64-tap 16-bit FIR filter

Aarya Agarwal
Columbia University
CSEE W4823
New York, New York
asa2276@columbia.edu

Stephen Ogunmwonyi
Columbia University
CSEE W4823
New York, New York
iso2107@columbia.edu

***Abstract*—This document discusses our design and implementation of a 64-tap 16-bit FIR filter for "Advanced Logic Design" (W4823) at Columbia University**

I. Introduction (*Heading 1*)

A 16-bit 64-tap FIR (Finite Impulse Response) filter is a type of digital filter used in signal processing that operates with 16-bit precision and processes input data using 64 filter coefficients, or "taps," to produce its output. It works by applying a weighted sum of the current and previous input samples, allowing it to perform tasks such as noise reduction, signal smoothing, and frequency shaping with high accuracy. This computation is also referred to as a discrete convolution. Common applications include audio signal processing, communication systems (like equalization and channel filtering), and biomedical signal analysis (e.g., ECG filtering).

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$$

x[n] is the input signal, *y[n]* is the output signal, *N* is the filter order (number of taps), b_i is the filter coefficient

Fig. 1. FIR Filter formula

II. Desgin

A. Controller

This controller ASM diagram describes the operation of a 64-tap FIR filter. The process starts by loading the coefficients and input data, then initializes counters and output values. For each iteration, the filter computes the output as a sum of the product of coefficients and input values, incrementing counters until all 64 taps are processed, after which it shifts the values and repeats the process.

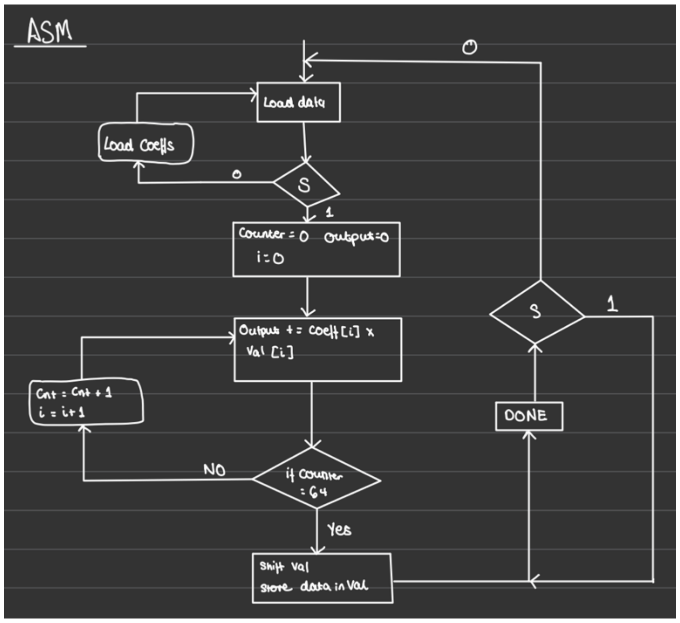


Fig. 2. FIR Filter controller ASM diagram

B. Datapath

This datapath block diagram shows the interaction between a FIFO, IMEM, SRAM, and an ALU for processing data. The FIFO stores incoming data, providing inputs to the IMEM Shift Register, which handles read and load operations before passing data to the ALU. The SRAM Coefficient Memory supplies coefficients to the ALU for computations, with signals like chip enable (CEN) and write enable (WEN) managing access. The ALU processes inputs, performs operations based on the opcode, and sends the results back through a multiplexer (MUX) and a register, completing the computational cycle.

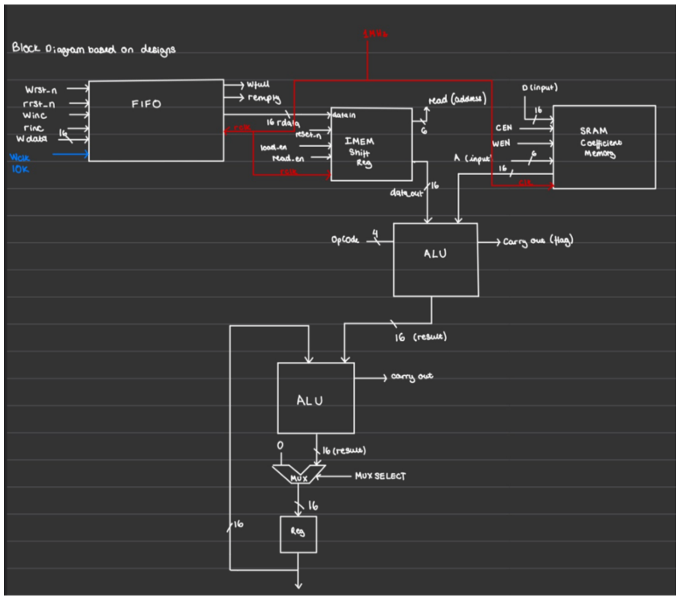


Fig. 3. FIR Filter datapath

III. Implementation

A number of key modules were designed and synthesized in Verilog. Their implementations and results are described as follows.

A. ALU

The provided ALU (Arithmetic Logic Unit) is a 16-bit design capable of performing a variety of arithmetic and logical operations based on a 4-bit opcode. It supports operations such as addition, subtraction, multiplication, division, logical shifts, rotations, and bitwise logical operations (AND, OR, XOR, NOR, NAND, XNOR). A carry-out signal is generated during addition, indicating if there is an overflow beyond 16 bits. This ALU is versatile and commonly used in processors, microcontrollers, and digital signal processing systems for computation and data manipulation tasks.

B. CMEM

The CMEM module is a 16-bit, 64-word single-port SRAM (Static Random Access Memory) designed for high-speed and high-density applications. It is implemented using a register file generator tailored for specific custom configurations, focusing on performance and power efficiency. The CMEMwrap file serves as a wrapper for the CMEM, providing a clean interface for integrating the memory into larger systems. It exposes input and output signals, including clock, address, chip enable, write enable, and data lines, which are directly mapped to the CMEM instance. This combination of CMEM and its wrapper is ideal for embedded systems, caches, or buffer storage applications, where efficient memory access and compact design are critical.

The CMEM memory module has a total cell area of approximately 12,740 µm², with timing analysis performed under typical operating conditions (1.2V, 25°C). Power analysis used a global operating voltage of 1.2V, with dynamic power units measured at 1mW.

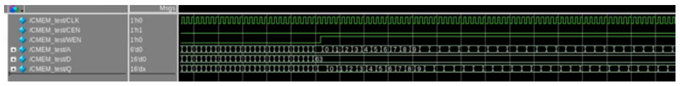


Fig. 4. CMEM post-synthesis waveforms

C. IMEM

The IMEM module functions as a 16-bit, 64-word memory array primarily used for storing and retrieving program instructions. It includes reset, load, and read functionalities, where the reset initializes all memory locations to zero, and the load feature allows sequential data loading. The memory uses a simple register-based design, with read_add serving as a pointer that cyclically increments to enable continuous data reads. The read operation is controlled through a read enable signal, ensuring precise and synchronized data output during each clock cycle. IMEM is typically utilized as Instruction Memory in processors, microcontrollers, and digital signal processing systems, playing a vital role in instruction fetch operations for sequential or pipelined execution.

The IMEM memory module has a total cell area of approximately 47,124 µm², with timing analysis conducted under typical operating conditions (1.2V, 25°C). Power analysis was performed with a global operating voltage of 1.2V, using dynamic power units of 1mW.

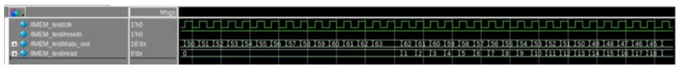


Fig. 5. IMEM post-synthesis waveforms

D. FIFO

The asynchronous FIFO design consists of multiple interconnected modules to facilitate data transfer between two clock domains. The FIFO memory module acts as the central storage, implementing a memory array where data is written and read asynchronously. The write pointer (wptr_full) and read pointer (rptr_empty) modules generate write and read pointers, respectively, while also managing the full and empty conditions to prevent overflow or underflow. To safely transfer pointers between clock domains, the two_ff_sync module implements a two-flip-flop synchronizer, ensuring metastability-free communication.

The primary FIFO module integrates all submodules, coordinating data flow between write and read operations based on independent clock inputs. The write pointer logic increments with each write operation and signals "full" when the FIFO reaches capacity, while the read pointer logic increments during reads and signals "empty" when no valid data remains. The design effectively decouples the producer and consumer clocks, enabling robust asynchronous data transfer. This asynchronous FIFO implementation is widely used in systems with clock domain crossing, such as data buffering between processors, interfaces, and communication subsystems.

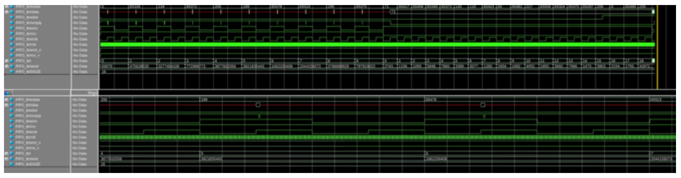


Fig. 6. FIFO post-synthesis waveforms

IV. Runtime Analysis

A. Output

Our FIR filter takes in input via the hardcodes found in our testbench "test_tb_modified." In our testing, we also created the following monitor in our controller/datapath module "test.v," which obviously has to be commented out in order for synthesis, which is why in synthesis there is greater reliance on the waveforms.

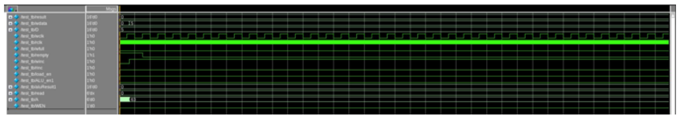


Fig. 7. FIR post-synthesis signals initializing

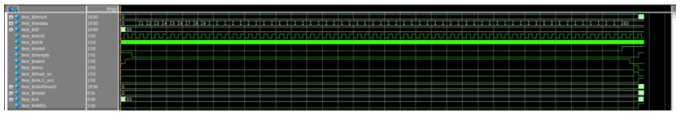
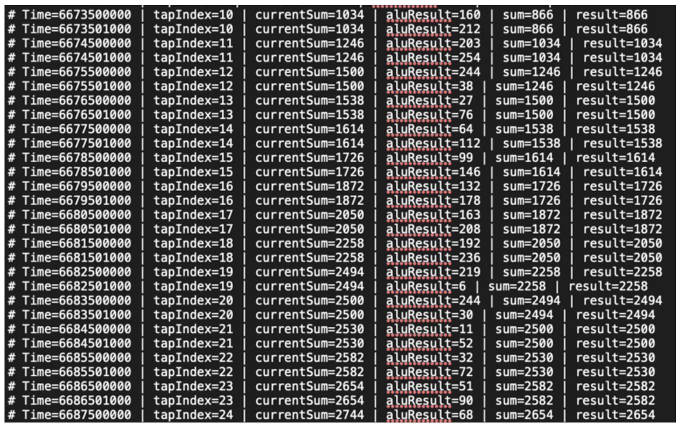


Fig. 8. FIR post-synthesis output signals and calculations



```
# Time=6673500000 | tapIndex=10 | currentSum=1034 | aluResult=160 | sum=866 | result=866
# Time=6673501000 | tapIndex=10 | currentSum=1034 | aluResult=212 | sum=866 | result=866
# Time=6674500000 | tapIndex=11 | currentSum=1246 | aluResult=203 | sum=1034 | result=1034
# Time=6674501000 | tapIndex=11 | currentSum=1246 | aluResult=254 | sum=1034 | result=1034
# Time=6675500000 | tapIndex=12 | currentSum=1500 | aluResult=244 | sum=1246 | result=1246
# Time=6675501000 | tapIndex=12 | currentSum=1500 | aluResult=38 | sum=1246 | result=1246
# Time=6676500000 | tapIndex=13 | currentSum=1538 | aluResult=27 | sum=1500 | result=1500
# Time=6676501000 | tapIndex=13 | currentSum=1538 | aluResult=76 | sum=1500 | result=1500
# Time=6677500000 | tapIndex=14 | currentSum=1614 | aluResult=64 | sum=1538 | result=1538
# Time=6677501000 | tapIndex=14 | currentSum=1614 | aluResult=112 | sum=1538 | result=1538
# Time=6678500000 | tapIndex=15 | currentSum=1726 | aluResult=99 | sum=1614 | result=1614
# Time=6678501000 | tapIndex=15 | currentSum=1726 | aluResult=146 | sum=1614 | result=1614
# Time=6679500000 | tapIndex=16 | currentSum=1872 | aluResult=132 | sum=1726 | result=1726
# Time=6679501000 | tapIndex=16 | currentSum=1872 | aluResult=178 | sum=1726 | result=1726
# Time=6680500000 | tapIndex=17 | currentSum=2050 | aluResult=163 | sum=1872 | result=1872
# Time=6680501000 | tapIndex=17 | currentSum=2050 | aluResult=208 | sum=1872 | result=1872
# Time=6681500000 | tapIndex=18 | currentSum=2258 | aluResult=192 | sum=2050 | result=2050
# Time=6681501000 | tapIndex=18 | currentSum=2258 | aluResult=236 | sum=2050 | result=2050
# Time=6682500000 | tapIndex=19 | currentSum=2494 | aluResult=219 | sum=2258 | result=2258
# Time=6682501000 | tapIndex=19 | currentSum=2494 | aluResult=6 | sum=2258 | result=2258
# Time=6683500000 | tapIndex=20 | currentSum=2500 | aluResult=244 | sum=2494 | result=2494
# Time=6683501000 | tapIndex=20 | currentSum=2500 | aluResult=30 | sum=2494 | result=2494
# Time=6684500000 | tapIndex=21 | currentSum=2530 | aluResult=11 | sum=2500 | result=2500
# Time=6684501000 | tapIndex=21 | currentSum=2530 | aluResult=52 | sum=2500 | result=2500
# Time=6685500000 | tapIndex=22 | currentSum=2582 | aluResult=32 | sum=2530 | result=2530
# Time=6685501000 | tapIndex=22 | currentSum=2582 | aluResult=72 | sum=2530 | result=2530
# Time=6686500000 | tapIndex=23 | currentSum=2654 | aluResult=51 | sum=2582 | result=2582
# Time=6686501000 | tapIndex=23 | currentSum=2654 | aluResult=90 | sum=2582 | result=2582
# Time=6687500000 | tapIndex=24 | currentSum=2744 | aluResult=68 | sum=2654 | result=2654
```

Fig. 9. FIR pre-synthesis monitor

B. Performance

Our design met slack and the following results are based on the pt results obtained.

Throughput: 22 [kS/s]
Maximum clock frequency: 1 MHz (Could potentially get faster with some tweaking of pipelining)
Total Power: 16uW
Max power: 0.2889 W
Area: 102245.77 um2
Accuracy: Worst case: 0% | Average: 98.4%