Master Thesis

FPGA-based vector floating-point unit with software-implemented division

Author(s):
Tecu, Dan

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006180614

Rights / License:
In Copyright - Non-Commercial Use Permitted
FPGA-Based Vector Floating-Point Unit with Software-Implemented Division

Master Thesis

Author: Dan Tecu

Professor: Prof. Dr. J. Gutknecht
Advisors: Dr. Ling Liu
Dr. Felix Friedrich

Department of Computer Science, ETH Zurich

March 31st, 2010 – September 30th, 2010
Acknowledgements

I would like to thank my advisors: Dr. Ling Liu, Dr. Felix Friedrich and Prof. Dr. J. Gutknecht for their help, support and for the opportunity to work in this group and design the things I always wanted to. I would like to thank them for trusting and considering my work; for validating it into the “real world”!

I am grateful to Ling for her promptness to answer my questions! She always interrupted her current tasks to help me when I needed it!

I would also like to take this opportunity to thank Ms. Judith Zimmermann for her absolutely great support during my studies!

Last but not least, I’d like to thank Andrea, for her help, support and understanding!

Dan Tecu

Zurich, September 28th, 2010
Contents

CONTENTS..........................................................................................................................................................3

1 INTRODUCTION..................................................................................................................................................4
  1.1 Motivation......................................................................................................................................................4
  1.2 Related Work................................................................................................................................................4
  1.3 The IEEE-754 Standard............................................................................................................................4

2 VECTOR PROCESSOR INTRODUCTION......................................................................................................6

3 THE VECTOR FLOATING POINT UNIT.........................................................................................................7
  3.1 Features.......................................................................................................................................................7
  3.2 Components................................................................................................................................................8
  3.3 Design Methodology..................................................................................................................................8
    3.3.1 General Organization..........................................................................................................................8
    3.3.2 Strategy.................................................................................................................................................9
  3.4 The Scalar Floating Point Addition/Subtraction Unit..............................................................................13
    3.4.1 The Interface and Block Diagram.......................................................................................................13
    3.4.2 Algorithm Description.......................................................................................................................14
    3.4.3 FPGA resources................................................................................................................................18
  3.5 The Scalar Floating Point Multiplication Unit.......................................................................................18
    3.5.1 The Interface and Block Diagram.......................................................................................................18
    3.5.2 Algorithm Description.......................................................................................................................19
    3.5.3 FPGA resources................................................................................................................................20
  3.6 The Vector Floating Point Horizontal Addition Unit...........................................................................21
    3.6.1 The Interface......................................................................................................................................21
    3.6.2 Performance......................................................................................................................................21
    3.6.3 Description......................................................................................................................................22
    3.6.4 FPGA resources................................................................................................................................24
  3.7 Exceptions................................................................................................................................................24

4 VERIFICATION FRAMEWORK..................................................................................................................25
  4.1 Block-Level Verification............................................................................................................................25
  4.2 Top-Level Verification................................................................................................................................25

5 FLOATING POINT DIVISION – SOFTWARE IMPLEMENTATION..................................................................28
  5.1 The Algorithm............................................................................................................................................28
  5.2 The Implementation..................................................................................................................................30

6 CONCLUSION..................................................................................................................................................33

7 REFERENCES....................................................................................................................................................34
1 Introduction

1.1 Motivation

Intensive mathematical computations are very common in medical applications based on signal processing. In order to make this possible, a reasonable speed and performance is required from the underlying hardware.

To achieve this goal, a vector processor has been constructed by extending the TRM processor. Of course, a vector architecture will definitely improve the performance of our processor by SIMD parallelism. However, in the signal processing applications only an integer based architecture is not enough. We must provide an accelerated way to run floating point computations.

The purpose of the current thesis is implementation of a vector floating point extension to our TRM based architecture. The implementation will be described in detail throughout this document. The instruction set and other details of architecture can be found in: [Liu].

1.2 Related Work

Beginning with the Xilinx XC4085XL, it became possible to implement FP addition and multiplication on FPGA. Since then, the architecture of FPGAs has become more heterogeneous, with more built-in silicon slices – DSPs and BRAMs. The architecture improvements make FPGAs have the potential to provide more FP computing power [Lienhar, Nichols, Roesler, Govindu].

However, most existing FP work on FPGAs does not target modern FPGAs, such as the Virtex-5 used in this thesis. Furthermore, most FP implementation on FPGA focus on the scalar FP addition and multiplication. To the author’s knowledge, there is no Vector FP implementation, especially the horizontal addition, on Virtex-5 reported. Different from the exist work, this thesis focus on the vector FP unit implementation on modern FPGA – Xilinx Virtex-5. The implementation results presented in this thesis shows that it is feasible to achieve high-performance vector FPU on Virtex-5.

1.3 The IEEE-754 Standard

This section briefly describes the IEEE Standard for Binary Floating-Point Arithmetic. The exact description can be found in: [IEEE-754].
The standard describes the format and requirements for the binary floating point arithmetic.

This standard defines four floating point formats in two groups, basic and extended. For each, we have two widths: single and double. Numbers are described by three fields (p represents the precision - the number of bits in the fraction):

- 1 bit sign: s
- Biased exponent: e = E + Bias
- Fraction (Significand): f = b₁b₂…bₚ⁻¹

The value of the number expressed with the above 3 parameters is given depending on the format the number is described in. By example, for the single basic format, the value of a number v, represented by s, e and f is shown below:

- if e == 255 and f != 0 then v is NaN (not a number)
- if e == 255 and f == 0 then v = (-1)ˢ * ∞ (∓∞)
- if 0 < e < 255 then v = (-1)ˢ * 2ₑ⁻¹₂₇ * (1.f) (normalized numbers; Bias is 127)
- if e == 0 and f != 0 then v = (-1)ˢ * 2⁻¹₂₆ * (0.f) (denormalized numbers)
- if e == 0 and f == 0 then v = (-1)ˢ * 0 = 0

The representation of the number in the single basic binary format (s, e and f) is:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGN</td>
<td>EXPONENT</td>
<td></td>
<td>SIGNIFICAND</td>
<td></td>
</tr>
</tbody>
</table>

As we’ve seen the set of values includes besides the “arithmetic” numbers, two infinities (+∞ and -∞) and “not a number” (NaN). Infinities could be useful in representing the result of a division by zero. NaN could be useful to represent the result of a negative number square root.

We will describe also another features of the standard, as they become important. Later we will describe the rounding modes and our decisions regarding IEEE-754 standard.
2 Vector Processor Introduction

To exploit the DLP in our target applications, ultra-sound image processing applications, a vector processor is designed and implemented. The vector processor VTRM consists of a scalar processor TRM and a vector processing unit TRMVector. The TRM and TRMVector are both fed by one instruction stream from the on-chip local instruction memory. Both cores share the same on-chip local data memory. The VTRM processor has 8 scalar registers and 8 vector registers that are 8 words wide. Each word is 32 bits. VTRM has 4K instruction memory and 8K data memory. Each instruction of VTRM is 18 bits [Liu]. The VTRM is implementation with two-stage pipelines. The first stage is for fetching instruction from 4k instruction memory (IM). The second stage performs instruction decoding and ALU operations or memory accesses. Following figure shows the VTRM architecture with each pipeline stage separated by black pipeline registers. The two-stage pipeline implementation avoid read / write hazard, therefore there is no hazard detection circuit. It takes one DSP slice to implement multiplication between two 32-bit integers, 8 DSP slices to implement multiplication between two vectors, each vector is 8 words wide, and 7 DSP slices to implement horizontal addition of vector elements. The VTRM runs at 116MHz. Every scalar or vector multiplication takes 6 cycles. Every LD instruction takes 2 cycles. Every horizontal addition instruction takes 4 cycles. All these happen even in the presence of dependancies.

VTRM architecture with 2 pipeline stages. The solid rectangular boxes are pipeline registers, the circles indicate logic, the boxes indicate storage elements.
3 The Vector Floating Point Unit

3.1 Features

The IEEE-754 standard does not require floating point to be fully implemented either in hardware or in software. Functionality can be split between software and hardware according to the current needs and optimization strategy.

Our current Vector FPU implements in hardware the following IEEE-754 features:

- Single precision (32 bits)
- Normalized significands
- Round to nearest even rounding mode
- Overflow and underflow exceptions

The remaining IEEE-745 features not implemented in hardware are:
1. Double precision (64 bits mode)
2. Extended precision
3. Round to infinity (plus/minus) and round to zero
4. Un-normalized significands
5. Inexact exception

The first two hardware-unimplemented features (1 and 2) are not proper for an FPGA implementation. They consume too much of the FPGA resources. They are also not well suited for our 32 bit registers. Regarding the last three features (3,4 and 5): they are not really used in reality. Experts on intensive mathematical computations declare that they have never used those extra rounding modes and that they do not miss the un-normalized significands. These two small simplifications bring us many advantages in implementation. As we will see later, they help us reduce the size of the adder (used for significands addition) from 52 to 28 bits. This further helps us gaining speed, reducing the area and power consumption. Please notice that a simplification in one floating point adder is significant (must be multiplied by a factor of 15), due to the fact that we use 15 floating point adders (8 for the vector floating point addition and 7 for the horizontal floating point addition).

If the inexact exceptions were implemented, they would almost never be used by the software. The inexact exception is supposed to be raised when the result needs actually more than 32 bits to be represented exactly (so, when some digits in 1 are being dropped during the rounding operation, the inexact exception must be raised). This happens very, very often. Except additions with 0, all other additions need at least one right shift. In conclusion, all normalized additions loose digits. As almost all the software products show, the program is not willing to interrupt or abort execution because an inexact exception has been detected. Getting an inexact result is expected when working with a 32 of 64 bits FPU (or even more). Applications that cannot accept inexact results usually use a software library implementing an arbitrary precision for floating point operations. That's
actually the reason why the inexact exception is almost never used in practice - we cannot expect getting exact results from the floating point hardware.

3.2 Components

The Vector Floating Point Unit is composed of the following units:

1. Vector floating point addition/subtraction unit containing 8 scalar floating point addition/subtraction units
2. Vector floating point multiplication unit containing 8 scalar floating point multiplication units
3. Vector horizontal floating point addition unit which contains 7 scalar floating point addition/subtraction units

I will describe each of these units in detail. The vector addition/subtraction and multiplication units, just instantiate 8 times the same unit: scalar addition/subtraction respectively multiplication. Consequently, the discussion on them reduces to a discussion on the scalar floating point addition/subtraction unit respectively the scalar floating point multiplication unit. On the other hand, the vector horizontal adder will require more attention than a simple scalar floating point adder.

However, before describing in detail each of these units, I have to describe first the design methodology used to achieve the performance – which was one of the key requirements.

3.3 Design Methodology

3.3.1 General Organization

There are no hard-coded numbers in the design. All the “numbers” are specified by using the defined constants in the include file: “fp_definitions.v”. The constants in there were prefixed with the “FP” prefix, to avoid conflict with constants in other files (Verilog has no package concept, all the entities live in the same pool, so conflicts are possible).

The units were designed to be scalable and generic as much as possible. The only exceptions from this rule are the right and the left shifter units used in the floating point addition/subtraction unit. If I tried to make these two units generic we would have lost speed and area. In these two cases it was important to exploit the “non-power of two” shifting quantities – please see more explanations under the description of the floating point addition/subtraction unit.

All the design files required for the vector floating point unit are in the directory FPU. This directory contains five subdirectories:
• fp_include – containing files defining constants, functions and settings used all over the vector floating point unit.
• fp_adder – containing all the design sub-units used inside the floating point addition/subtraction unit.
• fp_multiplier – containing all the design sub-units used inside the floating point multiplication unit.
• fp_horizontal_adder – containing all the design sub-units used inside the floating point horizontal addition unit.
• fp_testbench – contains testbench modules used for the verification in isolation of addition/subtraction unit and multiplication unit. In addition to this, it also contains a module which tests the correctness, the proper functioning and the proper connectivity of all of the vector floating point units integrated into the top level processor. This testbench does not check the components in isolation, but within their real environment, while running a program.

3.3.2 Strategy

The performance achieved is quite good: only 3 pipeline stages for floating point addition/subtraction unit, 3 pipeline stages for floating point multiplication unit and 9 pipeline stages for the floating point horizontal addition. In the end (taking into account the decode stage) we can perform a floating point addition, subtraction or multiplication every 4 clock cycles (even if there are dependencies between them). Similarly, we can perform a horizontal floating point addition (of 8 floating point numbers) every 10 clock cycles (even if there are dependencies between them). I will describe below the strategy employed to achieve this.

A floating point addition operation is a kind of complicated operation. It requires many steps that are essentially sequential – their exact description is given in the relevant paragraph. With a very big effort very few steps could be paralellized (see [Flynn 1]). But essentially, we are still left with enough sequential steps. In addition, please notice that the techniques described in [Flynn 1] are very well suited for ASIC design. Due to the FPGA particularities it is very tricky to make these optimizations work on an FPGA – it would require a lot of hacking and it would result in a technology independent code.

Given the complexity of the floating point addition, it is clear that it cannot be a “1 clock cycle operation” like the integer addition. The goal was to find the minimum number of clock cycles in which I can implement the floating point addition - given the target frequency (116MHz). How to achieve this and be sure that I found the minimum?

We could say initially that every logical step in the floating point addition would be a clock cycle and place our stages (pipeline registers) according to these logical steps. If there are too many steps, we could try to merge them by “guessing” which step is quick enough to be merged with the previous one or with the next one and still fit in one clock period. We could try out some combinations and see what we can get.
The above-described approach is not very likely to be successful. We have tried it in another projects. The wrong thing with this approach is that we try to determine the best partitioning of the stages. That is, we try to guess the delay of each stage and combine them so that each stage takes about the same time. The approach is wrong because we cannot guess very accurately. The synthesis tool comes with big surprises when it goes to optimization and post place-and-route timing.

Even if we can guess very accurately, there are still problems. Even if we knew precisely the delays (even the post-routing delays), at the RTL level we could place the registers (stages) only at certain logical places. By example, we can place them both at the input of an integer adder and at the output of the integer adder. We cannot do more than that – even if we had the absolute knowledge of the delays. We cannot place it somewhere inside the integer adder.

The solution to the problem presented above is finding a way that allows the synthesis tool to do that balancing. In the end, the tool knows the delays exactly. Moreover, the post place-and-route timing is very different from the pre place-and-route timing. These are details that only the tool can figure out – we cannot.

The idea was inspired by [Kilts]. It is a very elegant way which offers flexibility, clarity and technology independence. We don't guess where to place the registers inside the unit. Actually, we do not place the registers inside the unit. We design the unit as if it was a big and slow combinatorial circuit – with no clock and no reset inputs. This is what we call the core of the unit. It performs the function correctly, but slowly. Up to now we have the advantage of clarity.

Next: the core has an interface – inputs and outputs. We are going to instantiate the core in another (wrapper) module that has the same interface plus a clock and a reset input. Each port of the core will be connected to the respective port of the wrapper module – not directly, but through a configurable number of (shift) registers. All the input core ports will have the same number of registers from the wrapper input ports – and this number is a parameter. All the output core ports will have the same number of registers to the wrapper output ports – and this number is another parameter. All we have to do is to generate the configurable number of “buffer” shift registers, handling also the case when the number is zero (no registers on the input or output ports). The following figure illustrates very clearly the concepts described above.
The last thing to do is to enable the tool optimization: “Register Balancing” (in RTL synthesis) and “Retiming” in “Place and Route - PAR”. These options will enable the tool to move the flip-flops forward or backward (as required) in an optimal way to balance the delays and meet the timing requirements. So, we place the registers at a fixed place (at the inputs or at the outputs), and the tool will shift them along the logic until it meets the timing. Of course, the tool will preserve the correctness and the logical equivalence to the original version. Please notice that the tool will do this 2 times. It does it once during the synthesis: to balance it with respect to the individual components delay (this is the part that we might have a small chance to guess). It does it a second time during the place and route when it has the information of the interconnection delays (this is the part we have no chance to guess). This offers us the technology independence – we don’t need to be
concerned in any way with delays. Please see the following figure for a visual description of register balancing technique.

The only thing left to do is setting the parameter responsible for the number of stages, synthesize it and see if the timing is met. Setting this parameter means only writing a number: 2 (by example) – nothing more! If the timing is met, then decrease the parameter to see if we can get it within a smaller number of stages (clock cycles). If the timing is not met, then increase the parameter and try again. Please notice that the only difference in these trials is the parameter value – which is a number. We get a variable number of stages only by changing a single integer value. If we ever need to target a higher clock frequency we only need to change the parameter (increase it), so that we can allow more stages. We do not need to re-organize (to modify) the unit inside. The person who has to speed it up does not need to understand the internal details of the unit in order to do that. This gives us a great flexibility.

This methodology helped us achieve only 3 stages for addition/subtraction and 3 stages for multiplication.
3.4 The Scalar Floating Point Addition/Subtraction Unit

3.4.1 The Interface and Block Diagram

The unit has the following input ports:
- clk: 1 bit clock signal
- rst: 1 bit reset signal
- A: 32 bits left operand for addition/subtraction
- B: 32 bits right operand for addition/subtraction
- operation: 1 bit signal to select addition or subtraction
- start: 1 bit signal to indicate when operands are valid and the unit shall start evaluating the result

The unit has the following output ports:
- R: 32 bits result of the operation
- overflow: 1 bit signal to indicate whether the currently finished operation result has overflow
- underflow: 1 bit signal to indicate whether the currently finished operation result has underflow
- busy: 1 bit signal to indicate that the unit has got at least 1 operation pending (not yet finished) in the pipeline
- out_valid: 1 bit signal to indicate that the output result signal is evaluated (ready) and it is valid for sampling

The figure below shows the block diagram of the floating point addition/subtraction unit:
3.4.2 Algorithm Description

Our implementation follows very closely the algorithm and optimizations described in [Knuth].

The core module (to be found in file: “fp_adder_core.v”) evaluates the result \( R \) which can be \( A+B \) or \( A-B \) depending on the input port: “operation”. The evaluation is combinatorial, so the core unit has no clock or reset signal – according to our methodology described earlier. The wrapper unit to be found in file: “fp_adder.v”) contains the buffer registers around the core unit.
First, the exponent difference is evaluated. However, we need the absolute value of this difference. The exponent difference will then be applied to the right shift unit as the shift right quantity. The operand with a smaller exponent will have its significand shifted to the right with an amount that equals the absolute value of the exponents.

To evaluate the absolute value of the exponents difference, we use two 9 bits (not 8, to handle correctly the sign) integer adders in parallel, as integer subtracting units. We evaluate both:

\begin{align*}
\text{exponent}(A) - \text{exponent}(B) \\
\text{and} \\
\text{exponent}(B) - \text{exponent}(A)
\end{align*}

in parallel to gain speed and we choose the positive one. Apparently, one might argue that we waste an extra adder, so we use extra area. This is not true. If we used only one adder, then what do we do in case the result is negative? There would be two options.

First option would be changing the sign of the result. But this always uses an incrementer, which is about the same big and fast as an adder. And even worse: it would be now connected in a serial path (not in a parallel path), degrading further the timing. So, timing would be worse and the area would be almost the same.

The other option would be creating a right shifter which is able to interpret also the negative shift amounts. This would both increase the size of the shifter and slow down the unit. In the end, the extra size that the second adder takes it is not actually an extra size. We would pay for it later anyway. And we would also get a slower unit. In conclusion: using the two 9 bit adders is a very good solution.

The significand associated with the smaller exponent will be then shifted right with the absolute value of the exponents difference, evaluated above. The right shift amount that we are interested in is in the range from 0 to 26. To explain why, we have to turn to [Knuth]. In the book, the number of bits the significand is represented on is denoted by \( p \). If the exponent difference is bigger than \( p+2 \), then the result of the addition/subtraction is exactly the bigger operand (the one with the bigger exponent) – we can ignore the smaller operand. This happens because we operate with normalized numbers and we use the round to nearest even mode. Under this conditions, if the other operand is small enough, it cannot influence any of the bits in the representation of the result.

For this case \( p \) is 24 (23 bits in the significand plus one implicit bit – ignoring the sign bit). This is why we only care about the shift quantities between 0 and 26. Any other shift amount applied to the right shifter will generate a 0 on the shifter result. The exponent difference can be in the range from 0 to 255. This is one of the reasons we preferred to have an exact description adapted to this particular situation for the right shifter. We would
have wasted area and speed if we had chosen to have a generic right shifter. Please notice that in case we want to port this design to 64 bits (instead of the current 32), the right shifter must be updated, probably re-written. But this is not a big task. The function (definition) of the right shifter is very clear, simple and precise. It can be re-written without having to understand the rest of the addition unit.

Now, we are able to perform the operation (addition or subtraction) on the aligned significands. Again, according to [Knuth], we need an integer adder on 2*p+2 bits in order to compute the result precisely. This time we have to take the sign bit into consideration, so p=25. That is, we would need a 52 bits integer adder. This would not only be a big adder, but also a slow adder. [Knuth] describes a way to optimize this, without losing precision. Again, the conditions under this optimization works are: normalized significands and rounding to nearest even mode. According to [Knuth], if we take care of the last bit we could use only a p+2 bits adder without losing precision (actually p+3 in binary, p+2 in any other base bigger than 2). So, we can reduce the integer adder size from 52 to 28 bits (we preferred a 28 bits adder instead of 27 in order to easily manipulate the sign). The bit at index “p+3” will become the LSB (least significant bit) of the operands. The bits from index “p+4” to “2*p+2” will be dropped. If there is at least a bit in 1 among the bits being dropped then the LSB (bit p+3) will be 1; otherwise LSB will be 0.

Next, we need the first one detection. This module is described in the file: “fp_first_one_detector.v”. It is a generic, very scalable unit. It detects the position of the first bit in 1, from left to right. There are 2 cases here. First, the leading one is a few positions to the right of the decimal point. In this case we need a shift left to normalize the result. The shift left amount can only be between 0 and 26. The shift left amount is also provided by the first one detector. Again, for speed and area efficiency reasons, the left shifter is described in a non-generic way (similar to the right shifter). Again, the function (definition) of the left shifter is very clear, simple and precise. In case we want to go for a 64 bit version FPU, it can be re-written without having to understand the rest of the addition unit. It is important to notice that a left shift might generate an underflow. The exponent will require adjustment. More precisely, we will decrease from the exponent the amount we shifted the significand to the left. If the exponent becomes negative, that means an underflow has to be generated. We will activate the underflow output.

The second case is when we don't need a left shift. This happens, by example if the operation was addition and the exponents were equal. In this case the leading one is two positions away to the decimal point, to the left. In order to normalize the result we need a right shift of 1 position.

The last step is rounding. We perform rounding to nearest even. For a complete explanation why the rounding is done to the nearest even, please see [Goldberg] and [Knuth]. In principle, if we always round a quantity like “a.5” to “a+1”, we will not perform very well in statistical computations. We will have a bias. Rounding “a.5” to the nearest even means actually rounding it sometimes to “a”, sometimes to “a+1”.
It is important to notice that the last step, the rounding, may generate a new normalization step or even overflow. So, after the rounding (which may require an incrementation of the significand) we might need a 1 position right shift, together with an exponent increment. If the exponent overflows now, then the whole operation has overflow. We will set the overflow output in this case.

The `out_valid` generation is very simple for addition/subtraction. Because the core is pure combinatorial, we set the `out_valid` bit as soon as the start bit gets active. If we look at our unit from the wrapper’s perspective, we actually notice that between start and `out_valid` there is a shift register with a length that equals the number of stages.

Please see below a more detailed view of this unit:
3.4.3 FPGA resources

No DSP slices are used for the floating point addition/subtraction unit. The biggest adder is a 28 bits adder. The shifters use a variable shift amount. There’s actually no big gain in using a DSP slice here.

3.5 The Scalar Floating Point Multiplication Unit

3.5.1 The Interface and Block Diagram

The unit has the following input ports:
- clk: 1 bit clock signal
- rst: 1 bit reset signal
- A: 32 bits left operand for multiplication
- B: 32 bits right operand for multiplication
- start: 1 bit signal to indicate when operands are valid and the unit shall start evaluating the result

The unit has the following output ports:
- R: 32 bits result of the operation
- overflow: 1 bit signal to indicate whether the currently finished multiplication result has overflow
- underflow: 1 bit signal to indicate whether the currently finished multiplication result has underflow
- busy: 1 bit signal to indicate that the unit has got at least 1 multiplication pending (not yet finished) in the pipeline
- out_valid: 1 bit signal to indicate that the output result signal is evaluated (ready) and it is valid for sampling

The figure below shows the block diagram of the floating point multiplication unit:
3.5.2 Algorithm Description

The floating point multiplication follows also the approach described in [Knuth]. It is much simpler than the floating point addition.

For the exponents, we just need to add them and subtract the bias (which is 127). The significands will be multiplied as integers, using an integer multiplication unit.

The integer multiplication unit is implemented using two DSP slices. The exact description is given later. The important thing to notice is that it takes 2 clock cycles – it cannot be done with a combinatorial unit. This means that the core of the floating point multiplication is not a combinatorial unit as it was the case for floating point addition/subtraction. So, the core (described in the file: “fp_multiplication_core”) needs an input port for the clock an oe for the reset. It also means that the minimum number of stages required for the floating point multiplication is 2. We take care of this with the parameter: INT_MULT_DELAY (internal multiplication delay) which is set by default to 2 (and it is actually never changed to another value). For uniformity, the floating point addition/subtraction wrapper unit also has a parameter called INT_ADD_DELAY (internal addition delay) which is always set to...
0. If we ever decide to use DSP slices for implementing floating point addition, than we have to use registers and then this parameter will be bigger than zero even for addition.

After the significands multiplication is performed, it is interesting to notice that we may need no normalization at all or, in the worst case, one bit right shift normalization. This is because our both significands are between 1 and 2. Then their product will be bigger than 1 and strictly smaller than 4.

After the possible normalization, the rounding will be performed. This is very similar to the floating point addition rounding. Again, it may require re-normalization and exponent re-adjustment.

After the final exponent adjustment we can decide if the operation has overflow of underflow. We generate an overflow if the final exponent is bigger then 255 and we generate an underflow if the final exponent is smaller than 0.

In order to compensate for the 2 clock cycle delay of the integer multiplication on the significands, that path computing the exponents and the sign has also to be buffered. The buffering is done in a generic way, using the parameter INT_MULT_DELAY to generate the same number of stages for the exponent and sign path.

3.5.3 FPGA resources

Two DSP slices are used for the floating point multiplication unit. As our significands are 24 bits wide, we need exactly 2 DSP slices to implement the significands multiplication within 2 clock cycles in a fully pipelined way.

A DSP slice offers an 18x25 integer multiplier. One DSP slice is used to implement the multiplication:

\[
\text{significand}_A \times \text{significand}_B[17:0]
\]

in one single clock cycle. The other DSP slice is used to implement a multiplication and an addition:

\[
\text{significand}_A \times \text{significand}_B[23:18] \text{ (shifted left 18 positions)} + \text{significand}_A \times \text{significand}_B[17:0]
\]

in 2 clock cycles, by using the result from the previous DSP slice. Please notice that the shifting left with 18 positions is not acquired by simply by the arrangement of bits and not by using an expensive shift unit.
3.6 The Vector Floating Point Horizontal Addition Unit

3.6.1 The Interface

The unit has the following input ports:
- clk: 1 bit clock signal
- rst: 1 bit reset signal
- A: 256 bits organized as 8x32 bits vector components to be added together
- start: 1 bit signal to indicate when operands are valid and the unit shall start evaluating the result

The unit has the following output ports:
- R: 32 bits result of the operation
- overflow: 1 bit signal to indicate whether the currently finished horizontal addition result has overflow
- underflow: 1 bit signal to indicate whether the currently finished horizontal addition result has underflow
- busy: 1 bit signal to indicate that the unit has got at least 1 horizontal addition pending (not yet finished) in the pipeline
- out_valid: 1 bit signal to indicate that the output result signal is evaluated (ready) and it is valid for sampling

3.6.2 Performance

The floating point horizontal adder performs the addition of all the components of a vector, which are in number of 8. To perform the addition of 8 floating point numbers, without using the horizontal addition unit, we require 7 load-add instructions. A load takes 2 clock cycles, a floating point add takes 4 clock cycles. In total, it would take 42 clock cycles.

The role of the floating point horizontal addition unit is to improve on this as much as possible. To achieve the maximum performance, the horizontal adder is organized as a binary balanced tree. For 8 numbers the tree has a depth of 3 and uses 7 floating point addition/subtraction units. This offers the maximum paralellization possible. Please notice that by using less than 7 adder units, there is no way to achieve the same performance. Well, 4 adders could also do it, by reusing them. I did not want to adopt this solution because in future we would want to fully pipeline our vector processor. That is, we want to allow another horizontal addition instructions to be issued before the current one finishes execution. We would like to have up to 9 horizontal additions one after the other in the pipeline. If we reused the 4 adders this would not have been possible anymore. We would have to wait until the current horizontal addition finishes. This is why I stick to our more elegant implementation.
As we’ve seen before, on floating point adder could be implemented in 3 stages. Our horizontal adder has a depth of 3. This gives 9 stages in total. That is, we can perform one horizontal addition of 8 floating point numbers every 10 cycles, under any circumstances. We reached the theoretical minimum limit (given that one addition takes 3 stages), so we stop searching for improvement on floating point horizontal addition.

3.6.3 Description

The horizontal adder unit is written in a generic way (it is very scalable). That means we can configure the unit to add 4 floating point numbers, or 8, or 16, 32 ... (any power of 2). All it needs to be done is changing the value of parameter called TERMS_NUMBER in the file fp_definitions.v. This is all. The current value of this parameter is now 8. Please notice that the minimum number of numbers to be added is 4 (it would not make sense to set this parameter to 2).

With the current settings (that is TERMS_NUMBER set to 8), the horizontal adder structure is a balanced binary tree of depth 3 (please see the figure below). There are four adders on the first level, two on the second and 1 on the third. Each previous level provides (by its result) the operands for the next level. On a particular level, all the adders will finish computing their result at the same time (as all of them have 3 stages). The start bit for the horizontal addition is distributed to all the adders on the first level. The out_valid bit from the first level distributes the start bit for all the adders on the second level – and so on. Finally, the last level adder’s out_valid bit delivers the ready signal for the horizontal addition operation.

We could have connected (by example) all the start bit of the level 2 adders to the out_valid bit of the first adder of the first level (as all the out_valid bits of the first level adders are totally equivalent). We consider that in this way we would increase the fan-out of one adder unnecessarily and it would not scale very well. So, we connect the start bit of each adder on a level n to another out_valid bit adder from a level n-1. Please see below the structural organization of the floating point horizontal addition unit.
The provide the global overflow and underflow of the horizontal addition unit, I “or” together all the overflow (respectively underflow) bits of each addition unit. If an exception (overflow or underflow) is generated for a partial result (that is, an exception is generated as a higher level in the balanced tree), then the output of the horizontal adder signals that exception before the evaluation of the whole addition is ready. Our external circuitry in the vector TRM module will register this exception until horizontal adder finishes the current computation. When the current horizontal addition is ready, the exception is kept for one more clock cycle in the exception register (the same as for addition/subtraction and multiplication).

Please notice that unlike the vector floating point addition/subtraction and multiplication units, the horizontal addition unit does not have a combinatorial core that is buffered with registers within the wrapper unit. It already uses buffered addition units, which we already know they work fine with their current chosen number of stages.
3.6.4 FPGA resources

No DSP slices are used for the floating point horizontal addition unit. The unit uses only floating point addition units inter-connected in a balanced binary tree.

3.7 Exceptions

Each of the Vector FPU blocks generates two exceptions: overflow and underflow. In case of addition/subtraction and multiplication, an exception can be generated only at the same time with activating the `out_valid` (ready) signal. Within a vector we consider that an exception is thrown when at least one scalar component throws the exception. The exception is then registered (in a flip-flop). The next clock cycle the responsible flip-flop (their name is `overflow_vfp` and `underflow_vfp`) will be set to one. One clock after, the corresponding flip-flop will be cleared back – in this way getting prepared to store a new exceptional condition. The programmer has therefore 2 options: either check both exception flags right after executing such a vector floating point instruction or map those two flip-flops as interrupts – in this case no need for a check, the right routines are called automatically.

In case of horizontal addition, an exception can be generated before the result is valid, because a higher level on the tree generated an exception during a partial addition evaluation. In this case the exception is generated when it happens and it stays until 1 clock cycle after the horizontal addition is finished. The programmer does not need to treat this exception differently, because during the time the exception is valid, only for 1 clock cycle (the latest), stall signal is low. All other “n-1” previous clock cycles the stall signal was active.
4 Verification Framework

4.1 Block-Level Verification

The scalar floating point addition/subtraction and multiplication have each a block-level testbench associated with (tb_fp_adder and respectively tb_fp_multiplier). Inside these testbenches the units are instantiated. The testbench provides the data input, and the start signal, waits for the response of the unit and checks if the received data matches the expected data – so the testbench is also provided with the expected, correctly pre-computed data.

The block-level testbench does not test the components within their real environment. They cannot check if the device under test is properly connected in its environment. But the block level testbench can check in a great detail the functionality of the device under test.

Normal data as well as boundary conditions data are provided to the units under test. Comutativity is checked, as well as cases when one operand is zero (or both), or the result is zero. Plus/minus zero are also checked. Underflow/Overflow exceptions are checked too.

The testcases are completely automated. They report after each check (111 checks for addition/subtraction and 36 for multiplication) whether the result and overflow/underflow signals are OK or not. If they are OK the test goes further to the next combination. If something is wrong, the testcase stops the simulator, reports the error, the step and the pattern: expected data, received data. At the end of the simulation, the testcase will report the successful termination if this is the case.

The block-level testbenches check the units extensively in isolation.

4.2 Top-Level Verification

We want to be able to validate out units also in the context of the real environment and a running program. In order to achieve this, the module under the file: "tb_fp_VFPU_checker.v" is provided and integrated into our testbench structure.

The top-level testbench is focused on checking if the components are properly connected and if they interact properly with each other. It complements very well a block-level test. Finally, the top-level testbench guarantees that the overall system works as desired when running a real program.

The respective module is a module containing only inputs. All these inputs are signals from within the VectorTRM module (where our units are to be found), but also some
important signals from the scalar TRM module. Because the signals to be checked and inspected are in different hierarchies, the signals applied at our VFPU_checker module have to be referred by their full hierarchical name.

This testbench is also fully automated – when things are fine it just displays a message that the check has been done successfully. When something wrong happens, the simulation is stopped with an error message and the debug information (received and expected data) is printed.

The block-level testcases described before needed to be provided with the input data as well as the expected data, for verification. For this top-level testbench I took a more elegant approach: no expected data is required. The programmer does not need to compute the result in advance and save it – to compare it afterwards with the received result. The programmer does not even need to be too much concerned with the input data. Our testbench module acts like a spy which is waiting for events to happen. When the events like: vector floating point addition, subtraction, multiplication or horizontal addition happen, the spy gets active and starts to do all the required checks.

More precisely, our testbench waits for events on the decoding signals corresponding to our vector floating point operations: addition, subtraction, multiplication, horizontal addition (which of these has a signal associated that decodes the operation).

At all times our checker module checks that not more that one operation is decoded to start the execution. At all times it is also checked that no more than one operation, namely the expected operation, is ready to deliver the result.

Also the time it takes an operation to complete is checked (3 cycles for addition, subtraction, multiplication and 9 cycles for horizontal addition) are checked. During the time an operation is under evaluation, it is checked that stall signal is active, that the operands read from the vector register file are constant and not changing. We also check that the write enable signals to the scalar register file and the vector register file are low (that is nobody is trying to write to the registers while a vector floating point operation is pending).

When a vector floating point operation is finished, we check that the result available on the corresponding unit output equals the write data input to the vector register file (for addition, subtraction or multiplication) or the write data input to the scalar register file (for horizontal addition). We also check that one and only one write enable for either scalar or vector register file is enabled. This proves that when a unit is ready its result will be written to the right register.

The only thing left to be checked is that the received data output of the corresponding unit is the right result expected for the input operands. In order to perform this (important) check I use the language (Verilog) to compute the reference expected result. As Verilog can do floating point operations only on 64 bits, I have written some conversion functions.
In case a result has overflow/underflow the corresponding check is skipped (only for that scalar component) and a message is printed, informing the user about that.

Anytime a result or a signal does not conform to the expectations, an error message is printed and the simulation is stopped. Anytime a check is successfully performed a message is printed.

This “spy” checking module is by default enabled (in our top level testbench). It always performs the check in the background. Even if the intended program is not a program whose purpose is to test the vector floating point units, but it contains some vector floating point operations, the checks will be performed. And as we saw, not only the computed results are checked; also other associated signals which assure a quality automated self-check. The checker can be disabled (so, no more messages are printed and the simulation won’t stop if something wrong is detected), by setting the parameter `ENABLE_VFPU_VERIFICATION` to 0 in the `fp_tb_VFPU_checker.v` file.
5 Floating Point Division – Software Implementation

5.1 The Algorithm

The floating point division of two numbers is a floating point multiplication between a number and the other’s inverse. As we’ve seen, floating point multiplication is implemented efficiently in hardware. We decided to implement the inverse of a number in software. This way, we can just call a library function to implement floating point division. I use Newton’s method to find the inverse of a floating point number. The main reference for the current implementation is [Omondi].

I actually implement the vector floating point inverse of a number. Given a vector with 8 components, I return another vector with 8 components, each of them being the inverse of the corresponding one in the original. In the implementation section I discuss more about what happens if only one component is zero – by example. In the current section I discuss about the Newton’s algorithm for finding the inverse of one single component.

Newton’s method very well suited to finding an approximate solution of a non-linear function. It can be applied to a continuos and derivable function \( f \), whose derivative \( f' \) is non zero within an interval. If such function has got a root, and fulfills the above conditions in an interval around the root, we can find sequence of approximations of this root with the following procedure:

- take an approximate root
- draw the tangent line to the function’s graph in the point chosen before
- intersection of this tangent line gives a better approximation of the previous approximate root
We can repeat the above steps until we are satisfied with the approximation.

Writing formally the steps above (by writing the equations of the lines), we can infer a recurrence formula, called Newton’s formula:

\[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \]

In our case the function we want to evaluate is:

\[ f(x) = D - \frac{1}{x} \]

Function \( f \) is a function having the root \( 1/D \) (that is, exactly the number we are searching for: given \( D \) we want to find \( 1/D \)). Finding good approximations of the root of this function (for a given \( D \)) means actually finding good approximations for the inverse of \( D \): \( 1/D \).

Please notice that if \( x \) is not 0, then this function fulfills easily the conditions above. Making the necessary replacements, we get our recurrence formula:

\[ x_{n+1} = x_n \times (2 - D \times x_n) \]
Where D is the number we want to find an inverse for. The things we still need to evaluate are: the initial approximation and the number of steps to assure a satisfying precision. For our case, the convergence is fast: quadratic.

As shown in [Omondi], a good approximation for the initial, \( X_0 \) is:

\[
X_0 = 2.9142 - 2\cdot D
\]

Within this conditions and if \( 0.5 < D < 1.0 \), we can find the inverse of any D (within 0.5 and 1.0) in just 4 steps. These 4 steps are enough to assure that all our 32 bits (that we use to represent the number) are precise.

### 5.2 The implementation

The program to find the inverse is implemented in the file: “test_div.asm” (together with some lines of code also that test it).

Given an arbitrary number D represented as a floating point number on 32 bits, we cannot start computing directly the initial approximation \( X_0 \), with the formula above and then apply the recurrence. We have to make sure that D fulfills the condition:

\[
0.5 < D < 1.0
\]

Of course, D is an arbitrary number. Most likely, it won't fulfill this condition. We can scale it, so that it fits the requirements. Afterwards, we will need to scale it back with the inverted amount. Below, we will describe this process formally (this is actually what the program does).

The initial number to be inverted, D, is written like (we ignore the sign for the moment):

\[
D = (1.f) \times 2^{e-127}
\]

where f is the significand and e is the exponent stored in the number format.

We consider now a new number, G:

\[
G = D \times 2^{126-e} = (1.f) \times 0.5
\]

Number G fulfills now our condition, provided f is not zero. If f is zero (that is: 1.f is 1.0), finding the inverse is really easy:

\[
D^{-1} = (1.f) \times 2^{127-e} = (1.f) \times 2^{(254-e) - 127}
\]
So in case \( f = 0 \), the inverse has the same significand (zero) and the exponent: \((254-e)\). In this case we will have to generate underflow in case \( e > 254 \) (that is, in case \( e = 255 \)).

Further we consider that \( f \) is not 0. In this case we have:

\[
1.0 < 1.f < 2.0
\]

and this implies:

\[
0.5 < G < 1.0
\]

So, \( G \) fulfills now the condition from Newton’s method. We apply now the recurrence described in the previous section for \( G \). After the the four steps we get \( G^{-1} \). Let’s consider:

\[
G^{-1} = (1.F) \times 2^{127-E}
\]

\( F \) and \( E \) are known to us, as they are the result of the Newton’s method. On the other hand:

\[
1.0 < G^{-1} < 2.0
\]

That means that exponent of \( G^{-1} \) can only be zero (we work only with normalized numbers); that is: \( E = 127 \).

Now we know that:

\[
D^{-1} = G^{-1} \times 2^{126-e} = (1.F) \times 2^{127-E+126-e} = (1.F) \times 2^{126-e} = (1.F) \times 2^{(253-e)-127}
\]

So, the inverse of \( D \) has the significand \( F \) and the exponent \((253-e)\). If \( e > 253 \) (that is \( e == 254 \) or \( e == 255 \) then we have to generate underflow). Let’s summarize.

\( D \) is given with significand \( f \) and exponent \( e \). We want to compute the significand \((f')\) and the exponent \((e')\) of \( D^{-1} \), the inverse of \( D \). The following steps will achieve this:

- if \( f == 0 \) and \( e == 0 \) then generate “division by 0”. END
- if \( f == 0 \) and \( e == 255 \) then generate underflow. END
- if \( f == 0 \) and \( e !== 255 \) then \( f' = f \) and \( e' = 254-e \). END
- if \( f != 0 \) and \( e == 254 \) then generate underflow. END
- if \( f != 0 \) and \( e == 255 \) then generate underflow. END
- if \( f != 0 \) and \( e < 254 \) then \( f' = F \) and \( e' = 253-e \) (\( F \) is computed with Newton). END

The steps above are implemented exactly in the program from file: “test_div.asm”.

Please notice that this program computes the inverse of a vector (that is the inverse of 8 components). If a single one is zero, then a division by zero exception is thrown. If a single
one has underflow then an underflow exception is thrown. Please notice that floating point inverse on 32 bits with normalized numbers cannot throw an overflow!
6 Conclusion

Our goal has been achieved: we implemented the basic floating point operations on Virtex-5 FPGA, with a good performance.

The overall FPGA usage (not only for vector floating point units, but including also the rest of the processor) is about 20%. So, we can have more processors on a single FPGA.

The latency of the floating point units is excellent (4 cycles for vector floating point addition, subtraction and multiplication and 10 cycles for horizontal floating point addition of 8 numbers).

The speed achieved is 116MHz. For the current implementation, this speed is not limited by the vector floating point units. It is limited rather by the long path: decode-execute-writeback. With deeper pipelines (5 levels) we saw (in previous projects) we can get about 159MHz. Even though we did not try, I believe we could get the floating point units run at 159MHz (due to our flexible methodology to pipeline the FPUs, described in an early chapter). But this is speculation – it is not a fact yet.

All these facts show us that it is possible to implement not only floating point on (relatively modest) FPGAs, but also vector floating point. We got our results on a Virtex-5 FPGA. Currently (in 2010) better FPGAs exist (Virtex-6, Virtex-7). If we target one of those we might get (for free) a higher speed. Also, we can fit even more processors on an FPGA like those. We might even want to go for a double precision (64 bits), or other features.

Another key fact to be mentioned is the presence of DSP slices on the FPGA. We use those DSP slices to implement the integer multiplication, that we use further to evaluate the floating point multiplication. This is a very convenient solution, because the integer multiplication units inside the DSP slices come for free, offer a good speed and a low power consumption (at no extra area cost!).

Our conclusion is that implementing floating point arithmetic on FPGA is feasible, and this work proves it.
7 References

[Flynn 1] - Leading One Prediction: Implementation, Generalization and Application, 1991 by Nhon Quach, Michael J. Flynn

[Flynn 2] – An Improved Algorithm for High-Speed Floating-Point Addition, 1990 by Nhon T. Quach, Michael J. Flynn

[Flynn 3] - Reducing the Mean Latency of Floating-Point Addition, 1998 by Stuart F. Oberman, Michael J. Flynn


[Liu] - Vector Extension to the TRM3 Instruction Set Architecture, 2010 by Ling Liu


[Wirth] – The Tiny Register Machine (TRM), 2009 by Niklaus Wirth