Simulación de Circuitos Digitales usando SDLC++

Vicente González Ruiz

Depto de Arquitectura de Computadores y Electrónica

17 de mayo de 1999

1 Qué es SDLC++ y qué requiere

SDLC++ (Simulador Digital en Lenguaje C++) es un programa de computadora escrito en lenguaje C++, pensado para simular circuitos electrónicos digitales, descritos mediante redes de puertas lógicas.

Para usar SDLC++ es necesario disponer al menos de:

- Una computadora.
- Un compilador de C++ para dicha computadora.
- Un editor de ficheros ASCII.

2 Ventajas

Entre las principales ventajas podemos destacar:

- **El simulador consume pocos recursos.** La complejidad del circuito a simular depende de la capacidad de la computadora. Los alumnos han llegado a simular una versión reducida del microprocesador MIPS R2000 sin ningún problema. Además, es posible utilizar la compilación separada lo que permite compilar circuitos muy grandes sin tener que recompilar todo el código desarrollado hasta el momento.
- **Es posible un desarrollo jerárquico.** La descripción de un circuito se realiza describiendo las puertas lógicas que lo forman y las conexiones entre ellas. Una vez desarrollado un circuito, puede usarse como una "caja negra", lo que permite la construcción jerárquica de circuitos más complejos.
- **Concentra la precisión de la simulación sólo en el modelo de las puertas lógicas.** Puesto que todo circuito a simular es, independientemente de su complejidad, una red de puertas lógicas interconectadas, la "fidelidad" de la simulación depende sólo de como de buenos son los modelos implantados en las puertas lógicas. Así, es posible mejorar las

características del simulador, programando un modelo mejor. El simulador se ha escrito pensando en que, independientemente de lo complejo que sean estos modelos, la descripción de los circuitos no se vea afectada.

3 Inconvenientes

Como siempre, no hay nada perfecto:

- **El interfase de programación no es muy amigable.** SDLC++ se programa usando un editor de ficheros ASCII y se ejecuta desde la línea de comandos. No existe ningún "interfase gráfico" que permita describir los circuitos. Esto sin embargo, permite utilizar el simulador en ausencia de computadoras sin capacidades gráficas.

- **Es necesario conocer la sintaxis básica del lenguaje C++ y tener unas nociones sobre programación orientada a objetos.** Es un requerimiento lógico del simulador, ya que se programa en C++ y todos los elementos digitales son descritos como objetos. A pesar de todo, este inconveniente puede verse como un aspecto positivo si deseas aprender C++ y programar siguiendo esta filosofía de diseño del programas.

- **Actualmente, SDLC++ no realiza una simulación eléctrica al estilo SPICE.** Debido a la simpleza del modelo desarrollado de las puertas lógicas, la salida del simulador son niveles lógicos y no niveles eléctricos, aunque ésto, depende únicamente de que los modelos de las puertas puedan ejecutar una simulación eléctrica. Como ya hemos indicado, la descripción de los circuitos a partir de puertas no debe verse afectada por un cambio de este tipo.

- **Actualmente, no se tienen en cuenta los efectos de fan-out.** Pero, pueden programarse si se cambia la descripción de las puertas lógicas. Insistimos, el resto de los ficheros que describen circuitos son independientes.

- **No se contempla tiempo real.** Sin saber cómo de rápidas son las puertas lógicas, no es posible dar tiempos de simulación reales. En realidad ésto es imposible si no se lleva el circuito al silicio, cosa que no hace SDLC++. Sin embargo, puede medirse la velocidad de un circuito en términos de iteraciones por simulación. Así, si simulamos dos circuitos distintos que realizan el mismo trabajo, es fácil ver cuál es más rápido.

4 SDLC++ ahora: su estructura jerárquica

Actualmente, SDLC++ se compone de los siguientes ficheros escritos en C++.

1. `gates.cc/gates.h`. Almacenan la implementación de las puertas lógicas. Es un fichero muy importante para el simulador ya que su comportamiento depende básicamente de cómo de bien están simuladas.

2. `clock.cc/clock.h`. Implementan un reloj de periodo programable que se utiliza para excitar circuitos secuenciales.

3. `latches.cc/latches.h` almacenan la implementación de los principales cerrojos utilizados.
4. `flip-flops.cc/flip-flops.h` describen los flip-flops implementados.
5. `counters.cc/counters.h` describen los contadores implementados.

Debido a la clasificación realizada de forma natural, básicamente, los cerrojos se implementan utilizando puertas lógicas, los flip-flops utilizando cerrojos, y los contadores, utilizando flip-flops.

5 Las señales digitales

En el simulador, una señal digital se representa utilizando 8 bits, lo que permite un rango de valores que va desde 0 a 255. El simulador interpreta actualmente el valor 0 como el nivel lógico bajo y el valor 255 como el nivel lógico alto. Valores intermedios no son significativos, aunque si en un futuro se piensa en una simulación eléctrica, pueden llegar a serlo.

6 Las puertas lógicas

El modelo utilizado de puerta lógica simula:

1. **La operación lógica** de la puerta (NOT, AND, OR, etc).
2. **El tiempo de propagación** a través de la puerta. En concreto, cada puerta tiene asociada un tiempo de retardo igual a una iteración del simulador.
3. **El tiempo de establecimiento** de las señales lógicas en la entrada(s) a la puerta. En el modelo actual, para que una señal sea considerada debe permanecer constante al menos dos iteraciones del simulador.

A continuación se muestra la simulación de una puerta AND de dos entradas. Cada línea es una iteración del simulador.

```
iter   i0  i1 out
----  --- --- ---
   1  255   0   0
   2  255   0   0
   3  255   0   0
   4    0   0   0
   5    0   0   0
   6    0 255   0
   7  255 255   0
   8  255 255 255
   9  255 255 255
  10    0 255 255
  11    0 255   0
  12    0 255   0
  13  255   0   0
  14  255   0   0
  15  255   0   0
```

```
16      0   0   0
17      0   0   0
18      0   0   0
19    255   0   0
20    255 255   0
21    255 255 255
22      0 255 255
23      0 255   0
24      0 255   0
25    255 255   0
26    255 255 255
27    255   0 255
28      0   0   0
```

La puerta AND tiene el siguiente interfase:

```
class AND {
  WIRE delay;
public:
  AND();
  void run(WIRE &i0, WIRE &i1, WIRE &o);
};
```

El código anterior escrito en C++ declara una clase llamada AND. Un objeto de la clase AND posee una única variable llamada `delay` del tipo WIRE (unsigned char) y una función que lo manipula. La única función `run()`, tiene tres argumentos. Los dos primeros son las dos entradas a la puerta AND y el tercero es la salida. El paso de todos los parámetros se realizan por referencia, por lo que todos ellos pueden ser cambiados por la función `run()`.

El resto de puertas tiene una descripción semejante. La declaración de todas las clases de puertas forman el fichero `gates.h`, que como se puede ver, NO CONTIENE CÓDIGO EJECUTABLE. Esto es necesario si deseamos utilizar la compilación separada para desarrollar con rapidez grandes programas de simulación.

El fichero `gates.cc` contiene la implementación de las funciones declaradas en la clase asociada. En concreto, para la puerta AND, ésta es la implementación de la función `run()`:

```
void AND::run (WIRE &i0, WIRE &i1, WIRE &o) {
  WIRE tmp = i0 & i1;
  if(delay==tmp) o=tmp;
  delay=tmp;
  if(o<127) o=0; else o=255;
}
```

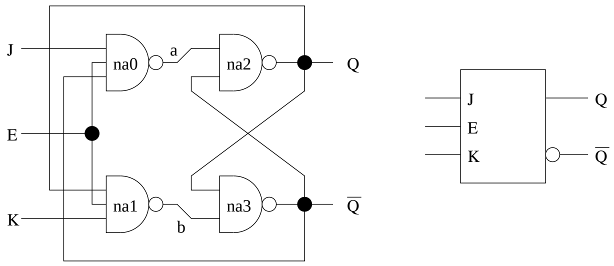
El código escrito realiza la siguiente tarea a grandes rasgos: (1) calcula la función and lógica bit a bit de las variables `i0` e `i1` y la almacena en una variable temporal llamada `tmp`, (2) comprueba si la anterior salida de la puerta AND es igual a la posible salida actual almacenada en `tmp`. Si coinciden, la salida `o` es igual a `tmp`, (3) guardamos la AND de los dos valores de entrada para la comparación anterior en la próxima iteración y (4) evitamos posibles ruidos por falta de inicializaciones de las variables lógicas.

7 Ejemplos de simulación

Pasemos a ver ahora unos cuantos ejemplos de simulación de circuitos digitales sencillos.

7.1 El cerrojo JK con entrada de habilitación

El cerrojo JK es un circuito secuencial digital y como tal presenta realimentaciones. Su esquema lógico se expone a continuación, junto con el símbolo lógico asociado:



La descripción del cerrojo JK con entrada de habilitación en SDLC++ es la siguiente:

```
class JK_LATCH_SINC {
  NAND3 na0,na1;
  NAND na2,na3;
  WIRE a,b;
public:
  void run(WIRE &J, WIRE &K, WIRE &E, WIRE &Q, WIRE &Qn);
};
```

El código anterior indica que un objeto de la clase `JK_LATCH_SINC` está compuesto de (1) 2 objetos de la clase `NAND3` llamados `na0`,`na1`, (2) 2 objetos de la clase `NAND` llamados `na2` y `na3` y (3) dos líneas conductoras llamadas `a` y `b`. La función que simula el cerrojo se llama `run()` y tiene tantos argumentos como entradas y salidas posee el cerrojo. El código anterior se almacena en el fichero `latches.h` y no genera código ejecutable.

La implementación del cerrojo se almacena en `latches.cc` y en concreto es:

```
void JK_LATCH_SINC::run(WIRE &J, WIRE &K, WIRE &E, WIRE &Q, WIRE &Qn) {
  na0.run(J,E,Qn,a);
  na1.run(K,E,Q,b);
  na2.run(a,Qn,Q);
  na3.run(b,Q,Qn);
}
```

Como puede verse, se trata simplemente de llamar a las 4 puertas NAND que forman el cerrojo, colocando las entradas y salida adecuadas. EL ORDEN EN QUE SE INVOCA A LAS PUERTAS ES IRRELEVANTE.

Pasemos a la simulación del cerrojo. Como todo programa en C, hay que escribir una función llamada `main()` que genere el proceso de simulación. En concreto, para el cerrojo JK supongamos el siguiente fichero llamado `main.cc`:

```
#include <iostream.h>
#include <iomanip.h>
#include "clock.h"
#include "gates.h"
#include "latches.h"

void main() {
  int iter=200;
  WIRE J=0,K=0,E=0,Q=0,Qn=1;
  CLOCK ck0(17),ck1(23),ck2(100);
  JK_LATCH_SINC l;
  while(iter--) {
    J=ck0.run(iter);
    K=ck1.run(iter);
    E=ck2.run(iter);
    l.run(J,K,E,Q,Qn);
    cout << setw(4) << (int)J
         << setw(4) << (int)K
         << setw(4) << (int)E
         << setw(4) << (int)Q
         << setw(4) << (int)Qn
         << '\n';
  }
}
```

Para compilar el simulador, podemos compilar todo el código necesario en una única compilación.

```
c++ main.cc latches.cc gates.cc clock.cc
```

El inconveniente de este método es que si sólo estamos modificando el fichero `main.cc`, no es necesario recompilar toda la biblioteca. Esta puede estar ya compilada de antemano utilizando los comandos:

```
c++ -c clock.cc
c++ -c gates.cc
c++ -c latches.cc
```

y para compilar nuestro ejemplo gastando menos tiempo, teclearíamos:

```
c++ main.cc latches.o gates.o clock.o
```

En ambos casos, se genera un fichero ejecutable llamado `a.out` que al ejecutarse provoca la salida de las variables lógicas `J`,`K`,`E`,`Q` y `Qn` por la salida estándar. Para redirigirla escribiremos:

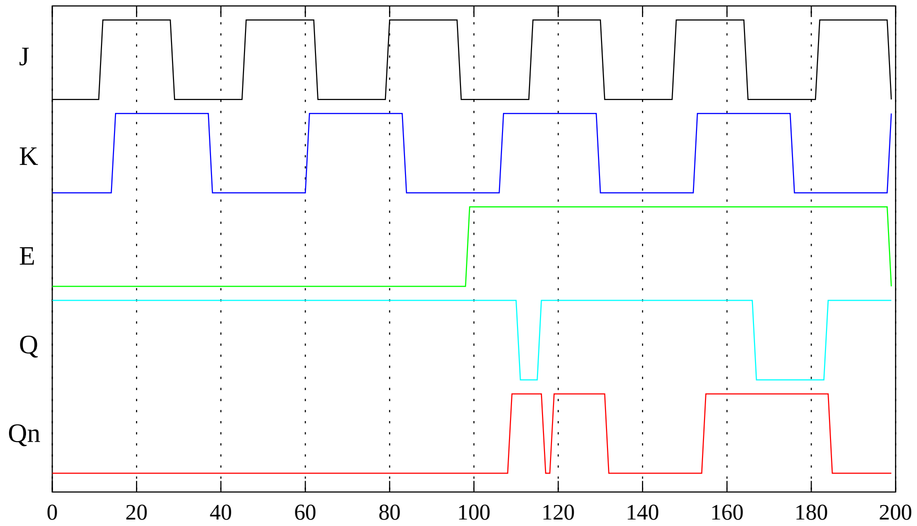
```
a.out > 1
```

El fichero `1` puede ser visualizado por ejemplo usando `gnuplot`, utilizando la configuración siguiente:

```
set grid
set yrange[0:26]
```

```
set noytics
plot "1" using ($1/60+21) title "J" with lines,\
"1" using ($2/60+16) title "K" with lines,\
"1" using ($3/60+11) title "E" with lines,\
"1" using ($4/60+06) title "Q" with lines,\
"1" using ($5/60+01) title "Qn" with lines
pause -1
```

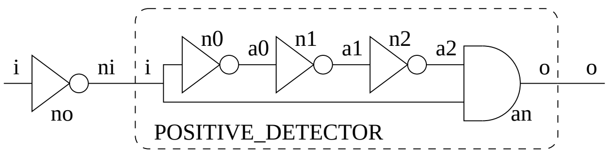
Y la salida (aproximadamente) provocada es:



7.2 El flip-flop JK activo por flanco negativo

El flip-flop JK es un cerrojo JK con entrada de habiltación, habilitado durante un intervalo de tiempo muy corto justamente después de la transición negativa de reloj.

El circuito que detecta las transiciones negativas de reloj se llama *detector de transiciones negativas* y se describe a continuación.



En SDLC++ el detector de transiciones negativas se puede declarar así:

```
class POSITIVE_DETECTOR {
  NOT n0,n1,n2;
  AND an;
  WIRE a0,a1,a2;
public:
  void run(WIRE &i, WIRE &o);
};
class NEGATIVE_DETECTOR {
```

```
  POSITIVE_DETECTOR pd;
  NOT no;
  WIRE ni;
public:
  void run(WIRE &i, WIRE &o);
};
```

y se implementa como:

```
void POSITIVE_DETECTOR::run(WIRE &i, WIRE &o) {
  n0.run(i,a0);
  n1.run(a0,a1);
  n2.run(a1,a2);
  an.run(i,a2,o);
}
void NEGATIVE_DETECTOR::run(WIRE &i, WIRE &o) {
  no.run(i,ni);
  pd.run(ni,o);
}
```

Podemos simular el detector de transiciones utilizando el siguiente "entrenador virtual" almacenado como `main.cc`:

```
#include <iostream.h>
#include <iomanip.h>
#include "clock.h"
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"

void main() {
  int iter=100;
  WIRE a,b;
  NEGATIVE_DETECTOR x;
  CLOCK ck(20);
  while(iter--) {
    a=ck.run(iter);
    x.run(a,b);
    cout << setw(4) << (int)a
         << setw(4) << (int)b
         << '\n';
  }
}
```

Para compilar el ejemplo, utilizaremos:

```
c++ -c gates.cc
c++ -c latches.cc
c++ -c clock.cc
c++ main.cc flip-flops.cc latches.o gates.o clock.o
```

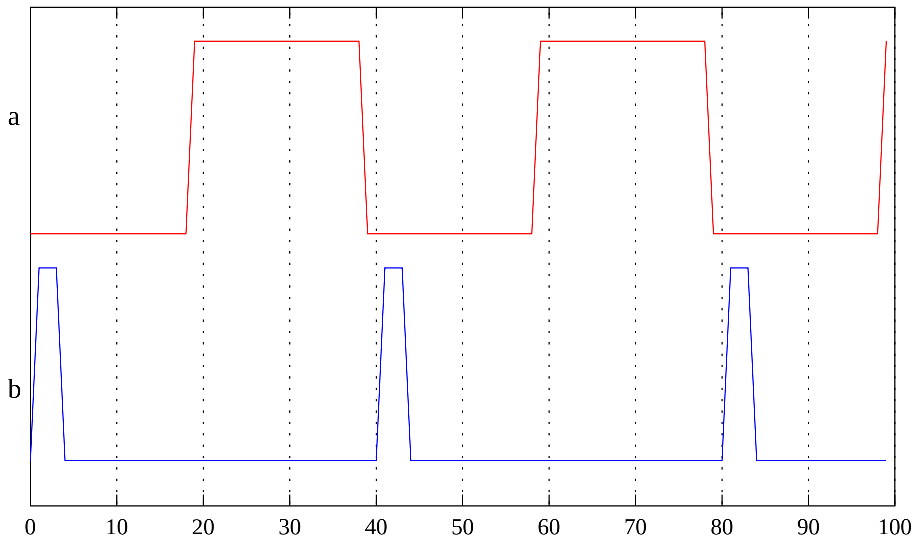
Ya que el detector de transiciones se almacena en el fichero `flip-flops.cc`, notar que de tener precompilados los ficheros `gates.cc`, `latches.cc` y `clock.cc`, sólo hubiéramos escrito el último comando. Ejecutamos el simulador escribiendo:

```
a.out > 1
```

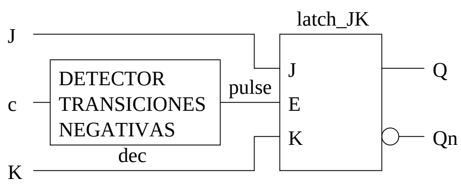
A continuación, utilizando `gnuplot` y ejecutando la siguiente configuración:

```
set grid
set yrange[0:11]
set nokey
set noytics
plot "1" using ($1/60+06) with lines,\
"1" using ($2/60+01) with lines
pause -1
```

obtenemos la siguiente salida:



Para formar el flip-flop a partir del cerrojo, conectamos el circuito detector de transiciones negativas a la entrada de habilitación del cerrojo JK. El esquema lógico podría ser el siguiente:



La clase que describe los componentes del flip-flop es:

```
class NEG_JK_FF {
  NEGATIVE_DETECTOR dec;
  WIRE pulse;
  JK_LATCH_SINC latch_JK;
public:
  void run(WIRE &J, WIRE &K, WIRE &c, WIRE &Q, WIRE &Qn);
};
```

y el código que la implementa es:

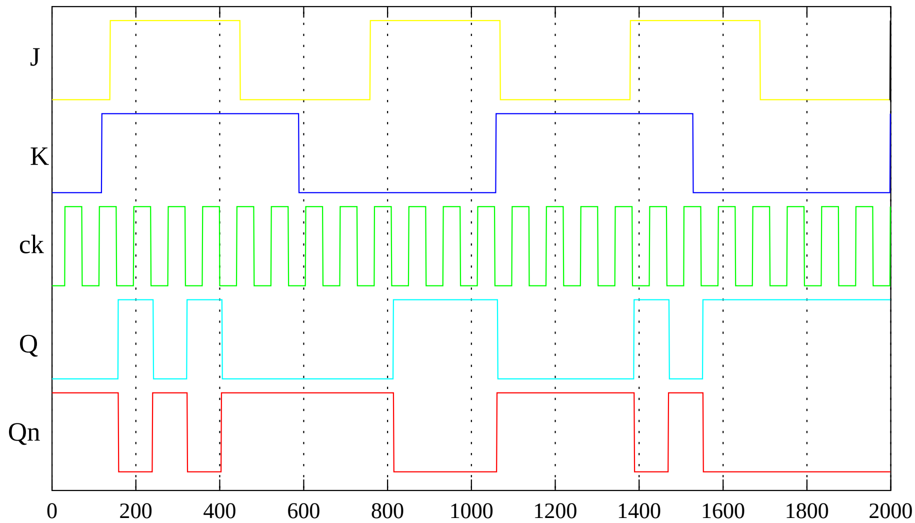
```
void NEG_JK_FF::run(WIRE &J, WIRE &K, WIRE &c, WIRE &Q, WIRE &Qn) {
  dec.run(c,pulse);
  latch_JK.run(J,K,pulse,Q,Qn);
}
```

La simulación del flip-flop JK puede realizarse con el siguiente `main.cc`:

```
#include <iostream.h>
#include <iomanip.h>
#include "clock.h"
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"

void main() {
  int iter=2000;
  WIRE J,K,c,Q=0,Qn=255;
  CLOCK ck0(310),ck1(470),ck2(41);
  NEG_JK_FF l;
  while(iter--) {
    J=ck0.run(iter);
    K=ck1.run(iter);
    c=ck2.run(iter);
    l.run(J,K,c,Q,Qn);
    cout << setw(4) << (int)J
         << setw(4) << (int)K
         << setw(4) << (int)c
         << setw(4) << (int)Q
         << setw(4) << (int)Qn
         << '\n';
  }
}
```

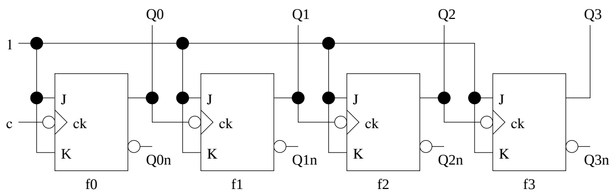
y se muestra a continuación:



Por último indicar que es curioso que el en laboratorio sean necesarias también 3 puertas NOT en cascada, para conseguir un pulso de habilitación de suficiente duración. Si se utiliza una sola puerta NOT, el flip-flop no funciona. Esto nos hace pensar en que, el modelo realizado de las puertas es bastante acertado.

7.3 Un contador binario asíncrono ascendente

Este ejemplo, tiene como objetivo demostrar el inconveniente de los contadores asíncronos: los flip-flops se disparan en serie, lo que provoca problemas de decodificación de estados si la frecuencia de reloj aumenta lo suficiente. Este sería su esquema lógico:



Veamos como se construye un contador binario asíncrono de 4 bits en SDLC++, que es simplemente 4 flip-flops JK conectados en cascada. En el fichero `counters.h` tenemos:

```
class AU_4BIN_COUNTER {
  NEG_JK_FF f0,f1,f2,f3;
  WIRE Q0n,Q1n,Q2n,Q3n;
public:
  void run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3);
};
```

Y en el fichero `counters.cc` tenemos:

```
WIRE Vcc=255;
void AU_4BIN_COUNTER::run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3) {
```

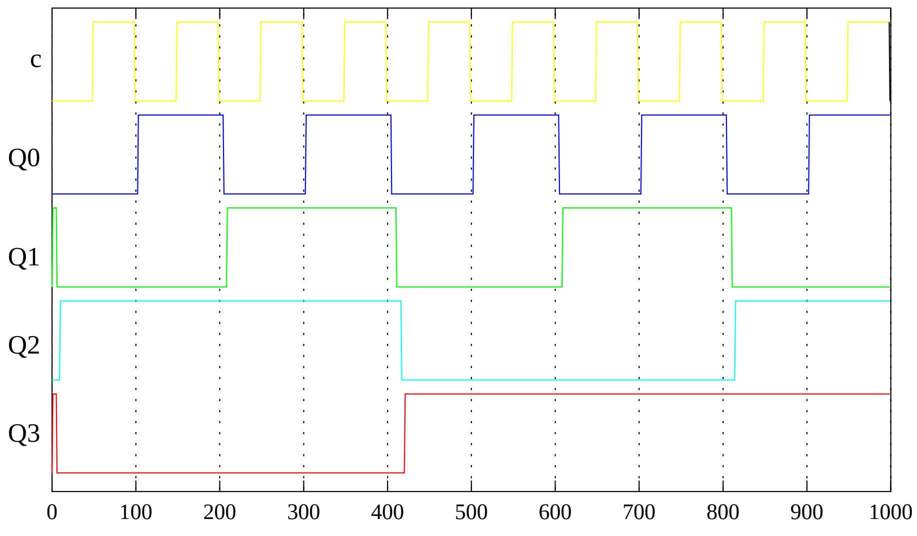
```
  f0.run(Vcc,Vcc,ck,Q0,Q0n);
  f1.run(Vcc,Vcc,Q0,Q1,Q1n);
  f2.run(Vcc,Vcc,Q1,Q2,Q2n);
  f3.run(Vcc,Vcc,Q2,Q3,Q3n);
}
```

El fichero de entrenamiento `main.cc` consiste en:

```
#include <iostream.h>
#include <iomanip.h>
#include "clock.h"
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "counters.h"

void main() {
  int iter=1000;
  WIRE c=0,Q0=0,Q1=0,Q2=0,Q3=0;
  CLOCK ck(50);
  AU_4BIN_COUNTER co;
  while(iter--) {
    c=ck.run(iter);
    co.run(c,Q0,Q1,Q2,Q3);
    cout << setw(4) << (int)c
         << setw(4) << (int)Q0
         << setw(4) << (int)Q1
         << setw(4) << (int)Q2
         << setw(4) << (int)Q3
         << '\n';
  }
}
```

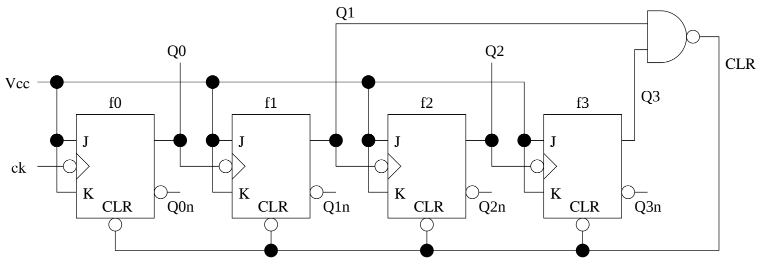
Y la salida generada es:



Pueden observarse los retrasos crecientes de las basculaciones de los biestables respecto de la transición negativa de reloj, cuando analizamos sobre todo los bits más significativos.

7.4 Un contador BCD asíncrono ascendente

Este último ejemplo tiene el objetivo de mostrar por qué un simulador puede ser muy útil. El circuito que vamos a simular en primer lugar ha sido extraido del libro *Fundamentos de Sistemas Digitales*, de T.L. Floyd. Se trata de un contador BCD asíncrono, que utiliza las entradas de CLEAR para truncar la secuencia de cuenteo una vez que se alcanza el estado 10. He aquí su esquema lógico,



que se implementa en SDLC++ de la siguiente forma. En `counters.h` almacenamos:

```
class AU_BCD_COUNTER_CLR {
  NEG_JK_FF_PC f0,f1,f2,f3;
  NAND na;
  WIRE Q0n,Q1n,Q2n,Q3n,CLR;
 public:
  void run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3);
};
```

En `counters.cc` escribimos:

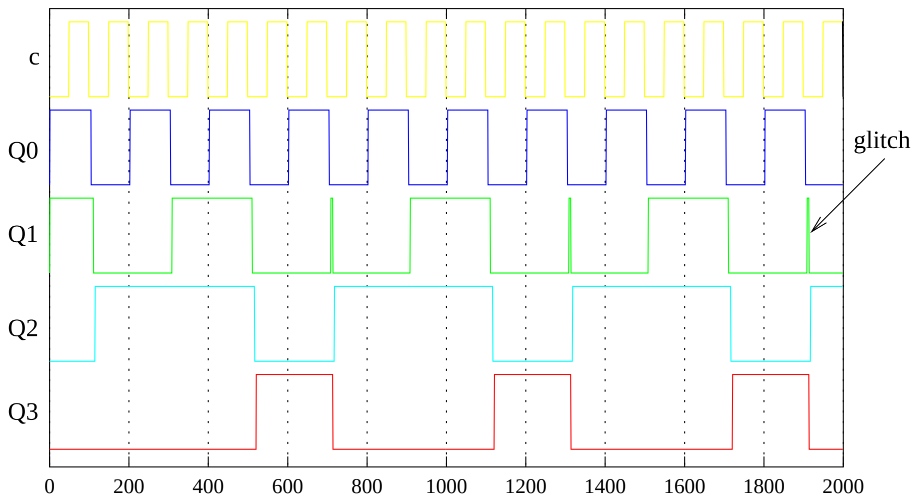
```
void AU_BCD_COUNTER_CLR::run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3) {
  f0.run(Vcc,Vcc,ck,Vcc,CLR,Q0,Q0n);
  f1.run(Vcc,Vcc,Q0,Vcc,CLR,Q1,Q1n);
  f2.run(Vcc,Vcc,Q1,Vcc,CLR,Q2,Q2n);
  f3.run(Vcc,Vcc,Q2,Vcc,CLR,Q3,Q3n);
  na.run(Q1,Q3,CLR);
}
```

Y como `main.cc` tenemos:

```
#include <iostream.h>
#include <iomanip.h>
#include "clock.h"
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "counters.h"
```

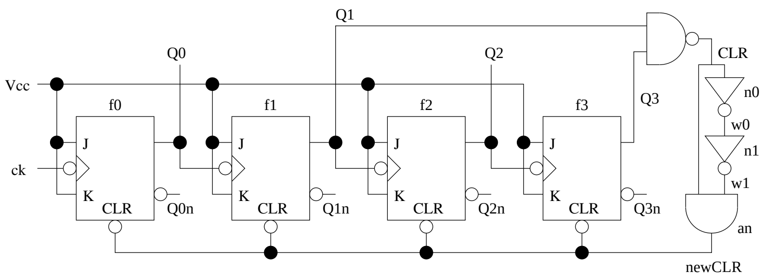
```
void main() {
  int iter=2000;
  WIRE c=0,Q0=0,Q1=0,Q2=0,Q3=0;
  CLOCK ck(50);
  AU_BCD_COUNTER_CLR co;
  while(iter--) {
    c=ck.run(iter);
    co.run(c,Q0,Q1,Q2,Q3);
    cout << setw(4) << (int)c
         << setw(4) << (int)Q0
         << setw(4) << (int)Q1
         << setw(4) << (int)Q2
         << setw(4) << (int)Q3
         << '\n';
  }
}
```

El esquema presentado es "inteligente", pero NO FUNCIONA, ni en el simulador ni en la realidad. Esta es su salida:



La razón es la siguiente. En el cuarto ciclo de reloj, se decodifica el estado 10 (esto ocurriría en el ciclo 10, si inicialmente hubiéramos partido del estado 0000) y los flip-flops 1 y 3 son borrados mediante la entrada asíncrona de CLEAR. Hasta aquí todo bien, pero la bajada de Q1 provoca que Q2 suba a 1, cuando debiera quedarse en cero. De hecho, justo en el instante en que se produce la bajada de Q1, la entrada CLEAR de Q2 es cero, pero sólo conserva este valor un tiempo igual al tiempo de retardo de la puerta NAND. Este tiempo es más pequeño que el tiempo que tarda en propagarse la señal a través de la entrada de reloj del flip-flop 2 y por tanto, no se borra. En consecuencia, este contador recorre los estados 4,5,6,7,8 y 9.

Para solucionar el problema, podemos hacer que el pulso de borrado sea algo más largo, suficiente para que la propagación de la transición de bajada del glitch sea anulada. Esto se puede hacer como se indica en la figura siguiente.



Que se implementa así. `counters.h`:

```
class AU_BCD_COUNTER_CLR {
  NEG_JK_FF_PC f0,f1,f2,f3;
  NAND na;
  WIRE Q0n,Q1n,Q2n,Q3n,CLR;
  // Circuito de retardo para incrementar la duraci'on del pulso de borrado
  NOT n0,n1;
  AND an;
  WIRE w0,w1,newCLR;
 public:
  void run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3);
};
```

`counters.cc`:

```
void AU_BCD_COUNTER_CLR::run(WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2, WIRE &Q3) {
  f0.run(Vcc,Vcc,ck,Vcc,newCLR,Q0,Q0n);
  f1.run(Vcc,Vcc,Q0,Vcc,newCLR,Q1,Q1n);
  f2.run(Vcc,Vcc,Q1,Vcc,newCLR,Q2,Q2n);
  f3.run(Vcc,Vcc,Q2,Vcc,newCLR,Q3,Q3n);
  na.run(Q1,Q3,CLR);
  n0.run(CLR,w0); n1.run(w0,w1); an.run(CLR,w1,newCLR);
}
```

Ahora la secuencia de cuenteo es correcta.