Habib University

Spring 2023

Dhanani School of Science & Engineering

Digital Signal Processing Lab - EE/CS 371L/330L-T7

Project Report

Group Members:

Omer Ali Rastgar

Usman Siddiqui

M. Ali Raza

Mohit Rai

Side Channels and Side Channel Attacks:

A side channel is an unintended information leakage that can be used to infer information about a system or its data. These side channels can arise from various sources such as the physical implementation of a system, the timing of its operations, or the power consumption patterns of its components.
Such side channels pose a security exploitative threat as they can be used to gain access to sensitive or confidential information. These attempts are known as side-channel attacks.

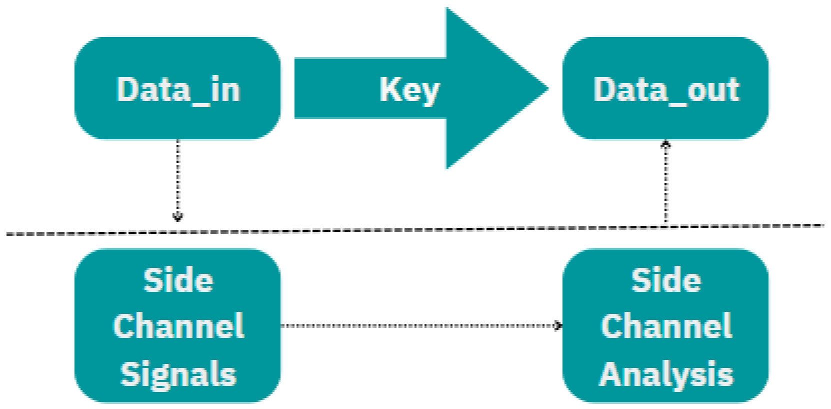


Fig 1. Side Channel Attack

Side channel attacks are particularly dangerous because they can often be carried out without direct access to the system or its data. Instead, an attacker can exploit a side channel remotely, by monitoring the system's behavior over a network or by using specialized equipment to measure its physical properties.

Timing Attacks:

Timing attacks are a type of side-channel attack that exploits the timing behavior of a system to extract sensitive information. Timing attacks involve measuring the time it takes for a system to perform a particular operation, such as a cryptographic operation, and using that information to infer secret information.
For example, let's say a cryptographic system takes longer to process inputs that are closer to the correct secret key. An attacker can exploit this timing difference to deduce information about the secret key by measuring the execution time of the system for different inputs.
Another example of a timing attack is when an attacker sends a series of carefully crafted requests to a web server and measures the response time. By analyzing the response time, the attacker can infer information about the server's internal state, such as the presence of a particular user account or the content of a secret file.

Timing attacks can be particularly difficult to defend against because they often involve subtle variations in the timing of system operations that are difficult to detect or correct. To mitigate the risk of timing attacks, developers can use techniques such as randomizing the timing of system operations, ensuring that cryptographic operations take a constant amount of time, and limiting the amount of information leaked through error messages or other side channels.

Example of a Timing Attack:

To better understand the significance of timing attack, let's go through a basic example - a sample piece of code for string comparison, which is shown as follows:

```
def string_compare(s1,s2):
    if len(s1) != len(s2):
        return False
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            return False
    return True
```

In the function above, if the lengths of the two strings are equal, we first compare the lengths of the two strings before comparing the individual characters. If we run into two distinct characters when iterating through the loop, we return false. If no difference is discovered, we return true.

Although it appears to be perfect, this implementation has a problem that most people don't instantly notice. Let's consider a situation where a website (or any other app, for that matter) might utilize this function to authenticate users.

Let's imagine that one of the two strings, s1, contains the user's inputted password. The second string, s2, would hold the user's database-retrieved correct password (or, in realistic implementations, its corresponding hash value).
If the needed password were a ten-digit integer, it would require a brute force assault with 1010 potential combinations to find it.

However, this is where it becomes interesting: assume that the attacker begins by guessing the first digit, starting with 0000000000, of the right password, which is 5263987149. He counts the number of seconds till the system responds false. He notes that the system takes a little bit longer to answer (x + x) when he attempts 5000000000 after receiving the same response time x for the first five guesses (i.e., from 0000000000 to 4000000000). This is so that the for loop does not return false on the first iteration since the initial digit of the user-inputted password and the stored password are the same. As a result, the loop completes another iteration, adding time (x). The attacker is aware that his current initial digit is right because of this. By looking at the pattern of x, he may use the same method to determine the remaining numbers. By using this, the attacker would only need a maximum of 10*10 attempts to discover the right password as opposed to 1010 combinations when attempting to brute force it.

In order to correct this problem, we must stop the execution-time disclosure of pattern information that would otherwise allow an attacker to determine which character in the array varies from the rest. This indicates that, regardless of the amount of valid or incorrect components included in the two strings, the loop must be executed a certain number of times.

```
def string_compare(s1,s2):
    if len(s1) != len(s2):
        return False
    flag = True
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            flag = False
    return flag
```

To achieve that, we create a variable called flag that is set to True after comparing the lengths of the strings. The identical if statement as previously is contained within the for loop, but this time we keep the boolean value in a flag variable rather than returning it as soon as a bad digit is encountered. This makes sure that no matter how many numbers match, the loop iterates over every element of the array and takes a certain amount of time. Therefore, the relevant function is no longer vulnerable to a timing-attack.

Although actual implementations wouldn't be as simple to exploit (and block) as those depicted above, this little function captures the idea of a fundamental timing-attack and aids in a clear understanding of it.

Power Analysis Side-Channel Attacks:

Power analysis side-channel attacks are a type of security vulnerability that exploit weaknesses in cryptographic implementations by analyzing power consumption patterns.

In a power analysis attack, an attacker measures the power consumption of a device while it performs cryptographic operations, such as encryption or decryption. By analyzing the power consumption patterns, an attacker can infer information about the secret key being used, and potentially even recover the entire key.

Power analysis side-channel attacks involve analyzing power consumption patterns to infer useful data about the cryptographic operations being performed by a device. The level of power consumption is correlated with the operations being performed, allowing an attacker to monitor the device's power rails during operation to detect current draw or fluctuations in voltage.

By performing statistical analysis on the power consumption data, an attacker can correlate their findings to information about the device's cryptographic operations, potentially enabling the recovery of secret keys. Therefore, it is important to design cryptographic implementations with countermeasures to prevent power analysis attacks, particularly in embedded systems where resources may be limited.

Hardware cryptographic devices are specialized processors that are optimized for performing cryptographic operations such as encryption and decryption. They are commonly used in high-security applications such as banking, military, and government environments to protect sensitive data.

Smart cards, such as credit cards and SIM cards, are examples of hardware cryptographic devices that are widely used in everyday life. These cards contain a secure chip that stores sensitive data and performs cryptographic operations to protect that data.

Any device that runs or houses some kind of secure process or data can also be considered a hardware cryptographic device. Examples include secure USB drives, hardware security modules, and encryption appliances.

Cryptography is the science of using mathematical algorithms to encrypt and decrypt data. It is a critical component of many security systems and is used to protect sensitive information such as financial transactions, personal data, and military secrets. Hardware cryptographic devices play a key role in ensuring the security of these systems by providing specialized hardware that can perform cryptographic operations quickly and securely.

The RSA Algorithm:

The RSA algorithm is a type of cryptographic algorithm that is commonly used to secure digital communications. This algorithm involves using two keys - a public key for encryption and a private key for decryption. The public key is used to encrypt the data, while the private key is used to decrypt it. The RSA algorithm is widely used in a variety of applications such as secure email, online banking, and digital signatures. By using this algorithm, organizations can securely transmit and store sensitive information while protecting it from unauthorized access.

When encrypting:

$$c_i = b_i^e \text{ modulo some integer } N.$$

When Decrypting::

$$b_i = c_i^d \text{ modulo some integer } N.$$

Here, the circled bits are the most computationally intensive, hence leave the largest signature to be backtracked from.

Brute Force Approach:

The most obvious way to go about it:

$$a^b = a * a * a * a.....(b\ times)$$

Unfortunately, this has an unacceptably high computational complexity:

$$O(b)$$

Binary Exponentiation:

But, there is also a better way to do it. Binary Exponentiation is a technique used to calculate the power of a number, in an efficient way, by representing the power in binary form and breaking it down into a sum of powers of two. For example, if we consider the base 5, raised to the power 12.

12=1100

$$12 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$$

This way, the final answer can be computed by multiplying the number raised to each power of two that makes up the sum. To further optimize this method, once we have computed the value of the number raised to a power of two, we can use that result to calculate the next power of two, and so on until we reach the final power. This technique reduces the number of multiplication operations required and therefore speeds up the calculation process.

$$5^{12} = 5^{2^3+2^2} = 5^{2^3} * 5^{2^2}$$

$$5^{12} = 5^8 * 5^4$$

$$5^1 = 5$$

$$5^2 = (5^1)^2 = 5^2 = 25$$

$$5^4 = (5^2)^2 = 25^2 = 625$$

$$5^8 = (5^4)^2 = 625^2 = 390625$$

$$5^{12} = 5^4 * 5^8 = 625 * 390625 = 244140625$$

Understanding the spectra:

Include slide 18

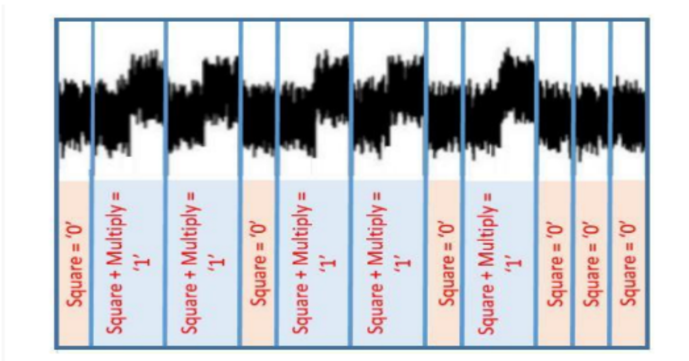
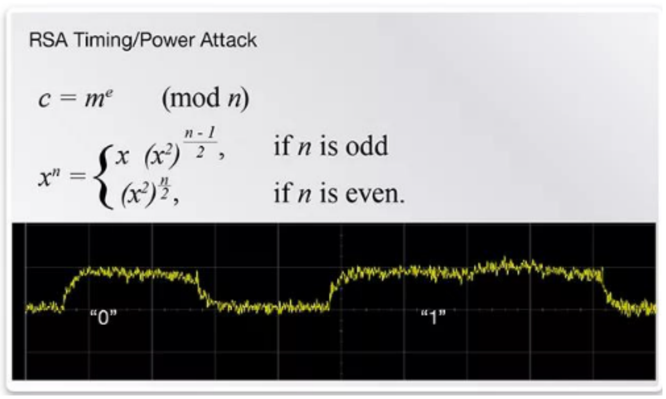


Figure 4. Power trace of a portion of an RSA exponentiation operation.



RSA Timing/Power Attack

$$c = m^e \quad (\text{mod } n)$$

$$x^n = \begin{cases} x\ (x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

Hence, from the given picture, we can observe that from the given key for the operation being performed on the data, the attacker could undermine the RSA operations on the chip, which could lead to information being leaked.

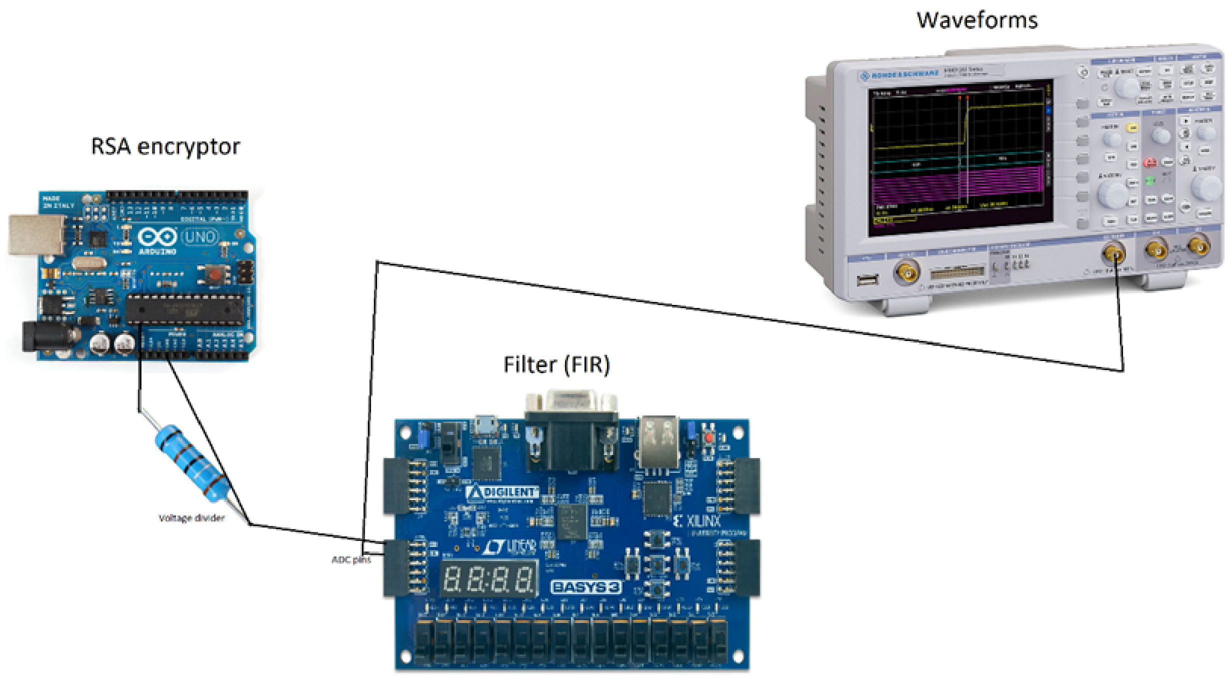
Other Power Analysis Side-Channel Attacks:

Other power analysis side-channel attacks include Simple Power Analysis (SPA) and Differential Power Analysis (DPA). SPA works in idealized scenarios but is less effective in real-world situations where there may be multiple processes running and noisy power signals.

DPA, on the other hand, uses statistical methods to analyze sets of measurements to identify data-dependent correlations. To perform DPA, an attacker collects traces of power consumption during cryptographic operations, and then splits the traces into subsets. The attacker then calculates the average power consumption for each subset and identifies correlations between the subsets.

A large enough dataset is required for DPA to be effective, but once this is obtained, correlations between subsets will always appear. This allows an attacker to gain insights into the cryptographic operations being performed and potentially recover secret keys. It is important to protect against power analysis attacks by designing cryptographic implementations that minimize power consumption variations and incorporate countermeasures to prevent power analysis attacks.

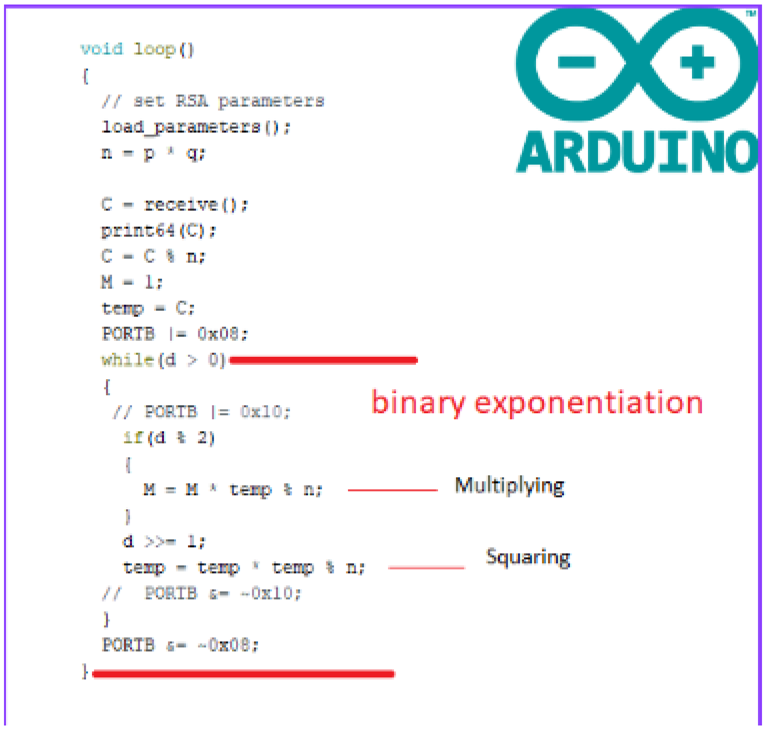
Items required for Simple Power Analysis Side-Channel Attack:



1. Identify Pin 7 on the Arduino board as the power pin that you want to measure the voltage of. This should be a pin that supplies power to the digital or analog circuitry that you are interested in analyzing.

2. Connect a voltage divider to Pin 7 of the Arduino board. The voltage divider should consist of two resistors in series, with one resistor connected to Pin 7 and the other resistor connected to ground. The resistor values should be chosen such that the output voltage of the divider is proportional to the current flowing through Pin 7. In this case, use a 100-ohm resistor for the resistor connected to Pin 7.
3. Connect a low-pass filter to the output of the voltage divider. The filter should be designed to remove any high-frequency noise or ripple from the voltage signal, and should have a cutoff frequency of 1000 Hz or lower.
4. Connect an oscilloscope to the output of the filter. The oscilloscope should be capable of measuring AC signals, and should have a bandwidth that is sufficient to capture the highest frequency component of interest in the voltage signal. A common choice for the oscilloscope bandwidth is 100 MHz or higher.
5. Observe the voltage signal on the oscilloscope. The voltage signal should be a DC signal that is proportional to the current flowing through Pin 7 of the Arduino board. You may need to adjust the gain and timebase settings on the oscilloscope to obtain a clear and stable waveform. In this case, the voltage signal is 4.8 V.
6. Calculate the power consumption using the voltage signal. Since the voltage signal is proportional to the current flowing through Pin 7, you can use Ohm's Law to calculate the power consumption as P = V^2/R, where V is the measured voltage and R is the resistance of the resistor connected to Pin 7. In this case, the power consumption is P = (4.8 V)^2 / 100 ohms = 0.23 W.

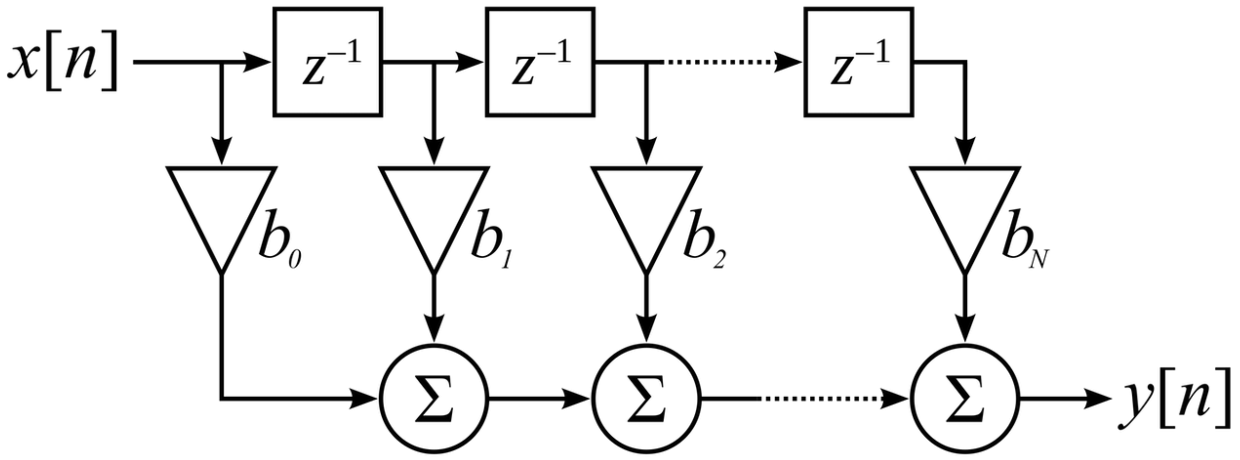
Behaviour modeling for RSA encryption using arduino:



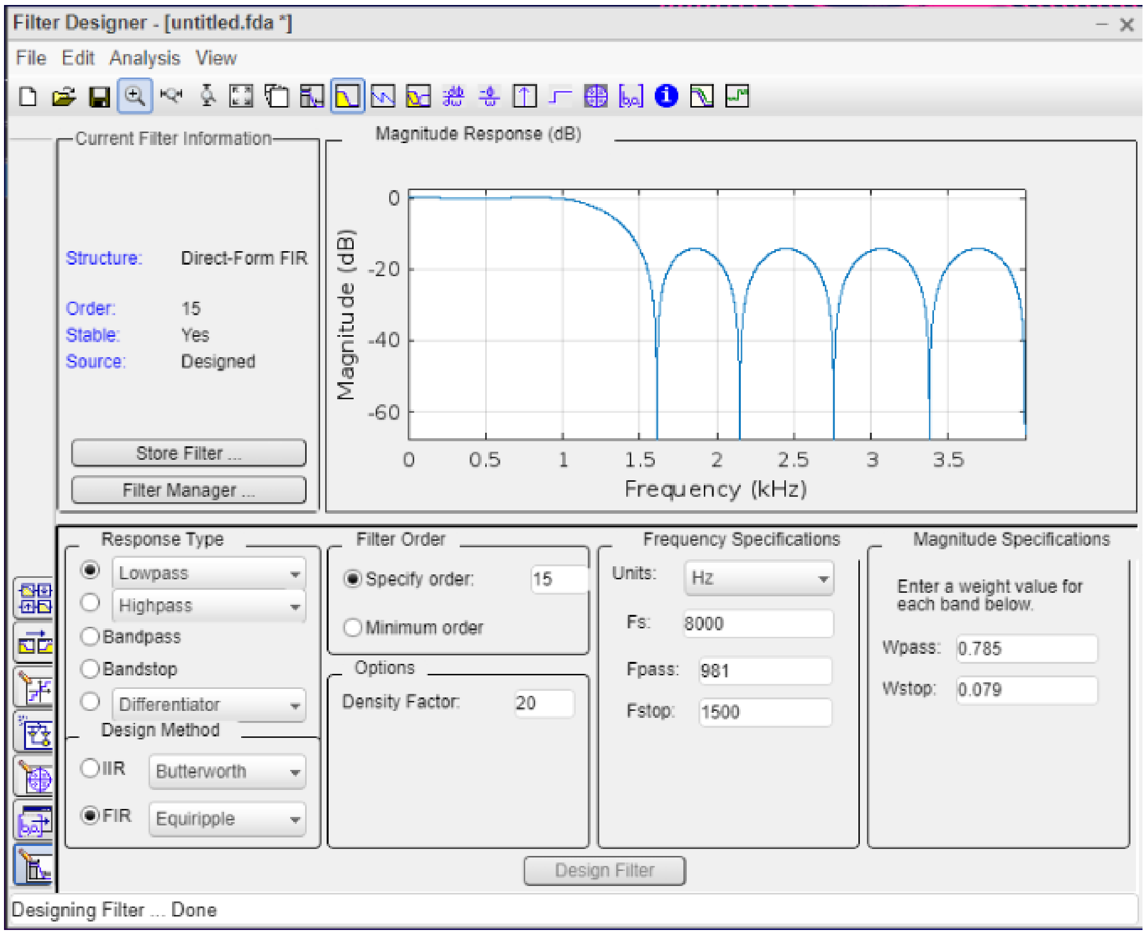
```
void loop()
{
  // set RSA parameters
  load_parameters();
  n = p * q;

  C = receive();
  print64(C);
  C = C % n;
  M = 1;
  temp = C;
  PORTB |= 0x08;
  while(d > 0)
  {
   // PORTB |= 0x10;
    if(d % 2)
    {
      M = M * temp % n;
    }
    d >>= 1;
    temp = temp * temp % n;
  //  PORTB &= ~0x10;
  }
  PORTB &= ~0x08;
}
```

Filter Components:

- A circular buffer to clock each sample into that properly accounts for the delays of the serial input
- Multipliers for each of the taps' coefficient value
- The accumulator registers the summing result from each tap's output.



Filter Design using Matlab:



Structure: Direct-Form FIR
Order: 15
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Filter Order: [✓ Specify order: 15] Minimum order

Options: Density Factor: 20

Frequency Specifications: Units: Hz; Fs: 8000; Fpass: 981; Fstop: 1500

Magnitude Specifications: Enter a weight value for each band below. Wpass: 0.785; Wstop: 0.079

Design Method: IIR Butterworth [✓ FIR Equiripple]

Designing Filter ... Done

1. Open the Filter Designer by typing "fdesign" in the MATLAB command window and pressing Enter. This will open the Filter Designer app.
2. In the app, select "FIR" as the filter type and "Lowpass" as the filter design method.
3. Specify the filter specifications such as the passband frequency, stopband frequency, passband ripple, and stopband attenuation. You can do this by either typing in the values or using the sliders provided.
4. Choose a filter design method. For example, you can select "Equiripple" if you want to design a filter with a very steep transition band.
5. Click on the "Design Filter" button to generate the filter coefficients and plot the frequency response of the filter.

6. To export the filter coefficients to the MATLAB workspace, click on the "Export Filter Coefficients" button and select "MATLAB" as the output format. This will generate a MATLAB script that you can copy and paste into your own MATLAB code.

Choice of Design Parameters:

Low-Pass

We have used a low pass filter because the signals that we need to use have a known frequency, i.e., 1000 Hz or lower . And we also know that the high frequency components are noise, so we need a low frequency component, and block out high frequency ones.

FIR (Equiripple)

An equiripple finite impulse response (FIR) filter is a type of digital filter that has a linear phase response and a frequency response that is equiripple in both the passband and stopband regions. Equiripple FIR filters are often preferred for applications that require a high degree of frequency selectivity and low passband and stopband ripple.

In the context of the current project, an equiripple FIR filter is suitable because it provides a high level of attenuation of high frequency noise while preserving the desired signal. With a high order of 15 and a density factor of 12, the filter is able to achieve a high level of accuracy and precision, which is important for applications that require reliable signal processing, such as medical or scientific purposes. Furthermore, the equiripple nature of the filter ensures that the filter's frequency response is very close to the desired response, with very little deviation or distortion, making it a suitable choice for this application.

Order=15

This determines the number of coefficients in the final equation describing the filter.

$$y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k)$$

In the above equation, the values h(k) represent the equations for all A. When the filter is run and done, we get the following equation, describing our filter’s behavior:

Fs=8000

This fs stands for sampling frequency. Because the filter is running on Basys3, we need to factor in the sampling rate, i.e. the rate at which the device itself is sampling the incoming continuous signal.

$$fs\ of\ the\ Basys3\ =\ 1,000,000\ Hz$$

This places an upper limit on the sampling frequency that the filter itself can have. In this case, we have chosen to downsample this to 8000 Hz, or in this case, every 125th sample. This concession was made mainly to improve computational performance.

$$fs\ =\ 8,000\ Hz$$

fpass=981 and fstop=1500

This range is chosen by trial and error. In the aim to minimize the stopband ripples. The idea is to maintain a steep attenuation within the passband, while keeping the ripples in the stopband to a minimum.

Further, the specific fpass and fstop specifically are selected because we first performed the experiment using the built in filter present in an oscilloscope. In doing so, we fund that the definite noise components began after the frequency defined in fpass, thus the number was chosen.

wpass=0.785 and wstop=0.079

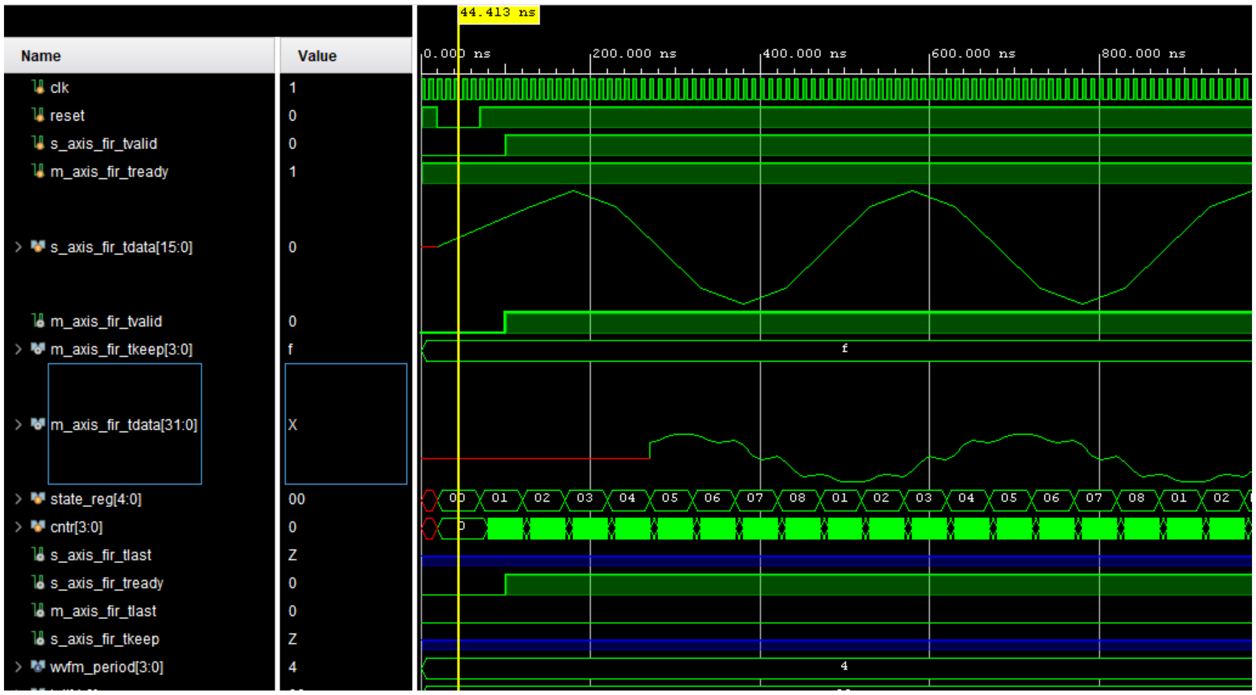
The formula for wpass and wstop depends on the type of filter design used. For FIR equiripple filters, the weights are typically specified in decibels (dB) and can be calculated using the following formulas:

$$wpass\ =\ 10^{(0.1*rpass)-1}$$
$$wstop\ =\ 10^{(0.1*rstop)-1}$$

where Rpass and Rstop are the maximum allowable ripples in the passband and stopband regions, respectively, in dB. These values are specified based on the desired filter performance.

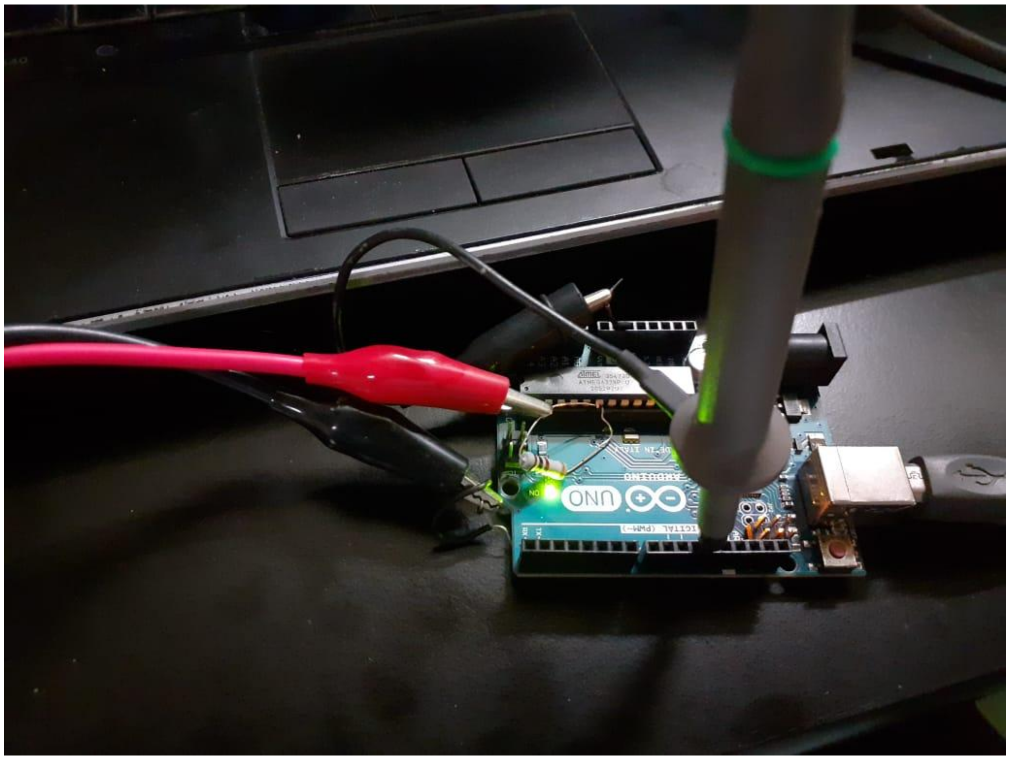
In the case of this filter design, since we need to use an FIR equiripple filter with a lowpass specification, and given the order of the filter, density factor, and sampling frequency, we have used the firpmord function in MATLAB to estimate the values of wpass and wstop based on our desired specifications. The firpmord function takes as input the desired filter specifications and returns the minimum order of the filter required to meet those specifications, as well as an estimate of the passband and stopband weights.

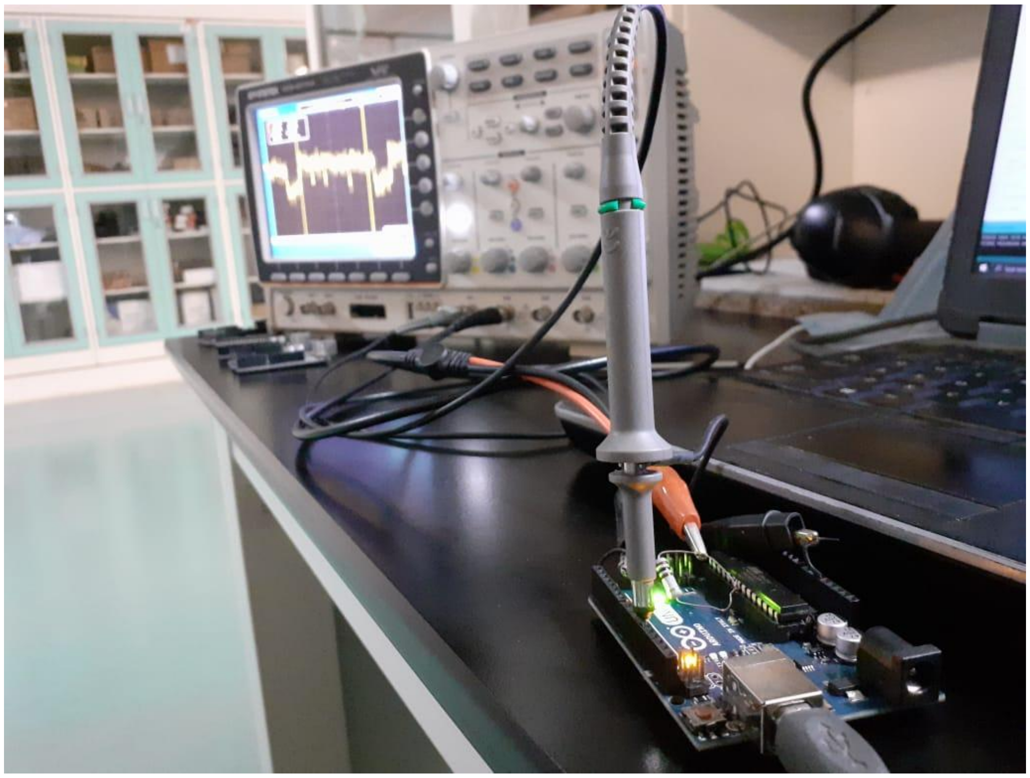
Testing Filter On Vivado:



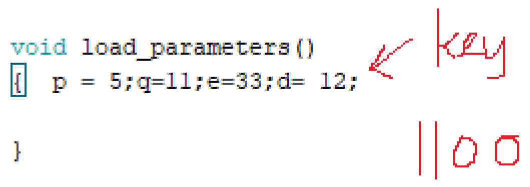
1. Open the Vivado Design Suite and create a new project. Specify the name and location of the project and select the target FPGA device.
2. In the Vivado Flow Navigator, click on "Create Block Design" to create a new block design for your filter.
3. In the block design, add a filter IP core. You can do this by clicking on the "IP Catalog" tab, searching for the filter IP core you want to use, and dragging it onto the block design canvas.
4. Configure the filter IP core by double-clicking on it and setting its parameters. For example, you might need to specify the filter coefficients, the filter type (e.g., lowpass, highpass, bandpass, or bandstop), the filter order, and the input and output data types.
5. Connect the input and output ports of the filter IP core to other blocks in the block design. For example, you might connect the filter output to a signal processing block that processes the filtered signal further.
6. Click on "Run Connection Automation" in the toolbar to automatically connect the unconnected ports in the block design.
7. Click on "Generate Output Products" to generate the HDL code for your block design.
8. Click on "Run Simulation" to simulate your filter. You can choose to simulate your filter using Vivado Simulator or an external simulator such as ModelSim.
9. After the simulation is complete, you can view the simulation results and waveform traces to verify the performance of your filter.

Making connections:

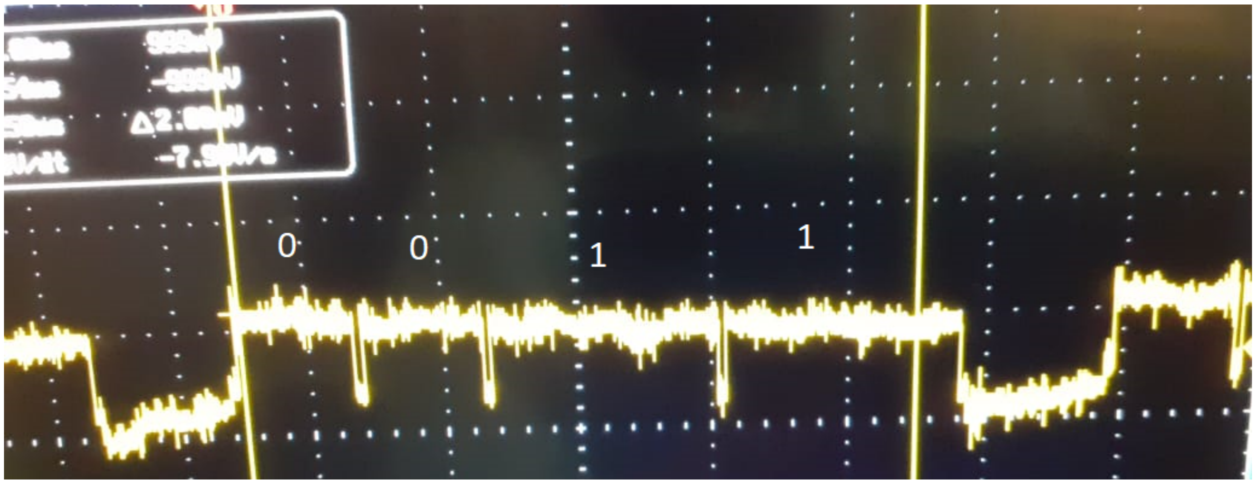


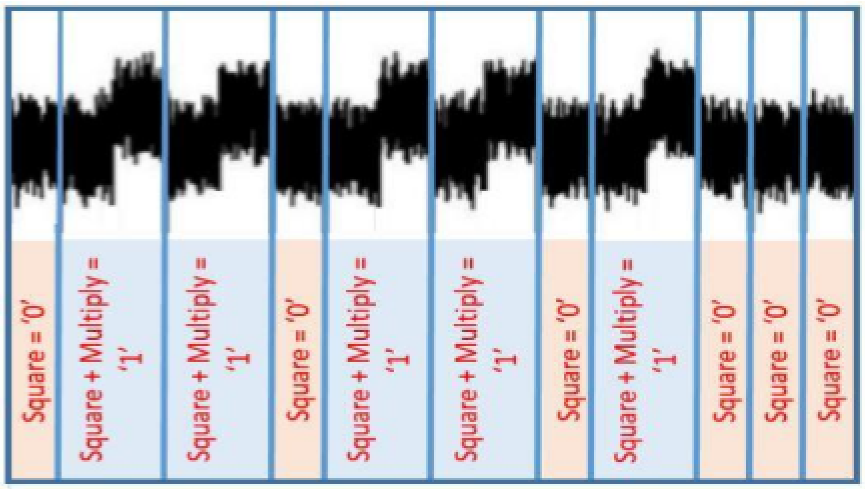


Results:

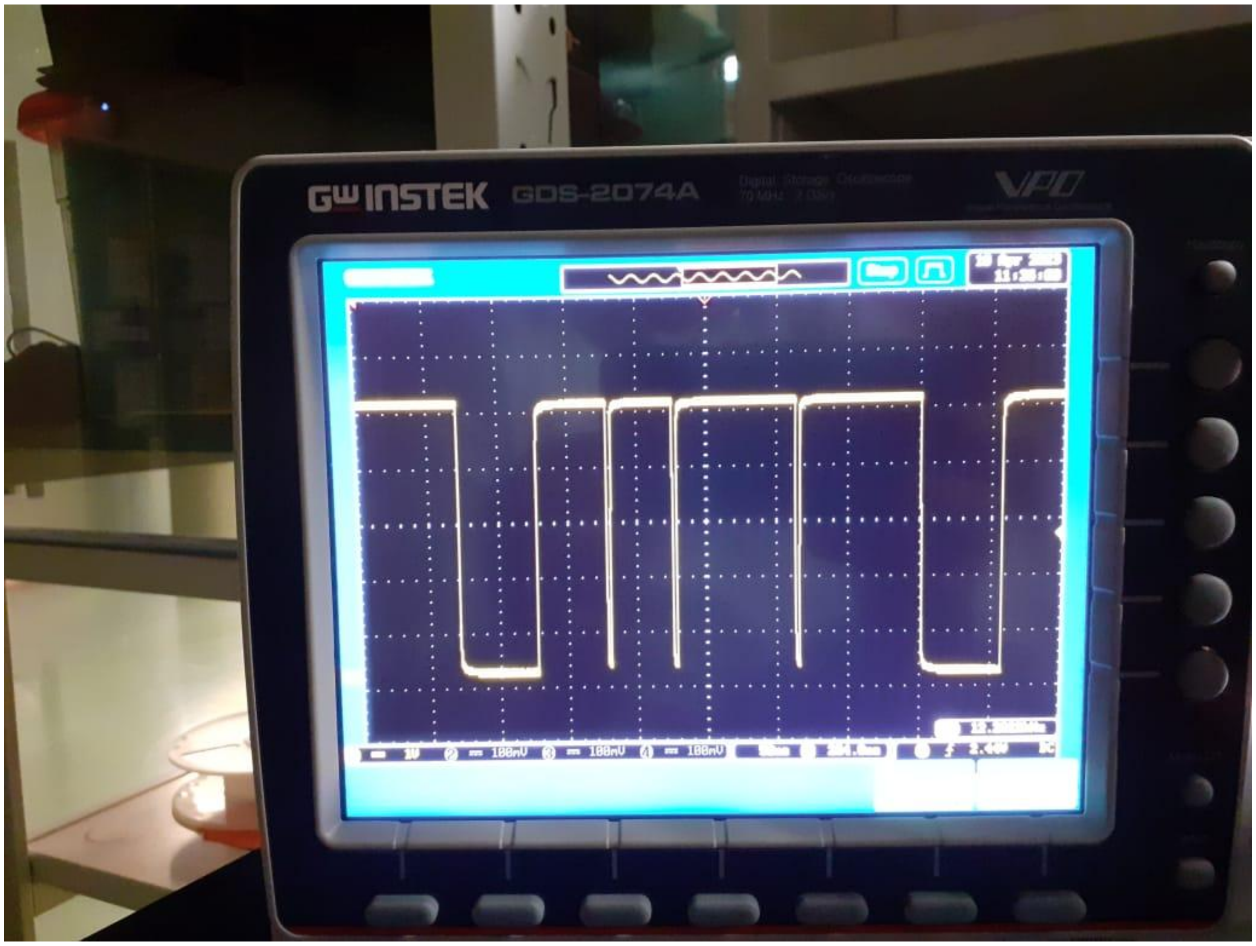


```
void load_parameters()
{  p = 5;q=11;e=33;d= 12;

}
```





The key is mentioned here agained, for reference.Avg power is taken hence integral of the above graph would give use average power which we can then observe that how much power is consumed for different operations. The binary number is flipped as the first one that is processed is the the


GW INSTEK
GDS-2074A
VPO
1V
100mV
100mV
100mV
2.40V
DC

Code for filter design

```
`timescale 1ns / 1ps

module FIR(
    input clk,
    input reset,
    input signed [15:0] s_axis_fir_tdata,
    input [3:0] s_axis_fir_tkeep,
    input s_axis_fir_tlast,
    input s_axis_fir_tvalid,
    input m_axis_fir_tready,
    output reg m_axis_fir_tvalid,
    output reg s_axis_fir_tready,
    output reg m_axis_fir_tlast,
    output reg [3:0] m_axis_fir_tkeep,
    output reg signed [31:0] m_axis_fir_tdata
    );


    always @ (posedge clk)
        begin
            m_axis_fir_tkeep <= 4'hf;
        end

    always @ (posedge clk)
        begin
            if (s_axis_fir_tlast == 1'b1)
                begin
                    m_axis_fir_tlast <= 1'b1;
                end
            else
                begin
                    m_axis_fir_tlast <= 1'b0;
                end
        end

    // 15-tap FIR
    reg enable_fir, enable_buff;
    reg [3:0] buff_cnt;
    reg signed [15:0] in_sample;
    reg signed [15:0] buff0, buff1, buff2, buff3, buff4, buff5, buff6, buff7, buff8, buff9, buff10, buff11,
buff12, buff13, buff14;
    wire signed [15:0] tap0, tap1, tap2, tap3, tap4, tap5, tap6, tap7, tap8, tap9, tap10, tap11, tap12, tap13,
tap14;
    reg signed [31:0] acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7, acc8, acc9, acc10, acc11, acc12,
acc13, acc14;
```

```
/* Taps for LPF running @ 1MSps with a cutoff freq of 400kHz*/
assign tap0 = 16'hFC9C;  // twos(-0.0265 * 32768) = 0xFC9C
assign tap1 = 16'h0000;  // 0
assign tap2 = 16'h05A5;  // 0.0441 * 32768 = 1445.0688 = 1445 = 0x05A5
assign tap3 = 16'h0000;  // 0
assign tap4 = 16'hF40C;  // twos(-0.0934 * 32768) = 0xF40C
assign tap5 = 16'h0000;  // 0
assign tap6 = 16'h282D;  // 0.3139 * 32768 = 10285.8752 = 10285 = 0x282D
assign tap7 = 16'h4000;  // 0.5000 * 32768 = 16384 = 0x4000
assign tap8 = 16'h282D;  // 0.3139 * 32768 = 10285.8752 = 10285 = 0x282D
assign tap9 = 16'h0000;  // 0
assign tap10 = 16'hF40C; // twos(-0.0934 * 32768) = 0xF40C
assign tap11 = 16'h0000; // 0
assign tap12 = 16'h05A5; // 0.0441 * 32768 = 1445.0688 = 1445 = 0x05A5
assign tap13 = 16'h0000; // 0
assign tap14 = 16'hFC9C; // twos(-0.0265 * 32768) = 0xFC9C

/* This loop sets the tvalid flag on the output of the FIR high once
 * the circular buffer has been filled with input samples for the
 * first time after a reset condition. */
always @ (posedge clk or negedge reset)
   begin
      if (reset == 1'b0) //if (reset == 1'b0 || tvalid_in == 1'b0)
         begin
            buff_cnt <= 4'd0;
            enable_fir <= 1'b0;
            in_sample <= 8'd0;
         end
      else if (m_axis_fir_tready == 1'b0 || s_axis_fir_tvalid == 1'b0)
         begin
            enable_fir <= 1'b0;
            buff_cnt <= 4'd15;
            in_sample <= in_sample;
         end
      else if (buff_cnt == 4'd15)
         begin
            buff_cnt <= 4'd0;
            enable_fir <= 1'b1;
            in_sample <= s_axis_fir_tdata;
         end
      else
         begin
            buff_cnt <= buff_cnt + 1;
            in_sample <= s_axis_fir_tdata;
         end
   end

always @ (posedge clk)
   begin
      if(reset == 1'b0 || m_axis_fir_tready == 1'b0 || s_axis_fir_tvalid == 1'b0)
```

```
            begin
                s_axis_fir_tready <= 1'b0;
                m_axis_fir_tvalid <= 1'b0;
                enable_buff <= 1'b0;
            end
        else
            begin
                s_axis_fir_tready <= 1'b1;
                m_axis_fir_tvalid <= 1'b1;
                enable_buff <= 1'b1;
            end
    end

/* Circular buffer bring in a serial input sample stream that
 * creates an array of 15 input samples for the 15 taps of the filter. */
always @ (posedge clk)
    begin
        if(enable_buff == 1'b1)
            begin
                buff0 <= in_sample;
                buff1 <= buff0;
                buff2 <= buff1;
                buff3 <= buff2;
                buff4 <= buff3;
                buff5 <= buff4;
                buff6 <= buff5;
                buff7 <= buff6;
                buff8 <= buff7;
                buff9 <= buff8;
                buff10 <= buff9;
                buff11 <= buff10;
                buff12 <= buff11;
                buff13 <= buff12;
                buff14 <= buff13;
            end
        else
            begin
                buff0 <= buff0;
                buff1 <= buff1;
                buff2 <= buff2;
                buff3 <= buff3;
                buff4 <= buff4;
                buff5 <= buff5;
                buff6 <= buff6;
                buff7 <= buff7;
                buff8 <= buff8;
                buff9 <= buff9;
                buff10 <= buff10;
                buff11 <= buff11;
                buff12 <= buff12;
```

```
          buff13 <= buff13;
          buff14 <= buff14;
        end
    end

  /* Multiply stage of FIR */
  always @ (posedge clk)
    begin
      if (enable_fir == 1'b1)
        begin
          acc0 <= tap0 * buff0;
          acc1 <= tap1 * buff1;
          acc2 <= tap2 * buff2;
          acc3 <= tap3 * buff3;
          acc4 <= tap4 * buff4;
          acc5 <= tap5 * buff5;
          acc6 <= tap6 * buff6;
          acc7 <= tap7 * buff7;
          acc8 <= tap8 * buff8;
          acc9 <= tap9 * buff9;
          acc10 <= tap10 * buff10;
          acc11 <= tap11 * buff11;
          acc12 <= tap12 * buff12;
          acc13 <= tap13 * buff13;
          acc14 <= tap14 * buff14;
        end
    end

   /* Accumulate stage of FIR */
  always @ (posedge clk)
    begin
      if (enable_fir == 1'b1)
        begin
          m_axis_fir_tdata <= acc0 + acc1 + acc2 + acc3 + acc4 + acc5 + acc6 + acc7 + acc8 + acc9
+ acc10 + acc11 + acc12 + acc13 + acc14;
        end
    end



endmodule
```