Task #1: Completed in Class

Task #2: Link below

https://github.com/thedubbers9/S25_EX4_tinytapeout/actions

Task #3:

Proposal Idea #1: [MAIN PROPOSAL]

Question 1:

RISC-E (As in reduced instruction set computer eric)

This processor will be von Neumann architecture (unified instruction and data memory).

This processor will be even simpler than the intel 4004. It is an EXTREEMLY simple processor. I will load instructions into the instruction memory for basic arithmetic. The processor will then run the instructions. The result will be visible in the data memory (off chip).

Question #2: technical description

This processor has four 8 bit registers. The off chip memory will have 2^10 locations. Each location will be 8 bits.

This will be a very basic ISA, just enough to be Turing complete. The full ISA specification is in the table below. I want to make my own ISA because the 4004 has been done many times before. I think it will be more fun to design my own from the ground up, even if it isn't quite as optimized as the 4004.

Format	**Assembly syntax**	**Semantics**
0000 dd ss	LOAD Rd, Rs	Rd ← MEM[Rs]
0001 dd ss	STORE Rd, Rs	MEM[Rs] ← Rd
1000 dd ss	ADD Rd, Rs	Rd ← Rs + Rd
1010 dd ss	AND Rd, Rs	Rd ← Rs & Rd
1011 dd ss	OR Rd, Rs	Rd ← Rs | Rd
1100 dd ss	XOR Rd, Rs	Rd ← Rs ^ Rd
1101 dd ss	SL Rd, Rs	Rd ← Rd << Rs
1110 dd ss	SRL Rd, Rs	Rd ← Rd >> Rs [Logical]
1111 dd ss	SRA Rd, Rs	Rd ← Rd >> Rs [Arithmetic]
0101 dd ss	NOT Rd, Rs	Rd ← ~Rs
0100 dd ss	LI Rd, IMM	Rd ← IMM
0010 dd xx	JUMP Rd	PC ← PC + 1 + Rd
0011 iiii	BRANCHz IMM	PC ← PC + 1 + IMM

0110 xxxx	HALT	Ends execution

Memory for instruction and data will be unified.

The processor will be multi-cycle. The goal of this is to use the same memory bus for both instruction fetch and data access. We do not have enough pins to do both within a single cycle.

In the first cycle, the processor will fetch the instruction from memory

In the second cycle, the processor will compute the ALU operation or read/write memory.

In the third cycle, the processor will complete the operation and fully store the result into memory or the register file.

Question #3:

I/Os:

Outputs (from chip): 10 bit memory address/data, 1 bit read/write bit, 1 bit write commit.

Inputs (to chip): 8 bit memory result

The outputs from the chip to the data memory are slightly convoluted because of the very small number of pins available. Here is the process for accessing the data memory:

By the end of the second clock cycle, the 10-bit output bus will get the address that we will be reading or writing. The read/write bit will also be set.

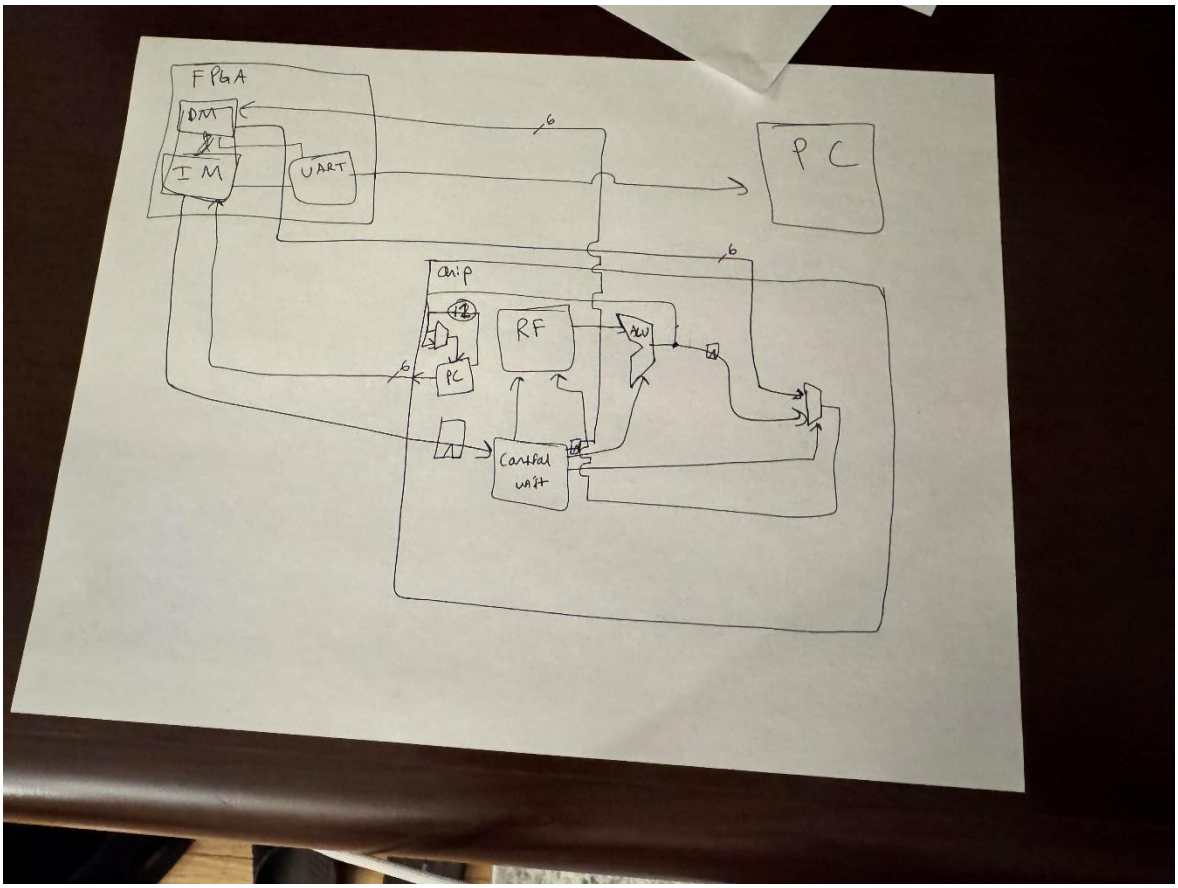
At the start of the third clock cycle, the CPU sets the write commit bit if the previous operation was a write. Eight of the 10 bits on the output bus will now be the data to be written to the memory. If the operation was a read, the read bits will be available and will be written back to the CPU.

Question #4:

On the host FPGA, there will be both a unified instruction/data memory. There will need to be some logic to handle the multi-cycle write operation. The host FPGA will also have a basic UART interface to a host PC so that we can program the instruction memory and read the results out of the data memory.

Question #5: I am not going to provide a module header here but instead draw a block diagram.

This block diagram has the pipeline registers added after the fact. This isn't a required part of this assignment, but I wanted to hopefully give a better idea of what I am envisioning.



I would like feedback on this proposal. First: Is there a better way to increase the number of effective I/Os for memory write other than the pipelined approach I am doing.

Second: Is this going to be too large of a processor (in terms of die area) for this project? What is an easy way to tell? I should be able to scale it down (or up) as needed.

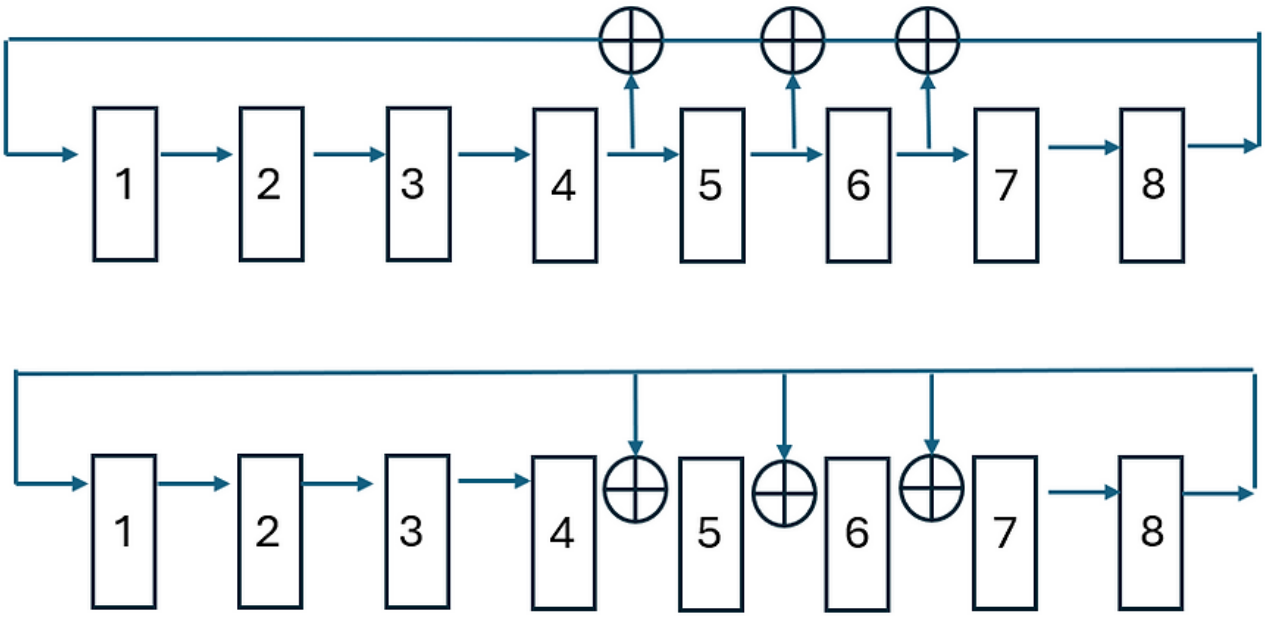
Proposal idea #2 [BACKUP PROPOSAL]:

For my backup proposal, I will propose a linear feedback shift register tree.

Question #1:

Title: Linear feedback shift register tree.

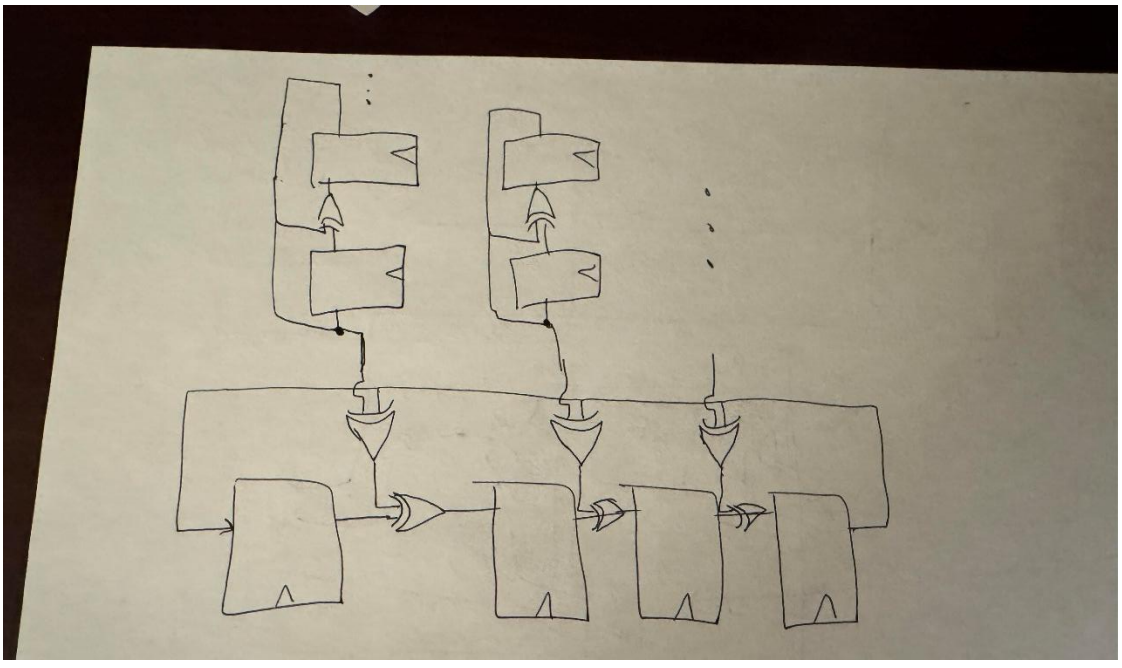
What will it do: We're all familiar with a standard linear feedback shift register. It looks something like the following:



Now, during the brainstorming session, I was thinking if there might be some way to improve this. I came up with a cool idea. I am not sure if it is actually any better than just a normal LFSR (it might be worse), but it could be cool to make. The idea is to make the LFSR a 2 (or more) dimensional array instead of a 1-D array. The output of each LFSR chain would become an input that is XORed to the next lower level.

Question #2: Technical description:

Here is a sketch that gives a better understanding of what I am envisioning. You can imagine it going to more than two dimensions (I just drew two here).



Here is my idea: I will dedicate half of the available chip area to one of these “multi-dimensional LFSRs”. The other half will then be for a standard LFSR of the same area. I will have a multiplexor at the output so I can choose which of the outputs I want to use. I will have a size 12 array of each of these types of LFSRs to produce 12 bits of output

Question #3: The I/Os will be as follows:

Output from Chip: 12 bits of LFSR output

Input to chip: 1 bit to choose between standard LFSR and LFSR “tree”. 11 bits unused.

Question #4:

I will need to have to offload some work to the host FPGA. The host FPGA will need to read out the LFSR values from the chip, and send them over UART to the host PC for analysis. Alternatively, the analysis algorithm might be able to be done directly on the FPGA, although this isn’t my primary plan.

I can then analyze how “good” the randomness is for both types of LFSRs to compare them.

Question #5:

I do not have a module header prepared at this time, but this is optional.