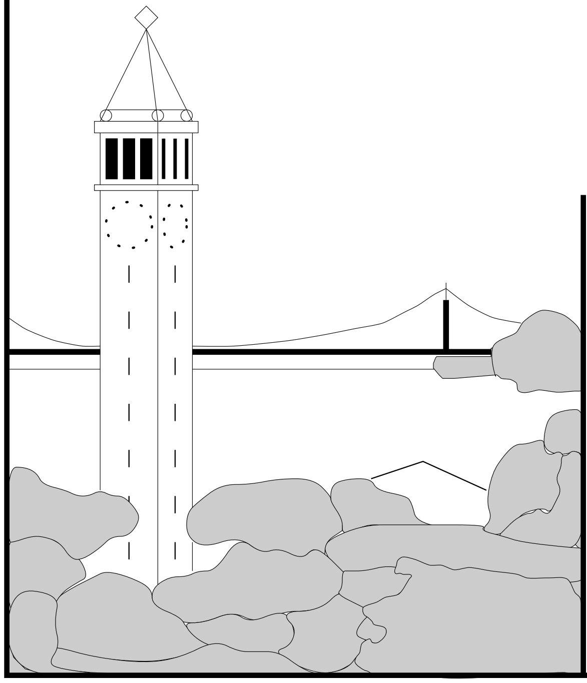
A 64-bit Partitionable Integer Multiplier for VIRAM1

Joseph Gebis



Report No. UCB//CSD-03-1288

November 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A 64-bit Partitionable Integer Multiplier for VIRAM1

by Joseph Gebis

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences. University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David Patterson
Research Advisor

(Date)

* * * * * * *

Professor John Wawrzynek
Second Reader

(Date)

Abstract

VIRAM1 (Vector Intelligent RAM 1) is a low-power multimedia processor with embedded DRAM designed at UC Berkeley in 2002 and fabricated in 2003. It includes a scalar core and four vector computation units, called lanes. The goals of the chip, low-power media processing, require that the vector lanes have efficient integer multipliers that can work with a variety of data sizes. In this report, I describe an efficient partitionable integer multiplier that is designed to work in VIRAM1's vector computation lanes. The multiplier is capable of operating with a latency of two cycles at 200 MHz in a 1.2 V, .18 μm process at 64, 32, or 16 bit data width sizes, while consuming less than 250 mW of power. I describe and evaluate design options for different parts of the multiplier, and analyze the results of the chosen options.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Goals

1.1 The VIRAM1 Datapath and its Integer Multiplier

IRAM (Intelligent RAM) is an approach to designing systems using embedded DRAM and processing on the same die. [FP+97] Vector IRAM (VIRAM) is an architecture that uses this approach to extend a traditional scalar microprocessor with vector processing. [PA+97-2] The first University of California, Berkeley implementation of a VIRAM processor is VIRAM1, which taped out in October of 2002.

From its beginning, VIRAM1 was designed to be a low-power multimedia processor. The IRAM architecture that it is based upon was developed to conserve power. [FP+97] Design decisions were evaluated in the context of executing multimedia applications on a portable device. Power, clock rate, and computation goals were all designed within a multimedia context. [KGW+00]

VIRAM1 contains 13 megabytes of DRAM, a MIPS core that executes scalar instructions, and four computation units (called lanes) that execute vector instructions. Each of the vector lanes, pictured in Figure 1.1, contains a register file, a memory interface, a flag register file and associated processing unit (used for conditional execution), and two arithmetic units. The first arithmetic unit can execute floating point instructions, integer multiplications, and basic integer operations; the second arithmetic unit can not execute floating point instructions or integer multiplications, to save circuit area. The circuitry used to execute floating point and basic integer operations has traditional design goals and constraints, and so I do not elaborate upon them further. The integer multiplier, which has more novel design specifications, is described in detail in this paper. [M00]

1.2 Multimedia Applications

Multimedia applications are becoming increasingly important in the design of modern microprocessors. As consumer demand for products that excel at executing multimedia applications outshadows demand for more powerful processors to execute business, technical, or scientific applications, the goals of a microprocessor designer must also change to reflect the new demand.

The growing demand for processors to execute multimedia applications goes hand-in-hand with consumer devices that are becoming increasingly popular. Cell phones, portable music players,

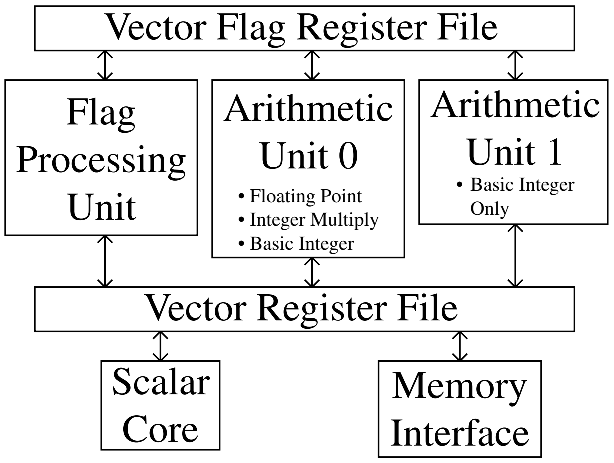


Figure 1.1: VIRAM1 vector lane

video recorders and players, and speech recognition devices all push the demand for microprocessors that can execute multimedia applications efficiently.

Many consumer multimedia devices benefit from power-efficient circuits. Portable devices operate on a battery: lower-power circuits translate into longer battery life, and longer operating time for the device without changing or recharging batteries.

Multimedia applications are largely defined by the application core, which is often a signal processing algorithm. Video encoding and decoding, including compression, is used by video playback devices and recorders. Audio algorithms can include decompression for playback devices, compression and encoding for recorders, and additional processing for speech recognition devices, such as echo cancellation or lexeme analysis and dictionary lookup.

Multimedia cores have a number of common features. They tend to operate on narrow width data items, often 8 or 16 bits; they are applied to large amounts of streamed data; and the cores themself may be individually rather simple, but are applied many times to the data. They also share a number of typical algorithms: impulse response filters are used in audio and video processing; edge detection filters can be used to recognize items in pictures or video; discrete cosine transforms are commonly used in compression operations; and array-type operations are used in many algorithms.

These multimedia devices, with their lower-power demands and common signal-processing-style computation cores, are the focus of VIRAM1's goals. They determine how design decisions were made and affect the microprocessor on all levels, from circuits to architecture.

1.3 The Importance of Partitionability

VIRAM1 targets multimedia applications, which typically process large amounts of relatively narrow-width streaming data. They also can require the ability to do higher-precision, wide calculations. These two seemingly at-odds requirements present a unique challenge in the design of arithmetic circuits.

Table 1.1 shows the data widths that are typical for the types of applications that VIRAM1 targets [KP02, K02]. The widths range from 8 to 32 bits.

Application	Data size	Description
From MiBench:		
cjpeg	32	JPEG image compresion algorithm
mad	32	MPEG audio compression decoder
tiffdither	16	TIFF image dithering algorithm
ispell	8	Spell checking utility
From DSPstone:		
adpcm	16	G.721 ADPCM transcoder
convolution	16	General convolution algorithm
fft	16	General Fast Fourier Transform algorithm

Table 1.1: Typical data sizes for target applications

One approach to dealing with the issue of the need to perform both narrow-width and higher-precision calculations is to use a single, high-precision arithmetic unit to perform all calculations. Narrow-width values can be extended to higher precisions, and then calculations on that data can be performed on the same hardware that operates on wider data. Unfortunately, the approach of using wide arithmetic units to perform calculations on narrow-width data means that a large amount of circuit potential remains unexploited. For example, a standard 64-bit multiplier will leave three-fourths of its circuits effectively unused when operating on 32-bit data, and fifteen-sixteenths of its circuits unused when operating on 16-bit data. [L97]

Unexploited circuit potential has a number of drawbacks. For one, it means that the arithmetic unit is not executing at as high a throughput as it could: if an arithmetic unit's circuits are all being effectively used, more calculations will be performed sooner. Unused circuitry also consumes area, which could have been used for other circuitry, and increases die size, which increases the cost of the chip. Wide arithmetic units that operate on narrow data may do so by extending the width of narrower data, which can waste power, if the wider calculation performs unnecessary subcalculations that consume power.

A more effective way to deal with the need to perform both narrow and wide operations is to use partitionable circuits. Partitionable circuits can execute wide operations, but can also be split up to execute multiple narrower operations on the same hardware. By performing multiple smaller operations at once, overall throughput for the arithmetic unit increases, and unnecessary circuitry that consumes space and power is reduced.

Another benefit of partitionable circuits is that partitioned circuits are generally easier to use in a more power-effective manner. If a partitioned circuit is operating on narrow data, but is not performing its maximum number of operations, it is clear which sections are in use or not. Consequently, partitionability makes it easier to design circuitry that reduces energy consumption in those blocks, for example by gating clocks or preventing inputs from changing. A non-partitioned circuit operating on a single narrow data item will not have as clear a path to reduce the power from unnecessary circuitry.

1.4 Multiplier Circuit Goals

Since VIRAM1 targets portable applications, a low operating voltage of 1.2 volts was chosen for logic circuits. Initial simulations showed that at this voltage, a 5 ns clock cycle (200 MHz) was attainable, and would lead to the desired performance for target applications.

Consequently, the integer multiplier has to operate on a 5 ns clock cycle. The latency for all multiply operations is two clock cycles.

Besides those initial voltage and cycle constraints, low power and low area are also desired goals.

Clock cycle	5 ns
Operating voltage	1.2 V
Latency	2 cycles
Area	< 2 mm^2 in .18 μm process
Power	< 250 mW

Table 1.2: Goals for VIRAM1 Vector Integer Multiplier

Chapter 2

Related Work and Background Information

2.1 Overall Multiplier Organization

Multiplier operation can be split into three main parts:

1. Partial product generation
2. Carry-save addition to generate summands
3. Carry-propagate addition to produce the final result.

Additionally, many multipliers employ Booth encoding, which adds an encoding and decoding step to the process. [B51] A multiplier that can operate on negative values will also require some additional steps: they may be explicit separate steps, or can be folded into one of the previous steps (for example, as part of partial product generation).

The general multiplication procedure, excluding Booth encoding, is shown in Figure 2.1 and proceeds as follows. First, individual bits of the multiplicand are "multiplied" by individual bits of the multiplier; since the operation involves only single bits, the calculation is identical to a logical-and operation.

Partial products generation results in many different bit-weight positions (a result of $2n$ bits, and therefore positions, may be generated from an $n * n$ multiplication). Each of those positions contains a number of partial products (from 1 to n for an $n * n$ multiplication) that must be added to generate the final result in that bit position.

The process of adding up the partial products is generally split into two parts: first, partial products of a single bit-weight are added in a carry-save manner, and then a final single carry-propagate addition is performed on the resulting summands. Splitting the addition in this way minimizes the number of carry-propagate adds that must be performed; since their latency is greater than that of a carry-save add, the overall latency of the adding is reduced.

Carry-save additions of partial products in a single bit-weight position are performed with an adder array. The adder array comprises a large number of compressors, which are simple circuits that reduce a large number of equal-value inputs to a smaller number of outputs by adding the inputs and producing some outputs with higher value. The simplest compressor is a 3-input, 2-output full adder that takes three equal-value inputs and produces a sum bit, with the same value as the input bits, and a carry bit, which has a value equal to twice that of the input bits.

The final addition is a carry-propagate addition, which takes as inputs the summands produced by the adder array and produces as outputs the final result of the multiplication.

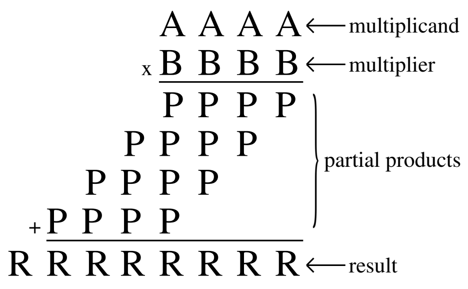


Figure 2.1: Multiplication

2.2 Partial Product Generation

One of the first steps in multiplication is partial product generation. In binary multiplication, partial product generation creates all of the terms that result from multiplying each bit in the multiplicand with each bit in the multiplier. An unsigned multiplicand A can be represented as:

$$a_{n-1}a_{n-2}\cdots a_2a_1a_0 = \sum_{i=0}^{n-1} a_i 2^i \qquad \text{(Equation 2.1)}$$

and unsigned multiplier B as:

$$b_{n-1}b_{n-2}\cdots b_2b_1b_0 = \sum_{j=0}^{n-1} b_i 2^i \qquad \text{(Equation 2.2)}$$

The result of multiplying A and B is then:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j 2^{(i+j)} \qquad \text{(Equation 2.3)}$$

Each of the intermediate terms that are added are the partial products.

The multiplication of each pair of bits is usually performed by logical-anding the bits. That process works correctly for positive numbers, but will produce incorrect results for negative numbers: it produces negative partial products, which can't be added directly by the adder array. Baugh and Wooley [BW73] address this issue by replacing negative terms with their two's complement forms and rearranging. If A and B now represent signed values, they can be represented as:

$$\begin{aligned} A &= -\,a_{n-1}2^{(n-1)} + \sum_{i=0}^{n-2} a_i 2^i \\ B &= -\,b_{n-1}2^{(n-1)} + \sum_{j=0}^{n-2} b_j 2^j \end{aligned} \qquad \text{(Equation 2.4)}$$

and their product P as

$$\begin{aligned} P = A \times B = & a_{n-1}b_{n-1} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{(i+j)} \\ & - \sum_{i=0}^{n-2} a_{n-1} b_i 2^{(n-1+i)} \\ & - \sum_{i=0}^{n-2} b_{n-1} a_i 2^{(n-1+i)} \end{aligned} \qquad \text{(Equation 2.5)}$$

The subtracted terms in the above equation can be replaced with their two's complement:

$$\begin{aligned} & - 2^{(2n-1)} + 2^{(2n-2)} + 2^{(n-1)} + \sum_{i=0}^{n-2} \overline{a_{n-1} b_i} 2^{(n-1+i)} \\ & - 2^{(2n-1)} + 2^{(2n-2)} + 2^{(n-1)} + \sum_{i=0}^{n-2} \overline{b_{n-1} a_i} 2^{(n-1+i)} \end{aligned} \qquad \text{(Equation 2.6)}$$

Baugh and Wooley continue to rearrange the above equation to match their circuit goals. Their arrangement simplifies dealing with negative numbers, but uses logical-ands and logical-ors. Those logical operations can be implemented, but it is usually easier to use circuits that perform logical nands and nors instead. Dada [D85] recognized that removing logical-ands and logical-ors is beneficial in many cases, and adapts Baugh and Wooley's formula by rearranging terms further:

$$- 2^{(2n-1)} + 2^{n} + \sum_{i=0}^{n-2} \overline{a_{n-1} b_i} 2^{(n-1+i)} + \sum_{i=0}^{n-2} \overline{b_{n-1} a_i} 2^{(n-1+i)} \qquad \text{(Equation 2.7)}$$

2.3 Adder Array

Once individual bits have been multiplied and partial products generated, they have to be added. To reduce latency, the addition is split into a carry-save add and a carry-propagate add. The carry-save add is performed by the adder array, which takes all of the bits in a single bit position and adds them.

Because of the relative complexity of adding partial products as compared to generating them, and because of the large number of partial products, the adder array is generally the largest part of a multiplier. Therefore, the size and layout efficiency of the adders in the array, and the routing between them, have a significant impact on the overall size of the multiplier.

The latency of the adder array can be significant as well. The type of adders in the array, and the way they are arranged, can have a large impact on the overall speed of the multiplier.

Unfortunately, the different aspects of adder array design are often at odds with each other. An array built from small adders using simple routing will consume a small amount of circuit area, but may have a large tree height that leads to long latency. The different design tradeoffs must be considered in the context of the overall goals of the multiplier. It's important to be able to measure the relative strengths and weaknesses of different designs to be able to compare them effectively.

The efficiency of an adder array can be measured in various ways. One can look at the total size of the array, but that will include the size of the particular adders used, and the size consumed by routing. Comparing those directly requires that different designs be implemented, which is very time-consuming.

Adder trees, which define how the adders in the array are connected, can be compared to exclude the variable of adder size. [SKG77] One way to compare trees is to take a look at the tree height: a smaller height means that there are fewer sequential adders required in an addition, which can mean that the latency of the addition will be lower. Another way to compare trees is to try to use some characteristic of the tree design to extrapolate information about the likely size of the tree. For example, feedthroughs, which are the number of wires that route a signal past a particular adder in a tree, can be used to get an idea of the routing complexity of that tree. [MJ92]

An important type of adder array is built from Wallace trees. [W64] The general scheme in Wallace tree design is to group partial products in threes, assign each of those groups to the inputs of a full adder, and then successively combine the resulting outputs with additional full adders. Wallace trees have the smallest tree height for trees made of full adders, but the design does not lead to regular layout, and the routing complexity can lead to a routing-dominated circuit of large size.

Another important type of tree used in adder arrays is Balanced Delay trees. [ZM86] Balanced Delay trees are defined by their goal of matching latency to the inputs of any particular adder in the tree. A subclass of Balanced Delay trees that is designed for multiplier adder arrays is Overturned Stairs trees. Overturned Stairs trees have a very regular, dense layout, and have a maximum number of feedthroughs of two for any tree height. The drawback to using Overturned Stairs trees instead of Wallace trees in adder arrays is that Overturned Stairs trees can require a greater tree height than Wallace trees to add the same number of inputs.

2.4 Final Adder

Once the partial products have been generated and reduced to two bits in each position, the final addition must be a carry-propagate addition that produces the final result. The area of the final adder can be relatively small compared to that of the adder array, because it has to operate on many fewer bits. For greater-width multiplications, the latency of this add can be comparable or greater than that of the adder array, because the addition has to proceed sequentially through the entire width of the result.

One important type of adder used as the final adder is a carry-lookahead adder. If the two summands of an addition are represented in this fashion:

$$\begin{aligned} A &= a_{n-1}a_{n-2}\cdots a_i\cdots a_2a_1a_0 \\ B &= b_{n-1}b_{n-2}\cdots b_i\cdots b_2b_1b_0 \end{aligned} \qquad \text{(Equation 2.8)}$$

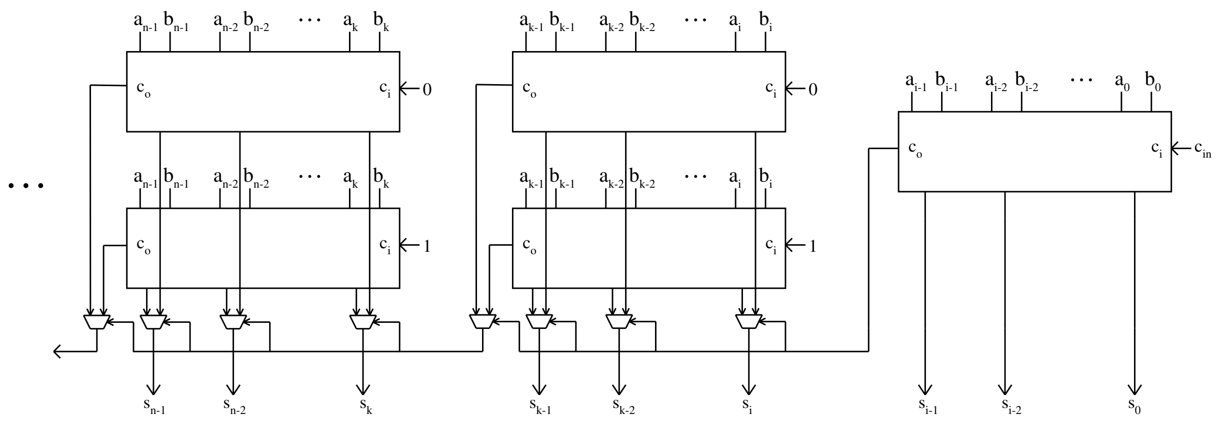
then the equations for each bit position of the resulting sum are:

$$s_i = a_i\,\overline{b_i}\,\overline{c_i} + \overline{a_i}\,b_i\,\overline{c_i} + \overline{a_i}\,\overline{b_i}\,c_i + a_i\,b_i\,c_i \qquad \text{(Equation 2.9)}$$

where c_i represents the carry-in to bit position i and the carry-out from the bit position $i-1$.

The carry-in bits can be expanded, and the resulting equations can be implemented directly for each bit position, leading to reduced serialism in the output bits and a lower overall latency. The largest problem with carry-lookahead adders is their large area requirement.

A carry-select adder, as Figure 2.2 shows, splits the full addition into smaller segments and adds those in a straightforward carry-propagate fashion. Each segment, except for the first, is added twice: once with a 0 for the carry-in, and once with a 1. When the first segment completes its addition, the resulting carry-out is used to select which of the results from the second segment is used for the final result. The carry-out of the selected result in the second segment is used to select which of the results is used from the third segment, and so on.



a_{n-1} b_{n-1} a_{n-2} b_{n-2} ⋯ a_k b_k
c_o
c_i
0
1
a_{k-1} b_{k-1} a_{k-2} b_{k-2} ⋯ a_i b_i
a_{i-1} b_{i-1} a_{i-2} b_{i-2} ⋯ a_0 b_0
c_{in}
. . .
s_{n-1} s_{n-2} s_k s_{k-1} s_{k-2} s_i s_{i-1} s_{i-2} s_0

Figure 2.2: Carry-select adder

2.5 Compressor

A compressor is simply an adder circuit. It takes as inputs a number of equally-weighted bits, adds them, and produces as output the sum, in the form of a bit with the same weight as the inputs and one or more bits that have a value greater than that of the inputs. Compressors are commonly used to reduce a large number of inputs to a smaller number, such as in a multiplier, where they are used to reduce the many partial products to a final summed value. The reduction in the number of individual bits representing the value leads to the name “compressor”.

Compressors are usually the basic subblock that multiplier adder arrays are built around. The logical design of a compressor is integral to the adder tree design, since the number of inputs and outputs determines the basic tree component.

3:2 compressors, which are just full adders, add three equally-weighted inputs to produce a sum bit that has the same weight as the inputs, and a carry bit which has a weight equal to twice that of the inputs. It is the basic blocks of many adder trees.

4:2 compressors have five inputs of equal weight, and produce three bits: a sum of equal weight to the inputs, and two carry bits which have a weight equal to twice that of the inputs. 4:2 compressors are often connected with one of the carry-out bits feeding an input on another block as a carry-in, which allows them to reduce 4 inputs to 2 outputs as well as propagating carries.

Because compressors are the basic circuit that multiplier adder arrays are built from, and because the adder array is such a large part of the overall multiplier, compressor circuit design plays a large role in determining the multiplier's performance. Compressors are a worthwhile target for attempting circuit optimizations, exploring different logic families, and concentrating design effort. As with any circuit optimizations, the typical tradeoffs of area, power, speed, and design time are all in effect.

2.6 Summary of Related Work

The main steps of multiplication are:

1. Partial product generation
2. Carry-save adding
3. Carry-propagate adding

For the first step, partial product generation, Dada's rearrangment of Baugh and Wooley's formula to deal with multiplying negative numbers works well and fits perfectly with the goals of this multiplier.

The second step, carry-save adding, combines partial products with the use of an array of adder trees. There are a number of choices of tree types to consider: Wallace trees are efficient but have irregular routing; Balanced Delay trees have a more regular layout, but are less efficient; other trees have lay in between. Trees can be evaluated on the height of the trees required to add a certain number of inputs, on routing complexity, or total size of the array.

The second step requires another decision: the type of compressor that the adder trees are built from. To some degree, the compressor type depends on the tree type: some trees must be built from blocks that add and produce certain numbers of bits. Most trees, though, use 3:2 compressors, or full adders. The usual concerns of area, latency, and power all apply to the circuit designs considered for the compressors.

The final step of carry-propagate addition presents us with the wide array of choices available for adders. The usual tradeoffs or layout area, latency, and power efficiency apply. Carry-lookahead adders are fast, but consume large amounts of area and power. Very simple adders can have a large latency. An adder that balances the tradeoffs, such as a carry-select adder, can obtain a large amount of the benefit in all of the characteristics to be considered.

Chapter 3

Modified Overturned Stairs Tree

The largest part of the multiplier is the adder array. The array contains a number of adders arranged in trees, which sum the partial products. The tree contains both the compressors used to perform the additions, and the arrangement of the interconnect between them.

Because the adder array is such a large part of the multiplier, its design is critical in determining the overall characteristics of the multiplier. The type of tree chosen for the arrangement of the adders has a large impact on the speed and area of the multiplier array.

The goals of the tree used in the adder array include low tree height to reduce the latency of the additions, a simple circuit to reduce the layout complexity, and regular overall layout to reduce the size of the entire array. To meet these goals, a modified version of an Overturned Stairs tree is used in the adder array.

3.1 Original Overturned Stairs Tree

Overturned Stairs (OS) trees [MJ90] are a form of Balanced Delay trees [ZM86]. Balanced Delay trees are defined by their organization which leads to equal latency to the inputs of any particular adder.

As Figure 3.1 shows, Overturned Stairs trees are created recursively, using branches which consist of full adders chained together to add inputs. Each branch terminates in a connector, which consists of two full adders that add two inputs from their own branch and three inputs from the previous branch, and generate three outputs for the next branch.

The tree starts off with just a single full adder for the first branch and the two terminating full adders for the second branch, and each successive branch is one full adder deeper than the previous. The successive increasing of branch height leads to the balanced-delay property, and creates the overturned-stairs characteristic shape. The final branch contains a terminating root full adder that receives the three outputs from that branch's connector.

3.2 Tree Heights

Different adder array designs that use the same basic adder blocks (for example, full adders) can be compared for latency by comparing the maximum tree height of each design.

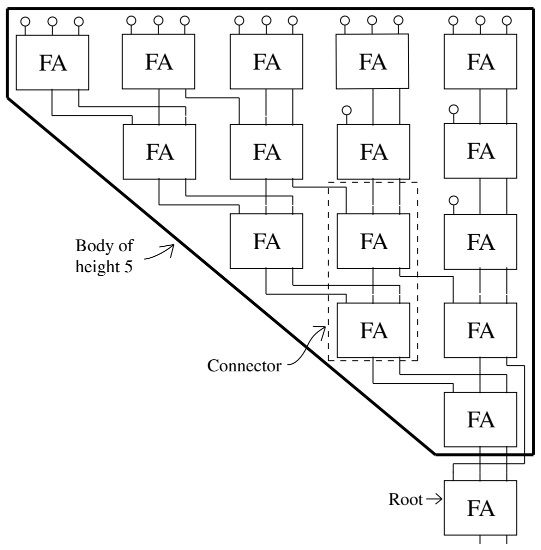


Figure 3.1: Overturned Stairs tree

Number of Inputs	OS Tree Height	Wallace Tree Height
4	2	2
8	4	4
16	6	6
24	7	7
32	9	8
40	10	8
48	10	9
56	11	9
64	12	10

Table 3.1: Tree height required for various numbers of inputs

As the examples in Table 3.1 demonstrate, the standard Overturned Stairs tree may have a greater tree height in general, but there are certain values of numbers of inputs that require the same height for both Overturned Stairs and Wallace trees [W64]. Therefore, no latency penalty is incurred by using Overturned Stairs trees instead of Wallace trees for adding those numbers of inputs; in fact, the simpler organization of Overturned Stairs trees will most likely lead to a simpler layout that has lower interconnect latency.

For adding 16, 32, and 64 bits, a Wallace trees requires heights of 6, 8, and 10 adders; an Overturned Stairs tree requires heights of 6, 9, and 12 adders.

3.3 Difficulties in Partitioning OS Tree

Partitioning a multiplier allows the same hardware to be used to perform a single large multiplication or several smaller multiplications. Designing a multiplier to be partitionable requires designing all of the subcomponents to be partitionable at the appropriate sizes. For the adder array, the partitioning requires that the trees used to add the partial products be able to be split into smaller trees.

Partitioning an adder tree is simplest if the tree can be split at the appropriate size without much modification. Such modification, usually involving additional routing between compressors, creates a number of inefficiencies. Adding routing generally expands the size of the adder array, which can either consume space that would otherwise be available to other circuits, or else expands the size of the entire chip and increases costs. Furthermore, additional routing negatively affects the adder array itself, since expanding the circuits means that the parasitic resistance and capacitance of the interconnect increases. Greater parasitics consume extra power, and can cause delay in the circuit. Additionally, modifying the circuits a large amount increases complexity, which increases design and verification time.

There is generally no straightforward way to partition the original OS tree in half or other useful fractions. It is possible to break the interconnect at appropriate places to create subtrees that can add the required number of inputs, but the resulting trees are not symmetrical, leading to great complexity in designing the partitioned version of the trees, and making it difficult to further partition the resulting subtrees.

3.4 Modified OS Tree

A modified version of the OS tree embodies many of the advantages of the original OS tree, but is easily partitionable. The basic idea is to start with OS trees of the minimum partition size, and combine them into successively larger OS trees. Figure 3.2 shows a 64-bit tree built from 16-bit trees in this way.

Unmodified OS trees produce two bits at each bit position. The same size, two bits, is still required at all partition sizes of the tree, to feed to the final carry-propagate adder.

The smallest-partition tree already produces outputs in the correct form for the final adder. The next-larger-partition tree contains two smallest-partition trees, and therefore produces four bits in each bit position. To turn that result into the correct form for the final adder, the modified OS tree uses 4:2 compressors to combine the results of the smaller trees. The next-largest-partition tree then simply consists of two smallest-partition trees and a set of 4:2 compressors in a combiner form.

The same process of using a set of 4:2 compressors to combine two smallest-partition trees into a next-larger-partition tree can be iterated to produce trees of any power of two multiple of the smallest partition.

The result at the appropriate level, from the smallest size up to the largest available partition that uses all the combiners, is selected with multiplexors and fed to the final adder.

Combining trees with compressors increases the number of inputs that the tree can add. The overall capacity of the multiplier is determined both by the number of inputs that each tree slice can add, and the width of the adder array. Extending the smallest multiplier blocks to wider lengths

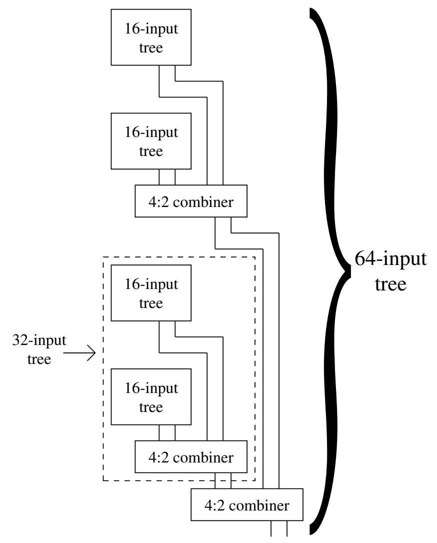


Figure 3.2: Modified tree

is relatively easy: the first and last slices of the smaller arrays must be slightly modified to be able to pass signals between blocks for larger multiplications.

In the multiplier designed for VIRAM1, the minimum partition size is 16 bits, and the larger sizes are 32 and 64 bits, which make use of 2 and 4 smaller OS trees respectively. Figure 3.3 shows the complete array of smaller multipliers, and which ones are used to perform various-sized multiplications.

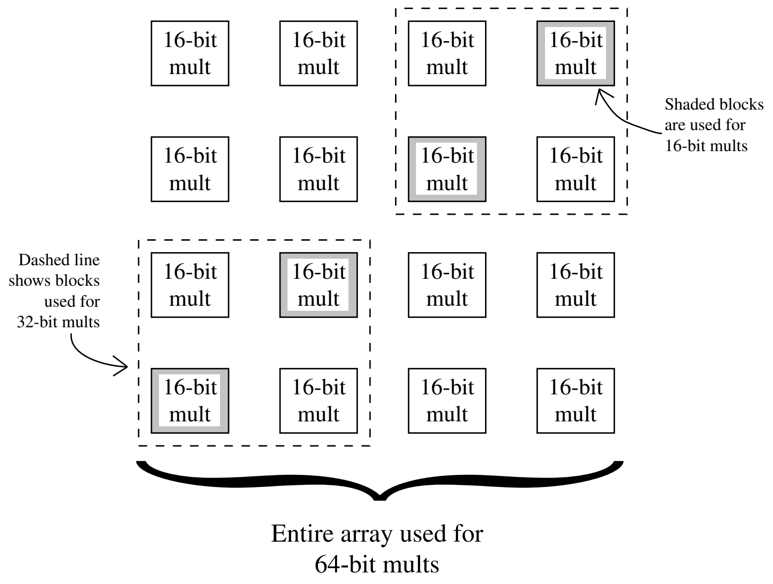

16-bit mult
Shaded blocks are used for 16-bit mults
Dashed line shows blocks used for 32-bit mults
Entire array used for 64-bit mults

Figure 3.3: Top-level 64-bit multiplier

3.5 Additional Modifications

There are a number of small additional modifications that must be made to let multiple smaller trees operate together.

For the multiplier to be able to handle negative numbers, certain bits on the input must be inverted and two additional single ‘1’ bits must be added at the appropriate place. When a single multiplication spans multiple blocks, individual blocks must be able to invert or add the correct bits for that block. Therefore, the steps taken to allow negation must be split up into four parts: negating the bottom bits, negating the left bits, adding the bit for the top left block, and adding the bit for the bottom left block. The appropriate steps must be activated for the correct blocks, based on the block location and the multiplication size.

A single multiplier requires some modification to the last few rows to handle some low-end bits that otherwise aren’t added, and the inputs to the first row must be chosen to represent some high-end bits that otherwise aren’t added. The modified multiplier must be able to independently select either one, or neither, of those modifications, in order to handle multiplications that span multiple blocks horizontally.

3.6 Comparison to Original OS Tree

The modified OS tree shares many of the characteristics of the original OS tree that it is built from, but does have some significant differences, from circuit performance to design complexity.

It can be difficult to accurately compare the original circuit to the modified version. One difficulty in comparison is that the original version can not perform multiple simultaneous multiplications, and so does not have the same functionality as the modified tree. To deal with this difficulty, the modified tree will be compared with both the original tree for the required sizes (16, 32, and 64 inputs), as well as an approach that uses the original trees to perform the required operations. The required functionality can be obtained by using a 64-bit multiplier, a 32-bit multiplier, two 16-bit multipliers, and a small number of multiplexors to select the correct output. Figure 3.4 shows an example of how the submultipliers can be connected to create a full multiplier that provides all the necessary functionality. For example, to perform four 16-bit multiplications with this design, two would be performed by the 16-bit multipliers and two would be performed by the 64-bit and 32-bit multipliers using sign-extended versions of the inputs. The design that uses multiple original trees will be referred to as the “combined original” tree design in following comparisons.

Another difficulty in comparison between the circuits is deciding which characteristics to examine. The characteristics that are important for VIRAM1, beyond the correct functionality, include layout area and power. Both of those characteristics are highly dependent on the layout of subblocks, which can vary with different fabrication technology and different layout engineers. To help partly mitigate those factors as sources of uncertainty, the longest path length (in terms of basic subblocks) will also be examined.

3.6.1 Longest Path

The tree height of the original OS tree varies, depending on the size of the tree. The modified OS tree is built out of smallest-size-partition OS trees, but has additional components that effectively

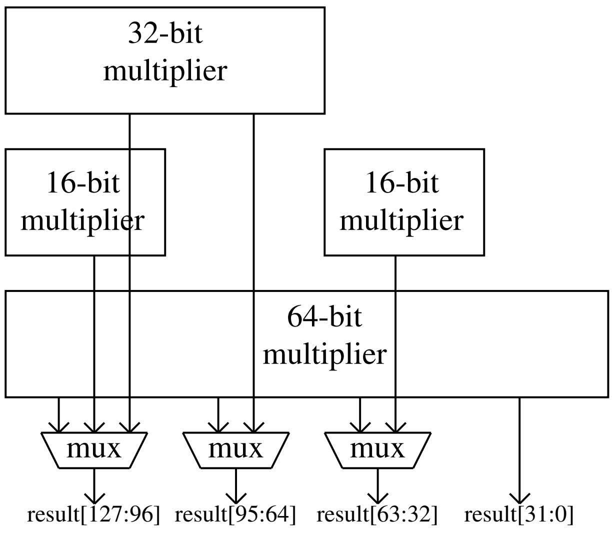


Figure 3.4: Full-functionality multiplier built from unmodified trees

add to the tree height. The original OS tree passes only through a number of full adders, and the final adder. The modified OS tree passes through a number of full adders, through a number of combiners (although that number is zero for the smallest-size-partition), a multiplexor, and then the final adder. Table 3.2 summarizes the heights for original OS trees, Wallace trees (for efficiency comparison), the combined original tree, and the modified OS tree.

Tree Type	Full Adders	Combiners	Muxes	Adder
Original, 16 input	6	0	0	1
Original, 32 input	9	0	0	1
Original, 64 input	12	0	0	1
Wallace, 16 input	6	0	0	1
Wallace, 32 input	8	0	0	1
Wallace, 64 input	10	0	0	1
Combined Original	12	0	1	1
Modified	6	2	1	1

Table 3.2: Longest-path elements for trees and multipliers

The individual Original and Wallace trees provide only part of the required functionality. The Combined Original tree uses multiple unmodified Overturned Stairs tree to provide all the required functionality. The Modified tree is the new approach that provides all the required functionality.

The comparison can be more direct by weighting the various components in terms of their "cost" in terms of full adders. Since the cost is being measured in terms of a different circuit type, it's important to determine what metric of cost is being considered equivalent between the circuits. In this case, since the comparison is for maximum tree height, the cost metric is in terms of latency.

The combiner circuit consists of a number of compressors which are approximately equal to

1.5 full adders. The multiplexor circuit is approximately equal to 2 full adders. The adder circuit is approximately equal to 2 full adders. Table 3.3 shows the results of the weighted comparison.

Tree Type	Full Adders	Combiners (wtd)	Muxes (wtd)	Adder (wtd)	Longest Path
Original, 16 input	6	0	0	$1 * 2$	8
Original, 32 input	9	0	0	$1 * 2$	11
Original, 64 input	12	0	0	$1 * 2$	14
Wallace, 16 input	6	0	0	$1 * 2$	8
Wallace, 32 input	8	0	0	$1 * 2$	10
Wallace, 64 input	10	0	0	$1 * 2$	12
Combined Original	12	0	$1 * 2$	$1 * 2$	16
Modified	6	$2 * 1.5$	$1 * 2$	$1 * 2$	13

Table 3.3: Longest-path weights for trees and multipliers

The individual Original and Wallace trees provide only part of the required functionality. The Combined Original tree uses multiple unmodified Overturned Stairs tree to provide all the required functionality. The Modified tree is the new approach that provides all the required functionality.

3.6.2 Layout Area

Area is an important characteristic of a circuit. Particularly for a large, repeated array such as in a multiplier, small differences in subblocks can add up to significant increases in die size. Furthermore, since there are four lanes in VIRAM1, any increase in a single lane will be multiplied by four times as much area consumed in the entire chip.

The best way to measure circuit area is to design the circuits and see exactly how much area they consume. That way, all of the routing complexity, additional support circuitry, and other issues relating to area consumption are all taking into account. Unfortunately, there are some roadblocks to doing the layout for every circuit that one wishes to compare. First of all, layout of a large circuit can take an extremely long time. Doing the layout for a number of circuits in order to compare them accurately can take prohibitively long. Additionally, the comparison is only valid inasmuch as it represents the skill and time that the designer was willing or able to put into the circuits. A better layout engineer, or one who is willing to spend more time optimizing the layout, may be able to produce the same circuit in a smaller area.

One way to ameliorate some of the uncertainty and time constraints when comparing layout area is to estimate the size of the entire block by looking at actual layout for common subblocks. For example, the original and modified OS trees both use the same compressors, multiplexors, and adders; by laying out those blocks, measuring their area, and examining how many of each of them are required for each tree type, the total size can be estimated.

The technique of estimating the total size by looking at layout of subblocks does raise some important issues. The technique does not work well for very different circuits: the original and modified OS trees can be compared reasonably accurately, while the comparison between an OS tree and a Wallace tree might not reflect the additional routing complexity of the Wallace tree, and a comparison between a tree based on 3:2 compressors and one that uses 4:2 compressors as basic subblocks can be even less accurate.

Table 3.4 shows the estimated size for the original OS tree multipliers, the complete multiplier built from unmodified trees, and the modified OS tree multiplier. The estimates are based on layout of the subblocks done in IBM's CMOS 7SF foundry technology, with a .18 μm gate and three levels of metal. Individual transistors were sized to meet timing goals.

Tree Type	Total Area, μm^2
Original, 16 input	73,427
Original, 32 input	314,616
Original, 64 input	1,300,416
Combined Original	1,882,010
Modified	1,600,445

Table 3.4: Area estimates

The individual Original trees provide only part of the required functionality. The Combined Original tree uses multiple unmodified Overturned Stairs tree to provide all the required functionality. The Modified tree is the new approach that provides all the required functionality.

3.6.3 Power

Because VIRAM1 is targeted at portable devices, power is an important concern. The more power that is consumed, the shorter the battery life will be.

To measure the power of the circuits, the basic circuit blocks were laid out. The circuits were then extracted using Avant!'s StarEX. Spice circuit descriptions of the complete circuits were built, using the extracted decks of the basic blocks as subcircuits. The 16-input multiplier was simulated in HSpice to obtain a baseline power report for comparison, and that value was compared with the power measurement from Synopsys's NanoSim to verify the reported results. The different designs were then simulated in NanoSim for 1000 cycles of random inputs, with the input vectors staying the same for each bit size (e.g., all runs with 16-bit inputs used the same vectors). Table 3.5 summarizes the results.

3.6.4 Qualitative Issues

There are a number of issues that arise from the modifications to the original tree that are not easy to summarize quantitatively.

The modified multiplier has a significant amount of additional layout complexity compared to the original trees. There is a significant amount of extra routing between the blocks that is not required for a single original multiplier. Furthermore, some additional basic subblocks are required (such as the combiners) in the modified multiplier; they are not very complex, but they do add to the design time. The single 16-bit multiplier subblock is also more complicated than a standalone 16-bit multiplier: it has to be able to handle either passing its results out of the low-bit end or

Tree Type	Power, mW	Power, % Above Base
Original, 16 input	8.24	-
Combined Original, four 16-bit mults (avg)	23.95	190.7%
Modified, single 16-bit mult	8.25	0.1%
Modified, four 16-bit mults (avg)	8.69	5.5%
Original, 32 input	29.09	-
Combined Original, two 32-bit mults (avg)	55.92	92.2%
Modified, single 32-bit mult	36.66	26.0%
Modified, two 32-bit mults (avg)	37.44	28.7%
Original, 64 input	143.75	-
Combined Original, 64-bit mult	146.49	1.9%
Modified, 64-bit mult	151.43	5.3%

Table 3.5: Power measurements

The individual Original trees provide only part of the required functionality. The Combined Original tree uses multiple unmodified Overturned Stairs tree to provide all the required functionality. The Modified tree is the new approach that provides all the required functionality. For 16-bit and 32-bit operations, the table shows results with the modified tree performing both a single operation and the maximum number of simultaneous operations.

adding the results in the same way that the standalone block does. Similarly, the 16-bit subblocks have to split up the steps needed to handle negative multiplications and enable them independently.

The 16-bit subblocks in different positions in the complete modified multiplier have different functionality requirements: for example, some of them never have to enable any of the modifications necessary to deal with negative numbers. One way to handle this is to use identical, flexible 16-bit subblocks, and simply never enable the modifications for blocks that do not require them. Another option is to design different blocks that only contain the modifications that they need. The tradeoff is between a simpler design that requires less design time but consumes unnecessary area and power, or a smaller design that takes longer to create. The area and power estimates for the design measured reflect the simpler design, with greater area and power values.

The complete original multiplier is built out of four smaller multipliers. Each one has a convenient layout shape, but it's not clear that there is an efficient manner to lay out all four together. The area numbers reported for the complete adder above do not reflect the potential wasted space from putting them together.

Additionally, the complete original multiplier requires three different trees to be built: the 16-input, 32-input, and 64-input trees. The trees are similar to each other, but they're not identical, so some additional layout time will be required, compared to the modified multiplier.

3.7 Limitations of Technique

The modifications to the original OS tree fit the requirements of VIRAM1 well, but the technique in general still has a number of important limitations that can reduce its attractiveness for certain applications.

The multiplier in VIRAM1 is required to perform 16-, 32-, and 64-bit multiplications. Therefore, the partitioning is somewhat limited: only two half-size partitioning steps are needed. Further partitioning (down to 8- or 4-bit multiplications) using the same technique would require more partitioning steps. Each additional step involves adding another level of combiners, which increases the latency for the multiplication.

Increasing the levels of partitioning incurs additional area penalties as well. VIRAM1's modified OS tree performs 64-bit multiplications with a total of 16 16-bit OS tree blocks. If the design was modified to also allow the multiplier to perform 8-bit multiplications, a total of 64 blocks would be requires; for 4-bit multiplications, a total of 256 blocks would be required.

Of course, the technique of modifying OS trees for partitioning that is described in this paper could be changed slightly for dealing with additional levels of partitioning. For example, if one additional level of partitioning was required, the same design that is presented in this paper could be used without splitting the 16-bit OS trees, but instead adding additional 8-bit OS trees to perform the additional smaller multiplications. Four of the 8-bit multiplications could be performed on the 16-bit adders, and the other four could be performed with the extra adders. There are a number of drawbacks to extending the adder this way: additional routing would be required to apply the input operands to both the original and the extra set of adders, the 16-bit multipliers performing 8-bit multiplications would consume unnecessary power, and the circuitry the makes the extra 8-bit adders is essentially duplicating hardware that is already there.

Chapter 4

4:2 Compressor

The modified Overturned Stairs (OS) tree that is used in VIRAM1 consists of smaller, unmodified OS trees that are combined to form larger trees capable of adding the appropriate number of inputs. Since the smaller trees perform carry-save additions, they produce two bits in each bit-weight position.

The final adder requires two bits as inputs for each bit-weight position. The smallest, 16-bit multipliers produce the necessary two bits, but as the size of the operands increases and more small trees are used to perform the calculation, the total number of resulting bits in each bit-weight position increases. For example, a 32-bit multiplication produces a result that is twice as wide as a 16-bit multiplication, but uses four times as many 16-bit blocks; therefore, instead of producing two bits at each position, it produces four bits. These bits have to be reduced, or compressed, to the appropriate number to feed to the final adder. That compression is accomplished by using a 4:2 compressor.

4.1 Compressor's Role

Compressors can come in a variety of forms, but their common characteristic is that they reduce a number of equal-weight input bits to a smaller number of output bits, by adding inputs and producing output bits that have a greater weight. A simple full adder is therefore a 3:2 compressor. It takes three equally-weighted input bits, and produces two outputs: a sum, which has the same weight as the input bits, and a carry, which has a weight equal to twice that of the input bits.

The compressors that are used in VIRAM1's multiplier need to take four input bits at each position, and produce two output bits. The compressor that fits this requirement is called a 4:2 compressor.

The combiners shown in Figure 4.1 are a series of 4:2 compressors that combine the outputs of two 16-bit OS trees and produce a single pair of output bits in each bit weight position.

4.2 Traditional Compressor

A 4:2 compressor actually takes five input bits and produces three output bits; they are called 4:2 compressors because in the way that they are normally used, one input bit and one output bit are

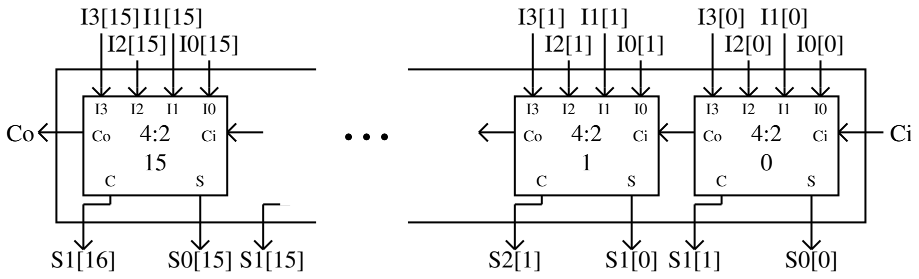


Figure 4.1: Combiner

used as a carry-in and carry-out, leaving four input bits and two output bits available to compress data.

The carry-in and carry-out bits have to have the characteristic that the value of the carry-out bit does not depend on the carry-in bit. That way, the compressors can be chained together without performing a long-latency carry-propagate add. The carry-save configuration of 4:2 compressors means that their latency does not depend on the number of compressors chained together, as it would if the carry were propagated.

The functionality of a 4:2 compressor can be achieved by used two 3:2 compressors, which are simply full adders. Figure 4.2, taken from [ZM86], shows two full adders in a 4:2 compressor configuration. The first full adder receives three input bits, and the second receives two input bits and the sum output bit of the first full adder.

The configuration shown in Figure 4.2 works, but is higher latency than is necessary.

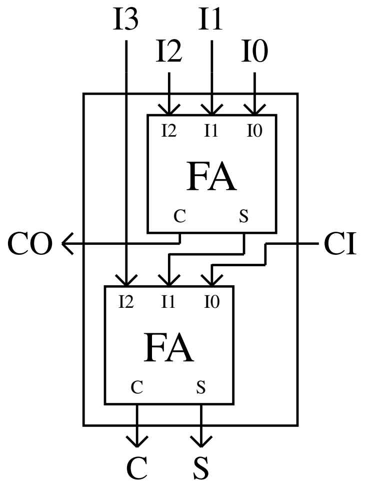


Figure 4.2: Traditional 4:2 compressor

4.3 New VIRAM1 Compressor

The first step in determining how hand-layout logic circuits for VIRAM1 would be designed was to examine a number of logic families. Basic families, as well as other ones such as those listed in [ABE96], were considered. A few basic circuits that would be commonly used were laid out, extracted with Avant!'s StarEX, and simulated with HSpice. The important characteristics of the different logic families, including power consumption, performance, ease of layout, and area were examined to determine the basic design style that would be used for hand layout.

The Complimentary Pass-Transistor Logic with Transmission Gate (CPL-TG) family described in [ABE96] seemed to offer the best match to the goals of VIRAM1, and was therefore used as a starting point for circuit design. Individual circuits were changed depending on their purpose and requirements.

Once the logic family and primitives were established, a number of different designs were examined. Figure 4.3 shows the final design. The sum bit is generated with a series of exclusive-or operations. The carry and carry-out bits are generated by using intermediate exclusive-or results to select appropriate inputs from a multiplexor. Carry-out does not depend on input D or E; input E is only applied at the last logic level, so using carry-out and input E as the carry-out and carry-in for a series of chained compressors will not increase the overall latency.

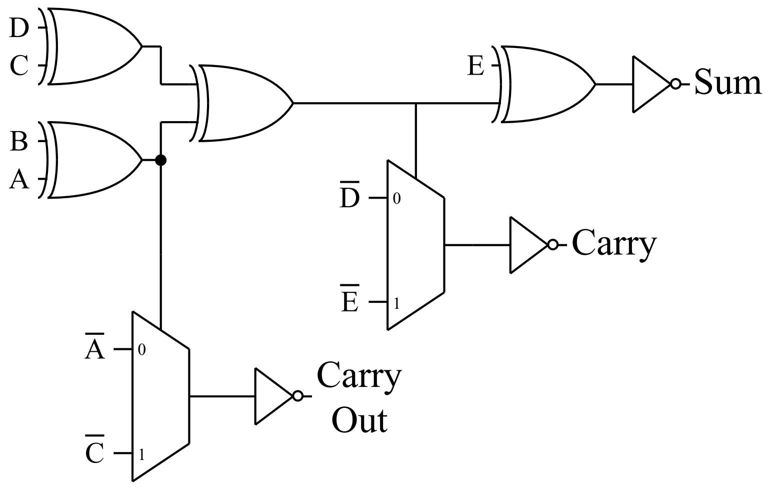


Figure 4.3: New VIRAM1 compressor

4.4 Comparison to Traditional Compressor

The 4:2 compressor in VIRAM1 is used in combiners that connect 16-bit multipliers to perform larger multiplications. Area, speed, and power are all concerns. To measure the performance of the new design and compare to the traditional design, both circuits were built from primitives and simulated using HSpice.

4.4.1 Latency

Both compressors were simulated with HSpice to determine their latency. Because the 4:2 compressors in VIRAM1 are used by chaining them together, the latency is measured by simulating them in that configuration. Table 4.1 summarizes the latency results.

Design	Latency, ns
Traditional Compressor	.234
New VIRAM1 Compressor	.193

Table 4.1: 4:2 compressor latency

4.4.2 Layout Area

Both compressors were laid out to get an accurate representation of their circuit area. Table 4.2 shows the resulting area.

Design	Area, μm^2
Traditional Compressor	292.6
New VIRAM1 Compressor	285.2

Table 4.2: 4:2 compressor latency

4.4.3 Power

The traditional and new compressors were simulated using HSpice with a large number of input vectors to determine their average power usage. Since the 4:2 compressors in VIRAM1 are chained together, and many of the combiners feed other combiners, the power was measured with each circuit driving inputs of a copy of that circuit (that is, the traditional compressor drove other traditional compressors, and the new compressor drove other new compressors). Table 4.3 shows the results.

Design	Power, μW
Traditional Compressor	58.3
New VIRAM1 Compressor	40.8

Table 4.3: 4:2 compressor power consumption

4.5 Conclusion

The VIRAM1 4:2 compress is approximately 3% smaller, 21% faster, and uses 43% less power than the traditional 4:2 compressor.

Chapter 5

Full Multiplier

The full multiplier contains a large number of straightforward smaller subcircuits that are not as novel as the modified adder tree or 4:2 compressor, but are nonetheless important to the multiplier's correct operation.

5.1 Description of Multiplier Sections

5.1.1 Partial Product Generation

Partial product generation is the first step in multiplication. The basic idea is straightforward: "multiply" individual bits of the multiplier and multiplicand together. For positive numbers in binary arithmetic, the multiplication operation is simply a logical-and. Dealing with negative numbers is slightly more complicated. [BW73] explains a method to deal with negative numbers simply, and it is refined in [D85] to simplify the generation by using only logical-nand circuits.

The partial products in VIRAM1 follow the description in [D85], except that both the partial product and its inverse are generated to feed the normal and negated inputs of the compressors in the adder trees.

5.1.2 Final Adder

The final adder in the multiplier is a carry-propagate adder. It takes two bits in each bit-weight position and adds them, producing the final sum.

The design of the adder is very straightforward. The only deviation from a regular adder is that it has to be partitionable to support partitioned multiplications. The modification is quite easy to achieve: the carry chain simply has to be cut at the appropriate places. A multiplexor then selects either the carry-out from the previous section, or a '0' bit as carry-in.

A number of basic adder designs were examined. The final design that is used in the multiplier is a carry-select design. Small sections of the result are added with chains of full adders. Each section, except the first, is added twice: once with a '1' bit as the carry-in, and once with a '0' bit. All additions proceed in parallel. Once the first addition completes, its carry-out result is used to select which of the additions for the next section is the correct one. The result of that section is used to select the next one, and so on.

In a normal carry-select adder, each section size is chosen so that it is as long as possible, while still producing its results before the outputs of the previous section are chosen. Making the section smaller than that would not speed up the addition, since the results can't be used before the previous section's correct carry-out bit is determined, and maximizing the lengths of each section reduces the number of sections and the number of serial selections that must be performed.

In VIRAM1, however, the requirement of partitionability limits the size of adder sections. Instead of treating the adder as if it is a single 128-bit adder, section sizes must be determined as if the adder were only a 32-bit adder. In practice, the limited size of the sections increases the number of sections in the adder, which slightly increases the latency for the addition. The increase amounts to approximately 7% for the straightforward carry-select adder design used in VIRAM1.

5.1.3 Negative Numbers

The partial product generation scheme used in VIRAM1 and proposed in [D85] handles multiplications of negative numbers. Two differences from normal partial product generation result:

- Certain bits must be complemented
- A few additional '1' bits must be added at the appropriate place

For multiplying an m-bit number A and an n-bit number B, the complemented bits can be represented by the following formula:

$$\begin{aligned} a_{(m-1)}b_i2^{(m-1+i)} &\text{ for } 0 \leq i < n-1 \\ b_{(n-1)}a_i2^{(n-1+i)} &\text{ for } 0 \leq i < m-1 \end{aligned} \qquad \text{(Equation 5.1)}$$

Figure 5.1a shows the partial products in their normal multiplication organization; Figure 5.1b shows them in the bit-slice organization that corresponds to the way that are normally laid out. In both cases, the bits that must be complemented are shaded. As Figure 5.1b shows, those complemented bits are on the left and bottom of the array. Note, however, that the bit at the very bottom left (representing $a_{(m-1)}b_{(n-1)}2^{(m+n-2)}$) is not inverted.

The '1' bits that must be added have the value:

$$2^{(m+n-1)}, 2^{(m-1)}, 2^{(n-1)} \qquad \text{(Equation 5.2)}$$

For an adder array that takes two inputs of the same size, those three bits reduce down to two:

$$2^{(2n-1)}, 2^n \qquad \text{(Equation 5.3)}$$

Figure 5.1 shows the two bits specified in Equation 5.3.

The modified Overturned Stairs tree used in VIRAM1 combines individual 16-bit multiplier blocks to perform larger multiplications. Since each basic multiplier block represents only a fraction of larger multiplications, they do not have the same bits complemented. Figure 5.2 shows four 16-bit multiplier blocks combined to perform a single 32-bit multiplication, with the complemented bits in each block shaded.

The requirement of being able to complement different bits depending on the size of the multiplication and each block's position slightly increases the complexity of the overall multiplier.

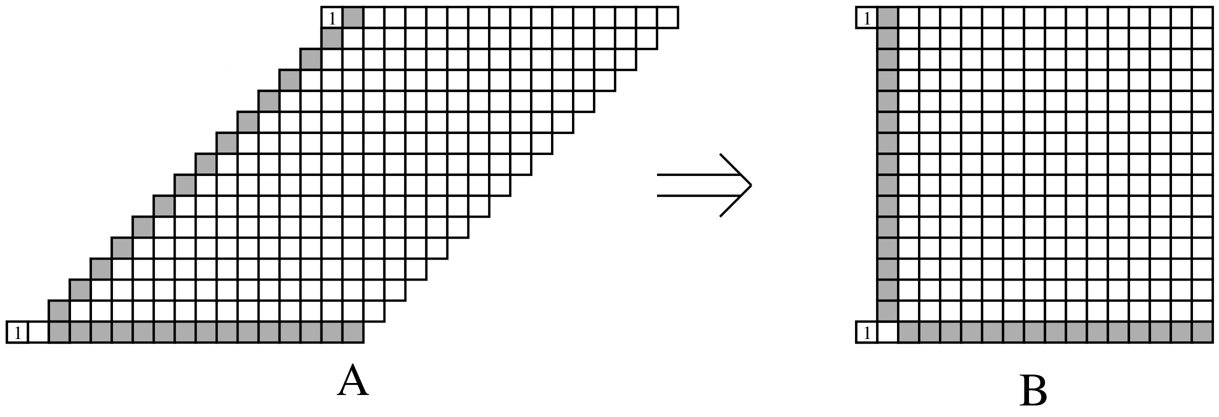


Figure 5.1: Negative numbers in 16-bit multiplication

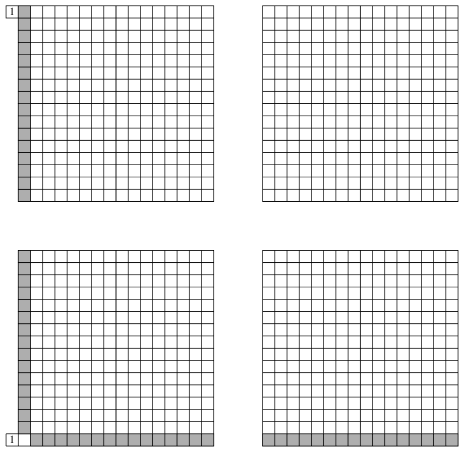


Figure 5.2: Negative numbers in 32-bit multiplication

5.1.4 Overall Multiplier Organization

VIRAM1's multiplier combines small blocks to perform larger multiplications. A total of sixteen 16-bit multipliers are used in the full multiplier.

Figure 5.3 shows the organization of the 16-bit multipliers and how they are combined to perform a 64-bit multiplication. The 16-bit multipliers are combined horizontally by simply passing in intermediate bits from the last row of one multiplier to the first row of the next-least-significant multiplier. The bits are passed in exactly the same way they are passed within each block, so no combiner is necessary.

There is a small modification required to pass bits horizontally between blocks in larger multiplications. Normally, the last few rows of each multiplier differ slightly from other rows by adding in extra partial products that result at the low end of a multiplication. For blocks that are combined horizontally, the last few rows must be able to operate either normally by adding in the extra bits, or as any other row and not adding in the extra bits. The modification is easy to make with a few multiplexors.

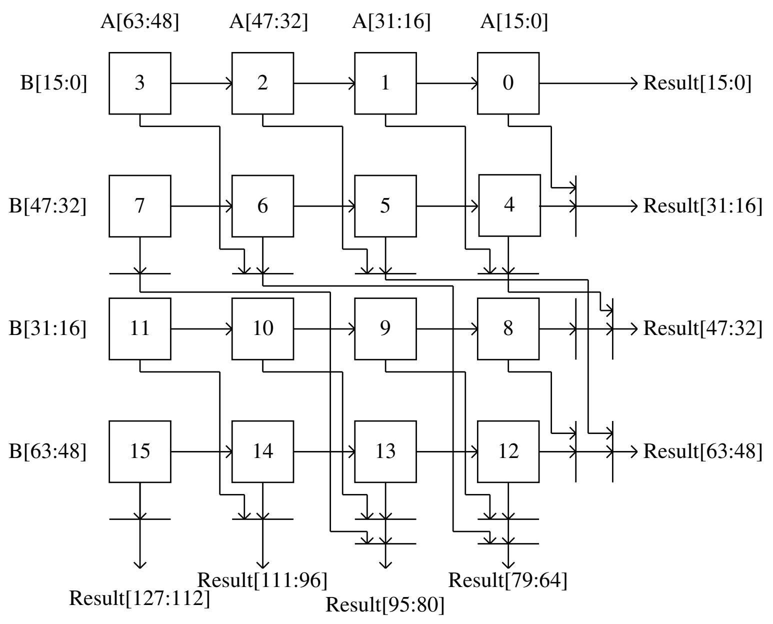


Figure 5.3: Blocks and combiners for 64-bit multiplication

Combining blocks vertically is slightly more complex. First, rows are combined in pairs: the first and second row are combined as one pair, and the third and fourth rows are combined as another. Then, the result from the first combiner is itself combined with the result from the second.

The combiners operate on the normal outputs from each block, which is a pair of bits in each position. Therefore, there is no need for modification of the blocks to allow for combining vertically.

Another potential complication with combining blocks is that they must be combined slightly differently for different-size multiplications. To reduce the need for input multiplexors, the blocks used for each size are chosen so that each block's input bits are the same, regardless of the multiplication size. The correct routing for the output of each block, however, changes depending on the multiplication size. Figure 5.4 shows the multiplier organization for 32-bit multiplications: note that the blocks are numbered the same as in Figure 5.3, but this time only half of them are active.

The routing for 16-bit multiplications is slightly different from both 32-bit and 64-bit multiplications. It is, though, quite simple: the four blocks used (numbered 0, 5, 10, and 15 in Figures 5.3 and 5.4) each produce 32 output bits. No combiners are needed for 16-bit multiplications.

Because the results are produced at different locations for different size multiplications, the correct output needs to be selected. Multiplexors along the edge of the adder array select the appropriate bits to feed to the final adder to produce the result. Some parts of the final result are always produced at the same location and therefore do not need to be selected with a multiplexor: for example, bits 15-0 of the final result are always produced at the side of block 0 (as Figures 5.3 and 5.4 show). Other parts of the result are produced in either two or three different locations, and must be selected appropriately. Each multiplexor is smaller than a full adder, and in total they consume less than 4% of the multiplier's area; their total latency is less than a full adder.

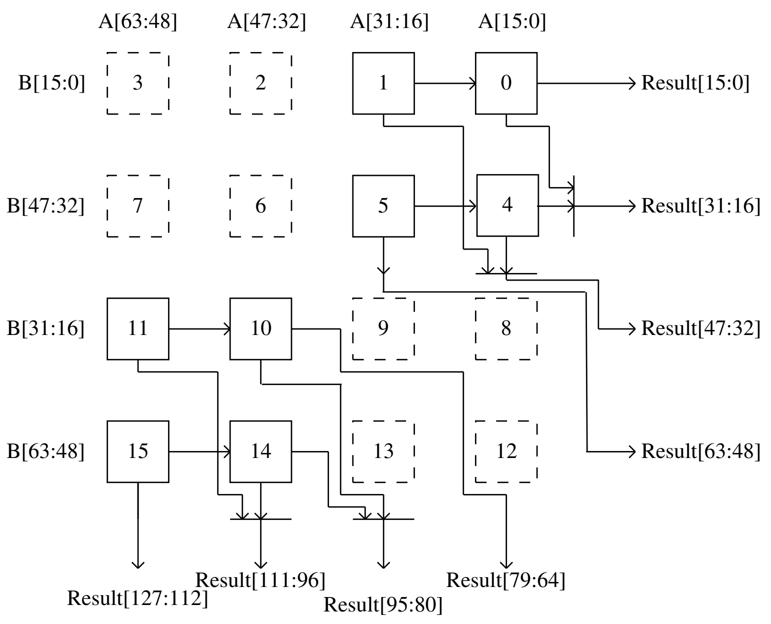


Figure 5.4: Blocks and combiners for 32-bit multiplication

5.2 Power, Performance, and Area

The important characteristics for VIRAM1 include power, performance, and area.

5.2.1 Power

Table 5.1 shows the important power results taken from Table 3.5 for the modified multiplier and the complete multiplier built from original trees. All results represent the average case when each multiplier is performing the maximum number of operations: e.g., four 16-bit multiplications, two 32-bit multiplications, or a single 64-bit multiplication. Both multipliers were simulated with the same basic circuit blocks.

Bit size	Tree Type	Power, mW
16 bit	Complete Original	23.95
	Modified	8.69
32 bit	Complete Original	55.91
	Modified	37.44
64 bit	Complete Original	146.49
	Modified	151.43

Table 5.1: Power measurements

The modified tree is significantly more power-efficient for both 16-bit and 32-bit multiplications. The most significant source of power savings comes from using fewer compressors to perform the calculation: in the original, smaller multiplications are performed by sign-extending inputs and performing the calculation on larger multipliers.

For 64-bit multiplications, the complete original is marginally more power-efficient than the modified version. The main source of extra power consumed in the modified version is the combiners that are used to allow the smaller multipliers to perform larger multiplications.

The target multimedia applications described in Table 1.1 all show data sizes of 32 bits or fewer. Unless a program were to perform nearly all 64-bit multiplications, the modified multiplier is going to consume less power.

5.2.2 Performance

The multiplier for VIRAM1 needs to be able to perform a multiplication in two 5 nanosecond cycles. The adder array consumes a significant part of the time required to perform a multiplication.

The original multiplier has a total effective circuit path length equivalent to 16 full adders. The modified multiplier has an effective circuit path length equivalent to 13 full adders. In practical terms, that means that if both circuits were completely custom-designed, the original multiplier would need to be designed with faster basic blocks, which would most likely consume more area and power.

5.2.3 Area

If both multipliers were built from the same basic subcircuits, the complete original multiplier would require a total of 1,882,010 μm^2, while the modified multiplier would require 1,600,445 μm^2. Hence, the modified multiplier uses about 15% less area.

5.2.4 Summary

The modified multiplier is approximately 20% faster and uses about 15% less area than a multiplier built from traditional blocks. The power consumed by the modified multiplier is about 60% less than what the traditional multiplier uses for 16-bit operations, and about 30% less for 32-bit operations; because of additional circuitry, and modified multiplier consumes about 3% more power than the traditional multiplier for 64-bit calculations.

5.3 Comparison to Synthesized Multiplier

An alternative to a hand-designed multiplier that was considered for VIRAM1 is a synthesized multiplier. A synthesized multiplier would be designed in Verilog and compiled to a circuit netlist using either Synopsys's Design Compiler or Module Compiler tools. The netlist would then be used to produce layout for the multiplier by using an automatic place and route tool.

There are two strong benefits to using a synthesized multiplier: synthesizing a circuit is much faster than laying it out by hand, and modifying a synthesized design is much quicker than with a custom-layout circuit.

Synthesizing does have drawbacks. The most significant problems that synthesized circuits have is that they tend to be larger and consume more power than custom-layout circuit.

If a design uses both custom-layout and synthesized blocks, usually the datapaths are designed by hand and control is synthesized. For each case, the benefits of each methodology are highlighted, and the drawbacks are minimized.

Datapaths are quite regular, often being design by repeating a bit-slice with few changes. That regularity means that they can usually be designed by hand in a reasonable amount of time. Furthermore, their specification does not often change, which means that they don't usually require modification once they've been designed. Datapaths are commonly the place where a large portion of both chip area and power are consumed, so designing them by hand can have a large benefit.

Control blocks generally don't have subcircuits that can be greatly optimized for power and area, which means that they don't stand to gain much from being designed by hand. They're often highly irregular and complex, which means that designing them by hand is extremely difficult. Most bugs and specification changes for a chip require the control to change, which is relatively easy to do in a synthesized block but can be very difficult in a custom-layout block.

Datapaths are often laid out by hand, but synthesizing them is becoming more common. With a large library of components from which to select, a good synthesis tool can do a reasonable job of minimizing circuit area and power. If the datapath is somewhat complicated, the benefits of synthesis become more important: the circuit will be quicker to design, and if any changes are needed, they will be easier to implement.

Unfortunately, an accurate comparison would require a complete implementation of both designs. Although synthesis would require less time than custom hand layout, a serious implementation would require several iterations of synthesizing, timing analysis, and re-synthesis. Time constraints prevent such an undertaking at this point; a good, in-depth analysis would be interesting future work.

Chapter 6

Summary and Future Work

6.1 Results

The initial goals of the vector integer multiplier for VIRAM1 were to design a partitionable multiplier that would operate in two 5 ns cycles at 1.2V, be smaller than 2 mm^2 in a .18 μm process, and consume less than 250 mW of power. The multiplier as designed meets the functional requirements, takes just over 1.6 mm^2, and operates at less than 155 mW of power for 64-bit multiplications — significantly less for smaller multiplications.

The multiplier meets the goals. In the process of designing the circuits, many benefits of the new approaches were shown, and some drawbacks were also revealed.

6.2 Benefits

The biggest benefit from the proposed design is that it saves significant power when performing 16-bit and 32-bit multiplications, which are the sizes most commonly required for the multimedia applications that VIRAM1 targets. The reduced power consumption translates into longer battery life, which is an important concern for portable devices.

The design performs well in other areas as well. The area of the proposed design is slightly lower than that of a multiplier built from single-size blocks. Additionally, the computational length of longest path in the new design is less than that of the old design, which allows the new design to be built with smaller, more power-efficient circuits.

The new design has another benefit: it can be built with only a single type of adder tree being designed. The design built from single-size multiplier blocks requires that 16-bit, 32-bit, and 64-bit trees all be designed and tested. The adder trees all share elements of their design, and all have common subblocks, but the trees are not identical. Tree design and testing was a significant amount of the time consumed for the proposed multiplier; duplicating that effort two more times would be a large effort.

6.3 Drawbacks

One drawback resulting from designing the multiplier from smaller multiplier blocks is that there are a greater number of blocks that must be designed. Most of them are quite simple, but they must all still be designed and verified. A multiplier built from single-size blocks would only require those blocks and a multiplexor to select the correct result. The multiplier design that was used requires multiplier blocks, combiners, and multiplexors; additionally, the multiplier blocks require a number of small modifications to make them operate correctly when being combined.

Another drawback to the technique of combining smaller blocks is that a large amount of routing between the blocks is required. While the total size may still be less than a multiplier built from single-size blocks, the savings end up being reduced by the area taken up for routing.

The routing penalty becomes more significant as more levels of partitionability are required. By the time that two smaller partitions are required, for a total of three sizes (such as in VIRAM1, which required 64-bit, 32-bit, and 16-bit multiplications) the routing is already significant; another level would make the routing dominant.

6.4 Future Work

There are a number of alternative approaches to the design of the multiplier that could provide interesting analyses.

6.4.1 Synthesized Multiplier

One of the most interesting comparisons that could be made is between a hand-layout design and a synthesized design. The two techniques are very different, so there is a big potential for large differences in their performance; quantifying those differences would be very useful to someone considering both design alternatives.

The benefits and drawbacks of hand-layout and synthesized design are known in a general sense, but it's difficult to tell exactly how much the effective the benefits of either approach would be. The multiplier circuit in VIRAM1 has a relatively large amount of complexity that could make either alternative more or less attractive than it might otherwise be.

6.4.2 Different Tree Designs

The tree design used here was a relatively simple modification from the original Overturned Stairs tree. More analysis of different approaches could provide new insight into better methods.

The 16-bit multiplier blocks that were used here were designed to be slightly different depending on where they were within the top-level hierarchy. That way, the extra circuitry that some blocks required would not be wasted in blocks that would never use them. Unfortunately, that approach led to a large amount of extra design and verification time. A quantification of the costs of designing only a single flexible block that will work in any location would be interesting and extremely useful in deciding whether the time savings of not designing blocks separately would be worth the additional costs of wasted area.

6.5 Conclusion

A multiplier for VIRAM1 was designed that meets the design requirements. The multiplier is able to calculate one 64-bit, two 32-bit, or four 16-bit operations at once. It operates in two cycles at 1.2 V in a .18 μm process.

The multiplier meets other goals, as well. It is approximately 1.6 mm^2, and consumes less than 155 mW of power.

The multiplier is based around an adder array that uses Overturned Stairs trees. The basic OS trees are used to form 16-input multiplier blocks, which are then combined with 4:2 compressors to perform larger multiplications. Because the blocks are split up, it is easy to perform two or four simultaneous smaller multiplications. The subblock design also makes it simple to turn off blocks that are not being used, and to thereby save power.

The 4:2 compressors used in VIRAM1 are based on Complimentary Pass-Transistor Logic with Transmission Gate logic. They are smaller, faster, and use less power than the simple alternative of using two 3:2 compressors, or full adders, chained together.

The current design works well, but there is further analysis that should be done to investigate possible improvements.

Bibliography

[ABE96] Abu-Khater IS, Bellaouar A, Elmasry MI. Circuit techniques for CMOS low-power high-performance multipliers. [Journal Paper] IEEE Journal of Solid-State Circuits, vol.31, no.10, Oct. 1996, pp.1535-46. Publisher: IEEE, USA

[B51] Booth A. A Signed Binary Multiplication Technique. [Journal Paper] Quarterly Journal of Mechanics and Applied Mathematics, vol.IV, pt.2, June 1951, pp.236-40. UK.

[BW73] Baugh CR, Wooley BA. A two's complement parallel array multiplication algorithm. [Journal Paper] IEEE Transactions on Computers, vol.C-22, no.12, Dec. 1973, pp.1045-7. USA.

[D85] Dadda L. Fast multipliers for two's-complement numbers in serial form. [Conference Paper] Proceedings of 7th Symposium on Computer Arithmetic (Cat. No. 85CH2146-9). IEEE Comput. Soc. Press. 1985, pp.57-63. Silver Spring, MD, USA.

[FP+97] Fromm R, Perissakis S, Cardwell N, Kozyrakis C, McGaughy B, Patterson D, Anderson T, Yelick K. The energy efficiency of IRAM architectures. [Conference Paper] ACM. Computer Architecture News, vol.25, no.2, May 1997, pp.327-37. USA.

[K02] Kozyrakis C. Scalable Vector Media-processors for Embedded Systems. Report No. UCB/CSD-02-1183. May 2002. Berkeley, CA, USA.

[KGW+00] Kozyrakis C, Gebis J, Martin D, Williams S, Mavroidis I, Pope S, Jones D, Patterson D, Yelick K. Vector IRAM: A Media-oriented Processor with Embedded DRAM. [Conference Paper] Conference Record of the Hot Chips XII Symposium. Aug. 2000. Palo Alto, CA.

[KP02] Kozyrakis C, Patterson D. Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. [Conference Paper] Proceedings 35th ACM/IEEE International Symposium on Microarchitecture. IEEE Comput. Soc. 2002. Instanbul, Turkey.

[L97] Lee RB. Multimedia extensions for general-purpose processors. [Conference Paper] 1997 IEEE Workshop on Signal Processing Systems. SiPS 97 Design and Implementation Formerly VLSI Signal Processing (Cat. No.97TH8262). IEEE. 1997, pp.9-23. New York, NY, USA.

[M00] Martin D. Vector Extensions to the MIPS-IV Instruction Set Architecture (The V-IRAM Architecture Manual) Revision 3.7.5. March 2000.

[MJ90] Mou Z-J, Jutand F. A class of close-to-optimum adder trees allowing regular and compact layout. [Conference Paper] Proceedings. 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.90CH2909-0). IEEE Comput. Soc. Press. 1990, pp.251-4. Los Alamitos, CA, USA.

[MJ92] Mou Z-J, Jutand F. 'Overturned-stairs' adder trees and multiplier design. [Journal Paper] IEEE Transactions on Computers, vol.41, no.8, Aug. 1992, pp.940-8. USA.

[PA+97] Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K. Intelligent RAM (IRAM): chips that remember and compute. [Conference Paper] 1997 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC. First Edition Vol.40 (Cat. No.97CH36014). IEEE. 1997, pp.224-5. New York, NY, USA.

[PA+97-2] Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K. A case for intelligent RAM. [Journal Paper] IEEE Micro, vol.17, no.2, March-April 1997, pp.34-44. Publisher: IEEE, USA.

[SKG77] Stenzel WJ, Kubitz WJ, Garcia GH. A compact high-speed parallel multiplication scheme. [Journal Paper] IEEE Transactions on Computers, vol.C26, no.10, Oct. 1977, pp.948-57. USA.

[W64] Wallace, CS. A Suggestion for a Fast Multiplier. [Journal Paper] IEEE Transactions on Electronic Computers, vol.EC-13, no.1, Feb. 1964, pp.14-7. USA.

[ZM86] Zuras D, McAllister WH. Balanced delay trees and combinatorial division in VLSI. [Journal Paper] IEEE Journal of Solid-State Circuits, vol.SC-21, no.5, Oct. 1986, pp.814-19. USA.