш код не кончится с исключением, вас никто не поймает, если вы не внедрите её. А если случится, вашу программу убьют за отсутствием метаинформации. Вот тут можно почитать.

Все остальные. Эта конвенция никак не называется, но мы будем называть её Unix64.
Аргументы: тоже в начале в регистрах, потом на стеке. В регистры в следующем порядке: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` и `xmm0`, ..., `xmm7`. С некоторыми комментариями: если делать `syscall`, то `rcx` перемещается в `r10`. Если vararg-функция, то количество `xmm`-регистров надо указать в `rax`. Причём можно использовать меньше `xmm`-регистров: можно передать три вещественных числа, засунуть на стек все и записать в `rax` ноль. А ещё обратите внимание на то, что тут «и», а не «или». То есть если вы предаёте `int`'ы и `float`'ы в рандомном порядке, то ниоткуда вы не узнаете, в каком именно, вам запишут целочисленные аргументы в первую группу (подряд), а вещественные — во вторую группу. Очень весело жить.
Сохраняемые регистры: `rbx`, `rbp`, `r12`, `r13`, `r14`, `r15`.
Возвращаемое значение: `al`, `ax`, `eax`, `rax`, `rdx:rax` или `xmm0` или `xmm0` и `xmm1`. Зачем `xmm1`, Скаков точно не помнит, насколько помнит, чтобы можно было вернуть `complex double`. Большие значения, как обычно, через первый аргумент, если там `this`, то всё как раньше (т.е. сначала возврат, потом `this`).
Тут есть другая фича: red zone: вам уже за возвращаемым значением дают 128 байт. **Обычно** писать данные ниже `rsp` не хорошо, их сохранность никто не гарантировал. Например, если там отладчик сидит и туда сохраняет свои данные. Но вообще не важно, кто у вас сидит, просто никто не гарантирует сохранность данных. Хотите хранить — опустите stack pointer, дальше работайте. Так вот red zone гарантирует, что вам 128 байт данных никто не испортит. Хотите сохранить 128 байт данных у себя (и не хотите никого вызывать) — можете не опускать стек.

Интересность. Из интересного, в x64 модно не делать `push`, а потом `call`, а модно в начале функции опустить стек ровно на столько, сколько максимум понадобится класть на стек (с учётом shadow space, если надо), при вызове других функций делать `mov`, а в конце вашей функции поднять стек обратно, на сколько опустили.

9 SIMD.

9.1 Мотивация, intrinsic, прочие мелочи.

Как мы помним, это single instruction multiple data. Отличная вещь, которая ускоряет ваш код, одна лишь беда: компиляторы с очень большим трудом умеют их применять. И если вы хотите чтобы ваш код был с SIMD, то у вас по сути три опции: либо написать ассемблерную вставку, либо писать код так, чтобы компилятор догадался применить SIMD (т.е. смотреть на вывод компилятора, и если у него не получилось в SIMD, то переписывать код). И есть третий вариант: использовать расширения компилятора и intrinsic'и. Из интересных расширений компилятора, советую почитать clang extended vectors — это фактически OpenCL вектора, которые по причине общности LLVM middle-end'а можно использовать в C/C++. И вам с большой вероятностью сгенерируют SIMD. Это всё ещё завязывается на конкретный компилятор, но лучше, чем завязываться на MSVC, например. Intrinsic'и же — это штука, которая с точки зрения C/C++ выглядят как функции, но на самом деле обёртка вокруг ассемблерных команд. В x86 они практически полностью общие, потому что Intel их задаёт в документации, поэтому их в принципе поддерживают.

9.2 MMX.

Это расширение мы рассмотрим подробно, чтобы была понятна идея, стиль написания и т.п., а другие расширения рассмотрим обзорно.

Регистры. Там регистры с `mm0` по `mm7`. Они 64-битные. Под эти 8 регистров отдали младшую часть регистров с `r0` по `r7`. Это, напоминание, 80-битные регистры сопроцессора x87. Адресация к `mm0`..`mm7` прямая, без stack pointer'а. Использовать вместе x87 и MMX не удастся, увы. Что неприятно. В 32-битным мире от сопроцессора вообще никуда не убежать (там по конвенциям возвращаемое значение в `ST0`), в x64 лучше, но использовать его вы всё равно можете, если хотите. Чтобы вам ещё меньше хотелось использовать MMX и x87 вместе, все команды MMX, кроме одной, помечают весь стек сопроцессора как занятый.

Команды. У любой команды, кроме `movq` и `modv` первым аргументом может быть только регистр. Вторым может быть регистр или память.

- `movq` пересылает данные из одного `mm(i)` в другой или из `mm(i)` и памятью. В 32-битном мире переслать данные между `mm(i)` и обычным регистром нельзя (битность не совпадает), а существует ли способ на x64 — сложный вопрос. Без констант, увы.
- `movd` — пересылает младшие 32 бита между `mm(i)` и регистром общего назначения/памяти. Если вы пишете в память или регистр общего назначения, то всё хорошо. Если в `mm(i)` — старшая часть обнуляется. Тоже без констант, увы.
- `packss(dw|wb)` (это тактовать как команды `packssdw` и `packsswb`). Рассмотрим `packssdw mm0, mm1`. Происходит следующее: `mm0` и `mm1` трактуются как пары знаковых dword-ов. После чего они преобразуются в word-ы при помощи signed saturation. Это значит с насыщением, то есть числа больше 32767 превращаются в 32767, а числа меньше -32768 — в -32768. Потом эти четыре word-а кладутся обратно в `mm0` с порядке

  1. Старшие 32 бита `mm1` идут в старшие 16 бит `mm0`.
  2. Младшие 32 бита `mm1` идут в следующие 16 бит `mm0`.
  3. Старшие 32 бита `mm0` идут в следующие 16 бит `mm0`.
  4. Младшие 32 бита `mm0` идут в младшие 16 бит `mm0`.

  `packsswb` делает то же самое, но с байтами. То есть входные регистры трактуются как четыре word'а, сжимаются до байтов и упаковываются как 8 байт в первый аргумент.

- packuswb — знаковое насыщение. То же самое, что и packsswb, но сжатие беззнаковое. То есть входные word'ы считаются знаковыми всё ещё, но тут их сжимают с насыщением в байты не от -128 до 127, а от 0 до 255. На самом деле эта команда уже очень полезна для обработки изображений, потому что обычно вы считаете в большей размерности, и вам потом приходится сжимать, причём с насыщением. И без этой команды вы бы делали это ветвлениями, что медленно. А так она даже без параллельности очень полезна.

- punpck(h|l)(bw|wd|dq). Пример: punpcklwd xmm0, xmm1. Пилит xmm0 и xmm1 на слова, отбрасывает старшую половину слов, и кладёт в старшую часть результата старший кусок xmm1, потом старший кусок xmm0, потом младший кусок xmm1, потом младший кусок xmm0.

- padd(b|w|d). Пилит входные регистры на куски соответствующего размера, попарно складывает (в модулярной арифметике), записывает в соответствующую часть результата.

- padds(b|w), paddus(b|w). Как предыдущее, но сложение с насыщением. В первом случае — знаковое насыщение, во втором — беззнаковое.

- psub(b|w|d), psubs(b|w), psubus(b|w) — вычитание, с теми же семантиками, что раньше.

- pmullw — как paddw, но умножает. Записывает в ответ младшую часть результата. pmulhw — записывает старшую часть результат умножения (со знаком). (Если хочется собрать себе нормальное полное умножение, после этого делаем unpack, он делает что хочется.) pmaddwd. Попарно полно умножает (со знаком) 16-битные куски регистров, берёт полное произведение, складывает два старших произведения и два младших произведения, записывает их как 32-битные числа в результат. Зачем этот мусор можно использовать? Например, для «горизонтального сложения» — сложения пар соседних word'ов в одном регистре. Загрузите 0x1010101 в регистр и засуньте вторым аргументом.

- pcmpeq(b|w|d), pcmpgt(b|w|d). Сравнивает на равенство либо на больше со знаком байты/word'ы/dword'ы. Если равно, соответствующий кусок регистра устанавливается в минус единицу, иначе в ноль. Так можно легко заполнить регистр всеми единицами: сделать ему pcmpeq с самим собой.

- pand, por, pxor. Побитовое и/или/хог. Ещё есть pandn — инвертирует первый аргумент и делает побитовое и со вторым.

- pssl(w|d|q), psrl(w|d|q), psra(w|d). Принимает регистр и константу. Делает сдвиг влево/вправо логический/вправо арифметический соответствующим кусочкам регистра.

- emms. Помечает весь стек сопроцессора пустым. Зачем? Чтобы перед вызовом функции либо после возврата сделать её, потому что конвенция вызовов этого требует.

Это буквально все команды MMX. Но их обычно хватает. Тут нет деления, но это ничего, для него обычно используют фиксированную точку. А вот чего реально плохо — отсутствие сдвигов байтов.
А ещё есть универсальная боль любого SIMD-программирования — это обработка краёв.

9.3 Больше MMX.

Больше команд MMX (и их друзей) добавили вместе с расширением с SSE:

- pmax(ub|sw), pmin(ub|sw) — минимум и максимум по байтам без знака или словам со знаком.

- pavg(b|w) — среднее арифметическое по байтам или словам без знака. Округляет вверх. Ура, можно делить байт на два (правда, округление не туда).

- pinsrv, pextsrv. Первая принимает регистр, память и фиксированное число, второе сначала память, потом регистр, потом число. pinsrv вынимает 16-битное число из памяти и засовывает в регистр по с заданному номеру (0 — младшие, 3 — старшие). pextsrc делает пересылку обратно.

- pmulhuw — ясно, кто.

- psadbw — побайтово (математически) вычитает байты без знака, считает модуль, просуммировать, записать в младшее слово.
- pshufw — позволяет переставить произвольным образом слова при копировании из регистра в регистр или из памяти в регистр. А именно `pshufw dst, src, idx` делает следующее:

```
dst[0] = src[idx01];
dst[1] = src[idx23];
dst[2] = src[idx45];
dst[3] = src[idx67];
```

  То есть в младшее слово `dst` сохраняются то слово `src`, номер которого записан в младших двух битах `idx`. И аналогично с другими словами.
- pmovmskb — взять старшие биты каждого байта, составить из них единое число, записать в регистр общего назначения.
- maskmovq — записать в edi только те байты первого аргумента, для которых во втором аргументе в этом байтике старший бит единица. Иначе оставить старые байты. Работает долго капец.
- movntq — записать в память, минуя кэш. Например, мы обрабатываем большой объём данных, которые используем один раз. Но увы, после неё кэш и память десинхронизированы.
- sfence — синхронизация movntq. То есть применение такое: делаем делаем много раз movntq, после этого sfence.
- `prefetch(T0|T1|T2|NTA)`. Изначально это было в 3DNow!, там были prefetch и prefetchw. Они оба работают как nop. Это **подсказка** кэш-подсистеме закэшировать эту область памяти. prefetchw подсказка закэшировать на запись. Вместо этого Intel сделали свой prefetch, который грузит в какие-то кэши в зависимости от конкретной команды.
  - `T0` — во все кэши.
  - `T1` — в кэши уровня 2 и выше.
  - `T2` — в кэши уровня 3 и выше или на выбор имплементации.
  - `NTA` — в не-темпоральные кэши.

  Раньше это чудо работало божественно, пихаешь `prefetch` и живёшь счастливо. Но тут есть проблема. Префетчить надо то, к чему будете обращаться через некоторое время. Например, то, к чему будет обращение через несколько итераций цикла. А через сколько? И тут это хрен посчитаешь: если слишком мало, он не успеет подгрузить кэш. Если слишком много, данные уже выгрузят. И тут в дело входят аппаратные префетчеры, которые стоят в современных процессорах и сами занимаются префетчингом. Если ваш код — это тупо цикл, то аппаратный префетчер сам догадается, насколько заранее надо делать `prefetch`. Впрочем, если вы обходите дерево в каком-то вам одному известном порядке (притом заранее известном, чтобы знать, что заранее префетчить), тут аппаратные префетчеры не помогут, и тут уже будет полезно написать `prefetch` самому (предварительно протестировав, насколько заранее его писать).

9.4 SSE.

Когда AMD склепали 3DNow! (расширение позволяло работать с `mm`-регистрами, как с двумя флотами), Intel это увидели и сделали расширение, в которых расщедрились на новые 8 регистров: с xmm0 по xmm7. Это 128-битные регистры, которые всегда интерпретируются как 4 флота. И они новые, никак не пересекаются с `mm`-регистрами.
Многие команды там имеют формы `ps` и `ss`. Первая форма векторная, она делает операцию со всеми 4 значениями. Вторая форма делает операцию только с младшим флотом, а остальные оставляет.

9.5 SSE2.

Там не столько добавии команд, сколько разрешили интерпретировать `xmm`-регистры как два дабла (суффиксы `pd` и `sd`). Также разрешили MMX-команды применять к `xmm`-регистрам, чтобы обрабатывать порции большей битности.
Но тут есть гадкий подарочек. Как загружать туда данные? Так: `movaps`, `movups`, `movdqa` и `movdqu`. Почему их 4? Команды с `ps` — пересылка значения как будто оно с плавающей точкой, в версии с `dq` — пересылка целого значения. Смысл в том, что если вы мешаете команды с целыми числами и с плавающей точкой, то могут возникать задержки (потому что в процессоре целочисленные и вещественные операции могут работать в разных доменах). Но это не главная проблема. Команды с `u` — невыровненное обращение к памяти, а с `a` — выровненное. Если вы пересылаете из регистра в регистр, то пофиг. А если один из аргументов — память, то если адрес не кратен 16, команды с `a` падают. И из интересного, так же работают все MMX-команды, когда они 128-битные. Так что либо выравнивайте адреса, либо используйте `movups`, и работайте с регистрами.

9.6 Кратко про AVX.

Там вы увидите у команд префикс `v`. Он там зачем? А он там затем, что изначально AVX трактовал новые `ymm` регистры как пару из двух `xmm`. Но мы знаем, что Microsoft сказали, что несколько `xmm`-регистров должны сохраняться. И проблема в том, что только `xmm`, а старшие байты `ymm` — нет. Поэтому сделали вам два набора инструкций: без `v` и с ним. В первом процессор сохраняет старшие части `ymm`-регистров. Во втором обнуляет. Первое — дорого по скорости. Поэтому чтобы это извращение не делать, у процессоров есть два режима, как работать с больше чем `xmm`. В первом — они знают, что старшие 128 бит во всех `ymm` — нули. Тогда команды просто `xmm` ничего не должны мержить. Во втором режиме страдает и мержит. И если в первом режиме обращаться на запись к `ymm` или `zmm`, вы перейдёте во второй режим. И если во втором режиме выполнять команды без `v`, они начнут безбожно тормозить.
Отсюда мораль. Во-первых, если вы работаете не только с `xmm`, используйте только команды с `v`. Во-вторых, если вы уже работали с `ymm` и `zmm`, в конце вычислений используйте `vzeroupper` — обнуляет всё, что старше 128 бит у всех `ymm` и `zmm`. Это заставит процессор перейти в нужный режим работы, и после вашего кода ничего не будет тормозить у того, кто в следующем куске кода использует только `xmm` и без `v`.

9.7 Интересный факт про страничную адресацию.

Операционные системы выделяют и раздают память страницами. Дальше уже `malloc` может как-то изворачиваться из этого, но по правам всё просто: если вам доступна какая-то область страницы, вы можете легко прочитать любое место этой страницы. В частности, это значит, что если вы делаете выровненное обращение к памяти с SIMD, и хотя бы один из байтов вам доступен, то вам доступны все.

10 Режимы работы процессора.

10.1 Реальный режим (real mode).

Изначально первый i8086-процессор работал в одном режиме, который никак не назывался, но потом его назвали real mode. Что это? Напомним, что у него был мегабайт адресного пространства (20 бит адресов) и 16-битные регистры. Как обращаться по 20 битам? Сделали сегменты, и была сегментная адресация. `mov al, [bx]` позволяет адресовать только 16 бит. И для 20 бит вам завезли отдельные регистры (сегментные регистры): `cs` (code segment), `ds` (data segment), `es` и `ss` (stack segment). Они используются только для расширения адреса до 20-битных значений. И на самом деле `mov al, [bx]` — сокращённая форма для `mov al, [ds:bx]`. Первая часть адреса называется сегмент, а вторая — смещение. Работает это как обращение в память по адресу `ds * 16 + bx`. Это значит, что адресация не уникальна. Именно отсюда, кстати, тот факт, что сравнение указателей в C и C++ — не total order. Из интересного, тут чуть больше одного мегабайта. Но если мы попытаемся так обратиться, будет

переполнение. И это долгое время даже поддерживалось, в том смысле, что в BIOS была опция Line A20 Enable (включить двадцать первую линию адреса). Если включить эту линию, но всё ещё переключиться в real mode, то обращаться будет без переполнения, и памяти будет чуть больше, чем мегабайт. В сегментные регистры можно нормально сделать `mov` из регистра общего назначения (или обратно). Константу и память туда, кажется, писать нельзя. Из интересного, можно прям явно написать `mov al, [cs:bx]`.
Так а зачем их четыре штуки? А затем, что по умолчанию вам для обращения в память ставят какой-то сегмент по умолчанию. Для переходов вам делают по умолчанию `cs`. Если вы делаете `call`, `push` или любую другую операцию со стеком, там используется `ss`. Если вы нормально обращаетесь в память, то тут зависит от того, какой адрес. Если в в адресе есть `bp`, то это `ss`. Иначе — `ds`. Зачем? Потому что `bp` — ваш единственный способ обратиться в стек в real mode (вспоминайте адресацию). Потому что `bp` использовался для такой штуки как stack frame (смотрите конспект по плюсам). `es` же встречался в качестве сегментного регистра по умолчанию довольно редко, например в качестве destination команды `movsb`.
В чём беда? Полное отсутствие разграничений доступа. Любой лох может обращаться ко всему мегабайту. Поэтому сразу стало понятно, что нам нужна аппаратная поддержка разграничений доступа. И в i80286 добавили новый режим.

10.2 Защищённый режим (protected mode).

Тут полностью меняется интерпретация сегментных регистров. Они всё ещё 16-битные и участвуют в адресации. Но это 16-битное значение — вообще не умножение на 16 и сложение. Эти 16 бит разбиваются на три части. Старшие 13 бит — индекс в специальной таблице в памяти. На эту таблицу указывает регистр, который никто не помнит, как называется, потому что дальше будет массовое переименование, и этот регистр будут именовать `GDTR`. Эта таблица содержит структуры, описывающие сегмент. Там адрес начала сегмента, лимит (т.е. ограничение от 0 до сколько по нему можно обращаться) и флажки. Например, поле прав. Или флажок, который говорит, что сегмент растёт с конца, что полезно стеку.
С точки зрения привелегий существуют четыре «кольца». Нулевое кольцо — полные права (исполнять можно любую хрень). Третье колько — минимальные права (пользовательский код). Остальные два кольца никогда не используются, поэтому нам не важно, что там можно. Младшие два бита сегментного регистра — уровень привилегий. Оно называется RPL (requested privilege level) для всех регистров, кроме `cs`, а для него называется CPL (current privilege level). Это самое CPL — те права, с которыми вы сейчас работаете. Если вы пытаетесь записать значение в `cs`, то там проверяют, что вы не пишете в CPL число меньше (т.е. больше привилегий), чем было. Иначе — это аппаратное исключение.
Хорошо, а как тогда переходить в ядро? Нужно же повысить себе права. Как это сделать? Изменение прав можно повесить на обработчик прерываний. По системному таймеру происходит табличка прерываний, и в этой табличке есть не только `ip`, но и `cs`. Обработчику прерываний похер, он может менять CPL как ему вздумается. А появившиеся потом `sysenter` и `syscall` сразу переключают кольцо на ноль.
Оставшийся один битик — переключение GDT и LDT. Если этот бит ноль (и используется GDT), то всё, что описано выше. Иначе есть регистр LDTR, который хранит сегментированный адрес локальной таблички дескрипторов. Разные процессы будут иметь разные таблички дескрипторов, которые можно переключать для каждого процесса. Это активно использовалось в третьей винде, и больше нигде. Если вы хотите эмулировать третью винду, удачи вам, потому что в 64-битных процессорах это всё вырезали.
Звучит офигенно, пока вы не замечаете одну забавную особенность. Весь ваш софт написан под real mode. Как с операционкой под protected mode запускать код под real mode? Официальный ответ: никак. Вы можете из реального режима запустить защищённый, но не обратно. Обратно только перезагрузив машину. Потом народ раскопал, что Intel сделали специальную недокументированную команду `loadall`, которая загружает полное состояние регистров из памяти (в том числе и того, в каком режиме мы). В i80386 была аналогичная команда, но с другим кодом, а потом её убрали.

10.3 Переход в 32 бита.

В i80386 официально разрешили делать переход из защищённого режима в реальный. А именно сделали регистр CR0, в который можно сделать присваивание, и определённый битик переключит вас между режимами. Делать это можно с правами нулевого кольца, естественно.
Но ещё защищённый режим разделился на два разных — 16-битный и 32-битный. Что за 32? В дескрипторе сегмента появился дополнительный битик, который позволяет сказать, что наш сегмент 32-битный. В первую очередь, нас интересует сегмент кода. Если он 32-битный, то opcode интерпретируются процессором как обращающиеся к 32-битным регистрам. Если поставить перед командой префикс `0x66`, то команда переключится на противоположную. В 16-битном режиме (любом из двух), команда станет 32-битной. В 32-битном режиме команда станет 16-битной. Сегменты почти не поменялись идейно, но команды теперь такие.
Как определяется сегмент по умолчанию теперь? Он всё ещё определяется исключительно базовым регистром. Если базовый регистр — `ebp` или `esp`, то ваш сегмент — это `ss`. Иначе `ds`. А теперь мастер-вопрос, кто тут база: `[ebp + eax]`? В nasm и yasm эту подставу можно решить так: `[ebp + eax * 1]`. Всё понятно, тут `ebp` — это база, а `eax` — это не база.

10.4 «Нереальный режим».

В защищённом режиме можно настроить лимит сегмента в 4Гб, загружаете во все сегментные регистры селектор на такие сегменты (и вам в невидимую для вас его часть запишут лимит в 4ГБ), возвращаетесь в real mode. Когда вы пишете в сегментный регистр уже в real mode, перестраивается только база, а лимит остаётся как был. Поэтому вы сидите на real mode, и у вас здоровые лимиты. Существует версия даже, что именно в таком режиме процессоры стартуют. Очень жаль, правда, у вас `ip` маленький :(.

10.5 V86.

Это аддон к защищённому режиму. Там адресация работает как в реальном режиме (теневая часть записывается по правилам реального режима), но оно работает с правами третьего кольца. Что это? Удобный режим для запуска старых программ. Защищённая операционка вместо запуска старого кода в real mode, запускает в V86. Как только программа сделает что-то, что запрещено системой привилегий (например, обратиться к внешнему устройству через порты ввода-вывода), программа падает, исполнение переходит в обработчик и ОС смотрит, что происходит. И после этого проэмулирует происходящее программно.

10.6 Страничная адресация.

Это вы помните, трансляция логических адресов в физические. Это дополнительный этап, который работает после сегментной адресации. Сначала всё чекается по сегментной системе, и получается «линейный адрес», который проходит через страничную адресацию (которую так-то можно и выключить где-то в регистре CR4, скорее всего; битик должен называться PG).
Базовый размер страницы 4Кб, и хрен куда ты это поменяешь. В i80386 режиме была двухуровневая табличка, её корень хранится в регистре CR3 (там линейный адрес), первый уровень транслирует старшие 10 бит, следующий уровень — следующий 10 бит, и всё у вас хорошо.
В районе Pentium у процессора появилась такая штука как Huge pages — уже на первом уровне может быть не указатель на подтабличку, а сразу страница размера 4Мб.
В Pentium Pro осознали, что 4Гб оперативки мало, и там были физические адреса 36-битные. Точнее, можно было использовать старую модель, а можно было включить Physical Address Extension, где вот такое. И тогда один элемент таблички трансляции занимает 8 байт. А значит в одной таблице 512 элементов, и одна страница описывает 9 бит адреса. И на самом деле уже тут появились три уровня, где первый уровень мелкий. Да, 64Гб, на 32-битной системе. Миф о том, что выше 4Гб нельзя возник от не-серверной Винды, где запретили на уровне лицензии выше 4Гб сначала оперативки, а потом адресного пространства. Понятно, что теперь Huge pages будет не 4Мб, а 2Мб.
В очень современных процессорах, можно сделать очень Huge pages, будет страница в гигабайт.

10.7 Мелочи.

В какой-то момент (то ли в i80286, то ли в i80386) добавили ещё два сегментных регистра, `fs` и `gs`. Нигде они не по умолчанию, просто под свободное пользование.

10.8 x86/64. Long Mode.

AMD сделали новый режим работы: long mode. Внутри него есть compatibility mode 16 и compatibility mode 32, а так же 64-bit mode. Перейти можно из защищённого режима в compatibility mode (и обратно), и можно из compatibility mode в 64-bit mode и обратно.
Режим совместимости очень похож на защищённый режим. С точки зрения пользовательского кода это буквально то же самое. Никакого расширения регистров не видно. Чтобы переключить режим совместимости на защищённый (и обратно), требуется нулевое кольцо. А чтобы перейти из 64-bit mode в compatibility mode никаких прав не надо, а требуется только загрузить правильный селектор в `cs`.
Все команды, которые урезали из 64-битного режима, урезали только из 64-bit mode. Но урезали не только команды, но и существенную часть сегментов. Потому что современные операционки не хотят использовать сегменты. От сегментной модели используются только права (CPL) и регистры `fs` и `gs`, которые используются для thread-local-переменных. Таблица сегментов не нужна, она вообще скорее всего одинаковая для всех программ. Вместо неё страницы.
Итого от регистра `cs` осталось только CPL и битность (16/32/64). Всё, база ноль, лимит максимум. В сегментах `ds`, `ds`, `es` осталось ничего, всем похрен, что вы с ними делаете. А вот от `fs` и `gs` осталось поле базы сегмента. Причём в табличке осталось 32-битное. Хотите нормальное — идите в machine-specific registers, настраивайте там.

10.9 WOW64.

Эмуляция 32-битных приложений на 64-битной винде. Увы, без поддержки DOS-приложений, потому что Long Mode не имеет V86 для compatibility mode, поэтому real mode эмулировать аппаратно нельзя. То что выкинули поддержку 16-битных приложений под третью винду — разумного объяснения нет, можно было бы и поддержать.
Сначала всё стартует с 64-битным nt.dll, но если это WOW64, то вам ещё грузят 32-битный nt.dll и ещё нужные 32-битные библиотеки. И когда что-то надо, вы попадаете в 64-битный nt.dll, который уже делает переход в 32-битный nt.dll. Если очень хочется, можно даже из 32-битного кода загрузить себе нужную вам 64-битную dll через nt.dll.