Floating Point project

Jamil Khatib

October 23, 2000

Contents

1 List of authors and changes

Name	Changes	Date	Contact address
Jamil Khatib	Initial release	17-7-2000	khatib@opencores.org
Jamil Khatib	Tests and Interfaces added	13-8-2000	khatib@opencores.org
Jamil Khatib	Design spec changed to generic CPU	23-10-2000	khatib@opencores.org

2 Project Definition

2.1 Introduction

Floating point numbers and calculations have great use in every day calculations like banks transactions, scientific calculations, graphics, egineering drawings and even games. These kind of calculations will be very solw if they are done without any hw support. This floating point unit core should deliver speed up the calculations. The hardware core and software libraries is going to be provided.

2.2 Objectives

The main objective of this project is to build an IEEE-754 compatible Floating-point unit core and its software. This core should give high performance and have the ability to interface to any CPU core. The project should provide a set of small calculation units. The final step should use the fpu core in a stand alon floating point co-processor.

3 Specifications

3.1 System Specification

- The core should be compliant with the IEEE-754 standard.
- The core is going to have several phases one phase for each precision.
- The precision has to be configurable and selectable.
- The first phase should provide the functionality then optimization for speed and area should be considred later.
- The Software part should provid the compiler, assembler, traps handling, higher precisions and extra calculations done in compiler libraries.
- Supported instructions **TBD**. web site has some suggestions.
- A stand alone co-processor has to be built.
- The functional units have to operate as a stand alone calculation units
- The FPU core assumes that the CPU has to issue the instructions and know when the result should com out. (not valid in the stand alone mode)
- The FPU has no storage registers, all values and results should be stored in the CPU. (not valid in the stand alone mode)

- The CPU should decide whether can issue FP instructions or not (detect hazards etc.). (not valid in the stand alone mode)
- This document will describe only the execution core unit, The stand alone mode will be described in the future.

3.2 External Interfaces

- The core should be as an execution unit of the cpu.
- The core should give exceptions to the CPU to store them.
- The CPU must have the ability to mask and unmask the exceptions.
- Check the FPU interface table

FPU interface

Direction	Size	Name
INPUT	[31:0]	operand A
INPUT	[31:0]	operand B
INPUT	[TBD:0]	operation type
INPUT	[1:0]	rounding mode:
		0 - To nearest (DEFAULT)
		1 - To $+\infty$
		2 - To $-\infty$
		3 - To ZERO
OUTPUT	[31:0]	result
		Compare Result (Mutually exclusive)
OUTPUT	1	larger then (A ¿ B)
OUTPUT	1	smaller then (A ¡ B)
OUTPUT	1	equal (A == B)
OUTPUT	1	unordered (A == NAN — B == NAN)
OUTPUT	1	infinity (A == INF — B == INF)
		Exceptions
OUTPUT	1	overflow
OUTPUT	1	underflow
OUTPUT	1	divide by zero
OUTPUT	1	SNAN: Invalid operation
OUTPUT	1	INE: inexact

3.3 Hardware specification

- In the first phase the CPU should be responsible for instruction decoding, data path control ...
- All calculations should be done on the minimum supported extended precision.

3.4 Software specification

3.5 Interface between SW and HW

4 Internal Blocks

The hardware can be divided into the following main blocks:

- **Execution Units**
  - All blocks should take the operands in decoded format
  - So they should have the ability to take non-normalized inputs.
  - They can generate non-normalized results
  - They should have the ability to take and generate the special values.
  - Each block should generate all 5 exceptions that are connected to the flag status block.
  - Execution units are composed of Add/Subtract, Multiply, Divide, Compare, square root and some transidantal units **(TBD)**
- **Status/Mask flags block**
  - This block is connected to all execution units.
  - The user should have the ability to read and mask the flags.
  - The flags should remain set as long as the software do not unset them.
  - The CPU should be interrupted upon any generation of unmasked exception.
  - The software should handle the flags to prevent frequent generation of interrupts and it also trap and handle all exceptions and modify the results upon the source of exception.
- **Decode unit** that decodes numbers into the the extended format with the hidden bit.
- **Encode/Round** result that rounds the extended results into the specified precision.

5 Design description

5.1 Decode Unit

5.1.1 Design notes

- It decodes numbers into the the extended format with the hidden bit.

- It is needed to enable all calculation be made on extended precision.
- The conversion is made as shown in the flow chart.
- The core uses single level of registers.

5.1.2 Timing and flow charts

5.2 Adder/Subtracter Unit

5.2.1 Design notes

- Invert the sign of the second operator in case of subtraction.
- Check if the one of the operands is a special number (NaN or Infinity)
- If NaN is one of the operands the result is NaN
- If ∞ check if the operation is $\infty - \infty$ then NaN and Invalid exception is generated
- The operands can be non-normalized and also the results
- check if the result over or under flows regarding the used precision.
- The unit needs two cycles to get the result.

5.2.2 Timing and flow charts

5.3 Compare Unit

5.3.1 Design notes

- Check if one of the operands is NaN so the result is False
- Check the sign of the operands
- Equalize the two exponents and then check the fractions
- There is no need to have a special case for Infiity
- This unit does not generate exceptions
- It needs two cycles to get the result

5.3.2 Timing and flow charts

5.4 Scripts, files and any other information

5.5 Design conventions and coding styles

5.6 Design Modeling

6 Testing and verifications

6.1 Simulation and Test benches

Test benches should be made for each execution block at least to verify their operations with any interference from other blocks.

Specifications of test benches **TBD**.

6.2 verification techniques and algorithms

The functional verifications will be made in two steps

1. Through SW verification by set of test vectors and checking versus IEEE-754 compliant software.*SoftFloat Library can be used*
2. Through prototyping on FPGA and testing it on real operations and calculations.

6.2.1 Verification Software

- The software should generate test vectors for the test benches (at least for the main test bench).
- It should generate numbers and operations randomly for the test bench.
- It should to read mathematical formulae and generate sequence of numbers and operations to the test bench.
- There should be a method to compare the results from the real calculations done on the CPU and from the fpu core.
- All results from test bench and software should be in readable format with the ability to show all bit representations.
- The software should have the ability to generate operations that give special numbers (infinity, zeros and NaN) and all kind of exceptions.
- The software should know when the special numbers are going to be generated and not to calculate them using the normal software flow, else it will fail because the results will be handled by host traps.

6.3 Test plans

Testing should take care of complex operations and well known bugs cased by sequence of operations.

Test plans, equations and operations are **TBD**

6.3.1 Adder/Subtractor unit

Tests to be run on the add/sub unit:

1. The same exponent
   (a) Addition on operands having the same sign
   (b) Addition on operands having different signs follow these tests
      - Operand1 is -ve and op2 is +ve and the result is +ve
      - Operand1 is +ve and op2 is -ve and the result is +ve
      - Operand1 is -ve and op2 is +ve and the result is -ve
      - Operand1 is +ve and op2 is -ve and the result is -ve

   (c) Different set of numbers should be applied
2. Different exponents
   (a) Do the same tests as above (all tests in 1)
   (b) let Op1 has the larger exponent
   (c) let Op2 has the larger exponent
3. Denormalized numbers
   (a) Op1 should be denormalized and op2 is normalized
   (b) Op2 should be denormalized and op1 is normalized
   (c) Both op1 and op2 should be denormalized
   (d) calculations as in 1 and 2 should be made but the result should be denormalized
4. Special numbers
   (a) NAN should be one of the operands.
   (b) both operands should be NAN
   (c) $\pm\infty$ should be one of the operands
   (d) Both operands should be $+\infty$ or $-\infty$
   (e) Operands should be NAN and ∞
   (f) op1 should be $+\infty$ and op2 should be $-\infty$

(g) op2 should be $+\infty$ and op1 should be $-\infty$

5. Do the same above tests (in 1 & 2) on the subtraction operation.
6. Do the same tests in 1,2,3 and let the result be very large to overflow or very small to underflow

7 Implementations

8 Reviews and comments

9 References