ISSN: 0975-766X
CODEN: IJPTFI
Available Online through
www.ijptonline.com
Research Article

HARDWARE EVALUATION OF SECOND ROUND SHA-3 CANDIDATES USING FPGA

R.Karthick*[1] and Dr.M.Sundhararajan[2]
[1] Research Scholar, Department of ECE, Bharath Institute of Higher Education and Research Bharath University, Chennai, India,
[2] Dean, Department of ECE, Bharath Institute of Higher Education and Research Bharath University, Chennai, India.
Email: karthickkiwi@gmail.com

Received on: 19.10.2016 *Accepted on: 20.11.2016*

Abstract

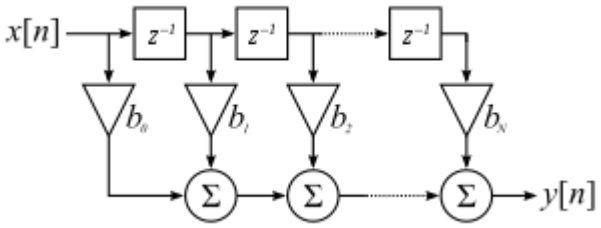
The major goal of our paper is that we propose a platform, a design strategy, and evaluation criteria, for a fair and consistent hardware evaluation of FPGA. For portable and home entertainment audio systems, however the general DSP chip based solution might consume too much power. Then a low-cost and low power solution for this purpose should be sought. There exist two possible approaches that can reduce the chip area and power consumption. One is the simplification of the FIR filter structure, and the other techniques include low power multiplier, adder, memory, supply voltage, etc. In This project we present a low-power FPGA that exploits the low power multiplier design. Power dissipation of integrated circuits is a major concern for VLSI circuit designers. Since power optimization is a major goal of this work we used Window Technique method structure that reduces power consumption over traditional Binary multiplier. Window Technique method is an improved version of tree based FPGA architecture. In addition, the Booth multiplication algorithms has been modified to include 4:2 compressors that are used to add the partial products generated from the multiplier unit. As 4:2 Compressors reduces number of adders needed in the addition process, it reduces number of full adders needed. Hence this further reduces the power consumption. The Booth multiplication algorithm is implemented using Spartan3 FPGA from Xilinx, and it could also implemented using Application specific Integrated Circuits (ASIC) when further reduction in power consumption is required. The result shows that the proposed architecture is faster than the conventional CMOS architecture, along with reduced power consumption.

1. **Introduction:** In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of *finite* duration, because it settles to zero in finite time. This is in contrast

to infinite impulse response (IIR) filters, which may have internal feedback and may continue to respond indefinitely (usually decaying). The impulse response of an Nth-order discrete-time FIR filter (i.e., with a Kronecker delta impulse input) lasts for $N + 1$ samples, and then settles to zero.

FIR filters can be discrete-time or continuous-time, and digital or analog.

Definition



A discrete-tie FIR filter of order N. The top part is an N-stage delay line with N+1 taps. Each unit delay is a z^{-1} operator in Z-transform notation.

The output *y* of a linear time invariant system is determined by convolving its input signal *x* with its impulse response *b*. For a discrete-time FIR filter, the output is a weighted sum of the current and a finite number of previous values of the input. The operation is described by the following equation, which defines the output sequence *y[n]* in terms of its input sequence *x[n]*:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i x[n-i]$$

where:

$x[n]$ is the input signal,

$y[n]$ is the output signal,

b_i are the *filter coefficients*, also known as *tap weights*, that make up the impulse response,

- N is the filter order; an Nth-order filter has $(N+1)$ terms on the right-hand side. The $x[n-i]$ in these terms are commonly referred to as *taps*, based on the structure of a tapped delay line that in many,

N is the filter order; an Nth-order filter has $(N+1)$ terms on the right-hand side. The $x[n-i]$ in these terms are commonly referred to as *taps*, based on the structure of a tapped delay line that in many implementations or block diagrams provides the delayed inputs to the multiplication operations. One may speak of a *5th order/6-tap filter*, for instance. Require no feedback. This means that any rounding errors are not compounded by summed iterations. The same relative error occurs in each calculation. This also makes implementation simpler.The main disadvantage of FIR

filters is that considerably more computation power in a general purpose processor is required compared to an IIR filter with similar sharpness or selectivity, especially when low frequency (relative to the sample rate) cutoffs are needed. However many digital signal processors provide specialized hardware features to make FIR filters approximately as efficient as IIR for many applications.

The impulse response $h[n]$ can be calculated if we set $x[n] = \delta[n]$ in the above relation, where $\delta[n]$ is the Kronecker delta impulse. The impulse response for an FIR filter then becomes the set of coefficients b_n, as follows

Window design method

In the Window Design Method, one designs an ideal IIR filter, then applies a window function to it – in the time domain, multiplying the infinite impulse by the window function. This results in the frequency response of the IIR being convolved with the frequency response of the window function. If the ideal response is sufficiently simple, such as rectangular, the result of the convolution can be relatively easy to determine. In fact one usually specifies the desired result first and works backward to determine the appropriate window function parameter(s). Kaiser windows are particularly well-suited for this method because of their closed form specifications.

Moving average example

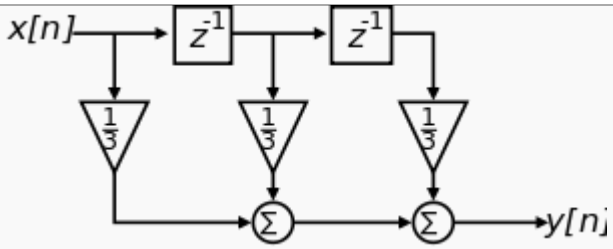


Fig. (a) Block diagram of a simple FIR filter (2nd-order/3-tap filter in this case, implementing a moving average)

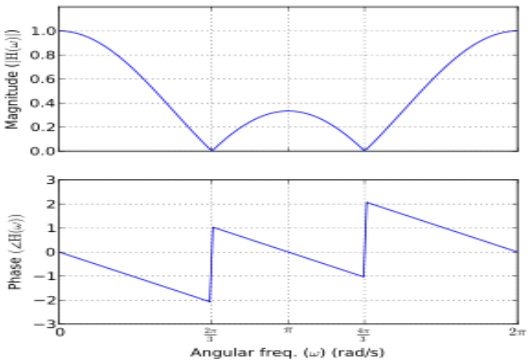


Fig. (b) Amplitude and phase responses

A moving average filter is a very simple FIR filter. It is sometimes called a boxcar filter, especially when followed by decimation. The filter coefficients, $b_0, \ldots, b_N$, are found via the following equation:

$$b_i = \frac{1}{N+1}$$

To provide a more specific example, we select the filter order:

$$N = 2$$

The impulse response of the resulting filter is:

$$h[n] = \frac{1}{3}\delta[n] + \frac{1}{3}\delta[n-1] + \frac{1}{3}\delta[n-2]$$

The Fig. (a) on the right shows the block diagram of a 2nd-order moving-average filter discussed below. To discuss stability and spectral topics we take the z-transform of the impulse response:

$$H(z) = \frac{1}{3} + \frac{1}{3}z^{-1} + \frac{1}{3}z^{-2} = \frac{1}{3}\frac{z^2+z+1}{z^2}$$

Fig. (b) on the right shows the pole-zero diagram of the filter. Zero frequency (DC) corresponds to (1,0), positive frequencies advancing counterclockwise around the circle to (-1,0) at half the sample frequency. Two poles are located at the origin, and two zeros are located at $z_1 = -\frac{1}{2} + j\frac{\sqrt{3}}{2}$, $z_2 = -\frac{1}{2} - j\frac{\sqrt{3}}{2}$.

The frequency response, for frequency ω in radians per sample, is:

$$H\left(e^{j\omega}\right) = \frac{1}{3} + \frac{1}{3}e^{-j\omega} + \frac{1}{3}e^{-j2\omega}$$

Fig. (c) on the right shows the magnitude and phase plots of the frequency response. Clearly, the moving-average filter passes low frequencies with a gain near 1, and attenuates high frequencies. This is a typical low-pass filter characteristic. Frequencies above π are aliases of the frequencies below π, and are generally ignored or filtered out if reconstructing a continuous-time signal. The following figure shows the phase response. Since the phase always follows a straight line except where it has been reduced modulo π radians (should be 2π), the linear phase property is demonstrated.

Binary multiplier

A **binary multiplier** is an electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders. A variety of computer arithmetic techniques can be used to implement a digital multiplier. Most techniques involve computing a set of *partial products*, and then summing the partial products together. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system. Early microprocessors also had no multiply instruction. The Motorola 6809, introduced in 1978, was one of the

earliest microprocessors with a dedicated hardware multiply instruction. It did the same sorts of shifts and adds as a "multiply routine", but implemented in the microcode of the MUL instruction.

As more transistors per chip became available due to larger-scale integration, it became possible to put enough adders on a single chip to sum all the partial products at once, rather than reuse a single adder to handle each partial product one at a time.

Because some common digital signal processing algorithms spend most of their time multiplying, digital signal processor designers sacrifice a lot of chip area in order to make the multiply as fast as possible; a single-cycle multiply–accumulate unit often used up most of the chip area of early DSPs.

A more advanced approach: an unsigned example

p0[7:0] = a[0] × b[7:0] = {8{a[0]}} & b[7:0]
p1[7:0] = a[1] × b[7:0] = {8{a[1]}} & b[7:0]
p2[7:0] = a[2] × b[7:0] = {8{a[2]}} & b[7:0]
p3[7:0] = a[3] × b[7:0] = {8{a[3]}} & b[7:0]
p4[7:0] = a[4] × b[7:0] = {8{a[4]}} & b[7:0]
p5[7:0] = a[5] × b[7:0] = {8{a[5]}} & b[7:0]
p6[7:0] = a[6] × b[7:0] = {8{a[6]}} & b[7:0]
p7[7:0] = a[7] × b[7:0] = {8{a[7]}} & b[7:0]

where {8{a[0]}} means repeating a[0] (the 0th bit of a) 8 times (Verilog notation).

For example, suppose we want to multiply two unsigned eight bit integers together: *a*[7:0] and *b*[7:0]. We can produce eight partial products by performing eight one-bit multiplications, one for each bit in multiplicand *a*:

Architectures obtained from this new design technique are more area efficient and have shorter interconnections than the classical Dadda CC for the design of twos complement multipliers.

2. Methodologies

2.1 Existing Methods

More advanced approach: signed integers

If *b* had been a signed integer instead of an unsigned integer, then the partial products would need to have been sign-extended up to the width of the product before summing. If *a* had been a signed integer, then partial product *p7* would need to be subtracted from the final sum, rather than added to it. The above array multiplier can be modified to support two's complement notation signed numbers by inverting several of the product terms and inserting a one to

the left of the first partial product term: There are a lot of simplifications in the bit array above that are not shown and are not obvious. The sequences of one complemented bit followed by non complemented bits are implementing a two's complement trick to avoid sign extension. The sequence of p7 (non complemented bit followed by all complemented bits) is because we're subtracting this term so they were all negated to start out with (and a 1 was added in the least significant position). For both types of sequences, the last bit is flipped and an implicit -1 should be added directly below the MSB. When the +1 from the two's complement negation for p7 in bit position 0 (LSB) and all the -1's in bit columns 7 through 14 (where each of the MSBs are located) are added together, they can be simplified to the single 1 that "magically" is floating out to the left.

```
                                        1  ~p0[7]  p0[6]  p0[5]  p0[4]  p0[3]  p0[2]  p0[1]  p0[0]
                                     ~p1[7] +p1[6] +p1[5] +p1[4] +p1[3] +p1[2] +p1[1] +p1[0]   0
                                  ~p2[7] +p2[6] +p2[5] +p2[4] +p2[3] +p2[2] +p2[1] +p2[0]   0      0
                               ~p3[7] +p3[6] +p3[5] +p3[4] +p3[3] +p3[2] +p3[1] +p3[0]   0      0      0
                            ~p4[7] +p4[6] +p4[5] +p4[4] +p4[3] +p4[2] +p4[1] +p4[0]   0      0      0      0
                         ~p5[7] +p5[6] +p5[5] +p5[4] +p5[3] +p5[2] +p5[1] +p5[0]   0      0      0      0      0
                      ~p6[7] +p6[6] +p6[5] +p6[4] +p6[3] +p6[2] +p6[1] +p6[0]   0      0      0      0      0      0
  1  +p7[7] ~p7[6] ~p7[5] ~p7[4] ~p7[3] ~p7[2] ~p7[1] ~p7[0]   0      0      0      0      0      0      0
 ---------------------------------------------------------------------------------------------------------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]   P[9]   P[8]   P[7]   P[6]   P[5]   P[4]   P[3]   P[2]   P[1]  P[0]
```

Where ~p represents the complement (opposite value) of p.

Booth's multiplication algorithm

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture.

The algorithm

Booth's algorithm examines adjacent pairs of bits of the N-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i, for i running from 0 to N-1, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator P remains unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to P; and where $y_i = 1$ and $y_{i-1} = 0$, the multiplicand times

2^i is subtracted from *P*. The final value of *P* is the signed product. The representation of the multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at $i = 0$; the multiplication by 2^i is then typically replaced by incremental shifting of the *P* accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest *N* bits of *P*. There are many variations and optimizations on these details. The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order –1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

A typical implementation

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let m and r be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in m and r.

1. Determine the values of *A* and *S*, and the initial value of *P*. All of these numbers should have a length equal to $(x + y + 1)$.

1. A: Fill the most significant (leftmost) bits with the value of **m**. Fill the remaining $(y + 1)$ bits with zeros.

2. S: Fill the most significant bits with the value of $(-\mathbf{m})$ in two's complement notation. Fill the remaining $(y + 1)$ bits with zeros.

3. P: Fill the most significant *x* bits with zeros. To the right of this, append the value of **r**. Fill the least significant (rightmost) bit with a zero.

2. Determine the two least significant (rightmost) bits of *P*.

1. If they are 01, find the value of $P + A$. Ignore any overflow.

2. If they are 10, find the value of $P + S$. Ignore any overflow.

3. If they are 00, do nothing. Use *P* directly in the next step.

4. If they are 11, do nothing. Use *P* directly in the next step.

3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let *P* now equal this new value.

4. Repeat steps 2 and 3 until they have been done *y* times.

5. Drop the least significant (rightmost) bit from *P*. This is the product of **m** and **r**.

Find 3 × (−4), with **m** = 3 and **r** = −4, and *x* = 4 and *y* = 4:

- m = 0011, -m = 1101, r = 1100
- A = 0011 0000 0
- S = 1101 0000 0
- P = 0000 1100 0
- Perform the loop four times :

1. P = 0000 110**0 0**. The last two bits are 00.
- P = 0000 0110 0. Arithmetic right shift.

2. P = 0000 011**0 0**. The last two bits are 00.
- P = 0000 0011 0. Arithmetic right shift.

3. P = 0000 001**1 0**. The last two bits are 10.
- P = 1101 0011 0. P = P + S.

P = 1110 1001 1. Arithmetic right shift.

4. P = 1110 100**1 1**. The last two bits are 11.

P = 1111 0100 1. Arithmetic right shift.

The product is 1111 0100, which is −12.

The above mentioned technique is inadequate when the multiplicand is the largest negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is −8). One possible correction to this problem is to add one more bit to the left of A, S and P. Below, we demonstrate the improved technique by multiplying −8 by 2 using 4 bits for the multiplicand and the multiplier:

- A = 1 1000 0000 0
- S = 0 1000 0000 0
- P = 0 0000 0010 0
- Perform the loop four times :

1. P = 0 0000 001**0 0**. The last two bits are 00.
- P = 0 0000 0001 0. Right shift.

2. P = 0 0000 000**1 0**. The last two bits are 10.

- P = 0 1000 0001 0. P = P + S.
- P = 0 0100 0000 1. Right shift.

3. P = 0 0100 000**0 1**. The last two bits are 01.

- P = 1 1100 0000 1. P = P + A.
- P = 1 1110 0000 0. Right shift.

4. P = 1 1110 000**0 0**. The last two bits are 00.

- P = 1 1111 0000 0. Right shift.

2.2 Proposed Technique

Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by:

$$M \times ''0\,0\,1\,1\,1\,1\,1\,0'' = M \times \left(2^5 + 2^4 + 2^3 + 2^2 + 2^1\right) = M \times 62$$

Where M is the multiplicand. The number of operations can be reduced to two by rewriting the same as

$$M \times ''0\,1\,0\,0\,0\,0\text{-}1\,0'' = M \times \left(2^6 - 2^1\right) = M \times 62.$$

In fact, it can be shown that any sequence of 1's in a binary number can be broken into the difference of two binary numb

$$\left(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots\right)_2 \equiv \left(\ldots 1\overbrace{0\ldots 0}^{n}0\ldots\right)_2 - \left(\ldots 0\overbrace{0\ldots 1}^{n}0\ldots\right)_2.$$

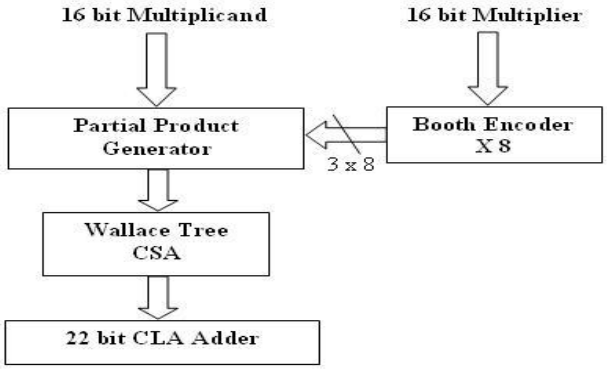
Hence, we can actually replace the multiplication by the string of ones in the original number by simpler operations, adding the multiplier, shifting the partial product thus formed by appropriate places, and then finally subtracting the multiplier. It is making use of the fact that we do not have to do anything but shift while we are dealing with 0s in a binary multiplier, and is similar to using the mathematical property that 99 = 100 − 1 while multiplying by 99.

This scheme can be extended to any number of blocks of 1s in a multiplier (including the case of single 1 in a block). Thus,

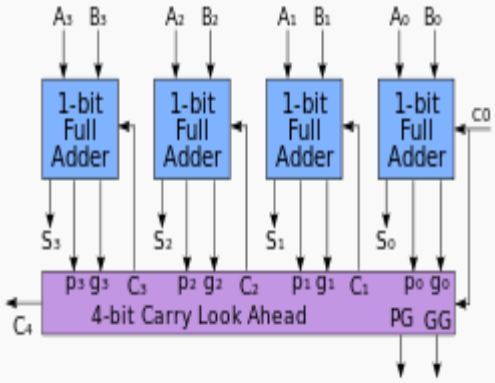
$$M \times ''0\,0\,1\,1\,1\,0\,1\,0'' = M \times \left(2^5 + 2^4 + 2^3 + 2^1\right) = M \times 58$$
$$M \times ''0\,1\,0\,0\text{-}1\,1\text{-}1\,0'' = M \times \left(2^6 - 2^3 + 2^2 - 2^1\right) = M \times 58.$$

Booth's algorithm follows this scheme by performing an addition when it encounters the first digit of a block of ones (0 1) and a subtraction when it encounters the end of the block (1 0). This works for a negative multiplier as well. When the ones in a multiplier are grouped into long blocks, Booth's algorithm performs fewer additions and subtractions than the normal multiplication algorithm.

however; the general-DSP-chip-based solution might be too costly and consume too much power: Then a low-cost and low power solution for this purpose should be sought.



Carry-lookahead adder



4-bit adder with carry look ahead.

A **carry-lookahead adder** (CLA) is a type of adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, *ripple carry adder* for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits. The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits. The Kogge-Stone adder and Brent-Kung adder are examples of this type of adder. Charles Babbage recognized the performance penalty imposed by ripple carry and developed mechanisms for *anticipating carriage* in his computing engines. Gerald Rosenberger of IBM filed for a patent on a modern binary carry-lookahead adder in 1957.

Theory of operation

A ripple-carry adder works in the same way as pencil-and-paper methods of addition. Starting at the rightmost (least significant) digit position, the two corresponding digits are added and a result obtained. It is also possible that there

may be a carry out of this digit position (for example, in pencil-and-paper methods, "9+5=4, carry 1"). Accordingly all digit positions other than the rightmost need to take into account the possibility of having to add an extra 1, from a carry that has come in from the next position to the right.

This means that no digit position can have an absolutely final value until it has been established whether or not a carry is coming in from the right. Moreover, if the sum without a carry is 9 (in pencil-and-paper methods) or 1 (in binary arithmetic), it is not even possible to tell whether or not a given digit position is going to pass on a carry to the position on its left. At worst, when a whole sequence of sums comes to ...99999999... (in decimal) or ...11111111... (in binary), nothing can be deduced at all until the value of the carry coming in from the right is known, and that carry is then propagated to the left, one step at a time, as each digit position evaluated "9+1=0, carry 1" or "1+1=0, carry 1". It is the "rippling" of the carry from right to left that gives a ripple-carry adder its name, and its slowness. When adding 32-bit integers, for instance, allowance has to be made for the possibility that a carry could have to ripple through every one of the 32 one-bit adders.

Carry lookahead depends on two things:

1. Calculating, for each digit position, whether that position is going to propagate a carry if one comes in from the right.
2. Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of 4 digits are chosen. Then the sequence of events goes something like this:

1. All 1-bit adders calculate their results. Simultaneously, the lookahead units perform their calculations.
2. Suppose that a carry arises in a particular group. Within at most 3 gate delays, that carry will emerge at the left-hand end of the group and start propagating through the group to its left.
3. If that carry is going to propagate all the way through the next group, the lookahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group* the lookahead unit is immediately (within 1 gate delay) able to tell the*next* group to the left that it is going to receive a carry - and, at the same time, to tell the next lookahead unit to the left that a carry is on its way.

The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move 4 times as fast, leaping from one lookahead carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group. The more bits in a group, the more

complex the lookahead carry logic becomes, and the more time is spent on the "slow roads" in each group rather than on the "fast road" between the groups (provided by the lookahead. Again, the group sizes to be chosen depend on the exact details of how fast signals propagate within logic gates and from one logic gate to another. For very large numbers (hundreds or even thousands of bits) lookahead carry logic does not become any more complex, because more layers of supergroups and supersupergroups can be added as necessary. The increase in the number of gates is also moderate: if all the group sizes are 4, one would end up with one third as many lookahead carry units as there are adders. However, the "slow roads" on the way to the faster levels begin to impose a drag on the whole system (for instance, a 256-bit adder could have up to 24 gate delays in its carry processing), and the mere physical transmission of signals from one end of a long number to the other begins to be a problem. At these sizes carry-save adders are preferable, since they spend no time on carry propagation at all.

Sizes carry-save adders are preferable, since they spend no time on carry propagation at all.

Carry look ahead method

Carry lookahead logic uses the concepts of *generating* and *propagating* carries. Although in the context of a carry lookahead adder, it is most natural to think of generating and propagating in the context of binary addition, the concepts can be used more generally than this. In the descriptions below, the word *digit* can be replaced by *bit* when referring to binary addition.

The addition of two 1-digit inputs *A* and *B* is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit does not carry (2+7=9)). In the case of binary addition, $A + B$ generates if and only if both *A* and *B* are 1. If we write $G(A, B)$ to represent the binary predicate that is true if and only if $A + B$ generates, we have:

$$G(A, B) = A \cdot B$$

The addition of two 1-digit inputs *A* and *B* is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum. In the case of binary addition, $A + B$

propagates if and only if at least one of *A* or *B* is 1. If we write $P(A,B)$ to represent the binary predicate that is true if and only if $A + B$ propagates, we have:

$P(A,B) = A + B$

Sometimes a slightly different definition of *propagate* is used. By this definition *A + B* is said to propagate if the addition will carry whenever there is an input carry, but will not carry if there is no input carry. It turns out that the way in which generate and propagate bits are used by the carry lookahead logic, it doesn't matter which definition is used. In the case of binary addition, this definition is expressed by:

For binary arithmetic, *or* is faster than *xor* and takes fewer transistors to implement. However, for a multiple-level carry lookahead adder, it is simpler to use $P'(A,B)$. Given these concepts of generate and propagate, when will a digit of addition carry? It will carry precisely when either the addition generates *or* the next less significant bit carries and the addition propagates. Written in boolean algebra, with C_i the carry bit of digit *i*, and P_i and G_i the propagate and generate bits of digit *i* respectively,

The addition of two 1-digit inputs *A* and *B* is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next

Implementation details

For each bit in a binary sequence to be added, the Carry Look Ahead Logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder). Below is a simple 4-bit generalized Carry Look Ahead circuit that combines with the 4-bit Ripple Carry Adder we used above with some slight adjustments:

For the example provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right:

6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate Putting 4 4-bit CLAs together yields four group propagates and four group generates. A Lookahead Carry Unit (LCU) takes these 8 values and uses identical logic to calculate C_i in the CLAs. The LCU then generates the carry input for each of the 4 CLAs and a fifth equal to C_{16}.

The calculation of the gate delay of a 16-bit adder (using 4 CLAs and 1 LCU) is not as straight forward as the ripple carry adder. Starting at time of zero

. To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i \cdot B_i$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

$$P_i = A_i \oplus B_i$$

$$P_i = A_i + B_i$$

The reason why this works is based on evaluation of $C_1 = G_0 + P_0 \cdot C_0$. The only difference in the truth tables between ($A \oplus B$) and ($A + B$) is when both A and B are 1. However, if both A and B are 1, then the G_0 term is 1 (since its equation is $A \cdot B$), and the $P_0 \cdot C_0$ term becomes irrelevant. The XOR is used normally within a basic full adder circuit; the OR is an alternate option (for a carry lookahead only) which is far simpler in transistor-count terms. The Carry Look Ahead 4-bit adder can also be used in a higher-level circuit by having each CLA Logic circuit produce a propagate and generate signal to a higher-level CLA Logic circuit.

Manchester carry chain

The Manchester carry chain is a variation of the carry-look ahead adder that uses shared logic to lower the transistor count. As can be seen above in the implementation section, the logic for generating each carry contains all of the logic used to generate the previous carries. A Manchester carry chain generates the intermediate carries by tapping off nodes in the gate that calculates the most significant carry value. Not all logic families have these internal nodes, however, CMOS being a major example. Dynamic logic can support shared logic, as can transmission gate logic. One of the major downsides of the Manchester carry chain is that the capacitive load of all of these outputs, together with the resistance of the transistors causes the propagation delay to increase much more quickly than a regular carry lookahead. A Manchester-carry-chain section generally won't exceed 4 bits.

5.1 Conclusion

For a complete hardware evaluation, there are plenty of evaluation platforms to be considered. Therefore, fixing one is crucial for conducting a fair and a consistent comparison. In this paper, we propose an evaluation platform and a consistent evaluation method to conduct a fair hardware evaluation of the remaining SHA-3 candidates. This proposal meets the requirements analyzed from actual hash applications and conditions of standard selection. The platform includes a SEBO-GII evaluation board, evaluation software, and appropriate interface definition. Using this method,

we implement all the second-round SHA-3 candidates and obtain the resulting cost and performance factors. This technical study provides a fair and a consistent evaluation scheme. At the end, we hope that by sharing our experience we contribute to the SHA-3 competition and by providing the proposed methodology we influence other similar future selections of the standard cryptographic algorithms.

References

1. X. Wang and H. Yu, "How to break MD5 and other hash functions, "in *Advances in Cryptology—EUROCRYPT 2005*, ser. Lecture Notes in computer Science. New York: Springer, 2005, vol. 3494.
2. X.Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1,"in *Advances in Cryptology—CRYPTO 2005*, ser. Lecture Notes in Computer Science. New York: Springer, 2005, vol. 3621.
3. W. E. Burr, "Cryptographic hash standards: Where do we go from here?," *IEEE Security Privacy*, vol. 4, no. 2, pp. 88–91, 2006.
4. National Institute of Standards and Technology (NIST), CryptographicHash Algorithm ompetition.
5. S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J. Schmidt, and A.Szekely, "Uniform aluation of hardware implementations of theround-two SHA-3 candidates," presented at the 2nd SHA-3 Candidate Conf., Santa Barbara, CA, 2010.

Corresponding Author:

R.Karthick*,

Email: karthickkiwi@gmail.com