# Carry-Free Radix-2 Subtractive Division Algorithm and Implementation of the Divider

Jen-Shiun Chiang, Hung-Da Chung and Min-Show Tsai

*Department of Electrical Engineering*
*Tamkang University*
*Tamsui, Taipei, Taiwan*
*E-mail: chiang@ee.tku.edu.tw*

## Abstract

A carry-free subtractive division algorithm is proposed in this paper. In the conventional subtractive divider, adders are used to find both quotient bit and partial remainder. Carries are usually generated in the addition operation, and it may take time to finish the operation, therefore, the carry propagation delay usually is a bottleneck of the conventional subtractive divider. In this paper, a carry-free scheme is proposed by using signed bit representation to represent both quotient and partial remainder. During the arithmetic operation, a special technique is used to decide the quotient bit, and the new partial remainder can be found further by a table lookup-like method. The signed bit format of the quotient can be converted by on-the-fly conversion to the binary representation. Based on this algorithm a 32-b/32-b divider is designed and implemented, and the simulation shows that the divider works well.

***Key Words*** **:** Divider, radix-2, quotient bit, partial remainder, carry propagation delay, high speed, Svobota-Tung division algorithm, signed digit, prescaling, table look-up, on-the-fly conversion

## 1. Introduction

Due to the progress of high-speed computation and multimedia application, the hardware implementation of all basic arithmetic operations becomes important in the design of microprocessors or DSP processors. Whereas the designs of fast and efficient adders and multipliers are well understood, the divider still remains a serious design challenge [8]. Generally, there are two techniques for performing division, the digit-recurrence approach and the Newton-Raphson method [12]. The digit-recurrence technique uses addition/subtraction and shift in a manner similar to the traditional paper-and-pencil approach. The Newton-Raphson method uses multiplication (multiplicative inverse) and addition to develop increasingly accurate approximations to the desired quotient [12]. This paper is concentrated on the first approach.

The digit-recurrence algorithm obtains the quotient digit-wisely. In the very well known SRT division [1, 5] the quotient digit is selected by inspecting a few of the most significant digits of both remainder and divisor. In 1963, Svoboda [11] published a division algorithm where the quotient digit is estimated without considering the divisor. In Svobota's approach, if the estimate is not accurate, an overflow occurs and the compensation is carried out. Later Tung [13, 14] investigated the implementation of the Svoboda division, and proposed a signed digit-set approach. However, the Svoboda-Tung algorithm has two drawbacks that prevent the VLSI implementation. (1) It is valid only for radixes greater than 4. (2) Because of the possible compensation owing to overflow on the iteration, the quotient digit is actually selected from an over-redundant digit-set (i.e. the quotient

digit may be greater than the radix) [10]. Therefore, the generation of some of the multiples of the divisor is not straightforward.

The algorithm proposed in this paper to describe the development of a new division approach is based on the Svoboda-Tung technique but overcomes the drawbacks mentioned in the previous paragraph. The possible compensation due to overflow on the iteration is avoided by "rewrite" the two most significant digits, such that the quotient digit is selected from the same digit-set as of the remainder. These approaches simplify the selection of the generation of multiples of the divisor and quotient digit.

This paper is structured as follows. Section II reviews very briefly the principles of the digit-recurrence and Svoboda-Tung divisions. Section III discusses the development of the new division algorithm. In section IV, we describe this division algorithm in the VLSI implementation and the simulation results. Finally, we make a conclusion in section V.

## 2. Basic Properties of the Division

Division instructions are executed in most of today's digital computers via a recursive procedure. The time required for digital division is spent primarily in the repeated execution of this recursive procedure [2, 10, 14]. Various division methods can be described by the following recursive formula [9].

$$R^{(j+1)}=r\times R^{(j)}-q_{j+1}\times D \quad (2.1)$$

where

j=0, 1,…. , n –1, is the recursion index;

D is the divisor;

$q_{j+1}$ is the (j+1)th quotient digit to the right of the radix point;

n is the word length of the quotient;

$q_0$ is the sign;

r is the radix;

$R^{(j+1)}$ is the partial remainder after the determination of the (j+1)th quotient digit;

$R^{(0)}$ is the dividend (initial partial remainder).

Without loss of generality, it is assumed that both the dividend $R^{(0)}$ and the divisor D are fractions, so is the generated quotient Q,

$$Q=q_0q_1q_2q_3q_4.\ldots.q_{n-1}q_n . \quad (2.2)$$

In Eq. (2.2) $q_0$ is the sign of the quotient determined by the following operation,

$$q_0=r_0^{(0)}\oplus d_0 . \quad (2.3)$$

In Eq. (2.3) $r_0^{(0)}$ and $d_0$ are signs of the dividend and divisor, respectively. The radix point is located between the sign, $q_0$, and the most significant digit, $q_1$. The final remainder could be either positive or negative depending on the method used. For the conventional restoring division, the sign of the remainder is identical to that of the dividend.

The division procedure can be verified by applying the recursive Eq. (2.1) repeatedly.

For j=0

$$R^{(1)}=r\times R^{(0)}-q_1\times D \quad (2.4)$$

For j=1

$$R^{(2)}=r\times R^{(1)}-q_2\times D$$
$$= r^2\times R^{(0)}-(r\times q_1+ q_2)D \quad (2.5)$$

…….

For j=n –1

$$R^{(n)}=r^n\times R^{(0)}-(r^{n-1}\times q_1+\ldots+r\times q_{n-1}+ q_n)D \quad (2.6)$$

The above iterative derivation shows that the division procedure consists of a sequence of additions, subtractions, or shifts corresponding to the negative, positive, or zero value of the successively generated quotient $q_{j+1}$, for j=0,1,2,..., n-1. Eq. (2.6) can be rewritten as follows:

$$\frac{R^{(0)}}{D}=\sum_{j=1}^{n} r^{-j}\times q_j+\frac{r^{-n}\times R^{(n)}}{D} \quad (2.7)$$

where $$Q=\sum_{j=1}^{n} r^{-j}\times q_j \quad (2.8)$$

and $$R=r^{-n}\times R^{(n)} \quad (2.9)$$

represent the quotient and final remainder respectively.

The SRT division [12] uses a redundant signed digit-set to represent the quotient Q and selects the quotient digit $q_{j+1}$ by inspecting a few of the most significant digits of both the remainder $R^{(j)}$ and divisor D. In 1963, Svoboda [9] published a division algorithm where $q_{j+1}$ is estimated independent of the divisor D. The estimate of $q_{j+1}$ is the most significant digit $r_1^{(j)}$ of the jth remainder. If the estimate is not accurate, an overflow occurs and the compensation is carried out. The Svoboda algorithm is valid for a divisor in the range $1\leqq D<1+1/r$ and uses the conventional digit-set S={0,1,…, r-1}. Tung [13, 14] investigated the implementation of the Svoboda division and proposed a signed digit-set $S_{<r,x>}=\{\bar{x}, ...., \bar{1}, 0, 1, ..., x\}$ (r is the radix, $\bar{x}$ stands for –x, and $\left\lceil \frac{r}{2} \right\rceil +1\leq x\leq r-1$ ) for the computation.

## 3. The New Division Algorithm

This radix-2 division algorithm to obtain the digit-wise is based on the recurrence of

$$R^{(j+1)} = 2\times R^{(j)} - r_1^{(j)}\times D \qquad (3.1)$$

Where

$R^{(j)}$ is the remainder after the jth iteration;

$r_1^{(j)}$ is the quotient digit selected at the (j+1)th step;

D is the divisor.

The dividend and divisor are in the IEEE 754 normalized format in this division algorithm. In order to avoid the possible overflow on the iteration, prescaling is proceeded first [6, 7]. If the divisor, D, is greater than $1.1_2$, prescaling is proceeded and the dividend, $R^{(0)}$, and divisor, D, are multiplied by $0.75_{10}$ (or $0.11_2$) respectively. By the prescaling procedure, the iteration of the operation will not generate any overflow, and the division operation can undergo smoothly [8]. In order to have carry-free subtraction, signed bit representation is applied to the quotient and partial remainder. In the signed bit representation, the signed bit is selected from set $\{\bar{1}, 0, 1\}$, where $\bar{1} = -1$. Since $-1 = 0 - 1$ (or $\bar{1} = 0 - 1$), $1 = 1 - 0$, and $0 = 0 - 0$, therefore, we can use two sets of data, positive part and negative part, to represent a signed bit number. For example

$$1\bar{1}0\bar{1}\bar{1}010 = 10000010 - 01011000,$$

We call 10000010 the positive part, and 01011000 the negative part. In the arithmetic operation, these two parts work concurrently, and the speed of the operation can be increased.

During the addition or subtraction operation, the digit may become +2 or –2 and thus may cause carry propagation. In order to prevent the carry propagation, we use two bits to represent each digit of the positive part and negative part of the signed bit number. For example

$$\begin{aligned} 1\bar{2}020\bar{1}0\bar{2} &= 10020000 - 02000102 \\ &= \underline{01}\ \underline{00}\ \underline{00}\ \underline{10}\ \underline{00}\ \underline{00}\ \underline{00}\ \underline{00} \\ &\quad - \underline{00}\ \underline{10}\ \underline{00}\ \underline{00}\ \underline{00}\ \underline{01}\ \underline{00}\ \underline{10}. \end{aligned}$$

By the above arrangement, the carry propagation in the addition or subtraction operation can be avoided.

After the addition or subtraction mentioned in the previous paragraph, the digit may be 2 or –2. If we do not make adjustment, the digit may diverge (overflow) in the future addition or subtraction. Therefore, the adjustment is made after the addition or subtraction, and we call this step as the "digit adjustment". In the digit adjustment, the positive part and negative part are independent and can work concurrently. Without loss of generality, let us discuss the positive part, and the negative part is in the same manner. In order to increase the efficiency, we partition the positive part and negative part of the number into several segments. Actually, the number of the segments can be decided by the designer, and for simplicity here we set each segment to four digits. In order to avoid the divergence of the digit, the digit can not be greater than 1, i.e., if the digit is 2, then it has to be adjusted. For example, 1002 has to be adjusted to a five-bit number, 01010, and the MSB of this five-bit number is the carry bit. This adjustment can be finished by two approaches. One is to precalculate the results and store them in a table, and we can look it up from the table later. The other approach is to use a 4-bit adder to finish it. For example, 1002 can be adjusted as

$$\begin{array}{rl} 1002 \rightarrow & \underline{0\,1}\ \underline{0\,0}\ \underline{0\,0}\ \underline{1\,0} \\ & \quad 1 \quad 0 \quad 0 \quad 0 \\ + & 0\ \ 0\ \ 0\ \ 1 \\ \hline & \underline{0\,0}\ \underline{0\,1}\ \underline{0\,0}\ \underline{0\,1}\ \underline{0\,0} \end{array}$$

All the higher bits are shifted left one position and are added to the lower bits. The divider that we will design uses the table lookup approach.

In the digit adjustment, there may be a carry generated, for example

$$2010 = \underline{1}\ 0\,0\,1\,0.$$

Therefore, we have to adjust it again. If there is a carry from the lower segment, a "1" has to be added to the current segment. If the current segment is 01111, a carry is generated to the higher segment. However, the largest number is 2222 in a segment, and the digit adjustment is 11110, thus we do not need to worry about the case of 11111 that may cause another carry.

Let us briefly summarize the digit adjustment. From the above discussion, the digit adjustment needs two passes. The first pass is to adjust each segment with digit 2 or $\bar{2}$. If a carry is generated in the first pass, the second pass is needed. In the second pass, the carry from the lower segment is added to its higher adjacent segment. Since the largest adjusted segment is 1110, the carry from the lower segment makes the current segment to be 1111 and no further carry is generated, therefore, it is carry free in the second pass. The implementation of the digit adjustment can be accomplished by Fig. 1. In Fig. 1, the top table of segment i is used to finish the digitadjustment of the first pass, and the carry is added to the bottom adder of segment i+1 to finish the second pass. Actually, the bottom adder can be replaced by a counter or some combinational circuit.

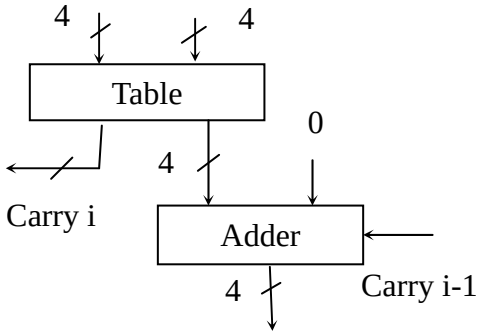


Figure 1. Digit Adjustment for the ith Segment

After the digit adjustment, we have to combine the positive and negative part of the number to a signed bit representation. We call this step as "refresh" step. In the refresh step, the digit in the positive and negative part may be +1 and -1 simultaneously, and these +1 and -1 can be canceled. For example

| | |
|---|---|
| Positive part | $10\,1\,0\;\;\bar{1}\,\bar{1}\,0\,1$ |
| Negative part | $0\,0\,0\,1\;\;\bar{1}\,\bar{1}\,0\,0$ |
| Signed bit number | $10\bar{1}\bar{1}\;\;\underline{0}\,\underline{0}\,0\,1$ |

In the procedure of quotient searching, '0' means shift, therefore the speed of the divison can be increased. The refresh step can thus increase the speed of the operation.

Now let us discuss the iteration steps. Since it is an operation of signed bit numbers, the dividend and divisor have to be converted to the signed bit number representation. Here we simply set the negative part to all zeroes and add them to the dividend and divisor, and convert the dividend and divisor to 2-bit representation for the positive part. The first iteration of the division is to subtract the divisor from the dividend to find the first partial remainder and set the first quotient bit to 1. After the subtraction, the partial remainder has to do "digit adjustment" and "refresh" to find the correct partial remainder. Then we have to decide the next quotient bit and next step operation. There are three cases for the next quotient bit.

(1) If the MSB of the partial remainder is 1, the next quotient bit is set to 1 and the next operation is subtraction (Ri–D).

(2) If the MSB of the partial remainder is $\bar{1}$ (–1), the next quotient bit is set to $\bar{1}$ and the next operation is addition (Ri+D).

(3) If the MSB of the partial remainder is 0, the next quotient bit is 0 and the next operation is shift.

From case (3) we find when the MSB of the partial remainder is 0, the next operation is simply shift. By this characteristic, we can simplify the operation further. By inspecting the most significant two digits, if the two digits are $1\,\bar{1}$ or $\bar{1}\,1$, they can be rewritten to be $0\,1$ or $0\,\bar{1}$ respectively but keep their values unchanged. We call this step as "rewrite". The rewrite step can simplify the iteration operation. After the quotient bit and next step operation is decided; the iteration will proceed until find the expected quotient.

Since the quotient is represented in the signed-bit format, we need to convert it to the binary number format. The well-known on-the-fly conversion is applied [8]. The formula of the on-the-fly conversion is listed as follows [5].

$$Q[K+1]=\begin{cases} Q[K]+2^{-K}, & q_{K+1}=1 \\ Q[K], & q_{K+1}=0 \\ T[K]+2^{-K}, & q_{K+1}=-1 \end{cases}$$

$$T[K+1]=\begin{cases} Q[K], & q_{K+1}=1 \\ T[K]+2^{-K}, & q_{K+1}=0 \\ T[K], & q_{K+1}=-1 \end{cases}$$

Since on-the-fly conversion and the division iteration can operate concurrently, we can find the quotient in binary format after the division iteration.

We have described the iteration operation of the division procedure and the procedure can be summarized as follows.

Step1 Prescaling
Step2 Convert the dividend and divisor to signed bit representation.
Step3 Subtract the divisor from the dividend to find the partial remainder and set quotient bit to 1.
Step4 Digit adjustment of the partial remainder.
Step5 Refresh the partial remainder.
Step6 Rewrite the partial remainder.
Step7 Decide the quotient bit and the operation of next iteration. On-the-fly conversion is used to convert the signed bit format of the quotient bit to the binary format.
Step8 Repeat Step 4 to Step 7 till finish the iteration times.

he flow chart of the division algorithm is shown in Fig. 2.

## 4. The Design and Simulation of the 32-b/32-b Divider

By the division algorithm mentioned in the previous section, a 32-b/32-b divider is designed in Verilog HDL. The architecture of the proposed divider is shown in Fig. 3. From the Verilog HDL

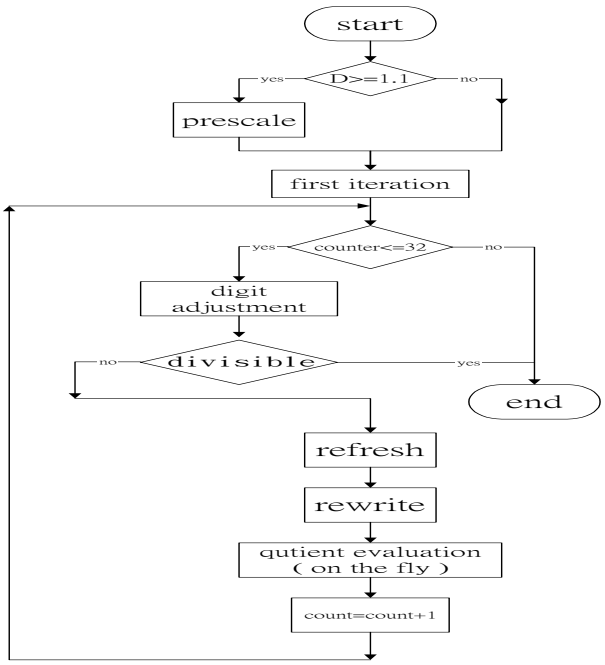


Fig. 2. The Algorithm of the Division

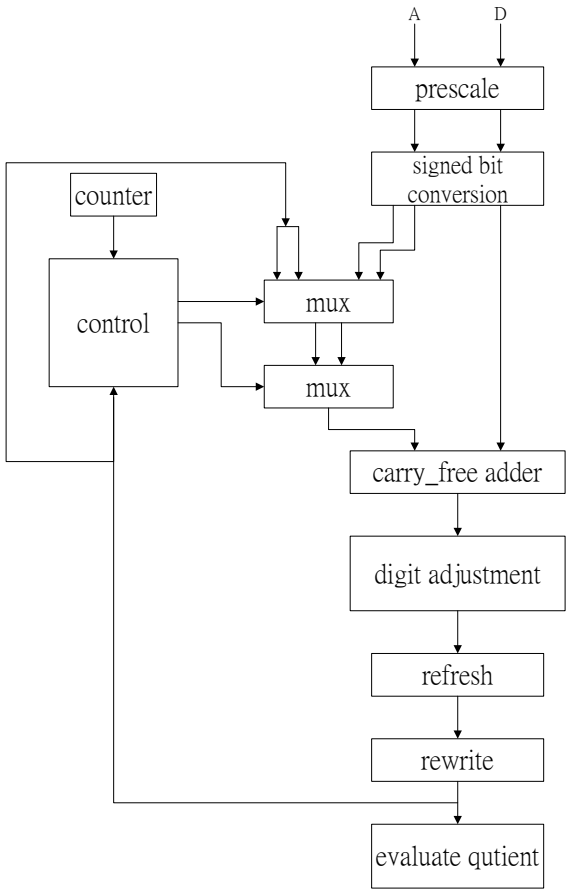


Fig. 3. The Architecture of the Divider

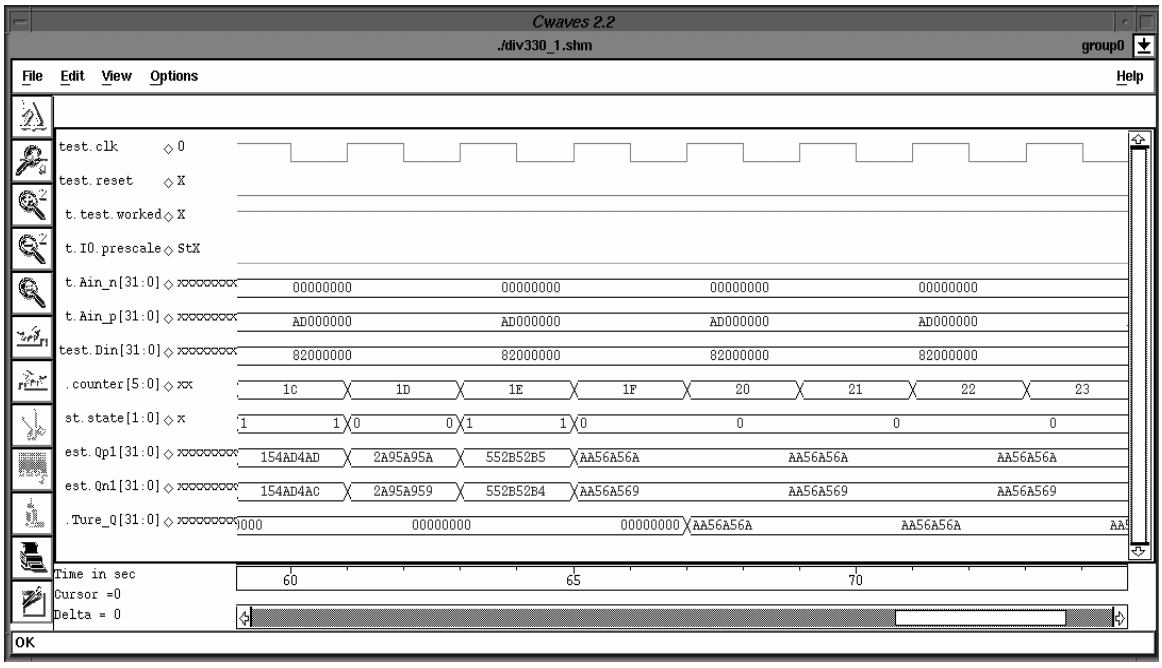


Fig. 4. Functional Simulation of the Division

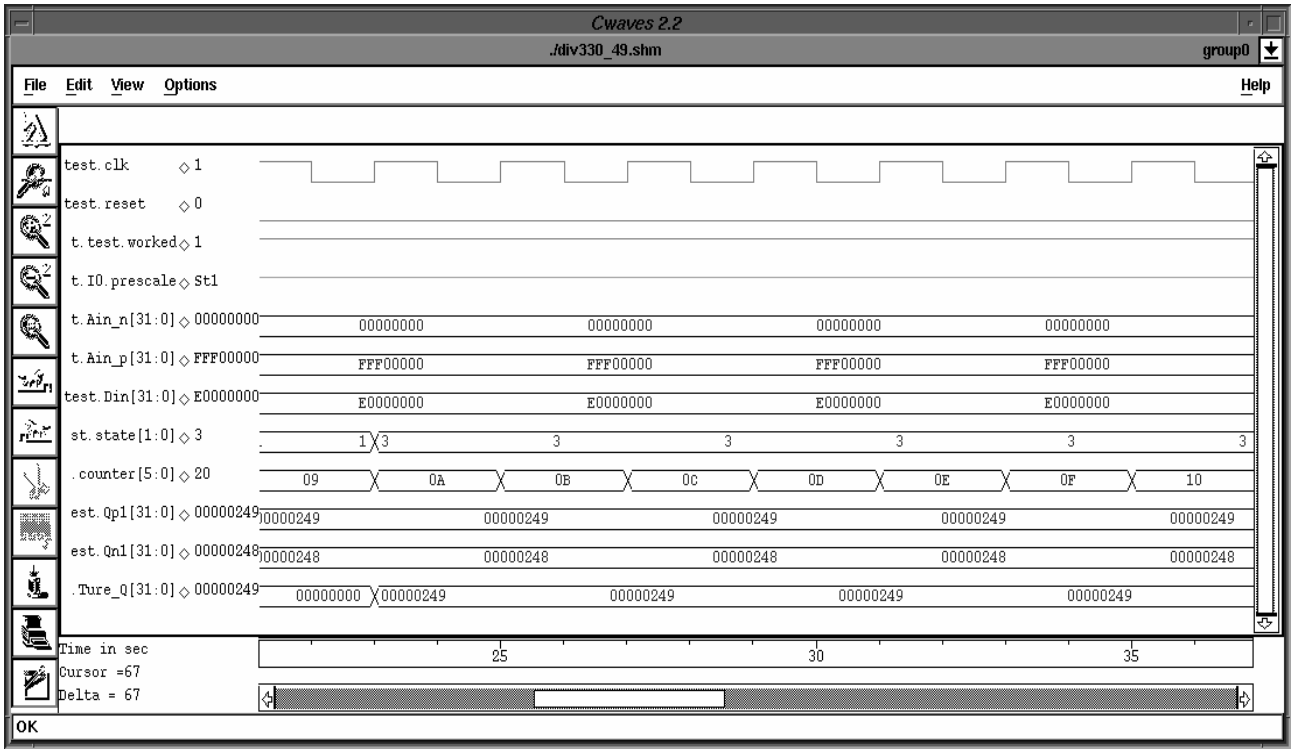


Fig. 5. Gate-level Simulation of the Division

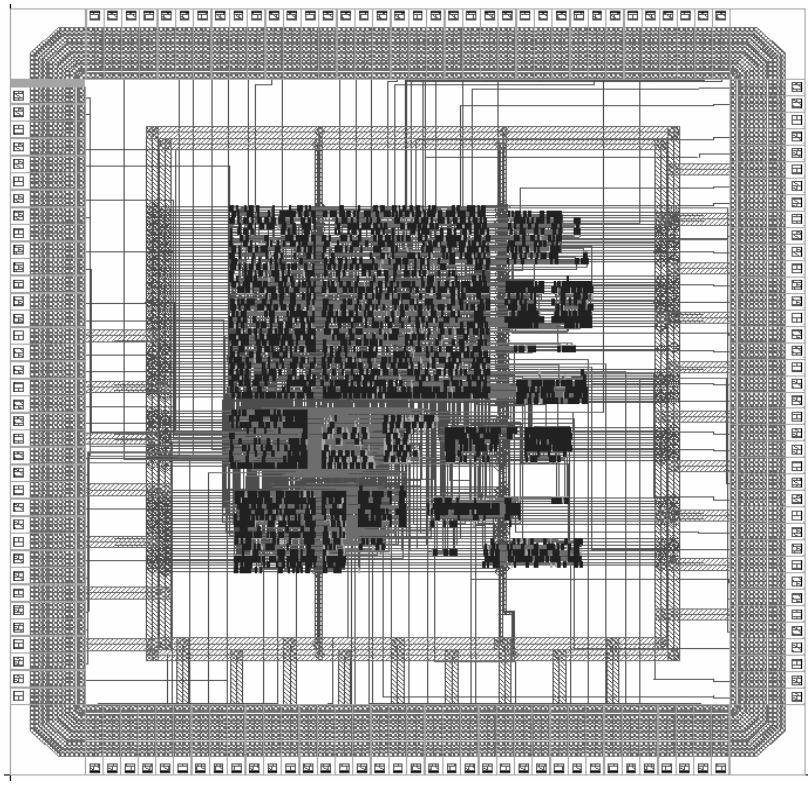

Fig. 6. VLSI Layout of the 32-b/32-b Divider

simulation, we find the speed and efficiency is very good. The Verilog HDL code is synthesized by the SYNOPSYS further, and the VLSI Layout (TSMC'S 0.6 $\mu$m process) is also finished. The core area is 3250 $\mu$m × 3250 $\mu$m. The functional simulation of the 32-b/32-b divider is shown in Fig. 2. The gate level simulation of the divider is shown in Fig. 5. From Fig. 5, we find that each iteration takes 7ns. The corresponding VLSI layout is shown in Fig. 6.

## 5. Conclusion

A new digit-recurrence division algorithm based on the Svoboda-Tung technique has been described. The new algorithm overcomes the drawbacks of the Svoboda-Tung division. In this division algorithm, we use "digit adjustment" and "refresh" technique to avoid the carry propagation generated by the addition or subtraction of the partial remainder and divisor. The "rewrite" technique can make the iteration operation more efficient and faster. Based on the division algorithm a 32-b/32-b divider is designed in Verilog HDL. The simulation shows that this divider works very well.

## References


[1] Atkins, D. E., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders", *IEEE Trans. on Comp*., vol. C-17, no. 10, Oct., pp. 925-934 (1968).

[2] Bashagha, A. E. and Ibrahim, M. K., "A New Digit-Serial Divider Architecture", *Int. J. of Electronics*, vol.75, no. 7, July, pp. 133-140 (1993).

[3] Burgess, N., "A Fast Division Algorithm for VLSI", *IEEE Int. Conf. On Computer Design*, pp. 560-563 (1991).

[4] Cortadella, J. and Lang, T., "High-Radix Division and Square-Root with Speculation", *IEEE Trans. on Comp.*, vol. 43, no. 8, pp. 919-931, Aug. (1994).

[5] Ercegovac, M. D., Lang, T., "ON-the-Fly Conversion of Redundant into Conversion Representations", *IEEE Trans. on Comp.*, vol. C-36, no. 7, July, pp. 895-897 (1987).

[6] Ercegovac, D. and Lang, T., "Simple Raduix-4 with Scaling", *IEEE Trans. on Comp*., vol. 39, no. 9, Aug., pp. 1204-1208 (1990).

[7] Ercegovac, M. D. and Lang, T., "Simple Radix-4 Division Unit with Operands Scaling", *IEEE Trans. on Comp.*, vol. 49, no. 9, Sept., pp. 1204-1208 (1990).

[8] Ercegovac, M. D. and Lang, T., *Division and Square Root: Digit-recurrence Algorithms and Implementations*, The Netherlands: Kluwer Academic Publishers (1994).

[9] Hwang, K., *Computer Arithmetic Principles, Architecture, and Design*, John Wiley and Sons (1979).

[10] Montuschj, P. and Cimiera, L., "Over-Redundant Digit Sets and the Design of Digit-by-Digit Division Units," *IEEE Trans. on Comp.*, vol. 43, no. 3, pp. 269-279, March (1994).

[11] Svoboda, A., "An Algorithm for Division", *Information Processing Machines*, no. 9, pp. 25-32, Sept. (1963).

[12] Swartzlander, E. E., *Computer Arithmetic*, vol. 1, Los Alamitos-California, IEEE Computer Society Press, pp. 156-157 (1990)

[13] Tung, C., "A Division Algorithm for Signed-Digit Arithmetic", *IEEE Trans. on Comp.*, vol. C-17, no. 9, pp. 887-889, Sept. (1968).

[14] Tung, C., "Signed-Digit Division Using Combinational Arithmetic", *IEEE Trans. on Comp.*, vol. C-19, no. 8, pp. 746-748, Aug. (1970).

[15] Williams, T. E. and Horowitz, M. A., "A 160ns 54-bit CMOS Division Implementation Using Self-Timing and Symmetrically Overlapped SRT Stages", *10th IEEE Symp. Computer Arithmetic*, Grenoble, France, pp. 210-217, June (1991).


***Manuscript Recceived: Nov. 18, 1999***
***Revision Received: Oct. 20, 2000***
***And Accepted: Nov. 4, 2000***