Assignment 4: Phloating Point PinKY

Phloating Point PinKY! Yup; "P3"... except this is assignment 4. Meh. You see how hard I try to make cute names for these things? No matter. In this project, you're going to be implementing high-performance floating point arithmetic in a pipelined version of PinKY -- you know, like the one you just built for assignment 3. It's a cool project even if the name isn't.

Let's start by reminding you of the floating-point instructions that PinKY is supposed to have -- but you didn't implement yet:

Instruction	Description	Functionality	Suffix Forms
`ADDF Rd, Op2`	ADD Floats	`Rd += Op2`	ADDF, ADDFS, ADDFEQ, ADDFNE
`FTOI Rd, Op2`	Convert Float TO Integer	`Rd = ((int)Op2)`	FTOI, FTOIS, FTOIEQ, FTOINE
`ITOF Rd, Op2`	Convert Integer TO Float	`Rd = ((float)Op2)`	ITOF, ITOFS, ITOFEQ, ITOFNE
`MULF Rd, Op2`	MULtiply floats	`Rd *= Op2`	MULF, MULFS, MULFEQ, MULFNE
`RECF Rd, Op2`	RECiprocal Float	`Rd = 1.0/Op2`	RECF, RECFS, RECFEQ, RECFNE
`SUBF Rd, Op2`	SUBtract Floats	`Rd -= Op2`	SUBF, SUBFS, SUBFEQ, SUBFNE

In many computers, floating-point values go in completely different registers from integer values. However, that's most often because they were an afterthought, and sometimes were not even handled by the main processor, but by a floating-point coprocessor on a separate chip. In contrast, the floating-point instructions were part of PinKY's design from day one, so we try to leverage as much of the existing integer infrastructure as possible.

In other words, it's not just that we use the same registers for floating-point or integer values, but that we can reuse most instructions too. For example, `LDR`, `MOV`, and `STR` still work perfectly well with 16-bit floating-point values. Perhaps it is less obvious, but the IEEE floating-point representation of zero is the same bit pattern used for the integer value zero -- so the zero flag and all the

conditional support works too, and thus so does control flow (e.g., using `ADDEQ pc,lab-.` as a conditional branch if zero to `lab`). The exception would seem to be the `SLT` instruction, and it is true that it doesn't work with floats, but it's close. In fact, thanks to IEEE 754 encoding, comparing two positive floats using the integer `SLT` instruction works as you'd hope -- it's just when one or both are negative that we have an issue. Of course, there is also nothing wrong with using integer operations to manipulate floating-point bit patterns; for example, `EOR 1,#0x8000` is essentially a floating-point negate of register 1 (although it does make 0 into -0) and `ADD 1,#0x0080` adds one to the exponent, thus multiplying a floating-point value by 2 (provided the exponent stays in range). The point is, you don't have to worry about any of this because they're already implemented -- **just the six floating-point instructions are new stuff to implement**.

Our Mutant Floating Point Format

Remember IEEE 754? The latest version of this standard (2008) is only 70 pages long. On the UK campus, you can get IEEE Std 754-2008 for free from this IEEE Xplore site. Conforming to the standard is not easy, and you will not demonstrate conformance -- after all, I do want you folks to survive this project. So, we'll be simplifying things quite a bit. What will be non-conforming? Well, to being with, we're using a 16-bit binary layout that is essentially what the standard calls **binary32** format, but missing the 16 least significant bits of the mantissa. *That is not the same as the binary16 format*, although it is very similar to what some GPUs have implemented. We're also not going to be very careful about the arithmetic, producing (disturbingly) approximate answers. The only subnormal value you'll deal with is positive 0, and we'll also ignore +/- infinity, NaN (both the quiet and signaling types), and rounding modes.

There is a lovely little formula on page 9 of the standard that says a floating point value is:

$$(-1)^S * 2^{E-bias} * (1 + 2^{1-p} * T)$$

Described in IEEE 754-2008 terms, we'll represent that as:

S: sign, 1 bit	E: encoded exponent, 8 bits	T: trailing part of significant, 7 bits

- S refers to the sign bit
- E is the value in the 8-bit exponent field
- The bias is 127 (which is also the maximum exponent value)
- The significand precision, p, is 8 (not 24, as it would be for binary32)
- The trailing significand, T, is the 7-bit pattern that logically comes immediately after the binary point

That's slightly confusing. Think of it this way: what Verilog would call `{1'b1, T}` as an unsigned integer, gives the value:

$$(-1)^S * 2^{E-bias-7} * \{1'b1, T\}$$

For example, the decimal value 3 is `(-1)^0 * 2^128-127-7 * 8'b11000000`, which is `1 * 2^-6 * 192`, which is 192/64. Thus, the encoding of 3.0 is `0x4040`.

The really cool thing about using this odd format is that you can play with it using ordinary binary32 floating-point math, as implemented by langauges like C, and simply ignore the last 16-bits of the binary32 result. The good news is that Verilog also knows about binary32 values as `real`, and provides built-in functions `$bitstoreal()` and `$realtobits()`. The bad news is that Icarus Verilog doesn't implement either one of those functions. Oh well. They aren't supposed to be synthesizable anyway. Oh, and did I mention that **aik doesn't understand floating point numbers either**? Well, it doesn't.

Fortunately, I built a little CGI form to play with math and conversions to/from our mutant 16-bit float format. Beyond allowing you to do some arithmetic on our mutant 16-bit floats and see the results, it also shows you what the integer value is that encodes each floating-point value. Enjoy. Incidentally, this format really brings home the imprecision of using floating point. For example, adding 10 and 0.1 results in 10 + 0.0996094 = 10.0625. Really? Yup. By the way, 100 + 1 = 101. For the repeating fraction generated by 0.1, truncation is not your friend. ;-)

You'll probably want to refer back to my slides on floating point for some insights on how to do each of these operations. Perhaps surprisingly, `addf` is the hardest one to implement. However, I'd strongly suggest starting with `F2I` and `I2F`. Why? Because they will make testing a lot less painful.

All The World's A Stage... And Do We Need More Of 'Em?

Of course, you should start with the pipeline design from the previous project. However, there is more to this. Think about what stages you should have and state and justify your decisions in the implementor's notes. In particular:

- The denormalization in addition and subtraction, and normalization of results, might best be handled by new stages... or maybe not?
- The reciprocal operation might use a 128-entry lookup table. Is that another stage?

I will suggest that the techniques described for implementing a barrel shifter or count-leading-zero operation are fast enough to be done within a single clock cycle, and could even be reasonably paired with another fast operation within one cycle. That said, there is more than one valid way to structure the pipeline to handle floating-point arithmetic -- just make sure to say something about what you did in the implementor's notes.

Due Dates

The recommended due date is Monday, December 3, 2018. By that time, you should definitely have at least submitted something that includes the assembler specification (`pinky.aik`), and Implementor's Notes including an overview of the structure of your intended design. That overview could be in the form of a diagram, or it could be a list of top-level modules, but it is important in that it ensures you are on the right track. Final submissions will be accepted up to just before the final exam on Tuesday, December 11, 2018.

Note that you can ensure that you get at least half credit for this project by simply submitting a tar of an "implementor's notes" document explaining that your project doesn't work because you have not done it yet. Given that, perhaps

you should start by immediately making and submitting your implementor's notes document? (I would!)

Submission Procedure

For each project, you will be submitting a tarball (i.e., a file with the name ending in `.tar` or `.tgz`) that contains all things relevant to your work on the project. Minimally, each project tarball includes the source code for the project and a semi-formal "implementors notes" document as a PDF named **`notes.pdf`**. It also may include test cases, sample output, a make file, etc., but should not include any files that are built by your Makefile (e.g., no binary executables). Be sure to make it obvious which files are which; for example, if the Verilog source file isn't **`pinky.v`** or the AIK file isn't **`pinky.aik`**, you should be saying where these things are in your implementor's notes.

Submit your tarball below. The file can be either an ordinary `.tar` file created using `tar cvf` *`file`*`.tar` *`yourprojectfiles`* or a compressed `.tgz` file file created using `tar zcvf` *`file`*`.tgz` *`yourprojectfiles`*. Be careful about using `*` as a shorthand in listing *`yourprojectfiles`* on the command line, because if the output tar file is listed in the expansion, the result can be an infinite file (which is not ok).

Your team name is .
Your password is

Advanced Computer Architecture.