
	Academic Year:	2022/2023	Term:	2nd term-2022	
	Course Code:	ELC4007	Course Title:	موضوعات مختارة في الالكترونيات 2	

Cairo University
Faculty of Engineering
Electronics and Communications Engineering Department
Fourth Year

Electronics project **FFE equalizer REPORT**

CONTENTS

Why equalizers?.....	3
Finite impulse response(FIR).....	4
Block diagram	5
Block diagram details.....	6
1.Data Synchronizer.....	6
2.Register.....	6
3.Counter	7
4.MUX_4×1	7
5.Multiplier.....	8
6.Adder	8
7.MUX_2×1.....	8
Measurements	61
Histograms	63
Discussion.....	66
Market Analysis	68
Region X	71
Country Y	73
Male vs Female	75
Age Groups	78
Conclusions	81
Recommendations	83
Group Meeting	86
Regulatory Bodies	87
Agents	88
Appendix A. Transcriptions with Institutions	91
Appendix B. Tables with Results	98

1. Why equalizers?

In high-speed buses and due to channel's low pass behaviour, we have many challenges such as losses due to high rates and Inter-symbol Interference (ISI) which distort our signal and hence BER (Bit Error Rate) increases.

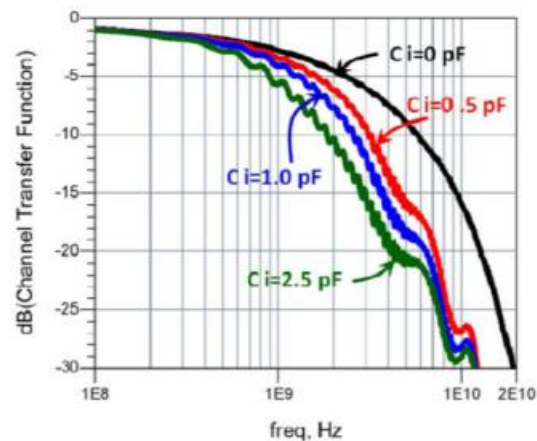


Fig 1 (Channel transfer function)

So, to decrease our BER and increasing the rate of transmitting of the data and increasing Eye diagram's opening, we must overcome channel's lossy transfer function by multiplying it with the inverse to have kind of flat response in our desired band of interest and that is exactly what Equalization is.

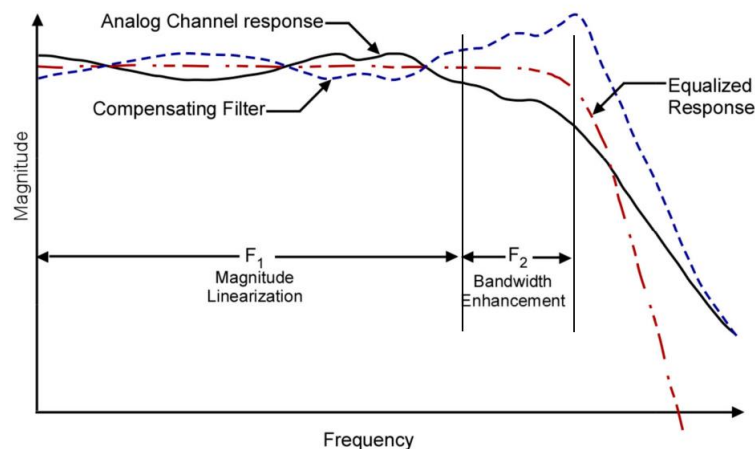


Fig 2 (Equalization technique)

2.Finite impulse response equalizer: -

Finite impulse response (FIR) equalizer in RX suppresses channel's loss by boosting high frequency contents of transmitted data and can be implemented in Digital or Analog domains.

FIR is simply a convolution between our desired data and some calculated coefficients to retain the data at high data rates.

$$y[n] = \sum_{k=0}^N h_k * x[n - k]$$

The more samples used in our FIR filter the better response and quality we get.

Advantages: -

- Simple to implement.
- Doesn't amplify noise.
- Easily cancels precursors.
- With sufficient dynamic range, can amplify high frequency content (rather than attenuate low frequencies).

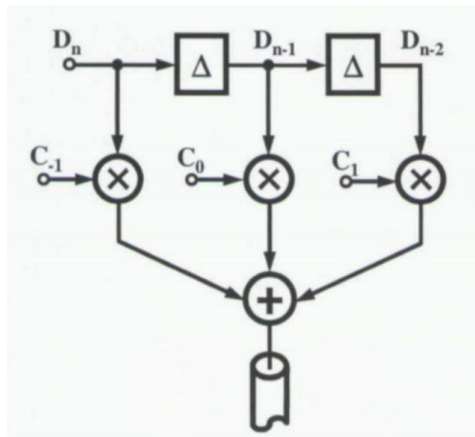


Fig 3 (FIR equalizer diagram)

3.Block diagram: -

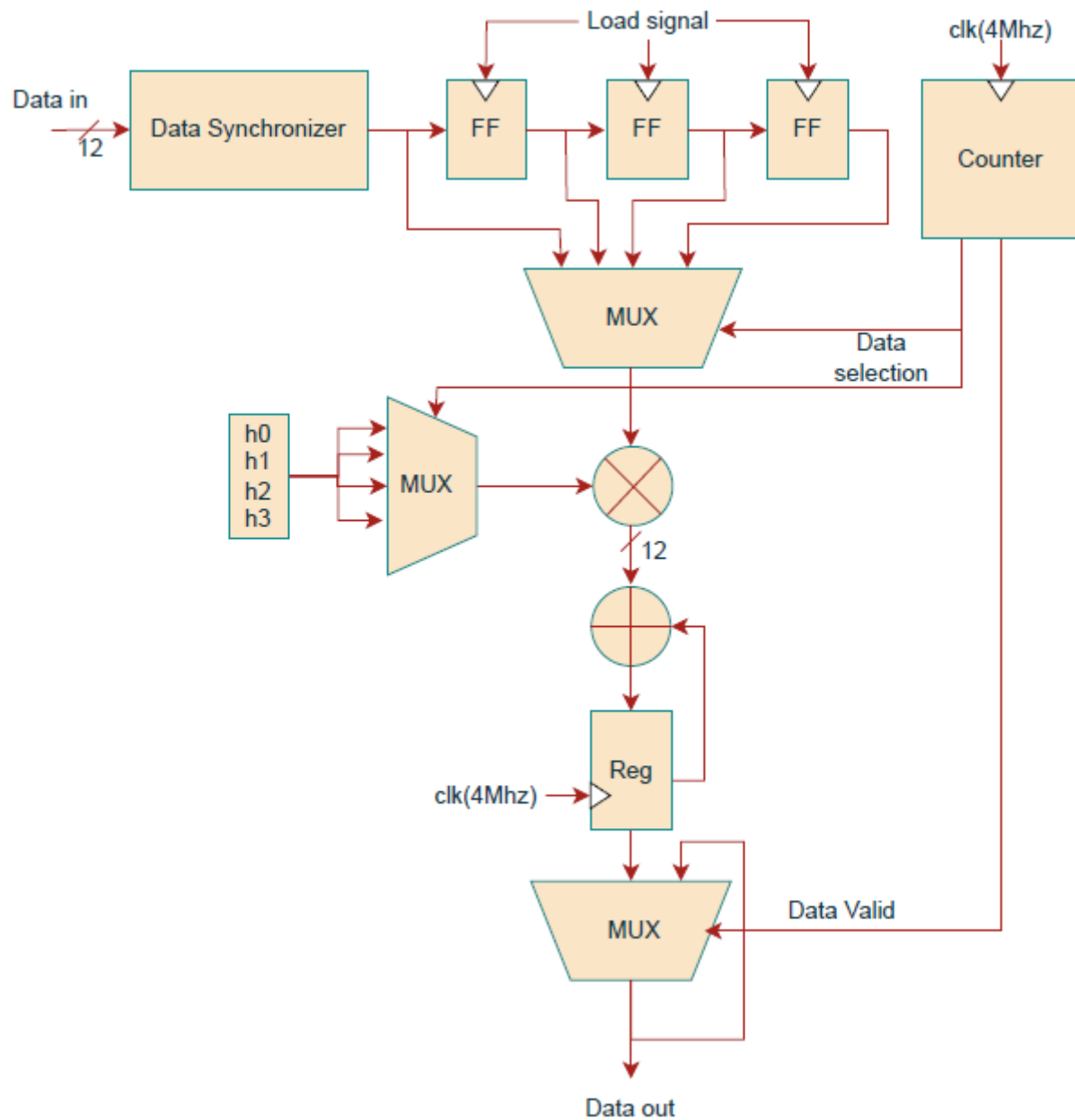


Fig 4 (Block diagram)

4.Block diagram Details: -

In this section, we are going to talk about each block in the block diagram in details.

1)Data synchronizer: -

signal	Data width	Direction
clk	1	In
Rst	1	In
unsync_bus	12	In
Bus_enable	1	In
Sync_bus	12	Out
enable_pulse_d	1	Out

This block is responsible for taking the unsynchronized data due to clock domain crossing and giving us synchronized data depending on the clock.

2) Register: -

signal	Data width	Direction
In	12	In
Rst	1	In
Clk	1	In
Out	12	Out

This block is responsible for saving the data to enable us from using the saved data later.

3) Counter: -

signal	Data width	direction
Rst	1	In
Clk	1	In
Counter_enable	1	In
Counter_done_seq	1	Out
Counter_done_comb	1	Out
Count	2	out

This block works as the control unit in our design according to the value of count we can determine the current state.

4) MUX_4x1: -

signal	Data width	direction
In1	12	In
In2	12	In
In3	12	In
In4	12	In
Sel	2	In
Out	12	Out

This block is responsible for data selection according to the value of sel signal.

5) Multiplier: -

signal	Data width	Direction
In1	12	In
In2	12	In
Out	12	out

This block is responsible for multiplying two operands and gives us the output of multiplication.

6) ADDER: -

signal	Data width	direction
in1	12	In
in2	12	In
out	12	out

This block is responsible for adding two operands and gives us the output of addition.

7) MUX_2x1: -

signal	Data width	direction
In1	12	In
In2	12	In
Sel	1	In
out	12	out

This block is responsible for data selection according to the value of sel signal.

Design specifications:

We have input data rate 4Mhz and we need throughput of 1Mhz so by using time sharing we can achieve the specs also the load signal frequency is 1Mhz.

Another problem is the two clocks are asynchronous so we deal with this using data synchronizer block to isolate metastability from the receiver.

Signals:

Name	Direction	Width(bits)
clk	Input	1
rst	Input	1
load_signal	Input	1
Data_in	Input	12
Data_out	Output	12

Notes: we used the load signal as the clock (1Mhz) signal of the delay flip flops.

Counter:

It acts as the controller. It is responsible for controlling multiplexers' selections.

Multiplier:

Explain the truncation of bits.....

5.Simulation: -

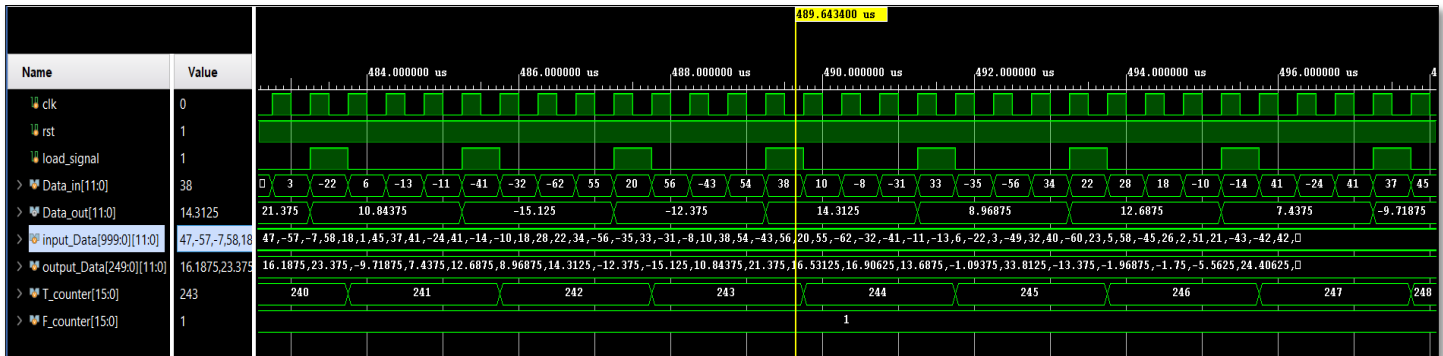


Figure 5 Design Simulation

6.Results: -

- The simulated output is equal to the model output generated by MATLAB.
- We ran the program using 1000 input data and recorded the number of failures that occurred.
- The T_counter represents the number of matches between the output of the design and the model output from MATLAB, and we observed that there were no failures and that the correctness between the design and model output was maintained.

7.Appendix: -

Design Code: [GITHUB REPO](#)

CODE

Top module

```
module TOP #(
    parameter data_width=12 , counter_max=3)
    ( input  wire          clk,
      input  wire          rst,
      input  wire          load_signal,
      input  wire [data_width-1:0]  Data_in,
      output wire [data_width-1:0]  Data_out
    );

    //parameters
    localparam counter_width = $clog2(counter_max);

    // internal wires
    wire [data_width-1:0]  Req0,Reg1,Reg2,Reg3;
    wire                   data_valid;
    wire                   data_valid_comb;
    wire [counter_width-1:0]  count;
    wire [data_width-1:0]  MUL_in1,MUL_in2,MUL_out;
    wire [data_width-1:0]  ADDER_in2,ADDER_out;
    wire [data_width-1:0]  OUT_MUX_IN;

    REGISTER delay_unit_0(
        .in(Data_in),
        .rst(rst),
        .clk(load_signal),
        .out(Req0)
    );

    REGISTER delay_unit_1(
        .in(Req0),
        .rst(rst),
        .clk(load_signal),
        .out(Reg1)
    );

    REGISTER delay_unit_2(
        .in(Reg1),
        .rst(rst),
        .clk(load_signal),
```

```

        .out(Reg2)
    );

    REGISTER delay_unit_3(
        .in(Reg2),
        .rst(rst),
        .clk(load_signal),
        .out(Reg3)
    );

    COUNTER Counter1(
        .clk(clk),
        .rst(rst),
        .Counter_enable(load_signal),
        .Counter_done_seq(data_valid),
        .Counter_done_comb(data_valid_comb),
        .count(count)
    );

    MUX_4x1 Mux_after_registers(
        .in1(Req0),
        .in2(Reg1),
        .in3(Reg2),
        .in4(Reg3),
        .sel(count),
        .out(MUL_in1)
    );

    MUX_4x1 Mux_constant_Multiplier(
        .in1(12'b0000_0001_0000),
        .in2(12'b1111_1111_1000),
        .in3(12'b0000_0000_0101),
        .in4(12'b1111_1111_1110),
        .sel(count),
        .out(MUL_in2)
    );

    MUL MUL_u0(
        .in1(MUL_in1),
        .in2(MUL_in2),
        .out(MUL_out)
    );

    ADDER ADDER_u0(
        .in1(MUL_out),
        .in2(ADDER_in2),
        .out(ADDER_out)
    );

```

```

REGISTER ADDER_reg(
    .in(ADDER_out),
    .rst(rst),
    .clk(clk),
    .out(OUT_MUX_IN)
);

REGISTER reg_delete(
    .in(ADDER_out),
    .rst(rst),
    .delete(data_valid_comb),
    .clk(clk),
    .out(ADDER_in2)
);

MUX_2x1 Mux_FFE_out(
    .in1(Data_out),
    .in2(OUT_MUX_IN),
    .sel(data_valid),
    .out(Data_out)
);

endmodule

```

Adder module

```

module ADDER #(parameter width=12)
( input wire signed [width-1:0] in1,
  input wire signed [width-1:0] in2,
  output wire signed [width-1:0] out
);

reg [width:0] tmp;

always @(*)
begin
    tmp=in1+in2;
end

assign out=tmp[width-1:0];

endmodule

```

Counter module

```
module COUNTER #(parameter counter_max = 3 ,
                  count_width= $clog2(counter_max))
( input wire      clk,
  input wire      rst,
  input wire      Counter_enable,
  output reg      Counter_done_seq,
  output wire     Counter_done_comb,
  output reg [count_width-1:0] count
);

  reg    flag;

always @(posedge clk or negedge rst ) begin
  if(!rst)
    begin
      count<=0;
      flag<=1;
    end
  else if(Counter_enable)
    begin
      count<=1;
      flag<=0;
    end
  else
    begin
      if(count<counter_max && flag==0)
        begin
          count<=count+1;
          flag<=0;
        end
      else
        begin
          count<=0;
          flag<=1;
        end
    end
  end
end

////////////////////////////////////
////////// put in always block to delay 1 clk cycle //////////
////////////////////////////////////
always @(posedge clk or negedge rst)
begin
  if (!rst)
```

```

begin
    Counter_done_seq<=0;
end
else
begin
    Counter_done_seq<=(count==counter_max);
end

end

assign Counter_done_comb=(count==counter_max);

endmodule

```

Multiplier module

```

module MUL #(parameter data_width=12)
( input  wire signed [data_width-1:0] in1,
  input  wire signed [data_width-1:0] in2,
  output wire signed [data_width-1:0] out
);

reg signed [(2*data_width)-1:0] tmp;

always @(*)
begin
    tmp=in1*in2;
end

assign out={tmp[2*data_width-1] , tmp[data_width-2:0]};

endmodule

```

MUX 2x1 module

```

module MUX_2x1 #(parameter data_width=12)
(
    input wire [data_width-1:0] in1,
    input wire [data_width-1:0] in2,
    input wire sel,
    output reg [data_width-1:0] out
);

always @(*)

```

```

begin
  case (sel)
    1'b0: out=in1;
    1'b1: out=in2;
    default: out=in1;

  endcase
end
endmodule

```

MUX 4x1 module

```

module MUX_4x1 #(
  parameter data_width=12
) (
  input wire [data_width-1:0] in1,
  input wire [data_width-1:0] in2,
  input wire [data_width-1:0] in3,
  input wire [data_width-1:0] in4,
  input wire [1:0] sel,
  output reg [data_width-1:0] out
);

always @(*)
begin
  case (sel)
    2'b00: out=in1;

    2'b01: out=in2;

    2'b10: out=in3;

    2'b11: out=in4;

    default: out=in1;
  endcase
end
endmodule

```

Register module

```

module REGISTER #(
  parameter width = 12)
( input wire      clk,

```



```

input wire      rst,
input wire [width-1:0] in,
input wire      delete,
output reg [width-1:0] out
);

always @(posedge clk or negedge rst)
begin
    if (!rst)
    begin
        out<=0;
    end

    else if(delete)
    begin
        out<=0;
    end

    else begin
        out<=in;
    end
end
endmodule

```

Testbench module

```

`timescale 1ns/1ps
module testbench ();

//parameters
parameter DATA_WIDTH =12,
           CLK_PER = 250;

// DUT signals
reg        clk;
reg        rst;
reg        load_signal;
reg  signed [DATA_WIDTH-1:0] Data_in;

```

```

wire signed [DATA_WIDTH-1:0] Data_out;

// Testbench signals
reg [11:0] input_Data [999:0];
reg [11:0] output_Data [249:0];
reg [15:0] T_counter;
reg [15:0] F_counter;
integer i;

initial
begin
    $dumpfile("FFE .vcd");
    $dumpvars;

    initialize();
    reset();

    $readmemb("D:/DIGITAL/Project/DATA/All_Inputs.txt", input_Data);
    $readmemb("D:/DIGITAL/Project/DATA/Expected_outputs.txt", output_Data);

    for(i=0;i<1000;i=i+1)
    begin
        @(posedge clk)
        Data_in= input_Data[i];
        if(load_signal)
        begin
            if(output_Data[(i/4)-1]== Data_out)
            begin
                T_counter=T_counter+1;
            end
            else
            begin
                F_counter=F_counter+1;
                $display (" EXPECTED OUTPUT = %b , SIM_OUTPUT=%b",output_Data[i/4] ,Data_out );
            end
        end
    end
end

// to make load high one cycle every 4 clock cycles
initial
begin
    reset();

    repeat(250)

```

```

begin
    @(posedge clk)
    load_signal<=1;
repeat(3)
    begin
        @(posedge clk)
        load_signal<=0;
    end
end
#CLK_PER;
#CLK_PER;
$stop;
end

// task reset
task reset();
begin
    rst=1;
    @(negedge clk)
    rst=0;
    @(negedge clk)
    rst=1;
end
endtask

//task initialize
task initialize();
begin
    clk=0;
    load_signal=0;
    Data_in=0;
    T_counter='b0;
    F_counter='b0;
end
endtask

// Clock Generation
always #CLK_PER clk=~clk;

// module instantiation
TOP #(.data_width(DATA_WIDTH))
dut(.Data_in  (Data_in),
    .load_signal (load_signal),
    .clk      (clk),
    .rst      (rst),
    .Data_out  (Data_out));

```

Endmodule

MATLAB Model

```
close all;
clear all;
clc;

%% vectors
all_data=[];
data_valid=[];
data_valid_bin=[];

output_vector=[];
output_vector_fixed=[];
output_vector_bin=[];

%% generate 1000 random integers between -128 and 128
random_integers = randi([-64,64], 1, 1000);

%% loop to store all input data
for i=1:length(random_integers)
    data=random_integers(i);
    all_data_bin=sfi(data,12,0).bin;
    all_data=[all_data ; all_data_bin ];
end

%% loop to store only valid data ( element every 4 elements)
for i=1:4:length(random_integers)
    data=random_integers(i);
    data_valid=[data_valid data];

    data_bin=sfi(data,12,0).bin;
    data_valid_bin=[data_valid_bin ; data_bin];
end

%% generation of FFE Mutliplier
data1=data_valid(1)*.5;
data2=data_valid(1)*-.25+.5*data_valid(2);
data3=data_valid(1)*.15625-.25*data_valid(2)+data_valid(3)*.5;
output_vector=[data1 data2 data3];

for i=4:length(data_valid)
```

```

    calc_data=data_valid(i)*.5 -.25*data_valid(i-1) +.15625*data_valid(i-2) -.0625*data_valid(i-3);
    output_vector=[output_vector calc_data];
end

%% loop to convert outputs to fixed point representation (1 sign + 6 integers + 5 fractions )
for i=1:250
    data=output_vector(i);

    data_fixed=sfi(data,12,5);
    output_vector_fixed=[output_vector_fixed data_fixed];

    data_bin=sfi(data,12,5).bin;
    output_vector_bin=[output_vector_bin ; data_bin];
end

```