

# Vector Processor

M.G. v/d Horst

December 4, 2007

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Processor</b>	<b>4</b>
2.1	RISC . . . . .	4
2.2	Vectors . . . . .	4
2.3	Adding instructions . . . . .	5
<b>3</b>	<b>Algorithms</b>	<b>6</b>
<b>4</b>	<b>Basic Matrix and Vector calculations</b>	<b>7</b>
4.1	Matrix Transposition . . . . .	7
4.2	Matrix-Vector multiplication . . . . .	8
4.2.1	Vectorize . . . . .	9
4.2.2	Rotate . . . . .	10
4.2.3	Shift . . . . .	11
4.2.4	Shuffle . . . . .	12
4.2.5	Matrices of different sizes . . . . .	12
4.2.6	Special Matrices . . . . .	13
4.2.7	Conclusion . . . . .	13
4.3	Matrix-Matrix multiplication . . . . .	14
<b>5</b>	<b>Number Representations</b>	<b>15</b>
5.1	Large numbers . . . . .	15
5.1.1	V large numbers . . . . .	15
5.1.2	A single large number . . . . .	16
5.2	Complex numbers . . . . .	16
<b>6</b>	<b>Filtering</b>	<b>19</b>
6.1	FIR . . . . .	19
6.2	Scaler . . . . .	19
6.3	IIR . . . . .	19

<b>7</b>	<b>Transforms</b>	<b>20</b>
7.1	Discrete Cosine Transform . . . . .	20
7.2	Fourier Transform . . . . .	20
<b>8</b>	<b>Finite State Machines</b>	<b>23</b>
8.1	Deterministic . . . . .	23
8.2	Non-Deterministic . . . . .	24
<b>9</b>	<b>Other Algorithms</b>	<b>28</b>
9.1	Rabin-Miller . . . . .	28
9.1.1	Algorithm . . . . .	28
9.1.2	Implementation . . . . .	29
9.2	CRC Checks . . . . .	31
9.3	Searches . . . . .	31
9.3.1	Binary Search . . . . .	31
9.3.2	Linear Search . . . . .	31
<b>10</b>	<b>Conclusions</b>	<b>32</b>

# Chapter 1

## Introduction

When we think of parallel computing, the first thing that springs to mind is often a picture of a vast number of processors working together to compute some desired result. This architecture, when all the processors do their own thing, is called Multiple Instructions Multiple Data (MIMD). In effect the whole architecture can be seen as a big processor capable of performing multiple instructions on multiple pieces of data (hence MIMD).

Next to the MIMD architecture is the Single Instruction Multiple Data (SIMD) architecture. In this architecture all the processors follow the same instructions, which is, of course, not as flexible as the MIMD architecture.

So why would we want to use the SIMD architecture? The answer is that the SIMD architecture is a lot simpler: we can actually build a big processor that behaves as the processors in the SIMD architecture. The control circuit of the big processor (which decodes the instructions and controls the arithmetic circuits) would be of roughly the same complexity as that of a small processor. Furthermore, we do not have to concern ourselves with synchronization and communication issues between the small processors, since they are all performing the same operation at the same time.

The vector processor which we examine in this report is a SIMD processor. We will examine the implementation of different algorithms on this processor. We hope to accomplish two goals. One is to discover which instructions should be in a vector processor, or at least which instructions are used in multiple algorithms. Secondly, we hope to discover a way of implementing algorithms on a vector processor when the implementation on a scalar processor is given.

# Chapter 2

## Processor

The goal of this report is to find the instructions that would be useful to include in a vector processor. To do this we will start out with a very simple processor and see which operations are necessary or useful for the implementation of the algorithms we examine.

### 2.1 RISC

We will start out with a RISC-like architecture for the vector processor. Specialized instructions can always be added later on.

First of all, the vector processor will have to be able to function like a general purpose RISC processor. This means that it has a large number of registers, that interaction with the memory is only done by reading to and writing from these registers, and that each instruction takes only one clock cycle.

Arithmetic and logical operations are performed on registers of  $W$  bits and the results are written to one or more registers. We will assume that common logical and arithmetic operations like addition, subtraction, multiplication, division, comparison, bit-wise and, bit-wise or, etc. are available. Also it is possible to assign a value to a register, and to copy values from register to register.

Furthermore some program control instructions are available (i.e. jumps and conditional jumps).

We will not concern ourselves (much) with the number representation in these registers. In practice the processor might have separate addition instructions, depending on whether the register contains an integer or a floating point number. But we will just assume that the appropriate representation and operations for the algorithm are chosen.

### 2.2 Vectors

The processor has the ability to work with scalars, like any other general purpose processor. But it also has the ability to work with vectors, which is row of scalars.

The number of elements in a vector is  $V$  and the word-size of the elements is  $W$  (i.e.  $W$  bits are used to represent the value of an element). The processor has several registers to contain such vectors, and the operations possible on these registers do not differ from the ones that can be performed on scalars.

To define some of the operations on vectors we will consider them to be arrays with  $V$  elements which can be indexed from 0. So  $VR_2(0)$  is the first element of vector register  $VR_2$ . Note that the processor does not necessarily need to be able to refer to vector elements (to copy the value of  $VR_1(3)$  to a scalar register for example). For many algorithms working with vectors will be enough, for others being able to refer to the first and last element of a vector suffices.

So, all the operations which can be performed on scalars can be performed element-wise on vectors. So instead of adding register  $R_1$  to register  $R_2$  and putting the result in register  $R_3$ , we can add vector register  $VR_1$  to vector register  $VR_2$  and put the result in  $VR_3$ . The net effect will be that the first statement performs 1 addition, while the second one performs  $V$  additions.

## 2.3 Adding instructions

In the next chapters, where we will consider different algorithms. In those chapters we will propose new instructions to add to the processor.

However, the idea is not to increase the instruction set until it has a special instruction for every algorithm. Rather the idea is to find a small set of instructions which can be used for many purposes.

It is tempting to add instructions to the vector processor which perform complicated operations like calculating the inproduct of two vectors or which add all the elements of the vector together. But generally we try to restrict ourselves to instructions which can be performed in  $O(1)$  time, as opposed to operations which take a processing time depending on  $V$ . Because if the processing time depends on  $V$  it would be hard to create a vector processor with a larger vector size, without having to increase the cycle time of that new processor.

## Chapter 3

# Algorithms

In the next chapters we will try to implement some algorithms on a vector processor. And we hope to find useful instructions to add to the vector processors instruction set.

But why would we want to implement an algorithm on a vector processor? After all, a general purpose processor can execute those algorithms just fine. The answer is speed. A vector processor can execute  $V$  calculations simultaneously, so perhaps it can complete the algorithm in  $\frac{1}{V}$ -th the time it takes a general purpose processor.

This speed increase can happen in two ways. First of all, we can try to reduce the time it takes to calculate a result. Secondly, if the algorithm is of a type that produces multiple results (like a filter for example), then we can try to calculate several results simultaneously. The net effect of both of these methods is the same; the speed increases. But what happens behind the scenes differs.

Another reason one might want to use a vector processor, is to execute several instances of the algorithm in parallel. An example could be to perform the same filtering operation on different streams. Or to perform different filtering operations on the same streams, or even to perform different filtering operations on different streams. While the individual execution of the algorithm does not improve with this method, the fact remains that all those algorithms together can be executed in parallel.

## Chapter 4

# Basic Matrix and Vector calculations

Calculations with matrices and vectors form an important part of many algorithms. So, not surprisingly, we will study them first.

In general the vector processor is well suited to speed up these calculations. Therefore, we will not give much attention to speeding up multiple calculations, or performing multiple calculations simultaneously.

The speed up achieved in one calculation can easily be used to perform multiple calculations faster. Furthermore, multiple calculations simultaneously often comes down to doing the same calculation, but with a different matrix or vector. So that is why, in this chapter, we will only focus on speeding up a single calculation.

### 4.1 Matrix Transposition

Matrix transposition is an operation that does not require any actual calculation. Given a  $M \times N$  matrix  $C$  we want to get its transposed  $C^T$ , which is a  $N \times M$  matrix.

For this operation we need to know how the matrix is represented in memory. Which will also be of interest for some of the MVM implementations in Section 4.2.

Note that it might be possible to load a  $V \times V$  matrix into the vector registers and then perform a transposition. But this requires that the vector processor has at least  $V$  vector registers. And on top of that it requires a specialized instruction to perform the transposition.

**Instruction:** Transpose(AVR:Array [0..V-1] of Vector Register)  
**Pre:**  $(\exists C :: (AVR[0] \ AVR[1] \ \dots \ AVR[V-1]) = C)$   
**Post:**  $(AVR[0] \ AVR[1] \ \dots \ AVR[V-1]) = C^T$

In most cases however, it will be too expensive to implement this instruction. Only when the vector processor has to transpose  $V \times V$  matrices often will it pay off.

So, if matrix  $C$  is in memory we assume that element  $c_{i,j}$  (with  $0 \leq i < M$  and  $0 \leq j < N$ ) has the memory location  $iN + j$  relative to some base address  $b$ . The objective of the transposition implementation is to move that same element to location  $jM + i$  relative to some base address  $t$ .

This can be solved by allowing vectors to be loaded from memory with a so-called "stride". Instead of reading the vector elements from consecutive memory locations the vector processor can load vector elements from memory locations with an equal spacing (stride).

**Instruction:** LoadWithStride(VR:Vector Register, Base, Stride, Nr: Integer)  
**Pre:** *True*  
**Post:**  $(\forall i : 0 \leq i < Nr : VR[i] = Memory[Base + i \times Stride])$

**Instruction:** Write(VR:Vector Register, Base, Nr: Integer)  
**Pre:** *True*  
**Post:**  $(\forall i : 0 \leq i < Nr : Memory[Base + i] = VR[i])$

The pseudo code when  $M \leq V$  then becomes:

```
n:=0;
do (n != N) ->
  LoadWithStride(v, b, N, M);
  Write(v, t, M);
  n:=n+1;
od
```

We can write a similar program when  $N \leq V$  and we have the Read and Write-WithStride instructions which are symmetric to the Write and LoadWithStride instructions.

When both  $N > V$  and  $M > V$ , we will have to divide the matrix into sub-matrices. The easiest thing to do, is to divide it into square sub-matrices, so that we can transpose every sub-matrix and place it at the correct memory location to form the transpose of the large matrix.

## 4.2 Matrix-Vector multiplication

Matrix-vector multiplication (MVM) is the calculation of:

$$r = Ca$$

Where  $C$  is a  $M \times N$  matrix and  $a$  is a vector of length  $N$  and  $r$  a vector of length  $M$ . Which comes down to:

$$r[i] = \sum_{j=0}^{N-1} c_{i,j}a[j] \text{ for } 0 \leq i < M$$

For simplicity, we will assume that  $N = M = V$  for now, and look at the other cases later on.

Now there are two things we can do. We can calculate each element of the result vector  $r$  one at a time. Or we can try to calculate all the elements of the result vector simultaneously.

Calculating one element of the result vector consists of element-wise multiplication of two vectors ( $c_i$  and  $a$  for result  $r[i]$ ), followed by adding the results of these multiplications all up to form the final result.

However, adding the elements up takes  $O(V)$  operations. And with  $V$  results to calculate, this comes down to  $O(V^2)$  operations to calculate the entire MVM calculation.

The MVM calculation only requires  $V^2$  multiplications. So with a vector processor that can handle  $V$  operations in parallel, we would assume that is possible to calculate it in  $O(\frac{V^2}{V}) = O(V)$  time.

This is possible, if we include an instruction which can add the vector elements together in one clock cycle, but that would go against our rule not to add instructions whose execution time depends on  $V$ .

So, it seems to be a better idea to calculate all the elements of the result vector simultaneously. We can do this in a loop that iterates  $V$  times. During each iteration we add a product of an element of  $C$  and an element of  $a$  to each of the  $V$  result vector's elements. The choice of our elements of  $C$  and  $a$  influences the other instructions that we will need.

Note however, that the operation during each iteration of the loop is a Multiply-Accumulate (MAC) operation. So adding a MAC operation to the instruction set might be a good idea.

**Instruction:** MAC(VR1, VR2, VR3:Vector Register)  
**Pre:**  $(\exists vr1 :: vr1 = VR1)$   
**Post:**  $(\forall i : 0 \leq i < V : VR1[i] = vr1[i] + VR2[i] \times VR3[i])$

### 4.2.1 Vectorize

The most straightforward way of accomplishing the MVM calculation would be to take the loop invariant:

$$r[i] = \sum_{j=0}^{n-1} c_{i,j}a[j] \text{ for } 0 \leq i < V$$

And let  $n$  run from 0 to  $N$ . Then, in each iteration of the loop, the operation we have to perform is  $r[i] := r[i] + c_{i,n}a[n]$  for every  $0 \leq i < V$ .

This is no problem, we can use the MAC operation on two vectors. One of the vectors has to contain a column of  $C$ , and the other one has to contain a copy of the element  $a[n]$  in each of its elements.

For the matrix, we can assume that it is in memory in some useful format, such that we can read a single column into a vector register with a single read operation.

Copying the element  $a[n]$  to each element of a vector register would take  $V$  operations, unless we introduce a new instruction called vectorize.

**Instruction:** Vectorize(VR:Vector Register, R:Register)

**Pre:** *True*

**Post:**  $(\forall i : 0 \leq i < V : VR[i] = R)$

This solution results in an algorithm that takes  $O(V)$  clock cycles:

```
n:=0
r:=0;
do (n != V) ->
  c:=Column n of C;    // Load a column of matrix C from memory
  t:=Element n of a;   // Load an element of vector a from memory
                      // or copy it from a vector register

  Vectorize(b,t);
  MAC(r,c,b);
  n:=n+1;
od
```

This is not a bad solution, but the vectorize operation requires the broadcasting of a single value to fill all vector elements. It might be possible to find a way without broadcasting.

#### 4.2.2 Rotate

We can avoid broadcasting (as was needed by the vectorize operation) by making sure that each element of vector  $a$  is needed during each iteration of the loop.

If we do this, then we need to choose the correct elements of matrix  $C$  to multiply with. The most obvious choice for the first iteration is the diagonal of  $C$ . In the second iteration we can take the sub-diagonal of  $C$  plus one other element (the upper right one), which, in effect, is the diagonal of the "rotated" matrix  $C$ . With rotated we mean that the rows have been rotated around (we will leave the direction open for now).

This can work, but depending on which way we rotate we will also have to rotate either the vector register containing the preliminary results, or the vector register containing the vector  $a$ .

For this rotating we introduce a new instruction:

**Instruction:** RotateLeft(VR:Vector Register)

**Pre:**  $(\exists vr :: vr = VR)$

**Post:**  $(\forall i : 0 \leq i < V - 1 : VR[i] = vr[i + 1]) \wedge VR[V - 1] = vr[0]$

If we assume that the required diagonals can be loaded from memory, then the pseudo-code of the algorithm becomes:

```
a:=Vector a;          // Load vector a from memory
n:=0;
r:=0;
do (n != V) ->
  c:=Diagonal of C;   // Load the correct values for c
  MAC(r,m,b);
  RotateLeft(r);
```

```

    n:=n+1;
od

```

Again, this algorithm requires  $O(V)$  instructions to complete, but now it contains less instructions inside the loop. Therefore it will run faster than the program discussed in Section 4.2.1.

For a more detailed description of this implementation, see the section "The Records" on the site <http://www.win.tue.nl/~mhorst>.

The main drawback of this implementation is that the matrix  $C$  needs to be placed in memory in such a way that we can easily access the required elements. However, if the matrix is in memory as described in Section 4.1 then we have a problem. In that case it would take a LoadWithStride instruction that also allows applies a modulus to the address calculation. Since we feel that such an instruction would not be usefull to other algorithms, we will not introduce it. Instead this implementation can only be used when  $C$  is in memory in some more suitable format.

### 4.2.3 Shift

The rotate operation is nice, but it requires the copying of a value from one end of the vector to the other end. This implicates some "long" wires in the processor. Would it be possible to get rid of this? In other words, can we make do with the shift operations only?

**Instruction:** ShiftRight(VR:Vector Register)  
**Pre:**  $(\exists vr :: vr = VR)$   
**Post:**  $(\forall i : 0 \leq i < V - 1 : VR[i + 1] = vr[i]) \wedge VR[0] = 0$

**Instruction:** ShiftLeft(VR:Vector Register)  
**Pre:**  $(\exists vr :: vr = VR)$   
**Post:**  $(\forall i : 0 \leq i < V - 1 : VR[i] = vr[i + 1]) \wedge VR[V - 1] = 0$

Note that the rotation operation can be implemented with a shift operation when it is possible to access the first and last element of a vector. In other words RotateLeft(VR); is the same as:

```

t:=VR[0];
ShiftLeft(VR);
VR[V-1]:=t;

```

And likewise, we can implement a shift operation with the help of a rotate operation.

To implement the MVM calculation with the shift operation, we will need to take the diagonals of  $C$ , but without the rotation we can only take elements which are on the same (sub)diagonal. It is not possible to "wrap around" and take exactly  $V$  elements. This leads us to the psuedo-code:

```

a:=Vector a;           // Load vector a from memory
n:=0;
r:=0;

```

```

do (n != V-1) ->
  c:=Diagonal of C; // Load the correct values for c
                    // padded with zeroes

  MAC(r,m,b);
  ShiftRight(r);
  n:=n+1;
od;
n:=0;
do (n != V) ->
  c:=Diagonal of C; // Load the correct values for c
                    // padded with zeroes

  MAC(r,m,b);
  ShiftLeft(a);
  n:=n+1;
od

```

This algorithm requires  $2V - 1$  iterations and thus more clock cycles than the previous solutions. But the LoadWithStride instruction allows us to load the correct elements of  $C$  when it is in memory in the format discussed in Section 4.1.

#### 4.2.4 Shuffle

Each element of the result vector is calculated with the help of  $V$  MAC operations. But it does not really matter, because of the associativity of addition, which of the MAC operations is performed first. The solutions in Section 4.2.1 and 4.2.2 perform exactly the same MAC operations, with the same elements when supplied with the same  $C$  and  $a$ , only the order in which they are performed differ.

So what if we could make that order arbitrary? As long as we can select one (previously unselected) element from each row of  $C$  in each iteration, we can write an algorithm which requires only  $V$  iterations and therefore only  $O(V)$  clock cycles.

**Instruction:** Shuffle(VR1, VR2, Pattern:Vector Register)

**Pre:** *True*

**Post:**  $(\forall i : (0 \leq i < V) \wedge (0 \leq \text{Pattern}[i] < V) : \text{VR1}[i] = \text{VR2}[\text{Pattern}[i]])$

But so far, this instruction has no advantages over the vectorize and rotate. (Note that with the correct pattern vector, the shuffle operation can mimic the rotate operation, and mimicing the shift operation is not that hard either). In fact, adding this instruction to the vector processor's instruction set will require a lot of hardware.

#### 4.2.5 Matrices of different sizes

When matrix  $C$  is not  $V \times V$ , but some other size, we can still use the algorithms described previously. When  $C$  is larger, we can divided into submatrices of

$V \times V$ . If  $C$  is smaller, or if the subdivision of  $C$  left us with smaller submatrices, we can fill them up with zeroes until they are large enough.

This method works, but it is a bit of a waste of processor resources to multiply with 0. Some of these multiplications might be avoided as explained in the section about special matrices (Section 4.2.6).

It might also be the case that the MVM calculation has to be performed multiple times. In such a case it might be possible to extend  $C$  with copies of itself until it is a  $lcm(V, M) \times N$  matrix (where  $lcm(a, b)$  denotes the least common multiple of  $a$  and  $b$ ). After this extension the matrix can be divided into submatrices of size  $V \times V$ .

However, we have to note that the number of operations increases linearly with the number of submatrices. And that the number of submatrices increases quadratically with the size of the matrix. An  $M \times N$  matrix requires, in general  $\frac{M}{V} \frac{N}{V} = \frac{NM}{V^2}$  submatrices.

#### 4.2.6 Special Matrices

Some matrices have special properties, that make it easier to perform a MVM calculation.

If some of the columns of  $C$  contain nothing but zeroes, then some of the iterations of the algorithm in Section 4.2.1 with the vectorize operation can be eliminated. The same holds when the shuffle operation is used. But the algorithms with the rotate and shift operations cannot be optimized in this case.

For banded (or diagonal) matrices the algorithm with the vectorize operation is the only one that cannot be optimized. The others only take as much iterations as the band is wide. For the rotate and shuffle-based algorithms the matrix does not even need to be banded exactly. The rotate-based algorithm can also handle matrices with a band and some non-zero components in the lower left and upper right corner.

The shuffle-based algorithm can handle any matrix efficiently, as long as the number of non-zero elements on each row is the same.

This brings us to triangular matrices. These matrices have a different number of non-zero elements on each row and therefore none of the algorithms are able to calculate the MVM efficiently (without multiplications with zero). The shift-based algorithm becomes more efficient (taking only  $V$  iterations), but this is not more efficient than any of the other implementations.

#### 4.2.7 Conclusion

First of all, the MAC instruction is used in every implementation and thus it clearly has to be added to the instruction set.

Adding the vectorize operation to the instruction set seems to be the best solution when we have no other information about the matrix  $C$ . It also performs rather well when  $M$  ( $C$  is a  $N \times M$  matrix) is not equal to  $V$ , or when  $C$  contains columns with only zeroes.

When matrix  $C$  is triangular the shift instruction might be a good choice. The other instruction would allow the MVM to be implemented just as efficiently, but they might require more hardware to implement. The rotate and shuffle instructions would certainly require more hardware. But whether the shift or vectorize instruction requires more hardware is not clear.

The rotate instruction becomes interesting for banded matrices or nearly-banded matrices (i.e. there are some non-zero elements in its lower left and upper right corner).

For matrices  $C$  where every row has less than  $M$  non-zero elements the shuffle instruction is the best choice, provided that the zero elements do not line up to form columns (in which case the vectorize instruction is better).

Note that for all the implementation (except the one with vectorize) we assumed that it was easy to get some relatively arbitrary elements of matrix  $C$  easily into a vector register. This implies that the matrix  $C$  is either some kind of precalculated constant, so that it can be placed in memory in the required format. Or that the vector processor possesses some specialized load instructions to load the correct values into a vector register. For this last option see Section 4.1.

### 4.3 Matrix-Matrix multiplication

Matrix-matrix multiplication (MMM) is the calculation of:

$$R = CD$$

Where  $C$  is a  $M \times N$  matrix,  $D$  a  $N \times L$  matrix and  $R$  the resulting matrix of  $M \times L$ .

The easiest way to implement the MMM is to see it as  $L$  matrix-vector multiplications (MVM). Each column of  $R$  is the result of such a MVM with matrix  $C$  and a column of  $D$ .

If we assume that  $M = N = L = V$  then the MMM requires  $V^3$  multiplications. The MVM implementation on a  $V \times V$  matrix with a vector of size  $V$  can be done in  $O(V^2)$  time. So performing  $V$  MVM's takes  $O(V^3)$  clock cycles. So this method is quite efficient and we will not spend more effort looking for a better one.

When not all of  $M, N$  and  $L$  are equal to  $V$  we will have to use the methods described in Section 4.2 to handle these different sizes. We can also use the fact that  $R^T = (CD)^T = D^T C^T$  if this makes it possible to match the matrix sizes more efficiently to the vector processor's vector size  $V$ . Assuming, of course, that we are allowed to produce the transposed result matrix, or if we can quickly calculate the transpose of a matrix (see Section 4.1).

Note that it is possible to implement matrix multiplication more efficiently when we have large matrices. But we will assume that the matrices are relatively small right now.

## Chapter 5

# Number Representations

In this chapter we will concern ourselves with representing numbers on the vector processor. But, we will be mainly interested in representing large and complex numbers, not how the  $W$  bits of a vector element can represent negative or fractional numbers.

### 5.1 Large numbers

By large numbers we mean numbers which require more than  $W$  bits to represent. Such numbers are often used in cryptography calculations.

We have two choices on representing large numbers. First of all, we can try to maintain the vector processor's ability to perform  $V$  calculations simultaneously. In other words we use  $x$  vectors to represent  $V$  numbers that are represented by  $xW$  bits.

Another choice is to use an entire vector register to represent a single  $VW$  bit number.

#### 5.1.1 $V$ large numbers

This implementation is quite straightforward. It does not differ much from the implementation of handling large numbers on a general purpose processor. In his book Knuth [?] describes how this can be done.

Special instructions that make it possible to catch the carry are useful here. Adding two  $W$  bit numbers together produces a new  $W$  bit number and a carry. Thus, it should be possible to put the carries of  $V$  element-wise additions into a vector register. In a similar manner a multiplication of two  $W$  bit numbers produces a  $2W$  bit number, thus it should be possible to divide the result of an element-wise multiplication over two vector registers.

Note that these features are not really necessary, but they would make the implementation more efficient. Thus:

**Instruction:** AddAndCarry(VR1, C, VR2, VR3:Vector Register)  
**Pre:** *True*  
**Post:**  $(\forall i : 0 \leq i < V : \langle C[i], VR1[i] \rangle = VR2[i] + VR3[i]$

**Instruction:** MultiplyAndCarry(VR1a, VR1b, VR2, VR3:Vector Register)  
**Pre:** *True*  
**Post:**  $(\forall i : 0 \leq i < V : \langle VR1b[i], VR1a[i] \rangle = VR2[i]VR3[i]$

Where  $\langle a, b \rangle$  denotes the number represented by concatenating the bits of  $a$  and  $b$ .

The only drawback to the vector processor is that it is not possible to have multiple large numbers of different length (in bits). They all have to have the same length. Furthermore if we are multiplying or adding  $V$  large numbers together we can only stop the algorithm when none of the  $V$  additions produces a carry.

### 5.1.2 A single large number

To represent large numbers using the  $V \times W$  bits of a vector register, we effectively have to implement the AddAndCarry and the MultiplyAndCarry operations for vector registers.

The AddAndCarry operation for numbers represented by large registers can be implemented with the help of the AddAndCarry operation presented in the previous section, the ShiftLeft operation and the And operation like this:

```
stop:=false;
carry:=0;
AddAndCarry(r, c, a, b);
do (not stop) ->
  carry:=carry or c[0];
  shiftleft(c);
  t:=( c = 0);
  And(stop, t);
  AddAndCarry(r, c, r, c);
od
```

Where we assume that the most significant vector element is at index 0.

Multiplication is a bit harder. We can model it as a matrix vector multiplication with a  $2V \times V$  matrix. But we have to keep in mind that we cannot use the MAC operation, since we have to worry about the carries. So, instead of MAC we have to use MultiplyAndCarry and AddAndCarry to calculate the results. This makes the matrix vector multiplication solution that uses the Vectorize operation the best choice (see Section 4.2.1).

## 5.2 Complex numbers

It is possible that an algorithm requires calculations with complex numbers, the Fourier Transform in Section 7.2 for example. A processor might have special

support for encoding complex numbers in the  $W$ -bit vector elements, but if this is not the case, another solution has to be found.

The most common representations of a complex number  $z$  are rectangular ( $z = a + bi$ ) and polar form ( $z = re^{-i\phi}$ ). The polar form is not well suited for calculations involving addition. Therefore we will assume that the rectangular form is being used. However, many of the considerations in this sections can be applied to complex numbers in polar form.

We can use the rectangular form ( $z = a + bi$ ) by storing the real ( $a$ ) and imaginary part ( $b$ ) of the different complex numbers in different vector elements. For now we will assume that the real and imaginary parts reside in different vectors.

So, a vector of  $V$  complex numbers  $c$  is actually represented by two vector registers, one containing the real ( $cr$ ) and the other containing the imaginary part ( $ci$ ). Adding two vectors of complex numbers together is easy, it can be accomplished by adding the real and imaginary parts of the two vector together respectively.

Multiplication is also quite straightforward. The real parts are multiplied together and the imaginary parts are multiplied together, and the results are subtracted from each other. This results in the real part of the resulting vector. For the imaginary part of the result vector we multiply the real and imaginary parts of the first and second vector together and vice versa. The results are added up to form the imaginary part of the result vector.

Now, instead of having separate vector registers for the components of the complex numbers, we could put them all into one single register. Aside from the fact that this allows us to only represent  $\frac{1}{2}V$  complex numbers this complicates multiplication a bit.

Addition has not been affected, we can just add two vector registers containing complex numbers together and obtain the result in one operation. But multiplication is another matter entirely, to implement it we have to know how the imaginary and real parts are distributed in the vector. The two most logical distributions would be: 1) to store the real components in the even vector elements and the imaginary components in the odd elements. Or 2) to store the real components in the first half of the register and the imaginary components in the other half. Other distributions are possible but these are the two most interesting ones.

To multiply, in effect, we have to divide the vector register over two registers (one for the real and one for the imaginary parts) and apply the same algorithm as before. But this would not lead to a very efficient implementation.

First of all, to calculate the real part of the new vector of complex numbers, we have to multiply the real part of one multiplicand with the real part of the other multiplicand and the imaginary part of the one with the imaginary part of the other. This can now be done with a single multiplication of the two vectors, without having to modify anything. However, the results of the multiplications have to be subtracted from each other to form the real part of the result vector. This is a bit problematic.

A similar problem occurs for calculating the imaginary parts of the result vector. The real part of one multiplicand has to be multiplied by the imaginary part of

the other multiplicand and vice versa. This can be done with a single multiplication operation, provided we use an operation that switches the imaginary and real parts of one of the multiplicands first. For option 1) we need the Shuffle operation to switch them around. For option 2) we can use a RotateLeftOver to rotate the vector over  $\frac{1}{2}V$  positions. But the results of the multiplications have to be added together to form the imaginary part of the result vector, which forms a similar problem as the one for the real part of the result vector.

The solution is quite simple, but harder to express in words. Therefore we present the following pseudo-code:

```
r1:=a * b;
t:=b;RotateLeftOver(t, 1/2 V);
r2:=a * t;
Shuffle(t, r1, pattern1);
t:= -t;
Shuffle(t, r2, pattern2);
Shuffle(r1, r2, pattern3);
r:= r1 + t;
```

For option 1) we have to replace the `t:=b;RotateLeftOver...` line with a Shuffle operation, of course.

In any case, `pattern1` is designed to take the results of the imaginary-imaginary multiplications and place them at the same position in vector `t` as the results of the real-real multiplications in `r1`. `Pattern2` is designed to take the results of the real-imaginary multiplications and place them at the same position in vector `t` as the results of the imaginary-real multiplications in `r2` (note that this does not overwrite any value put in `t` by our previous shuffle operation). The last pattern, `pattern3`, is used to copy the imaginary-real multiplications in `r2` to vector `r1`, note this only overwrites the results of the imaginary-imaginary multiplications in `r1`, but that does not matter, since we already stored them in `t`. And after all this shuffling around, the only thing necessary to calculate the final result, is the addition of the vectors `r1` and `t`.

## Chapter 6

# Filtering

Many filter algorithms make use of Matrix vector multiplications (see Section 4.2).

### 6.1 FIR

MVM.

### 6.2 Scaler

See the scaler document on [www.win.tue.nl/~mhorst](http://www.win.tue.nl/~mhorst).

### 6.3 IIR

Block-state with states representing the output. Block-state with A, B, C and D.

Incremental Block-state ?

Clustered Look-ahead with incremental output computation (unstable).

## Chapter 7

# Transforms

### 7.1 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) and its inverse can be implemented using a matrix vector multiplication as presented in Section 4.2.

The 2-D DCT can be implemented using matrix-matrix multiplication and matrix transposition (Sections 4.3 and 4.1), as explained in [?].

### 7.2 Fourier Transform

The fourier transform is usually calculated with the so-called fast fourier transform (FFT) implementation. If the vector processor is going to be used in a signal processing environment, special FFT functionality may come in handy. However, we will assume no such functionality exists and attempt to implement an  $N$ -point DFT on the vector processor.

We assume that  $N$  is a power of two. The calculation of the FFT happens in  $\log_2 N$  stages. If we denote  $S_{i,j}$  as output  $i$  of stage  $j$  with  $0 \leq i < N$  and  $0 \leq j \leq M$  (with  $M = \log_2 N$ ) then the FFT calculation of a sequence  $x[i]$  can be characterized with the following formulas:

$$S_{i,0} = x[\text{bitrev}(i)]$$

where the function  $\text{bitrev}(i)$  returns the number obtained by reversing the order of the  $M$  bits representing  $i$ .

$$S_{i,j} = \begin{cases} S_{i,j-1} + W_{2^j}^{-i} S_{i+2^{j-1},j-1} & \text{for } 0 \leq i \bmod 2^j < 2^{j-1} \\ S_{i-2^{j-1},j-1} - W_{2^j}^{-(i-2^{j-1})} S_{i,j-1} & \text{for } 2^{j-1} \leq i \bmod 2^j < 2^j \end{cases}$$

for  $1 \leq j \leq M$  where  $W_n = e^{j\frac{2\pi}{n}}$ .

With this definition the outputs of stage  $M$  form the DFT of the sequence  $x[i]$  ( $0 \leq i < N$ ). I.e.  $X[i] = \sum_{n=0}^{N-1} x[n]W_N^{-in}$  is the DFT of the sequence  $x[i]$  and  $S_{i,M} = X[i]$ .

We want to implement this calculation on the vector processor. The first step in implementing the DFT, is to divide the different  $S_{i,j}$ 's over one or more vectors. It might be tempting to just assign  $S_{i,j}$  to some vector element  $s[i]$ , but this would make the calculation of  $S_{i,j+1}$  a bit hard.

A better solution is to divide the elements  $S_{i,j}$  for a given  $j$  over two vector registers. We will call these registers  $s1$  and  $s2$  and they adhere to the following invariant:

$$\begin{aligned} s1[i] &= S_{(i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j), j} && \text{for } 0 \leq i < \frac{1}{2}N \\ s2[i] &= S_{2^{j+1} + (i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j), j} && \text{for } 0 \leq i < \frac{1}{2}N \end{aligned}$$

What we have done is dividing the  $S_{i,j}$ 's into two groups. We have done this in such a way that for the calculation of the next stage only one of the groups ( $s2$ ) has to be multiplied by some power of  $W_n$ . So, it then becomes quite easy to establish:

$$\begin{aligned} r1[i] &= S_{(i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j), j+1} && \text{for } 0 \leq i < \frac{1}{2}N \\ r2[i] &= S_{2^{j+1} + (i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j), j+1} && \text{for } 0 \leq i < \frac{1}{2}N \end{aligned}$$

The shuffle instruction introduced in Section 4.2.4 can then be used to maintain the invariant under  $j := j + 1$ .

**Instruction:** Shuffle(VR1, VR2, Pattern: Vector Register)

**Pre:** *True*

**Post:**  $(\forall i : (0 \leq i < V) \wedge (0 \leq \text{Pattern}[i] < V) : VR1[i] = VR2[\text{Pattern}[i]])$

With this invariant we can write the following pseudo code:

```

j:=0;
Initialize;           // Initialize s1 and s2
do j != M ->
  w:=Stage j values for W; // Load constants from memory
  r2:=s2 * w;
  r1:=s1 + r2;
  r2:=s1 - r2;
  p1:=Stage j pattern 1; // Load pattern from memory
  p2:=Stage j pattern 2; // Load pattern from memory
  Shuffle(s1, r1, p1);
  Shuffle(s1, r2, p2);
  Shuffle(s2, r2, p1);
  Shuffle(s2, r1, p2);
  j:=j + 1;
od

```

The patterns for the different stages are quite simple:

$$Pattern1[i] = \begin{cases} (i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j) & \text{for } 0 \leq i \operatorname{mod} 2^{j+1} < 2^j \\ -1 & \text{for } 2^j \leq i \operatorname{mod} 2^{j+1} < 2^{j+1} \end{cases}$$

$$Pattern2[i] = \begin{cases} -1 & \text{for } 0 \leq i \operatorname{mod} 2^{j+1} < 2^j \\ -2^{j+1} + (i \operatorname{div} 2^j)2^{j+1} + (i \operatorname{mod} 2^j) & \text{for } 2^j \leq i \operatorname{mod} 2^{j+1} < 2^{j+1} \end{cases}$$

Note that the symmetry in the patterns allows us to use just one pattern if we have a rotate operation which allows a rotation over an arbitrary number of elements.

**Instruction:** RotateRightOver(VR:Vector Register, R:Register)  
**Pre:**  $(\exists vr :: vr = VR)$   
**Post:**  $(\forall i : 0 \leq i < V - 1 : VR[i] = vr[(i - R) \bmod V])$

```

j:=0;
Initialize;                // Initialize s1 and s2
do j != M ->
  w:=Stage j values for W; // Load constants from memory
  r2:=s2 * w;
  r1:=s1 + r2;
  r2:=s1 - r2;
  p1:=Stage j pattern 1;  // Load pattern from memory
  Shuffle(s1, r1, p1);
  Shuffle(s2, r2, p1);
  RotateRightOver(p1, 2^j)
  Shuffle(s1, r2, p1);
  Shuffle(s2, r1, p1);
  j:=j + 1;
od

```

This saves a memory access, but requires that  $V = N$ .

So, we have shown how to implement a FFT algorithm on the vector processor. At this point we can efficiently code a  $2V$ -point DFT (for larger vector sizes we will need to use more registers). We assumed that we can represent a complex number in each vector element. This is very likely impossible, but we can use the techniques presented in Section 5.2 to represent complex numbers.

FIXME: I have to check all this, there are no formal proofs and I might have made a mistake.

## Chapter 8

# Finite State Machines

Finite state machines (FSM) can be used to model many algorithms. Most notably is regular expression matching.

A FSM consists of a finite number of states (hence the name) and a function that describes states transitions depending on the current state and an input. The inputs belong to an alphabet, and the machine also has a start state. More formally a finite state machine may be defined by:

$$(S, \Sigma, T, s)$$

Where  $S$  denotes the collection of states,  $\Sigma$  the input alphabet,  $q$  denotes the start state ( $s \in S$ ) and  $T$  the transition function. For deterministic FSMs the transition function has the type  $T : S \times \Sigma \rightarrow S$ . For non-deterministic transition functions the type is  $T : S \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathbb{P}(S)$ .

However, this does not allow the machine to accomplish much. FSM which are acceptors are described by  $(S, \Sigma, T, s, F)$ , where  $F \subseteq S$  denotes the final or accepting states of the FSM. This type of FSM accepts an input string when it reaches a state in  $F$ . When it does not the string is not accepted. This property can be used to check whether an input string satisfies a regular expression for example.

It is also possible to have the finite state machine generate some output. In this case the machine is described by  $(S, \Sigma, \Lambda, T, O, s)$ . Where  $\Lambda$  denotes the output alphabet and  $O$  and output function. When this function is of the type  $O : S \times \Sigma \rightarrow \Lambda$  the FSM is called a Mealy machine. And when the function is of type  $O : S \rightarrow \Lambda$  the FSM is called a Moore machine.

### 8.1 Deterministic

A deterministic FSM (DFSM) can be implemented by keeping track of the state and using the transition function (usually in the form of a look-up table) to find the next state.

We will assume a look-up table is used. If the transition function is actually a function which can be calculated other possible implementation methods might be available. For example the IIR filter; it can be modelled as a DFSM, but the implementations in Section 6.3 are more efficient.

For a Mealey machine the look-up table also provides an output. For Moore machine's and acceptors there is a separate look-up table providing either the corresponding output, or whether the state is an accepting state.

The only thing the implementation has to keep track of, is the state of the DFSM. Assuming it can be encoded into a single vector element, we can try to run  $V$  DFSM simultaneously.

Another option is to try and improve the speed of the DFSM, this can be done by transforming the DFSM into a DFSM with a larger alphabet ( $\Sigma \times \Sigma$  instead of  $\Sigma$ ), which allows it to accept multiple inputs (of the original DFSM) at the same time. However, this will also increase the size of the look-up table.

So, we will try to run several DFSMs simultaneously. These instances can be related in different ways: they can be totally independent, or share the same input stream, or share the same DFSMs.

Whatever is the case, the implementation on the vector processor faces a problem. The look-up table is in memory and thus requires memory accesses. And the memory address depends on the location of each instance's DFSM (they might share the table if all the DFSMs are the same), the current input (again, shared if the instances have the same input stream), and the state.

Aside from the possibility of requiring access to the same memory element, there is a large chance that the instances need to access very different memory locations. Simply loading a vector from adjacent memory locations, or the LoadWithStride instruction are possible. But different addresses for every vector element requires a lot of hardware. Never the less, it seems like the only option.

**Instruction:** LoadFrom(VR, Address:Vector Register)  
**Pre:** *True*  
**Post:**  $(\forall i : 0 \leq i < V : VR[i] = Memory[Address[i]])$

The look-up table is then like a  $|\Sigma| \times |S|$  matrix  $C$ . If we assume that element  $c_{i,j}$  of  $C$  is at memory location  $i|\Sigma| + j + b$  for some base address  $b$ , then we can write the pseudo-code for a state transition as:

```
address := base + states * sizeof(Sigma) + inputs;
LoadFrom(states, address);
```

Where **base** denotes the base address for the look-up tables, **states** the current states and **inputs** the current inputs of the DFSMs.

## 8.2 Non-Deterministic

A non-deterministic FSM (NFSM) has a transition function that produces a set of states. In effect the NFSM can reside in multiple states at the same time.

Also, a transition can take place without an input symbol. The  $\epsilon$  symbol is used for this. So, in between inputs one or more state transitions can take place.

This seems all very complicated, but for every NFSM there is an equivalent DFSM. So we could just convert the NFSM to a DFSM and implement it in the same way. However, the number of states of the equivalent DFSM is exponential compared to that of the NFSM. So a method of simulating an NFSM can come in handy.

Generally, a NFSM is an acceptor. Technically it is possible to have a non-deterministic Moore or Mealey machine, but they are not often used. Therefore we will limit ourselves to acceptors.

To simulate a NFSM we can use backtracking, we just try one of the possible states in the transition function and see if we end up in an accepting state, if not we backtrack to try another possible state. But this is hard to implement on a vector processor, the different NFSM instances may have to backtrack at different points in time.

Another possibility is to keep track of all possible states. We simulate the machine and if an accepting state is in the set of possible states, then the input is accepted. If the set of possible states becomes empty then the input is rejected.

We can model the a set of  $V$  states by assigning a value to each element of a vector register, indicating whether the state is in the set (true) or not (false).

Then (if we forget about  $\epsilon$  for a moment), we can use a matrix vector multiplication algorithm to model a state transition. We would use a different matrix for every input and use logical and and or operations instead of multiplication and accumulation.

But we can also envision the states of the NFSM containing tokens if the states are in the set of possible states. A transition because of an input then results in the movement of the tokens, a bit like a Petri-net. We can capture this movement of these tokens with the Shuffle operation with a specific pattern for each input.

However, there is one problem, if two tokens have to be moved to the same state they result in a single token inside that state (unlike a Petri-net). The shuffle operation does not allow us to give a vector element a value that is the combination of different source vector elements. But we can accomplish this by multiple shuffle operations on the vector representing the set of states. After these operations have been completen we combine the results together with element-wise logical or operations and we have the new state.

So, basically, all that we need is a look-up table containing entries for each input. The table contains a number of shuffle operations in each entry and we can apply them to the current set-of-states vector to obtain a new one. The remaining problem is  $\epsilon$ .

There are several ways to solve this. One is to also use a shuffle pattern for  $\epsilon$  based transitions and apply them until nothing changes. However, it is also possible to redesign the transition function in such a way that no  $\epsilon$  based transitions occur. We will assume this last solution for our pseudo-code:

```

stateset:=[false, false, ..., false];
stateset[startstate]:=true;
stop:=false;
do (not stop) ->
  input:="get input";           // Retrieve input
  nr:=Memory[nrbase+input];     // Look-up table
  address:=Memory[addrbase+input]; // Look-up table
  nextset:=[false, false, ..., false];

  do (nr != 0) ->               // Apply shuffles
    Load(Pattern, address);
    Shuffle(temp, stateset, pattern);
    nextset:=nextset or temp;
    address := address + V;
    nr := nr - 1;
  od

  stateset:=nextset;

  temp:= (stateset = 0);
  And(stop, temp);
  stop:= stop or stateset[acceptstate];
od;

if (stateset[acceptstate]) -> "input accepted"
[] (temp)                   -> "input rejected"
fi

```

Note that we assumed that there was a single accepting state. But for a given NFSM it is easy to obtain one with only one accepting state, just place  $\epsilon$  transitions to a new accepting state from the previously accepting states. Another possibility is to compare the `stateset` vector with one vector containing the set of accepting states and use the Or operation.

Of course, this solution only works when the NFSM has  $V$  or less states. It is possible to simulate a NFSM with more states in this manner, but the number of operations required to simulate it grow quadratically. For example, if two vectors are needed to represent the state we need to perform shuffles for state transitions between the states in the same register (2), but also shuffles for state transitions between the registers (2). For a total of 4 sets of shuffle operations.

On a different note, we can use the  $W$  bits of a vector element for much more things than just representing true or false. Each bit allows us to represent a token. So, we could simulate several NFSM with the same transition function and the same input at the same time. This can be usefull for when we the NFSM is used to search for a substring in a larger input. Each bit  $W$  represents a different starting point of the search.

This means we can either search only for substrings in strings of length  $W$ , or we can search for substrings in a larger string, as long as the substring we are looking for cannot be longer than  $W$ . Furthermore, we can use multiple vectors, for example  $x$ , to represent  $xW$  different tokens, allowing us to search

for substrings of length  $xW$ . The nice thing is, that the number of operations we have to perform grows linearly with  $x$ .

FIXME: Maybe work this out a bit further?

DRAFT

## Chapter 9

# Other Algorithms

### 9.1 Rabin-Miller

The Rabin-Miller test is used to test if a number is composite, i.e. if it can be factored into the product of two or more primes. The result of the test is either that the number is composite, or that it is unknown whether the number is composite or prime.

Despite this, it is often used to test whether a number is prime. The Rabin-Miller test has the property that it can be repeated with a different parameter. By repeating it several times (with a different parameter) the probability that the test results in "unknown" while the number is composite drops. So, when the results remains "unknown" after enough repetitions we can say with a high certainty that the tested number is prime.

Rabin-Miller is often used in systems where a different, large, prime numbers are needed (such as cryptography). The system randomly generates a bit string and tests whether the generated number is prime using (among others) the Rabin-Miller test. In most cases this is faster than any other method of obtaining a large prime.

#### 9.1.1 Algorithm

The algorithm for the Rabin-Miller test is not at all complicated. First of all we need the number which the algorithm is going to test, we call it  $n$ . We need to have that  $n$  is odd (this is easily checked and should  $n$  be even then it is composite, unless  $n = 2$ ). Furthermore we have parameter  $a$  which  $1 < a < n-1$ . We can vary this parameter with different runs of the test.

Because  $n$  is odd we can find an odd  $m$  and a  $k > 0$  such that  $n - 1 = m2^k$ . Once these are found we generate a sequence starting with  $a^m \bmod n$ . Each number in the sequence is the square of its predecessor modulo  $n$ . The last element in the sequence is  $a^{n-1}$ . So we generate:  $[a^m, a^{2m}, a^{4m}, a^{8m}, \dots, a^{2^k m}]$  modulo  $n$ .

If we use  $*$  to denote a number that is not 1 nor  $-1$  then the sequence generated by the Rabin-Miller test has one of the following four forms:

1.  $[1, \dots, 1]$
2.  $[*, \dots, *, -1, 1, \dots, 1]$
3.  $[*, \dots, *]$
4.  $[*, \dots, *, 1, \dots, 1]$

The first two forms indicate that the result is "unknown", so  $n$  is said to pass the Rabin-Miller test for base  $a$ . The other two sequences are only generated by composite  $n$ , so in that case  $n$  is said to fail the Rabin-Miller test for base  $a$ .

We do not need to generate the entire sequence to determine the result of the test. Once the result of the test is known we can stop generating to sequence. So, if the first element of the sequence ( $a^m \bmod n$ ) is 1 then the result is "pass". If during the generation of the sequence we get a  $-1$  the result is pass, otherwise the result is failed.

### 9.1.2 Implementation

The Rabin-Miller test can be implemented on the vector processor quite efficiently. We can run  $V$  tests with different bases in parallel. As soon as one of the tests generates a failed, the test can return a "failed" result for all bases.

The numbers used in Rabin-Miller tests are often large, so we can use the methods described in 5.1 to represent large numbers. For the multiplication modulo  $n$  we can use montgomery multiplication [?].

We can do the element-wise comparing to  $-1$  and  $1$ , but we will also need some way to calculate the aggregation of these results. Therefore we introduce:

**Instruction:** And(R:Register, VR:Vector Register)

**Pre:** *True*

**Post:**  $R = (\forall i : 0 \leq i < V : VR[i])$

**Instruction:** Or(R:Register, VR:Vector Register)

**Pre:** *True*

**Post:**  $R = (\exists i : 0 \leq i < V : VR[i])$

However, these instructions need a tree of and and or gates respectively to be implemented on the vector processor. Which requires a calculation time of  $O(\log V)$ . And we want to keep such operations out of the processor's instruction set.

But with one of the  $W$  bits of a vector element representing "true" or "false" respectively, a simple tree of gates is all we need. This means that building a processor with vectors of size  $2V$ , these instructions will take one gate delay longer to accomplish. We feel that this does not seriously hamper the scalability of the vector processor.

So, the pseudo-code becomes:

```
ones:=[1,1,1,...,1];
mones:=[-1,-1,-1,...,-1];
passed:=[false, false, false, ..., false];
```

```

result:="failed";
i:=0;

b:=(bases ^ m) mod n;
passed:=(b = ones);
passed:= passed or (b = mones);
And(t1, passed);
if t1 -> i:=k; fi;

do (i != k) ->
  b:=(b * b) mod n;
  passed:= passed or (b = mones);
  failed:= (not passed) and (b = ones);
  And(t1, passed);
  Or(t2, failed);
  if (t1 or t2) -> i:=k-1; fi;
  i:=i+1;
od

if t1 -> result:="passed";

```

Where `bases` denotes a vector containing the  $V$  bases for which we do the test. Note that once  $n$  passes a single of the  $V$  parallel tests, we still continue calculating the sequence of that test. We could set up a new test with another based instead. But that would incur extra overhead. First of all, setting up the new test requires clock cycles, during which we cannot continue the other tests. Furthermore, we would need to keep track of how far we are into the sequence for each test, which would also incur extra overhead.

Another thing to note, is that the `And` and `Or` instructions are only used to see if the main loop can be terminated early. We could do without these instructions if we do not mind that the algorithm generates the entire sequence. This generates the following pseudo-code:

```

ones:=[1,1,1,...,1];
mones:=[-1,-1,-1,...,-1];
passed:=[false, false, false, ..., false];
result:="failed";
i:=0;

b:=(bases ^ m) mod n;
passed:=(b = ones);
passed:= passed or (b = mones);

do (i != k) ->
  b:=(b * b) mod n;
  passed:= passed or (b = mones);
  i:=i+1;
od

```

```
And(t1, passed);  
if t1 -> result:="passed";
```

Now the And instruction is used only once, so we could replace it with  $\log(V)$  and instructions on the vector elements, instead of using the special instruction. The advantage is that this saves a few instructions from the loop, the disadvantage is that the loop actually has to make  $k$  iterations. In general, this will not be worth our while since the number of saved instructions will be small compared to those implied by the statement  $b:=(b * b) \bmod n;$ .

## 9.2 CRC Checks

CRC checks with a  $V \times W + 1$  bits polynomial is easy and  $V$  simultaneous CRC's with (possibly different)  $W + 1$  bits polynomials is easy.

## 9.3 Searches

### 9.3.1 Binary Search

Possible in theory, but not much to gain here.

### 9.3.2 Linear Search

This can be done faster.

**Chapter 10**

**Conclusions**

DRAFT

## Bibliography

DRAFT