Exercise Session 10

Computer Architecture and Systems Programming

Autumn Semester 2024

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

In this session...

- Questions Regarding Attacklab
- Lecture Recap Floating Point
  - FP Basics
  - FP: Normalised, Denormalised, Special Values
  - FP: Rounding
  - FP: Operations
  - SSE3
- Exam Questions on Floating Point
- Recap of floating point
- Preview assignment 8: Floating point

Lecture Recap

Basics

Numeral Systems

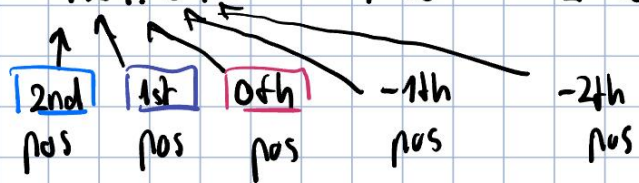
- **Should** be known be now, just a quick reminder
- **Very important** for fundamental understanding as well as for other courses (CN Ternary IP Addresses Exam Task)

Basics: Numeral systems (Decimal)

Systems@ETH zürich

Number system: Decimal

$$124.573 = 1 \cdot 10^{2} + 2 \cdot 10^{1} + 4 \cdot 10^{0} + 5 \cdot 10^{-1} + 7 \cdot 10^{-2} + 3 \cdot 10^{-3}$$



$$\Rightarrow d_2 \cdot 10^{2} + d_1 \cdot 10^{1} + d_0 \cdot 10^{0} + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2} + d_{-3} \cdot 10^{-3}$$

$$= \sum_{i=-3}^{2} d_i \cdot 10^{i}$$

Generally: $N = \sum_{i=-n}^{-m} d_i \cdot 10^{i}$, $d_i \in \{0, \ldots, 9\}$ Base 10

Basics: Numeral systems (Binary)

Number system Binary

$$110.010 = 1\cdot 2^{2} + 1\cdot 2^{1} + 0\cdot 2^{0} + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3}$$

2nd pos | 1st pos | 0th pos | -1th pos | -2th pos

$$\Rightarrow b_2\cdot 2^{2} + b_1\cdot 2^{1} + b_0\cdot 2^{0} + b_{-1}\cdot 2^{-1} + b_{-2}\cdot 2^{-2} + b_{-3}\cdot 2^{-3}$$

$$= \sum_{i=-3}^{2} b_i \cdot 2^{i}$$

Generally: Base 2

$$N = \sum_{i=-n}^{-m} b_i \cdot 2^{i}, \quad b_i \in \{0,1\}$$ Base 2

Basics: Numeral systems Generally

Zürich

Number Systems for ANY Base

Number system Base b: $\sum_{i=-m}^{n} v_i \cdot b^i$, $v_i \in \{0, \dots, b-1\}$

Example Base 10: $\sum_{i=-m}^{n} v_i \cdot 10^i$, $v_i \in \{0, \dots, 10-1\} = \{0, \dots, 9\}$

Base 2: $\sum_{i=-m}^{n} v_i \cdot 2^i$, $v_i \in \{0, \dots, 2-1\} = \{0, 1\}$

Base 16: $\sum_{i=-m}^{n} v_i \cdot 16^i$, $v_i \in \{0, \dots, "16-1"\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

Basics: Numeral systems Generally

Lamadrid Base Name System applied on this website

Bases 1-10		Bases 11-20					
1	unary	11	undecimal	21	unvigesimal	40	quadragesimal
2	binary	12	**duodecimal**	22	duovigesimal	50	quinquagesimal
3	ternary	13	tridecimal	23	trivigesimal	60	**sexagesimal**
4	quaternary	14	tetradecimal	24	tetravigesimal	70	septuagesimal
5	quinary	15	pentadecimal	25	pentavigesimal	80	octogesimal
6	senary	16	**hexadecimal**		…	90	nonogesimal
7	septenary	17	heptadecimal	30	trigesimal	100	centesimal
8	octal	18	octodecimal	32	duotrigesimal	120	centovigesimal
9	nonary	19	enneadecimal	36	hexatrigesimal	144	centotetraquadragesimal
10	**decimal**	20	**vigesimal**		…	360	trecentosexagesimal

- This concept applies to any base b

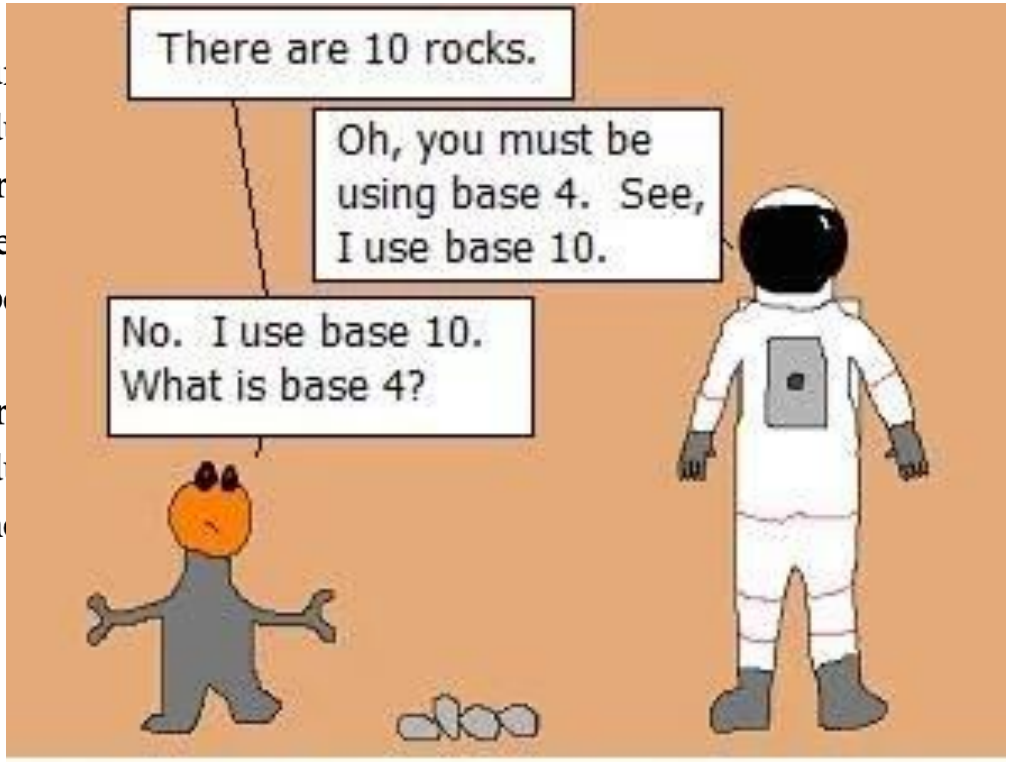
Basics: Numeral systems Generally

Lamadrid Base Name System applied on this website

Bases 1-10		Bases 11-20	
1	unary	11	undecimal
2	binary	12	**duodecimal**
3	ternary	13	tridecimal
4	quaternary	14	tetradecimal
5	quinary	15	pentadecimal
6	senary	16	**hexadecimal**
7	septenary	17	heptadecimal
8	octal	18	octodecimal
9	nonary	19	enneadecimal
10	**decimal**	20	**vigesimal**

21	u
22	d
23	tr
24	te
25	p
30	tr
32	d
36	h

- You can also understand bad jokes now

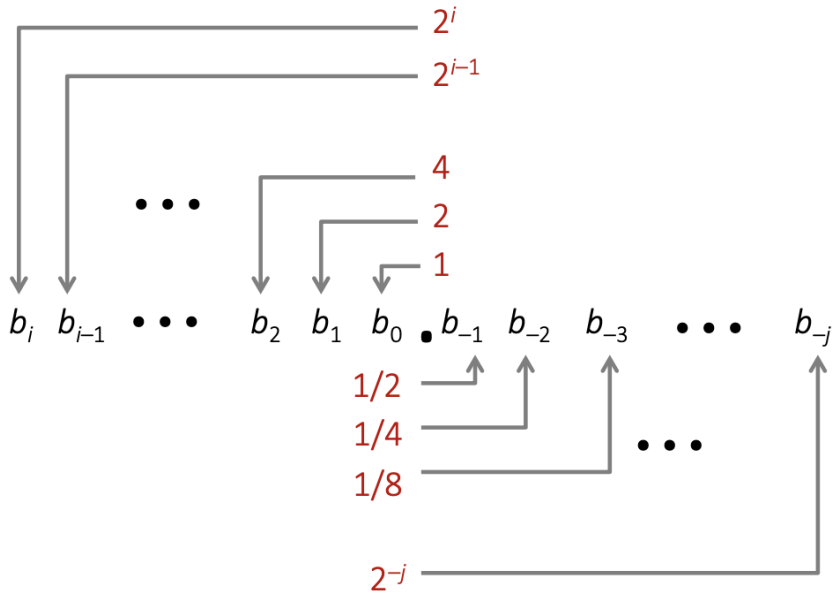


FP Recap

Fractional binary numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$



FP Recap

Floating point representation (recap from Digital Circuits?)

- Numerical form:

$$(-1)^s . M . 2^E$$

  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two
- Encoding
  - MSB (Most Significant Bit) s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

- **S,M,E** refer to numerical form
- **s, exp, frac** refer to binary encoding

FP Recap

Original precisions

IEEE 754 Single Precision (32 bits):

IEEE 754 Double Precision (64 bits):

FP Recap

Precision	Significand bits	Exponent bits	Total
Half	11	5	16
Single	24	8	32
Double	53	11	64
Quadruple	113	15	128
Octuple	237	19	256
Google bfloat16	*7*	*8*	*16*
Nvidia TensorFloat	*10*	*8*	*19*
AMD fp24	*17*	*7*	*24*

Lecture Recap

Floating Points in C

FP in C

- C99 guarantees two levels:
  - `float` - single precision
  - `double` - double precision
- `long double` is *usually* quadruple precision

- Conversions/casting
  - Casting between `int`, `float`, and `double` changes bit representation
  - `double/float` → `int`
    - Truncates fractional part (like rounding toward zero)
    - Not defined when out of range or NaN: Generally sets to `TMin`
  - `int` → `double`
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - `int` → `float`
    - Will round according to rounding mode

Rounding of FP in C

From/To	Int	Float	Double
Int	-	No loss for small values; precision loss for large values	No loss
Float	Truncates fractional part; overflow for large values	-	No loss
Double	Truncates fractional part; overflow for large values	Precision loss for large/precise values	-

Lecture Recap

FP: Normalised, Denormalised, Special Values

FP Recap

- There are **3 types** of encodings

- **1. Normalised**: values in the normal range
- **2. Denormalised**: values which are very small
- **3. Special Numbers**: Infinity, 0, NaN

FP Recap

- There are **3 types** of encodings

- **1. Normalised**: values in the normal range
- **2. Denormalised**: values which are very small
- **3. Special Numbers**: Infinity, 0, NaN

FP Recap: 1. Normalised

- Condition: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as biased value: $E = Exp - Bias$
  - *Exp*: unsigned value exp
  - *Bias* = $2^{e-1} - 1$, where e is number of exponent bits
    - Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)

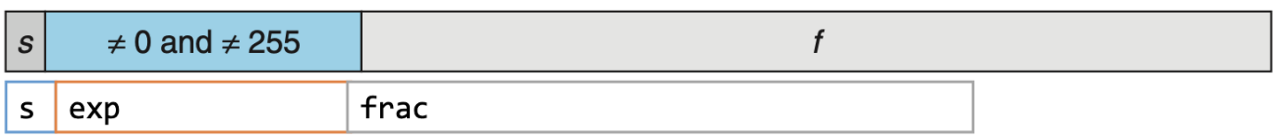
- Significand coded with implied leading 1: $M = 1.xxx…x_2$
  - xxx…x: bits of frac
  - Minimum when 000…0 ($M = 1.0$)
  - Maximum when 111…1 ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for "free"

- **S,M,E** refer to numerical form
- **s, exp, frac** refer to binary encoding

Numerical form:

$$(-1)^s . M . 2^E$$

1. Normalized



Normalized encoding example

- Value: `float F = 15213.0;`
  15213_{10} $= 11101101101101_2$
  $= 1.1101101101101_2 \times 2^{13}$

- Significand
  - *M* = $1.\underline{1101101101101}_2$
  - frac = $\underline{1101101101101}0000000000_2$

- Exponent
  - *E* = 13
  - *Bias* = 127
  - *Exp* = $140 = 10001100_2$

- Result:

s	exp	frac
0	10001100	11011011011010000000000

- Bias=(2^e)-1, for e=8: 127
- E=Exp-Bias ⇔ Exp=E+Bias
  ⇔ Exp=13+127
  =140

FP Recap

- There are **3 types** of encodings

- **1. Normalised**: values in the normal range
- **2. Denormalised**: values which are very small
- **3. Special Numbers**: Infinity, 0, NaN

FP Recap: 2. Denormalised

- Condition: `exp = 000…0`

- Exponent value: $E = -Bias + 1$ (instead of E = 0 – Bias)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - xxx…x: bits of `frac`
- Cases
  - `exp = 000…0, frac = 000…0`
    - Represents value 0
    - Note distinct values: +0 and –0 (why?)
  - `exp = 000…0, frac ≠ 000…0`
    - Numbers very close to 0.0
    - Lose precision as get smaller

2. Denormalized

FP Recap

- There are **3 types** of encodings

- **1. Normalised**: values in the normal range
- **2. Denormalised**: values which are very small
- **3. Special Numbers**: Infinity, 0, NaN

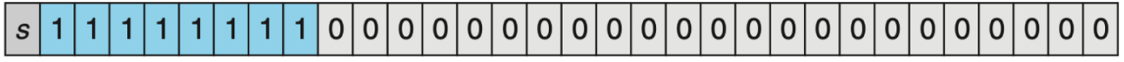
FP Recap: 3. Special Values

- Condition: `exp = 111…1`

- Case: `exp = 111…1, frac = 000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g. 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -∞

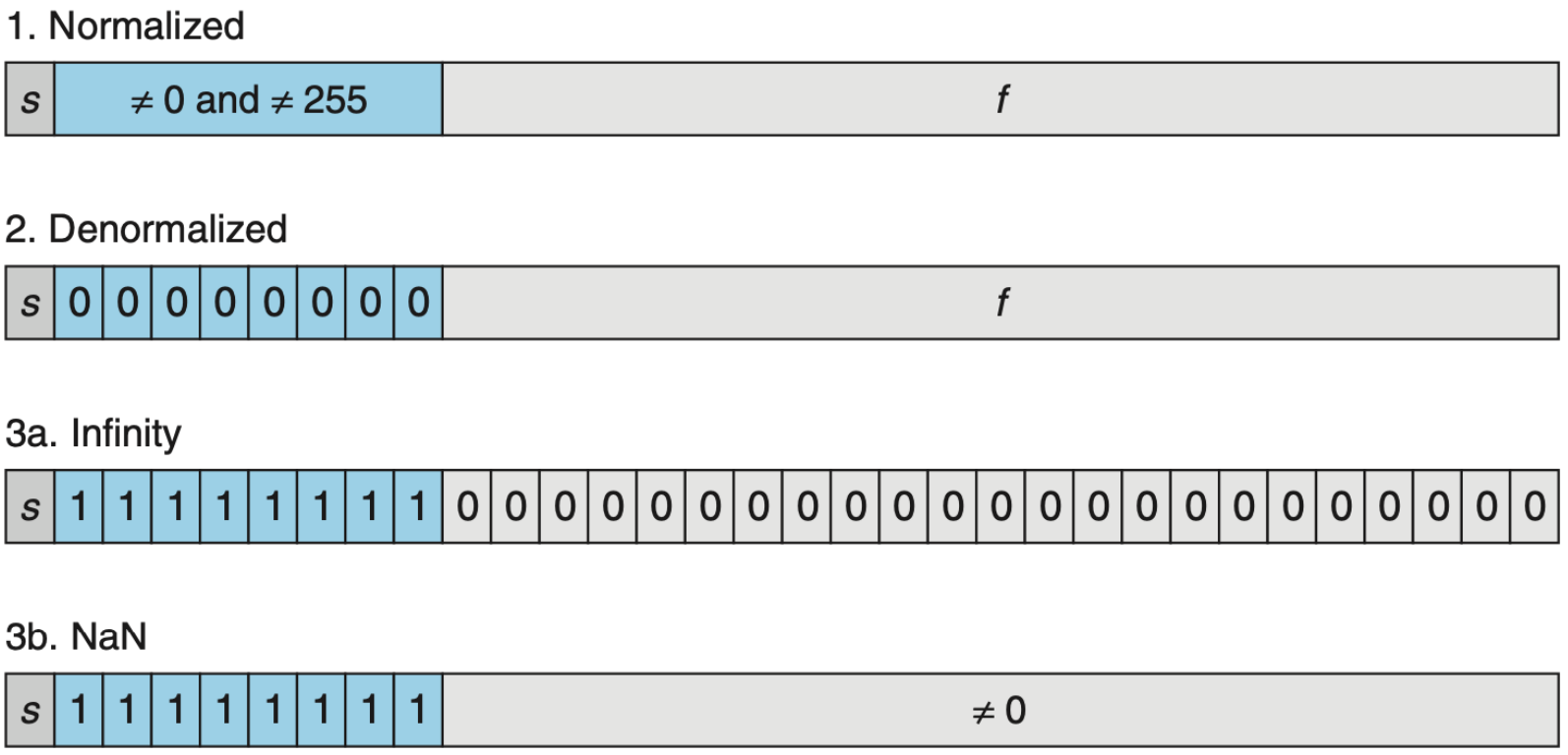
- Case: `exp = 111…1, frac ≠ 000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ - ∞, ∞ * 0

3a. Infinity



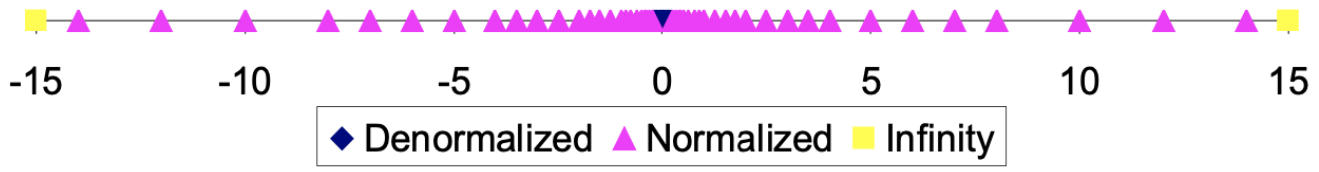
3b. NaN

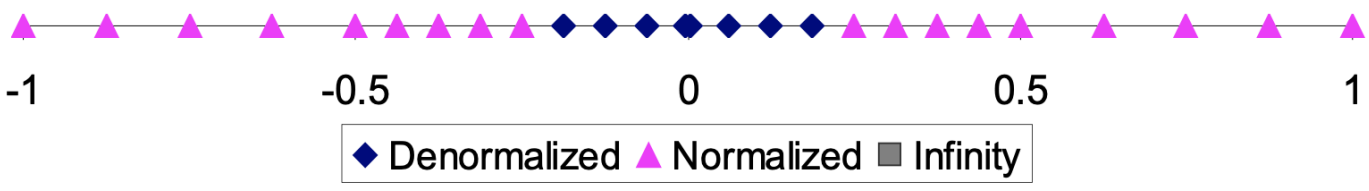
FP Recap: Normalised, Denormalised, Special Values **Graphical**



FP Recap

- *Really* tiny 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$





Lecture Recap

FP: Rounding Modes

FP Recap: Rounding Modes

Rounding

- Rounding modes (illustrate with CHF rounding)

	CHF 1.40	CHF 1.60	CHF 1.50	CHF 2.50	CHF -1.50
Towards zero	1	1	1	2	-1
Round down (-∞)	1	1	1	2	-2
Round up (+∞)	2	2	2	3	-1
Nearest Even *(default)*	1	2	2	2	-2

- 4 Rounding Modes IEEE 754 supports

FP Recap: Round to even Decimal

Closer look at “round-to-even”

- Default rounding mode for IEEE FP
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under- estimated
- Applying to other decimal places / bit positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

Value	Result	Description
1.2349999	1.23	(less than half way)
1.2350001	1.24	(greater than half way)
1.2350000	1.24	(half-way – round up)
1.2450000	1.24	(half way – round down)

FP Recap: Round to even Binary

Rounding binary numbers

- Binary fractional numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$

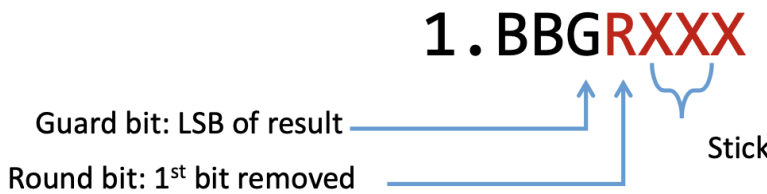
- Examples

  Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Result
$2\,{}^{3}/_{32}$	10.00011_2	10.00_2	< ½ : down	2
$2\,{}^{3}/_{16}$	10.00110_2	10.01_2	> ½ : up	$2\,{}^{1}/_{4}$
$2\,{}^{7}/_{8}$	10.11100_2	11.00_2	= ½ : up	3
$2\,{}^{5}/_{8}$	10.10100_2	10.10_2	= ½ : down	2 ½

- **Binary** equivalent of ".5" (decimal) is ".1000…" (binary)
- **Even in binary**: means 0 as lsb
- **Odd in binary**: means 1 as lsb

FP Recap: Rounding



Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Round up Conditions
Trivial Cases (we are not in the middle)

- **R=1,S=1** => Up (as Number is >0,5)
- **R=0, S=_** => Down (as Number is <0,5)

Round to even Cases (we are exactly in the middle)

- **G=1, R=1, S=0** => Up (to make it even, as G=1 means it not even)
- **G=0, R=1, S=0** => Down (to keep it even, as G=0 means its even)

FP Recap: Round to even Binary

Postnormalize

- Issue
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Lecture Recap

FP: Operations

FP Recap: Multiplication

Floating point multiplication

- Exact Result: $(-1)^s\ M\ \ 2^E$
  - Sign s: s_1 ^ s_2
  - Significand M: $M_1 * M_2$
  - Exponent E: $E_1 + E_2$

$$(-1)^{s1}\ M_1\ \ 2^{E1}\ \ \text{x}\ \ (-1)^{s2}\ M_2\ \ 2^{E2}$$

- Fixing
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision

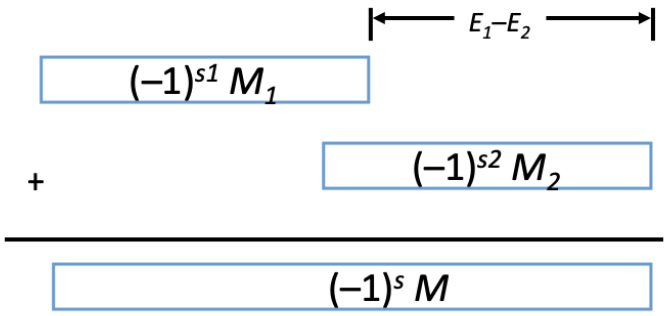
- Implementation
  - Biggest chore is multiplying significands

FP Recap: Addition

Floating point addition

$(-1)^{s1} M_1 \; 2^{E1} \; + \; (-1)^{s2} M_2 \; 2^{E2}$

Assume $E_1 > E_2$



- Exact result: $(-1)^s M \; 2^E$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E_1

- Fixing
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit **`frac`** precision

FP Operations Remark

- The **float lab** is about implementing those

- **Highly recommend doing this lab**: stuff like this is **very likely to appear as a coding exercise in the exam**

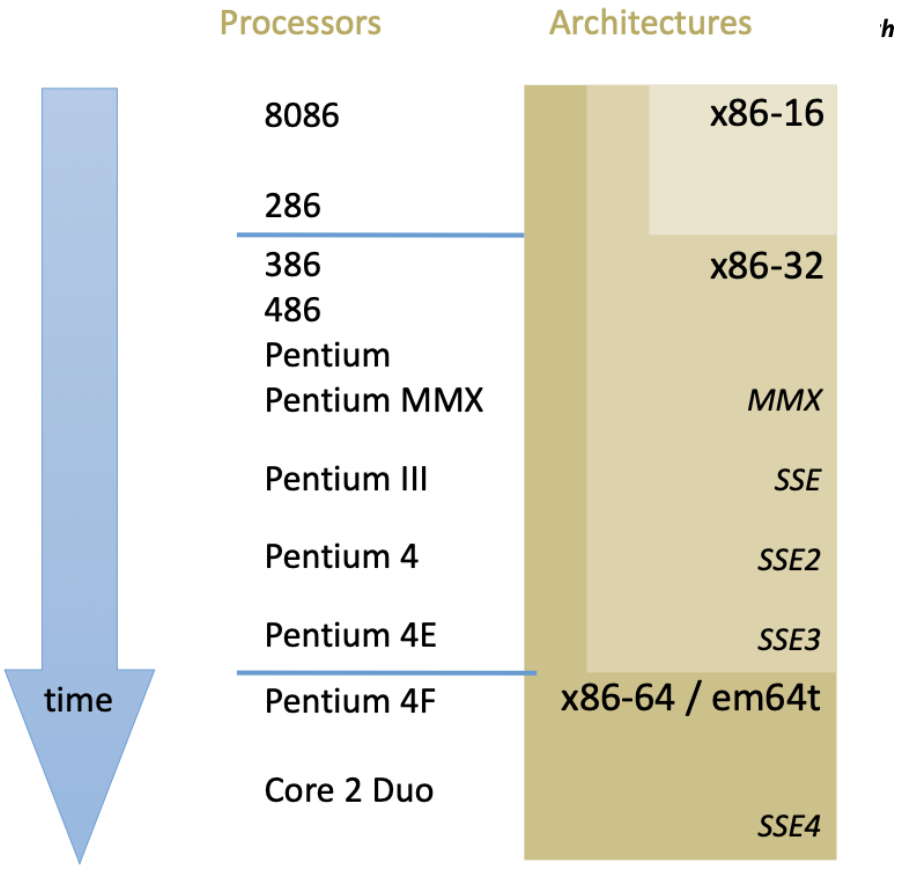
Lecture Recap

SSE3 – Streaming SIMD Extension 3

SSE3 Introduction

- **SSE3** (Streaming SIMD Extensions 3): Part of a family of instruction set **extensions** of x86, to accelerate computation

- **SSE** introduced %xmm registers
- **SSE2** extended %xmm to support both single and double prec.
- **SSE3** added more specialced SIMD instructions
- **AVX** introduced wider %ymm regs (256 bits)
- **AVX-512** extended to 512-bit %zmm regs



SSE3 Registers

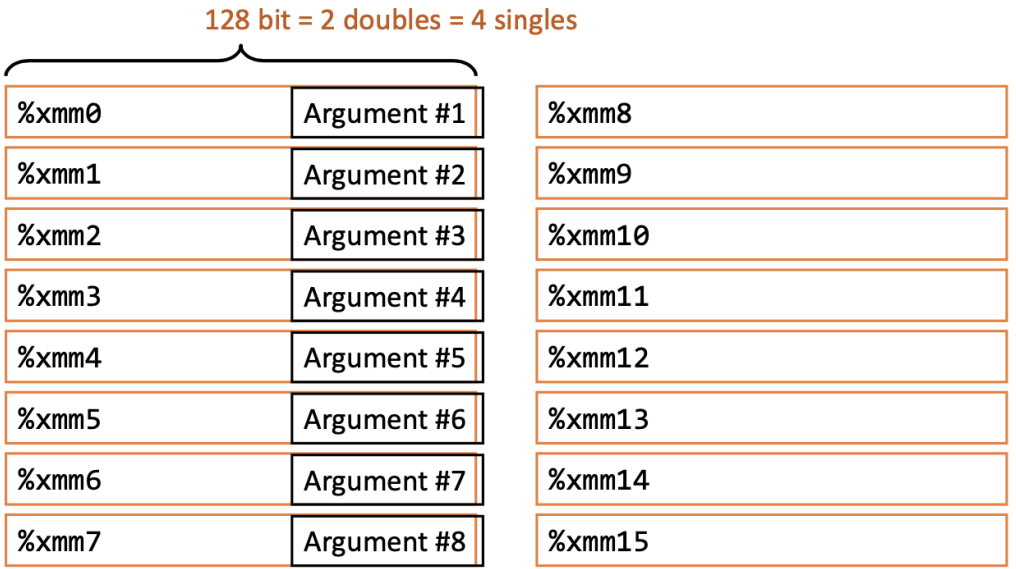
- **General purposes registers**: %rax, %rdi, etc.
- **Purpose**: Integer operations, memory addresses and control flow: they handle arithmetic computations, pointer manipulation passing arguments etc.

- **SIMD Registers**: %xmm0, %xmm1, etc.
- **Purpose**: Specialised for floating point computations and SIMD operations
- **Why separate registers**? We have separate hardware for floating point calculations (Floating point Unit FPU)

SSE3 Registers

SSE3 registers

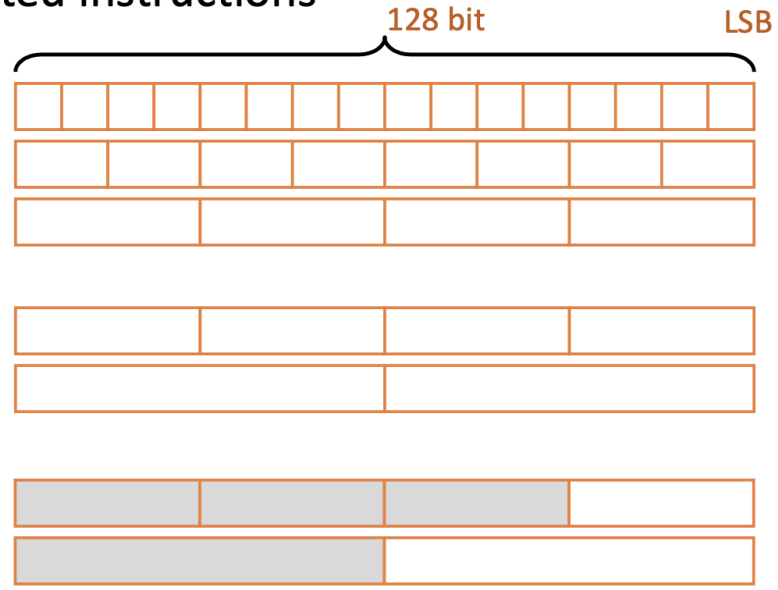
- All caller saved
- %xmm0 for floating point return value



SSE3 Registers

SSE3 registers

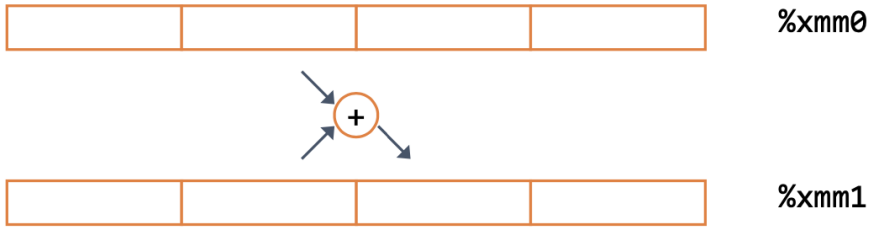
- Different data types and associated instructions
- Integer vectors:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
- Floating point vectors:
  - 4-way single
  - 2-way double
- Floating point scalars:
  - single
  - double



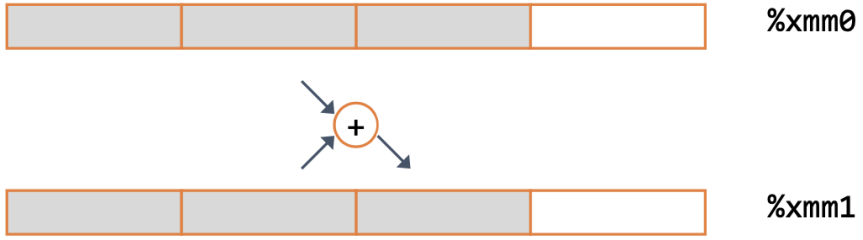
FP Registers

SSE3 instructions: examples

- Single precision 4-way vector add: `addps %xmm0 %xmm1`

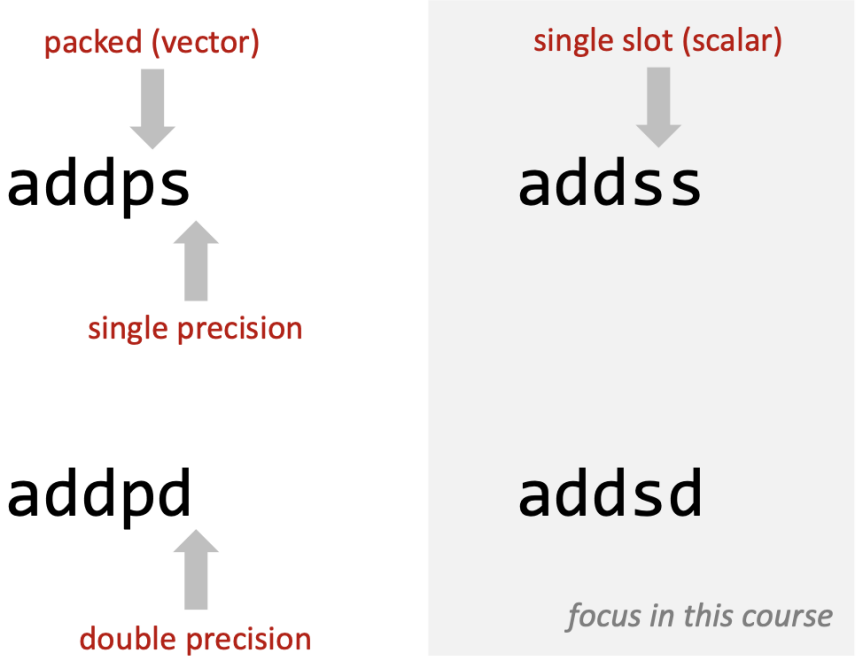


- Single precision scalar add: `addss %xmm0 %xmm1`



SSE3 Registers

SSE3 instruction names



SSE3 Registers

SSE3 basic instructions

- Moves:

Single	*Double*	*Effect*
`movss`	`movsd`	D ← S

  - Usual operand form: reg → reg, reg → mem, mem → reg

- Arithmetic:

Single	*Double*	*Effect*
`addss`	`addsd`	D ← D + S
`subss`	`subsd`	D ← D – S
`mulss`	`mulsd`	D ← D x S
`divss`	`divsd`	D ← D / S
`maxss`	`maxsd`	D ← max(D,S)
`minss`	`minsd`	D ← min(D,S)
`sqrtss`	`sqrtsd`	D ← sqrt(S)

SSE3 Registers

x86-64 FP code example

- Compute inner product of two vectors
  - Single precision arithmetic
  - Uses SSE3 instructions

```
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;

    for (i = 0; i < n; i++) {
        result += x[i]*y[i];
    }
    return result;
}
```

```
ipf:
   xorps   %xmm1, %xmm1          # result = 0.0
   xorl    %ecx, %ecx            # i = 0
   jmp     .L8                   # goto middle
.L10:                            # loop:
   movslq  %ecx,%rax             # icpy = i
   incl    %ecx                  # i++
   movss (%rsi,%rax,4), %xmm0    # t =  y[icpy]
   mulss (%rdi,%rax,4), %xmm0    # t *= x[icpy]
   addss   %xmm0, %xmm1          # result += t
.L8:                             # middle:
   cmpl    %edx, %ecx            # i:n
   jl      .L10                  # if < goto loop
   movaps  %xmm1, %xmm0          # return result
   ret
```

- **Recall**: SSE3 is simply an **extension** of x86: so it look like x86 you have seen until now with additional instructions

Floating Point Exam Questions

HS15 Q6

Remark FP Exam Questions

- In (almost) every exam there are **floating point questions** like the following
- So its **worth it** to be able to solve them quickly & correctly as they are not hard

- They may come in the form of programming exercises in your case (**do the fp lab!**)

Recall General Format of s,exp,frac

Floating point representation (recap from Digital Circuits?)

- Numerical form:

$$(-1)^s . M . 2^E$$

  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two
- Encoding
  - MSB (Most Significant Bit) s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

HS15 Question 6

Systems@ETH Zürich

Question 6

[16 points]

Consider the following 8-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is $2^{3-1} - 1 = 3$.
- The last 4 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E is the exponent.

The rules are like those in the IEEE standard (i.e. normalized and denormalized numbers, and the same representation of 0, infinity, and NAN).

Fill in the table below for this format. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{3}{4}$.
- **E:** The integer value of the exponent.
- **Value:** The numeric value represented by the number.

  Note: you need not fill in entries marked with "—".

HS15 Question 6

Description	Binary	M	E	Value
Minus zero				-0.0
—	01000101			
Smallest denormalized				
Largest normalized				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Remark FP Exam Questions

- Ill solve some as an example: then you can solve some on your own and we can compare

- **FP is not hard**: once you converted 2-3 fps you grasped the concept and thus are easy points in the exam

HS

ms@ETH Zürich

Description	Binary	M	E	Value
Minus zero	1 000 000	0	−2	−0.0
—	01000101			
Smallest denormalized				
Largest normalized				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Minus zero (3 bits exp, 4 bits frac / bias = $2^{3-1} - 1 = 3$)

s = 1 (negative)

exp = 0 0 0

frac = 0 0 0 0

s exp frac

binary 1 | 0 0 0 | 0 0 0 0

$M = 0.0000 = 0$

$E = -\text{bias} + 1 = -3 + 1 = -2$

Value: $(-1)^s \cdot 0.0000 \cdot 2^{-2} = -0.0$

implicit leading 0 (denorm)

Description	Binary	M	E	Value
Minus zero	1 000 000	0	−2	−0.0
—	01000101	21/16	1	2.625
Smallest denormalized				
Largest normalized				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Value 0|1 00|0 1 0 1

S = 0

implicit leading 1: all its normalized

M = 1.0 1 0 1 $= 1 + \frac{1}{4} + \frac{1}{16} = \frac{16+4+1}{16} = \frac{21}{16}$ M

exp = $(1\ 0\ 0)_2 = (4)_{10}$, $E = \text{exp} - \text{Bias} = 4 - 3 = 1$ M

Value: $(-1)^S \cdot M \cdot 2^E = (-1)^0 \cdot \frac{21}{16} \cdot 2^1 = 2.625$ M

Description	Binary	M	E	Value
Minus zero	1 000 000	0	−2	−0.0
—	01000101	21/16	1	2.625
Smallest denormalized	0 000 0001	1/16	−2	1/64
Largest normalized				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Smallest (pos.) denorm (3 bits exp, 4 bits frac / bias $= 2^{3-1} - 1 = 3$)

s = 0 (positive)

exp = 0 0 0 (denorm)

frac = 0 0 0 1

s exp frac

binary 0 | 0 0 0 | 0 0 0 1

1/2 1/4 1/8 1/16

$M = 0.0001 = 1/16$

$E = -\text{bias}+1 = -3+1 = -2$

Value: $(-1)^0 \cdot 0.0001 \cdot 2^{-2} = 2^{-4} \cdot 2^{-2} = 2^{-6} = 1/64$

implicit leading 0 (denorm)

Try the next one on your own!

Systems@ETH zürich

Description	Binary	M	E	Value
Minus zero	1 000 000	0	−2	−0.0
—	01000101	21/16	1	2.625
Smallest denormalized	0 000 0001	1/16	−2	1/64
Largest normalized				
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Description	Binary	M	E	Value
Minus zero	1 000 000	0	−2	−0.0
—	01000101	21/16	1	2.625
Smallest denormalized	0 000 0001	1/16	−2	1/64
Largest normalized	0 110 1111	31/16	3	15.5
One				1.0
—				5.5
Positive infinity		—	—	$+\infty$

Largest normalised

(3 bits exp, 4 bits frac / bias $= 2^{3-1} - 1 = 3$)

s = 0 (positive)

exp = 1 1 0 (denorm)

frac = 1 1 1 1

why not 1 1 1?

binary: s | exp | frac = 0 | 1 1 0 | 1 1 1 1

$M = 1.1111 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{16+8+4+2+1}{16} = \frac{31}{16}$

$E = exp - bias$, $exp = 6 \Rightarrow E = 6 - 3 = 3$

Value: $(-1)^0 \cdot 1.1111 \cdot 2^3 = \frac{31}{16} \cdot 2^3 = \frac{31}{2} = 15.5$

careful: normalised bias

implicit leading 1 normalised

Remark FP Exam Questions

- The rest work analogous, i.e. once you know the formulas and how to calculate its **trivial** (i.e. easy points in the exam)

Floating Point Exam Questions

HS19 Q11

HS19 Question 11

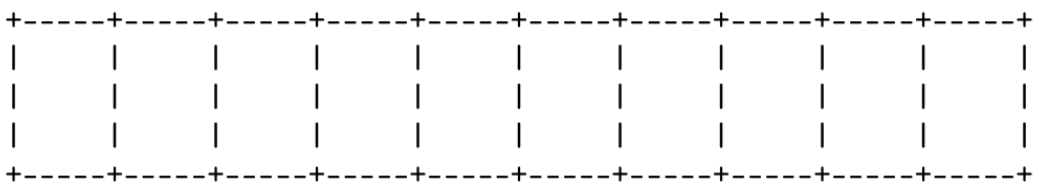
Systems@ETH Zürich

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part, and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



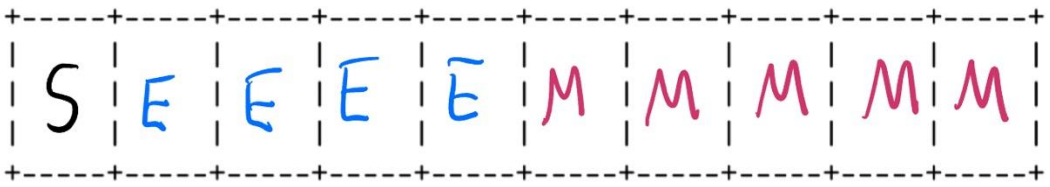
HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



- You need to know the layout of Sign, Exponent, Mantissa
- Also that there is always **one sign bit**

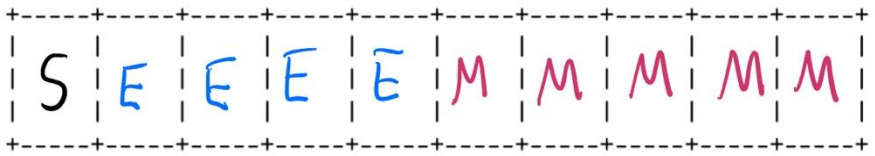
HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



The **bias** for this format is 7. Explain why.

(2 points)

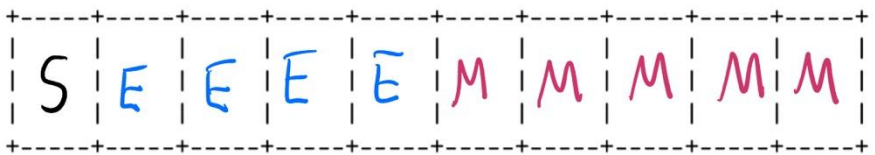
HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



The **bias** for this format is 7. Explain why.

(2 points)

e = # bits for exp = 4

$$\text{Bias} = 2^{e-1} - 1 = 2^{4-1} - 1 = 8 - 1 = 7$$

HS19 Question 11

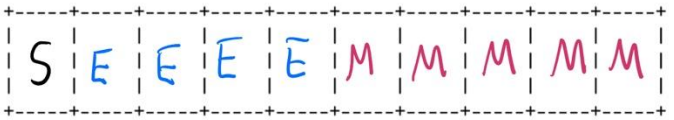
The **bias** for this format is 7. Explain why.

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



How is the real number 1/2 represented in binary in this system? Show your working.

HS19 Question 11

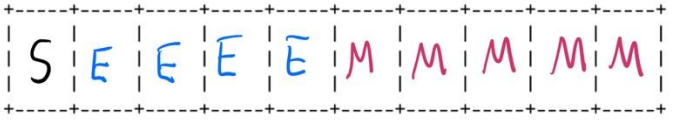
The **bias** for this format is 7. Explain why.

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



How is the real number 1/2 represented in binary in this system? Show your working.

start: $\frac{1}{2}$ = 0.1 in Binary

Translate to leading 1: ⇒ 1.0 for E = −1

$\bar{E} = exp - Bias$

$\Leftrightarrow exp = E + Bias = -1 + 7 = 6$

Exp (4 bits): 1 0 1 1 0

Sign: : 0 (as it's positive)

Fraction (5): 0 0 0 0 0

Total: 0 | 0 1 1 0 | 0 0 0 0 0

HS19 Question 11

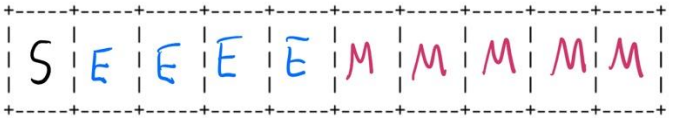
The **bias** for this format is 7. Explain why.

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



What real number is represented by the binary value 1111000000 in this system? Show your working

(4 points)

HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)

S	E	E	E	E	M	M	M	M	M

The **bias** for this format is 7. Explain why.

What real number is represented by the binary value 1111000000 in this system? Show your working

Given: $1|1\,1\,1\,1\,0|0\,0\,0\,0\,0$

Sign: 1 ⇒ negative

exp: $(1\,1\,1\,0)_2 = (14)_{10}$ ⇒ $E = exp - Bias$ ⇔ $E = 14 - 7 = 7$

mantissa: $(0\,0\,0\,0\,0)_2$

Numerical Form: $(-1)^S \cdot M \cdot 2^E$

⇒ $(-1)^1 \cdot 1.0\,0\,0\,0\,0 \cdot 2^7 = -2^7 = -128$

↑ implicit leading 1 (as exp ≠ 11..1, exp ≠ 00..0 ⇒ normalized)

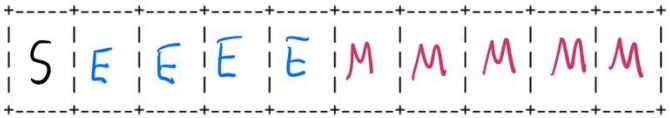
HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)



The **bias** for this format is 7. Explain why.

What number in this system is represented by the smallest negative denormalized value?

Give your answer as a decimal number, and show your working.

(4 points)

HS19 Question 11

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)

S	E	E	E	E	M	M	M	M	M

The **bias** for this format is 7. Explain why.

What number in this system is represented by the smallest negative denormalized value?

Give your answer as a decimal number, and show your working.

Want: smallest, neg. denorm., neg ⇒ $s = 1$, denorm ⇒ exp = 0 0 0 0

"smallest neg" if frac = 1 1 1 1 1

Value: s | exp | frac

1 | 0 0 0 0 | 1 1 1 1 1

E (denorm): $-\text{Bias} + 1 = -7 + 1 = -6$

$$(-1)^1 \cdot 0.11111 \cdot 2^{-6} = -1 \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32}\right) \cdot 2^{-6} = -0.96875 \cdot 2^{-6}$$

↑ implicit leading 0 as its denorm

Floating Point Exam Questions

HS17 Question 3

HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What is the *bias* for this format?

(2 points)

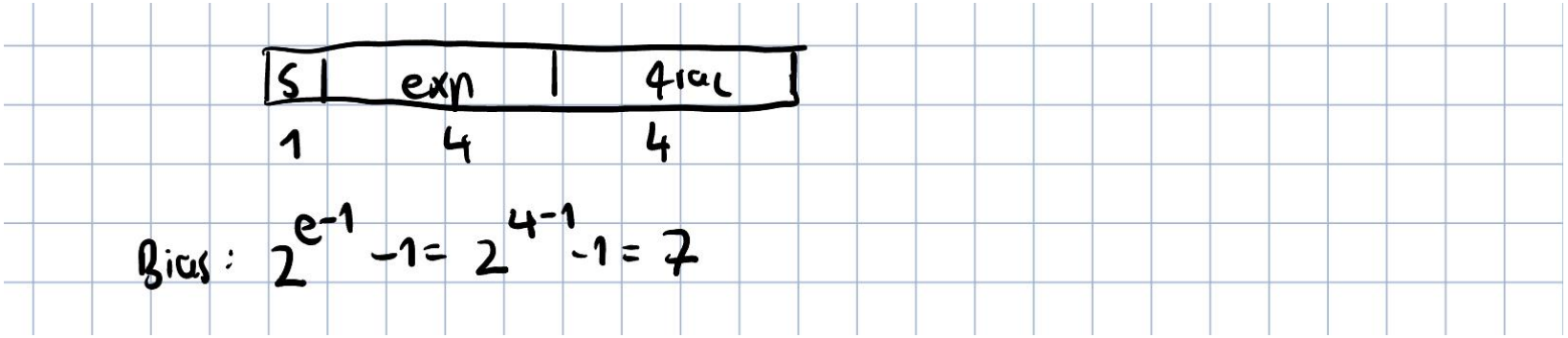
HS17 Question 3

Systems@ETH Zürich

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What is the *bias* for this format?

(2 points)



HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What integer is the largest positive normalized value below infinity?

Give your answer as a decimal number, and show your working.

(4 points)

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What integer is the largest positive normalized value below infinity?

Give your answer as a decimal number, and show your working.

Systems@ETH zürich

Searching: largest normalised value (Bias = 7)

s = 0

Binary: 0 | 1 1 1 0 | 1 1 1 1

exp = 1 1 1 0

E = exp - Bias, exp = 13 ⇒ E = 14 - 7 = 7

frac = 1 1 1 1

Interval: $(-1)^0 \cdot 1.1111 \cdot 2^7 = (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32}) \cdot 2^7 =$ hard ???

simply shift instead of calculating ints directly

128 64 32 16 8

$= (1\ 1\ 1\ 1\ 1\ 0\ 0\ 0)_2$

$= (248)_{10}$

HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What real number is represented by the binary value 100000000 in this system? Show your working

(2 points)

HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

What real number is represented by the binary value 100000000 in this system? Show your working

(2 points)

- Do not calculate anything!
- By the bit representation we know it must be **-0**

HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

[continued]

How is the real number 1 represented in binary in this system? Show your working.

(4 points)

HS17 Question 3

Systems@ETH Zürich

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

[continued]

How is the real number 1 represented in binary in this system? Show your working.

(4 points)

- **Sign=0** (obvious)
- Normalised means implied leading 1 => to get 1.0 we need **frac=0**
- **E=**exp-bias (normalized), we want E to be 0, so this implies exp=bias => exp=7, i.e. **exp=0111**

$$: 0\ 0111\ 0000_2.$$

HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

[continued]

How is the real number -1/32 represented in binary in this system? Show your working.

(4 points)

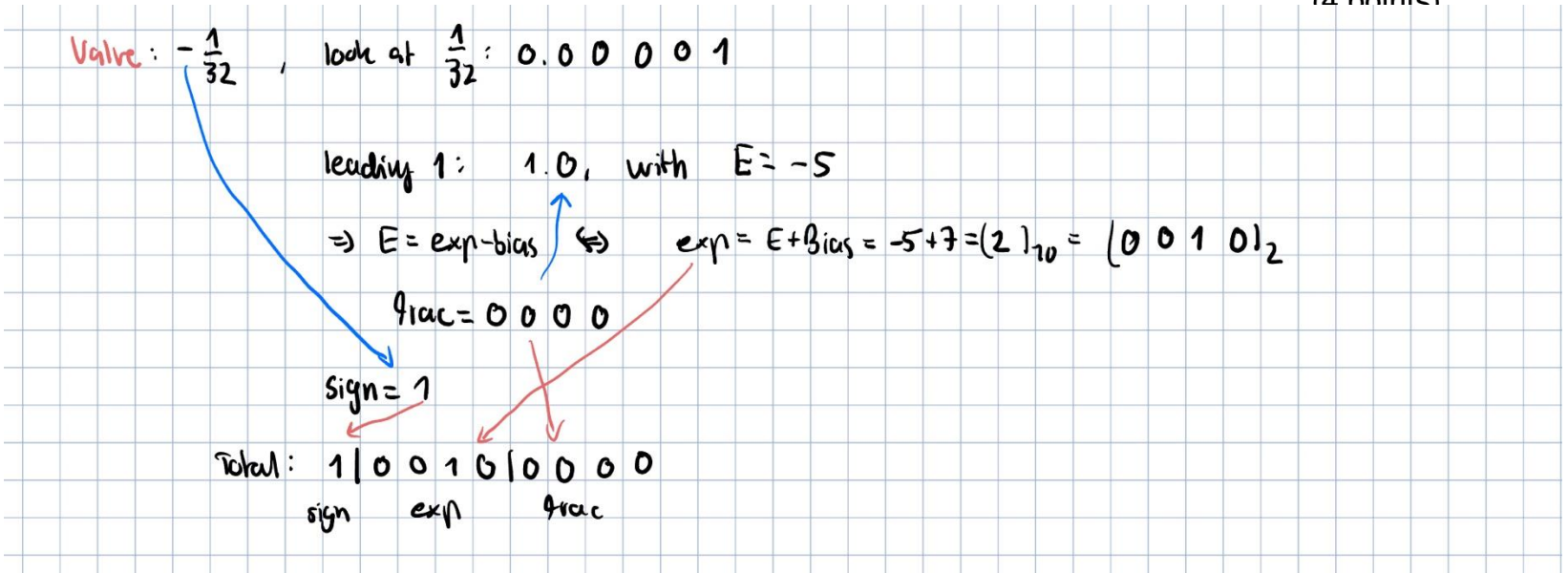
HS17 Question 3

Consider a floating point format which uses 9 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 4 bits to represent the exponent.

[continued]

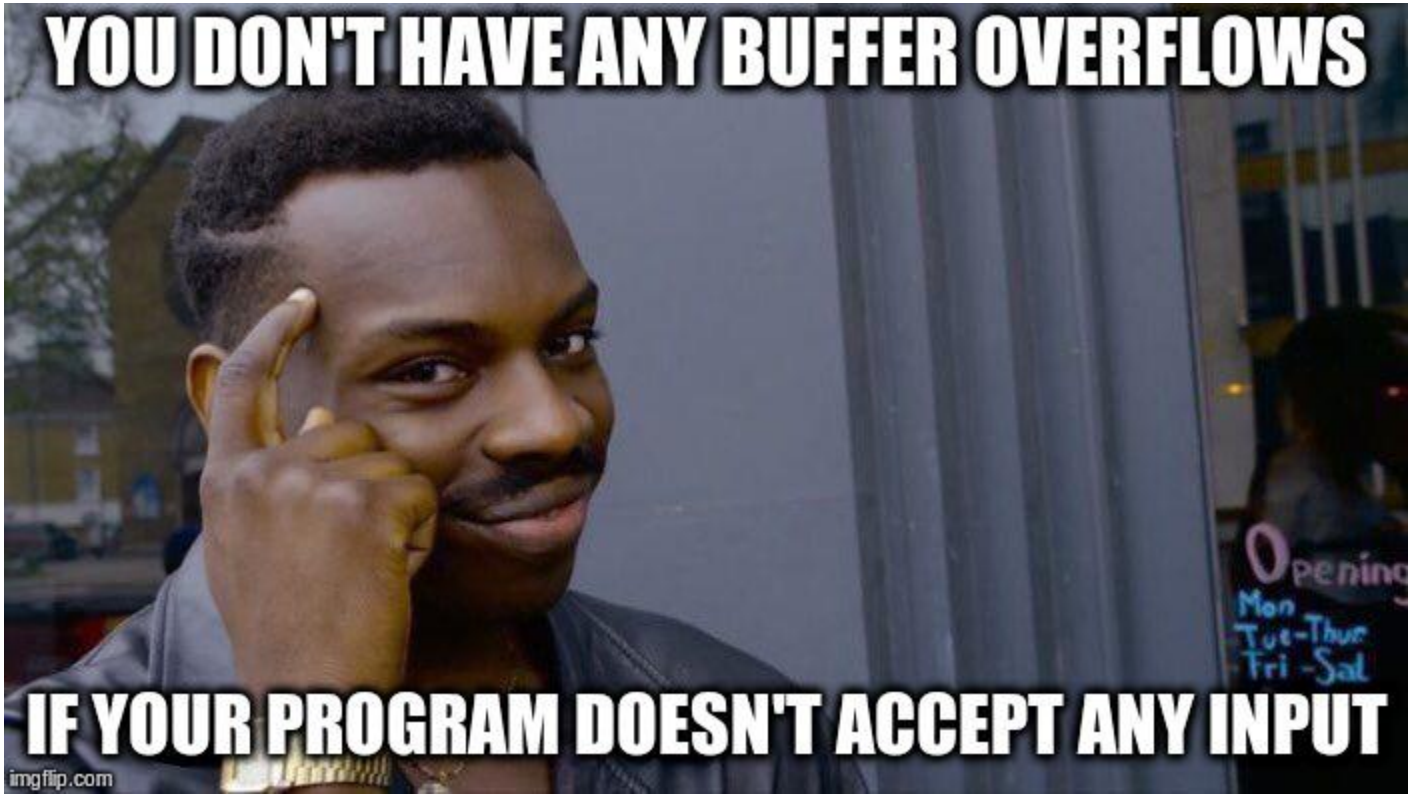
How is the real number -1/32 represented in binary in this system? Show your working.

(4 points)



Q&A

Questions about the Attacklab ?



Floating point

Recap for the Assignment

Floating Point Representation

Systems@ETH Zürich

Numerical Form
"Scientific Notation"

Sign S ∈ {0,1}
Exponent E
Mantissa M ∈ [1.0, 2.0)

$F = (-1)^S \cdot M \cdot 2^E$

Encoding
"Bit Pattern"

Sign S ∈ {0,1}
exp ~E
frac ~M

1	8	23	*float*
1	11	52	*double*

Casting

Integer Types

- What happens here?

```
1. unsigned int foo;
2. long bar = (long) foo;
```

OK.
int must fit in a long

Floats

What happens here?

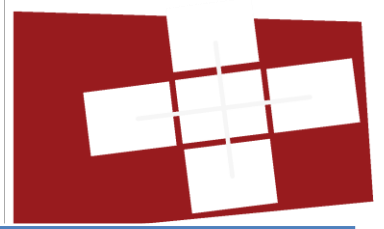
```
1. int i;
2. long long l;
3. float f;
4. double d;

1. i = (int) f;
2. i = (int) d;
3. f = (float) d;
4. d = (double) i;
5. d = (double) f;
```

Floats <-> Integers

- Casting between floats, doubles and integers generally changes the bit representation!

```
1.  int i = 0xABCDABCD;
2.  float f = (float) i;
3.
4.  int *i2 = (int *) &f;
5.
6.  printf(“%x, %x”, i, *i2);
7.
8.  // Prints abcdabcd, cea864a8
```

Floats <-> Integers



From	To	Description

double/float
 	
`float f1 = 1.12345;`
`float f2 = 1.999999;`
	int
	
	`(int) f1 == ?`
	`(int) f2 == ?`
		Truncate the fractional part.
		Out of range, NaN , inf
		-> Default handler assigns ***INT_MIN***.

int
 	
`long long l1=0x7FFFFFFFFFFFFFFF;`
`long long l2=0xFFFFFFFF;`
	double
	
	`double d1 = (double) l1;`
	`double d2 = (double) l2;`
		exact conversion if integral value is max 54 bits
		`l1 == (long long) d1;`
		`l2 == (long long) d2;`

int	float	round using rounding mode

Floats <-> Integers


```
float f2 = 1.50f;
float f3 = 1.50f;

printf("%f", f2+f3);
printf("%i", (int)(f2+f3));
printf("%i", (int)f2 + (int)f3);
```

```
f2 + f3 == 3.00000
(int)(f2 + f3) == 3
(int)f2 + (int)f3 == 2
```

Casting

- What happens here?

```
1. int i;
2. long long l;
3. float f;
4. double d;

1. i = (int) f;
2. i = (int) d;
3. f = (float) d;
4. d = (double) i;
5. d = (double) f;
```

(1) Truncate fractional part, assign TMin/TMax if integral part under-/overflows `int`

(2) same as (1)

(3) Loss of precision and `float` is set to +/-inf if exponent of `double` does not fit in exponent of float

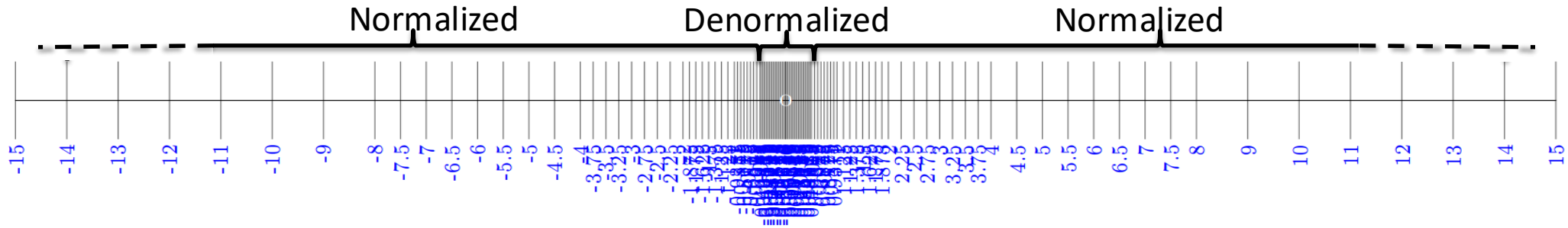
(4) No loss, every (32-bit) `int` can be represented exactly as `double`.

(5) No loss of precision. C standard guarantees that every `float` can be represented exactly as a `double`.
(only tricky for denormalized values)

$F=(-1)^S \cdot M \cdot 2^E$ S exp frac

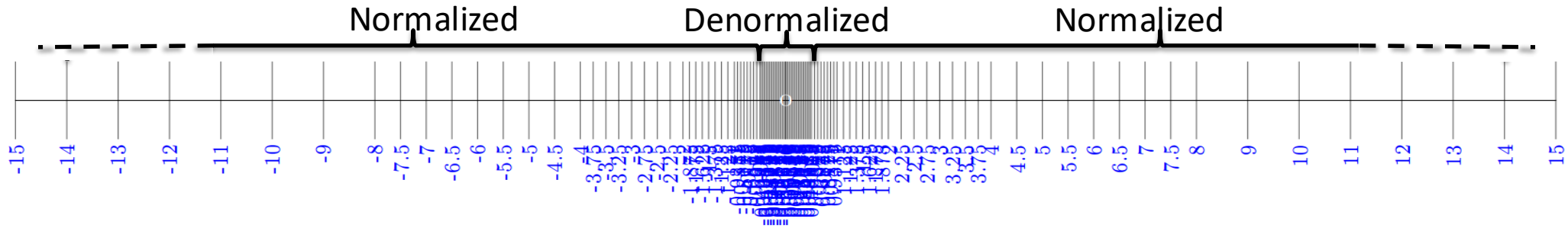
Normalized / Denormalized

- **Normalized: exp != {000…0, 111…1}**
  - Good for bigger values
  - Not equi-spaced
- **Denormalized number: exp == 000…0**
  - Good for very small values
  - Equi-spaced [-δ + eps, δ - eps], δ=smallest normalized number
  - zero



Normalized / Denormalized

- **Denormalized not a number: exp == 111…1**
  - +/- infinity if frac = 000…0
  - NaN else



NORMALIZED

Exponent

- Must be able to have negative exponents -> encode as *biased value*
  - exp = E + bias

We will see in a few slides why we subtract 1 here

- bias = $2^{e-1} - 1$:
  - For Single precision?
  - For Double precision?

- Normalized: exponent never all zeros nor all ones

$F=(-1)^S \cdot M \cdot 2^E$ S exp frac

NORMALIZED

Significand

- In binary: M ∈ [1.0, 2.0)
- The encoding always assumes a leading 1
- Remove it to save one bit!
- What are the max and min values for the significand?

DENORMALIZED

Exponent

- Express values very close to 0:
  exponent must be as negative as possible

- Exp is all zero and the exponent is evaluated as
  - exp = -bias + 1

DENORMALIZED

Significand

- We are close to zero: $M \in [0.0, 1.0)$
- The encoding always assumes a leading 0
- Remove it to save one bit!
  (like with the 1 in normalized case)

$F=(-1)^{S}\cdot M\cdot 2^{E}$ S exp frac

Special Values

Sign	Fraction	Exponent	Description
	000…0	111…1	Infinity (+ / -) operation overflows
	!= 000…0	111…1	Not-a-Number (NaN) No numeric value sqrt(-1)
0	000…0	000…0	Zero (like integer 0)
1	000…0	000…0	Minus 0

-0 ?

In IEEE arithmetic, it is natural to define log 0 = -∞ and log x to be a ***NaN when x < 0****. Suppose that x represents a small negative number that has underflowed to zero. Thanks to* ***signed zero, x will be negative, so log can return a NaN****. However, if there were no signed zero, the log function could not distinguish an underflowed negative number from 0 and would therefore have to return -∞.*

Tiny floating point example

The exam usually has a question on tiny floats

- 8-bit floating point representation
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of $2^{4-1} = 7$.
  - the last three bits are the `frac`

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

$F=(-1)^{S}\cdot M\cdot 2^{E}$ S exp frac

	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

Creating a floating point number

- Steps
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

- Case study
  - Convert 8-bit unsigned numbers to tiny floating point format

Value	Binary
128	10000000
13	00001101
17	00010001
19	00010011
138	10001010
63	00111111

Normalize

Requirement

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
  - Decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

$F=(-1)^S \cdot M \cdot 2^E$ S exp frac

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: bitwise OR of remaining bits

Rounding:

Round = 1, Sticky = 1

⇒ up if > 0.5

Round = 1, Sticky = 0, Guard = 1

⇒ round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

$F=(-1)^S \cdot M \cdot 2^E$ S exp frac

Postnormalize

Issue:
Rounding may have caused overflow
Shift right once & increment exponent

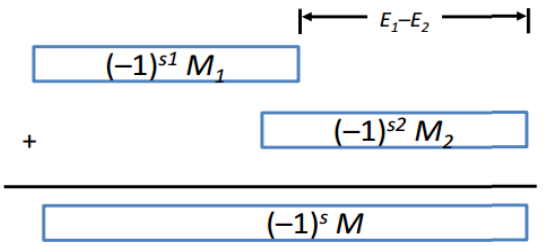
Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		144
63	10.000	5	1.000/e=6	64

A possible Exam Question

- You have an 8 bit floating point representation with 3 fraction bits.
  - Give the floating point representations of
    - 144
    - 64
  - Is the conversion exact?

Addition (normalized)

- Signed align and add (Assume E1 > E2)
  - Shift the first operand by the difference of their exponents
  - Add the M and s bits



- shift and adjust exponent until M ∈ [1.0, 2.0)
- Round
- Post-Normalize

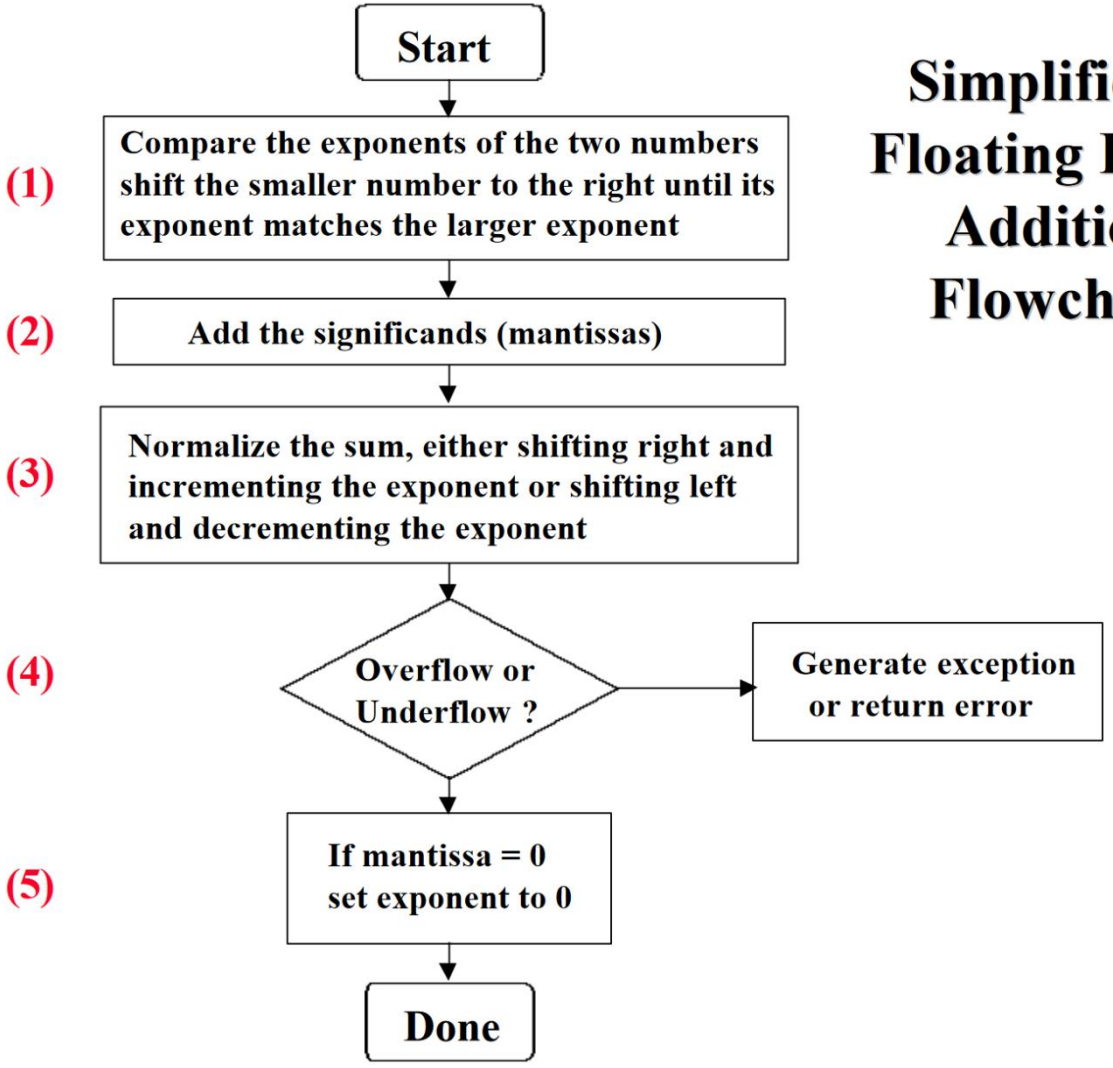
Multiplication (normalized)

- $F = (-1)^{S1 \otimes S2} * (M1 * M2) * 2^{E1+E2}$
  ($\otimes$ denotes XOR)
- $M = M1 * M2$
- $S = S1 \otimes S2$
- $E = E1 + E2$
- shift and adjust exponent until $M \in [1.0, 2.0)$

- Round M to fit fraction bits
- Post-normalize and check if exponent still in range

$F=(-1)^S \cdot M \cdot 2^E$ S exp frac

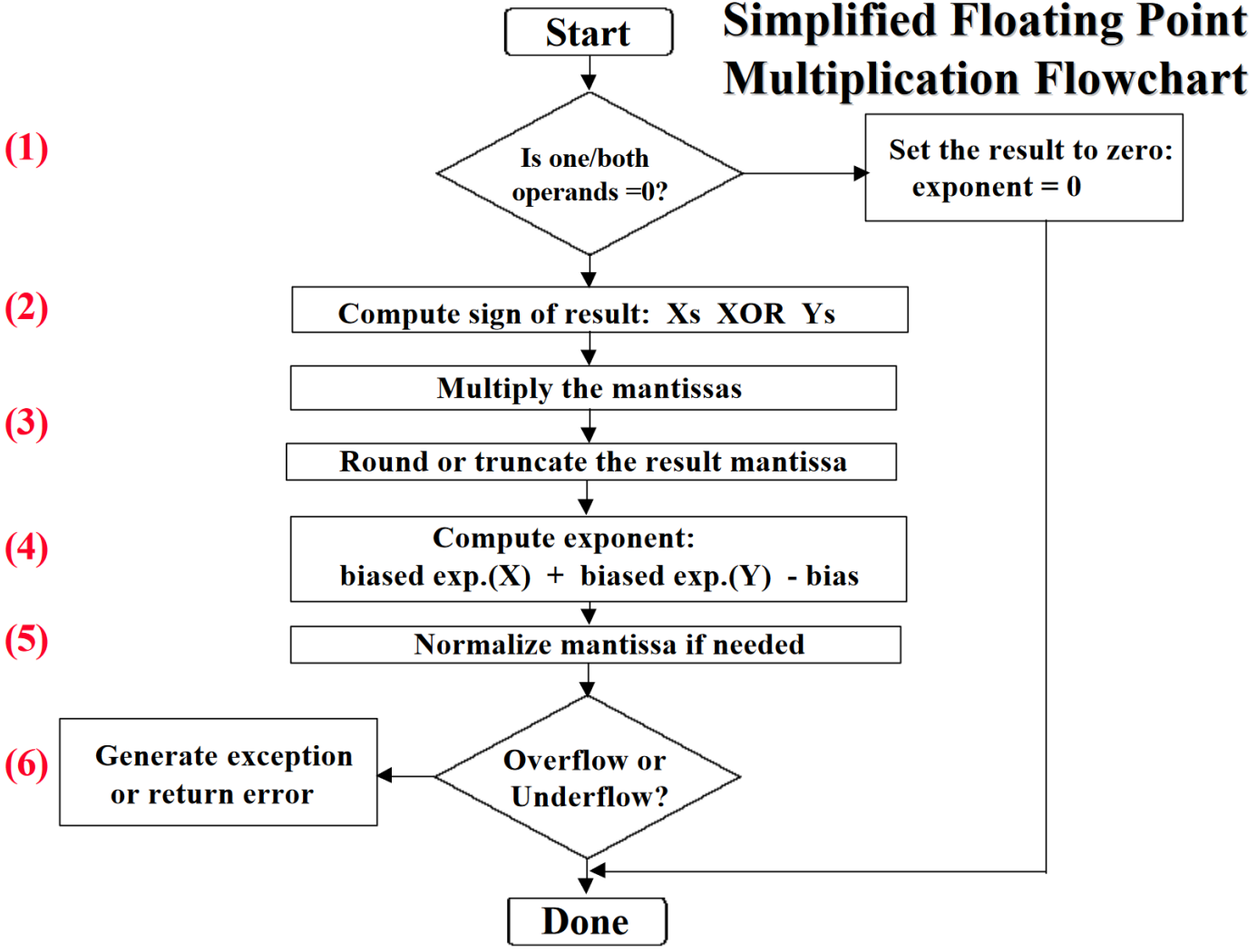
Simplified Floating Point Addition Flowchart



Systems@ETH Zürich

http://meseec.ce.rit.edu/eecc250-winter99/250-1-27-2000.pdf

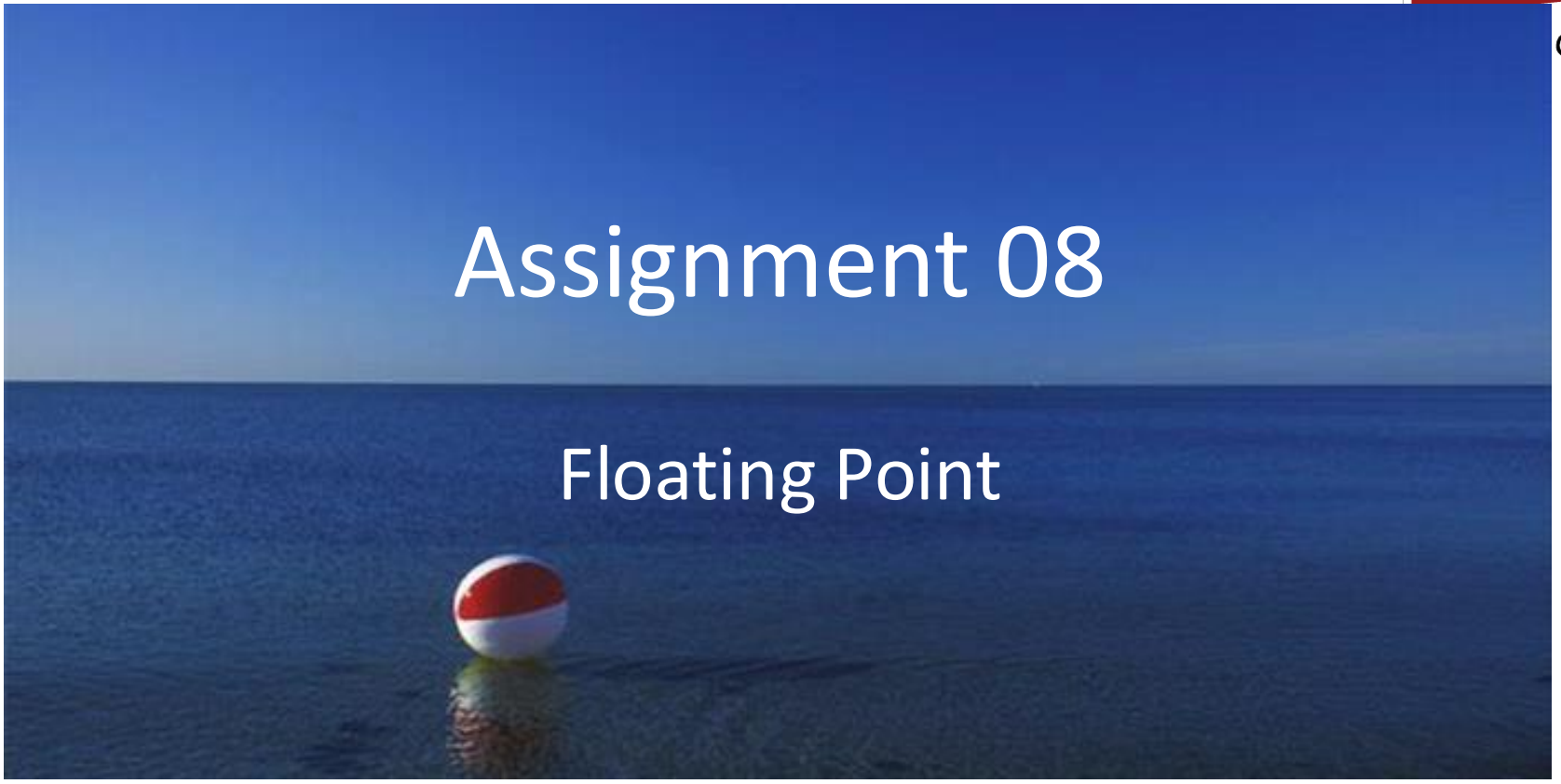
$F=(-1)^S \cdot M \cdot 2^E$ S exp frac



http://meseec.ce.rit.edu/eecc250-winter99/250-1-27-2000.pdf

If you are really interested in further knowledge about floating-point have a look at the following reprint of the paper:

What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg, published 1991 in Computing Surveys


Assignment 08
Floating Point

Part 1: Pen & Paper exercise

- Conversion of Decimal to IEEE floating point numbers

- Checks basic understanding of floating point numbers

Part 2: Now it's your turn!

- Implement your floating point handler in C!
  - No use of floats/doubles
  - Use the given skeleton

Your `float_t`

- You will represent the float as a struct

```
1. typedef struct float_t {
2.     uint8_t sign;
3.     uint8_t exponent;
4.     uint32_t mantissa;
5. };
```

- Challenge: Can you use bit fields for this and simply cast the pointer?

https://www.geeksforgeeks.org/bit-fields-c

Conversion

- The only time you are allowed to use floats is when you convert them to/from your float_t

```
1. float_t fp_encode(float x);
```

```
1. float fp_decode(float_t x);
```

A possible strategy for testing

- Create some float numbers and convert them into your float_t.
  - Choose some “regular” numbers
  - Don’t neglect the corner cases
    - Denormalized
    - NaN
    - Inf
    - -0
- Perform some additions, multiplications, negations using your implementation
- Convert the results back to float and compare

Simple Example Test

```
1. void main() {
2.     float f1 = 1.123;
3.     float f2 = 550;
4.     float f3;
5.     float_t ft1 = fp_encode(f1);
6.     float_t ft2 = fp_encode(f2);
7.     float_t ft3;
8.
9.     f3 = f1+f2;
10.    ft3 = fp_add(ft1, ft2);
11.
12.    assert(f3 == fp_decode(ft3));
13.}
```

Have a nice week!