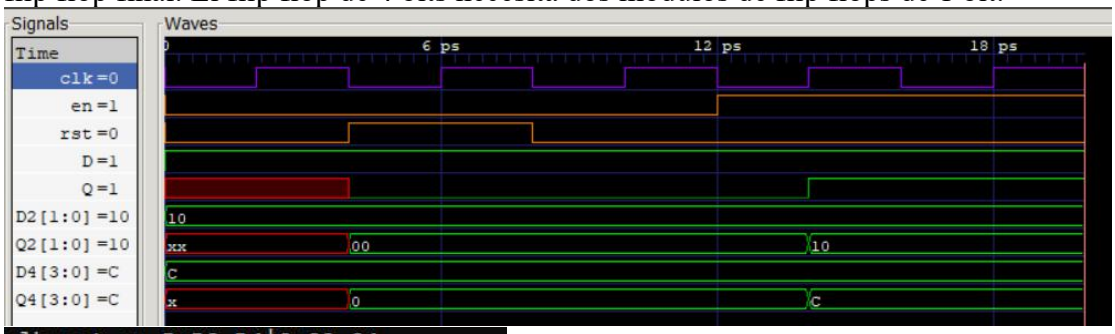
Laboratorio #9

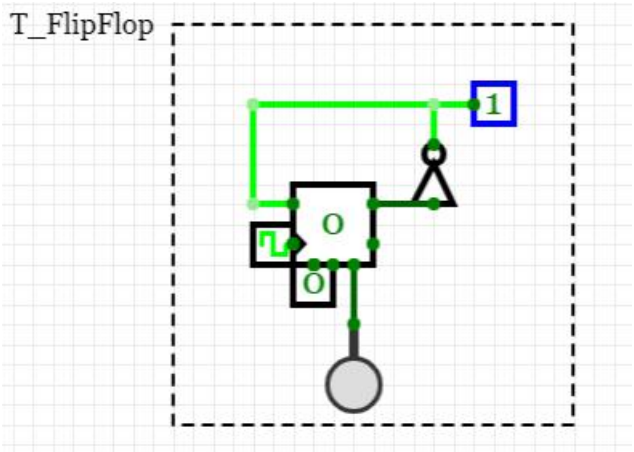
Link Repositorio: https://github.com/Cue19275/DIGITAL1UVG

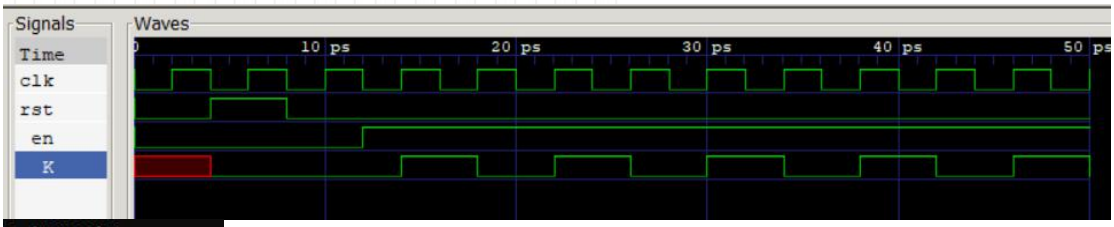
Ejercicio #1:
Se usan always y condicionales para hacer el primer flipflop de 1 bit. En base a este se construyen los módulos del flip flop de 2 y con este se construye el de 4 bits. Cada módulo necesita el mismo número de flipflops de 1 bit que el número de bits de dicho flip flop final. El flip flop de 4 bits necesita dos módulos de flip flops de 1 bit.




```
clk rst en D D2 D4|Q Q2 Q4
------------------|----------------
VCD info: dumpfile EJ1_tb.vcd opene
0 0 0 1 10   1100|x xx  xxxx
1 0 0 1 10   1100|x xx  xxxx
0 1 0 1 10   1100|0 00  0000
1 1 0 1 10   1100|0 00  0000
0 0 0 1 10   1100|0 00  0000
1 0 0 1 10   1100|0 00  0000
0 0 1 1 10   1100|0 00  0000
1 0 1 1 10   1100|1 10  1100
0 0 1 1 10   1100|1 10  1100
1 0 1 1 10   1100|1 10  1100
0 0 1 1 10   1100|1 10  1100
gtkwave Ej1_tb.vcd Ej1_tb.gtkw
```

Ejercicio #2:
Este flip flop (tipo T) es una variante particular de un flipflopD de 1 bit. Para generar el código del flip flop tipo T lo primero que se debe de hacer es volver a crear un flipflop de 1 bit como en el ejemplo anterior. Luego se crea un modulo para el flipflop tipo T, en el cual las unicas entradas son el clock, el reset y el enable, con una salida única. Se cablea de forma que la entrada del flipflop tipo D siempre sea el negado de la salida de este. De esta forma despues del reset que pone la salida en 0, esta irá oscilando continuamente gracias a la señal de clock.

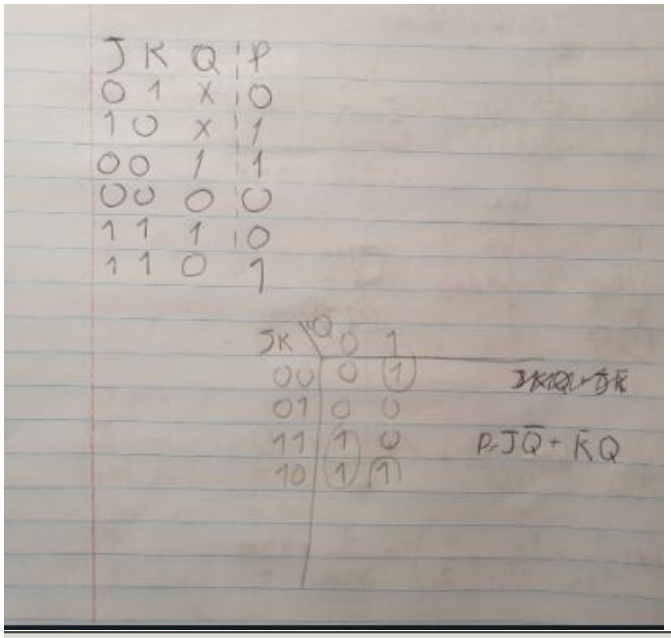


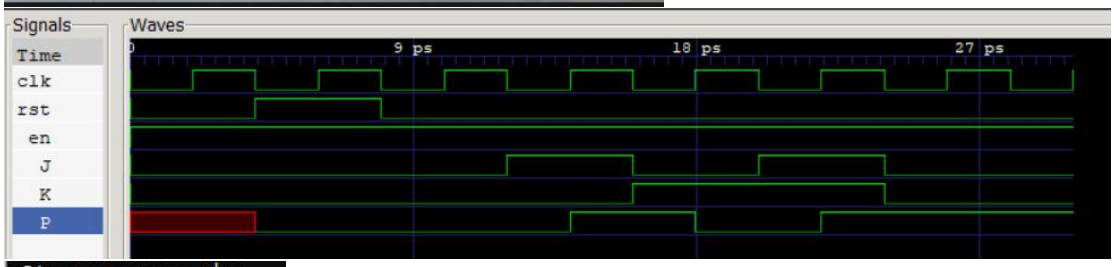



```
Lab9_Ej2
clk rst en|K
----------|-
VCD info: dumpf
0  0  0 |x
1  0  0 |x
0  1  0 |0
1  1  0 |0
0  0  0 |0
1  0  0 |0
0  0  1 |0
1  0  1 |1
0  0  1 |1
1  0  1 |0
0  0  1 |0
1  0  1 |1
0  0  1 |1
1  0  1 |0
0  0  1 |0
1  0  1 |1
0  0  1 |1
1  0  1 |0
0  0  1 |0
1  0  1 |1
0  0  1 |1
1  0  1 |0
0  0  1 |0
1  0  1 |1
0  0  1 |1
1  0  1 |0
gtkwave Ej2_tb.
```

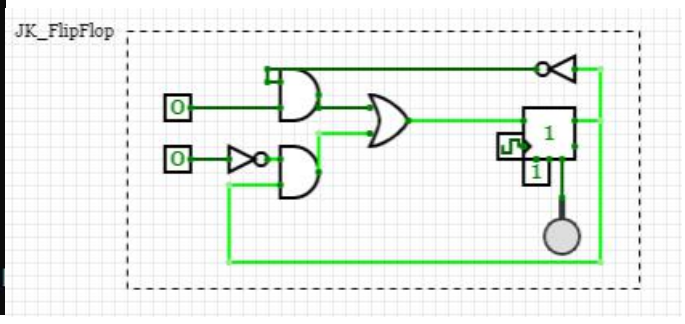
Ejercicio #3:

Para construir el flipflop JK se vuelve a necesitar de un flipflop tipo D de un bit. La entrada de dicho flipflop, es el resultado de una nube combinacional cuyas entradas son: Dos inputs (J y K), y la salida del mismo flipflop. Esta lógica combinacional estádescrita en el Kmap de abajo. Una vez con esto resuelto, se crea el módulo en verliog indicando que la entrada al módulo de un flip flop se dará a raíz de dicha lógica combinacional.





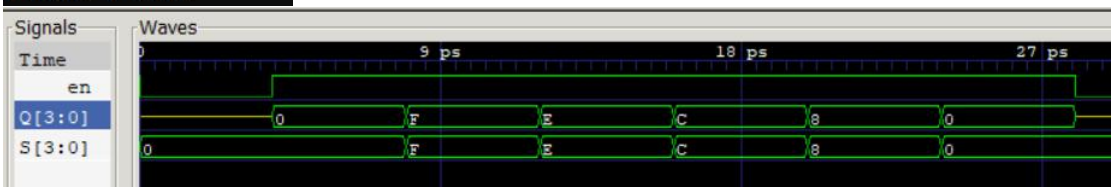

```
clk rst en J K|P
--------------|-
VCD info: dumpfile
0  0  1  0 0 |x
1  0  1  0 0 |x
0  1  1  0 0 |0
1  1  1  0 0 |0
0  0  1  0 0 |0
1  0  1  0 0 |0
0  0  1  1 0 |0
1  0  1  1 0 |1
0  0  1  0 1 |1
1  0  1  0 1 |0
0  0  1  1 1 |0
1  0  1  1 1 |1
0  0  1  0 0 |1
1  0  1  0 0 |1
0  0  1  0 0 |1
1  0  1  0 0 |1
gtkwave Ej3_tb.vcd
```



Ejercicio #4:

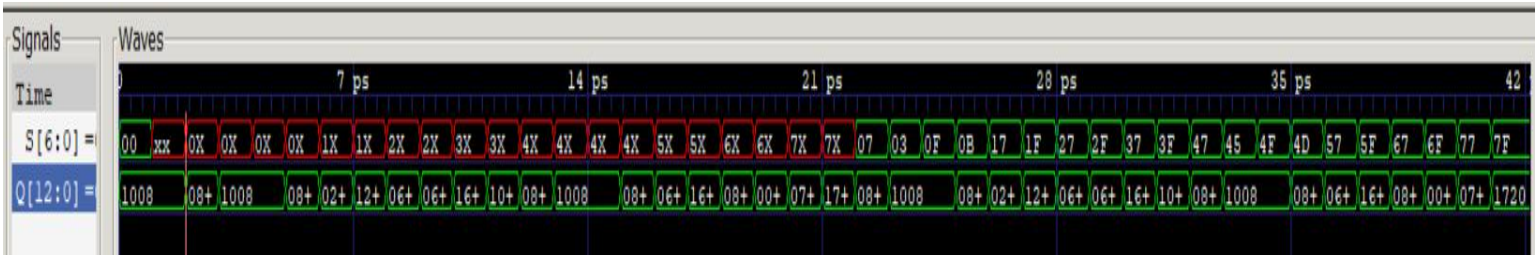
Para crear el buffer tri estado, se crea un módulo que utiliza un assign de salida que es habilitado gracias al operador ternario. Gracias a este, se puede conseguir la alta impedancia cuando el enable esta en LOW.

```
en S    | Q
--------|------
VCD info: dumpfile E
0 0000  | zzzz
1 0000  | 0000
1 1111  | 1111
1 1110  | 1110
1 1100  | 1100
1 1000  | 1000
1 0000  | 0000
0 0000  | zzzz
```



Ejercicio #5:

Para crear una ROM con cases, es bastante facil. Solamente se tiene que establecer un input, de 7 bits en este caso, el cual servira como el selector para el case. Ojo, que dada a la naturaleza de la tabla que se quiere implementar, se tiene que declarar el case como un casex. Luego de esto se indica dentro del case las posiblidades descritas en la tabla a implementar.



```
Lab9_Ej5
S       | Q
--------|--
VCD info: dumpfile Ej5_tb.v
0000000 | 1000000001000
xxxxxx0 | 1000000001000
00001x1 | 0100000001000
00000x1 | 1000000001000
00011x1 | 1000000001000
00010x1 | 0100000001000
0010xx1 | 0001001000010
0011xx1 | 1001001100000
0100xx1 | 0011010000010
0101xx1 | 0011010000100
0110xx1 | 1011010100000
0111xx1 | 1000000111000
1000x11 | 0100000001000
1000x01 | 1000000001000
1001x11 | 1000000001000
1001x01 | 0100000001000
1010xx1 | 0011011000010
1011xx1 | 1011011100000
1100xx1 | 0100000001000
1101xx1 | 0000000001001
1110xx1 | 0011100000010
1111xx1 | 1011100100000
0000111 | 0100000001000
0000011 | 1000000001000
0001111 | 1000000001000
0001011 | 0100000001000
0010111 | 0001001000010
0011111 | 1001001100000
0100111 | 0011010000010
0101111 | 0011010000100
0110111 | 1011010100000
0111111 | 1000000111000
1000111 | 0100000001000
1000101 | 1000000001000
1001111 | 1000000001000
1001101 | 0100000001000
1010111 | 0011011000010
1011111 | 1011011100000
1100111 | 0100000001000
1101111 | 0000000001001
1110111 | 0011100000010
1111111 | 1011100100000
```