

a) A continuación se muestra el efecto de hacer uno o dos llamados a *meter.log()*:

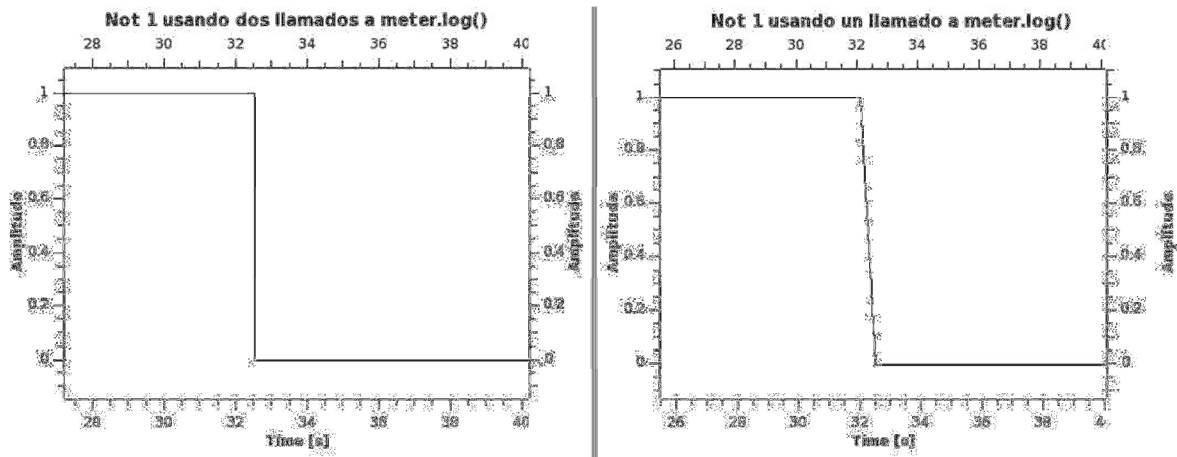


Ilustración 1: Diagrama temporal de Not#1 con uno y dos llamados a *meter.log()*

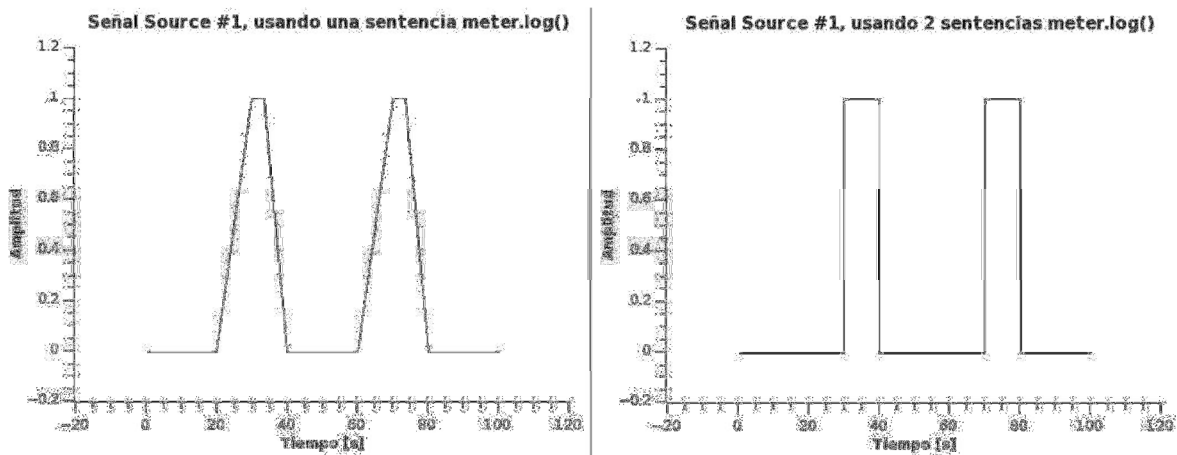


Ilustración 2: Diagrama temporal de Source#1 con uno y dos llamados a *meter.log()*

Como se puede apreciar, el problema yace en la discretización del tiempo que se ha hecho para simular la propagación de las señales. Se dispone sólo de un conjunto de puntos en donde se conoce los valores de las señales medidas con *meter*. Además, y para peor, estas señales no se encuentran distribuidas en intervalos regulares del eje temporal. Por esto, si hacemos solo un llamado, tendremos para las transiciones una interpolación lineal que se refleja como una pendiente que en realidad es inexistente. Si el programa solo registra que en un instante t_1 el valor de la señal es 0, y en el instante t_2 la señal pasa a tener valor 1, al graficar no tenemos como decirle al programa que la interpolación que debemos hacer es dejar la señal en 0 desde t_1 a t_2 . Por defecto se interpola en forma lineal, lo cual no representa la realidad. En un gráfico de

puntos esto no es un problema. Para corregir, se hacen dos llamados a *meter.log()*, de tal manera de interpolar manteniendo el valor desde t1 hasta t2, y hacer el cambio recién ahí, como se aprecia en las imágenes. De todas formas, es solo para solucionar problemas al momento de graficar las señales, pues no interfiere con la correcta simulación de la propagación de las señales.

b)

Al comenzar la simulación, todos los atributos se inicializan en 0 (todas las salidas están con valor 0). Si no le decimos a nuestro programa que al comenzar tiene que actualizarse, es decir, hacer un *update()* (para programar en la cola de prioridad y finalmente recalculer las salidas) ocurren dos cosas: primero, sólo empezaremos a registrar valores para las señales una vez ocurrido el primer cambio (vendrá dado por el primer cambio en alguna de las señales de entrada, source), y por lo tanto no tendremos gráfica para puntos anteriores (desde t=0), y, lo que es aún peor, el circuito puede no estar obedeciendo las leyes lógicas de cada componente inicialmente. Perfectamente podría existir una compuerta NOT con el mismo valor en su entrada y en su salida, y si no corregimos este desperfecto indicándole al programa que actualice el valor de todas sus salidas este error se arrastra generando una propagación errónea de los datos, como se puede apreciar en las imágenes.

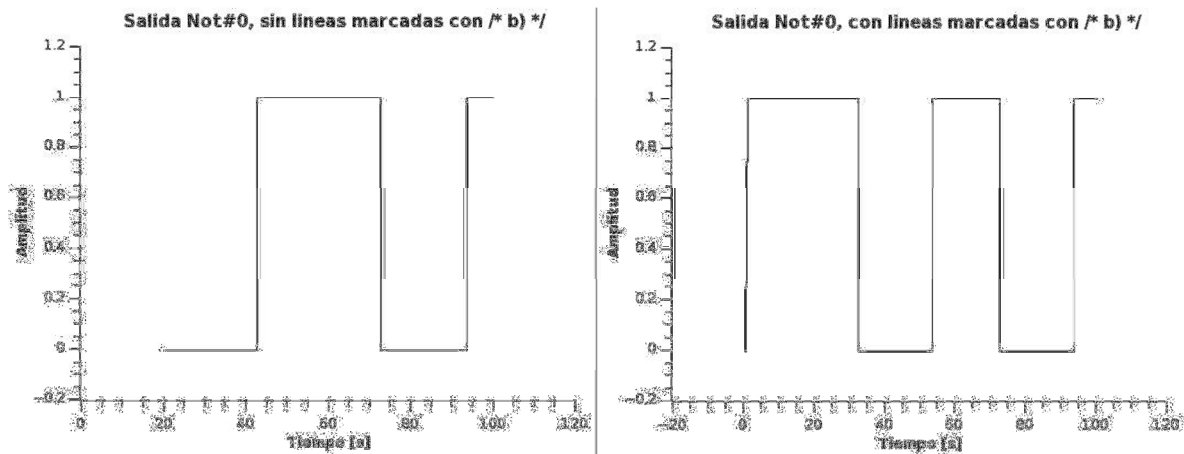


Ilustración 3: Efecto de hacer *update()* al momento de comenzar la simulación

b2)

Las sentencias están ahí para asegurarnos que no se registren valores (usando el método *meter.log()*) si no se producen variaciones en las señales de salida de alguna compuerta.

La primera sentencia: *if (output != newValue)* en el método *update()* está ahí por un tema de eficiencia en la simulación. El propósito de *update()* es actualizar los valores de salida de la compuerta y notificar al Wire de salida que corresponda. Éste a su vez notifica a las compuertas que tenga conectadas en su otro extremo para que programen un evento. Si no se ha modificado el valor de la salida, no es necesario hacer estas notificaciones, lo cual genera un ahorro de operaciones y hará nuestro programa más eficiente.

Por otro lado, la segunda sentencia: *if(output != computeOutput())* en el método *inputChanged()*, que a priori pareciera cumplir el mismo propósito que la anteriormente mencionada, en realidad cumple una función más importante. Debido a que dentro de *inputChanged()* se programan los eventos futuros, no queremos que se programen eventos que no generen cambios en las señales o salidas de nuestras componentes. Esto por un tema de eficiencia, sin dudas, pero más aún, cada vez que se programe un evento donde en realidad la salida no cambia, esto llamará de todas formas al método *update()*, lo cual volverá a programar un evento, produciendo un potencial loop infinito en nuestro código. Esto porque la simulación concluye una vez que todos los eventos de la cola de prioridad sean procesados y no queden más por extraer. Si nuestro circuito posee algún nivel de realimentación entre las compuertas, se programarán infinitos eventos que ya no aportan a la simulación, pues ya se ha llegado a un estado estacionario, y terminaremos con un archivo de salida cuyo tamaño crecerá hasta saturar la memoria del disco (problema grave).

Cabe destacar que lo importante, entonces, es que no se ejecute el código de *inputChanged()* si la variable de salida en realidad no ha cambiado su valor. Para evitar esta situación basta con que una de las dos sentencias analizadas esté presente. Eliminar la primera reduce la eficiencia de nuestro código, pero aún se realizará la simulación acorde a lo esperado. Eliminar la primera y la segunda genera un archivo de salida que no termina de crecer. Eliminar sólo la segunda no afecta en forma grave, pues el trabajo de filtrar cuando las salidas no cambian su valor la hace el método *update()* con anterioridad, pero no está demás tener las dos sentencias para asegurar robustez en la simulación. En efecto, si hacemos llamados a *inputChanged()* como se hace en el método *main()* para inicializar valores de las salidas de las compuertas, no estaremos pasando por el filtro que otorga el método *output()* y nuestra gráfica puede resultar incorrecta. En general, la propagación de las señales será similar, pero no igual, a que si incluimos las dos sentencias, como se puede apreciar en las siguientes imágenes:

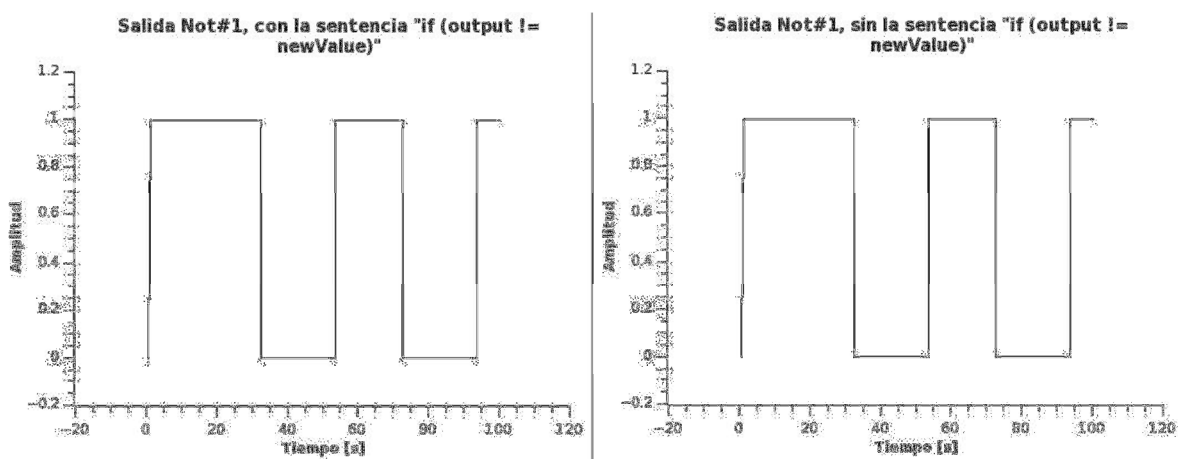


Ilustración 4: Simulación con y sin la sentencia *if(output != newValue)*

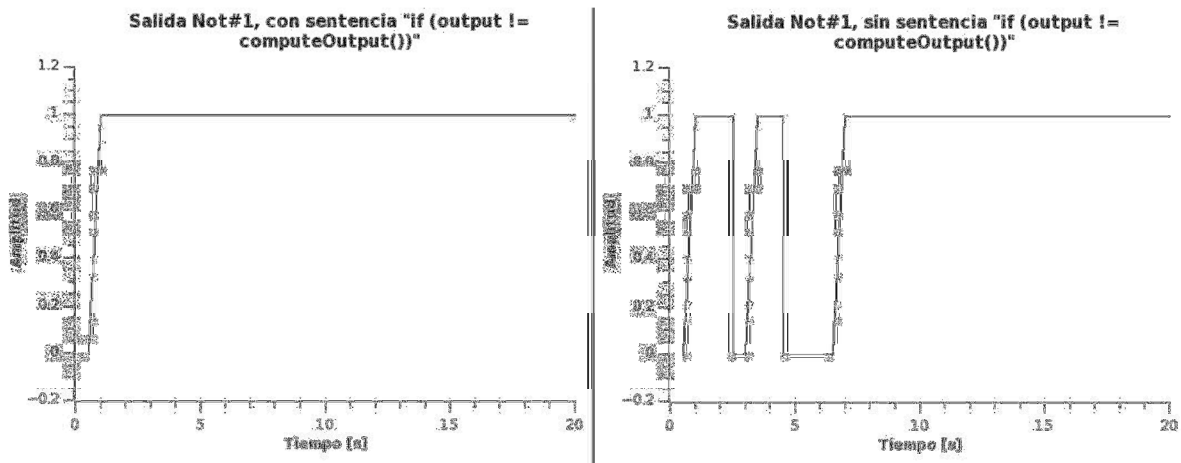


Ilustración 5: Simulación con y sin la sentencia `if(output != computeOutput())`

c)

Para simular el circuito basta hacer modificaciones dentro de la clase `digital circuit`, de la siguiente forma:

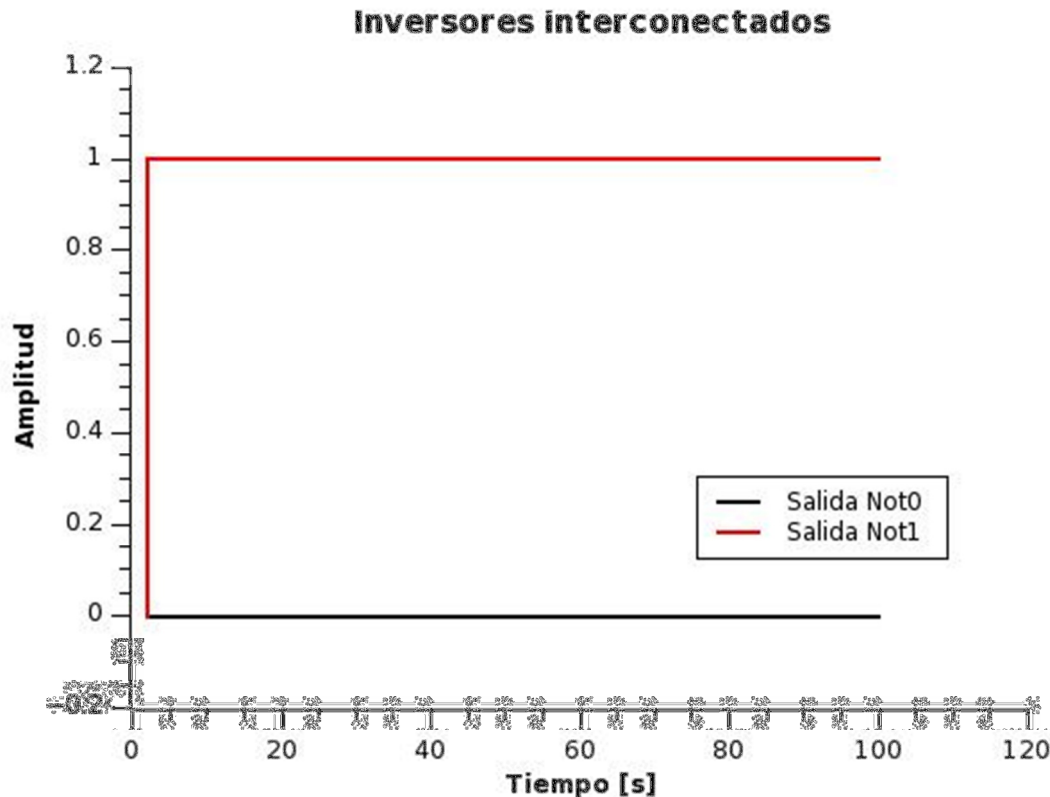
```
not0.connectOutput(w1);
not0.connectInput(w0);

not1.connectOutput(w0);
not1.connectInput(w1);

meter.addTestPoint(w0);
meter.addTestPoint(w1);

not0.inputChanged();
not1.inputChanged();
```

Una vez hechos todos los cambios pertinentes, se procede a simular y graficar las dos salidas de las compuertas NOT:



Es importante preguntarse cuál era el resultado esperado (¿qué debiese reflejar la simulación?) y si lo que en realidad se observa es consistente con la forma de simular programada.

Lo primero: sabemos que ambas compuertas inician su salida en 0 lógico. Es razonable asumir que, dado que ambas compuertas poseen retardos idénticos, ambas actualizarán sus estados simultáneamente, y al tener ambos un 0 lógico en su entrada, sus salidas pasarán a tener valor 1. Este proceso debería repetirse, generando una oscilación permanente, que debiese reflejarse en la gráfica de la simulación. Sin embargo este no es el comportamiento observado, pues se obtiene un estado estable.

En respuesta a la segunda interrogante, analicemos cómo estamos simulando realmente. Cada vez que se extrae un elemento de la cola de prioridad se hace un llamado a *update()* para que la componente que corresponda actualice su valor. Es en este punto donde se producen las diferencias entre lo pensado y lo observado, pues no estamos realmente actualizando todas las compuertas simultáneamente. En otras palabras, aunque existan dos o más eventos en la cola de prioridad programados con el mismo tiempo (eventos simultáneos temporalmente) las actualizaciones de los valores de salida se realizan secuencialmente, y cada valor modificado influye sobre el cálculo del siguiente. Es decir, si tenemos dos eventos simultáneos e1 y e2, primero calculamos la nueva salida para la componente asociada a e1, y luego al calcular el nuevo valor de salida para e2, el cambio en e1 es tomado en cuenta, siendo que no debiese.

Programando con un modelo para el avance del tiempo adaptado, similar al modelo descrito en el enunciado de la tarea para tiempo “continuo”, el resultado sería distinto, y podríamos observar la oscilación esperada para casos como el de los NOT interconectados.

Nota: se sugiere modificar levemente el código de la clase Simulator para evitar cometer esta misma falta a nivel de gráfico temporal. Con esta modificación, se realizan todos los llamados a `update()` de eventos simultáneos antes de graficar:

```
public void simulate(Meter meter) {
    ChangeEvent e;
    meter.writeHeader();
    while ( (e=pq.poll())!=null) {
        if (e.getTime() > currentTime) {
            meter.log(currentTime);

            currentTime = e.getTime();
            meter.log(currentTime);

            e.update();
        } else
            e.update();
    }
}
```

d)

La clase `And.java` se crea en forma análoga a la clase `Or.java`:

```
public class And extends Gate {
    public And (float delay, Simulator s) {
        super(delay, nextId++, s);
    }
    public void connectInputA(Wire w){
        inA = w;
        w.connectTo(this);
    }
    public void connectInputB(Wire w){
        inB = w;
        w.connectTo(this);
    }
    public boolean computeOutput(){
        boolean a = inA.getValue();
        boolean b = inB.getValue();
        return(a && b);
    }
}
```

```

    private Wire inA, inB;
    private static int nextId = 0;
}

```

e)

La clase Nand.java se crea utilizando la idea de la clase Wire.java para tener múltiples compuertas conectadas en su extremo (ArrayList). Éste se recorre con un “for mejorado” para obtener los valores de entrada a la compuerta y así calcular el valor lógico de salida.

```

import java.util.*;

public class Nand extends Gate {

    private ArrayList<Wire> pqu = new ArrayList<Wire>();
    public Nand (float delay, Simulator s) {
        super(delay, nextId++, s);
    }
    public void connectInput(Wire w){
        pqu.add(w);
        w.connectTo(this);
    }
    public boolean computeOutput(){
        Wire w;
        boolean aux = true;
        for(Wire s: pqu)
            aux= s.getValue() && aux;
        return (!aux);
    }
    private static int nextId = 0;
}

```

Usando compuertas Nand y otras, se simula el circuito correspondiente a un flip-flop D. El resultado de la simulación es el siguiente:

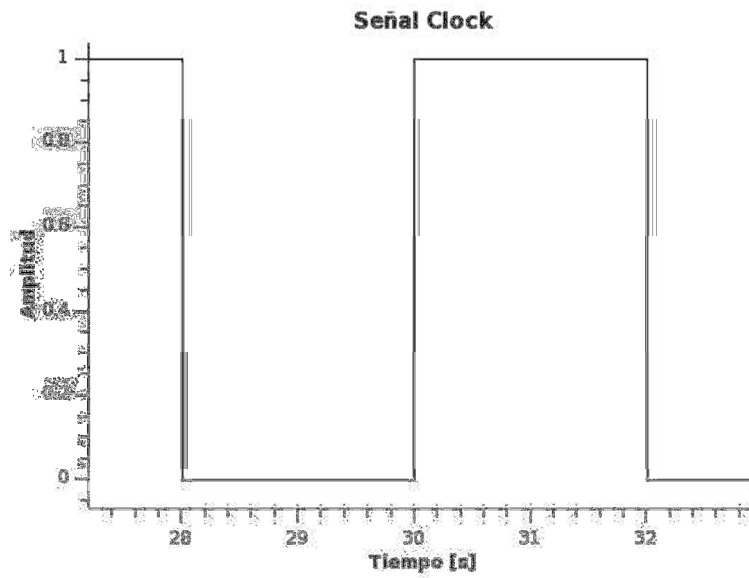


Ilustración 6: Diagrama temporal para la señal de reloj generada

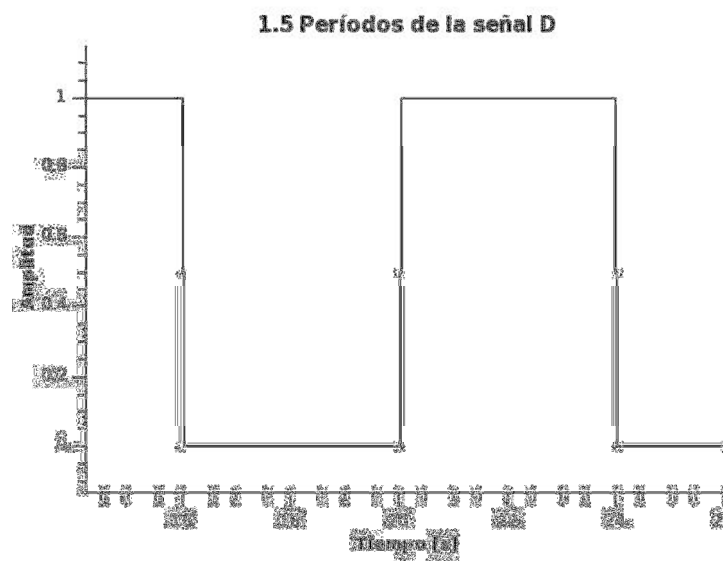


Ilustración 7: Diagrama temporal para la señal de entrada al filp-flop, D

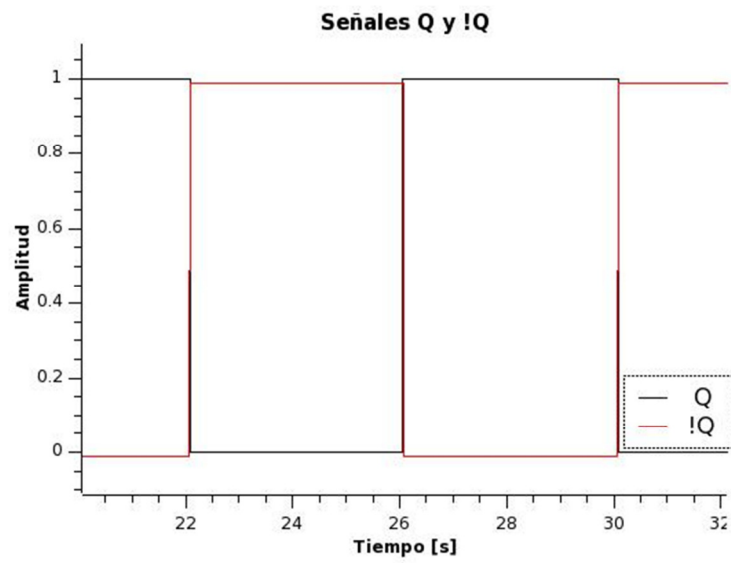


Ilustración 8: Diagrama temporal para las señales de salida Q y !Q

Es importante destacar que el período del reloj debe ser mayor al retardo de calcular las salidas, es decir, debemos considerar la ruta crítica de componentes dentro del flip flop para asegurarnos que alcance a establecerse la señal de salida Q antes entre cada ciclo. Esto establece una cota para la frecuencia máxima del reloj si deseamos observar correctamente el funcionamiento del flip-flop.