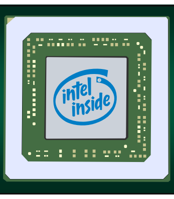
Intro to the Intel x64

Part 2

1



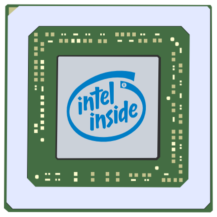
The Intel x64

It was simple at first…

2

The Intel x64

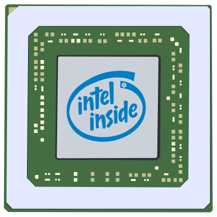
- The Intel x64 is the main processor used by servers, laptops, and desktops
- It has evolved continuously over a 40+ year period



3

The Original x86

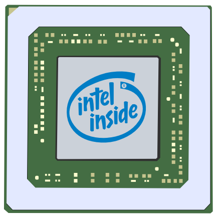
- First "x86" was the 8086
- Released in 1978
- Attributes:
  - 16-bit registers
  - 16 registers
  - could access of 1MB of RAM *(in 64KB blocks using a special "segment" register)*



4

What to call the processor

- The classic term "x86" refers to the 32-bit and 16-bit processor family
- With move to 64-bit, the term "x64" is used to differentiate the newest design from the previous

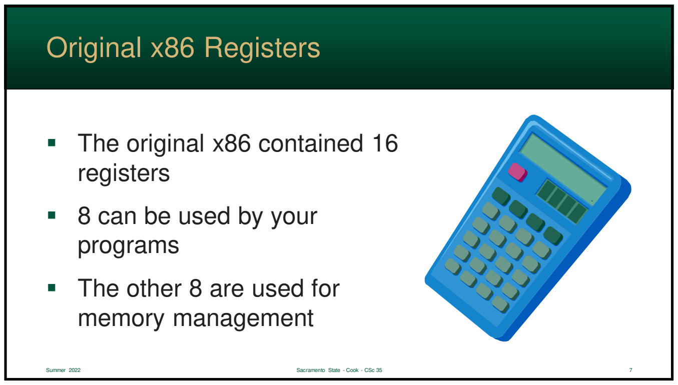


5

Original x86 Registers

It was simple at first…

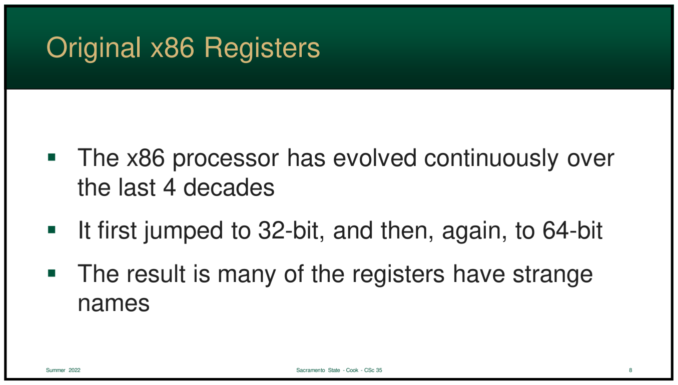
6



Original x86 Registers

- The original x86 contained 16 registers
- 8 can be used by your programs
- The other 8 are used for memory management

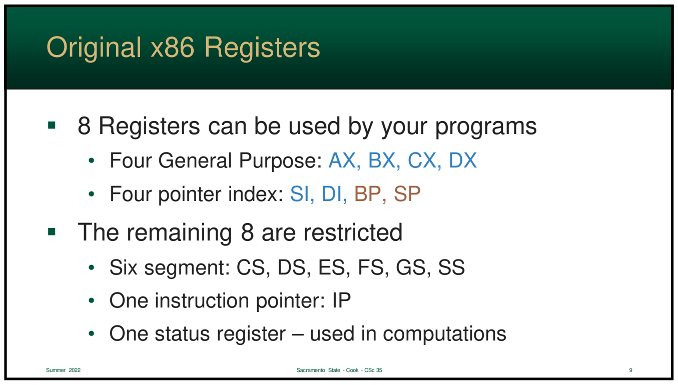
7



Original x86 Registers

- The x86 processor has evolved continuously over the last 4 decades
- It first jumped to 32-bit, and then, again, to 64-bit
- The result is many of the registers have strange names

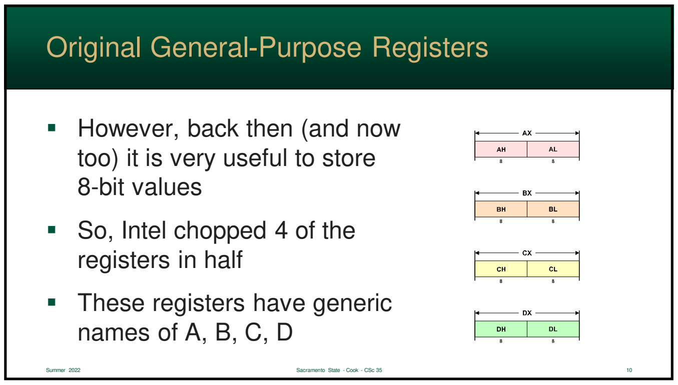
8



Original x86 Registers

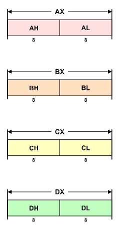
- 8 Registers can be used by your programs
  - Four General Purpose: AX, BX, CX, DX
  - Four pointer index: SI, DI, BP, SP
- The remaining 8 are restricted
  - Six segment: CS, DS, ES, FS, GS, SS
  - One instruction pointer: IP
  - One status register – used in computations

9

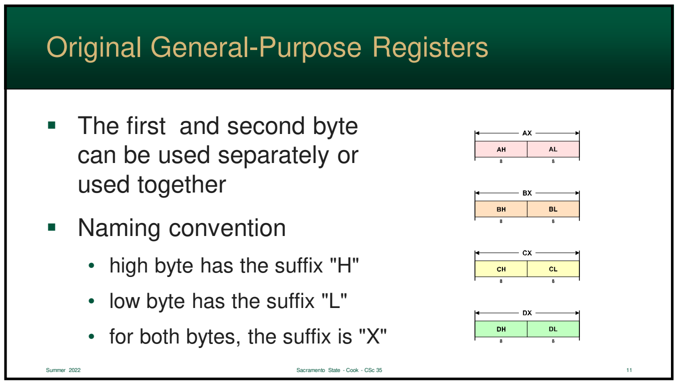


Original General-Purpose Registers

- However, back then (and now too) it is very useful to store 8-bit values
- So, Intel chopped 4 of the registers in half
- These registers have generic names of A, B, C, D

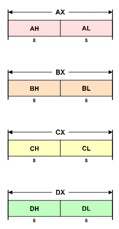


10

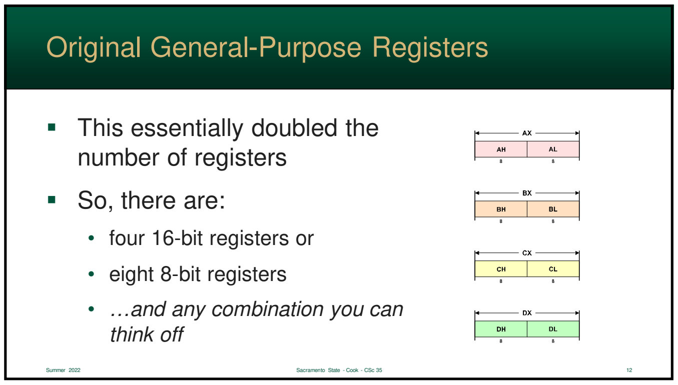


Original General-Purpose Registers

- The first and second byte can be used separately or used together
- Naming convention
  - high byte has the suffix "H"
  - low byte has the suffix "L"
  - for both bytes, the suffix is "X"

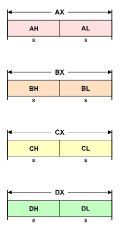


11

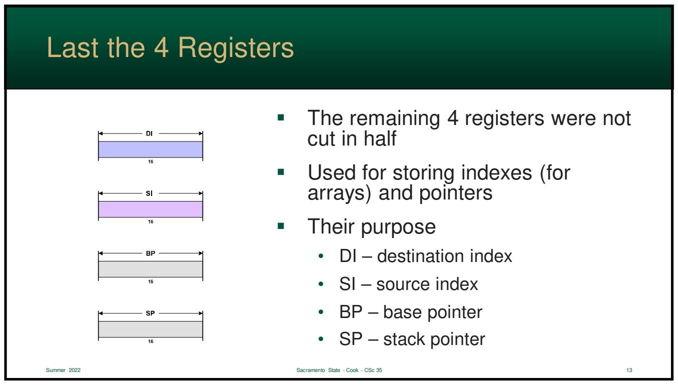


Original General-Purpose Registers

- This essentially doubled the number of registers
- So, there are:
  - four 16-bit registers or
  - eight 8-bit registers
  - *…and any combination you can think off*



12

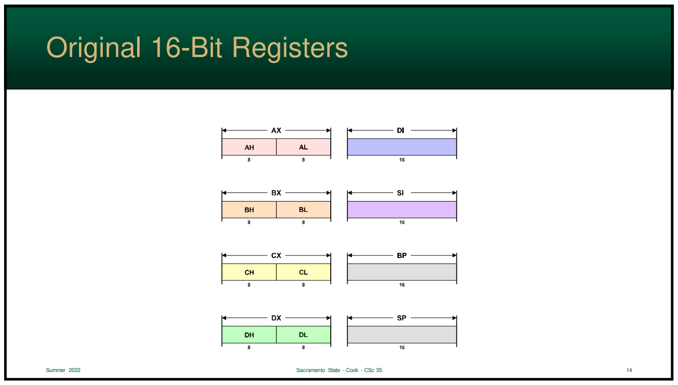


Last the 4 Registers

- The remaining 4 registers were not cut in half
- Used for storing indexes (for arrays) and pointers
- Their purpose
  - DI – destination index
  - SI – source index
  - BP – base pointer
  - SP – stack pointer

Summer 2022 Sacramento State - Cook - CSc 35 13

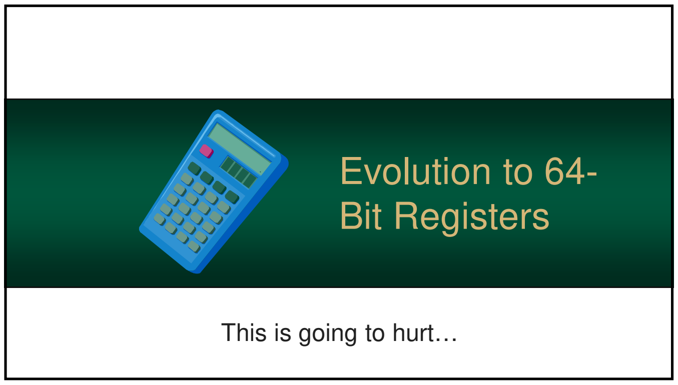
13



Original 16-Bit Registers

Summer 2022 Sacramento State - Cook - CSc 35 14

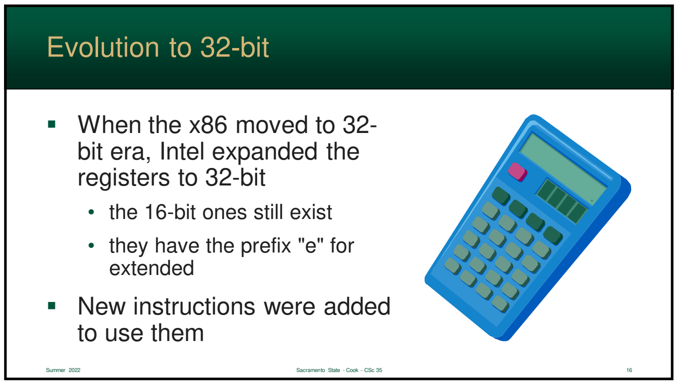
14



Evolution to 64-Bit Registers

This is going to hurt…

15

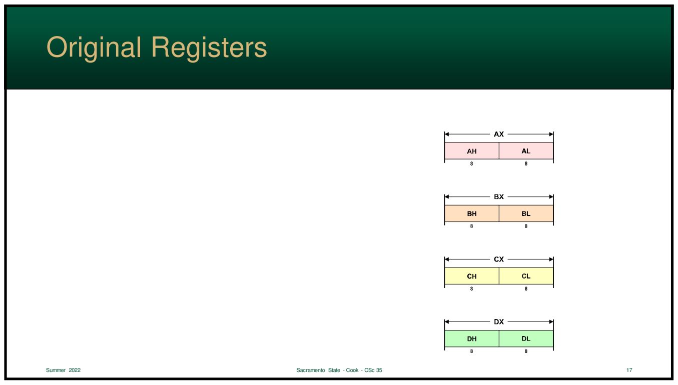


Evolution to 32-bit

- When the x86 moved to 32-bit era, Intel expanded the registers to 32-bit
  - the 16-bit ones still exist
  - they have the prefix "e" for extended
- New instructions were added to use them

Summer 2022 Sacramento State - Cook - CSc 35 16

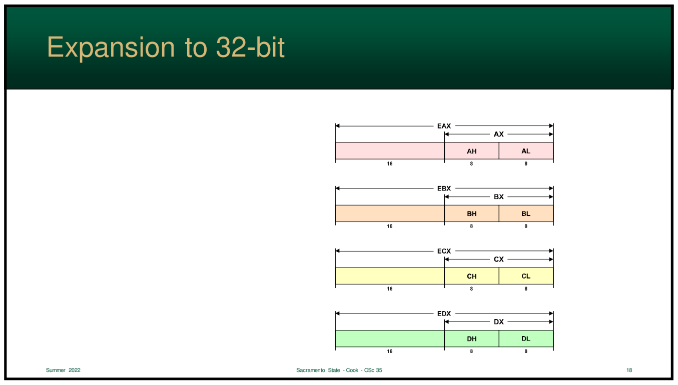
16



Original Registers

Summer 2022 Sacramento State - Cook - CSc 35 17

17

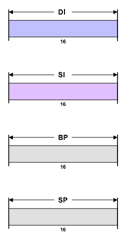


Expansion to 32-bit

Summer 2022 Sacramento State - Cook - CSc 35 18

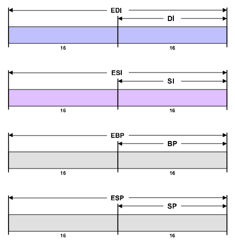
18

Original Registers



19

Expansion to 32-bit



20

Evolution to 64-bit

- At this point, Intel had decided to abandon the x86 in lieu of their new Itanium Processor
- The Itanium was a radically different design and was completely incompatible
- Advanced Micro Devices (AMD), to Intel's chagrin, decided to – *once again* – extend the x86

21

Evolution to 64-bit

- Registers were extended again
  - 64-bit registers have the prefix *"r"*
  - 8 additional registers were added
  - also, it is now possible to get 8-bit values from all registers (hardware is more consistent!)
- Some old, archaic, features were dropped

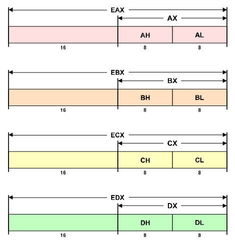
22

Evolution to 64-bit

- The AMD-64 was a huge commercial *success*
- The Itanium was a commercial *failure*
- Intel, dropped the Itanium and started making 64-bit x86 using AMD's design

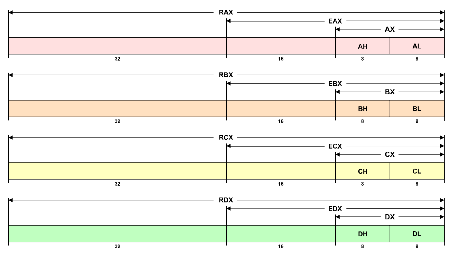
23

Expansion to 64-bit



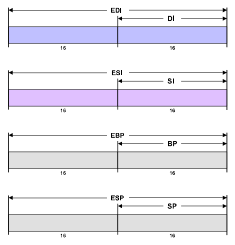
24

Expansion to 64-bit



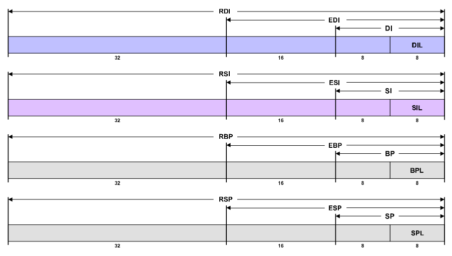
25

Expansion to 64-bit



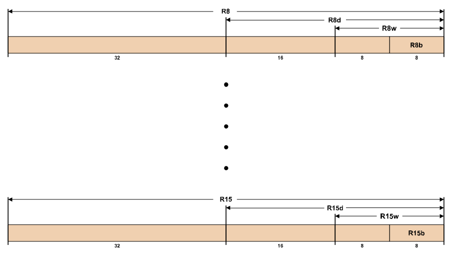
26

Expansion to 64-bit



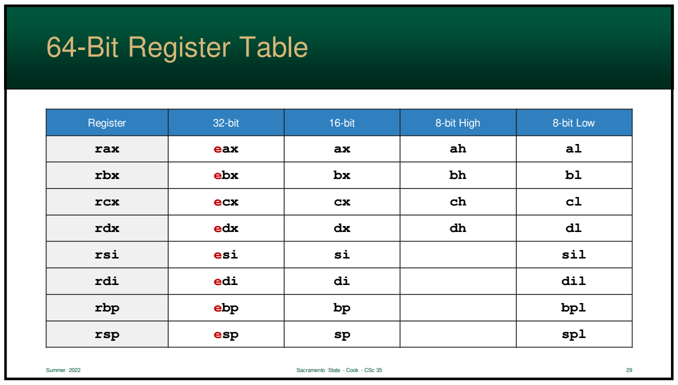
27

New 64-bit Registers: R8…R15



28

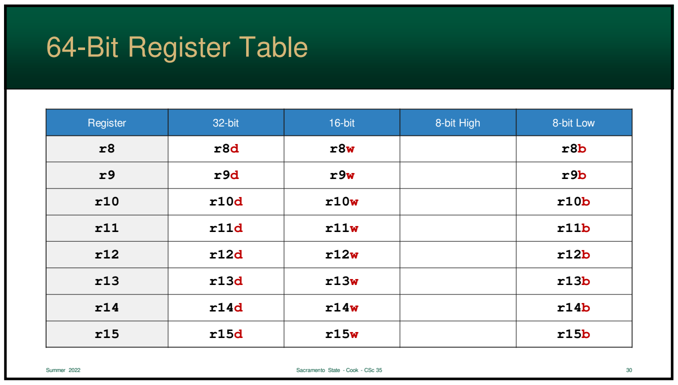
64-Bit Register Table



Register	32-bit	16-bit	8-bit High	8-bit Low
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl

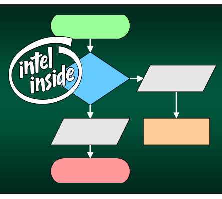
29

64-Bit Register Table



Register	32-bit	16-bit	8-bit High	8-bit Low
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

30



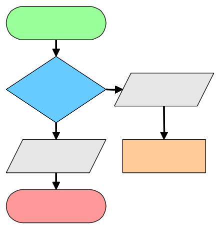
Basic Intel x86 Instructions

Feel the pow-wah of the x86!

31

Basic Intel x86 Instructions

- Each x86 instruction can have up to 2 operands
- Operands in x86 instructions are very versatile
- Each operand can be either a memory address, register or an immediate value



32

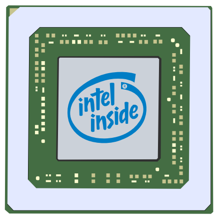
Types of Operands

- Registers
- Address in memory
- Register pointing to a memory address
- Immediate

33

Intel x86 Instruction Limits

- There are some limitations…
- Some instructions must use an immediate
- Some instructions require a *specific* register to perform calculations



34

Intel x86 Instruction Limits

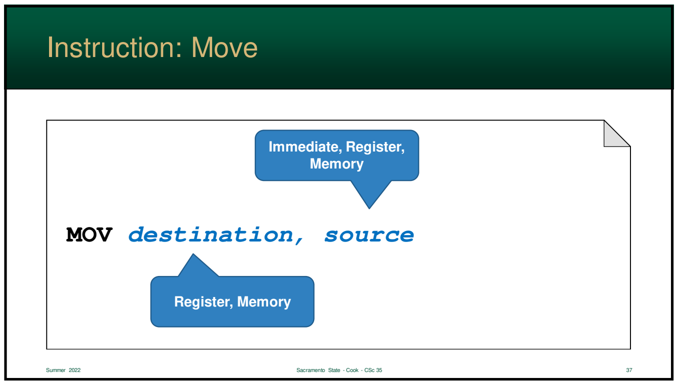
- A register must always be involved
  - processors use registers for all activity
  - both operands cannot access memory at the same time
  - *the processor has to have it at some point!*
- Also, obviously, the receiving field cannot be an immediate value

35

Instruction: Move

- The Intel *Move Instruction* combines transfer, load and store instructions under one name
- … well, that's something the assembler does for us – but, we'll cover that soon
- "Move" is a tad confusing – it copies data

36



Instruction: Move

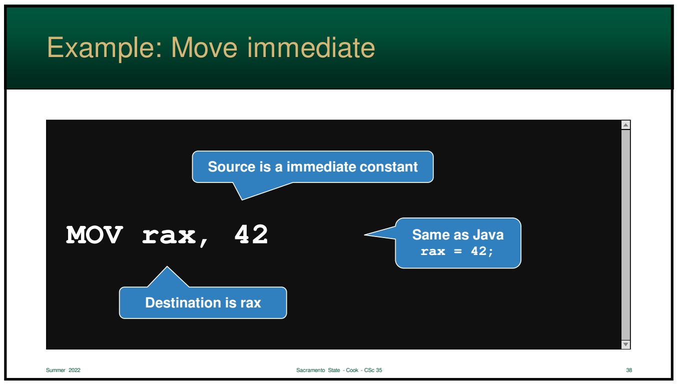
Immediate, Register, Memory

```
MOV destination, source
```

Register, Memory

Summer 2022 Sacramento State - Cook - CSc 35 37

37



Example: Move immediate

Source is a immediate constant

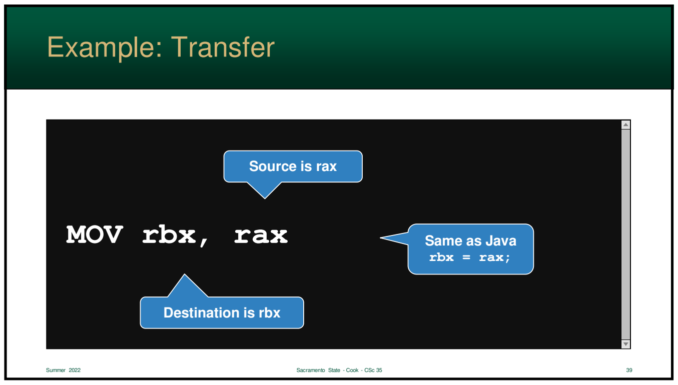
```
MOV rax, 42
```

Same as Java `rax = 42;`

Destination is rax

Summer 2022 Sacramento State - Cook - CSc 35 38

38



Example: Transfer

Source is rax

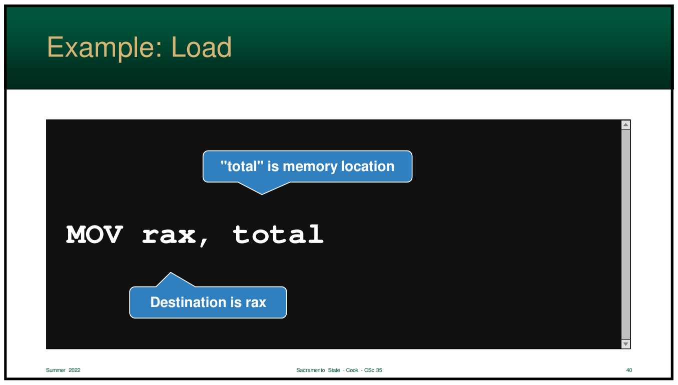
```
MOV rbx, rax
```

Same as Java `rbx = rax;`

Destination is rbx

Summer 2022 Sacramento State - Cook - CSc 35 39

39



Example: Load

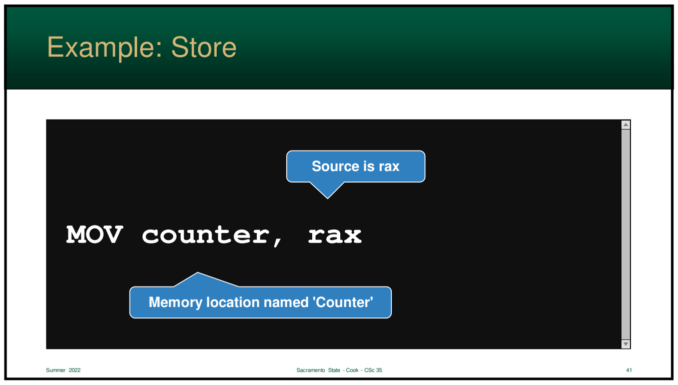
"total" is memory location

```
MOV rax, total
```

Destination is rax

Summer 2022 Sacramento State - Cook - CSc 35 40

40



Example: Store

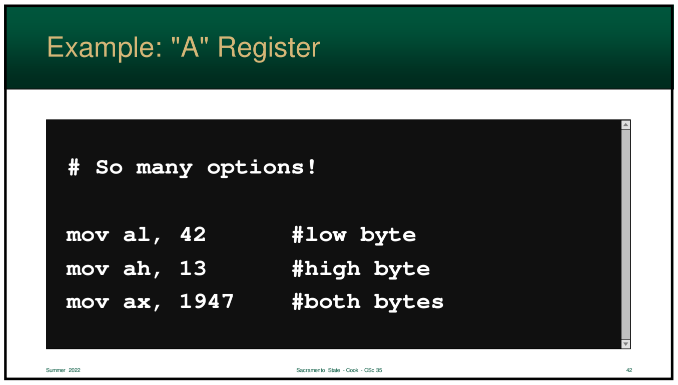
Source is rax

```
MOV counter, rax
```

Memory location named 'Counter'

Summer 2022 Sacramento State - Cook - CSc 35 41

41



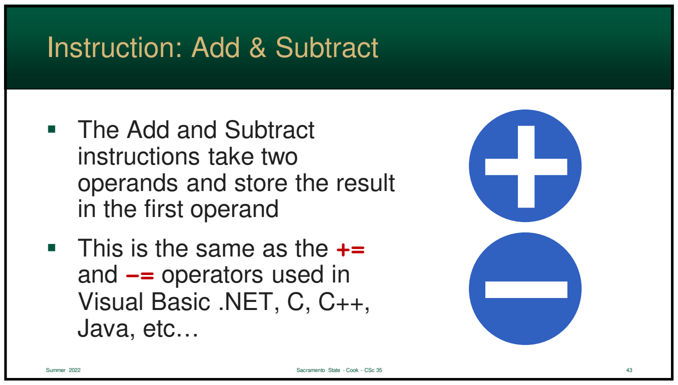
Example: "A" Register

```
# So many options!

mov al, 42      #low byte
mov ah, 13      #high byte
mov ax, 1947    #both bytes
```

Summer 2022 Sacramento State - Cook - CSc 35 42

42

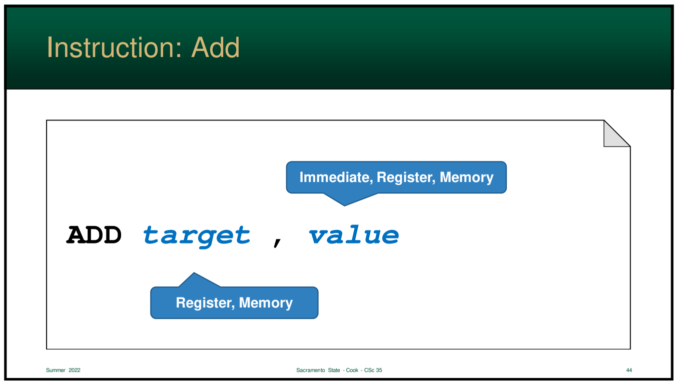


Instruction: Add & Subtract

- The Add and Subtract instructions take two operands and store the result in the first operand
- This is the same as the **+=** and **−=** operators used in Visual Basic .NET, C, C++, Java, etc…

Summer 2022 Sacramento State - Cook - CSc 35 43

43



Instruction: Add

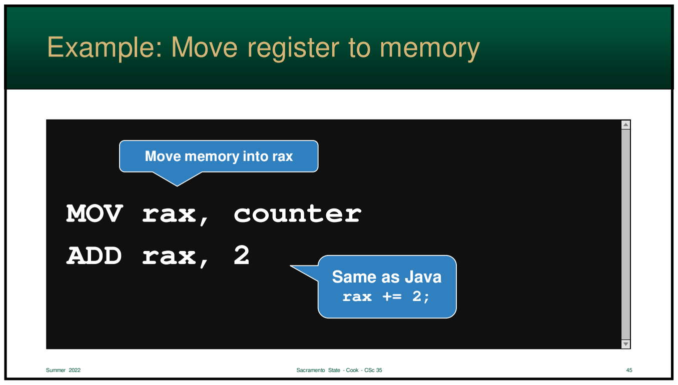
Immediate, Register, Memory

```
ADD target , value
```

Register, Memory

Summer 2022 Sacramento State - Cook - CSc 35 44

44



Example: Move register to memory

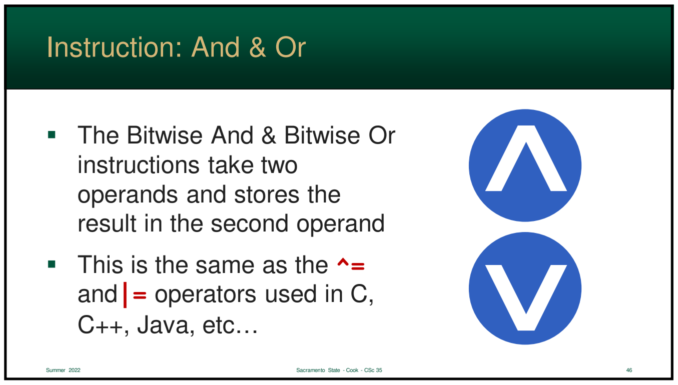
Move memory into rax

```
MOV rax, counter
ADD rax, 2
```

Same as Java
`rax += 2;`

Summer 2022 Sacramento State - Cook - CSc 35 45

45

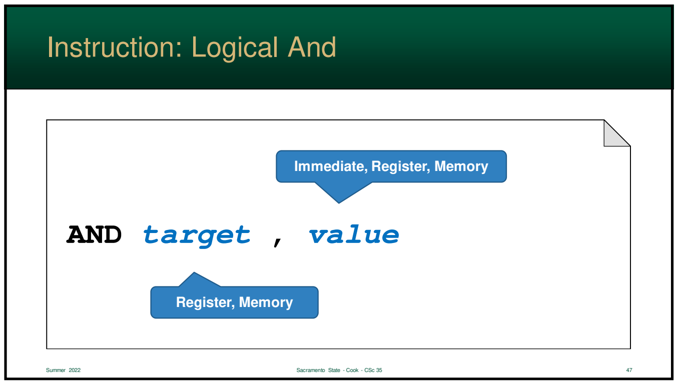


Instruction: And & Or

- The Bitwise And & Bitwise Or instructions take two operands and stores the result in the second operand
- This is the same as the **^=** and **|=** operators used in C, C++, Java, etc…

Summer 2022 Sacramento State - Cook - CSc 35 46

46



Instruction: Logical And

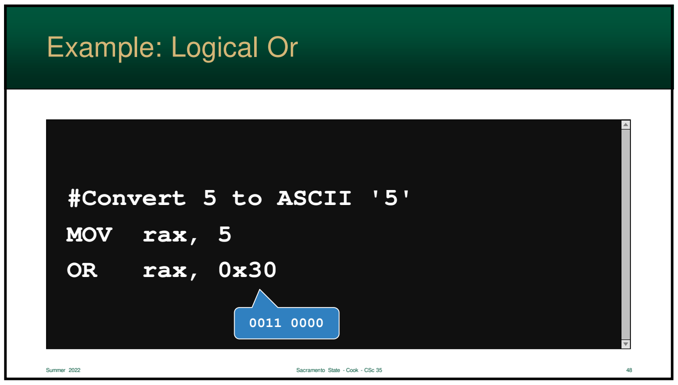
Immediate, Register, Memory

```
AND target , value
```

Register, Memory

Summer 2022 Sacramento State - Cook - CSc 35 47

47



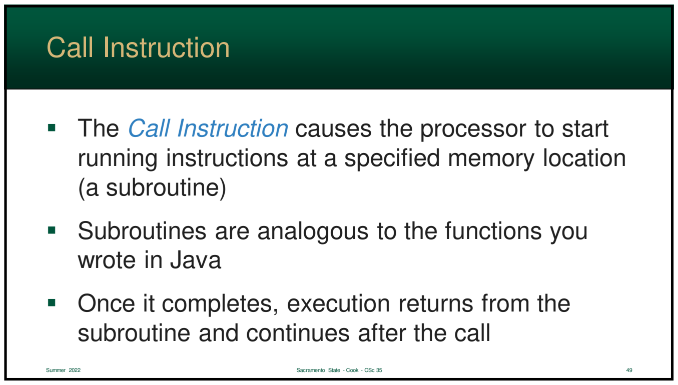
Example: Logical Or

```
#Convert 5 to ASCII '5'
MOV  rax, 5
OR   rax, 0x30
```

0011 0000

Summer 2022 Sacramento State - Cook - CSc 35 48

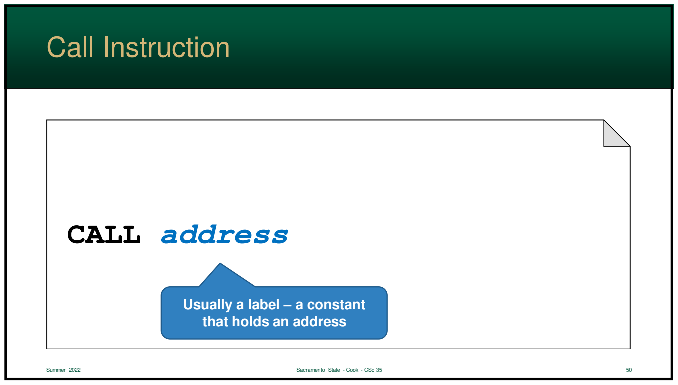
48



Call Instruction

- The *Call Instruction* causes the processor to start running instructions at a specified memory location (a subroutine)
- Subroutines are analogous to the functions you wrote in Java
- Once it completes, execution returns from the subroutine and continues after the call

49

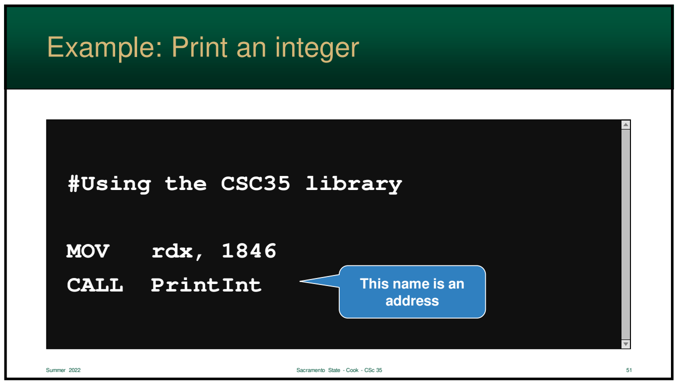


Call Instruction

```
CALL address
```

Usually a label – a constant that holds an address

50



Example: Print an integer

```
#Using the CSC35 library

MOV   rdx, 1846
CALL  PrintInt
```

This name is an address

51