FPGA Implementation of ECG Signal Processing Using Fast Fourier Transform (FFT)

Marcu-Cristian Petric

Technical University of Cluj-Napoca

February 15, 2025

Abstract

This paper presents the design and implementation of an FPGA-based system for real-time ECG signal processing. The system filters ECG data through Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) to reconstruct the filtered signal. It then computes R-R intervals and the heart rate, displayed on a 7-segment display, while detecting cardiac conditions such as bradycardia and tachycardia. The efficient real-time data monitoring and low power consumption demonstrate the FPGA's suitability for resource-constrained medical applications. **Keywords:** *ECG signal processing, FPGA, Fast Fourier Transform (FFT), R-peak detection, Bradycardia, Tachycardia, Real-time monitoring, Medical signal processing, Low power consumption*

Contents

1 Introduction

The *electrocardiogram* (ECG or EKG) is a diagnostic tool used to verify the heart's electrical activity. It is one of the least invasive methods for diagnosing and monitoring cardiac conditions. However, traditional software-based processing systems suffer from latency and high significant power consumption, limiting their application in real-time or portable settings [DCJ+21].

Modern ECG systems must handle tasks like noise filtering, peak detection, and anomaly identification with minimal delay. Achieving this in real time while keeping power consumption low is a challenge that traditional processors struggle to meet. This is where hardware-based solutions like Field-Programmable Gate Arrays (FPGAs) become valuable.

Field-Programmable Gate Arrays (FPGAs) are preferred when time-critical calculations are required due to their low latency and absence of operating system (OS) overhead. Compared to CPUs or GPUs, FPGAs can have significantly lower power consumption. In some cases, FPGAs consume up to 10 times less power than general-purpose processors. [LA18] For instance, a study on FPGA-based ECG signal processing achieved a latency of less than 1 second and power consumption around 28 mW, showing the potential of hardware-accelerated solutions in medical devices. [DCJ+21].

2 Theoretical Considerations

2.1 ECG Interpretation

Our heartbeat is controlled by regular *electric signals* sent by the sinoatrial node (SA node), a group of cells in the heart's right atrium. These signals spread through the heart muscles, causing the atria and then the ventricles to contract, and can be measured on the surface of our skin [fBI23]. A normal ECG pattern, as seen in Figure 2.1, is composed of:

1. `P` wave: This is the first wave and shows how the electrical signal spreads across the atria, causing them to contract and immediately relax.
2. QRS complex: This complex is composed of the `Q`, `R`, and `S` waves, showing the electrical impulse spreading across the ventricles, which then contract.
3. `T` wave: This wave shows how the electrical impulse stops spreading and the ventricles relax.

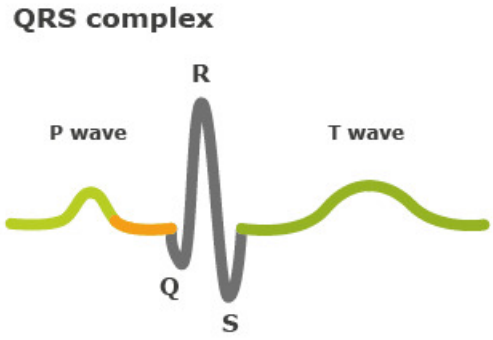

Figure 2.1: ECG waveform [fBI23]

2.2 Discrete and Fast Fourier Transform (DFT and FFT)

The *Discrete Fourier Transform (DFT)* is used to transform a discrete sequence $x[n]$ (such as time domain) into its frequency-domain representation $X[k]$. The DFT is mathematically represented as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \left(\cos\left(\frac{2\pi kn}{N}\right) - j \sin\left(\frac{2\pi kn}{N}\right) \right)$$

This transformation decomposes the input signal $x[n]$ into a sum of cosinusoidal components, where each component corresponds to a particular frequency. The result, $X[k]$, represents the magnitude and phase of each frequency component at index k [OS10].

The *Fast Fourier Transform (FFT)* is an efficient algorithm that rapidly computes *DFT* by factorizing the *DFT* matrix intro a product of mostly zero factors [VL92]. This reduces the complexity of computing the *DFT* from $O(N^2)$ to $O(N \log N)$ where n is the data size.

2.3 AXI4-Stream Protocol

The *AXI4-Stream* protocol, part of the AMBA 4 specification, is used as a standard interface to connect components in high throughput applications like processing pipelines. AXI4-Stream is often used in FPGA designs to connect IP cores such as *FFT* [oCN25].

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of **ACLK**.
ARESETn	Reset source	The global reset signal. **ARESETn** is active-LOW.
TDATA	Sender	The actual data that is passing across the interface.
TVALID	Sender	**TVALID** indicates that the sender provides valid data. The data is transferred when both **TVALID** and **TREADY** are asserted.
TREADY	Receiver	**TREADY** indicates that the receiver can accept the data in the current cycle.

Figure 2.2: Reproduced from [oCN25]

A valid timing behavior works as follows:

1. The sender places valid data on `TDATA` and *high* on `TVALID`.
2. If the receiver is ready to accept data, it asserts `TREADY` and the transfer completes (this is called a *handshake*). The next data word will be sent in the following cycle.
3. Otherwise, the sender must hold `TDATA` and `TVALID` steady waiting for the receiver.

2.4 Cardiac Anomaly

The classification of cardiac function, including normal, bradycardia, tachycardia, and life-threatening conditions, is based on the duration of consecutive R-R intervals. The table below summarizes these thresholds [Ass23]::

Condition	R-R Interval (ms)	Heart Rate (BPM)
Dangerously Low	> 2000 ms	< 30 BPM
Bradycardia	1000 ms – 2000 ms	30 BPM – 60 BPM
Normal Function	600 ms – 1000 ms	60 BPM – 100 BPM
Tachycardia	300 ms – 600 ms	100 BPM – 200 BPM
Dangerously High	< 300 ms	> 200 BPM

Table 2.1: Cardiac function classification based on R-R intervals and BPM, including life-threatening conditions.

3 Design

3.1 Data Processing Pipeline

Figure 3.1 presents the data processing pipeline.

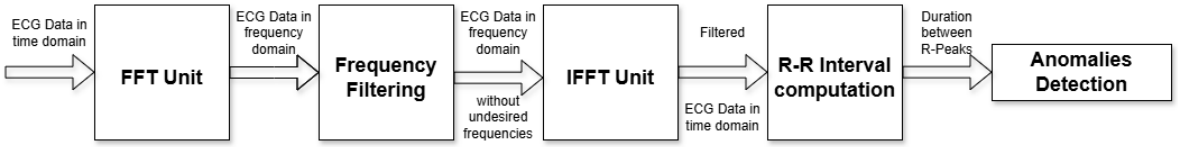

Figure 3.1: Processing pipeline

It consists of *five* main stages.

1. `FFT Unit`: The Fast Fourier Transform algorithm is applied to the raw ECG data, transforming the signal from the *time domain* into the *frequency domain*.

2. `Frequency filtering`: Undesired frequencies, including high-frequency noise and low-frequency baseline wander, are filtered out during this stage.

3. `IFFT Unit`: The filtered ECG data in *frequency* domain are transformed back into *time* domain using the Inverse Fast Fourier Transform (IFFT) to be further processed.

4. `R-R Interval computation`: In this stage, R-Peaks are detected using the filtered data, and the interval between successive peaks is computed. These intervals will be used for anomaly detection.

5. `Anomaly Detection`: Using the previously computed R-R Interval, the system assesses cardiac functioning and identifies potential anomalies, such as bradycardia (slow heart rate) or tachycardia (fast heart rate).

3.2 Undesired frequencies

Removing certain frequency ranges from the ECG signal is essential for correctly interpreting the cardiac function in the later stages of the pipeline. The targeted undesired frequencies are the following:

- **Below 0.5 Hz:** These frequencies cause baseline wander, primarily due to the patient's movement or respiration. Removing this range of frequencies ensures the ECG baseline remains stable [Khe19].

- **Above 50 Hz:** High-frequency noise, such as powerline interference and electromyographic (EMG) noise from muscle activity, must be removed so that the waveform remains clear and free from distortions by non-cardiac-related sources [MCFR19].

Electromyographic (EMG) noise refers to electrical signals generated by muscle activity near the ECG electrodes. It typically occurs in the frequency range above 50 Hz and overlaps with the ECG signal, obscuring diagnostically significant features.

Removing EMG noise is crucial for two reasons:

- Signal clarity: EMG noise can distort the ECG waveform, making it difficult to detect key features like R-peaks, which are necessary for heart rate and rhythm analysis.
- Diagnostic accuracy: The presence of EMG noise can lead to false interpretations, potentially resulting in incorrect diagnoses of cardiac conditions.

By removing these frequency ranges, only clean, diagnostically important data is retained after signal reconstruction.

3.3 R-Peak Detection algorithm

Algorithm 1 R-Peak Detection

Initialize: $adaptive\_threshold \leftarrow INITIAL\_THRESHOLD$
$sample\_count \leftarrow 0$
$curr\_sample \leftarrow X_t$
$prev\_sample, prev\_prev\_sample \leftarrow 0$
while $input\_tvalid = TRUE$ **do**
  if $curr\_sample > prev\_sample$ ***and*** $prev\_sample > prev\_prev\_sample$ **then**
    if $curr\_sample > adaptive\_threshold$ ***and*** $sample\_count > MIN\_SAMPLES$ **then**
      $r\_peak \leftarrow TRUE$
      $sample\_count \leftarrow 0$
      $adaptive\_threshold \leftarrow \frac{adaptive\_threshold + curr\_sample}{2}$
    end
  end
  $prev\_prev\_sample \leftarrow prev\_sample$
  $prev\_sample \leftarrow curr\_sample$
  $sample\_count \leftarrow sample\_count + 1$
end

The R-Peak detection algorithm implemented here uses a *local maxima detection* approach, which is simple to implement, efficient, and resource-conscious making it ideal for real-time processing on FPGA. By comparing the current sample with the two previous samples, it identifies peaks in the signal while dynamically adapting its threshold. This eliminates the need for complex mathematical computations.

4 Implementation

4.1 Implementation details

The FPGA implementation is built on DILIGENT Nexys A7-100T, which is based on the latest Artix-7™ family from Xilinx®. Considering the board's specifications [Dig], the following system parameters were selected. These specifications offer sufficient accuracy for the ECG signal processing while ensuring the system operates within the capabilities of the board:

Specification	**Value**
FPGA Part Number	XC7A100T-1CSG324C
Look-up Tables (LUTs)	63,400
Flip-Flops	126,800
ECG data size	16 bits
FFT size	TBD
Sapling frequency	TBD
ECG Data type	Fixed Point

Table 4.1: System Specifications

4.2 FFT IP Core Configuration

The FFT IP Core used in the system is the **Fast Fourier Transform v9.1 LogiCORE IP** from Xilinx. This component is responsible for both *FFT* and *IFFT*, depending on the value of input port `S_AXIS_CONFIG`.

The selected configurations meet the system's requirements while simplifying the core's integration. Although not the fastest or most resource-efficient setup, these choices make the component well-suited for simulation and prototyping purposes.

Configuration	**Value**
Number of Channels	1
Transform Length	512
Target Clock Frequency (MHz)	TBD
Target Data Throughput (MSPS)	TBD
Data Format	Fixed Point
Scaling Options	Scaled
Rounding Modes	Truncation
Input Data Width	16
Phase Factor Width	16
Outputting Order	Natural Order

Table 4.2: IP Core Configuration

4.3 Elaborated Design

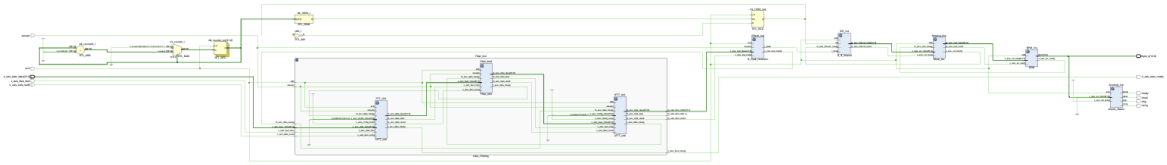

Figure 4.1: ECG Data before and after filtering

4.4 Entities

4.4.1 xFFT_Unit

xFFT_Unit

```
COMPONENT xFFT_unti
  PORT (
    aclk : IN STD_LOGIC;
    aresetn : IN STD_LOGIC;
    s_axis_config_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    s_axis_config_tvalid : IN STD_LOGIC;
    s_axis_config_tready : OUT STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : OUT STD_LOGIC;
    m_axis_data_tready : IN STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC;
    event_frame_started : OUT STD_LOGIC;
    event_tlast_unexpected : OUT STD_LOGIC;
    event_tlast_missing : OUT STD_LOGIC;
    event_status_channel_halt : OUT STD_LOGIC;
    event_data_in_channel_halt : OUT STD_LOGIC;
    event_data_out_channel_halt : OUT STD_LOGIC
  );
END COMPONENT;
```

The `xFFT_Unit` is generated by Vivado based on the IP Core configured in subsection 4.2. It implements both FFT and IFFT algorithms, depending on `s_axis_config_tdata`. The unit has multiple ports but the ones used in our VHDL implementation can be found in table 4.3.

Because we are using two different xFFT units, one for FFT and another for IFFT, `s_axis_config_tdata` (on 16 bits) should take the following **constant** values:

1. `x0001` - for FFT.

Signal	Description
aclk aresetn	Clock input , for synchronous processing. Active-low reset input.
s_axis_config_tdata s_axis_config_tvalid s_axis_config_tready	16-bit input data, for configuration purposes. Input signal, indicates when the configuration data is valid. Output signal, indicates when the module is ready to accept configuration data.
s_axis_data_tdata s_axis_data_tvalid s_axis_data_tready s_axis_data_tlast	32-bit input data, representing the data stream. Input signal, indicates when the data on s_axis_data_tdata is valid. Output signal, indicates when the module is ready to accept data. Input signal, indicates the last data in the stream.
m_axis_data_tdata m_axis_data_tvalid m_axis_data_tready	32-bit output data, representing the processed data. Output signal, indicates when the output data on m_axis_data_tdata is valid. Input signal, indicates when the downstream module is ready to accept output data.

Table 4.3: I/O Signals for the xFFT_Unit

2. x0000 - for IFFT.

The input data on port: s_axis_data_tdata is composed of

1. [31:16] - the 16 higher bits are used for the real part of the input, represented by the ECG data in time domain for FFT and in frequency domain for IFFT.

2. [15:0] - the 16 lower bits are used for the imaginary part of the input (and will always be x0000 for FFT Unit).

The output m_axis_data_tdata represents:

1. for FFT - amplitude of frequency bins in **natural** (ascending) **order**.

2. for IFFT - filtered ECG signal in time domain.

4.4.2 Filtering_Unit

Filtering Unit

```
entity Filter_Unit is
  Port (
    aclk          : in  STD_LOGIC;  -- Clock
    aresetn       : in  STD_LOGIC;  -- Active-low reset
    -- Input AXI Stream
    s_axis_data_tdata : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
       -- 16-bit real + 16-bit imaginary
    s_axis_data_tvalid: in  STD_LOGIC;
    s_axis_data_tready: out STD_LOGIC;
    s_axis_data_tlast : in  STD_LOGIC;
    -- Output AXI Stream
    m_axis_data_tdata : out STD_LOGIC_VECTOR(31 DOWNTO 0);
       -- Filtered frequency-domain data
    m_axis_data_tvalid: out STD_LOGIC;
    m_axis_data_tready: in  STD_LOGIC;
    m_axis_data_tlast : out STD_LOGIC
  );
end Filter_Unit;
```

Signal	Description
aclk	Clock input (synchronous to the filter logic)
aresetn	Active-low reset input
s_axis_data_tdata	Input AXI Stream data (frequency-domain data, 32-bit wide)
s_axis_data_tvalid	Input AXI Stream valid signal (data is valid when high)
s_axis_data_tready	Output AXI Stream ready signal (ready to accept new data)
s_axis_data_tlast	Input AXI Stream last signal (indicates the last data in the stream)
m_axis_data_tdata	Output AXI Stream data (filtered frequency-domain data, 32-bit wide)
m_axis_data_tvalid	Output AXI Stream valid signal (data is valid when high)
m_axis_data_tready	Input AXI Stream ready signal (indicates when data can be accepted)
m_axis_data_tlast	Output AXI Stream last signal (indicates the last data in the stream)

Table 4.4: I/O Signals for the Filter Unit

The **Filter_Unit** VHDL entity is designed to filter out unwanted frequencies from a signal in the frequency domain using a simple band-pass filter. The input data,

s_axis_data_tdata, represents the magnitude of each frequency bin, entering the component in ascending order. The filter processes these frequency bins by counting them and setting all outputs to zero, except for those falling within the defined range between LOW_FREQ_INDEX and HIGH_FREQ_INDEX. This effectively isolates the desired frequency components while removing the unwanted ones.

4.4.3 R_Peak_Detection

R_Peak_Detection

```
entity R_Peak_Detection is
    Port (
        aclk : in STD_LOGIC;
        aresetn : in STD_LOGIC;
        -- Input AXI Stream
        s_axis_ecg_tvalid : in STD_LOGIC;
        s_axis_ecg_tready : out STD_LOGIC;
        s_axis_ecg_tdata : in STD_LOGIC_VECTOR(31 downto 0)
            ;
        -- Output
        r_peak : out STD_LOGIC
    );
end R_Peak_Detection;
```

Signal	Description
aclk	Clock input (input), synchronous to the R-Peak detection logic.
aresetn	Active-low reset input (input).
s_axis_ecg_tvalid	Input AXI Stream valid signal, indicates valid ECG data.
s_axis_ecg_tready	Output AXI Stream ready signal, ready to accept ECG data.
s_axis_ecg_tdata	Input ECG data (32-bit data, input) representing ECG sample.
r_peak	Output signal indicating whether an R-peak is detected, high when detected.

Table 4.5: I/O Signals for the R-Peak Detection Unit

The **R_Peak_Detection** entity implements the algorithm described in Algorithm 1, using the *AXI4-Stream* protocol to receive filtered ECG data in the time domain. The entity sets the r_peak signal to HIGH when an R-Peak is detected. The INITIAL_THRESHOLD is set to 0x00100000 (32 in decimal) to ensure the detection starts at a reasonable level based on signal amplitude. At each step, a constant DRIFT value of 0x00000001 (1 in decimal) is subtracted from the current threshold. This gradual reducing compensates maintains detection accuracy by ensuring the threshold remains sensitive to dynamic signal changes.

4.4.4 R_R_Interval

R_Peak_Interval

```
entity R_R_Interval is
    port (
        aclk : in std_logic; -- axi4 clock
        clk : in std_logic; -- 100 Hz ( 10 ms )
        r_peak: in std_logic; -- 1 when a R peak is
           detected
        m_axis_interval_tvalid : OUT STD_LOGIC;
        m_axis_interval_tready : IN STD_LOGIC;
        m_axis_interval_tdata : OUT STD_LOGIC_VECTOR(31
           DOWNTO 0)
    );
end R_R_Interval;
```

Signal Name	Description
aclk	AXI4 clock signal (input).
r_peak	Input signal, set to '1' when an R-peak is detected in the ECG signal.
m_axis_interval_tvalid	Output signal, '1' when the interval data is valid and ready to be sent.
m_axis_interval_tready	Input signal, indicates if the receiver is ready to accept the interval data.
m_axis_interval_tdata	Output signal, holds the computed R-R interval value (in 100 Hz clock ticks).

Table 4.6: I/O Signals for the R_R_Interval Entity

The R_R_Interval entity is designed to compute the time between consecutive R-peaks in an ECG signal. The operation is as follows:

- S_READ, the module waits for an R-peak to be detected, then starts counting the time between consecutive peaks if it didn't already start doing so, otherwise, it stop the counter and goes to S_WRITE state.
- In S_WRITE, the R-R interval value is output through m_axis_interval_tdata, and only goes back to S_READ when the receiver is ready (i.e., m_axis_interval_tready is HIGH).

4.4.5 Move_Avr

Move_Avr

```
entity Move_Avr is
    Generic (
        WINDOW_SIZE : integer := 8
    );
    Port (
        aclk : IN STD_LOGIC;
        s_axis_val_tvalid : IN STD_LOGIC;
        s_axis_val_tready : OUT STD_LOGIC;
        s_axis_val_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
           ;
        m_axis_sum_tvalid : OUT STD_LOGIC;
        m_axis_sum_tready : IN STD_LOGIC;
        m_axis_sum_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO
           0)
    );
end Move_Avr;
```

Port Name	Description
aclk	Clock signal (input) for synchronous processing.
s_axis_val_tvalid	Input signal indicating when input data on s_axis_val_tdata is valid.
s_axis_val_tready	Output signal indicating when the module is ready to accept input data.
s_axis_val_tdata	32-bit input data (input) representing the next value in the data stream.
m_axis_sum_tvalid	Output signal indicating when the output data on m_axis_sum_tdata is valid.
m_axis_sum_tready	Input signal indicating when the downstream module is ready to accept output data.
m_axis_sum_tdata	32-bit output data representing the computed moving average (output).

Table 4.7: Port descriptions for the Move_Avr entity.

Generic Name	Type	Description
WINDOW_SIZE	Integer	Number of samples in the moving average window (default: 8).

Table 4.8: Generic parameters for the Move_Avr entity.

Using a sliding window approach, the Move_Avr entity implements a *moving average filter*. It computes the average of the most recent R-R Intervals Its role is to ensure

steady values in case of noise within the pipeline components to avoid detection of false R-Peaks.

The moving average is calculated using a running sum:

$$sum_{new} = sum_{old} - x[index] + input$$

$$average = \frac{sum_{new}}{\texttt{WINDOW SIZE}}$$

4.4.6 BPM

BPM

```
entity BPM is
    port (
        aclk : IN STD_LOGIC;
        s_axis_val_tvalid : IN STD_LOGIC;
        s_axis_val_tready : OUT STD_LOGIC;
        s_axis_val_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
           ;
        bpm: out STD_LOGIC_VECTOR(15 downto 0)
    );
end BPM;
```

Signal Name	Description
aclk	Clock input, for synchronous processing.
s_axis_val_tvalid	Input signal, indicates when the input data on s_axis_val_tdata is valid.
s_axis_val_tready	Output signal, indicates when the module is ready to accept input data.
s_axis_val_tdata	32-bit input data (input), representing the next value in the data stream.
bpm	16-bit output, holds the computed beats per minute (BPM) value.

Table 4.9: I/O Signals for the BPM Entity

BPM unit computes the no. of *beats per minute* (BPM) based on the time interval between consecutive R-R peaks, measured in milliseconds (ms). The R-R interval represents the duration between successive heartbeats.

The heart rate in beats per minute is calculated by finding the number of heartbeats that occur in one minute, based on the time taken for one heartbeat.

$$\left.\begin{aligned} BPM &= \frac{1}{interval_{min}} \\ 1min &= 60000ms \end{aligned}\right\} \implies BPM = \frac{60000}{interval_{ms}}$$

The resulting BPM is displayed on a 7-segment display, providing a real-time visual indication of the heart rate.

4.4.7 Anorm_Detect

BPM

```
entity Anorm_Detect is
    port (
        aclk : IN STD_LOGIC;
        s_axis_val_tvalid : IN STD_LOGIC;
        s_axis_val_tready : OUT STD_LOGIC;
        s_axis_val_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
           ;
        brady: out std_logic; -- Bradycardia
        tachy: out std_logic; -- Tachycardia
        dng: out std_logic; -- Dangerous heart function
        dead: out std_logic -- 0 BPM
    );
end Anorm_Detect;
```

Signal Name	Description
aclk	Clock input (input), for synchronous processing.
s_axis_val_tvalid	Input signal, indicates when the input data on s_axis_val_tdata is valid.
s_axis_val_tready	Output signal, indicates when the module is ready to accept input data.
s_axis_val_tdata	32-bit input data (input), representing the next value in the data stream.
brady	Output signal, indicates Bradycardia condition.
tachy	Output signal, indicates Tachycardia condition.
dng	Output signal, indicates Dangerous heart function.
dead	Output signal, indicates 0 BPM condition.

Table 4.10: I/O Signals for the Anorm_Detect Entity

The Anorm_Detect unit detect cardiac problems based on the time interval between consecutive R-R peaks, measured in milliseconds (ms). The thresholds from Table 2.1 are used to classify different anomalies. Each condition is mapped to a LED on the FPGA.

This unit operates in parallel with the BPM unit, using the R-R time interval instead of the calculated *no. of beats per minute*. By avoiding the latency introduced by BPM computation, the Anorm_Detect unit provides immediate detection results.

5 Results

5.1 Artificial Data Filtering Tests

To test the filtering module composed of `FFT Unit, Filtering Unit` and `IFFT Unit`, artificial data was generated using *numpy* Python library. The test data is a sum of three cosinusoidal components each defined by its amplitude and frequency, it can be expressed as:

$$x(t) = \sum_{i=1}^{N} A_i \cdot \sin(2\pi f_i t)$$

where:

- N is the number of frequency components,
- A_i is the amplitude of the i-th component,
- f_i is the frequency of the i-th component (in Hz).

The specific frequencies and amplitudes used in this example are represented as vectors:

$$\mathbf{f} = \begin{bmatrix} 0.1 \\ 5 \\ 100 \end{bmatrix} \text{ Hz}, \quad \mathbf{A} = \begin{bmatrix} 5 \\ 0.5 \\ 0.2 \end{bmatrix}$$

Substituting these values into the formula, the signal becomes:

$$x(t) = 5 \cdot \sin(2\pi \cdot 0.1 \cdot t) + 0.5 \cdot \sin(2\pi \cdot 2 \cdot t) + 0.2 \cdot \cos(2\pi \cdot 100 \cdot t)$$

where t is the time vector interval $[0, \text{duration}]$.

Among the three components of the input signal, only the second one falls within the desired range of $[0.5Hz, 50Hz]$. As a result, it is the only component that remains visible in the output signal after filtering.

The three graphs in figure 5.1 are:

1. The input signal consisting of three components.
2. FFT Output: The frequency spectrum showing the amplitudes of all components as bins.
3. Filtered Ouput: Filtered: The reconstructed signal, with only the component (with a frequency of $2Hz$] is visible.

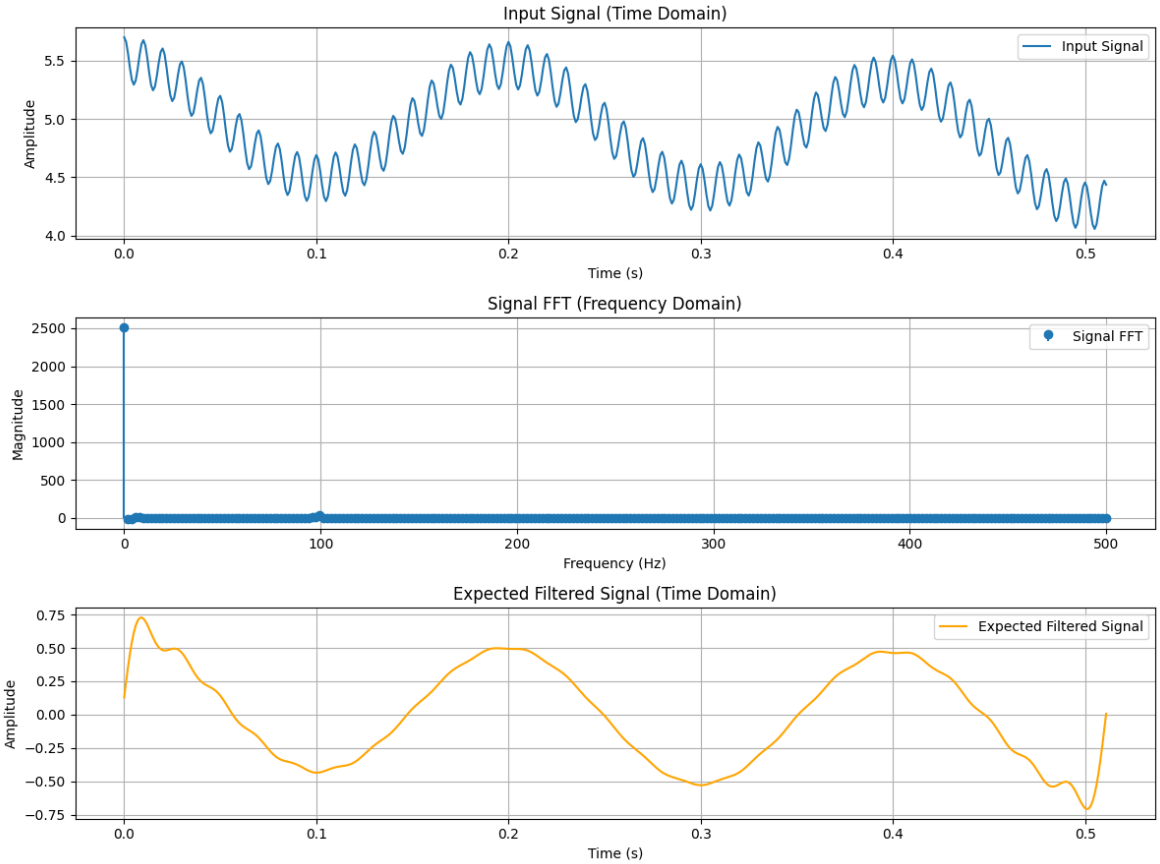

Figure 5.1: Sample Data Signals

A testbench was developed for the *Filtering Stage* of the pipeline to analyze the system's filtering response using these data before progressing to a more complex ECG dataset. The waveform output is seen in figure 5.2.

reall.png

Figure 5.2: Filtered Sample Data Signals

5.2 Real ECG Data Filtering Tests

Using real, unfiltered ECG data obtained from Physionet, the *Filtering Stage* was tested once again. The output shows a cleaner, more readable signal with clear separation of the `Q`, `R`, and `S` waves. These filtered data are now ready for subsequent processing to detect R-peaks.

The ECG data processing involves two key transformations:

Original Data Format: The ECG signal is stored in .dat format using the WFDB standard (MIT-BIH database), where:

- Data is stored as continuous samples
- Each sample represents amplitude at a specific time point
- Sampling frequency: $f_s = 360$ Hz

Data Transformation for FPGA: The signal is transformed for FPGA processing:

- Normalized and scaled to 16-bit integer range: $[-32767, 32767]$
- Formula: $signal_{scaled} = \left\lfloor \frac{signal}{max(|signal|)} \times 32767 \right\rfloor$
- Each sample formatted as 32-bit hex: `0000XXXX`
  - Upper 16 bits: Padded with zeros
  - Lower 16 bits: Signal value in hex

Example: A sample value of 16384 would be represented as `00004000`

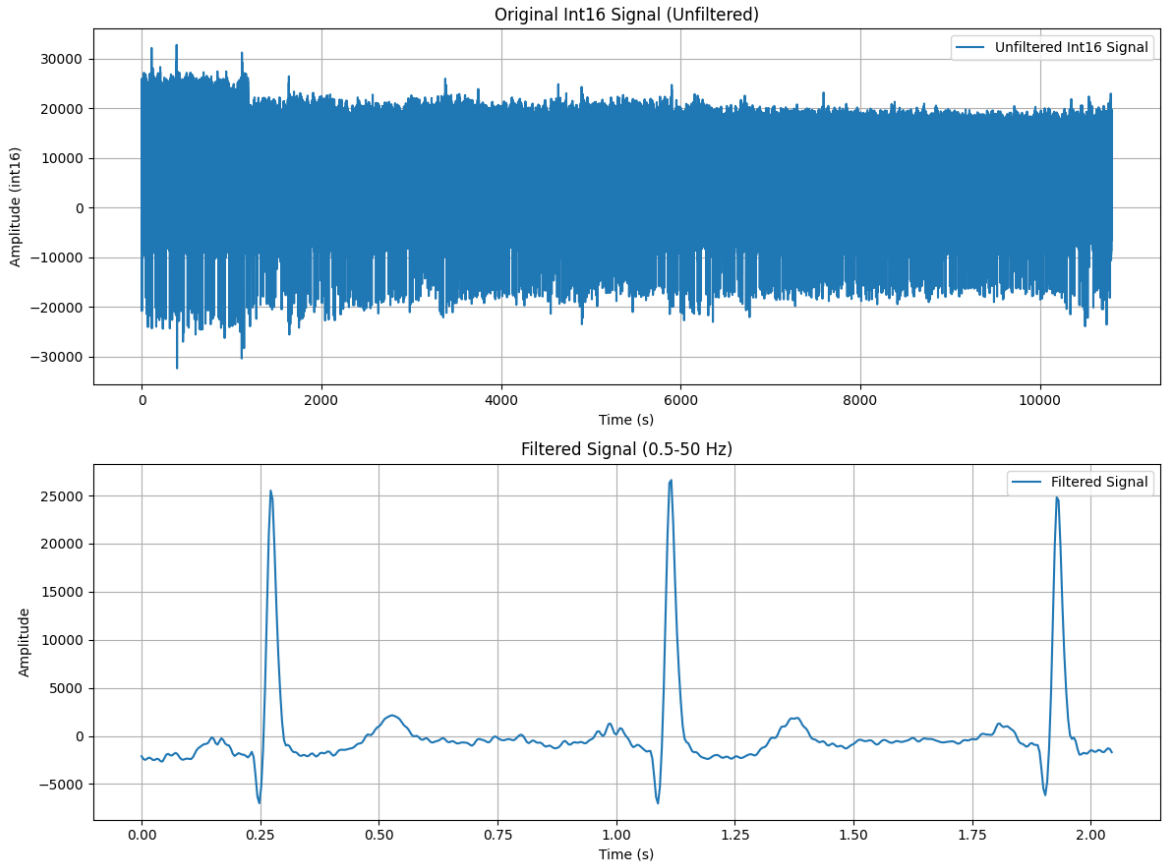

Figure 5.3: ECG Data before and after filtering

5.3 Implementation obtained parameters

5.3.1 FFT IP Core

The FFT (Fast Fourier Transform) IP core used in this design has the transform size fixed at 512, with a 16-bit output data width. Each FFT core requires 12 DSP slices and 4 Block RAMs. Two FFT IP cores are employed in the system for parallel processing.

The table below summarizes the resource utilization for a single FFT IP core:

Resource	Usage
DSP Slices	12
Block RAMs	4

Table 5.1: Resource Estimates for a Single FFT IP Core

The FFT IP core operates with a latency of 16,490 cycles for a transform length of 512. At a 100 MHz clock frequency, this corresponds to a latency of 16.49 microseconds. The performance metrics are summarized in the table below:

Metric	Value
Transform Length	512
Transform Cycles	16,490
Latency (μs)	16.49

Table 5.2: Performance Estimates for a Single FFT IP Core

The usage of two FFT IP cores in this system effectively doubles the resource requirements but enables simultaneous computation of FFTs, which improves the overall data processing throughput. The latency and resource efficiency of the IP core are well-suited for real-time ECG processing applications.

5.3.2 FPGA implementation

Table 5.3 summarizes the resource utilization of the top module and its submodules. The analysis includes the number of Slice LUTs, Slice Registers, Block RAM Tiles, and DSPs used in the design. The *top_module* serves as the main design entity, while the submodules handle specific functionalities such as filtering, R-peak detection, and anomaly detection.

The *Filter_inst* module is the most resource-intensive, utilizing multiple FFT units for efficient data processing. Each FFT unit contributes significantly to the Slice LUTs, Slice Registers, Block RAM Tiles, and DSPs used in the design. Other submodules, such as *MovAvg_inst* and *RPeak_inst*, consume relatively fewer resources, reflecting their specific and less computationally demanding roles.

This breakdown provides valuable insights into the hardware requirements of each component, aiding in optimization and ensuring the design fits within the available FPGA resources.

Name	**Slice LUTs (63400)**	**Slice Registers (126800)**	**Block RAM Tiles (135)**	**DSPs (240)**
top_module	5225	8461	4	24
Anomaly_inst	0	4	0	0
BPM_inst	523	17	0	0
Filter_inst	4459	8144	4	24
FFT_Unit	2170	4039	2	12
Filter_mod	53	66	0	0
IFFT_Unit	2170	4039	2	12
MovAvg_inst	134	68	0	0
RPeak_inst	99	129	0	0
RR_inst	4	66	0	0

Table 5.3: Resource Utilization Report for Top Module and Submodules

6 Summary and Conclusions

The implemented pipeline effectively processes ECG data by filtering unwanted frequencies and preserving critical components essential for cardiac analysis. Designed to address real-time signal processing challenges, the pipeline ensures reliable data transformation through a robust filtering mechanism. This stage eliminates common noise, such as baseline wander and powerline interference, while retaining features like the P wave, QRS complex, and T wave.

Testing with synthetic and real ECG datasets validated the filtering stage's performance. Synthetic data enabled controlled evaluation, while real-world data demonstrated the system's practical effectiveness in producing clean signals with minimal distortion. This ensures the filtered signals are ready for further analysis, such as R-peak detection and cardiac anomaly identification.

The system's real-time capability, achieved through hardware-efficient FFT IP cores, ensures low-latency operation suitable for applications like clinical monitoring or wearable devices. The results confirm the pipeline's ability to deliver clean ECG signals, making it a reliable foundation for downstream tasks.

In conclusion, this pipeline demonstrates robust performance for real-time ECG signal processing, providing a scalable and efficient solution. Future improvements, such as adaptive filtering and machine learning-based anomaly detection, could enhance its applicability further, broadening its impact in healthcare and research.

Bibliography

[Ass23] American Heart Association. All about heart rate (pulse), 2023. Accessed: 2025-01-11. URL: https://www.heart.org/en/healthy-living/fitness/fitness-basics/all-about-heart-rate-pulse.

[DCJ+21] MP Desai, G Caffarena, R Jevtic, DG Márquez, and A Otero. A low-latency, low-power fpga implementation of ecg signal characterization using hermite polynomials. electronics 2021, 10, 2324, 2021.

[Dig] Digilent. Nexys a7 reference manual. https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual. Accessed: 2025-01-11.

[fBI23] National Center for Biotechnology Information. Electrocardiogram (ecg or ekg), June 2023. URL: https://www.ncbi.nlm.nih.gov/books/NBK536878/.

[Khe19] R. Kher. Signal processing techniques for removing noise from ecg signals. 2019.

[LA18] J. W. Lee and J. C. D. S. Aguiar. Power and performance comparison of fpga-based and cpu-based systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1419–1428, 2018. URL: https://ieeexplore.ieee.org/document/8356824, doi:10.1109/TVLSI.2018.2832853.

[MCFR19] J. P. Madeiro, P. C. Cortez, J. M. S. M. Filho, and Priscila Rocha Ferreira Rodrigues. Techniques for noise suppression for ecg signal processing. *Developments and Applications for ECG Signal Processing*, 2019.

[oCN25] Technical University of Cluj-Napoca. Structure of computer systems laboratory 7 text. Technical report, 2025.

[OS10] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing*. Pearson Education, 3rd edition, 2010.

[VL92] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, 1992.