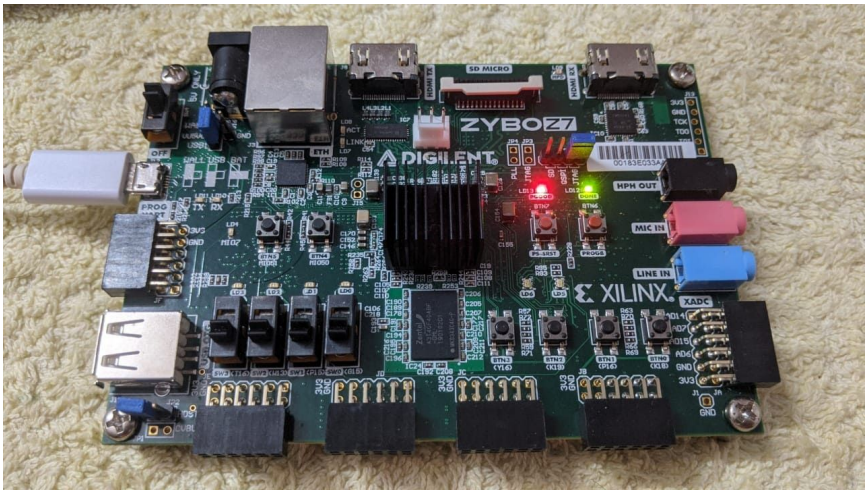
**EE2003 Computer Organisation Course Project**
**DIGITAL SPECTRUM ANALYSER USING HW ACCELERATOR FOR COMPUTING FFT**
**Documentation**

Team: Akash Reddy (EE17B001), Arjun Menon V (EE18B104), Muralekrishnan R (EP18B026)
Aug-Nov 2020

**Github Repository:** https://github.com/murale127/HW-Spectrum-Analyzer

## Problem Statement:

Design and implement an FFT-based digital spectrum analyser on the Zybo Z7-20 FPGA evaluation board based on Xilinx's Zynq 7000 SoC. A hardware accelerator for computing the Radix-2 FFT is designed, implemented on the Zynq7 PL and then interfaced with the Zynq7 PS (ARM Cortex A9) using an AXI bus interface. The spectrum analyser is implemented using a C++ application which takes in input data from Data Memory (DDR RAM) on the Zynq7 PS and sends output to a Laptop using UART protocol. A python application is written and run on the Laptop to store the serially input data in a file and plot it on the Laptop's screen.
The speedup provided by the FFT HW accelerator is computed by comparing the solution with a software version of the FFT algorithm that is run on the Zynq7 PS. The speedup and resource usage of the various hardware optimisations on the accelerator are also computed and compared.




**TARGET BOARD: ZYBO Z7-20**

## System Description:

***HW Accelerator Design:***

The HW accelerator was designed using Vivado HLS tool and is based on the implementation of FFT in Reference 2. HW optimisation directives (loop unrolling, function pipelining and block level data flow being the main optimisations) are added to the baseline code provided in this repository; additionally, the fft module is modified to cater to the needs of the spectrum analyser- the input data is multiplied by a time domain window (Hanning, Hamming or Blackman) before the FFT stage and the magnitude-spectrum of the FFT is returned along with the FFT in cartesian form. Both time domain window multiplication and computation of magnitude spectrum of FFT use DSP48 slices on the FPGA, and provides speedup in comparison to an implementation on the PS. This design is then synthesised and packaged as a custom IP.

We also implemented a 256-point FFT version of the IP for complex valued float inputs (branch ver4_256). This has a latency of 2500 cycles and an II of 550 cycles. However, this design was resource expensive, consuming close to 250 DSP slices and 49000 LUTs (this exceeds the number of DSP48 slices on the target board). The resource usage can be improved by migrating from floating point operations to fixed point as a significant number of resources is consumed for floating point operations.
Our current stable versions of the IP (ver2.9 and ver3) both compute the 32-point FFT; ver2.9 supports floating point inputs and uses axi-lite interfaces for all ports on the IP while ver3 supports fixed point inputs (32 bit fixed point with 24 integer bits) and uses the axi-stream interface for vector inputs and outputs.

The current version of the implementation is not interfaced with an ADC as we do not have access to an analog signal generator, instead the input is stored in the Data Memory of the PS and fed to the peripheral.

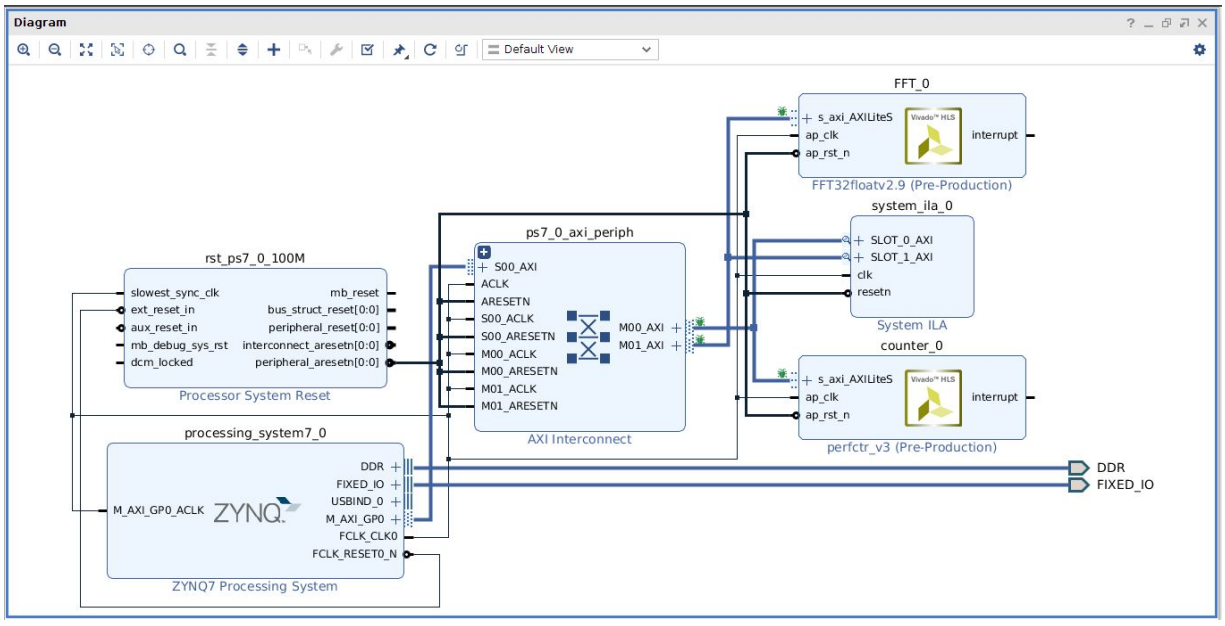
***Peripheral Interfacing and Bus Architecture:***

The block level design of the system is performed using Vivado. The following peripherals are interfaced with the Zynq7 Processor (ver 2.9):

1. FFT32 Peripheral exported as an IP from VHLS- the peripheral has the following ports:
    a. data_IN: input data port

b. win_mode: 8-bit one hot encoded input selecting window to be multiplied by (currently only 3 windows are supported, but this can be extended based on the requirements)
c. data_OUT: output data (cartesian coordinates) port
d. mag_OUT: magnitude-squared spectrum of output data
e. All ports are interfaced as axi-lite slaves (memory mapped)

2. A simple performance counter IP exported from VHLS- this is used to track the number of cycles elapsed during the SW and HW runs of the application program; interfaced as an axilite slave
3. UART1 peripheral on the Zynq7 PS is made active- this is used to send output data from the processor's DMEM to a laptop for plotting
4. AXI4 interconnect- to interface the slave axilite ports to the AXI4 master ports on Zynq7 PS
5. A JTAG based Integrated Logic Analyser (ILA) is added to the design to debug signals of interest.




**BLOCK DESIGN FOR VERSION 2.9**

In ver3, additional Xilinx AXI DMA IPs are used to interface the High Performance AXI slave port (HP0) on Zynq with the AXI-stream ports on the IP. The DMA is used in simple transfer mode (as opposed to Scatter-Gather mode in which the CPU transfers the burden of memory transfer to the DMA), and uses FIFOs to move Memory Mapped data to Stream interfaces and vice versa. The IP design is incomplete as there are some issues in interfacing the DMA with the FFT peripheral (refer Status section).

***On Bus Architecture:***

Zynq7 supports three types of AXI bus interfacing between custom IP on the Programmable Logic (PL) and the PS- AXI4-lite, AXI4-memory mapped and AXI4-stream.

AXI4 memory mapped and AXI4 lite both use 5 channels for transferring data from the master device to the slave device:

1. Write Response Channel from S to M
2. Write Data Channel from M to S
3. Write Address Channel from M to S
4. Read Address Channel from M to S
5. Read Response Channel from S to M

This architecture supports faster transfer of data by enabling out of order data transfer, burst transfer etc. AXI4 Memory Mapped bus supports transfer of upto 256 words following a single address and handshake cycle (burst mode), while AXI-lite supports transfer of only word per handshake cycle. AXI-Stream on the other hand uses only a transfer channel and does not have a handshake cycle- hence it can support a burst transfer of unlimited length.

The IP block design is synthesised and the bitstream for the board is generated after which the design is exported to Vitis IDE for developing an application project over the IP design. The exported hardware design creates drivers to the instantiated peripherals and provides header files for the same. The application program uses functions provided by these header files to initialise, read and write to these structures.

***Spectrum Analyser:***

The output of the spectrum analyser is a time-series of magnitude response plots of 32-point windowed FFTs of the input data. This is then plotted using a python program. The details of the Short Time Fourier Transform are given below:

Number of FFT Bins = 32

Windowing Functions: Hamming window, Hann window, Blackman window; function determined by an 8-bit input (window chosen by considering trade-offs on main lobe width, side lobe peaks etc.)

Window Shift = 32 points (no overlap between consecutive windows)

Sampling Frequency is to be determined by the input source and ADC, since we are reading input from RAM, this is left as a variable. However, the peripheral is fast enough to process speech and audio signals in real time (audio is typically sampled at 44.1kHz).

***Serial Communication and Python Application:***
The magnitude-squared spectrum of the input signal is transmitted serially to a laptop through the UART1 peripheral on the board.
UART settings: 115200 bps Baud Rate, 8N1 (8 input bits per packet, no parity and 1 stop bit)
Since the UART Tx Buffer in Zynq can hold only 64 bytes at a time, the buffer is reset after the transfer of every 16 float values.
We used the serial module in python to connect to UART and store data into a file. The values in this file were then read, converted to dB scale and plotted at a time-series of stem plots.

---

## Design Choices:

***Choice of Processor:***
We had considered two processors initially- RISCV based shakti E-class SoC and ARM cortex A9 based Zynq7 PS. Both processors support floating point instructions. The target board includes the cortex A9 processor and is interfaced with the Programmable Logic part of the Zynq SoC. We use the Zynq processor in our design for better usage of blocks on the FPGA. We also found more online resources for developing a project (bus architecture, interfacing custom IP, deploying on a board etc.) using the Zynq processor compared to the Shakti processor.

***Number representation:***
1. The peripheral can be designed to operate on float variables using HLS- the required floating point arithmetic units is synthesised by HLS automatically
2. In comparison to float, fixed point computations require fewer FPGA resources and are faster. Hence a fixed point implementation would help us achieve better speed up.
3. The dynamic range of a 32-bit fixed point signed representation is 2^31 which gives approx 186dB of dynamic range in magnitude spectrum. Since most windows have ripples contributing > -100 dB, this seems good enough.
4. Ver2.9 uses floating point representation while ver3 uses fixed point representation. Since the Zynq PS does not support fixed point computation, a floating point to fixed point converter IP is used.

***Hardware Optimisations:***
1. Used #pragma HLS compiler directives in Vivado HLS

2. PIPELINE adds registers to the datapath of a function (a function is implemented as a module by HLS) enabling lower initiation intervals (number of clock cycles between the start of two consecutive function calls). This improves the overall latency of the system.
3. DATAFLOW enables pipelining at the block level when there are multiple function calls in cascade. Registers or BRAMs are placed in the datapath to improve the Initiation Interval (II).
4. LOOP UNROLL is used to replace an iterative loop (for loops for eg) with a parallel implementation. Multiple instances of the loop body are created, wherever applicable, bringing down the trip count of a loop. The amount of parallelism can be controlled using the parameter <factor>.

---

## **Status:**

***branch baseline_v1:***
Implemented a baseline version of FFT32 for floating point without adding any optimisation directives. The FFT function defined in Reference 2 was used for this. Interfaced the peripheral with Zynq7 using AXI-lite bus and compared this with the Software implementation on Zynq7.

1. Latency of FFT Peripheral = 1761, Initiation interval = 1761
2. The peripheral took around 1800 clock cycles to compute the FFT
3. Read and Write Stages to the peripheral took about 600 cycles each since the interfaces were AXI-lite
4. In comparison, the software implementation took around 2200 cycles to complete, making it faster than the FPGA implementation.
5. The implementation can be sped up by optimising the hardware for speed and by using a faster bus architecture.

***Version 2 (branch v2_arjun):***
Added PIPELINE, DATAFLOW and LOOP UNROLL directives to FFT implementation in HLS.

1. Latency of FFT Peripheral = 300 cycles, II = 83 cycles
2. Compared to baseline implementation, version 2 has close to a 6 fold improvement in latency and over 20x speedup in II. Version 2 also resulted in more resource usage with about 5 times the number of DSP48 slices being used (all 5 stages of the FFT are now parallelised due to DATAFLOW)

***Version 2.9 (branch ver3:SpecAnalv2.9/):***
Added time domain window multiplication to the peripheral- window is chosen using a switch case, which gets synthesised as a MUX logic. In addition to returning the output in cartesian form, the magnitude squared spectrum is computed and returned. Both window multiplication and polar form conversion were pipelined, additionally the loops in window multiplication were factored by a factor of 2.

1. Uses AXI-lite interface for all ports. Using data flow optimisation with axi-lite resulted in warnings on FIFO size being limited, hence we removed this optimisation from ver2.9 design.
2. Latency = 439 cycles, II = 439 cycles
3. Resource Usage: 52 DSP48 slices and 12500 LUTs
4. We did not compute the phase spectrum of the FFT as it was not required for our target application. This lets us save DSP slices and LUTs that would have been used otherwise for implementing the floating point arctan function using the CORDIC algorithm
5. IP interfaced with Zynq7 PS as mentioned in "*Peripheral Interfacing and Bus Architecture"* section
6. C++ Application code for HW speedup computation, UART interfacing and computing 32-point STFT of input data
7. Python application to receive data from UART buffers of laptop, store in a file and generate a time-series stem plot of magnitude spectrum (converted to dB scale before plotting) in PC
8. ver2.9 is an end-to-end working solution of our design

***Version 3 (branch ver3:SpecAnalv3/):***
Improvement on ver2.9 with float replaced by 32 bit fixed point and input and output vectors interfaced as axi-stream, to enable burst memory access. Currently this work is stalled due to issues in interfacing the peripheral with AXI DMA.

1. Latency = 263 cycles, II = 70 cycles
2. Resource Usage: 112 DSP48 slices, 24000 LUTs
3. IP interfacing includes AXI DMA for Memory Mapped to Stream (MM2S) and Stream to Memory Mapped (S2MM) connections. In addition, an AXI-stream interfaced Floating Point IP by Xilinx is used for converting floating point input data (stored in RAM) to fixed point and vice versa
4. IP interfacing and C++ application incomplete

***Issues with burst memory access:***
Memory access is the bottleneck in version 2.9 with the read and write stages requiring over 3000 cycles to complete. The number of cycles required can be brought down

drastically by using burst  transfer to and from the peripheral. This can be achieved in three ways:

1. Interfacing input and output ports as AXI-stream interfaces and using DMA or Data Mover IPs for MM2S and S2MM connections. However, the DMA uses additional control signals such as TLAST, TKEEP, TSTRB which are not automatically generated during High level synthesis. UG902 documentation by Xilinx suggests using side channels in AXI-stream for including these signals to the design, but we could not find a simple way to implement side channels when the input data is complex valued (written as structures in C++).
2. Interfacing input and output ports as AXI Memory Mapped (Full) interfaces. Functionally this is similar to interfacing the ports as AXI-stream with MM2S and S2MM connections as both support burst transfers upto 256 words using a single handshake cycle. The AXI Full bus is however more complex than AXI-stream and AXI-lite buses and we could not find resources for interfacing a custom IP with an AXI Full bus.
3. Custom HDL-based bus architecture based on AXI protocol for interfacing the PS with the FFT peripheral.

---

## **Results:** *(ver2.9)*

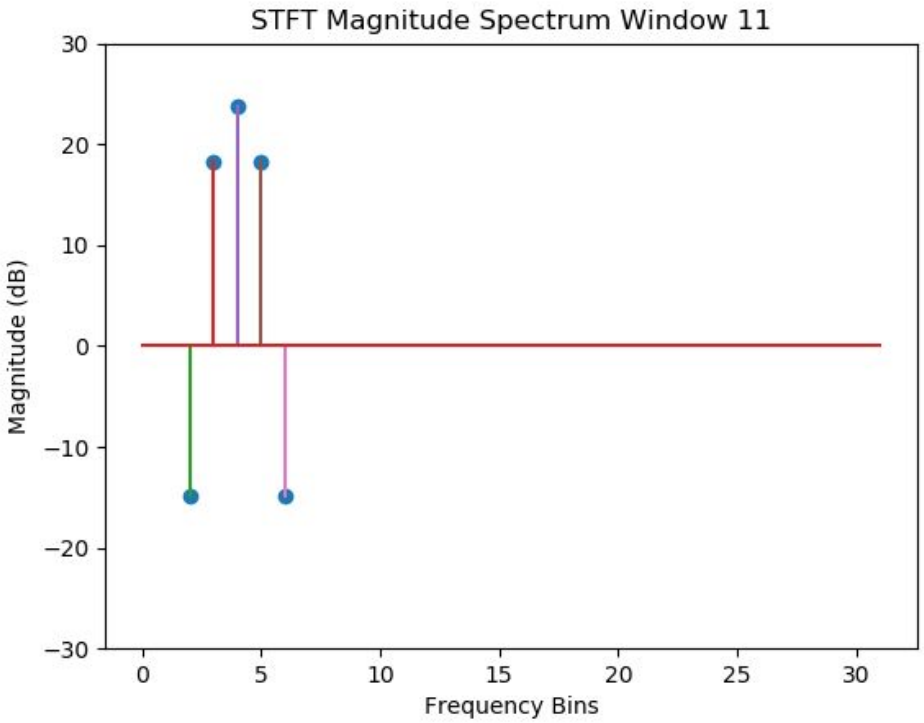
The version 2.9 IP has a critical delay of 7.7ns , hence it is capable of operating at clock frequencies < 129MHz. We use the 100 MHz clock from the PL fabric Phase Locked Loop on the Zynq device. Each 32-point FFT takes close to 4000 cycles (including memory access, using axi-lite) which corresponds to about 40 µs. This allows real-time computation of the 32-point STFT of audio signals which are generally sampled at 44.1kHz.

***Software vs Hardware Implementation:***

As is evident from the results, the bottleneck in FFT computation is caused by memory access (read and write cycles). In terms of computing FFT, the number of cycles taken is approximately equal to the latency predicted by Vivado HLS synthesis reports. Overall, the HW implementation provides around 2x speedup compared to SW although this can be increased further by using axi-stream interfaces and fixed point representation (version 3).

```
File  Edit  View  Search  Terminal  Help
Window Mode Used: 1

PASS SW
SW FFT cycles: 7050
PASS HW
HW FFT cycles:
Data Write Stage:  1462
FFT computation: 518
Read Stage: 1922
Net HW time: 3902

Window Mode Used: 2

PASS SW
SW FFT cycles: 7004
PASS HW
HW FFT cycles:
Data Write Stage:  1467
FFT computation: 519
Read Stage: 1924
Net HW time: 3910

Window Mode Used: 4

PASS SW
SW FFT cycles: 7063
PASS HW
HW FFT cycles:
Data Write Stage:  1492
FFT computation: 517
Read Stage: 1923
Net HW time: 3932
```

**32-POINT FFT RUNS (SW VERSUS HW) FOR HANNING, HAMMING AND BLACKMAN WINDOWED INPUTS**




**SCREEN GRAB FROM STEM PLOT OF MAG SPECTRUM (dB) OF FFT OF exp(j*pi*n/4)- side bands due to windowing**

---

## References:


***FFT on HW:***

1. FFT on Hardware using Vivado HLS- baseline implementation: Vivado HLS Example: FFT
2. Source Code for Nitin Sir's HLS implementation of FFT: Nitin Chandrachoodan / Demos with FPGAs · GitLab
3. Analysing Vivado HLS synthesis: EE5332-2019-03-11
4. Primer on FFT and its Digital architecture: The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation
5. Parallel Programming for FPGAs by Ryan Kastner, et. al. : Chapters 4 and 5 in https://arxiv.org/pdf/1805.03648.pdf
6. Software Implementation of FFT in C: Numerical Recipes in C is a good reference (Chapter 12) Numerical Recipes in C

***Spectrum analyser:***

1. FFT Spectrum Analyzer: Fast Fourier Transform » Electronics Notes
2. About FFT Spectrum Analyzers
3. The Fundamentals of FFT-Based Signal Analysis and Measurement
4. The Short-Time Fourier Transform
5. Time Domain Windows and their Freq char: The Rectangular Window

***Xilinx Vivado and Vivado HLS design:***

1. UG871 Vivado Design Suite Tutorial (HLS): Vivado Design Suite Tutorial: High-Level Synthesis (UG871)
2. UG902 Vivado Design Suite Tutorial (HLS): Vivado Design Suite User Guide: High-Level Synthesis
3. UG1037 Vivado Design Suite Tutorial (AXI Reference Guide): Vivado Design Suite: AXI Reference Guide (UG1037)
4. UG761 AXI Reference Guide: UG761 AXI Reference Guide
5. PG021 AXI DMA IP: AXI DMA v7.1 LogiCORE IP Product Guide
6. PG060 Floating Point IP: Floating-Point Operator v7.1 LogiCORE IP Product Guide
7. Zynq Training Playlist by M. Sadri (videos 7 and 9): ZYNQ Training - session 07 part I - AXI Stream Interfaces in Detail (RTL Flow), ZYNQ Training - session 09 - part IV - Transfer Data from PL to PS using AXI DMA

***Python Serial Communication:***

1. Playlist on Arduino and Pyserial: https://youtube.com/playlist?list=PLb1SYTph-GZJb1CFM7ioVY9XJYlPVUBQy

***Real Time Plots in Python:***

1. https://www.youtube.com/watch?v=Ercd-Ip5PfQ&feature=emb_logo
2. https://inst.eecs.berkeley.edu/~ee123/sp16/lab/lab3/lab3-Part_I_Time-Frequency-Spectrogram.html
3. https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.animation.FuncAnimation.html

***UART on Zynq***

1. https://www.youtube.com/watch?v=W7eNZBf1qZs
2. https://github.com/aslaamshaafi/Zynq_7000_SDK/blob/UART_MIO/project_1.sdk/test_uart/src/helloworld.c