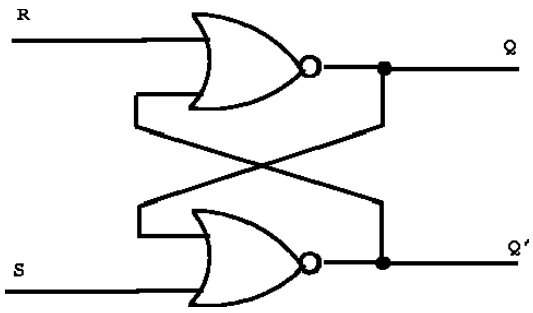
Tarea 1: Compuertas Digitales como Objetos de Software

Felipe Acevedo 2703001-7
Cristobal Barrientos 2821064-7
Thomas Dixon 2704559-6

Para la pregunta a) y b) se utiliza el siguiente circuito simulado:

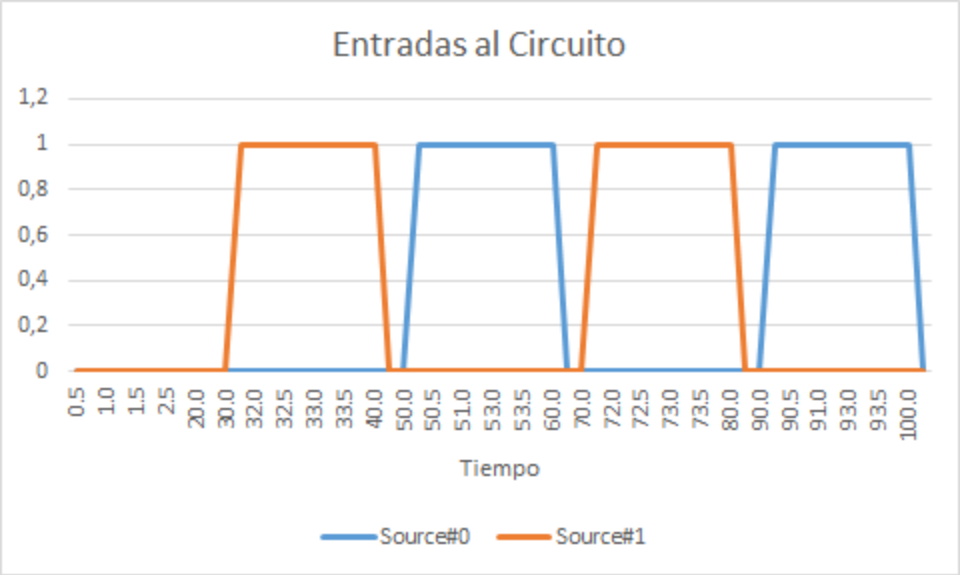


a) Para el código entregado analice el código del método simulate() de la clase Simulator. Haga los gráficos de las señales para ver el efecto de dejar sólo un llamado a meter.log() o dos (uno antes y otro después de e.update()). ¿Cuál es la ventaja de poner dos llamados en lugar de uno?

Método simulate() ubicado en Clase Simulator

```
 public void simulate(Meter meter) {
     ChangeEvent e; //Objeto cambio de evento
     meter.writeHeader();
/*Mientras existan eventos en la cola de prioridad*/
     while ( (e=pq.poll())!=null) {
       if (e.getTime() > currentTime) { // Si el tiempo guardado en el evento es mayor
          currentTime = e.getTime();
          /*Primer meter.log()*/
          meter.log(currentTime);  // Graba en archivo de salida el evento
          e.update();                        // Actualiza el valor de la salida de la compuerta
         /*Segundo meter.log()*/
          meter.log(currentTime); //Graba en archivo de salida el evento después de actualizar
       } else
          e.update();
   }
 }
```

Gráfico de las señales con dos llamado a meter.log()



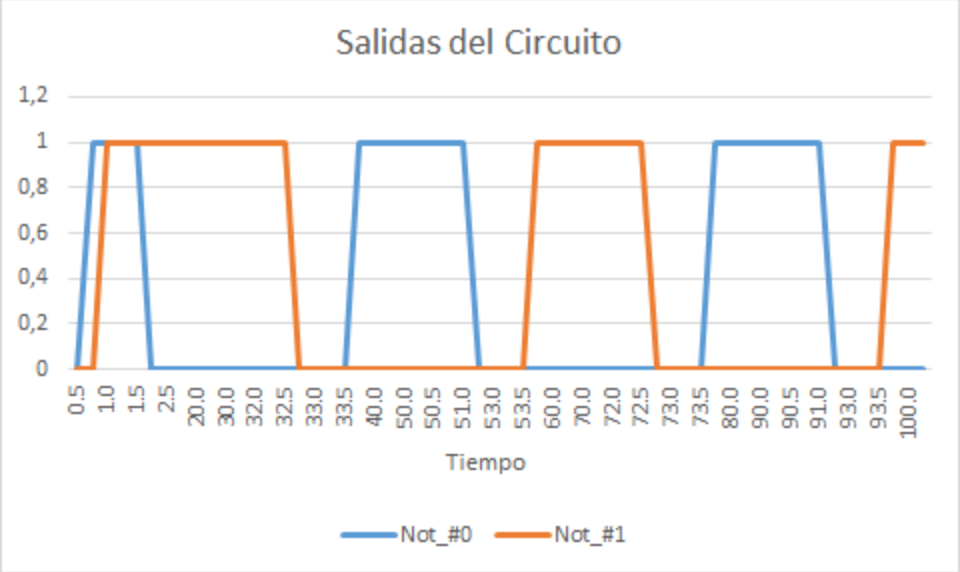
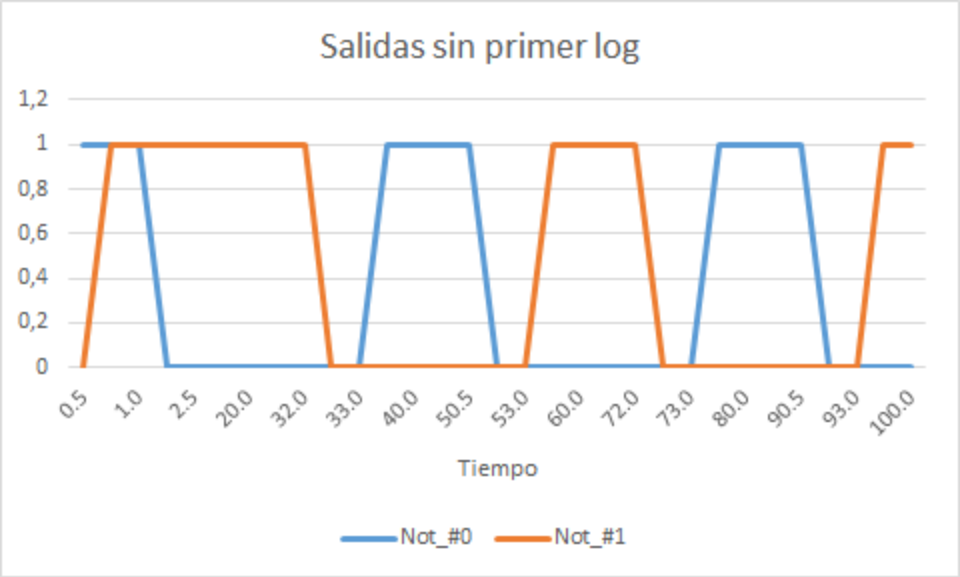


Gráfico de las señales con un llamado a meter.log()

Eliminando el primero

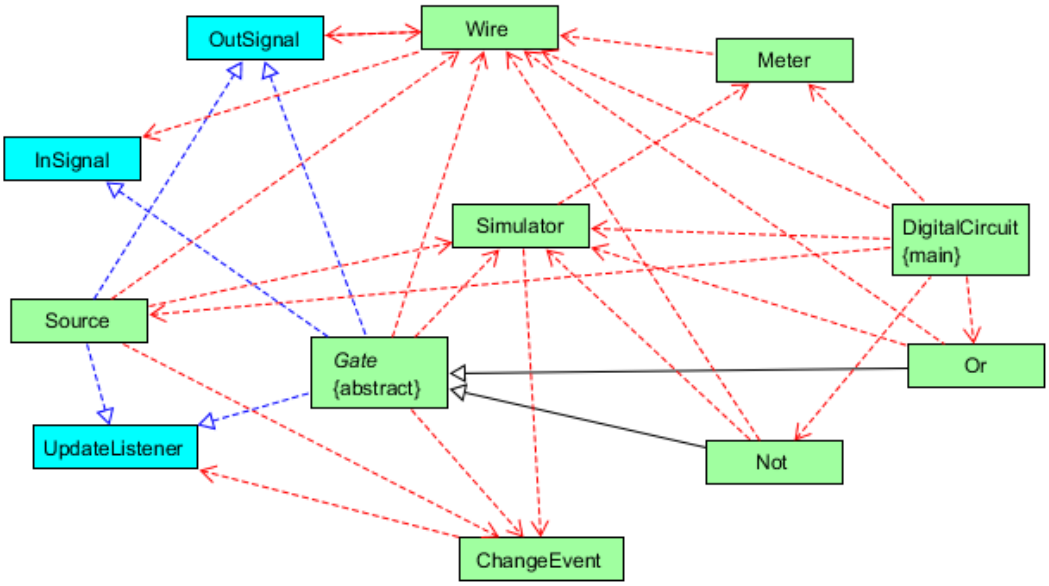


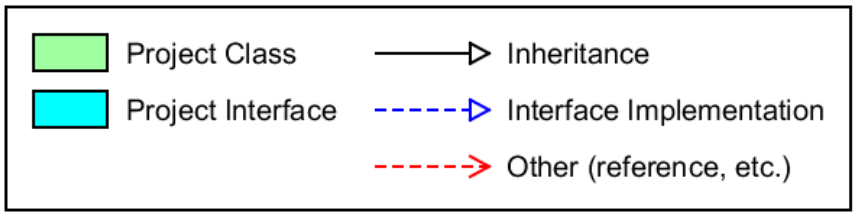
Eliminando el segundo

Ventaja de poner dos meter.log() en lugar de uno

Permite conocer el tiempo exacto en que cambia de valor la salida de una compuerta. Ya que registra el estado de la salida antes y después de realizar un cambio sin modificar el tiempo en que ocurre.
En el gráfico la hace parecer un cambio más instantáneo, al tener una pendiente más abrupta.

Diagrama de referencias de clases e interfaces.





b) Para el código entregado para DigitalCircuit.java, vea el efecto de incluir o no las líneas de código marcadas con / b) */. Describa la diferencia observada y por qué se produce.*

Código mencionado

Archivo: DigitalCircuit.java

/*Antes: inicialización de circuito ->creación de compuertas y cables

```
46  or0.inputChanged();  /* b)  */
47  or1.inputChanged();  /* b)  */
48  not0.inputChanged(); /* b)  */
49  not1.inputChanged(); /* b)  */
```

/* Despues: simulate(....

Al incluir no incluir las líneas de código antes mencionadas y graficar el archivo de salida (luego de correr el programa) se obtiene:

Primero analicemos el método inputChanged()

Esta implementado por la Clase Gate

```
public void inputChanged() {
/*Si la salida de la compuerta calculada es distinta a la actual*/
    if (output != computeOutput()) {
      /*Crea un evento de cambio para que ocurra en un tiempo “delay” más*/
      ChangeEvent e = new ChangeEvent(this, simulator.getTime()+delay);
      /*Agrega el evento a la cola de proridad*/
      simulator.add(e);
    }
 }
```

Lo que ocurre al quitar el codigo mostrado al inicio de la pregunta es no inicializar las compuertas producto de que se insertaron fuentes, esto ocasiona que las salidas de las compuertas no correspondan a lo esperado cuando se hace el primer set.

Por ejemplo la entrada del not_0 va a ser 0 y su salida también, lo que repercute en el cálculo de las salidas de las demás compuertas. Al inicializar las compuertas se insertan eventos a la cola de prioridades, lo que hace que luego de un tiempo de “delay” se calcule la salida para cada una de estas.

b.2) Para el circuito original entregado en DigitalCircuit.java, vea el efecto de las condiciones: "if (output != newValue)" en método update() e "if (output != computeOutput())" en método inputChanged() , ambas en la clase Gate. ¿Se pueden eliminar? ¿Qué se gana con tenerlas?

Efecto de condiciones

La condición “if (output != newValue)" en método update()“ permite primero evitar que se cambie el valor de salida de una compuerta y luego evitar que se notifique el cambio al cable de salida de dicha compuerta. Esto en el caso que luego de calcular una salida, se obtenga la misma que se tenía en el caso anterior.

La condición “if (output != computeOutput)" en método inputChanged()“ permite evitar que al notificar a una compuerta que su entrada a cambiado se agregue un evento de cambio en caso de que al calcular su salida esta se mantenga en su estado actual.

¿Se pueden eliminar?

Si se pueden eliminar, más aún al eliminar la condición en el método update() no se tendrá ninguna variación en el resultado de la simulación, sin embargo al eliminar la condición en inputChanged() ésta provocará que se almacenen más cambios de estado de los necesarios, lo que hará aumentar el tamaño del archivo de salida.

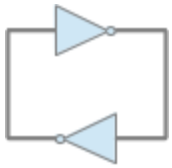
¿Qué se gana con tenerlas?

Al eliminar la condición en el método update() esta no provoca mayor problema ya que se volverá a evaluar la salida en inputChanged() y no se agregara el cambio de evento, lo único que ocurrirá es que notifica el cambio siempre que se solicita un update, sin embargo si se elimina en la condición en el segundo método (inputChanged()) esté almacenará siempre el evento aunque no exista cambio posterior en la salida.

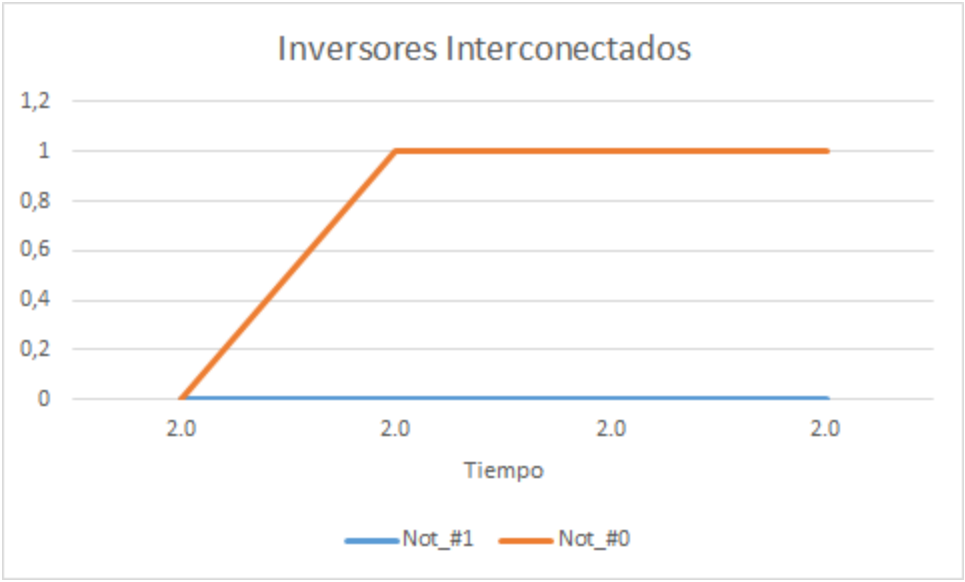
Lo que se gana con tenerlas es evitar en el primer caso un llamado innecesario a un método y luego evitar que se aumente en demasía el tamaño del archivo de salida al almacenar muchos eventos de cambio para cambios en las salidas que se mantendrán en el valor anterior.

c) Cambie y el archivo DigitalCircuit.java para simular el circuito de la figura 1. Utilice 2 [ns] (en realidad las unidades de tiempo son irrelevantes) como retardo de cada inversor.
Muestre un diagrama temporal para la salida de ambos inversores. ¿Obtiene usted un estado estable? ¿Cómo explica usted el resultado?

Circuito a Simular: Inversor Interconectados



Resultados obtenidos:



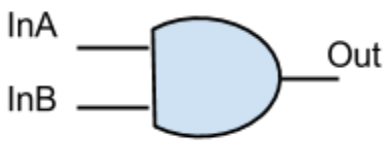
Si, se obtiene un estado estable al primer cambio a los 2[ns], esto se debe a que, si bien ambas compuertas al ejecutar el inputchange() agregan un evento para poder cambiar sus salidas, la primera cambia su salida pero la segunda al comprobar si su entrada alteraría su salida en el update(), el circuito ya estaba estable por lo que no cambia su salida y el circuito se mantiene estable.

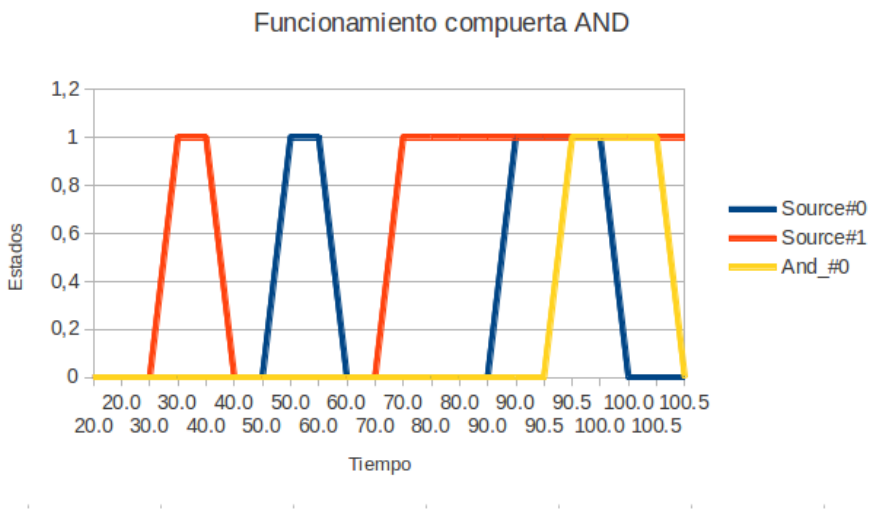
d) Cree la clase And.java. Utilice los mismos tipos de constructores de Or.java

```
/*Clase And.java*/
public class And extends Gate {
  /*Constructor*/
  public And (float delay, Simulator s) {
    super(delay, nextId++, s);
  }
  /*Métodos para conectar entradas*/
  public void connectInputA(Wire w){
    inA = w;
    w.connectTo(this);
  }
  public void connectInputB(Wire w){
    inB = w;
    w.connectTo(this);
  }

/*Método para calcular salida*/
  public boolean computeOutput(){
    boolean a = inA.getValue();
    boolean b = inB.getValue();
    return(a && b);
  }
  /*atributos de la clase: entradas y número de compuerta*/
  private Wire inA, inB;
  private static int nextId = 0;
}
```

Para probarla, hacemos un circuito básico compuesto por una compuerta and y un delay de 0.5 [unidad de tiempo], y graficamos su salida para las entradas presentadas.

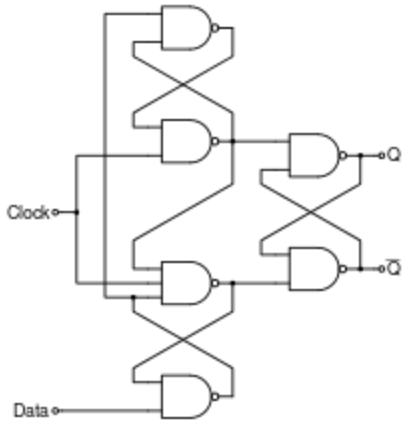




Se puede observar que la compuerta cumple su objetivo, al cambiar a estado ON (amarillo en 1 en 90,5) sólo después de que las dos fuentes están ON.

e) Cree la clase Nand.java, en este caso incluya el método:
void connectInput(Wire w);

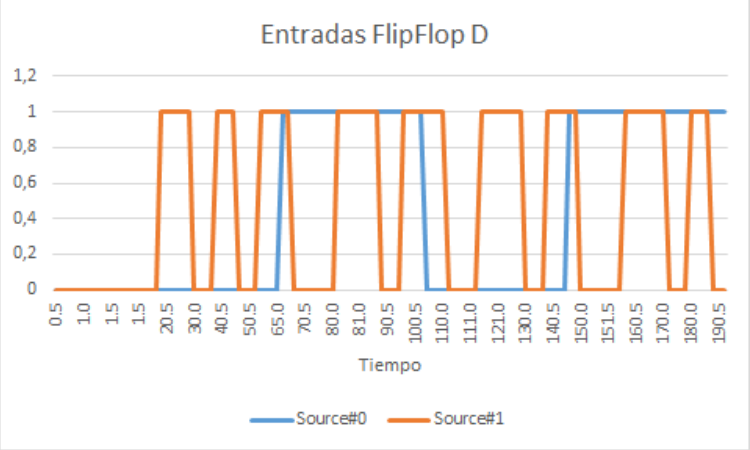
Circuito a Simular: FlipFlop D



Implementación y dificultades:

Para implementar la compuerta Nand con múltiples entradas se encontró el desafío de poder almacenar un número indefinido de referencias a entradas. Para la solución se utilizó un ArrayList de Wire (ArrayList <Wire>) lo que permite almacenar un número indefinido de Wire y luego recorrer sus elementos y así poder computar la salida de la compuerta.

Resultados obtenidos:



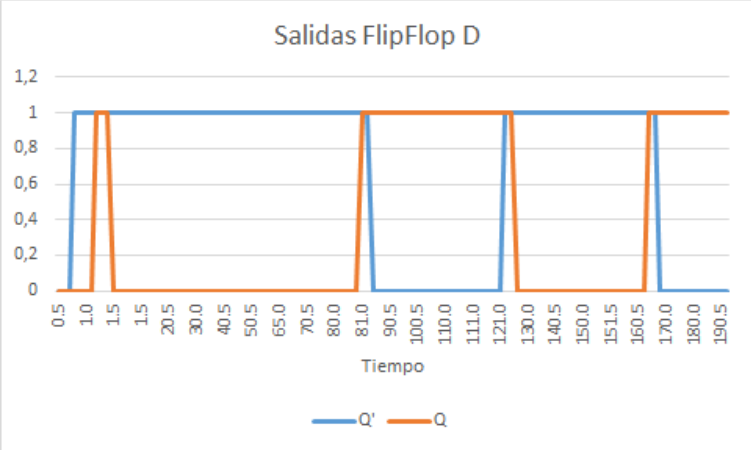

Salidas FlipFlop D
1,2
1
0,8
0,6
0,4
0,2
0
0.5
1.0
1.5
1.5
20.5
30.0
40.5
50.5
65.0
70.5
80.0
81.0
90.5
100.5
110.0
111.0
121.0
130.0
140.5
150.0
151.5
160.5
170.0
180.0
190.5
Tiempo
Q'
Q

Diagrama de clases e interfaces y sus referencias: