EKG ARRHYTHMIA DETECTER

Musheera Khandaker and Walter Stadolnik

HEART DISEASE

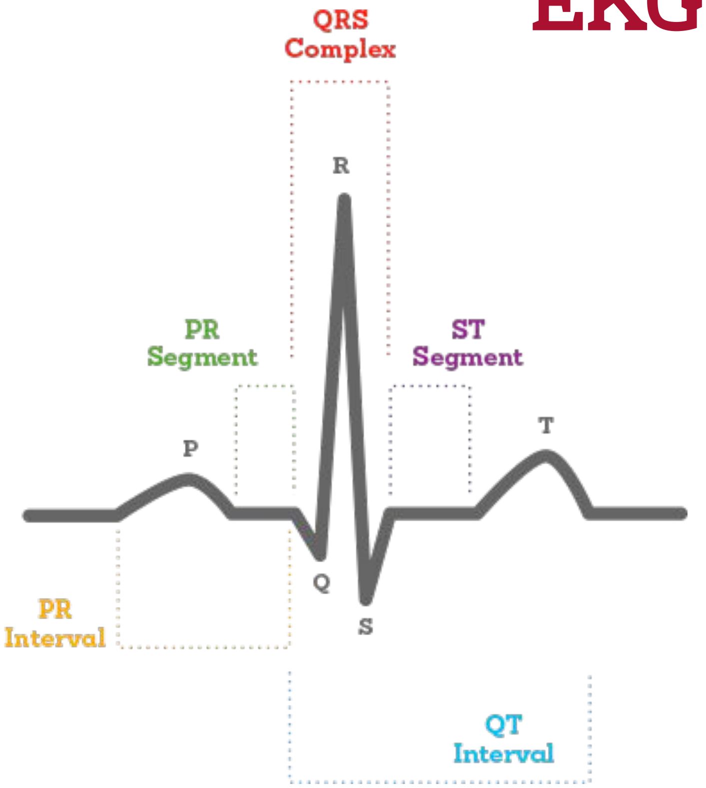
#1

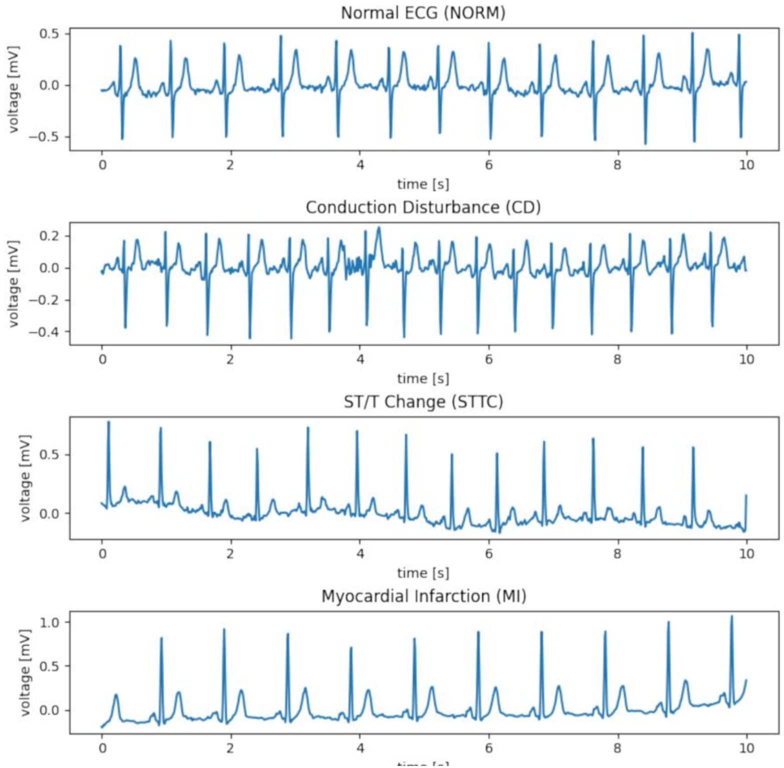
Leading cause of death globally

800,000

Heart Attacks per year in the US

EKG Signals


QRS
Complex
R
PR
Segment
ST
Segment
P
T
Q
S
PR
Interval
QT
Interval


Normal ECG (NORM)
Conduction Disturbance (CD)
ST/T Change (STTC)
Myocardial Infarction (MI)
voltage [mV]
time [s]

Process

1

INPUT DATA

Read in noisy input data from EKG sensor

2

BANDPASS FILTER

IIR filter that attenuates noise

3

PEAK DETECTION

Use simple moving average filter and square data

4

THRESHOLD DETECTION

Detect EKG signal components by comparison with thresholds

5

DIAGNOSIS

Detecting arrhythmia based on EKG features

Overview of IIR Filters

- The simplest representations of IIR filters are difference equations, or rational system functions:

Difference Equation:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k].$$

System Function:

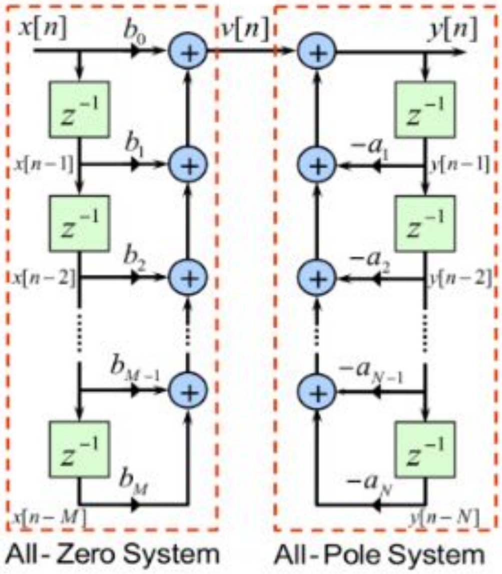
$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}$$

- Unlike FIR filters, which do not use feedback, IIR filters contain both Feed-forward terms (*b* coefficients) and Feedback terms (*a* coefficients)
- IIR filters are often chosen over FIR filters in applications with limited hardware resources, because the number of coefficients needed to achieve the same filtering quality is lower.

Image Credits: Prof. Vinay Ingle, EECE5666

IIR Filter Structures

Direct Form I IIR Filter Structure



Direct Form II IIR Filter Structure

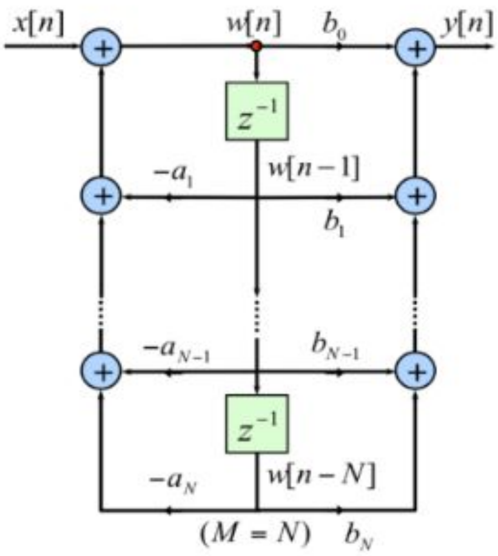
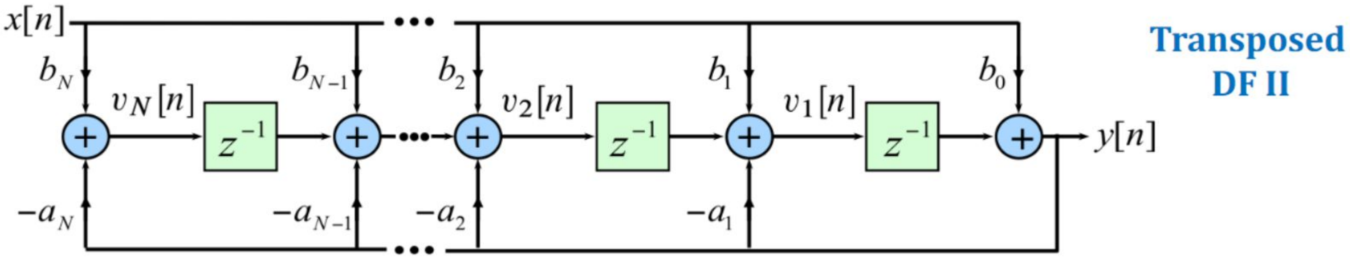


Image Credits: Prof. Vinay Ingle, EECE5666

Direct Form II Transposed Structure

- Known to be the most efficient structure for IIR filter implementation in terms of hardware resource usage
  - Can utilize a single tapped delay line, rather than two for DFI
  - Fun fact: this is also the structure used in MATLAB's *filter* function



$$y[n] = v_1[n-1] + b_0 x[n]$$

$$v_k[n] = v_{k+1}[n-1] - a_k y[n] + b_k x[n], \quad k = 1 \dots, N-1$$

$$v_N[n] = b_N x[n] - a_N y[n]$$

Image Credits: Prof. Vinay Ingle, EECE5666

Software Implementation

- Chose to use Butterworth filters for bandpass filtering
  - Butterworth filters have a maximally flat frequency response in the passband, which makes them a good choice for ECG filtering
- Used MATLAB to generate the *a* and *b* coefficients for a 6th order lowpass filter with a cutoff frequency of 30Hz, and a highpass filter with a cutoff of 0.5Hz.

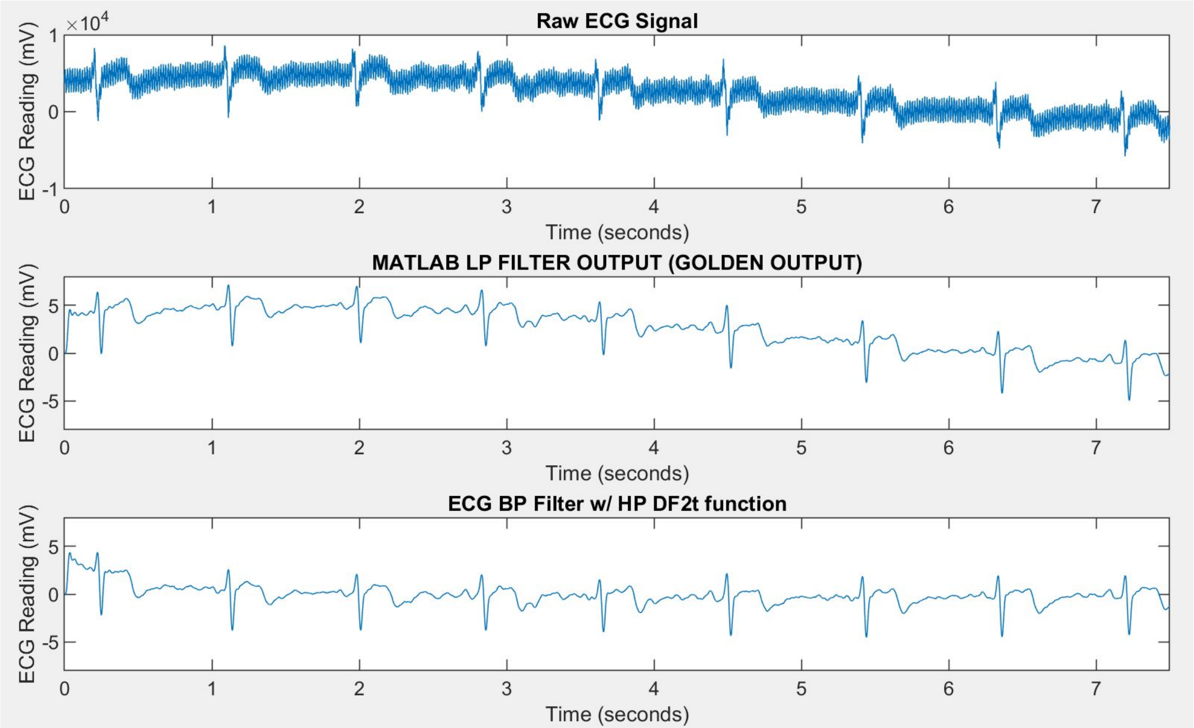
```
% Deriving LP Buttersworth Filter coefficients
fc1 = 30;
[b1,a1] = butter(6,fc1/(fs/2),'low');
b1_int = int32(b1*(2^24));
a1_int = int32(a1*(2^24));
```

- Also implemented a custom function for DFII IIR filtering

```
% Transposed DF2 Filter Function
function y = filter_FPGA_DF2t(b,a,x)
```

MATLAB Filter Outputs

- Exported results to CSV for use as “golden” filter outputs
- Adapted the filter function to Python for use on the ZYNQ board



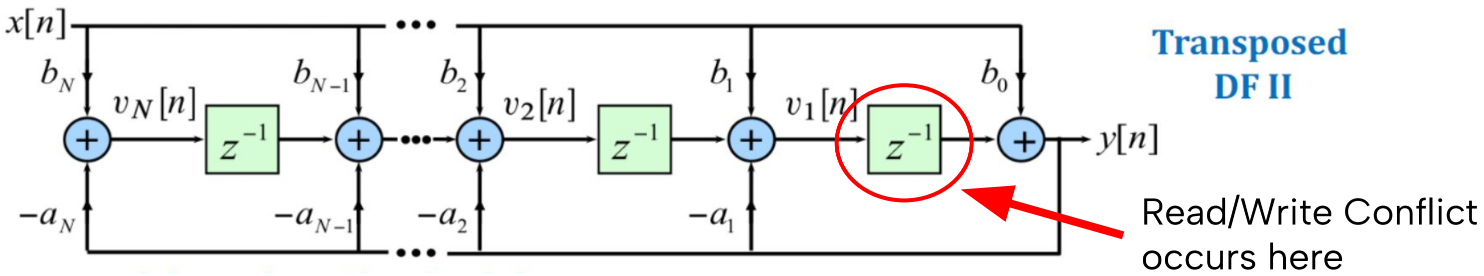
Hardware Implementation

- Used 32-bit AXI-Stream for PS-PL interface
- Decided to multiply floating point ECG data by 1000 and casting to int to simplify sending data over AXI-Stream

- First major issue arose with accurately representing coefficients. Our smallest coefficient is 0.000000495352235
  - Our first approach to this was to scale up the coefficients by a factor of 2^N, and then scale down the output by bit-shifting it to the right N times for transfer over AXI-Stream
    - Ran into issues with needing unreasonably large data types for the delay line (as large as ap_int<128>), and we eventually scrapped this approach
  - Switched to using fixed point data types for the coefficients and the delay line array. We eventually settled on ap_fixed<64,32> to represent all fixed point data, to ensue there were sufficient bits to represent both the integer and fractional parts (more on optimizing this later)

Data Dependency Problem

- Because the IIR filter uses feedback, unlike the FIR filter we did in class, there were more issues with read/write data dependencies



- Conflict occurs at the last stage of the tapped delay line:
  - Value is read each iteration when calculating y[n], and also written to when shifting data through the delay line in each iteration
- Our initial approach to solve this was to create a *temp* variable that would be updated with the state of v1[n] whenever new data is shifted into v1[n]. We would then read from this *temp* variable when calculating the y[n] value, to try to avoid reading from v1[n] and immediately writing to it
  - This *seemed* to work when we tested it

Data Dependency cont.

- The reason it didn't cause an II violation was because we didn't make the *temp* variable static
    - Caused the value of *temp* to be re-initialized to zero each time the while-loop repeated, preventing the data dependency issue from showing up in synthesis, but producing an output of all zeros when tested on the PYNQ board

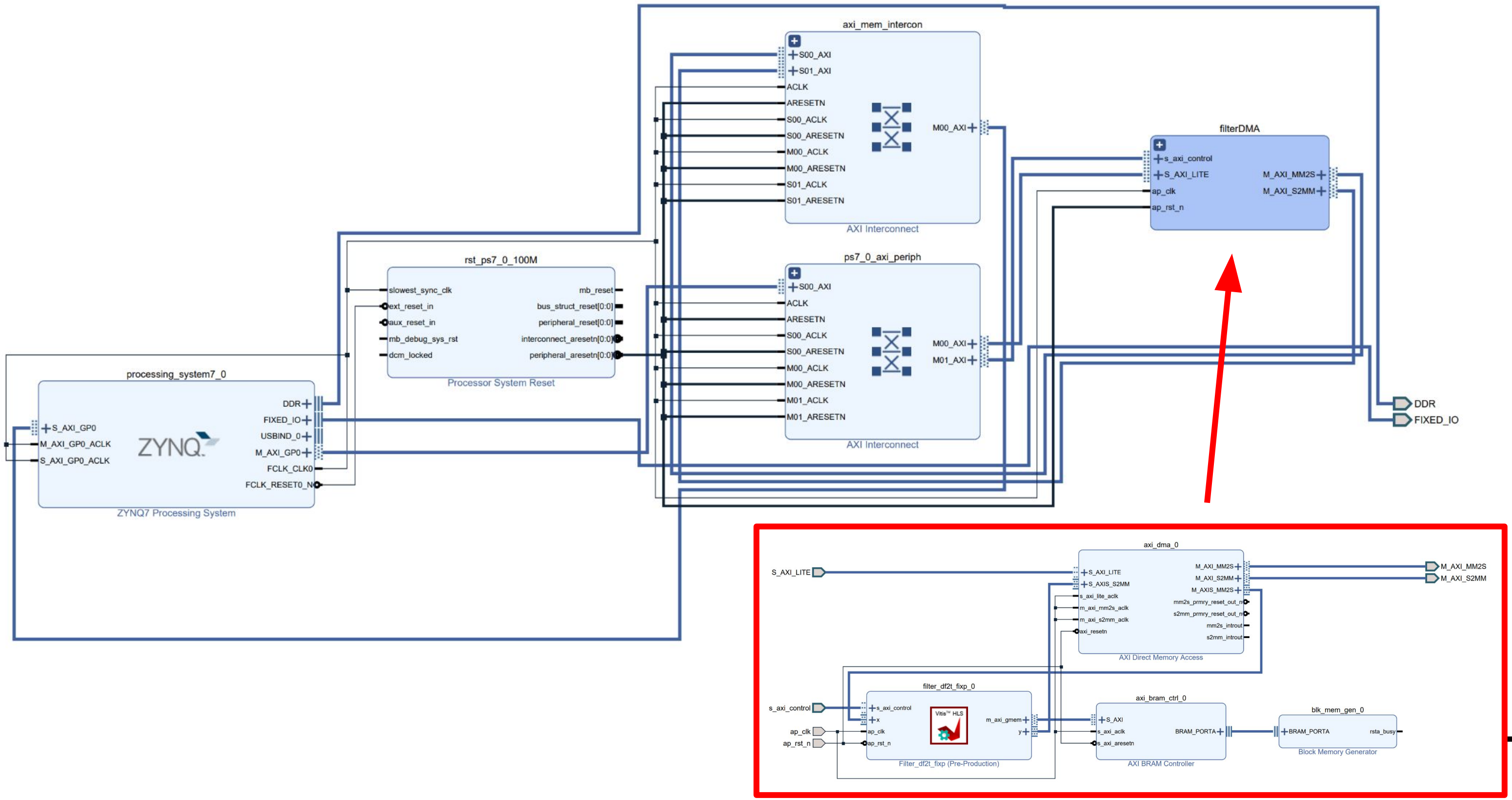
- It took us longer than it should have to diagnose this problem, because the filter function actually worked perfectly when run on our testbench.
    - It seems that the simulation by default assumed the *temp* variable was preserved between iterations, even though it wasn't declared as *static*

- Our actual solution was to pipeline the shifting loop with #pragma HLS PIPELINE II=2
    - Seems to eliminate the data dependencies by padding each loop with an additional clock cycle, which separates the read and write operations

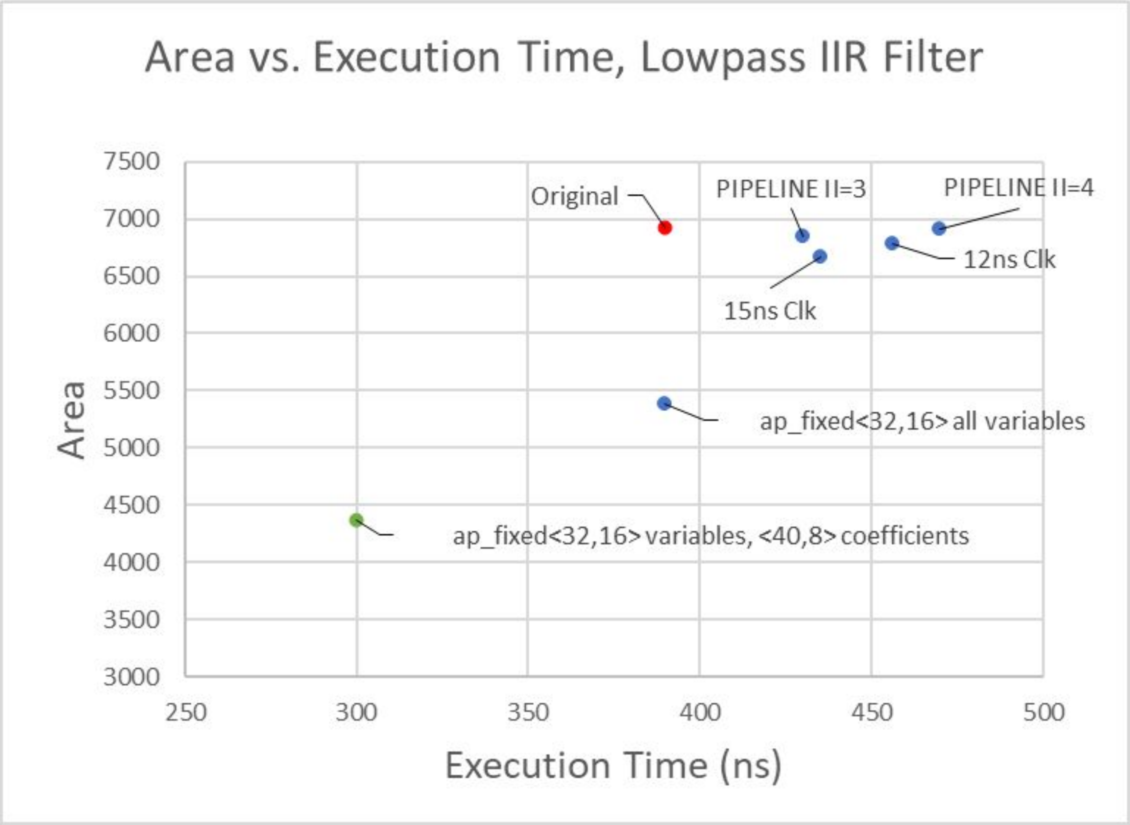
Other Issues

- Initially, we tried initializing the filter coefficients within the filter function instead of using a .coe file, but Vivado was not a fan
  - Eventually used our testbench to print the fixed-point coefficient values in hex to allow us to generate the .coe file
- There were initially some minor issues with bus width mismatches that we had to sort out in Vivado, mostly involving the 64-bit size of our .coe file
- Still trying to work out a data dependency fix for the high pass filter.
  - Only has 2 *a* and 2 *b* coefficients, which means that it effectively skips the delay line shifting loop altogether, and the strategy of using the pipeline pragma to add a delay no longer applies.

Block Diagram

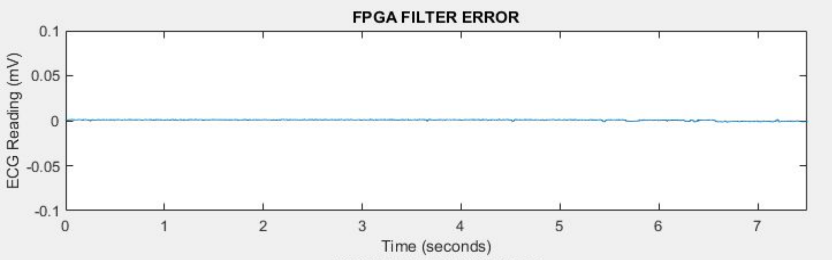


Optimizations

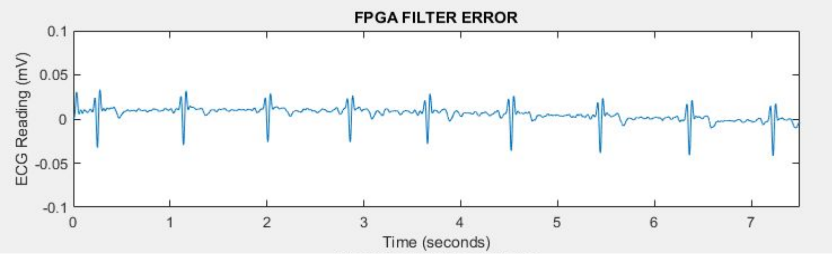


Data Type Optimization

- To optimize our fixed point coefficients and delay line variables for minimum hardware resource usage, we used MATLAB to find the error between our testbench output and the golden output.
- We then decreased the number of bits allotted to the fractional and integer portions until we observed a significant error
- Settled on ap_fixed<32,16> for delay line variables, ap_fixed<40,8> for coefficients



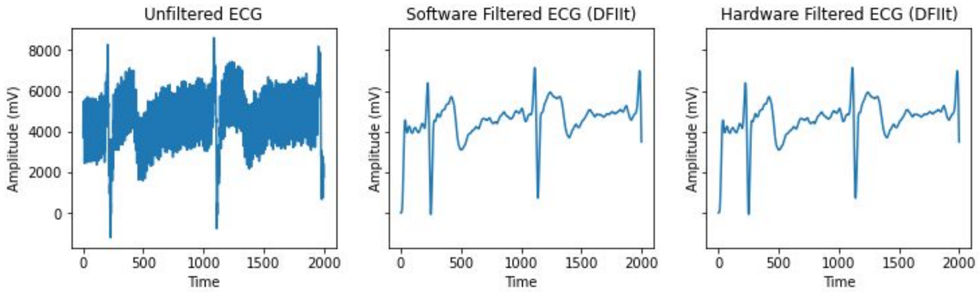
ap_fixed<128,32> for delay line data and coefficients (Very little error)



ap_fixed<56,32> for delay line data and coefficients (Relatively large degree of error)

Results: Before + After Optimization

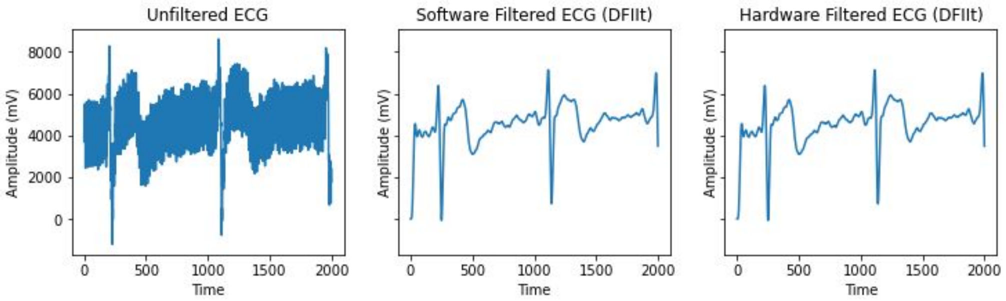
Lowpass Filter Output *Before* Optimizations



Average Speedup over software:

12.45

Lowpass Filter Output *After* Optimizations



Average Speedup over Software:

13.41

Speedup due to optimization: **1.08** (expected 1.3 based on VITIS sims)

Conclusions

Data Types

Advantages of switching between data types for different purposes

Data Dependencies

Paying attention to read/ write operations, static variables in while (1)

Data Precision

While decreasing precision can improve hardware resource usage, you can't use too few bits

Testbenches

Can be extremely useful, but at times misleading, as they may still interpret HLS code differently than hardware

Future Work

Peaks in Hardware

Add peaks detection and arrhythmia detection in hardware

Test Other Structures

Would be interesting to prove that the DFII transposed form has the best hardware efficiency, and see if another form has better speedup

Real-Time processing

Ideally, we would connect an ECG to an ADC on the PYNQ to enable real-time processing. This would allow us to investigate filter stability