Cx1106
Computer Organization and Architecture

Computer Arithmetic

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

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

int: whole no. | radix point: fractional | weight/expo

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

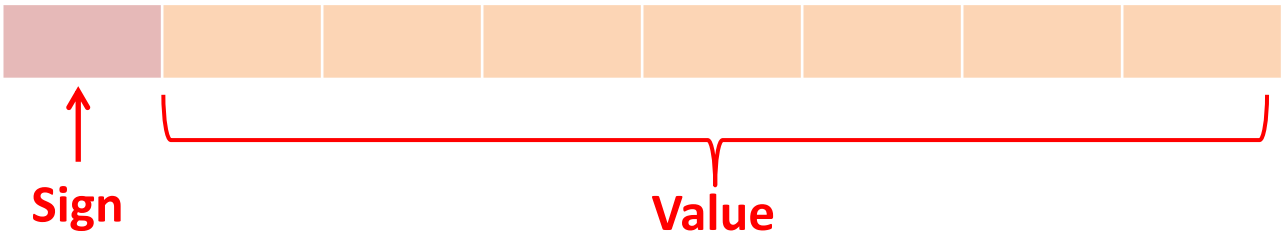
Base 16: $1234_{16} = (1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = 4660_{10}$

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.

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

Two's Complement To Decimal Conversion

Example: What is the decimal representation for 10001110_2

MS

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

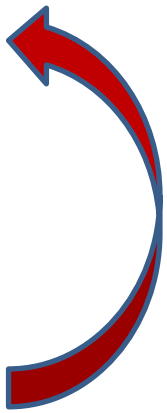
Two's Complement Representation

- 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**



- 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

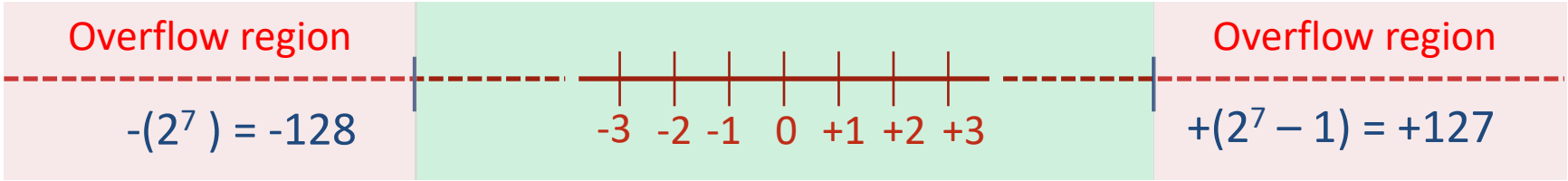
Unsigned and Signed Integer Representations

Decimal	Unsigned
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Decimal	Two's Complement
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
+0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

to be able to extend range of representation

Carry vs Overflow Example



Example: 48 + 19 = 67
Add the two binary numbers.

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

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?

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)

overflow has occured

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

adding two "true" results should be < 0

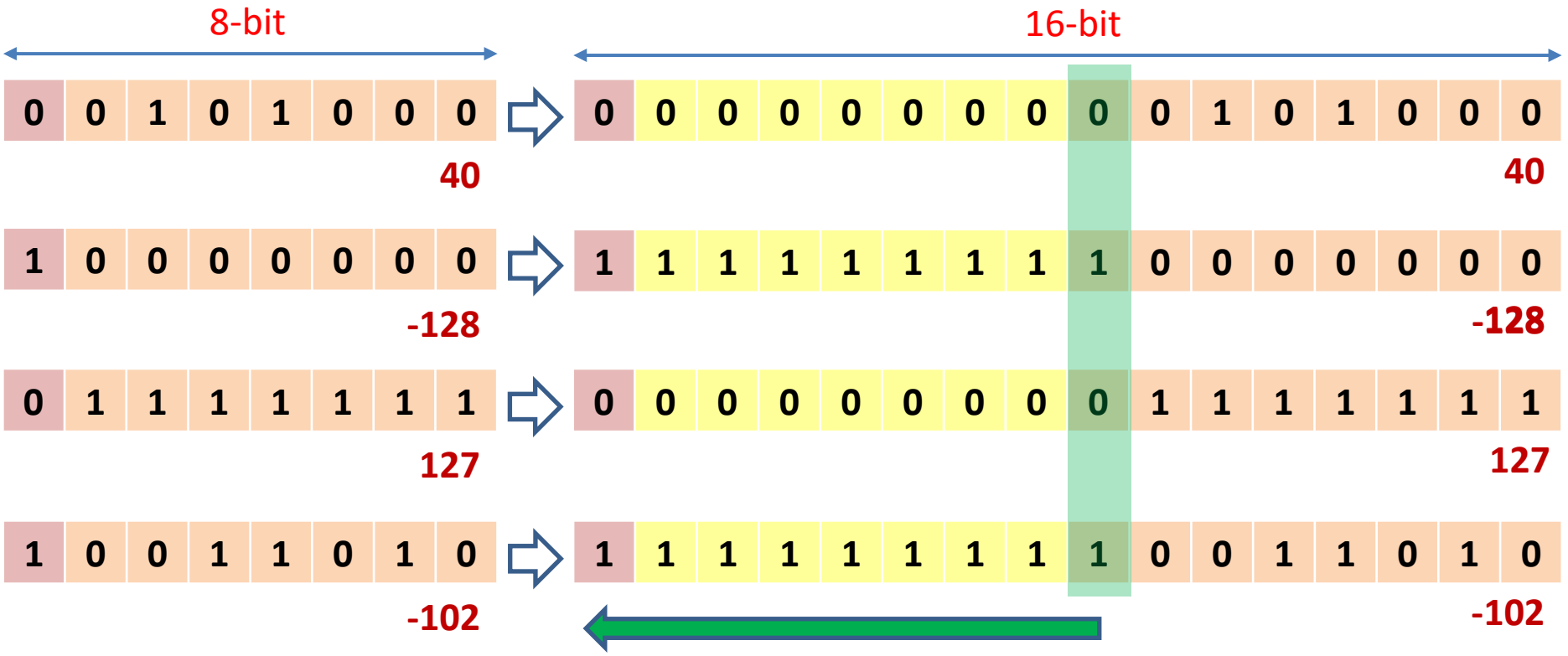
Carry vs. Overflow

- 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

Sign Extension

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



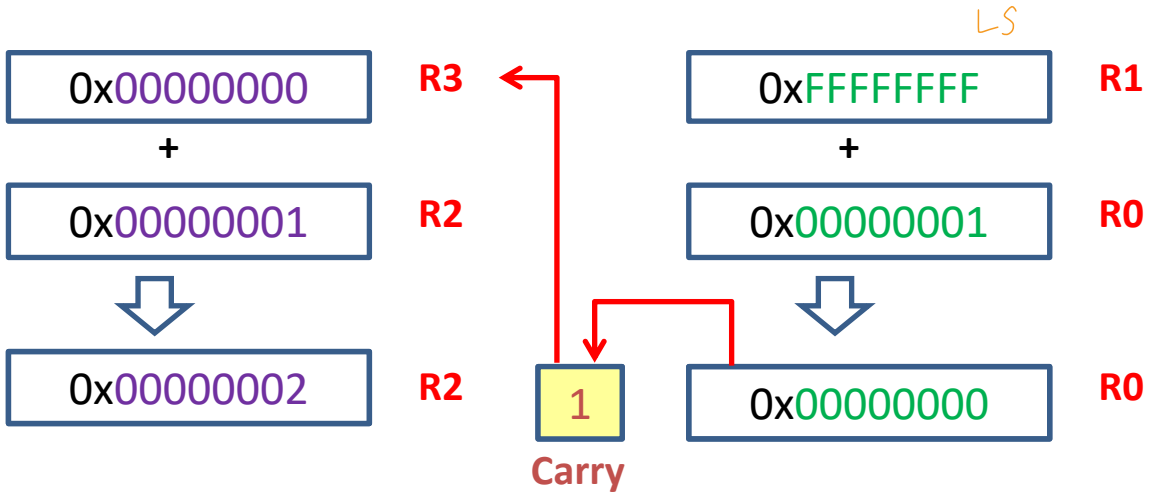
- Sign extension simply copies the sign bit (MSB) into the higher order bits

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
```



CE1006/CZ1006
Computer Organisation and Architecture

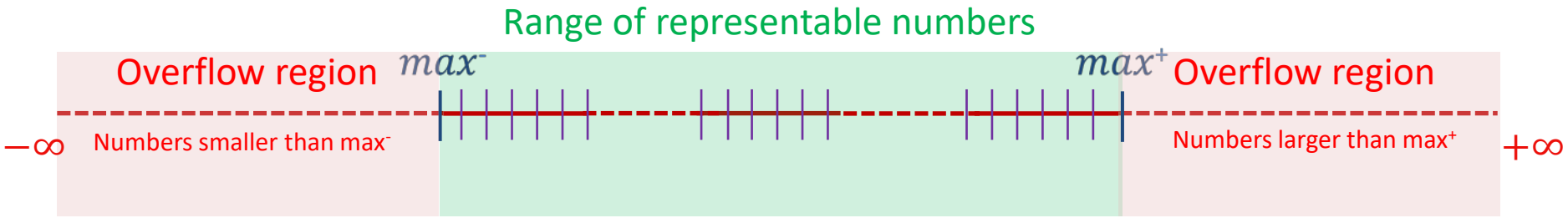
Fixed and Floating Point Number System

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

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

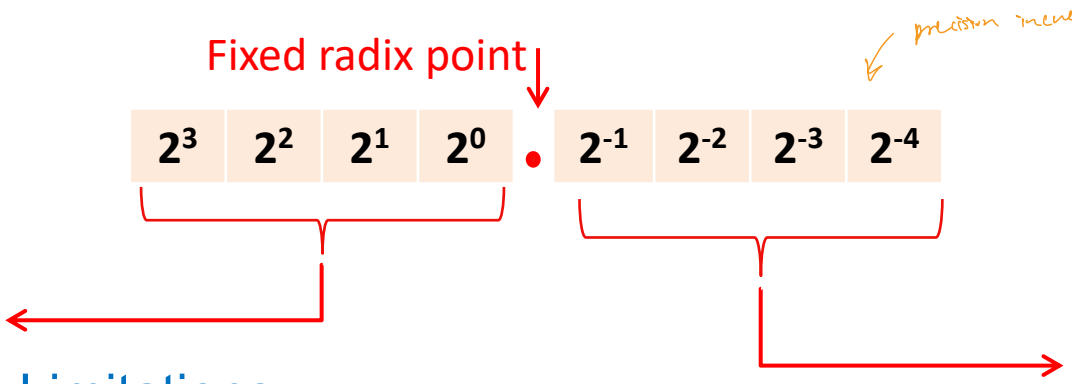
- Precision: Amount of information used to represent each number
  - Example: 1.666 has higher precision than 1.67
  - The number of tick marks provides an indication of precision



Fixed-Point Representation

- Fixed-point format can represent integer and/or fractional values

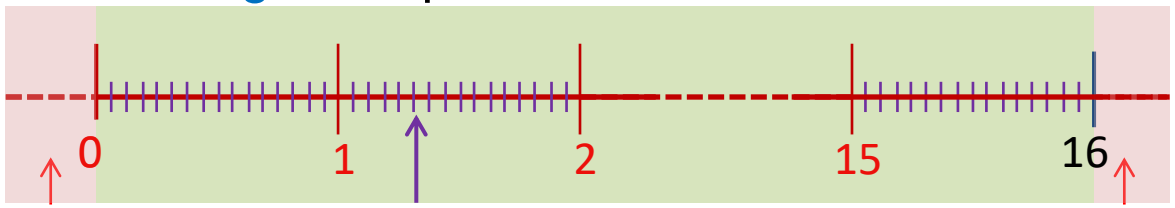
Unsigned Integer	
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15



- Limitations
  - Precision is limited by the range of integer values
  - Bits are not fully utilised for certain values (e.g. 15.5 or 1111.1000_2)

not utilised

Range of representable numbers



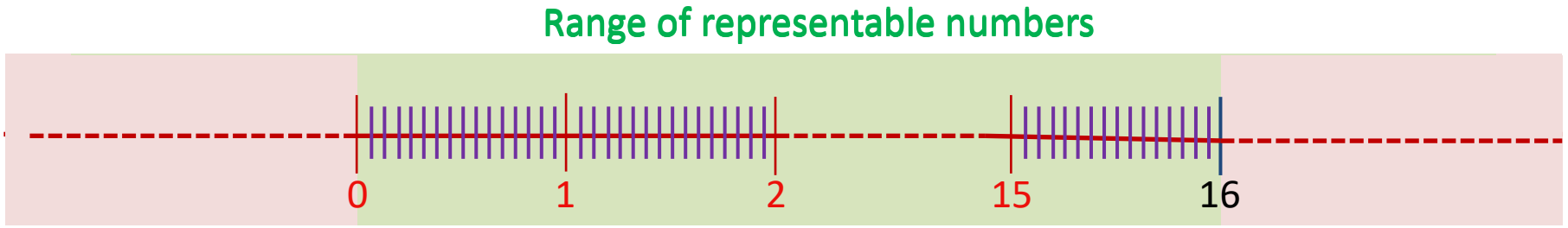
Overflow region — Uniform density (fixed resolution of 2^{-4}) — Overflow region

Fractional Values	
0000	0.0000
0001	0.0625
0010	0.1250
0011	0.1875
0100	0.2500
0101	0.3125
0110	0.3750
0111	0.4375
1000	0.5000
1001	0.5625
1010	0.6250
1011	0.6875
1100	0.7500
1101	0.8125
1110	0.8750
1111	0.9375

Range and Precision Trade-off

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

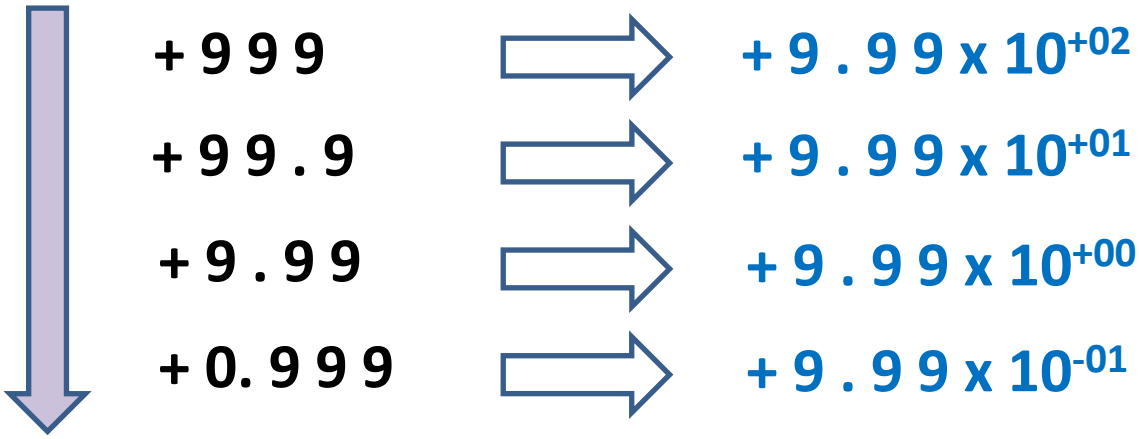
2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}



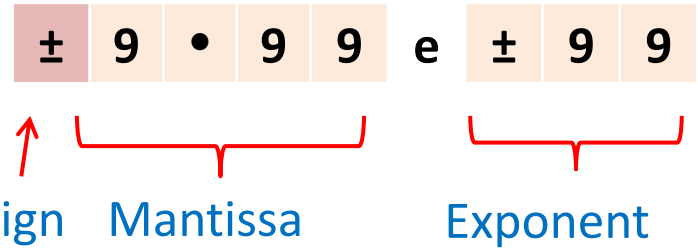
- When radix point floats from LSB to MSB
  - Range of representable numbers reduces
  - Precision increases
- The example above illustrates the concept of floating-point, but how do we represent a floating-point number?

Floating Point Representation

Precision increases for smaller numbers

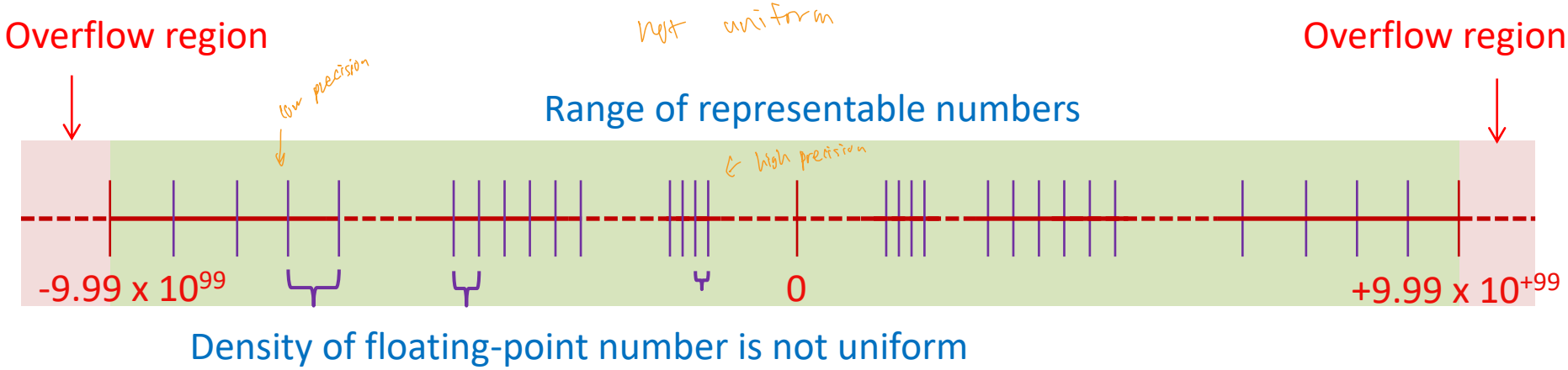


Simple decimal (Radix = 10) floating point format



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

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.

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	

Underflow

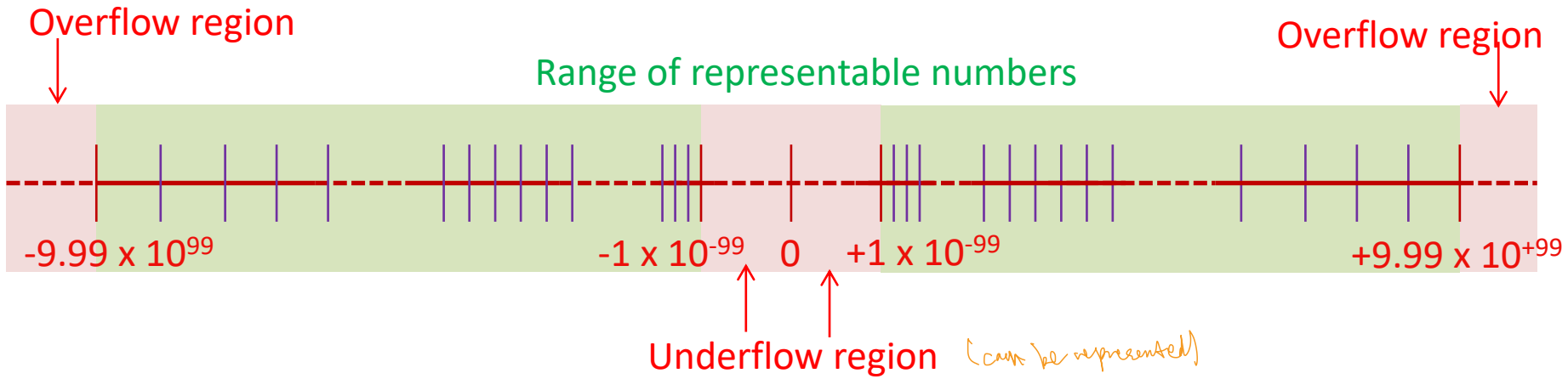
- Normalisation results in underflow regions where values close to zero cannot be represented

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.

CE1006/CZ1006
Computer Organisation and Architecture

IEEE 754

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

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

IEEE 754 Normalised Numbers

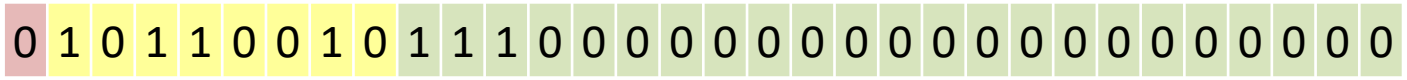
$$(-1)^S \times (1.F)_2 \times 2^{E - Bias}$$

← under the assumption that it is normalized a/r

- 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:
    $(-1)^S \times (\mathbf{1} + f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + f_4 \times 2^{-4} + \ldots)_2 \times 2^{E-\mathbf{Bias}}$

Converting Single Precision To Decimal

- Find the decimal value of these single precision number:

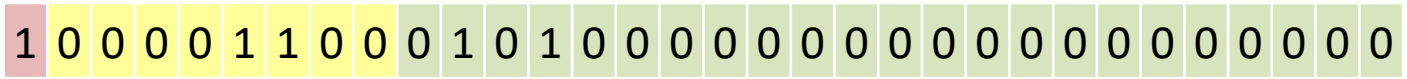


Sign = 0 (positive)

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 = $+1.875 \times 2^{51}$



Sign = 1 (negative)

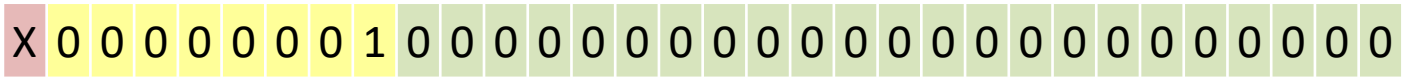
Exponent = 00001100_2 = 12; E – Bias = 12 – 127 = -115

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

Value in decimal = -1.3125×2^{-115}

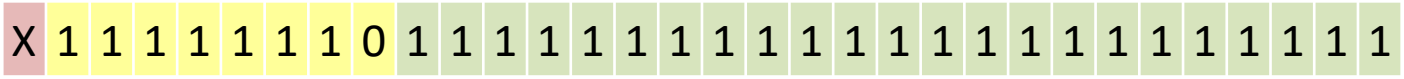
Representable Range for Normalised Single Precision

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

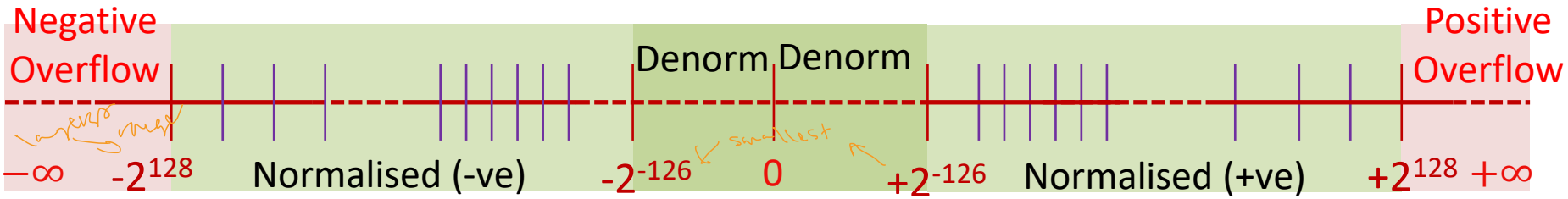


Exponent = $(00000001)_2$ = 1; E - Bias = 1 – 127 = -126
1 + Fraction = $(1.000\ldots000)_2$ = 1
Value in decimal = **1×2^{-126}**

- Largest magnitude normalised number



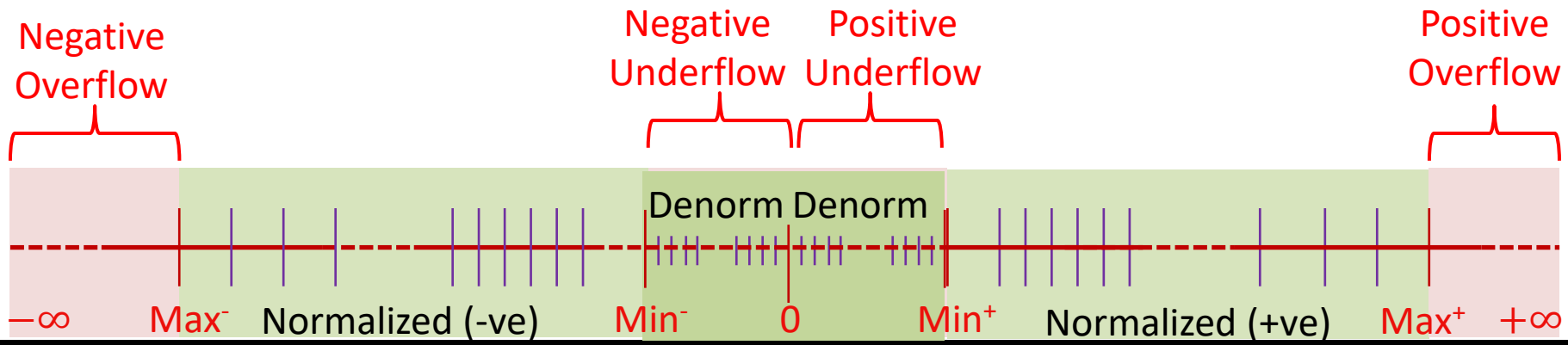
Exponent = $(11111110)_2$ = 254; E - Bias = 254 – 127 = 127
1 + Fraction = $(1.111\ldots111)_2 \approx 2$
Value in decimal = $2 \times 2^{127} \approx 2^{128}$



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

Denorm can now be represented



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.

CE1006/CZ1006
Computer Organisation and Architecture

Computer Arithmetic related consideration during coding

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

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 → make sure doesn't overflow range of data type
- 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.

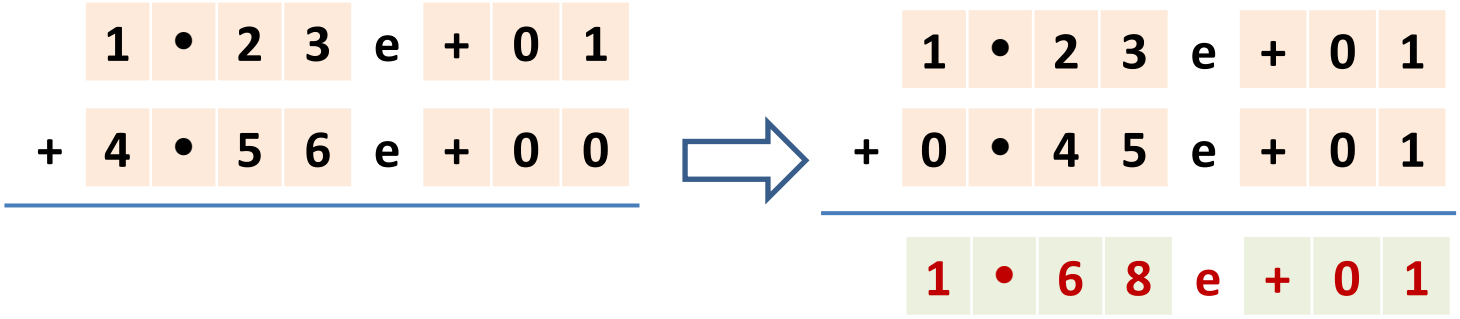
Effects of Rounding

- Rounding refers to removing of LSB(s) so that the result can fit into the representable bits.
- Rounding can be round-up, round-down or round to nearest representable number.
- The limits imposed in the width of the representable bits could be from the registers width, data type etc.
- 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 $0.004*10^1$

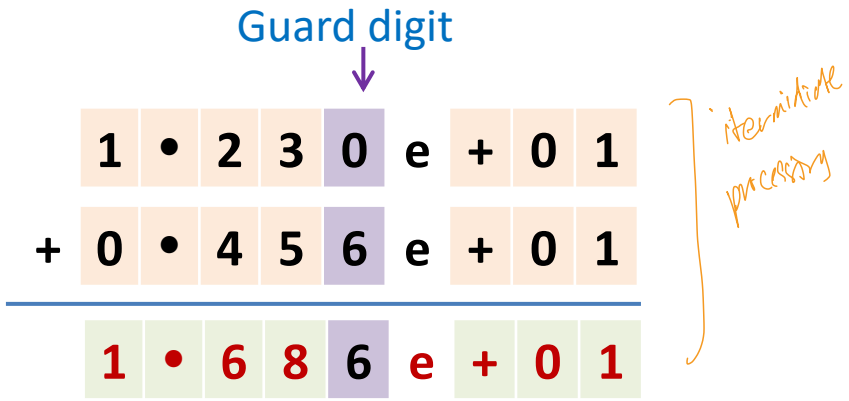
- Common to have intermediate register with width larger than regular data registers to allow intermediate processing to be done at higher precision and thus reducing the amount of rounding error.

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.

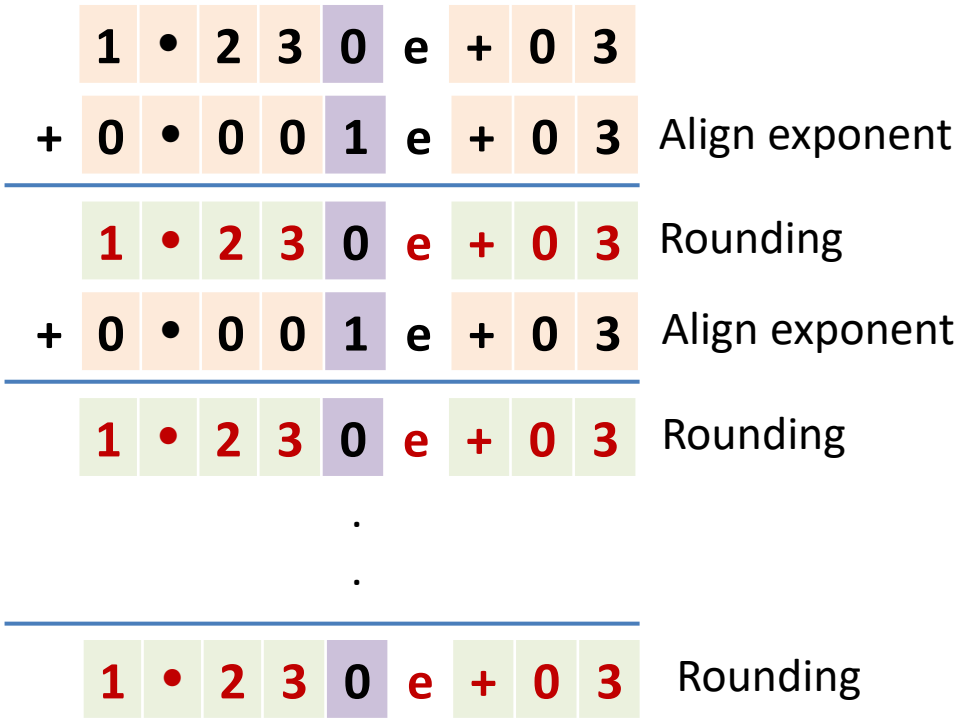


- Rounding is then performed to ensure that the result fits into the three significant digits in the mantissa

Handing numbers with different magnitudes

- Suppose we want to compute the following :

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



Handing numbers with different magnitudes

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

better approach.

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

Key: add smaller numbers first

	1.000e+00	
+	1.000e+00	
	2.000e+00	
+	1.000e+00	
	3.000e+00	
+	1.000e+00	
	4.000e+00	
+	1.000e+00	
	0.005e+03	Align
+	1.230e+03	
	1.240e+03	Rounding

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.

Live lecture

Which of the following statement(s) are false for processor pipeline?

A. Pipelining involves splitting an instruction into simpler stages and execute these stages in parallel

B. Pipeline requires independent resources to operate each of the pipeline stages to yield good efficiency

✓ C. Pipeline allows stages of the same instruction to be executed in parallel

D. Pipeline's efficiency is maximised if there are minimum conflicts during execution

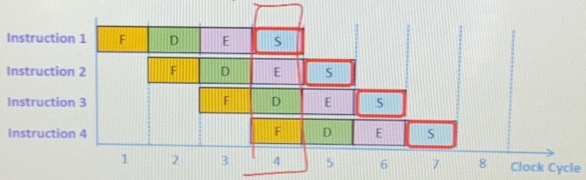
**Insufficient resources to execute each of the pipeline stages will lead to

✓ A. Decrease in pipeline efficiency → time extends to push out instruction

✓ B. Resource conflicts

C. Increase in pipeline efficiency

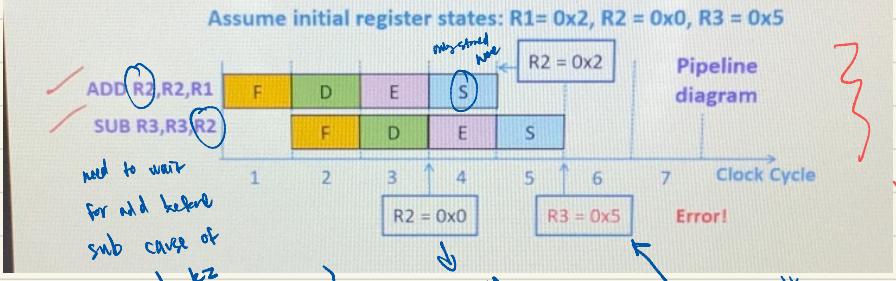
D. Slower processor speed



**What is the effect of data dependency conflict?

A. Data in one register is corrupted by data in another register
✔B. Data in register/memory is not updated in time for its usage.
✔C. Incorrect results obtained from executing affected instructions
D. Data in one instruction gets modified half way through its execution

↳ not considered in this module → correct but



need to wait for add before sub since of operand R2 (data dependency problem)

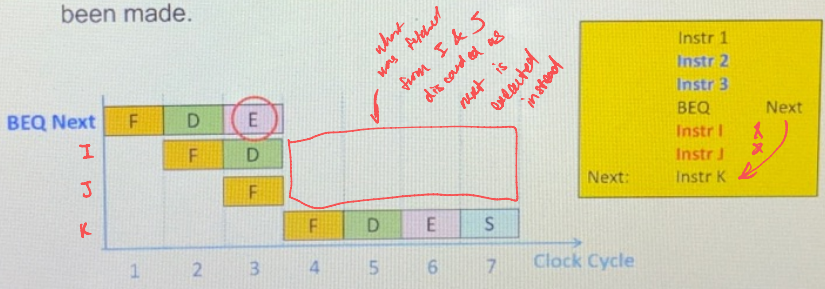
suppose to be 0x2

wrong result

diagram shows to verify

Which statement below is FALSE?

A. Delayed branching utilize delay slots to execute code.
B. In CPU that supports delayed branching, instructions in delayed slots are not flushed away.
✔C. There are no restrictions to the type of instructions that can be inserted in the delay slot.
D. The delay slot instruction is executed after the branch decision has been made.



Unless get delay instruction executed
↳ delayed branching ; instruction completes and not discarded

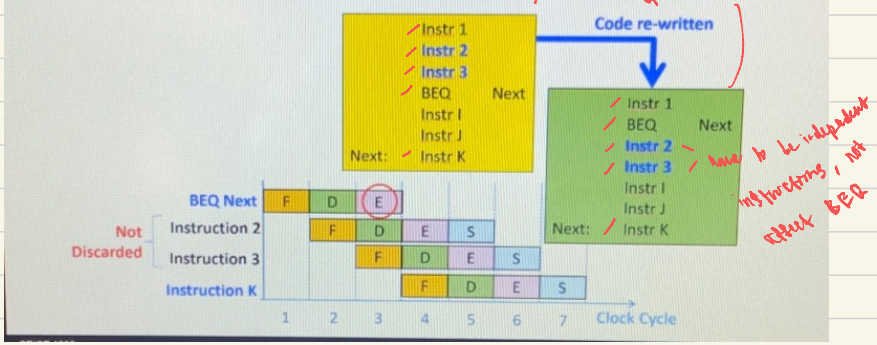
What should the programmer do if he can't find sufficient instructions to be placed in the delay slots?

✓ A. Patch with NOP instructions

B. Do nothing and let the processor hardware handle it

C. Patch with same instruction used in the other delay slot

D. Patch with the same Branch Instruction

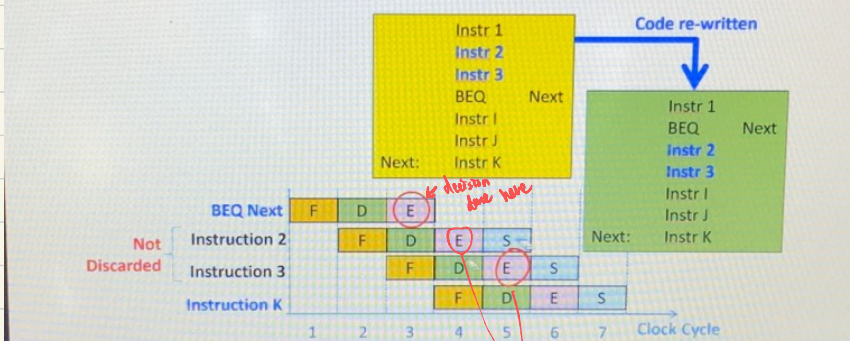


Will the delay slot instructions be able to influence the branch decision?

A. Yes

✓ B. No

C. Depend on the instructions placed in the delay slots



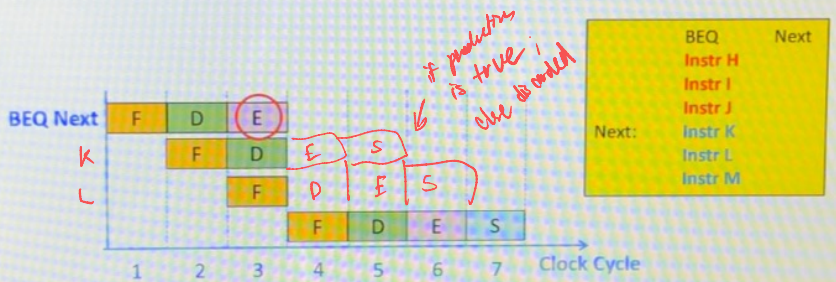
In Dynamic branch prediction, how would the processor behave if it predicts that the branch would be taken?

A. Flush the pipeline early to prepare for the jump instruction

✓B. Load the instruction at the jump target instead of the next instruction in the code

C. Load instructions into the delay slots

D. Discard the instructions after the branch



but question (but similar to exam qns)

Pipelines

4. Consider a processor (not ARM processor) with 4 pipeline stages: Fetch Instruction (F), Decode (D), Execute (E) and Store (S). Assume that

- Branch target address is calculated at the execute stage
- Instruction length for every instruction is one word long
- Each pipeline stage takes 1 cycle to complete
- No Resource Conflicts ← usually added in case it is a correct answer lol
- Delayed Branching is enabled

Identify and describe ALL pipeline conflicts the code in Figure 9.3 has when executed in the pipeline processor above. Suggest workaround for pipeline conflicts identified.

4 stage pipeline, 1 cycle, each instruction 1 word long

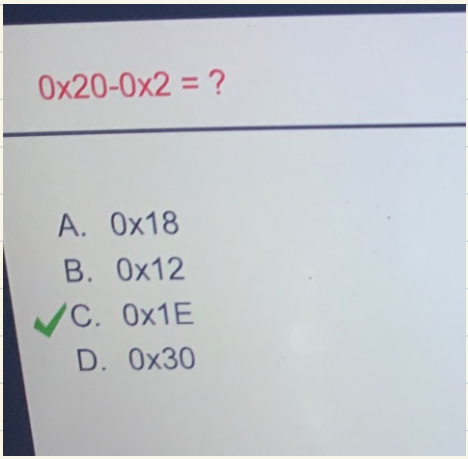
```
        MOV   R6, #0x900 ; I1
        MOV   R5, #0     ; I2
        MOV   R4, #0x800 ; I3
        MOV   R3, #0x300 ; I4
Loop    LDR   R0, [R3]   ; I5
        LDR   R1, [R4]   ; I6
        ADD   R2, R1, R0 ; I7
        ADD   R5, R5, #1 ; I8
        CMP   R5, #5     ; I9
        BNE   Loop       ; I10
        ADD   R4, R4, #1 ; I11
        ADD   R3, R3, #1 ; I12
        ADD   R4, R4, R2 ; I13
        STR   R2, [R6]   ; I14
```

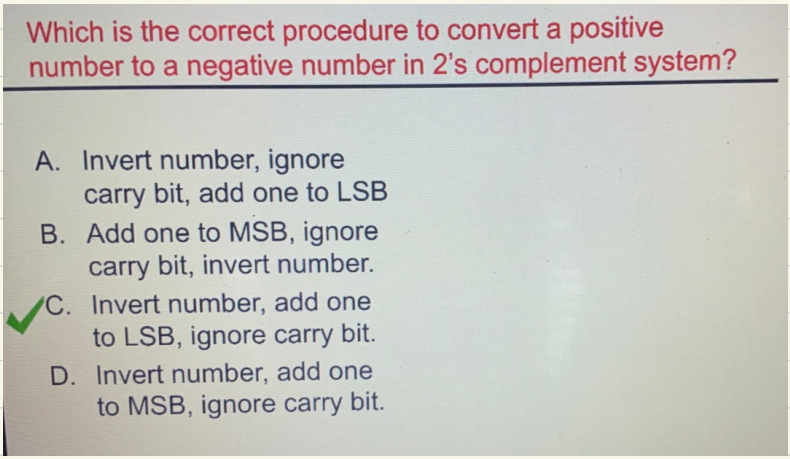
data dependencies

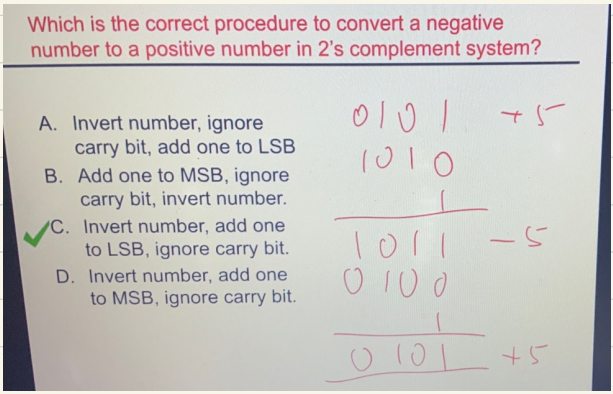
↳ data dependency (something about R2 being passed?)

STR & LDR

⇒ 9 stage instruction


0x20-0x2 = ?
A. 0x18
B. 0x12
C. 0x1E
D. 0x30


Which is the correct procedure to convert a positive number to a negative number in 2's complement system?
A. Invert number, ignore carry bit, add one to LSB
B. Add one to MSB, ignore carry bit, invert number.
C. Invert number, add one to LSB, ignore carry bit.
D. Invert number, add one to MSB, ignore carry bit.


Which is the correct procedure to convert a negative number to a positive number in 2's complement system?
A. Invert number, ignore carry bit, add one to LSB
B. Add one to MSB, ignore carry bit, invert number.
C. Invert number, add one to LSB, ignore carry bit.
D. Invert number, add one to MSB, ignore carry bit.
0101 +5
1010
1
1011 -5
0100
1
0101 +5

Which of the following is/are true for carry bit?

i: Carry bit is set when a '1' is carried into the MSB of the result during computation

ii: Carry bit is set when a '1' is carried out of the MSB of the result during computation

iii: Carry bit set implies error in a signed number system

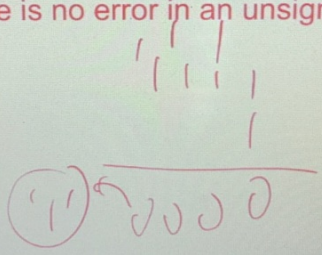
iv: Carry bit set implies there is no error in an unsigned number system.

A. i

✓B. ii

C. i, iii, iv

D. ii, iv



iii not true indicates error

**Which of the following is/are true for overflow bit?

A. Overflow bit is set when a '1' is carried into the MSB of the result during computation

✓B. To detect overflow, the signed bit of the operands and result has to be evaluated

✓C. Overflow bit set implies that there is an error in the computation in a signed number system

✓D. Overflow implies insufficient memory to contain the result

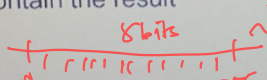
→ not even a carry

→ determines if V flag is to be set or not

V = 1 → error in signed number system

if result true, overflow

↳ eg



carry purpose

- indicated error in unsigned number
- Multi-Precision Arithmetic

Why do we only need to cater for 1 carry bit when performing multi-precision addition?

A. Only one carry bit is available in the processor
✓B. Addition of 2 n-bit numbers will yield maximum (n+1) bit result.
C. You can only carry one bit at a time
D. Only one carry bit is defined in arithmetic computation

4-bit

5-bit

1 1 1 1
1 1 1 1 0

there will always be one carry bit (never >1)

Which of the following statement is FALSE?

A. 2.5 can be represented in a fixed point system.
B. 2.5 is not an integer.
C. 2.5 can be represented in a floating point system.
✓D. 2.5 fully utilise the memory space provided in a binary numbering system with 4 integer and 4 fractional bits

$1.F * 2^{E-bias}$

fixed point system

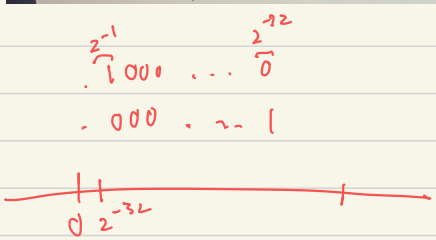
0 0 1 0 . 1 0 0 0

**What is the advantage of IEEE754 floating point representation over a fixed point representation?

- ✓ A. Larger range than fixed point number given the same data bits representation
- B. Require less complex hardware to implement
- C. Allow uniform precision over the entire range
- ✓ D. Higher precision than fixed point number for small numbers, given the same data bits representation

E.g.
32 fixed
32 IEEE 754

same no. of bits, range of this is bigger though



$1.F * 2^{E-bias}$

$* 2^{-128}$

precision is uniform for fixed point

Why do we need a special representation of the value ZERO in the IEEE754 standard?

- A. Its special
- B. Easier to relate Zero with a representation of consist of all 0's
- ✓ C. Zero cannot be represented in the format used in the normalised mode
- D. So that computation will yield Zero regardless to the value of sign bit

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

Why do we need normalization of floating point numbers?

A. To align the radix point for ease of performing addition → align exponent

✓ B. To standardise the representation format to avoid synonymous representation

C. To standardise the exponent representation format for multiplication operation

D. To represent numbers very close to zero → issue with normalization

floating pts has many representations for 1 number

that's why need to normalize

**What is the effect of normalization?

→ normal can't do

A. Numbers very close to zero can be represented

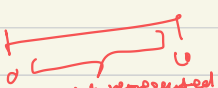
✓ B. Numbers very close to zero cannot be represented

✓ C. Each number only has one form of representation

D. Each number could have multiple forms of representation → opposite of C

cannot be 0

$1.0 * 10^{1}$ → smallest no in this case



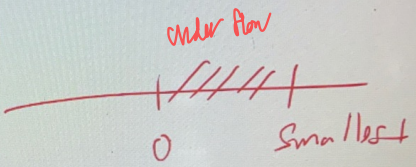
What is underflow?

A. Insufficient data bits to fill the memory

✓ B. Crossing into region closed to zero that is not representable by the number representation system used

C. A negative overflow

D. Insufficient data bits to represent values closed to zero



Why are the function of Guard bits?

A. To prevent result from overflowing

B. To check if result has overflowed

✓ C. Allows the precision of intermediate data to be preserved so as to obtain a more accurate final result.

D. To prevent loss of MSB of result

Accumulator : special registers that are long in width (in ARM, > 32 bits)

Which of the following statement is TRUE?

A. Accumulation is usually carried out after doing division to preserve accuracy.

accuracy lose

✓B. Division leads to truncation of LSBs.

→ left shift; LSB truncated out

C. Multiplication is computationally more intensive than division.

→ is the most complex; not multiplication

D. Performing all required accumulation before division always gives the most accurate result.

→ provided overflow does not occur

not always

Which of the following will give rise to the smallest single precision IEEE754 number (normalised mode)?

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

A. F = 0,
E = 1111 1110

✓B. F = 000 0000 0000 0000 0000 0000,
E = 0000 0001

C. F = 100 0000 0000 0000 0000 0000,
E = 1111 1110

D. F = 0,
E = 1111 1111

$1.0 * 2^{1-127}$

$1.1 * 2^{25-127}$

$1 * 2^{-126}$

$1.1 * 2^{127}$