resistor tree branch currents, provided that the particular resistor in question is not part of a fundamental cut set which includes other resistors;

inductor link voltages, provided that the particular inductor in question is not part of a fundamental loop which includes other inductors;

capacitor tree branch currents, provided that the particular capacitor in question is not part of a fundamental cut set which includes other capacitors.

ACKNOWLEDGMENT

The author is grateful for the vital contributions made to the contents of this note by Dr. R. M. Warten.

REFERENCES

[1] T. R. Bashkow, "The A matrix, new network description," *IRE Trans. Circuit Theory*, vol. CT-4, pp. 117–119, September 1957.
[2] P. R. Bryant, "The explicit form of Bashkow's A matrix," *IRE Trans. Circuit Theory (Correspondence)*, vol. CT-9, pp. 303–306, September 1962.
[3] E. Kuh and R. A. Rohrer, "The state-variable approach to network analysis," *Proc. IEEE*, vol. 53, pp. 672–686, July 1965.
[4] S. Seshu and M. Reed, *Linear Graphs and Electrical Networks*. Reading, Mass: Addison-Wesley, 1961.
[5] "Automated digital computer program for determining responses of electronic circuits to transient nuclear radiation (SCEPTRE)," vol. 1, IBM Electronics Systems Center, Owego, N. Y., Rept. 66-928-611, September 1966.

High-Speed Binary-to-Decimal Conversion

M. S. SCHMOOKLER, MEMBER, IEEE

***Abstract*—This note describes several methods of performing fast, efficient, binary-to-decimal conversion. With a modest amount of circuitry, an order of magnitude speed improvement is obtained. This achievement offers a unique advantage to general-purpose computers requiring special hardware to translate between binary and decimal numbering systems.**

***Index Terms*—Algorithms, arithmetic, conversion, data processing, logic.**

INTRODUCTION

There has been a trend in the design of computer processors to include special hardware for translating between binary (used internally by the computer in calculations) and decimal (used by the programmer on the outside). Translation between binary and BCD number systems is also of value in controlling the growing variety of display devices.

In the past, the conversion task has been handled almost exclusively by programmed subroutines. This is usually satisfactory, as little total computer time is spent doing such conversions. However, as new application areas open up for computer systems, it is conceivable that greater speed in conversion will be required. Considering such advances in the state of the art, two important trends in technology should be taken into account. Greater use of micrologic circuits will make it more economical to incorporate additional hardware for special functions. Also, new automated design techniques will take advantage of these circuits, along with read-only memories, to provide more customized systems. Such techniques offer an endless variety of machines differing in speed and instruction set, but all from the same production line.

This note describes algorithms designed to obtain fast, efficient conversion from binary to BCD number systems, as well as additional means for improving the entire conversion process. The algorithms first require scaling the binary numbers to fractions and consist of iteratively multiplying the fractions by some power of ten, particularly by one thousand in the fastest method, and re-encoding the overflow from each iteration. The multiplication is accomplished by shifting and adding in one pass through the adder.

A scheme for precisely converting fixed-point integers by these methods is also described. It consists of compensating by hardware means for errors introduced by scaling and truncating.

Use of these algorithms requires no more hardware than Couleur's[1] method of doubling, used currently in IBM System/360, and can offer up to ten times improvement in speed for floating-point output. Furthermore, since the usual method of converting from decimal to binary also requires iterative multiplication by ten, some economy is achieved by the use of common hardware.

Special instructions should be designed with the overall task in mind. Hardware to speed only the conversion is not worthwhile if excessive computing time is taken with scaling, unpacking, and other editing functions. One should consider what portion of the total computer time is devoted to this task, and how much improvement can be gained. An important factor is the I/O equipment in the system and how it will be used. For example, for a machine used mostly as a peripheral processor, the complete conversion need only be fast enough to keep up with the printers attached to it.

BINARY-TO-DECIMAL CONVERSION THEORY

Multiplication by Ten

The familiar method of converting binary fractions to decimal consists of repeated multiplication by ten. After each iteration, the integer portion of the product is removed and stored as the next-lower-order digit of the decimal fraction. To demonstrate, a binary fraction has the equivalent decimal representation $0.\ d_1d_2d_3$, where d_i are integers and $0 \leq d_i \leq 9$. Then we may write

$$f_1 = \frac{d_1}{10^1} + \frac{d_2}{10^2} + \frac{d_3}{10^3} + \cdots .$$

To find d_1, multiply by ten:

$$10 \cdot f_1 = d_1 + \frac{d_2}{10^1} + \frac{d_3}{10^2} + \cdots = d_1 + f_2.$$

Since f_2 is a proper fraction, $(0 \leq f_2 < 1)$, d_1 is the integer portion and is shifted off to become the first decimal digit. The next digit d_2 is obtained by multiplying f_2 by ten, and so on, until the desired number of digits is obtained.

Multiplication by ten can be effected by appropriate shifting and a single addition, since $10f = 8f + 2f$. By proper positioning of f at the adder inputs, the multiplication can be executed in one operation. Therefore, if gating circuits are provided which shift the fraction as it is transmitted to the adder inputs, the overall speed is determined by the number of additions required. Since each multiplication by ten results in a new decimal digit, one digit per operation is developed.

The result should be compared with Couleur's method which requires one operation per bit. That is, if shifting and adding can be performed in one operation, then the binary number can be shifted left one bit (doubled) every operation. Since a decimal digit is formed for every $\log_2 10 = 3.32$ bits of the binary quantity, the method described above inherently affords a more than three times improvement over doubling.

Multiplication by One Thousand

Another significant speed improvement can be obtained by first converting to radix one thousand:

$$f_1 = \frac{k_1}{10^3} + \frac{k_2}{10^6} + \frac{k_3}{10^9} + \cdots ,$$

$$1000 \cdot f_1 = k_1 + f_4 \quad \text{where} \quad 0 \leq k_i \leq 999.$$

The fractional portion has been denoted f_4 since it is equal to the fraction which would be obtained after three iterations of multiplication by ten.

Manuscript received June 30, 1966; revised December 28, 1967.
The author is with IBM Corporation, Systems Development Division, Poughkeepsie, N. Y. 12602

[1] J. F. Couleur, "BIDEC: A binary-to-decimal or decimal-to-binary converter," *IRE Trans. Electronic Computers*, vol. EC-7, pp. 313–316, December 1958.

The integers k_i can be converted to decimal using a decoder, or other means, on subsequent cycles. However, this decoding can be done concurrently with multiplication of the fractions which follow by one thousand. It will be shown later that the decoding can be done without excessive circuitry, in two operation times. The decoding, therefore, adds only two cycles after the last k_i is obtained.

Multiplication by one thousand required two additions per k-digit since

$$1000 \cdot f = 2^{10} \cdot f - 2^5 \cdot f + 2^3 \cdot f,$$

or alternately,

$$1000 \cdot f = 2^{10} \cdot f - 2^4 \cdot f - 2^3 \cdot$$

The term *addition* (as used here) includes the process of subtraction as well. Since two additions are required for every three decimal digits ultimately developed, this method is one and a half times as fast as multiplication by ten. However, since all the operands can be available at the same time, one can perform the two additions in one operation time by using a carry-save-adder[2] in front of the adder. Thus, three digits can be obtained every operation, which is a three-fold improvement over multiplication by ten, and a nearly ten-fold improvement over doubling.

Multiplication by One Hundred

For completeness, a compromise of the above schemes should be mentioned. This uses an intermediate radix of one hundred. Noting that $100f = 2^6 \cdot f + 2^5 \cdot f + 2^2 \cdot f$, one sees that two additions are needed to develop two decimal digits. If no carry-save-adder is available, this method is no better than multiplication by ten (and is actually inferior, due to the greater complexity). However, with a carry-save-adder, a two-fold improvement is obtainable. The decoder for this method requires only about one-third the circuitry as that needed for multiplication by one thousand, and can be done in one operation time. This method has merit if one wishes to reduce the decoding circuitry, or if the word length is fairly short.

Binary-to-Decimal Converters

Typical arrangements for carrying out the algorithms already described are illustrated here. The principles involved in the method of multiplication by ten are basic to each of the methods. Fig. 1 shows a block diagram for a converter using this method. The two-times and eigh -times multiples of the binary fraction are added each cycle. Simultaneously, the integer register is shifted one decimal digit to the left. The integer portion of the adder output is inserted into the right-hand digit position of this register. The fraction portion of the adder output is returned to the fraction register. The numerical example in Fig. 2 demonstrates this operation.

Fig. 3 shows an arrangement using multiplication by one thousand. The required multiples are added each cycle, with the integer portion shifted into an over-flow register. The integer portion can be any binary integer up to 999. Decoders convert it first to radix 100 and then to radix 10 on subsequent cycles. Note the parallelism allowing multiplication while the overflows from the two previous cycles are both partly converted.

Integer Conversion

Since the conversion method presented here converts only binary fractions, some consideration must be given to conversion of integers. Suppose a machine permits integers having a maximum of six digits, and the binary equivalent of 857 is to be converted. The correct decimal digits would be produced if the number is first scaled down to the equivalent of 0.000857. This may be accomplished by either dividing by the binary equivalent of 10^6, or multiplying by 10^{-6}. However, the fraction 0.000857 cannot be expressed exactly in the binary number system. Therefore, some error is introduced by the scaling process. Floating-point numbers must also be scaled, but this

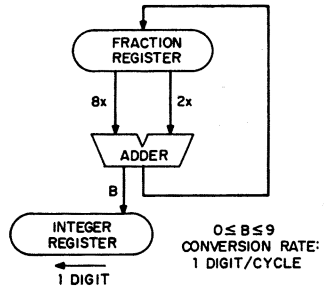

Fig. 1. Binary to BCD, 1 digit per cycle.

Integer Register (BCD)				Fraction Register (Binary)	Operation
				0.1011	0.6875
				1.011	Times 2
				101.1	Times 8
				110.111	Add
			0110	0.111	Shift integer
				1.11	Times 2
				111.0	Times 8
			0110	1000.11	Add
		0110	1000	0.11	Shift integer
				1.1	Times 2
				110.0	Times 8
		0110	1000	111.1	Add
	0110	1000	0111	0.1	Shift integer
				1.0	Times 2
				100.0	Times 8
	0110	1000	0111	101.0	Add
0110	1000	0111	0101	0.0	Shift integer
6	8	7	5		

Fig. 2. Binary to BCD, 1 digit per cycle (numerical example).

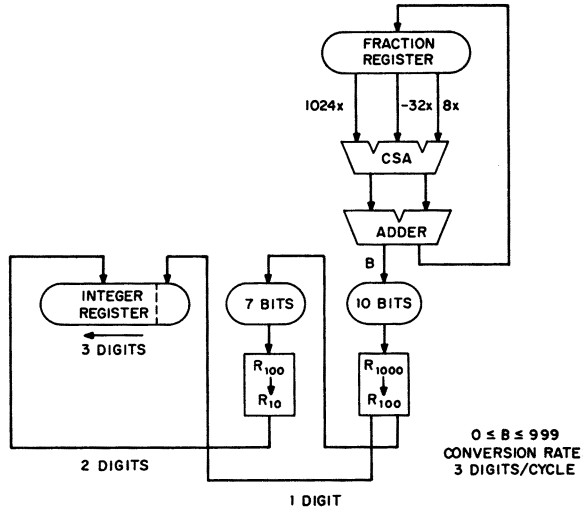

Fig. 3. Binary to BCD, 3 digits per cycle.

is more complicated, since the scaling factor must take the exponent into account. Floating-point results are usually considered precise only to a limited number of digits. Fixed-point results, however, are often considered exact integers. It would not do, for instance, if 10 000 were converted to 9999. For this reason special care is needed to compensate for small errors introduced in the scaling and conversion processes.

Errors due to scaling and conversion come about in several ways. Suppose a hypothetical machine has a fixed-point word of 12 bits, not including sign. Since the largest integer it can contain is 4095, four digits should be developed by the conversion instruction.

We choose to do scaling by multiplying by 10^{-4}. Since we are using fixed-point multiplication, we must actually multiply by the integer part of $10^{-4} \cdot 2^{24}$, which develops a double-word-length integer (24 bits), and then shift the binary point (figuratively) to the left of

[2] O. L. MacSorley, "High-speed arithmetic in binary computers," *Proc. IRE*, vol. 49, pp. 67–91, January 1961.

this double word. The value of $10^{-4} \cdot 2^{24}$ is 1677.7216. Truncating the fraction results in an error, after multiplication and conversion, of

$$E_s = -0.7216 \cdot 2^{-24} \cdot 10^4 \cdot I$$
$$= -4.301 \cdot 10^{-4} \cdot I$$

where I is the integer being converted, and E_s is the *scaling error*. The maximum scaling error is found for I equal to 4095:

$$E_{s(\max)} = -1.76.$$

A variable error whose magnitude can be greater than unity cannot be tolerated. In this example, scaling was done by multiplying by 1677. The error would be smaller if 1678 were used. Then

$$E_s = +0.2784 \cdot 2^{-24} \cdot 10^4 \cdot I$$
$$= +1.66 \cdot 10^{-4} \cdot I$$

and $E_{s(\max)} = +0.68$, which is tolerable. In this case we are fortunate. For other word lengths, both values of E_s might be greater than unity. If we had elected to scale by dividing by 10^4, a scaling error would occur when the remainder was dropped.

Another source of error results from truncating the product after scaling. The maximum error due to this source is 10^4 times the portion lost in truncation. If the product were truncated to 12 bits (one word), the error would have the range

$$0 \geq E_t > -10^4 \cdot 2^{-12} = -2.44.$$

However, by carrying 16 bits of the product, this source of error is limited to

$$E_{t(\max)} = -0.15.$$

A third source of error results if the adder which is used for multiplication by ten is not also extended this extra four bits. The 4 bits are merely shifted left three places for one iteration, which is equivalent to multiplication by eight instead of ten. For a 12-bit adder, the range of error is

$$0 \geq E_a > -10^4 \cdot 2^{-12} \cdot (0.2) = -0.488.$$

By extending the adder 4 additional bits, this error is eliminated. Summarizing the errors,

$$E_s = 0 \text{ to } +0.68, \quad E_t = 0 \text{ to } -0.15 \quad E_a = 0.$$

Total error

$$E = -0.15 \text{ to } +0.68.$$

Care has been taken to limit the total error range to less than unity, in this instance 0.83. The correct converted integer can be obtained by adding a constant after scaling such that the final error would be in the range $0 \leq E_f < 1.0$. Here we find that if the binary quantity 0010 is added to the low-order 4 bits of the 16-bit fraction, a correction can be obtained equal to $E_c = 10^4 \cdot 2^{-15} \cdot (0.8) = +0.24$. The corrected final error is

$$E_f = E + E_c, \quad +0.09 < E_f < +0.92.$$

Since only a four-digit integer is retained by the conversion, this error does not affect the final result.

The addition required for correcting the integer might best be done as part of the conversion instruction.

Binary-to-Decimal Double-Precision Floating Point

For the purpose of illustrating a point, assume the exponent to be zero. Assume also that the conversion instruction develops only four digits but, for double precision, eight are needed. First, using double-precision multiplication, the binary fraction is multiplied by 10^4. The higher-order word of the product is an integer, which is then scaled separately and converted to decimal. The low-order word of the double-word product is converted directly without further scaling.

Improving the Overall Problem

Having a fast instruction to do conversion is not worthwhile if corresponding improvements are not made in scaling, rounding, unpacking, leading zero suppression, and other editing chores. Suppose separate instructions are available for converting from binary integers (scaled) to decimal and from binary floating point to decimal. The integer conversion instruction could automatically add the correction factor and suppress leading zeros. The fraction conversion instruction could mask the exponent field and add a rounding factor to the fraction at the beginning of the instruction. If n digits are developed, the rounding factor would be $0.5 \cdot 10^{-n}$ in floating-point binary. Both instructions could also do unpacking. This might be executed by shifting the decimal digits 8 bits instead of 4 bits during conversion, and stuffing the extra 4 bits with the proper prefix. Width control can also be done automatically by both instructions by loading the number of digits needed into an index register.

When converting floating point from binary to decimal, a very significant amount of time is required for handling the exponent. Suppose the binary number is $2^b \cdot f$, where f is a proper fraction and b is a signed exponent. The corresponding decimal representation is $10^t \cdot d$, where $d \leq f$.

$$2^b \cdot f = 10^t \cdot d$$

$$b \cdot \log_{10} 2 + \log_{10} f = t + \log_{10} d$$

$$t = b \cdot \log_{10} 2 + \log_{10} (f/d).$$

One chooses, as t, the smallest integer such that $t \geq b \cdot \log_{10} 2$. This may be written as $t = [b \cdot \log_{10} 2]^+$. Once t is determined, the decimal fraction is calculated as

$$d = 2^b \cdot f / 10^t.$$

The quantity 10^t is not calculated, but is obtained from a table stored in memory. The following outline shows a typical procedure for the steps indicated.

1) Determine decimal exponent.
   a) Right shift until exponent is right adjusted.
   b) If zero, set decimal exponent to zero, and terminate; if not zero, proceed.
   c) Multiply by log 2 (fixed-point multiplication).
   d) Add 1.
   e) Store integer portion of result into index register (this is decimal exponent).
2) Determine decimal fraction.
   a) Load floating-point number.
   b) Divide by 10^t (obtained from table, using indexed address).

If the exponent field has 8 bits, the table of exponents contains 78 entries. A machine with a large memory could afford a larger table which would speed up the procedure. The table would have 256 entries, one for each binary exponent. Each entry might contain both the decimal exponent t and the factor 10^{-t}. The decimal fraction could be obtained by multiplying the binary fraction by 10^{-t}, since multiplication is presumed to be faster than division. To facilitate the procedure, a special instruction can be provided which loads an index register directly from the exponent field.

Summary

This note has described ways to speed up number conversion. Other types of conversion, such as between fixed-point and floating-point numbers, can also be considered. Some studies should be made, in various computing environments, to determine if extra hardware for conversion is justified. If it is, careful planning is required to get the most overall improvement without too much loss in flexibility. With a modest amount of circuitry, one should expect an order of magnitude improvement over present methods.