Floating Point Unit
Digital IC project 2001

Martin Kasprzyk, e00mk

January 19, 2002

Contents

1 Introduction

The goal of this project was to implement a FPU (**F**loating **P**oint **U**nit) for the Leon processor. The Leon VHDL model implements a processor conforming to the Sparc V8 architecture and is available for free on the net (www.gaisler.com). For now the FPU should only have addition, subtraction and compare instruction implemented and should be 100% IEEE754 compliant. In the report there will be references to this standard and what it requires of the hardware.

We will start of looking at floating point numbers followed by a quick overview of what is needed to know about the Leon processor. Then we will go into the implementation of the FPU.

The first step in the design process was to examine the FPU interface using a dummy FPU. After that the FPU was developed and tested stand alone using a testbench that simulated the FPU interface of the Leon processor. When the FPU was working as it should, design analyzer was used to test if the code was synthesizable. Finally the FPU was simulated together with the Leon processor using the testbench that came with the Leon VHDL model.

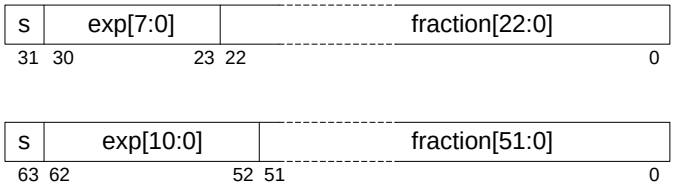

Figure 1: Floating-point formats

2 Floating-point numbers

In a floating point system with radix 2 and sign and magnitude representation, floating-point numbers are represented as:

$$(-1)^S * F * 2^E$$

Where S is the sign bit, F is the fraction and E the exponent. The factor 2^E specifies the position of the radix point with respect to the digits of F. So by changing the exponent we move the radix point. The natural numbers $31.42{*}10^2$ and $3.142 * 10^3$ represent the same number but have the radix point at different position from this come the name floating-point.

When working with floating-point numbers that have a fraction with limited bit width it is important to eliminate as many non-significant digits as possible, e.g. use 0.314 instead of 0.00314, to have maximal precision.

A significant is said to be normalized if it is within permitted range and has no leading zeros, otherwise it is unnormalized. There are two sorts of unnormalized types, supernormal and subnormal. A supernormal fraction is unnormalized because its magnitude is to large and a subnormal fraction because it has leading zeros. Normalizing a subnormal value involves shifting the fraction left and decreasing the exponent. Normalizing a supernormal value involves shifting the fraction right and increasing the exponent.

Since normalized values always have a leading one, that bit does not have to be stored (called hidden bit) and by doing this we gain one extra bit. It is also convenient if a zero could be represented with only zeros and not having a signed exponent. This is achieved by storing a biased exponent instead of the real one. These two methods are used in the IEEE754 standard where a floating-point number is represented as:

$$(-1)^S * (1 + F) * 2^{(E-Bias)}$$

There are two different precisions for floating-point numbers that we will use, single and double precision. These two data formats are shown in figure 1 and their definitions are shown in table 1.

2.1 Addition and subtraction

Let us look at how addition and subtraction is performed with floating point numbers since we have to implement these two operations in hardware. First

normalized value ($0 < exp < 255$)	$(-1)^s * 2^{exp-127} * 1.fraction$
subnormal value ($exp = 0$)	$(-1)^s * 2^{-127} * 0.fraction$
signaling NaN	s=u exp=255 fraction=.0uu...uu (at least one bit of fraction must be nonzero)
quiet NaN	s=u exp=255 fraction=.1uu...uu
$-\infty$	s=1 exp=255 fraction=.000...00
$+\infty$	s=0 exp=255 fraction=.000...00

normalized value ($0 < exp < 2047$)	$(-1)^s * 2^{exp-1023} * 1.fraction$
subnormal value ($exp = 0$)	$(-1)^s * 2^{-1023} * 0.fraction$
signaling NaN	s=u exp=2047 fraction=.0uu...uu (at least one bit of fraction must be nonzero)
quiet NaN	s=u exp=2047 fraction=.1uu...uu
$-\infty$	s=1 exp=2047 fraction=.000...00
$+\infty$	s=0 exp=2047 fraction=.000...00

Table 1: Floating-point single and double format definitions

we must adjust the operands to make the exponents equal i.e. have the radix points aligned. We do this by shifting the smaller operands fraction right by the magnitude of the exponent difference steps. This operation is called pre-arithmetic shift.

Next the fractions are added or subtracted. The resulting fraction can be subnormal or supernormal. If the result is supernormal the the overflow bit is a one and the fraction must be shifted one step right and the exponent increased by one. In the other case the leading ones must be counted (or predicted) and the fraction must be shifted left by the amount of leading zeros and the exponent is decreased by the same amount. This operation is usually called post-arithmetic shift or postnormalization shift.

Next rounding is performed, in which a one can be added to the retained bits or the bits outside the precision simply get chopped away. In the case of adding a one, the operation can result in a supernormal value and then the fraction must be shifted by one step right and the exponent increased by one.

2.2 Overflow and underflow

The range of numbers that can be represented using floating point representation is shown in figure 2. Spill occurs when a computation leads to an exponent that is too large or to small to represent. Exponent overflow occurs when the exponent of a result is larger then the largest permissible exponent, i.e. number is outside X_{max} or Y_{max}. Exponent underflow occurs when the exponent of the result is smaller then the smallest permissible exponent, i.e. number smaller the X_{min} or Y_{min}.

When an underflow occurs the easiest solution would be to set the result to zero, but the standard specifies use of gradual underflow instead. The result

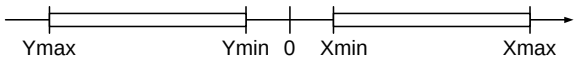

Figure 2: Range of representable floating-point numbers

will then be denormalized (subnormal). By using this method X_{min} and Y_{min} will move closer to zero.

When an overflow occurs the result is set plus infinity (∞) or minus infinity depending on the sign of the result. Infinity is represented by a special bit pattern and has the following properties.

$$-\infty < x$$

$$x < \infty$$

$$\infty + x = \infty$$

$$\infty - x = \infty$$

There are some operations where it is not clear what the result should be e.g. $\infty - \infty$. The result could be 0 or $\pm\infty$. Since the result is undefined, another bit pattern known as NaN (Not-a-Number) will result. The standard specifies two sorts of NaNs: signalling and quiet. The SNan (signalling NaN) should cause an exception.

2.3 Rounding

Let us start by define a name for the value of the least significant bit in the result for any precision as an *ulp* (**u**nit in the **l**east **p**osition). This definition will be useful when were going to explain the four different rounding methods that the IEEE754 requires. The rounding mode is selected by setting the RD bits is the floating-point state register.

The simplest rounding method is to discard the bits outside the used precision. This method is called chopping or round-towards-zero.

A better and more used method is round-to-nearest-value (even). If the discarded bits magnitude is larger then one half ulp then one is added to the retained digits. If the magnitude is smaller then one half ulp then chopping is performed. In the case when the discarded bits are equal to one half ulp then we will round to the nearest value that is even (least significant bit is zero).

For round-to-plus-infinity if the discarded bits magnitude is one half ulp or larger the one is added to the retained bits otherwise chop. This is reversed for round-to-minus-infinity.

3 Leon processor

Not much knowledge about the Leon processor was needed since the interface is well defined. For more information read [1] and [2].

3.1 FPU interface

Now let us look at the co-processor interface that will be used by the FPU. The interface allows the FPU to operate in parallel with the processors integer unit to increase performance. The FPU interface is enabled by setting of the FPU element of the configuration record in `target.vhd`. The FPU type should be `fpu_fpc`. When the FPU is selected the testbench will also run FPU testcase.

The FPU is connected using the following two records.

```
type cp_unit_in_type is record                -- FPU input
  op1      : std_logic_vector (63 downto 0); -- operand 1
  op2      : std_logic_vector (63 downto 0); -- operand 2
  opcode   : std_logic_vector (9 downto 0);  -- opcode
  start    : std_logic;                      -- start
  load     : std_logic;                      -- load operands
  flush    : std_logic;                      -- cancel operation
end record;

type cp_unit_out_type is record               -- Fpu output
  res      : std_logic_vector (63 downto 0); -- result
  cc       : std_logic_vector (1 downto 0);  -- condition codes
  exc      : std_logic_vector (5 downto 0);  -- exception
  busy     : std_logic;                      -- eu busy
end record;
```

The waveform diagram for the FPU interface can be seen in figure 3. First the start signal is asserted together with a valid opcode, then on the next clock cycle load signal is asserted together with the operands. Since the instructions will take more then one clock cycle the busy signal must be asserted and kept high until the cycle before the result is ready. The result, exceptions and condition codes must be valid from the cycle after the de-assertion of the busy signal until the next assertion of the start signal. The FPU can anytime be reseted to idle condition be asserting the flush signal for two clock cycles.

3.2 Condition codes and exceptions

Compare instructions are the only instructions that affect the condition codes. The meaning of the condition codes are explained in table 2. There are two different sort of compare instruction and the difference between them is if an unordered result (if at least one of the operands is a NaN or a SNaN) should result in a not valid operand exception or not.

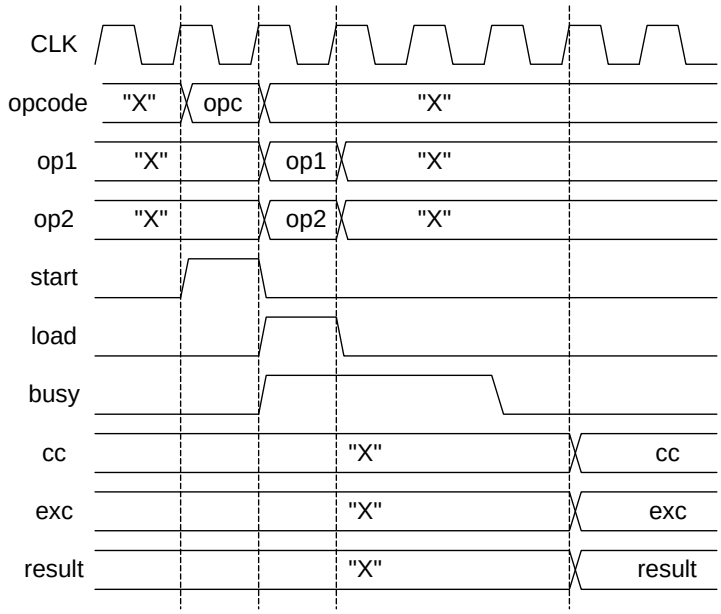

Figure 3: Instruction execution waveform diagram

fcc	Relation
0	$op1 = op2$
1	$op1 < op2$
2	$op1 > op2$
3	$op1 \,?\, op2$ (unordered)

Table 2: Floating-point condition codes

NV	OF	UF	DZ	NX

NV An operand is improper for the operation to be performed e.g. $\infty - \infty$. 1 = invalid operand, 0 = valid operand(s).

OF The rounded result would be larger in magnitude than the largest normalized number in the specified format. 1 = overflow, 0 = no overflow.

UF The rounded result is inexact and would be smaller in magnitud than the smallest normalized number in specified format. 1 = underflow, 0 = no underflow.

DZ X$\div$0, where X is subnormal or normalized (but not zero). 1 = division-by-zero, 0 = no division-by-zero.

NX The rounded result of an operation differs from the infinitely precise result. 1 = inexact result, 0 = exact result.

Table 3: Floating-point exceptions

The different exceptions are explained in table 3. At this moment the division-by-zero exception is not used since division instructions are not implemented. Compare instructions can never cause overflow, underflow or inexact exceptions since they have no meaning when you compare size of two operands. The invalid operand exception is also generated if the result is a SNaN, which means that at least one of the operands was a SNaN while performing a none compare instruction.

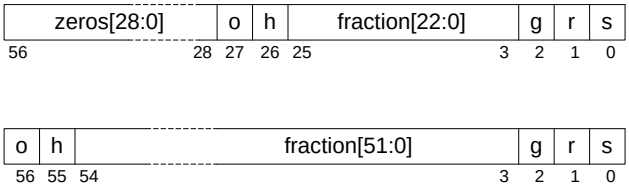

Figure 4: Internal floating-point format

4 Implementation

The coding style was inspired by the integer unit code (`iu.vhd`), where every pipeline stage was a separat process using records to pass values between each other. Procedures implementing variable amount left and right shift and the leading zero counter are found att the beginning of the architecture body.

The goal was to implement a working FPU and because of the short time to complete the project no optimization for speed has been done. The code is written using a rather high-level VHDL as the Leon processor.

4.1 Internal data format

Since the FPU should be able to perform operations on both single and double precision operands the internal format must be able to handle both. The internal exponent format used is double precision format. For the fraction we also use double precision but we append some extra bits as shown in figure 4.

The hidden bit (H) is retrieved and appended to the fraction. In front of it is the overflow bit (O) that is used to detect supernormal results. Note that for the two different precisions these two bits are at different locations. We also append the guard (G), round (R) and sticky (S) bit. These bits give us some extra precision and are also needed for rounding.

For the larger operand theses bits are set to zero. For the smaller operand that is shifted right during pre-arithmetic shifting the guard and round bits give some extra bit positions to hold the shifted value. The sticky bit is calculated by performing an OR operation on the bits that got shifted out.

4.2 Shift registers

Variable amount shift registers are implemented as a series of muxes. The input vector goes through a mux that either shifts the vector 32 position or lets it pass unchanged depending on the shift amount. Then the output from that mux goes through a mux that either results in a 16 step shift or lets it pass unchanged, and so on. The final mux can shift the data one step giving us a possibility to shift 0 to 63 positions. The pre-arithmetic right shifter also has some extra logic to calculate the sticky bit.

4.3 Leading zero counter

The leading zero hardware starts of by finding the leading one and setting the rest of the bits to zero. Then the vector is decoded by performing an OR operation on some predefined bit positions in the vector. For every bit in the amount there are different bit positions that are used in this operation.

4.4 Overview

Let us now take a look at what operation the FPU performs. Figure 5 shows the organization of the FPU.

In the first pipeline stage the operands are unpacked depending on precision and checked if they are special numbers i.e. NaN:s. The opcode is examined and control signals are set to perform selected operation.

Next pipeline stage is the pre normalization stage. The exponent difference is calculated and the smaller operand is shifted right by that amount. Hidden bits are retrieved, the sticky bit is calculated and the rest of the extra bits are appended to the operands.

Next addition or subtraction takes place and a test if the operation should result in a special value performed.

In the the post normalization stage the result is shifted right and the exponent is increased by one if the result is supernormal. Otherwise leading zeros are counted and the result is shifted left that many steps and the exponent is decremented by that amount. Tests if the normalization results in underflow or overflow are performed.

In the last pipeline stage rounding is performed and if rounding results in a supernormal result it is normalized by shifting one step right and increasing the exponent by one. A test if overflow has occurred is performed. Fraction and exponent for special results are generated in this stage. Finally the result packed.

4.5 Conclusions

The floating point unit was tested stand alone using a testbench that simulated the FPU interface of the LEON processor. Operands were selected so that the entire FPU functionality was tested. When the FPU was working as it should, design analyzer was used to test if the code was synthesizable. Finally the FPU was tested with the Leon processor.

For now only addition, subtraction and compare instructions are implemented, so the next step would be to add the rest of the instructions. Also some optimizations for speed could be done in the future.

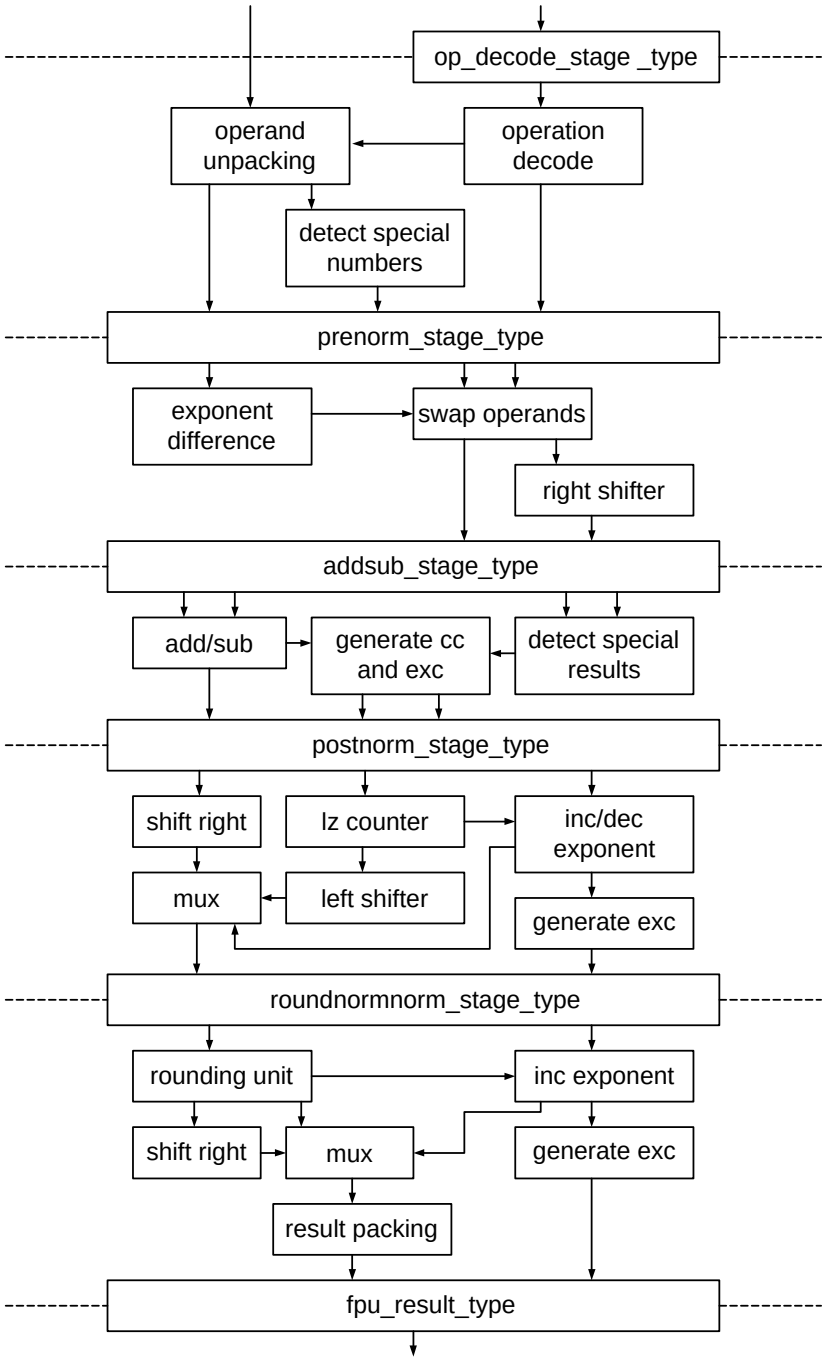

Figure 5: Floating-point unit overview

References

[1] Jiri Gaisler. *The LEON Processor User's Manual, version 2.3.5.* Gaisler Research, July 2001.

[2] SPARC International Inc. *The SPARC Architecture Manual, Version 8.*