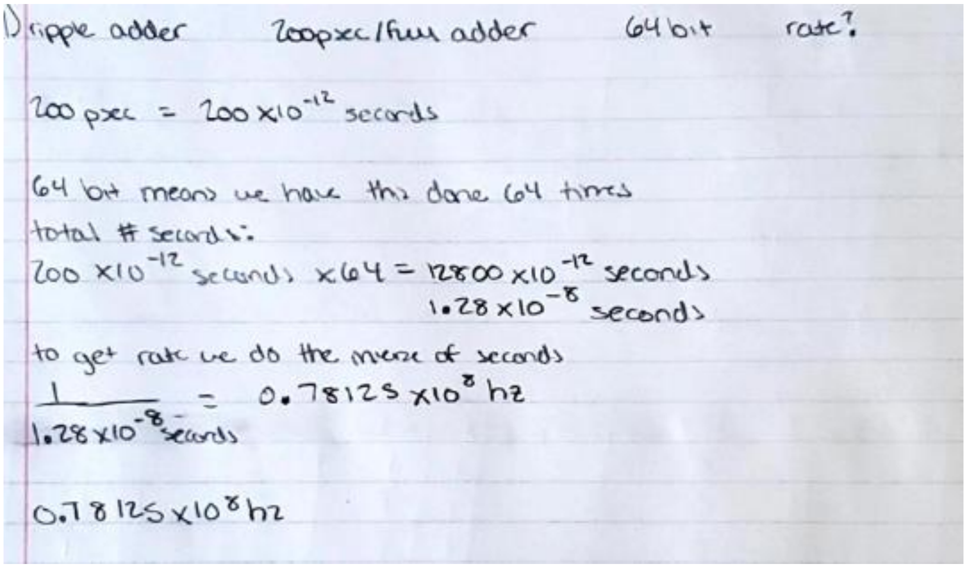
Homework 4

1) The answer I got was 0.78125*10^8 hz

To get this, I took the number of seconds it took for one full adder and multiplied it by 64. Since this is a ripple adder, the number of additions it performs is the same as the bit-size of the machine. In this case it is 64-bit so 64 additions. Once I had the full number of seconds the whole adder operation would take, I found the inverse of that number to get the clock rate.



1) ripple adder 200psec/full adder 64 bit rate?

200 psec = $200 \times 10^{-12}$ seconds

64 bit means we have this done 64 times

total # seconds:

$200 \times 10^{-12}$ seconds $\times 64 = 12800 \times 10^{-12}$ seconds

$1.28 \times 10^{-8}$ seconds

to get rate we do the inverse of seconds

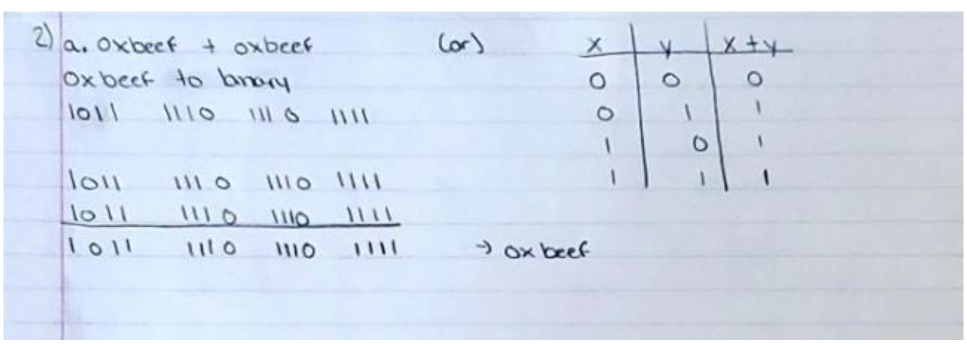
$\frac{1}{1.28 \times 10^{-8} \text{ seconds}} = 0.78125 \times 10^{8}$ hz

$0.78125 \times 10^{8}$ hz

2)

a. Answer: 0xbeef

For this problem, we just converted the 0xbeef to the binary form. Then we or them together. In or operations, anything with a one is it is only a zero when both the numbers are zero.



2) a. 0xbeef + 0xbeef (or)

0xbeef to binary

1011 1110 1110 1111

1011 1110 1110 1111

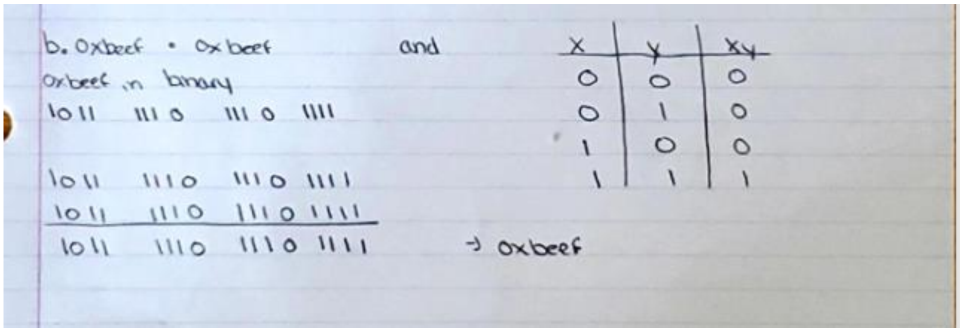
1011 1110 1110 1111

1011 1110 1110 1111 → 0xbeef

| x | y | x+y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

b. Answer: 0xbeef

For b we started with putting 0xbeef into binary. This is an and operation, so we only get a 1 when there are two ones and together. This gives us 0xbeef at the end.



b. 0xbeef • 0x beef and

0xbeef in binary

1011 1110 1110 1111

1011 1110 1110 1111

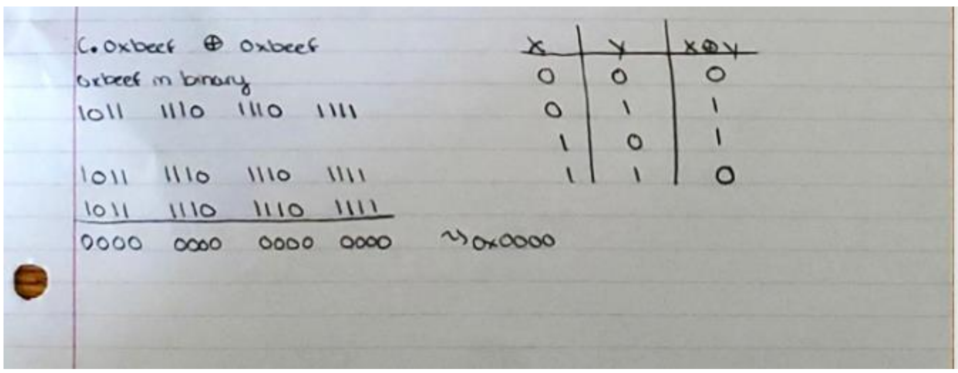
1011 1110 1110 1111

1011 1110 1110 1111 → 0xbeef

| x | y | xy |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

c. Answer: 0x0000

For part c, I started by putting 0x beef into binary. All our ones lined up in this case, so everything became a zero.



c. 0xbeef ⊕ 0xbeef

0xbeef in binary

1011 1110 1110 1111

1011 1110 1110 1111

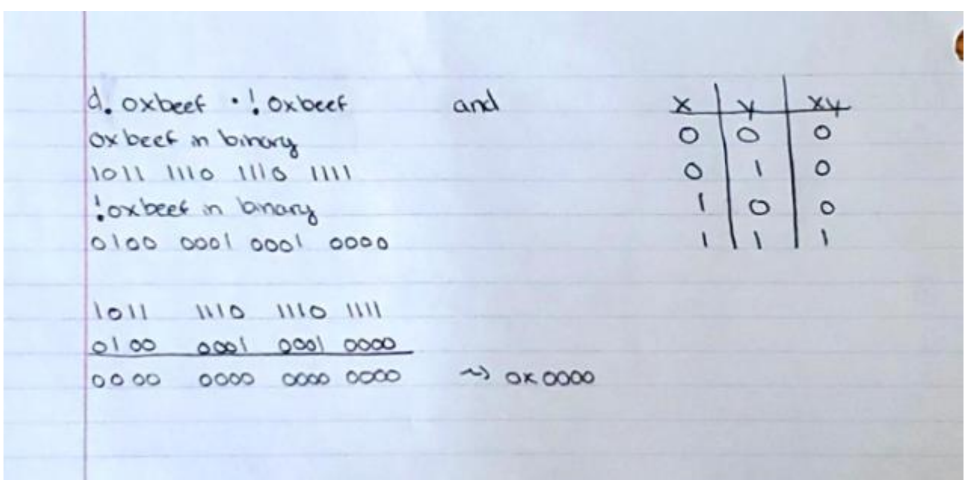
1011 1110 1110 1111

0000 0000 0000 0000 → 0x0000

| x | y | x⊕y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

d. Answer: 0x000

For part d, start by putting 0xbeef into binary. Once it was in binary, I did the not function on it which switches zeros to one and ones to zeros. Then these two binary numbers were put through the and operation. Since no ones lines up every bit became zero.



d. oxbeef • !oxbeef and

ox beef in binary

1011 1110 1110 1111

!oxbeef in binary

0100 0001 0001 0000

1011 1110 1110 1111

0100 0001 0001 0000

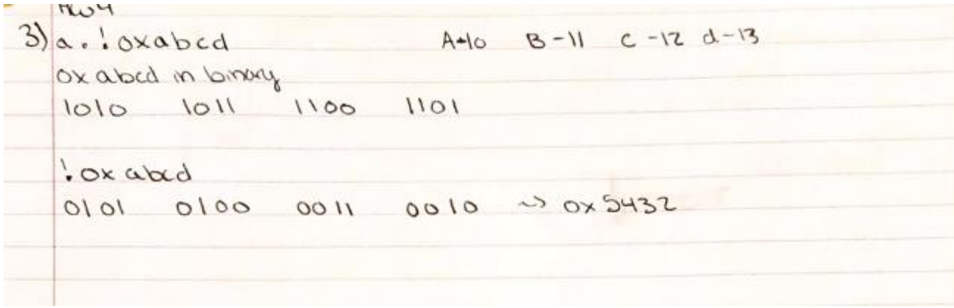
0000 0000 0000 0000 → ox 0000

| x | y | xy |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

3)

A. Answer: 0x5432

I started this problem by converting 0xabcd to binary. Then I had to run the not operator on the binary string. To this you switch zeros to ones and ones to zeros.


HW4

3) a. !oxabcd    A-10  B-11  C-12  d-13

ox abcd in binary

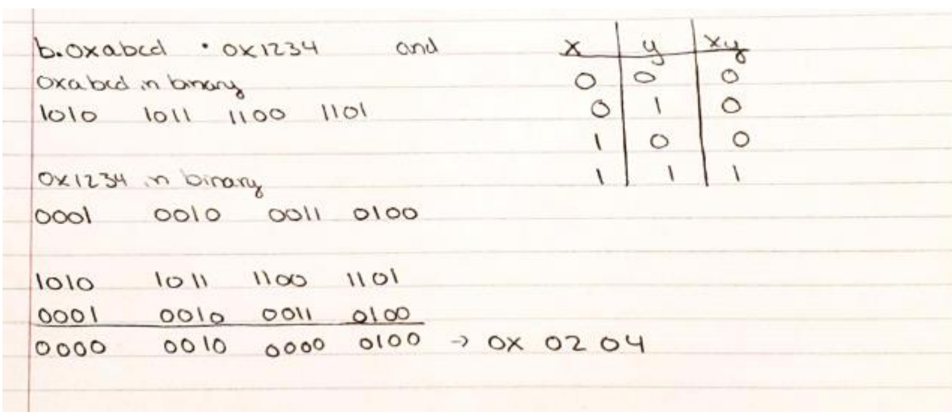
1010 1011 1100 1101

!ox abcd

0101 0100 0011 0010 → ox 5432

B. Answer: 0x0204

The first step for this problem was to put 0xabcd and 0x1234 into binary. When the operation is performed on these two binary strings, the only place we will get a zero is where both strings have a zero.


b. oxabcd · ox1234   and

| x | y | xy |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

oxabcd in binary

1010 1011 1100 1101

ox1234 in binary

0001 0010 0011 0100

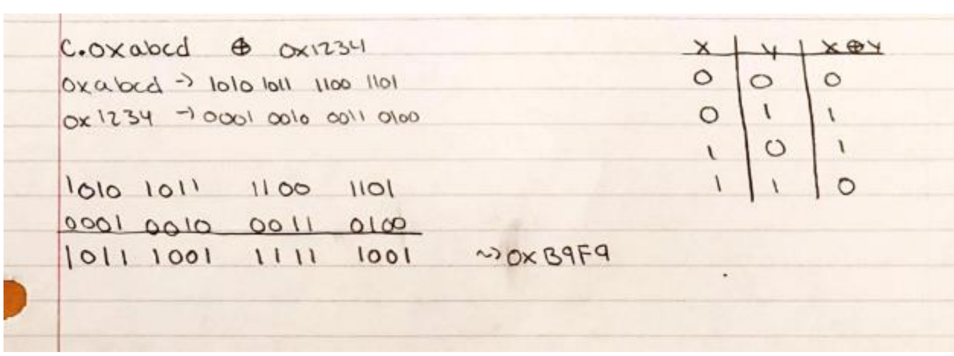
1010 1011 1100 1101

0001 0010 0011 0100

0000 0010 0000 0100 → 0x 02 04

C. Answer: 0xb9f9

The first thing I did for this problem was put 0xabcd and 0x1234 into binary. The operation we are performing on these binary strings is xor which gives us a one where only one string has a one and a zero everywhere else.


c. oxabcd ⊕ ox1234

ox abcd → 1010 1011 1100 1101

ox 1234 → 0001 0010 0011 0100

| x | y | x⊕y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

1010 1011 1100 1101

0001 0010 0011 0100

1011 1001 1111 1001 → 0x B9F9

D. Answer: 0xbbfd

The first step here was to put 0xabcd and 0x1234 to binary. The operation that was performed here was or so the only place we get a zero is when both strings have a zero at that bit.

d. 0xabcd + 0x1234 or

0xabcd ⇝ 1010 1011 1100 1101

0x1234 ⇝ 0001 0010 0011 0100

| x | y | x or y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1010 1011 1100 1101

0001 0010 0011 0100

1011 1011 1111 1101 ⇝ 0xBBFD

4)Answer: 0xf60e

I got this answer by first putting 0x0234 and 0x5de7 into binary. 0x5def then needed to have not applied to it by switching the ones for zeros and zeros for ones. These two strings then had or applied to them. The next step was to put 0xabcd in binary and apply not to the binary string. The result of the first operation was then xor'ed with the binary string we got from !0xabcd.

4) a. (0x0234 + !0x5de7) ⊕ !0xabcd

0x0234 to binary

0000 0010 0011 0100

0x5de7 to binary A-10 B-11 C-12 D-13 E-14 F 15

0101 1101 1110 0111

!0x5de7

1010 0010 0001 1000

0x0234 + !0x5de7 or

0000 0010 0011 0100

1010 0010 0001 1000

1010 0010 0011 1100

| x | y | x+y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

0xabcd to binary

1010 1011 1100 1101

!0xabcd

0101 0100 0011 0010

(0x0234+!0x5de7) ⊕ !0xabcd

| x | y | x ⊕ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

1010 0010 0011 1100

0101 0100 0011 0010

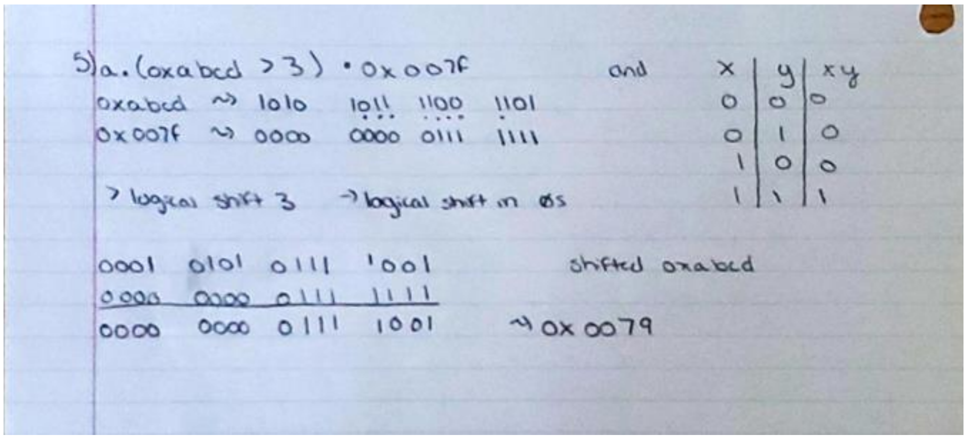
1111 0110 0000 1110

0xf60e

5)

a. Answer: 0x0079

I got this answer by first changing 0xabcd into binary. This binary string was then logically shifted by 3. Next, I changed 0x007f into binary. Finally, I took the shifted binary string and 0x007f and applied the and to them.



5)a. (0xabcd > 3) • 0x007f

0xabcd ~> 1010 1011 1100 1101

0x007f ~> 0000 0000 0111 1111

> logical shift 3 → logical shift in 0s

0001 0101 0111 1001 shifted 0xabcd

0000 0000 0111 1111

0000 0000 0111 1001 ~> 0x 0079

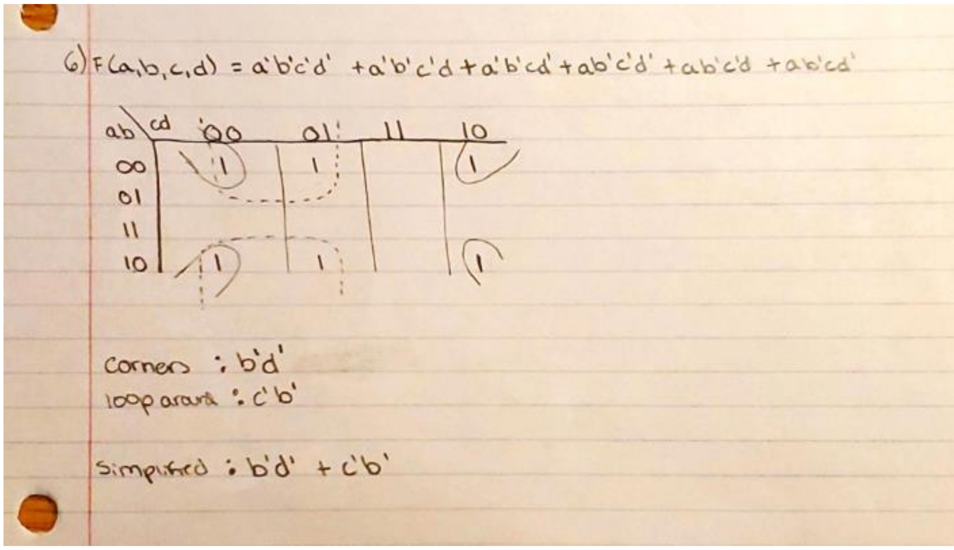
and

| x | y | xy |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

b) A right shift in general can be used for division. Shifting right by three is a division by 8. The number we divide by is 2^(number of shifts). Thus, moving to the right three times is a division by 8. This pattern holds for any shift to the right. A logical shift like this ignores the most significant bit which might change the sign of the resulting number, so this should only be done logically when you know the resulting number is positive as the shifted in zeros will keep the result positive. However, if you need to keep the most significant bit you should do an arithmetic shift rather than a logical one as it does not get rid of the most significant bit.

6) simplified function: b’d’+c’b’

I got this by looking at the function string and adding a one where the values matched up. If there is a not on the variable, then that variable is 0 and if there is not a not then it should be represented as a 1. I put one in the boxes that represented the minterms of the function we were given. Then I made the largest groups I could to simplify.




7) The simplified function I got was w’xyz’

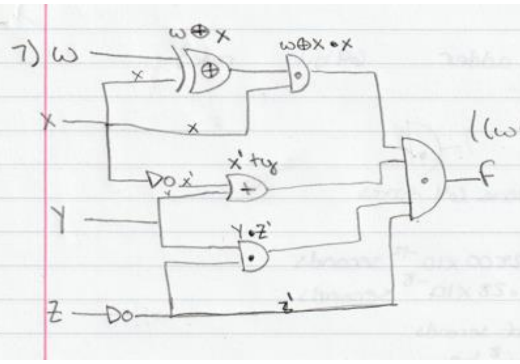
The first image here was my attempt at finding the function of the circuit. The result I got here was ((w xor x)*x)*(x’+y)*(y*z’)*(z’)

The top of the next image is my truth table. The first four columns list out the 16 combinations of zeros and ones for four variable inputs. The next column shows the values for w x or x (Note: I left areas that were zero blank later in the table to make it a bit easier for me and others to read). The next column over is ((w xor x)*x). Then I did a x’ column followed by (x’+y). The column after that shows the values for ((w xor x)*x)*(x’+y). Then I have a z’ column followed by (y*z’). Second to last is a column for ((w xor x)*x)*(x’+y)*(y*z’). Finally, I have the column that has the full function I found ((w xor x)*x)*(x’+y)*(y*z’)*(z’).

Next in this image is the kmap (karnaugh map). There was only one row in the truth table where I got a one for our function. This row was w’xyz’. The area that reflects this has a one in the kmap. This is 0110 the way I drew my kmap.

From the kmap, I can simplify the function to w’xyz’. This was the one box with a one in it from the truth table.

Finally, we have my attempt at a simplified circuit. W and Z get a not symbol on their lines and all four of the inputs are and together with an and gate.

7) 


$((w \oplus x) \cdot x) \cdot (x' + y) \cdot (y \cdot z') \cdot (z')$

| | w | x | y | z | $w \oplus x$ | $(w \oplus x) \cdot x$ | $x'$ | $x' + y$ | $((w \oplus x) \cdot x) \cdot (x'+y)$ | $z'$ | $y \cdot z'$ | $((w \oplus x) \cdot x) \cdot (x'+y) \cdot (y \cdot z')$ | $((w \oplus x) \cdot x) \cdot (x'+y) \cdot (y \cdot z') \cdot z'$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 0 | 0 | 0 | 0 | 0 | | 1 | 1 | | 1 | | | |
| 2. | 0 | 0 | 0 | 1 | 0 | | 1 | 1 | | 0 | | | |
| 3. | 0 | 0 | 1 | 0 | 0 | | 1 | 1 | | 1 | 1 | | |
| 4. | 0 | 0 | 1 | 1 | 0 | | 1 | 1 | | 0 | | | |
| 5. | 0 | 1 | 0 | 0 | 1 | 1 | | | | 1 | | | |
| 6. | 0 | 1 | 0 | 1 | 1 | 1 | | | | 0 | | | |
| 7. | 0 | 1 | 1 | 0 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | 1 |
| 8. | 0 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 0 | | | |
| 9. | 1 | 0 | 0 | 0 | 1 | | 1 | 1 | | 1 | | | |
| 10. | 1 | 0 | 0 | 1 | 1 | | 1 | 1 | | 0 | | | |
| 11. | 1 | 0 | 1 | 0 | 1 | | 1 | 1 | | 1 | 1 | | |
| 12 | 1 | 0 | 1 | 1 | 1 | | 1 | 1 | | 0 | | | |
| 13. | 1 | 1 | 0 | 0 | 0 | | | | | 1 | | | |
| 14. | 1 | 1 | 0 | 1 | 0 | | | | | 0 | | | |
| 15. | 1 | 1 | 1 | 0 | 0 | | | 1 | | 1 | 1 | | |
| 16 | 1 | 1 | 1 | 1 | 0 | | | 1 | | 0 | | | |

| wx \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | | 1 |
| 11 | | | | |
| 10 | | | | |

$F = w'xyz'$