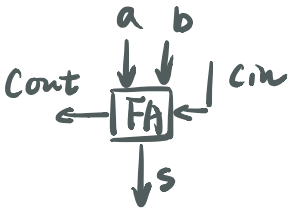
Lab 2: Multipliers



Introduction

In this lab you will build two multiplier circuits and then simulate them using *ModelSim* software with your own testbench. Please read the accompanying documents *Testbench Tutorial* and `Creating Generic Hardware`.

Part I: Carry Save Multiplier

A simple multiplier performs multiplication in hardware in a manner similar to performing it by hand. This type of multiplier is called an *Array Multiplier*. However, array multipliers are quite slow. Faster multipliers can compute a product of two numbers more quickly. For Part I you will design an **unsigned** 8x8 *Carry Save Multiplier* (CSM). Figure 1 shows an example of 4-bit carry save multiplier architecture.

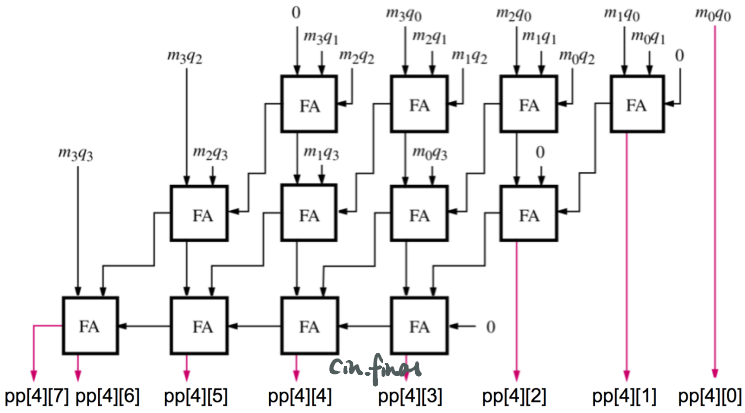
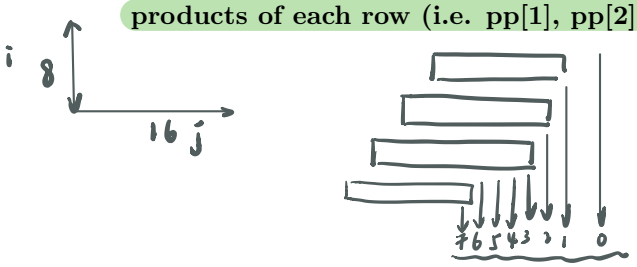


Figure 1: 4-bit Multiplier Carry Save Array (Figure from pg. 354, "Computer Organization and Embedded Systems", 6th ed., Hamacher, Vranesic, Zaky and Manjikian)

Figure 1 shows the multiplication of the quotient q0-q3 by the multiplicand m0-m3 to produce the 8-bit result, p0-p7. This 4x4 multiplier uses 3 rows of full-adders (FA) while your multiplier should consist of 7 rows of 8 full-adders each. The input to and output from each adder is refereed to as a 'partial product (pp)'. For the 4x4 multiplier shown in Figure 1, the final output is given as pp[4][7:0].

Structural coding, by instantiating several rows, is encouraged. You may find the `generate` statement helpful for instantiating many copies of your sub-modules. See the accompanying document *Creating Generic Hardware* for a tutorial.

You must not use the * and + Verilog operators in the design of your circuit. To ensure this, the values of each row of your multiplier will be declared as outputs. **The automarker will test the partial products of each row (i.e. pp[1], pp[2], etc.).**



The Testbench

After you've created your 8x8 unsigned CSM module, the next thing you should do is to simulate it to make sure it is correct. For this, you will write a testbench to test every possible combination of inputs (all 65536 of them), and compare the result of your multiplier to the value returned by the Verilog multiplication operator *. Please see the accompanying document *Testbench Tutorial*, which describes a very similar testbench to the one required for this lab, except for testing an adder rather than a multiplier.
If a mismatch occurs between the expected product and your module's output, you should print a message to the ModelSim console using the `$display` command indicating which test case failed. This will help you locate the bug. If all test cases pass, print a message indicating this as well. Run your testbench in ModelSim to verify that your multiplier is correct. If the testbench finds errors in some test cases, debug and fix your multiplier until all results pass. Don't forget that you can set the radix of the displayed signals to *Decimal* in the wave window.

Part II: Wallace Tree Multiplier

In part II you must design an even faster multiplier than the CSM, namely the Wallace Tree Multiplier (WTM). The WTM takes uses the fact that the full-adder blocks used in multipliers have 3 inputs (namely A, B and Cin), but only two numbers are added in most multiplier designs. This results in the WSM having a smaller delay, particularly for larger multipliers.

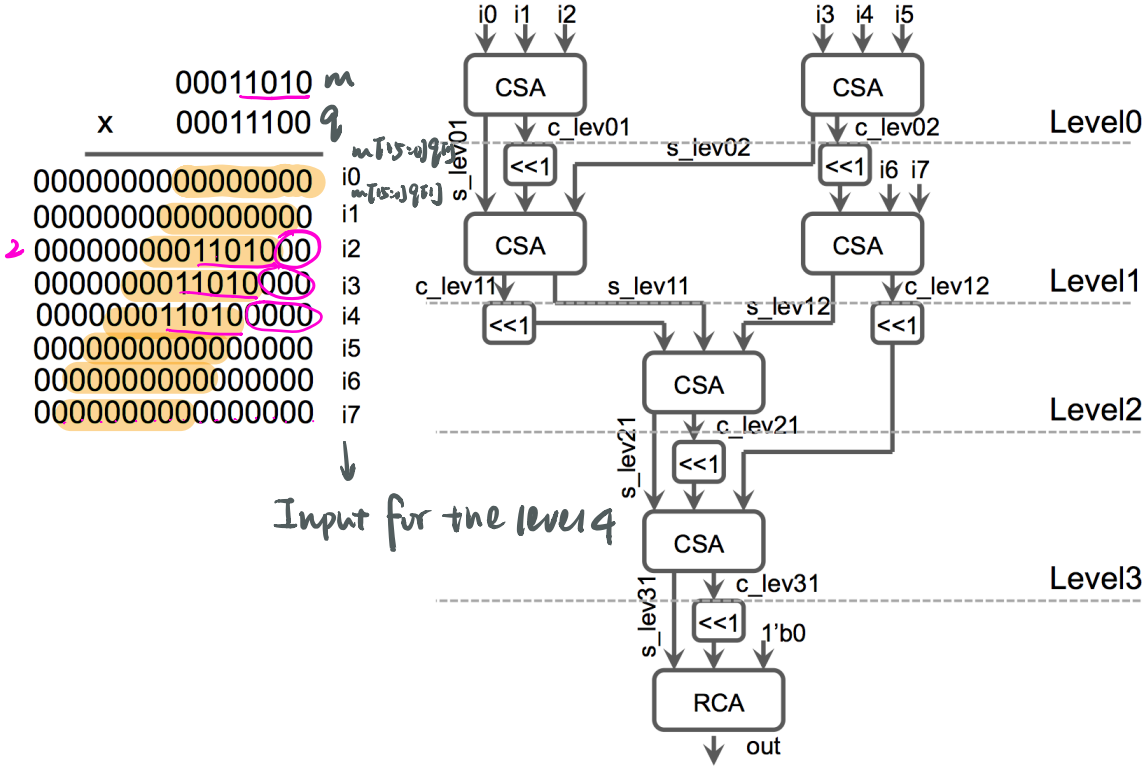


Figure 2: 8-bit Wallace Tree Multiplier Architecture

Figure 2 shows the block diagram of a 8x8 WTM. You must design an **unsigned** 8x8 WTM for Part II. The implementation is based on a pipeline of carry save adders (CSA) followed by a ripple carry adder (RCA) as the final stage. The inputs to the CSAs at each level are also shifted left by 1 bit before being input. The

calculation on the left shows the values for i0 - i7, which are the inputs to the CSAs in level 0 and level 1. To help you, the provided lab kit includes parameterized code for the CSA and RCA. Once again, to ensure you are performing all the steps correctly, The automarker will test the carry and save outputs of Level2 and Level3.

Please be warned that the WTM is a conceptually difficult multiplier so take your time to understand the design clearly before attempting to implement it. You are encouraged to learn more about the WTM from online resources as well.

You must repeat the steps provided in Part I to implement a WSM, test it using ModelSim using your own testbench. **Once again, you must not use the * and + Verilog operators in the design of your circuit.**

Preparation

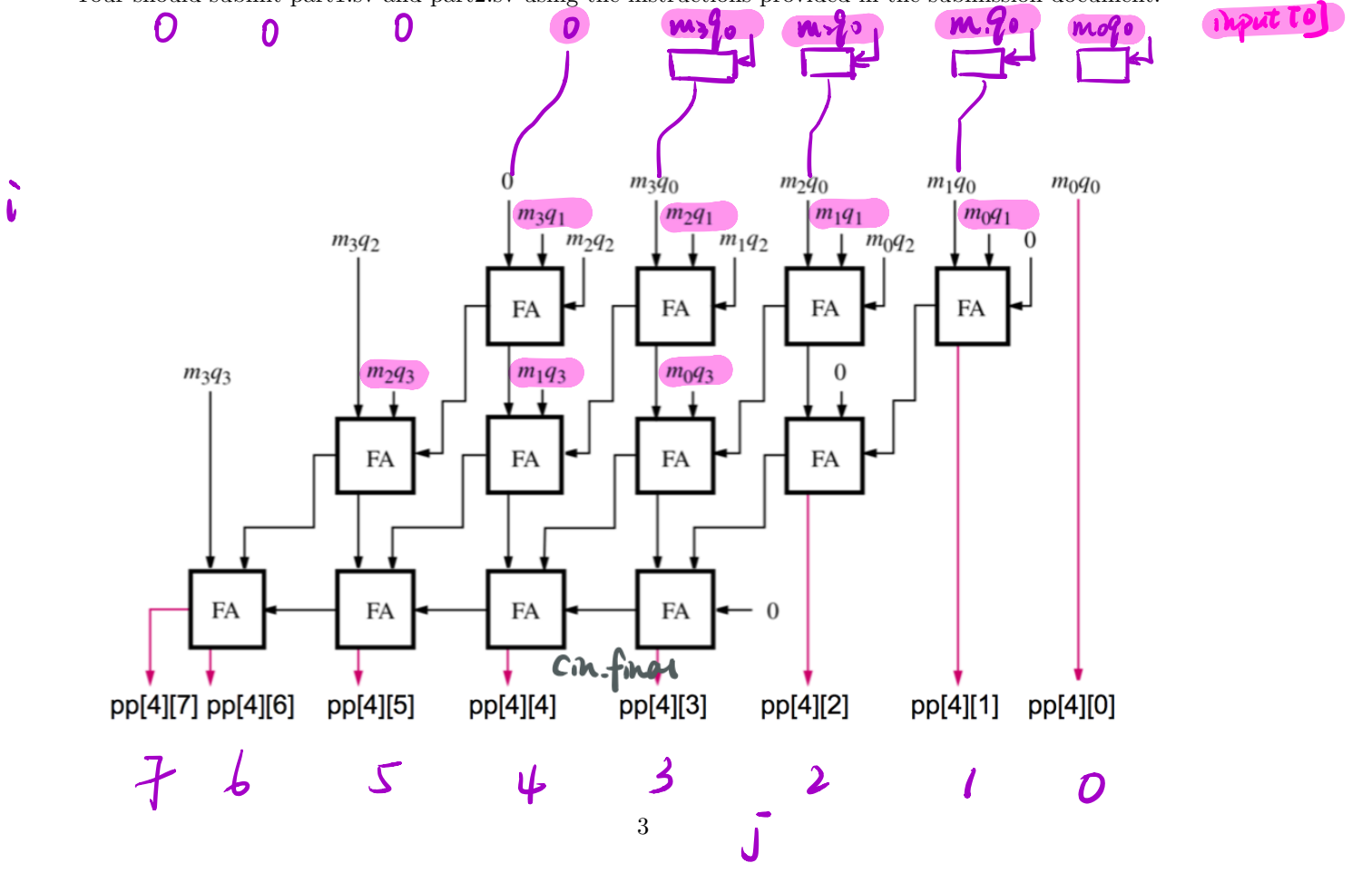
- Familiarize yourself with the Generate statement by reading the `Creating Generic Hardware` tutorial.
- Download the starter kit for this lab. Separate files are provided for parts 1 and 2.
- Name the finished code part1.sv and part2.sv respectively.

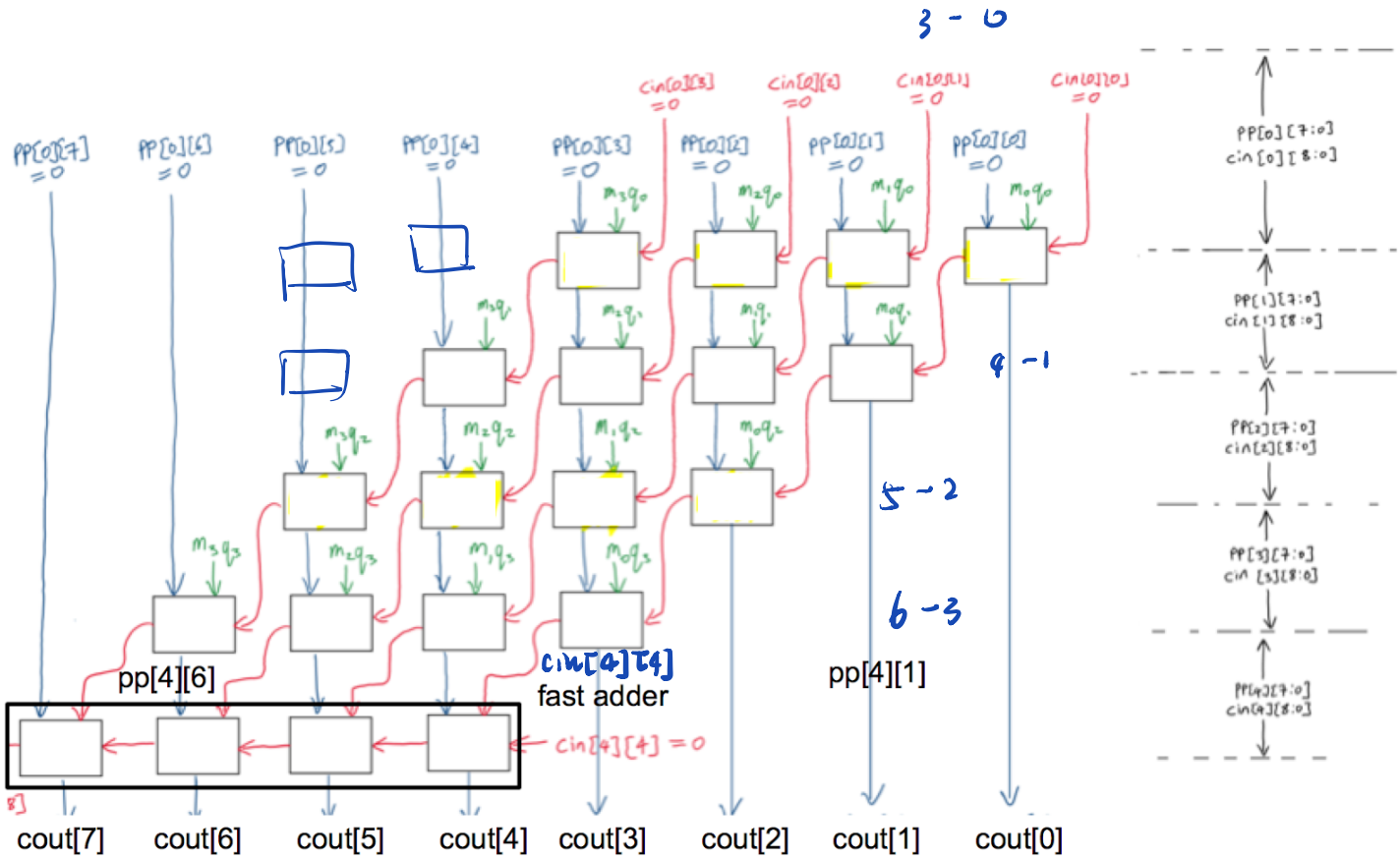
In-Lab

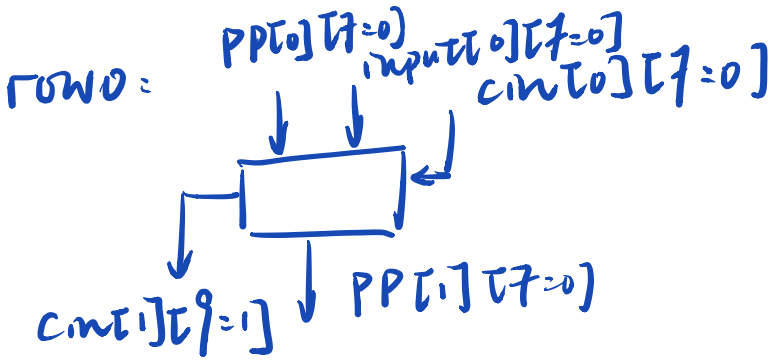
Go through the provided tester for both part1.sv and part2.sv to make sure that your code is able to be recognized by the automarker

Submission

Your should submit part1.sv and part2.sv using the instructions provided in the submission document.








```
    0 0 0 1   m
  x 0 1 0 1   q
  -----------
    0 0 0 1     m[3:0]q[0]
  0 0 0 0       m[3:0]q[1]
0 0 0 1
0 0 0 0
-----------
0 0 0 0 1 0 1
```

A Suggestion for a Fast Multiplier*

C. S. WALLACE†

Summary—It is suggested that the economics of present large-scale scientific computers could benefit from a greater investment in hardware to mechanize multiplication and division than is now common. As a move in this direction, a design is developed for a multiplier which generates the product of two numbers using purely combinational logic, *i.e.*, in one gating step. Using straightforward diode-transistor logic, it appears presently possible to obtain products in under 1 μsec, and quotients in 3 μsec. A rapid square-root process is also outlined. Approximate component counts are given for the proposed design, and it is found that the cost of the unit would be about 10 per cent of the cost of a modern large-scale computer.

Introduction

A CONTEMPORARY computer spends a large percentage of its time executing multiplication, and to a lesser extent, division. The recent advent in very large machines of "bookkeeping" controls (operating in advance of the arithmetic unit to execute memory fetches, stores and address modification, etc.) has tended to increase this percentage by relieving the arithmetic unit of many trivial burdens. The arithmetic unit of such a machine, when used for scientific computations, will spend nearly half its time multiplying or dividing. Paradoxically, the amount of hardware built into large machines specifically for these operations is rarely very great. Thus the situation has arisen, viewed in the context of a very large machine involving a heavy investment in memory, peripheral equipment and controls, that it may be advantageous to the economy of the machine as a whole to increase the hardware investment in the operations of multiplication and division, even beyond the point where an increment of this investment yields an equal incremental increase in multiplication-division speeds. This paper will describe a type of multiplication-division unit designed primarily for high speed, and will discuss its economics.

Lines of Approach

Multiplication of binary fractions is normally implemented as the addition of a number of summands, each some simple multiple of the multiplicand, chosen from a limited set of available multiples on the basis of one or more multiplier digits. The author can see no good reason to depart from this general scheme. Acceleration of the process must then be based on one or more of the following expedients:

1) Reduction in the number of summands;
2) Acceleration of the formation of summands;
3) Acceleration of the addition of summands.

The last will be discussed first.

Addition

The basic addition processes usually employed in computers add two numbers together. The possibility exists of adding together more than two numbers in a single adder to produce a single sum. However, the logical complexity of the adder required appears to grow quite disproportionately to the resulting increase in speed, and there appears to be no advantage in trying to sum even three numbers at a time into a single sum.

An expedient now quite commonly used [3]–[5] is to employ a pseudoadder which adds together three numbers, but rather than producing a single sum, produces two numbers whose sum equals that of the original three. In the context of the basic problem of adding together many summands, one pass through such an adder reduces the number of summands left to be summed by one, as does a pass through a conventional adder. The advantage of the pseudoadder is that it can operate without carry propagation along its digital stages and hence is much faster than the conventional adder. A simple form for such an adder is a string of full adder circuits of the normal sort, where the carry inputs are used for the third input number, and the carry outputs for the second output number. This and other possible forms are discussed by Robertson [3]. In multiplication, one pseudoadder is usually used, and storage is provided for two numbers. On each pass through the adder, the two stored numbers and one multiple of the multiplicand are added, and the resulting two numbers returned to storage.

Thus, the time required varies linearly with the number of summands. In any scheme employing pseudoadders, the number of adder passes occurring in a multiplication before the product is reduced to the sum of two numbers, will be two less than the number of summands, since each pass through an adder converts three numbers to two, reducing the count of numbers by one. To improve the speed of the multiplication, one must arrange many of these passes to occur simultaneously by providing several pseudoadders.

Assuming that all summands are generated simultaneously, the best possible first step is to group the summands into threes, and introduce each group into its own pseudoadder, thus reducing the count of numbers by a factor of 1.5 (or a little less, if the number of summands is not a multiple of three). The best possible second step is to group the numbers resulting from the

* Received June 18, 1963. This work was supported in part by the Atomic Energy Commission under Contract No. AT (11-1)-415, and was done while the author was temporarily on the staff of the University of Illinois, Urbana.

† Basser Computing Department, School of Physics, University of Sydney, Sydney, N.S.W., Australia.

first step into threes and again add each group in its own pseudoadder. By continuing such steps until only two numbers remain, the addition is completed in a time proportional to the logarithm of the number of summands.

Successive steps may use the same set of pseudoadders (using progressively fewer of the set in each step) by using temporary storage registers for the outputs of the pseudoadders. However, for current transistor-diode circuitry, there are strong arguments for using separate pseudoadders for each step, without storage for intermediate resuts. The equipment cost is little if at all increased, since the additional pseudoadders required will not need many more components than the flip-flop registers eliminated, and the control circuitry is greatly simplified. Although a quite simple three-transistor pseudoadder stage can be designed with present components to give a delay time of about 60 nsec, problems of distribution of gating signals and flip-flop recovery time would make it very difficult to make successive passes through the same pseudoadders more often than about once per 150 nsec. Thus the purely combinational adder would have a considerable speed advantage. As an example of the arrangement resulting from these considerations, Fig. 1 shows a set of 18 pseudoadders connected to take 20 summands

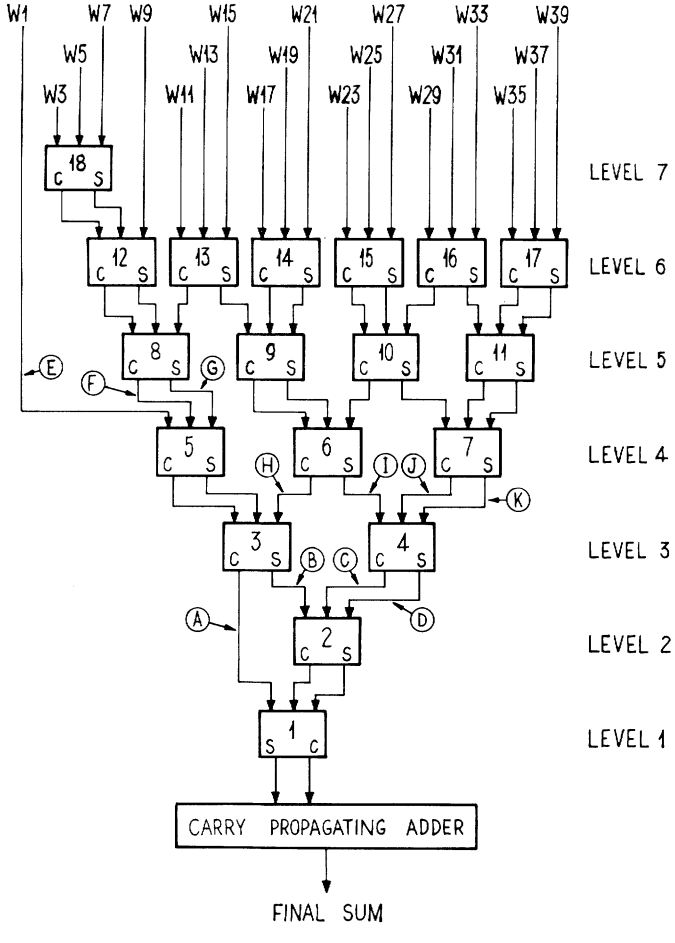


Fig. 1—The adder tree.

(W_1 to W_{39}) and express their sum as the sum of two numbers. These are shown added in a conventional carry-propagating adder to produce the final product. The design of this adder will not be discussed, as several excellent designs having carry-propagation times of the order of 100 nsec have been devised [1], [5].

Generation of Summands

In the simplest form of multiplication, there are as many summands as multiplier digits, each either 0 or 1 times the multiplicand. A wide range of schemes involving recoding the multiplier into a new (possibly redundant) form using some negative digits have been developed to reduce the number of summands [3]. Since all summands are to be generated simultaneously, and then summed very quickly, it is desirable that the recoding scheme used should 1) require only multiples of the multiplicand obtainable by shifting and complementing, and 2) be a local recoding in which each recoded digit depends only on a small group of original multiplier digits. The best system found gives base-four recoded multiplier digits which can be $+2$, $+1$, 0, -1 or -2, and each is determined entirely by three adjacent original binary multiplier digits. Considering the process as a base-four recoding, digits 0, 1, 2, 3 are recoded into digits 0, 1, -2, -1, respectively, if the next less significant original base-four digit is 0 or 1, and into 1, 2, -1, 0 if the next less significant original digit is 2 or 3. The number of summands is half the number of binary multiplier digits. Attempts to reduce the number further appear to require multiples not obtainable by shifting. Some complications arise in the pseudoadder structure because of the negative multiplier digits, which, in a two's complement system, require correction digits to be added in. However, detailed examination shows that these problems are superable without loss of speed or undue circuit cost.

Division

Most normal division processes are essentially serial, and hence, not well suited to employ efficiently the highly parallel multiplier. The best approach seems to be to generate reciprocals by an iterative process of multiplications. The following algorithm is a version of one first described by Wheeler [2]. Given a positive normalized fraction x, and some approximation p to $1/x$, set $a_1 = px$ and $b_1 = p$ and iterate

$$a_{n+1} = a_n(2 - a_n), \qquad b_{n+1} = b_n(2 - a_n).$$

This process converges quadratically, a_n to 1, and b_n to $1/x$. Simple logic which inspects the first six digits of x can be used to generate a p of the form $1 \cdot q\ r\ s\ t$ such that $|1 - px| \leq 1/32$, *i.e.*, a_1 has the form $\bar{d} \cdot dddddefghjk$, etc. p can be recoded to give three summands. The first iterative step should increase to 10 the number of similar digits immediately after the binary point. It can be shown that an a_2 of this form is obtainable by use of a

multiplier which is not exactly $2-a_1$, but rather

$$1 - 2^{-5}(\bar{d}\cdot efghj),$$

where the number in brackets is regarded as a signed two's complement fraction. This multiplier can be obtained directly from a_1, and requires only four summands after recoding. Similarly, an approximate multiplier can be used in the next iterative step requiring seven summands, and one in the third step requiring 12 summands. Only three iterative steps are needed to produce a 40-bit reciprocal. If these approximate multipliers are used in both multiplications of the iterative step, the correct answer is produced. The advantage of using the approximate multipliers is twofold. Firstly, the smaller number of recoded multiplier digits allows the multiplication to be done by only part of the pseudoadder tree, and the multiplication time, especially in the earlier steps, is thereby reduced. (In Fig. 1, summands for px and the first iterative step can be introduced at points A, B, C and D. and those for the second step at points E to K.) Secondly, the number of digits in b_n is small in the early steps. This fact, together with the fact that the more significant digits of the a's need not be formed, means that, for a 40-bit word length, both multiplications in each of the first and second iterative steps can be performed simultaneously by splitting the pseudoadders into two shorter-word-length sections. Depending on the word length, some extension of some of the pseudoadders may be necessary. The a-multiplication in the last iterative step need not be done. Thus, a 40-bit reciprocal can be generated with only four passes through the multiplier, and at least the first three of these can be quicker than a full multiplication.

Square Root

A variant of the reciprocal iteration can be made to yield reciprocal roots. Given x positive and normalized, and p an approximation to root $1/x$, set $a_1=p^2x$, $b_1=p$ and iterate

$$a_{n+1} = a_n(1\tfrac{1}{2} - a_n)^2; \qquad b_{n+1} = b_n(1\tfrac{1}{2} - a_n),$$

where b_n converges quadratically to root $1/x$. Once again, approximate multipliers can be used. The possibility of doing two multiplications simultaneously has not been investigated in detail. However, even if this is not feasible, the process would appear to be quicker than the customary Newton method using repeated divisions.

Speed and Cost

A detailed examination has been made of a design for a multiplication-division unit for 40-bit numbers. The design is based on saturating complementary diode-transistor AND-OR-NOT circuits. Each pseudoadder stage requires three transistors and 18 diodes, and involves two stages of logic. Multiplier recoding requires one stage. A pnp adder is shown in Fig. 2.

To estimate speeds, it is assumed that:

1) Each stage of logic introduces a propagation delay of 30 nsec.
2) The delay of high-current drivers for fanout of multiplicand and recoded multiplier digits is 100 nsec.
3) The settling time of the carry-propagating adder is 100 nsec.
4) The result will be gated into a register with a settling time of 100 nsec.

On these assumptions, which are believed to be realistic for the present fairly cheap components in good packaging, the multiplication time becomes 750 nsec.

The reciprocal-generating time, excluding prenormalization, is 2220 nsec. The time for a complete division is therefore about 3 μsec. The time for generating a reciprocal root, assuming that no multiplications are done simultaneously, is 6 μsec.

The tree of pseudoadders requires 750 full-adder circuits. Generation of summands requires 840 single-transistor logic stages for multiplication, and a further

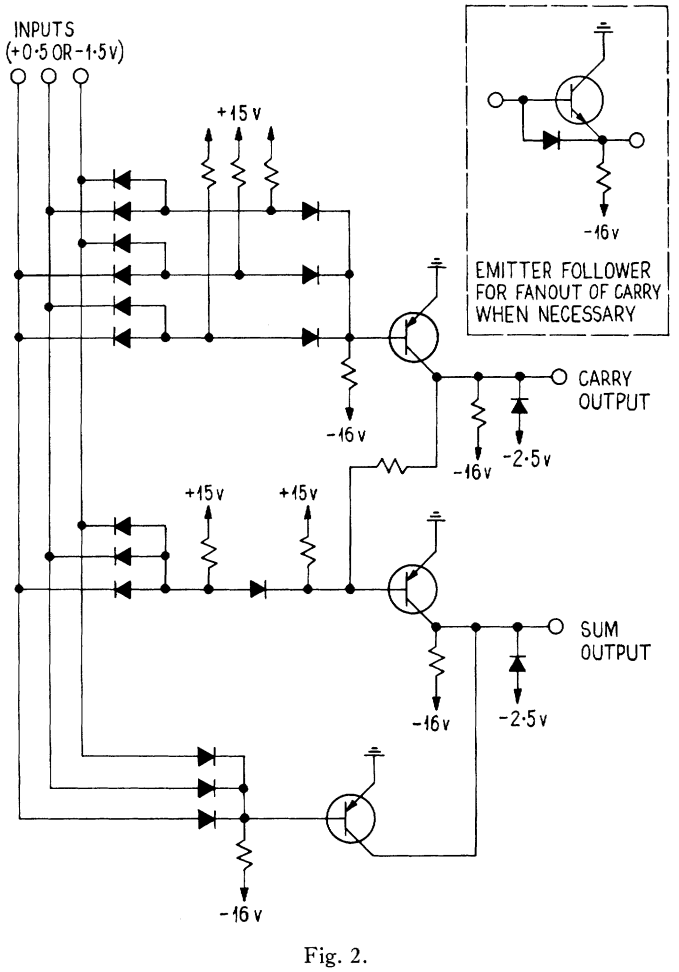


Fig. 2.

561 stages for division if division summands are introduced in later stages of the adder tree to save time. A total of 140 high-current drivers are needed. The total semiconductor cost, excluding the carry-propagating adder and operand and result registers, which would be present in a conventional arithmetic unit, is 4591 transistors, and 33,083 diodes.

Discussion

The multiplier unit requires a great deal of equipment, amounting perhaps to 10 per cent of the total semiconductor complement of a very large modern computer, but probably, because of its simplicity, costing rather less than 10 per cent of the cost of the computer. In a sense, this equipment is used inefficiently. It is useful for only some arithmetic operations, and even in these, circuits with delay times of 30 nsec are used only about once per microsecond. However, some mismatch between propagation delay and repetition rate is apparently inherent in the type of circuit postulated, and equally bad mismatches could probably be found in many present computers. If the word length is increased, the equipment cost rises as the square of the word length, and the times as the logarithm of the word length. The inefficiency, or ratio between propagation rate and repetition rate, rises logarithmically. However, for 40-bit words, the inefficiency is not intolerable.

The speeds achievable appear to be greater by a factor of at least four than those obtained in conventional units. Multiplication and division times would be reduced to approximate parity with the time required for, *e.g.*, floating point addition. Parity in the times for all arithmetic operations would be of considerable benefit to machines employing advanced controls, as at present these advanced controls tend to run out of work when long arithmetic operations are performed, unless many levels of look-ahead buffering are provided, in which case the control organization, especially in the treatment of jumps, becomes very complicated and difficult to design. Machines without look-ahead buffering between memory and arithmetic units should benefit even more. At present, such machines tend to be designed with memories and controls which can keep up with the faster arithmetic operations, and which therefore lie idle during multiplications and divisions. If one can assume that, *e.g.*, a third of all arithmetic operations in scientific computers are multiplications, and that these at present take about four times as long as additions, etc., the use of the fast unit would approximately double the speed of computation.

A simpler unit might be preferable in some cases, in which half as many adders are used to perform a multiplication in two steps. The equipment cost would be almost halved, and the multiplication time almost doubled. Reciprocal times would not be affected, as the half-sized adder tree would still be large enough to do in one step the largest multiplications required in the iteration.

References

[1] D. B. G. Edwards, "A parallel arithmetic unit using a saturated transistor fast carry circuit," *Proc. IEE*, vol. 107, pt. B, p. 673; November, 1960

[2] M. V. Wilkes, D. J. Wheeler, and S. Gill, "Preparation of Programs for an Electronic Digital Computer," Addison-Wesley, Cambridge, Mass.; 1951.

[3] J. E. Robertson, "Theory of Computer Arithmetic Employed in the Design of the New Computer at the University of Illinois," presented at Conf. on Theory of Computing Machine Design, University of Michigan, Ann Arbor, June, 1960.

[4] "Whirlwind I Computer Block Diagrams," Project Whirlwind, Rept. No. R-127-1, September, 1947.

[5] J. Lucking and J. O'Neil, English Electric KDF 9 Logic Diagrams, private communication; January, 1963.