

Systematic derivation of IIR filter programs for a vector processor

M.G. v.d. Horst

December 4, 2007

1 Introduction

In this article we will examine how an IIR filter can be implemented on a vector processor.

An IIR filter is a program that produces an output stream R given an input stream A according to the following relationship:

$$R(i) = \sum_{j=0}^N c_j A(i-j) + \sum_{j=1}^N d_j R(i-j)$$

where both $A(i) = 0$ and $R(i) = 0$ for $i < 0$.

By examining all possible choices in the program derivation we hope to obtain all possible (logical) programs that implement this filter.

FIXME: This is a very basic introduction, rewrite it once the rest is finished.

2 Vector Processor

FIXME: Description of the vector processor goes here.

3 Single Output Program Derivation

In this section we will derive several programs that calculate one element of the output stream at a time. These outputs will be placed in the `Output` array, while the inputs are read from the `Input` array.

Although the calculation never ends and the arrays are assumed to have an infinite length, these programs can actually be used in practice. The system just has to read in a new (finite) array of inputs and output the outputs that have already been calculated from time to time. This can happen interleaved with the calculation of the outputs, or, if the vector processor is not the only processing element of the system, it can happen in parallel. Of course this interleaving or

parallel process needs to be synchronized properly with the calculation process, but that is beyond the scope of this text.

In any case, this section is about programs that calculate one output at a time, so the basis is:

```
var i: integer;
    out: scalar;

i:=0;
do forever →

    ...

    {out = R(i)}
    Output [i] := out;

    ...

    i:=i+1;
od
```

But we would still like to make use of the vector processor's capability to perform vector calculations.

Since we require one output at a time we can model the vector operations as a set of V parallel processes that produce the required output. This has the advantage that we can use design techniques suited for such parallel system.

So, in fact, we try to simulate the parallel system with the vector processor. The different processes communicate via channels. These channels can be represented by a vector register and the shift operation can be used to move values between the processes. In a similar manner the MAC operation can be seen as broadcasting one scalar value to all the processes and having them multiply and add the result to a communication channel.

There are two obvious choices to design such systems. One is to have such a system calculate the output stream R directly. Another is to have several systems each calculating part of the output streams.

In both approaches we will be looking for recurrence relations between the different processes we simulate. But these are specific recurrence relations: we need that the value communicated by a process depends on a value that has been calculated in the past. This seems obvious, but for truly parallel systems it is possible to have a value that depends on a value calculated by an other process *currently*, in such a case the processes will wait for that "current" value and thus make it fall in the past. However, on the vector processor we can only simulate systems that operate in lock step. It is not possible to simulate one process waiting for another, because that would destroy the vector parallelism¹.

¹Perhaps unless processes wait in groups, then we could simulate each group of processes separately

3.1 One system

We use a vector register \mathbf{b} to indicate the state of the parallel system. To read the output of this system into the variable `out` we add the statement `out := b[N]` to the program. (We do not write the output directly to memory because of the load-store architecture of the processor).

So we have a parallel system of N processor that has to calculate $R(i)$. There are several techniques to obtain this system.

3.1.1 LRLA High

One of the possibilities is a last-received-last-accumulated strategy with a high generalisation. This basically means that we choose the following output streams for the components:

$$b_n(i) = \sum_{j=0}^n c_{j+N-n} A(i-j) + \sum_{j=1}^n d_{j+N-n} R(i-j) \text{ for } 0 \leq n \leq N$$

Note that the indices for c and d are not according to the standard LRLA High strategy. The reason we did this, is to make sure we find a recurrence relation:

$$\begin{aligned} & b_n(i) \\ = & \{ \text{Definition of } b_n \} \\ & \sum_{j=0}^n c_{j+N-n} A(i-j) + \sum_{j=1}^n d_{j+N-n} R(i-j) \\ = & \{ \text{Split off } j=0 \text{ and } j=1 \text{ respectively } (n \geq 1) \} \\ & \sum_{j=1}^n c_{j+N-n} A(i-j) + \sum_{j=2}^n d_{j+N-n} R(i-j) + c_{N-n} A(i) + d_{N-n+1} R(i-1) \\ = & \{ \text{Dummy change } j := j+1 \} \\ & \sum_{j=0}^{n-1} c_{j+N-n+1} A(i-j-1) + \sum_{j=1}^{n-1} d_{j+N-n+1} R(i-j-1) + c_{N-n} A(i) + d_{N-n+1} R(i-1) \\ = & \{ \text{Definition of } b_n \} \\ & b_{n-1}(i-1) + c_{N-n} A(i) + d_{N-n+1} R(i-1) \end{aligned}$$

And for b_0 we have $b_0(i) = c_N A(i)$.

This leads to the following program:

Pre: $\llbracket \text{Input} \rrbracket_n = A(n)$ for $n \geq 0$
 $\llbracket \mathbf{c} \rrbracket_n = c_{N-n}$ for $0 \leq n \leq N$
 $\llbracket \mathbf{d} \rrbracket_0 = 0$
 $\llbracket \mathbf{d} \rrbracket_n = d_{N-n+1}$ for $1 \leq n \leq N$

Post: $\llbracket \text{Output} \rrbracket_n = R(n)$ for $n \leq i$

```
var i: integer;
    b: vector;
    out, a: scalar;
```

```
i:=0;
b:=0;
a:=Input[i];
b:=MAC(b, c, a);
```

```

do forever →
  {[[b]]n = bn(i) for 0 ≤ n ≤ N}
  out := b[N]
  {out = R(i)}
  Output[i] := out

  b := shiftright(b);
  {[[b]]n = bn-1(i) for 1 ≤ n ≤ N ∧ [[b]]0 = 0}

  a := Input[i+1];
  {a = A(i+1)}
  b := MAC(b, c, a);
  {[[b]]n = bn-1(i) + cN-nA(i+1) for 1 ≤ n ≤ N ∧ [[b]]0 = b0(i+1)}

  b := MAC(b, d, out);
  {[[b]]n = bn(i+1) for 0 ≤ n ≤ N}

  i := i+1;
od

```

If we assume the processor can execute multiple operations in parallel, and we remove the index variable because of zero-overhead looping we get the program below. In it we also make the memory addressing explicit by introducing variables that contain the input and output address.

```

var b: vector;
    out, a: scalar;
    ia, oa: address;

//i:=0;
(b:=0 || ia:=1 || oa:=0 || a:=Input[0]);
b:=MAC(b, c, a);
a,ia:=Input[ia],ia+1;
do forever →
  {[[b]]n = bn(i) for 0 ≤ n ≤ N ∧ a = A(i+1)}
  (b:=shiftright(b) || out:=b[N]);
  {[[b]]n = bn-1(i) for 1 ≤ n ≤ N ∧ [[b]]0 = 0 ∧ out = R(i)}
  (b:=MAC(b, c, a) || Output[oa],oa:=out,oa+1);
  {[[b]]n = bn-1(i) + cN-nA(i) for 1 ≤ n ≤ N ∧ [[b]]0 = b0(i+1)}
  (b:=MAC(b, d, out) || a,ia:=Input[ia],ia+1);
  {[[b]]n = bn(i+1) for 0 ≤ n ≤ N ∧ a = A(i+2)}
  //i:=i+1;
od

```

Note that we postponed the output operation and advanced the input operation. We did this to allow our program to be run on vector processors that support only one scalar operation in parallel to a vector operation. Advancing inputs and postponing outputs usually requires some extra buffering space, so we would expect to need more registers. However with the amount of advancing and postponing that we have applied we do not incur such a penalty.

3.1.2 LRLA Low

It is also possible to choose a low generalisation for the LRLA strategy:

$$b_n(i) = \sum_{j=N-n}^N c_j A(i-j) + \sum_{j=1+N-n}^N d_j R(i-j) \text{ for } 0 \leq n \leq N$$

Then we get the recurrence relation:

$$\begin{aligned} & b_n(i) \\ = & \{ \text{Definition of } b_n \} \\ & \sum_{j=N-n}^N c_j A(i-j) + \sum_{j=1+N-n}^N d_j R(i-j) \\ = & \{ \text{Split off } j = N - n \text{ and } j = 1 + N - n \text{ respectively } (n \geq 1) \} \\ & \sum_{j=N-n+1}^N c_j A(i-j) + \sum_{j=2+N-n}^N d_j R(i-j) + \\ & \quad c_{N-n} A(i - N + n) + d_{N-n+1} R(i - 1 - N + n) \\ = & \{ \text{Definition } b_n(i) \} \\ & b_{n-1}(i) + c_{N-n} A(i - N + n) + d_{N-n+1} R(i - 1 - N + n) \end{aligned}$$

But this relationship indicates that an output along the channel b does not depend on previous outputs, but on current outputs. This makes it impossible to use vector parallelism, since we would have to calculate $b_0(i)$ before $b_1(i)$, which has to be calculated before $b_2(i)$, etc.

But maybe we could use the relation if we use the invariant $[\mathbf{b}]_n = b_n(i + N - n)$ (instead of $[\mathbf{b}]_n = b_n(i)$), but this would be just a detour which leads to the same recurrence relation between the vector elements (and hence the same program) as in the previous section.

Another possibility would be to alter the indices in the A and R stream to lead to a more appropriate recurrence relation. This would look like:

$$b_n(i) = \sum_{j=N-n}^N c_j A(N - n + i - j) + \sum_{j=1+N-n}^N d_j R(N - n + i - j) \text{ for } 0 \leq n \leq N$$

But this is the same generalisation as in the previous section, just apply a dummy change ($j := j + N - n$) and we have the exact same generalisation.

3.1.3 FRLA High

Another strategy is the first-received-last-accumulated strategy. We will apply it here with a high generalisation:

$$b_n(i) = \sum_{j=0}^n c_j A(N - n + i - j) + \sum_{j=1}^n d_j R(N - n + i - j) \text{ for } 0 \leq n \leq N$$

We altered the indices of A and R to make sure we would find a recurrence

relation with a lesser index i on the right hand side:

$$\begin{aligned}
& b_n(i) \\
= & \{ \text{Definition of } b_n \} \\
& \sum_{j=0}^n c_j A(N - n + i - j) + \sum_{j=1}^n d_j R(N - n + i - j) \\
= & \{ \text{Split off } j = n \ (n \geq 1) \} \\
& \sum_{j=0}^{n-1} c_j A(N - n + i - j) + \sum_{j=1}^{n-1} d_j R(N - n + i - j) + \\
& \quad c_n A(N + i - 2n) + d_n R(N + i - 2n) \\
= & \{ \text{Definition } b_n(i) \} \\
& b_{n-1}(i - 1) + c_n A(N + i - 2n) + d_n R(N + i - 2n)
\end{aligned}$$

So we found a recurrence relation which can be implemented with a shift operation. But only for the b channel. We will also have to implement channels that communicate the necessary A and R 's.

It is possible to implement the channel communicating the elements of the A stream. But for the R stream there is a problem. These values are not available until we have calculated them. And process 1 needs the value $R(N - 2 + i)$, even though we are still calculating $R(i)$ in process N . So this implementation is not going to work.

3.1.4 FRLA Low

Instead of the high generalisation we can also use a low generalisation with the FRLA strategy.

$$b_n(i) = \sum_{j=N-n}^N c_{j-N+n} A(i-j) + \sum_{j=1+N-n}^N d_{j-N+n} R(i-j) \text{ for } 0 \leq n \leq N$$

Note that we altered the index of c and d again to get a recurrence relation:

$$\begin{aligned}
& b_n(i) \\
= & \{ \text{Definition of } b_n \} \\
& \sum_{j=N-n}^N c_{j-N+n} A(i-j) + \sum_{j=1+N-n}^N d_{j-N+n} R(i-j) \\
= & \{ \text{Split off } j = N \} \\
& \sum_{j=N-n}^{N-1} c_{j-N+n} A(i-j) + \sum_{j=1+N-n}^{N-1} d_{j-N+n} R(i-j) + \\
& \quad c_n A(i-N) + d_n R(i-N) \\
= & \{ \text{Dummy change } j := j - 1 \} \\
& \sum_{j=N-n+1}^N c_{j-N+n-1} A(i-j+1) + \sum_{j=2+N-n}^N d_{j-N+n-1} R(i-j+1) + \\
& \quad c_n A(i-N) + d_n R(i-N) \\
= & \{ \text{Definition of } b_n \} \\
& b_{n-1}(i+1) + c_n A(i-N) + d_n R(i-N)
\end{aligned}$$

This is a problematic recurrence relation, we effectively have to look into the future to calculate $b_n(i)$. Unless we somehow repair the indices of A and R in the generalisation to lead to a more favorable recurrence relation. This leads to:

$$b_n(i) = \sum_{j=N-n}^N c_{j-N+n} A(2N - 2n + i - j) + \sum_{j=1+N-n}^N d_{j-N+n} R(2N - 2n + i - j)$$

But this is just the same generalisation as in the previous section. Just apply the dummy change $j := j + N - n$ to this one and we end up with the one of the previous section. So this will not lead to an alternate solution.

3.2 Two systems

In the previous sections we looked at a program with one parallel subsystem as a basis. However, it is also possible to calculate the recursive and non-recursive part of the output streams with two distinct systems.

It seems like there is little to gain here; the implementation in the previous section required one shift operation and two MAC operations. If we use two systems we will need at least one shift and one MAC operation for each of them. So in total we will need more vector operations.

However, if the processor is capable of executing multiple vector operations in parallel we might gain something. The operations in the program in the previous sections are all performed on one vector register and thus cannot be done in parallel. However, simulating two parallel systems will result in two vector registers and thus it might be possible to do these vector operations in parallel.

In any case we do not have to apply all design strategies (like we did in the last section) because we know that there is only one that will work for the R stream; the LRLA High strategy.

We use the following generalisation:

$$q_n(i) = \sum_{j=1}^n d_{j+N-n} R(i-j) \text{ for } 0 \leq n \leq N$$

Which leads to these recurrence relations:

$$\begin{aligned} q_n(i) &= q_{n-1}(i-1) + d_{N-n+1} R(i-1) \text{ for } 1 \leq n \leq N \\ q_0(i) &= 0 \end{aligned}$$

However, for the A stream there are multiple strategies that will lead to a valid recurrence relationship. So we examine each one.

3.2.1 LRLA High

For the second system we choose the following generalisation:

$$p_n(i) = \sum_{j=0}^n c_{j+N-n} A(i-j) \text{ for } 0 \leq n \leq N$$

Note that $q_N(i) + p_N(i) = R(i)$.

This leads us to the following recurrence relation:

$$\begin{aligned} p_n(i) &= p_{n-1}(i-1) + c_{N-n} A(i) \text{ for } 1 \leq n \leq N \\ p_0(i) &= c_N A(i) \end{aligned}$$

Which, in turn, leads us to the following program:

Pre: $\llbracket \text{Input} \rrbracket_n = A(n)$ for $n \geq 0$
 $\llbracket c \rrbracket_n = c_{N-n}$ for $0 \leq n \leq N$
 $\llbracket d \rrbracket_0 = 0$
 $\llbracket d \rrbracket_n = d_{N-n+1}$ for $1 \leq n \leq N$

Post: $\llbracket \text{Output} \rrbracket_n = R(n)$ for $n \leq i$

```

var i: integer;
    p,q: vector;
    out, a: scalar;

i:=0;
p:=0;
q:=0;
a:=Input[i];
p:=MAC(p, c, a);
do forever →
  { $\llbracket p \rrbracket_n = p_n(i) \wedge \llbracket q \rrbracket_n = q_n(i)$  for  $0 \leq n \leq N$ }
  out:=p[N] + q[N];
  {out = R(i)}

  p:=shiftright(p);
  { $\llbracket p \rrbracket_n = p_{n-1}(i)$  for  $0 \leq n \leq N$ }
  q:=shiftright(q);
  { $\llbracket q \rrbracket_n = q_{n-1}(i)$  for  $0 \leq n \leq N$ }

  a:=Input[i+1];
  {a = A(i+1)}
  p:=MAC(p, c, a);
  { $\llbracket p \rrbracket_n = p_n(i+1)$  for  $0 \leq n \leq N$ }

  q:=MAC(q, d, out);
  { $\llbracket q \rrbracket_n = q_n(i+1)$  for  $0 \leq n \leq N$ }

  Output[i]:=out
  i:=i+1;
od

```

Or, more compactly, with parallel vector operations and zero-overhead looping:

```

var p,q: vector;
    out, a: scalar;
    ia, oa: address;

//i:=0;
(p:=0 || q:=0 || ia:=1 || oa:=0 || a:=Input[0]);
p:=MAC(p, c, a);
do forever →
  { $\llbracket p \rrbracket_n = p_n(i) \wedge \llbracket q \rrbracket_n = q_n(i)$  for  $0 \leq n \leq N$ }
  (p:=shiftright(p) || q:=shiftright(q) ||

```

```

        out := p[N] + q[N] || a, ia := Input[ia], ia+1);
    {[p]n = pn-1(i) for 0 ≤ n ≤ N}
    {[q]n = qn-1(i) for 0 ≤ n ≤ N}
    {out = R(i) ∧ a = A(i + 1)}
    (p := MAC(p, c, a) || q := MAC(q, d, out) ||
     Output[oa], oa := out, oa+1);
    {[p]n = pn(i + 1) ∧ [q]n = qn(i + 1) for 0 ≤ n ≤ N}
    // i := i + 1;
od

```

If the processor does support the execution of multiple vector operations in parallel, but only if the vector operations are different (i.e. different functional units of the processor are used), then we can rewrite the program to:

```

var p, q: vector;
    out, a, tp: scalar;
    ia, oa: address;

// i := 0;
(p := 0 || q := 0 || ia := 1 || oa := 0 || a := Input[0]);
p := MAC(p, c, a);
{[p]n = pn(i) for 0 ≤ n ≤ N}
(p := shiftright(p) || tp := p[N] || a, ia := Input[ia], ia+1);
do forever →
    {[p]n = pn-1(i) for 0 ≤ n ≤ N}
    {[q]n = qn(i) for 0 ≤ n ≤ N}
    {tp = pN(i) ∧ a = A(i + 1)}
    (p := MAC(p, c, a) || q := shiftright(q) ||
     out := tp + q[N] || a, ia := Input[ia], ia+1);
    {[p]n = pn(i + 1) for 0 ≤ n ≤ N}
    {[q]n = qn-1(i) for 0 ≤ n ≤ N}
    {out = R(i) ∧ a = A(i + 2)}
    (p := shiftright(p) || q := MAC(q, d, out) ||
     Output[oa], oa := out, oa+1 || tp := p[N]);
    {[p]n = pn-1(i + 1) for 0 ≤ n ≤ N}
    {[q]n = qn(i + 1) for 0 ≤ n ≤ N}
    {tp = pN(i + 1)}
    // i := i + 1;
od

```

Note however that we need to use the statement `out := tp + q[N]`. So we can not hold on to our assumption that there is a load-store-like architecture between vector registers and scalar registers. The reason for this is simple; we are simulating two parallel systems so we need to read out the results of both systems (this could be done in parallel), combine these results (add them up in this case) and write the final result to memory. This makes a total of three operations which means that we would need 3 clock cycles per iteration. But by allowing the statement `out := tp + q[N]`, we can avoid this.

So this program is faster than the program derived in section 3.1.1, but only if the vector processor supports multiple vector operations in parallel.

3.2.2 FRLA High

The FRLA high strategy did not work for one system, because we would need to look into the future concerning stream R . However, our second system only depends on the A stream, so looking into the future is no problem; we just wait until the inputs are available.

So we choose the following generalisation:

$$p_n(i) = \sum_{j=0}^n c_j A(N - n + i - j) \text{ for } 0 \leq n \leq N$$

Note that, again, $q_N(i) + p_N(i) = R(i)$.

The generalisation leads us to the following recurrence relation:

$$\begin{aligned} p_n(i) &= p_{n-1}(i-1) + c_n A(N + i - 2n) \text{ for } 1 \leq n \leq N \\ p_0(i) &= c_0 A(N + i) \end{aligned}$$

Now there are, again, two possibilities. We can use a load-with-stride operation to obtain all the elements of the A stream we need, or we could introduce a channel to communicate the values of A .

We will examine this last option first. So we define a channel $a_n(i) = A(N + i - 2n)$. Since we need a recurrence relation with a lesser index i , we can only use $a_n(i) = a_{n+1}(i-2)$.

This leads us to the following program:

Pre: $\llbracket \text{Input} \rrbracket_n = A(n)$ for $n \geq 0$
 $\llbracket c \rrbracket_n = c_n$ for $0 \leq n \leq N$
 $\llbracket d \rrbracket_0 = 0$
 $\llbracket d \rrbracket_n = d_{N-n+1}$ for $1 \leq n \leq N$

Post: $\llbracket \text{Output} \rrbracket_n = R(n)$ for $n \leq i$

```

var i: integer;
    p,q,a,la: vector;
    out, t: scalar;

i:=0;
p:=0;
q:=0;
// establish  $\llbracket a \rrbracket_n = a_n(-1) \wedge \llbracket la \rrbracket_n = a_n(0)$  for  $0 \leq n \leq N$ 
...
{  $\llbracket a \rrbracket_n = a_n(-1) \wedge \llbracket la \rrbracket_n = a_n(0)$  for  $0 \leq n \leq N$  }
do forever →
  {  $\llbracket p \rrbracket_n = p_n(i) \wedge \llbracket q \rrbracket_n = q_n(i)$  for  $0 \leq n \leq N$  }
  {  $\llbracket a \rrbracket_n = a_n(i-1) \wedge \llbracket la \rrbracket_n = a_n(i)$  for  $0 \leq n \leq N$  }
  out := p[N] + q[N];
  {out = R(i)}

p:=shiftright(p);

```

```

    {[p]n = pn-1(i) for 0 ≤ n ≤ N}
    q:=shiftright(q);
    {[q]n = qn-1(i) for 0 ≤ n ≤ N}
    a:=shiftright(a);
    {[a]n = an(i+1) for 0 ≤ n < N}

    t:=Input [i+1-N];
    {t = A(i+1-N)}
    a[N]:=t;
    {[a]n = an(i+1) for 0 ≤ n ≤ N}
    p:=MAC(p, c, a);
    {[p]n = pn(i+1) for 0 ≤ n ≤ N}

    q:=MAC(q, d, out);
    {[q]n = qn(i+1) for 0 ≤ n ≤ N}

    a, la:=la, a;
    {[a]n = an(i) ∧ [la]n = an(i+1) for 0 ≤ n ≤ N}

    Output [i]:=out
    i:=i+1;
od

```

Note that we now need more vector operations than we did in the previous section. In addition we now need to perform three operations on vector **a**. First of all we need to shift it, we need to assign a value to **a**[N] and we need to assign the value of **la** to it. This means we need at least three operations per loop, so this is not a faster program than in the previous section. And, at most, it is just as fast as the program of section 3.1.1.

But we still had another option; if the vector processor supports it we can load the *A* stream into the vector **a** directly with a so-called load-with-stride operation. This gives us a program with the same preconditions, but with the following statements:

```

var i: integer;
    p,q,a: vector;
    out: scalar;

i:=0;
p:=0;
q:=0;
do forever →
    {[p]n = pn(i) ∧ [q]n = qn(i) for 0 ≤ n ≤ N}
    out:=p[N] + q[N];
    {out = R(i)}

    p:=shiftright(p);
    {[p]n = pn-1(i) for 0 ≤ n ≤ N}
    q:=shiftright(q);
    {[q]n = qn-1(i) for 0 ≤ n ≤ N}

```

```

a:=Input [N+i+1,-2+N+i+1..V(-2)+N+i+1];
{[[a]]n = an(i + 1) for 0 ≤ n ≤ N}

p:=MAC(p, c, a);
{[[p]]n = pn(i + 1) for 0 ≤ n ≤ N}

q:=MAC(q, d, out);
{[[q]]n = qn(i + 1) for 0 ≤ n ≤ N}

Output [i] := out
i := i+1;
od

```

This program is a lot like the one in section 3.1.1. The differences are that instead of a loading a scalar value into scalar register **a** a set of values is loaded into vector register **a**. Furthermore the MAC operation with **a** is now a vector-vector MAC instead of a vector-scalar MAC. But that is as far as the differences go. If the load-with-stride operation is supported by the vector processor the two programs have equal performance.

Note that if a processor supports only positive strides (i.e. 2 instead of -2) we can change the indices everywhere to have the parallel systems produce their outputs at $p_0(i)$ and $q_0(i)$ respectively.

References