Proposal of implementation of FFE EQUALIZER

Submitted to: Dr. Kareem Osama and Eng/Mahmoud Yehia.

Submitted by:
Omar Muhammad Mustafa AbdelLatif (9202989) – Omar Amr Mahmoud Hafez (9202967)
Omar Muhammad Tolba (9202985) – Salma Wael Gamal El dien (9202654)
Reem Mohamed Ashour (9202569) – Nermin Hassan Ali AbdelZaher (9203620)

***Abstract* – This paper proposes the implementation of a Feed Forward Equalizer (FFE) using only one adder and one multiplier. The specifications include an input data frequency of 1 MHz and an FFE clock frequency of 4 MHz. The output data is assumed to be a 12-bit signed value. The output is available after every 4 cycles of the FFE clock. This paper details the proposed design, provides pseudo code to represent the design, includes Verilog code to mimic the design behavior, and presents the netlist of the Verilog code.**

I. Proposed architecture of the FFE Equalizer

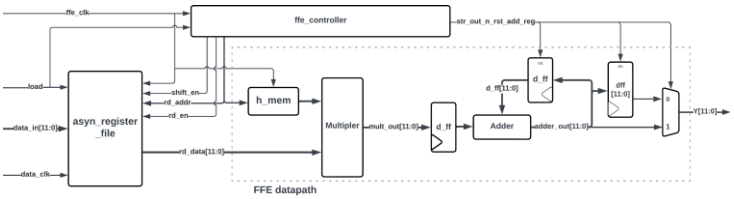


Figure 1 *Proposed architecture of FFE Equalizer with critical path breaking illustration.*

1. *asyn_register_file:* This module is responsible for handling the reception of the data since it follows a slower clock than the FFE clock and the reading happens to be faster. So, before the reception of the data the older received data will be read from the oldest to the newest to compute the terms that depend on older instances of data in. The received data will always be stored in entry zero of the memory and the older instances will be shifted in the memory to the next entries.

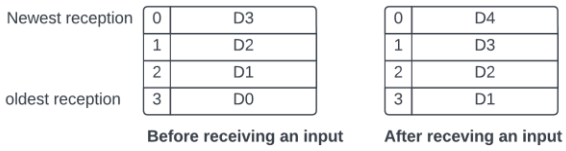
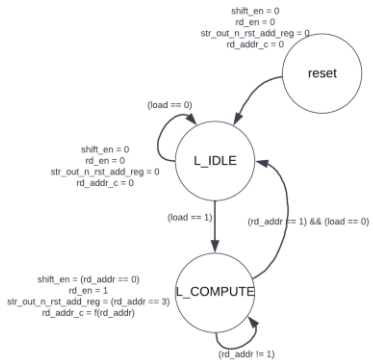


Figure 2 *data in memory behavior illustration.*

2. *ffe_datapath*: This module is responsible for computing the FFE of the received data while accumulating on the terms using only one adder and one multiplier. The adder output is accumulated in the register to be used for adding multiple terms and it's reset when the *str_out_n_rst_add_reg* is set. And finally, on computing the FFE, the output is bypassed to output through the multiplexer and after that the output is kept unchanged until the next data computations are ready.

3. *ffe_controller:* This module is responsible for tracing the calculations on different instances of the received data by assigning the proper read address to the *asyn_register_file* and *ffe_datapath* and it sets the control signal *str_out_n_rst_add_reg* whenever the read address equals to the value of three as this is the value that corresponds to the proper timing moment that is needed for the synchronization of path to operate on all the received data and with the registers placed in the path to enhance the speed performance of the system.



current state	rd_addr	rd_addr_c
IDLE or RESET	x	0
L_COMPUTE	0	3
L_COMPUTE	1	0
L_COMPUTE	2	1
L_COMPUTE	3	2

Figure 3 *Controller FSM and rd_addr_c output function*

N.B.
1 - This design has only one adder in *ffe_datapath* module
2 - Giving that the data clock and FFE clock are multiples of each other and if we assumed that they are derived from the same source and the difference is introduced by clock divider circuit, so in such case no synchronizers are needed between the reading and writing process from the memory as they are from the same source and multiples of each other. However, on the other hand if they are derived from different sources, so to prevent any possible violation from happing, a synchronizer is placed on the load signal branch going to the controller so to control the reading process properly without making any violations.

II. Pseudo codes

i) Pseudo code of ***ffe_datapath***

Module ffe_datapath with parameters: IN_OUT_BUS_WIDTH, DEPTH, ADDR_SIZE
Initialize memory array h_mem with predefined values: [1024, -512, 320, -128]
Define registers: multiplier_output_register, adder_output_register, final_output_register
Define wires: multiplier_output_comb, adder_output_comb, multiplier_result

On each clock cycle or reset:
If reset is active:
Reset the multiplier output, adder output, and final output registers to 0
Else:
If store and reset signal is active:
Reset the adder output register
Update the final output register with the adder output
Else:
Update the adder output register with the adder output

Always update the multiplier output register with the multiplier output
Compute the multiplier result by multiplying the coefficient from memory with the read data
Extract the relevant portion of the multiplier result
Compute the adder output by adding the multiplier output and the adder output

Assign the final output:
If the store and reset signal is active:
Set the output y to the adder output
Else:
Set the output y to the final output register
End Module

ii) Pseudo code of ***ffe_controller***

Module ffe_controller with parameters: DEPTH, ADDR_SIZE
Initialize states: RESET, IDLE, COMPUTE
Initialize counters: ZERO, ONE, TWO, THREE

Define registers: current_state, next_state, rd_addr, rd_addr_c

On each clock cycle or reset:
If reset is active:
Set the current state to RESET
Else:
Update the current state to the next state

On each clock cycle or reset:
If reset is active:
Reset the read address to 0
Else:
Update the read address to the temporary address

State and output logic:
If the current state is RESET:
Set the temporary read address to ZERO
Disable shift, read, and output signals
Move to the IDLE state

If the current state is IDLE:
Set the temporary read address to ZERO
Disable shift, read, and output signals
If the load signal is active:
Move to the COMPUTE state
Else:
Remain in the IDLE state

If the current state is COMPUTE:
Enable read signal, disable shift and output signals
By default, remain in the COMPUTE state

If the read address is ZERO:
Set the temporary read address to THREE
Enable the shift signal

If the read address is ONE:
Set the temporary read address to ZERO
If the load signal is inactive:
Move to the IDLE state

If the read address is TWO:
Set the temporary read address to ONE
If the read address is THREE:
Set the temporary read address to TWO
Enable the output signal
If the read address does not match any case:
Move to the RESET state
If the state does not match any case:
Move to the RESET state
End Module

III. Netlist of the proposed design with critical path modification.

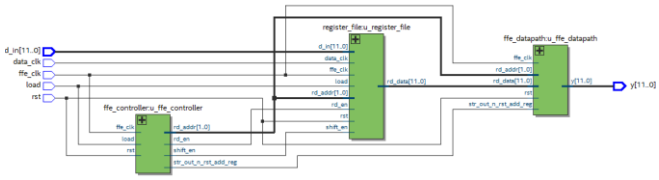


Figure 4 *top module of the FFE Equalizer*

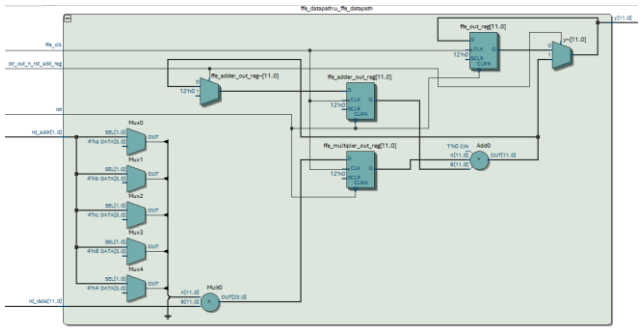


Figure 5 *FFE datapath architecture*

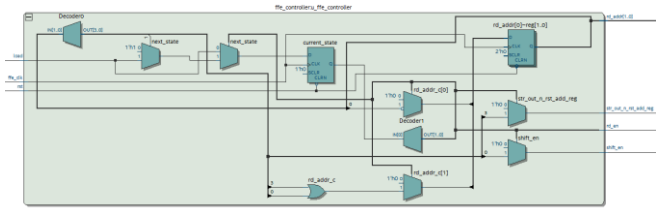


Figure 6 *FFE controller architecture*

IV. Waveforms captured from the proposed design

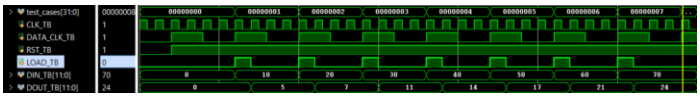


Figure 7 *simulation waveform*

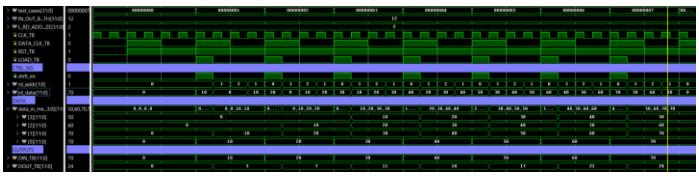


Figure 8 *DUT SIGNALS (ports & internal signals)*

N.B. The difference between the obtained value s and the actual hand computed values is due to the approximations done to multiple by fraction factors, values of the h.

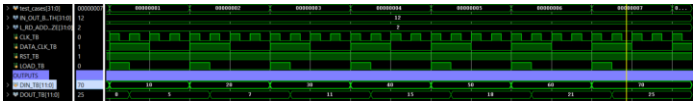


Figure 9 *Improved accuracy (increasing no of bits)*

V. Conclusion.

In this paper, we propose two implementations of the FFE Equalizer for different output modes: one aimed at improving the critical path, and the other without such improvements, impacting the timing required for a new input to influence the output. The number of bits in the internal buses significantly affects the system's accuracy, which has been thoroughly discussed. Both designs incorporate approximations due to the multiplication of input values by fractional coefficients.

VI. Verilog codes for the proposed design with critical path modifications and accuracy testing, and the Verilog code of the testbench.

The code is available on GitHub at the following link: *https://github.com/Omarmuhammadmu/FFE_digital_implementation*

To activate the critical path design, uncomment the line 32 in *ffe_datapath.v* and *ffe_controller.v* files

```
// `define CRITICAL_PATH_BREAKING
```

To activate the accuracy increasing design, uncomment the line 33 in *ffe_datapath.v*

```
// `define INCREASE_ACCURACY
```

VII. References

[1] Palermo, S. (n.d.). *Lecture 8: Adaptive Equalization*. Retrieved from https://people.engr.tamu.edu/spalermo/ecen689/lecture8_ee720_rx_adaptive_eq.pdf