**EE2003 Computer Organisation Course Project**
**Documentation Draft**

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

**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-stream 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 PC using uart. A python application is written and run on the PC to store the serially input data in a file and plot it on the PC'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.

**Approach:**

***HW Accelerator Design-***
The HW accelerator was designed using Vivado HLS tool and is based on the implementation of FFT in https://gitlab.com/chandrachoodan/teach-fpga/-/tree/master. HW optimisation directives (loop unrolling, function pipelining and block level dataflow 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 of the FFT in dB scale is returned along with the FFT in cartesian form. Both time domain window multiplication and computation of log10 of magnitude of FFT use DSP48 slices on the FPGA, providing speedup in comparison to an implementation on the PS.
Currently, we are also working on a 256 frequency bins version of the code, which would provide a better time-frequency trade-off for signals sampled at high sampling rates (speech and audio for example).

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-***

Peripheral Interfacing is done using vivado IP integrator. The following peripherals are interfaced with the Zynq7 Processor:

1. FFT32 Peripheral exported as an IP from VHLS- the peripheral has the following ports:
    a. data_IN: input data port, AXI-stream interface
    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); AXI-lite slave interface
    c. data_OUT: output data (cartesian coordinates) port, AXI-stream interface
    d. mag_OUT: magnitude of output data in dB scale, AXI-stream interface
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 host PC for plotting
4. AXI4 interconnect- to interface the slave axilite ports to the axilite master ports on Zynq7 PS
5. AXI Data Mover- Since AXI-stream interfaces include only a Data Transfer Channel as opposed to the Read and Write Address and Transfer channels on an AXI4 (lite or memory mapped) bus, we need to add the AXI data mover IP between the AXI-stream ports and the PS so that the ports are memory mapped to the address space of the PS. An alternate approach would have been to interface the input and output ports as m_axi ports (AXI4 memory mapped). However m_axi ports with compiler directives to add dataflow to the design raised errors during HLS synthesis.

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 (which includes typedef structs for each peripheral) 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).

**Status:**

1. Implemented a baseline version of FFT32 for floating point (branch baseline_v1) without adding any optimisation directives. The FFT function defined in https://gitlab.com/chandrachoodan/teach-fpga/-/tree/master was used for this. Interfaced the peripheral with Zynq7 using AXI-lite bus and compared this with the Software implementation on Zynq7.
    a. Latency of FFT Peripheral = 1761, Initiation interval = 1761

    b. The peripheral took around 1800 clock cycles to compute the FFT
    c. Read and Write Stages to the peripheral took about 600 cycles each since the interfaces were AXI-lite
    d. In comparison, the software implementation took around 2200 cycles to complete, making it faster than the FPGA implementation.
    e. The implementation can be sped up by optimising the hardware for speed and by using a faster bus architecture.
2. Version 2 (branch v2_arjun): Added PIPELINE, DATAFLOW and LOOP UNROLL directives to FFT implementation in HLS.
    a. 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.
    b. 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).
    c. 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>.
    d. Latency of FFT Peripheral = 300 cycles, II = 83 cycles
    e. 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)
3. Version 3 (branch ver3): 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 is computed in decibel scale. Both window multiplication and polar form conversion were pipelined, additionally the loops in window multiplication were factored by a factor of 2.
    a. Latency and II increased to 365 cycles and 99 cycles respectively. The increase in II was due to the conversion to polar form. This module (polarOUT) involves computing the magnitude square of the output and then computing its logarithm. We used the log10 function from hls_math.h library.
    b. The polarOUT module also consumes a good amount of DSP slices- 27 slices per instance, with about 19 slices being used by the log10 function.

The for loop in this module was not unrolled in order to conserve the number of DSP slices being used. With factor=2 loop unrolling, the II of this module was around 82 cycles; however, the DSP48 utilization of the entire system had shot up to 206/220.

c. In version 3, we are also migrating from the axilite bus interface to AXI-Stream interface. The IP design and Application development in SDK are still work in progress.
d. Version 3 will be a complete working solution of our design.
e. Remaining work:
    i. Interface FFT32 IP AXI-stream ports with Zynq7 using an AXI Data Mover IP.
    ii. Export IP design and write C++ application project on standalone OS for spectrum analysis of test data using drivers for FFT peripheral, performance counter and uart peripheral (on PS)
    iii. Detailed Documentation of the Project and Comparisons on the basis of speed and Resource Usage

Note: In addition to ver3, we also parallely worked on a 256-point version of the version 3 design (branch ver4_256). The latency and II were close to 2500 cycles and 547 cycles respectively. However, the implementation exceeded the number of DSP48 slices on the board. We chose to not go ahead with this version as we would have to move some of the computation away from the PL (log(mag(FFT)) computation and/or multiplication by window) to make the solution implementable.

**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)
   More references to be updated