The Floating-Point Unit of the Jaguar x86 Core

Jeff Rupley, John King, Eric Quinnell, Frank Galloway, Ken Patton, Peter-Michael Seidel, James Dinh, Hai Bui, Anasua Bhowmik

AMD Austin and Bangalore

***Abstract*—The AMD Jaguar x86 core uses a fully-synthesized, 128-bit native floating-point unit (FPU) built as a co-processor model. The Jaguar FPU supports several x86 ISA extensions, including x87, MMX, SSE1 through SSE4.2, AES, CLMUL, AVX, and F16C instruction sets. The front end of the unit decodes two complex operations per cycle and uses a dedicated renamer (RN), free list (FL), and retire queue (RQ) for in-order dispatch and retire. The FPU issues to the execution units with a dedicated out-of-order, dual-issue scheduler. Execution units source operands from a synthesized physical register file (PRF) and bypass network. The back end of the unit has two execution pipes: the first pipe contains a vector integer ALU, a vector integer MUL unit, and a floating-point adder (FPA); the second pipe contains a vector integer ALU, a store-convert unit, and a floating-point iterative multiplier (FPM). The implementation of the unit focused on low-power design and on vectorized single-precision (SP) performance optimizations. The verification of the unit required complex pseudo-random and formal verification techniques. The Jaguar FPU is built in a 28nm CMOS process.**

Keywords; AMD Jaguar; floating-point unit; x87; SSE; AVX; MMX; AES; CLMUL; F16C; industry implementation

I. INTRODUCTION

The AMD Jaguar x86 core [1][2][3] is a fully-synthesized, two-wide, out-of-order superscalar core implemented in a 28nm CMOS process, targeted at low-power, low-cost form factors. Jaguar is the next-generation architecture of the AMD Bobcat x86 core [4], supporting many x86 ISA extensions, including the floating-point specific sets x87, MMX, SSE1 through SSE4.2, AES, CLMUL, AVX and F16C.

This paper describes the microarchitecture of the Jaguar 128-bit native FPU in detail. The architecture section describes the Jaguar FPU co-processor model, the ISA register file formats required, new register rename optimizations, the optimized, fully-synthesized 4-read, 3-write PRF, and the execution units. The vector integer execution units include two vector ALUs and a vector integer multiplier, all on a shared bypass network. The floating-point execution units include a store-convert unit (STC) that drives results to the main-core data cache, a floating-point adder (FPA) that shares roots with the AMD K7 [5][6] FPA, and a floating-point iterative multiplier derived from the Bobcat FPM design [7] and K7 divide/square-root algorithms [6].

The verification section describes a high-level summary of the many verification models used by our team, as well as the industry tools for formal verification works. Finally, the results section describes the IPC uplifts of the Jaguar core FPU over the Bobcat core FPU on various binaries (such as SPECFP 2006). The Jaguar FPU is built in a 28nm CMOS process, capable of executing in a variety of voltage ranges and frequencies of up to and beyond 2GHz.

II. ARCHITECTURE

The Jaguar FPU is first and foremost a "co-processor" architectural model – meaning that the FPU contains a dedicated decode, rename, out-of-order scheduler, and in-order retire units. The FPU accepts up to two micro-instructions per cycle from the dispatch unit in the main processor and is able to retire up to two micro-instructions per cycle. Figure 1 shows a block diagram of the Jaguar FPU architecture.

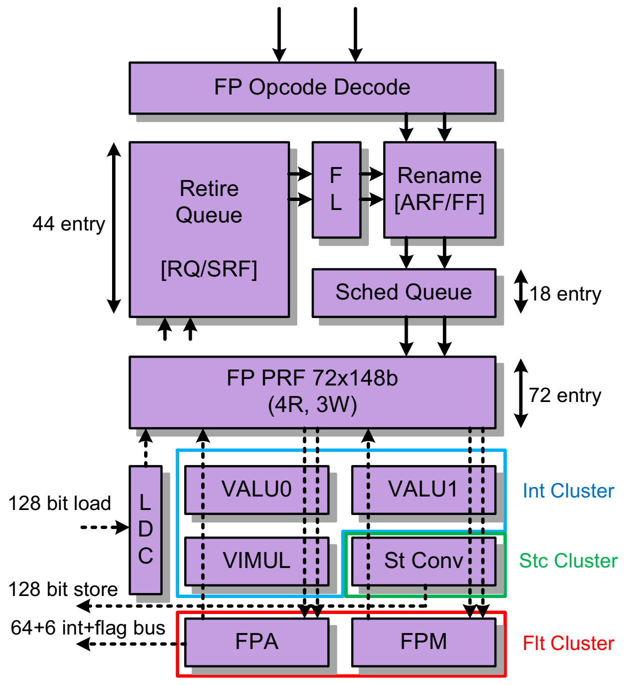

Figure 1. Jaguar FPU Block Diagram

The FPU front-end contains a dual micro-instruction decoder and renamer that drives an 18-entry scheduler queue (SQ) and a 44-entry retire queue (RQ)/status register file (SRF). Operand data is stored into a pointer-renamed floating-point physical register file (PRF) that has 72 physical entries of 148-bits each (bits in excess of 128-bits are for internal bookkeeping and classification). The PRF drives four 128-bit read ports to feed two execution pipes and has three 128-bit write ports: two write ports for results from the two execution pipes and one dedicated write port for incoming data from the main processor data cache.

The two execution pipes contain three major execution units each: Pipe0 contains a vector integer arithmetic logic unit (VALU0), a vector integer multiplier (VIMUL), and a floating-point adder (FPA); Pipe1 contains a vector integer arithmetic logic unit (VALU1), a floating-point store and convert unit (STC), and a floating-point multiplier (FPM). The Pipe1 STC drives a 128-bit data bus to the main processor data cache, while Pipe0 drives a 64-bit integer data and 6-bit flag bus (EFLAGS) to the main processor integer unit. The execution pipes have three separate bypass networks, organized into integer, store/convert, and float clusters.

A. *Decode, Schedule and Retire*

The front-end control and instruction flow of the FPU begins with the instruction dispatch interface to the FPU from the main processor. This dispatch interface uses a token system to track machine resources that each instruction will need, such as queue entries and registers, with optimizations for early freeing of tokens in cases where the resources were not needed. Once dispatched, instructions proceed to the dual micro-instruction decoders. These decoders translate micro-instructions into internal FP operations (ops). The FPU renamer renames two ops per cycle and writes into the retire queue and the out-of-order scheduler.

The scheduler is a single, unified, dual-pipe, out-of-order scheduler. It tracks dependencies and issues ops to the execution units as execution sources become ready, with appropriate control signals used to read operands from the PRF or the bypass network. The non-shifting scheduler uses an age matrix scheme for picking the oldest-ready ops for each pipe [8]. Some ops, such as floating-point adds and multiplies, can only issue in one pipe based on execution resources. Other ops, such as those executed on vector integer ALUs, can issue in either pipe. Ops are statically bound to a single pipe before being written into the scheduler queue using a counter-based pipe-balancing algorithm. The scheduler can issue up to two ops per cycle – one op to each execution pipe. The scheduler pre-reserves result bus cycles on each pipe using each op's pre-assigned latency to avoid result bus contention.

The retire unit can retire up to two FPU ops per cycle and coordinates architectural register file updates, status word updates, and exception processing with the retire unit in the main processor.

B. *PRF, Rename, and ZBits*

The Jaguar FPU supports x87/MMX, SSE, and AVX register sets via a unified rename unit and unified physical register file. The PRF size was a key design consideration; the goals of high performance, low power, and ISA support were in tension, so multiple techniques were used to keep the PRF size manageable while still allowing for high performance and full ISA support. Ultimately, the PRF is sized to primarily support 128-bit xmm operands and uses a selection of extra bits and class fields to map the remaining ISA formats. Figure 2 shows the many ISA register formats supported by the PRF.

x87/MMX registers are architecturally 80-bits wide, SSE registers are architecturally 128-bits wide, and AVX extends the SSE registers up to 256-bits wide. For all formats, excluding 256-bit AVX, the Jaguar PRF implements each format as a 128-bit renamed microarchitectural register. For AVX 256-bit ymm registers, the renamer allocates two microarchitectural registers – one for the lower 128-bits and one for the upper 128-bits.

Register renaming is done using an indirect, physical register scheme and a map table which maps microarchitectural registers to physical register numbers. Two tables are used: a speculative table used for renaming of incoming instructions, and a retire-time checkpoint table which holds the non-speculative architectural rename state.

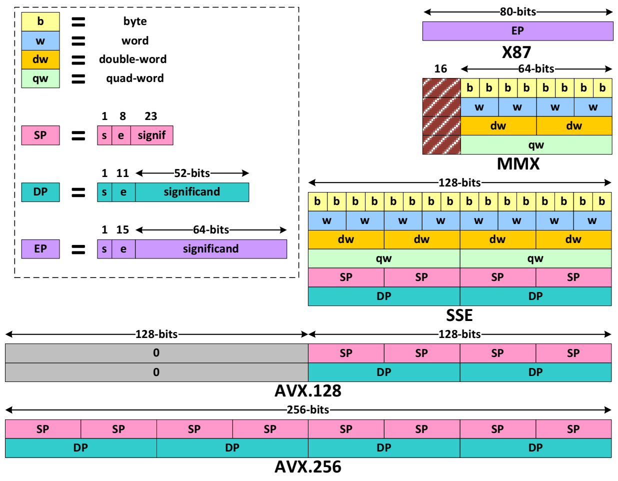

Figure 2 Supported PRF ISA formats

AVX 128-bit instructions define the upper 128-bits of a ymm 256-bit result to be zeroed out. Since the Jaguar FPU renaming is done at a 128-bit granularity, a direct implementation would require holding a dedicated microarchitectural register of all zeros. The Jaguar FPU optimization of these static-zero cases uses an additional renamer bit, one per microarchitectural renamed register called a "zero-bit" (ZBit), to track "all-zero" registers. This bit allows for the architectural state to be correctly represented without allocating an entire PRF entry full of zeros.

When the ZBit is set by an instruction, the renamer does not allocate a new physical register to map to the zeroed out architectural register. When the zeroing instruction retires, the physical register that had been previously backing that same architectural register is freed as usual. However, no new register will be mapped, and instead only the ZBit will be set in the architectural map table. This scheme allows for more physical registers to be available for other in-flight instructions, ultimately allowing the FPU to keep the number of PRF entries low. This also allows 128-bit AVX instructions to perform identically to their 128-bit SSE counterparts, using only a single op, a single physical register, and a single scheduler entry.

When younger ops pass through the renamer, they look up the ZBit for each of their sources and pass it through to the scheduler. If the ZBit is set, the source is marked "ready", and that op will not have a dependency on any dataflow into that register. When an issued op uses ZBits, the bit is sent to the bypass network to inject the all-zeros data for the specified source operand while suppressing PRF reads, saving power.

Software operations that are intended to zero-out a register (such as XORPS xmm3, xmm3) are detected in the FPU decoder and renamer and immediately apply the ZBit optimizations. The Jaguar FPU further improves the performance of these cases by turning the instruction into NOPS that do not execute or use a scheduler entry.

C. *ISA Register Reclaim*

The ZBit optimization scheme in the FPU was so powerful at improving power and performance metrics that the technique has been applied to cases outside simple zero-register instructions. Two new major techniques employ the ZBit optimization to reclaim architectural registers when not in use: microcode temporary register reclamation and x87 register caching.

In microcode cases, the FPU implements eight temporary renamed registers (ftmp0-ftmp7) for use by complex instructions to store intermediate results. When microcode is not being executed, these registers are not defined to hold usable data, so holding entries for the ftmps in the PRF is inefficient. In Jaguar's FPU, when a microcode sequence ends, these unused ftmp registers may be marked "dead" and their physical registers reclaimed by setting the ZBits in the renamer. While the data values of the ftmps may not be all-zeros as specified by ZBits, the data value in the dead registers is inconsequential, so the optimization may be used in good faith to add entries back to the PRF free list.

The second FPU ISA optimization employs the ZBit renaming to x87 register caching. Since the Jaguar FPU is optimized for SSE and AVX single-precision performance, x87 performance is explicitly de-emphasized. With this target market, the design assumes that the eight x87 registers themselves will often be unused.

Following this assumption, the FPU employs an optimization during state-restore instructions such as FXRSTOR or XRSTOR, which are instructions that load values from a memory image into all of the FPU registers – including the x87 registers. Rather than restoring the x87 values to the PRF during the instruction sequence, microcode instead copies the x87 values from the memory image into internal scratch storage in the main processor data cache unit and sets the ZBits in the FPU. The sequence then sets a status bit in an internal control register indicating that the x87 registers are unmapped, or "cached". If an x87 instruction is detected while the x87 registers are cached, the main core takes a fault and vectors to a sequence which restores the x87 values from the scratch storage space back into the FPU PRF.

D. *Vector ALUs*

The Jaguar FPU implements two symmetric vector ALU units in the back-end of the co-processor. The dual implementation is designed to support the execution of two non-dependent 128-bit vector integer instructions per cycle. The ALUs handle all non-multiply packed integer arithmetic for MMX, SSE, and AVX instruction classes. Both ALUs share a common bypass network with the vector integer multiplier that allows for zero-cycle result forwarding.

Each vector ALU has a 128-bit dataflow consisting of two replicated 64-bit data paths (Hi and Lo). The 64-bit datapaths themselves are symmetrical, with the exception of an additional PCMP*STR*/MPSADBW path named the "SADSTR" sub-unit that is only available in the Pipe1 ALU. Figure 3 shows the pipe instantiations of all 4 vector ALUs.

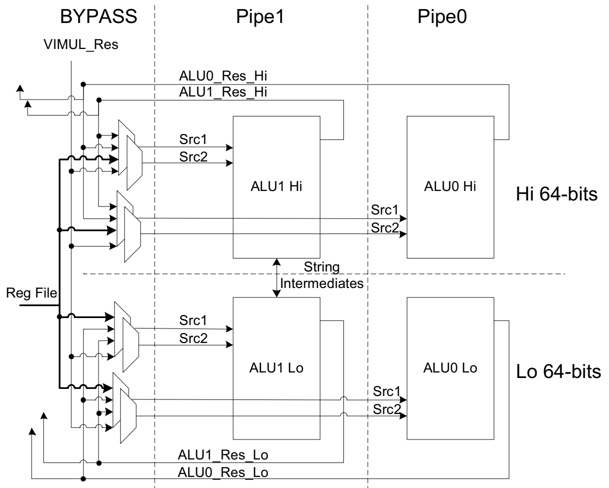

Figure 3. Vector ALU pipe instantiation

Each 64-bit datapath has 5 common functions: packed-add, packed-shift, packed-logicals, data packing, and miscellaneous operations to support the SSE4.2 string instructions. The majority of complex packed integer operations occur in the packed-add or packed-shift datapaths. Each path speculatively computes different operand sizes (byte/word/dword/qword) and selects the result after the control decides which operation is correct. The vector integer packed adder 64-bit datapath uses a singular 80-bit monolithic adder that employs zero-injections as carry-kills to handle the several collections of parallel byte/word/dword addition operations required.

As already mentioned, Jaguar added a SADSTR accelerator to the Pipe1 ALU unit. SSE4.1 and SSE4.2 introduced a new sum-of-absolute-differences instruction (MPSADBW) and four new string compare instructions (PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM), each of which requires an obscene number of packed parallel word/byte additions. While a microcoded implementation of the instructions could have iteratively utilized the 80-bit monolithic adder already present in the 64-bit ALU, performance requirements of new code and media accelerators motivated a higher performance solution. The SADSTR accelerator unit is effectively a large array (64 x 9-bit per 64-bit ALU) of byte/word adders dedicated to the shared task of massively parallel byte/word additions. However, while a single accelerator was a justified tradeoff for performance and power, replicating the array to both pipes was not as attractive an investment of hardware.

E. Vector Integer Multiply

The FPU vector integer multiply (VIMUL) unit primarily handles packed integer multiplies of various bit-widths and sign modes. Additionally, for reasons that include shared control logic and similar latencies (albeit in the presence of different dedicated execution hardware) the vector integer multiply unit supports Advanced Encryption Standard (AES) and carry-less multiplication (i.e. PCLMULDQDQ, or CLMUL) instructions. The VIMUL is located in Pipe0.

The VIMUL, like the vector ALU units, is split into two symmetrical 64-bit datapaths. The primary structure of the unit is an integer multiplier composed of four radix-4 partial product trees (16 x 16 signed mult) that conditionally combine results based on the required operation and destination format. The arrays may be further combined to support up to a 32-bit x 32-bit multiply, the largest single operation being PMULDQ which is the signed 32-bit multiply introduced in SSE4.1. Figure 4 shows the primary execution stage of a 16x16 multiply.

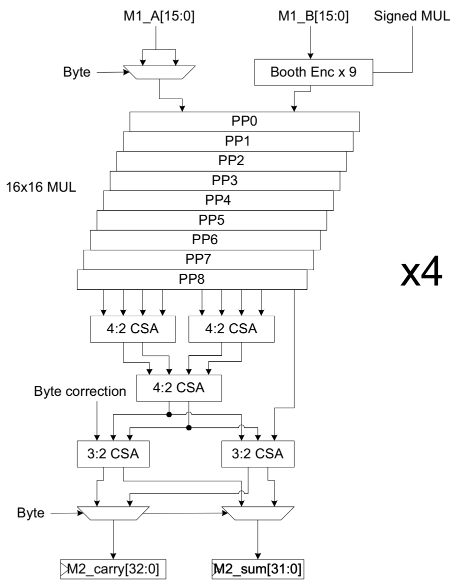

Figure 4. Vector integer multiply 16x16-bit radix-4 tree

The Jaguar Advanced Encryption Standard implementation follows the FIPS-197 government specification for 128-bit encryption, where Nr and Nk equal 4. The encryption itself is broken into individual programmable iterations for encrypt/decrypt as specified in the AES ISA instruction set instead of a native full 10 round loop. The unit architecture uses the un-glorious GF(2^8) cipher look-up tables for ease of implementation and verification.

The carry-less multiplication datapath is additional hardware in the VIMUL that is not symmetric across 64-bit datapaths. CLMUL uses a dedicated accelerator of Galois partial-product trees for a low-latency calculation. The 64-entry single-pass pseudo partial-product tree is possible primarily due to the replication of each 64-bit operand to each 64-bit datapath lane. Each execution lane then uniquely creates each partial product as confined by a local 64-bit XOR tree result. Figure 5 shows the mapping of the non-symmetric CLMUL implementation.

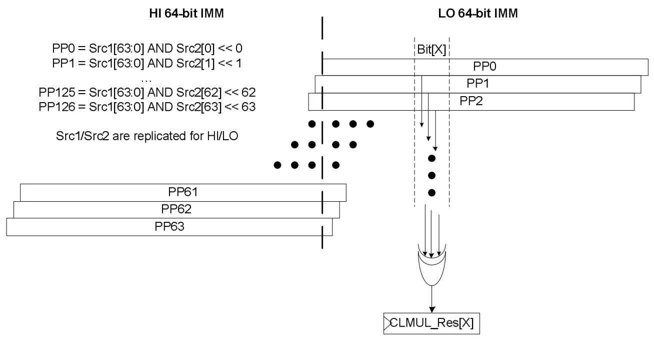
HI 64-bit IMM
LO 64-bit IMM
PP0 = Src1[63:0] AND Src2[0] << 0
PP1 = Src1[63:0] AND Src2[1] << 1
...
PP125 = Src1[63:0] AND Src2[62] << 62
PP126 = Src1[63:0] AND Src2[63] << 63
Src1/Src2 are replicated for HI/LO
Bit[X]
PP0
PP1
PP2
PP61
PP62
PP63
CLMUL_Res[X]

Figure 5 CLMUL Galois partial product tree

F. Store/Convert

The Jaguar FPU store/convert unit (STC) performs two major functions: data type conversions (e.g. CVTPS2PI instructions) and storing data from the PRF to the main processor data cache. The STC unit is only available on Pipe1 of the FPU. The unit is split into two non-symmetrical datapaths, with a "Hi" 64-bit datapath for SSE/AVX support, and a "Lo" 80-bit datapath to support both x87 as well as SSE/AVX. The STC unit is also the sole execution unit for the F16C half-precision conversion instructions.

The STC has the privilege of handling most of the corner-case and miscellaneous operations required in the x86 floating-point ISA. Specifically, the STC holds the x87 constant ROM and overloads the conversion datapaths for x87 Rounding Control (RC) and Precision Control (PC) of the constants. Additionally, the STC handles all microfault sequences regarding internal formatting or denormal results. The Jaguar FPU does not handle floating-point denormal numbers natively (with the exception of those described later in the Floating-Point Adder), so any denormal handling or rounding is done in a fault or trap by the STC.

G. Floating-Point Adder

The Jaguar FPA unit handles a large collection of floating-point operations ranging from x87 to AVX ISA additions. The unit is capable of variable latency, supporting 1, 2, and 3-cycle operations. The FPA handles transcendental 68-bit significand internal-precision (IP), x87 EP (which may be rounded to 24, 53, or 64 bits), scalar/packed DP, and scalar/packed SP. The main operations include floating-point addition/subtraction, floating-point compare/max/min, logicals, moves in floating-point format, and all FPU-to-main core architectural register operations. The FPU-to-main core operations include integer PRF data writes, EFLAGS results, and condition code flags. The unit is implemented in Pipe0 only, with a 64-bit "Hi" datapath which supports SSE/AVX, and 80-bit "Lo" datapath which supports x87/SSE/AVX. The FPA is part of the "FLT" cluster which may forward zero-cycle results to the FPA or the FPM.

The Jaguar FPA unit takes its architectural roots all the way back to AMD's K7 FPU [5][6]. The Jaguar FPA main architecture still uses a classic dual-path implementation which employs a FAR and CLOSE path to speculatively handle the two cases of floating-point addition while the exponent difference computes the correct operating range. The main arithmetic unit also uses the K7 3 x 72-bit speculative adders: one adder for holding the unrounded results (needed for x87 denormal handling), one adder for calculating the rounded result, and one adder for handling a rounded result that overflows on an add or requires a 1-bit left shift on a subtract. The block diagram of the FPA datapath is shown in Figure 9.

The floating-point rounding injection mechanism also shares the same high-level architecture from K7. When assuming a round-up in round-to-nearest-even (RN) mode, a singular '1' is inserted at the round-bit position. This forces a round for any fraction greater than 0.5 the LSB. The nearest-even correction for odd rounds may be done trivially after the actual addition. For overflow or one-left-shift cases, the table may be left or right shifted by one bit-position. For round to positive/negative infinity (RP/RM), the rounding constants consist of the positive/negative sign of the final result extended to all bits from the round-bit down. Should any data exist at all in a bit-position below the LSB, this constant will carry the data to the LSB and force a round in the correct direction. Figure 6, 7, and 8 show the RNE, RM, and RP Jaguar FPA rounding constants.

While the high-level architecture and constants hold a common ancestry to the K7 architecture, the Jaguar FPA has significant differences at a micro-architectural level. The new FPA supports a plethora of new ISA instructions, including all the SSE and AVX additions, a new pipeline for higher frequency in a vastly different process, and as is common with the Jaguar FPU goals, a new emphasis on performance acceleration of vector SP floating-point operations. As a small additional feature, the Jaguar FPA also handles all SSE and AVX denormals natively and requires no microtraps or fixups in post-processing (x87 denormals still require microtraps).

Floating-point compare/min/max ops are handled by a shorter datapath than the main dual-path compute and use dedicated comparators rather than the 3 main adders. In Jaguar, the short compares were updated to new ISA requirements and piped as 2-cycle operations.

All main-core integer result/flag uops were re-piped in Jaguar to have variable latencies. All x87 and MMX ISA integer writes are three cycles, while all SSE and AVX writes are two cycles. As a note, the latencies of main-core integer communications are not dictated by execution time, but rather by the control mechanisms used to "jam" the FPU result data onto the result or flag bus on the main core. Table I shows the FPA floating-point add/sub, compare/min/max, and main-core integer bus latencies.

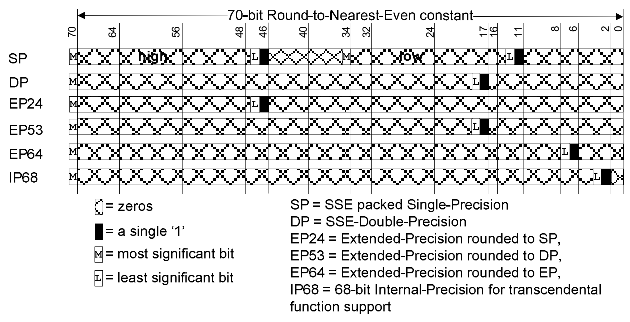
70-bit Round-to-Nearest-Even constant
70
64
56
48
46
40
34
32
24
17
16
11
8
6
2
0
SP
DP
EP24
EP53
EP64
IP68
high
low
= zeros
= a single '1'
= most significant bit
= least significant bit
SP = SSE packed Single-Precision
DP = SSE-Double-Precision
EP24 = Extended-Precision rounded to SP,
EP53 = Extended-Precision rounded to DP,
EP64 = Extended-Precision rounded to EP,
IP68 = 68-bit Internal-Precision for transcendental function support

Figure 6. FPA Round-to-Nearest-Even (RNE) constants

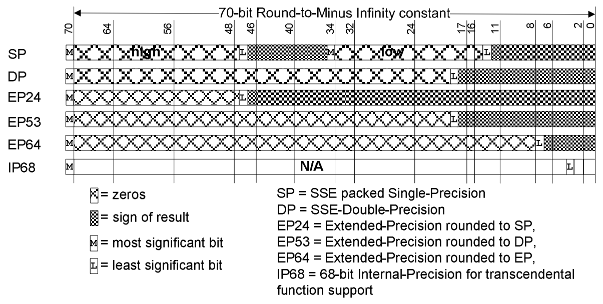
70-bit Round-to-Minus Infinity constant
70
64
56
48
46
40
34
32
24
17
16
11
8
6
2
0
SP
DP
EP24
EP53
EP64
IP68
high
low
N/A
= zeros
= sign of result
= most significant bit
= least significant bit
SP = SSE packed Single-Precision
DP = SSE-Double-Precision
EP24 = Extended-Precision rounded to SP,
EP53 = Extended-Precision rounded to DP,
EP64 = Extended-Precision rounded to EP,
IP68 = 68-bit Internal-Precision for transcendental function support

Figure 7. FPA Round-to-Minus Infinity (RM) constants

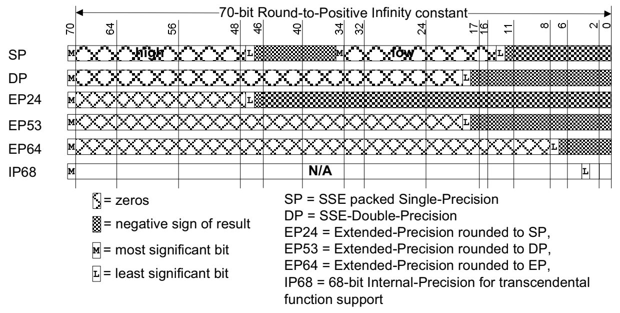
70-bit Round-to-Positive Infinity constant
70
64
56
48
46
40
34
32
24
17
16
11
8
6
2
0
SP
DP
EP24
EP53
EP64
IP68
high
low
N/A
= zeros
= negative sign of result
= most significant bit
= least significant bit
SP = SSE packed Single-Precision
DP = SSE-Double-Precision
EP24 = Extended-Precision rounded to SP,
EP53 = Extended-Precision rounded to DP,
EP64 = Extended-Precision rounded to EP,
IP68 = 68-bit Internal-Precision for transcendental function support

Figure 8. FPA Round-to-Positive Infinity constants

TABLE I FPA add/sub, cmp/min/max, and integer bus latencies

Operation	Precision	Latency	Throughput
Add/Sub	SP/DP/EP/IP	3	1
Compare/Min/Max	SP/DP	2	1
Integer data/flag bus	SSE/AVX SP/DP	2	1
	x87/MMX integer/EP/IP	3	1

The FPA's one or two cycle operations, referred to as "short" operations, are allowed to tunnel, or execute underneath, 3-cycle float operations as long as there are no result bus collisions. SSE/AVX short ops generally take 1 cycle, while x87 short ops are de-emphasized and take 2 cycles. Short operations are not necessarily adder specific (and include logicals, movs, compares, etc.) and generally are replicated in the FPM to allow for parallel pipe short op execution.

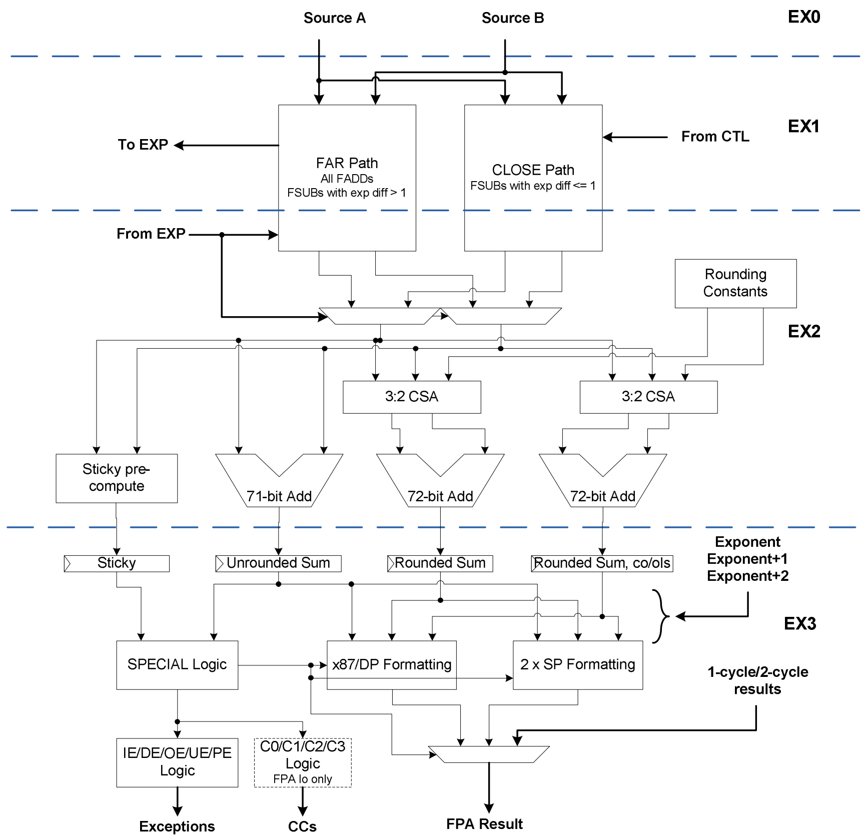

Figure 9 Jaguar FPA datapath using 3 x 72-bit adders for unrounded, rounded, and overflow/one-left-shift cases

The SPECIAL logic unit of the Jaguar FPA handles all special data results including zero, infinity, SNaN, QNaN (indefinites), and denormals. The unit also handles all exceptions (IE, DE, PE, UE, OE, CC flags) according to IEEE-754 standard's [9] and the APM rules [10]. With the addition the dot-product operation of SSE4.1/AVX, the Jaguar FPA utilizes a new "dot-product exception" (DPX) to specifically handle the bizarre internally-serial exception cases of the instruction via a new full-core dot-product microfault (shared with the FPM) that alternates execution modes between a fast mode and a slow, denormal-aware mode. The DPX logic also handles dot-product post-execution zero masking, undefined NaN propagations, and cross-register AVX256 MXCSR exception detections.

H. Floating-Point Multiplier

The Jaguar floating-point multiplier handles all floating-point operations that require multiplication, as well as division, reciprocal, and square root operations. The unit is capable of supporting a large range of variable latencies, including the previously described 1 and 2-cycle "short" ops in common with the FPA. The FPM handles IP, EP, scalar/packed DP, and scalar/packed SP formats. The unit does not handle denormals natively in any ISA family, requiring a core microfault to handle the numerical cases. The unit is implemented in Pipe1 only, with a 64-bit "Hi" datapath which supports SSE/AVX, and an 80-bit "Lo" datapath which supports x87/SSE/AVX. The FPM is part of the FLT cluster which may forward zero-cycle results to the FPA or the FPM. Figure 10 shows the FPM architectural block diagram.

The Jaguar FPM unit is designed to accelerate single precision operands using very small silicon area (at the expense of double and extended precision performance) by using a 76-bit x 27-bit booth-encoded radix-4 iterative multiplier, as first described in detail by Tan, *et.al.* [7]. The FPM implements an evolution of the K7 division/square-root algorithms [6] based on Goldschmidt's method [11] as applied to an iterative multiplier.

The FPM supports a 2-cycle 128-bit packed SP multiply latency, fully pipelined with dedicated SP execution paths. Multiply DP takes a 4-cycle latency, with iteration every other cycle and a one-half throughput (meaning the unit blocks new multiply ops during iterations). Multiply EP takes a 5-cycle latency, with two iterations and a throughput of one-third. Table II shows the floating-point multiply, divide and square-root latencies of the Jaguar FPM.

Divides, remainders, and square root operations are long latency ops that use a shared path with the DP and EP multiply paths, albeit with a dedicated feedback pipe and special DIVSQRT control state machine. Figure 11 highlights these special DIVSQRT architectural blocks in the FPM. The DIVSQRT control machine will signal to the scheduler 6 cycles before the instruction is finished and block any other floating-point multiply-based ops during iteration. Short operations are allowed to tunnel underneath the multiply/long ops as long as the result bus doesn't have collisions.

As already mentioned, the DIVSQRT state machine implements the K7 division/square-root algorithms [6] as modified for iteration [7]. This special pipelining and iterative feedback allows the Jaguar FPM to uniquely accelerate packed SP divide and square root operations by introducing a "multiplicative leapfrog" interleaving method to overlap computation and hide iterative dependencies.

As an illustrative example, consider in Figure 12 the algorithmic program definition and terminology presented in [6] for SP packed square root calculation. The algorithm itself requires seven dependent steps to calculate a rounded result. The Jaguar DIVSQRT machine additionally constrains this algorithm to not allow early-exit calculations – fixing all square root latencies regardless of operand values. This constraint allows the low SP (L) and high SP (H) calculations at each step of the algorithm to be mapped explicitly to dedicated, non-overlapping sections of the FPM. Table III shows the pipeline assignment and implementation of the packed SP square root calculation in detail.

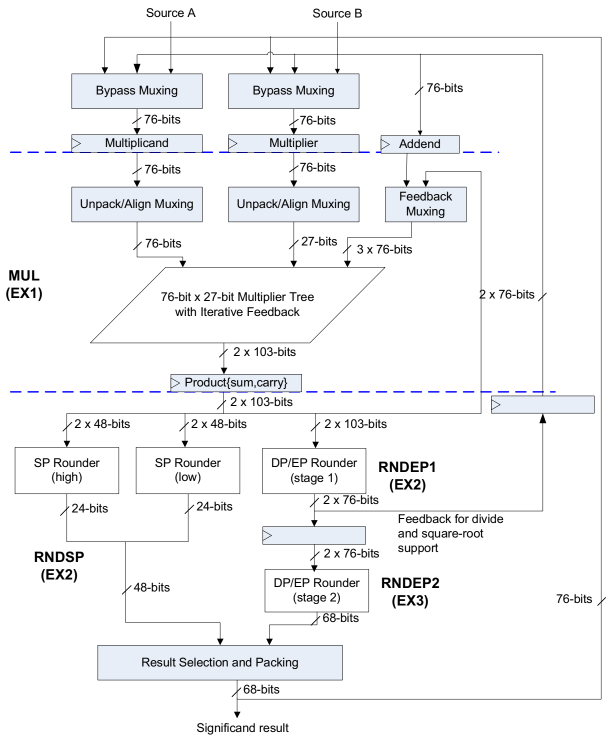

Figure 10. Jaguar iterative FPM block diagram

TABLE II FPM multiply, divide and square-root latencies

Operation	Precision	Latency	Throughput
Multiply	Scalar/packed SP	2	1
	Scalar/packed DP	4	1/2
	EP/IP	5	1/3
Divide	Scalar SP	14	n/a
	Packed SP	19	
	Scalar/packed DP	19	
	EP/IP	22	
Square-Root	Scalar SP	16	n/a
	Packed SP	21	
	Scalar/packed DP	27	
	EP/IP	35	

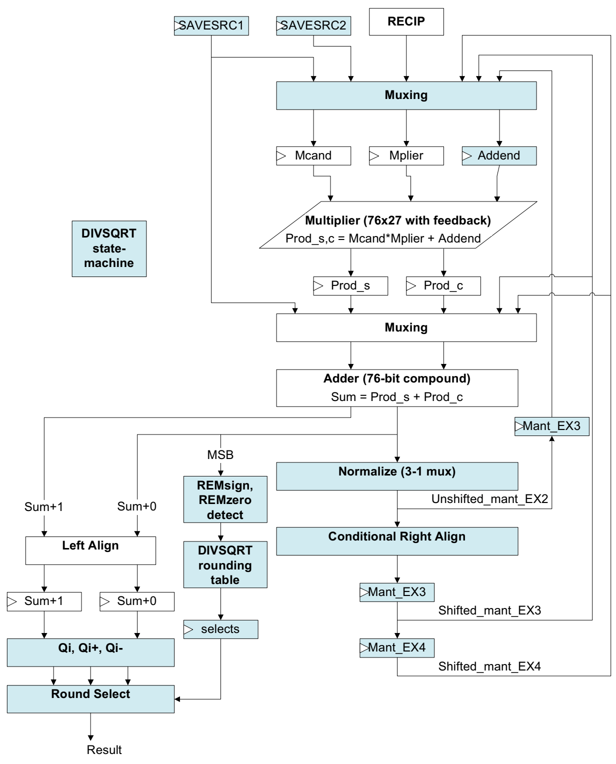

Figure 11 Jaguar iterative DIVSQRT block diagram

***Program** SSE/AVX SQRTPS*
Input = (a,pc,rc) Output = (s_f)
x_0= RECIPSQRT_ESTIMATE(a)
t_0 = ITERMUL_27x76(x_0, x_0)
d_f = ITERMUL_27x76(x_0, a)
n_0 = ITERMUL_27x76(t_0, a)
r_f = COMP3(N_0)
s_i = LASTMULADD_27x76(r_f<<13, d_f>>13, d_f, 25)
rem = BACKMUL_25x25(s_i, s_i, a)
s_f = ROUNDSEL(s_i, rem, pc, rc)

Figure 12 Jaguar iterative DIVSQRT block diagram [6][7]

TABLE III Jaguar SSE/AVX SQRTPS “multiplicative leapfrog” interleaving pipeline diagram

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
Reciprocal Sqrt (EX0)	x_0.L	x_0.H																			
		x_0.L	x_0.H																		
Multiplier (EX1)			t_0.L	d_f.L	t_0.H	n_0.L	d_f.H	n_0.H	s_i.L		s_i.H		rem.L					rem.H			
Adder (EX2)				t_0.L	d_f.L	t_0.H	n_0.L	d_f.H	n_0.H	s_i.L	s_i.L	s_i.H	Qi+/-.L	rem.L	rem.L	s_i.H		Qi+/-.H	rem.H	rem.H	
Unshifted Mantissa (EX3)					t_0.L		t_0.H					s_i.L					s_i.H				
Shifted Mantissa (EX3)						d_f.L		n_0.L	d_f.H	n_0.H		s_i.L	s_i.H				s_i.H				
Shifted Mantissa (EX4)							d_f.L	d_f.L		d_f.H				s_i.H	s_i.H						
Round Select																s_f.L					s_f.H

III. Verification

The Jaguar FPU was functionally verified using three primary mechanisms: pseudo-random verification at an FP testbench level, pseudo-random verification at a core level, and formal verification at an execution-unit level.

The Jaguar FPU testbench was a System Verilog model that isolated the FPU RTL in an environment surrounded by software interfaces called “Bus Functional Models” (BFM). The BFMs included emulation models for all pieces of necessary main-core interface that drove the I/O busses of the FPU itself, including the microcode engine, load-store unit, front end modules, and bus unit, among others. The testbench model was designed to specifically increase simulation cycles-per-second in tests on a smaller, more focused functional model.

The testbench used an exhaustive instruction database as the source of a constrained random stimulus generator which fed the FPU's issue pipes. A software-based golden model was used to check and compare the instruction results, while System Verilog checkers and assertions monitored the FPU's internal activity. Operand values were chosen via constrained random generators for all data formats (SP, DP, EP, packed integers of various sizes) to cover the wide variety of numeric cases. Random System Verilog sequences were generated to model all FP microcode and branches for complex FP instructions. The testbench was central to driving the 194,083 coverage points required for FPU verification closure.

For full core testing, a new architectural level tool called “Gemini” used a relative weighting algorithm to generate interesting FP instruction sequences. Additionally, the tool used a similar algorithm to generate numerics, such as denormals, infinities, NaNs, and other corner cases. JG FP RTL architects used this tool to create targeted FP self-checking tests and more FP-centric full-core qualification regressions. Outside of Gemini, additional legacy full-chip random exercisers and directed testing rounded out the functional verification effort.

The traditional verification of the FPU was complemented by the formal verification of selected functions of the implementation. The formal work provided improved coverage for the functionalities of the FP multiplications, the reciprocal approximation module, parts of the STC, the vector integer multiplier, and the AES/CLMUL engines. The formal verification work reduced the amount of coverage analysis needed by the pseudo-random engines, reducing verification resource requirements. The FPU’s formal verification was based on theorem proving with the ACL2 theorem prover [12] and on property checking using Jasper’s commercial model checking tool [13]. The verification of the FP multiplier built on the formal verification effort of the Bobcat FP multiplier [14]. The extension of this effort needed to take into consideration the enhanced clock gating schemes, the possible scheduling of a larger variety of short ops concurrent with FP multiplications and the extension to the 64-bit “Hi” datapath, where adjustments were needed for a different register file organization and interface, and the removal of logic for the functionality that was just needed in the 80-bit “Lo” datapath. For the vector integer multiplier we combined the theorem proving and the model checking approaches to reduce the combined verification effort compared to using just either of the two approaches and to make the verification more resilient to late changes in the RTL.

IV. Results

The Jaguar FPU was implemented in a 28nm bulk CMOS node with a fully-synthesized (including the PRF) auto-place and route standard-cell methodology. The FPU was constrained to a rectangular region of the core floorplan, consuming 0.39 mm^2 and using 37k flops. Assuming a 2GHz frequency, the FPU is capable of a theoretical peak of 16 SP GFLOPS, or 6 DP GFLOPS. The FPU, like the core, can run at a variety of voltages and frequencies, depending on the required power and performance constraints of the system. Table IV shows the basic physical and performance metrics of the FPU.

Table IV Jaguar FPU implementation details

Node	28nm bulk CMOS
Area	0.39 mm^2 (479µm x 834µm)
# of Flops	37345
SP GFLOPS @2GHz	16 GFLOPS
DP GFLOPs @2GHz	6 GFLOPS

The performance analysis of Jaguar FPU used several benchmarks including the SPECFP 2006 benchmark suite with its reference input set and some media applications obtained from selected software providers. The SPECFP 2006 suite was compiled using the Open64 compiler, while the media applications were compiled from different commercially available compilers by the software providers. Figure 13 shows the instruction mix of integer, MMX, x87, 128-bit SSE instructions, and "other" (non-vector) SSE instructions in the binaries chosen for discussion.

The performance studies were conducted using an in-house trace-driven cycle-accurate simulator for the Jaguar microarchitecture. Each benchmark is simulated for about 1 billion instructions, taken from different parts of the program being tested, ensuring that the traces accurately capture the program behavior.

Figure 14 shows the instructions-per-cycle (IPC) improvement of the Jaguar 128-bit FPU feature as compared to the 64-bit Bobcat FPU under the same performance methodology. The native 128-bit execution capabilities of Jaguar show significant IPC improvements in several benchmarks, specifically those with large number of 128-bit SSE instructions, such as 436_cactus, mainconcept, winmedencode, 434_zeusmp, and 454_calculix. On the other hand, the IPC of iTunes (x87 compile), MovieMaker51, 447_dealii and 450_soplex do not improve much with the 128-bit FPU as these applications have a higher dependence on other ISA extensions. Benchmarks like povray and lbm do not get much IPC improvement as their performance is limited by other microarchitectural bottlenecks. Figure 14 also shows the IPC improvement due to the register reclaim ZBit features described in Section II, which universally have some level of performance uplift over Bobcat style static register assignment.

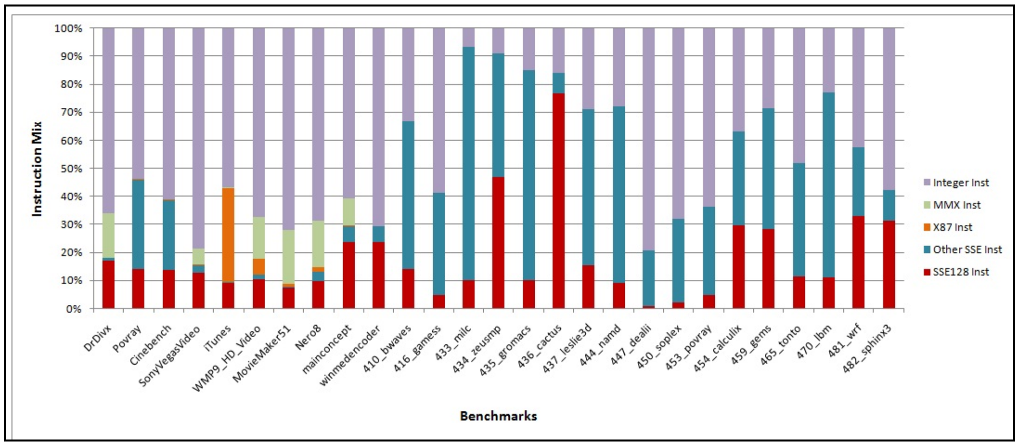

Figure 13. ISA instruction mix for various SPEC06 and media application binaries

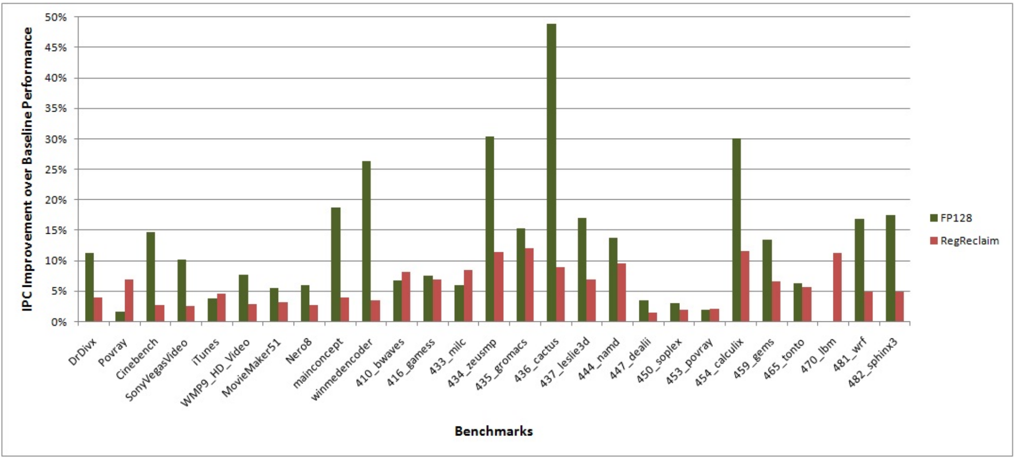

Figure 14. IPC benchmark improvements of FP128 and register reclaim features (Jaguar FPU vs the Bobcat FPU)

V. Conclusion

The new Jaguar x86 core uses a two-pipe, out-of-order co-processor model FPU for floating-point and media instruction execution. The architecture has 128-bit native execution mode, a new ZBit register reclaim renamer, a unified dual-pipe age-matrix scheduler, and an optimized, fully-synthesized 4-read/3-write 72-entry PRF. The unit contains two vector ALUs, a vector integer multiplier, dedicated AES and CLMUL cryptography hardware, a string and sum-of-absolute differences accelerator, a 128-bit store-convert unit, a 128-bit floating-point adder that has been improved from the K7 architecture, and a 128-bit floating-point iterative multiplier that has been enhanced from the Bobcat.

The Jaguar FPU was verified using a variety of testbenches and verification models, including formal verification via ACL2 theorem proving and Jaspergold model checking. The performance modeling used AMD in-house tools on benchmark binaries such as SPECFP 2006. The FPU is built in a 28nm bulk CMOS process, using 0.39 mm^2 of die area, and is capable of executing at a range of voltage and frequency points – up to and exceeding 2GHz.

Acknowledgement

The authors would like to acknowledge the contributions from the Bobcat FPU architects Carl Lemonds, Dimitri Tan, and Betty McDaniel; Jay Fleischman and Kelvin Golveas for their scintillating floating-point consultation; Todd Swanson and Xiang Wu for their verification efforts early in the project; and finally all of the Jaguar core design team for continuing to achieve small miracles.

References

[1] Rupley, J., "AMD's 'Jaguar': A next generation low power x86 core," *Hot Chips 24,* Aug, 2012.

[2] AMD, BIOS and Kernel Developer Guide (BKDG) for AMD Family 16h Models 00h-0Fh Processors, 2013, http://www.amd.com

[3] Singh, T.; Bell, J.; Southard, S.; "Jaguar: A Next-Generation Low-Power x86-64 Core," *ISSCC 2013*, pp. 12-13.

[4] Burgess, B.; Cohen, B.; Denman, M.; Dundas, J.; Kaplan, D.; Rupley, J.; "Bobcat: AMD's Low-Power x86 Processor," *IEEE Micro*, 2011, pp. 16-25.

[5] A. Scherer, M. Golden, N. Juffa, S. Meier, S. Oberman, H. Partovi, F. Weber, "An out-of-order three way superscalar multimedia floating-point unit," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.,* 1999.

[6] S.F. Oberman, "Floating-Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 106-115, 1999.

[7] Tan, D.; Lemonds, C.E.; Schulte, M.J.; "Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support," in *IEEE Transactions on Computers,* Vol. 58, Issue 2, pp. 175-187, 2009

[8] Sassone, P., Rupley, J., Brekelbaum, E., Loh, G., Black, B., "Matrix Scheduler Reloaded," in *Proceedings of ISCA-34*, Jun. 2007

[9] ANSI and IEEE, *IEEE-754 Standard for Binary Floating-Point Arithmetic*, 1985.

[10] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual,* Volumes 1-5, rev 3.19, March 2012.

[11] R.E. Goldschmidt, "Applications of Division by Convergence," Master's Thesis, Massachusetts Inst. of Technology, 1964.

[12] ACL2 Web site. http://www.cs.utexas.edu/users/moore/acl2/

[13] Jasper Web site. *www.jasper-da.com/.*

[14] Seidel, P., Formal Verification of an Iterative Low-Power x86 Floating-Point Multiplier with Redundant Feedback. In Proceedings ACL2 2011, published in EPTCS 70, 2011, pp. 70-83.