A single-chip IBM System/390 floating-point processor in CMOS

by S. Dao-Trong
K. Helwig

A floating-point processor with the IBM System/390® architecture is implemented in one CMOS VLSI chip containing over 70 000 cells (equivalent inverters), using a transistor channel length of 0.5 μm. All floating-point instructions are hard-wired, including the binary integer multiplications. The chip is implemented in a 1-μm technology with three layers of metal. All circuits are realized in standard cells except for a floating-point register and a multiplier array macro, which are custom designed to save chip area. Instructions are performed in a five-stage pipeline with a maximum operating frequency of 37 MHz. The chip measures 12.7 mm × 12.7 mm, and dissipates 2 W. It is part of the chip set which forms the core of the IBM Enterprise System/9000™ Type 9221 entry-level models.

Introduction

Floating-point processors used to be seen as an option which could be added to the main CPU for performing scientific applications. In the models of the entry-level type (designated 9221) of the new IBM Enterprise System/9000™ (ES/9000™) line, floating-point processing is becoming an inherent part of the CPU. This paper describes the IBM ES/9000 Type 9221 floating-point processor, which is tightly coupled to the CPU and carries out all IBM System/390® floating-point instructions. All instructions are hardware-coded, so no microinstructions are needed. Moreover, binary integer multiplication is also implemented on the floating-point unit to improve overall performance.

The floating-point processor was implemented in 1-μm CMOS technology, using the standard cell approach, with only two custom-designed macros—the 60 × 58-bit multiplier macro and the floating-point register macro. The data flow design was done in a top-down manner, starting with an abstract functional block description which was broken down in more and more detail until the gate implementation level was reached. This method is especially appropriate for a standard cell approach, in which only logic gates from a standard library are available and no detailed knowledge about the transistor level is required. A mixed-level representation was created which was used for understanding, documenting, and most of all, early floor-planning, to make sure that all the logic fit on the chip and that the wiring delays were acceptable.

Design goals

Because of the inherent character of the floating-point processor in the IBM Enterprise System/9000 Type 9221 models, a global optimization of the processor chip set was the ultimate design goal. First, system partitioning with

respect to packaging requirements was done. Our overall goal was to obtain the maximum system performance with the given CMOS process [1]. Wide buses were used to achieve high performance. Critical interconnections were split into an internal processor bus and a floating-point bus. The chip set thus built consisted of two multichip modules, one containing the fixed-point unit, the instruction processing unit, and two cache chips, and the other with only the floating-point processor unit and the clock chip. See [1] for additional details.

Second, local chip optimization was done. For the floating-point unit, the cycle was dictated by the CPU. Optimization was then focused on pipeline structure, chip area, and instruction implementation. Much effort was expended to make sure that the structure would fit on one chip. Although there were many implementation alternatives with more promising cycle times, we chose one which best fit the overall requirements of chip area and cycle time.

Chip structure

• *Pipelining*

While the CPU is based on a four-stage pipeline, the floating-point processor requires a five-stage pipeline to perform its most used instructions (add, subtract, and multiply) in one cycle for double-precision operands. The CPU resolves operand addresses, provides operands from the cache, and handles all exceptions for the floating-point processor. The five stages of the pipeline are instruction fetch, which is executed on the CPU, register fetch, operand prealignment, addition, and normalization and register store.

To preserve synchronization with the CPU, a floating-point wait signal is raised whenever a floating-point instruction needs more than one cycle. The CPU then waits until this wait signal disappears before it increments its program counter and starts the next serial instruction, which is kept on the bus.

Because the IBM System/390 architecture requires that interrupts be precise, a wait condition is also invoked whenever an exception may occur. To minimize the impact on performance, extra logic was incorporated to predetermine the interrupt situations, as described later in the discussion of control flow.

• *Data flow*

The data flow of the IBM ES/9000 Type 9221 floating-point processor was designed in a top-down manner. Starting with global function blocks, we broke down the next hierarchical level into more detailed functions that were finally transformed into gate-level functions, including fine tuning with respect to logic transformation, input assignment, and signal buffering. The last is very critical for CMOS technology.

Figure 1 shows the data flow of the IBM ES/9000 Type 9221 floating-point processor. It has been designed to perform the most used add/subtract and multiply instructions in one cycle for single- and double-precision operands. Many bypass buses are used to avoid wait cycles when the results of the foregoing instructions are used. A wait cycle is needed only if the result of one instruction is used immediately by the next sequential instruction.

The data flow shows two parallel paths for fraction processing: one add-path where all non-multiply instructions are implemented, and one multiply-path especially designed for multiply and divide. The add-path is 60 bits wide and consists of an operand switcher, an aligner, an adder, and a normalizer shifter. Instead of using two aligners on each side of the operand paths, we used a switcher to switch operands, thereby saving one aligner. The switcher is also needed for other instructions, and requires many fewer circuits. The resulting delay is not critical.

The multiply-path consists of a Booth encoder for the 58-bit multiplier, a multiplier macro which forms the 58×60-bit product terms *sum* and *carry*, and a 92-bit adder which delivers the result product. The sign and exponent paths are adjusted to be consistent with the add-path. The exponent path resolves all exception and true zero situations, as defined by the IBM System/390 architecture. The implementation of all other instructions is merged into the add-path and multiply-path, and requires only minimal additional logic. The data flow in Figure 1 thus shows more function blocks and multiplexer stages than needed for only add, subtract, and multiply operations.

The data flow was then partitioned into smaller parts (typically registers with their input control). For every partition, such as FB, we described the functions in bit-level detail using a block description form to reflect all functions required. All interface signals were named appropriately. **Figure 2** shows an example of the detailed documentation for the partition FB. This was used for further transformation into gate-level and high-level documentation of the data flow, and also to design the control flow. We were able to stabilize the interfaces between partitions at a very early stage. Cell count prediction was then easily made, so that early floor-planning could be done effectively. Because of our numerous wide internal buses, the chip layout was based exclusively on wiring studies. (It was not obvious that the data flow would fit on the chip.) Using this high-level documentation, we obtained a chip floor plan at a very early stage. It helped us to adjust nets and bus buffering to achieve a well-balanced design. In fact, it was found that delay problems arise primarily in the long buses. Also, chip floor-planning ensured that we could implement all of

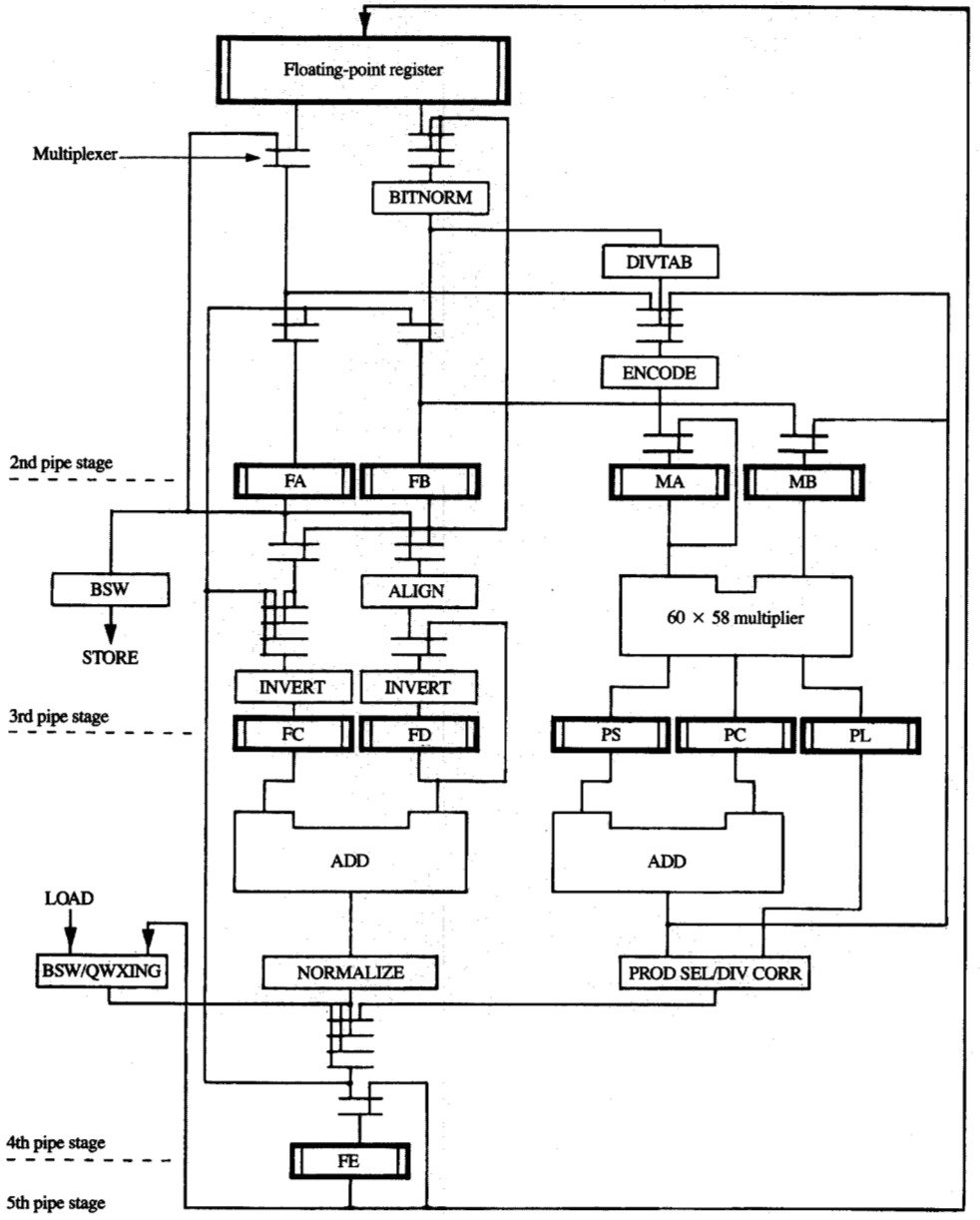


Figure 1

Floating-point data flow.

our logic on a single chip. **Figure 3** shows the chip partition and the cell count prediction for each partition. The main buses are globally wired. The control logic part was assumed to comprise about 20% of the data flow part.

Arithmetic implementation

Floating-point instructions are partitioned into three main groups: 1) addition/subtraction, load; 2) multiplication; and 3) division. These are the instructions most used in

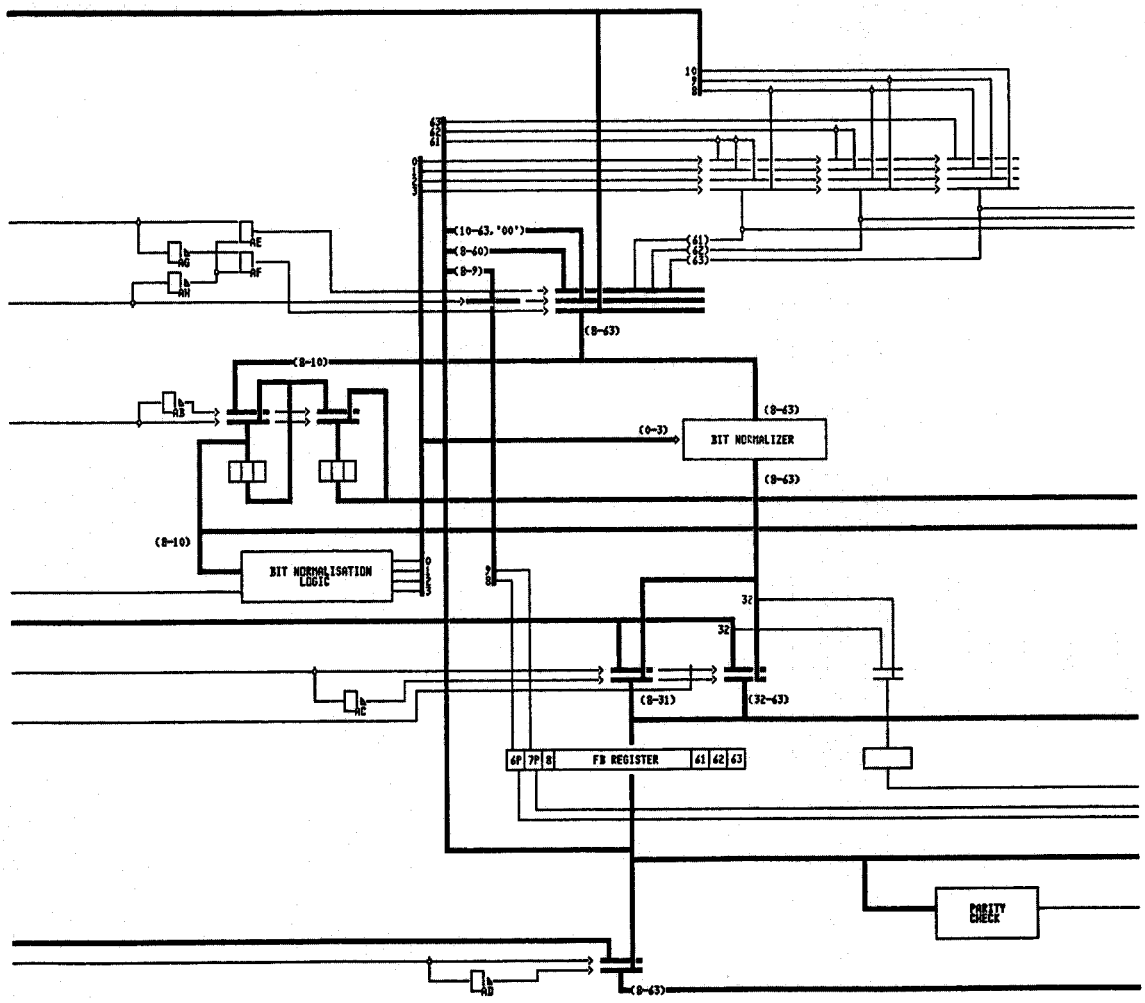


Figure 2

Detailed documentation of partition FB.

scientific applications. The first two groups of instructions are performed in one cycle, and division is made as fast as possible. The following subsections describe the control flow to these three main groups of instructions.

- *Addition, subtraction, load*

During the first two pipelined stages, only instruction and operand fetching are done. All data processing is concentrated in the third and fourth pipelined stages. In the fifth stage, the result is written back to the floating-point register. Hardware implementation is done with minimal logic, provided the cycle remains within the design range.

Effective addition, load

Loads are treated like addition, with one operand equal to zero. During stage 3 the exponents of both operands are compared in order to determine the amount of alignment shift. The operand with the smaller exponent is then passed to the aligner for prealignment. In stage 4 the aligned operands are added. The addition may produce a carry-out, which results in a shift right by one digit position, in accordance with the IBM System/390 architecture. The exponent must then be incremented by one. If the instruction requires postnormalization, leading zeros are detected, and the normalization shift amount is determined. The exponent is then decreased accordingly.

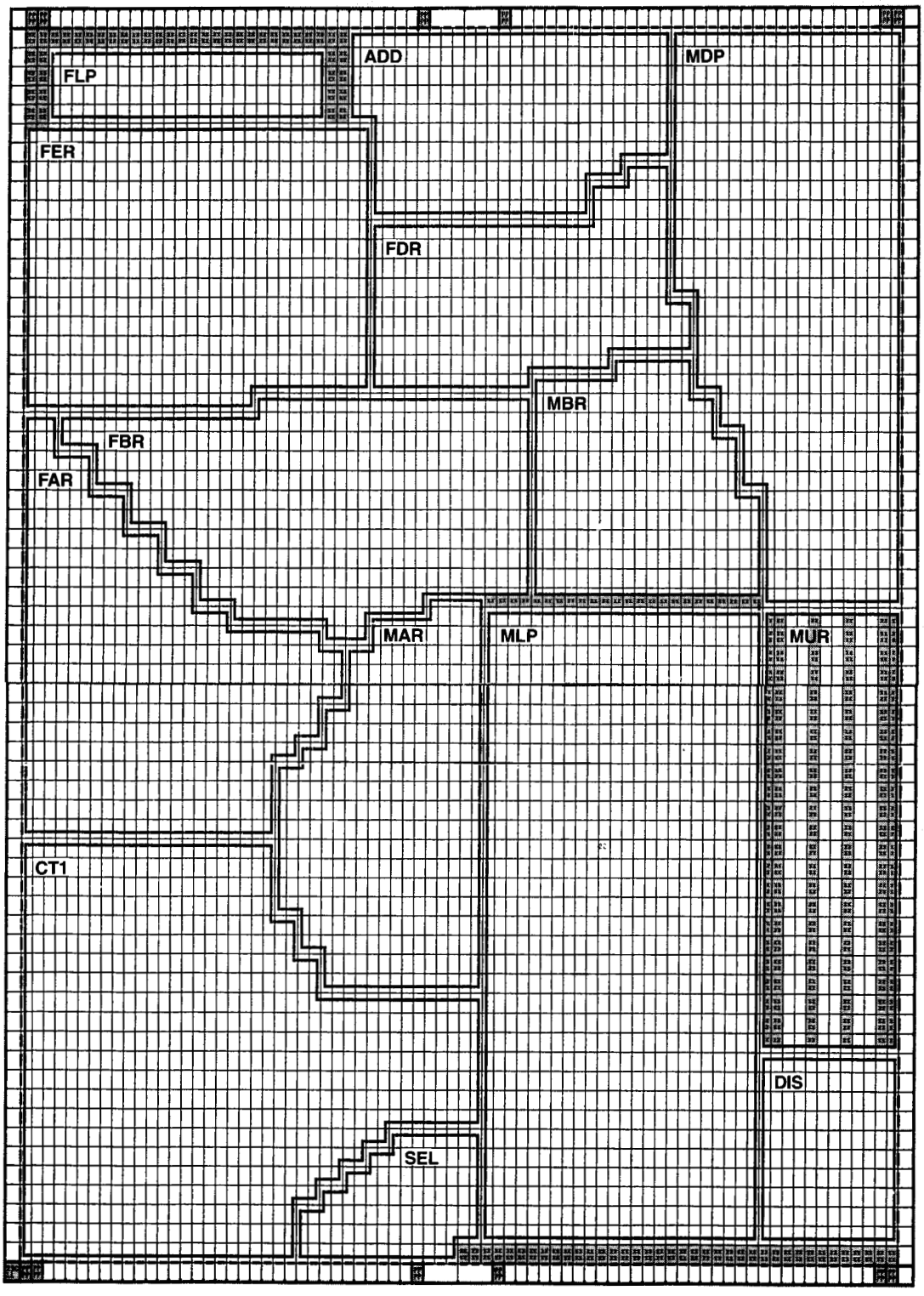


Figure 3

Early floor-planning. Areas PP are reserved pads; areas XX are reserved for wiring. From upper left to lower right, IDs of the areas are as follows: FLPT (floating-point register macro), ADD (adder), MDP (multiplier postprocessing), FER (FE partition), FDR (FD partition), MBR (MB partition), FAR (FA partition), FBR (FB partition), MAR (MA partition), MLP (multiplier macro), MUR (multiplier register), CT1 (control logic block), DIS (oscillator), and SEL (self-test logic).

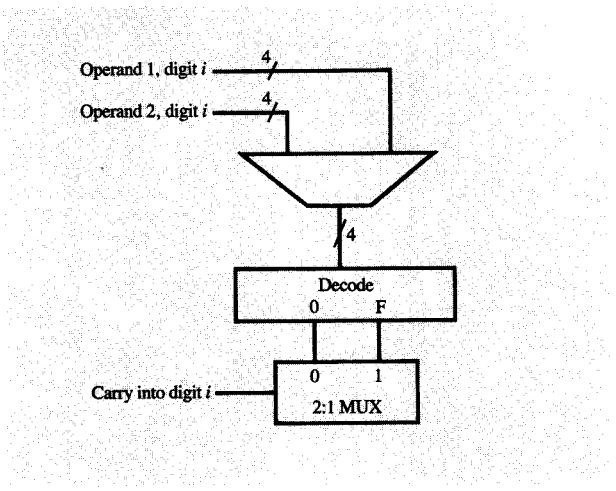


Figure 4

Zero-digit detection.

Table 1 Exception wait situations.

Effective addition/subtraction	*Exponent = 7F*	*Exponent < 0D*	*Unnormalized operand*
Yes	Yes	—	No
Yes	—	Yes	Yes
No	—	Yes	—

Since time is still available in stage 4, the exponent calculation is made sequentially after that of addition, using only one exponent adder with an input multiplexer to select whether an exponent increase, an exponent adjustment, or a multiply/divide exponent is required.

Leading-zero detection is made by calculating the hexadecimal digit sums without a propagated carry-in. Hexadecimal sums 0 and F for the digit position i are determined and fed into a multiplexer. The carry-in to this digit position selects whether or not the result digit is zero. This carry bit comes from the same carry-lookahead circuit used for the adder, so no additional circuit is needed. **Figure 4** shows the realization. By using this additional logic, the shift amount can be determined at nearly the same time as the addition result.

Exponent exception, either overflow or underflow, is also detected in this stage. Meanwhile, the next instruction has already been started. As mentioned earlier, a wait may be raised at stage 3 to hold execution of the next serial instruction. In our case of an effective addition, the wait situation is met when

- The intermediate result exponent is 7F and will overflow when an exponent increment is caused by a carry-out from the adder.
- The intermediate result exponent is smaller than 0D, and a normalization is required for unnormalized operands. Here the exponent must be decreased by the normalization shift amount, which can be at most 0D (14 in the decimal system), thus producing an exponent underflow.

Table 1 shows the cases in which an exponent exception condition is met for the effective addition and subtraction. This represents a very small percentage of cases, so performance is only minimally affected.

Effective subtraction

To avoid recomplementation of the result, we always subtract the smaller operand from the greater one. Most frequently, subtractions are done by means of normalized fractions. Here the greater operand can always be determined whenever the exponents are unequal. In stage 3 the operand with the smaller exponent is passed to the aligner whose output is to be complemented. When the exponents are equal, a fraction-compare circuit compares the operands and selects the smaller operand to be negated. Thus, we have a complemented block on both sides of stage 3 in our data flow. In stage 4 the subtraction is performed, delivering a positive result. The sign is the sign of the greater operand. Normalization is done in the same way as for an effective addition. No exponent overflow can occur.

For the case of subtraction with unnormalized operands, except for the case in which the exponents are equal, we cannot determine which operand is the smaller one. One additional cycle is needed. At first we calculate $A - B$. Depending on whether the result is positive or negative, we calculate in the following cycle $A - B$ or $B - A$, respectively. The result is then taken from the second subtraction. The sign is determined accordingly.

- *Multiplication*

Multiplication is implemented by using a multiplier macro custom-designed to save chip area. Because the overall design goal was not to minimize cycle time but to optimize the global chip set, we used the modified Booth algorithm with serial addition of partial product terms [2], although the Wallace tree method would have been more efficient [3]. The latter method, however, is not appropriate for a regular structure, which in turn is very important when custom design is required and chip area and turnaround time are to be minimized. When this method is used, the multiplication time is proportional to the number of additions of partial products, which is $N/2$, where N is the number of multiplication bits.

The design goal was to increase the arithmetic speed by reducing the number of additions and sequential delay

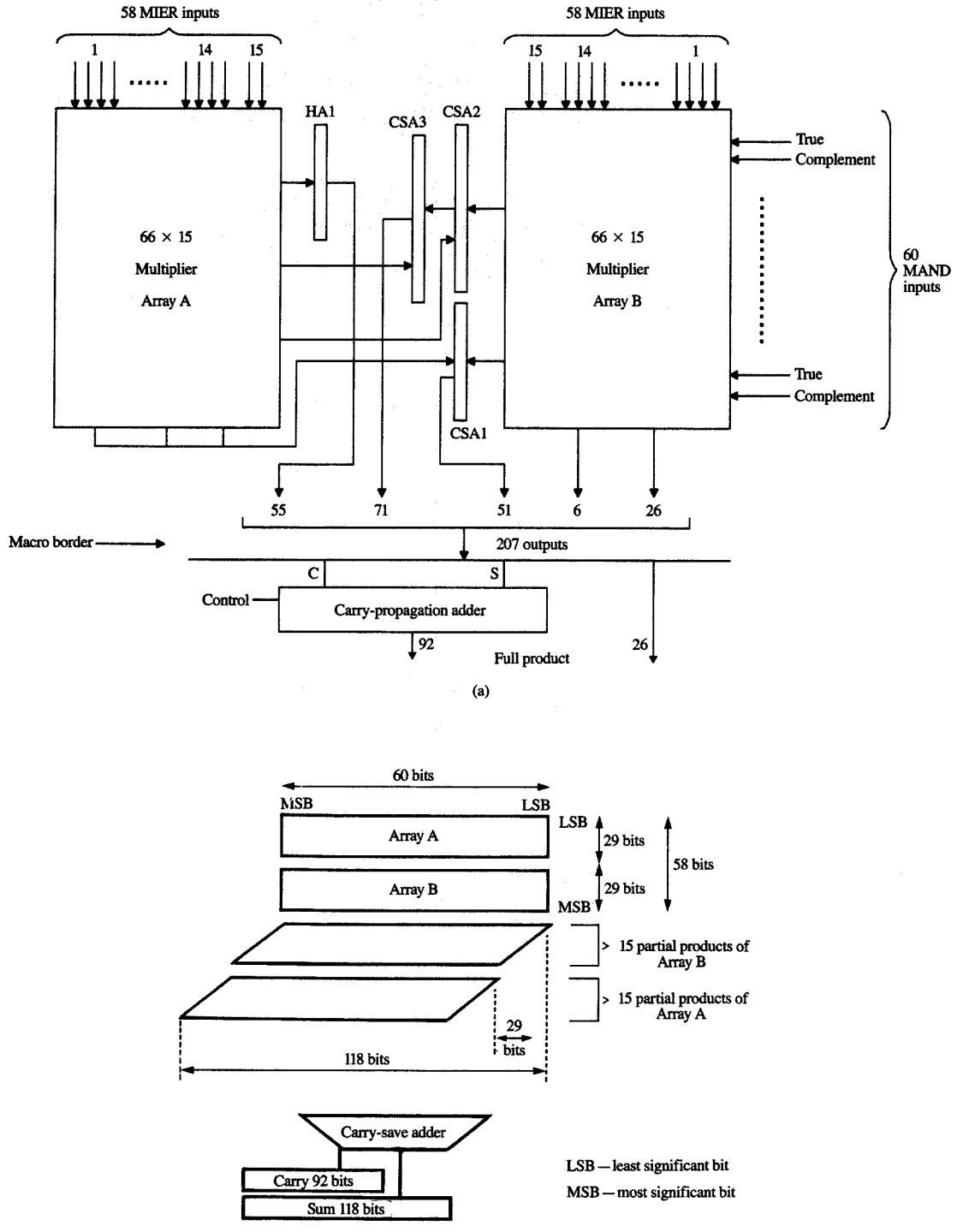


Figure 5

Multiply array: (a) physical partitioning; (b) logical description. Part (a) reprinted from [1] with permission; © 1990 IEEE.

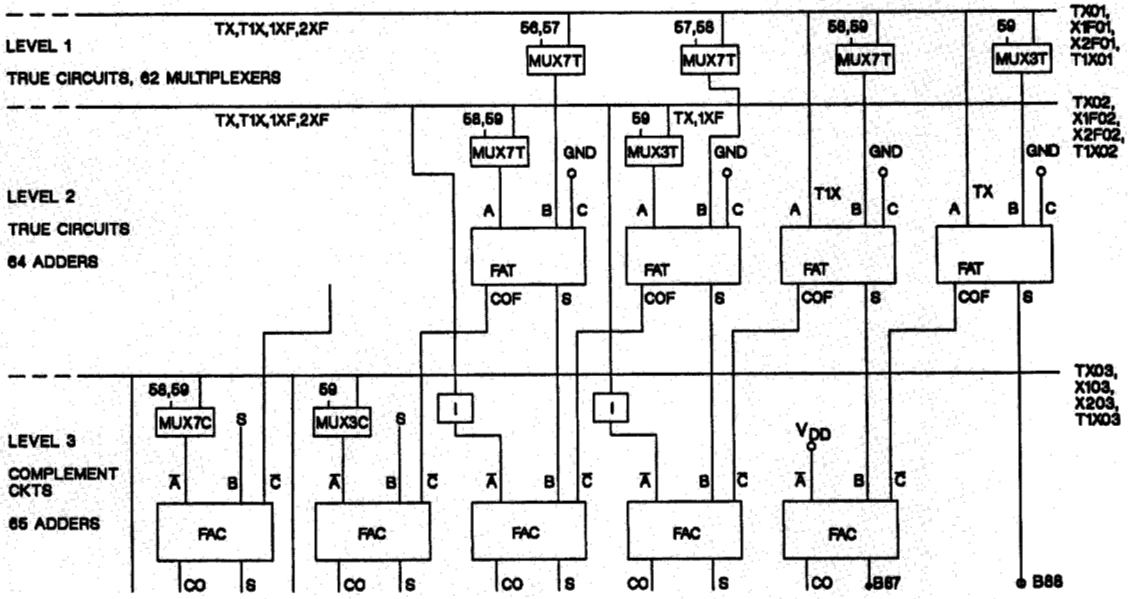


Figure 6

Multiplier array, upper corner.

stages. The modified Booth method is used to meet the first goal, and a partitioning of the multiplier array into two identical half parts to meet the latter goal.

Partitioning the array into two parts is not trivial. A special carry network has been developed to move the carry bits out of the lower significant array properly [4]. The advantages are the following: First, the carry network reduces the total multiplier delay by nearly 50%; second, it favors the structural regularity of the multiplier macro cell, as does the Booth algorithm. This regularity has the following advantages:

- The custom design is easier to handle (e.g., checking is easier).
- Only one array need be designed.
- The performance is enhanced, and the area used is reduced, by employing repeated circuits and repeated interconnection wiring. Circuit design and layout can be made very dense.
- The multiplier bit width is easily extendable.
- Regularity also means short wiring, which is a major factor affecting the performance of CMOS designs.
- Design turnaround time is reduced.

In the Type 9221 floating-point processor, the multiplier must support 60 × 58-bit multiplication, which is required for our division algorithm. A block diagram of the 60 × 58 multiplier is shown in **Figure 5**. The border of the macro cell is indicated by a dashed line. The registers at the inputs as well as at the outputs of the macro cell are part of the automatically wired logic circuitry of the chip. Inside the macro cell there are no latches, only multiplexers and adders.

The multiplier (MIER) input and output registers and the multiplicand (MAND) input registers act as input and output ports to the array. These registers are separate scan paths which enhance the testability of the multiplier. The decoders for the 2 × 29 multiplier inputs are placed before the multiplier input registers, to keep the delay of the multiplier and the rest of the data flow well balanced. The multiplier output registers are inserted before the final summation of the most significant product terms. This location was selected on the basis of the delay characteristics of the macrocell versus the cycle time of the system. Carry-save adders (CSA1, CSA2, and CSA3) and half adders (HA1) between Array A and Array B are used to combine the sums and carries coming from both arrays. Arrays A and B are mirrored on the chip, permitting all array bits to be concentrated in the middle channel for summation. The multiplicand inputs cross Array B, the carry-save adders, the half adders, and finally Array A, in straight lines.

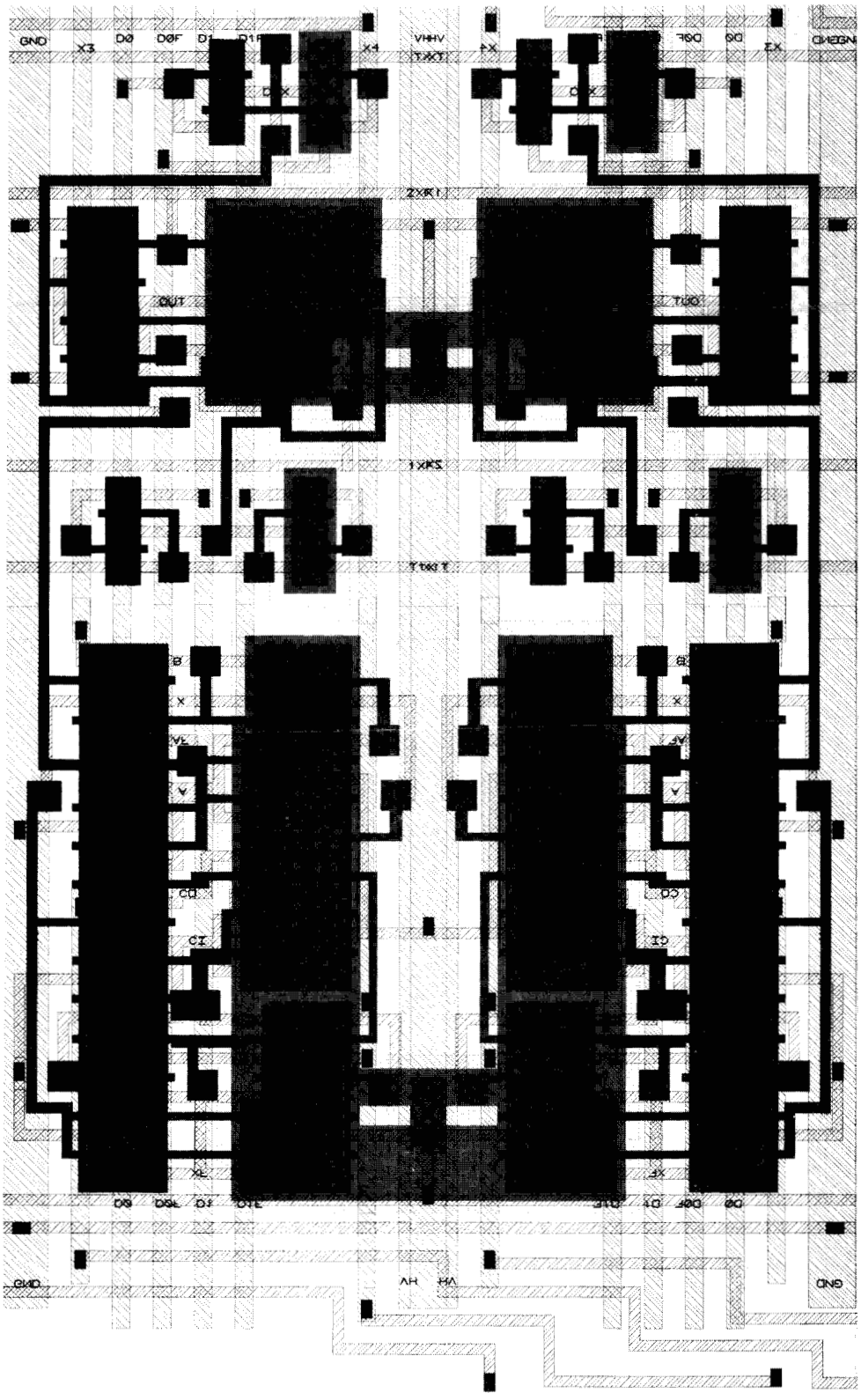
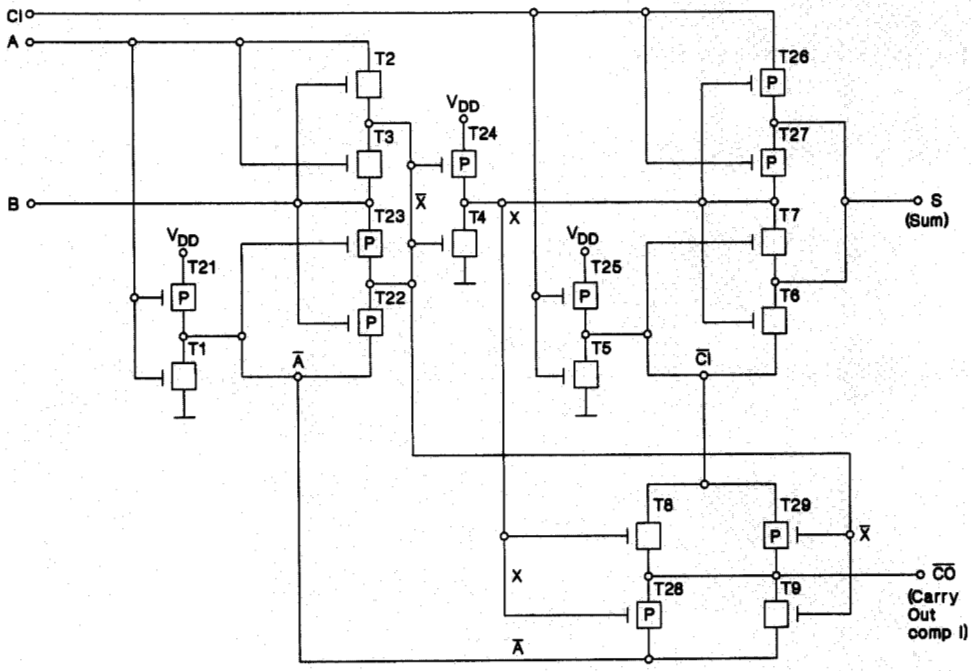
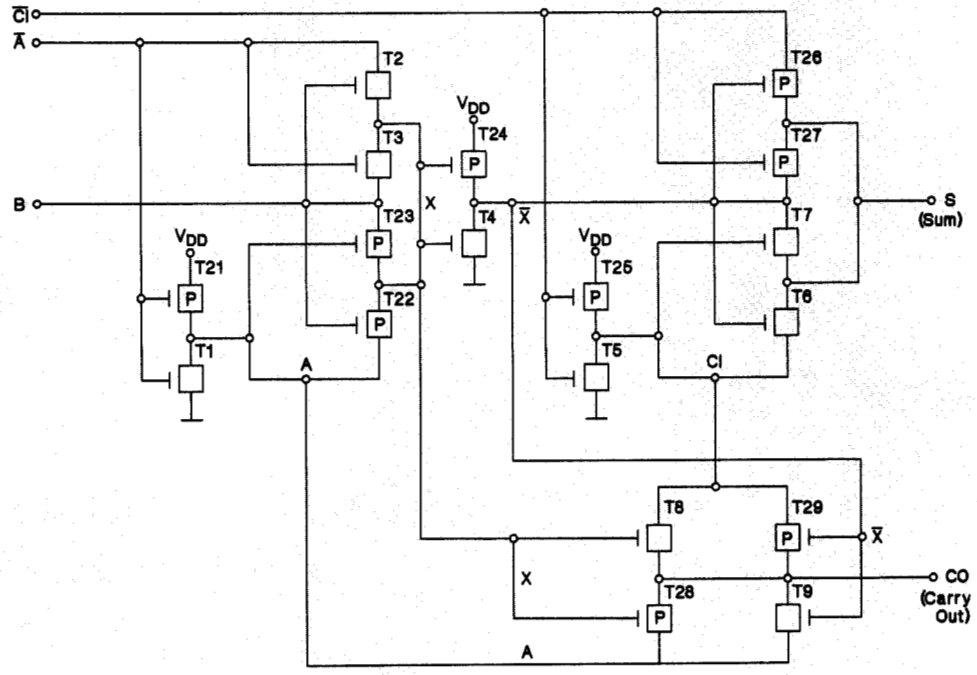


Figure 7

Mirrored pair of array elements.



(a)



(b)

Figure 8

Adder with (a) true inputs and (b) complemented inputs. Transistors are n-type unless marked P for p-type.

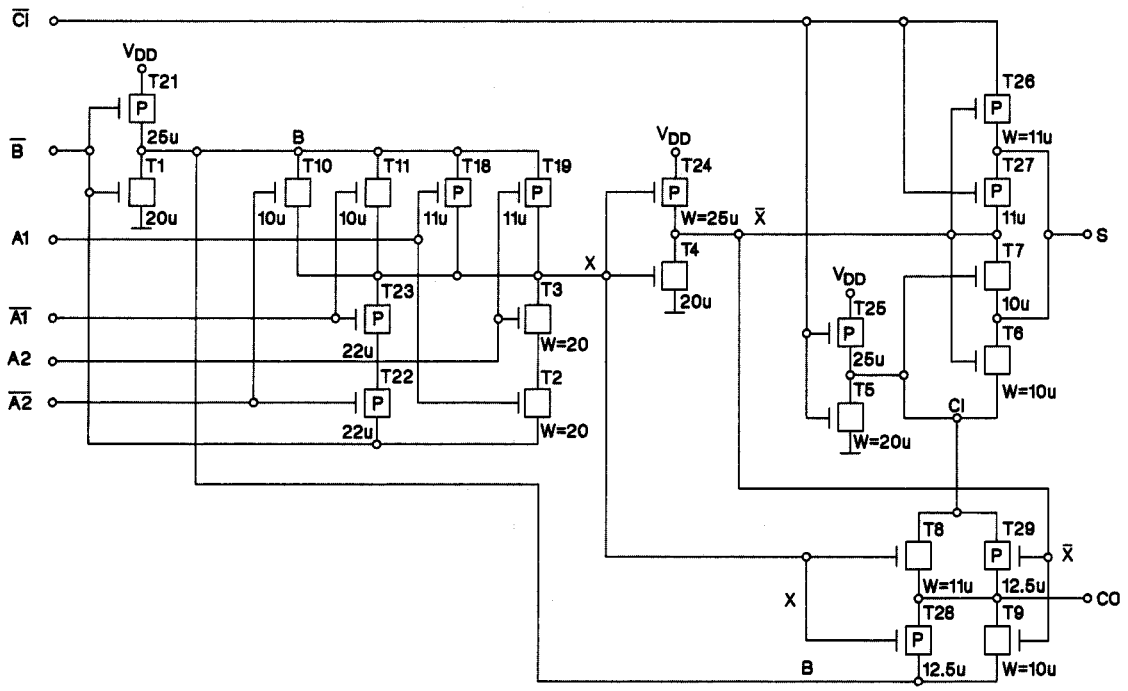


Figure 9

AND circuit integrated into the full adder. Transistors are n-type unless marked P for p-type.

To summarize, the multiplier uses the modified Booth algorithm, and consists of two identical arrays with 15 rows and 66 columns each. The first row contains only multiplexers for the selection of the Booth-encoded signals. Rows 2 through 15 contain multiplexers and full adders (carry-save adders).

Circuit organization

The structure of the multiplier arrays is partly shown in **Figure 6**, which contains the upper right corner of one array. The first row (Level 1) consists only of multiplexers (MUX7T, MUX3T). Full adders (FAT, FAC) are placed from row 2 on. To save inverters (and delay time), the circuits in row 2 are different from the circuits in row 3, a pattern that is repeated throughout the remainder of the array. The full adders in row 2 have only true input signals, a complemented carry output, and a true sum output. The full adders in row 3 have two complemented input signals, while the other input and output signals are true.

Signals 56 through 59 represent the last four (low-order) multiplicand input signals. These signals are inputs to the multiplexers. The four horizontal lines (TX01, $\cdots$) are Booth-encoded signals leading to all multiplexers of one row. In the macrocell layout, the circuits of a row are shifted by two pitches to the right compared with the circuits of the next higher row. This guarantees a rectangular array layout. There are different types of multiplexers and full adders derived from one multiplexer layout or from one full adder layout. One unique array element was optimally designed, with many personalization possibilities. It can be used on every row of the array, repeatedly. **Figure 7** shows the layout of a mirrored pair of array elements which makes the placement of the circuits in the array much simpler. Only at the right edge of the array are there irregularities. The full adder with an integrated AND-function was designed with special care from both layout and performance viewpoints, since it contributed to the slowest delay path.

Circuit description

The delay characteristics of the full adders used in the multiplier arrays directly affect the performance of the macrocell. Both the sum and the carry paths of the full

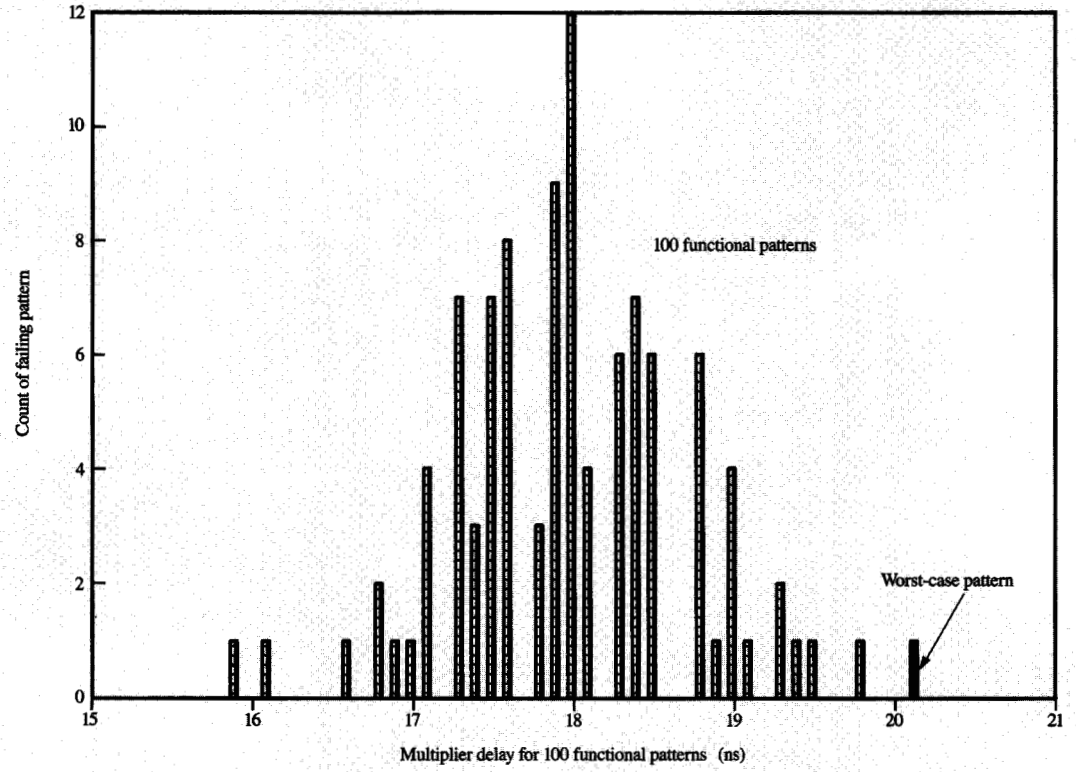


Figure 10

Pattern-dependent delay.

adder circuit must be optimized. Two different adder cells are used within the multiplier array. This scheme eliminates a delay of one inverter in each array row. **Figures 8(a)** and **8(b)** show the two full adders with alternating polarities for the carry-in (CI) and the input signal A. Input B is fed by the sum output of a previous adder and has the same polarity throughout. The circuits are nearly identical; only the two connections X and X-not are different. This eases the integration of the full adders into an array. A full adder cell consists of three inverters and twelve transmission gates. Eight transmission gates are used to generate two exclusive-OR functions, while four gates generate the carry-out function. If one follows the delay paths from one full adder to the next, a maximum of three transmission gates in series between two inverters are encountered.

Worst-case paths are, however, caused by irregularities due to the necessary AND-function between the full adders at the right edge (Figure 6). The circuit for this block is shown in **Figure 9**. The transmission gates T2, T3, T10, T11, T18, T19, T22, and T23 perform the AND-function, which is integrated into the first exclusive-OR function of the full adder. This integration makes the additional delay modest.

Test results

The typical delay of the multiplier macrocell from input to output register was measured to be 18 ns. This delay is the sum of two multiplexer delays and 14 carry-save adder delays inside the arrays, and of two carry-save adder delays and an on-chip driver delay outside the arrays. It also includes the setup time for the registers. The pattern dependency of the multiplier delay can be seen in **Figure 10**. The delays, which vary between 16 ns and 19.5 ns, are the result of applying 100 functional patterns. Finally, **Figure 11** shows a distribution of multiplier delays resulting from the application of a worst-case pattern to several chips out of five experimental lots.

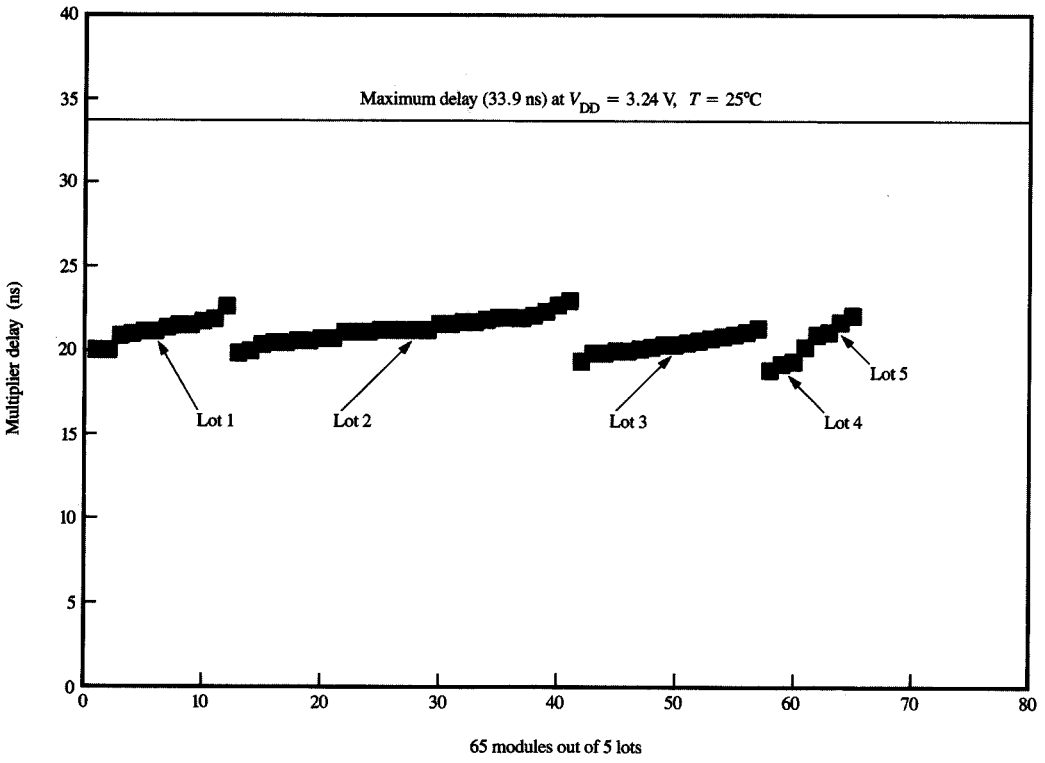


Figure 11

Worst-case delay distribution.

Verification and simulation

For simulation and verification purposes, a one-to-one logic gate model of the multiplier was written and then was merged into the entire floating-point model for circuit simulation. With this multiplier model, data were generated to further test and stress the multiplier macro, which was built on a test site. Because the macro was designed manually, additional effort was needed to check the correctness of the design. To this end, a topology-checking program named MACH1 from our Engineering Design System (EDS) was used to check the transistor shapes against the logic gate model. A library of transistor shapes was created for defining the correspondence to logic gates. The program then checked the shape connections for correct function. By using both simulation and shape checking, a correct multiplier macro was built and merged onto the floating-point chip.

Binary multiplication

Although not a floating-point instruction, binary integer multiplication is also implemented on the floating-point processor to improve performance. The floating-point processor is very highly coupled to the CPU and is an integral part of it. Because the Booth encoders are implemented outside the multiplier macro, modification to the Booth coefficients can be easily made to match the requirements for signed multiplication [5].

- *Division*

Division is performed by the floating-point processor using a modified Newton convergence method. The starting seed for the division comes from a table which is seven bits wide. In the following, we describe only operations on the fractional part of the operands. Exponent calculation is simple and is calculated separately.

Let A and B be the fractional parts of the dividend and divisor, respectively. Then let A and B be bit-normalized; i.e., the most significant bit is nonzero (for hex-format, a bit-normalization must be done before processing):

$$A = 0.1a_1a_2a_3 \cdots a_{55},$$
$$B = 0.1b_1b_2b_3 \cdots b_{55}. \quad (1)$$

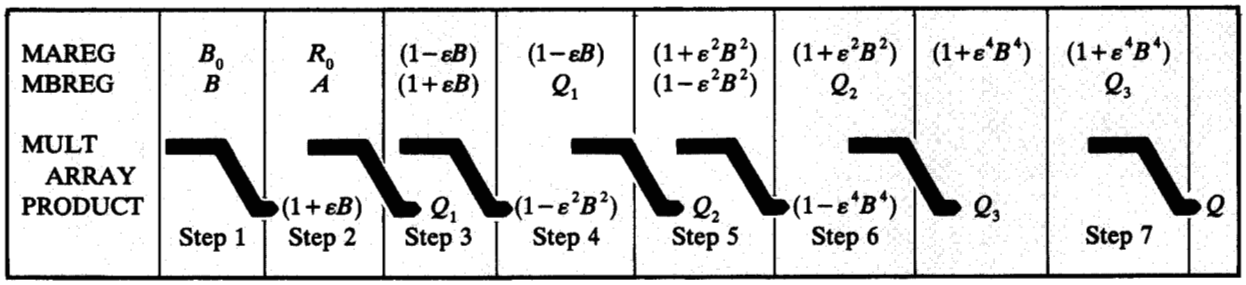


Figure 12

Example divide in pipeline structure.

Step 1 $B \cdot R_0 = 0.73216312 \times 1.36$
$= 0.99574184 = 1 + \varepsilon B,$
$1 - \varepsilon B = 1.00425816.$

Step 2 $A \cdot R_0 = 0.59786552 \times 1.36$
$= 0.81309710.$

Step 3 $(1 + \varepsilon B)(1 - \varepsilon B) = 0.99998186 = 1 - \varepsilon^2B^2,$
$1 + \varepsilon^2B^2 = 1.00001814.$

Step 4 $A \cdot R_0 \cdot (1 - \varepsilon B) = 0.81309710 \times 1.00425816$
$= 0.81655939.$

Step 5 $(1 - \varepsilon^2B^2)(1 + \varepsilon^2B^2) = 0.99998186 \times 1.00001814$
$= 0.99999999 = 1 - \varepsilon^4B^4,$
$1 + \varepsilon^4B^4 = 1.00000001.$

Step 6 $A \cdot R_0 \cdot (1 - \varepsilon B)(1 + \varepsilon^2B^2) = 0.81655939 \times 1.00001814$
$= 0.81657420.$

Step 7 $A \cdot R_0 \cdot (1 - \varepsilon B)(1 + \varepsilon^2B^2)(1 + \varepsilon^4B^4) = 0.81657420 \times 1.00000001$
$= 0.81657420 = Q.$

Comparing multiplications:

$B \cdot Q = 0.73216312 \times 0.81657420 = 0.59786551 < A,$
$B(Q + 1LSB) = 0.73216312 \times 0.81657421 = 0.59786552 = A.$

However, the low part of the product, not shown here, is not zero, so that the full product is greater than A.

Thus, the correct result is

$Q = 0.81657420.$

Note: In hex format, the result must be corrected according to the bit normalization done before processing. This step may increment the result exponent by one in the low-order bit.

Figure 13

Example divide A/B for $A = 0.59786552$ and $B = 0.73216312$.

Let R_0 be the start seed, which is the seven-bit approximation of the inverse of B:

$$R_0 = \frac{1}{B} + \varepsilon; \tag{2}$$

ε has at least seven leading zero bits.

Division is performed as follows:

$$\frac{A}{B} = \frac{A \cdot R_0}{B \cdot R_0} = \frac{A \cdot R_0}{B\left(\frac{1}{B} + \varepsilon\right)} = \frac{A \cdot R_0}{1 + \varepsilon B}$$

$$= \frac{A \cdot R_0(1 - \varepsilon B)}{(1 + \varepsilon B)(1 - \varepsilon B)} = \frac{A \cdot R_0(1 - \varepsilon B)}{1 - \varepsilon^2 B^2}$$

$$= \frac{A \cdot R_0(1 - \varepsilon B)(1 + \varepsilon^2 B^2)}{(1 - \varepsilon^2 B^2)(1 + \varepsilon^2 B^2)} = \frac{A \cdot R_0(1 - \varepsilon B)(1 + \varepsilon^2 B^2)}{1 - \varepsilon^4 B^4}$$

$$= \frac{A \cdot R_0(1 - \varepsilon B)(1 + \varepsilon^2 B^2)(1 + \varepsilon^4 B^4)}{(1 - \varepsilon^4 B^4)(1 + \varepsilon^4 B^4)}$$

$$= \frac{A \cdot R_0(1 - \varepsilon B)(1 + \varepsilon^2 B^2)(1 + \varepsilon^4 B^4)}{1 - \varepsilon^8 B^8}. \tag{3}$$

Because $\varepsilon^8 B^8$ has at least 64 leading zero bits, the quotient of A/B can be approximated by

$$\frac{A}{B} \approx A \cdot R_0(1 - \varepsilon B)(1 + \varepsilon^2 B^2)(1 + \varepsilon^4 B^4) = Q. \tag{4}$$

We need to perform only seven multiplications because the last multiplication result, which is $(1 - \varepsilon^8 B^8)$, can be discarded. It was mathematically proven that the approximation so found is at most one least significant bit away from the correct result. The precise result of the division is then determined by making the two comparing multiplications

$$A \gtrless B \cdot Q$$

and

$$A \gtrless B(Q + 1LSB), \tag{5}$$

where LSB indicates the least significant bit.

The terms $(1 - \varepsilon B)$, $(1 + \varepsilon^2 B^2)$, and $(1 + \varepsilon^4 B^4)$ used in Equation (3) are derived from the terms $(1 + \varepsilon B)$, $(1 - \varepsilon^2 B^2)$, and $(1 - \varepsilon^4 B^4)$ respectively, using special built-in circuitry. Hardware needed is minimal, whereas for the same function others need a table-lookup ROM [6].

The sequence of the division is shown in **Figure 12**. The complete division including the correcting steps is implemented in hardware and requires 14 cycles. An example of a divide using decimal numbers is illustrated in **Figure 13**. Let $A = 0.59786552$ and $B = 0.73216312$. The divide table delivers the rounded inverse of the truncated B operand, which is 0.73. Thus, $R_0 = 1/0.73 = 1.36$.

Table 2 Performance summary—floating-point chip.

Clock	37 MHz maximum
Register-to-register instruction	1 cycle typical
Register-to-memory instruction	3 cycles typical
LINPACK [7] performance	4.1 MFLOPS
Power dissipation	2 W

Summary

Figure 14 shows a wiring plot of the chip. Wiring amounted to more than 30 meters at the first iteration, resulting in an unwirable design. After two iterations, including new floor-planning, a design requiring only 18 meters of wire was produced. The chip measures 12.7 mm × 12.7 mm and contains more than 70 000 wired cells (equivalent inverter). All single-, double-, and extended-precision floating-point instructions, and the binary integer multiplication, are performed as defined by the IBM System/390 principles of operation. The data types are short format (one sign bit, seven exponent bits, and 24 fraction bits), long format (one sign bit, seven exponent bits, and 56 fraction bits), and extended format (one sign bit, seven exponent bits, and 112 fraction bits). Instructions that are heavily used, such as addition, multiplication, load, and store, are performed in one cycle, except for extended operands. In pipelining, mode, long results can be sdelivered every cycle, which is 27 ns. **Table 2** shows a performance summary of the floating-point chip.

References

1. H. Schettler, K. Getzlaff, K. Klein, C. W. Starke, J. Wilczynski, and A. Bhattacharyya, "A CMOS Mainframe Processor with a 0.5 μm Channel Length," *IEEE J. Solid State Circuits* **25**, 1166 (1990).
2. O. L. McSorley, "High Speed Arithmetic in Binary Computers," *Proc. IRE* **49**, 67 (1961).
3. C. S. Wallace, "A Suggestion for Parallel Multiplier," *IEEE Trans. Electron. Computers* **EC-13**, 14 (1964).
4. K. G. Getzlaff, S. Dao-Trong, and K. Helwig, "Multiplizierwerk (Multiplier)," European Patent Office, Reg. Number 89102956.3, February 1989.
5. A. D. Booth, "A Signed Binary Multiplication Algorithm," *Quart. J. Mech. Appl. Math.* **IV**, Part 2, 163 (1951).
6. K. Kaneko, T. Okamoto, M. Nakajima, Y. Nakakura, S. Gokita, J. Nishikawa, Y. Tanikawa, and H. Kadota, "A VLSI RISC with 20 MFLOPS Peak, 64-Bit Floating Point Unit," *IEEE J. Solid State Circuits* **24**, 1331 (1989).
7. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.

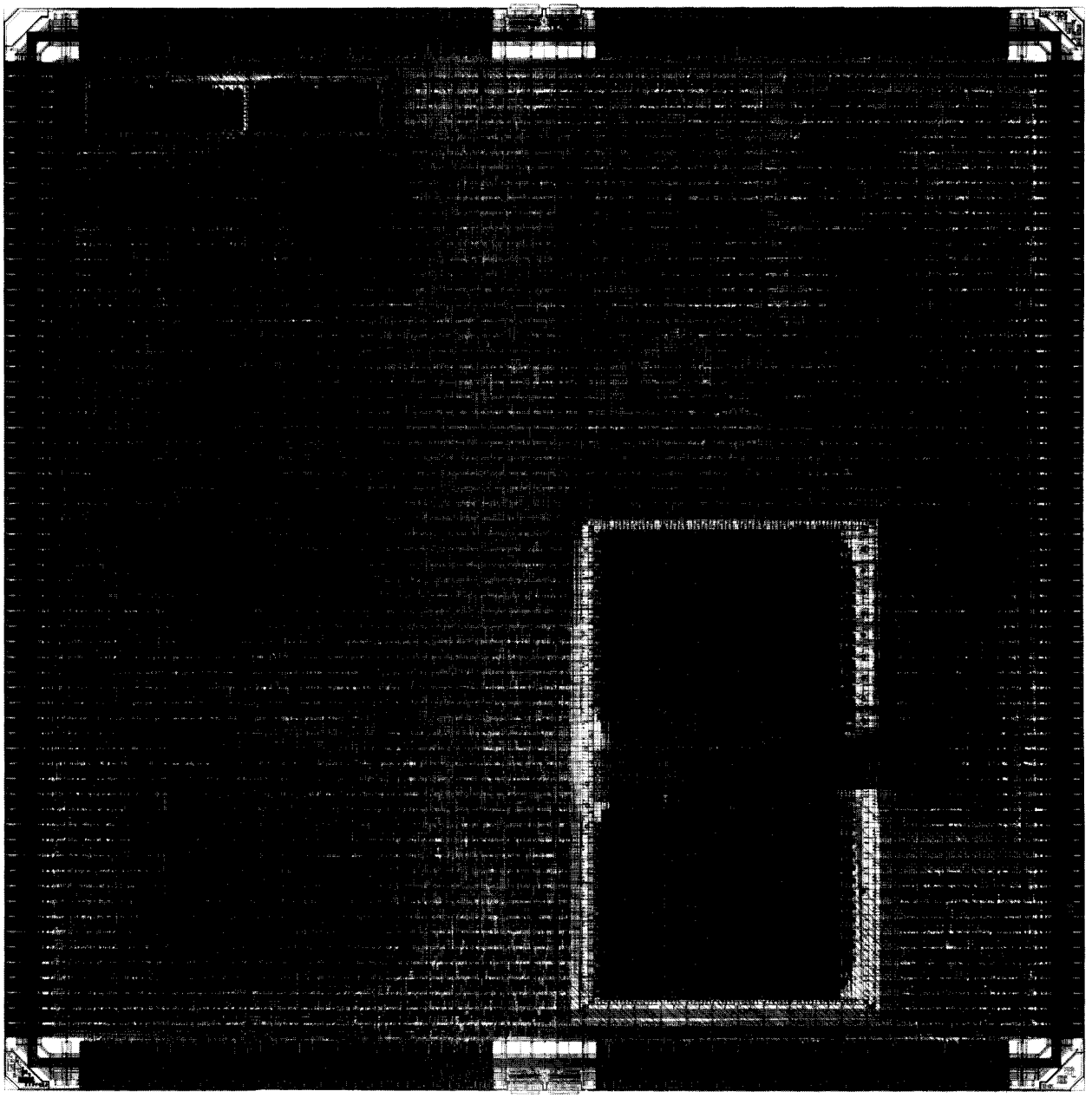


Figure 14

Floating-point chip wiring.

Received April 18, 1991; accepted for publication September 18, 1991

Son Dao-Trong *IBM Germany, Schoenaicherstrasse 220, 7030 Boeblingen, Germany (DAOTRONG at BOEVM2).*
Dr. Dao-Trong received his Dipl.-Ing. degree in electrical engineering from the University of Karlsruhe. In 1979 he joined the University, where he worked on logic synthesis tools in a gate-array environment. He received his Doctor-Ing. degree in 1984 and joined the IBM Laboratory in Boeblingen, where he first worked on packaging design and noise analysis. Since 1987 he has worked on floating-point design as a team leader. In 1989, Dr. Dao-Trong received a patent on the multiplier array which is embedded in the floating-point chip of the recent Enterprise System/9000 9221 models. He is currently working on the design of the next model, stressing design methodology.

Klaus Helwig *IBM Germany, Schoenaicherstrasse 220, 7030 Boeblingen, Germany (HELWIG at BOEVM4).* Mr. Helwig received the Dipl.-Ing. degree in electrical engineering from the Technical University of Vienna, Austria, in 1965. From 1966 to 1968, he was employed with Siemens AG, Munich, Germany, where he was engaged in the development of MOS transistors. In 1968, he joined the Components Group of the IBM Laboratories, Boeblingen, Germany, working on digital integrated circuit designs especially for monolithic memory applications. During the past years, Mr. Helwig has been engaged in the design of CMOS memory and logic arrays.