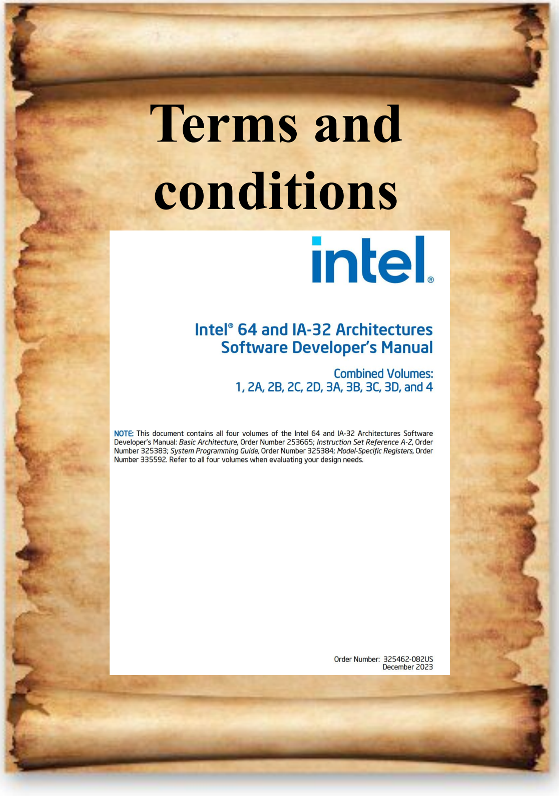
Recitation 6

Computer Architecture (section 1)

What is an Instruction Set Architecture (ISA)?

- A contract between software and hardware.



Terms and conditions

intel®

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

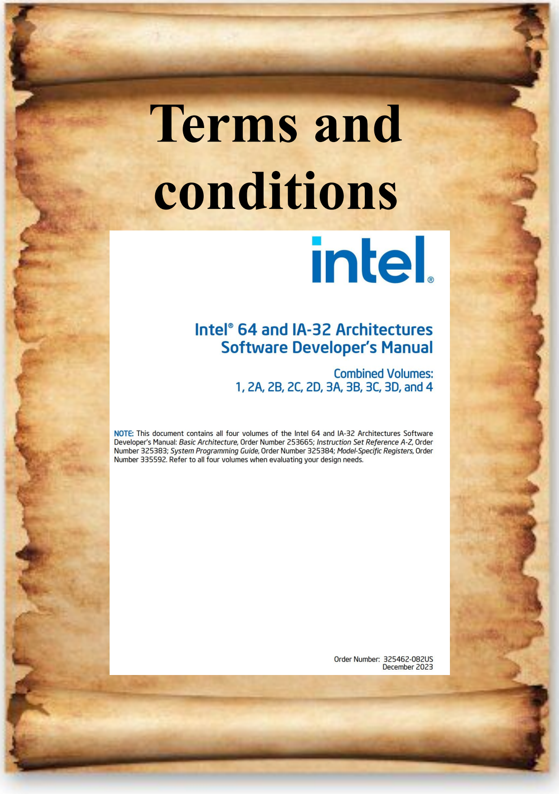
Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *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: 325462-082US
December 2023

What is an Instruction Set Architecture (ISA)?

- A contract between software and hardware.
- Hardware provides a specification, software can use this specification to do computation.



Terms and conditions

intel®

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

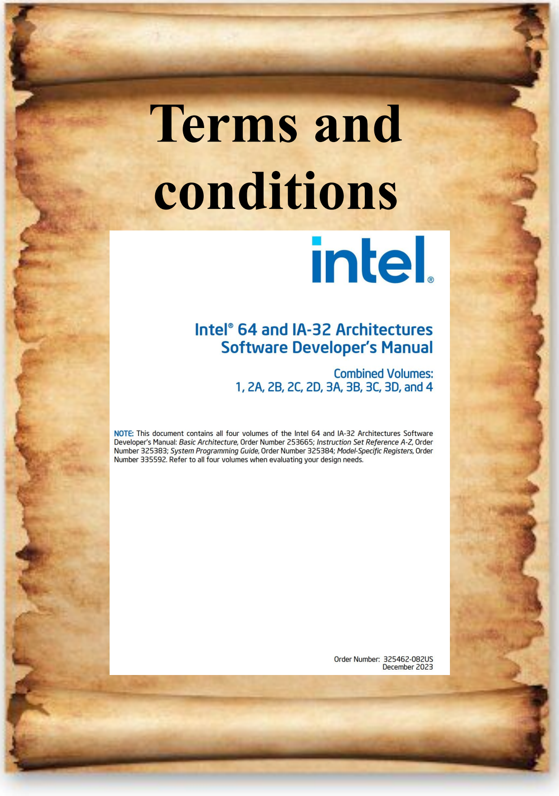
Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *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: 325462-082US
December 2023

What is an Instruction Set Architecture (ISA)?

- A contract between software and hardware.
- Hardware provides a specification, software can use this specification to do computation.
- The ISA specifies ***all*** the hardware understands.
  - This is the machine code.



Terms and conditions

intel

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

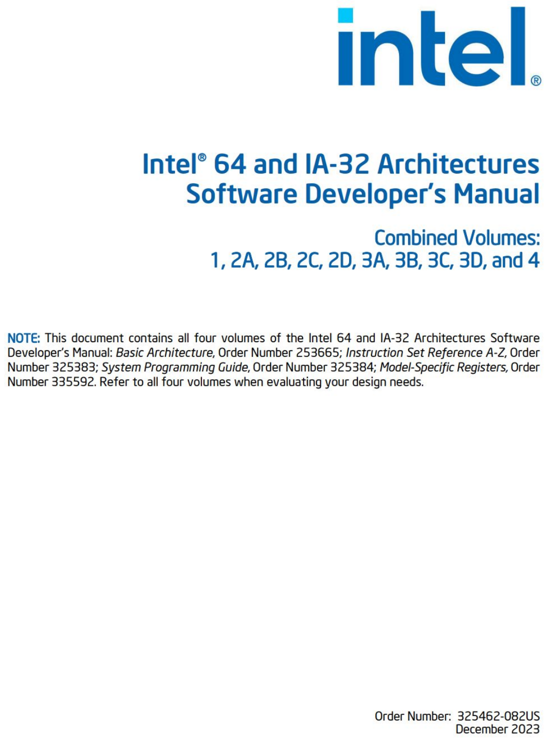
Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *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: 325462-082US
December 2023

The x86 ISA

- Initially developed by Intel.
  - Today’s market drivers are Intel and AMD.
- Now competing with ARM.
  - RISC-V in the future?



intel®

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

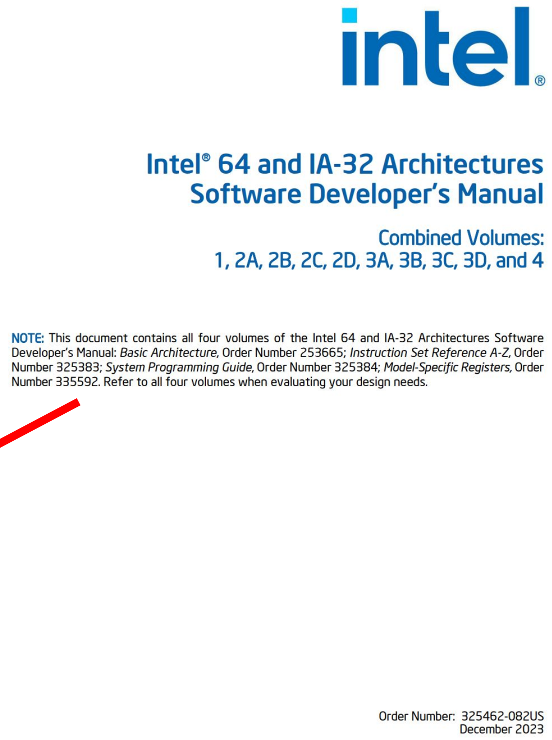
Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer’s Manual: *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: 325462-082US
December 2023

The x86 ISA

- Initially developed by Intel.
  - Today’s market drivers are Intel and AMD.
- Now competing with ARM.
  - RISC-V in the future?



intel®

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

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

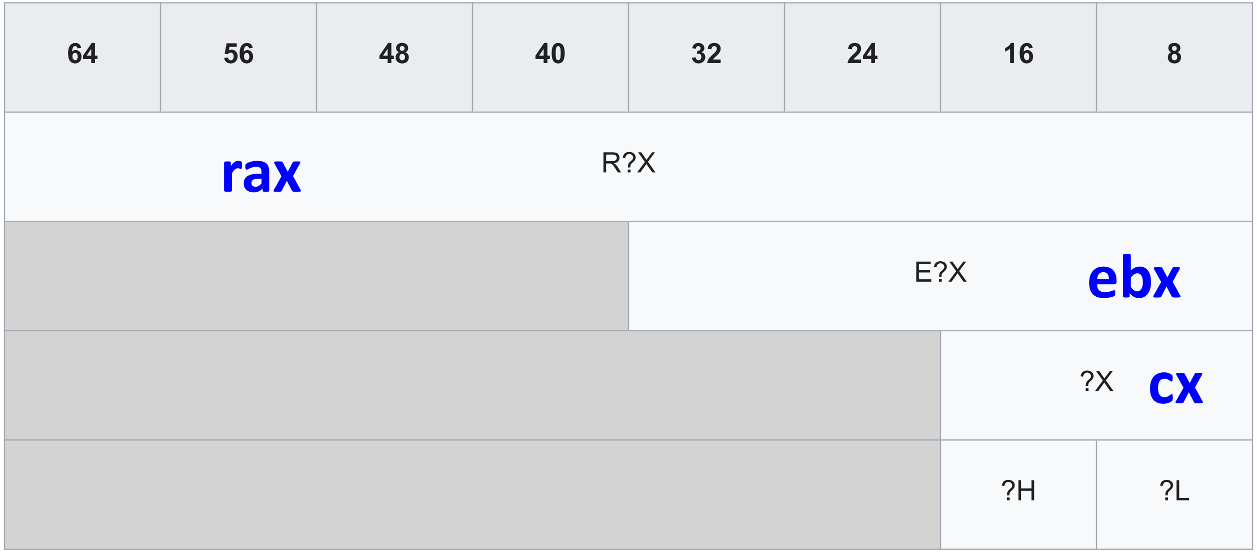
NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer’s Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number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

Order Number: 325462-082US
December 2023

5082 pages and growing!

The x86-64 ISA - General Purpose Registers

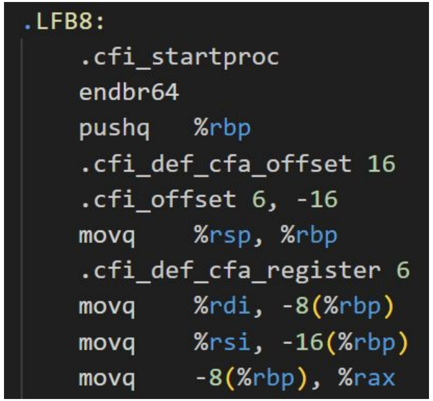
(A, B, C and D)



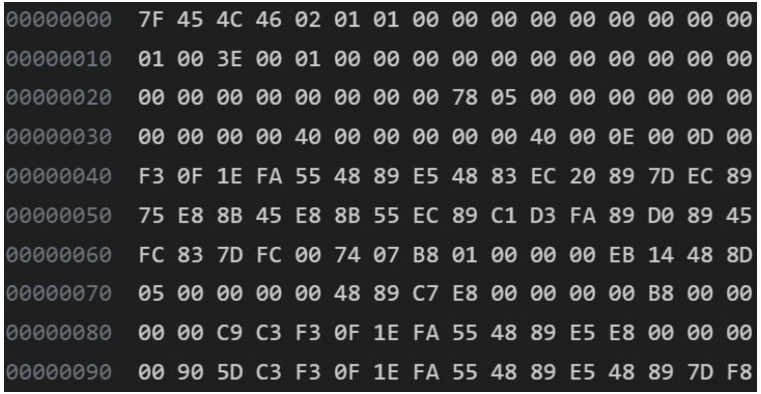
64	56	48	40	32	24	16	8
	rax			R?X			
				E?X			ebx
						?X	cx
						?H	?L

Assembly

- A low-level programming language that corresponds (almost 1:1) with machine code.



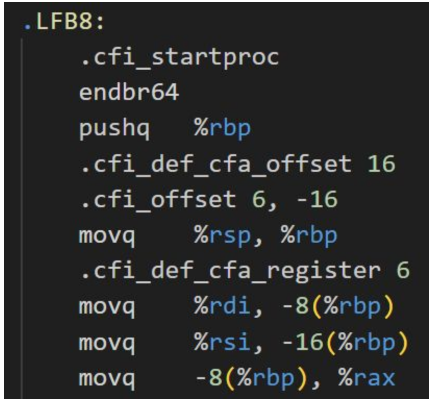
```
.LFB8:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movq    -8(%rbp), %rax
```



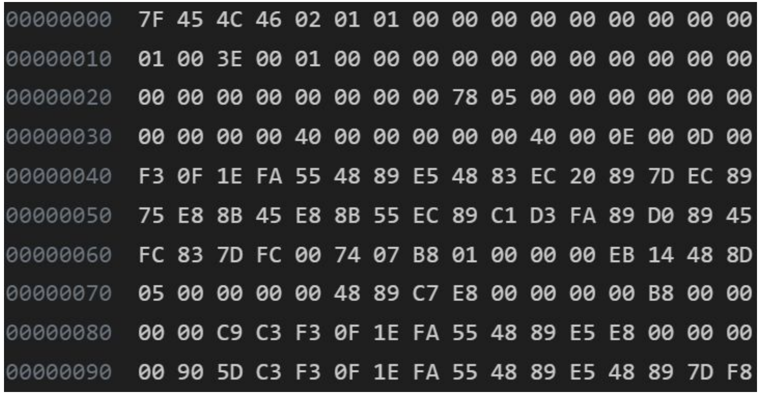
```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010  01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00
00000020  00 00 00 00 00 00 00 00 78 05 00 00 00 00 00 00
00000030  00 00 00 00 40 00 00 00 00 00 40 00 0E 00 0D 00
00000040  F3 0F 1E FA 55 48 89 E5 48 83 EC 20 89 7D EC 89
00000050  75 E8 8B 45 E8 8B 55 EC 89 C1 D3 FA 89 D0 89 45
00000060  FC 83 7D FC 00 74 07 B8 01 00 00 00 EB 14 48 8D
00000070  05 00 00 00 00 48 89 C7 E8 00 00 00 00 B8 00 00
00000080  00 00 C9 C3 F3 0F 1E FA 55 48 89 E5 E8 00 00 00
00000090  00 90 5D C3 F3 0F 1E FA 55 48 89 E5 48 89 7D F8
```

Assembly

- A low-level programming language that corresponds (almost 1:1) with machine code.
- The assembler converts assembly to machine code.



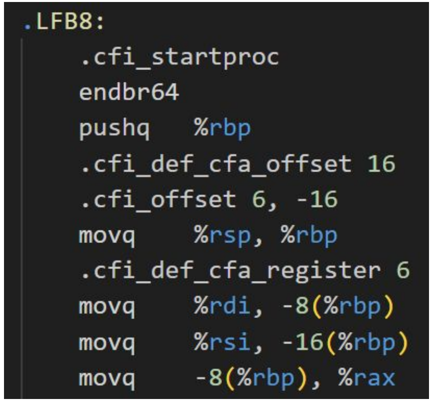
```
.LFB8:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -8(%rbp)
	movq	%rsi, -16(%rbp)
	movq	-8(%rbp), %rax
```



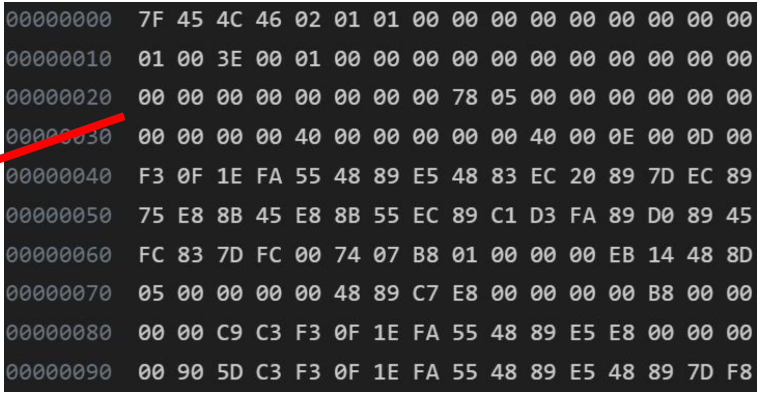
```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010  01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00
00000020  00 00 00 00 00 00 00 00 78 05 00 00 00 00 00 00
00000030  00 00 00 00 40 00 00 00 00 00 40 00 0E 00 0D 00
00000040  F3 0F 1E FA 55 48 89 E5 48 83 EC 20 89 7D EC 89
00000050  75 E8 8B 45 E8 8B 55 EC 89 C1 D3 FA 89 D0 89 45
00000060  FC 83 7D FC 00 74 07 B8 01 00 00 00 EB 14 48 8D
00000070  05 00 00 00 00 48 89 C7 E8 00 00 00 00 B8 00 00
00000080  00 00 C9 C3 F3 0F 1E FA 55 48 89 E5 E8 00 00 00
00000090  00 90 5D C3 F3 0F 1E FA 55 48 89 E5 48 89 7D F8
```

Assembly

- A low-level programming language that corresponds (almost 1:1) with machine code.
- The assembler converts assembly to machine code.



```
.LFB8:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -8(%rbp)
	movq	%rsi, -16(%rbp)
	movq	-8(%rbp), %rax
```



```
00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010  01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00
00000020  00 00 00 00 00 00 00 00 78 05 00 00 00 00 00 00
00000030  00 00 00 00 40 00 00 00 00 00 40 00 0E 00 0D 00
00000040  F3 0F 1E FA 55 48 89 E5 48 83 EC 20 89 7D EC 89
00000050  75 E8 8B 45 E8 8B 55 EC 89 C1 D3 FA 89 D0 89 45
00000060  FC 83 7D FC 00 74 07 B8 01 00 00 00 EB 14 48 8D
00000070  05 00 00 00 00 48 89 C7 E8 00 00 00 00 B8 00 00
00000080  00 00 C9 C3 F3 0F 1E FA 55 48 89 E5 E8 00 00 00
00000090  00 90 5D C3 F3 0F 1E FA 55 48 89 E5 48 89 7D F8
```

Can’t have a human program in this

x86 Assembly - Syntax

```
movl $3, %eax
```

x86 Assembly - Syntax

```
movl $3, %eax
```

Instruction

x86 Assembly - Syntax

Source Operand

```
movl $3, %eax
```

Instruction

x86 Assembly - Syntax

Source Operand

```
movl $3, %eax
```

Instruction **Dest operand**

x86 Assembly - Syntax

Source Operand

```
movl $3, %eax
```

Instruction **Dest operand**

x86 permits 0-3 operands

With 2 operands, the order is source, destination

x86 Assembly - Instruction Suffixes

4 bytes `movl $4, %eax`

x86 Assembly - Instruction Suffixes

4 bytes	`movl $4, %eax`
1 bytes	`movb $1, %ah`

x86 Assembly - Instruction Suffixes

4 bytes	`movl $4, %eax`
1 bytes	`movb $1, %ah`
2 bytes	`movw $2, %ax`

x86 Assembly - Instruction Suffixes

4 bytes	`movl $4, %eax`
1 bytes	`movb $1, %ah`
2 bytes	`movw $2, %ax`
8 bytes	`movq $8, %rax`

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
movl %ebx, %eax
```

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
movl $3, %eax
```

x86 Assembly - Addressing Modes

- Register
- Immediate
- **Absolute (Direct)**
- Indirect
- Indirect with offset
- Indexed
- Indexed with offset
- Scaled indexed
- Scaled indexed with offset

```
movl $5, 0x123456
```

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
movl $2, (%eax)
```

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
movl $2, -8(%ebp)
```

Address = %ebp + (-8)

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
leal (%ebx,%ecx), %eax
```

Address = %ebx + %ecx

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
leal -8(%ebx,%ecx), %eax
```

Address = %ebx + %ecx + (-8)

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
leal (,%ecx,4), %eax
```

Address = %ecx*4

x86 Assembly - Addressing Modes

Register

Immediate

Absolute

Indirect

Indirect with offset

Indexed

Indexed with offset

Scaled indexed

Scaled indexed with offset

```
leal -8(,%ecx,4), %eax
```

Address = %ecx*4 + (-8)

x86 Assembly - Common Instructions

- `movl %eax,%ebx`
  - Move source to destination

x86 Assembly - Common Instructions

- `movl %eax,%ebx`
  - Move source to destination
- `leal -8(%eax), %ebx`
  - Load effective address

x86 Assembly - Common Instructions

- `movl %eax,%ebx`
  - Move source to destination
- `leal -8(%eax), %ebx`
  - Load effective address
- `addl/subl %eax,%ebx`
  - Add/sub source to/from destination

x86 Assembly - Common Instructions

- `movl %eax,%ebx`
  - Move source to destination
- `leal -8(%eax), %ebx`
  - Load effective address
- `addl/subl %eax,%ebx`
  - Add/sub source to/from destination
- `imull %eax,%ebx`
  - Multiply source and destination

x86 Assembly - Common Instructions

- `movl %eax,%ebx`
  - Move source to destination
- `leal -8(%eax), %ebx`
  - Load effective address
- `addl/subl %eax,%ebx`
  - Add/sub source to/from destination
- `imull %eax,%ebx`
  - Multiply source and destination
- `incl %eax`
  - Increment by 1

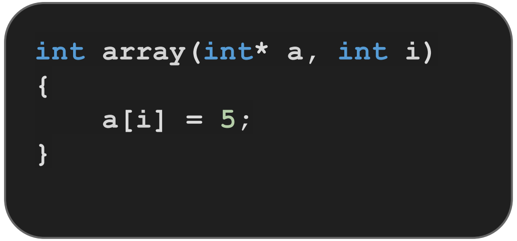
x86 Assembly - Common Instructions

- `movl %eax,%ebx`
    - Move source to destination
- `leal -8(%eax), %ebx`
    - Load effective address
- `addl/subl %eax,%ebx`
    - Add/sub source to/from destination
- `imull %eax,%ebx`
    - Multiply source and destination
- `incl %eax`
    - Increment by 1
- `sal %al,%ebx`
    - Shift destination bits left by source bits

x86 Assembly - Common Instructions

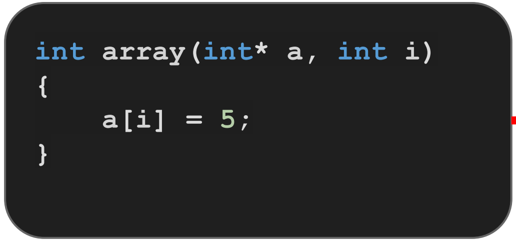
- `movl %eax,%ebx`
  - Move source to destination
- `leal -8(%eax), %ebx`
  - Load effective address
- `addl/subl %eax,%ebx`
  - Add/sub source to/from destination
- `imull %eax,%ebx`
  - Multiply source and destination
- `incl %eax`
  - Increment by 1
- `sal %al,%ebx`
  - Shift destination bits left by source bits
- `sar %al,%ebx`
  - Shift destination bits right by source bits (keeps sign) vs `shr`

x86 Assembly - Indexing an array

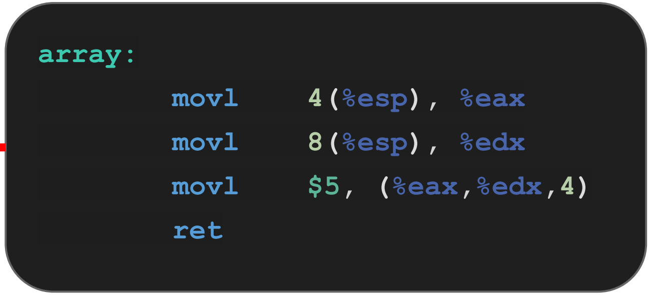


```c
int array(int* a, int i)
{
    a[i] = 5;
}
```

x86 Assembly - Indexing an array

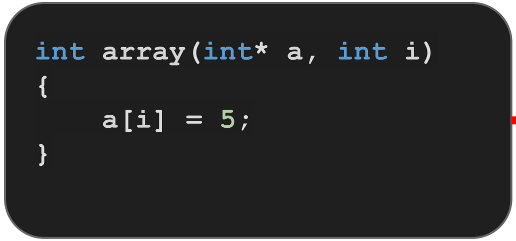


```c
int array(int* a, int i)
{
    a[i] = 5;
}
```

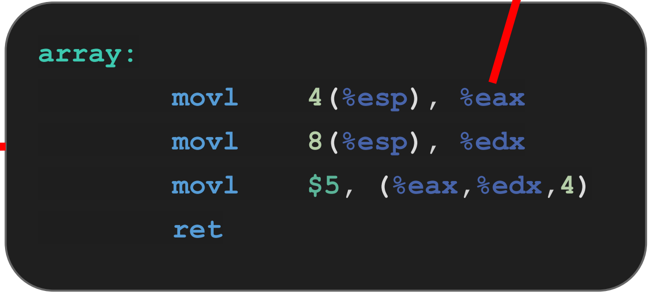


```
array:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        movl    $5, (%eax,%edx,4)
        ret
```

x86 Assembly - Indexing an array



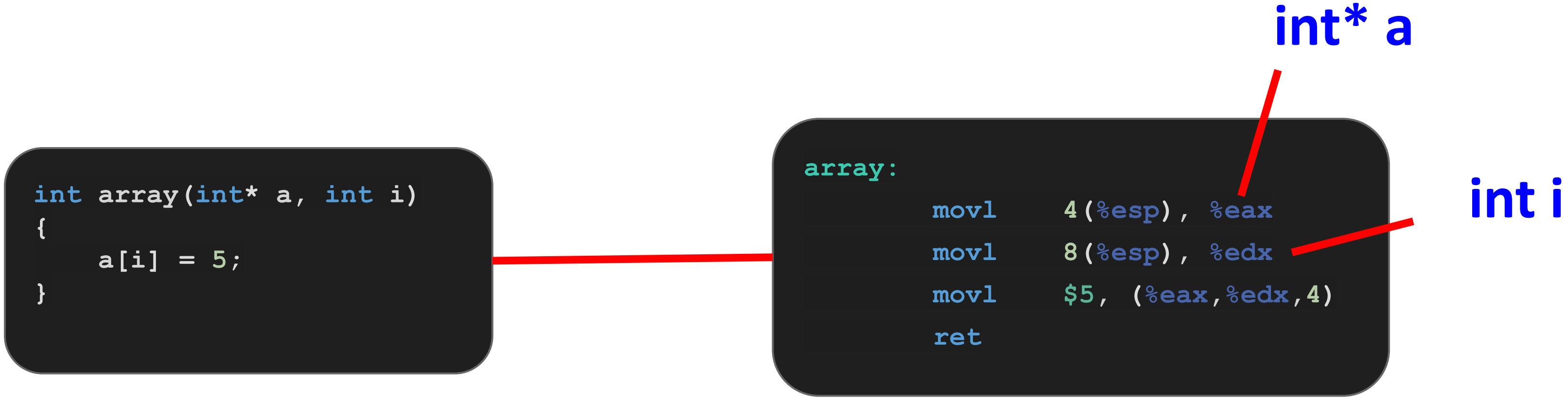
```
int array(int* a, int i)
{
    a[i] = 5;
}
```



```
array:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        movl    $5, (%eax,%edx,4)
        ret
```

int* a

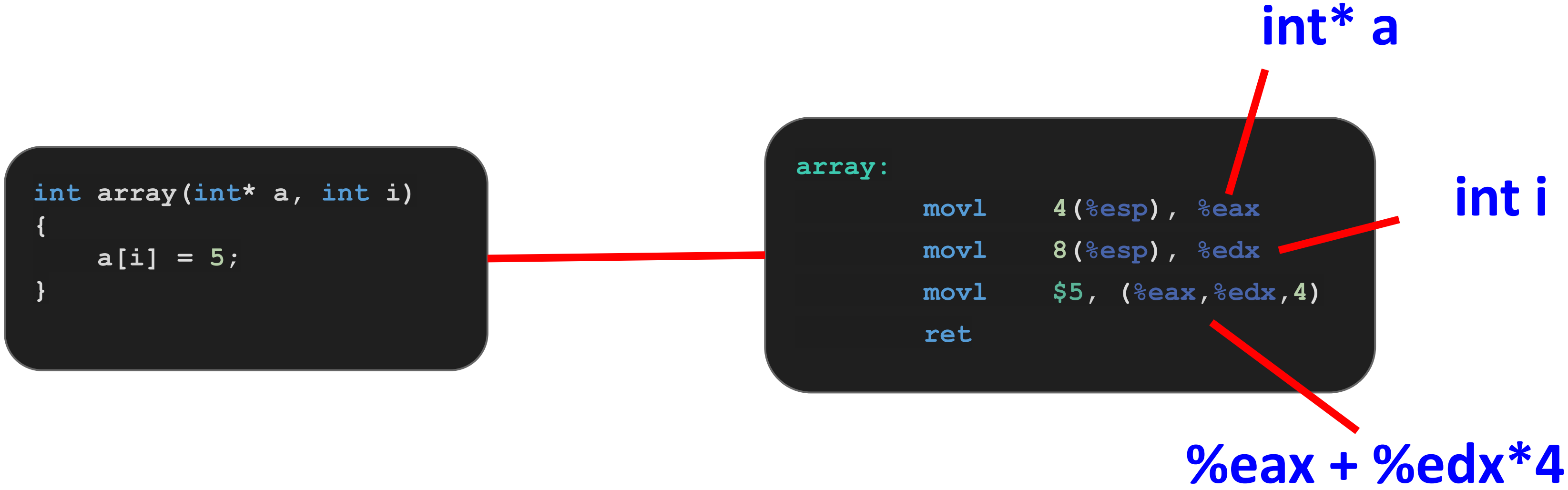
x86 Assembly - Indexing an array



```
int array(int* a, int i)
{
    a[i] = 5;
}
```

```
array:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        movl    $5, (%eax,%edx,4)
        ret
```

x86 Assembly - Indexing an array



```
int array(int* a, int i)
{
    a[i] = 5;
}
```

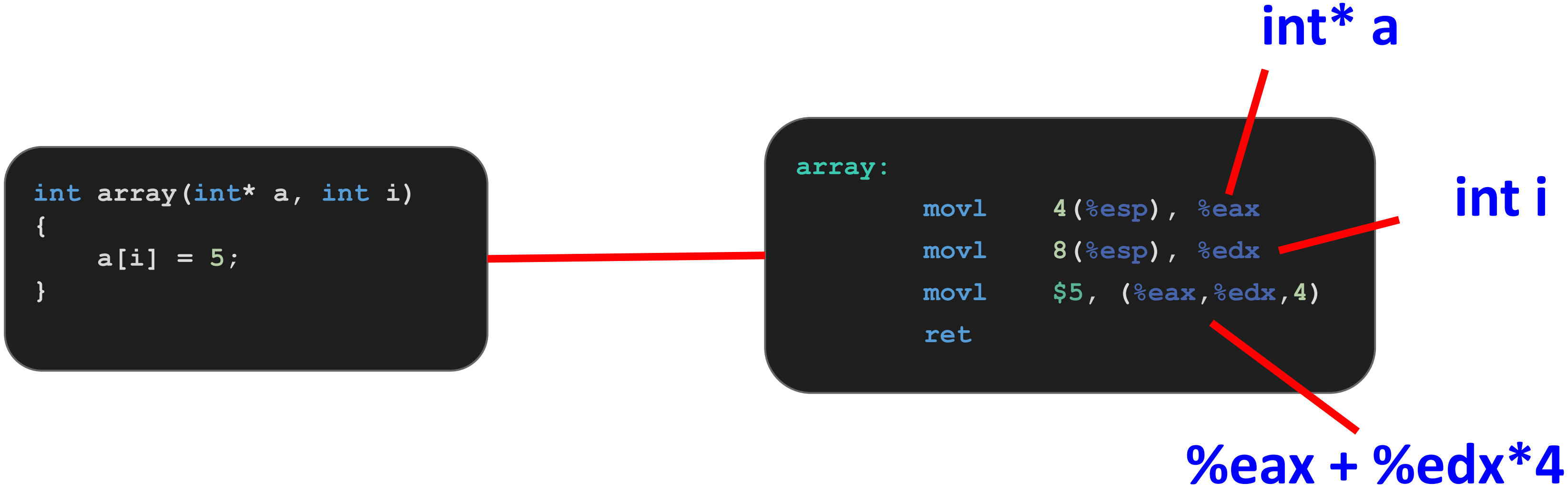
```
array:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        movl    $5, (%eax,%edx,4)
        ret
```

int* a

int i

%eax + %edx*4

x86 Assembly - Indexing an array



```
int array(int* a, int i)
{
    a[i] = 5;
}
```

```
array:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        movl    $5, (%eax,%edx,4)
        ret
```

Try it out today! -> https://godbolt.org/

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
    - Each instructions has its own side-effects on the flags

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
  - Each instructions has its own side-effects on the flags
- Carry (CF) - Arithmetic carry/ borrow

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
    - Each instructions has its own side-effects on the flags
- Carry (CF) - Arithmetic carry/ borrow
- Parity (PF) - Odd or even number of bits set

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
  - Each instructions has its own side-effects on the flags
- Carry (CF) - Arithmetic carry/ borrow
- Parity (PF) - Odd or even number of bits set
- Zero (ZF) - Result was zero

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
  - Each instructions has its own side-effects on the flags
- Carry (CF) - Arithmetic carry/ borrow
- Parity (PF) - Odd or even number of bits set
- Zero (ZF) - Result was zero
- Sign (SF) - Most significant bit was set

x86 Assembly - Flags

- Flag registers are special registers that are set by some instructions
  - Each instructions has its own side-effects on the flags
- Carry (CF) - Arithmetic carry/ borrow
- Parity (PF) - Odd or even number of bits set
- Zero (ZF) - Result was zero
- Sign (SF) - Most significant bit was set
- Overflow (OF) - Result does not fit into the location

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`
- `cmpl %eax,%ebx`

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`
- `cmpl %eax,%ebx`
  - Calculates `%ebx-%eax` and sets flags accordingly

x86 Assembly - Setting Flags

```
cmpl %eax,%ebx
```

eax	ebx
7	7

CF	
PF	
ZF	🟩
SF	
OF	

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`
- `cmpl %eax,%ebx`
    - Calculates `%ebx-%eax` and sets flags accordingly
- `testl %eax,%ebx`

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`
- `cmpl %eax,%ebx`
  - Calculates `%ebx-%eax` and sets flags accordingly
- `testl %eax,%ebx`
  - Calculates `%ebx&%eax` and sets flags accordingly

x86 Assembly - Setting Flags

- One way to set flags is by using `cmp` and `test`
- `cmpl %eax,%ebx`
  - Calculates `%ebx-%eax` and sets flags accordingly
- `testl %eax,%ebx`
  - Calculates `%ebx&%eax` and sets flags accordingly
  - `testl %eax,%eax` is the same as `cmpl $0,%eax`

x86 Assembly - Branches

- `je` *`label`*
  - Jump if zero

CF	
PF	
ZF	🟩
SF	
OF	

x86 Assembly - Branches

- `je label`
  - Jump if zero
- `jne/jnz label`
  - Jump if non-zero

CF	
PF	
ZF	
SF	
OF	

x86 Assembly - Branches

- je *label*
  - Jump if zero
- jne/jnz *label*
  - Jump if non-zero
- js *label*
  - Jump if negative

CF	
PF	
ZF	
SF	🟩
OF	

x86 Assembly - Branches

- `je` *label*
  - Jump if zero
- `jne/jnz` *label*
  - Jump if non-zero
- `js` *label*
  - Jump if negative
- `jns` *label*
  - Jump if non-negative

CF	
PF	
ZF	
SF	
OF	