Accelerating Systems with Programmable Logic Components

UPPSALA
UNIVERSITET

Accelerating The Fast Fourier Transform

Dirk Stober
Ghaith Sankari

5th June 2020

Table of Contents

Contents

1 Project description

In this project we are using three different ways to implement The FFT algorithm where the goal is to accelerate it. We implemented a C-software algorithm and two hardware implementations, one using Verilog and one using C++ with the Vivado HLS tool. The hardware implementations are limited to calculating the FFT of 1024 samples, while the software implementation can convert samples of size 2^n. They are all implemented using fixed point arithmetic. All the three implementations have been tested using the 256k-1024 samples on the MiniZed.

2 Optimization & On-Chip and off-Chip Memories

The memories that reside next to CPU core inside the same chips are called on Chip memories. These kind of memories are used to decrease the painful latency that is caused due to the fetching of the data by the processing core from the off-chip memory which resides outside of the core chip. The latency is always related and proportional to the number of instructions that the CPU core can execute in a certain number of cycles. As the latency increases, the total number of cycles to process the instructions increases and vice-versa.
The off-Chip memories are bigger storage size to store the program data which cannot fit on the on-chip memories. The on-Chip memories store the data that the CPU core might use again but due to the compact reasons of chips, they are really smaller in size compared to the off-chip memories .
Thus, using processing chips that include on-chip memories increases dramatically the overall performance of the code. The Single-core ARM® Cortex™-A9 processor which is embedded on the MiniZed board has two on-chip memories (L1: 32 KB and L2: 128 KB–8 MB) which can guarantee better performance of the code.

For optimizing the code, in the HLS we tried to pipeline the loops which are used for the multiplications, additions, and subtractions of FFT calculations. However, this resulted in using more resources than the MiniZed can provide. Furthermore, the hardware-Verilog optimizations will be discussed in section 4.5.

3 Software Implementation

3.1 Data Types and algorithm

This part was implemented using recursion to continuously split the input array into even and odd elements until both elements are of size 1. After that, the FFT is applied by multiplying the odd element with the exponential factor and recombining the result with the even elements to calculate the output. To model the real and imaginary parts of a complex number, a structure is used. Moreover, separate functions were used to multiply, add and subtract complex numbers. A simplified FFT recursive algorithms is represented in the pseudo code below. We initially used single precision floating point numbers for testing. However, we changed to fixed point data types for a fair performance comparison between the three implementations. The library *libfixmath*[1] was used for the conversions from floating to fixed point representations.

[1] PetteriAimonen. *Fix-point math library.* `https://github.com/PetteriAimonen/libfixmath/`.

Algorithm 1: Recursive FFT pseudo code

```
Result: X[] as frequency domin sequence
#points,x[],X[],TEMP[],Cnt;
if #points != 1 then
    (TEMP(0...(N/2)-1))=FFT(even(x[0....#points -2 ]);
    (TEMP'(0...(N/2)-1))=FFT(odd(x[0....#points -1 ]);
    while Cnt is less than (#points/2)-1 do
        X[Cnt]=TEMP[Cnt]+ W* TEMP'[Cnt];
        X[Cnt+(#points/2)]=TEMP[Cnt]- W* TEMP'[Cnt];
        Cnt++;
    end
    return X[0....#points-1];
else
end
```

3.2 Euler's equation

To calculate the exponential twiddle factor, we initially used the fixed-point cosine and sine function, this however produced less precise results than using the floating point functions. It also didn't lead to any performance improvements and thus the floating point function are used to calculate the sine and cosine values. One could have used a lookup table similar to the one used in the hardware implementation to speed up the execution, but this was not done as it would have limited the flexibility of the software implementation.

3.3 Evaluation and Testing

The given test benches were used to testify the correct implementation of the program. The algorithm was successfully tested with all the provided test benches without any calculation errors. Also, the algorithm was tested with the timing libraries to check the performance when running it on the MiniZed board that has the Single-core ARM® Cortex™-A9 processor.
As a result, using floating point implementation was around 1.5x faster than the fixed point implementation because the ARM® Cortex™-A9 processor has a high performance VFPv3 floating point unit. Also, transforming the data form floating points to fixed points caused a latency. More detailed performance comparisons are represented in the evaluation section.

4 Verilog-Implementation

4.1 Data Type

Similar to the HLS implementation fixed points were chosen over floating point to implement the FFT-Algorithm. The main reason for this is that, fixed point arithmetics are easier to implement and faster. We used 32-bit wide fixed point values with a precision of 16. They function in a very similar way to integers. The only downside is a slight loss in precision, as well as increased difficulty in debugging, as floating point representation can be easily translated to a decimal representation. In the beginning, there was also the hope that the fixed-point butterfly operation, would not need to be pipelined, which would have allowed for an easier implementation.

4.2 Data-path

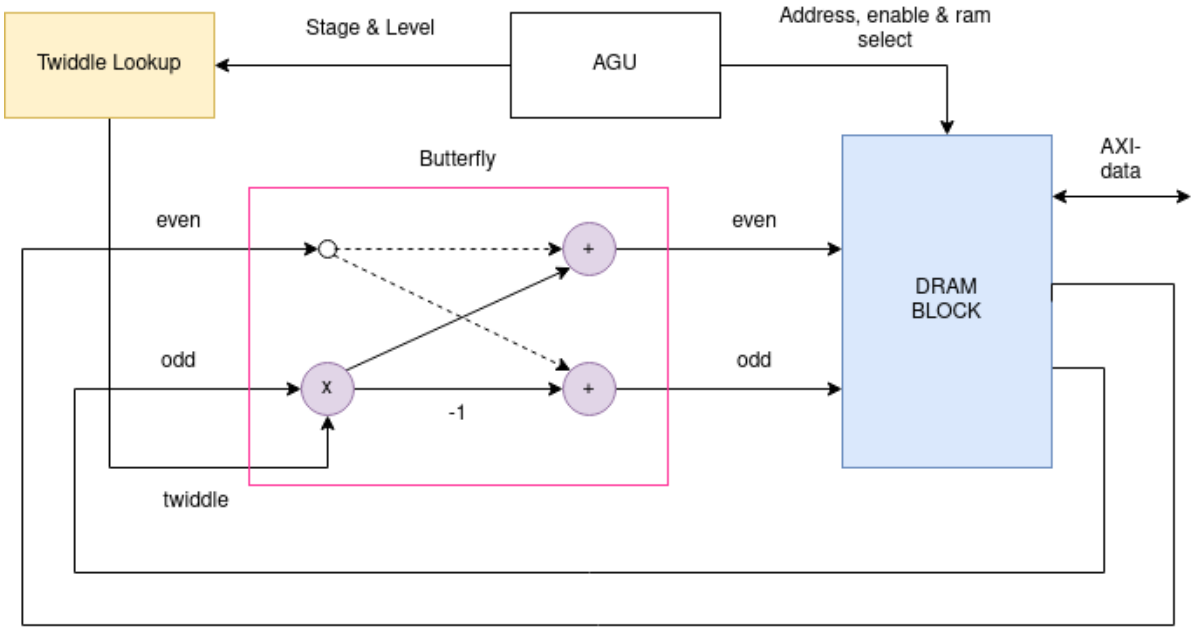

Figure 1: Data-path components

4.2.1 Complex Multiplier

The complex multiplier consists of 4 multiplier IPs and two adders. The real and imaginary outputs are multiplied accordingly and fed to the adders to calculate the real and imaginary output. As the input values have a precision of 16, they are shifted to the right by 16 after multiplication. This is done by configuring the IP to output the correct 32 bits ($output[31+16:16]$). There is no checking for overflow or underflow and also no rounding leading to all number being round to $-\infty$. The IP uses 6 pipeline stages and after the additions another register is added.

4.2.2 Butterfly

The core piece of the datapath is the butterfly units. It applies a fft to two inputs, getting the necessary twiddle factor from a lookup table. It consist of a complex-multiplier, subtracter and adder. The odd input is multiplied by the rotation factor and than subtracted from the

even input for the odd output and added to the even input for the even input. As the complex multiplier has a pipeline length of 7, the even input has to be delayed by the same amount before, being send to the adder-/subtracter unit. The total pipeline length of the butterfly unit is 7.

4.2.3 RAM-Block

The ram-block is used to store the values of the data points between the stage. The first intuition was to use one block to both read and write to. It is however not possible, to synthesis a RAM that can and read/write from 4 different addresses at once. If it would be possible to use the butterfly unit without a pipeline it would be possible to write to and read from the same two cells in one clock cycle. Unfortunately, Vivado did not allow us to synthesis such a asynchronous ram.
It is thus necessary to either use a very long pipeline, or a second RAM to guarantee that no values are overwritten. This second RAM allows us to write and read from different location, changing the directions at each stage. The values are still written and read to the same addresses, this is possible by reordering the data when it arrives from the CPU as explained in this paper[2]. This is done by bit reversing the addresses of the incoming data.

4.2.4 Twiddle-factor Lookup

To calculate the twiddle factor for each butterfly operation, a lookup table is used that outputs the rotation factor depending on the current stage and the identifier of the lower address. Because of the relation between cosine and sine, it would have been possible to implement the lookup table using a quarter of the memory, but this would require additional adders and logic and BRAM is widely available, such that it has been decided to save all 512 complex results of the rotation factor calculations $(\cos(-2 * \pi * k/n), \sin(-2 * \pi * k/n))$. The lookup table is generated using C and the libfixmath library.

[2] D. Cohen. "Simplified control of FFT hardware". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24.6 (1976).

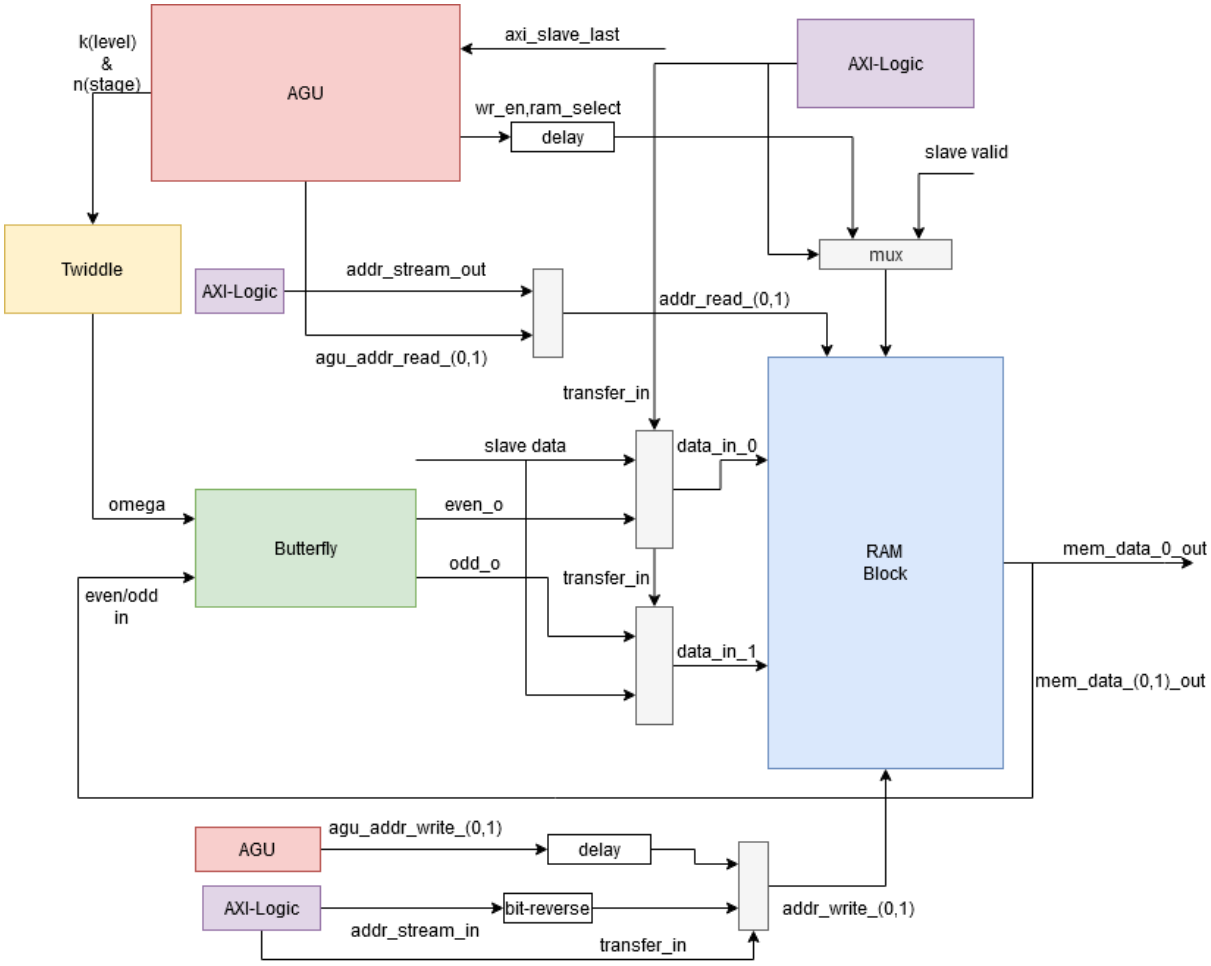

Figure 2: Diagram of control and data signals

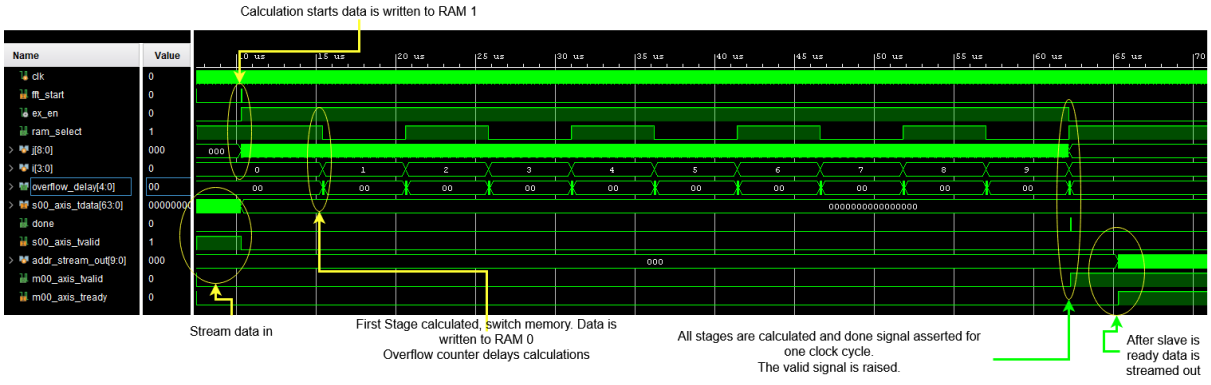

Figure 3: Simulation showing one FFT calculation

4.3 Control Path

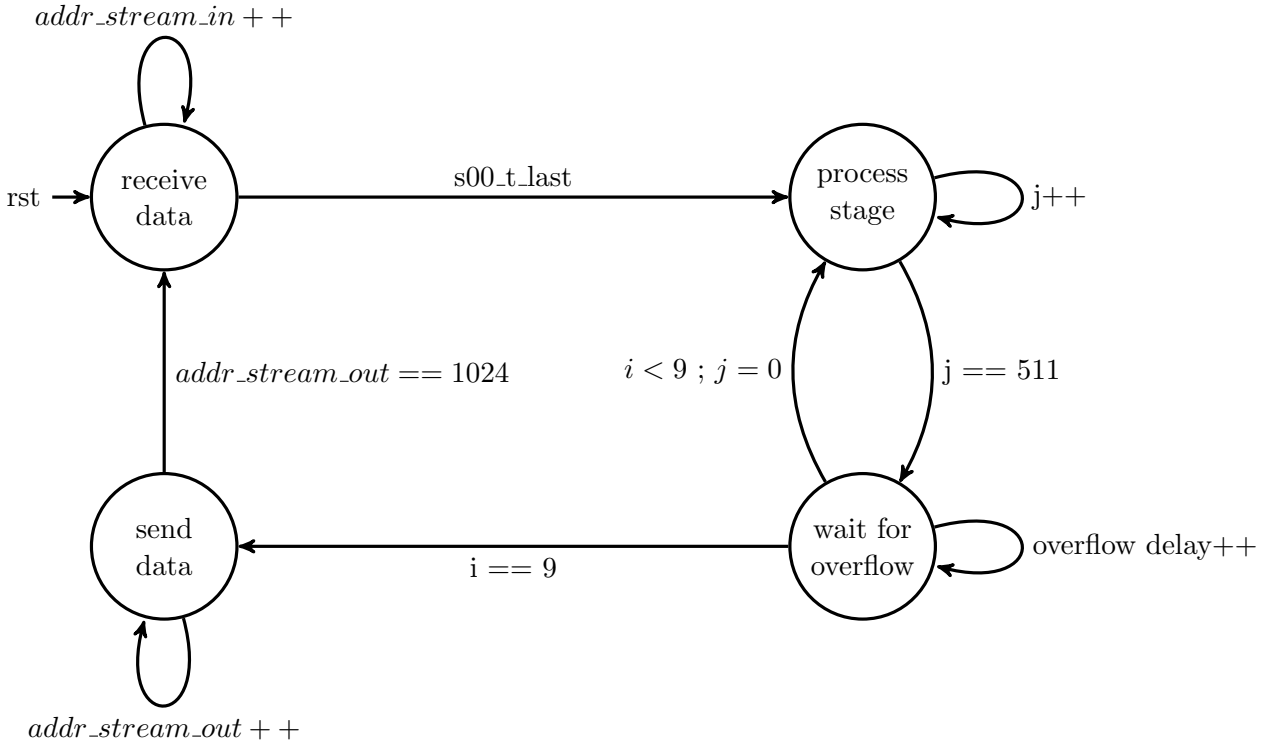

Figure 4: Control Flow Graph

The calculation are mostly controlled using the AddressGenerationUnit (AGU). A calculation is started once the last signal of the incoming stream is asserted. The AGU cycles through the different addresses, providing both addresses for the butterfly unit and the current stage and iteration to the twiddle look-up table. After completing all stages the done signal is asserted for one cycle and the axi master valid pin is set, allowing for data to be transferred back. After all data has been sent, new data can be received. It is important to note that the first value of the frequency domain is sent twice, which means 1025 words have to be transferred through the axi interface and the first one discarded.

4.4 AGU

The core piece of the FFT is the address generation unit , which controls which values are fed into the butterfly unit together. It is inspired by this paper about implementing a 32-FFT[3]. It has three internal counters, keeping track of the current stage (i : 0 - 9), the current level inside a stage (j : 0 - 512) and one to flush the pipeline (overflow delay: 0 - 8) before progressing to the next stage. Depending on the stage the offset from the lower to the upper address can be calculated, by shifting 1 left by the current stage. The lower address is incremented by the value of the inc register ($lower\_addr = (lower\_addr + inc)\%1023$). Its initial value is 2 and it is shifted left in each stage and set to one for the final stage.

[3]George Slade. "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation". In: (Mar. 2013).

stage:	0	1	2	3	4	5	6	7	8	9
offset:	1	2	4	8	16	32	64	128	256	512
inc:	2	4	8	16	32	64	128	256	512	1

Table 1: Increment and offset of addresses for the FFT stages

Once the level counter is about to overflow a delay logic is triggered, which delays the execution for 8 cycles to allow the pipeline stages to be emptied.

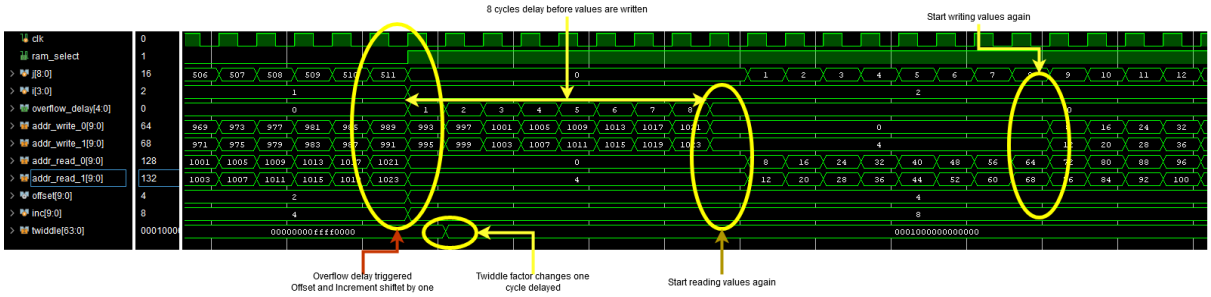

Figure 5: Simulation showing delayed write addresses and flushing of pipeline

4.5 Pipelining

The most difficult part of implementing the FFT was to implement the correct delays in combination with the pipelined butterfly unit. First, we implemented the butterfly unit without any pipeline stages. There was however still a need to delay the write address by one because the RAM could not be read and written to/from the same address and the lookup table takes one cycle to produce the necessary twiddle factor. RAM select and the write address were delayed by one cycle to counteract this delay. This design however could not be implemented, because it did not meet the required timing constraints (Figure 6). After looking at the report it became clear that the multiplier had to be split up, to allow the design to run at a desired frequency of 150 MHz.

Intra-Clock Paths - clk_fpga_0 - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	-12.562	22	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_1/DIBDI[27]	18.864	11.747	7.117	6.7	clk_fpga_0	clk_fpga_0
Path 2	-12.543	22	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_1/DIADI[27]	18.852	11.755	7.097	6.7	clk_fpga_0	clk_fpga_0
Path 3	-12.528	22	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_1/DIADI[27]	18.833	11.755	7.078	6.7	clk_fpga_0	clk_fpga_0
Path 4	-12.517	22	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_1/DIBDI[27]	18.824	11.747	7.077	6.7	clk_fpga_0	clk_fpga_0
Path 5	-12.279	19	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_0/DIBDI[31]	18.544	11.376	7.168	6.7	clk_fpga_0	clk_fpga_0
Path 6	-12.189	19	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_0/DIBDI[31]	18.507	11.376	7.131	6.7	clk_fpga_0	clk_fpga_0
Path 7	-11.973	19	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_0/DIADI[31]	18.235	11.376	6.859	6.7	clk_fpga_0	clk_fpga_0
Path 8	-11.865	19	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_0/DIADI[31]	18.165	11.376	6.789	6.7	clk_fpga_0	clk_fpga_0
Path 9	-11.795	16	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_1/DIBDI[20]	18.096	12.018	6.078	6.7	clk_fpga_0	clk_fpga_0
Path 10	-11.755	17	4	design_1_i/fft...g_0/CLKBWRCLK	design_1_i/fft_0...reg_0/DIBDI[28]	18.020	11.984	6.036	6.7	clk_fpga_0	clk_fpga_0

Figure 6: Negative slack when using unpipelined butterfly unit

Instead of designing our own multiplier, as in the unpipelined version, it was decided to use the Vivado mutliplier IP, which allows the multiplication of custom length bit vectors. We use the use the optimum number of pipeline stages, according to Vivado, which is 6. Another

pipeline stage is added after the addition and subtraction of the temporary results. Using this multiplier in the butterfly unit, our total data path has a delay of 8 (see Figure 5) before the data can be written to the RAM. To accommodate this the write addresses and the write enable signals are delayed by this amount.

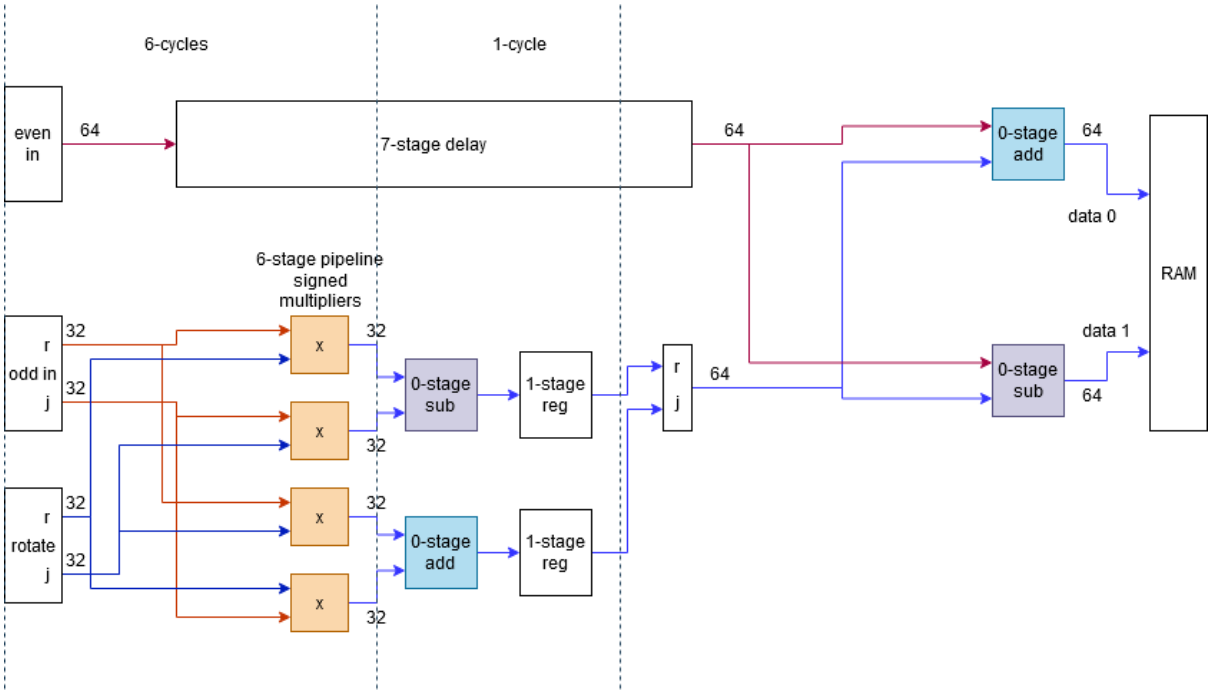

Figure 7: Pipelined butterfly unit

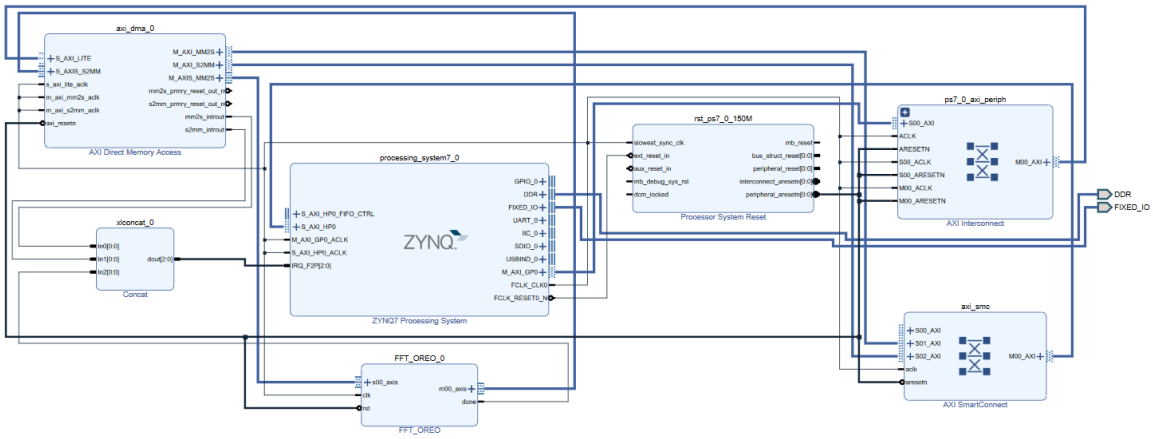

Figure 8: Integration of the Verilog FFT IP in the system

5 HLS Implementation

5.1 Description

For the High Level Synthesis implementation, a different methodology was used to implement the FFT algorithms since HLS doesn't support recursion. The iterative method was followed instead of the recursion. An outer for loop that iterate up to the $(\log_2(FFTpoints))$ which represents the number of stages needed to split the input array into even and odd elements to perform the Arithmetic needed to obtain the frequency domain results of the FFT. First we used fixed point functions to calculate both sine and cosine, but this resulted in HLS generating a CORDIC IP. This leads to less precise results, than using a lookup table. To force HLS to use a lookup table the sine and cosine calculations are done using floating point operations. The HLS design was implemented and then exported as an IP to Vivado. Then, a block design was created to connect all necessary components for the transmission of data between the Zynq and the IP. The design was tested successfully in the SDK.

5.2 Software to Hardware mapping

After the HLS synthesises a C/C++ code, it produces a hardware descriptive implementation of the high level code. After Mapping to Verilog, different parts of the high level statements are mapped differently. For instance, The function arguments of the FFT function in HLS are defined as HLS stream ($hls :: stream < data_stream >$) and mapped as AXI stream ports and the variable vectors to produce the FFT results were mapped to Block Rams. Moreover, the HLS identifies the behaviour of the a loop as a state machine that has different control states which are relevant to arguments inside the loop. The control states represent different synthesized circuits which are executed in each clock cycle. For example, in one of the loops in the FFT function the odd and even elements calculations will be mapped to their synthesised circuit representations like multipliers and accumulators and then will be controlled and executed by the HLS's extracted state machine over a certain number of cycles.

5.3 System Integration

The AXI4-Stream bus was used as the handshaking protocol to receive and send data between the IP and the the Zynq. The AXI Direct Memory Access IP was used because the AXI bus is memory mapped and needs a direct memory access between the memory and the AXI4-Stream target peripherals or IPs and the Zynq. To integrate the IP with a system that uses the AXI bus, some HLS pragmas were introduced to the input and the output vectors of the FFT to be supported with AXI ports and be able to handshake with the other IPs. Furthermore, the FFT funtion was also introduced using a pragma as an AXI-lite which enables it to be configured and interact with the Zynq PS side . Refer to figure 9 for the overall illustration of the block design.

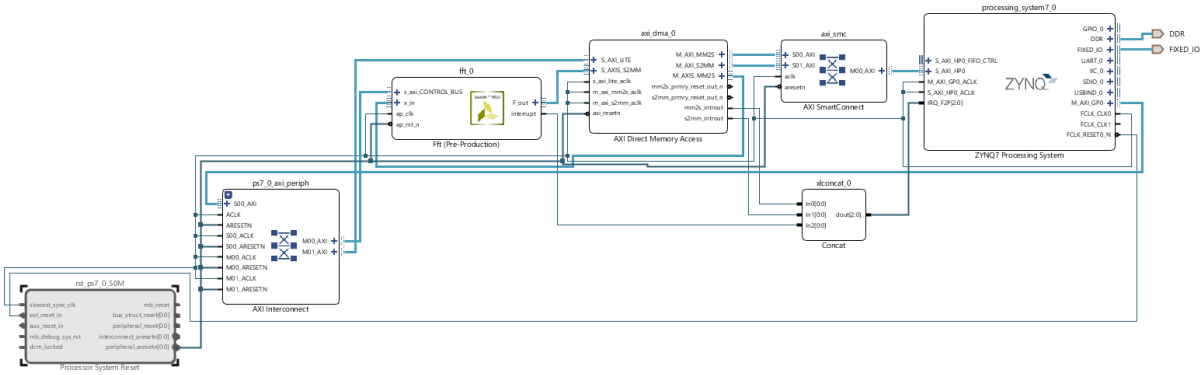

Figure 9: Integration of the HLS FFT IP in the system

5.4 HDL and HLS Schematics

It's challenging to compare the schematics of both designs, but what we can observe is that the HDL produced much less hardware cells (216) and nets (739) where all the hardware components(AGU, butterfly, RAM blocks,etc..) are represented as blocks, whereas the HLS produced 2550 cells and 5102 nets where most the hardware components are extracted in to one schematic. As previously mentioned HLS uses a CORDIC IP, when using the in build fixed point cosine functions.

6 Evaluation

To test the performance of all implementation, we used the 256k sample test data. We calculated 1024 samples at a time and added up the total execution times. For both Verilog and HLS we also included the time necessary to send the data to the device. From the evaluation results tables below we can observe that the execution time of Hardware-Verilog implementation is 3.8x less than the Software implementation and 3.05x less than the HLS. The Software implementation is only 1.2x slower than the HLS. Refer to table 2 and table 3 for the execution time results.

# Points & Samples	Software	HLS	Hardware-Verilog
1024-1024:	6351 us	5077 us	1664 us

Table 2: The execution time of the FFT among the 3 implementations for one 1024 sample

# 1024 points 256k samples:	Software	HLS	Hardware-Verilog
Total Execution time:	1637161 us	1299812 us	426188 us
Average Execution time:	6395 us	5077 us	1664 us

Table 3: The total and average execution time of the FFT among the 3 implementations for 256k samples

In terms of power we looked at the provided reports from the Vivado implementation, which suggest that the HLS uses less than half the power of the Verilog implementation. It seems seems a bit unlikely that this is the case as the HLS versions uses more components than the Verilog implementation. The C-implementation could be improved by using a lookup table.

The Verilog implementation took the most time to implement, but allowed for the most precise optimization and the best results. It is however more difficult to configure it to a new problem size, when comparing it to the HLS implementation and especially the C-implementation that can solve the FFT for different problem sizes already. To conclude we managed to accelerate the calculation of the FFT algorithm on a FPGA, using both HLS and Verilog. Using Verilog we managed to achieve a speedup of more than 2x with the added benefit of being able to use the CPU for different purposes. Refer to table 4 and table 5 for the utilization and power consumption comparisons.

Component:	LUT	BRAM	DSP	Slice
Max:	14400	50	66	4400
Verilog:	692	5	16	221
HLS:	4494	4	56	1475

Table 4: Utilization of HLS and Verilog implementation

Type:	Total	Signals	Data	Dsp	BRAM
Verilog :	0.064	0.009	0.009	0.029	0.018
HLS :	0.027	0.005	0.005	0.013	< 0.001

Table 5: Power Consumption report Vivado in Watt

References

[1] D. Cohen. "Simplified control of FFT hardware". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24.6 (1976).

[2] PetteriAimonen. *Fix-point math library.* `https://github.com/PetteriAimonen/libfixmath/`.

[3] George Slade. "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation". In: (Mar. 2013).