Mapping VLSI IIR architectures on vector processor programs

M.G. v.d. Horst

December 4, 2007

1 Introduction

In this article we will examine how VLSI architectures for IIR filters can be mapped to vector processor programs.

The order of the IIR filter is N, which is given for a certain filter. The size of a vector of the vector processor is V, which is given for a certain processor. There is one design variable P which indicates the size of the block of outputs.

FIXME: Expand the introduction.

2 Vector Processor

We will write pseudo code for the programs for our vector processor. These programs will contain at most a single processor instruction per line to ease the calculation of the number of clock cycles.

2.1 General description

For the loop operator we assume that the processor is capable of so-called zero-overhead looping. This means that if the number of loop iterations is known before entering the loop (which is the case in our programs), then it will only take one clock cycle to set up the loop, but no further costs are made during the iterations (except for the costs associated with the instructions in the iterations themselves of course).

For the other operations we assume that the processor is capable of executing both a single vector and a single scalar operation in parallel, no more. It is possible that both operations access the same (vector) registers. This is no problem, each operation uses the value of the register as it was before the parallel execution of both operations started. So even if a scalar register is updated in the scalar operation its initial value can still be used in the vector operation. The only restriction is that it is not possible for both operations to write to the same register or memory location. There are no other restrictions on this parallelism, so it is possible that both operations access memory.

The vector processor has a load-store architecture. We will denote the memory with arrays, each array indicating a different part of the memory. There are load and store operations to read and write from and to these memory arrays.

We will use assignment operations to indicate updates of the registers. These updates may involve only a single operation like addition, or abstraction, or multiplication, etc. Unless denoted otherwise these operations are performed element-wise on the elements of a vector.

2.2 Definitions

In this section we present some of the representations and the processor's built-in instructions and their pre- and postconditions.

To describe the relationship between the program variables and their mathematical counterparts we use the abstraction function. For example, we consider a vector register `v` to be an array of length V and the standard array-to-vector abstraction function tells us which vector it represents.

$$[\![\texttt{v}]\!]_i = \texttt{v[i]} \text{ for } 0 \leq i < V$$

(For simplicity we assume that the index of a vector runs from 0 to its length minus one.) In a similar manner the abstraction function for a scalar register `s` is $[\![\texttt{s}]\!] = \texttt{s}$. Of course these are rather trivial examples, but we will use more complicated abstraction functions later on. For example when we want to express how a matrix is stored in memory.

In the next few sections we will describe the operations that are available on the vector processor. Such a description will be made in the following way:

Operation:	How the operation appears in the program text
Parameters	The types of the operation's parameters
Type	Type of the operation (vector or scalar). We can only execute operations of different types in parallel.
Pre:	Precondition of the operation, often used to indicate the parameter's initial values
Post:	Postcondition of the operation. We use the convention that the value of a parameter which is not mentioned here does not change.

2.2.1 Load Store Operations

The memory of the vector processor is seen as an array. We can use the abstraction function to derive a vector that represents the memory, in the same way a vector register represents a vector:

$$[\![\texttt{Mem}]\!]_i = \texttt{Mem[i]}$$

Operation:	`v,addr:=Mem[addr..addr+V),addr+incr`
Parameters	`v:vector, Mem:memory, addr:address, incr:scalar`
Type	Vector
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge\ [\![\texttt{incr}]\!] = b \ \wedge\ [\![\texttt{Mem}]\!] = M$
Post:	$[\![\texttt{v}]\!]_i = M_{a+i}$ for $0 \leq i < V \ \wedge\ [\![\texttt{addr}]\!] = a + b$

Operation:	`v,addr:=Mem[addr,..,(V-1)(stride)+addr],addr+incr`
Parameters	`v:vector, Mem:memory, addr:address,` `stride, incr:scalar`
Type	Vector
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge \ [\![\texttt{incr}]\!] = b \ \wedge \ [\![\texttt{stride}]\!] = s \ \wedge \ [\![\texttt{Mem}]\!] = M$
Post:	$[\![\texttt{v}]\!]_i = M_{a+is}$ for $0 \leq i < V \ \wedge \ [\![\texttt{addr}]\!] = a + b$

Operation:	`Mem[addr..addr+V),addr:=v,addr+incr`
Parameters	`v:vector, Mem:memory, addr:address, incr:scalar`
Type	Vector
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge \ [\![\texttt{incr}]\!] = b \ \wedge \ [\![\texttt{v}]\!] = v$
Post:	$[\![\texttt{Mem}]\!]_{a+i} = v_i$ for $0 \leq i < V \ \wedge \ [\![\texttt{addr}]\!] = a + b$

Operation:	`Mem[addr,..,(V-1)(stride)+addr],addr:=v,addr+incr`
Parameters	`v:vector, Mem:memory, addr:address,` `stride, incr:scalar`
Type	Vector
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge \ [\![\texttt{incr}]\!] = b \ \wedge \ [\![\texttt{stride}]\!] = s \ \wedge \ [\![\texttt{v}]\!] = v$
Post:	$[\![\texttt{Mem}]\!]_{a+is} = v_i$ for $0 \leq i < V \ \wedge \ [\![\texttt{addr}]\!] = a + b$

Our vector processor allows non-aligned memory access. So the address in the load and store operations for a vector register does not have to be a multiple of V.

Operation:	`s,addr:=Mem[addr],addr+incr`
Parameters	`s:scalar, Mem:memory, addr:address, incr:scalar`
Type	Scalar
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge \ [\![\texttt{incr}]\!] = b \ \wedge \ [\![\texttt{Mem}]\!] = M$
Post:	$[\![\texttt{s}]\!] = M_a \ \wedge \ [\![\texttt{addr}]\!] = a + b$

Operation:	`Mem[addr],addr:=s,addr+incr`
Parameters	`s:scalar, Mem:memory, addr:address, incr:scalar`
Type	Scalar
Pre:	$[\![\texttt{addr}]\!] = a \ \wedge \ [\![\texttt{incr}]\!] = b \ \wedge \ [\![\texttt{s}]\!] = s$
Post:	$[\![\texttt{Mem}]\!]_a = s \ \wedge \ [\![\texttt{addr}]\!] = a + b$

2.2.2 Scalar Operations

The available scalar operations are standard arithmetic operations like addition, subtraction, multiplication, etc.

Note that the elements of a vector are also scalars, but we will not use them in scalar operations directly. There is also a load-store-like architecture for transferring values between scalar and vector registers. To accomplish this transfer we have the following operations:

Operation:	`s:=v[i]`
Parameters	`s:scalar, v:vector, i:scalar`
Type	Scalar
Pre:	$[\![\texttt{v}]\!] = v$
Post:	$[\![\texttt{s}]\!] = v_i$

Operation:	`v[i]:=s`
Parameters	`v:vector, s,i:scalar`
Type	Scalar
Pre:	$[\![\mathtt{s}]\!] = s$
Post:	$[\![\mathtt{v}]\!]_i = s$

2.2.3 Vector Operations

Operation:	`r:=MAC(r, v, s)`
Parameters	`r,v:vector, s:scalar`
Type	Vector
Pre:	$[\![\mathtt{r}]\!] = r \;\wedge\; [\![\mathtt{v}]\!] = v \;\wedge\; [\![\mathtt{s}]\!] = s$
Post:	$[\![\mathtt{r}]\!]_i = r_i + v_i * s$ for $0 \leq i < V$

Operation:	`r:=MAC(r, v, s)`
Parameters	`r,v1,v2:vector`
Type	Vector
Pre:	$[\![\mathtt{r}]\!] = r \;\wedge\; [\![\mathtt{v1}]\!] = v1 \;\wedge\; [\![\mathtt{v2}]\!] = v2$
Post:	$[\![\mathtt{r}]\!]_i = r_i + v1_i * v2_i$ for $0 \leq i < V$

Operation:	`v:=shiftleft(v)`
Parameters	`v:vector`
Type	Vector
Pre:	$[\![\mathtt{v}]\!] = v$
Post:	$[\![\mathtt{v}]\!]_i = v_{i+1}$ for $0 \leq i < V-1 \;\wedge\; [\![\mathtt{v}]\!]_{V-1} = 0$

Operation:	`v:=shiftright(v)`
Parameters	`v:vector`
Type	Vector
Pre:	$[\![\mathtt{v}]\!] = v$
Post:	$[\![\mathtt{v}]\!]_i = v_{i-1}$ for $1 \leq i < V \;\wedge\; [\![\mathtt{v}]\!]_0 = 0$

Operation:	`s:=Sum(v)`
Parameters	`s:scalar, v:vector`
Type	Vector
Pre:	$[\![\mathtt{v}]\!] = v$
Post:	$[\![\mathtt{s}]\!] = \sum_{i=0}^{V-1} v_i$

2.2.4 Parallel execution

The vector processor is capable executing a vector and a scalar operation in parallel. We denote this with:

```
(V || S);
```

Meaning that the vector operation `V` and the scalar operation `S` are executed simultaneously. The preconditions of both operations should hold before the parallel execution starts. And then the postcondition of both operations hold when the parallel execution ends.

Any two vector and scaler operations may be executed in parallel. The only exception is when the postconditions conflict with each other. This can happen if both operations write to the same register for example. Even though the scalar

operation can never write to an entire vector register (it is a scalar operation after all), it could write to one element of a vector register. If this register is also written to by the vector operation then we have a conflict. The processor has no mechanism to resolve these conflicts, so we should avoid them all together.

2.2.5 Looping

To repeat a group of instructions `A` we use the following construction:

```
do X times →
  A
od
```

This will have the effect that `A` is executed exactly `N` times. The instructions described so far all cost one clock cycle to perform and this one is no exception. It takes one clock cycle to set up this loop, but after that there is no overhead for the loop. So, it just takes `X` times the number of clock cycles it takes to execute `A` plus one clock cycle for setting up the loop to execute this operation.

3 Architectures

There are several VLSI architectures which can be used to implement an IIR filter. However, we are looking for programs that scale well in P; the number of clock cycles per block of outputs should remain constant, independent of P. And at the same time we want a linear relationship between P and V, so that there is a linear relationship between the throughput (P divided by the number of clock cycles) and the "amount" of hardware (V).

Another possibility is to have programs that scale well in N; the number of clock cycles per output should remain constant, independent of N. Which would require some linear relationship between N and V. But these implementations are beyond the scope of this paper.

This section contains an overview of several VLSI architectures. Two of them turn out to be suited for our vector programs. They also (partly) share a common architecture, which we will examine.

3.1 Overview

A good overview of several VLSI architectures for IIR filters was made by Parhi and Messerschmitt [**?**, **?**]. Table 1 shows an overview of the number of multiplications needed per block for each implementation (we assumed block diagonal state update matrices for the block-state architectures).

The implementation of the VLSI architectures as vector programs is effectively nothing more than simulating the VLSI circuit with a vector processor. The same number of multiplications have to be performed, so we can estimate the number of required vector operations by dividing the number of multiplications of a VLSI implementation by V.

Architecture	# Multiplications
Block state	$2NP + 2N + \frac{P(P+1)}{2}$
Incremental block state	$2N(P + P \text{ div } I) + \frac{P(I+1)}{2} +$ $\frac{P \bmod I(P+P \bmod I)}{2} + NI(P \text{ div } I - 1)$
Clustered look-ahead (with Incr. Outp. Comp.)	$(2N + \frac{P+1}{2})P$ if $N < P$ or, $P(3N+1) - \frac{N(N+1)}{2}$ if $N \geq P$
Scattered look-ahead	$PN(\log_2 P + 2) + P$

Table 1: Number of multiplications

To obtain an implementation in which the number of clock cycles, and thus the number of vector operations, does not depend on P, we will have to have a linear relationship between V and P. Thus $P = \alpha V$.

If we divide the number of multiplications of the block state implementation by P we still retain a factor P in the estimated number of vector operations.

The scattered look-ahead architecture is better. The estimated number of vector operations contains a factor $\log_2 P$. This is better, but not yet as good as possible.

The incremental block-state and clustered look-ahead with incremental output computation architectures both require an amount of multipliers that is linear in P. So we end up with a constant number of estimated vector operations per block of outputs. And that is what we want; doubling the block size requires a vector processor with double the vector length.

However, this is only a first, rough, estimation. We still need to find some way of actually implementing these architectures on a vector processor. This is the subject of our next section, where we will look at the problems in implementing the incremental block state and clustered look-ahead architectures.

3.2 Similarities

Both the architectures are quite similar; there is a basic piece of processing required followed by a "tail" of components.

This so-called tail causes difficulties because the results of one of the tail-components is fed into the next tail-component. Furthermore the length of this tail depends on P. But we should have a way of implementing it without using a number of vector operations that depend on P. And because one components output is needed for the next one we cannot divide the components easily over some vector operations.

More formally; we wish to calculate some results R for each component r of the tail of length T:

$$R(qT) = f(B(qT), A(qT))$$
$$R(qT+i) = f(R(qT+i-1), A(qT+i)) \text{ for } 1 \leq i < T$$

The $A(qT+i)$ denote inputs needed by the tail component, these inputs are readily available and do not depend on the calculation of a previous component in the tail. The calculation is represented by the function f, which takes

input from the previous component and from somewhere else to calculate the result. The $B(qT)$ denotes the output of the "previous" component of the first component of the tail, i.e. the output provided by some component of the implementation which is not part of the tail itself. The index q indicates the output block we are currently calculating.

As you may note it is very hard to write down the calculation of $R(qT+i)$ $(0 \leq i < T)$ for a single index q with a constant number of operations, independent of r (at least without knowing anything about the function f).

Now this is what we do to solve this problem: instead of calculating all R's in a consecutive order we leave some holes between them. In other words, instead of some invariant $[\![\mathtt{r}]\!]_i = R(qT + i)$ we choose:

$$[\![\mathtt{r}]\!]_i = R((q-i)T + i)$$

We choose this invariant because all we have to do to maintain it under `q:=q+1`, is to establish:

$$[\![\mathtt{r}]\!]_i = f(R((q+1-i)T + i - 1), A((q+1-i)T + i))$$

Which can be done easily with:

$$r[i] := f(r[i-1], A((q+1-i)T + i))$$

The exact implementation depends on the function f of course. But we can see that the dependency problem has been fixed. In general we only need to shift the vector `r` to the right and apply some calculations. Any other data needed by the calculation (A for example) needs to be calculated in this new order, or, if only sequential computation is possible, can be placed in memory and loaded into a vector register when a stride of $-T+1$ is used.

Note that what we actually do is delay the calculation of the tail components until the input that they need is available. This is akin to placing pipeline stages between these tail components in the VLSI architecture.

This technique can be applied to both the architectures. In the case of the incremental block-state the variables A, B, R just happen to be vectors with N elements instead of scalars.

The technique also allows for easy implementation. If the function f can be calculated by a scalar program, then the calculation of f for each tail-component is the same program, but now it is executed in parallel T times.

4 Incremental Block State

In this section we examine how the incremental block state architecture can be implemented on a vector processor.

4.1 Description

An IIR filter can be be described by the state space form;

$$
\begin{aligned}
x(n+1) &= Ax(n) + bu(n) \\
y(n) &= c^T x(n) + du(n)
\end{aligned}
$$

Where $x(n)$ denotes the state vector at time instant n. And the n-th input and output are denoted by $u(n)$ and $y(n)$ respectively. This means that A is a $N \times N$ matrix, b and c are vectors of length N and d is a scalar.

Note that there are many possible combinations of A, b, c and d that describe the same filter. However, for convenience we usually choose one where A is a block diagonal matrix with block size 2. Finding this representation is akin to splitting the filter in to parallel second order sections.

From the state space form we can create the block state space form with the help of the following matrices:

$$
A^{(X)} = A^X
$$

$$
B^{(X)} = \begin{pmatrix} A^{X-1}b & A^{X-2}b & \dots & b \end{pmatrix}
$$

$$
C^{(X)} = \begin{pmatrix} c^T & c^T A & \dots & c^T A^{X-1} \end{pmatrix}
$$

$$
[D^{(X)}]_{i,j} = \begin{cases} 0 & \text{for } i < j \\ d & \text{for } i = j \\ c^T A^{i-j-1} b & \text{for } i > j \end{cases}
$$

In addition we can start working with vector of inputs and outputs:

$$
\begin{aligned}
u^{(X)}(n) &= \begin{pmatrix} u(n) & u(n+1) & \dots & u(n+X-1) \end{pmatrix}^T \\
y^{(X)}(n) &= \begin{pmatrix} y(n) & y(n+1) & \dots & y(n+X-1) \end{pmatrix}^T
\end{aligned}
$$

With these definitions the following statement holds:

$$
x((q+1)P) = A^{(P)} x(qP) + B^{(P)} u^{(P)}(qP)
$$

This allows us to advance the state of the system with P steps in a single calculation.

To calculate the outputs the incremental block state architecture uses a tail of P div $I - 1$ components with the following specifications:

$$
\begin{pmatrix} y^{(I)}(qP+rI) \\ x(qP+rI+I) \end{pmatrix} = \begin{pmatrix} C^{(I)} & D^{(I)} \\ A^{(I)} & B^{(I)} \end{pmatrix} \begin{pmatrix} x(qP+rI) \\ u^{(I)}(qP+rI) \end{pmatrix}
$$

So this is effectively our function f for the method described in Section 3.2.

These components all calculate a set of I outputs. Where I is an additional design parameter with the constraints that $1 \leq I \leq P$.

The last set of outputs of a block P is calculated by:

$$
\begin{aligned}
y^{(I+P \bmod I)}(qP + (P \text{ div } I - 1)I) &= C^{(I+P \bmod I)} x(qP + (P \text{ div } I - 1)I) \\
&\quad + D^{(I+P \bmod I)} u^{(I+P \bmod I)}(pQ + (P \text{ div } I - 1)I)
\end{aligned}
$$

A schematic overview of the VLSI circuit is displayed in figure 1.

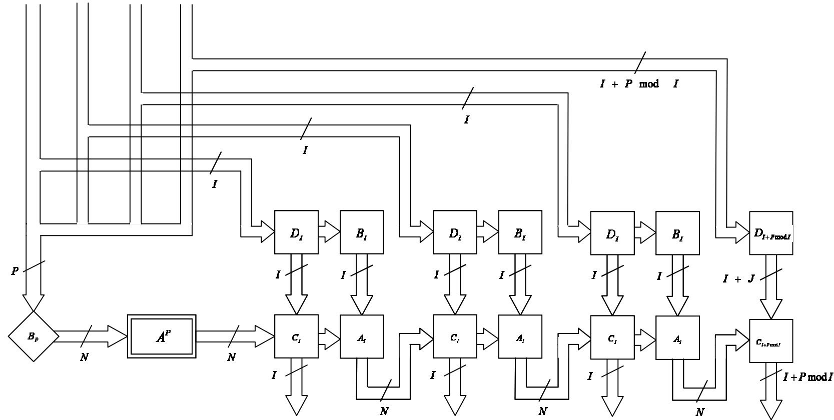

Figure 1: Schematic overview of the Incremental Block State architecture

4.2 Analysis

The multiplication with the matrix $B^{(P)}$ produces N outputs which each take P multiplications and additions to calculate. The multiplications can happen in parallel, but the additions form a problem. We do not want to perform NP additions in such a way that it costs $O(P)$ clock cycles. Therefore we need an intra-vector operation (also called a vector reduction operation) that sums the elements of a vector. This also means that we need $\lceil \frac{P}{V} \rceil$ of such operations for each of the N results. Since we already have $P = \alpha V$, this is no problem.

The multiplication with matrix $A^{(P)}$ provides no difficulties, since the number of operations does not depend on P.

Then there is the tail of the architecture. We can use the method described in section 3.2, but there is a small problem; the last element of the tail is a bit different from all the other elements. To make things easier we could choose I such that $P \bmod I = 0$.

If we take a closer look at the tail elements, they all consist of four matrix multiplications. The matrices $A^{(I)}, B^{(I)}, C^{(I)}$ and $D^{(I)}$ have sizes $N \times N, N \times I, I \times N$ and $I \times I$ respectively. And the length of the tail is $P \text{ div } I - 1$ plus the last tail element.

If we choose $P \text{ div } I = V$, then we can represent the inputs of each tail-component in $I + N$ vectors. Each element r of such a vector then represents the input for tail-component r. We need $I + N$ such vectors, because the input of a tail-component consists of N scalars describing the state and I scalars containing the inputs of the filter. In a similar manner the outputs of each tail-component can be represented in $I + N$ vectors of length $P \text{ div } I$, because each tail component produces I filter outputs and N states (except for the last component, which produces no state). So, when we choose $P \text{ div } I = V$ the vector registers are used in their entirety, (except for the last position of the vector containing the state) which is the most efficient choice we can make.

Since P will not occur in the final clock cycle count, we would like to choose it as large as possible (given N and V). This way we will minimize the number of

clock cycles per output. This leads us to the choice $P = IV$, since it satisfies all our demands $P = \alpha V, P \bmod I = 0$ and $P \operatorname{div} I = V$.

Note that we could relax the restriction $P \operatorname{div} I = V$ to $P \operatorname{div} I = \beta V$, but this does not help us since it will just lead us to choose $P = \gamma IV$, which will only serve to increase the number of clock cycles in both the basic part (multiplication with $B^{(P)}$) and the tail part (every V components we would have to switch over to a new set of vector registers).

4.3 Implementation

To actually implement the architecture on a vector processor we choose an array of vectors to hold the input state required by each tail-component. Like this:

$$Inv_0 : [\![\texttt{x(i)}]\!]_r = x((q-r)P + rI)_i \text{ for } 0 \leq r < V \ \wedge \ 0 \leq i < N$$

Note that we use `x(i)` to indicate the `i`-th element of the array. We do not use `x[i]` to avoid confusion with the notation for elements of a vector.

Also note that initialization for $q = 0$ is easy. We just set each vector to zero.

4.3.1 State update (Matrix A)

We can calculate the next state by establishing:

$$\begin{aligned}
&[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP) + B^{(P)}u^{(P)}(qP) \text{ for } 0 \leq i < N \\
&[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I) + B^{(I)}u^{(I)}((q+1-r)P + (r-1)I) \\
&\qquad \text{for } 0 \leq i < N \ \wedge \ 1 \leq r < V
\end{aligned}$$

Since A is block diagonal with a block size of two, every result of the matrix multiplication with $A^{(X)}$ is the result of at most two multiplications and one addition.

We make another assumption; that the matrix A is block diagonal with a block size of 2 for the first $2N_2$ rows, and diagonal for the N_1 last rows. This effectively means a parallel decomposition of the filter contains N_2 second order sections and N_1 first order sections (i.e. $N = 2N_2 + N_1$) and that we have ordered them to ease our implementation.

So we place the following restrictions on A:

$$Pred_0 : \begin{array}{l}
A_{2i,j} = 0 \text{ for } (j < 2i \ \vee \ j > 2i+1) \ \wedge \ 0 \leq i < N_2 \\
A_{2i+1,j} = 0 \text{ for } (j < 2i \ \vee \ j > 2i+1) \ \wedge \ 0 \leq i < N_2 \\
A_{i,j} = 0 \text{ for } i \neq j \ \wedge \ 2N_2 \leq i < N
\end{array}$$

This predicate specifies that for the first $2N_2$ rows A is a block diagonal matrix with a block size of 2, and for the last $N_1 = N - 2N_2$ rows the matrix is diagonal.

We store this matrix in memory according to the following predicate:

$$
MemA: \begin{array}{l}
[\![\texttt{MemA}]\!]_{4iV} = A^{(P)}_{2i,2i} \text{ for } 0 \le i < N_2 \\
[\![\texttt{MemA}]\!]_{4iV+j} = A^{(I)}_{2i,2i} \text{ for } 0 \le i < N_2 \ \wedge \ 1 \le j < V \\
[\![\texttt{MemA}]\!]_{4(i+1)V} = A^{(P)}_{2i,2i+1} \text{ for } 0 \le i < N_2 \\
[\![\texttt{MemA}]\!]_{4(i+1)V+j} = A^{(I)}_{2i,2i+1} \text{ for } 0 \le i < N_2 \ \wedge \ 1 \le j < V \\
[\![\texttt{MemA}]\!]_{4(i+2)V} = A^{(P)}_{2i+1,2i+1} \text{ for } 0 \le i < N_2 \\
[\![\texttt{MemA}]\!]_{4(i+2)V+j} = A^{(I)}_{2i+1,2i+1} \text{ for } 0 \le i < N_2 \ \wedge \ 1 \le j < V \\
[\![\texttt{MemA}]\!]_{4(i+3)V} = A^{(P)}_{2i+1,2i} \text{ for } 0 \le i < N_2 \\
[\![\texttt{MemA}]\!]_{4(i+3)V+j} = A^{(I)}_{2i+1,2i} \text{ for } 0 \le i < N_2 \ \wedge \ 1 \le j < V \\
[\![\texttt{MemA}]\!]_{4N_2V+iV} = A^{(P)}_{2N_2+i,2N_2+i} \text{ for } 0 \le i < N_1 \\
[\![\texttt{MemA}]\!]_{4N_2V+iV+j} = A^{(I)}_{2N_2+i,2N_2+i} \text{ for } 0 \le i < N_1 \ \wedge \ 1 \le j < V
\end{array}
$$

This allows us to write the following pseudo code:

Pre: $[\![\texttt{N1}]\!] = N_1 \ \wedge \ [\![\texttt{N2}]\!] = N_2$
$Inv_0 \ \wedge \ MemA$

Post: $[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP)$ for $0 \le i < N$
$[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I)$
for $0 \le i < N \ \wedge \ 1 \le r < V$

```
proc PartialUpdate1(MemA:memory,
                    x:array [0..N) of vector,
                    N1, N2:constants);
var i,s:scalar;
    aa: address;
    m,t:vector;
|[
  i:=0;
  aa:=0;
  do N2 times →
    (x(i):=shiftright(x(i)) || s:=x(i)[0]);
    (x(i+1):=shiftright(x(i+1)) || x(i)[0]:=s);
    (t:=x(i) || s:=x(i+1)[1]);
    (m,aa:=MemA[aa..aa+V),aa+V || x(i+1)[0]:=s);
    x(i):=m * t;
    m,aa:=MemA[aa..aa+V),aa+V;
    x(i):=MAC(x(i), m, x(i+1));
    m,aa:=MemA[aa..aa+V),aa+V;
    x(i+1):=m * x(i+1);
    m,aa:=MemA[aa..aa+V),aa+V;
    (x(i+1):=MAC(x(i+1), m, t) || i:=i+2);
  od
  do N1 times →
    (x(i):=shiftright(x(i)) || s:=x(i)[0]);
    (m,aa:=MemA[aa..aa+V),aa+V || x(i)[0]:=s);
    (x(i):=m * x(i) || i:=i+1);
  od
```

```
]|
```

Note that a vector processor may not have the capability to access several vector registers as if they form an array. In this case the loop needs to be rolled out. This may, in fact, be desirable in other cases too. If N_2 and/or N_1 are fairly small it may not be more advantageous to unroll it. However, since we do not know the values of N_2 and N_1 we will not perform this unrolling.

Many more small optimizations may be possible. We could, for example, save memory by calculating $A^{(P)}x(qP)$ with scalar operations. With the current program the memory contains $4N_2 + N_1$ vectors with values from either $A^{(P)}$ or $A^{(I)}$. By calculating $A^{(P)}x(qP)$ separately in scalar registers (and later put the results back in the x vectors) all the calculations with the $A^{(I)}$ matrices can use vectors with the same value for each element. And these vectors can thus be replaced by scalers containing that value. This results in storing $4N_2 + N_1$ scalars in memory (instead of vectors) for $A^{(I)}$ and $4N_2 + N_1$ scalars for $A^{(P)}$. Which results in a significant reduction of memory usage. However, the number of scalar operations in this case will exceed the number of vector operations and thus (with our vector processor model) it will increase the number of clock cycles needed to perform the procedure.

The `PartialUpdate1` procedure requires 2 clock cycles to initialize the variables `i` and `aa` and 2 clock cycles to start each loop. The loops take, respectively, 11 and 3 clock cycles per iteration. Thus we get a total of $4 + 11N_2 + 3N_1$ clock cycles. Because $N = 2N_2 + N_1$ we can simplify this to:

$$3N + 5N_2 + 4 \text{ clock cycles}$$

4.3.2 State update (Matrix $B^{(I)}$)

To update the state we still have to calculate the results of the multiplications with $B^{(I)}$ and $B^{(P)}$. We will focus on $B^{(I)}$ first, this one can be easily implemented by multiplying with one column of $B^{(I)}$ at a time.

We store $B^{(I)}$ in memory according to the following predicate:

$$MemBI: \ [\![\texttt{MemBI}]\!]_{iN+j} = B^{(I)}_{j,i} \text{ for } 0 \le i < I \ \wedge \ 0 \le j < N$$

Pre: $[\![\texttt{qP}]\!] = qP \ \wedge \ MemBI$
$[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP)$ for $0 \le i < N$
$[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I)$
for $0 \le i < N \ \wedge \ 1 \le r < V$
$[\![\texttt{Input}]\!]_i = u(i)$ for $qP + (V-1)(-P+I) \le i < qP + I$

Post: $[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP)$ for $0 \le i < N$
$[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I) +$
$B^{(I)}u^{(I)}((q+1-r)P + (r-1)I)$ for $0 \le i < N \ \wedge \ 1 \le r < V$

```
proc PartialUpdate2(Input, MemBI:memory,
                    x:array [0..N) of vector,
                    qP:constant);
var i,s:scalar;
```

```
    ia, ma:address;
    v:vector;
|[
  ia:=qP;
  ma:=0;
  do I times →
    (v,ia:=Input[ia,..,(V-1)(-P+I)+ia],ia+1 || i:=0);
    (v:=shiftright(v) || s,ma:=MemBI[ma],ma+1);
    do N times →
      (x(i):=MAC(x(i), v, s) || s,ma:=MemBI[ma],ma+1);
      i:=i+1;
    od
  od
]|
```

Note that unrolling the inner loop, when N is small, is very desirable. Currently the inner loop costs $2N+1$ clock cycles, but if we can get rid of the index variable `i`, then we only need N clock cycles. So unrolling would save $IN+1$ clock cycles for the entire procedure.

The `PartialUpdate2` procedure requires $3+I(3+2N)$ clock cycles, if we expand that we get:

$$2IN + 3I + 3 \text{ clock cycles}$$

4.3.3 State Update (Matrix $B^{(P)}$)

Now we only need to look at the multiplication with $B^{(P)}$ (a $N \times P$ matrix). We do not want P to get into the number of required clock cycles, so we will need to perform the calculation row by row and thus use intra-vector operations. Because a row of matrix $B^{(P)}$ has length P it will have to be split up among several vectors, I to be exact. So this brings us to the following program:

We store $B^{(P)}$ in memory according to the following predicate:

$$MemBP: \quad [\![\texttt{MemBP}]\!]_{iVN+jV+k} = B^{(P)}_{j,iV+k} \quad \text{for } 0 \le i < I \ \wedge\ 0 \le j < N \ \wedge\ 0 \le k < V$$

Pre: $[\![\texttt{qP}]\!] = qP \ \wedge\ MemBP$
$[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP)$ for $0 \le i < N$
$[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I)+$
$B^{(I)}u^{(I)}((q+1-r)P + (r-1)I)$ for $0 \le i < N \ \wedge\ 1 \le r < V$
$[\![\texttt{Input}]\!]_i = u(i)$ for $qP \le i < (q+1)P$

Post: $[\![\texttt{x(i)}]\!]_0 = A^{(P)}x(qP) + B^{(P)}u^{(P)}(qP)$ for $0 \le i < N$
$[\![\texttt{x(i)}]\!]_r = A^{(I)}x(q+1-r)P + (r-1)I)+$
$B^{(I)}u^{(I)}((q+1-r)P + (r-1)I)$ for $0 \le i < N \ \wedge\ 1 \le r < V$

```
proc PartialUpdate3(Input, MemBP:memory,
                    x:array [0..N) of vector,
                    qP:constant);
var i,s1,s2:scalar;
```

```
    ia,ma:address;
    v,m,t:vector;
|[
  ia:=qP
  (m:=MemBP[0..0+V) || ma:=0);
  do I times →
    (v,ia:=Input[ia..ia+V),ia+V || i:=0);
    t:=v * m;
    do N times →
      (s1:=Sum(t) || s2:=x(i)[0]);
      (m,ma:=MemBP[ma..ma+V),ma+N || s1:=s1+s2);
      (t:=v * m || x(i)[0]:=s1);
      i:=i+1;
    od
  od
]|
```

Again we see that we would like to unroll the inner loop, since the statement `i:=i+1` would become obsolete when the loop is unrolled, resulting in a decrease in the clock cycle count of the procedure by $IN + 1$. However, the total clock cycle count of `PartialUpdate3` is now:

$$4IN + 3I + 3 \text{ clock cycles}$$

4.3.4 Output Computation

Next to the state update the program also needs to calculate the outputs of the filter. The outputs are calculated with the help of the multiplications with the matrices $C^{(I)}$ and $D^{(I)}$.

We store $C^{(I)}$ and $D^{(I)}$ in memory according to the following predicate:

$$MemCD: \begin{array}{l} [\![\texttt{MemC}]\!]_{iN+j} = C^{(I)}_{i,j} \text{ for } 0 \le i < I \ \wedge\ 0 \le j < N \\ [\![\texttt{MemD}]\!]_{\frac{i(i+1)}{2}+j} = D^{(I)}_{i,j} \text{ for } 0 \le i < I \ \wedge\ 0 \le j \le i \end{array}$$

These multiplications can then be implemented in the following program:

Pre: $[\![\texttt{qP}]\!] = qP \ \wedge\ Inv_0 \ \wedge\ MemCD$
$[\![\texttt{Input}]\!]_i = u(i)$ for $qP + (V-1)(-P+I) \le i < qP + I$

Post: $[\![\texttt{Output}]\!]_{(q-r)P+rI+i} = y((q-r)P + rI + i)$
for $0 \le r < V \ \wedge\ 0 \le i < I$

```
proc CalculateOutputs(Input,Output,MemC,MemD:memory,
                      x:array [0..N) of vector,
                      qP:constant);
var ia,oa,da,ca:address;
    out,v:vector;
    j,s1,s2:scalar;
|[
  i:=0;
```

```
  oa:=qP;
  da:=0;
  ca:=0;
  s2,ca:=MemC[ca],ca+1;
  do I times →
    (out:=0 || ia:=qP);

    do i+1 times →
      (v,ia:=Input[ia,..,(V-1)(-P+I)+ia],ia+1 ||
          s1,da:=MemD[da],da+1);
      (out:=MAC(out, v, s1) || j:=0);
    od
```

$\{[\![\texttt{out}]\!]_r = D^{(I)}u^{(I)}((q-r)P + rI + i) \text{ for } 0 \leq r < V\}$

```
    do N times →
      (out:=MAC(out, x(j), s2) || s2,ca:=MemC[ca],ca+1);
      j:=j+1;
    od
```

$\{[\![\texttt{out}]\!]_r = C^{(I)}x((q-r)P + rI + i) +$
$D^{(I)}u^{(I)}((q-r)P + rI + i) \text{ for } 0 \leq r < V\}$
$\{[\![\texttt{out}]\!]_r = y((q-r)P + rI + i) \text{ for } 0 \leq r < V\}$

```
    (Output[oa,..,(V-1)(-P+I)+oa],oa:=out,oa+1 || i:=i+1);
  od
]|
```

Again we see that unrolling the (second) inner loop is not a bad idea. We would get rid of the statement `j:=j+1` and thereby save $IN + 1$ clock cycles. But as it stands now the number of clock cycles required by the procedure `CalculateOutputs` is:

$$I^2 + 2IN + 5I + 6 \text{ clock cycles}$$

4.3.5 Main program

The main program is now a combination of the procedures presented before:

Pre: $[\![\texttt{qP}]\!] = [\![\texttt{oa}]\!] = qP \ \wedge\ [\![\texttt{N1}]\!] = N_1 \ \wedge\ [\![\texttt{N2}]\!] = N_2$
$Inv_0 \ \wedge\ Pred_0 \ \wedge\ MemA \ \wedge\ MemBI \ \wedge\ MemBP \ \wedge\ MemCD$
$[\![\texttt{Input}]\!]_i = u(i) \text{ for } qP + (V-1)(-P+I) \leq i < (q+1)P + I$

Post: $[\![\texttt{Output}]\!]_{(q-r)P+rI+i} = y((q-r)P + rI + i)$
$\quad \text{for } 0 \leq r < V \ \wedge\ 0 \leq i < I$
$Inv(q := q+1)$

```
proc CalculateBlock(Input, Output,
                    MemA, MemBI, MemBP,
                    MemD, MemC:memory,
                    x:array [0..N) of vector,
                    N1, N2, qP:constants);
|[
```

```
  CalculateOutputs(Input,Output,MemC,MemD,x,qP);
  PartialUpdate1(MemA,x,N1,N2);
  PartialUpdate2(Input,MemBI,x,qP);
  PartialUpdate3(Input,MemBP,x,qP);
]|
```

This procedure can be executed multiple times to calculate multiple blocks of outputs. By incrementing `qP` with P after each call the output stream will be calculated.

In any case the clock cycle cost of the main program is the sum of the procedures it calls, resulting in a total of:

$$I^2 + 8IN + 11I + 3N + 5N_2 + 16 \text{ clock cycles}$$

4.4 Clock cycles per output

To be able to compare our architectures we take a look at the number of clock cycles per output. To do this we assume the worst case scenario: the parallel decomposition of the filter contains second order sections only.

This results in $I^2+8IN+11I+\frac{11}{2}N+16$ clock cycles per P outputs. And since $P = IV$ holds we can calculate which choice for I will lead to the minimum number of clock cycles per output. The optimum turns out to be located at:

$$I_{opt1} = \sqrt{16 + \frac{11N}{2}}$$

So the number of clock cycles per output is:

$$CPO_{IBS1} = 8\frac{N}{V} + O(\frac{\sqrt{N}}{V})$$

Note, however that we could save $3(IN + 1)$ clock cycles in total if N is small. Because if N is small we can unroll some of the loops in the program and get a total clock cycle count of $I^2 + 5IN + 11I + \frac{11}{2}N + 13$. This time the optimum choice for I is:

$$I_{opt2} = \sqrt{13 + \frac{11N}{2}}$$

And the number of clock cycles per output in that case is:

$$CPO_{IBS2} = 5\frac{N}{V} + O(\frac{\sqrt{N}}{V})$$

This results in a significant reduction in processing time.

4.5 Conclusion

The program presented in this section scales well with the vector size V; it requires a fixed number of clock cycles (for a given filter order N) and the number of outputs calculated in those clock cycles is linear with V.

The program can be implemented on our vector processor. However it is possible to implement it on any vector processor with the same scalability if that vector processor has two crucial capabilities: the intra-sum and the memory access with stride.

The intra-sum operation is a intra-vector or vector reduction operation which adds up all the elements in the vector and thus produces a scalar output. It is needed to calculate the result of a matrix multiplication where the height of the matrix is independent of V, but the width is not.

Accessing the memory with a "stride" (a distance other than 1 between the vector elements) is necessary to simulate the pipelining of the VLSI architecture. Moreover this stride is negative in our current program, can cover multiple vector lengths and is not necessarily a multiple of the vector length. The stride can be made positive by transforming the program, but the fact that it can be quite large might still pose a problem. However, this, or an operation like it, is needed to effectively implement this program.

5 Clustered Look-ahead

In this section we examine the clustered look-ahead architecture with incremental output computation (also called direct form block filter). This filter may suffer from stability problems, but these can be solved by increasing its order as described by Lim and Liu [**?**].

5.1 Description

An IIR filter of order N can be described by:

$$\begin{aligned} y(n) &= \textstyle\sum_{i=1}^{N} a_i y(n-i) + z(n) \\ z(n) &= \textstyle\sum_{i=0}^{N} b_i u(n-i) \end{aligned}$$

Where $y(n)$ and $u(n)$ denote the n-th output and input of the filter respectively.

The relation for $y(n)$ can be rewritten [**?**]:

$$y(n) = \sum_{j=0}^{N-1} \left[\sum_{k=j+1}^{N} a_k r_{j+P-N-k} \right] y(n-j-P+N) + \sum_{j=0}^{P-N-1} r_j z(n-j)$$

Where:

$$\begin{aligned} r_i &= 0 && \text{for } i < 0 \\ r_0 &= 1 \\ r_i &= \textstyle\sum_{k=1}^{N} a_k r_{i-k} && for i > 0 \end{aligned}$$

In addition we can start working with vectors of inputs and outputs:

$$\begin{aligned} z^{(X)}(n) &= \begin{pmatrix} z(n) & z(n+1) & \dots & z(n+X-1) \end{pmatrix}^T \\ y^{(X)}(n) &= \begin{pmatrix} y(n) & y(n+1) & \dots & y(n+X-1) \end{pmatrix}^T \end{aligned}$$

Then we get:

$$\begin{array}{ll} y^{(N)}(qP+P) = Ay^{(N)}(qP) + Bz^{(P)}(qP+N) & \text{for } P \geq N \\ y^{(P)}(qP+P) = Cy^{(N)}(qP+P-N) + Dz^{(P)}(qP+P) & \text{for } P < N \end{array}$$

The remaining $P-N$ outputs when $P \geq N$ are calculated in a non-recursive manner, with a list of tail-components using $y(n) = \sum_{i=1}^{N} a_i y(n-i) + z(n)$ as the function f from Section 3.2.

The matrices have the following values:

$$\begin{array}{ll} A_{i,j} = \sum_{l=1}^{j} a_{N-l+1} r_{P-N+i-j+l-1} & C_{i,j} = \sum_{l=1}^{j} a_{N-l+1} r_{i-j+l-1} \\ B_{i,j} = r_{P-N+i-j} & D_{i,j} = r_{i-j} \end{array}$$

Matrix A is $N \times N$, matrix B is $N \times P$, matrix C is $P \times N$ and matrix D is $P \times P$. So clearly, only when $P \geq N$ holds is the amount of required hardware for the VLSI implementation linear with P.

5.2 Analysis

There are several stages in this calculation.

First, there is the calculation of $z^{(P)}$. This requires $P(N+1)$ multiplications, which can be implemented on a vector processor easily as $O(N+1)$ operations on vectors of length P. This corresponds with our demand that $P = \alpha V$.

Secondly, we have the multiplication with matrix B or D. In case of B we need the intra-vector operations to make sure that P does not get into the number of required clock cycles. So again this corresponds with $P = \alpha V$. For matrix D however, we will need P^2 multiplications, so there is no way of mapping that on a vector processor with $P = \alpha V$ without getting P into the number of required clock cycles. Since multiplication with D takes place only when $P < N$ we would like to have $P \geq N$.

Thirdly, there is the multiplication with A or C. Since we already have $P \geq N$ we know that multiplications with C do not occur. And multiplication with matrix A can be implemented easily without P becoming part of the number of clock cycles, since it is a $N \times N$ matrix.

And, finally, there is the tail of components that calculate the remaining $P-N$ outputs. Each of these components require N multiplications and therefore it should be easy to implement. However, the length of the tail is $P-N$. So we would like $P - N = \beta V$, so that we can make efficient use of all the vector elements.

But this leads to $(\alpha - \beta) = \frac{N}{V}$ and the chances that α and β can be integers for a given N and V are very small. In fact, it would require that N is a multiple of V, but in most cases N is fairly small compared to V, so most likely $\frac{N}{V} \leq 1$.

Since $V \geq N$ is common, we will limit ourselves to that case in this report. Since we are interested in scalability we also choose $\alpha = 1$ or $P = V$, so that for $\lim_{V \to \infty}$ we have $\beta \to 1$. We do not choose $\beta = 1$ because the number of clock cycles will depend on $\lceil \alpha \rceil$ and $\lceil \beta \rceil$. And for $\beta = 1$ we get $\alpha > 1$ when $V \geq N$. While we get $\beta < 1$ for $alpha = 1$.

5.3 Implementation

In this section we will look at the implementation of the architecture in detail. The implementation consists of several components; the tail, multiplication with A, multiplication with B, and the calculation of the values of z.

Since the values of z are needed by the tail component as well as in the multiplication with B we introduce a buffer to keep them. This buffer adheres to the following invariant:

$$Inv_0: \begin{array}{l} [\![\texttt{Memz}]\!]_{P^2+P+((q-1) \bmod (P+1))P-i-1} = z(qP+N-i-1) \\ \quad \text{for } 0 \leq i < P^2+P \\ [\![\texttt{Memz}]\!]_{((q-1) \bmod (P+1))P-i-1} = z(qP+N-i-1) \\ \quad \text{for } 0 \leq i < ((q-1) \bmod (P+1))P \end{array}$$

This invariant states that we keep a circular buffer containing values of z. Furthermore, the buffer is duplicated such that all values occur twice. This has been done because we need to read vectors from the buffer where some elements may cross the buffer boundary, and our vector processor does not have the capability to do modulo addressing.

Note that this invariant is easy to establish when $q = 0$, in that case the entire memory segment `Memz` can be filled with zeros.

5.3.1 Tail Calculation

Because the tail is the part of the calculation that will drive the rest, we will handle it first. Note that there only is always a tail because we decided that $P = V$ and thus $P \geq N$.

Each tail component needs $N + 1$ inputs, N of these inputs are actually the previous outputs of the entire system. The other input is the result of a matrix-vector multiplication (one of the results of a multiplication with $z^{(P)}$). And we assume that that one is available.

The filter coefficients need to be stored in memory according to the following predicate:

$$Mema: \ [\![\texttt{Mema}]\!]_i = a_{i+1} \text{ for } 0 \leq i < N$$

Pre: $[\![\texttt{N}]\!] = N \ \wedge \ [\![\texttt{P}]\!] = P \ \wedge \ [\![\texttt{qP}]\!] = qP$
$[\![\texttt{qmP}]\!] = (q \bmod (P+1))P \ \wedge \ [\![\texttt{PP}]\!] = P^2$
$Inv_0(q := q+1) \ \wedge \ Mema$
$[\![\texttt{Output}]\!]_{(q-i)P+N+i-j} = y((q-i)P+N+i-j)$
for $0 \leq i < V \ \wedge \ 1 \leq j \leq N$

Post: $[\![\texttt{Output}]\!]_{(q-i)P+N+i} = y((q-i)P+N+i)$ for $0 \leq i < V$

Using the invariant:

$$[\![\texttt{y}]\!]_i = \sum_{j=1}^{n} a_j y((q-i)P+N+i-j) + z((q-i)P+N+i) \text{ for } 0 \leq i < V$$

we can come up with the following procedure:

```
proc CalculateTail(Mema,Memz,Output:memory,
                   N,P,qP,qmP,PP:constant);
var ma,ia,oa,za:address;
    r,v:vector;
    s:scalar;
|[
  oa:=qP+N;
  ma:=0;
  za:=PP+qmP;
  (r:=Memz[za,..,(V-1)(-P+1)+za] || ia:=oa-1);
  {n:=0}
  do N times →
    (v,ia:=Output[ia,..,(V-1)(-P+1)+ia],ia+-1 ||
         s,ma:=Mema[ma],ma+1);
    r:=MAC(r, v, s);
    {n:=n+1}
  od
  Output[oa,..,(V-1)(-P+1)+oa]:=r,oa;
]|
```

Note that this procedure actually calculates V outputs, instead of $P-N$. This means that the last $V-(P-N)$ outputs are calculated unnecessarily (they have already been calculated before). But, except for possibly higher round-off errors, these calculations produce the same outputs, so except for the efficiency loss, there is no real harm done.

The total clock cycle count for the `CalculateTail` procedure is:

$$2N + 6 \text{ clock cycles}$$

5.3.2 Multiplication with A

We also need a piece of code that calculates the outputs qP through $qP+N-1$. This is done by multiplying some existing outputs with the matrix A. Since A is a $N \times N$ matrix the procedure will not contain P in the number of clock cycles. However, we would like to make the most efficient use of the vector processor, so we extend the matrix A to be a $V \times N$ matrix. The extra elements all contain zeroes.

This leads to the following predicate for the memory layout:

$$MemA: \begin{array}{l} [\![\texttt{MemA}]\!]_{jV+i} = A_{i,j} \text{ for } 0 \le i < N \ \wedge\ 0 \le j < N \\ [\![\texttt{MemA}]\!]_{jV+i} = 0 \text{ for } N \le i < V \ \wedge\ 0 \le j < N \end{array}$$

Pre: $[\![\texttt{N}]\!] = N \ \wedge\ [\![\texttt{qP}]\!] = qP \ \wedge\ MemA$
$[\![\texttt{bz}]\!]_i = (Bz^{(P)}(qP+N))_i$ for $0 \le i < N$
$[\![\texttt{bz}]\!]_i = 0$ for $N \le i < V$
$[\![\texttt{Output}]\!]_{qP+i} = y(i)$ for $0 \le i < N$

Post: $[\![\texttt{Output}]\!]_{qP+P+i} = y(qP+P+i)$ for $0 \le i < N$
$[\![\texttt{Output}]\!]_{qP+P+i} = 0$ for $N \le i < V$

Note that we also place some values in output locations $qP + P + N$ through $qP + P + V$. This causes no harm, since these values are overwritten later on.

```
proc MultiplyA(MemA,Output:memory,
                 qP,N:constant,
                 bz:vector);
var ma,ia,oa:address;
    r:vector;
    s:scalar;
|[
  oa:=qP+P;
  ma:=0;
  (r:=bz || ia:=qP);
  do N times →
    (v,ma:=MemA[ma..ma+V),ma+V || s,ia:=Output[ia],ia+1);
    r:=MAC(r, v, s);
  od
  Output[oa..oa+V),oa:=r,oa+V;
]|
```

The procedure `MultiplyA` requires a total of:

$$2N + 4 \text{ clock cycles}$$

5.3.3 Multiplication with B

Now that we have procedures for the calculation of the outputs (y) we still need some procedures to calculate z and $Bz^{(P)}$ to fill the `Memz` buffer and `bz` vector used in the procedures above.

We will look at the calculation for $Bz^{(P)}$ first. We place the matrix B (which is a $N \times V$ matrix because $P = V$) in memory according to the following predicate:

$$MemB: \ [\![\texttt{MemB}]\!]_{jV+i} = B_{j,i} \text{ for } 0 \le i < V \ \wedge \ 0 \le j < N$$

Pre: $[\![\texttt{N}]\!] = N \ \wedge \ MemB$
$[\![\texttt{z}]\!]_i = z(qP + N + i)$ for $0 \le i < P$

Post: $[\![\texttt{bz}]\!]_i = (Bz^{(P)}(qP + N))_i$ for $0 \le i < N$
$[\![\texttt{bz}]\!]_i = 0$ for $N \le i < V$

```
proc CalculateBz(MemB, Memz:memory,
                   N:constant,
                   bz,z:vector);
var s1,s2,oa:scalar;
    ma:address;
    m,t:vector;
|[
  (bz:=0 || ma:=0);
  (m:=MemB[ma..ma+V) || oa:=-1);
  do N times →
```

```
    t:=z * m;
    (s1:=Sum(t) || oa:=oa+1);
    (m,ma:=MemB[ma..ma+V),ma+N || bz[oa]:=s1);
  od
]|
```

The `CalculateBz` procedure requires in total:

$$3N + 3 \text{ clock cycles}$$

5.3.4 Calculation of z

Now all that remains to do is to find a procedure to fill the `Memz` buffer. We need to establish $Inv_0(q := q + 1)$ and $[\![\texttt{z}]\!]_i = z(qP + N + i)$ for $0 \leq i < P$ according to `CalculateBz` and `CalculateTail` respectively. Fortunately the vector `z` can be written to the `Memz` buffer to establish Inv_0.

To calculate z we need the filter coefficients to be stored according to the following predicate:

$$Memb: \ [\![\texttt{Memb}]\!]_i = b_i \text{ for } 0 \leq i \leq N$$

Pre: $[\![\texttt{N}]\!] = N \ \wedge \ [\![\texttt{qP}]\!] = qP \ \wedge \ Memb \ \wedge \ Inv_0$
$[\![\texttt{Memb}]\!]_i = b_i$ for $0 \leq i \leq N$
$[\![\texttt{Inputs}]\!]_i = u(i)$ for $qP \leq i < (q + 1)P + N$

Post: $Inv_0(q := q + 1)$
$[\![\texttt{z}]\!]_i = z(qP + N + i)$ for $0 \leq i < P$

```
proc Calculatez(Inputs,Memb,Memz:memory,
                N,qP:constant,
                z:vector);
var s:scalar;
    ba,ia,oa: address;
    v:vector;
|[
  ba:=0;
  oa:=qP+N;
  (z:=0 || ia:=qP);
  do (N+1) times →
    (v,ia:=Inputs[ia..ia+V),ia+1 || s,ba:=Memb[ba],ba+1);
    z:=MAC(z, v, s);
  od
  Memz[oa..oa+V):=zV;
]|
```

This procedure (`Calculatez`) requires in total:

$$2N + 5 \text{ clock cycles}$$

5.3.5 Main program

Now that we have all the necessary procedures, we construct the main program.

Pre: $[\![\texttt{N}]\!] = N \ \wedge\ [\![\texttt{P}]\!] = P \ \wedge\ [\![\texttt{qP}]\!] = qP \ \wedge\ [\![\texttt{qmP}]\!] = (q \bmod (P+1))P$
$Mema \ \wedge\ MemA \ \wedge\ Memb \ \wedge\ MemB \ \wedge\ Inv_0$
$[\![\texttt{Inputs}]\!]_i = u(i)$ for $qP \le i < (q+1)P + N$
$[\![\texttt{Output}]\!]_{qP+i} = y(i)$ for $0 \le i < N$
$[\![\texttt{Output}]\!]_{(q-i)P+N+i-j} = y((q-i)P + N + i - j)$
for $0 \le i < V \ \wedge\ 1 \le j \le N$

Post: $Inv_0(q := q+1)$
$[\![\texttt{Output}]\!]_{qP+P+i} = y(qP + P + i)$ for $0 \le i < N$
$[\![\texttt{Output}]\!]_{qP+P+i} = 0$ for $N \le i < V$
$[\![\texttt{Output}]\!]_{(q-i)P+N+i} = y((q-i)P + N + i)$ for $0 \le i < V$

```
proc CalculateBlock(
          Mema,MemA,Memb,MemB,Memz:memory,
          Input, Output:memory,
          N,P,qP,qmP:constant);

var bz, z:vector;
 |[
  Calculatez(Input, Memb, Memz, N, qP,  z);
  CalculateBz(MemB, Memz, N, bz, z);
  MultiplyA(MemA, Output, qP, N, bz);
  CalculateTail(Mema, Memz, Output, N, P, qmP, PP);
 ]|
```

This procedure can be called several times to calculate the output stream of the filter. After each call the value of q will have to be increased by one and thus the values of `qP` and `qmP` will have to be updated.

The `CalculateBlock` procedure requires:

$$9N + 18 \text{ clock cycles}$$

5.4 Clock cycles per output

The number of clock cycles per output can be calculated, and turns out to be:

$$CPO_{CLA} = 9\frac{N}{V} + O(\frac{1}{V})$$

5.5 Conclusion

The program presented in this section scales well with V. And just as with the incremental block state architecture the crucial operations for this scalability are the intra-sum operation and the ability to load and store vectors in memory with a ”stride”.

But unlike the incremental block state program of section 4 this program has some drawbacks. First of all the stability is an issue that needs to be handled

separately [?]. Secondly we only have a linear speed-up if $V \geq N$. And finally, the number of clock cycles per output (CPO_{CLA}) is worse than the number of clock cycles per output for the incremental block state (CPO_{IBS1} or CPO_{IBS2}).

6 Conclusion

In this report we presented two vector processor programs which implement an IIR filter on a vector processor. These implementations are based on the incremental block state and block direct form presented by Parhi and Messerschmitt in [?]. And like their VLSI counterparts the throughput of the implementation scales linearly with the amount of hardware (which, in the case of the vector processor, is the vector size).

Interestingly the clustered look-ahead implementation, which is the best implementation for VLSI is not the best for the vector processor. At least, not with our vector processor model.

A reason for this difference is that the clustered look-ahead program contains a multiplication with a $N \times N$ matrix, which cannot be implemented in such a way that the (unrelated) vector size V is used optimally. The incremental block state program also contains $N \times N$ matrices, but these are exactly V in number, making it possible to use the vector size efficiently.

So please note that other models may yield other results. If the vector processor would support true VLIW parallelism, for example, then most of the loops would take one clock cycle per iteration. This would severely reduce the difference between the implementations.

However, even when the difference is small it might still be better to choose the incremental block state implementation. The reason for this is the stability problems of the clustered look-ahead approach. These stability problems can be solved, but this often means that the order of the filter (N) needs to be increased.

Next to presenting the two programs this report also makes an important observation; there are two features of the vector processor that are crucial to achieve scalability. One of them is the intra-sum operation, which adds all the elements of a vector together. And the other is that it must be possible to access the memory with a so-called "stride", i.e. accessing memory locations that are not necessarily adjacent, but can be several locations apart.

This is not to say that it is impossible to use these implementations on a vector processor lacking such capabilities. It may be very well possible to substitute a procedure for these critical instructions. If the clock cycle count of those procedures depends on V then the scalability will be lost, but the resulting implementation might still offer a speed up that is higher than those of other implementations.

The most important observation in this report, however, is the general method presented in Section 3.2. This method allows us to apply the incremental output computation technique to vector processor programs. So this method is not only limited to translating IIR implementations, but can be used to translate any

specification or VLSI implementation (which is based on the incremental output computation technique) to a vector processor program.

References