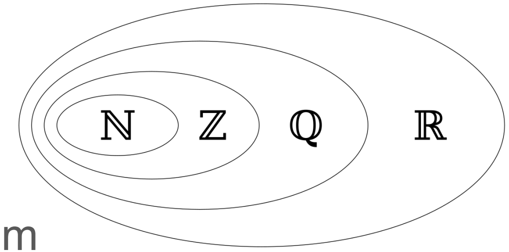
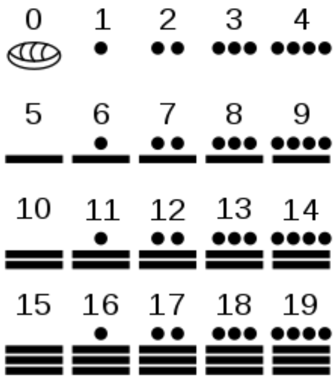
Numbering Systems Review



- Natural numbers: Things that we count and the base system
  - unary: 111 1111   1111111111111, 11101101111=324
  - base 7: Sun, Mon,Tues, … Sat, Sun
  - base 12: Jan, Feb, .. Dec
  - base 20: e.g., the Mayan system
  - base 60: from the Sumerians: used for time, angles, geographical coordinates
  - base 2, 8 (2^3), 16 (2^4)

- Integers: Positive, Zero, and Negative
- Rational: e.g., ⅓, 2.3 but not… π
- Real Numbers: e.g., π
- Complex Numbers: eg., 3 + 2í where í = √-1

- The concept of both $-\infty$ and ∞



Computer Limitations and Representation

- Recall the Universal Computer
  - There is a limited tape size to perform calculation
- Recall the von Neumann and Harvard architecture
  - There is a predefined width to registers and memory
- Abstract representations with limited sizes for:
  - Natural Numbers & Zero: unsigned char, unsigned int
  - Integers: short int, int, long int
  - Rational/Real
    - Fix Point ---
    - Floating Point float, double

- An encoding of each will include one or more of the following:

sign	whole	fractional	expon sign	exponent

Expanded Notation

- Recall from grade school

1234 =	1 x 10^3	
	+ 2 x 10^2	
	+ 3 x 10^1	
	+ 4 x 10^0	

thousands	hundreds	tens	ones
1	2	3	4

Expanded Notation for other Bases

Radix Point

BASE	Columns					
Base 16	4096	256's	16's	1's	1/16	1/256
Base 10	100's	100's	10's	1's	1/10	1/100
Base 8	512's	64's	8's	1's	1/8	1/64
Base 2	8's	4's	2's	1's	1/2	1/4

Expanded Notation

- Recall from grade school

154 =	1 x 10^2	
	+ 5 x 10^1	
	+ 4 x 10^0	

Expanded Notation

- Base 16

  16# 9A

		Base 10 Value
`0x9A =`	`9 x 16^1`	
	`+ A x 16^0`	

- Base 8

  0o232

  8# 232

		Base 10 Value
`0232 =`	`2 x 8^2`	
	`+ 3 x 8^1`	
	`+ 2 x 8^0`	

10	A
11	B
12	C
13	D
14	E
15	F

Expanded Notation

- Base 2: 2# 1001 1010

1001 1010 =	1 x 2^7	
	+ 0 x 2^6	
	+ 0 x 2^5	
	+ 1 x 2^4	
	+ 1 x 2^3	
	+ 0 x 2^2	
	+ 1 x 2^1	
	+ 0 x 2^0	

Real Numbers

- 45.34
- \- 45.34
- 1011110.0100
- \- 1011110.0100

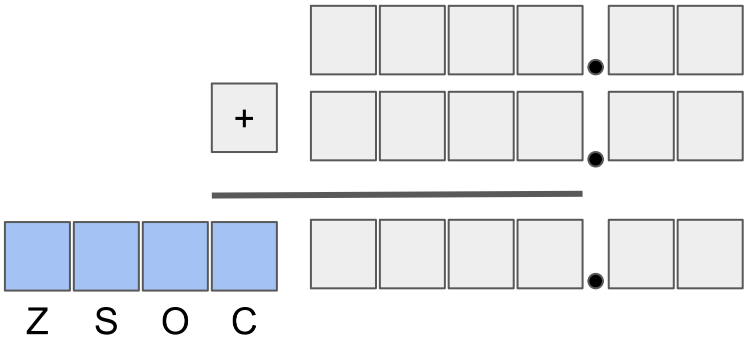
radix point

1000 ← 1000000.1011110 → 01

- But what about may limitations on the computer!

Fixed Point Numbers

- There is a need to represent Rational numbers
- Consider operations on US currency. (dollars and cents)
  - Assign the decimal point (radix) after position 2
  - Provide a register of size 6



Scientific Notation

- All numbers represented as: $m \times 10^N$
- Simplifies operations on large and small numbers.
  - Distance between sun and earth: 92,000,000 = 9.2×10^7
  - Distance between sun and mars: 143,000,000 = 1.43×10^8
- Floating point representation
  - a representation of scientific notation
  - introduces the notion of infinity, and NaN (0 / 0 = ?, 0 x infinity = ?)
- Pieces of the representation
  - Assume a register of size 16
  - A half precision: float16
  - $-1.1011101 \times 2^{1001}$

$$\begin{array}{rr} & 14.3 \times 10^7 \\ - & \underline{9.2 \times 10^7} \\ & 5.1 \times 10^7 \end{array}$$

Floating Point Encoding

always 1: so we don't store it

Original number: - 0.000100101

Recall Scientific Notation: $\boxed{-}\,\boxed{1}.\boxed{0100101} \times 2^{\boxed{-\,100}}$

- Components to Encode
  - sign: negative
  - significant or the mantissa: 0100101
  - exponent: - 100
    - Issue: negative exponents
    - Solution: store values with a bias
- Bias:
  - Shift all numbers along the number line by N
  - Typically it is half the range:
    - `3 bits ->                011 == 3`
    - `5 bits ->             0 1111 == 15`
    - `8 bits ->          0111 1111 == 127`
    - `11 bits ->     011 1111 1111 == 1023`

Number	
-4	
-3	
-2	
-1	
0	000
1	001
2	010
3	011

Floating Point Encoding

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Recall Scientific Notation: $- 1.0100101 \times 2^{-4}$

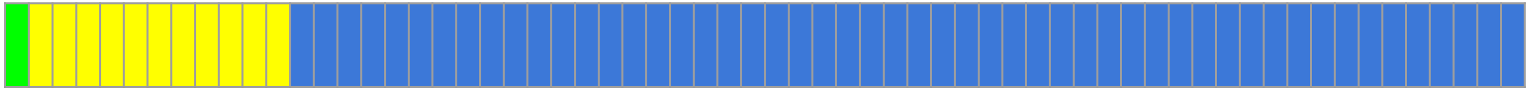
- Formats:
  - float16 (half): `1 + 5 + 10 = 16,      0 1111 = 15`

s	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m

  - float32 (single): `1 + 8 + 23 = 32,    0111 1111 = 127`

s	e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m

  - float64 (double): `1 + 11 + 52 = 64,  01 1111 1111 = 1023`



Floating Point Encoding

Recall Scientific Notation: - 1.0100101 x 2^{-4}

- Consider a new format: c122f8 (quarter)
  - c122f8 (quarter): `1 + 3 + 4 = 8,` `011 = 3`

- Components
  - sign: 1
  - mantissa: 0100~~10~~ ; Drop the extra bits.
  - expon: -4 + 3 = -1 Opps, number is two small.

1	e	e	e	0	1	0	0

Floating Point Encoding

Recall Scientific Notation: $- 1.0100101 \times 2^{-4}$

- Half Precision
  - float16 (half): `1 + 5 + 10 = 16,` `0 1111 = 15`

- `Components`
  - `sign: 1`
  - `mantissa: 010010 ; fill in least significant bits with zero (0)`
  - `expon: -4 + 15 = 11 → 1011`

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

Floating Point Encoding

Recall Scientific Notation: $- 1.0100101 \times 2^{-4}$

- Single Precision
  - float32 (single): `1 + 8 + 23 = 32,      0111 1111 = 127`
- Components
  - sign: 1
  - mantissa: 010010 ; fill in least significant bits with zero (0)
  - expon: -4 + 127 = 123 → 0111 1011

1	0	1	1	1	1	0	1	1	0	1	0	0	1	0	0	0	0	m	m	m	m	m	m	m	m	m	m	m	m	m	m

Base *N* to Base 10

Also See Expanded Notation

Algorithm

- set v = 0
- For each digit (from left to right)
  - v = v * base; # Multiple by the base
  - v = v + digit; # Add the next digit
- print v

Consider: 145 == 2# 10010001

$v{:}\ \boxed{} = \boxed{v} * 2 + \boxed{d}$

1 0 0 1 0 0 0 1

v:		=		* 2 +	
v:		=		* 2 +	
		=		* 2 +	
		=		* 2 +	
		=		* 2 +	
		=		* 2 +	
		=		* 2 +	
		=		* 2 +	

Base Conversion

Base 10 to Base 2

- The whole portion is divided by the new base, repeatedly
  - Dividend / Divisor = (Quotient, Remainder)
  - The concatenation of the Remainders provide you with the final digits
- The fraction portion is multiplied by the new base, repeatedly
  - Multiplier * Multiplicant = (Overflow, Product)
  - The concatenation of the Overflows provide you with the final digits
- Consider the examples via the spreadsheet: Base Conversion

Base: 2, 8, 16

1. Convert each digit to binary
2. Merge the bits
3. Rechunk
4. Convert each chunk to the appropriate digit

- Consider the examples via the spreadsheet: Rechunk

Decimal Real to Binary Real

1. Split the number at the radix point: *whole . fractional*

2. With the whole part,

```
number = whole
while (number != 0 ){
    number = number / 2
    push ( number % 2 )
}
pop_all();
```

3. With the fractional part

```
max = 10 ** stringlength(fractional)
number = fractional
while (number != 0 ) {
    number = number * 2
    if ( number > max ) {
       emit 1
       number = number - max
    } else {
       emit 0
    }
}
```

4. Put the two pieces together

Whole Part: Decimal Real to Binary Real

1. Example 39.234

2. With the whole part,

```
number = whole
while (number != 0 ){
    number = number / 2
    push ( number % 2 )
}
pop_all();
```

number: 39

number = 39 / 2	→ 19
push (39 % 2)	→ 1
number = 19 / 2	→ 9
push(19 % 2)	→ 1
number = 9 / 2	→ 4
push(9 % 2)	→ 1
number = 4 /2	→ 2
push(4 % 2)	→ 0
number = 2 / 2	→ 1
push (2 % 2)	→ 0
number = 1 / 2	→ 0
push (1 % 2)	→ 1

Fractional Part: Decimal Real to Binary Real

1. Example 39.234

3. With the fractional part

```
max = 10 ** numdigits(fractional)
number = fractional
while (number != 0 ) {
    number = number * 2
    if ( number > max ) {
       emit 1
       number = number - max
    } else {
       emit 0
    }
}
```

max = 10 ** |234| == 1000

number = 234

number = number * 2 = 468

number = 468 * 2 = 936

number = 936 * 2 = 1872 - 1000 = 872

number = 872 * 2 = 1744 - 1000 = 744

number = 744 * 2 = 1488 - 1000 = 488

number = 488 * 2 = 976

number = 976 * 2 = 1952 - 1000 = 952

0011101

Decimal Real to Binary Real

1. Split the number at the radix point: *whole . fractional*

2. With the whole part,

```
whole = number
while (number != 0 ){
    number = number / 2
    push ( number % 2 )
}
pop_all();
```

3. With the fractional part

```
max = 10 ** ( | fractional | )
fractional = number
while (number != 0 ) {
    number = number * 2
    if ( number > max ) {
       emit 1
       number = number - max
    } else {
       emit 0
    }
}
```

4. Put the two pieces together

100111	.	0011101

Whole Part: Decimal Real to Binary Real

1. Example 45.45

2. With the whole part,

```
whole = number
while (number != 0 ){
    number = number / 2
    push ( number % 2 )
}
pop_all();
```

101101

number: 45

number = 45 / 2 → 22
push (45 % 2) → 1
number = 22 / 2 → 11
push (22 % 2) → 0
number = 11 / 2 → 5
push (11 % 2) → 1
number = 5 / 2 → 2
push (5 % 2) → 1
number = 2 / 2 → 1
push (2 % 2) → 0
number = 1 / 2 → 0
push (1 % 2) → 1
number = 0 / 0

1
0
1
1
0
1

Fractional Part: Decimal Real to Binary Real

1. Example 45.45

3. With the fractional part

```
max = 10 ** ( | fractional | )
fractional = number
while (number != 0 ) {
    number = number * 2
    if ( number > max ) {
        emit 1
        number = number - max
    } else {
        emit 0
    }
}
```

max = 10 ** |45| == 100

number = 45

number = number * 2 = 90

number = number * 2 = 180 - 100 = 80

number = 80 * 2 = 160 - 100 = 60

number = 60 * 2 = 120 - 100 = 20

number = 20 * 2 = 40

number = 40 * 2 = 80

number = 80 * 2 = 160 = 100 = 60

0111001 111001

Decimal Real to Binary Real

1. Split the number at the radix point: *whole . fractional*

2. With the whole part,

```
whole = number
while (number != 0 ){
    number = number / 2
    push ( number % 2 )
}
pop_all();
```

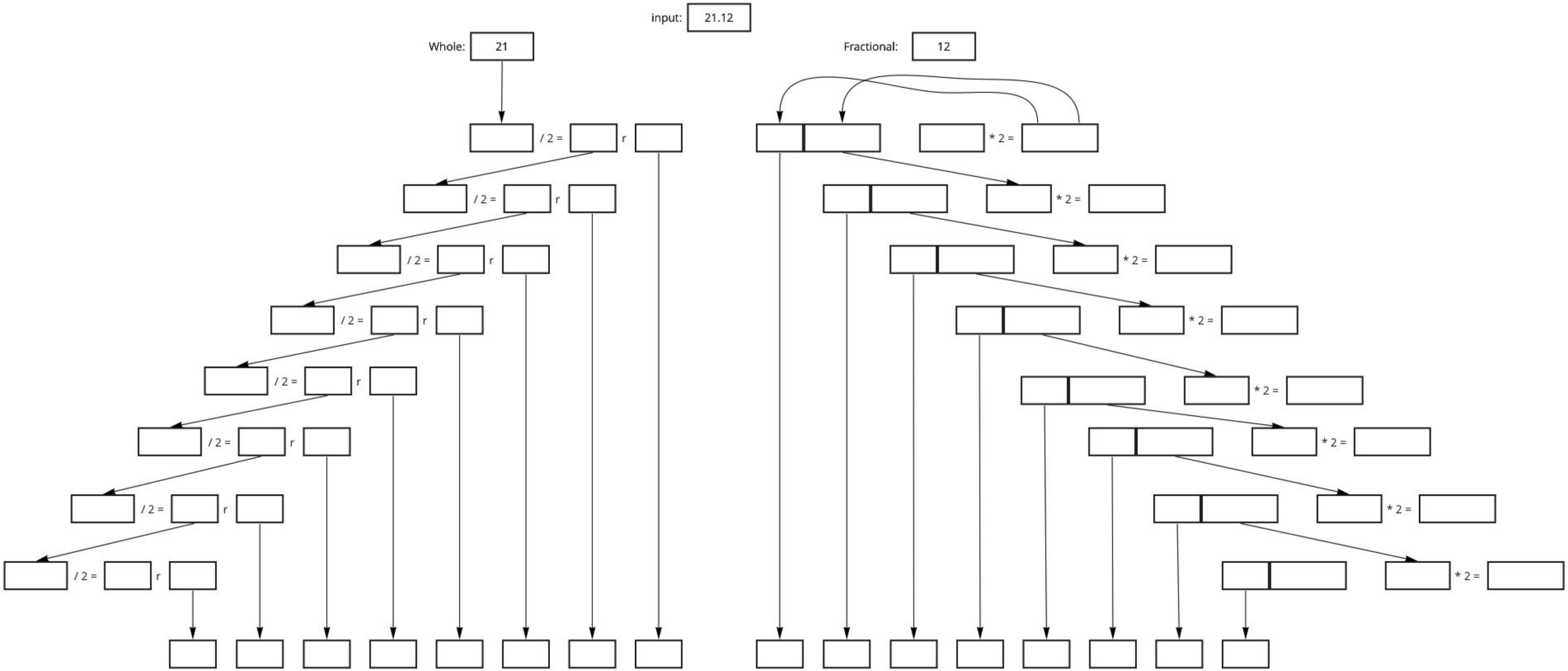
3. With the fractional part

```
max = 10 ** ( | fractional | )
fractional = number
while (number != 0 ) {
    number = number * 2
    if ( number > max ) {
       emit 1
       number = number - max
    } else {
       emit 0
    }
}
```

4. Put the two pieces together

101101	.	0111001 111001

Real: Decimal to Binary (Confusing?)



24567865438 --9's → 75432134561

Method of Complements

- A technique to encode both positive and negative numbers
  - uses the same algorithm to perform addition
  - subtraction perform my addition of complements

- Complement: *a thing that completes or brings to perfection*
- Radix 10: *(the radix or base is the number of unique digits to represent a number)*
  - *10's complement*
    - 7 + x = 10 : x is the 10s complements of 7 x = 3
    - 46 + y = 100 : y is the 10s complements of 46 y = 54
  - 9's complement
    - 7 + a = 9 : a is the 9s complements of 7 a = 2
    - 46 + b = 99 : b is the 9s complements of 46 b = 53

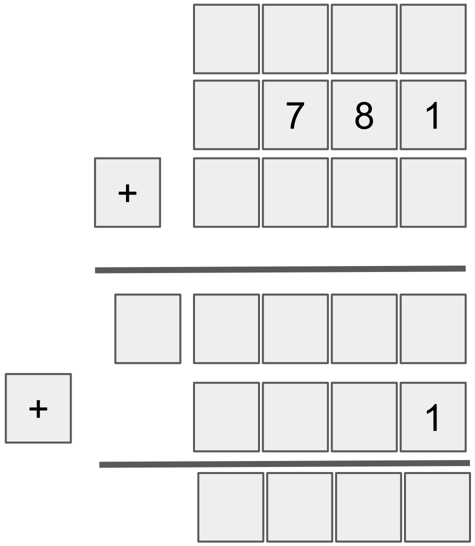
- The math:

2nd Grade	10's complement	9's complement
$$\begin{array}{r} 45 \\ \underline{-\ 11} \\ 34 \end{array}$$	$$\begin{array}{r} 45 \\ \underline{+\ 89} \\ \cancel{1}34 \end{array}$$	$$\begin{array}{r} 45 \\ \underline{+\ 88} \\ \cancel{1}33 \end{array} + 1 = 34$$

Algorithm: Subtraction via Complements

- Example: 873 - 218

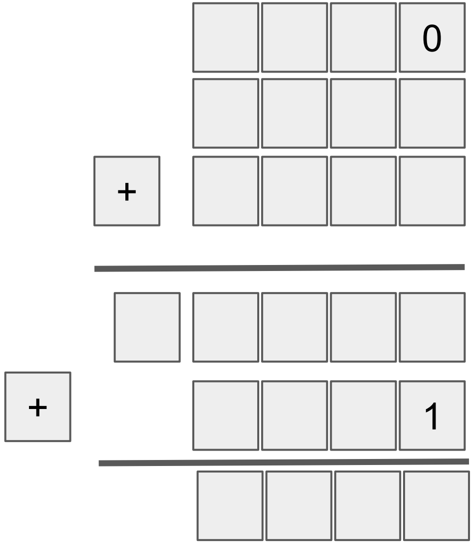
1. Take the nines complement of the subtrahend (218)
2. Add the complement to the minuend (873)
3. Drop the leading "1" is dropped.
4. Add 1



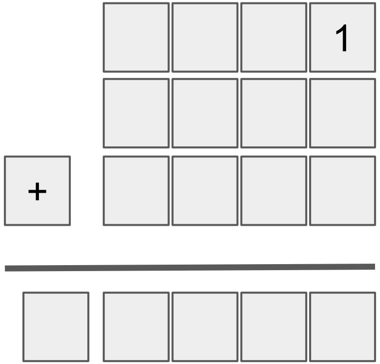
Algorithm: Subtraction via Complements

- Example: 873 - 218

1. Take the nines complement of the subtrahend (218)
2. Add the complement to the minuend (873)
3. Drop the leading "1" is dropped.
4. Add 1



- Optimization:



Method of Complements

- A technique to encode both positive and negative numbers
    - uses the same algorithm to perform addition
    - subtraction perform my addition of complements

- Recall: 10s complement = 9's complement + 1 (*Radix 10)*
        - *9's complement can be performed on each individual digit*
- Hence: 2's complement = 1's complement + 1 (*Radix 2)*

- Radix 2: *(A special case)*
    - 2's Complement: take the 1's complement and add 1
        - 0101 + 1001 = 1000
        - 10111 + 01001 = 10000
    - 1's Complement: take each bit and take its complement
        - 0101 + 1010 = 1111 + 1 = ~~1~~ 0000
        - 10111 + 01000 = 11111 + 1 = ~~1~~ 00000

Method of Complements

A technique to encode both positive and negative numbers

- MSB used to denote the sign bit (0 positive, 1 negative)
- Table assumes a 4-bit represent

Use 1's complement to represent negative numbers

- Divide the number range in half
- Encode a positive and a negative value for each number
- Pros/cons:
  - ease to compute
  - positive and negative representations of zero

1's Complement

	Positive	Negative
0	`0000`	`1111`
1	`0001`	`1110`
2	`0010`	`1101`
3	`0011`	`1100`
4	`0100`	`1011`
5	`0101`	`1010`
6	`0110`	`1001`
7	`0111`	`1000`

Method of Complements

A technique to encode both positive and negative numbers

- MSB used to denote the sign bit (0 positive, 1 negative)
- Table assumes a 4-bit represent

Use 1's complement to represent negative numbers

- Divide the number range in half
- Encode a positive and a negative value for each number
- Pros/cons:
  - ease to compute
  - positive and negative representations of zero

Use 2's complete to represent negative numbers

- Hold Zero as special
- Fold the resulting range to assign the result
- Pros/cons:
  - Not symmetric: extra negative number
  - Need to add one to each negative number
  - Consider -7 - 1 ==   1001 -1 = 1000

2's Complement

	Positive	Negative
0	0000	
1	0001	1110 + 1 = 1111
2	0010	1101 + 1 = 1110
3	0011	1100 + 1 = 1101
4	0100	1011 + 1 = 1100
5	0101	1010 + 1 = 1011
6	0110	1001 + 1 = 1010
7	0111	1000 + 1 = 1001
8		1000

Comparison of 1's and 2's Complement

1's Complement

	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

2's Complement

	Positive	Negative
0	0000	
1	0001	1110 + 1 = 1111
2	0010	1101 + 1 = 1110
3	0011	1100 + 1 = 1101
4	0100	1011 + 1 = 1100
5	0101	1010 + 1 = 1011
6	0110	1001 + 1 = 1010
7	0111	1000 + 1 = 1001
8		1000

Algorithm: Subtraction via Complements

- Example: 13 - 9 == 13 + -9

1. Convert 13 and 9 into binary (01101 & 01001)
2. Take the 2's complement of the subtrahend (9)
   - 01001 -> 10110 + 1 = 10111
3. Add the complement to the minuend
4. Drop the leading "1" is dropped.

- ~~Optimization~~: Addition of adding one is baked in!

	1	1	1	1	0
	0	1	1	0	1
+	1	0	1	1	1
~~1~~	0	0	1	0	0

Method of Complements

- A technique to encode both positive and <u>negative</u> numbers
  - uses the same algorithm to perform addition
  - subtraction perform my addition of complements

- Recall: 10s complement = 9's complement + 1 (*Radix 10)*
- Hence: 2's complement = 1's complement + 1 (*Radix 2)*

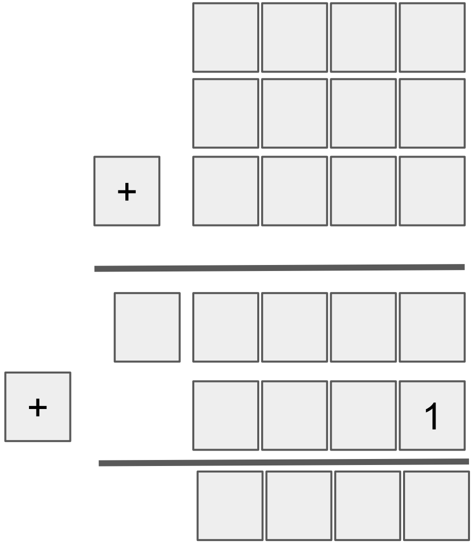
- Radix 2:
  - 2's Complement: take the 1's complement and add 1
    - 0101 + 1001 = 1000
    - 10111 + 01001 = 10000
  - 1's Complement: take each bit and take its complement
    - 0101 + 1010 = 1000
    - 10111 + 01000 = 10000

- Negative numbers are stored as 2's complement numbers (Assume 8 bit quantity)
  - Example: -28
  - Convert 28 to binary :
  - Flip all of its bits :
  - Add 1 :

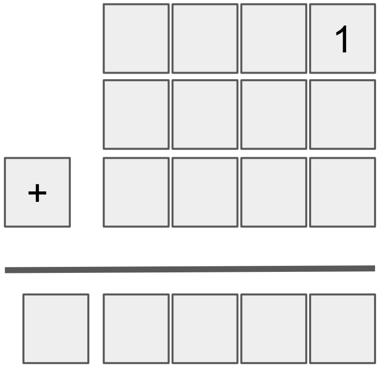
Algorithm: Subtraction via Complements

- Example: 13 - 9

1. Convert 13 and 9 into binary
2. Take the 1s complement of the subtrahend (9)
3. Add the complement to the minuend
4. Drop the leading "1" is dropped
5. Add 1



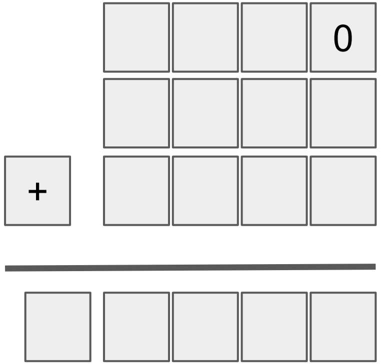
- Optimization:



Algorithm: Subtraction via Complements

- Example: 13 - 9

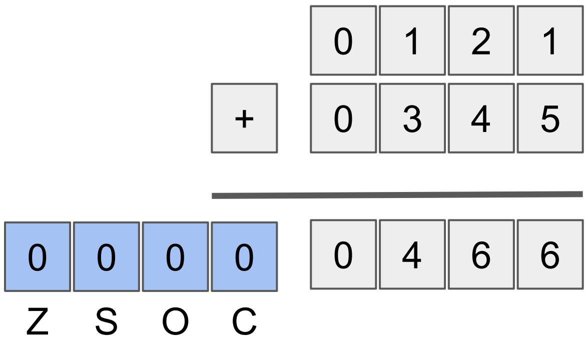
1. Convert 13 and 9 into binary
2. Take the 2s complement of the subtrahend (9)
3. Add the complement to the minuend
4. Drop the leading "1" is dropped.

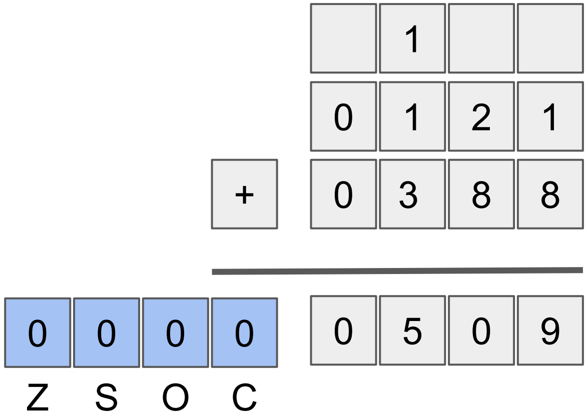


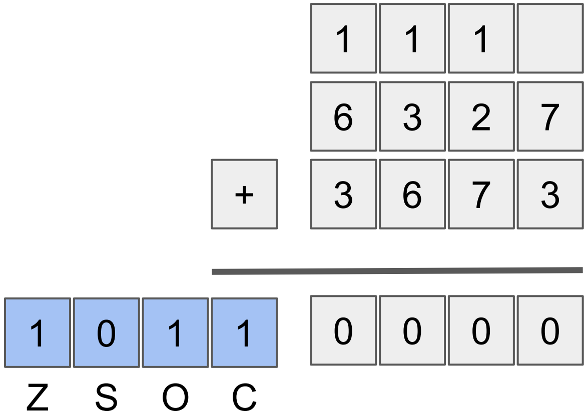
- ~~Optimization~~: Addition of adding one is baked in!

Review of Mathematical Operations:

- First, introduce some status values:
  - Zero, Sign, Overflow, Carry
- Assume a register of size 4:







In Binary

A + B = C S

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

					1	1	0	0
					0	1	1	0
			+		0	0	1	1
0	0	0	0		1	0	0	1
Z	S	O	C					

					1	1	1	0
					1	1	0	1
			+		1	0	1	1
0	0	1	1		1	0	0	0
Z	S	O	C					

					1	1	0	0
					1	0	1	1
			+		0	1	1	0
0	0	1	1		0	0	0	1
Z	S	O	C					

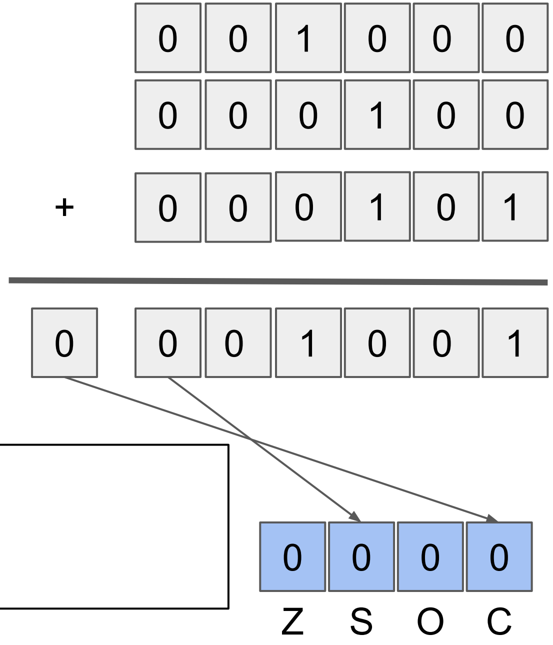
Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
   - if negative result: compute 2's complement first

➢ 4 + 5:
- 7 + - 6:
- -3 + 5:
- -18 - 4:

```
2. 000100, 000101
5. 9 (check)
```



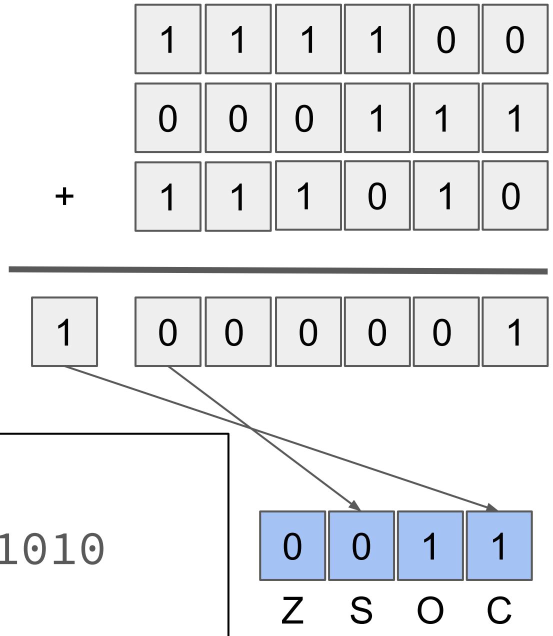
Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
   - if negative result: compute 2's complement first

- 4 + 5:
- ➢ 7 + - 6:
- -3 + 5:
- -18 - 4:

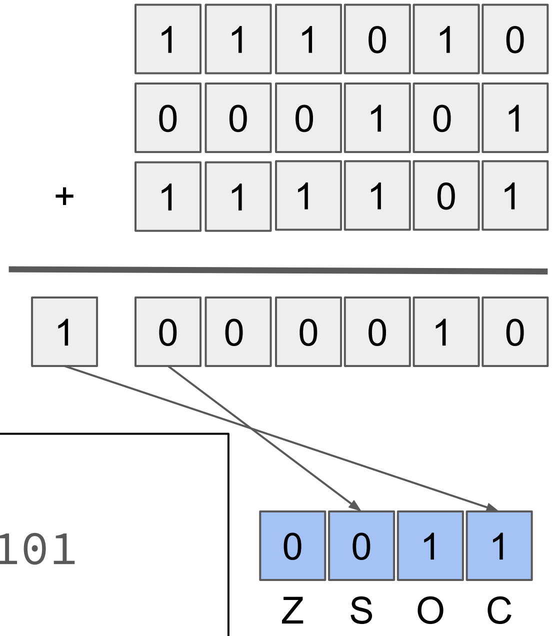
```
2. 000111, - 000110
3. -000110 = 111001+1 = 111010
5. 1 (check)
```



Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
   - if negative result: compute 2's complement first



- 4 + 5:
- 7 - 6:
- ➢ (-3) + 5:
- -18 - 4:

```
2.-000011, 000101
3.-000011 = 111100+1 = 111101
5.2 (check)
```

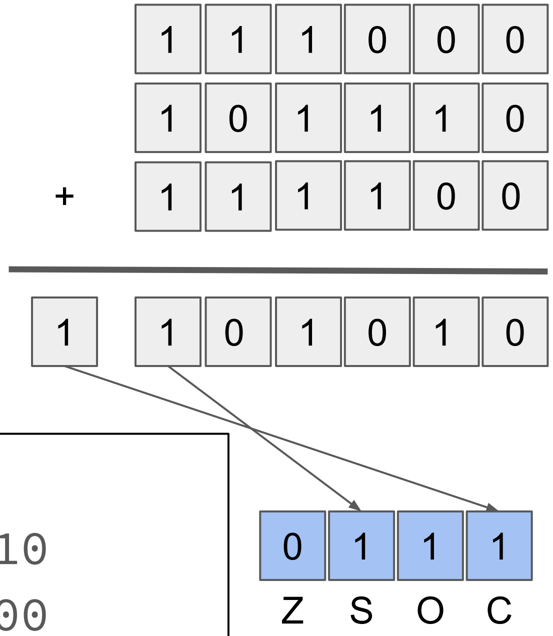
Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
    - if negative result: compute 2's complement first

- 4 + 5:
- 7 - 6:
- -3 + 5:
- ➢ (-18) + (-4):

```
2.-010010, - 000100
3.-010010 = 101101+1= 101110
  -000100 = 111011+1= 111100
5. 101010 =-010101+1= -010110
   -22 (check)
```



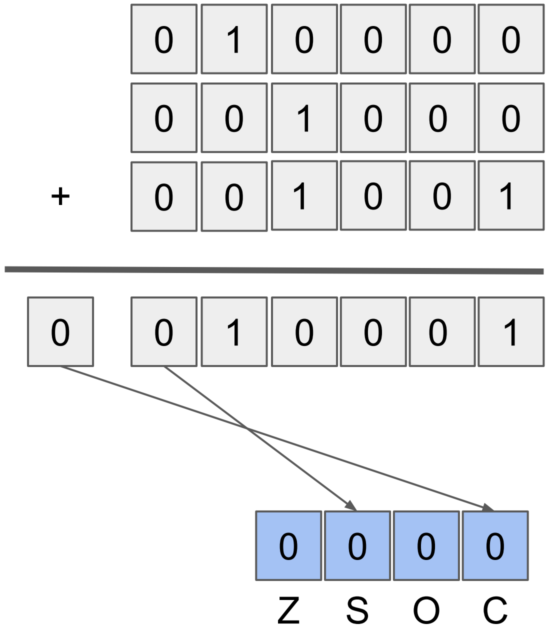
Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
   - if negative result: compute 2's complement first

➢ 8 + 9:
- 7 - 4
- -5 + 2
- -16 - 3:

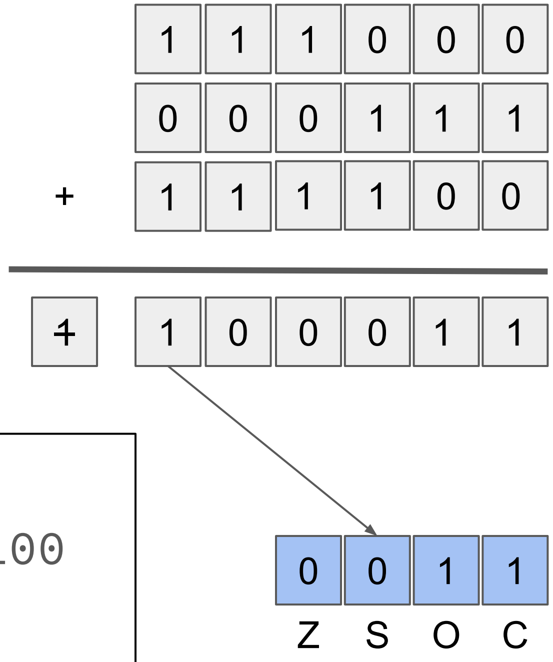
```
2. 1000 + 1001
5. 17 (check!)
```



Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
   - if negative result: compute 2's complement first



- 8 + 9
- 7 + (- 4):
- -5 + 2
- (-16) + (-3)

```
2. 000111 + -000100
3.-000100 = 111011+1 = 111100
5. 3 (check!)
```

Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. convert result to decimal
    - if negative result: compute 2's complement first

- 8 + 9
- 7 - 4
- ➢ (-5) + 2:
- -16 - 3

```
2. -000101 +  000010
3. -000101 =  111010+1 = 111011
5. 1111101 = -000010+1 = 000011
   = -3  (check!)
```

	0	0	0	1	0	0
	1	1	1	0	1	1
+	0	0	0	0	1	0
~~0~~	1	1	1	1	0	1

Z	S	O	C
0	1	0	0

Practice: Addition and Subtraction

Steps:

1. transform the operation to addition
2. convert abs(X) to binary
3. compute 2's complement for negative numbers
4. perform binary addition
5. Validate step: convert result to decimal
    - if negative result: compute 2's complement first

	1	0	0	0	0	0
	1	1	0	0	0	0
+	1	1	1	1	0	1
~~1~~	1	0	1	1	0	1

- 8 + 9
- 7 - 4
- -5 + 2
- ➢ (-16) + (-3)

```
2. -010000 + -000011
3. -010000 =  101111+1 = 110000
   -000011 =  111100+1 = 111101
5.  101101 = -010010+1 = -010011
   = -19 (check)
```

Z	S	O	C
0	1	1	1

BCD: Addition

- Addition performed on the nibble level: 6+7

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

if (overflow or invalid code) then

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

N	Code	N	Code
0	0000	8	1000
1	0001	9	1001
2	0010		1010
3	0011		1011
4	0100		1100
5	0101		1101
6	0110		1110
7	0111		1111

BCD: Binary Coded Decimal

- Another encoding for numbers,
  where precision is required
- Four bits are used to encode each digit
- Perform addition per nibble
- Example: 246 + 127

							0				1				
				0	0	1	0	0	1	0	0	0	1	1	0
			+	0	0	0	1	0	0	1	0	0	1	1	1
0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	1

N	Code	N	Code
0	0000	8	1000
1	0001	9	1001
2	0010		1010
3	0011		1011
4	0100		1100
5	0101		1101
6	0110		1110
7	0111		1111

BCD: Addition

- Addition performed on the nibble level: 7+9

	1	1	1	0
	0	1	1	1
+	1	0	0	1
1	0	0	0	0

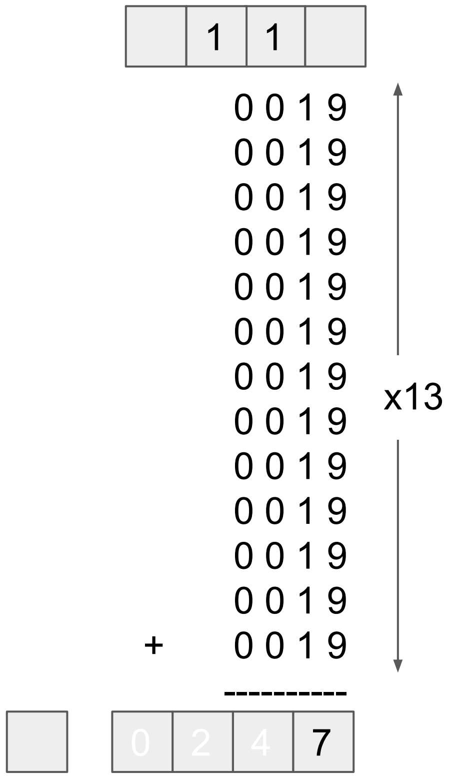
if (overflow or invalid code) then

+	0	1	1	0
1	0	1	1	0

N	Code	N	Code
0	0000	8	1000
1	0001	9	1001
2	0010		1010
3	0011		1011
4	0100		1100
5	0101		1101
6	0110		1110
7	0111		1111

Comments on Multiplication

- Consider: 9 x 13 = ? 117
- What is carry value for the 10's column?



Algorithm for Multiplication

```
product = 0;
for (d = 0 ; d < 4 ; d ++ ) {
    if (B[d] == 1) {
        product += A;
    }
    A = A << 1;
}
```