Arithmetic Logic Unit

Part 7

1

Adding Binary Integers

1 + 1 = 10

2

Adding Binary Integers

- Computer's add binary numbers the same way that we do with decimal
- Columns are aligned, added, and "1's" are carried to the next column
- In computer processors, this component is called an *adder*

3

Adding Base 10 Numbers

	1	1		
	2	7	8	1
+	3	7	2	1
	6	5	0	2

4

Adding Binary Example

		1	1	1		1	1		
118		0	1	1	1	0	1	1	0
51	+	0	0	1	1	0	0	1	1
169		1	0	1	0	1	0	0	1

5

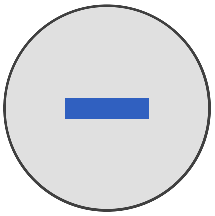
Negative Binary Integers

Have a positive attitude about negatives

6

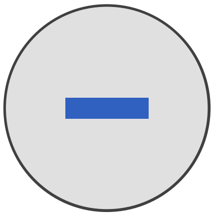
Negative Binary Numbers

- When we write a negative number, we generally use a "-" as a prefix character
- However, binary numbers can only store ones and zeros



Negative Binary Numbers

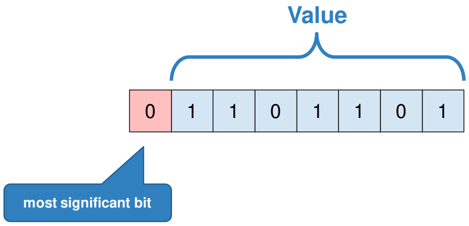
- So, how we store a negative a number?
- When a number can represent both positive and negative numbers, it is called a *signed integer*
- Otherwise, it is *unsigned*



Signed Magnitude

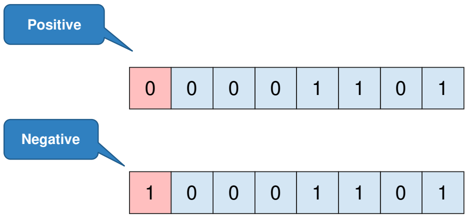
- One approach is to use the most significant bit (msb) to represent the negative sign
- If positive, this bit will be a zero
- If negative, this bit will be a 1
- This gives a byte a range of -127 to 127 rather than 0 to 255

Signed Magnitude



0	1	1	0	1	1	0	1

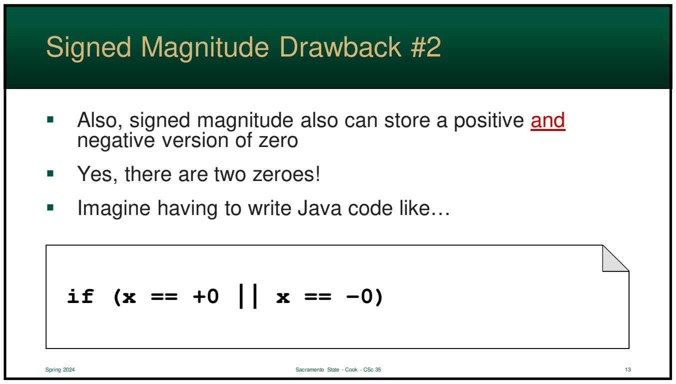
Signed Magnitude: 13 and -13



0	0	0	0	1	1	0	1
1	0	0	0	1	1	0	1

Signed Magnitude Drawback #1

- When two numbers are added, the system needs to check and sign bits and act accordingly
- For example:
    - if both numbers are positive, add values
    - if one is negative subtract it from the other
    - etc…
- There are also rules for subtracting

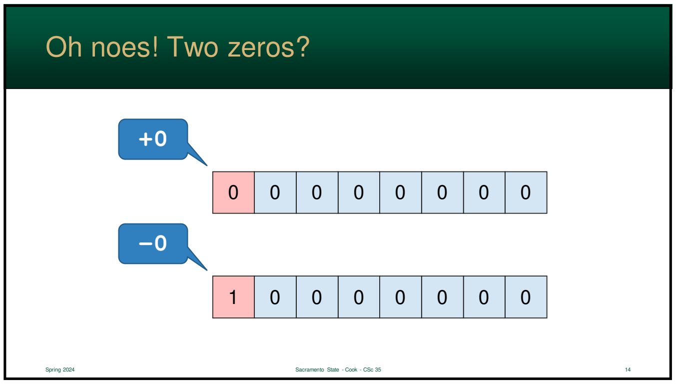


Signed Magnitude Drawback #2

- Also, signed magnitude also can store a positive and negative version of zero
- Yes, there are two zeroes!
- Imagine having to write Java code like…

```
if (x == +0 || x == -0)
```

13

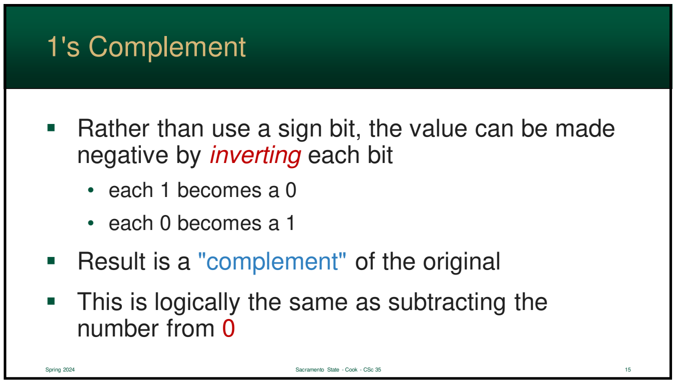


Oh noes! Two zeros?

0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0

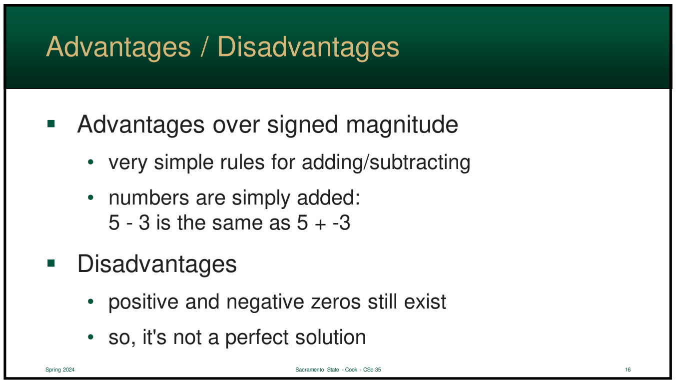
14



1's Complement

- Rather than use a sign bit, the value can be made negative by *inverting* each bit
  - each 1 becomes a 0
  - each 0 becomes a 1
- Result is a "complement" of the original
- This is logically the same as subtracting the number from 0

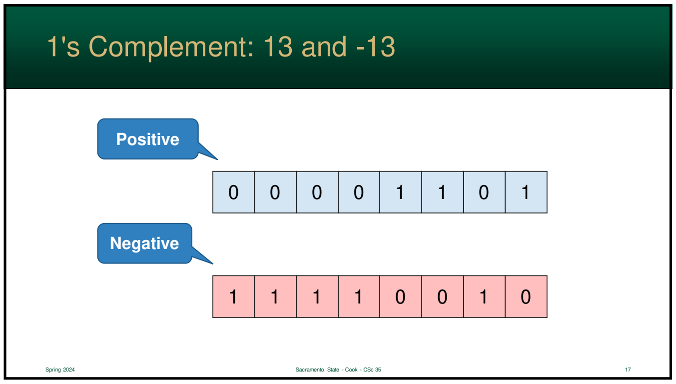
15



Advantages / Disadvantages

- Advantages over signed magnitude
  - very simple rules for adding/subtracting
  - numbers are simply added:
    5 - 3 is the same as 5 + -3
- Disadvantages
  - positive and negative zeros still exist
  - so, it's not a perfect solution

16

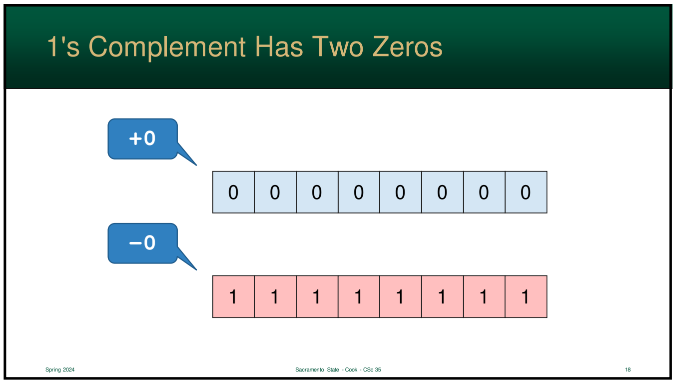


1's Complement: 13 and -13

0	0	0	0	1	1	0	1

1	1	1	1	0	0	1	0

17

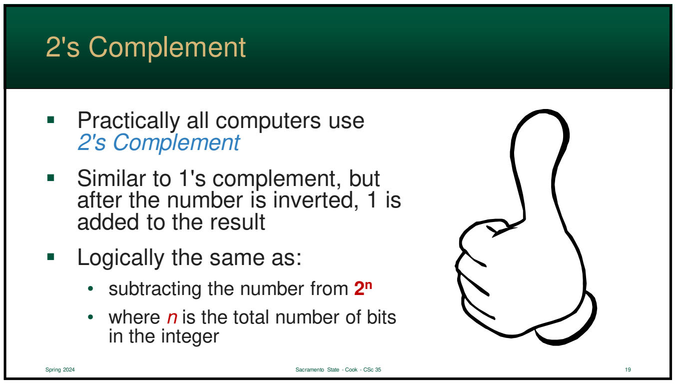


1's Complement Has Two Zeros

0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1

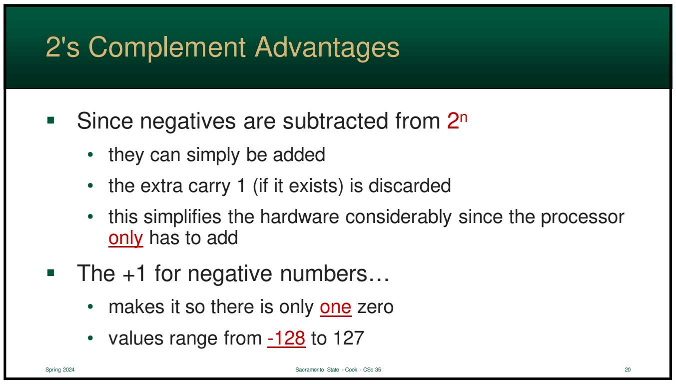
18



2's Complement

- Practically all computers use *2's Complement*
- Similar to 1's complement, but after the number is inverted, 1 is added to the result
- Logically the same as:
  - subtracting the number from **2^n**
  - where *n* is the total number of bits in the integer

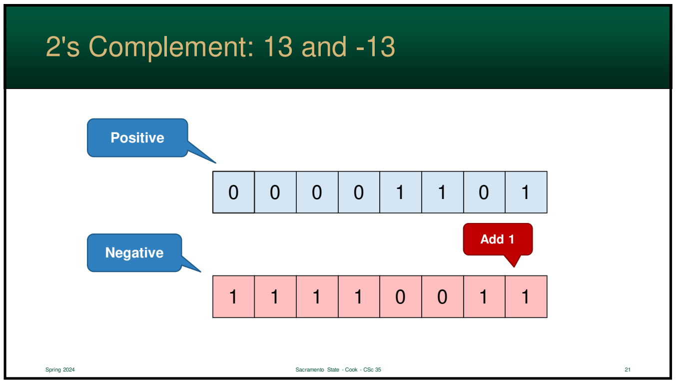
19



2's Complement Advantages

- Since negatives are subtracted from 2^n
  - they can simply be added
  - the extra carry 1 (if it exists) is discarded
  - this simplifies the hardware considerably since the processor only has to add
- The +1 for negative numbers…
  - makes it so there is only one zero
  - values range from -128 to 127

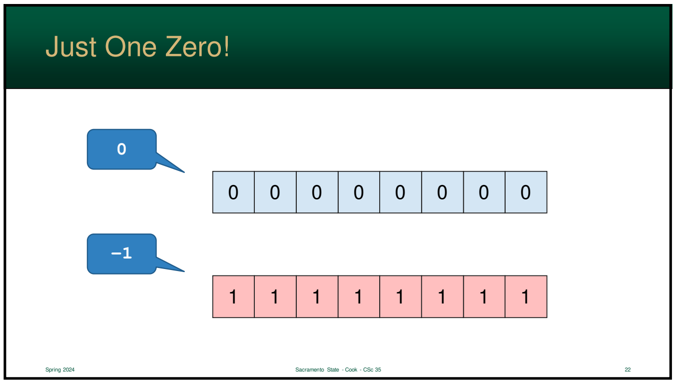
20



2's Complement: 13 and -13

Positive	0	0	0	0	1	1	0	1
								Add 1
Negative	1	1	1	1	0	0	1	1

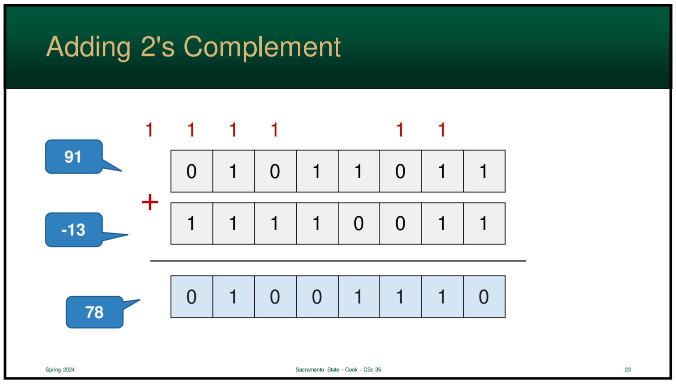
21



Just One Zero!

0	0	0	0	0	0	0	0	0
−1	1	1	1	1	1	1	1	1

22



Adding 2's Complement

	1	1	1	1			1	1	
91		0	1	0	1	1	0	1	1
-13	+	1	1	1	1	0	0	1	1
78		0	1	0	0	1	1	1	0

23

Unsigned or Signed?

- In reality, processors don't know *(or care)* if a number if unsigned or signed
- The hardware works the same either way
- It's your responsibility to keep track if it's signed/unsigned

24

It's Your Responsibility

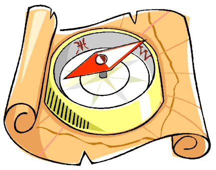
- In many cases, you must use the correct instruction - based on whether *you* are treating the data as signed or unsigned
- *With great programming power comes great responsibility*

Relative Addressing

Jumping of the instruction pointer

Relative Addressing

- In *relative addressing*, a value is added to a instruction pointer (e.g. program counter)
- This allows access a fixed number of bytes *up or down* from the instruction pointer



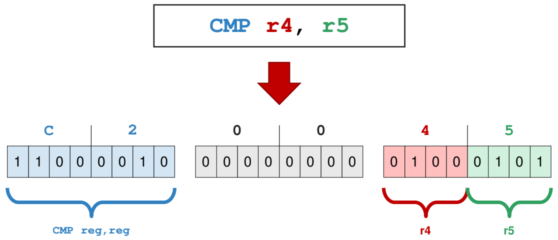
Relative Addressing

- Often used in conditional jump statements
  - jumps are often short – not a large number of instructions
  - so, the instruction only stores the value to add to the program counter
  - practically all processors us this approach
- Also used to access local data – load/store

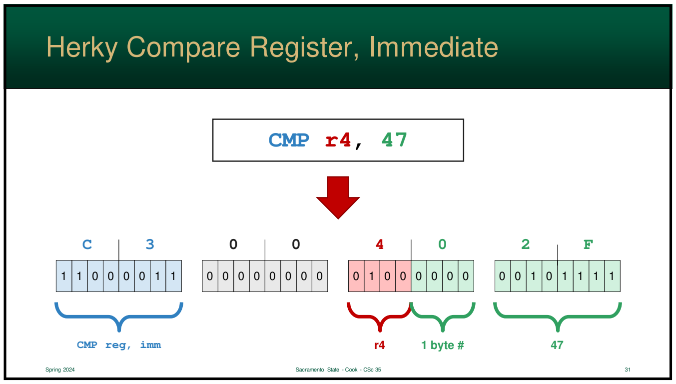
Relative Addressing Advantages

- The instruction can just store the *difference* (in bytes) from the current instruction address
- It takes less storage than a full 64-bit address
- It also allows a program to be stored anywhere in memory – *and it will still work!*

Herky Compare Register, Register

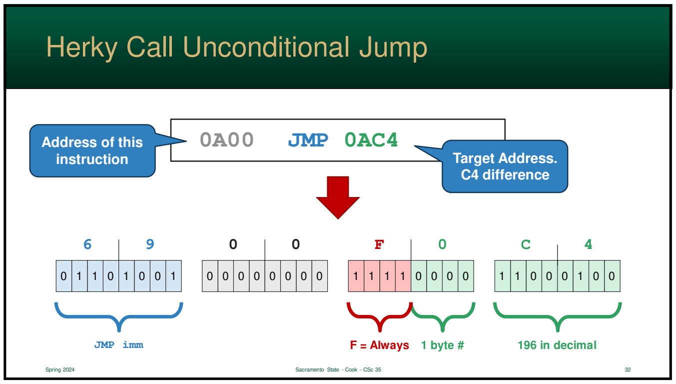


`CMP r4, r5`



Herky Compare Register, Immediate

`CMP r4, 47`

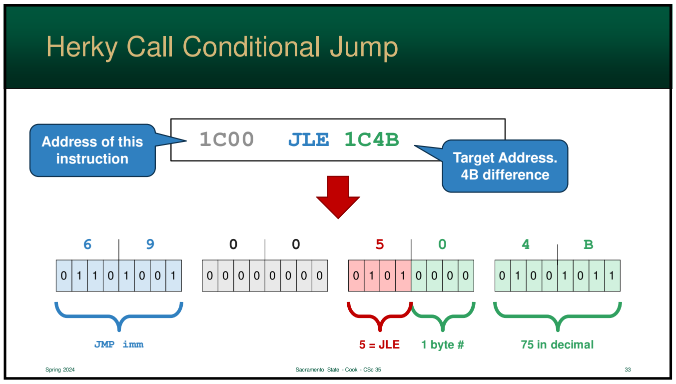


Herky Call Unconditional Jump

`0A00 JMP 0AC4`

Address of this instruction

Target Address. C4 difference



Herky Call Conditional Jump

`1C00 JLE 1C4B`

Address of this instruction

Target Address. 4B difference

Multiplying Binary Numbers

11 × 11 = 1001

Multiplying Binary Numbers

- Many processors today provide complex mathematical instructions
- However, the processor only needs to know how to add
- Historically, multiplication was performed with successive additions

Multiplying Scenario

- Let's say we have two variables: A and B
- Both contain integers that we need to multiply
- Our processor can *only* add (and subtract using 2's complement)
- How do we multiply the values?

Multiplying: The Bad Way

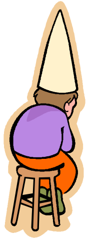
- One way of multiplying the values is to create a For Loop using one of the variables – A or B
- Then, inside the loop, continuously add the other variable to a running total

Multiplying: The Bad Way

```
total = 0;
for (i = 0; i < A; i++)
{
   total += B;
}
```

Multiplying: The Bad Way

- If A or B is large, then it could take a long time
- This is incredibly inefficient
- Also, given that A and B could contain drastically different values – the number of iterations would vary
- Required time is not constant



Multiplying: The Best Way

- Computers can multiply by using long multiplication – *just like you do*
- Number of additions is fixed to 8, 16, 32, 64 depending on the size of the integer
- The following example multiplies 2 unsigned 4-bit numbers

Unsigned Integer: 13 × 10

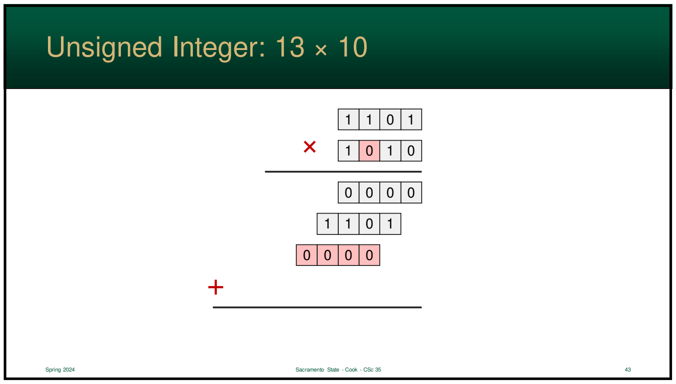
```
      1 1 0 1
  ×   1 0 1 0
  -----------
      0 0 0 0


+
-------------
```

Unsigned Integer: 13 × 10

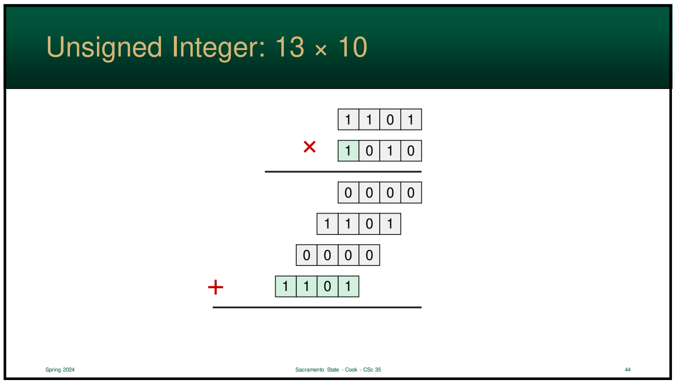
```
      1 1 0 1
  ×   1 0 1 0
  -----------
      0 0 0 0
    1 1 0 1


+
-------------
```



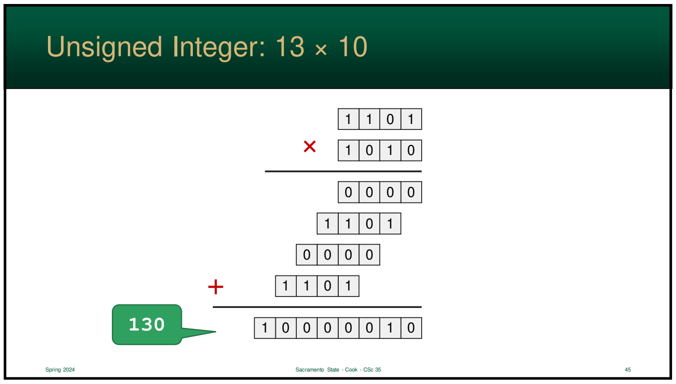
Unsigned Integer: 13 × 10

```
        1 1 0 1
    ×   1 0 1 0
    -----------
        0 0 0 0
      1 1 0 1
    0 0 0 0
+
-------------
```



Unsigned Integer: 13 × 10

```
        1 1 0 1
    ×   1 0 1 0
    -----------
        0 0 0 0
      1 1 0 1
    0 0 0 0
+ 1 1 0 1
-------------
```



Unsigned Integer: 13 × 10

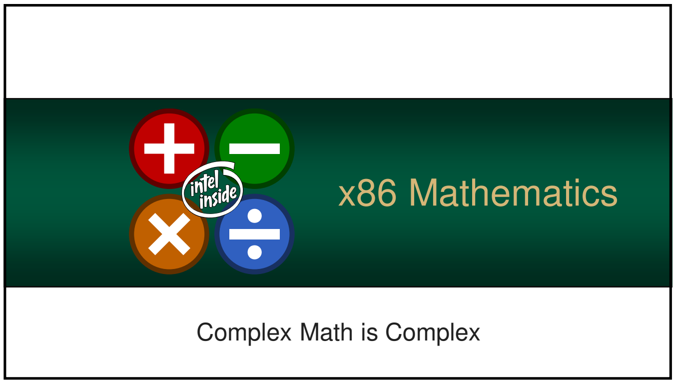
```
              1 1 0 1
          ×   1 0 1 0
          -----------
              0 0 0 0
            1 1 0 1
          0 0 0 0
      + 1 1 0 1
      ---------------
130     1 0 0 0 0 0 1 0
```

Multiplication Doubles the Bit-Count

- When two numbers are multiplied, the product will have twice the number of digits
- Examples:
  - 8-bit × 8-bit → 16-bit
  - 16-bit × 16-bit → 32-bit
  - 32-bit × 32-bit → 64-bit
  - 64-bit × 64-bit → 128-bit

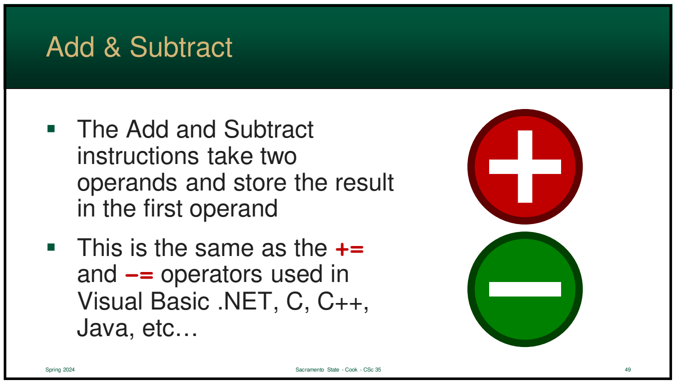
Multiplication Doubles the Bit-Count

- So, how do we store the result?
- It is often too large to fit into any single existing register
- Processors can…
  - fit the result in the original bit-size *(and raise an overflow if it does not fit)*
  - …or store the new double-sized number



x86 Mathematics

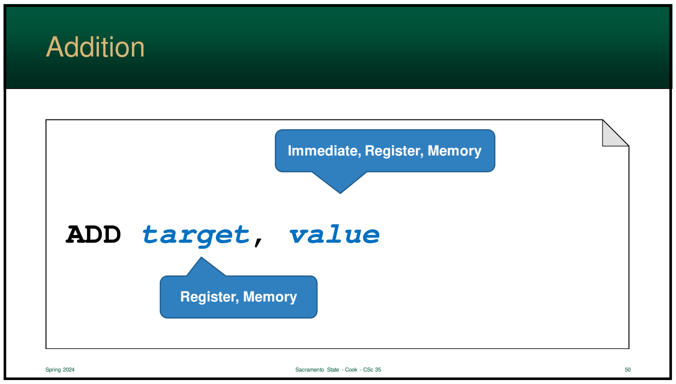
Complex Math is Complex



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…

49



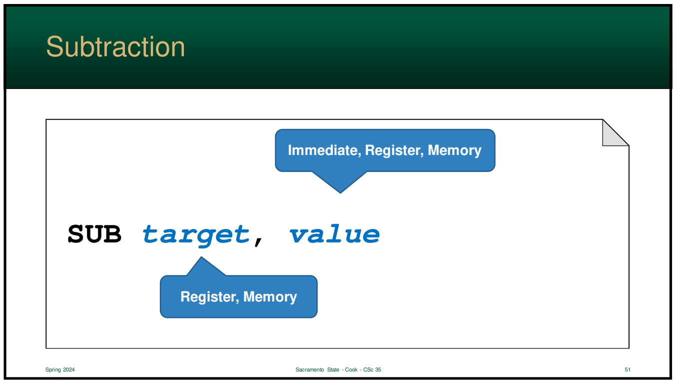
Addition

Immediate, Register, Memory

```
ADD target, value
```

Register, Memory

50



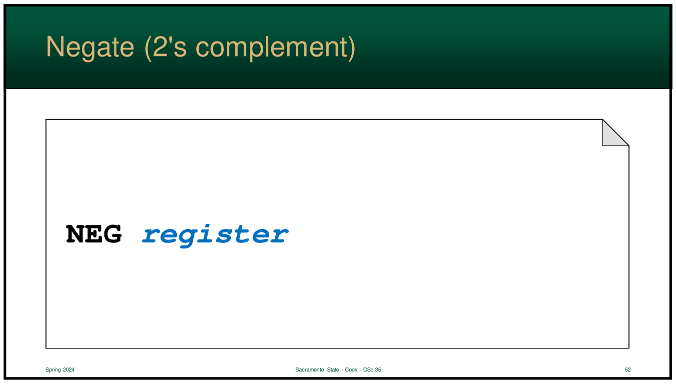
Subtraction

Immediate, Register, Memory

```
SUB target, value
```

Register, Memory

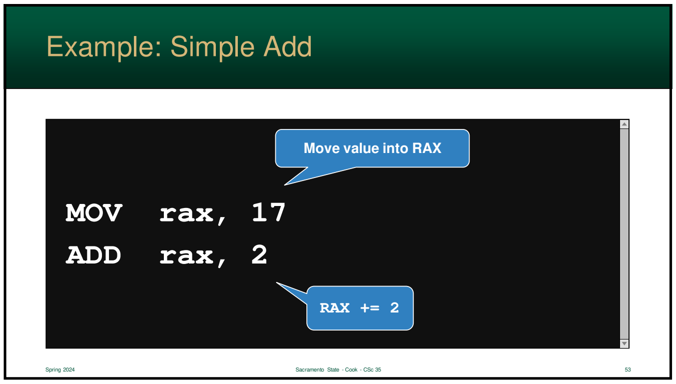
51



Negate (2's complement)

```
NEG register
```

52



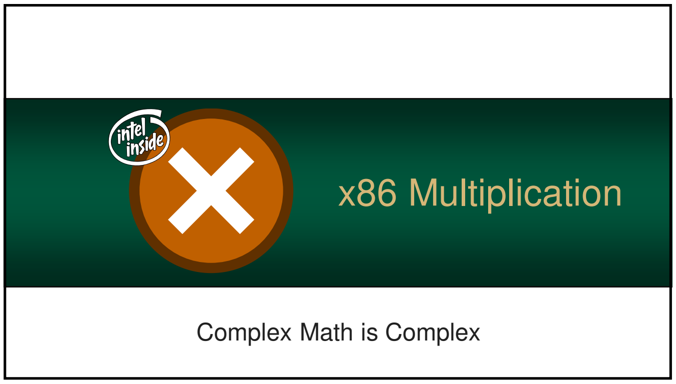
Example: Simple Add

Move value into RAX

```
MOV   rax, 17
ADD   rax, 2
```

RAX += 2

53



x86 Multiplication

Complex Math is Complex

54

Multiplication & Division

- The x86 treats multiplication quite differently than add/subtract
- Why? Intel was designed as a business processor and high-precision math is paramount

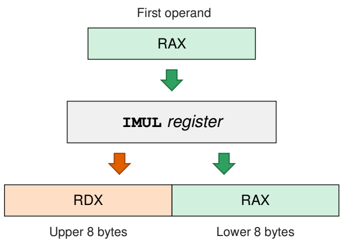
Multiplication Review

- Remember: when two *n* bit numbers are multiplied, result will be *2n* bits
- So…
  - two 8-bit numbers → 16-bit
  - two 16-bit numbers → 32-bit
  - two 32-bit numbers → 64-bit
  - two 64-bit numbers → 128-bit

Multiplication on the x86

- Intel stores the product into two registers
  - RAX will contain the lower 8 bytes
  - RDX will contain the upper 8 bytes
- This maintains the high-precision result
- Instruction inputs are strange
  - first operand is must be stored in RAX
  - second operand must be a register or memory

x86 Multiplication



Multiply - Signed

```
IMUL operand
```

Register or Memory only

Multiply - Unsigned

```
MUL operand
```

Register or Memory only

Signed Multiply: 1846 by 42

```
MOV    rax, 1846      #First operand
MOV    rbx, 42        #Need register for MUL
IMUL   rbx            #RAX gets low 8 bytes
                      #RDX gets high 8 bytes
```

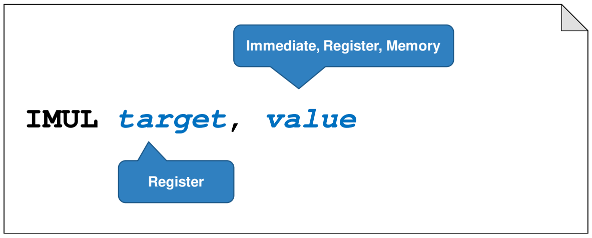
Multiplication Tips

- Even though you are just using RAX as input, both RAX and RDX will change
- Be aware that you might lose important data, and backup to memory if needed

Additional x86 Multiply Instructions

- Over time, designers requested a low-precision version of multiplication
- Intel added "short" IMUL instructions that store into a single register
- Please Note: these do not exist for MUL

IMUL (few more combos)



Immediate, Register, Memory

`IMUL target, value`

Register

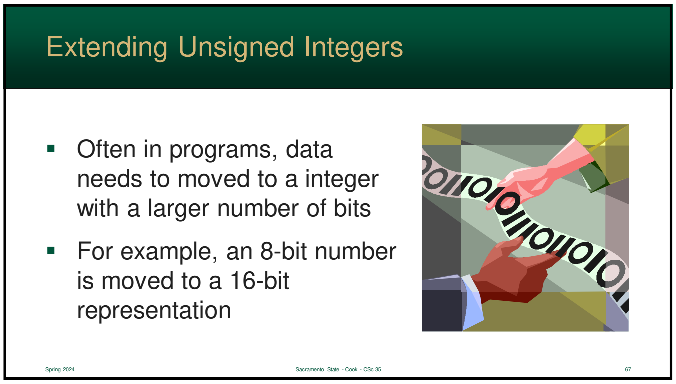
Signed Multiply: 1846 by 42

```
MOV    rax, 1846
IMUL   rax, 42
```

This works, but could cause an overflow

Extending Byte Size

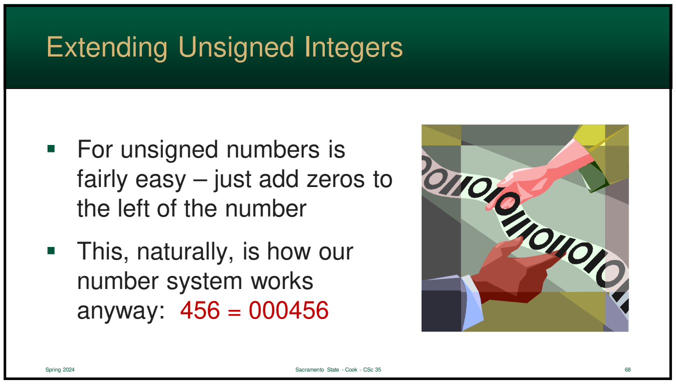
Converting from 8-bit to 16-bit and more



Extending Unsigned Integers

- Often in programs, data needs to moved to a integer with a larger number of bits
- For example, an 8-bit number is moved to a 16-bit representation

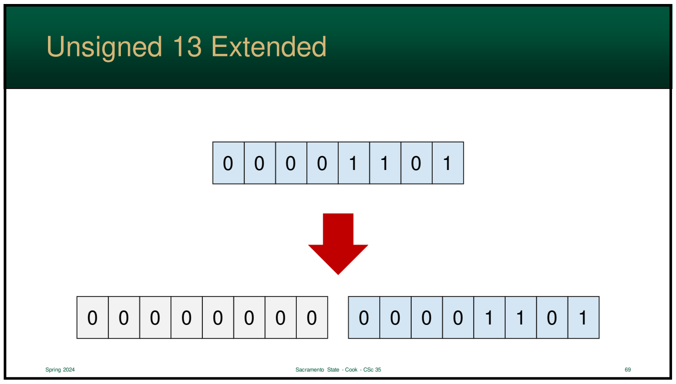
67



Extending Unsigned Integers

- For unsigned numbers is fairly easy – just add zeros to the left of the number
- This, naturally, is how our number system works anyway: 456 = 000456

68

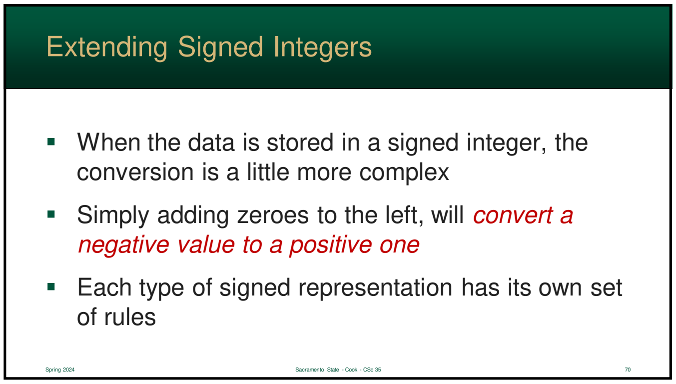


Unsigned 13 Extended

0 0 0 0 1 1 0 1

0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1

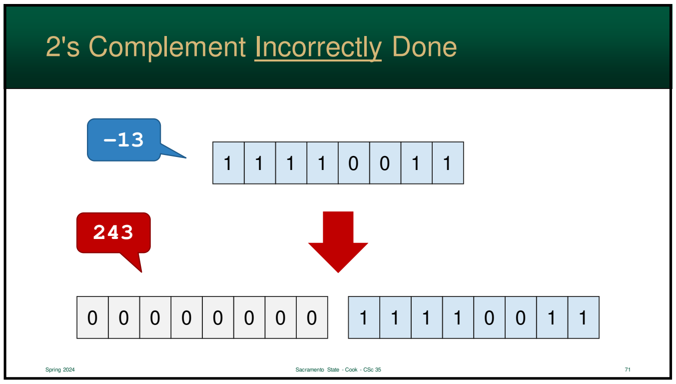
69



Extending Signed Integers

- When the data is stored in a signed integer, the conversion is a little more complex
- Simply adding zeroes to the left, will *convert a negative value to a positive one*
- Each type of signed representation has its own set of rules

70



2's Complement Incorrectly Done

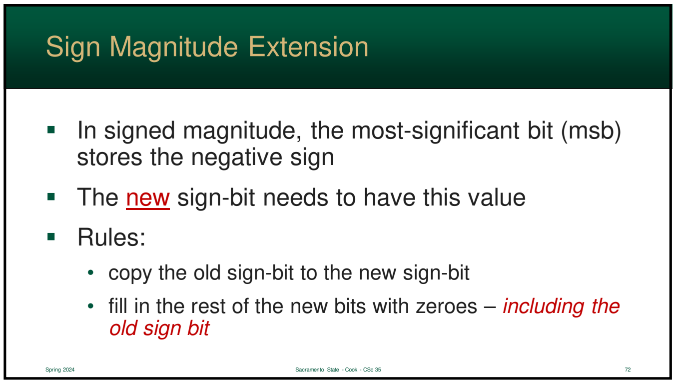
-13

1 1 1 1 0 0 1 1

243

0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1

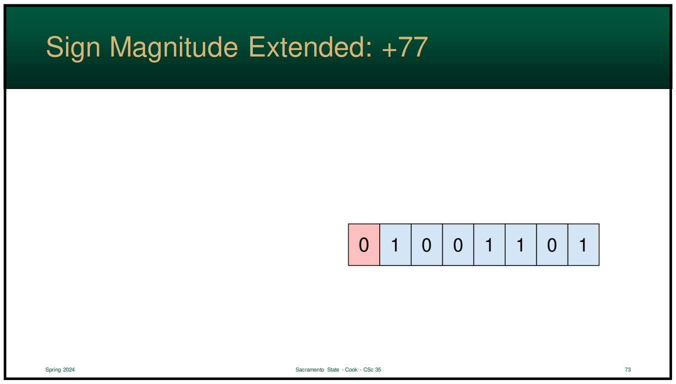
71



Sign Magnitude Extension

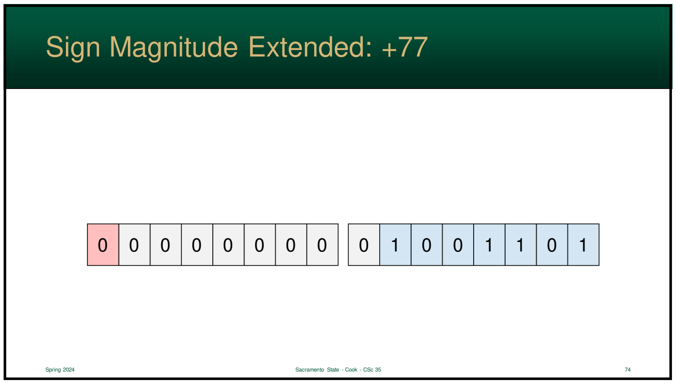
- In signed magnitude, the most-significant bit (msb) stores the negative sign
- The new sign-bit needs to have this value
- Rules:
  - copy the old sign-bit to the new sign-bit
  - fill in the rest of the new bits with zeroes – *including the old sign bit*

72



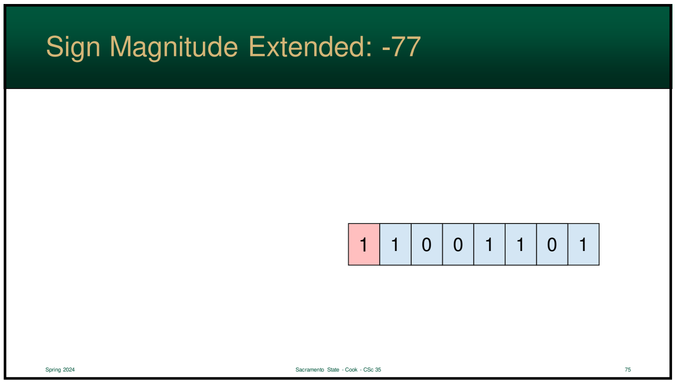
Sign Magnitude Extended: +77

0	1	0	0	1	1	0	1



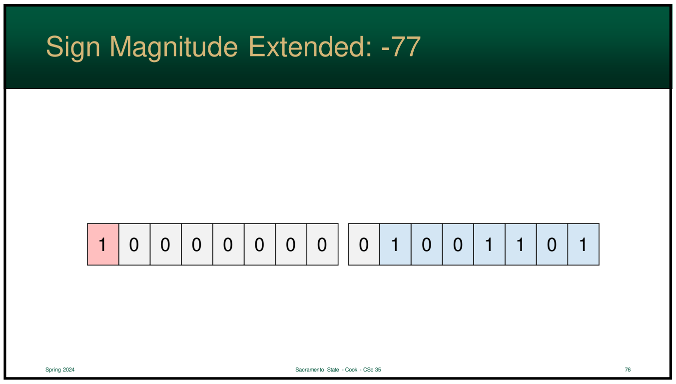
Sign Magnitude Extended: +77

0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1



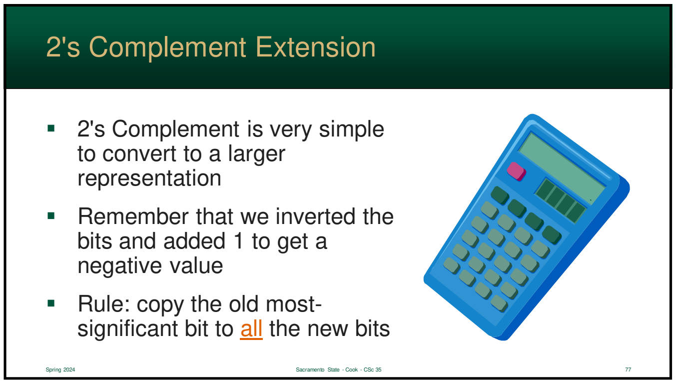
Sign Magnitude Extended: -77

1	1	0	0	1	1	0	1



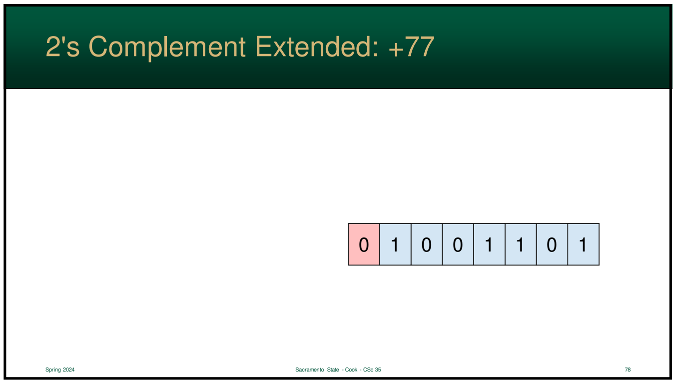
Sign Magnitude Extended: -77

1	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1



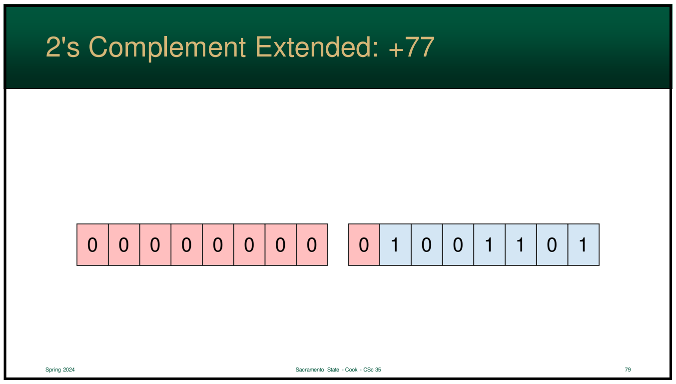
2's Complement Extension

- 2's Complement is very simple to convert to a larger representation
- Remember that we inverted the bits and added 1 to get a negative value
- Rule: copy the old most-significant bit to all the new bits



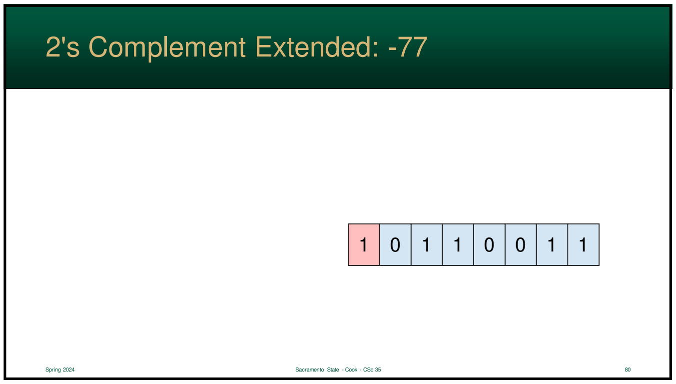
2's Complement Extended: +77

0	1	0	0	1	1	0	1



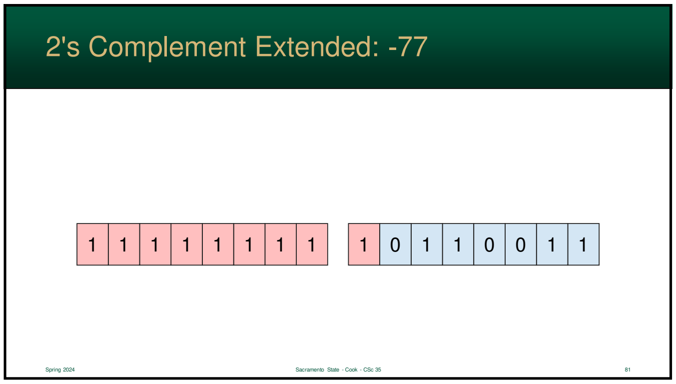
2's Complement Extended: +77

0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1



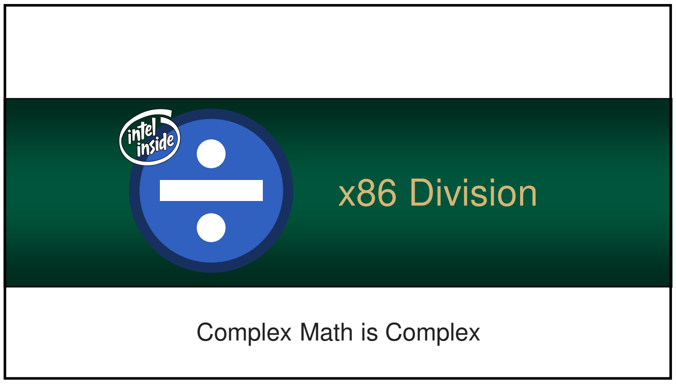
2's Complement Extended: -77

1	0	1	1	0	0	1	1



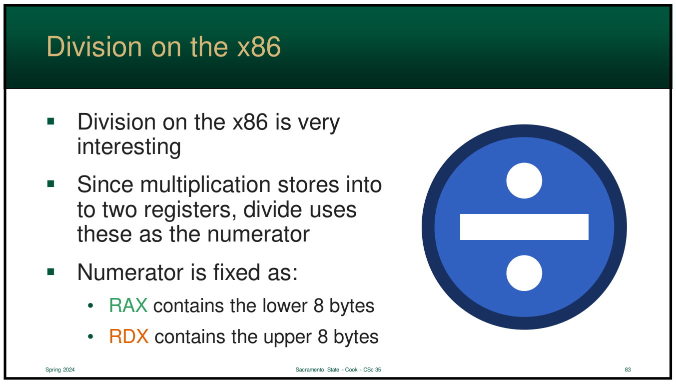
2's Complement Extended: -77

1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1



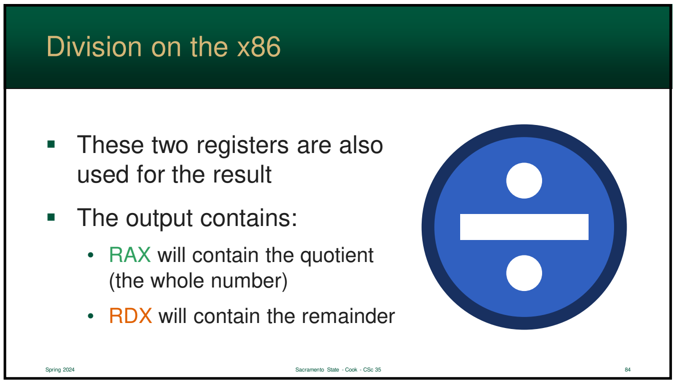
x86 Division

Complex Math is Complex



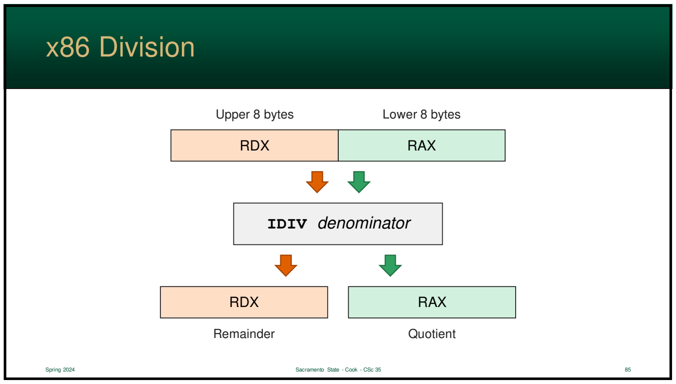
Division on the x86

- Division on the x86 is very interesting
- Since multiplication stores into to two registers, divide uses these as the numerator
- Numerator is fixed as:
  - RAX contains the lower 8 bytes
  - RDX contains the upper 8 bytes



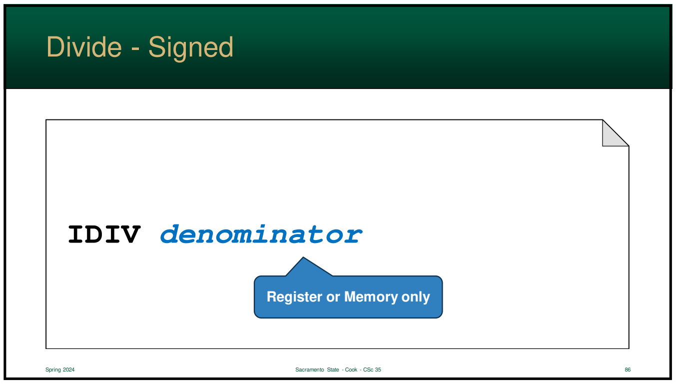
Division on the x86

- These two registers are also used for the result
- The output contains:
  - RAX will contain the quotient (the whole number)
  - RDX will contain the remainder



x86 Division

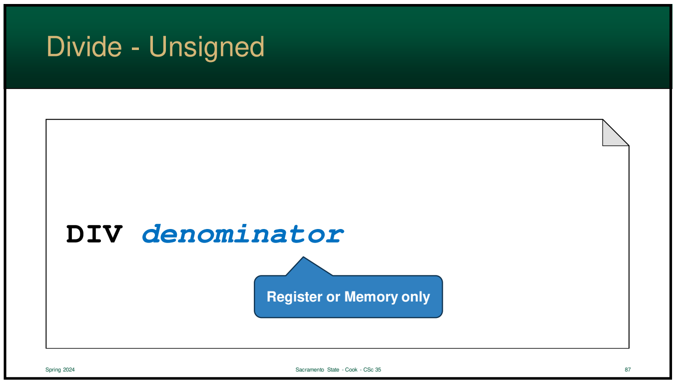
85



Divide - Signed

`IDIV denominator`

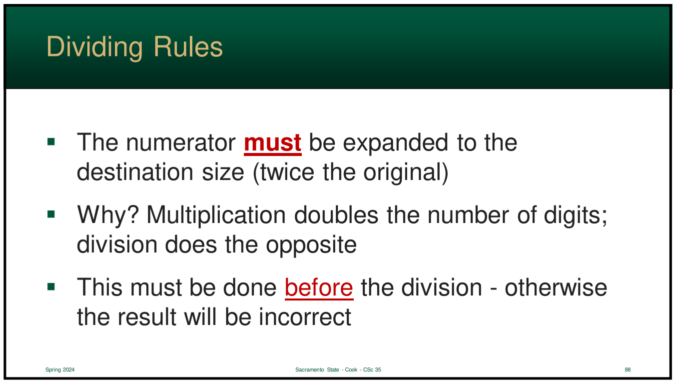
86



Divide - Unsigned

`DIV denominator`

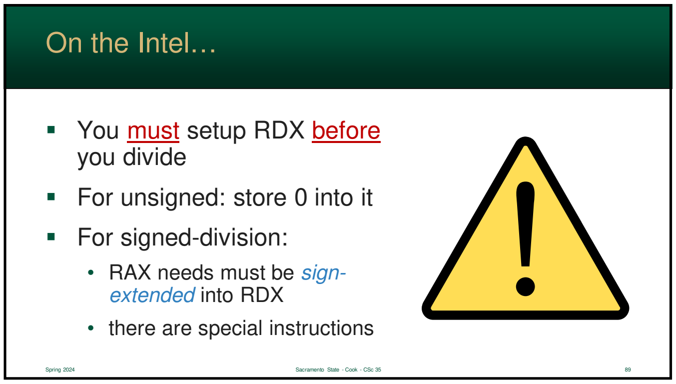
87



Dividing Rules

- The numerator **must** be expanded to the destination size (twice the original)
- Why? Multiplication doubles the number of digits; division does the opposite
- This must be done before the division - otherwise the result will be incorrect

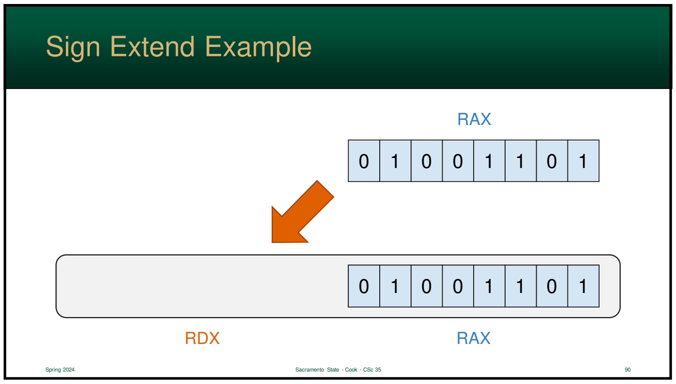
88



On the Intel…

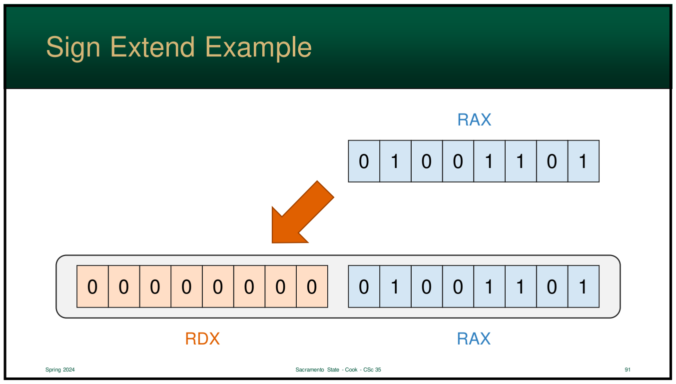
- You must setup RDX before you divide
- For unsigned: store 0 into it
- For signed-division:
  - RAX needs must be *sign-extended* into RDX
  - there are special instructions

89



Sign Extend Example

90



Sign Extend Example

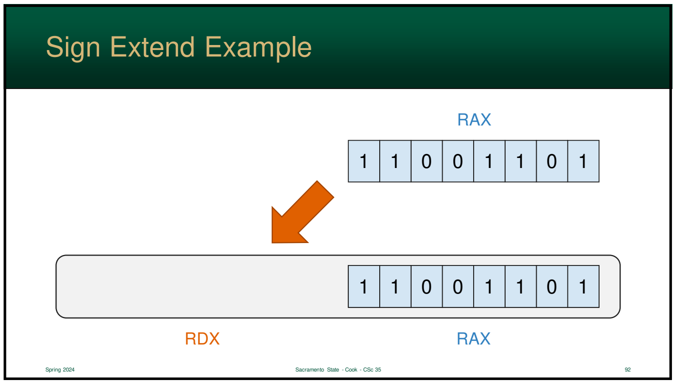
RAX

0	1	0	0	1	1	0	1

0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1

RDX RAX

91



Sign Extend Example

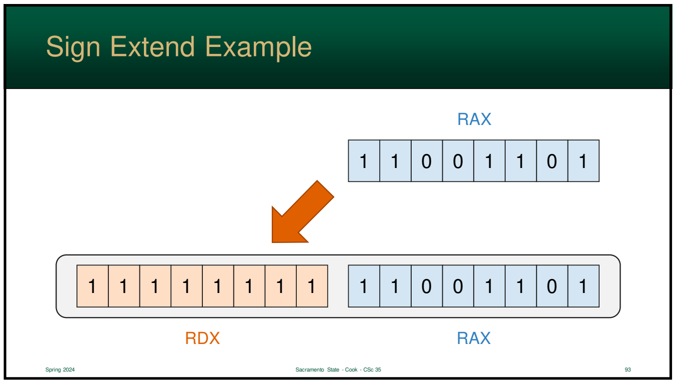
RAX

1	1	0	0	1	1	0	1

								1	1	0	0	1	1	0	1

RDX RAX

92



Sign Extend Example

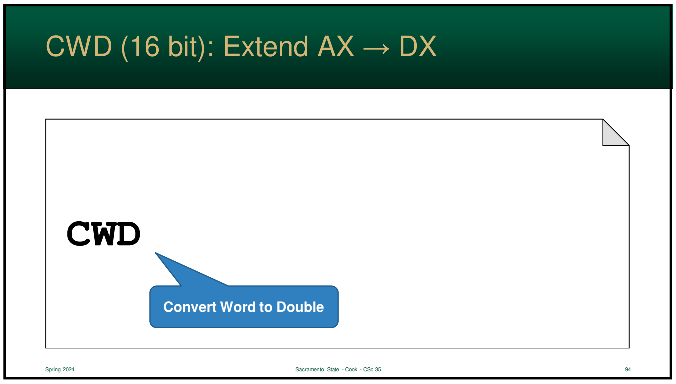
RAX

1	1	0	0	1	1	0	1

1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1

RDX RAX

93

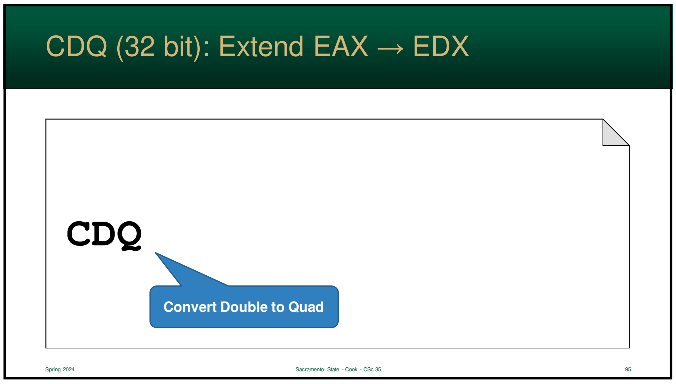


CWD (16 bit): Extend AX → DX

```
CWD
```

Convert Word to Double

94

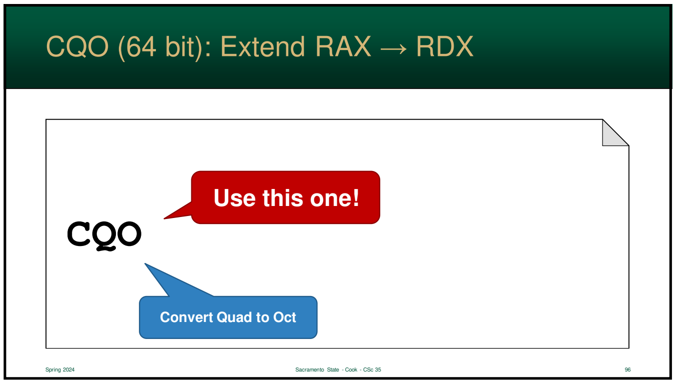


CDQ (32 bit): Extend EAX → EDX

```
CDQ
```

Convert Double to Quad

95



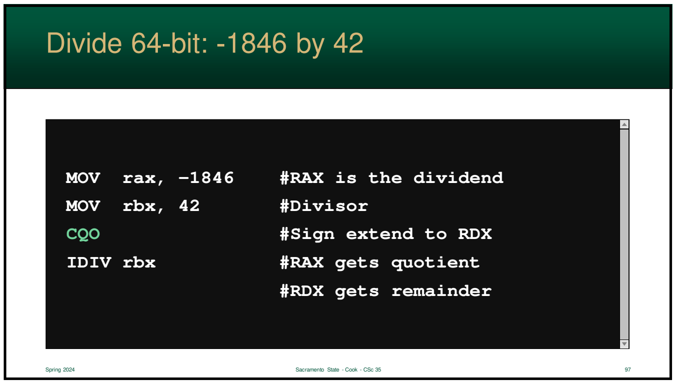
CQO (64 bit): Extend RAX → RDX

```
CQO
```

Use this one!

Convert Quad to Oct

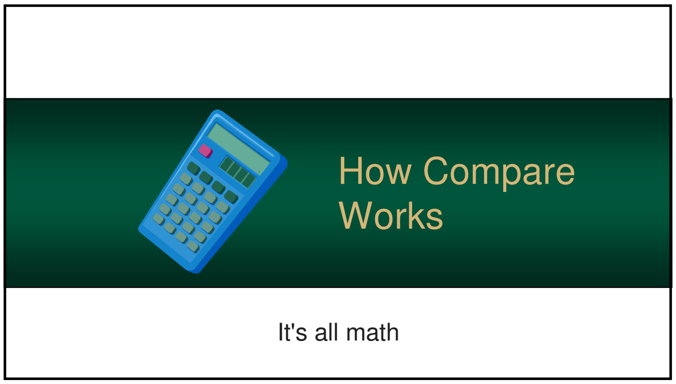
96



Divide 64-bit: -1846 by 42

```
MOV  rax, -1846      #RAX is the dividend
MOV  rbx, 42         #Divisor
CQO                  #Sign extend to RDX
IDIV rbx             #RAX gets quotient
                     #RDX gets remainder
```

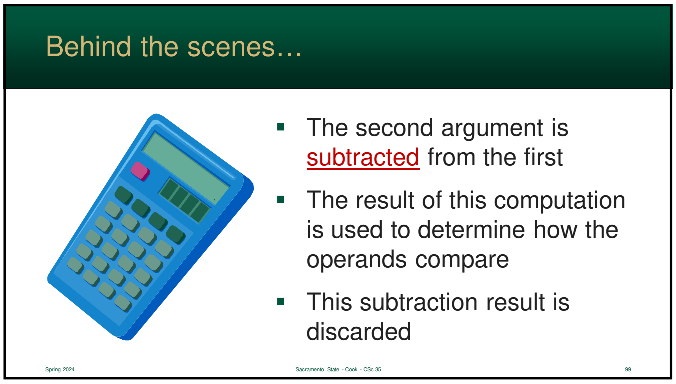
97



How Compare Works

It's all math

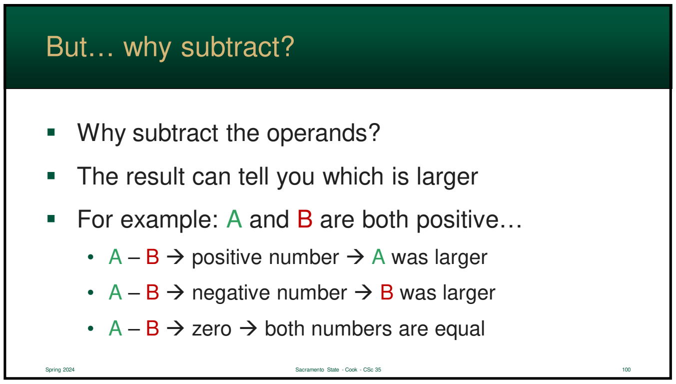
98



Behind the scenes…

- The second argument is subtracted from the first
- The result of this computation is used to determine how the operands compare
- This subtraction result is discarded

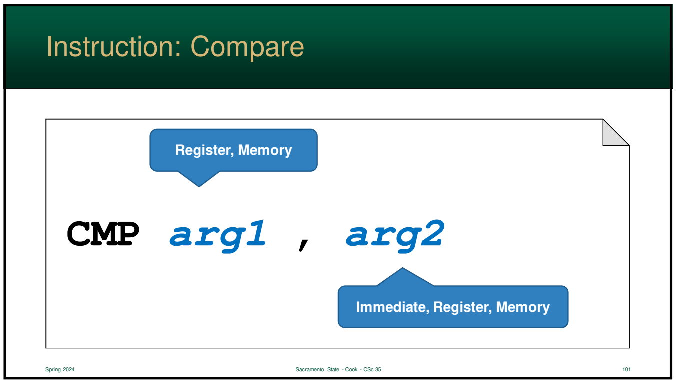
99



But… why subtract?

- Why subtract the operands?
- The result can tell you which is larger
- For example: A and B are both positive…
  - A – B → positive number → A was larger
  - A – B → negative number → B was larger
  - A – B → zero → both numbers are equal

100



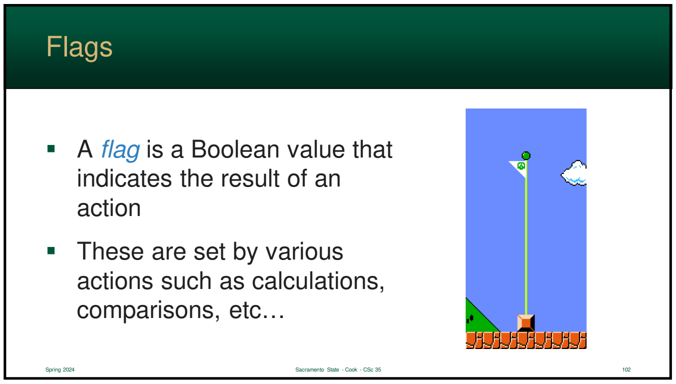
Instruction: Compare

Register, Memory

CMP *arg1* , *arg2*

Immediate, Register, Memory

101



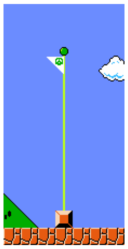
Flags

- A *flag* is a Boolean value that indicates the result of an action
- These are set by various actions such as calculations, comparisons, etc…

102

Flags

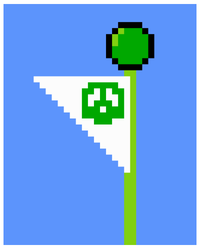
- Flags are typically stored as individual bits in the *Status Register*
- You can't change the register directly, but numerous instructions use it for control and logic



103

Zero Flag (ZF)

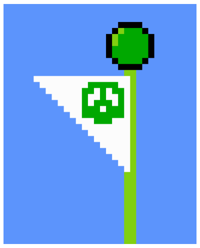
- True if the last computation resulted in zero (all bits are 0)
- For compare, the zero flag indicates the two operands are equal
- Used by quite a few conditional jump statements



104

Sign Flag (SF)

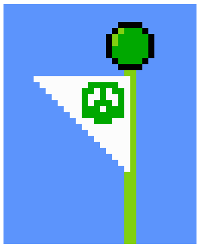
- True of the *most significant bit* of the result is 1
- This would indicate a negative 2's complement number
- Meaningless if the operands are interpreted as unsigned



105

Carry Flag (CF)

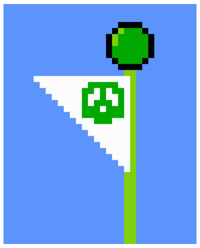
- True if a 1 is "borrowed" when subtraction is performed
- …or a 1 is "carried" from addition
- For unsigned numbers, it indicates:
  - exceeded the size of the register on addition
  - or an underflow (too small value) on subtraction



106

Overflow Flag (OF)

- Also known as "signed carry flag"
- True if the sign bit changed *when it shouldn't have*
- For example:
  - (negative – positive) should be negative
  - a positive result will set the flag
- For signed numbers, it indicates:
  - exceeded the register size
  - i.e. the value was too big/small



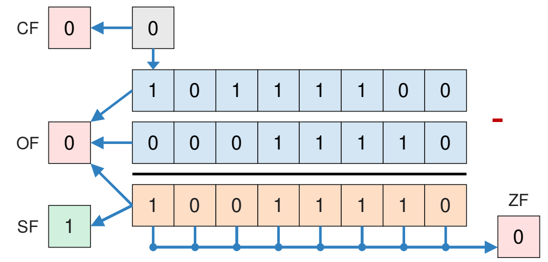
107

x86 Flags Used by Compare

Name	Description	When True
CF	Carry Flag	If a bit was "carried" or "borrowed" during math.
ZF	Zero Flag	All the bits in the result are zero.
SF	Sign Flag	If the most significant bit is 1.
OF	Overflow Flag	If the sign-bit changed when it shouldn't have.

108

-68 vs. 30 (if interpreted as signed)
188 vs. 30 (if interpreted as unsigned)



109

Jump on Equality

Jump	Description	When True
JE	Equal	ZF = 1
JNE	Not equal	ZF = 0

110

Signed Jump Instructions

Jump	Description	When True
JG	Jump Greater than	SF = OF, ZF = 0
JGE	Jump Greater than or Equal	SF = OF
JL	Jump Less than	SF ≠ OF, ZF = 0
JLE	Jump Less than or Equal	SF ≠ OF

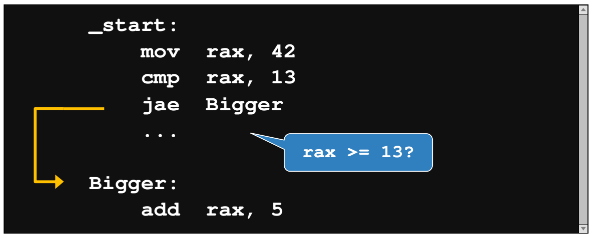
111

Unsigned Jumps

Jump	Description	When True
JA	Jump Above	CF = 0, ZF = 0
JAE	Jump Above or Equal	CF = 0
JB	Jump Below	CF = 1, ZF = 0
JBE	Jump Below or Equal	CF = 1

112

Unsigned Conditional Jump Example



```
_start:
    mov  rax, 42
    cmp  rax, 13
    jae  Bigger
    ...

Bigger:
    add  rax, 5
```

rax >= 13?

113