Assignment 6: Floating Point

CSE 30: Computer Organization and Systems, Fall 2023

[illegible]ul Cao and Bryan Chin

[illegible]sday Nov 14, 2023

Please read over the [illegible] before starting to get a sense of what you will need to get done in the next [illegible]: Everyone procrastinates but it is important to know that you are procrast[illegible] yourself enough time to finish. Start early, start often. You MUST run the assignment on the pi-cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.

ACADEMIC INTEGRITY REMINDER: you should do this assignment on your own. If you work with others, be sure to cite them in your submission. Never copy from others.

Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on Edstem.

Version updates:

- 1.0 [Nov 8] Final Draft
- 1.1 [Nov 8] Fix midpoint due date Sunday -> Friday
- 1.2 [Nov 9] Fix: somehexnums.txt 0x8000 to 0x4000 since it is 15 bit representation.
- 1.3 [Nov 9] Clarify: style won't be regraded during resubmission
- 1.4 [Nov 10] Prerelease: Midpoint answers are visible before due date.
- 1.5 [Nov 12] Fix git clone link, fix # of bits in mantissas in page 4 table to be 8 bits

Table of Contents

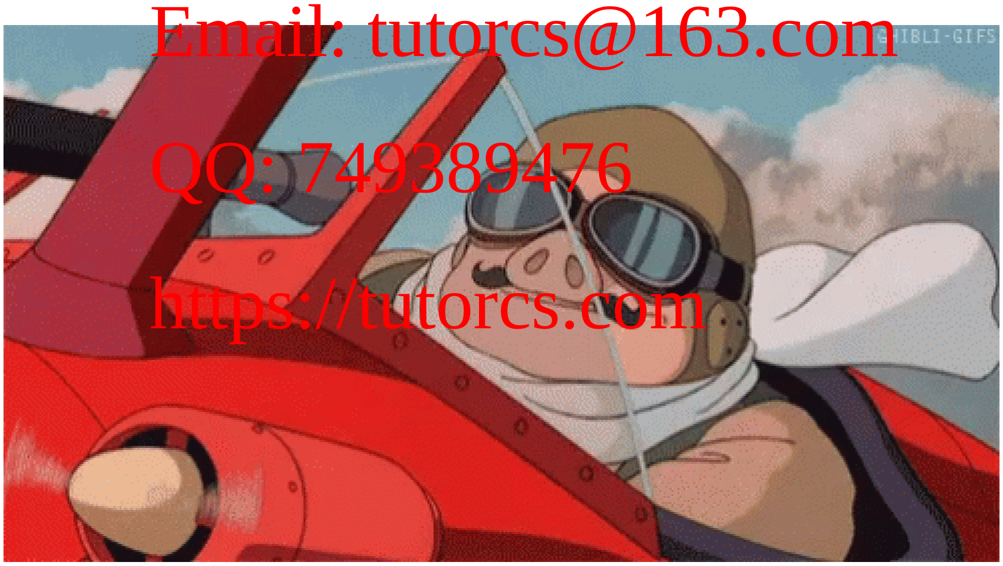
Learning Goals

- Programming in ARM assembly
    - Bit masking
    - Functi
    - Branc
- Working with s
- Coordinating ARM assembly and C code (you aren't writing in C)

Assignment

At the peak of time where pirates and bounty hunters are in the air. Porco Rosso makes his rounds in the vast ocean to capture any air pirates that disturb the peace near Adriano hotel.

On the radio, Porco Rosso tunes in to listen to his next job, however he discovers an issue. The coordinates given out are in 15-bit floating-point format. He doesn't know how to convert from this format and only knows the standardized IEEE 754 format. Gina only has devices that are written in ARM so Porco plans to rely on your assembly skills to create the conversion function. Help him write and test code to convert the coordinates into IEEE format!



A note about representing number literals

In the number `8'b1101_0011`:

- `8` is the number of binary bits in this number, in base-10.
- `b` means binary. Other formats are `d` for decimal, `o` for octal, and `h` for hexadecimal.
- To conserve space, we may also write the bits in hexadecimal, `0xd3` is equivalent to `8'b1101_0011`.

- _ is a spacer character that is only there to make it easier to read. It has no numerical meaning. A '_' is usually placed every four digits.

You can read more about where this number literal representation comes from here. (Note: Anything past the first slide is irrelevant to this course, but will be useful in CSE 140 & 141.)

The 15-bit FP

The 15-bit floating-po[...]to, but not the same as, the one we studied in class. It has a sign bit, 6 bits [...] bias value of 31 (base-10), and 8 bits of mantissa. Note that we include [...]esent infinities and subnormal numbers.

The following figure illustrates the bit assignments:

FP Format (15-bit)		
sign (1 bit)	exponent (6 bits, bias = 31)	mantissa (8 bits)

Points to note:

1. There is an implied **"1."** in front of the mantissa, *unless* it is a subnormal number.
2. Subnormal numbers have an exponent field of `6'b000000` which represents 2^{-30} and implies a **"0."** in front of the mantissa.
3. "Infinite" numbers have an exponent field equal to `6'b111111` with **ANY** value in the mantissa.

The following table shows how to interpret the exponent fields and mantissa fields.

Exponent/mantissa	represents	Notes
111111/mmmmmmmm	**infinity**	**infinity**
111110/mmmmmmmm	2^31 x 1.mmmmmmmm	normal number
111100/mmmmmmmm	2^29 x 1.mmmmmmmm	normal number
111000/mmmmmmmm	2^25 x 1.mmmmmmmm	normal number
100000/mmmmmmmm	2^1 x 1.mmmmmmmm	normal number
011111/mmmmmmmm	2^0 x 1.mmmmmmmm	normal number

001111/mmmmmmmm	2^-16 x 1.mmmmmmmm	normal number
000011/mmmmmmmm	2^-28 x 1.mmmmmmmm	normal number
000001/mmmmmmmm	[illegible] x 1.mmmmmmmm	normal number
000000/mmmmmmmm	[illegible] x 0.mmmmmmmm	**subnormal** number (no leading 1)

``````````````````````````````````````````````````````````````````````````````

Exponent bits are sh[illegible]o help you distinguish it from the sign bit and mantissa.



| Number | Encoding in 15-bits |
|---|---|
| +0.0 | 15'b000_0000_0000_0000 (15 bits of 0 in binary) |
| -0.0 | 15'b100_0000_0000_0000 |





| Number | `15-Bit Representation | Binary Representation | Base-10 Representation |
|---|---|---|---|
| +∞ | 15'b011_1111_xxxx_xxxx | | +Inf |
| -∞ | 15'b111_1111_xxxx_xxxx | | -Inf |
| Most positive # | 15'b011_1110_1111_1111 | 2^31 * 9'b1.11111111 | 4286578688 |
| Smallest positive # (subnormal) | 15'b000_0000_0000_0001 | 2^-30 * 9'b0.00000001 | 2^-38 ≅ 3.637978807e-12 |
| Most negative # | 15'b111_1110_1111_1111 | -2^31 * 9'b1.11111111 | -4286578688 |
| Smallest negative # (subnormal) | 15'b100_0000_0000_0001 | -2^-30 * 9'b0.00000001 | -2^-38 ≅ -3.637978807e-12 |
``````````````````````````````````````````````````````````````````````````````

IEEE-754 Single Precision Format

IEEE-754 Single Precision Format		
sign (1 bit)	ex[illegible] (8 bits, [illegible]	mantissa (23 bits)

Subnorms

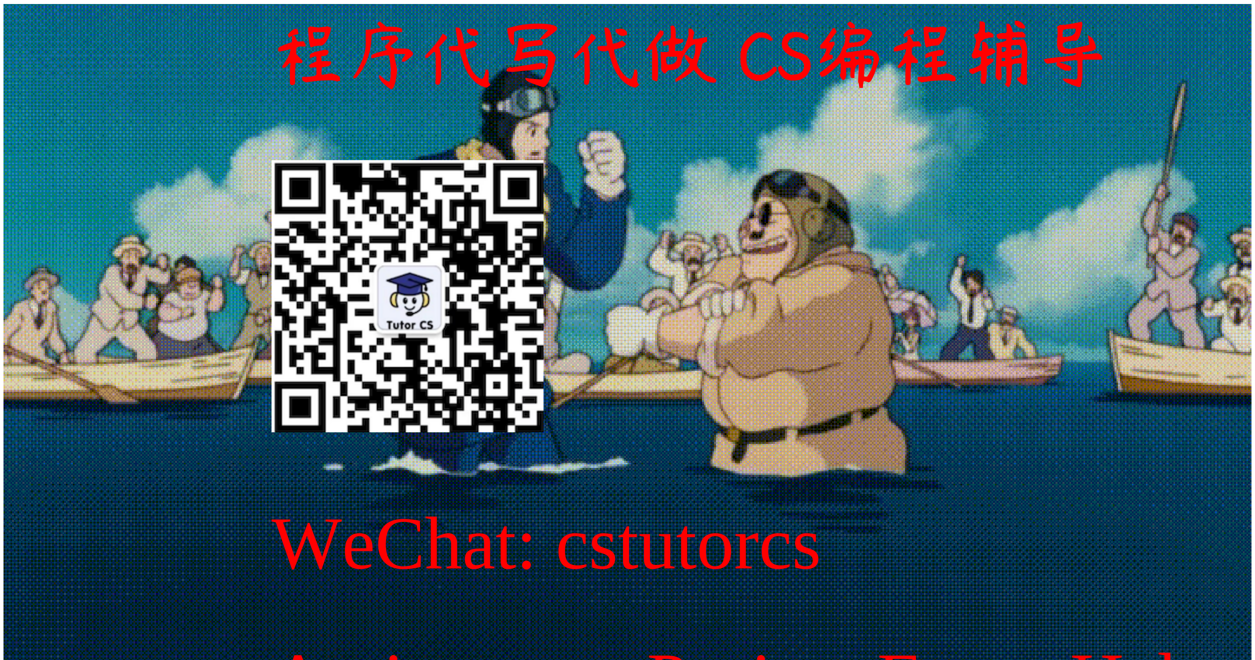
The bias for the IEEE [illegible]-10) and the format uses an implied "1." for normal numbers, as usual. The smallest possible exponent is -126 represented by `8'b0000_0001` for normal numbers, whereas `8'b0000_0000` represents subnormal numbers. For subnormal numbers, we prepend the mantissa with "0." instead of "1." similar to how subnormal numbers are evaluated in our 15-bit FP format.

Infinities

In IEEE single precision, any exponent of all 1's (`8'b1111_1111`) represents a number too large to represent. For example, `0xff80_0000` is a number with a negative sign bit, all 1's for the exponent and all 0's for the mantissa. This represents negative infinity (-Inf). Similarly, `0x7f80_0000` represents positive infinity (+Inf). **Note that the mantissa bits are all 0.** Non-0 mantissa bits represent another kind of IEEE special number (`NaN`, "not a number") which is not required in this assignment since our 15-bit floating point format does not use NaN.

Summary of Select Conversions

FP	15-bit FP	IEEE-754 Single
`+0`	`15'b000_0000_0000_0000`	`0x00000000`
`-0`	`15'b100_0000_0000_0000`	`0x80000000`
`2^-38`	`15'b000_0000_0000_0001`	`0x2c800000`
`-2^-38`	`15'b100_0000_0000_0001`	`0xac800000`
`4286578688`	`15'b011_1110_1111_1111`	`0x4f7f8000`
`-4286578688`	`15'b111_1110_1111_1111`	`0xcf7f8000`
`+Inf`	`15'b011_1111_xxxx_xxxx`	`0x7f800000`
`-Inf`	`15'b111_1111_xxxx_xxxx`	`0xff800000`



Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**.

Need help or instructions? See the Edstem FAQ. (Do NOT wait until the end to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following link:
https://github.com/cse30-fa23/hw6-starter

1. Download the files in the repository.
   a. You can either use `git clone https://github.com/cse30-fa23/hw6-starter.git` directly from pi-cluster, or download the repo locally and `scp` the folder to `pi-cluster` if that doesn't work.
2. Fill out the fields in the `README` before turning in.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, the reference solution binary will be placed on **Saturday 11/11 morning** at:

`/home/linux/ieng6/cs30fa23/public/bin/fpconvert-a6-ref`

Makefile: The `Makefile` provided will create a `fpconvert` executable from the source files provided. Compile it by typing `make` into the terminal. Run `make clean` to remove files generated by `make`.

How the Pr[illegible]s

Your program will tak[illegible]gument and read it in. This file is a txt file storing the 15 bit FP numbers. T[illegible]plemented for you in `main.c`) will parse the input file and call the `fpconve`[illegible]ou will implement in assembly on each 15-bit FP number to convert it into IEEE floating point format, and print the result to `stdout`.

Once you compiled the program with `make`, you can run it as follows:


```
./fpconvert somehexnums.txt
```

where `somehexnums.txt` is the name of the input `txt` file that holds the `hex` numbers you want to convert.

Input Guarantees

- fpconvert will be only given valid 15-bit wide numbers.

Program Implementation

Files to Edit

You need to edit `fpconvert.S` and `convert_inf.S`

Functions to Implement

You will need to implement 1 function within `fpconvert.S`:

- `fpconvert(n)`: This is the function that will do most of the floating-point conversion.
  - **Argument:** `n` the 15-bit FP number to convert
  - **Returns:** n's equivalent IEEE 754 single precision representation.
  - If `n` is ±infinity, you **MUST** call `convert_infinity(n)` to do the conversion instead.

You need to implement 1 function within `convert_inf.S`

- `convert_infinity(n)`:

- ○ **Argument:** `n`, the 15-bit FP number to convert (should only be ±infinity)
- ○ **Returns:** the FP number's equivalent IEEE 754 single precision representation.

NOTE:

- 32-bit ARM st[illegible]ed into the function in registers r0-r3; `n` only symbolizes th[illegible] in one argument. You cannot directly use `n` in your assembly pro[illegible]rst argument.
- As registers a[illegible]ur 15-bit floating point format will always only occupy the least signi[illegible]ber 17 bits will be padded with 0's.
- Return value [illegible]).

Calling a Function in ARM

To call a function in ARM, you **must** use the `bl` "branch and link" instruction. It is **not sufficient or correct** to use a regular branch instruction. Without branch-and-link, the return operations in the epilogue of the function will not work and return as expected.

Developing Your Code

Development Process

To make development easier, you should first implement the conversion of normal numbers. Test your code on a range of normal numbers (smallest, largest). For the smallest numbers, you should familiarize yourself with their scientific notation representations. You can also check the IEEE column of the output to see if it matches the expected IEEE version. Additionally, be sure to check the special cases of +0.0, -0.0, +Inf, and -Inf.

After thoroughly testing the functionality of your code, you should consider subnormal numbers. Subnormal numbers are represented when the exponent field is `6'b000000`.

After implementing the conversion of subnormal numbers, your code should be able to produce all of the values in the Summary of Select Conversions table.

Development Tips

Before you write assembly code, think about the algorithm.

- How are the 15-bit format and the 32-bit IEEE format similar and different?
- How do I break down the 15-bit format into the 3 individual fields?
- How does each field convert from the 15-bit format to the 32-bit IEEE format?

You should find the bitwise instructions useful for this assignment. In particular, you will want to make use of bitmasks.

While an immediate can only be 8 bits wide, you can use left and right shifts to move the mask into the right position. For example, if you need the bitmask `0xFF00`, you can shift the immediate `0xFF` left by 8 bits.

Testing your C

To run your code you … holds the hex numbers that you want to convert, separated by a new l…

Example text input fil… `ums.txt`:

```
0x0000
0x4000
0x3f00
0x7f00
0x3eff
0x0001
0x7eff
0x4001
```


NOTE: you should make sure each hexadecimal number only has four digits, otherwise you may get unexpected results.

Checking For Exact Output Match

A common technique is to redirect the outputs to files and compare against the reference solution[1]:

```
./your-program args > output ; our-reference args > ref
diff -s output ref
```


This command will output lines that differ with a < or > in front to tell which file the line came from.

Debugging Tips

The public autograder will only `printf` test some features. DO NOT RELY ON THE AUTOGRADER. (Many CSE 30 students have been burned by this.) Test your code using your own test cases!

GDB treats ARM assembly labels as functions except those that **begin with the prefix "`.L`".** If you want to use GDB to debug your ARM code, you will need to prefix your labels with "`.L`".

---

[1] You might want to check out `vimdiff` on the pi-cluster (https://www.tutorialspoint.com/vim/vim_diff.htm).

Thus, ARM code for the given C if statement would look like the code snippet on the right, rather than the snippet on the left.


```
if (r5 == 99) {
    r3 = r3 + 2;
} else {
    r3 = r3 + 3;
}
r4 = r4 - 1;
```

GDB will not recognize labels:

```
    cmp r5, 99
    bne else
    add r3, r3, 2
    B end_if
else:
    add r3, r3, 3
end_if:
    sub r4, r4, 1
```

GDB will recognize labels:

```
    cmp r5, 99
    bne .Lelse
    add r3, r3, 2
    b .Lend_if
.Lelse:
    add r3, r3, 3
.Lend_if:
    sub r4, r4, 1
```


Allowed ARM Instructions

You are only allowed to use the instructions provided in the ARM ISA Green Card. Failure to comply will result in a score of 0 on the assignment.

Style Requirements

Reading raw assembly is hard and debugging will be nigh impossible if you don't put comments! To encourage you to make your life easier, style will be worth 2 points in this assignment on a holistic readable/unreadable basis. You will get full style points as long as your code is reasonably commented to be readable (so that someone who doesn't know ARM can still roughly understand what it's doing), so don't worry if you can't get all the details right. However, you will get no style points if it's not (e.g. very inconsistent indentation, sparse or unreadable comments). In addition, **staff won't be able to provide any assistance other than styling advice unless code is readable**. For reference, here is the Style Guideline for ARM assembly. **We strongly recommend** you to use comments after each instruction to help describe what step occurs like what is done in the style guide. **Note: style will not be graded for resubmission.**

Midpoint (5 Points)

This part of the assignment is due earlier than the full assignment, on

Friday 11/10 at 11:59 pm PST. There are no late submissions on the Midpoint.

Complete the Gradescope assignment "HW6: Checkpoint", an Online Assignment that is done entirely through Gra[illegible]ignment consists of a few quiz questions and a free-response questi[illegible]ument your algorithm in plain English or C code.

Discuss your im[illegible] the following functions: `fpconvert` and `convert_infinit`[illegible] should call `convert_infinity` when appropriate.

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled "HW6 (Coding): Floating Point". You will submit ONLY the following files:

   ```
   fpconvert.S
   convert_inf.S
   README.md
   ```


   Submission will open by Saturday morning. You should test your code extensively on pi-cluster before submitting to gradescope.

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint.

   Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
   a. Check that all required files were submitted.
   b. Check that `fpconvert.S` and `convert_inf.S` compiles.
   c. Runs some sanity tests on the resulting `fpconvert` executable.

Grading Breakdown [5 + 45 points]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests as this is an insufficient test of your program.***

To encourage you to write your own tests, we are not providing any public tests that have not already been detailed in this writeup.

The assignment will be graded out of 50 points and will be allocated as follows:

- Midpoint write… **part of the assignment is due earlier than the full assignment,** …omplete the Gradescope assignment "Homework 6: Checkpoint".
- Code compile… point
- Style: 2 point…
- Public tests w…nples.
- Private tests with hidden test cases.

NOTE: The tests expect an EXACT output match with the reference binary. There will be NO partial credit for any differences in the output. Test your code - do NOT rely on the autograder for program validation.

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster without warnings. **Any assignment that does not compile will receive 0 credit.**

[Optional] Bells and Whistles[2] (epsilon points)

Write a new function `add_fp(a, b)` that takes in 2 numbers `a` and `b` that are in the 15-bit floating point format. It should add these 2 numbers together and return the value. However, what makes this complicated is that `a` and `b` may not have the same exponent! You'll need to make the exponents the same first before you can add them.

This part of the assignment will **NOT** be graded - and does not need to be submitted. It is completely up to you to try writing a program which achieves the above output.

The Bells and Whistles component may be submitted to a separate Gradescope assignment: Homework 6 Optional: Bells and Whistles.

[2] In our experience, students like extra credit. However, we find that extra credit isn't used by students who need it the most. Thus, we have an extra credit where the number of points assigned is epsilon, where epsilon is a very small number [0, 1).