SIMULADOR DIGITAL EN LENGUAJE C: SDLC

V. G. Ruiz, J. A. Martínez, I. García
Depto de Arquitectura de Computadores y Electrónica
Universidad de Almería
Tel: 950 215074
Fax: 950 215070
e-mail: vruiz@iron.ualm.es
e-mail: jamartin@iron.ualm.es
e-mail: inma@iron.ualm.es

RESUMEN.- Este trabajo presenta un software educativo de utilidad práctica en laboratorios de enseñanza de sistemas digitales y arquitectura de computadores. SDLC permite simular tanto el comportamiento lógico como su desarrollo temporal. Se ha desarrollado una biblioteca de funciones que simulan dispositivos de diferentes complejidades, incluyendo desde las sencillas puertas NOT, AND, OR, etc, hasta una memoria o una unidad aritmético lógica. Todo ello hace que con SDLC se puedan simular desde los más sencillos problemas de diseño hasta el ensamblaje de una pequeña computadora. Una de las principales virtudes de este software educativo es su escaso costo y gran flexibilidad, ya que los únicos requerimientos necesarios para usarlo es un ordenador y un compilador de lenguaje C. Por último, con SDLC el alumno aprende a construir un simulador digital.

1.- SIMULANDO HARDWARE PARALELO CON SOFTWARE SECUENCIAL

La simulación de un sistema digital se realiza actualmente con éxito con lenguajes de programación paralelos como Verilog. En Verilog los circuitos básicos se describen en diferentes módulos que se interpretan en tiempo compartido. Esta última característica fuerza al sistema operativo de la computadora utilizada a permitir el procesamiento pseudo-concurrente (el procesamiento realmente paralelo es imposible si no disponemos de varios procesadores). Por el contrario, los lenguajes de programación secuenciales no precisan que el sistema operativo permita la multitarea y por eso están más implantados. Quizás de entre todos ellos el lenguaje C sea el más extendido y por esa razón ha sido el elegido como herramienta de diseño del simulador de dispositivos electrónicos digitales que se presenta en este trabajo.

La no concurrencia de los lenguajes secuenciales frente a los paralelos fuerza a que el problema sea enfocado desde un punto de vista alternativo: la programación orientada al bit. Una señal lógica se crea en alguna parte del sistema digital y termina mezclándose con otras señales para formar otras diferentes. Por lo tanto, si seguimos el rastro a cada uno de los bits que circulan por el sistema, es posible averiguar sus estados y temporizar cuando se producen los cambios de estado. Esta tarea puede realizarse secuencialmente. En C 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 procesa (de su procesador), y esto es justamente lo que un programa en C puede hacer muy eficientemente. El comportamiento paralelo del sistema simulado va a ser obtenido mediante la acumulación de numerosas ejecuciones secuenciales, a las que llamaremos iteraciones. En cada una de las cuales, el sistema evolucionará desde un estado inicial hasta otro final (instante en el que conoceremos 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 LOGICAS Y SU TRATAMIENTO EN C

Las señales digitales presentan dos estados: 0 o 1, bajo o alto, Vcc o GND. Como primera decisión de diseño del modelo de simulación debemos definir como va a ser el tipo de dato que soporte un bit de información: la estructura BIT. 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 Algebra de Boole [4,7]; AND, OR, XOR, etc, por lo que la simulación de las puertas lógicas 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. Se puede considerar otro tipo de dato alternativo como el short int, que se almacena en 16 bits. La ventaja del uso de datos definidos como short int básicamente se traduce en una visualización más adecuada durante la depuración de los programas de simulación. Utilizando variables tipo char la visualización de los estados lógicos 0 y 1 se corresponde con los símbolos "^@" y "^A" respectivamente.

3.- EL TIEMPO DE SIMULACION

Con el objetivo de conocer que circuito digital es más rápido que otro, es necesario simular el tiempo de excitación. Por esta razón se asocia a cada bit un tiempo de simulación que dependerá de los circuitos que ha atravesado ese bit. La temporización puede realizarse con tiempo real (segundos, nanosegundos, etc) o con número de pasos o tics. La primera solución tiene la ventaja de mostrarnos en todo momento el tiempo real transcurrido desde que un bit se ha creado, pero carga el proceso de simulación con computación en punto flotante innecesaria. Por otra parte, si medimos el tiempo transcurrido en pasos, estos pueden ser acumulados en una variable entera (unsigned long, que en todos los sistemas dispone al menos de 32 bits de precisión). Así, para calcular el tiempo real, sólo debemos de conocer cuantos pasos se han producido y a cuanto tiempo corresponde un paso.

Por tanto, la estructura BIT va a constar de dos campos: el primero albergará el valor del bit en ese instante y el segundo almacenará el tiempo de vida del bit hasta ese instante de simulación.

4.- LAS PUERTAS BASICAS

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 los ficheros de biblioteca.

Para simplificar el análisis del sistema digital simulado se ha decidido asignar un tiempo de simulación de un paso a cada una de las puertas. De esta forma, un número de pasos acumulados en una estructura 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 ha de pasar por un número mayor de puertas antes de alcanzar un terminal de 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 los circuitos diseñados.

6.- LOS CIRCUITOS COMBINACIONALES BASICOS

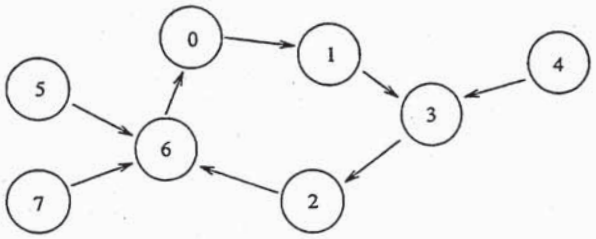
Son bastante sencillos de implementar en SDLC a partir de la biblioteca de puertas lógicas puesto que implementan funciones lógicas sencillas. Se trata de codificadores, multiplexores, sumadores, incluso una pequeña ALU. Todos estos circuitos se distinguen de los secuenciales por no presentar ninguna realimentación desde su salida hacia su entrada. Además, todos se simulan en una iteración.

7.- ELEMENTOS SIMPLES DE MEMORIA: LOS BIESTABLES

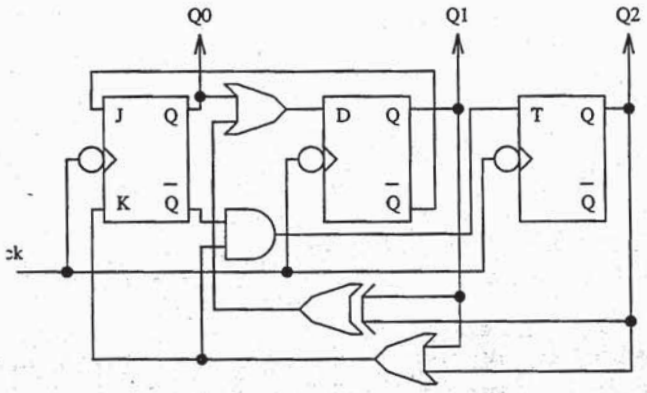
Un biestable es un dispositivo digital que memoriza un bit, y las diferentes formas que existen de realizar esto dan lugar a la gama de biestables conocidos. La simulación de un biestable introduce un grado de dificultad respecto del circuito combinacional: las realimentaciones. Debido a que usamos un lenguaje de programación secuencial, no es posible en una sola pasada (una iteración) simular un biestable, pues su salida depende de su primera pasada. Es necesario el uso de un lazo para volver a calcular las salidas en función de las entradas y las salidas en el instante anterior. Otra característica de los biestables es que deben de almacenar el bit de información entre diferentes iteraciones. Por esta razón las variables que contienen el estado del biestable se declaran de tipo static y las implementaciones se efectúan mediante macros que el preprocesador de C expandirá antes de la compilación.

8.- UN EJEMPLO EN SDLC: SIMULACION DE UN GENERADOR DE SECUENCIAS

Un generador de secuencias 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 SDLC de la siguiente forma:

```
/*
 * generador.c
 * Simulacion de un generador de secuencia.
 * Problemas practicos de diseño 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 éste 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    40  | 1    28  | 1    39  |
| 0   | 1    80  | 1    68  | 0    79  |
| 1   | 1   120  | 1   108  | 0   119  |
| 0   | 0   160  | 0   148  | 0   159  |
| 1   | 0   200  | 0   188  | 0   199  |
| 0   | 0   240  | 0   228  | 1   239  |
| 1   | 0   280  | 0   268  | 1   279  |
| 0   | 0   320  | 1   308  | 1   319  |
| 1   | 0   360  | 1   348  | 1   359  |
| 0   | 0   400  | 1   388  | 0   399  |
| 1   | 0   440  | 1   428  | 0   439  |
| 0   | 1   480  | 1   468  | 0   479  |
| 1   | 1   520  | 1   508  | 0   519  |
| 0   | 0   560  | 0   548  | 0   559  |
| 1   | 0   600  | 0   588  | 0   599  |
| 0   | 0   640  | 0   628  | 1   639  |
```

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 Q2 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.

9.- CONCLUSION 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 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.

10.- ACCESO A SDLC

Los ficheros con el código fuente de las 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. Tras descomprimir el fichero sdlc.tar.gz, el fichero sd.tar se desglosa en el directorio ./sdlc. El compilador utilizado deberá soportar como mínimo el estandar ANSI C. Esta es una descripción de los ficheros más relevantes (para mayor información leer el fichero LEEME). **defs.h**: define los estados lógicos de una línea física y el tipo de dato BIT. **lib.c**, **lib.h**: implementa un reloj y un generador de bits aleatorio. **puertas.c**, **puertas.h**: implementa las puertas combinacionales básicas (and, or, xor, etc) y un buffer triestado (salida en alta impedancia). **ccb.c**, **ccb.h**: describe los circuitos combinacionales básicos (decodificadores, multiplexores, etc). **ff.c**, **ff.h**: implementa los elementos de estado básicos (flip-flops). **conygen.h**: describe las macros que crean contadores y generadores de secuencias. **ram.h**: implementa las memorias RAM. **tpuertas.c**: simula las puertas básicas. **tccb.c**: simula los circuitos combinacionales básicos. **tff.c**: simula los elementos de estado básicos. **contador.c**: simula un contador en binario natural módulo 8. **contador16.c**: simula un contador en binario natural módulo 16. **generador.c**: simula un generador de secuencias. **tram.c**: simula una memoria RAM estática de 8x8 bits.

11.- 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 Programación C". Prentice-Hall, 1991.

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