Grant Lance

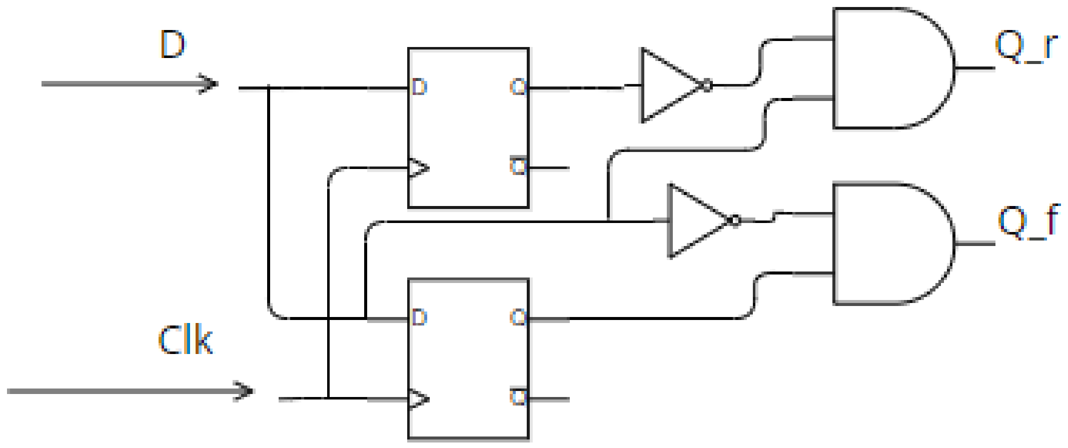
ECE 472 - Spring 2024

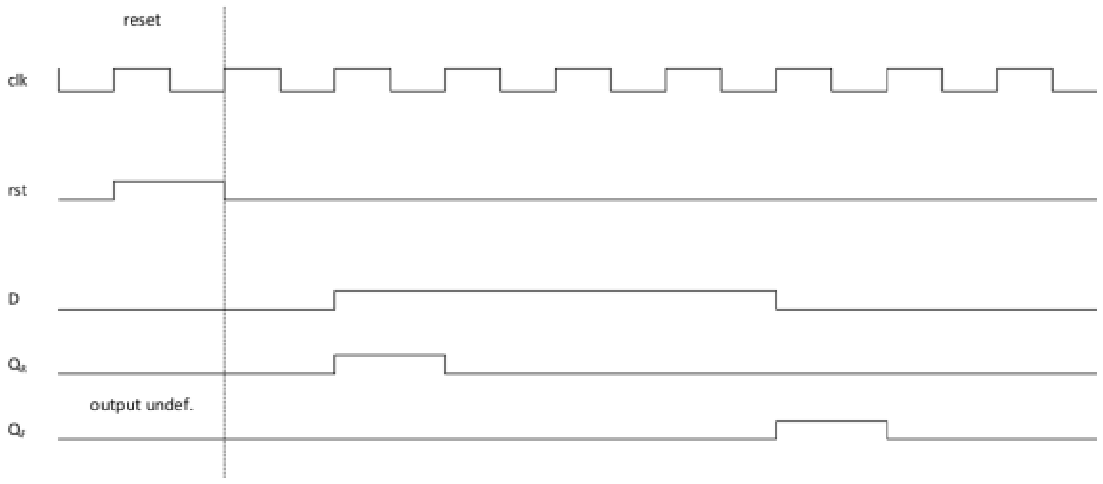
April 18, 2024

Homework Assignment 2

1) **detect_edge**

   a) Gate Level Diagram



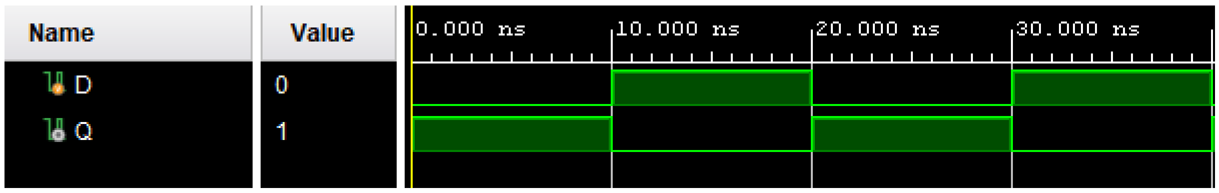


i) Above is the gate level diagram for the detect_edge block. I used a free, online tool to create gate level diagrams- they had limited choices for DFFs. I will be using my DFF from HA1 which includes an enable, reset, and a reset value. Also, it has no inverted output. That being stated they would be arranged in the same way. Assuming the DFF is initialized with a value of zero, the following would occur. When the signal D is raised to one, the upper DFF will change its output from zero to one. This output will be inverted. At the positive (rising) edge of D, this will cause the AND gate (upper) to see 1 and 1 at this time. This will cause the output of Q_r to go to one for a clock cycle. At the negative edge of D, the lower AND gate will see a negated D (which is 1) and the 1 from the Q of the lower DFF. This will result in Q_f being 1 for a clock cycle.

b) Vivado Implementation

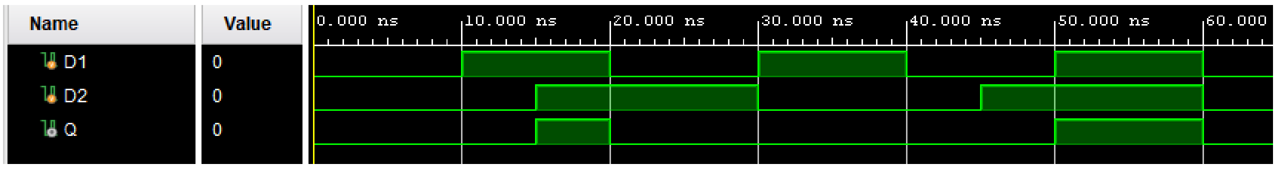
i) NOT/Inverter

(1) To match the gate diagram created, a NOT/inverter was implemented and validated. A screenshot of the test bench output is below validating that the NOT functions as intended.



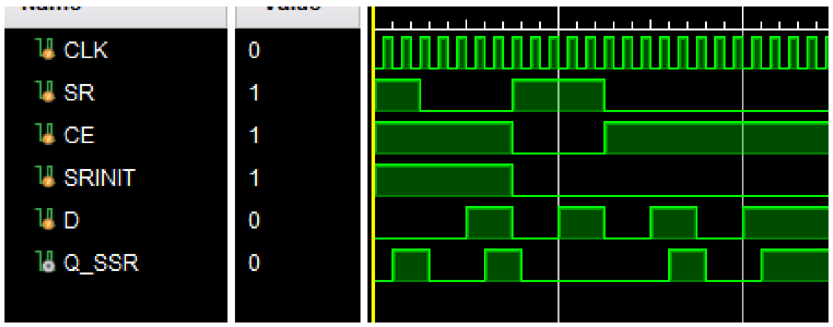
ii) AND

(1) Following the design of the gate diagram, an AND needed to be implemented. Like the inverter, a screenshot of the test bench is below validating the function of the AND gate.



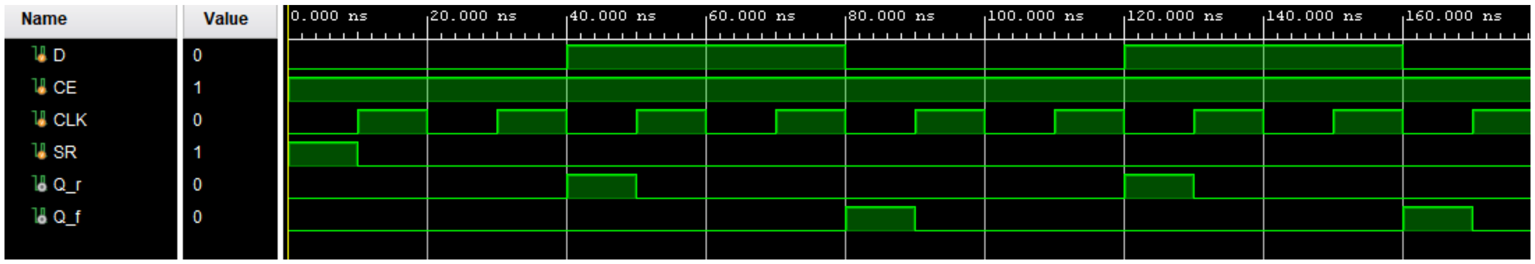
iii) DFF

(1) The DFF used for this assignment came from HA1. Per the assignment handout, modules from that assignment may be recycled. A change was made so that if SRINIT is not initialized or set (not 1), then the output will default to 0. A screenshot of the test bench below demonstrates this functionality. Also the DFF was switched to trigger at posedge instead of negedge.



iv) All combined to create detect_edge.v

(1) After combining all three modules, a test bench was created and the function of the block was validated. As can be seen below the screenshot of the output waveform for my block matches the desired output from the assignment handout.



2) **nxmBitShiftReg**

a) Basically, there are four operations/things this block needs to be able to do:

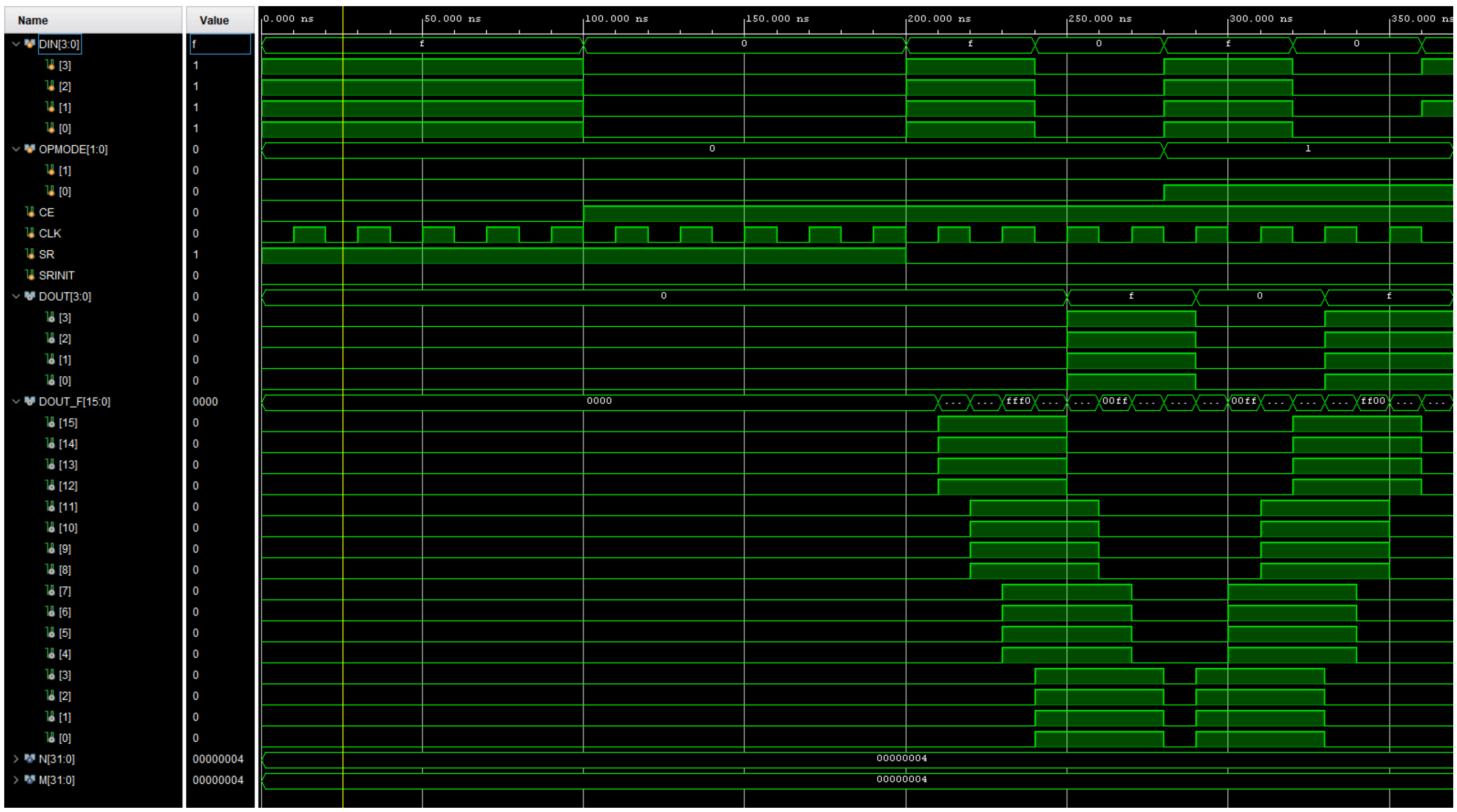
i) Concatenate a new word onto the head of DOUT_F so that all bits appear to “shift” to the right. The word that is “pushed off” will go to DOUT.

ii) Concatenate a new word onto the tail of DOUT_F so that all bits appear to “shift” to the left. The word “pushed off” will go to DOUT.

iii) Concatenate the head of DOUT_F onto the tail and vice versa (causing the bits to appear to “rotate” left or right).

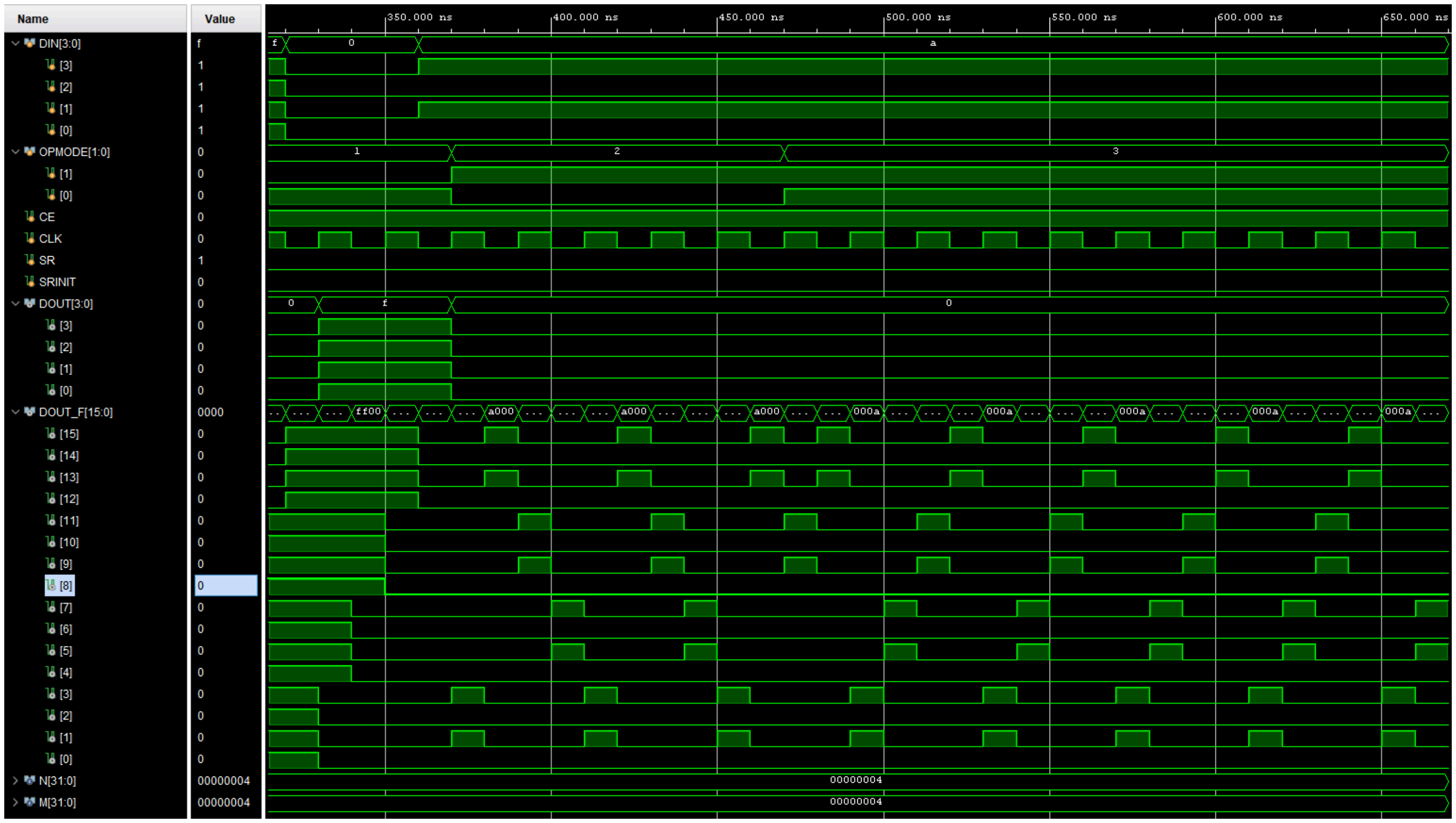
b) In order to dynamically scale, the concatenation was done in terms of the parameters N/M (word size and width).

c) The screenshots on the next page shows the block performing as expected.For more on the bitwise operations see the handwritten notes in the appendix.



d) Above shows about the first half of nxmBitShiftReg_tb.v During the first part of the simulation it can be seen that the block does nothing while CE != 1. Later as it starts to "shift" to the right, DOUT is zero (as that is what is being "pushed off"), and the bits "shift" to the right. The reverse occurs just after. '

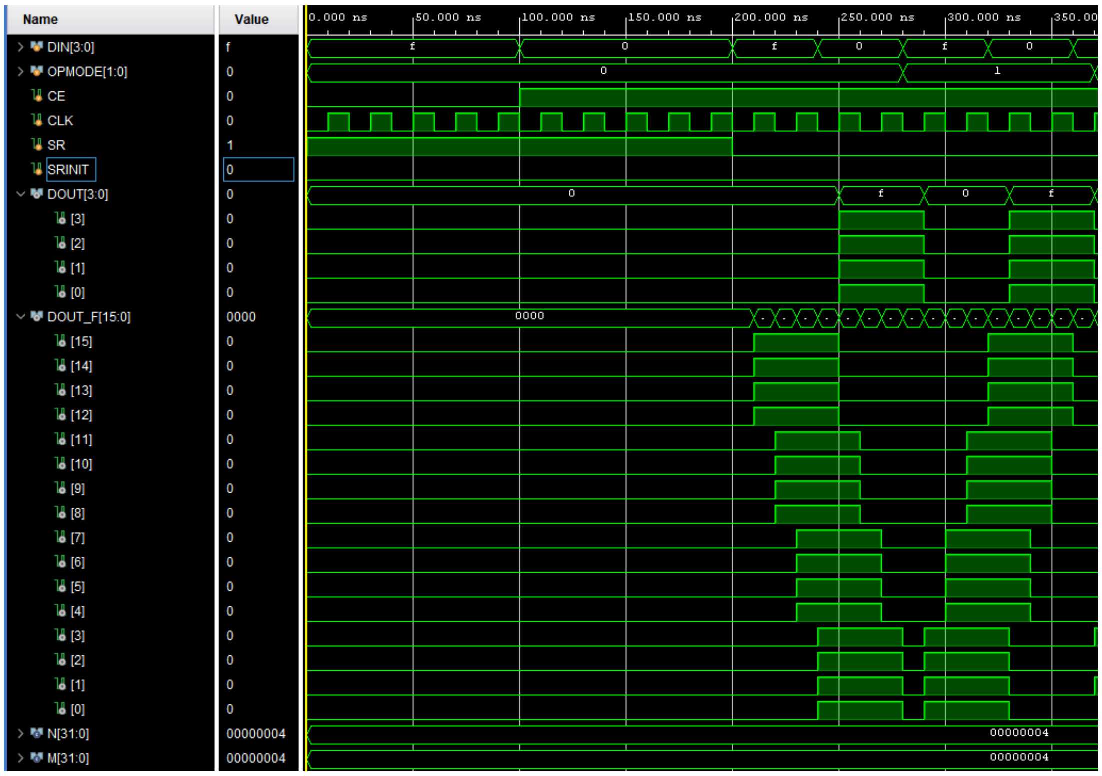
   i) Note after emailing with the TA I learned that what is pushed off should be sent to DOUT. Otherwise, from what I gathered my bitwise operations were correct. See appendix C/D. The output of the updated block with corrected DOUT output is below.



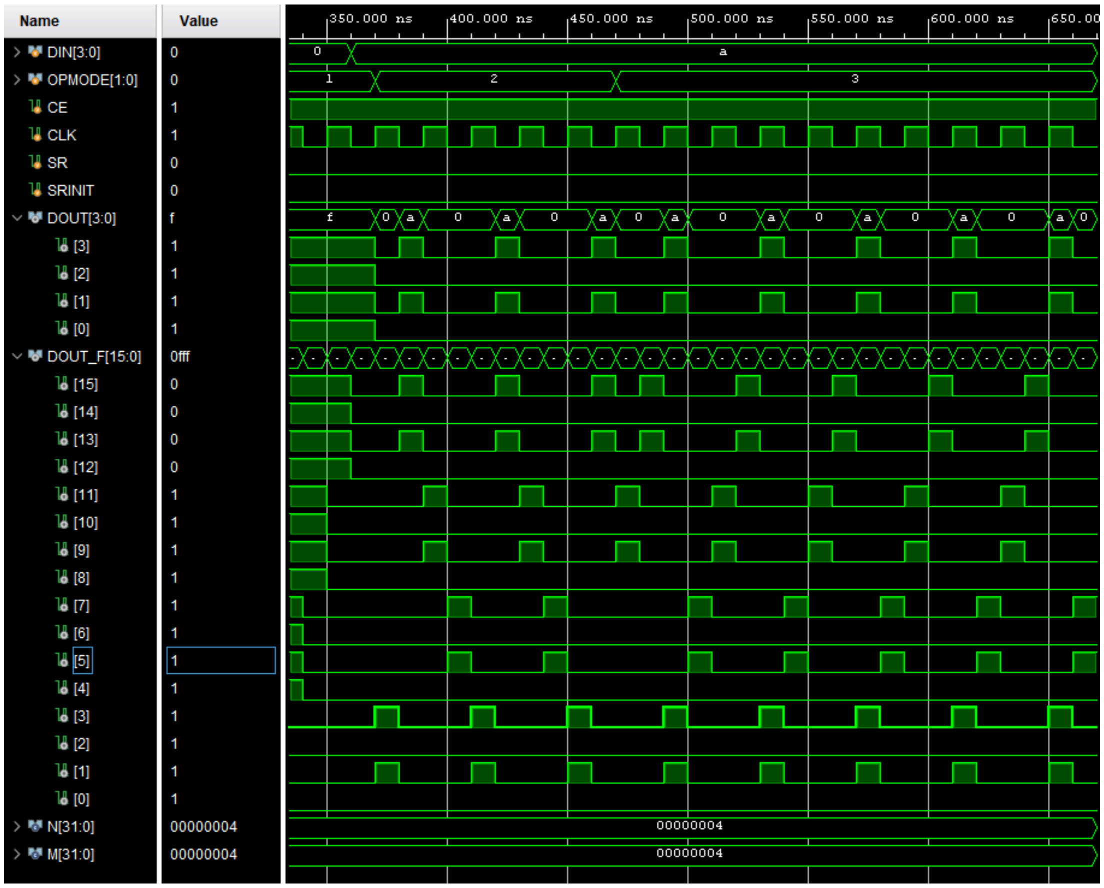
e) Above shows the rotation left and right working as you would expect. Decimal 10 (4’b1010) was used as it is different from decimal 15 (4’b1111) but still easy to see when debugging. The screenshot above shows that the word is rotated around left and right. DOUT is zero as nothing is “pushed off” in this operation.

   i) Note after emailing with the TA I learned that what is pushed off should be sent to DOUT. Otherwise, from what I gathered my bitwise operations were correct. See appendix C/D.The output of the updated block with corrected DOUT output is below.

f) Below is a screenshot of the output where DOUT has been corrected to include what has been “rotated out” during left and right rotation.

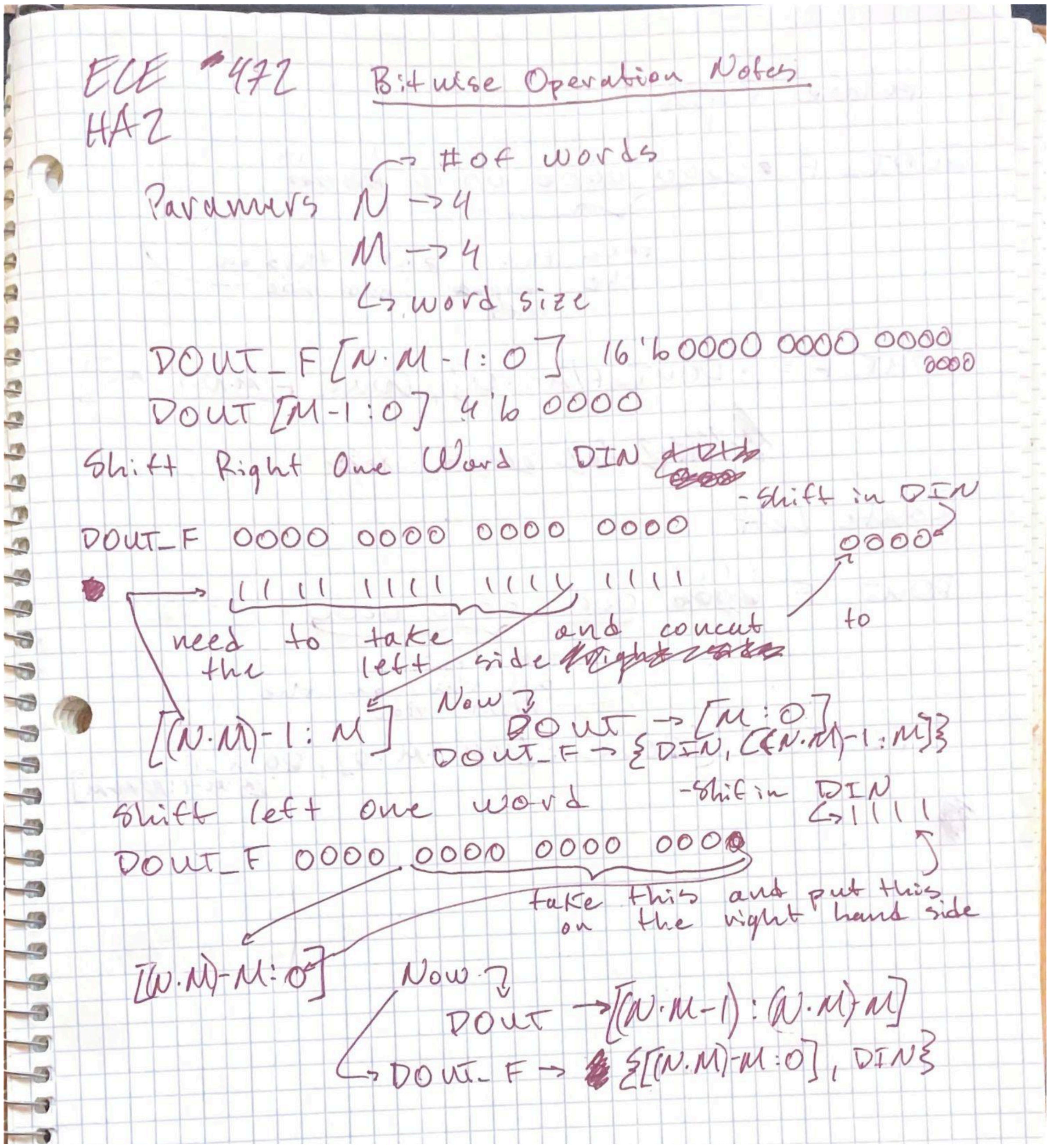


Name	Value
DIN[3:0]	f
OPMODE[1:0]	0
CE	0
CLK	0
SR	1
SRINIT	0
DOUT[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0
DOUT_F[15:0]	0000
[15]	0
[14]	0
[13]	0
[12]	0
[11]	0
[10]	0
[9]	0
[8]	0
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0
[0]	0
N[31:0]	00000004
M[31:0]	00000004



Name	Value
DIN[3:0]	0
OPMODE[1:0]	0
CE	1
CLK	1
SR	0
SRINIT	0
DOUT[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1
DOUT_F[15:0]	0fff
[15]	0
[14]	0
[13]	0
[12]	0
[11]	1
[10]	1
[9]	1
[8]	1
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	1
[1]	1
[0]	1
N[31:0]	00000004
M[31:0]	00000004

3) **Appendix**

a) 

ECE #472 Bitwise Operation Notes
HA2

Paramers $N \to 4$ → # of words
$M \to 4$ ↳ word size

DOUT_F [N·M-1:0] 16'b0000 0000 0000 0000
DOUT [M-1:0] 4'b 0000

Shift Right One Word DIN ~~& DIN~~ ~~0000~~

- Shift in DIN

DOUT_F 0000 0000 0000 0000 0000 to

1111 1111 1111 1111

need to take the left side and concat ~~Right~~ ~~[illegible]~~

[(N·M)-1 : M]

Now ↴
DOUT → [M:0]
DOUT_F → {DIN, [(N·M)-1:M]}

Shift left one word -Shift in DIN ↳1111

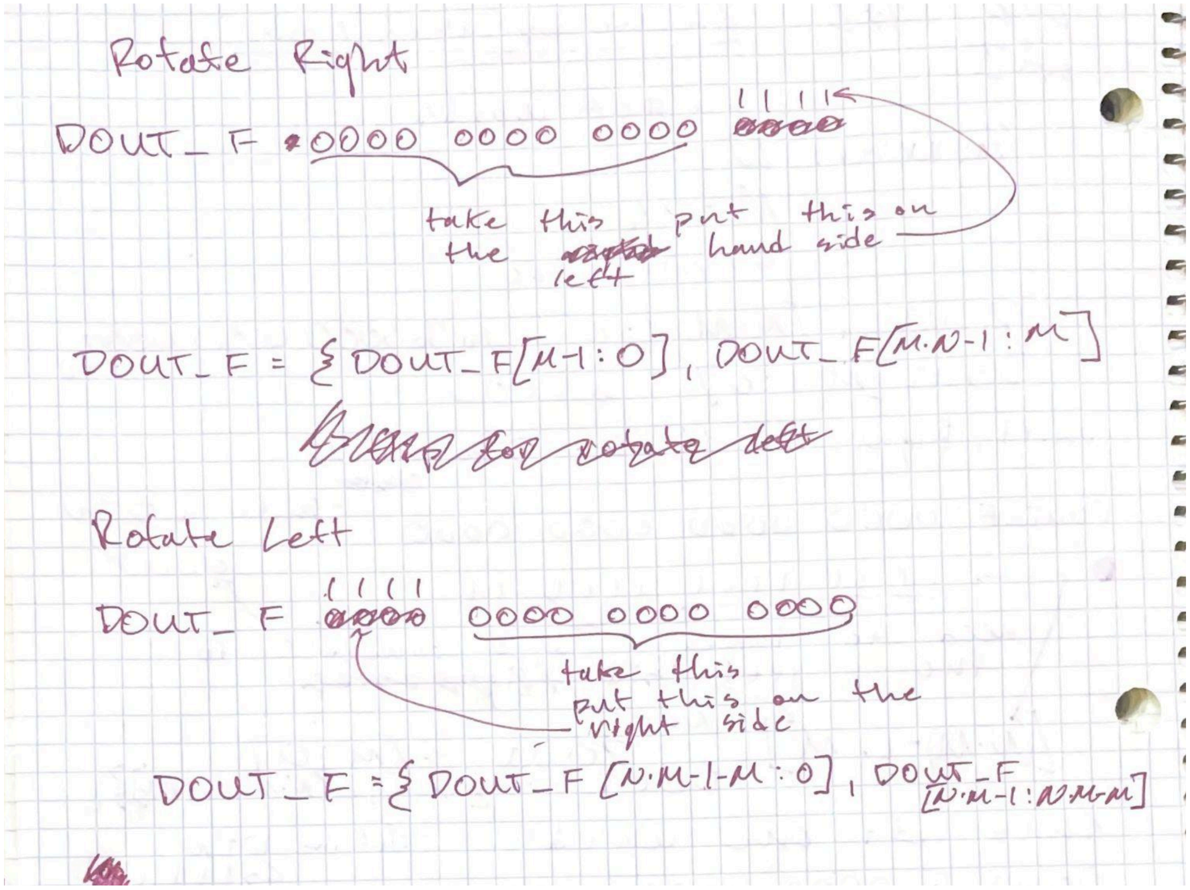
DOUT_F 0000 0000 0000 0000

take this and put this on the right hand side

[(N·M)-M:0]

Now ↴
DOUT → [(N·M-1):(N·M)-M]
↳ DOUT_F → {[(N·M)-M:0], DIN}

c)



Rotate Right

```
                                    1111
DOUT_F ~~0~~ 0000 0000 0000 ~~0000~~
```

take this put this on the ~~[illegible]~~ left hand side

DOUT_F = { DOUT_F[M-1:0], DOUT_F[M·N-1:M]

~~Same for rotate left~~

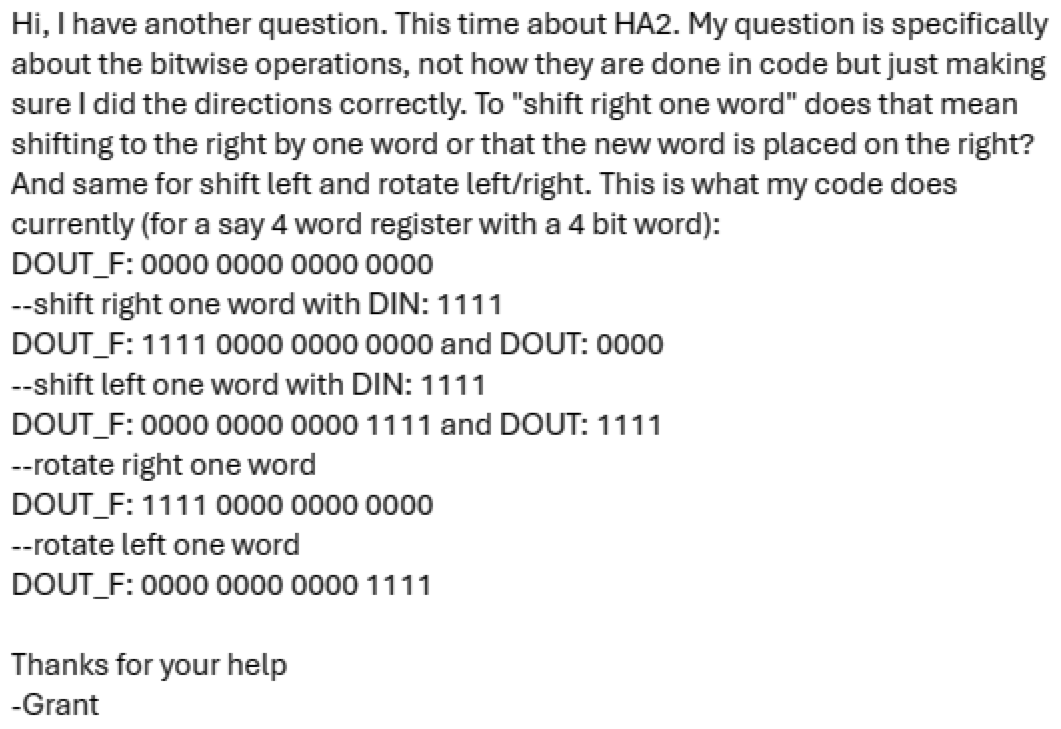
Rotate Left

```
        1111
DOUT_F ~~0000~~ 0000 0000 0000
```

take this put this on the right side

DOUT_F = { DOUT_F [N·M-1-M:0], DOUT_F [N·M-1:N·M-M]

b)

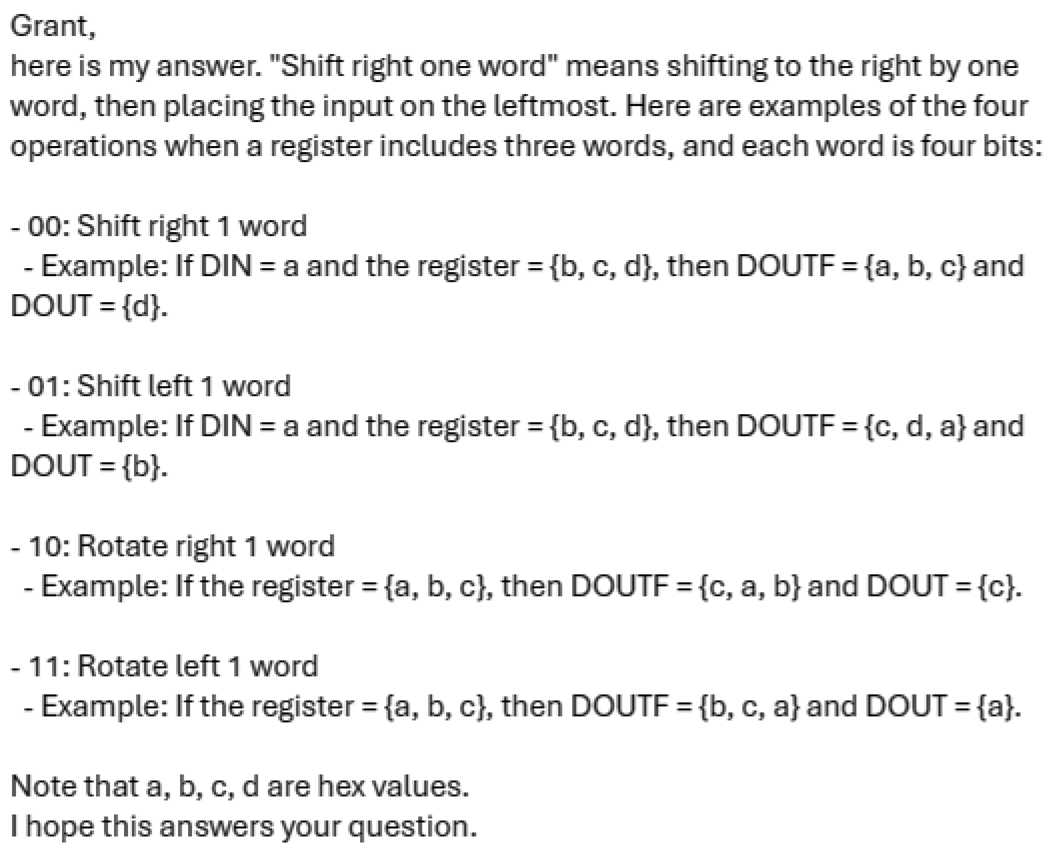


Hi, I have another question. This time about HA2. My question is specifically about the bitwise operations, not how they are done in code but just making sure I did the directions correctly. To "shift right one word" does that mean shifting to the right by one word or that the new word is placed on the right? And same for shift left and rotate left/right. This is what my code does currently (for a say 4 word register with a 4 bit word):

```
DOUT_F: 0000 0000 0000 0000
--shift right one word with DIN: 1111
DOUT_F: 1111 0000 0000 0000 and DOUT: 0000
--shift left one word with DIN: 1111
DOUT_F: 0000 0000 0000 1111 and DOUT: 1111
--rotate right one word
DOUT_F: 1111 0000 0000 0000
--rotate left one word
DOUT_F: 0000 0000 0000 1111
```

Thanks for your help
-Grant

c)



Grant,
here is my answer. "Shift right one word" means shifting to the right by one word, then placing the input on the leftmost. Here are examples of the four operations when a register includes three words, and each word is four bits:

- 00: Shift right 1 word
  - Example: If DIN = a and the register = {b, c, d}, then DOUTF = {a, b, c} and DOUT = {d}.

- 01: Shift left 1 word
  - Example: If DIN = a and the register = {b, c, d}, then DOUTF = {c, d, a} and DOUT = {b}.

- 10: Rotate right 1 word
  - Example: If the register = {a, b, c}, then DOUTF = {c, a, b} and DOUT = {c}.

- 11: Rotate left 1 word
  - Example: If the register = {a, b, c}, then DOUTF = {b, c, a} and DOUT = {a}.

Note that a, b, c, d are hex values.
I hope this answers your question.

d)