Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-Point Unit

Kingshuk Karuri, Rainer Leupers, Gerd Ascheid, Heinrich Meyr

Institute for Integrated Signal Processing Systems, RWTH Aachen University, Germany

Monu Kedia

Indian Institute of Technology, Kharagpur, India

Abstract

Multimedia and communication algorithms from embedded system domain often make extensive use of floating-point arithmetic. Due to the complexity and expense of the floating-point hardware, final implementations of these algorithms are usually carried out using floating-point emulation in software, or conversion (manually or automatically) of the floating-point operations to fixed point operations. Such strategies often lead to semi-optimal and imprecise software implementation.

This paper presents the design and implementation of a Floating-Point Unit (FPU) for an Application Specific Instruction set Processor (ASIP) suitable for embedded systems domain. Using a state-of-the-art Architecture Description Language (ADL) based ASIP design framework, the FPU is implemented in such a modular way that it can be easily adapted to any other RISC like processor. The implemented operations are fully compliant to the IEEE 754 standard which facilitates portable software development. The benchmarking, in terms of energy, area and speed, of the designed FPU highlights the trade-offs of having a hardware FPU w.r.t. software emulation of floating-point operations.

1. Introduction

Floating-point arithmetic, although extremely common in general purpose computing, was rarely used in embedded systems world until recently. While a number of communication and multimedia algorithms are designed and simulated using floating-point arithmetic, the implementation platforms for such algorithms often leave out any hardware floating-point unit in favor of software emulation or float to fixed point conversion. While the former can be extremely inefficient in terms of speed and energy consumption, the later leads to tedious and error prone software modifications and loss of precision. Moreover, lack of a standard in fixed point, non-integer arithmetic means practically zero portability for any application converted from floating to fixed point. In recent times, sensing this need of floating-point arithmetic hardware, many prominent embedded DSP and media processor vendors such as TI (e.g. TI C67x) [1] and Phillips (e.g. TriMedia) [2] have added FPUs to their cores.

Implementation of an *efficient* and *correct* FPU is an extremely difficult, involved and time consuming task. While major processor vendors can afford to invest the required manpower and time to add FPUs to their products, designers of small, application specific embedded ASIPs often leave out floating-point instructions, even when they are required, due to the high development efforts involved.

This paper presents the design and implementation of a modular and portable FPU - specially suitable for use in small ASIPs - using a state-of-the-art Architecture Description Language (ADL) [7]. The floating-point instructions are very loosely coupled to the main processor core, and can be easily ported to other architectures by minimal changes. We have developed a framework for testing the correctness and the compatibility of the developed instructions w.r.t. the prominent floating-point arithmetic (*IEEE 754* [3]) standard. This paper also presents a summary of area requirements for different classes of FPU instructions, and an estimate of the energy and speed benefits of them. Such results can assist a designer to take decisions about whether to add or leave out a certain instruction in his architecture.

The rest of this paper is organized as follows. After a discussion of the related work in the next section, we introduce the IEEE 754 standard in section 3 and the required design automation tools in section 4. Section 5 covers the implementation details of the FPU, and section 6 presents the testing architecture developed. The area and energy consumption results are presented in section 7. The final section summarizes our work and presents some future directions.

2. Related Work

The basics of floating-point arithmetic are well covered in [5]. A comprehensive treatment of floating-point arithmetic can be found in [6] which also describes the IEEE 754 standard in brief. The details of the standard can be found in [3], and a compliant software reference implementation of the standard is available in [4].

[10] shows that software emulation of IEEE 754 standard is extremely inefficient and proposes a software-oriented floating-point format for automotive control systems. There exists a number of optimized, but non IEEE FPU implementations in hardware, too. Some of them do not support operations on de-normalized numbers to save area [11, 12], whereas some adopt a representation format completely different from the standard [18, 15]. Such implementations may work well for specialized applications, but the resulting hardware and the software can not ported to other systems.

There also exists a number of IEEE 754 compliant floating-point cores [13, 14], pre-designed components [17] or data-path libraries [16]. Such components can be used as separate blocks in embedded System-on-Chip (SoC) designs, but can not be incorporated into ASIP cores.

In contrast to the other works cited here, our FPU is designed to be part of small, embedded ASIP cores. We provide an ADL based implementation of floating-point operations that can be easily integrated to a wide variety of RISC like processor cores, and can be extended or curtailed according to the needs of the target application. Therefore, we have left out several complex features (e.g. pipelining the FPU), in favor of modularity and flexibility. However, once the FPU is ported to a specific processor, the modular design makes it very easy to incorporate processor specific pipelining, data forwarding and other hardware features into it.

3. IEEE 754 Standard for Binary Floating-point Arithmetic

IEEE 754 is the floating-point arithmetic standard followed almost universally. This section provides a very brief overview of this standard for a better understanding of the following sections.

IEEE 754 is an extremely complex standard that specifies the format and representation of floating-point numbers, the allowed operations and their expected outcomes, the exception cases, handling of overflow and underflow etc. A few important aspects of the specification are summarized below:

- **Representation Format:** In the standard, a floating point number is represented using a *significand* and an *exponent*. One bit is reserved for the *sign* of the number. Depending on the sizes of the exponent and the significand, two groups - *basic* and *extended* - are defined in the standard. Moreover, each group has single and double precision representations resulting in a total of four formats with different bit widths.

  The single and double precision representations in the basic group require 32 and 64 bits (C *float* and *double* data types), respectively, and offer sufficient precision for almost all computations found in embedded applications.

- **Floating-point Operations:** Addition, subtraction, multiplication, comparisons, division, square root, and remainder are the arithmetic operations which are specified in the standard. Apart from these, conversions between integer and floating-point formats, between different floating-point formats, and between floating-point numbers and decimal strings are also specified.

- **Exception Handling:** The standard defines the precise semantics of different floating-point exceptions such as overflow/underflow, divide by zero and Not-A-Number (*NAN*) arithmetic. Five floating point exceptions must be properly detected and signaled. The signaling can mean setting a flag, raising a trap or possibly both.

- **Rounding modes:** Rounding takes a number considered as infinitely precise and modifies it to fit in the destination format. There are four rounding modes specified in the standard that determines the outcome when the result of an operation is too wide to fit into the format.

4. Design Automation Tools

It would have been normal for us to implement the FPU in a Hardware Description Language (HDL) like Verilog or VHDL. However, since our goal was to implement an FPU that can be easily integrated into any RISC like processor architecture, we decided to use

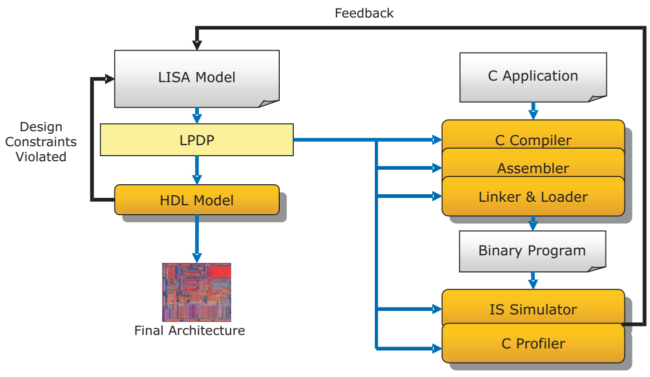


Figure 1. LISA Based ASIP Design Framework

the LISA Architecture Description Language (ADL). LISA allows designers to specify the *instruction-set* and the *micro-architecture* of a processor at an abstraction higher than that of HDLs. The processor specific software tools (i.e. instruction set simulator, assembler, linker and loader) and an HDL model, in VHDL/Verilog/SystemC, can be automatically generated from such a description using *LISATek Processor Development Platform (LPDP)* [8]. LPDP also supports *semi-automatic* generation of a C compiler from the LISA model, but it requires some extra design effort. As figure 1 shows, LISATek tools assist ASIP designers to iteratively refine their processor architectures.

In LISA, the instruction set of a processor is specified in a hierarchical fashion. This feature allows a modular implementation of the instructions with maximum code re-usability. In the next section, we will show how we used this feature to implement the FPU in a modular way. The generation of the target specific software tools assisted us in quickly developing a testing architecture. The generated HDL models are almost as efficient as those written manually [9], and provided us with a mechanism to benchmark our FPU in terms of area and timing.

Apart from the LISATek tools, we have also used Synopsys Design Compiler [20] and Prime Power [21] tools for obtaining results of hardware generation.

5. Implementation of Floating-point Instructions

This section presents the details of the FPU implementation using LISA. Instead of designing a completely new architecture, we decided to implement the FPU in an already existing LISA processor model. We selected a simple RISC like processor model, named *LTRISC*, as the base architecture. The architecture has a RISC instruction set with 16 32-bit *General Purpose Registers (GPRs)* and four pipeline stages - *Fetch, Decode, Execute* and *Writeback*. The instruction words are 32-bit wide with sparse instruction coding, and the instruction set implements almost all integer fixed-point operations including addition, subtraction, comparisons, logical operations and integer multiplication. The following subsections describe the different aspects of design and integration of the FPU in the LTRISC architecture.

5.1. Instruction Short-listing for Implementation

Implementation of all the formats and operations specified by the IEEE 754 standard can be very costly in terms of area. However, as mentioned in section 3, the *basic* format in the standard provides sufficient precision for almost all embedded applications. Therefore, we decided to implement operations only for the *basic single and double precision* format.

Embedded processors seldom require and can afford more than a few of the operations specified in the standard. Therefore, we had to short-list only a limited set of operations for our implementation. We decided to leave out the conversion operations to and from integers, strings and other floating point formats. We also eliminated more advanced operations such as the square root, remainder and division, since even in some advanced architectures such as the *Intel IA64* [19], these operations are emulated in software. There is very little need to over-design the FPU with such expensive operations.

Therefore, we selected a *minimal set* of instructions using which all the required arithmetic and comparison operations in the standard can be executed either in hardware, in software or using a combination of both. For the same reason, we decided to support all the rounding modes and exception signaling in the hardware since they might be needed for software emulation of more advanced operations (or for future hardware extensions to implement such operations). A list of the implemented operations are supplied below.

1. **Arithmetic:** We selected *Addition*, *subtraction* and *multiplication* for hardware implementation. As mentioned already, square root, division and remainder operations can be easily implemented in software using iterative techniques like Newton-Raphson method.
2. **Comparison:** We decided to implement *Less than*, *less than equal to* and *equal to* in hardware. Greater than, greater than equal to and not equal to can be easily handled by the compiler with the implemented instructions.
3. **Memory:** Since our base architecture is a RISC, we decided to implement load and store instructions for single and double precision floating-point values.

5.2. Instruction Coding

The reason for selecting LTRISC as our base architecture was its simple and clean RISC instruction-set and the well structured hierarchical organization of its instructions - called the *coding tree* (This is a misnomer since the coding *tree*, in reality, is a *Directed Acyclic Graph (DAG)*). The well organized instruction-set allowed us to implement the floating-point instructions as an isolated part of the LISA coding tree that can be easily detached from the other portions of the instruction-set, and incorporated into other RISC like processors.

A part of the LTRISC coding tree with the floating-point instructions is shown in figure 2. In the figure, each node represents a LISA *operation* - the basic LISA unit for describing the assembly coding, syntax and behavior of the instructions. An instruction usually consists of many operations. The advantage of such design is that the common functionalities between a set instructions can be put into an operation that can be *shared* by these instructions. This ensures maximum code reuse.

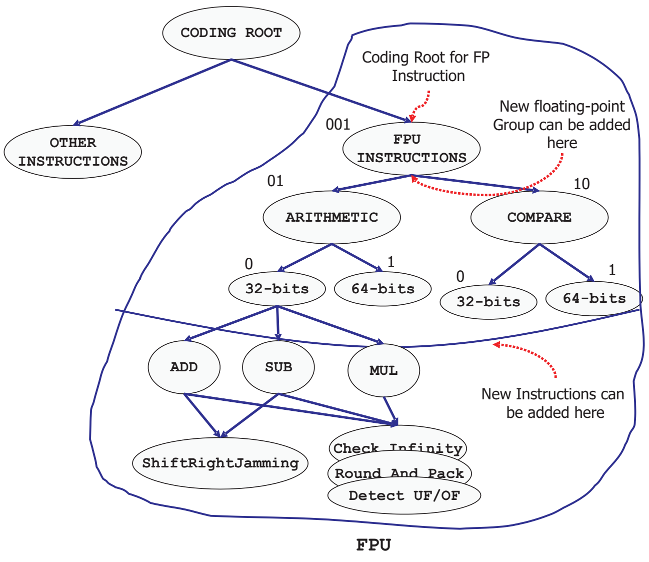


Figure 2. Coding Tree of LTRISC with Floating-point Instructions

The floating-point instructions are added as a completely separate group in the original coding tree. All LISA operations beneath the node labeled *FPU INSTRUCTIONS* in figure 2 constitute our FPU. The floating-point instructions all have *001* as the first 3-bits of their instruction coding. This differentiates them from every other instruction group. The rest of the bits are defined in a way that preserves modularity based on the formats and the functionalities of the instructions.

The floating-point instructions are grouped into three different categories - *arithmetic*, *comparison* and *load-store* (only arithmetic and comparison have been shown in figure 2 for the sake of convenience). Further subgroups are defined under each category according to precisions and formats. A certain number of bits in the coding distinctively mark instructions belonging to each group. For example, instructions in *arithmetic* category have *01* as the fourth and fifth bits in their instruction coding.

5.3. Implementation Details

LISA allows designers to describe the behavior of any *LISA operation* using plain ANSI C code. An *instruction data path* is defined by the combined behavior of the LISA operations constituting the instruction. The control path and the decoding logic is implicit in the LISA coding-tree.

As shown in figure 2, the main implementation of the FPU is defined by the behaviors of the operations such as *add, sub* and *mul* (and other instructions not shown for sake of brevity). The C behavior of these instructions is modeled after the IEEE 754 compliant software implementation provided by [4] which is efficient, but contains C constructs (such as *labels* and *loops*) without any hardware equivalents. Therefore, we adapted and changed the C code according to our needs. Moreover, we grouped the common functionalities (e.g. *shift with right jamming*, *round and pack* etc.) into operations that are shared by other operations. This led to maximum code reuse during design and resource sharing in

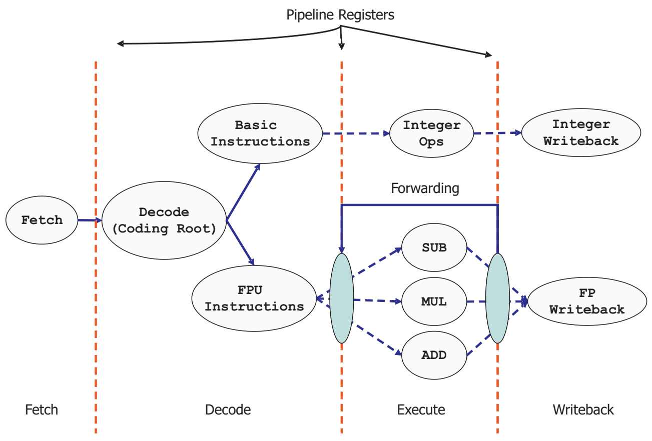


Figure 3. FPU in the LTRISC Pipeline

hardware model generation from LISA.

We added a new floating-point register file consisting of 16 32-bit registers in the architecture. The floating-point instructions exclusively access this register file. The double precision operations use 8 *pairs* of 32-bit registers as 8 double precision registers.

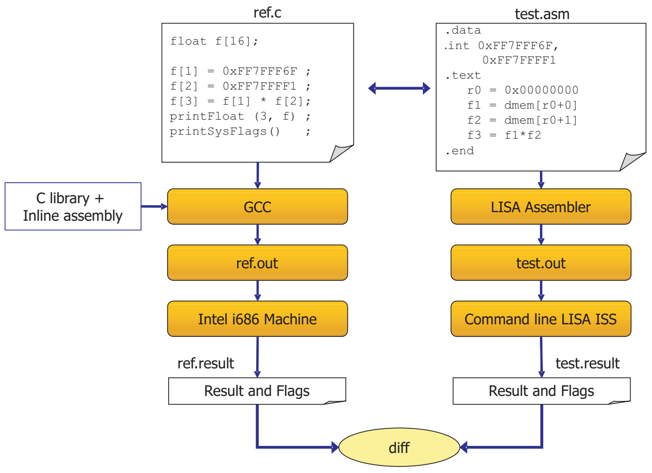
LISA allows designers to assign each operation in different pipeline stages. Figure 3 shows the FPU in the LTRISC pipeline structure. The pipeline has four stages (Fetch, Decode, Execute and Writeback) which are connected by pipeline registers shown with dotted lines. The operations that define the behavior of different floating-point instructions (such as add, mul and sub) are all assigned to the *Execute* stage of the pipeline. They are *activated* from the coding root (i.e. the decode operation) indirectly. Note that there is no *interaction* between the integer and the floating point operations. The FPU is completely independent of other parts of the processor in its instruction coding and implementation. As shown in the figure, the results of the floating-point operations are *forwarded* from the output pipeline registers of the Execute stage to its input. Thus, all floating-point operations are single cycle, zero latency operations.

New instructions	16
New LISA Operations	79
Extra lines of LISA code	4891
Extra resources	16×32-bit registers and several pipeline registers
Extra lines of generated Verilog code	22360

Table 1. Statistics of the Implemented FPU

In table 1 the statistics of the implemented FPU is presented in detail. It is possible to integrate the FPU with minimal effort to any other RISC like processor model written in LISA. The only thing that *must* be modified is the coding of the different instructions which requires only a few lines' change in the LISA code. The size of the register file and the distribution of the instructions over different pipelines *might* need to be changed depending on the processor architecture.

6. Testing


```
float f[16];

f[1] = 0xFF7FFF6F ;
f[2] = 0xFF7FFFF1 ;
f[3] = f[1] * f[2];
printFloat (3, f) ;
printSysFlags()   ;
```

```
.data
.int 0xFF7FFF6F,
     0xFF7FFFF1
.text
   r0 = 0x00000000
   f1 = dmem[r0+0]
   f2 = dmem[r0+1]
   f3 = f1*f2
.end
```

Figure 4. Testing Architecture for the FPU

An FPU implementation can not be complete without a robust and extensive testing mechanism. Therefore, we have also developed a testing framework for our implemented FPU using the software tools generated from LTRISC LISA model. This testing architecture is pretty generic and can be adapted to other processor models. This section describes our testing architecture in detail.

The major goal of our testing was to check the IEEE 754 compliance of our FPU. We decided to treat any outcome that does not conform to this standard as incorrect. For this, we needed an *reference* implementation of IEEE 754 that is *robust* and already *well tested* . Fortunately, such an implementation was readily available at hand in the form of the FPU of any Intel i686 machine.

Our testing architecture is depicted in figure 4. A test case (*test.asm*) written in LTRISC's assembly language is assembled and linked through LISATek generated tools. The resulting executable is executed on an LTRISC *Instruction Set Simulator (ISS)* that dumps the results and the flags after each floating-point operation. This *operation trace* is compared to that produced by a reference application written in C (*ref.c*) and compiled and executed on an Intel i686 machine. The reference application performs the same floating-point operations with the same inputs as the LTRISC assembly file, and a set of predefined function calls print out the outcome of the operations and the flags using *inline assembly*. The LTRISC operations are IEEE 754 compliant if the outputs of ref.c and test.asm match. Writing the ref.c in a way that it exactly does the same things as test.asm is left to the user.

We did not invest the extra effort in retargeting the LTRISC compiler to make use of the newly available floating-point instructions. If such a target specific compiler is available, the *ref.c* file can be directly compiled and executed on ISS, and the test.asm file can be done away with.

We have written around 50 test cases to verify different corner cases regarding exceptions, NaN arithmetic, over and underflow and de-normalized number arithmetic. For the cases investigated so far, our testing architecture revealed one bug that has been corrected in the FPU.

7. Results

This section presents some results of our FPU implementation w.r.t. area and energy consumption. The results were obtained by generating synthesizable HDL models from our LTRISC implementation automatically, and then synthesizing them using Synopsys Design Compiler [20].

7.1. Area and Speed

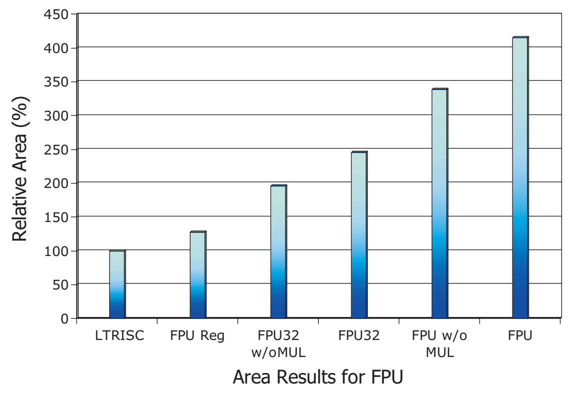


Figure 5. Area of LTRISC with FPU w.r.t. Original LTRISC

Figure 5 presents the area overhead of our FPU relative to the original LTRISC model. All the models have been synthesized using a 0.13 μm library under 40ns (25 MHz) clock speed. The hardware models have been obtained with highest possible optimizations in the LISA hardware generator. We have synthesized the HDL descriptions of six different LTRISC configurations as described below -

1. Basic LTRISC
2. LTRISC with floating point registers. This non-combinational area overhead for floating-point instructions is almost 30% of original LTRISC area.
3. LTRISC containing single precision (32-bit) FPU without multiplication. Single precision add, subtract and three comparison operations increase the area by another 70%.
4. LTRISC with complete single precision FPU (i.e. FPU with multiplication). A single precision floating-point multiplication alone requires almost 50% of the original LTRISC area.
5. LTRISC containing single (32-bit) and double (64-bit) precision FPU without any multiplication. This causes an area increase of almost 240% in the original core.
6. LTRISC with complete FPU which is a little more than 4 times as big as the original LTRISC. Therefore, the FPU alone has more than *3 times* the size of the original LTRISC.

As can be seen from the results, the multiplication (both single and double precision) operations by far are the most expensive operations. They alone account for an area increase of almost 125% of the original core. Therefore, depending on the frequency of multiplications in the target applications, the architect must decide whether to incorporate multiplications in the design or not.

The initial area results presented in figure 5 has been obtained with a quite long clock period of 40 ns. Shortening the clock cycle usually results in area increase of the core. Figure 6 shows how the area of our FPU increases with shorter clock cycles. As can be seen, the area of the complete FPU increases by around 70% of the original LTRISC area when the clock period is halved to 20 ns. Further shortening of clock leads to cycle time violation. For the single precision FPU, however, the area increase due to shorter clock is minimal. The single precision FPU can be run with a minimum clock length of 15 ns (i.e. a maximum clock speed of around 66 MHz).

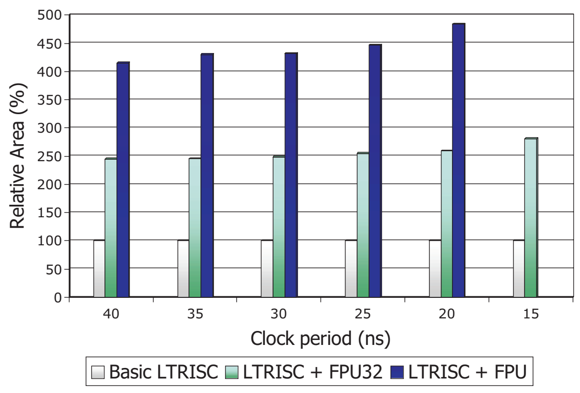


Figure 6. Area Increase in FPU with Shortening Cycle Time

As is shown in [10], software emulation of even the simplest floating-point operations take around 20 hardware cycles for completion. Therefore, for floating-point intensive applications, the implemented FPU will do better than faster fixed-point machines even with a clock speed of 40 ns. However, a better solution will be to split the FPU into several pipeline stages that will improve clock speed and area, both.

7.2. Energy Consumption

Figure shows the power consumption figures for our FPU, (obtained using the same 0.13 μm library) in comparison to the original LTRISC model. The power figures were obtained by simulating integer, single and double precision floating point implementation of two applications - *factorial computation* and *Fibonacci series generation upto the 6th term* on the generated HDL models. The integer and floating-point version of the applications contained the same set of operations - only the integer arithmetic was replaced by floating-point arithmetic.

As can be seen, the single precision floating point operations, on average, consume 2.5 times power (i.e.

1.5 times more power) than that of integer operations, and the double precision operations eat up, on average, around 5 times of the same. However, when we consider that several integer arithmetic operations are required to emulate one floating point operation, the savings in *energy consumption* become clear. For example, a single precision addition consuming 1.5 times more power than an integer addition still consumes only a fraction of energy of software emulation which takes around 50 cycles [10]. Although it is difficult to make a general comment on power consumption by simulating two simple applications, it is obvious that our implemented FPU can lead to at least *one order of magnitude* less *energy consumption* than floating point emulation.

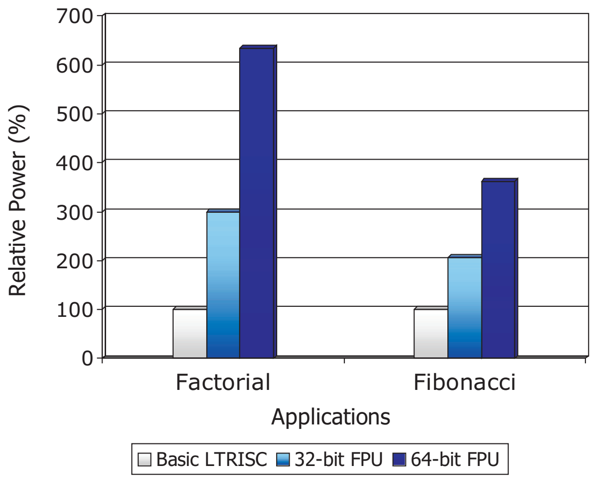


Figure 7. Power Consumption by the FPU

7.3. Comparison with Commercially Available FPUs

For the sake of completeness of this section, it is necessary to compare our implementation with some other commercially available FPUs. Such comparison, however, is fraught with difficulties due to the differences in specification formalisms (ADL vs. RTL), technology libraries, clock frequencies, effects of pipelining etc.

Many commercial embedded processors, such as [23, 22], provide FPU implementations that can be used with their base processor cores. Xtensa 6[22] has a 32-bit FPU , compliant to the single precision IEEE 754 standard, implemented using 25 K gates, whereas our single precision operations require around 29 K gates. However, the Xtensa FPU can have *multi-cycle* operations, and therefore, can operate at a much higher clock frequency. The ARM VFP9-S co-processor[23] has 32 single precision registers compared to our 16. The area of this unit is between 100-130 K gates, whereas our implementation takes only 62 K gates. This is not a fair comparison since the VFP9-S can operate at a much higher clock frequency and implements the IEEE 754 standard fully.

A better comparison with individual FPU blocks written in RTL can be done by considering the data presented in [24]. The combined area of single and double precision FPU blocks (excluding division) in this implementation, is around 66 K gates compared to our 62 K gates. Since the synthesis results are shown using a 0.18 μm library, a direct comparison is still not possible.

From the above discussions it can be conjectured that the area results of our implemented FPU is comparable to those of the commercially available FPUs. It is not possible to have a really meaningful comparison of timing and power consumption without considering the effects of pipelining (which most commercially available FPUs implement) our FPU.

8. Conclusions

This paper presents the design and implementation of IEEE 754 compliant floating-point instructions in an ASIP. The FPU is designed in a modular way and can be integrated to other architectures with minimal effort. Moreover, depending on the requirements of the applications, the FPU can be extended or cut down to basic operations easily. These features make it suitable for use in small embedded processors. A testing architecture has been developed for the FPU and basic testing of the corner cases have been done. The benefits of the hardware implementation of the FPU w.r.t. software emulation has also been studied.

In future, we plan to make the FPU pipelined and study its effects on area, timing and energy consumption. Implementation of other advanced operations such as division and square-root is another possible work. But more importantly, we want to look into the compiler retargeting issues for the floating-point instructions so that we can perform more intensive testing of the FPU.

References

[1] http://dspvillage.ti.com/
[2] http://www.semiconductors.philips.com/products/nexperia/
[3] http://grouper.ieee.org/groups/754/
[4] Softfloat distribution available at http://www.cs.berkeley.edu/ jhauser/arithmetic/SoftFloat.html
[5] J. L. Hennessy, D. A Patterson: *Computer Architecture : A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc. ISBN 981-4033-227
[6] D. Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Volume 23, No 1, March 1991
[7] A. Hoffmann, T. Kogel, A. Nohl et.al:*A Novel Methodology for the Design of Application-Specific Instruction-set Processors (ASIPs) using a Machine Description Language*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 20, Issue: 11, Nov. 2001
[8] http://www.coware.com/products/lisatek.php
[9] O. Schliebusch, A. Chattopadhyay, E. M. Witte et. al: *Optimization Techniques for ADL-driven RTL Processor Synthesis*, IEEE Workshop on Rapid System Prototyping (RSP), June 2005.
[10] D. A. Connors, Y. Yamada, W. W. Hwuy: *A Software-Oriented Floating-Point Format for Enhancing Automotive Control Systems*, Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES98). December, 1998
[11] K. Taek-jun, J.Sondeen, J.Draper: *Design Trade-Offs in Floating-Point Unit Implementation for Embedded and Processing-In-Memory Systems*, IEEE International Symposium on Circuits and Systems, 2005, May 2005
[12] http://www.gaisler.com/doc/grfpu_dasia.pdf
[13] http://www.dcd.pl/
[14] http://www.nallatech.com/mediaLibrary/images/english/2432.pdf
[15] J. Dido, N. Geraudie, L. Loiseau et.al: *A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs*, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays
[16] R. Hossain, J.C. Herbert, J.E. Gouger, R. Bechade : *Datapath library reuse in the design of a high performance floating point unit*, Proceedings of Eleventh Annual IEEE International ASIC Conference, Sept. 1998
[17] C. Brunelli, F. Campi, J. Kylliainen, J. Nurmi : *A reconfigurable FPU as IP component for SoCs* Proceedings of 2004 International Symposium on System-on-Chip, November, 2004.
[18] S. Lee,I. Park: *Low cost floating-point unit design for audio applications* IEEE International Symposium on Circuits and Systems, May, 2002
[19] M. C. Hasegan, Bob Norin: *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic*, Intel Technology Journal Q4, 1999
[20] http://www.synopsys.com/products/logic/design_compiler.html
[21] http://www.synopsys.com/products/power/primepower_ds.pdf
[22] http://www.tensilica.com/products/x6_floating_point.htm
[23] http://www.arm.com/products/CPUs/VFP9-S.html
[24] http://www.asics.ws/doc/efpu_brief.pdf