



Part 4

Arithmetic Logic Unit

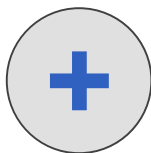


Adding Binary Integers

$$1 + 1 = 10$$

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*



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

3

Adding Base 10 Numbers

$$\begin{array}{r} 1 1 \\ 2 7 8 1 \\ + 3 7 2 1 \\ \hline 6 5 0 2 \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

4

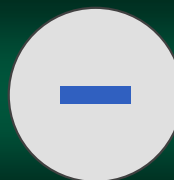
Adding Binary Example

$$\begin{array}{r} 1 1 1 1 \\ 182 \\ 00110110 \\ + 51 \\ 00110011 \\ \hline 11101001 \\ 233 \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

5

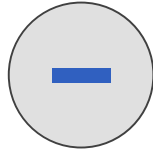


Negative Binary Integers

Have a positive attitude about negatives

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



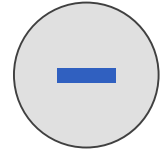
2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

7

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*



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

8

Signed Magnitude

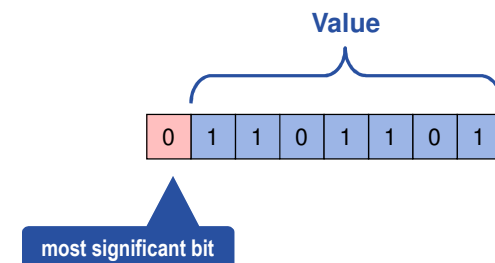
- One approach is to use the most significant bit (msb) to represent the negative sign
- If positive, this bit will be a zero
- If negative, this bit will be a 1
- This gives a byte a range of -127 to 127 rather than 0 to 255

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

9

Signed Magnitude

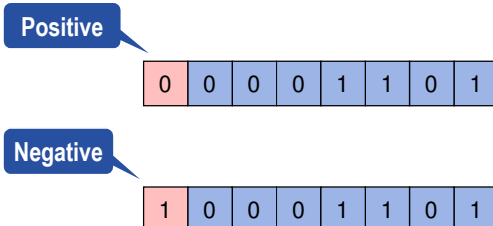


2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

10

Signed Magnitude: 13 and -13



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

11

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

12

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...

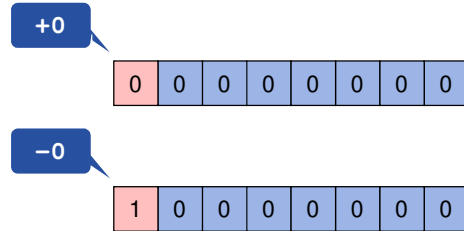
```
if (x == +0 || x == -0)
```

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

13

Oh noes! Two zeros?



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

15

Advantages / Disadvantages

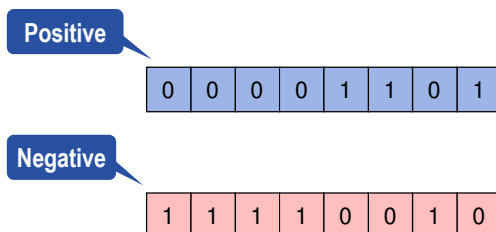
- Advantages over signed magnitude
 - very simple rules for adding/subtracting
 - numbers are simply added: $5 - 3$ is the same as $5 + -3$
- Disadvantages
 - positive and negative zeros still exist
 - so, it's not a perfect solution

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

16

1's Complement: 13 and -13

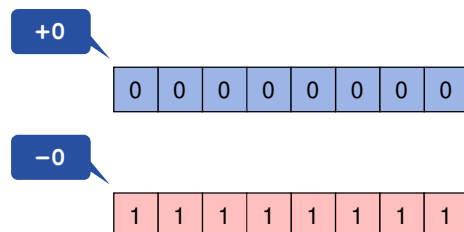


2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

17

1's Complement Has Two Zeros



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

18

2's Complement

- Practically all computers nowadays 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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

19

2's Complement Advantages

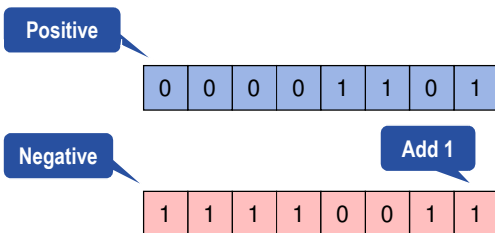
- Since negatives are subtracted from 2^n
 - they can simply be added
 - the extra carry 1 (if it exists) is discarded
 - this simplifies the hardware considerably since the processor only has to add
- The +1 for negative numbers...
 - makes it so there is only one zero
 - values range from -128 to 127

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

20

2's Complement: 13 and -13

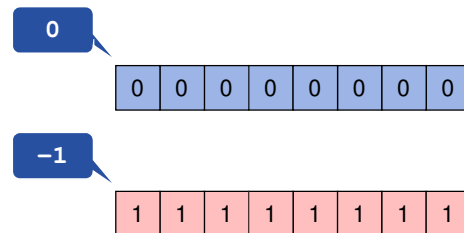


2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

21

Just One Zero!

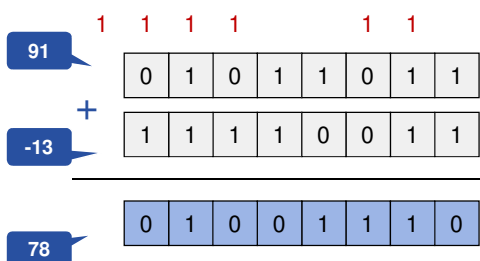


2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

22

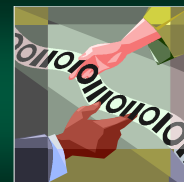
Adding 2's Complement



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

23



Extending Bytes

Converting from 8-bit to 16-bit and more

Extending Unsigned Integers

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



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

25

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: $000456 = 456$



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

26

Unsigned 13 Extended

0 0 0 0 1 1 0 1



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

27

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

28

2's Complement Extended Incorrectly

-13

1 1 1 1 0 0 1 1

243



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

29

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*

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

30

Sign Magnitude Extended: +77

0 1 0 0 1 1 0 1



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

31

Sign Magnitude Extended: -77

1 1 0 0 1 1 0 1



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

32

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

33

2's Complement Extended: +77

0 1 0 0 1 1 0 1



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

34

2's Complement Extended: -77

1 0 1 1 0 0 1 1



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

35

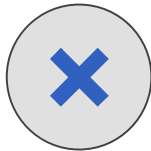


Multiplying
Binary
Numbers

$$11 \times 11 = 1001$$

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



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

37

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?

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

38

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

39

Multiplying: The Bad Way

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

40

Multiplying: The Bad Way

- If one of the operands – A or B – is large, then the computation 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



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

41

Multiplying: The Best Way



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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

42

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

43

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

44

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 0000 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

45

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 0000 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

46

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 1101 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

47

Unsigned Integer: 13×10

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 1101 \\ \hline \end{array}$$

$$\begin{array}{r} + \\ \hline \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

48

Unsigned Integer: 13×10

$$\begin{array}{r}
 1101 \\
 \times 1010 \\
 \hline
 0000 \\
 1101 \\
 0000 \\
 \hline
 \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

49

Unsigned Integer: 13×10

$$\begin{array}{r}
 1101 \\
 \times 1010 \\
 \hline
 0000 \\
 1101 \\
 0000 \\
 \hline
 \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

50

Unsigned Integer: 13×10

$$\begin{array}{r}
 1101 \\
 \times 1010 \\
 \hline
 0000 \\
 1101 \\
 0000 \\
 \hline
 1101
 \end{array}$$

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

51

Unsigned Integer: 13×10

$$\begin{array}{r}
 1101 \\
 \times 1010 \\
 \hline
 0000 \\
 1101 \\
 0000 \\
 \hline
 1101 \\
 \hline
 10000010
 \end{array}$$

130

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

52

Multiplication Doubles the Bit-Count

- When two numbers are multiplied, the product will have twice the number of digits
- Examples:
 - 8-bit \times 8-bit \rightarrow 16-bit
 - 16-bit \times 16-bit \rightarrow 32-bit
- Often processors...
 - will store the result in the original bit-size
 - and flag an overflow if it does not fit

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

53

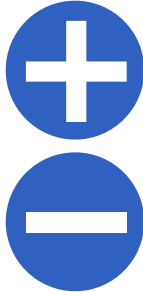


x86
Mathematics

Complex Math is Complex

Add & Subtract

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



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

55

Addition

Immediate, Register, Memory

ADD *value*, *target*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

56

Subtraction

Immediate, Register, Memory

SUB *value*, *target*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

57

Negate (2's complement)

NEG *register*

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

58

Example: Simple Add

```
MOV $17, %rax
ADD $2, %rax
```

Move value into RAX

RAX += 2

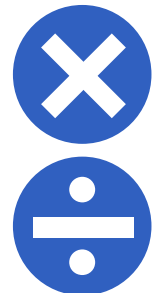
2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

59

Multiplication & Division

- Multiplication and division are far more complex
- The x86 treats this type of math differently than add/subtract
- It requires specific registers to be used: *RAX*, *RDY*



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

60

Multiplication Review

- Remember: when two n bit numbers are multiplied, result will be $2n$ bits
- So...
 - two 8-bit numbers \rightarrow 16-bit
 - two 16-bit numbers \rightarrow 32-bit
 - two 32-bit numbers \rightarrow 64-bit
 - two 64-bit numbers \rightarrow 128-bit



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

61

Multiplication on the x86

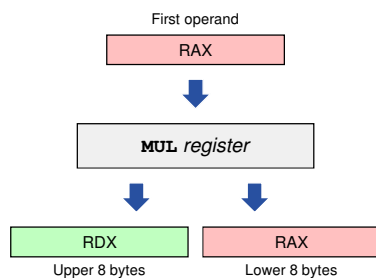
- Instruction inputs are strange
 - first operand is must be stored in **RAX**
 - second operand must be a register (can't be a immediate)
- Result is stored into two registers
 - rax** will contain the lower 8 bytes
 - rdx** will contain the upper 8 bytes

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

62

x86 Multiplication



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

63

Multiply - Unsigned

MUL *operand*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

64

Multiply - Signed

IMUL *operand*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

65

Unsigned Multiply: 1846 by 42

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

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

66

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



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

67

Additional x86 Multiply Instructions

- x86 also contains versions of the IMUL instruction that take multiple operands
- Allows "short" multiplication – just stored in 1 register
- Please note: these do **not** exist for MUL



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

68

IMUL (few more combos)

```
IMUL immediate, reg
IMUL memory, reg
IMUL reg, reg
```

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

69

Signed Multiply: 1846 by 42

```
MOV    $1846, %rax
IMUL   $42, %rax
```

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

70

Division on the x86

- Division on the x86 is very interesting
- Like multiplication, it uses 2 registers
- The dividend (number being divided) uses **two** registers
 - RAX contains the lower 8 bytes
 - RDX contains the upper 8 bytes



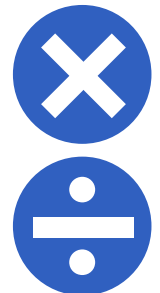
2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

71

Division on the x86

- These two registers are used for the result
- The output contains:
 - RAX will contain the quotient
 - RDX will contain the remainder

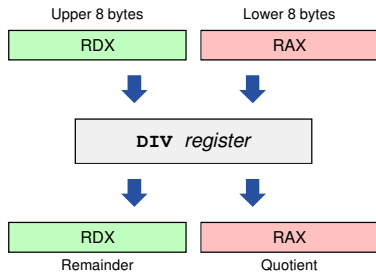


2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

72

x86 Division



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

73

Divide - Unsigned

DIV *denominator*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

74

Divide - Signed

IDIV *denominator*

Register, Memory

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

75

Dividing Rules

- Sign Magnitude
 - create a new sign that is a XOR of the old ones
 - clear the old sign bits and expand with zeros
- 2's complement
 - the numerator must be expanded to the destination size (twice the original)
 - this must be done beforehand
 - otherwise the result will be incorrect

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

76

Division Tips

- Even you don't store anything into RDX, you must clear it
- For signed-division...
 - RDX needs to contain *the sign-extended* value of RAX
 - this is covered in the next section



2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

77

CWD Instruction: 16-Bits: AX:DX

CWD

Convert Word Double

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

78

CDQ Instruction: 32-Bits: EAX:EDX

CDQ

Convert Double Quad

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

79

CQO Instruction: 64-Bits: RAX:RDX

CQO

Convert Quad Oct

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

80

Divide 16-bit: -1846 by 42

```
MOV    $-1846, %ax    #AX is the dividend
MOV    $42, %bx       #Divisor
CWD                    #Extend the sign to DX
IDIV   %bx            #AX gets quotient
                        #DX gets remainder
```

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

81

Divide 64-bit: -1846 by 42

```
MOV    $-1846, %rax   #RAX is the dividend
MOV    $42, %rbx      #Divisor
CQO                    #Extend the sign to RDX
IDIV   %rbx           #RAX gets quotient
                        #RDX gets remainder
```

2/20/2018

Sacramento State - Cook - CSc 35 - Spring 2018

82