An Assembly primer

Guillaume DIDIER
L3 ENS - 2010/2021

An Assembly primer

Guillaume DIDIER

L3 ENS - 2010/2021

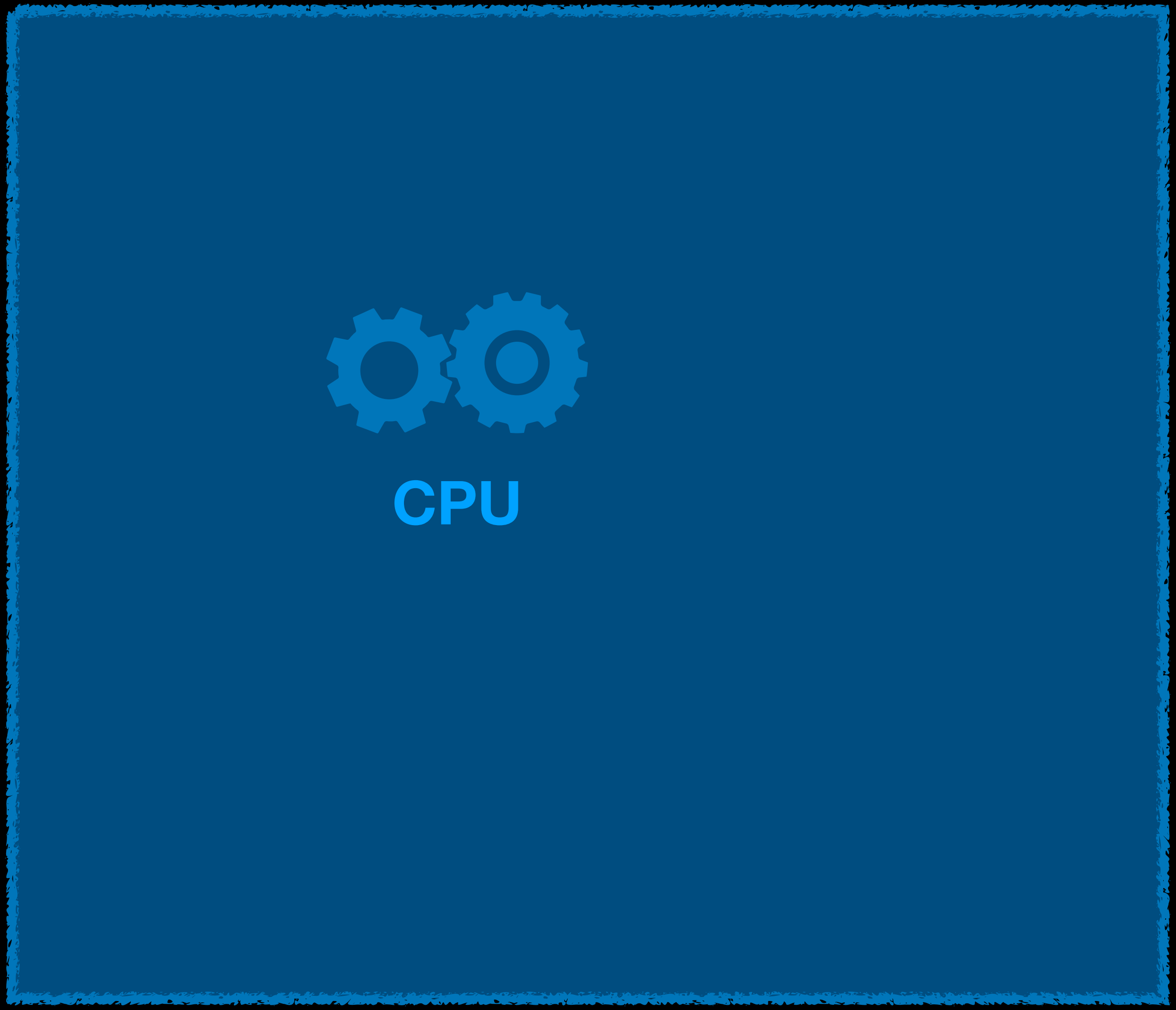
This lecture is fully preemptible
Feel free to interrupt with questions

Assembly / machine code view (ISA)

(user mode)

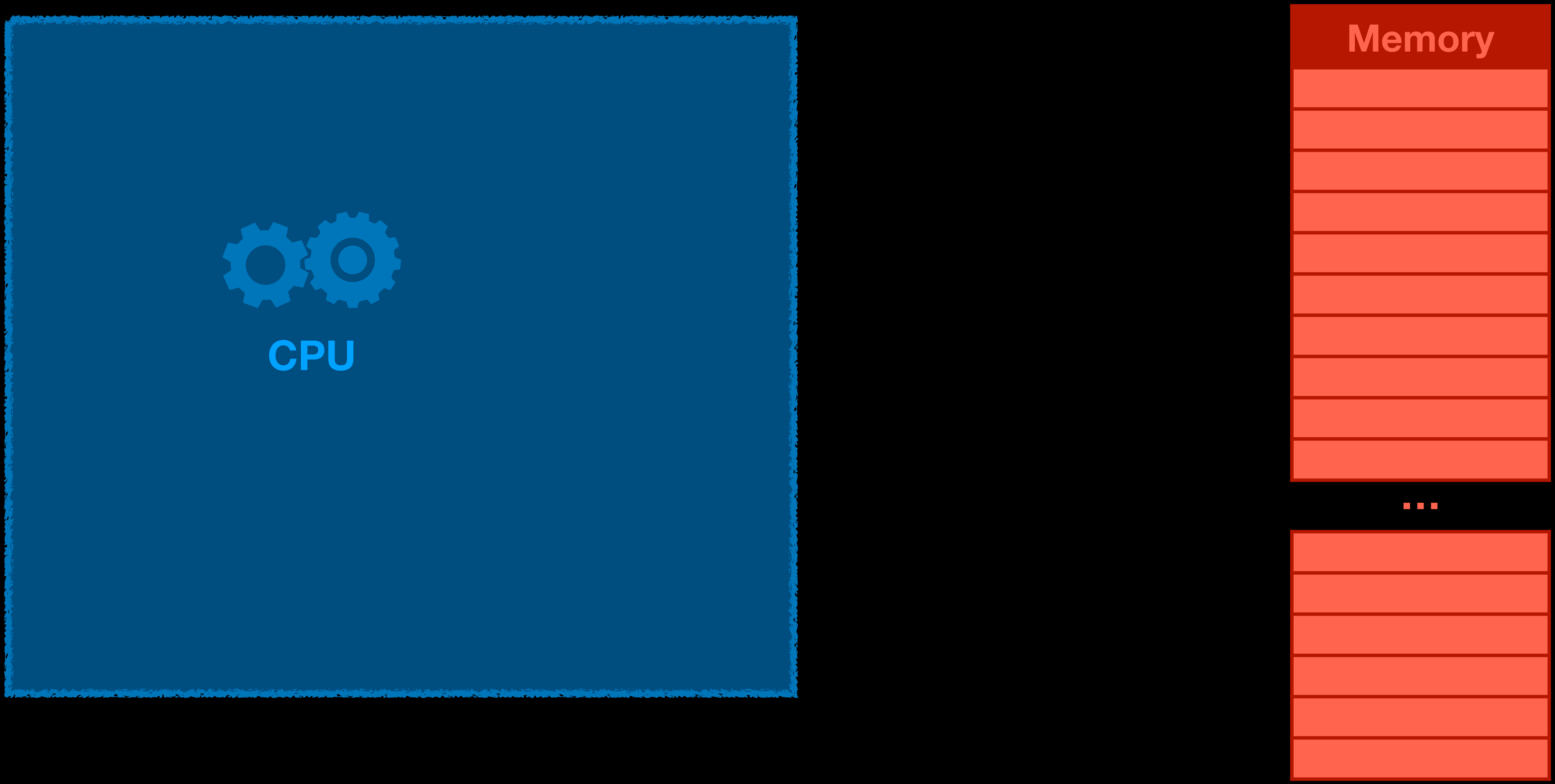
Assembly / machine code view (ISA)

(user mode)



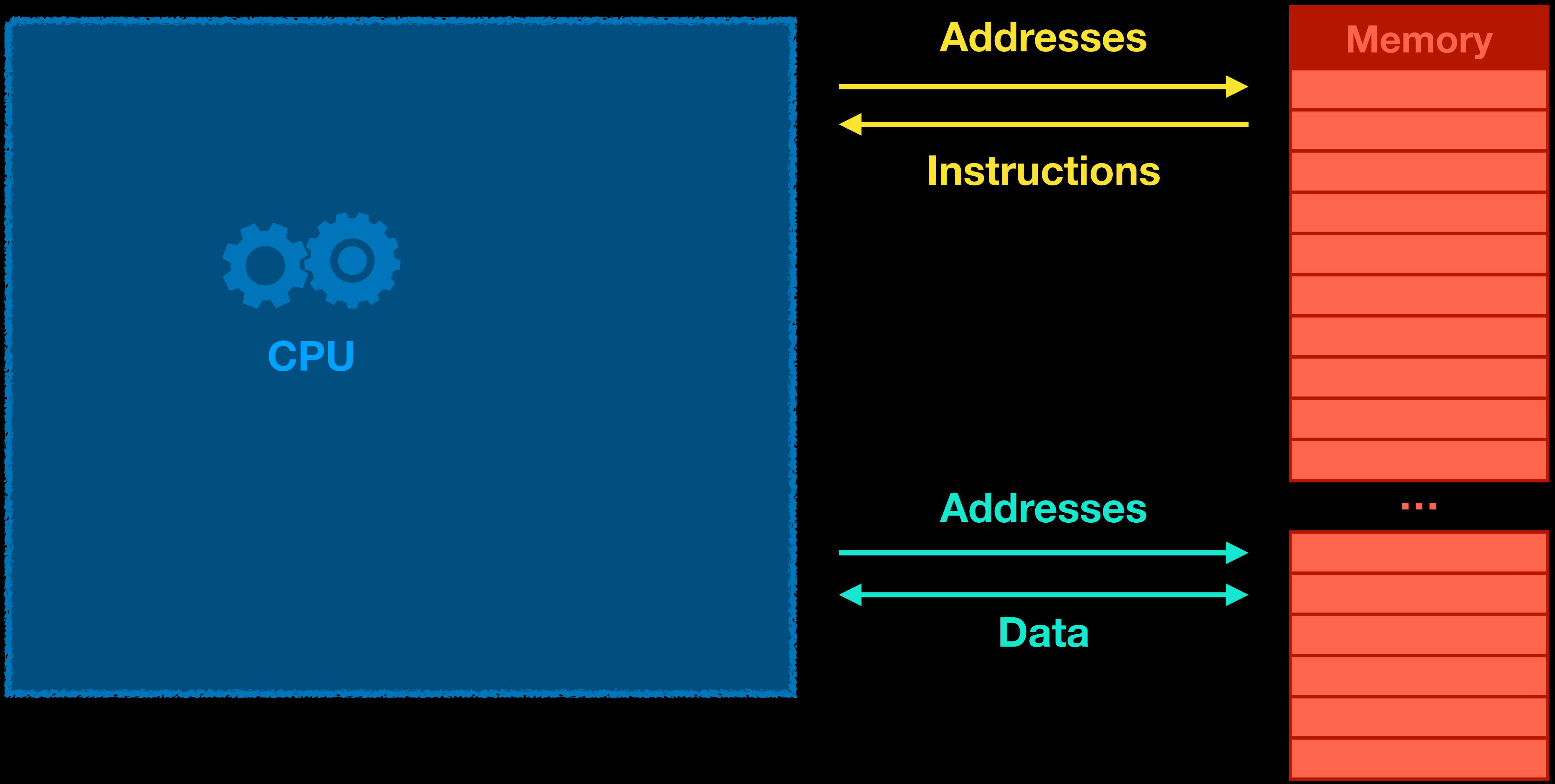
Assembly / machine code view (ISA)

(user mode)



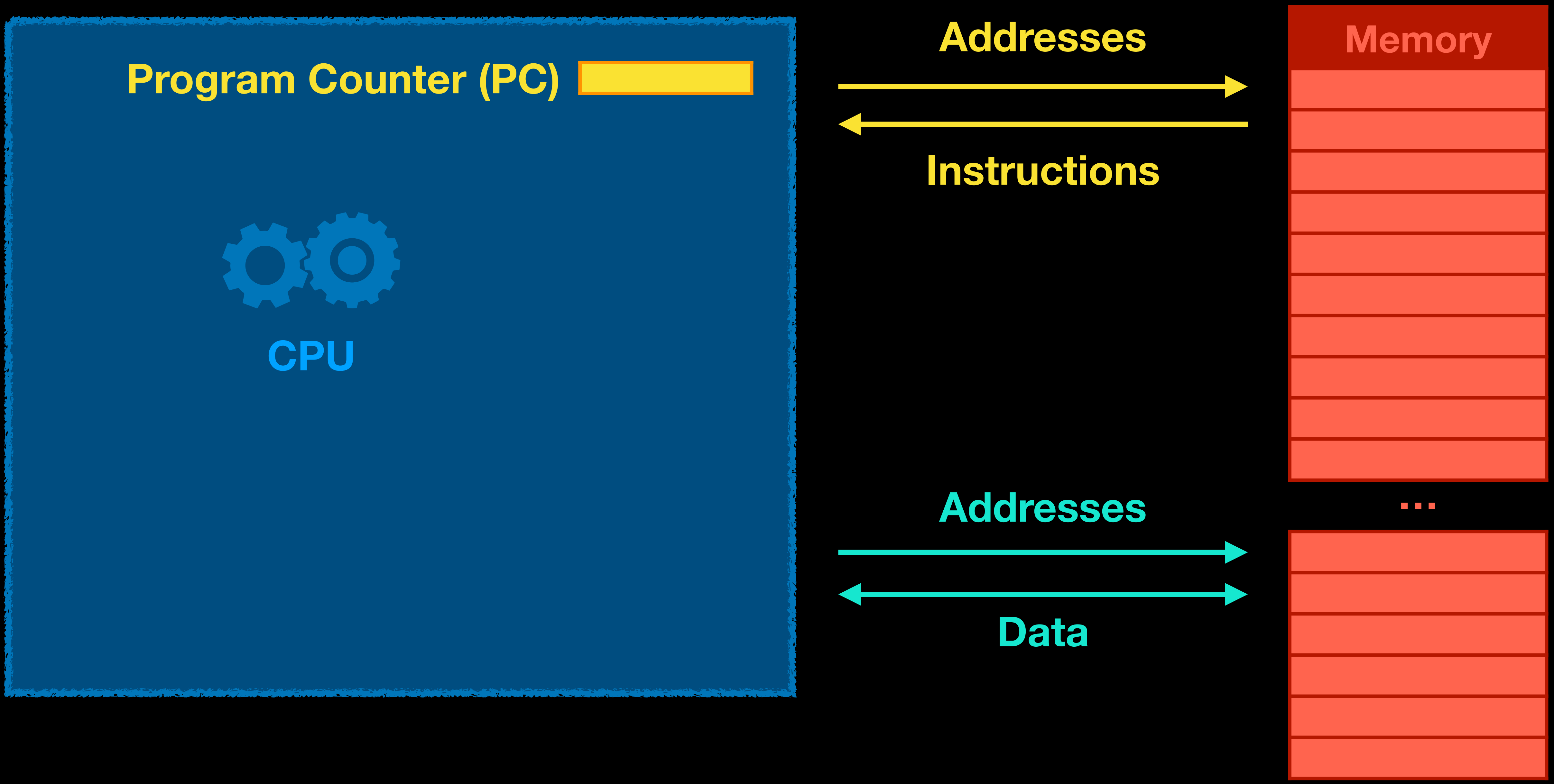
Assembly / machine code view (ISA)

(user mode)



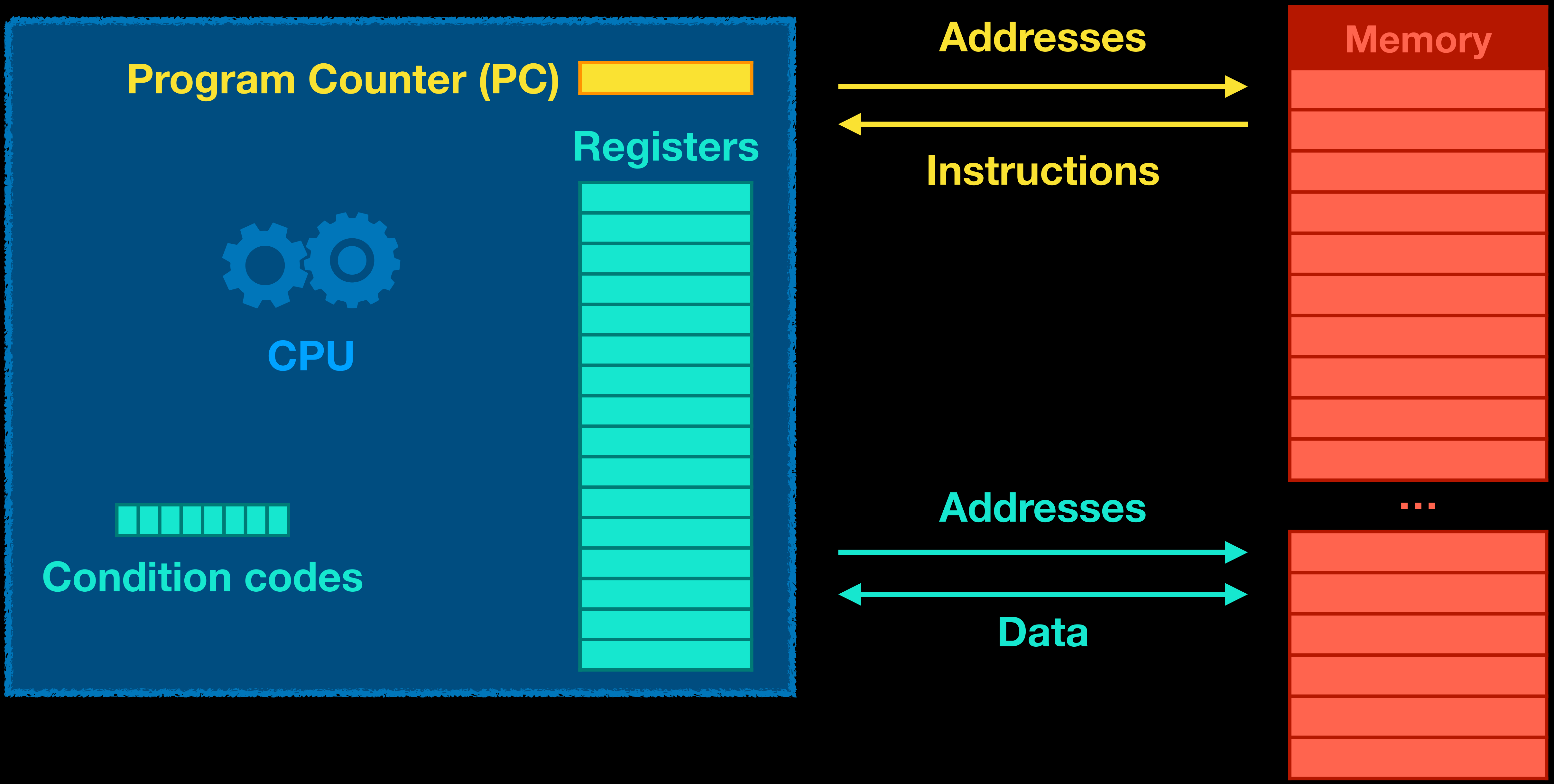
Assembly / machine code view (ISA)

(user mode)



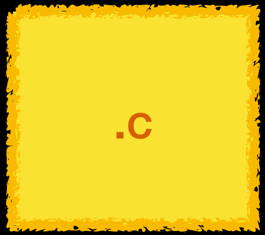
Assembly / machine code view (ISA)

(user mode)



How to turn C code into a running process

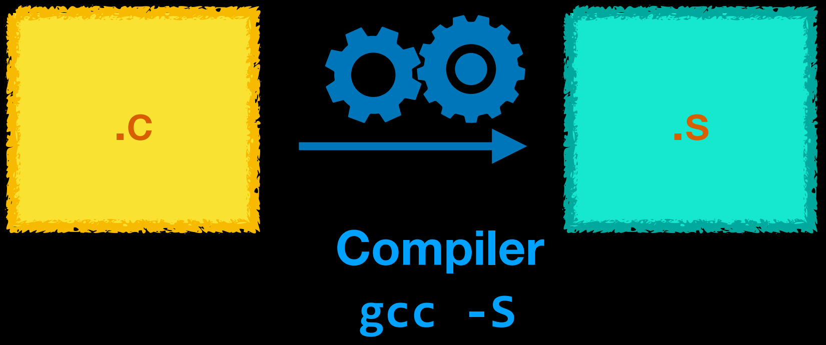
How to turn C code into a running process



```c
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

How to turn C code into a running process



.c

Compiler
`gcc -S`

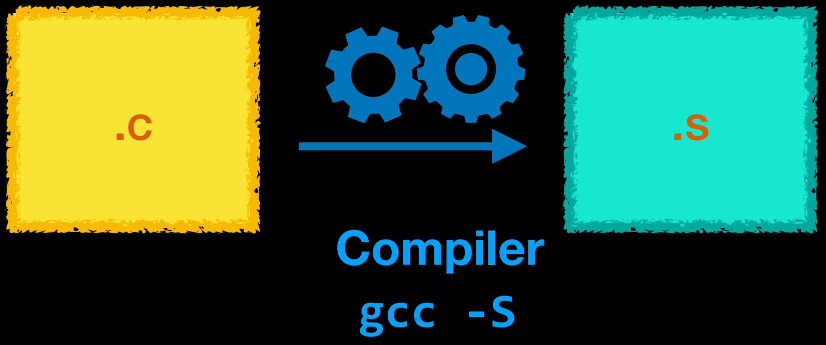
.s

```c
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

How to turn C code into a running process

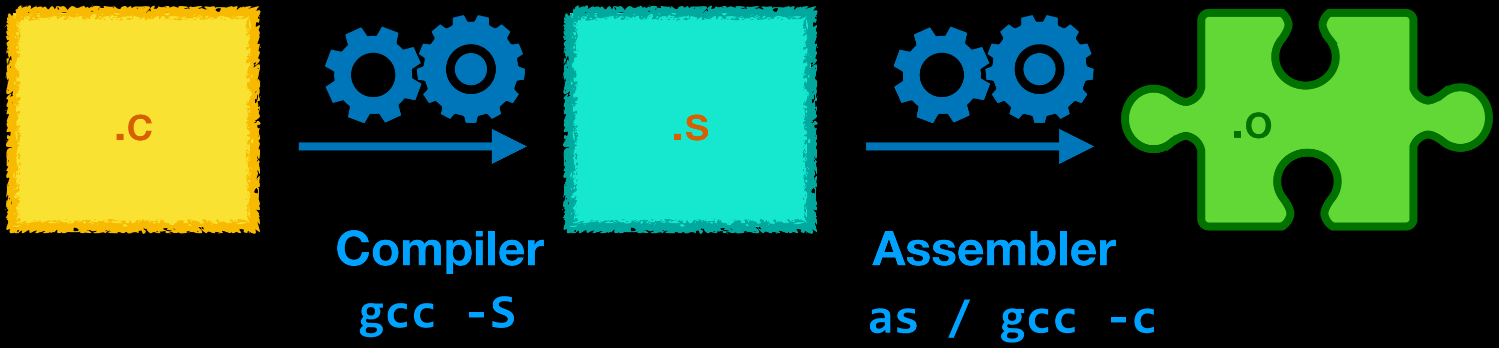


```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

How to turn C code into a running process



.c

Compiler
gcc -S

.s

Assembler
as / gcc -c

.o

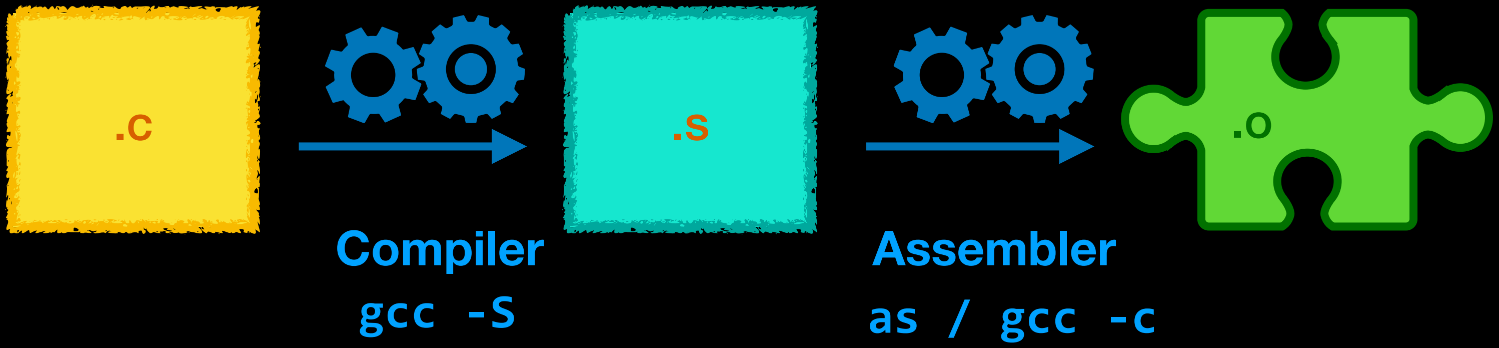
```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

How to turn C code into a running process



.c

Compiler
gcc -S

.s

Assembler
as / gcc -c

.o

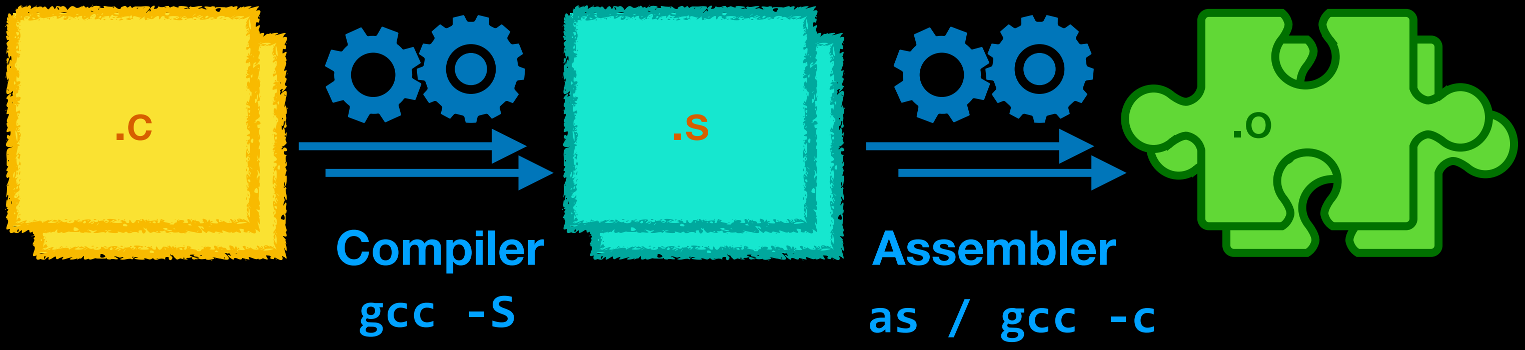
```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

How to turn C code into a running process



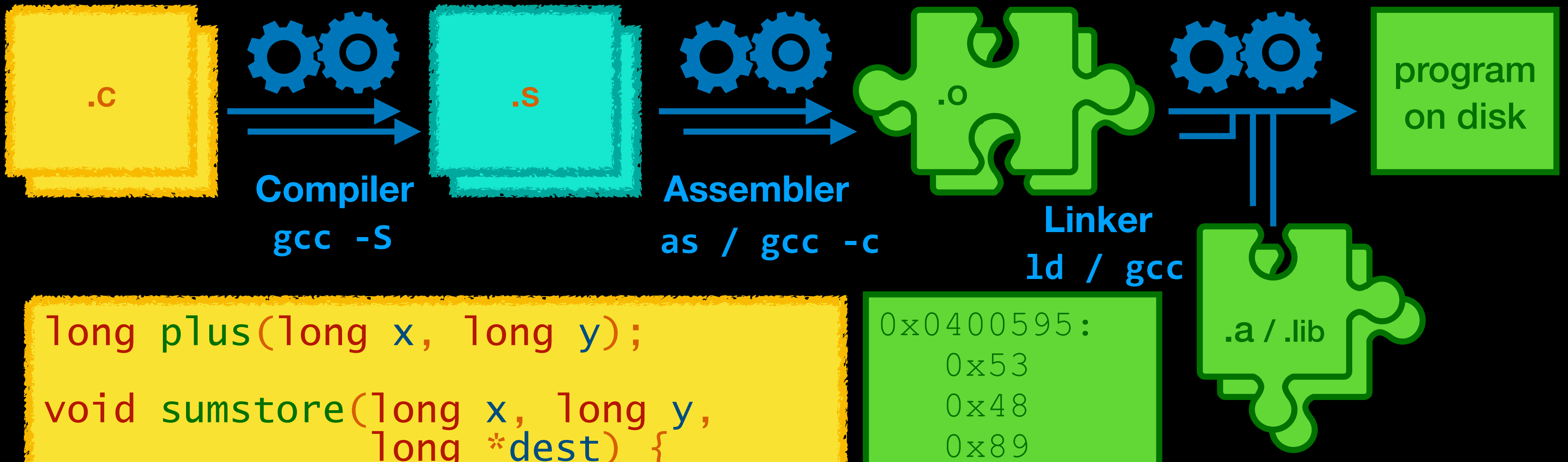
```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

How to turn C code into a running process



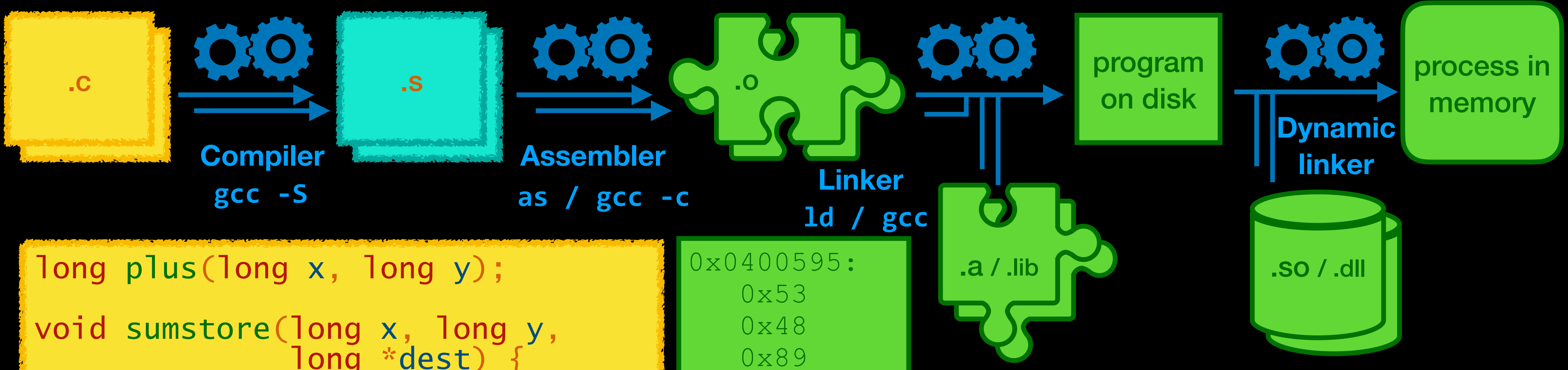
```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

How to turn C code into a running process



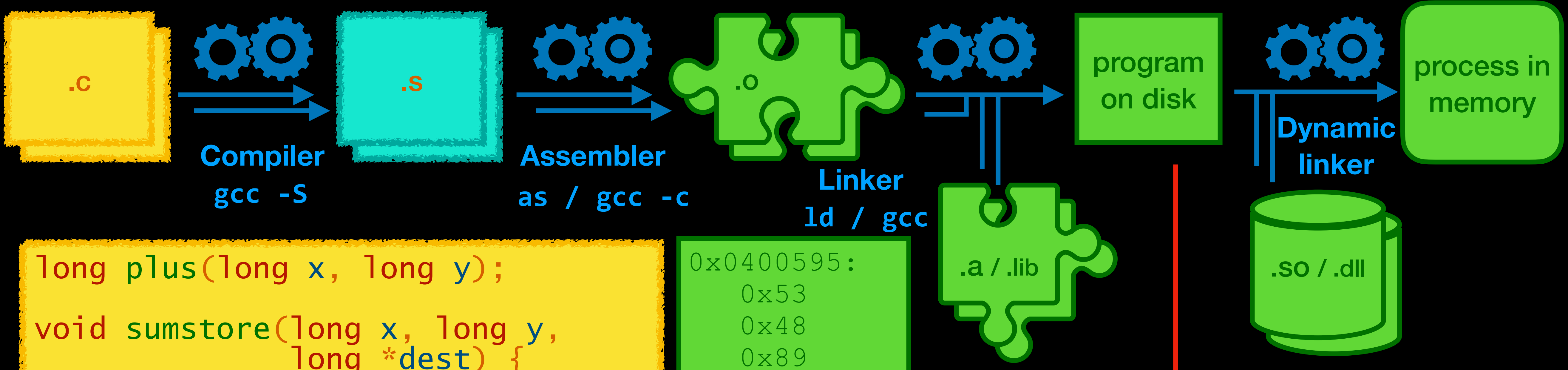
```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```

How to turn C code into a running process

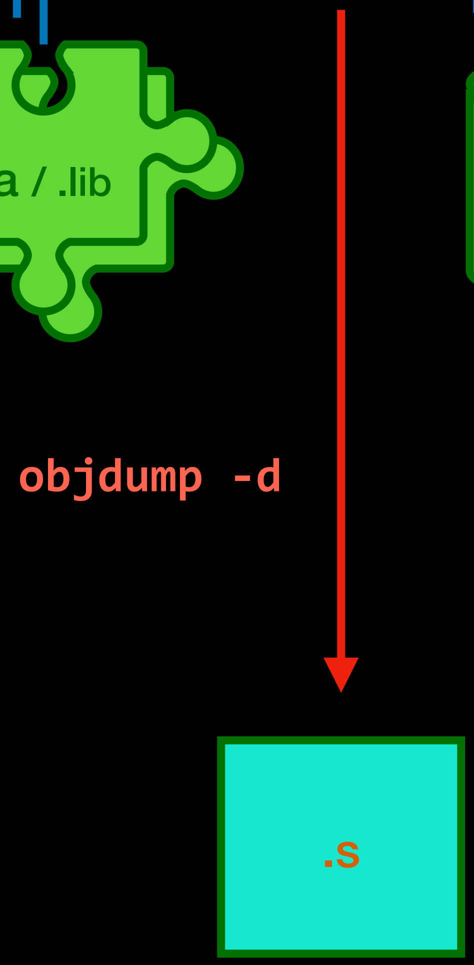


```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

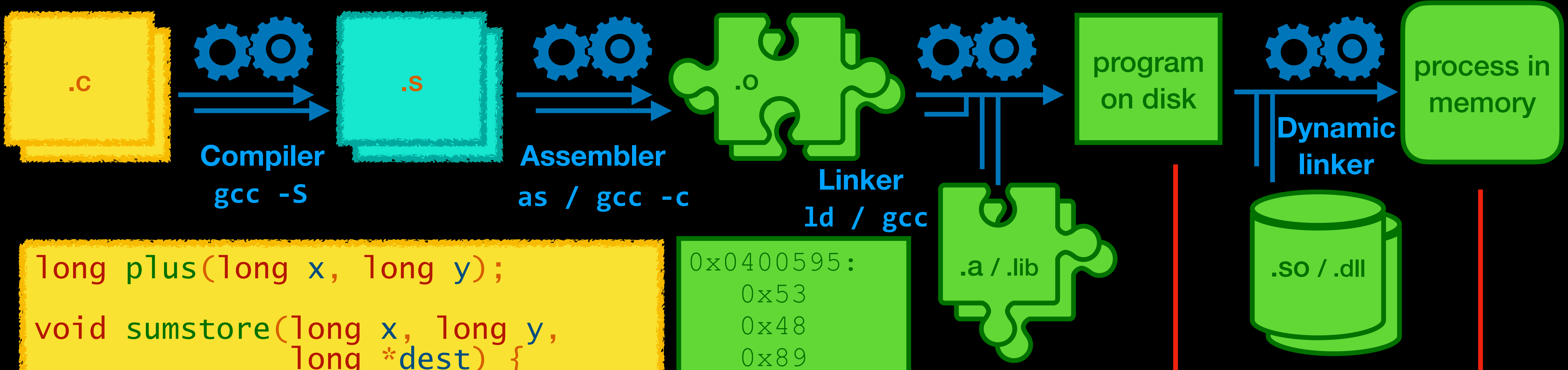
```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```



```
0000000000400595 <sumstore>:
  400595:   53                push    %rbx
  400596:   48 89 d3          mov     %rdx,%rbx
  400599:   e8 f2 ff ff ff    callq   400590 <plus>
  40059e:   48 89 03          mov     %rax,(%rbx)
  4005a1:   5b                pop     %rbx
  4005a2:   c3                retq
```

How to turn C code into a running process

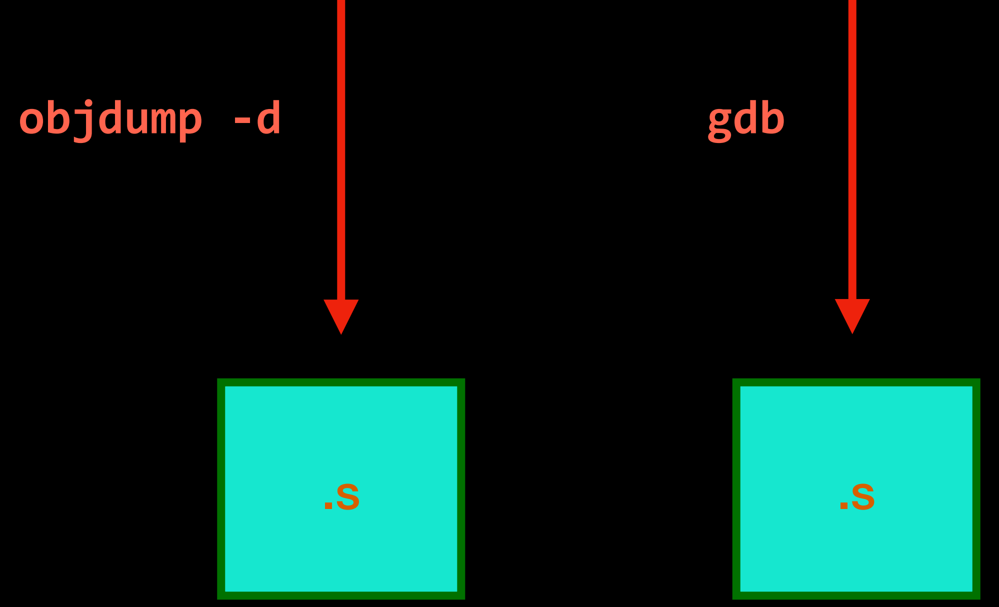


```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
0x0400595:
   0x53
   0x48
   0x89
   0xd3
   0xe8
   0xf2
   0xff
   0xff
   0xff
   0x48
   0x89
   0x03
   0x5b
   0xc3
```



```
0000000000400595 <sumstore>:
  400595:  53                push   %rbx
  400596:  48 89 d3          mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff    callq  400590 <plus>
  40059e:  48 89 03          mov    %rax,(%rbx)
  4005a1:  5b                pop    %rbx
  4005a2:  c3                retq
```

Why we care ?

Why we care ?

- Same productivity in line of code per day in C and asm, you get more done by writing C.

Why we care ?

- Same productivity in line of code per day in C and asm, you get more done by writing C.
- System programmers *write* little asm but *read* it a lot.

Why we care ?

- Same productivity in line of code per day in C and asm, you get more done by writing C.
- System programmers *write* little asm but *read* it a lot.
- Even more for security researchers.

Why we care ?

- Same productivity in line of code per day in C and asm, you get more done by writing C.
- System programmers *write* little asm but *read* it a lot.
- Even more for security researchers.
- Also useful to understand code performance.

Why we care ?

- Same productivity in line of code per day in C and asm, you get more done by writing C.
- System programmers *write* little asm but *read* it a lot.
- Even more for security researchers.
- Also useful to understand code performance.
- You may need it for your work!

x86 has a long history

x86 has a long history

- First non Intel CPUs before 1970.

x86 has a long history

- First non Intel CPUs before 1970.
- 1978 : birth of x86, Intel 8086 is a 16-bit micro processor

x86 has a long history

- First non Intel CPUs before 1970.
- 1978 : birth of x86, Intel 8086 is a 16-bit micro processor
- 1985 : extension to 32-bit

x86 has a long history

- First non Intel CPUs before 1970.
- 1978 : birth of x86, Intel 8086 is a 16-bit micro processor
- 1985 : extension to 32-bit
- 2003 (AMD)-2004 (Intel) : 64-bit extension.

x86 has a long history

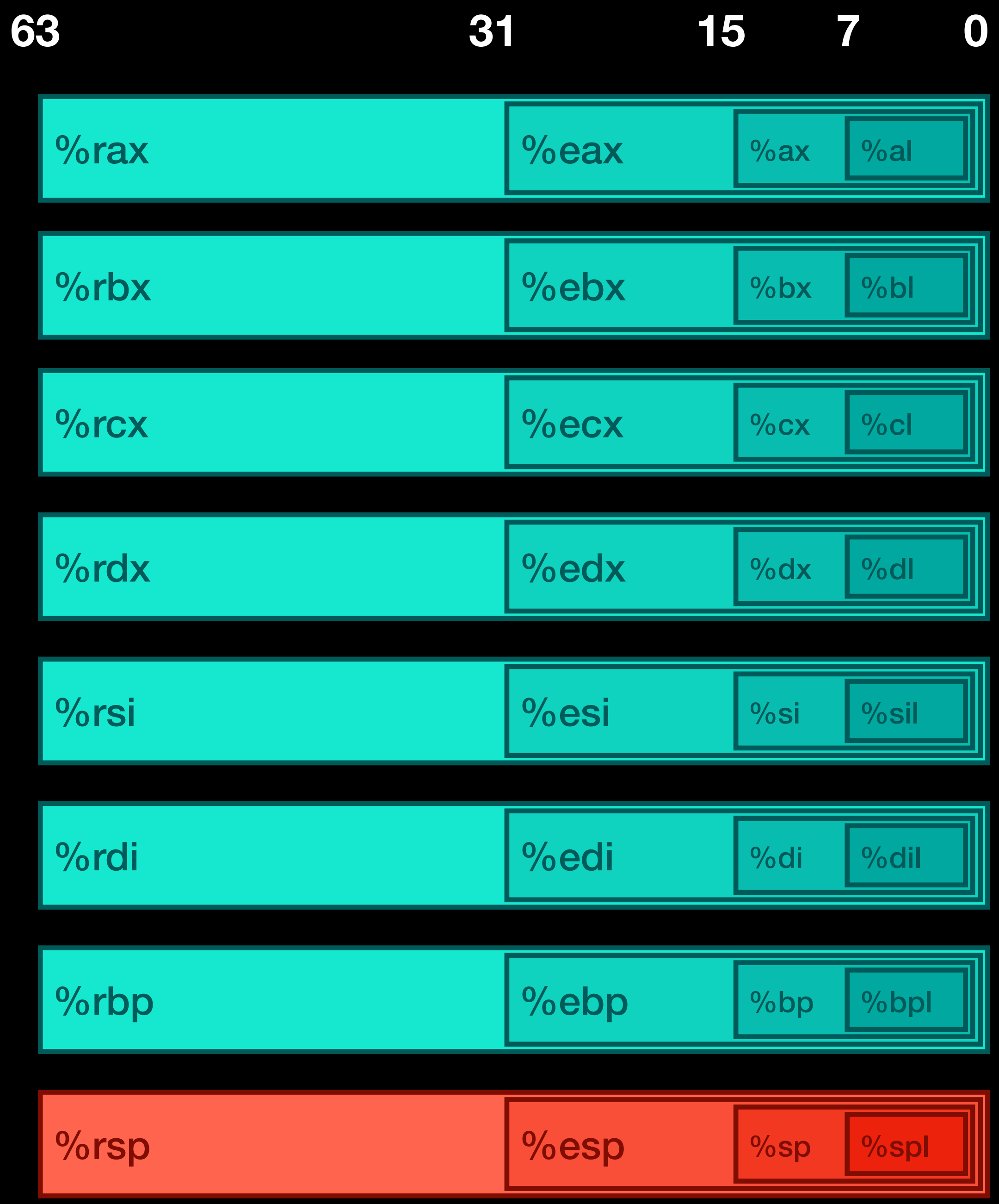
- First non Intel CPUs before 1970.
- 1978 : birth of x86, Intel 8086 is a 16-bit micro processor
- 1985 : extension to 32-bit
- 2003 (AMD)-2004 (Intel) : 64-bit extension.
- *A lot of crufts left-over of x86 long and convoluted history.*

x86 has a long history

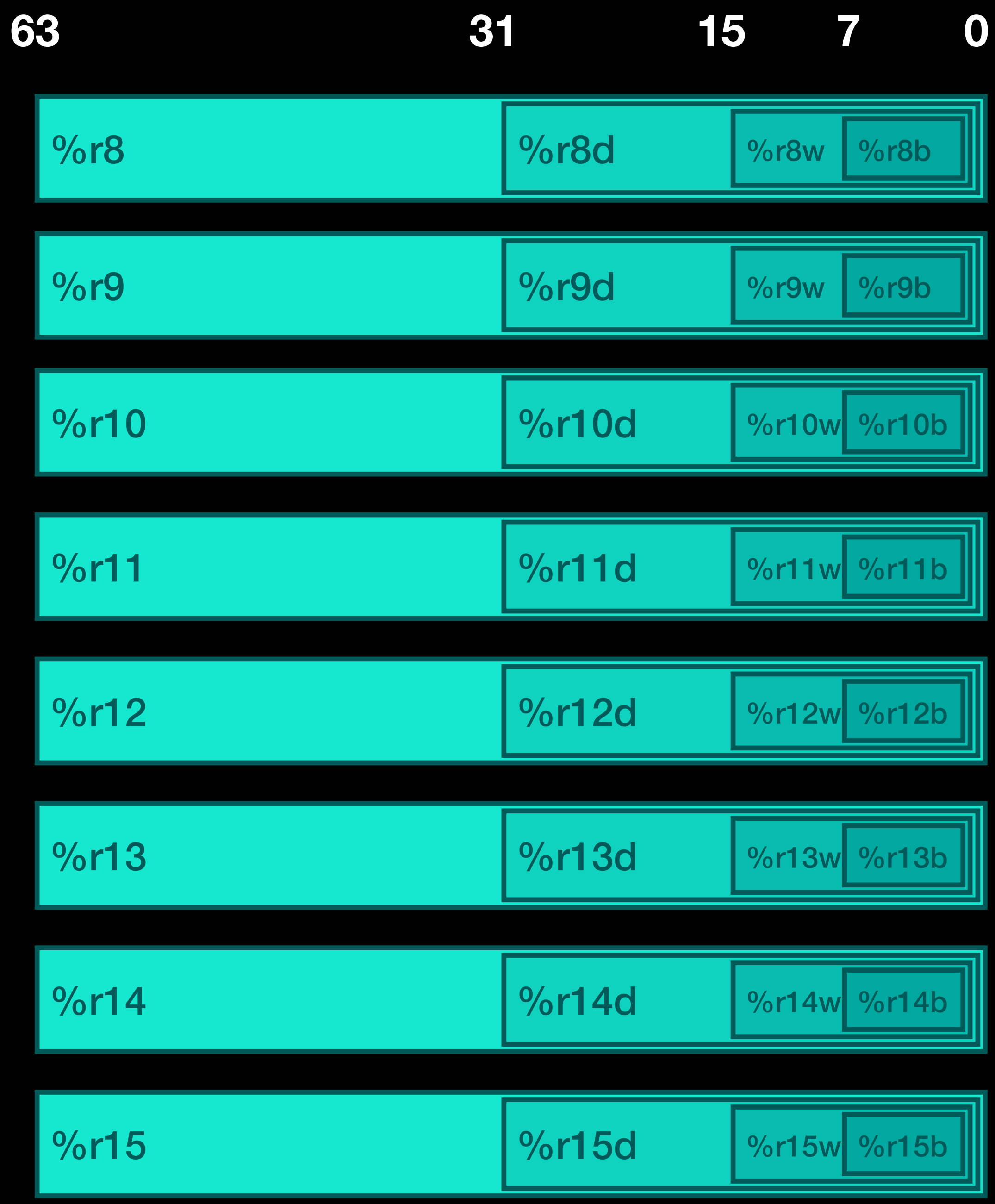
Registers

Registers

- 16 registers, each 64 bits (8 bytes).



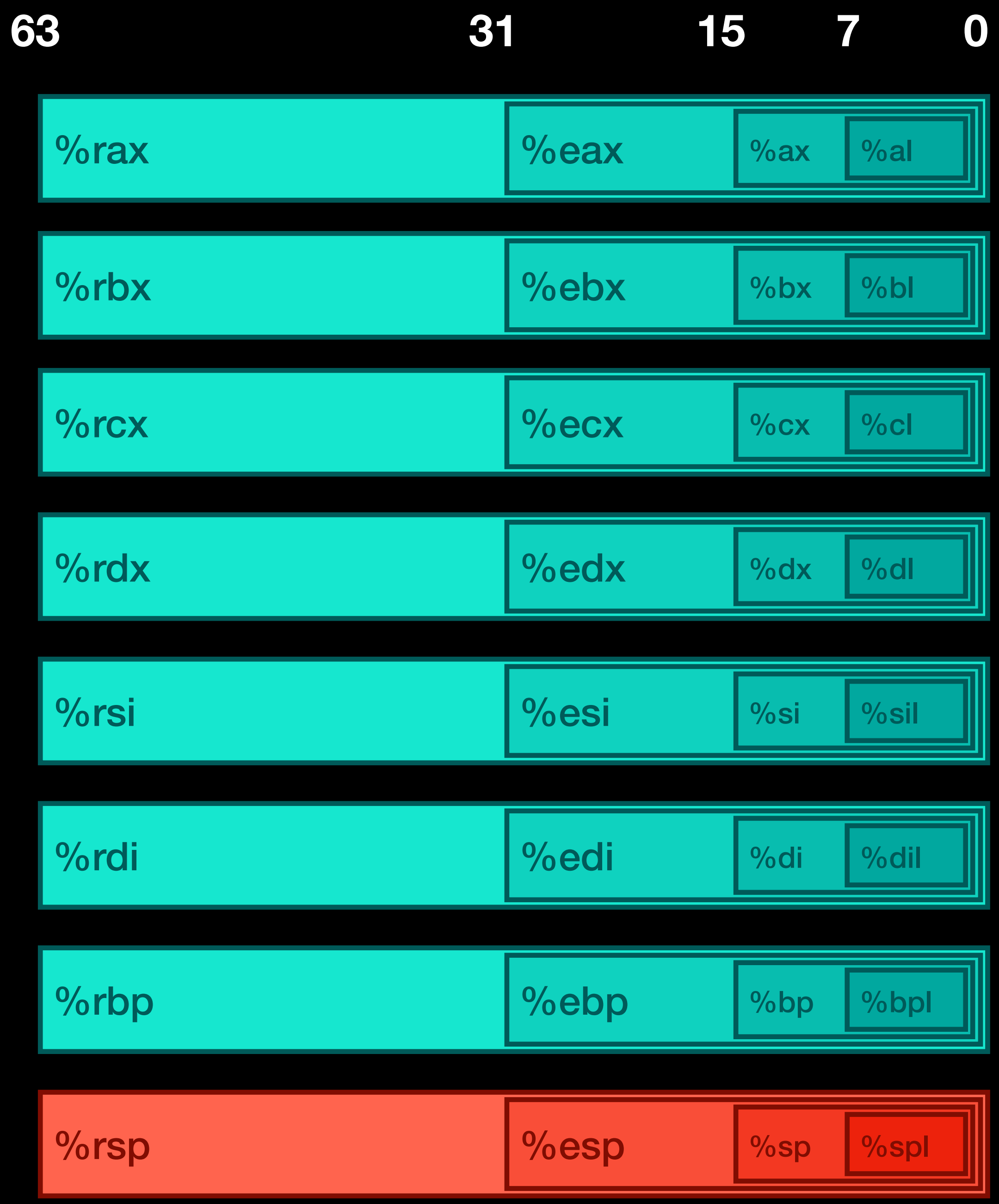
63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	



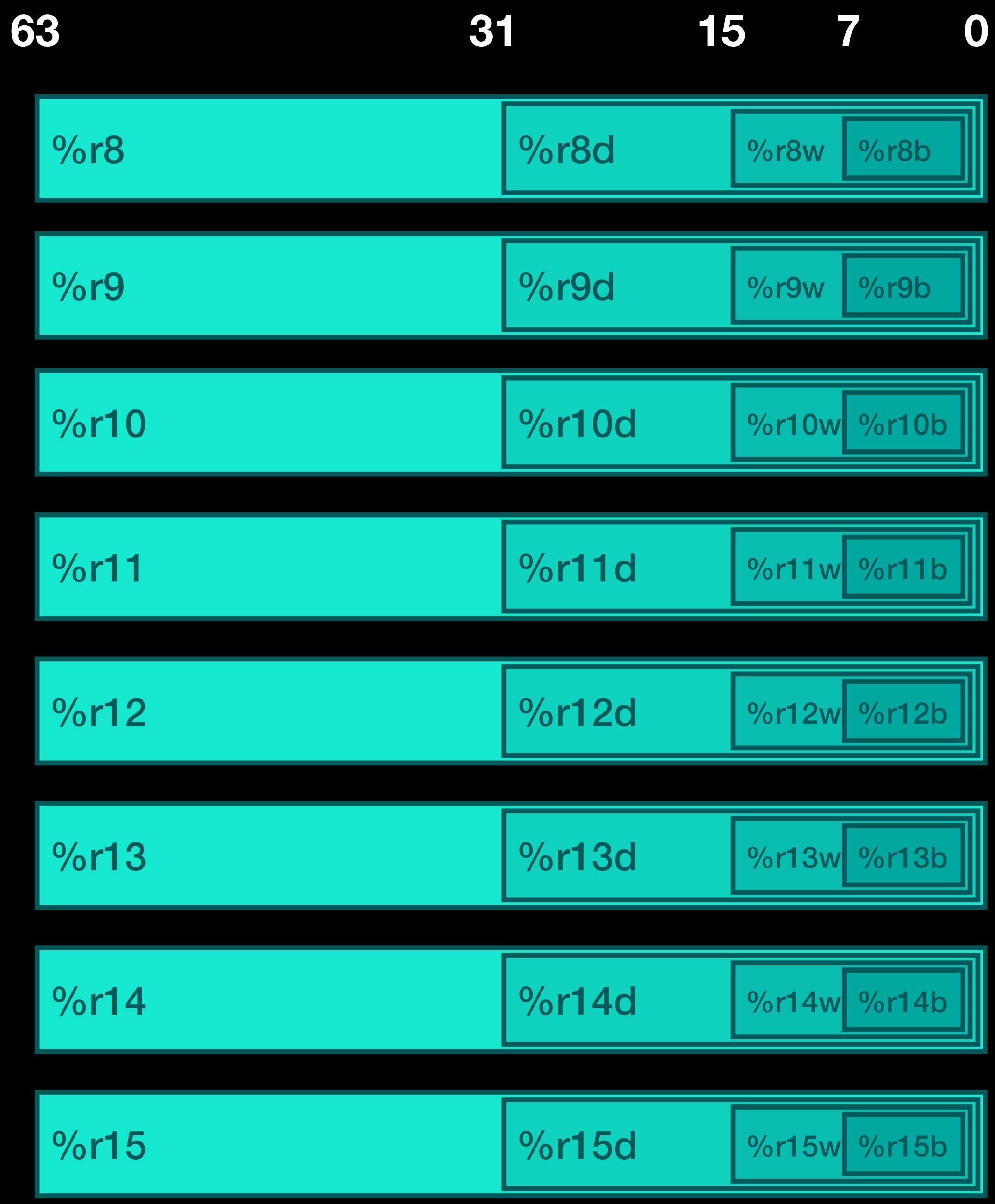
63	31	15	7	0
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Registers

- 16 registers, each 64 bits (8 bytes).
- What the CPU mostly operates on.



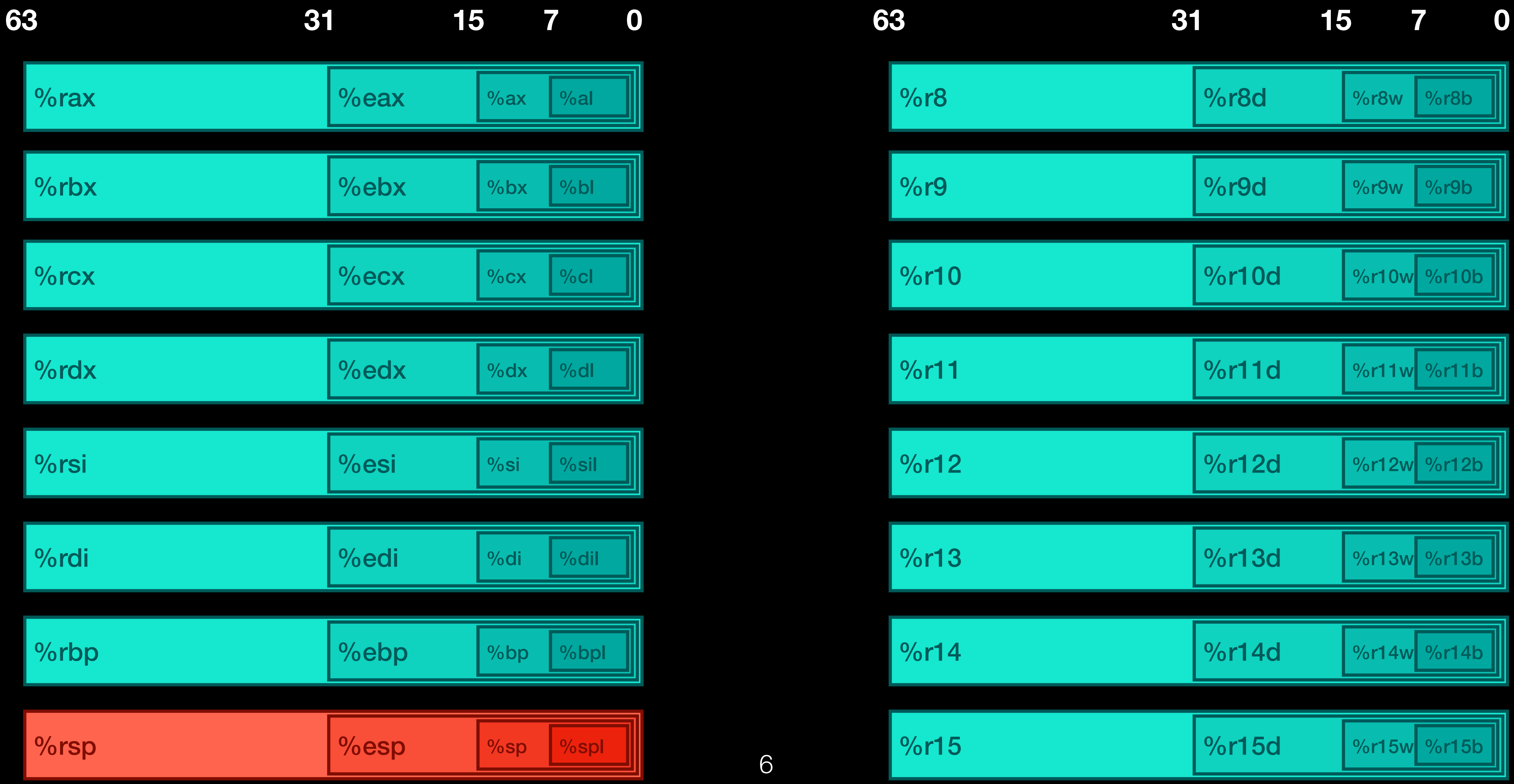
63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	



63	31	15	7	0
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Registers

- 16 registers, each 64 bits (8 bytes).
- What the CPU mostly operates on.
- Suffixes on instructions used to make the size of operands explicit (can be omitted in some cases): q, l, w, b, for resp. 8, 4, 2 and 1 bytes values



63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	

63	31	15	7	0
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Assembler syntax

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel
used by	unix, gcc, linux, macOS etc	the microsoft world

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel
used by	unix, gcc, linux, macOS etc	the microsoft world
operand order	destination operand is last	destination operand comes first

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel
used by	unix, gcc, linux, macOS etc	the microsoft world
operand order	destination operand is last	destination operand comes first
example	`movl $1, 0x8(%rsp,%rsi,4)`	`mov dword ptr [rsp+rsi*4h+8h], 1`

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel
used by	unix, gcc, linux, macOS etc	the microsoft world
operand order	destination operand is last	destination operand comes first
example	`movl $1, 0x8(%rsp,%rsi,4)`	`mov dword ptr [rsp+rsi*4h+8h], 1`
other differences	size suffix, immediate with $, addressing mode with D(,,), register with %, 0x for hex	no size suffix, addressing with [], `size ptr` size specification when needed, no % or $, h suffix for hex

Assembler syntax

- Anatomy of an assembler instruction

```
mnemonic operand, operand …
```

- Additionally labels can be inserted to point at specific points

```
label: instruction
```

- For x86 asm, two syntax exist, and have the operands in opposite orders:

syntax	AT&T	Intel
used by	unix, gcc, linux, macOS etc	the microsoft world
operand order	destination operand is last	destination operand comes first
example	`movl $1, 0x8(%rsp,%rsi,4)`	`mov dword ptr [rsp+rsi*4h+8h], 1`
other differences	size suffix, immediate with $, addressing mode with D(,,), register with %, 0x for hex	no size suffix, addressing with [], `size ptr` size specification when needed, no % or $, h suffix for hex

In this presentation we exclusively use AT&T syntax.

Moving data around

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:

Moving data around

- To move a « quad word » (8 bytes) of data:
  ```
  movq src, dest
  ```
- Operand types:
  - Register: One the 16 quad word registers above

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only
  - Memory (at most one of the two)

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only
  - Memory (at most one of the two)
    - As many bytes as needed starting at an address in register

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only
  - Memory (at most one of the two)
    - As many bytes as needed starting at an address in register
    - Simple example `(%rax)`

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only
  - Memory (at most one of the two)
    - As many bytes as needed starting at an address in register
    - Simple example `(%rax)`
    - Other addressing mode exist (see later)

Moving data around

- To move a « quad word » (8 bytes) of data:
  `movq src, dest`
- Operand types:
  - Register: One the 16 quad word registers above
    - Example: `%rax`, `%r13`
    - `%rsp` is special
    - Some instructions have special cases
  - Immediate: an integer constant of 1,2 or 4 bytes
    - `$0x400`, `$42`, `$-1`
    - Same as in C but prefixed with $
    - Source operand only
  - Memory (at most one of the two)
    - As many bytes as needed starting at an address in register
    - Simple example `(%rax)`
    - Other addressing mode exist (see later)

Cannot do memory-memory transfer in a single instruction

Example

Example

```
movq
```

Example

	Source	Dest	Example	C analog
`movq`				

Example

	Source	Dest	Example	C analog
`movq`	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`

Example

	Source	Dest	Example	C analog
`movq`	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`

Example

	Source	Dest	Example	C analog
movq	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`
	Reg	*Reg*	`movq %rdi, %rax`	`tmp2 = tmp1;`

Example

	Source	Dest	Example	C analog
`movq`	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`
	Reg	*Reg*	`movq %rdi, %rax`	`tmp2 = tmp1;`
		Mem	`movq %rax, (%rsi)`	`*p = tmp;`

Example

	Source	Dest	Example	C analog
`movq`	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`
	Reg	*Reg*	`movq %rdi, %rax`	`tmp2 = tmp1;`
		Mem	`movq %rax, (%rsi)`	`*p = tmp;`
	Mem	*Reg*	`movq (%rsp), %rax`	`tmp = *p;`

Example

	Source	Dest	Example	C analog
`movq`	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`
	Reg	*Reg*	`movq %rdi, %rax`	`tmp2 = tmp1;`
		Mem	`movq %rax, (%rsi)`	`*p = tmp;`
	Mem	*Reg*	`movq (%rsp), %rax`	`tmp = *p;`

```
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Example

	Source	Dest	Example	C analog
movq	*Imm*	*Reg*	`movq $42, %rax`	`tmp = 42;`
		Mem	`movq $2020, (%rsp)`	`*year = 2020;`
	Reg	*Reg*	`movq %rdi, %rax`	`tmp2 = tmp1;`
		Mem	`movq %rax, (%rsi)`	`*p = tmp;`
	Mem	*Reg*	`movq (%rsp), %rax`	`tmp = *p;`

```
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
   movq     (%rdi), %rax
   movq     (%rsi), %rdx
   movq     %rdx, (%rdi)
   movq     %rax, (%rsi)
   ret
```

Addressing modes

Addressing modes

- Most general form :
  `D(Rb,Ri,S)` : Refers to addr = Reg[Rb] + Reg[Ri] * S + D
  - `D:` Constant displacement encoded on 1,2 or 4 bytes
  - `Rb:` Base register: Any of the 16 integer registers
  - `Ri:` Index register: Any except %rsp
  - `S:` scale: One of 1,2,4 or 8 (1 by default if omitted)

Addressing modes

- Most general form :
  `D(Rb,Ri,S)` : Refers to addr = Reg[Rb] + Reg[Ri] * S + D
  - `D:` Constant displacement encoded on 1,2 or 4 bytes
  - `Rb:` Base register: Any of the 16 integer registers
  - `Ri:` Index register: Any except %rsp
  - `S:` scale: One of 1,2,4 or 8 (1 by default if omitted)
- Any element can be omitted.
  - `0x48000` - only D
  - `0x42(,%rsi,2)` - Ri, S and D
  - `(%rdi, %rsi)` - Rb and Ri (S = 1)

Arithmetic operations

Arithmetic operations

Instruction		Computation	
`addq`	*Src, Dest*	Dest = Dest + Src	
`subq`	*Src, Dest*	Dest = Dest - Src	
`imul`	*Src, Dest*	Dest = Dest * Src	
`salq`	*Src, Dest*	Dest = Dest << Src	*Also known as* `shlq`
`sarq`	*Src, Dest*	Dest = Dest >> Src	*Arithmetic right shift*
`shrq`	*Src, Dest*	Dest = Dest >> Src	*Logical right shift*
`xorq`	*Src, Dest*	Dest = Dest ^ Src	
`andq`	*Src, Dest*	Dest = Dest & Src	
`orq`	*Src, Dest*	Dest = Dest | Src	
`incq`	*Dest*	Dest = Dest + 1	
`decq`	*Dest*	Dest = Dest - 1	
`negq`	*Dest*	Dest = – Dest	*Two complement*
`notq`	*Dest*	Dest = ~Dest	*Bitwise negation*

…

This is not an exhaustive list.

Jumping

Jumping

- Redirect the control flow elsewhere than the next instruction:
  - Define a *label* to jump to (`label:` before an instruction),
  - `jmpq label` will redirect execution to the label.
  - The assembler and linker will encode the correct numbers in the instruction for you.

Jumping

- Redirect the control flow elsewhere than the next instruction:
    - Define a *label* to jump to (`label:` before an instruction),
    - `jmpq label` will redirect execution to the label.
    - The assembler and linker will encode the correct numbers in the instruction for you.
- Also possible to jump to a address in a register:
  `jmpq *%rax`

Jumping

- Redirect the control flow elsewhere than the next instruction:
  - Define a *label* to jump to (`label:` before an instruction),
  - `jmpq label` will redirect execution to the label.
  - The assembler and linker will encode the correct numbers in the instruction for you.
- Also possible to jump to a address in a register:
  `jmpq *%rax`

```
void loop(long a) {
    a = a + a;
    while(1) {
        a++;
    }
}
```

Jumping

- Redirect the control flow elsewhere than the next instruction:
    - Define a *label* to jump to (`label:` before an instruction),
    - `jmpq label` will redirect execution to the label.
    - The assembler and linker will encode the correct numbers in the instruction for you.
- Also possible to jump to a address in a register:
  `jmpq *%rax`

```
void loop(long a) {
    a = a + a;
    while(1) {
        a++;
    }
}
```

```
loop:
    addq     %rdi, %rdi
while:
    addq     $1, %rdi
    jmpq     while
```

Condition codes

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
  - `addq a, b` sets flags depending on `t = a + b`.

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
    - `addq a, b` sets flags depending on t = a + b.
- 2 instruction set conditions, discard result:
    - `testq a,b` sets flags depending on t = a & b (think of it as `andq`)
    - `cmpq a, b` sets flags depending on t = b - a (think of it as a `subq`)

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
  - `addq a, b` sets flags depending on t = a + b.
- 2 instruction set conditions, discard result:
  - `testq a,b` sets flags depending on t = a & b (think of it as `andq`)
  - `cmpq a, b` sets flags depending on t = b - a (think of it as a `subq`)
- Used by the `setXX` (`sete`, `setge`, …) (Conditionally set a byte (8 bit) to 0 or 1),

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
    - `addq a, b` sets flags depending on t = a + b.
- 2 instruction set conditions, discard result:
    - `testq a,b` sets flags depending on t = a & b (think of it as `andq`)
    - `cmpq a, b` sets flags depending on t = b - a (think of it as a `subq`)
- Used by the `setXX` (`sete`, `setge`, …) (Conditionally set a byte (8 bit) to 0 or 1),
- Mostly used by conditional jump instructions (see next slide).

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
  - `addq a, b` sets flags depending on `t` = `a` + `b`.
- 2 instruction set conditions, discard result:
  - `testq a,b` sets flags depending on `t` = `a` & `b` (think of it as `andq`)
  - `cmpq a, b` sets flags depending on `t` = `b` - `a` (think of it as a `subq`)
- Used by the `setXX` (`sete`, `setge`, …) (Conditionally set a byte (8 bit) to 0 or 1),
- Mostly used by conditional jump instructions (see next slide).

Flag	**Name**	**When is it set**
CF	Carry flag	Arithmetic operation generates a carry or a borrow out of the MS bit
PF	Parity flag	LS byte of the result contains an even number of 1 bits

OF	Overflow flag	two's-complement overflow
		`(a > 0 && b > 0 && t < 0) ‖`
		`(a < 0 && b < 0 && t >= 0)`

SF	Sign flag	`t < 0` (as two complement signed)
ZF	Zero flag	`t == 0`

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
  - `addq a, b` sets flags depending on t = a + b.
- 2 instruction set conditions, discard result:
  - `testq a,b` sets flags depending on t = a & b (think of it as `andq`)
  - `cmpq a, b` sets flags depending on t = b - a (think of it as a `subq`)
- Used by the `setXX` (`sete`, `setge`, …) (Conditionally set a byte (8 bit) to 0 or 1),
- Mostly used by conditional jump instructions (see next slide).

Flag	**Name**	**When is it set**
CF	Carry flag	Arithmetic operation generates a carry or a borrow out of the MS bit
PF	Parity flag	LS byte of the result contains an even number of 1 bits

OF	Overflow flag	two's-complement overflow
		`(a > 0 && b > 0 && t < 0) ||`
		`(a < 0 && b < 0 && t >= 0)`

SF	Sign flag	`t < 0` (as two complement signed)
ZF	Zero flag	`t == 0`

```
example:
    movq    $42, %rax
    movq    $12, %rdx
    cmpq    %rdx,%rax
    setge   %al
```

Condition codes

- Flags set depending on the result of the last arithmetic instruction.
  - `addq a, b` sets flags depending on `t` = `a` + `b`.
- 2 instruction set conditions, discard result:
  - `testq a,b` sets flags depending on `t` = `a` & `b` (think of it as `andq`)
  - `cmpq a, b` sets flags depending on `t` = `b` - `a` (think of it as a `subq`)
- Used by the `setXX` (`sete`, `setge`, …) (Conditionally set a byte (8 bit) to 0 or 1),
- Mostly used by conditional jump instructions (see next slide).

Flag	Name	When is it set
CF	Carry flag	Arithmetic operation generates a carry or a borrow out of the MS bit
PF	Parity flag	LS byte of the result contains an even number of 1 bits

OF	Overflow flag	two's-complement overflow
		`(a > 0 && b > 0 && t < 0) ||`
		`(a < 0 && b < 0 && t >= 0)`

SF	Sign flag	`t < 0` (as two complement signed)
ZF	Zero flag	`t == 0`

```
example:
   movq    $42, %rax
   movq    $12, %rdx
   cmpq    %rdx,%rax
   setge   %al
# al is 1 as 42 >= 12
```

Branches

(conditional `jmp label`)

Branches

(conditional `jmp label`)

Instruction	Condition	Flags
`jo`	overflow	OF = 1
`jno`	not overflow	OF = 0
`jb / jnae`	below / not above or equal	CF = 1
`jnb / jae`	not below / above or equal	CF = 0
`je / jz`	equal / zero	ZF =1
`jne / jnz`	not equal / zero	ZF = 0
`jbe / jna`	below or equal / not above	(CF OR ZF) = 1
`jnbe / ja`	neither below nor equal / above	(CF OR ZF)= 0
`js`	sign	SF = 1
`jns`	no sign	SF = 0
`jp / jpe`	parity even	PF = 1
`jnp / jpo`	parity odd	PF = 0
`jl / jnge`	less / not greater or equal	(SF XOR CF) = 1
`jnl / jge`	not less / greater or equal	(SF XOR CF) = 0
`jle / jng`	less or equal / not greater	((SF XOR OF) OR ZF) = 1
`jnle / jg`	not less or equal / greater	((SF XOR OF) OR ZF) = 0

Branches

(conditional `jmp label`)

Instruction	Condition	Flags
`jo`	overflow	OF = 1
`jno`	not overflow	OF = 0
`jb / jnae`	below / not above or equal	CF = 1
`jnb / jae`	not below / above or equal	CF = 0
`je / jz`	equal / zero	ZF =1
`jne / jnz`	not equal / zero	ZF = 0
`jbe / jna`	below or equal / not above	(CF OR ZF) = 1
`jnbe / ja`	neither below nor equal / above	(CF OR ZF)= 0
`js`	sign	SF = 1
`jns`	no sign	SF = 0
`jp / jpe`	parity even	PF = 1
`jnp / jpo`	parity odd	PF = 0
`jl / jnge`	less / not greater or equal	(SF XOR CF) = 1
`jnl / jge`	not less / greater or equal	(SF XOR CF) = 0
`jle / jng`	less or equal / not greater	((SF XOR OF) OR ZF) = 1
`jnle / jg`	not less or equal / greater	((SF XOR OF) OR ZF) = 0

Below / Above refers to unsigned
Less / Greater to 2 complement signed

Branches

(conditional `jmp label`)

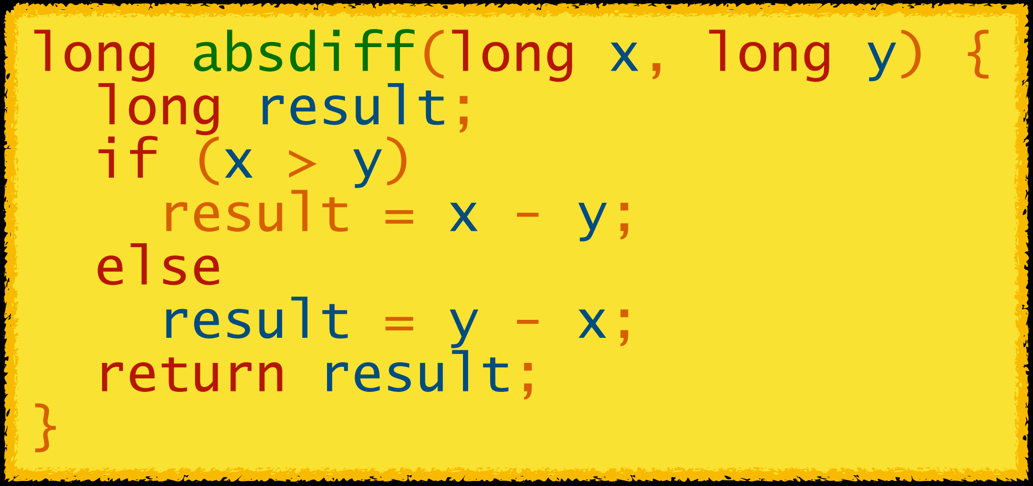
Instruction	Condition	Flags
`jo`	overflow	OF = 1
`jno`	not overflow	OF = 0
`jb / jnae`	below / not above or equal	CF = 1
`jnb / jae`	not below / above or equal	CF = 0
`je / jz`	equal / zero	ZF =1
`jne / jnz`	not equal / zero	ZF = 0
`jbe / jna`	below or equal / not above	(CF OR ZF) = 1
`jnbe / ja`	neither below nor equal / above	(CF OR ZF)= 0
`js`	sign	SF = 1
`jns`	no sign	SF = 0
`jp / jpe`	parity even	PF = 1
`jnp / jpo`	parity odd	PF = 0
`jl / jnge`	less / not greater or equal	(SF XOR CF) = 1
`jnl / jge`	not less / greater or equal	(SF XOR CF) = 0
`jle / jng`	less or equal / not greater	((SF XOR OF) OR ZF) = 1
`jnle / jg`	not less or equal / greater	((SF XOR OF) OR ZF) = 0

Below / Above refers to unsigned
Less / Greater to 2 complement signed

```
example:
   movq  $0,   %rax
   testq %rdi, %rdi
   jz end
   movq  $42,  %rax
end:
# what will %rax contain ?
```

if else, switch case

if else, switch case



```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

if else, switch case

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

x in %rdi,
y in %rsi,
result in %rax

if else, switch case

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

x in %rdi,
y in %rsi,
result in %rax

```
absdiff:
   cmpq %rsi, %rdi
   # set flags for x-y
   jle .L4
   movq %rdi, %rax
   subq %rsi, %rax
   ret
.L4: # x <= y
   movq %rsi, %rax
   subq %rdi, %rax
   ret
```

if else, switch case

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

x in %rdi,
y in %rsi,
result in %rax

```
absdiff:
   cmpq %rsi, %rdi
   # set flags for x-y
   jle .L4
   movq %rdi, %rax
   subq %rsi, %rax
   ret
.L4: # x <= y
   movq %rsi, %rax
   subq %rdi, %rax
   ret
```

```
long switch_eg(long x, long y, long z) {
  long w = 1;
  switch(x) {
  case 1:
    w = y * z;
    break;
  case 2:
    w = y / z; /* Fall Through */
  case 3:
    w += z;
    break;
  case 5:
  case 6:
    w -= z;
    break;
  default:
    w = 2;
  }
  return w;
}
```

if else, switch case

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

x in %rdi,
y in %rsi,
result in %rax

```
absdiff:
   cmpq %rsi, %rdi
   # set flags for x-y
   jle .L4
   movq %rdi, %rax
   subq %rsi, %rax
   ret
.L4: # x <= y
   movq %rsi, %rax
   subq %rdi, %rax
   ret
```

```
long switch_eg(long x, long y, long z) {
  long w = 1;
  switch(x) {
  case 1:
    w = y * z;
    break;
  case 2:
    w = y / z; /* Fall Through */
  case 3:
    w += z;
    break;
  case 5:
  case 6:
    w -= z;
    break;
  default:
    w = 2;
  }
  return w;
}
```

x in %rdi,
y in %rsi,
z in %rdx,
result in %rax

if else, switch case

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

x in %rdi,
y in %rsi,
result in %rax

```
absdiff:
   cmpq %rsi, %rdi
   # set flags for x-y
   jle .L4
   movq %rdi, %rax
   subq %rsi, %rax
   ret
.L4: # x <= y
   movq %rsi, %rax
   subq %rdi, %rax
   ret
```

```
long switch_eg(long x, long y, long z) {
  long w = 1;
  switch(x) {
  case 1:
    w = y * z;
    break;
  case 2:
    w = y / z; /* Fall Through */
  case 3:
    w += z;
    break;
  case 5:
  case 6:
    w -= z;
    break;
  default:
    w = 2;
  }
  return w;
}
```

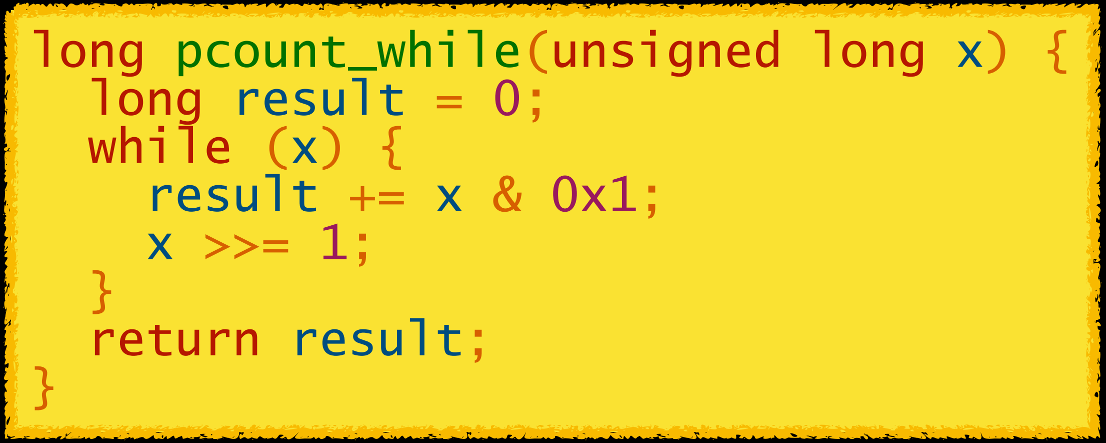
x in %rdi,
y in %rsi,
z in %rdx,
result in %rax

```
switch_eg:
  movq %rdx, %rcx
  cmpq $6, %rdi # x:6
  ja .L8 # Use default
  jmp *.L4(,%rdi,8)
# … code for each case.
.Li: # in in 3,5,7,8,9
  …
```

```
.section .rodata .align 8
.L4:
  .quad .L8 #x == 0
  .quad .L3 #x == 1
  .quad .L5 #x == 2
  .quad .L9 #x == 3
  .quad .L8 #x == 4
  .quad .L7 #x == 5
  .quad .L7 #x == 6
```

Loops

Loops



```
long pcount_while(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Loops

```
long pcount_while(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

```
long pcount_goto_jtm(unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if (x)
    goto loop;
  return result;
}
```

Loops

```
long pcount_while(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

```
long pcount_goto_jtm(unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if (x)
    goto loop;
  return result;
}
```

```
long pcount_goto_dw(unsigned long x) {
  long result = 0;
  if (!x)
    goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

Loops

```c
long pcount_while(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

```c
long pcount_goto_jtm(unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if (x)
    goto loop;
  return result;
}
```

```
pcount_goto_jtm:
  movl  $0,   %eax
  jmp   .L2
.L3:
  movq  %rdi, %rdx
  andl  $1,   %edx
  addq  %rdx, %rax
  shrq  %rdi
.L2:
  testq %rdi, %rdi
  jne   .L3
  rep ret
```

```c
long pcount_goto_dw(unsigned long x) {
  long result = 0;
  if (!x)
    goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

Loops

```
long pcount_while(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

```
long pcount_goto_jtm(unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if (x)
    goto loop;
  return result;
}
```

```
pcount_goto_jtm:
  movl  $0,   %eax
  jmp   .L2
.L3:
  movq  %rdi, %rdx
  andl  $1,   %edx
  addq  %rdx, %rax
  shrq  %rdi
.L2:
  testq %rdi, %rdi
  jne   .L3
  rep ret
```

```
long pcount_goto_dw(unsigned long x) {
  long result = 0;
  if (!x)
    goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

```
pcount_goto_dw:
  testq %rdi, %rdi
  je    .L4
  movl  $0,   %eax
.L3:
  movq  %rdi, %rdx
  andl  $1,   %edx
  addq  %rdx, %rax
  shrq  %rdi
  jne   .L3
  rep ret
.L4:
  movl  $0,   %eax
  ret
```

an often abused instruction: `leaq`

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
  - `Src` is an address mode expression
  - `Dest` is a register, to be set to the address denoted by expression

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
  - `Src` is an address mode expression
  - `Dest` is a register, to be set to the address denoted by expression
- Used to compute address (instead of accessing them),
  - e.g translating `p = &x[i];`

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
  - `Src` is an address mode expression
  - `Dest` is a register, to be set to the address denoted by expression
- Used to compute address (instead of accessing them),
  - e.g translating `p = &x[i];`
- Does **not** set condition codes,

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
  - `Src` is an address mode expression
  - `Dest` is a register, to be set to the address denoted by expression
- Used to compute address (instead of accessing them),
  - e.g translating `p = &x[i];`
- Does **not** set condition codes,
- Can be used to compute expressions of the form x + k*y for k in {1,2,4,8}

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
  - `Src` is an address mode expression
  - `Dest` is a register, to be set to the address denoted by expression
- Used to compute address (instead of accessing them),
  - e.g translating `p = &x[i];`
- Does **not** set condition codes,
- Can be used to compute expressions of the form x + k*y for k in {1,2,4,8}

```
long m12(long x)
{
  return x*12;
}
```

an often abused instruction: `leaq`

- Load effective address: `leaq Src, Dest`:
    - `Src` is an address mode expression
    - `Dest` is a register, to be set to the address denoted by expression
- Used to compute address (instead of accessing them),
    - e.g translating `p = &x[i];`
- Does **not** set condition codes,
- Can be used to compute expressions of the form x + k*y for k in {1,2,4,8}

```
long m12(long x)
{
  return x*12;
}
```

```
m12:
    leaq (%rdi,%rdi,2), %rax # t <- x+x*2
    salq $2, %rax # return t<<2
    ret
```

The stack

The stack

- `%rsp` is special. It points to a location in memory called *the stack*.

`%rsp` →

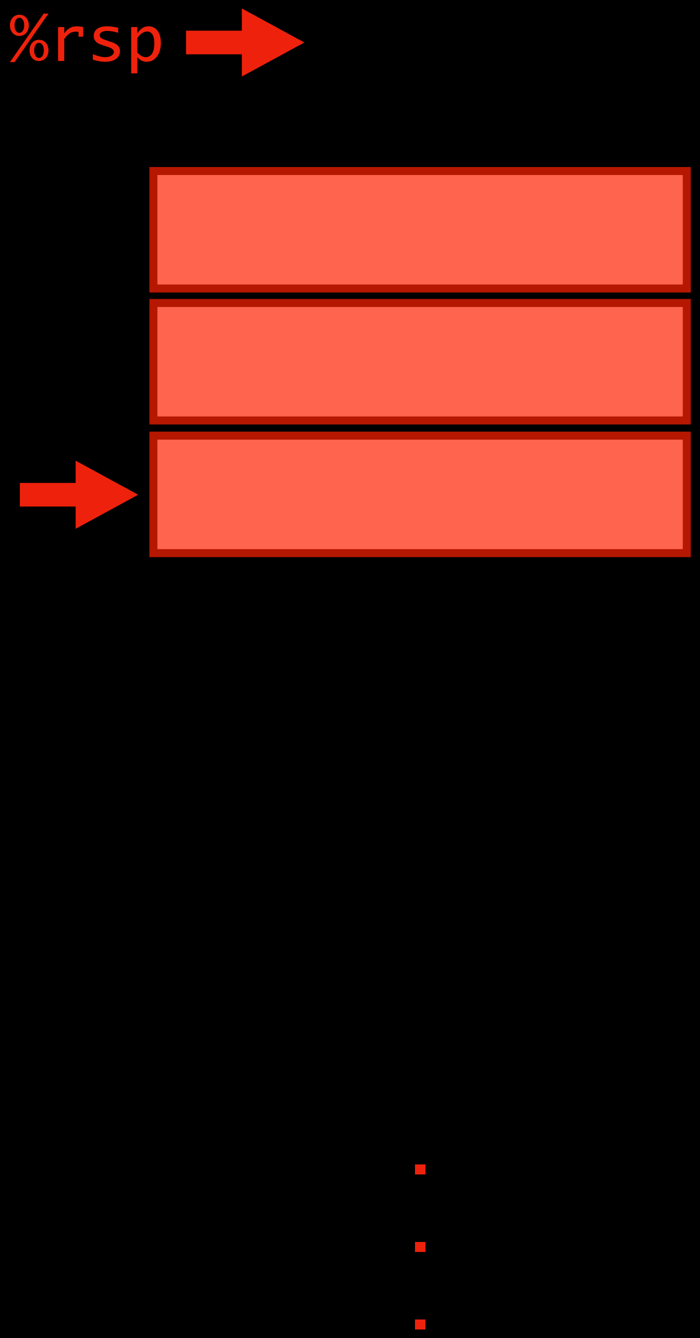
The stack

- `%rsp` is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : `pushq` and `popq`

`%rsp` →

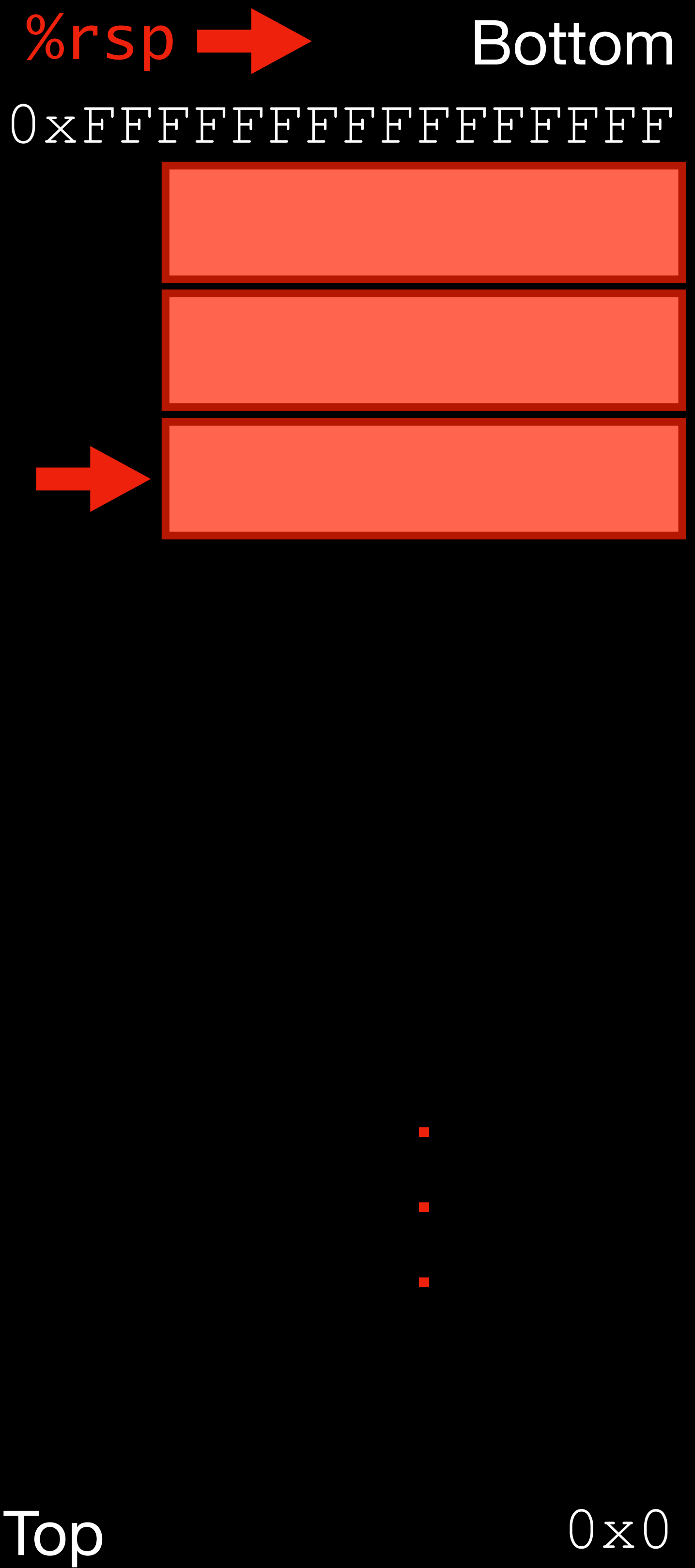
The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq



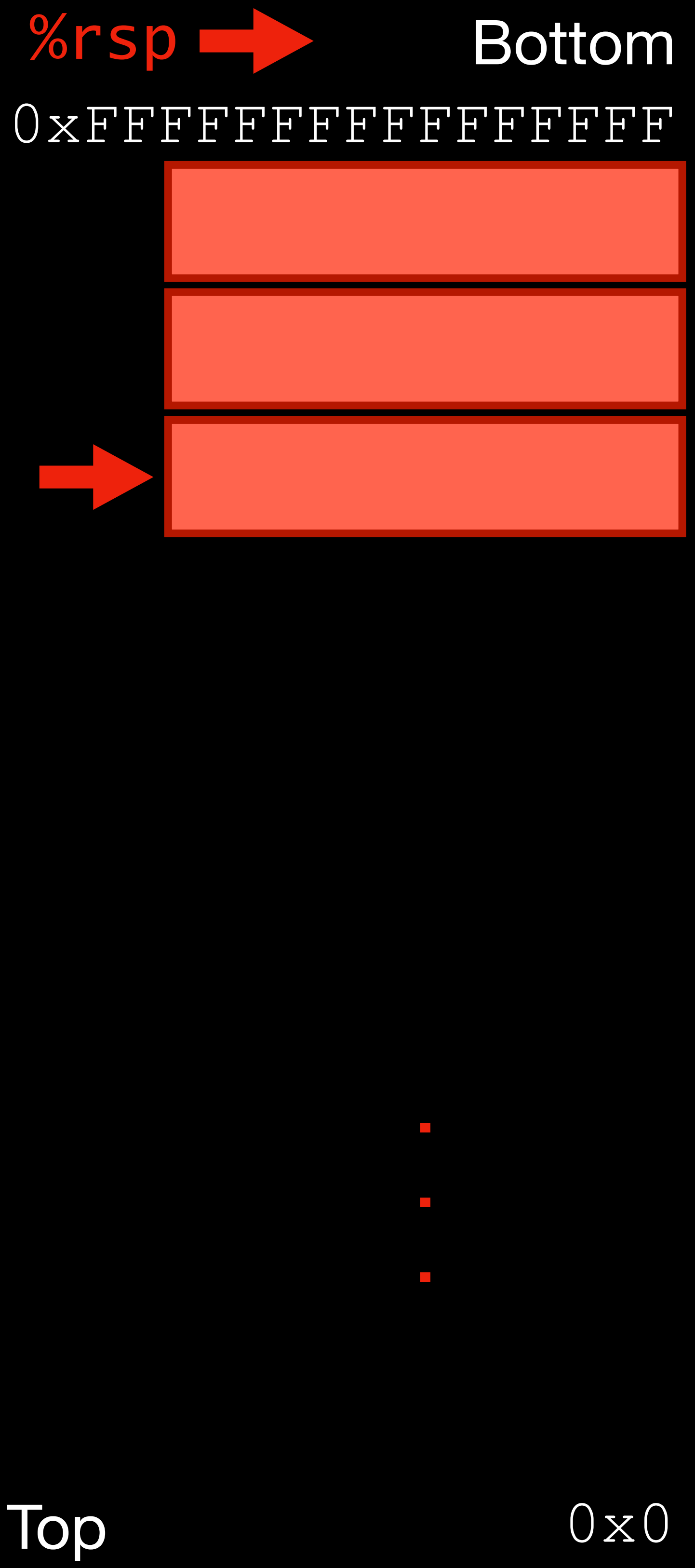
The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**



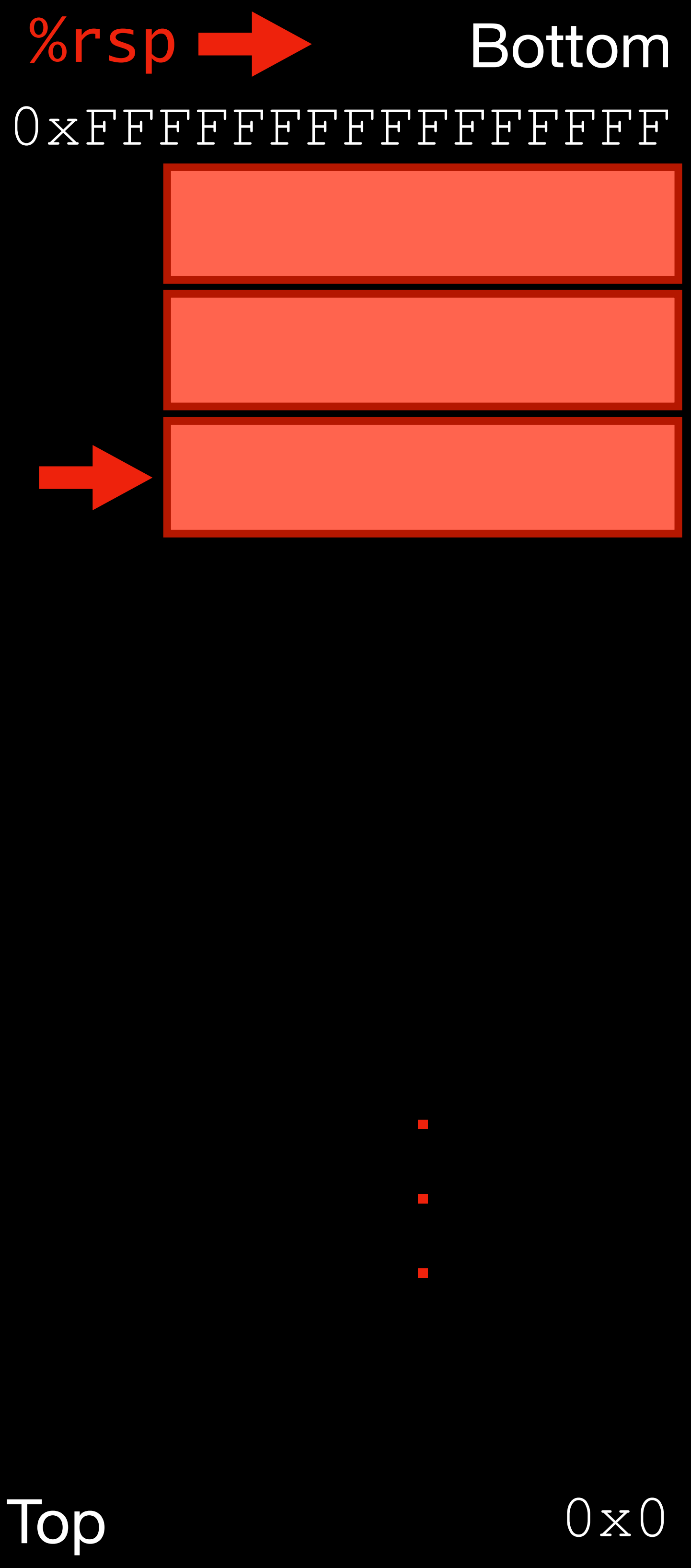
The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
    1. %rsp -= 8
    2. write the src at (%rsp)



The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
  1. %rsp -= 8
  2. write the src at (%rsp)
- popq dest
  1. read value from (%rsp) to dest
  2. %rsp += 8.

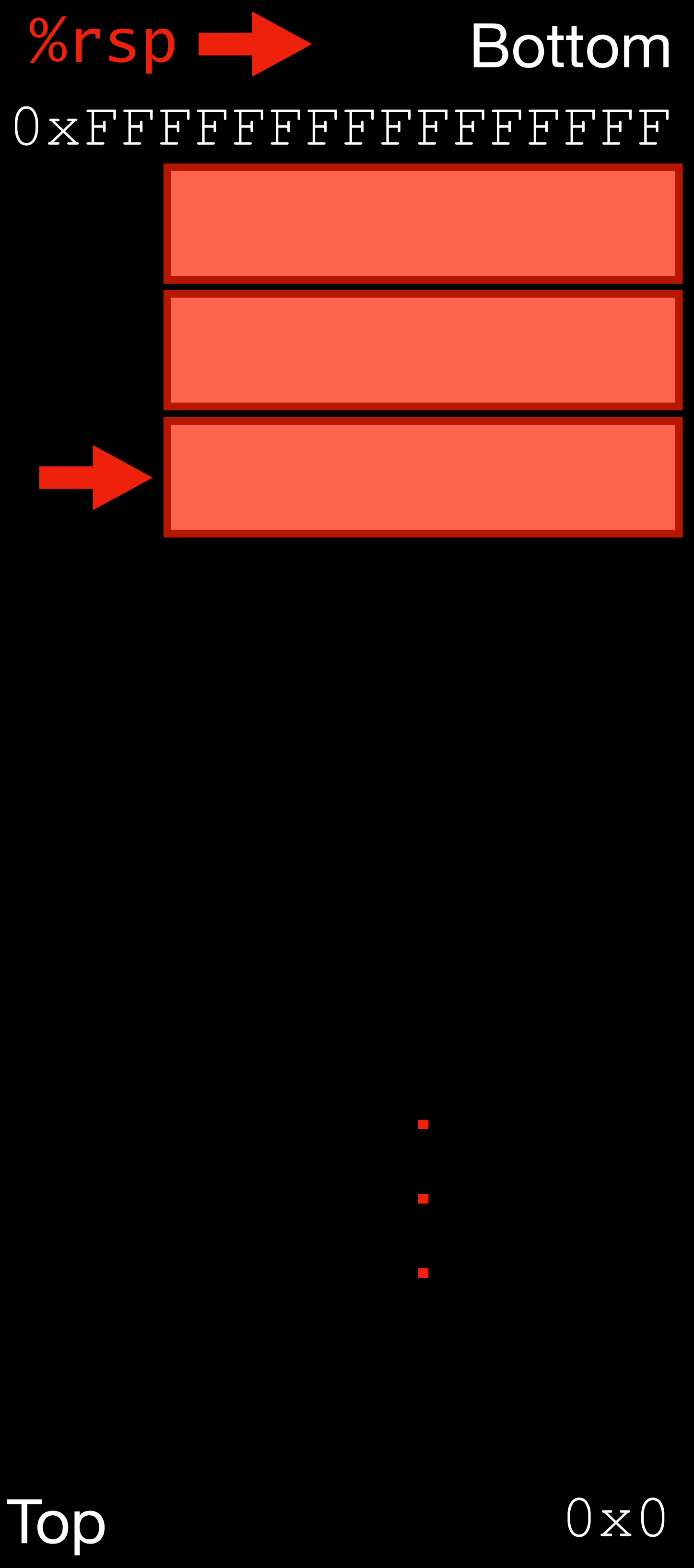


The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
    1. %rsp -= 8
    2. write the src at (%rsp)
- popq dest
    1. read value from (%rsp) to dest
    2. %rsp += 8.

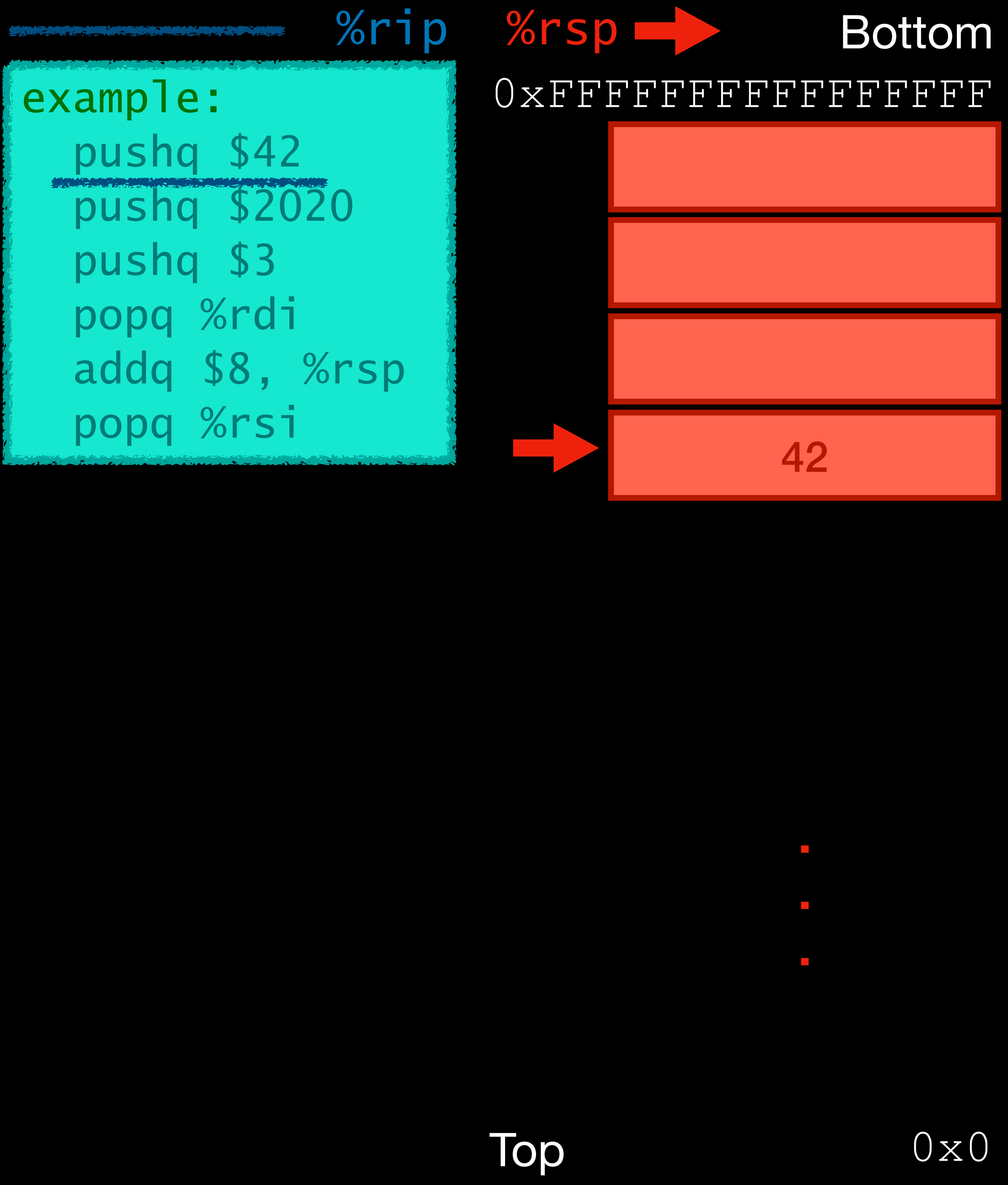
%rip

```
example:
    pushq $42
    pushq $2020
    pushq $3
    popq %rdi
    addq $8, %rsp
    popq %rsi
```



The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
    1. %rsp -= 8
    2. write the src at (%rsp)
- popq dest
    1. read value from (%rsp) to dest
    2. %rsp += 8.



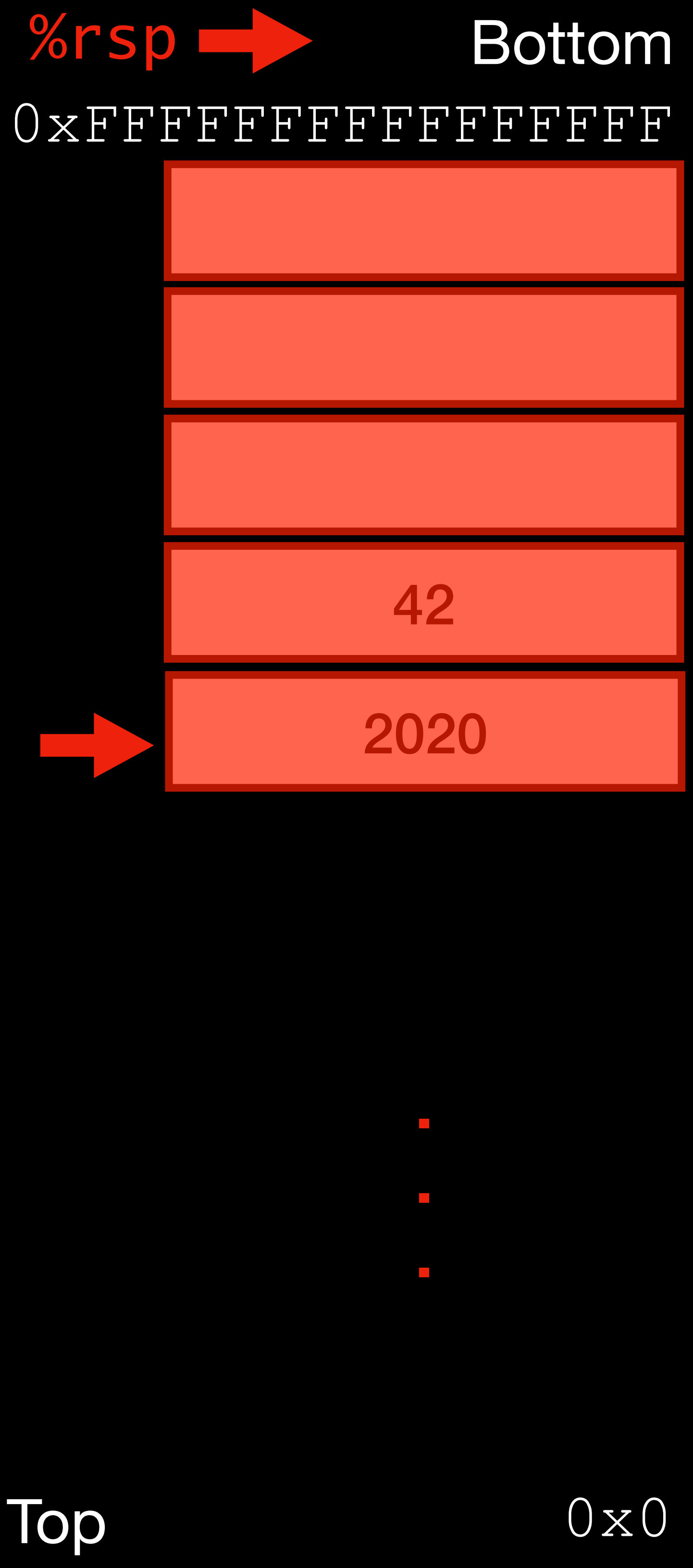
```
example:
   pushq $42
   pushq $2020
   pushq $3
   popq %rdi
   addq $8, %rsp
   popq %rsi
```

The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
  1. %rsp -= 8
  2. write the src at (%rsp)
- popq dest
  1. read value from (%rsp) to dest
  2. %rsp += 8.

%rip

```
example:
   pushq $42
   pushq $2020
   pushq $3
   popq %rdi
   addq $8, %rsp
   popq %rsi
```

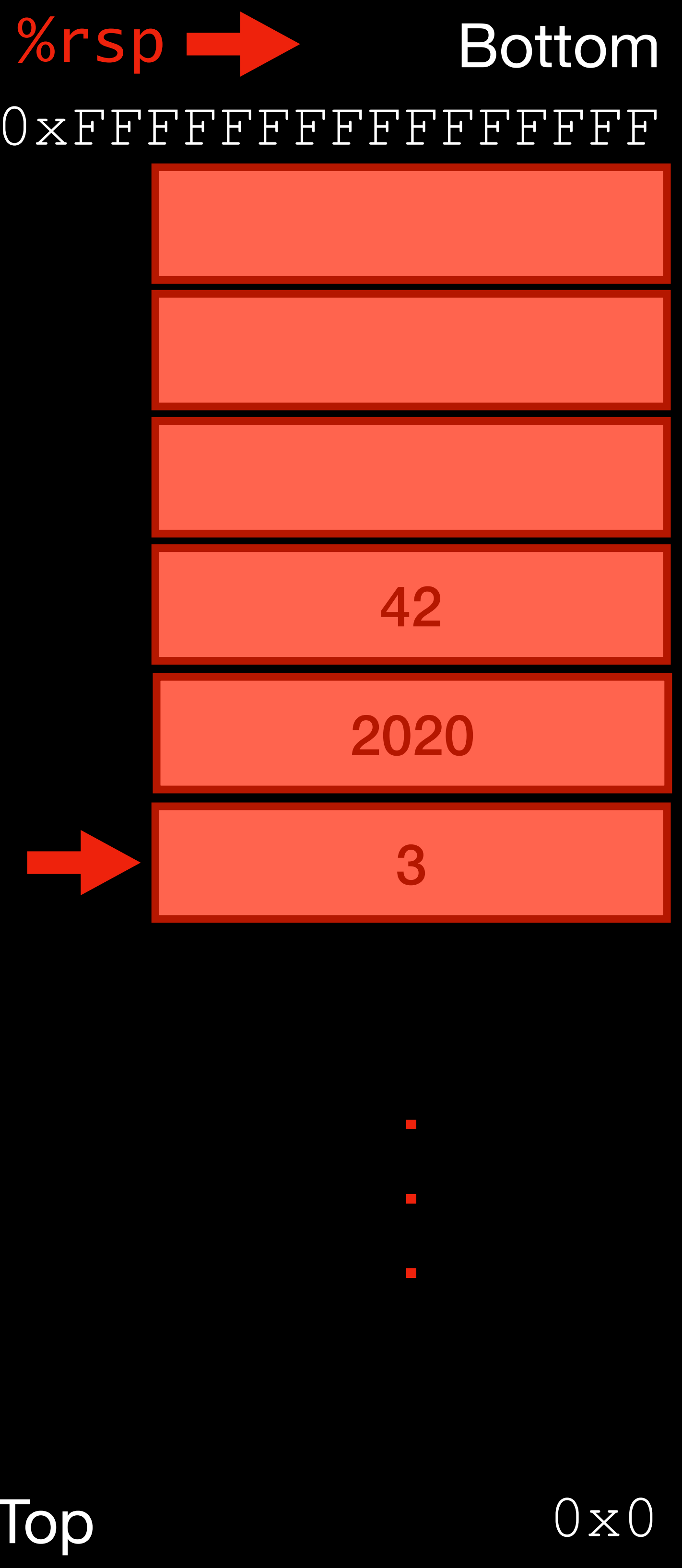


The stack

- `%rsp` is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : `pushq` and `popq`
- The x86 stack grows **down**
- `pushq src` :
  1. `%rsp` -= 8
  2. write the `src` at `(%rsp)`
- `popq dest`
  1. read value from `(%rsp)` to `dest`
  2. `%rsp` += 8.

%rip

```
example:
  pushq $42
  pushq $2020
  pushq $3
  popq %rdi
  addq $8, %rsp
  popq %rsi
```

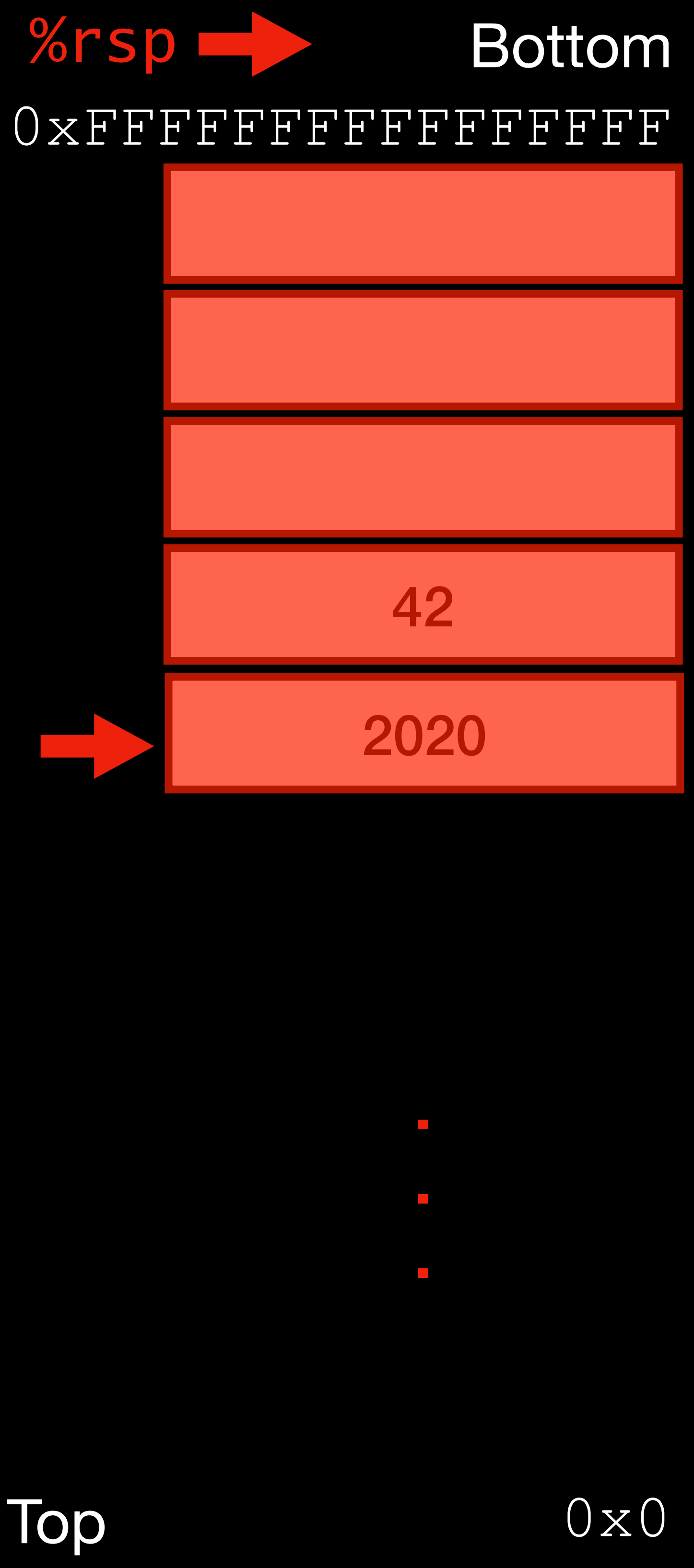


The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
  1. %rsp -= 8
  2. write the src at (%rsp)
- popq dest
  1. read value from (%rsp) to dest
  2. %rsp += 8.

%rip

```
example:
  pushq $42
  pushq $2020
  pushq $3
  popq %rdi
  addq $8, %rsp
  popq %rsi
```



%rdi = 3

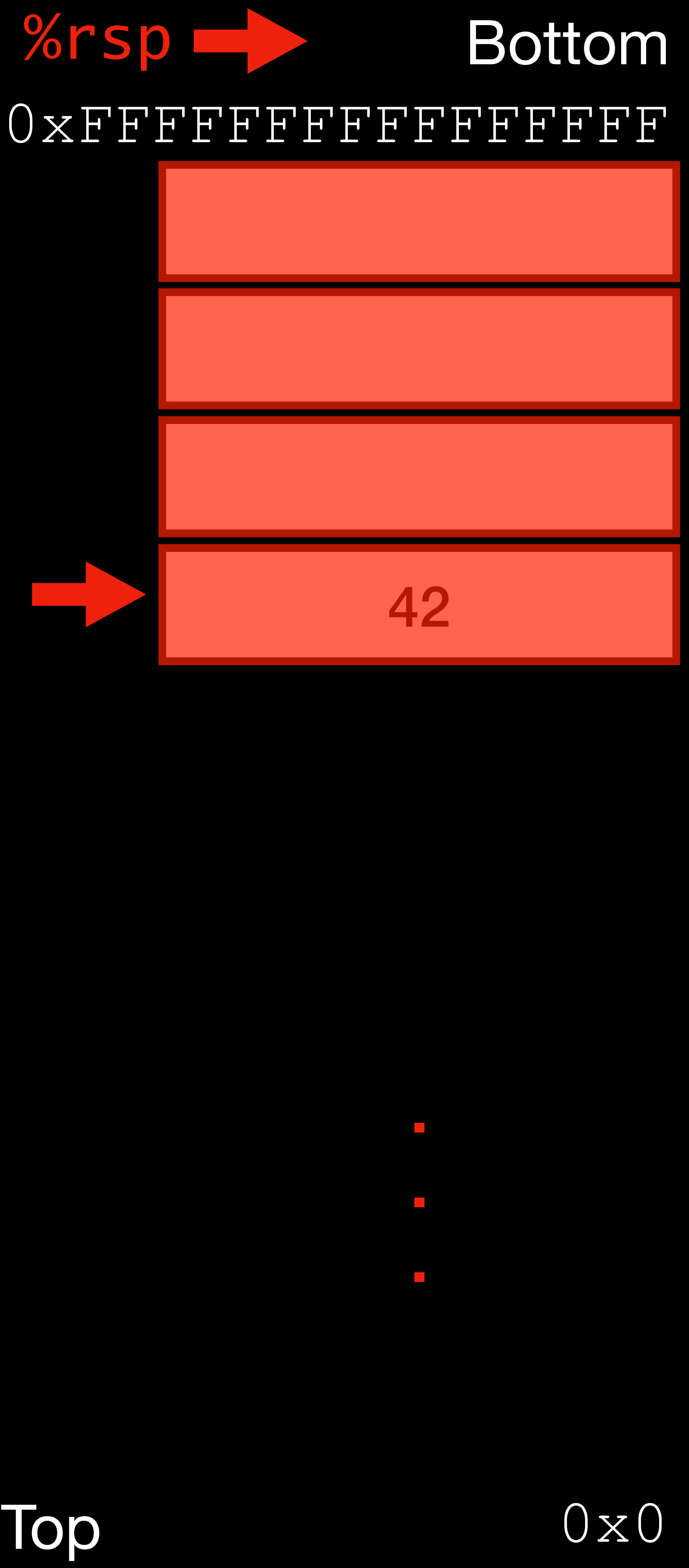
The stack

- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
  1. %rsp -= 8
  2. write the src at (%rsp)
- popq dest
  1. read value from (%rsp) to dest
  2. %rsp += 8.

%rip

```
example:
   pushq $42
   pushq $2020
   pushq $3
   popq %rdi
   addq $8, %rsp
   popq %rsi
```

%rdi = 3



The stack

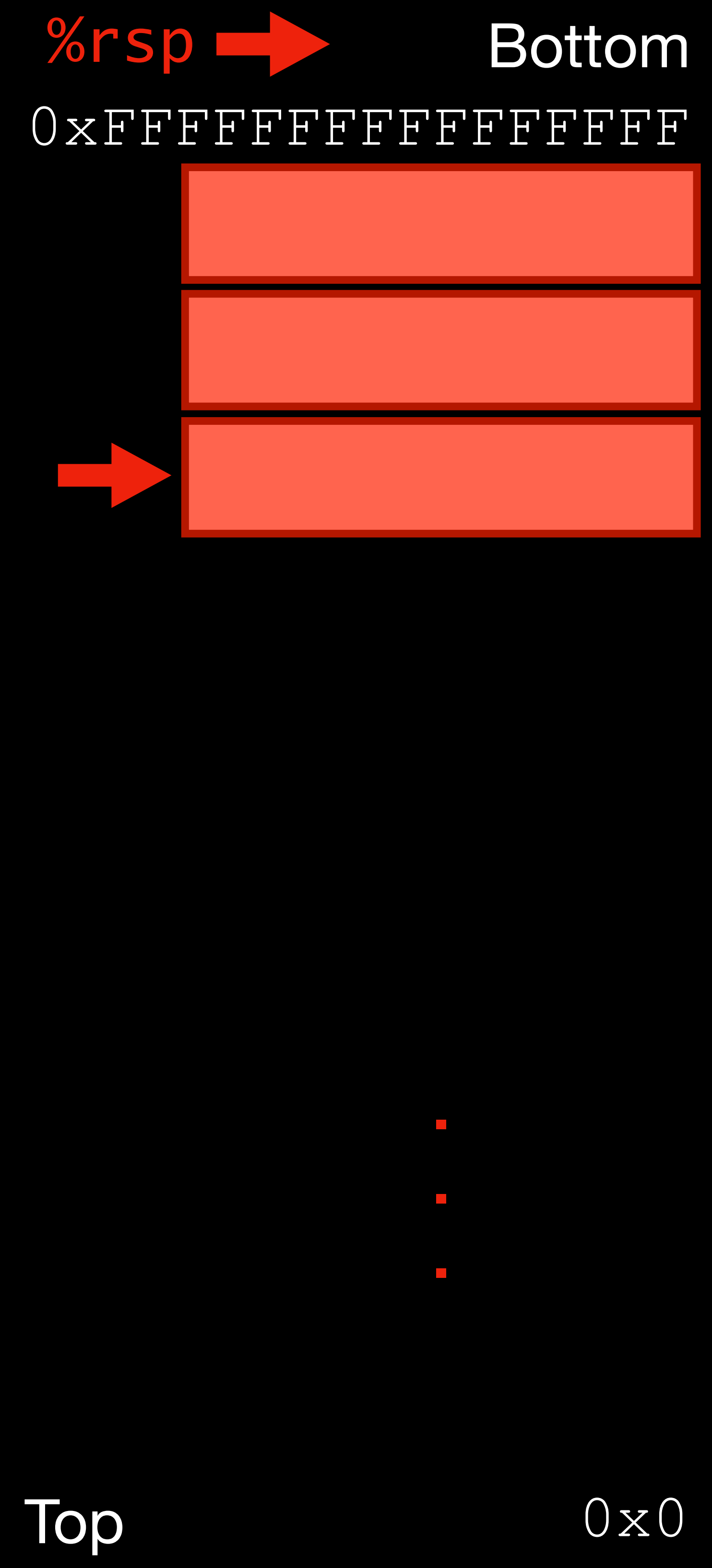
- `%rsp` is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : `pushq` and `popq`
- The x86 stack grows **down**
- `pushq src` :
  1. `%rsp` -= 8
  2. write the `src` at `(%rsp)`
- `popq dest`
  1. read value from `(%rsp)` to `dest`
  2. `%rsp` += 8.

%rip

```
example:
   pushq $42
   pushq $2020
   pushq $3
   popq %rdi
   addq $8, %rsp
   popq %rsi
```

`%rsi = 42`

`%rdi = 3`



The stack

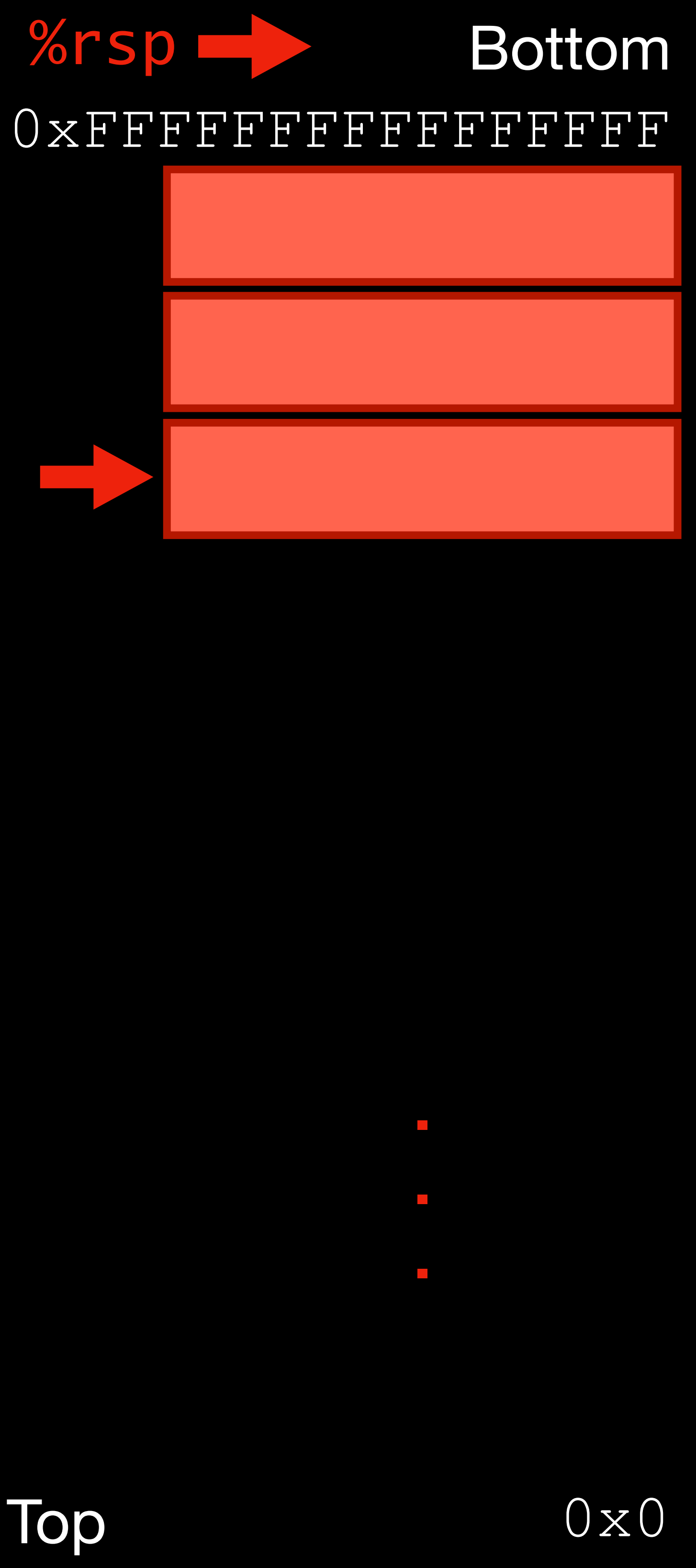
- %rsp is special. It points to a location in memory called *the stack*.
- A stack is a LIFO structure. The x86 stack is implemented using to instructions : pushq and popq
- The x86 stack grows **down**
- pushq src :
  1. %rsp -= 8
  2. write the src at (%rsp)
- popq dest
  1. read value from (%rsp) to dest
  2. %rsp += 8.
- You can also
  - drop a value with addq $8, %rsp,
  - make room for values using subq, and access them with D(%rsp).

%rip

```
example:
   pushq $42
   pushq $2020
   pushq $3
   popq %rdi
   addq $8, %rsp
   popq %rsi
```

%rsi = 42

%rdi = 3



Building functions

Building functions

- Problem:
    - Jump to function code
    - Go back where you came from ?

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)

Building functions

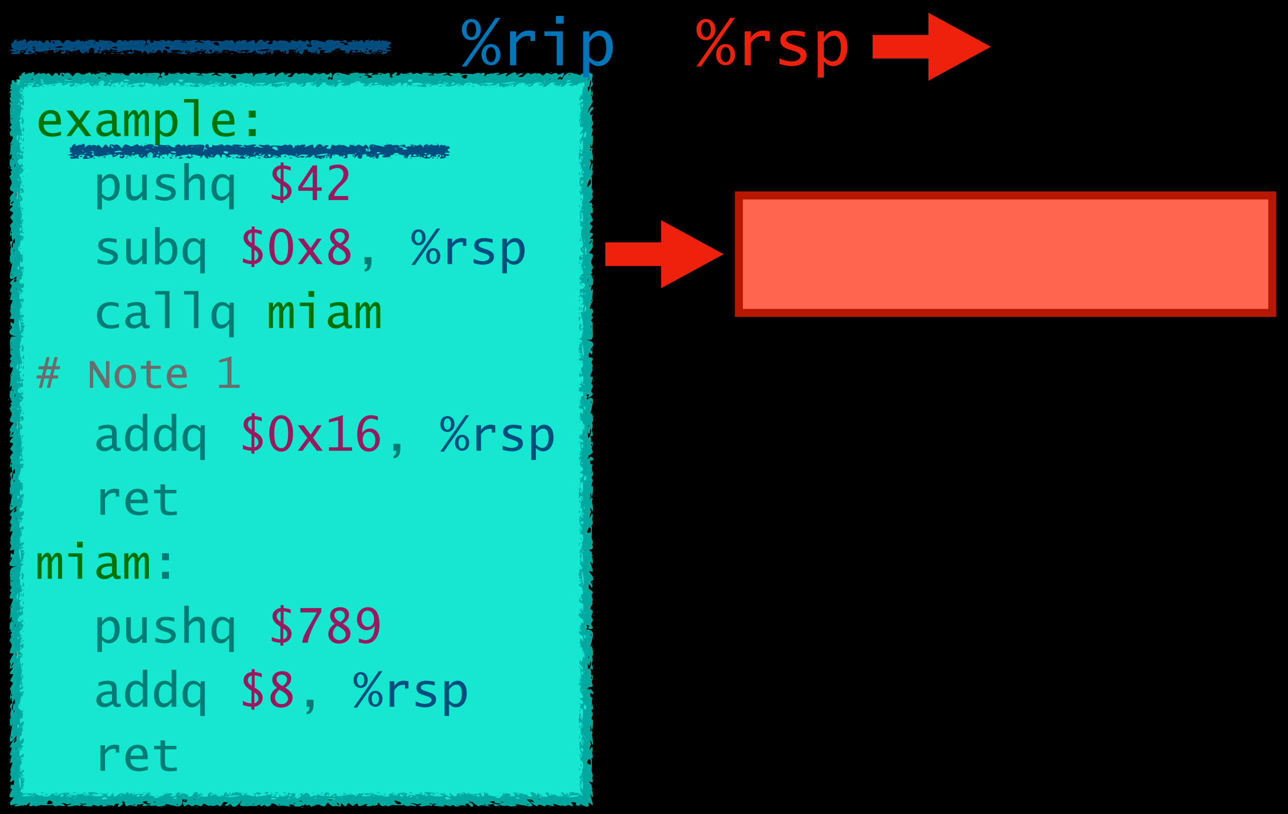
- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`

Building functions

- Problem:
    - Jump to function code
    - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
    1. push the return address
    2. jump to the function code
- The `retq` instruction is used at the end of the function
    1. pop the return address
    2. jump back to it.
    3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

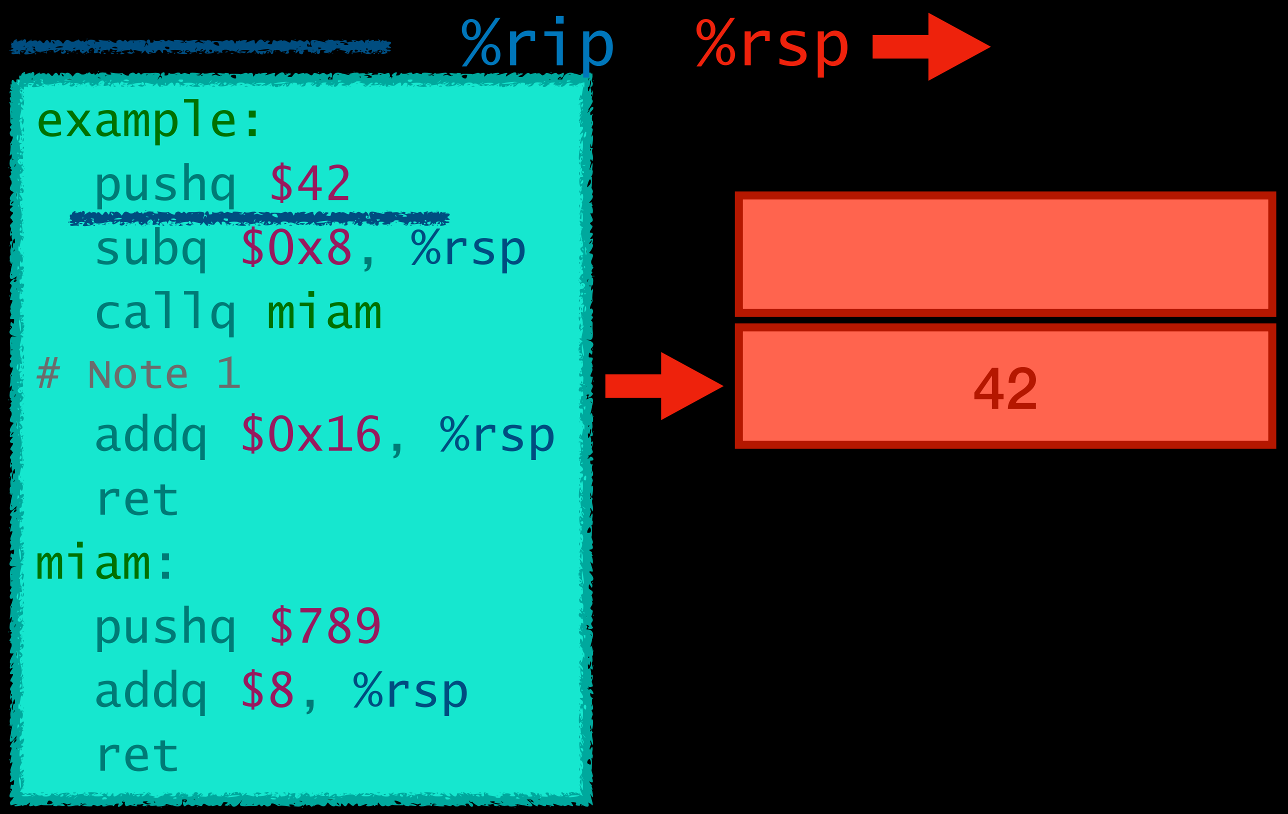


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

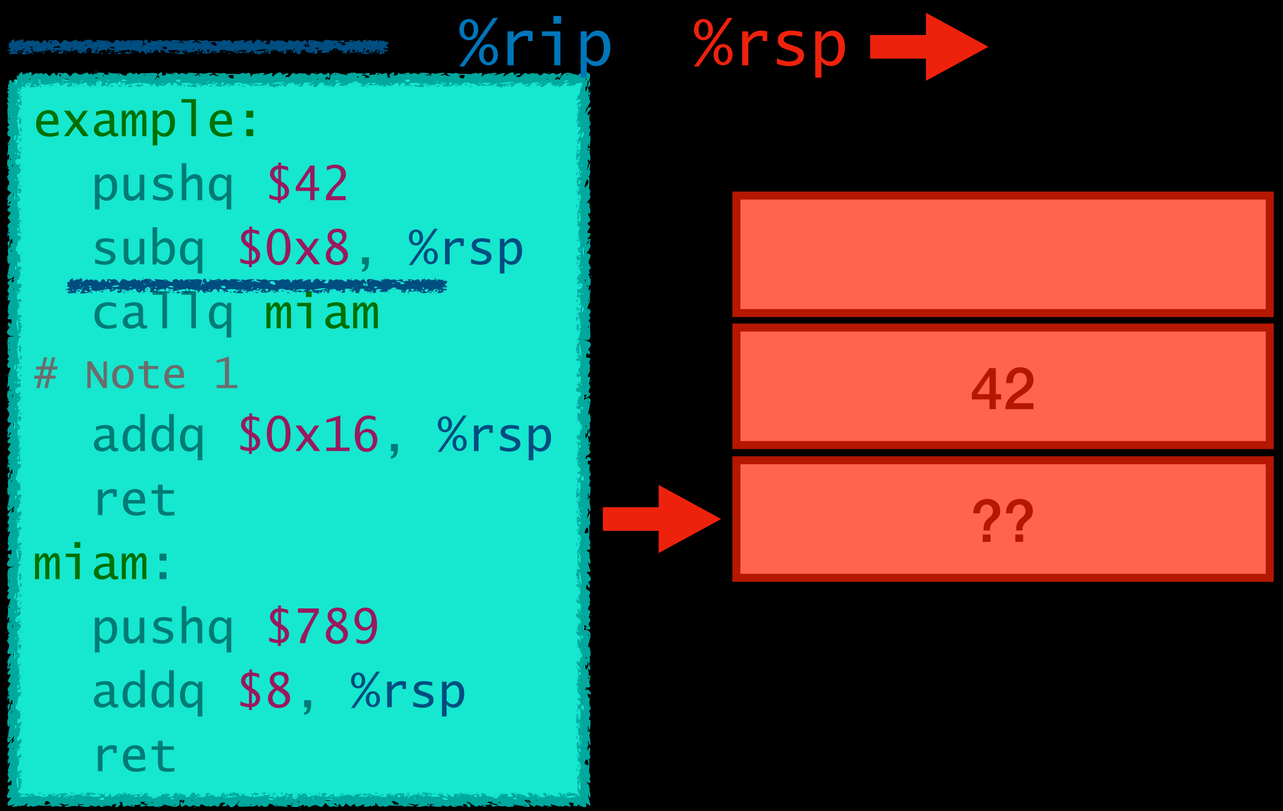


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

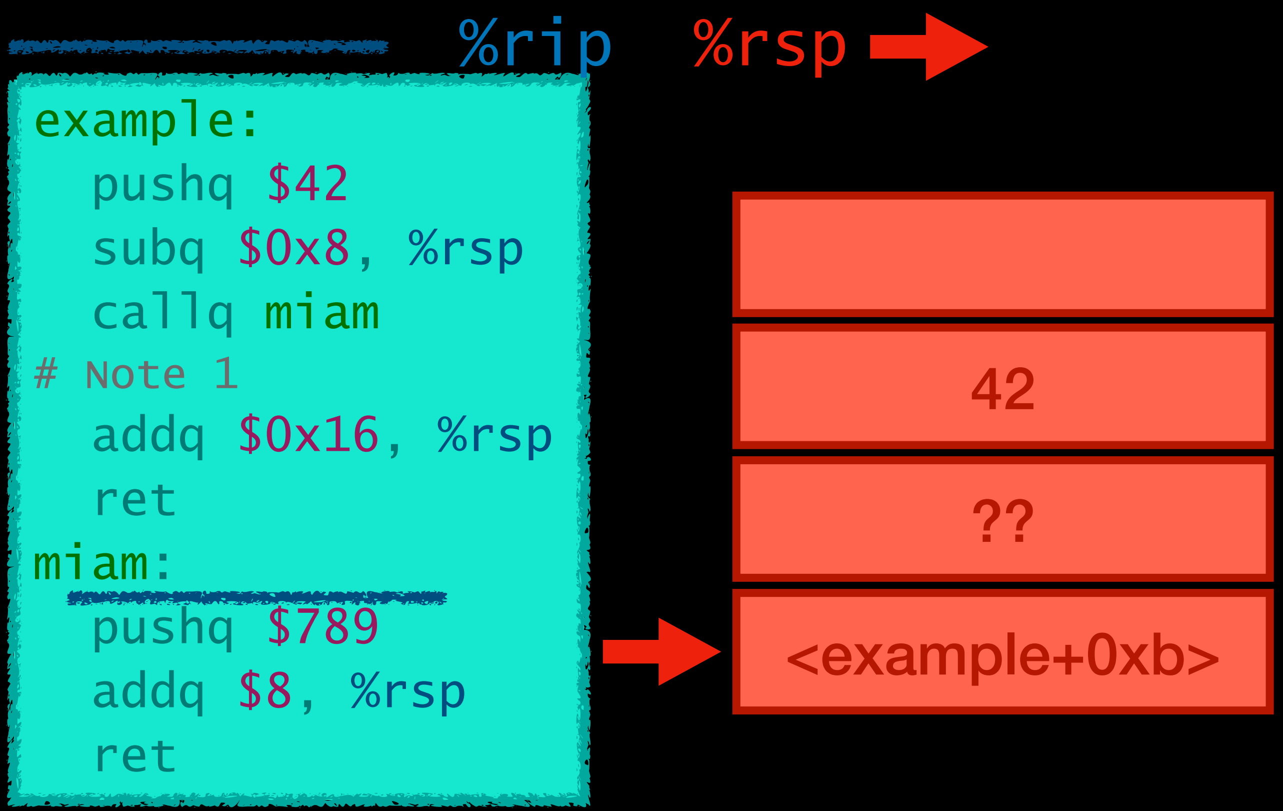


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

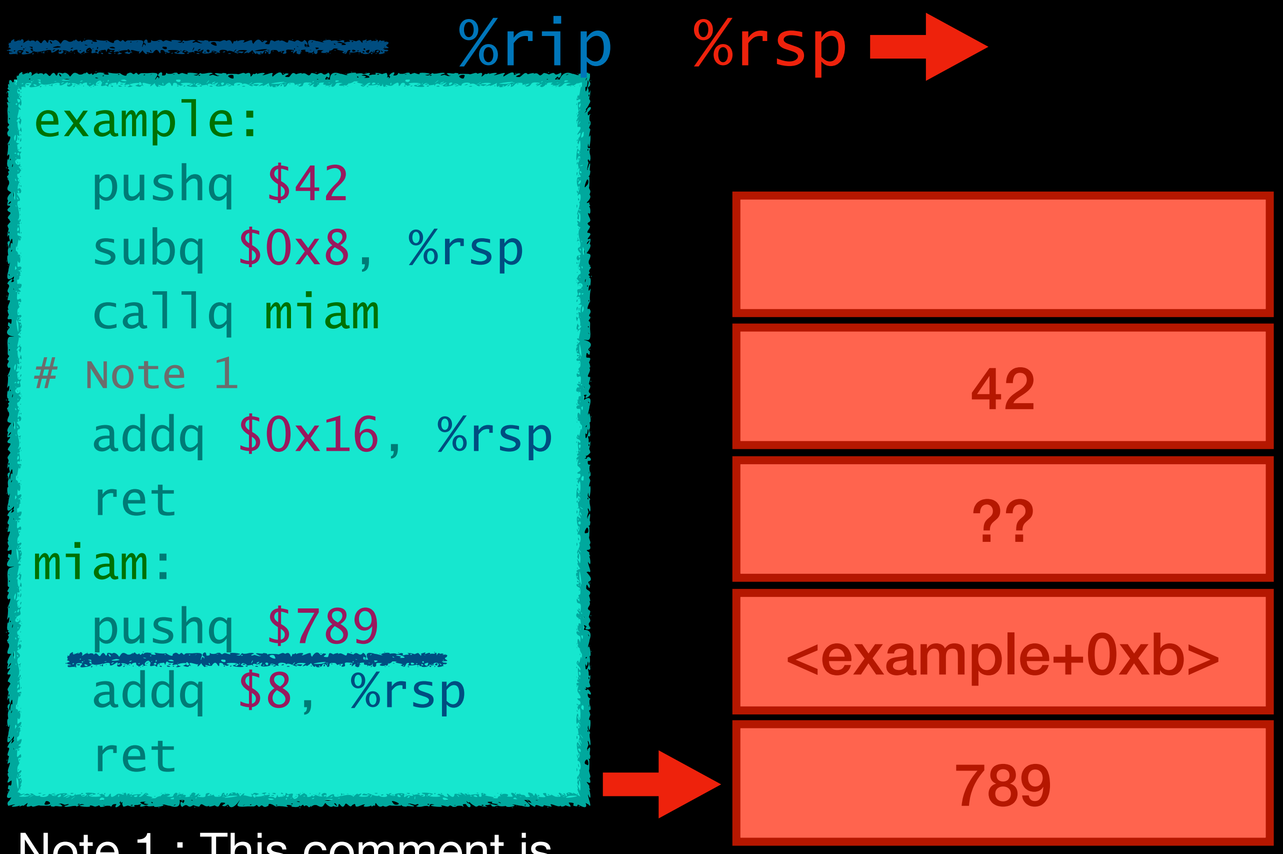


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

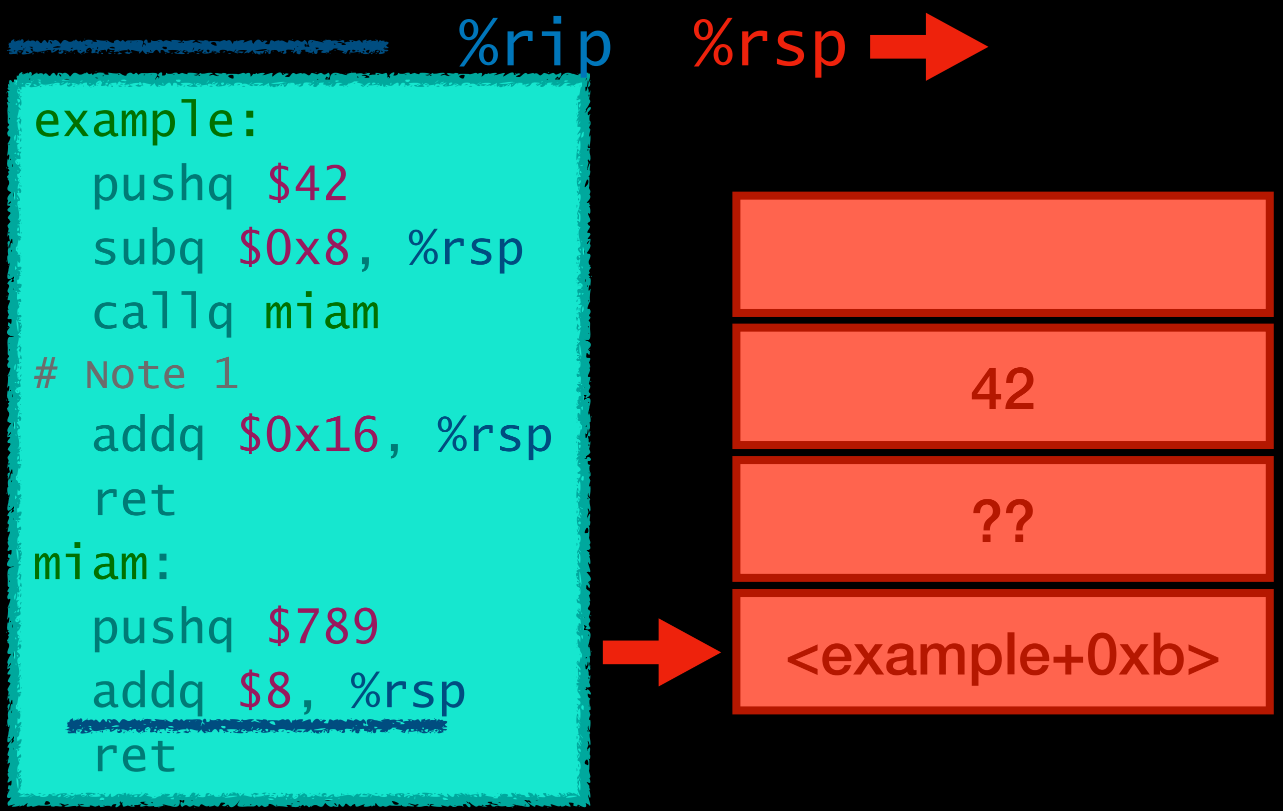


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

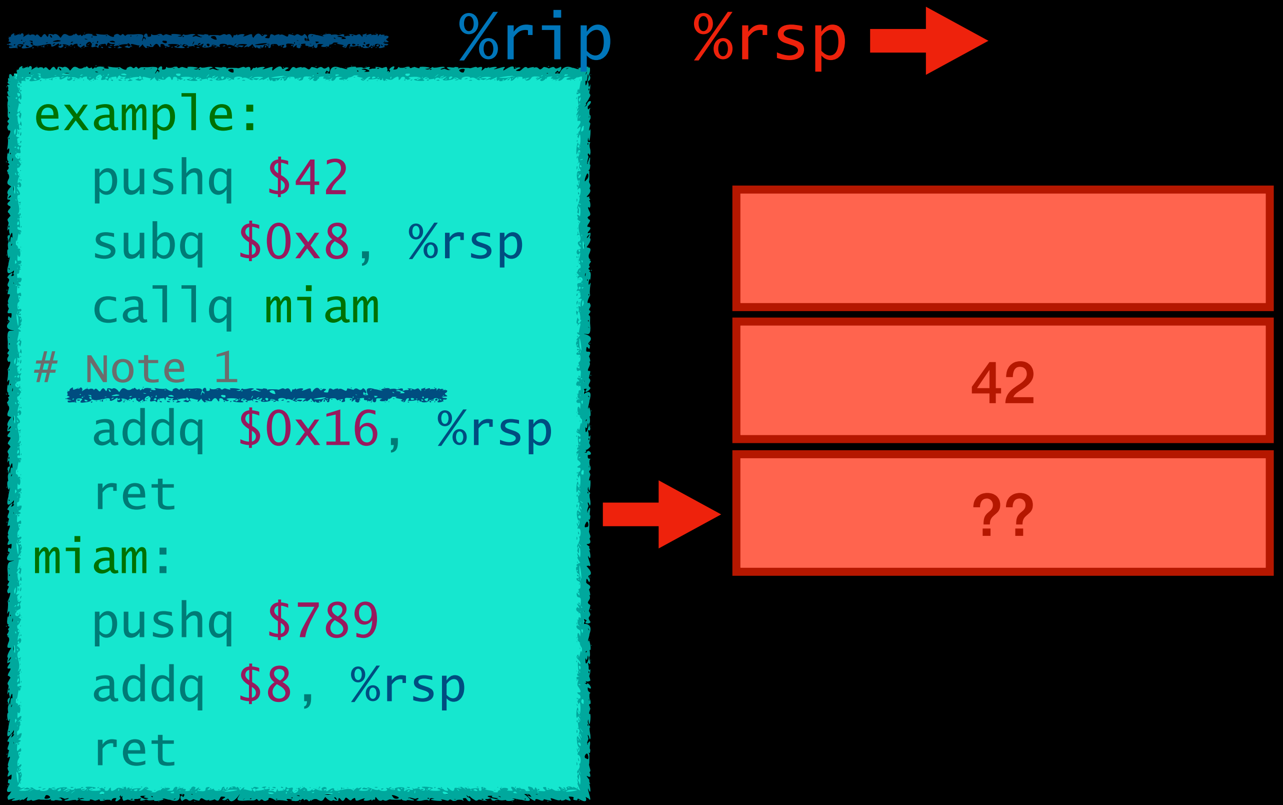


```
example:
   pushq $42
   subq $0x8, %rsp
   callq miam
# Note 1
   addq $0x16, %rsp
   ret
miam:
   pushq $789
   addq $8, %rsp
   ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`

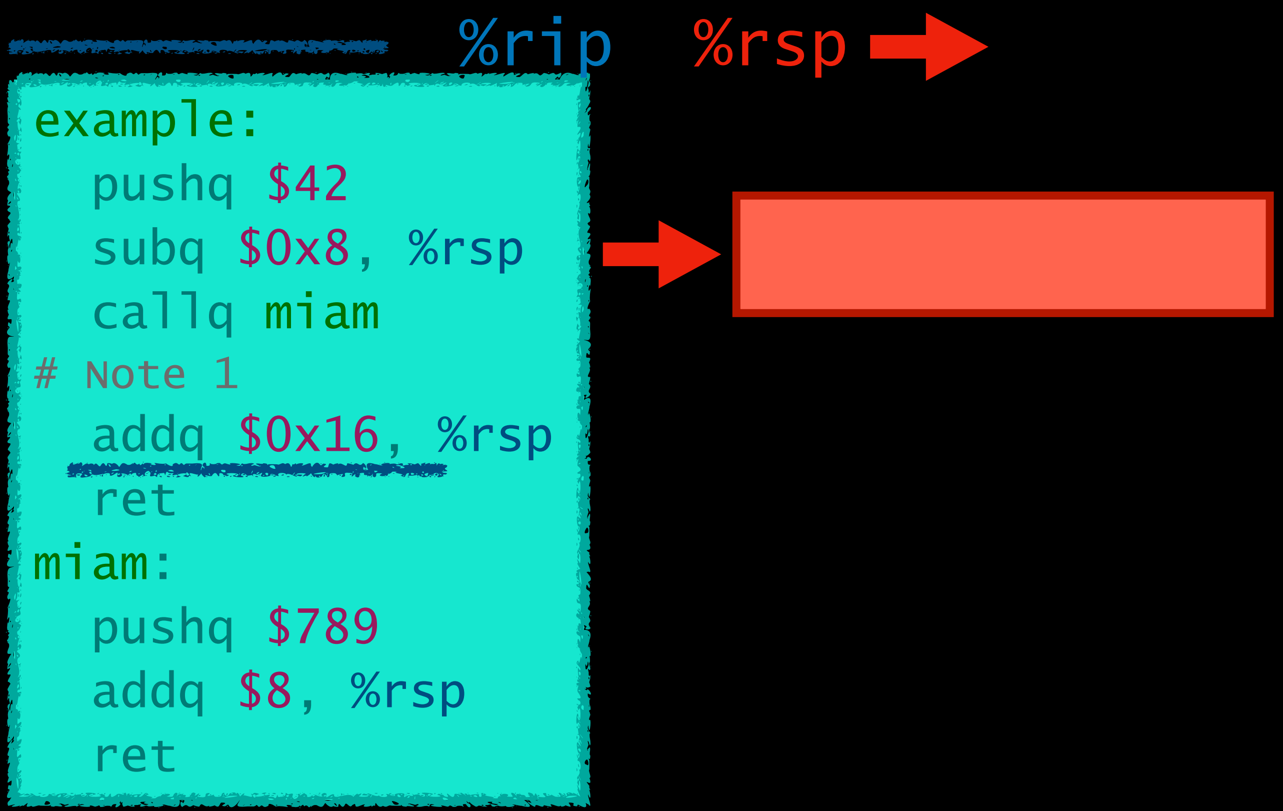


```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

Building functions

- Problem:
  - Jump to function code
  - Go back where you came from ?
- Solution: Use the stack !
- A `callq` instruction
  1. push the return address
  2. jump to the function code
- The `retq` instruction is used at the end of the function
  1. pop the return address
  2. jump back to it.
  3. (That can be though of as `popq %rip`)
- Functions can modify `%rsp` to use the stack but have to restore it to the correct value before `ret`
- Make sure you balance the `pushq` and `subq $8` with the `popq` and `addq $8` to `%rsp`



```
example:
  pushq $42
  subq $0x8, %rsp
  callq miam
# Note 1
  addq $0x16, %rsp
  ret
miam:
  pushq $789
  addq $8, %rsp
  ret
```

Note 1 : This comment is 11 (0xb) machine code bytes in `example`

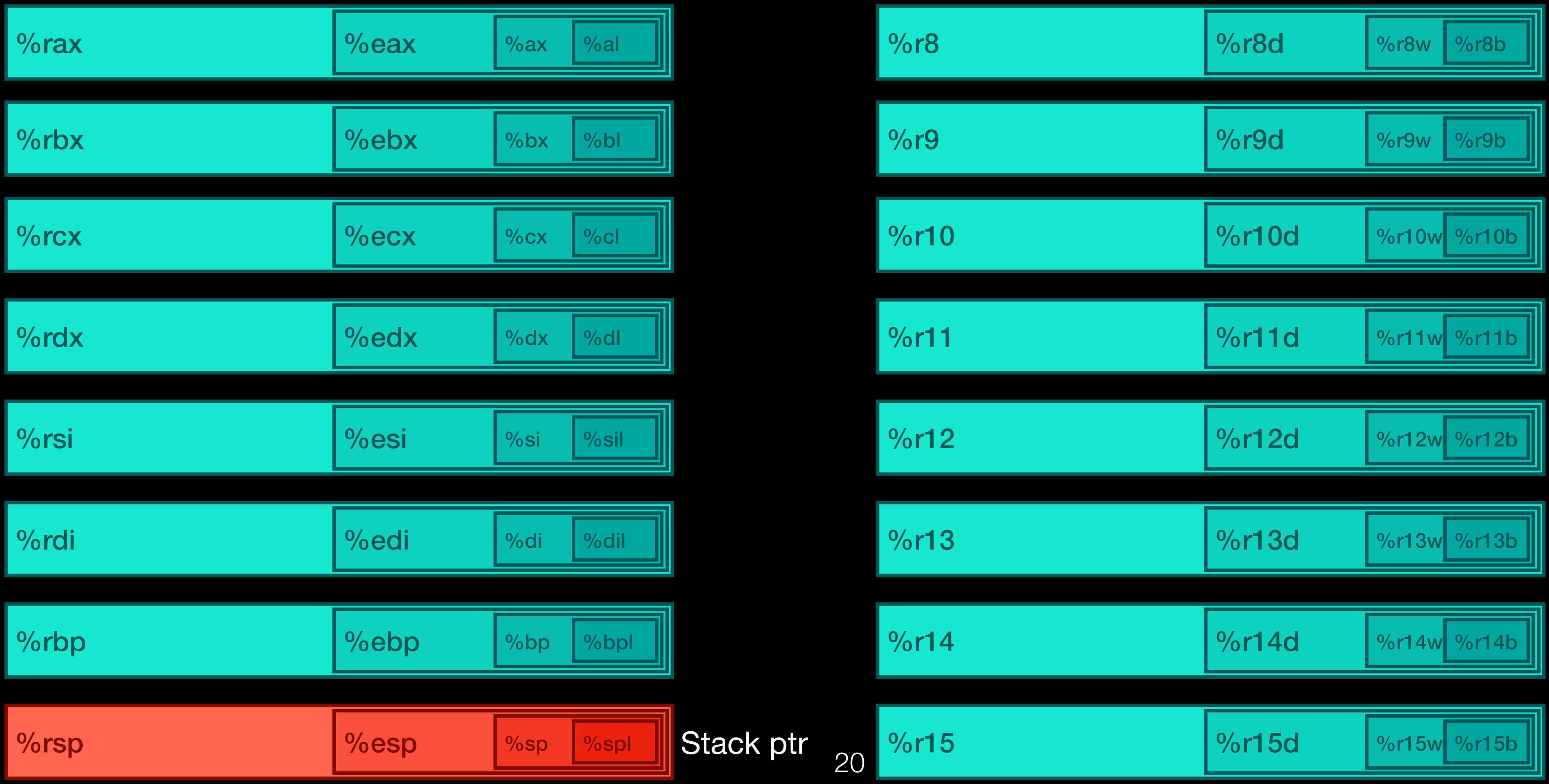
Calling functions

Calling functions

- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?

Calling functions

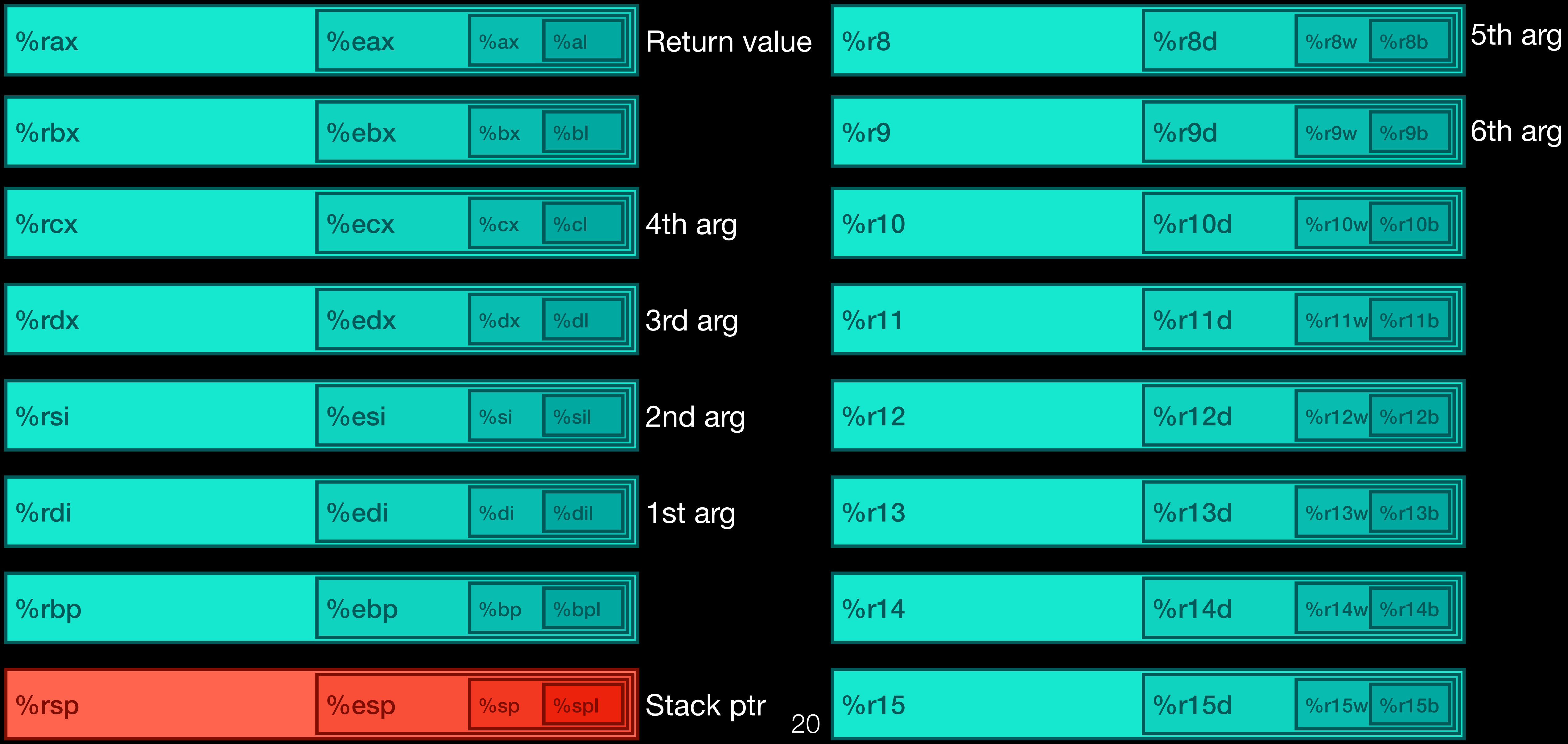
- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?



%rax	%eax	%ax	%al		%r8	%r8d	%r8w	%r8b
%rbx	%ebx	%bx	%bl		%r9	%r9d	%r9w	%r9b
%rcx	%ecx	%cx	%cl		%r10	%r10d	%r10w	%r10b
%rdx	%edx	%dx	%dl		%r11	%r11d	%r11w	%r11b
%rsi	%esi	%si	%sil		%r12	%r12d	%r12w	%r12b
%rdi	%edi	%di	%dil		%r13	%r13d	%r13w	%r13b
%rbp	%ebp	%bp	%bpl		%r14	%r14d	%r14w	%r14b
%rsp	%esp	%sp	%spl	Stack ptr	%r15	%r15d	%r15w	%r15b

Calling functions

- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?
- We use a convention (ABI) that defines how to use registers. (We present the linux x86_64 one)

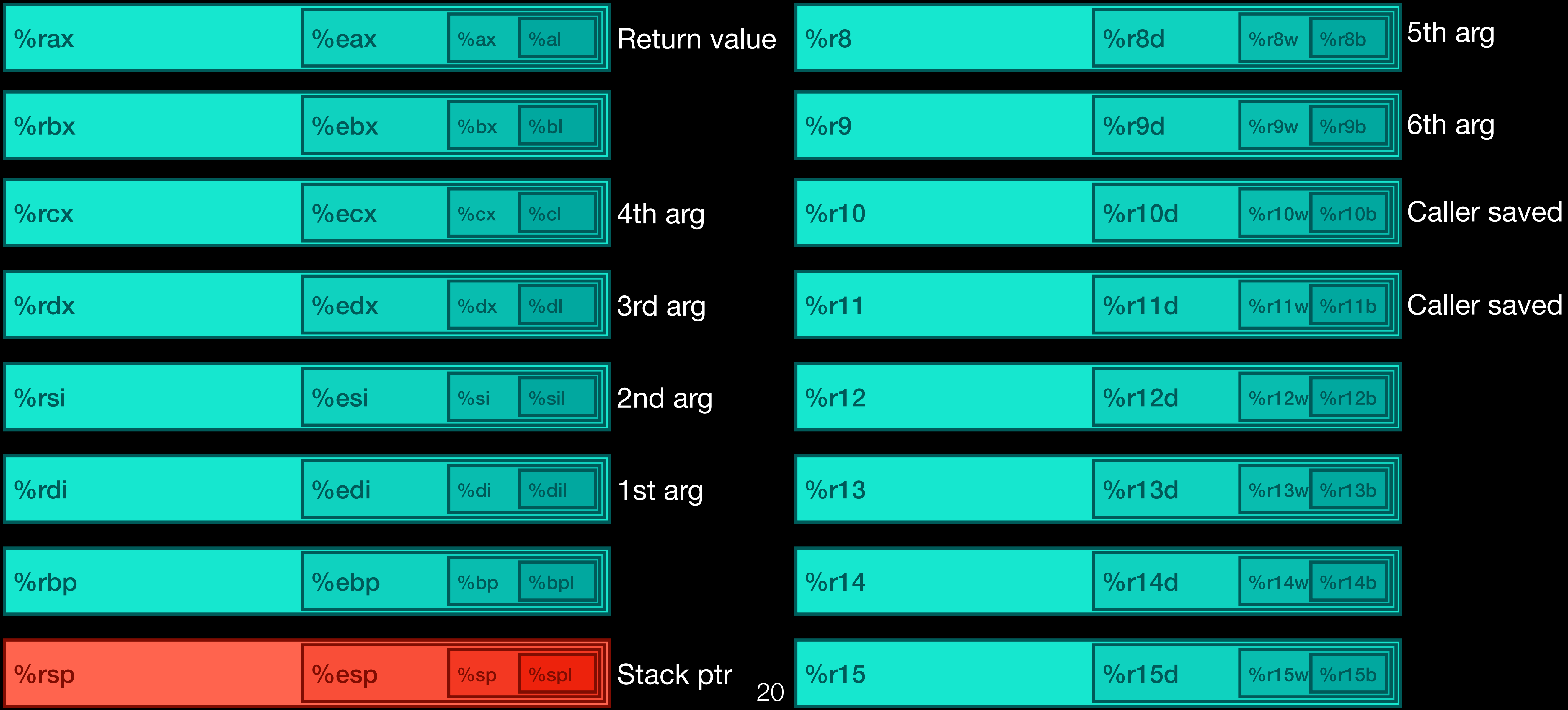


%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	4th arg
%rdx	%edx	%dx	%dl	3rd arg
%rsi	%esi	%si	%sil	2nd arg
%rdi	%edi	%di	%dil	1st arg
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	Stack ptr

%r8	%r8d	%r8w	%r8b	5th arg
%r9	%r9d	%r9w	%r9b	6th arg
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Calling functions

- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?
- We use a convention (ABI) that defines how to use registers. (We present the linux x86_64 one)
- *Caller* saved register must be saved on the stack before calling functions if needed.

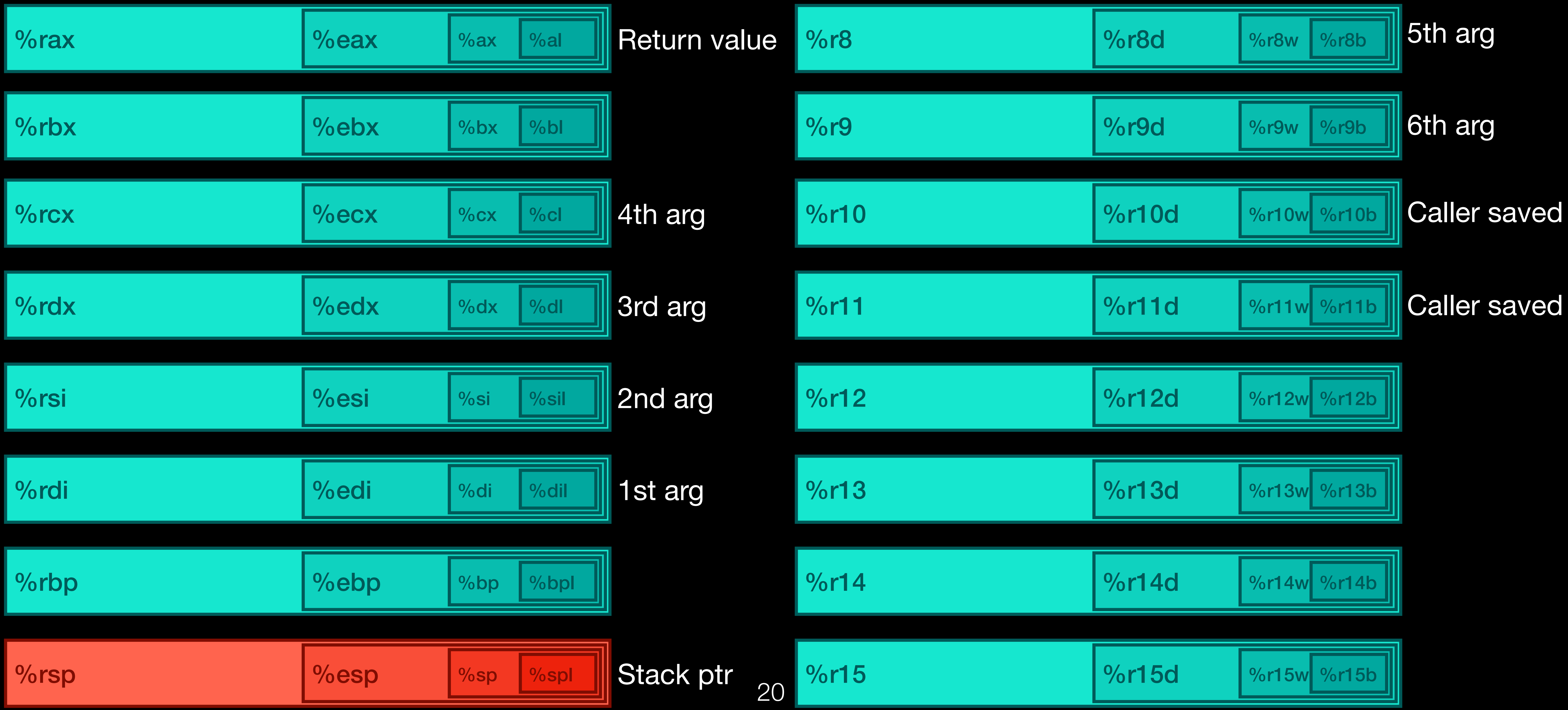


%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	4th arg
%rdx	%edx	%dx	%dl	3rd arg
%rsi	%esi	%si	%sil	2nd arg
%rdi	%edi	%di	%dil	1st arg
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	Stack ptr

%r8	%r8d	%r8w	%r8b	5th arg
%r9	%r9d	%r9w	%r9b	6th arg
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Calling functions

- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?
- We use a convention (ABI) that defines how to use registers. (We present the linux x86_64 one)
- *Caller* saved register must be saved on the stack before calling functions if needed.
- *Callee* saved must be saved by the function that wants to use it.



%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	4th arg
%rdx	%edx	%dx	%dl	3rd arg
%rsi	%esi	%si	%sil	2nd arg
%rdi	%edi	%di	%dil	1st arg
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	Stack ptr
%r8	%r8d	%r8w	%r8b	5th arg
%r9	%r9d	%r9w	%r9b	6th arg
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Calling functions

- How do we make sure we can keep our register values when we call functions ?
  How can we pass a function parameters and get a return value ?
- We use a convention (ABI) that defines how to use registers. (We present the linux x86_64 one)
- *Caller* saved register must be saved on the stack before calling functions if needed.
- *Callee* saved must be saved by the function that wants to use it.
- Too many arguments ? Use the stack !

%rax	%eax	%ax	%al	Return value	%r8	%r8d	%r8w	%r8b	5th arg
%rbx	%ebx	%bx	%bl	Callee saved	%r9	%r9d	%r9w	%r9b	6th arg
%rcx	%ecx	%cx	%cl	4th arg	%r10	%r10d	%r10w	%r10b	Caller saved
%rdx	%edx	%dx	%dl	3rd arg	%r11	%r11d	%r11w	%r11b	Caller saved
%rsi	%esi	%si	%sil	2nd arg	%r12	%r12d	%r12w	%r12b	Callee saved
%rdi	%edi	%di	%dil	1st arg	%r13	%r13d	%r13w	%r13b	Callee saved
%rbp	%ebp	%bp	%bpl	Callee saved	%r14	%r14d	%r14w	%r14b	Callee saved
%rsp	%esp	%sp	%spl	Stack ptr	%r15	%r15d	%r15w	%r15b	Callee saved

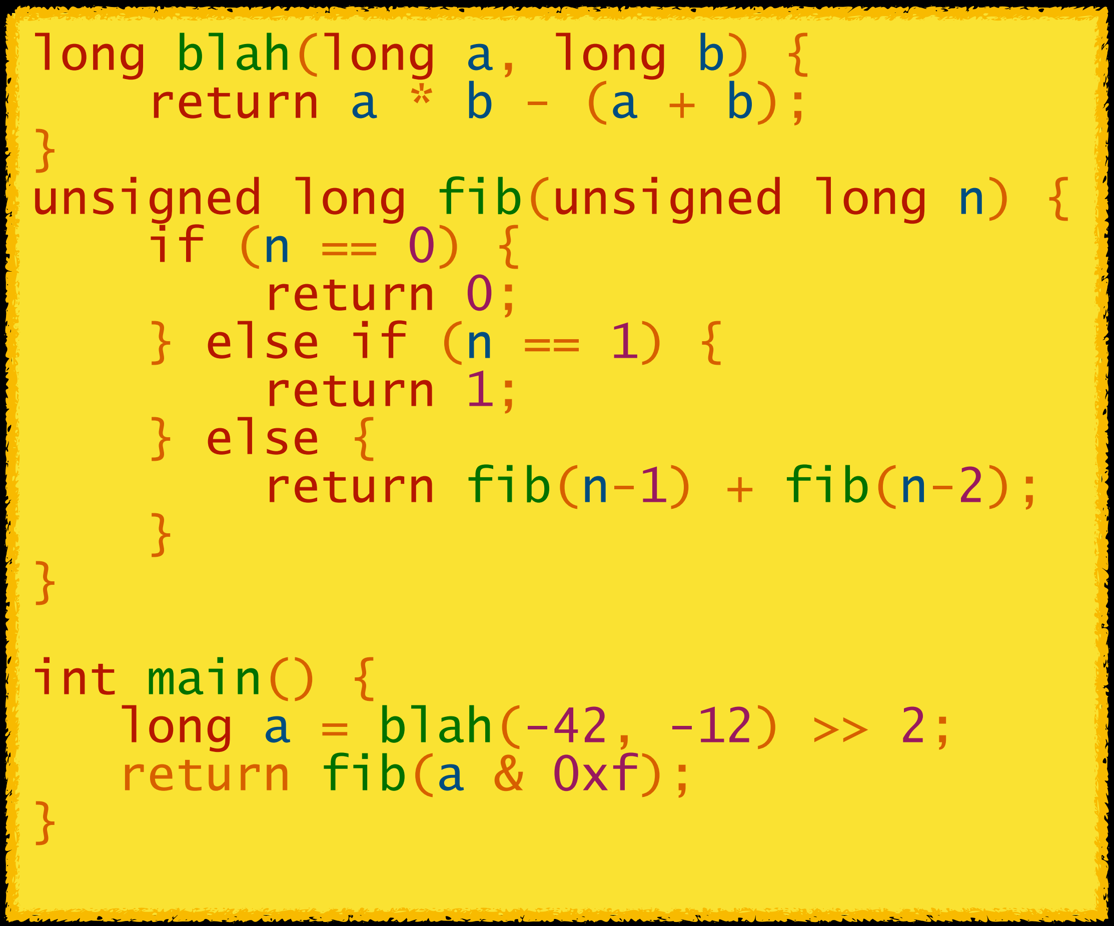
An example

An example

```
long blah(long a, long b) {
    return a * b - (a + b);
}
unsigned long fib(unsigned long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

int main() {
   long a = blah(-42, -12) >> 2;
   return fib(a & 0xf);
}
```

An example



```
long blah(long a, long b) {
    return a * b - (a + b);
}
unsigned long fib(unsigned long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

int main() {
   long a = blah(-42, -12) >> 2;
   return fib(a & 0xf);
}
```

Which argument goes in which register ?

An example

```c
long blah(long a, long b) {
    return a * b - (a + b);
}
unsigned long fib(unsigned long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

int main() {
   long a = blah(-42, -12) >> 2;
   return fib(a & 0xf);
}
```

Which argument goes in which register ?

```
blah:
  movq    %rsi, %rax
  imulq   %rdi, %rax
  addq    %rdi, %rsi
  subq    %rsi, %rax
  retq
```

An example

```
long blah(long a, long b) {
    return a * b - (a + b);
}
unsigned long fib(unsigned long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

int main() {
   long a = blah(-42, -12) >> 2;
   return fib(a & 0xf);
}
```

Which argument goes in which register ?

```
blah:
  movq     %rsi, %rax
  imulq    %rdi, %rax
  addq     %rdi, %rsi
  subq     %rsi, %rax
  retq
```

```
main:
  movq     $-42, %rdi
  movq     $-12, %rsi
  callq    blah
  shrl     $2,   %eax
  andl     $15,  %eax
  movq     %rax, %rdi
  jmp      fib # TAILCALL
```

An example

```c
long blah(long a, long b) {
    return a * b - (a + b);
}
unsigned long fib(unsigned long n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

int main() {
   long a = blah(-42, -12) >> 2;
   return fib(a & 0xf);
}
```

Which argument goes in which register ?

```
blah:
  movq     %rsi, %rax
  imulq    %rdi, %rax
  addq     %rdi, %rsi
  subq     %rsi, %rax
  retq
```

```
main:
  movq     $-42, %rdi
  movq     $-12, %rsi
  callq    blah
  shrl     $2,   %eax
  andl     $15,  %eax
  movq     %rax, %rdi
  jmp      fib # TAILCALL
```

```
fib:
  pushq    %r14
  pushq    %rbx
  movq     %rdi, %rbx
  testq    %rdi, %rdi
  je       LBB1_3
  cmpq     $1,   %rbx
  jne      LBB1_5
  movl     $1,   %ebx
LBB1_3:
  movq     %rbx, %rax
  jmp      LBB1_4
LBB1_5:
  leaq     -1(%rbx), %rdi
  callq    fib
  movq     %rax, %r14
  addq     $-2,  %rbx
  movq     %rbx, %rdi
  callq    fib
  addq     %r14, %rax
LBB1_4:
  popq     %rbx
  popq     %r14
  retq
```

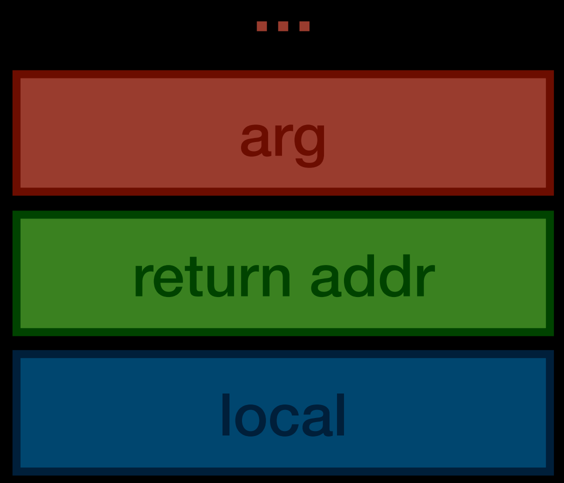
Stack frame

Stack frame

- Each function call takes up space on the stack

Stack frame

- Each function call takes up space on the stack



...

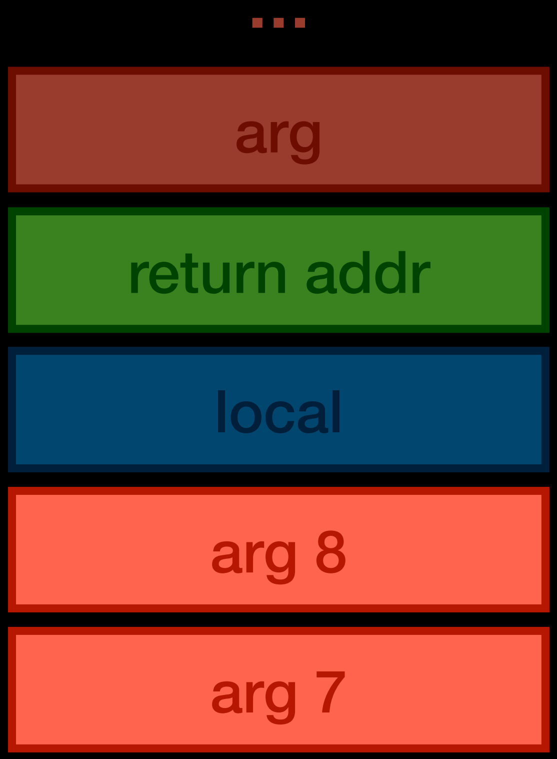
arg

return addr

local

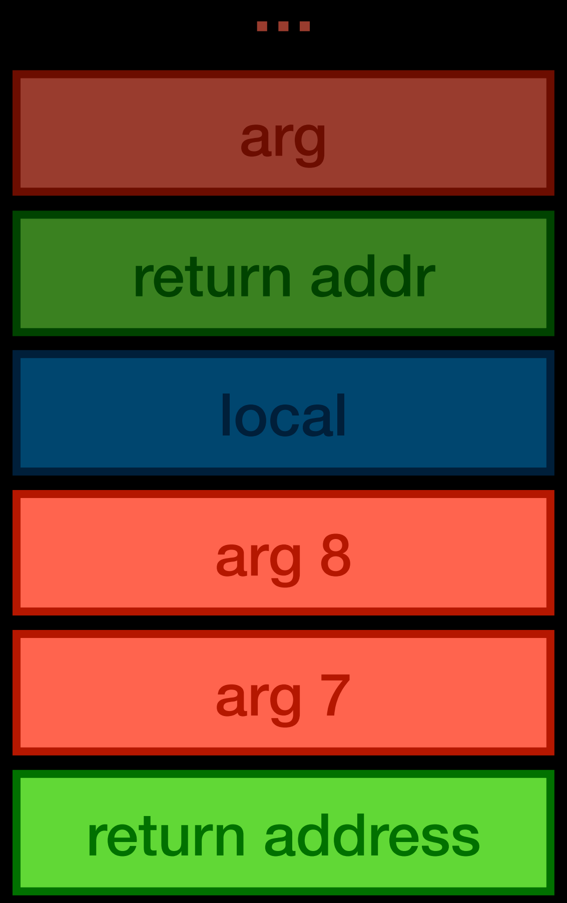
Stack frame

- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack



Stack frame

- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack
  - Then the return address



...

arg

return addr

local

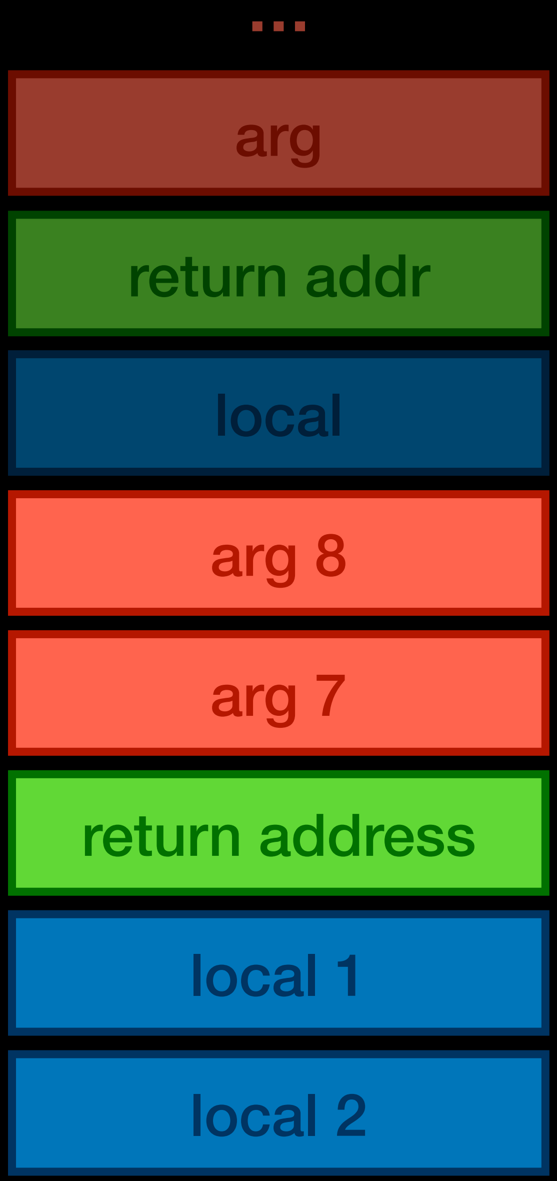
arg 8

arg 7

return address

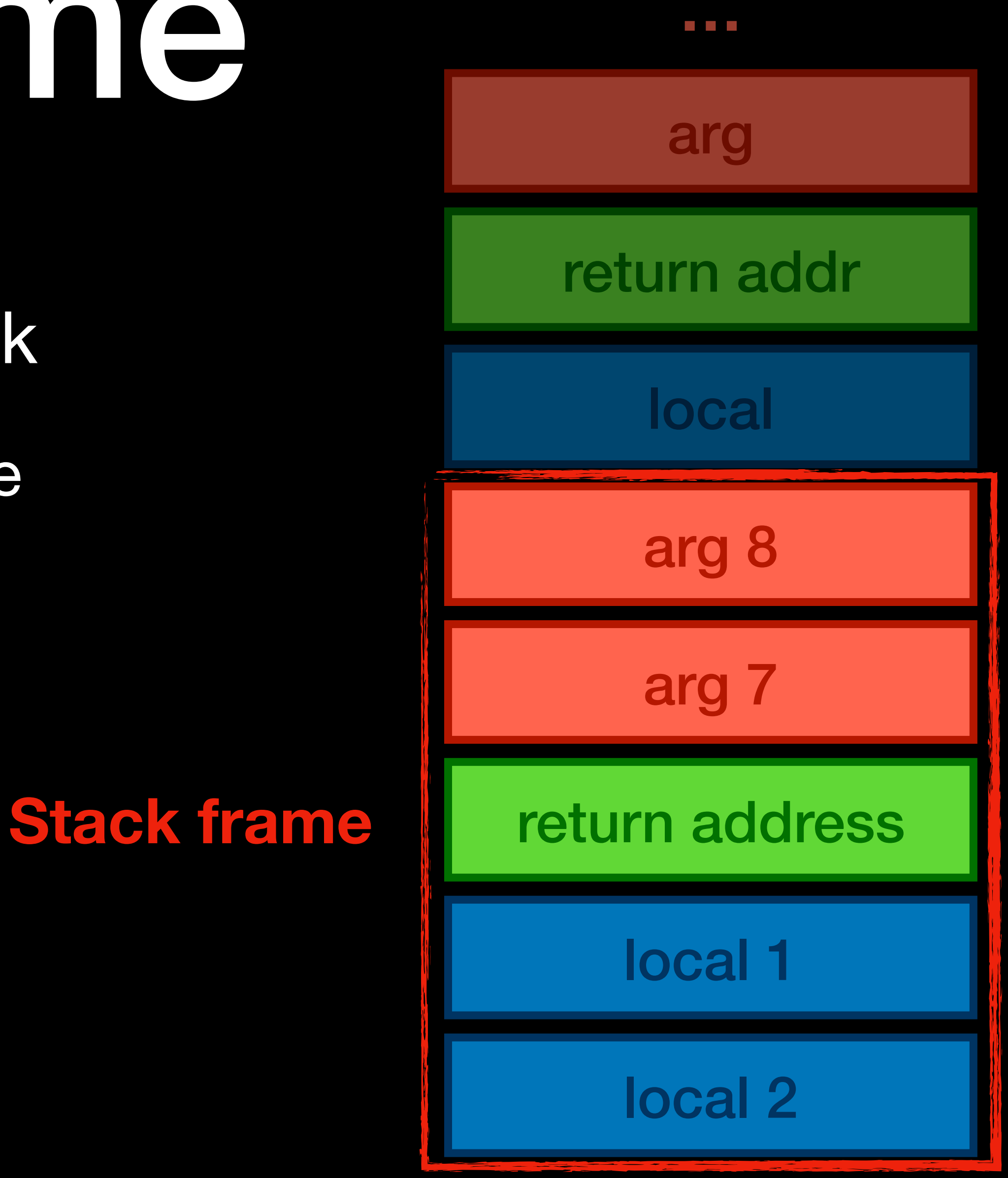
Stack frame

- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack
  - Then the return address
  - Then storage for local variable / spilling register



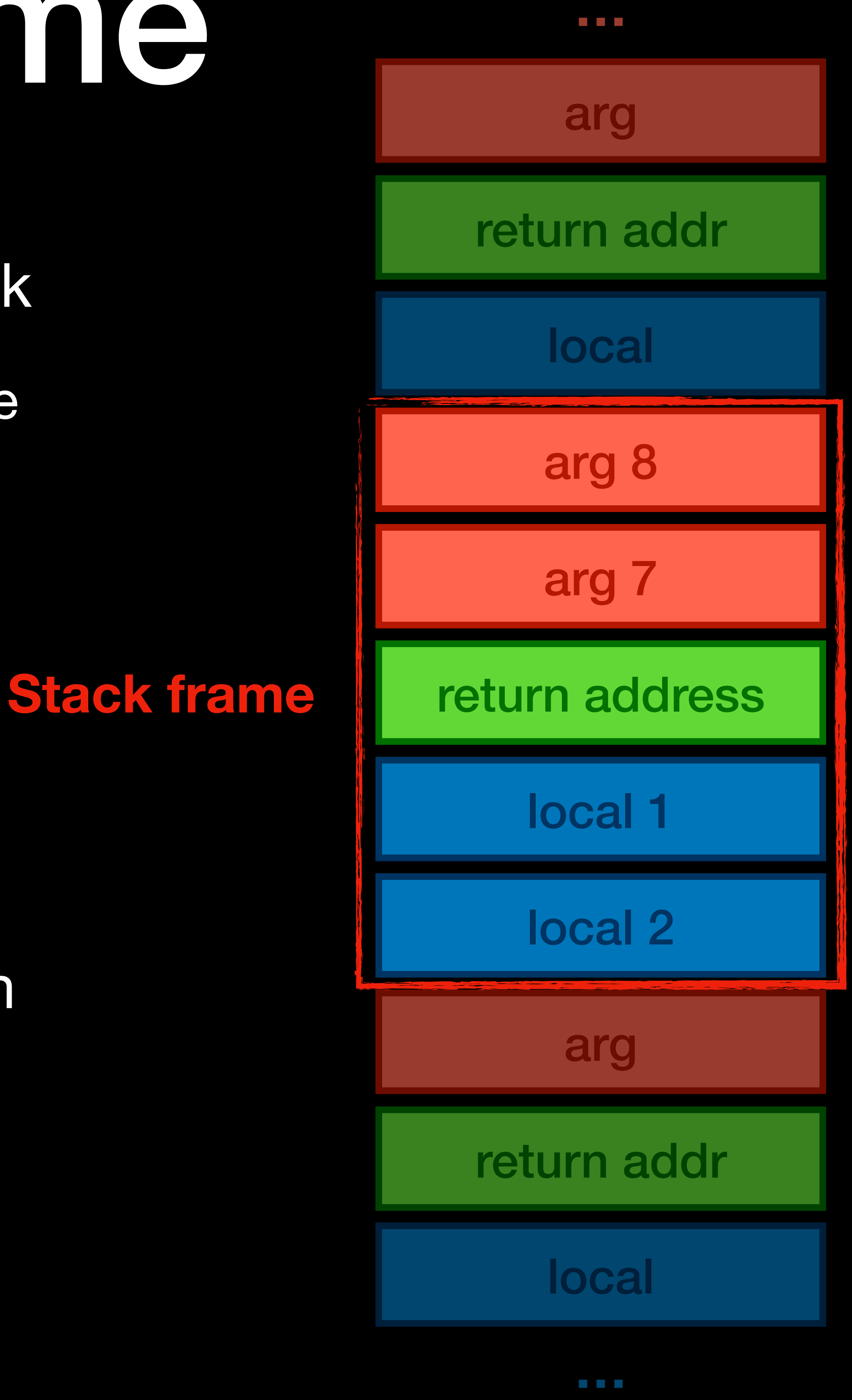
Stack frame

- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack
  - Then the return address
  - Then storage for local variable / spilling register
- We call all this the function's *stack frame*

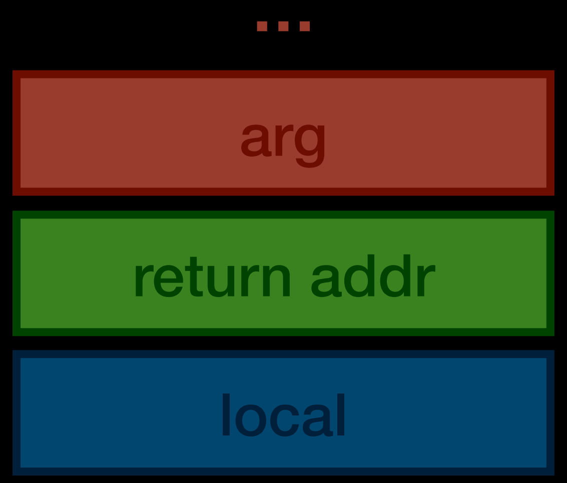


Stack frame

- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack
  - Then the return address
  - Then storage for local variable / spilling register
- We call all this the function's *stack frame*
- Each function call gets its own stack frame on the stack.



Stack frame



- Each function call takes up space on the stack
  - First any arguments that had to be pushed on the stack
  - Then the return address
  - Then storage for local variable / spilling register
- We call all this the function’s *stack frame*
- Each function call gets its own stack frame on the stack.
- Stack frame are freed up when function exit, in reverse ordre

Closing words

Closing words

This is not an exhaustive presentation, here’s a list of things we haven’t touched

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)
- Some integer instructions

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)
- Some integer instructions
- Floating point, vector instructions

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)
- Some integer instructions
- Floating point, vector instructions
- Everything about privileges / OSs / performance / debugging.

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)
- Some integer instructions
- Floating point, vector instructions
- Everything about privileges / OSs / performance / debugging.
- Microarchitecture (how CPU really work)

Closing words

This is not an exhaustive presentation, here's a list of things we haven't touched

- Mixing different register sizes (`movz`/`movs`)
- Some integer instructions
- Floating point, vector instructions
- Everything about privileges / OSs / performance / debugging.
- Microarchitecture (how CPU really work)
- See bibliography on the course website

Questions