# Simulador Digital en Lenguaje C: SDLC

Vicente González Ruiz *vruiz@{iron,gogh,filabres}.ualm.es*
Jose Antonio Martínez García *jamartin@{iron.dali,filabres}.ualm.es*
Inmaculada García Fernández *inma@{iron,data,filabres}.ualm.es*
*Depto de Arquitectura de Computadores y Electrónica*
*Universidad de Almería*

29 de enero de 2001

**Resumen**

Este documento describe una forma sencilla para la simulación en lenguaje C de dispositivos digitales. Se emula tanto su comportamiento lógico como su transcurrir temporal, pudiéndose así diseñar sistemas digitales a bajo coste. Usando un compilador de lenguaje C K&R [6] y sin recurrir a la programación concurrente, es posible portar la aplicación a multitud de computadores que posean un compilador de lenguaje C. Tras una introducción a la simulación secuencial, se exponen diferentes implementaciones de dispositivos digitales sencillos y el resultado de su simulación. El trabajo finaliza con la simulación de una pequeña memoria SRAM.[1]

## 1 Simulando hardware paralelo con software secuencial.

Un dispositivo electrónico, al igual que cualquier otro elemento físico, tiene un funcionamiento inminentemente paralelo, debido a que todo él trabaja simultáneamente. Si necesita más o menos tiempo para realizar una tarea determinada, es porque existe una dependencia temporal entre las diferentes señales eléctricas (analógicas o digitales) que lo recorren. Si somos capaces de seguir el rastro a estos datos, podremos predecir el comportamiento del dispositivo simulado.

La simulación de un sistema digital (que puede tratarse desde una simple puerta lógica hasta un complejo computador), se realiza actualmente con éxito con lenguajes de programación paralelos como Verilog [5]. Con él, se describen los circuitos básicos en diferentes módulos que se interpretan en paralelo. Aunque esto normalmente se hace en una única CPU, el intérprete dedica un

[1] Los ficheros con el código fuente de bibliotecas de funciones y ejemplos de circuitos implementados en SDLC, junto con este documento, pueden encontrarse vía `anonymous ftp` en: `dali.ualm.es:/pub/sdlc.tar.gz`. El fichero `sdlc.tar` se desglosa en el directorio `./sdlc`.

cuantum de tiempo de CPU a cada módulo (lo que en el ámbito de los sistemas operativos se conoce con el nombre de tiempo compartido) y de esta forma, se simula un comportamiento concurrente. Simular un sistema digital en lenguaje C no puede hacerse de la misma forma, porque el compilador genera código secuencial. La argucia necesaria para soslayar este problema es realizar una programación orientada al bit. Una señal lógica se crea en alguna parte del sistema y se destruye en otra (normalmente se mezcla con otras, para generar nuevas señales lógicas), y su existencia puede temporizarse de forma secuencial. En C (y también en la mayoría de los lenguajes de programación usados actualmente) si es posible mantener multitud de variables cuyo valor varía en el tiempo, sin más que encerrarlas en un lazo de programación (un bucle). La capacidad de simulación de una computadora bajo estas circunstancias dependerá de la cantidad de variables lógicas que pueda albergar (de su memoria) y de la velocidad con que las relacione (de su procesador). El comportamiento paralelo del sistema simulado va a ser obtenido mediante la acumulación de numerosas ejecuciones secuenciales, en cada una de las cuales, el sistema evolucionará desde un estado inicial hasta otro final (instante en el que conocemos los resultados de la simulación). El mínimo paso evolutivo dependerá de la sutileza con que se desea modelar. En nuestro caso, este paso viene dado por un único cambio estable en cualquiera de las señales digitales simuladas.

## 2 Las señales lógicas y su tratamiento de C.

La lógica digital sólo trabaja con dos estados: 1 o 0, alto o bajo, Vcc o GND. Como primera decisión de diseño del modelo, crearemos un tipo llamado `BIT` que tomará estos dos valores. Una variable declarada con este tipo, representa un terminal conductor del sistema digital que estamos diseñando. En C, se dispone de operadores lógicos suficientes para simular el comportamiento de todas las funciones lógicas usadas en el álgebra de Boole [4, 7]: AND, OR, XOR, ..., por lo que la simulación de las puertas lógicas básicas es una tarea sencilla. Sin embargo, debemos de tomar una decisión con respecto el tipo de dato que será el encargado de representar un bit de información. En C, los tipos de datos más pequeños son el carácter (`char`) y el carácter sin signo (`unsigned char`), y aunque es posible trabajar con bits, por razones de velocidad, trabajaremos con el tipo `unsigned char`, aunque desperdiciemos 7 bits por byte para conocer el valor de una variable del tipo `BIT`. El tipo se declara `unsigned` para evitar el signo del tipo `char` que es irrelevante. [6]

## 3 El tiempo de simulación.

Con el objetivo de discernir que circuito digital es más rápido que otro, es necesario simular el tiempo de excitación de los circuitos. Por esta razón se asocia a cada bit un tiempo de simulación, lo que llamamos 'tiempo de vida del bit'. Esta medición puede realizarse con una variable real (`float` o `double`) o con una variable entera (`short`, `int` o `long`). La ventaja de trabajar con un

tiempo real es que podemos medir tiempos reales (por ejemplo, en $n$segs), pero carga al proceso de simulación con computación en punto flotante innecesaria. Si medimos el tiempo con una variable entera (suficientemente grande, para evitar desbordamientos, por lo que el tipo `unsigned long` es el más recomendable), mediremos **pasos**, que en cualquier momento pueden traducirse en tiempo real sin más que conocer a cuanto tiempo equivale un paso. Esta discusión se plasma en el fichero `defs.h` que a continuación es mostrado:

```
/*
 * Definiciones para la simulacion de circuitos digitales.
 * gse. 1995.
 */

/*
 * Posibles estados para una linea.
 */
#define LO  0 /* El 0 logico */
#define HI  1 /* El 1 logico */
#define Z   2 /* Alta impedancia. */

#define Vcc HI
#define GND LO

/*
 * Un BIT contiene informacion acerca de su valor
 * y de cuando se origino.
 */
typedef struct {
  unsigned char bit;   /* Bit de informacion: 1 o 0 */
  unsigned long time;  /* Tiempo acumulado sobre el bit */
} BIT;
```

Se contempla la situación de alta impedancia como posible estado de un bit debido a que es muy útil en la construcción de las memorias y buses [2]. De esta forma es posible conectar múltiples dispositivos digitales a una misma línea física. Esto se puede hacer si en un instante dado, sólo uno de ellos presenta su salidas activas (a 1 o a 0) y el resto en alta impedancia.

# 4 Las puertas básicas.

Por suerte, la mayoría de los sistemas digitales existentes basan su funcionamiento en una corta lista de dispositivos básicos: las puertas lógicas [4]. Esto hace posible que la simulación de sistemas digitales complejos se reduzca a la construcción de programas en los que se llaman a una serie de funciones en lenguaje C que describen las puertas básicas. Las puertas simuladas se implementan en el fichero de biblioteca `puertas.c`:

```
/*
 * puertas.c
 * Implementacion de las puertas logicas fundamentales.
 * gse. 1995.
 */

#include "defs.h"
```

```
#include "puertas.h"

#define MAX(a,b)        ((a)>(b) ? (a):(b))

/*
 * Puerta Not.
 */
void Not(in,out)
BIT in,*out; {
  /* La salida es la inversion de la entrada */
  out->bit=!in.bit;
  /* Tiempo de inversion = 1 unidad de tiempo */
  out->time=in.time+1;}

/*
 * Puerta Or.
 */
void Or(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = in1.bit | in2.bit;
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Puerta Xor.
 */
void Xor(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = in1.bit ^ in2.bit;
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Puerta And.
 */
void And(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = in1.bit & in2.bit;
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Puerta And de 3 entradas.
 */
void And3(in1,in2,in3,out)
BIT in1,in2,in3,*out; {
  out->bit = in1.bit & in2.bit & in3.bit;
  out->time = MAX(in1.time,in2.time);
  out->time = MAX(out->time,in3.time)+1;}

/*
 * Puerta Nand.
 */
void Nand(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = !(in1.bit & in2.bit);
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Puerta Nand de 3 entradas.
```

```
 */
void Nand3(in1,in2,in3,out)
BIT in1,in2,in3,*out; {
  out->bit = !(in1.bit & in2.bit & in3.bit);
  out->time = MAX(in1.time,in2.time);
  out->time = MAX(out->time,in3.time)+1;}

/*
 * Puerta Nor.
 */
void Nor(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = !(in1.bit | in2.bit);
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Puerta NXor.
 */
void NXor(in1,in2,out)
BIT in1,in2,*out; {
  out->bit = !(in1.bit ^ in2.bit);
  out->time = MAX(in1.time,in2.time)+1;}

/*
 * Buffer tri-estado.
 */
void Buffer(in,enable,out)
BIT in,enable,*out; {
  /* Comprobamos que nadie mas ha escrito */
  if(out->bit==Z)
    if(enable.bit==HI) {
      out->bit = in.bit;
      out->time = MAX(in.time,enable.time) + 1;}}
```

Se ha pensado en dar un tiempo de simulación de un paso a cada una de las puertas descritas, aunque esto es totalmente programable. De esta forma, el número de pasos acumulados en un `BIT` indica el número de puertas por las que ha pasado hasta ese momento. Circuitos combinacionales construidos a partir de ellas, generan bits con vida más larga, puesto que han de pasar por un número mayor de puertas antes de alcanzar la salida.

# 5 Algunas funciones de ayuda.

En un circuito digital es frecuente el uso de dispositivos síncronos dependientes de una señal de reloj. Podemos construir un reloj (un generador de onda cuadrada) realimentando una puerta `NOT`. También es de ayuda un generador de bits pseudo-aleatorio [3], para testear con comodidad los circuitos diseñados. Estas funciones se han creado en el fichero `lib.c`:

```
/*
 * lib.c
 * Simulacion de circuitos digitales: biblioteca
 * de funciones.
 * gse. 1995.
```

```
 */

#include "defs.h"

/*
 * Construimos el reloj mediante un oscilador binario
 * realimentando un inversor. El tiempo de vida del
 * reloj se declara nulo para que no influya en el resto
 * de los bits temporizados.
 */
void Clock(out)
BIT *out; {
  Not(*out,out);
  out->time=0;}

/*
 * Un generador de numeros enteros pseudo-aleatorio
 * entre 0 y 65536.
 */
unsigned Random() {
  static unsigned next=17;
  next *= 1103515245 + 12345;
  next %= 65537;
  return next;}

/*
 * Genera numeros enteros pseudo-aleatorios
 * entre 0 y 1.
 */
BIT RandomBIT() {
  BIT b;
  b.bit=Random()%2;
  b.time=0;
  return b;}
```

# 6 Los circuitos combinacionales básicos.

En esta sección se tratan aquellos circuitos digitales, que implementan una determinada función lógica. Son bastante sencillos de describir y simular. Se trata de codificadores, decodificadores, multiplexores, etc. Se distinguen por no presentar ninguna realimentación desde su salida hacia su entrada. En el fichero `ccb.c` se presentan diferentes ejemplos:

```
/*
 * ccb.c
 * Circuitos combinacionales basicos.
 * gse. 1995.
 */

#include "defs.h"
#include "puertas.h"

/*
 * Decodificador 2:4.
 * Ver Taub, pag: 108.
```

```
 */
void Decod4(in,out)
BIT in[2],out[4]; {
  BIT w0,w1;
  Not(in[0],&w0);
  Not(in[1],&w1);
  And(w0,w1,&out[0]);
  And(w0,in[1],&out[1]);
  And(in[0],w1,&out[2]);
  And(in[0],in[1],&out[3]);}

/*
 * Decodificador de 3:8, con entrada de seleccion.
 */
void Decod8(in,select,out)
BIT in[3],select,out[8]; {
  BIT w0,w1,w2;
  BIT o0,o1,o2,o3,o4,o5,o6,o7;
  Not(in[0],&w0);
  Not(in[1],&w1);
  Not(in[2],&w2);
  And3(w0,w1,w2,&o0);
  And3(w0,w1,in[2],&o1);
  And3(w0,in[1],w2,&o2);
  And3(w0,in[1],in[2],&o3);
  And3(in[0],w1,w2,&o4);
  And3(in[0],w1,in[2],&o5);
  And3(in[0],in[1],w2,&o6);
  And3(in[0],in[1],in[2],&o7);
  And(select,o0,&out[0]);
  And(select,o1,&out[1]);
  And(select,o2,&out[2]);
  And(select,o3,&out[3]);
  And(select,o4,&out[4]);
  And(select,o5,&out[5]);
  And(select,o6,&out[6]);
  And(select,o7,&out[7]);}
```

# 7 Elementos simples de memoria: los biestables.

Un biestable es un dispositivo digital que memoriza un bit. Las diferentes formas de realizar esta tarea dan lugar a la familia de biestables conocidos. La simulación de un biestable introduce un grado de dificultad respecto a la de un circuito combinacional: las realimentaciones. Debido a que usamos un lenguaje de programación secuencial, no podemos en una sola pasada simular un biestable, pues su salida depende de esa primera pasada. Es necesario el uso de lazos para volver a calcular las salidas en función de las entradas y las salidas en el instante anterior. La mayoría de los biestables se han implementado en el fichero `ff.c` que a continuación es mostrado:

```
/*
 * ff.c
 * Implementacion de los biestables basicos.
 * gse. 1995.
 */
```

```
#include "defs.h"
#include "puertas.h"
#include "ff.h"

/*
 * Flip Flop SR asincrono construido con puertas Nand.
 *
 *          +------+
 *    R ----+      |
 *          | NAND +--+-- Q
 *        +-+      |  |
 *        | +------+ /
 *         \       /
 *           \   /
 *             X
 *           /   \
 *         /       \
 *        | +------+ \
 *        +-+      |  |
 *          | NAND +--+-- Qn
 *    S ----+      |
 *          +------+
 *
 *     S R | Q Qn
 *    -----+-----
 *     0 0 | 1 1  (estado indeseable)
 *     0 1 | 0 1  (reset)
 *     1 0 | 1 0  (set)
 *     1 1 | Q Qn (estado anterior)
 */
void FFSR_Nand(S,R,Q,Qn)
BIT S,R,*Q,*Qn;{
  unsigned i;
  ITER(i,2){
    Nand(R,*Qn,Q);
    Nand(S,*Q,Qn);}}
/*
 * Flip Flop SR asincrono contruido con puertas Nor.
 *
 *          +------+
 *    R ----+      |
 *          | NOR  +--+-- Q
 *        +-+      |  |
 *        | +------+ /
 *         \       /
 *           \   /
 *             X
 *           /   \
 *         /       \
 *        | +------+ \
 *        +-+      |  |
 *          | NOR  +--+-- Qn
 *    S ----+      |
 *          +------+
 *
 *     S R | Q Qn
```

```
 *    -----+-----
 *     0 0 | Q Qn (estado anterior)
 *     0 1 | 0 1  (reset)
 *     1 0 | 1 0  (set)
 *     1 1 | 0 0  (estado indeseable)
 */
void FFSR_Nor(S,R,Q,Qn)
BIT S,R,*Q,*Qn;{
  unsigned i;
  ITER(i,2){
    Nor(R,*Qn,Q);
    Nor(S,*Q,Qn);}}
/*
 * Flip-Flip SR sincrono activo en cuando el reloj
 * esta en alta.
 * Construido con puertas Nand.
 *
 *         +------+
 *   S ----+      |     +------+
 *         | NAND +--R--+      +-- Q
 *      +--+      |     |      |
 *      |  +------+     |      |
 *  ck -+               | FFSR |
 *      |  +------+     |      |
 *      +--+      |     |      |
 *         | NAND +--S--+      +-- Qn
 *   R ----+      |     +------+
 *         +------+
 *
 *     ck S R | Q Qn
 *    --------+------
 *     0  x x | Q Qn (FF deshabilitado)
 *     1  0 0 | Q Qn (estado anterior)
 *     1  0 1 | 0 1  (reset)
 *     1  1 0 | 1 0  (set)
 *     1  1 1 | 1 1  (estado indeseable)
 *
 * Cuando ck=0, y el estado anterior ha sido Q=Qn=1,
 * el FF evoluciona a un estado de reposo (Q=Not(Qn)),
 * que evidentemente no es el estado
 * anterior tal y como se indica en la tabla de verdad.
 */
void FFSR_Nand_Sinc(S,R,ck,Q,Qn)
BIT S,R,ck,*Q,*Qn;{
  BIT a,b;
  Nand(S,ck,&a);
  Nand(R,ck,&b);
  FFSR_Nand(b,a,Q,Qn);}
/*
 * Flip-Flip SR sincrono activo en cuando el reloj
 * esta en alta.
 * Construido con puertas Nor.
 *
 *         +------+
 *   S ----+      |     +------+
 *         | AND  +--S--+      +-- Q
 *      +--+      |     |      |
```

```
 *      |  +------+     |      |
 *  ck -+               | FFSR |
 *      |  +------+     |      |
 *      +--+      |     |      |
 *         | AND  +--R--+      +-- Qn
 *   R ----+      |     +------+
 *         +------+
 *
 *     ck S R | Q Qn
 *    --------+------
 *     0  x x | Q Qn (FF deshabilitado)
 *     1  0 0 | Q Qn (estado anterior)
 *     1  0 1 | 0 1  (reset)
 *     1  1 0 | 1 0  (set)
 *     1  1 1 | 0 0  (estado indeseable)
 *
 * Cuando ck=0, y el estado anterior ha sido Q=Qn=0,
 * el FF evoluciona a un estado de reposo (Q=Not(Qn)),
 * que evidentemente no es el estado
 * anterior tal y como se indica en la tabla de verdad.
 */
void FFSR_Nor_Sinc(S,R,ck,Q,Qn)
BIT S,R,ck,*Q,*Qn;{
  BIT a,b;
  And(S,ck,&a);
  And(R,ck,&b);
  FFSR_Nor(b,a,Q,Qn);}
/*
 * Flip-Flip D asincrono construido con un
 * Flip-Flip SR Nor.
 *
 *             +------+
 *   D ---+--S-+      +- Q
 *        |    |      |
 *       -+-   |      |
 *       \ /   | FFSR |
 *        o    |      |
 *     Dn |    |      |
 *        +--R-+      +- Qn
 *             +------+
 *
 *     D | Q Qn
 *    ---+------
 *     0 | 0 1
 *     1 | 1 0
 */
void FFD(D,Q,Qn)
BIT D,*Q,*Qn;{
  BIT Dn;
  Not(D,&Dn);
  FFSR_Nor(D,Dn,Q,Qn);}
/*
 * Flip-Flip D sincrono activo cuando el reloj
 * esta en alta.
 * Construido con un Flip-Flip SR sincrono a base de
 * puertas Nor.
 *
```

```
 *                +------+
 *   D -+-----S---+      |     +------+
 *      |         | AND  +--R--+      +-- Q
 *      |      +--+      |     |      |
 *     -+-     |  +------+     |      |
 *     \ / ck -+               | FFSR |
 *      O      |  +------+     |      |
 *   Dn |      +--+      |     |      |
 *      |         | AND  +--S--+      +-- Qn
 *      +-----R---+      |     +------+
 *                +------+
 *
 *    ck D | Q Qn
 *    -----+------
 *     0 x | Q Qn
 *     1 0 | 0 1
 *     1 1 | 1 0
 */
void FFD_Sinc(D,ck,Q,Qn)
BIT D,ck,*Q,*Qn;{
  BIT Dn;
  Not(D,&Dn);
  FFSR_Nor_Sinc(D,Dn,ck,Q,Qn);}
/*
 * Flip-Flip JK asincrono.
 * Construido con un Flip-Flip SR asincrono a base de
 * puertas Nor.
 *
 *        +--------------------------+
 *        |   +------+               |
 *        +---+      |     +------+  |
 *            | AND  +--S--+      +--|--+-- Q
 *   J -------+      |     |      |  |  |
 *            +------+     |      |  |  |
 *                         | FFSR |  |  |
 *            +------+     |      |  |  |
 *   K -------+      |     |      |  |  |
 *            | AND  +--R--+      +--+--|-- Qn
 *        +---+      |     +------+     |
 *        |   +------+                  |
 *        +-----------------------------+
 *
 *     J K | Q Qn
 *    -----+------
 *     0 0 | Q Qn
 *     0 1 | 0 1
 *     1 0 | 1 0
 *     1 1 | indeterminado pero coherente (Q=Not(Qn))
 *
 * El valor de la indeterminacion depende del numero de
 * iteraciones del FFJK.
 */
void FFJK(J,K,Q,Qn)
BIT J,K,*Q,*Qn;{
  unsigned i;
  BIT S,R;
  ITER(i,2){
```

```
    And(J,*Qn,&S);
    And(K,*Q,&R);
    FFSR_Nor(S,R,Q,Qn);}}
/*
 * Flip-Flip JK sincrono.
 * Construido con un Flip-Flip SR sincrono a base de
 * puertas Nor.
 *
 *        +--------------------------+
 *        |   +------+               |
 *        +---+      |     +------+  |
 *            | AND  +--S--+      +--|--+-- Q
 *   J -------+      |     |      |  |  |
 *            +------+     |      |  |  |
 *                     ck -+ FFSR |  |  |
 *            +------+     |      |  |  |
 *   K -------+      |     |      |  |  |
 *            | AND  +--R--+      +--+--|-- Qn
 *        +---+      |     +------+     |
 *        |   +------+                  |
 *        +-----------------------------+
 *
 *     ck J K | Q Qn
 *    --------+------
 *     0  x x | Q Qn
 *     1  0 0 | Q Qn
 *     1  0 1 | 0 1
 *     1  1 0 | 1 0
 *     1  1 1 | indeterminado pero coherente (Q=Not(Qn))
 *
 * El valor de la indeterminacion depende del numero de
 * interaciones del FFJK.
 */
void FFJK_Sinc(J,K,ck,Q,Qn)
BIT J,K,ck,*Q,*Qn;{
  unsigned i;
  BIT S,R;
  ITER(i,2){
    And(J,*Qn,&S);
    And(K,*Q,&R);
    FFSR_Nor_Sinc(S,R,ck,Q,Qn);}}
```

Todos los biestables descritos mediante esta serie de funciones basan el contenido de sus salidas en variables definidas externamente a estas funciones. Estas variables representan los terminales físicos conductores del circuito diseñado. Cuando ocurre que en el transcurso de la simulación existen líneas internas al circuitos (no visibles desde el exterior) que deben mantener su estado y que no siempre se almacenan sobre variables activas en ese instante, las variables se declaran del tipo `static` [6] y las funciones deben de convertirse en macros, pues de lo contrario, si invocamos la función en dos puntos distintos del código, estaremos usando las misma variable estática para representar dos líneas conductoras distintas. Esto le ocurre en los biestables implementados en el fichero `ff.h`:

```
/*
 * ff.h
 * Implementacion de los biestables basicos.
```

```
 * gse. 1995.
 */

#define ITER(i,n)       for((i)=0;(i)<(n);(i)++)

/*
 * FFT asincrono.
 * Un Flip-Flip T es un JK con sus entradas
 * T=J=K (cortocircuitadas).
 * Por supuesto, para T=1 este Flip-Flip oscila
 * 'coherentemente'.
 *
 *  T | Q Qn
 * ---+------
 *  0 | Q Qn
 *  1 | indeterminado pero coherente.
 *
 */
#define FFT(T,Q,Qn)     FFJK(T,T,Q,Qn)

/*
 * FFT sincrono.
 * Un Flip-Flip T es un JK con sus entradas T=J=K
 * (cortocircuitadas).
 * Por supuesto, para T=1 este Flip-Flip oscila
 * 'coherentemente'.
 *
 *  ck T | Q Qn
 * ------+------
 *   0 x | Q Qn
 *   1 0 | Q Qn
 *   1 1 | indeterminado pero coherente.
 */
#define FFT_SINC(T,ck,Q,Qn)     FFJK_SINC(T,T,ck,Q,Qn)

/*
 * Flip-Flip Master-Slave SR.
 * Cuando ck=1 funciona el Master y cuando ck=0
 * lo hace el Slave. Debido a esto, las entradas SR,
 * no actuan a la salida hasta que transcurre medio
 * ciclo de reloj.
 *
 *  S R | Q Qn
 * -----+------
 *  0 0 | Q Qn
 *  0 1 | 0 1
 *  1 0 | 1 0
 *  1 1 | 0 1 (estado indeseable)
 */
#define FFSRMS(S,R,ck,Q,Qn) {\
  BIT nck_FFSRMS;\
  static BIT q_FFSRMS,qn_FFSRMS;\
  Not(ck,&nck_FFSRMS);\
  FFSR_Nor_Sinc(S,R,ck,&q_FFSRMS,&qn_FFSRMS);\
  FFSR_Nor_Sinc(q_FFSRMS,qn_FFSRMS,nck_FFSRMS,Q,Qn);}
/*
 * Flip-Flip Master-Slave JK.
```

```
 * Cuando ck=1 funciona el Master y cuando ck=0
 * lo hace el Slave. Debido a esto, las entradas JK,
 * no actuan a la salida hasta que transcurre medio
 * ciclo de reloj.
 *
 *  J K | Q Qn
 * -----+------
 *  0 0 | Q Qn
 *  0 1 | 0 1
 *  1 0 | 1 0
 *  1 1 | Qn Q
 */
#define FFJKMS(J,K,ck,Q,Qn) {\
  unsigned i_FFJKMS;\
  BIT nck_FFJKMS;\
  Not(ck,&nck_FFJKMS);\
  ITER(i_FFJKMS,2) {\
    BIT a_FFJKMS,b_FFJKMS;\
    static BIT q_FFJKMS,qn_FFJKMS;\
    And(J,*(Qn),&a_FFJKMS);\
    And(K,*(Q),&b_FFJKMS);\
    FFSR_Nand_Sinc(a_FFJKMS,b_FFJKMS,ck,&q_FFJKMS,&qn_FFJKMS);\
    FFSR_Nand_Sinc(q_FFJKMS,qn_FFJKMS,nck_FFJKMS,Q,Qn);}}
/*
 * Flip-Flip Master-Slave D.
 * Cuando ck=1 funciona el Master y cuando ck=0
 * lo hace el Slave. Debido a esto, la entrada D,
 * no afecta a la salida hasta que
 * transcurre medio ciclo de reloj.
 *
 *  D | Q Qn
 * ---+------
 *  0 | 0 1
 *  1 | 1 0
 */
#define FFDMS(D,ck,Q,Qn) {\
  BIT Dn_FFDMS;\
  Not(D,&Dn_FFDMS);\
  FFSRMS(D,Dn_FFDMS,ck,Q,Qn)}
/*
 * Flip-Flip Master-Slave T.
 * Cuando ck=1 funciona el Master y cuando ck=0
 * lo hace el Slave. Debido a esto, la entrada T,
 * no afecta a la salida hasta que
 * transcurre medio ciclo de reloj.
 *
 *  T | Q Qn
 * ---+------
 *  0 | Q Qn
 *  1 | Qn Q
 */
#define FFTMS(T,ck,Q,Qn)        FFJKMS(T,T,ck,Q,Qn)
```

Las macros son expandidas por el programa preprocesador `cpp` [6]. Las variables que sirven de argumentos a la macro deben de llamarse de forma distinta a las variables locales de la macro (las que se declaran dentro), porque de lo contrario

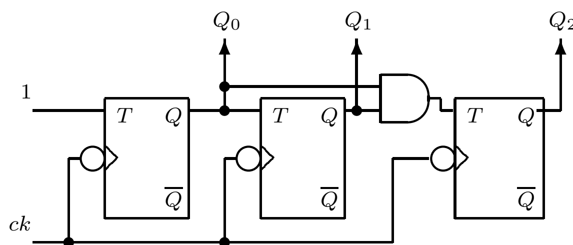
el compilador no las distingue. Este es el principal inconveniente del uso de las macros y para soslayarlo las variables internas se declaran con un sufijo igual al nombre de la macro. Otra consecuencia de usar macros en vez de funciones es que el código ejecutable generado en la compilación es más largo, pero también es más rápido puesto que no se producen las llamadas a las funciones.

# 8 Simulación de un contador binario síncrono módulo 8.

Un contador binario es un circuito secuencial que posee $log_2(N)$ flip-flops donde $N$ es el número de estados [7] (en nuestro caso $N = 8$, por lo que necesitamos 3 biestables). Su regla de diseño es la siguiente:

> *Conmutar el flip-flop i-ésimo cuando los $i-1$ flip-flops de peso inferior (que conmutan con más frecuencia) cumplan la ecuación:*
>
> $$Q_{i-1} = Q_{i-2} = \ldots = Q_0 = 1$$

Esto se puede hacer con biestables T y puertas AND. Nuestro contador módulo 8 presenta el siguiente diagrama lógico:




El contador se describe como una macro en el fichero `conygen.h`:

```
/*
 * conygen.h
 * Contadores y generadores de secuencias.
 * gse. 1995.
 */

/*
 * Contador binario modulo 8.
 * Digital Logic and Computer Desig.
 * M. Morris Mano. pag 245.
 */
#define Contador8(cuenta,ck,Q2,Q1,Q0) {\
  static BIT Contador8_Q0n,\
  Contador8_Q1n,Contador8_Q2n;\
  BIT Contador8_T2;\
  FFTMS(cuenta,ck,Q0,&Contador8_Q0n)\
  FFTMS(*(Q0),ck,Q1,&Contador8_Q1n)\
  And(*(Q0),*Q1,&Contador8_T2);\
  FFTMS(Contador8_T2,ck,Q2,&Contador8_Q2n)}
```

```
/*
 * Contador binario modulo 16.
 */
#define Contador16(cuenta,ck,Q3,Q2,Q1,Q0){\
  static BIT Contador16_Q0n,Contador16_Q1n,Contador16_Q2n,Contador16_Q3n;\
  BIT Contador16_T2,Contador16_T3;\
  FFTMS(cuenta,ck,Q0,&Contador16_Q0n)\
  FFTMS(*(Q0),ck,Q1,&Contador16_Q1n)\
  And(*(Q0),*(Q1),&Contador16_T2);\
  FFTMS(Contador16_T2,ck,Q2,&Contador16_Q2n)\
  And(Contador16_T2,*(Q2),&Contador16_T3);\
  FFTMS(Contador16_T3,ck,Q3,&Contador16_Q3n)}
```

El programa de simulación es tan sencillo como el siguiente:

```
/*
 * contador.c
 * Simulacion de un contador modulo 8.
 * Digital Logic and Computer Design.
 * M. Morris Mano. pag 245.
 * gse. 1995.
 */

#include <stdio.h>
#include "defs.h"
#include "lib.h"
#include "ff.h"
#include "conygen.h"

int main() {
  char tecla;
  BIT cuenta,Q0,Q1,Q2,ck;
  Q0.time=Q1.time=Q2.time=0;
  printf("Simulacion de un contador modulo 8.\n");
  printf("+-----+----------+----------+----------+\n");
  printf("| ck  | Q2       | Q1       | Q0       |\n");
  printf("+-----+----------+----------+----------+\n");
  printf("| bit | bit time | bit time | bit time |\n");
  printf("+-----+----------+----------+----------+\n");
  for(tecla=getc(stdin);tecla!='q';tecla=getc(stdin)) {
    Clock(&ck);
    cuenta.bit=1;
    cuenta.time=0;
    Contador8(cuenta,ck,&Q2,&Q1,&Q0);
    printf("| %d   | %d %6d | %d %6d | %d %6d |\n",
    ck.bit,
    Q2.bit,Q2.time,Q1.bit,Q1.time,Q0.bit,Q0.time);}
  return(0);}
```

A continuación se muestra el resultado de una simulación:

```
Simulacion de un contador modulo 8.
+-----+----------+----------+----------+
| ck  | Q2       | Q1       | Q0       |
+-----+----------+----------+----------+
| bit | bit time | bit time | bit time |
```

```
+-----+----------+----------+----------+
| 0   | 1    236 | 1    156 | 1     78 |
| 1   | 1    312 | 1    232 | 1    154 |
| 0   | 1    388 | 1    308 | 1    230 |
| 1   | 1    464 | 1    384 | 1    306 |
| 0   | 1    540 | 1    460 | 1    382 |
| 1   | 1    616 | 1    536 | 1    458 |
| 0   | 1    692 | 1    612 | 1    534 |
| 1   | 1    768 | 1    688 | 1    610 |
| 0   | 1    844 | 1    764 | 1    686 |
| 1   | 1    920 | 1    840 | 1    762 |
| 0   | 1    996 | 1    916 | 1    838 |
| 1   | 1   1072 | 1    992 | 1    914 |
| 0   | 1   1148 | 1   1068 | 1    990 |
| 1   | 1   1224 | 1   1144 | 1   1066 |
| 0   | 1   1300 | 1   1220 | 1   1142 |
| 1   | 1   1376 | 1   1296 | 1   1218 |
| 0   | 1   1452 | 1   1372 | 1   1294 |
| 1   | 1   1528 | 1   1448 | 1   1370 |
```

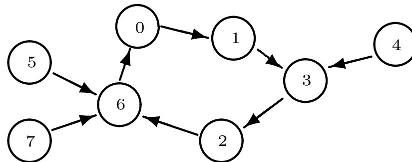
Las cuatro columnas principales representan las cuatro variables lógicas que estamos simulando: $ck$, $Q_2$, $Q_1$ y $Q_0$ (el reloj y las salidas de los cuatro biestables). Bajo la columna etiquetada con `bit`, aparecen los valores lógicos, y bajo la columna `time` los tiempos de respuesta acumulados a través de todo el circuito, para esa iteración. Por ejemplo, el biestable $Q_0$ necesita 19 pasos en generar su bit de salida. Esto quiere decir que necesita evaluar 19 puertas para construirlo. A este tiempo se le llama tiempo de conmutación o tiempo de propagación del biestable. Para el biestable $Q_1$ la simulación indica 38 pasos, que es la acumulación de sus 19 pasos y los 19 del biestable que le precede. Su tiempo de conmutación sería realmente 38 si en contador fuese asíncrono (estos circuitos se llaman contadores de rizado [7]), pero como en un contador síncrono todos los biestables trabajan en paralelo, debemos de restar 38-19=19 pasos, para encontrar el tiempo de conmutación de $Q_2$. La simulación indica un tiempo de conmutación de 58 pasos para $Q_3$, cosa que sería válida para un contador de rizado. Igual que para $Q_2$, para encontrar el tiempo de conmutación en el caso síncrono, necesitamos restar el tiempo acumulado en $Q_1$, teniendo 58-38=20 pasos. Por tanto, para conmutar $Q_2$ necesitamos un ciclo más. Esto es debido a que existe una puerta AND intercalada entre $Q_1$ y $Q_2$.

Para averiguar el periodo mínimo (o la frecuencia máxima) de la señal de reloj, necesitamos averiguar el tiempo máximo de excitación de cualquier etapa del contador que funcione simultáneamente. En paralelo trabajan el biestable $Q_0$, el biestable $Q_1$ y el circuito biestable $Q_2$ junto con la puerta AND. Ya hemos averiguado que necesitamos 20 pasos para esta última etapa. Por tanto, el periodo del reloj debe de ser de 20 pasos.

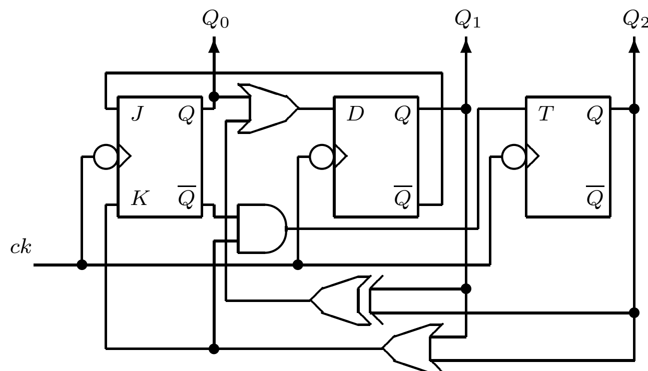
En la siguiente iteración, se muestran los tiempos acumulados a los de la anterior y podemos ver que se trata de 20 pasos.

# 9 Simulación de un generador de secuencias.

Un generador de secuencias [1] es un circuito secuencial que sigue una serie de estados predeterminados. Estos se describen con un diagrama de flujo o con una tabla de transiciones. El diagrama de flujo del generador que vamos a simular es el siguiente:




El circuito secuencial que pasa por estos estados tiene el siguiente diagrama lógico:




Circuito que se implementaría en lenguaje C de la siguiente forma:

```
/*
 * generador.c
 * Simulacion de un generador de secuencia.
 * Problemas practicos de diseno logico.
 * Gascon de Toro. pag. 350.
 * gse. 1995.
 */

#include <stdio.h>
#include "defs.h"
#include "lib.h"
#include "ff.h"

int main() {
  char tecla;
  printf("Simulacion de un generador de secuencia.\n");
  printf("Tabla de transiciones:\n");
  printf("\n");
  printf("Est Act | Est Sig\n");
  printf("---------+---------\n");
  printf("Q2 Q1 Q0 | Q2 Q1 Q0\n");
  printf("---------+---------\n");
```

```
  printf(" 0  0  0 |  0  0  1\n");
  printf(" 0  0  1 |  0  1  1\n");
  printf(" 0  1  0 |  1  1  0\n");
  printf(" 0  1  1 |  0  1  0\n");
  printf(" 1  0  0 |  0  1  1\n");
  printf(" 1  0  1 |  1  1  0\n");
  printf(" 1  1  0 |  0  0  0\n");
  printf(" 1  1  1 |  1  1  0\n\n");
  printf("+-----+----------+----------+----------+\n");
  printf("| ck  | Q2       | Q1       | Q0       |\n");
  printf("+-----+----------+----------+----------+\n");
  printf("| bit | bit time | bit time | bit time |\n");
  printf("+-----+----------+----------+----------+\n");
  for(tecla=getc(stdin);tecla!='q';tecla=getc(stdin)) {
    unsigned i;
    static BIT Q0,Q0n,Q1,Q1n,Q2,Q2n,ck,a,K0,D1,T2;
    Clock(&ck);
    for(i=0;i<2;i++) {
      Or(Q0,a,&D1);
      And(Q0n,K0,&T2);
      Or(Q1,Q2,&K0);
      Xor(Q1,Q2,&a);
      FFJKMS(Q1n,K0,ck,&Q0,&Q0n)
      FFDMS(D1,ck,&Q1,&Q1n)
      FFTMS(T2,ck,&Q2,&Q2n)}
    printf("| %d   | %d %6d | %d %6d | %d %6d |\n",
    ck.bit,Q2.bit,Q2.time,Q1.bit,Q1.time,Q0.bit,Q0.time);}
  return 0;}
```

Se trata de un circuito síncrono activo en el flanco de bajada (↓). En este circuito se introduce una novedad respecto del contador visto: las realimentaciones. Estas repercuten en la simulación puesto que un estado (no válido) alcanzado tras una iteración, puede ser relevante para alcanzar el siguiente estado (que si sea válido). Este efecto se consigue iterando dos veces el generador por cada cambio en la señal de reloj. La salida correspondiente a la simulación del generador se presenta a continuación:

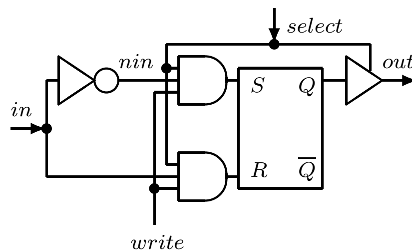
```
Simulacion de un generador de secuencia.
Tabla de transiciones:

 Est Act | Est Sig
---------+---------
Q2 Q1 Q0 | Q2 Q1 Q0
---------+---------
 0  0  0 |  0  0  1
 0  0  1 |  0  1  1
 0  1  0 |  1  1  0
 0  1  1 |  0  1  0
 1  0  0 |  0  1  1
 1  0  1 |  1  1  0
 1  1  0 |  0  0  0
 1  1  1 |  1  1  0

+-----+----------+----------+----------+
| ck  | Q2       | Q1       | Q0       |
+-----+----------+----------+----------+
| bit | bit time | bit time | bit time |
+-----+----------+----------+----------+
```

```
| 1    | 1    164 | 1    115 | 1    159 |
| 0    | 1    326 | 1    277 | 1    321 |
| 1    | 1    488 | 1    439 | 1    483 |
| 0    | 1    650 | 1    601 | 1    645 |
| 1    | 1    812 | 1    763 | 1    807 |
| 0    | 1    974 | 1    925 | 1    969 |
| 1    | 1   1136 | 1   1087 | 1   1131 |
| 0    | 1   1298 | 1   1249 | 1   1293 |
| 1    | 1   1460 | 1   1411 | 1   1455 |
| 0    | 1   1622 | 1   1573 | 1   1617 |
| 1    | 1   1784 | 1   1735 | 1   1779 |
| 0    | 1   1946 | 1   1897 | 1   1941 |
| 1    | 1   2108 | 1   2059 | 1   2103 |
| 0    | 1   2270 | 1   2221 | 1   2265 |
| 1    | 1   2432 | 1   2383 | 1   2427 |
| 0    | 1   2594 | 1   2545 | 1   2589 |
```

Puede verse que los tiempos de generación de los bits a la salida de los biestables es mayor que el caso del contador, que no están ordenados y que dependen de la iteración. Esto es debido a las realimentaciones del circuito (que se simulan mediante dos iteraciones por cada cambio en la señal de reloj). Es más complejo encontrar el mínimo periodo que puede suministrarse con el reloj, que se calcula averiguando la etapa más lenta que funcione en paralelo con las demás. Para ello debemos mirar en el esquema lógico suministrado anteriormente para el generador. Encontramos que el biestable $Q_2$ tiene dos niveles de puerta a su entrada $T$. Por tanto el periodo del reloj es 1+1+19 pasos, donde 19 es el tiempo de excitación del biestable $T$. Los tiempos asociados a cualquier biestable pueden simularse ejecutando el programa `tff`, que los testea.

# 10 Simulación de una memoria SRAM.

La creación de una memoria RAM estática comienza con el diseño de su celda básica, que es capaz de memorizar un bit de información [4, 2]. La celda está basada en un biestable SR asíncrono junto con una circuitería auxiliar que acondiciona al biestable para ser leído o escrito. El diagrama lógico de una celda básica se expone a continuación:




Una memoria SRAM es una matriz rectangular de celdas básicas junto a una lógica de decodificación de direcciones (un decodificador). La memoria normalmente se organiza en bloques de 8 celdas, formando bytes de memoria. Esta

agrupación de celdas es habilitada para su lectura o escritura por una misma línea de selección, conectada a la entrada *select* de las celdas básicas. Dicha línea de selección es una de las salidas del decodificador de direcciones, que deberá tener tantas salidas como bytes existen en la memoria. La celda básica de memoria construida responde a la siguiente tabla de verdad:

| *select* | *write* | $Q^{n+1}$ | *out* |
|---|---|---|---|
| 0 | 0 | $Q^n$ | Z |
| 0 | 1 | $Q^n$ | Z |
| 1 | 0 | $Q^n$ | $Q^n$ |
| 1 | 1 | *in* | *in* |

El superíndice usado indica el instante de tiempo. La razón por la que se han contemplado salidas triestado, que son capaces de presentar una alta impedancia (Z), es la de poder conectar muchos bits a una misma línea del bus de salida, y en definitiva, muchas memorias al mismo bus [2]. De no existir este estado, necesitaríamos puertas OR con tantas entradas como palabras (en este son caso palabras de 8 bits) diferentes podemos escribir sobre el bus. El fichero que implementa una memoria SRAM de 8x8 bits se llama `ram.h`:

```
/*
 * ram.h
 * Memorias RAM.
 * gse. 1995.
 */

/*
 * Celda basica de memoria estatica.
 * pag. 295. Morris Mano.
 */
#define BIT_SRAM(in,select,write,out) {\
  BIT nin_BIT_SRAM,S_BIT_SRAM,R_BIT_SRAM,o_BIT_SRAM;\
  static BIT Q_BIT_SRAM,Qn_BIT_SRAM;\
  Not(in,&nin_BIT_SRAM);\
  And3(select,nin_BIT_SRAM,write,&R_BIT_SRAM);\
  And3(select,in,write,&S_BIT_SRAM);\
  FFSR_Nor(S_BIT_SRAM,R_BIT_SRAM,&Q_BIT_SRAM,&Qn_BIT_SRAM);\
    printf("%d:%d ",Q_BIT_SRAM.bit,Q_BIT_SRAM.time);\
  Q_BIT_SRAM.time=Qn_BIT_SRAM.time=0;\
  Buffer(Q_BIT_SRAM,select,out);}

/*
 * Un byte de SRAM es un array de 8 celdas basicas.
 */
#define BYTE_SRAM(in,select,write,out) {\
  BIT_SRAM(in[0],select,write,out[0])\
  BIT_SRAM(in[1],select,write,out[1])\
  BIT_SRAM(in[2],select,write,out[2])\
  BIT_SRAM(in[3],select,write,out[3])\
  BIT_SRAM(in[4],select,write,out[4])\
  BIT_SRAM(in[5],select,write,out[5])\
  BIT_SRAM(in[6],select,write,out[6])\
  BIT_SRAM(in[7],select,write,out[7]);printf("\n");}

/*
```

```
 * RAM estatica de 8x8 bits.
 *
 * enable:    Habilita la memoria para su uso.
 *            Un 0 provoca que la salida esta en
 *            alta impedancia (Z).
 * write:     Indica una operacion de escritura.
 * direccion: Bus de direcciones que designa la
 *            direccion a leer o escribir.
 * in:        Bus de datos de entrada de 8 bits.
 * out:       Bus de datos de salida de 8 bits.
 */
#define SRAM_8x8(enable,write,direccion,in,out) { \
  unsigned i_SRAM_8x8;\
  BIT palabra_SRAM_8x8[8];\
  for(i_SRAM_8x8=0;i_SRAM_8x8<8;i_SRAM_8x8++)\
    {(out[i_SRAM_8x8])->bit=Z;(out[i_SRAM_8x8])->time=0;}\
  Decod8(direccion,enable,palabra_SRAM_8x8);\
  BYTE_SRAM(in,palabra_SRAM_8x8[0],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[1],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[2],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[3],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[4],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[5],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[6],write,out)\
  BYTE_SRAM(in,palabra_SRAM_8x8[7],write,out)}
```

Como puede verse, existen macros que declaran la celda básica, una palabra de memoria de 8 bits, y la memoria total de 8x8 bits. El programa de simulación de la memoria es `tram.c`:

```
/*
 * tram.c
 * Simulacion de una memoria RAM statica.
 * Capacidad: 8 bytes.
 * gse. 1995.
 */

#include <stdio.h>
#include "defs.h"
#include "lib.h"
#include "ff.h"
#include "ram.h"

int main() {
  char tecla;
  printf("Simulando una memoria SRAM de 8x8 bits.\n");
  for(;tecla!='q';tecla=getc(stdin)) {
    BIT enable,write,address[3],in[8],out[8];
    address[0]=RandomBIT();
    address[1]=RandomBIT();
    address[2]=RandomBIT();
    write=RandomBIT();
    enable=RandomBIT();
    in[0]=RandomBIT();
    in[1]=RandomBIT();
    in[2]=RandomBIT();
    in[3]=RandomBIT();
    in[4]=RandomBIT();
```

```
    in[5]=RandomBIT();
    in[6]=RandomBIT();
    in[7]=RandomBIT();
    SRAM_8x8(enable,write,address,in,&out)
    printf("enable write address    in      out\n");
    printf("------ ----- ------- -------- --------\n");
    printf("     %d     %d     %d%d%d %d%d%d%d%d%d%d%d \
%d%d%d%d%d%d%d%d\n",
    enable.bit,write.bit,
    address[0].bit,address[1].bit,address[2].bit,
    in[0].bit,in[1].bit,in[2].bit,in[3].bit,
    in[4].bit,in[5].bit,in[6].bit,in[7].bit,
    out[0].bit,out[1].bit,out[2].bit,out[3].bit,
    out[4].bit,out[5].bit,out[6].bit,out[7].bit);
    printf("%6d %5d %7d %8d %8d\n",enable.time,write.time,
    address[0].time,in[0].time,out[0].time);}
  return 0;}
```

La ejecución del anterior programa genera la salida:

```
Simulando una memoria SRAM de 8x8 bits.
134479872 926154752 892403712 808714240 825688064 943063040 909115392 825294848 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     0     010 00100110 22222222
     0     0       0        0        0
134479872 926154752 892403712 808714240 825688064 943063040 909115392 825294848 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     0     000 00010100 22222222
     0     0       0        0        0
0 0 0 1 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     1     0     110 10100100 22222222
     0     0       0        0        0
```

```
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     1     001 01011000 22222222
     0     0       0        0        0
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     1     111 01001001 22222222
     0     0       0        0        0
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     1     000 11010010 22222222
     0     0       0        0        0
0 0 0 1 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 0:27 0:27 1:27 1:27 0:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     1     1     110 11100110 22222222
     0     0       0        0        0
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 0:27 0:27 1:27 1:27 0:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
```

```
------ ----- ------- -------- --------
     0     1     001 11001110 22222222
     0     0       0        0        0
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 0:27 0:27 1:27 1:27 0:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     0     000 01001101 22222222
     0     0       0        0        0
0 0 0 0 0 0 0 0 1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 1:27 1:27 1:27 1:27 1:27
1:27 1:27 1:27 0:27 0:27 1:27 1:27 0:27
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
1:26 1:26 1:26 1:26 1:26 1:26 1:26 1:26
enable write address    in      out
------ ----- ------- -------- --------
     0     0     000 01010010 22222222
     0     0       0        0        0
```

Se muestra el contenido de la RAM, su temporización, y cuales son las entradas y las salidas de la memoria. Inicialmente, toda la memoria se encuentra inicializada a 1, pues este es el estado inicial del biestable NOR usado para la construcción de la celda básica. Puede comprobarse que cada una de las palabras de la memoria sigue la tabla de verdad asociada a una celda básica. El cálculo de los tiempos de escritura y lectura se puede también deducir:

1. **Tiempo de escritura**: es el número de pasos gastados en escribir un byte en cualquier dirección de la memoria. Para ello debemos habilitarla ($enable = 1$), indicar su escritura ($write = 1$), imponer una dirección de memoria e insertar un byte en el bus de entrada de datos. Si miramos el esquema lógico de la celda básica, podemos ver que el tiempo de escritura está almacenado en la salida de los biestables que forman cada celda de memoria. Este es el tiempo que se presenta junto con el contenido de cada celda. Puede observarse que no es contante, pues el tiempo gastado en escribir la dirección 7 es 6 pasos, mientras que necesitamos 7 para el resto. Esto es consecuencia de que el decodificador de direcciones no genera todas sus salidas con el mismo tiempo. Como tiempo de escritura escogemos el peor de los casos (y que también es el más frecuente), esto es, 7 pasos.

2. **Tiempo de lectura**: se obtiene directamente observando los tiempos acumulados en los bits del bus de salida de datos. También es variable en función del decodificador de direcciones. Si hemos de escoger uno, será el

superior (4 pasos), que es el tiempo de lectura gastado en leer en cualquier dirección menos en la más alta, que es 3 pasos.

## 11 Conclusión y trabajo futuro.

Evidentemente, un simulador digital lógico es una herramienta útil en el diseño y testeo de sistemas digitales. El procedimiento presentado permite, con unos requerimientos de hardware y software muy básicos, construir sistemas tan grandes como el mayor programa que seamos capaz de compilar y ejecutar. SDLC puede se puede utilizar a muchos niveles. Puede ser usado para comprobar ejercicios, para diseñar prácticas en una asignatura de electrónica digital, en la que se estudien circuitos de la complejidad expuesta en este trabajo, o para construir sistemas más grandes y complejos, que pueden ir desde implementaciones hardware de algoritmos software (por ejemplo, un compresor de datos) hasta el diseño de una CPU, que ejecute su propio repertorio de instrucciones.

## Referencias


[1] M. Gascón de Toro. *Problemas Prácticos de Diseño Lógico.* Paraninfo, 1990.

[2] David A. Patterson John L. Hennessy. *Organización y Diseño de Computadores.* McGraw-Hill, 1995.

[3] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1981.

[4] M. Morris Mano. *Digital Logic and Computer Design.* Prentice-Hall, 1979.

[5] Donald E. Thomas Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[6] Brian W. Kernighan Dennis M. Ritchie. *El Lenguaje de Programamción C.* Prentice-Hall, 1991.

[7] Herbert Taub. *Circuitos Digitales y Microprocesadores.* McGraw-Hill, 1983.