**Nicholas Olexa**

**1/20/2023**

**3/8/2023 (updated)**

**4/6/2023 (updated)**

**FIR filter design**

In order to gain more control over transmit bandwidth than an Rf analog filter would allow, a digital filter is necessary. When constructing a digital filter, there are two options: a finite impulse response filter and an infinite impulse response filter. The former option is the simplest choice, due to the lack of feedback necessary for operation. The simplest way to implement an FIR filter is through convolution between the signal that needs to be transmitted and the filter kernel, and this is the technique that was chosen for this HDL module.

The DIGITALS system is operating in a unique mode, having ‘8’ separate signal generators (essentially, look up tables) being run out of phase with each-other to generate the communication signals. When these eight generated signals are *spliced* together, the desired waveform is constructed. If convolution were to be performed on this signal, the convolution block would need to be running at 8x the sample generator clock (8 x 307.2 MHz) in order to handle the effective sample rate of the generated signal. This of course is not feasible. The better approach is to pipeline the process by allowing 8 separate convolution processes to occur in parallel, one for each signal generator (or LUT). This reduces the local sample rate (per lane) to be equal to the generator clock (307.2 MHz), essentially down-sampling the communication signal. In order for this pipelined filtering technique to work, the FIR filter kernel itself must also be down sampled by a factor of 8, as if each lane were only interacting with every eighth sample of the original filter kernel. In a similar sense, the filter kernel in each lane of the pipeline will be interacting with every eighth sample of the communication signal. Upon reconstruction, the filtered waveform will be equivalent to a non-down-sampled signal being convolved with a non-down-sampled filter kernel. The sample rate of the kernel must match that of the signal for them to be placed on the same time scale, where one sample represents the same period of time between the two. The filter kernel, being constructed from Sinc() functions, is limited in its upper cutoff frequency since the kernel must be sampled at the Nyquist rate of twice per period. This means the maximum upper cutoff frequency is theoretically limited to less than 153.6 MHz.

To achieve the desired filtering effect, a special reconstruction algorithm was determined to allow a seamless re-building of the original waveform. The nature of convolution is windowed integration. As one signal is overlapped and shifted over another, the multiply-sum result changes. Since there are eight convolution processes occurring in parallel, the multiply-sum results must be combined in a way as to represent the original signal being convolved with the original filter. In this case, the down-sampling technique is to split each set of eight samples generated by the look-up-tables into eight separate lanes. This way, all information required to reconstruct the net convolution result will be available from instantaneous and previously generated sample data. In the example below, four lanes of traffic are used to show how the algorithm works.

KEY

|  |  |
| --- | --- |
| **co\_LX** | convolution output from lane ‘X’ |
| **conv\_sum\_LX** | reconstructed convolution result from lane ‘X’ |
| **[i]** | data generated in current clock cycle |
| **[i-1]** | data generated in previous clock cycle |

Table 1 – key for convolution reconstruction equations

conv\_sum\_L0 = co\_L0[i] + co\_L1[i-1] + co\_L2[i-1] + co\_L3[i-1] = output\_sample(0, 4, 8, 12, 16, etc…)

conv\_sum\_L1 = co\_L0[i] + co\_L1[i] + co\_L2[i-1] + co\_L3[i-1] = output\_sample(1, 5, 9, 13, 17, etc…)

conv\_sum\_L2 = co\_L0[i] + co\_L1[i] + co\_L2[i] + co\_L3[i-1] = output\_sample(2, 6, 10, 14, 18, etc…)

conv\_sum\_L3 = co\_L0[i] + co\_L1[i] + co\_L2[i] + co\_L3[i] = output\_sample(3, 7, 11, 15, 19, etc…)

Eq. 1:4 – example equation set for 4-lane convolution sum reconstruction.

In the above equations, the blue data represents samples coming into the reconstruction block during the current clock cycle, while the green data represents samples captured in the previous cycle. This simple algorithm shows the nature of the reconstruction process. As the ‘windowing’ function (the function being shifted across the other) shifts across the signal, new multiply-sum data is added to the running summation window (blue), and the old data is shifted out of the running summation window (green). Data is computed from this sum every clock cycle, generating the net convolution result. In this example, the convolution-sum samples from each of the four lanes (conv\_sum\_L0 – conv\_sum\_L3) are the first four respective samples of the convolution result. The FIR block implemented in our system has (8) convolution blocks for the In-phase samples (I), and (8) convolution blocks for the quadrature samples (Q). The I and Q samples are reconstructed separately with their own reconstruction blocks and are packaged together before being sent to the DAC for conversion to the analog domain.

When the user inputs the desired center frequency and bandwidth of the filter, the software will compute the filter coefficients through a simple technique, and then load them into the FIR filter block. The current configuration of the FIR block is to use a 31-sample filter kernel. The effective kernel, then is actually (31 x 8 + 1 = ) 249 samples in length, giving a very clean impulse response. After the loading is completed, data can be transmitted through the block, and the kernel coefficients can be reloaded on the fly. The software takes the given center frequency and bandwidth and calculates the desired ‘corner frequencies’ of the filter. The corner frequencies are translated as the frequency of sinc(at) functions, which generate a window in the frequency domain. The lower corner frequency is the cut-on frequency of the high pass filter, and the higher corner frequency is the cut-off frequency of the low-pass filter. Being ideal rectangular pulses in the frequency domain, the corner frequencies are the transition points where the rectangular pulse starts traveling from ‘1’ to ‘0’, as opposed to a traditional 3dB bandwidth. The sinc(at) functions which represent the time-domain response of the low-pass filter, and the high-pass filter are convolved together, producing a bandpass filter kernel. The basic relationship between a sinc(at) in the time domain and a ***rect(f/a)*** in the frequency domain is given below.

*sinc math*

* sinc(a\*t) = sin(π\*a\*t)/ π\*a\*t
* a = 2 \* fdesired
* effective frequency = ½ \* a
* effective period = 2 / a = 1 / fdesired
* first zero crossing = 1/ a = 1/ fdesired

*Fourier transform*

* sinc(at) -> F(t) -> (1/a) \* rect (f / a)

*rectangular pulse*

* rect(f/a)
* a = pulse width
* T = 2a, f = 1/(2a)

All of the math for generating the filter, as well as simulation for the down-sampled convolution process is captured in python files. For a basic description of what each python file does, check out the README.txt file. Before any VHDL was written, everything was worked out and simulated in python to give a clear picture of what the hardware needed to do. Below in Fig. 1 is the flow diagram for the convolution filter block. Each convolution block (in yellow) is loaded with a copy of the filter kernel (31 samples, all *real* data). The input data from the left-hand side is split into eight individual sample streams and convolved in a continuous fashion with the loaded filter kernel. Data streamed out of the convolution blocks is run through the algorithm in Eq. 1:4 above (albeit, with 8 convolution sums), and finally streamed back into its original format for processing by the next block in the overall system (mapping to JESD in this case). The entire block is run on the same clock that generates the samples, allowing for a straightforward processing technique.

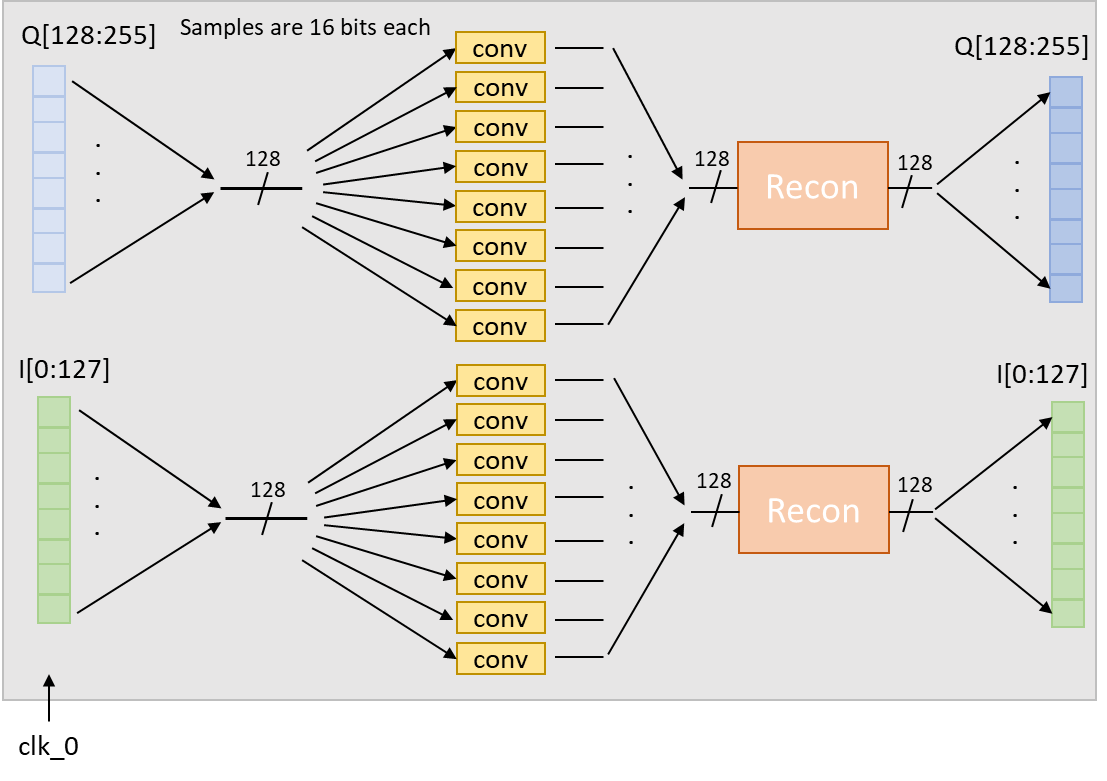


Fig. 1 – convolution filter flow diagram. Iteration 1

**Current System Limitations:**

The convolution filter can NOT handle high frequency signals due to the sample rate limitation. The maximum range of accessible frequency space is fs/ 2 = (307.2e6 \* 8)/2 = 2457.6 MHz / 2 = 1228.8 MHz. Upon down-sampling the BPSK signal, the new maximum range becomes 1228.8 MHz / 8 = 153.6 MHz. This means any bandpass filter must terminate its upper frequency cutoff edge at 153.6 MHz, otherwise it becomes **under-sampled**, meaning it no longer represents its original ***sinc()***response.

**Updated Filter Design:**

To reconcile the max frequency response limitations, a new design was developed. The idea here is to run the convolution filter block at twice the rate of the sample gen clock but split the incoming data into four sample streams instead of eight. In this way, we only need four convolution blocks per I/ Q channel greatly simplifying the hardware. At the same time, the internal sample rate of the convolution blocks is **doubled**, which also doubles the effective max frequency reach of the filter. To achieve this, the input sample stream is split into two by use of a multiplexer. The MUX will switch between the bottom four and top four samples of the incoming sample stream, which is only possible by running the internal clock at double the rate of the sample gen clock.

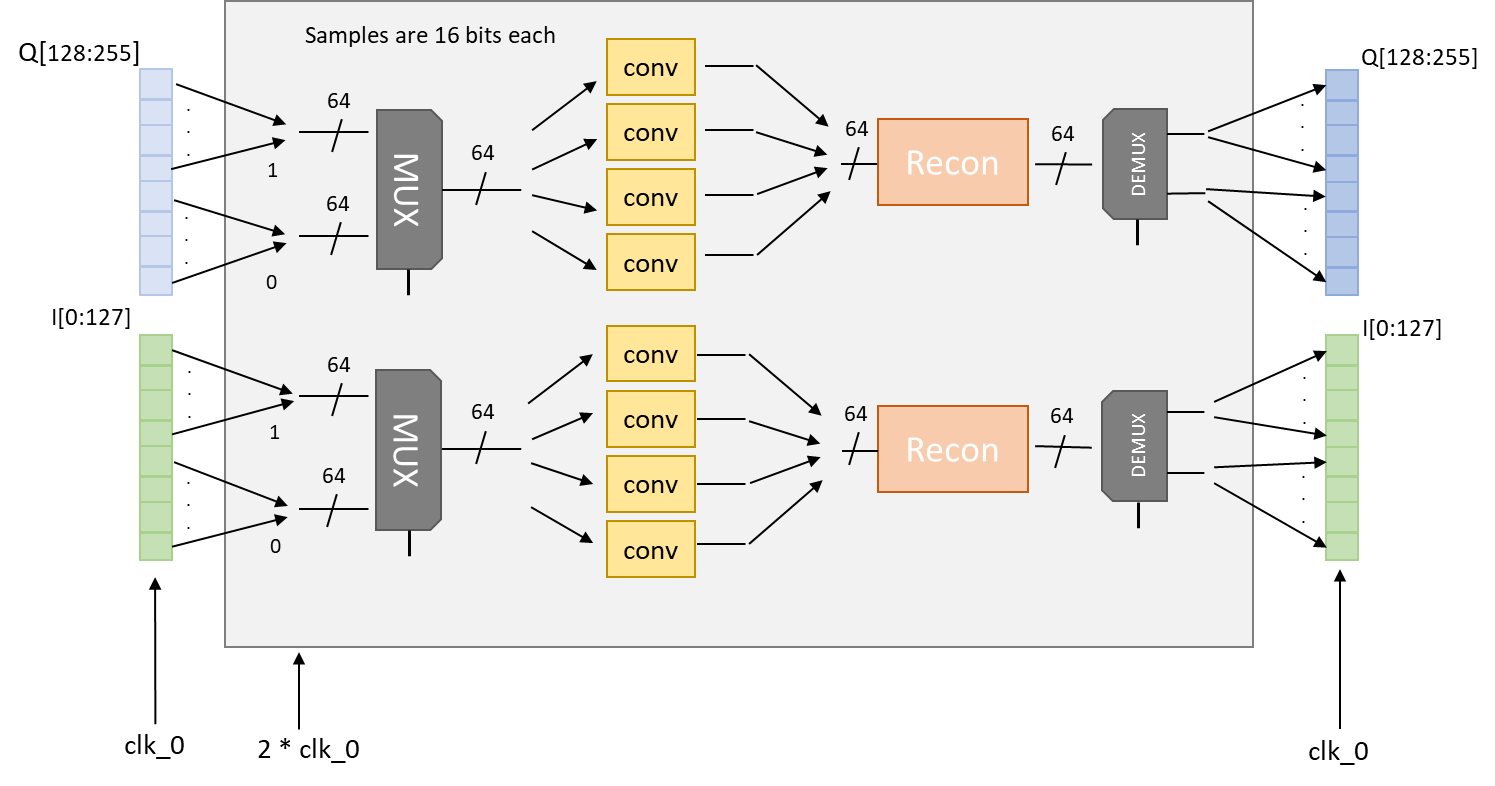


Fig. 2 – Convolution filter flow diagram. Iteration two w/ double bandwidth

Another improvement of this system is the technique used to generate the filter kernel. Kernels are generated directly in the frequency domain given the desired center frequencies and bandwidths. **Multiple** pass bands can be achieved and is a practical use of this filtering scheme due to the increase in useful bandwidth. The frequency step size is a factor of the sample rate and the number of samples in the filter kernel. *(Note: this generation technique has not been fully developed. It does not currently work…)*

N – kernel length

Npos,f = N/2

df = fs / 2 \* Npos,f

With a kernel size of 31 samples and a sample frequency of 307.2MHz (as opposed to 153.6 MHz from iter. 1), the frequency resolution is 9.9 MHz, giving a relatively sharp band-pass response even with bandwidths in the range of 25 MHz.

Shown below are the python simulation outputs using the sinc-function kernel generation, combined with the 4-lane, 2x sample rate filtering technique shown in Fig. 2. Fig. 3 shows part of the input BPSK signal, the ideal filter kernel (512 samples long), and the filtered BPSK signal time domain response. Fig. 4 shows the BPSK signal (fc = 100MHz, w/ PRN rate of 5 x 1.024 MHz), the filter kernel (w/ 25MHz BW) and filtered BPSK signal computed through direct convolution. This is the ideal filtering result. Fig. 6 shows the post filtering BPSK reconstruction in time and frequency. Note how the time-domain result matches that of Fig. 3 (albeit with differing amplitude), and the frequency domain result matches that of Fig. 4. Fig. 5 shows the difference between the ideal filter FFT used to produce the convolution result of Fig. 3, and the down-sampled kernel used to produce the convolution result of Fig. 6. Note how the down-sampled kernel produces nearly the same result as the ideal kernel, due to the nature of the reconstruction algorithm. Remember, the frequency domain response of the down-sampled kernel only stretches to ¼ that of the ideal kernel (since there are 4-lanes of traffic), but the frequency resolution of each is the same since the *sampling rate* **and** the *number of samples* is cut by **¼**.

Chart, histogram

Description automatically generated

Fig. 3 – Filtered BPSK signal time domain response

Chart, histogram

Description automatically generated

Fig. 4 – Filtered BPSK signal before and after

Chart

Description automatically generated

Fig. 5 –Down-sampled kernel time and frequency domain response

Histogram

Description automatically generated

Fig. 6 – reconstruction of filtered BPSK signal

The main issue in running the system at such a high clock speed (614.4 MHz) is getting the timing to meet in implementation (layout). The original implementation of the convolution block is given in Fig. 7. below. The multiplication products are added together two samples at a time, resulting in an adder tree structure. Cascading ‘add’ operations together with large sample values results in a ‘carry chain’ to develop in the FPGA, which will put several nanoseconds of combinatorial delay between registers. Getting timing to meet is very challenging with this structure. Even explicitly instantiating DSP blocks to do addition is not enough to meet timing due to the large distance between DSP blocks on chip.

Diagram, schematic

Description automatically generated

Fig. 7 – Parallel FIR filter implementation (cascading adder chain)

The only way to reconcile the issues described above is to utilize the DSP blocks in a multiply-accumulate chain, which results in the DSP modules being chained ***locally*** on chip. The downside to this method is the length of the DSP chain is limited by overflow. Since we are multiplying 16-bit waveform samples and 27-bit kernel samples, the result will be 16 + 27 = 43-bit products. The maximum summation output of the DSP48 module (on the Virtex-Ultrascale FPGA) is 48-bits, so the maximum number of 43-bit results we can chain together (or accumulate) is 32, giving us the maximum kernel length of our FIR filter. The resultant frequency resolution is given by: df = fs / 2 \* N 🡪 df = 614.4e6 / (2\*32) = 9.6 MHz. Fig. 8 below shows the ideal FIR filter multiply-accumulate chain, referred to as a ‘systolic’ filter. Note that our FGPA utilizes DSP48 modules instead of DSP58 seen in the figure.

Diagram, engineering drawing, schematic

Description automatically generated

Fig. 8 – Systolic FIR filter (multiply-accumulate architecture)

A second challenge in meeting timing is making sure that the DSP blocks are configured in a fully pipelined manner. Given that our device has a speed grade of ‘-2’, the max operating frequency of the DSP48 module is 661 MHz. Note that if a device of speed grade ‘-1’ was used, the max freq. is limited to 594 MHz, which means our double-bandwidth implementation (seen in Fig. 2) simply will **not** work due to the inability to meet timing. Complete pipelining of the DSP unit is crucial, meaning we must utilize all pipeline registers inside the DSP blocks.

The desired implementation of the DSP block pipeline is seen in Fig. 8. There must be two cycles of delay between the first multiplier receiving a sample, and the next multiplier receiving a sample. This is achieved by cascading the DSP blocks together utilizing the internal DSP-chain routing on the chip. The configuration of the DSP blocks in the systolic filter chain are shown in Fig. 9. Notice how there are two pipeline registers (highlighted in yellow) between the A/B inputs and the multiplier unit. This gives the desired effect of properly aligning calculated sample/ kernel products and their corresponding addends, which is the desired configuration seen in Fig. 8. Also, this configuration utilizes *all* pipeline registers in the DSP unit, a requirement for running the unit at close to its maximum frequency. For reference, the *BCIN / PCIN* and *BCOUT / PCOUT* pins in Fig. 10 are used to allow the DSP modules to chain together on chip. This feature is also necessary to aid in timing closure due to the large distance between DSP columns on the chip.

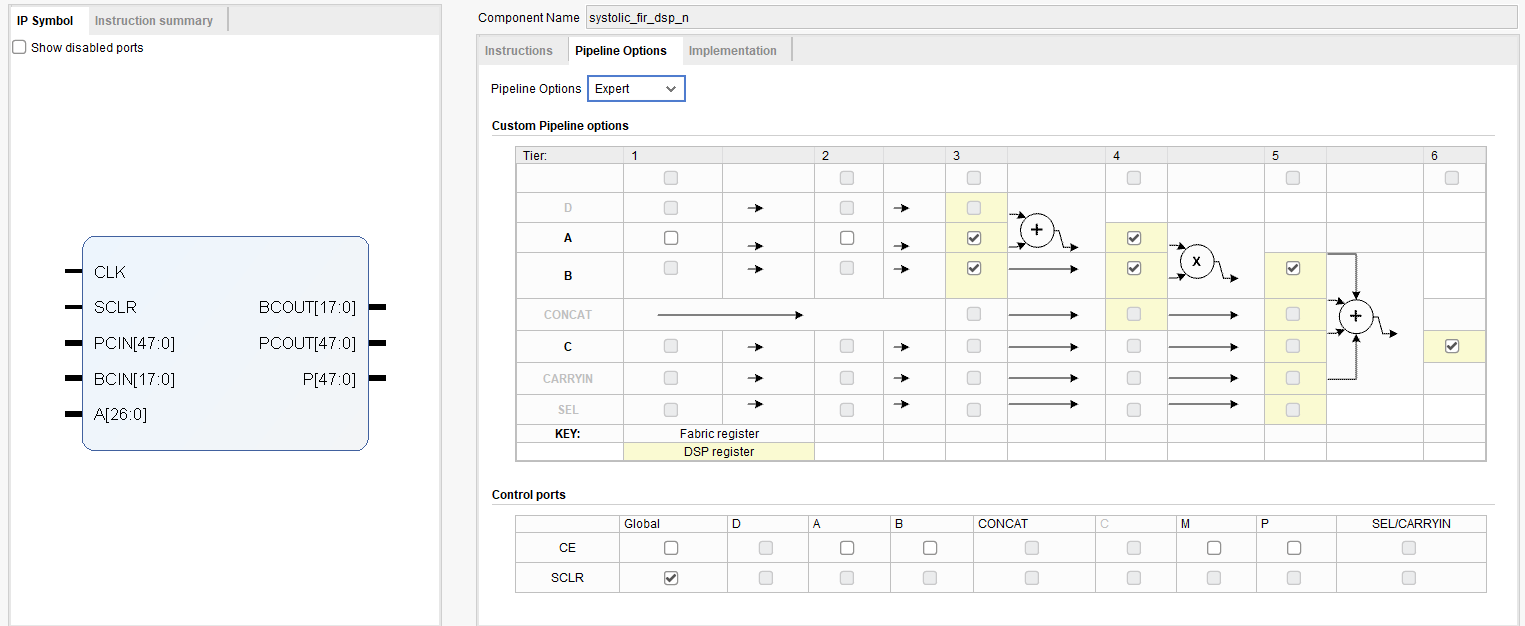


Fig. 9 – DSP48 pipeline configuration for systolic FIR filter

Finally, to compute the convolution result shown in Eq 1-4, the summations for each output lane must be fully pipelined. This is the **bottleneck** of the design when it comes to **timing closure**, due to the nature of the final summation. When the CLB’s (configurable logic blocks) are used to do addition, the use an internal adder called a *‘carry-8’*, which allows addition to be carried out in stages of 8-bits per stage. This issue with adding four 16-bit numbers together is the requirement that the path of the carry-adder chain must achieve a maximum delay of one period (~1.6 ns), which is impossible when adding several large numbers together. The technique is to pipeline the summation in a similar manner as with the DSP chain. In this case, we start with the first 16-bit sample in the sum and sign-extend it to 18-bits so we can catch the overflow that results from future additions. Next, we add to it the lower 8-bits of the second required sample in the sum padded with zeros on the upper 10-bits. This acts to sign-extend the lower 8-bits, treating them exclusively *unsigned*. Next, we add the upper 8-bits of the second sample to the running sum. To sign-extend the upper 8-bits, the two most significant bits are repeated on the LHS, while the RHS is padded with eight zeros. This process is repeated twice more to achieve the final sum for each output lane in the reconstruction block. Since each cycle in the pipelined-summation adds only 8-bits of the next sample, the samples that make up the sum must be passed through a pipeline of their own, such that they align with the running total in each consecutive cycle. Fig. 10 shows the VHDL code required to capture the input samples (conv\_res(i) and conv\_res\_prev(i)), sign-extend them, and pipeline the data to align to the running sum. Fig. 11 shows the VHDL code required to achieve the running sum, utilizing the sign-extended, pipelined input samples.

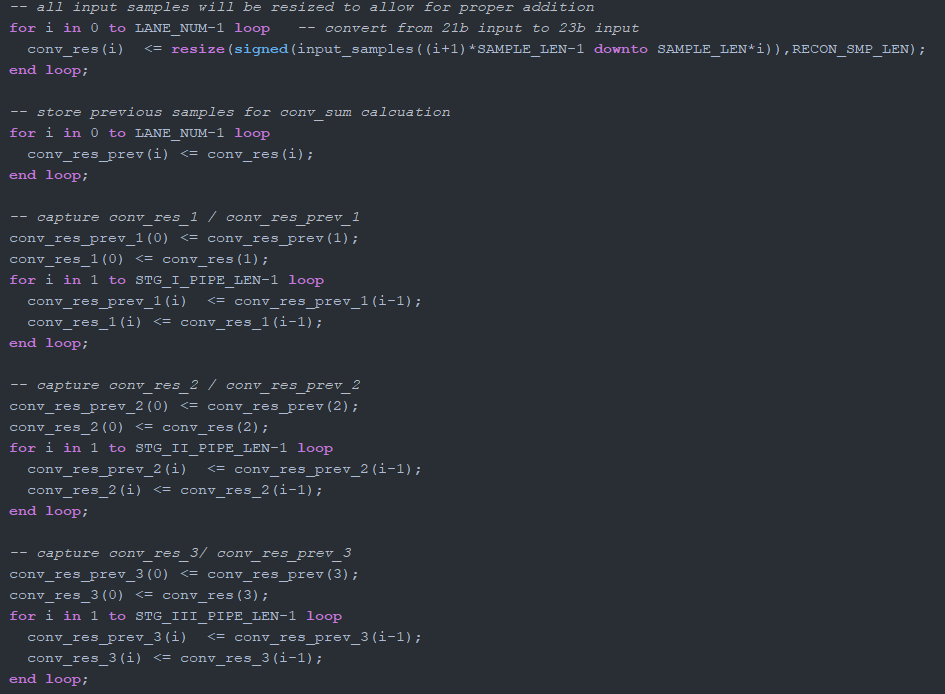


Fig. 10 – Convolution result pipeline – used for sample alignment in reconstruction sum

Text

Description automatically generated

Fig. 11 – Convolution reconstruction – fully pipelined for use with 8-bit carry adders

With the fully pipelined convolution sum ***and*** the pipelined reconstruction algorithm finished, ideally the system should achieve timing closure. Unfortunately, after running implementation there were still 12 nets that failed to meet timing, on average, by about 0.1 ns. After playing around with the implementation settings (place and route), I was able to get achieve only 1 net failure. Floor-planning was utilized to move associated LUT + Flip-Flop around on the chip to get that 1 net to meet timing. Fig. 12 shows the initial timing result and Fig. 13 shows the final timing summary for the design.

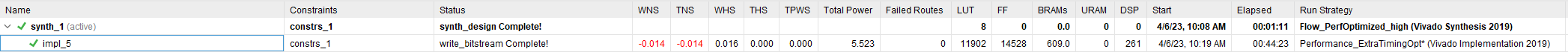


Fig. 12 – initial timing results – 1 failed net at -0.014ns slack

Graphical user interface, application

Description automatically generated with medium confidence

Fig. 13 – final timing summary post floor-planning. Timing closure achieved w/ logic running at 614.4 MHz

Keep in mind, this system architecture is setup for testing, and only has about ¼ of the usual hardware modules in use (bpsk-gen x2, cpu, JESD, sample mapper, FIR filter, misc). We must develop an implementation strategy that allows for a fixing of the closed-timing place and route for the systolic filter module. Manual floor-planning each time the design changes will not be possible with more hardware in the system architecture. ***This is an ongoing issue***.

Fig. 14, 15, 16 and 17 show the system output of a PRN code being passed through the new double bandwidth systolic FIR filter! Data is shown being passed at ~100 MHz, ~200 MHz and ~300 MHz, with a 25 MHz pass-band, showing the full 300 MHz bandwidth the double bandwidth systolic FIR filter achieves.

**Chart

Description automatically generated**

Fig. 14 – Systolic FIR filter operation – passing PRN code at 109.9875 MHz w/ 25 MHz (mixed to 1.3 GHz)

Chart, histogram

Description automatically generated

Fig. 15 – Systolic FIR filter operation – passing PRN code at 210 MHz w/ 25 MHz (mixed to 1.4 GHz)

Chart, histogram

Description automatically generated

Fig. 16 – Systolic FIR filter operation – passing PRN code at 310 MHz w/ 25 MHz BW (mixed to 1.5 GHz)

**A picture containing graphical user interface

Description automatically generated**

Fig. 17 – Systolic FIR filter operation – passing PRN code at 210 MHz w/ 25 MHz BW, full filter passband

**Things to work on:**

* Better bandwidth resolution
  + Use larger number of DSP’s in chain
  + How to handle overflow?
* Perfect timing closure
  + Which settings are required?
  + Can we **fix** the implementation results of the FIR block ??
  + Hierarchical design?
* Improve the filter kernel generation code
  + Centering of the kernel at the desired center frequency is not straight forward
  + The amplitude of coefficients generated is ¼ of what the python code generates
    - Why??
* Handling of rounding in systolic filter chain produces very low sample values
  + Does this have to do with the low values generated by the kernel-gen code?
  + How much right-shifting is necessary given our low max kernel value?