Five Elements of a Computer
- Control
- Datapath
- Memory
- Input
- Output

Negative Numbers
- Sign/Magnitude
- One's Complement
- Two's Complement
- Pros, Cons of Each

Memory Management
- Static
- The Stack
- The Heap

C Topics
- Pointers!
- malloc, free
- Handles
- Pass by Value vs Pass by Reference
- Arrays
- Structs
- typedef

Memory Management Allocation Schemes
- Best-fit
- First-fit
- Next-fit
- Slab
- Buddy
MIPS
- R, I, J format instructions (on your green sheet!)
- MAL vs TAL
- MIPS to Binary, Binary to MIPS
- Difference between branches, jumps

Various Other Things
- Floats
- CALL (Compile, Assemble, Link, Load)

Garbage Collection
- Reference Counting
- Mark and Sweep
- Copying
- Pros and Cons
Socratic Method,
Understanding Floats

Float Cheat Sheet

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>E bits</td>
<td>F bits</td>
</tr>
</tbody>
</table>

Normalized Float:
\[-1]^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}

Denormalized Float:
\[-1]^S \times \text{Significand} \times 2^{(1 - \text{Bias})}

Bias = \(2^{(-1) - 1}\)

(0 and all 1s)

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0\times 2 + 2 = 4</td>
</tr>
<tr>
<td>2</td>
<td>1 (all 0s)</td>
<td>2</td>
</tr>
</tbody>
</table>

Just as in sign and magnitude, the sign bit encodes the sign of the number, 0 means positive, 1 means negative.

Float Cheat Sheet

The significand is encoded as a fixed point unsigned number, such that the most significant bit has a value of \(2^{(-1)}\). Accordingly, the significand always has a value < 1.

The exponent is encoded as an unsigned integer with a bias. The bias rotates the number ring such that the value zero no longer corresponds with the bitpattern all 0s. Usually this is a bad thing, but here, it’s what we want.

Warm Up

Turn these decimal numbers into binary:
22, 1.5, 5/64, 22/32

Now normalize them.
Now make them into (single precision) floats.
Warm Up

Turn these decimal numbers into binary:
22, 1.5, 5/64, 22/32
10110, 1.1, 0.00101, 0.10110
Now normalize them.
1.011x2^3, 1.1x2^0, 1.01x2^-3, 1.011x2^-1
Now make them into (single precision) floats.
[04+127][00110...0]
[00+127][010...0]
[0-3+127][00010...0]
[0-1+127][000110...0]

Question:
Why bother with a bias? Can we just use a Two’s comp. exponent representation?

Related questions:
Talk to your neighbor about these!
Which of the following two (single precision) floats is bigger?
0x7F00 0000 or 0x0080 0000
Which of the following two integers is bigger?
0x7F00 0000 or 0x0080 0000
Now assume we used a two’s complement exponent instead, which of the two floats is bigger?
0x7F00 0000 or 0x0080 0000
What would zero encode as with a two’s complement exponent?

Sure, it works. But …
Biased exponent => existing integer hardware comparators still work!
Zero = 0x4000 0000 => kind of weird.
Most negative exponent = 0b1000 0000

Question:
Why is the bias $2^{(E-1)} - 1$ (0 and all 1s)?

It’s a design choice.

$2^{(E-1)} - 1$ splits the representation about 1.0
Half the positive floats are < 1, Half are > 1

Question:
Why is the implicit denorm exponent (1-Bias)?

Related questions:
What is the smallest non-zero denorm?
What is the second smallest, third?
What is the step size for denormalized numbers?
How many positive denoms are there?
What is the value of the largest denom?
How does this value relate to step size and # denoms?
What is the value of the smallest normalized float?
What is the step size b/w this smallest normal and its greater neighbor?
How far apart are the smallest normal and the largest denom?
Suppose the denorm exponent were (0-Bias), as the normal pattern suggests, what would the step size be?
Considering the number of steps and the step size, what would the largest denorm’s value be?
Question: Why is the implicit denorm exponent (1-Bias)?

Related questions:
- What is the smallest non-zero denorm? \(-1\)\(2^{-128}\) = \(2^{-149}\)
- What is the second smallest, third? \(2^{-149} = 2^{-148} \times 2^{-148} = 2^{-296}\) (2-23)\(-126\) = \(2^{-149}\)
- How does this value relate to step size and # denorms? Stepsize \# denorms
- What is the value of the largest denorm? \((2^{23} - 1) \times 2^{-149}\) = \(2^{-126} - 2^{-149}\)
- How does this value relate to step size and # denorms? Stepsize \# denorms
- What is the value of the smallest normalized float? \((-1) \times 1.0 = 2^{-126}\)
- What is the step size b/w this smallest normal and its greater neighbor? \(2^{-149}\)
- How far apart are the smallest normal and the largest denorm? \(2^{-149}\)

Suppose the denorm exponent were (0-Bias), as the normal pattern suggests, what would the step size be? \(2^{-150}\)

Considering the number of steps and the step size, what would the largest denorm’s value be? \((2^{23}) \times (2^{-150}) = 2^{-127} - 2^{132}\)

Question: Why is \(2^{24} + 1.0 = 2^{24}\), but \((2^{24} + 2^1) + 1.0 = (2^{24} + 2^2)\)?

Related questions:
- Why are there floats \(X\) such that \(X + 1.0 = X\)?
- What is the smallest such number? (Hint: Think about lab 6.)
- What do the bottom-most bits of \(2^n\)'s significand look like?
- What about \((2^n + 2^1)\)'s significand?
- What are the four rounding modes that floats use?
- How would they round the following binary numbers to the nearest integer? 0.00, 0.001, 0.010, 0.011, 0.100, 0.101, 0.110, 0.111
- Which patterns do the lower bits of the significands of \(2^n\) and \((2^n + 2^1)\) match with? What about after you add 1.0?
- What rounding modes could make the stated question possible?

Question: Why is the implicit denorm exponent (1-Bias)?

Implicit Exponent = (0-Bias) => Gaps in Representation
Using (1-Bias):

Denoms Smallest Norm Exponent Next Norm Exponent

Using (0-Bias):

Question: Why is \(2^{24} + 1.0 = 2^{24}\), but \((2^{24} + 2^1) + 1.0 = (2^{24} + 2^2)\)?

Only 23 significand bits means 1.0 is just barely too small relative to \(2^n\)’s implicit 1 to be saved.

Floating point unit has 2 guard bits used for intermediate computation. The bits of the first computation look like this: [1][00...00]/[1][0]

The bits of the second computation look like this: [1][00...00]/[1][0]

Need to round to fit the guard bits in the significand. Default (aka unbiased or round to even) rounding mode would round to [1][00...000] and [1][00...010] respectively.

Round towards \(+\)infinity would do the same.
MIPS, C, and You

Ropes

```c
struct ropeNode {
    char * string; // 0x0 offset
    struct ropeNode * next; // 0x4 offset
}
typedef struct ropeNode rope;

rope weave(char * str, rope r) {
    // append str to the front of r (copy str)
}
void fray(rope r) {
    // free all memory associated with r
}
```

Weave, C->MIPS

```c
struct ropeNode {
    char * string; // 0x0 offset
    struct ropeNode * next; // 0x4 offset
}

rope weave(char * str, rope r) {
    rope end = (rope) malloc (sizeof(struct ropeNode));
    end->string = (char *) malloc ((strlen(str) + 1)*sizeof(char));
    strcpy(end->string, str);
    end->next = r;
    return end;
}
```

Weave, in MIPS

```
weave:
    # prologue
    addiu $sp, $sp, -16
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    # body
    move $s0, $a0
    move $s1, $a1
    li $a0, 8
    jal malloc
    move $s2, $v0
    move $s1, $a1
    li $a0, 8
    jal strlen
    addiu $v0, $v0, 1
    jal strcpy
    sw $v0, 0($s2)
    move $s0, $v0
    move $s1, $a0
    jal fray
    move $s0, $a0
    lw $t0, 4($s0)
    beq $t0, $zero, done
    free($r->next);
    free($r);
    # fill me in!
```

Fray, C->MIPS

```c
struct ropeNode {
    char * string; // 0x0 offset
    struct ropeNode * next; // 0x4 offset
}

void fray(rope r) {
    if (r->next != NULL)
        fray(r->next);
    free(r->string);
    free(r);
}
```

Fray, in MIPS

```
fray:
    # prologue
    addiu $sp, $sp, -8
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    # body
    move $s0, $a0
    lw $t0, 4($s0)
    beq $t0, $zero, done
    free($r);
    free($r);
    # fill me in!
```

```