CE1006/CZ1006

Computer Organisation and Architecture

Computer Arithmetic

Oh Hong Lye

Lecturer

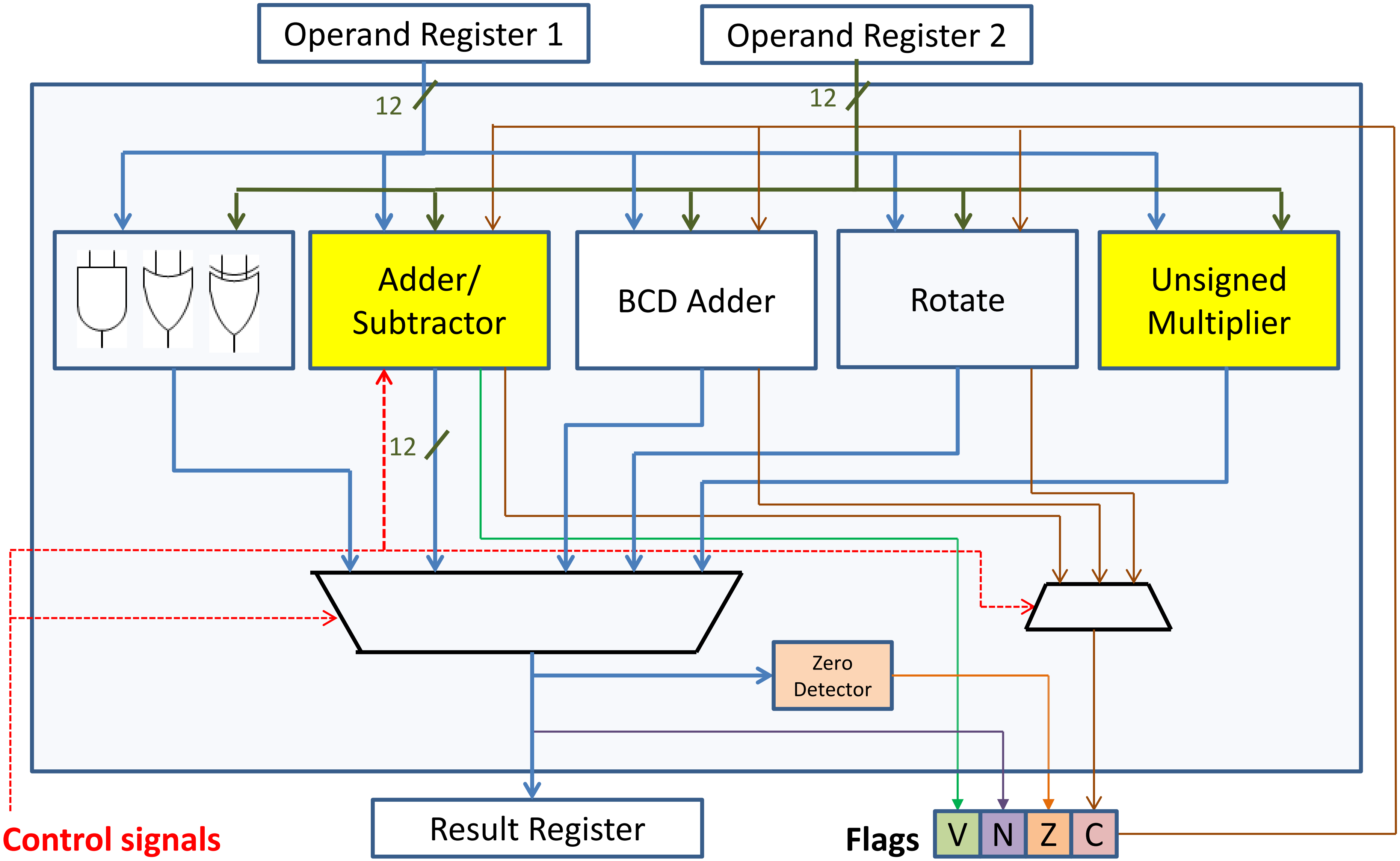
SCSE, Nanyang Technological University.

CE1006/CZ1006
Computer Organisation and Architecture

Addition – Single and Multiple Precision

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

ALU of VIP Processor



Multi-Precision Arithmetic

- How can we add operands that are larger than 12-bits (e.g. 36 bit operands)?
  - Note that we only have a single 12-bit adder in the ALU

Example: Adding two 36-bit operands in the VIP Processor

	← 12-bit →	← 12-bit →	← 12-bit →
	0 1 1 1 1 1 0 1 0 0 1 1	1 0 1 1 0 1 1 0 0 0 1 1	1 1 0 0 0 1 0 1 1 0 1 1
+	0 1 0 0 0 1 1 0 1 1 0 1	1 1 1 0 0 0 0 1 1 0 0 1	1 0 1 1 1 1 0 0 0 1 0 1

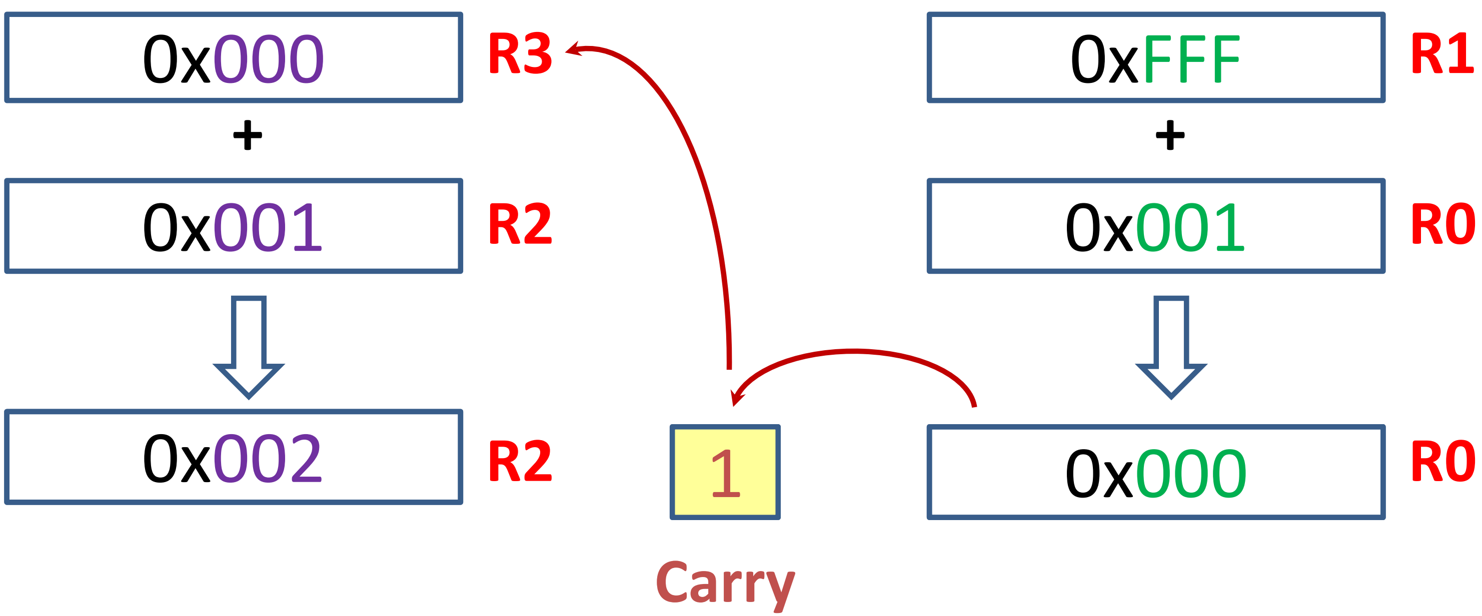
- The solution is to **reuse** the 12-bit adder for multi-precision addition
- **Multi-precision arithmetic** involves the computation of numbers whose precision is larger than what is supported by the maximum size of the processor register (**Single-Precision**)

Multi-Precision Addition in VIP

Example:

```
  0x000FFF
+ 0x001001
----------
  0x002000
```

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



CE1006/CZ1006

Computer Organisation and Architecture

Multiplication

Oh Hong Lye

Lecturer

SCSE, Nanyang Technological University.

Binary Multiplication (Partial Product)

Example: 9_{10} x 11_{10} (unsigned)

Test multiplier bit starting from LSB

If multiplier bit = 1, **add** multiplicand to result

Shift multiplicand

Repeat until all the multiplier bits have been tested

					1	0	0	1	Multiplicand (9_{10})
				x	1	0	1	1	Multiplier (11_{10})
					1	0	0	1	Add multiplicand and shift
				1	0	0	1		Add multiplicand and shift
			0	0	0	0			Shift multiplicand
+		1	0	0	1				Add multiplicand and shift
	0	1	1	0	0	0	1	1	Result (99_{10})

Multiplying two n-bit binary values together may require up to 2n bits to hold the results

Software Multiplication in VIP

- Assuming that a multiplier is **not** present in VIP, how can we implement multiplication in **assembly code**?

Rotate multiplier right with carry and test carry

↓

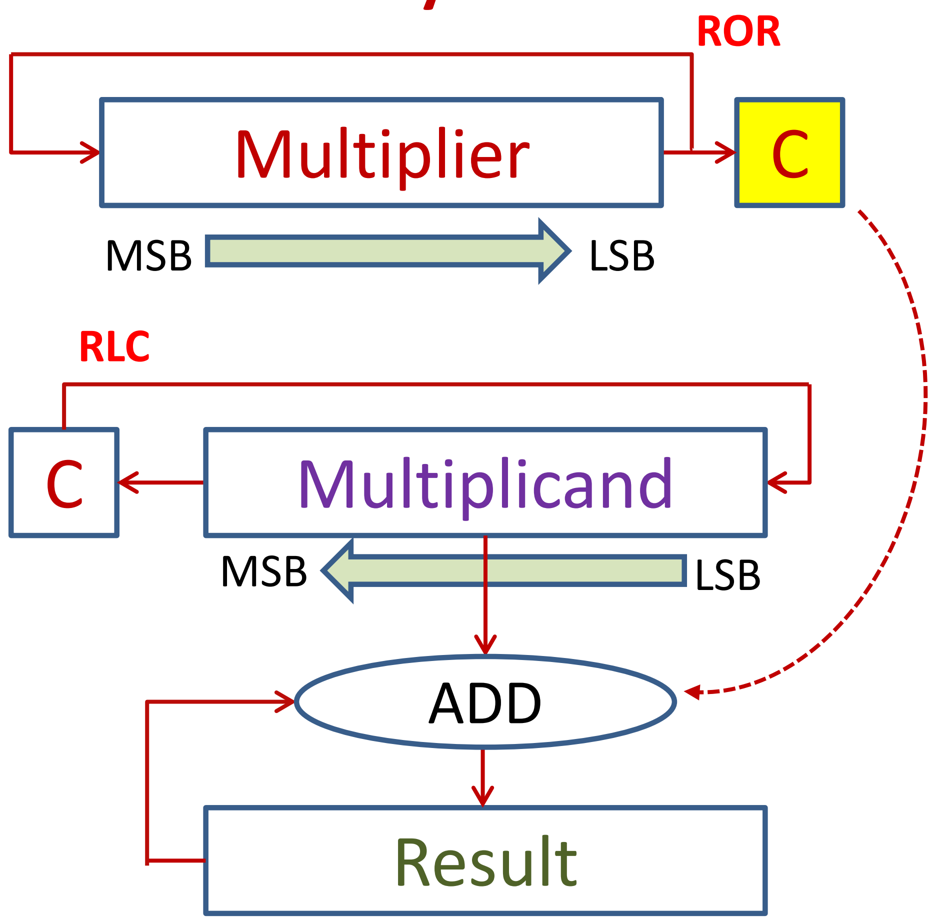
If carry = 1, add multiplicand to result

↓

Rotate multiplicand left

↓

Repeat until all the multiplier bits have been tested



Software Multiplication in VIP

Example: 9_{10} x 11_{10} (unsigned)

1. Rotate multiplier right with carry and test carry
2. If carry = 1, add multiplicand to result
3. Rotate multiplicand left
4. Repeat until all the multiplier bits have been tested

Initial State

R2	0000 0000 1011	Multiplier
R3	0000 0000 1001	Multiplicand
R0	0x000	Result
AR	0x004	Number of Multiplier bits

C

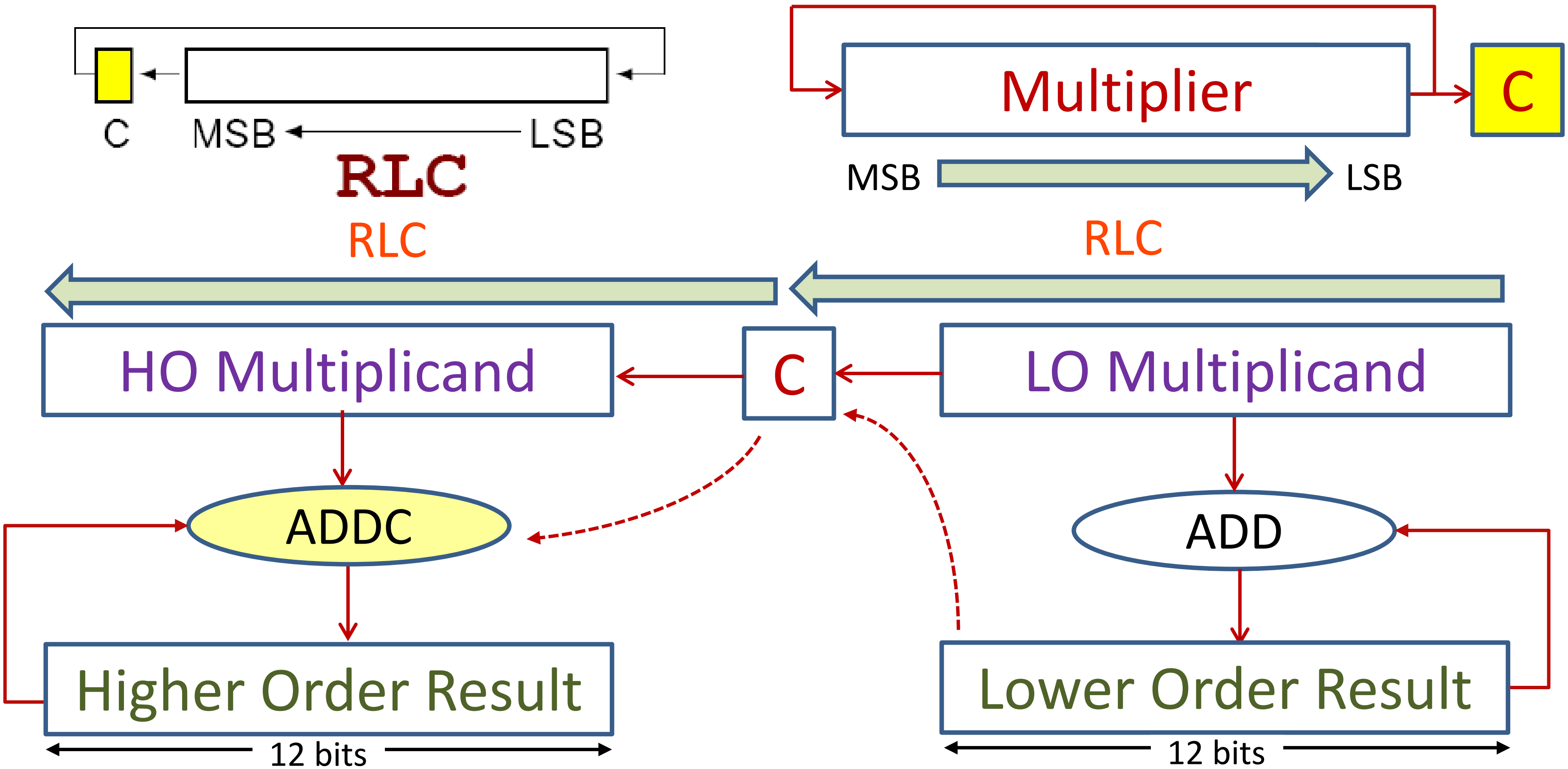
+

-1

```
Start   ROR R2          1
        JNC Skip
        ADD R0,R3       2
Skip    BCSR 1
        RLC R3          3
        JDAR Start      4
```

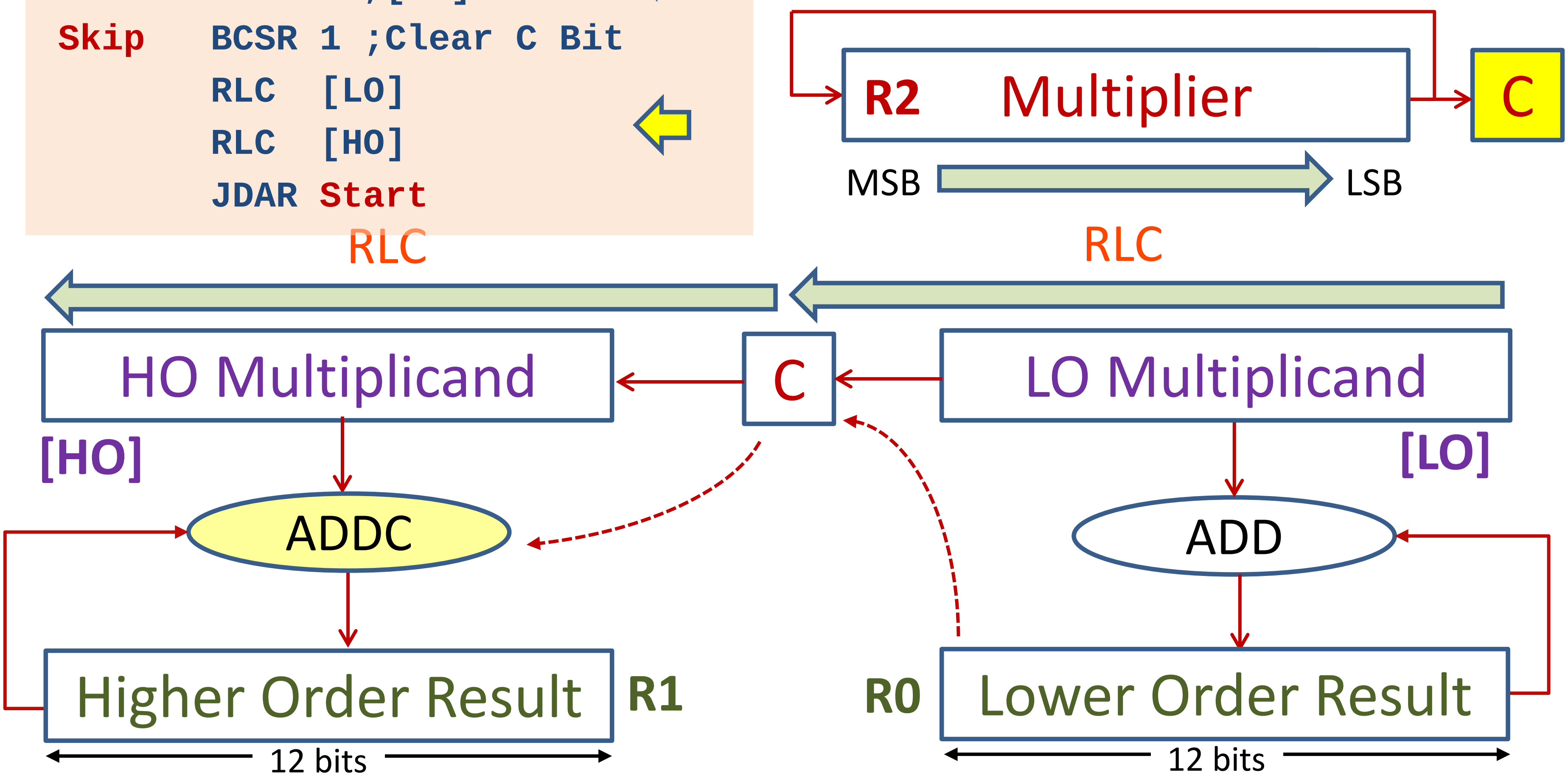
Software Multiplication in VIP

- Recall that when we multiply two **n**-bit binary values, we may require as many as **2n** bits to hold the results
- How can we multiply two **12-bit operands** in software?



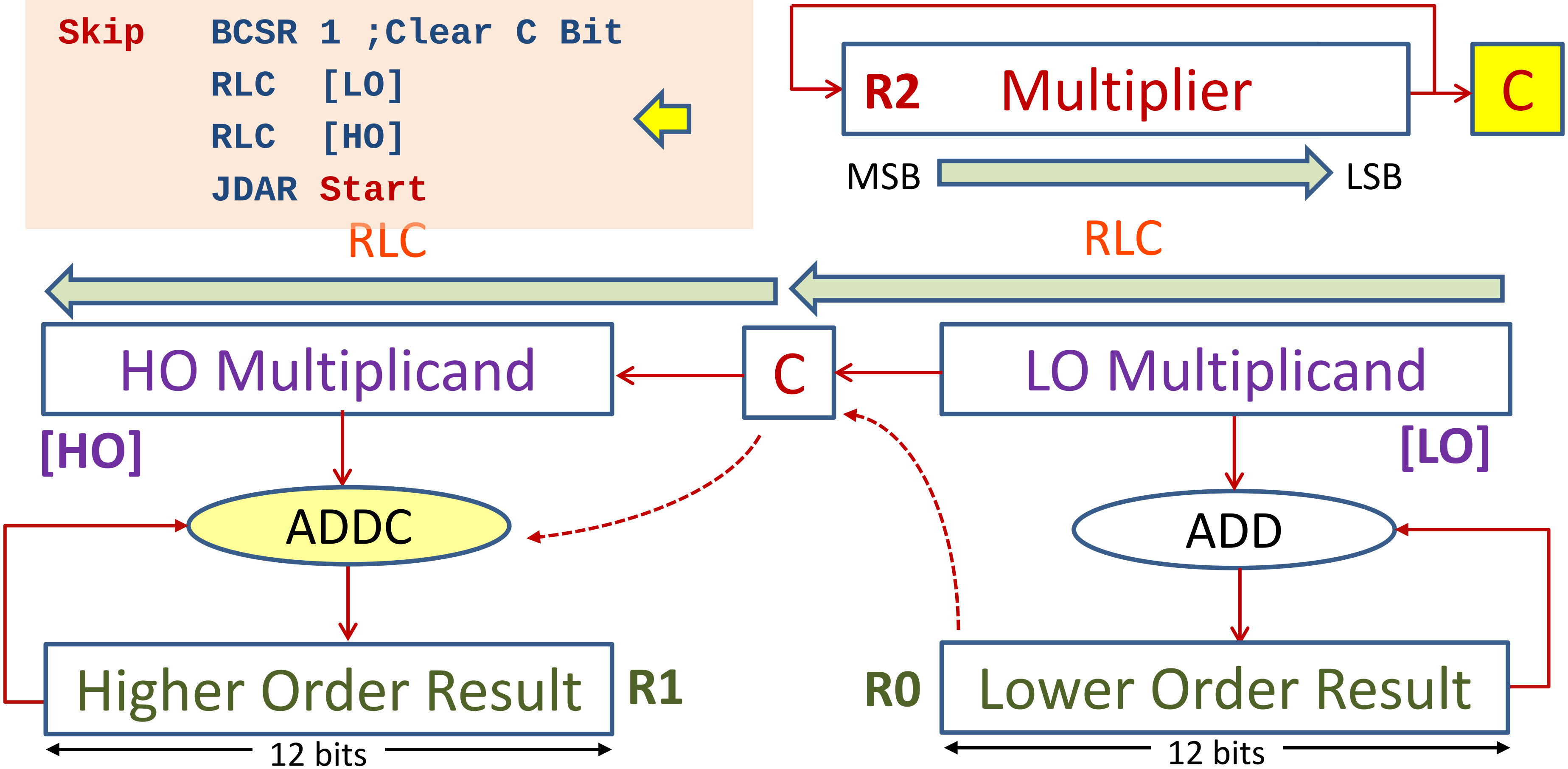
Software Multiplication in VIP

```
Start   ROR  R2
        JNC  Skip
        ADD  R0,[LO]
        ADDC R1,[HO]
Skip    BCSR 1 ;Clear C Bit
        RLC  [LO]
        RLC  [HO]
        JDAR Start
```



Software Multiplication in VIP

```
Start   ROR  R2
        JNC  Skip
        ADD  R0,[LO]
        ADDC R1,[HO]
Skip    BCSR 1 ;Clear C Bit
        RLC  [LO]
        RLC  [HO]
        JDAR Start
```



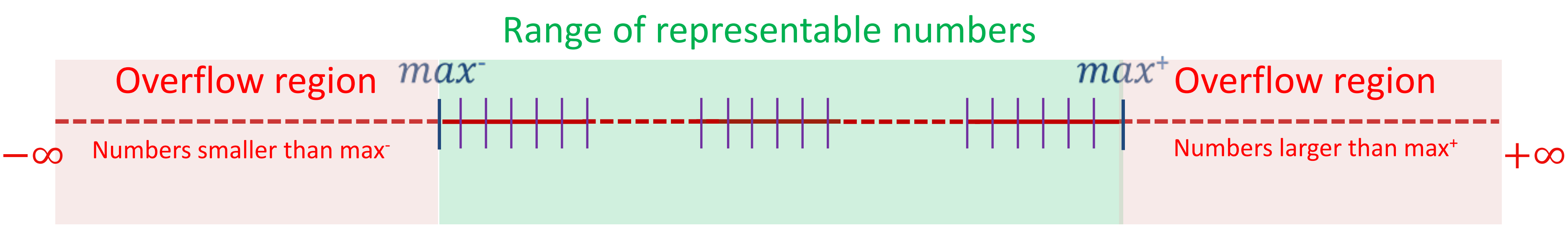
CE1006/CZ1006
Computer Organisation and Architecture

Fixed and Floating Point Number System

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

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

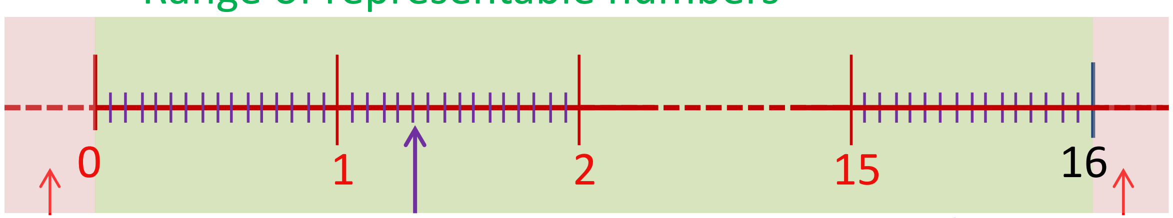
Unsigned Integer	
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Fixed radix point

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

- **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)

Range of representable numbers



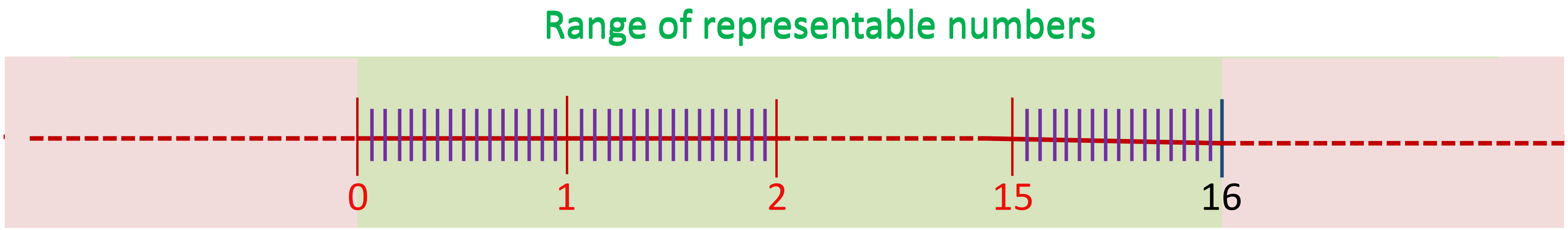
Overflow region Uniform density (fixed resolution of 2^{-4}) Overflow region

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 if the radix point can **float** between digits in a number when needed?

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

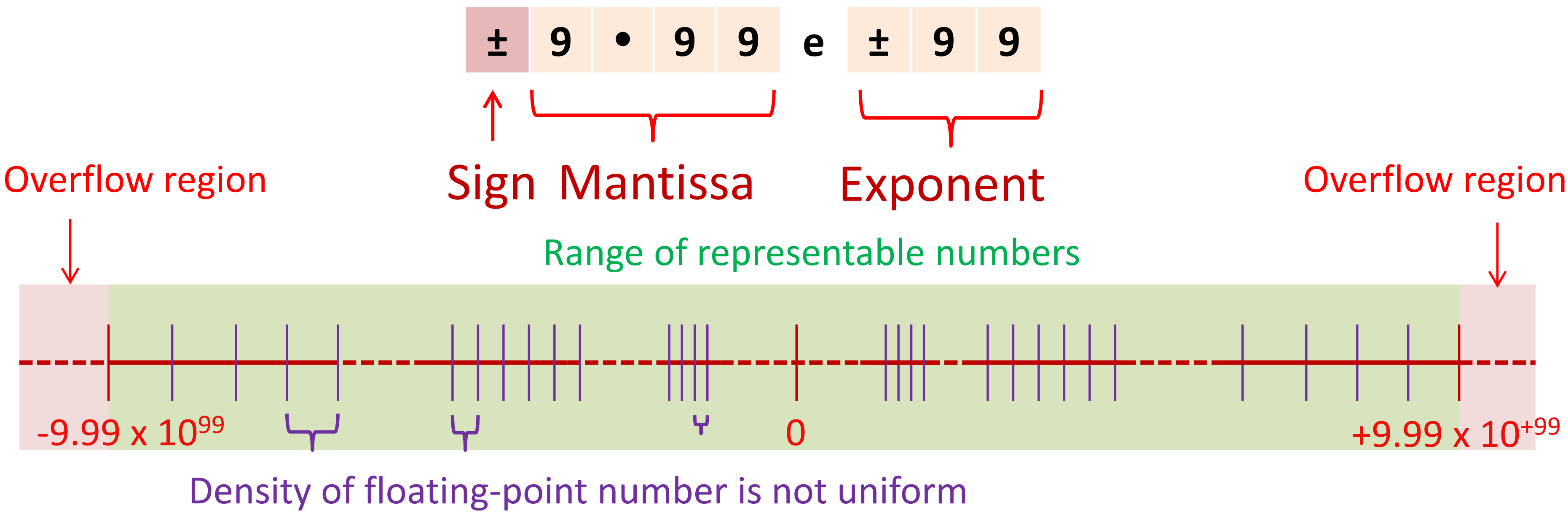
+999	$\Rightarrow$	$+9.99 \times 10^{+02}$
+99.9	$\Rightarrow$	$+9.99 \times 10^{+01}$
+9.99	$\Rightarrow$	$+9.99 \times 10^{+00}$
+0.999	$\Rightarrow$	$+9.99 \times 10^{-01}$

Simple decimal floating point format

$$\underset{\text{Sign}}{\pm}\ \underbrace{9 \bullet 9\ 9}_{\text{Mantissa}}\ e\ \underbrace{\pm\ 9\ 9}_{\text{Exponent}}$$

- **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 ($-9.99e^{+99}$ to $+9.99e^{+99}$)
- **Size of exponent** determines **range** of representable numbers
- **Size of mantissa** and **value of exponent** determines **precision** of values
- **Small numbers** can be represented with **good precision** while sacrificing precision for larger numbers to achieve greater range

CE1006/CZ1006
Computer Organisation and Architecture

Normalisation, Underflow and Rounding

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

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	

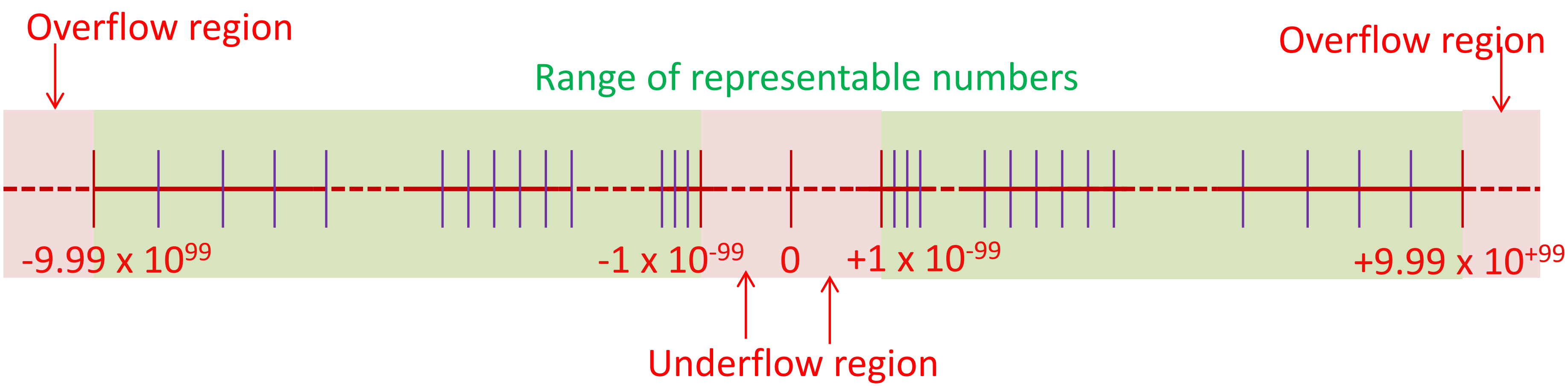
- **Normalisation** is necessary to avoid synonymous representation by maintaining one **non-zero digit** before the radix-point
  - In decimal number, this digit can be from 1 to 9
  - In binary number, this digit should be 1
- **Normalisation** can **maximise** number of bits of precision

±	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.**

Guard Bit and Rounding

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

	1	•	2	3	e	+	0	1
+	4	•	5	6	e	+	0	0

	1	•	2	3	e	+	0	1
+	0	•	4	5	e	+	0	1
	1	•	6	8	e	+	0	1

- To improve accuracy, guard digits (bits) are used to maintain precision during floating point computations

Guard digit

	1	•	2	3	0	e	+	0	1
+	0	•	4	5	6	e	+	0	1
	1	•	6	8	6	e	+	0	1

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

1	•	6	8	6	e	+	0	1

$\Rightarrow$

1	•	6	9	e	+	0	1

Rounding Error

- Floating point (or fixed point) numbers are inherently inaccurate as **limited number of bits** in the computer are used to represent very large or very small numbers
  - Example: $0.1_{10} = 0.0001100110011 \ldots_2$
  - Since this number cannot be represented in a finite amount of space, it has to be rounded down when it is stored (this introduces **rounding error**)

- Suppose we want to compute the following :

$1.23\text{x}10^3 + 1.00\text{x}10^0 + 1.00\text{x}10^0 + 1.00\text{x}10^0 + 1.00\text{x}10^0 + 1.00\text{x}10^0$

	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

Increasing Accuracy in Addition/Subtraction

Example:

$1.00\text{x}10^0 + 1.00\text{x}10^0 +$
$1.00\text{x}10^0 + 1.00\text{x}10^0 +$
$1.00\text{x}10^0 + 1.23\text{x}10^3$

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

CE1006/CZ1006
Computer Organisation and Architecture

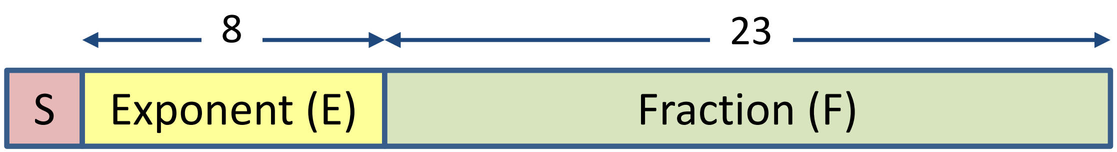
IEEE 754

Oh Hong Lye
Lecturer
SCSE, Nanyang Technological University.

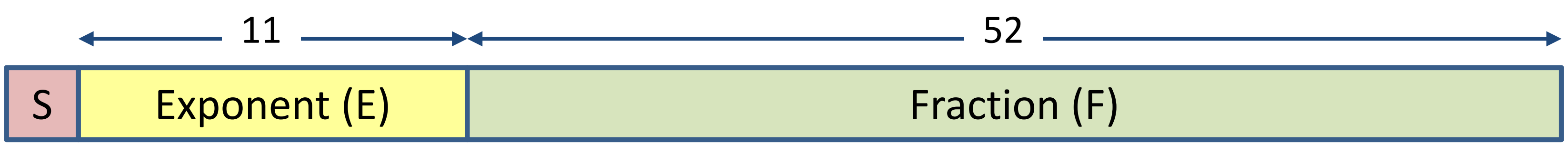
IEEE 754 Floating Point Standard

- 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

S	E	Fraction (F) = $f_1f_2f_3f_4$....

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

- 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:
    $(-1)^S$ x $(\mathbf{1} + f_1 x2^{-1} + f_2 x2^{-2} + f_3 x2^{-3} + f_4 x2^{-4} + \ldots)_2$ x $2^{E\text{-}\mathbf{Bias}}$

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}

Representable Range for Normalised Single Precision

In normalised mode, exponent is from 00000001 to 11111110

- Smallest magnitude normalised number

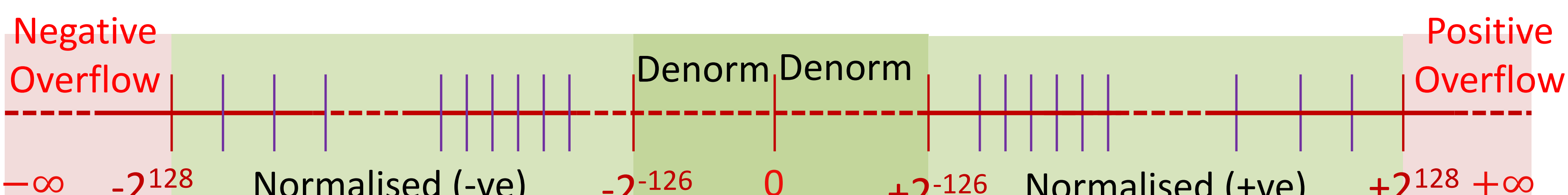
X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

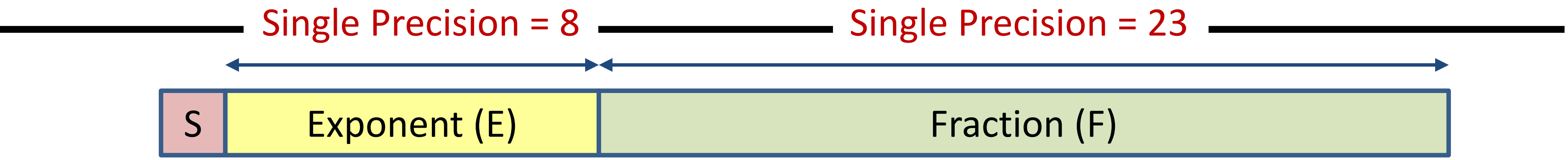
- Largest magnitude normalised number

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	1	1

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



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