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

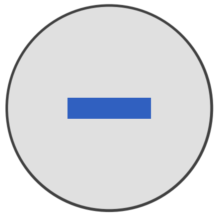
Negative Binary Integers

Have a positive attitude about negatives

6

Negative Binary Numbers

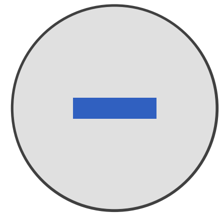
- When we write a negative number, we generally use a "-" as a prefix character
- However, binary numbers can only store ones and zeros



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*



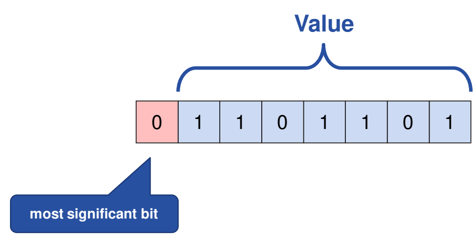
8

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

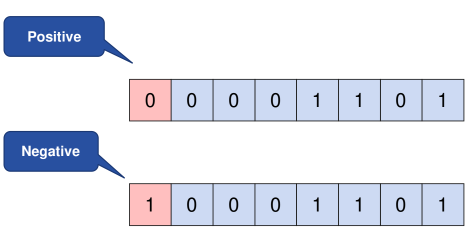
9

Signed Magnitude



10

Signed Magnitude: 13 and -13



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

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)
```


13

Oh noes! Two zeros?

+0

0	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

Positive

0	0	0	0	1	1	0	1

Negative

1	1	1	1	0	0	1	0

17

1's Complement Has Two Zeros

+0

0	0	0	0	0	0	0	0

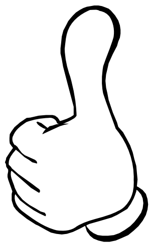
−0

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
Negative (Add 1)	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

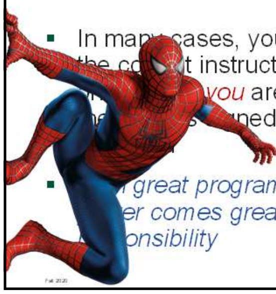
23

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

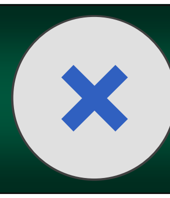
It's Your Responsibility



- In ma... cases, you must use the co... t instruction - based ... *you* are treating ... ned or
- *... great programming ... er comes great ... onsibility*

25

Multiplying Binary Numbers

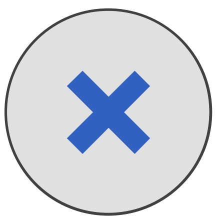


11 × 11 = 1001

26

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



27

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?

28

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

29

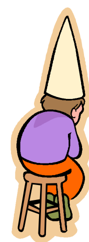
Multiplying: The Bad Way

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


30

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



31

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

32

Unsigned Integer: 13 × 10

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


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


33

Unsigned Integer: 13 × 10

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

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


34

Unsigned Integer: 13 × 10

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

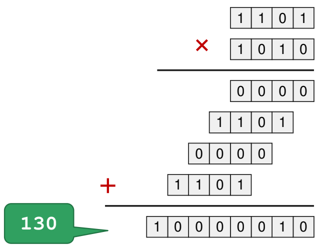

35

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
---------------
```


36

Unsigned Integer: 13 × 10



37

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

38

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

39

x86 Mathematics

Complex Math is Complex

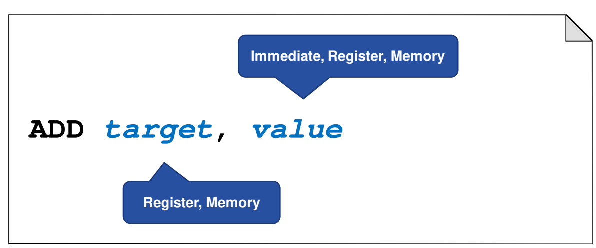
40

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…

41

Addition



42

Subtraction

Immediate, Register, Memory

```
SUB target, value
```

Register, Memory

43

Negate (2's complement)

```
NEG register
```


44

Example: Simple Add

Move value into RAX

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

RAX += 2

45

x86 Multiplication

Complex Math is Complex

46

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

47

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

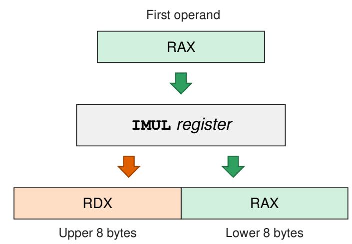
48

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

49

x86 Multiplication



50

Multiply - Signed

```
IMUL operand
```

Register or Memory only

51

Multiply - Unsigned

```
MUL operand
```

Register or Memory only

52

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
```


53

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

54

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

55

IMUL (few more combos)

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

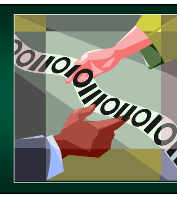

56

Signed Multiply: 1846 by 42

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

This works, but could cause an overflow

57



Extending Bytes

Converting from 8-bit to 16-bit and more

58

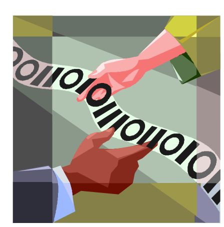
Extending Unsigned Integers

- 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

59

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



60

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

61

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

62

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

63

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*

64

Sign Magnitude Extended: +77

0	1	0	0	1	1	0	1

65

Sign Magnitude Extended: +77

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

66

Sign Magnitude Extended: -77

1	1	0	0	1	1	0	1

67

Sign Magnitude Extended: -77

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

68

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

69

2's Complement Extended: +77

0	1	0	0	1	1	0	1

70

2's Complement Extended: +77

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

71

2's Complement Extended: -77

1	0	1	1	0	0	1	1

72

2's Complement Extended: -77

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

73

x86 Division

Complex Math is Complex

74

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

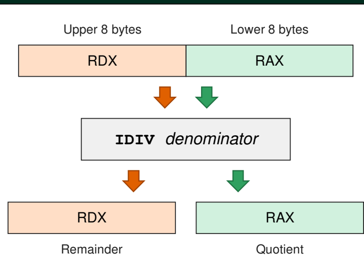
75

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

76

x86 Division



77

Divide - Signed

```
IDIV denominator
```

Register or Memory only

78

Divide - Unsigned

```
DIV denominator
```

Register or Memory only

79

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

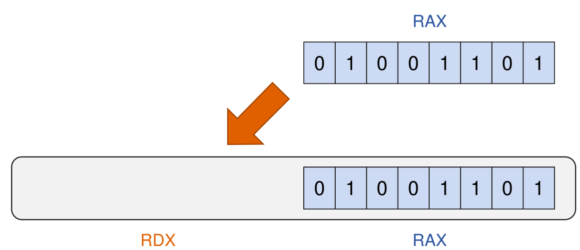
80

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

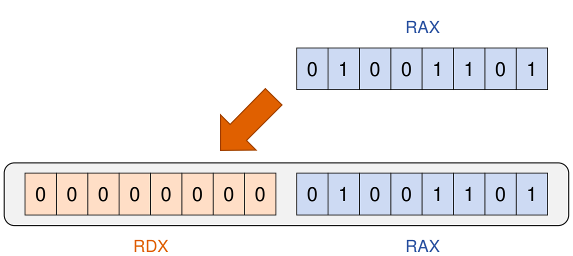
81

Sign Extend Example



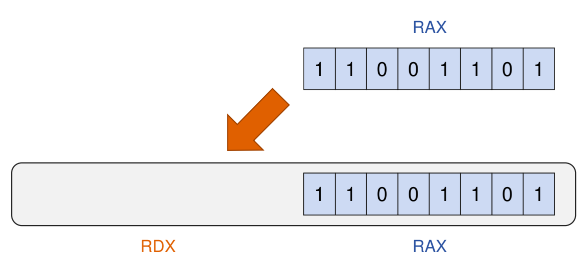
82

Sign Extend Example



83

Sign Extend Example



84

Sign Extend Example

RAX

1	1	0	0	1	1	0	1

RDX								RAX							
1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1

85

CWD (16 bit): Extend AX → DX

CWD

Convert Word Double

86

CDQ (32 bit): Extend EAX → EDX

CDQ

Convert Double Quad

87

CQO (64 bit): Extend RAX → RDX

Use this one!

CQO

Convert Quad Oct

88

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
```


89

How Compare Works

It's all math

90

Behind the scenes…

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

91

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

92

Instruction: Compare

Register, Memory

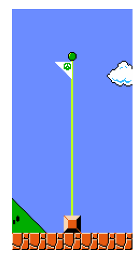
CMP ***arg1*** **,** ***arg2***

Immediate, Register, Memory

93

Flags

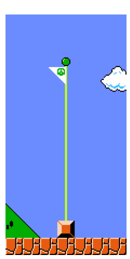
- A *flag* is a Boolean value that indicates the result of an action
- These are set by various actions such as calculations, comparisons, etc…



94

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



95

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

96

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

97

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

98

Overflow Flag (OF)

- Also known as "signed carry flag"
- True if the sign bit changed *when it shouldn't*
- For example:
  - (negative – positive number) should be negative
  - a positive result will set the flag
- For signed numbers, it indicates:
  - exceeded the size of the register on addition
  - or an underflow (too small value) on subtraction

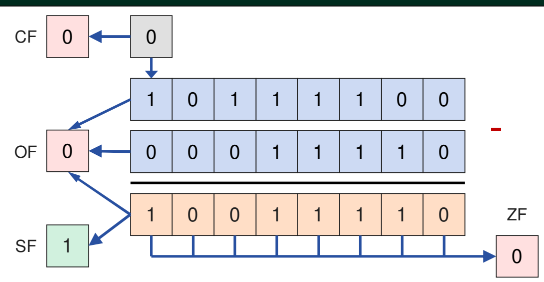
99

x86 Flags Used by Compare

Name	Description	When True
CF	Carry Flag	If an extra 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.

100

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



101

Conditional Jumps

- Conditional jumps (aka *branching*) will only jump if a certain condition is met
- What happens
  - processor jumps if and only if a specific status flag is set
  - otherwise, it simply continues with the next instruction

102

Conditional Jumps

- x86 contains a large number of conditional jump statements
- Each takes advantage of status flags (such as the ones set with compare)
- x86 assembly has several names for the same instruction – which adds readability

103

Jump on Equality

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

104

Conditional Jump Example

```
_start:
    cmp  rax, 13
    je   Equal
    ...
Equal:
    ...
```

rax = 13?

105

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

106

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

107

Unsigned Conditional Jump Example

```
_start:
    mov  rax, 42
    cmp  rax, 13
    jae  Bigger
    ...
Bigger:
    add  rax, 5
```

rax >= 13?

108