University of Dublin

TRINITY COLLEGE

Bitslice Vector Computation

Student: John Lennon (10705273)

Master in Computer Science

CS7092: MCS Dissertation

Supervisor: Dr. David Gregg

Submitted to the University of Dublin, Trinity College

May 2015

DECLARATION

I, John Lennon, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

_____ _____

Name Date

Abstract

This work explores an alternative way of representing arrays of numbers in computers. A *bitslice* representation is considered. This approach takes advantage of bitwise-level parallelism. In the past, bitslicing approaches have served to produce more efficient cryptographic implementations, such as the Data Encryption Standard (DES) symmetric-key algorithm. In general however, bitslicing is rarely seen outside of the realm of cryptography. To help bridge the gap into more mainstream fields, this dissertation focuses on the development of a bitslice arithmetic library. Functions to *add*, *subtract*, and *multiply* bitslice values are built from the ground up, using only bitwise operations. Conversion algorithms are developed to allow conversion from standard arrays to bitslice arrays, and vice versa. Logical and arithmetic shifting routines are also developed. Support for signed bitslice numbers and fixed-point bitslice numbers are added. Interesting properties arise from developing the library, such as the inherent support for arbitrary-sized bitslice values (where computing 9-bit values is no more complicated than computing 8-bit values). This work examines the practical challenges of developing such a library. It examines the performance of bitslice computation compared to that of conventional approaches, by measuring and comparing relative clock cycles for each operation. It is found that in the case of 8-bit values, bitslice computation performs faster than conventional computation when the functionality is mapped from gate-level logic.

Dedicated to my father, John Edward Lennon,

who sadly passed away in 2010.

Acknowledgements

I would like to thank my supervisor, Dr. David Gregg, for offering me such an interesting topic, and for all the help and guidance given throughout. I wish to thank the lecturing staff of the *School of Computer Science & Statistics* for their high standard of teaching. In particular, I would like to thank Dr. Arthur Hughes and Dr. Jonathan Dukes for teaching me the fundamentals of programming in my Junior Freshman year, as well as Course Director Mike Brady for his support throughout. I wish to thank my family and friends for their support and encouragement. I wish to offer a special thanks to a fellow student, Miles McGuire, for his encouragement throughout our degree. Thanks also to Killian Devitt, for his support and hilarity. Finally, I wish to thank Sarah Gallagher for her unending support throughout this journey.

Abbreviations

ALU	Arithmetic Logic Unit
AND	Logical Conjunction
$BLAS$	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Network
DES	Data Encryption Standard
NOT	Logical Negation
OR	Logical Disjunction
$SIMD$	Single Instruction Multiple Data
$SWAR$	SIMD Within A Register
XOR	ExclusiveOR Operation

List of Figures

Contents

Chapter 1

Introduction

This dissertation focuses on an alternative way of representing arrays of numbers in computers. A *bitslicing* approach is considered, which takes advantage of *bitwise*-level parallelism. The most appropriate way of explaining this alternative representation of bits is by way of an example. Consider a sixty-four element array of 8-bit numbers. Another way of representing this entire structure is by instead creating an eight element array of 64-bit values, where the first element of the new array is used to store the first bit of every number from the original array. The second element of the new array is used to store the second bit of every number from the original array, and so on. Effectively this new structure *packs* all of the *i*'th bits together within each of its elements. This representation will henceforth be referred to as a "bitslice" or "bitsliced" *array* or *structure*. A more thorough explanation of the bitslice concept is described in Section 3.1.

While bitslice approaches have been used in the past to develop more efficient cryptographic implementations, such as the Data Encryption Standard (DES), bitslicing is rarely seen outside of the realm of cryptography. There exists no native support for bitslice values on modern computing architectures. Therefore, the primary task of this dissertation is to develop a library of functions that can provide support for arithmetic operations on bitsliced structures. Functions to *add*, *subtract*, and *multiply* bitslice values will be developed, as well as functions to perform logical and arithmetic shifts on bitslice "bits". Support for signed and unsigned bitsliced values will be added. So too will support for fixed-point bitslice values. Conversion algorithms will also be developed to allow users to convert conventional arrays of numbers into bitslice structures, and vice versa. In addition, BLAS (Basic Linear Algebra Subprograms) functions: *vector scale* and *dot product* shall be implemented from a bitslice perspective.

C++ shall be the programming language of choice to develop this library, as it provides sufficient support for bit-level manipulation. Before going any further, I shall outline the structure of this dissertation. I first consider the background of bitslicing and vector programming in my literature review in Chapter 2. Chapter 3 describes the development of the library itself, as well as extra functionality and library enhancements which were added once the core library was written. Chapter 4 covers the findings from the research conducted, as well as a discussion of the measurement results. Finally, a summary of the entire body of work shall be provided in Chapter 5.

Chapter 2

Literature Review

The term *bitslicing* or simply *bitslice* was first coined by Matthew Kwan after a paper entitled *A Fast New DES Implementation in Software* was presented by Dr. Eli Biham at the fourth international workshop of Fast Software Encryption (FSE4), in Haifa, Israel in January 1997 [1]. The "fast new implementation" eluded to in Biham's paper is one which adopts a bitsliced approach to the implementation of the Data Encryption Standard (DES) cryptographic symmetric-key algorithm [2]. In this, Biham treats the processor as a SIMD computer (Single Instruction Multiple Data). What this allowed Biham to achieve was that rather than having to encrypt many 64-bit *words* (the natural unit of data within a computer), one after the other, he instead was able to encrypt the first bit from every word simultaneously, before moving onto the second bit from every word. This was achieved because of the nature of the bitslice representation briefly explained in Chapter 1. It seems a rather novel approach, but it is one which has roots in much older computing architectures.

One such architecture was the Cray CDC6600 from 1964. It is widely regarded as the world's first *supercomputer*. A supercomputer is one which happens to be the world's fastest at performing a certain task [3]. Typically, supercomputers have found their usage in areas such as military research, weather forecasting and simulation, scientific research, and industrial design. All of these areas involve large computations on large data sets. During the 1970's, a supercomputer took the form of a *vector machine*. Vector machines, such as the Cray-1 from 1976, encode multiple operations into single instructions by storing multiple values within a single register. Another word for this approach is SWAR (SIMD Within A Register) [4].

The benefit of this approach is essentially increased *throughput* of operations. While vector machines might seem antiquated in today's world, their impact continues to resonate. Referring back to Biham's approach of encrypting a portion of all 64-bit words simultaneously, we can draw a connection between the architectures of old, and this new, seemingly novel, bitslice approach.

A similar concept to instruction-level parallelism is the notion of bitwise-level parallelism, which is at the core of the bitslicing approach covered in this work. Ultimately, it allows increased throughput, as is the case with SIMD approaches. Other than being adopted in cryptographic endeavours [5], bitslicing is rarely seen outside of this realm. One area where a bitslicing approach may prove useful in the future is in the area of Convolutional Neural Networks (CNN) [6] [7].

In establishing an approach to the development of a bitslice arithmetic library, two core research questions are posed:

1. What are the practical challenges associated with developing a bitslice arithmetic library?
2. How does the performance of bitslice computation compare to that of conventional computation?

In the chapters that follow, I hope to provide answers to these research questions.

Chapter 3

Methodology

This chapter describes the development of the bitslice arithmetic library. In Section 3.1, I explain the concept of bitslicing in depth. In Section 3.2, I describe all of the functions that are contained within the bitslice arithmetic library. Section 3.3 describes the additional functionality that was added once the core library was developed. Further library enhancements are discussed in Section 3.4. Following on from the initial description of bitslicing in Chapter 1, I shall now describe more thoroughly the concept of a bitslice before moving onto the development of the library.

3.1 Bitslicing Explained

It may be helpful to expand upon the description of bitslicing from Chapter 1, with the help of an illustration. Figure 3.1 depicts a standard array of sixteen *uint8_t* elements

stacked on top of one another (shown on the left). To the right of this array is another array which happens to be the bitslice representation of the first array.

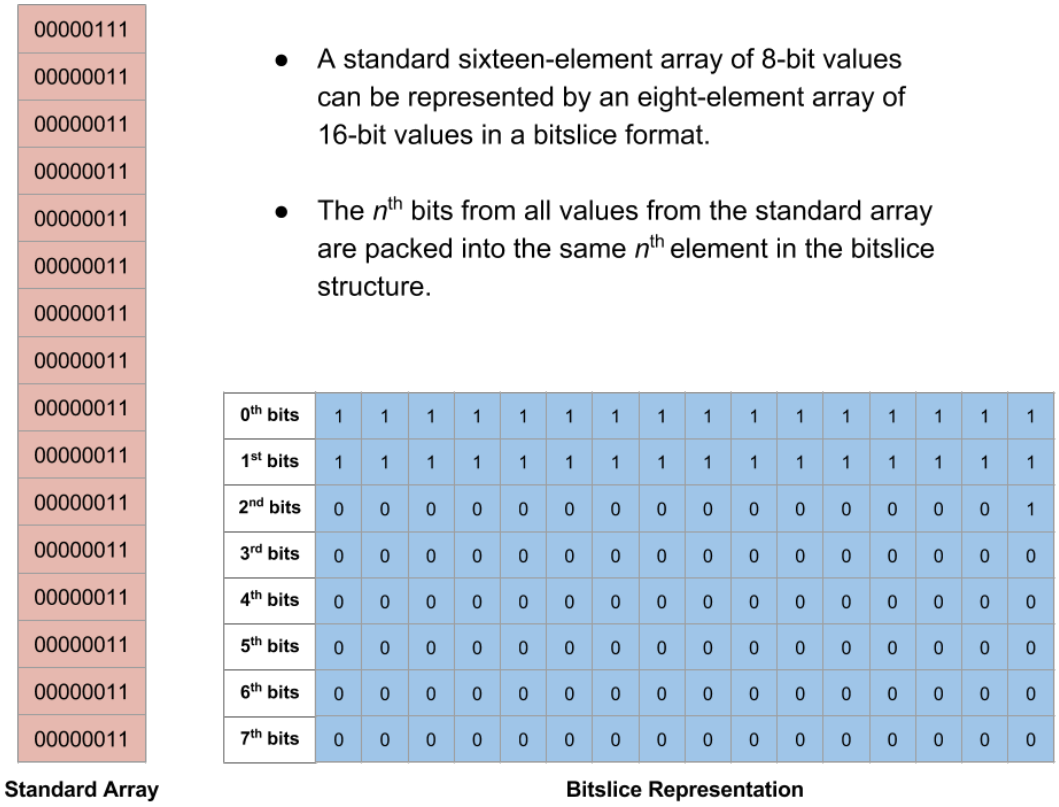


Figure 3.1: Bitslicing Explained

A bitslice value is read vertically in this case. Each vertical *slice* represents one of the sixteen values contained in the original array. The least significant bits (LSB) from all of the numbers from the standard array are positioned at the top of the bitslice structure and the most significant bits (MSB) from all of the numbers from the standard array are positioned at the bottom of the bitslice structure.

3.2 Bitslice Arithmetic Library

Following on from the bitslice structure described in Section 3.1, let us now consider how we can perform arithmetic between two bitslice structures, where each corresponding bitslice value is either added, subtracted, or multiplied. Because a bitslice representation is so different compared to conventional representations, there was no option to use standard arithmetic functions to help build the bitslice library. The use of such standard functions would only yield garbage results. Instead, core functions such as addition and subtraction needed to be built from the ground up, using only bitwise operations (AND, OR, NOT, XOR). In the following sections, I shall outline the development of addition, subtraction, and unsigned multiplication. Support for signed multiplication takes the form of a sophisticated algorithm known as the modified *Baugh-Wooley* algorithm [8] and is covered in detail in Section 3.2.5.

3.2.1 Addition

The bitslice addition function accepts three bitslice structure parameters (two of which act as operands whilst the third acts as a result container). Figure 3.2 shows the logic gate for the *full-adder*. The functionality of the full-adder was mapped to the bitslice addition routine. Rather than operate on just one number at a time, as is the case with conventional approaches, the bitslice adder operates on a portion of all numbers simultaneously. The resulting code is shown in Listing 3.1.

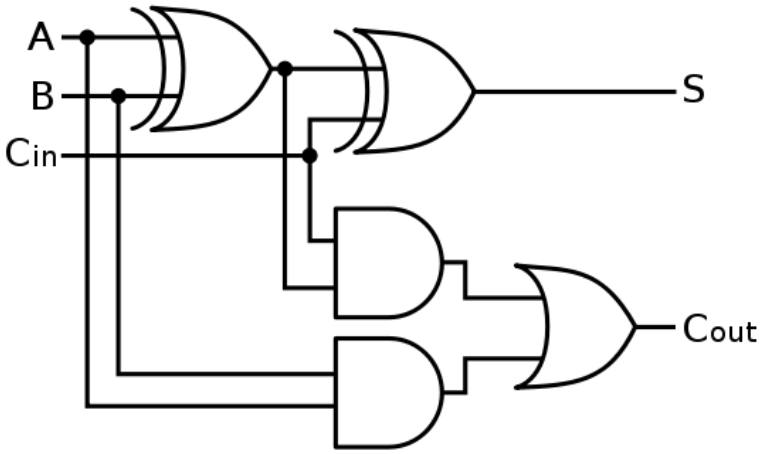


Figure 3.2: Full-Adder Logic Gate

```
template <typename T>
inline void bitslice_add(T * __restrict__ a, T * __restrict__ b, T *
    c, const int size){
    T carry = 0;
    T xxor = a[0] ^ b[0];
    T aand = a[0] & b[0];
    c[0] = carry ^ xxor;
    carry = aand;
    for (int i = 1; i < size; i++){
        xxor = a[i] ^ b[i];
        aand = a[i] & b[i];
        c[i] = carry ^ xxor;
        carry = (carry & xxor) | aand;
    }
}
```

Listing 3.1: Bitslice Addition

3.2.2 Subtraction

A similar approach to that of bitslice addition from Section 3.2.1 was adopted for subtraction, where the gate-level logic shown in Figure 3.3 was mapped into the bitslice subtraction routine, producing the code shown in Listing 3.2.

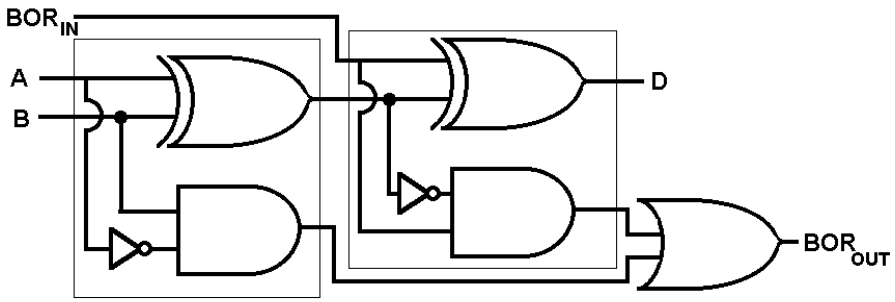


Figure 3.3: Subtractor Logic Gate

```
template <typename T>
inline void bitslice_sub(T * __restrict__ a, T * __restrict__ b, T *
    c, const int size){
     T borrow = 0;
     T xxor = a[0] ^ b[0];
     c[0] = borrow ^ xxor;
     T aand1 = ~a[0] & b[0];
     T aand2 = borrow & ~xxor;
     borrow = aand1 | aand2;
     for (int i = 1; i < size; i++){
          xxor = a[i] ^ b[i];
          c[i] = borrow ^ xxor;
          aand1 = ~a[i] & b[i];
          aand2 = borrow & ~xxor;
          borrow = aand1 | aand2;
     }
}
```

Listing 3.2: Bitslice Subtraction

3.2.3 Bit-Shifting

Bit shifting of bitslice numbers differs dramatically to that of conventional numbers. Applying conventional bit shifting operations to any of the elements of a bitslice structure would yield garbage results. What is required is a shifting of the elements within a bitslice structure, as opposed to the bits themselves. Three bitslice bit-shifting routines were developed:

- Logical Left Shift
- Logical Right Shift
- Arithmetic Right Shift

Logical Left Shift

The following function shifts all the bitslice values (contained within a bitslice structure) left by "numPlaces" which is passed as an argument to the function. I have included two versions of the function in the code snippet in Listing 3.3. This is a good example of the computer science concept of space-time trade-off, where one implementation (of a particular function) contains more lines of code and executes faster than another implementation (of the same function) that contains fewer lines of code.

```
template <typename T>
inline void bitslice_shift_left(T a[], const int size, const int
    numShifts){
    if (numShifts > 0){
        int count = 0;
        while (count < numShifts){
            for (int i = size-1; i >= 1; i--){
                a[i] = a[i-1];
            }
            a[0] = 0;
            count++;
        }
    }
}
/* Alternative (but slower) implementation */
template <typename T>
inline void bitslice_shift_left(T a[], const int size, const int
    numShifts){
    memmove(a+1, a, sizeof(T)*(size-1));
    a[0] = 0;
}
```

Listing 3.3: Logical Left Shift

Logical Right Shift

The following function shifts all the bitslice values (contained within a bitslice structure) right by "numPlaces" which is passed as an argument to the function.

```
template <typename T>
inline void bitslice_shift_right(T a[], const int size, const int
    numShifts){
    if (numShifts > 0){
        int count = 0;
        while (count < numShifts){
            for (int i = 0; i < size-1; i++){
                a[i] = a[i+1];
            }
            a[size-1] = 0;
            count++;
        }
    }
}
/* Alternative (but slower) implementation */
template <typename T>
inline void bitslice_shift_right(T a[], const int size, const int
    numShifts){
    memmove(a, a+1, sizeof(T)*(size-1));
    a[size-1] = 0;
}
```

Listing 3.4: Logical Right Shift

Arithmetic Right Shift

The following function shifts all the bitslice values (contained within a bitslice structure) right by "numPlaces" which is passed as an argument to the function. Furthermore, this function maintains the sign of the number.

```
template <typename T>
inline void bitslice_arithmetic_shift_right(T a[], const int size,
    const int numShifts){
    if (numShifts > 0){
        int count = 0;
        while (count < numShifts){
            for (int i = 0; i < size-1; i++){
                a[i] = a[i+1];
            }
            a[size-1] = a[size-2];
            count++;
        }
    }
}
/* Alternative (but slower) implementation */
template <typename T>
inline void bitslice_arithmetic_shift_right(T a[], const int size,
    const int numShifts){
    T tmp = a[size-2];
    memmove(a, a+1, sizeof(T)*(size-1));
    a[size-1] = tmp;
}
```

Listing 3.5: Arithmetic Right Shift

3.2.4 Unsigned Multiplication

Shift/Add Multiplication

With conventional approaches to multiplication, values can be shifted left or right to achieve the same result as multiplication and division respectively. For example, shifting a binary value left by n places yields a result which is the same as having multiplied the original value by 2^n. However, this concept does not lend itself to bitslice structures. This is because each and every bit within a bitslice structure is inextricably linked to its adjacent bits which happen to belong to neighbouring, yet disjoint, bitslice values. Therefore, the shifting functions of Section 3.2.3 were used to shift the bitslice bits. Unfortunately however, when one shifts a bitslice value using these functions, its neighbouring bitslice values get shifted also (whether the user wishes this or not). This was not an ideal situation, as it resulted in an extremely inefficient shift/add multiplication routine, where the bitslice structure had to be reset with every pass. It is the sole reason for not fully adopting this approach for unsigned multiplication.

Listing 3.6 shows the code for the function, which multiplies all bitsliced values contained in bitslice structure A by all bitsliced values contained in bitslice structure B, storing all bitsliced results in bitslice structure C. Volatile copies of A and B were also passed for shifting and resetting purposes.

```
template <typename T>
inline void bitslice_unsigned_mul(T a[], T b[], T a_copy[], T b_copy
    [], T c[], const int size){
    /* Storage for sub-result bitsliced structures. To be OR'd with
    each other at the end */
    std::vector<std::vector<T> > aggregated_results;
    /* Reuseable containers for temporary results */
    T * res = (T *) calloc (size, sizeof(T)*size);
    std::vector<T> sub_result;
    /* A mask to isolate the LSB of the bitsliced value in each pass.
     Also used to mask out unwanted bits that may be present in sub-
    result structures. */
    unsigned mask = 0x1;
    /* Pass this value into memcpy when making copies of 'a' and 'b'
    */
    const int NUM_BYTES_TO_COPY = sizeof(T) * size;
    /* For each bitslice value... */
    for (int i = 0; i < sizeof(T)*8; i++){
        /* Keep track of number of shifts to be performed upon the
    multiplicand. */
        int numShifts = 0;
        while (numShifts < size){
            /* if the LSB of the bitsliced multiplier is set.. */
            if (b_copy[0] & mask){
                /* Shift the multiplicand left by the appropriate
    number of places */
                bitslice_shift_left(a_copy, size, numShifts);
                /* Add the shifted multiplicand to an accumulated
    result. Upon first pass the shifted multiplicand must be added to
    a zeroed structure. */
                if (numShifts == 0)
                    bitslice_add(c, a_copy, res, size);
                else
                    bitslice_add(res, a_copy, res, size);
                /* reset the copy of 'a' (the multiplicand) */
                memcpy(a_copy, a, NUM_BYTES_TO_COPY);
            }

            /* When we reach the end of the bitslice multiplier, mask
     out unwanted bits from 'res' and store into a sub-result
    structure. */
            if (numShifts == size-1){
                /* mask out all garbage values before pushing res
    into sub-results vector */
                for (int j = 0; j < size; j++){
                    res[j] &= mask;
                    sub_result.push_back(res[j]);
                }
```

```
            }
            /* Shift the multiplier right by one bit, for next
    iteration */
            bitslice_shift_right(b_copy, size);
            numShifts++;
        }

        /* Store sub-result into final result vector */
        aggregated_results.push_back(sub_result);
        sub_result.clear();
        /* Update mask for next bitslice value */
        mask <<= 0x1;
        /* Reset the copies of 'a' and 'b' for next bitslice
    multiplication */
        memcpy(a_copy, a, NUM_BYTES_TO_COPY);
        memcpy(b_copy, b, NUM_BYTES_TO_COPY);
    }

    /*
        OR all bitsliced sub-result structures together;

        *|0|0|0   |   0|*|0|0   |   0|0|*|0   |   0|0|0|*   =   *|*|*|*
        *|0|0|0   |   0|*|0|0   |   0|0|*|0   |   0|0|0|*   =   *|*|*|*
        *|0|0|0   |   0|*|0|0   |   0|0|*|0   |   0|0|0|*   =   *|*|*|*
        *|0|0|0   |   0|*|0|0   |   0|0|*|0   |   0|0|0|*   =   *|*|*|*
    */
    for (unsigned i = 0; i < aggregated_results.size(); i++){
        std::vector<T> tmp = aggregated_results.at(i);
        for (unsigned j = 0; j < tmp.size(); j++){
            c[j] |= tmp.at(j);
        }

    }
}
```

Listing 3.6: Shift/Add Multiplication

Repeated Addition

After concluding that the shift/add approach was not a viable one, I then attempted a *repeated addition* approach. Since multiplication can be thought of as a process of repeated addition when dealing with conventional numbers, it seemed reasonable to assume I could reuse my bitslice addition functionality within my multiplication routine. This proved successful. The function is shown in Listing 3.7.

```
template <typename T>
inline void bitslice_mul(T * __restrict__ a, T* __restrict__ b, T *
    __restrict__ c, const int size){
    for (int i = 0; i < size; i++){
        c[i] = a[i] & b[0];
    }
    for (int i = 1; i < size; i++){
        T carry = 0;
        T xxor, aand;
        for (int j = i; j < size; j++){
            T current = a[j-i] & b[i];
            xxor = current ^ c[j];
            aand = current & c[j];
            c[j] = carry ^ xxor;
            carry = (carry & xxor) | aand;
        }
    }
}
```

Listing 3.7: Bitslice Unsigned Multiplication

3.2.5 Signed Multiplication

Signed numbers can be represented using either sign-magnitude, one's complement, or two's complement form. After conducting some research into these alternative forms, two's complement seemed to be the wiser choice to support, since the other two forms contain ambiguities when representing the value zero. Section 10.3 of *Digital Design: a systems approach* covers this in detail [9].

Booth's Algorithm

At this point, three core routines have been developed for the bitslice arithmetic library: addition, subtraction, and unsigned multiplication. The next step would be to add support for two's complement signed number multiplication. The first approach considered was to adopt Booth's multiplication algorithm [10]. Booth's algorithm involves the comparison of all adjacent bits of a predetermined value. Based on the outcome of each comparison, one of two predetermined values gets added to a result value and is then shifted right. Once all comparisons are made, the end result is the product. This was not a viable option for the bitslice library due to two main factors. Firstly, dense logic was required to compare adjacent bits within an individual bitslice value. Secondly, shifting of a bitslice value was required with every pass. These two factors combined made this approach untenable. An approach known as the modified *Baugh-Wooley* algorithm [8] was selected instead to provide the necessary support for two's complement signed number multiplication, which will now be discussed in the following section.

Modified Baugh-Wooley Algorithm

The bitslice signed multiplication routine is shown in Listing 3.8. The algorithm itself is known as the modified *Baugh-Wooley* algorithm and is fully described in Chapter 11 of *Computer Arithmetic: Algorithms and Hardware Designs* by Parhami Behrooz [8]. Table 3.1 illustrates the algorithm for two 4-bit operands: *a* and *b*. One must ensure that the result container (*p* in this case) is twice the size of the operands. The algorithm focuses on performing logical conjunctions between bits from *a* and from *b*. This results in a set of *partials* which are added together to produce the result. Logical negation occurs in the most significant bit (MSB) of each partial except for the final partial. The logical conjunctions contained in the final partial are all negated except for the MSB.

Table 3.1: Modifed Baugh-Wooley Algorithm

			1	~b0&a3	b0&a2	b0&a1	b0&a0
			~b1&a3	b1&a2	b1&a1	b1&a0	0
		~b2&a3	b2&a2	b2&a1	b2&a0	0	0
1	b3&a3	~b3&a2	~b3&a1	~b3&a0	0	0	0
p7	p6	p5	p4	p3	p2	p1	p0

```
template <typename T>
inline void bitslice_signed_mul(T * __restrict__ a, T * __restrict__
    b, T * __restrict__ c, const int size){
     for (int i = 0; i < size-1; i++){
         c[i] = a[i] & b[0];
     }
     c[size-1] = ~(a[size-1] & b[0]);
     c[size] = -1;
     T * partial = (T *) calloc(size*2, sizeof(T));
     for (int i = 1; i < size-1; i++) {
         partial[i-1] = 0;
         for (int j = i; j < (i+size-1); j++) {
             partial[j] = a[j-i] & b[i];
         }
         partial[i+size-1] = ~(a[size-1] & b[i]);
         bitslice_add(c, partial, c, size*2);
     }
     partial[size-2] = 0;
     for (int j = size-1; j < size*2-2; j++){
         partial[j] = ~(a[j-size-1] & b[size-1]);
     }
     partial[size*2-2] = a[size-1] & b[size-1];
     partial[size*2-1] = -1;
     bitslice_add(c, partial, c, size*2);
     free(partial);
}
```

Listing 3.8: Bitslice Signed Multiplication

3.3 Additional Functionality

3.3.1 Fixed-Point Support

Fixed-point signed multiplication is essentially bitslice signed multiplication followed by a bitslice arithmetic right shift. Both operands are assumed to have the same precision i.e. the same number of decimal places. The initial product from the signed multiplication routine will contain twice the number of decimal places as had the operands, hence the need for an arithmetic right shift to adjust the result to the correct precision. The function is shown in Listing 3.9.

```
template <typename T>
inline void bitslice_fp_signed_mul(T * __restrict__ a, T *
    __restrict__ b, T * __restrict__ c, const int size, const int
    places){
     bitslice_signed_mul(a,b,c,size);
     bitslice_arithmetic_shift_right(c,size*2,places);
}
```

Listing 3.9: Fixed-Point Multiplication

3.3.2 Conversion Algorithms

Two conversion algorithms were developed to allow the user to transition to bitslice computation with greater ease. The first algorithm accepts a pointer to a conventional array of numbers and in return produces a bitsliced representation of that array. The second algorithm does the opposite. These algorithms exhibit *isomorphic* behaviour: when passing a conventional array into the bitslice conversion routine, we then pass the result of that into the conventional conversion routine. What we get back is an identical array to what we had initially passed in. The same is also true if we pass a bitslice structure into the conventional conversion routine, and then pass the result of that into the bitslice conversion routine. What we get back is an identical bitslice array to what we had initially passed in.

```
template <typename T,typename P>
inline void bitslice_convert (std::vector<P> numbers, std::vector<T>
    &bitslices){
     T mask = 1;
     T element = 0;
     int n = 0;
     for (int i = 0; i < sizeof(P)*8; i++){
          for (int j = 0; j < numbers.size(); j++){
               element |= ( (numbers.at(j) & mask) >> i ) << n++;
          }
          bitslices.push_back(element);
          element = 0;
          mask <<= 1;
          n = 0;
     }
}
```

Listing 3.10: Bitslice Conversion

```
template <typename T, typename P>
inline void bitslice_normalize (std::vector<T> bitslices, std::vector
    <P> &numbers){
     T mask = 1;
     P element = 0;
     for (int i = 0; i < sizeof(T)*8; i++){
          for (int j = 0; j < sizeof(P)*8; j++){
               element |= ((bitslices.at(j) & mask) >> i) << j;
          }
          numbers.push_back(element);
          element = 0;
          mask <<= 1;
     }
}
```

Listing 3.11: Bitslice to Conventional Values Conversion

3.3.3 Horizontal Addition

Horizontal addition is the process of adding all of the values contained within an individual structure together. From a conventional approach, this is fairly straight forward, as one can simply iterate through an array using a looping contruct, all the while maintaining a running sum. From a bitslice perspective however, this process is far from trivial. The code in Listing 3.12 shows the implementation. It involves a process of bitslice addition where the operands are the bitslice structure which is passed into the function, and a logically shifted version of that structure. This allows each vertical slice to be added to every other slice in that structure. The sum is contained in the left-most bitslice of the resulting structure. The other slices contain garbage results.

```
template <typename T>
inline void bitslice_horizontal_add(T * __restrict__ a, T *
    __restrict__ result, const int size){
    T * tmp = (T *) malloc (sizeof(T) * size);
    for (int i = 0; i < size; i++){
        tmp[i] = a[i] << 1;
    }
    bitslice_add(a,tmp,result,size);
    int count = 0;
    while (count++ < sizeof(T)*8 - 1){
        for (int i = 0; i < size; i++){
            tmp[i] <<= 1;
        }
        bitslice_add(result,tmp,result,size);
    }
    free(tmp);
}
```

Listing 3.12: Bitslice Horizontal Addition

3.3.4 BLAS Routines

BLAS stands for Basic Linear Algebra Subprograms. It is a collection of mathematical operations which were created to promote modularisation, portability and efficiency of program code [11].

Vector Scale

Vector scale takes an array of values and scales each value in that array by a constant. From a bitslice perspective, the constant must take the form of a bitslice structure. This bitslice structure simply contains the same value for each bitslice. Following this, we scale the bitslice structure by performing bitslice multiplication. The code in Listing 3.13 shows the implementation.

```
template <typename T>
inline void bitslice_blas_scale(T * __restrict__ vec, T *
    __restrict__ scale, T * __restrict__ result, const int size){
    bitslice_signed_mul(vec,scale,result,size);
}
```

Listing 3.13: Bitslice Vector Scale

Dot Product

Considering vectors A and B, the dot product of A and B is the sum of the products of the corresponding values contained in A and B. From a bitslice perspective, this translates to a combination of signed multiplication and horizontal addition. The code in Listing 3.14 shows the implementation.

```
template <typename T>
inline void bitslice_blas_dot(T * __restrict__ a, T * __restrict__ b,
    T * __restrict__ c, const int size){
    T * tmp = (T *) calloc (size*2,sizeof(T));
    bitslice_signed_mul(a,b,tmp,size);
    bitslice_horizontal_add(tmp,c,size*2);
    free(tmp);
}
```

Listing 3.14: Bitslice Dot Product

3.4 Library Enhancements

3.4.1 Generic Types

While initial development and testing took place using explicit types such as ***uint8_t***, ***uint16_t***, ***uint32_t***, and ***uint64_t***, the functions contained in the library were re-factored using generic types by way of C++ templates. This gave the library flexibility in that the user could call the same function using structures comprised of different types. An example of this is shown in Listing 3.15. This also means that as technology advances, larger native conventional data types can still be handled by the library without any change to the code itself.

```
/* Explicit Types */
inline void bitslice_add(uint8_t a[], uint8_t b[], uint8_t c[], const
    int size)
{
  ...
}

/* Generic Types */
template <typename T>
inline void bitslice_add(T a[], T b[], T c[], const int size)
{
  ...
}
```

Listing 3.15: Using Generic Types

3.4.2 Inline Functions

Function inlining was a proactive measure taken to aid faster execution time. By declaring a function *inline*, the compiler can weigh up the cost of whether to make a function call or to simply insert the body of the function in place of its invocation.

3.4.3 Control Flow

The arithmetic library was built using only bitwise operations. No control flow exists in the arithmetic functions themselves, other than looping constructs e.g. *for*-loops. The lack of conditional-statements (*if*-statements, *switch*-statements) means less branching and ensures a more sequential flow of execution.

3.4.4 Compiler Optimisations

The GNU Compiler Collection was utilised to compile the library [12]. The use of the *restrict* command on some of the parameters being passed into the functions informs the compiler that the pointers being passed in point to different locations in memory. The compiler can then remove certain error checks from its compiling sequence, thus enabling a saving to be made in terms of processing time. Figure 3.16 shows this command in use.

```
template <typename T>
inline void bitslice_function(T * __restrict__ a, T * __restrict__ b)
{
    // function body
}
```

Listing 3.16: Restrict Keyword Usage

When compiling the bitslice library, the *-O3* flag was used to invoke all available compiler optimisations. For an exhaustive list of these optimisations, the GNU Compiler Collection should be referenced [12].

```
g++ bitslice.cpp -O3 && ./a.out > output.txt
```

Listing 3.17: Compiling the Bitslice Library

3.4.5 Memory Leak Prevention

Many memory allocations exist within the library. For every allocation of memory from the heap, there needs to be a corresponding deallocation or *free* to ensure a memory leak does not transpire. *Valgrind* is a program that was used to identify any memory leaks that might exist in the library [13]. An example of the output of Valgrind after it was run on the bitslice library is shown in Listing 3.18.

```
/*==15205== Memcheck, a memory error detector
==15205== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et
    al.
==15205== Using Valgrind-3.10.1 and LibVEX; rerun with -h for
   copyright info
==15205== Command: ./a.out

...BITSLICE VECTOR COMPUTATION LIBRARY...

==15205==
==15205== HEAP SUMMARY:
==15205==     in use at exit: 0 bytes in 0 blocks
==15205==   total heap usage: 1,172 allocs, 1,172 frees, 129,040
   bytes allocated
==15205== All heap blocks were freed -- no leaks are possible
==15205== For counts of detected and suppressed errors, rerun with: -
   v
==15205== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
    0)*/
```

Listing 3.18: Valgrind Output

3.4.6 Printing Bitslice Arrays

The *bitset* functionality from the standard library was utilised to allow a user to convert a value to its binary representation. I made a conscious decision to display the bitslice arrays where the elements are stacked on top of one another. This ensured that the user could more easily interpret what values are contained within. Each resulting vertical slice represents each bitslice value. The printing functions are shown in Listing 3.19 and 3.20.

```
template <typename T>
inline void print_bitslice_arrays(T a[], T b[], T c[], const int size
    ){
     for (int i = size-1; i >= 0; i--){
          std::cout << "bit " << i << ":\t"
              << (std::bitset<sizeof(T)*8>) a[i] << "\t"
              << (std::bitset<sizeof(T)*8>) b[i] << "\t"
              << std::endl;
     }
     std::cout << "\nResult;\n";
     for (int i = size-1; i >= 0; i--){
          std::cout << "bit " << i << "\t" << (std::bitset<sizeof(T)
    *8>) c[i] << std::endl;
     }
}
```

Listing 3.19: Printing Bitslice Structures

```
template <typename T>
inline void print_bitslice_fp_array(T a[], const int size, const int
    decimal_place){
     std::cout << "Fixed-Point Output (" << decimal_place << " decimal
     place/s);" << std::endl;
     for (int i = size-1; i >= 0; i--){
          if (i == decimal_place){
               switch(sizeof(T)*8){
                    case 64:
                         std::cout << "dec  :\t
    ................................................................"
   << std::endl;
                         break;
                    case 32:
                         std::cout << "dec  :\t
    ................................" << std::endl;
                         break;
                    case 16:
                         std::cout << "dec  :\t................" << std::
    endl;
                         break;
                    case 8:
                         std::cout << "dec  :\t........" << std::endl;
                         break;
               }
               std::cout<<"bit " << i << ":\t" << (std::bitset<sizeof(T)
    *8>) a[i] << std::endl;
          }else{
               std::cout << "bit " << i << ":\t" << (std::bitset<sizeof(
    T)*8>) a[i] << std::endl;
          }
     }
}
```

Listing 3.20: Printing Bitslice Fixed-Point Structure

Chapter 4

Research Findings

4.1 Experimental Evaluation

4.1.1 Time Stamp Counter

In order to measure the performance of the library's functions, a time stamp counter technique was used. The code in Listing 4.1 allows the contents of the time stamp register to be read into a 64-bit data type [14]. The contents of this register represents the number of elapsed clock cycles since reset.

```
unsigned long long __rdtsc (void) {
  unsigned cycles_low, cycles_high;
  asm volatile ("CPUID\n\t"
    "RDTSC\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
    "%rax", "%rbx", "%rcx", "%rdx");
  return ((unsigned long long)cycles_high << 32) | cycles_low;
}
```

Listing 4.1: Reading the Time Stamp Counter

For every library function that is invoked, two calls are made to capture the contents of the time stamp counter. One call is made before the library function is invoked, and one after. The difference between the two values is the number of clock cycles elapsed since the library function was first invoked until its completion. An example of this is shown in Listing 4.2.

```
start = __rdtsc();
bitslice_add(a, b, sums, size);
stop = __rdtsc();
elapsed_clock_cycles = stop - start;
```

Listing 4.2: Calling rdtsc() before and after library function invocation

4.1.2 Multiple Passes

The reason for wrapping all of the library's function invocations within a loop was to repeatedly capture the elapsed clock cycles and then subsequently extract the median clock cycle for each function once the loop terminated. I opted for one hundred passes. This allowed me to extract a more realistic performance metric, since the processor could be dealing with other computationally-heavy processes at a particular point in time. Opting for the extraction of one elapsed clock cycle measure from one invocation would have most likely yielded an unrealistic measure of that function's performance.

```
const int PASSES = 100;
int count = 0;
while (count++ < PASSES){
  // 1. invoke all library functions
  // 2. store all elapsed clock cycles
}
// 3. extract the median clock cycle for each function
```

Listing 4.3: Multiple Passes

4.1.3 Multiple Environments

The reason for testing in multiple environments was to help reduce the potential risk of receiving false positives in the results. This was a worthwhile endevour since it confirmed to me that the relative performance of the bitslicing approach to the conventional approach was consistent.

Table 4.1: Environments

	Environment A	**Environment B**	**Environment C**
Operating System	Ubuntu 14.04.2 LTS	Windows 7 Professional (SP1) 64-bit	OS X Yosemite 10.10.3
Processor	Intel Xeon E5-2630L v2 @ 2.40GHz	Intel Core i5-3570K @ 3.40GHz	Intel Core i5 2.6GHz
Memory	512 MB	32 GB	8 GB

4.1.4 Bitslice Structure Sizes

As well as testing the capabilities of the bitslice approach on arithmetic functions, different sizes of bitslice values were also tested. The performance over different sizes is shown in Figure 4.8. Of particular interest is the fact that the bitslice library can handle bitslice values of any size. This is an advantage the bitslice approach has over the conventional approach, which is restricted to particular types such as 8-bit, 16-bit, 32-bit, and 64-bit.

4.1.5 Conventional Approach

The conventional approach comprises of standard numerical arrays containing the same values as are present in the bitslice structures. When a particular operation is being considered, the standard arrays are traversed and the corresponding elements act as the operands to that particular operation. Listing 4.4 shows two standard arrays (*alpha* and *beta*) being traversed using a *for*-loop and each of the corresponding elements are being added. This entire process is wrapped in an outer loop. The elapsed clock cycles are recorded for each pass. The median clock cycle is then extracted after the outer loop terminates.

```
/* Conventional Addition */
start = __rdtsc();
for (int i = 0; i < sizeof(base_type)*8; i++){
  normal_add_results[i] = alpha[i] + beta[i];
}
stop = __rdtsc();
/* Store elapsed clock cycles for this pass */
normal_add_times[count] = stop - start;
```

Listing 4.4: Conventional Approach

4.2 Results & Discussion

Figure 4.1 shows the performance of bitslice addition compared with conventional addition. The Y-axis represents the number of clock cycles. The operations were performed on sixty-four 8-bit numbers. This process was repeated over one hundred passes and elapsed clock cycles were recorded with every pass for both approaches. Bitslice addition was developed by mapping the functionality of the full-adder (from gate-level digital logic) to the bitslice addition routine, described in detail in Section 3.2.1. Bitslice addition takes fewer clock cycles to complete than conventional addition. This is due to the bitwise-level parallelism that exists in the bitslice implementation. Effectively, a portion of all the numbers are being operated on simultaneously. Only bitwise operations are used in the implementation. The median clock cycle value for bitslice addition was 1885 cycles. For conventional addition, it was 2175 cycles.

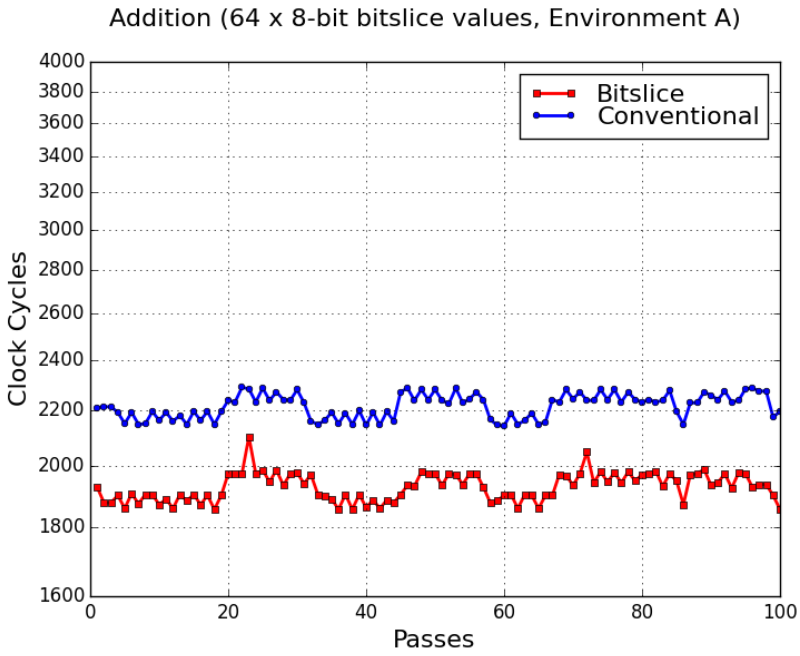
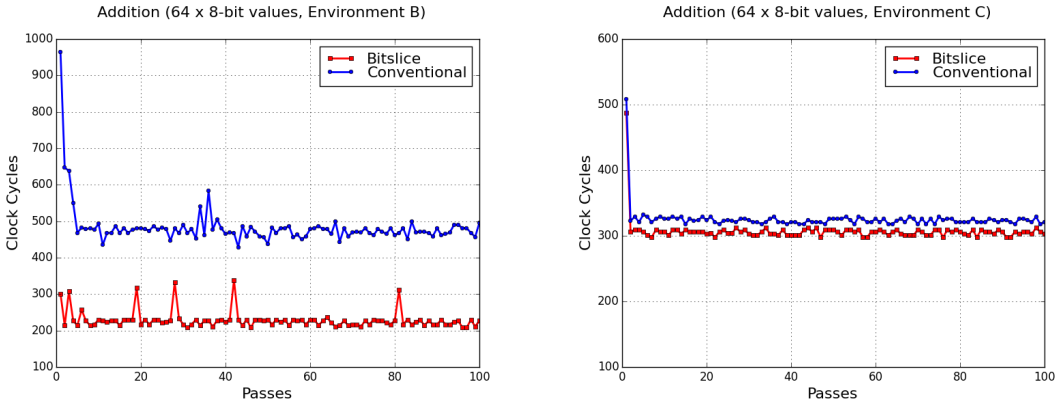


Figure 4.1: Addition (using 8-bit values)

The same approach was adopted within different environments as can be seen in Figures 4.2a and 4.2b. What this shows us is that in general, bitslice addition on 8-bit values performs faster than conventional addition.



(a) Environment B

(b) Environment C

Figure 4.2: Multiple Environments

Figure 4.3 shows the performance of bitslice subtraction compared with conventional subtraction. A similar approach to the implementation of bitslice addition was adopted for subtraction. The functionality of the subtractor gate from gate-level digital logic was mapped to the bitslice subtraction routine. This process is described in detail in Section 3.2.2. As with bitslice addition, bitslice subtraction out-performs conventional subtraction by taking less clock-cycles to complete its work. Again, this is due to the bitwise-level parallelism that exists in the bitslice subtraction implementation. The median clock cycle value for bitslice subtraction was 1845 cycles. For conventional subtraction, it was 2145 cycles.

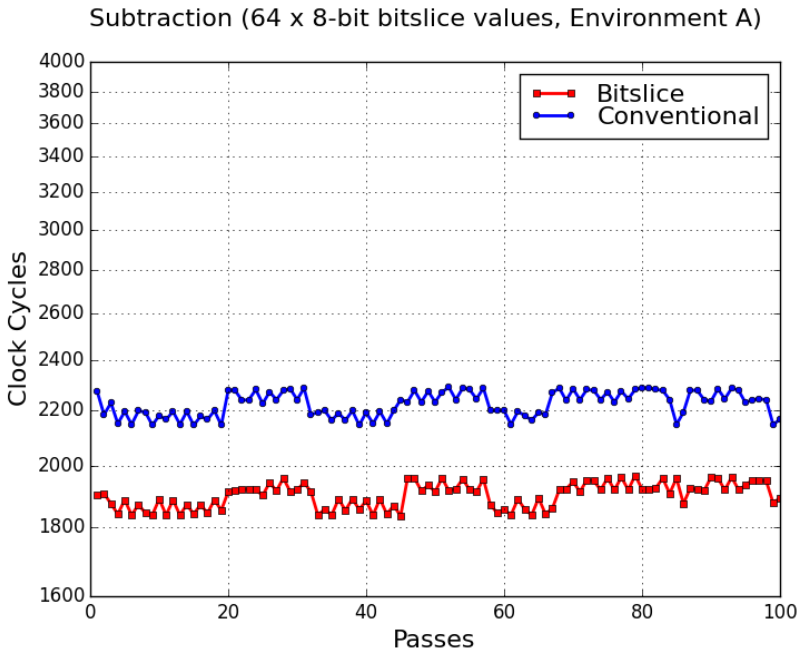


Figure 4.3: Subtraction (using 8-bit values)

Figure 4.4 shows the performance of bitslice unsigned multiplication compared with conventional multiplication. The implementation of unsigned bitslice addition is described in Section 3.2.4. We begin to see a reduction in the performance difference between the two approaches. This is largely due to the fact that this routine is a process of repeated bitslice addition. The median clock cycle value for bitslice unsigned multiplication was 1938 cycles. For conventional unsigned multiplication, it was 2172 cycles.

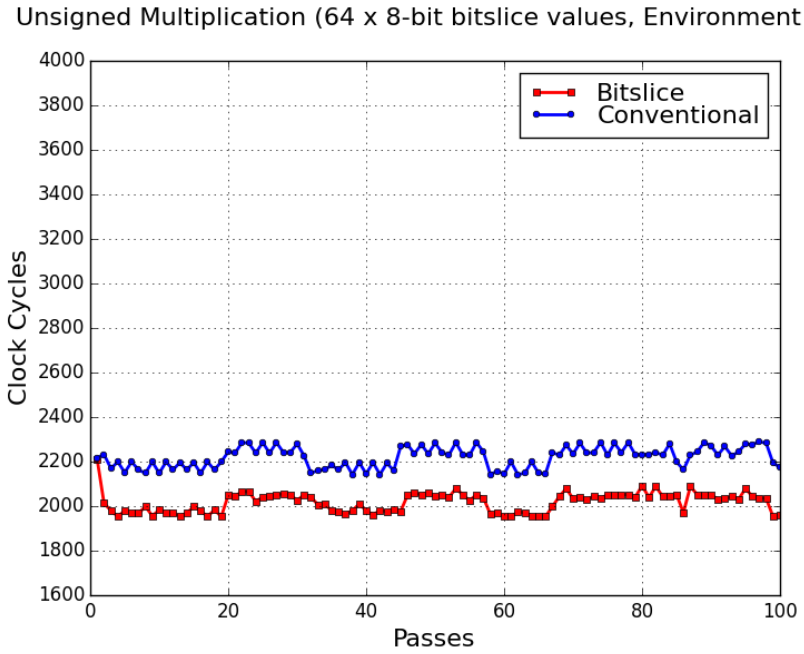


Figure 4.4: Unsigned Multiplication (using 8-bit values)

Figure 4.5 shows the performance of bitslice two's complement signed multiplication compared with conventional multiplication. The modified Baugh-Wooley algorithm was adopted in the implementation of the bitslice routine, the implementation of which is fully described in Section 3.2.5. This is the first bitslice function to move away from an approach of mapping gate-level logic, and the effects of this are seen in the

performance. We see that the bitslice approach is slightly slower than the conventional approach. At this point it is worth taking into consideration that the arithmetic logic unit (ALU) present in modern computing architectures does not support the bitslice format. The bitslicing approach performs reasonably well considering this. The median clock cycle value for bitslice signed multiplication was 2586 cycles. For conventional signed multiplication, it was 2172 cycles.

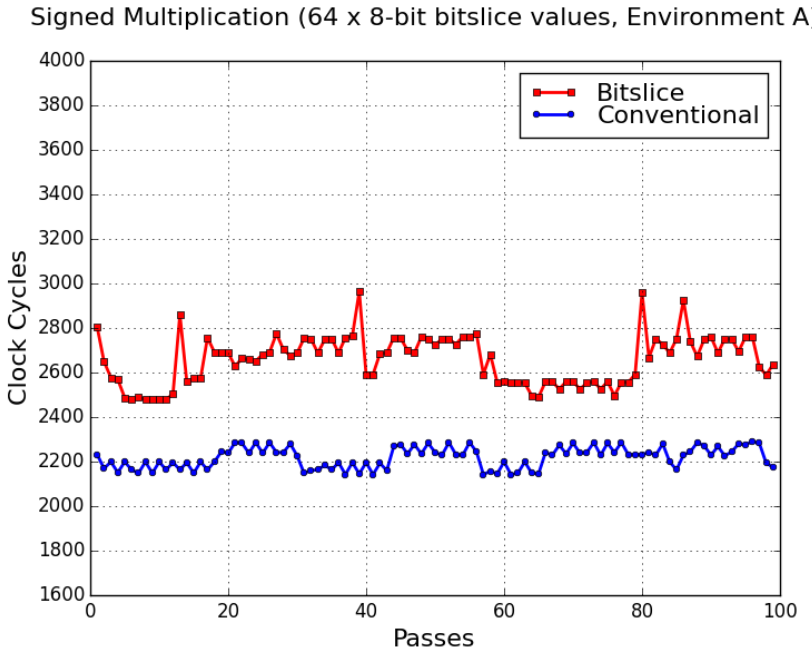


Figure 4.5: Signed Multiplication (using 8-bit values)

Figure 4.6 shows the performance of bitslice vector scale compared with conventional vector scale. The bitslice vector scale implementation is described in Section 3.13. Because the conventional approach is scaling an array by a constant, clearly the constant is being saved in a register which is utilised for all remaining elements of the array. The bitslice approach must represent a constant as a bitslice structure

containing the same bitslice value. The performance of bitslice vector scale is slower as a result. The median clock cycle value for bitslice vector scale was 2562 cycles. For conventional vector scale, it was 2118 cycles.

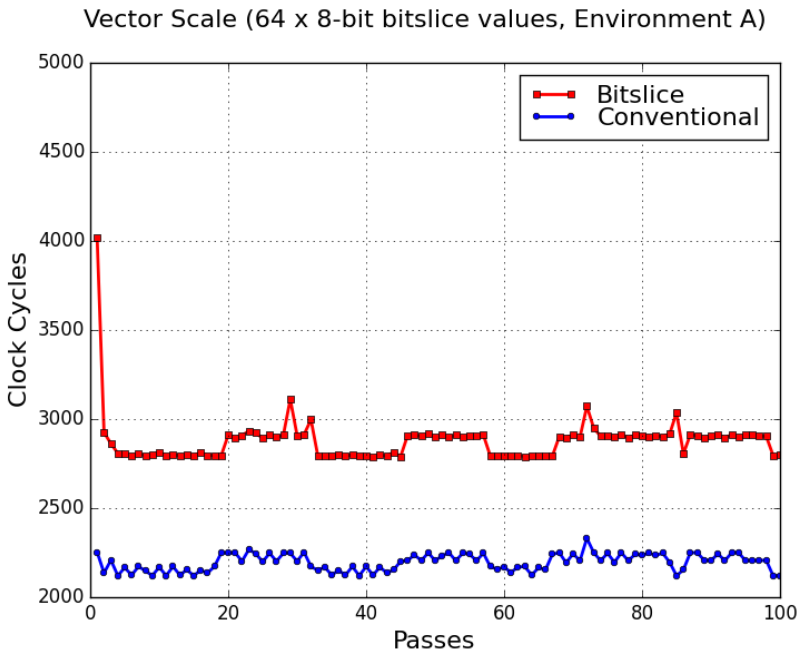


Figure 4.6: Vector Scale (using 8-bit values)

Figure 4.7 shows the performance of bitslice dot product compared with conventional dot product. The implementation of bitslice dot product is described in Section 3.3.4. The fact that bitslice dot product is a combination of bitslice signed multiplication and horizontal addition, it performs slower than its conventional counterpart. The median clock cycle value for bitslice dot product was 7477 cycles. For conventional dot product, it was 2127 cycles.

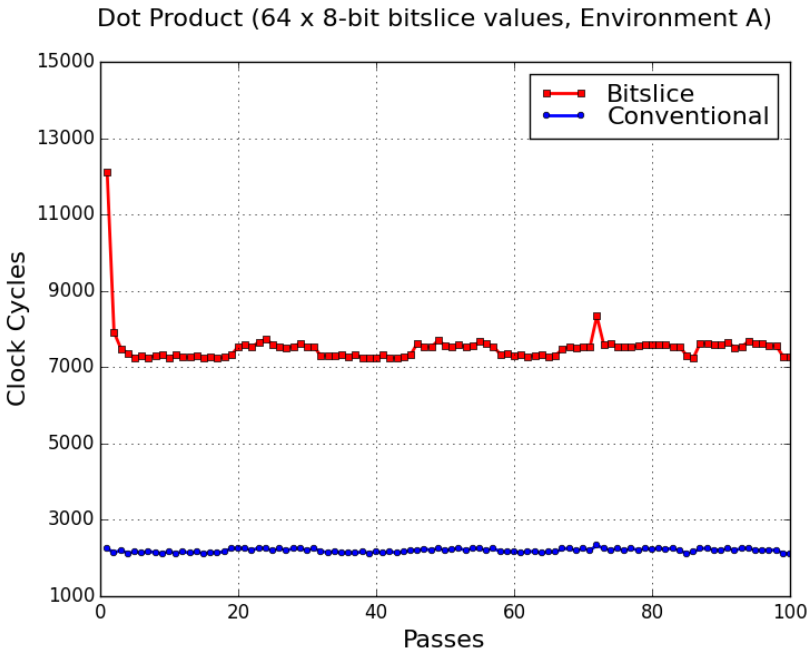


Figure 4.7: Dot Product (using 8-bit values)

Figure 4.8 shows the median clock cycles for bitslice and conventional addition, over multiple bit-sizes. What this figure conveys is that 8-bit, 16-bit, and 32-bit bitslice addition is generally faster than its conventional counterparts. 64-bit bitslice addition is slightly slower than 64-bit conventional addition. An advantage the bitslicing approach has over the conventional approach is that it can handle arbitrary-sized values. This can be seen in the figure where the median clock cycles were also measured for 12-bit, 25-bit, 44-bit, and 83-bit sizes.

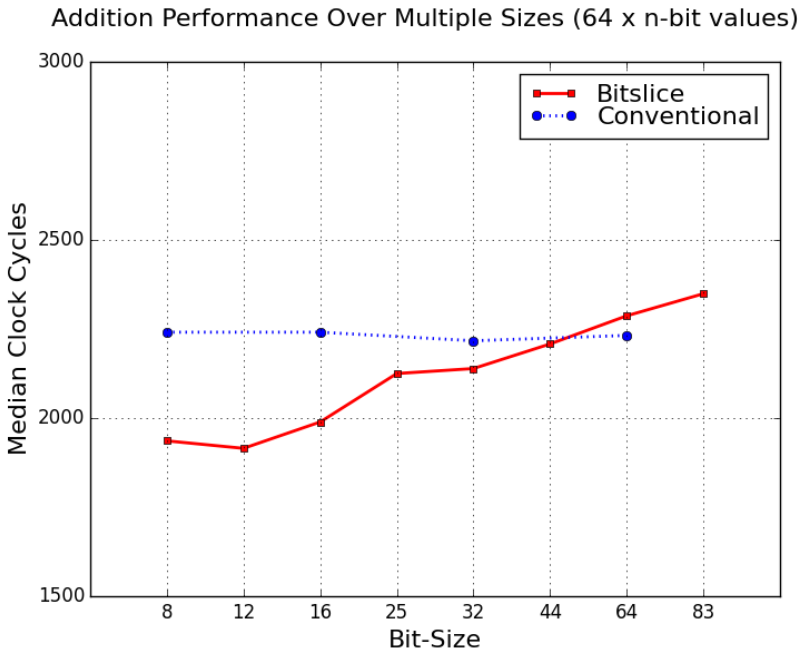


Figure 4.8: Addition Performance Over Multiple Sizes

Figure 4.9 shows the median clock cycles for bitslice and conventional subtraction, over multiple bit-sizes. What this figure conveys is that 8-bit, 16-bit, and 32-bit bitslice subtraction is generally faster than its conventional counterparts. 64-bit bitslice subtraction is slightly slower than 64-bit conventional subtraction. Again, as was the case with Figure 4.8, unusual-sized values were tested.

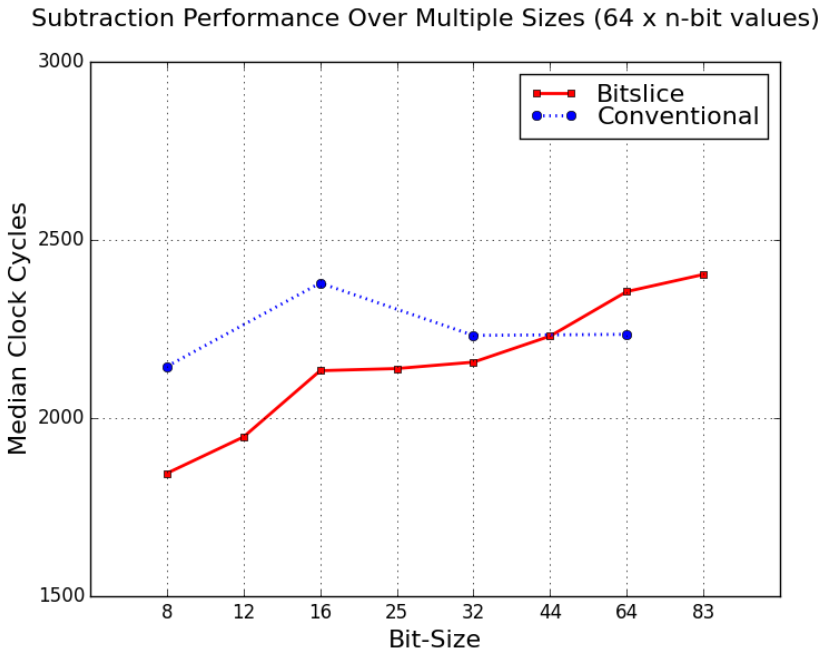


Figure 4.9: Subtraction Performance Over Multiple Sizes

Figure 4.10 shows the median clock cycles for bitslice and conventional unsigned multiplication, over multiple bit-sizes. What this figure conveys is that the time it takes to compute bitslice unsigned multiplication increases dramatically as the bit-sizes increase.

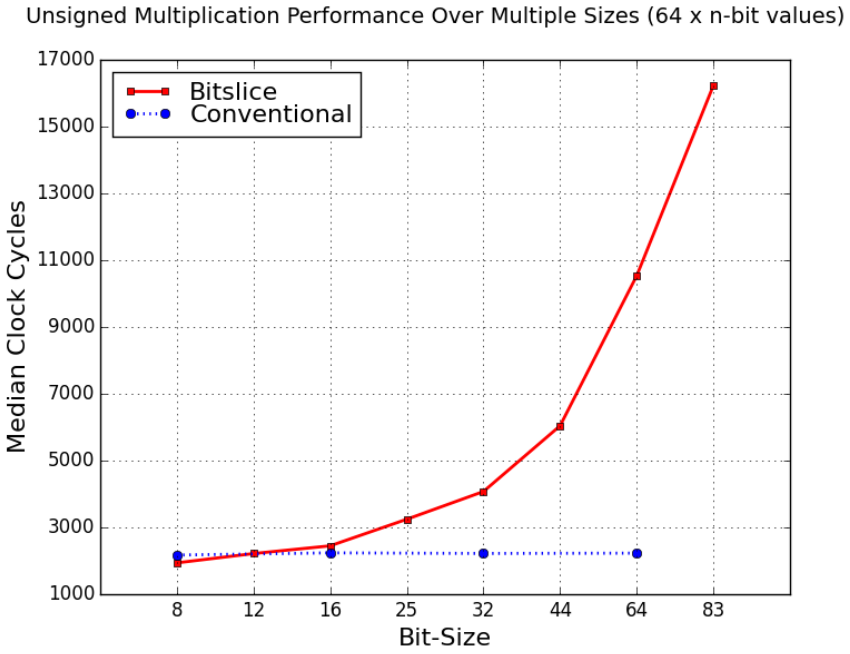


Figure 4.10: Unsigned Multiplication Performance Over Multiple Sizes

Figure 4.11 shows the median clock cycles for bitslice and conventional signed multiplication, over multiple bit-sizes. We see that the performance of bitslice signed multiplication becomes significantly slower as the bit-size increases.

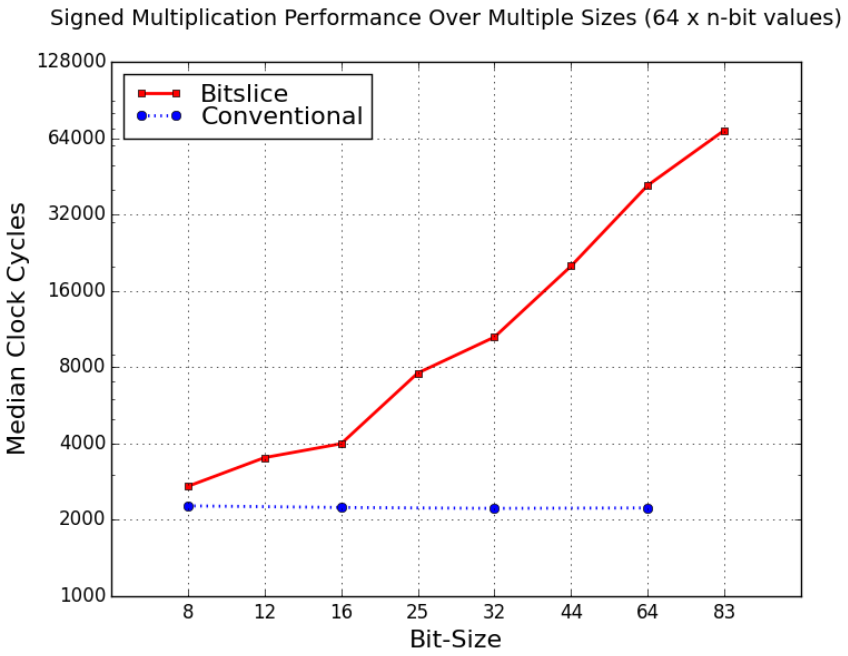


Figure 4.11: Signed Multiplication Performance Over Multiple Sizes

Figure 4.12 shows the median clock cycles for bitslice and conventional vector scale, over multiple bit-sizes. We see that the performance of bitslice vector scale becomes significantly slower as the bit-size increases.

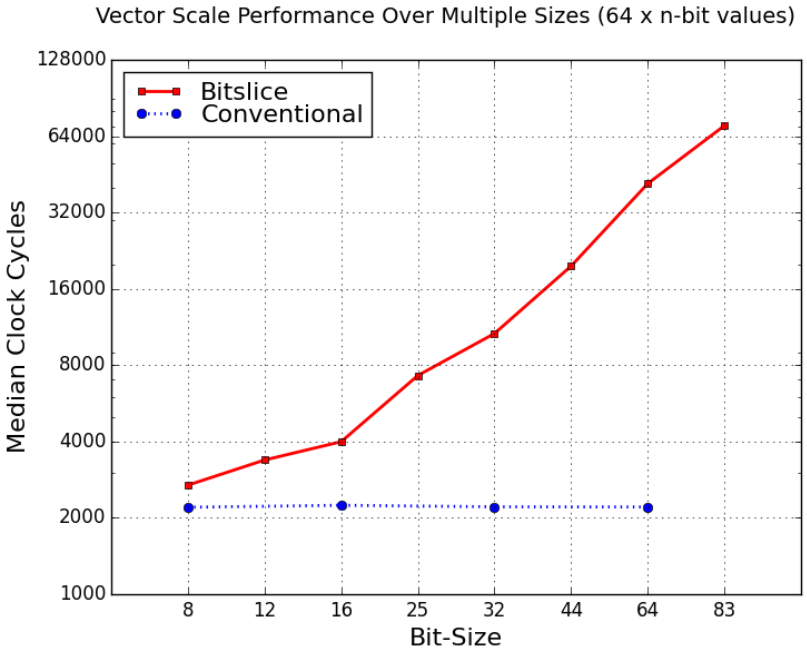


Figure 4.12: Vector Scale Performance Over Multiple Sizes

Figure 4.13 shows the median clock cycles for bitslice and conventional dot product, over multiple bit-sizes. We see that the performance of bitslice dot product becomes significantly slower as the bit-size increases.

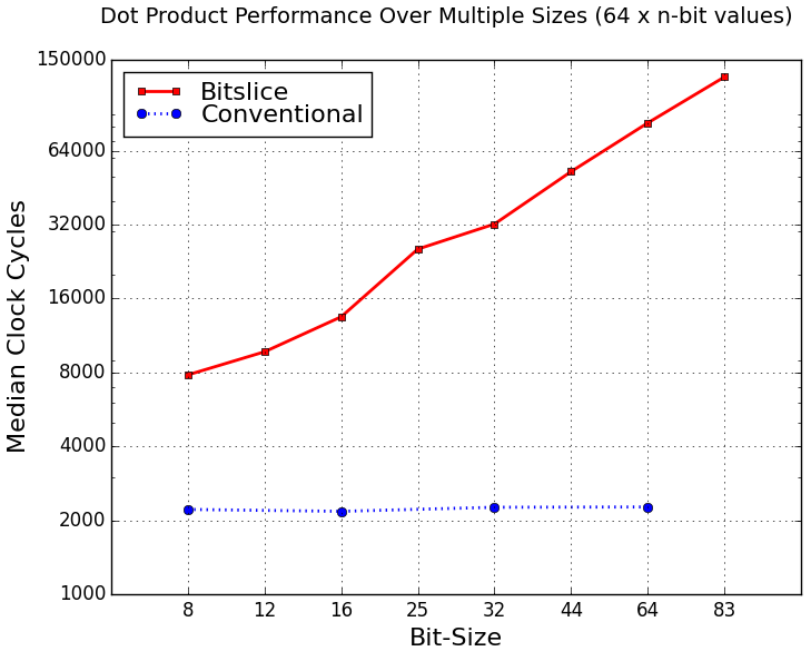


Figure 4.13: Dot Product Performance Over Multiple Sizes

The results are promising for bitslice adoption in the cases of small bit-sizes such as 8-bit values. The highest level of compiler optimisation was used. This optimisation most likely does more for the conventional approach than for the bitsliced approach. I must also point out that the ALU is very likely aiding the conventional approach more so than the bitslice approach. I therefore believe that bitslice vector computations could potentially prove a viable option to those who seek to perform multiple arithmetic operations in parallel. Particularly those who may wish to perform arithmetic on unusual-sized values. Another contributing factor to the speed of the bitslice operations lies in the ommission of control flow. In other words, no conditional statements are present (no *if*-statements, no *switch*-statements). Additionally, the operations are built purely from bitwise operations: which is the language and structure of the processor rather than the language and structure of the programmer.

Chapter 5

Conclusions

In exploring this relatively novel concept, it seemed fitting to develop the mathematical building blocks which could underly future bitslicing endeavours. Two research questions were initially posed in relation to the task at hand:

1. What are the practical challenges associated with developing a bitslice arithmetic library?
2. How does the performance of bitslice computation compare to that of conventional computation?

In response to the first research question, the main challenge lay in the concept of bitslicing itself, as it required one to think differently about number representation. A further challenge involved the need to build addition and subtraction routines from the ground up, using only bitwise operations. This was an essential measure since

no hardware support exists for the bitslice format in modern computing architectures. This process involved a mapping of gate-level logic functionality to the bitslice addition and bitslice subtraction routines respectively.

In terms of performance, bitslice addition and bitslice subtraction perform well compared to their conventional counterparts. This is witnessed across multiple environments. Unsigned multiplication also performs well compared to the conventional approach but only when we deal with 8-bit values. There is quite a noticable performance hit as bit-size increases. When we move to signed multiplication we start to see the conventional approach out-perform the bitslice approach in all cases. Bitslice vector scale and bitslice dot product routines are significantly slower than their conventional counterparts. Another point at which the performance drops for bitslicing is when we move away from mapping gate-level logic functionality. This is seen when one compares the performance of unsigned and signed multiplication of 8-bit values. The former, mapped from gate-level logic, the latter stemming from an algorithm.

I see potential for further work to be carried out in this area. An obvious next step would be in adding a division routine for bitslice values. This would not be a trivial task and one would need to research either the gate-level logic that can provide such functionality or find an appropriate algorithm that achieves the same result. Another area to look at would be adding support for floating-point numbers. Further areas of consideration could involve overflow and underflow detection, similar to the condition control flags seen in assembly language.

A selling point for the bitslice approach lay in the support it has for values of arbitrary size. What I have attempted to achieve here is not only investigate the

practical challenges, and evaluate the performance of the bitslice approach, but to lay the foundations to allow the bitslicing approach to flourish. It was for this reason that I developed conversion algorithms to enable the user to convert standard arrays to bitslice arrays, and back again. Furthermore, the library was developed with a generic-programming mindframe. Templates were adopted to ensure that the functions would not require continuous updates, as larger conventional data-types appear in the future. When larger data-types such as 128-bit and 256-bit do eventually surface, the bitslice library will be ready to embrace them, and more parallelism will be achieved as a result.

Bibliography

[1] Matthew Kwan. Bitslice DES. `http://www.darkside.com.au/bitslice/`, 1998. [Online; accessed 1-April-2015].

[2] Eli Biham. A fast new DES implementation in software. In *Fast Software Encryption*, pages 260–272. Springer, 1997.

[3] Boelie Elzen and Donald MacKenzie. The social limits of speed: development and use of supercomputers. *Annals of the History of Computing, IEEE*, 16(1):46–61, 1994.

[4] Randall J Fisher and Henry G Dietz. Compiling for SIMD within a register. In *Languages and Compilers for Parallel Computing*, pages 290–305. Springer, 1999.

[5] Philipp Grabher, Johann Großschädl, and Dan Page. Light-weight instruction set extensions for bit-sliced cryptography. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 331–345. Springer, 2008.

[6] Olivier Garbe and Christopher Mills. Data bit-slicing apparatus and method for computing convolutions, June 25 1996. US Patent 5,530,661.

[7] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[8] Parhami Behrooz. Computer Arithmetic: Algorithms and Hardware Designs. *Oxford University Press*, 19, 2000.

[9] William J Dally and R Curtis Harting. *Digital Design: a systems approach*. Cambridge University Press, 2012.

[10] Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[11] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[12] The GCC Team. Optimize Options - Using the GNU Compiler Collection (GCC). `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options`, 2015. [Online; accessed 22-March-2015].

[13] Valgrind Developers. Valgrind. `http://valgrind.org/`, 2015. [Online; accessed 31-March-2015].

[14] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation, September*, 2010.