2023

Under the Hood

Assembly, System Calls, and Hardware

David Sankel

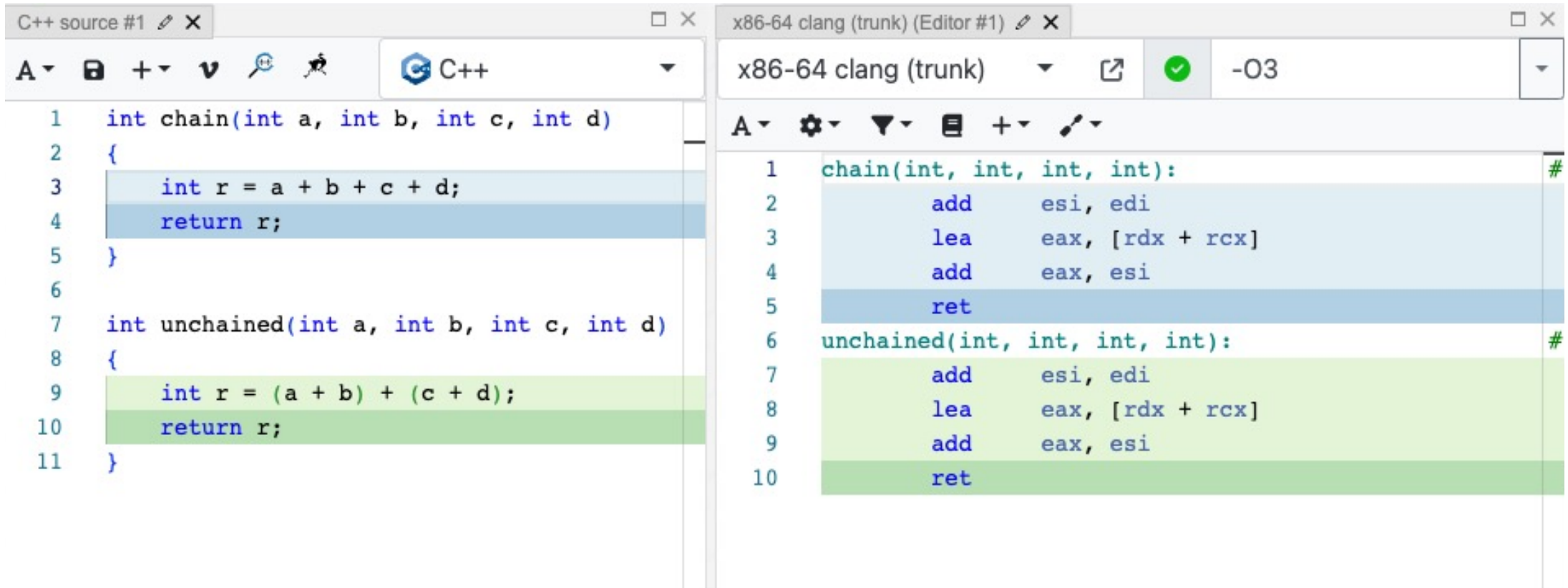
C++ now

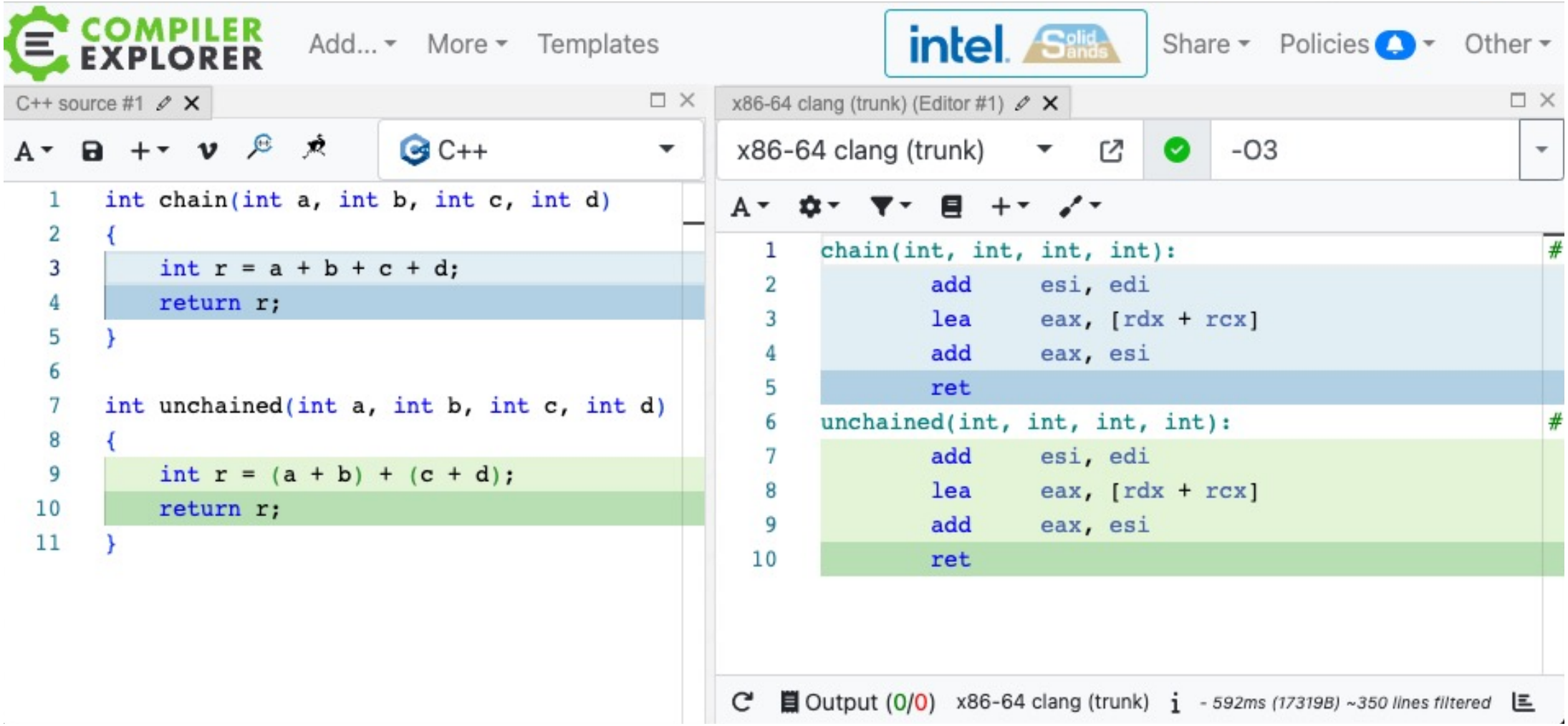
Under the Hood

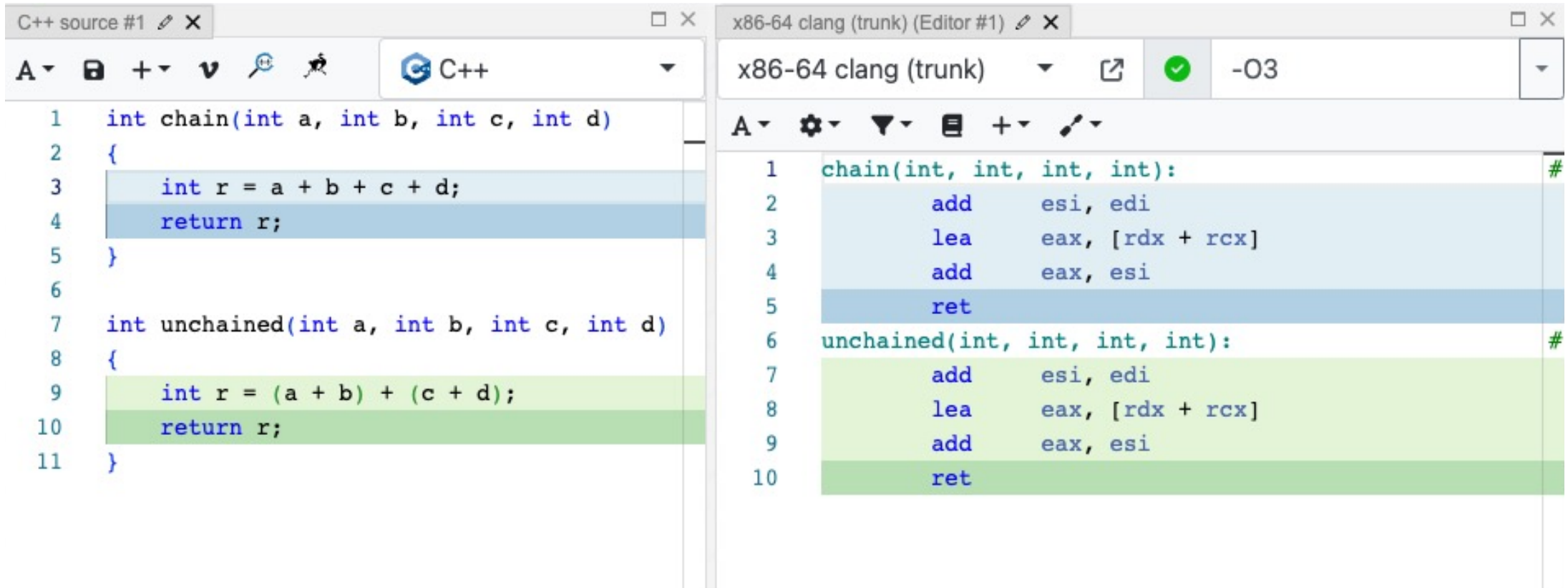
Assembly, System Calls, and Hardware

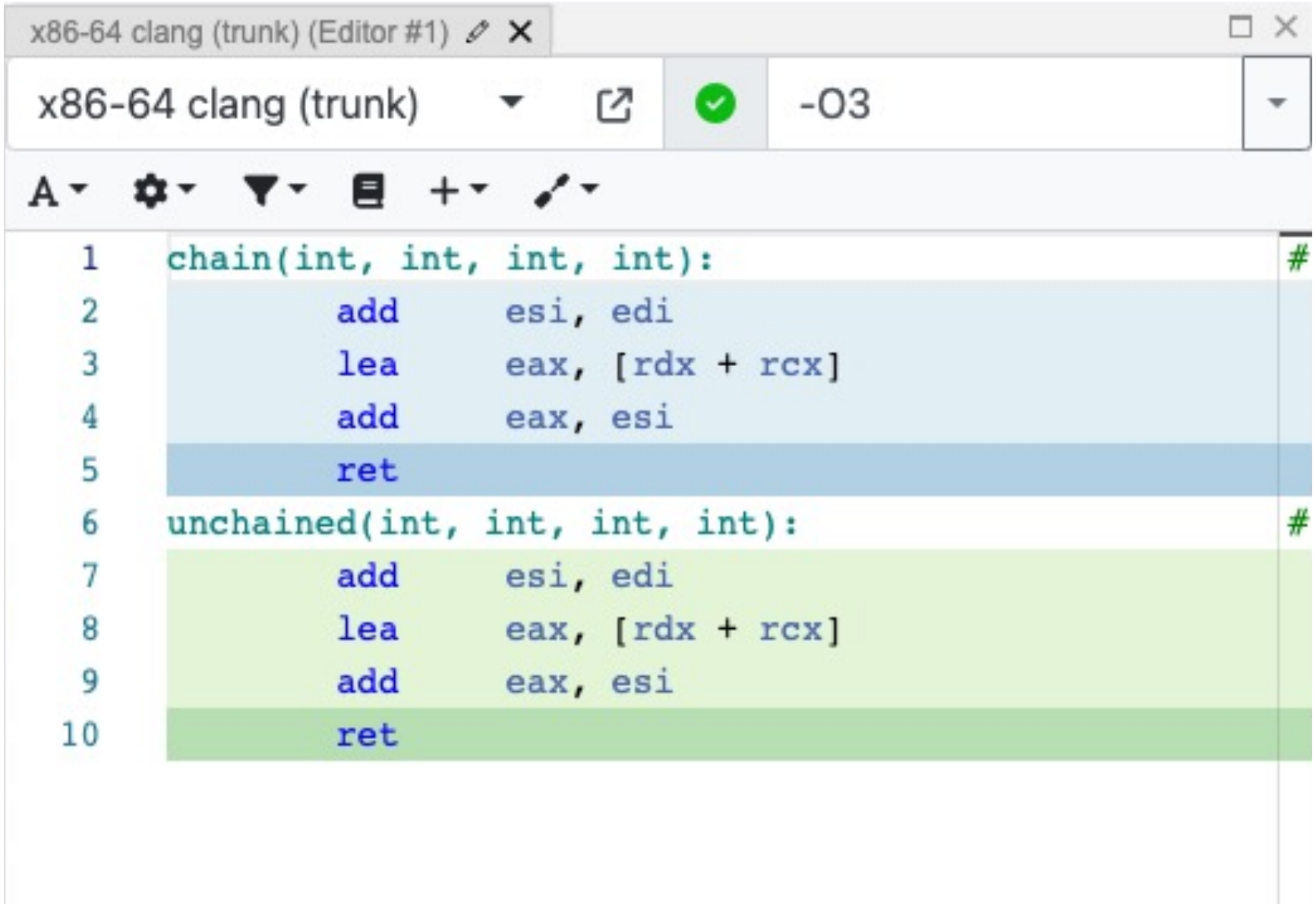
David Sankel | Principal Scientist

C++Now 2023









x86-64 clang (trunk) (Editor #1)

x86-64 clang (trunk) -O3

```
1   chain(int, int, int, int):                #
2           add     esi, edi
3           lea     eax, [rdx + rcx]
4           add     eax, esi
5           ret
6   unchained(int, int, int, int):            #
7           add     esi, edi
8           lea     eax, [rdx + rcx]
9           add     eax, esi
10          ret
```

x86-64 clang (trunk) (Editor #1)

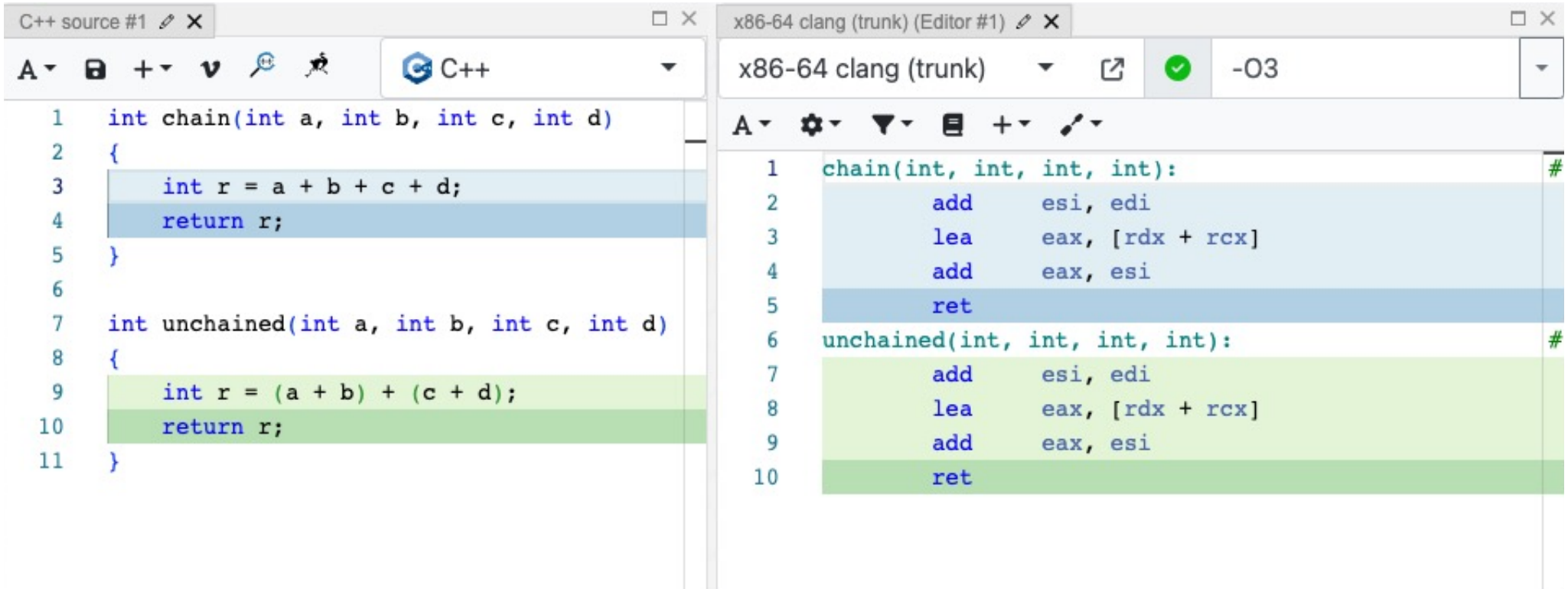
x86-64 clang (trunk) -O3

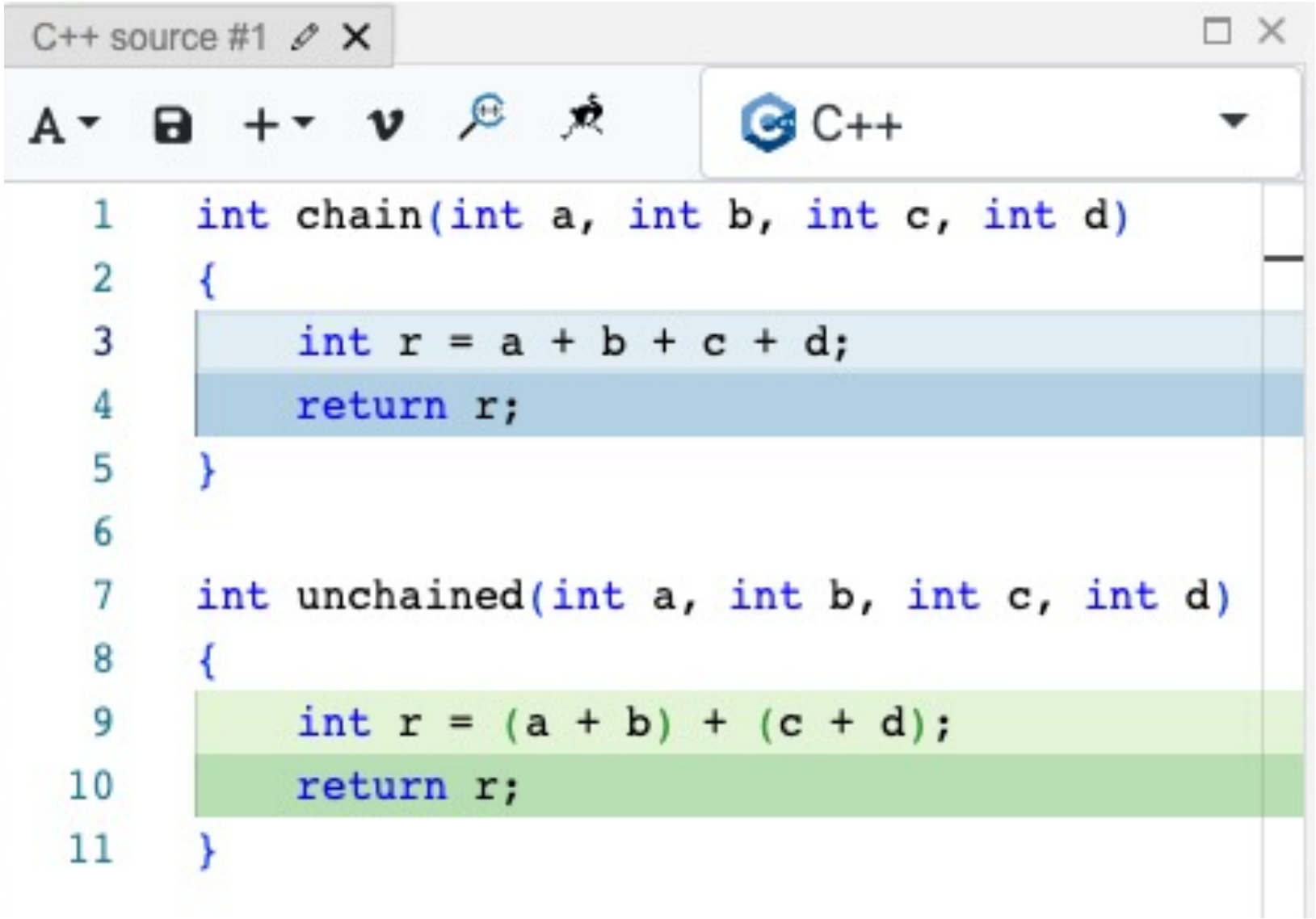
```
1   chain(int, int, int, int):                #
2           add     esi, edi
3           lea     eax, [rdx + rcx]
4           add     eax, esi
5           ret
6   unchained(int, int, int, int):            #
7           add     esi, edi
8           lea     eax, [rdx + rcx]
9           add     eax, esi
10          ret
```

x86-64 clang (trunk) (Editor #1)

x86-64 clang (trunk) -O3

```
1   chain(int, int, int, int):                #
2           add     esi, edi
3           lea     eax, [rdx + rcx]
4           add     eax, esi
5           ret
6   unchained(int, int, int, int):            #
7           add     esi, edi
8           lea     eax, [rdx + rcx]
9           add     eax, esi
10          ret
```





Don’t make performance claims without benchmarks.

(… unless you can show identical generated assembly. In that case, go right ahead)

Don’t make performance claims without benchmarks.

(… unless you can show identical generated assembly. In that case, go right ahead)

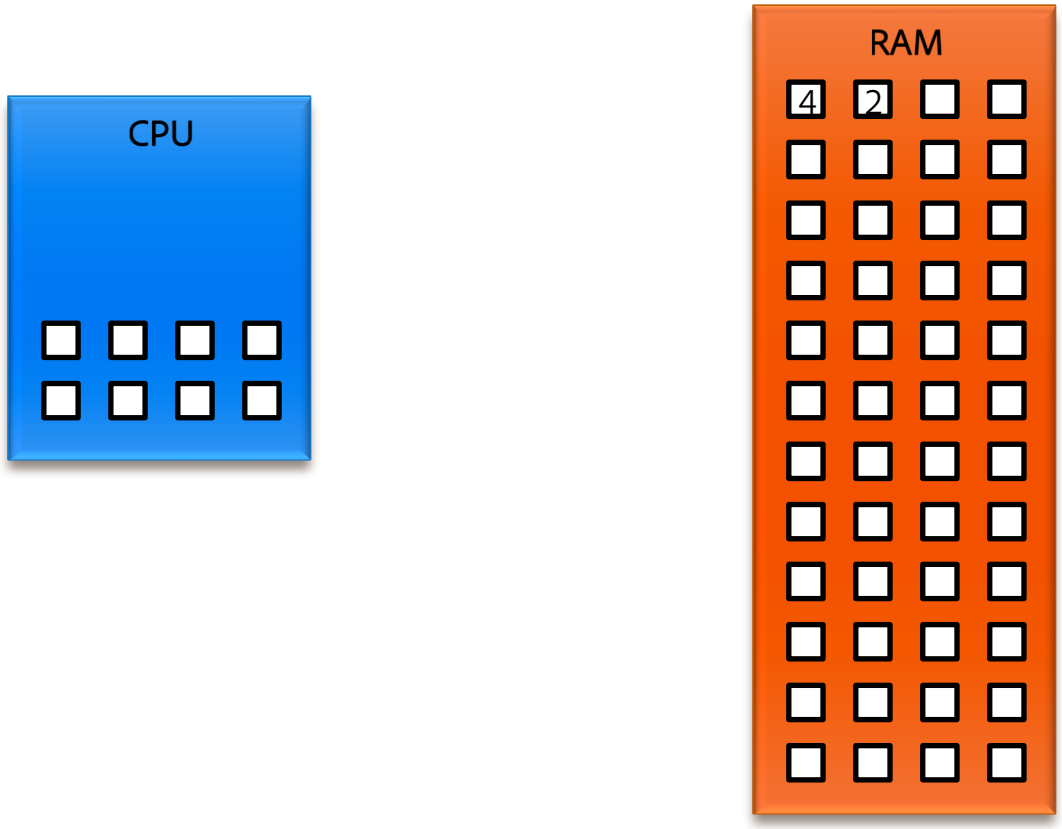
What is manual memory management?

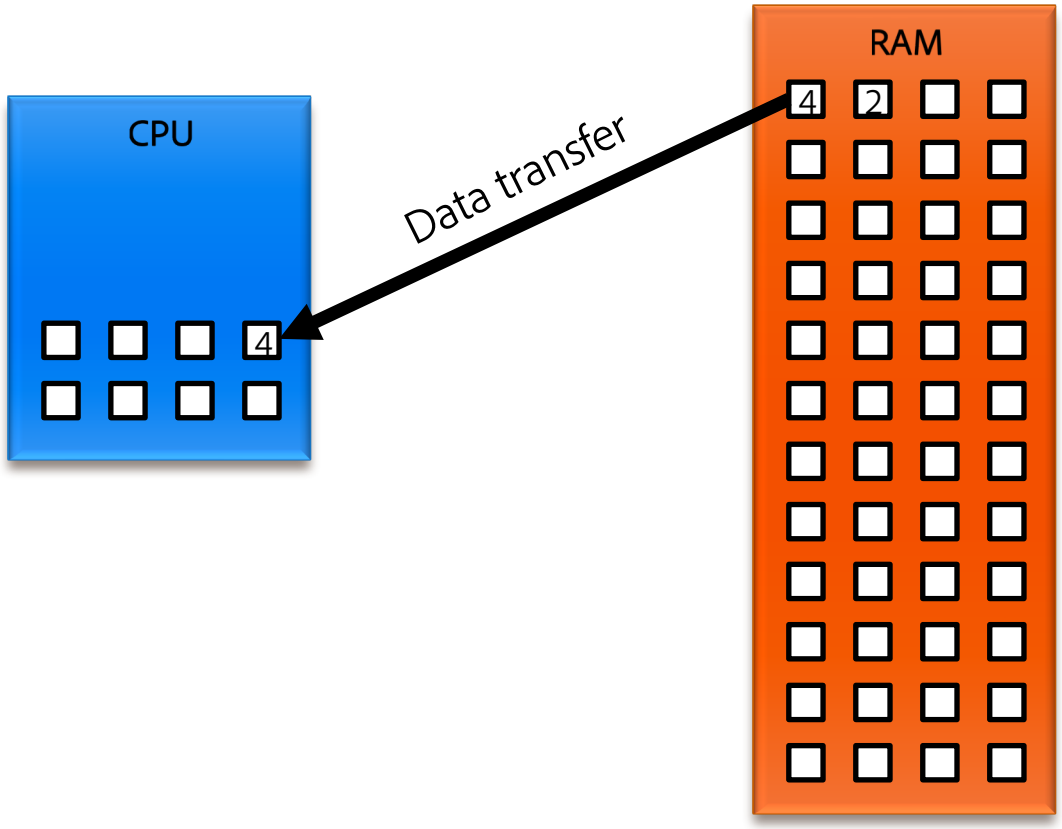
What is manual memory management?

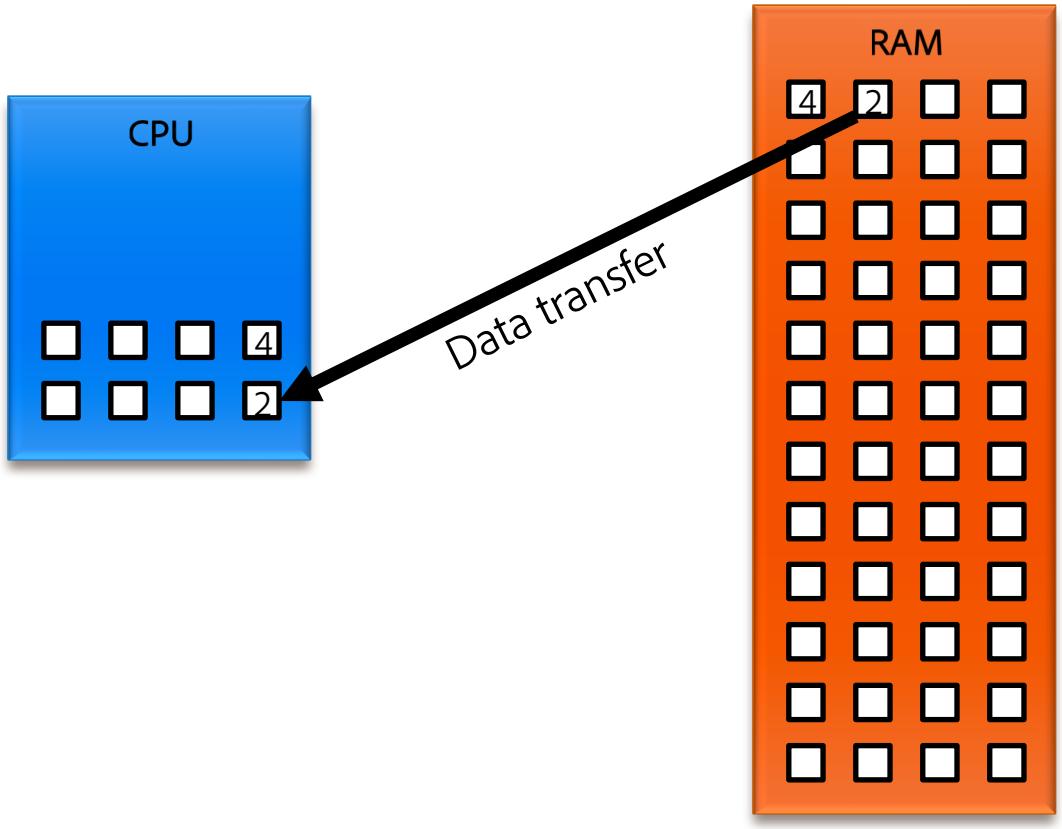
In computer science, **manual memory management** refers to the usage of manual instructions by the programmer to identify and deallocate unused objects, or garbage.

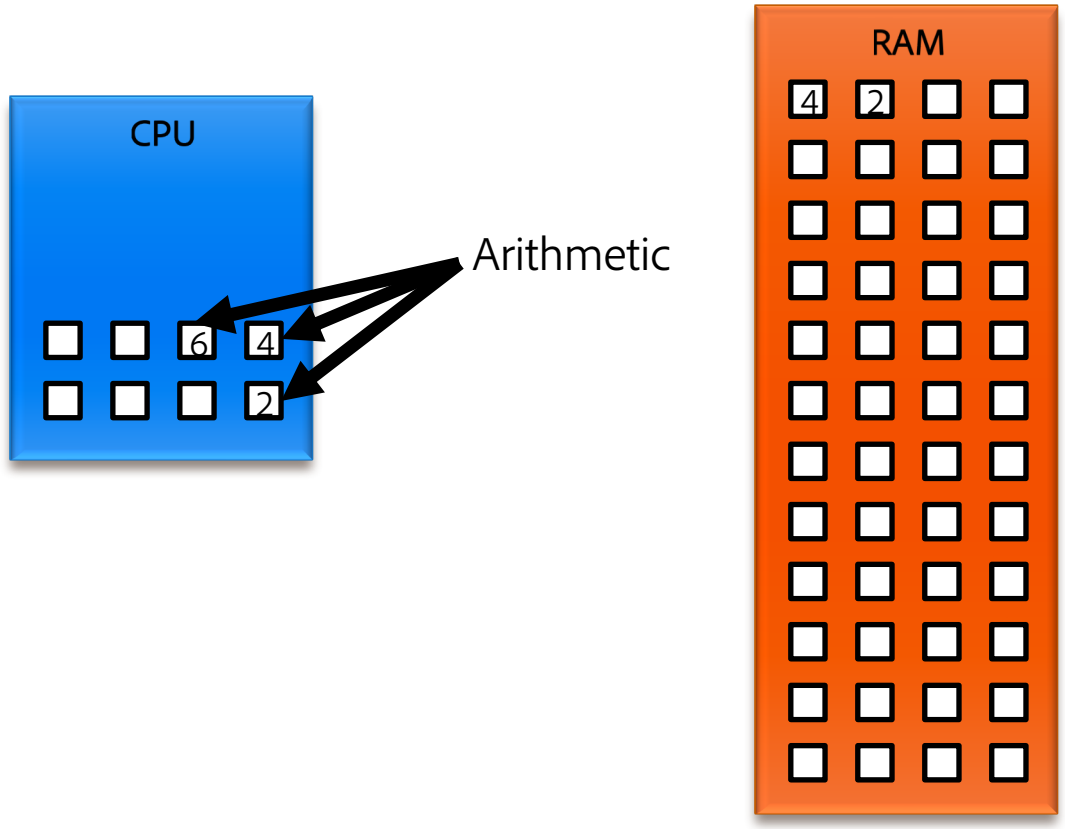
—Wikipedia

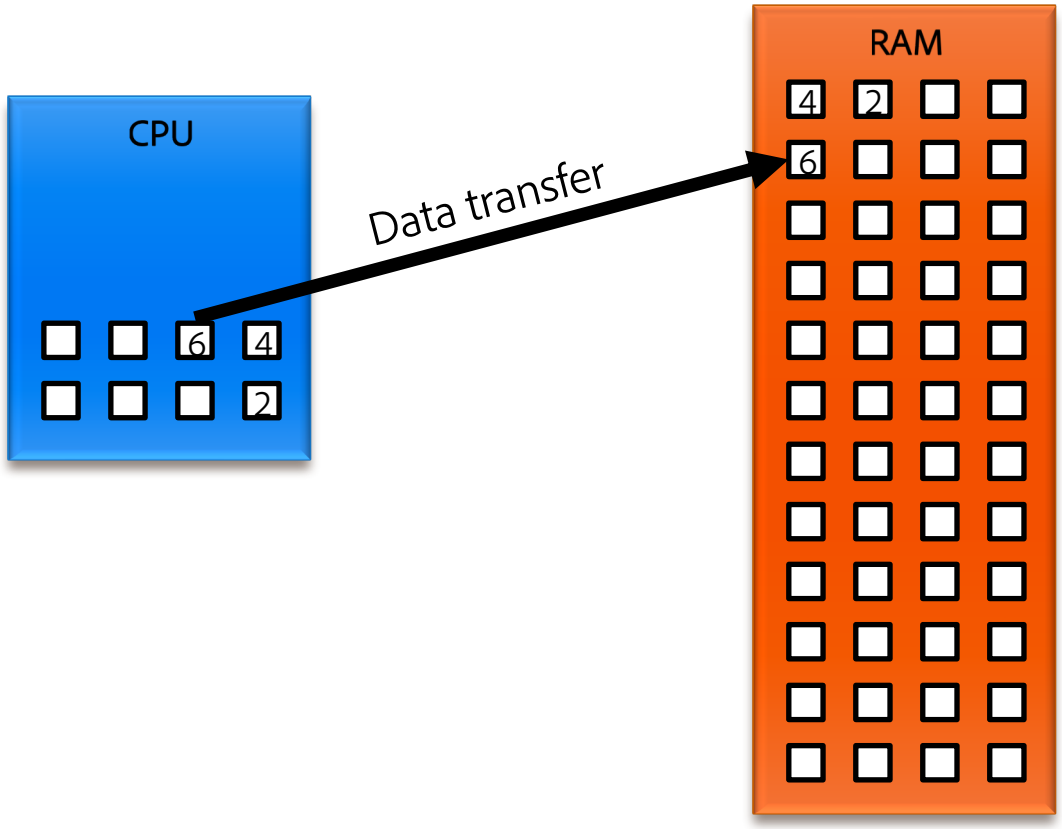
In assembly language, instructions are required to move information between the RAM and CPU



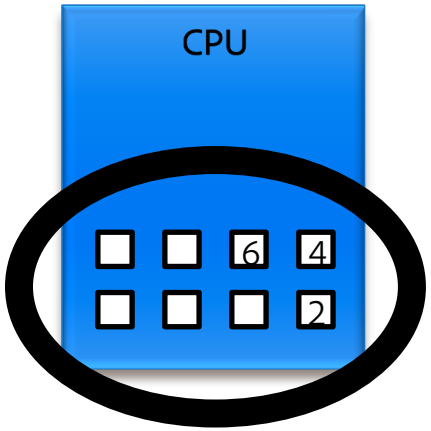


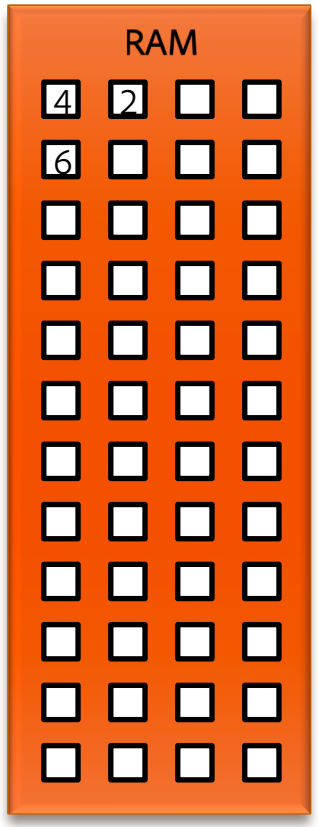




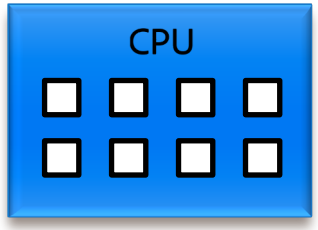


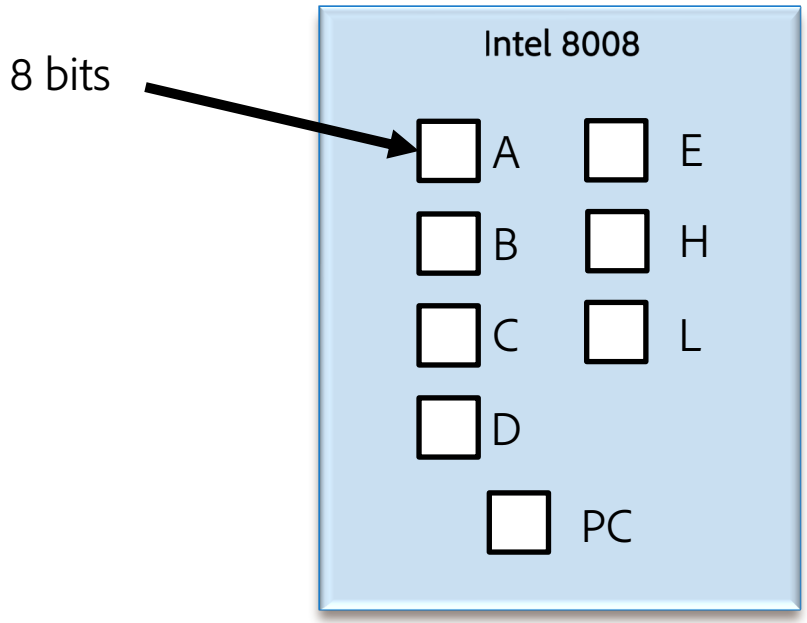
Registers



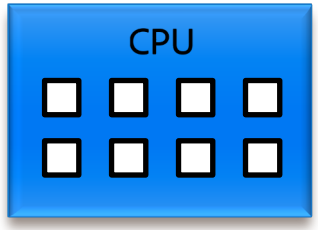


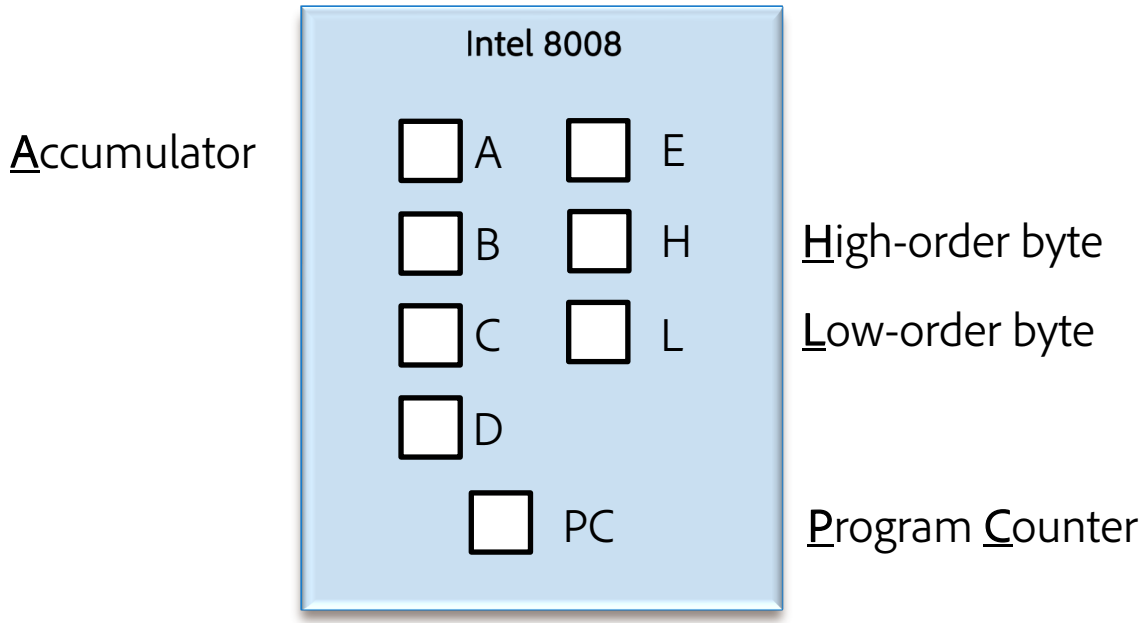
Registers



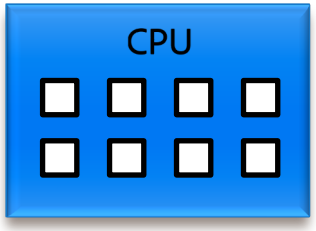


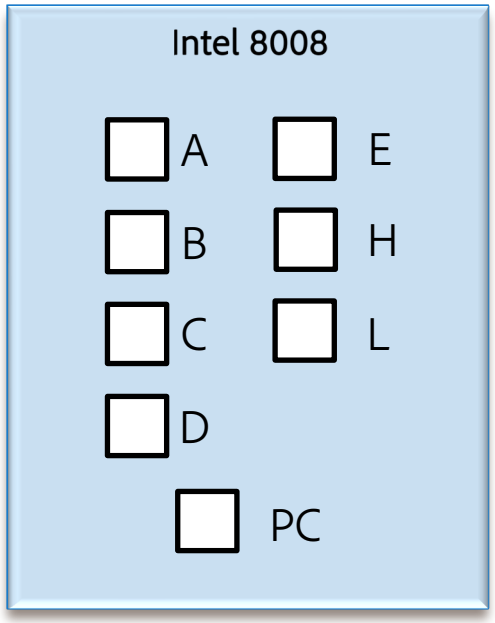
Registers



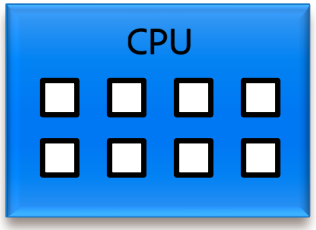


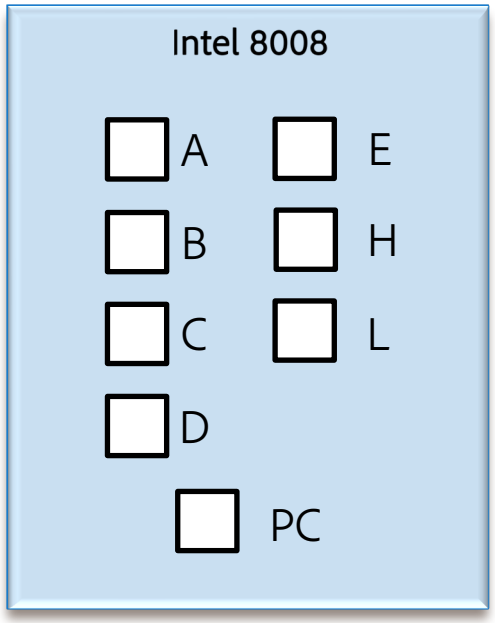
Registers

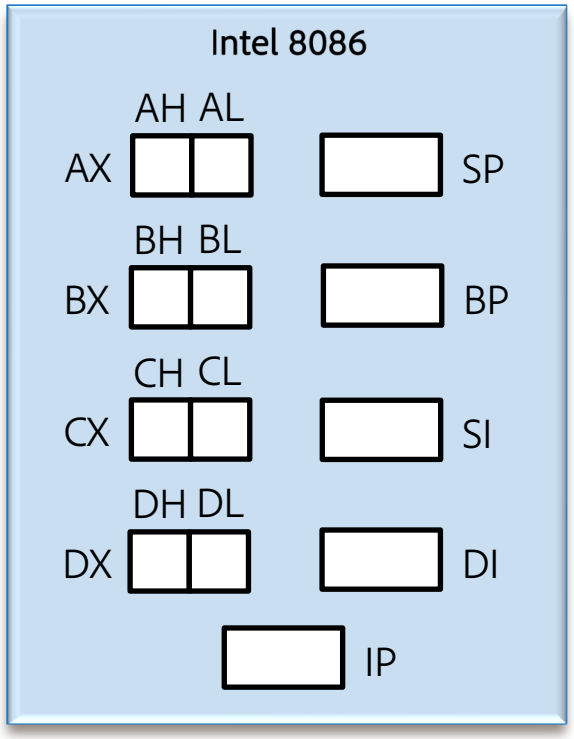




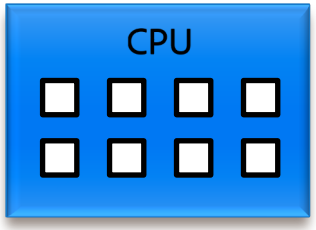
Registers

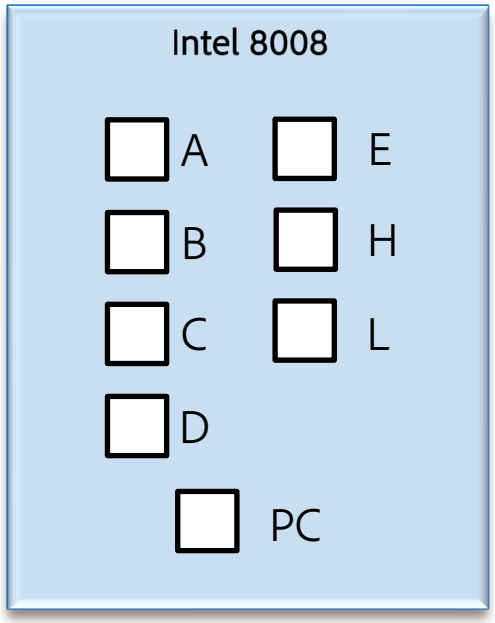


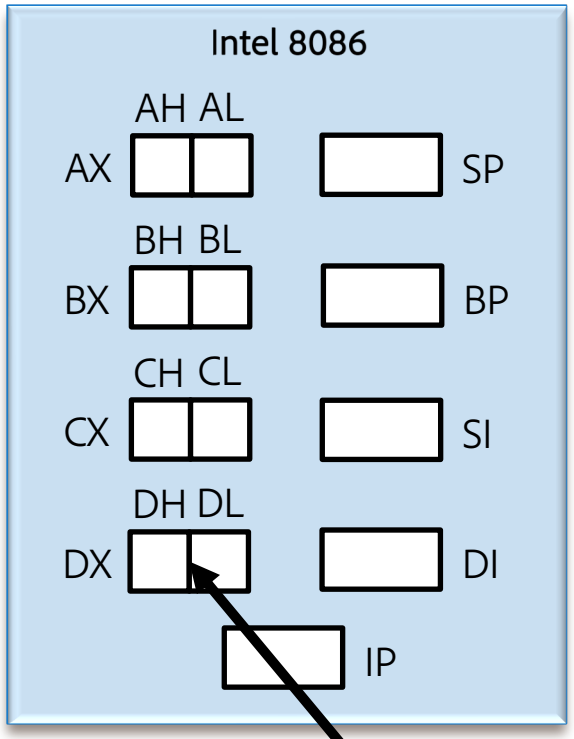




Registers

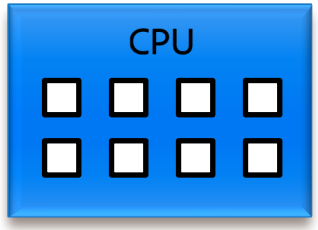


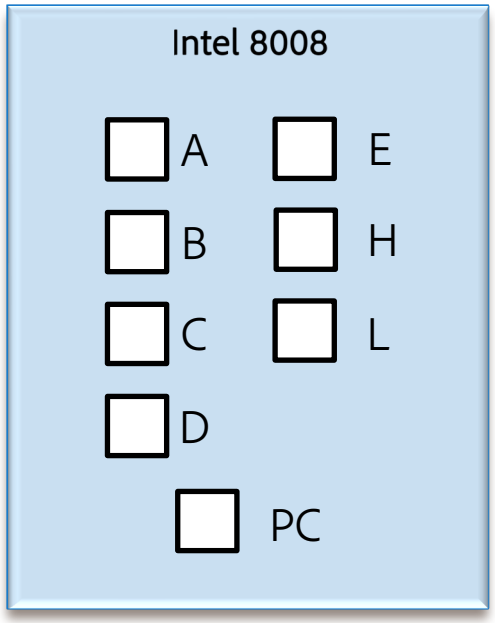


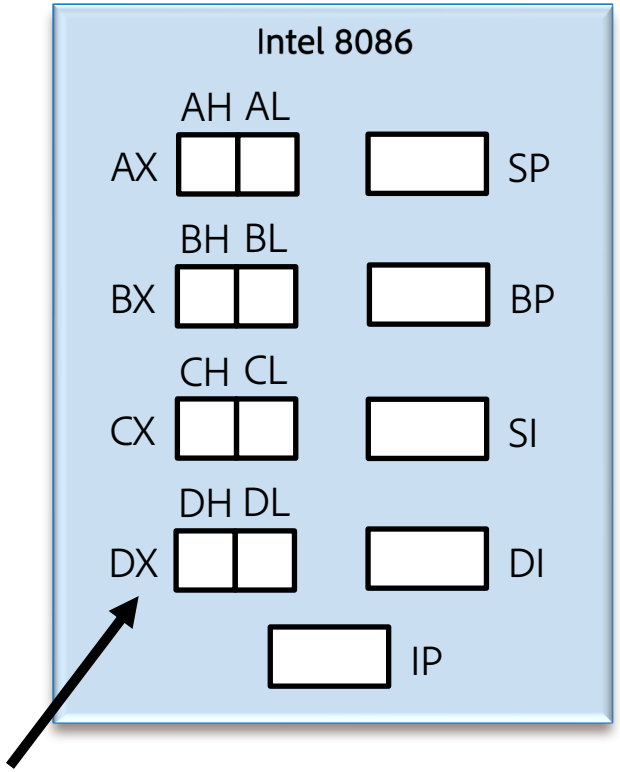


D High and D Low 8bit registers

Registers

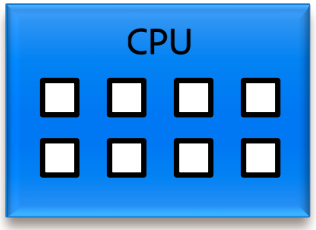


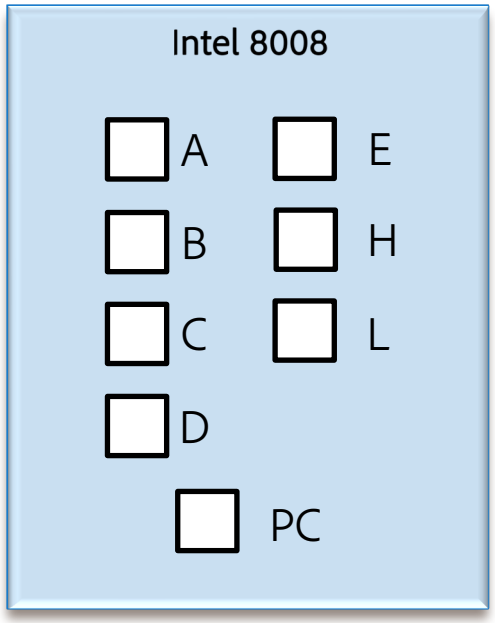


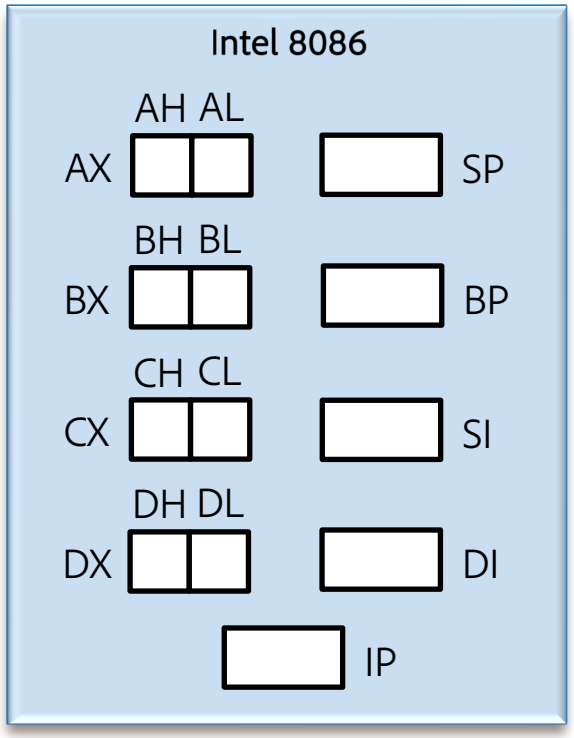


DX 16 bit register. X is a placeholder as in version 3.x or x86.

Registers







Accumulator

Base

Count

Data

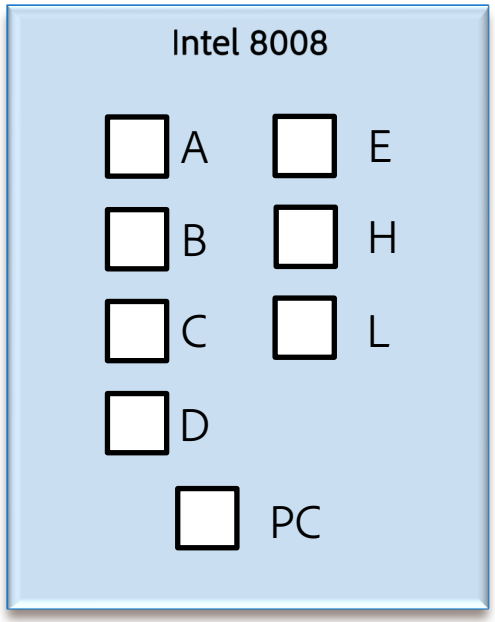
Stack Pointer

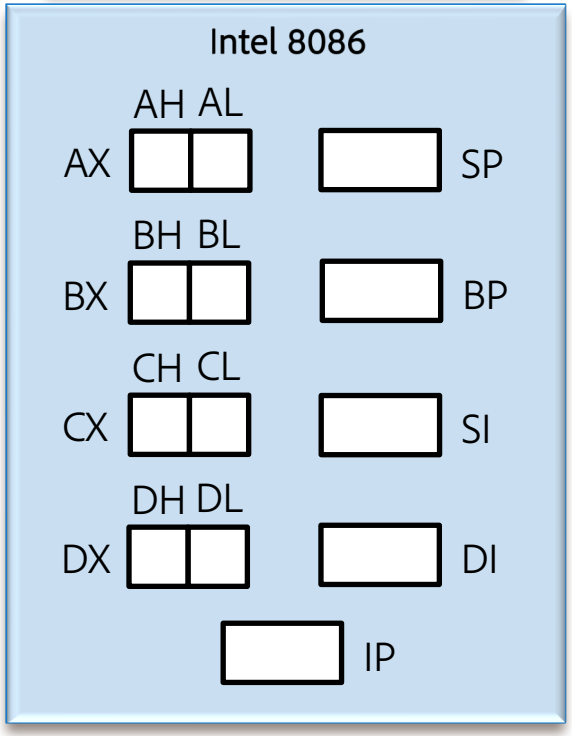
Base Pointer

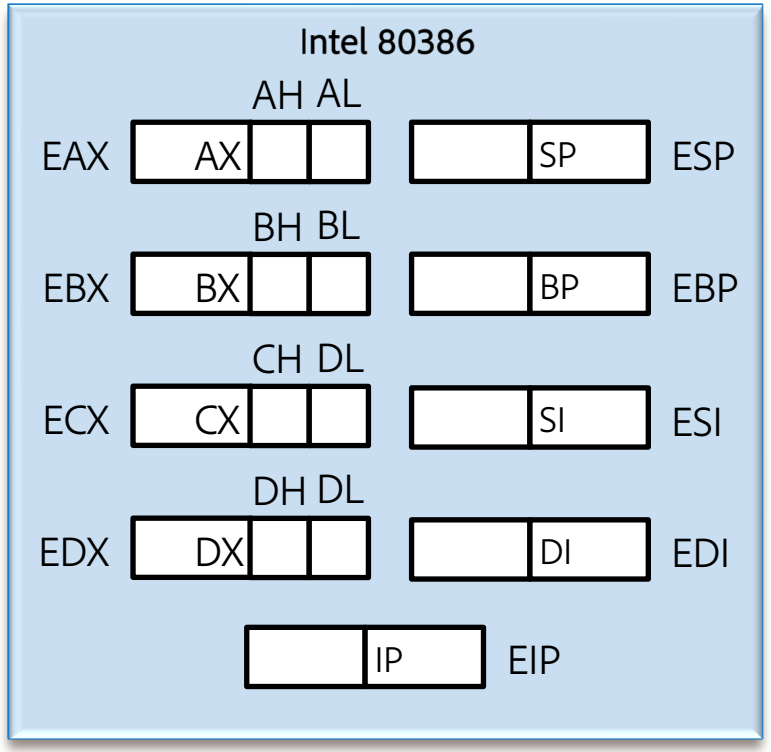
Source Index

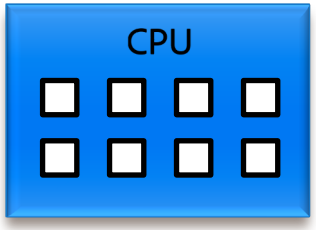
Destination Index

Instruction Pointer

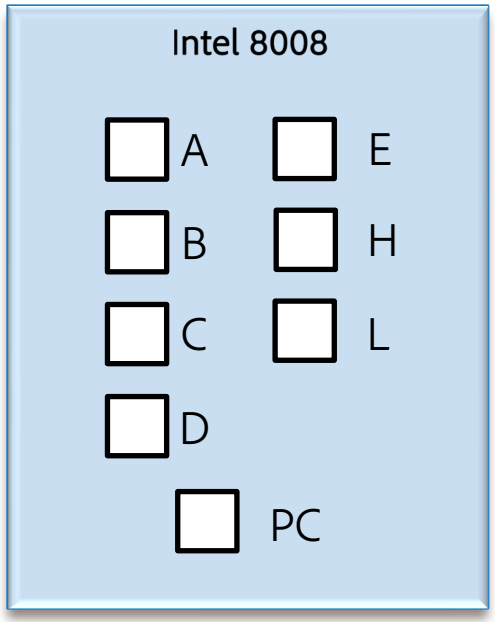


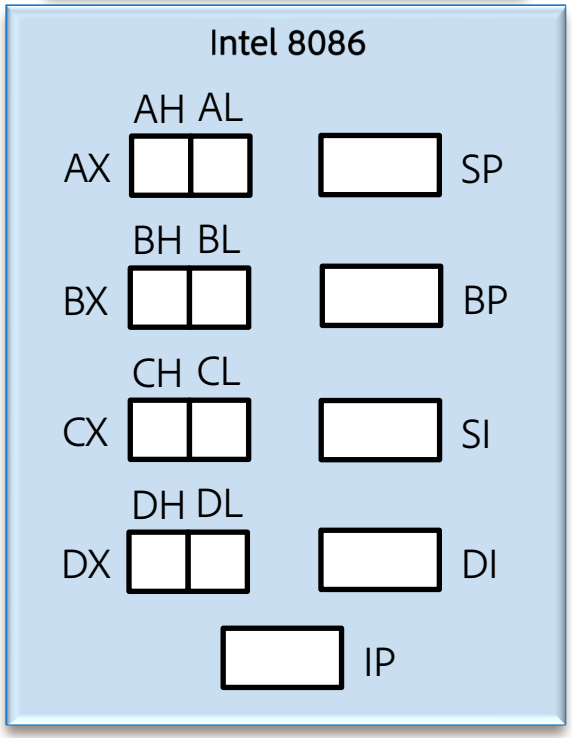




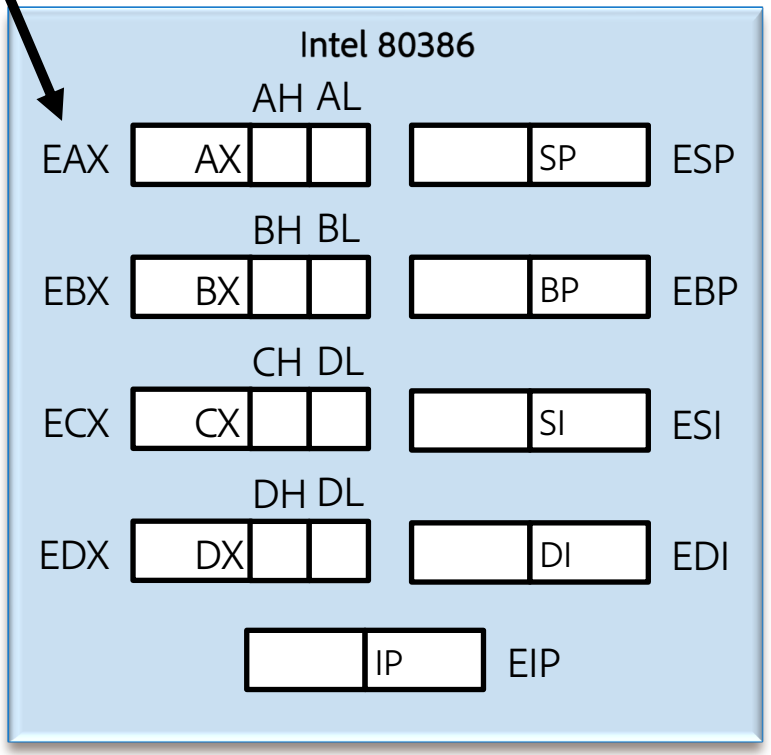


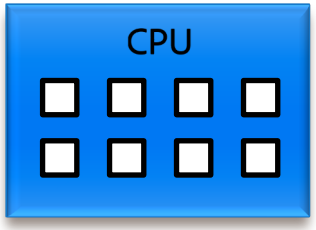
Adobe

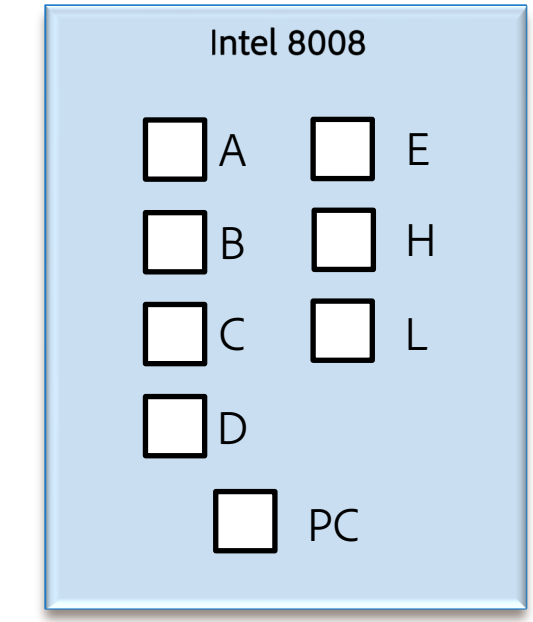


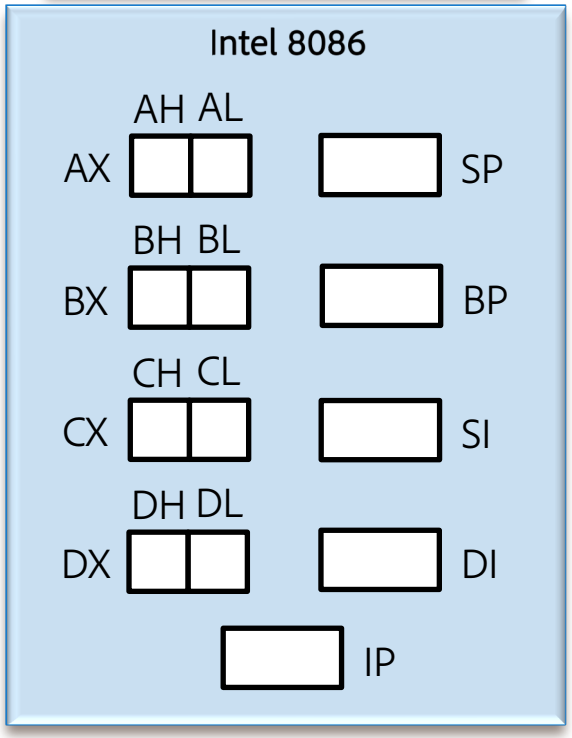


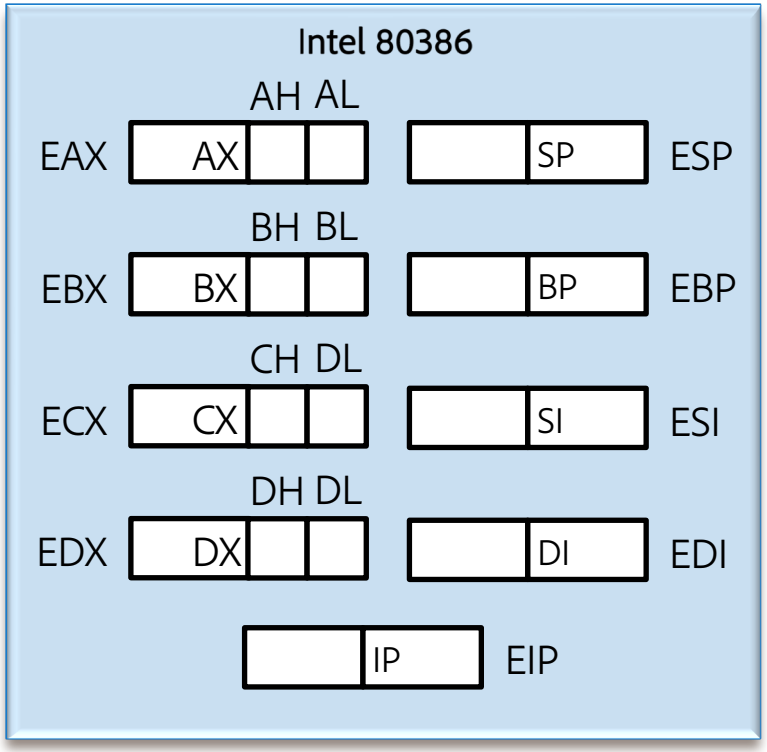
Extended Accumulator X

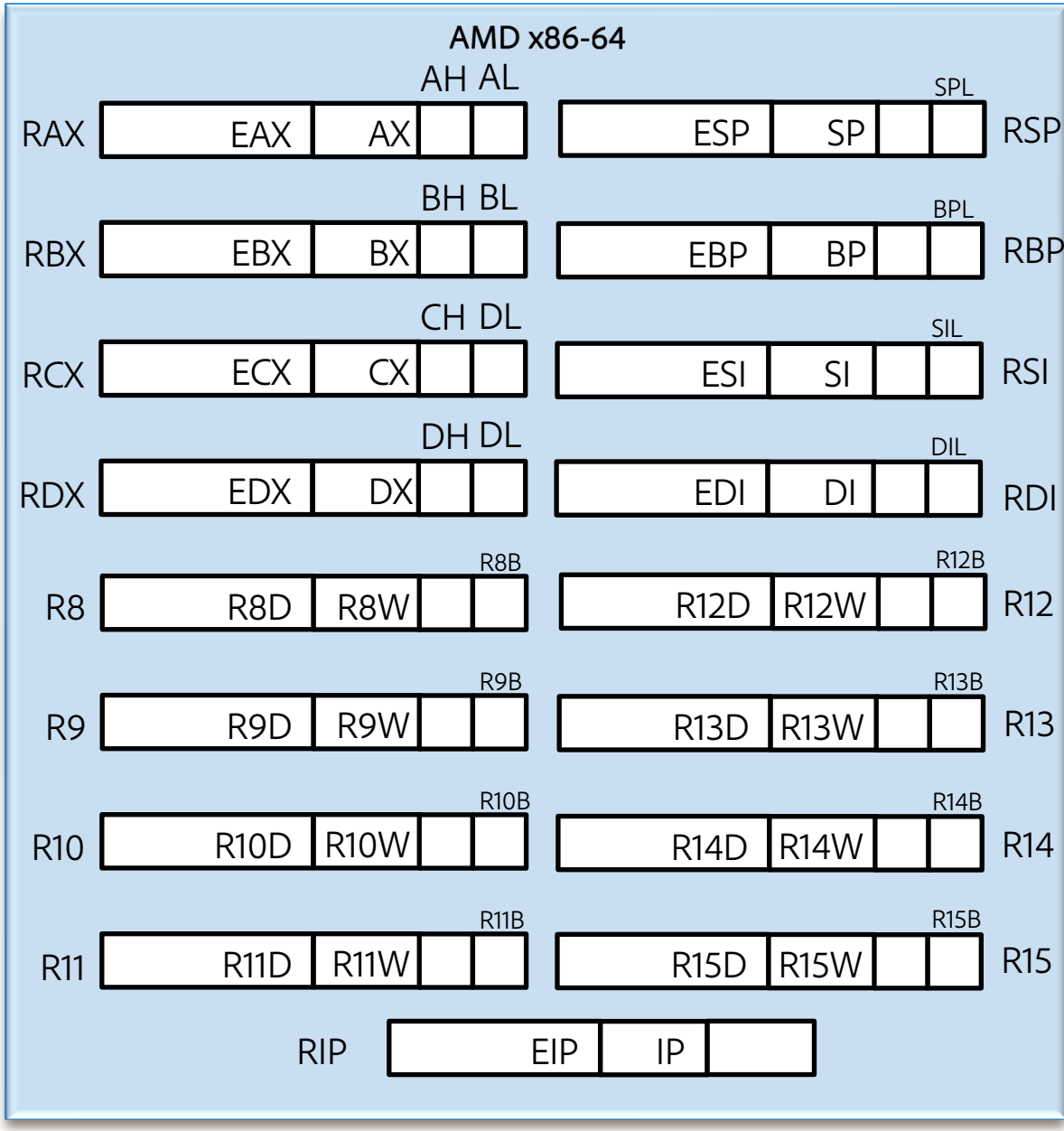




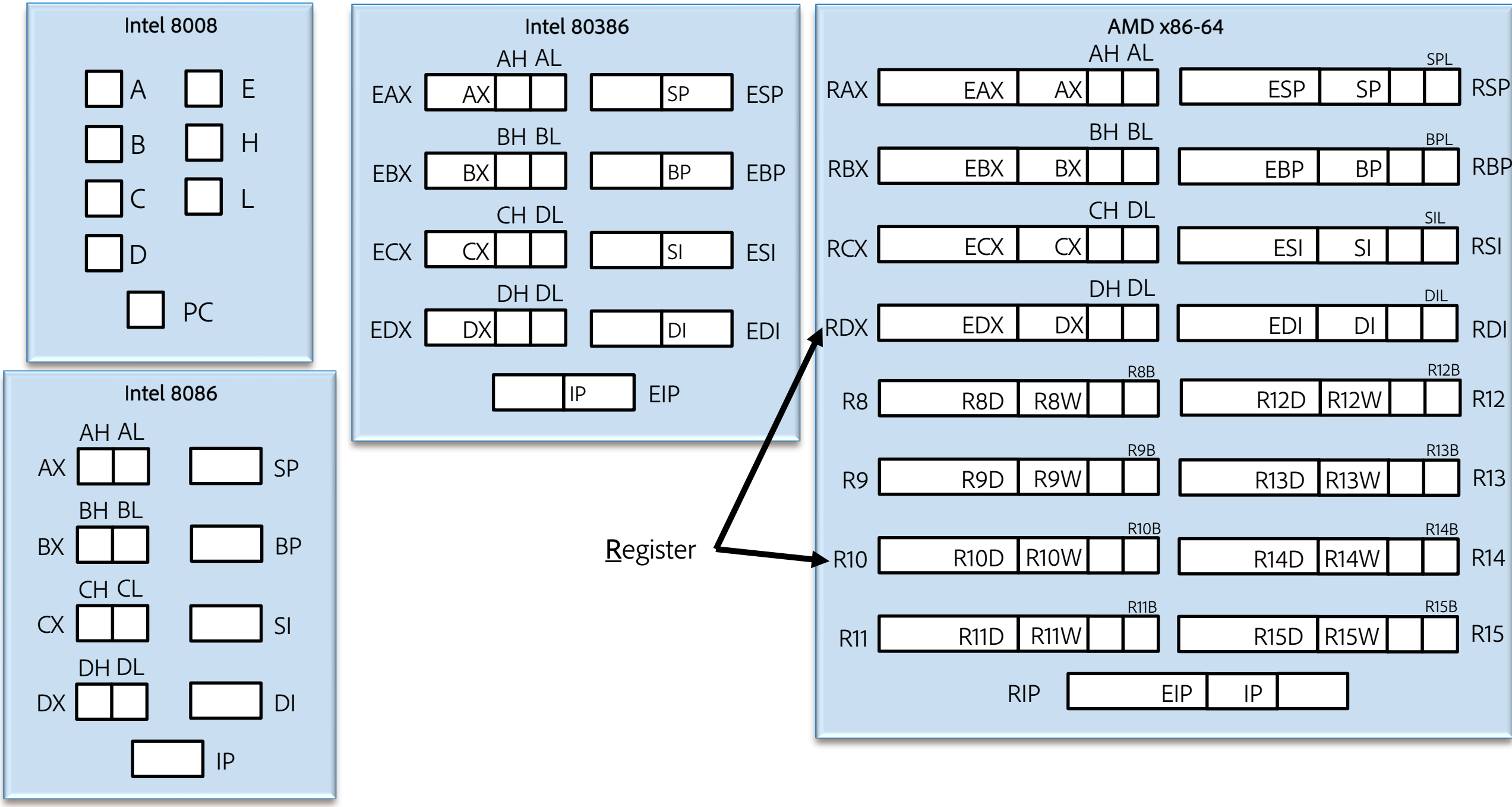


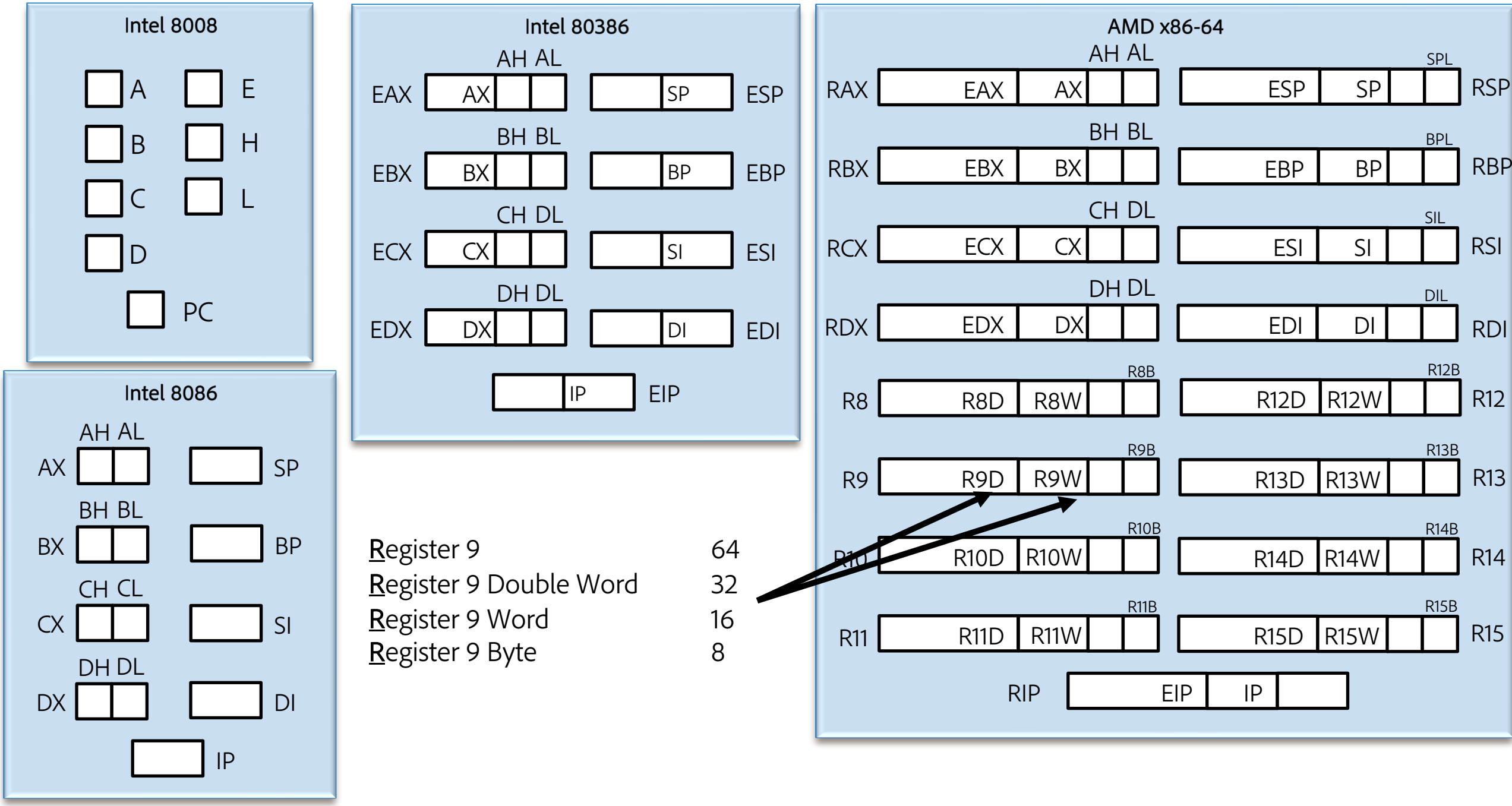




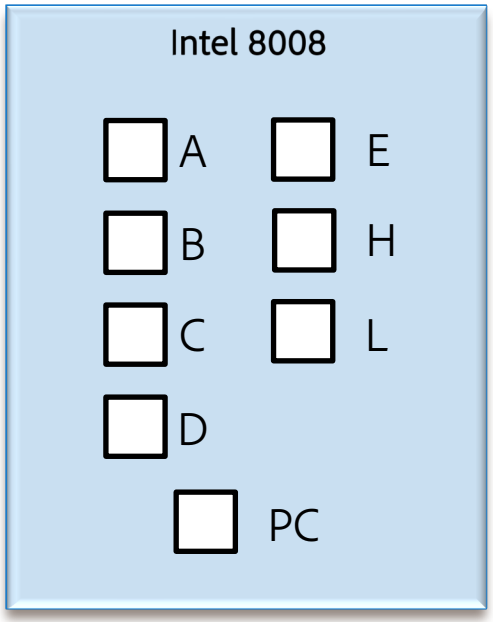


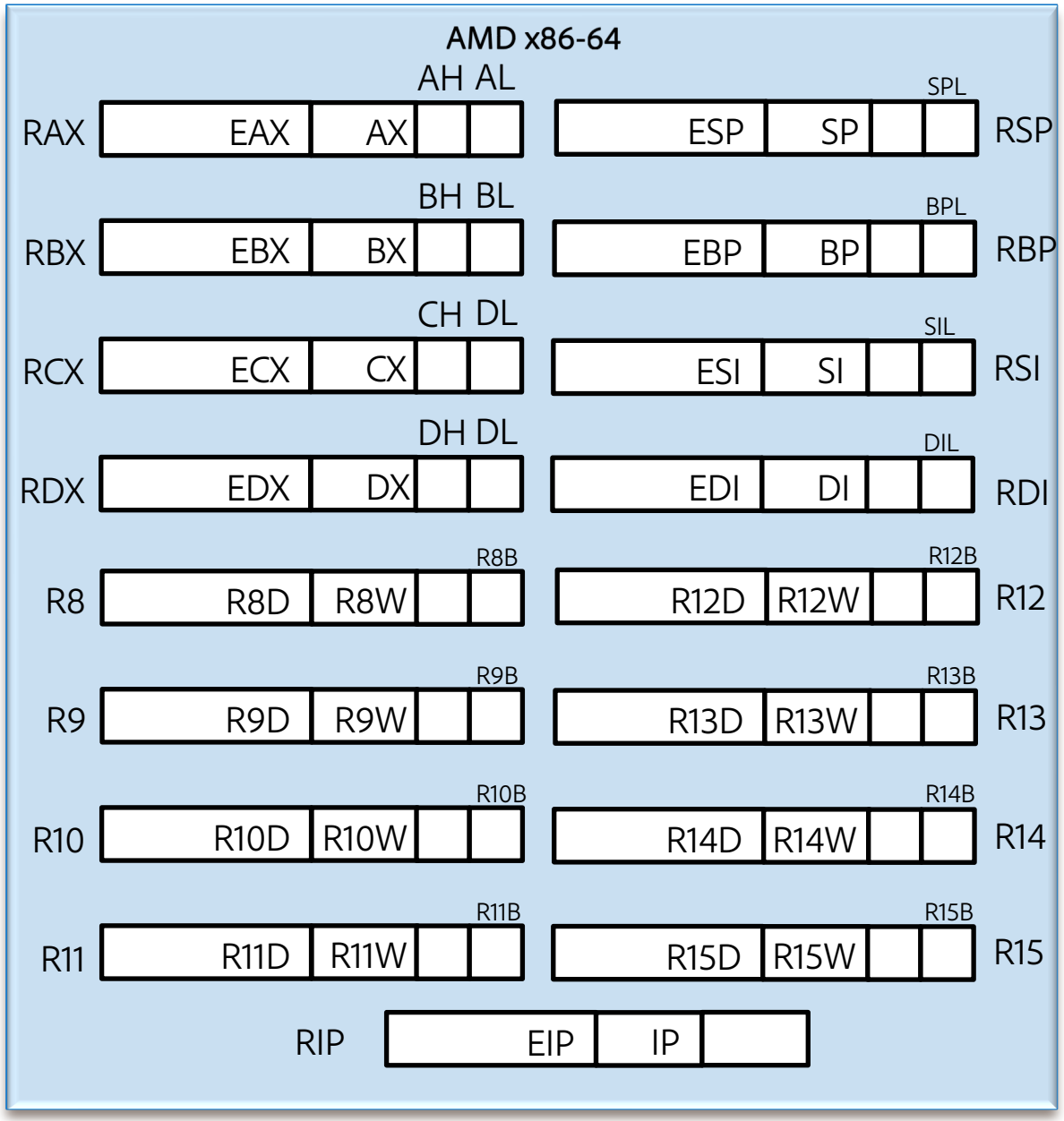
Adobe

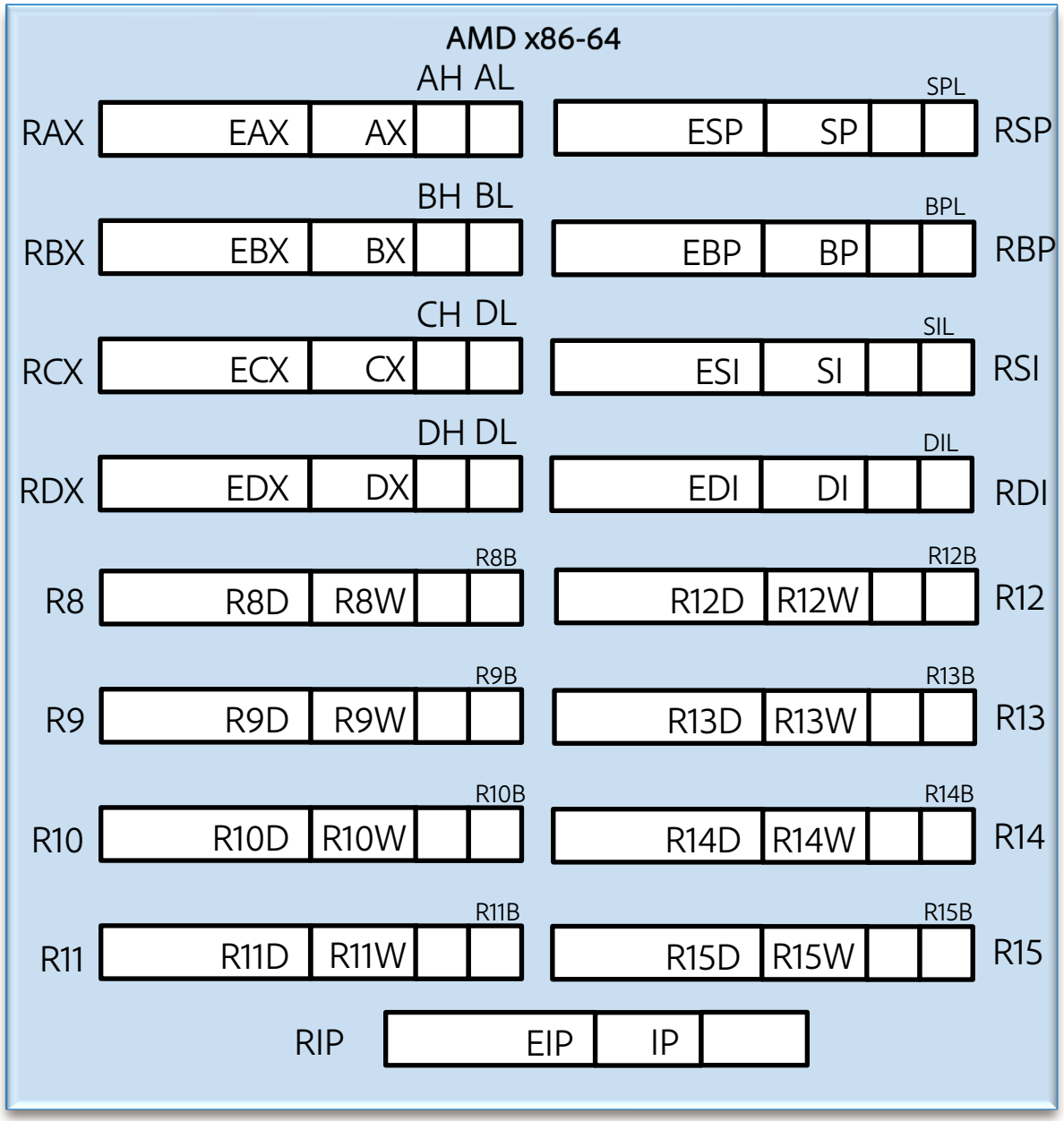




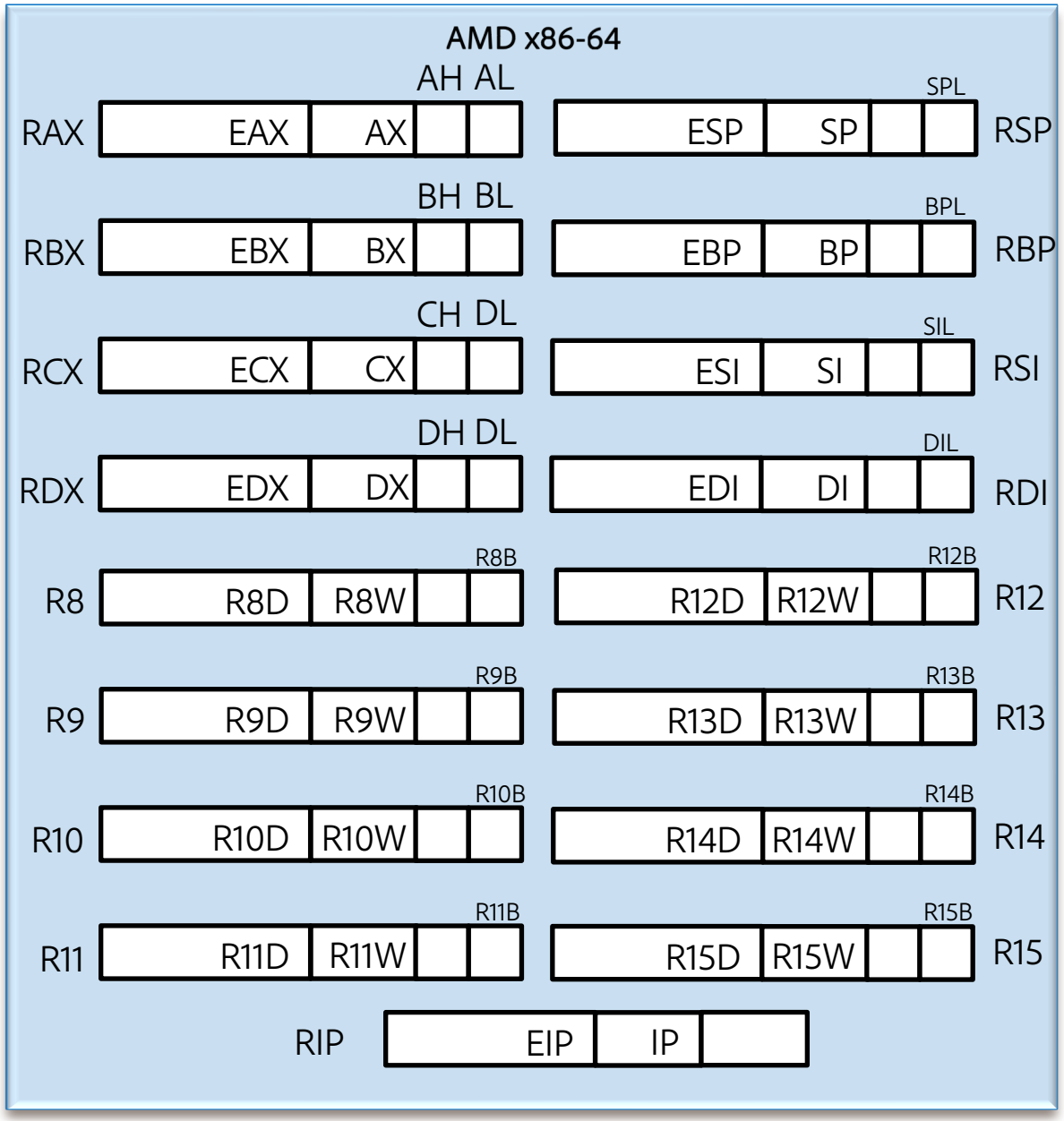
Registers



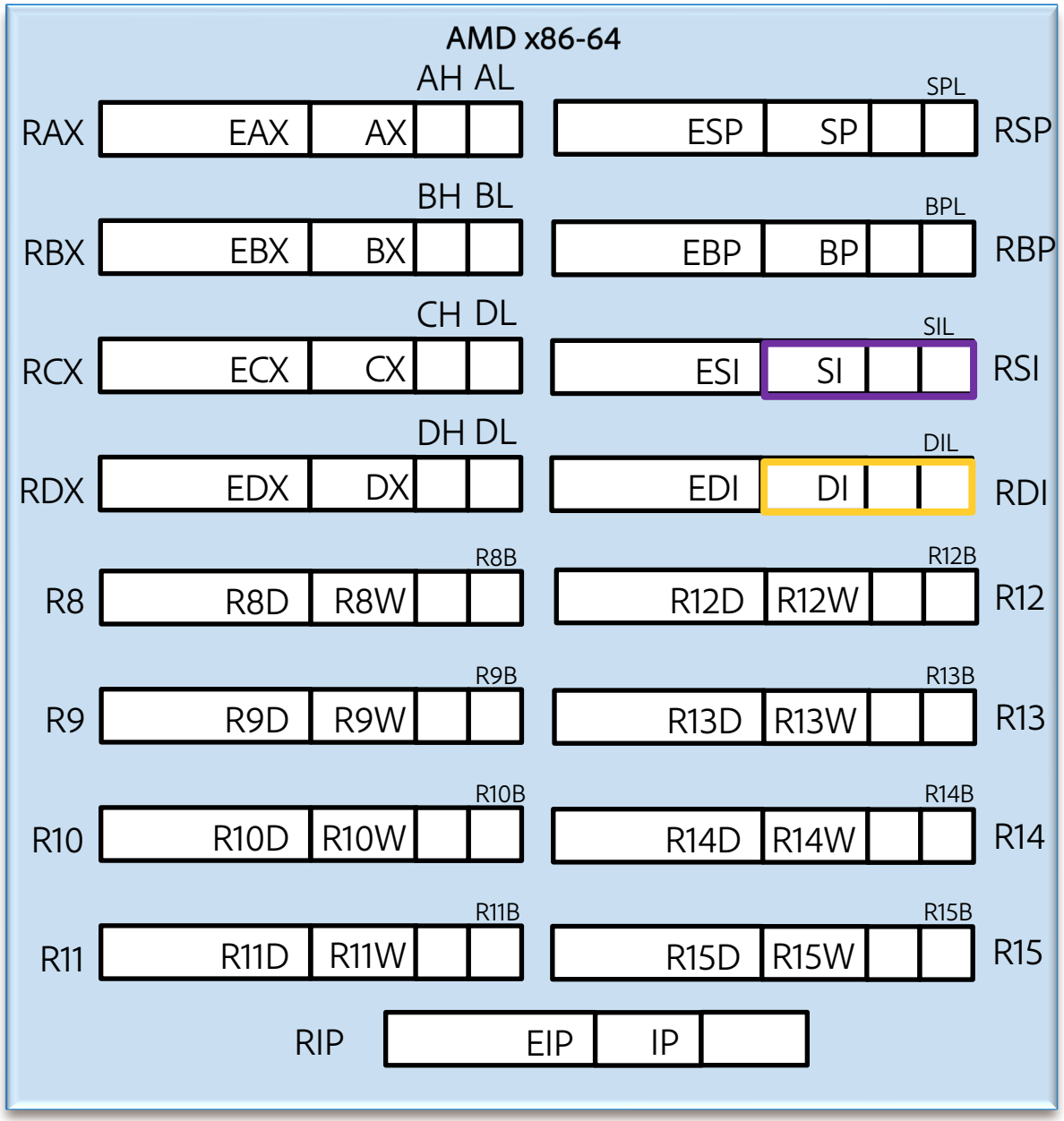




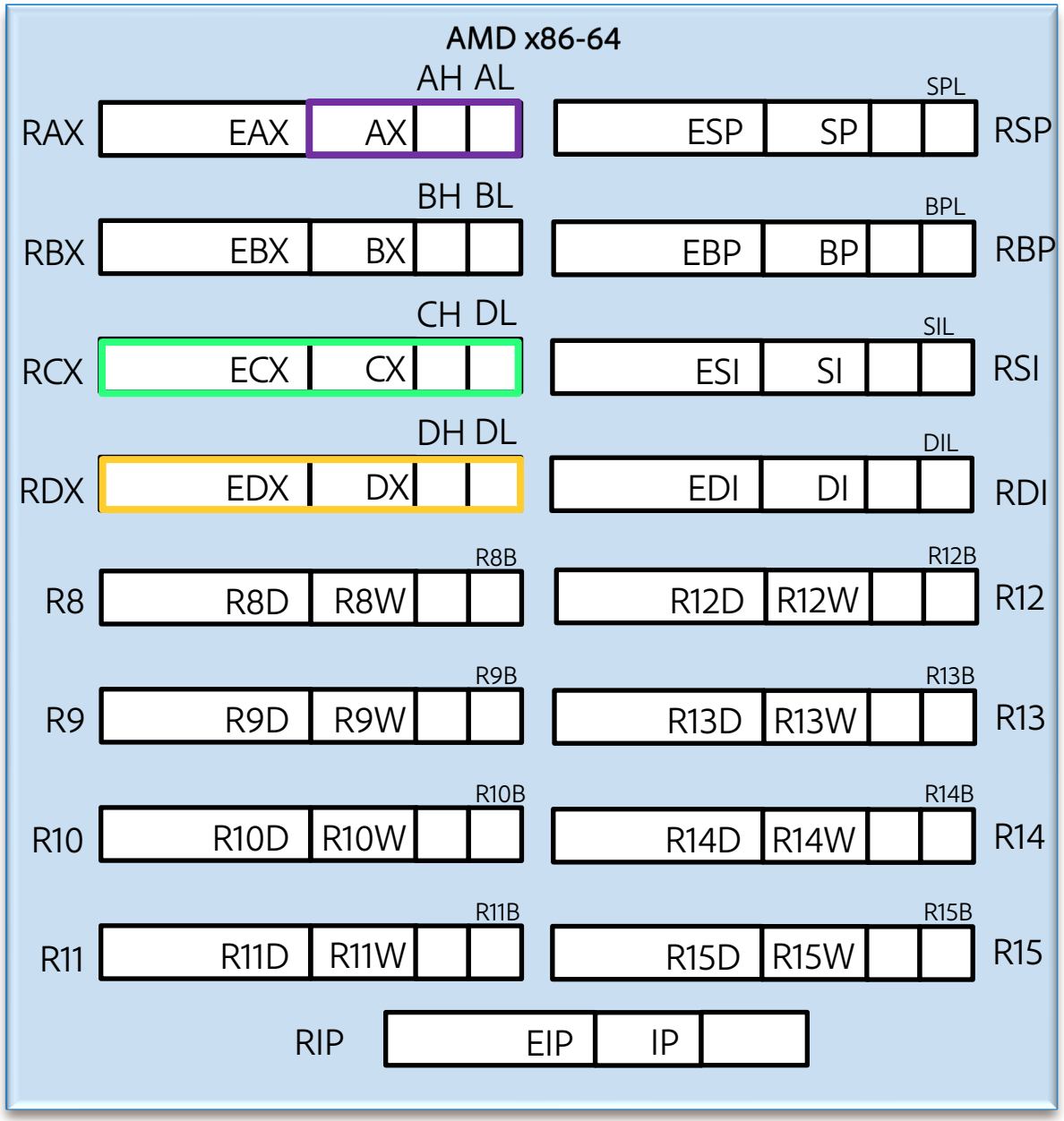

```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```




```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```




```
add esi edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

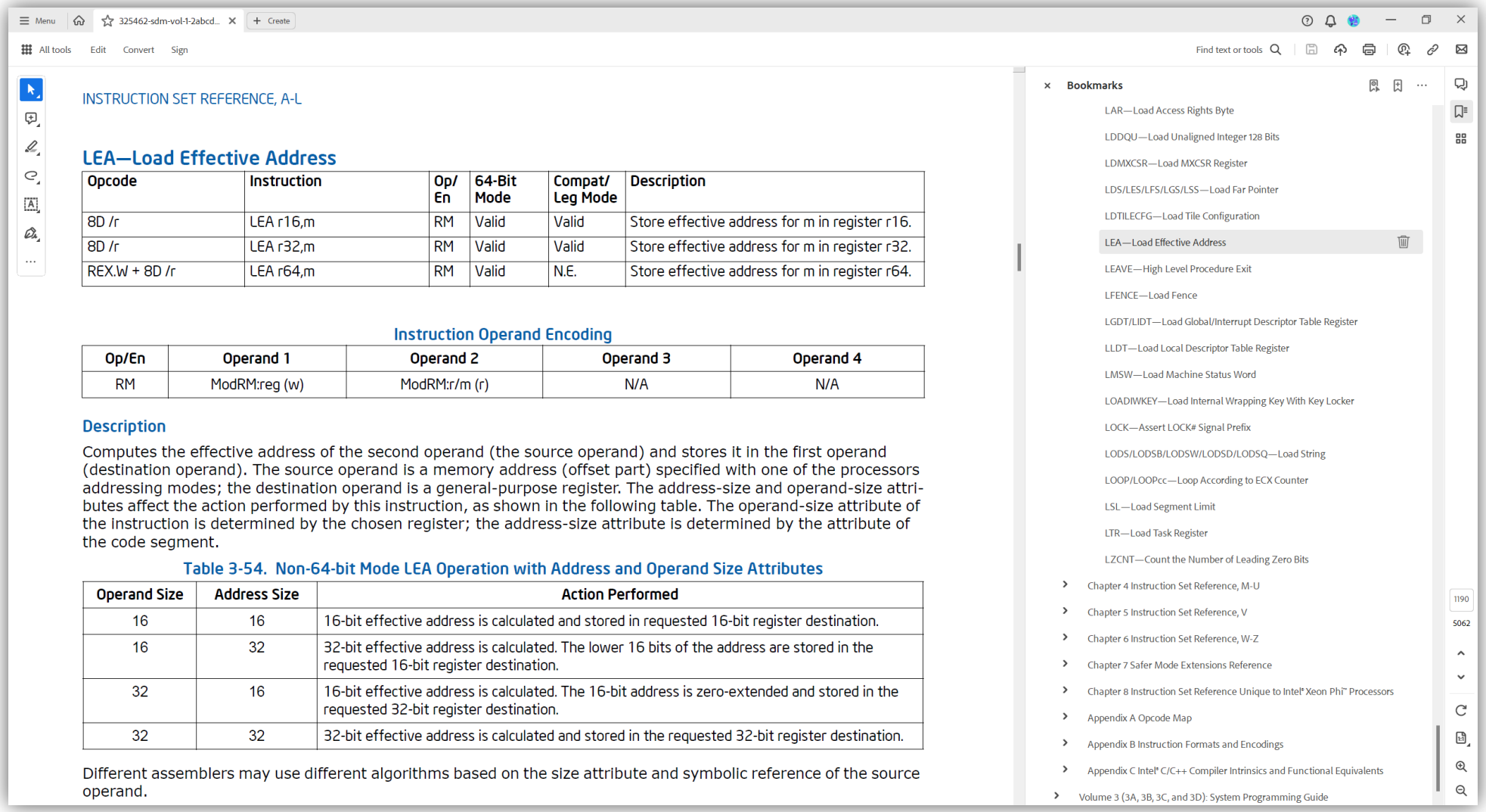



```
add esi, edi
lea eax [rdx + rcx]
add eax, esi
ret
```


```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

Intel 64 and IA-32 Architectures Software Developer's Manual

- 5062 pages (C++ standard is 2120)
- Excellent reference for Intel assembly
- There's a PDF freely available



INSTRUCTION SET REFERENCE, A-L

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for m in register r16.
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for m in register r32.
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for m in register r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-54. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

Bookmarks

- LAR—Load Access Rights Byte
- LDDQU—Load Unaligned Integer 128 Bits
- LDMXCSR—Load MXCSR Register
- LDS/LES/LFS/LGS/LSS—Load Far Pointer
- LDTILECFG—Load Tile Configuration
- LEA—Load Effective Address
- LEAVE—High Level Procedure Exit
- LFENCE—Load Fence
- LGDT/LIDT—Load Global/Interrupt Descriptor Table Register
- LLDT—Load Local Descriptor Table Register
- LMSW—Load Machine Status Word
- LOADIWKEY—Load Internal Wrapping Key With Key Locker
- LOCK—Assert LOCK# Signal Prefix
- LODS/LODSB/LODSW/LODSD/LODSQ—Load String
- LOOP/LOOPcc—Loop According to ECX Counter
- LSL—Load Segment Limit
- LTR—Load Task Register
- LZCNT—Count the Number of Leading Zero Bits
- Chapter 4 Instruction Set Reference, M-U
- Chapter 5 Instruction Set Reference, V
- Chapter 6 Instruction Set Reference, W-Z
- Chapter 7 Safer Mode Extensions Reference
- Chapter 8 Instruction Set Reference Unique to Intel® Xeon Phi™ Processors
- Appendix A Opcode Map
- Appendix B Instruction Formats and Encodings
- Appendix C Intel® C/C++ Compiler Intrinsics and Functional Equivalents
- Volume 3 (3A, 3B, 3C, and 3D): System Programming Guide

1190
5062

```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for m in register r16.
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for m in register r32.
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for m in register r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

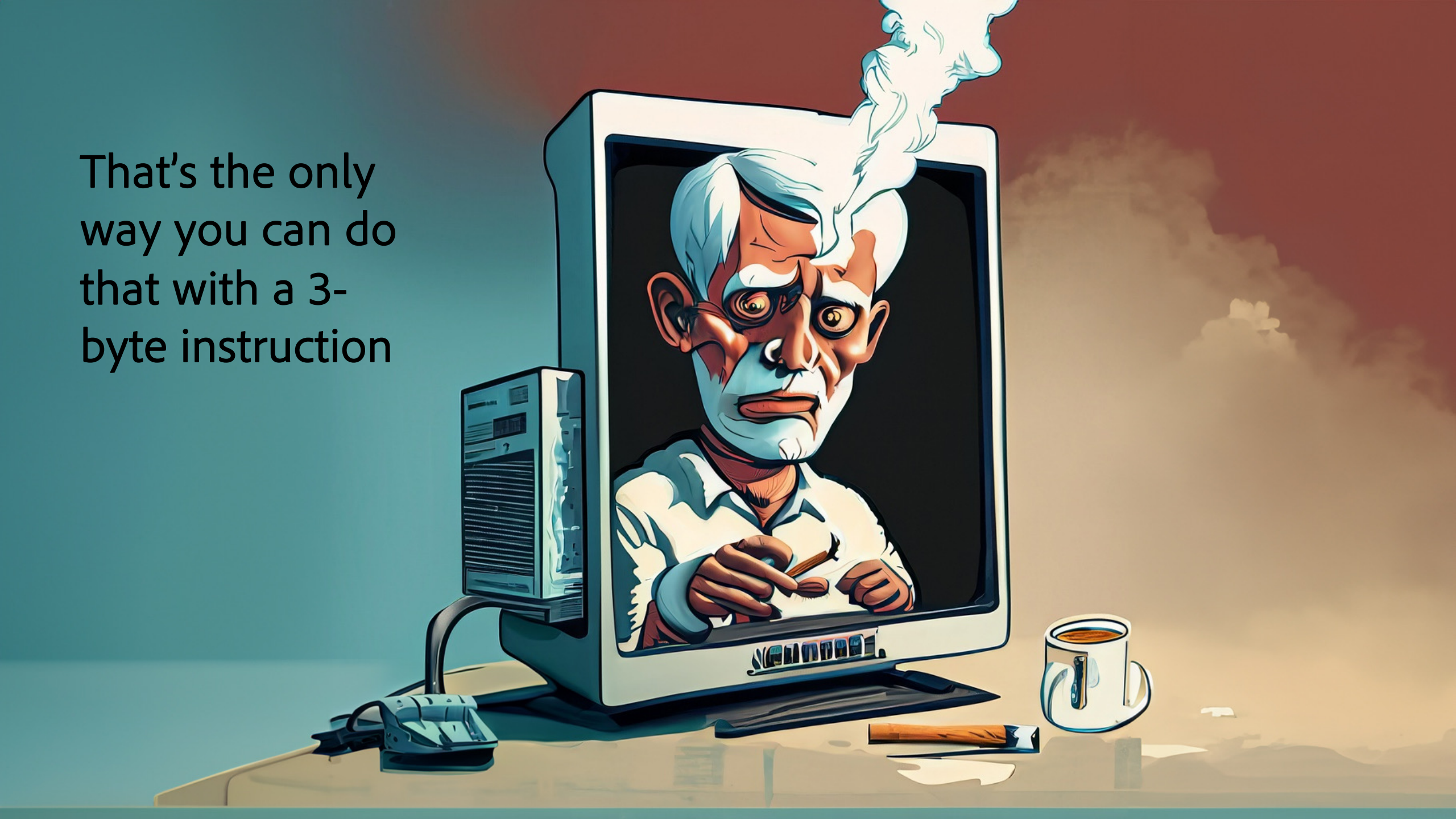
```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

Compiler, you are acting absurd

- There are add instructions y'know
- You're using 64 bit registers instead of 32 bit
- This is just weird

```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

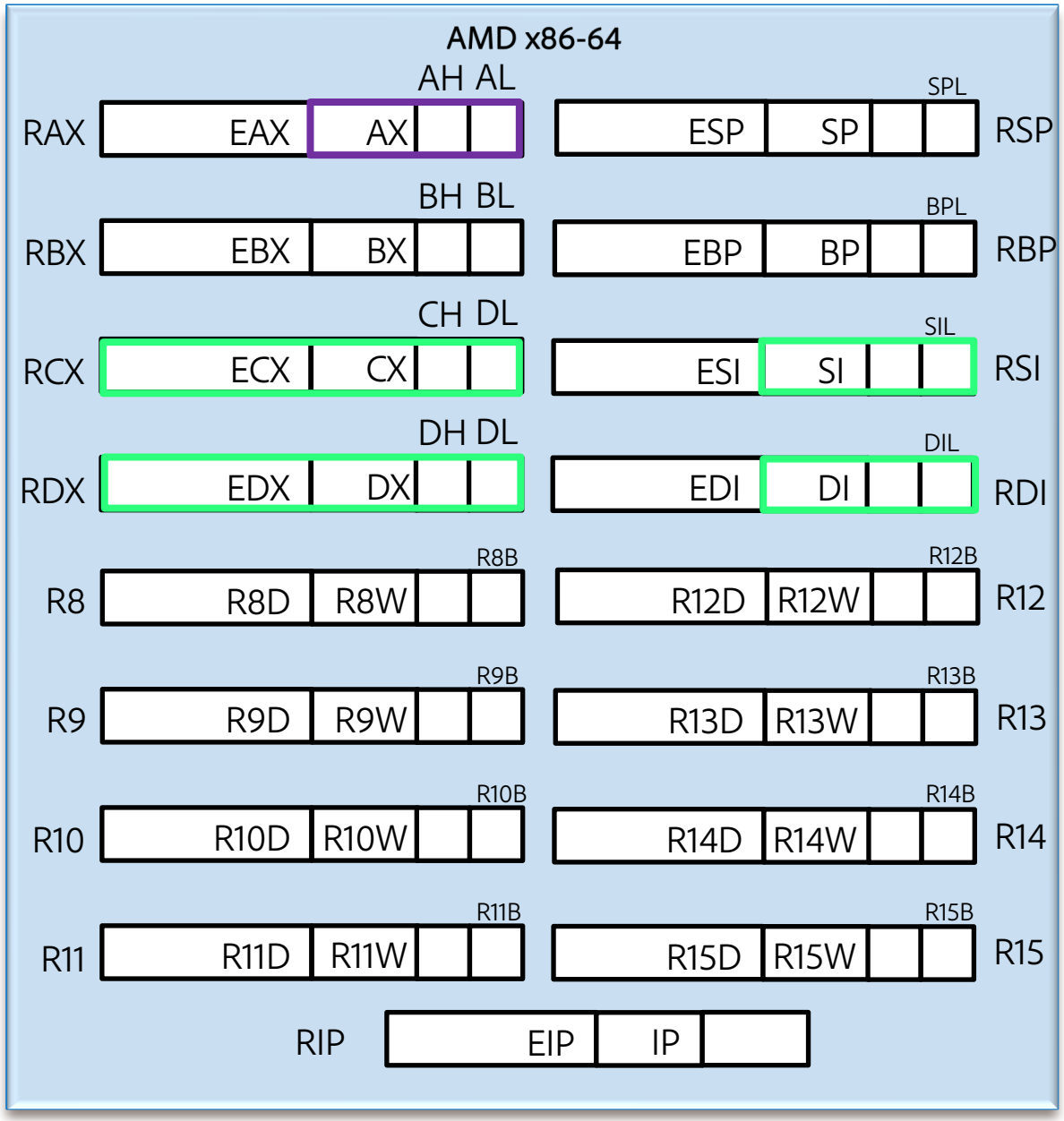
That’s the only way you can do that with a 3-byte instruction



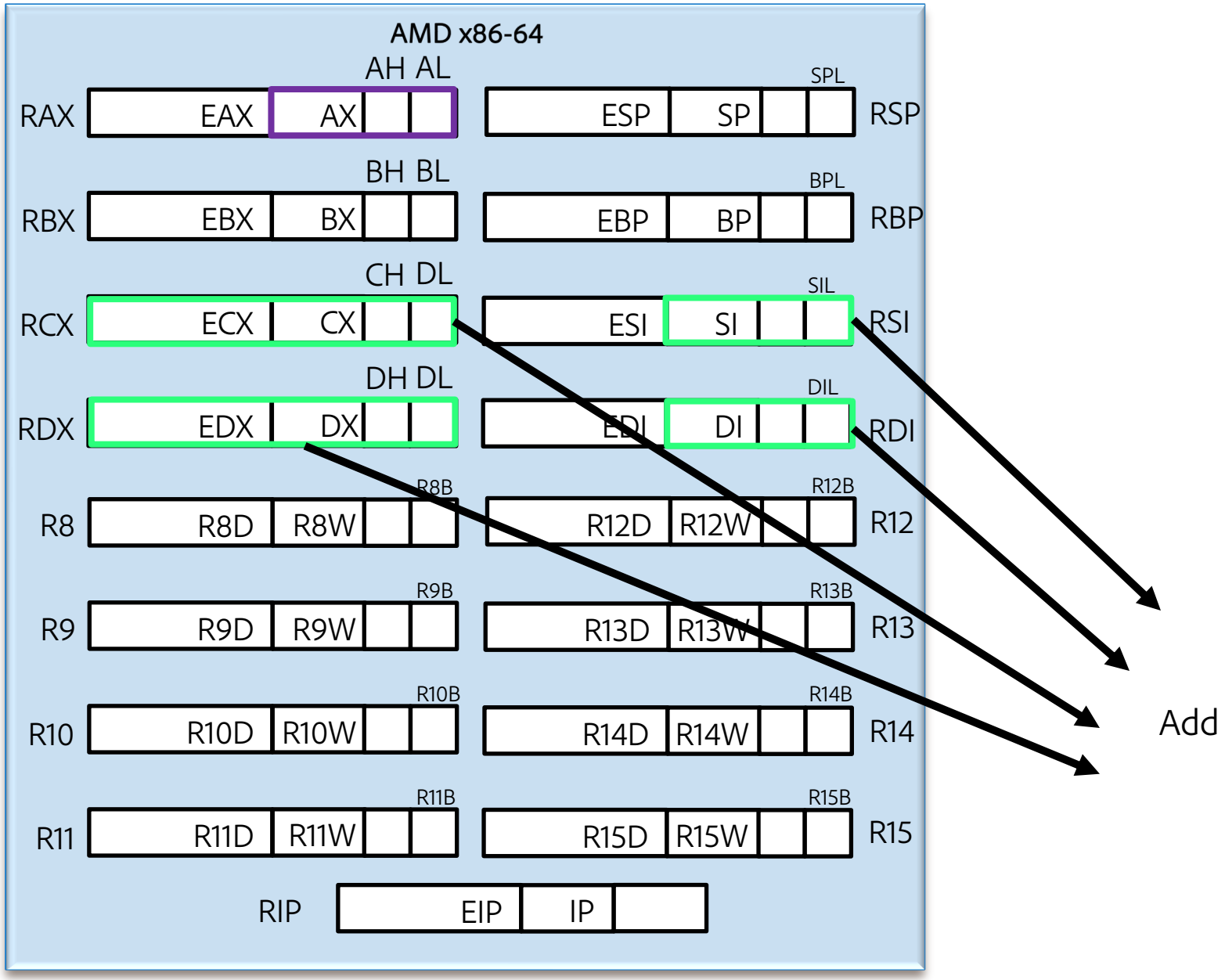
Optimized compiler generated assembly

- Fast
- Not always pretty
- Works

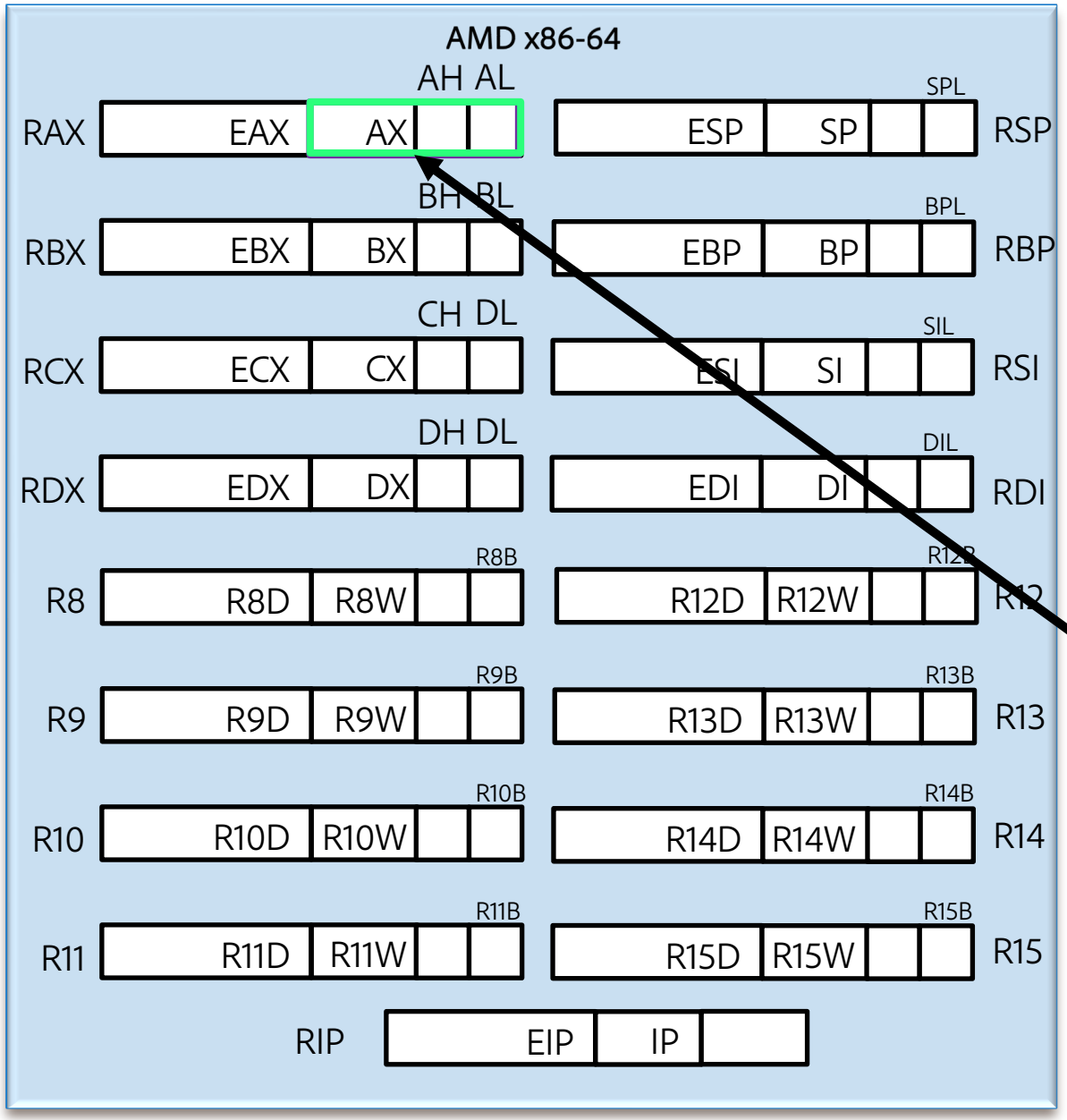
```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```




```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

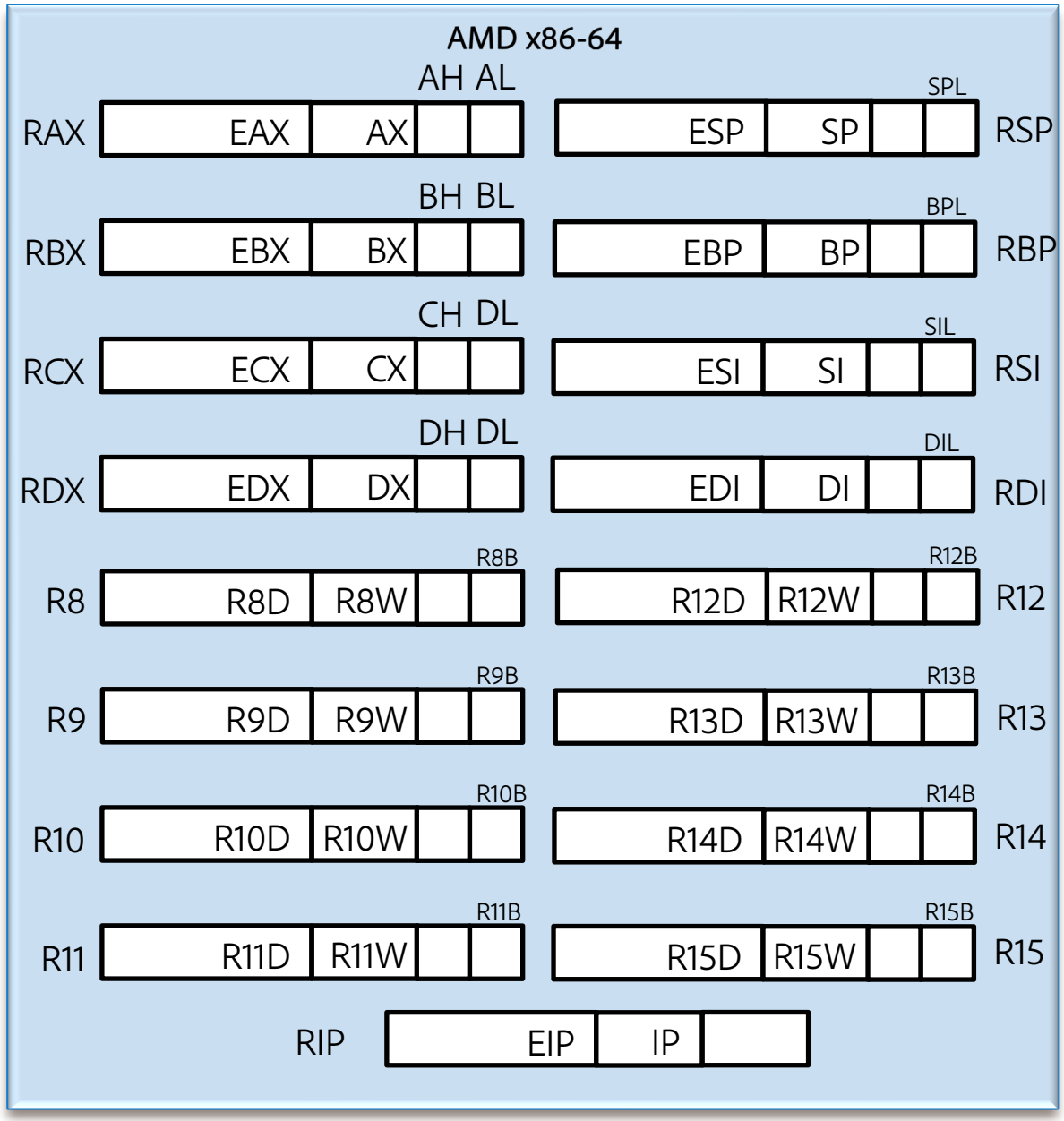



```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```




```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

Add



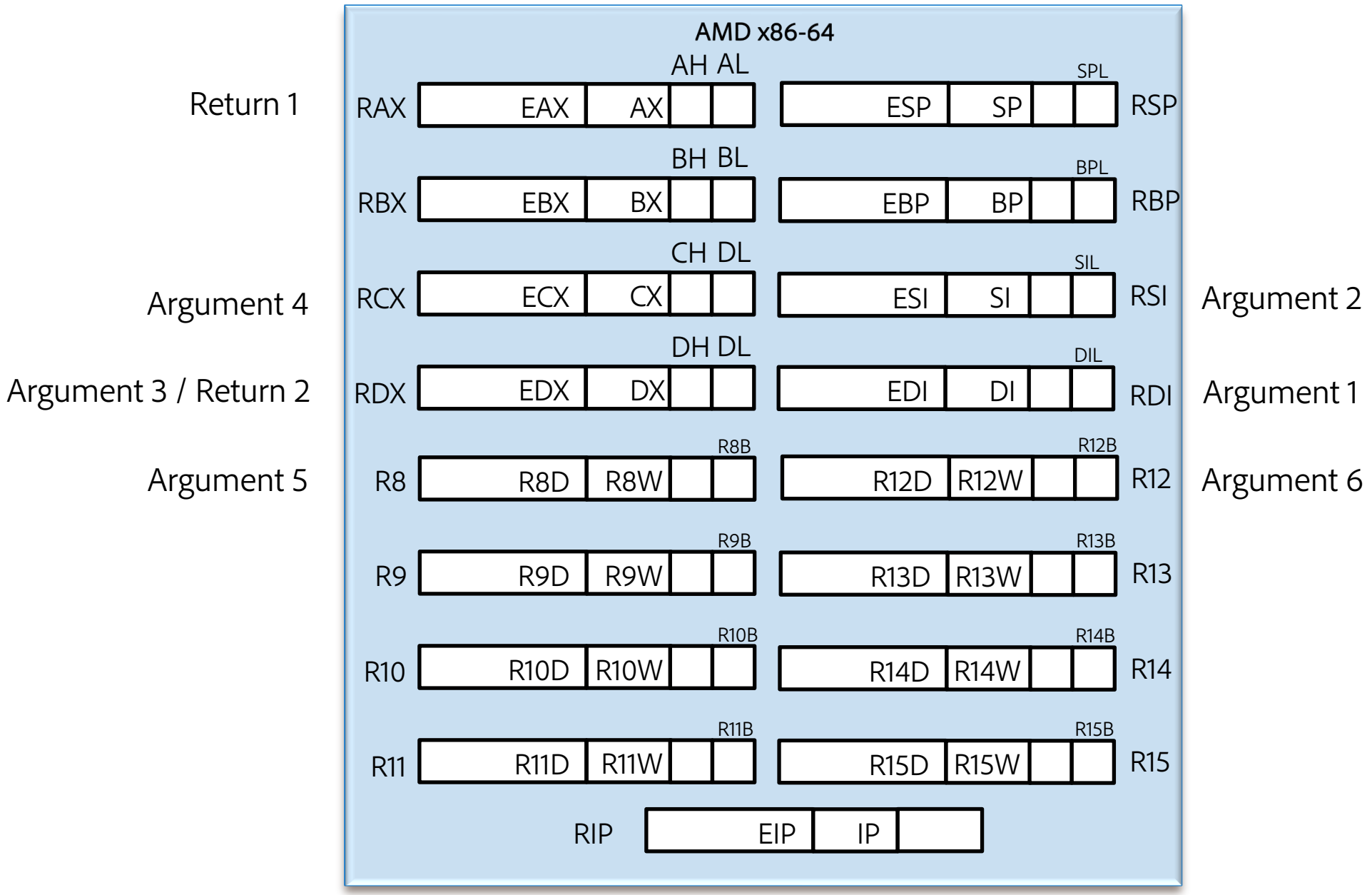
How do functions work?

```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

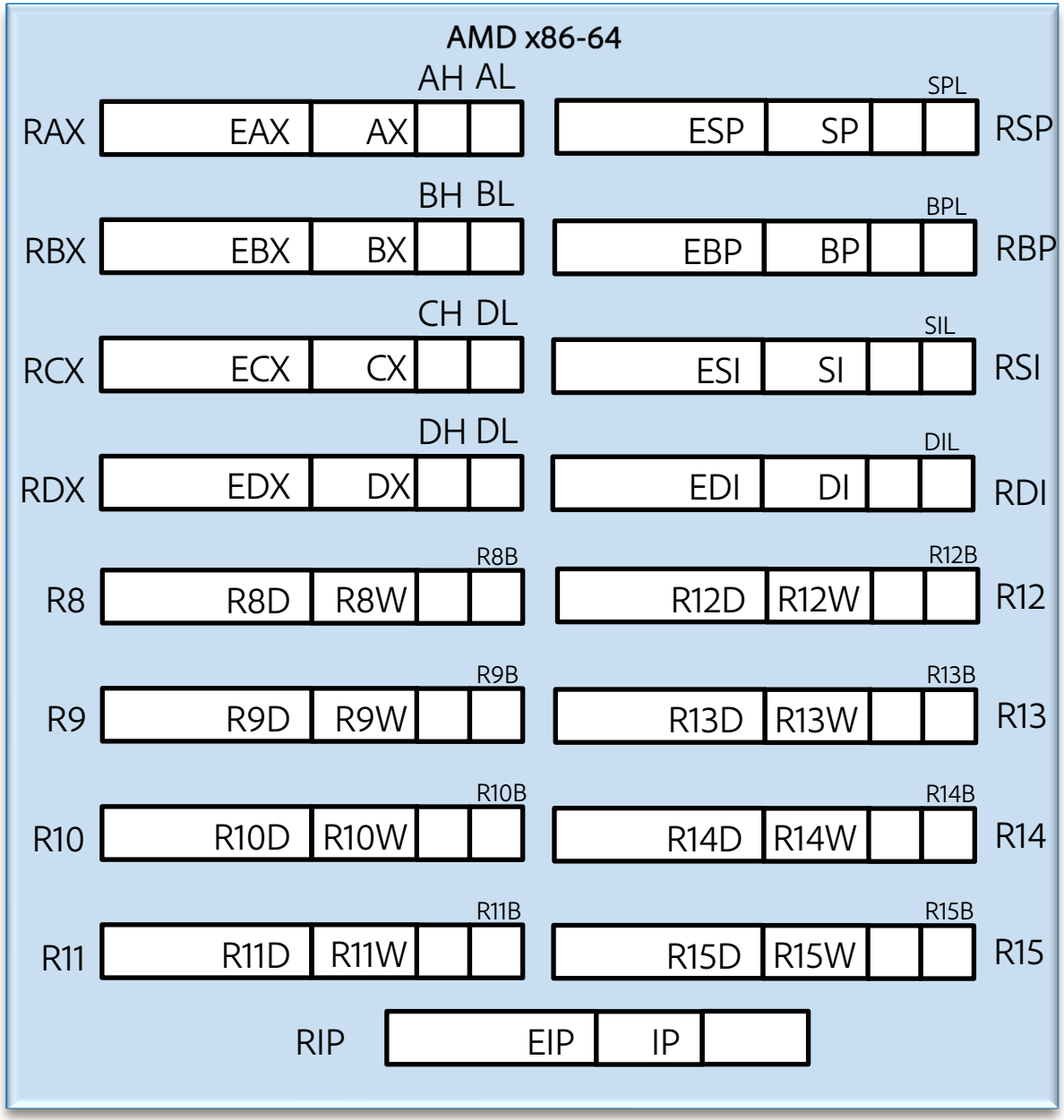
Call a function

- Set memory/registers to argument values according to a calling convention
- Make backups of registers you don't want clobbered
- `call` (put RIP on the stack and set RIP to the function address)

System V AMD64 ABI (Linux/macOS)

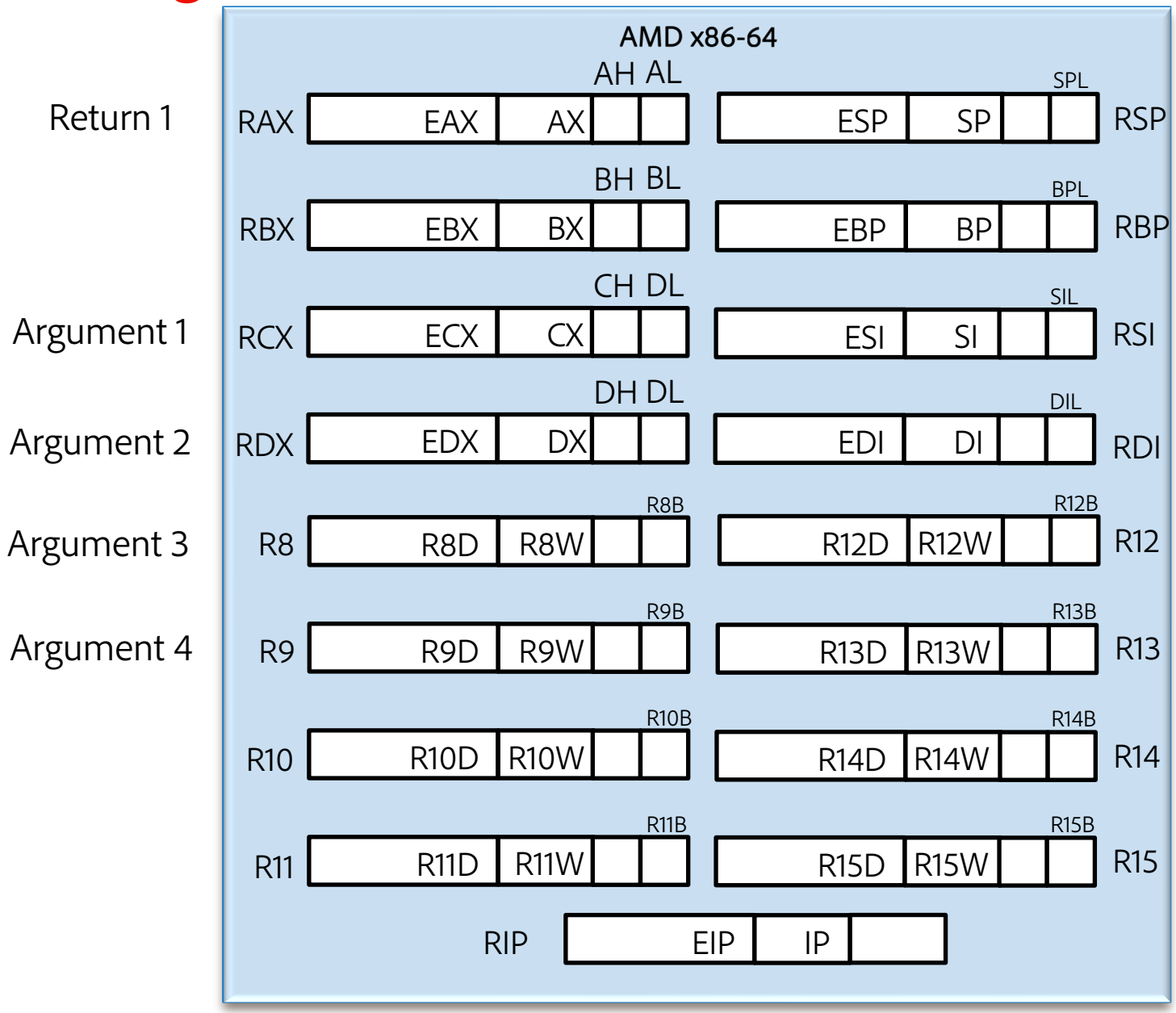


System V AMD64 ABI (Linux/macOS)

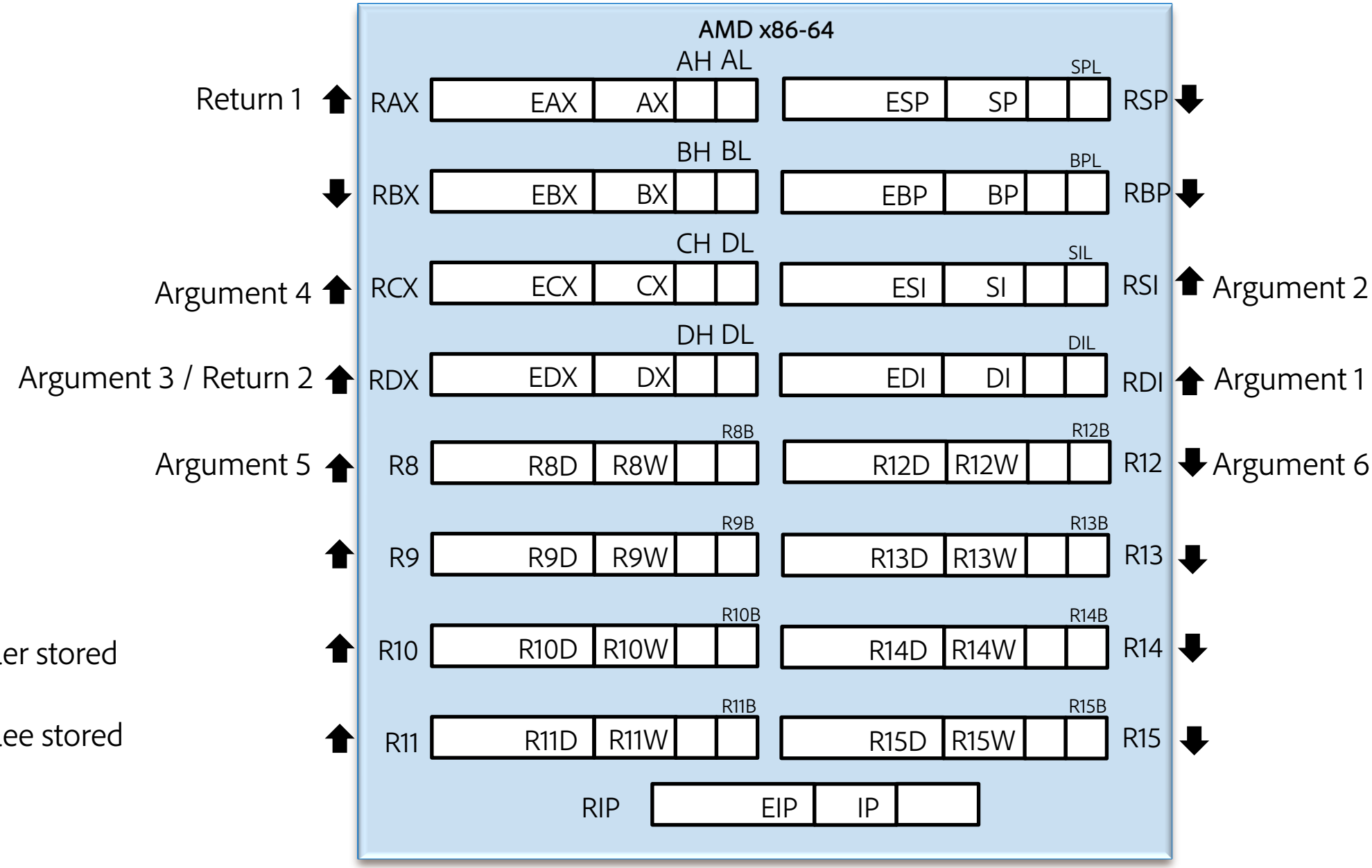



```
add esi, edi
lea eax, [rdx + rcx]
add eax, esi
ret
```

Microsoft x64 calling convention



System V AMD64 ABI (Linux/macOS)



MOV

```
mov r10, 255        ; set register 10 to the number 255
mov r10, [255]      ; set register 10 to the number at address 255 in RAM
mov [255], r10      ; set RAM at address 255 to the number in register 10
mov r11, r10        ; set register 11's number to register 10's number
```

MOV

```
mov r10, 255          ; set register 10 to the number 255
mov r10, [255]        ; set register 10 to the number at address 255 in RAM
mov [255], r10        ; set RAM at address 255 to the number in register 10
mov r11, r10          ; set register 11's number to register 10's number
```

MOV

```
mov r10, 255        ; set register 10 to the number 255
mov r10, [255]      ; set register 10 to the number at address 255 in RAM
mov [255], r10      ; set RAM at address 255 to the number in register 10
mov r11, r10        ; set register 11's number to register 10's number
```

MOV

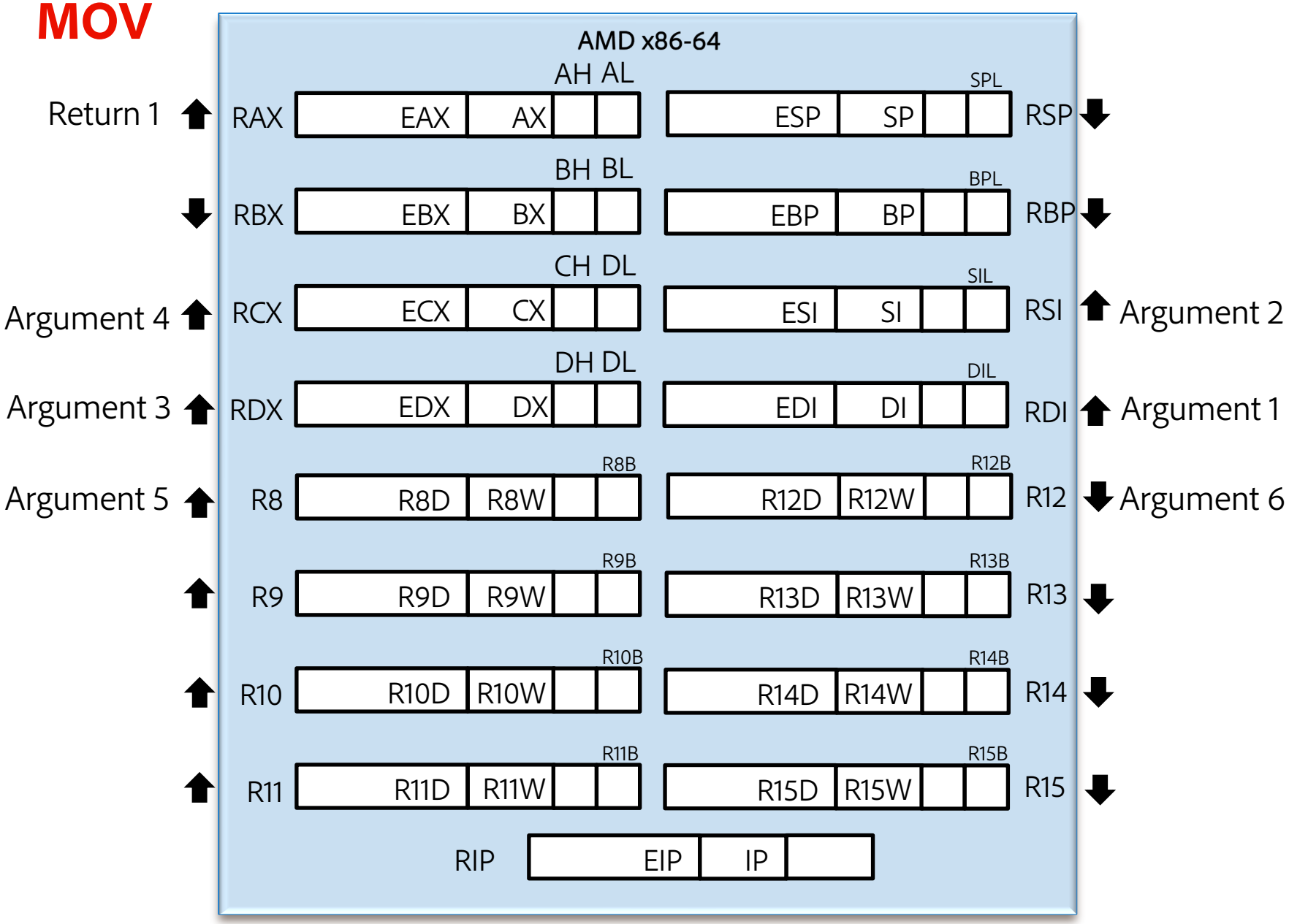
```
mov r10, 255        ; set register 10 to the number 255
mov r10, [255]      ; set register 10 to the number at address 255 in RAM
mov [255], r10      ; set RAM at address 255 to the number in register 10
mov r11, r10        ; set register 11's number to register 10's number
```

MOV

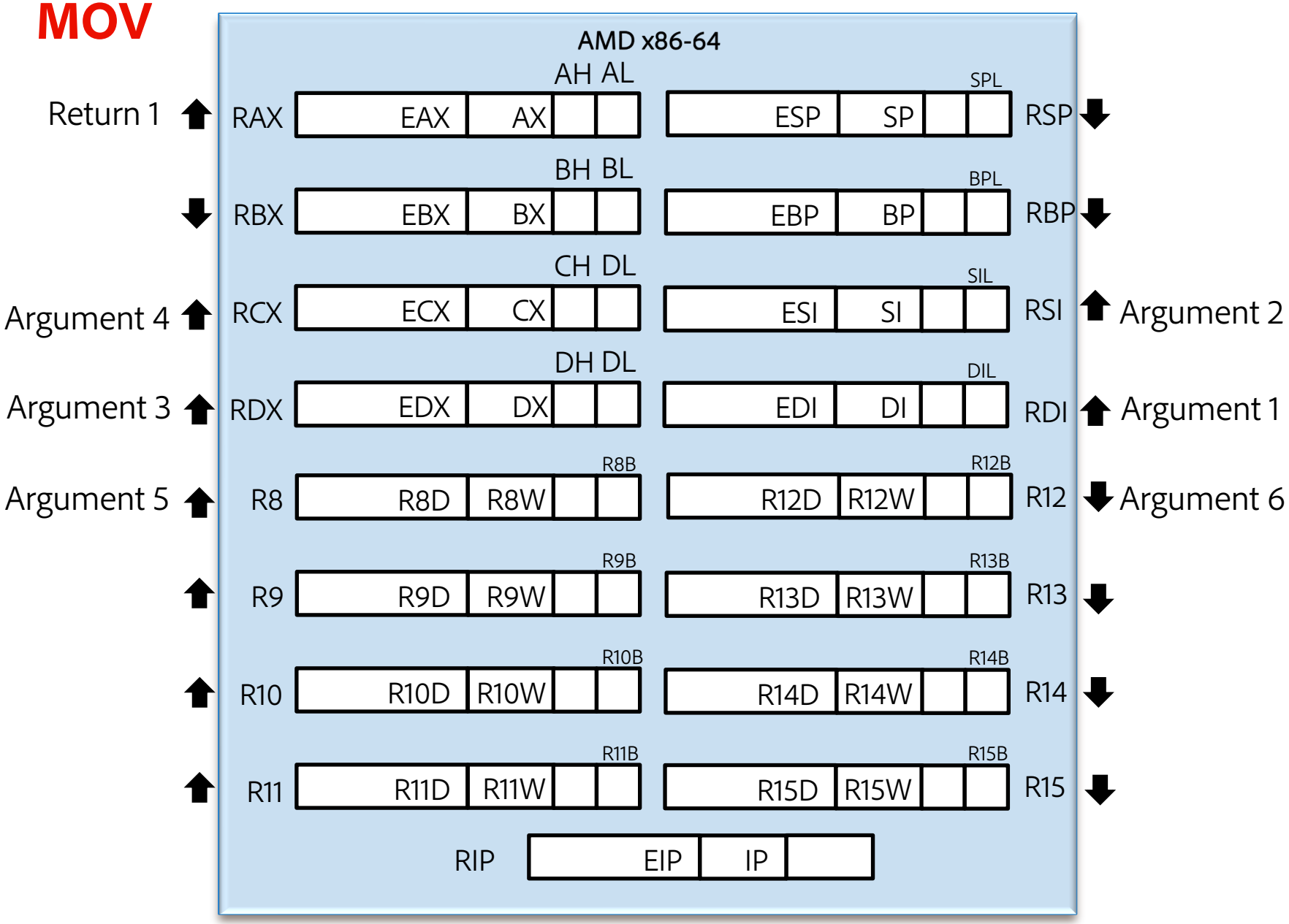
```
mov r10, 255        ; set register 10 to the number 255
mov r10, [255]      ; set register 10 to the number at address 255 in RAM
mov [255], r10      ; set RAM at address 255 to the number in register 10
mov r11, r10        ; set register 11's number to register 10's number
```

MOV

```
mov r10, 255
mov r10, [255]
mov [255], r10
mov r11, r10
```



MOV



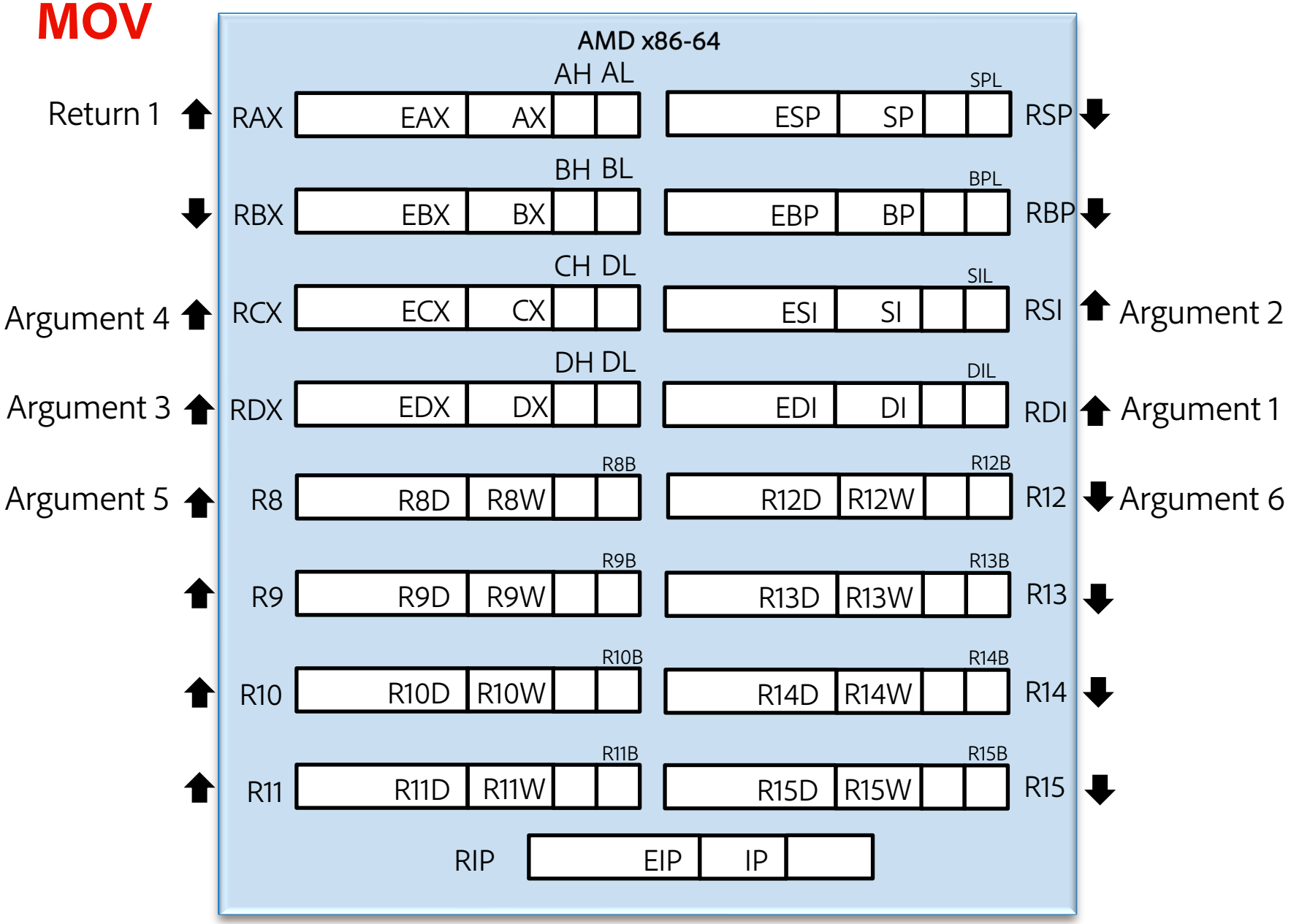

```
mov r10, 255
mov r10, [255]
mov [255], r10
mov r11, r10
```

Write,

```
int main() { return 3; }
```

, in assembly.

MOV




```
mov r10, 255
mov r10, [255]
mov [255], r10
mov r11, r10
```

Write,

```
int main() { return 3; }
```

, in assembly.

```
mov eax, 3
ret
```

What does this output?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << (void*)argv[argc] << std::endl;

}
```

What does this output?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << (void*)argv[argc] << std::endl;

}
```

```
0
```

What does this do?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << argv[argc+1] << std::endl;

}
```

What does this do?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << argv[argc+1] << std::endl;

}
```

```
SSH_TTY=/dev/pts/2
```

What does this do in Compiler Explorer?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << argv[argc+1] << std::endl;

}
```

```
LD_LIBRARY_PATH=/opt/compiler-explorer/gcc-snapshot/lib:/opt/compiler-explorer/gcc-snapshot/lib32:/opt/compiler-explorer/gcc-snapshot/lib64
```

How about this?

```
#include <iostream>

int main( int argc, char ** argv ) {

  std::cout << argv[argc+1] << std::endl;
  std::cout << argv[argc+2] << std::endl;
  std::cout << argv[argc+3] << std::endl;

}
```

```
SSH_TTY=/dev/pts/2
PWD=/home/david/Documents/C++Now/2023/examples
MAIL=/var/spool/mail/david
```

Hrm…

```
#include <iostream>

int main( int argc, char ** argv ) {

  for(int i = argc+1; argv[i]; ++i)
    std::cout << argv[i] << std::endl;

}
```

```
SSH_TTY=/dev/pts/2
PWD=/home/david/Documents/C++Now/2023/examples
MAIL=/var/spool/mail/david
SSH_TTY=/dev/pts/2
PWD=/home/david/Documents/C++Now/2023/examples
MAIL=/var/spool/mail/david
XDG_DATA_DIRS=/home/david/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share
OLDPWD=/home/david/Documents/C++Now/2023
LANG=en_US.UTF-8
MAKEFLAGS=
MFLAGS=
XDG_SESSION_TYPE=tty
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
SHLVL=1
MAKELEVEL=1
PATH=/home/david/Programs/ginger/bin:/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/lib/jvm/default/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl
MOTD_SHOWN=pam
SSH_CLIENT=fe80::f894:8d01:d0a4:16b%enp1s0 64412 22
LOGNAME=david
USER=david
DEBUGINFOD_URLS=https://debuginfod.archlinux.org
_=/usr/bin/make
MAKE_TERMOUT=/dev/pts/2
XDG_RUNTIME_DIR=/run/user/1000
VISUAL=vim
MAKE_TERMERR=/dev/pts/2
XDG_SESSION_CLASS=user
HOME=/home/david
HG=/usr/bin/hg
TERM=xterm
XDG_SESSION_ID=5
SSH_CONNECTION=fe80::f894:8d01:d0a4:16b%enp1s0 64412 fe80::2e27:d7ff:fe3b:680c%enp1s0 22
SHELL=/bin/bash
```

ELF Loading

```
position            content                     size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     4
                  [ argv[0] (pointer) ]         4   (program name)
                  [ argv[1] (pointer) ]         4
                  [ argv[..] (pointer) ]        4 * x
                  [ argv[n - 1] (pointer) ]     4
                  [ argv[n] (pointer) ]         4   (= NULL)

                  [ envp[0] (pointer) ]         4
                  [ envp[1] (pointer) ]         4
                  [ envp[..] (pointer) ]        4
                  [ envp[term] (pointer) ]      4   (= NULL)

                  [ auxv[0] (Elf32_auxv_t) ]    8
                  [ auxv[1] (Elf32_auxv_t) ]    8
                  [ auxv[..] (Elf32_auxv_t) ]   8
                  [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0

(0xbffffffc)      [ end marker ]                4   (= NULL)

(0xc0000000)      < top of stack >              0   (virtual)
------------------------------------------------------------------------
```

Credit to grugq and scut

ELF Loading

```
position            content                     size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     4
                  [ argv[0] (pointer) ]         4   (program name)
                  [ argv[1] (pointer) ]         4
                  [ argv[..] (pointer) ]        4 * x
                  [ argv[n - 1] (pointer) ]     4
                  [ argv[n] (pointer) ]         4   (= NULL)

                  [ envp[0] (pointer) ]         4
                  [ envp[1] (pointer) ]         4
                  [ envp[..] (pointer) ]        4
                  [ envp[term] (pointer) ]      4   (= NULL)

                  [ auxv[0] (Elf32_auxv_t) ]    8
                  [ auxv[1] (Elf32_auxv_t) ]    8
                  [ auxv[..] (Elf32_auxv_t) ]   8
                  [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0

(0xbffffffc)      [ end marker ]                4   (= NULL)

(0xc0000000)      < top of stack >              0   (virtual)
------------------------------------------------------------------------
```

Credit to grugq and scut

ELF Loading

```
position            content                     size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     4
                  [ argv[0] (pointer) ]         4   (program name)
                  [ argv[1] (pointer) ]         4
                  [ argv[..] (pointer) ]        4 * x
                  [ argv[n - 1] (pointer) ]     4
                  [ argv[n] (pointer) ]         4   (= NULL)

                  [ envp[0] (pointer) ]         4
                  [ envp[1] (pointer) ]         4
                  [ envp[..] (pointer) ]        4
                  [ envp[term] (pointer) ]      4   (= NULL)

                  [ auxv[0] (Elf32_auxv_t) ]    8
                  [ auxv[1] (Elf32_auxv_t) ]    8
                  [ auxv[..] (Elf32_auxv_t) ]   8
                  [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0

(0xbffffffc)      [ end marker ]                4   (= NULL)

(0xc0000000)      < top of stack >              0   (virtual)
------------------------------------------------------------------------
```

Credit to grugq and scut

ELF Loading

```
position            content                     size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     4
                  [ argv[0] (pointer) ]         4   (program name)
                  [ argv[1] (pointer) ]         4
                  [ argv[..] (pointer) ]        4 * x
                  [ argv[n - 1] (pointer) ]     4
                  [ argv[n] (pointer) ]         4   (= NULL)

                  [ envp[0] (pointer) ]         4
                  [ envp[1] (pointer) ]         4
                  [ envp[..] (pointer) ]        4
                  [ envp[term] (pointer) ]      4   (= NULL)

                  [ auxv[0] (Elf32_auxv_t) ]    8
                  [ auxv[1] (Elf32_auxv_t) ]    8
                  [ auxv[..] (Elf32_auxv_t) ]   8
                  [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0

(0xbffffffc)      [ end marker ]                4   (= NULL)

(0xc0000000)      < top of stack >              0   (virtual)
------------------------------------------------------------------------
```

Credit to grugq and scut

ELF Loading

```
position            content                     size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     4
                  [ argv[0] (pointer) ]         4   (program name)
                  [ argv[1] (pointer) ]         4
                  [ argv[..] (pointer) ]        4 * x
                  [ argv[n - 1] (pointer) ]     4
                  [ argv[n] (pointer) ]         4   (= NULL)

                  [ envp[0] (pointer) ]         4
                  [ envp[1] (pointer) ]         4
                  [ envp[..] (pointer) ]        4
                  [ envp[term] (pointer) ]      4   (= NULL)

                  [ auxv[0] (Elf32_auxv_t) ]    8
                  [ auxv[1] (Elf32_auxv_t) ]    8
                  [ auxv[..] (Elf32_auxv_t) ]   8
                  [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0

(0xbffffffc)      [ end marker ]                4   (= NULL)

(0xc0000000)      < top of stack >              0   (virtual)
------------------------------------------------------------------------
```

Credit to grugq and scut

ELF Auxiliary Vectors

LD_SHOW_AUXV=1 ./main

```
AT_SYSINFO_EHDR:      0x7ffd9e178000
AT_MINSIGSTKSZ:       1440
AT_HWCAP:             b7ebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x557a0890f040
AT_PHENT:             56
AT_PHNUM:             13
AT_BASE:              0x7fafa043f000
AT_FLAGS:             0x0
AT_ENTRY:             0x557a08910080
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7ffd9e166599
AT_HWCAP2:            0x0
AT_EXECFN:            ./main_argv
AT_PLATFORM:          x86_64
```

ELF Auxiliary Vectors

User ID and Group ID

```
AT_SYSINFO_EHDR:      0x7ffd9e178000
AT_MINSIGSTKSZ:       1440
AT_HWCAP:             b7ebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x557a0890f040
AT_PHENT:             56
AT_PHNUM:             13
AT_BASE:              0x7fafa043f000
AT_FLAGS:             0x0
AT_ENTRY:             0x557a08910080
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7ffd9e166599
AT_HWCAP2:            0x0
AT_EXECFN:            ./main_argv
AT_PLATFORM:          x86_64
```

ELF Auxiliary Vectors

```
AT_SYSINFO_EHDR:      0x7ffd9e178000
AT_MINSIGSTKSZ:       1440
AT_HWCAP:             b7ebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x557a0890f040
AT_PHENT:             56
AT_PHNUM:             13
AT_BASE:              0x7fafa043f000
AT_FLAGS:             0x0
AT_ENTRY:             0x557a08910080
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7ffd9e166599
AT_HWCAP2:            0x0
AT_EXECFN:            ./main_argv
AT_PLATFORM:          x86_64
```

Pointer to 16 random bytes used to implement a stack canary.

Pointer to page containing the Virtual Dynamic Shared Object (VDSO)

ELF Auxiliary Vectors

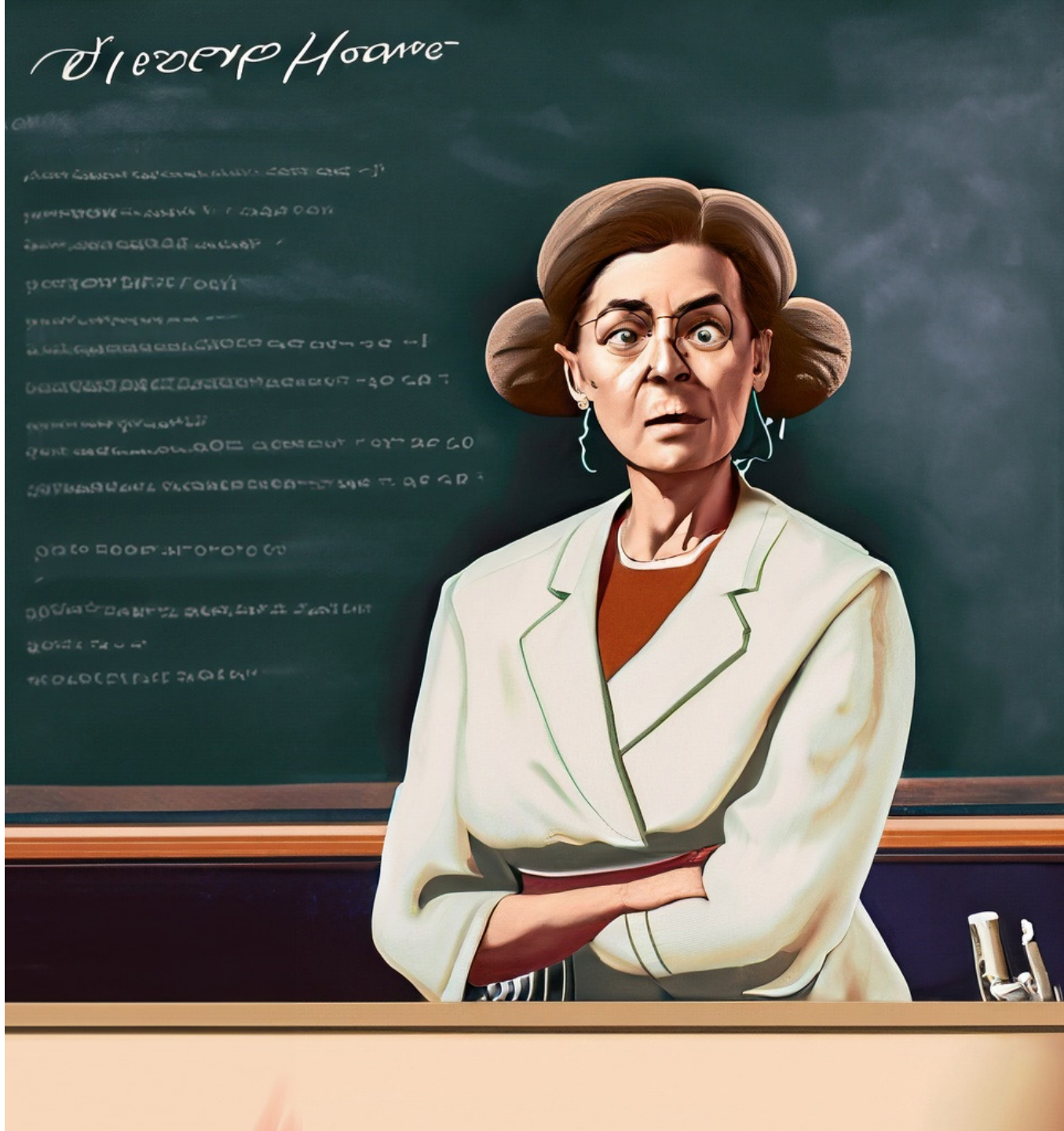
```
AT_SYSINFO_EHDR:      0x7ffd9e178000
AT_MINSIGSTKSZ:       1440
AT_HWCAP:             b7ebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x557a0890f040
AT_PHENT:             56
AT_PHNUM:             13
AT_BASE:              0x7fafa043f000
AT_FLAGS:             0x0
AT_ENTRY:             0x557a08910080
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7ffd9e166599
AT_HWCAP2:            0x0
AT_EXECFN:            ./main_argv
AT_PLATFORM:          x86_64
```

Virtual Dynamic Shared Object (VDSO)

In the old days, system calls were made using interrupts.

```
mov eax, 1      ; exit system call
mov ebx, 42     ; argument
int 0x80
```

And they were slow…

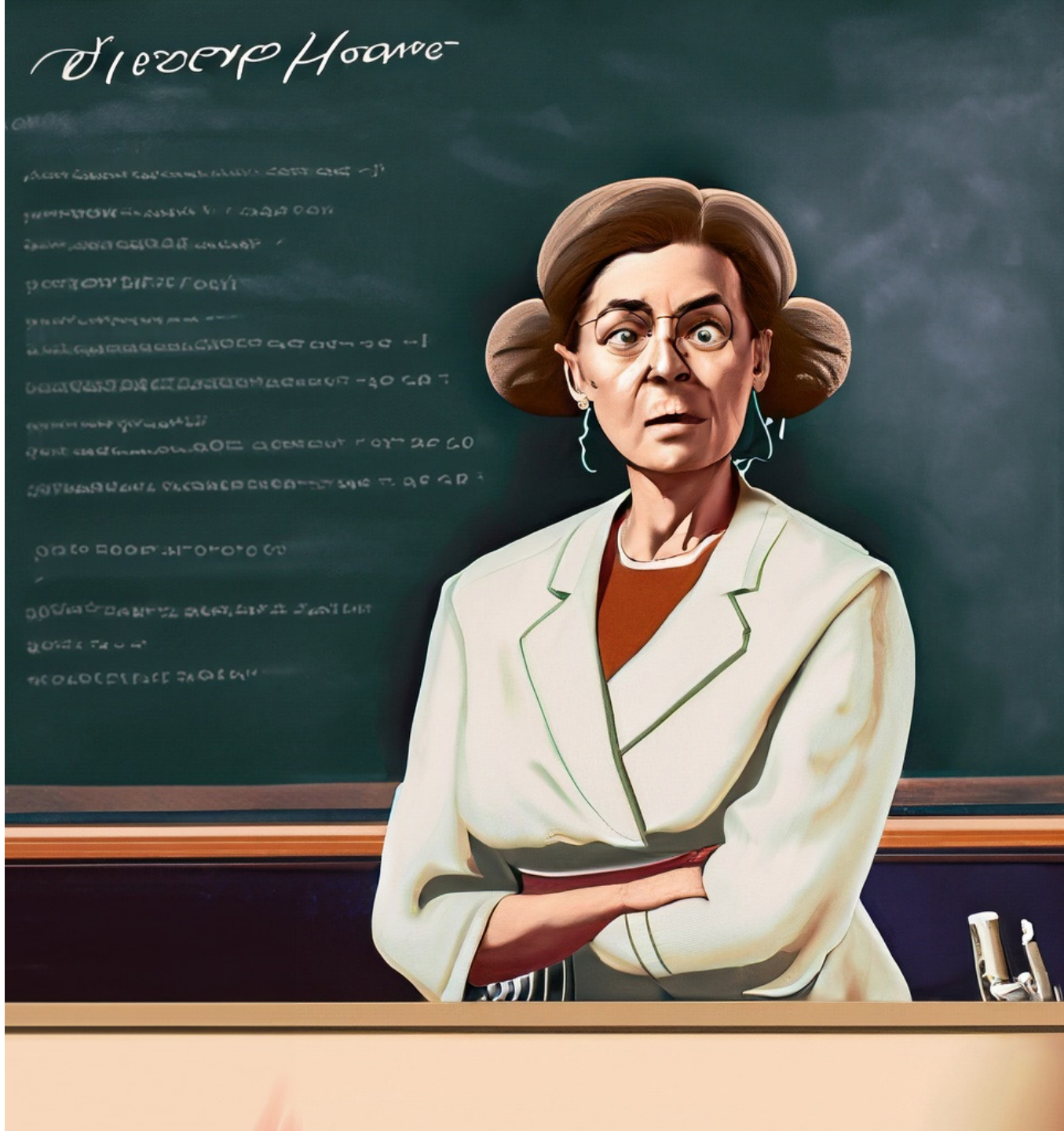


Virtual Dynamic Shared Object (VDSO)

Intel: We can do better. Thus, sysenter/sysexit

AMD: We can do better. Thus, syscall/sysret

In 32-bit mode, ‘int 0x80’ works, sometimes ‘sysenter’ works, and sometimes ‘syscall’ works.

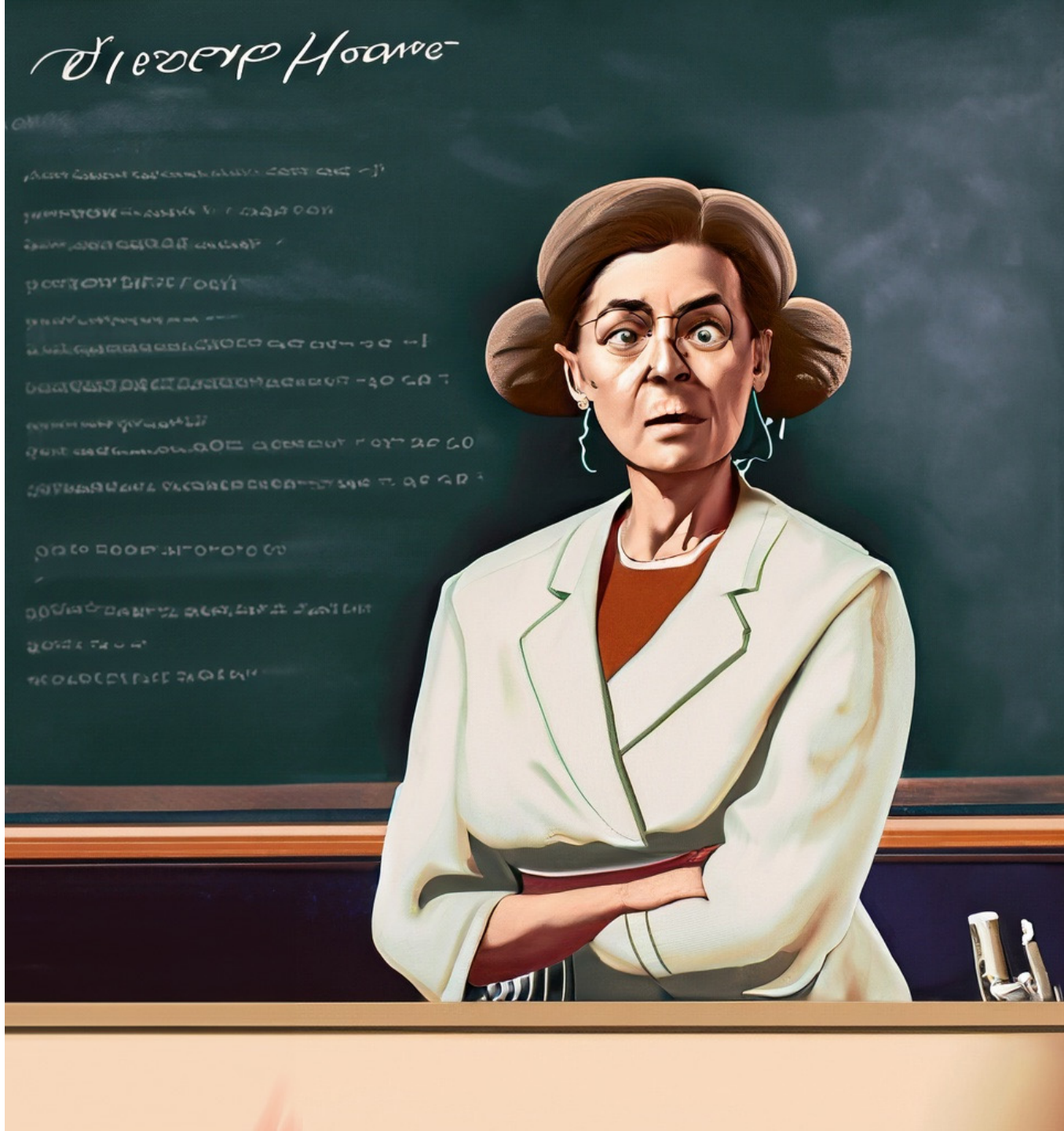


Virtual Dynamic Shared Object (VDSO)

Linux devs make VDSO

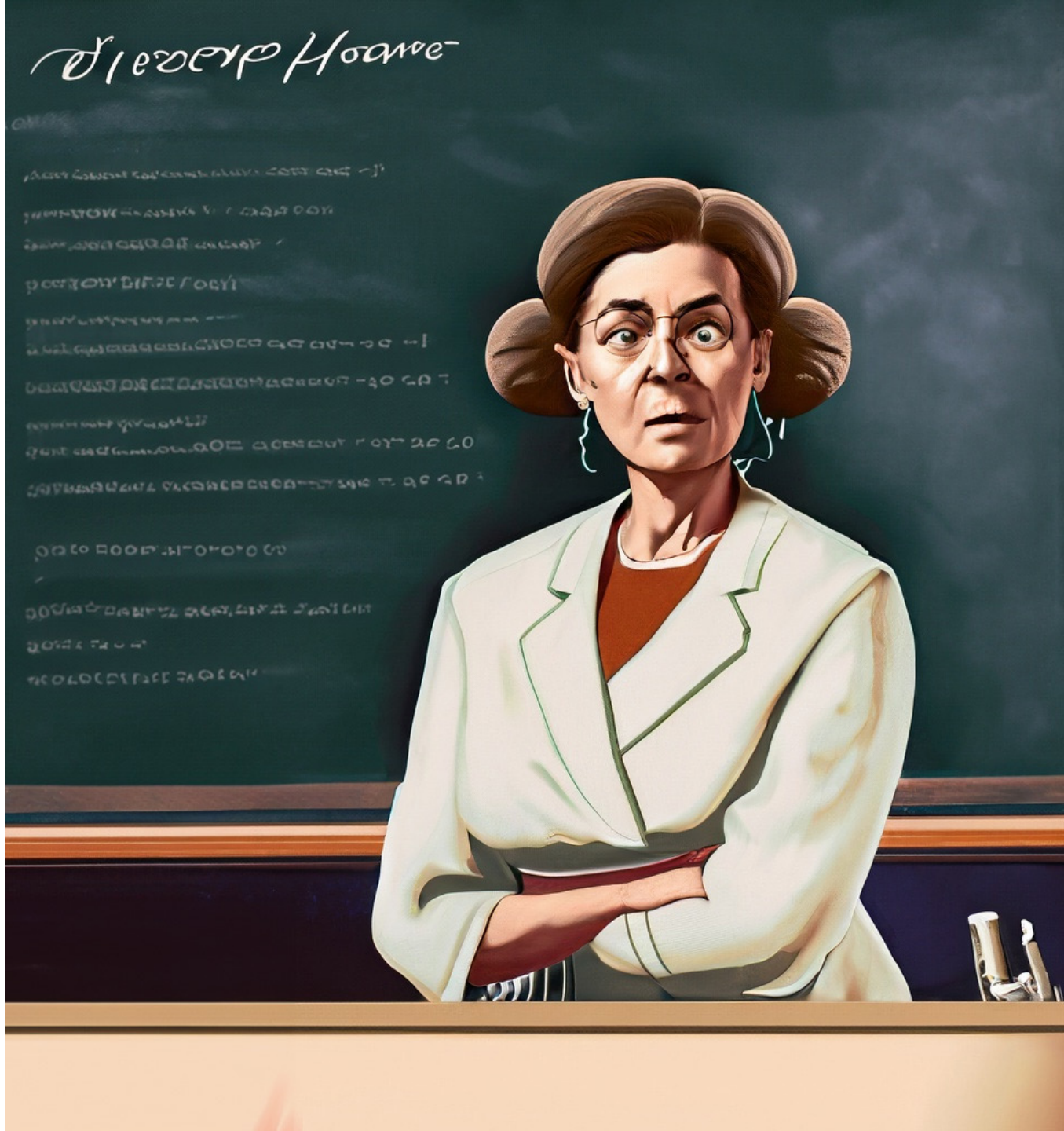
- Dynamic Library
- Linked to all executables
- Provides __kernel_vsyscall function that chooses among the variations
- Used by glibc

Again, VDSO is found in the ELF Auxiliary Vectors



Virtual Dynamic Shared Object (VDSO)

A kernel developer realized it would be useful to put other functions in VDSO (e.g. __vdso_clock_gettime)



vdso(7) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | NOTES | ARCHITECTURE-SPECIFIC NOTES | SEE ALSO | COLOPHON

Search online pages

VDSO(7) **Linux Programmer's Manual** **VDSO(7)**

NAME top

vdso - overview of the virtual ELF dynamic shared object

SYNOPSIS top

```
#include <sys/auxv.h>

void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);
```

DESCRIPTION top

The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take

ed by the vDSO.

ted since Linux 3.15)
ted since Linux 3.15)
ted since Linux 3.15)

ed by the vDSO. All of
the "__vdso_" prefix,
the names below.

ed by the vDSO.

```
__vdso_clock_gettime   LINUX_2.6
__vdso_getcpu          LINUX_2.6
__vdso_gettimeofday    LINUX_2.6
```

i386 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__kernel_sigreturn	LINUX_2.5
__kernel_rt_sigreturn	LINUX_2.5
__kernel_vsyscall	LINUX_2.5
__vdso_clock_gettime	LINUX_2.6 (exported since Linux 3.15)
__vdso_gettimeofday	LINUX_2.6 (exported since Linux 3.15)
__vdso_time	LINUX_2.6 (exported since Linux 3.15)

x86-64 functions

The table below lists the symbols exported by the vDSO. All of these symbols are also available without the "__vdso_" prefix, but you should ignore those and stick to the names below.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6
__vdso_time	LINUX_2.6

i386 functions

The table below lists the symbols exported by the vDSO.

symbol	version	
__kernel_sigreturn	LINUX_2.5	
__kernel_rt_sigreturn	LINUX_2.5	
__kernel_vsyscall	LINUX_2.5	
__vdso_clock_gettime	LINUX_2.6	(exported since Linux 3.15)
__vdso_gettimeofday	LINUX_2.6	(exported since Linux 3.15)
__vdso_time	LINUX_2.6	(exported since Linux 3.15)

x86-64 functions

The table below lists the symbols exported by the vDSO. All of these symbols are also available without the "__vdso_" prefix, but you should ignore those and stick to the names below.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6
__vdso_time	LINUX_2.6

__vdso_time LINUX_2.6 (exported since Linux 3.15)

x86-64 functions

The table below lists the symbols exported by the vDSO. All of these symbols are also available without the "__vdso_" prefix, but you should ignore those and stick to the names below.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6
__vdso_time	LINUX_2.6

x86/x32 functions

The table below lists the symbols exported by the vDSO.

symbol	version
__vdso_clock_gettime	LINUX_2.6
__vdso_getcpu	LINUX_2.6
__vdso_gettimeofday	LINUX_2.6

x86_64 syscalls

```
stdout          equ 1                   ; Constants
SYS_WRITE       equ 1
SYS_EXIT        equ 60
    .data                               ; Data
string  db 10,"Hello, world!",10
    .code
_start:                                 ; Assembly
    mov edx, sizeof string
    mov rsi, offset string
    mov edi, stdout
    mov eax, SYS_WRITE
    syscall
    mov eax, SYS_EXIT
    syscall
    end _start
```

x86_64 syscalls

```
stdout          equ 1           ; Constants
SYS_WRITE       equ 1
SYS_EXIT        equ 60
    .data                       ; Data
string  db 10,"Hello, world!",10
    .code
_start:                         ; Assembly
    mov edx, sizeof string
    mov rsi, offset string
    mov edi, stdout
    mov eax, SYS_WRITE
    syscall
    mov eax, SYS_EXIT
    syscall
    end _start
```

x86_64 syscalls

```
stdout          equ 1                   ; Constants
SYS_WRITE       equ 1
SYS_EXIT        equ 60
    .data                               ; Data
string  db 10,"Hello, world!",10
    .code
_start:                                 ; Assembly
    mov edx, sizeof string
    mov rsi, offset string
    mov edi, stdout
    mov eax, SYS_WRITE
    syscall
    mov eax, SYS_EXIT
    syscall
    end _start
```

x86_64 syscalls

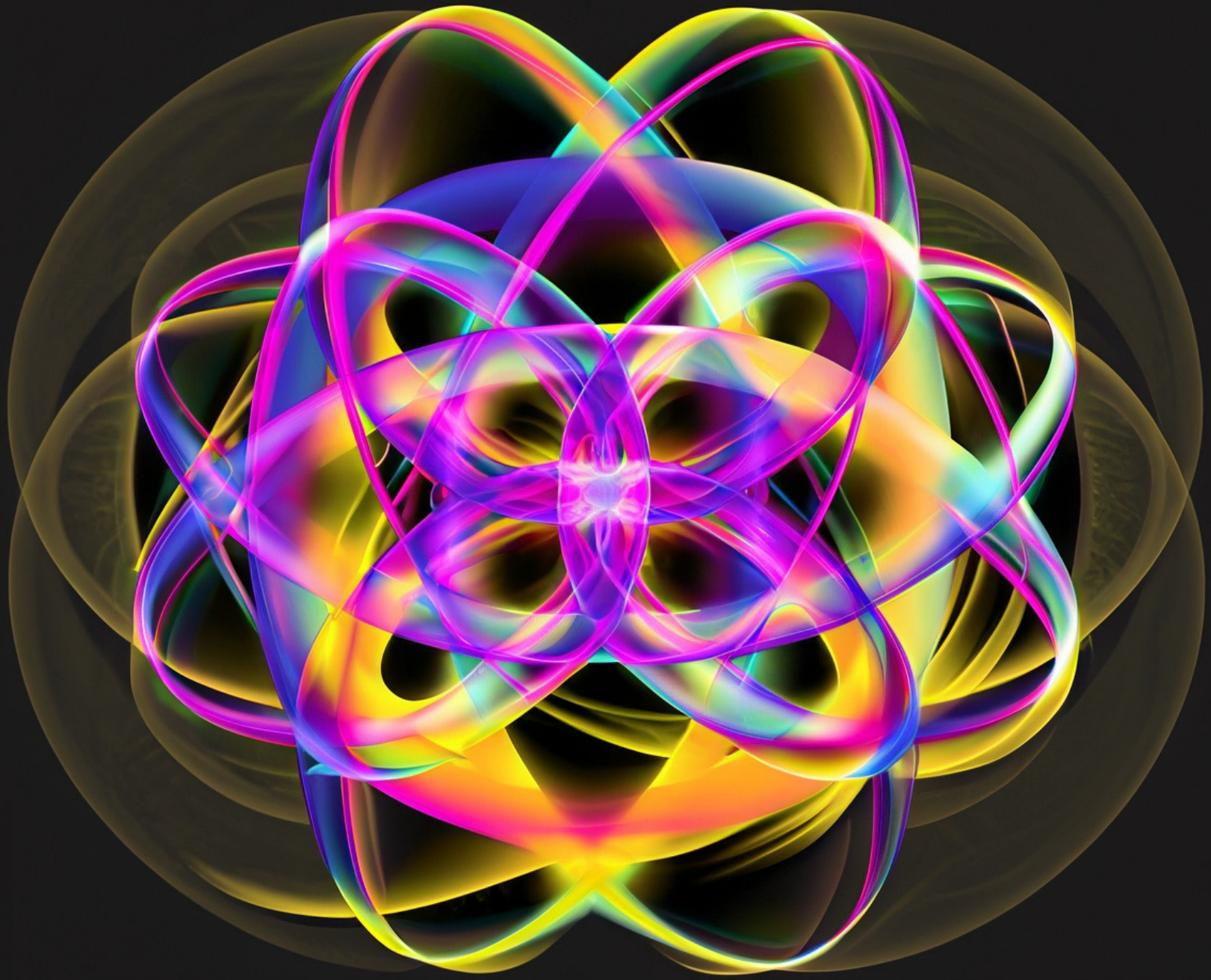
```
stdout          equ 1                   ; Constants
SYS_WRITE       equ 1
SYS_EXIT        equ 60
    .data                               ; Data
string  db 10,"Hello, world!",10
    .code
_start:                                 ; Assembly
    mov edx, sizeof string
    mov rsi, offset string
    mov edi, stdout
    mov eax, SYS_WRITE
    syscall
    mov eax, SYS_EXIT
    syscall
    end _start
```

x86_64 syscalls

```
stdout          equ 1                   ; Constants
SYS_WRITE       equ 1
SYS_EXIT        equ 60
    .data                               ; Data
string  db 10,"Hello, world!",10
    .code
_start:                                 ; Assembly
    mov edx, sizeof string
    mov rsi, offset string
    mov edi, stdout
    mov eax, SYS_WRITE
    syscall
    mov eax, SYS_EXIT
    syscall
    end _start
```

Syscalls in Linux

- Prefer using glibc
- Failing that, use `syscall` in glibc
- Failing that, use nolibc



[D]eveloping lock-free code may be cool but trying to find bugs in it is most definitely not.

—Fedor Pikus

[D]eveloping lock-free code may be cool but trying to find bugs in it is most definitely not.

—Fedor Pikus

A look at Atomics

```
void set( int & value, bool & is_ready ) {
    value = 42;
    is_ready = true;
}
```

A look at Atomics

```
void set( int & value, bool & is_ready ) {
    value = 42;
    is_ready = true;
}
```

```
set(int&, bool&):
        mov     dword ptr [rdi], 42
        mov     byte ptr [rsi], 1
        ret
```

A look at Atomics

```
void set( int & value, bool & is_ready ) {
    value = 42;
    is_ready = true;
}
```

```
void set_atomic( int & value, std::atomic<bool> & is_ready ) {
    value = 42;
    is_ready.store(true, std::memory_order_release);
}
```

```
set(int&, bool&):
        mov     dword ptr [rdi], 42
        mov     byte ptr [rsi], 1
        ret
```

A look at Atomics

```
void set( int & value, bool & is_ready ) {
    value = 42;
    is_ready = true;
}

void set_atomic( int & value, std::atomic<bool> & is_ready ) {
    value = 42;
    is_ready.store(true, std::memory_order_release);
}
```

```
set(int&, bool&):
        mov     dword ptr [rdi], 42
        mov     byte ptr [rsi], 1
        ret

set_atomic(int&, std::atomic<bool>&)
        mov     dword ptr [rdi], 42
        mov     byte ptr [rsi], 1
        ret
```

A look at Atomics

```
int get( int & value, bool & is_ready ) {
    if( is_ready ) {
        return value;
    } else {
        return 0;
    }
}
```

A look at Atomics

```
int get( int & value, bool & is_ready ) {
    if( is_ready ) {
        return value;
    } else {
        return 0;
    }
}
```

```
get(int&, bool&):
        xor     eax, eax
        cmp     byte ptr [rsi], 0
        je      .LBB1_2
        mov     eax, dword ptr [rdi]
.LBB1_2:
        ret
```

A look at Atomics

```
int get( int & value, bool & is_ready ) {
    if( is_ready ) {
        return value;
    } else {
        return 0;
    }
}
int get_atomic( int & value, std::atomic<bool> & is_ready ) {
    if( is_ready.load(std::memory_order_acquire) ) {
        return value;
    } else {
        return 0;
    }
}
```

```
get(int&, bool&):
        xor     eax, eax
        cmp     byte ptr [rsi], 0
        je      .LBB1_2
        mov     eax, dword ptr [rdi]
.LBB1_2:
        ret
```

A look at Atomics

```
int get( int & value, bool & is_ready ) {
    if( is_ready ) {
        return value;
    } else {
        return 0;
    }
}
int get_atomic( int & value, std::atomic<bool> & is_ready ) {
    if( is_ready.load(std::memory_order_acquire) ) {
        return value;
    } else {
        return 0;
    }
}
```

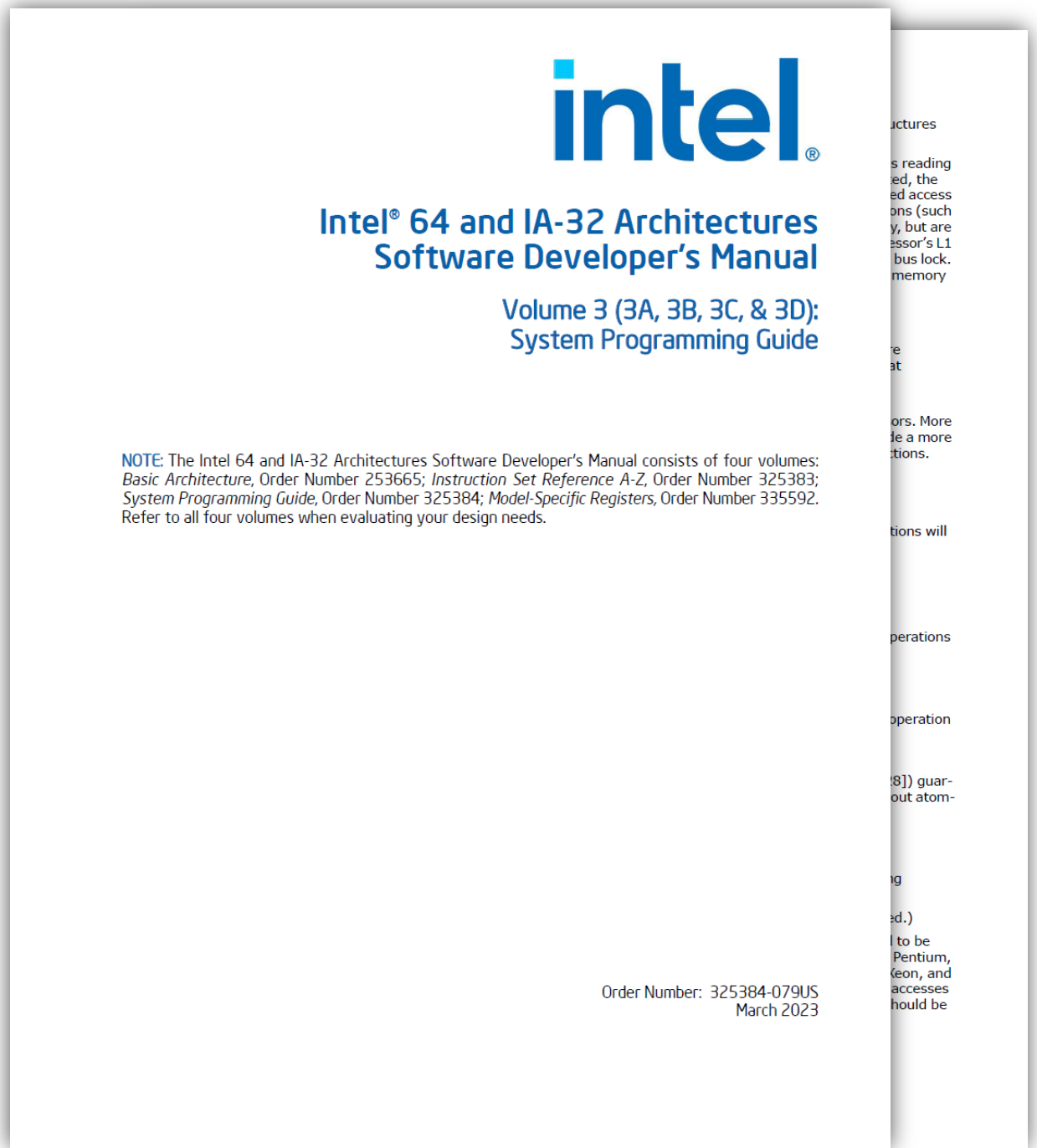
```
get(int&, bool&):
        xor     eax, eax
        cmp     byte ptr [rsi], 0
        je      .LBB1_2
        mov     eax, dword ptr [rdi]
.LBB1_2:
        ret
get_atomic(int&, std::atomic<bool>&):
        movzx   ecx, byte ptr [rsi]
        xor     eax, eax
        test    cl, 1
        je      .LBB3_2
        mov     eax, dword ptr [rdi]
.LBB3_2:
        ret
```

C’mon compiler

‘mov byte ptr [rsi], 1’ is atomic
‘cmp byte ptr [rsi], 0’ is not atomic
‘movzx ecx, byte ptr [rsi]’ is atomic

- None of the “atomic” things have a LOCK annotation
- It’s strange that atomic code sometimes look the same as non-atomic code
- What’s going on here?

Maybe you should read the manual



Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325384-079US
March 2023

9.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte.
- Reading or writing a word aligned on a 16-bit boundary.
- Reading or writing a doubleword aligned on a 32-bit boundary.

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary.
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus.

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line.

Processors that enumerate support for Intel® AVX (by setting the feature flag CPUID.01H:ECX.AVX[bit 28]) guarantee that the 16-byte memory operations performed by the following instructions will always be carried out atom-

[D]eveloping lock-free code may be cool but trying to find bugs in it is most definitely not.

—Fedor Pikus

[D]eveloping lock-free code may be cool but trying to find bugs in it is most definitely not.

—Fedor Pikus

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

CISC and RISC

Complex Instruction Set Computer (CISC) – Intel

Reduced Instruction Set Computer (RISC) – ARM (newer Macs, Raspberry Pi, etc.)

```
void aincr(std::atomic<int> & value){
  value++;
}
```

```
; Intel x86
lock add DWORD PTR [rdi], 1
ret

; AArch64
.LBB0_1:
        ldaxr   w8, [x0]
        add     w8, w8, #1
        stlxr   w9, w8, [x0]
        cbnz    w9, .LBB0_1
        ret
```

ARM v8.1-a

ARM v8.1-a

```
mov     w8, #1
ldaddal w8, w8, [x0]
ret
```

ARM v8.1-a

```
mov     w8, #1
ldaddal w8, w8, [x0]
ret
```

Intel and CISC

- On modern Intel CPUs there are >556 registers (Segmented Memory, SSE/AVX, debug, etc.)
- >2000 instructions

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

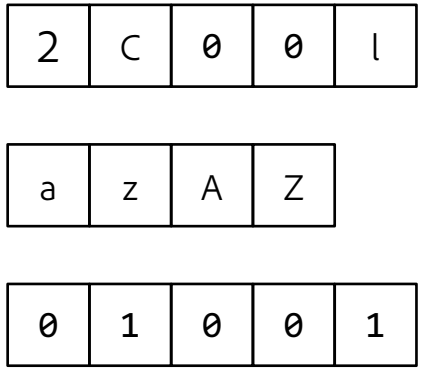
PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask



Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

Craziest Intel Instruction

PCMPESTRM xmm, xmm, imm8 - Packed Compare Explicit Length Strings, Return Mask

RAX = length of first string

RDX = length of second string

Imm8 is the comparison:

- Bits 0 and 1 specify data type: uint8, uint16, int8, int16
- Bits 2 and 3 specify test: subset, ranges, match, substring search
- Bit 4: compliment result
- Bit 5: compliment only valid bits
- Bit 6: bit mask or a byte mask

More cool things to explore

- Models for CPU-level concurrency (x86-TSO)
- Segmented Memory
- io_uring
- Return-oriented programming
- Instruction pipelining
- Microbenchmarking

Assembly Resources

- Creel's Assembly Tutorials https://www.youtube.com/@WhatsACreel
- Intel's Software Developer Manuals https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- Defuse online assembler https://defuse.ca/online-x86-assembler.htm
- Agner Fog's optimization page https://www.agner.org/optimize/
- Atomics in AArch64 https://cpufun.substack.com/p/atomics-in-aarch64
- Register count for Intel https://blog.yossarian.net/2020/11/30/How-many-registers-does-an-x86-64-cpu-have

Linux Resources

- VDSO manual https://man7.org/linux/man-pages/man7/vdso.7.html
- Article on glibc/system call controversies https://lwn.net/Articles/655028/
- Assembly that launches main https://github.com/torvalds/linux/blob/16a8829130ca22666ac6236178a6233208d425c3/tools/include/nolibc/arch-x86_64.h#L193


Adobe
Artwork by Dan Zucco

Bonus: calling main

```
pop rdi                    ; argc   (first arg, rdi)
mov rsi, rsp               ; argv[] (second arg, rsi)
lea rdx, [rsi+rdi*8+1]  ; then a NULL then envp (third arg, rdx)
mov environ, rdx           ; save environ
xor ebp, ebp               ; zero the stack frame
mov rax, rdx               ; search for auxv (follows NULL after last env)
loop:
add rax, 8                 ; search for auxv using rax, it follows the
cmp rbp, [rax-8]           ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax             ; save it into _auxv
and rsp, -16               ; x86 ABI: esp must be 16-byte aligned before call
call main                  ; main() returns status code, we'll exit with it.
mov edi, eax               ; retrieve exit code (32 bit)
mov eax, 60                ; NR_exit == 60
syscall                    ; really exit
```

```
pop rdi                    ; argc   (first arg, rdi)
mov rsi, rsp               ; argv[] (second arg, rsi)
lea rdx, [rsi+rdi*8+1]     ; then a NULL then envp (third arg, rdx)
mov environ, rdx           ; save environ
xor ebp, ebp               ; zero the stack frame
mov rax, rdx               ; search for auxv (follows NULL after last env)
loop:
add rax, 8                 ; search for auxv using rax, it follows the
cmp rbp, [rax-8]           ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax             ; save it into _auxv
and rsp, -16               ; x86 ABI: esp must be 16-byte aligned before call
call main                  ; main() returns status code, we'll exit with it.
mov edi, eax               ; retrieve exit code (32 bit)
mov eax, 60                ; NR_exit == 60
syscall                    ; really exit
```

```
pop rdi                    ; argc   (first arg, rdi)
mov rsi, rsp               ; argv[] (second arg, rsi)
lea rdx, [rsi+rdi*8+1]  ; then a NULL then envp (third arg, rdx)
mov environ, rdx           ; save environ
xor ebp, ebp               ; zero the stack frame
mov rax, rdx               ; search for auxv (follows NULL after last env)
loop:
add rax, 8                 ; search for auxv using rax, it follows the
cmp rbp, [rax-8]           ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax             ; save it into _auxv
and rsp, -16               ; x86 ABI: esp must be 16-byte aligned before call
call main                  ; main() returns status code, we'll exit with it.
mov edi, eax               ; retrieve exit code (32 bit)
mov eax, 60                ; NR_exit == 60
syscall                    ; really exit
```

```
loop:
add rax, 8                  ; search for auxv using rax, it follows the
cmp rbp, [rax-8]            ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax              ; save it into _auxv
and rsp, -16                ; x86 ABI: esp must be 16-byte aligned before call
call main                   ; main() returns status code, we'll exit with it.
mov edi, eax                ; retrieve exit code (32 bit)
mov eax, 60                 ; NR_exit == 60
syscall                     ; really exit
hlt                         ; ensure it does not return
```

```
loop:
add rax, 8                ; search for auxv using rax, it follows the
cmp rbp, [rax-8]          ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax            ; save it into _auxv
and rsp, -16              ; x86 ABI: esp must be 16-byte aligned before call
call main                 ; main() returns status code, we'll exit with it.
mov edi, eax              ; retrieve exit code (32 bit)
mov eax, 60               ; NR_exit == 60
syscall                   ; really exit
hlt                       ; ensure it does not return
```

```
loop:
add rax, 8                ; search for auxv using rax, it follows the
cmp rbp, [rax-8]          ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax            ; save it into _auxv
and rsp, -16              ; x86 ABI: esp must be 16-byte aligned before call
call main                 ; main() returns status code, we'll exit with it.
mov edi, eax              ; retrieve exit code (32 bit)
mov eax, 60               ; NR_exit == 60
syscall                   ; really exit
hlt                       ; ensure it does not return
```

```
loop:
add rax, 8                ; search for auxv using rax, it follows the
cmp rbp, [rax-8]          ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax            ; save it into _auxv
and rsp, -16              ; x86 ABI: esp must be 16-byte aligned before call
call main                 ; main() returns status code, we'll exit with it.
mov edi, eax              ; retrieve exit code (32 bit)
mov eax, 60               ; NR_exit == 60
syscall                   ; really exit
hlt                       ; ensure it does not return
```

```
loop:
add rax, 8                ; search for auxv using rax, it follows the
cmp rbp, [rax-8]          ; ... NULL after last env (rbp is zero here)
jnz loop
mov _auxv, rax            ; save it into _auxv
and rsp, -16              ; x86 ABI: esp must be 16-byte aligned before call
call main                 ; main() returns status code, we'll exit with it.
mov edi, eax              ; retrieve exit code (32 bit)
mov eax, 60               ; NR_exit == 60
syscall                   ; really exit
hlt                       ; ensure it does not return
```

Questions/Comments

About the artist

Dan Zucco

London-based 3D art and motion director Dan Zucco creates repeating 2D patterns and brings them to life as 3D animated loops. Inspired by architecture, music, modern art, and generative design, he often starts in Adobe Illustrator and builds his animations using Adobe After Effects and Cinema 4D. Zucco's objective for this piece was to create a geometric design that felt like it could have an infinite number of arrangements.

Made with

Adobe Illustrator Adobe After Effects


Adobe
Artwork by Dan Zucco