A Graphical SIMD Programming Tool

Abstract

The main aim of this project is to create an application to compile and optimise SIMD programs, from visual rather than textual descriptions. The application should be simple, and easy to use. It should allow the user to be able to intuitively input the desired code in a visual way, using a graph of independent instructions. The application should be independent of the target architecture to make porting code easier, and the output should be an optimised version of the original code, using SIMD instructions.

I show that using such an application can:

i) Reduce programming error;
ii) Improve productivity;
iii) Improve resulting code quality.

Index

ABSTRACT ..... **- 4 -**

1. BACKGROUND ..... **- 5 -**

1.1. PARALLELISM ..... - 5 -
1.2. TAXONOMY OF PARALLEL PROCESSORS ..... - 6 -
1.2.1. SISD – Single Instruction, Single Data ..... *- 6 -*
1.2.2. MISD – Multiple Instruction, Single Data ..... *- 6 -*
1.2.3. SIMD – Single Instruction, Multiple Data ..... *- 7 -*
1.2.4. MIMD – Multiple Instruction, Multiple Data ..... *- 7 -*
1.3. MORE ABOUT SIMD ..... - 8 -
1.3.1. Main Advantage of SIMD ..... *- 8 -*
1.4. BACKGROUND ON THE TWO PROCESSORS CHOSEN ..... - 9 -
1.4.2. Altivec ..... *- 9 -*
1.4.3. Comparison: SSE2 v's Altivec ..... *- 9 -*
1.5. GCC SIMD INTRINSICS: ..... - 9 -
1.6. EXAMPLE SIMD PROGRAM: ..... - 10 -
1.6.1. Original C Code: ..... *- 10 -*
1.6.2. SIMD Code: ..... *- 10 -*

2. INTRODUCTION ..... **- 11 -**

2.1. DEFINITIONS ..... - 11 -
2.1.1. Operation: ..... *- 11 -*
2.1.2. Instruction: ..... *- 11 -*
2.1.3. Intrinsic: ..... *- 11 -*
2.1.4. Link: ..... *- 11 -*
2.2. COMMON ERRORS / PROBLEMS ..... - 11 -
2.3. OPTIMISATION ..... - 12 -
2.4. MOTIVATION ..... - 12 -
2.5. PURPOSE ..... - 12 -
2.5.1. Aims: ..... *- 12 -*
2.6. OUTLINE OF FOLLOWING CHAPTERS ..... - 13 -

3. INTRODUCTION TO APPLICATION: APPLICATION USSAGE ..... **- 14 -**

3.1. INITIAL START UP ..... - 14 -
3.2. WORD CREATION ..... - 14 -
3.3. LOAD / STORE ..... - 15 -
3.4. CREATE AN OPERATION ..... - 15 -
3.5. CALCULATE ..... - 16 -
3.6. OTHER OPTIONS ..... - 16 -
3.6.1. Convert ..... *- 16 -*
3.6.2. Delete A Word / Operation ..... *- 16 -*

4. INTRODUCTION TO APPLICATION: PRODUCTION OVERVIEW ..... **- 17 -**

4.1. INTERFACE DESIGN ..... - 17 -
4.2. CODE GENERATION ..... - 17 -
4.3. TILING AND OPTIMISATION ..... - 17 -
4.4. TESTING ..... - 17 -

5. CODE CREATION ..... **- 18 -**

5.1. OPTIONS PANEL ..... - 18 -
5.2. ERROR CHECKING ..... - 18 -
5.2.1. Word Creation ..... *- 18 -*
5.2.2. Data Type ..... *- 18 -*
5.2.3. Subword Size ..... *- 18 -*
5.3. HOW TO COPE WITH LOADS / STORES ..... - 19 -
5.4. DATA CONVERSION ..... - 19 -
5.5. CREATE AN ORDERED LIST OF THE OPERATIONS ..... - 19 -
5.5.1. Dependencies ..... *- 19 -*
5.6. TARGET INDEPENDENCE ..... - 20 -
5.6.1. Machine Description files ..... *- 20 -*
5.7. GENERATE CODE FOR OPERATIONS NOT SUPPORTED ..... - 22 -

5.7.1. C Library - 22 -

6. LAYOUT AND DESIGN CHOICES - 23 -

6.1. PROBLEM - 23 -
6.2. BASIC DESIGN - 23 -
6.3. OPTIONS PANEL - 23 -
6.3.1. Word Creation Panel - 23 -
6.3.2. Operations Panel - 24 -
6.4. WORD DESIGN - 24 -
6.4.1. Word Design - 24 -
6.4.2. Link Design - 25 -
6.4.3. Link Design Survey - 27 -
6.4.4. Final Choice - 28 -
6.5. MENUS - 28 -

7. TILING AND OPTIMISATION - 29 -

7.1. WHAT IS TILING? - 29 -
7.2. HOW DOES THE APPLICATION MAKE USE OF SUCH TILING? - 30 -
7.3. WHAT ABOUT OPERATIONS DRAWN WHICH ARE NOT "ALL TO ALL"? - 31 -
7.4. WHAT TO DO IF THE INSTRUCTION IS NOT SUPPORTED? - 31 -
7.5. CREATE CODE - 32 -
7.6. OPTIMISATION - 32 -
7.6.1. Basic Operation Tiling - 32 -
7.6.2. Logical Operation Optimisation - 32 -
7.6.3. Permutation Concatenation - 32 -

8. TESTING - 33 -

8.1. CORRECTNESS - 33 -
8.2. DOES THE OUTPUT CODE REALLY PERFORM THE REQUIRED TASK? - 33 -
8.2.1. C Library - 33 -
8.2.2. Special Functions - 33 -
8.2.3. Permutations in SSE2 - 33 -
8.3. UNIVERSAL CODE - 33 -
8.4. SPEED OF OUTPUTTED CODE - 34 -
8.4.1. Logical Optimisation - 34 -
8.4.2. Speed of permutations in SSE2 - 34 -
8.5. EFFICIENCY - 35 -
8.5.1. Permutations - 36 -

9. A LARGER EXAMPLE - 38 -

9.1. INTRODUCTION - 38 -
9.2. IMPLEMENTATION - 38 -
9.2.1. SubBytes - 38 -
9.2.2. ShiftRows - 39 -
9.2.3. MixColumns - 39 -
9.2.4. AddRoundKey - 41 -

10. CONCLUSIONS - 42 -

10.1. FUTURE WORK - 42 -

LIST OF FIGURES

FIGURE 1: GRAPHICAL REPRESENTATION OF MATRIX MULTIPLICATION USING 2 PROCESSORS ..... - 6 -
FIGURE 2: SISD PROCESSOR OVERVIEW ..... - 6 -
FIGURE 3: MISD PROCESSOR OVERVIEW ..... - 7 -
FIGURE 4: SIMD PROCESSOR OVERVIEW ..... - 7 -
FIGURE 5: MIMD PROCESSOR OVERVIEW ..... - 7 -
FIGURE 6: HOW A SIMD MODULE COMPUTES ADDITION ..... - 8 -
FIGURE 7: SCREEN SHOT OF APPLICATION ON START-UP ..... - 14 -
FIGURE 8: EXAMPLE OF A MULTIPLY ADD SATURATED OPERATION, WITH LOADS AND STORES. ..... - 16 -
FIGURE 9: APPLICATION TOOLCHAIN ..... - 17 -
FIGURE 10: EXAMPLE ERROR MESSAGE ..... - 18 -
FIGURE 11: STANDARD GRAPHICAL REPRESENTATION FOR A WORD ..... - 24 -
FIGURE 12: CHOSEN WORD REPRESENTATION. ..... - 25 -
FIGURE 13: TITLE BAR CHOICE ..... - 25 -
FIGURE 14 LINKAGE DESIGN CHOICE 1 ..... - 25 -
FIGURE 15 LINKAGE DESIGN CHOICE 2 ..... - 26 -
FIGURE 16 LINKAGE DESIGN CHOICE 3 ..... - 26 -
FIGURE 17 LINKAGE DESIGN CHOICE 4 ..... - 26 -
FIGURE 18 LINKAGE DESIGN CHOICE 5 ..... - 26 -
FIGURE 19: GRAPH SHOWING THE RESULTS OF THE SURVEY ON LINK TYPE ..... - 27 -
FIGURE 20 TILING EXAMPLE ..... - 29 -
FIGURE 21: EXAMPLE TREE TO BE TILED ..... - 30 -
FIGURE 22: FIRST ROUND OF TILING ..... - 30 -
FIGURE 23: SECOND ROUND OF TILING ..... - 30 -
FIGURE 24: TABLE SHOWING AVERAGE TIMES FOR PERMUTATIONS IN BOTH SSE2 CODE, AND BASIC C CODE ..... - 35 -
FIGURE 25: TABLE SHOWING TIMES TAKEN TO CREATE SMALL PROGRAMS ..... - 35 -
FIGURE 26: TABLE SHOWING TIMES TAKEN TO CONVERT SMALL PROGRAMS ..... - 36 -
FIGURE 27: TABLE SHOWING TIME TAKEN TO CREATE PERMUTATION OPERATIONS BY HAND ..... - 37 -
FIGURE 28: THE 'STATE' IN A) STANDARD MATRIX FORM, B) MODIFIED TO FIT IN A SINGLE WORD ..... - 38 -
FIGURE 29: SHIFT ROWS DIAGRAM AND CORRESPONDING SCREEN SHOT ..... - 39 -
FIGURE 30: MIX COLUMNS OPERATION ..... - 39 -
FIGURE 31: COLUMN MULTIPLICATION BROKEN DOWN ..... - 40 -
FIGURE 32: ALTERNATIVE REPRESENTATION OF FIGURE 31 ..... - 40 -
FIGURE 33: ADD ROUND KEY SCREEN SHOT ..... - 41 -

1. BACKGROUND

1.1. Parallelism

Parallelism is the act of simultaneously performing a number of small tasks at the same time. The main concept behind this is that solving a problem can often be split up into smaller tasks which can be computed at the same time, reducing the overall time of the computation.

A simple example of this is matrix multiplication. Given a n x m matrix, A, and an m x 1 matrix B, with which we want to compute the n x 1 matrix C = AB. The obvious way to compute such a calculation is to step through each calculation, one after the other as the following pseudo code shows:

```
MatrixMultiplication(A,B)
begin
  C ← 0
  for i=0 upto m step 1 do
    for j=0 upto n step 1 do
      C[i] = C[i] + (A[i,j].B[j,1])
  return C
end
```

This method works perfectly, and will always return the correct value, in polynomial time for any given matrices. However, taking advantage of parallelism, and using more than just a single processor, we can decrease the computational time dramatically. For example, if we chose to split the workload over 2 processors we could double the performance:

```
MatrixMultiplicationParallel(A,B)
begin
  C ← 0
  par
    on processor 1
      for i=0 upto m step 1 do
        for j=0 upto n/2 step 1 do
          C[i] = C[i] + (A[i,j].B[j,1])
    on processor 2
      for i=0 upto m step 1 do
        for j=n/2 upto n step 1 do
          C[i] = C[i] + (A[i,j].B[j,1])
  return C
end
```

Graphically, this can be seen as:

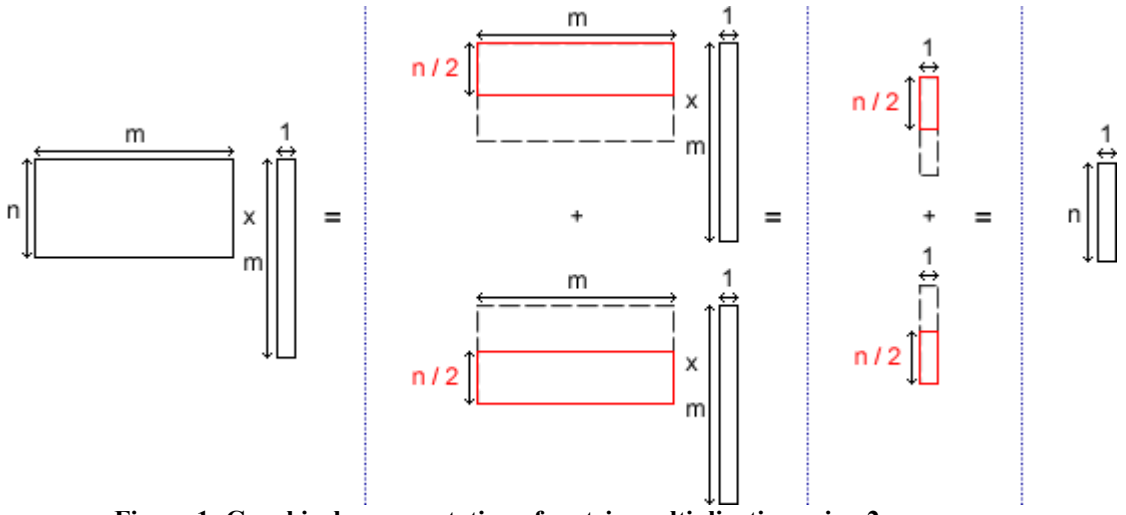


Figure 1: Graphical representation of matrix multiplication using 2 processors

The benefits of such an operation are obvious. Each processor only calculates a percentage of the computation, the results of which are then combined at the end to produce the result. However, it is seldom possible to get quite the performance gain that we might expect on paper. Problems occur as communication between the two (or more) processors is quite costly on time. Another method may be preferable.

1.2. Taxonomy of Parallel Processors

Any computer, whether it is sequential or parallel, operates by executing instructions on data. A stream of instructions tells the computer what to do with a stream of data. However there are different ways in which this occurs, and the most popular classification is one proposed by Michael J, Flynn back in 1966 [13]. It relies, not on the actual structure of the machine, but on the way that the instructions and data flow through it. More specifically, it relies on the number of simultaneous instructions and data streams that flow through the computer during execution. Depending on the number of the streams, computers can be divided into four classes, each of which I shall now briefly introduce.

1.2.1. SISD – Single Instruction, Single Data

This is the standard sequential computer. The computer takes only a single stream of instructions, and operates on only one stream of data. This type of processor does not perform any parallelism.

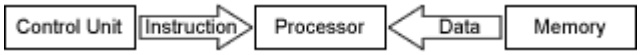


Figure 2: SISD processor overview

An example of use would be to compute the sum of N numbers, the computer would have to gain access to memory N times; each time adding the next number in the sequence to the running total, a total of O(N) operations.

1.2.2. MISD – Multiple Instruction, Single Data

In this model, a number of processors, each with their own control units, execute independent streams of instructions on a common data stream. The type of computation that can be performed efficiently on such a computer is rather specialised, and due to this, no commercial computers exist which have this design.

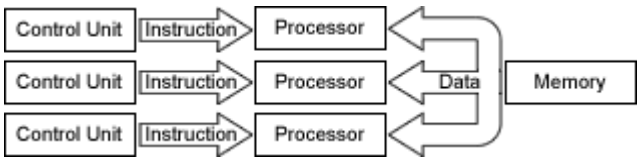


Figure 3: MISD processor overview

An example of a computation that would benefit from such a computer, would be naively calculating if the number N is prime. Using a standard SISD processor, N would be divided by numbers from 2 to √N consecutively, which in the worst case, would take O(√N) steps. However, on a MISD computer with √N processors, this could be done in a single step.

1.2.3. SIMD – Single Instruction, Multiple Data

A SIMD computer consists of N identical processors, each with its own local memory. All processors are controlled by a single control unit, under a single instruction steam; however each has its own data stream. At every step of execution, the control unit transmits the same instruction to each processor simultaneously, each of which then executes the same operation on different data streams.

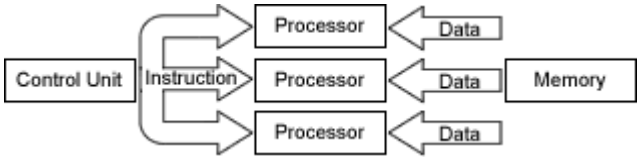


Figure 4: SIMD processor overview

A good example of a use for this is in computer graphics, when inverting the colours of an RGB image. Each pixel of the image is represented by three separate integers, each >= 0 and <= 255. To perform the invert operation, for each pixel the same operation is applied, subtracting each of its colour components from 255. On a SISD computer, each of these would need to be done in turn. However, on a SIMD computer with, for instance 3 processors, each colour could be sent to a separate processor, decreasing the operational time 3-fold.

1.2.4. MIMD – Multiple Instruction, Multiple Data

This is the most general and most powerful class of computers. A computer with N processors, which can perform N instructions on N streams of data at the same time. This also means that the processors can effectively be working asynchronously. The processors are not constrained to run the same program, perhaps running different sub problems of a larger problem. MIMD computers are typically used to solve problems that lack the structure required by a SIMD processor; however algorithms that make use of such a computer are difficult to create. As a result, very few commercial computers make use of such a design.

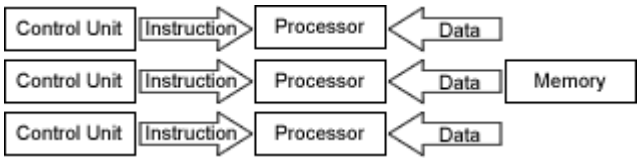


Figure 5: MIMD processor overview

In practice, the vast majority of commercial computers run under the design of SISD. While the benefits of the other designs are clear to see, actually implementing programs on them is harder. For example, early attempts to create a purely SIMD computer failed

due to the fact that it is simply not flexible enough to accommodate general purpose code. To get round this and still gain the benefits of SIMD, almost all commercial processors now use SISD to handle the majority of the code, but with SIMD modules within them, which can handle the special cases when needed.

1.3. More about SIMD

In the past, there were a number of dedicated processors called digital signal processors (DSPs) to handle parallel instructions [14]. These were a special purpose CPU used for digital signal processing, and as such provided ultra fast instruction sequences such as 'shift and add' and 'multiply and add'. However, these were complete processors with their own instruction set, which was often difficult to use. SIMD designs on the other hand, rely on the general purpose portion of the CPU to handle the program details, and the SIMD instructions only handle the data manipulation. This type or processor is much more general purpose, compared to DSPs, which tend to include instructions to handle specific types of data, sound or video. Almost all modern processors now have a SIMD unit in them, which can easily be used. Examples are:

PowerPC's (IBM) AltiVec,
Intel's MMX, SSE, SSE2 and SSE3,
AMD's 3DNow!
SPARCs (Sun) VIS

The fundamental principle behind SIMD is to have large registers split into smaller subwords. SIMD systems typically only include instructions that can be applied to all the data at the same time. **NB:** the operation performs the same action on each subword. This is true for all operations, as while SIMD can be used for operations other than arithmetic (such as permutations) - like arithmetic, they are all moved at the same time.

e.g.: Suppose we have 2 words of length 128bit each, with 4 subwords of length 32bit. figure 6 shows how a SIMD module would compute a + b = c;

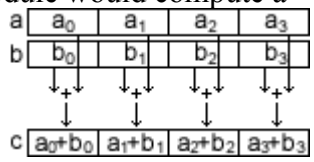


Figure 6: How a SIMD module computes addition

As the diagram suggests, it is calculated by $c_i = a_i + b_i$, where a_i is subword i of a, and $0 \leq i \leq w$, where w is the number of subwords in register a. All these are calculated in parallel in a single step.

1.3.1. Main Advantage of SIMD

The main advantage of SIMD is its ability to take a number of sets of data, and apply the same operation to each one at the same time. A classic example of this data parallelism is inverting an RGB picture to produce its negative. To do this requires an iteration through the pixels (an array of uniform integers) and to perform the same invert operation on each one (one instruction, multiple data points). Whereas typically this would be computed one at a time, a SIMD system could take, for instance, 8 data items at a time, and so vastly decrease the time of the operation. This is the main reason that SIMD instructions are chiefly used in image processing and computer graphics - where it is often the case that the same operation has to be applied to all the data.

Another key advantage with SIMD systems is that the workload can be placed on the programmer rather than the hardware. Whereas other systems that exploit parallelism

(such as super scalar architecture) rely on a large amount of logic code, which looks at the instructions in hand to assess which can be performed at the same time (this is a costly operation in itself, and so reduces the efficiency of the processor). SIMD on the other hand, allows the programmer to specifically optimise the code before hand. This eliminates the need for the system to monitor what needs to be done at the same time, during execution, and so increased performance is acquired.

1.4. Background On The Two Processors Chosen

1.4.1. SSE2

SSE2 was first released by Intel as part of the Pentium 4 chip in 2004. It is an extension of an earlier version of the instruction set, SSE, which itself was meant as a replacement for the older MMX instruction set. SSE2 has since been extended by SSE3, but, since this is still not as widely used, or supported, I decided upon using SSE2 in my application. SSE2 is also supported by AMD's range of processors starting with the Opteron and Athlon 64 chips which were introduced in 2003. As a result, the SSE2 instruction set is one of the most widely compatible out of all the SIMD instruction sets.

1.4.2. Altivec

The Altivec (also known as Velocity Engine and VMX) instruction set was created by Apple Computer, IBM and Motorola. Used in many Apple computers, it was the most powerful SIMD instruction set throughout the late 90's and early 00's. However the introduction of SSE2 and the more recent SSE3, brought with them more instructions, and Altivec lost out as the top instruction set. The shift of power shown most notably as Apple has recently started bringing out computers for the first time using Intel processors. However Altivec does still have its advantages over SSE2.

1.4.3. Comparison: SSE2 v's Altivec

Both SSE2 and Altivec support 8, 16, 32 and 64bit integer, but while SSE2 supports both 64bit double and single precision floating point operations, Altivec only supports 64bit single precision floating point operations. However SSE2 has just eight 128-bit vector registers provided, compared with Altivec's thirty two. Both contain arithmetic, logical, shift, conversion and comparison instructions, although the specifics of each vary. All SSE2 instructions have 2 sources and a single destination, whereas Altivec instructions range from 1 to 3 sources. Perhaps the most notable advantage of Altivec over any of the intel instruction sets, is the inclusion of a permute operation. This is a very powerful operation which enables a programmer to permute each 8bit subword of the whole word in a single instruction, a process that can be quite arduous using SSE2 instructions (see §2.3)

1.5. GCC SIMD Intrinsics:

"*An **intrinsic** is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions*." [15]

GCC comes with a number of libraries that contain SIMD intrinsics. By default, SSE2 and Altivec libraries are often included, (although they can easily be obtained if this is not the case). The main advantage of intrinsics is that they provide a high level language interface to assembly language. In my application, I have decided to make use of this, as it reduces the complexity of the programming, making the resulting code easier to understand and use. Another benefit is that the user does not have to be concerned with register names, register allocation and memory location of data; as when using intrinsics

the compiler manages these things for them, but, the huge gains from using SIMD instructions are still evident.

1.6. Example SIMD Program:

I give now a small example of a program written in normal c code, and then the same program written for a processor supporting SSE2 instructions, making use of the GCC SIMD intrinsics.

Suppose we have two char arrays, of length n, each char being 8bits long, of which we wish to compute the sum. Using basic c code, we can compute the result as in 1.6.1. However, we note that SSE2 instructions take 128bit words as input and that our 8bit chars fit nicely into this. So, loading 16 of these elements into a single word and using SIMD intrinsics, we can operate on all 16 using a single instruction, as in 1.6.2.

1.6.1. Original C Code:

```
void addC( char* A, char* B, char* C, int n ){
     char a, b, c;
     for( int i = 0; i < n; i = i + 1 ){
          a = A[ i ];
          b = B[ i ];
          c = a + b;
          C[ i ] = c;
     }
}
```

1.6.2. SIMD Code:

```
void addSSE2( char* A, char* B, char* C, int n ){
     __m128i a, b, c;
     for( int i = 0; i < n; i = i + 16 ){
          a = _mm_load_si128( A + i );
          b = _mm_load_si128( B + i );
          c = _mm_add_epi8( a, b );
          _mm_store_si128( C + i, c );
     }
}
```

It is clear to see that the resultant SIMD code should run, about, 16 times faster, and so the advantage of SIMD instructions is evident.

2. INTRODUCTION

2.1. Definitions

Throughout this thesis, I will use a number of terms, and for clarity I will briefly describe them now.

2.1.1. Operation:

Is the name given to a process taking one or more words and producing a single outputted word.
E.g. the Add operation: a + b = c, where a, b, c are words.

2.1.2. Instruction:

Each operation has one or more "instructions" which specify the operation to be performed, and the data involved (i.e. word or subword size).
E.g. the Add operation has instructions PADDB, PADDW, PADDD amongst others, each taking 128bit words, but acting on different subword sizes.

2.1.3. Intrinsic:

Is the name given to a function call in C which maps down to the related processor instruction.
E.g. in SSE2 the above instructions have the intrinsic codes _mm_add_epi8, _mm_add_epi16, _mm_add_epi32 respectively.

2.1.4. Link:

The name given to a line drawn in my application by the user, from a source word to a destination word, representing an operation.

2.2. Common Errors / Problems

There are many common errors which can easily occur when a programmer is writing software that uses SIMD parallelism. As mentioned above, with an instruction set such as any of those offered by Intel, each operation has a number of instructions to perform the same task, but on varying subword sizes.

e.g.: The following SSE2 intrinsics all perform an addition operation on 2 words of length 128bit, but each perform the addition in a different way [16].

- _mm_add_epi8 Adds the 16 signed or unsigned 8-bit integers in `a` to the 16 signed or unsigned 8-bit integers in `b`.
- _mm_add_epi16 Adds the 8 signed or unsigned 16-bit integers in `a` to the signed or unsigned 16-bit integers in `b`.
- _mm_add_epi32 Adds the 4 signed or unsigned 32-bit integers in `a` to the 4 signed or unsigned 32-bit integers in `b`.
- _mm_add_epi64 Adds the 2 signed or unsigned 64-bit integers in `a` to the signed or unsigned 64-bit integers in `b`.

The vast majority of Intel's operations take a similar form, and it is all too easy to use the wrong one by mistake. Using Intel's instruction set again, another common area for mistakes is in performing a permutation operation. As apposed to Altivec, Intel's SIMD hardware does not have a permutation operation, and so to perform such an operation requires a number of other operations in a correct and unique sequence for each permutation. There is a huge margin for error here. Even the most simple permutation operation consists of one or more shifts, shuffles and masks. Not only do these have to

be applied in the correct order, but the values / offsets of each of the shifts, shuffles and masks have to be exact or the whole sequence of instructions is wrong Since for each of these between 4 and 16 values have to be calculated accurately and this process repeated for each instruction, the margin for error is clear. Appendix C has an example permutation for each subword size, as a demonstration of the problem faced.

2.3. Optimisation

As well as trying to avoid the functional errors mentioned above, there are many other things to occupy the programmer, such as optimisation. For example, there are times when a sequence of instructions can be replaced by a shorter sequence, or possibly even a single instruction, performing the same task. E.g. Using the Altivec instruction set, a programmer with 3 floating point words A,B and C, with subword size 32-bit, wishing to compute (A*B) + C, would naively use two operations, a multiply followed by an addition. However there is an operation vec_madd(), which will take all 3 words and compute the same result in a single step.

Another type of possible optimisation is scheduling. While clearly the order of the code is important for the program to operate correctly, some operations take longer to calculate than others. So it is possible to re-order the code, so that two (or more) independent operations can run concurrently, decreasing the overall computation time. Unfortunately, commercial processors only contain a single SIMD module, so in general the benefits of such optimisation would not be felt with pure SIMD instructions; however some gain could be achieved if the SIMD code it interleaved with normal C code, which could be run at the same time.

2.4. Motivation

A major problem when programming low level code where performance is critical, is that one cannot always intuitively know which instructions to use to optimise the code. Furthermore, for someone who does not have experience with the instruction sets themselves, finding the functionally correct instruction to perform the desired operation can also be a task in itself. As many instruction sets (i.e. Intel's MMX, SSE, SSE2 and SSE3 series) have many different instructions to perform the same operation, depending on subword size, it can be confusing which one to use, and thus a daunting task.

2.5. Purpose

My application is focused on solving these problems. For a programmer who is comfortable and experienced with programming in low level code, it will free them from thinking about the optimisation of the code, and from a number of other pitfalls as described above. For a person who is not comfortable with this style of programming, it will allow them to easily write a program using a much more visual method; my application will automatically take care of the code creation.

The user should be able to load my application, and using a very intuitive and simple interface, literally draw the program they want. The application will then produce optimised code using associated SIMD instructions, which the user can then use in their program. My application is not aimed at completely removing the need for programming code itself, but as a tool to enable a programmer to create correct, high quality fragments of code much more efficiently.

2.5.1. Aims:

- Supply an intuitive and simple to use application by investigating layout and user interface design.
- Reduce programming error by automatically handling code creation and constant error checks throughout.
- Improve productivity, by eliminating the need for the user to think about the code creation
- Improve resulting code quality by automatically optimising the code being generated

2.6. Outline of Following Chapters

The remainder of the thesis is structured as follows:

Chapter 3: A Brief introduction into the usage of the application.
Chapter 4: A Brief overview of the different parts of the application code to facilitate better understanding of the interconnection of the parts of the application; and a grounding for later discussions where more detail is shown.
Chapter 5: A more detailed look at the different parts of the application, with reasons and design choices.
Chapter 6: A discussion of the design and layout choices made, crucial in producing an intuitive and simple application usable by all.
Chapter 7: A look at the tiling process, where the output code is generated.
Chapter 8: Testing the application to see if it meets with the desired aims.
Chapter 9: A larger example program to show the usefulness of my application.
Chapter 10: Finally, the conclusions made and further work to be carried out.

3. Introduction to Application
Application Usage

Abstract

The application is designed to be as simple as possible to use, in order to create the desired code in the minimum time with the minimum effort. Details on the interface design are presented in **chapter 6.** This section concentrates on simply giving a quick introduction into the usage of the application, and the basic reasons behind each choice.

3.1. Initial Start Up

When the application is first started, the user is faced with a set of menus at the top, a row of options to configure a new word, and a row of operations that are supported by the application, and subsequently, by the processors.

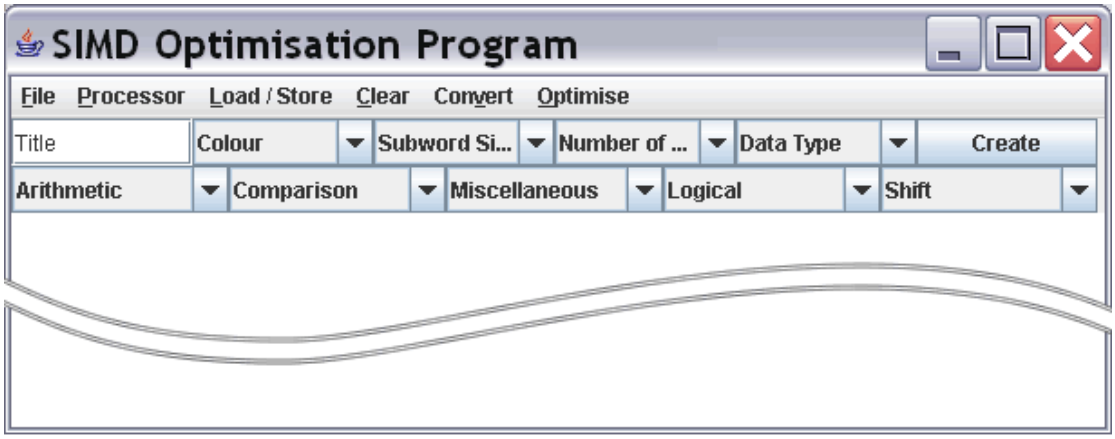


Figure 7: Screen shot of application on start-up

In order to get started the user can first choose a processor with which they want to design code for. The default setting here is "universal" which will allow the user to create code for any type of processor, but at the risk of using code other than SIMD instructions, as operations vary from one processor to the next (more on this in **§5.6**). If a processor is chosen other than the universal one, then the operations will be restricted to those supported by the selected processor.

3.2. Word Creation

The next step is to create the first word. To do this the user simply has to use the first row of options just below the top menus. Each option here is key to defining the correct word, and ensuring the correct outputted code;

- **Title**: Used to create a title for the word, which will also be used as its name in the outputted code.
- **Colour**: Used simply as a visual aid, the word and links from that word will be coloured accordingly.
- **Number of Sub words**: The number of subwords of the word to display in the application. While this has no effect on the outputted code, it can be useful to reduce clutter, and keep the application visually simpler if only using, for example, the lower ½ of every word. It also helps to reduce error, by restricting a user to the maximum number of subwords corresponding to the subword size.
- **Size of Subwords**: Very important, as while, each instruction has no way to know the size of the subwords of each word, and so relies on the user to use the correct instruction for each operation (see §2.3). The size defined here is used by

my application to select the correct instruction for each operation chosen by the user.

- **Data Type**: Just like subword size, the instructions are indiscriminate of which data type is held within a word. But the data type is very important, as there is a huge difference between a floating point and an integer, and likewise, for many operations there is a difference between signed integer and unsigned integer, and as such, my application uses this option to choose the correct instruction for each operation.

3.3. Load / Store

Once a user had created a word, there are a number of options open to them. The first is that they are able to, using the menus at the top, load a value into this word. Choosing "load / save" – "load" and then clicking on the word they wish to load a value into, will highlight the word in yellow, indicating for the duration of the programming that this word will have a value loaded into it. In the outputted code, a basic load template will be created, however it is up to the user to fill in the specific details of the load itself. Likewise, a user can save a word, by choosing "load / save" – "save" and clicking on the word which they wish to save, highlighting the word in green, as can be seen in figure 8.

The decision to leave the loads and stores as templates, rather that asking the user to specify the details, was made for a number of reasons. The main reason is that there are a vast number of possible loads and stores available to the user which are unique for each instruction set. To allow the programs to be truly portable, it would be extremely difficult to correctly, and sufficiently define the loads and stores. Another reason is that the main aim of this application is to increase the efficiency of a programmer. Using my application to create a load or store, selecting the correct one, and then filling in the details, would be a lot more time consuming, than simply filling in the correct details in the template in the outputted code.

3.4. Create an Operation

Alternatively the user may wish to start to create the code of their program. To do this, the user must have at least 2 words in the workspace (even the simplest operation requires at least 2 words; a source and a destination). To create the program, the user then clicks on the operation they wish to perform from the set of provided operations grouped in drop down menus. Then, clicking once on the word which will act as the source of the operation, and then again on the word which will act as the destination. This will create a visual link between the two words, from the bottom of the source word to the top of the destination word. For the vast majority of operations, more than one source word is required, and so the user must repeat the process from a second (and possibly third) word.

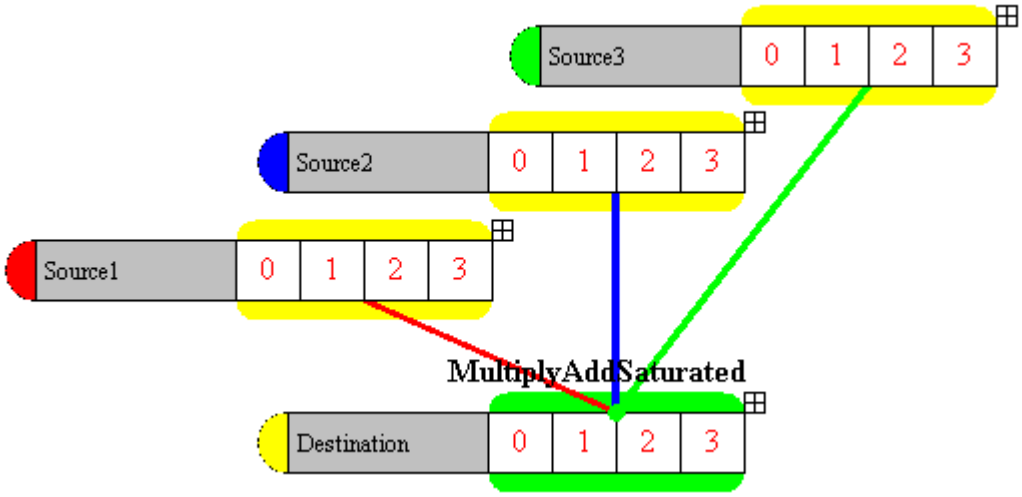


Figure 8: Example of a Multiply Add Saturated Operation, with Loads and Stores.

The order in which the words are input is important at this stage. For many of the instructions, the order of the arguments affects the result. To comply with this, my application uses the order in which the words are added as the order of the arguments in the outputted code.

3.5. Calculate

Once the user has completed drawing out the program, they simply have to click on the Optimise menu at the top, and select the processor they want the code created for. As mentioned earlier, if the universal code was chosen from the start, then any processor can be chosen at this stage. A file chooser then pops up, inviting the user to select the location and filename with which they wish to save the outputted code.

3.6. Other Options

There are a number of other options available to the programmer when creating the code. The menus at the top have options to:

3.6.1. Convert

This is used to convert a word from one data type to another. When coding there are times that it may be necessary to use an operation that acts on different size subwords than the previous operation. While normally when coding by hand, no conversion would be necessary (unless from integer to floating point, or vice versa), the control features of my application require that the user shows that they consciously want to act on the word in a different way to ensure correct code is produced, and to reduce errors. Clearly the option to change datatypes is given at the point also.

3.6.2. Delete A Word / Operation

As with any program, there are times when a user is going to make a mistake, and so the ability to delete a word or an operation is provided. Simply clicking the delete option, followed by clicking on either the word or tracing out the path of the operation, will remove the error.

4. Introduction to Application Production Overview

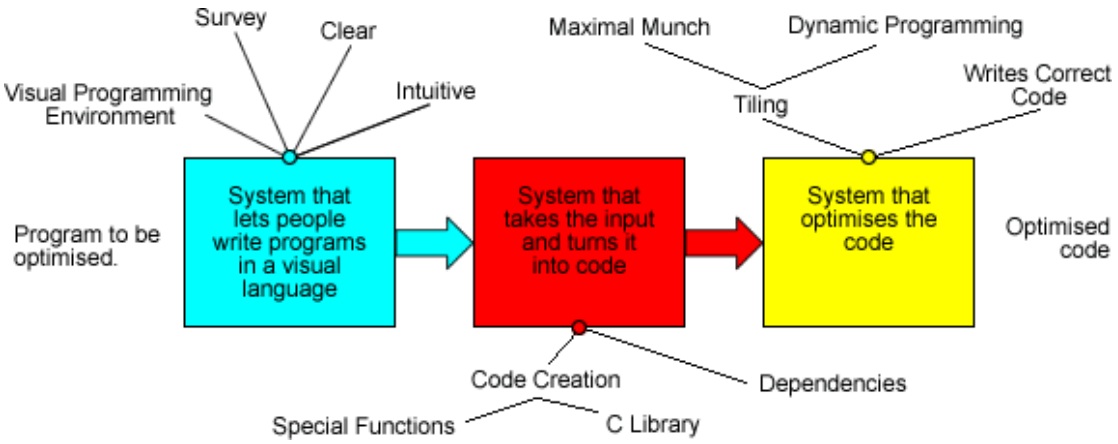


Figure 9: Application Tool-chain

4.1. Interface Design

The first part of the project was the creation of a basic user interface for the application. While this was not planned to be the final choice, it was important to still get a good design from the start, in order to allow the application to be user friendly, and easily changeable.

Once the rest of the application was complete, I revisited the design & layout in order to make it the most efficient and effective for a user. After a number of different test designs and layouts, I decided upon one which would allow the greatest possible space to complete the designs of the programs, while allowing the quickest possible access to the instruction sets. This is looked into in more detail in chapter 6, where I discuss the various decisions made and describe the results of a survey carried out in order to drive out the full benefits of the application.

4.2. Code Generation

This was the part of the project that took the most effort, as there are many different aspects to consider in order to achieve the best code possible. Allowing the user the freedom they require to create the code they want is also a challenge, and there are many aspects which overlap with the design layout choices and I will talk about them also. Other things considered include how to make the application target independent; how to deal with generating code for operations not supported by a given processor; how to cope with loads and stores which are not like other operations; and how to handle data conversion. All of these will be covered in detail in chapter 5.

4.3. Tiling and Optimisation

The last part of the main application's creation was the implementation of some basic optimisation techniques. I have used a number of different optimisation techniques, but they all fall under the broad heading of tiling. I would have liked to have included more of these in this application, but due to time limitations this was not possible.

4.4. Testing

Finally, testing had to be carried out on the application to make sure that it did fulfil its desired aims - of allowing a user to efficiently create optimised, platform independent code, while saving themselves time and effort in the process.

5. CODE CREATION

Abstract

There were various different aspects to consider when creating the code, and many different steps which have to be executed; starting with what the user inputs, and ending in the final outputted code.

5.1. Options Panel

The operations in the options panel are determined by the processor chosen. If the universal code is selected, then the operations available are a collection of operations from the different processors. If a specific processor is chosen, then only operations supported by that processor are available.

5.2. Error Checking

To ensure that the code created is correct, a number of checks are made when the user first tries to input some code. It is all very well allowing a user to draw out any program they like, but if the program they create is not supported by any available processor in the application, then the operation is pointless. Due to this, every operation drawn out by the user is validated on input, and if not supported a descriptive error message is displayed to alert the user.

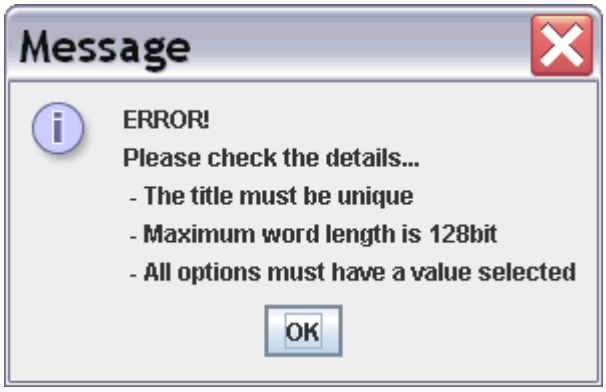


Figure 10: Example error message

5.2.1. Word Creation

The very first check performed is when a user creates a word. The word has to comply with a number of constraints. Firstly the title can not already exist in the program, otherwise a contradiction could occur in the resulting code and errors are more likely. Secondly the number of subwords multiplied by the subword size can not exceed the maximum word size, 128bits. The colour must be chosen purely to conform with the application, and finally a datatype chosen so the application can create correct code. See figure 10.

5.2.2. Data Type

Checking that the datatypes of the two words linked are supported by the chosen operation is also important. This is a fundamental check, as clearly integer and floating point instructions act in a very different way. However, many of the instructions also act differently on signed or unsigned integers, and some do not act on both.

5.2.3. Subword Size

Another check that has to be made is whether the operation supports the size of the subwords of the two words. Most operations support many different sizes of subwords,

but very few act on all sizes, and so these need to be checked. The application also only allows words of the same subword size to be used in the same operation to reduce ambiguity of the programs, and to assist in creating correct code. If a user wants to use words of different subword sizes in the same operation, then they are required to use the convert function on one of them first.

5.3. How to Cope with Loads / Stores

One aspect of the design was how to cope with loads/stores which are not operations like the others. It was decided the best way to handle these was using pseudo-operations. While the other operations clearly show the links from one word to the other, in the case of Loads and Stores, this is not suitable. So instead, a load or store is created using the menu at the top, followed by simply clicking on the words that to be loaded or stored. To show that such an action will be performed, the word is then highlighted accordingly. As described in §3.3, the outputted code is only a template, from which the user can create the correct code.

5.4. Data Conversion

Data conversion was another aspect of the design that was important. When coding, the only time that a data conversion needs to take place is between integer and floating point data types. However, I chose in my application to expand on this.

When creating the words, to assist the user to create correct programs, the user has to select the size of each subword. There are a number of reasons for this. The first and main reason is that for almost all types of operation, there is a specific instruction for each subword size. While the instruction itself operates on the word irrespective of what the word contains, the way that it acts on that word can be different for different subword sizes.

To this end, the user is required to input the size of each subword, to assist them throughout the programming process. When the user tries to create an operation, the application checks that the chosen operation supports subwords of that size. Due to this, I have added pseudo operations “convert” with which the user can convert each word to another word of a different data type and/or of different subword size. While the outputted code will not include a conversion of a word to the same data type, as the words themselves contain no such information, the application itself will act on them differently.

5.5. Create an ordered list of the Operations

Once the user is happy with the program that they have drawn out, they select Optimise from the menu at the top and select the processor that they intend to create the code for, from which a file chooser appears, allowing the user to save the code wherever they want.

5.5.1. Dependencies

In order to create this code, the first stage is to convert the list of entered operations into an ordered list. This is a critical stage, as the dependencies of every operation must be checked, in order to make sure that when the program is finally run, the operations occur in the right order, otherwise clearly the program would fail.

This is a non-trivial task, as there are a number of different aspects to allow for. To start with, the application creates a list of “safes”. These are the words which only act as sources, and are not dependent on any other part of the program. The next step is to run

though and find the destinations of each of these “safe” words, and for each destination, check each of the other source words. The first destination word that is found, for which each source word that is a “safe” is then placed in an ordered list, with its sources, and used as the first instruction in the program code. The destination word is then marked as safe, and this process continues until the entire program is in order. Care must also be taken with respect to the operations which are not “all to all” operations. When a user draws out such operations, rather than just a single link being drawn, upto 16 from each source may be present. In this case the algorithm must group all of these together.

The number of sources is dependent on the operation, and can vary from one to three, so for each destination, a different number of sources have to be checked. But there is also error checking to be done at this point, to make sure that each operation has the correct number of sources. If not, the application aborts, and an error message is shown to the user. Infinite loops must also be checked, as if such a loop occurs, clearly the resulting code would not work correctly. Again, if such a loop occurs, an error message is provided, naming the word in question.

The order in which the user adds each source is also important, as this is the order in which they want the sources to be in the final code. So this must also be kept the same in the ordered list. Once the list is complete, it is time to create the code. This is done uniquely for each processor, as selected by the user, and is discussed in Chapter 7.

5.6. Target Independence

Yet another consideration is how to make the application target independent. Initially the details for each processor were hardcoded into the application, it was later realised that a better idea would be to use machine description files. A key aspect of this application is to be user friendly. With the instructions hardcoded into the application itself, it makes it complicated to add in further instructions, and even harder to add more processors. Due to this, it was decided machine description files should be used instead.

I also tried to make the process of adding a new processor as simple as possible, however, it is still not trivial. While only a couple of lines need to be added in the application code itself, up to three new files need to be created. The first is a description file, describing the instructions supported by the processor (N.B: to be universal this file must also include the operations supported by the other processors within the application, and their description files must also be changed accordingly if the new processor brings with it new operations). The second file is used to produce the output code. Unfortunately due to the huge diversity between individual processors, a global file for this was not possible, as each processor has many specific details which need to be addressed. However, to assist in the process a template file has been created, which only needs to be customised for the processor in question. Finally as the chances that the processor supports every instruction of the other processors, a file needs to be created which handles the special case operations (see §5.7), and any operations of the other processors not directly supported, and not already in the included c library (§5.7.1) need to be added.

5.6.1. Machine Description files

Each processor has its own file, which describes all the operations supported and not supported, and any special cases that occur. The aim of these is to make the process of adding additional processors and/or instructions much more simple. My application acts as a shell, with the specifics of its operation taken directly from these files throughout the application. The description files contain key details of each instruction needed

within my application. I tried to make them as simple and intuitive as possible to both read and write.

5.6.1.1. Example Code

The example below shows the different parts of the files, each piece crucial to the application:

```
* Arithmetic Start

 - Absolute
name: Absolute: type: s: datatype: int: subString: 8:
maxLength: 128: op: vec_abs: inputs: 1: special: none
name: Absolute: type: s: datatype: int: subString: 16:
maxLength: 128: op: vec_abs: inputs: 1: special: none
name: Absolute: type: s: datatype: int: subString: 32:
maxLength: 128: op: vec_abs: inputs: 1: special: none

* Arithmetic End
```

5.6.1.2. Grouping

In the files, each operation is placed into one of seven groups; Arithmetic, Comparison, Logical, Miscellaneous, Shift, Floating Point and Convert. The start and the end of each group, is defined by "'*' + Group Name + 'Start' or 'End'", as seen in the example above. The application uses these "groups" to split up the instructions; to quickly find the correct part and type of instructions when needed; and to list the operations in the Operations menu within the application.

5.6.1.3. Operation details

For each operation that is input, comments (i.e. name and/or description) can be input using a '-' before. Next comes defining the instructions for each operation. In Altivec the vast majority of operations have the same instruction for both signed and unsigned integers, as well as different subword sizes. So if this application were created just for Altivec then the description files would be a lot simpler. However SSE2 is not so simple. As described in part §2.3 most operations have a number of different instructions depending on datatype and subword size. To get round this problem, and to make the description files consistent, it was thought the best way would be to list each datatype and subword size supported by either or both processors.

This means that the details of every instruction, for each datatype and subword size, have to be input. Note, it is important here that for each processor, even those specifications that are not supported by the processor whose file it is must still be input, as this is necessary to make the application truly universal. The best way to create a new file would be to copy an existing one, and change the instructions within. Any additional operations or datatypes / subword sizes that are input, must also be input into the other processor files.

The name of the operation must be input for each instruction. This name has to be the same for each instruction which performs the same operation, and must be grouped together within the file. Next the type, and datatype on which the instruction performs, followed by the subword size and maximum length of the word. The instruction for that operation is then listed, along with how many inputs (sources) that operation takes. Finally, any 'special' details are added at the end. These can be universal, but can also be unique to the processor, and help account for variances between instruction sets.

They are used as flags which, during code creation, are read in. Special action can be taken accordingly. If no special details are required, then ‘none’ is simply put.

5.7. Generate Code for Operations not supported

One significant problem facing a programmer trying to create code for a specific processor is that the instruction set is unique for each one. While often there is an overlap between different instruction sets, as the majority of the types of operations from one processor to the next are similar, the details, specifics and variety of each operation can vary dramatically. This problem becomes greater when a programmer tries to create the same code for two different processors. It is likely that in a program there will be one or more instructions which do not have an identical counterpart in the other instruction set. For such instructions it may be possible to compute the same operation, but using a number of instructions instead of just the one. Unless the user is very competent with the given instruction sets, it is a chore to work out how to do a task on one processor, only to have to look up the details and specifics of how to compute the same procedure with another processor. For this reason, each processor has its own code creation file with specifics for that processor. If an instruction is chosen which does not have a single instruction, the application looks for a predefined sequence to perform the operation.

5.7.1. C Library

Aside from the difficulties of finding the correct operation or series of operations to compute the same task on different processors, another problem is that sometimes there are instructions supported by one processor that simply are not supported by another, no matter what number of instructions are used. To get around this problem I have created a library. For each operation that is not supported by a processor, there is a function in the library which uses basic C code to generate the correct result. While clearly this is not ideal, it is a problem which occurs when creating the same code in a variety of different instruction sets, for which inevitably there are going to be consequences. However, the result is that the outputted code is still written in SIMD instructions wherever possible, and the code is workable on a number of processors. An advantage of using a C library is also the fact that the same instruction can be used for various processors if more than one processor does not support an instruction of another. This means that the code is only in one place, and if at some point a better way is found to perform an instruction, then the changed library can be used by all processors.

6. LAYOUT AND DESIGN CHOICES

6.1. Problem

Since this project is about the creation of a visual programming environment which will allow the user to be able to draw out a SIMD program, the design and layout of the application is possibly the most important part. A user friendly interface is essential. The application must also be quick to use. It would be pointless to have a complicated but simple-to-use application, if it takes longer to create some code than had they written it out by hand. As such, there are a large number of design and layout choices that have to be made.

To a degree, the user friendliness of an application can be down to the individual person using it. Someone who is very competent with computers could find menu systems very simple and easy to use, while to another person this could be too much work, and they may prefer, for example, simple buttons.

6.2. Basic Design

There were a number of basic design ideas that I tried to retain throughout. I wanted to keep the workspace in which the programmer would use to create their designs as large as possible. A clean look was desirable to reduce clutter, as it is important to ensure that the layout adopted for the screen is visually pleasing and effective [1], pg 23. It also had to be quick to use, and so the use of menus should be kept to a minimum. A very simple design was also required, while still allowing the user the freedom and flexibility to create the code they desire. Consistency is also crucial in good design practice in both font and layout, as well as menus and error messages[1], pg 56. Due to this, I have used the same font throughout, and error messages each have a title, with direct and descriptive information below.

6.3. Options Panel

The Options panel is key to the productivity of the application. It contains the word creation panel, and the operations panel. Part of a good interface is to guide a user's attention [8], pg 102. The use of the word creation panel and the operations panel was part of this. By far the most common interaction with my application is either creating a new word, or inserting an operation. To this end, the use of panels was chosen to speed up the process by reducing the need to search through menus to find the desired option. Its position at the top of the workspace was only natural, due to the contents of each sub-panel.

6.3.1. Word Creation Panel

Word creation is probably the second most used part of the application, but the details needed to create a new word could easily make it the longest part to use. The aim of having a panel to create a word is to make the application as easy and quick to use as possible. Had a menu been used instead, then the user would have to constantly be going through menus to create the code, as apposed to simply having all the details there in front of them, as good design also involves the minimum user input [1], pg 56. Following on from this the word creation panel is also designed to remember the details of the last word created. Once one word has been created, it is likely that at least the next two words are going to be identical apart from name and perhaps colour. To create a new word, the user only has to alter the name and click 'create'. If there is no particular preference for a name, the user does not even have to alter this, as once a word has been made, the application automatically adds a number to the end of the title in the word creation panel, so the next title will be ready and unique.

6.3.2. Operations Panel

The operations panel is subdivided into six categories. The reason for this is twofold. The first is the 'magic number seven ±2' phenomenon [8], pg 65, which states that the maximum number of 'chunks' of information a human can remember is seven plus or minus two. It is for this reason that, for example, the maximum number of grouped items, without a break, in any Microsoft Word menu is seven. The second reason is that the human brain can easily distinguish between five or so items without even thinking [10]. The operations fit naturally into six categories, and so this was a good choice.

6.4. Word Design

Perhaps the most important part of the design process, involved deciding on a look for the words, and the way in which visually the links of the operations are displayed. An important part of creating a user friendly application is to make it as natural and familiar as possible [1], pg 55. The most obvious and also most common visual representation of a word is a rectangle, divided evenly according to the number of subwords it contains, as in figure 11. But this alone does not give any detail as to the bit size of the word or its subwords. Also other information such as the name of the word would be useful. So the first step was to decide upon the design on the words.

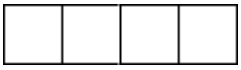


Figure 11: Standard graphical representation for a word

6.4.1. Word Design

I decided to use the basic design as in figure 11 due to its simple and intuitive design. However I also wanted to improve on it, to supply more information to a user at a single glace. To aid the user, five extra details had to be included in each word:

i. Word Name
ii. Word Size
iii. Subword Size
iv. Datatype
v. Colour

The benefits of the i to iv are clear, but perhaps not colour. I wanted to give the user the option to colour the words as they saw fit, as a tool to aid them in the design process.

One way to show the word size and subword size would be to simply have a couple of numbers beside the string. However I wanted to keep clutter to a minimum. So another option would be to include the details in the pop out title bar along with the datatype. While this would work well, I decided upon a third option, to have the size of the boxes proportional to the size of the subwords, as in figure 12. This way, in a single glance, the user can easily see the subword size. Numbering the subwords is also extremely useful. While this allows a user to be able to easily see how many subwords there are, when it comes to drawing out operations which act on different subwords rather than the word as a whole (such as a permute operation or because the user does not want to apply the operation to the word in its current state), then the numbered subwords reduce the chances of incorrectly clicking on the wrong subword. These numbers are also coloured according to the datatype; red = signed integer, blue = unsigned integer, black = floating point.

Figure 12: Chosen word representation.

Including the word name was also important. Due to the choice of the link design (see §6.4.2) the name could not be either above or below the word. I also wanted to include colour in the word. So after a number of designs, I decided upon that as shown in figure 13. The title is held in a box to the left of the word. There is also a semi-circle coloured as chosen by the user on creation. It is not always necessary to have the title displayed, and so to reduce clutter, the title can be retracted, as in the example on the right, simply by clicking anywhere on the title bar or the semi-circle. There is also a chequered box in the top right corner which is used to move the word around the screen. I chose to use a box rather that simply clicking and holding somewhere on the word, as this is more intuitive to the user, giving them a set place to click.

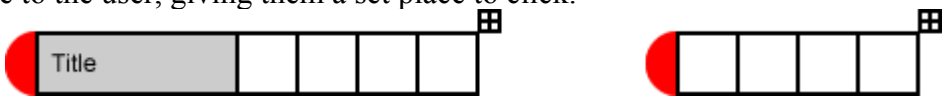


Figure 13: Title bar choice

6.4.2. Link Design

It was difficult to decide on the final way to represent the links. Designs that looked good on paper, suddenly did not work so well in practice; and designs that did seem to work well initially, had problems of their own, as I shall now explain.

Initially I drew out a number of designs taking suggestions from people I talked to, to get a broader collection and ideas other than my own biased opinion. I finally decided upon five designs;

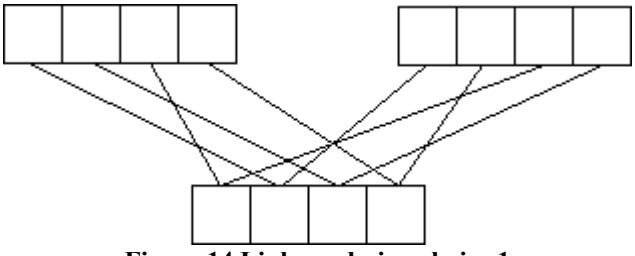


Figure 14 Linkage design choice 1

This first design, figure 14, is the most basic. Simple lines from the source subwords to the destination subwords.

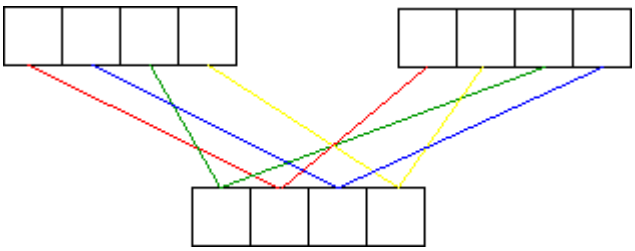


Figure 15 Linkage design choice 2

The second design, figure 15, is a variation of the first, but this time, each line is a different colour in order to try to make each individual link stand out clearly.

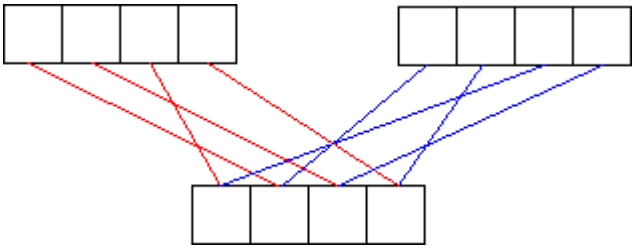


Figure 16 Linkage design choice 3

The third, figure 16, is again a variation of the same theme. This time however all the links from any given word are the same colour. A nice option here is the ability to make the colour the same as that chosen for the word by the user.

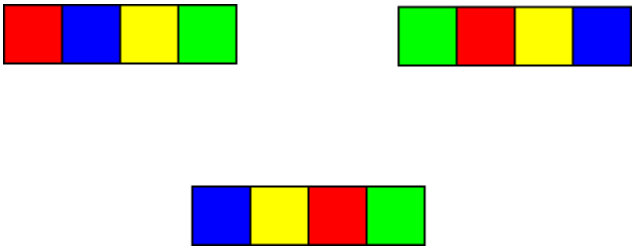


Figure 17 Linkage design choice 4

Options four, figure 17, and five, figure 18, are quite different from the other designs. Neither has any physical links between the words, but instead rely on a coding technique. This design shows the links by colour matching.

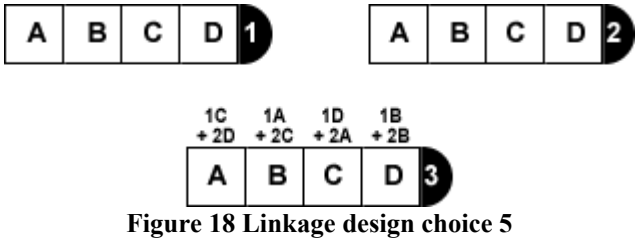


Figure 18 Linkage design choice 5

Similarly, option five relies on code to represent the links. Each word is given a unique number by the application (as seen on the right of each word), and each subword a letter. The contents of the destination subwords are then shown directly above each one as a basic formula, e.g. in the case of subword A “1C + 2D” represents word 1, subword C and word 2, subword D being acted on by the given operation.

Clearly each design has its pros and cons. The physical links in the first three designs I thought would aid the user by being able to follow the links, and easily see how the word is permuted. The varying use of colour being used attempts to make it easier to

distinguish between the links. Design four is very bold and the links stand out so much a user can easily see which two are linked. Option five is perhaps the neatest, and by looking at the destination it appears easy to identify the sources.

I had my own ideas about which I should use. I thought that option five, being the neatest, would be the best choice, with the use of code making the chance of misreading it very low. Option four looked good, but what happens when there are sixteen subwords, all different colours? How long would it take to find the different links? Out of the first three designs, I thought design two or three would be best but again I couldn't decide which. To become more informed, I decided to perform a survey. I timed how long it took people to work out the links in each design, on varying numbers of subwords; and asked for their opinions. The results were quite interesting.

6.4.3. Link Design Survey

I surveyed ten people, timing how long it took them to follow a link from one source subword to its destination and then to the second source subword. I thought this the fairest test as it tests in both directions, both from and to the sources. Test sizes of 4, 8 and 16 subwords were used to get a spread of results. The results of the tests are in figure 19.

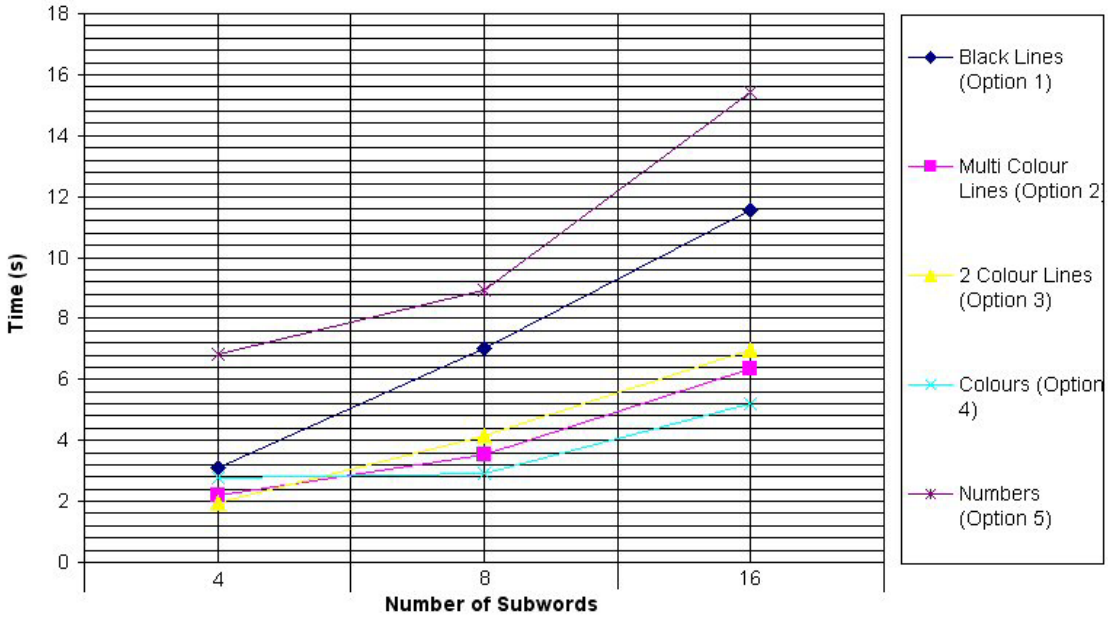


Figure 19: Graph Showing the Results of the Survey on Link Type

The most unexpected outcome of the survey was just how slow the timings in option five (figure18) were compared to the others. Before I ran the tests, I thought that this would have been quicker than designs one, two or three, due to the absence of confusion created by all the different criss-crossing lines. Clearly, adding the colour to the lines in options two and three has made them easier to read, and so improved the timings. The clear winner is the use of colours, option 4, figure 17.

During the survey, I also asked what they thought of each design. Initially most people liked the idea of option five, as it was the cleanest, and they all thought the clearest to read. However, in practice they decided that it was too much work, searching through to find the particular details, and not very intuitive at all as to which subwords were linked. Option one was the least favourite, as although it does not take quite as long as option five, people found it a lot more confusing. Options two and three were much clearer. However, while some people favoured the use of a different colour for each link, as in option three, other people found the use of so many colours too manic. While they still

managed to read off the details faster than from option two, they preferred option two's neatness and consistency, also being the most pleasing to the eye behind option five. Option four was the clear favourite. Using a smaller number of subwords, people responded that they did not even have to think about which were linked, the eye simply picked up the links. As can be seen in figure 19 the time difference between 4 and 8 subwords is just 0.11s, however there is a sharp increase from 8 to 16.

6.4.4. Final Choice

Following the survey, I finally decided upon option three. Clearly from the results, the choice was between options two, three and four. I decided against option four, as although it was the quickest to use, it turns out to be impractical. Firstly, the ability to use a single subword as a source for more than one subword is not there. For the colour system to work each subword can only be used once. The lack of any physical link between the words is also a negative, as it is not intuitive which words are linked. Also due to the nature of the way that a link is created, as the colours are dependent on each operation, there would have to be another copy of the word every time it was a source. Clearly becoming cluttered and confusing very quickly. So the choice was then down to option two or there, whether to have many colours, or just the two. I decided upon the use of just two colours as while multiple colours was quicker, this was only a marginal difference, and a number of people expressed a distaste at having so many colours.

6.5. Menus

The use of menus is also important. They are there to hold the options available to the user, which are not used enough to warrant a place on the workspace. Care has to be placed in grouping items together to form different menus, and in naming them [8], pg 265. Grouping with too many items in each menu can cause confusion, and take a user longer to find what they want. Likewise, too few groupings can create an unnecessary search of the different menus to find the desired operation. A balance needs to be found, with intuitive and descriptive names of the menus summarising what is contained in them. In my application, the menu items fitted into a number of distinct groups with no confusion as to what was in each;

- File (New, Save, Load and Exit)
- Processor (Processor selection)
- Special (Load, Store and Convert)
- Clear (Clear Link or Delete String)
- Optimise (Create the Code)

Notice here that the load and stores have been relegated to menu items, as they are not used frequently enough to warrant a position on the workspace.

7. TILING AND OPTIMISATION

7.1. What is Tiling?

Tiling, at its most basic, is taking a tree-like representation of the given program, where each node only contains a single operation: addition or subtraction; memory fetch or store; conditional jump etc. Overlaid on top of this is a set of legal instructions for a given processor. The aim of this is to cover the entire tree in non-overlapping 'tiles'. The optimal tiling of a tree is the instruction sequence of least cost (the shortest sequence of instructions or the sequence that takes the lowest total time).

We look now at an example extracted from [3] pg 178, given the following set of basic instructions:

ADDI	$r_1 \leftarrow r_0 + a$
ADD	$r_1 \leftarrow FP + r_1$
LOAD	$r_1 \leftarrow M[r_1 + 0]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow ri \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow r_0 + x$
ADD	$r_2 \leftarrow FP + r_2$
LOAD	$r_2 \leftarrow M[r_2 + 0]$
STORE	$M[r_1 + 0] \leftarrow r_2$

An optimised version of this is:

LOAD	$r_1 \leftarrow M[FP + a]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow ri \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow FP + x$
MOVEM	$M[r_1] \leftarrow M[r_2]$

The below diagram shows how the tiles have been placed in both cases, for the same program:

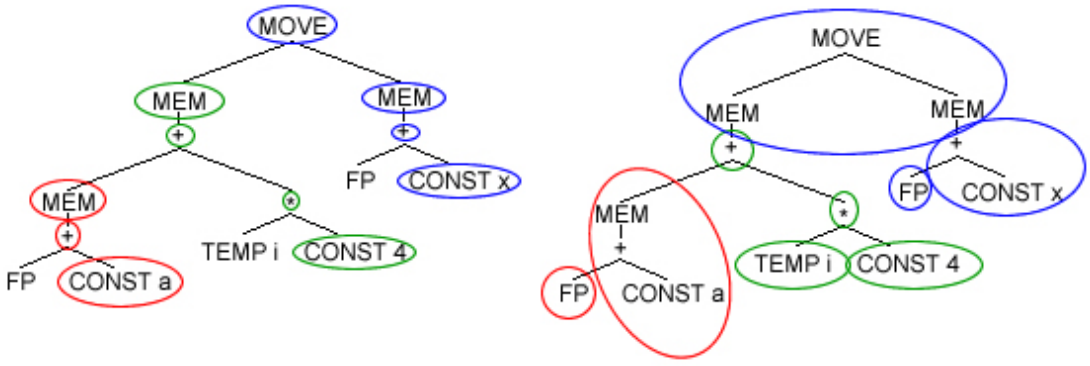


Figure 20 Tiling example

There are a number of different techniques that can be employed to lay the tiles, but the two main ones are 'Maximal Munch' and 'Dynamic Programming', [3] pgs 180 – 183. Maximal munch works by starting at the root of the tree, and finding the largest tile that fits it, and any leaf nodes. This tile is then placed, leaving several subtrees leading from it. The same procedure is then carried out for each subtree, and so on, until the whole tree is covered in tiles. This method is very simple, and will always result in an optimal tiling. However, the result is not always the optimum. To find the optimum tiling, Dynamic Programming can be used. However, this is much more complicated, and

takes longer to run. It works by assigning a cost to each node of the tree. Then, working bottom up, it calculates what tiles can go on the leaf node, and assigns the cost to that tile. Then, moving up the tree, it calculates the cost of adding each tile, knowing the cost of each of the leaves (as it is working up, these have already been calculated) the tile with the lowest cost is then chosen, and the process continues.

In a small example of Dynamic Programming (figures taken from [3] pg 182), suppose we have the following tree:

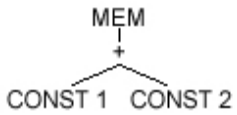


Figure 21: Example Tree to be Tiled

The only tiles that cover the two CONST's are ADDI. So the cost of each of the leaf nodes from the + are 1. There are a number of instructions that match the + node, so we get the following:

Figure 22: First Round of Tiling

So clearly, either of the two ADDI tiles are optimal, and so the + node gets the cost 2. Next, look at the MEM node. Again there are a number of tiles that correspond to this node, as shown:

Figure 23: Second Round of Tiling

So even though we calculated the cost of the + node being 2, we can place the LOAD instruction over the whole thing, apart from one of the CONSTS, and since we have already calculated the cost of the missed out CONST (we know its cost is 1), we find that the total cost of the program is 2. Note: If we had used maximal munch it would have returned the same result, however this is not always the case.

7.2. How does the application make use of such Tiling?

Once the code has been input, and the ordered list and dependencies checked, the next stage is the Tiling. My application tiles the code, using the maximal munch technique. Since the list being tiled is already a list of higher level operations, it makes little sense

to tile using the dynamic programming technique, as there are very few operations that can actually replace a number of smaller operations.

Firstly taking the ordered list, starting from the top and working down, it performs the following steps for each link:

1. Was the operation drawn as an "all to all" operation? If it is, continue to step 2, if not something has to be done first (§7.3).
2. Can any optimisation take place (§7.6)?
3. Refer back to the description file and find the instruction corresponding to such an operation, datatype and word / subword size.
4. Is the instruction supported by the processor (§7.4)? If not a special case op has to be performed, and steps 4 and 5 will be skipped (§5.7).
5. Does the instruction have any special details (§5.6.1.3)?
6. How many sources are needed? Use any subsequent links needed to populate the sources making use of any special details found in step 4.
7. Correct code is output.

7.3. What about operations drawn which are not "all to all"?

If the operation was not drawn as an "all to all" operation, then before the instruction to perform the operation can be applied, the order of the subwords of the source words has to be altered. Collecting the details of the permutation to be made, from the list of links, the application first applies a permutation operation using the given instruction set, as it would, had the user drawn out a permutation operation. This is then inserted into the outputted code ahead of the operation the user had intended.

If the Altivec instruction set has been chosen, then this is a relatively simple task, as the permute operation is a single instruction, with the permutation variable being calculated easily. However, in the case of the SSE2 a permutation is much harder to work out, as one major drawback of using the SSE2 instruction set, is the lack of a permute operation. This proves to be a big headache to programmers, and a difficult problem to resolve in my application. There are a number of ways to calculate a permute. Sometimes there are quick routes to perform the desired operation that a programmer can see. However this is not always the case. My application has to calculate a correct permutation operation every time, and as we are looking to create efficient output code, this operation should use the minimum number of instructions. The problem is there is a huge variety of ways to create a permute operation. Looking into the fastest way to create any permute using any instructions could provide enough material for a project on its own. The route I chose to go down involves using shifts, shuffles, ands, and masks. While this may not always result in the fastest possible permutation, it will be a very quick one, and one which can work every time. There is still room for optimising the sequence of instructions here too; and the result is that my application produces the minimum number of instructions needed, using only these 4 operations.

7.4. What to do if the instruction is not supported?

The application is aimed at being target processor independent; however there are many operations which are not universal across each processor. If such an operation is drawn, which is not directly supported by the target processor, one of two things must happen. The ideal solution is a short sequence of instructions which can perform the desired operation while still using SIMD instructions. However, this is not always possible, and so the only alternative is to use standard C code (§5.7) to perform the desired instruction. For each operation that is not directly supported, the application performs

either of the above. In the first case the correct sequence of SIMD instructions is hardcoded into the file '"processorname"SpecialOps.java' and in the second case, the alternative C code is in the corresponding C library.

7.5. Create code

If the instruction is supported by the processor, then first the application checks how many sources, n, are needed for such an instruction. Since the lists of links are ordered, these will be the following n-1 links. Knowing the intrinsic, the number of sources, the name of each source, the name of the destination (the destination of any of the above links) and making use of any special details, the correct code is then output.

7.6. Optimisation

Optimisation actually takes place just before the code creation. The application runs through the ordered list of links, checking for any patterns matching the tiles below. If such a match occurs, the list of links is replaced by the corresponding shorter list as described below. Note that we must be careful here, as although it may be possible to simply match a tile, the reason why the tile was not used by the user could be because one or more of the intermediate steps replaced by the tile are actually used elsewhere in the code . In which case, placing the tile would break the program.

7.6.1. Basic Operation Tiling

Due to the level of the code which we are using, there are not too many opportunities to apply the basic tiling technique in the spirit of the fundamental idea as described in §7.1. In fact the only one I found was the multiply add accumulate, an Altivec operation, as described below.

7.6.1.1. Multiply Add

Multiply Add (vec_madd(a,b,c)) is a floating point operation supported by the Altivec instruction set, and is identical to the sequence (a * b) + c. The program looks for a matching pattern, and if found, the links are replaced by corresponding links representing a single multiply add operation. This results in reducing the number of operations by one.

7.6.2. Logical Operation Optimisation

The second type of tiling that the application exploits is logic optimisation. The application looks for a number of patterns of logical operations, which can be replaced by much more simple ones.

(A ^ B) v A = A	Reduces # instructions by 2
(A v B) ^ A = A	Reduces # instructions by 2
(A ^ B) v (A ^ C) = A ^ (B v C)	Reduces # instructions by 1
(A v B) ^ (A v C) = A v (B ^ C)	Reduces # instructions by 1

7.6.3. Permutation Concatenation

The third type of tiling the application is looking to perform is the concatenation of permutations. While this may be a fairly obvious thing not to do when drawing out the code, nevertheless it could happen, and it is yet another example of where tiling can be used to optimise the code. One permutation followed immediately by another, can clearly be combined into a single permutation.

8. TESTING

Abstract

There are a number of different ways in which the application needs to be tested. Assuming the layout has already been completed as in chapter 6 this still leaves:

8.1. Correctness

The first thing to check when creating an application such as this, is whether it behaves correctly for each use, with all the options performing the desired action; and secondly, whether or not the output is correct for the given input, i.e. does the code match the drawn out program? The answer in both cases turns out to be yes. While many tests have been performed, the proof of the first part cannot be shown on paper. However, for the latter, Appendix A has an example, and the corresponding output code. Careful scrutiny shows that the output is correct. Chapter 9 expands on this part of the testing, so it was felt further examples here are not needed.

8.2. Does the output code really perform the required task?

The second step is to test whether or not the outputted code really performs the required task. However, the outputted code, as seen above, matches what the user has put in, and since we know that the operations are correct for the given processors, we know that the code works. However, some of the code it not directly created from the user's input. There are special functions, using either the C library or a set of special instructions, to perform those operations not directly supported by the processor. In the case of SSE2 the permutations are not straightforward, and are created dynamically by the application, so these too must be tested.

8.2.1. C Library

Appendix D shows 3 examples of varying special operations. In each case the correct output was obtained. The number of examples was limited in the interest of saving trees, but the remainder of the library also works correctly.

8.2.2. Special Functions

Each of these does perform the desired operation. Appendix E has a couple of examples, but as in the C Library above, the number of examples has been limited.

8.2.3. Permutations in SSE2

A major piece of the program is its ability to handle permutations for a user automatically. In Altivec, this only requires a single permute operation, and so the main advantage is its ability to reduce user error by calculating the values for the permute operation. However in SSE2, this is a much bigger problem, and the application takes the input and calculates, not only the correct set of instructions to perform the permute, but also an optimal set. This is a difficult task to perform, and so tests must be made to make sure that the generated code is correct, and does in fact perform the desired permutation. Appendix B lists an example permute operation for each of the 4 subword sizes (64bit, 32bit, 16bit & 8bit) and the corresponding generated code. Using C code, each of these, amongst others, were tested and the correct output was found in each case. A test file for the 8 bit permute, as well as the results are included in Appendix C**,** as part of §8.4.2**,** testing the speed of the outputted code.

8.3. Universal code

Another aim of the application was to create code that would be universally applicable. That is, to draw out the code irrespective of the intended processor, and then having the ability to create code for any processor supported by the application. From knowledge of the instructions, and through the work and research performed, to the best of our knowledge the code produced does perform the same operations. Unfortunately, despite our best efforts, we were unable to use a computer supporting Altivec operations, so we were unable to test thoroughly this aspect of the application.

8.4. Speed of Outputted Code

The final speed of the outputted code was also a consideration. A number of tiling techniques were used to try to optimise the set of operations before the code was generated. On the whole, the output code is identical to that which would be done by hand, so there is no gain from testing the general speed difference between code generated by a user and that done by hand. However we can test the different tiling techniques.

8.4.1. Logical Optimisation

Recall from §7.6.2 that a number of logical optimisations are performed shortly before the code is created. Appendix B shows an example program drawn out by a user, who has not made use of logical optimisation. The first example shows F = ((A ^ B) v A) + E. However, this is the same as F = A + E, as (A ^ B) v A = A. The code below shows that in each case, the optimisation has been made. However, obviously, if the intermediate step is used as a source elsewhere, i.e. B, then it must still be calculated. Example 2 shows that this is the case, when B is used as a source for the extra subtract. Note also, that the previous optimisation is still in effect, and A is still carried through as before.

8.4.2. Speed of permutations in SSE2

We know that SSE2 permutations are slower, in general, than those of Altivec code, as more than just a single instruction is often necessary on subwords of 8 or 16bit. One of the main aims for the application was to enable the user to produce correct and efficient SIMD code, and while the application does achieve this - is it possible to improve on the speed of the code using other means? The problem with a permutation using SSE2 is that when dealing with 8 or even 16bit subwords the number of instructions needed can be quite large. While my application does not necessarily find the optimum sequence of instructions (see §7.3), it does find the optimum using shifts and shuffles. But could a completely different approach be better? I.e. storing the word in a single C array, and calling out the subwords in the correct order. The downside of this approach is the extra overhead of storing and fetching from memory, as opposed to holding all the data in the registers. However, with the number of instructions that are necessary to perform an 8 or 16bit permute, could this be quicker?

To test this I created several permute operations using my application for each subword size, and proceeded to test the time taken by these versus the same permute using basic C code. Appendix F shows an example of the code used to test the performance of the 8bit permutations. The table below shows the results of the tests.

Sub Word Size	Average Time with SSE2	Average Time using basic C	Average No Cycles with SSE2	Average No Cycles with C
8bits	8.736×10^{-8}s	8.362×10^{-8}s	226.59	216.88
16bits	2.139×10^{-8}s	3.907×10^{-8}s	55.48	101.34
32bits	0.462×10^{-8}s	2.055×10^{-8}s	11.99	53.29

64bits	0.464×10^{-8}s	1.079×10^{-8}s	12.04	27.99

Figure 24: Table Showing Average Times for Permutations in both SSE2 Code, and basic C Code

From this, it is clear that the SSE2 permutation vastly outperforms that of basic C code in all but 8bit subwords. In the case of the 8bit subwords, the C code is only marginally quicker. So while the outputted code is slightly slower, it is more than sufficient for all but the most high performance demands. In such a case, it may also be worth investing some time looking into a better set of SSE instructions to perform the permutation, as there may be a quicker one than what my application produces. Note, the times for the SSE2 permutations on 32 and 64bit subwords are almost identical, as SSE2 has no 64bit shuffle support, and so in both cases a 32bit shuffle is used, and in both cases only the single operation is necessary.

8.5. Efficiency.

Another important aim of the application, as well as creating correct, efficient code, was to increase user productivity, to reduce the time it takes to write out a program. To this end, I created a number of small programs, using pseudo code, and asked 10 people to create them using SIMD instructions for both SSE2 and Altivec. As my application is aimed at allowing a user to create segments of code, as apposed to an entire program, each test was only 10 operations long. Before being asked to create the code, I talked each participant through a sample piece of code using each of the instruction sets. As reference manuals to find the instructions, they were each given Microsoft's online SSE2 instructions manual [16] and a pdf copy of the AltiVec Technology, Programming Interface Manual [17], and 5 minutes to look over them, to familiarise themselves with the format. They were then given the pseudo code and asked to produce the same code using both instruction sets. Both loads and stores were not requested, as my application only produces a template for these, and so would still need to be completed by hand even if using my application (see §3.3). Also, only operations acting on the whole word were chosen, as apposed to any sort of permutation, as this will be covered in another test. Once they were finished, they were then asked to complete the same task, only this time using my application. Again, they were given 5 minutes to prepare. The below table shows the times for each participant:

	SSE2	Altivec	My Application
Person 1	8mins 33s	5mins 03s	3mins 28s
Person 2	7mins 58s	6mins 05s	4mins 03s
Person 3	8mins 30s	4mins 36s	3mins 34s
Person 4	9mins 47s	6mins 19s	3mins 53s
Person 5	7mins 36s	4mins 31s	3mins 21s
Person 6	8mins 17s	5mins 07s	3mins 49s
Person 7	7mins 19s	5mins 04s	4mins 33s
Person 8	8mins 22s	4mins 38s	4mins 28s
Person 9	8mins 36s	4mins 40s	3mins 21s
Person 10	7mins 48s	4mins 53s	3mins 32s

Average Time:	8mins 17s	5mins 6s	3mins 48s

Figure 25: Table Showing Times Taken to Create Small Programs

As can be seen, the average time for both instruction sets is significantly greater than that of my application. But there is also a big difference between the instruction sets themselves. This is because Altivec instructions are much simpler to understand and code than those of SSE2. Whereas in Altivec, each operation just has a single instruction for all supported subword sizes, in SSE2 there is a different instruction, in general, for each subword size. This adds extra time to the look up process, as for each operation not only do you have to find the right instruction, but the one which acts on the right subword size. Another possible factor is the layout of the manuals, using the Altivec manual takes on average a shorter time to find the desired operation. It also has to be noted that in the tests, two people spelt an instruction wrong. While this is not disastrous, and would hopefully be spotted quickly after running the code, it is another example of the advantage my application gives. Another point that has to be made from the above table is that not only is the time taken to create the code quicker using my application but also, this code is universal, i.e. in only the time it takes to click a button the code is created for either instruction sets. The time taken to create the code by hand for both instruction sets is the sum of the 2 columns; whereas using my application it is simply the time stated.

Next, I gave ½ the people an example program using SSE2 instructions, and the other ½ Altivec, and asked them to convert it to the other instruction set. The results are in the table below:

	By Hand		Using My Application	
	Altivec to SSE2	SSE2 to Altivec	Altivec to SSE2	SSE2 to Altivec
Person 1	8mins 18s	4mins 29s	3mins 52s	3mins 55s
Person 2	7mins 32s	4mins 40s	3mins 38s	3mins 47s
Person 3	7mins 15s	5mins 43s	4mins 12s	4mins 05s
Person 4	8mins 23s	5mins 05s	3mins 38s	4mins 17s
Person 5	8mins 6s	4mins 34s	3mins 56s	3mins 52s
Average Time:	7mins 55s	4mins 54s	3mins 54s	3mins 59s

Figure 26: Table Showing Times Taken to Convert Small Programs

While these results do not show anything particularly new, it was used to demonstrate that as well as creating new code, my application can be used to efficiently convert code from one instruction set to another.

8.5.1. Permutations

The decision was made to not ask the volunteers to create permutation code, as this is more demanding without knowing the instructions thoroughly, and without any previous attempts. However, as a guide, I created a permutation for each subword size using SSE2 instructions, with the times shown below. The results show just how long and difficult it is to perform permutation operations by hand. I would consider myself to have quite a bit of practice and experience at performing such operations, and yet it still took the best part of half an hour to perform a permutation of 8bit subwords. When this is compared to just 1minute 30 seconds using the application, it really shows how much time and effort it saves. For the other subword sizes too, the difference in the times is remarkable.

Subword Size	8	16	32	64
Time by hand	≈ 29mins	6mins 34sec	1min 48sec	1min 1sec
Time using application	1min 30sec	38sec	23sec	18sec

Figure 27: Table Showing Time Taken to Create Permutation Operations by Hand

9. A LARGER EXAMPLE

9.1. Introduction

Finally, following on from the testing, I thought a real world example would be good to further show the usefulness of the application. AES (Advanced Encryption Standard) is a block cipher, created by John Daemen and Vincent Rijmen. First introduced in 2001 after winning a world wide competition ran by the National Institute for Standards and Technology, it was adopted by the US as an encryption standard, used by U.S. Government organizations to protect sensitive material. However it is also used throughout the world, due to its security, speed in both hardware and software, little memory use and relatively simple implementation.

Using the cipher creators own book, [9] pgs 31-62, for specific details, we continue. AES has a fixed block length of 128bits, supporting key lengths of 128, 192 or 256 bits. The input consists of a plain text block, a 4x4 array of bytes named the 'state', and a key, with the output being a 4x4 array of bytes, the cipher text block. The encryption consists of an initial key addition, followed by N-1 rounds (where N depends on the key length) of the following:

- SubBytes; Each byte is replaced by another, according to a look-up table.
- ShiftRows; A byte transposition, where each row in the array is cyclically shifted by a set amount, resulting in each column containing an element of each of the original columns.
- MixColumns; A mixing operation on each of columns of the state.
- AddRoundKey; The state is combined with the round key, derived from the cipher key.

A final round of the above is then completed, omitting the mixColumns operation.

9.2. Implementation

Firstly, note that the block length is fixed at 128bits, subdivided into a 4x4 array of bytes. This can be visualised as in figure 28 a), but this fits perfectly into a 128bit word, with 8bit subwords, as in figure 28 b). This then prepares it to be operated on by 128bit SIMD instructions.

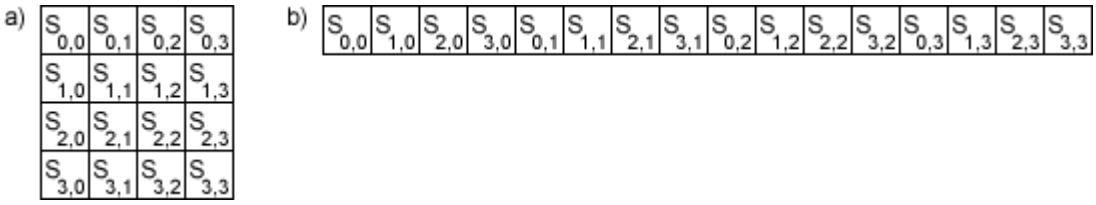


Figure 28: The 'State' in a) Standard Matrix Form, b) Modified to fit in a Single Word

9.2.1. SubBytes

Unfortunately, this stage of the sequence does not lend itself well for my application. It is the only non-linear transformation of the cipher, as each byte is replaced by some value from a look-up table. However, this is not just a problem with my application, but more a problem with the nature of the instruction set. When completing a load and store using SIMD instructions, they assume that the whole word is to be loaded in one go; whereas in this case, we really want to use each subword as addresses. It is an operation that is not mapped well onto the instruction set, and therefore the application does not cope well with it by virtue of the fact that it is a difficult problem to realise.

9.2.2. *ShiftRows*

As described above, the shift rows operation takes the state array, as in figure 29 a) and shifts each row cyclically. The top row stays as it is, the second row shifts to the left by one, the third to the left by two, and the fourth row by three. Since we are storing the array as in a single word, we apply the function to the whole array in one go. This shift operation can be realised as a permutation of the state, and in my application, results in figure b). This took just over 1 minute to complete using my application.

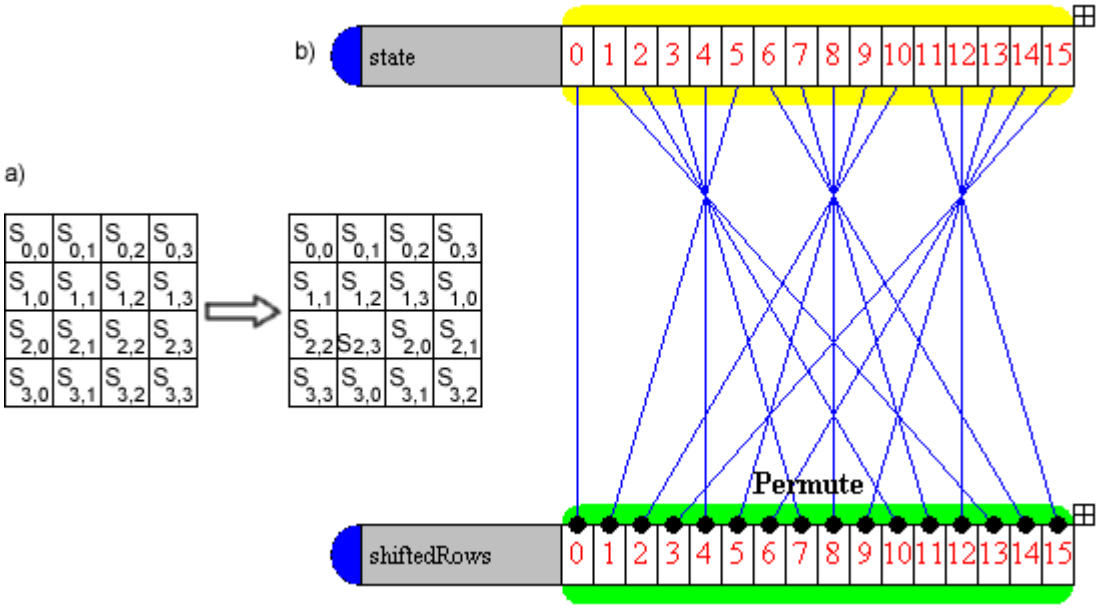


Figure 29: Shift rows diagram and corresponding screen shot

9.2.3. *MixColumns*

The next stage is to mix the columns. Whereas the previous operation acted by cyclically shifting the rows, this stage acts on the columns, multiplying them by a constant matrix, as in figure 30.

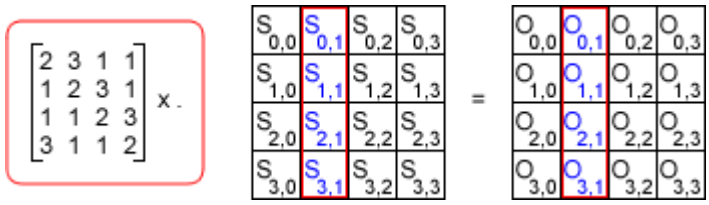


Figure 30: Mix Columns Operation

It is not intuitively clear how this can be done easily with the matrix described as a single word as in figure 28 b), however, following work in [10] pgs 148-149, there is a fast sequence of parallel instructions for the PLX instruction set which computes the column transformations.

The first thing to note is that there is a standard algorithm for calculating multiplication by 02, called xtime(x) [9] pgs 53-54. Again, a parallel version of this is shown in [10];

```
1    P0 and         R06, R01, R13  // mask elements with 0x80
2    P0 pcmp.1.eq   R06, R06, R13  // compare elements to 0x80
3    P0 and         R06, R06, R14  // mask elements with 0x1B
4    P0 pshift.1.l  R05, R01, 0x01 // apply 1bit shirt to each element
5    P0 xor         R05, R05, R06  // compute result
                                                          (9.1)
```

Where we are assuming that the state matrix is in R01, and R13 and R14 are both 128bit words with each subword filled with the bytes 0x80 and 0x1B respectively. It is clear that his performs the desired operation, if we note that a simple multiplication by 2, when dealing with bits, is to simply shift everything to the left by one (the fourth line). The three lines before this check for overflow.

Using this, any multiplication of the state by 01, 02 or 03 can easily be calculated. Multiplication by 01 is simply the state itself, by 02 is using the instructions above, and by 03 is simply the state ⊕ (state x 02). With this in mind, we can move onto computing the mix column. [10] pg 149 gives the instructions for computing the mix column as;

```
1   P0 permute     R03, R01, R10  // construct the a matrix
2   P0 permute     R04, R01, R11  // construct the b matrix
3   P0 and         R06, R01, R13  // construct the c matrix
4   P0 pcmp.1.eq   R06, R06, R13  // construct the c matrix
5   P0 and         R06, R06, R14  // construct the c matrix
6   P0 pshift.1.1  R05, R01, 0x01 // construct the c matrix
7   P0 xor         R05, R05, R06  // construct the c matrix
8   P0 xor         R06, R01, R05  // construct the d matrix
9   P0 permute     R06, R06, R09  // construct the d matrix
10  P0 xor         R01, R03, R04  // add the matrices
11  P0 xor         R01, R01, R05  // add the matrices
12  P0 xor         R01, R01, R06  // add the matrices
```

(9.2)

It is not intuitively clear just how this set of operations performs the desired task, so I'll explain it in more detail now. Looking back at figure 30, let us look at just one of these column multiplications, in particular the first. Such a multiplication is performed as in figure 31.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S_{0,0} \\ S_{1,0} \\ S_{2,0} \\ S_{3,0} \end{bmatrix} = \begin{bmatrix} 2S_{0,0} + 3S_{1,0} + 1S_{2,0} + 1S_{3,0} \\ 1S_{0,0} + 2S_{1,0} + 3S_{2,0} + 1S_{3,0} \\ 1S_{0,0} + 1S_{1,0} + 2S_{2,0} + 3S_{3,0} \\ 3S_{0,0} + 1S_{1,0} + 1S_{2,0} + 2S_{3,0} \end{bmatrix}$$

Figure 31: Column multiplication broken down

Next note, that each of these values $S_{0,0}$ etc are in fact bytes. As such each value held in binary form, and in binary addition is performed using the exclusive or operation, xOr. This means that we can re-write each line of the result matrix in figure 31 as, for example, $2S_{0,0} \oplus 3S_{1,0} \oplus 1S_{2,0} \oplus 1S_{3,0}$. Rotating the matrix now, in its new representation, so that the rows become the columns results the first part of figure 32.

$$\begin{vmatrix} 2S_{0,0} \\ \oplus \\ 3S_{1,0} \\ \oplus \\ 1S_{2,0} \\ \oplus \\ 1S_{3,0} \end{vmatrix} \begin{vmatrix} 1S_{0,0} \\ \oplus \\ 2S_{1,0} \\ \oplus \\ 3S_{2,0} \\ \oplus \\ 1S_{3,0} \end{vmatrix} \begin{vmatrix} 1S_{0,0} \\ \oplus \\ 1S_{1,0} \\ \oplus \\ 2S_{2,0} \\ \oplus \\ 3S_{3,0} \end{vmatrix} \begin{vmatrix} 3S_{0,0} \\ \oplus \\ 1S_{1,0} \\ \oplus \\ 1S_{2,0} \\ \oplus \\ 2S_{3,0} \end{vmatrix} = \begin{vmatrix} 1S_{2,0} \\ \oplus \\ 1S_{3,0} \\ \oplus \\ 2S_{0,0} \\ \oplus \\ 3S_{1,0} \end{vmatrix} \begin{vmatrix} 1S_{3,0} \\ \oplus \\ 1S_{0,0} \\ \oplus \\ 2S_{1,0} \\ \oplus \\ 3S_{2,0} \end{vmatrix} \begin{vmatrix} 1S_{0,0} \\ \oplus \\ 1S_{1,0} \\ \oplus \\ 2S_{2,0} \\ \oplus \\ 3S_{3,0} \end{vmatrix} \begin{vmatrix} 1S_{1,0} \\ \oplus \\ 1S_{2,0} \\ \oplus \\ 2S_{3,0} \\ \oplus \\ 3S_{0,0} \end{vmatrix}$$

Figure 32: Alternative representation of figure 31

Since addition is commutative we can reorder the columns in the first part of figure 32 as shown. Now note that each row of this result contains each element of the original column matching up nicely with our representation of the original matrix in a single

word, figure 28. However the elements of each row are out of order with the exception of the third.

The first two lines of the code permute the first to rows into the correct positions and noting that they are being multiplied by the constant 1, nothing else has to be done to these. We know that the elements of row three are already in the correct positions; however they need to be multiplied by 2. Looking back at (9.1) above, we already known how to perform this using the xtime(x) function which results in lines three through seven. Row four is multiplied by 3. We know from above that in binary this results in the state ⊕ (state x 02). Since the previous operation has already multiplied the row by two and as is was not permuted we simply have to perform the xOr operation of this and the state to get the result. This then needs to be permuted into the correct order, line nine. Finally, as described in figure 28 each of these lines then need to be xOr'd together to get the final result, lines ten through twelve.

This operation as drawn out in my application can be seen in appendix G. This took between six and seven minutes to draw out in my application, but by hand this would have taken a lot longer. From §8.5.1 an approximate guess would be about an hour and a half! Looking at the length of the output code, especially for the SSE2 instructions the amount of effort my application has saved it quite considerable.

9.2.4. AddRoundKey

The final stage is to add the key to the state, performed by combining the state with a round key with the bitwise operation XOR. This is the simplest part of the encryption, as shown by figure 33.

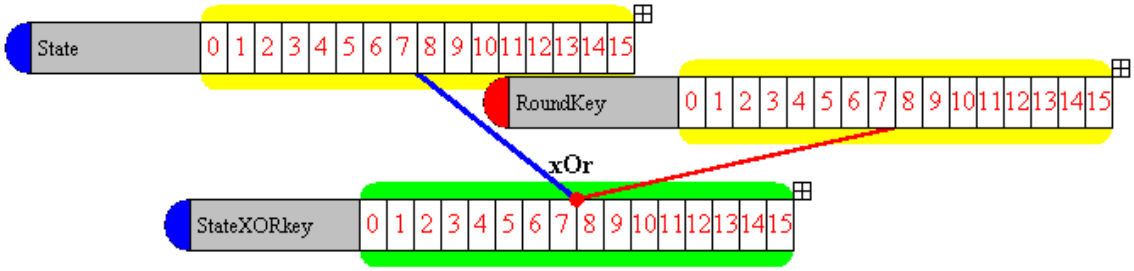


Figure 33: Add Round Key Screen Shot

All of these parts when combined form the basis of the AES encryption. The output code for each operation can be found in appendix G. I hope the benefits of my application are self evident from this example; as while the final stage would have taken a similar amount of time by hand, in the middle two stages, the application is much more efficient at creating the code. By hand this code would have taken approximately 2 hours to complete using SSE2 instructions, however using my application the time was just under 10 minutes to produce code for both instruction sets.

10. CONCLUSIONS

Looking back at the original aims of this project in §2.5, it appears that all the requirements have been met. The use of the survey to get peoples opinions on the different layout and design choices in Chapter 5 resulted in the program meeting the first requirement, to make an intuitive and simple to use application, as further demonstrated by the times recorded during testing in §8.5 This also shows that the application fulfils another aim, in improving productivity. Programming error has been reduced due to the numerous error checks that are present in the application, and again from the testing I have shown that the resulting code quality is of a high standard. The larger example at the end I hope helps to show all of these qualities, and the success of the application, the time saved being quite considerable.

I have learnt a lot from this project. Time management skills have been essential in creating the project in the allotted time, and meeting personal deadlines throughout. I've learnt the importance of planning programs, and looking further ahead when first starting. My programming skills have improved considerably throughout the project, as I learnt new skills and techniques. A fair amount of time was spent researching various aspects to both fully understand the work the project is aimed at and its background, as well as to provide a high quality application to the user. I spent a lot of time on this project and am pleased with the way it turned out, with its success proved above.

10.1. Future Work

Despite the success of the application, there are still a few of areas I would have liked to have pursued had time allowed. Firstly, a tool for automatically producing matrix multiplication code. As shown in the example above, matrix multiplication can be realised using SIMD instructions, and more efficiently than other means. However, often programming such a sequence can be taxing. So a tool to produce these automatically would be a pleasing addition. Another feature would be to have more instructions on a single line. Whereas my application requires a new variable to be initialised for each instruction, which is clearly not optimal, it would be a smart feature if the application could see which variables were not used elsewhere in the program, and concatenate a number of lines into a single instruction.
e.g.

```
c = __m_addepi8(a,b);
f = __m_subepi8(d,e);
g = __m_xor(c,f);
```

could be re-written as:

```
h = __m_xor(__m_addepi8(a,b), __m_subepi8(d,e));
```

Following on from the same idea, another useful addition would be to allow a user to set the names of the words to follow on. As stated above, a drawback of the application is that every word has to have a unique name. When programming however generally it is the case that the variable name will be carried through the program; my application as it stands does not support this. Adding more instruction sets to the application would also add to its usefulness.

Bibliography:

[1] C. Faulkner *The Essence of Human – Computer Interaction.*
ISBN: 0-13-751975-3. Date: 1998

[2] D. Sima, T. Fountain, P. Kacsuk *Advanced Computer Architecture: A Design Space Approach.*
ISBN: 0-201-42291-3. Date: 1997

[3] A. Appel *Modern Compiler Implementation in Java*
ISBN: 0-521-82060-X. Date: October 2002

[4] Deitel *Java How to Program*
ISBN: 0-131-48398-6. Date: July 1, 2003

[5] R. Eckstein, M. Loy, D. Wood *Java Swing*
ISBN: 1-565-92455-X. Date: 1998

[6] J. L. Hennessy & D. A. Patterson *Computer Architecture A Quantitative Approach.*
ISBN: 1-558-60596-7. Date: 2002

[7] Hwang Briggs *Computer Architecture and Parallel Processing.*
ISBN: 0-070-31556-6. Date: 1984

[8] Jenny Preece *Human-Computer Interaction*
ISBN: 0-471-49278-7. Date: 1994

[9] The Design of Rijndael *Joan Daemen and Vincent Rijmen*
ISBN: 3-540-42580-2. Date: 2002

[10] Application Specific Systems, Architectures, and Processors *Various*
ISBN: 0-7695-0716-6. Date: July 2002

[11] GCC ISA Extensions
http://personals.ac.upc.edu/alvarez/media/gcc-isa-extensions.html

[12] Usable GUI Design: A Quick Guide.
http://benroe.com/files/gui.html

[13] Flynn's Taxonomy
http://en.wikipedia.org/wiki/Flynn%27s_taxonomy

[14] SIMD
http://en.wikipedia.org/wiki/SIMD

[15] MMX, SSE, and SSE2 Intrinsics
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/vclrfMMXSSESSE2Intrisics.asp

[16] Streaming SIMD Extensions 2 Instructions
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/vcrefWillamettenewinstructions.asp

[17] AltiVec Technology: Programming Interface Manual
http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

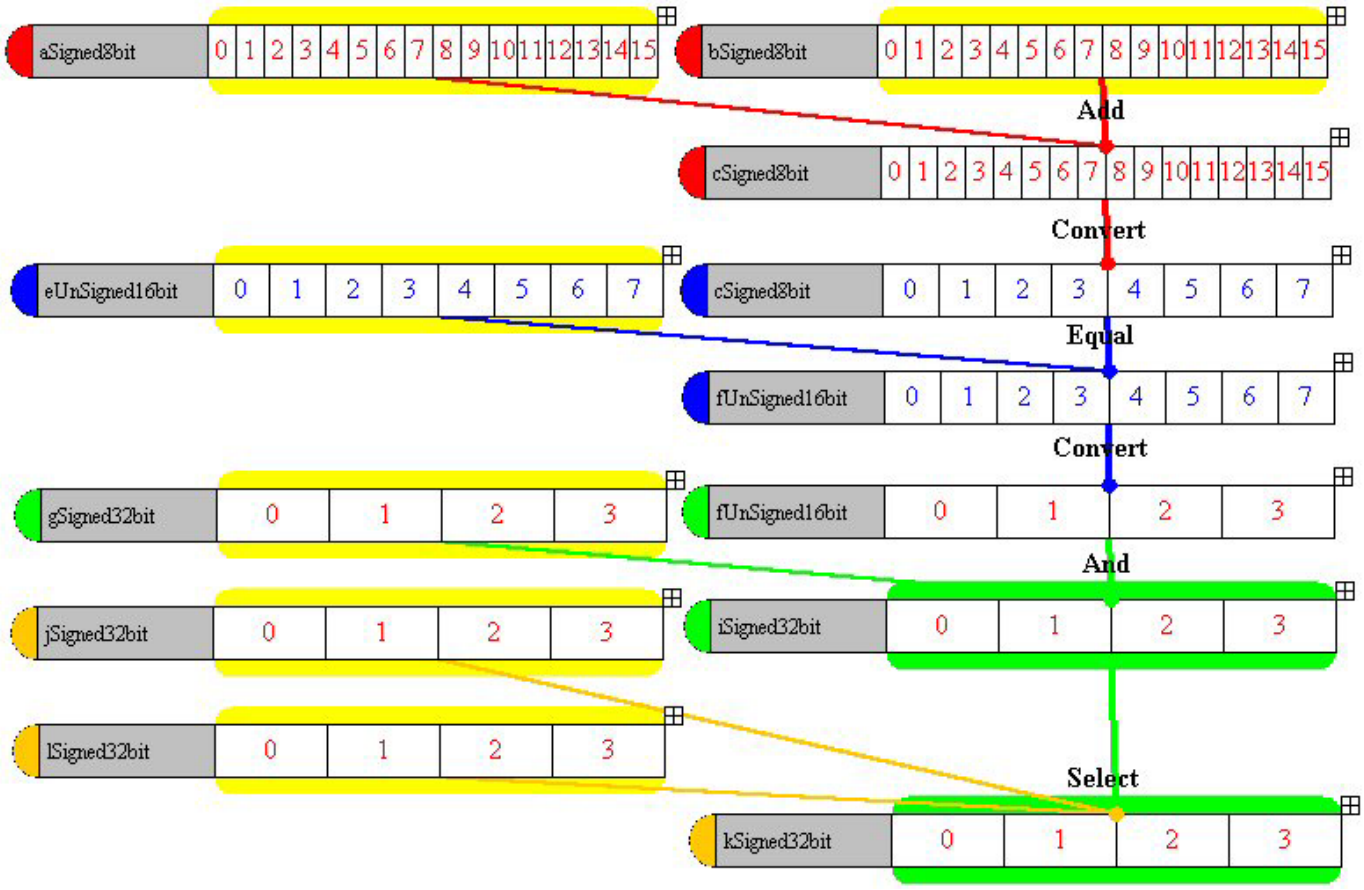
[18] Velocity Engine
http://developer.apple.com/hardwaredrivers/ve/index.html

[19] AltiVec/SSE Migration Guide
http://developer.apple.com/documentation/Performance/Conceptual/Accelerate_sse_migration/migration_intro/chapter_1_section_1.html

Appendix A:
Correctness of outputted code

Example Program:

I give here a single example program as drawn out in my application. A more substantial example can be seen in appendix G.



SSE2 output code:

```
// Initialise Variables

__m128i aSigned8bit;
__m128i bSigned8bit;
__m128i cSigned8bit;
__m128i eUnSigned16bit;
__m128i fUnSigned16bit;
__m128i gSigned32bit;
__m128i iSigned32bit;
__m128i jSigned32bit;
__m128i kSigned32bit;
__m128i lSigned32bit;

// Load

aSigned8bit = _mm_load_si128((__m128i*)p);
bSigned8bit = _mm_load_si128((__m128i*)p);
eUnSigned16bit = _mm_load_si128((__m128i*)p);
gSigned32bit = _mm_load_si128((__m128i*)p);
jSigned32bit = _mm_load_si128((__m128i*)p);
lSigned32bit = _mm_loadu_si128((__m128i*)p);

// Program
```

```
cSigned8bit = _mm_add_epi8(aSigned8bit,bSigned8bit);
fUnSigned16bit = _mm_cmpeq_epi16(eUnSigned16bit,cSigned8bit);
iSigned32bit = _mm_and_si128(gSigned32bit,fUnSigned16bit);
kSigned32bit = _mm_or_si128(_mm_and_si128(
lSigned32bit,iSigned32bit),_mm_and_si128( lSigned32bit,jSigned32bit))

// Store

iSigned32bit = _mm_store_si128((__m128i*)p, v);
kSigned32bit = _mm_store_si128((__m128i*)p, v);
```

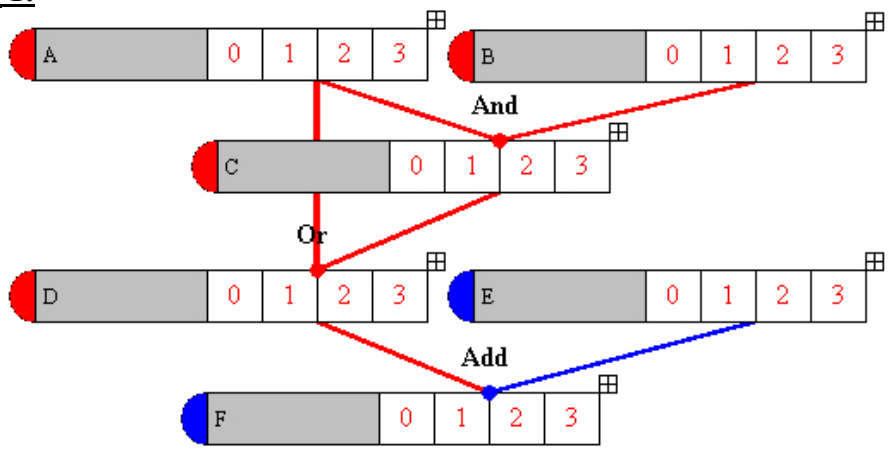
Altivec output code:

```
// Initialise Variables

__128i aSigned8bit;
__128i bSigned8bit;
__128i cSigned8bit;
__128i eUnSigned16bit;
__128i fUnSigned16bit;
__128i gSigned32bit;
__128i iSigned32bit;
__128i jSigned32bit;
__128i kSigned32bit;
__128i lSigned32bit;

// Load

aSigned8bit = vec_ld(0,p);
bSigned8bit = vec_ld(0,p);
eUnSigned16bit = vec_ld(0,p);
gSigned32bit = vec_ld(0,p);
jSigned32bit = vec_ld(0,p);
lSigned32bit = vec_ld(0,p);

// Program

cSigned8bit = vec_add(aSigned8bit,bSigned8bit);
fUnSigned16bit = vec_cmpeq(eUnSigned16bit,cSigned8bit);
iSigned32bit = vec_and(gSigned32bit,fUnSigned16bit);
kSigned32bit = vec_sel(jSigned32bit,iSigned32bit, lSigned32bit);

// Store

iSigned32bit = vec_st(v,0,p);
kSigned32bit = vec_st(v,0,p);
```

Appendix B:
Logical Optimisation

Here is an example of the logical optimisation present in my application. This first example shows how the two operations have been removed completely. The second example shows how the application is wary that the user may use an intermediate step, but still carries the source word though, instead of performing the 'Or' operation.

Example 1:



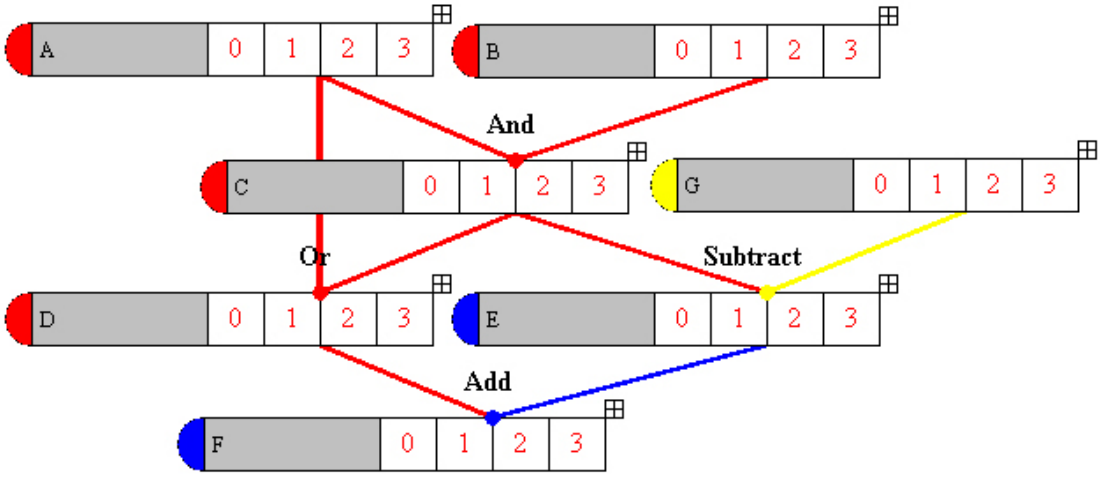
Outputted Code: SSE2

```
// Initialise Variables

__m128i A;
__m128i E;
__m128i F;

// Load

// Program

F = _mm_add_epi16(A,E);

// Store
```

Outputted Code: Altivec

```
// Initialise Variables

__128i A;
__128i E;
__128i F;

// Load

// Program

F = vec_add(A,E);

// Store
```

Example 2:



Outputted Code: SSE2

```
// Initialise Variables

__m128i A;
__m128i B;
__m128i C;
__m128i E;
__m128i F;
__m128i G;

// Load

// Program

C = _mm_and_si128(A,B);
E = _mm_sub_epi16(C,G);
F = _mm_add_epi16(A,E);

// Store
```

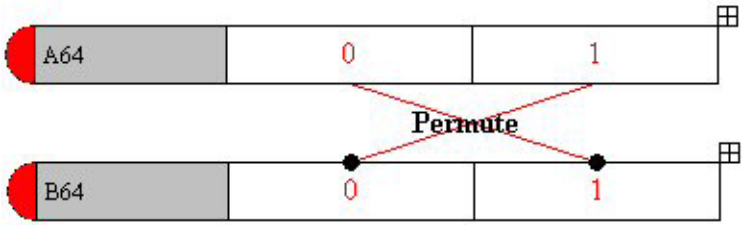
Outputted Code: Altivec

```
// Initialise Variables

__128i A;
__128i B;
__128i C;
__128i E;
__128i F;
__128i G;

// Load

// Program

C = vec_and(A,B);
E = vec_sub(C,G);
F = vec_add(A,E);

// Store
```

Appendix C:
Permutations

An example permutation in SSE2 instructions for each of the subword sizes supported. Both a screen shot and the corresponding output code has been supplied.

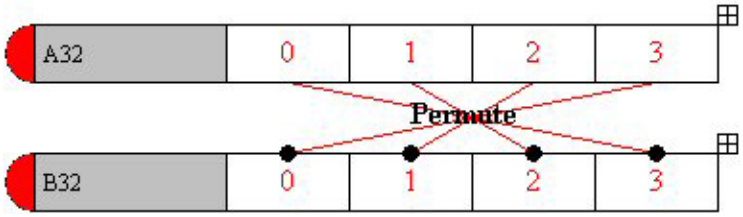
64bit Subword Permutation:



Outputted Code:

```
B64 = _mm_shuffle_epi32(A64,78);
```

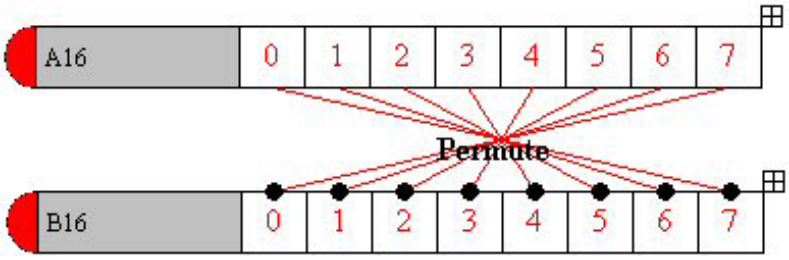
32bit Subword Permutation:



Outputted Code:

```
B32 = _mm_shuffle_epi32(B32,228);
```

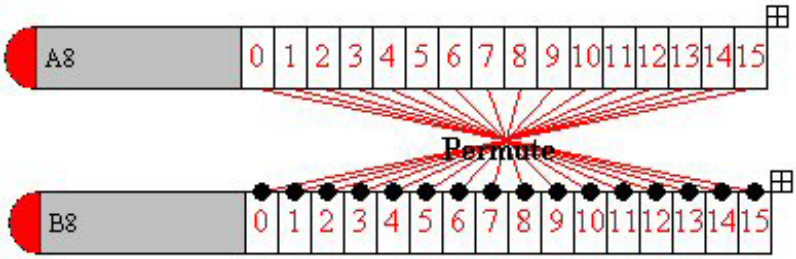
16bit Subword Permutation:



Outputted Code:

```
B16 = _mm_shuffle_epi32(A16,108);
B16 = _mm_shufflelo_epi16(B16,78);
B16 = _mm_shuffle_epi32(B16,216);
B16 = _mm_shufflelo_epi16(B16,177);
B16 = _mm_shufflehi_epi16(B16,27);
```

8bit Subword Permutation:



Outputted Code:

```
__m128i bytes1, bytes2, bytes3, bytes4;
char order1Bytes[16] __attribute__((aligned(16)))
      = { 0,15,0,15,0,15,0,15,0,15,0,15,0,15,0,15};
bytes1 = _mm_load_si128((__m128i*)order1Bytes);
a8Odd1 = _mm_and_si128(a8,bytes1);
char order2Bytes[16] __attribute__((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes2 = _mm_load_si128((__m128i*)order2Bytes);
a8Odd2 = _mm_and_si128(a8,bytes2);
char order3Bytes[16] __attribute__((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes3 = _mm_load_si128((__m128i*)order3Bytes);
a8Even1 = _mm_and_si128(a8,bytes3);
char order4Bytes[16] __attribute__((aligned(16)))
      = { 15,0,15,0,15,0,15,0,15,0,15,0,15,0,15,0};
bytes4 = _mm_load_si128((__m128i*)order4Bytes);
a8Even2 = _mm_and_si128(a8,bytes4);
a8Odd1 = _mm_shufflelo_epi16(a8Odd1,27);
a8Odd1 = _mm_shufflehi_epi16(a8Odd1,75);
a8Odd1 = _mm_shuffle_epi32(a8Odd1,216);
a8Odd1 = _mm_shufflelo_epi16(a8Odd1,30);
a8Odd1 = _mm_shufflehi_epi16(a8Odd1,27);
a8Odd1 = _mm_shuffle_epi32(a8Odd1,216);
a8Odd1 = _mm_shufflehi_epi16(a8Odd1,177);
a8Even2 = _mm_shufflelo_epi16(a8Even2,27);
a8Even2 = _mm_shufflehi_epi16(a8Even2,75);
a8Even2 = _mm_shuffle_epi32(a8Even2,216);
a8Even2 = _mm_shufflelo_epi16(a8Even2,30);
a8Even2 = _mm_shufflehi_epi16(a8Even2,27);
a8Even2 = _mm_shuffle_epi32(a8Even2,216);
a8Even2 = _mm_shufflehi_epi16(a8Even2,177);
a8Odd1 = _mm_srli_si128(a8Odd1,1);
a8Even2 = _mm_slli_si128(a8Even2,1);
b8 = _mm_or_si128(a8Odd1,a8Odd2);
b8 = _mm_or_si128(b8,a8Even1);
b8 = _mm_or_si128(b8,a8Even2);
```

Appendix D:
Testing the C Library

Below are 3 example tests of the supplied C Library, with the output in each case shown below.

Average 8bit Signed:

C Code:

```
#include "cLibrary.h"

int main(void){
      int i = 0;

      char nos[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
      char a[16] = {0,1,2,3,4,5,6,7,8,-9,-10,-11,-12,-13,-14,-15};
      char b[16] = {15,14,13,12,11,10,9,-8,-7,-6,-5,-4,-3,-2,-1,0};

      char *c = average8s(a,b);

      printf("\na\n");
      for(i = 0; i < 16; i++) {
        printf("%d -> average8s(%d,%d) = %d\n",
               nos[i], a[i], b[i], c[i]);
      }

    return 0;
}
```

Output:

```
0 -> average8s(0,15) = 8
1 -> average8s(1,14) = 8
2 -> average8s(2,13) = 8
3 -> average8s(3,12) = 8
4 -> average8s(4,11) = 8
5 -> average8s(5,10) = 8
6 -> average8s(6,9) = 8
7 -> average8s(7,-8) = 0
8 -> average8s(8,-7) = 1
9 -> average8s(-9,-6) = -7
10 -> average8s(-10,-5) = -7
11 -> average8s(-11,-4) = -7
12 -> average8s(-12,-3) = -7
13 -> average8s(-13,-2) = -7
14 -> average8s(-14,-1) = -7
15 -> average8s(-15,0) = -7
```

Absolute Saturated 32bit:

C Code:

```
#include "cLibrary.h"

int main(void){
      int i = 0;

      int nos[4] = {0,1,2,3};
      int a[4] = {0,99999,-999999,-12345};

      int *b = absoluteSaturated32s(a);

      for(i = 0; i < 4; i++) {
```

```
        printf("%d -> absoluteSaturated32s(%d) = %d\n",
               nos[i], a[i], b[i]);
      }

    return 0;
}
```

Output:

```
0 -> absoluteSaturated32s(0) = 0
1 -> absoluteSaturated32s(99999) = 99999
2 -> absoluteSaturated32s(-999999) = 999999
3 -> absoluteSaturated32s(-12345) = 12345
```

Max 32bit Unsigned:

C Code:

```
#include "cLibrary.h"

int main(void){
      int i = 0;

      unsigned short nos[8] = {0,1,2,3,4,5,6,7};
      unsigned short a[8] = {0,10,20,30,40,50,60,-50};
      unsigned short b[8] = {0,5,25,25,45,45,65,65};

      unsigned short *c = max16u(a,b);

      for(i = 0; i < 8; i++) {
        printf("%d -> max16u(%d,%d) = %d\n",nos[i], a[i],b[i],c[i]);
      }

    return;
}
```

Output:

```
0 -> max16u(0,0) = 0
1 -> max16u(10,5) = 10
2 -> max16u(20,25) = 25
3 -> max16u(30,25) = 30
4 -> max16u(40,45) = 45
5 -> max16u(50,45) = 50
6 -> max16u(60,65) = 65
7 -> max16u(65486,65) = 65486
```

Appendix E:
Testing SSE2 Special Ops

A couple of examples of the special functions supplied by my application to perform those operations not directly supported by an instruction set.

vec_abs(a) :

```
t = _mm_cmpgt_epi8(zero,a);
a = _mm_sub_epi8(_mm_xor_si128(a,t),t);
```

C Code:

```
#include <stdio.h>
#include <xmmintrin.h>
#include <unistd.h>
#include <sys/time.h>

__m128i a, t, zero;
char inp1[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
char inp_sse1[16] __attribute__((aligned(16))) = { 0, 1, 2, 3, 4, 5,
6, 7, 8, -1, -2, -3, -4, -5, -6, -7 };
char zero2[16] __attribute__((aligned(16))) = { 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0};
char out[16];

int main(void){
      int i = 0;

      a = _mm_load_si128((__m128i*)inp_sse1);
      zero = _mm_load_si128((__m128i*)zero2);

      printf("a\n");
      _mm_store_si128((__m128i*)out,a);
      for(i = 0; i < 16; i++) {
        printf("%d -> %d\n",inp1[i], out[i]);
      }

      t = _mm_cmpgt_epi8(zero,a);
      a = _mm_sub_epi8(_mm_xor_si128(a,t),t);

      printf("\nABS(a)\n");
      _mm_store_si128((__m128i*)out,a);
      for(i = 0; i < 16; i++) {
        printf("%d -> %d\n",inp1[i], out[i]);
      }
      printf("\n");

      return 0;
}
```

Output:

```
a
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> -1
10 -> -2
11 -> -3
```

```
12 -> -4
13 -> -5
14 -> -6
15 -> -7

ABS(a)
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 1
10 -> 2
11 -> 3
12 -> 4
13 -> 5
14 -> 6
15 -> 7
```

vec_max(a,b) :

```
t = _mm_cmpgt_epi8(a,b);
c = _mm_or_si128(_mm_andnot_si128(t,b),_mm_and_si128(t,a));
```

C Code:

```
#include <stdio.h>
#include <xmmintrin.h>
#include <unistd.h>
#include <sys/time.h>

__m128i a, b, c, t, zero;
char inp1[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
char inp_sse1[16] __attribute__((aligned(16))) = { 0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
char inp_sse2[16] __attribute__((aligned(16))) = { 15, 14, 13, 12, 11,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
char zero2[16] __attribute__((aligned(16))) = { 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0};
char out[16];

int main(void){
      int i = 0;

      a = _mm_load_si128((__m128i*)inp_sse1);
      b = _mm_load_si128((__m128i*)inp_sse2);
      zero = _mm_load_si128((__m128i*)zero2);

      printf("a\n");
      _mm_store_si128((__m128i*)out,a);
      for(i = 0; i < 16; i++) {
        printf("%d -> %d\n",inp1[i], out[i]);
      }

      printf("\nb\n");
      _mm_store_si128((__m128i*)out,b);
      for(i = 0; i < 16; i++) {
        printf("%d -> %d\n",inp1[i], out[i]);
      }
```

```
        t = _mm_cmpgt_epi8(a,b);
        c = _mm_or_si128(_mm_andnot_si128(t,b),_mm_and_si128(t,a));

        printf("\nMAX(a,b)\n");
        _mm_store_si128((__m128i*)out,c);
        for(i = 0; i < 16; i++) {
          printf("%d -> %d\n",inp1[i], out[i]);
        }
        printf("\n");

        return 0;
}
```

Output:

```
a
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 9
10 -> 10
11 -> 11
12 -> 12
13 -> 13
14 -> 14
15 -> 15

b
0 -> 15
1 -> 14
2 -> 13
3 -> 12
4 -> 11
5 -> 10
6 -> 9
7 -> 8
8 -> 7
9 -> 6
10 -> 5
11 -> 4
12 -> 3
13 -> 2
14 -> 1
15 -> 0

MAX(a,b)
0 -> 15
1 -> 14
2 -> 13
3 -> 12
4 -> 11
5 -> 10
6 -> 9
7 -> 8
8 -> 8
9 -> 9
10 -> 10
11 -> 11
12 -> 12
```

```
13 -> 13
14 -> 14
15 -> 15
```

Appendix F:
8bit Permutation Speed: SSE2 V's Basic C code

An example piece of code used to test the speed of the SSE2 permutations output by my application, compared with using basic C code.

```
#include <stdio.h>
#include <xmmintrin.h>
#include <unistd.h>
#include <sys/time.h>

// For C:
// Initital order of the word
char inp1[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
char out[16];

// For SSE2
// Initialise variables
__m128i one, b, c, oneEven1, oneEven2, oneOdd1, oneOdd2, even, odd,
zero;

// Create the original word
char inp_sse1[16] __attribute__((aligned(16))) = { 0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
char out_sse[16];

// The masks created to assist the permutation
char evenTemp[16]  __attribute__((aligned(16))) = { 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
char even2Temp[16] __attribute__((aligned(16))) = { 0, 15, 0, 15, 0,
15, 0, 15, 0, 15, 0, 15, 0, 15, 0, 15};
char oddTemp[16]   __attribute__((aligned(16))) = { 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
char odd2Temp[16]  __attribute__((aligned(16))) = { 15, 0, 15, 0, 15,
0, 15, 0, 15, 0, 15, 0, 15, 0, 15, 0};

// Used for the timing
typedef unsigned long long uint64;

int main(void){
  int i=0;
  int j=0;
  double diffOps;
  double diffTimes;
  uint64 timeStart, timeEnd, opsStart, opsEnd;

  // Counts the number of operations
  inline uint64 grab_tsc()
  {
    register uint32 __a;
    register uint32 __d;

    __asm__ __volatile__( "rdtsc" : "=a"( __a ), "=d"( __d ) );

    return ( ( uint64 )( __d ) << 32 ) |
           ( ( uint64 )( __a ) <<  0 ) ;
  }

  // Gets the current time.
  inline uint64 grab_tod()
  {
```

```
  struct timeval  tv;
  struct timezone tz;

  gettimeofday( &tv, &tz );

  return ( tv.tv_sec * 1000 * 1000 ) + tv.tv_usec;
}

// Get time at start
opsStart  = grab_tsc();
// Get number of ops at start
timeStart = grab_tod();

// Perform the C permute, a lot of times to get a good average
for(j=0; j < 1000000000; j++){
      for(k=0; k < 16; k++){
          out[k]  = inp1[15 - k];
    }
}

// Get end time
opsEnd  = grab_tsc();
// and end number of ops
timeEnd = grab_tod();

// Calculate the differences
diffOps = (double)(opsEnd - opsStart);
diffTimes = 0.000001*(double)(timeEnd - timeStart);

// Print out the results
printf("\nC: ");
printf("\nTime: %f", diffTimes);
printf("\nOps:  %f\n", diffOps);

// Get time at start
opsStart  = grab_tsc();
// Get number of ops at start
timeStart = grab_tod();

// Load the details
even  = _mm_load_si128((__m128i*)evenTemp);
even2 = _mm_load_si128((__m128i*)even2Temp)
odd   = _mm_load_si128((__m128i*)oddTemp);;
odd2  = _mm_load_si128((__m128i*)odd2Temp);
one   = _mm_load_si128((__m128i*)inp_sse1);

// Perform the permute a number of times
for(j=0; j < 1000000000; j++){
  oneEven1 = _mm_and_si128(one,even);
  oneEven2 = _mm_and_si128(one,even2);
  oneOdd1 = _mm_and_si128(one,odd);
  oneOdd2 = _mm_and_si128(one,odd2);
  oneOdd1 = _mm_shufflelo_epi16(oneOdd1,228);
  oneOdd1 = _mm_shufflehi_epi16(oneOdd1,228);
  oneOdd2 = _mm_shuffle_epi32(oneOdd2,108);
  oneOdd2 = _mm_shufflelo_epi16(oneOdd2,78);
  oneOdd2 = _mm_shuffle_epi32(oneOdd2,216);
  oneOdd2 = _mm_shufflelo_epi16(oneOdd2,177);
  oneOdd2 = _mm_shufflehi_epi16(oneOdd2,27);
  oneEven1 = _mm_shufflelo_epi16(oneEven1,228);
  oneEven1 = _mm_shufflehi_epi16(oneEven1,228);
  oneEven2 = _mm_shuffle_epi32(oneEven2,108);
  oneEven2 = _mm_shufflelo_epi16(oneEven2,78);
  oneEven2 = _mm_shuffle_epi32(oneEven2,216);
```

```
    oneEven2 = _mm_shufflelo_epi16(oneEven2,177);
    oneEven2 = _mm_shufflehi_epi16(oneEven2,27);
    oneOdd2 = _mm_slli_si128(oneOdd2,1);
    oneEven2 = _mm_srli_si128(oneEven2,1);
    one = _mm_or_si128(oneOdd1,oneOdd2);
    one = _mm_or_si128(one,oneEven1);
    one = _mm_or_si128(one,oneEven2);
  }

  // Get end time
  opsEnd  = grab_tsc();
  // and end ops
  timeEnd = grab_tod();

  // Calulate the difference
  diffOps = (double)(opsEnd - opsStart);
  diffTimes = 0.000001*(double)(timeEnd - timeStart);

  // Print out the results
  printf("\nSSE2: ");
  printf("\nTime: %f", diffTimes);
  printf("\nOps:  %f\n", diffOps);

  return 0;
}
```

Appendix G:
AES Code

The output code of my application for each of the operations performed in the AES encryption as described in §9.2. A screen shot for the Mix Column operation is also supplied

Shift Rows:
A screen shot was supplied for this in §9.2.2. I provide here the output code for such an operation:

SSE2:
```
// Initialise Variables
__m128i state;
__m128i shiftedtRows;

// Load
state = _mm_load_si128((__m128i*)p);

// Program
__m128i bytes1, bytes2, bytes3, bytes4;
char order1Bytes[16] __attribute__((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes1 = _mm_load_si128((__m128i*)order1Bytes);
stateOdd1 = _mm_and_si128(state,bytes1);
char order2Bytes[16] __attribute__((aligned(16)))
      = { 0,15,0,15,0,15,0,15,0,15,0,15,0,15,0,15};
bytes2 = _mm_load_si128((__m128i*)order2Bytes);
stateOdd2 = _mm_and_si128(state,bytes2);
char order3Bytes[16] __attribute__((aligned(16)))
      = { 15,0,15,0,15,0,15,0,15,0,15,0,15,0,15,0};
bytes3 = _mm_load_si128((__m128i*)order3Bytes);
stateEven1 = _mm_and_si128(state,bytes3);
char order4Bytes[16] __attribute__((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes4 = _mm_load_si128((__m128i*)order4Bytes);
stateEven2 = _mm_and_si128(state,bytes4);
stateOdd2 = _mm_shufflelo_epi16(stateOdd2,156);
stateOdd2 = _mm_shufflehi_epi16(stateOdd2,198);
stateOdd2 = _mm_shuffle_epi32(stateOdd2,216);
stateOdd2 = _mm_shufflehi_epi16(stateOdd2,45);
stateOdd2 = _mm_shufflelo_epi16(stateOdd2,198);
stateEven1 = _mm_shufflelo_epi16(stateEven1,216);
stateEven1 = _mm_shufflehi_epi16(stateEven1,45);
stateEven1 = _mm_shuffle_epi32(stateEven1,216);
stateEven1 = _mm_shufflehi_epi16(stateEven1,99);
stateEven1 = _mm_shufflelo_epi16(stateEven1,216);
stateOdd1 = _mm_srli_si128(stateOdd1,1);
stateEven2 = _mm_slli_si128(stateEven2,1);
shiftedtRows = _mm_or_si128(stateOdd1,stateOdd2);
shiftedtRows = _mm_or_si128(shiftedtRows,stateEven1);
shiftedtRows = _mm_or_si128(shiftedtRows,stateEven2);

// Store
shiftedtRows = _mm_store_si128((__m128i*)p, v);
```

Altivec:
```
// Initialise Variables
__128i state;
__128i shiftedtRows;
```

```
// Load
state = vec_ld(0,p);

// Program
shiftedtRows =
vec_perm(state,state,67703239093342157453309360348923395);

// Store
shiftedtRows = vec_st(v,0,p);
```

Add Round Key:

Again a screen shot was supplied for this in §9.2.4. I provide here the output code for such an operation:

SSE2:

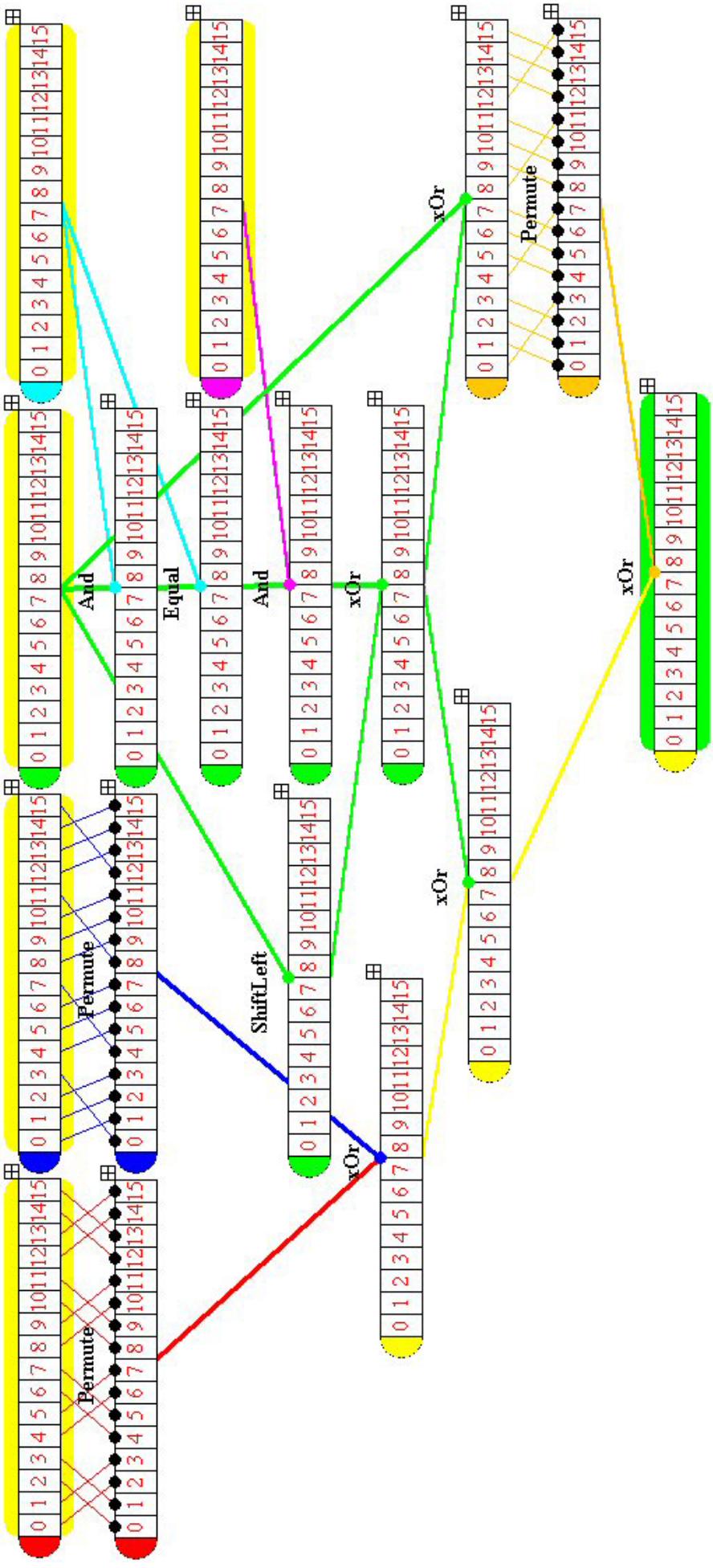
```
// Initialise Variables
  m128i State;
  m128i RoundKey;
  m128i StateXORkey;

// Load
state =  mm load si128((  m128i*)p);
State =  mm load si128((  m128i*)p);
RoundKey =  mm load si128((  m128i*)p);

// Program
StateXORkey =  mm xor si128(State,RoundKey);

// Store
shiftedtRows =  mm store si128((  m128i*)p, v);
StateXORkey =  mm storeu si128((  m128i*)p, v);
```

Altivec:

```
// Initialise Variables
__128i State;
__128i RoundKey;
__128i StateXORkey;

// Load
state = vec_ld(0,p);
State = vec_ld(0,p);
RoundKey = vec_ld(0,p);

// Program
StateXORkey = vec_xor(State,RoundKey);

// Store
shiftedtRows = vec_st(v,0,p);
StateXORkey = vec_st(v,0,p);
```

Mix Columns:

The screen shot for this operation was too large to include in the body of the text. So it has been included below. Under that is the corresponding output code.


And
Equal
And
xOr
xOr
Permute
xOr
Permute
ShiftLeft
xOr
Permute
xOr

SSE2:

```
// Initialise Variables
  __m128i RowA;
  __m128i RowA2;
  __m128i RowB;
  __m128i RowB2;
  __m128i RowC;
  __m128i RowC1;
  __m128i RowC2;
  __m128i RowC3;
  __m128i RowC4;
  __m128i RowC5;
  __m128i RowC6;
  __m128i 0x80;
  __m128i 0x1B;
  __m128i RowD;
  __m128i RowD2;
  __m128i Result1;
  __m128i Result2;
  __m128i Result;

// Load
RowA =  _mm_load_si128((  __m128i*)p);
RowB =  _mm_load_si128((  __m128i*)p);
RowC =  _mm_load_si128((  __m128i*)p);
0x80 =  _mm_load_si128((  __m128i*)p);
0x1B =  _mm_load_si128((  __m128i*)p);

// Program
  __m128i bytes1, bytes2, bytes3, bytes4;
char order1Bytes[16]   __attribute__  ((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes1 =  _mm_load_si128((__m128i*)order1Bytes);
RowAOdd1 =  _mm_and_si128(RowA,bytes1);
char order2Bytes[16]   __attribute__  ((aligned(16)))
      = { 0,15,0,15,0,15,0,15,0,15,0,15,0,15,0,15};
bytes2 =  _mm_load_si128((  __m128i*)order2Bytes);
RowAOdd2 =  _mm_and_si128(RowA,bytes2);
char order3Bytes[16]   __attribute__  ((aligned(16)))
      = { 15,0,15,0,15,0,15,0,15,0,15,0,15,0,15,0};
bytes3 =  _mm_load_si128((  __m128i*)order3Bytes);
RowAEven1 =  _mm_and_si128(RowA,bytes3);
char order4Bytes[16]   __attribute__  ((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes4 =  _mm_load_si128((  __m128i*)order4Bytes);
RowAEven2 =  _mm_and_si128(RowA,bytes4);
RowAOdd2 =  _mm_shufflehi_epi16(RowAOdd2,177);
RowAOdd2 =  _mm_shufflelo_epi16(RowAOdd2,177);
RowAEven1 =  _mm_shufflehi_epi16(RowAEven1,177);
RowAEven1 =  _mm_shufflelo_epi16(RowAEven1,177);
RowAOdd1 =  _mm_srli_si128(RowAOdd1,1);
RowAEven2 =  _mm_slli_si128(RowAEven2,1);
RowA2 =  _mm_or_si128(RowAOdd1,RowAOdd2);
RowA2 =  _mm_or_si128(RowA2,RowAEven1);
RowA2 =  _mm_or_si128(RowA2,RowAEven2);
char order1Bytes[16]   __attribute__  ((aligned(16)))
      = { 0,15,0,15,0,15,0,15,0,15,0,15,0,15,0,15};
bytes1 =  _mm_load_si128((  __m128i*)order1Bytes);
RowBOdd1 =  _mm_and_si128(RowB,bytes1);
char order2Bytes[16]   __attribute__  ((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes2 =  _mm_load_si128((  __m128i*)order2Bytes);
RowBOdd2 =  _mm_and_si128(RowB,bytes2);
char order3Bytes[16]   __attribute__  ((aligned(16)))
```

```
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes3 =  mm load si128((  m128i*)order3Bytes);
RowBEven1 =  mm and si128(RowB,bytes3);
char order4Bytes[16]  __attribute__ ((aligned(16)))
      = { 15,0,15,0,15,0,15,0,15,0,15,0,15,0,15,0};
bytes4 =  mm load si128((  m128i*)order4Bytes);
RowBEven2 =  mm and si128(RowB,bytes4);
RowBEven2 =  mm shufflehi epi16(RowBEven2,177);
RowBEven2 =  mm shufflelo epi16(RowBEven2,177);
RowBOdd1 =  mm srli si128(RowBOdd1,1);
RowBEven2 =  mm slli si128(RowBEven2,1);
RowB2 =  mm or si128(RowBOdd1,RowBOdd2);
RowB2 =  mm or si128(RowB2,RowBEven1);
RowB2 =  mm or si128(RowB2,RowBEven2);
RowC2 =  mm and si128(RowC,0x80);
RowC3 =  mm cmpeq epi8(RowC2,0x80);
RowC4 =  mm and si128(RowC3,0x1B);
RowC5 =  mm slli si128(RowC,FILL IN);
RowC6 =  mm xor si128(RowC5,RowC4);
RowD =  mm xor si128(RowC6,RowC);
char order1Bytes[16]  attribute  ((aligned(16)))
      = { 0,15,0,15,0,15,0,15,0,15,0,15,0,15,0,15};
bytes1 =  mm load si128((  m128i*)order1Bytes);
RowDOdd1 =  mm and si128(RowD,bytes1);
char order2Bytes[16]  attribute  ((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes2 =  mm load si128((  m128i*)order2Bytes);
RowDOdd2 =  mm and si128(RowD,bytes2);
char order3Bytes[16]  attribute  ((aligned(16)))
      = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
bytes3 =  mm load si128((  m128i*)order3Bytes);
RowDEven1 =  mm and si128(RowD,bytes3);
char order4Bytes[16]  attribute  ((aligned(16)))
      = { 15,0,15,0,15,0,15,0,15,0,15,0,15,0,15,0};
bytes4 =  mm load si128((  m128i*)order4Bytes);
RowDEven2 =  mm and si128(RowD,bytes4);
RowDOdd1 =  mm shufflehi epi16(RowDOdd1,177);
RowDOdd1 =  mm shufflelo epi16(RowDOdd1,177);
RowDOdd1 =  mm srli si128(RowDOdd1,1);
RowDEven2 =  mm slli si128(RowDEven2,1);
RowD2 =  mm or si128(RowDOdd1,RowDOdd2);
RowD2 =  mm or si128(RowD2,RowDEven1);
RowD2 =  mm or si128(RowD2,RowDEven2);
Result1 =  mm xor si128(RowA2,RowB2);
Result2 =  mm xor si128(Result1,RowC6);
Result =  mm xor si128(Result2,RowD2);

// Store
Result =  mm store si128((  m128i*)p, v);
```

Altivec:

```
// Initialise Variables
  128i RowA;
  128i RowA2;
  128i RowB;
  128i RowB2;
  128i RowC;
  128i RowC1;
  128i RowC2;
  128i RowC3;
  128i RowC4;
  128i RowC5;
  128i RowC6;
  128i 0x80;
```

```
  128i 0x1B;
  128i RowD;
  128i RowD2;
  128i Result1;
  128i Result2;
  128i Result;

// Load
RowA = vec_ld(0,p);
RowB = vec_ld(0,p);
RowC = vec_ld(0,p);
0x80 = vec_ld(0,p);
0x1B = vec_ld(0,p);

// Program
RowA = vec_perm(RowA,RowA,267403296323899026504624886689923380 5);
RowB = vec_perm(RowB,RowB,133967343828550821820025462103461454 0);
RowC2 = vec_and(RowC,0x80);
RowC3 = vec_cmpeq(RowC2,0x80);
RowC4 = vec_and(RowC3,0x1B);
RowC5 = vec_sl(RowC);
RowC6 = vec_xor(RowC5,RowC4);
RowD = vec_xor(RowC6,RowC);
RowD2 = vec_perm(RowD,RowD,398770443039193079424616837647355419 0);
Result1 = vec_xor(RowA2,RowB2);
Result2 = vec_xor(Result1,RowC6);
Result = vec_xor(Result2,RowD2);

// Store
Result = vec_st(v,0,p);
```