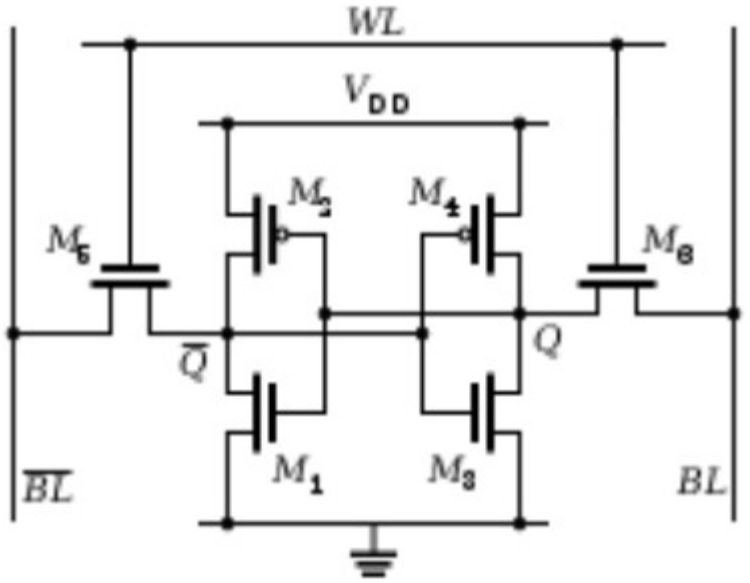
Процессор. Часть 2

Лекция №3
АКОС 2019-2020

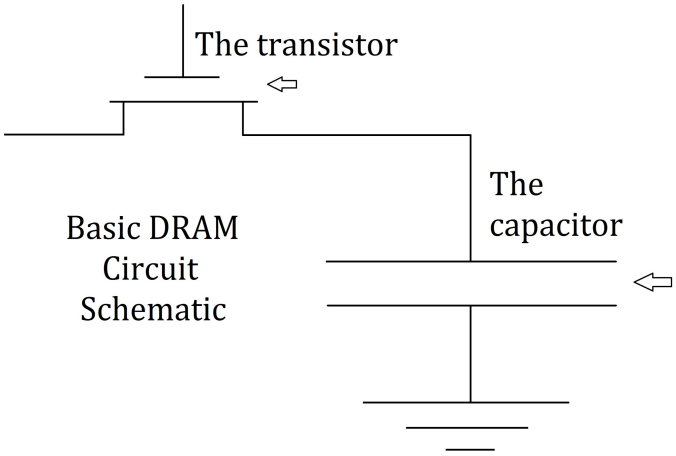
SRAM

- Время чтения - 1 такт
- Время записи - 2 такта
- Тактовая частота - в зависимости от размеров транзитора



DRAM

- 1 транзистор + 1 конденсатор
- Конденсатор требует перезарядки после каждого чтения
- Чтение/запись затратны по времени
- Периодическая регенерация (каждые 64 нс) из-за утечек



DRAM v.s. SRAM

DRAM

- 2 элемента на ячейку
- Медленный доступ
- Требуется схема для перезарядки конденсаторов

SRAM

- 6 элементов на ячейку
- Время доступа ограничивается тактовой частотой
- Простая шина

Локальность доступа

- Локальность по времени
  - фрейм текущей функции
  - статические и thread-local переменные, используемые в настоящий момент
- Локальность по расположению данных
  - код программы или библиотеки
  - структуры и массивы (в т.ч. - многомерные)

Информация в Linux

- `/sys/devices/system/cpu/cpu0/cache`
  - `index0`
  - `index1`
  - `index2`
  - `ibdex3`
- cpuX - отдельное ядро (они одинаковые, так что всё равно)
- indexY - отдельные кэши кэша

Уровни кэша (в x86)

- L1 (index0 + index1 для Intel) - самый близкий кэш микроинструкций и текущих данных, с которыми оперируют микроинструкции
- L2 - общий кэш, связанный с ограниченным количеством ядер (как правило - одним, но не всегда)
- L3 - общий кэш, связанный со всеми ядрами

Характеристики кэшей

- Уровень в иерархии кэшей [`level`]: число от 0 до 2
- Размер кэша [`size`]: от 64Кбайт (L1) до 35Мб (L3 в топовых Intel Xeon, их тех, что сейчас можно купить в Москве, для Core-серии характерный размер 8Мб)
- Тип [`type`]: Data,Instruction или Unified
- Какими ядрами используется [`shared_cpu_list`]: номера ядер

Ключевая проблема

- Запихать невпихуемое

(вся память в несколько Gb не может быть ужата до нескольких Mb)

Причины кэш-промахов

- Первое обращение к определенной области памяти
- Данные были выгружены из-за ограниченного размера кэша
- Данные были выгружены из-за ограниченной ассоциативности

Как устроен кэш

- **Блок** - минимально адресуемый объем данных в кэше. 64 байта для Intel
- **Кэш-линия** - блок + метаданные, определяющие адрес в памяти
- **Набор** - связан с некоторым адресом в оперативной памяти. Для L3 - 12 линий по 64 байт = 768 байт
- Весь кэш состоит из независимых наборов. Для Core i3 размер кэша L3 3Мб = 64 байта в блоке * 12 линий в наборе * 4096 наборов

Характеристики кэшей

- Размер данных в кэш-линии [`coherency_line_size`]: 64 для Intel
- Количество ассоциативных каналов в кэше [`ways_of_associativity`]: 8 для уровней L1-L2, 12-16 для уровня L3
- Количество наборов в кэше [`number_of_sets`]: 512 для Core i3, 8192 для Xeon E3-1230

Что значит Cache-Friendly

- По возможности использовать непрерывные блоки данных
- Выравнивать данные по границе кэш-линии: `_Alignas(64)` в Си 2011

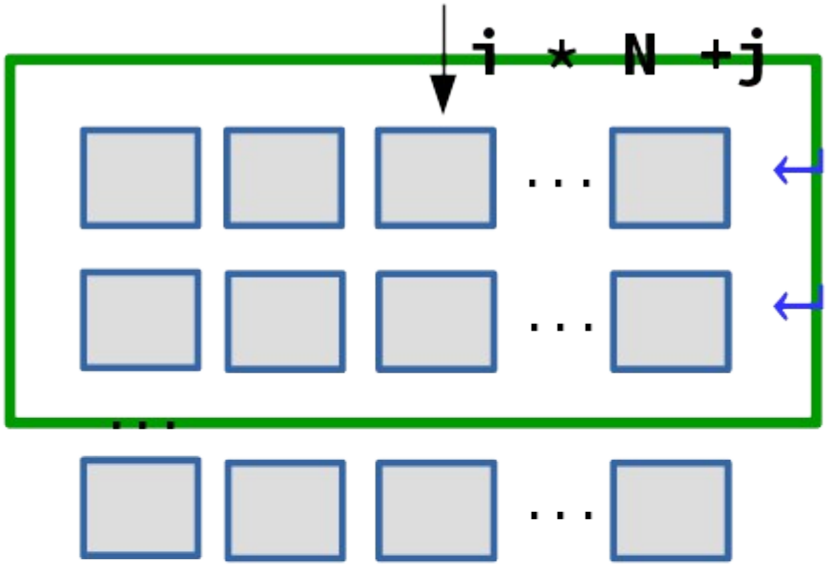
Hint:

64 байта = 512 бит = 4 регистра SSE
= 2 регистра AVX
= 1 регистр AVX-512

Как использовать кеш?

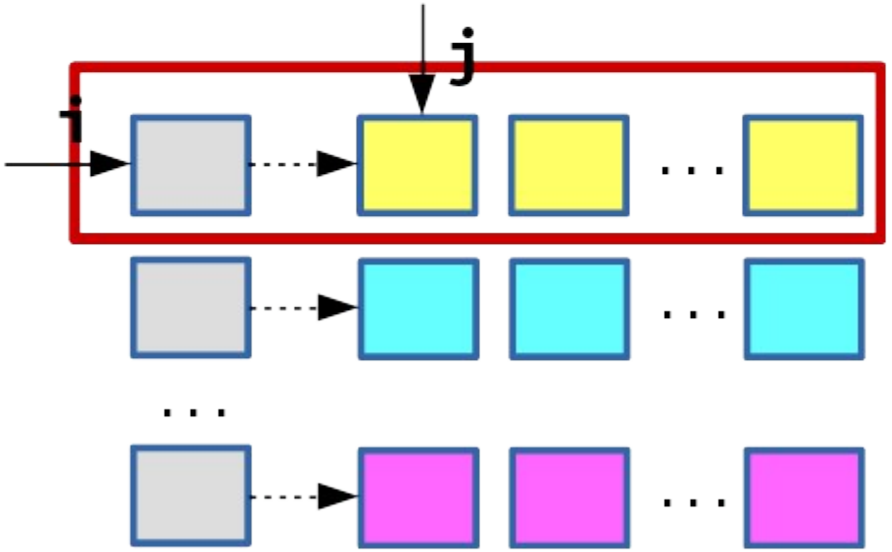
- В наборах инструкций нет команд управления кешем
- Компиляторы Си/C++ и пр. могут генерировать код для предзагрузки данных
- Увеличить вероятность попадания в кеш можно размещая данные последовательно

Многомерные массивы



В кэш-память попадает максимально возможный непрерывный блок

В кэш-память попадает только одна строка матрицы, поскольку нарушено условие непрерывности расположения данных



Дополнительные ограничения

```
void some_func(const restrict * data) /* C99 */
{
    // компилятор имеет право сгенерирвать код для загрузки
    // содержимого по указателю 'data' в кэш
    . . .
}

/* C++, CLang/GCC */
#define restrict __restrict__

/* C++, MSVC */
#define restrict __restrict
```

Наглядный пример: **matrix dot**

```
using std::vector;
typedef vector<vector<int>> matrix_t;

extern matrix_t dot(const matrix_t &A, const matrix_t &B)
{
    size_t rows = A.size();
    matrix_t R(rows, vector<int>(rows, 0));
    for (size_t i=0; i<rows; ++i) {
        for (size_t j=0; j<rows; ++j) {
            int sum = 0;
            for (size_t k=0; k<rows; ++k) {
                sum += A[i][k] * B[j][i];
            }
            R[i][j] = sum;
        }
    }
    return R;
}
```

1000x1000 - 16.425 секунд
Core i3 / 3Mb L3 / 1.9GHz

Наглядный пример: **matrix dot**

```
using std::vector;
typedef vector<int> matrix_t; // одномерный массив
                                        // BT - транспонир.
matrix_t dot(size_t N, const matrix_t &A, const matrix_t &BT)
{
    matrix_t R(N*N);
    for (size_t i=0; i<N; ++i) {
        for (size_t j=0; j<N; ++j) {
            int sum = 0;
            for (size_t k=0; k<N; ++k) {
                sum += A[i*N+k] * BT[i*N+k];
            }
            R[i*N+j] = sum;
        }
    }
    return R;
}
```

1000x1000 - 10 секунд
Core i3 / 3Mb L3 / 1.9GHz

Профилирование

```
valgrind --tool=cachegrind \
    ПРОГ [АРГ0][ ... АРГn]
```

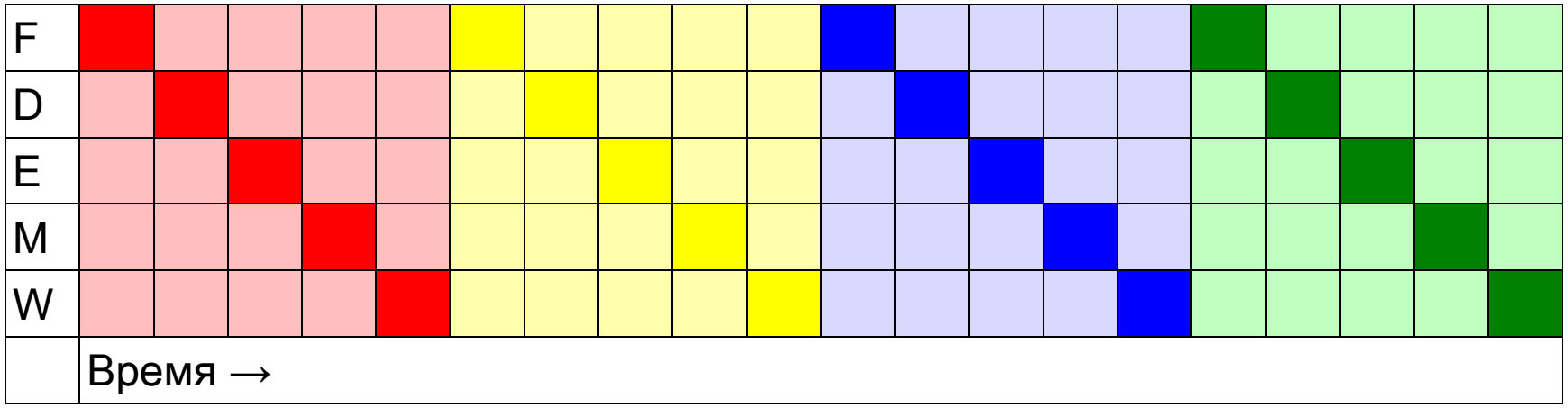
- Требуется отладочная информация (-g)
- Терпение.... valgrind работает медленно
- Чем дольше ждём - тем точнее результат
- На выходе получаем XML-файл `cachegring.out.`*`PID`*
- Смотрим файлик в KCacheGrind/QCacheGrind [`https://sourceforge.net/projects/qcachegrindwin/`]

Pipelining & Out-of-order execution

ЕЩЁ ПРО ОПТИМИЗАЦИИ

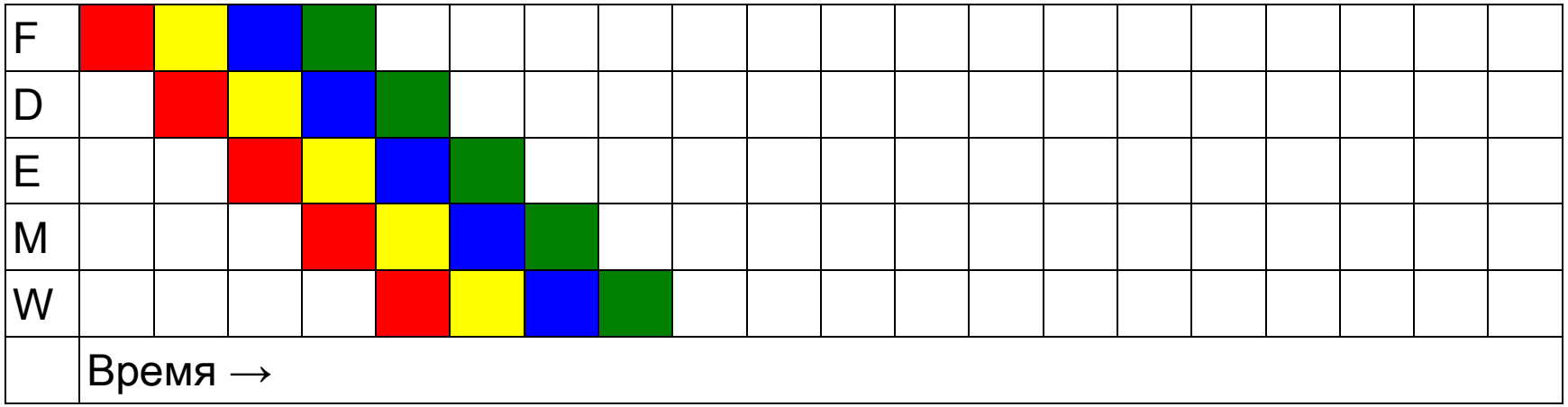
Стадии выполнения команд

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Register Write Back



Стадии выполнения команд

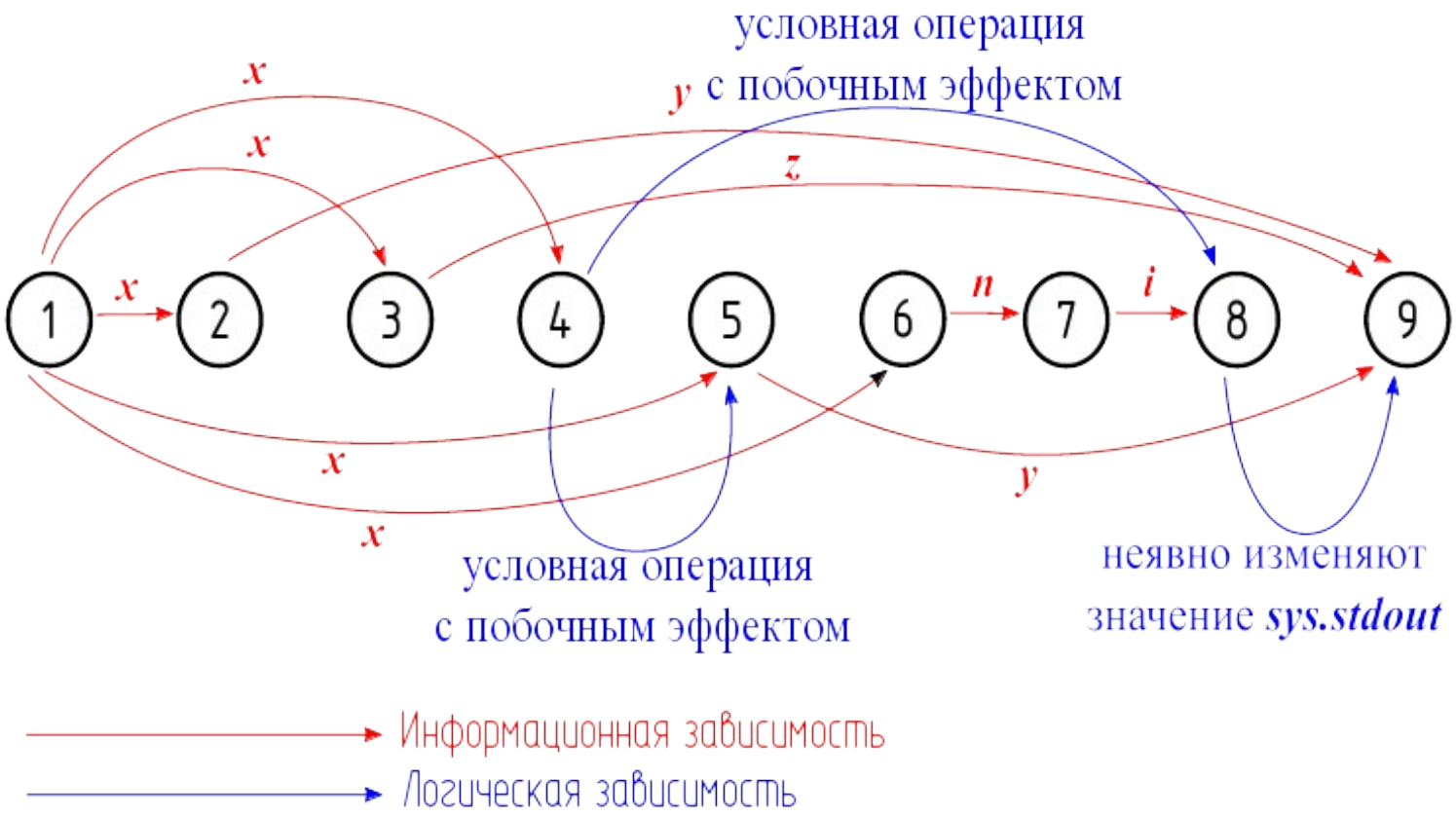
- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Register Write Back



Сверхдлинные конвейеры

- Общая идея: упрощаем инструкции, выполняемые на каждом шаге
- Но набор инструкций зафиксирован ISA
  - Для CISC-архитектуры используется микрокод
  - Для RISC это тоже возможно
- Длины конвейера:
  - Современные Intel/AMD/ARM: **8..15**
    Только в самых современных процессорах Intel топовая частота 4ГГц, наиболее распростаненны от 1 до 2,5ГГц
  - PowerPC: **20**
    Порог 4ГГц (POWER6) пройден в 2010 году, на 22nm - 5ГГц (POWER7)
  - Pentium 4: **31**
    10 лет назад характерная частота ~3ГГц

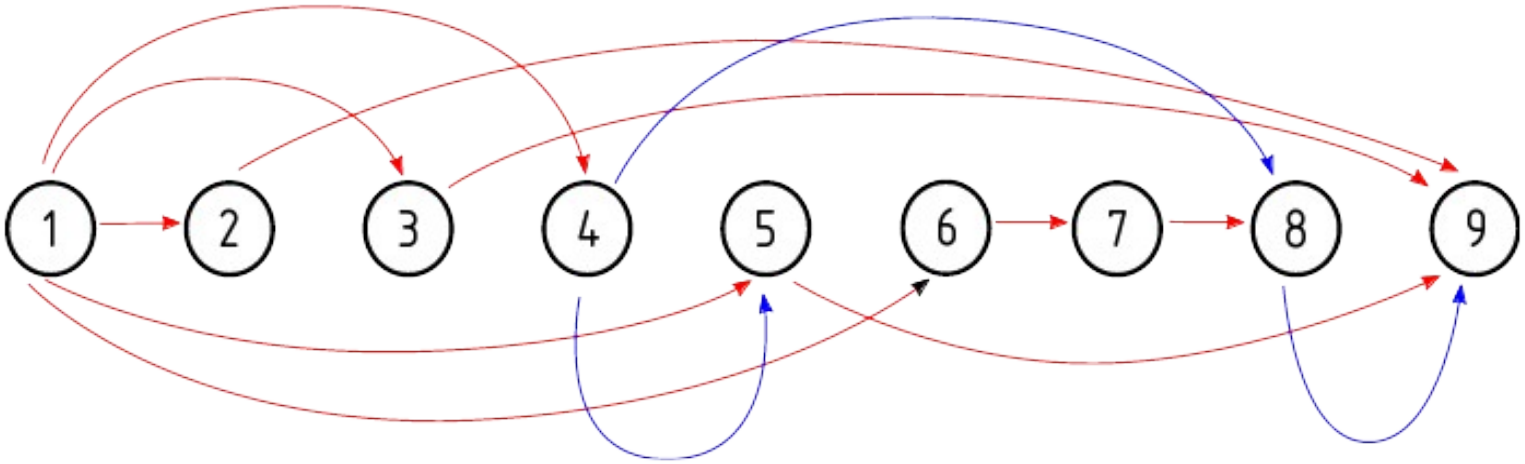
```
1:      input(x)
2:      y = x
3:      z = x + 1
4:      if x < 0:
5:          y = -x
_:        else:
6:        n = x * 2
7:          for i in range(0, n):
8:            print(i)
9:    print(y, z)
```



```
1:      input(x)
2:      y = x
3:      z = x + 1
4:      if x < 0:
5:          y = -x
_:        else:
6:       n = x * 2
7:          for i in range(0, n):
8:           print(i)
9:   print(y, z)
```

1-й ярус	1
2-й ярус	2 3 4 6
3-й ярус	5 7
4-й ярус	8
5-й ярус	9

1. Первый ярус содержит все узлы, которые не зависят от других узлов
2. ***k*+1** ярус содержит все узлы, которые зависят от узлов графа, принадлежащих ***k***-му, ***k-1***-му, ***k-2***-му … ***1***-му ярусу.



Идея спекулятивного выполнения

- Распараллеливаем поток инструкций на несколько функциональных устройств
- В случае условного выполнения - все равно выполняем; если не угадали ветку - результат просто отбрасываем
- Это эффективно для микроинструкций
- Компилятор может переставлять процессорные инструкции местами

В переводе с английского:
взрыв ядерного реактора, фиаско, облом [www.multitran.ru]

MELTDOWN

Meltdown

- Критическая уязвимость, опубликованная 26 января 2018 г.
- Windows, Linux, Mac, - не имеет значения; проблема в железе, а не софте
- Скомпрометированы все процессоры Intel, старшие процессоры PowerPC, и последние ARM-Cortex

Уязвимость использует два фактора:

1. Спекулятивное выполнение инструкций
2. Нахождение в кеше данных, которые могут быть "чужими"

https://github.com/IAIK/meltdown

Спекулятивное выполнение

Псевдо-ассемблер x86:

```
1: mov     address → %rax
2: div     %rbx / $0
3: mov     [%rax] → %rbx
```

- Инструкция div выполняется долго и завершается ошибкой
- Несмотря на это спекулятивно выполняются все последующие инструкции
- Это приводит к загрузке в кеш данных без проверки доступа
- Результат div отбрасывается, но данные остаются в кеше

Побочный канал связи

- Побочный канал связи - способ косвенно выяснить недоступные данные
- Для проверки нахождения данных в кеше - можно замерить время доступа

Использование Timing-attack

Псевдо-ассемблер x86:

```
1: mov     address → %rax
2: div     %rbx / $0
3: mov     [%rax] → %rbx
4: %rbx =  (%rbx & 0xF) << 6
5: mov     [arr+%rbx] → %rbx

...
N-1: rdtsc
N  : mov   [arr+%rbx] → %rbx
N+1: rdtsc
...
```

- Побочный канал связи - способ косвенно выяснить недоступные данные
- Для проверки нахождения данных в кеше - можно замерить время доступа
- Работает медленно, но верно.

Как бороться?

- Минимизация доступа к памяти ядра или гипервизора
- Рандомизация размещения данных ядра в памяти
- Снижение точности perf-таймеров
- Запрет на кеширование при спекулятивном выполнении

Защита от Meltdown - снижение производительности до 30%

Кодирование команд

RISC v.s. CISC

- CISC: переменное количество байт на команду
- RISC: размер команды - машинное слово
- Для гарвардиких архитектур может быть два размера машинных слов (AVR)

Кодирование инструкций AVR

ADD – Add without Carry

Description

Adds two registers without the C Flag and places the result in the destination register Rd.

	Operation:		
(i)	(i) Rd ← Rd + Rr		
	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

Кодирование инструкций AVR

LDI – Load Immediate

Description

Loads an 8-bit constant directly to register 16 to 31.

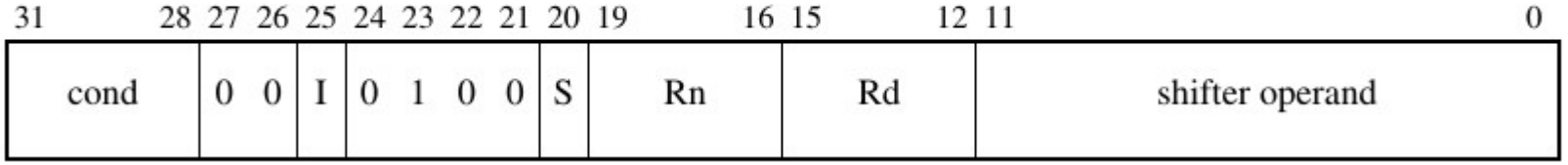
	Operation:		
(i)	Rd ← K		
	Syntax:	**Operands:**	**Program Counter:**
(i)	LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK

Кодирование ARM-32

ADD



- **cond** - условие выполнения команды
- **I** - флаг, определяющий, что закодировано в **shifter_operand**
- **S** - флаг, определяющий, нужно ли обновлять регистр статуса
- **Rn**, **Rd** - первый аргумент операции и куда записать результат

Примеры:

ADD **r0**, **r1**, **r2** // r0 ← r1 + r2
ADD**S** **r0**, **r1**, **r2** // r0 ← r1 + r2, обновить флаги Z,V,C,N
ADD**EQ** **r0**, **r1**, **r2** // Если Z, то r0 ← r1 + r2
ADD **r0**, **r1**, **r2, lsl #3** // r0 ← r1 + (r2 << 3) [5bit shift; 2bit type; 0; 4bit reg]
ADD **r0**, **r1**, **#0xFF** // r0 ← r1 + 0b0000'1111'1111
ADD **r0**, **r1**, **#0x200** // r0 ← r1 + (0b0000'0010 ROR 0b1100)

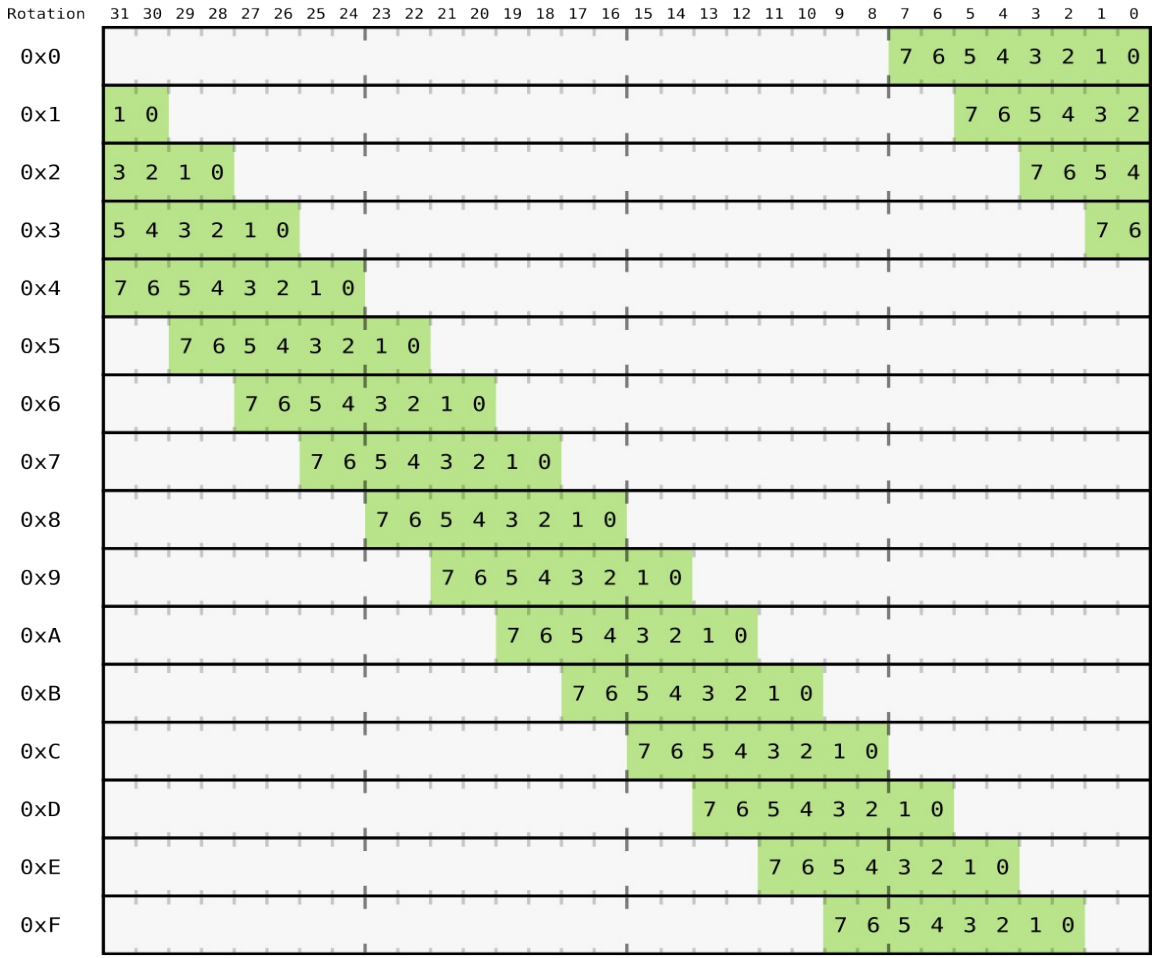
Циклический сдвиг

ADD **r0**, **r1**, **#0x200**

```
0x200 = 512 =
0b10'0000'0000

Lo(Instr) =
0b1100'0000'0010
  rot  immediate

0b1100 = 0xC
0b0000'0010
```

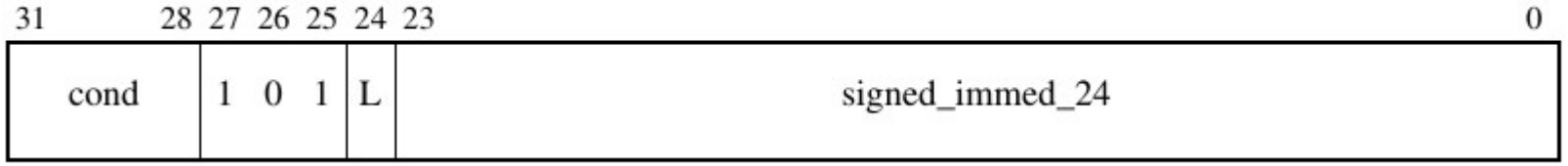


Условное выполнение команд

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

Кодирование ARM-32

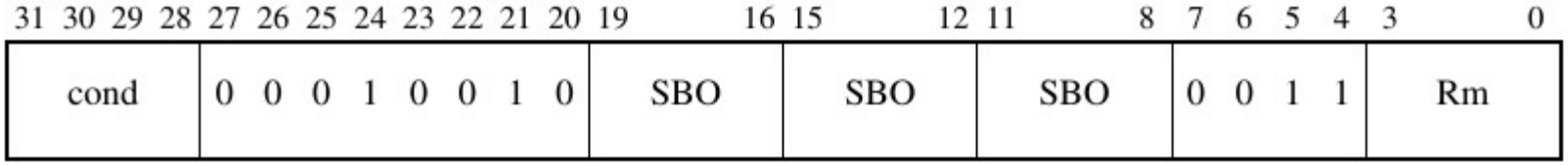
B, BL



+/- 2^{23} - это +/- 8Мб

Branch with Link and eXchange

BLX (2)



BLX (2) calls an ARM or Thumb subroutine from the ARM instruction set, at an address specified in a register.

It sets the CPSR T bit to bit[0] of Rm. This selects the instruction set to be used in the subroutine.

The branch target address is the value of register Rm, with its bit[0] forced to zero.

It sets R14 to a return address. To return from the subroutine, use a `BX R14` instruction, or store R14 on the stack and reload the stored value into the PC.

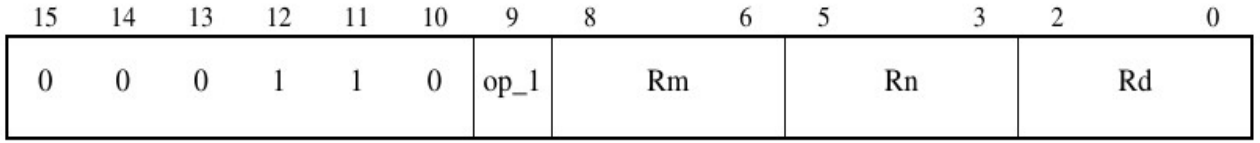
Расширения ARM

- Thumb -- 16 битные инструкции
- Jazelle -- декодирование байткода Java

Кодирование команд Thumb

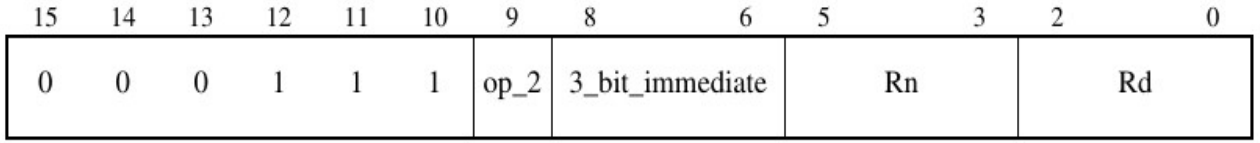
Format 1

```
<opcode1>  <Rd>, <Rn>, <Rm>
<opcode1> := ADD | SUB
```



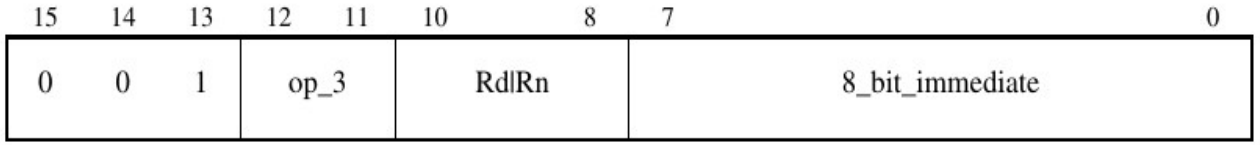
Format 2

```
<opcode2>  <Rd>, <Rn>, #<3_bit_immed>
<opcode2> := ADD | SUB
```



Format 3

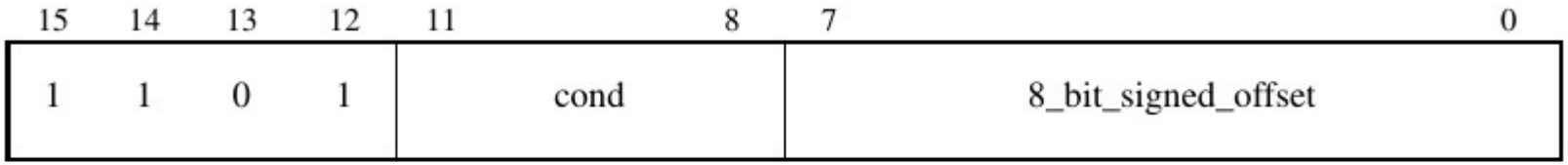
```
<opcode3>  <Rd>|<Rn>, #<8_bit_immed>
<opcode3> := ADD | SUB | MOV | CMP
```



Кодирование команд Thumb

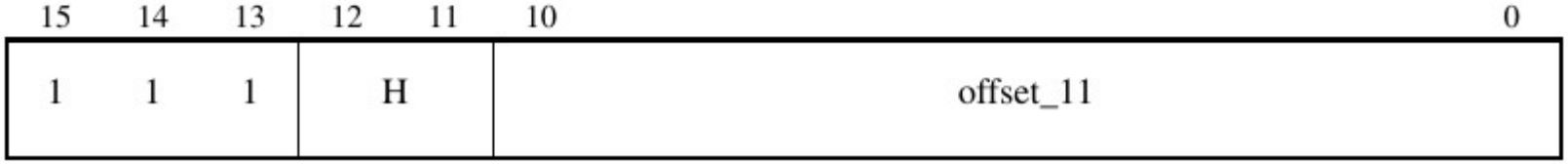
Conditional branch

```
B<cond>  <target_address>
```

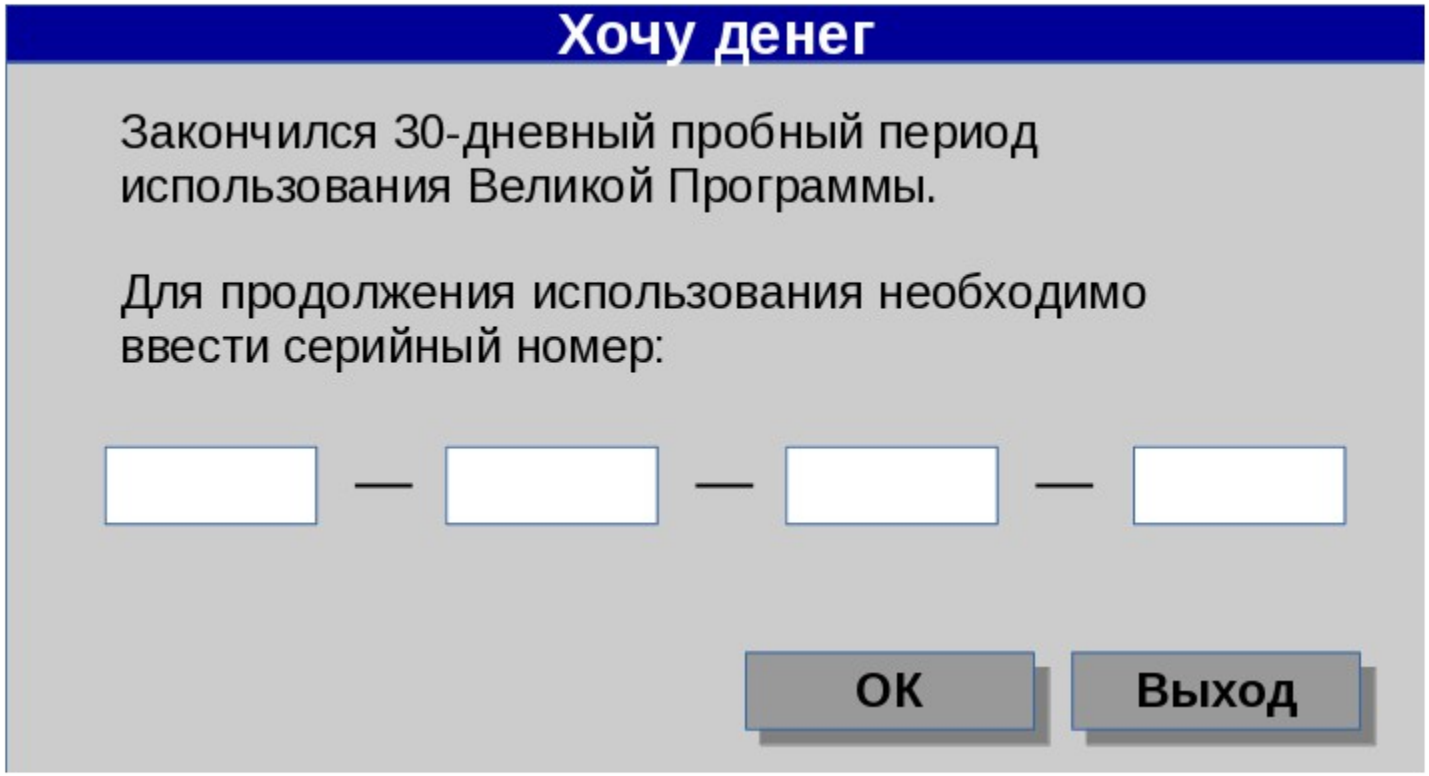


Unconditional branch

```
B    <target_address>
BL   <target_address>          ; Produces two 16-bit instructions
BLX  <target_address>          ; Produces two 16-bit instructions
```



Зачем понимать кодирование команд



Уголовный кодекс РФ, глава 21 «Преступления против собственности»

Статья 159.6. Мошенничество в сфере компьютерной информации

1. Мошенничество в сфере компьютерной информации, то есть хищение чужого имущества или приобретение права на чужое имущество путем ввода, удаления, блокирования, модификации компьютерной информации либо иного вмешательства в функционирование средств хранения, обработки или передачи компьютерной информации или информационно-телекоммуникационных сетей, - наказывается штрафом в размере до ста двадцати тысяч рублей или в размере заработной платы или иного дохода осужденного за период до одного года, либо обязательными работами на срок до трехсот шестидесяти часов, либо исправительными работами на срок до одного года, либо ограничением свободы на срок до двух лет, либо принудительными работами на срок до двух лет, либо арестом на срок до четырех месяцев.

Как ломают

```
void TrialEndDialog_OkPressed()
{
  const char * sn =
Dialog_GetSerialNumber();
  if (CheckSerialNo(sn)) {
    ContinueLaunch();
  }
  else {
    ShowErrorMessage();
    ExitProgram();
  }
}
```

Как ломают

```
. . . . . .
01: callq   CheckSerialNo    ;; вызов [bool CheckSerialNo]
02: testb   $1, %al          ;; сравнение результата с 1
03: jejmp   $05              ;; если OK, то переход к 05
04: jmp     $08              ;; переход к 08
05: movb$0, %al              ;; очистка регистра AL
06: callq   ContinueLaunch   ;; вызов [bool ContinueLaunch]
07: jmp     $0C              ;; переход к строке после if {}
08: movb$0, %al              ;; очистка регистра AL
09: callq   ShowErrorMessage ;; вызов [void ShowErrorMessage]
0A: movb$0, %al              ;; очистка регистра AL
0B: callq   ExitProgram      ;; вызов [void ExitProgram]
. . . . . .
```

Как ломают

```
. . . . . .
01: callq    CheckSerialNo     ;; вызов [bool CheckSerialNo]
02: testb    $1, %al           ;; сравнение результата с 1
03: je       $05               ;; если OK, то переход к 05
04: jmp      $08               ;; переход к 08
05: movb$0, %al                ;; очистка регистра AL
06: callq    ContinueLaunch    ;; вызов [bool ContinueLaunch]
07: jmp      $0C               ;; переход к строке после if {}
08: movb$0, %al                ;; очистка регистра AL
09: callq    ShowErrorMessage  ;; вызов [void ShowErrorMessage]
0A: movb$0, %al                ;; очистка регистра AL
0B: callq    ExitProgram       ;; вызов [void ExitProgram]
. . . . . .
```