

# CdM-16

Д.В. Иртегов

Н.Н. Репин

и др.

# Что такое CdM-16

- 16-битный процессор (платформа уровня 2), существующий в трех формах
  - Программный эмулятор на Python
  - Микросхема Logisim  
(в настоящее время есть проблемы с тактовой частотой)
  - Программный эмулятор на Java, пригодный для использования в качестве микросхемы Logisim  
(в процессе, но уже на достаточно продвинутых стадиях)
- Можно использовать в проектах курса «Цифровые платформы»
- Или в каких-нибудь еще

# Почему еще полезен CdM-16?

- Сравнение CdM-16 и CdM-8[e] позволит вам понять некоторые проблемы и вопросы, с которыми вообще сталкиваются разработчики ISA (Instruction Set Architecture)
- Казалось бы, какие проблемы могут быть при разработке виртуального процессора для эмулируемой среды?
  - Что нам стоит дом построить, нарисуем – будем жить!
- А вот не так-то все просто.
- Например, переделка PC в 16-битный регистр в CdM-8e создала целый ряд... ну, не то, что проблем, но, скажем так, алогичностей в архитектуре.

# Что означает «16-битный»?

- 16-битные регистры общего назначения
- Аппаратная 16-битная арифметика
- 16-битный интерфейс с оперативной памятью
- 16-битное адресное пространство
  - 64кб памяти в манчестерском (фон-неймановском) режиме
  - 64кб памяти кода и 64кб памяти данных в гарвардском режиме
- Большинство команд 16-битные (есть несколько 32-битных)

# Что еще может означать «16-битность»?

- В документации по другим процессорам вы можете увидеть другие трактовки 16-битности
  - Например, ARM имеет подмножество команд Thumb, которое называют 16-битным, потому что там большинство команд имеет длину 16 бит.
    - Но эти команды манипулируют 32-битными регистрами и адресуют 32-битное адресное пространство
  - Процессор i386SX часто называли 16-битным, потому что он имел 16-битный интерфейс с оперативной памятью,
    - Но он также, как и Thumb имел 32-битные регистры и адресное пространство, а команды у него переменной длины

# Что вообще означает «разрядность»?

- Раскрою небольшой секрет
- Есть разрядность процессора (ширина тракта данных/АЛУ/регистров)
- Есть разрядность ISA (системы команд)
  - Разрядностью ISA можно называть длину команды, как в ARM/Thumb
  - Но чаще разрядностью ISA называют длину адреса (указателя)
  - Для операционной системы важна длина адреса (определяет режим работы диспетчера памяти)
- Разрядности CPU и ISA могут не совпадать
- Почти все современные CPU (Intel/AMD x64, ARMv9, IBM System/Z) 64-разрядные
- Но все они могут исполнять 32-разрядные программы (x86, ARMv8, System/370)

# Чем CdM-16 лучше, чем CdM-8?

- В 256 раз больше памяти (без всяких банков!). По объему памяти, система сравнима с «нормальными» компьютерами, такими, как DEC LSI-11/Электроника-60 или ZX Spectrum
  - Только проблема со скоростью. Тактовые частоты LSI-11 или Z80 измерялись мегагерцами, в Logisim вы не можете поднять ее выше 5кГц.
- 8 регистров общего назначения – почти во всех подпрограммах, которые вы пишете, все скалярные значения можно разместить в регистрах
- Новые форматы команд, например, трехадресное сложение или сложение с константой

# Почему нельзя было просто расширить регистры CdM-8?

- Возможно, это упростило бы и разработку процессора, и переучивание?
- CdM-8 может обойтись парой команд ld/st
- У 16-битного процессора команд load/store должно быть больше
- Кроме 16-битных значений, мы должны уметь работать с 8-битными (ASCII, UTF-8, 8-битные регистры ввода-вывода)
  - Ldw (загрузить 16-битное слово)
  - Ldb (загрузить байт с расширением нулями)
  - Ldsb (загрузить байт с расширением знаком)
  - Stw
  - Stb
- У CdM-8 просто нет свободных кодов команд для такого!



# Где взять коды команд?

- Уменьшить количество регистров в процессоре
  - Меньше 4 регистров??? Вы издеваетесь?
  - На самом деле, бывали процессоры с 2 регистрами (MC 6800), с одним арифметическим регистром (MOS 6502) и вообще без нумеруемых регистров (Lilith/Кронос, Transputer, Java bytecode)
- Уменьшить количество регистров в команде
  - Одноадресные команды.
    - Например, назвать r0 аккумулятором,
    - и считать что у всех команд (неявный) второй операнд – это r0.  
add r1 -> r0 += r1
    - На самом деле, популярная идея была в прошлом:
      - CDC 1604, БЭСМ-6, Intel 4004...8085, Microchip PIC, MOS 6502, MCS-48
- С load/store архитектурой это все плохо сочетается

# Расширить команду

- До 12 бит?
  - Видал я и такое не раз
  - Microchip PIC: ПЗУ для программ имеет 12- или 14-битную шину
  - Команды, не выровненные на единицу адресации (байт или слово)
  - И такое я видал: CDC 1604, CDC 6600, БЭСМ-6, Intel 432
  - Все равно извращение
- 16-битная команда – раззудись, плечо!
- Возможные форматы для load/store архитектуры:

Opcode 8 бит	R1 4 бит	R2 4 бит
--------------	----------	----------

16 регистров, 2 операнда (MIL-STD 1750A)

Opcode 7 бит	R1	R2	R3
--------------	----	----	----

8 регистров, 3 операнда (Thumb)

# Что мы выбрали для CdM-16

- После нескольких раундов обсуждений мы остановились на архитектуре с 8 регистрами и трехадресными командами
- Причины:
  - Для ручного программирования 16 регистров многовато
  - Трехадресные команды Load/Store позволяют познакомиться с продвинутыми режимами адресации
  - Получается больше похоже на Thumb и «взрослые» RISC-процессоры, такие, как ARM

# Источники вдохновения

- На самом деле, 16-битных процессоров с load/store архитектурой в истории не так много
- Большинство популярных 16-битных процессоров или процессоров с 16-битными командами: PDP-11, Intel 8086, x86 - это так называемые CISC (Complex Instruction Set Computer)
- CISC не означает, что у процессора много команд
- Это означает, что у большинства команд есть «режимы адресации». Например, команда ADD может сложить регистр с операндом в памяти или даже два операнда в памяти.
- Это означает сложную схему кодирования команд, и сложный микрокод для их интерпретации
- Мода на такие компьютеры началась в 1960е (IBM System/360) и в некотором роде продолжается до сих пор: x86/x64 и IBM System/Z до сих пор где-то ездят
- Это все совсем другая сказка
- Надеюсь, я ее вам в этом семестре расскажу
- Но не сейчас

# Load/store (RISC) процессоры

- Большинство load/store ISA были разработаны в 80е и 90е годы
- ARM, MIPS, RISC V, SPARC, PA-RISC, DEC Alpha, Intel Itanium, MMIX да тыщи их
- Все это 32- или 64-разрядные процессоры с 32-битными командами
  - Logisim такое не потянет
  - В рамках проекта вы вряд ли напишете программу больше 64 килобайт
  - А значит вам больше 16-разрядного процессора и не нужно

# Популярные load/store архитектуры с 16-битными командами

- Atmel AVR: 32 8-битных арифметических регистра, 3 16-битных индексных (на самом деле, регистровые пары) двухадресные команды
- MIL-STD-1750A: 16 16-битных регистров общего назначения, двухадресные команды
- ARM/Thumb: 8 32-битных регистров общего назначения  
трехадресные команды add/sub, ld/st  
есть команды с «короткими константами»  
большинство команд двухадресные

# Что такое «короткие константы»?

- 16-битные данные и адреса означают, что такие команды, как `ldi`, `jsr` и `br*` должны иметь 16-битное адресное поле
- То есть полная длина таких команд должна быть 32 бита (4 байта)
- Это не всегда нужно
- Большая часть целочисленных констант в реальных программах — это небольшие числа
- Чаще всего встречается 0
- В программах для `load/store` ISA, большая часть констант — это не числа, а адреса

# Базовая и относительная адресация

- В программах для load/store ISA, большая часть констант – это не числа, а адреса
- В 32- и 64-битных программах, адреса могут составлять большую часть объема программы. 16-битные программы в этом смысле переходный этап.
- Можно ли сэкономить при размещении адресных констант?
- Обращения к памяти обладают локальностью: длинные серии обращений делаются к небольшим регионам памяти
  - Локальные переменные и параметры вашей функции (стековый кадр)
  - Статические переменные вашего модуля
  - Поля структур, переданных как параметры
  - Циклы и условные операторы внутри подпрограммы
  - Вызовы подпрограмм внутри модуля



# Базовая адресация

- Локальные переменные и параметры функции (стековый кадр)
- Статические переменные модуля
- Поля структур, переданных как параметры
- Занимаем регистр, который указывает куда-то поблизости от соответствующих регионов памяти  
(с 8 регистрами мы можем себе это позволить)
- Обращаемся к полям или переменным по смещениям от этого регистра  
Смещения будут короткие, скорее всего меньше 8 бит
- В CdM-8 есть рудиментарная поддержка такого: команда `ldsa`

# Поддержка базовой адресации в CdM-16

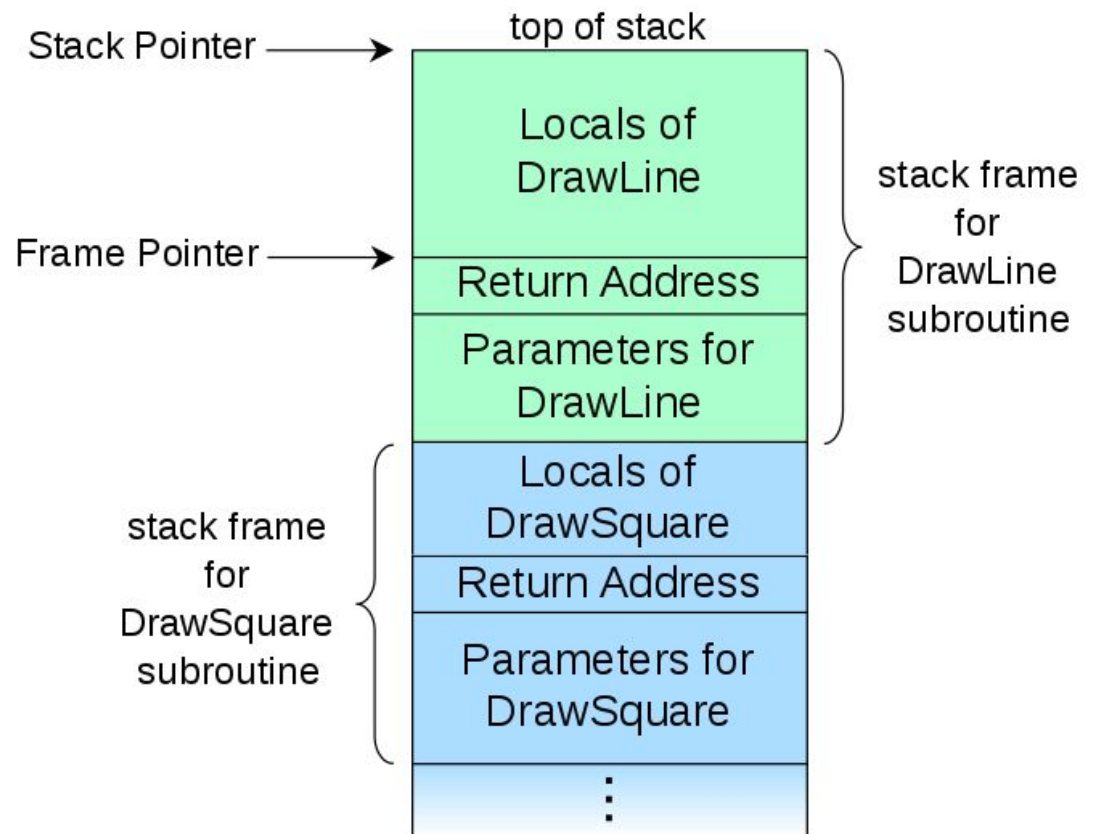
- Команды  $ls^*/ss^* rd, imm6$  (регистр и 6-битная константа)



- $Lsw\ rd, imm6 \rightarrow rd := mem[r7 + imm6 * 2]$
- Действительность чуть богаче: есть две команды  $lsw$ , с положительным и отрицательным  $imm6$ , что позволяет адресовать 128 байт вперед и назад от  $r7$
- Регистр  $r7$  также известен как  $fp$  (frame pointer)
- $Lsb/lssb\ rd, imm6 \rightarrow rd := mem[r7 + imm6]$
- То есть байтовых переменных вы можете адресовать всего 64  
Учитывайте это когда размещаете локальные переменные!

# Как предполагается использовать fp/r7?

- Создаете в стеке место для локальных переменных: *стековый кадр* (stack frame) или *запись активации*
- Новое для каждого вызова (поддерживает рекурсию)
- Пушим параметры в стек
- Зовем jsr DrawLine
- Первые команды Drawline:  
push r7  
ldsp r7  
addsp -frame\_size
- Используйте ls\*/ss\* для доступа к параметрам и переменным



# Почему fp, а не sp?

- Вы можете завести массив в качестве локальной переменной, и он (вместе с остальными переменными) не влезет в 128 байт. В результате, вы сможете добраться 6-битным смещением только до некоторых переменных и не сможете добраться до параметров
- Вы можете захотеть завести динамический массив или делать push/pop внутри функции. Это приведет к передвижению вершины стека, а значит все смещения от вершины стека надо пересчитывать.
- Компилятору это легко сделать, он железный, поэтому в ARM/Thumb аналоги команд ls/ss используют смещения от SP (код для ARM/Thumb чаще пишут компиляторы, чем люди)
- Для ручного программирования удобнее FP.
- В CdM-8 для выделенного FP не хватало ни регистров, ни кодов команд

# Но можно и по-другому!

- Если вам не нужны локальные переменные, но нужна серия операций над статическими переменными
  - Сохраняете r7 (соглашение о вызовах требует, чтобы он сохранялся при вызовах)
  - Наводите r7 на начало или в середину блока статических переменных
  - Используете ls\*/ss\* для работы с переменными
  - Для получения смещения переменной можно использовать символьную арифметику (вычитание меток)
  - Восстанавливаете r7
- Если вам нужна серия операций над полями структуры
  - Сохраняете r7
  - Кладете указатель на структуру в r7
  - ...

# Что, если r7 нужен как frame pointer?

- Есть трехадресные команды  $ld^*/st^* \text{ rb, ri, rd/rs}$
- $Ldw \text{ rb, ri, rd} \rightarrow Rd := mem[rb+ri]$
- Rb – базовый регистр, Ri – индексный (любые регистры общего назначения)
- Можно использовать для обращения к элементам массива (не забыть умножить индекс на размер элемента!)
- Можно использовать для обращения к полям структуры  
значения в ri можно загружать командой  $ldi \text{ rd, imm6}$ ,  
она занимает 2 байта в отличие от  $ldi$  с 16-битной константой
- Можно использовать для обращения к статическим переменным

# Относительная адресация (базовая адресация с РС в качестве базы)

- Циклы и условные операторы
- Вызовы подпрограмм внутри модуля
- Константы в сегменте кода (для команды `lc`)
- CdM-16 имеет «короткую» форму команд `jsr` и `br*`, у которых адрес формируется сложением значений РС и 9-битного адресного поля команды. Эти команды занимают 2 байта, в то время, как соответствующие команды с полным адресом занимают 4 байта
- Ассемблер сам выбирает короткую форму команд, если знает адрес целевой метки и она попадает в диапазон 512 байт от места обращения
- Для внешних меток, ассемблер не знает адрес и не может использовать короткую форму, даже если после линковки она окажется допустимой

# Относительная адресация для констант

- Для ldc формы со смещением относительно PC нету,  
но можно написать

```
ldpc rb          # rb := PC, 2 байта
add rb, _-const_location # 2 байта
# Если вы уверены, что const_location не далее 64 байт от _
lcw rb, rd       # 2 байта
```
- Вроде бы, по длине это не лучше, чем

```
ldi rb, const_location # 4 байта
lcw rb, rd             # 2 байта
```
- Но упрощает работу линкеру  
(т.наз. *позиционно-независимый код*)
- А если вам нужно загрузить несколько констант подряд,  
можно и по длине кода сэкономить



# Модель памяти

- В CdM-8 модель памяти была простая.
  - Ячейка памяти 1 байт,
  - адресуемая единица памяти 1 байт,
  - все операции с памятью и регистрами работают с 1 байтом
- Мы заявили, что CdM-16 поддерживает операции с 16-битными (2 байта) и 8-битными (1 байт) значениями
- У 32- и 64-битных процессоров типов данных еще больше
- Как должна выглядеть память такого процессора для программиста?

# Варианты модели памяти

- 16-битная шина памяти, единица адресации - 2 байта (слово)
- У всех (ну хорошо, у всех которые я знаю) ISA до 1964 года, память была устроена именно так
  - только шина памяти могла быть шире, а слова длиннее. Например,
  - CDC 1604 и БЭСМ-6 слова были 48 бит,
  - PDP-6 и DEC 10/20 – 36 бит
  - Кронос – 32 бита

# Как работать с байтами, если адресация по словам?

- Складывать по одному байту в слово
  - Удобно адресоваться
  - Но вы что, издеваетесь???
- Заставить программиста (компилятор) с этим разбираться
  - Программист может упаковать несколько байтов в слово, но потом вынужден сам доставать нужные байты сдвигами или чем-то вроде
  - CDC-1604, БЭСМ-6
  - Можно тоже спросить про «вы издеваетесь», но люди с этим жили
- Сделать отдельные команды для манипуляции с байтами и отдельное представление для указателя на байт
  - DEC 10/20 (36 бит на 8 нацело не делится)  
была поддержка 6-битных и 9-битных «байтов»
  - Lilith/Кронос
  - Писать для такой машины компилятор С – удовольствие ниже среднего, но я участвовал в таком проекте

# То есть, все-таки адресация по байтам?


- Первая (во всяком случае первая широко известная) ISA такого типа – IBM System/360 (анонсирована в 1964 году)
  - Ширина шины памяти и регистров – 32 бита, адресация по байтам
- Еще очень известная ISA – PDP-11 (первая машина линейки – 1970)
  - 16 бит, адресация тоже по байтам
- Я думаю, все компьютеры, с которыми вам придется иметь дело, используют такую же структуру памяти, только шина может быть 64 или даже 128 бит.

# Порядок байтов (endianness)



- Если вы программно реализуете 16- и 32-битную арифметику на основе байтовой, вы можете раскладывать байты как хотите (главное, чтобы весь ваш код понимал раскладку одинаково)
- Если машина имеет аппаратную 16- и более битную арифметику, она диктует вам раскладку байтов
- Есть два логичных варианта:
  - Little endian (младший байт по меньшему адресу): PDP-11, x86
  - Big endian (младший байт по большему адресу): IBM 360, MC 68000
  - У ARM это переключается программно, но, например, Android и MacOS/M1 работают только в little endian
- CdM-16 little endian

# Выравнивание (alignment)

Как программист видит память

Data	               
Address	0 1 2 3 4 5 6 7 8 9 A B C D E F

Как процессор видит память

Data	   
Address	0 1 2 3 4 5 6 7 8 9 A B C D E F



# Почему выравнивание важно

- Операции с невыровненными словами (с нечетным адресом) требуют два цикла памяти,
- То есть вдвое медленнее, чем с выровненными
- Поддержка требует дополнительной схемотехники
  - можете заглянуть внутрь `cdm16.circ` в микросхему Bus Control, и ужаснуться
- Многие процессоры это вообще отказываются делать
- На 32-разрядной машине, 32-битные слова должны быть выровнены на 4 байта (указатель должен делиться на 4)

# Выравнивание в CdM-16

- CdM-16 может работать с невыровненными 16-битными данными, но это медленно и этого следует избегать
- CdM-16 не может работать с невыровненными командами
- Все команды имеют длину 2 или 4 байта,
- Нету команд длиной 1 или 3 байта
- Передача управления по нечетному адресу генерирует аппаратное исключение
- Стек также обязан быть выровненным (SP всегда четный)
- Нет команд push и pop для байтов
- В ассемблере есть директива align, которая выравнивает код или данные, если это необходимо



# Еще плюшка, специфичная для Logisim

- Микросхема памяти Logisim может иметь ширину шины 16 бит, но такая память не поддерживает операции с отдельными байтами
- Можно сделать два отдельных банка памяти объемом 32кб для четных и нечетных адресов
- Но в такую память неудобно загружать образы, созданные cocol (надо написать дополнительную программу, которая поделит образ на четные и нечетные байты)
- В рамках проекта CdM-16 разработана библиотека Java, которая реализует 16-битную микросхему памяти с побайтовым доступом и удобной загрузкой образов

# Более формальное описание CdM-16

- 11 16-битных регистров, доступных программисту
  - 8 регистров общего назначения r0..r7, включая специальный регистр FP (r7)
  - SP, PC, PS (PSW), аналогичные одноименным регистрам CdM-8

r0
r1
r2
r3
r4
r5
r6
R7/FP

SP
PC
PS

# Команды обработки данных

- Add[c]/sub[c] rs0, rs1, rd
- Add/sub/cmp rd, imm6 (сложение с короткой константой)
- Cmp rs, rd (сравнение, неразрушающее вычитание)
- Neg/sxc/scl rs, rd
- And/or/xor/bic rs0, rs1, rd
  - Вопрос на сообразительность:  
почему нет побитовых операций с короткой константой?
- Если нужна двухадресная форма, вы можете использовать один и тот же регистр в качестве rs[01] и rd
- Sh[lr]/ro[rl]/rc[rl]/shra rs, rd, imm3+1
  - вы не можете сдвинуть на 16 бит одной командой, только на 8
  - Нельзя сдвинуть одной командой на переменное число бит
- Not/cmp/bit/move rs, rd

# Что нужно иметь в виду

- **Все** арифметические команды, сдвиги и сравнения 16-битные
- При загрузке байтов из памяти, нужно внимательно следить, знаковый или беззнаковый байт вам нужен
- Если нужна 8-битная арифметика (особенно сравнения), нужно приводить промежуточные значения к 8 битам командами
  - Sxc (расширить младший байт регистра со знаком)
  - Scl (расширить без знака, т.е. очистить старший байт)
- Если вам нужен 8-битный циклический сдвиг, вас ждут приключения

# Команды для работы с памятью

- Все команды load имеют три формы: w (16 бит), b (байт, расширенный нулями) и sb (байт, расширенный знаком)
- Все команды store имеют две формы: w и b
- Основные команды:
  - ld\*/st\* rs0, rs1, rd (адрес равен rs0+rs1)
  - ld\*/st\* rs, rd (адрес равен rs)
  - Lc\* rs0, rs1, rd (адрес в банке кода, в CdM-8 это называлось ldc)
  - Ls\*/ss\* rd, imm6 (адрес равен r7+-imm6\*size)
  - Ldi rd, imm16; ldi rd, +-imm6
- Стек: push rs; push +-imm6; pop rd; addsp imm9 – только 16-битные значения!
- push/pop для SP, PC, PS
  - Pop PC (popc) == rts

# Команды передачи управления

- Коды условий точно такие же, как у CdM-8
- B\* imm16/imm9 (ассемблер сам выбирает более короткую возможную форму)
- Jsr imm16/imm9
- Jsrr rd (вычисляемый jsr)
- Rts (porc) – возврат из подпрограммы
- Вычисляемый переход по адресу в регистре  
stpc rs -> pc := rs

# Исключения

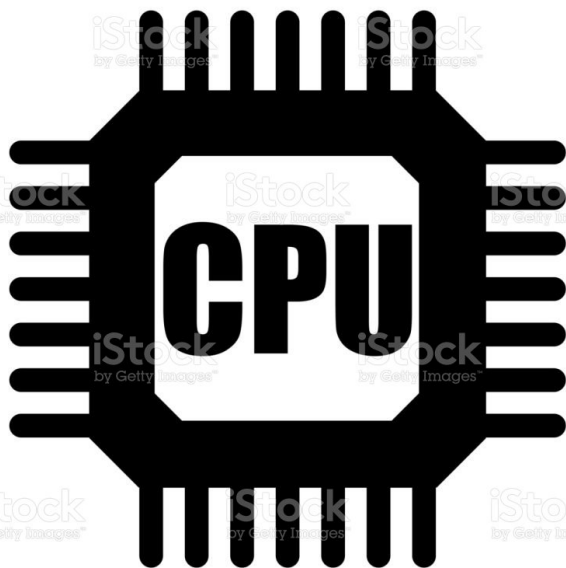
- Исключения похожи на прерывания, но возникают внутри процессора, а их обработка начинается до (точнее, происходит вместо) завершения команды
- Сохраненный PC указывает на команду, которая стриггерила исключение
- Под них зарезервированы первые 16 векторов
- В данный момент реализованы исключения:
  - Аппаратный сброс (включение)
  - Зарезервированная/недопустимая команда (да, у нас много пока неиспользуемых кодов команд)
  - Попытка передачи управления по нечетному адресу
  - Попытка записи нечетного значения в SP
  - Double fault (исключение вылетело при попытке обработки исключения, например мусор в векторе)
- При включении процессора, вместо передачи управления по адресу 0, происходит аппаратное исключение 0

# Еще про исключения

- Для нормальной работы процессора необходимо проинициализировать вектора исключений
- В CdM-16 эти вектора лежат начиная с нулевого байта программной памяти
- Можно использовать директиву `asect 0x0`
- Вектор `double fault` проще навести на команду `halt`
- Если вы не хотите обрабатывать остальные исключения, то же самое можно сделать и с ними
- Нулевой вектор указывает на начало вашей программы (то, куда надо передать управление при сбросе или включении)



# Пример кода



Cpu

```
asect 0
main: ext      # Declare labels
default_handler: ext # as external
```

```
# Interrupt vector table (IVT)
# Place a vector to program start and
# map all internal exceptions to default_handler
dc main, 0 # Startup/Reset vector
dc default_handler, 0 # Unaligned SP
dc default_handler, 0 # Unaligned PC
dc default_handler, 0 # Invalid instruction
dc default_handler, 0 # Division by zero
align 0x80 # Reserve space for the rest of IVT
```

```
# Exception handlers section
rsect exc_handlers # This handler halts processor
default_handler> halt
```

```
# Main program section
rsect main
main> # your code here
end.
```

# Пока не реализованные мечты

- Команды умножения и деления
- Групповые push/pop (в младшем байте команды лежит битовая маска регистров, которые надо сохранить или восстановить)
- Почему не реализованы:
  - нужно усложнить секвенсор и тракт данных
  - Для умножений и делений он должен понимать, что операция АЛУ может занимать много тактов, а результат надо раскладывать по нескольким регистрам
  - Для групповых push/pop нужно уметь делать циклы в микрокоде

# Направления дальнейшего развития

- Системный и пользовательский режимы и диспетчер памяти
- Операционная система
- Сопроцессор с плавающей точкой
- Компилятор C, например на основе LLVM