Grant Lance

ECE 474 Spring 2024

Homework Assignment 6

FIFO Memory Implementation

Section 1 - Original Attempt/Submission

Note that this document is in two sections. The first details the original submission. The second section goes over the changes made to the original submission given the extension. The original submission namely didn't do a few things:

1) Make sure that the read and write pointer were synchronized before continuing
2) Use gray codes to index the memory cells
3) Allow for read and write operations to occur simultaneously (given that there are memory cells to read and that a user isn't reading and writing to the same cell at the same time).

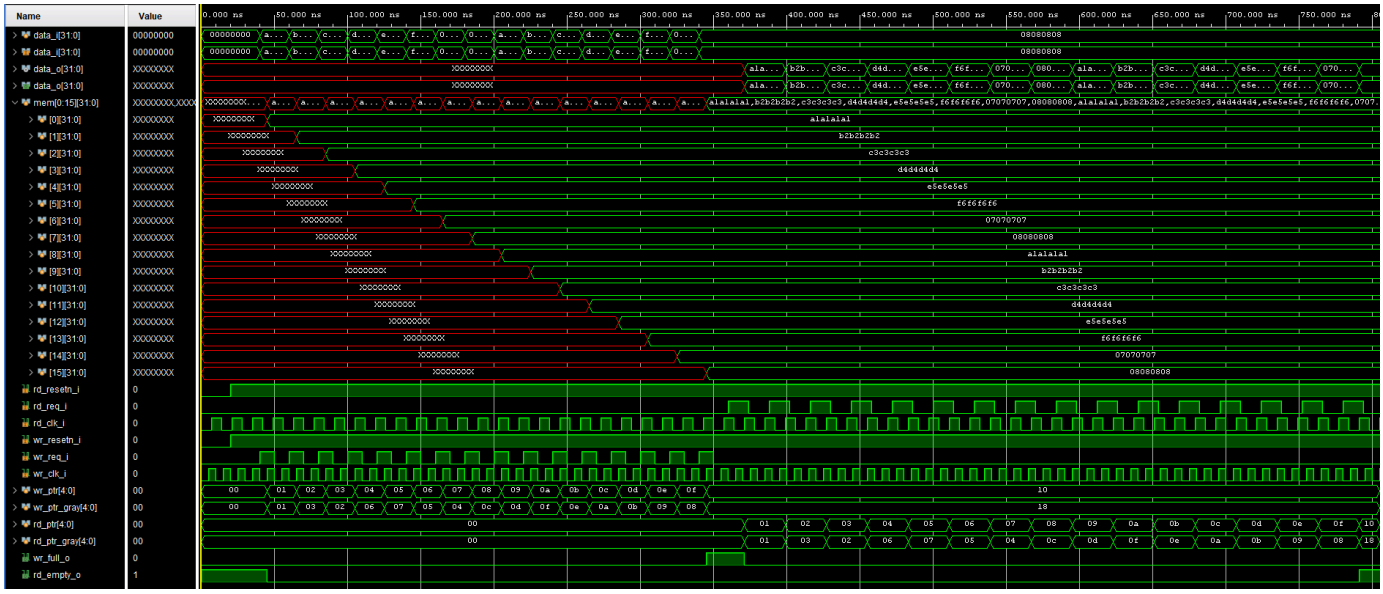


Figure 1: Screenshot 1

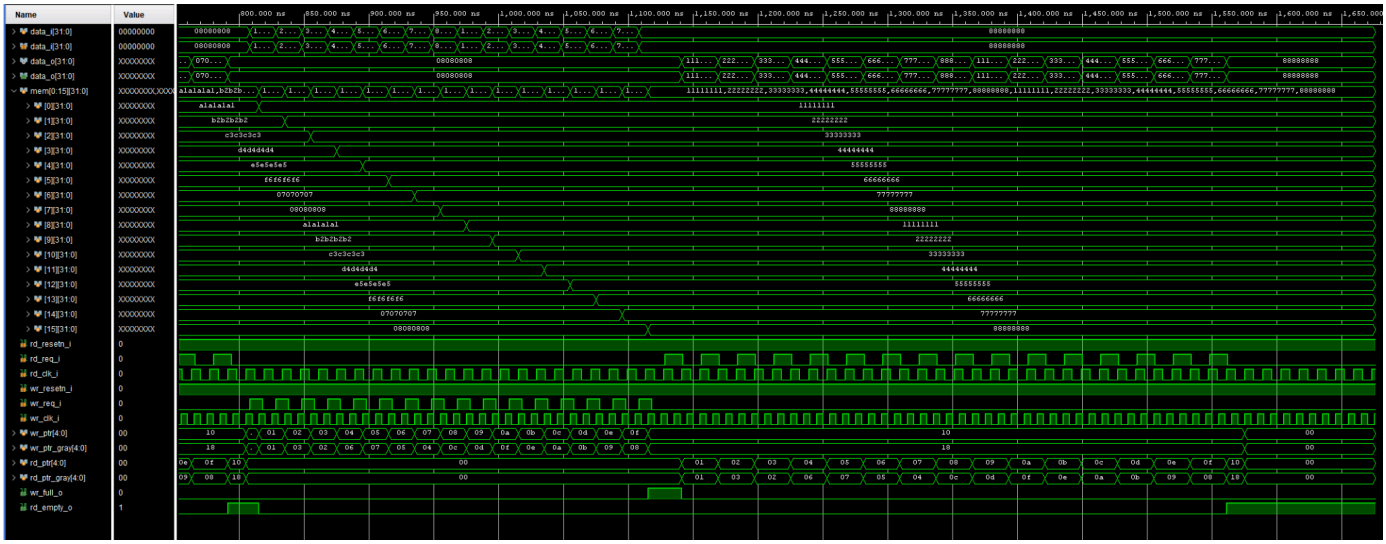


Figure 2: Screenshot 2

1. Verify for correct Binary to Gray code conversion
    a. Looking at the above screenshots (1 and 2), and Table 1 in the appendix. It can be seen that the pointer and gray code is incremented and converted correctly. The pointer is interpreted in hex. Using the table it can be seen that they both increment from zero to 16 (one bit past the final address). Screenshot 3 shows the gray codes with more zoom.
2. Check that the FIFO full status flag/empty flag is asserted when FIFO becomes full
    a. The bottom two output waveforms (Figure 1 and 2) are the signals for wr_full_o and rd_empty_o. These are the two flags that help the FIFO manage its memory. This entails maintaining input/output order and determining the state of the cells (full/empty). Note that this FIFO does not actually re-write the data cell to zero after the data has been sent out. Because of this the cells are still technically "full" but the FIFO see them as "empty" or writable. The advantage of this is that it is very easy to determine the full/empty state from the users point of view with ease. Rather than determining the the cells are full/empty (data/zero) the FIFO, upon reset, will reset its write pointer causing the FIFO to write over whatever it had originally saved. Further, the pointers/gray codes take advantage of the inherent bit width of the addresses. By being one bit larger than the address, it is very easy to see when the pointer is out of bounds. For the read pointer this is simple. In this case it is reset. When the write pointer enters this case, the pointer is reset and the

full flag is raised so that the FIFO does not write to its head on the next clock cycle. Had the flag not worked correctly, the FIFO would have reset the write pointer and data would have been written to cell 0 before being emptied. However, the FIFO will not reset the write pointer to zero until the full flag is lowered. The same can be said about the empty flag and the read pointer. Using this logic the full flag is raised when no more cells are writable and the empty flag is raised when all cells are writable (no data for output).

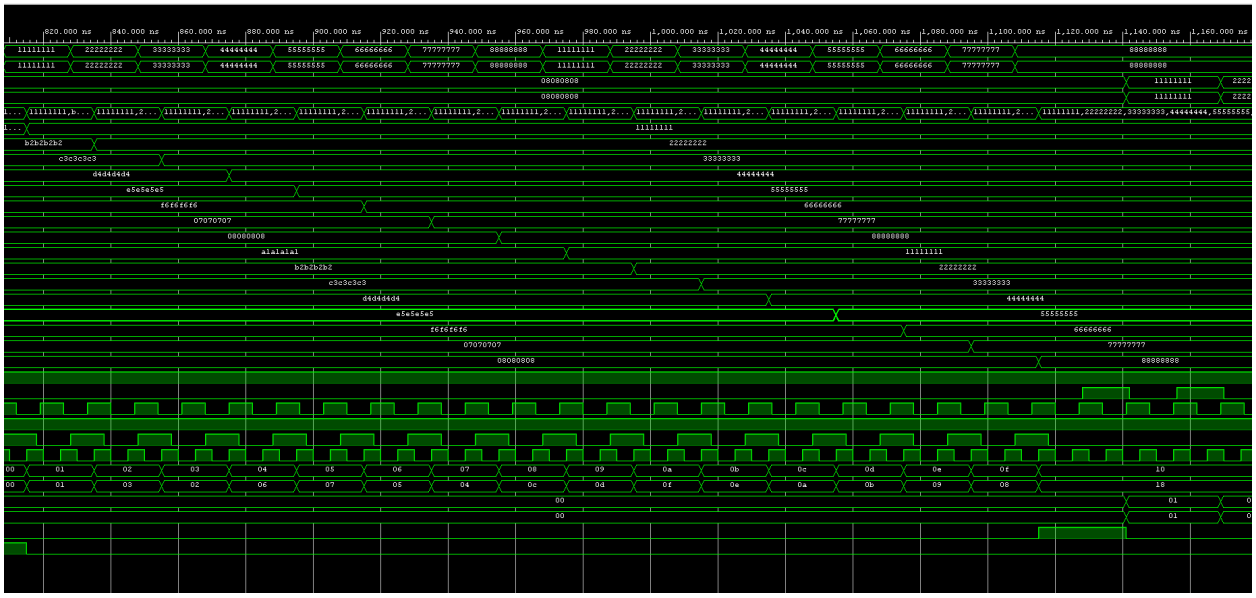


Figure 3: Screenshot 3

Hex	Binary	Gray	Gray (as hex)
0	0	0	0
1	1	1	1
2	10	11	3
3	11	10	2
4	100	110	6
5	101	111	7
6	110	101	5
7	111	100	4
8	1000	1100	C
9	1001	1101	D

A	1010	1111	F
B	1011	1110	E
C	1100	1010	A
D	1101	1011	B
E	1110	1001	9
F	1111	1000	8
10	10000	11000	18

Table 1: Table shows incrementing gray codes with corresponding hex values. Gray as hex shows what the output waveform values would be (Gray codes converted to hex).

Section 2 - Final Attempt/Submission

As mentioned previously, the changes to the FIFO include:

1) Make sure that the read and write pointer were synchronized before continuing
    a) In order to do this two registers were created to flag the status of the read and write pointers. Without the pointer synchronized, the FIFO will not complete a read/write operation. This protects the data from being corrupted
2) Use gray codes to index the memory cells
    a) Previously, the pointer was used to index the cells in the FIFO. This was changed so that the gray code is passed to the FIFO synchronization block and is then incremented by the block.
3) Allow for read and write operations to occur simultaneously (given that there are memory cells to read and that a user isn't reading and writing to the same cell at the same time).
    a) This is made possible by a combination of read/write pointer synchronization and making use of the cell status. This way the FIFO can read/write at different time intervals, but not corrupt the input/output data.

Below are screenshots 4 and 5 - from the end of the new test bench. It is similar to the original test bench except for the addition of cell_status and synchronization registers. The end of this screenshot shows how the FIFO can read and write at the same time. Notice that rd_req_i and wr_req_i are both being triggered at the same time. At the start of this section there is some data in the FIFO, so read starts there. As the reading is being done faster than the writing however, the read pointer eventually catches up with the write pointer (notice the shorter blips in cell_status - where 1 means cell is in use by memory). When the read pointer catches up with the write pointer, the flags make the read operation wait until the write operation is completed. This

way the user is not reading an empty cell. Otherwise, the FIFO will output 0 to data_o. This is managed in part by cell_status and in part by the synchronization registers.

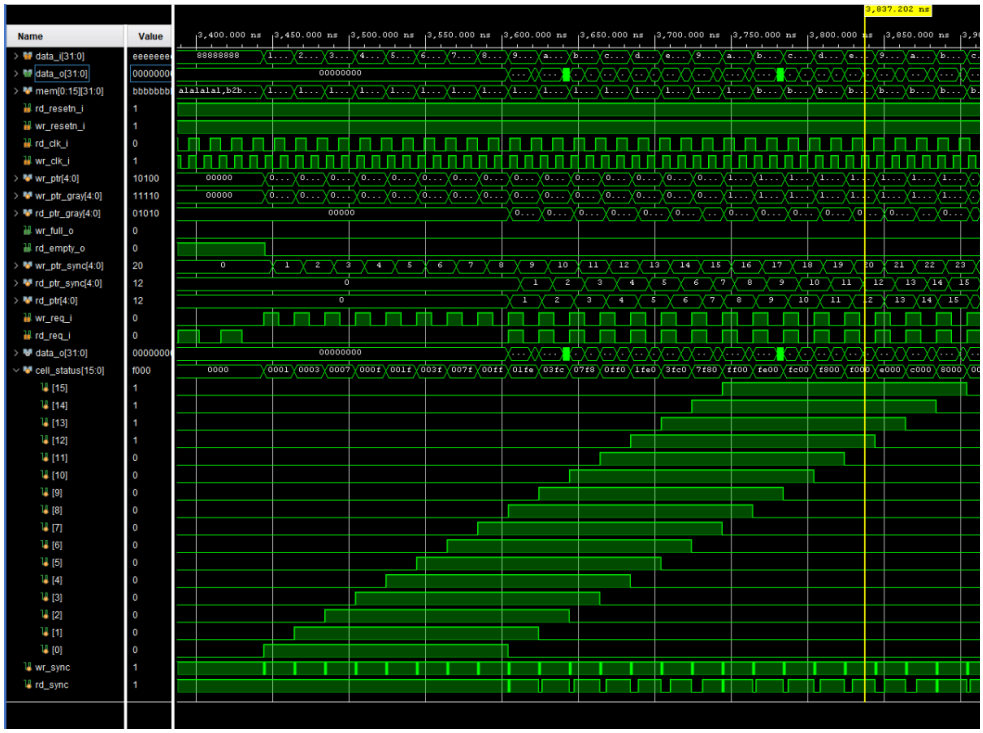


Figure 4: Screenshot 4

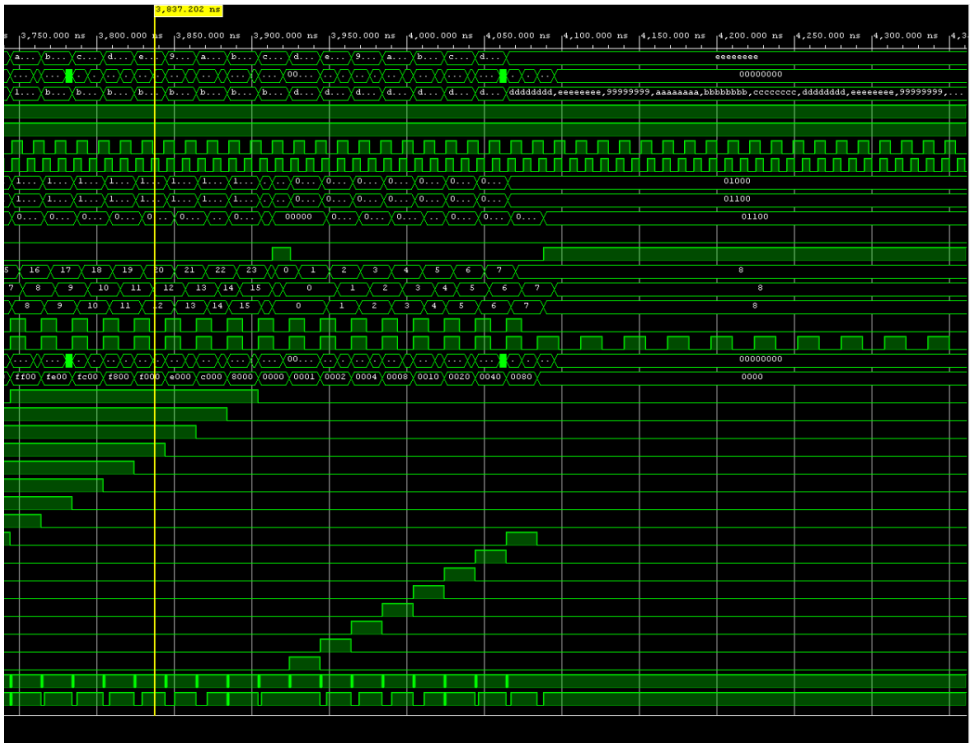
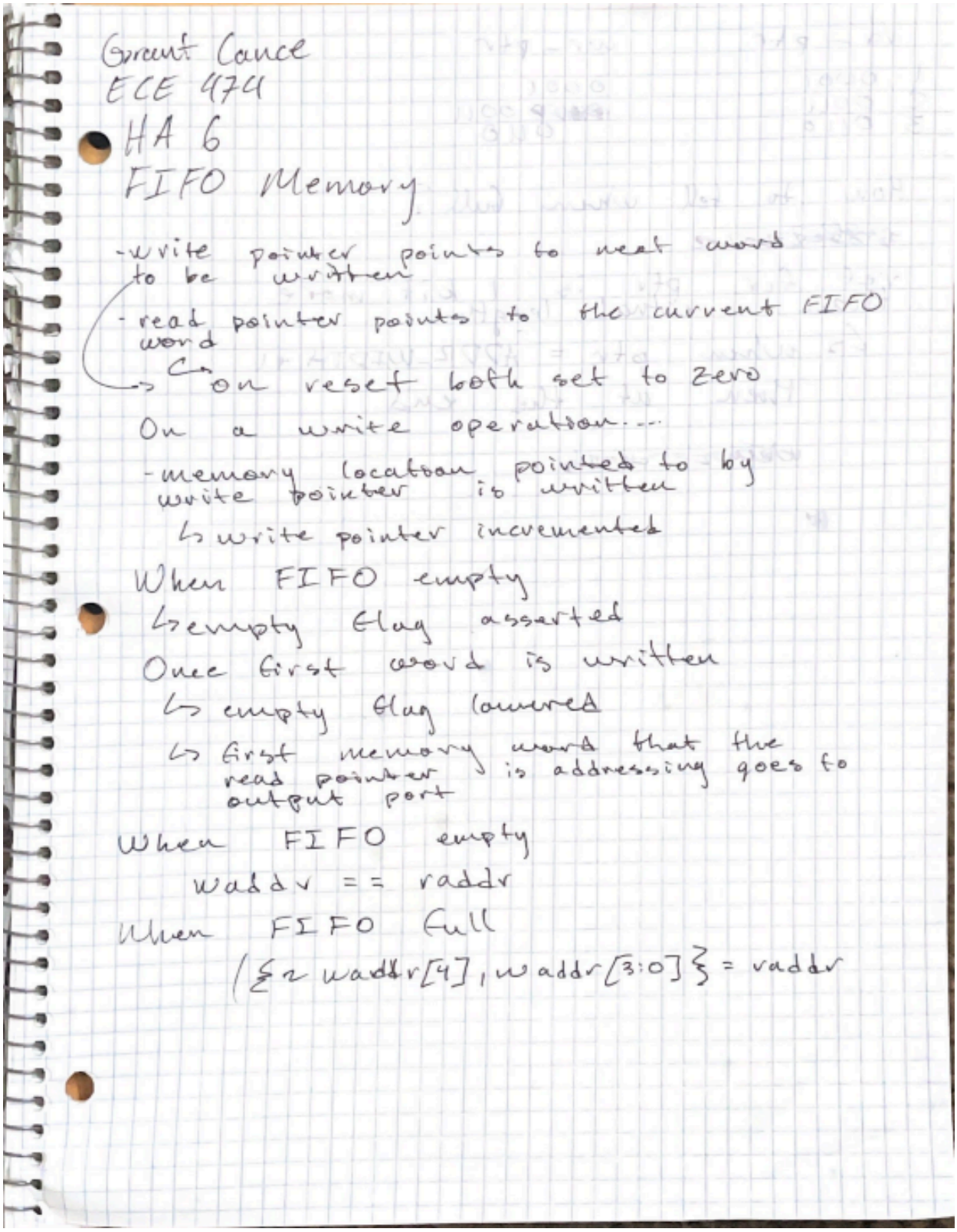


Figure 5: Screenshot 5

Note Appendix:



Grant Cance
ECE 474
HA 6
FIFO Memory

- write pointer points to next word to be written
- read pointer points to the current FIFO word
  - ↳ on reset both set to zero

On a write operation....
- memory location pointed to by write pointer is written
  - ↳ write pointer incremented

When FIFO empty
- ↳ empty flag asserted

Once first word is written
- ↳ empty flag lowered
  - ↳ first memory word that the read pointer is addressing goes to output port

When FIFO empty
waddr == raddr

When FIFO full
{~waddr[4], waddr[3:0]} = raddr

	rd_ptr	wr_ptr
1	0001	0001
2	0011	0011
3	0110	0110

How to tell when full?

~~[illegible]~~

ref for ptr is 1 bit more than actual length

<> when ptr = ADDR_WIDTH + 1 then at the end

~~[illegible]~~