

EECS 151/251A Homework 8

Due Monday, April 12th, 2021

For this Homework

Please include a short (1-2 sentence) explanation with your answer, unless otherwise noted.

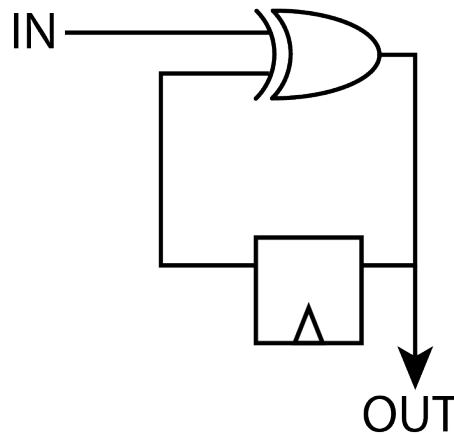
Problem 1: Loop Unrolling

Let's take another look at the Parity Checker FSM from Lecture 7. Most FSMs are loops, so we can also apply the same loop unrolling method to them as well.

- (a) Modify the Parity Checker FSM to be a Mealy Machine implementation. Show a circuit diagram of your modified design.

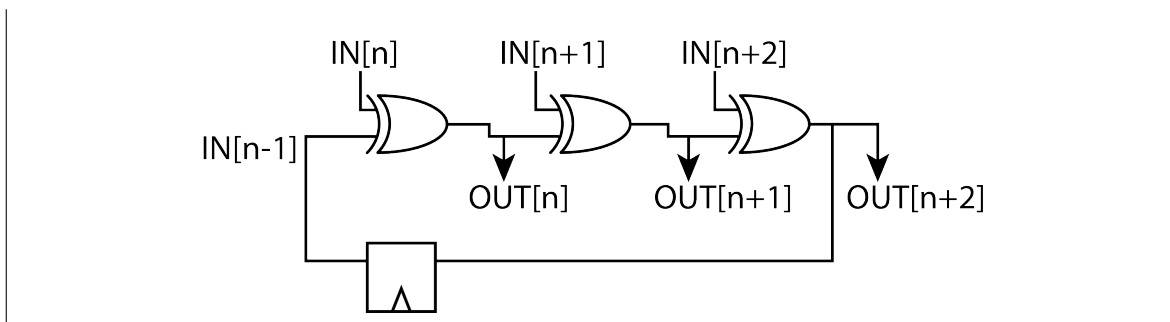
Solution:

To make this a Mealy machine, we simply just move the output to before the flip-flop.



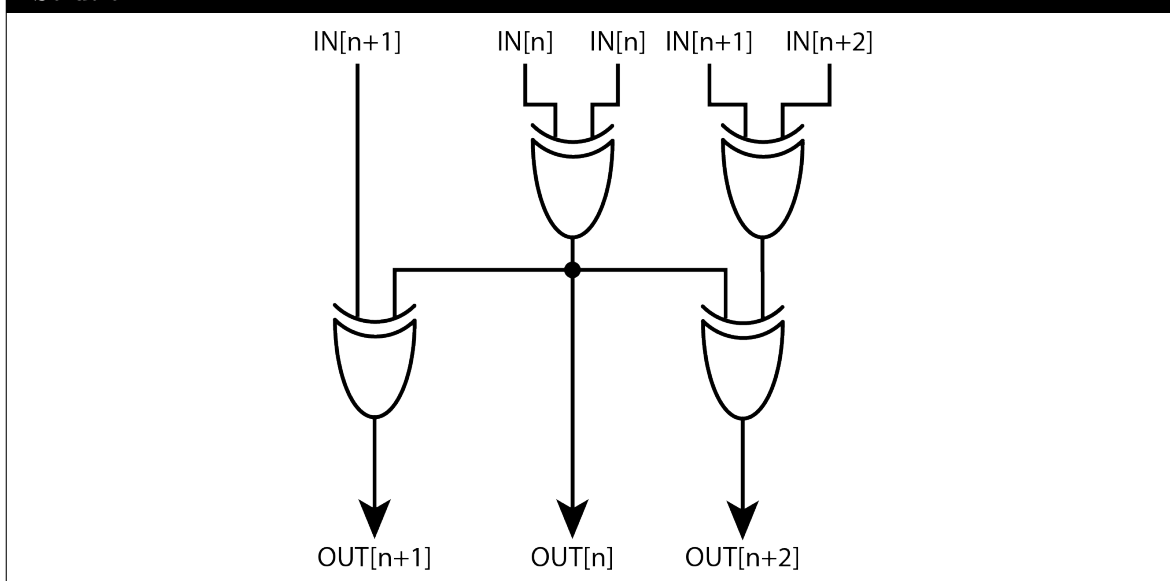
- (b) The input to the Parity Checker now arrives 3 bits at a time. Apply the loop unrolling technique to increase the throughput of the design. Show a circuit diagram of your modified design.

Solution:



- (c) The unrolled loop has a long critical path from the registered feedback value back to the input as this must pass through all the parity check stages. Is there a way you can modify the design to have a shorter critical path? Discuss how you would make the modification. (*Hint: Tree structures*)

Solution:



Problem 2: C-Slowing

In digital signal processing, a common technique is to employ filter banks, which are bandpass filters used in parallel with the same input stream to decompose the input into its frequency components. One such filter topology is the biquad filter (Wikipedia Article), which is so named as its transfer function is the ratio of two quadratic functions (i.e. it has 2 poles and 2 zeroes). The filter bank in question is thus made up of several of these filters all working in parallel.

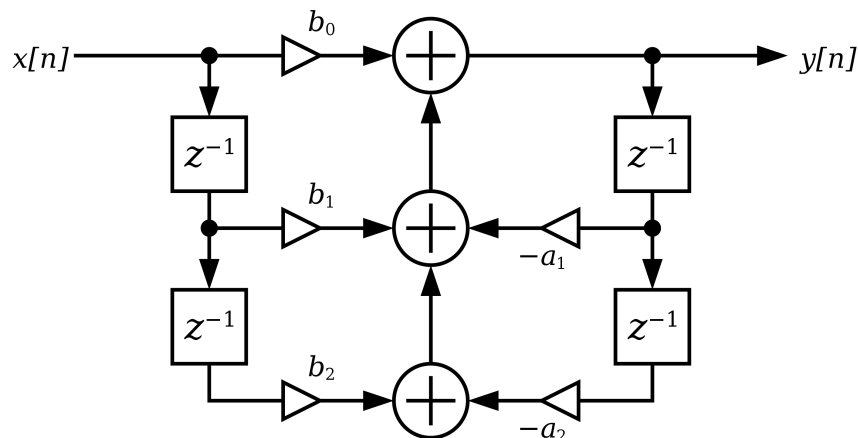
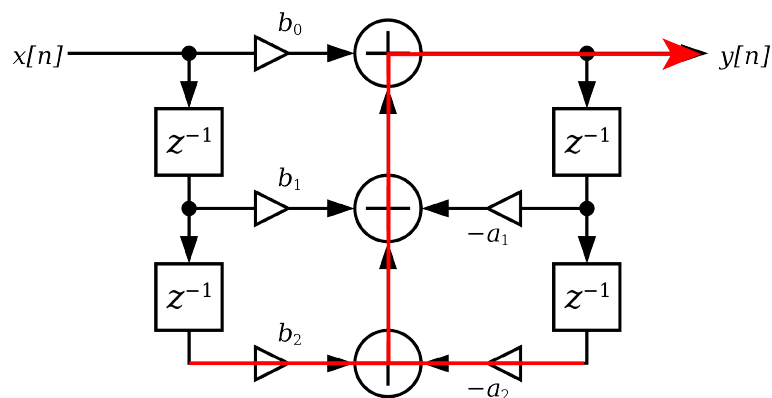


Figure 1: Block Diagram of a Digital Biquad Filter

Where z^{-1} represents 1 cycle of delay, and the triangle amplifiers represent multiplication.

- (a) What is the critical path of the biquad filter design?

Solution:

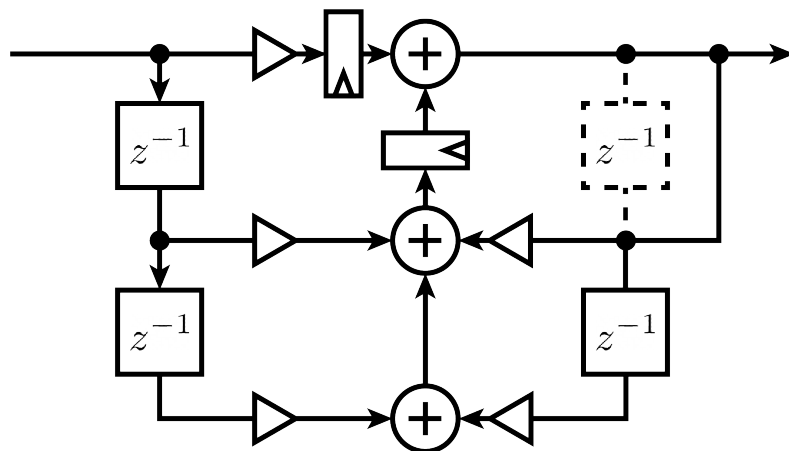


- (b) Assuming the multipliers and adders cannot be pipelined, how would you apply C-slowng to maximize throughput? What is the optimal value for C?

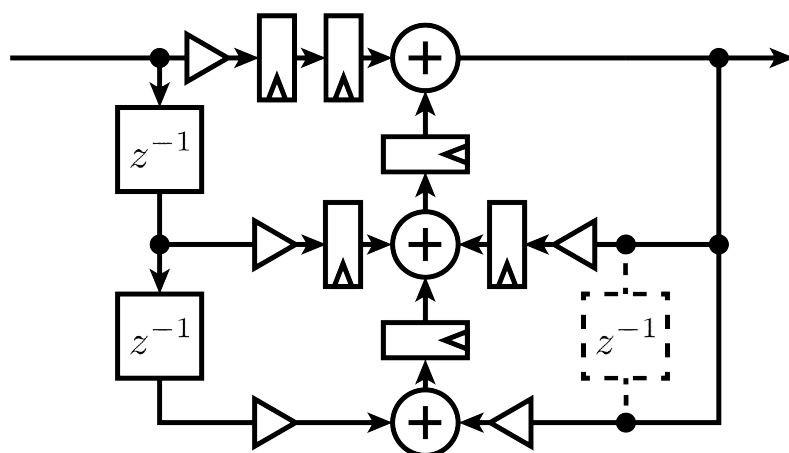
Solution:

We can pipeline before all the adders. Since the input will depend on previous values of the input, we will have to save this if we mux to a different input, so there is some parallelization that must happen on the feed-forward path. With this circuit, the maximum value for C is 3.

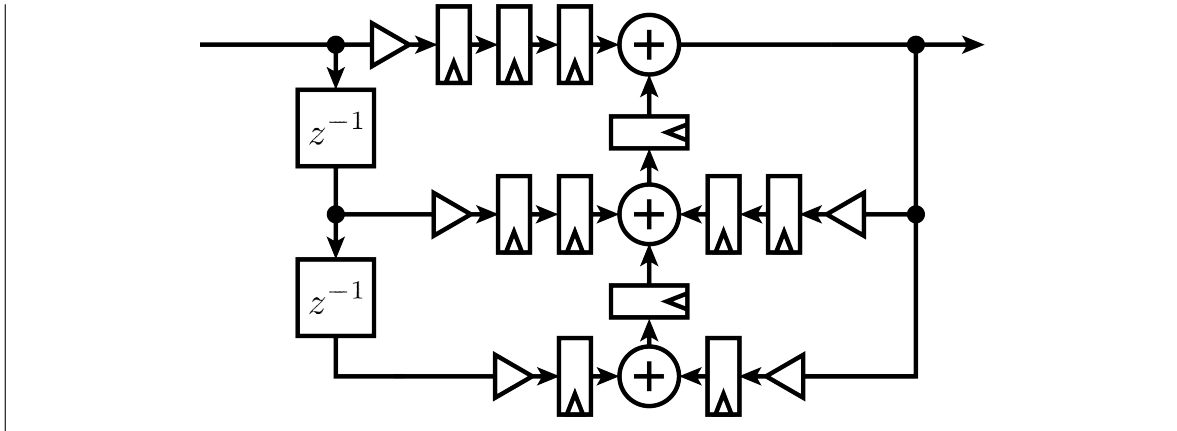
We can pipeline the first adder as shown below.



Note that because we have pipelined the adder, the result of the addition is delayed by one cycle, and therefore the output is now $y[n - 1]$. We therefore can drop the first delay element in the feedback path. We can continue in the same fashion for the second pipeline stage.



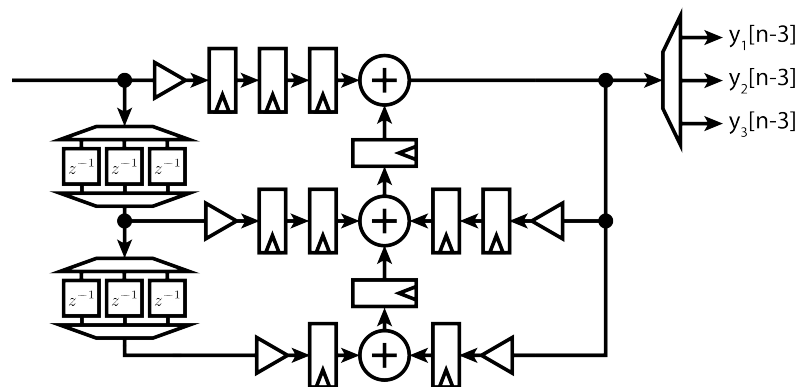
Again, we can drop the feedback delay element as the output is now delayed by 2 cycles. Somewhat counterintuitively, we need to delay the $y[n - 1]$ path to keep the delay along this line correct. This is because the second adder has been pipelined, which means it is processing the sum a cycle behind. By the time the data is passed to this stage, the *current* $y[n - 2]$ will become the next $y[n - 1]$, just in time for the second adder to accept the value. We can then apply the same pattern to pipeline the design one more time.



- (c) Show a block diagram of your design, including any additional blocks/circuits you need to achieve the C-slowng.

Solution:

The previous solution will allow us to C-slow at $C = 3$. In order for this to be possible, we need to take care of the feedforward path. The delay elements here depend on the next input each cycle, so we can't have them accept samples from different sources at different clock cycles. In order to C-slow, we will need to introduce some amount of parallelization. One way is to triplicate the delay elements and sequentially activate the one corresponding to each input, and then mux the output to each respective output signal.



Another approach would be to unroll the feedforward path and take 3 samples of the input at a time ($\{x[n], x[n-1], x[n-2]\}$). This will also allow us to choose between 3 separate inputs.

Problem 3: FIFO Redesign

A known issue with the basic FIFO design is that a naïve solution may have an intolerably long delay from clock edge to expressing the full/empty signal. This could potentially be an issue for both the producer and the consumer of the FIFO. On the producer side, the logic for generating the next data packet may be long enough that it needs to see a valid full flag near the beginning

of the clock cycle. Likewise, the consumer may also need to know if the queue is empty early on in the clock cycle.

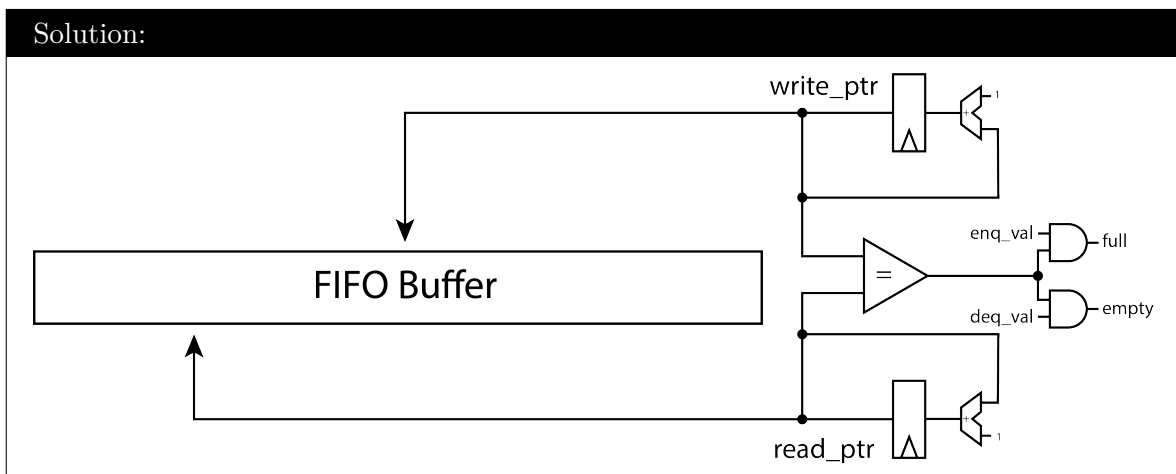
For the parts below, the memory blocks will be simple dual port (single asynchronous read, single synchronous write). The read/write pointers are implemented with a wrap-around counter that you should represent as a fixed block (i.e. the counter cannot be redesigned). The FIFO is instantiated with the following interface:

```

module fifo #(
  parameter N
)(
  input clk,
  input rst,
  input [N-1:0] enqueue_bits,
  output [N-1:0] dequeue_bits,
  input enqueue_val,
  output enqueue_rdy,
  output dequeue_val,
  input dequeue_rdy,
  output full,
  output empty
)

```

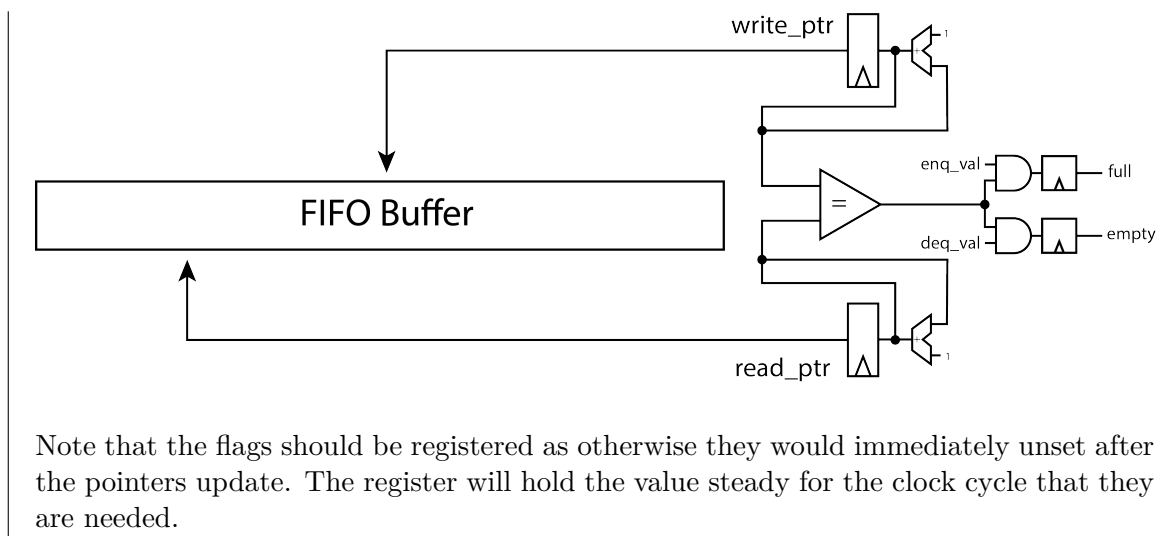
- (a) Show a block diagram of a naïve 8-bit wide, 16-entry deep FIFO that does not address the long full/empty flag delay.



- (b) Modify your design from part (a) to solve the long delay problem. Explain your design decisions and the solution that your improved FIFO implements in 4-5 sentences.

Solution:

To allow the flag to be expressed as soon as possible, we must precompute the flag by incrementing the write and read pointers one cycle ahead when doing the comparison.



Problem 4: Cache Review

Design a 64x64 fully associative memory with 64-bit words (1 word per line) using flip-flops and logic gates. The memory address is 8-bits long, however, so the processor may miss when requesting data or writing data to the memory. The memory will behave in the following way when encountering such a situation:

- In the case of a read miss, the memory is to express a `NO_DATA` flag to indicate to the processor that the memory address is invalid.
- Each memory location has a corresponding `valid` bit. When data is written to a memory location, its corresponding `valid` bit is set. The memory initializes with all `valid` unset.
- If the processor attempts to write to the memory, it will overwrite the first (in ascending order of memory line locations) non-`valid` line in the memory.
- On a write, if all the memory locations are valid, the memory will use a replacement policy called "not recently used" (an approximation of "least recently used"). The policy is implemented as follows:
 - Whenever a memory location is accessed for a read, a corresponding `used` bit will be set for that location.
 - A shift register will rotate through the memory locations in descending order, pointing to a single memory line at a time.
 - At the beginning of the clock cycle, the memory line that the shift register is pointing at will have its `used` bit unset.
 - When writing to a fully valid memory, the first (in ascending order) non-used line is overwritten.
- The "find first row" operation for both finding the first invalid line or first non-used line should happen combinational (i.e. within 1 clock cycle). *Note: You don't need to worry about the delay for this circuit, just make it as simple as possible.*

For this problem please turn in:

- A block diagram of your design. You may represent the flip-flops in the memory as a large block, but indicate how the flip-flops are distributed (data bits, valid/used bits, tag, etc.). The "find first row" operation logic should be drawn at the gate level using basic logic gates (4 input maximum) and MUXes. Also, be sure to indicate the input and output pins clearly.
- An overview (1-2 paragraphs) of how the memory operates, making note of the design decisions you made.