FIR filters

Bernard Goossens

May 13, 2025

Contents

1 Introduction

Mathematically, a FIR filter of order N is the dot product of a vector of samples $x[N+1]$ and a vector of coefficients $h[N+1]$.

Given an input sequence $x[n]$ of n samples, $n \gg N$, the FIR filter produces an output sequence $y[n]$, where $\forall\ i,\ N \leq i \leq n$ (for $i < N$, the computed $y[i]$ are not physically meaningful):

$$y[i] = \sum_{k=0}^{N} h[k] \cdot x[i-k]$$

The coefficients $h[N+1]$ of a symmetric FIR low-pass filter with $N+1$ taps depend on a normalized cutoff frequency f_c. In the hardware design the exact values are not important. Only the number of bits matter, to size the arithmetic operators and the registers.

When the cutoff frequency has been fixed, the coefficients are themselves fixed and they can be defined as an array of constants. This is crucial because the hardware can be designed according to the exact coefficients. Multiplying by a small constant can be implemented as a succession of shifts and adds instead of a multiplier.

Two successive values in the y sequence are computed according to the two following formulas:

$$y[i] = h[0] \cdot x[i] + ... + h[N] \cdot x[i - N]$$

$$y[i+1] = h[0] \cdot x[i+1] + ... + h[N] \cdot x[i + 1 - N]$$

The computation of $y[i+1]$ overlaps the computation of $y[i]$, reusing the $x[i-N]$ to $x[i+1]$ inputs. It means that these inputs should be saved. The FIR filter can use a $N+1$ bit shift register.

In the remaining of the document, N is the number of taps, not the filter order. It is set to 16 (16 coefficients and 16 taps, FIR filter of order 15).

The product of two 8-bit signed integers is a 16-bit signed integer. Sixteen sums of such products expand to a 20-bit signed integer.

The cutoff frequency f_c is set as 0.03125, according to the following formula:

$$f_c \approx \frac{1}{2 \times 16} = \frac{1}{32} \approx 0.03125$$

The filter coefficients are computed from f_c and then scaled to 8-bit signed integers.

The coefficients used in the rest of the text are fixed as the following symmetric sequence (these values are also choosen because they will provide many interesting optimizations from the High-Level Synthesis (HLS) as many of them are powers of 2): { 2, 4, 8, 12, 16, 18, 20, 22, 22, 20, 18, 16, 12, 8, 4, 2}.

All the implementations presented in this report are coded in C/C++ and have been tested on a Pynq-Z1 board using the Xilinx Vitis HLS tool, version 2024.2. The Pynq-Z1 board has a ZYNQ XC7Z020-1CLG400C FPGA.

2 A FIR filter

Listing 1 shows the code to implement a FIR filter (file *fir_seq.cpp* in the archive).

Listing 1: A FIR filter IP

```
#include "ap_int.h"
#define N 16
const ap_int<8> h[N] = {
   2,  4,  8, 12, 16, 18, 20, 22,
  22, 20, 18, 16, 12,  8,  4,  2
};
void fir(
  ap_int<8>   x,
  ap_int<20> *y){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE s_axilite port=y
  static ap_int<8> shift_reg[N] = {0};
  ap_int<20> acc = 0;
  ap_uint<5> i;
  SHIFT_LOOP: for (i = N-1; i > 0; i--)
    shift_reg[i] = shift_reg[i-1];
  shift_reg[0] = x;
```

```
19    MAC_LOOP: for (i = 0; i < N; i++)
20      acc += shift_reg[i] * h[i];
21    *y = acc;
22  }
```

The code first updates the shift register (SHIFT_LOOP), then computes the dot product (MAC_LOOP) with an accumulation of the N products.

This code is first run as a C++ program by the Vitis simulator with the testbench *main* function presented in section 6.1.

Then, the fir function is synthesized, using the Vitis synthesizer. The synthesis produces a report containing a schedule of the different operations done in the RTL.

Figure 1 shows the schedule of the run. The $lnxx$ name designates the line number xx in the code. For example, 0_write_ln18 is the $shift\_reg[0]$ initialization in line 18 of the code shown on listing 1.

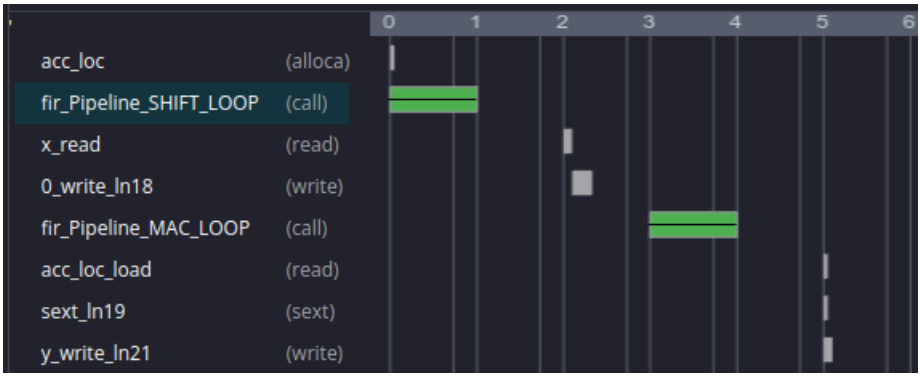

Figure 1: Schedule of the FIR component

The MAC_LOOP is scheduled after the SHIFT_LOOP. The input sample is saved in the $shift\_reg[0]$ register in between the two loops. The output of the computation is saved to the y location after the MAC_LOOP.

Figure 2 shows the schedule of the SHIFT_LOOP.

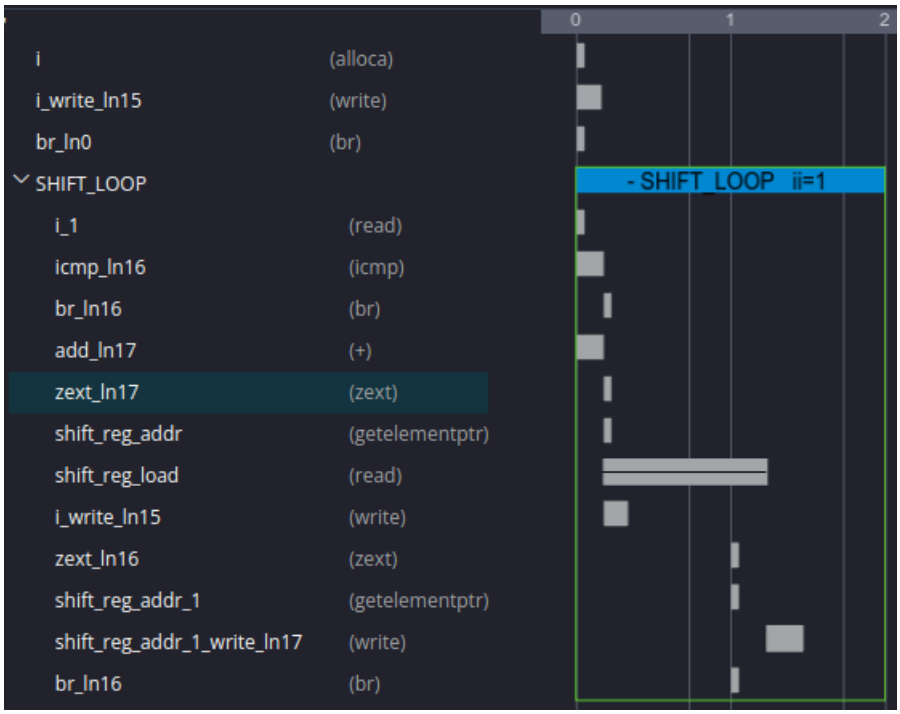

Figure 2: Schedule of the SHIFT_LOOP

Each iteration of the SHIFT_LOOP takes two cycles (latency). But a new iteration can start every cycle (throughput), as the $II = 1$ value shows (Initiation Interval).

Figure 3 shows the schedule of the MAC_LOOP.

Figure 3: Schedule of the MAC_LOOP

Each iteration of the MAC_LOOP takes five cycles (latency). But a new iteration can start every cycle (throughput), as the $II = 1$ value shows (Initiation Interval). In the first cycle, the shift register and the h constant are addressed and loaded ($shift\_reg\_addr$, $shift\_reg\_load$, h_addr, h_load). In the second cycle, the two values are multiplied (mul_ln20). The multiplier latency is two cycles. In the fourth cycle, the product is added to the accumulator (acc_1). The adder latency is one cycle. The accumulator is updated in the fifth cycle, which ends the iteration.

From these detailed figures, we can reconstitute the schedule of a call to function fir.

At the start of the run of the fir function, i.e. cycle 0, the input x is present.

The $shift\_reg$ array has been nullified. It is declared as *static*, which means that it keeps its value across the successive calls to function fir. The SHIFT_LOOP control is prepared.

The SHIFT_LOOP first iteration starts at cycle 1 with $shift\_reg[15] = shift\_reg[14]$ and ends with $shift\_reg[1] = shift\_reg[0]$ at cycle 16 (14 iterations in 14 cycles plus the latency of the last one, i.e. two cycles). There is a one cycle delay before the start of the loop, and a one cycle delay after, which are not shown on the schedule.

Cycle 18 is the loop exit, i.e. cycle 1 on figure 1.

At cycle 19 (cycle 2 on figure 1), the x input is consumed and $shift\_reg[0]$ is set.

The MAC_LOOP first iteration starts at cycle 21 (one cycle delay at the start of the loop). The loop runs for 20 cycles (15 iterations plus five cycles for the last one), from cycle 21 to 40, starting with $acc+ = shift\_reg[0] * h[0]$ and ending with $acc+ = shift\_reg[15] * h[15]$. Cycle 42 after the loop is cycle 4 on figure 1 (one cycle delay at the end of the loop).

At cycle 43 (cycle 5 on figure 1), the dot product is saved in the accumulator acc and can be output to $*y$. This is the first filtered value $y[0]$.

A new x, i.e. $x[1]$, can be input at cycle 44 and a new $*y$, i.e. the second filtered value $y[1]$, is output at cycle 87.

Variable i is defined as an *ap_uint*<5>, with five bits, as its successive values range from 0 to $N = 16$ for the MAC_LOOP. With *ap_uint*<4>, the loop would never end when run on the FPGA as i would never reach the limit $N = 16$ because the successor of 15 on four bits is zero.

Figure 4 shows the result of the synthesis. The full run is completed in 44 cycles (the latency of a function in the Vitis terminology is the number of cycles between its input and its output, i.e. one cycle less than the IP latency). A new run can start every 44 cycles (Initiation Interval II). The implementation uses no RAM and one DSP (to implement the multiplier).

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	IT LA	INTERVAL	TR CC	PI	BRAM	DSP	FF	LUT	URAM
fir (2)	43	430.000	-	44	-	no	0	1	212	373	0

Figure 4: Synthesis resources of the FIR component

Figure 5 shows that the timing estimation is 5.645 ns plus 2.70 ns of uncertainty, i.e. an estimated cycle duration of 8.345 ns. The total latency is 440 ns (44 cycles).

The size of the IP is given by the LUTs (LookUp Table), the FF (Flip-Flops) and the DSP employed: 1 DSP (one multiplier), 212 FF (16*8 bits for the shift register, 20 bits for the accumulator and 64 bits for various intermediate inter-cycle storings), and 373 LUTs (adders, multiplexers, ...).

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	5.645 ns	2.70 ns

Figure 5: Synthesis timing estimation of the FIR component

Figure 6 shows the Vivado construction to use the FIR component. It is composed of a Zynq7 Processing System in charge of running the testbench code. It also contains the FIR component, connected to the Zynq7 through an AXI interconnect IP (Axi SmartConnect). A System Reset IP (Processor System Reset) is added to provide the clocking and reset signals to all the IPs in the design.

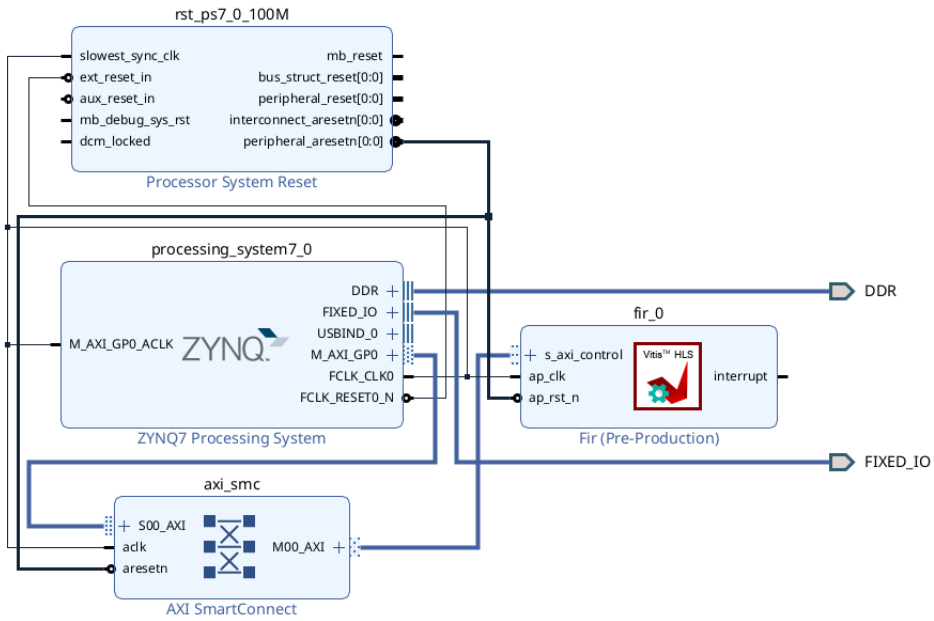

Figure 6: Vivado construction including the FIR component

Figure 7 shows the resources used in the FPGA to implement the whole system. The number of LUTs and the number of FFs is larger than what was shown by the Vitis synthesizer because it includes the resources used by the other IPs (Zynq7 Processing System, Axi SmartConnect and Processor System Reset).

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	562	0	0	53200	1.06
LUT as Logic	560	0	0	53200	1.05
LUT as Memory	2	0	0	17400	0.01
LUT as Distributed RAM	0	0			
LUT as Shift Register	2	0			
Slice Registers	710	0	0	106400	0.67
Register as Flip Flop	710	0	0	106400	0.67
Register as Latch	0	0	0	106400	0.00
F7 Muxes	1	0	0	26600	<0.01
F8 Muxes	0	0	0	13300	0.00

Figure 7: The FPGA resources used to implement the system including the FIR component

3 Pipelining to accelerate the throughput (and the latency)

Listing 2 (file *fir_pipeline.cpp*) shows the code to implement the same FIR filter with a HLS PIPELINE pragma. The pragma indicates that the synthesizer will organize the hardware in order to be able to start a new run after a single cycle (the Initial Interval II, set to 1 in the pragma).

Listing 2: A pipelined FIR filter IP

```
#include "ap_int.h"
#define N 16
const ap_int<8> h[N] = {
   2,  4,  8, 12, 16, 18, 20, 22,
  22, 20, 18, 16, 12,  8,  4,  2
};
void fir(
  ap_int<8>   x,
  ap_int<20> *y){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE s_axilite port=y
#pragma HLS PIPELINE II=1
  static ap_int<8> shift_reg[N] = {0};
#pragma HLS ARRAY_PARTITION variable=shift_reg complete
  ap_int<20> acc = 0;
  ap_uint<5> i;
  SHIFT_LOOP: for (i = N-1; i > 0; i--)
    shift_reg[i] = shift_reg[i-1];
  shift_reg[0] = x;
  MAC_LOOP: for (i = 0; i < N; i++)
    acc += shift_reg[i] * h[i];
  *y = acc;
}
```

To allow a new run after one cycle, the shift register must be updated in the first cycle. Hence, the SHIFT_LOOP is automatically fully unrolled and the registers are placed in FFs to allow a parallel access to the 16 cells. This is the role of the HLS ARRAY_PARTITION complete pragma.

Some form of saving of the intermediate computations must be added to all the pipeline stages of the design. For this reason, the synthesizer tries to minimize the number of stages through a highly parallelized organization of the products and sums. The 15 sums are arranged as a binary tree, forming a divide and conquer reduction. To compute the 16 products, only two multipliers are used. The synthesizer takes advantage of the constant array h.

The multiplications by 2, 4, 8 and 16 are simple shifts (latency 0: a shift is a simple bit selection and concatenation). The multiplications by 12, 18 and 20 are implemented by one shift and one add (one adder latency, i.e. less than a cycle). Only the multiplications by 22 are implemented by two DSP multipliers (the multiplier latency is two cycles).

With these tricks, the synthesizer is able to squeeze the computation of the dot product in five cycles (more or less one cycle per level of the reduction tree).

With pipelining, a new x can be input every cycle. The pipeline is controlled by a five states Finite State Machine (FSM).

Figures 8 to 12 show the schedule of the run.

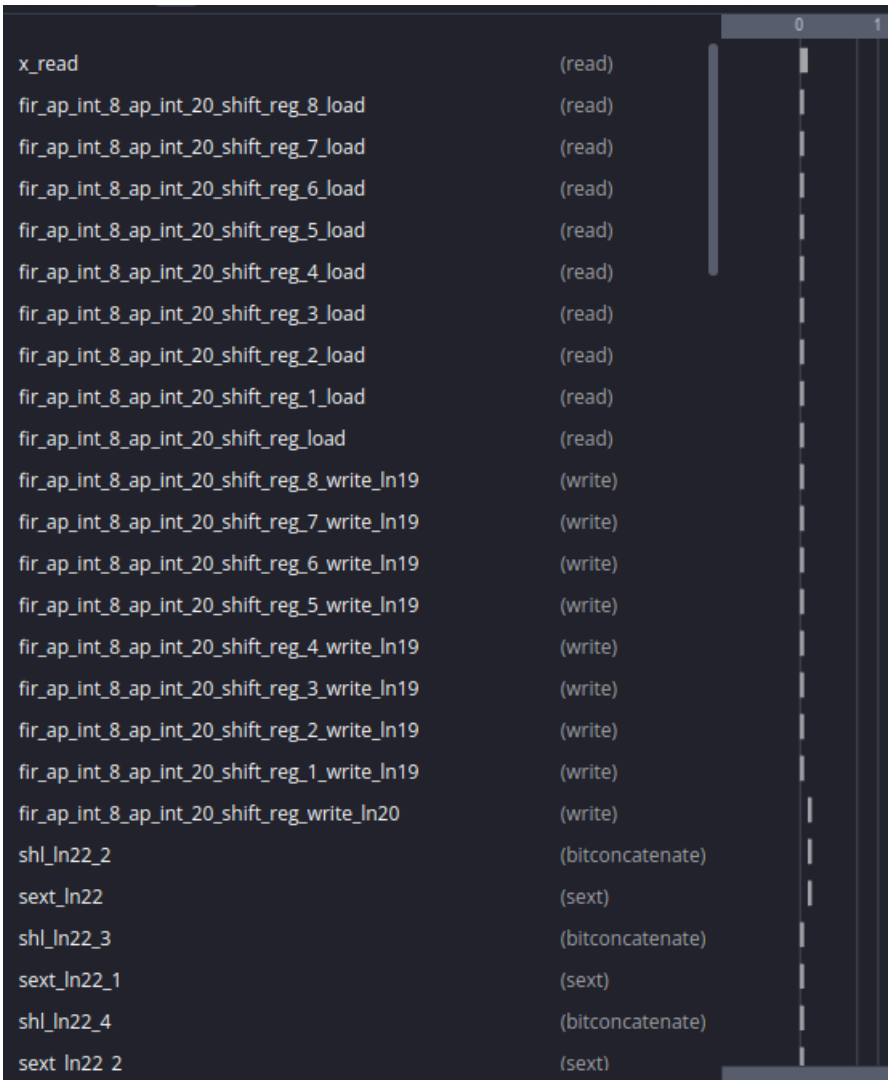

Figure 8: Schedule of the pipelined FIR component (part 1)

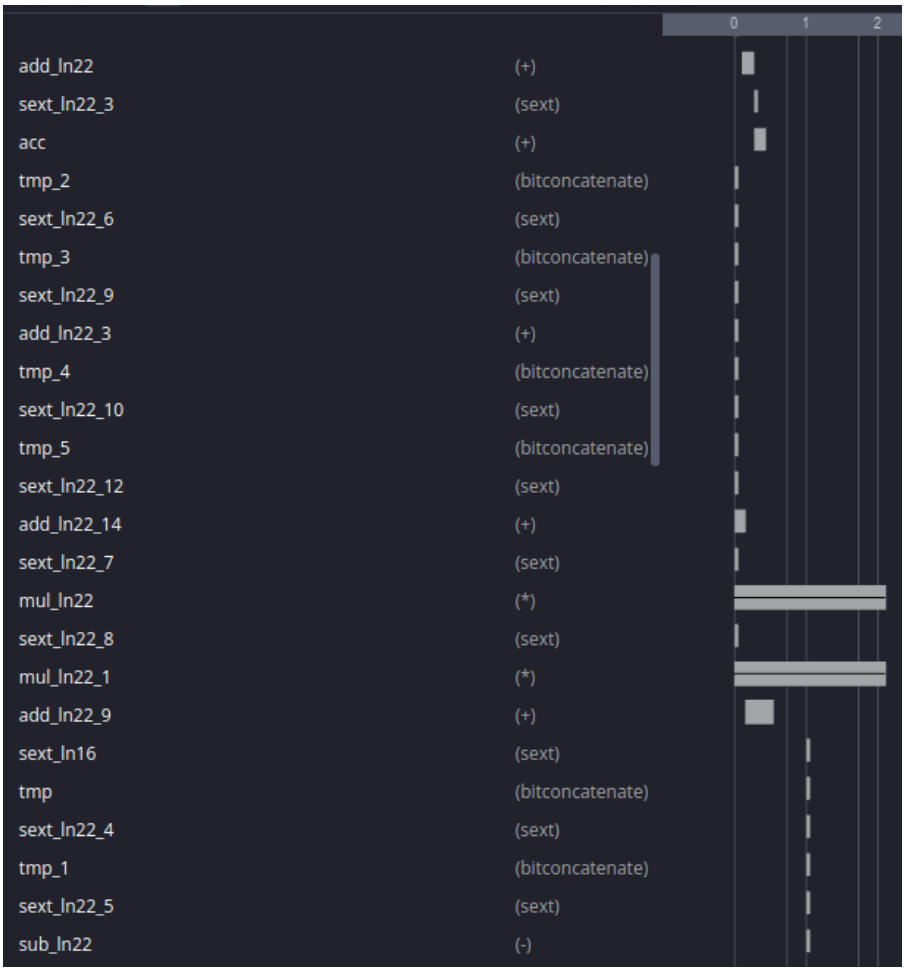

Figure 9: Schedule of the pipelined FIR component (part 2)

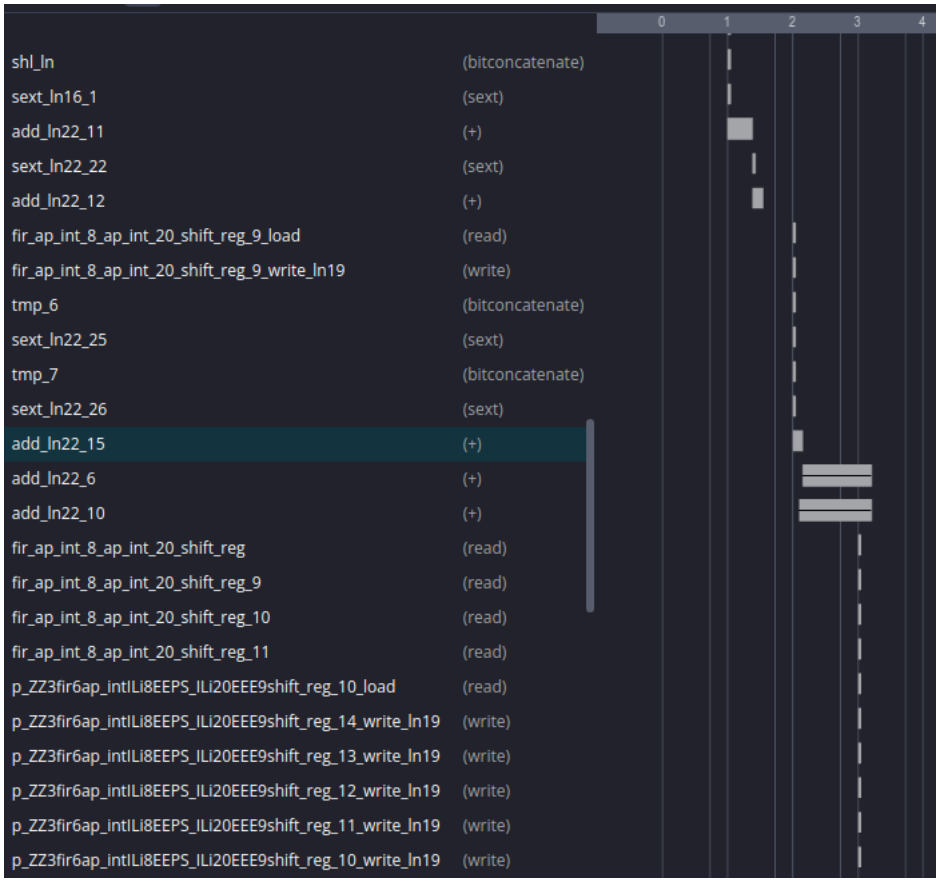

Figure 10: Schedule of the pipelined FIR component (part 3)

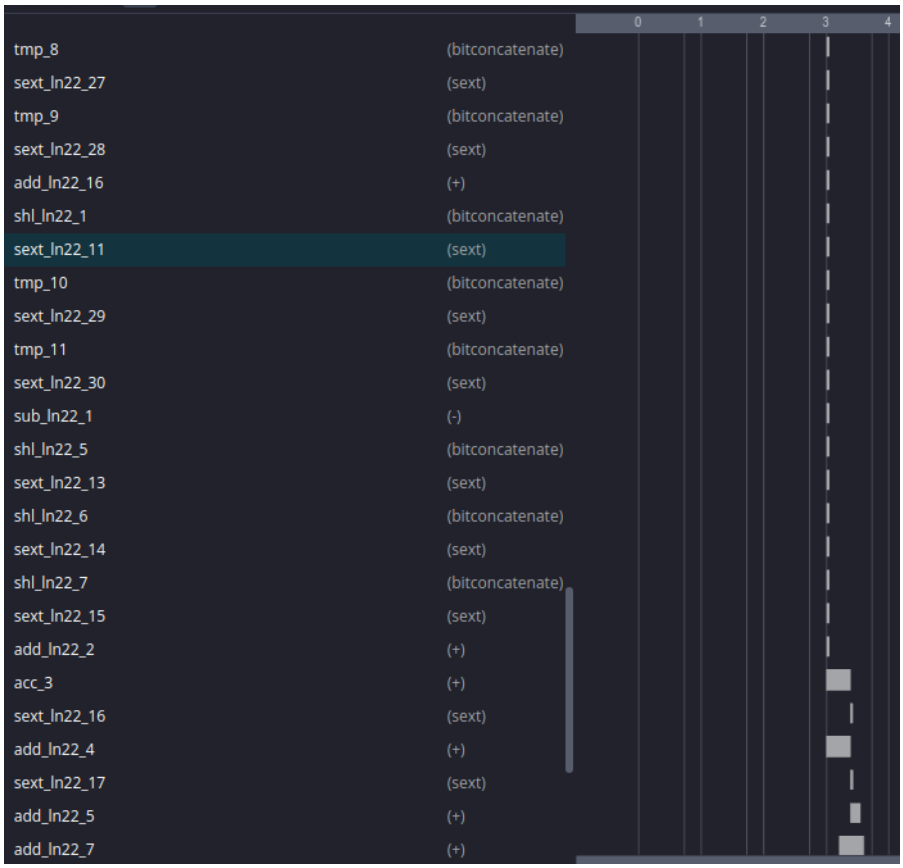

Figure 11: Schedule of the pipelined FIR component (part 4)

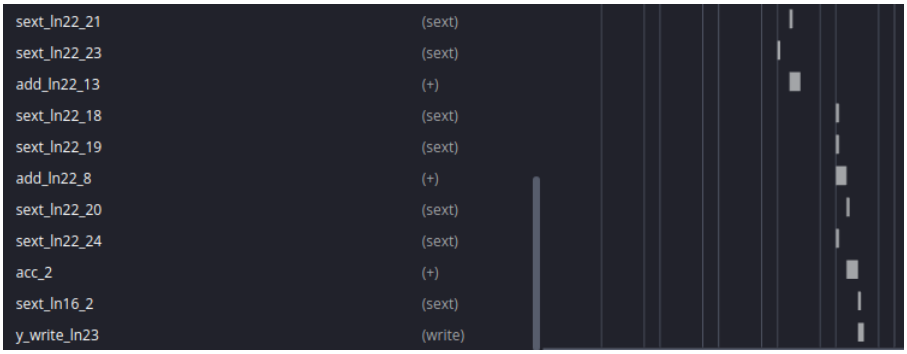

Figure 12: Schedule of the pipelined FIR component (part 5)

The steps of the schedule are further detailed on Verbatim 1 to highlight what is computed from the $x0$ to the $x15$ inputs. The dotted lines mark the separation between the FSM states. The numbers in the left column identify

the 15 adders involved in the sum reduction.

On the first line (shl_ln22_2), $2 * x0$ is computed from a shift left of the $x0$ input. On the second line (shl_ln22_3), $4 * x1$ is computed from a shift of two positions on the left of $x1$. On the third line (add_ln_22), the sum $2x0 + 4x1$ is computed.

```
   shl_ln22_2 : 2x0
   shl_ln22_3 : 4x1
1: add_ln_22  : 2x0 + 4x1
2: acc        : (2x0 + 4x1) + 8x2
   tmp_2      : 16x5
   tmp_3      : 2x5
   tmp_4      : 16x6
   tmp_5      : 4x6
   add_ln22_3 : 16x5 + 2x5
   add_ln22_14: 16x6 + 4x6
   mul_ln22   : x7 * 22
   mul_ln22_1 : x8 * 22
3: add_ln22_9 : 18x5 + 20x6                                        FSM 1
-----------------------------------------------------------------------
   tmp        : 16x3
   tmp_1      : 4x3
   sub_ln22   : 16x3 - 4x3
   shl_ln     : 16x4
4: add_ln22_11: ((2x0 + 4x1) + 8x2) + 12x3
5: add_ln22_12: (((2x0 + 4x1) + 8x2) + 12x3) + 16x4                FSM 2
-----------------------------------------------------------------------
   tmp_6      : 16x9
   tmp_7      : 4x9
   add_ln22_15: 16x9 + 4x9
6: add_ln22_6 : 22x8 + 20x9
7: add_ln22_10: (18x5 + 20x6) + 22x7                               FSM 3
-----------------------------------------------------------------------
   tmp_8      : 16x10
   tmp_9      : 2x10
   add_ln22_16: 16x10 + 2x10
   shl_ln22_1 : 16x11
   tmp_10     : 16x12
   tmp_11     : 4x12
   sub_ln22_1 : 16x12 - 4x12
   shl_ln22_5 : 8x13
   shl_ln22_6 : 4x14
   shl_ln22_7 : 2x15
8: add_ln22_2 : 8x13 + 2x15
9: acc_3      : 4x14 + (8x13 + 2x15)
10:add_ln22_4 : 16x11 + 12x12
11:add_ln22_5 : (16x11 + 12x12) + (4x14 + (8x13 + 2x15))
12:add_ln22_7 : (22x8 + 20x9) + 18x10
13:add_ln22_13: ((((2x0 + 4x1) + 8x2) + 12x3) + 16x4)
              + ((18x5 + 20x6) + 22x7)                             FSM 4
```

```
----------------------------------------------------------------------
14:add_ln22_8 : ((22x8 + 20x9) + 18x10)                          FSM 5
              + ((16x11 + 12x12) + (4x14 + (8x13 + 2x15)))
15:acc_2      : (((((2x0 + 4x1) + 8x2) + 12x3) + 16x4)
              + ((18x5 + 20x6) + 22x7)) + (((22x8 + 20x9) + 18x10)
              + ((16x11 + 12x12) + (4x14 + (8x13 + 2x15))))
y             :  2x0 +  4x1 +  8x2  + 12x3  + 16x4  + 18x5 + 20x6 + 22x7
              + 22x8 + 20x9 + 18x10 + 16x11 + 12x12 + 8x13 + 4x14 + 2x15
----------------------------------------------------------------------
```

Verbatim 1: computation of the dot product

Figure 13 shows an interesting detail in the schedule. The MAC_LOOP computations read the $shift\_reg$ variable before the shift. The synthesizer has removed the dependency between the MAC_LOOP and the SHIFT_LOOP. shl_ln22_2 computes $x*2$ with x being the new input, rather than $shift\_reg[0]*2$ after shift.

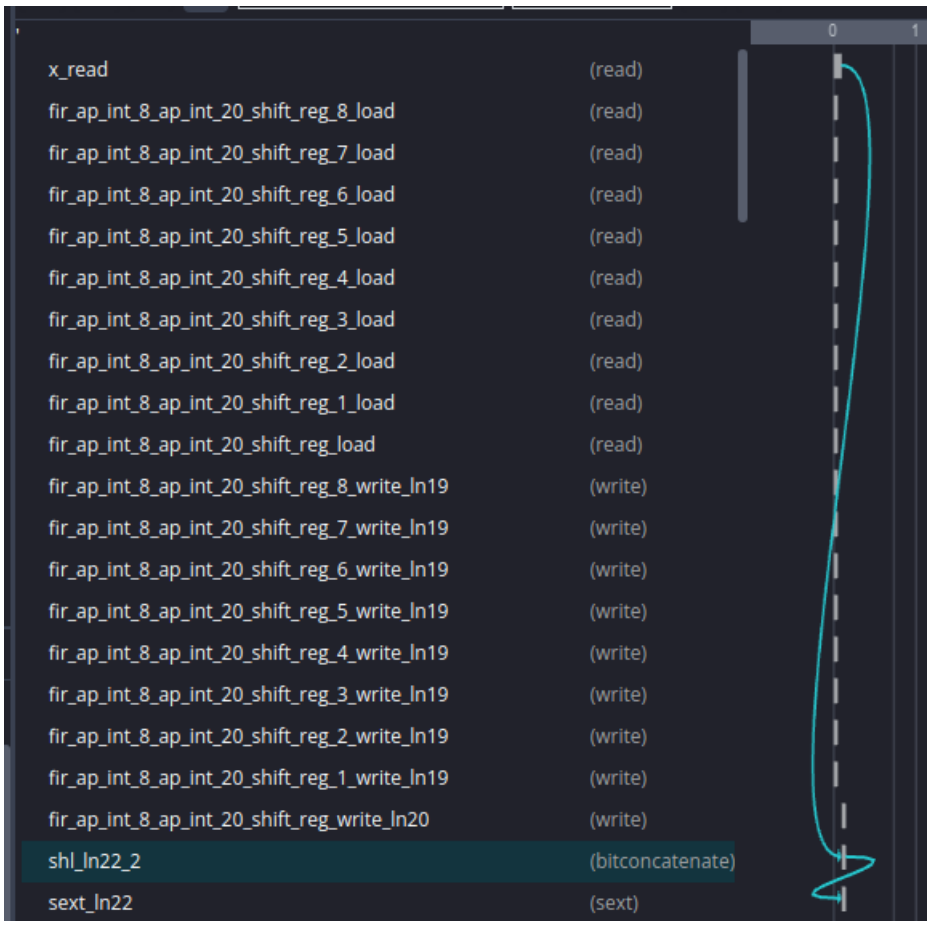

Figure 13: The MAC_LOOP is independent of the SHIFT_LOOP

Figure 14 shows the result of the synthesis. The full run is completed in five cycles (50 ns; the Vitis latency is four cycles or 40ns, i.e. one cycle less than the completion time). A new run can start every cycle (Initiation Interval II=1). The implementation uses two DSPs (two multipliers), 346 FFs and 385 LUTs. There are more FFs because of the FSM inter-stage savings, and more LUTs because of the reduction tree of adders.

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
fir	4	40.000	1	yes	0	2	346	385	0

Figure 14: Synthesis resources of the pipelined FIR component

Figure 15 shows that the timing estimation is 5.944 ns plus 2.70 ns of uncertainty, i.e. an estimated cycle duration of 8.644 ns.

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	5.944 ns	2.70 ns

Figure 15: Synthesis timing estimation of the pipelined FIR component

The Vivado construction is the same as the one shown on figure 6.

Figure 16 shows the resources used in the FPGA to implement the whole system.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	570	0	0	53200	1.07
LUT as Logic	568	0	0	53200	1.07
LUT as Memory	2	0	0	17400	0.01
LUT as Distributed RAM	0	0			
LUT as Shift Register	2	0			
Slice Registers	767	0	0	106400	0.72
Register as Flip Flop	767	0	0	106400	0.72
Register as Latch	0	0	0	106400	0.00
F7 Muxes	1	0	0	26600	<0.01
F8 Muxes	0	0	0	13300	0.00

Figure 16: The FPGA resources used to implement the system including the pipelined FIR component

4 Optimizing to further reduce the latency

There are two main optimizations which can be applied to enhance the FIR IP.

The first one concerns the critical path. From the way the synthesizer implemented the reduction, we can push the idea a little further by removing the

DSP multipliers and compute everything with shifts and adds. This will reduce the latency to two cycles (there are five levels of additions and they fit in two FPGA cycles).

The second improvement concerns the size of the data. As the h values are constants, we can further reduce the width of the intermediate computations, which saves FFs, i.e. area (and probably power, but this is difficult to measure on an FPGA, even though Xilinx provides estimators).

The synthesizer is forced to squeeze the computation within two cycles with the HLS LATENCY MAX=1 pragma (LATENCY=1 cycle means that the FIR IP completes in two cycles). As the synthesizer does not complain, it means that the constraint is satisfied.

Listing 3 shows the improved C code (file *fir_reduction_fast.cpp*).

Listing 3: An optimized FIR filter IP

```
#include "ap_int.h"
#define N 16
void fir(
  ap_int<8>   x,
  ap_int<16> *y){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE s_axilite port=y
#pragma HLS PIPELINE
#pragma HLS LATENCY MAX=1
  static ap_int<8> shift_reg[N] = {0};
#pragma HLS ARRAY_PARTITION variable=shift_reg complete
  ap_int< 9> srh0;
  ap_int<10> srh1;
  ap_int<11> srh2;
  ap_int<12> srh3;
  ap_int<10> srh3a;
  ap_int<11> srh3b;
  ap_int<12> srh4;
  ap_int<13> srh5;
  ap_int< 9> srh5a;
  ap_int<12> srh5b;
  ap_int<13> srh6;
  ap_int<10> srh6a;
  ap_int<12> srh6b;
  ap_int<13> srh7;
  ap_int< 9> srh7a;
  ap_int<10> srh7b;
  ap_int<12> srh7c;
  ap_int<11> srh7ab;
  ap_int<13> srh8;
  ap_int< 9> srh8a;
  ap_int<10> srh8b;
  ap_int<12> srh8c;
  ap_int<11> srh8ab;
  ap_int<13> srh9;
  ap_int<10> srh9a;
  ap_int<12> srh9b;
  ap_int<13> srh10;
  ap_int< 9> srh10a;
  ap_int<12> srh10b;
  ap_int<12> srh11;
  ap_int<12> srh12;
  ap_int<10> srh12a;
```

```
ap_int<11> srh12b;
ap_int<11> srh13;
ap_int<10> srh14;
ap_int< 9> srh15;
//----------------
ap_int<11> srh0_1;
ap_int<13> srh2_4;
ap_int<13> srh3_12;
ap_int<14> srh5_6;
ap_int<14> srh7_8;
ap_int<14> srh9_10;
ap_int<13> srh11_13;
ap_int<11> srh14_15;
//-----------------
ap_int<13> srh0_1_2_4;
ap_int<13> srh11_13_14_15;
ap_int<15> srh5_6_9_10;
ap_int<15> srh3_7_8_12;
//-----------------
ap_int<14> srh0_1_2_4_11_13_14_15;
ap_int<16> srh3_5_6_7_8_9_10_12;
//-----------------
srh0     = ((ap_int< 9>)x)<<1;//2x0
srh1     = ((ap_int<10>)(shift_reg[ 0]))<<2;//4x1
srh2     = ((ap_int<11>)(shift_reg[ 1]))<<3;//8x2
srh3a    = ((ap_int<10>)(shift_reg[ 2]))<<2;//4x3
srh3b    = ((ap_int<11>)(shift_reg[ 2]))<<3;//8x3
srh3     =  (ap_int<12>)srh3a + (ap_int<12>)srh3b;//12x3
srh4     = ((ap_int<12>)(shift_reg[ 3]))<<4;//16x4
srh5a    = ((ap_int< 9>)(shift_reg[ 4]))<<1;//2x5
srh5b    = ((ap_int<12>)(shift_reg[ 4]))<<4;//16x5
srh5     =  (ap_int<13>)srh5a + (ap_int<13>)srh5b;//18x5
srh6a    = ((ap_int<10>)(shift_reg[ 5]))<<2;//4x6
srh6b    = ((ap_int<12>)(shift_reg[ 5]))<<4;//16x6
srh6     =  (ap_int<13>)srh6a + (ap_int<13>)srh6b;//20x6
srh7a    = ((ap_int< 9>)(shift_reg[ 6]))<<1;//2x7
srh7b    = ((ap_int<10>)(shift_reg[ 6]))<<2;//4x7
srh7c    = ((ap_int<12>)(shift_reg[ 6]))<<4;//16x7
srh7ab   =  (ap_int<11>)srh7a + (ap_int<11>)srh7b;//6x7
srh7     =  (ap_int<13>)srh7ab + (ap_int<13>)srh7c;//22x7
srh8a    = ((ap_int< 9>)(shift_reg[ 7]))<<1;//2x8
srh8b    = ((ap_int<10>)(shift_reg[ 7]))<<2;//4x8
srh8c    = ((ap_int<12>)(shift_reg[ 7]))<<4;//16x8
srh8ab   =  (ap_int<11>)srh8a + (ap_int<11>)srh8b;//6x8
srh8     =  (ap_int<13>)srh8ab + (ap_int<13>)srh8c;//22x8
srh9a    = ((ap_int<10>)(shift_reg[ 8]))<<2;//4x9
srh9b    = ((ap_int<12>)(shift_reg[ 8]))<<4;//16x9
srh9     =  (ap_int<13>)srh9a + (ap_int<13>)srh9b;//20x9
srh10a   = ((ap_int< 9>)(shift_reg[ 9]))<<1;//2x10
srh10b   = ((ap_int<12>)(shift_reg[ 9]))<<4;//16x10
srh10    =  (ap_int<13>)srh10a + (ap_int<13>)srh10b;//18x10
srh11    = ((ap_int<12>)(shift_reg[10]))<<4;//16x11
srh12a   = ((ap_int<10>)(shift_reg[11]))<<2;//4x12
srh12b   = ((ap_int<11>)(shift_reg[11]))<<3;//8x12
srh12    =  (ap_int<12>)srh12a + (ap_int<12>)srh12b;//12x12
srh13    = ((ap_int<11>)(shift_reg[12]))<<3;//8x13
srh14    = ((ap_int<10>)(shift_reg[13]))<<2;//4x14
srh15    = ((ap_int< 9>)(shift_reg[14]))<<1;//2x15
//-------------------------
srh0_1    = (ap_int<11>)srh0   + (ap_int<11>)srh1;
srh2_4    = (ap_int<13>)srh2   + (ap_int<13>)srh4;
srh3_12   = (ap_int<13>)srh3   + (ap_int<13>)srh12;
```

```
  srh5_6    = (ap_int<14>)srh5   + (ap_int<14>)srh6;
  srh7_8    = (ap_int<14>)srh7   + (ap_int<14>)srh8;
  srh9_10   = (ap_int<14>)srh9   + (ap_int<14>)srh10;
  srh11_13  = (ap_int<13>)srh11  + (ap_int<13>)srh13;
  srh14_15  = (ap_int<11>)srh14  + (ap_int<11>)srh15;
  //-------------------------
  srh0_1_2_4 = (ap_int<13>)srh0_1 + (ap_int<13>)srh2_4;
  srh11_13_14_15 = (ap_int<13>)srh11_13 + (ap_int<13>)srh14_15;
  srh5_6_9_10 = (ap_int<15>)srh5_6 + (ap_int<15>)srh9_10;
  srh3_7_8_12 = (ap_int<15>)srh3_12 + (ap_int<15>)srh7_8;
  //----------------------------
  srh0_1_2_4_11_13_14_15 = (ap_int<14>)srh0_1_2_4 + (ap_int<14>)
      srh11_13_14_15;
  srh3_5_6_7_8_9_10_12 = (ap_int<16>)srh3_7_8_12 + (ap_int<16>)
      srh5_6_9_10;
  //-----------------
  *y = (ap_int<16>)srh0_1_2_4_11_13_14_15 + (ap_int<16>)
      srh3_5_6_7_8_9_10_12;
  shift_reg[15] = shift_reg[14];
  shift_reg[14] = shift_reg[13];
  shift_reg[13] = shift_reg[12];
  shift_reg[12] = shift_reg[11];
  shift_reg[11] = shift_reg[10];
  shift_reg[10] = shift_reg[9];
  shift_reg[9] = shift_reg[8];
  shift_reg[8] = shift_reg[7];
  shift_reg[7] = shift_reg[6];
  shift_reg[6] = shift_reg[5];
  shift_reg[5] = shift_reg[4];
  shift_reg[4] = shift_reg[3];
  shift_reg[3] = shift_reg[2];
  shift_reg[2] = shift_reg[1];
  shift_reg[1] = shift_reg[0];
  shift_reg[0] = x;
}
```

The intermediate variables srh have the size required by the product of the x maximum value (0xff) and the matching h constant (the product should have enough bits to show the correct sign). For example, $srh0$ receives the product of x by 2, which fits on a 9-bit word.

Variable $srha\_b...\_d$ is the sum $srh\_a + srh\_b + ... + srh\_d$. For example, $srh0\_1\_2\_4$ is the sum $srh\_0 + srh\_1 + srh\_2 + srh\_4$.

The schedule of the run is similar to the one of the preceding implementation, with two cycles instead of five. It is shown on figures 17 to 21.

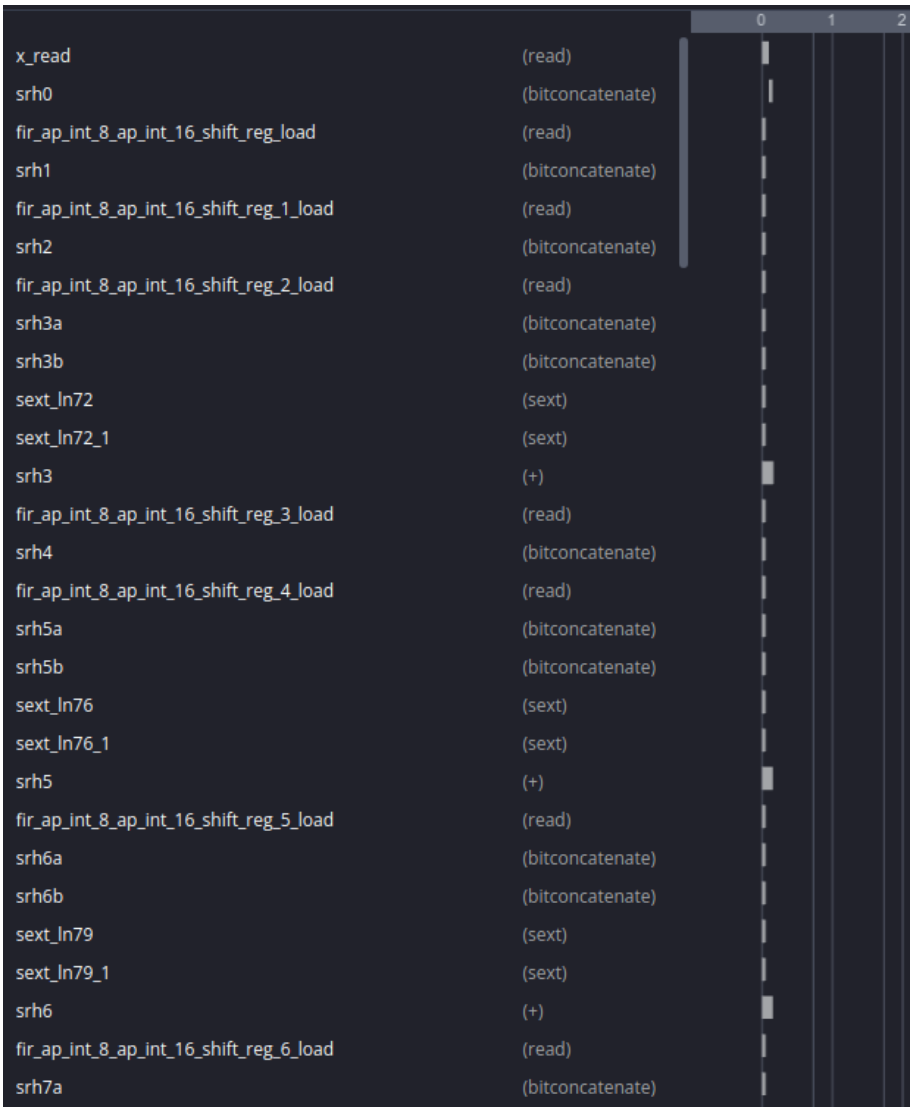

Figure 17: Schedule of the optimized FIR component (part 1)

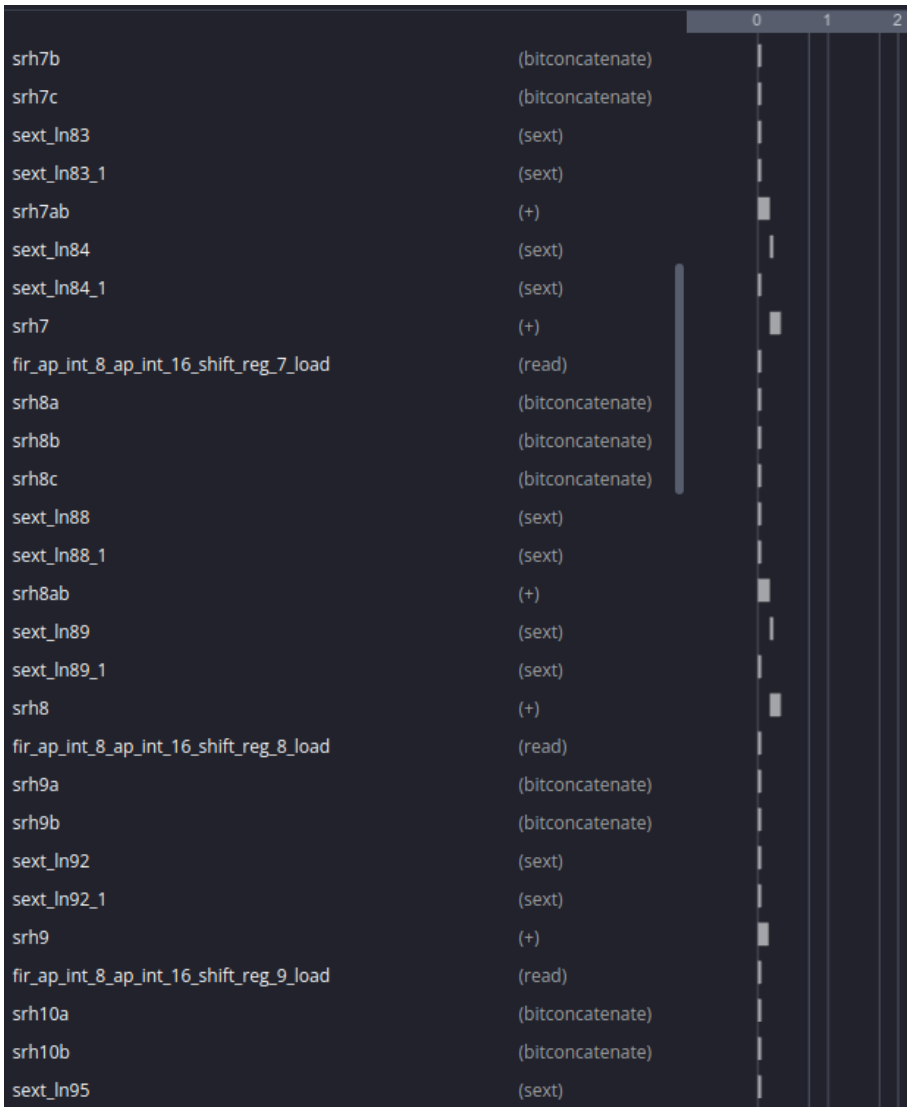

Figure 18: Schedule of the optimized FIR component (part 2)

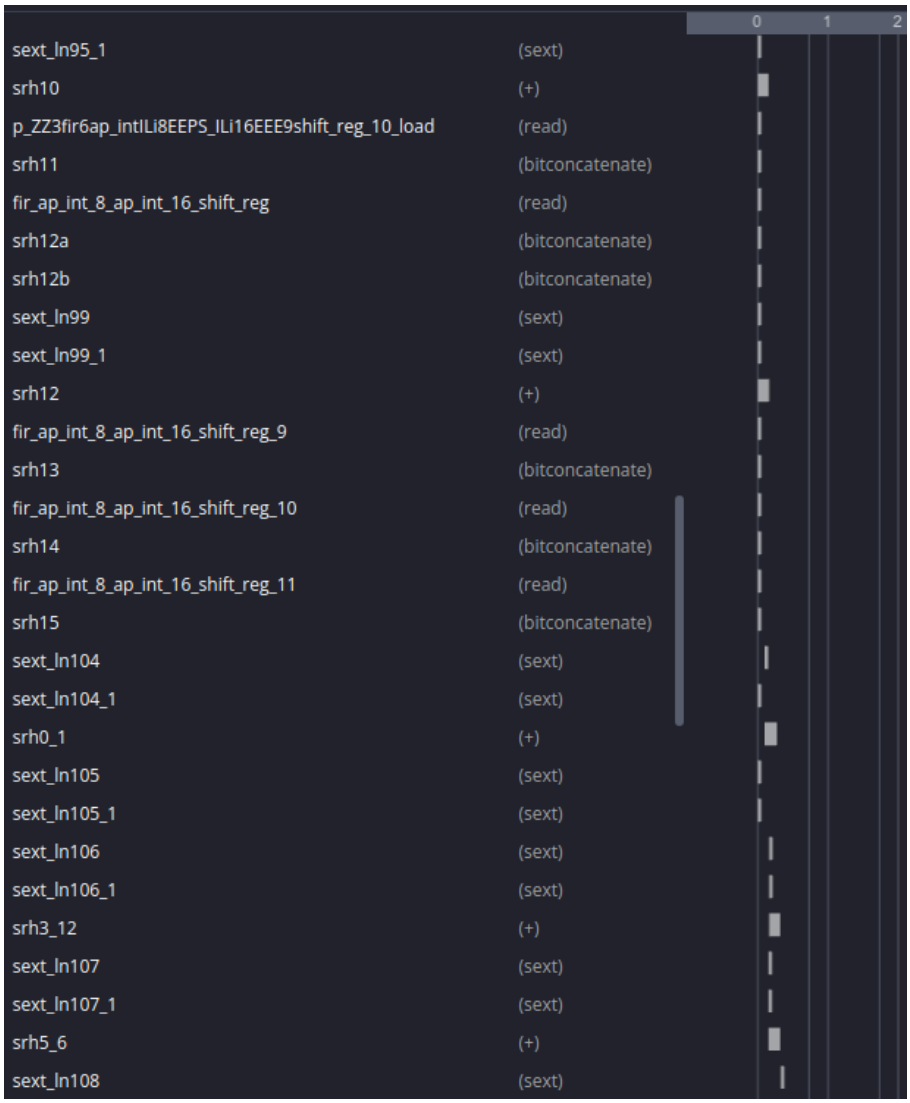

Figure 19: Schedule of the optimized FIR component (part 3)

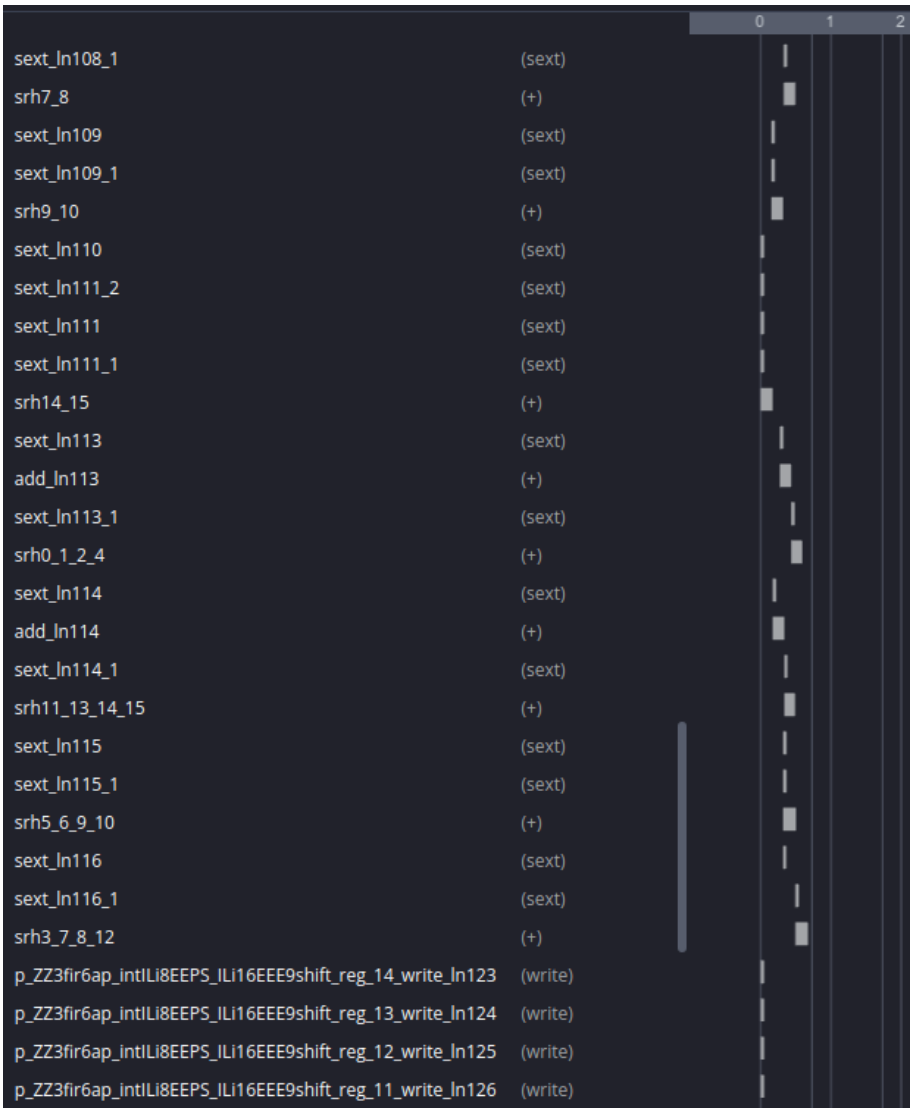

Figure 20: Schedule of the optimized FIR component (part 4)

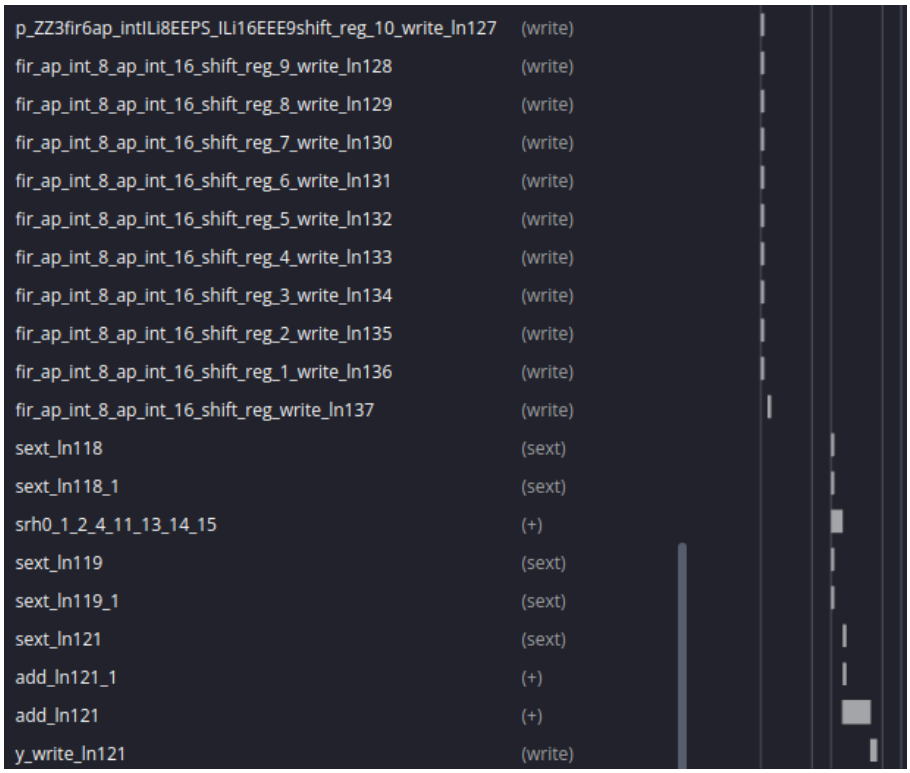

Figure 21: Schedule of the optimized FIR component (part 5)

Figure 22 shows the result of the synthesis. The full run is completed in two cycles (20 ns; the Vitis latency is one cycle or 10ns, i.e. one cycle less than the completion time). A new run can start every cycle (Initiation Interval II=1). The implementation uses no DSP, 246 FFs (100 less) and 452 LUTs (67 more).

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
fir	1	10.000	1	yes	0	0	246	452	0

Figure 22: Synthesis resources of the optimized FIR component

Figure 23 shows that the timing estimation is 6.769 ns plus 2.70 ns of uncertainty, i.e. an estimated cycle duration of 9.469 ns.

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	6.769 ns	2.70 ns

Figure 23: Synthesis timing estimation of the optimized FIR component

5 The limits of HLS

An interesting question is: could the synthesizer achieve the same latency from a purely HLS code ? I tried the Vitis synthesizer on a second version of the optimized filter code shown on listing 4 (file *fir_reduction_fast_2.cpp*).

Listing 4: An optimized FIR filter IP

```
#include "ap_int.h"
#define N 16
const ap_int<8> h[N] = {
   2,  4,  8, 12, 16, 18, 20, 22,
  22, 20, 18, 16, 12,  8,  4,  2
};
void fir(
  ap_int<8>   x,
  ap_int<20> *y){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE s_axilite port=y
#pragma HLS PIPELINE II=1
#pragma HLS LATENCY max=1
  static ap_int<8> shift_reg[N] = {0};
#pragma HLS ARRAY_PARTITION variable=shift_reg complete
  SHIFT_LOOP: for (int i = N-1; i > 0; i--)
    shift_reg[i] = shift_reg[i-1];
  shift_reg[0] = x;
  ap_int<20> p;
#pragma HLS BIND_OP variable=p op=mul impl=fabric
  ap_int<20> acc = 0;
  MAC_LOOP: for (int i = 0; i < N; i++) {
    p = shift_reg[i] * h[i];
    acc += p;
  }
  *y = acc;
}
```

The HLS BIND_OP pragma forces the synthesizer to use LUTs rather than DSPs (this is what *fabric* means) to implement the *mul* operators used in the computation of variable p, i.e. shifts and adds.

However, the synthesis raises a timing violation as figure 24 shows.

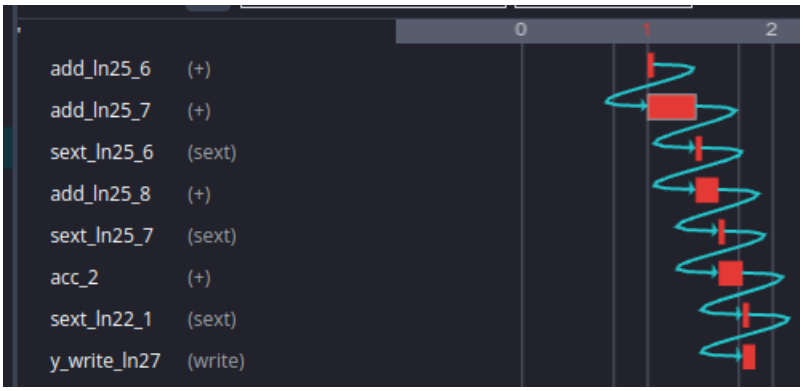

Figure 24: Timing violation

So, the answer seems to be no.

But, the code on listing 5 is OK with the synthesis, even though I had to use a few tricks to fit it in the two cycles latency constraint (file *fir_reduction_fast_3.cpp*).

Listing 5: An optimized FIR filter IP

```
#include "ap_int.h"
#define N 16
const ap_int<8> h[N] = {
   2,  4,  8, 12, 16, 18, 20, 22,
  22, 20, 18, 16, 12,  8,  4,  2
};
void fir(
  ap_int<8>   x,
  ap_int<20> *y){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE s_axilite port=y
#pragma HLS PIPELINE II=1
#pragma HLS LATENCY max=1
  static ap_int<8> shift_reg[N] = {0};
#pragma HLS ARRAY_PARTITION variable=shift_reg complete
  SHIFT_LOOP: for (int i = N-1; i > 0; i--)
    shift_reg[i] = shift_reg[i-1];
  shift_reg[0] = x;
  ap_int<20> p = ((ap_int<9>)shift_reg[0] << 1) + ((ap_int<9>)
      shift_reg[15] << 1);
#pragma HLS BIND_OP variable=p op=mul impl=fabric
  ap_int<20> acc = p;
  MAC_LOOP: for (int i = 1; i < N-1; i++) {
    switch(h[i]){
      case  4: p =  (ap_int<10>)shift_reg[i] << 2; break;
      case  8: p =  (ap_int<11>)shift_reg[i] << 3; break;
      case 16: p =  (ap_int<12>)shift_reg[i] << 4; break;
      case 12: p = ((ap_int<10>)shift_reg[i] << 2) +
                   ((ap_int<11>)shift_reg[i] << 3); break;
      case 18: p = ((ap_int< 9>)shift_reg[i] << 1) +
                   ((ap_int<12>)shift_reg[i] << 4); break;
      case 20: p = ((ap_int<10>)shift_reg[i] << 2) +
                   ((ap_int<12>)shift_reg[i] << 4); break;
      case 22: p = ((ap_int< 9>)shift_reg[i] << 1) +
                   ((ap_int<10>)shift_reg[i] << 2) +
                   ((ap_int<12>)shift_reg[i] << 4); break;
    }
    acc += p;
  }
  *y = acc;
}
```

Figure 25 shows the result of the synthesis. The 2 cycles latency constraint is fulfilled. The implementation uses no DSP, 249 FFs (3 more) and 460 LUTs (8 more).

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
fir	1	10.000	1	yes	0	0	249	460	0

Figure 25: Synthesis resources of the optimized FIR component

6 Testing

6.1 Simulating

The three FIR filter IPs can all be tested with the same testbench presented on listing 6 (file *tb_fir.cpp*):

Listing 6: A FIR filter IP testbench

```
#include <stdio.h>
#include <stdlib.h>
#include "ap_int.h"
#define SAMPLES 600
void fir(
  ap_int<8>   x,
  ap_int<20> *y);//ap_int<16> for the optimized reduction
int main () {
  FILE      *fin;
  FILE      *fout;
  ap_int<20> y;//ap_int<16> for the optimized reduction
  ap_int<8>  x;
  int        signal;
  fin  = fopen("/path/to/input.dat","r");
  fout = fopen("/path/to/output.dat","w");
  for (unsigned int i = 0; i < SAMPLES; i++) {
    fscanf(fin, "%d", &signal);
    x = (ap_int<8>)signal;
    fir(x, &y);
    fprintf(fout,"%d\n", (int)y);
  }
  fclose(fout);
  fclose(fin);
  printf ("Comparing against output data \n");
  if (system("diff -w /path/to/output.dat /path/to/out.gold.dat"
      )) {
    fprintf(stdout, "
        ******************************************\n");
    fprintf(stdout, "FAIL: Output DOES NOT match the golden
        output\n");
    fprintf(stdout, "
        ******************************************\n");
    return 1;
  }
  else {
    fprintf(stdout, "
        ******************************************\n");
    fprintf(stdout, "PASS: The output matches the golden output
        !\n");
    fprintf(stdout, "
        ******************************************\n");
    return 0;
  }
}
```

The sample sequence is stored in the *input.dat* file and the filtered sequence is output to the *output.dat* file. Then, the *output.dat* file is compared to a *out.gold.dat* file containing the expected filtered sequence.

The filtered sequence is built from successive calls to the *fir* function, successively introducing the values of the input sequence.

This testbench program is to be run on a processor simulating the FIR IP. In

this case, the compiler does not take into account the HLS pragmas and simply runs the code of the *fir* function as any C program (the programmer should be aware that the *ap_int*<width> types are transformed in C types; for example *ap_uint*<5> type is extended to a *char* or *int8_t* standard C type).

6.2 Running on the FPGA

To test the *fir* function on the FPGA, there are two parts. The first one is the FIR IP, built as a hardware component through the synthesis. The second one is a new *main* function to drive the FIR IP and mimicking the testbench one, placed in a *helloworld.c* file and shown on listing 7.

Listing 7: A FIR filter IP helloworld driver

```
#include "xparameters.h"
#include "xfir.h"
#include <stdio.h>
#include <stdlib.h>
#define SAMPLES 600
#define MASK 0xfff00000  //0xffff0000 for the optimized reduction
#define SIGN(v) (v&0x00080000)  //0x00008000 for the optimized reduction
int input[SAMPLES] = {
    #include "input_init_ram.dat"
};
int golden[SAMPLES] = {
    #include "output_init_ram.gold.dat"
};
int output[SAMPLES];
int convert(u32 v){
  if (SIGN(v)) return v|MASK;
  else return v;
}
int main(){
  XFir_Config *cfg_ptr;
  XFir         ip;
  cfg_ptr = XFir_LookupConfig(XPAR_FIR_0_BASEADDR);
  XFir_CfgInitialize(&ip, cfg_ptr);
  int signal;
  printf("Starting FIR processing...\n");
  for (int i = 0; i < SAMPLES; i++){
    signal = input[i];
    XFir_Set_x(&ip, (int8_t) signal);
    XFir_Start(&ip);
    while (!XFir_IsDone(&ip));
    output[i] = convert(XFir_Get_y(&ip));
  }
  printf("Comparing with golden output...\n");
  for (unsigned int i = 0; i < SAMPLES; i++){
    if (output[i] != golden[i]){
      printf("FAIL: Output DOES NOT match golden output\n");
      return 1;
    }
    else{
      printf("PASS: Output matches golden output\r\n");
      return 0;
    }
  }
}
```

The *xparameters.h* file is generated by the synthesis. It serves to define the XPAR constants like XPAR_FIR_0_BASEADDR which is the memory mapped address of the FIR IP.

File *xfir.h* is also generated by the synthesis. It defines the XFir_ functions to drive the FIR IP, like XFir_CfgInitialize which is called to initialize (or reset) the FIR IP.

Before running the helloworld.c code, the data files must be copied in the *hello_world/src* folder (*input_init_ram.dat* and *output_init_ram.gold.dat*).

The *main* function is to be run on a processor attached to the *fir* component. It connects to the FIR IP (XFir_LookupConfig and XFir_CfgInitialize).

Then, it runs a loop which launches the FIR IP for each input sample. The sample is sent to the FIR IP (XFir_Set_x). Then the FIR IP is started (XFir_Start). The helloworld code waits for the FIR IP to finish (XFir_IsDone). Then it gets the output (XFir_Get_y).

The value returned by function XFir_Get_y is a 20-bit signed value extended to a 32-bit unsigned one (the result of the XFir_Get_y function is of type *u32* which is unsigned). Hence, a conversion function sets bits 21-31 if bit 20 is a 1, to transform the unsigned value into a signed one (bits 17-31 if bit 16 is a 1 for the optimized reduction returning a 16-bit result).

7 Discussion and conclusion

HLS is today still considered as a research or academic tool, not a professional way to implement IPs. But, this FIR example shows that synthesizers are very efficient.

HLS development saves a lot of time. The C code for the non optimized pipelined FIR IP has less than 25 lines when the Verilog code is distributed on 4 files and 1257 lines (even though the Verilog code contains many lines related to the Xilinx FPGA organization). The highly optimized C code has 138 lines. The synthesizer takes care of the FSM, of the pipelining and of the critical path.

HLS also saves time in debugging. The classic way to debug an IP when implemented directly in Verilog or VHDL is to go to timelines. Even though you can do the same in HLS from the generated Verilog, most of the time it is not necessary because, what runs in simulation (with the testbench *main* function) runs on the board (with the *helloworld* driver). If not, in HLS we use programmer's classic debugging features, either printf equivalent (during the run on the FPGA, save the values to be examined to memory and print them in the *helloworld* driver), or even the classic *gdb* application.

HLS in a few years will become as natural for hardware designers than compilers are for software developers. A long time ago, assembly language programmers were sceptical about compilers. They would claim that compiled code was less efficient than their own hand written assembly codes. Today, everybody agrees that for a large program, no programmer can compete with an optimized compilation level.

Still, hardware designers pretend that they need the Verilog code and the timelines. Same analogy with high level languages plus compiler. Nobody even looks at the assembly code generated by the compiler. Nobody debugs at the assembly level language (well, nearly nobody: sometimes people want to check

if they can optimize further, or at least see how the compiler was smart). One day, nobody will look at timelines anymore.

Verilog programmers are also convinced that HLS does not allow timing optimizations. The example in section 5 shows that HLS is quite smart, even though the programmer has to be smart too to achieve an optimal timing.

However, on small IPs like the FIR one, a Verilog programmer or an architect programming in C can build code which the HLS tools cannot fully infer. The optimized reduction implementation presented in section 4 and the HLS version in section 5 illustrates this. But on complex designs, like a full processor, HLS builds better RTL than synthesis from Verilog or VHDL code.

The Xilinx synthesizer is oriented towards Xilinx FPGAs rather than ASICs. We need an open source HLS tool as free of use and as standardized than the *gcc* compiler.

However, when the target is a first quick IP implementation on an FPGA to test a design, HLS is a must.