ELSEVIER

Theoretical Computer Science 279 (2002) 3–27

Theoretical Computer Science

www.elsevier.com/locate/tcs

Computer arithmetic and hardware: "off the shelf" microprocessors versus "custom hardware"

Daniel Etiemble

Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road Toronto, Ontario, Canada M5S 3G4

Abstract

This paper discusses the relationship between computer arithmetic and hardware implementation. First, we examine the impact of computer arithmetic on the overall performance of today's microprocessors. By comparing their evolution over the last 10 years, we show that the performance of arithmetic operators is far less critical than the performance of the memory hierarchy or the branch predictors. We then discuss the potential for improvement in arithmetic performance, both for pipelined and non-pipelined operations. We then examine the possible impact of new technologies, such as MMX technology or asynchronous control of microprocessors, on computer arithmetic. Finally, we show that programmable logic devices now permit a cost-effective implementation of specific arithmetic number representations, such as serial arithmetic or logarithmic representations.

Keywords: Arithmetic performance; Asynchronous operators; Computer arithmetic; Computer performance; Latency; Microprocessors; Multimedia instructions; Programmable logic; System performance; Throughput

1. Introduction

Performance of computers has climbed up steadily for more than 50 years and the progression rate has increased with the developments of microprocessors in the last 25 years. The big gap between low-end computers and high-end supercomputers is narrowing, because both use the same basic components. Most notably, the first supercomputer to break the Teraflop "wall" on a typical "LINPACK benchmark" was built with about 8000 Pentium Pro microprocessors. This Intel microprocessor is very close to the Pentium II now used in most of the desktop or workstation PCs. Computer users rarely hear about computer arithmetic, although they hear a lot about caches, disks,

E-mail address: de@eecg.toronto.edu (D. Etiemble).

0304-3975/02/$ - see front matter
PII: S0304-3975(00)00424-2

2D or 3D graphic cards, etc. The only opportunity to read about arithmetic in journal papers is when arithmetic bugs are discovered and revealed. The most famous was the Pentium bug on division operations. Intel announced some other less important arithmetic bugs for the Pentium II. In some sense, arithmetic looks marginal in the overall performance of standard microprocessors. In the first part of this paper, we will examine the different aspects of performance in "main stream" computers (PCs and workstations) and explain why the arithmetic is not the driving factor for improving performance, by discussing the impact of latencies of arithmetic operations on the instruction execution rate. Then, we will show that performance of arithmetic operations is improving incrementally: if there is still room for continuous progress in arithmetic performance, there are very few possibilities for real breakthroughs. Next, we will discuss the possible impact of new technologies: one is the multimedia extension of the instruction sets that is now used in most of the modern microprocessors, and the other corresponds to the "asynchronous approach" for controlling microprocessors. In the last section, we show that "special purpose" hardware can now be used for implementing specific arithmetic number representations with a good performance/cost ratio, using the efficient programmable logic components that are now available.

2. Performance of "off the shelf" microprocessors

2.1. Overall performance

Overall performance of standard microprocessors has been climbing steadily. Performance evaluation of computers is not straightforward and using benchmarks is very debatable, as discussed in [4]. However, benchmarks give some insight into the evolution of performance over time. One difficulty with benchmarks is that they become as obsolete as the computers, and they have to be replaced by new benchmarks. One example is the famous SPEC suite. The first instance of this suite is SPEC89, that was replaced by a new suite called SPEC92 in 1992. The performance of computers that was measured before had to be reevaluated according to the new SPEC92 scores. The SPEC92 score indicates how many times a machine is faster than the reference machine, which was a VAX-11 780. In 1995, a new version of the suite called SPEC95 was defined, with extended versions of the previous benchmarks and a new reference machine that totally changed the SPEC scores.

Fig. 1 shows the relative integer performance (equivalent to the SPECint92 scores) for the best RISC microprocessors and the best Intel microprocessors from 1986 to 1995. The scores for the last 3 years cannot be easily derived because there is no simple equivalence between SPECint92 and SPECint95 scores. The figure shows that the integer performance increases at a 55%/year rate for the best RISC processor, and at a 40% and then 50%/year rate for Intel processors. If the integer performance of the best RISC processor is roughly two times the performance of the Intel processor, it is not enough to menace the predominance of Intel microprocessors on the PC

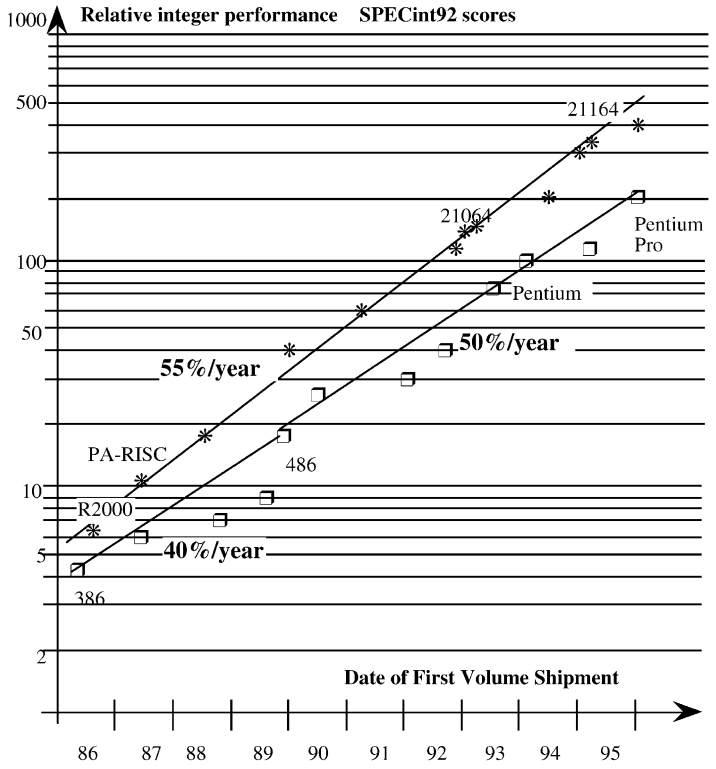

Fig. 1. Relative integer performance for best RISC and Intel microprocessors.

market. Moreover, if we compare RISC and Intel microprocessors of equivalent cost, the performance gap is only 20–40%. What is important is the exponential increase of integer performance.

The reference machine for the SPEC95 suite is a SparcStation 10/40 (1993) with a 40-MHz SuperSparc microprocessor, no secondary cache and a 64-MB main memory. The SPEC92 score for this machine is roughly 41 SPECint92 and 34 SPECfp92. These two values cannot be used as a scaling factor between the two scores, as SPEC95 and SPEC92 do not use exactly the same benchmarks, and when using the same benchmark they do not use the same working set. The SPECfp95 performance for the best mid-1997 workstations, together with the forecast for 1998, are given in Table 1.

The increase in the rate of FP performance is greater than the one of integer performance. However, as we will show in the next sections, these growing rates do not result from a spectacular progress in arithmetic operations. Just the opposite, the performance of arithmetic operations has improved only slightly in the last ten years. It is the progress in all aspects of processor microarchitecture that explains the spectacular progress in performance: exponential growth of clock frequency (about 35%/year), greater instruction rate (from several clock cycles per instruction to several instructions per clock cycle), improved cache hierarchy, improved memory bandwidth, etc.

Table 1
SPECfp95b for today's workstations

Date	Machine	Processor	Clock	SPECfp95b
Mid-97	HP-9000	PA-8200	200	20.1
Mid-97	AlphaStation 600	21164	600	19.9
3Q-98	Digital-Compaq	21264	600	60

Table 2
Latency (in clock cycles) of DP floating-point operations

Microprocessor	FADD	FMUL	FMAC	FDIV	SQR
Alpha 21064	6	6	na	61	
Alpha 21164	4	4	na	22/60	
Alpha 21264	4	4	na	15	32
Ultra Sparc 1	3	3	na	22	22
Ultra Sparc 3	4	4	na	17	24
R10000	2	2	na	19	33
PA-8000	3	3	3	31	31
IBM-P2SC	2	2	2	?	?

2.2. Floating-point operators and performance

Today, every standard microprocessor implements the same set of FP operators: add, subtract, multiply, divide and square root, both with the IEEE-754 single- and double-precision formats. The square root operation defined in the Instruction Set Architecture (Alpha, HP-PA, IA32, MIPS, PowerPC, and SPARC) has been added to the most recent implementations of these ISAs. Some ISAs (HP-PA, PowerPC) implement the multiplication-accumulation operation, which is a four-operand one: $Result = operand1 \times operand2 + operand3$.

In modern microprocessors, the clock cycle is determined either by the access to the instruction/data primary cache, or by the execution time of the integer arithmetic and logic unit for which the critical path is the subtract operation time on 32-bit or 64-bit integer operands. As the FP operations are far more complicated than the integer subtract, they all need several clock cycles to finish. The latency parameter, i.e. the number of clock cycles from the beginning to the end of an operation, depends on the complexity of the operation and on the design philosophy that is used for implementing the ISA.

Most FP operations are pipelined: a new operation can start every clock cycle. Addition and subtraction, multiplication and multiplication-accumulation have a throughput of one instruction/cycle. More complex operations, using a sequential scheme, cannot be pipelined and have roughly the same latency and throughput figures. Tables 2 and 3 give the corresponding numbers for several well-known microprocessors.

Table 3
Throughput (in clock cycles) of DP floating-point operations

Microprocessor	FADD	FMUL	FMAC	FDIV	SQR
Alpha 21064	1	1	na	61	
Alpha 21164	1	1	na	22/60	
Alpha 21264	1	1	na	13	30
Ultra Sparc 1	1	1	na	22	22
Ultra Sparc 3	1	1	na	17	24
R10000	1	1	na	19	33
PA-8000	1	1	1	31	31
IBM-P2SC	1	1	1	?	?

2.3. Impact on performance of three microprocessor features

A good method to evaluate the impact of arithmetic operators on microprocessor performance is to compare some features in the most recent microprocessors with the same features in very old ones. For microprocessor history, 5–10-years old corresponds to very old. The three features that we consider are the FP operators, the memory hierarchy and the branch predictors. The rate of evolution of each feature over time is a good indicator of its impact on the overall performance.

2.3.1. Floating-point operators

The implementation of FP operators strongly depends on the available transistor budget. The first scalar RISC microprocessors, which were implemented in 1985 or 1986 (MIPS R2000 or R3000, Cypress CYC701 implementation of the SPARC ISA) did not have enough transistors to implement the FP operators on the same chip as the CPU: they used a FP coprocessor, implemented on a separate chip. Some years later (1991), with more transistors, the MIPS R4000 implemented the FP operators on the CPU chip, but there was not still enough transistors to implement all FP operations with separate FP operators: several operations shared some units. With this implementation, the FP latency and throughput depend on the data dependencies, i.e. on the scheduling of the instructions by the compiler.

All superscalar microprocessors since the beginning of the 1990s use separate on-chip FP units to implement FP operations. The number of operators of each type is again a function of the transistor budget, which depends on the chip area.

Table 2 presents the latency values and Table 3 presents throughput values both for different ISA implementations (Alpha, Sparc, MIPS, HP-PA and PowerPC) and for different implementations of the same ISA. For instance, 21064, 21164 and 21264 are three successive Alpha chips and UltraSparc 1 and UltraSparc 3 are the first and the last implementations of three different Sun UltraSparc processors. One can compare the values for different ISAs, and the evolution of values between successive generations of a given microprocessor. Some observations can be derived from Table 2:

- Latencies of non-pipelined operations have decreased over time: the most spectacular example is the division with Alpha architecture. There is a four-time improvement

Table 4
SPECfp95 performance for processors with different clock frequencies

Microprocessor	Frequency (MHz)	Feature size (μm)	SPECfp95b
21164	500	0.35	18.3
R10000	200	0.35	17.2
PA-8000	185	0.5	18.3
P2SC	135	0.29	14.5

between the latency of DP FP division between 21064 and 21264. There is also a slight improvement between US-1 and US-3, with a counter-example for the square root operation.

- There is no clear trend for latencies of add and multiply operations. From an older implementation to a newer one, operation latency can decrease (21064 to 21164), remain constant (21164 to 21264) and even increase (US-1 to US-3). It depends on the evolution of clock frequencies from one generation to the next one.

We should point out that the clock latency is not significant by itself. FP operation performance depends both on the operation latencies and the clock frequency. Table 4 shows that similar FP performance on benchmarks can be obtained with clock frequencies ranging from 135 to 500 MHz. The execution time for a given FP operation is approximately the same whatever VLSI implementation is used for a given technology (we will show in this paper that only slight incremental improvement is possible for the basic FP operations). If one processor with the clock frequency F needs 2 clock cycles for one operation, another processor with the clock frequency 2F needs 4 clock cycles for a similar operation. This corresponds to the well-known distinction between the "brainac" and the "speed demon" approaches. As detailed in [10], the first philosophy focuses on powerful instructions and great flexibility in processing order, where the second one depends on a very fast clock, with simpler instructions and a more streamlined implementation structure.

2.3.2. Caches and performance

The significant improvement in transistor budgets available from mid-1980s to now has also led to a significant evolution of the cache structure of standard microprocessors. Fig. 2 shows the most important steps in the evolution. Part (a) corresponds to the cache structure of the first RISC (R2000, R3000) or the Intel 386 microprocessors. The primary cache (L1) is off-chip and there is no secondary cache. Part (b) corresponds to the integration of the primary cache within the CPU chip, generally with separate instruction and data L1 caches. An SRAM-based L2 cache is connected between the CPU chip and the Main Memory. This situation is typical of scalar CPU like the Intel 486, or most of the first superscalar CPUs (Pentium, SuperSparc, 21064, etc). Part (c) corresponds to the next step in cache integration. The CPU includes the L2 cache on chip (21164) or on a separate chip in a common specific package (Pentium

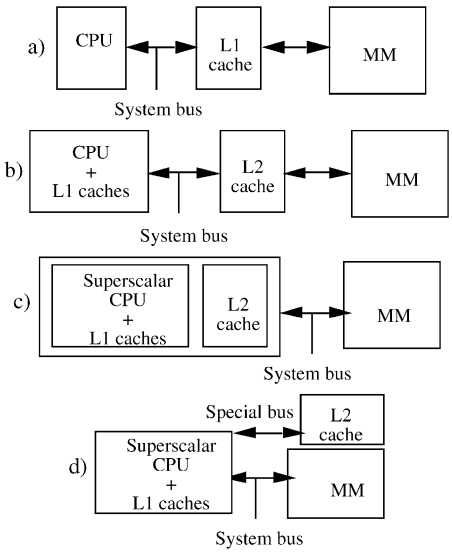

Fig. 2. Evolution of the cache structure of standard microprocessors.

Pro). Here, it is significant that the L2 cache operates at the CPU clock frequency, and not at the system bus clock frequency as in the previous approach. However, the (c) approach has some drawbacks. Being on-chip, the available size is too small for an efficient secondary cache (the 21164 L2 cache has only 96-KB). The Pentium Pro special package is too expensive for low-cost PCs. Part (d) illustrates the most recent step in this evolution. The L2 cache is again off-chip, but it is connected to the CPU by a special bus, operating at half the CPU clock frequency. Most of the recent microprocessors use this approach: 21264, UltraSparc 3 and Pentium II. For the Pentium II, the CPU and the L2 cache share a common special package called "Socket 1".

Table 5 shows the cache sizes for the microprocessors in Table 2, plus some Intel microprocessors. In contrast to the slight improvement in operation latencies in Table 2, we see a significant evolution of cache sizes in Table 4. The size of the instruction and data caches is the maximum size compatible with the transistor budget and a balanced share of resources in the whole microprocessor design. From 21064 to 21264, the on-chip cache size has increased by a factor of 8. The cache size has doubled from UltraSparc 1 to 3. Hewlett Packard, who relied on big off-chip primary caches in all their microprocessors up to the PA-8200, switched to huge on-chip caches, with 0.5 MB for instructions and 1 MB for data in the announced PA-8500. The evolution of memory bandwidth is probably even more significant. Memory bandwidth of several GB/s is currently available in high performance microprocessors, especially those which are intended for numerical and database applications. The P2SC, which is used by IBM for high-end applications, has a relatively slow clock frequency, but it has large on-chip caches and a very high memory bandwidth.

Table 5
Cache sizes and memory bandwidth

Microprocessor	L1-instructions	L1-data	Memory bandwidth
Alpha 21064	8 kB	8 kB	
Alpha 21164	8 kB	8 kB	0.4 GB
Alpha 21264	64 kB	64 kB	2 GB
Ultra Sparc 1	16 kB	16 kB	1.3 GB
Ultra Sparc 3	32 kB	32 kB	2.4 GB
R10000	32 kB	32 kB	0.54 GB
PA-8000	na	na	0.77 GB
PA-8500	512 kB	1024 kB	
IBM-P2SC	32 kB	32 kB	2.2 GB
Pentium	8 kB	8 kB	
Pentium Pro	8 kB	8 kB	
Pentium II	16 kB	16 kB	

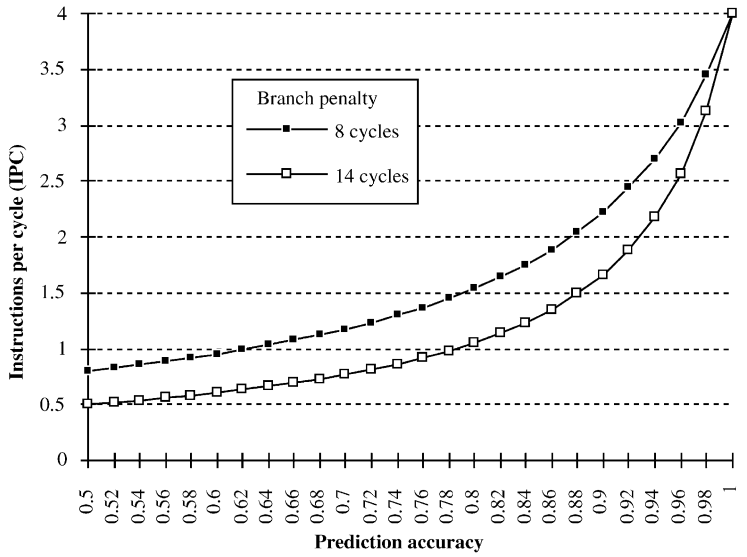

Fig. 3. Actual IPC according to prediction accuracy for a 4-issue superscalar microprocessor.

Tables 2–5 indicate the relatively low importance of arithmetic operations on the overall performance of a computer: accessing data through the memory hierarchy is far more critical than FP operation speed.

2.3.3. Branch prediction and performance

With superscalar microprocessors issuing 4 instructions per clock cycle, the prediction of conditional branches has become much more important than with scalar microprocessors. It is beyond the scope of this paper to go into too many details. To illustrate the problem, we show (Fig. 3) the actual IPC (Instructions executed by Cycle) for a "perfect" superscalar processor fetching 4 instructions per clock cycle,

Table 6
Branch predictor parameters

Microprocessor	Predictor type	Predictor size
Alpha 21064	na	
Alpha 21164	2-bit	4 kB
Alpha 21264	Dynamic 2-level (1G/1L)	35 kB
Ultra Sparc 1	2-bit	1 kB
Ultra Sparc 3	2-bit	32 kB
R10000	2-bit	1 kB
PA-8000	2-bit	0.5 kB
IBM-P2SC	na	

according to the prediction accuracy. The two curves correspond to two values of the branch misprediction penalty: 8 and 14 cycles are examples of the values that can be found with presently used deep pipelines. The figure shows that 88% (resp. 93%) prediction accuracy is necessary to achieve just one-half of the peak performance of the processor, and more than 96% prediction accuracy is needed to get 75% of the peak performance. These numbers do not need much explanation.

Table 6 gives some information on the approach used for branch prediction and the size of the corresponding table. Most referenced microprocessors use local 1-bit or 2-bit counters. The 21264 uses a 2-level scheme, which dynamically chooses the best prediction between the prediction of a global predictor and the prediction of a local predictor. This need for nearly "perfect" prediction of conditional branches has led to several changes in Instruction Set Architecture. The Conditional Move instruction, that has been recently added to all major ISAs, is a minor change which allows a simple implementation of the transformation called "if conversion" in code optimization. The new INTEL ISA, called IA64, which is based on guarded instructions and speculative loads, is a major change in ISA to reduce the impact of conditional branches on overall performance.

2.3.4. Features and overall performance

The examination of three basic features of modern microprocessors shows that the performance of arithmetic operations is far less critical than the performance of memory hierarchy or branch predictors. This does not mean that arithmetic operations have no influence on overall performance. In the next section, we examine the possibilities to improve the performance of FP operations and we discuss the real issues to consider.

3. Improving performance of arithmetic operations

3.1. Reducing latency of floating point-operations

The most important FP functional units are the multiplier and the ALU. Both are pipelined. Fig. 4 shows the main components of the multiplier that are implemented

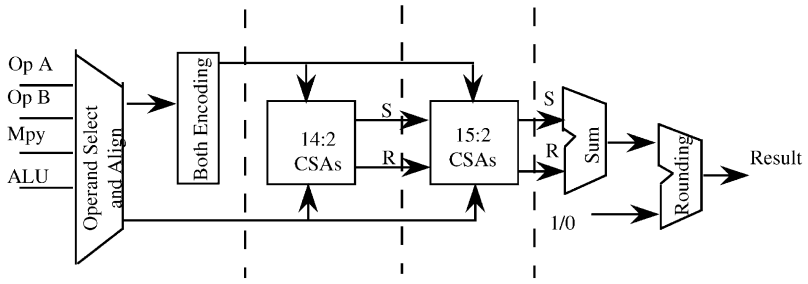

Fig. 4. PA7100 FP multiplier.

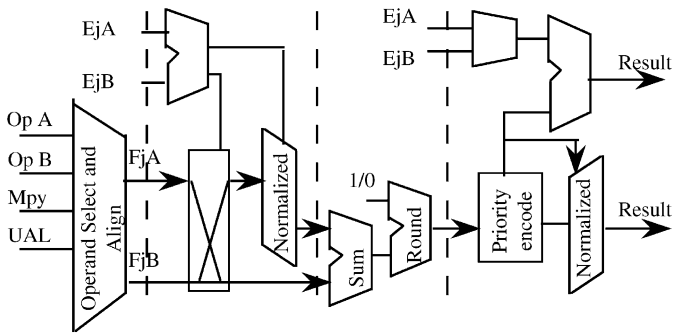

Fig. 5. PA-7100 FP ALU.

in the PA-7100 processor. Fig. 5 shows the FP ALU for the same processor. In both figures, dashed lines separate the different stages of the pipeline for each operation, each stage corresponding to one-half of the clock cycle. Both operations have a 2-cycle latency. Readers familiar with the algorithms for the FP multiply, add or subtract operations can easily recognize the different steps involved in these operations. When the latency of these operations is 2, it is impossible to reduce the clock latency from 2 to 1. When the latency is $n>2$, it is very difficult and often impossible to reduce it from n to $n-1$. Moreover, the latency of pipelined operations is not a critical issue, because compiler techniques like loop unrolling or software pipelining can hide most of the FP operation latency, as shown by Table 7. The table shows the cycle by cycle execution for the DAXPY loop ($y[i]=a\times x[i]+y[i]$ with double-precision operands) for a scalar processor. In this particular example, we use Alpha-like assembly pnemonics and we assume that the latency of integer instructions is 1, the latency of FP loads is 2 and the latency of FP MUL and ADD is 5. With these values, each iteration of the loop needs 13 cycles without any optimization because of the stalls associated with the data dependencies. When the loop is unrolled 4 times, it needs 6 cycles and when it is software pipelined, it needs 9 cycles. In these two cases, there are no pipeline stalls and the latencies of FP operations are totally hidden. With slightly

Table 7
Example of code optimizations to hide pipelined operator latencies

Cycle	Non-optimized code	Unrolled code	Software-pipelined code
1-Loop	LD F1,(R1)	LD F1,(R1)	SD F4,0(R2)
2	LD F2,(R2)	LD F3, 8(R1)	ADDD F4,F2,F3
3	MULTD F1,F0,F1	LD F5,16(R1)	MULTD F3,F0,F14
4		LD F7,24(R1)	LD F1,24(R1)
5		LD F2,(R2)	LD F2,24(R2)
6		LD F4,8(R2)	SUB R6,R7,R1
7		LD F4,16(R2)	ADDI R1,R1,8
8	ADDD F2,F2,F1	LD F4,24(R2)	ADDI R2,R2,8
9	SUB R6,R7,R1	MULTD F1,F0,F1	BNEQ R6,Loop
10	ADDI R1,R1,8	MULTD F3,F0,F3	
11	ADDI R2,R2,8	MULTD F5,F0,F5	
12	SD F2,-8(R2)	MULTD F7,F0,F7	
13	BNEQ R6,Loop	SUB R6,R7,R1	
14		ADDD F2,F2,F1	
15		ADDD F4,F4,F3	
16		ADDD F6,F6,F5	
17		ADDD F8,F8,F7	
18		SD F2,(R2)	
19		SD F4,8(R2)	
20		SD F6,16(R2)	
21		SD F8,24(R2)	
22		ADDI R1,R1,32	
23		ADDI R2,R2,32	
24		BNEQ R6,Loop	

different values for different operation latencies or for superscalar microprocessors, the situation is similar when we use non-unit latency pipelined operations. Reducing the latency of these operations is not essential to get optimal or nearly optimal performance.

3.1.1. Non-pipelined operations

The situation is quite different for non-pipelined operations, such as division or square root. They use an iterative scheme to get the result: as they use the same hardware for each iteration, one operation must complete before the next one can start. Reducing the latency is important if the corresponding operation raises a processor stall, when subsequent instructions are waiting for the results of the div or sqrt instruction. The actual influence of these operations on performance is application dependent. First, it depends on the frequency of these operations in the applications. But even rare non-pipelined operations can degrade performance: it depends on the interlock distances between DIV/SQRT instructions and the consuming instruction. The interlock distance is the number of clock cycles between the producing and the consuming instructions. When the distance is less than the latency of the operation, the processor generally stalls. There are some exceptions that we do not consider here. Oberman and Flynn [7] have extensively studied the impact of non-pipelined FP operations on the overall

Table 8
Latency versus area trade-off for dividers

Divider type	Latency (cycles)	Area (rbe)
1-bit SRT	>40	<3000
2-bit SRT	[20, 40]	3110
4-bit SRT	[10, 20]	4070
8-bit SRT + self-timed	[4, 10]	6665
Very high radix	<4	>100 000

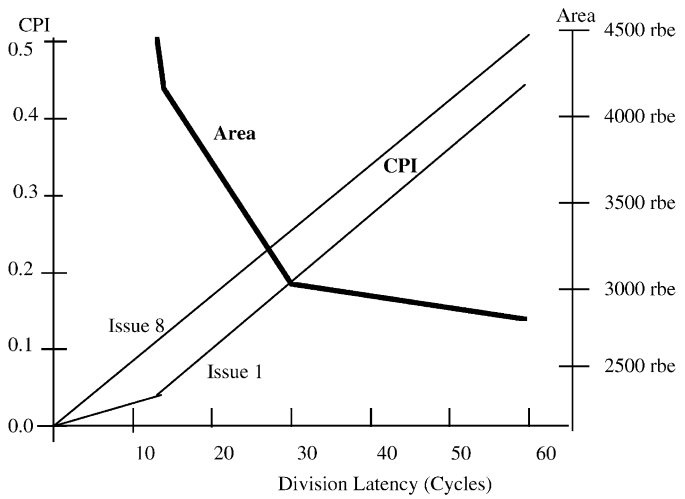

Fig. 6. CPI impact of division and chip area according to division latency.

performance of the processor. As shown in Table 8, the latency of the FP divider results from the latency versus area trade-off. Small dividers using 1-bit SRT algorithm have a large latency. On the other hand, dividers using a very high radix have small latency but very large chip area. In Table 8, latencies are given in clock cycles and chip area is measured in equivalent rbe. (One rbe equals the area of a 1-bit six-transistor static storage cell.)

Fig. 6 gives the additional cycles per instruction (CPI) that result from non-pipelined division operations (thick line) both for scalar (1-issue) and 8-issue superscalar microprocessors according to the division latency. Fig. 6 also gives the chip area (thin line) according to the division latency. As the ideal CPI of an 8-issue processor is 0.125, we can observe that the CPI impact of division can become very significant with high-performance up-to-date microprocessors, even when division operations are very rare. In fact, the real impact on performance depends on many different factors: applications, compiler optimizations, and features of the processor. Compiler optimizations can increase interlock distances. Modern microprocessor features, like register renaming and "out-of-order" execution can reduce the "urgency" of results, because

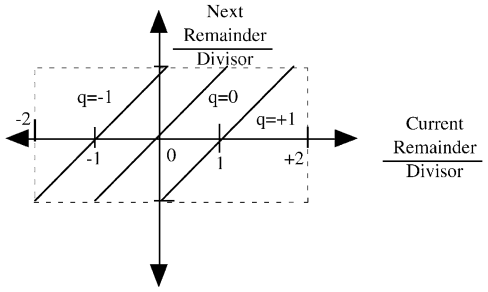

Fig. 7. Radix-2 SRT division diagram.

the processor can continue executing subsequent instructions while it is waiting for the results of the previous ones.

3.1.2. The self-timed radix-2 divider

If compiler techniques and modern microprocessor features reduce the impact of division and square root latencies, they cannot avoid performance penalties in some critical situations, where the non-pipelined latency of the operation is the bottleneck. The following loop, used in [13], illustrates this situation: $z[i] = (a \times x[i] \times x[i] + b \times x[i] + c)/(a \times y[i] \times y[i] + b \times y[i] + c)$. One possible approach to attack the latency problem is to use reciprocal approximation formulas instead of the SRT algorithm, which is currently used for the implementation of dividers in modern microprocessors. The drawback of this approach is that it needs rather large ROMs. One example of this approach is presented in [9]. For IEEE division, the author reports a latency of 9 clock cycles for double-precision numbers with a throughput of 7 clock cycles and a latency of 7 clock cycles for single-precision numbers with a throughput of 5 clock cycles. These results do not show any advantage compared to some implementations of the well-known SRT algorithm [17]. We now briefly present this implementation.

Williams and Horowitz use the radix-2 SRT division algorithm, whose implementation was shown to be more efficient (speed vs. area tradeoff) than using a higher radix. Fig. 7 illustrates graphically a stage of this division algorithm, which uses quotient digits in the set $-1,\ 0,\ +1$. Fig. 8 summarizes the hardware requirements for the SRT division stage. The quotient digit selection is based on an approximation of the partial remainder in each stage, formed by the most significant bits of this partial remainder. Only a 3-bit carry-propagate adder (CPA) is needed to combine the sum and the carry bits examined by the quotient selection logic. All of the less significant bits of the partial remainder can be computed using a carry-save adder (CSA).

Fig. 9 shows how the simple scheme can be transformed to exploit concurrency between operations and advantages of self-timing. Replicating the CPA's for each possible quotient digit allows each CPA to start operation before the actual quotient value is known. The quotient value arrives at the multiplexor to choose the correct result of the 3-bit addition. Each CPA whose input depends on the divisor or the negation of

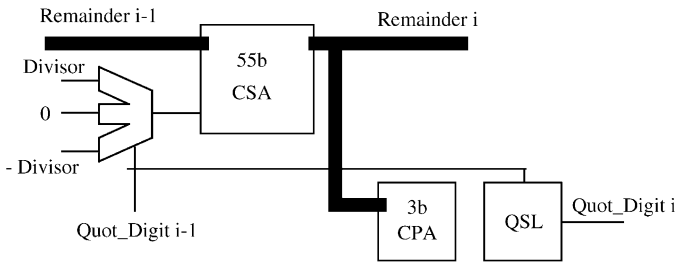

Fig. 8. Data flow required for each SRT division stage.

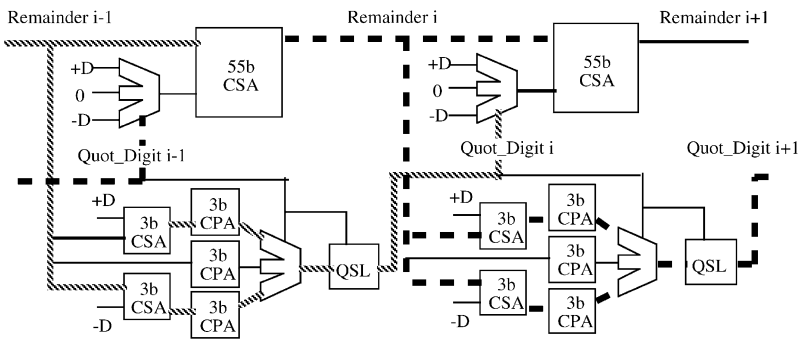

Fig. 9. Data flow through a pair of stages with overlapped execution, showing the two symmetric critical paths.

the divisor is preceded by a 3-bit CSA. As explained in [17], "the overlapping of execution between neighboring stages allows the delay through a stage to be the average rather than the sum of the propagation delays through the remainder and quotient digit selection paths (the corresponding paths are highlighted with dashed and dotted lines in Fig. 9)... If the critical path goes through the quotient path in one stage, it will likely go through the partial remainder path in the next stage, and vice versa".

The previous feature, with average propagation delays instead of sums of propagation delay, can be extended one step further by using the self-timed approach to implement the whole divisor. C-elements and completion detectors have been added to the scheme presented in Fig. 9 to achieve asynchronous control, with forward data propagation, and backward reset propagation, as shown in Fig. 10. This block diagram corresponds to the implementation that was proposed in [17], with a five-stage ring. After the control logic, initialized by the GO signal, has controlled the multiplexor to input the dividend into stage A, the multiplexor is switched to close the loop around the ring. For double-precision operands, the division ring loops a maximum of 11 times to fill the five shift registers with the rest of the quotient digits up to the total of 54 bits that are needed.

We described briefly this self-timed divider because the scheme of Fig. 10 has been implemented in a recent, but not very popular, 64-bit microprocessor, the

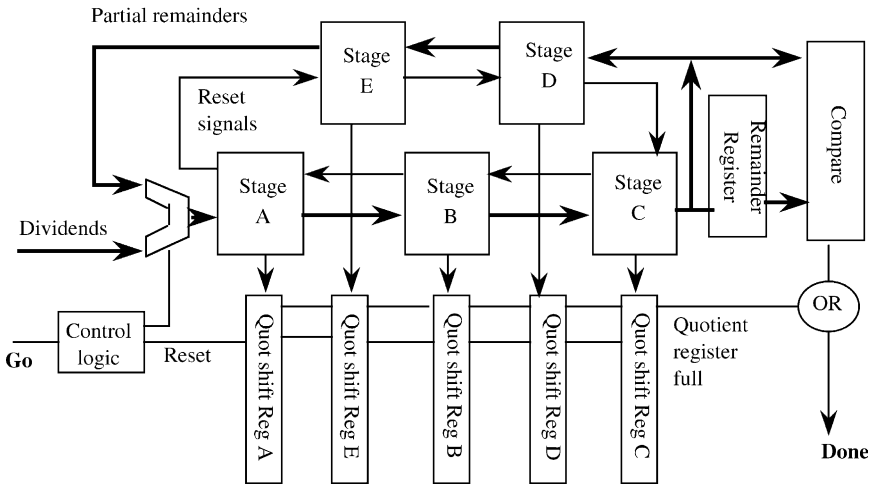

Fig. 10. Block diagram for the division circuit (5 stages which iterate using self-timing).

Table 9
Latencies of SPARC64 arithmetic operators

Operation	SP latency (cycles)	DP latency (cycles)
FP divide	4	7
FP pipelined operations	4	4
Integer divide	2–23 (avg. 9)	2–39 (avg. 17)

SPARC64 [18]. This HAL MCM microprocessor was designed and presented in 1995. The FDIV units can operate in one of the two modes. In one mode, the FDIV unit returns the result at the earliest possible clock cycle. In that case, the latency may vary according to the specific fabrication characteristics, supply voltage or temperature. In the second mode, the FDIV returns its result after a scan-programmed number of clock cycles, using separate values for single- and double-precision. If the second mode is one or two clock cycles slower than the first mode, it is still faster than the synchronous design. As shown in Table 9, the latencies of the self-timed divider are small compared to the values for other microprocessors (Table 2). The SP division latency is the same as the latency of add, mull or multiplication-accumulation. The DP division latency is only 1.75 times the latency of multiplication, compared to 3.75 for the 21124, 4.25 for the UltraSparc-3, 9.5 for the R10000 and more than 10 for the PA-8000.

3.2. Improving performance of pipelined operations

In the previous section, we showed that there is no hope for a significant reduction of latency. This does not mean that performance of pipelined FP operations cannot be

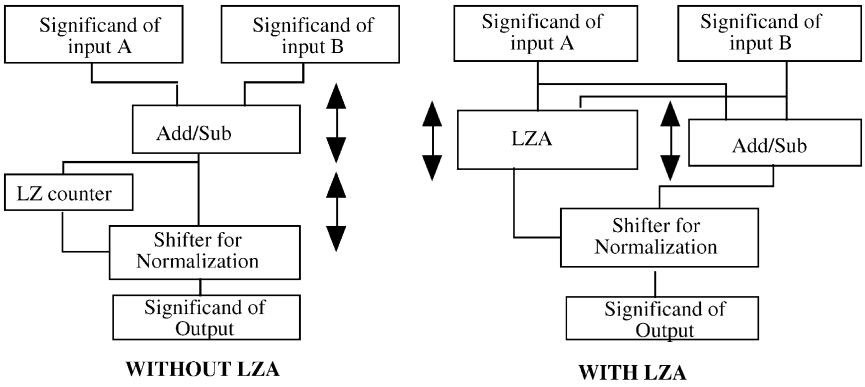

Fig. 11. Example of "logical" optimization in FP ALU: The Leading-Zero Anticipatory Logic.

improved. In this section, we will examine the possible improvements by considering two examples: the FP ALU and the FP multiplier.

3.2.1. The FP ALU

The basic components of the FP ALU have already been presented in Fig. 5. The main building blocks correspond to the main steps of the algorithm that is used to implement the different operations of the ALU. There are very few opportunities for improvement at this "algorithmic" level.

However, some improvement is possible at "logic" levels. The Leading-Zero Logic is a good example of optimization. When doing the subtract of two significands of FP operands, it is necessary to calculate the number of leading zeros to do the following shift for normalization. In the classical implementation, shown in the left part of Fig. 11, the number of leading zeros is computed after the Sub operation by a LZ counter. Sub and LZ count operations are sequential ones. By implementing the Leading-Zero count from the two significand inputs instead of the sub output [12], the Sub operation and the LZ operation (now called LZA) can be realized in parallel, with the shortest critical path.

If the "logical" optimizations is worth considering, it is clear that most of the potential improvement comes from the optimization at "transistor" level, with the best implementation of the Adder/Substractor according to the CMOS circuit styles. We will give one example for the FP multiplier.

3.2.2. The FP multiplier

The FP multiplier has been widely studied for many years. This circuit is so classical that it is a typical arithmetic benchmark circuit, on which the evolution of performance has been examined year after year in VLSI journals such as the IEEE Journal of Solid-State Circuits. Two main options can be used with or without redundant number representations. However, the most commonly used scheme for a 54×54-bit FP multiplier is presented in Fig. 12.

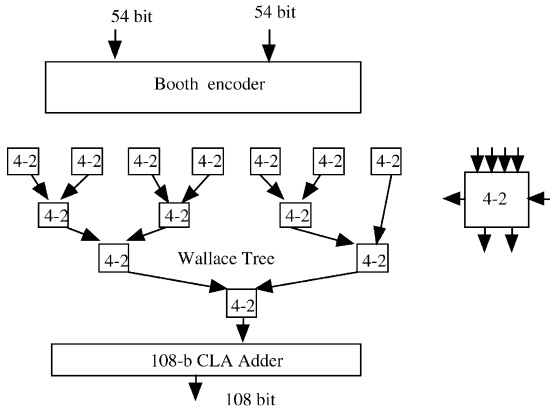

Fig. 12. Classical "functional scheme" for a 54×54-bit multiplier.

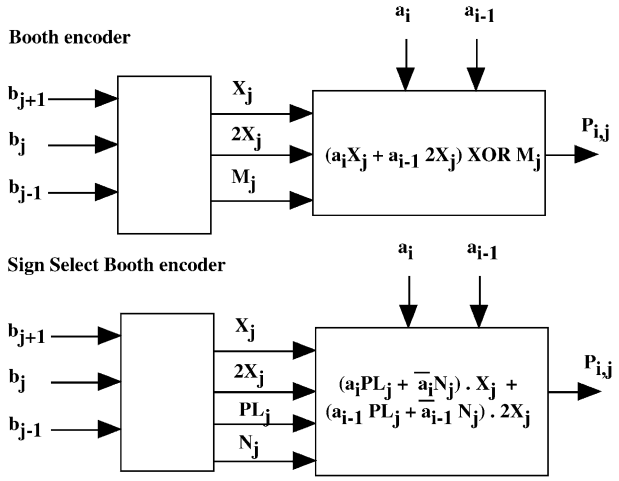

Fig. 13. Optimization at "logic level" in Booth encoders.

The 54-bit input multiplicand is decomposed into 27 54-bit summands by the Booth encoder. Then, a Wallace tree of $4-2$ compressors reduces the summands into two final summands that are finally added in a 108-bit CLA adder. Once again, there is no room for a significant improvement in the "functional level" of the multiplier.

As for the FP ALU, some improvement can be obtained at the "logic" level, especially with the Sign Select Booth Encoder. This is illustrated in Fig. 13. The Booth Encoder delivers the $P_{i,j}$ bit according to the a_i a_{i-1} pair of bits of the multiplicand and the b_{j+1} b_j b_{j-1} set of bits of the multiplier. The Booth encoder implements an XOR operation. The variant called Sign-select Booth encoder [3] replaces the XOR operation by a set of usual AND and OR operations. As the XOR operation is more complex to

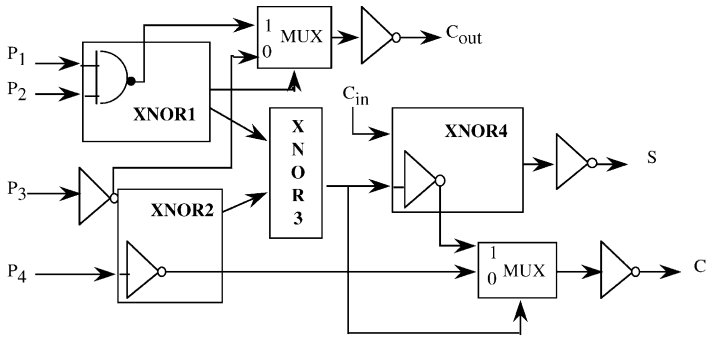

Fig. 14. Implementation of the 4–2 compressor in [3].

Table 10
Implementation of the 4–2 compressor

Gate	Transistor	Implementation
XNOR1	10	Nand + Or-Nand
XNOR2	8	2 inverters + 2 transmission gates
XNOR3	6	Floating inverter + transmission gates
XNOR4	8	2 inverters + 2 transmission gates
MUX	4	2 transmission gates
Inverters	2	

implement in CMOS technology than any other usual Boolean operations, hence the new implementation without XOR gates is more efficient.

Most of the performance improvements come from VLSI implementation of the multiplier. One key issue is the most efficient implementation of the main basic bloc, which is the 4–2 compressor. Without going into details, we can notice that the 4–2 compressor used in [3] uses 4 XNOR gates with 3 different implementations as shown in Fig. 14 and Table 10. If the theoretical critical path between inputs and outputs of the 4–2 compressor is 3 XNOR gates, it is important to customize the implementation of these XNOR gates to minimize the value of the critical path according to the fan-out of each gate.

The overall implementation of the compressor array is the second key issue. The most efficient method for optimizing the implementation has been presented in [8]. It uses full adders, but could be extended to 4–2 compressors. It is based on the fact that the delays between inputs and outputs are not equal. In the 4–2 compressor presented in Fig. 14, the delay between C_{in} and S is smaller than the delay between any P_i input and S output. Similarly, the delay between P_3 or P_4 and C is smaller than the delay between the same inputs and S. Some inputs can be called "fast" inputs and some outputs can be called "fast" outputs. The algorithm described in [8] considers the entire multiplier array, which is called the Wallace tree in Fig. 12, and minimizes

delays in the whole array by proper ordering of the "fast" and "slow" input signals. The method has been generalized in [11].

From the ALU and multiplier examples, we can observe that the most significant progress has come from optimization of the VLSI implementation, both at the circuit and layout levels. It is a global VLSI optimization problem, in which arithmetic issues are a specific part.

4. Arithmetic needs for MMX technology?

The importance of media (video, audio, graphics, and communication) applications is continuously growing in the personal computing business. They have led to a significant extension of most of the current Instruction Set Architectures, called MMX for the Intel IA32, VIS for the Sparc ISA and MVI for the Alpha ISA. It is beyond the scope of this paper to fully describe the media extension of these instruction sets. We only consider the arithmetic issues associated with the MMX technology. Any other multimedia extension could be used as they share many common features.

Instead of having only the usual fixed formats (32 or 64 bits for integers, 32 bits (SP) or 64 bits (DP) for floating-point representation), MMX has several subformats. A 64-bit MMX register contains either a 64-bit quadword, or two 32-bit doublewords, or four 16-bit words or eight 8-bit bytes (using Intel terminology). MMX instructions use the SIMD approach: they implement a parallel operation on each part of the sub-word (B, W, and DW). Main SIMD instructions are the arithmetic, logic, compare, shift, pack and unpack instructions. Arithmetic instructions use signed or unsigned operands with unsaturated or saturated operations. Compared to a normal 64-bit ALU, the SIMD ALU should be able to operate on sub-ALUs, whose length corresponds to the operand length, without propagating carries through the boundary between two successive sub-ALUs. For each operand length, the sub-ALU operates either in normal or in saturated mode. There is no particular implementation problem. As presented in [5], "each SIMD adder is capable of performing add, subtract and compare of 8-byte, 4-word and 2-doubleword data types. The adders are optimized to perform these operations with roughly the same speed as a normal 32-bit adder".

From an architectural point of view, the big question is the mapping of the MMX registers in the register spaces of the processor. A costly solution would be to define a new set of MMX registers. The other option is to use either integer or FP registers for MMX registers. Only Alpha ISA uses the first approach. Sparc and Intel use the second approach. FP operations and MMX operations are generally exclusive. So, these two formats can share the same registers without conflicts. With this approach, the MMX instructions can use the same multicycle latencies as the FP instructions, which makes the implementation of the most critical MMX instructions easier. In that case, there are no specific arithmetic problems and implementing MMX instructions is only a VLSI problem. With the Alpha approach, it is far more complicated to make the media instructions compatible with the timing of the integer pipelines.

Table 11
PERR instruction timing table

1A	1B	2A	2B
$S10_{0:7} = \lvert A_{0:7} - B_{0:7}\rvert$	$S20_{0:8} = S10_{0:7} + S11_{0:7}$	$S30_{3:9} = S20_{3:8} + S21_{3:8}$	$S40_{4:10} = S30_{4:9} + S31_{4:9}$
$S11_{0:7} = \lvert A_{8:15} - B_{8:15}\rvert$	$S21_{0:8} = S12_{0:7} + S13_{0:7}$	$S31_{3:9} = S22_{3:8} + S23_{3:8}$	
$S12_{0:7} = \lvert A_{16:23} - B_{16:23}\rvert$	$S22_{0:8} = S14_{0:7} + S15_{0:7}$		
$S13_{0:7} = \lvert A_{24:31} - B_{24:31}\rvert$	$S23_{0:8} = S16_{0:7} + S17_{0:7}$	$S40_{0:3} = S30_{0:2} + S31_{0:2}$	
$S14_{0:7} = \lvert A3_{32:39} - B_{32:39}\rvert$			
$S15_{0:7} = \lvert A_{40:47} - B_{40:47}\rvert$	$S30_{0:2} = S20_{0:2} + S21_{0:2}$		
$S16_{0:7} = \lvert A_{48:55} - B_{48:55}\rvert$	$S31_{0:2} = S22_{0:2} + S23_{0:2}$		
$S17_{0:7} = \lvert A_{56:63} - B_{56:63}\rvert$			

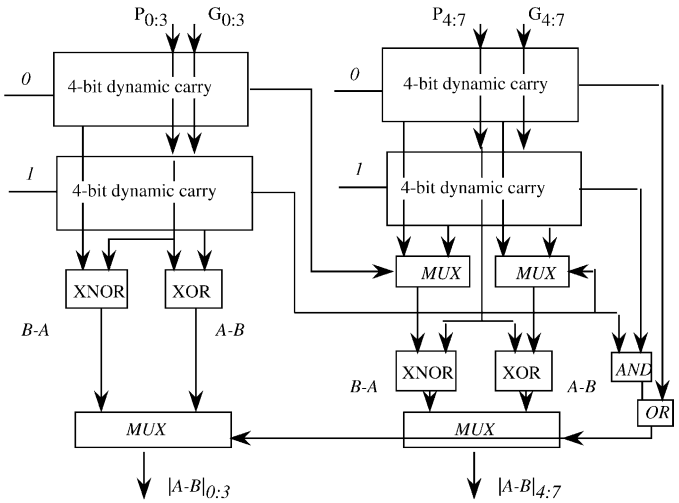

Fig. 15. Eight-bit difference logic in 21164PC microprocessor.

As an example of this problem, we show how the pixel error instruction (PERR) is implemented in the 21164PC [1], which is a 550-MHz Alpha processor implemented with a 0.35-μm technology. The PERR instruction calculates the sum of absolute difference of pairs of eight bytes. The design constraint is that a 2 clock implementation of this instruction be compatible with the normal integer pipeline of the 21164PC. Table 11 shows how the calculation is decomposed into four half-cycles. In phase 1B, the low-order bits of S20 and S21 (resp. S22 and S23) are added to get the low-order bits of S30 (resp. S31) before getting the high-order bits of S20 and S21 (resp. S22 and S23). This way, S30 and S31 are calculated in phase 1B (low-order bits) and 2A (high-order bits) and S40 is calculated in phase 2A (low-order bits) and 2B (high-order bits). The circuit diagram for the eight-bit absolute difference logic is shown in Fig. 15.

5. Arithmetic and asynchronous microprocessors

In synchronous microprocessors, all clocking signals are derived from a common global clock, and any register or flip-flop is assumed to be controlled by the same "synchronous" clock signal. As explained in [19], "Today's VLSI chips incorporate very large numbers of transistors and it is becoming increasingly difficult to maintain global clock synchrony over a large chip area... As clock rates rise and the chip area over which the clock must be distributed expands, clock skew becomes ever more difficult to control. Remarkable engineering techniques have been employed to contain the problem, but only at the cost of considerable silicon area and high peak supply currents".

To overcome these limitations, computer architecture researchers are actively considering asynchronous processor design. Asynchronous architectures by nature allow modular design. Each functional block can be optimized without being synchronized to a global clock, which simplifies interfacing. The main argument is that an asynchronous system exhibits the average performance of all the individual components, rather than the synchronous worst-case performance of a single component. Simplified control and reduced power dissipation are other arguments for the asynchronous approach. Interested readers will find more details about asynchronous approach in [16, 19]. The first one examines the key architecture issues that concern designers and compares six developmental asynchronous architectures. The second one details the asynchronous implementation of the ARM microprocessor; it also presents all the asynchronous background that is needed to understand the design choices.

A study of asynchronous approach for arithmetic operators has begun. It tries to quantify the potential advantage of the "average delay" versus the worst case delay of the synchronous approach. In [2], the authors' present statistical carry-lookahead adders, whose average delay is much lower than $\log_2(N)$ and whose overhead is lower than for the CLA adder. A similar study can be found in [14] for ripple-carry adders, carry-skip adders and carry-select adders.

For computer architects, the relationship between arithmetic issues and the performance of the overall architecture should be considered as carefully for the asynchronous approach as for the synchronous one. For synchronous architectures, we have seen that the improved performance results mainly from the instruction rate (throughput) of the operations and not from the latency. Do reduced latencies for asynchronous arithmetic operations mean increased IPC (instruction throughput)? This is an open question, and only experimental results can give strong arguments in favor or against the asynchronous approach. Asynchronous implementation has also some traditional drawbacks: can we easily and efficiently implement precise interrupts? What about context switches?

The fundamental question is whether the asynchronous approach is a good alternative for implementing general-purpose microprocessors. In that case, asynchronous processors should exhibit similar or better performance than synchronous ones, even for such features as precise interrupts or context switches. Or, are the features of

the asynchronous approach more suitable for specialized "embedded" microprocessors? Reduced power dissipation is one of these features. Another alternative is to consider an asynchronous implementation of some parts of an overall synchronous implementation. The self-timed FP divider, which is used in the HAL64 microprocessor, is a good example of an efficient use of the asynchronous approach within a synchronous processor.

6. Specific hardware for specific arithmetic representations

The floating-point number system is widely used for representing real numbers in computers, but many other number systems have been proposed to achieve various goals: improve the accuracy, avoid overflows or underflows, accelerate some computations. Logarithmic number representation and serial arithmetic are two examples of non-standard number representations. As any arithmetic, they can be implemented by software or by hardware. In this section, we consider the hardware implementation of specific arithmetic representations by discussing two possible approaches: the custom or semi-custom VLSI implementation and the programmable logic approach.

6.1. The coprocessor approach

Designing a customized VLSI circuit, defined as a coprocessor of a general purpose microprocessor, has been considered as the natural way to implement specific hardware operations that cannot be directly implemented within the microprocessor. This approach derives from the "rules of thumb" that are used in microprocessor design: the frequently used operations should be fast and thus implemented by hardware. The rare operations should be implemented in software. As non-standard arithmetic representations are only used for some specific applications, the corresponding operations are rare. In that case, the goal of an arithmetic coprocessor is to speed up the execution of these operations compared to their execution by software. However, the coprocessor also has to deal with the performance/cost ratio issue.

As we already mentioned in this paper, an exponential increase in microprocessor performance has been attained during the last 25 years. Without going into details, this evolution continues because of the following items:

- New CMOS technologies are regularly becoming available to processor designers. If 0.25 μm feature size is common in 1998, future 0.18 μm CMOS technologies have already been announced. The scaling of CMOS technologies leads to increased clock frequencies and larger transistor budgets (chip area).
- Each processor manufacturer delivers several releases of each microprocessor model.
- New microprocessors are designed for a given Instruction Set architecture. For the ×86 ISA, the 486, the Pentium and the Pentium Pro (and Pentium II) represent three different microarchitectural implementations of the same ISA: the 486 was a scalar microprocessor, the Pentium was a statically scheduled 2-way superscalar

microprocessor, and the Pentium II (following the Pentium Pro) is an "out-of-order" superscalar microprocessor, with on-the-fly translation of ×86 to "RISC-like" instructions.
- New ISAs are announced. The Intel IA64 is supposed to kill the old IA32 (×86) ISA.

This short "life cycle" for each microprocessor release, which corresponds to the Moore's law, is economically possible only because of the huge market for PCs. The microprocessor sales range in millions of components. If coprocessor releases do not follow the microprocessor releases, performance mismatches will occur. But the coprocessor releases cannot follow the microprocessor releases for business reasons: the market for the coprocessors is too small, because most of the applications do not need these specific arithmetic representations. The coprocessor approach is a dead end because it uses a costly approach, either with custom or semicustom VLSI design.

6.2. Programmable logic devices

Hopefully, it is possible to implement specific hardware in a cost-effective way by using the programmable logic approach. Programmable logic devices (PLD), for which Altera is a supplier, or field programmable gate arrays (FPGA), for which Actel, Altera or Xilinx are examples of suppliers, are typical examples of low-cost hardware support. Interested readers will find more detail on programmable logic at the following WEB sites: www.{altera, actel, xilinx}.com. Implementing specific number representations may be quite simple if the user is provided a "programming" environment, e.g. a set of arithmetic operators, a methodology for control and a validation system.

6.2.1. Serial arithmetic in FPGAs

Tisserand [15] has developed a framework for on-line algorithms in FPGAs. This framework includes
- A VHDL library, with packages, elementary circuits (FA cell, PPM cell, etc.), on-line operators (adders, multipliers, dividers, etc.).
- A methodology for control.
- A validation system.

The framework has been used to developing two significant applications. In LIP in Lyon, a multilayer perceptron has been implemented [14]. It uses the following equation: $s = \tanh(\theta + \Sigma w_i x_i)$.

In EPFL in Lausanne [15], a proportional integral–differential regulator has been designed for positioning a mirror. The size and power dissipation of the FPGA implementation are, respectively, 1/4 and 1/20 compared to the corresponding DSP implementation.

6.2.2. Logarithmic or semi-logarithmic representation

Other non-standard representations could be implemented with PLDs or FPGAs. Some applications, such as signal and image processing, some transforms, numerical

control and wavelets use far more MUL, DIV or SQRT operations than ADD/SUB operations. In this case, logarithmic or semi-logarithmic representations seem to be more effective [6]. Once again, the FPGA approach is the only cost-effective hardware implementation when using these number representations.

7. Concluding remarks

Discussing computer arithmetic and hardware relationships means considering the performance/cost issues. We showed that computer arithmetic is relatively marginal in the performance growth of standard microprocessors. The main reason is that most arithmetic operations easily meet the requirements of modern microprocessors. At the same time, the mismatch between processor performance and the main memory performance is growing, making it far more critical to get data quickly into the processor. With superscalar microprocessors, the control flow of instructions through branches is more and more critical, and branch predictions have more impact on performance than arithmetic operations.

However, there is still room for improving arithmetic operations, especially the latency of non-pipelined operations like the division and square root. If this was possible, the pipelining of the division would be a real breakthrough.

As arithmetic performance is only a part of the overall performance of a microprocessor, arithmetic issues must be considered with all the architectural issues, and not alone. The most important points are:

- The benchmarking of applications. Before doing a lot of work to improve latency of an operation, we must know what is the real impact of this operation on the overall performance. Moreover, we must be sure that this operation is really useful and even used.
- Improving performance of arithmetic operations needs considering the VLSI implementation of these operations. There are very few opportunities for a significant improvement just at the algorithmic level.
- Recent arithmetic bugs in Intel microprocessors, both on the Pentium and the Pentium II, show that, as computer arithmetic is a VLSI problem, it is also a design-checking problem.
- While arithmetic "breakthroughs" are still hoped for, "incremental" progress is very helpful.

Specialized applications also cannot avoid the performance/cost issues. However, the situation has significantly changed in recent years, as large scale PLDs and FPGAs now exist and are the cost-effective solution for "specialized" arithmetic. They are general-purpose circuits to customize hardware according to the needs of a specialized arithmetic representation. So, designing arithmetic circuits is not only a hobby for Ph.D. students (and university professors). By using a "cost-effective" approach, it can lead to useful designs.

Acknowledgements

I would like to thank Professor Zvonko Vranesic of the University of Toronto in Ontario, Canada for his help in correcting and polishing English in this article.

References

[1] D.A. Carlson, R.W. Castelino, R.O. Mueller, Multimedia extensions for a RISC 550-MHz RISC microprocessor, IEEE J. Solid-State Circuits 32 (11) (1997) 1618–1624.
[2] A. De Gloria, M. Olivieri, Statistical carry lookahead adders, IEEE Trans. Comput. 45 (3) (1996) 340–347.
[3] G. Gotoa, A. Inoue, R. Ohe, S. Kashiwakura, S. Mitairi, T. Tsuru, T. Izawa, A 4.1-ns compact 54×54-b multiplier utilizing sign-select booth encoders, IEEE J. Solid-State Circuits 32 (11) (1997) 1676–1681.
[4] J.L. Hennessy, D. Patterson, Computer Architecture: a Quantitative Approach, 2nd ed., Morgan Kaufmann, Los Altos, CA, 1996.
[5] M. Mittal, A. Peleg, U. Weiser, MMX^{TM} technology architecture overview, Intel Technol. J. Q3 (1997) 1–12.
[6] J.-M. Muller, A. Scherbyna, A. Tisserand, Semi-logarithmic number systems, IEEE Trans. Comput. 47 (2) (1998) 145–151.
[7] S. Oberman, M. Flynn, Design issues in division and other floating-point operations, IEEE Trans. Comput. 46 (2) (1997) 154–161.
[8] V.G. Oklobdzija, D. Villeger, S.S. Liu, A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach, IEEE Trans. Comput. 45 (3) (1996) 294–305.
[9] P.M. Seidel, High-speed redundant reciprocal approximation, Proceedings of Third Real Numbers and Computers, Paris, April 1998, pp. 219–229.
[10] J. Smith, S. Weiss, PowerPC 601 and Alpha 21064: a tale of two RISCs, Computer 27 (6) (1994) 46–58.
[11] P. Stelling, C. Martel, V. Oklobdzija, R. Ravi, Optimal circuits for parallel multipliers, IEEE Trans. Comput. 47 (3) (1998) 273–285.
[12] H. Suziki, H. Morinaka, J. Makino, Y. Nakase, K. Mashiko, T. Sumi, Leading-zero anticipatory logic for high-speed floating point addition, IEEE J. Solid-State Circuits 31 (8) (1996) 1157–1164.
[13] P. Tirumalai, D. Greenley, N. Beylin, K. Subramanian, UltraSPARC^{TM}: compiling for maximum floating-point performance, Proceedings of COMPCON'96, pp. 408–416.
[14] A. Tisserand, Adéquation Arithmétique Architecture, Problèmes et Etudes de cas, Ph.D. dissertation (in french), Ecole Normale Supérieure de Lyon, September 1997.
[15] A. Tisserand, M. Dimmler, FPGA implementation of real-time digital controllers using on-line arithmetic, in: Field Programmable Logic and Applications, Springer, Berlin, 1997.
[16] T. Werner, V. Akella, Asynchronous processor survey, Computer 30 (1997) 67–76.
[17] T. Williams, M. Horowitz, A zero-overhead self-timed 160-ns 54-b CMOS divider, IEEE J. Solid-State Circuits 26 (11) (1991) 1651–1661.
[18] T. Williams, N. Patkar, G. Shen, SPARC64: a 64-b 64-active-instruction out-of-order-execution MCM processor, IEEE J. Solid-State Circuits 30 (11) (1995) 1215–1225.
[19] J.V. Woods, P. Day, S.B. Further, J.D. Garside, N.C. Paver, S. Temple, AMULET1: an asynchronous ARM microprocessor, IEEE Trans. Comput. 46 (4) (1997) 385–397.