LSI’s ZSPFastFloat Floating Point Format and ZSPFF Coprocessor Implementations for ZSP DSP Cores

Ramon Trombetta
LSI Logic Corp.
500 N. Central Expwy, Suite 430
Plano, Texas 75044, USA
(+1) 972-244-5106
ramont@lsil.com

Tim O’Gara
LSI Logic Corp.
500 N. Central Expwy, Suite 430
Plano, Texas 75044, USA
(+1) 972-244-5106
togara@lsil.com

Abstract

DSP architectures can be divided into fixed-point (FXP) and floating-point (FLP) DSPs. FXP DSPs only offer integer arithmetic but tend to be very fast, consume less power, and cost less than FLP DSPs. However, FLP DSPs offer higher precision and dynamic range using FLP arithmetic. There are three major reasons for an increasing demand for FLP support in FXP DSP architectures. First, algorithm reference code is often written in C/C++ using floating point numbers, and it is not trivial to convert an FLP algorithm to an FXP algorithm. FLP support reduces development costs because there is less time spent porting the floating point algorithm to the DSP platform. Additionally, custom algorithms may benefit from the dynamic range provided by using floating-point arithmetic. Finally, in some cases it may be impossible to maintain the required accuracy and dynamic range using FXP arithmetic. Therefore, it is beneficial to have a FLP format that is handled efficiently by the FXP DSP.

Keywords

Floating Point, Coprocessor, ZSP, DSP

Introduction

Most DSP development tools support only very basic FLP operations on the C level, but the DSP hardware is often strictly FXP oriented, not offering any FLP support. The lack of hardware support dramatically limits any emulated FLP operations and often makes the basic FLP operations impractical in real applications. There are at least three methods used to support FLP arithmetic on FXP DSPs without moving to FLP DSPs. A FXP DSP with a highly-optimized FLP software library, a FXP DSP tightly-coupled with a FLP coprocessor and a FXP DSP with a loosely-coupled memory-mapped FLP accelerator. All of the hardware solutions add production cost (increased silicon area of FLP hardware) as a trade-off to reduce development cost and power consumption. The software solution requires much more processing cycles and may not be practical for many real systems.

The optimal system solution may only be determined some time after the project evaluation phase when the implementation has already started. Thus, it is desirable that all mentioned methods are built on a common FLP data format and instruction set for basic FLP operations. Only then is it possible to evaluate and profile code and algorithm in all three system environments.

First, we will give an overview of a FXP DSP followed by a brief description of the IEEE-754 floating point standard. Next, we will describe our proposed FLP format and the description of a FLP coprocessor instruction set.

LSI’s ZSP DSP cores

Two generations of ZSP architectures are presently available as either licensable cores or standard off-the-shelf components. Both generations are based on load/store type superscalar RISC architectures. They are targeted at a wide range of applications.

ZSP instructions are scheduled in hardware at run time. The dynamic grouping caters strictly to in-order instruction execution allowing predictable VLIW-like instruction packets without the attendant difficulties of programming and optimizing assembly code for a VLIW DSP core. The RISC-style orthogonal instruction set makes code generation simple and straightforward. It also helps to boost C compiler performance. All cores share a 16-bit basic instruction set, even as ZSP-G2 makes selective use of 32-bit instructions. This ensures backward compatibility and high code density. All cores can support Von Neumann or Harvard memory organization. All ZSP cores also deploy similar static branch prediction, hardware looping mechanisms, special addressing modes and bit operations. Their native FXP data path has a data width of 16-bit, but all of them offer strong 32/40-bit support.

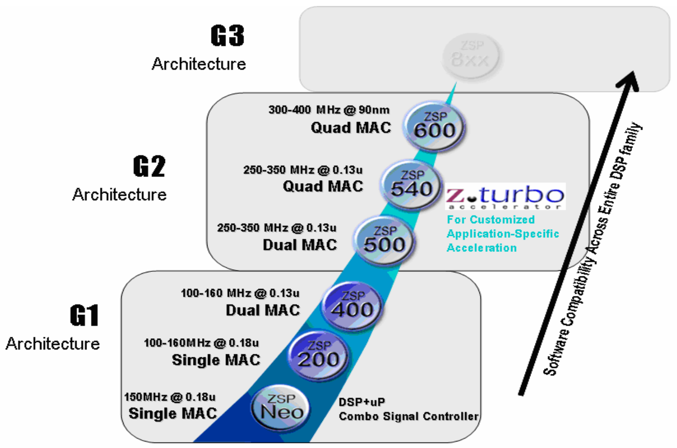

Figure 1: ZSP DSP cores

ZSP G1 (First Generation) Architecture

The ZSP G1 architecture is a low-cost, mid-range performance DSP platform. The ZSP G1 has a locked 5-stage pipeline and, depending on the implementation, up to four instructions can be fetched and executed every cycle. Also depending on the number of MAC units it is capable of executing up to two multiply accumulate operations per cycle. It has two 16-bit ALUs that can be combined as a 32-bit ALU. The G1 cores support arithmetic, logical, shift, rounding, normalization and bit-manipulation operations. G1 cores feature sixteen 16-bit general-purpose registers that can be paired to create eight 32-bit registers.

ZSP G2 Architecture

LSI Logic's second generation DSP cores are also based on a 16-bit RISC superscalar machine, but with an eight-stage pipeline. Depending on the core implementation they can issue 4 (ZSP500 - dual MAC and ZSP540 - quad MAC) or 6 (ZSP600 – quad MAC) instructions per cycle.

The architecture defines a general-purpose register file and an address register file. The general-purpose register file contains sixteen 16-bit registers that can be combined in even/odd pairs for 32-bit arithmetic. Each 32-bit register pair also has an associated guard byte to support eight 40-bit accumulators. The address register file contains eight 32-bit pointer registers and eight associated 16-bit index registers.

The G2 architecture has two dedicated address generation units (AGUs), each driving its own dedicated load/store port. The two AGUs allow the core to issue any combination of two loads or stores per cycle. Each data port in the e.g. ZSP500 is 32-bits wide, allowing a total of 64-bits (4 words) of data transfer per cycle. There are no alignment restrictions on any of the load/store operations. The coprocessor interface (CPI) defines a dedicated interface to user definable modules. The CPI is central to this paper and is described later in detail.

Coprocessor Interface

To facilitate cost effective support for application-specific arithmetic features, the core defines a flexible coprocessor interface and the associated instruction set hooks that can be used to expand the arithmetic capabilities of the architecture. The coprocessor interface is a fully registered interface that supports both tightly and loosely coupled coprocessors. When used in a tightly coupled environment, the coprocessor effectively becomes part of the core pipeline and can be considered an extension to the existing arithmetic data path. The instruction set support for the interface provides direct read and write capabilities to both the address registers and the general purpose data registers.

In a tightly coupled coprocessor interface, the coprocessor effectively provides a user configurable or user definable data path that has direct access to, and is directly accessible by, the core. When used in a loosely coupled mode, the core issues commands to the coprocessor in a fire-and-forget manner (the core does not expect the coprocessor to return data). When the associated coprocessor completes its current task, it can interrupt the core where the ISR can then retrieve data or take other appropriate action. See [6] for an example of a loosely coupled coprocessor. The system designer has the freedom to decide on the partition of hardware and software with the coprocessor interface. Figure 2 illustrates the general concept of how a coprocessor fits in the G2 pipeline when used in a tightly coupled mode.

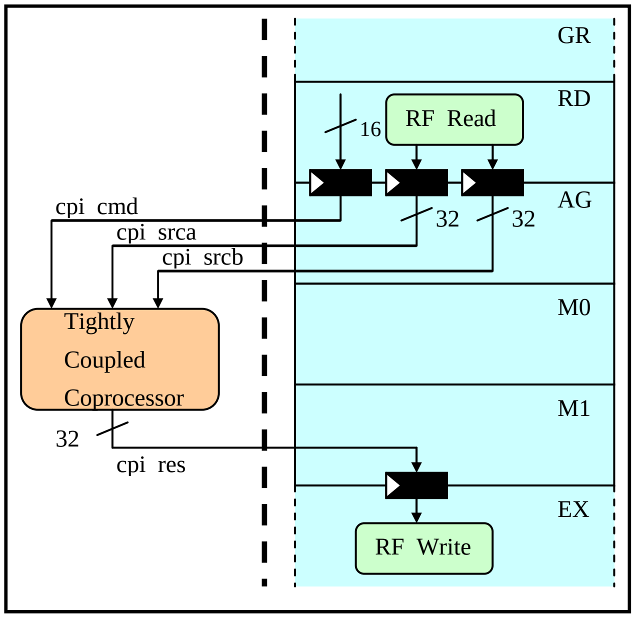

Figure 2: Coprocessor Interface

Operands are read in the RD stage and the coprocessor request (cpi_cmd, cpi_srca, and cpi_srcb) is registered and issued to the coprocessor in the AG stage. The coprocessor designer is responsible for partitioning the coprocessor design so that it returns data (cpi_res) to the core during the M1 stage. The coprocessor result data is registered in the core and written to the register file during EX. Data forwarding is supported from the coprocessor result to the core data paths for more efficient execution.

When used as a tightly coupled device, the coprocessor relies on the core for memory access (the core is responsible for transferring data between the MSS and the coprocessor). Loosely coupled coprocessors can be designed to have direct access to the system memory or its own dedicated memory, freeing up core data bandwidth for other services.

The instruction set support for the coprocessor interface consists of two types of user configurable coprocessor commands: CPOUT and CPCOM. The CPOUT instructions send request and data. The core does not expect data back from the coprocessor with this type of operation. The CPCOM instructions are used to send and retrieve data. Both types of instructions are configurable with a user-defined 16-bit command field that is sent directly to the coprocessor. Since the coprocessor commands are user-defined, this command bus can be used to support multiple coprocessors in the same system. The ZSP development tools facilitate the modeling and simulation of coprocessors with integrated coprocessor development support. Coprocessors can be modeled in C and connected to the G2 core simulator with an easy-to-use API. Compiler and assembler support allows programmers to define user configurable coprocessor instructions for immediate code development.

IEEE-754 Floating Point Standard

There are many different floating point representations, but the one most often used for standard applications, reference code, etc. is the IEEE-754 standard. It defines four common floating point formats. The first two are the single and double precision formats (32-bit and 64-bit). The other two are extended formats for holding intermediate results (>44 bits and >80 bits).

Single Precision

The single precision format stores a floating point number in a 32-bit word with 1 sign bit (bit 31), an 8-bit biased exponent (bits 30-23) and a 23-bit fraction (bits 22-0).

S	8-bit exponent ***E***	23-bit unsigned fraction ***M***

As the exponent is biased by –127, the value of the floating point number is:

$$F = (-1)^S 1.M * 2^{E-127}$$

The smallest positive number:

$$F_{\min} = 1.0 * 2^{1-127} = 2^{-126}$$

The largest positive number:

$$F_{\max} = (2 - 2^{-23}) * 2^{254-127} = (1 - 2^{-24}) * 2^{128}$$

Many IEEE implementations deal with normalized numbers only. Demoralized numbers allow the representation of numbers between 2^{-149} and 2^{-126}:

$$F = (-1)^S 0.M * 2^{E-127}$$

In addition to normalized and demoralized numbers, the IEEE standard reserves some bit combinations for special values.

Double Precision

The IEEE double precision format stores a floating point number in a 64-bit word with 1 sign bit, an 11-bit biased exponent and a 52-bit unsigned fraction.

S	11-bit exponent ***E***	52-bit unsigned fraction ***M***

The exponent is biased by -1023; the value of the floating point number is:

$$F = (-1)^S 1.M * 2^{E-1023}$$

Extended Formats

The IEEE standard also defines two extended formats, a 44-bit and an 80-bit representation. Those are used for intermediate results and have a higher precision and range than the corresponding 32-bit and 64-bit formats.

The IEEE standard defines special bit patterns for values like +/- infinity. It also specifies how special situations like underrun, overrun and rounding are handled and represented.

ZSPFastFloat Floating Point Format

The IEEE approach does not map to the ZSP architecture very efficiently; therefore, two new data formats are defined for the ZSP to allow the coprocessor and the ZSP core to handle floating point numbers efficiently. The formats are not IEEE-754 compliant but instead are based on the ZSP native 16-bit data width.

The ZSPFF formats are straightforward in order to reduce the complexity of the implementation:

- All numbers are denormalized
- No assumed nor hidden bits
- No exponent bias (exponent is a signed two’s complement value)
- No special values for infinity, NaN, ...

ZSPFF16

Exponent:	16-bit signed integer
Fraction:	16-bit signed integer

ZSPFF32

Exponent:	16-bit signed integer
Fraction:	32-bit signed integer

The ZSPFF floating point number is given by:

$$F = M * 2^E$$

	IEEE Single	**IEEE Double**	**ZSPFF16**	**ZSPFF32**
Length (bits)	32	64	32	48
Exponent (bits)	8	11	16	16
Fraction (bits)	23 + 1	52 + 1	16	32
Bias	127	1023	0	0
Range	2^{128}	2^{1024}	2^{32767}	2^{32767}
Precision	2^{-23} or 10^{-7}	2^{-52} or 10^{-15}	2^{-16} or 10^{-5}	2^{-32} or 10^{-10}

Table 1: Comparison of IEEE and ZSPFF floating point formats

As the format of fractional part of the ZSPFF numbers is identical to the most commonly used fractional FXP format Q1.15 (and Q1.32) all arithmetic operations are supported by hardware – all ZSP cores offer Q1.x support. This allows the programmer to switch seamlessly from fractional Q1.x numbers to floating point and also mixing in block FLP numbers can be done easily.

ZSPFastFloat Support Library

To be useful, the floating point format must have a library to convert among the different formats. The conversion functions needed are:

- IEEE to ZSPFF
- ZSPFF to IEEE
- Integer to ZSPFF
- ZSPFF to Integer

The IEEE conversion functions are required when porting algorithms already written assuming the IEEE floating point format. To enable IEEE floating point software on the ZSP platform, the support library will provide an interface layer between the IEEE compliant software and the ZSPFF coprocessor. It will do exception checking, insert special values (replace limited ZSPFF numbers with IEEE special values) and automatic format conversion.

The support layer will use the coprocessor for fast floating arithmetic. Thus the precision will suffer compared to a true IEEE implementation. As shown in Table 1, the ZSPFF formats offer lower precision than the IEEE data formats. This will introduce some error into the arithmetic results.

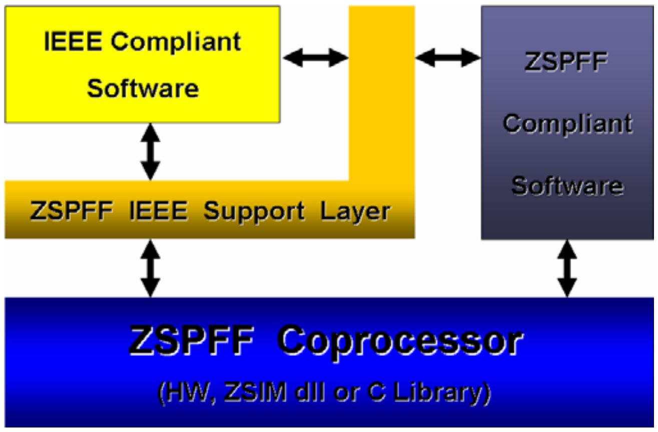

Figure 3: ZSPFF Support Library Architecture

ZSPFF Tightly-Coupled Coprocessor

The ZSPFF tightly-coupled coprocessor (CP) will offer non IEEE compliant floating point support. It is implemented for usage with G2 ZSP cores and the G2 CP interface. The CP instructions have different execution times and the ZSPFF32 instructions need extra data uploads before they can be issued (see Table 3).

Arithmetic Functionality

Only basic arithmetic functions are implemented in the CP hardware:

- ADD / SUB
- MUL / MAC / MACN
- DIV

All arithmetic functions can work on any combination of long and short float values. All functions store a ZSPFF32 result in the internal destination register. There are two versions of each of the operations for delivering short float results back to the ZSP or leaving the ZSPFF32 result in the CPs internal destination register.

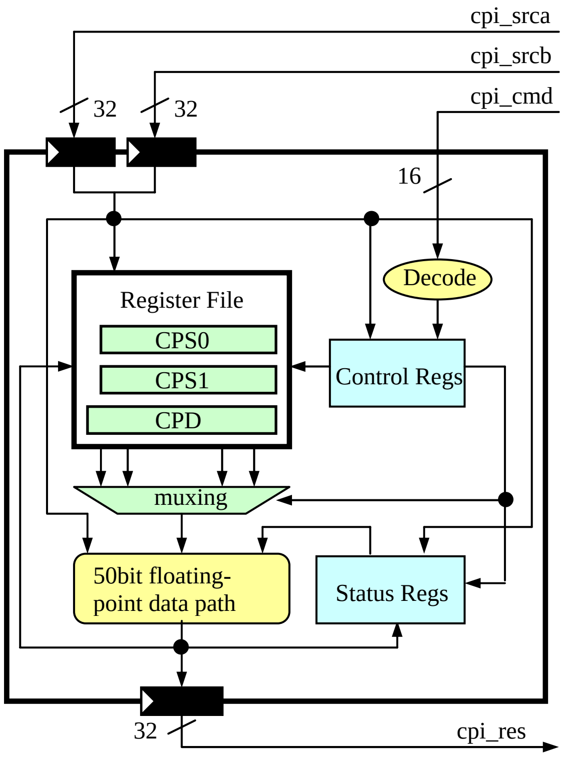

Figure 4: ZSPFF Coprocessor Block Diagram

Rounding / Guard Bits

The CP provides automatic rounding. IEEE requires rounding-to-nearest for minimizing the round off error compared to truncation. For removing bias the ZSPFF CP uses round-to-nearest towards 0 and |Fmax| in an alternating pattern.

For enabling proper rounding and multiplies of the fraction the data path features two guard bits (CPGB).

Saturation

All arithmetic functions perform automatic saturation on the final result. Thus no special values for positive or negative infinity are needed.

Internal Registers

Using the G2 CP instructions it is impossible to send 2 ZSPFF32 numbers to the CP at one time or to get one ZSPFF32 result back from the CP. Thus the source registers and the result register in the CP must be user accessible via dedicated register read and write instructions.

To implement the MAC instruction, the FLP CP will need to have a result register capable of holding the 16-bit exponent and 32-bit mantissa.

- Source Register CPS0 (48bit)
- Source Register CPS1 (48bit)
- Destination Register CPD (48bit)

The CP data path destination has two guard bits, resulting in a 50bit destination register. However, the guard bits are not exposed to the user.

Hardware Flag Register Bits:

Z	-	Zero flag, result is zero
OV	-	Overflow flag, result did overflow and was saturated
SOV	-	Sticky version of OV
DZ	-	Divide by zero flag, division by zero occurred
SDZ	-	Sticky version of DZ

Sticky hardware flags are set by hardware, but have to be cleared manually. The next arithmetic result will not overwrite or clear them.

Interrupts for HW conditions

Any combination of the 5 hardware flags can be tied to a ZSP interrupt line.

ZSPFF Coprocessor Instructions

The CP instruction set is tailored towards the few basic arithmetic functions. However each instruction comes in a variety of flavors to determine which sources or destinations have to be used. Figure 6 shows the basic instruction types available. CPI Type describes the kind of CP instruction functionality.

CPOUT instructions can send data to the CP, but no data is returned to the DSP core. CPCOM instruction can send data to the CP and a 32-bit data value will be returned to the DSP. As the CP is tied to the core pipeline the core will be stalled by the CPI while it waits for data to be returned.

As every instruction can be used as CPOUT coprocessor command, all functions can be triggered and the programmer will have to read out the results later. Thus stall cycles due to excessive division instructions can be avoided.

The instruction set covers all combinations of CP internal data and transferred data for arithmetic instructions. All arithmetic instructions can work on data internal to the CP as illustrated later in the multi instruction ZSPFF32 operation example code. All arithmetic instructions can also work on data passed with the CP instruction as illustrated in the single cycle ZSPFF16 operation example code. Additionally, there are instructions that can use data passed with the CP instruction as well as data internal to the CP. All instructions write the result into the destination register CPD. Instructions that return data to the DSP (.S instructions) will round off the fractional part to 16-bits. Then the 16-bit exponent and the 16-bit fractional part will be sent to the core as 32-bit return value. The 48-bit result in CPD stays untouched and can be used for the next instruction as input.

Command	**CPI Type**	**Description**
CP_INIT	CPOUT	Init the coprocessor
ADD	CPOUT	Perform add and store result in CPD
ADD.S	CPCOM	Perform add and store result in CPD, round, truncate and return result
SUB	CPOUT	Perform subtract and store result in CPD
SUB.S	CPCOM	Perform subtract and store result in CPD, round, truncate and return result
MUL	CPOUT	Perform multiply and store result in CPD
MUL.S	CPCOM	Perform multiply and store result in CPD, round, truncate and return result
MAC	CPOUT	Perform multiply &add and store result in CPD
MAC.S	CPCOM	Perform multiply & add and store result in CPD, round, truncate and return result
MACN	CPOUT	Perform multiply & subtract and store result in CPD
MACN.S	CPCOM	Perform multiply & subtract and store result in CPD, round, truncate and return result
DIV	CPOUT	Perform division and store result in CPD
DIV.S	CPCOM	Perform division and store result in CPD, round, truncate and return result
READ	CPCOM	Read CP internal register and return value
MOVE	CPOUT	Write value to CP internal register

Table 2: ZSPFF16/32 CP Instruction Types

ZSPFF32 bit results (48-bit) will have to be read by the DSP core with two instructions. One instruction reads the 16bit exponent and the second returns the 32-bit fraction.

Example Code

Single Cycle ZSPFF16 Operation

Example ZSPFF16 instruction: **cpcom a0, a1, ADD.S**

The 32-bit register a0 will hold operand 1 and a1 will hold operand 2. The result of the ZSPFF16 add will be written to a0 (a0 += a1).

As the cpcom CP instructions can only send two 32-bit words to the CP, this type of operation fits the ZSPFF16 operations very well. ZSPFF16 operations are the only instructions that don't need any register upload or download for execution.

Multi Instruction ZSPFF32 Operation

A typical ZSP32 operation is of the format 32-bit = 32-bit *(+,-,/) 32-bit. The G2 CP interface can not handle this data bandwidth. For this reason the CP has internal ZSPFF32 source and destination registers to perform the ZSPFF32 operation. However data has to be loaded into these registers prior to the execution, e.g.:

```
cpout   r0, MOVES0.E    // move 16-bit exponent into CPS0
cpout   a0, MOVES0.F    // move 32-bit fraction into CPS0
cpout    r1, MOVES1.E     // move 16-bit exponent into CPS1
cpout    a1, MOVES1.F     // move 32-bit fraction into CPS1

cpout    MULS01           // CPS0 * CPS1

cpcom    r2, READCPD.E    // copy 16-bit exponent (CPD.E) to r2
cpcom.e  r4, READCPD.F    // copy 32-bit fraction (CPD.F) to r4/r5
```

ZSPFF Memory-Mapped Coprocessor

In cases where a dedicated CP interface is not available, a memory-mapped accelerator can be used. This type of CP addresses ZSP G1 cores and in some cases also G2 cores.

The ZSPFF memory based CP is loosely coupled and is not tied into the DSP's pipeline, nor does it have access to any DSP resources except the memory. The DSP has to load floating-point operands from memory and store the operands into the memory-mapped space. Uploading the instruction opcode to the memory mapped register starts the floating-point hardware. After the accelerator finishes execution, it can interrupt the DSP or rely on the DSP to poll a data ready flag.

Because of the overhead involved, this solution is better suited for executing block operations instead of individual floating-point operations. Realizing that any operation consumes more than one DSP clock cycle, the instruction set of the accelerator includes special multi-cycle operations. For example, it can perform FLP IIR filtering very efficiently if setup to work on a block of data. In most cases of block processing, this FXP DSP+Accelerator can outperform a comparable floating-point DSP and also the tightly coupled CP, because of the low overhead per floating-point operation and the fact that none of the fixed-point processor resources are consumed after the setup of the floating-point operations. The address pointer driven, memory-based block operations are added to the basic instructions provided by the CP core – those instructions stay untouched to guarantee code compatibility with both the software library and the tightly coupled CP.

If block processing is required on a core with a CP interface, the DSP does not have to copy instruction opcode to memory mapped registers, but can use the CP interface to communicate with the memory-coupled FLP accelerator.

Results

	IEEE single prec.	ZSPFF16 SW	ZSPFF32 SW	ZSPFF16 Copr.	ZSPFF32 Copr.
ADD/SUB	137	58	64	2	2
MUL	117	43	49	2	2
MAC (N)	247	76	83	2	2
DIV	656	228	251	8	15

Table 3: ZSPFF Performance / Cycle Count on ZSP500

Table 1 shows precision and range of the ZSPFF format compared to the IEEE standard. In Table 3, the cycle counts for IEEE and ZSPFF calculations and the ZSPFF coprocessor are listed. As cycle counts vary from ZSP core to ZSP core, the listed numbers are ZSP500 based and used as an example. Both software implementations, the IEEE and the ZSPFF operations, are hand-optimized assembly code. Those cycle counts are achieved by using the ZSP G2 CP interface. The overhead involved in setting up memory mapped CPs depends on the system design. The same is true for instructions used for memory based block processing.

Summary

In conclusion, fixed-point DSPs can benefit from floating-point support which can be implemented in a number of different ways. As shown with the ZSP fixed-point DSP core family, choosing a floating-point data format that fits the DSP arithmetic capabilities and native data width, adding hardware or software extensions and providing appropriate development tool support results in a DSP solution that achieves floating-point performance and enables floating-point applications on fixed-point systems. The nice cost-to-performance ratio of the fixed-point DSP can be maintained while increasing the range of applications for the fixed-point DSP to include those that benefit from floating-point support.

References

1. LSI Logic, "ZSP200 Very Low Power, Compact DSP Core", Document Order No. R20084
2. LSI Logic, "ZSP400 Digital Signal Processor Architecture Technical Manual", Document DB14-000121-03
3. LSI Logic, "ZSP500 Digital Signal Processor Core Technical Manual", Document DB14-000221-03
4. LSI Logic, "ZSP600 Digital Signal Processor Core", Document DB14-000289-00
5. Wichman, S., Trombetta, R., Chiang, P., "Motion Estimation Using the ZSP500 DSP Core," Proceedings of the ISPC, March 2003.
6. D. Wilson, "An Efficient Viterbi Decoder Implementation for the ZSP500 DSP Core," Proceedings of ISPC, March 2003.
7. IEEE, "IEEE Standard for Binary Floating-Point Arithmetic, 1985", ANSI/IEEE Std 754-1985