

гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru

Программирование на языках Ассемблера

Лекция 1

Синёв Николай Иванович
Кафедра 14



История Ассемблера

```

MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER PAGE  2

C000          ORG    ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START   LDS    #STACK

*****+
* FUNCTION: INITA - Initialize ACIA
* INPUT: none
* OUTPUT: none
* CALLS: none
* DESTROYS: acc A

0013        RESETA EQU    %00010011
0011        CTLREG EQU    %00010001

C003 86 13  INITA   LDA A #RESETA  RESET ACIA
C005 B7 80 04
C008 86 11  LDA A #CTLREG  SET 8 BITS AND 2 STOP
C00A B7 80 04  STA A ACIA

C00D 7E C0 F1  JMP     SIGNON  GO TO START OF MONITOR

*****+
* FUNCTION: INCH - Input character
* INPUT: none
* OUTPUT: char in acc A
* DESTROYS: acc A
* CALLS: none
* DESCRIPTION: Gets 1 character from terminal

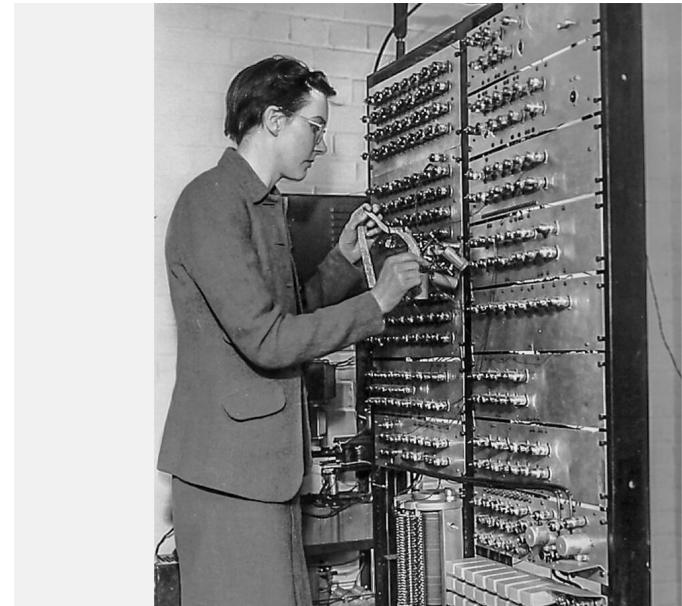
C010 B6 80 04  INCH    LDA A ACIA    GET STATUS
C013 47          ASR A      SHIFT RDRF FLAG INTO CARRY
C014 24 FA          BCC      INCH    RECEIVE NOT READY
C016 B6 80 05          LDA A ACIA+1  GET CHAR
C019 84 7F          AND A #\$7F  MASK PARITY
C01B 7E C0 79          JMP     OUTCH  ECHO & RTS

*****+
* FUNCTION: INHEX - INPUT HEX DIGIT
* INPUT: none
* OUTPUT: Digit in acc A
* CALLS: INCH
* DESTROYS: acc A
* Returns to monitor if not HEX input

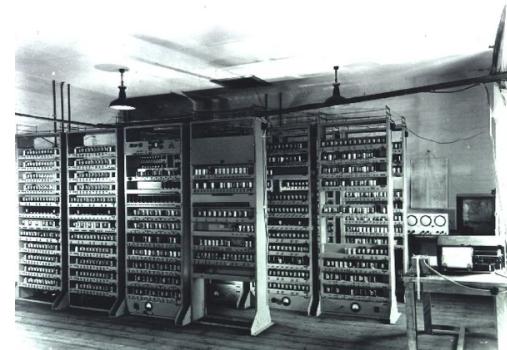
C01E 8D F0  INHEX   BSR     INCH    GET A CHAR
C020 81 30          CMP A #'0  ZERO
C022 2B 11          BMI     HEXERR  NOT HEX
C024 81 39          CMP A #'9  NINE
C026 2F 0A          BLE     HEXRTS  GOOD HEX
C028 81 41          CMP A #'A
C02A 2B 09          BMI     HEXERR  NOT HEX
C02C 81 46          CMP A #'F
C02E 2B 05          BGT     HEXERR
C030 80 07          SUB A #7  FIX A-F
C032 84 0F  HEXRTS  AND A #\$0F  CONVERT ASCII TO DIGIT
C034 39          RTS

C035 7E C0 AF  HEXERR  JMP     CTRL   RETURN TO CONTROL LOOP

```



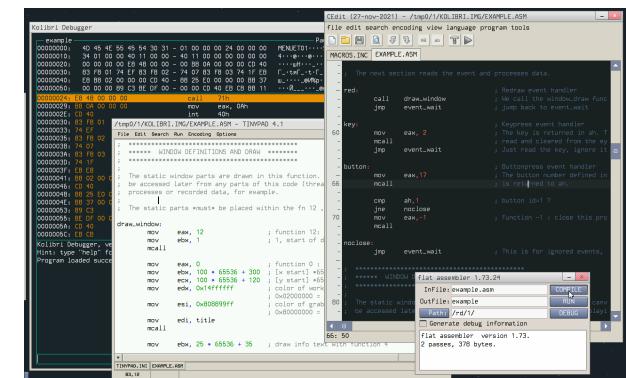
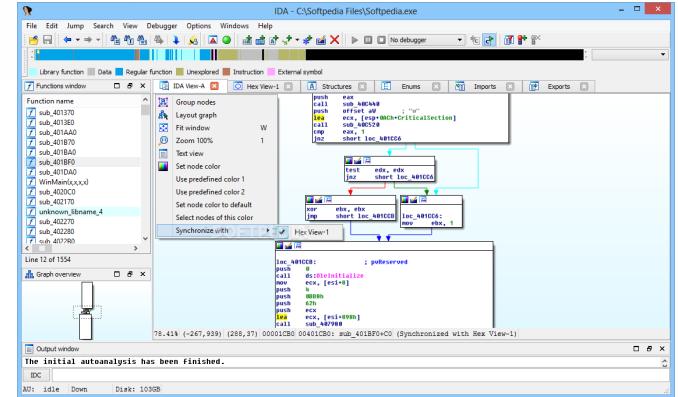
Кэтлин Бут
1922-2012



Для ARC 1947 год, для EDSAC - 1948

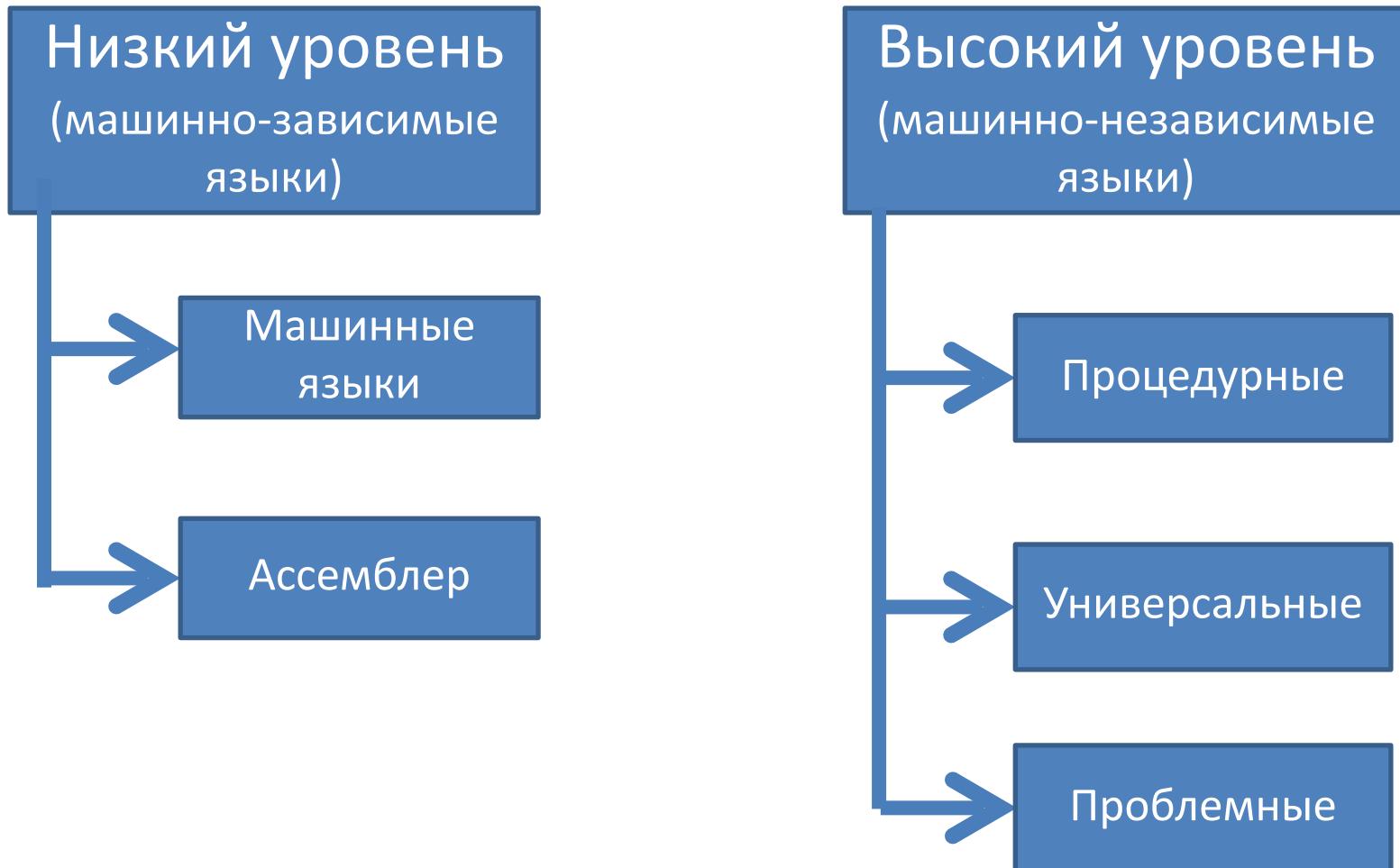
Использование ассемблера

- Разработка загрузчиков для различных систем (BIOS, UEFI)
- Использование в реверс-инжениринге бинарных и программных кодов
- Разработка под старые ОС и системы
- Поиск и отладка ошибок в релизных продуктах
- Разработка полноценных ОС и софта под них
- Хакинг и взлом софта



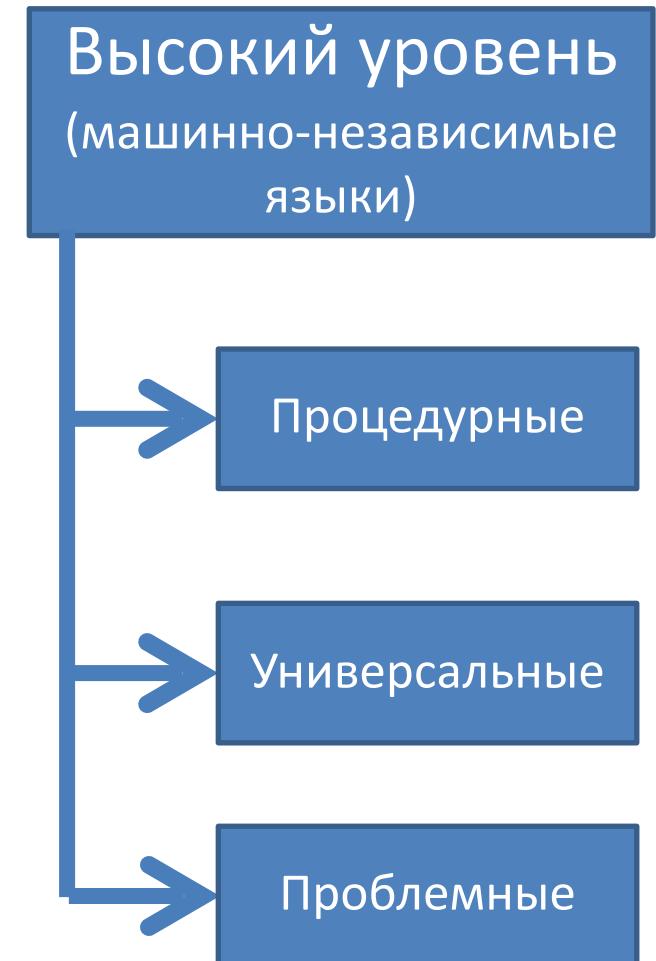
Введение в ассемблер

Классификация языков программирования



Классификация языков программирования

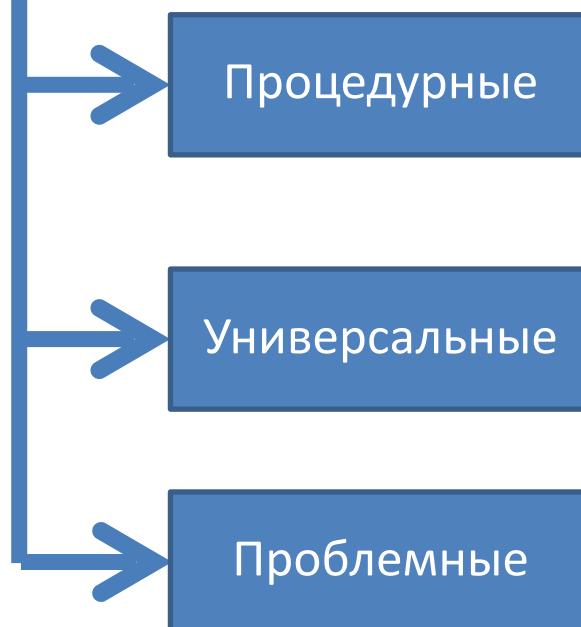
- + алфавит языка значительно шире машинного
- + набор операций выбирается из соображений удобства формулирования алгоритмов решения задач определенного класса
- + конструкции команд (операторов) отражают содержательные виды обработки данных и задаются в удобном для человека виде
- + используется аппарат переменных и действия с ними
- + поддерживается широкий набор типов данных



Классификация языков программирования

Требуют использования соответствующих программ-переводчиков (**компилятор**/**транслятор**) для представления программы на языке машины, на которой она будет исполняться

Высокий уровень
(машино-независимые языки)



Классификация языков программирования

Низкий уровень
(машино-зависимые языки)

Машинные языки

Ассемблер

Все языки низкого уровня ориентированы на определенный тип компьютера

Машинный язык – это совокупность машинных команд, которая отличается количеством адресов в команде, назначением информации, задаваемой в адресах, набором операций, которые может выполнять машина.

- процесс написания программы на машинном языке очень трудоемкий и утомительный

+ Контроль каждой команды и каждой ячейки памяти

Классификация языков программирования

Низкий уровень
(машино-зависимые
языки)

Машинные
языки

Ассемблер

Перевод программы с языка ассемблера на машинный язык осуществляется специальной программой, которая называется **ассемблером** и является, по сути, простейшим транслятором.

Классификация языков программирования

Низкий уровень
(машино-зависимые языки)

Машинные языки

Ассемблер

Язык ассемблера – это символьический язык, позволяющий описать исходную программу непосредственно на уровне команд процессора.

Преобразование записанных на языке ассемблера команд и данных в машинные коды называется **трансляцией** (ассемблирование).

Пример ассемблерного кода

Hello, world! для Linux x86 (AT&T-синтаксис)

```
.data
msg:
.ascii "Hello, world!\n"
len = . - msg      # символу len присваивается длина строки

.text
.global _start      # точка входа в программу
_start:
    movl $4, %eax      # системный вызов № 4 – sys_write
    movl $1, %ebx      # поток № 1 – stdout
    movl $msg, %ecx    # указатель на выводимую строку
    movl $len, %edx    # длина строки
    int $0x80          # вызов ядра

    movl $1, %eax      # системный вызов № 1 – sys_exit
    xorl %ebx, %ebx    # выход с кодом 0
    int $0x80          # вызов ядра
```

ТУФ 逢X80 # ВРІЗОВ 87b9
ХОЛг 86P 逢X # ВРІХОД С КОДОМ 0
ШОЛг 869X` 逢 # СНСЛЕННРІН ВРІЗОВ 逢 – sys_exit

Hello, world! для Linux ARM (EABI)

```
.data
msg:
.ascii "Hello, world!\n"
len = . - msg      @ в GAS для ARM комментарии начинаются с @ или заключаются в /* */

.text
.global _start      @ точка входа в программу
_start:
    mov r7, #4      @ системный вызов № 4 – sys_write
    mov r0, #1      @ поток № 1 – stdout
    ldr r1, =msg    @ указатель на выводимую строку
    ldr r2, =len    @ длина строки
    swi #0          @ вызов ядра

    mov r7, #1      @ системный вызов № 1 – sys_exit
    mov r0, #0      @ выход с кодом 0
    swi #0          @ вызов ядра
```

СМТ #0 @ ВРІЗОВ 87b9
ШОЛ L0` #0 @ ВРІХОД С КОДОМ 0
ШОЛ L1` #1 @ СНСЛЕННРІН ВРІЗОВ 逢 – sys_exit

Hello, world! для Linux x86 (Intel-синтаксис)

```
.intel_syntax
.data
msg:
.ascii "Hello, world!\n"
len = . - msg      # символу len присваивается длина строки

.text
.global _start      # точка входа в программу
_start:
    mov %eax, 4      # системный вызов № 4 – sys_write
    mov %ebx, 1      # поток № 1 – stdout
    mov %ecx, OFFSET FLAT:msg    # указатель на выводимую строку
                                # OFFSET FLAT означает использовать тот адрес,
                                # который msg будет иметь во время загрузки
    mov %edx, len    # длина строки
    int 0x80          # вызов ядра

    mov %eax, 1      # системный вызов № 1 – sys_exit
    xor %ebx, %ebx    # выход с кодом 0
    int 0x80          # вызов ядра
```

ТУФ 0X80 # ВРІЗОВ 87b9
ХОЛг 86P 0 # ВРІХОД С КОДОМ 0
ШОЛ 869X` 1 # СНСЛЕННРІН ВРІЗОВ 逢 – sys_exit

Linux x86 GNU Assembler inline in C

```
int main()
{
    int sum = 0, x = 1, y = 2;
    asm ("add %1, %0" : "=r"(sum) : "r"(x), "0"(y)); // sum = x + y;
    printf("sum = %d, x = %d, y = %d", sum, x, y); // sum = 3, x = 1, y = 2
    return 0;
}
```

}

Легенда:

Классификация языков программирования

Отличие транслятора от компилятора

Задача **одна**: создать машинный код из исходного символьного текста программы

Есть отличие

Транслятор преобразует одну символьическую команду ассемблера в одну машинную команду – преобразование **«один к одному»**.

Компилятор с языка высокого уровня преобразует один символьический оператор языка в последовательность из нескольких машинных команд – преобразование **«один к нескольким»**.

Чем выше уровень языка программирования, тем к большей последовательности команд процессора будут преобразовываться операторные конструкции языка

Работа в машинных кодах

- Программы хранятся в оперативной памяти в машинных кодах
- Биты обрабатывают группами фиксированного размера
- Группы по n бит могут записываться и считываться за одну базовую операцию
- **Группа из n бит** называется **словом** информации, а **значение n – длиной** слова
- Длина слова современных компьютеров составляет 16-64 бит. Восемь последовательных битов называются байтом.

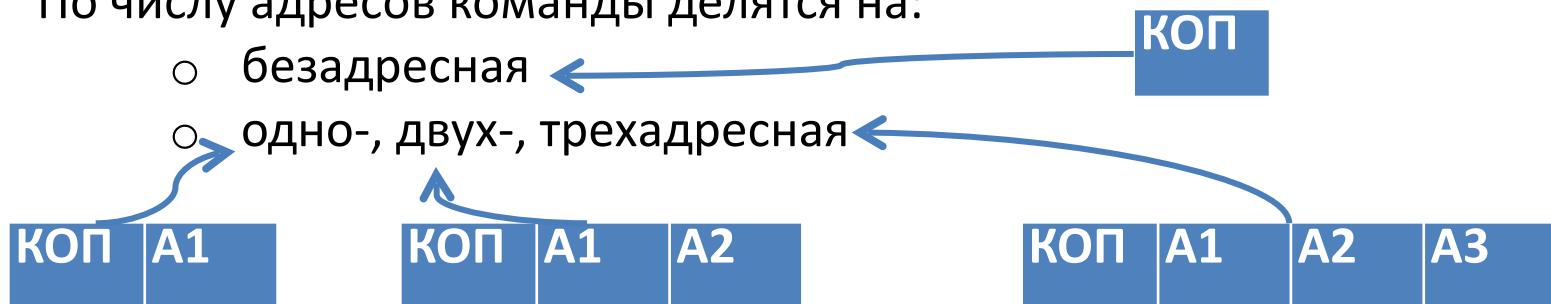
1 килобайт (Кбайт) = 2^{10} = 1 024 байт

1 мегабайт (Мбайт) = 2^{10} Кбайт = 2^{20} байт = 1 048 576 байт

1 гигабайт (Гбайт) = 2^{10} Мбайт = 2^{30} байт = 1 073 741 824 байт

Работа в машинных кодах

- Программа состоит из команд, при исполнении которых происходит выполнение задуманного алгоритма
- Для работы программы необходимо знать адреса слов, по которому данные хранятся
- Распределение полей в формате команды может изменяться при смене способа адресации. Длина команды зависит от числа адресных полей.
- По числу адресов команды делятся на:
 - безадресная
 - одно-, двух-, трехадресная



КОП – код операции; **A1, A2, A3** – адреса operandов

Работа в машинных кодах

! У каждой машины **своя** система команд

КОП	Содержание
01	Ввод
02	Печать
03	Сложение
04	Умножение
05	Деление
06	Останов
07	Чтение в регистр
08	Запись из регистра в ОЗУ

Своя
гипотетическая
система команд



Рассмотрим работу в
машинных командах для
расчета примера

$$y = (a + b)/c^2$$

Работа в машинных кодах

Пример для 3-х адресной машины; $y = (a + b)/c^2$

Адрес команды	Операция				Примечание	Мнемонический код
	КОП	A1	A2	A3		
0001	01	50	00	00	Ввод А	
0002	01	51	00	00	Ввод В	
0003	01	52	00	00	Ввод С	
0004	03	50	51	53	В ячейку 53 A+B	Сложение A,B,Y.
0005	04	52	52	54	В ячейку 54 C*C	Умножение C,C,Re.
0006	05	53	54	53	В ячейку 53 Y	Деление Y,Re,Y.
0007	02	53	00	00	Вывод Y	
0008	06	00	00	00	Останов	

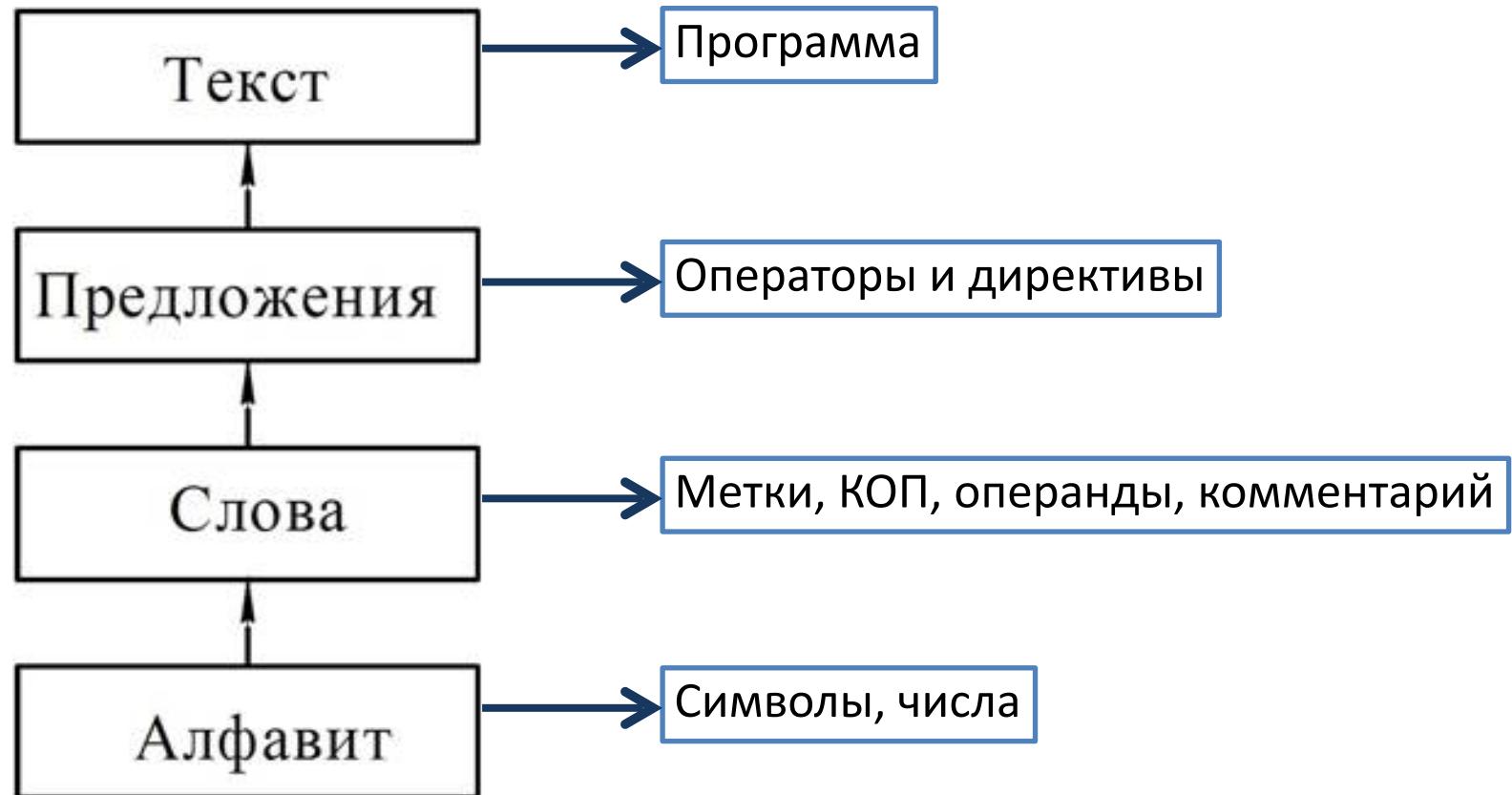
Работа в машинных кодах

Пример для 1-адресной машины; $y = (a + b)/c^2$

В случае 1- и 2-х адресных машин используют регистры

Адрес команды	Операция		Примечание	Мнемонический код
	КОП	A1		
0001	01	50	Ввод А	
0002	01	51	Ввод В	
0003	01	52	Ввод С	
0004	07	52	Чтение С в Re	Чтение С
0005	04	52	B Re C*C	Умножение С
0006	08	53	B яч 53 C*C	Запись Y
0007	07	51	B Re A	Чтение А
0008	03	52	B Re A+B	Сложение В
0009	05	53	B Re Y	Деление Y
0010	08	53	B яч 53 Y	Запись Y
0011	02	53	Печать Y	
0012	06	00	Останов	

Синтаксис ассемблера



Синтаксис ассемблера

Алфавит

- латинские буквы от A до Z,
- цифры от 0 до 9,
- спецсимволы: @ \$? . , ; : () [] “ ‘ пробел и
символ подчеркивания _.



A stylized font sample featuring a variety of characters and symbols. It includes all the letters from A to Z in a cursive, handwritten style. Special characters like '€' and '₹' are also present. There are several punctuation marks: '@', '\$', '?', '.', ',', ';', ':', '(', ')', '[', ']', '“', '‘', ' пробел' (space), and the underlining symbol '_'. The characters are arranged in a loose, vertical cluster.

Синтаксис ассемблера

Слово (1/4)

- **Метка** – это необязательная часть оператора. Метка не должна содержать более 31 символа, начинаться может только с буквы, точки, символа подчеркивания или @.

@frag

Нельзя в качестве метки использовать имена регистров, команд, директив

После трансляции метка получит значение внутрисегментного адреса (смещения) первого байта команды.

Синтаксис ассемблера

Слово (2/4)

- **КОП (код операции).** Это обязательная часть любого оператора, исполняемого или неисполнимого.

Ассемблер имеет ряд операторов, которые позволяют управлять процессом ассемблирования и формирования листинга. Эти операторы называются **директивами**.

Синтаксис ассемблера

Слово (3/4)

- **Операнд(ы).** Это имена элементов, над которыми производятся действия или определяются начальные значения данных.

Количество operandов зависит от машины, на которой идёт работа, а также от кода операции.

Операндов может и не быть.

Синтаксис ассемблера

Слово (4/4)

- **Комментарий.** Признаком комментария является точка с запятой или решетка.

Комментарий может начинаться на любой строке исходного модуля и содержать любые печатные символы, включая пробел и русские буквы.

Все символы после точки с запятой не вырабатывают кодов и только выводятся в листинге.

Длина комментария ограничена длиной строки.

Синтаксис ассемблера

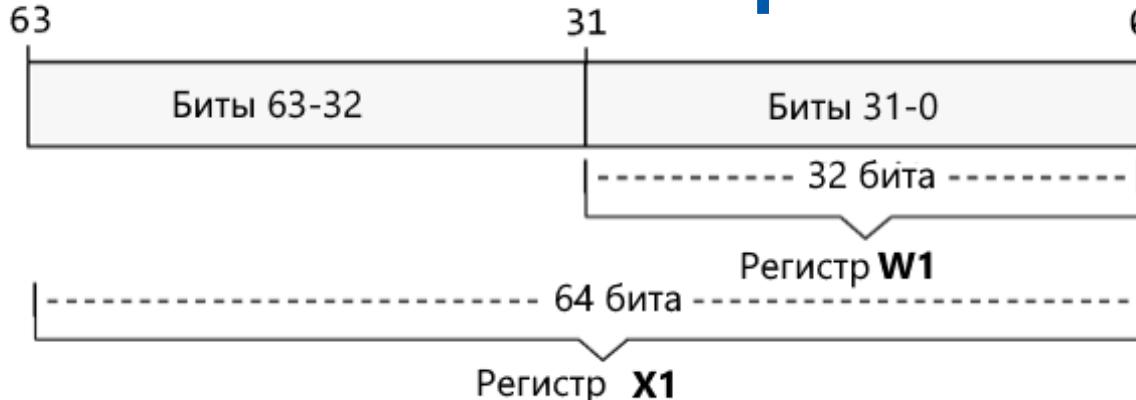
Текст

- это готовая программа

Абсолютная – помещается в место, указанное программистом.

Перемещаемая – её распределяет в свободное памяти загрузчик, также он может разбить программу на куски, обеспечивая ссылки

Регистры ARM



- **X0–X7:** передачи параметров в функцию и возвращения результата.
- **X8:** передачи в функцию адреса блока памяти при непрямой адресации.
- **X9–X15:** следует сохранять при вызове функций. Ответственность за сохранение ложится на вызывающую сторону, которая вызывает функцию. Затронутые регистры сохраняются в кадре стека вызывающей функции, что позволяет подпрограмме изменять эти регистры..
- **X16–X17:** используются для хранения промежуточных результатов между вызовами функций.
- **X18** зарезервирован, и его не следует использовать
- **X19–X28:** - это регистры, сохраняемые вызываемой функцией в стеке, что позволяет функции изменять эти регистры, но также требует их восстановления перед возвратом к вызывающей стороне.
- **X29:** используются как указатель на фрейм стека, еще называется **FP** (frame pointer).
- **X30:** используются для хранения адреса возврата из функции, еще называется **LR** (link register).
- **(FP, X29)** должен хранить валидное значение.

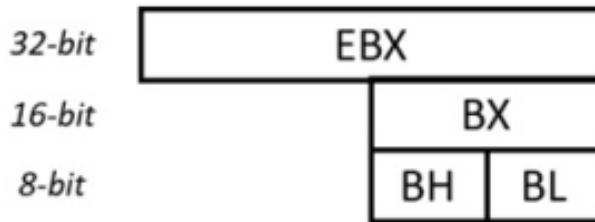
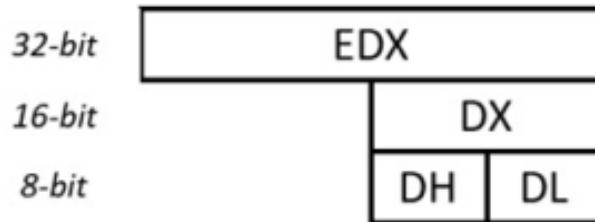
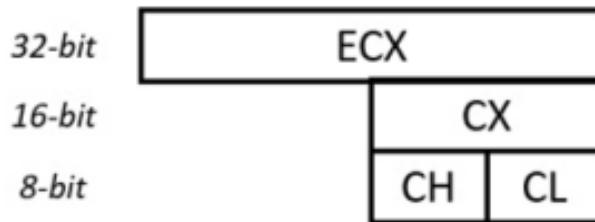
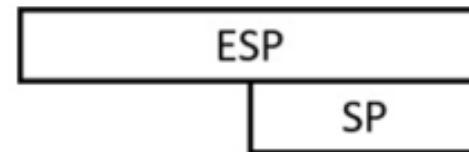
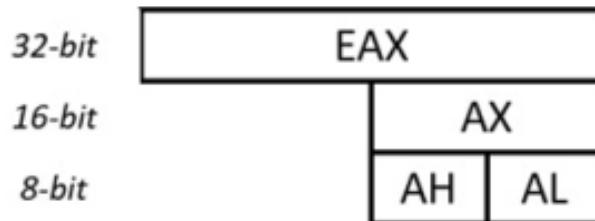
Регистры ARM

- **SP** (Stack Pointer) или указатель стека хранит адрес верхушки стека. 64 бита. **WSP** - 32 битная версия.
- **XZR** - нулевой регистр. 64 бита. **WZR** - 32 битная версия.
- **LR** (Link Register) перекрывает регистр **X30**. Можно свободно использовать для обычных вычислений; однако его основная цель — хранить адреса возврата при вызове функции
- **PSTATE**: передачи в функцию адреса блока памяти при непрямой адресации.

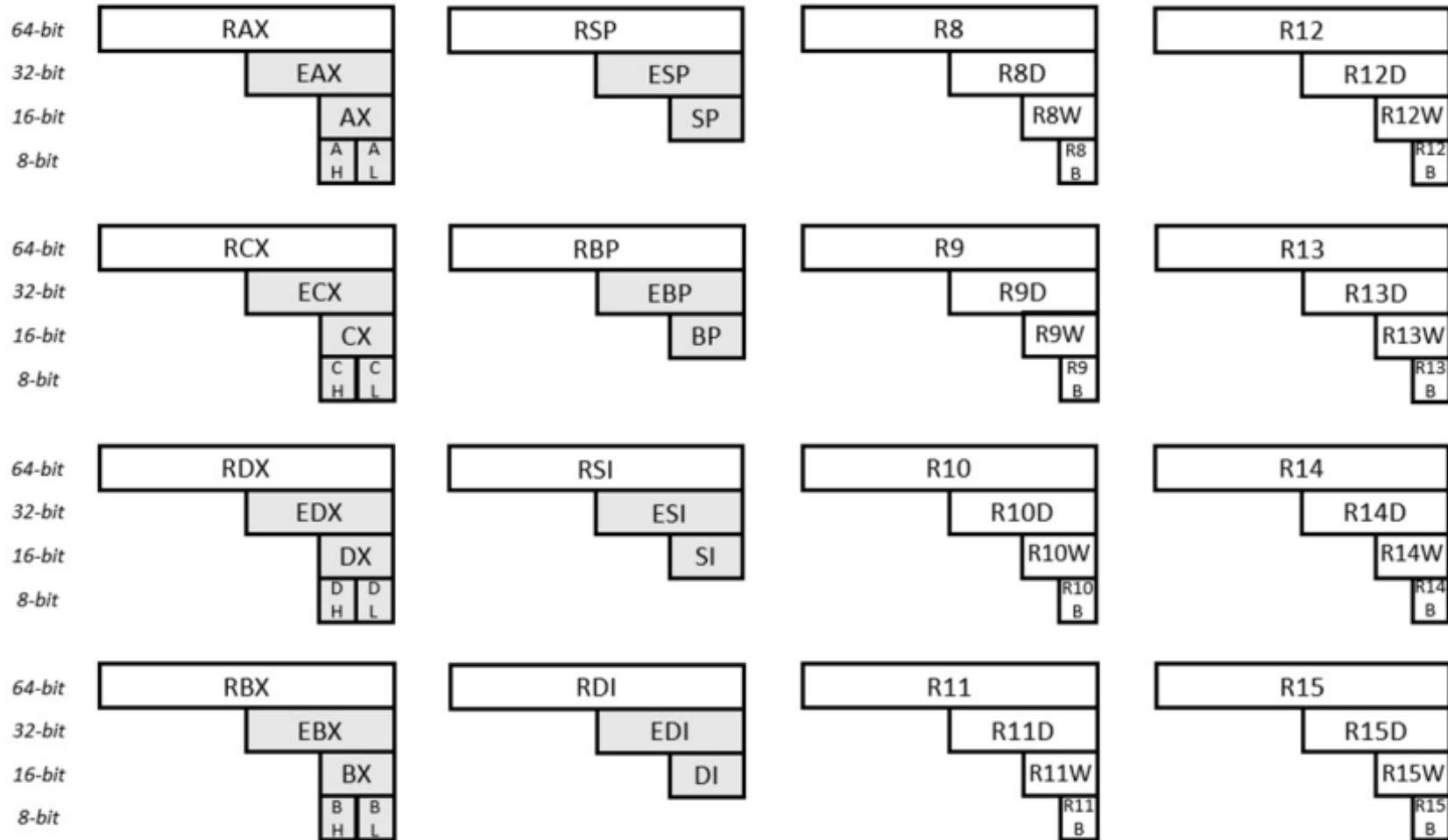


- **N**: флаг знака
- **Z**: флаг нуля
- **C**: флаг переноса
- **V**: флаг переполнения
- **SS**: бит 21- бит одношагового состояния (single-step state bit), используется отладчиками для пошаговой отладки программы
- **IL**: бит 20 - флаг недопустимого состояния исключения
- **DAIF**: биты 9-6 - флаги отключения прерываний

Регистры GAS



Регистры GAS



Регистры GAS

- **EAX** (Accumulator): для арифметических операций
- **ECX** (Counter): для хранения счетчика цикла
- **EDX** (Data): для арифметических операций и операций ввода-вывода
- **EBX** (Base): указатель на данные
- **ESP** (Stack pointer): указатель на верхушку стека
- **EBP** (Base pointer): указатель на базу стека внутри функции
- **ESI** (Source index): указатель на источник при операциях с массивом
- **EDI** (Destination index): указатель на место назначения в операциях с массивами
- **EIP**: указатель адреса следующей инструкции для выполнения
- **EFLAGS**: регистр флагов, содержит биты состояния процессора
- Шестнадцать 64-разрядных регистров RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 и R15
- Шестнадцать 32-разрядных регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D и R15D
- Шестнадцать 16-разрядных регистров AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W и R15W
- Шестнадцать 8-разрядных регистров AL, AH, BL, BH, CL, CH, DL, DH, DIL, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B и R15B

Регистры GAS

- **EAX** (Accumulator): для арифметических операций
- **ECX** (Counter): для хранения счетчика цикла
- **EDX** (Data): для арифметических операций и операций ввода-вывода
- **EBX** (Base): указатель на данные
- **ESP** (Stack pointer): указатель на верхушку стека
- **EBP** (Base pointer): указатель на базу стека внутри функции
- **ESI** (Source index): указатель на источник при операциях с массивом
- **EDI** (Destination index): указатель на место назначения в операциях с массивами
- **EIP**: указатель адреса следующей инструкции для выполнения
- **EFLAGS**: регистр флагов, содержит биты состояния процессора
- Шестнадцать 64-разрядных регистров RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 и R15
- Шестнадцать 32-разрядных регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D и R15D
- Шестнадцать 16-разрядных регистров AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W и R15W
- Шестнадцать 8-разрядных регистров AL, AH, BL, BH, CL, CH, DL, DH, DIL, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B и R15B

Регистры GAS. Регистр флагов RFLAGS

Биты

- **CF** (0): Флаг переноса (Carry flag)
- **PF** (2): Флаг четности
- **AF** (4): Флаг настройки
- **ZF** (6): Флаг нуля (Zero flag)
- **SF** (7): Флаг знака (Sign flag)
- **TF** (8): Флаг прерывания выполнения (Trap flag)
- **IF** (9): Флаг разрешения прерывания
- **DF** (10): Флаг направления обработки разрядов
- **OF** (11): Флаг переполнения (Overflow flag)
- **IOPL** (12-13): Уровень привилегий ввода-вывода (I/O privilege level)
- **NT** (14): Флаг вложенной задачи (Nested task flag)
- **RF** (16): Флаг возобновления (Resume flag)
- **VM** (17): Флаг режима виртуальной машины 8086
- **AC** (18): Флаг проверки выравнивания (Alignment check flag)
- **VIF** (19): Флаг виртуального прерывания (Virtual interrupt flag)
- **VIP** (20): Флаг ожидания виртуального прерывания
- **ID** (21): Флаг ID

Пример программы на ARM64 Apple

```
1 .global _start          // Устанавливаем точку входа в программу для компоновщика
2 .align 2                // Для MacOS требуется выравнивание в 4 байта
3
4 // _start - точка входа в программу
5 _start:
6     mov X0, #1           // значение 1 представляет стандартный поток вывода (консоль)
7     adr X1, message      // передаем адрес строки для вывода на консоль
8     mov X2, #19           // размер строки в байтах
9     mov X16, #4            // номер системного вызова Unix для записи в поток (на консоль)
10    svc #0x80             // вызываем системную функцию с номером 4
11
12 // выход из программы
13    mov X0, #0             // устанавливаем код возврата
14    mov X16, #1            // системный вызов 1 завершает программу
15    svc #0x80             // вызываем системную функцию с номером 1
16
17 message: .ascii "Hello World!\n"
18 message: .ascii "Hello World!\n"
19
20    svc #0X80             // ВЫЗЫВАЕМ СИСТЕМНЮЮ ФУНКЦИЮ С НОМЕРОМ 1
21    mov X0, #1              // СИСТЕМНЫЙ ВЫЗОВ 1 ЗАВЕРШАЕТ ПРОГРАММУ
22    mov X0, #0              // ВОЗВРАЩАЕМ КОД ВОЗВРАТА
23
24    .section .text, "A"       // СЕКЦИЯ ТЕКСТА
```

Пример программы на GAS x86_x64

```
1 .globl _start
2
3 .data
4 message: .asciz "Hello World\n"      # текст выводимого сообщения
5
6 .text
7 _start:
8
9     movq $message, %rsi    # в RSI - адрес строки
10    movq $1, %rdi          # в RDI - дескриптор вывода в стандартный поток (консоль)
11    movq $18, %rdx          # в RDX - длина строки
12    movq $1, %rax          # в RAX - номер функции для вывода в поток
13    syscall                # вызываем функцию Linux
14
15   movq $60, %rax          # в RAX номер системной функции завершения приложения - число 60
16   syscall
17
18   .section .rodata
19   .align 16
20   _exit: .quad 0
21
22   .section .text
23   .align 16
24   _start: .quad _exit
```

Программирование на языках Ассемблера

Лекция 2

Синёв Николай Иванович
Кафедра 14



Секции в исходном коде

ARM64 Apple

```
.text
// основная секция всей программы
// чаще всего опускается по
умолчанию
// все, что объявляется тут – не
меняется

.global _main
// секция ссылается на основную
метку входа
// в MacOs рекомендуется делать
_main

.align 2
// требование MacOs по
обязательному выравниванию в 4
байта

_main:
// исходный код

.data
// секция определения данных и
переменных
// изменяемые данные
```

GAS

```
.globl _start
// секция ссылается на основную
метку входа

.data
// секция определения данных и
переменных
// изменяемые данные

.text
// основная секция всей программы
// чаще всего опускается по
умолчанию
// все, что объявляется тут – не
меняется

_start:
// исходный код
```

Типы данных

GAS

.ascii: строка в двойных кавычках
.asciz: строка ascii, которая заканчивается 0вым байтом
.byte: целое число размером в 1 байт
.short (.word): целое число размером в 2 байта (слово)
.long: целое число размером в 4 байта (так называемое двойное слово)
.quad: целое число размером в 8 байт (четверное слово)
.octa: целое число размером в 16 байт (восьмеричное слово)
.float: число с плавающей точкой одинарной точности
.double: число с плавающей точкой двойной точности

ARM64 Apple

.ascii: строка в двойных кавычках
.asciz: строка ascii, которая заканчивается 0–вым байтом
.byte: целое число размером в 1 байт
.short: целое число размером в 2 байта (так называемое полуслово – hlf-word)
.word: целое число размером в 4 байта (слово)
.quad: целое число размером в 8 байт (двойное слово)
.octa: целое число размером в 16 байт (четверное слово)
.float: число с плавающей точкой одинарной точности
.double: число с плавающей точкой двойной точности

Операция копирования / перемещения

GAS

movq: копирует 64-битное число
movl: копирует 32-битное число
movw: копирует 16-битное число
movb: копирует 8-битное число

```
.globl _start
.text
_start:
    movq $60, %rax    # номер функции для выхода из программы
    movq $0, %rdi     # возвращаемый результат
    syscall           # вызываем системную функцию
```

ARM64 Apple

mov Куда, Откуда

```
.global _main

_main:
    mov X0, #0
    mov X16, #1      // номер функции для выхода из программы
    svc #0x80        // вызываем системную функцию
```

Системный вызов

GAS

```
movq $60, %rax // номер функции для выхода из программы
syscall // вызываем системную функцию (обработчик пребываний)
```

<https://syscalls.mebbeim.net/?table=x86/64/x64/v6.5>

60	0x3c	exit	__x64_sys_exit	kernel/exit.c:989		int error_code			
61	0x3d	wait4	__x64_sys_wait4	kernel/exit.c:1804		pid_t upid	int *stat_addr	int options	struct rusage *ru
62	0x3e	kill	__x64_sys_kill	kernel/signal.c:3796		pid_t pid	int sig		

ARM64 Apple

```
mov X16, #1 // номер функции для выхода из программы
SVC #0x80 // вызываем системную функцию (обработчик пребываний)
```

<https://opensource.apple.com/source/xnu/xnu-1504.3.12/bsd/kern/syscalls.master>

```
#include <sys/appleapiopts.h>
#include <sys/param.h>
#include <sys/system.h>
#include <sys/types.h>
#include <sys/sysent.h>
#include <sys/sysproto.h>

0      AUE_NULL      ALL    { int nosys(void); }   { indirect syscall }
1      AUE_EXIT      ALL    { void exit(int rval); }
2      AUE_FORK      ALL    { int fork(void); }
3      AUE_NULL      ALL    { user_size_t read(int fd, user_addr_t cbuf, user_size_t nbytes); }
4      AUE_NULL      ALL    { user_size_t write(int fd, user_addr_t cbuf, user_size_t nbytes); }
5      AUE_OPEN_RWTC  ALL    { int open(user_addr_t path, int flags, int mode); }
6      AUE_CLOSE     ALL    { int close(int fd); }
7      AUE_WAIT4     ALL    { int wait4(int pid, user_addr_t status, int options, user_addr_t rusage); }
8      AUE_NULL      ALL    { int nosys(void); }   { old creat }
9      AUE_LINK      ALL    { int link(user_addr_t path, user_addr_t link); }
10     AUE_UNLINK    ALL    { int unlink(user_addr_t path); }
```

ВЫВОД В КОНСОЛЬ

GAS

```
.globl _start  
  
.data  
message: .asciz "Hello World\n"  
  
.text  
  
_start:  
  
    movq $message, %rsi  
    movq $1, %rdi  
    movq $18, %rdx  

```

ARM64 Apple

```
.global _main // Устанавливаем точку входа в программу  
.align 2      // Для MacOS требуется выравнивание в 4 байта  
  
.text  
  
_main:  
  
    adr X1, message // передаем адрес вывода на консоль  
    mov X0, #1 // в поток вывода  
    mov X2, #18 // размер строки в байтах  

```

GAS

```
.global _start
.data
output_str: .asciz "Output value: %d\n"
.text
_start:
    leaq output_str(%rip), %rdi #загружаем адрес
строки output_str в rdi
    movq $66, %rsi   #заносим в esi 66
    xor %rax, %rax  #обнуляем через xor eax
    call printf      #вызов печати
    movq $60, %rax   # номер функции для выхода из
программы
    movq $0, %rdi    # возвращаемый результат
    syscall          # вызываем системную функцию
```

Сборка и запуск:

```
as asm.s -o asm.o && ld -dynamic-linker /
lib64/ld-linux-x86-64.so.2 -lc -o asm asm.o
&& ./asm
```

Вывод в консоль через C Library

ARM64 Apple

```
.global _main
.align 2
_main:
    mov x1, #66      //сохраняем число в регистр
    str x1, [sp]     //сохраняем (store) число в стековый регистр
    adr x0, output_str //загружаем адрес строки в x0
    bl _printf        //вызываем С-функцию вывода

    mov x0, #0        // код возврата
    mov X16, #1 // системный вызов 1 завершает программу
    svc #0x80 // вызываем системную функцию с номером 1

output_str:
    .asciz "Output value: %d\n"
```

Сборка и запуск:

```
as arm_example.s -o arm.o -arch arm64 && ld -o arm arm.o
-lSystem -syslibroot `xcrun -sdk macosx --show-sdk-path` 
_main -arch arm64 && ./arm
```

Вывод в консоль через C Library GAS

```
.global _start
.data
output_str: .asciz "Output value: %d\n"
.text
_start:
    pushq %rbp # сохраняем старое значение RBP в стек
    movq %rsp, %rbp # копируем текущий адрес из RSP в RBP

    leaq output_str(%rip), %rdi #загружаем адрес строки output_str в rdi
    movq $66, %rsi #заносим в esi 66
    xor %rax, %rax #обнуляем через xor eax
    call printf #вызов печати
    movq %rbp, %rsp # удаляем локальные переменные и очищаем стек
    popq %rbp # восстанавливаем в RBP значение для вызывающего кода

    movq $60, %rax # номер функции для выхода из программы
    movq $0, %rdi # возвращаемый результат
    syscall
```

Сборка и запуск:

```
as asm.s -o asm.o && ld -dynamic-linker /
lib64/ld-linux-x86-64.so.2 -lc -o asm asm.o
&& ./asm
```

Замечание!

На некоторых системах с CPU AMD Ryzen

необходимы принудительно выровнять регистры стека по границе 16 байт!

Это нужно сделать только один раз в коде.

Или хватит только этих строк

Это может приводить к «ошибке сегментирования» или «segmentation fault» при запуске

Вывод в консоль через C Library

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1 st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4 th integer argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2 nd argument to functions	No
%rdi	used to pass 1 st argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-%r15	callee-saved registers	Yes
%xmm0-%xmm1	used to pass and return floating point arguments	No
%xmm2-%xmm7	used to pass floating point arguments	No
%xmm8-%xmm15	temporary registers	No
%mmx0-%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2-%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

GAS

Используемые регистры
под аргументы

Основные операции (ARM64 Apple)

Операция сложения

```
ADD Xd, Xn, Xm      // Xd = Xn + Xm
ADD Xd, Xn, #imm    // Xd = Xn + #imm
ADD Xd, Xn, #imm, shift // сложение со сдвигом –
непосредственный operand #imm сдвигается с помощью выражения shift
ADD Xd, Xn, Xm, shift #N // сложение со сдвигом – регистр Xm
сдвигается с помощью выражения shift #N
ADD Xd, Xn, Xm, extend #N // сложение с расширением – регистр Xm
расширяется с помощью выражения extend #N
```

```
.global _main
.align 2
_main:
    mov X0, #0
    mov X1, #12      // X1 = 12
    add X0, X1, #22  // X0 = X1 + 22 = 12 + 22 = 34

    mov X1, #12      // X1 = 12
    mov X2, #10      // X2 = 10
    add X0, X1, X2   // X0 = X1 + X2 = 12 + 10 = 22

    mov X0, #0        // код возврата
    mov X16, #1       // номер функции для выхода из программы
    svc #0x80         // вызываем системную функцию
```

Основные операции (GAS)

Операция сложения

addq: для сложения 64-разрядных чисел
addl: для сложения 32-разрядных чисел
addw: для сложения 16-разрядных чисел
addb: для сложения 8-разрядных чисел

```
.globl _start
.text
_start:
    movq $13, %rcx  # RCX = 13
    movq $22, %rdi  # RDI = 22
    addq %rcx, %rdi # RDI = RCX + RDI = 13 + 22 = 35

    movq $60, %rax  # номер функции для выхода из программы
    movq $0, %rdi   # возвращаемый результат
    syscall         # вызываем системную функцию
```

Основные операции (ARM64 Apple)

Операция вычитания

```
SUB Xd, Xn, Xm      // Xd = Xn - Xm
SUB Xd, Xn, #imm    // Xd = Xn - #imm
SUB Xd, Xn, #imm, shift // вычитание со сдвигом –
непосредственный operand #imm сдвигается с помощью выражения
shift
SUB Xd, Xn, Xm, shift #N // вычитание со сдвигом – регистр Xm
сдвигается с помощью выражения shift #N
SUB Xd, Xn, Xm, extend #N // вычитание с расширением – регистр
Xm расширяется с помощью выражения extend #N
```

```
.global _main
.align 2
_main:
    mov X1, #22
    sub X0, X1, #2      //X0 = X1 - 2 = 22 - 2 = 20
    // X0 = X2 + младший байт из X1, сдвинутый влево на 4 бита = 0x2 + (0xFD
<< 4) =0x2 + 0xD0 = 0xD2
    mov X1, #20
    mov X2, #5
    sub X0, X1, X2      //X0 = X1 - X2 = 20 - 5 = 15
    mov X0, #0            //код возврата
    mov X16, #1           //номер функции для выхода из программы
    svc #0x80             //вызываем системную функцию
```

Основные операции (GAS)

Операция вычитания

subq: для вычитания 64-разрядных чисел
subl: для вычитания 32-разрядных чисел
subw: для вычитания 16-разрядных чисел
subb: для вычитания 8-разрядных чисел

```
.globl _start
.text
_start:
    movq $11, %rcx    # RCX = 11
    movq $55, %rdi    # RDI = 55
    subq %rcx, %rdi  # RDI = RDI - RCX= 55 - 11 = 44

    movq $60, %rax    # номер функции для выхода из программы
    movq $0, %rdi     # возвращаемый результат
    syscall           # вызываем системную функцию
```

Основные операции (GAS)

Операция инкремента / декремента

incq: для инкремента 64-разрядных чисел

incl: для инкремента 32-разрядных чисел

incw: для инкремента 16-разрядных чисел

incb: для инкремента 8-разрядных чисел

decq: для декремента 64-разрядных чисел

decl: для декремента 32-разрядных чисел

decw: для декремента 16-разрядных чисел

decb: для декремента 8-разрядных чисел

```
.globl _start
.text
_start:
    movq $4, %rdi
    incq %rdi      # RDI = RDI + 1 = 4 + 1 = 5
    decq %rdi      # RDI = RDI - 1 = 5 - 1 = 4

    movq $60, %rax # номер функции для выхода из программы
    movq $0, %rdi  # возвращаемый результат
    syscall         # вызываем системную функцию
```

Основные операции (ARM64 Apple)

Операция умножения

MUL Xd, Xn, Xm
MUL Wd, Wn, Wm

```
.global _main
.align 2
_main:
    mov X1, #22
    mov x0, #5
    mov x1, #2
    mul x0, x0, x1 // X0 = X0 * X1 = 5 * 2 = 10

    mov X0, #0
    mov X16, #1      // номер функции для выхода из программы
    svc #0x80        // вызываем системную функцию
```

Основные операции (GAS)

Операция умножения

mulq: для умножения беззнаковых 64-разрядных чисел – результат в RDX:RAX
mull: для умножения беззнаковых 32-разрядных чисел – результат в EDX:EAX
mulw: для умножения беззнаковых 16-разрядных чисел – результат в DX:AX
mulb: для умножения беззнаковых 8-разрядных чисел – результат в AX

imulq: для умножения знаковых 64-разрядных чисел – результат в RDX:RAX
imull: для умножения знаковых 32-разрядных чисел – результат в EDX:EAX
imulw: для умножения знаковых 16-разрядных чисел – результат в DX:AX
imulb: для умножения знаковых 8-разрядных чисел – результат в AX

```
.globl _start
.text
_start:
    movq $2, %rdi
    movq $4, %rax
    mulq %rdi          # RAX = RAX * RDI
    movq %rax, %rdi    # RDI = RAX = 8
    movq $0xC000000000000002, %rdi
    movq $4, %rax
    mulq %rdi          # RDX:RAX = RDI * RAX
    movq %rdx, %rdi    # помещаем в RDI старшие 64 бита результата – RDI =
RDX = 3

    movq $60, %rax    # номер функции для выхода из программы
    movq $0, %rdi     # возвращаемый результат
    syscall           # вызываем системную функцию
```

Основные операции (ARM64 Apple)

Операция деления

SDIV Xd, Xn, Xm //деление знаковое
UDIV Xd, Xn, Xm //деление беззнаковое

```
.global _main
.align 2
_main:
    mov X2, #45
    mov X3, #5
    sdiv X0, X2, X3 // X0=X2/X3 = 45/5 = 9

    mov X2, #49
    mov X3, #5
    udiv X0, X2, X3 // X0=X2/X3 = 49/5 = 9
    //остаток высчитывать нужно самим, ARM64 его не высчитывает
    mul X3, X3, X0 // X3=X3 * X0 = 5 * 9 = 45
    sub X3, X2, X3 // X3=X2 - X3 = 49 - 45 = 4 - остаток

    mov X0, #0.          // код выхода из программы
    mov X16, #1           // номер функции для выхода из программы
    svc #0x80             // вызываем системную функцию
```

Основные операции (GAS)

Операция деления

```
.globl _start
.text
_start:
    movq $22, %rax
    movq $0, %rdx    # расширяем RDX нулем для корректности деления
    movq $5, %rcx
    divq %rcx        # RAX = RAX / RCX
    # RAX =4 (результат), RDX = 2 (остаток)
    movq %rax, %rdi   # RDI = RAX = 4
    movq %rdx, %rdi   # RDI = RDX = 2

    movq $-5, %rcx
    movq $-22, %rax
    cqo      # расширяем регистр RDX знаковым битом из RAX
    idivq %rcx        # RAX = RAX / RCX
    # RAX =4 (результат), RDX = -2 (остаток)
    movq %rax, %rdi   # RDI = RAX = 4

    movq $60, %rax  # номер функции для выхода из программы
    movq $0, %rdi   # возвращаемый результат
    syscall         # вызываем системную функцию
```

divq: для деления беззнаковых 64-разрядных чисел
divl: для деления беззнаковых 32-разрядных чисел
divw: для деления беззнаковых 16-разрядных чисел
divb: для деления беззнаковых 8-разрядных чисел
idivq: для деления знаковых 64-разрядных чисел
idivl: для деления знаковых 32-разрядных чисел
idivw: для деления знаковых 16-разрядных чисел
idivb: для деления знаковых 8-разрядных чисел

Расширение для знаковых

cbw: преобразует байт в AL в слово в AX через расширение знаком
cwd: преобразует 16-разрядное число в AX в 32-разрядное в DX:AX через расширение знаком
cdq: преобразует 32-разрядное число в EAX в 64-разрядное в EDX:EAX с помощью расширения знаком
cqo: преобразует 64-разрядное число в RAX в 128-разрядное в RDX:RAX через расширение знаком
cwd: преобразует 16-разрядное число в AX в 32-разрядное в EAX с помощью расширения знаком
cdqe: преобразует 32-разрядное число в EAX в 64-разрядное в RAX с помощью расширения знаком

Пример (ARM64 Apple)

$$Y = (a + b) / c^2$$

Беззнаковые числа

```
.global _main          //Устанавливаем точку входа в программу
.align 2              //Для MacOS требуется выравнивание в 4 байта

_main:
    mov x0, #150      //ввод A
    mov x1, #130      //ввод B

    add x1, x0,x1    //A + B
    mov x0, #10        //ввод C
    mul x0, x0, x0    //C^2
    sdiv x0, x1, x0   //y = (A + B)/C^2

//код выхода из программы в x0
    mov X16, #1        //системный вызов 1 завершает программу
    svc #0x80          //вызываем системную функцию с номером 1
```

Пример (ARM64 Apple)

$$Y = (a + b) / c^2$$

Беззнаковые числа с выводом в консоль

```
.global _main
.align 2
_main:

    mov x0, #150          //ввод A
    mov x1, #130          //ввод B

    add x1, x0,x1        //A + B
    mov x0, #10            //ввод C
    mul x0, x0, x0        //C^2
    sdiv x0, x1, x0       //y = (A + B)/C^2

    str x0, [sp]           //сохраняем (store) число в стековый регистр
    adr x0, output_str    //загружаем адрес строки в x0
    bl _printf             //вызываем С-функцию вывода

    mov x0, #0
    mov X16, #1            //системный вызов 1 завершает программу
    svc #0x80              //вызываем системную функцию с номером 1

output_str:
    .asciz "Output value: %d\n"
```

Пример (ARM64 Apple)

$$Y = (a + b) / c^2$$

Беззнаковые числа с выводом в консоль с остатком

```
.global _main
.align 2
_main:

    mov x0, #150          //ввод А
    mov x1, #130          //ввод В

    add x1, x0,x1        //A + B
    mov x0, #10            //ввод С
    mul x0, x0, x0        //C^2
    sdiv x2, x1, x0        //y = (A + B)/C^2
    mul x3, x0, x2        //C^2 * X2
    sub x3, x1, x3        //A+B - X3

    str x2, [sp]           //сохраняем (store) число в стековый регистр
    str x3, [sp, #8]        //сохраняем (store) число в стековый регистр + 8 байт смещение
    adr x0, output_str     //загружаем адрес строки в x0
    bl _printf              //вызываем С-функцию вывода

    mov x0, #0
    mov X16, #1             //системный вызов 1 завершает программу
    svc #0x80                //вызываем системную функцию с номером 1

output_str:
    .asciz "Output value: %d, %d\n"
```

Пример (ARM64 Apple)

$$Y = (a + b) / c^2$$

```
.global _main
.align 2
_main:

    mov x0, #150          // ввод A
    mov x1, #130          // ввод B
    add x1, x0,x1         // A + B
    mov x0, #10            // ввод C
    mul x0, x0, x0         //C^2
    sdiv x2, x1, x0        //y = (A + B)/C^2
    mul x3, x0, x2         //C^2 * X2
    sub x3, x1, x3         //A+B - X3

    stp x29, x30, [sp, #-16] //сохраняем 2 регистра связанных со стеком
    sub sp, sp, #32          //выделили 32 байта в стеке

    str x2, [sp]             //сохраняем (store) число в стековый регистр
    str x3, [sp, #8]          //сохраняем (store) число в стековый регистр + 8 байт смещение
    adr x0, output_str        //загружаем адрес строки в x0
    bl _printf                //вызываем С-функцию вывода

    add sp, sp, #32          //очищаем выделенные 32 байта со стека
    ldp x29, x30, [sp], #16 //восстанавливаем значение регистров

    mov x0, #0
    mov x16, #1              //системный вызов 1 завершает программу
    svc #0x80                //вызываем системную функцию с номером 1

output_str:
    .asciz "Output value: %d, %d\n"
```

Беззнаковые числа с выводом в консоль
с остатком и восстановлением стека

Пример (GAS)

$$Y = (a + b) / c^2$$

Беззнаковые числа

```
globl _start

.text
_start:
    movq $150, %rdi #ввод A
    movq $130, %rbx #ввод B
    addq %rdi, %rbx #B = A+B
    movq $10, %rax #ввод C
    mulq %rax        #c^2
    movq %rax, %rdi # c^2 в rdi
    movq %rbx, %rax #A+B в rax
    movq $0, %rdx   #расширение для беззнаковых
    divq %rdi       #y = (A+B)/c^2
    movq %rax, %rdi #целую часть на вывод с ret кодом
    movq $60, %rax # номер функции для выхода
    syscall         # вызываем системную функцию
```

Пример (GAS)

$$Y = (a + b) / c^2$$

```
.global _start
.data
output_str: .asciz "Output value: %d\n"
.text
_start:
    movq $150, %rdi #ввод А
    movq $130, %rbx #ввод В
    addq %rdi, %rbx #B = A+B
    movq $10, %rax #ввод С
    mulq %rax      #c^2
    movq %rax, %rdi #c^2 в rdi
    movq %rbx, %rax #A+B в rax
    movq $0, %rdx  #расширение для беззнаковых
    divq %rdi      #y = (A+B)/c^2

    leaq output_str(%rip), %rdi #загружаем адрес строки output_str в rdi
    movq %rax, %rsi # заносим Y в RSI
    xor %eax, %eax #обнуляем через xor rax, в rax возврат от функции printf
    call printf     #вызов печати

    movq $0, %rdi  # возвращаемый результат
    movq $60, %rax # номер функции для выхода
    syscall        # вызываем системную функцию
```

Беззнаковые числа с выводом на консоль

Пример (GAS)

$$Y = (a + b) / c^2$$

```
.global _start
.data
output_str: .asciz "Output value: %d, %d\n"
.text
_start:
    movq $150, %rdi #ввод A
    movq $130, %rbx #ввод B
    addq %rdi, %rbx #B = A+B
    movq $10, %rax #ввод C
    mulq %rax        #c^2
    movq %rax, %rdi #c^2 в rdi
    movq %rbx, %rax #A+B в rax
    movq $0, %rdx   #расширение для беззнаковых
    divq %rdi       #y = (A+B)/c^2, в rdx – остаток

    leaq output_str(%rip), %rdi #загружаем адрес строки output_str в rdi
    movq %rax, %rsi # заносим Y в RSI
    xor %eax, %eax #обнуляем через xor rax, в rax возврат от функции printf
    call printf      #вызов печати

    movq $0, %rdi   # возвращаемый результат
    movq $60, %rax  # номер функции для выхода
    syscall         # вызываем системную функцию
```

Беззнаковые числа с выводом в консоль
с остатком

Программирование на языках Ассемблера

Лекция 3

Синёв Николай Иванович
Кафедра 14



Работа с типами данных

GAS

.ascii: строка в двойных кавычках
.asciz: строка ascii, которая заканчивается 0вым байтом
.byte: целое число размером в 1 байт
.short (.word): целое число размером в 2 байта (слово)
.long: целое число размером в 4 байта (так называемое двойное слово)
.quad: целое число размером в 8 байт (четверное слово)
.octa: целое число размером в 16 байт (восьмеричное слово)
.float: число с плавающей точкой одинарной точности
.double: число с плавающей точкой двойной точности

ARM64 Apple

.ascii: строка в двойных кавычках
.asciz: строка ascii, которая заканчивается 0–вым байтом
.byte: целое число размером в 1 байт
.short: целое число размером в 2 байта (так называемое полуслово – hlf-word)
.word: целое число размером в 4 байта (слово)
.quad: целое число размером в 8 байт (двойное слово)
.octa: целое число размером в 16 байт (четверное слово)
.float: число с плавающей точкой одинарной точности
.double: число с плавающей точкой двойной точности

Работа с типами данных

ARM64 Apple

```
.global _start
.align 2
.data
    m_byte: .byte 1          // определяем один
байт, который равен 1
    m_word: .word 18         // определяем
одно слово (4 байта), которое равно 18
    m_short: .short 3        // определяем
двуихбайтное число, которое равно 3
    m_quad: .quad 1248       // определяем
двойное слово (8 байт), которое равно 1248
    m_octa: .octa 12         // определяем
четверное слово (16 байт), которое равно 12
    message1: .ascii "Test1\n" // определяем
строку
    message2: .asciz "Test2"   // определяем
строку, которая заканчивается нулевым байтом
.text
_start:
    mov x0, #0
    mov X16, #1      //системный вызов 1
завершает программу
    svc #0x80        //вызываем системную
функцию с номером 1
```

GAS

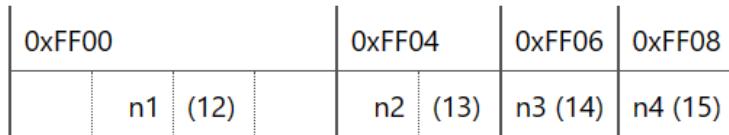
```
.global _start
.data
    m_byte: .byte 1
    m_word: .word 18 # 2 байта
    m_short: .short 3 # 2 байта
    m_quad: .quad 1248
    m_octa: .octa 12
    message1: .ascii «Test1\n»
    message2: .asciz «Test2»

.text
_start:
    movq $0, %rdi    # возвращаемый результат
    movq $60, %rax    # номер функции для выхода
    syscall           # вызываем системную функцию
```

Расположение данных в памяти

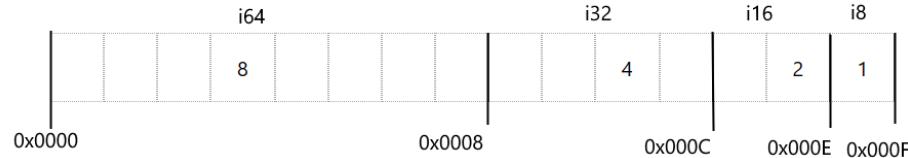
ARM64 Apple

```
.global _start
.align 2
.data
    n1: .word 12      // 4 байта
    n2: .short 13     // 2 байта
    n3: .byte 14      // 1 байт
    n4: .byte 15      // 1 байт
.text
_start:
    mov x0, #0
    mov X16, #1      //системный вызов 1
завершает программу
    svc #0x80        //вызываем системную
функцию с номером 1
```



GAS

```
.global _start
.data
    i64: .quad 8 # 8 байт
    i32: .long 4 # 4 байта
    i16: .word 2 # 2 байта
    i8: .byte 1. # 1 байт
.text
_start:
    movq $0, %rdi    # возвращаемый результат
    movq $60, %rax   # номер функции для выхода
    syscall           # вызываем системную функцию
```



Массивы данных

ARM64 Apple

```
.global _start
.align 2
.data
    bytes: .byte 74, 0112, 0b00101010, 0x4A, 0X4a
.text
_start:
    mov x0, #0
    mov X16, #1      //системный вызов 1
завершает программу
    svc #0x80        //вызываем системную
функцию с номером 1
```

GAS

```
.global _start
.data
    bytes: .byte 74, 0112, 0b00101010, 0x4A, 0X4a
.text
_start:
    movq $0, %rdi    # возвращаемый результат
    movq $60, %rax   # номер функции для выхода
    syscall          # вызываем системную функцию
```

Работа с массивом данных

ARM64 Apple

```
.global _start
.align 2
.data
    bytes: .byte 74, 0112, 0b01001010, 0x4A, 0X4a
.text
_start:
    adrp x1, bytes@PAGE      //загружает
выровненный адрес страницы (4 КБ) относительно
текущего счетчика программ.
    add x1,x1,bytes@PAGEOFF //используется для
добавления младших 12 бит – смещения bytes к
этому адресу страницы, что дает правильный адрес

    ldr x2, [X1] // загружен элемент с
индексом 0
    // загрузим элемент с индексом 3
    // каждый элемент размером в 1 байт
    ldr x2, [x1, #(2 * 1)] //загружаем само
значение по адресу массива + смещение
    add x2, x2, 2 // прибавляем к значению + 2, в
x2 = 76
    ldr x0, [x1]    //в x0 = 74
    mov x0, x2      //x0 = 76
    mov X16, #1     //системный вызов 1
завершает программу
    svc #0x80      //вызываем системную
функцию с номером 1
```

GAS

```
.global _start
.data
    bytes: .byte 74, 0112, 0b01001010, 0x4A, 0X4a
.text
_start:
    movq $bytes, %rbx # помещаем в RBX адрес
переменной bytes
#leaq bytes(%rip), %rbx #загружаем адрес строки
массива в rbx
    addq $2, %rbx      # прибавляем к адресу в RBX 2
байта
    movq (%rbx), %rdi # значение rbx в rdi
    movq $60, %rax # номер функции для выхода
    syscall           # вызываем системную функцию
```

Определение набора данных

ARM64 Apple

```
.global _start
.align 2
.data
    nums: .fill 10, 8, 5

    nums2: .rept 3 //повторяем 3 раза
        .quad 0, 1, 2
    .endr

.text
_start:
    adrp x1, bytes@PAGE
    add x1,x1,bytes@PAGEOFF
    ldr x0, [x1]
    mov X16, #1 //системный вызов 1
    завершает программу
    svc #0x80 //вызываем системную
    функцию с номером 1

.bss
buffer: .zero 1000. //содержит
неинициализированные данные, для которых известен
размер
```

GAS

```
.global _start
.data
    nums: .fill 10, 8, 5

    nums2: .rept 3 # повторяем 3 раза
        .quad 0, 1, 2
    .endr

        buffer1: .skip 1024, 22 # каждый из 1024
байтов равен 22
        buffer2: .space 512 # каждый из 512
байтов равен 0
.text
_start:
    movq ($nums), %rdi # возвращаемый результат
    movq $60, %rax # номер функции для выхода
    syscall # вызываем системную функцию

.bss
buffer: .zero 1000

.section .rodata # по сути – константные данные
const: .quad 6
```

Константы

ARM64 Apple

```
.global _start
.align 2
.equ var1, 1
.equ var2, 2
.equ var3, 3

.equ first, 0
.equ second, first + 8
.equ third, second + 8

.equ intSize, 4

.data
num1: .quad intSize
num2: .quad intSize + 4
message: .ascii "Hello!\n"
.equ count, . - message // длина строки

.text
_start:
    mov x0, var1 // X0 = 1
    mov x0, var2 // X0 = X0 + 2 = 3
    mov x0, var3 // X0 = X0 + 3 = 6

    mov x0, third
    mov x16, #1 //системный вызов 1
завершает программу
    svc #0x80 //вызываем системную
функцию с номером 1
```

GAS

```
.global _start
.equ var1, 1
.equ var2, 2
.equ var3, 3

.equ first, 0
.equ second, first + 8
.equ third, second + 8

.equ intSize, 4

.data
num1: .quad intSize
num2: .quad intSize + 4

message: .ascii "Hello!\n"
count = . - message # длина сообщения

.text
_start:
    movq $var1, %rdi # RDI = 1
    addq $var2, %rdi # RDI = RDI + 2 = 3
    addq $var3, %rdi # RDI = RDI + 3 = 6

    movq $third, %rdi # RDI = 16

    movq $60, %rax # номер функции для выхода
    syscall # вызываем системную функцию
```

Выравнивание

ARM64 Apple

```
.global _start
.align 2
.data
//плохо
num1: .byte 11
num2: .byte 12
num3: .quad 13
num4: .word 14

//лучше, но не оптимально
num1: .byte 11
.align 2
num2: .byte 12
.align 2
num3: .quad 13
num4: .word 14

//оптимально
num3: .quad 13
num4: .word 14
num1: .byte 11
.align 2
num2: .byte 12
.text
_start:
    mov x0, third
    mov X16, #1      //системный вызов 1
завершает программу
    svc #0x80        //вызываем системную
функцию с номером 1
```

GAS

```
.global _start
.data
#плохо
num1: .byte 11
num2: .byte 12
num3: .quad 13
num4: .word 14

#лучше, но не оптимально
num1: .byte 11
.p2align 2      #balign, align – зависит от
реализации
num2: .byte 12
.p2align 2
num3: .quad 13
num4: .word 14

#оптимально
num3: .quad 13
num4: .word 14
num1: .byte 11
.p2align 2
num2: .byte 12

.text
_start:

    movq $0, %rdi    # RDI = 16
    movq $60, %rax   # номер функции для выхода
    syscall          # вызываем системную функцию
```

Переходы безусловные

ARM64 Apple

```
.global _start
.align 2
_start:
    mov x0, 11
    b exit
    mov x0, 22      // не выполняется
exit:
    mov x0, #0
    mov X16, #1     //системный вызов 1
завершает программу
    svc #0x80       //вызываем системную
функцию с номером 1

.global _start
.align 2
_start:
    mov x0, 11
    adr x1, exit    // загружаем в X1 адрес метки
exit:
    b x1           // переход по адресу в
регистре X1
    br x1           // переход по адресу в
регистре X1
exit:
    mov x0, #0
    mov X16, #1     //системный вызов 1
завершает программу
    svc #0x80       //вызываем системную
функцию с номером 1
```

GAS

```
.globl _start
.text
_start:
    movq $11, %rdi      # RDI = 11
    jmp exit             # переходим к метке exit
    movq $22, %rdi      # не выполняется
exit:                           # метка exit
    movq $60, %rax      # номер функции для выхода
    syscall              # вызываем системную функцию

.globl _start
.data
pointer: .quad exit        # переменная pointer
хранит адрес метки exit
.text
_start:
    movq $11, %rdi
    movq $exit, %rbx     # в регистре RBX адрес
метки exit
    jmp *%rbx            # переход по адресу, который
хранится в регистре RBX
    jmp *pointer          # переход по адресу, который
хранится в переменной pointer
exit:                           # метка exit
    movq $60, %rax      # номер функции для выхода
    syscall              # вызываем системную функцию
```

Переходы условные Флаги

ARM64 Apple

- **N (Negative)**: флаг знака. Равен 1, если результат операции представляет отрицательное число. Равен 0, если результат - положительное число или 0.
- **Z (Zero)**: флаг нуля. Равен 1, если результат операции равен нулю. Равен 0, если результат операции НЕ равен нулю.
- **C (Carry)**: флаг переноса. Установка этого флага зависит от контекста.
Обычно он устанавливается, если при выполнении арифметической операции произошел перенос. Подобное состояние еще называют беззнаковым переполнением (unsigned integer overflow)
- **V (oVeflow)**: флаг переполнения, устанавливается, если при выполнении арифметической операции произошло переполнение со знаком (signed integer overflow). При сложении этот флаг устанавливается в следующих случаях:
 - Если складываются два положительных числа, а результат отрицательный
 - Если складываются два отрицательных числа, а результат положительный
- При вычитании флаг устанавливается в следующих случаях:
 - Если из положительного числа вычитается отрицательное, а результат отрицательный
 - Если из отрицательного числа вычитается положительное, а результат положительный

GAS

- **CF (флаг переноса)**: устанавливается, если происходит беззнаковое переполнение, то есть при сумме с переносом или вычитании с заимствованием. Если переполнение не происходит, то флаг не устанавливается.
- **OF (флаг переполнения)**: устанавливается, если происходит переполнение со знаком, а именно когда переполняется бит, следующий за старшим знаковым битом.
- **SF (флаг знака)**: устанавливается, если старший бит результата установлен. В противном случае флаг знака сброшен (то есть флаг знака отражает состояние старшего бита результата).
- **ZF (флаг нуля)**: устанавливается, если результат вычисления дает 0. Если результат ненулевой, флаг сброшен

Переходы условные Команды

ARM64 Apple

- **ADDS** (сложение с установкой флага переноса при переполнении результата)
- **ADCS** (сложение со флагом переноса с установкой знака переноса при переполнении результата)
- **SUBS** (вычитание с установкой флага переноса при переполнении результата)
- **SBCS** (вычитание бита переноса с установкой флага переноса при переполнении результата)
- **NEGS** (умножает число на -1 и устанавливает флаги)
- **NGCS** (использует флаг переноса для получения отрицательного значения и устанавливает флаги)
- **ANDS**: аналог инструкции AND с добавлением установки флагов
- **BICS**: аналог инструкции BIC с установкой флагов
- **CMP A, B**: сравнивает A в отношении B. Фактически эквивалентна инструкции SUBS XZR, A, B
- **CMN A, B**: сравнивает A в отношении -B. Фактически эквивалентна инструкции ADDS XZR, A, B
- **TST A, B**: проверяет, установлены или биты B внутри A. Эквивалентна инструкции ANDS XZR, A, B

GAS

Флаги устанавливаются в результате различных операций. В частности, инструкции **add**, **sub**, **and**, **or**, **xor** и **not** влияют на установку флагов. А **mov** или **lea** не влияют.

Переходы условные Команды

ARM64 Apple

GAS

B.{condition} label , где условие:

EQ, Z == 1, Равенство значений

NE, Z == 0, Неравенство значений

CS или **HS**, C == 1 больше или равно >= (без знака)

CC или **LO**, C == 0. меньше < (без знака)

MI, N == 1, Значение отрицательное

PL, N == 0, Значение положительное или равно 0

VS, V == 1, Есть переполнение знака

VC, V == 0, Нет переполнения знака

HI, C == 1 && Z == 0 (C-флаг установлен, а Z-флаг сброшен)

больше > (без знака)

LS, !(C == 1 && Z == 0) (C-флаг сброшен, а Z-флаг установлен или сброшен), меньше или равно <= (без знака)

GE, N == V (N-флаг и V-флаг совпадают), больше или равно >= (со знаком)

LT, N != V (N-флаг и V-флаг различаются) >меньше < (со знаком)

GT, Z == 0 && N == V (Z-флаг сброшен, N-флаг и V-флаг совпадают), больше > (со знаком)

LE, !(Z == 0 && N == V) (Z-флаг установлен, N-флаг и V-флаг различаются), меньше или равно <= (со знаком)

AL, любое значение. Всегда истинно

- **jc**: выполняет переход к метке, если флаг переноса установлен
- **jnc**: выполняет переход к метке, если флаг переноса НЕ установлен
- **jo**: выполняет переход к метке, если флаг переполнения установлен
- **jno**: выполняет переход к метке, если флаг переполнения не установлен
- **js**: выполняет переход к метке, если флаг знака установлен
- **jns**: выполняет переход к метке, если флаг знака не установлен
- **jz**: выполняет переход к метке, если флаг нуля установлен
- **jnz**: выполняет переход к метке, если флаг нуля не установлен
- **clc**: сбрасывает в 0 флаг переноса (CF)
- **setc**: устанавливает флаг переноса (CF)
- **lahf**: копирует флаги состояния из регистра %eflags в регистр %ah
- **sahf**: сохраняет флаги состояния из регистра %ah в регистр %eflags

Переходы условные Команды

ARM64 Apple

EQ (равно) - **NE** (не равно)

HS/CS (Флаг переноса установлен) - **LO/CC** (Флаг переноса сброшен)

MI (Результат отрицательный) - **PL** (Результат положительный или равен 0)

VS (Переполнение знака) - **VC** (Нет переполнения знака)

HI (беззнаковое больше) - **LS** (Беззнаковое меньше или равно)

GE (Больше чем или равно со знаком) - **LT** (Меньше чем со знаком)

GT (Больше чем со знаком) - **LE** (Меньше чем или равно со знаком)

Переходы условные Команды. Специальные инструкции перехода

ARM64 Apple

B{condition} label, когда перед переходом выполнена команда **cmp**

BEQ label: переход, если Z = 1 (значения равны)

BNE label: переход, если Z = 0 (значения не равны)

BCS/BHS label: переход, если C = 1 (первое значение больше или равно, беззнаковое сравнение)

BCC/BLO label: переход, если C = 0 (первое значение меньше, беззнаковое сравнение)

BMI label: переход, если N = 1

BPL label: переход, если N = 0

BVS label: переход, если V = 1

BVC label: переход, если V = 0

BHI label: переход, если C = 1 и Z = 0 (первое значение больше, беззнаковое сравнение)

BLS label: переход, если C = 0 и Z = 1 (первое значение меньше или равно, беззнаковое сравнение)

BGE label: переход, если N = V (первое значение больше или равно, сравнение со знаком)

BLT label: переход, если N != V (первое значение меньше, сравнение со знаком)

BGT label: переход, если Z = 0 и N = V (первое значение больше, сравнение со знаком)

BLE label: переход, если или Z = 1, или N = !V (первое значение меньше или равно, сравнение со знаком)

Переходы условные

ARM64 Apple

GAS

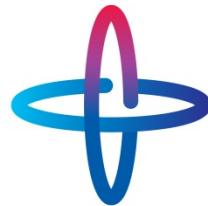
```
.global _start
.align 2
_start:
    mov x0, #-1
    mov x1, #3
    adds x0, x0, x1      // x0 = -1 + 3 = 2

    // проверяем условие CS – установлен флаг
переноса С
    b.cs carry_set // если флаг установлен – переход
к метке carry
    mov x0, 1      // если флаг не установлен, X0=1
    b exit
carry_set:
    mov x0, 2      // если флаг установлен, X0=2
exit:
    mov X16, #1      //системный вызов 1 завершает
программу
    svc #0x80      //вызываем системную функцию с
номером 1

.global _start
.align 2
_start:
    mov x0, #1
    mov x1, #0
    cmp x1, x0      // сравним X0 и X1
    bcs carry_set    // если произошел перенос
    mov x0, #255     // если не равны, X0=255
    b exit
carry_set:
    mov x0, #1      // если перенос
exit:
    mov X16, #1      //системный вызов 1
завершает программу
    svc #0x80      //вызываем системную
функцию с номером 1
```

```
.globl _start
.text
_start:
    movq $0xfffffffffffffff, %rcx
    movq $3, %rdx
    addq %rcx, %rdx      # RDX = RDX + RCX
    jc carry_set        # если флаг переноса
установлен, переход к метке carry_set
    movq $0, %rdi        # если флаг переноса не
установлен, RDI = 0
    jmp exit
carry_set:                  # если флаг переноса
установлен
    movq $2, %rdi        # RDI = 2
exit:                      # метка exit
    movq $60, %rax      # номер функции для выхода
    syscall             # вызываем системную функцию

.globl _start
.text
_start:
    movq $1, %rbx
    movq $0, %rax
    cmpq %rbx, %rax      # сравниваем RAX и RBX.
Фактически вычитаем RAX – RBX
    jc carry_set        # если произошел перенос
    movq $2, %rdi        # если нет переноса
    jmp exit
carry_set:                  # если есть перенос
exit:
    movq $60, %rax      # номер функции для выхода
    syscall             # вызываем системную функцию
```



гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru

Программирование на языках Ассемблера

Лекция 4

Синёв Николай Иванович
Кафедра 14



Загрузка/выгрузка значений по адресу

ARM64 Apple

```
.global _main
.align 2
.data
    num1: .quad 15
.text
_main:
    adrp X0, num1@PAGE           //загружает
выровненный адрес страницы (4 КБ) относительно
текущего счетчика программы.
    add X0, X0, num1@PAGEOFF    //используется
для добавления младших 12 бит – num1 bytes к
этому адресу страницы, что дает правильный адрес

    ldr x1, [x0] //в x1 загружаем адрес значение
по адресу x0
    mov x1, #20 //в x1 заносим новое число
    str x1, [x0] //Сохраняем значение из x1 по
адресу x0

    mov X16, #1
    svc #0x80
```

GAS

```
.global _start
.data
num1: .quad 15
.text
_start:
    movq $num1, %rbx # $ перед переменной
загружает адрес в регистр (например в адрес
0x40300)
    movq (%rbx), %rax # (регистр) – берет
значение по адресу в скобках (по адресу
0x40300 получает значение 15)

    movq $20, (%rbx) #по адресу переменной
num1 сохраняем значение 20

    leaq num1(%rip), %rax # в rax загружается
адрес num1 = 0x40300
    movq (%rax), %rbx #по адресу 0x40300
получаем значение 20

    movq %rbx, %rdi
    movq $60, %rax
    syscall
```

Косвенная адресация (1/2) GAS

```
.globl _start

.data
num1: .quad 9
num2: .quad 10
nums: .quad 11, 12, 13, 14, 15, 16
count: .quad .- nums

.text
_start:
    movq $nums, %rbx          # помещаем в RBX адрес переменной nums
    addq $8, %rbx             # прибавляем к адресу в RBX 8 байт
    movq (%rbx), %rdi         # помещаем в RDI значение по адресу из RBX (RDI = 12)

    movq $nums, %rbx          # помещаем в RBX адрес переменной nums
    addq $16, %rbx            # прибавляем к адресу в RBX 16 байт – $2 * $8 = $16
    movq (%rbx), %rdi         # помещаем в RDI значение по адресу из RBX (RDI = 13)

    movq $nums, %rbx          # помещаем в RBX адрес переменной nums
    subq $16, %rbx            # вычитаем от адресу в RBX 16 байт
    movq (%rbx), %rdi         # помещаем в RDI значение по адресу из RBX (RDI = 9)

#адресация со смещением
    movq $nums, %rbx          # помещаем в RBX адрес массива nums
    movq 8(%rbx), %rdi        # RDI = значение по адресу (RBX+8) = 12

#индексный регистр
    movq $nums, %rbx          # помещаем в RBX адрес переменной nums
    movq $24, %rsi             # RSI – индексный регистр
    movq (%rbx, %rsi), %rax   # в AL число по адресу (RBX + RSI) = 14
    movq %rax, %rdi            # RDI = 14
```

Косвенная адресация (2/2) GAS

```
#индексный регистр и множитель
movq $nums, %rbx          # помещаем в RBX адрес переменной nums
movq $2, %rsi             # RSI – индексный регистр
movq (%rbx, %rsi, 8), %rdi # в RDI число по адресу (RBX + RSI * 8) = 13
movq %rax, %rdi           # RDI = 13

#имя переменной как смещение
movq $2, %rsi             # RSI – индексный регистр
movq nums(, %rsi, 8), %rdi # в RDI число по адресу ($nums + RSI * 8)
movq %rax, %rdi           # RDI = 13

movq $60, %rax
syscall
```

Косвенная адресация Apple ARM 64 (1/2)

```
.global _main
.align 2
.data
    num1: .quad 9
    num2: .quad 10
    nums: .quad 11, 12, 13, 14, 15, 16
    count: .quad .- nums
.text
_main:
    adrp    X1, num1@PAGE
    add     X1, X1, num1@PAGEOFF

    ldr x2, [x1]
    mov x0, x2

    //Адресация со смещением по тексту кода
    ldr x2, [x1, 16]      //в x2 загружаем значение по адресу x1 + смещение равно 16 байтам
                           //в x2 будет первый элемент nums = 11
    mov x0, x2            //x0 = 11

    //адрес массива
    adrp    X1, nums@PAGE
    add     X1, X1, nums@PAGEOFF

    //Адресация со смещением
    ldr x2, [x1, 16]      //в x2 загружаем значение по адресу x1=nums + смещение равно 16
                           //байтам
                           //в x2 будет первый элемент nums = 13
    mov x0, x2            //x0 = 13
```

Косвенная адресация Apple ARM 64 (2/2)

//Адресация со смещением отрицательным

```
ldr x2, [x1, -16]      //в x2 загружаем адрес значение по адресу x1=nums – смещение равно 16
байтам
```

```
        //в x2 будет первый элемент num1 = 9
mov x0, x2              //x0 = 9
```

//Адресация со смещением через регистр

```
mov x3, #16
ldr x2, [x1, x3]        //в x2 загружаем адрес значение по адресу x1=nums + x3
                        //в x2 будет первый элемент nums = 13
mov x0, x2              //x0 = 13
```

//Адресация со смещением через множитель

```
ldr x2, [x1, 3*8]       //смещение 3 * 8 = 24 байтам, x2 = 14
mov x0, x2              //x0 = 14
```

//Регистр–смещение со сдвигом

```
mov x3, #3
ldr x2, [x1, x3, lsl 3] // эквивалентно ldr x2, [x1, 3*8]
mov x0, x2              //x0 = 14
```

//Преиндексная адресация

```
ldr x2, [x1, #8]!        //сначала X1=X1 + 8, после чего делается ldr x2, [x1]
mov x0, x2              //x0 = 12
ldr x2, [x1, #-8]!       //сначала X1=X1 – 8, после чего делается ldr x2, [x1]
                        //x2 = 11
```

//Постиндексная адресация

```
ldr x2, [x1], #8         //сначала ldr x2, [x1], а потом X1=X1 + 8
                        //x2 = 11
mov x0, x2              //x0 = 11

mov x16, #1
svc #0x80
```

Циклы GAS (1/2)

Сумма элементов

```
.globl _start

.data
nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
count: .quad (. - nums)/8 # количество элементов в массиве nums

.text
_start:
    movq $nums, %rbx      # помещаем в RBX адрес массива nums
    movq count, %rcx       # помещаем в RCX количество элементов в массиве nums
    movq $0, %rdi          # будущая сумма элементов массива
main_loop:
    addq (%rbx), %rdi      # складываем значение по адресу из RBX с числом в RDI
    addq $8, %rbx           # перемещаемся к следующему элементу массива

#1 способ зацикливания через 2 инструкции
subq $1, %rcx            # отнимаем 1 от значения в RCX
jnz main_loop             # если счетчик RCX не равен 0, то переходим обратно к main_loop

#2 способ зацикливания
# loopq main_loop         # альтернативный способ вместо предыдущих 2 инструкций

    movq $60, %rax
    syscall
```

Циклы GAS (2/2)

Сумма элементов

```
.globl _start

.data
nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
count: .quad (.−nums)/8           # количество элементов в массиве nums

.text
_start:
    movq count, %rcx             # помещаем в RCX количество элементов в массиве nums
    movq $0, %rdi                # будущая сумма элементов массива
    movq $0, %rsi                # индексный регистр
main_loop:
    addq nums(%rsi, 8), %rdi      # складываем значение по адресу из (nums+rsi*8) с числом в RDI
    incq %rsi                    # прибавляем 1 – получаем индекс следующего элемента
    loopq main_loop              # если счетчик RCX не равен 0, то переходим обратно к main_loop

    movq $60, %rax
    syscall
```

Циклы Apple ARM 64 (1/6)

Реализация while

```
.global _main
.align 2

.text
_main:

    mov x0, #0           // помещаем в регистр X0 число 0
    while:
        cmp x0, #5         // сравнивание значение регистра X0 с числом 5
        b.ge end_while      // если X0 равно или больше 5, то выходим из цикла – переход к метке
    end_while
        add x0, x0, #1       // X0 = X0 + 1
        b while              // переход обратно к метке while

end_while:          // завершение цикла

    mov x0, #0
    mov X16, #1
    svc #0x80
```

Циклы Apple ARM 64 (2/6)

Реализация do..while

```
.global _main
.align 2

.text
_main:

mov x0, #0          // помещаем в регистр X0 число 0

do_while:
    add x0, x0, #1    // X0 = X0 + 1 – выполняемые действия цикла
    cmp x0, #5        // сравнение значение регистра X0 с числом 5
    b.lt do_while      // если X0 меньше числа 5, то переходим обратно к метке while

mov x0, #0
mov X16, #1
svc #0x80
```

Циклы Apple ARM 64 (3/6)

Реализация for

```
.global _main
.align 2

.text
_main:

    mov x0, #0           // помещаем в регистр X0 число 0

    for_start:           // метка, на которую проецируется цикл
        cmp x0, #5         // сравниваем с некоторым пределом
        b.ge for_end       // условие – если счетчик больше или равен пределу, выход из цикла
        // выполняемые действия
        add x0, x0, 1       // действия цикла – увеличение счетчика
        b for_start         // повторяем цикл
    for_end:

    mov x0, #0
    mov X16, #1
    svc #0x80
```

Циклы Apple ARM 64 (4/6)

Оптимизация for и while

```
.global _main
.align 2

.text
_main:

    mov x0, #0          // помещаем в регистр X0 число 0
    b compare           // переходим к проверке условия
    while:              // собственно действия цикла while
        add x0, x0, #1   // X0 = X0 + 1 – выполняемые действия цикла
    compare:             // проверка условия
        cmp x0, #5       // сравнивание значение регистра X0 с числом 5
        b.lt while        // если X0 меньше числа 5, то переходим обратно к метке while

    mov x0, #0
    mov X16, #1
    svc #0x80
```

Циклы Apple ARM 64 (5/6)

Оптимизация do..while

```
.global _main
.align 2

.text
_main:

    mov x0, #5

do_while1:
    subs x0, x0, #1           // собственно действия цикла while
    b.ne do_while1             // X0 = X0 - 1 – выполняемые действия цикла
                                // проверка условия – если X0 != 0, то переходим обратно к метке
do_while1

    mov x0, #-5
do_while2:
    add x0, x0, #1            // собственно действия цикла while
    b.ne do_while2             // X0 = X0 + 1 – выполняемые действия цикла
                                // проверка условия – если X0 != 0, то переходим обратно к метке
do_while2

    mov x0, #0
    mov X16, #1
    svc #0x80
```

Циклы Apple ARM 64 (6/6)

Сумма элементов

```
.global _main
.align 2
.data
    nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
    count: .quad (.−nums)/8
.text
_main:

//загружаем адрес массива
adrp X1, nums@PAGE
add X1, X1, nums@PAGEOFF

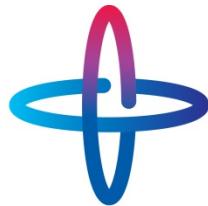
//загружаем адрес кол-ва элементов массива
adrp X2, count@PAGE
add X2, X2, count@PAGEOFF

ldr x0, [x2]           //сохраняем в x0 кол-во элементов
eor x4, x4, x4         //обнуляем регистр x4 для суммы (эквивалент команды xor в arm)

do_while:
    ldr x3, [x1], #8   // цикл do_while
    add x4, x4, x3     //загружаем с постиндексом
    subs x0, x0, #1     //считаем сумму с x4
    b.ne do_while       // X0 = X0 - 1 - выполняемые действия цикла
                        // проверка условия – если X0 != 0, то переходим обратно к метке
do_while

mov x0, x4 //Выводим сумму через код возврата (как пример)

mov X16, #1
svc #0x80
```



гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru

Отладка приложения

Практическое руководство

Отладка программ в GAS в консоли

Сборка с отладочными символами

```
as asm.s -o test.o -gstabs+ && ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc test.o -o test
```

```
> gdb test
GNU gdb (Ubuntu 13.1-2ubuntu2.1) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...
(gdb) █
```

(gdb) █
Бередитнің алшорас тұрғы test...
Тәле „аудиодар мәнди“ тәртіпке алынғанда көрсетілген „word“...
Көмекшіл аудиодар мәнди...

Отладка программ в GAS в консоли

Установка точки останова на метке `_start`

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...
(gdb) b _start
Breakpoint 1 at 0x401000: file asm.s, line 9.
(gdb) run
Starting program: /home/sprainbrains/Work/Local/Assembler/test

Breakpoint 1.2, 0x00007ffff7fe4de0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb) █
```

(gdb) █

Установка точки останова на строке 12

```
Breakpoint 1 at 0x401016: file asm.s, line 13.
(gdb) █
```

(gdb) disassemble _start

Dump of assembler code for function _start:

```
0x0000000000401000 <+0>:    mov    0x403040,%rcx
0x0000000000401008 <+8>:    mov    $0x0,%rdi
0x000000000040100f <+15>:   mov    $0x0,%rsi
End of assembler dump.
(gdb) █
```

Дисассемблированный листинг

Отладка программ в GAS в консоли

Следующая итерация step over

```
End of assembler dump.
(gdb) n
Single stepping until exit from function _start,
which has no line number information.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1.1, _start () at asm.s:9
9          movq count, %rcx      # помещаем в RCX количество элементов в массиве nums
(gdb) n
δ          шовд count, %rcx      # помещаем в RCX количество элементов в массиве nums
всеякботиј т.т. Гетиј () ят асшасија:9
```

Следующая итерация step in

```
(gdb) disassemble
Dump of assembler code for function _start:
⇒ 0x0000000000401000 <+0>:    mov    0x403040,%rcx
    0x0000000000401008 <+8>:    mov    $0x0,%rdi
    0x000000000040100f <+15>:   mov    $0x0,%rsi
End of assembler dump.
(gdb) s
10          movq $0, %rdi      # будущая сумма элементов массива
(gdb) ■
```

```
(dqr) ■
10          movq $0, %rdi      # будущая сумма элементов массива
(gdb) a
```

Отладка программ в GAS в консоли

Вывод все регистров на экран

```
(gdb) i r
rax          0x38          56
rbx          0x0           0
rcx          0x8           8
rdx          0x7ffff7fccfb0 140737353928624
rsi          0x7ffff7ffe880 140737354131584
rdi          0x7ffff7ffe2c0 140737354130112
rbp          0x0           0x0
rsp          0x7fffffff060 0x7fffffff060
r8           0x7fffffff33b 140737488347963
r9           0x1           1
r10          0x7ffff7fc8858 140737353910360
r11          0x206          518
r12          0x401000        4198400
r13          0x7fffffff060 140737488347232
r14          0x0           0
r15          0x0           0
rip          0x401008        0x401008 <_start+8>
eflags       0x206          [ PF IF ]
cs           0x33          51
ss           0x2b          43
ds           0x0           0
es           0x0           0
fs           0x0           0
gs           0x0           0
(gdb) ■
```

```
(дqr) ■
ds           0x0           0
fs           0x0           0
es           0x0           0
qs           0x0           0
```

Вывод определенного регистра

```
(gdb) i r rax
rax          0x38          56
(gdb) ■
```

Форматированный вывод

```
(gdb) p/x $rax
$1 = 0x38
(gdb) p/d $rax
$2 = 56
(gdb) p/tt $rax
$3 = 111000
(gdb) ■
```

```
(дqr) ■
```

Выход из отладки

```
(gdb) exit
A debugging session is active.

Inferior 1 [process 4357] will be killed.

Quit anyway? (y or n) y
```

Документация (λ ον ι)

Отладка программ в GAS в TUI - консоли

Запуск TUI интерфейса отладчика путем запуска `layaout` с отображение регистров

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/qdb/documentation/>.
```

For help, type "help".

Type "apropos word" to search for commands related to "word"....

Reading symbols from test...

(gdb) layout regs

(ðqp) ʃəλouf ւeðs

Отладка программ в GAS в TUI - консоли

TUI интерфейс отладчика

Интерактивная область исходного кода. Можно прокручивать мышкой

[Register Values Unavailable]

```
asm.s
1 .globl _start
2
3 .data
4 nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
5 count: .quad (. nums)/8 # количество элементов в массиве nums
6 .text
7 _start:
b+ 8    movq count, %rcx      # помещаем в RCX количество элементов в массиве nums
b+ 9    movq $0, %rdi        # будущая сумма элементов массива
b+ 10   movq $0, %rsi        # индексный регистр
1 main_loop:
12    addq nums(%rsi, 8), %rdi # складываем значение по адресу из (nums+rsi*8) с числом в RDI
```

Значения регистров

```
exec No process In:  
warning: Source file is more recent than executable.  
(gdb) b asm.s:6  
Breakpoint 1 at 0x401000: file asm.s, line 8. ←  
(gdb) b asm.s:10  
Breakpoint 2 at 0x40100f: file asm.s, line 10. ←  
(gdb) info breakpoints  
Num Type Disp Enb Address What  
1 breakpoint keep y 0x0000000000401000 asm.s:8  
2 breakpoint keep y 0x000000000040100f asm.s:10  
(gdb)
```

L?? PC: ??

Точки останова

Отладка программ в GAS в TUI - консоли

Процесс отладки

Register group: general									
rax	0x1c	28	rbx	0x0	0	rcx	0x8	8	
rdx	0x7fff7fc9040	140737353912384	rsi	0xfffff7ffe88	140737354131592	rdi	0x7ffff7ffe2e0	140737354130144	
rbp	0x0	0x0	rsp	0x7fffffff280	0x7fffffff280	r8	0xff	255	
r9	0xf	15	r10	0x7ffff7fc3860	14073735388988	r11	0x202	514	
r12	0x401000	4198400	r13	0x7fffffff280	14073748347776	r14	0x0	0	
r15	0x0	0	rip	0x401008	0x401008 <_start+8>	eflags	0x206	[PF IF]	
cs	0x33	51	ss	0x2b	43	ds	0x0	0	
es	0x0	0	fs	0x0	0	gs	0x0	0	
k0	0x800080	8388736	k1	0x44004	278532	k2	0x0	0	
k3	0x0	0	k4	0x0	0	k5	0x0	0	
k6	0x0	0	k7	0x0	0				

```
asm.s
2
3 .data
4 nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
5 count: .quad (. - nums)/8          # количество элементов в массиве nums
6 .text
7 _start:
8     movq count %rcx             # помещаем в RCX количество элементов в массиве nums
9     movq $0, %rdi              # будущая сумма элементов массива
10    movq $0, %rsi              # индексный регистр
11 main_loop:
12     addq nums(%rsi, 8), %rdi   # складываем значение по адресу из (nums+rsi*8) с числом в RDI
13     incq %rsi                 # прибавляем 1 - получаем индекс следующего элемента
14     loopq main_loop           # если счетчик RCX не равен 0, то переходим обратно к main_loop
```

Изменение регистров после выполнения предыдущей инструкции

Текущая позиция

Запустили отладку

Следующий шаг step over

```
multi-thre Thread 0x7ffff7fa57 In: _start
Breakpoint 1 at 0x401000: file asm.s, line 8.
(gdb) b asm.s:10
Breakpoint 2 at 0x40100f: file asm.s, line 10.
(gdb) info breakpoints
Num      Type            Disp Enb Address          What
1        breakpoint      keep y  0x0000000000401000 asm.s:8
2        breakpoint      keep y  0x000000000040100f asm.s:10
(gdb) run
Starting program: /home/sinyov/Documents/Assembler/test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _start () at asm.s:8
(gdb) n
(gdb) █
```

```
    .as_dqdp(■  
    .u  
    .dqdp)  
    base = () first = 't'  
    break;
```

Отладка программ в GAS VS Code

Необходимые платины, которые надо поставить

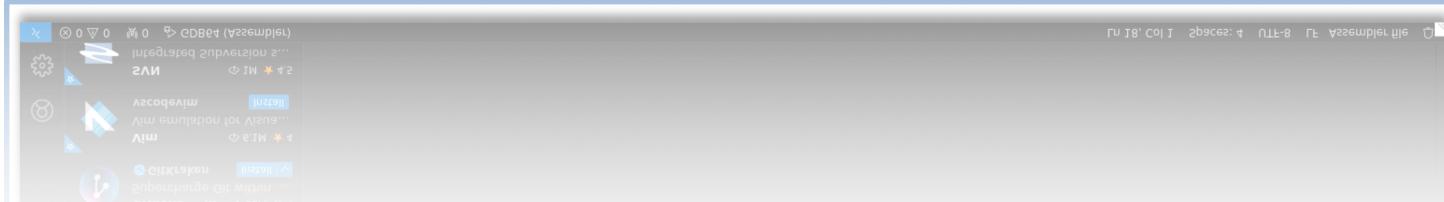
The screenshot shows the Visual Studio Code interface with the title bar "asm.s - Assembler - Visual Studio Code". The left sidebar is titled "EXTENSIONS" and displays the "INSTALLED" section with the following extensions:

- ASM Code Lens (by maziac)
- gas-highlight (by Benjamin Jurk)
- GDB Debug (by DamianKoper)
- GNU Assembler Language (by Bas du Pré)

The main editor area contains assembly code for a program named "asm.s". The code defines a global variable _start, a data section with an array of integers, and a main loop that calculates the sum of the array elements. Comments in the code provide explanatory text for each part.

```
.globl _start
...
.data
...
nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
count: .quad (. - nums)/8      # количество элементов в массиве nums
.text
_start:
    movq count, %rcx          # помещаем в RCX количество элементов в массиве nums
    movq $0, %rdi             # будущая сумма элементов массива
    movq $0, %rsi              # индексный регистр
main_loop:
    addq nums(%rsi, 8), %rdi   # складываем значение по адресу из (nums+rsi*8) с числом в RDI
    incq %rsi                 # прибавляем 1 - получаем индекс следующего элемента
    loop main_loop            # если счетчик RCX не равен 0, то переходим обратно к main loop
    movq $60, %rax
    syscall
```

The bottom of the sidebar also lists "RECOMMENDED" extensions: GitLens, GitKraken, Vim, vscodevim, and SVN.



Отладка программ в GAS

В папку «.vscode», которая должна быть в одной директории с вашим asm файлом/файлами создать 2 файла: launch.json tasks.json

Launch.json

```
{  
    "version": "0.2.0",  
    "configurations": [  
        {  
            "type": "gdb",  
            "request": "launch",  
            "name": "GAS Build & Debug",  
            "program": "${fileDirname}/${fileBasenameNoExtension}",  
            "stopOnEntry": false,  
            "preLaunchTask": "gas build"  
        }  
    ]  
}
```

Структура папки

```
sinyov@ASSIST:~/Documents/Assembler$ ls -aR  
.:  
.. .. asm asm.o asm.s test test.o .vscode  
./.vscode:  
.. .. launch.json tasks.json  
sinyov@ASSIST:~/Documents/Assembler$
```

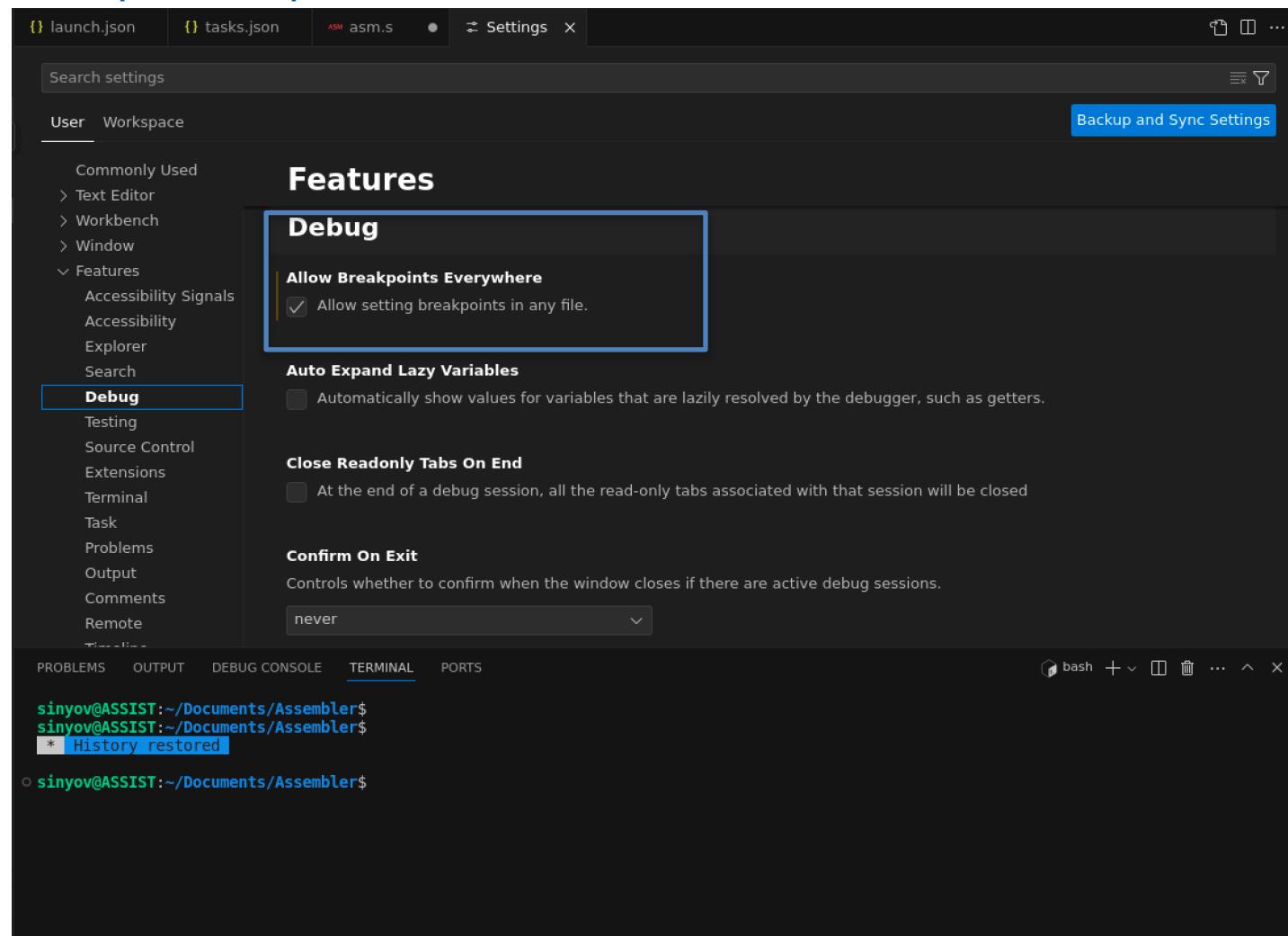
Чтобы открыть папку .vscode в VSCode, нажмите на нее.

tasks.json

```
{  
    "version": "2.0.0",  
    "tasks": [  
        {  
            "label": "gas build",  
            "type": "shell",  
            "command": "rawfilename=${fileDirname}/${fileBasenameNoExtension}; as --gstabs+ ${file} -o $rawfilename.o &&  
ld -m elf_x86_64 $rawfilename.o -o $rawfilename",  
            "problemMatcher": {  
                "pattern": {  
                    "regexp": "error"  
                }  
            },  
            "presentation": {  
                "focus": true,  
                "panel": "dedicated",  
                "reveal": "silent",  
                "clear": true  
            }  
        }  
    ]  
}
```

Отладка программ в GAS VS Code

Активировать настройки точек останова



The screenshot shows the VS Code settings interface. The left sidebar has a 'Debug' section selected. In the main area, the 'Features' section is expanded, and the 'Debug' subsection is highlighted with a blue border. Inside this section, the 'Allow Breakpoints Everywhere' option is shown with a checked checkbox and the description: 'Allow setting breakpoints in any file.' Below it are other options like 'Auto Expand Lazy Variables' and 'Close Readonly Tabs On End'. At the bottom of the settings page, there's a terminal window showing a bash session with history restored.

launch.json tasks.json asm.asm Settings

Search settings

User Workspace

Backup and Sync Settings

Features

Debug

Allow Breakpoints Everywhere
 Allow setting breakpoints in any file.

Auto Expand Lazy Variables
 Automatically show values for variables that are lazily resolved by the debugger, such as getters.

Close Readonly Tabs On End
 At the end of a debug session, all the read-only tabs associated with that session will be closed

Confirm On Exit
Controls whether to confirm when the window closes if there are active debug sessions.
never

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

sinyov@ASSIST:~/Documents/Assembler\$
sinyov@ASSIST:~/Documents/Assembler\$ * History restored
sinyov@ASSIST:~/Documents/Assembler\$

Отладка программ в GAS VS Code

Процесс отладки. Важно, регистры могут не отображаться... нужно использовать консоль

The screenshot shows the Visual Studio Code interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Toolbar:** RUN AND DEBUG, GAS Build & Debug, ...
- Sidebar:** VARIABLES (Registers, WATCH), CALL STACK (Paused on breakpoint, _start, asm.s (8)), BREAKPOINTS (asm.s, asm.s), HEXADECIMAL CALCULATOR.
- Editor:** File: asm.s - Assembler - Visual Studio Code. The code is:

```
1 .globl _start
2
3 .data
4 nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
5 count: .quad (. - nums)/8          # Количество элементов в массиве nums
6 .text
7 start:
8     movq count, %rcx             # помещаем в RCX количество элементов в массиве nums
9     movq $0, %rdi               # будущая сумма элементов массива
10    movq $0, %rsi               # индексный регистр
11    main_loop:
12        addq nums(%rsi, 8), %rdi   # складываем значение по адресу из (nums+rsi*8) с числом в RDI
13        incq %rsi                # прибавляем 1 - получаем индекс следующего элемента
14        loopq main_loop          # если счетчик RCX не равен 0, то переходим обратно к main_loop
15
16    movq $60, %rax
17    syscall
18
19
```
- Bottom Bar:** PROBLEMS, OUTPUT, DEBUG CONSOLE (selected), TERMINAL, PORTS. The DEBUG CONSOLE tab shows the output of the assembly code for the breakpoint:Breakpoint 1, _start () at /home/sinyov/Documents/Assembler/asm.s:8
8 movq count, %rcx # \320\277\320\276\320\274\320\265\321\211\320\260\320\265\320\274 \320\262 RCX \320\272\320\276\320\273\320\270\321\207\320\265\321\201\321\202\320\262\320\276 \321\215\320\273\320\265\320\274\320\265\320\275\321\202\320\276\320\262 \320\262 \320\274\320\260\321\201\320\270\320\262\320\265 nums
- Status Bar:** Ln 8, Col 1, Spaces: 4, UTF-8, LF, Assembler file.

Отладка программ в GAS SAMS

The screenshot shows the GAS SAMS debugger interface. On the left, the assembly code for a program named GASSumx64.asm is displayed:

```
1 .data
2 a: .quad 0
3 b: .quad 0
4 inputFormat: .asciz "%lld%lld"
5 outputFormat: .asciz "%lld"
6
7 .extern printf
8 .extern scanf
9 .text
10 .global main
11 main:
12     movq %rsp, %rbp # for correct debugging
13     subq $32, %rsp
14     andq $-16, %rsp
15     movq $inputFormat, %rcx
16     movq $a, %rdx
17     movq $b, %r8
18     call scanf
19     movq (%a), %rdx
20     addq (%b), %rdx
21     movq $outputFormat, %rcx
22     call printf
23     movq %rbp, %rsp
24     xorq %rax, %rax
25     ret
26
27
```

At the bottom, the build log shows:

```
[13:44:43] Построение начато...
[13:44:44] Программа построена успешно.
[13:45:13] Программа выполняется...
[13:45:13] Программа выполнена успешно. Время выполнения: 0.034 с
[13:45:36] Программа выполняется...
[13:45:36] Программа выполнена успешно. Время выполнения: 0.035 с
```

The right side of the interface features two windows: "Ввод" (Input) and "Вывод" (Output). The "Ввод" window contains the input values "10 11". The "Вывод" window contains the output value "21". Arrows point from the text labels "Консоль ввода" and "Консоль вывода" to their respective windows.

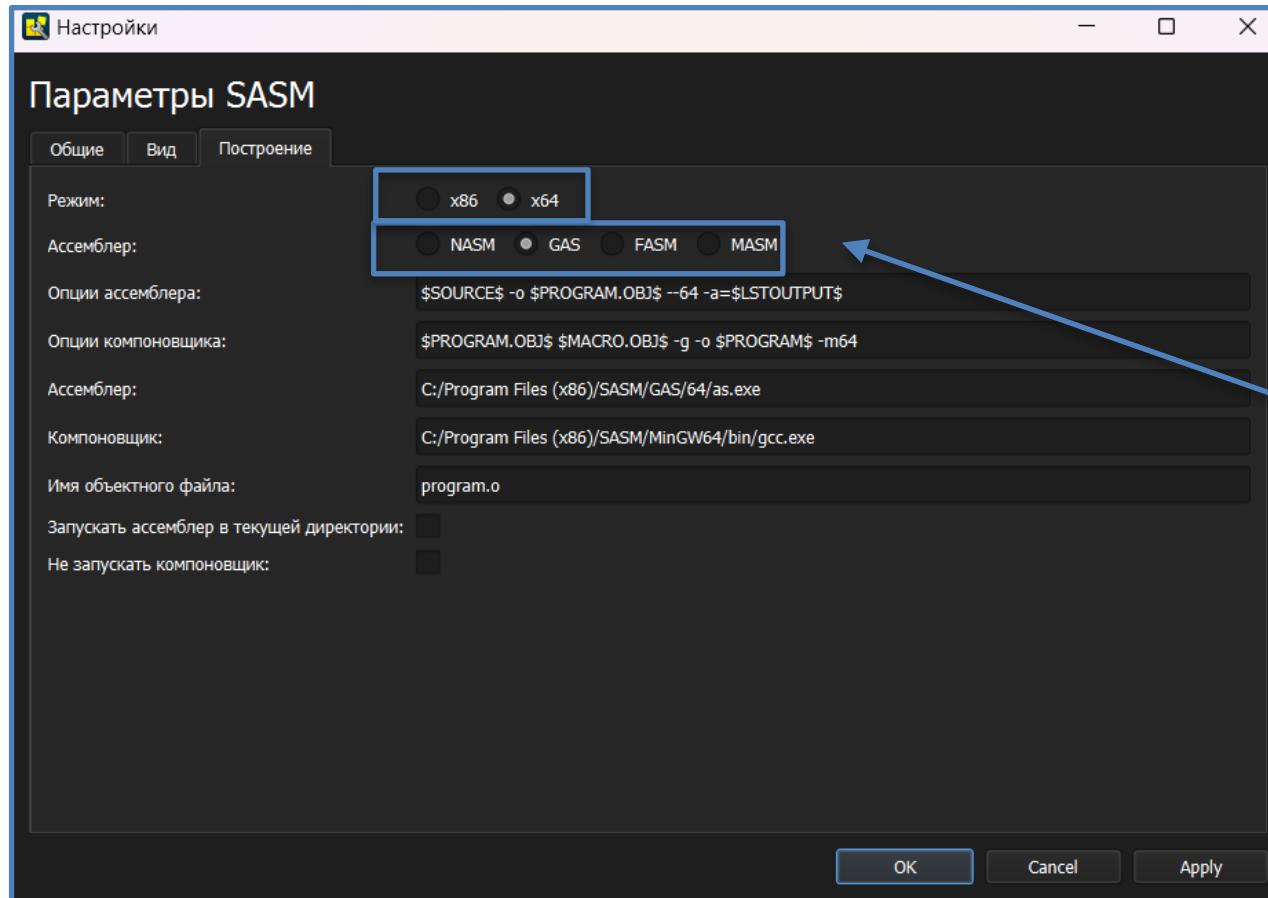
Скачать через сайт
или github

[http://
dman95.github.io/
SASM/](http://dman95.github.io/SASM/)

Есть под Windows и
Linux.

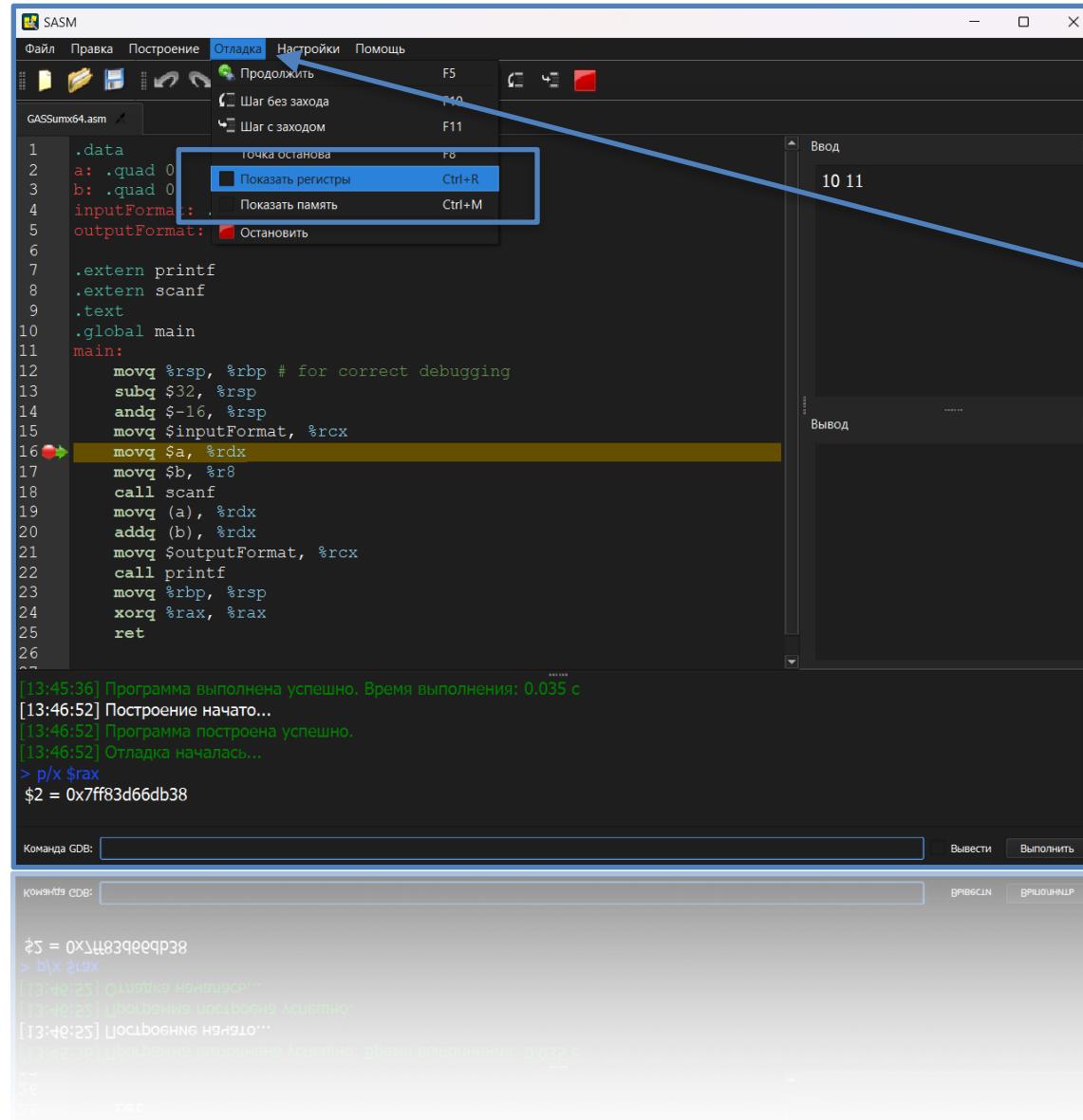
Использует
прослойки MinGW
для работы.

Отладка программ в GAS SAMS



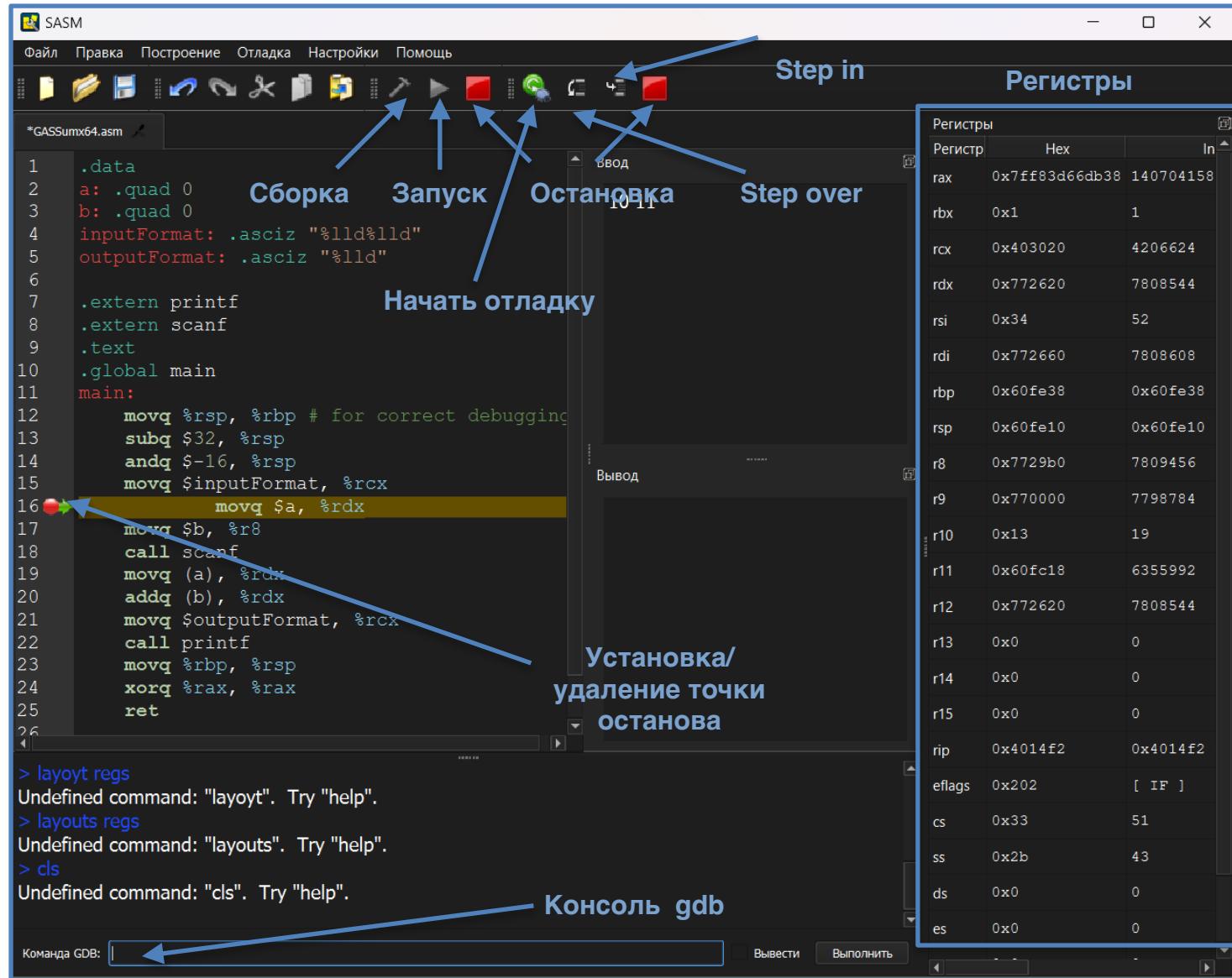
Необходимые
настройки для
работы

Отладка программ в GAS SAMS



Включаем
отображение
регистров при
отладке

Отладка программ в GAS SAMS



Отладка программ Apple ARM64

В среде Xcode и терминале отладчика lldb

The screenshot shows the Xcode IDE with an assembly file named 'test.s' open. The assembly code implements a sum of array elements algorithm. A blue arrow points from the text 'Сборка и запуск отладки (CMD + r)' to the play button icon in the top-left corner of the Xcode window. Another blue arrow points from the text 'Установка точки останова' to the instruction at line 19: 'adrp X1, nums@PAGE'. The text 'Окно отладчика LLDB' is positioned above the LLDB console window, which displays the message 'Message from debugger: killed Program ended with exit code: 9'. The text 'Окно вывода значений переменных' is positioned above the variable value output window.

```
.data
11     nums: .quad 1, 2, 3, 4, 5, 6, 7, 8
12     count: .quad (. - nums)/8
13
14 .text
15 _main:
16
17
18 //загружаем адрес массива
19 adrp X1, nums@PAGE
20 add X1, X1, nums@PAGEOFF
21
22 //загружаем адрес кол-ва элементов массива
23 adrp X2, count@PAGE
24 add X2, X2, count@PAGEOFF
25
26 ldr x0, [x2]          //сохраняем в x0 кол-во элементов
27 eor x4, x4, x4        //обнуляем регистр x4 для суммы (эквивалент команды xor в arm)
28
29 do_while:             // цикл do_while
30     ldr x3, [x1], #8   //загружаем с постиндексом
31     add x4, x4, x3    //считаем сумму с x4
32     subs x0, x0, #1    // X0 = X0 - 1 - выполняемые действия цикла
33     b.ne do_while      // проверка условия - если X0 ≠ 0, то переходим обратно к метке do_while
34
35
36     mov x0, x4 //Выводим сумму через код возврата (как пример)
37
38     mov X16, #1
39
40
```

Сборка и запуск отладки (CMD + r)

Установка точки останова

Окно отладчика LLDB

Message from debugger: killed
Program ended with exit code: 9

Окно вывода значений переменных

Отладка программ Apple ARM64

Запущенная отладка

The screenshot shows the Xcode debugger interface with the following details:

- Top Bar:** Shows the project name "test", file "main.m", and build configuration "test1". Status bar indicates "Paused test".
- Left Sidebar:** Monitors CPU usage (0%), Memory (3.8 MB), Energy Impact (Zero), Disk (Zero KB/s), and Network (Zero KB/s). A thread monitor shows "Thread 1 Queue:... (serial)" with one main thread.
- Code Editor:** Displays assembly code for "main.m". A specific line is highlighted:

```
19 adr  X1, nums@PAGE
20 add  X1, X1, nums@PAGEOFF
```

A tooltip for this line states: "Thread 1: breakpoint 1.1 (1)".
- Bottom Registers View:** Shows the state of general purpose registers (X0-X15) and floating-point registers. A blue arrow points from the text "Кнопки линейного прохода по коду" to the left edge of the register table.
- Bottom Buttons:** Includes standard debugger controls like step, break, and quit.
- Bottom Status:** Shows the current line (Line: 24) and column (Col: 29).

Annotations:

- Кнопки линейного прохода по коду**: Points to the left edge of the register table.
- Все регистры**: Points to the "General Purpose Registers" section in the bottom register view.

Отладка программ Apple ARM64

Формат вывода значений

```
21
22 //загружаем адрес кол-ва элементов в регистр X2
23 adrp X2, count@PAGE
24 add X2, X2, count@PAGEOFF
25
26 ldr x0, [x2]           // со x
27 eor x4, x4, x4         /
28
29 do_while:              // цикл
30     ldr x3, [x1], #8    // загрузка
31     add x4, x4, x3      // счёта
32     subs x0, x0, #1      // x0
33     b.ne do_while        // прерывание
34
35 Print Description of "General Purpose Registers"
36 Copy
37 View Value As >
38 Edit Value...
39 Edit Summary Format...
40
41 Add Expression...
42 Delete Expression
43
44 > E
45 > F
46 > C
47 > D
48 > B
49 > A
50
51 x0 Show Types
52 x1 Show Raw Values
53 x2 Sort By >
54 x3 Debug Area Help
55 All c
```

Custom Type...
Default
Binary
Boolean
Bytes (char)
Bytes (hex)
Bytes (hex with ASCII)
OSType
C String
Complex
Float
Unicode 16
Unicode 32
Decimal
✓ Hex
Pointer
Octal
Unsigned Decimal
Vector of char
Vector of sint8
Vector of uint8
Vector of sint16
Vector of uint16
Vector of sint32
Vector of uint32
Vector of sint64
Vector of uint64
Vector of float32
Vector of float64
Vector of uint128

во элементов
р x4 для суммы

вком

пняемые действии

если $X0 \neq 0$,

(как пример)

Вывод регистров в терминале отладки

```
(lldb) register read -all
General Purpose Registers:
    x0 = 0x0000000000000001
    x1 = 0x000000016fdfff568
(lldb)
```

Вывод регистров в терминале отладки определенного регистра

```
        esr = 0xf2000000
exception = 0x00000000
(lldb) register read x0
x0 = 0x0000000000000001
(lldb) |
```

Форматированный вывод регистров

Программирование на языках Ассемблера

Лекция 5

Синёв Николай Иванович
Кафедра 14



Логические операции

ARM64 Apple

AND{S} Xd, Xs, Operand2

```
.global _start
.align 2
_start:
    mov w0, #12          // помещаем в регистр W0
число 12 - 1100
    and w0, w0, #6       // помещаем в регистр W0 = W0
AND 6 = 1100 AND 0110 = 0100 = 4

    mov     X16, #1
    svc     #0x80
```

GAS

AND{} Xs, Xd

```
.globl _start
.text
_start:
    movq $12, %rdi  # помещаем в регистр RDI
число 12 - 1100
    andq $6, %rdi   # rdi = rdi AND 6 = 1100 AND
0110 = 0100 = 4

    movq $60, %rax
    syscall
```

ORR{S} Xd, Xs, Operand2

```
.global _start
.align 2
_start:
    mov w0, #12          // помещаем в регистр W0
число 12 - 1100
    orr w0, w0, #6       // помещаем в регистр W0 = W0
OR 6 = 1100 OR 0110 = 1110 = 14

    mov     X16, #1
    svc     #0x80
```

OR{} Xs, Xd

```
.globl _start
.text
_start:
    movq $12, %rdi  # помещаем в регистр rdi
число 12 - 1100
    orq $6, %rdi   # rdi = rdi OR 6 = 1100 OR
0110 = 1110 = 14

    movq $60, %rax
    syscall
```

Логические операции

ARM64 Apple

GAS

EOR{S} Xd, Xs, Operand2

```
.global _start
.align 2
_start:
    mov w0, #12          // помещаем в регистр W0
число 12 - 1100
    eor w0, w0, #6        // помещаем в регистр W0 = W0
XOR 6 = 1100 XOR 0110 = 1010 = 10

    mov     X16, #1
    svc     #0x80
```

XOR{} Xs, Xd

```
.globl _start
.text
_start:
    movq $12, %rdi      # помещаем в регистр RDI
число 12 - 1100
    xorq $6, %rdi       # rdi = rdi XOR 6 = 1100 XOR
0110 = 1010 = 10

    movq $60, %rax
    syscall
```

BIC{S} Xd, Xs, Operand2

```
.global _start
.align 2
_start:
    mov w0, #12          // помещаем в регистр W0
число 12 - 1100
    orr w0, w0, #6        // помещаем в регистр W0 = W0
AND NOT 6 = 1100 AND NOT 0110 = 1000 = 8

    mov     X16, #1
    svc     #0x80
```

NOT Xd

```
.globl _start
.text
_start:
    xorq %rdi, %rdi    # rdi = 0
    movq $12, %rdi      # помещаем в регистр rdi
число 12 - 00000000 00000000 00000000 000001100
    orq $6, %rdi       # rdi =NOT(rdi)=NOT(12)=
11111111 11111111 11111111 11110011

    movq $60, %rax
    syscall
```

Логические операции

ARM64 Apple

GAS

ORN Xd, Xs, Operand2

```
.global _start
.align 2
_start:
    mov w0, #0b1100 // помещаем в регистр w0 число
12 - 1100
    mov w1, #0b0110 // помещаем в регистр w1 число
6 - 0110
    orn w0, w0, w1 // w0 = w0 orn w1 = 1100 or
not 0110 = 1100 or 1001 = 1101 = 1101 = -3

    mov      X16, #1
    svc      #0x80
```

Поразрядная инверсия

```
.global _start
.align 2
_start:
    mov x0, #0b1100 // помещаем в регистр x0 число
12 - 1100
    orn x0, xzr, x0 // инвертируем число в x0

    //Эквивалент
    mov x0, #0b1100 // помещаем в регистр x0 число
12 - 1100
    mvn x0, x1      // инвертируем число в x1 и
помещаем его в x0

    mov      X16, #1
    svc      #0x80
```

NEG Xd

```
.globl _start
.text
_start:
    movq $-12, %rdi # помещаем в регистр RDI
число -12
    neg %rdi # rdi = -1 * rdi = -1 * -12 = 12

    movq $60, %rax
    syscall
```

Операции сдвига

ARM64 Apple

mov Xd, Xs, [Оператор_сдвига]

количество_разрядов

Где оператор сдвига

- **LSL**: логический сдвиг влево
 - **LSR**: логический сдвиг вправо
 - **ASR**: арифметический сдвиг вправо
- (С Xcode 13 использовать вызовы ниже)

LSL Xd, Xs, #1 - логический сдвиг влево

на 1 разряд

LSR Xd, Xs, #1 - логический сдвиг вправо

на 1 разряд

ASR Xd, Xs, #1 - арифметический сдвиг

вправо на 1 разряд

GAS

shl count, dest - логический сдвиг влево

shr count, dest - логический сдвиг вправо

sar count, dest - арифметический сдвиг
вправо

Операции сдвига

ARM64 Apple

GAS

```
.global _start
.align 2
_start:
    mov X1, #13
    lsl x0, x1, #2 // сдвигаем число из X1 (число 13)
на 2 разряда влево

    //Логический сдвиг вправо
    mov X1, #13
    lsr x0, x1, #2 // сдвигаем число из X1 (число 13)
на 2 разряда влево

    //Арифметический сдвиг вправо
    mov X1, #-13      //
0b1111111111111111111111111111110011
    asr x0, x1, #2      // сдвигаем число из X1
(число 13) на 2 разряда вправо

    mov    X16, #1
    svc    #0x80
```

```
.globl _start
.text
_start:
    //Логический сдвиг влево
    movq $5, %rdi      # в RDI число 5 или
00000101
    shlq $1, %rdi      # сдвигаем число в RDI на
1 разряд влево = 00000101 << 1 = 00001010 = 10

    //Логический сдвиг вправо
    movq $69, %rdi     # в RDI число 69 или
01000101
    shrq $2, %rdi      # сдвигаем число в RDI на
2 разряда вправо = 01000101 >> 2 = 00010001 = 17

    //Арифметический сдвиг вправо
    movq $-32, %rdi    # в RDI число -8 или
FFFFFE0h
    sarq $4, %rdi      # сдвигаем число в RDI на
4 разряда вправо = FFFFEE0 >> 4 = FFFFFFFE = -2

    movq $60, %rax
    syscall
```

Операции вращения (цикл. сдвига)

ARM64 Apple

GAS

mov Xd, Xs, [Оператор_вращения]

количество_разрядов

Где оператор

- **ROR**: вращение вправо

(С Xcode 13 использовать вызов ниже)

ROR Xd, Xs, #1 - вращение вправо на 1

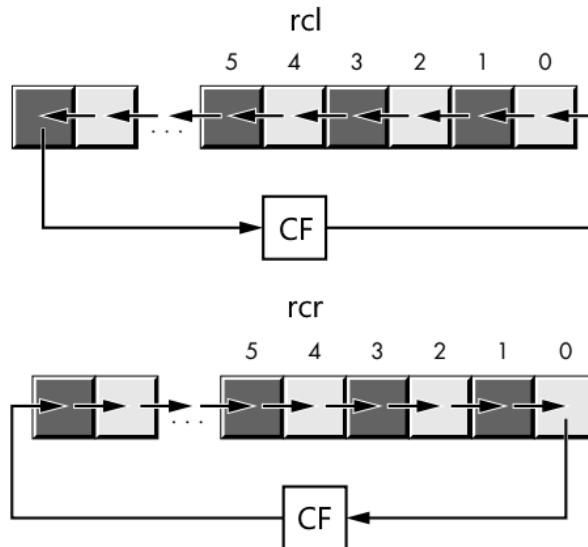
разряд

rol count, dest - вращение влево

ror count, dest - вращение вправо

rcl count, dest - вращение влево с переносом

rcr count, dest - вращение вправо с переносом



Операции вращения (цикл. сдвига)

ARM64 Apple

```
.global _start
.align 2
_start:

    //Вращение вправо
    mov X1, #-13          //
0b1111111111111111111111111111110011
    ror x0, x1, #2         // сдвигаем число из X1
(число 13) на 2 разряда вправо

    mov      x16, #1        //системный вызов 1 завершает
программу
    svc      #0x80          //вызываем системную функцию с
номером 1
```

```
.globl _start
.text
_start:
    //Вращение влево
    movq $131, %rax      # в AL число 131 или
10000011
    rolb $2, %al          # вращаем число в AL на 2
разряда влево = 10000011 << 2 = 00001110 = 14

    //Вращение вправо
    movq $131, %rax      # в AL число 131 или
10000011
    rorb $2, %al          # вращаем число в AL на 2
разряда вправо = 10000011 >> 2 = 11100000 = 224

    //Вращение влево с переносом
    movq $131, %rax      # в AL число 131 или
10000011
    rclb $2, %al          # вращаем число в AL на 2
разряда влево = 10000011 << 2 = 00001101 = 13

    movq %rax, %rdi
    movq $60, %rax
    syscall
```

GAS

Стек

ARM64 Apple

GAS

```
.global _start
.align 2
_start:

    str x1, [sp, #-16]!      // сохраняем значение из
X1 по адресу (SP - 16) и обновляем значение SP
    ldr x0, [sp], #16        // получаем данные в X0
по адресу в SP и обновляем значение SP (SP = SP + 16)

//сохранение и загрузка сразу в 2 регистра
    mov x2, 22              // данные для сохранения
в стек
    mov x3, 33
    stp x2, x3, [sp, #-16]!  // сохраняем значение
из X2, X3 по адресу (SP - 16) и обновляем значение SP
    ldp x0, x1, [sp], #16    // получаем данные в
X0, X1 по адресу в SP и обновляем значение SP (SP =
SP + 16)

    mov     X16, #1
    svc    #0x80
```

```
.globl _start
.text
_start:

#сохранение и загрузка на/с стека
    movq $11, %rdi
    movq $33, %rdx

    pushq %rdi
    pushq %rdx
    popq %rdi
    popq %rdx

#сохранение флагов состояний
    pushfq    # сохраняем значения флагов
    movb $255, %al
    addb $2, %al    # 255 + 2 = 257 – флаг CF будет
 установлен
    popfq      # восстанавливаем значения флагов

#восстановление состояние стека 1
    pushq %rdi
    pushq %rdx
    addq $16, %rsp      # прибавляем к адресу в RSP 16
байт

#восстановление состояние стека 2
    subq $16, %rsp # резервируем в стеке 16 байт
### некоторая работа со стеком
    addq $16, %rsp      # восстанавливаем значение
стека

    movq %rax, %rdi
    movq $60, %rax
    syscall
```

Стек

ARM64 Apple

```
.global _start
.align 2
_start:
    mov x1, 11
    mov x2, 12
    mov x3, 13

    sub sp, sp, #32      // выделяем в стеке 32 байта
    str x1, [sp]          // сохраняем значение из X1
по адресу SP
    str x2, [sp, #8]      // сохраняем значение из X2
по адресу SP + 8
    str x3, [sp, #16]      // сохраняем значение из X3
по адресу SP + 16

    ldr x0, [sp, #8]      // получаем в X0 число по
адресу SP + 8
    add sp, sp, #32      // очищаем ранее выделенные в
стеке 32 байта

    mov X16, #1
    svc #0x80
```

GAS

```
.globl _start
.text
_start:

#косвенная адресация
subq $16, %rsp # резервируем в стеке 16 байт
movq $11, %rdx
movq %rdx, (%rsp)      # помещаем в стек
значение регистра RDX
movq (%rsp), %rdi      # в RDI помещаем значение
по адресу из RSP – число 11
addq $16, %rsp      # восстанавливаем значение
стека

pushq $12
pushq $13
pushq $14
pushq $15
movq 16(%rsp), %rdi      # 16(%rsp) – адрес
значения 13
addq $32, %rsp      # восстанавливаем значение
стека

subq $16, %rsp      # резервируем в стеке 16 байт
movq $12, %rcx
movq $13, %rdx
movq %rcx, 8(%rsp)      # 8(%rsp) = 12
movq %rdx, (%rsp)      # (%rsp) = 13
movq (%rsp), %rdi      # rdi= 13
addq 8(%rsp), %rdi      # rdi = rdi + 12
addq $16, %rsp      # восстанавливаем значение
стека

movq %rax, %rdi
movq $60, %rax
syscall
```

Программирование на языках Ассемблера

Лекция 6

Синёв Николай Иванович
Кафедра 14



ФУНКЦИИ В GAS

```
.globl _start

.text
_start:
    call sum      # вызываем функцию sum

    movq %rax, %rdi
    movq $60, %rax
    syscall
# определяем функцию sum
sum:
    movq $7, %rdi
    movq $5, %rsi
    addq %rsi, %rdi
    ret
```

ФУНКЦИИ В GAS. Заботимся о стеке

```
.globl _start
.text
_start:
    movq $3, %rdi
    movq $9, %rsi
    call sum

    movq $60, %rax
    syscall

sum:
    pushq %rsi # сохраняем регистр RSI в стек
    addq %rsi, %rdi # в RDI результат
    ret
```

```
.globl _start
.text
_start:
    movq $3, %rdi
    movq $9, %rsi
    call sum

    movq $60, %rax
    syscall

sum:
    popq %rsi # извлекаем из стека в регистр RSI адрес в
    addq %rsi, %rdi # в RDI результат
    ret
```

В обоих случаях получаем
segfault!

ФУНКЦИИ В GAS. Заботимся о стеке

```
.globl _start

.text
_start:
    movq $3, %rdi
    movq $9, %rsi
    call sum

    movq $60, %rax
    syscall

sum:
    pushq %rsi # сохраняем регистр RSI в стек
    popq %rsi # извлекаем из стека в регистр RSI адрес возврата

    addq %rsi, %rdi # в RDI результат
    ret
```

Адрес возврата валидный.

ФУНКЦИИ В GAS. Заботимся о стеке

```
.globl _start

.text
_start:
    movq $125, %rdi
    call sum
    addq %rax, %rdi      # RDI = 140

    movq $60, %rax
    syscall

sum:
    pushq %rdi          # сохраняем в стек регистр rdi
    movq $5, %rdi
    movq $10, %rax
    addq %rdi, %rax
    popq %rdi           # восстанавливаем из стека регистр
    rdi
    ret
```

Функции в GAS. Передача параметров

Согласно ABI, каждая функция должна сохранять значения этих регистров: %rbp, %rbx, %r12, %r13, %r14, %r15

Следующие регистры используются для передачи значений в функцию:

- 1 параметр - %rdi
- 2 параметр -%rsi
- 3 параметр -%rdx
- 4 параметр -%rcx
- 5 параметр -%r8
- 6 параметр -%r9

Если параметров > 6 - то их следует передавать через стек в виде чисел .quad

Возвращаемое значение из функции в %rax (второе можно хранить в %rdx).

Если нужно больше - выделяем переде работой в стеке данные и туда потом сохраняем.

Функции в GAS.

Передача параметров через регистры

```
.globl _start

.text
_start:
    movq $3, %rcx      # первый параметр для функции sum
    movq $4, %rdx      # второй параметр для функции sum
    call sum

    movq $60, %rax
    syscall

sum:
    movq %rcx, %rdi   # в RDI копируем число из RCX
    addq %rdx, %rdi   # складываем с числом из RDX
    ret
```

Функции в GAS.

Передача параметров через стек

```
.globl _start

.text
_start:
    pushq $1  # 24(%rsp)
    pushq $2  # 16(%rsp)
    pushq $3  # 8(%rsp)

    call sum

    addq $24, %rsp

    movq $60, %rax
    syscall

sum:
    movq 24(%rsp), %rdi    # RDI = 1
    addq 16(%rsp), %rdi    # RDI = RDI + 2 = 3
    addq 8(%rsp), %rdi     # RDI = RDI + 3 = 6
    ret
```

Не оптимальный способ

Функции в GAS.

Передача параметров через стек

```
.globl _start

.text
_start:
    subq $3, %rsp # резервируем для параметров 3 байта

    # помещаем в стек три числа по 1 байту
    movb $3, 2(%rsp)
    movb $4, 1(%rsp)
    movb $5, (%rsp)
    call sum

    addq $3, %rsp # восстанавливаем указатель стека

    movq %rax, %rdi # для проверки состояния rax
    # перемещаем значение в rdi

    movq $60, %rax
    syscall

sum:
    xorq %rax, %rax # обнуляем регистр
    movb 10(%rsp), %al # RAX = 2
    addb 9(%rsp), %al # RAX = RAX + 4 = 6
    addb 8(%rsp), %al # RAX = RAX + 6 = 12
    ret
```

Помним про
выравнивание на x64
системах стека по
границе 16 байт (в
некоторых - 8)

Функции в GAS. Возврат значений

```
.globl _start

.text
_start:
    # устанавливаем параметры для функции sum
    movq $5, %rdi
    movq $6, %rsi
    call sum          # после вызова в RAX – результат
    сложения
    movq %rax, %rdi      # помещаем результат в RDI

    movq $60, %rax
    syscall

# sum выполняет сложение двух чисел
# RDI – первое число
# RSI – второе число
# RAX – результат сложения
sum:
    movq %rdi, %rax
    addq %rsi, %rax
    ret
```

Функции в GAS.

Глобальные и локальные переменные(1/6)

```
.globl _start

.text
_start:
    movq $11, %rdi      # в RDI параметр для функции
sum
    call sum            # после вызова в RAX – результат
сложения
    movq %rax, %rdi    # помещаем результат в RDI

    movq $60, %rax
    syscall

sum:
    # добавляем в стек число 5 – условная безымянная
    # локальная переменная
    pushq $5            # RSP указывает на адрес числа 5
    movq %rdi, %rax     # в RAX значение параметра из RDI
    addq (%rsp), %rax   # rax = rax + (%rsp) = rax + 5
    addq $8, %rsp        # освобождаем стек
    ret
```

Функции в GAS.

Глобальные и локальные переменные (2/6)

```
.globl _start

.text
_start:
    movq $11, %rdi      # в RDI параметр для функции
sum
    call sum            # после вызова в RAX – результат
    movq %rax, %rdi    # помещаем результат в RDI
    movq $60, %rax     # RDI = 20
    syscall

sum:
    # добавляем в стек число 5 – условная безымянная
    # локальная переменная
    pushq $5           # RSP указывает на адрес числа 5
    addb $4, (%rsp)    # увеличиваем переменную на 4 – она
    # равна 9
    movq %rdi, %rax   # в RAX значение параметра из RDI
    addq (%rsp), %rax # rax = rax + (%rsp) = rax + 9
    addq $8, %rsp      # освобождаем стек
    ret
```

Функции в GAS.

Глобальные и локальные переменные (3/6)

```
.globl _start

.text
_start:
    movq $11, %rdi      # в RDI параметр для функции sum
    call sum              # после вызова в RAX – результат
    сложения
    movq %rax, %rdi      # помещаем результат в RDI

    movq $60, %rax        # RDI = 16
    syscall

sum:
    subq $8, %rsp          # резервируем для двух
    переменных в стеке 8 байт

    movl $5, 4(%rsp)       # По адресу (rsp+4) первая
    локальная переменная, которая равна 5
    movl %edi, (%rsp)       # По адресу (rsp) вторая
    локальная переменная, которая равна ECX

    movl 4(%rsp), %eax      # в EAX значение первой
    переменной
    addl (%rsp), %eax       # EAX = EAX + вторая
    переменная

    addq $8, %rsp          # освобождаем стек
    ret
```

Функции в GAS.

Глобальные и локальные переменные (4/6)

```
pushq %rbp      # сохраняем старое значение регистра RBP
movq %rsp, %rbp # помещаем указатель стека RSP в регистр RBP
subq пространство_для_локальных_переменных, %rsp

movq %rbp, %rsp # удаляем локальные переменные и
очищаем стек
popq %rbp # восстанавливаем в RBP значение для
вызывающего кода
ret       # возвращаемся в вызывающий код
```

Сокращенный эквивалент

```
enter $N_байтов, $0
```

```
leave
```

Функции в GAS.

Глобальные и локальные переменные (5/6)

```
.globl _start

.text
_start:
    movq $11, %rdi      # в RDI параметр для функции sum
    call sum             # после вызова в RAX – результат сложения
    movq %rax, %rdi      # помещаем результат в RDI

    movq $60, %rax      # RDI = 16
    syscall

sum:
    pushq %rbp          # сохраняем старое значение RBP в стек
    movq %rsp, %rbp      # копируем текущий адрес из RSP в RBP
    subq $16, %rsp        # выделяем место для двух переменных по 8 байт
    movq $7, -8(%rbp)    # По адресу -8(%rbp) первая локальная переменная, равная
    movq %rdi, -16(%rbp) # По адресу -16(%rbp) вторая локальная переменная, равная
    RDI
    movq -8(%rbp), %rax  # в RAX значение из -8(%rbp) – первая локальная
    переменная
    addq -16(%rbp), %rax # RAX = RAX + -16(%rbp) – вторая локальная переменная

    movq %rbp, %rsp      # восстанавливаем ранее сохраненное значение RSP
    popq %rbp            # восстановив RBP

    ret
```

Функции в GAS.

Глобальные и локальные переменные (6/6)

```
.globl _start

.text
_start:
    movq $11, %rdi      # в RDI параметр для функции sum
    call sum             # после вызова в RAX – результат сложения
    movq %rax, %rdi     # помещаем результат в RDI

    movq $60, %rax      # RDI = 16
    syscall

sum:
    enter $16, $0        # сохраняем значения RSP и RBP и выделяем в стеке 16 байт
    movq $7, -8(%rbp)    # По адресу -8(%rbp) первая локальная переменная, равная
7
    movq %rdi, -16(%rbp) # По адресу -16(%rbp) вторая локальная переменная, равная
RDI
    movq -8(%rbp), %rax  # в RAX значение из -8(%rbp) – первая локальная
переменная
    addq -16(%rbp), %rax # RAX = RAX + -16(%rbp) – вторая локальная переменная

    leave                # восстанавливаем ранее сохраненное значение RSP и RBP

    ret
```

Функции в Apple ARM64

```
.global _main
.align 2

_main:
bl sum

mov x0, 1
mov X16, #1
svc #0x80

// определение функции sum
sum:
mov x0, #5
add x0, x0, #7
ret
```

Функции в Apple ARM64

Передача и возврат параметров

```
.global _main
.align 2

_main:
    mov x0, #5          // первый параметр
    mov x1, #6          // второй параметр
    bl sum              // вызываем функцию sum

    mov x16, #1
    svc #0x80

    // функция sum сладывает два числа
    // параметры:
    // x0 - первое число
    // x1 - второе число
    // x0 - возврат
sum:
    add x0, x0, x1
    ret
```

Функции в Apple ARM64

Работа со стеком

```
.global _main
.align 2

_main:
    mov x0, #10
    bl sum
    mov x16, #1
    svc #0x80

sum:
    sub sp, sp, #16      // выделяем место в стеке для хранения параметров и локальной переменной
    str x0, [sp, #8]     // сохраняем в стек значение параметра из регистра X0
    mov x0, #13          // условная локальная переменная
    str x0, [sp]          // сохраняем в стек значение локальной переменной

    // здесь могут быть действия, которые изменяют X0 и X1

    ldr x2, [sp, #8]      // восстанавливаем значение параметра
    ldr x3, [sp]          // восстанавливаем значение локальной переменной
    add x0, x2, x3        // основная работа – вычисление суммы

    add sp, sp, #16        // очищаем фрейм стека на 16 байт
    ret
```

Функции в Apple ARM64

Работа с фреймом стека (1/4)

```
sub sp, sp, #32      // перемещаем указатель стека на эту свободную область в стеке
mov fp, sp          // помещаем в регистр fp текущий адрес, на который указывает sp

str x0, [fp]          // сохраняем в стек значение из регистра x0
str x1, [fp, #-8]     // сохраняем в стек значение из регистра x1
str x2, [fp, #-16]    // сохраняем в стек значение из регистра x2
str x3, [fp, #-24]    // сохраняем в стек значение из регистра x3
// .....
ldr x3, [fp, #-24]    // извлекаем из стека значение в регистр x3
ldr x2, [fp, #-16]    // извлекаем из стека значение в регистр x2
ldr x1, [fp, #-8]     // извлекаем из стека значение в регистр x1
ldr x0, [fp]          // извлекаем из стека значение в регистр x0
```

Функции в Apple ARM64

Работа с фреймом стека (2/4)

```
.global _main
.align 2

_main:
    mov x0, #10
    bl sum
    mov x16, #1
    svc #0x80

sum:
    sub sp, sp, #16      // выделяем место в стеке для хранения параметров и локальной
переменной
    mov fp, sp            // указатель фрейма FP указывает на тот же адрес, что и SP
    str x0, [fp, #8]       // сохраняем в стек значение параметра из регистра X0
    mov x0, #13            // условная локальная переменная
    str x0, [fp]           // сохраняем в стек значение локальной переменной

// здесь могут быть действия, которые изменяют X0 и X1

    ldr x2, [fp, #8]       // восстанавливаем значение параметра
    ldr x3, [fp]             // восстанавливаем значение локальной переменной
    add x0, x2, x3          // основная работа – вычисление суммы

    add sp, sp, #16          // очищаем фрейм стека на 16 байт
    ret
```

Функции в Apple ARM64

Работа с фреймом стека (3/4)

```
.global _main
.align 2
_main:
    mov x0, #10
    bl start
    mov x16, #1
    svc #0x80

start:
    str lr, [sp, #-16]! // сохраняем регистр LR/X30 в стек
    sub sp, sp, #16      // выделяем место в стеке для локальных переменных
    mov fp, sp           // устанавливаем указатель фрейма FP
    mov x0, #10          // первая локальная переменная
    str x0, [fp, #8]     // сохраняем ее во фрейме стека
    mov x0, #15          // вторая локальная переменная
    str x0, [fp]         // сохраняем ее во фрейме стека

    // извлекаем локальные переменные и передаем их в качестве параметров
    ldr x0, [fp, #8]     // первый параметр
    ldr x1, [fp]          // второй параметр
    bl sum
    add sp, sp, #16      // очищаем место в стеке
    ldr lr, [sp], #16    // восстанавливаем регистр X30/LR
    ret

sum:
    str fp, [sp, #-16]! // сохраняем регистр FP/X29 в стек
    sub sp, sp, #16      // выделяем 16 байт в стеке
    mov fp, sp           // указатель фрейма FP указывает на SP
    str x0, [fp, #8]     // сохраняем во фрейм стека первый параметр из регистра X0
    str x1, [fp]          // сохраняем во фрейм стека второй параметр из регистра X1

    // здесь могут быть действия, которые изменяют X0 и X1

    ldr x2, [fp, #8]     // восстанавливаем значение первого параметра
    ldr x3, [fp]          // восстанавливаем значение второго параметра
    add x0, x2, x3        // основная вычисление суммы

    add sp, sp, #16      // очищаем фрейм стека на 16 байт
    ldr fp, [sp], #16    // восстанавливаем регистр X29/FP
    ret
```

Функции в Apple ARM64

Работа с фреймом стека (4/4)

```
.global _main
.align 2

_main:
    mov x0, #10
    bl start
    mov x16, #1
    svc #0x80

start:
    stp x29, x30, [sp, #-32]! // выделяем 32 байта в стеке и сохраняем регистры FP/X29 и LR/X30
    add fp, sp, #16           // устанавливаем указатель фрейма FP на SP+16
    mov x0, #10                // первая локальная переменная
    str x0, [fp, #8]           // сохраняем ее во фрейме стека
    mov x0, #15                // вторая локальная переменная
    str x0, [fp]                // сохраняем ее во фрейме стека

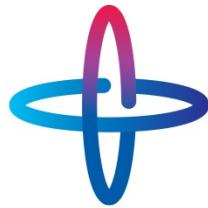
    // извлекаем локальные переменные и передаем их в качестве параметров
    ldr x0, [fp, #8]           // первый параметр
    ldr x1, [fp]                // второй параметр
    bl sum
    ldp x29, x30, [sp], #32     // восстанавливаем регистры X29/FP и X30/LR и очищаем стек
    ret

sum:
    stp x29, x30, [sp, #-32]! // выделяем 32 байта в стеке и сохраняем регистры FP/X29 и LR/X30
    add fp, sp, #16           // устанавливаем указатель фрейма FP на SP+16
    str x0, [fp, #8]           // сохраняем во фрейм стека первый параметр из регистра X0
    str x1, [fp]                // сохраняем во фрейм стека второй параметр из регистра X1

    // здесь могут быть действия, которые изменяют X0 и X1

    ldr x2, [fp, #8]           // восстанавливаем значение первого параметра
    ldr x3, [fp]                // восстанавливаем значение второго параметра
    add x0, x2, x3              // основная вычисление суммы

    ldp x29, x30, [sp], #32     // восстанавливаем регистры X29/FP и X30/LR и очищаем стек
    ret
```



гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru

Что с зачетом?

Классический зачёт у нас у групп: 1241, 1242, 1243 и 7241

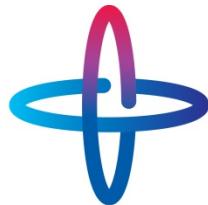
Дифференцируемый зачет - 1245

Чтобы получить зачёт надо:

1. Сдать все отчеты и получить по ним «принято»
2. Защитить все лабораторные
3. Суммарно за семестр набрать **21 балл**
 - Для **1245**: 21-25 баллов - 3-ка, 26-31 - 4-ка, 32 - 35 - 5-ка

Проставление зачета:

1. Список тех, кто уже получил/не получил зачёт будет отправлен в групповой чат
2. Для **1245** вышлю список с предварительными оценками
3. Те, кто со всем согласен - проставлю зачет/оценку сам, **приходить на зачетной неделе не нужно!**
4. Кому не хватило баллов, кто хочет оценку повыше, у кого пока нет зачёта:
 - На зачетной неделе прийти сдать долги, выполнить очно 6 ЛР или получить вопрос по курсу (все зависит от решения поставленной задачи на итоговую оценку/зачет). Иметь средства для выполнения задания на ассемблере



гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru

Время встреч на зачетной неделе:

1241: 27.05.2024 с 12:00 - 14:00, БМ, 52-33(а) 3 этаж

1242: 28.05.2024 с 12:00 - 14:00, БМ, 52-33(а) 3 этаж

1243: 28.05.2024 с 14:00 - 16:00, БМ, 52-33(а) 3 этаж

1245: 29.05.2024 с 12:00 - 14:00, БМ, 52-33(а) 3 этаж

7241: 30.05.2024 с 12:00 - 14:00, БМ, 52-33(а) 3 этаж



гуп

Государственный университет
аэрокосмического приборостроения

www.guap.ru



ФСЁ