Introduction to Computer Systems

Course introduction

The Instructor – Dr. Danny Seidner

- **You may call me Danny (preferable) or Dr. Seidner**

- **What is the course about?**
  - It is the basics of computer systems
  - We will learn how numbers are represented in computers
  - We will learn how code is represented in computers
  - We will show the process of writing code, compiling it to what is called “machine language” and how a CPU, the main component in a computer, performs the code
  - These basics are really **important** for **you** – It gives you a look at the “Big Picture” and explains a lot on computers

- **The course is given in Zoom meeting**
  - Starts on 11:00 on Fridays
  - You can open your microphones and ask questions anytime
  - It is the interest of all of us that you will ask questions
  - I will have a break here and there. You can ask for a break.

More technicalities

- **Recitations are also given in Zoom meeting**
- **Course material**
  - Slides are available in the course web site
  - Messages will be given in the course web site – "לוח הודעות" – You MUST check the messages!
  - Additional material (exercises, solutions, material for exam) are already there
- **We have 10 lectures**
  - Actually these are 10 topics – and we will be flexible about how many topics per lecture
  - We might require more than one Zoom Meeting to cover a topic. Sometimes – less than one lecture.
  - We start today with representing numbers (~3 lectures), than we will discuss Boolean Algebra, build components, build arithmetic circuits and registers and then, define a CPU & implement it
  - On the way there we will learn Assembly language and discuss compilation – important stuff!
- **Course grading**
  - The exam grade is the course grade
- **Any questions?**
  - A good opportunity to exercise opening the microphone and ask something …

Introduction to Computer Systems

Lecture #1

Unsigned Fixed Point Numbers

נקודה קבועה

Lecture#1 Agenda - Fixed point Numbers

- **Unsigned integers - שלמים ללא סימן**
  - Base 10 & examples
  - Base 2 & examples
  - Conversions to decimal
  - Conversions from decimal
  - Base K equations & range of possible numbers
  - Hexadecimal numbers
- **Fractions - שברים**
  - Fraction equations & range of possible numbers
  - Examples
  - Quantization - כימות
  - 2/3 in decimal
  - 2/3 in binary
  - How to use in C [Advanced topic]

Unsigned integers

- We discuss unsigned numbers (for now)
- These are 0,1,2,3,… etc.
- We also discuss fractions, e.g., 0.15, 32.33, etc.

Base 10 integers – Decimal numbers - מס' עשרוניים

- Here is a base 10 example:

$$\overset{3}{5}\overset{2}{0}\overset{1}{9}\overset{0}{7}$$

$$[5097]_{10} = 5\cdot10^3+0\cdot10^2+9\cdot10^1+7\cdot10^0$$

- Every digit has a different weight that depends on its position
  - First position (starting from the right) has a weight of $10^0=1$
  - Second position has a weight of $10^1=10$
  - 3rd position has a weight of $10^2=100$
  - i^{th} position has a weight of 10^{i-1} [or if we start the index from 0, then 10^i]

- We describe the number $[X]_{10}$ as a string of decimal digits:

$$[X]_{10} = [X_{n-1}, X_{n-2}, \dots X_2, X_1, X_0]_{10} \qquad (X_i \in \{0,9\})$$

Base 10 integers

- $[X]_{10} = [X_{n-1}, X_{n-2}, \dots X_2, X_1, X_0]_{10}$ $(X_i \in \{0,9\})$
- The value of $[X]_{10}$ is given by the formula:

$$X = X_{n-1} \cdot 10^{n-1} + X_{n-2} \cdot 10^{n-2} + \dots + X_2 \cdot 10^2 + X_1 \cdot 10^1 + X_0 \cdot 10^0$$

or

$$X = \sum_{i=0}^{n-1} X_i \cdot 10^i$$

Range of base 10 integers

n ספרות

- The range that can be represented by n digits is 0-(10^n-1)

- With 3 digits we can represent numbers in the range of 0-999

- We have 10^3 possible digit combinations

000

001

002

So the values will be 0-999

999

Base 2 integers – Binary numbers - מס' בינאריים

bit = סיבי"ת = ספרה בינארית

8 4 2 1

2^3 2^2 2^1 2^0

- $[X] = [X]_2 = [X_{n-1}, X_{n-2}, \ldots X_3, X_2, X_1, X_0]$ $X_i \in \{0,1\}$ called a bit
- We use Decimal numbers in daily life probably since we have ten fingers and started counting with fingers
- Computers use base 2 since it is easy to build circuits that are "ON" or "OFF" which means only 2 combinations. And, the circuits almost do not waste energy in these two states (only when the value is changing we waste energy).
- The value of [X] is given by:

4 2 1

$$X = X_{n-1}\cdot 2^{n-1} + X_{n-2}\cdot 2^{n-2} + \ldots + X_2\cdot 2^2 + X_1\cdot 2^1 + X_0\cdot 2^0 = \sum_{i=0}^{n-1} X_i \cdot 2^i$$

Range of binary integers - טווח המספרים

- The range that can be represented by n bits is 0-(2^n-1)
- We have 2^n possible digit combinations

  From [0,0,…,0] till [1,1,…,1]

  So the values will be 0-(2^n-1)

In 4 bits we can represent the numbers 0-15:

Binary numbers example

In 4 bits we can represent the numbers 0-15:

2^3 =8	2^2 =4	2^1 =2	2^0 =1		
[X_3	X_2	X_1	X_0]	value	Hex digit
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F

More example – 8 bit numbers

2^8 = 256 no.-s

[00000000]=0

[00000001]=1

[00000010]=2

[00000011]=3

…

[11111100]=252

[11111101]=253

[11111110]=254

[11111111]=255

Let us watch some specific examples that show how to evaluate binary numbers:

[11111100]= 255-(2^1+2^0)=255-3=252

[1111]= 15

[11110000]=240=255-15

[11110101]=245=240+5

[11110]= 30 = 15<<1=15*2

Shit left means x2

[111100]= 60 = 30<<1=30*2

[11110000]=240=15<<4=15*2^4=15*16

הזזה שמאלה במקום אחד – הכפלה בבסיס (במס' בינאריים – הכפלה ב-2)

Shifting left 1 position means multiply by 2

- $[X] = [X_{n-1}, X_{n-2}, \dots X_2, X_1, X_0]$
- $[Y] = [Y_n, Y_{n-1}, \dots Y_2, Y_1, Y_0] = [X_{n-1}, X_{n-2}, \dots X_2, X_1, X_0, 0]$
- Note that $Y_i = X_{i-1}$ and $Y_0 = 0$

- Thus value of [Y] is given by:

$$Y = \sum_{i=0}^{n} Y_i \cdot 2^i = \sum_{i=1}^{n} Y_i \cdot 2^i = \sum_{i=1}^{n} X_{i-1} \cdot 2^i$$

we will replace i-1 with j ($j=i-1 \Rightarrow i=j+1$)

$$Y = \sum_{j=0}^{n-1} X_j \cdot 2^{j+1} = 2 \cdot [\ \sum_{j=0}^{n-1} X_j \cdot 2^j\ ] = 2 \cdot X$$

This is clear. The weight of each bit is x2. Same in base K: $\boxed{[X]_K << 1 = K \cdot [X]_K}$

This also means that if X_0, the LSB (Least significant Bit) is 0, [X] is ***even***!

Converting from decimal to binary המרה מעשרוני לבינארי

Brut force:

- Find the largest power of 2 that is smaller than the number

$[X]=[???]_2=181 > 128=2^7$ but $2^8=256> 181$

so $181= [X] = [\overset{128}{1}, \underbrace{X_6,X_5,X_4,X_3,X_2,X_1,X_0}]$

=> This is 53 = 181-128

$64=2^6>53$ but $53>2^5=32$

thus $[X] = [1,0,1,X_4,X_3,X_2,X_1,X_0]$

$181= [X] = [\overset{128}{1}, 0,\overset{32}{1},\underbrace{X_4,X_3,X_2,X_1,X_0}]$

=> This is 21 = 53-32

$21>2^4=16$

thus $[X] = [1,0,1,1,X_3,X_2,X_1,X_0]$

$181= [X] = [\overset{128}{1}, 0,\overset{32}{1},\overset{16}{1},\underbrace{X_3,X_2,X_1,X_0}]$

=> This is 5 = 21-16

$8=2^3>5$ but $5>2^2=4$

thus $[X] = [1,0,1,1,0,1,X_1,X_0]$

$181= [X] = [\overset{128}{1}, 0,\overset{32}{1},\overset{16}{1},0,\overset{4}{1},\underbrace{X_1,X_0}]$

=> This is 1 = 5-4

$2=2^1>1$ but $1=2^0$

so we get $181= [X] = [\overset{128}{1}, 0,\overset{32}{1},\overset{16}{1},0,\overset{4}{1},0,\overset{1}{1}]$

Converting from decimal to binary המרה מעשרוני לבינארי

$$[X] = [X_{n-1}, X_{n-2}, \dots \quad X_2, X_1, \,\big|\, X_0]_K$$

$$\begin{tikzcd}[column sep=small] {[X]_K = [} & X_{n-1}, \arrow[dr] & X_{n-2}, \dots \arrow[dr] & X_2, \arrow[dr] & X_1, \arrow[dr] & X_0]_K \arrow[dr] & \\ {[X]_K / K =} & & {[X_{n-1},} & X_{n-2}, \dots & X_2, & X_1]_K & X_0 \end{tikzcd}$$

Divide by base:

- When we divide by base, the right-hand side digit becomes a fraction (or 0) since it is in the range 0 to K-1. So integer division will give us the right-hand side digit as the remainder. The result is the original number shifted right by 1 position. $[X]_K>>1= \lfloor [X]_K / K \rfloor \ (X_0)$
- So in our case we divide $[X]_2$ by 2. If $[X]_2$ is even the remainder is 0 and this will be our LSB. If $[X]_K$ is odd, the remainder is 1 and so is the LSB.
- We then continue the process with the division result – getting the next bit of the binary representation.
- $181/2=90(1) \Rightarrow X_0=1;$ $90/2=45(0) \Rightarrow X_1=0;$ $45/2=22(1) \Rightarrow X_2=1;$
  $22/2=11(0) \Rightarrow X_3=0;$ $11/2=5(1) \Rightarrow X_4=1;$ $5/2=2(1) \Rightarrow X_5=1;$
  $2/2=1(0) \Rightarrow X_6=0;$ $1/2=0(1) \Rightarrow X_7=1;$ $0/2=0(0) \Rightarrow X_i=0$ for $i>7;$

So: 181 = [10110101]

Converting from binary to decimal המרה מבינארי לעשרוני

Brut force – Using the value equation:

- X = $\sum_{i=0}^{n-1} X_i \cdot 2^i$

- So for [10110101] we get

  X= $1\cdot2^7+0\cdot2^6+1\cdot2^5+1\cdot2^4+0\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0=128+32+16+4+1=181$

Converting from binary to decimal המרה מבינארי לעשרוני

"recursive" calculation – good for SW loop:

- $X=(\ldots((X_{n-1}\cdot K+X_{n-2})\cdot K+X_{n-3})\cdot K+\ldots+X_1)\cdot K+X_0$
- So for [10110101] we have

$$X=((((((1\cdot 2+0)\cdot 2+1)\cdot 2+1)\cdot 2+0)\cdot 2+1)\cdot 2+0)\cdot 2+1=181$$

Easy to write c-code:

```
for(i=0;i<8;i++)
{
  x=x*2+new_bit;
}
```

$$\begin{tikzcd}[1] \arrow[d] \\ [10] \arrow[d] \\ [101] \arrow[d] \\ [1011] \arrow[d, dashed, no head] \\ [10110101]\end{tikzcd}$$

The MSB is multiplied by 2^7, the next bit by 2^6, etc., till the LSB

Another explanation is that we start calculating the value from the MSB and down: the MSB is 1. In order to add the next digit, let's shift it to the left, i.e., multiply it by 2, and then add the next digit. Now we have [10]=2·[1]+0. In order to add the next digit, let us shift the 2 MSBs to the left, i.e., multiply the [10] by 2 and then add the next digit. So we get 2· [10]+1=[101]= 2·2+1=5, and so on

Summary:

Converting from any base to decimal

Brut force – Using the value equation will always work:

$$X = \sum_{i=0}^{n-1} X_i \cdot K^i$$

המרה לעשרוני: לפי הנוסחה

Converting from decimal to any base

- Divide (integer division) the number by the base
- You get Result (quotient) & a remainder
- The remainder is the next digit on the left

  (1st remainder is the rightmost digit)
- Continue this (divide the result by K and get the next digit – and so on) until the Result is 0

המרה מעשרוני:

- חלק (חלוקה בשלמים) בבסיס
- השארית היא הספרה הבאה של ההמרה (התחל מימין)
- המשך לחלק את התוצאה בבסיס לקבלת הספרה הבאה.

Summary:

Converting from any base to decimal

Brut force – Using the value equation will always work:

$$X = \sum_{i=0}^{n-1} X_i \cdot K^i$$

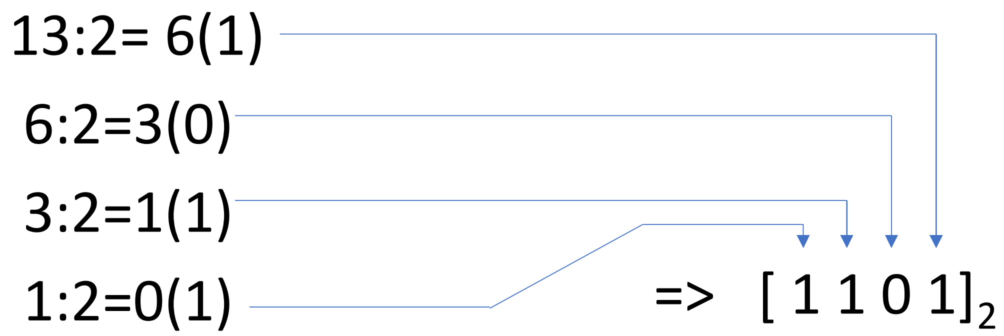
המרה לעשרוני: לפי הנוסחה

$$[1101]_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1 = 8+4+0+1 = 13$$

Converting from decimal to any base

המרה מעשרוני:

- חלק (חלוקה בשלמים) בבסיס
- השארית היא הספרה הבאה של ההמרה (התחל מימין)
- המשך לחלק את התוצאה בבסיס לקבלת הספרה הבאה.



13:2= 6(1)

6:2=3(0)

3:2=1(1)

1:2=0(1)

$\Rightarrow [1\,1\,0\,1]_2$

Hexadecimal numbers

- This works since:
- We will split i to n/4 groups of 4 bits. The serial number of the group is d and inside the group the serial number of the bits is m.
- m= 3210 3210 3210 3210 3210 3210 3210 3210
- d = 7 6 5 4 3 2 1 0
- [0111 0101 1100 0010 0101 1111 1101 0011]
- i 31 30 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
- $i=4\cdot d+m \quad i=0:n-1,\ d=0:n/4-1,\ m=0:3$
- We will mark the binary number formed by the d-th group as H_d. $H_d=\sum_{m=0}^{3} X_{(4d+m)}\cdot 2^m$

Thus $X=\sum_{i=0}^{n-1} X_i\cdot 2^i=\sum_{d=0}^{\frac{n}{4}-1}\sum_{m=0}^{3} X_{(4d+m)}\cdot 2^{(4d+m)}=\sum_{d=0}^{\frac{n}{4}-1} H_d\cdot 2^{4d}$

Which is indeed a Hex number equation ($2^{4d}=16^d$).

Base K integers

- $[X]_K = [X_{n-1}, X_{n-2}, \dots X_2, X_1, X_0]_K$ ($X_i \in \{0, K-1\}$,i.e., K digits)
- The value of $[X]_K$ is given by:

$$X = X_{n-1} \cdot K^{n-1} + X_{n-2} \cdot K^{n-2} + \dots + X_2 \cdot K^2 + X_1 \cdot K^1 + X_0 \cdot K^0$$

or

$$X = \sum_{i=0}^{n-1} X_i \cdot K^i$$

Note: Shift left = multiply by K. Shift right = divide by K.

Range of base K integers

$A_1\,;\quad A_1 \cdot q^1;\ A_1 \cdot q^2;\quad \dots\quad A_1 \cdot q^{N-1}$

- The range that can be represented by n digits is 0-(K^n-1)

- We have K^n possible digit combinations

  From $[0,0,\dots,0]_K$ till $[K-1,K-1,\dots,K-1]_K$

  So the values will be 0-(K^n-1), see below

q=K $\quad A_1$=1

$$X = (K-1)\cdot K^{n-1}+\dots+(K-1)\cdot K^2+(K-1)\cdot K^1+(K-1)\cdot K^0 = (K-1)\cdot(K^{n-1}+\dots+K^2+K^1+\underset{\uparrow}{K^0})$$

- Using geometric series sum equation (sum $= A_1 \cdot \frac{q^N-1}{q-1}$ where we have N elements in the series) we get:

$$X = \sum_{i=0}^{n-1}(K-1)\cdot K^i = (K-1)\cdot\left[1\cdot\frac{K^n-1}{K-1}\right] = K^n-1$$

Hexadecimal numbers

- Hex numbers mean we use K=16 as our base.
- We need 16 digits. These are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- The table on the right shows the relation between 4 bits binary numbers, decimal numbers and the Hex Digits

[X_3	X_2	X_1	X_0]	value	Hex digit
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F

Hexadecimal numbers

- Hex numbers are for humans! It is easier to handle hex numbers than long binary numbers.

- Let’s compare [11101011100001001011111111010011] to [11101011010001001011111111010011]. It is hard to say the long list of 1-s & 0-s.

- We will convert both to hex numbers:

Starting from the LSB we collect each 4 bits to a group

[111 0101 1100 0010 0101 1111 1101 0011]

then we replace each group with its hex digit (adding leading zeros if required)

$[75C25FD3]_{16}$ The other number is $[75A25FD3]_{16}$. Easier to compare

Hexadecimal numbers

- This works since:
- We will split i to n/4 groups of 4 bits. The serial number of the group is d and inside the group the serial number of the bits is m.
- m= 3210 3210 3210 3210 3210 3210 3210 3210
- d = 7 6 5 4 3 2 1 0
- [0111 0101 1100 0010 0101 1111 1101 0011]
- i 31 30 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
- $i=4\cdot d+m$ $i=0:n-1$, $d=0:n/4-1$, $m=0:3$
- We will mark the binary number formed by the d-th group as H_d. $H_d=\sum_{m=0}^{3} X_{(4d+m)} \cdot 2^m$

Thus $X = \sum_{i=0}^{n-1} X_i \cdot 2^i = \sum_{d=0}^{\frac{n}{4}-1} \sum_{m=0}^{3} X_{(4d+m)} \cdot 2^{(4d+m)} = \sum_{d=1}^{\frac{n}{4}-1} H_d \cdot 2^{4d}$

Which is indeed a Hex number equation ($2^{4d} = 16^d$).

Converting decimal to Hex - example

Let’s convert 181 to Hex

181 : 16 = 11 (5)

So 181 = $[B\,5]_{16}$

11 : 16 = 0 (11)

We already know that 181 = $[10110101]_2$ and indeed $[\underbrace{1011}_{B}\underbrace{0101}_{5}]_2 = [B5]_{16}$

Fractions

Base 10 fractions – Decimal numbers

- Here is a base 10 example:

  $[5097.125]_{10} = 5\cdot10^3+0\cdot10^2+9\cdot10^1+7\cdot10^0+1\cdot10^{-1}+2\cdot10^{-2}+5\cdot10^{-3}$

  3 2 1 0 -1 -2 -3

- Every digit still has a different weight that depends on the position
  - First position to the right of the binary point has weight of $10^{-1}=1/10=0.1$
  - 2nd position has weight of $10^{-2}=1/100=0.01$
  - 3rd position has weight of $10^{-3}=1/1000$
  - m-th position to the right of the binary point has weight of $10^{-m}=1/10^m$

- The value of $[X]_{10}$ where we have n digits of integer and m digits of fraction is give by

$$X = \sum_{i=-m}^{n-1} X_i \cdot 10^i$$

Binary fractions

- Similarly to base 10 we can use negative powers of 2 as well:

  $[1011.101]_2 = 1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0+1\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3}$

  3 2 1 0 -1 -2 -3

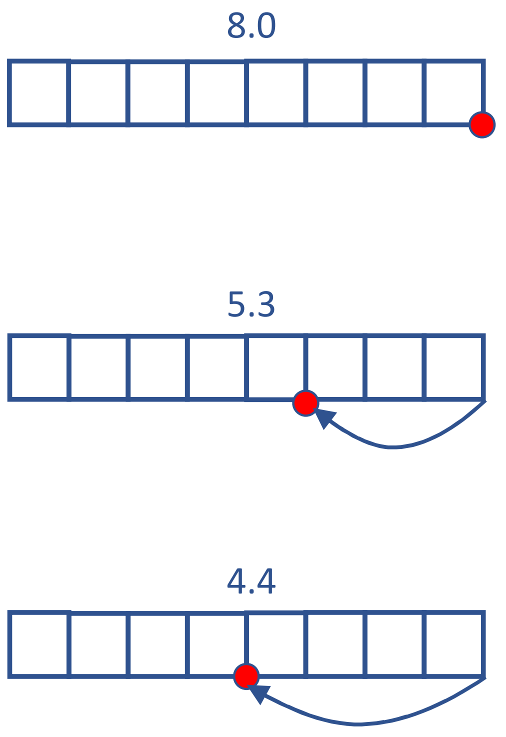
- Every bit still has a different weight that depends on the position
  - First position to the right of the decimal point has weight of $2^{-1}=1/2=[0.1]_2$
  - 2nd position has weight of $2^{-2}=1/2^2=1/4=[0.01]_2$
  - 3rd position has weight of $2^{-3}=1/2^3=1/8=[0.001]_2$
  - m-th position to the right of the decimal point has weight of $2^{-m}=1/2^m$

- The value of [X] where we have n bits of integer and m bits of fraction (we denote this [n.m] format) is given by:

$$X = \sum_{i=-m}^{n-1} X_i \cdot 2^i \qquad \text{(where } X_i \text{ is 0 or 1)}$$

Binary fractions - examples

[8.0]	[5.3] $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$	[4.4] $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$ $\frac{1}{16}$
========	=========	=========
[00000000]=0	[00000.000]=0	[0000.0000]= 0
[00000001]=1	[00000.001]= $^{1}/_{8}$	[0000.0001]= $^{1}/_{16}$
[00000010]=2	[00000.010]= $^{1}/_{4}$	[0000.0010]= $^{1}/_{8}$
[00000011]=3	[00000.011]= $^{3}/_{8}$	[0000.0011]= $^{3}/_{16}$
[00000100]=4	[00000.100]= $^{1}/_{2}$	[0000.0100]= $^{1}/_{4}$
[00000101]=5	[00000.101]= $^{5}/_{8}$	[0000.0101]= $^{5}/_{16}$
[00000110]=6	[00000.110]= $^{3}/_{4}$	[0000.0110]= $^{3}/_{8}$
[00000111]=7	[00000.111]= $^{7}/_{8}$	[0000.0111]= $^{7}/_{16}$
[00001000]=8	[00001.000]= 1	[0000.1000]= $^{1}/_{2}$
[00001001]=9	[00001.001]= $1^{1}/_{8}$	[0000.1001]= $^{9}/_{16}$
[00001010]=10	[00001.010]= $1^{1}/_{4}$	[0000.1010]= $^{5}/_{8}$
[00001011]=11	[00001.011]= $1^{3}/_{8}$	[0000.1011]= $^{11}/_{16}$
[00001100]=12	[00001.100]= $1^{1}/_{2}$	[0000.1100]= $^{3}/_{4}$
[00001101]=13	[00001.101]= $1^{5}/_{8}$	[0000.1101]= $^{13}/_{16}$
[00001110]=14	[00001.110]= $1^{3}/_{4}$	[0000.1110]= $^{7}/_{8}$
[00001111]=15	[00001.111]= $1^{7}/_{8}$	[0000.1111]= $^{15}/_{16}$
[00010000]=16	[00010.000]= 2	[0001.0000]= 1
[00010001]=17	[00010.001]= $2^{1}/_{8}$	[0001.0001]= $1^{1}/_{16}$
.	.	.
.	.	.
.	.	.
[01111111]=127	[01111.111]=$15^{7}/_{8}$	[0111.1111]= $7^{15}/_{16}$
[10000000]=128	[10000.000]=16	[1000.0000]= 8
.	.	.
.	.	.
[11101111]=239	[11101.111]=$29^{7}/_{8}$	[1110.1111]= $14^{15}/_{16}$
[11110000]=240	[11110.000]=30	[1111.0000]= 15
[11110001]=241	[11110.001]=$30^{1}/_{8}$	[1111.0001]= $15^{1}/_{16}$
.	.	.
.	.	.
[11111100]=252	[11111.100]=$31^{1}/_{2}$	[1111.1100]= $15^{3}/_{4}$
[11111101]=253	[11111.101]=$31^{5}/_{8}$	[1111.1101]= $15^{13}/_{16}$
[11111110]=254	[11111.110]=$31^{3}/_{4}$	[1111.1110]= $15^{7}/_{8}$
[11111111]=255	[11111.111]=$31^{7}/_{8}$	[1111.1111]= $15^{15}/_{16}$

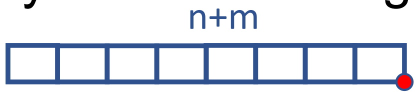


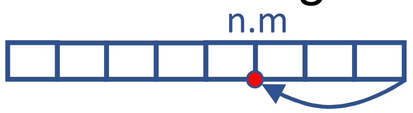
Binary fractions - examples

[8.0]	[5.3] Divide by 8	[4.4] Divide by 16
========	=========	=========
[00000000]=0	[00000.000]= 0/8=0	[0000.0000]= 0/16= 0.0
[00000001]=1	[00000.001]= 1/8=0.125	[0000.0001]= 1/16= 0.0625
[00000010]=2	[00000.010]= 2/8=0.25	[0000.0010]= 2/16= 0.125
[00000011]=3	[00000.011]= 3/8=0.375	[0000.0011]= 3/16= 0.1875
[00000100]=4	[00000.100]= 4/8=0.5	[0000.0100]= 4/16= 0.25
[00000101]=5	[00000.101]= 5/8=0.625	[0000.0101]= 5/16= 0.3125
[00000110]=6	[00000.110]= 6/8=0.75	[0000.0110]= 6/16= 0.375
[00000111]=7	[00000.111]= 7/8=0.875	[0000.0111]= 7/16= 0.4375
[00001000]=8	[00001.000]= 8/8=1.0	[0000.1000]= 8/16= 0.5
[00001001]=9	[00001.001]= 9/8=1.125	[0000.1001]= 9/16= 0.5625
[00001010]=10	[00001.010]= 10/8=1.25	[0000.1010]= 10/16= 0.625
[00001011]=11	[00001.011]= 11/8=1.375	[0000.1011]= 11/16= 0.6875
[00001100]=12	[00001.100]= 12/8=1.5	[0000.1100]= 12/16= 0.750
[00001101]=13	[00001.101]= 13/8=1.625	[0000.1101]= 13/16= 0.8125
[00001110]=14	[00001.110]= 14/8=1.75	[0000.1110]= 14/16= 0.875
[00001111]=15	[00001.111]= 15/8=1.875	[0000.1111]= 15/16= 0.9375
[00010000]=16	[00010.000]= 16/8=2.0	[0001.0000]= 16/16= 1.0
[00010001]=17	[00010.001]= 17/8=2.125	[0001.0001]= 17/16= 1.0625
.	.	…
.	.	…
.	.	…
[01111111]=127	[01111.111]=127/8=$15^{7}/_{8}$=15.875	[0111.1111]=127/16=$7^{15}/_{16}$
[10000000]=128	[10000.000]=128/8=16.0	[1000.0000]=128/16=8
.	.	.
.	.	.
[11101111]=239	[11101.111]=239/8=$30^{7}/_{8}$=30.875	[1110.1111]=255/16=$14^{15}/_{16}$
[11110000]=240	[11110.000]=240/8=30.0	[1111.0000]=255/16=15
[11110001]=241	[11110.001]=241/8=$30^{1}/_{8}$=30.125	[1111.0001]=255/16=$15^{1}/_{16}$
.	.	.
.	.	.
[11111100]=252	[11111.100]=252/8=$31^{4}/_{8}$=31.5	[1111.1100]=255/16=$15^{12}/_{16}$
[11111101]=253	[11111.101]=253/8=$31^{5}/_{8}$=31.625	[1111.1101]=255/16=$15^{13}/_{16}$
[11111110]=254	[11111.110]=254/8=$31^{6}/_{8}$=31.75	[1111.1110]=255/16=$15^{14}/_{16}$
[11111111]=255	[11111.111]=255/8=$31^{7}/_{8}$=31.875	[1111.1111]=255/16=$15^{15}/_{16}$

Range of binary numbers with fraction

- We see that when we move the binary point 1 position to the left, we actually divide the number by 2. This is so since all bits have weights of powers of 2 which are smaller by 1 than before.
- Thus and shifting the binary point from [8.0] to [5.3] means dividing by 8 (=2^3) and shifting the binary point from [8.0] to [4.4] means dividing by 16 (=2^4)
- A binary number having n+m integer bits will have a range of 0 – ($2^{(n+m)}$-1)





- If we shift the binary point m position to the left to get a format of [n.m] we actually divide by 2^m, thus the range of the resulting numbers is

  [0 to ($2^{(n+m)}$-1)] / 2^m = [0 to ($2^{(n+m)}$ -1)/2^m] = 0 to 2^n-2^{-m}

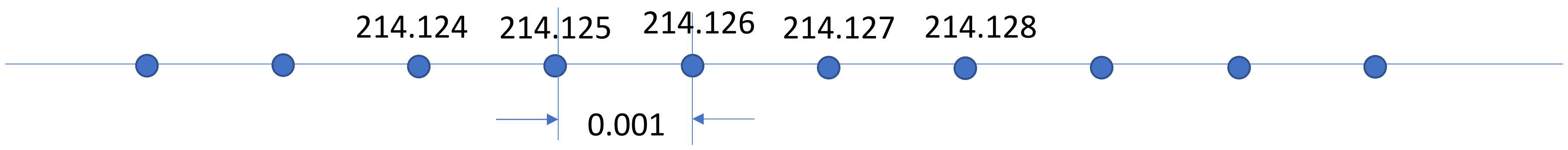
$$\frac{2^{n+m}-1}{2^m} = \frac{2^{n+m}}{2^m} - \frac{1}{2^m} = 2^n - 2^{-m}$$

- And we see that for [5.3] the max value is indeed 2^5-2^{-3}=32-1/8=$31^7/_8$=31.875
- And for [4.4] the max value is indeed 2^4-2^{-4}=16-1/16=$15^{15}/_{16}$=15.9375

What about 2/3?

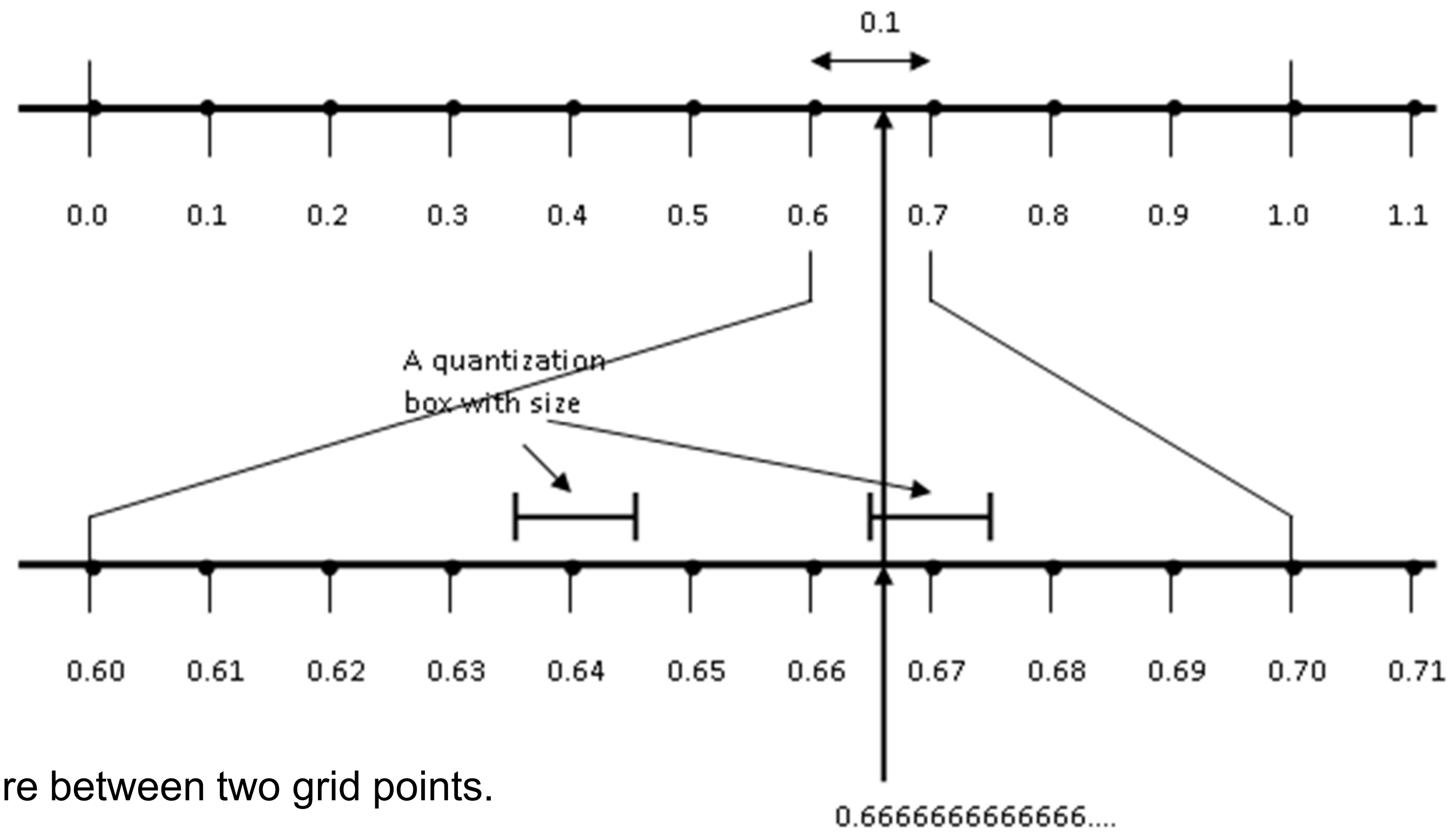
- Surprise!
- If we have a finite length system we CANNOT represent MOST of the numbers.

- With [5.3] decimal format we can only represent values that are on a grid on which numbers are 0.001 apart:



- **We have infinite values between every two consecutive grid points that CANNOT be represented by our numbers!**

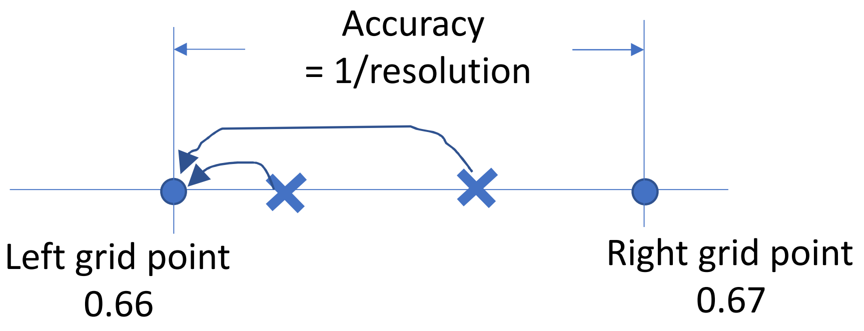
Representing 2/3 in decimal with accuracy of 0.01



2/3 is somewhere between two grid points.

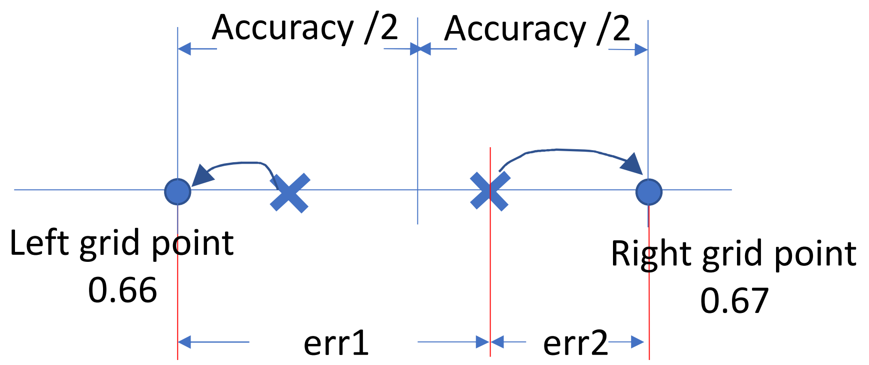
We need to choose one of these grid point to represent 2/3   This is called Quantization - כימות

קיצוץ Truncation vs. עיגול rounding



0.666666.....

T(0.666666...)= 0.66



err1= 0.666666...- 0.66 = 0.00666...

err2= 0.67-0.666666...= 0.66999...-0.6666...=0.003333

err2<err1 => we choose the right-hand side grid point of 0.67

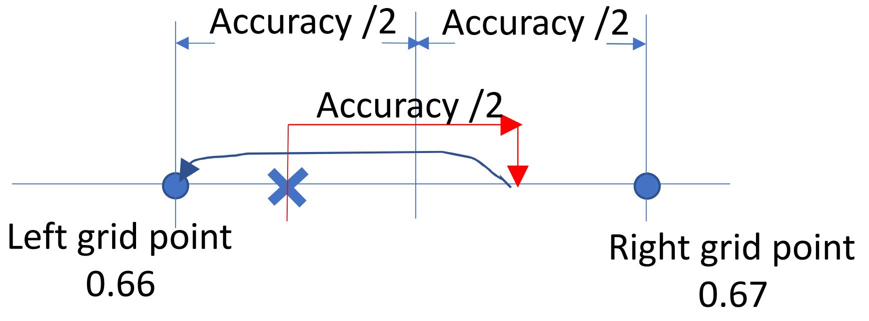
R(0.666666...)= 0.67

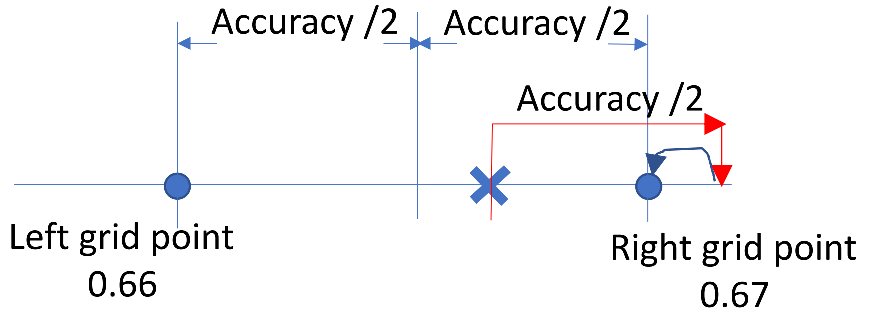
Truncation error = [0 , Accuracy)
Average Error ~ Accuracy/2

Rounding error = [-Accuracy/2, Accuracy/2)
Average Error ~ 0

A better way for rounding

Add Accuracy/2, then truncate.

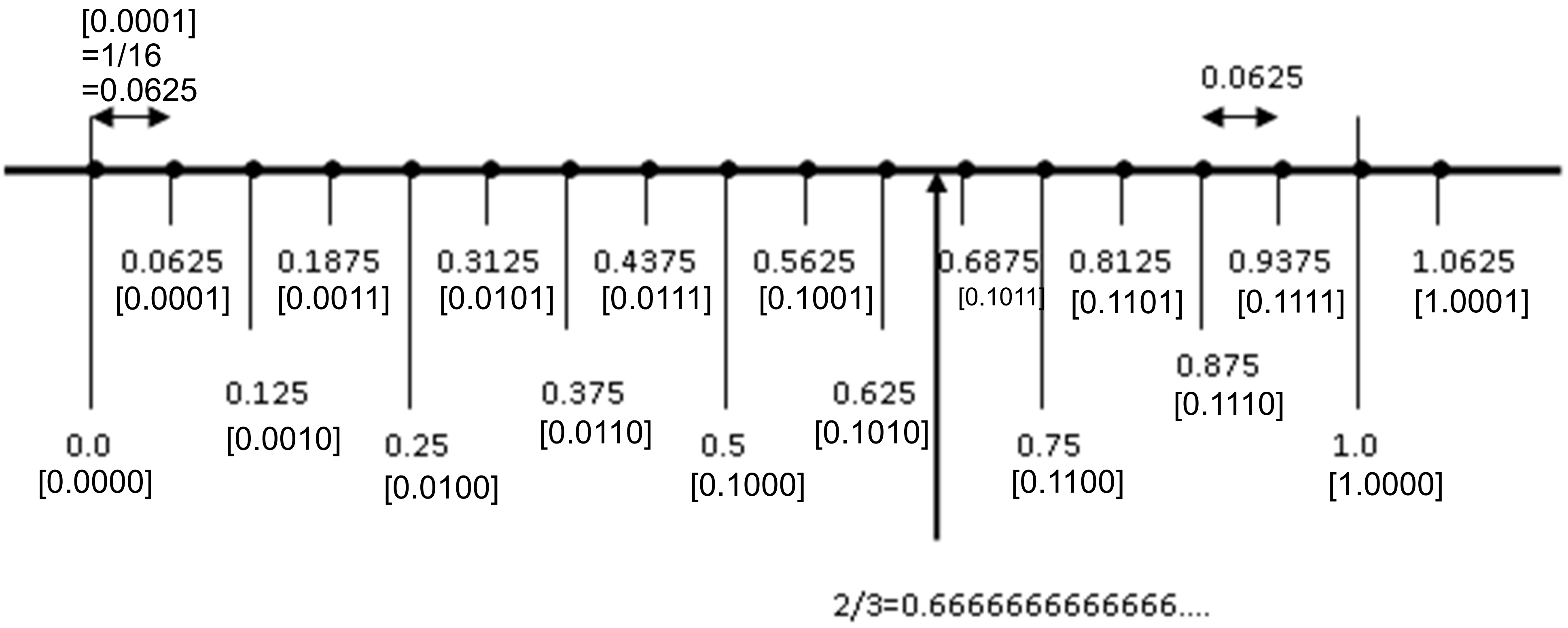




If the value we try to represent is below the half-way point between the two grid points, adding Accuracy /2 will not get us above the right-hand side grid point and truncation will bring us to the left-hand side point.
If the value we try to represent is above the half-way point, adding Accuracy /2 will get us above the right-hand side grid point and truncation will bring us to the right-hand side point.

R(0.666666…)= T(0.666666…. + 0.01/2)= T(0.66666…+0.005) = T(0.671666…)= 0.67

Representing 2/3 in binary with [1.4] format



We need to choose one of these grid point to represent 2/3. It is [0.1010] or [0.1011]

What is the accurate binary representation of 2/3?

We assume it is $[0.\ X_{-1}X_{-2}X_{-3}X_{-4}X_{-5}X_{-6}\ .....]_2$

To "expose" X_{-1} we will multiply [X] by 2:
$2*(2/3) = 4/3 = 1^1/_3 = [X_{-1}.X_{-2}X_{-3}X_{-4}X_{-5}X_{-6}\ .....]_2$
This means that $X_{-1}=1$ and that $[0.X_{-2}X_{-3}X_{-4}X_{-5}X_{-6}\ .....]_2=1/3$

To "expose" X_{-2} we will multiply the new fraction by 2:
$2*(1/3) = 2/3 = [X_{-2}.X_{-3}X_{-4}\ X_{-5}X_{-6}\ .....]_2$
This means that $X_{-2}=0$ and that $[0.X_{-3}X_{-4}X_{-5}X_{-6}\ .....]_2=2/3$
Since this also equals $[0.\ X_{-1}X_{-2}X_{-3}X_{-4}X_{-5}X_{-6}\ .....]_2$ we see that $X_{-1}=X_{-3}=X_{-5}$ … and also $X_{-2}=X_{-4}=X_{-6}$ …, thus $2/3=[0.10\ 10\ 10\ 10\ …]_2$

We can use this technique to find any fraction in any base, i.e., by multiplying by the base and "exposing" the next fraction digit.

What is the accurate binary representation of 2/3?

Let's check that result:

$[0.10\ 10\ 10\ 10\ \ldots]_2 = [0.10]_2 + [0.0010]_2 + [0.000010]_2 + \ldots =$

$= ½ + (½\ )\cdot(¼) + (½\ )\cdot(¼)^2 + (½\ )\cdot(¼)^3 + (½\ )\cdot(¼)^4 + \ldots =$

$= A_1 + \quad A_1\cdot q \quad + \quad A_1\cdot q^2 \quad + \quad A_1\cdot q^3 \quad + \quad A_1\cdot q^4 + \ldots =$

$$= \frac{A_1}{1-q} = \frac{(½)}{1-(¼)} = \frac{2}{3}$$

Another way: $X = [0.10\ 10\ 10\ 10\ \ldots]_2 = ?$

Thus, $4*X = [10.10\ 10\ 10\ 10\ \ldots]_2$ or $4*X = 2+X$ Thus $4*X - X = 2$ and $X = 2/3$

Representing 2/3 in binary with [1.4] format

So now let's add Accuracy/2, then truncate.

Accuracy is $[0.0001]_2=1/16$
Accuracy/2 is $[0.00001]_2 = 1/32$

Thus:
R(2/3)=R([0.10101010 …])=
= T([0.10101010 …]+[0.00001]) =
= T([0.1011|0010 …])=
= [0.1011] = 11/16 = 0.6875

$$\begin{array}{rl} & 0.1010|1010\ldots \\ + & 0.0000|1 \\ & \text{============} \\ & \boxed{0.1011}|\cancel{0010}\ldots \qquad = 0.6875 \end{array}$$

To get smaller error we need more fraction bits.

Summary of binary fractions

- $[X] = [X_{n-1}, X_{n-2}, \ldots X_2, X_1, X_0 . X_{-1}, X_{-2}, X_{-3} \ldots X_{-m}]$ = $[n.m]$ format

  The value of $[X]_K$ is given by: $X = \sum_{i=-m}^{n-1} X_i \cdot 2^i$ (where X_i is 0 or 1)

  The range of [X] is 0 to $(2^n - 2^{-m})$

- To find a binary representation of $X = [0 . X_{-1}, X_{-2}, X_{-3} \ldots]$

  You expose the next fraction bit (starting with X_{-1}) by multiplying the fraction by 2 and splitting the result to integer (the current bit) and fraction (to be used for next bits calc.) [BTW, this method works in all bases]

- To check correctness for periodic fractions, you can use geometric series or multiply by K^P where K=the base and P=length of period

- To round a number to [n.m] add $2^{-(m+1)}$, [0.000…001] with m 0-s, and truncate excess bits.

Summary of binary fractions - examples

- $[X] = [X_4, X_3, X_2, X_1, X_0 . X_{-1}, X_{-2}, X_{-3}]$ = [5.3] format

  The value of $[X]_K$ is given by: $X = \sum_{i=-3}^{5-1} X_i \cdot 2^i$ (where X_i is 0 or 1)

  The range of [X] is 0 to $(2^n - 2^{-m})$, i.e., 0 to $(2^5 - 2^{-3}) = 31^7/_8$

- To find a binary representation of $X = [0 . X_{-1}, X_{-2}, X_{-3} \dots] = (1/5)$

  2*(1/5)=2/5=**0**+(2/5) => (1/5) = [0.**0** …]

  2*(2/5)=4/5=**0**+(4/5) => (1/5) = [0.**00**…]

  2*(4/5)=8/5=**1**+(3/5) => (1/5) = [0.**001**…]

  2*(3/5)=6/5=**1**+(1/5) => (1/5) = [0.**0011** 0011 0011 …]

- Check correctness: $S_\infty = \frac{A_1}{1-q} = \frac{3/16}{1-1/16} = \frac{3}{15} = 1/5$ or 16*X=3+X => X=1/5

- Round [0.001100110011…] to [1.3] add $2^{-(3+1)}$, [0.0001] with 3 zeros, and truncate excess bits: [0.00110011…]+[0.0001]=[0.010|00110011…]=> [0.010]

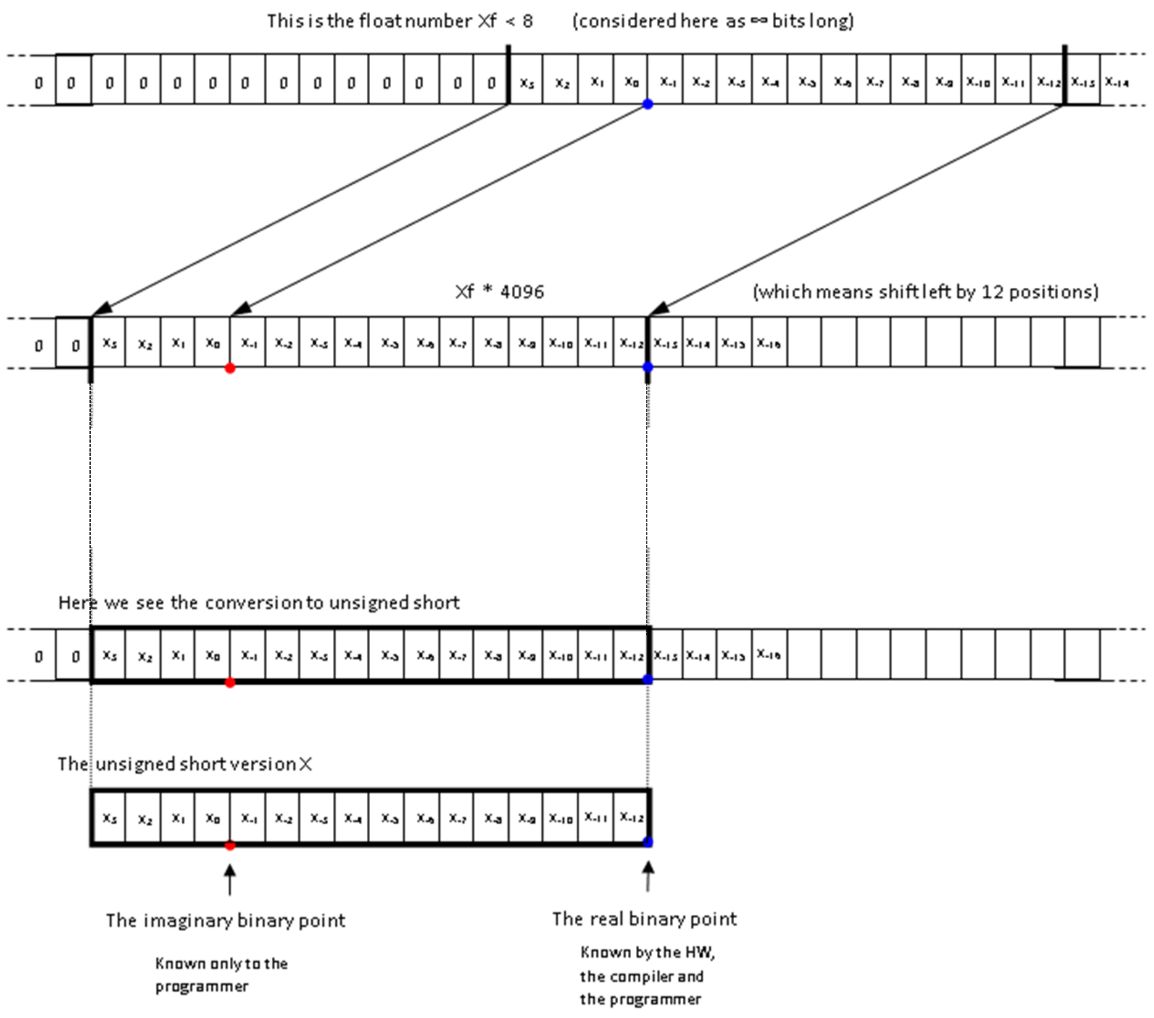
using fractions in fixed point C code [Advanced topic]

Say we want to keep a positive float number Xf in a fixed-point unsigned integer X using a 4.12 bit format. The following C code demonstrates conversion from Xf to X. We first demonstrate conversion using truncation and then conversion using rounding. This is followed by converting X back to Xf2. Xf2 is not equal to Xf since we lost accuracy in the conversion from Xf to X.

```
float Xf, Xf2;
unsigned short X,Y,Z;

/* converting using truncation */
X = (unsigned short)( Xf *(1<<12));



/* converting back */
Xf2 = (float)(X)/(1<<12);
```

using fractions in fixed point C code

[Advanced topic]



This is the float number Xf < 8 (considered here as ∞ bits long)
Xf * 4096 (which means shift left by 12 positions)
Here we see the conversion to unsigned short
The unsigned short version X
The imaginary binary point
Known only to the programmer
The real binary point
Known by the HW, the compiler and the programmer

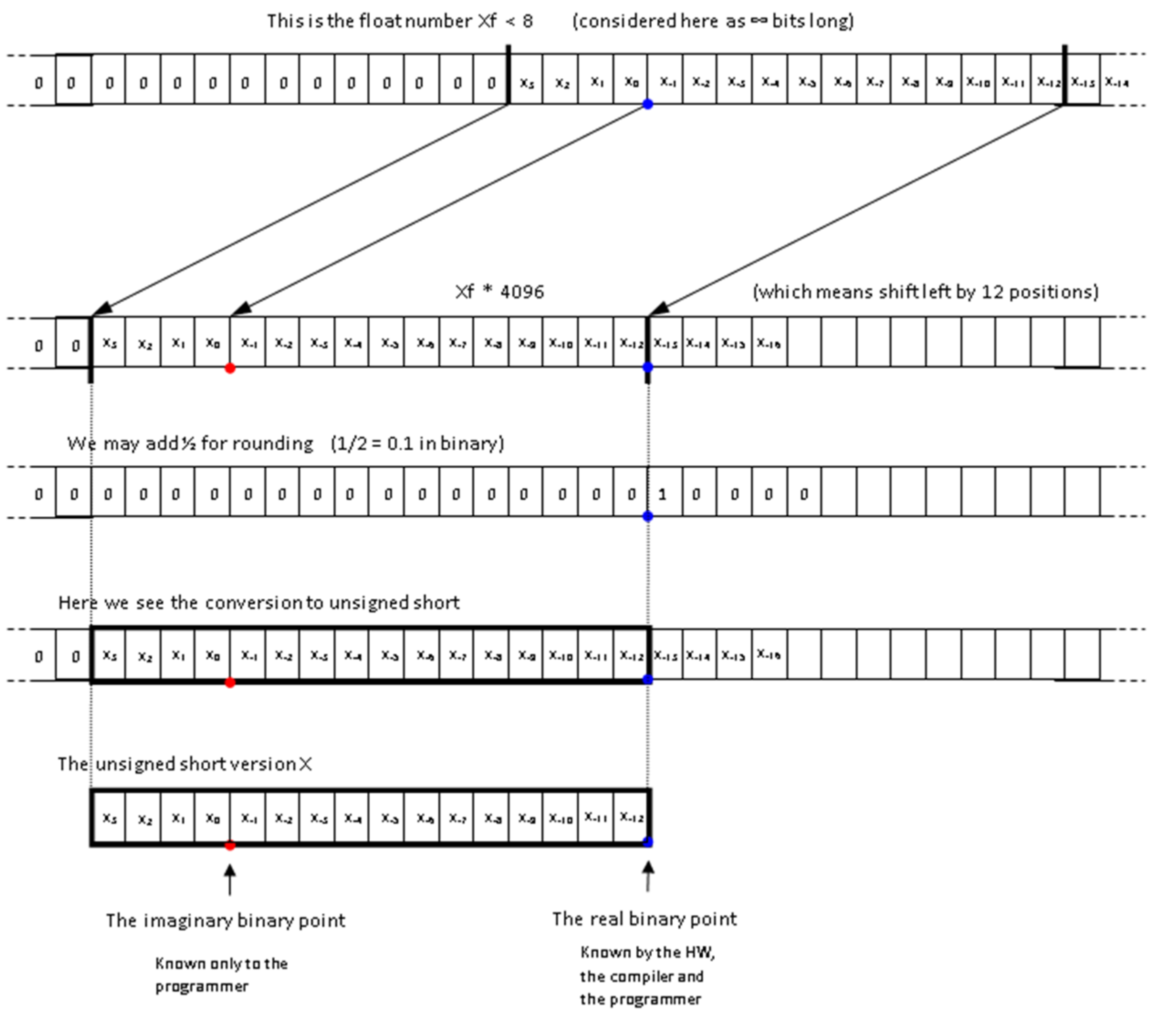
using fractions in fixed point C code [Advanced topic]

Say we want to keep a positive float number Xf in a fixed-point unsigned integer X using a 4.12 bit format. The following C code demonstrates conversion from Xf to X. We first demonstrate conversion using truncation and then conversion using rounding. This is followed by converting X back to Xf2. Xf2 is not equal to Xf since we lost accuracy in the conversion from Xf to X.

```
float Xf, Xf2;
unsigned short X,Y,Z;

/* converting using truncation */
X = (unsigned short)( Xf *(1<<12));

/* converting using rounding */
X = (unsigned short)( Xf *(1<<12) +0.5);

/* converting back */
Xf2 = (float)(X)/(1<<12);
```

using fractions in fixed point C code

[Advanced topic]

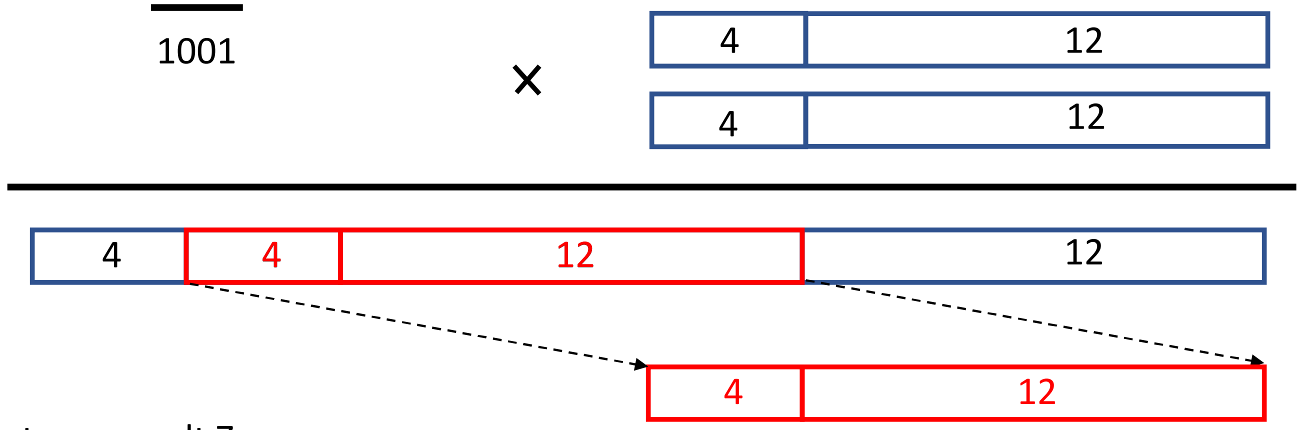

This is the float number Xf < 8 (considered here as ∞ bits long)
Xf * 4096 (which means shift left by 12 positions)
We may add ½ for rounding (1/2 = 0.1 in binary)
Here we see the conversion to unsigned short
The unsigned short version X
The imaginary binary point
Known only to the programmer
The real binary point
Known by the HW, the compiler and the programmer

using fractions in fixed point C code [Advanced topic]

Say we want to keep a positive float number Xf in a fixed-point unsigned integer X using a 4.12 bit format. The following C code demonstrates conversion from Xf to X. We first demonstrate conversion using truncation and then conversion using rounding. This is followed by converting X back to Xf2. Xf2 is not equal to Xf since we lost accuracy in the conversion from Xf to X.

```
float Xf, Xf2;
unsigned short X,Y,Z;

/* converting using truncation */
X = (unsigned short)( Xf *(1<<12));

/* converting using rounding */
X = (unsigned short)( Xf *(1<<12) +0.5);

/* converting back */
Xf2 = (float)(X)/(1<<12);
```

$$\begin{array}{r} 11 \\ \times\ 11 \\ \hline 1001 \end{array}$$



Say we want to multiply X and Y and get an integer result Z:

```
Z = (X*Y) >>12;
or    Z = (X*Y + (1<<11)) >>12;
```

$$X*2^{12}\ *\ Y*2^{12} = X*Y*2^{24}$$

Here the (1<11) stands for 0.5 and causes rounding of the result.

C# code (tested) [Advanced topic]

```
                                                                                          4         12
ushort x = (ushort)(2 * 4096 + 1);        // x=    8,193 =    0x2001 =               [0010 0000 0000 0001]   (Xf = 2.000244141)
ushort y = (ushort)(3 * 4096 + 1);        // y=   12,289 =    0x3001 =               [0011 0000 0000 0001]   (Yf = 3.000244141)
                                                                                          4         12
int z0 = 0;
ulong z1 = 0;
ushort z2 = 0;
ushort z3 = 0;
                                                                                4         12
z0 = x * y;                               // z0= 100,683,777 = 0x06005001 = [0000 0110 0000 0000 0101 0000 0000 0011]   (Zf = 6.001220763)
z1 = (ulong)(x * y);                      // z1= z0= 100,683,777
                                                                                          4         12
z2 = (ushort)((x * y) >> 12);             // z2= (ushort)(24,581.0002) = 24,581 = 0x6005 =   [0110 0000 0000 0101]   (Z2f = 6.001220703)
z3 = (ushort)((x * y + (1 << 11)) >> 12); // z3= (ushort)(24,581.5002) = 24,581 = 0x6005 =   [0110 0000 0000 0101]   (Z3f = 6.001220703)
                                                                                          4         12

MessageBox.Show("rslt0=" + z0 + "   rslt1=" + z1 + "     rslt2=" + z2 + "     rslt3=" + z3);
// prints: rslt0=100683777   rslt1=100683777   rslt2=24581   rslt3=24581
```

```
                                                                                          4         12
x = (ushort)(1.75 * 4096 + 1);            // x=    7,169 =    0x1C01 =               [0001 1100 0000 0001]   (Xf = 1.750244141)
y = (ushort)(2.50 * 4096 + 3);            // y=   10,243 =    0x2803 =               [0010 1000 0000 0011]   (Yf = 2.500732422)
                                                                                4         12  4         12
z0 = x * y;                               // z0= 73,432,067 = 0x04607C03 = [0000 0100 0110 0000 0111 1100 0000 0011]   (Zf = 4.376892269)
z1 = (ulong)(x * y);                      // z1= z0= 73,432,067
                                                                                          4         12
z2 = (ushort)((x * y) >> 12);             // z2= (ushort)(17,927.75) = 17,927 = 0x4607 =   [0100 0110 0000 0111]   (Z2f = 4.376708984)
z3 = (ushort)((x * y + (1 << 11)) >> 12); // z3= (ushort)(17,928.25) = 17,928 = 0x4608 =   [0100 0110 0000 1000]   (Z3f = 4.376953125)
                                                                                          4         12

MessageBox.Show("rslt0=" + z0 + "   rslt1=" + z1 + "     rslt2=" + z2 + "     rslt3=" + z3);
// prints: rslt0=73432067   rslt1=73432067   rslt2=17927   rslt3=17928
```

End of

Lecture #1

Unsigned Fixed Point Numbers