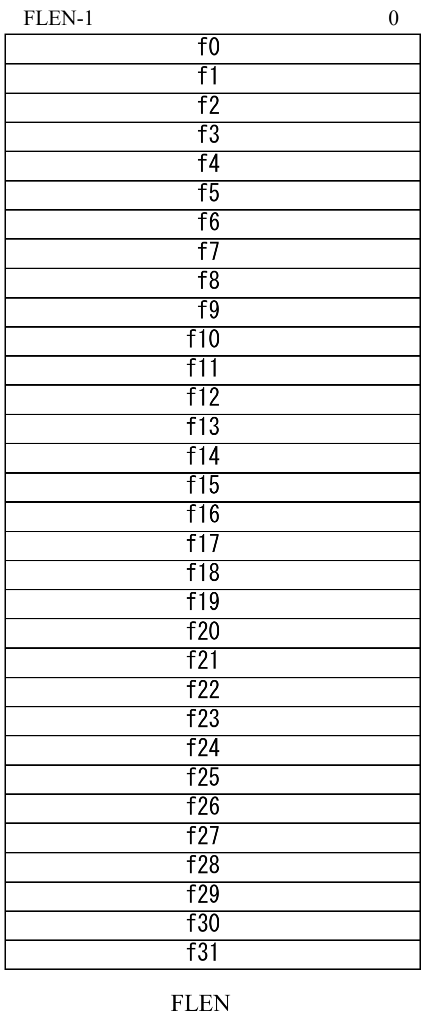
Computer Organization – Verilog – HW2

Email: iustCompOrg+4012@gmail.com

In this assignment you are tasked to create simple fixed point circuitry to be later used as an integrated FPU (Fixed/Floating Point Unit) in your RISC-V core. Any RISC-V processor implementation must support a base integer ISA (In our case, RV32I). In addition, an implementation may support one or more extensions. The standard instruction-set extension for single-precision floating-point, which is named “F", adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard.



But in this problem, you are going to create a Fixed-Point Unit and related circuitry to make your RISC-V core capable of running the `FADD` and `FSQRT` instructions rather approximately. (i.e. It will be less precise than single precision floating point)

As you know a RISC-V core includes a register file with 32 registers each having the length of XLEN (XLEN = 32 for RV32I). RISC-V standard specifies a new register file again with 32 registers each having the length of FLEN (FLEN = 32 for RV32IF). This new registers will be exclusively used for “F” extension instructions and will not be treated as having integer values.

RV32F standard extension instructions that are required to be implemented in your RISC-V core:

`FLW, FSW, FADD, FSQRT`

31-25	24-20	19-15	14-12	11-7	6-0	
imm[11:0]		rs1	010	rd	0000111	FLW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	FSW
0000000	rs2	rs1	rm	rd	1010011	FADD
0101100	00000	rs1	rm	rd	1010011	FSQRT

Fixed Point Numbers:

With decimal numbers, we're used to the idea of using a decimal separator such as a point or a comma to separate integer and fractional parts. We can do the same thing in binary and use the bits to represent any powers of two we want.

For example, think of 6.75 as being $4 + 2 + \frac{1}{2} + \frac{1}{4}$.

In binary this can be shown as:

8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
0	1	1	0	1	1	0	0

In this style we can separate the fractional part with a point. We are choosing to interpret the values as being fixed point, but from the hardware logic point of view it's just an 8-bit integer. If we comply with a rule that the positon of the fixed point is consistent, we'll get the expected result from mathematical operations.

Q Notation

To express the number of integer and fractional bits we use Q number format: $Qi.f$ where i is the number of integer bits and f is the number of fractional bits. For example 0110.1100 has four integer and four fractional bits, so is Q4.4.

All the usual binary math yields correct results when used with fixed-point numbers. Addition works in the same way as for integers:

```
        0011.1010        3.6250

+       0100.0001      + 4.0625

=       0111.1011      = 7.6875
```

We can also do subtraction using two's complement to express negative numbers:

```
        0011.1010        3.6250

+       1110.1000      - 1.5000

=       0010.0010      = 2.1250
```

Range and Precision

Using two's complement, a 4-bit value ranges from -8 to +7. $[1000 \rightarrow 0111]$

A 4-bit fraction can represent numbers as small as $\frac{1}{16}$ and as large as $\frac{15}{16}$. $[0001 \rightarrow 1111]$

Therefore, Q4.4 notation ranges from -8 to +7.9375 with a precision of 0.0625.

- **Problem 1)** Find the range and precision of Q22.10.

Converting to and from Integers

Conversion to and from regular binary integers simply requires using the appropriate left or right shift. For example, if we're using Q16.16 format we need to left-shift an integer 16 positions to create the Q16.16 fixed-point number:

```
1000101                 → Decimal 69
 << 16                  → Left shift 16 positions

        100 0101 0000 0000 0000 0000

     Showing all bits in Q16.16 notation:

 0000 0000 0100 0101.0000 0000 0000 0000
```

Now we can perform all arithmetic operations on this number with other Q16.16 numbers. For example, let's add decimal 14.84375 to this number:

```
     0000 0000 0100 0101.0000 0000 0000 0000          69.0

+    0000 0000 0000 1110.1101 1000 0000 0000     +    14.84375

=    0000 0000 0101 0011.1101 1000 0000 0000     =    83.84375
```

If you need to convert a Q16.16 number back to a regular integer, you have to right-shift 16 positions:

```
0000 0000 0101 0011.1101 1000 0000 0000   → Decimal 83.84375
                  >> 16                   → Right shift 16 positions

                  0000 0000 0101 0011

             Showing all bits in Q16.16 notation:

             0000 0000 0100 0101.0000 0000 0000 0000
```

Using the fixed-point Q22.10 notation you don't need to include additional FPU hardware for your RISC-V core to be capable of executing `FADD`, `FSW`, `FLW`. But the control unit needs to be altered to accommodate these new instructions. In the next section you are going to learn about square root calculation and hardware implementation.

Square Root Calculation

In this part you are going to design an Algorithmic State Machine (ASM) to calculate the square of integer and fixed-point numbers. Although lower-latency methods exist, the method explained here includes a straightforward digit-by-digit approach, using only subtraction and bit shifts. Most square roots are irrational, so the algorithm would result only an approximate answer. The number we're finding the root of is known as the *radicand*. To explain this algorithm, consider the following example:

Consider the radicand to be `10101001` (169 in decimal).

This algorithm works with pairs of digits. Before calculations, radicand is split into pairs starting with the least significant. Our radicand here becomes `10_10_10_01`.

```
10   10   10   01                      Our radicand
↓    ↓    ↓    ↓
10                         Bring down the most significant pair
01                                     Subtract 01
--
01                The answer is NOT negative so the first digit is 1

                  Bring down the next pair of digits and append to previous
01   10                                   result
01   01              Append 01 to the existing answer, 1, and subtract
-------
00   01                 The result is NOT negative so next digit is 1

                  Bring down the next pair of digits and append to previous
00   01   10                              result
     11   01        Append 01 to the existing answer, 11, and subtract
-------------
     10   01            The result is negative so the next digit is 0
                         We discard the result because it's negative.
     01   10   01   Keep the existing digits, 0110, and append the next pair
     01   10   01      Append 01 to the existing answer, 110, and subtract
     ------------
     00   00   00       The result is NOT negative so next digit is 1
```

There are no more pairs of digits, and the result of our last step is 0, so our answer is exact `1101` (13 in decimal):

```
1 1 0 1 → One digit for each pair of digits in the radicand
```

- **Problem 2)** Design the module, `SQRT_int`, in *Verilog* based on the algorithm above. Write a *testbench* for your module and check the results.

Now that you are familiar square root calculation, you are going to add support for fixed-point calculation. Supporting fixed-point square roots is straightforward: we just run more iterations to account for the fractional digits. Unlike with division, there's no possibility of overflow, as the root of the radicand greater than 1 is always smaller that the radicand itself.

Your new version of `sqrt` module needs a new parameter *FBITS* for the fractional bits in the radicand. Using the Q22.10 format parameters will be set as follows:

```
WIDTH = 32

FBIT3S = 10
```

For this new radicand you need to perform $\frac{32+10}{2}$ iterations.

- **Problem 3)** Design the module, `SQRT`, in *Verilog* which gets an *integer* input and return a *fixed-point* value. Write a *testbench* for your module and check the results.

Notes:

Send all your assignment related files (*Top module* and *Testbench* [.v] and [.vvp] and [.vcd] files) along with a detailed report in [.pdf] format all in one zip file to the email address of the class.

For this assignment you can work in groups of two, each participant must submit the assignment individually with their respective name and student number.

If you have any questions regarding this assignment, feel free to contact us.

Please submit your homework, simulations and projects in the following format:

Name_StudentNumber_Verilog_HW2 (BillGates_12345678_Verilog_HW2)

Good Luck!