CSE 351
Section 3

Download the Section Handout!

https://courses.cs.washington.edu/courses/cse351/20su/sections/03/cse351_sec3.pdf

Solutions will be posted this evening.

Floating Point

Floating Point Notation

Scientific Notation in Binary

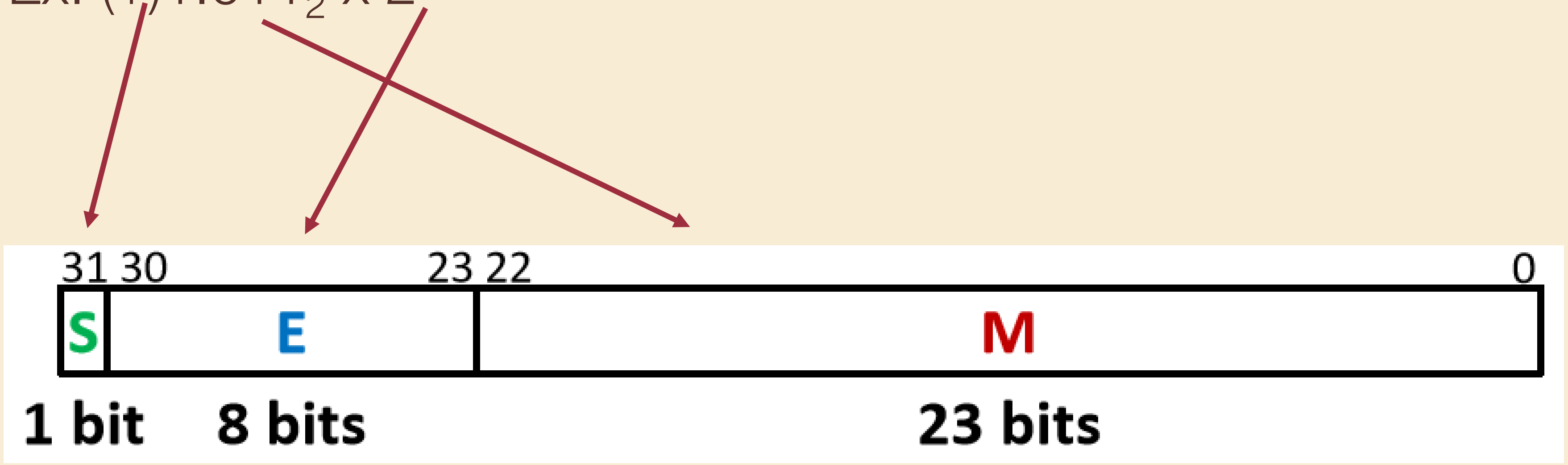
Ex: $2.75_{10} = 10.11_2 = \underset{\text{sign}}{(+)}1.\underset{\text{mantissa}}{011}_2 \times 2^{\overset{}{\underset{\text{exponent}}{1}}}$

sign exponent mantissa

Floating Point Representation

Sign, Exponent, Mantissa are fields at the bit level

Ex: (+)$1.011_2 \times 2^1$



Exponent and the E - Field

Stored with a bias of $2^{w-1}-1$ (w is the bit width of E)

Ex: $(+)1.011_2 \times 2^1$

Bias = $2^{8-1}-1 = 127$

E = Exp + Bias = 1 + 127 = 128 or 0b1000 0000

Stored at the bit level in the E field

Mantissa and the M - Field

Implicit Leading 1 in the M Field

Ex: (+)$1.011_2 \times 2^1$

This 1 is implicit for extra precision!

Mantissa = 1.011

M Field = 0b 01100 … 0

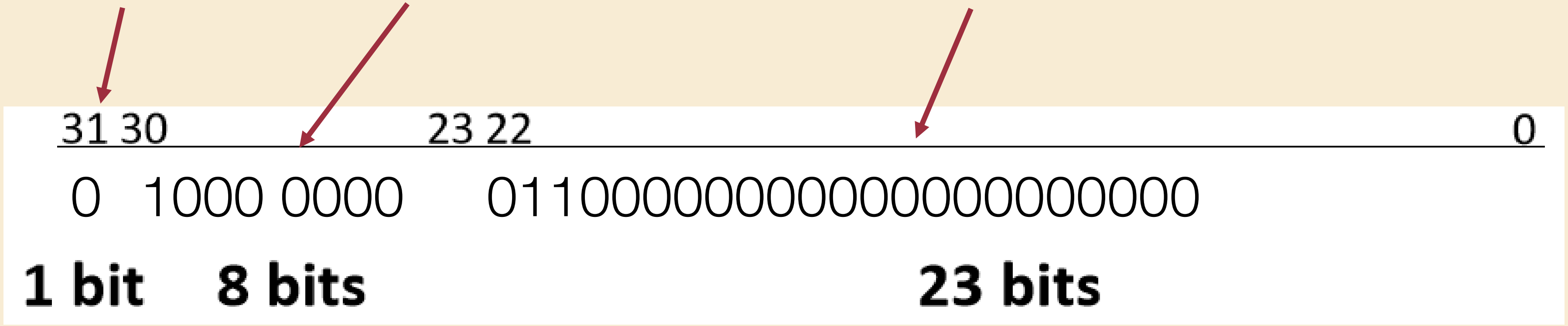
Stored at the bit level in the M field

Putting it Together

Implicit Leading 1 in the M Field

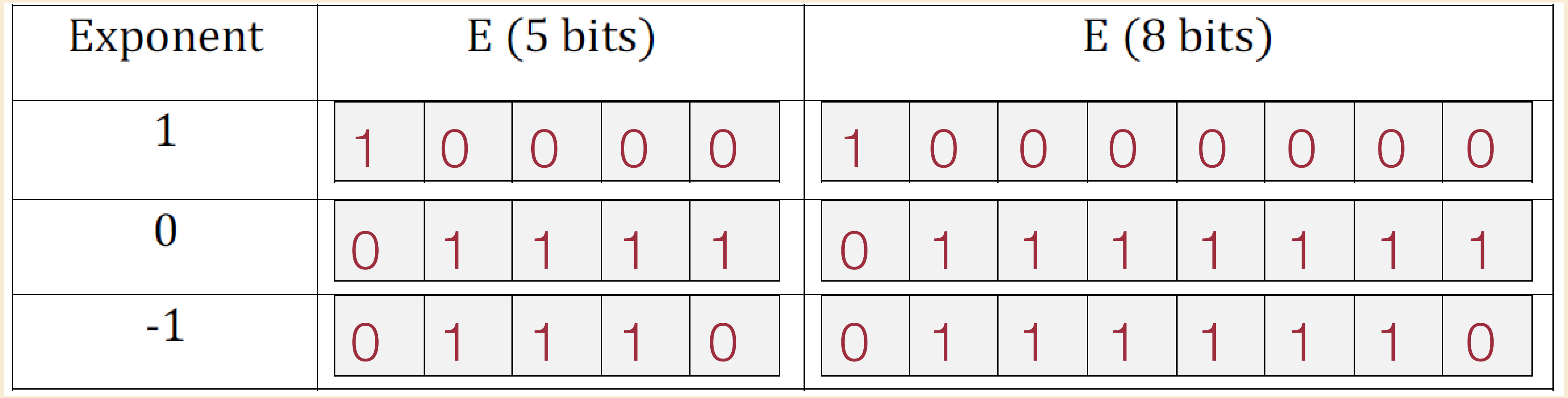
Ex: $(+)1.011_2 \times 2^1$

S = 0 E = 0b 1000 0000 M = 0b 01100 … 0



Practice!!! Exercises 1 and 2

E (5 bits) = 2^(5-1) - 1 = 15



Exponent	E (5 bits)	E (8 bits)
1	1 0 0 0 0	1 0 0 0 0 0 0 0
0	0 1 1 1 1	0 1 1 1 1 1 1 1
-1	0 1 1 1 0	0 1 1 1 1 1 1 0

Practice!!! Exercise 6

Convert 3145728.125_{10} into single precision floating point representation

1. Convert to binary scientific notation

$3145728.125_{10} = 2^{21}+2^{20}+2^{-3} = 1.10000000000000000000000001_2 \times 2^{21}$

2. Get S, E, and M Fields

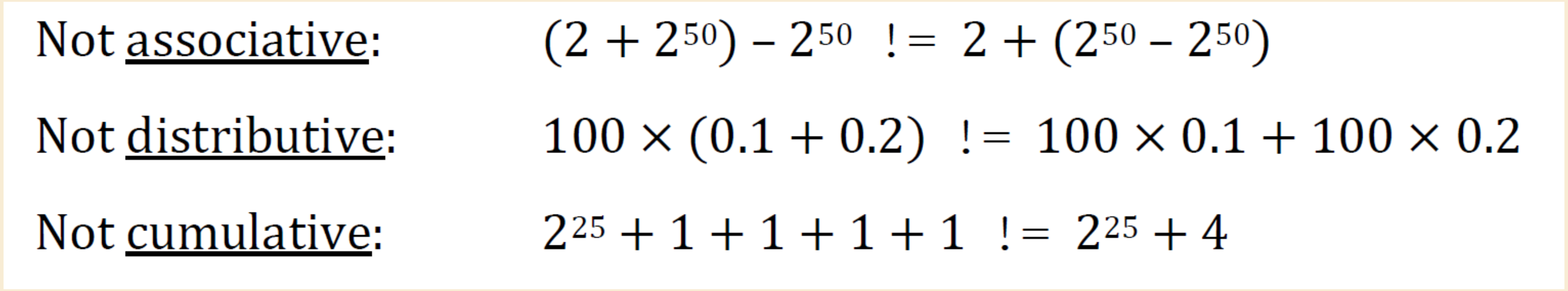
   a. Positive number, so S = 0

   b. Exp = 21, so E = 21 + 127 = 148 = 0b 1001 0100

   c. Mant = $1.10000000000000000000000001_2$ so M = 0b 100…..0

0	1001010 0	10000000000000000000000

Floating Point Gotchas



Not associative: $(2 + 2^{50}) - 2^{50} \;\; != \;\; 2 + (2^{50} - 2^{50})$

Not distributive: $100 \times (0.1 + 0.2) \;\; != \;\; 100 \times 0.1 + 100 \times 0.2$

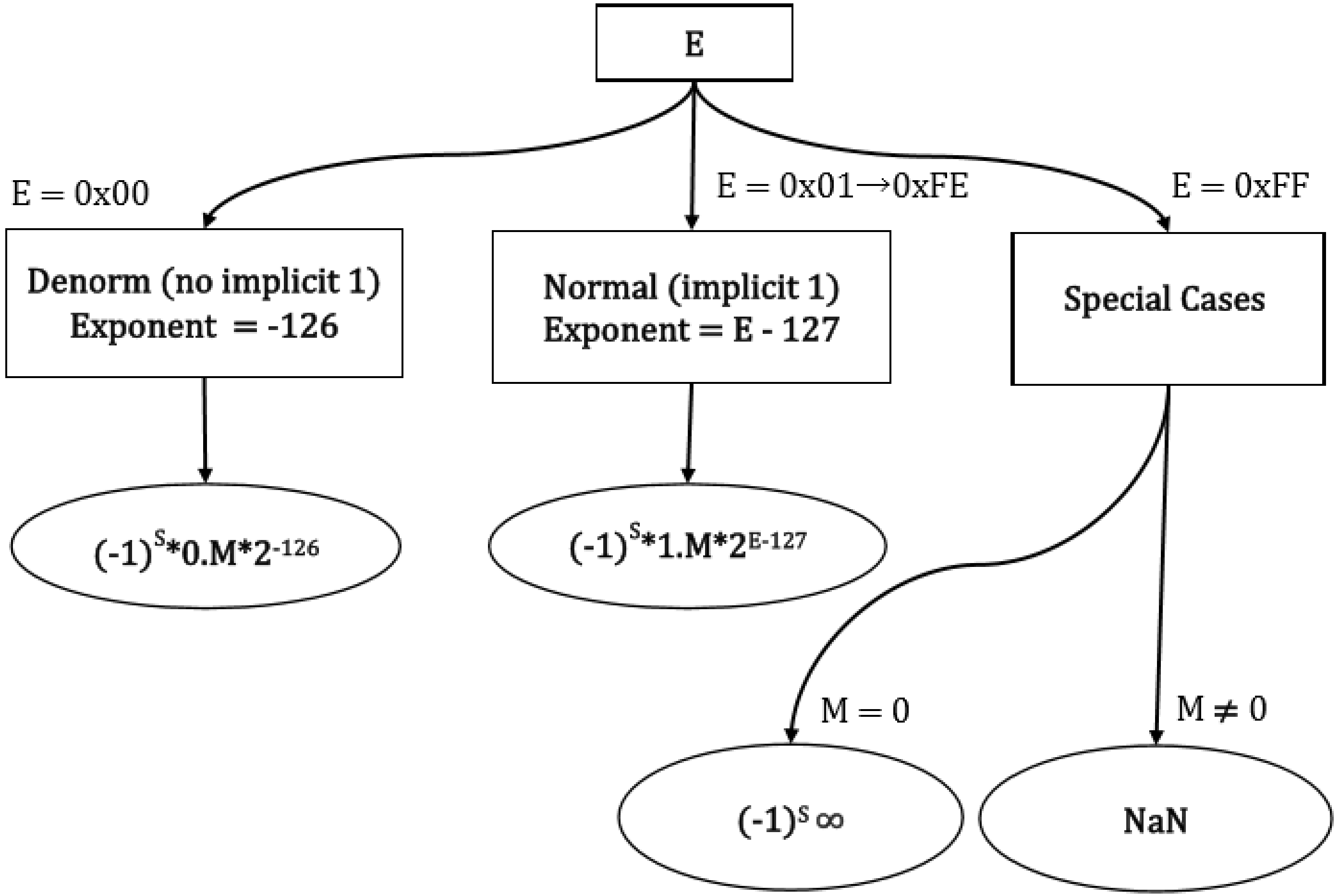
Not cumulative: $2^{25} + 1 + 1 + 1 + 1 \;\; != \;\; 2^{25} + 4$

A. Only 23 bits of mantissa, so 2 + 2^50 = 250 (2 gets rounded off). So LHS = 0, RHS = 2.

B. 0.1 and 0.2 have infinite representations in binary point, so the LHS and RHS suffer from different amounts of rounding.

C. 1 is 25 powers of 2 away from 225, so 225 + 1 = 225, but 4 is 23 powers of 2 away from 225, so it doesn't get rounded off.

IEEE 754 Float (32 bit) Flowchart



x86-64 Assembly

x86-64 Assembly

Assembly language is a human-readable representation of machine code instructions (generally a one-to-one correspondence). Assembly is machine-specific because the computer architecture and hardware are designed to execute a particular machine code instruction set.

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers. It was developed by Intel and AMD and its 32-bit predecessor is called IA32. x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

Data and Instructions

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions. The subset of x86-64 instructions that we will use in this course take either one or two operands, usually in the form:

```
instruction operand1, operand2
```

There are three options for operands:

- Immediates: constants (e.g. `$0x400`)
- Registers: fast memory accessible to the CPU (e.g. `%rax, %edx`)
- Memory: memory addresses computed with `D(Rb, Ri, S)`
  - such as `0x400(%rdi, %rsi, 4)` = ***(%rdi + 4 * %rsi) + 0x400***

Operand Size

The operation determines the effect of the operands on the processor state and has a suffix (“b” for byte, “w” for word, “l” for long, “q” for quad word) that determines the bit width of the operation. Sometimes the operation size can be inferred from the operands, so the suffix is omitted for brevity. For example:

- `movb src, dst` - copies 1 byte from src to dst
- `movw src, dst` - copies 2 bytes from src to dst
- `movl src, dst` - copies 4 bytes from src to dst
- `movq src, dst` - copies 8 bytes from src to dst

Interpreting Instructions

What do the following assembly instructions do?

X86-64 instruction	English equivalent
movq $351, %rax	Move the number 351 into 8-byte (quad) register “rax”
addq %rdi, %rsi	Add the 64-bit value of %rdi to %rsi
movq (%rdi), %r8	Move the 64-bit data at the address stored in %rdi to %r8
leaq (%rax,%rax,8), %rax	Compute 9 * %rax, and store the 64-bit result in %rax

Exercise 1

Symbolically, what does the following code return? Remember, register %rax is used to store the return value.

```
movl (%rdi), %eax               # %rdi -> x    *x
leal (%eax,%eax,2), %eax        # %rax -> r    *x * 3
addl %eax, %eax                                (*x * 3) * 2
andl %esi, %eax                 # %rsi -> y    (*x * 6) & y
subl %esi, %eax                                ((*x * 6) & y) - y
ret
```

The GNU Debugger (GDB)

The GNU Debugger is a powerful debugging tool that will be critical to Lab 2 (releasing tomorrow!) and Lab 3 and is a useful tool to know as a programmer moving forward.

There are tutorials and reference sheets available on the course webpage, but we'll be doing a short demo of the basics in class (you can find the handout to follow along on the course website).

We've also provided an (optional, but highly recommended) guided tutorial on Gradescope which even walks you through the first "phase" of Lab 2!