Low-Cost Concurrent Error Detection for Floating-Point Unit (FPU) Controllers

Michail Maniatakos, *Member*, *IEEE*, Prabhakar Kudva, *Senior Member*, *IEEE*, Bruce M. Fleischer, *Member*, *IEEE*, and Yiorgos Makris, *Senior Member*, *IEEE*

Abstract—We present a nonintrusive concurrent error detection (CED) method for protecting the control logic of a contemporary floating-point unit (FPU). The proposed method is based on the observation that control logic errors lead to extensive data path corruption and affect, with high probability, the exponent part of the IEEE-754 floating-point representation. Thus, exponent monitoring can be utilized to detect errors in the control logic of the FPU. Predicting the exponent involves relatively simple operations; therefore, our method incurs significantly lower overhead than the classical approach of duplicating the control logic of the FPU. Indeed, experimental results on the openSPARC T1 processor using SPEC2006FP benchmarks show that as compared to control logic duplication, which incurs an area overhead of 17.9 percent of the FPU size, our method incurs an area overhead of only 5.8 percent yet still achieves detection of over 93 percent of transient errors in the FPU control logic. Moreover, the proposed method offers the ancillary benefit of also detecting 98.1 percent of the data path errors that affect the exponent, which cannot be detected via duplication of control logic. Finally, when combined with a classical residue code-based method for the fraction, our method leads to a complete CED solution for the entire FPU which provides a coverage of 94.1 percent of all errors at an area cost of 16.32 percent of the FPU size.

Index Terms—Error correction, control logic, floating point, IEEE-754

1 Introduction

As aggressive scaling continues to push technology into smaller feature sizes, various design robustness concerns continue to arise. Among them, the frequent occurrence of transient errors [1], [2] has resurfaced as a contemporary problem of interest. This problem is mainly attributed to strikes by neutrons or alpha particles and the corresponding single event upsets (SEUs) in memory bits, or single event transients in combinational logic, which may potentially result in a soft error [3], [4]. However, several other factors such as design marginalities, negative bias temperature instability, coupling, power supply noise, and so on, [5], [6] also threaten the robustness of modern microprocessor units. The increasing severity of the above threats has spawned renewed efforts in developing cost-effective concurrent error detection (CED) methods for various key components of a circuit.

Arithmetic and logic units (ALUs) are fundamental building blocks of any microprocessor. Practically, every instruction goes through the ALUs. Thus, enterprise microprocessor designs include reliability features that protect ALUs against errors [7], [8], [9], [10]. Floating-point units (FPUs), in particular, are among the most crucial and hardest to protect [11], [12], mostly due to the inexact calculation nature of floating-point arithmetic. And while progress is being made on solutions using error detecting/correcting codes for the data path portion of an FPU [13], [14], [15], little is known about its control logic, where either duplication [16] or triple modular redundancy (TMR) [17] techniques are usually applied. Control logic errors might have catastrophic impact on the FPU output [18], [19], jeopardizing application execution and providing the end user with erroneous results. Furthermore, the size of control logic in modern FPUs is significant, often amounting up to 20 percent of the FPU size, thus rendering the application of error detection methods necessary.

In this study, we propose an elegant novel method to protect the control logic of an FPU by monitoring the exponent part of the floating-point representation. Our method is based on the conjecture that a control logic error will incorrectly guide the data path and, by extension, severely alter the expected outcome of the performed operation. As a result, it is highly likely that a control logic error will modify the value of the exponent portion of the floating-point output. Given that it is relatively straightforward to calculate the correct exponent through simple operations, monitoring exponent correctness leads to an inexpensive yet very efficient CED method for the FPU control logic. Furthermore, it provides the ancillary benefit of detecting errors in the exponent part of the representation and, when combined with a residue code-based error detection method for the fraction, it results in a very low-cost CED solution for the entire FPU.

The rest of the paper is organized as follows: Section 2 briefly describes existing techniques for the protection of FPUs, followed by a short introduction to the IEEE-754 floating-point arithmetic standard in Section 3. Section 4 describes the proposed exponent monitoring-based CED

- *M. Maniatakos is with the Department of Electrical Engineering, New York University Abu Dhabi, Abu Dhabi, UAE, PO Box 129188. E-mail: michail.maniatakos@nyu.edu.*
- *P. Kudva and B.M. Fleischer are with the T.J. Watson Research Center, IBM, Yorktown Heights, NY 10598. E-mail: {kudva, fleischr}@us.ibm.com.*
- *Y. Makris is with the Department of Electrical Engineering, University of Texas at Dallas, Richardson, TX 75080-3021. E-mail: yiorgos.makris@utdallas.edu.*

Manuscript received 29 Sept. 2011; revised 2 Feb. 2012; accepted 4 Mar. 2012; published online 28 Mar. 2012.
Recommended for acceptance by M. Hsiao.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-09-0698.
Digital Object Identifier no. 10.1109/TC.2012.81.

method, followed by an actual CED implementation on the FPU of a modern microprocessor, which is presented in Section 5. Section 6 describes the development of the simulation-based experimental infrastructure. The merit figures of the proposed method, namely the attained coverage and incurred overhead, are assessed in Section 7, followed by a discussion of its role in developing a cost-effective CED for the entire FPU in Section 8.

2 Error Detection in FPUs

Several error detection methods have been proposed for protecting FPUs. Most of them, however, target the data path and have been ported from the corresponding techniques for integer arithmetic, while methods specifically designed to protect the FPU control logic have yet to be developed.

2.1 Data Path

The most popular technique for reliable arithmetic operations is residue codes [20], [21], [22]. Low-cost residue codes are single arithmetic error detecting codes with unidirectional error detecting capabilities. The efficiency of residue codes depends on the selection of the check base b. The higher the base, the more errors the code will detect, yet also the more expensive the hardware overhead which will be incurred. Popular base selections are $b = 15$ (4 bits) and $b = 3$ (2 bits). In both cases, the resulting *modulo* circuit is highly simplified and the theoretical error detection percentage is $1 - (1/2^4) = 93.4\%$ for $b = 15$ and $1 - (1/2^2) = 75\%$ for $b = 3$. Residue codes have been successfully applied to various designs [14], [23], [24].

Other techniques include Berger codes [25], [26] and two-rail checkers [23], [27]. Berger codes are optimal unidirectional error detecting codes. ALUs using Berger-encoded operands have been shown to be strongly fault secure [14], [18].

Finally, besides arithmetic codes, custom solutions have been proposed to protect the FPU data path. In [13], a reduced precision checker has been added to determine whether the result is correct within some error bound. However, reduced precision checking has only been applied to the floating-point adder (FPA). In [15], the authors use the multiplier circuitry to detect errors that may happen in the divider. Similarly, this technique applies specifically to the floating-point divider (FPD).

2.2 Control Logic

The simplest and most straightforward CED solution for random logic, such as the control logic of the FPU, is duplication [16]. The main advantage of duplication is simplicity and applicability to any given design. However, the >100 percent area overhead (including the comparators) and the extra delay required for checking make duplication less appealing for modern FPUs. Furthermore, control in modern, pipelined FPUs is distributed across multiple components, necessitating manual and tedious effort to identify and replicate it.

TMR [17] has similar properties to duplication, with the added advantage of error correction. However, the hardware overhead of >200 percent (including the comparators and the voter) makes it prohibitive for commercial designs.

TABLE 1
IEEE-754 Floating-Point Value Layout

Precision	Bits	Sign	Exponent	Fraction	Bias
Single	32	1 [31]	8 [30-23]	23 [22-0]	127
Double	64	1 [63]	11 [62-52]	52 [51-0]	1023
Quad	128	1 [127]	15 [126-112]	112 [111-0]	16383

3 Floating-Point Arithmetic

The IEEE-754 standard is the most commonly used floating-point representation for real numbers on computers. Many computer languages allow or require that some or all arithmetic be carried out using IEEE-754 formats and operations. This section provides a very brief overview of the standard, to assist in better understanding the proposed method. Extensive information on floating-point arithmetic can be found in [28], [29].

Floating-point representation essentially represents real numbers in scientific notation. Scientific notation expresses numbers as a base and an exponent. For example, 123.45 could be represented as $1.2345 * 10^2$ or $12.345 * 10^1$. IEEE 754 floating-point numbers have three components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (i.e., 1). The exponent base (i.e., 2) is also implicit and need not be stored. Thus, IEEE-754 floating-point numbers are represented as $(-1)^{sign} * 1.fraction * 2^{exponent}$. Table 1 shows the layout (the number of bits and the bit ranges for each field) for the most commonly used precisions. Quadruple precision is rarely implemented in hardware and is usually performed by software traps.

The sign bit is 0 for positive numbers and 1 for negative numbers. The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent to get the stored exponent. For single-precision floating-point numbers, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. The mantissa represents the precision bits of the number. To maximize the quantity of representable numbers, floating-point numbers are stored in normalized form. This basically puts the radix point after the first nonzero digit. For example, 123.45 in floating point would be $(-1)^0 * 1.9289062 * 2^6$ in single precision. Consequently, the stored representation would be $42F6E666_{16}$.

Finally, exponent field values of all 0s and all 1s are reserved by IEEE to denote special values in the floating-point scheme. Zero is a special value denoted with an exponent field of zero and a fraction field of zero. While -0 and $+0$ are distinct values due to the extra sign bit, they both compare as equal. The values $+\infty$ and $-\infty$ are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in the IEEE-754 standard. Finally, the value not a number (NaN) is used to represent a value that does not represent a real number. NaN is represented by a bit pattern with an exponent of all 1s and a nonzero fraction.

There are two categories of NaN: quiet NaN (QNaN, most significant fraction bit set, propagates freely through most arithmetic operations) and signaling NaN (SNaN, most significant fraction bit clear, signals an exception when used in operations). Semantically, QNaN denotes indeterminate operations, while SNaN denotes invalid operations.

4 Proposed CED Method

Our method is based on the conjecture that an error in the control logic will lead to extensive data path corruption, which will propagate to the exponent part of the floating-point representation. Numerous examples can be provided to show the impact of control errors on the exponent and justify our approach:

- *Special case control:* The control logic identifies whether the input operands are NaN, infinity, 0, and so on. Mishandling of special cases due to errors will result in a completely different output with an incorrect exponent. For example, any operation with NaN results in a NaN. If the control logic mistreats a normal operand as NaN, then the operation $3 * 5$ will result in NaN ($\text{exponent} = 255$) instead of 15 ($\text{exponent} = 130$).
- *Algorithm stage control:* All floating-point operations go through several stages before generating the final results. In case a stage is skipped or repeated (e.g., one more or one less division round is performed) due to a control error, the result will be incorrect and is very likely to be reflected in the exponent.
- *Select lines:* Control logic is responsible for driving the correct operands to the data path. In case an error occurs and the control drives a 0 instead of a 7, the operation $7 * 18$ will result in a 0 ($\text{exponent} = 0$) instead of 126 ($\text{exponent} = 133$).
- *Operation control:* Along with the operands, the control logic also drives the signals for the operation selection. Therefore, if due to an error the operation changes, say from addition to subtraction, then the operation $2.0 + 1.9$ will result in 0.1 (exponent of 123) instead of 3.9 (exponent of 128).

These are only a few examples of data path corruption due to control logic errors, supporting our conjecture that errors in the FPU control logic can be detected by monitoring the data path. Since the exponent part of the data path is likely to be affected and fairly simple to calculate [28], we seek to develop a low-cost CED method for the control logic by predicting—through additional logic—the exponent part of the floating-point result and comparing to the actual value.

4.1 Calculating the Exponent

In this section, we discuss the exponent calculation for each of the three types of FPU functions, namely arithmetic operations, conversions, and other operations. We note that the exponent is calculated independently of the fraction operation; hence, the result is not exact because possible fraction normalization may affect the final value of the exponent.

4.1.1 Arithmetic Operations

The first category is arithmetic operations, such as additions, subtractions, multiplications, and divisions. We remind that the IEEE-754 representation of normalized floating-point operands is $(-1)^s * 1.f * 2^e$, where s is the sign, f is the fraction, and e the exponent. Thus, exponent calculation in multiplication and division is simple, i.e.,

$$\begin{aligned}&\left((-1)^{s_1} * 1.f_1 * 2^{e_1}\right) * \left((-1)^{s_2} * 1.f_2 * 2^{e_2}\right)\\ &\quad = (-1)^{s_1+s_2} * 1.f_1 * 1.f_2 * 2^{e_1+e_2}\end{aligned} \tag{1}$$

for multiplication and

$$\begin{aligned}&\left((-1)^{s_1} * 1.f_1 * 2^{e_1}\right)/\left((-1)^{s_2} * 1.f_2 * 2^{e_2}\right)\\ &\quad = (-1)^{s_1+s_2} * 1.f_1/1.f_2 * 2^{e_1-e_2}\end{aligned} \tag{2}$$

for division. So, we simply need to add (for multiplication) or subtract (for division) the exponents, operations which can be performed by the same hardware structure. In case the fraction overflows and needs to be normalized, the exponent needs to be adjusted accordingly. Hence, for arithmetic operations, we can only predict the exponent of normalized results with a ± 1 accuracy. Consequently, if an erroneous result differs from the correct result by 1, error masking will occur. However, our conjecture is that in the presence of a control logic error, the data path is corrupted extensively; hence, the probability of such masking is very low. Indeed, the results presented in Section 7 corroborate this conjecture.

For addition and subtraction, the exponent is the largest of the two operand exponents; therefore, a simple comparator suffices to calculate it (similar to the multiplication/division cases, normalization may be required). This does not apply in the case of cancelation (i.e., when there is a subtraction of operands with equal exponents or exponents that differ by 1). In this case, the exponent can take a wide range of values and cannot be computed accurately without information from the fraction. To moderate cost, our CED method only checks if the operands are equal (so the result is zero) and ignores the rest of the cases. Nevertheless, the results shown in Section 7.6 indicate that the consequent error masking is very small.

4.1.2 Conversions

Another common operation performed in FPUs is conversion from/to integer/floating-point representations. The exponent of the results can be exactly calculated by appropriately offsetting the input operand. For floating-point precision conversions (single to double and vice versa), the exponent needs to be offset by ± 896, because the actual exponent is $e_s - 127$ in single precision and $e_d - 1{,}023$ in double precision. Thus, for single to double conversion, the exponent is $e_d = (e_s - 127) + 1{,}023$ and for double to single $e_s = (e_d - 1{,}023) + 127$.

4.1.3 Other Operations

Modern FPUs usually implement more operations, such as absolute value, negation, and comparison. In all these operations, the exponent is very simple to calculate. Negation/absolute value operations affect only the sign (i.e., the exponent is the same). Comparison operation

TABLE 2
Sample Floating-Point Instructions

Operation	Result Exponent	Fraction Normal.	Sign
Addition	$max(e_1, e_2)$	Yes	$sign(max(e_1, e_2))$
Subtraction	$max(e_1, e_2)$[1]	Yes	$sign(max(e_1, e_2))$
Multiplication	$e_1 + e_2$	Yes	$sign(e_1) \oplus sign(e_2)$
Division	$e_1 - e_2$	Yes	$sign(e_1) \oplus sign(e_2)$
Single to Double	$e_1 + 896$	No	$sign(e_1)$
Double to Single	$e_1 - 896$	No	$sign(e_1)$
Negation	e_1	No	$-sign(e_1)$
Absolute Value	e_1	No	$+$

[1] *Equal or different-by-1 exponents may lead to cancelation.*

results are implementation specific, as the output result is the comparison result and not a floating-point number. For example, SPARC ISA defines the exponent field of the output as 0, and the comparison result is stored in the flags field. Table 2 summarizes the exponent operation for common FPU operations.

4.2 Sign Calculation and Flags

Using the hardware present for exponent calculation, the sign bit of the result can also be calculated, thus expanding the error detecting capability of the presented method. Table 2 shows the sign bit for a sample instruction set. Therefore, we can include the sign bit to form an extended signature which our method checks against.

Similarly, most modern FPUs have an extra flags field, which we can use to expand the aforementioned signature. For example, as discussed in Section 4.1.3, the comparison operation in both SPARC ISA and Intel x87 ISA stores the result in the flags field (2 bits for SPARC, 4 bits for x87). Since we already have comparison operations to determine the result exponent for addition/subtraction, those flags can easily be generated without extra overhead. Furthermore, the x87 defines flags for 0, NaN, and infinity results that can also be extracted from the CED hardware.

4.3 Error Recovery

The proposed CED methodology is nonintrusive and operates independently of the FPU. Error signaling occurs through comparison after the instruction is executed; thus, on-the-fly correction of the result is not possible. However, the CED successfully pinpoints the corrupted instruction and the FPU may easily recover via reissuing. Reissuing can be performed either at the software level (FPU signals invalid operation) or completely transparently at the hardware level (by adding extra hardware). We also note that while recovery from a transient error can be achieved via reissuing, repeated failure of an instruction indicates a failing unit which should be replaced or substituted via software traps.

5 CED Implementation

In this section, we present an actual implementation of the proposed CED methodology. Since our method can be easily applied to any IEEE-754 compliant FPU, we selected the latest open-source FPU implementation provided by Sun through the openSPARC project.

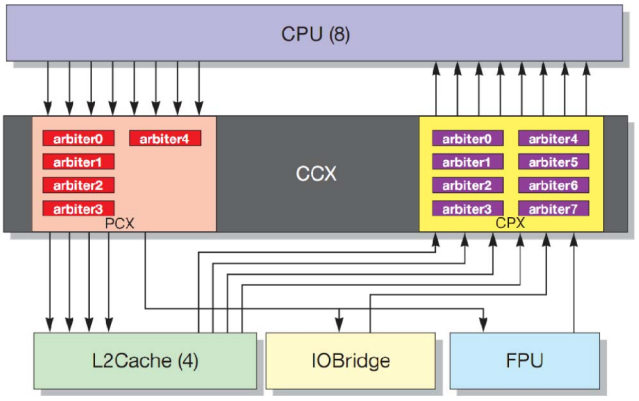

Fig. 1. T1 PCX—FPU—CPX interface [31].

5.1 Test Vehicle

The test vehicle of our study is the register transfer-level (RTL) model of the openSPARC T1 microprocessor [30], the open source version of the UltraSPARC T1 microprocessor.

The openSPARC T1 processor has eight SPARC processor cores which have full hardware support for four threads. Thus, there can be up to 32 threads running in parallel, greatly increasing throughput. Each of the cores has an instruction and a data cache, as well as an instruction and a data translation look-aside buffer (TLB). The eight cores are connected through a crossbar (CCX) to an on-chip L2-cache, as well as the J-Bus controller (IOBridge) and the shared FPU.

Each SPARC CPU core can send a packet to the FPU, using the cache-processor crossbar (CPX). Conversely, the FPU can send a packet to any one of the eight cores using the processor-cache crossbar (PCX). Fig. 1 presents the communication between the FPU and the eight SPARC cores.

A floating-point instruction is delivered from the cores to the FPU in either one- (single-operand instructions) or two-packet transfer. One source operand is transferred in each cycle and the crossbar always provides a two-cycle transfer. In case of single-operand instructions, an invalid transfer is produced in the second cycle [31].

Since the FPU is a single-shared resource, each of the eight SPARC cores may have a maximum of one FPU instruction waiting to be executed. Thus, the FPU can hold up to eight instructions at a given time. The FPU implements the SPARC V9 floating-point instruction set, and is fully compliant with the IEEE 754 standard [32]. The floating-point register file and floating-point state register are in the SPARC core floating-point front-end unit, which is unique for every core (and not shared like the FPU).

The FPU includes three execution pipelines:

- *FPA*: executes additions, subtractions, comparisons, and conversions.
- *Floating-point multiplier (FPM)*: executes multiplications.
- *FPD*: executes divisions.

Incoming instructions are stored in a 16 entry $\times 155$ bit FIFO queue (unless the FIFO is empty, in which case it is bypassed). In each cycle, one instruction may be issued from the FIFO and one instruction may complete and exit the FPU.

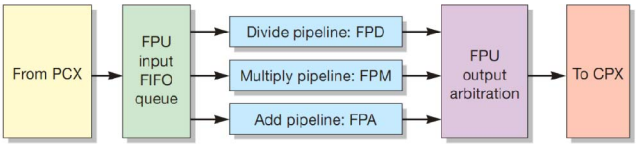

Fig. 2. T1 FPU functional block diagram [31].

Fig. 2 shows a block diagram with the three independent pipelines. NaN source propagation is supported by steering the NaN source through the pipeline to the result:

- *Input FIFO queue:* The FIFO stores the instructions waiting to be executed. For single-source instructions, the first operand slot in the FIFO is forced to zero. The unused region of a single-precision instruction is also forced to zero. The FIFO also stores five tag bits generated by the source operands. These tag bits store information about special cases (zero fraction, zero exponent, etc.).
  Eight entries, issued in FIFO order, are dedicated to the FPA and FPM pipelines. Similarly, there are eight entries for the FPD pipeline, having issue priority over the FPA/FPM entries.
- *Adder pipeline:* The adder is fully pipelined (four stages), but certain integer conversions require a final pass through the final stage. The FPA pipeline has a latency of 4 clock cycles (or 5 for integer conversions).
- *Multiplier pipeline:* The multiply pipeline has six pipeline stages and all instructions have a fixed latency of 7 clock cycles.
- *Divider pipeline*: Division is executed in a nonblocking, dedicated data path with seven pipeline stages. Latency varies between 7 and 32 cycles for single-precision operations and 61 cycles for double precision.
- *Output arbitration:* Only one instruction can exit the FPU per cycle. The FPD pipeline has priority over the FPA and FPM pipelines, which, in turn are prioritized in a round-robin fashion. If an instruction is stalled at the final stage of a pipeline due to output arbitration, the corresponding input FIFO queue will not issue a new instruction.

5.2 Implementing CED

Table 3 summarizes the instructions that the T1 FPU implements. The pipeline where each of these instructions is executed is also listed in the third column of the table, with most instructions executed in the Add pipeline.

TABLE 3
openSPARC T1 Floating-Point Instructions

Mnemonic	Description	Pipe
`FADD(s,d)`	Addition	FPA
`FSUB(s,d)`	Subtraction	FPA
`FMUL(s,d),FsMULd`	Multiplication	FPM
`FDIV(s,d)`	Division	FPD
`FCMP[E](s,d)`	Comparison	FPA
`FsTOd`	Single To Double	FPA
`FdTOs`	Double To Single	FPA
`F(i,x)TOs`	Integer/Long to Single	FPA
`F(i,x)TOd`	Integer/Long to Double	FPA
`F(s,d)TO(i,x)`	Single/Double To Integer/Long	FPA

Since the FPU provides support for all IEEE-754 floating-data types, including NaN, zero, and infinity, special care must be exercised during exponent calculation in these cases. Table 4 summarizes all combinations involving special numbers. The exponent field suffices to determine whether an operand is NaN or infinity (i.e., all bits are 1, symbolized as MAX_EXP) but, for zero, a comparison is also required to determine whether the mantissa bits are zero. As can be seen in Table 4, in most cases involving a special number the resulting exponent is a MAX_EXP. There are, however, a few combinations, such as num division by infinity or some operations with 0, where this is not the case.

As discussed in Section 4.2, given the added hardware to calculate the exponent, we can also calculate the comparison flag fields of the output packet, further increasing the efficiency of the proposed CED. Specifically, there are 5 bits that can be calculated: The 2-bit `Fcc_flags`, where the field is `00` when the two operands are equal, `01` when the first operand is greater, `10` when the second operand is greater, and `11` when the two operands are unordered (in case of NaNs). Besides the 2-bit `Fcc_flags`, the `FCMP` bit is 1 when there is a comparison operation, and the `PCX_flags` are copied from the respective field of the input packet.

5.2.1 Signature Calculation and Validation

The 11-bit exponent, along with the 1-bit sign field and the 5-bit flag field {`Fcc_flags, FCMP, PCX_flags`}, form an extended 17-bit signature which serves as the basis for our CED method. Specifically, the predicted signature of an FPU instruction is compared to the actual signature of the outgoing FPU packet when the instruction is executed. Fig. 3 shows the block diagram of the proposed CED method, with the details of the signature calculation block

TABLE 4
Floating-Point Special Cases (Add/Sub/Mul/Div)

op2/op1	NaN	+∞	-∞	+0	-0	+num	-num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+∞	NaN	+∞/NaN +∞/NaN	NaN/-∞ -∞/NaN	+∞/-∞ NaN/+0	+∞/-∞ NaN/-0	+∞/-∞ +∞/+0	+∞/-∞ -∞/-0
-∞	NaN	NaN/+∞ -∞/NaN	-∞/NaN +∞/NaN	-∞/+∞ NaN/-0	-∞/+∞ NaN/+0	-∞/+∞ -∞/-0	-∞/+∞ +∞/+0
+0	NaN	+∞/+∞ NaN/+∞	-∞/-∞ NaN/-∞	+0/+0 +0/NaN	+0/-0 -0/NaN	+num/+num +0/+∞	-num/-num -0/-∞
-0	NaN	+∞/+∞ NaN/-∞	-∞/-∞ NaN/+∞	+0/+0 +0/NaN	-0/+0 -0/NaN	+num/+num +0/-∞	-num/-num -0/+∞

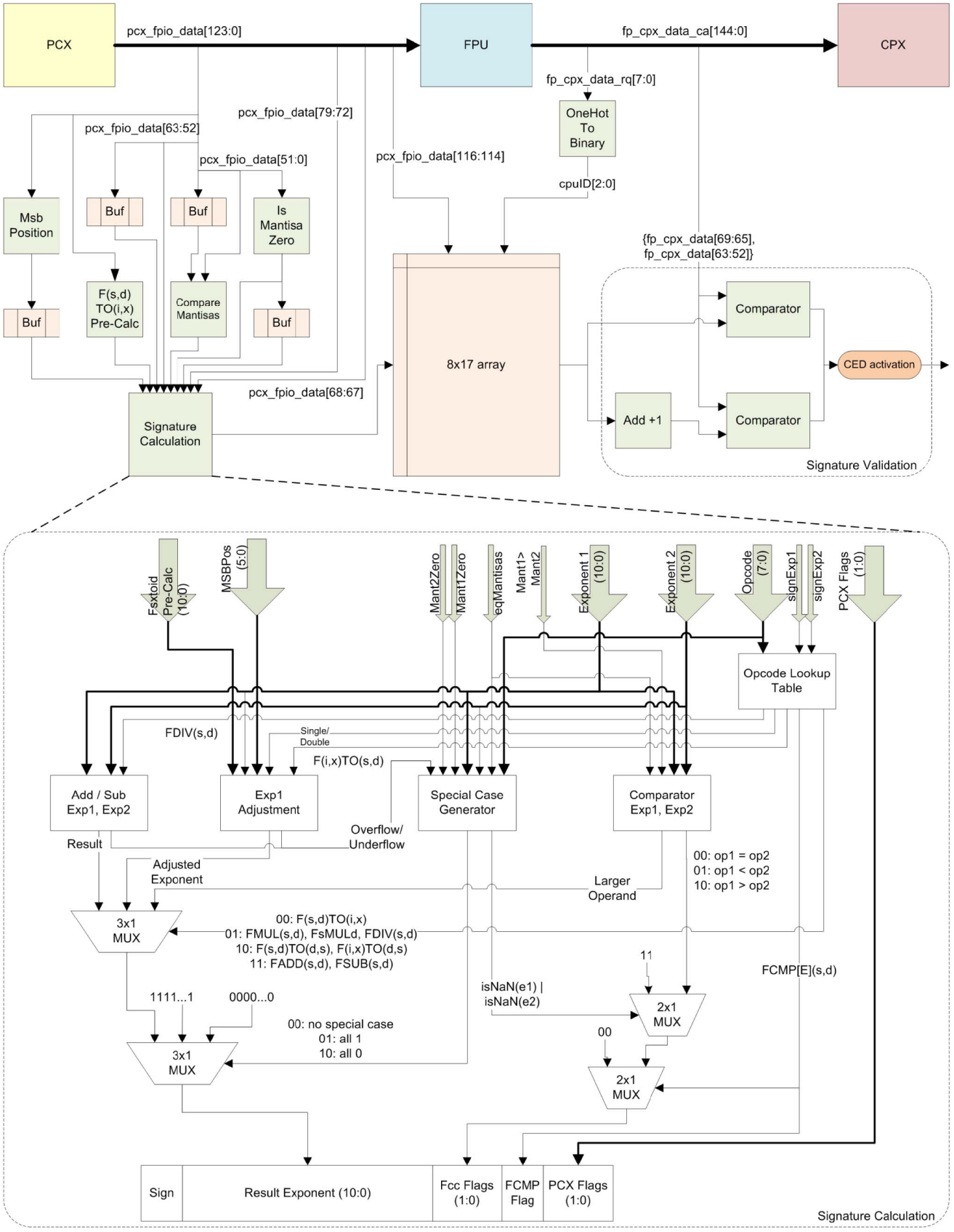

Fig. 3. Block diagram of CED and signature calculation.

expanded in the bottom part of the figure. Besides the exponent calculation rules described in the previous section, we point out that in the case of single-precision operands, the three most significant bits of the exponent field in the predicted signature are set to zero, as is also done in the corresponding packets exiting the FPU. As for predicting

the flag fields, the only additional component is a mantissa comparator, which is necessary when the exponents are equal. Overall, the inputs to the signature calculator are the exponents, the operation code, the mantissa comparison result, the MSB position of the operand, the precalculated "exponent" field of the `F(s,d)TO(i,x)` result, and the `PCX_flags`, which are copied from the input packet.

Given that the exponent can be predicted with a ± 1 accuracy, validating the signature of an instruction leaving the FPU requires a twofold comparison, as indicated in the right part of Fig. 3. Specifically, we compare both the calculated exponent and its incremented-by-one version to the actual exponent of the instruction exiting the FPU and we only activate the CED output if both comparisons fail. For this scheme to be correct, we need to always store the smaller of the two possible exponent values as the calculated signature. Therefore, for subtraction and division operations, where mantissa normalization may decrement the result exponent, we decrement our calculated exponent by 1. Thus, the final signature will be checked for its actual value and its incremented-by-1 value. This eliminates the need for an additional subtractor and comparator at the signature validation stage of the CED method.

5.2.2 Comparison Synchronization

The FPU is a shared resource with multiple floating-point instructions in flight. Moreover, the latency of some floating-point instructions (i.e., division) is variable and cannot be predicted a priori. Hence, it is not possible for our CED method to predict which instruction should exit the FPU at each time. Instead, the signatures calculated for each incoming instructions are stored in an array which is indexed using the CPU ID of the outgoing instruction. We point out that the FPU outputs the CPU ID in one-hot encoding; this needs to be converted before accessing the proper entry in the buffer. The CPU ID is a unique identifier because each of the eight cores can have a maximum of one outstanding FPU instruction. A thread with an outstanding FPU instruction switches out while waiting for the FPU result. This allows up to eight instructions to be in the FPU. Therefore, storing the signatures requires an 8 entry $\times 17$ bit memory with 1 read and 1 write port. We do not need more ports because only one instruction can arrive at the FPU within two cycles, and only one instruction can exit the FPU in a given clock cycle.

Table 5 shows the information extracted from the incoming (PCX) and the outgoing (CPX) packets to assist in signature calculation and comparison synchronization. Most floating-point instructions require a two-packet transfer; hence, buffers are added to hold the information necessary from the first incoming packet before we can correctly calculate the exponent. Fig. 3 puts everything together, showing all the CED modules.[1]

Finally, we would like to note that the proposed CED method operates at the boundary of the FPU and validates the signature of the final output. Hence, unlike CED which is based on duplication and comparison of control logic blocks within the FPU, our method has no false alarms.

1. To avoid cluttering the figure, the CED shows the exponent of a double-precision operand. In case of single precision, the exponent bits are driven accordingly.

TABLE 5
PCX/CPX Packet Format

	Bits	Description
PCX	123	Packet valid bit
	122:118	PCX packet request type
	116:144	CPU ID
	113:112	Thread ID
	79:72	Operation code
	68:67	Condition codes FSR_fcc
	65:64	Rounding mode
	63:0	Floating point operand
CPX	144	Packet valid bit
	143:140	CPX packet return type
	135:134	Thread ID
	76:72	Floating-point flags
	69	Compare operation
	68:67	Condition codes
	66:65	Condition codes from PCX
	63:0	Floating point result

6 Experimental Setup

To assess the effectiveness of our method, we built an extensive simulation infrastructure to perform error injection experiments. Since our target is transient errors, we need to perform a large number of injections; therefore, the infrastructure must support very fast simulations and error impact evaluation.

6.1 Experiment Flow

Fig. 4 shows the data flow of our experiments. Two different types of workload are used during the evaluation:

- *Synthetic benchmarks:* A python-based assembly generator was developed to generate multithreaded assembly utilizing floating-point instructions. The assembly generator can generate up to 32 different assembly files, one for each thread. The user can specify the percentage of the floating-point instructions in the file, as well as the desired number of

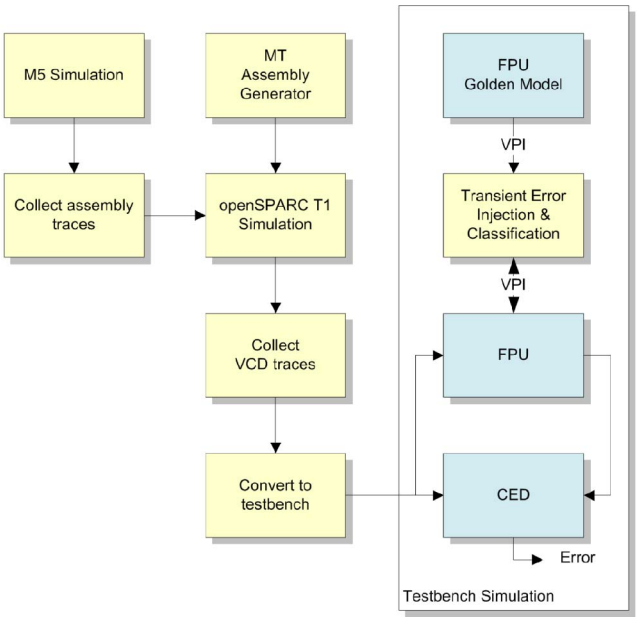

Fig. 4. Simulation infrastructure.

instructions for each pipeline (FPA, FPM, FPD). Floating-point registers are randomly selected for each instruction. Ten of the registers contain special values (NaNs, Inf, 0) to ensure operations with special numbers. Synthetic benchmarks ensure utilization of all resources and appearance of every possible special case.

- *SPEC2006FP benchmarks:* To accurately evaluate the in-field performance of our CED method, SPEC2006FP benchmarks are used. Since the RTL model of the openSPARC does not support system calls (because no operating system is loaded), the M5 simulator [33] is used to execute the SPEC benchmarks. The execution trace is collected, distributed to different threads, and then loaded to the RTL similarly to synthetic benchmarks.

The chosen workload is simulated in the openSPARC T1 environment using `sims`, and value change dump traces are collected at the input of the FPU. These traces are then converted to a separate test bench using Synopsys `vcat`. This test bench can be simulated using Synopsys `vcs` without the need to simulate the entire microprocessor model.

Statistical fault injection (SFI) is used to evaluate the effectiveness of the CED against soft errors. Recent work [34] has showed that SFI measurements closely match the results of the actual proton and neutron irradiation of a chip, strengthening the quality of our evaluation. Error injection is performed during simulation by mutating the microprocessor model using the parallel saboteurs technique. An extensive description of the RTL error injection method can be found in [35].

The transient error injection is controlled by a python script through verilog procedural interface calls. The same script is used for error classification, with the help of a golden model that runs in parallel with the injected FPU model.

6.2 Hardware Synthesis

To provide hardware overhead estimates, we synthesized the FPU model using Synopsys Design Compiler targeting a 90 nm library. The timing constraints were set to a clock period of 1 GHz, to match the running speed of the UltraSPARC T1. The total area of the synthesized FPU is $816{,}660\ \mu m^2$ ($587{,}947\ \mu m^2$ combinational, $181{,}110\ \mu m^2$ non-combinational, and $47{,}601\ \mu m^2$ net interconnect area). Fig. 5 shows the hierarchy of the FPU along with the area percentage of each main module. The largest module is the 54×54 multiplier. The division pipeline is rather small (and, naturally, rather slow at the same time). The model also contains a few more very small modules, such as repeaters (to optimize timing) and scan-control modules, which are not shown in the figure. These modules along with the pipeline registers and the wiring add up to the remaining 23.4 percent of the FPU area.

6.3 Control Duplication

The simplest and most straightforward CED solution for random logic, such as the control logic of the FPU, is duplication. The control modules in the openSPARC T1 FPU are relatively small. Therefore, blindly duplicating these modules and continuously monitoring the outputs of each pair of duplicates through a comparator are an

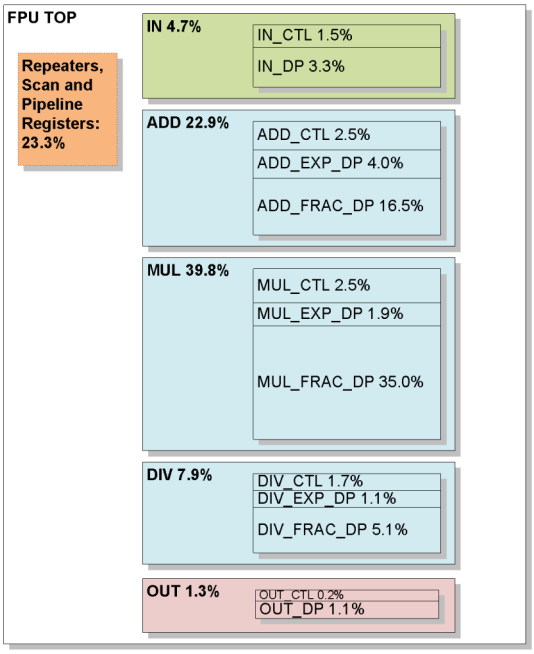

Fig. 5. FPU hierarchy and area breakdown.

appealing option. However, there are two limitations in this approach:

- A transient error in a control module monitored through duplication will trigger the CED output even if the module is not actively participating in instruction execution at the time of the strike, or if the error is later masked by the subsequent FPU logic and never reaches the output. In essence, duplication-based CED of the distributed control modules may cause false alarms.
- Control discrepancies do not only originate from errors in modules that are considered pure control but may also occur as a result of errors in other parts of the FPU, such as the `In` and `Out` modules or the pipeline registers between the various stages. Such control discrepancies will not be detected by a distributed duplication-based control module CED method.

The latter is, indeed, a serious limitation. The experimental results presented in Section 7 demonstrate that only 43 percent of the control errors affecting the outgoing packet originate from the pipeline controller modules. This happens mainly because parts of the `In` and `Out` modules, which are part of the data path, carry crucial information that eventually propagates to the corresponding control logic of the FPU pipelines (e.g., whether the exponent is zero or all ones, whether the fraction is zero, etc.). The same applies to the registers between the pipeline stages. The corresponding control errors are even more crucial to detect than those originating from the clearly marked control modules. The reason for this is that the `In` and `Out` modules are always used for every incoming instruction; thus, errors there have a high probability of propagating to the output. Similar arguments hold for the pipeline registers and repeaters, which are part of the information flow in the CPU and may also cause control errors.

TABLE 6
Error Classification Statistics

Module	Non-masked errors					
	FP-100	FP-1	bwaves	milc	zeusmp	leslie3d
fpu_top	2.78%	2.01%	2.65%	3.51%	3.40%	2.63%
in	3.97%	2.79%	3.89%	6.97%	6.48%	5.12%
in_ctl	12.1%	11.0%	8.46%	8.60%	7.48%	6.71%
in_dp	2.78%	1.87%	2.86%	5.46%	5.33%	4.39%
add	1.43%	1.28%	1.90%	5.28%	3.91%	2.47%
add_ctl	2.04%	1.76%	1.85%	5.31%	4.62%	2.71%
add_exp	0.44%	0.57%	0.58%	4.07%	2.36%	1.37%
add_frac	0.59%	0.43%	1.15%	2.79%	2.68%	1.24%
mul	1.98%	1.31%	1.82%	0.36%	0.69%	2.23%
mul_ctl	3.71%	2.81%	2.94%	1.60%	1.97%	2.71%
mul_exp	2.29%	1.48%	1.81%	0.10%	0.35%	2.15%
mul_frac	1.60%	0.89%	0.51%	0.03%	0.28%	2.62%
div	7.70%	5.51%	3.25%	0.35%	0.90%	1.24%
div_ctl	11.9%	9.53%	5.71%	2.31%	1.94%	2.80%
div_exp	4.50%	3.34%	2.14%	0.00%	0.00%	0.21%
div_frac	4.45%	3.83%	3.28%	0.01%	0.16%	1.14%
out	4.74%	3.49%	5.28%	5.64%	5.65%	4.23%
out_ctl	16.4%	15.8%	15.6%	16.8%	16.3%	16.2%
out_dp	4.37%	3.06%	4.65%	5.32%	4.75%	4.37%
Total	**2.13%**	**1.45%**	**1.40%**	**1.51%**	**1.53%**	**2.55%**

Based on the above discussion, in order for a duplication-based CED scheme to effectively detect all control errors, it needs to incorporate not only the clearly marked control modules but also every location wherein an error will result in a control discrepancy. The overhead of such a duplication-based scheme for detecting all control errors in the openSPARC T1 FPU is 17.9 percent.

7 CED Evaluation

This section describes the experimental results that support our conjectures. We simulated the openSPARC T1 microprocessor using different types of workload. First, two sets of synthetic workload: The first one (`FP-100`) consists of 100 percent floating-point operations, to resemble applications with intense need for floating-point calculations. The second (`FP-1`) consists of 1 percent floating-point instructions, matching the profile of common applications that place very little demand on the FPU. Furthermore, four SPEC2006FP benchmarks, namely `bwaves`, `milc`, `zeusmp`, and `leslie3d` were utilized to evaluate the effectiveness of the proposed method using real workload. Seven million transient errors were injected in total, uniformly distributed over workload, time, and location across the FPU.

7.1 FPU Error Impact Analysis

Error masking: The first set of results, shown in Table 6, present statistics regarding the impact of injected errors on the FPU output. As expected in a transient error injection campaign, masking is very high; indeed, among the injected errors, only 1.76 percent on average reach the FPU output (i.e., nonmasked errors).

The key observation from these results is that the average nonmasked error rate of control modules (`_CTL`) is 8.9 percent (i.e., one out of nine transients affects the FPU), a percentage that is much higher than the average nonmasked rate of data path modules, which is only 1.43 percent. In other words, errors in the control logic are six times more likely to affect the output than errors in the data path. This statement supports our claim that despite the control logic being smaller, protecting it is necessary for ensuring reliable FPU operation.

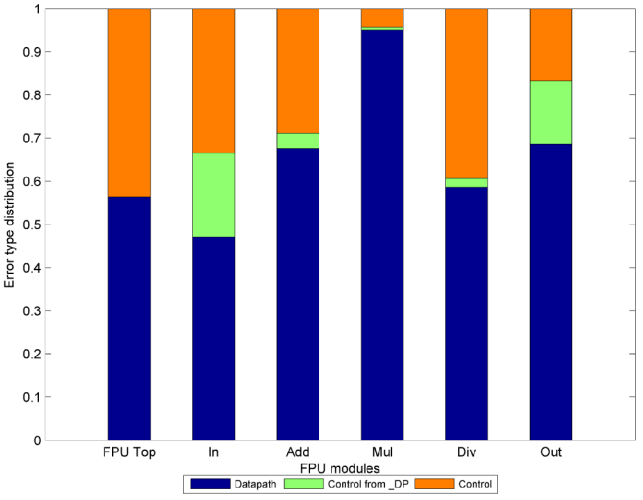

Fig. 6. Error classification in FPU modules.

We also point out that the percentage of nonmasked errors varies among the different control modules, with `in_ctl`, `div_ctl`, and `out_ctl` having higher percentages. This is expected for the `in_ctl` and `out_ctl` modules, because all instructions have to go through them and they always contain critical information regarding instruction execution. As for `div_ctl`, division instructions have 10 times more latency than other instructions (up to 61 cycles), so the probability that the FPD pipeline will be occupied by a valid instruction during the workload execution is much higher.

The necessity to carve out the entire control logic throughout the FPU is highlighted by the output error classification results presented in Fig. 6. Data path errors are defined as errors in the data path modules affecting the final result, with everything else defined as a control error. It is clear that for `In` and `Out` modules, many of the control faults originate from the `_DP` units. Thus, as discussed in Section 6.3, these parts have to be included in an efficient duplication scheme.

Finally, among the different benchmarks, the average number of corrupted bits in the output packet for control logic errors is 5 bits. This supports our conjecture that control logic errors lead to extensive data path corruption.

7.2 CED Fault Coverage

The coverage achieved by our CED method for control errors, as shown in Table 7, is 93.8 percent on average for the given workloads. The proposed CED method does not reach 100 percent coverage for three reasons: First, some level of error masking occurs due to the ± 1 comparison of the output exponent (around 1.9 percent). Second, some control logic errors affect only the fraction portion of the result and never propagate to the exponent (around 3.7 percent). Finally, cancelation counts for the rest of the missed faults.

In comparison, duplication-based CED, as described in Section 6.3, offers 100 percent coverage, yet with possible

TABLE 7
Exponent Monitoring Coverage

Workload	Control Logic Fault Coverage
FP-1	95.7%
FP-100	94.5%
bwaves	95.4%
milc	91.7%
zeusmp	93.3%
leslie3d	92.2%
Average	93.8%

false alarms and at a much higher cost, as discussed in Section 7.3. However, exponent monitoring provides the ancillary benefit of also covering 98.1 percent of the errors that affect the exponent, which control logic duplication is unable to detect. The implication of this ancillary benefit is that no additional CED method is needed to cover the exponent portion.

7.3 Area and Delay Overhead

The second set of results, shown in Table 8, discuss and compare the area and delay overhead of our proposed exponent monitoring method to traditional duplication for performing CED in the FPU control logic.

In terms of area overhead, the cost of duplication is 17.9 percent of the FPU size, of which 16.1 percent is to replicate the control logic and 1.8 percent is to compare. In contrast, the proposed exponent monitoring incurs only one-third of this cost, for a total of 5.8 percent of the FPU size.

The proposed CED method operates in parallel with and is nonintrusive to the FPU functionality; therefore, the only added delay is for comparing the output packet signature to the one stored in the CED memory array. We note that since the CPU ID is broadcasted one clock cycle before the packet exits the FPU, the CED extracts the correct signature from the memory in advance. Thus, no delay overhead is added by the exponent monitoring method. In contrast, duplication-based CED incurs a nontrivial delay overhead while comparing the control logic output at each pipeline stage.

7.4 Power Overhead

Estimates of power overhead per benchmark are presented in Fig. 7. The estimates are calculated using Synopsys PrimeTime, representing averaged power analysis based on toggle rates.

The power overhead of the proposed method is approximately three times less than control duplication (8.3 percent versus 24.7 percent). Besides the reduced area footprint, an additional advantage of the exponent monitoring method,

TABLE 8
Exponent Monitoring versus Duplication

FPU Control CED Method	Area Overhead	Coverage of Control	Coverage of Exponent	Coverage of Fraction
Duplication	17.9%	100%	-	-
Monitor Exponent	5.8%	93.8%	98.1%	15%

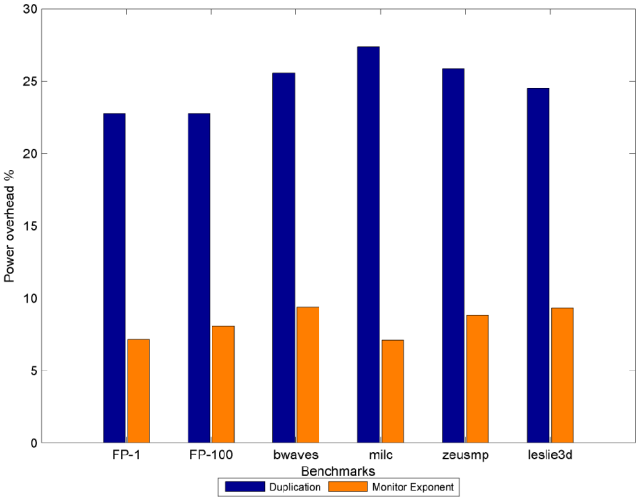

Fig. 7. Power overhead.

which contributes to the lower power consumption result, is the fact that it is only active during issue and commit of the floating-point operation. In contrast, a control logic duplicate is active throughout all stages of the floating-point operation (up to 61 cycles for double-precision divisions). Given the extensive efforts to reduce power in modern designs, exponent monitoring constitutes a very appealing option for a low-power error detection mechanism.

7.5 Error Recovery

Since our CED method operates independently of the FPU, the erroneous instruction can be easily detected and reissued. Two different implementations can be used:

1. *Software trap*: With no additional hardware overhead, the FPU may signal an exception for an erroneous result (the IEEE-754 standard defines a precise exception model). The operating system will be responsible for clearing the exception and reissuing the instruction. The SPARC T1 FPU has an `fp_exception_other` trap than can be used to notify the OS for detected errors.
2. *Hardware reissue*: In case that error containment is required, the FPU can store the instructions in a buffer and reissue the ones that were detected as erroneous. As expected, the buffer adds a significant amount of extra logic to the system, but error recovery is completely transparent to the user.

Fortunately, the SPARC T1 FPU already has a 16-entry FIFO for storing the incoming floating-point instructions. Thus, in our current implementation, we use this buffer for error correction by applying some minor modifications:

- An extra bit is added per entry, indicating that the instruction has been successfully executed. This extends the 155×16 FIFO to 156×16 bits.
- In case the FIFO is empty, the instruction bypasses the FIFO and goes directly to the corresponding pipeline. The design needs to be modified to force the instruction to always go through the FIFO as well as to the pipelines.

TABLE 9
Cancelation Analysis

Workload	Errors in Add/Sub	Possible Cancellation	Actual Cancellation	Equal Operands	Range Detection
bwaves	28.72%	14.39%	3.54%	13.19%	0.11%
milc	78.18%	0.05%	0.0%	0.01%	0.05%
zeusmp	66.83%	3.88%	3.64%	2.03%	0.36%
leslie3d	18.16%	5.42%	5.34%	2.23%	0.52%

All detected errors were corrected by our CED. Furthermore, the performance penalty for transparently correcting the erroneous results is trivial for all the given workloads, because a reissue penalty of up to 61 cycles (double-precision division) will have a negligible impact in a million cycles workload execution.

7.6 Cancelation

Finally, we assess the impact of cancelation on the effectiveness of the proposed CED method and we examine the utility of various strategies in ameliorating this problem. As shown in the third column of Table 9, the fault coverage loss due to cancelation ranges from 0 to 5.34 percent during execution of real-world heavy floating-point applications. Below, we compare four approaches for dealing with cancelation:

- *Disable CED if cancelation possible:* Whenever the exponents are the same or differ by 1, the CED is disabled. This is not an appealing option, because there are many cases where cancelation does not actually occur, leading to a coverage loss of up to 14.39 percent, as shown in the second column of Table 9.
- *Disable CED if cancelation detected:* This approach reduces the fault coverage loss down to less than 5.34 percent, but detecting cancelation requires information from the fraction, which comes at a much higher area overhead cost.
- *Disable CED if cancelation possible and equal operands:* In most of the cases, cancelation occurs from subtracting the same operands (with the result being 0). By simply subtracting the fourth from the second column of Table 9, we can extract that the cancelation attributed to nonequal operands is 1.2, 0.04, 1.85, and 3.19 percent for the four benchmarks respectively. Since the added CED hardware already has circuitry for detecting equal operands, this is the most appealing option and is what our CED method actually implements. There is no additional overhead, and the fault coverage loss due to cancelation is contained under 3.19 percent.
- *Detect invalid range of exponent:* Further fault coverage loss of up to 0.52 percent can be recovered through a mechanism for checking the validity of the exponent, as shown in the last column of Table 9. Validity may be checked based on the fact that the exponent, after cancelation, will lie within ± 63 of the calculated one. This option incurs extra overhead (i.e., adders and comparators) which does not easily justify the fault coverage gain of up to 0.52 percent; hence, we opted to omit it.

8 Cost-Effective CED for Entire FPU

We now examine the utility of exponent monitoring in developing a cost-effective CED method for the entire FPU. We note that, as previously shown in Table 8, exponent monitoring also detects around 15 percent of the errors that only affect the fraction portion of the floating-point representation, which the duplication method is unable to detect. Yet this percentage is very small, hence additional steps need to be taken to provide a complete FPU CED solution. To this end, we investigate how our method can be combined with base-15 residue code for the fraction, to reduce the overall CED cost for the FPU. Specifically, we compare three alternative scenarios: 1) using base-15 residue codes for the fraction and the exponent but leaving the control unprotected; 2) adding control logic duplication to the above solution; and 3) combining exponent monitoring with base-15 residue code only for the fraction (because the exponent is already covered).

The results reported in Table 10 demonstrate two key points: First, if control is left unprotected, the overall fault coverage would be a mere 59.2 percent. This shows, once again, that protection of control logic is necessary in modern FPUs. Second, the proposed exponent monitoring method enables a complete FPU CED solution that provides almost equivalent coverage to the duplication-based solution (i.e., 94.1 percent versus 96.2 percent), yet incurs almost half of the cost (i.e., 16.32 percent versus 29.78 percent), thereby constituting a very appealing option.

9 Conclusion

We presented a novel method for detecting transient errors in the control logic of a modern FPU. We demonstrated that errors in the control logic lead to an extensive corruption of the data path and, by extension, have a high probability of affecting the exponent field of the operation output. Therefore, independently calculating and validating the exponent of the outgoing packet provides very high coverage to such errors. As demonstrated on the openSPARC T1 processor, the proposed exponent monitoring-based CED method costs less than one-third of the cost of duplicating the control

TABLE 10
Comparison of CED Solutions for Entire FPU

Detection Method			Coverage				Hardware
Control	Exponent	Fraction	Control	Exponent	Fraction	Total	Overhead
-	Res-15	Res-15	0%	94.3%	94.3%	**59.2%**	**14.82%**
Duplication	Res-15	Res-15	100%	94.3%	94.3%	**96.2%**	**29.78%**
Exponent Monitoring		Res-15	93.8%	98.1%	94.3%	**94.1%**	**16.32%**

logic, while maintaining over 93 percent of its coverage. Moreover, in conjunction with a known residue code-based method for the fraction of the floating-point representation, it facilitates a complete CED solution which offers over 94 percent coverage for the entire FPU at the cost of 16.32 percent of its size, which to our knowledge, constitutes the most cost-effective approach to date.

REFERENCES

[1] Y. Savaria, N.C. Rumin, V.K. Agarwal, and J.F. Hayes, "Soft-Error Filtering: A Solution to the Reliability Problem of Future VLSI Digital Circuits," *Proc. IEEE,* vol. 74, no. 5, pp. 669-683, May 1986.

[2] H. Liang, P. Mishra, and K. Wu, "Error Correction On-Demand: A Low Power Register Transfer Level Concurrent Error Correction Technique," *IEEE Trans. Computers,* vol. 56, no. 2, pp. 243-252, Feb. 2007.

[3] A.H. Johnston, "Radiation Effects in Advanced Microelectronics Technologies," *IEEE Trans. Nuclear Science,* vol. 45, no. 3, pp. 1339-1354, June 1998.

[4] E. Normand, "Single Event Upset at Ground Level," *IEEE Trans. Nuclear Science,* vol. 43, no. 6, pp. 2742-2750, Dec. 1996.

[5] C. Metra, M. Favalli, and B. Ricco, "On-Line Detection of Logic Errors due to Crosstalk, Delay, and Transient Faults," *Proc. Int'l Test Conf.,* pp. 524-533, 1998.

[6] T. Karnik, P. Hazucha, and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 2, pp. 128-143, Apr.-June 2004.

[7] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Computers,* vol. C-31, no. 7, pp. 589-595, July 1982.

[8] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability,* vol. 51, no. 1, pp. 63-75, Mar. 2002.

[9] M. Mueller, L.C. Alves, W. Fischer, M.L. Fair, and I. Modi, "RAS Strategy for IBM S/390 G5 and G6," *IBM J. Research and Development,* vol. 43, no. 5, pp. 875-888, 1999.

[10] N. Quach, "High Availability and Reliability in the Itanium Processor," *IEEE Micro,* vol. 20, no. 5, pp. 61-69, Sept./Oct. 2000.

[11] J. Gaisler, "Concurrent Error-Detection and Modular Fault-Tolerance in a 32-Bit Processing Core for Embedded Space Flight Applications," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing,* pp. 128-130, 1994.

[12] A. Naini, A. Dhablania, W. James, and D.D. Sarma, "1 GHz HAL SPARC64R Dual Floating Point Unit with RAS Features," *Proc. IEEE Symp. Computer Arithmetic,* pp. 173-183, 2001.

[13] P. Eibl, A. Cook, and D. Sorin, "Reduced Precision Checking for a Floating Point Adder," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance of VLSI Systems,* pp. 145-152, 2009.

[14] J.C. Lo, "Reliable Floating-Point Arithmetic Algorithms for Error-Coded Operands," *IEEE Trans. Computers,* vol. 43, no. 4, pp. 400-412, Apr. 1994.

[15] S.M.H. Shekarian, A. Ejlali, and S.G. Miremadi, "A Low Power Error Detection Technique for Floating-Point Units in Embedded Applications," *Proc. IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing,* vol. 1, 2008.

[16] TEMIC, "TSC692E Floating-Point Unit User's Manual for Embedded Real Time 32 Bit Computer (ERC32)," 1996.

[17] W.L. Gallagher and E.E. Swartzlander, "Fault-Tolerant Newton-Raphson and Goldschmidt Dividers Using Time Shared TMR," *IEEE Trans. Computers,* vol. 49, no. 6, pp. 588-595, June 2000.

[18] J.C. Lo, S. Thanawastien, T.R.N. Rao, and M. Nicolaidis, "An SFS Berger Check Prediction ALU and Its Application To Self-Checking Processor Designs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 11, no. 4, pp. 525-540, Apr. 1992.

[19] G.G. Langdon and C.K. Tang, "Concurrent Error Detection for Group Look-Ahead Binary Adders," *IBM J. Research and Development,* vol. 14, no. 5, pp. 563-573, 1970.

[20] A. Avizienis, "Arithmetic Algorithms for Error-Coded Operands," *IEEE Trans. Computers,* vol. C-22, no. 6, pp. 567-572, June 1973.

[21] E. Kinoshita, H. Kosako, and Y. Kojima, "Floating-Point Arithmetic Algorithms in the Symmetric Residue Number System," *IEEE Trans. Computers,* vol. C-23, no. 1, pp. 9-20, Jan. 1974.

[22] A. Sasaki, "The Basis for Implementation of Additive Operations in the Residue Number System," *IEEE Trans. Computers,* vol. C-17, no. 11, pp. 1066-1073, Nov. 1968.

[23] D.A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-out-of-n Codes," *IEEE Trans. Computers,* vol. C-22, no. 3, pp. 263-269, Mar. 1973.

[24] A. Avizienis, G.C. Gilley, F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. Computers,* vol. C-20, no. 11, pp. 1312-1321, Nov. 1971.

[25] J.M. Berger, "A Note on Error Detection Codes for Asymmetric Channels," *Information and Control,* vol. 4, no. 1, pp. 68-73, 1961.

[26] M.A. Marouf and A.D. Friedman, "Design of Self-Checking Checkers for Berger Codes," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing,* vol. 8, pp. 179-184, 1978.

[27] M. Nicolaidis, "Self-Exercising Checkers for Unified Built-in Self-Test (UBIST)," *IEEE Trans. Computer-Aided Design,* vol. 8, no. 3, pp. 203-218, Mar. 1989.

[28] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys,* vol. 23, no. 1, pp. 5-48, 1991.

[29] W. Stallings, *Computer Organization and Architecture: Designing for Performance.* Prentice-Hall, 2009.

[30] Sun Microsystems, "OpenSPARC T1 Specifications," http://www.opensparc.net/opensparc-t1/index.html, 2013.

[31] Sun Microsystems, "OpenSPARC T1 Microarchitecture Specification," 2006.

[32] D. Stevenson et al., "IEEE Standard for Binary Floating Point Arithmetic," *ACM SIGPLAN Notices,* vol. 22, no. 2, pp. 9-25, 1987.

[33] S.K. Reinhardt, N. Binkert, A. Saidi, and R.D.K. Lim, "Using the M5 Simulator," *Proc. ISCA Tutorials and Workshops,* June 2005.

[34] P.N. Sanda, J.W. Kellington, P. Kudva, R. Kalla, R.B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C.R. Jones, "Soft-Error Resilience of the IBM POWER6 Processor," *IBM J. Research and Development,* vol. 52, no. 3, pp. 275-284, 2008.

[35] M. Maniatakos, N. Karimi, A. Jas, C. Tirumurti, and Y. Makris, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," *IEEE Trans. Computers,* vol. 60, no. 9, pp. 1260-1273, Sept. 2011.

Michail Maniatakos is an assistant professor of electrical and computer engineering at New York University Abu Dhabi. He is a 2012 PhD graduate of the Electrical Engineering Department at Yale University, where he also received an MSc in computer engineering in 2009. Prior to Yale, he received his BSc in computer science and MSc in embedded systems from the University of Piraeus in 2006 and 2007 respectively. His research interests include robust microprocessors, hardware security and heterogeneous microprocessor architectures. He is a member of IEEE.

Prabhakar Kudva received the PhD degree in computer science from the University of Utah. He is a research staff member in the Reliability- and Power-Aware Microarchitectures Department, IBM T.J. Watson Research Center. His research interests include reliability and power-aware architectures, as well as high-end system and ASIC design methodologies. He is a senior member of the IEEE.

Bruce M. Fleischer received the AB degree in physics from Harvard University in 1981 and the MS and PhD degrees in electrical engineering from Stanford University in 1987 and 1989, respectively. He is a research staff member in the VLSI Design Department, IBM Thomas J. Watson Research Center. He joined the IBM Research Division in 1989. Since 1992, he has been working on microprocessor circuit design. He was a member of the floating-point team for the z900 microprocessor, and the lead circuit designer for the floating-point units in the z990, P6, and BlueGene/Q microprocessors. He is the author or coauthor of a textbook chapter, multiple technical articles, and 15 patents. He is a member of the IEEE.

Yiorgos Makris received the diploma of computer engineering and informatics from the University of Patras, Greece, in 1995, and the MS and PhD degrees in computer science and engineering from the University of California, San Diego, in 1997 and 2001, respectively. He then spent more than 10 years as a faculty of electrical engineering and of computer science at Yale University and he is currently an associate professor of electrical engineering at the University of Texas at Dallas, where he leads the Trusted and Reliable Architectures Research Group. His current research interests include soft-error mitigation in digital circuits, machine learning-based testing of analog/RF circuits, mitigation of hardware Trojans, as well as test and reliability of asynchronous circuits. He serves on the organizing and program committees of many conferences in the areas of test and reliability and is the program chair for the 2013 IEEE VLSI Test Symposium. He is a senior member of the IEEE.