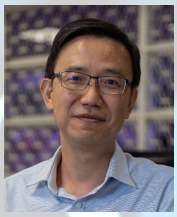
Cx1106
Computer Organization and Architecture

Computer Arithmetic



Oh Hong Lye
Senior Lecturer
School of Computer Science and Engineering, Nanyang Technological University.
Email: hloh@ntu.edu.sg

- This chapter is on computer arithmetic.
- You have been introduced many computer arithmetic concepts in Cx1105 and also during the first half of this course so this chapter serves as a supplement of what was taught.

Positional Numbering System

- Position of each numeric digit is associated with a weight.
- Each numeric value is represented through increasing powers of a **radix** (or base)
- Examples for decimal (base 10), hexadecimal (base 16) and binary (base 2) positional number notation

Base 10: $765.43_{10} = (7 \times 10^2) + (6 \times 10^1) + (5 \times 10^0) + (4 \times 10^{-1}) + (3 \times 10^{-2})$

Base 16: $1234_{16} = (1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = 4660_{10}$

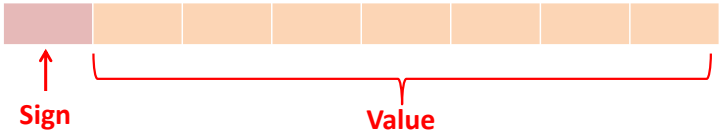
Base 2: $10101_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 21_{10}$

- **Binary numbers** are the basis for all data representation in digital computer systems.

- The decimal, binary, hexadecimal number system that you have been using are members of a number representation system known as the positonal numbering system.
- In the positional numbering system, each digit is associated with a weight.
- Let's take a look at the example below.
- For a base 10 system, the radix is 10 so each position is associated with some powers of 10.
    - Numbers to the left of the radix point, in this case, the decimal point, has an integer weightage, while those to the right of the decimal point has a fractional weightage.
- Similarly for base 16, which is the hexadecimal system you have been using.
- And lastly the base 2, which is the binary system.
- In particular, binary is the basis of computer memory. Primarily because it can be easily represented using transistor ON/OFF state.

Positive and Negative Numbers

- To represent signed integers, computer systems allocate the Most Significant Bit (MSB) to indicate the sign of a number
  - MSB = 0 indicates a positive number
  - MSB = 1 indicates a negative number
- Remaining bits contain the value of the number, which can be interpreted in different ways



- Signed binary integers can be expressed using different number format, for this course, we will touch on the most commonly used format in computing
  - Two's Complement Representation

- The rest of the discussion in this chapter will focus mainly on the binary system.
- First, how do we represent signed integers in a binary system?
- In computer system, the MSB of a binary number is typically used to indicate the sign of a number.
- 1=>negative, 0=> positive.
- There are a few ways in which binary numbers can represented, we will focus on the most commonly used representation system which is the 2's complement number representation system.

Two's Complement To Decimal Conversion

Example: What is the decimal representation for 10001110_2

Table of weights

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-128	64	32	16	8	4	2	1
1	0	0	0	1	1	1	0
-128	0	0	0	8	4	2	0

-128 + 8 + 4 + 2 = -114

The decimal representation for 10001110_2 = -114

Oh Hong Lye / Cx1106 4

- In a 2's complement number system, the most significant bit is not just a sign bit, it also has a negative weightage as shown in the slide.
- The value of the number is similarly derived by calculating the Sum of Product of the individual bits and their corresponding weightage.
- Note again that the weightage of the MSB is negative, which is why if the MSB is a '1', the final value will always be negative.

Two's Complement Representation

- Positive numbers are represented as normal binary
- Negative numbers are created by negation
  - Invert the positive value of the number (flip the bits)
  - Add one to the inverted result (ignoring any overflow)

Example:

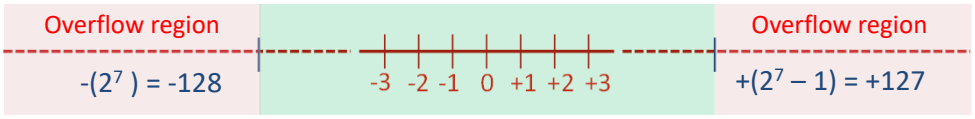
+106	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**
+106	0	1	1	0	1	0	1	0
Invert	1	0	0	1	0	1	0	1
Add 1	0	0	0	0	0	0	0	1
-106	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**

Negate

- Most Significant Bit (MSB) indicates the sign of a number (same as Signed Magnitude)
  - MSB = 0 indicates a positive number
  - MSB = 1 indicates a negative number

- As mentioned earlier, the MSB in a 2's complement number is the signed bit, MSB='1' implies the number is negative and '0' implies the number is positive.
- You can use the negation process to compute the negative equivalent of a positive number and vice versa.
- The negation process involves the following steps
  - First invert all the bits (also known as 1's complement).
  - Then add 1 to the LSB of the result and ignore any bits to the left of the MSB.
- The same process is applied when you want the convert the negative number back to positive.

Carry vs Overflow Example



Example: 48 – 19 = 48 + (-19) = 29
First, negate 19 (00010011 -> 11101101)
Then add the two numbers and discard any carries emitting from the high order bit.

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

- Carry does not always mean we have an error
- So how do we detect overflow?

Oh Hong Lye / Cx1106 6

- We will spend the next few slides talking about the difference between the Carry and Overflow Flag in a typical processor.
- Both Carry and Overflow conditional flags were discussed in the first half.
- We understand that Carry Flag is set when there is a ‘1’ that gets carried out of the MSB of the result, as shown in the example in the slide.
- But notice that in this example, 48-19=29 and the results of computation is correct even though the carry bit is set, i.e. a ‘1’ gets carried out of the MSB.
- This means that Carry bit set does not imply that there is an error in computation for this case.
    - This case here really apply to the scenario where signed numbers are involved.
    - So Carry bit set in a signed number system doesn’t mean there is an error, it also doesn’t mean there is an overflow condition in the result.
- Which brings up the next question: how do we detect overflow condition in result? An overflow condition would imply that there is an error in the result.

Detecting Overflow in Two's Complement Numbers

- Overflow can be easily detected by checking the Most Significant Bit (MSB) of the operands and result
- Conditions for overflow
  - In addition (Result = A + B)
    - If MSB(A) = MSB(B), and MSB(Result) ≠ MSB(A)
  - In subtraction (Result = A – B)
    - If MSB(A) ≠ MSB(B) and MSB(Result) ≠ MSB(A)

Operation	Conditions		Result
A + B	A > 0	B > 0	< 0
A + B	A < 0	B < 0	> 0
A – B	A > 0	B < 0	< 0
A – B	A < 0	B > 0	> 0

- To detect overflow condition in a signed number system, we need to check the overflow flag.
- Condition required to set the overflow flag is more complex than that of the Carry flag.
  - To detect if the overflow flag needs to be set, the processor needs to evaluate the sign bit of the result and the operands.
- The table illustrates the testing needed to detect overflow when performing addition and subtraction of two numbers A and B.
- Basically, these conditions correspond to the scenario where the results are not feasible. E.g. adding two positive numbers but getting a negative result, adding two negative numbers but getting a positive result.
- Under such condition, the overflow flag is set which implies that there is an error with the result if the system used is a signed number system.

Carry vs. Overflow

- Unsigned Numbers
  - Carry = 1 always indicates an overflow (new value is too large to be stored in the given number of bits)
  - The overflow flag means nothing in the context of unsigned numbers
- Signed numbers
  - Overflow = 1 indicates an overflow
  - Carry flag can be set for signed numbers, but this does not necessarily means an overflow has occurred.

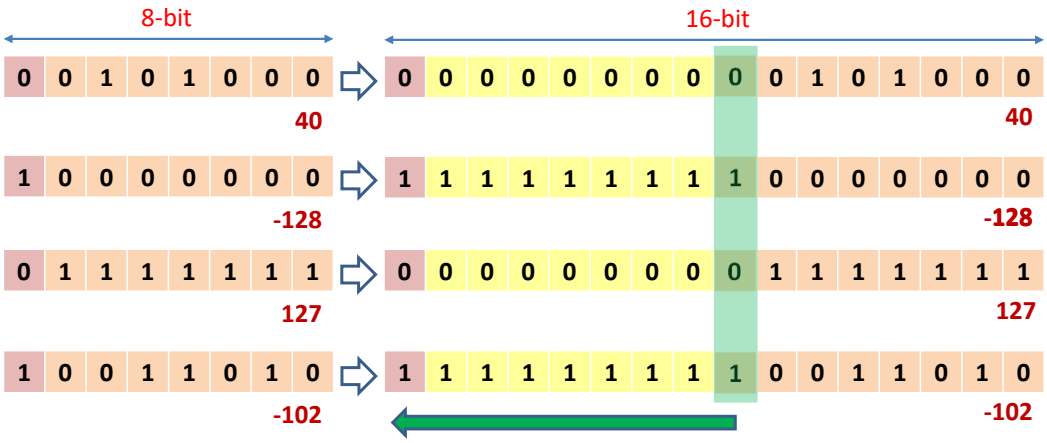
Expression	Result	Carry?	Overflow?	Corrected Result?
0100 (+4) + 0010 (+2)	0110 (+6)	No	No	Yes
0100 (+4) + 0110 (+6)	1010 (-6)	No	Yes	No
1100 (-4) + 1110 (-2)	1010 (-6)	Yes	No	Yes
1100 (-4) + 1010 (-6)	0110 (+6)	Yes	Yes	No

Oh Hong Lye / Cx1106 8

- A recap on the meaning of carry and overflow.
- In an unsigned number system,
  - Carry Flag set indicates an error as it means that the result is too large to be stored in the given number of bits.
  - While overflow flag has no meaning here.
- For a signed number system,
  - Overflow flag set implies error.
  - Carry flag may be set but it may not imply error has occur.
- The table below illustrate how the situation described could occur, e.g. In the third row, the Carry flag is set but the results is still correct. -4-2 = -6. Notice that for this case. Overflow Flag is not set.

Sign Extension

- In two's complement, sign extension is needed to convert a smaller size operand to a larger size operand



- Sign extension simply copies the sign bit (MSB) into the higher order bits

Oh Hong Lye / Cx1106 9

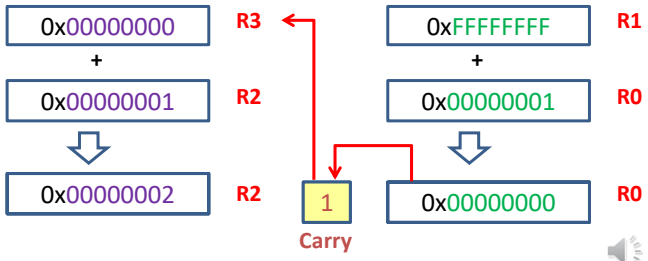
- You have also learnt about sign extension previously
- Its basically the method used when converting a smaller size to a larger size data.
- For example to convert a number stored in a 8 bit data type to a 16 bit data type variable, sign extension is used to ensure that the value in the two variables are the same.

Multi-Precision Arithmetic

- How can we add operands that are larger than 32-bits (e.g. 64 bit operands) if we have only a single 32-bit ALU?
- The solution is to reuse the 32-bit adder for multi-precision addition
- Multi-precision arithmetic involves the computation of numbers whose precision is larger than what is supported by the maximum size of the processor register (Single-Precision)

```
  00000000 FFFFFFFF
+ 00000001 00000001
  00000002 00000000
```

```
ADD   R0,R0,R1 ; add lower word with carry out
ADC   R2,R2,R3 ; add upper word with carry in
```



Oh Hong Lye / Cx1106 10

- What we have been doing are know as single-precision arithmetic, where operands are kept to the processor data width. For the 32 bit ARM processor, that would mean 32 bit operands.
- So what happen if the operands are larger than 32 bits?
- For example, what do we need to do to add two 64 bits operand?
- Solution is to perform 32bit addition multiple times, this is illustrated in the slide.
  - The ARM registers are 32bit wide so at any instance, it can only store 32bit data.
  - To add two 64bit operands, we need to add the lower order word first, followed by the higher order word.
  - Note however that when we add the higher order word, the C bit has to be added as well. Carry bit is only set if there are any '1' overflowing from the MSB of the lower order word. This is the same as the carry operation we always do with our paper caculation.
  - ARM assembly instructions wise, the instruction that add the operands and the carry bit is ADC.

CE1006/CZ1006
Computer Organisation and Architecture

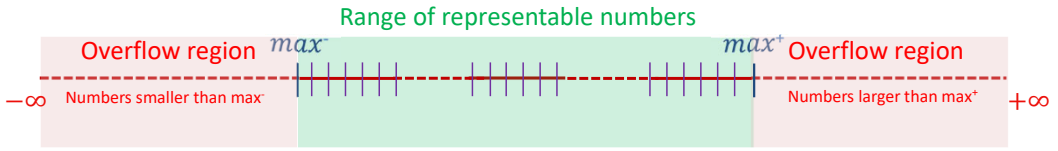
Fixed and Floating Point Number System

Oh Hong Lye
Lecturer
School of Computer Science and Engineering, Nanyang Technological University.
Email: hloh@ntu.edu.sg

- Next topic in this chapter is the fixed and floating point number system, which is the two main type of number system we typically deal with.

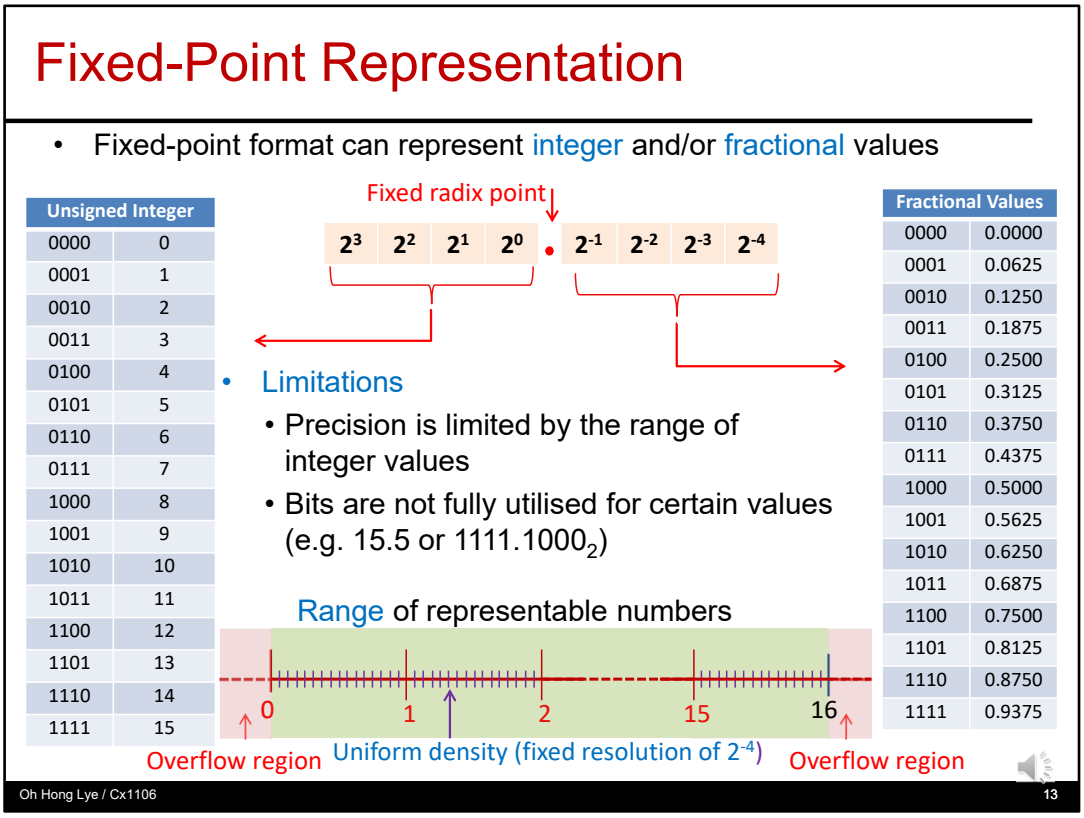
Range and Precision

- Range: Interval between smallest (max^-) and largest (max^+) representable number
  - Example: Range of two's complement is $-(2^{(N-1)})$ to $(2^{(N-1)}-1)$
  - Each tick mark is a representable number in the range
- Precision: Amount of information used to represent each number
  - Example: 1.666 has higher precision than 1.67
  - The number of tick marks provides an indication of precision



Oh Hong Lye / Cx1106 12

- A few definition before we go into details of this topic.
- Range is the interval between smallest to largest representable number in a system.
- Precision, on the other hand, is the amount of information used to represent each number. E.g. 1.666 represent information in a higher precision compared to 1.67.
- Precision correspond to the interval between adjacent tick marks in the number line shown. Each tick mark correspond to a representable number in the number system used.
- In a binary number system, it correspond to increasing the LSB by 1.

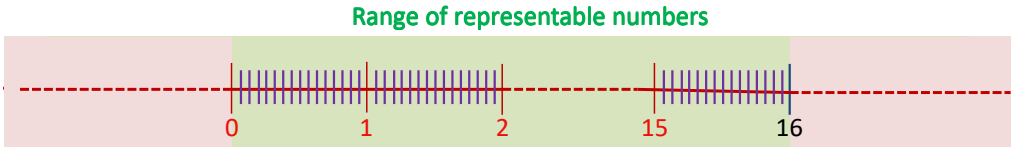


- Let's start with the numbering system which you have been using so far, the fixed point system.
- So far, we have been using fixed point number to represent integers.
    - The 4 bit integer you see here has a range of 0 to 15. Any number that is outside this range goes into the overflow region.
    - Similar to the decimal system, binary system has a radix point too and it is known as the binary point.
- And similarly, where digits after the decimal point has a fractional weightage, the bits after the binary points also has a fractional weightage.
- The smallest resolution that this system can support determine its precision, and this as mentioned earlier, correspond to its LSB.
- Fixed point system has a fixed resolution as its radix point position is fixed.
    - For this example, the resolution is 2**-4.
- Some limitation of fixed point system are
    - Given a fixed total number of bits allocated for a number, precision is limited by the range of integer.
        - If you allocate more bits for integer, the range will increase but the precision will reduce.
    - Memory used to store these data bits are not fully utilized in some cases, eg.15.5, where there are some trailing zeroes which doesn't really need to be stored.

Range and Precision Trade-off

- What happen when the radix point moves/float from one end of a number to the other end?

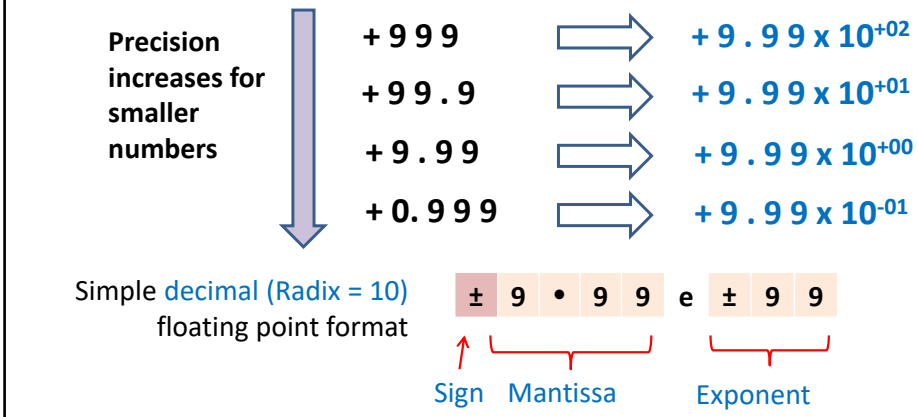
2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}



- When radix point floats from LSB to MSB
  - Range of representable numbers reduces
  - Precision increases
- The example above illustrates the concept of floating-point, but how do we represent a floating-point number?

- From the previous slide, we can deduce that when the radix point float from one end of the number to the other, the representable range and precision will change accordingly.
- Specifically, when the radix point move from LSB to MSB, the representable range reduces and the precision increases.
- When the range is large, we may not be too concern with the precision as the numbers we deal with are relatively large.
- But when the numbers we use gets smaller, we would be more concern with the precision now as these fractions are more significant now when compared with the numbers on the integer side.
- The above example illustrate the concept of a floating point number, in which we can dynamically choose between having a large range or high precision by shifting the radix point.
- But how can we realize such a system?

Floating Point Representation



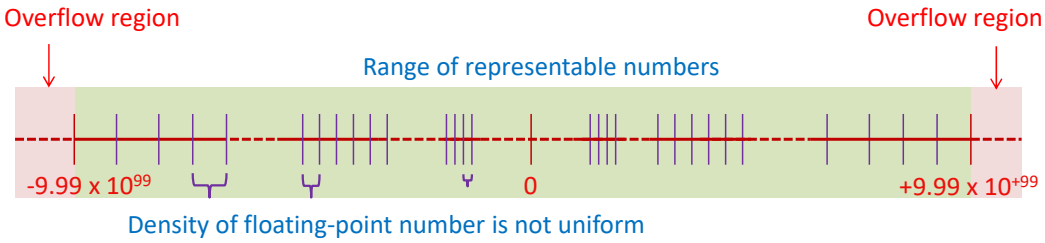
Simple decimal (Radix = 10) floating point format

- Three main fields needed for floating-point representation:
  - Sign – denote positive/negative number
  - Mantissa – base value
  - Exponent – specifies position of radix point

Oh Hong Lye / Cx1106 15

- The slide here shows how we can realise the floating number system that we discussed in the previous slide.
- Take these decimal numbers with different precision for example.
- We can express these numbers in a format which has three fields called sign, mantissa and Exponent.
- By representing number in this manner, we are able to 'move' the position of the radix point by changing the exponent value.
- This is known as floating Point Number system.
- Comparing with a fixed point number, we can see that it doesn't belong to a positional numbering system, actual value is calculated by evaluating the sign, mantissa and exponent value.

Floating Point Representation



- Floating point representation can represent values across a wide range ($-9.99*10^{+99}$ to $+9.99*10^{+99}$).
- Size of exponent determines range of representable numbers.
- Size of mantissa and value of exponent determines the representable precision.
- Small numbers can be represented with good precision. When representing large numbers, precision can be sacrificed to achieve greater range.

- For a floating point number, the size of the exponent determine the range
- While the size of the mantissa, together with the exponent value, determines the precision.
- With this representation format, we are able to have good precision when number value is small, and yet could achieve large range, of course with the trade-off in precision.

Normalisation

- In the simple decimal floating-point format, there are multiple representations for the same value

±	1	•	0	0	e	±	0	1	**Normalised**
±	0	•	1	0	e	±	0	2	
±	0	•	0	1	e	±	0	3	

- Normalisation is necessary to avoid synonymous representation by maintaining one non-zero digit before the radix-point
  - In decimal number, this digit can be from 1 to 9
  - In binary number, this digit should be 1
- Normalisation can maximise number of bits of precision

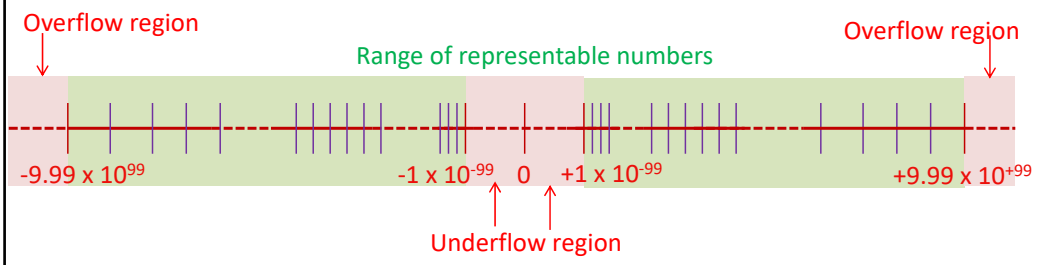
±	5	•	4	2	e	±	0	3	**Normalised**
±	0	•	5	4	e	±	0	4	

- One issue with floating point numer is that you can have multiple representations for the same number value.
- And this may potentially create confusion.
- So normalization is needed to standardise the representation.
- Normalisaton means that there should be a non-zero digit to the left of the radix point. As shown in the slide.
- Choosing this method of normalization also has an advantage of maximizing the number of bits of representable precision since the number of leading zeroes are minimised.

Underflow

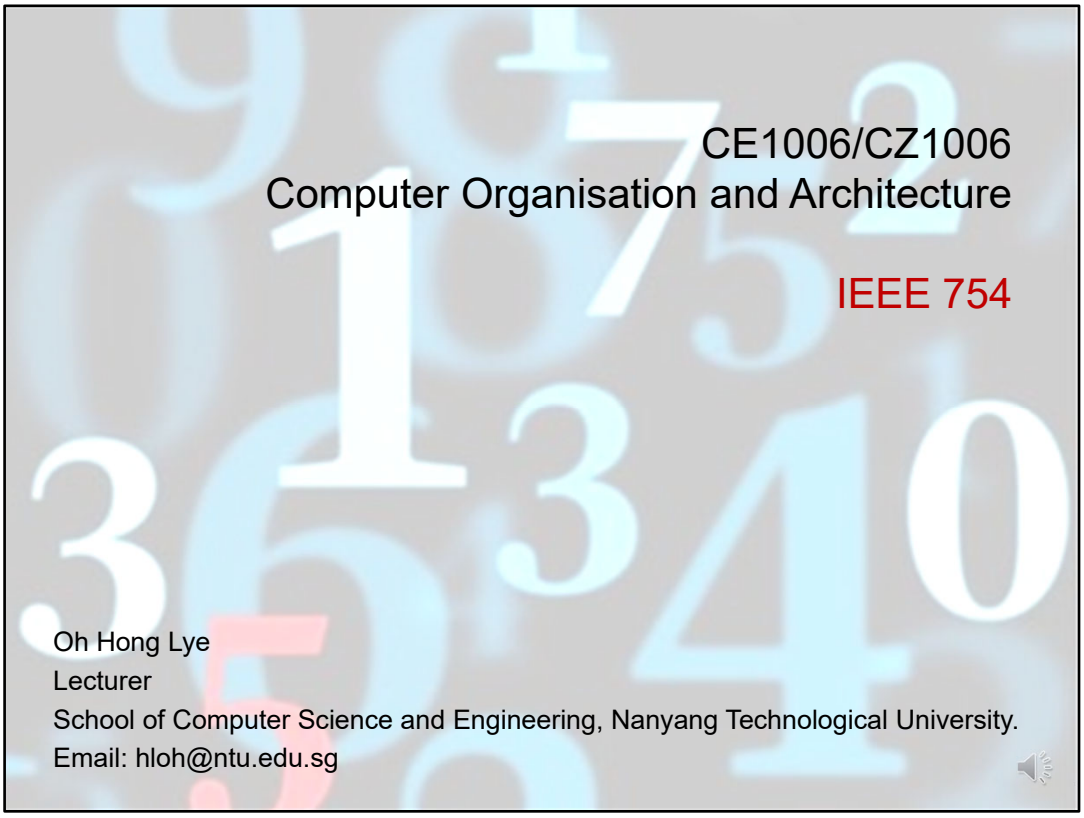
- Normalisation results in underflow regions where values close to zero cannot be represented

Smallest positive normalised number	+	1	•	0	0	e	-	9	9
Smallest negative normalised number	-	1	•	0	0	e	-	9	9



- Underflow occurs when a value is too small to be represented
- Floating-point overflow and underflow can cause programs to crash if not handled properly.

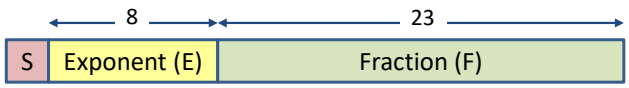
- But normalization has another side effect of creating underflow region.
- Underflow region refers to region close to zero which cannot be represented by the floating point number.
- In the example here, since the digit to the left of the decimal point has to be non zero for normalised number, that means the smallest positive normalized number is 1*10^-99, very closed to zero but not zero.
- Floating point overflow and underflow may cause the program to crash if not handled properly.
- But you'll see later that typical floating point format standard has some provision to handle the underflow scenario.



- This section is on the IEEE754 floating point number standard used in the industry.
- It is also the standard behind the floating point data type you declared in programming languages such as C.

IEEE 754 Floating Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
- Single Precision Floating-Point Numbers (32-bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

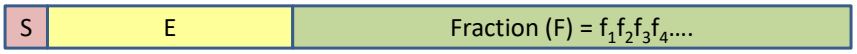


- Double Precision Floating-Point Numbers (64-bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction

Oh Hong Lye / Cx1106 20

- Because of the more complicated format, floating point system needs to be standardised.
- Else program designed by one party may not work with another.
- One of the most commonly used floating point number system is the IEEE754 standard.
- It defined both a single precision and double precision format.
- Single precision format is 32bit wide, has 1 sign bit, 8 Exponent and 23 Fractional bits. This correspond to the Float data type in C.
- Double precision format is 64bit wide, has 1 sign bit, 11 Exponent and 52 Fractional bits. It correspond to the Double data type.
- Note however that the 'Exponent' and 'Fractional' field specify in the IEEE754 format is not the Exponent and Mantissa value of a floating point number. We will touch on the details in the next slide.

IEEE 754 Normalised Numbers



$$(-1)^S \times (1.F)_2 \times 2^{E - Bias}$$

- **Sign** bit
  - S = 0 (positive); S = 1 (negative)
- **Exponent**
  - Biased representation (00000001 to 11111110)
  - Value of exponent = E - **Bias**
  - **Bias** = **127** (Single Precision) and **1023** (Double Precision)
- **Fraction**
  - Assumes hidden **1.** (not stored) for normalised numbers
  - Value of normalised floating point number is:
    $(-1)^S \times (\mathbf{1} + f_1x2^{-1} + f_2x2^{-2} + f_3x2^{-3} + f_4x2^{-4} + \ldots)_2 \times 2^{E-Bias}$

- This slide shows how the exponent and fraction field in the IEEE754 number is used to derive the actual value of the floating point number.
- Specifically
  - The Mantissa is derived by attaching a leading '1' to the left of the fractional bits. Giving the expression 1.F.
  - The actual exponent value is given by the expression E-Bias where E is the value from the Exponent Field in a IEEE754 number representation and Bias is = 127 and 1023 for Single and Double precision IEEE754 number respectively.
  - The E field in IEEE754 is a positive number, Bias allows the actual exponent to span across positive and negative number ranges.

Converting Single Precision To Decimal

- Find the decimal value of these single precision number:

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

Sign = 0 (positive)

Exponent = 10110010_2 = 178; E – Bias = 178 – 127 = 51

1 + Fraction = $(1.111)_2 = 1 + 2^{-1} + 2^{-2} + 2^{-3} = 1.875$

Value in decimal = $+1.875 \times 2^{51}$

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

Sign = 1 (negative)

Exponent = 00001100_2 = 12; E – Bias = 12 – 127 = -115

1 + Fraction = $(1.0101)_2 = 1 + 2^{-2} + 2^{-4} = 1.3125$

Value in decimal = -1.3125×2^{-115}

- Some working example for student to check their understanding in deriving the value of a IEEE754 floating point number.
- The first example has a sign bit = 0 so it's a positive number
  - Exponent is 178, so E-127 gives you 51
  - Plug into the expression 1.F*2^(E-Bias) gives the value 1.875*2^51.
- For the second example, sign bit is '1' so the number is negative. Final value is –1.3125*2^(-115).

Representable Range for Normalised Single Precision

- In normalised mode, exponent is from 00000001 to 11111110
- Smallest magnitude normalised number

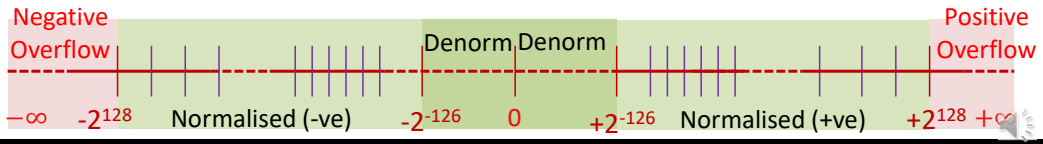
X 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Exponent = $(00000001)_2$ = 1; E - Bias = 1 – 127 = -126
1 + Fraction = $(1.000\ldots000)_2$ = 1
Value in decimal = **1×2^{-126}**

- Largest magnitude normalised number

X 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

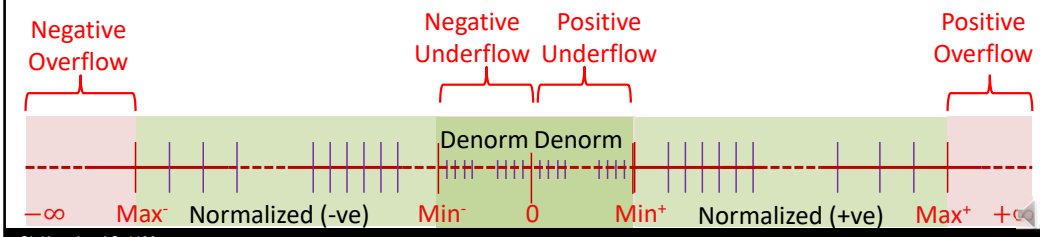
Exponent = $(11111110)_2$ = 254; E - Bias = 254 – 127 = 127
1 + Fraction = $(1.111\ldots111)_2 \approx 2$
Value in decimal = $2 \times 2^{127} \approx 2^{128}$



- The smallest normalised number for single precision IEEE754 standard correspond to E=1 and Fraction=0.
  - That gives you a value of 1*2^(-126)
- The largest normalised floating point number correspond to E=11111110 and Fraction=1111…1111 (all ones).
  - That gives a mantissa value of approximately 2 so final value is approximately 2^128.
- We can derive the range for normalised IEEE754 number using the min/max magnitude. As show in the diagram in the slide.
- Note that we didn't use the number 1111 1111 as the largest value of E, neither did we use 0 as its smallest value. Both these numbers are used for special cases in IEEE754. More on that in the next slide.

IEEE 754 Encoding

Mode	Sign	Exponent	Fraction
Normalized	1 / 0	00000001 to 11111110	Anything
Denormalized	1 / 0	00000000	Non zero
Zero	1 / 0	00000000	0000 … 0000
Infinity	1 / 0	11111111	0000 … 0000
Not a Number (NaN)	1 / 0	11111111	Non zero



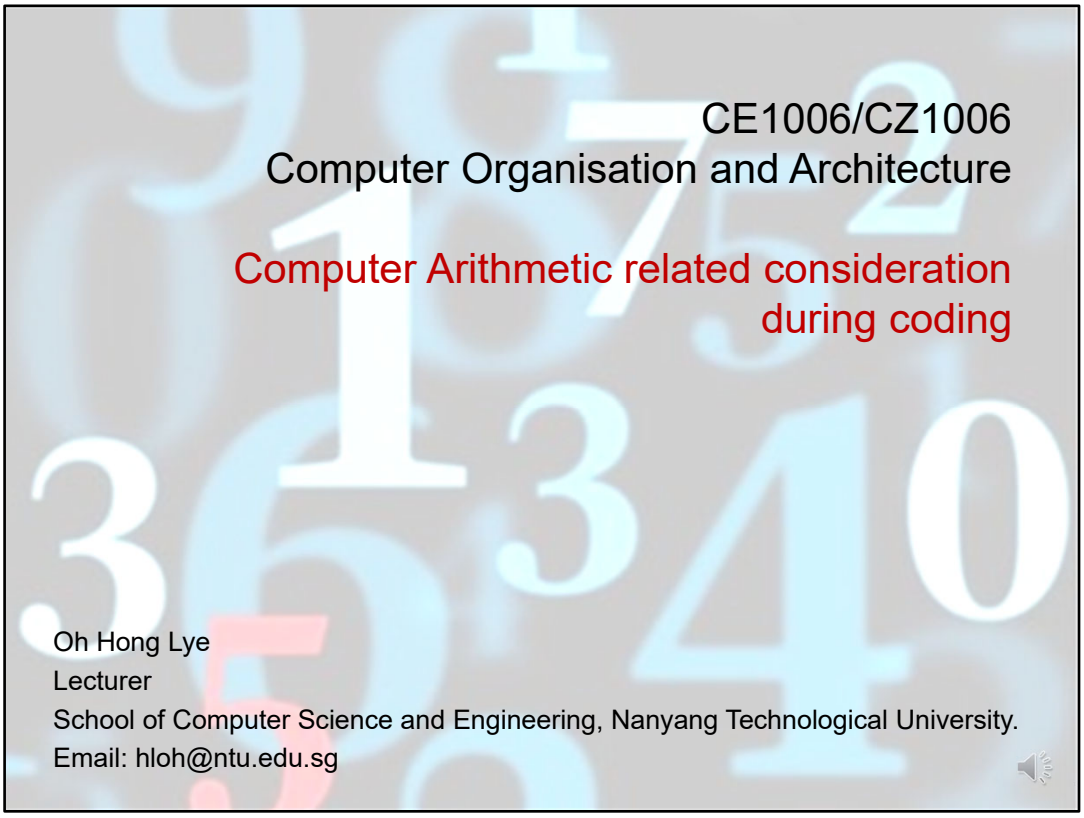
Oh Hong Lye / Cx1106 24

- As mentioned in the previous slide, some numbers are reserved for special use case in IEEE754.
- The table in this slide illustrate the various use cases.
- In normalised mode, the E value ranges from 1 to 1111 1110 while fraction can take on any value.
- Zero, Infinity and NaN are special use cases and is represented by special numbers in E and Fraction.
- IEEE754 has a special Denormalised mode that allows it to represent numbers in its underflow region.
    - In denorm mode, the MSB of the mantissa is a 0 and not a 1, i.e. 0.F instead of 1.F. Exponent is given a value zero and Fraction can be any non-zero value. Take note however that the exponent value used in the denorm mode is 2^(-126) and not 2^(-127).
    - In any case, this course will not dealt into details of denormalised mode.

Fixed Point vs Floating Point Number System

- Given the same number of bits to represent a data, e.g. 32 bits.
- Floating Point (IEEE754)
  - Max Range ≈ $2*2^{128}$ (-2^{128} to ~2^{128}).
  - Max Precision (near to zero) less than 2^{-126}
- Fixed Point
  - Max Range (Radix right of LSB, unsigned) ≈ 2^{32}
  - Max Precision (Radix left of MSB) 2^{-32}
- Floating point yield a larger range and better precision at small numbers with the same number of bits representation.
- One usually needs the best precision when the numbers are small.
- However, Fixed point number has the advantage of having uniform precision across entire range.
- Floating point number's precision changes across the range and the very coarse precision at the two end of the range may not be desirable to the intended algorithm.

- Summarising the pros and cons of fixed point vs floating point number representation
- Given the same number of data bits e.g. 32bits
  - The single precision IEEE754 floating point number, when compared to a 32bit fixed point number
    - Has a larger range
    - Is able to achieve a higher precision. This occurs when the number is near to zero.
- So floating point number yield a larger range and better precision compared to a fixed point number occupying the same amount of memory (32 bit for this case).
- However, the fixed point number has the advantage in that the precision value for a fixed point number is uniform across the entire range while that of floating point number varies across the range.
- So depending on the application requirement, you may find one is more suitable over another.



- This last section on computer arithmetic talks about some considerations to take note during coding.

Effects of four operators

- Addition/Subtraction
  - Addition/Subtraction will cause the result to increase/decrease
  - When done sufficient number of times, overflow at Min or Max end of the representable range will eventually occur
  - Take note of the range of the data type used when coding in High level language, or the width of registers and memory when coding in assembly
- Multiplication
  - Multiplication in binary is similar to an arithmetic left shift
  - Overflow at Min or Max end of the representable range
  - Similar consider as Add/Sub.
- Division
  - Division in binary is similar to an arithmetic right shift
  - Truncation of LSB leads to loss in precision
  - Reduces the magnitude of the result so it get further away from the Min/Max of the range.

- First is on the effects of the four basic operations add, subtract, multiply and divide, specifically dealing with overflow and accuracy of result.
- Addition and subtraction, if done multiple times, may potentially lead to result overflowing beyond one end of the range.
  - When that occurs, the result obtained will be wrong.
  - One example is when the result is larger than what a particular data type could hold, such as storing a value larger than 255 to the unsigned character data type,
  - Or trying to store a value larger than 255 to a 8-bit register.
- Similarly, multiplication will result in a larger number if the multiplier is larger than one, which means potentially may lead to overflow condition.
- Division with a divisor larger than one, on the other hand, gave raise to a different issue, which is the truncation of the lower order bits or digits.
  - This is easily illustrated by the fact division is similar to arithmetic right shift.
  - Any truncation of lower order bits mean a loss in accuracy of the data.
- So take note of the effects of these operators so as to have the best accuracy without running into overflow condition.

Effects of Rounding

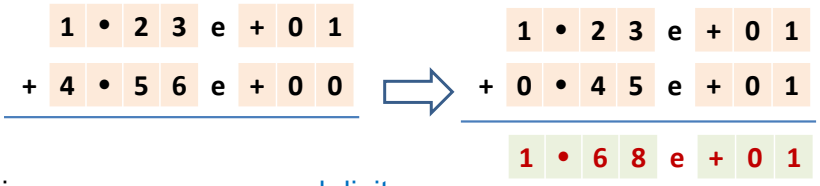
- Rounding refers to removing of LSB(s) so that the result can fit into the representable bits.
- The limits imposed in the width of the representable bits could be from the registers width, data type etc.
- Rounding can be round-up, round-down or round to nearest representable number.
- As the rounded number is an approximation of the raw result, a certain amount of rounding error is incurred.
- Below an example of rounding off a floating point number to 2 decimal points, incurring an error of $0.004*10^1$

- Common to have intermediate register with width larger than regular data registers to allow intermediate processing to be done at higher precision and thus reducing the amount of rounding error.

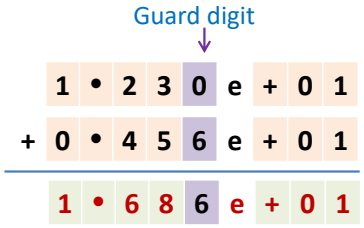
- Next is the rounding process.
- Rounding is a process in which the lower order bits of a data is truncated in order to fit the data into a register or memory location of a smaller data width.
    - Some approximation needs to be done when performing the rounding and this is by rounding up, down or to the nearest representable number.
- Since the rounded number is just an approximation of the actual data, a certain amount of rounding error is incurred in the process.
    - For example, rounding 1.686*10^01 to 1.69*10^01 result in rounding error of 0.004*10^01.
- It is common to have intermediate registers that have larger data width compared to the regular data registers in a processor. This is to preserve the intermediate data without truncating the lower order bits. Reducing the rounding error.

Rounding Error Mitigation

- When adding/subtracting two numbers, the exponents must be aligned such that they are the same



- To improve accuracy, guard digits (bits) are used to maintain precision during floating point computations. An intermediate register with a wider width could also be used.



- Rounding is then performed to ensure that the result fits into the three significant digits in the mantissa

Oh Hong Lye / Cx1106 29

- Floating point arthimetic is different from the fixed point arthimetic you have been doing previously.
- When adding/subtracting, you need to make sure the exponent has the same value in order to compute the correct result.
- Guard bits are used to maintain the precision during computation before doing rounding at the last step.
- The additional Guard digit in the example allows the least order digit 6 to be preserved.
- This in turn allows a more accurate result to be obtained after performing the rounding. For this case, the final result 1.69*10^1 is closer to the actual result of 1.686*10^1, compared to the previous result 1.68*10^1 obtained when there is no guard digit to store the least order digit ‘6’.
- Actually, the same method is applicable to fixed point numbers as well, always try to maintain as many bits of information during computation and perform any rounding or any operation that will result in bit truncation last.
    - For the simple reason that any bit truncation means losing data bits and losing data bits means losing precision.

Handing numbers with different magnitudes

- Suppose we want to compute the following :

$1.23x10^3 + 1.00x10^0 + 1.00x10^0 + 1.00x10^0 + 1.00x10^0$

	1 • 2 3 0 e + 0 3	
+	0 • 0 0 1 e + 0 3	Align exponent
	1 • 2 3 0 e + 0 3	Rounding
+	0 • 0 0 1 e + 0 3	Align exponent
	1 • 2 3 0 e + 0 3	Rounding
	.	
	.	
	1 • 2 3 0 e + 0 3	Rounding

Oh Hong Lye / Cx1106 30

- As data width is finite, so computer typically face limitation
  - when dealing with very large number as it may not have enough bits for required range.
  - Or very small numbers where there may be insufficient bits to achieve the required precision.
- On top of that, we often need to do rounding so as to fit the data into the limited number of bits available.
  - This introduces rounding error.
- In this example, we are performing addition of a few small numbers to a large number.
  - Recall that we need to align the exponent for floating point number addition.
  - If we perform a rounding after adding the first data, the '1' at the end would be truncated.
- This sequence will happen for the rest of the additions. Which means we are going to get the same 1.23*e^3 at the end of the day and this is not the answer we wanted.
- What should we do instead?

Handing numbers with different magnitudes

Example:
$1.00x10^0 + 1.00x10^0 +$
$1.00x10^0 + 1.00x10^0 +$
$1.00x10^0 + 1.23x10^3$

- The order of evaluation can affect accuracy of result
  - Add/subtract operands with similar size of magnitude first.

```
  1 • 0 0 0 e + 0 0
+ 1 • 0 0 0 e + 0 0
-----------------
  2 • 0 0 0 e + 0 0
+ 1 • 0 0 0 e + 0 0
-----------------
  3 • 0 0 0 e + 0 0
+ 1 • 0 0 0 e + 0 0
-----------------
  4 • 0 0 0 e + 0 0
+ 1 • 0 0 0 e + 0 0
-----------------
  0 • 0 0 5 e + 0 3   Align
+ 1 • 2 3 0 e + 0 3
-----------------
  1 • 2 4 0 e + 0 3   Rounding
```

Oh Hong Lye / Cx1106 31

- A different result would be obtained if we change the sequence of accumulation.
- If we allow the small number to accumulate first before doing any truncation, it will allow these small numbers to accumulate to a significant value and not be truncated off during rounding process.
- In this example, a raw intermediate value 1.235*10^3 is obtained.
- After rounding, we get 1.24*10^3, i.e. the significance of the '1's are not truncated as in the case of the previous slide.
- In general, when performing arithmetic operation between numbers with huge difference in magnitude, always add/subtract between numbers that are of similar magnitude before combining them to other larger numbers.

Maximising Accuracy during computation

- Two issues: overflow and precision.
- From previous slides, addition, subtraction and multiplication may potentially lead to result overflowing the range of the representable number used.
- Division will lead to loss in precision as bits are loss due to truncation of LSB(s).
- Some rule of thumb
  - Accumulate/subtract numbers with small magnitude first to allow their magnitude to be comparable to big magnitude numbers.
  - Take note of the range of the number system used, which, depending on the usage scenario, can be a factor of the data type, number format, register/memory width etc.
  - Apply threshold to check for overflow if possible.
  - If no overflow checks are done, take note of the number/value of accumulation/multiplication that can be done without triggering overflow.
  - To preserve as much precision as possible, always do division last as far as possible.

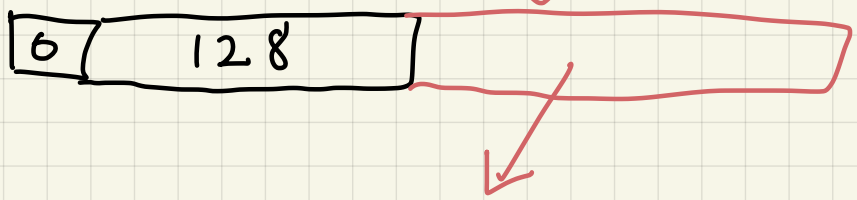
- To summarise what we have discuss in this section of overflow and accuracy.
- Some general rule of thumb that we can use when designing algorithm
  - Accumulate numbers with small magnitude first before merging them to calculation involving large numbers so that they would not be truncated off prematurely.
  - Take note of the range of the various data type, number format, register, memory etc
  - And apply threshold to check for overflow condition. Remember that if an overflow occur during computation, that automatically implies that the result obtain is wrong.
  - Try to perform division operation last to preserve as much precision as possible for the intermediate data.
- This is the last slide for the computer arithmetic chapter.

what is 2.5 float?

1) write as: $1.F \times 2^{E-127}$

$2.5 = 1.25 \times 2^{1}$

$= 1.25 \times 2^{128-127}$

∴ 

how to represent?

$2^{-1} = 1/2$

$2^{-2} = 1/4$

$2^{-3} = 1/8$

$2^{-4} = 1/16$

⋮

$0.25 = 1/4$

∴ it is one bit at 2^{-2} slot