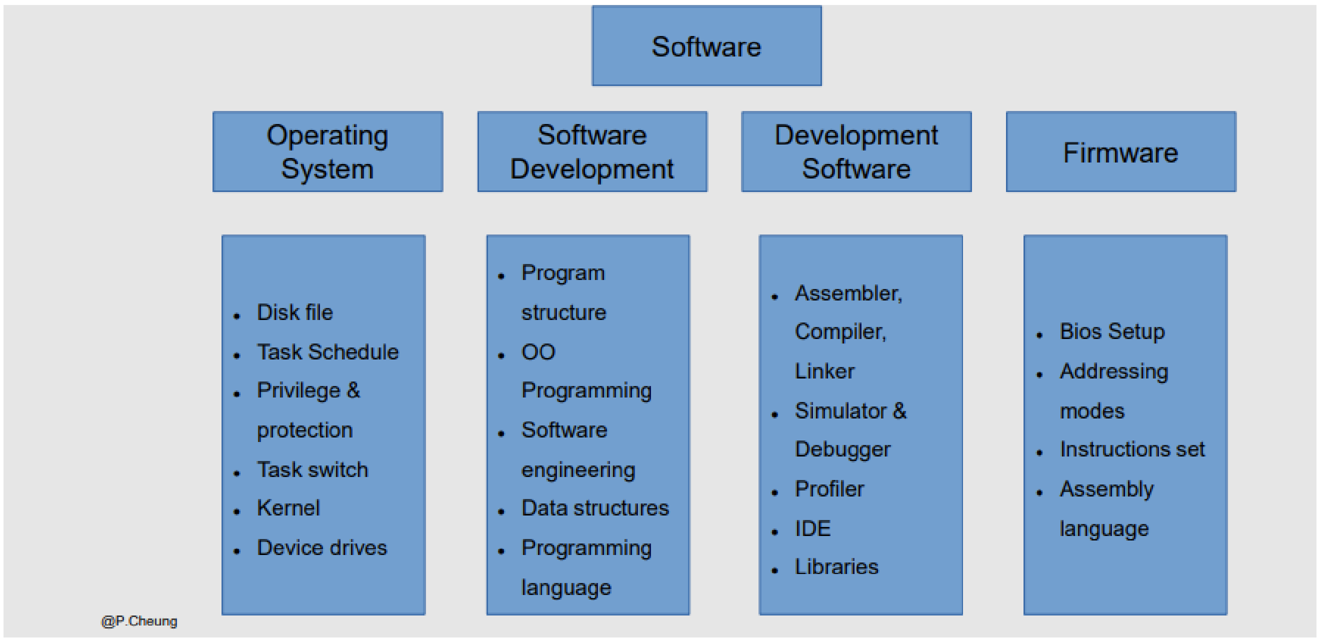
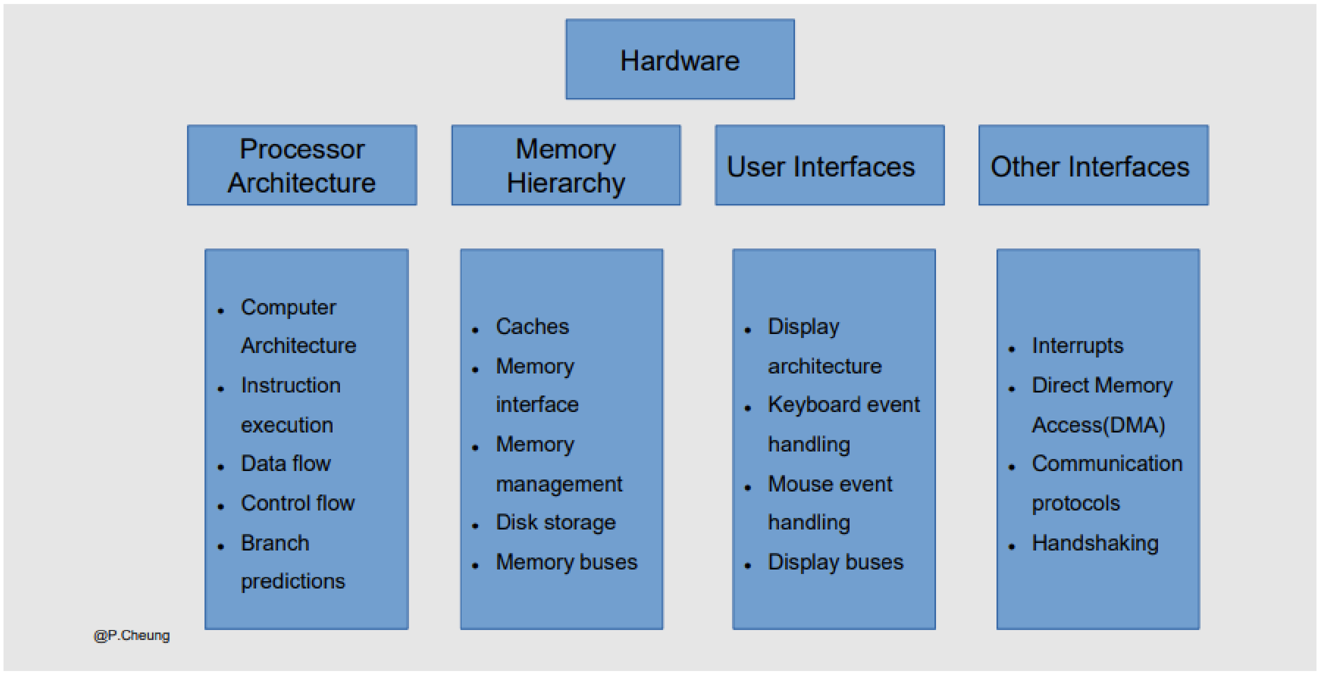
Machine-Level Programming I, Introduction to Assembly

Machine Level representation of Programs

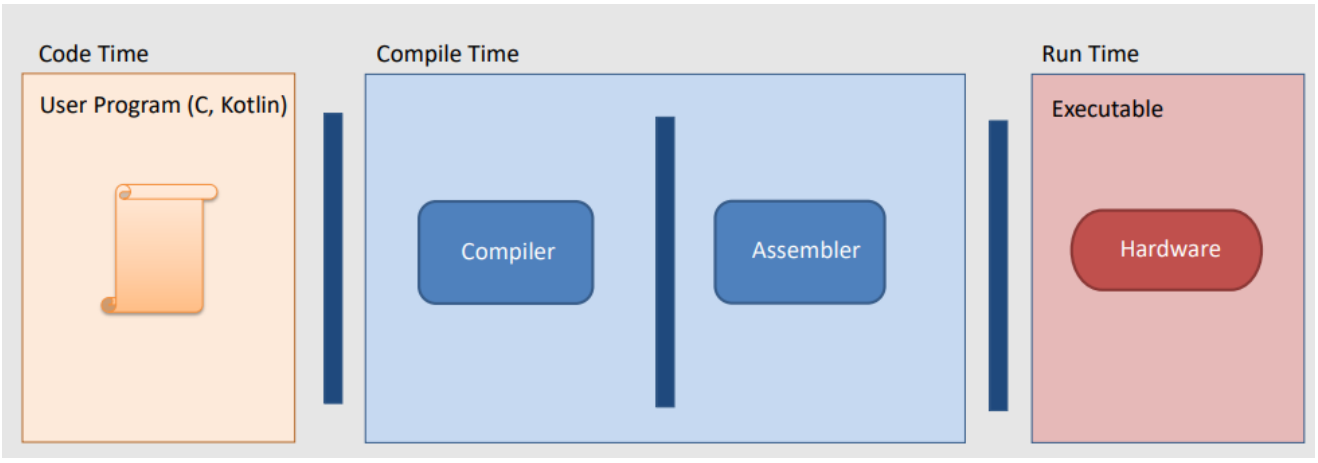
Software Interface



Hardware Interface

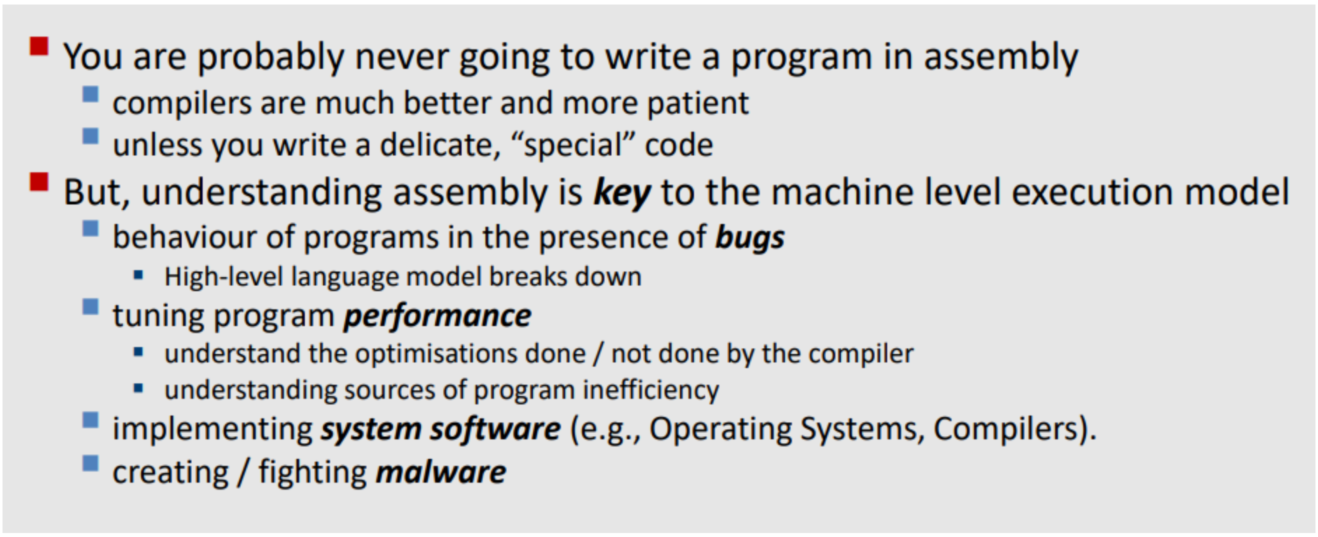


Code / Compiler / Run time



Note: The compiler and assembler are just programs (developed using the same principle)

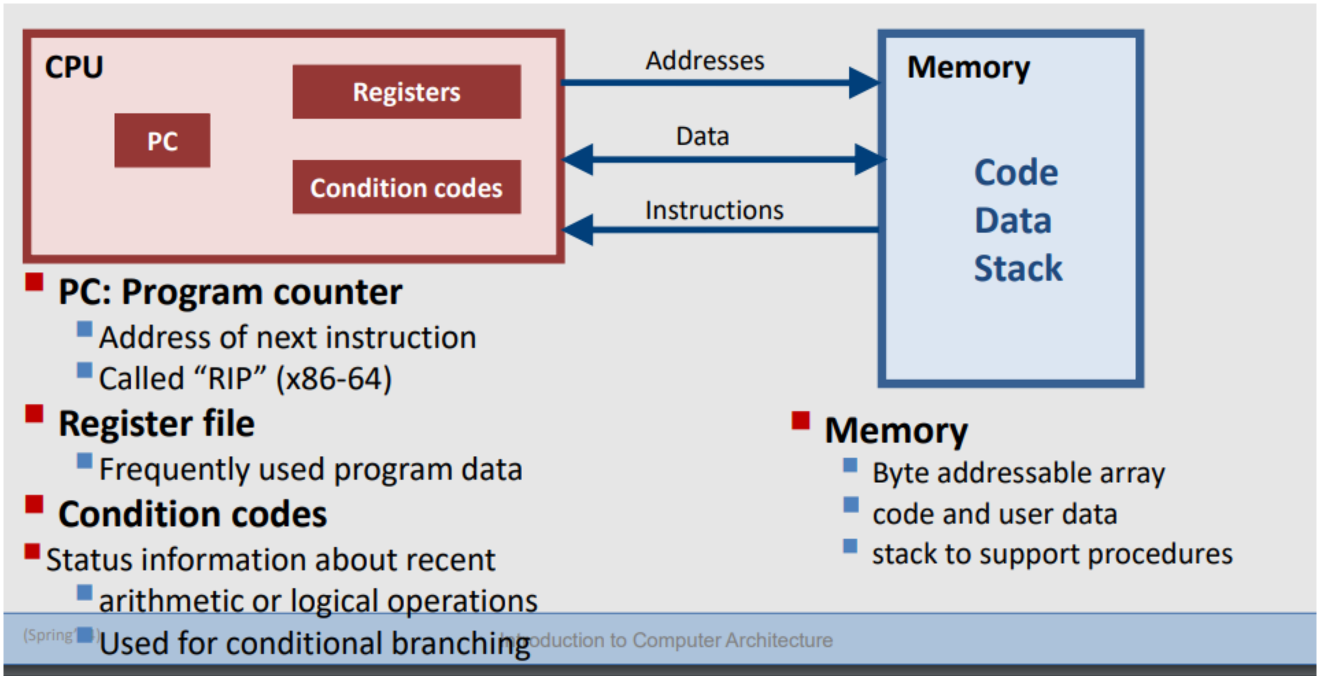
Why write Assembly Code?



- You are probably never going to write a program in assembly
  - compilers are much better and more patient
  - unless you write a delicate, “special” code
- But, understanding assembly is ***key*** to the machine level execution model
  - behaviour of programs in the presence of ***bugs***
    - High-level language model breaks down
  - tuning program ***performance***
    - understand the optimisations done / not done by the compiler
    - understanding sources of program inefficiency
  - implementing ***system software*** (e.g., Operating Systems, Compilers).
  - creating / fighting ***malware***

Program Encodings: The assembler language in context

Assembly/Machine code view

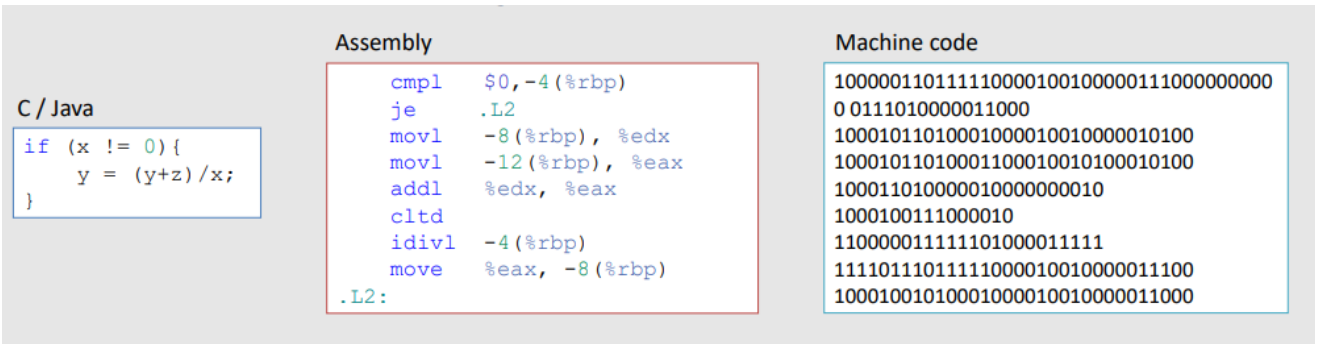


- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Frequently used program data
- **Condition codes**
- Status information about recent
  - arithmetic or logical operations
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - code and user data
  - stack to support procedures

(Spring'1 Introduction to Computer Architecture

16 registers in this target machine

C/Java, assembly, and machine code



C / Java

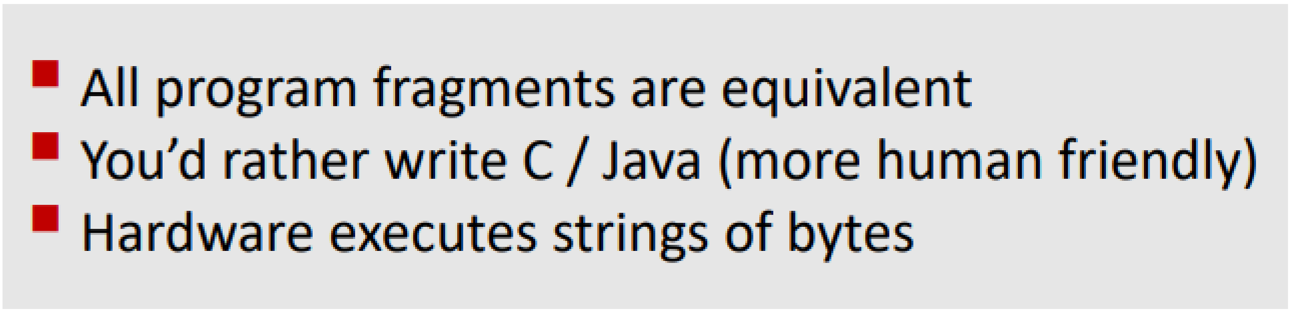
```
if (x != 0){
     y = (y+z)/x;
}
```

Assembly

```
    cmpl    $0,-4(%rbp)
    je      .L2
    movl    -8(%rbp), %edx
    movl    -12(%rbp), %eax
    addl    %edx, %eax
    cltd
    idivl   -4(%rbp)
    move    %eax, -8(%rbp)
.L2:
```

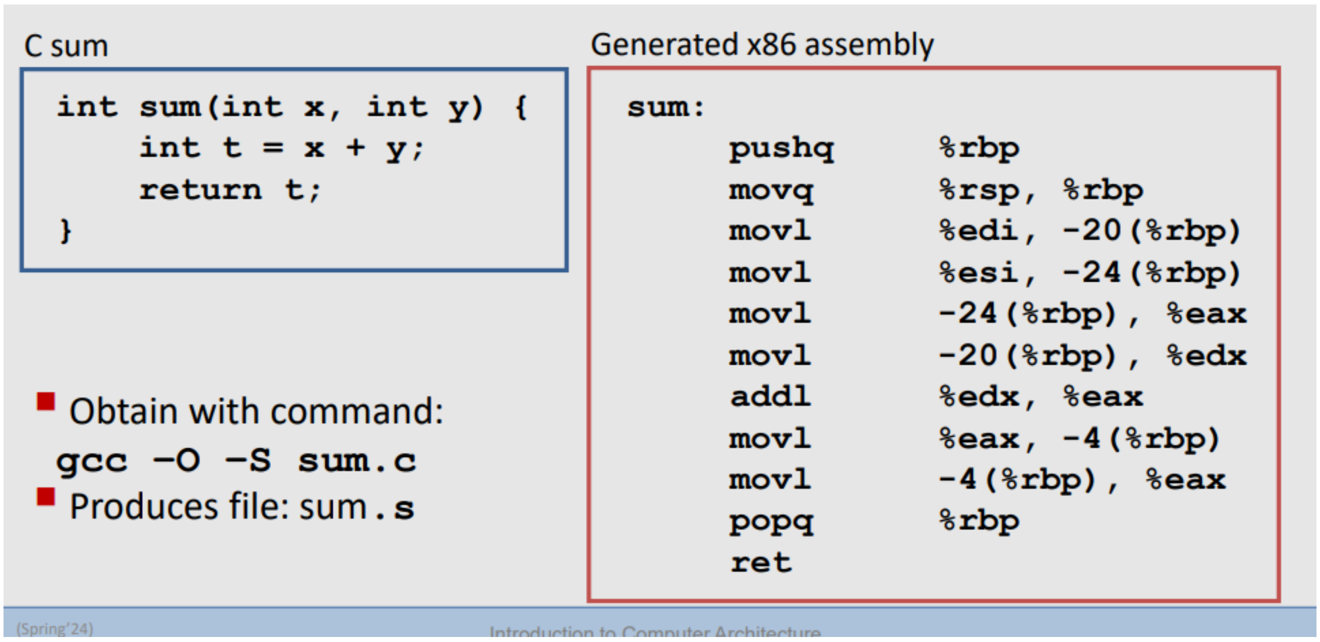
Machine code

```
1000001101111110000100100000111000000000
0 0111010000011000
100010110100010000100100000010100
100010110100011000100101000010100
1000110100000100000000010
1000100111000010
1100000111111010000011111
111101110111111000010010000011100
100010010100010000100100000011000
```



- All program fragments are equivalent
- You’d rather write C / Java (more human friendly)
- Hardware executes strings of bytes

Compiling into Assembly



C sum

```
int sum(int x, int y) {
    int t = x + y;
    return t;
}
```

- Obtain with command:

```
gcc -O -S sum.c
```

- Produces file: `sum.s`

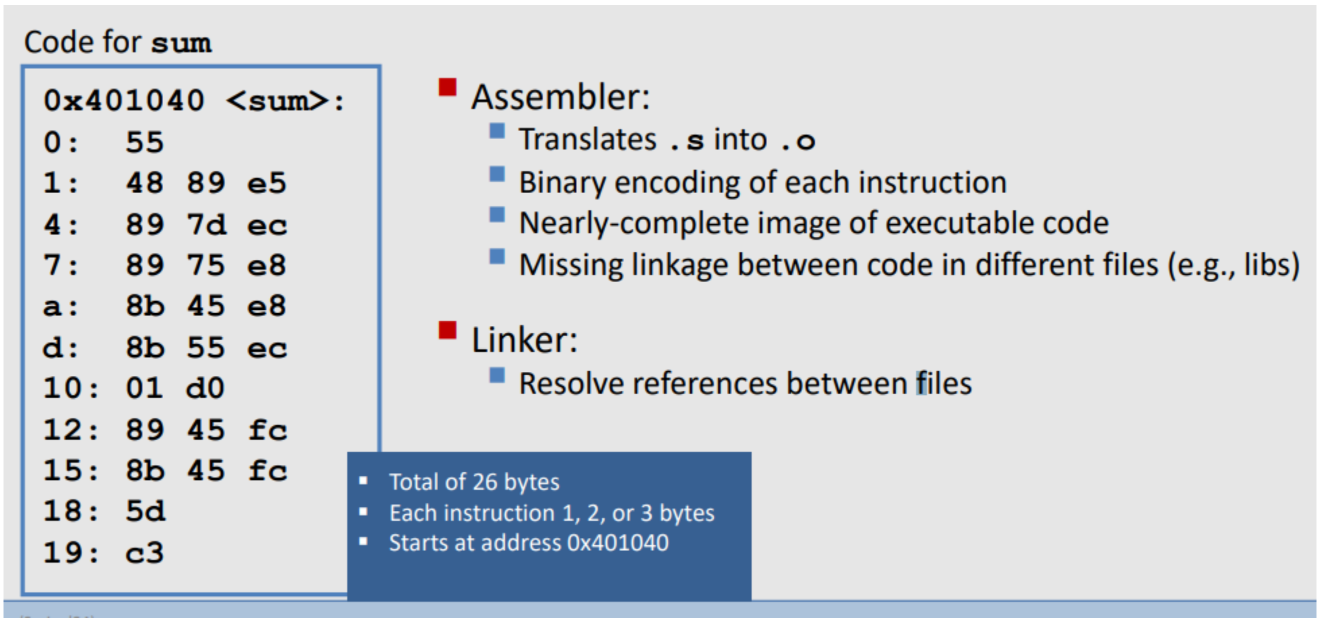
Generated x86 assembly

```
sum:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    -24(%rbp), %eax
        movl    -20(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

(Spring'24) Introduction to Computer Architecture

-S: In the current directory, `sum.s` is created

Object code



Code for **sum**

```
0x401040 <sum>:
0:   55
1:   48 89 e5
4:   89 7d ec
7:   89 75 e8
a:   8b 45 e8
d:   8b 55 ec
10:  01 d0
12:  89 45 fc
15:  8b 45 fc
18:  5d
19:  c3
```

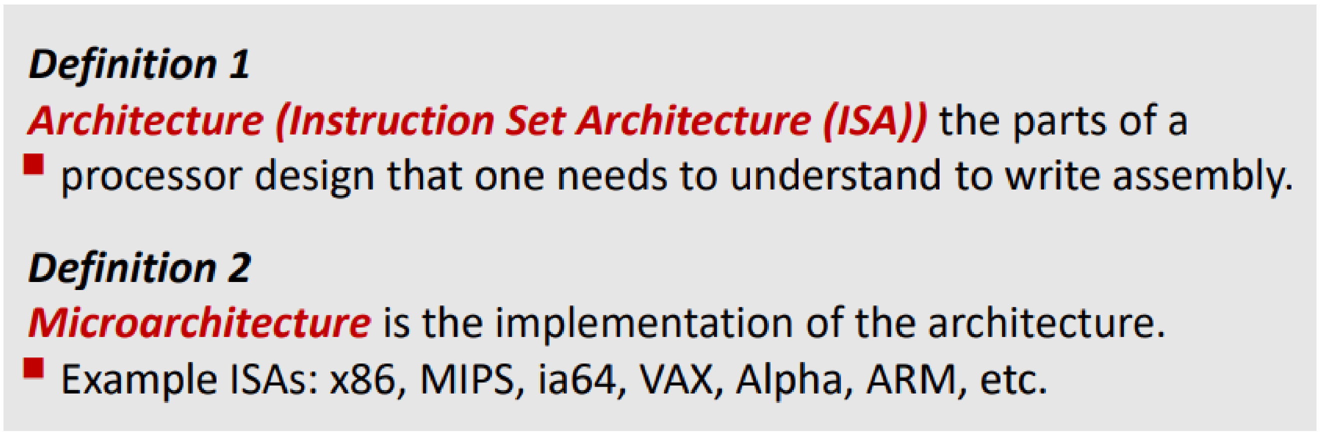
- Total of 26 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

- Assembler:
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkage between code in different files (e.g., libs)
- Linker:
  - Resolve references between files

Code stored at memory location `0x401040`

Recap on ISA

Instruction Set Architecture (ISA)



Definition 1

Architecture (Instruction Set Architecture (ISA)) the parts of a

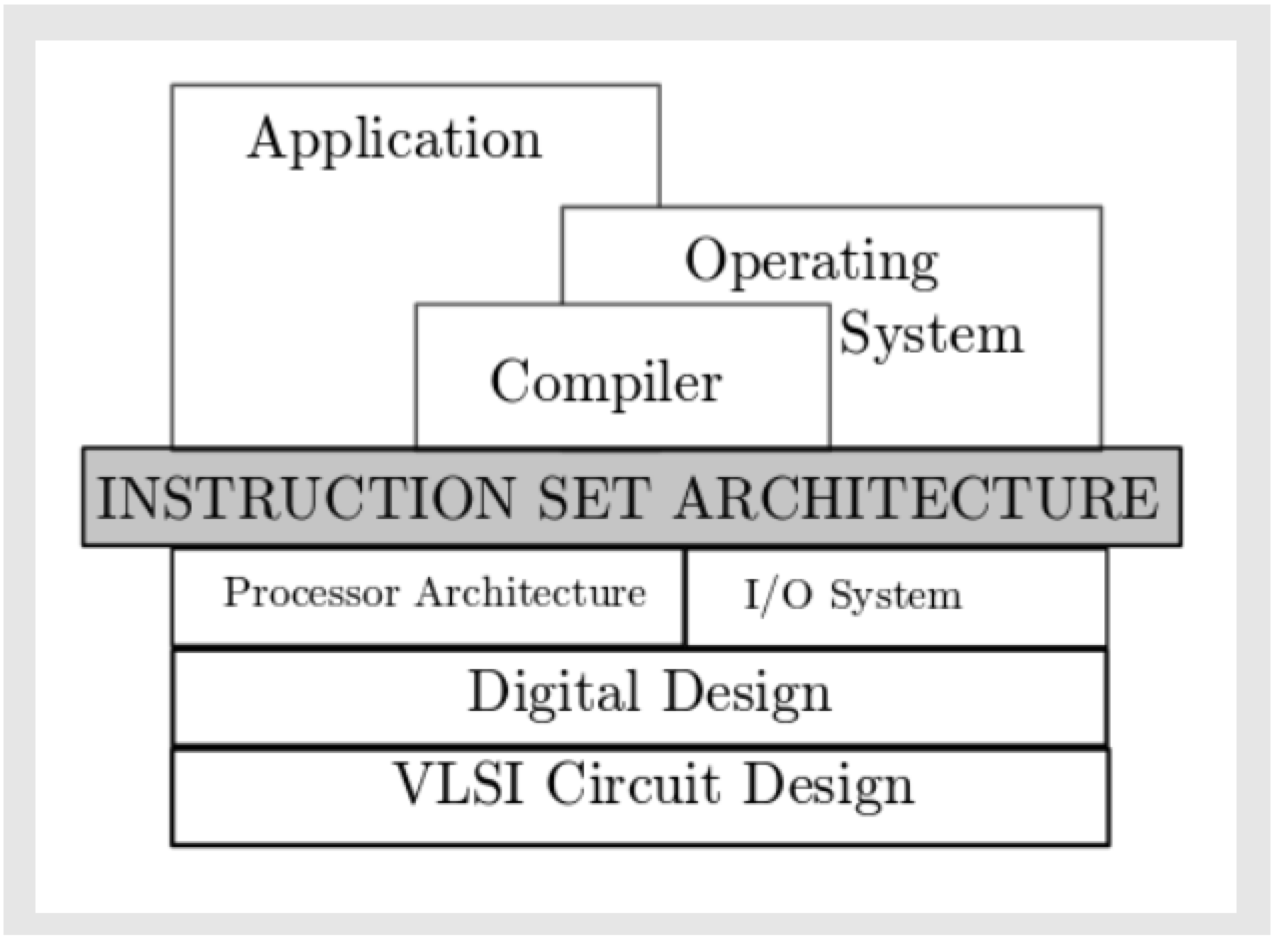
- processor design that one needs to understand to write assembly.

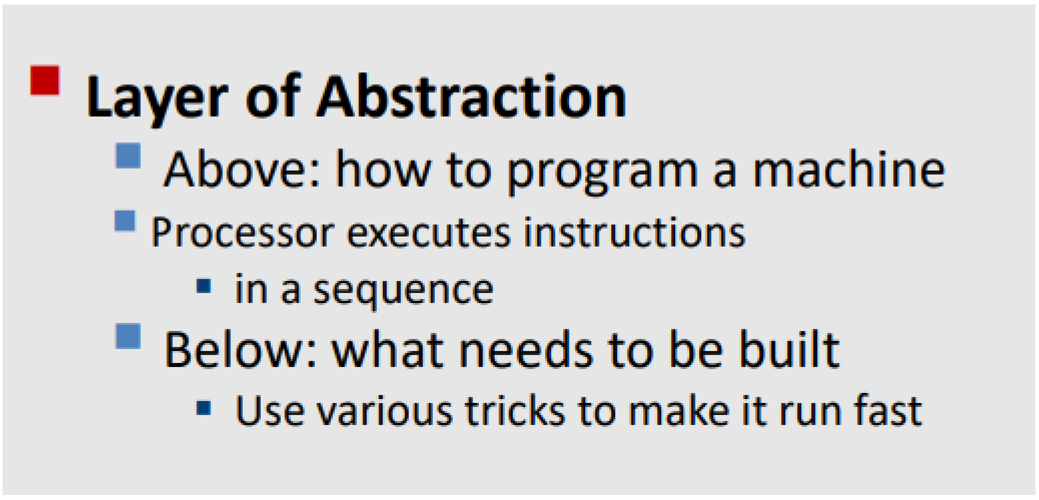
Definition 2

Microarchitecture is the implementation of the architecture.

- Example ISAs: x86, MIPS, ia64, VAX, Alpha, ARM, etc.

Instruction Set Architecture – big picture

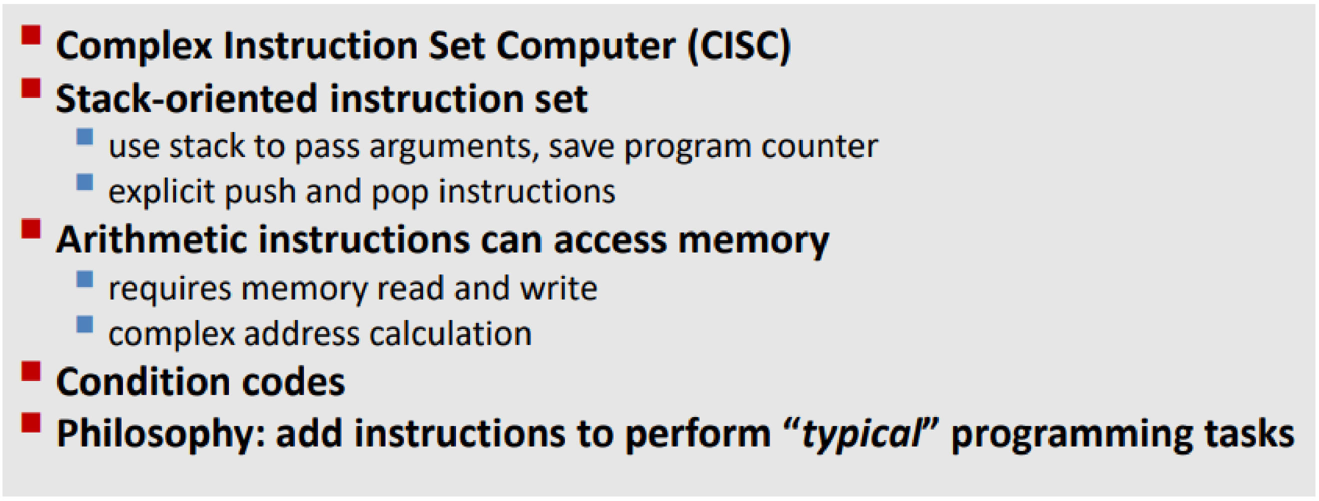




- Layer of Abstraction
  - Above: how to program a machine
  - Processor executes instructions
    - in a sequence
  - Below: what needs to be built
    - Use various tricks to make it run fast

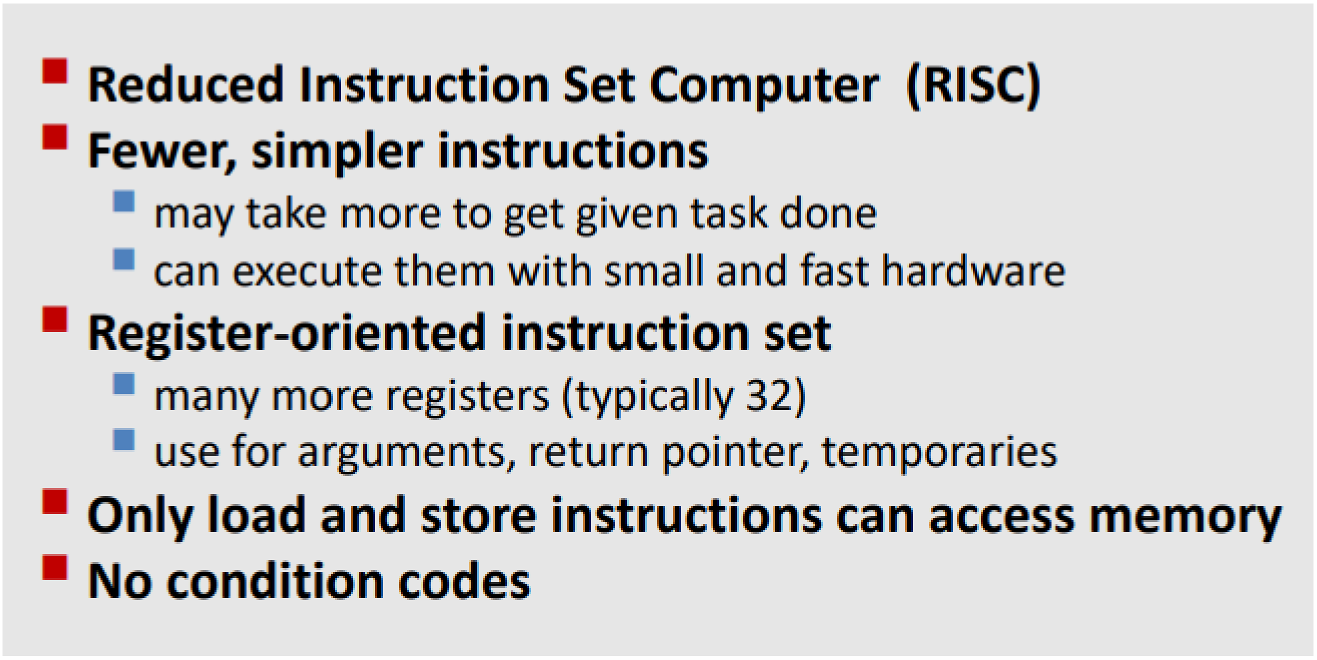
Allows the software to communicate with the hardware

CISC Instruction Sets



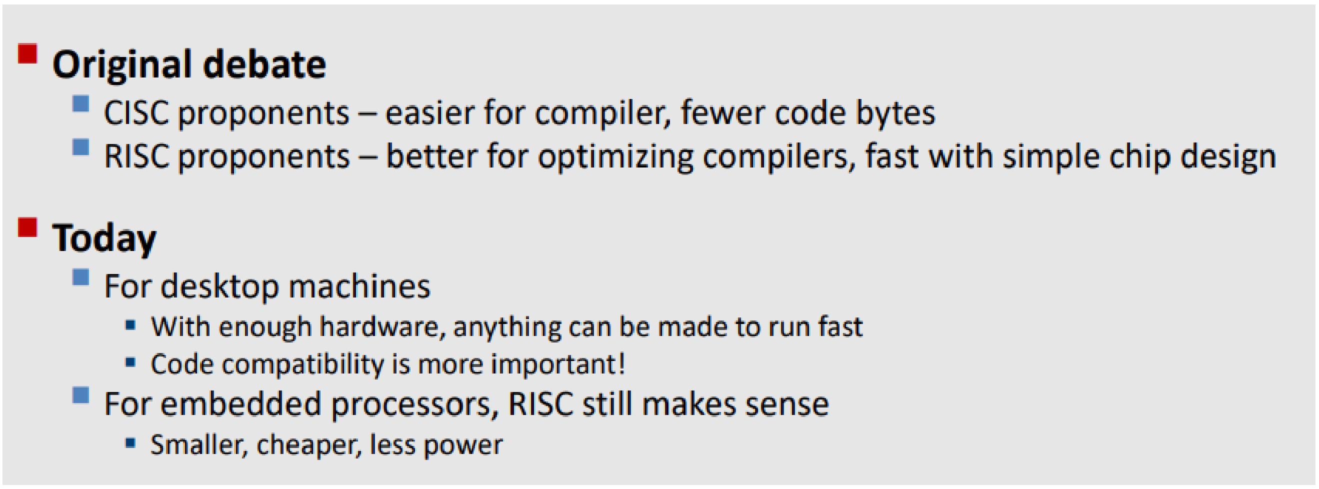
- **Complex Instruction Set Computer (CISC)**
- **Stack-oriented instruction set**
  - use stack to pass arguments, save program counter
  - explicit push and pop instructions
- **Arithmetic instructions can access memory**
  - requires memory read and write
  - complex address calculation
- **Condition codes**
- **Philosophy: add instructions to perform "*typical*" programming tasks**

RISC Instruction Sets



- Reduced Instruction Set Computer (RISC)
- Fewer, simpler instructions
  - may take more to get given task done
  - can execute them with small and fast hardware
- Register-oriented instruction set
  - many more registers (typically 32)
  - use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
- No condition codes

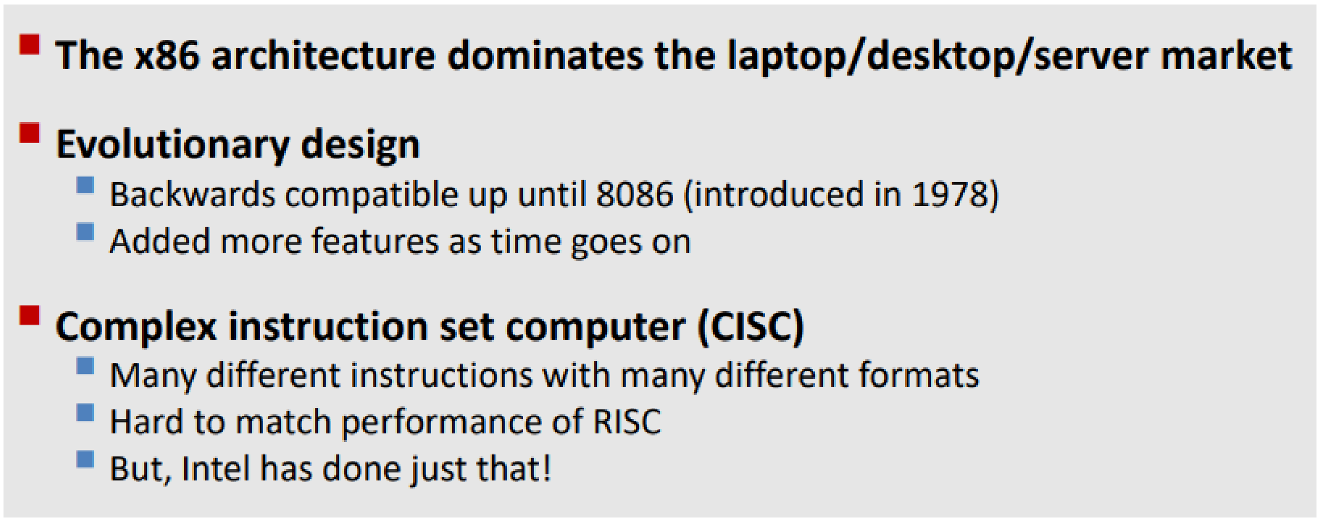
CISC vs. RISC



- Original debate
  - CISC proponents – easier for compiler, fewer code bytes
  - RISC proponents – better for optimizing compilers, fast with simple chip design
- Today
  - For desktop machines
    - With enough hardware, anything can be made to run fast
    - Code compatibility is more important!
  - For embedded processors, RISC still makes sense
    - Smaller, cheaper, less power

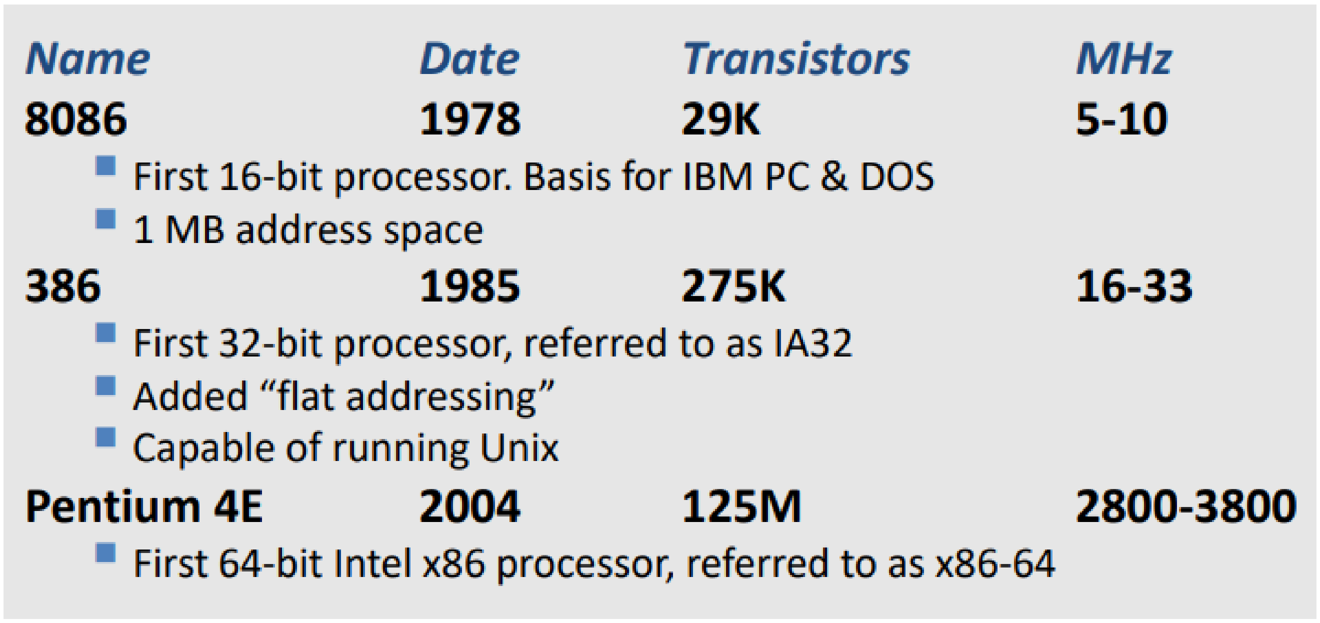
History of x86

Intel x86 Processors



- **The x86 architecture dominates the laptop/desktop/server market**
- **Evolutionary design**
  - Backwards compatible up until 8086 (introduced in 1978)
  - Added more features as time goes on
- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
  - Hard to match performance of RISC
  - But, Intel has done just that!

Intel x86 Evolution: Key Milestones



Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**

- First 16-bit processor. Basis for IBM PC & DOS
- 1 MB address space

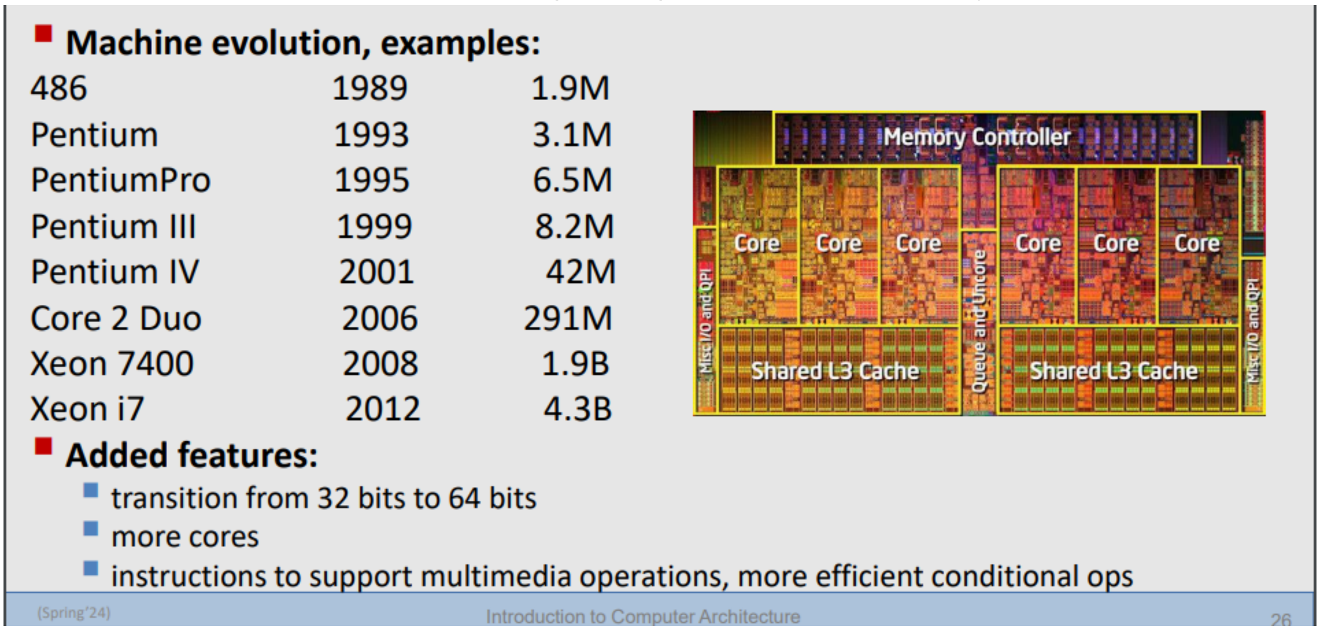
Name	*Date*	*Transistors*	*MHz*
386	**1985**	**275K**	**16-33**

- First 32-bit processor, referred to as IA32
- Added “flat addressing”
- Capable of running Unix

Name	*Date*	*Transistors*	*MHz*
Pentium 4E	**2004**	**125M**	**2800-3800**

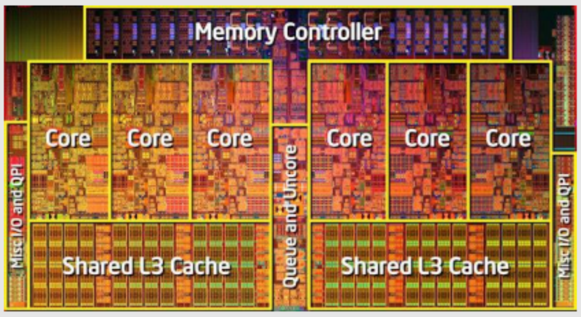
- First 64-bit Intel x86 processor, referred to as x86-64

Intel x86 Processors: Overview



- **Machine evolution, examples:**

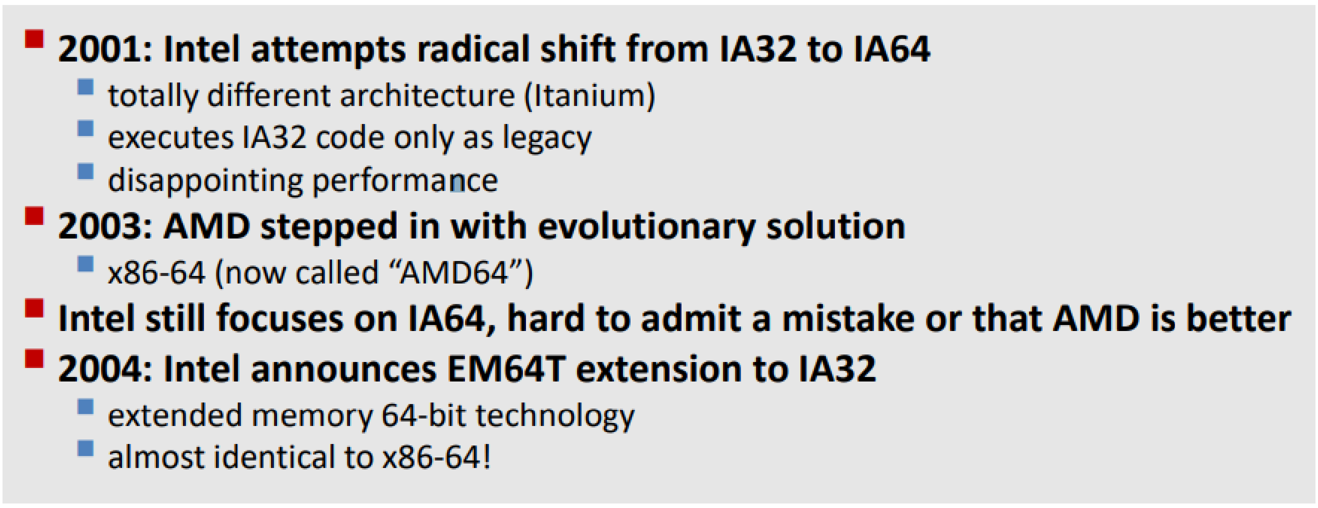
486	1989	1.9M
Pentium	1993	3.1M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium IV	2001	42M
Core 2 Duo	2006	291M
Xeon 7400	2008	1.9B
Xeon i7	2012	4.3B



- **Added features:**
  - transition from 32 bits to 64 bits
  - more cores
  - instructions to support multimedia operations, more efficient conditional ops

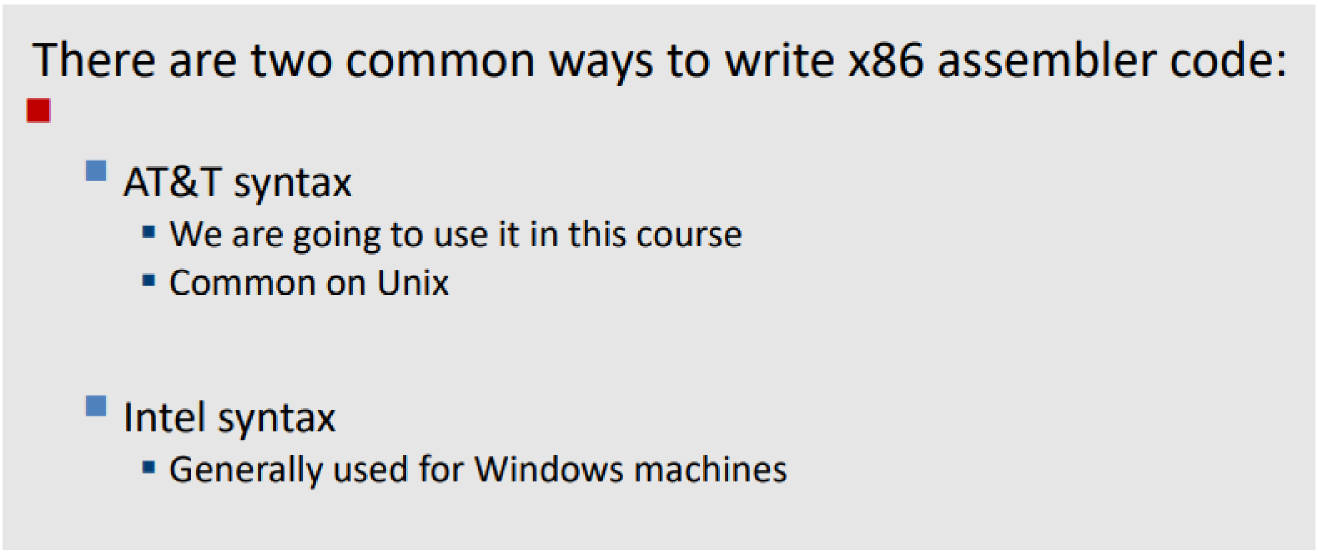
(Spring'24) Introduction to Computer Architecture 26

Intel's 64-Bit History



- **2001: Intel attempts radical shift from IA32 to IA64**
  - totally different architecture (Itanium)
  - executes IA32 code only as legacy
  - disappointing performance
- **2003: AMD stepped in with evolutionary solution**
  - x86-64 (now called "AMD64")
- **Intel still focuses on IA64, hard to admit a mistake or that AMD is better**
- **2004: Intel announces EM64T extension to IA32**
  - extended memory 64-bit technology
  - almost identical to x86-64!

A quick note on syntax

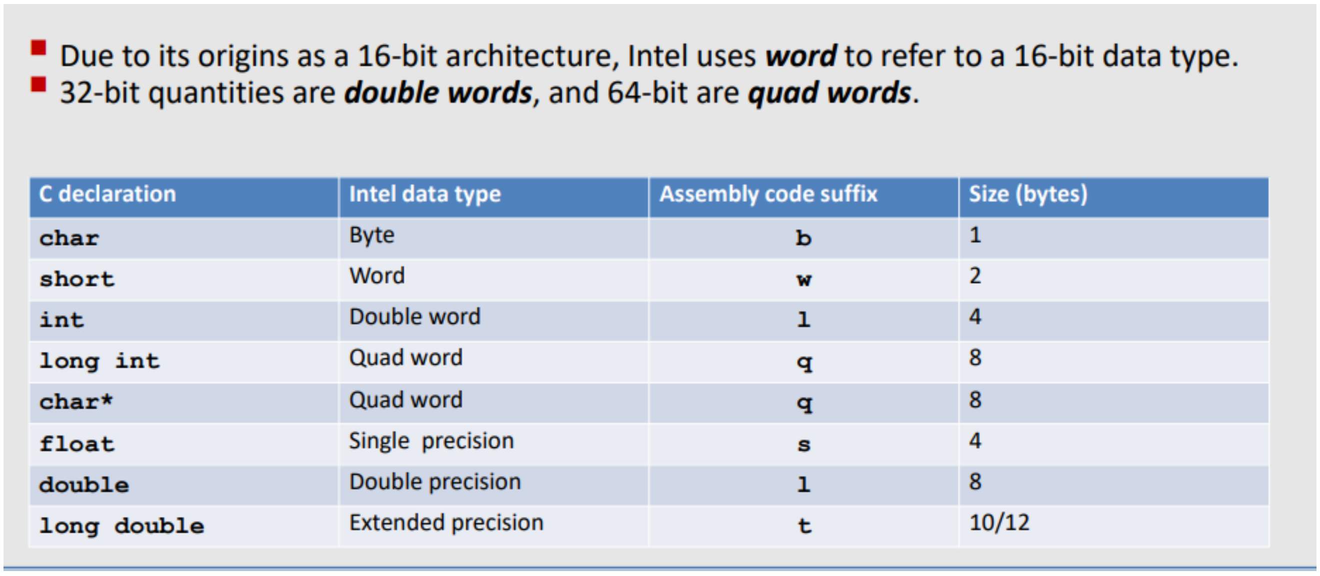


There are two common ways to write x86 assembler code:

-
  - AT&T syntax
    - We are going to use it in this course
    - Common on Unix
  - Intel syntax
    - Generally used for Windows machines

Instruction format

C Data Types in a x86-64

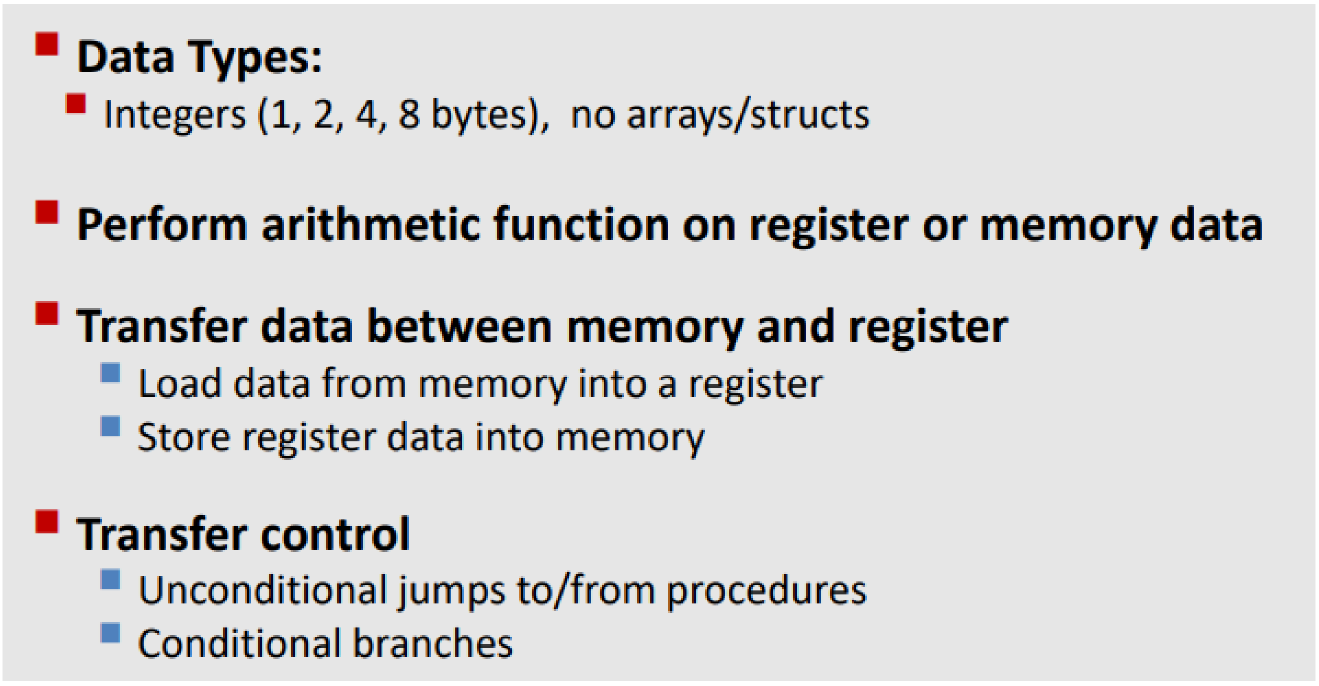


- Due to its origins as a 16-bit architecture, Intel uses ***word*** to refer to a 16-bit data type.
- 32-bit quantities are ***double words***, and 64-bit are ***quad words***.

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Quad word	`q`	8
`char*`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

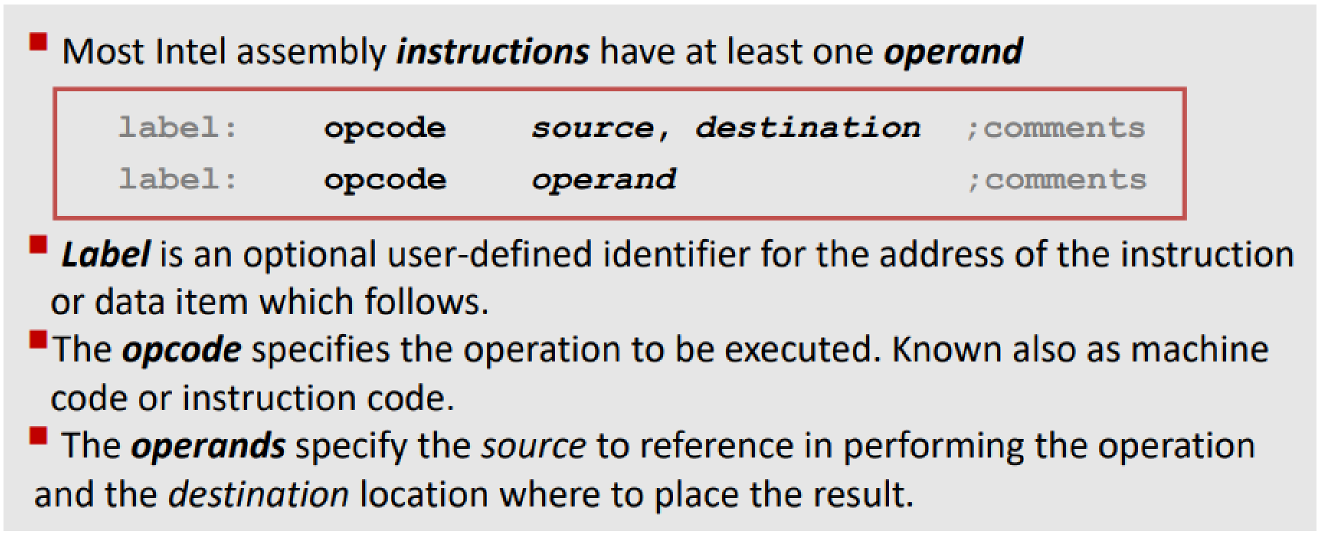
We will only look at **the first 5**.

Assembly characteristics: Data Types and Instructions



- **Data Types:**
  - Integers (1, 2, 4, 8 bytes), no arrays/structs
- **Perform arithmetic function on register or memory data**
- **Transfer data between memory and register**
  - Load data from memory into a register
  - Store register data into memory
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Instruction format



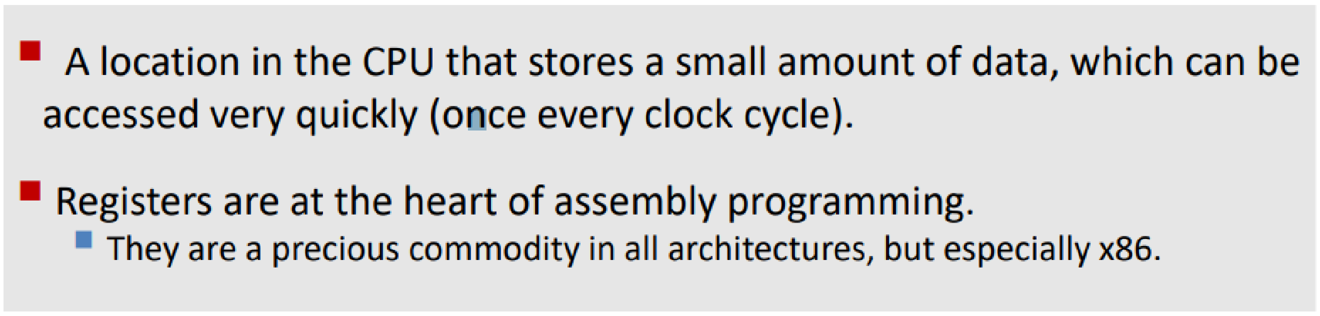
- Most Intel assembly ***instructions*** have at least one ***operand***

```
label:     opcode     source, destination     ;comments
label:     opcode     operand                 ;comments
```

- ***Label*** is an optional user-defined identifier for the address of the instruction or data item which follows.
- The ***opcode*** specifies the operation to be executed. Known also as machine code or instruction code.
- The ***operands*** specify the *source* to reference in performing the operation and the *destination* location where to place the result.

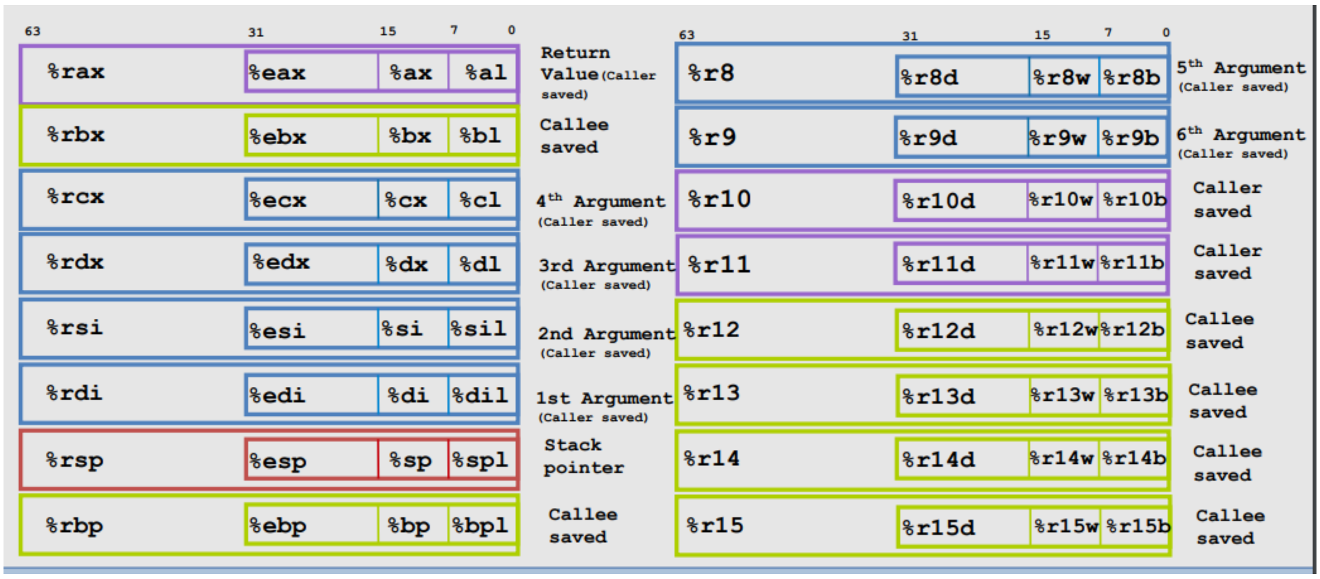
Introduction to assembly: Registers, operands

What is a register?



- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle).
- Registers are at the heart of assembly programming.
  - They are a precious commodity in all architectures, but especially x86.

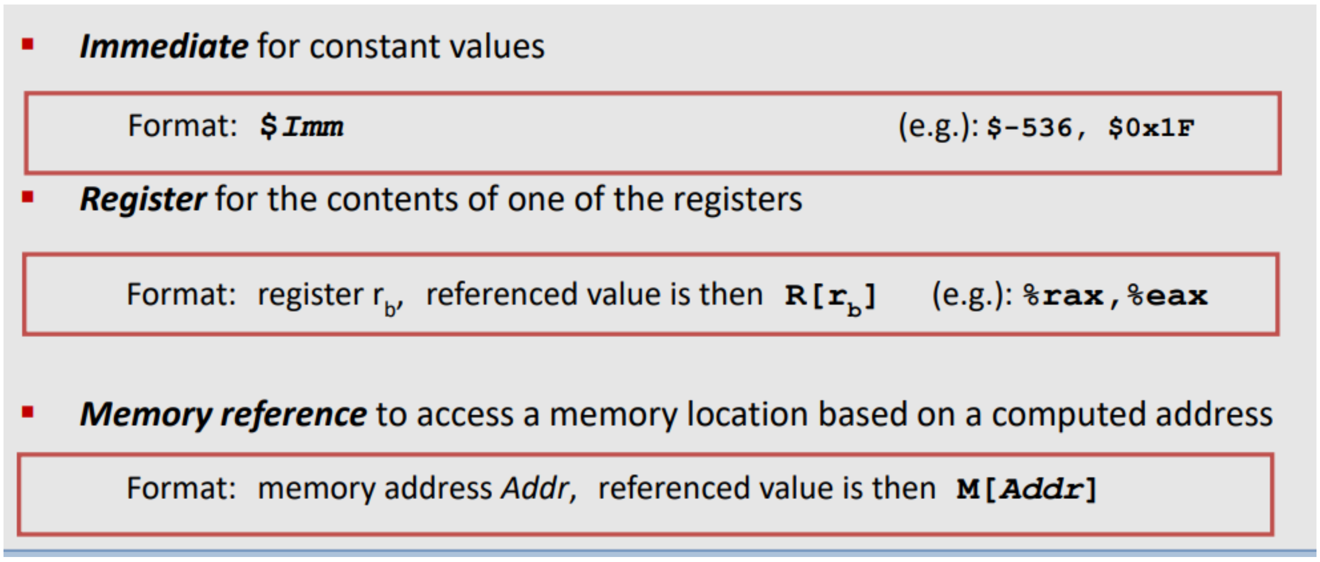
x86-64 Integer Registers



63	31	15	7	0	
%rax	%eax	%ax	%al		Return Value (Caller saved)
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th Argument (Caller saved)
%rdx	%edx	%dx	%dl		3rd Argument (Caller saved)
%rsi	%esi	%si	%sil		2nd Argument (Caller saved)
%rdi	%edi	%di	%dil		1st Argument (Caller saved)
%rsp	%esp	%sp	%spl		Stack pointer
%rbp	%ebp	%bp	%bpl		Callee saved

63	31	15	7	0	
%r8	%r8d	%r8w	%r8b		5th Argument (Caller saved)
%r9	%r9d	%r9w	%r9b		6th Argument (Caller saved)
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

Operand Types



- ***Immediate*** for constant values

  Format: `$Imm` (e.g.): `$-536, $0x1F`

- ***Register*** for the contents of one of the registers

  Format: register r_b, referenced value is then $R[r_b]$ (e.g.): `%rax,%eax`

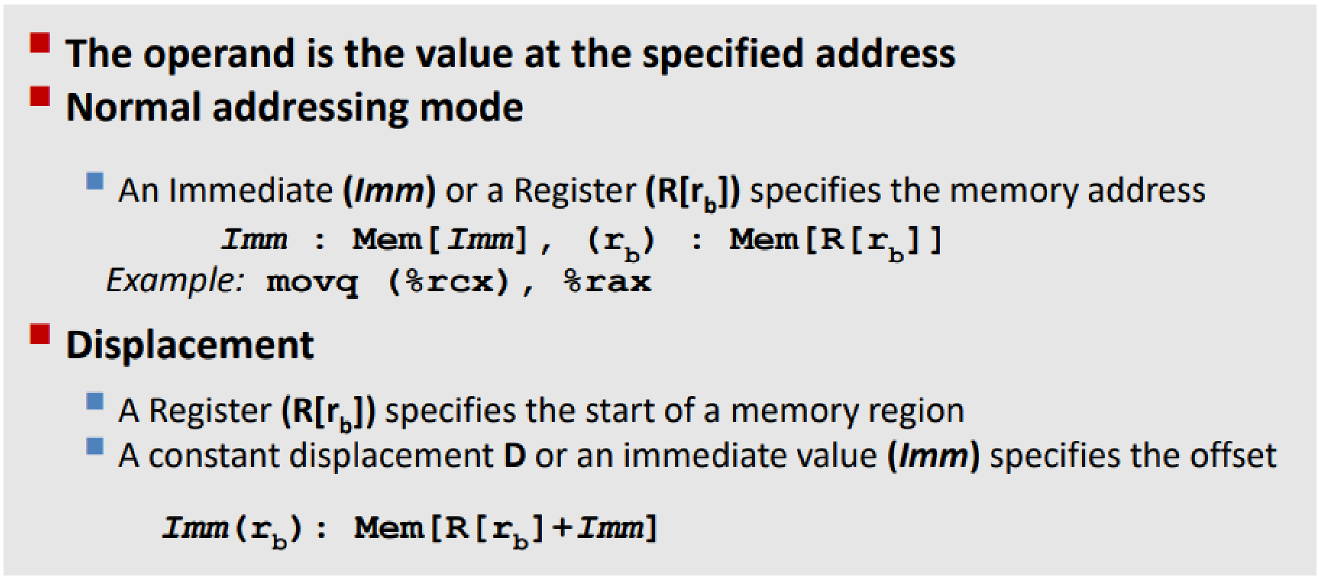
- ***Memory reference*** to access a memory location based on a computed address

  Format: memory address *Addr*, referenced value is then `M[`*`Addr`*`]`

Memory addressing modes and data move instructions

Can only access memory with 64 bits

Memory operands: Simple memory addressing modes



- **The operand is the value at the specified address**
- **Normal addressing mode**
  - An Immediate **(*Imm*)** or a Register **($R[r_b]$)** specifies the memory address

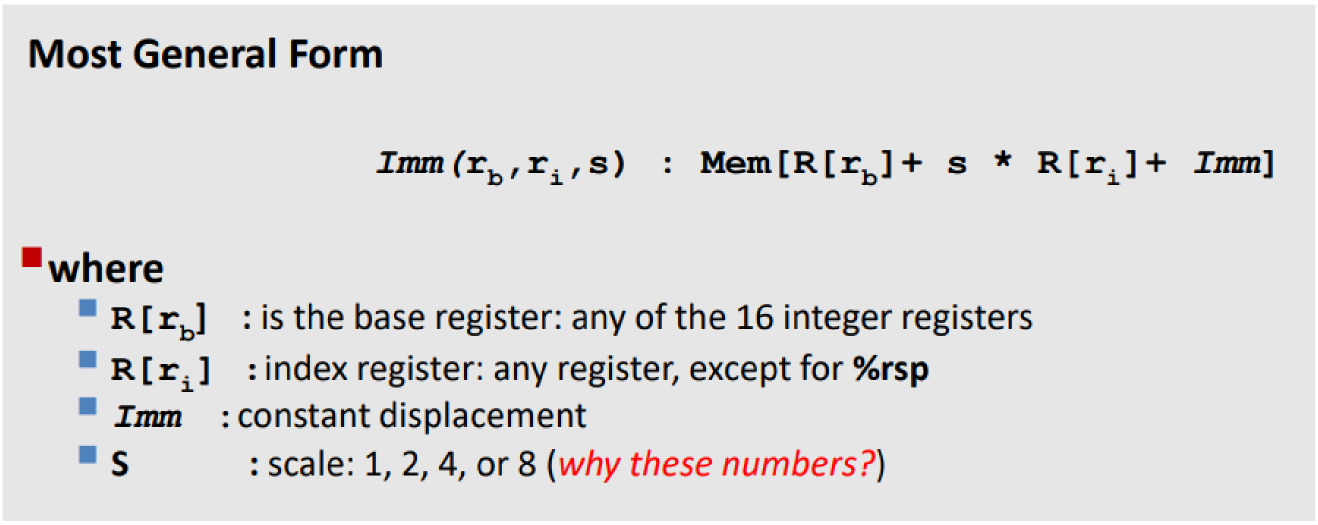
    Imm : $Mem[Imm]$, (r_b) : $Mem[R[r_b]]$

    Example: `movq (%rcx), %rax`
- **Displacement**
  - A Register **($R[r_b]$)** specifies the start of a memory region
  - A constant displacement **D** or an immediate value **(*Imm*)** specifies the offset

    $Imm(r_b)$: $Mem[R[r_b]+Imm]$

$Imm : Mem[Imm] = Of\ Type\ Memory(Absolute)$

Memory operands: Complete memory addressing modes



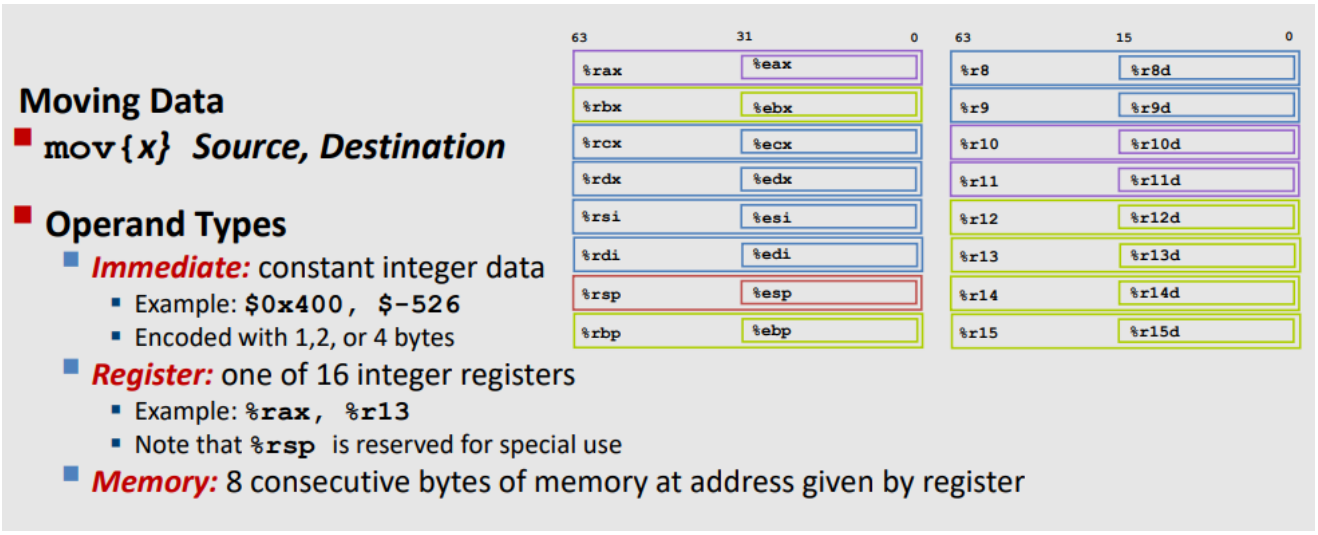
Most General Form

$$Imm(r_b, r_i, s) \quad : \quad Mem[R[r_b] + s * R[r_i] + Imm]$$

- where
  - $R[r_b]$: is the base register: any of the 16 integer registers
  - $R[r_i]$: index register: any register, except for **%rsp**
  - *Imm* : constant displacement
  - S : scale: 1, 2, 4, or 8 (*why these numbers?*)

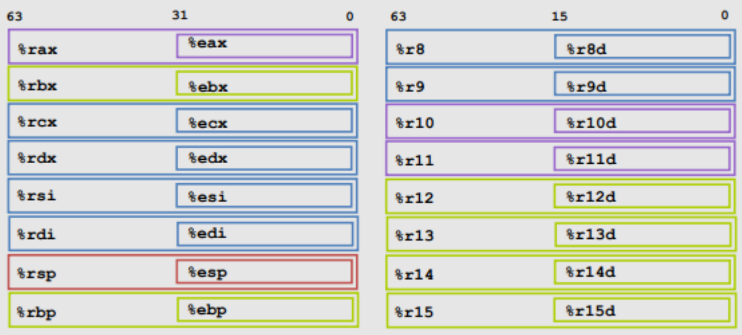
The numbers are the byte sizes of the data types we are using.

Moving data instructions



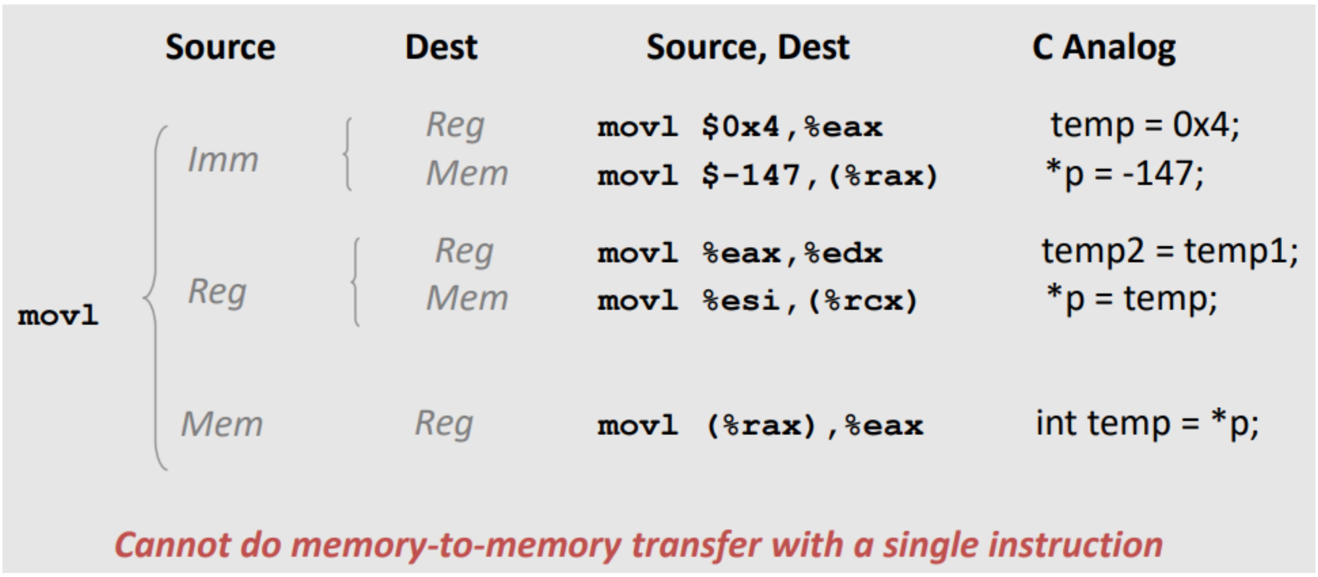
Moving Data

- **mov{*x*} *Source, Destination***
- **Operand Types**
  - ***Immediate:*** constant integer data
    - Example: **$0x400, $-526**
    - Encoded with 1,2, or 4 bytes
  - ***Register:*** one of 16 integer registers
    - Example: **%rax, %r13**
    - Note that **%rsp** is reserved for special use
  - ***Memory:*** 8 consecutive bytes of memory at address given by register



x = suffix = Indicates how many bytes are being copied

`movl` operand combinations



	Source	Dest	Source, Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	temp = 0x4;
		Mem	`movl $-147,(%rax)`	*p = -147;
	Reg	*Reg*	`movl %eax,%edx`	temp2 = temp1;
		Mem	`movl %esi,(%rcx)`	*p = temp;
	Mem	*Reg*	`movl (%rax),%eax`	int temp = *p;

Cannot do memory-to-memory transfer with a single instruction

Memory operands: Special cases memory addressing modes

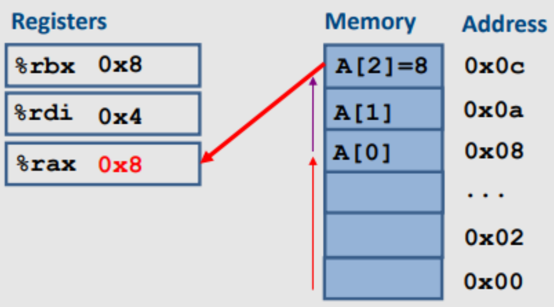
- **Base register + index register**
  - Can be used to access **array elements** when **start of array is dynamically determined**

$$(r_b,r_i) \ : \ \mathrm{Mem}[R[r_b] + R[r_i]]$$

where
$R[r_b]$ = the start of the array, and
$R[r_i]$ = byte index of array element

Example assembly:

```
movw (%rbx,%rdi), %rax
```



w = 2 bytes

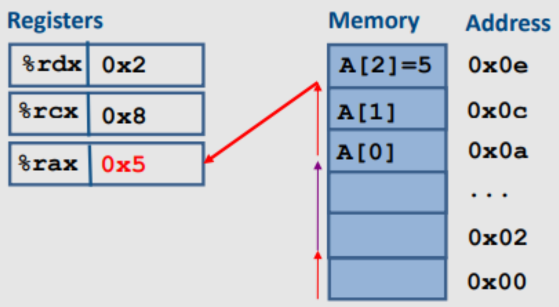
- **Relative base index = base register + index register + displacement**
  - Can be used to access **array of objects, arrays within objects,** and **arrays on the stack**

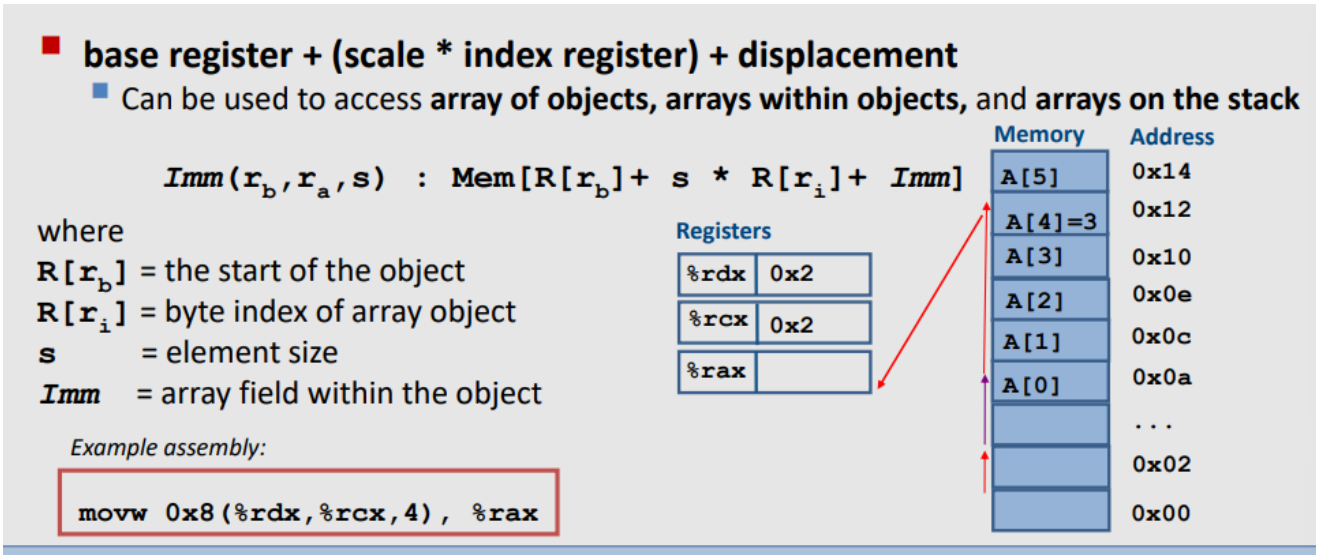
$$Imm(r_b,r_i) \ : \ \mathrm{Mem}[R[r_b] + R[r_i] + Imm]$$

where
$R[r_b]$ = the start of the object
$R[r_i]$ = byte index of array object
Imm = array field within the object

Example assembly:

```
movw 0x4(%rdx,%rcx), %rax
```





- **base register + (scale * index register) + displacement**
  - Can be used to access **array of objects, arrays within objects,** and **arrays on the stack**

$$Imm(r_b, r_a, s) \ : \ \mathrm{Mem}[R[r_b] + s * R[r_i] + Imm]$$

where
$R[r_b]$ = the start of the object
$R[r_i]$ = byte index of array object
s = element size
Imm = array field within the object

Example assembly:

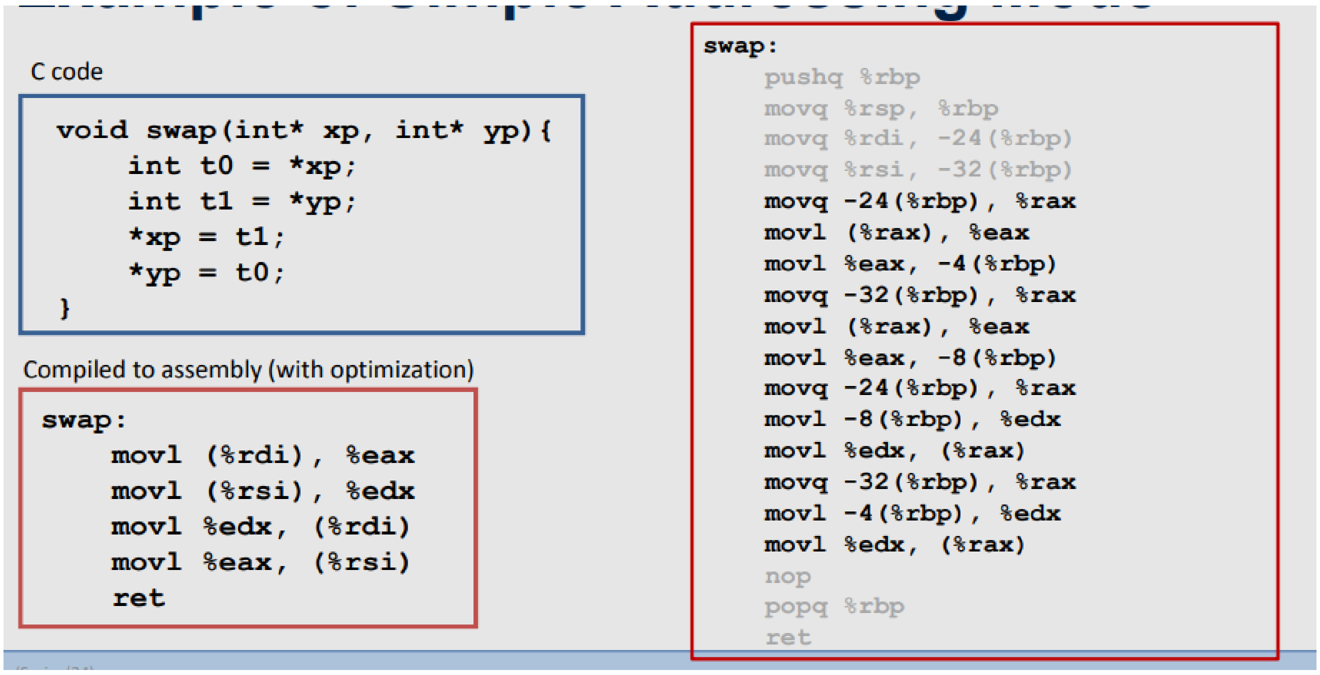
```
movw 0x8(%rdx,%rcx,4), %rax
```

Registers

%rdx	0x2
%rcx	0x2
%rax	

Memory	Address
A[5]	0x14
A[4]=3	0x12
A[3]	0x10
A[2]	0x0e
A[1]	0x0c
A[0]	0x0a
	...
	0x02
	0x00

Example 1 of Simple Addressing Mode



C code

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

Compiled to assembly (with optimization)

```
swap:
    movl  (%rdi), %eax
    movl  (%rsi), %edx
    movl  %edx, (%rdi)
    movl  %eax, (%rsi)
    ret
```

```
swap:
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, -24(%rbp)
    movq %rsi, -32(%rbp)
    movq -24(%rbp), %rax
    movl (%rax), %eax
    movl %eax, -4(%rbp)
    movq -32(%rbp), %rax
    movl (%rax), %eax
    movl %eax, -8(%rbp)
    movq -24(%rbp), %rax
    movl -8(%rbp), %edx
    movl %edx, (%rax)
    movq -32(%rbp), %rax
    movl -4(%rbp), %edx
    movl %edx, (%rax)
    nop
    popq %rbp
    ret
```

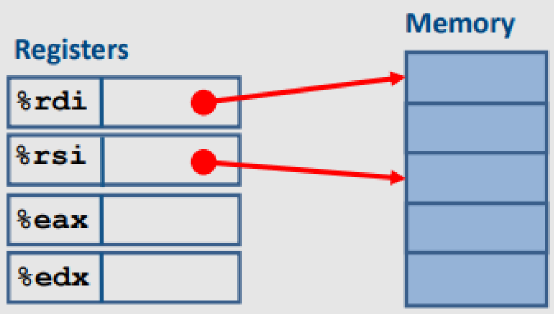
RHS = Just compilation
LHS = Compilation + optimisation

I = 4 bytes

Example 2 of Simple Addressing Mode

C code

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

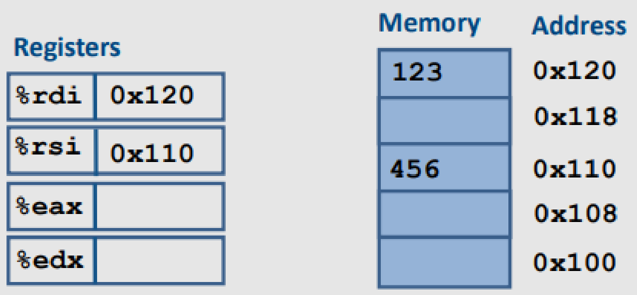


Compiled to assembly (with optimization)

```
swap:
     movl  (%rdi), %eax
     movl  (%rsi), %edx
     movl  %edx, (%rdi)
     movl  %eax, (%rsi)
     ret
```

C code

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

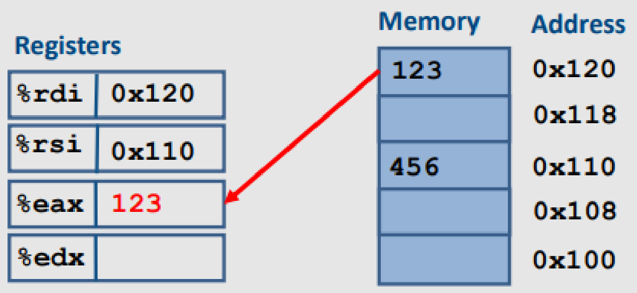


Compiled to assembly (with optimization)

```
swap:
     movl  (%rdi), %eax
     movl  (%rsi), %edx
     movl  %edx, (%rdi)
     movl  %eax, (%rsi)
     ret
```

C code

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

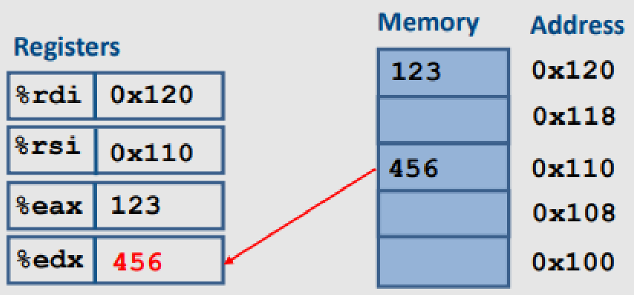


Compiled to assembly (with optimization)

```
swap:
     movl (%rdi), %eax
     movl (%rsi), %edx
     movl %edx, (%rdi)
     movl %eax, (%rsi)
     ret
```

C code

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

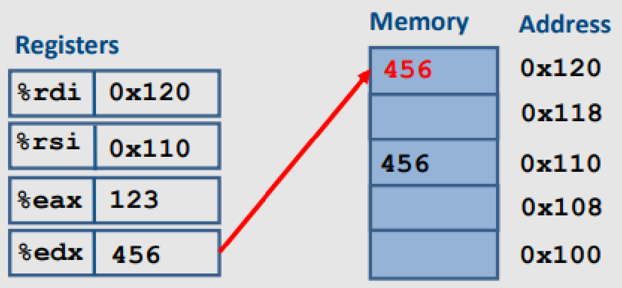


Compiled to assembly (with optimization)

```
swap:
     movl (%rdi), %eax
     movl (%rsi), %edx
     movl %edx, (%rdi)
     movl %eax, (%rsi)
     ret
```

C code

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

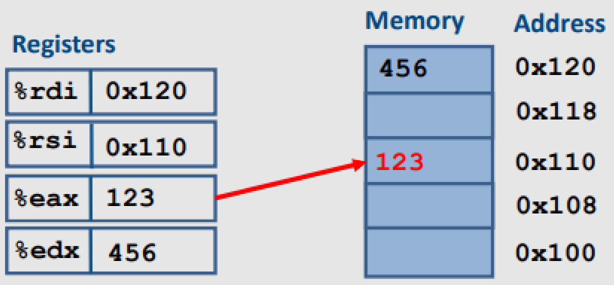


Compiled to assembly (with optimization)

```
swap:
     movl (%rdi), %eax
     movl (%rsi), %edx
     movl %edx, (%rdi)
     movl %eax, (%rsi)
     ret
```

C code

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



Compiled to assembly (with optimization)

```
swap:
     movl (%rdi), %eax
     movl (%rsi), %edx
     movl %edx, (%rdi)
     movl %eax, (%rsi)
     ret
```