CSE251: System Programming

4. Assembly (1)

Seongil Wi

Department of Computer Science and Engineering

UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY
2007

Notification: HW1

- Print the input data by using the various data types

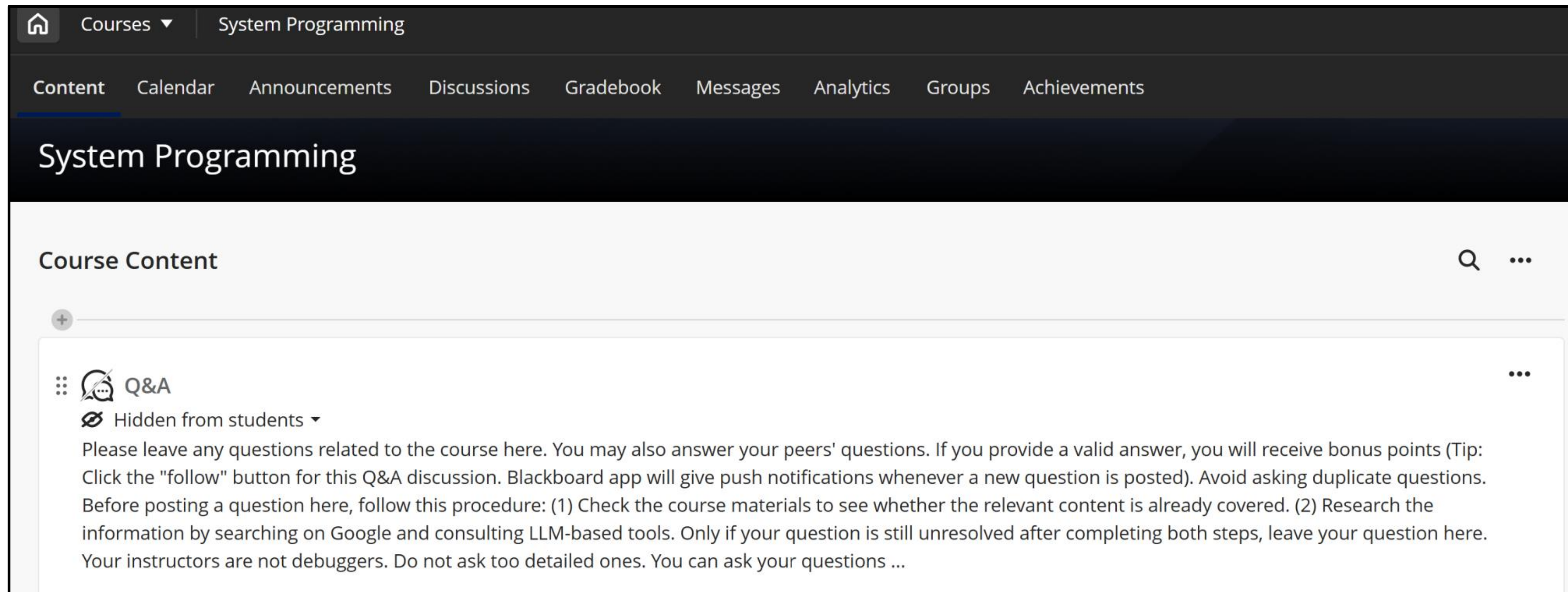
- Due March 24, 2026 : 11:59 pm
- To access your assigned server, please read the notice “Guide to SSH Access and Linux Basics” on Blackboard.

- Responsible TA: Donggyu Kim (dgkim@unist.ac.kr)

- If you have any requests or questions, ask the TAs via Blackboard.

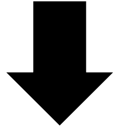
Blackboard Discussion and Participation

- We will use Blackboard discussions for Q&A
- If you answer other students' questions, and if the answer is valid one, you will receive bonus point!
  - Only the first student who posts a valid answer will receive bonus point. I will reply to indicate who receives the participation point.

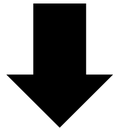


Recap: Compiler

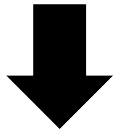
1. Preprocessor



2. Compiler

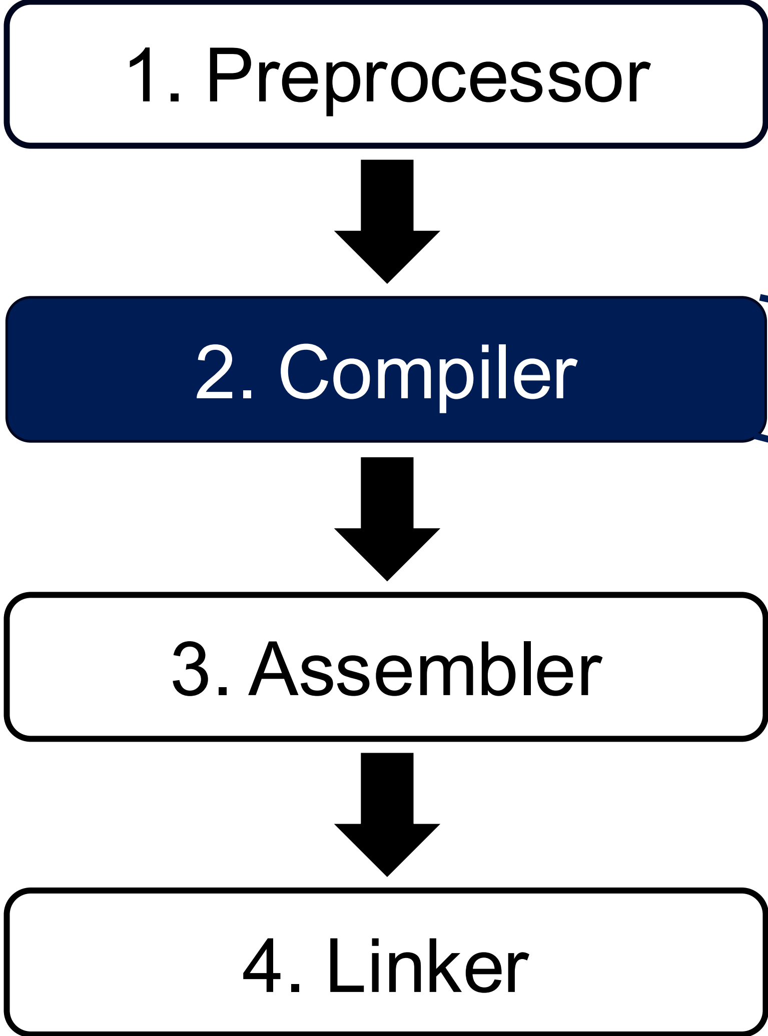


3. Assembler



4. Linker

Recap: Compiler



test.i

```
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

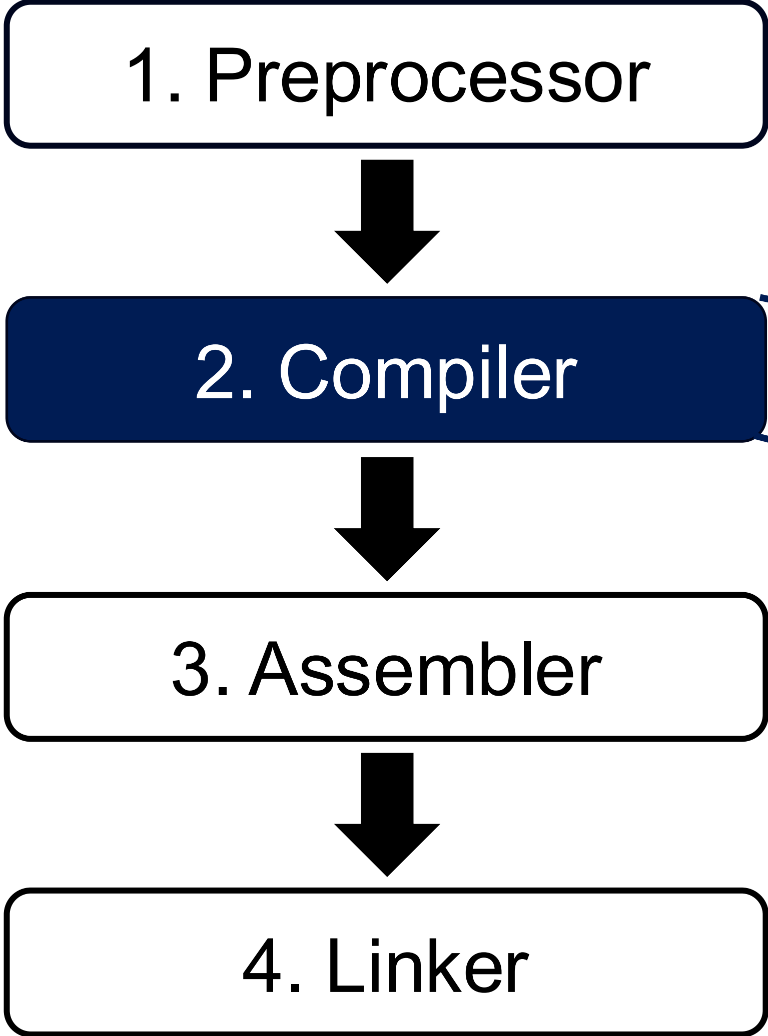
```
$gcc -S -masm=intel test.i -o
test.s
```

test.s

```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

The last human-readable format

Today's Topic



test.i

```
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

```
$gcc -S -masm=intel test.i -o
test.s
```

test.s

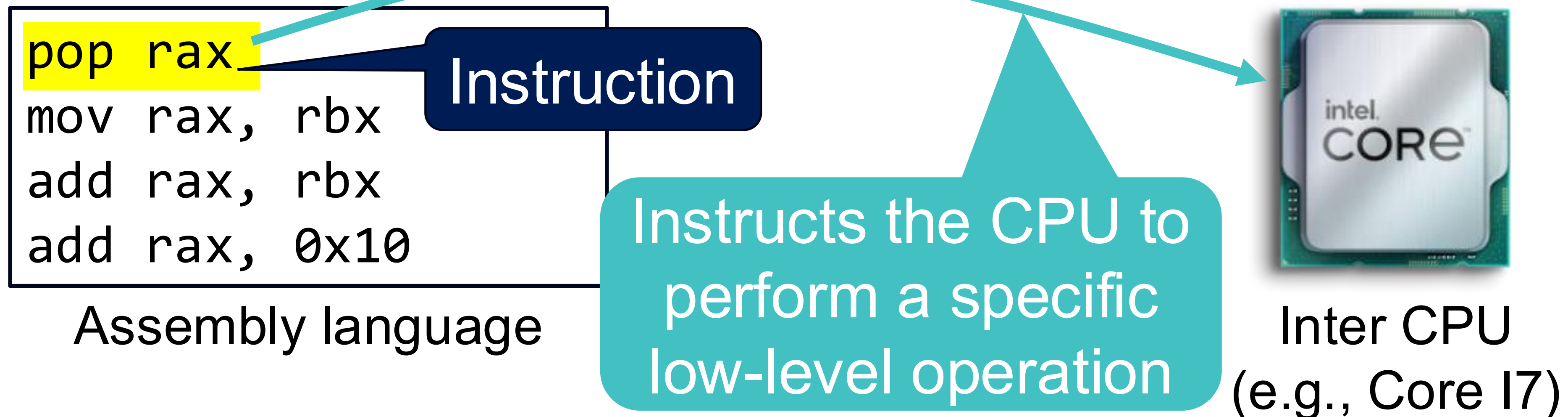
```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

Today's topic!

Compiler

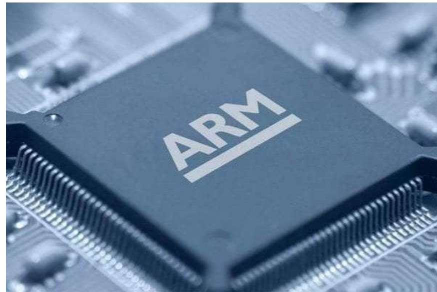
- Translates the source into the *assembly-language program*

- Assembly language (a.k.a, assembly, asm)
  - Low-level programming language that helps to communicate directly with computer hardware



Instruction Set

- The commands understood by a given architecture

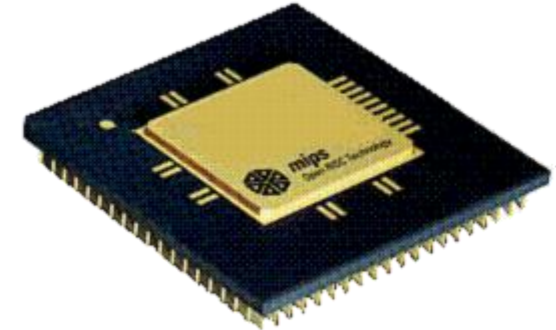


ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Instruction Set

- The commands understood by a given architecture

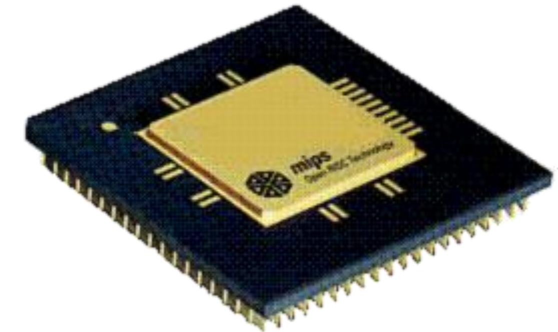
Different chips have different instruction sets

ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Instruction Set

- The commands understood by a given architecture

Different chips have different instruction sets

RM's
ins ctions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

Our focus:
Intel X86-64 (a.k.a., X64)

MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

x86-64 Architecture

x86 Instruction Set Architecture

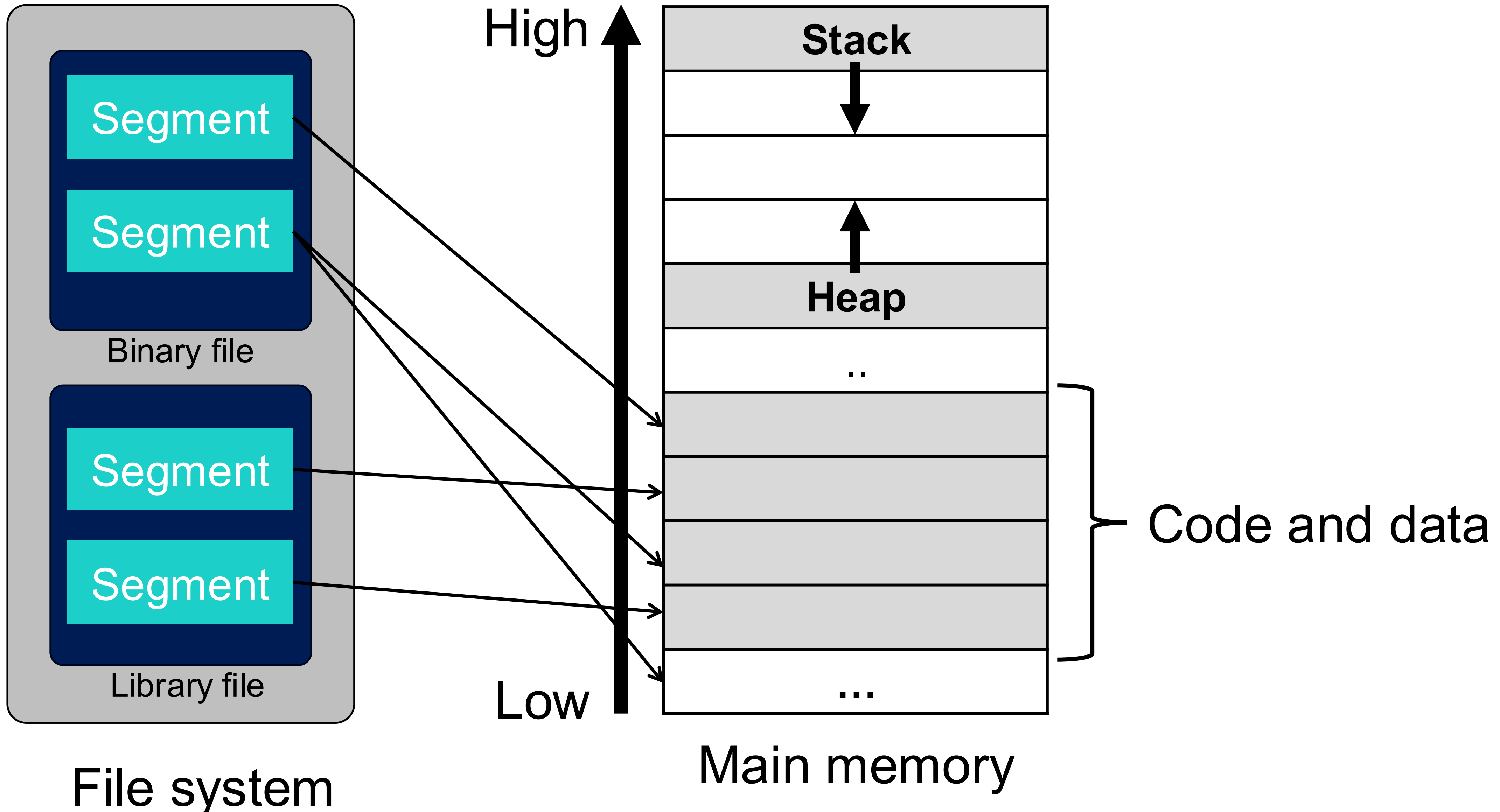
- Developed by Intel in 1978
- CISC (Complex Instruction Set Computer) architecture
- The most popular ISA

History of x86 ISA

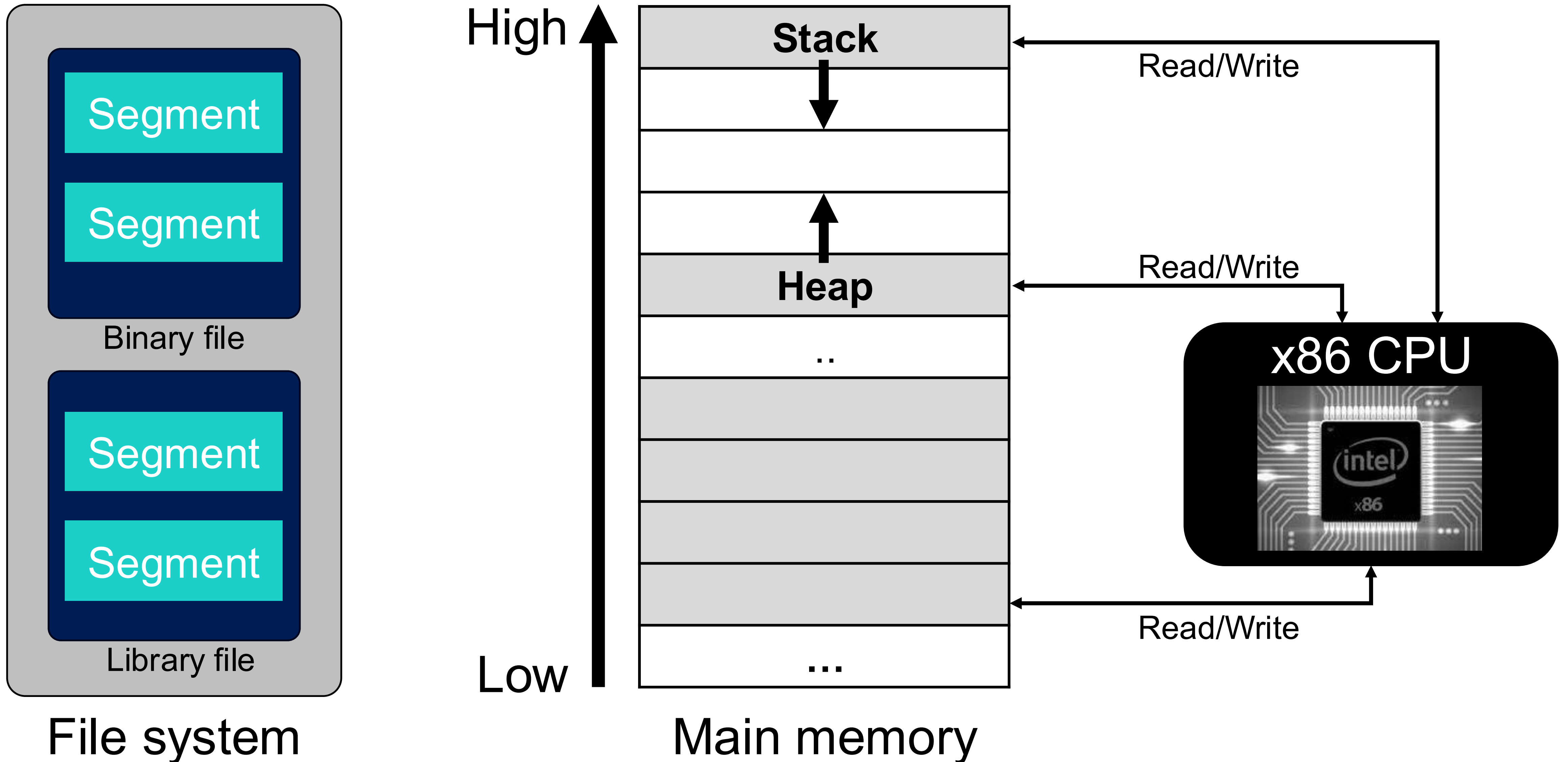
- (**8086**) 16-bit address space (in 1978)
- (**x86** or **IA-32**) 32-bit address space (in 1985)
- (**x86-64** or **x64** or **AMD64**) 64-bit address space (in 2003)

This course will focus on x86-64 (AMD64) architecture

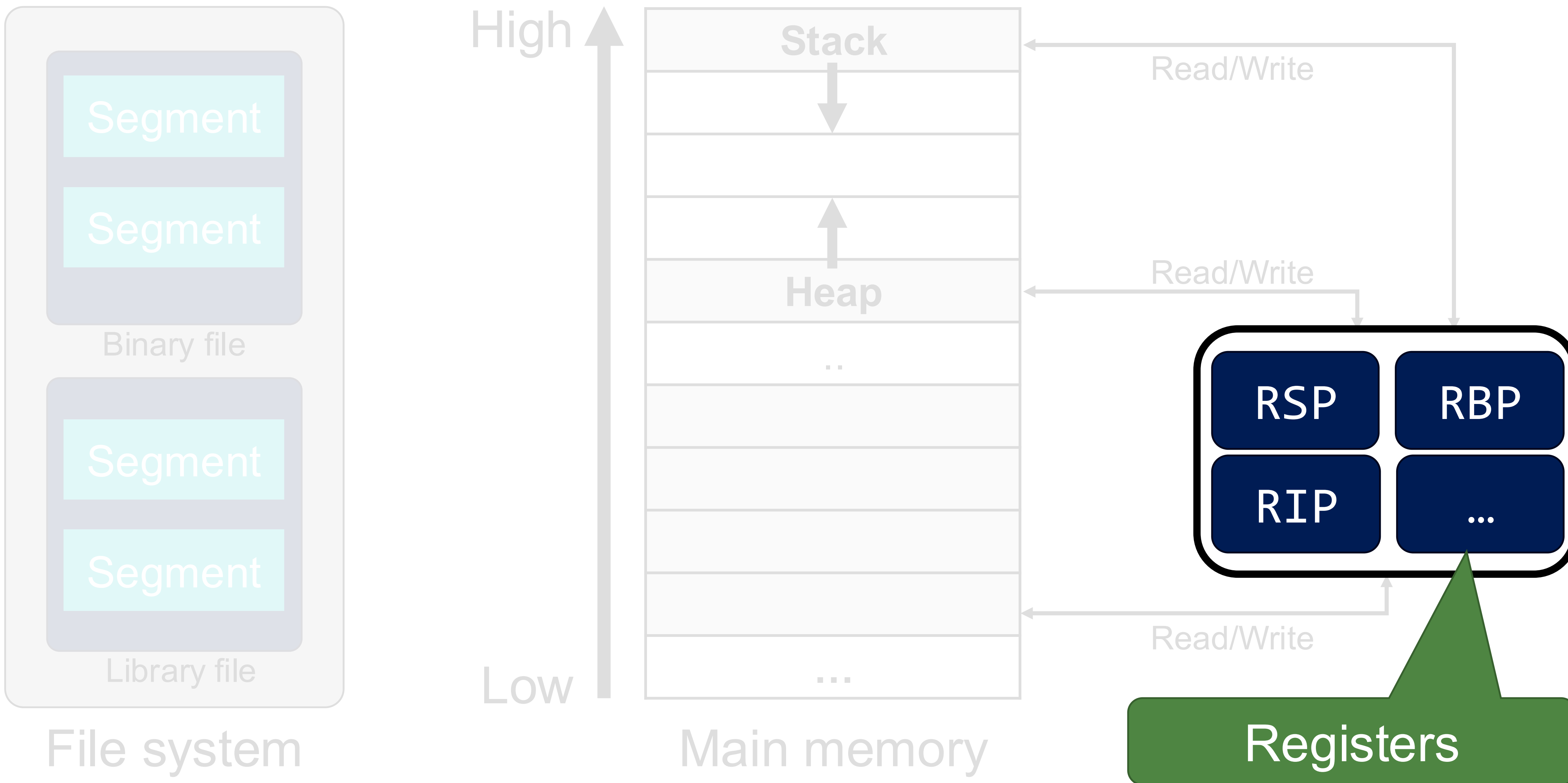
Memory Layout and CPU Registers



Memory Layout and CPU Registers



Memory Layout and CPU Registers



Registers

- Reside in CPU
- Temporary storage of data/instruction
- **Faster** than main memory
- Expensive

Registers in x86-64

- Program counter (instruction pointer)
  - **RIP:** points to the instruction to execute
- Stack pointers
  - **RSP:** points to the top of the stack
  - **RBP:** points to the base of the current stack frame
- Status register (FLAGS register)
  - **RFLAGS:** contains the current condition flags
- Other general purpose registers
  - **RAX**, **RBX**, **RCX**, **RDX**, **RSI**, **RDI**
  - **R8**, **R9**, **R10**, **R11**, **R12**, **R13**, **R14**, **R15**

All of them have a size of 64-bit

Size of Registers

- x86 registers are 32-bit. Intel says they have a size of a ***double word***

- x86-64 registers are 64-bit. Intel says they have a size of a ***quad word***

Wait, what is a “word”?

Word

- A word is the natural ***unit*** of data used by a processor
- Typically, a word is 32 bits on a 32-bit machine, and 64 bits on a 64-bit machine

However, Intel says a word is **16 bits** on both x86 and x86-64!

History of x86 Processors

- 1978: 8086
- 1982: 80286

16-bit processor,
Registers (`SP`, `BP`, `IP`, ...)

- 1985: 80386
- 1989: 80486
- ...

32-bit processor,
Registers (`ESP`, `EBP`, `EIP`, ...)

- 2003: Opteron
- 2005: Prescott
- 2006: Core 2
- 2008: Core i7
- …

64-bit processor,
Registers (`RSP`, `RBP`, `RIP`, ...)

History of x86 Processors

- 1978: 8086
- 1982: 80286
- 1985: 80386

16-bit processor,
Registers (SP, BP, IP, ...)

A "word" is 16 bits due to backward compatibility with the original 16-bit processor

- 2003: Opteron
- 2005: Prescott
- 2006: Core 2
- 2008: Core i7
- ...

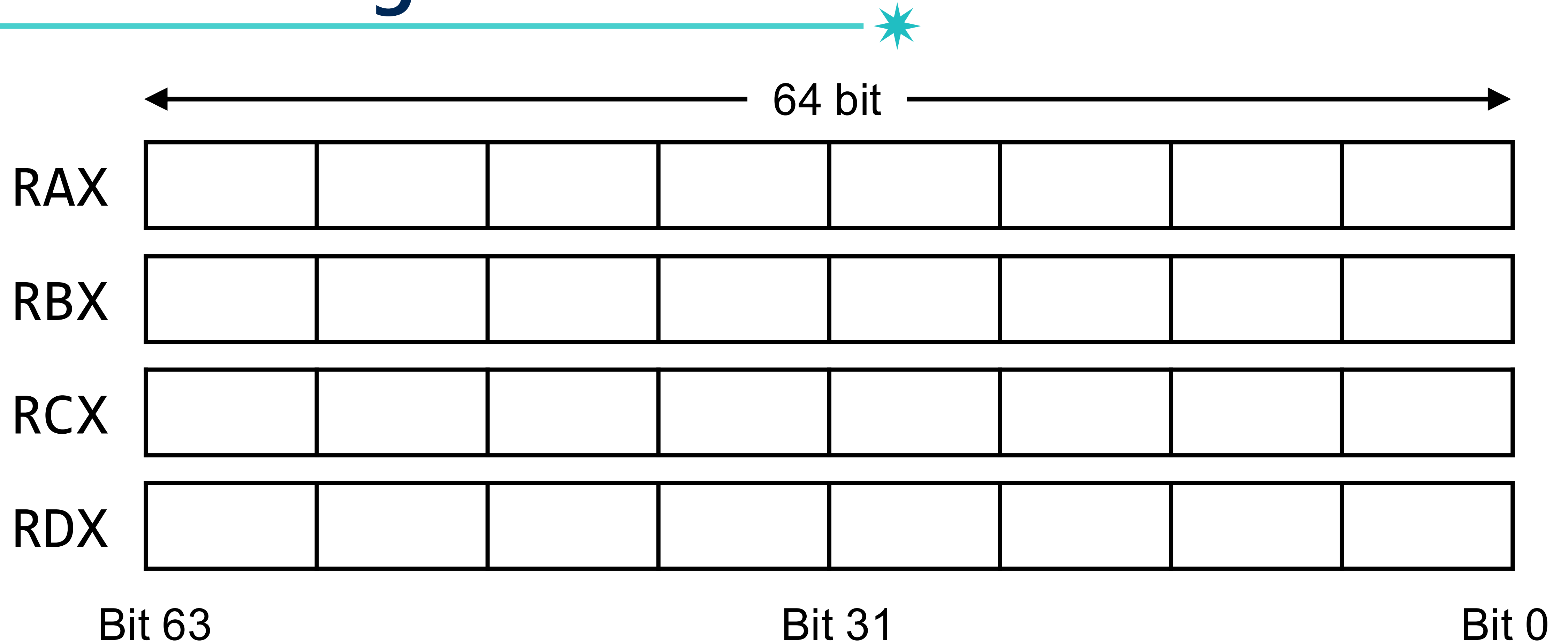
64-bit processor,
Registers (RSP, RBP, RIP, ...)

Intel x86 Naming Convention

- Word = 16 bits

- Double Word (DWORD) = 32 bits
- Quad Word (QWORD) = 64 bits

X86-64 Register Access



X86-64 Register Access

64 bit

RAX					EAX
RBX					EBX
RCX					ECX
RDX					EDX

Bit 63 Bit 31 Bit 0

X86-64 Register Access

64 bit

RAX							AX
RBX							BX
RCX							CX
RDX							DX

Bit 63 Bit 31 Bit 0

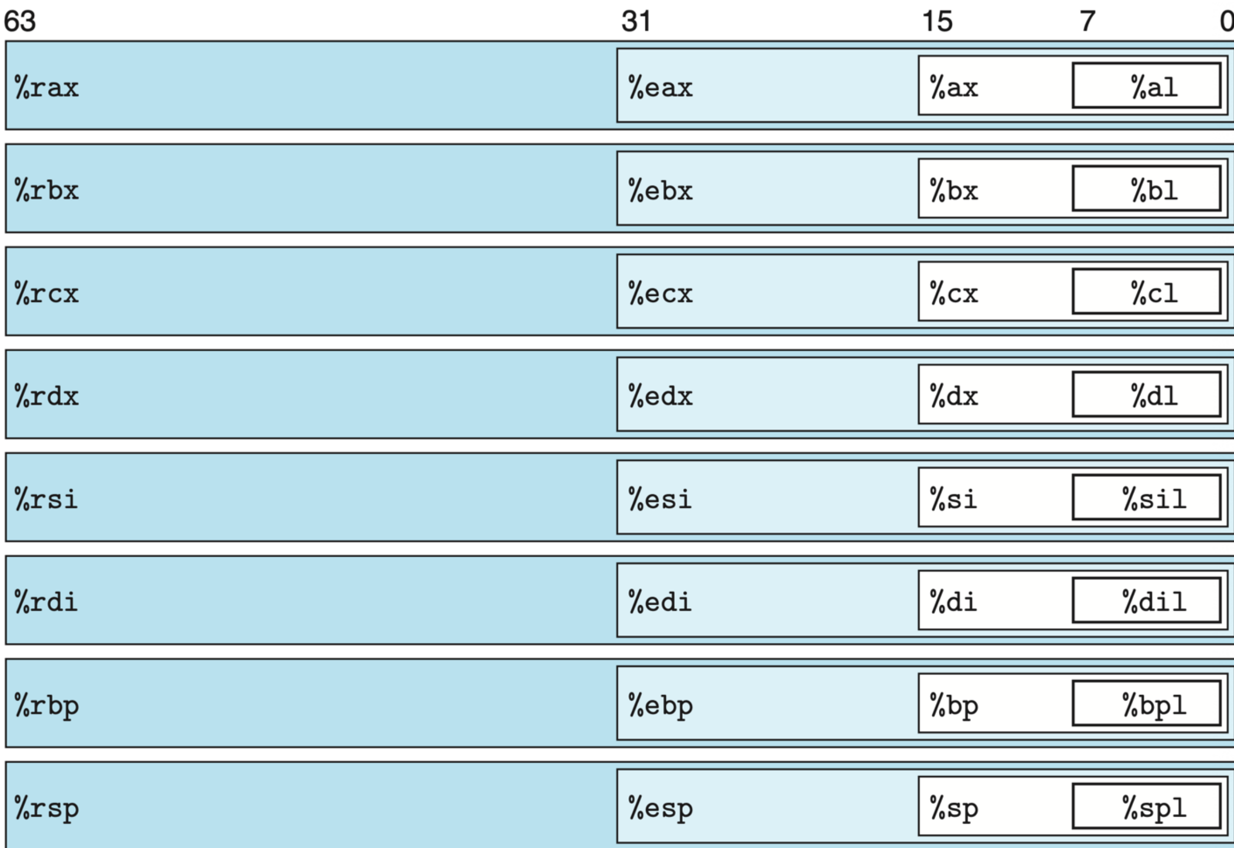
X86-64 Register Access

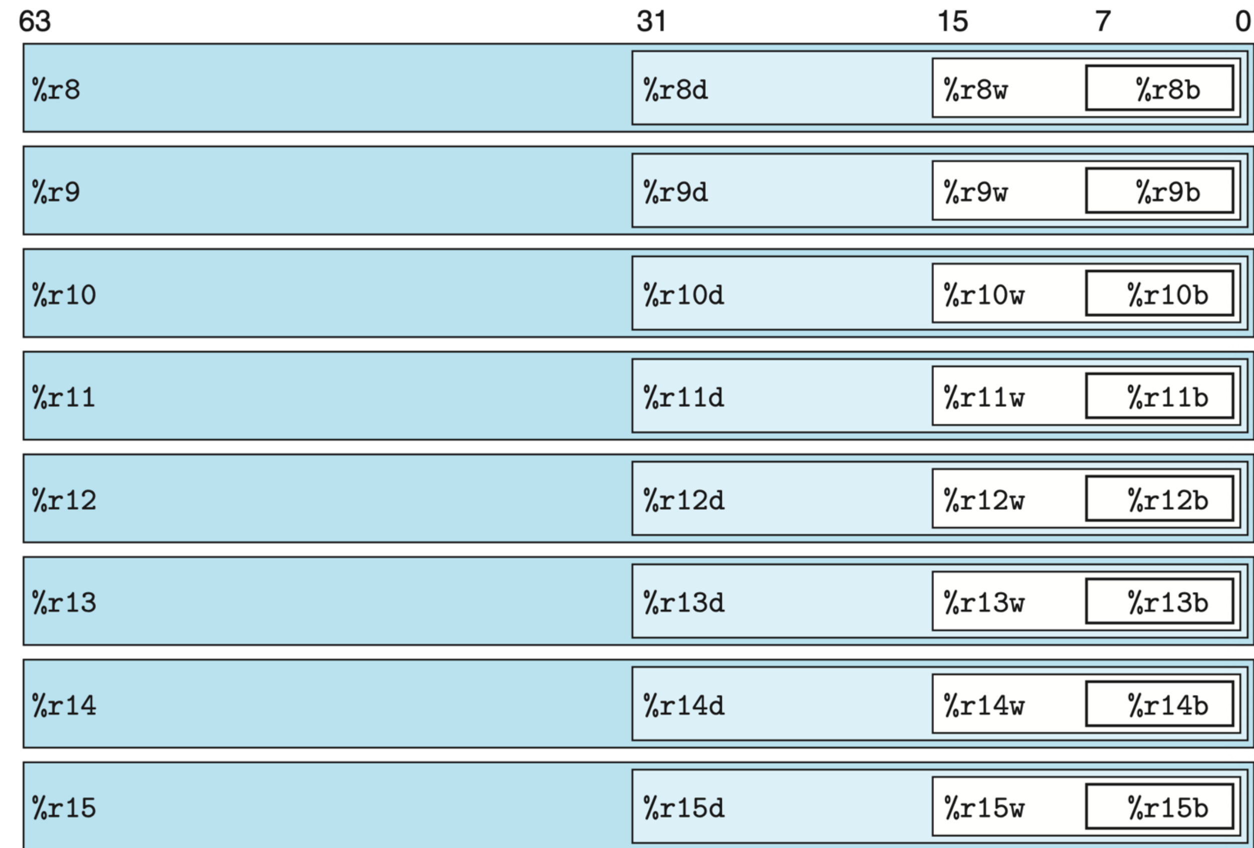
64 bit

RAX							AH	AL
RBX							BH	BL
RCX							CH	CL
RDX							DH	DL

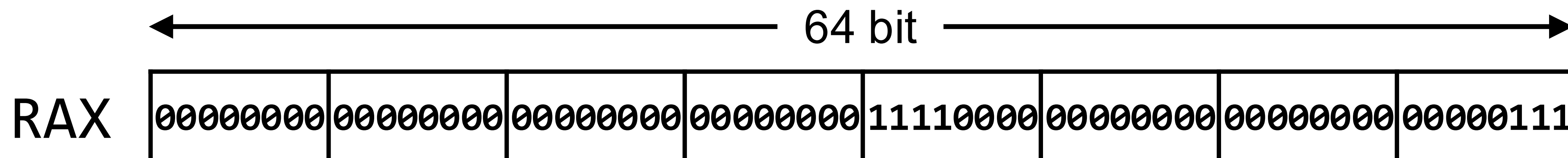
Bit 63 Bit 31 Bit 0

X86-64 Register Access (Summary)





X86-64 Register Access (Example)



What is the value of EAX?

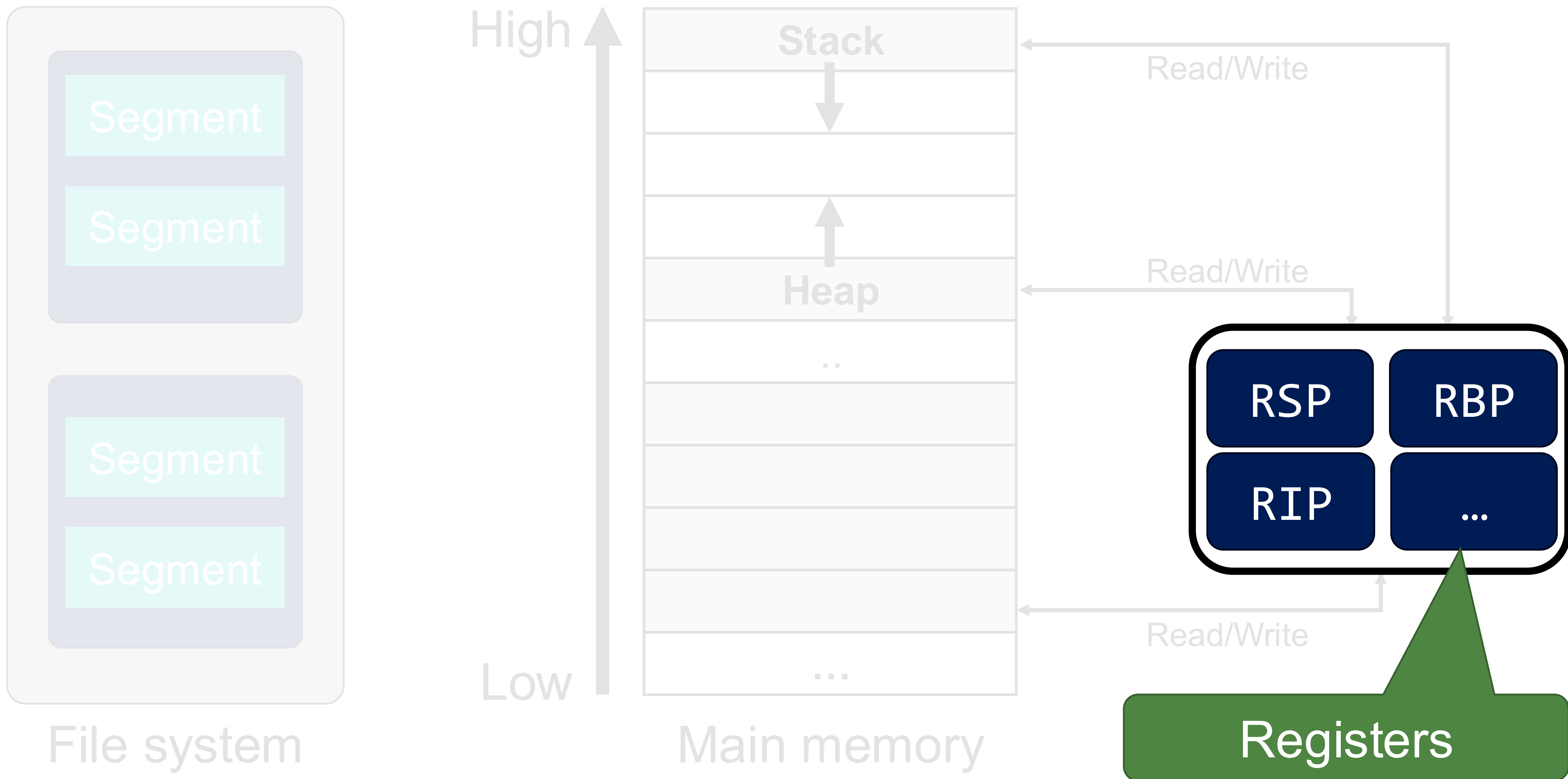
EAX 11110000 00000000 00000000 00000111

AX 00000000 00000111

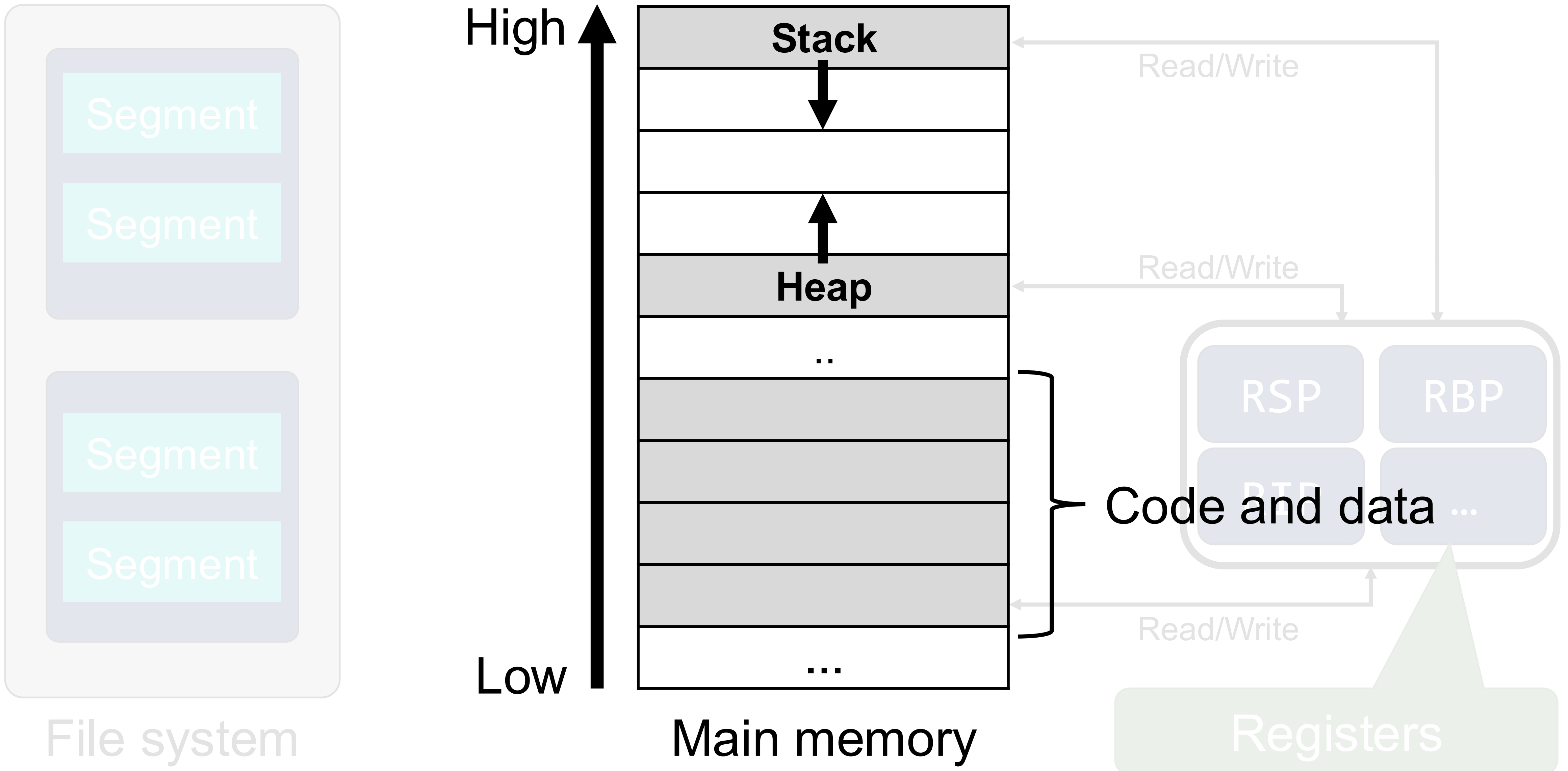
AH 00000000

AL 00000111

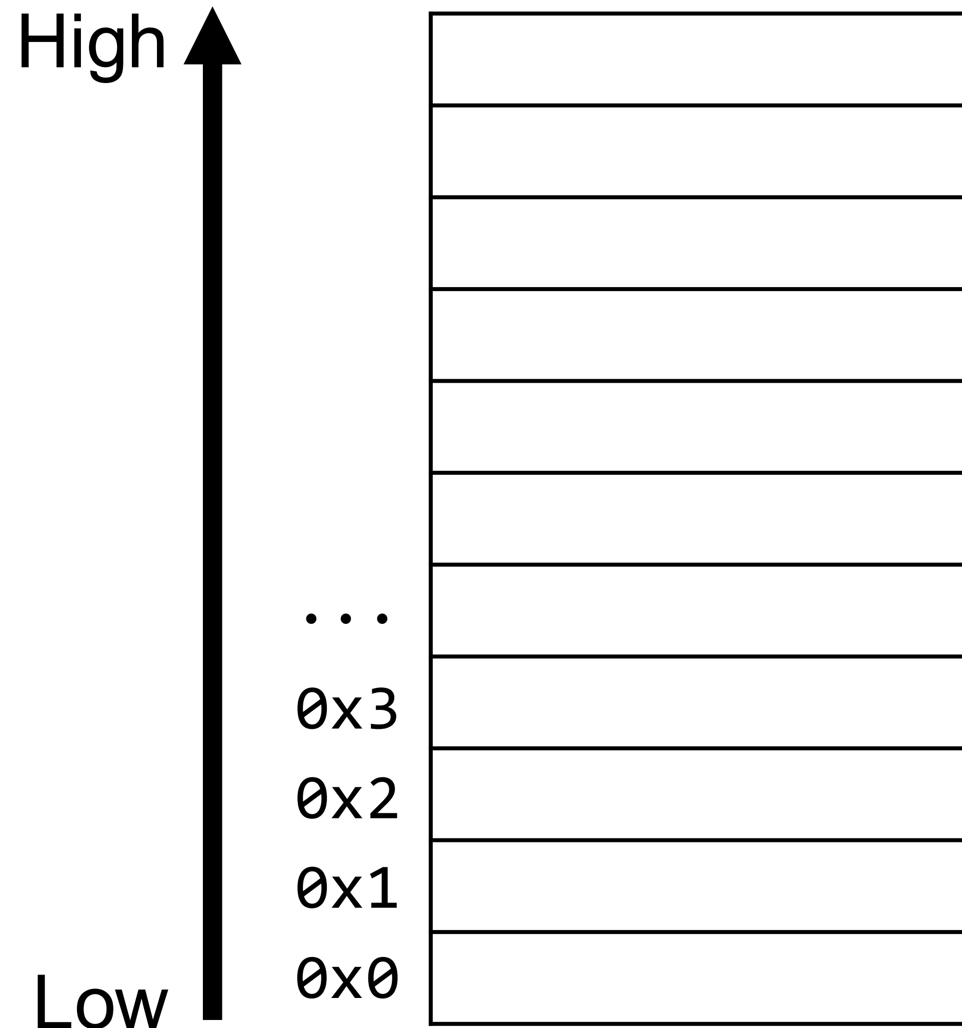
Memory Layout and CPU Registers



Memory Layout and CPU Registers

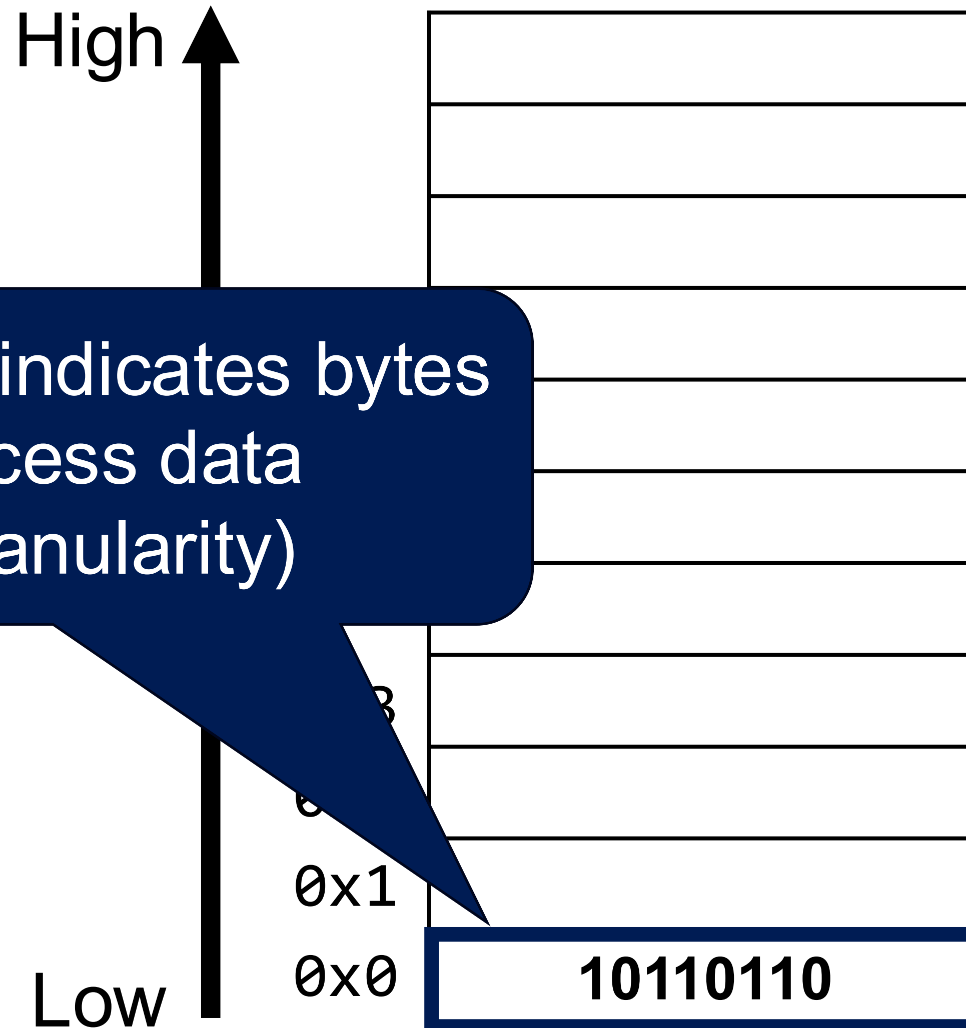


Memory Structure in X86



Just a large 1-dimensional array

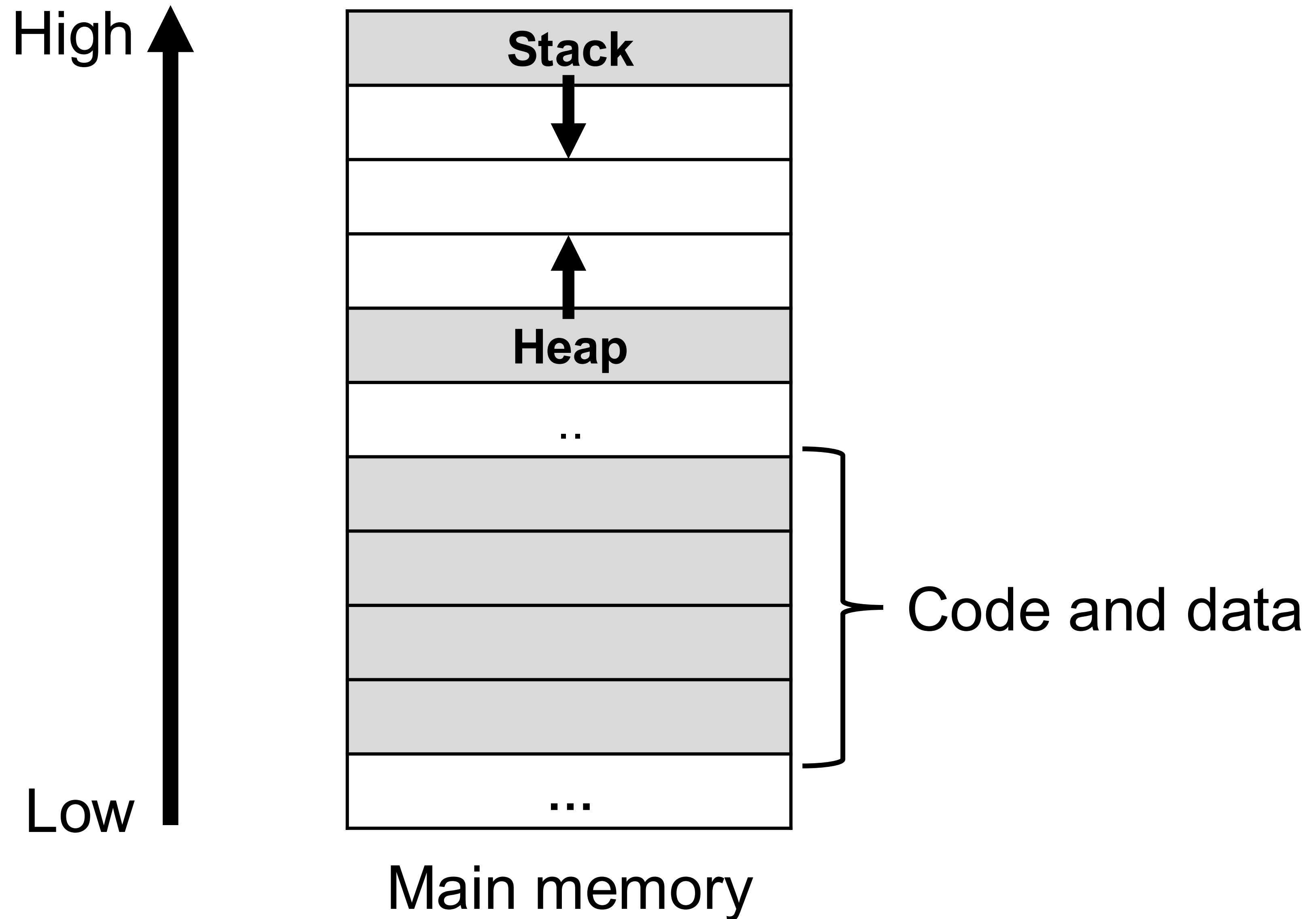
x86 Memory Access = Byte Addressing



Memory address indicates bytes
(We can access data
at a byte granularity)

Just a large 1-dimensional array

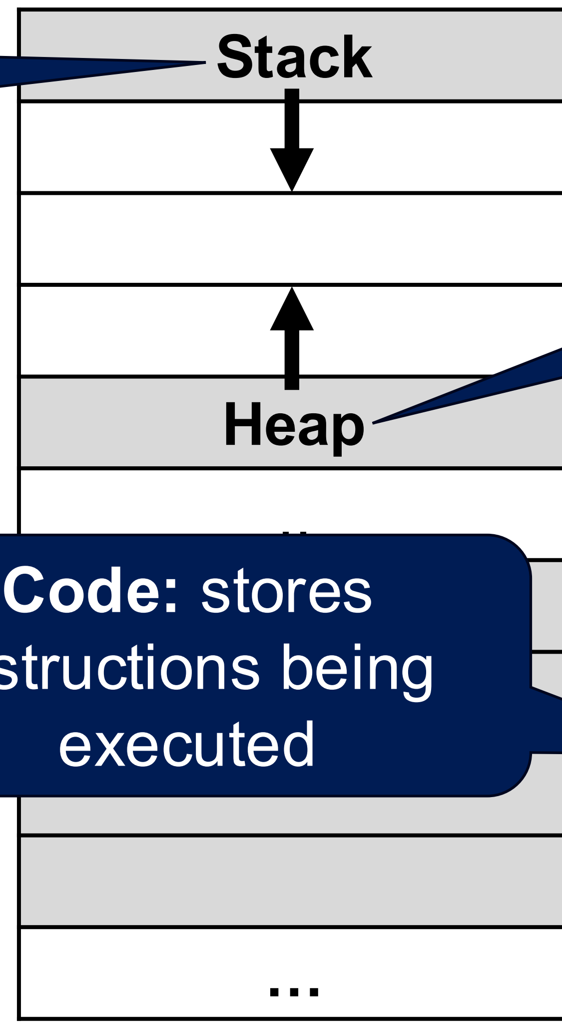
Memory Layout (Overview)



Memory Layout (Overview)

Stack: stores local variables

Heap: dynamic memory for programmer to allocate (e.g., via `malloc()` or `realloc()`)



Code: stores instructions being executed

Data: stores global variables

Assembly Basics

Basic Formats

- Three formats of Instructions
  - 2 operands
  - 1 operands
  - 0 operands

Basic Format #1: Instructions with 2 Operands


```
mov rax, rbx
```

Opcode

Operand 1

Operand 2

Basic Format #2: Instructions with 1 Operand


```
inc rax
```

Opcode

Operand

Basic Format #3: Instructions with 0 Operand

```
ret
```

Intel vs AT&T Format

- There are two ways to represent x86 assembly code
  - **Intel Syntax:** the original assembly syntax introduced by Intel
  - **AT&T Syntax:** used by UNIX and Linux.

AT&T

```
mov %eax, %ebx
```

Intel

```
mov ebx, eax
```

We will use the Intel syntax

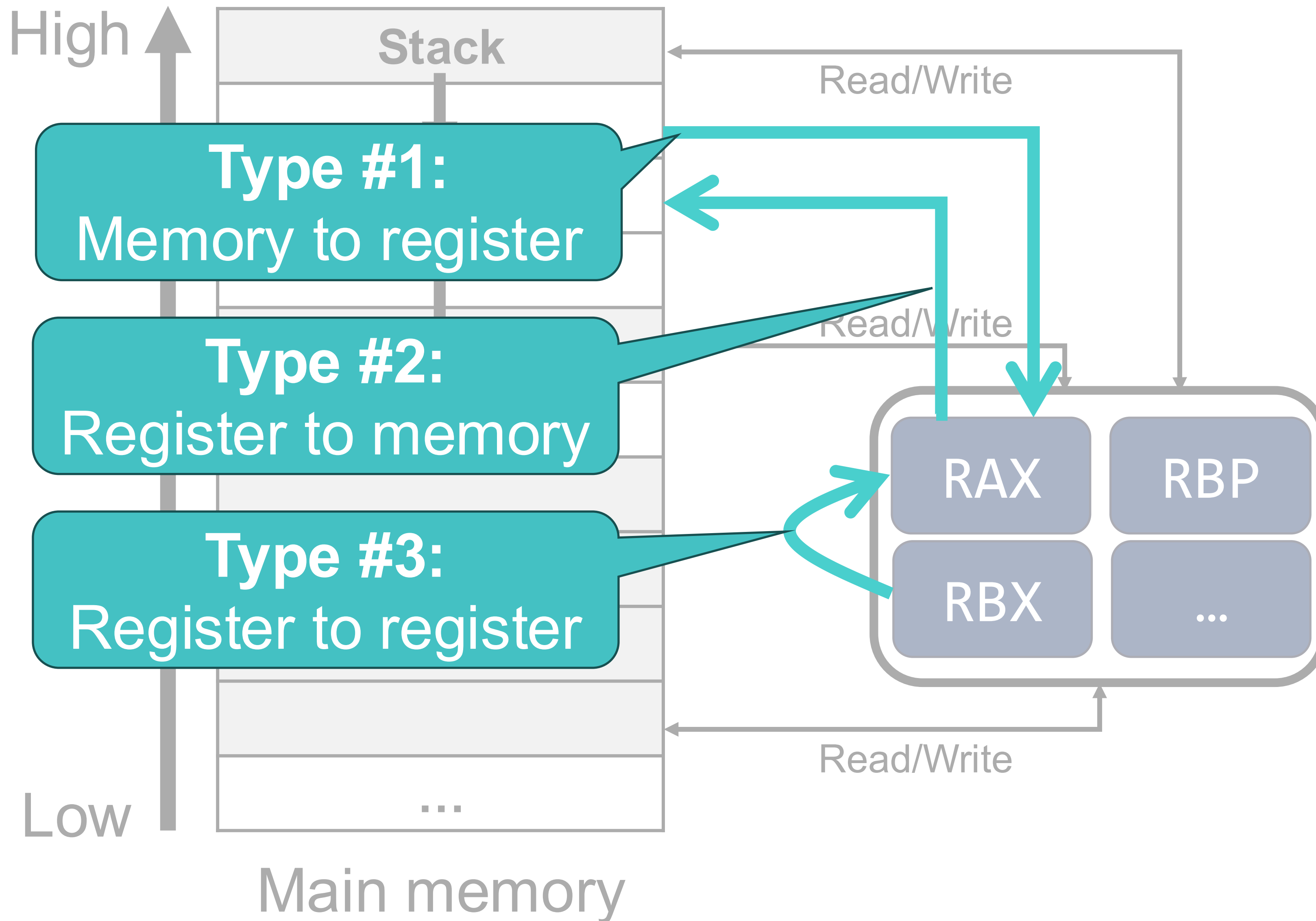
Opcode Decides Semantics

Data transfer

```
mov rax, rbx
sub rsp, 0x8
inc rax
```

```
rax ← rbx
rsp ← rsp – 0x8
rax ← rax + 1
```

Data Transfer (mov)



Data Transfer (`mov`)

Type #1:
Memory to register

Type #2:
Register to memory

Type #3:
Register to register

```
mov rax, rbx
```

Register to Register

```
mov rax, rbx
```

Registers (Initial state)

rax	0xdeadbeefdeadbeef
rbx	0x0000000000000004

Register to Register

```
mov rax, rbx
```

Registers (After execution)

rax	0x0000000000000004
rbx	0x0000000000000004

Data Transfer (`mov`)

Type #1:
Memory to register

```
mov rax, [rbx]
```

Type #2:
Register to memory

Type #3:
Register to register

```
mov rax, rbx
```

Memory to Register

Memory pointed by rbx

```
mov rax, [rbx]
```

Registers (Initial state)

rax	0xdeadbeefdeadbeef
rbx	0x0000000000000008

Address	Value
...	
0x18	...
0x10	0x4122a13c4122a13c
0x8	0x273faace273faace
0x0	0xbffa0220bffa0220

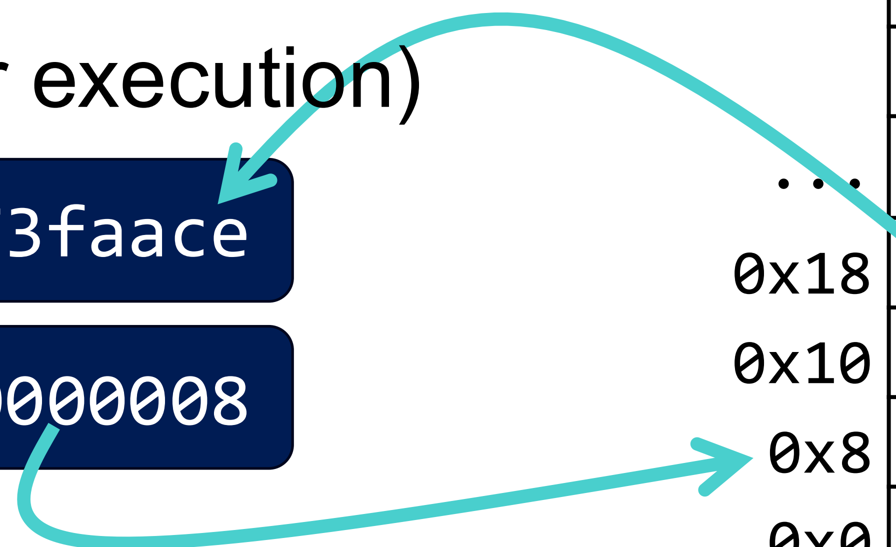
Memory to Register

Memory pointed by rbx

```
mov rax, [rbx]
```

Registers (After execution)

rax	0x273faace273faace
rbx	0x0000000000000008



Data Transfer (mov)

Type #1:
Memory to register

```
mov rax, [rbx]
```

Type #2:
Register to memory

```
mov [rax], rbx
```

Type #3:
Register to register

```
mov rax, rbx
```

Register to Memory

```
mov [rax], rbx
```

Registers (Initial state)

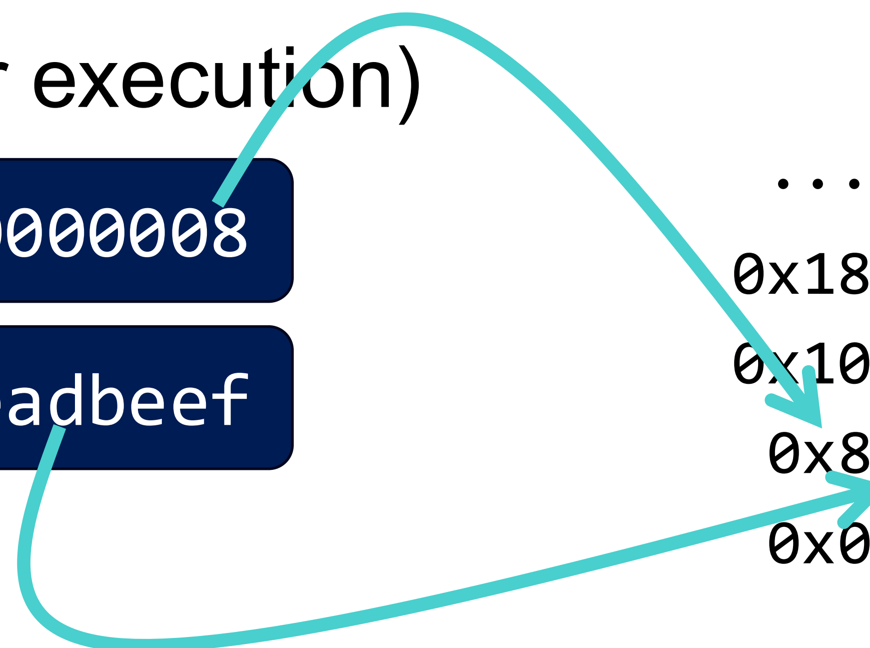
Register	Value
rax	0x0000000000000008
rbx	0xdeadbeefdeadbeef

Memory (Initial state)

Address	Value
...	
0x18	...
0x10	0x4122a13c4122a13c
0x8	0x273faace273faace
0x0	0xbffa0220bffa0220

Register to Memory

```
mov [rax], rbx
```



Memory (After execution)

...	
0x18	...
0x10	0x4122a13c4122a13c
0x8	0xdeadbeefdeadbeef
0x0	0xbffa0220bffa0220

Data Transfer (mov)

Type #1:
Memory to register

```
mov rax, [rbx]
```

Type #2:
Register to memory

```
mov [rax], rbx
```

Type #3:
Register to register

```
mov rax, rbx
```

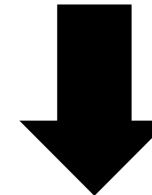
Type #4:
Constant to register/memory

```
mov rax, 42
mov BYTE PTR [rax], 42
```

Constant to Register

```
mov rax, 0x8
```

eax 0xdeadbeefdeadbeef



eax 0x0000000000000008

Constant to Memory

```
mov BYTE ptr [rax], 0x8
```

Pointer directive

Pointer Directive

```
mov [rsi], al     ; ok
mov [rsi], 1      ; error
```

Error: ambiguous operand size for ‘mov’

Because it could be any of the followings

- `mov BYTE PTR [rsi], 1`
- `mov WORD PTR [rsi], 1`
- `mov DWORD PTR [rsi], 1`
- `mov QWORD PTR [rsi], 1`

Therefore, we need pointer directive ☺

Examples: Moving Data Around

- `mov eax, ebx`
- `mov al, bl`

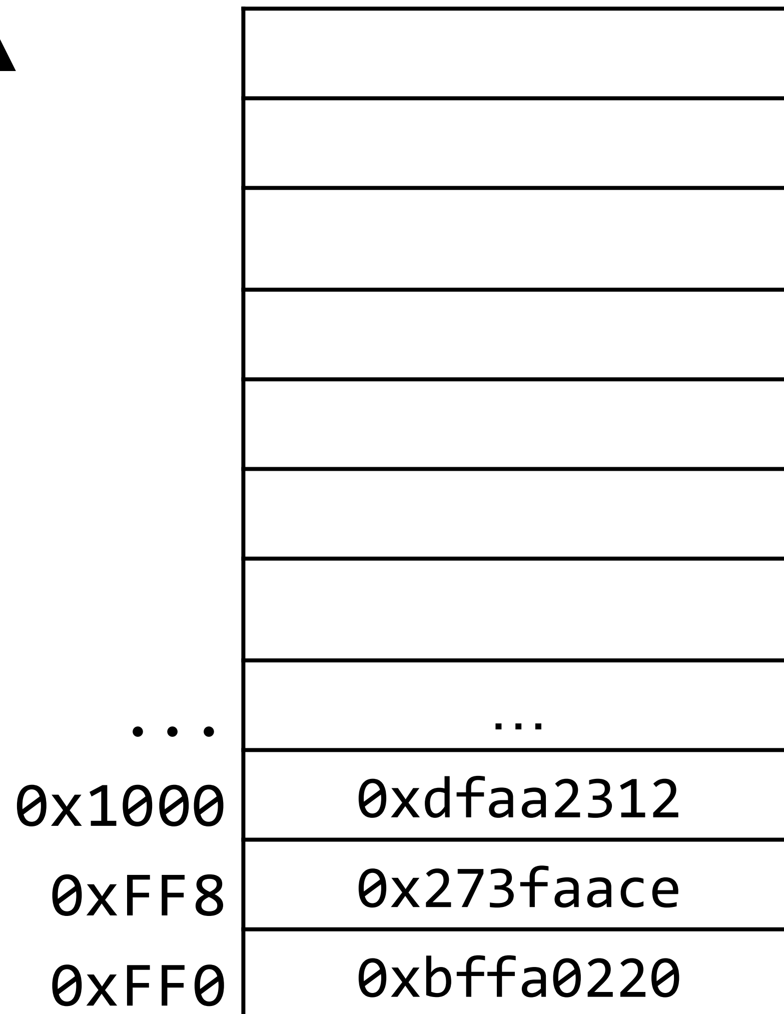
  } Register to Register
- `mov [rax], rbx` } Register to Memory (Write data to memory)
- `mov rcx, [rbx]`
- `mov rcx, [rbx + rbx * 4]`
- `mov al, BYTE PTR [rsi]`

  } Memory to Register (Read data from memory)
- `mov rax, 42`
- `mov BYTE PTR [rax], 42`

  } Constant to Memory/Register

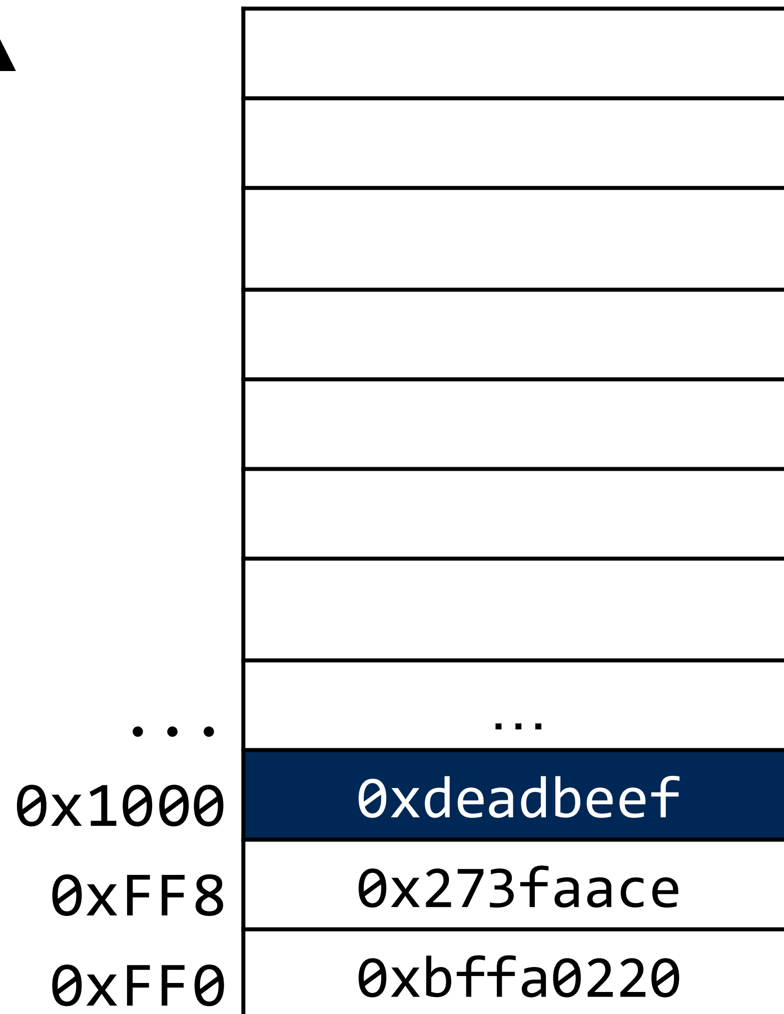
Storing a DWORD in Memory

```
mov [eax], 0xdeadbeef      ; eax = 0x1000
```



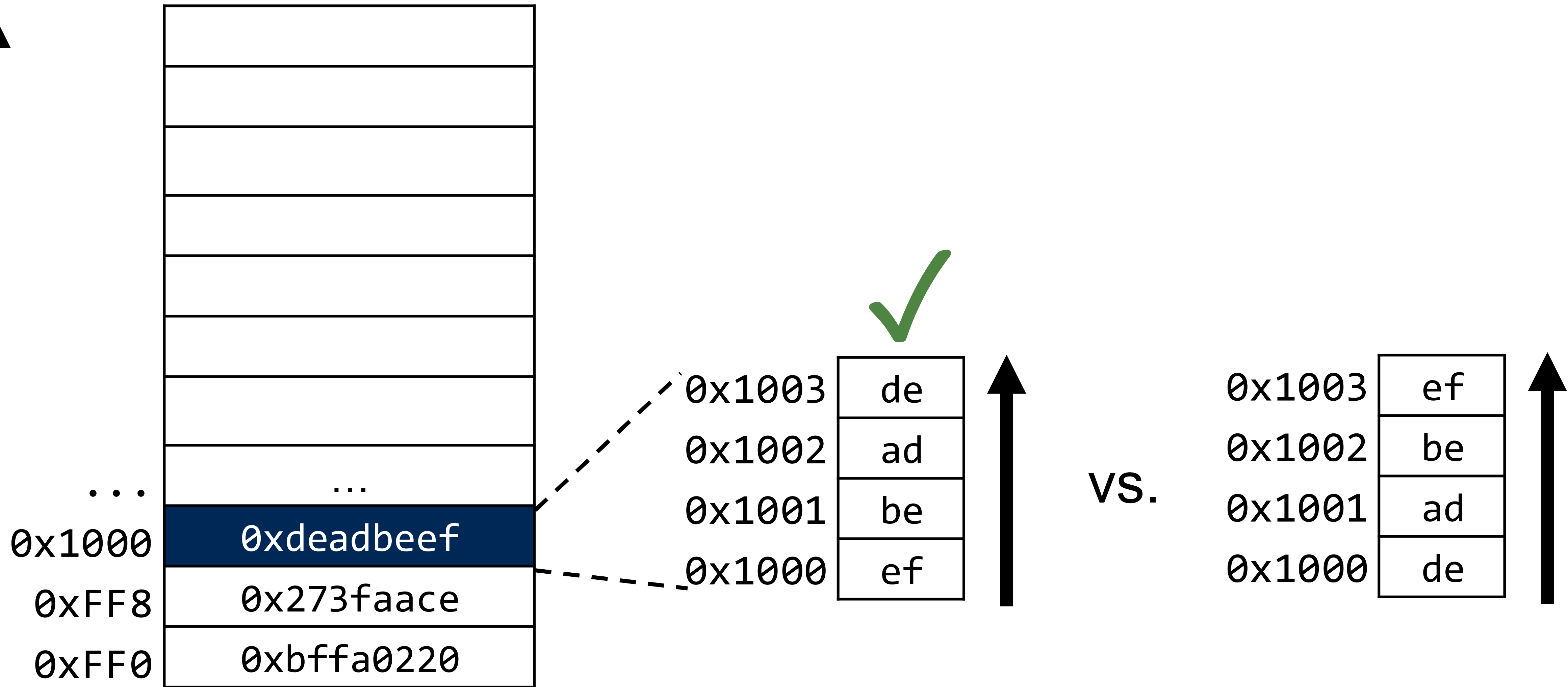
Storing a DWORD in Memory


```
mov [eax], 0xdeadbeef      ; eax = 0x1000
```



Storing a DWORD in Memory

```
mov [eax], 0xdeadbeef     ; eax = 0x1000
```



Recap: Endianness

- The order in which a sequence of bytes are stored in memory

- Big Endian = The MSB goes to the lowest address
- Little Endian = The LSB goes to the lowest address

X86 (and X86_64) uses Little Endian

Addressing Modes

Specify how a memory operand is interpreted to derive an effective address

- **register**
  - ✓`mov rax, [rax]`
- **register + register**
  - ✓`mov rax, [rax + rbx]`
- **displacement**
  - ✓`mov rax, [0x1000]`
- **register + register × scale + displacement**
  - ✓`mov rax, [rax + rbx * 4 + 0x1000]`

Addressing Modes

Specify how a memory operand is interpreted to derive an effective address

$$\left[\texttt{Register} + \texttt{Register} \times \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix} + \begin{matrix} \texttt{displacement} \\ \text{(32-bit integer)} \end{matrix} \right]$$

Load Effective Address (lea)


```
lea rax, [rbx]
lea rax, [rbp-0x8]
```

} Memory address to Register

Same as '&' in C

What is the Difference?


```
mov rax, [rbp + 0x10]
vs.
lea rax, [rbp + 0x10]
```

```
rax ← *(rbp + 0x10)
vs.
rax ← (rbp + 0x10)
```

Arithmetic/Logical Operations

- `add rax, [rbx]`
- `sub rsp, 0x40`
- `inc rcx`
- `dec rdx`
- `and [rax + rcx], rbx`
- `xor edx, ebx`
- `shr rax, 1      // shift logical right`
- `sar rax, 1      // shift arithmetic right`
- ...

Control Flows

```
if ( x ) {
  /* A */
}
else {
  /* B */
}
```

```
while (x) {
}
```

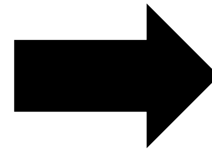
```
for (i = 0; i < n ; i++) {
}
```

How to represent in assembly?

Control Flows in Assembly (1)

- There are only “if” and “goto” (no “else”)

```
if ( x ) {
   /* A */
}
else {
   /* B */
}
```



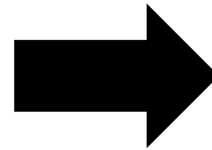
How assembly looks like…

```
   if (!x) goto F;
   /* A */
   goto DONE;
F:
   /* B */
DONE:
```

Control Flows in Assembly (2)

- There are only “if” and “goto” (no “else”)

```
while (x) {
    /* body */
}
```



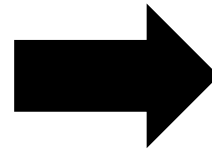
How assembly looks like…

```
WHILE:
   if (!x) goto DONE;
   /* body */
   goto WHILE;
DONE:
```

Control Flows in Assembly (3)

- There are only “if” and “goto” (no “else”)

```
for (i = 0; i < n ; i++) {
  /* body */
}
```

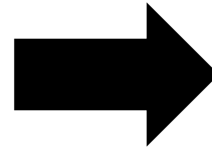


How assembly looks like…

```
   i = 0;
LOOP:
   if (i >= n) goto DONE;
   /* body */
   i++;
   goto LOOP;
DONE:
```

Control Flows in Assembly (Example)


```
if (!x) goto F;
```

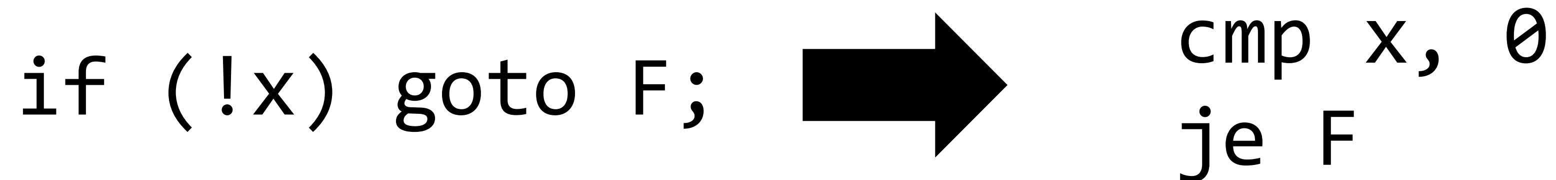


Test if x is zero

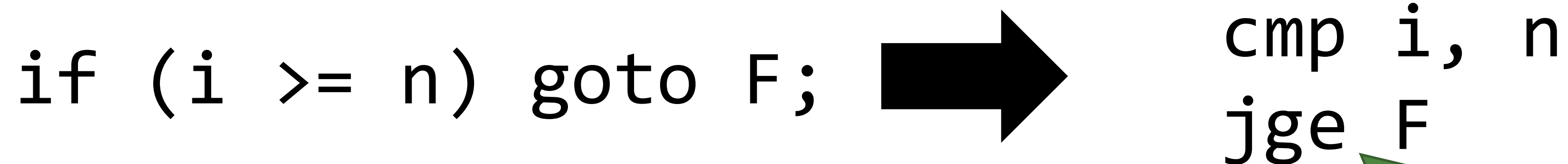
```
cmp x, 0
je F
```

If x=zero then goto F

Control Flows in Assembly (Example)



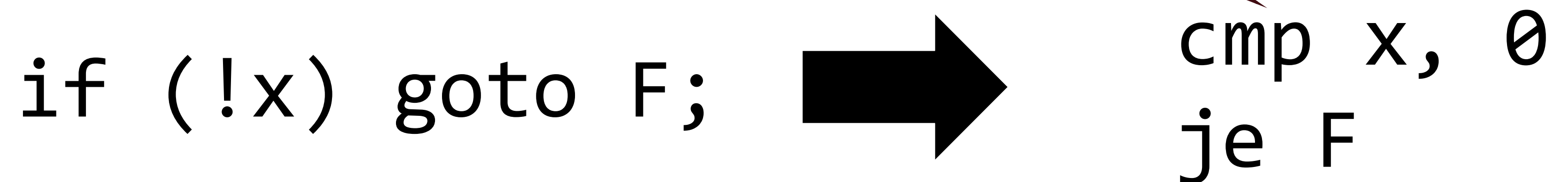
Test if `i >= n`

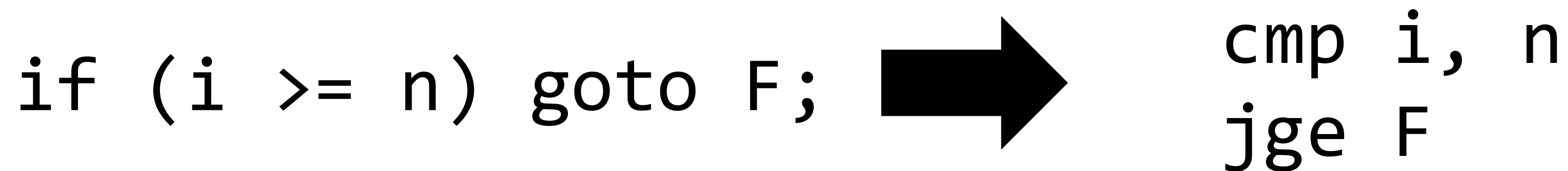


If `i >= n` then goto F

Control Flows in Assembly (Example)

Where do we store the result of comparison (cmp)?





RFLAGS: Storing the Processor State

- RFLAGS is a status ***register*** used in x86-64, which is essentially a collection of status flag bits
  - In X86, it is called EFLAGS

- There are approximately 20 different flag bits used in x86-64, but we are mainly interested in 6 condition flags:
  - OF: Overflow flag
  - SF: Sign flag
  - ZF: Zero flag
  - AF: Auxiliary carry flag
  - PF: Parity flag
  - CF: Carry flag

63–32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

`cmp` Only Affects RFLAGS

- `cmp` is the same as `sub`, except that it only affects RFLAGS, but not the destination operand. For example, `cmp rax, rbx` will not change the `rax` register

Branch Instructions

Assume that a comparison instruction (`cmp`) precedes the branch instruction

Branch Instruction	Condition	Description
`ja`	`CF = 0 and ZF = 0`	Jump if above
`jb`	`CF = 1`	Jump if below
`je`	`ZF = 1`	Jump if equal
`jl`	`SF ≠ F`	Jump if less
`jle`	`ZF = 1 or SF ≠ F`	Jump if less or equal
`jna`	`CF = 1 or ZF = 1`	Jump if not above
`jnb`	`CF = 0`	Jump if not below
`jz`	`ZF = 1`	Jump if zero (same as `je`)
… (many more)		

Examining the `ja` Case

```
cmp rax, rbx
ja label              ; jump to label if rax > rbx
```

- `cmp` is the same as `sub` except that it only updates `RFLAGS`.
- `CF = 0` implies that `rax - rbx` did not produce any carry.
- `ZF = 0` implies that the result of subtraction is not zero. Hence, `rax ≠ rbx`.
- From both the conditions, we can say that `rax > rbx`.

Summary

- We learned how to move around data
  - `mov, lea, push, pop, etc.`
- We learned how to perform arithmetic and logical operations
  - `add, sub, and, or, etc.`
- We also learned how to control program flows
  - `cmp, jmp, ja, jz, etc.`

Note

- Refer to the cheat sheet below
  - https://websec-lab.github.io/courses/2026s-cse251/metarials/x86-64-ref-sheet.pdf

- This sheet will be provided during the final exam

- Although the textbook covers AT&T syntax, this class uses Intel syntax.

Question?