A 320-MFLOPS CMOS Floating-Point Processing Unit for Superscalar Processors

Nobuhiro Ide, Hiroto Fukuhisa, Yoshihisa Kondo, Takeshi Yoshida, Masato Nagamatsu, *Member, IEEE*, Junji Mori, Itaru Yamazaki, and Kiyoji Ueno

Abstract— **A CMOS pipelined floating-point processing unit (FPU) for superscalar processors is described. It is fabricated using a 0.5-μm CMOS triple-metal layer technology on a 61-mm^2 die. The FPU has two execution modes to meet precise scientific computations and real-time applications. It can start two FPU operations in each cycle, and this achieves a peak performance of 160 MFLOPS double or single precision with an 80-MHz clock. Furthermore, the original computation mode, twin single-precision computation, doubles the peak performance and delivers 320-MFLOPS single precision. Its full bypass reduces the latency of operations, including load and store, and achieves an effective throughput even in nonvectorizable computations. An out-of-order completion is provided by using a new exception prediction method and a pipeline stall technique.**

I. Introduction

High-performance floating-point processing units (FPU's) for microprocessors have been developed [1]–[8]. Recent FPU's are integrated with a RISC core processor, cash memory, and memory management unit on one chip and these processors adopt parallel processing such as in a superscalar architecture. Along with the improved performance of microprocessors or systems, it is essential to improve the performance of the FPU. However, the hardware resources of these FPU's were not fully exploited and they are not well adapted to a superscalar architecture.

In this paper, a new FPU for superscalar processors is described. It achieves high performance through the interaction of design choices that meet a high performance of RISC core.

This FPU is designed to be tightly coupled with the RISC core and adopts a RISC approach to meet its speed requirements. This eliminates coprocessor protocol such as instruction-issuing overhead and data transfer, and accomplishes low latency of arithmetic operations.

The FPU adopts two scalar architectures to make effective use of machine parallelism and it permits out-of-order completion not only between FPU operation and core operation but also between FPU operations.

A new exception prediction technique achieves precise exception handling for recovery and restart mechanisms. The prediction and handling technique for the superscalar processors has been improved.

Symmetrical full bypass reduces the latency and speeds up program execution. This achieves flexible connections between the execution units and an efficient data flow.

Two types of execution mode are implemented to achieve high performance regardless of the kind of application, because more and more applications require floating-point computing. They are well adapted to both of scientific applications and real-time applications.

In the following section, an overview of the FPU is described. Two types of execution modes are described in this section. In Section III, the architecture and implementation are described. Sections IV, V, and VI describe the features of the FPU. In Section IV, full bypass is described. The original computation mode called twin single-precision computation appears in Section V. Exception prediction and handling are described in Section VI. In Section VII, design methodology is described. Conclusions are presented in Section VIII.

II. Overview of the FPU

An 80-MHz floating-point processing unit for superscalar processors with a RISC-type core processor has been designed. It is fabricated using a 0.5-μm CMOS triple-metal layer technology, and contains about 290K transistors on a 61-mm^2 die. Table I shows the features of the FPU. It consists of a fully independent ALU, a multiply/divide unit (MDU), and a register file. The FPU can start two new floating-point operations for the ALU and the MDU in every clock cycle. This architecture delivers a peak performance of 160 MFLOPS double or single precision. Furthermore, 320-MFLOPS single-precision operation is obtained because each execution unite can compute two sets of single-precision data concurrently.

The FPU has two types of execution modes. One is the *scientific mode* and the other is the *real-time mode*. The scientific mode is implemented for precise scientific computations and engineering applications that need high precision and much execution power. This mode completely conforms to the IEEE-754 standard [9], including all four rounding modes and exception status reporting, for a software exception handling routine or a hardware exception handling unit, even in an out-of-order execution case. On the other hand, the real-time mode conforms to a subset of the IEEE-754 standard. In this mode, denormalized numbers and infinity are identified as zero and maximum. The FPU runs without an exception trap and software exception handling. All exceptions are handled by the

Manuscript received July 25, 1992; revised October 26, 1992.

N. Ide, H. Fukuhisa, Y. Kondo, and T. Yoshida are with the ULSI Research Laboratories, Toshiba Corporation, 1, Komukai, Toshiba-cho, Saiwai-ku, Kawasaki 210, Japan.

M. Nagamatsu, J. Mori, I. Yamazaki, and K. Ueno are with the Semiconductor Device Engineering Laboratory, Toshiba Corporation, 1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki 210, Japan.

IEEE Log Number 9206366.

TABLE I
FPU FEATURES

Size	61 mm^2
Process	0.5 μm (gate length)
	CMOS triple-metal layer technology
Supply voltage	3.3 V
Frequency	80 MHz
Peak performance	320 MFLOPS
	(twin single-precision @80 MHz)
	160 MFLOPS
	(double/single-precision @80 MHz)

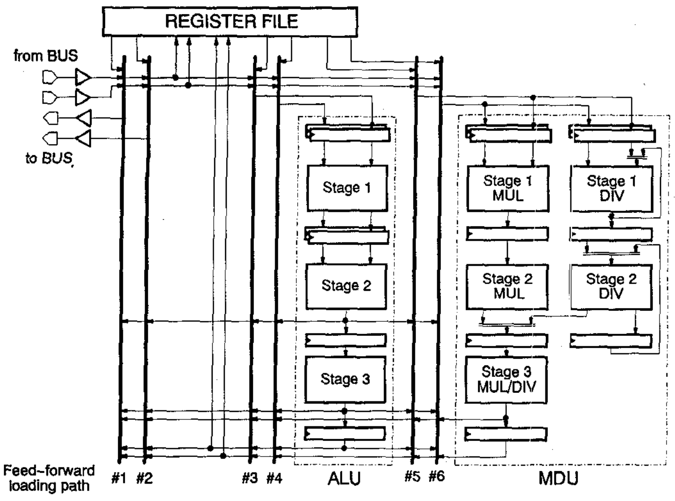


Fig. 1. Block diagram.

hardware exception handling unit. This unit outputs the default numbers such as a quite not-a-number (QNaN), maximum, and zero, according to the exception. Furthermore, this mode supports the computation mode called twin single-precision computation and achieves 320 MFLOPS. This mode is very useful for real-time applications because of no trap handling and greater execution power.

III. ARCHITECTURE

The block diagram of the FPU is shown in Fig. 1. The ALU, the MDU and the register file employ full 64-b-wide data paths. They can compute double-precision data directly.

Table II shows all supported operations, their latency and throughput. Most operations are completed within three execution cycles. These operations provide the basis for systems conforming with the requirements and recommendations of the IEEE-754 standard, including the original operation "swap." The FPU adopts the RISC approach in the execution units, and the execution pipelines are simple in design for high speeds that the RISC-type core requires. All frequently used operations are directly implemented in the hardware as shown in Table II. The hardware transfers a rare denormalized number, QNaN, and infrequent complex exceptions defined in the IEEE-754 standard to its closely coupled software exception handling routine for processing.

TABLE II
SUPPORTED OPERATIONS

Module	Operation	Latency (# of clocks)	Throughput (# of clocks)
ALU	Addition, Subtraction	3	1
	Convert to double-precision	3	1
	Convert to single-precision	3	1
	Convert to 32-bit integer	3	1
	Compare	1	1
	Absolute	2	1
	Negate	2	1
	Swap	2	1
	Move	2	1
MDU	Multiplication	3	1
	Division	17 (Double)	14
		10 (Single)	7

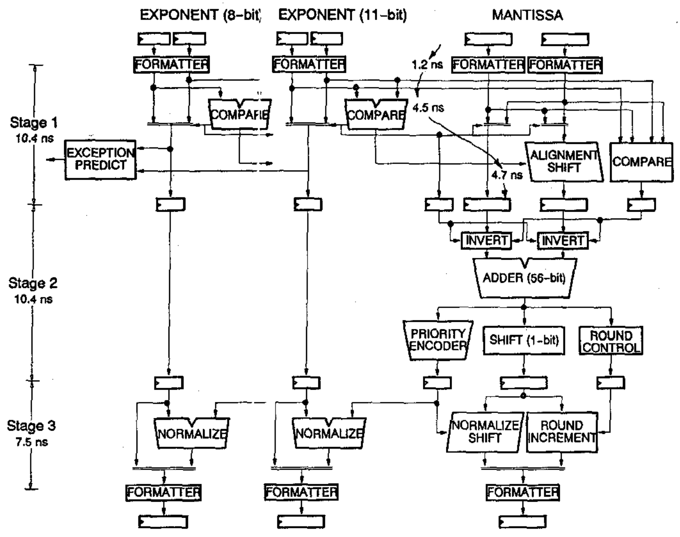


Fig. 2. ALU block diagram.

A. ALU

The ALU block diagram is shown in Fig. 2. It is fully three-stage pipelined so that it can start a new ALU operation in each cycle. It performs most floating-point operations as shown in Table II. "Swap" operation performs swapping the 32 most significant bits and the 32 least significant bits of data. This is implemented to access single-precision data that is stored in the 32 most significant bits of the register. The "compare" operation takes one cycle to report the condition by a flag bit which shows true or false according to the condition code such as "equal," "greater than," "less than," and so on. This flag can be tested by a "floating-point condition jump" instruction of the core processor at the next cycle, and a program sequence can jump quickly.

The first stage performs exponent comparison and mantissa alignment. A critical speed path at the first stage consists of the formatter, the exponent comparator, the swapper, and the alignment barrel shifter, as shown in Fig. 2. Pass transistors are used for selectors in these modules in order to achieve a high speed and small area. For example, a 1-b cell of the shifter

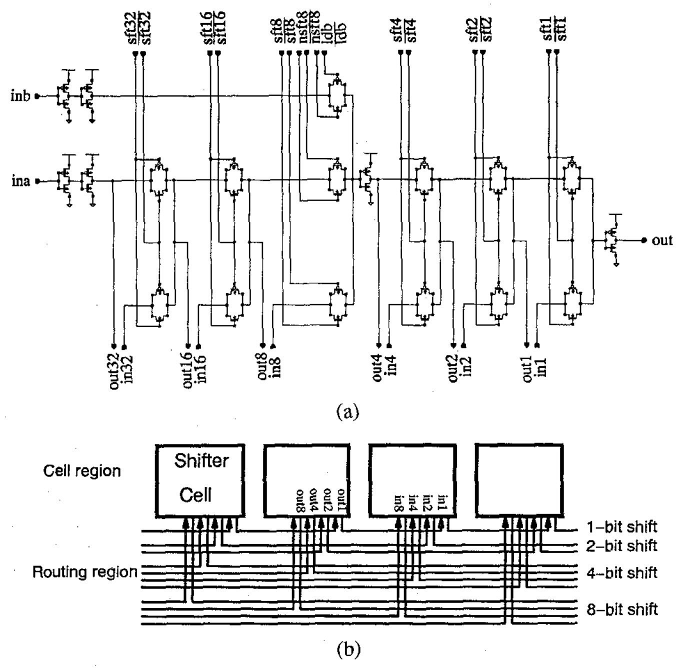


Fig. 3. Barrel shifter. (a) Shifter cell. (b) Layout image.

is shown in Fig. 3(a). The cell region and the routing region were laid out separately, as shown in Fig. 3(b). It reduces routing tracks and power supply lines in comparison with a conventional layout.

The second stage performs mantissa addition or subtraction and normalization shift step scanning. An adder is a key module to attain high speed. A carry select adder (CSA) with two-level carry select logic has been adopted. The adder is constructed from 16-b and 12-b CSA units, which are constructed from 4-b Manchester-type adders hierarchically, as shown in Fig. 4(a). The speed of carry select logic circuit greatly influences the propagation of the carry. Thus, p-channel load circuit was adopted for multibit input OR circuits in it (Fig. 4(b)). This circuit operates faster than pure CMOS NOR gate because serially connected p-channel transistors are avoided and gate capacitance is reduced by half.

The third stage performs either rounding or normalization because these operations are not required at the same time. The third stage takes about half a clock cycle to execute and it allows bypass in the same cycle.

B. MDU (Multiply/Divide Unit)

The multiplier and the divider have independent mantissa data paths for the first and second stages while the mantissa data path for the third stage and the exponent data path are shared in common. Hence, the MDU can start either a multiply or a divide operation in each cycle. The autonomy of the multiplier and the divider allows them to run in parallel. Multiplication can start and complete even while the divider is in operation. These instructions generally do not conflict for resources until the third stage. When a conflict happens at the third stage, the control unit determines priority as follows: division then multiplication, because division is a much longer operation and has always started before multiplication.

Fig. 5(a) shows the mantissa block diagram of the multiplier. The first stage performs partial-product generation and carry-save tournament addition of the mantissa and approximate exponent computing. It uses a modified Booth algorithm [10] and a Wallace tree [11] with 4–2 compressors. This stage is most critical speed path of the multiplier. Thus, an n-channel pass transistor was used for XOR circuit of the 4–2 compressor in order to shorten the propagation delay [12]. The Wallace's tree employs a 55×55-b (hidden bit: 1 b, fraction bit: 52 b, Booth sign bit: 1 b, shift bit: 1 b) double-precision full array. This array is laid out as small as 3.2×3.7 mm^2 by using a third metal for long connections between the 4–2 compressor blocks. Fig. 5(b) shows the layout image of the array.

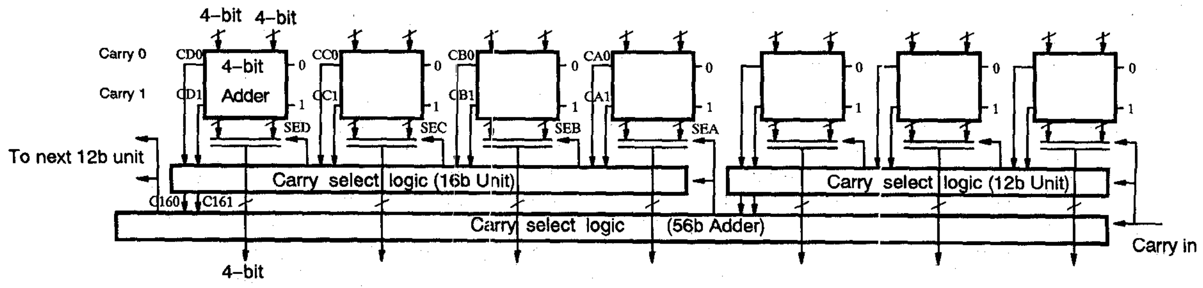
The second stage performs the final carry propagate addition and normalization of the mantissa. The same type CSA which is used in the ALU was used for the final adder and it achieves 7.5 ns for 112-b addition. The third stage performs rounding including bypassing.

The divider's mantissa block diagram of the first stage is illustrated in Fig. 6. The divider uses a radix-4 SRT-division algorithm [13], [14] employing the five symbols, 2, 1, 0, -1, and -2. It computes a 4-b quotient in each cycle by using a two-level cascaded carry-save adder for computation of the next partial remainder in the same cycle, yielding a significant speedup. Each cycle sees two iterations performed. In this method, it is difficult to select the next quotient digit maintaining partial remainders in a carry-save form. The divider checks only the 9 most significant bits of the partial remainder and the divisor in order to produce the next quotient digit. The two 9-b pieces of the partial remainder are reduced from carry-save form into a single binary absolute number, using a 9-b carry-propagate adder. The next quotient digit is selected by comparing it with the half divisor or one and half times the divisor. In this divider, the two mantissa carry-save adders and the quotient selection logic perform in a parallel and pipelined manner. While the mantissa carry-save adder computes the current partial remainder according to the current quotient digit, the quotient selection logic determines the next quotient digit in each half cycle.

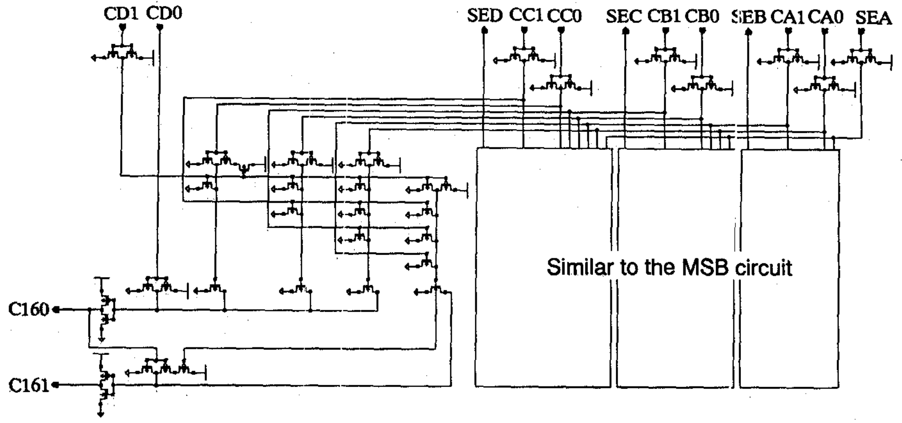
The first stage performs these iterations, taking 14 cycles for double precision and 7 cycles for single precision. The second stage performs carry propagate addition of the full mantissa's partial remainder and the result, taking two cycles. The third stage performs rounding, using the same data path of the multiplier. Thus, the divider requires a total of 17 cycles for the double-precision divide and 10 cycles for single-precision divide.

C. Register File

The register file consists of two 32×32-b register files, as shown in Fig. 7. They have four write and six read independent ports. Up to six readouts from the same address or different addresses and up to four write-ins into different addresses are accomplished in the same cycle. This structure allows two 64-b data loads and stores through the bus, four 64-b operand readouts for the ALU and the MDU, and two result write-ins from the ALU and the MDU, all in parallel, in each cycle.

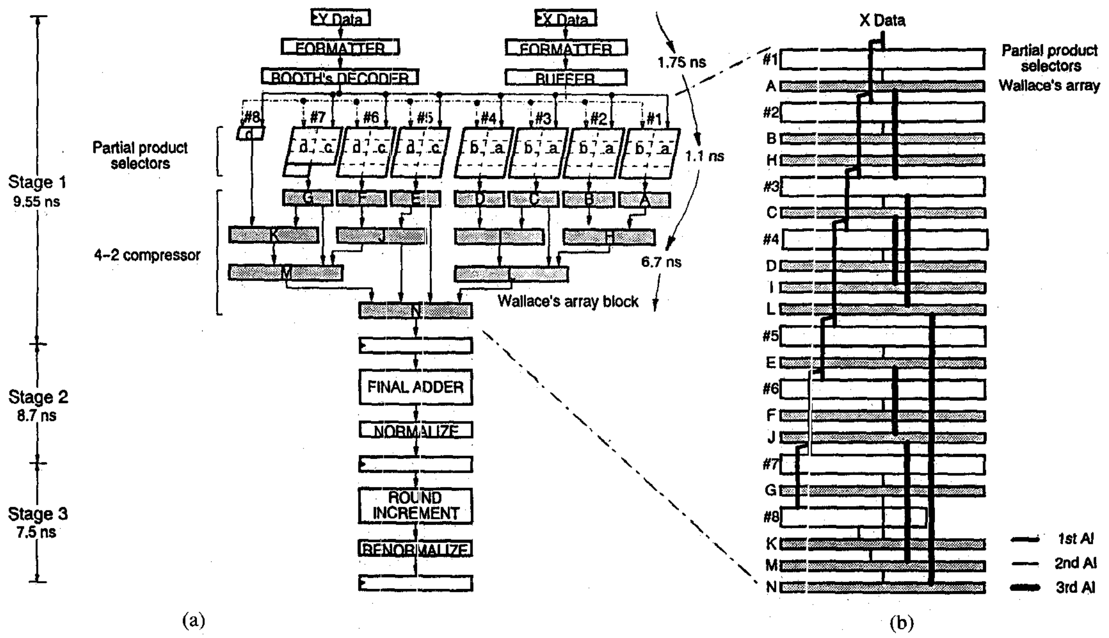

4-bit 4-bit
Carry 0
Carry 1
4-bit Adder
To next 12b unit
Carry select logic (16b Unit)
Carry select logic (12b Unit)
Carry select logic (56b Adder)
Carry in
4-bit

(a)


CD1 CD0
SED CC1 CC0 SEC CB1 CB0 SEB CA1 CA0 SEA
Similar to the MSB circuit
C160
C161

(b)

Fig. 4. 56-b adder. (a) Adder block diagram. (b) Carry select logic for 16-b adder unit.



(a) (b)

Fig. 5. MUL block diagram. (a) MUL mantissa block diagram. (b) Layout image of Wallace's array.

Fig. 8 describes a sample timing chart for the start-up sequence for a double-precision Livermore loop $Q = Q + Z[k] * X[k]$, demonstrating the performance for peak rating.

IV. Bypassing

Latency of the FPU operation heavily affects the performance of most floating-point intensive programs such as the Livermore loop, because an operation needs the result from the previous operation in many cases. There are two ways to reduce latency. One is to reduce the number of stages for FPU operations and the other is to exploit the overlap between succeeding operations. Bypass is a technique of the latter category and can reduce effective latency. This technique can improve the performance of wide range of floating-point intensive programs, including those that are hard to vectorize.

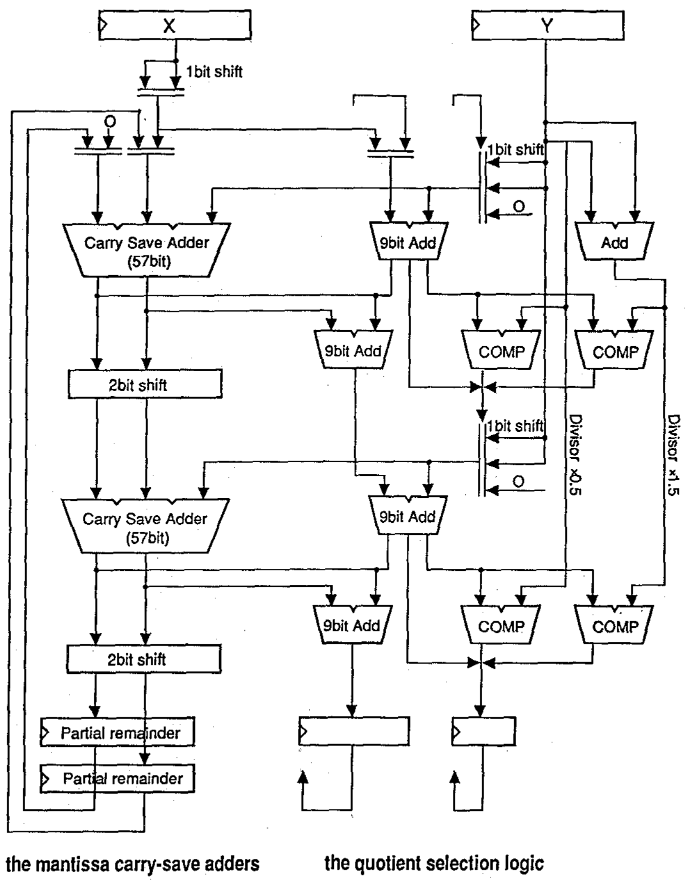


Fig. 6. DIV block diagram.

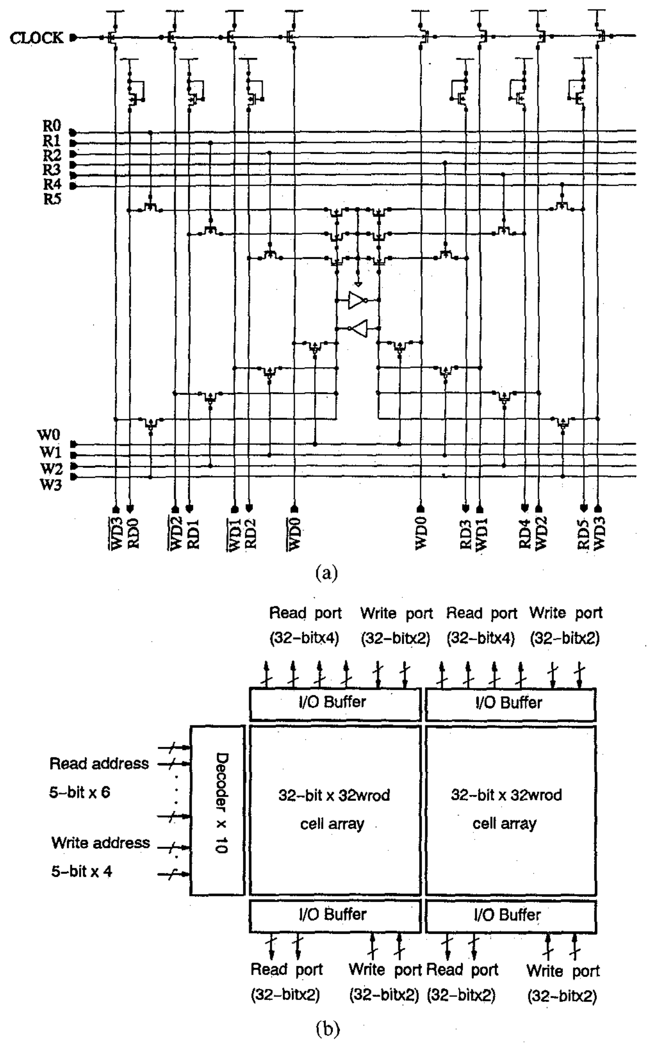


Fig. 7. Ten-port register file. (a) One-bit cell of ten-port RAM. (b) Register-file block diagram.

The ALU, the MDU, and the register file are mutually connected by six independent bypass paths, as shown in Fig. 1. Paths #1 and #2 route data on the bus. Paths #3 and #4 route data to the ALU, and paths #5 and #6 route data to the MDU. The results of addition, subtraction, conversion, multiplication, and division can be by passed from the third stage and the write-back stage of the execution pipeline. The other operations' results, except comparison, can be obtained from the second, the third, and the write-back stage. The path structure is symmetrical and makes it possible to route data between any execution unit without waiting for a write-in into the register file, including the load or store operation. The latency of each operation is described in Table I. Fig. 8 describes an example. In this case, four instructions, floating-point multiplication, addition, and two load operations run in parallel, and there are two bypass. The loading data from the bus are given to the MDU as sources, and written into the register file. The results of the MDU are given to the ALU and written into the register file.

The control unit holds the destination register number of each instruction, and selects the bypassed data by comparing them with the source register number of the current instruction. If multiple instructions, whose destinations are the same, are in operation in the same pipeline, the control unit determines priority in descending order. They are from the second stage, from the third stage, then from the write-back stage, because the latest data have to be routed. The control unit also holds the ordering number which shows the logical order of instruction issue in the same cycle. The control unit selects the bypassed data, by comparing ordering numbers, when multiple instructions, that have the same destination, are in operation in a different pipeline at the same stage.

The fully bypass requires a wide 64-b $\times$ 6 bandwidth, and have to be wired in a short length. The third-layer metal lines, which are laid out over the active region of the ALU and the register file, fulfill this requirement with little layout area overhead, while keeping the wiring minimum. As a result, the signal delay is reduced and makes it possible to take the results and route them from the second or third stage of the pipeline.

V. Twin Single-Precision Computation

The ALU and the MDU can execute operations in each cycle not only upon a set of data, but also upon two sets of single-precision data concurrently, by using the original twin single format, as shown in Fig. 9(a). This format consists of two concatenated IEEE single-precision data as shown in Fig. 9(b). This computation mode doubles the single-precision peak performance and 320 MFLOPS is achieved with an 80-MHz clock. The most frequently used operations such as addition,

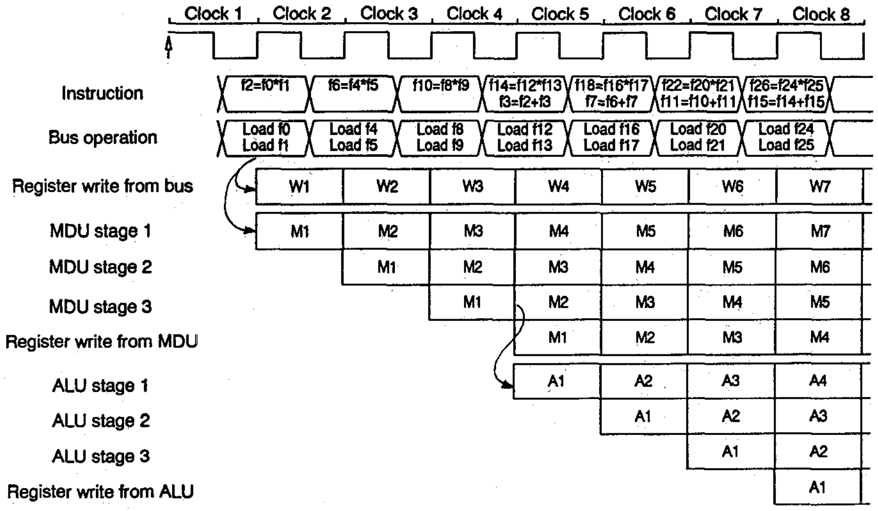


Fig. 8. Example of start-up sequence double-precision Livermore $Q = Q + Z[k] * X[k]$.

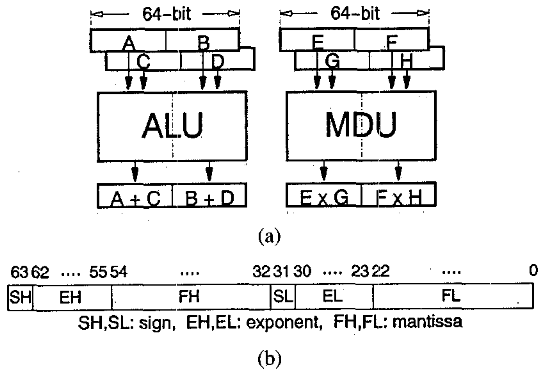


Fig. 9. Twin single-precision computation. (a) Examples of twin single mode. (b) Twin single format.

subtraction, multiplication, division, absolution, and negation are supported.

This execution mode is implemented for real-time applications such as digital signal processing. These applications need much greater execution power because of their simple large-class computations. On the other hand, high precision is not so important. In many cases, single precision is sufficient to meet the requirements.

One way to achieve higher performance is to integrate more multiple execution units in the FPU. However, this is prohibitive because of the large amount of space. In this FPU, the higher bit field of the double-precision mantissa data paths, which are not used in single-precision operations, are used effectively to accomplish the requirements. The mantissa data paths of the ALU and the MDU are divided into two parts and configured as two independent single-precision data paths in this execution mode. These data paths can be controlled individually. Since the basic modules of the ALU and the divider are constructed in a bit-slice fashion or a simple tree structure, they can be efficiently divided into two parts using little hardware, such as switches, which connect the two parts. Similarly, the multiplier which adopts a two-dimensional array structure can be divided cleverly. In the multiplier, the outputs from the a, b, c, and d blocks of the partial-product selectors in Fig. 5 are used for double-precision operations. The outputs from the a block are used for single-precision operations, and the outputs from the a and d blocks are used for twin single-precision operations.

However, two sets of exponent data path, one double-precision wide and the other single-precision wide, are employed because they are much smaller than the mantissa data paths.

The layout overhead for the implementation of twin single-precision computations is kept as low as 4% of the whole FPU, because the mantissa data paths that occupy almost the whole area of the FPU are efficiently divided.

VI. Exception Prediction and Handling

The IEEE standard assumes that when an exception trap occurs, it is possible to identify the instruction that triggers the trap. This FPU permits out-of-order completion if the latency of the succeeding operation is shorter than that of the leading one. Furthermore, an operation of the core processor (IU: integer processing unit) can be completed before an FPU operation. For this architecture, problems arise when an exception trap occurs. The problem is illustrated by the following program fragment and Fig. 10(a):

```
fdiv fr1, fr2, fr3   /* fr3 = fr1 / fr2... D1 */
fmul fr4, fr5, fr1   /* fr1 = fr4 * fr5... M2 */
```

In this example, all FPU instructions except fdiv have six stages and an IU instruction has five stages. The F stage performs the instruction fetch. The D stage performs decode, cache access, and the operand readout. The E stages are arithmetic execution stages. The M stage performs memory access, and the W stage performs write-back to the register.

When the exception is detected in the final stage at clock 13 in Fig. 10(a), the processor tries to recover and to restart from $D1$ (fdiv). However, it is impossible because $M2$ (fmul) has completed and the content of register fr1 has been changed. This problem arises between FPU instructions which start at the same cycle in parallel because of a two scalar architecture.

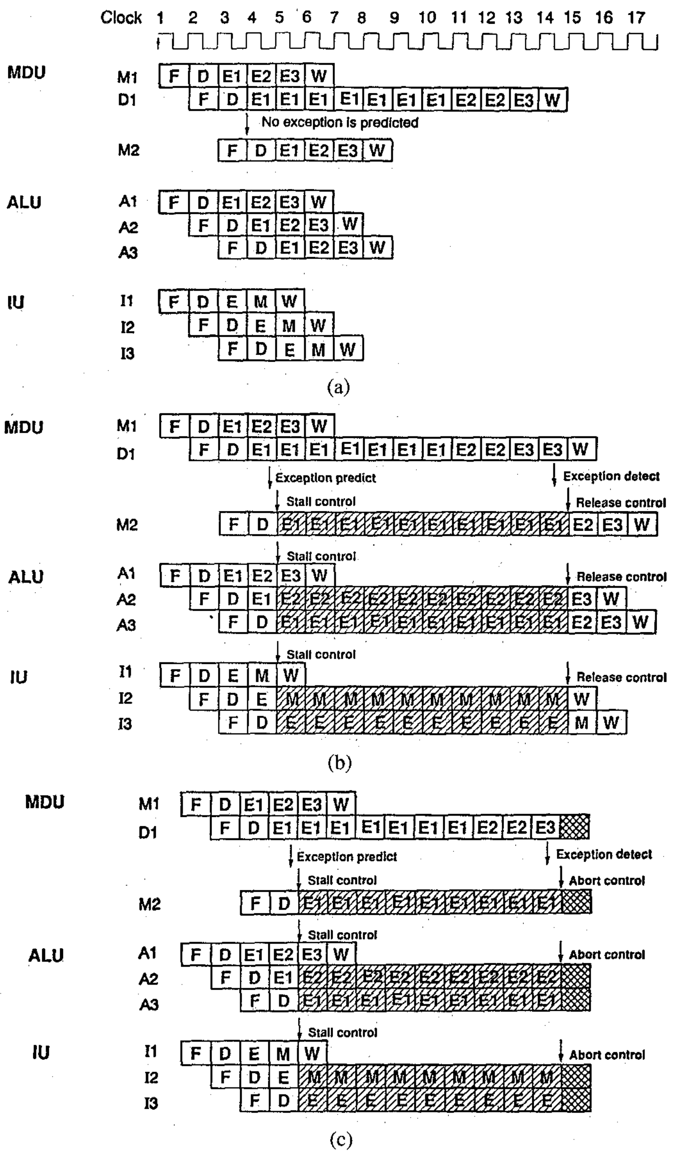


Fig. 10. Example of exception prediction control. (a) No exception is predicted. (b) An exception is predicted but does not occur. (c) An exception is predicted and occurs.

The state of the FPU must be held for the software handling of exceptions that occur at the last stage of the pipeline. This type of exception includes overflow exception and underflow exception.

There are a few approaches [15] to avoid the problem. For example, the SPARC chip adopts an instruction queue called "floating-point deferred-trap queue" [16]. This method permits out-of-order completion between the FPU and IU instructions. However, the FPU instructions have to complete in-order with each other. Another example is a history buffer that is implemented in the Motorola 88110 [17], [18]. In the superscalar, the control scheme for this method tends to be complex, and hardware tends to be large, because it requires a large bit-width stack for the history buffer and additional ports as large as the scalar on the register file. MIPS R2010, R3010, and R4000 have adopted an exception prediction and a stall mechanism [1], [19]. However, the MIPS's mechanism does not support a superscalar architecture.

TABLE III
COMPARISON CONSTANTS

Exception type	Operation	Precision	
		Double	Single
Underflow	Addition, Subtraction	53, 1	24, 1
	Multiplication	1	1
	Division	1	1
	Convert to single-precision	0	0
Overflow	Addition, Subtraction	2046	254
	Multiplication	2046	254
	Division	2046	254
	Convert to single-precision	2046	254
Convert overflow	Convert to 32-bit integer	21	21

This FPU adopts an exception prediction and a stall mechanism which is improved for the superscalar architecture because this technique can be implemented simply with small hardware and is well adapted to RISC approach. This achieves precise and fast software exception handling even in an out-of-order completion case. The FPU predicts overflow exception, underflow exception (the IEEE-754 standard defined tininess), and conversion overflow from the floating-point number to a 32-b integer, when the trap mode of these exceptions is enabled. These exceptions are quickly predicted at the first stage and then the FPU outputs a stall request signal to the core processor. The core processor is expected to receive it and stall the whole system until the FPU pipeline is released.

Fig. 10 shows an example case in which three kinds of instructions, MDU ($M1, D1, M2$), ALU ($A1$–$A3$), and IU ($I1$–$I3$), start in the same cycle and are executed in parallel. The prediction unit at the first stage categorizes the operations into two cases. In the first case (Fig. 10(a)), one of the present operations ($D1$) will never cause an exception. So, an out-of-order completion is allowed. In the second case (Fig. 10(b), (c)), an exception may occur later but cannot be determined until the last stage at clock 13, $D1$, which causes the prediction continues. On the other hand, the pipeline executions of the other operations ($M2, A2, A3$), including the IU operations ($I2, I3$), which start at the same clock cycle or after $D1$ has started, are frozen in order to hold the state of the FPU for exception handling. In case an exception does not occur at clock 13 (Fig. 10(b)), the frozen pipelines are released and continued again. Otherwise, all the pipeline executions are aborted (Fig. 10(c)), and the software handling routine is invoked to recover and restart.

Exceptions are predicted at the first stage by comparing the approximate exponent before normalization with the constants that are determined by the maximum shift steps of normalization and renormalization, as shown in Table III. The key to making this work is how rare the false positives are.

The MDU predicts overflow and underflow. The maximum shift step in the normalization and renormalization phase of the MDU is 1 or 0. Thus, the prediction unit decides that overflow may occur if and only if the approximate exponent is larger than or equal to 2046 for double precision or 254 for single precision, the maximum exponent of normalized number. Underflow is predicted if and only if the approximate exponent is less than or equal to 1, the minimum exponent. The

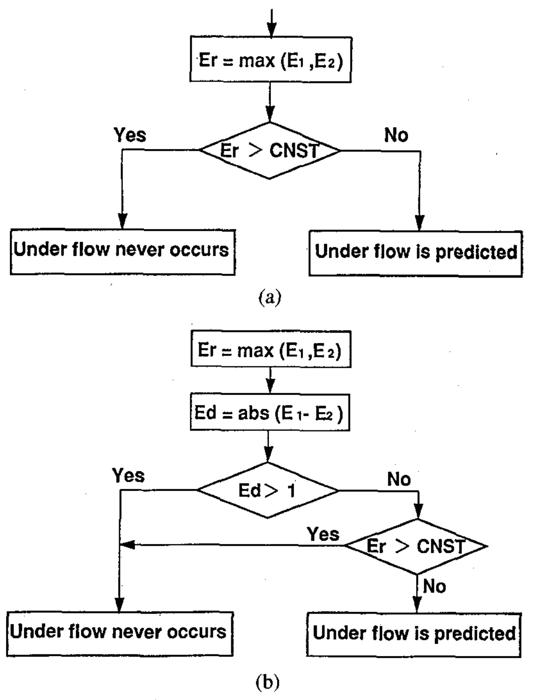


Fig. 11. Underflow prediction method. (a) Conventional prediction method. (b) New prediction method.

ALU predicts overflow, underflow, and conversion overflow. Overflow is predicted by the same way as the MDU.

The ALU incorporates a new underflow prediction method for addition and subtraction. This method performs comparisons twice for the prediction, whereas the conventional method employs a plain constant comparison. The conventional method compares the approximate exponent Er with 53 for double precision or with 24 for single precision as shown in Fig. 11(a), and it determines that an underflow will never occur if Er is larger than the constants. This method is very simple, but the prediction is conservative. The new method compares the difference Ed in the two exponents with 1 at first, as shown in Fig. 11(b). Then, it compares Er with the constant 53 or 24 only when Ed is 1 or 0. Otherwise, it determines that no underflow exception occurs later. The new approach is based on the fact that a normalization shift of more than 2 b occurs only if the difference between the two exponents is 1 or 0. A normalization shift is no more than one when the difference in the two exponents is larger than one. This method makes prediction more precise than the conventional method. As a result, the possibility of pipeline freezing decreases, and a much more effective throughput is guaranteed.

The prediction unit is simply implemented with comparators and constant generating circuits. For example, this only occupies as little as 1% of the whole ALU.

All exceptions are handled by hardware when the exception trap mode is in the disable mode. The exception handling unit outputs default constant numbers such as QNaN, infinity, and maximum, according to the IEEE-754 standard. The default constant number is prepared according to a predicted exception by the prediction unit before it is detected at the last stage. If an exception actually occurs, the constant is selected as the result instead of the execution pipeline output. Otherwise, the ordinary pipeline output is chosen as the result. Thus, effective throughput is guaranteed because extra cycles for exception handling are not necessary.

All exceptions are reported into a status register. It has six exception flags (V: invalid, Z: division by zero, O: overflow, U: underflow, I: inexact, X: unimplemented by hardware) and an exception sticky flag. These flags hold the latest status of the FPU as if the instructions run sequentially. However, there are two problems to reporting exceptions in this way under a superscalar architecture. The first problem is that it is necessary to cancel the status of an outstripped instruction when an instruction is completed out of order. For example, $D1$'s status must not be written into the flags because it is outstripped by $A2, A3$, and $M2$, as shown in Fig. 10(a). The second problem is that multiple write requests into the same flag field may happen at the same time because of the superscalar architecture. Another method, holding the status of the ALU and the MDU in different registers, is simpler. However, this method is difficult to keep the compatibility with conventional processors, and this is very serious.

This FPU uses an ordering check flag, which shows whether an instruction is outstripped by succeeding instructions or not, to avoid the former problem. Each instruction has this flag and it is set at the second execution cycle if the instruction is outstripped. There are two cases. The first case is when the order of the instructions is changed by a pipeline stall causing exception prediction. The second case is when floating-point division is outstripped. If an ordering check flag of an instruction is set, its status is written only into the exception sticky flag field without the exception flag field, because this instruction is outstripped. Otherwise, the status is written into all fields of the register. In the example above, $D1$'s ordering check flag is set at clock 4. Then its status is not written into the register except the sticky field at clock 14.

Each instruction also has an ordering number which shows the order of issue in the same cycle to solve the second problem. When a write conflict to the exception register occurs, the status to be held is selected by comparing the ordering numbers as follows. If no exception trap occurs at the current cycle, the status of the instruction that has the largest ordering number is written into the register, because this instruction is the least one of the sequence. Otherwise, if an exception trap occurs, the status of the instruction that has the smallest ordering number and causes the trap is written, because this instruction is the first one that causes the exception trap.

This way of reporting exceptions can keep compatibility with conventional processors and helps precise exception handling.

VII. Design Methodology

A behavioral model of a processor including the FPU is written in the C language because it runs fast and is easy to debug. An in-house developed hierarchical hardware description language (H^2DL) [20] is used for the register-transfer-level model and an in-house mixed-level simulator FAL [21], [22] is used for simulations. The control logic and the data path are divided clearly in this model. An in-house

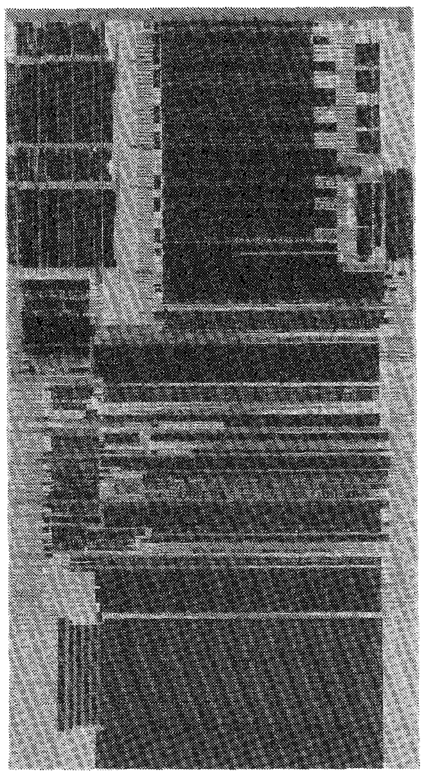


Fig. 12. Layout of the FPU.

transistor-level timing simulator RSIM-FX [23] was used to check out the critical speed paths and SPICE was used for detail circuit simulation on the paths, before and after layout. Logic design was verified by comparing FAL results against RSIM-FX results. The transistor-level model was designed by using CADENCE's schematic design entry. The test vector of "a compact test suite for P754 arithmetic" [24] is used to test its conformance with the IEEE 754 floating-point standard.

A labor-intensive, hand-optimized design methodology is chosen for the data-path layout in order to achieve the cycle speed and a high density. CADENCE's physical design entry is used to lay out task. Net lists of the critical speed paths that were extracted from the physical layout database with parasitic resistance and capacitance were used for intensive SPICE simulations. An in-house logic synthesizing tool was used for the control logic design in order to make modification easier, followed by logic and speed optimization using SYNOPSYS.

VIII. Conclusion

A 80-MHz pipelined floating-point processing unit (FPU) for superscalar processors has been described with a die size of 61 mm^2, using a 0.5-μm CMOS triple-metal layer technology. A die photograph is shown in Fig. 12. The FPU has two execution modes to meet precise scientific computations and the real-time applications. It can start two FPU operations in each cycle, and this achieves a peak performance of 160 MFLOPS double or single precision with an 80-MHz clock. Its twin single-precision computation delivers 320 MFLOPS single precision. It also allows an out-of-order completion. Its full bypass and precise exception prediction using a new method make it possible to achieve a large amount of effective throughput. The FPU will be integrated on a high-performance superscalar processor.

Acknowledgment

Advances in FPU design do not come without enormous efforts expended by many people. H. Takano helped to design the divider from RTL through physical layout. T. Sato tested its conformance with the IEEE 754 floating-point standard. The authors wish to thank H. Tago for his advice and encouragement throughout the work.

References

[1] C. Rowen, M. Johnson, and P. Ries, "The MIPS R3010 floating-point coprocessor," *IEEE Micro*, pp. 53–62, June 1988.

[2] M. Birman *et al.*, "Design of a high-speed arithmetic datapath," in *Proc. ICCD*, Oct. 1988, pp. 214–216.

[3] B. J. Benschneider *et al.*, "A 50 MHz uniformly pipelined 64 b floating-point arithmetic processor," in *ISSCC Dig. Tech. Papers*, Feb. 1989, pp. 50–51.

[4] T. Nakayama, S. Kojima, H. Igarashi, K. Tamada, and T. Toba, "An 80 b, 6.7 MFLOPS floating-point processor with vector/matrix instructions," in *ISSCC Dig. Tech. Papers*, Feb. 1989, pp. 52–53.

[5] L. Kohn and S. W. Fu, "A 1 000 000 transistor microprocessor," in *ISSCC Dig. Tech. Papers*, Feb. 1989, pp. 54–55.

[6] R. K. Montoye, P. W. Cook, E. Hokenek, and R. P. Havreluk, "An 18 ns 56-b multiply-adder circuit," in *ISSCC Dig. Tech. Papers*, Feb. 1990, pp. 46–47.

[7] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC system/6000 floating-point execution unit," *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[8] P. Chai *et al.*, "A 120 MFLOPS CMOS floating-point processor," in *Proc. CICC*, May 1991, pp. 15.1.1–15.1.4.

[9] *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.* Los Alamitos, CA: IEEE Computer Society, 1985.

[10] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, part 2, 1951.

[11] C. S. Wallace, "A suggestion for fast multipliers," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14–17, Feb. 1964.

[12] J. Mori *et al.*, "A 10-ns 54 × 54-b parallel structured full array multiplier with 0.5-μm CMOS technology," *IEEE J. Solid-State Circuits*, vol. 26, no. 4, pp. 600–606, Apr. 1991.

[13] J. E. Robertson, "A new class of digital division methods," *IEEE Trans. Comput.*, vol. C-7, pp. 218–222, Sept. 1958.

[14] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. Comput.*, vol. C-17, no. 10, pp. 925–934, Oct. 1968.

[15] M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[16] *The SPARC Architecture Manual Ver. 8*, SPARC International, Inc., Menlo Park, CA, 1991.

[17] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 562–573, May 1988.

[18] K. Diefendorff and M. Allen, "Organization of the Motorola 88110: A superscalar RISC microprocessor," in *Proc. Micro Architecture Symp. Japan*, Nov. 1991, pp. 77–87.

[19] J. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach.* Palo Alto, CA: Morgan Kaufmann, 1990.

[20] M. Miyata, S. Nishio, and I. Yamazaki, "A hierarchical hardware description language," in *Proc. Electronic Design Automation Conf.*, 1984, pp. 189–193.

[21] M. Sekine, "An advanced design system: Design capture, functional test generation, mixed level simulator and logic synthesis," in *Proc. CICC*, May 1989, pp. 19.4.1–19.4.6

[22] M. Aihara and M. Sekine, "A mixed level simulator mega-FAL with novel data structure oriented to HDL statements," in *Proc. CICC*, May 1990, pp. 10.3.1–10.3.4.

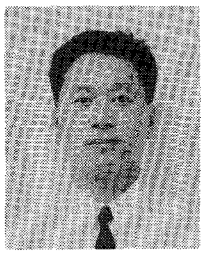
[23] N. Kimura and J. Tsujimoto, "Calculation of total dynamic current of LSI using a switch level timing simulator (RSIM-FX)," in *Proc. CICC*, May 1991, pp. 8.3.1–8.3.4.

[24] "A compact test suite for P754 arithmetic—ver. 2.0," Computer Sci. Div., Univ. of California, Berkeley, 1983.

Nobuhiro Ide was born in Tokyo, Japan, on July 20, 1963. He received the B.S. and M.S. degrees in electrical engineering from Waseda University, Tokyo, Japan, in 1986 and 1988, respectively.

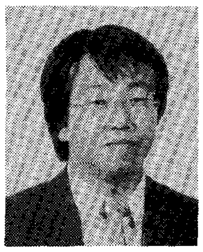
In 1988 he joined the ULSI Research Laboratories of Toshiba Corporation, Kawasaki, Japan, where he was engaged in microprocessor development. His current research interests include FPU architecture, microprocessor architecture, VLSI design methodology, and parallel processing.

Mr. Ide is a member of the Institute of Electronics, Information and Communication Engineers of Japan.



Hiroto Fukuhisa received the B.S. and M.S. degrees in electrical engineering from Chiba University, Chiba, Japan, in 1988 and 1990, respectively.

In 1990 he joined ULSI Research Laboratories of Toshiba Corporation, Kawasaki, Japan. His current research interests include microprocessor architecture and VLSI design methodology. Mr. Fukuhisa is a member of the Information Processing Society of Japan.



Yoshihisa Kondo received the B.S. and M.S. degrees in electrical engineering from the University of Electro-Communication, Tokyo, Japan, in 1982 and 1984, respectively.

In 1984 he joined the ULSI Research Laboratories of Toshiba Corporation, Kawasaki, Japan, where he was engaged in microprocessor development. His current research interests include FPU architecture, microprocessor architecture, VLSI design methodology, and high-speed logic/circuit technology.

Mr. Kondo is a member of the Institute of Electronics, Information and Communication Engineers of Japan.

Takeshi Yoshida received the B.E. and M.E. degrees from the University of Tsukuba in 1987 and 1989, respectively.

In 1989 he joined the ULSI Research Laboratories, Toshiba Corporation, Kawasaki, Japan, where he has been engaged in the development and research of microcomputer systems. His current research interests include microprocessor architecture, parallel processing, and VLSI design methodology.

Mr. Yoshida is a member of the Information Processing Society of Japan.

Masato Nagamatsu was born in 1959 in Oita city, Oita prefecture, Japan. He received the B.S. and M.S. degrees in electrical engineering from Tokyo University, Tokyo, Japan, in 1982 and 1984, respectively.

In 1985 he joined the Semiconductor Device Engineering Laboratory of Toshiba Corporation, Kawasaki, Japan, where he was engaged in microprocessor and peripheral VLSI development. His current research interests include RISC microprocessor design, microprocessor architecture, VLSI design methodology, and parallel processing.

Mr. Nagamatsu is a member of the Institute of Electronics Information and Communication Engineers of Japan.

Junji Mori was born in Aichi Prefecture, Japan, on March 9, 1965. He graduated from Higashiyama Engineering High School, Nagoya, Japan, and from Toshiba Computer School, Kawasaki, Japan in 1983 and 1984, respectively.

In 1984 he worked for Toshiba Computer School as an instructor, and in 1985 he joined the Semiconductor Device Engineering Laboratory, Toshiba Corporation, Kawasaki, Japan. He has been engaged in the research and development of CMOS graphics processor and high-performance logic VLSI including microprocessors.

Itaru Yamazaki graduated from Yawatahama Technical High School, Ehime, Japan, and from Toshiba Computer School, Kawasaki, Japan, in 1987 and 1988, respectively.

In 1988 he worked for Toshiba Computer School as an instructor, and in 1989 he joined the Semiconductor Device Engineering Laboratory, Toshiba Corporation, Kawasaki, Japan. He has been engaged in the research and development of high-performance logic VLSI including microprocessors.

Kiyoji Ueno received the B.S. degree in information engineering from Tohoku University, Miyagi, Japan, in 1988.

In 1988 he joined the Semiconductor Device Engineering Laboratory, Toshiba Corporation, Kawasaki, Japan. He has been engaged in the design of CMOS logic VLSI.