

# Efficient Checksum Calculation using Reduction Trees

Kent Wires, Dale Parson and Jesse Thilo

{wires,dparson,jthilo}@agere.com

*Agere Systems*

## Abstract<sup>1</sup>

As network traffic throughput requirements continue to increase dramatically, network processors are adding specialized instructions that accelerate the more common arithmetic operations. An example is the checksum, which appears in several popular protocols to guard against using corrupted packets. This paper presents two approaches to checksum calculation that provide significant performance improvement over conventional methods. These approaches use reduction trees similar to those found in parallel multipliers to produce two operands for final summation. Reduction trees are appropriate when an array of words is available to be summed at one time, in contrast to serial summing of word pairs. The two methods vary in the design of the reduction tree. The first approach, *leveled reduction based checksumming*, uses predefined array sizes of partial sums, leading to a single circuit design. This approach has the benefit of fixed, design-once circuit parameters. The second approach, *3-to-2 reduction based checksumming*, takes a greedy approach to reducing as many application words as possible at each stage. This approach has the benefit of parameterized circuit design amenable to automatic generation of reduction stages to fit application array size. Both approaches are suitable for a variety of circuit implementations.

**Keywords:** adder, checksum, error detection, multiplier, network processor, packet, reduction tree

## 1. Checksums in network processing

Network processing uses checksums for detecting errors in packets. A checksum is a sum, typically an inverted one's complement sum, of all non-checksum fields in a proscribed part of a packet. For example, the Internet Protocol requires a 16-bit inverted one's complement checksum summed over the 16-bit words in the IP packet header; the Transmission Control Protocol requires a similar 16-bit checksum computed over the TCP header, payload, and portions of the IP header [1].

1. This hardware reduction-based checksum calculation mechanism has been submitted to the U.S. Patent Office in a patent application by Agere Systems Inc. and the authors.

Checksums reside in dedicated fields within their respective packet headers. For a given protocol packet, checksum calculation must occur twice, once on egress in order to insert the checksum into its outgoing packet, and once on ingress in order to compare the arriving checksum to the expected checksum, where the expected checksum is a summation of the incoming fields. The standard procedure for one's complement checksums is to sum all pertinent fields in an incoming packet, including the inverted checksum, and then test for zero. If the result is not zero, then an error has occurred.

Traditionally, checksums are computed using a series of high-latency additions, with some speedup achieved through the use of multiple parallel additions, overflow accumulation, and caching partial checksums for later reuse [2, 3, 4]. Figure 1 shows a simple conventional implementation. In this implementation a 32-bit adder with a feedback path from the output is used to accumulate the appropriate number of 16-bit words to be summed into a 32-bit sum. This sum is partitioned into a 16-bit sum and a 16-bit overflow, which are added together to produce a 17-bit result. The most significant bit of this result is added to the least significant sixteen bits to achieve one's complement addition. It can be shown that this increment cannot lead to overflow, as long as the 32-bit partial sum does not overflow. The incrementer output is then inverted to produce the inverted checksum.

Because of the frequency of use of checksum calculations and the volume of data contributing to a checksum, many network processor architectures contain instructions and corresponding data paths for accelerating checksum summation. The Agere APP550 network processor, for example, contains instructions such as *fStartChecksum16* and *fStopChecksum16* for initiating and terminating one's complement summation on an incoming packet [5]. These high level instructions initiate data path summation that operates in parallel with other APP550 instructions and data paths that are orthogonal to checksum calculation. When an APP550 classification program reaches the end of the portion of a packet needed for a checksum, it invokes *fStopChecksum16* to turn off the parallel checksum circuitry and retrieve the resulting sum. Thus, rather

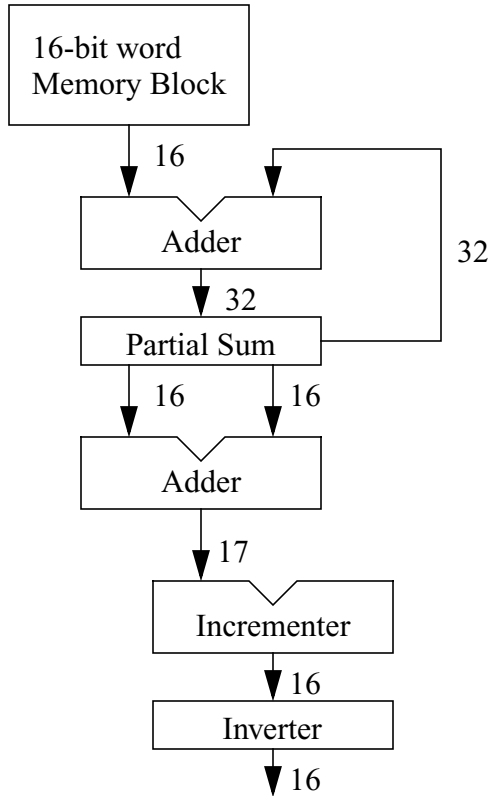


Figure 1: Conventional Checksum Unit

than writing a series of add-and-store instructions, the APP550 programmer simply starts and later stops a checksum computation in hardware. There are additional, protocol-specific instructions such as *fStartIpv4HdrCksum* and *fGetIpv4HdrCksum* for computing and testing standard protocol checksums without requiring additional instructions for packet boundary monitoring or result testing. These instructions increase throughput via parallel computation while reducing the amount of code that a programmer must write.

While a software-based approach to checksum calculation on a standard processor architecture requires summing a series of words, one or a few at a time, a processor architecture with instructions such as *fStartChecksum16* and *fStopChecksum16* can employ parallel summation of larger arrays of packet data. The coarse granularity of these instructions with respect to the number of packet words being summed led us to investigate means for many-word checksum summation. The results reported in this paper represent the outcome of our investigation into further enhancements for parallel checksum formation. Section 2 examines our two, complementary approaches to parallel checksum formation. Section 3 discusses three alternative

approaches to circuit implementation. Section 4 shows some example results. Section 5 gives a summary of two approaches to accelerated multiplier design that inspired our approaches to checksum formation.

## 2. Checksum reduction trees

The optimal delay for parallel checksum formation is achieved when there are  $N$  adders that can perform in parallel, and is given by

$$D_{\min} \approx (\log_2 N_{\text{words}}) \cdot d_{\text{add16}} + d_{\text{add16}} + d_{\text{ovfinv}}$$

where  $D_{\min}$  is the minimum delay,  $N_{\text{words}}$  is the number of 16-bit words to be checksummed,  $d_{\text{add16}}$  is the delay associated with each 16-bit addition, and  $d_{\text{ovfinv}}$  is the delay associated with handling overflow from the final addition and inverting the result. The logarithmic scaling of this result is due to the logarithmic (depth:size) ratio of a balanced reduction tree as illustrated in this section.

### 2.1 Leveled reduction based checksumming

Both of our approaches use multiple stages of summation hardware consisting of parallel full adders, with the two approaches distinguished by the number of words summed at each stage and the means for determining these numbers.

*Leveled reduction based checksumming* uses predefined array sizes of partial sums, leading to a single circuit design. Figure 2 illustrates the concept. Each stage in Figure 2 consists of an array of 16-bit words to be summed, with the most significant bit position on the left, where the final stage consists of two words to be added at the bottom of the figure. The number of words to be summed at each stage  $x$  is given by

$$x_{i+1} = \text{floor} ( 3/2 \cdot x_i )$$

giving the bottom-to-top stage sizes (in 16-bit words) of 2, 3, 4, 6, 9, 13, 19, etc. The number of words at each stage is fixed in this approach.

Given a 12-word array to be summed in the first stage at the top of Figure 2, the 12 rows are reduced to 9, because 9 is the greatest value  $\leq 12$  in the sequence 2, 3, 4, 6, 9, 13, 19, etc. Nine rows are thus summed into the first 6 rows of the second stage of Figure 2, and the remaining 3 rows of the first stage propagate to the second stage unchanged.

Summation of the first 9 rows of the first stage of Figure 2 proceeds as follows. Partition each column of bits into vertical groups of 3 bits. There are three such 3-bit vertical groups in each column of this 9-word stage. Each 3-bit group supplies the inputs to a full adder circuit, whose outputs advance to the second stage. The

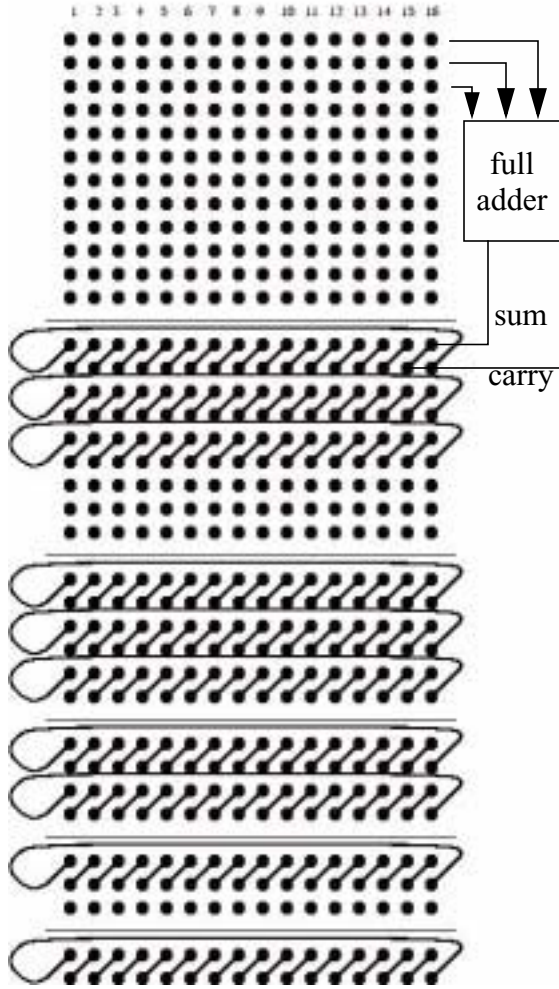


Figure 2: Leveled Reduction Checksumming

second stage of Figure 2 shows bit pairs connected by diagonal lines. The upper right bit in each such pair represents the *sum output* of a full adder coming from the first stage, while the lower left bit in the next most significant column represents this adder's *carry output*. Figure 2 shows one example *full adder* that sums the first three bits in the first column of the first stage into the first bit in the first column of the second stage, with the carry output of this adder becoming the second bit of the second column of the second stage. Other vertical groups of three bits sum similarly, with the respective carry out bits propagating to the next most significant bit column in the next stage. The most significant carry output bit propagates back to the least significant column in accordance with one's complement addition. This carry-around propagation avoids the extra overflow addition step needed by conventional serial summation.

When addition of the first stage is completed, the

summed bits plus the untouched original bits contribute to a new 9-word array, which is similarly summed to the 6-word array of the third stage. Reiterating this method, the matrix is reduced to two 16-bit rows. These rows are added together, and the sum is incremented if the addition produces an overflow. The checksum is computed by inverting the output.

The total reduction delay of this computation method is approximately

$$D_L \approx M_{Lstages} \cdot d_{FA}$$

where  $D_L$  is the total delay,  $M_{Lstages}$  is the number of stages required to reduce the original matrix to two rows, and  $d_{FA}$  is the delay of a full adder. Since each full adder in a given stage can function simultaneously, the delay of each stage equals  $d_{FA}$ . Thus, the total delay is  $d_{FA}$  times the number of reduction stages.

In this example, the original 12-row matrix is reduced to nine, then six, four, three, and finally two levels. If we estimate the delay of a full adder to be two gate delays [6], the total delay of the reduction unit is ten gate delays.

## 2.2 3-to-2 reduction based checksumming

*3-to-2 reduction based checksumming* differs from the leveled reduction approach by reducing as many words as possible at each stage. Where leveled reduction fits multiple words to be summed into fixed size summation stages, 3-to-2 reduction derives the size of the summation stages from the number of words to be summed. Leveled reduction gives a fixed summation circuit; 3-to-2 reduction gives a circuit synthesis algorithm.

Figure 3 illustrates 3-to-2 reduction. Once again, each stage consists of an array of 16-bit words to be summed, with the most significant bit position on the left, where the final stage consists of two words to be added at the bottom of the figure. The number of words to be summed at each stage  $x$  is given by

$$x_{i-1} = \text{ceiling} ( 2/3 \cdot x_i )$$

Stage size is determined by the number of initial words to be reduced, giving stage sizes 12, 8, 6, 4, 3 and 2 in Figure 3. Reduction consists of adding 3 bits within a column, with the carry bit propagating to the next most significant column of the next stage, and with carry-around from the most significant to the least significant column, as it does for leveled reduction. As in Figure 2, Figure 3 shows diagonal lines connecting associated sum and carry bits from the preceding stage, with the carry bit in the more significant column. The primary change from leveled reduction is 3-to-2's greedy

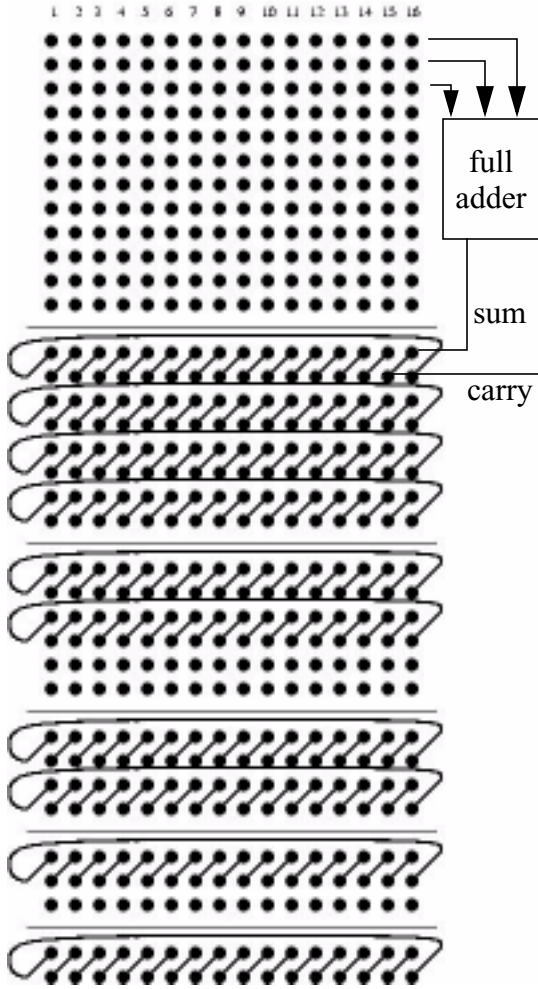


Figure 3: 3-to-2 Reduction Checksumming

approach of reducing as many bits as possible at each stage. Once the number of words in the initial input array is known, a fixed circuit can be synthesized, using number of first stage words as a synthesis parameter.

The total reduction delay of this computation method is approximately

$$D_{3to2} \approx M_{3to2stages} \cdot d_{FA}$$

where  $D_{3to2}$  is the total delay,  $M_{3to2stages}$  is the number of stages required to reduce the original matrix to two rows, and  $d_{FA}$  is the delay of a full adder. Since each full adder in a given stage can function simultaneously, the delay of each stage equals  $d_{FA}$ . Thus, the total delay is  $d_{FA}$  times the number of reduction stages.

In this example, the original 12-row matrix is reduced to eight, then six, four, three, and finally two levels. If we estimate the delay of a full adder to be two gate delays

[6], the total delay of the reduction unit is ten gate delays.

### 3. Circuit implementations

This section describes three basic implementation configurations in which the reduction methods can be implemented.

#### 3.1 Single unit designs

The single unit designs follow the general form of the unit shown in Figure 4. In each case, a block of memory for which the checksum is to be calculated is sent to the M-level reduction unit. The reduction unit implements either leveled or 3-to-2 reduction.

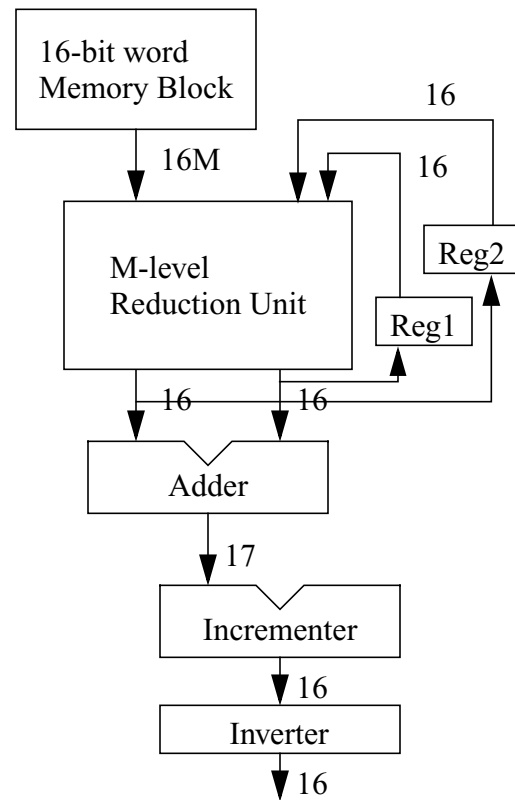


Figure 4: Single Reducer Checksum Unit

If the original block of memory is too large for the fixed-size reduction unit, the largest possible block is reduced and the resulting sum is stored in the registers on the feedback path. The process reiterates, summing remaining rows from the memory array with partial results in the feedback registers, until all memory rows have been summed.

If the original block of memory is smaller than the number of levels that the unit can handle, the implementation can determine which level to send the

memory block to for processing. For example, using leveled reduction, a memory block consisting of seventeen 16-bit words should be sent to the 19-level stage, rather than the 28-level stage, of a 28-level reduction unit. If the implementation does not have this capability, the result will still be correct, but a suboptimal number of stages will be used to process the block. In addition, power can be saved by shutting down unused summation levels and replacing their output with zeroes.

Once the final output rows from the reduction unit have been obtained, they are sent to the adder. If this addition results in an overflow, the sum is incremented. The result is inverted to produce the checksum. Alternatively, to increase performance, two sums can be calculated simultaneously, one assuming an overflow so that the carry in to the adder is set, and one assuming no overflow so that the carry in to the adder is not set. The carry out from the adder can then be used to select (via a MUX) which sum to use as output.

Using delay equations from the previous section, the total delay of this computation method is approximated by

$$D \approx M \cdot d_{FA} + d_{add16} + d_{ovfinv}$$

where  $D$  is the total delay,  $M$  is the number of levels of the reduction unit used,  $d_{FA}$  is the delay of a full adder,  $d_{add16}$  is the delay associated with each 16-bit addition, and  $d_{ovfinv}$  is the delay associated with handling overflow from the final addition and inverting the result.

### 3.2 Multiple unit designs

Alternatively, multiple reduction units can be utilized in parallel as shown in Figure 5. Circuit implementation replaces the single M-Level Reduction Unit of Figure 4 with  $N$  M-Level Reduction Units ( $N = 3$  in Figure 5) and associated feedback registers, operating in parallel on the contents of the 16-bit word Memory Block. Each of these parallel Reduction Units feeds its two output words to a final,  $2N$ -Level Reduction Unit that sums these outputs. In this implementation,  $N_{units}$  reducers simultaneously process  $M$  16-bit words, and the two word outputs from each of these units are subsequently sent to a  $(2N_{units})$ -level reducer. The two output words of this unit are sent to an adder, incrementer, and inverter, similar to the output rows of the reducer in the single unit design.

### 3.3 Partial reduction hybrid designs

A reduction unit can also be used in conjunction with an existing checksum unit, in the case that resources are limited but some acceleration is required. In this

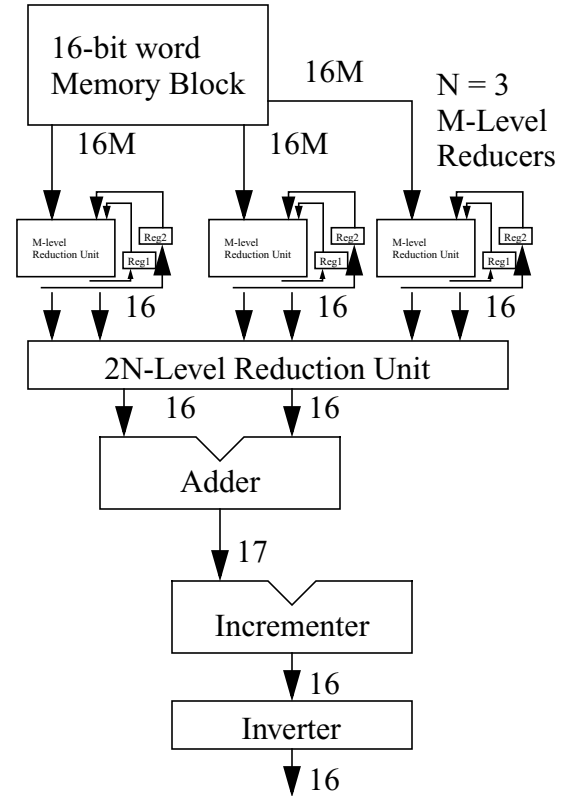


Figure 5: Multiple Reducer Checksum Unit

configuration, shown in Figure 6, an M-to-k level reducer is used to process M-k rows of the input matrix, producing a k-row result. In an M-to-k level reducer, the M-row input is reduced as normal until the stage at which there are k remaining rows is reached. The k remaining rows make up the output of the unit. This output is appended to the remaining rows of the input matrix, if there are any, and either sent back through the reduction unit, or sent to the input of an existing checksum unit for further processing.

## 4. Results and discussion

This section presents theoretical results for several checksumming methods for a given example, as well as a discussion of those methods.

### 4.1 An example

In this example, the checksum of a 160-word block of memory is computed. 16-bit two-level carry-look-ahead adders are used where appropriate, and the total delay of this type of adder is nine gate delays. For each of the conventional implementations, it is assumed that the overflow additions can be performed in parallel with the data word additions. Total delay results are given in terms of gate delays, and gate counts are also given. The

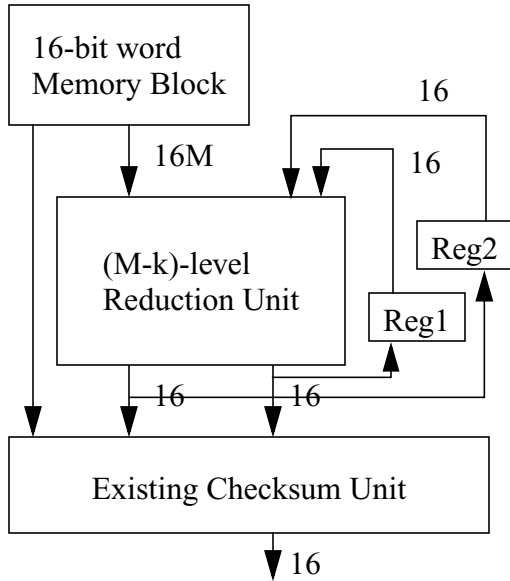


Figure 6: Hybrid Checksum Unit

results appear in Table 1.

Note that the gate count for each hybrid design includes the gates attributed to the checksum unit which is assumed to exist. The checksum unit used in each hybrid design is the one-adder conventional implementation. In both hybrid cases included in the table, the designs provide speedup in the range of 2.5 to 3.5 over the conventional design by utilizing partial reduction.

The eighty-adder conventional implementation represents the best performance that can be obtained using a conventional method in this case. It provides a speedup of about 16 over the single adder implementation, however it requires 76 times as many logic gates. In comparison, the one-pass leveled implementation with three 63-level units achieves a speedup of about 34 over the one-adder conventional implementation, but requires roughly half as many logic gates as the highest performing conventional implementation. The corresponding 3-to-2 implementation achieves the same speedup of about 34 over the single adder model, and requires roughly 40 percent as many logic gates as the highest performing conventional model.

## 4.2 Design considerations

The reduction units are inherently modular, due to the fact that they maintain a 16-bit word size throughout each stage of operation, and due to the commutative property of one's complement addition. Since there are so many design configuration options, there are many

Table 1: Checksum results for 160-word memory block

Implementation	Gates	Delay ( $\Delta_G$ )
Conventional		
one adder	400	1459
two adder	784	748
eighty adder	30736	91
Leveled, single unit		
one pass (211 level)	16928	43
three passes (63 level)	5088	71
eleven passes (19 level)	1568	147
Leveled, three units		
one pass (63 level)	15168	43
two passes (28 level)	6768	53
Leveled hybrid (141-to-42)	8320	565
3-to-2, single unit		
one pass (160 level)	12848	43
three passes (55 level)	4448	73
eleven passes (17 level)	1408	149
3-to-2, three units		
one pass (54 level)	13008	43
two passes (27 level)	6528	53
3-to-2 hybrid (160-to-48)	9360	448

design variables that need to be taken into account when designing a reduction-based checksumming unit.

### 4.2.1 Leveled versus 3-to-2 reduction

When deciding between utilizing the leveled or the 3-to-2 reduction method for a given unit, the following guidelines should be noted:

- If the goal is to minimize the worst-case delay, and the maximum number of input words is known, the 3-to-2 reduction method should be used, because it can be optimized for this case. The series of levels would be determined using  $x_{i-1} = \text{ceiling} ( 2/3 \cdot x_i )$ , and  $x_n$  would

be the maximum number of input words.

- If the goal is to minimize the average delay, or if the maximum number of input words is not known, the leveled reduction method should be used. The series of levels would be determined using  $x_{i+1} = \text{floor} ( 3/2 \cdot x_i )$ , and  $x_0$  would be 2.
- If the unit is going to be pipelined between reduction stages, it may be more effective to utilize the 3-to-2 reduction method, because in general this method reduces more operand bits in the earlier stages and thus reduces the pipeline register requirements.
- Note that in the 160-word example, although the 3-to-2 reduction method provides slightly better speedup than the leveled reduction method, this is not always the case. The 3-to-2-based models in this case are examples of designs optimized for a particular number of input words.

#### 4.2.2 Single unit, multiple unit and hybrid

When deciding amongst utilizing a single-unit, a multiple-unit, or a hybrid design, and when determining the parameters of each unit, these guidelines should be noted:

- Hybrid units such as those presented in the previous section should only be used when there is an existing checksum unit, that existing unit is required to produce the checksum output, and there are not enough resources to include an entire reducer.
- Partial reduction units with smaller M and k values may be more effective than those with larger M and k values because they require less area, they are not limited to process only larger memory blocks, and they can be used iteratively at the input of the existing checksum unit to greatly reduce the pressure on it.
- A tradeoff exists in these designs between overall delay time and total area. A single-pass, single unit design will provide the optimum delay, but will also occupy the most area. If resources are limited, many multiple-pass, multiple-unit, and hybrid designs should be considered to optimally manipulate this tradeoff.

### 5. Related work

In addition to conventional approaches to checksumming previously cited [2, 3, 4], there are two approaches to parallel multipliers that are closely related to our two reduction-based approaches to checksum formation.

The Dadda Reduction Method is an approach to multiplier design that inspires our leveled approach [7,8]. Dadda sums a fixed array of partial products in

one or more summation stages. The leveled stage size of  $x_{i+1} = \text{floor} ( 3/2 \cdot x_i )$  comes from Dadda.

The Wallace Tree Reduction Method is an approach to multiplier design that inspires our 3-to-2 approach [8,9]. Wallace Tree determines stage sizes based on available partial products to be summed. The 3-to-2 stage size of  $x_{i-1} = \text{ceiling} ( 2/3 \cdot x_i )$  comes from Wallace.

### 6. Conclusions

As network standards call for increasingly greater throughput, it is important for the supporting network processors to perform required arithmetic functions efficiently. This paper presents several methods and circuits for efficiently calculating checksums, and provides guidelines for designing checksum units that include reduction trees. Due to the inherent modularity of the reduction units, designs using these units are highly configurable. These units are shown to provide significant performance improvements over conventional checksumming methods.

### 7. References

- [1] W. R. Stevens, *TCP/IP Illustrated, Volume 1 — The Protocols*. Boston, MA: Addison Wesley, 1994.
- [2] L. Spracklin, "Checksum Determination Using Parallel Computations on Multiple Packed Data Elements," United States Patent 5960012, 1997.
- [3] D. Henriksen, "Checksum Generator with Minimum Overflow," United States Patent 6324670, 1999.
- [4] B. Raghunath, "Method and Apparatus to Reduce the Cost of Preparing the Checksum for Out Bound Data in Network Communication Protocols by Caching," United States Patent 6412092, 1999.
- [5] Douglas E. Comer, *Network Systems Design using Network Processors, Agere Version*. Upper Saddle River, NJ: Prentice Hall, 2005. See the Agere Systems CDROM that accompanies the book for detailed documentation on APP550 checksum operations.
- [6] I. Koren, *Computer Arithmetic Algorithms*. Upper Saddle River, NJ: Prentice Hall, 1993.
- [7] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, pp. 349-356, 1965.
- [8] K. Bickerstaff, M. J. Schulte and E. E. Swartzlander, Jr., "Parallel Reduced Area Multipliers," *Journal of VLSI Signal Processing*, Vol. 9, pp. 181-192, 1995.
- [9] C. S. Wallace, "Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, Vol. EC-13, pp. 14-17, 1964.