Лекция 12
Ассемблер, часть 1

Программа на ассемблере

- Суффикс .s (например, hello.s) – чистый ассемблер
- Суффикс .S (hello.S) – препроцессор C + ассемблер, т. е. можно использовать #define, #include, #if и т. д.
- Компиляция:
  hello.s → [as] → hello.o → [ld] → hello
- Простой вариант:
  gcc hello.s -ohello
- Gcc передаст много опций (нужных и ненужных) в ld, смотри gcc -v

Варианты синтаксиса x86/x64

- Синтаксис AT&T – используется в мире Unix по умолчанию, as по умолчанию
  - .att_syntax // в начале .S-файла
  - .att_syntax noprefix // позволяет писать названия регистров без %, например eax вместо %eax
- Синтаксис Intel – используется в документации Intel, Windows – более распространенный
  - .intel_syntax

Отличия AT&T и Intel

- Разный порядок аргументов!
  Например, пересылка значения (присваивание) регистров: eax = ebx
  - mov %ebx, %eax // AT&T, откуда → куда
    movl %ebx, %eax // явное указание размера
  - mov eax, ebx // Intel куда ← откуда
- Разная форма записи адресного выражения
  - incb arr(%eax, %ebx, 8) // AT&T
  - inc byte ptr [arr + eax + ebx * 8] // Intel

Адресное пространство

- Архитектура фон Неймана – оперативная память состоит из ячеек, каждая из которых имеет свой индивидуальный номер (адрес)
- Различаются “виртуальное” и “физическое” адресные пространства (но это мы рассмотрим позже), пока работаем с адресным пространством программ на C, C++, asm…
- На x86, armv7 адресное пространство 32-битное, то есть адрес – число типа uint32_t
- На x64, armv7 адресное пространство 48-битное, адрес хранится в типе int64_t

Размещение кода и данных в программе

- Статическое размещение – место в адресном пространстве резервируется и указываются значения ячеек памяти при компиляции программы
  - Неперемещаемое (позиционно-зависимое) – при компиляции программы (точнее, при компоновке в ld) указываются фиксированные адреса в памяти, по которым будут размещаться код и данные
  - При загрузке программы на выполнение она размещается по заданным адресам:
    - X86 – стандартный адрес начала 0x08048000
    - X64 – стандартный адрес начала 0x00400000
    - ARMv7 – стандартный адрес 0x10000
    - см. опцию -Ttext-segment

Перемещаемое (позиционно-независимое) размещение

- Статическое – место в адресном пространстве резервируется при компиляции программы
- Адрес размещения в памяти определяется при загрузке программы на выполнение, каждый раз может быть разным

Абсолютное vs перемещаемое

- ld по умолчанию генерирует позиционно-зависимые (абсолютно размещенные) исполняемые файлы (не .so-библиотеки!)
- В некоторых дистрибутивах Linux в gcc внесены изменения, что по умолчанию генерируется перемещаемый исполняемый файл
  - нужны специальные опции gcc, чтобы генерировать неперемещаемый файл (-fno-pie)

Абсолютное vs перемещаемое

- Перемещаемая программа (position-independent program)
  - Сложнее в написании (напр. нельзя просто использовать адреса глобальных переменных)
  - Может быть чуть медленнее (требуются операции вычисления адресов)
  - При загрузке на выполнение требуется настройка программы на работу по текущим адресам загрузки
- Динамически-загружаемые модули (.so-файлы такие всегда!)
- Если программа каждый раз загружается по новому адресу, это может служить дополнительной защитой от атак на программу (Address space layout randomization - ASLR)

Структура единицы трансляции

- Программа состоит из секций – логических частей программы
- Компоновщик объединяет содержимое секций из входных объектных файлов, размещает секции в исполняемом файле
- Стандартные секции (минимальный набор)
  - .text – код программы и read-only data
  - .data – глобальные переменные
  - .bss – глобальные переменные, инициализированные нулем

Дополнительные секции

- Можно определять секции с произвольными именами
- Стандартные дополнительные секции:
  - .rodata
    .section .rodata, “a”
- Нестандартные секции:
  - .string для размещения строк:
    .section .string, “aMS”, @progbits, 1

Правила использования секций

- Программный код должен размещаться в секции .text
- Константы и константные строки могут размещаться в .text или в .rodata
- Глобальные переменные размещаются в .data или .bss

Определение данных

- Глобальные переменные:

```
.byte 1, 2, 3, '\n'
.short 10, 11
.int 0xff00ff00
.quad  -1
.float   1.5
.double    2.0
```

- Строки

```
.ascii “abc”
.asciz  “Hello”     // добавляется неявный \0
```

- Резервирование памяти под массив

```
.skip 4 * 1024, 0
```

Метки (labels, symbols)

- Метки – это символические константы, значение которых известно при компиляции или компоновке программы
  - Метка как адрес, по которому размещается инструкция при выполнении программы
  - Метка как константное значение
- По умолчанию метки видны только в текущей единице компиляции (в том числе объявленные после использования)
- Чтобы сделать метку доступной компоновщику используется .global NAME

Точка входа в программу

- Программа должна иметь точку входа – метку, на которую передается управление в начале выполнения программы
- Если компилируем без стандартной библиотеки (-nostdlib), точка входа должна называться _start и должна экспортироваться (.global _start)
- Если компилируем со стандартной библиотекой, точка входа называется main и должна экспортироваться (.global main)

Регистры процессора

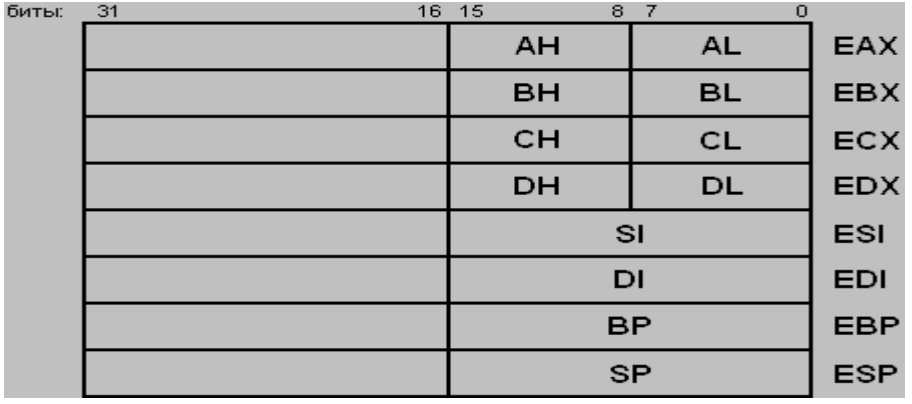
- Регистры процессора – ячейки памяти, находящиеся в процессоре
  - Очень быстрые
  - Их мало или очень мало
  - Несколько функциональных групп регистров
- Вся совокупность регистров – регистровый файл (register file)
- Регистры имеют “индивидуальные” имена

Регистры общего назначения

- General purpose register (GPR)
- Используются для:
  - Хранения аргументов для операций
  - Сохранения результатов операций
  - Хранения адреса или индекса для косвенного обращения к памяти или косвенных переходов
  - Размещения наиболее часто используемых переменных
- X86 – 8 32-битных регистров общего назначения

Регистры общего назначения

- 32-битные %eax, … %esp
- %esp – указатель стека
- 16-битные %ax… %bp
- 8-битные %al, …
- %ebp обычно указатель кадра стека, но может быть GPR
- Регистры неоднородны – в некоторых инструкциях используются фиксированные регистры



Специфика РОН

- %eax – 32-битный “аккумулятор”
- %eax:%edx – пара регистров как 64-битный “аккумулятор”
- %cl – счетчик сдвига
-
- X64 – 16 64-битных регистров rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8 … r15
- Их можно использовать и для хранения 32-, 16-, 8-битных значений
- В некоторых случаях можно использовать rip – текущий адрес в программе

Управляющие регистры

- %eip – (instruction pointer) – адрес инструкции, следующей за текущей
- %eflags – регистр флагов процессора
- %cr0 … %cr4 - прочие управляющие регистры
- %dr0 … %dr4 – отладочные регистры
- %cs, %ss, %ds, %es, %fs, %gs – “сегментные” регистры

Floating-point registers

- %st(0) … %st(7) – регистры FPU – каждый имеет размер 80 бит – в настоящее время deprecated
- %mm0 … %mm7 – регистры MMX (deprecated)
- %xmm0 … %xmm7 – регистры SSE
- %ymm0 …, %zmm0 ... - AVX
- SIMD – single instruction multiple data – за одну инструкцию обрабатывается несколько значений

Инструкция процессора

- Инструкция – минимальное цельное действие, которое может выполнить процессор
- Выполняется или не выполняется целиком, то есть не изменяет состояние процессора “частично” по сравнению с описанием (ну почти – см. Meltdown)
- Инструкции исполняются последовательно, кроме инструкций передачи управления

Инструкция mov

- Пересылка данных
  movSFX SRC, DST
- Куда пересылаем – второй аргумент!
- SFX – размер пересылаемых данных:
  - 'movb – байт
  - 'movw – слово (16 бит)
  - 'movl – двойное слово (32 бит)
  - 'movq – 64 бит
- Типы пересылок:
  - Регистр-регистр
  - Регистр-память
  - Память-регистр

Методы адресации

- Возможные типы аргументов операции определяются поддерживаемыми процессором методами адресации
- Методы адресации:
  - Регистровый – указывается имя регистра
    movl %esp, %ebp
  - Непосредственный (immediate) – аргумент задается в инструкции – знак $
    movb **$16**, %cl
  - Прямой (direct) – адрес ячейки памяти задается в инструкции
    movl %eax, var1

Методы адресации

- Методы адресации (продолжение)
  - Косвенный (indirect) только для jmp и call
    call *%eax
  - Относительный (relative) только для jmp и call
    jmp loop
    вычисляется смещение относительно текущего значения EIP и адреса метки, в инструкции сохраняется смещение; при выполнении перехода восстанавливается абсолютный адрес перехода
  - Обращение к памяти общего вида OFFSET(BREG, IREG, SCALE) – рассмотрим позже

Преобразования целых

- Расширение нулями:
  ```
  movzbl    var, %eax   // 8 → 32 бита
  movzwl    var, %eax   // 16 → 32 бита
  ```
- Расширение знаковым битом:
  ```
  movsbl    var, %eax
  movswl    var, %eax
  cdq                   // eax → eax:edx
  ```

Арифметика

- Арифметические инструкции:

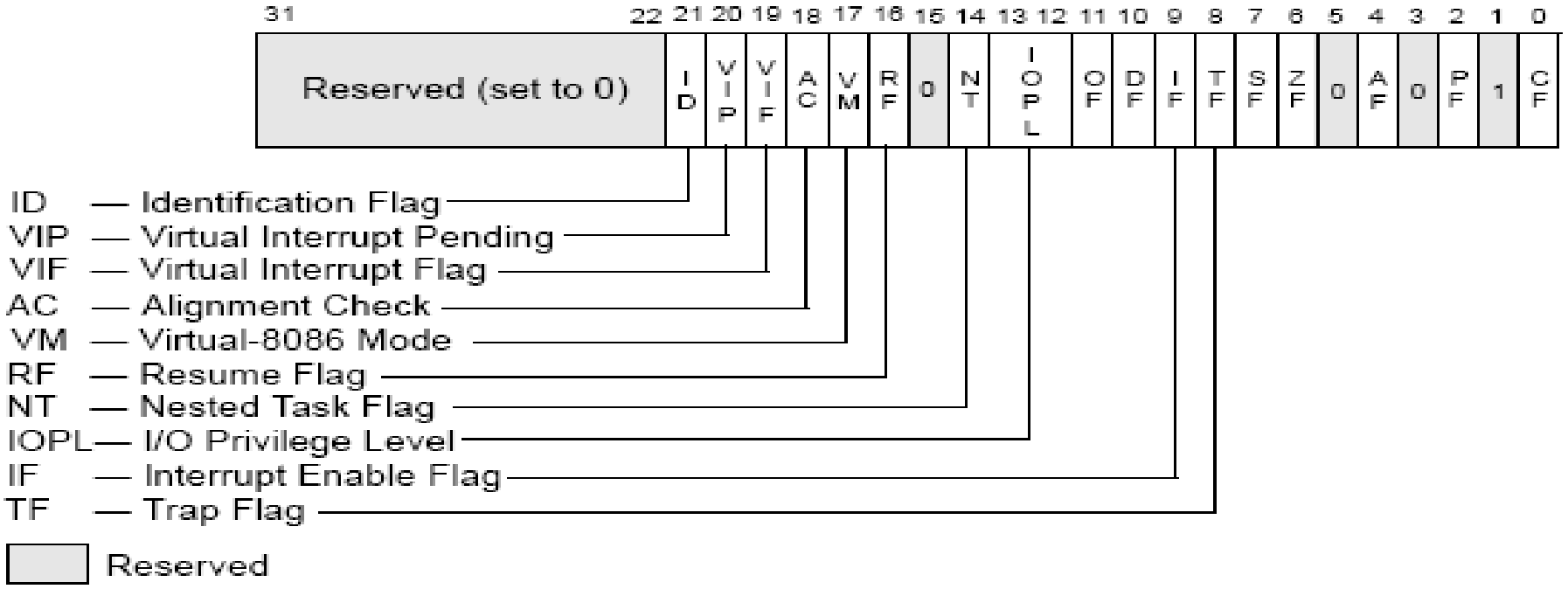
```
add     SRC, DST      // DST += SRC
sub     SRC, DST      // DST -= SRC
cmp     SRC1, SRC2  // SRC2 – SRC1
and     SRC, DST       // DST &= SRC
or   SRC, DST          // DST |= SRC
xor SRC, DST       // DST ^= SRC
test     SRC1, SRC2 // SRC1 & SRC2
not DST                // DST = ~DST
neg     DST                 // DST = -DST
inc DST                // ++DST
dec     DST                 // --DST
```

Флаги результата операции

- Регистр EFLAGS содержит специальные биты-флаги результата операции
- Для x86 они называются: ZF, SF, CF, OF
  - ZF (бит 6) – флаг нулевого результата
  - SF (бит 7) – флаг отрицательного результата
  - CF (бит 0) – флаг переноса из старшего бита
  - OF (бит 11) – флаг переполнения

Регистр EFLAGS

- Нас интересуют: CF, ZF, SF, OF



Примеры

1(1) + 2(2) = 3(3), ZF=0, SF=0, CF=0, OF=0

0(0) + 0(0) = 0(0), ZF=1, SF=0, CF=0, OF=0

130(-126)+0(0)=130(-126), ZF=0,SF=1,CF=0, OF=0

130(-126)+126(126)=0(0), ZF=1, SF=0, CF=1, OF=0

127(127)+127(127)=254(-2), ZF=0,SF=1,CF=0,OF=1

Сдвиги

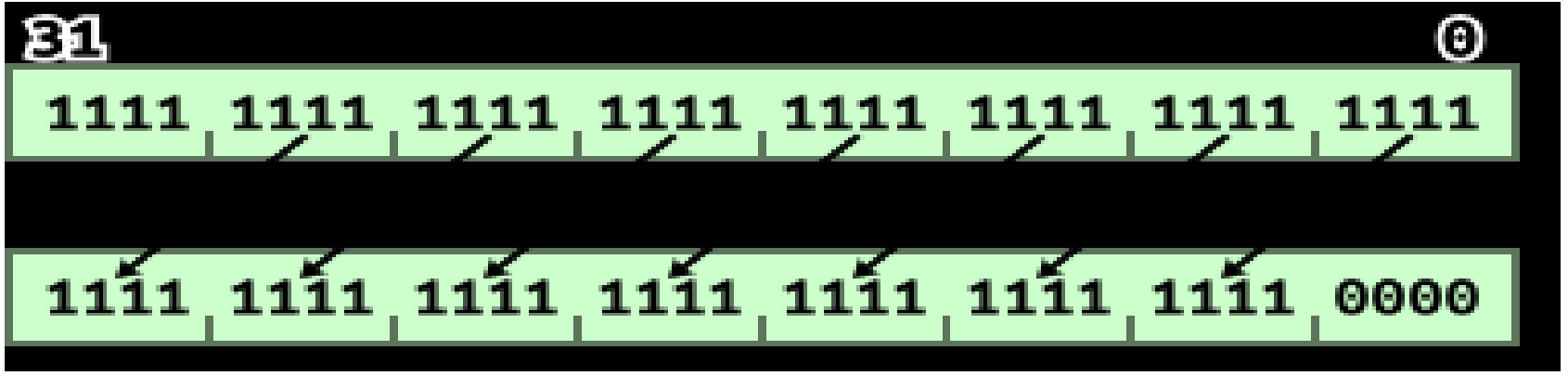
- Арифметические сдвиги влево/вправо
```
sal %eax          // %eax <<= 1
sal $2, %eax      // %eax <<= 2
sal %cl, %eax     // %eax <<= %cl & 0x1F
sar %eax          // %eax >>= 1
sar $5, %eax      // ...
sar %cl, %eax     // ...
```
- Логические сдвиги влево/вправо
```
shl [CNT, ] DST   // сдвиг влево
shr [CNT, ] DST // сдвиг вправо
```

Вращения

- Вращение влево/вправо
  rol [CNT,] DST
  ror [CNT,] DST
- Вращение через CF влево/вправо
  rcl [CNT,] DST
  rcr [CNT,] DST

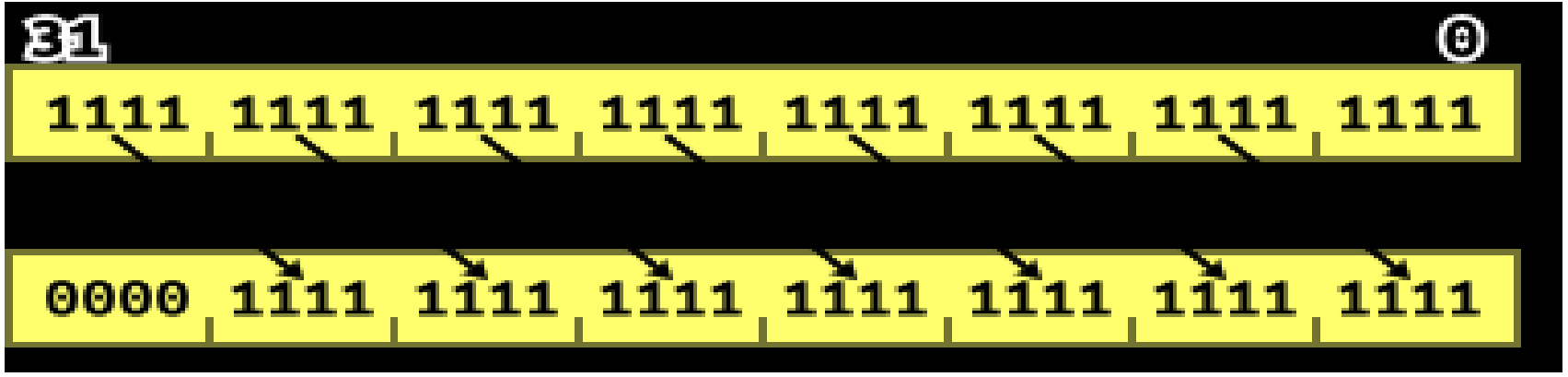
asl/lsl

- asll $4, %eax



lsr

- LSRL $4, %eax

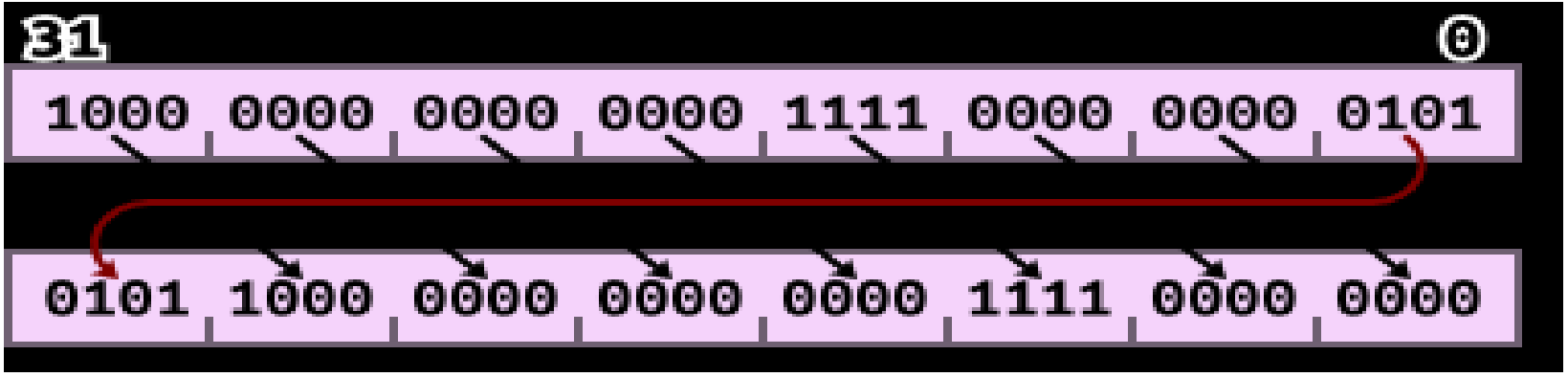


Arithmetical Shift Right

- ASRL $4, %eax

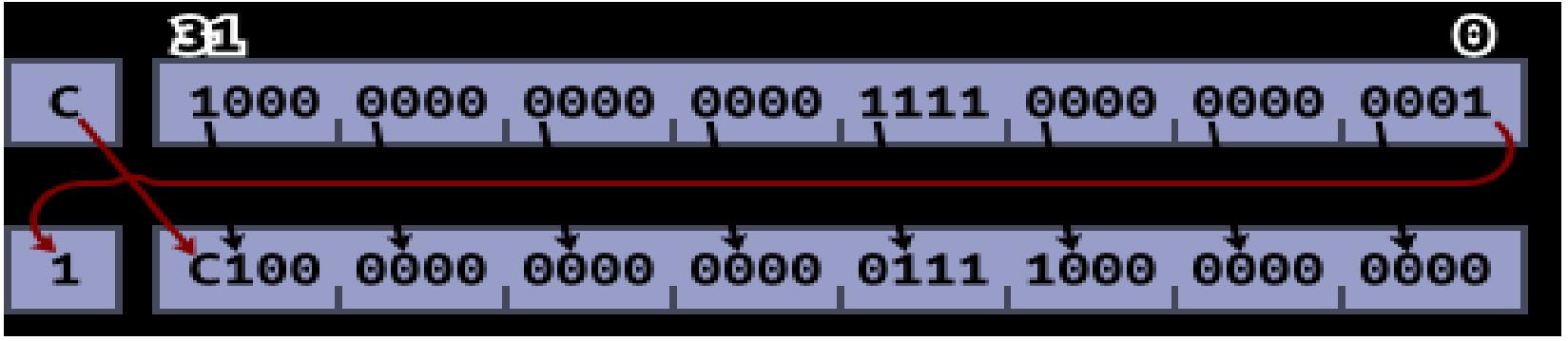
ror

- RORL $4, %eax



rcr

- RCRL %eax


31
0
C
1000 0000 0000 0000 1111 0000 0000 0001
1
C100 0000 0000 0000 0111 1000 0000 0000

Передача управления

- Безусловная передача управления:
  JMP ADDR/LABEL
- Условная передача управления
  J*cc* ADDR/LABEL
  (рассмотрим далее)
- Вызов подпрограммы
  CALL ADDR/LABEL
- Программное прерывание
  INT NUM
  (примерно как CALL)

Условные переходы

- Переход на метку выполняется только если установлена соответствующая комбинация флагов результата
- Условные переходы по равенству/неравенству
  JE / JZ переход если == или 0
  JNE / JNZ переход если != или не 0

Условные переходы

- Для операций с беззнаковыми числами
  JA / JNBE переход если >
  JAE / JNB / JNC переход если >=
  JB / JNAE / JC переход если <
  JBE / JNA переход если <=
- Для операция со знаковыми числами
  JG / JNLE переход если >
  JGE / JNL переход если >=
  JL / JNGE переход если <
  JLE / JNG переход если <=

Условные переходы

- Специальные случаи
  JO переход если OF == 1
  JNO переход если OF == 0
  JS переход если SF == 1
  JNS переход если SF == 0