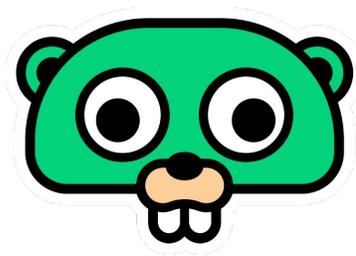


Понимание ассемблера

Го

Панасюк Игорь

Бэкенд-разработчик
Яндекс Финтех



GoFunc

О чем поговорим



1. `$go:string."`Зачем?"`[sv]`



О чем поговорим

1. `$go:string."Зачем?"[sv]`
2. `$go:string."Предыстория"[sv]`



О чем поговорим

1. `$go:string."`Зачем?"`][sv]`
2. `$go:string."`Предыстория"`][sv]`
3. `$go:string."`Умные слова"`][sv]`



О чем поговорим

1. `$go:string."`**Зачем?**`"[sv]`
2. `$go:string."`**Предыстория**`"[sv]`
3. `$go:string."`**Умные слова**`"[sv]`
4. `$go:string."`**Синтаксис**`"[sv]`



О чем поговорим

1. `$go:string."`**Зачем?**`"[sv]`
2. `$go:string."`**Предыстория**`"[sv]`
3. `$go:string."`**Умные слова**`"[sv]`
4. `$go:string."`**Синтаксис**`"[sv]`
5. `$go:string."`**Примеры**`"[sv]`



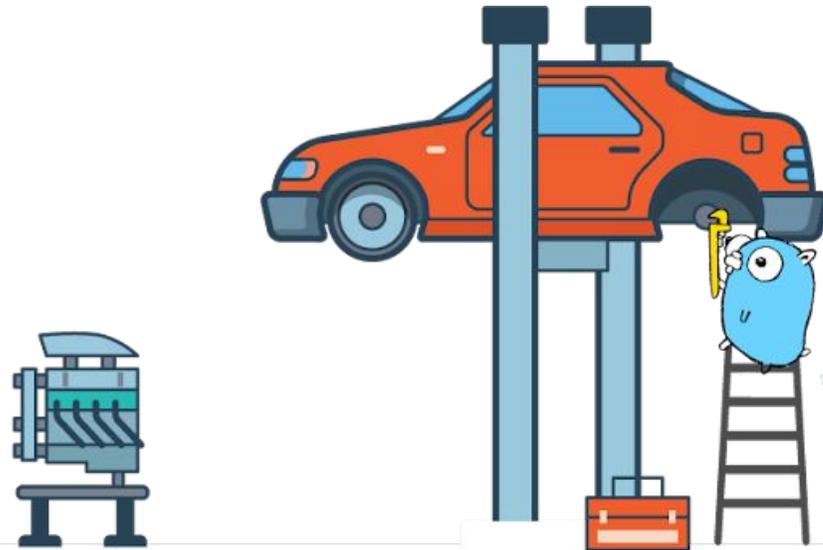
О чем поговорим

1. `$go:string."Зачем?"[SB]`
2. `$go:string."Предыстория"[SB]`
3. `$go:string."Умные слова"[SB]`
4. `$go:string."Синтаксис"[SB]`
5. `$go:string."Примеры"[SB]`
6. `go:string."Выводы" SRDDATA dupok size=12`

Wikipedia



Ассемблер — это **низкоуровневый язык** программирования, который представляет собой промежуточное звено между **машинным кодом** и **высокоуровневыми** языками программирования



07;

Зачем нам вообще этот ваш ассемблер

Мотивация



```
func SliceContainsV0(s []uint8, target uint8) bool {  
    return slices.Contains(s, target)  
}
```

Мотивация



```
func SliceContainsV0(s []uint8, target uint8) bool {  
    return slices.Contains(s, target)  
}
```

```
func SliceContainsV1(s []uint8, target uint8) bool
```

МОТИВАЦИЯ



```
func SliceContainsV0(s []uint8, target uint8) bool {  
    return slices.Contains(s, target)  
}
```

```
func SliceContainsV1(s []uint8, target uint8) bool
```

```
go test -bench=. -benchmem -cpu=1  
goos: darwin  
goarch: arm64  
pkg: asm/simd/slice_contains  
cpu: Apple M3 Pro  
BenchmarkSliceContains/SliceContainsV1      63768      18330 ns/op      0 B/op      0 allocs/op  
BenchmarkSliceContains/SliceContainsV0      4381      256653 ns/op     0 B/op      0 allocs/op
```

Мотивация



<u>DEG</u> · RAD		256653 / 18330 = 14,0018003273				
14,0018003273						
x^y	$x!$	\pm	C	()	%	\div
asin	sin	1/x	7	8	9	\times
acos	cos	$\sqrt{\quad}$	4	5	6	-
atan	tan	ln	1	2	3	+
lg	π	e	0		,	=



Rob Pike



Also, perhaps most important: it is how we talk about the machine. Knowing assembly, even a little, means understanding computers better.



02;

Предыстория

Или что за зверь такой План в

Предыстория



Bell Labs

- Plan 9 operation system

The screenshot displays the Plan 9 operating system interface. At the top, it shows the date and time: "Sun Jun 11 12:45". Below this is a row of user avatars and names: rsc, /dev/n, jmk, skipt, tklopp, john, brucee, and lorenz. To the right, there are three windows titled "olive(4)", "anna", and "achille", each showing a red waveform. The main window is a mail client displaying a message from "New Cut Paste Snarf Sort Zerox Delcol" with a subject line "New Cut Paste Snarf Sort Zerox Delcol". The message content includes a file named "body.jpg" and a list of recipients. Below the mail window, there is a terminal window showing the text "Plan 9 from Bell Labs" and a list of words: "3-6 augrim, 4 -ym, 5 -ime, -ime, 7 agrum, algrim. β. 4-6 jarosme, augorisme(e, augrisme, 7-9 algorism, algorithm. Ia. OFr. augorisme, algorisme, augorime; ad. med.L. algorism-us (cf. Sp. guarismo cipher), f. Arab. al-Khowārizmī the native of Khwārazm (Khiva), surname of the Arab mathematician Abu Ja'far Mohammed Ben Musa, who flourished early in the 9th c., and through the translation of whose work on Algebra, the Arabic numerals became generally known in Europe. (Cf. 'Euclid' = plane geometry.) Algorisme being popularly reduced in OFr. to augorime, English also shows two forms, the popular augrime, ending in agrim, agrum, and the learned algorism which passed".

Предыстория



Bell Labs

- Plan 9 operation system
- Plan 9 assembler

The screenshot displays a Plan 9 desktop environment. At the top, a status bar shows the date and time: "Sun Jun 11 12:45". Below this is a row of icons for various users: rsc, jmk, skipt, tklopp, john, brucee, and lorenz. To the right, there are three windows titled "olive(4)", "anna", and "achille", each showing a red waveform plot. The main window is a mail client interface with a header "Mail Newcol Kill Putall Dump Exit" and a list of messages. A message is selected, showing its content: "New Cut Paste Snarf Sort Zerox Delcol" and a list of recipients. Below the message list, there is a large image of a white bunny, labeled "Plan 9 from Bell Labs". To the right of the bunny image, there is a window titled "3-6 augrim, 4 -ym, 5 -ime, -ime, 7 agrum, algrim. β. 4-6" which contains text about the word "algorism". At the bottom right, there is a window showing a starry space scene. The desktop background is black with a grid of small white dots.

Go ассемблер



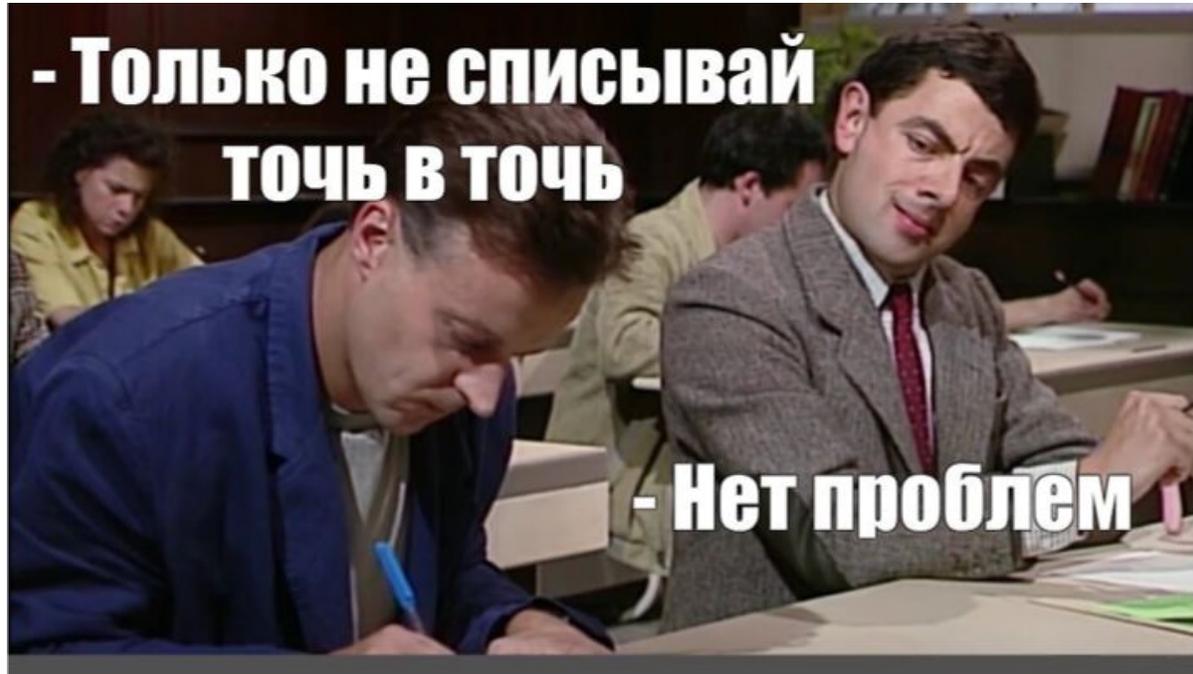
- На первой же странице документации ссылается на Plan 9

A Manual for the Plan 9 assembler

Rob Pike

rob@plan9.bell-labs.com

Plan 9 and Go assemblers



```
// go assembler
TEXT sum(SB), NOSPLIT, $0
    MOVD first+0(FP), R0
    MOVD second+8(FP), R1
    ADD R0, R1
    MOVD R1, ret+16(FP)
    RET
```

```
// Plan 9 assembler
TEXT sum(SB), $0
    MOVL    arg1+0(FP), R0
    ADDL    arg2+4(FP), R0
    RTS
```

Стандартная библиотека



- `math`
- `crypto`
- `reflect`
- `runtime`
- `sync`
- `syscall`
- `internal packages`

Предыстория



Go assembler

- Базируется на Plan 9 assembler

Предыстория



Go assembler

- Базируется на Plan 9 assembler
- Является частью Go toolchain

Предыстория



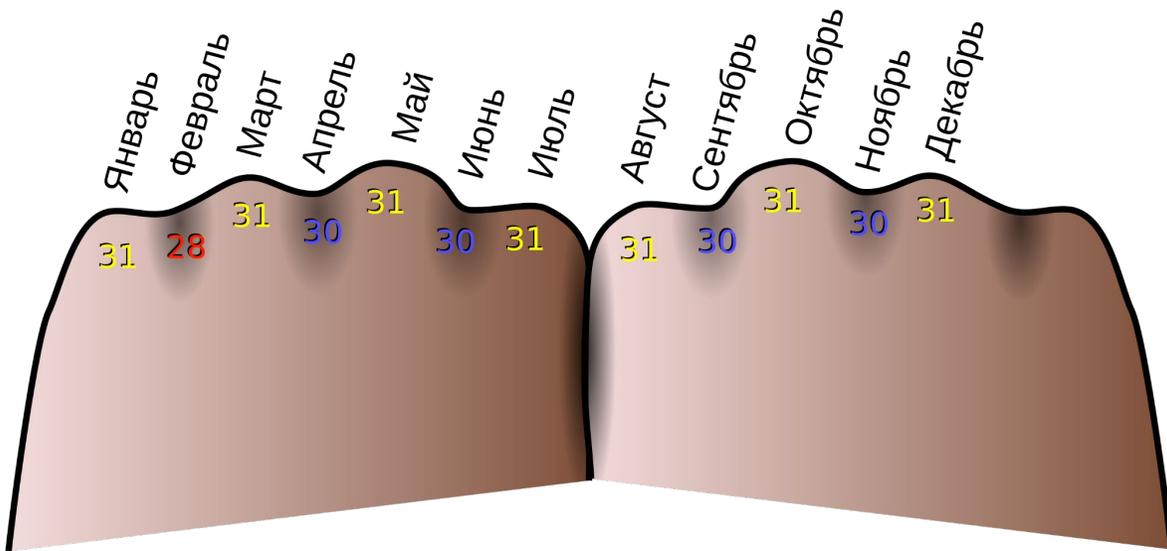
Go assembler

- Базируется на Plan 9 assembler
- Является частью Go toolchain
- Изначально был добавлен в язык для рантайма

03;

Умные слова

Мнемоника



Способ легче запоминать информацию

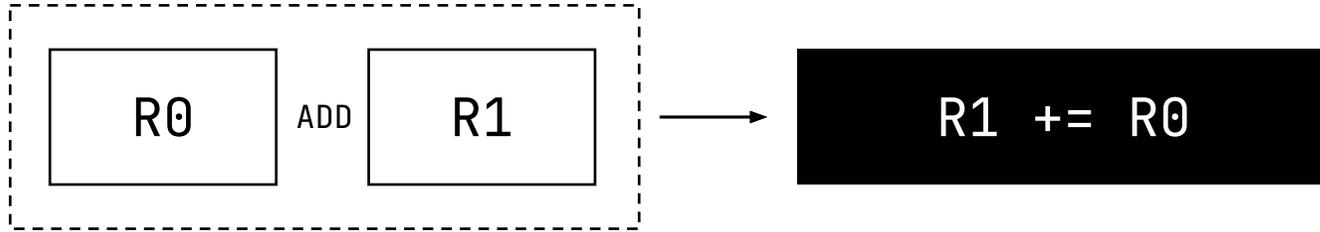
CALL - 0xf

MOVVD - 0x123

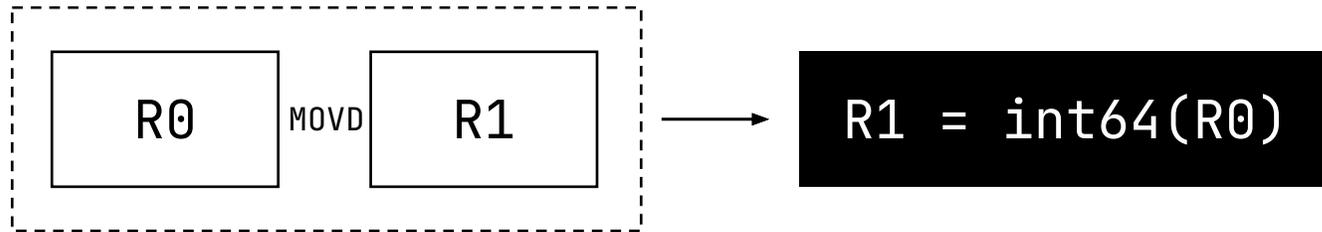
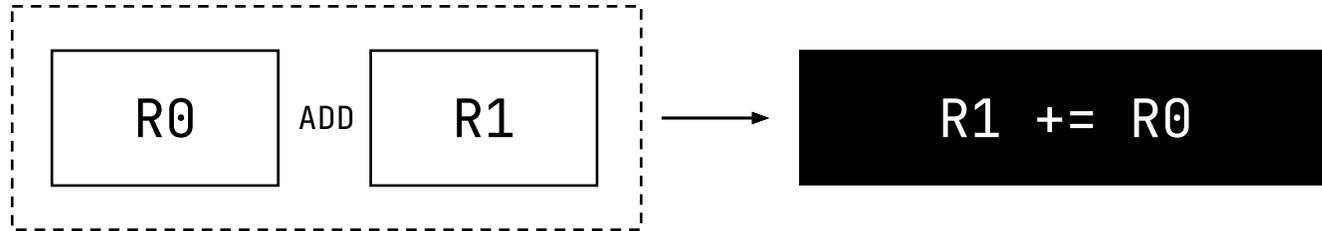
ADD - 0x777

RET - 0x444

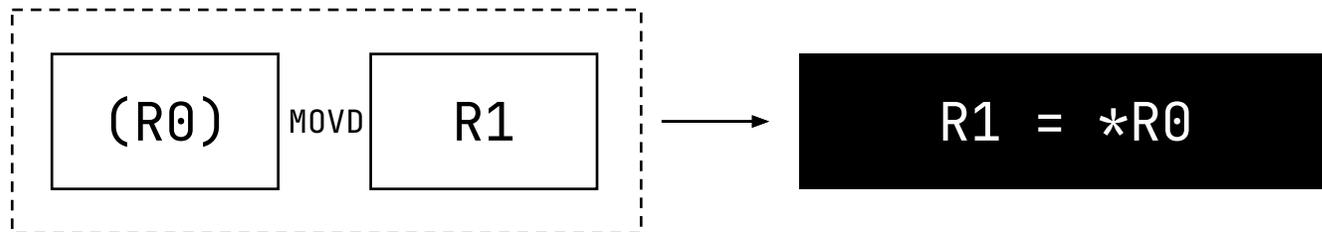
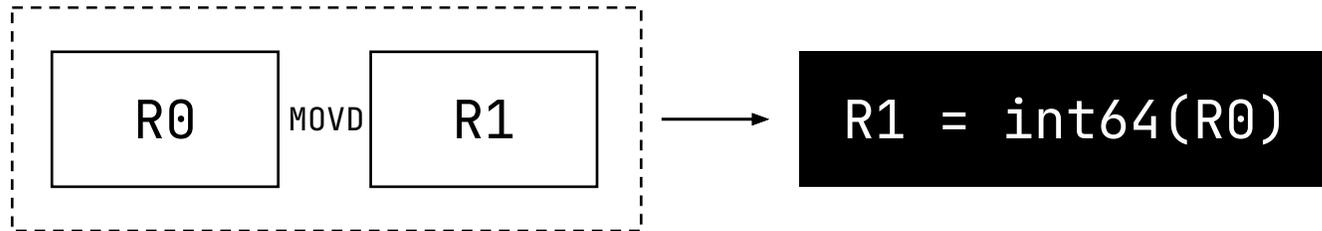
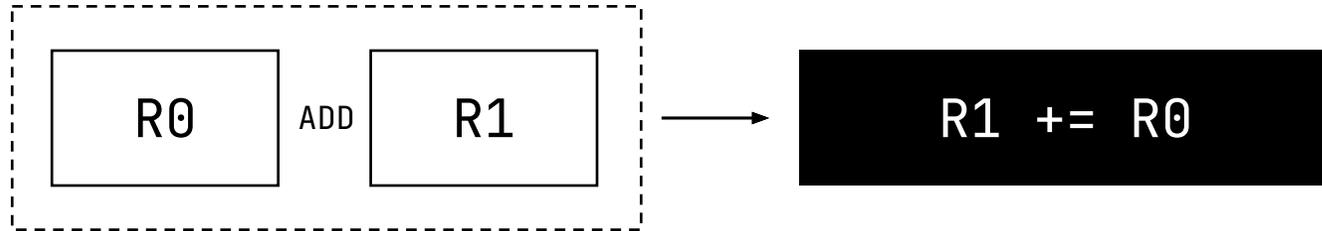
Регистр и память



Регистр и память



Регистр и память



Архитектура



```
func main() {  
    println(runtime.GOOS) // darwin  
    println(runtime.GOARCH) // arm64  
}
```



Архитектура

```
func main() {  
    println(runtime.GOOS) // darwin  
    println(runtime.GOARCH) // arm64  
}
```

GOOS: aix, android, darwin, dragonfly, freebsd, illumos, ios, js, linux, netbsd, openbsd, wasip1, plan9, solaris, wasip1, windows...

GOARCH: 386, amd64, arm, arm64, loong64, mips, mips64, mips64le, mipsle, ppc64, ppc64le, riscv64, s390x, wasm...

Машинное слово



В вычислительной и иной программируемой технике машинным словом называется **единица данных**, которая выбрана **естественной** для данной **архитектуры** процессора

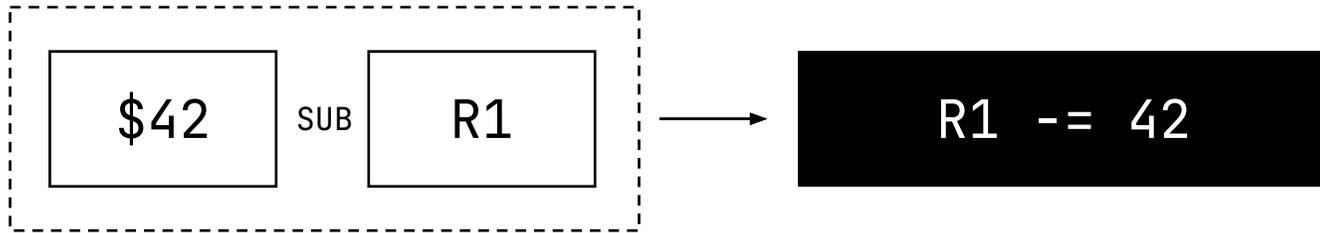
Машинное слово



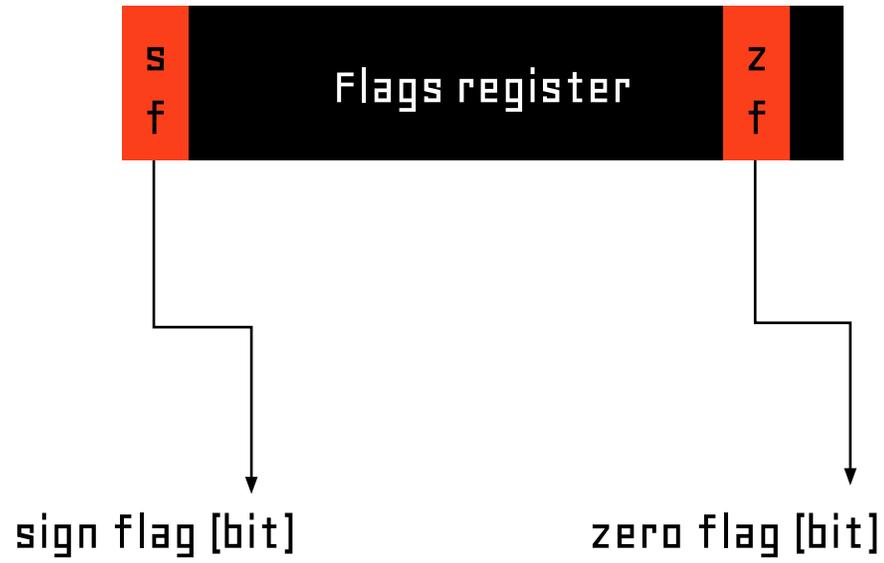
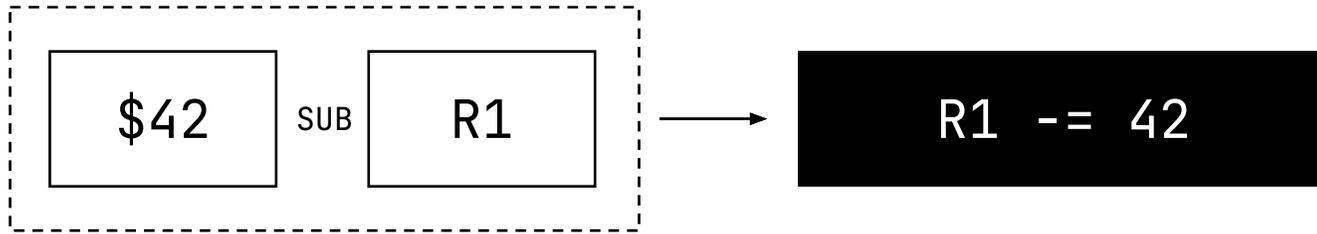
В вычислительной и иной программируемой технике машинным словом называется **единица данных**, которая выбрана **естественной** для данной **архитектуры** процессора

- `arm64` - 4 bytes
- `amd64 [x86-64]` - 2 bytes

Flags register



Flags register



Program counter

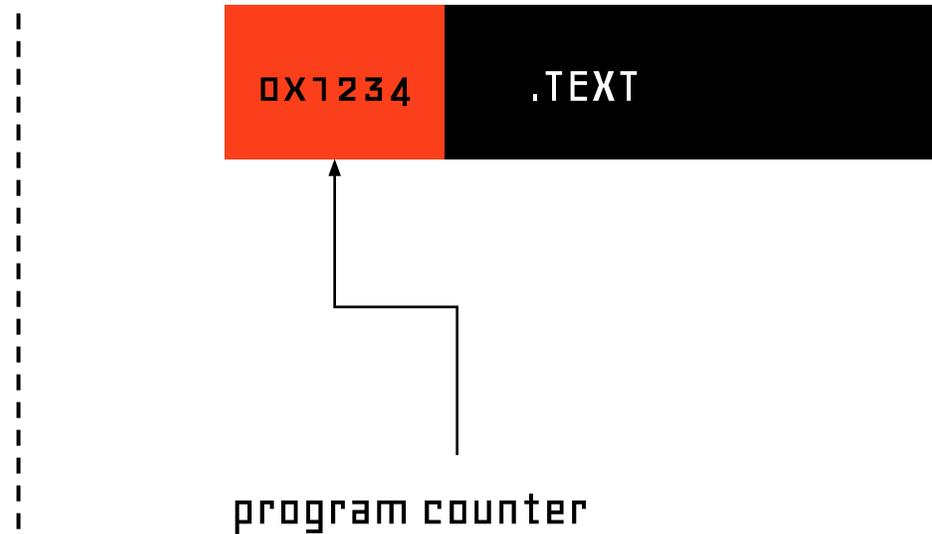


PC = program counter [IP = instruction pointer, IAR = instruction address register] — **регистр** процессора, который **указывает**, какую **команду** нужно выполнять следующей

Program counter



PC = program counter [IP = instruction pointer, IAR = instruction address register] — **регистр** процессора, который **указывает**, какую **команду** нужно выполнять следующей



□ 4;

Синтаксис

Различие в мнемониках



```
// arm64
TEXT ·sum(SB), NOSPLIT, $0
// MOVD - MOV double word
MOVD first+0(FP), R0
MOVD second+8(FP), R1
ADD R0, R1
MOVD R1, ret+16(FP)
RET
```

```
// amd64
TEXT ·sum(SB), NOSPLIT, $0
// MOVQ - MOV quad word
MOVQ first+0(FP), AX
MOVQ second+8(FP), DX
ADDQ AX, DX
MOVQ DX, ret+16(FP)
RET
```

Различие в мнемониках



```
// arm64
TEXT ·sum(SB), NOSPLIT, $0
// MOVD - MOV double word
MOVD first+0(FP), R0
MOVD second+8(FP), R1
ADD R0, R1
MOVD R1, ret+16(FP)
RET
```

```
// amd64
TEXT ·sum(SB), NOSPLIT, $0
// MOVQ - MOV quad word
MOVQ first+0(FP), AX
MOVQ second+8(FP), DX
ADDQ AX, DX
MOVQ DX, ret+16(FP)
RET
```

Различие в мнемониках



```
// arm64
TEXT ·sum(SB), NOSPLIT, $0
    // MOVD - MOV double word
    MOVD first+0(FP), R0
    MOVD second+8(FP), R1
    ADD R0, R1
    MOVD R1, ret+16(FP)
    RET
```

```
// amd64
TEXT ·sum(SB), NOSPLIT, $0
    // MOVQ - MOV quad word
    MOVQ first+0(FP), AX
    MOVQ second+8(FP), DX
    ADDQ AX, DX
    MOVQ DX, ret+16(FP)
    RET
```



Арифметические и логические операции

```
ADD R0, R1 // R1 += R0
SUB R0, R1, R7 // R7 = R1 - R0
MUL $42, R0 // R0 *= 42
LSR R0, R4, R7 // R7 = R4 >> R0
MVN R6, R6 // R6 = ^R6
AND $1, R6 // R6 &= 1
```



Арифметические и логические операции

```
ADD R0, R1 // R1 += R0
SUB R0, R1, R7 // R7 = R1 - R0
MUL $42, R0 // R0 *= 42
LSR R0, R4, R7 // R7 = R4 >> R0
MVN R6, R6 // R6 = ^R6
AND $1, R6 // R6 &= 1
```



Арифметические и логические операции

```
ADD R0, R1      // R1 += R0
SUB R0, R1, R7  // R7 = R1 - R0
MUL $42, R0    // R0 *= 42
LSR R0, R4, R7  // R7 = R4 >> R0
MVN R6, R6     // R6 = ^R6
AND $1, R6     // R6 &= 1
```



Арифметические и логические операции

```
ADD R0, R1      // R1 += R0
SUB R0, R1, R7  // R7 = R1 - R0
MUL $42, R0     // R0 *= 42
LSR R0, R4, R7  // R7 = R4 >> R0
MVN R6, R6     // R6 = ^R6
AND $1, R6     // R6 &= 1
```



Арифметические и логические операции

```
ADD R0, R1      // R1 += R0
SUB R0, R1, R7  // R7 = R1 - R0
MUL $42, R0     // R0 *= 42
LSR R0, R4, R7  // R7 = R4 >> R0
MVN R6, R6     // R6 = ^R6
AND $1, R6     // R6 &= 1
```



Арифметические и логические операции

```
ADD R0, R1      // R1 += R0
SUB R0, R1, R7  // R7 = R1 - R0
MUL $42, R0     // R0 *= 42
LSR R0, R4, R7  // R7 = R4 >> R0
MVN R6, R6      // R6 = ^R6
AND $1, R6      // R6 &= 1
```



Инструкции для работы с памятью

```
MOVD R0, R1 // R1 = R0 (64-bit operation)
MOVW R0, R1 // R1 = R0 (32-bit operation)
MOVD (R0), R1 // R1 = *R0
MOVD.P 16(R0), R1 // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1 // R0 -= 16; R1 = *R0
MOVD (R0), (R1) // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1   // R0 -= 16; R1 = *R0
MOVD (R0), (R1)      // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
```

```
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // +R0 = +R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```

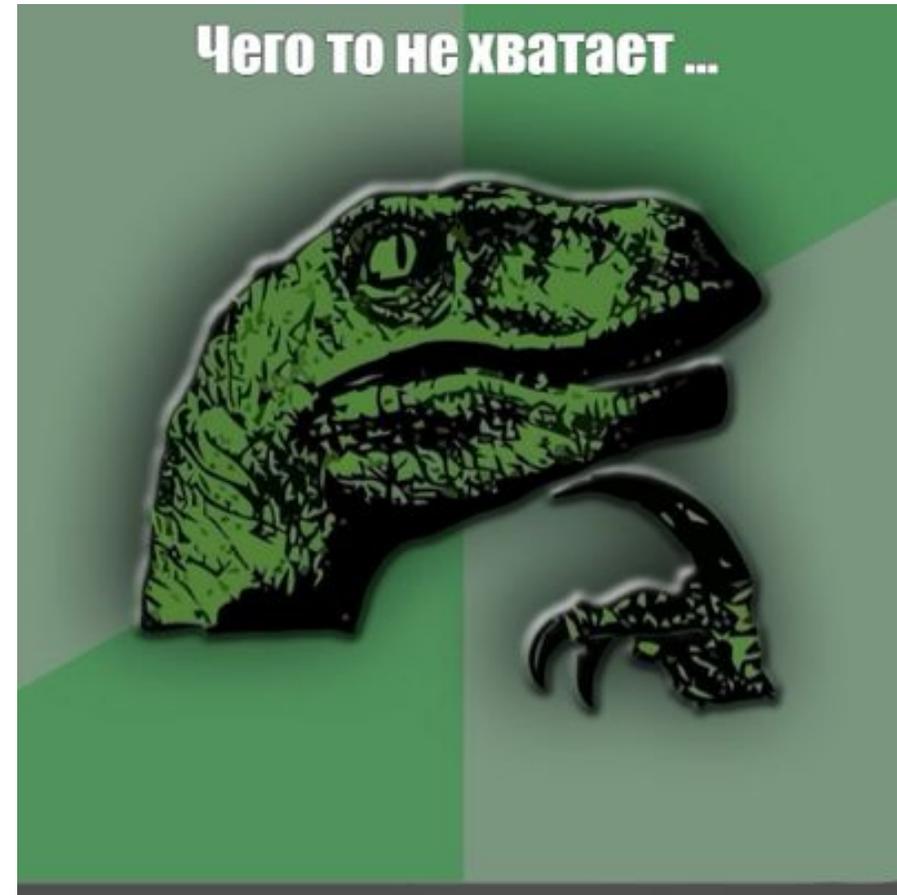
Инструкции для работы с памятью

```
MOVD R0, R1           // R1 = R0 (64-bit operation)
MOVW R0, R1           // R1 = R0 (32-bit operation)
MOVD (R0), R1         // R1 = *R0
MOVD.P 16(R0), R1     // R1 = *R0; R0 += 16
MOVD.W -16(R0), R1    // R0 -= 16; R1 = *R0
MOVD (R0), (R1)       // *R0 = *R1 - INVALID
```

```
LDP a+0(FP), (R0, R1)
```

```
// if R5 ≤ R2 {
//     R3 = R13
// }
// if R5 > R2 {
//     R1 = R13
// }
```

```
CSEL LE, R13, R3, R3
CSEL GT, R13, R1, R1
```



Инструкции управления



Labels [goto]

```
CMP $1, R4 // Branch less equal (LE)  
BLE L0 // if R4 ≤ 1 { L0 code }  
B L1
```

```
L0:  
// branch
```

```
L1:  
// another branch
```

Инструкции управления



Labels [goto]

```
CMP $1, R4 // Branch less equal (LE)
BLE L0 // if R4 ≤ 1 { L0 code }
B L1
```

L0:

// branch

L1:

// another branch

Инструкции управления



Labels [goto]

```
CMP $1, R4 // Branch less equal (LE)
BLE L0     // if R4 ≤ 1 { L0 code }
B L1
```

L0:

```
// branch
```

L1:

```
// another branch
```

Инструкции управления



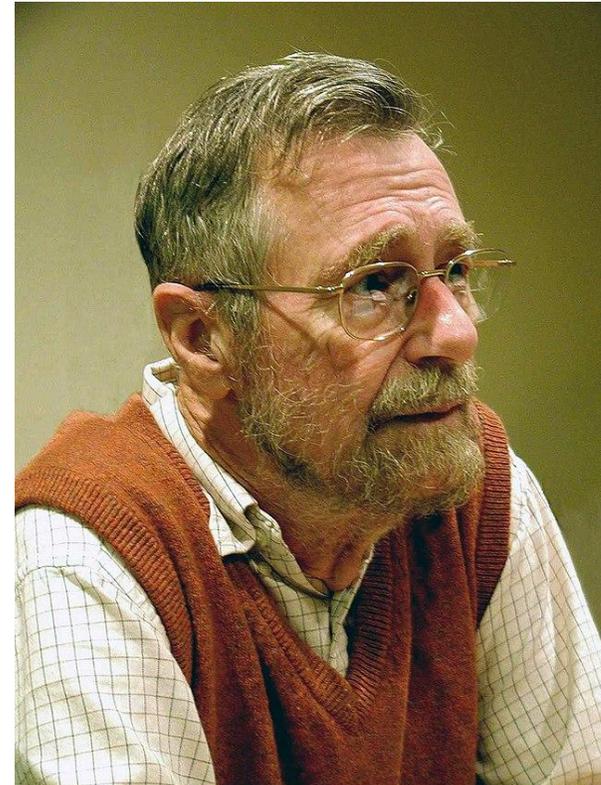
Labels [goto]

```
CMP $1, R4 // Branch less equal (LE)
BLE L0 // if R4 ≤ 1 { L0 code }
B L1
```

```
L0: // branch
```

```
L1: // another branch
```

Edgar Dijkstra: Go To Statement Considered Harmful



Инструкции управления



Program counter

```
// cycle
// for {
//     R0 += 1
//     R0 *= 2
// }
//
test:
    ADD $1, R0
    MUL $2, R0
    B -2(PC)
```

Глобальные переменные



```
DATA data◇+0(SB)/20,$"Hello, GoFunc-2024!\n"
```

```
GLOBAL data◇(SB),NOPTR,$20
```

```
// inside function
```

```
MOVD $text◇+0(SB), R0
```



Глобальные переменные



```
DATA data◇+0(SB)/20,$"Hello, GoFunc-2024!\n"  
GLOBAL data◇(SB),NOPTR,$20
```

```
// inside function  
MOVD $text◇+0(SB), R0
```



Глобальные переменные



```
DATA data◇+0(SB)/20,$"Hello, GoFunc-2024!\n"  
GLOBL data◇(SB),NOPTR,$20
```

```
// inside function  
MOVD $text◇+0(SB), R0
```



Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len  int
//   Cap  int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT -SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0-24
...
```

Объявление функций



```
// slice_sum.go file
func SumSlice(x []int32) int64

// slice_sum_arm64.s file
// Include standard directives
#include "textflag.h"

// type SliceHeader struct {
//   Data uintptr
//   Len   int
//   Cap   int
//}

// NOSPLIT - Don't insert the preamble to check if the stack must be split
// $0-24 (0 - stack frame size, 24 - arguments size), "-24" is optional here
TEXT ·SumSlice(SB), NOSPLIT, $0
...
```

Синтаксис



Go assembler

- Платформозависимый, за исключением нескольких мнемоник [RET, TEXT, CALL...]

Синтаксис



Go assembler

- Платформозависимый, за исключением нескольких мнемоник [RET, TEXT, CALL...]
- Нестандартное название регистров и порядок аргументов

Синтаксис



Go assembler

- Платформозависимый, за исключением нескольких мнемоник [RET, TEXT, CALL...]
- Нестандартное название регистров и порядок аргументов
- Нестандартные разделители в сигнатуре функции

Синтаксис



Go assembler

- Платформозависимый, за исключением нескольких мнемоник [RET, TEXT, CALL...]
- Нестандартное название регистров и порядок аргументов
- Нестандартные разделители в сигнатуре функции
- Отсутствие некоторых инструкций из коробки

Синтаксис



Виртуальные регистры

- PC [program counter]

Синтаксис



Виртуальные регистры

- PC [program counter]
- SB [static base pointer]
 - нужен для обращения к глобальным переменным и функциям

Синтаксис



Виртуальные регистры

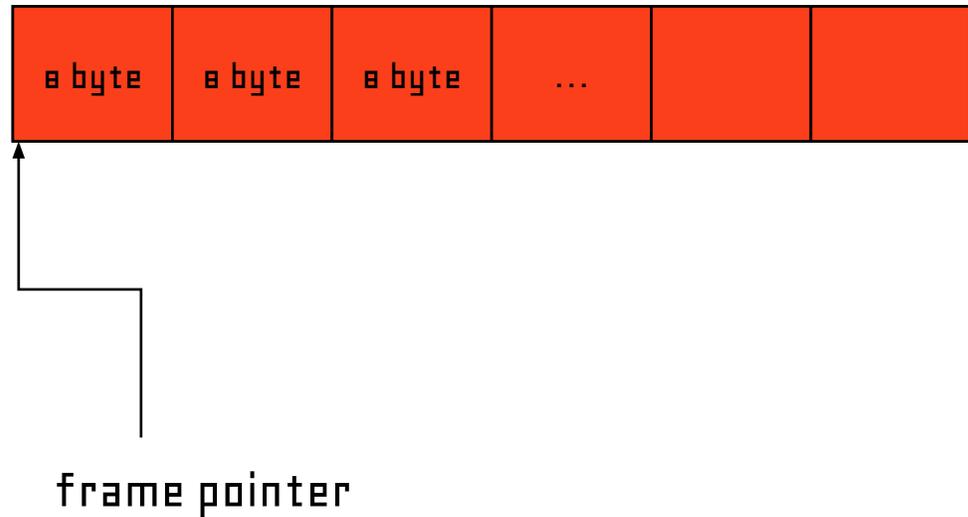
- PC [program counter]
- SB [static base pointer]
 - нужен для обращения к глобальным переменным и функциям
- SP [stack pointer]
 - указывает на конец памяти, выделенной под стек
 - по семантике взаимодействия похож на frame pointer

Виртуальные регистры



Frame pointer

- Указывает на начало списка аргументов
- Абстракция для программистов

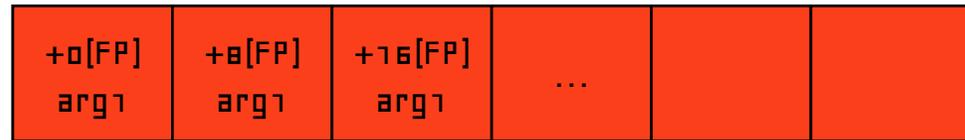


Виртуальные регистры



Frame pointer

- Обращаемся к аргументам, используя положительное смещение
 - $a+0[FP]$ – первый аргумент функции [v byte]
 - $a+v[FP]$ – второй аргумент функции



frame pointer

Виртуальные регистры

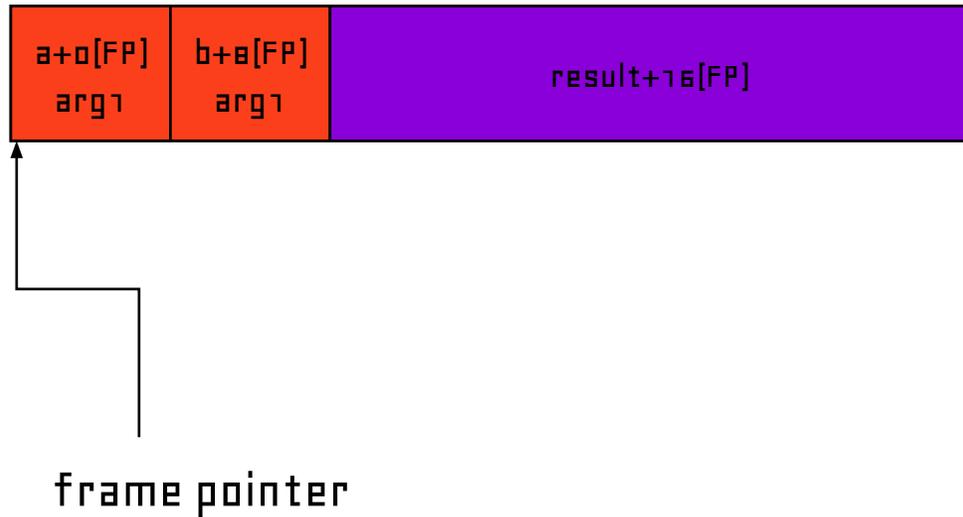


- Обязательный нейминг аргументов
 - lolkek+0[FP] – *valid*
 - +16[FP] – *invalid*

Виртуальные регистры



- Результат функции должен быть размещен после аргументов



Соглашение о вызовах



ABI stable

- Стабильный ABI для разработчиков

Соглашение о вызовах



ABI stable

- Стабильный ABI для разработчиков

ABI internal

- Внутренний ABI компилятора [`go tool compile -S -S`]

Синтаксис



Директивы компилятора

- NOSPLIT
- ABIInternal
- NOPTR [for DATA and GLOBAL]
- NEEDCTXT [for closures]

...

```
#include "textflag.h"
```

05;

Примеры

Или откуда можно выключать x_2 воспроизведение

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r
```

```
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0
```

```
    LDP slice_base+0(FP), (R0, R1)  
    ZERO(R2)
```

loop:

```
    CBZ R1, done  
    MOVW (R0), R9  
    ADD R9, R2  
    ADD $4, R0  
    SUB $1, R1  
    B loop
```

```
// Header слайса 24 байта  
// R0 - data pointer, R1 - length
```

```
// for R1  $\neq$  0 {}  
// R9 = s[i]  
// R2 += R9  
// R0++  
// R1--
```

done:

```
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                   // R9 = s[i]  
    ADD R9, R2                       // R2 += R9  
    ADD $4, R0                       // R0++  
    SUB $1, R1                       // R1--  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)  
    ZERO(R2)  
  
loop:  
    CBZ R1, done  
    MOVW (R0), R9  
    ADD R9, R2  
    ADD $4, R0  
    SUB $1, R1  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

// Header слайса 24 байта
// R0 - data pointer, R1 - length

// for R1 \neq 0 {}
// R9 = s[i]
// R2 += R9
// R0++
// R1--

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                   // R9 = s[i]  
    ADD R9, R2                       // R2 += R9  
    ADD $4, R0                       // R0++  
    SUB $1, R1                       // R1--  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                   // R9 = s[i]  
    ADD R9, R2                      // R2 += R9  
    ADD $4, R0                      // R0++  
    SUB $1, R1                      // R1--  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                    // R9 = s[i]  
    ADD R9, R2                       // R2 += R9  
    ADD $4, R0                       // R0++  
    SUB $1, R1                       // R1--  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
Loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                   // R9 = s[i]  
    ADD R9, R2                       // R2 += R9  
    ADD $4, R0                       // R0++  
    SUB $1, R1                       // R1--  
    B Loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice sum



```
#define ZERO(r) \  
    MOVD $0, r  
  
// func SumSlice(s []int32) int64  
TEXT ·SumSlice(SB), NOSPLIT, $0  
  
    LDP slice_base+0(FP), (R0, R1)    // Header слайса 24 байта  
    ZERO(R2)                        // R0 - data pointer, R1 - length  
  
loop:  
    CBZ R1, done                    // for R1 ≠ 0 {}  
    MOVW (R0), R9                    // R9 = s[i]  
    ADD R9, R2                        // R2 += R9  
    ADD $4, R0                        // R0++  
    SUB $1, R1                        // R1--  
    B loop  
  
done:  
    MOVD R2, ret+24(FP)  
    RET
```

Slice contains in Go



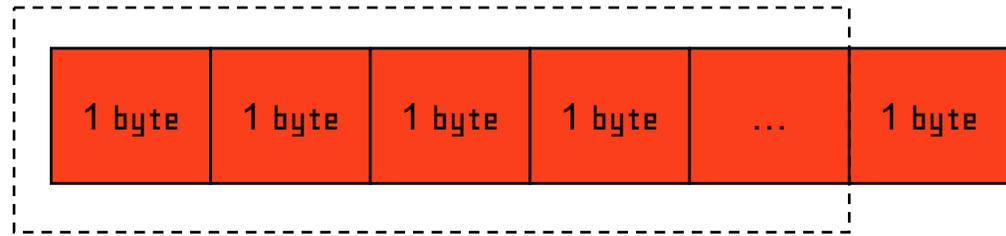
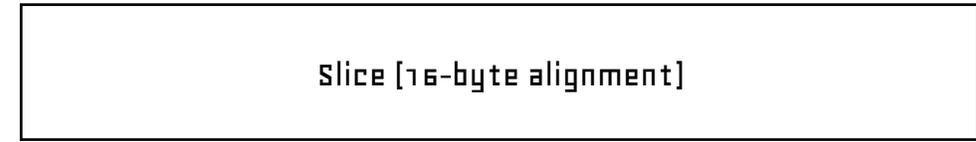
```
// 16-bit alignment for example
func Contains(s []uint8, target uint8) bool {
    for _, e := range s {
        if e == target {
            return true
        }
    }

    return false
}
```

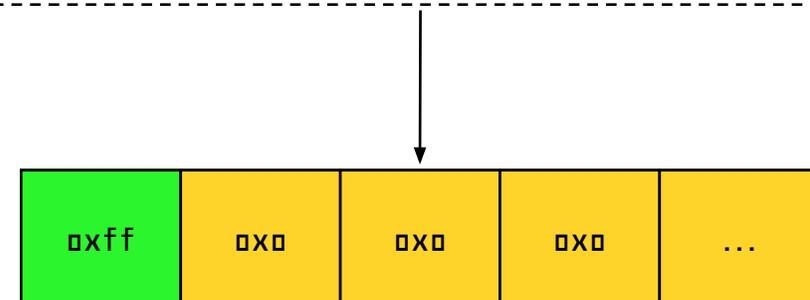
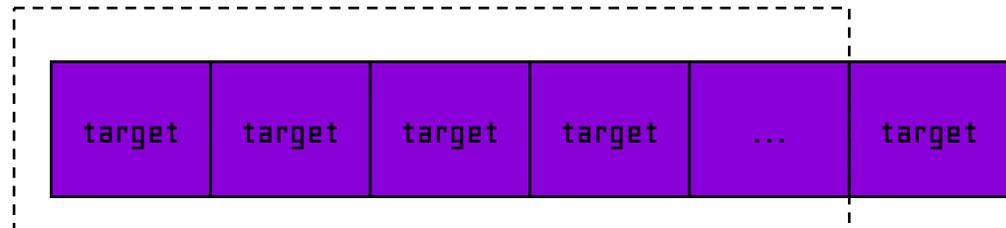
Slice contains with SIMD instructions



```
// 16-bit alignment to simplify
func ContainsSIMD(s []uint8, target uint8) bool {
    for i := 0; i < len(s); i += 16 {
        if compare(
            s[i:i+16],
            Duplicate(target, 16),
        ) != 0 {
            return true
        }
    }
    return false
}
```



COMPARE [16 byte]



Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1) // R0 - data pointer, R1 - length
    MOVB target+24(FP), R2 // R2 = target
    VDUP R2, V1.B16 // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16] // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16 // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2 // V2 = sum(V3)
    VMOV V2.H[0], R4 // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```

Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
    MOVB target+24(FP), R2           // R2 = target
    VDUP R2, V1.B16                 // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no                       // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16]          // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16    // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2
    VMOV V2.H[0], R4                 // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```

Slice contains with SIMD instructions

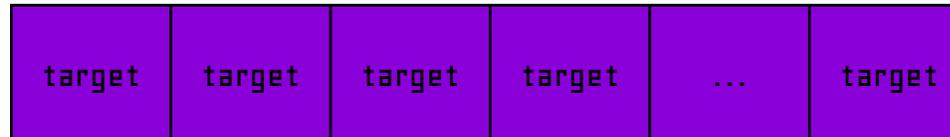


```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
    MOVW target+24(FP), R2           // R2 = target
    VDUP R2, V1.B16                  // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no                       // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16]           // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16     // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2                 // V2 = sum(V3)
    VMOV V2.H[0], R4                 // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```



Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
    MOVB target+24(FP), R2           // R2 = target
    VDUP R2, V1.B16                 // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no                       // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16]          // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16    // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2
    VMOV V2.H[0], R4                 // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```

Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
    MOVB target+24(FP), R2          // R2 = target
    VDUP R2, V1.B16                 // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no                      // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16]         // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16   // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2
    VMOV V2.H[0], R4                // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```

Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
```

```
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
```

```
LDP slice_base+0(FP), (R0, R1)
```

```
MOVB target+24(FP), R2
```

```
VDUP R2, V1.B16
```

```
loop:
```

```
CBZ R1, no
```

```
VLD1.P 16(R0), [V2.B16]
```

```
VCMEQ V1.B16, V2.B16, V3.B16
```

```
VADDV V3.B16, V2
```

```
VMOV V2.H[0], R4 // Half precision
```

```
CBNZ R4, yes
```

```
SUB $16, R1
```

```
B loop
```

```
no:
```

```
MOVD $0, R5
```

```
MOVD R5, ret+32(FP)
```

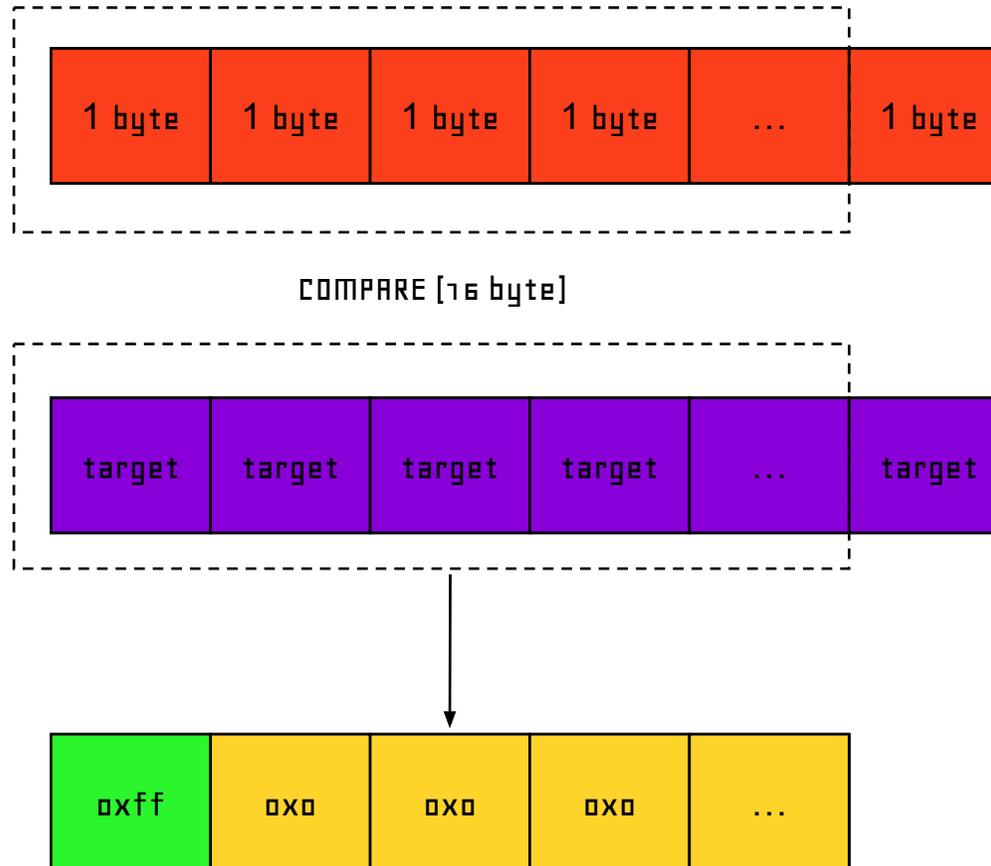
```
RET
```

```
yes:
```

```
MOVD $1, R5
```

```
MOVD R5, ret+32(FP)
```

```
RET
```



Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
    MOVB target+24(FP), R2           // R2 = target
    VDUP R2, V1.B16                 // V1 = [R2, R2, R2...R2]

loop:
    CBZ R1, no                       // if R1 == 0 {no code}
    VLD1.P 16(R0), [V2.B16]          // V2 = *(R0)[:16]; R0 += 16
    VCMEQ V1.B16, V2.B16, V3.B16    // compare V1 and V2. V3 = [0000000 111111 000000...]
    VADDV V3.B16, V2                // V2 = sum(V3)
    VMOV V2.H[0], R4                // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
    CBNZ R4, yes
    SUB $16, R1
    B loop

no:
    MOVD $0, R5
    MOVD R5, ret+32(FP)
    RET

yes:
    MOVD $1, R5
    MOVD R5, ret+32(FP)
    RET
```



Slice contains with SIMD instructions



```
//func SliceContainsV1(s []uint8, target uint8) bool
```

```
TEXT ·SliceContainsV1(SB), NOSPLIT, $0
```

```
    LDP slice_base+0(FP), (R0, R1)    // R0 - data pointer, R1 - length
```

```
    MOVB target+24(FP), R2           // R2 = target
```

```
    VDUP R2, V1.B16                 // V1 = [R2, R2, R2...R2]
```

```
loop:
```

```
    CBZ R1, no                       // if R1 == 0 {no code}
```

```
    VLD1.P 16(R0), [V2.B16]          // V2 = *(R0)[:16]; R0 += 16
```

```
    VCMEQ V1.B16, V2.B16, V3.B16    // compare V1 and V2. V3 = [0000000 111111 000000...]
```

```
    VADDV V3.B16, V2                // V2 = sum(V3)
```

```
    VMOV V2.H[0], R4                 // V2=[int16,int16,int16...]; R4 = V2[0], H - half precision
```

```
    CBNZ R4, yes
```

```
    SUB $16, R1
```

```
    B loop
```

```
no:
```

```
    MOVD $0, R5
```

```
    MOVD R5, ret+32(FP)
```

```
    RET
```

```
yes:
```

```
    MOVD $1, R5
```

```
    MOVD R5, ret+32(FP)
```

```
    RET
```



06;

Выводы

Выводы



- У ассемблера из БД toolchain нестандартный синтаксис

Выводы



- У ассемблера из БП toolchain нестандартный синтаксис
- При использовании ассемблера теряется портативность

Выводы



- У ассемблера из `БП toolchain` нестандартный синтаксис
- При использовании ассемблера теряется портативность
- Ассемблер стоит использовать, если вы знаете, что делаете

Спасибо за внимание!

Панасюк Игорь

Бэкенд-разработчик

Яндекс Финтех



@IGORWALTHER