Computadores Avanzados

Manual de Verilog

Antonio Morán Muñoz

Manual de Verilog

Escuela superior de ingeniería informática

Diseño Digital

Manual de Verilog

Antonio Morán Muñoz
Instituto de Investigación en Informática (I3A)

Esta publicación está basada en la plantilla NIST Handbook de Overleaf

Septiembre 2023

Departamento de Sistemas Informáticos

Universidad de Castilla-La Mancha

National Institute of Standards and Technology Handbook XXXX
Natl. Inst. Stand. Technol. Handbook XXXX, 10 pages (Month Year)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.HB.XXXX

Foreword

Delete if not applicable

Preface

Delete if not applicable

Abstract

Required

Key words

Required, alphabetized, separated by semicolon, and end in a period.

Table of Contents

List of Tables

List of Figures

Glossary

Delete if not applicable

1. Conmutación Segmentada

1.1 Objetivo

Asimilar el funcionamiento de wormhole y comprender la utilidad de incluir canales virtuales en el diseño de las redes de interconexión.

1.2 Desarrollo

La práctica consiste en resolver una serie de cuestiones relacionadas con la técnica de conmutación wormhole. Para realizar esta práctica se hará uso del simulador SimuRed, descrito en el apéndice A. Debéis poner a cero los paquetes iniciales de precalentamiento o de relleno, para que obtenga adecuadamente las estadísticas.

a) **Para comprobar la manera en la que se produce la transmisión de datos cuando se aplica wormhole, selecciona una topología toro 2D 4 × 4, con buffers con capacidad para 2 flits, paquetes de 3 flits (incluida la cabecera), retardos de 1 ciclo, encaminamiento determinista y un canal virtual; y simula la transferencia de 2 paquetes: uno enviado desde el nodo 0 al nodo 5 y otro enviado desde el nodo 2 también al nodo 5. Ambos envíos comienzan en el ciclo 0.**

   La única complejidad en este apartado es la de hacer una traza que recoja los paquetes que nos piden. Para ello, será necesario crear un fichero de texto con la extensión `.trc` y en el incluir una línea por cada paquete que se quiera enviar (en el manual del simulador se explican con más detalle los campos que incluir en cada línea). El Listing 1 muestra el resultado.

Listing 1. Traza a simular.

```
0  0  5
0  2  5
```

   Durante la ejecución de la traza paso a paso, es interesante ver lo que ocurre cuando ambos paquetes compiten por los recursos en el nodo 1. En este caso, ante la colisión, el paquete proveniente del nodo 0 pasa primero, flit a flit. Además, cuando dicho paquete termina de salir de este nodo, el segundo paquete le sigue inmediatamente, pudiéndose apreciar la principal diferencia con *Virtual Cut-Through*, ya que con esta técnica de conmutación, el paquete entero se bloquearía en el buffer del nodo 1 hasta que los buffers de entrada del nodo 5 estuviesen completamente libres, ya que la conmutación se realiza a nivel de paquete.

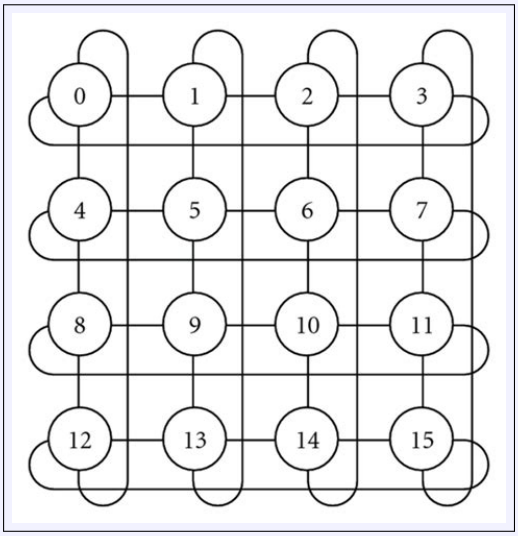
Wormhole

Es una técnica de conmutación en la que el control de flujo se ejerce sobre cada uno de los flits en los que se divide el mensaje, al contrario que en el *Virtual Cut-Through*, en la que el control de flujo se lleva a cabo a nivel de paquete. Esto significa que el tamaño de los buffers se reduce mucho en wormhole, ya que sólo se tienen que asegurar los recursos para un flit, no para el paquete entero. La latencia viene dada por la siguiente fórmula:

$$T_{WH} = D(t_r + t_s + t_w) + max(t_s, t_c) \left\lceil \frac{L}{W} \right\rceil$$

Toro

Es un tipo de topología de red directa y ortogonal. Pertenece a la familia de topologías n-cubo k-arias. En un toro, cada nodo está conectado con dos vecinos a lo largo de dos o más dimensiones. Un toro 2D 4x4 se podría visualizar como cuatro colgantes de perlas cerrados, con cuatro perlas cada uno. Estos cuatro colgantes estarían dispuestos verticalmente, de manera que cada perla estuviera en línea y conectada con la inmediatamente superior y la inferior. Además, cada perla de la cadena más superior, estaría conectada a la perla correspondiente de la cadena más inferior. Estas características, diferencian al toro de la malla, ya que los nodos situados en los extremos de esta, no están conectados entre sí.



b) Para el ejemplo del apartado anterior, analiza los resultados estadísticos que se obtienen, explicando cada uno de ellos. Repite la misma simulación para otros tamaños de paquete y de buffer (8 y 32 flits, por ejemplo). Observa lo que ocurre

y comenta los resultados (indica lo que ocurre cuando los paquetes deben usar el mismo canal o lo que pasa cuando el tamaño del paquete es mucho mayor que el del buffer).

Tras salir del cuadro de simulación interactiva, en el recuadro blanco de la pestaña de simulación se muestran las estadísticas de la simulación. A continuación se muestran los resultados obtenidos tras simular la traza del apartado anterior. Dentro de dichas estadísticas se incluyen diferentes tasas:

- Tasa de creación = nº total de flits / nº total de ciclos / nº total de nodos
- Tasa de envíos = nº total de flits / nº total de nodos
- Tasa de recepción = nº total de flits / nº total de ciclos / nº total de nodos

```
----------------------------------------
Item: 1, Variable: 0.3
----------------------------------------
Ciclos: 14
Paquetes Generados: 2
Paquetes Enviados:  2
Paquetes Recibidos: 2
-------------------------------------
Tasa  Creacion (flits/ciclo/nodo): 0.0267857142857143
Tasa de  Envio (flits/ciclo/nodo): 0.375
Tasa Recepcion (flits/ciclo/nodo): 0.0267857142857143
-------------------------------------
Latencia Cabeza:                10
Latencia Cabeza (sin bloqueos):  8
Latencia Paquete:               12
Desviacion Estandar:            2
Latencia Paquete (sin bloqueos): 10
Ciclos de Bloqueo en Red:       2
------------------------------
Camino medio (nodos):       3
Camino medio (canales):     2


SIMULACION COMPLETADA CON EXITO.
Tiempo de simulacion: 0 h, 0 m, 3 s.
```

¿Qué ocurre cuando los paquetes deben usar el mismo canal? Lo que ocurre es que se produce una situación de bloqueo, ya que para salir del nodo 1, solo existe un buffer de salida. Al llegar al mismo tiempo ambos paquetes, uno de ellos se tiene que bloquear para que el otro paquete use el buffer.

¿Qué pasa cuando el tamaño del paquete es mucho mayor que el del buffer? Lo que ocurre es que el paquete que usa primero el buffer, lo retiene durante más ciclos, aumentando a su vez los ciclos de bloqueo y, en definitiva, la latencia media de la comunicación.

c) ¿Cuál es la diferencia de funcionamiento si está activado o no el adelantamiento en colas o buffers?

La diferencia es que cuando el adelantamiento en colas está activado, el flit que entre en una cola se colocará en la primera posición libre de esta, mientras que con el adelantamiento desactivado dicho flit se colocaría al final de la cola y tendría que atravesar todas las posiciones intermedias para llegar al comienzo de la cola. La consecuencia del adelantamiento de colas es que se reduce la latencia, pues se reducen el número de "saltos" que el paquete tiene que dar hasta llegar a su destino. Esta diferencia será más evidente cuanto mayor sea el tamaño del buffer.

d) Propón como deberías configurar el simulador para que funcionara como VCT.

VCT realiza la conmutación a nivel de paquete, por lo que un nodo no puede aceptar un paquete si no puede asegurar que, en caso de bloqueo, puede almacenar el mensaje entero. Por lo tanto, para simular VCT en el simulador, bastaría con incrementar el tamaño de las colas por encima del tamaño de los paquetes.

e) Reproduce una situación en la que se muestre cómo el uso de canales virtuales puede reducir la contención provocada por wormhole, y con ello aumentar la utilización de los canales.

Simplemente con aumentar el número de canales virtuales a dos, se aprecia la diferencia, en especial si el tamaño de los paquetes es muy grande. En este caso, el paso de los paquetes de un nodo a otro se lleva a cabo de manera multiplexada en el tiempo, ya que el cable que los une es compartido por ambos paquetes, a pesar de que los exista un buffer para cada paquete. De esta manera, cuando ambos paquetes llegan al buffer de salida, el envío a través del cable que conecta ambos nodos se realiza de manera alterna: en t_i pasa pasa un flit de un paquete, y en t_{i+1} pasa un flit del otro paquete.

Si en el simulador aumentamos el tamaño de los buffers y desactivamos el adelanto en colas, se aprecia mejor la alternancia (véase la Figura 1).

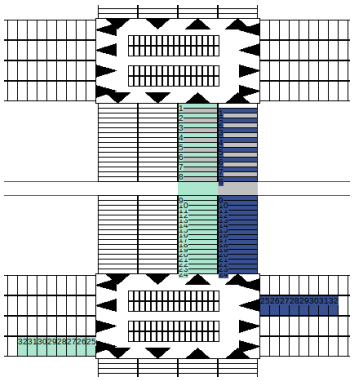


Fig. 1. Canales virtuales.

2. Conmutación Segmentada

2.1 Objetivo

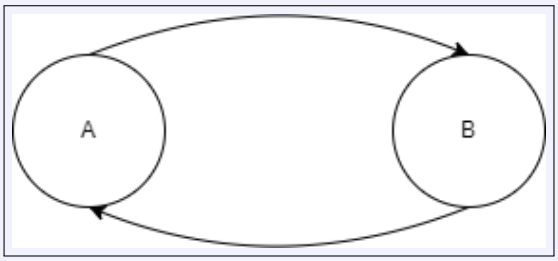
Entender el problema de los deadlocks, y saber reproducirlos.

2.2 Desarrollo

Se trata de crear una situación de interbloqueo (*deadlock*) en diferentes topologías o en su caso analizar las circunstancias que lo impiden.

Interbloqueo

También llamado bloqueo mortal (*deadlock*), consiste en una dependencia de A en B y de B en A en el mismo instante de tiempo. Esto ocurre cuando, durante su recorrido, un recurso (e.g. un buffer) requerido por A está siendo utilizado por B. Al mismo tiempo, durante su recorrido, B requiere del uso de un recurso que está siendo utilizado por A. De esta manera, ninguno de los dos podrá avanzar, ya que dichos recursos no podrán ser liberados nunca.



a) Un caso muy sencillo se puede obtener en una red de interconexión con topología toro 2D, considerando canales unidireccionales.

Para obtener un interbloqueo en una dimensión, es conveniente (y para dimensiones

de cuatro nodos, necesario) que los paquetes recorran el camino más largo posible. De esta manera, será más probable que ambos paquetes se encuentren en sus caminos. Asimismo, es necesario que ambos paquetes salgan de nodos situados a la mayor distancia posible entre ellos ya que, de lo contrario, con cuatro nodos se rompería cualquier ciclo posible. La Figura 2 muestra una situación de interbloqueo en la que el paquete A sale del nodo 0 al 1, y el paquete B del nodo 2 al 3. Véase que si hubiéramos lanzado el paquete B desde el nodo 1 al 2, se rompería el ciclo de dependencia, ya que el paquete A podría alcanzar el nodo 1 sin problemas.

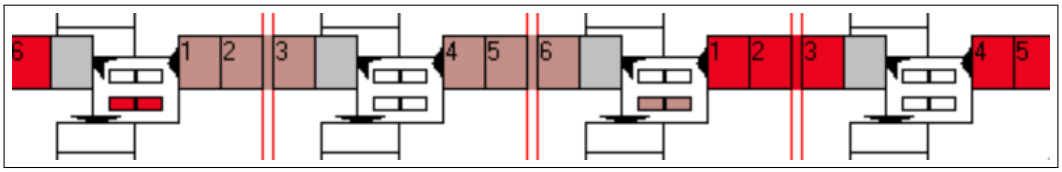


Fig. 2. Interbloqueo en Simured.

b) Plantear casos más complejos que intervengan nodos de dos dimensiones distintas, utilizando la topología que consideréis oportuna y aplicando canales bidireccionales o unidireccionales. En su caso si no se consigue el bloqueo analizar las circunstancias que lo impiden.

En todos los casos, se utilizará el simulador *SimuRed*, descrito en el apéndice A.

En este caso el interbloqueo se tiene que dar de tal manera que todos los paquetes tengan que utilizar las dos dimensiones en su recorrido. Para ello, y en primer lugar, hay que utilizar un algoritmo de encaminamiento totalmente adaptativo, ya que de lo contrario se rompería cualquier ciclo. En segundo lugar, los canales tienen que ser bidireccionales. En tercer lugar, en este caso será necesario lanzar cuatro paquetes. Por último, la longitud del paquete debería ser mayor que el doble de la longitud de las colas. En este caso, para generar el interbloqueo, hay que lanzar los paquetes en cruz. El Listing 2 muestra un ejemplo de traza que enviaría los paquetes en cruz.

Listing 2. Traza para provocar un interbloqueo en dos dimensiones.

```
0  9  6
0  6  9
0  10  5
0  5  10
```

Al utilizar un algoritmo de encaminamiento determinista, es posible que no se produzca un interbloqueo a la primera. Por eso, es probable que se tengan que realizar varias ejecuciones con la misma traza hasta que aparezca. La Figura 3 muestra un posible interbloqueo en dos dimensiones usando Simured.

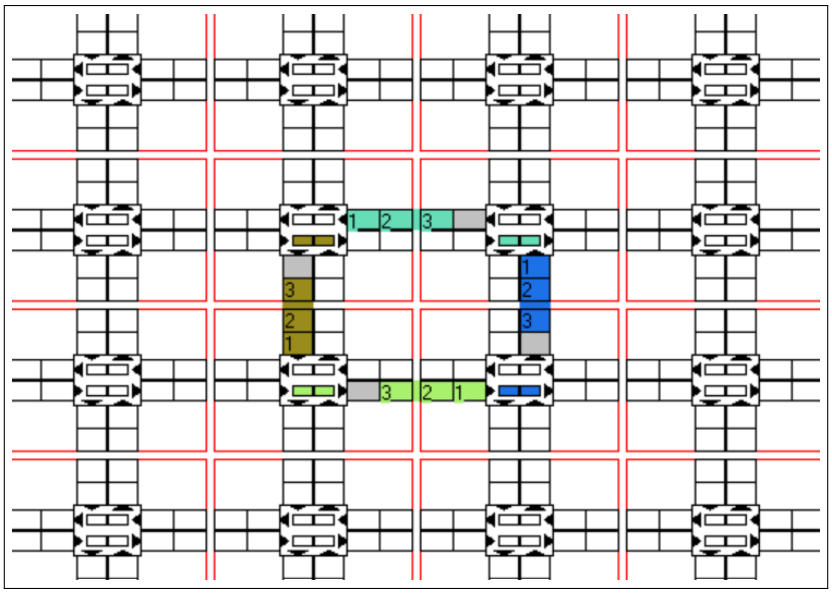


Fig. 3. Interbloqueo en dos dimensiones.

3. Encaminamiento determinista y adaptativo

3.1 Objetivo

Comprobar y entender el funcionamiento de algoritmos de encaminamiento deterministas y adaptativos.

3.2 Desarrollo

La práctica consiste en resolver una serie de cuestiones relacionadas con el mecanismo de encaminamiento en redes de interconexión. Para realizar esta práctica se hará uso del simulador *SimuRed*, descrito en el apéndice A.

a) Reproduce una situación con la que se pueda observar el beneficio introducido por el encaminamiento adaptativo en lo que se refiere a nivel de prestaciones de la red, comparándolo con el caso de encaminamiento determinista.

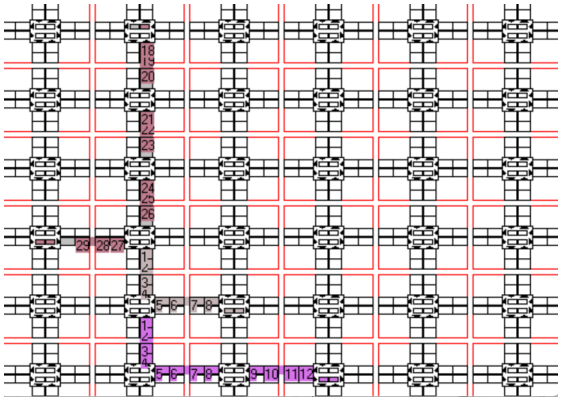
Se trata de realizar una pequeña traza para alcanzar el objetivo, y observar mediante una simulación interactiva la evolución de los paquetes por la red. Utilizar para ello el algoritmo de encaminamiento determinista y el totalmente adaptativo.

Para comprobar el beneficio del encaminamiento adaptativo frente al determinista es necesario crear una situación de congestión en la que varios paquetes hagan uso de los mismos recursos. De esta manera, se creará un cuello de botella en ese punto que el encaminamiento adaptativo será capaz de solventar y el determinista no. En este caso, se va a usar la traza mostrada en el Listing 3.

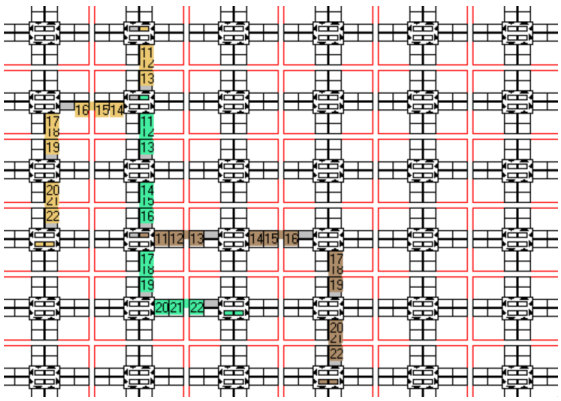
Listing 3. Traza para provocar congestión.

```
0  9  6
0  6  9
0  10  5
0  5  10
```

En la Figura 4 se puede ver el resultado de lanzar dicha traza con ambos encaminamientos. Como se puede ver la reducción en el número de ciclos requeridos para completar la simulación es notable cuando se emplea el algoritmo adaptativo. No obstante, hay que destacar que el encaminamiento totalmente adaptativo tomará dará lugar a diferentes resultados, pudiendo darse casos en que se produzca congestión como en el caso del encaminamiento determinista.



(a) Encaminamiento determinista



(b) Encaminamiento adaptativo

```
_____________________
Item: 1, Variable: 0.3
_____________________________________________
Ciclos: 103
Paquetes Generados: 3
Paquetes Enviados:  3
Paquetes Recibidos: 3
_____________________________________________
Tasa  Creación (flits/ciclo/nodo): 0.0258899676375405
Tasa de  Envio (flits/ciclo/nodo): 2.66666666666667
Tasa Recepción (flits/ciclo/nodo): 0.0258899676375405
_____________________________________________
Latencia Cabeza:                   43.6666666666667
Latencia Cabeza (sin bloqueos):    14
Latencia Paquete:                  74.6666666666667
Desviación Estándar:               23.6971634495683
Latencia Paquete (sin bloqueos):   45
Ciclos de Bloqueo en Red:          29.6666666666667
_____________________________________
Camino medio (nodos):         5
Camino medio (canales):       4


SIMULACIÓN COMPLETADA CON ÉXITO.
Tiempo de simulación: 0 h, 0 m, 11 s.
```

(c) Estadísticas encaminamiento determinista

```
_____________________
Item: 1, Variable: 0.3
_____________________________________________
Ciclos: 45
Paquetes Generados: 3
Paquetes Enviados:  3
Paquetes Recibidos: 3
_____________________________________________
Tasa  Creación (flits/ciclo/nodo): 0.0592592592592593
Tasa de  Envio (flits/ciclo/nodo): 2.66666666666667
Tasa Recepción (flits/ciclo/nodo): 0.0592592592592593
_____________________________________________
Latencia Cabeza:                   14
Latencia Cabeza (sin bloqueos):    14
Latencia Paquete:                  45
Desviación Estándar:               0
Latencia Paquete (sin bloqueos):   45
Ciclos de Bloqueo en Red:          0
_____________________________________
Camino medio (nodos):         5
Camino medio (canales):       4


SIMULACIÓN COMPLETADA CON ÉXITO.
Tiempo de simulación: 0 h, 0 m, 5 s.
```

(d) Estadísticas encaminamiento adaptativo

Fig. 4. Encaminamiento determinista vs adaptativo

b) Para comprobar las diferencias de una forma más clara entre los dos tipos de en-

caminamiento (en este caso determinista y el adaptativo de Duato), se realizará una evaluación más completa.

Consistirá en obtener la latencia para diferentes tasas de inyección y comparar mediante las correspondientes gráficas los resultados para ambos algoritmos de encaminamiento. Un número recomendable de paquetes son 20000 **descartando, de cara a los resultados, del 5 al 10% de los iniciales. Para este apartado utilizar la opción de guardar estadísticas en un fichero e ir acumulando los resultados en una misma gráfica para poder realizar las oportunas comparativas.**

c) Finalmente realizar una comparativa con diferente número de canales virtuales y justificar adecuadamente los resultados obtenidos.

4. Encaminamiento determinista y adaptativo

4.1 Objetivo

- Poner de manifiesto el problema de llevar datos compartidos a varias caches.
- Comparar protocolos de coherencia basados en invalidación y basados en actualización.

4.2 Desarrollo

Se trata de poner de manifiesto el problema denominado falsa compartición (false sharing). Esta situación se da cuando unos bloques de datos son compartidos por varios procesadores. Un procesador modifica un determinado dato que pertenece a un bloque dado. Otro procesador modifica otro dato distinto pero que pertenece al mismo bloque. El bloque es llevado a las caches de ambos procesadores y, si se usa un protocolo de coherencia basado en invalidación, se producirán elevadas tasas de fallos en cache y de transferencias de bloques.

Para realizar esta práctica el alumno debe usar el simulador SMPCache (apéndice B).

a) El alumno debe crear trazas de accesos a memoria que reproduzcan esa situación. Para ello usará el código mostrado en el Listing 4, el cual realiza una operación elemental sobre vectores que consiste en hacer una copia de un vector en otro. Dicha copia la realizan entre varios procesadores, repartiéndose por igual las componentes del vector. Para poder ver el efecto del problema de la falsa compartición, el alumno debe considerar al menos dos formas de hacer ese reparto: una que produzca el problema de la falsa compartición y otra que no.

```
1  // Programa que copia un vector en otro
2  // Genera trazas para el simulador SMPCache
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  const int NUM = 1000; // Longitud del vector
8  const int N = 8;      // Maximo numero de procesadores
9  const int READ = 2;   // Lectura de datos
10 const int WRITE = 3;  // Escritura de datos
11 const int FETCH = 0;  // Captura de instrucciones
12
13 int n, i, proc;
14
15 // Funcion para repartir los datos entre los
      procesadores
16 int DisData(char key, int i) {
```

```
17      int proc;
18      switch (key) {
19          // INSERTAR CODIGO
20      }
21      return proc;
22  }
23
24  // Funcion que escribe en un fichero un acceso a
        memoria.
25  // Parametros: fichero, tipo de acceso y direccion
        accedida
26  void WriteAccess(FILE *f, int type, void *address) {
27      fprintf(f, "%d %p\n", type, address);
28  }
29
30  int main(int argc, char *argv[]) {
31      char a[NUM], b[NUM]; // Vectors
32      FILE *f[N]; // File per processor
33      char filename[30];
34      char key;
35
36      if ((argc == 2) && (INSERTAR CODIGO)) {
37          key = argv[1][0];
38      } else {
39          printf("Syntax: ...\n");
40          exit(0);
41      }
42
43      for (n = 1; n <= N; n *= 2) {
44          for (proc = 0; proc < n; proc++) {
45              sprintf(filename, "traza%c%d_%d.prg",
                    key, n, proc + 1);
46              f[proc] = fopen(filename, "w");
47          }
48          for (i = 0; i < NUM; i++) {
49              a[i] = b[i]; // Example operation
50              proc = DisData(key, i);
51              WriteAccess(f[proc], READ, &b[i]);  //
                    Read b[i]
52              WriteAccess(f[proc], WRITE, &a[i]); //
                    Write to a[i]
53          }
```

```
54          for (proc = 0; proc < n; proc++) {
55              fclose(f[proc]);
56          }
57      }
58      return 0;
59  }
```

Listing 4. Código para realizar copia de vectores y generar trazas para SMPCache.

Se deben hacer pruebas con 2, 4 y 8 procesadores, y por tanto se deben generar las trazas necesarias para cada uno de esos tres casos. Por ejemplo, cuando se simule un SMP con 4 procesadores son necesarios 4 ficheros de traza, uno para cada procesador. La configuración base del SMP será la dada por los siguientes parámetros:

- **Protocolo de coherencia: MESI**
- **Procesadores: 8**
- **Arbitraje del Bus: LRU**
- **Tamaño de palabra: 8 bits**
- **Palabras por bloque: 128**
- **Bloques en la memoria: 8192**
- **Bloques en la cache: 64**
- **Correspondencia: Asociativa por conjuntos**
- **Conjuntos: 32 (2 vías)**
- **Reemplazo: LRU**

Se debe tomar nota de los resultados que produce el simulador, y que son el número de transferencias de bloques en el bus y la tasa de fallos o aciertos de cache.

El código del Listing 5 es el que hay que incluir en la función `DisData`. Los :

```
1  int DisData(char key, int i) {
2      int proc;
3      switch (key) {
4          case 'C':
5              proc = (n * i) / NUM; /* Reparto por
                   bloques */
6              break;
7          case 'I':
8              proc = i % n; /* Reparto ciclico */
```

```
9                 break;
10        }
11        return proc;
12    }
```

Listing 5. Funcion para repartir los datos entre los procesadores.

También nos tenemos que asegurar de que se pasan los parámetros 'C' e 'I' al programa:

```
1  if ((argc == 2) && (argv[1][0]=='C' ||
      argv[1][0]=='E')) {
2      key = argv[1][0];
3  } else {
4      printf ("Sintaxis: C (Continuo) | E
          (entrelazado)\n");
5      exit(0);
6  }
```

Listing 6. Comprobación de los parámetros del programa.

b) Con las trazas generadas en el apartado anterior realiza una comparación de los protocolos MESI (invalidación) y Dragon (actualización). Varía el número de procesadores para realizar la comparación.

c) Finalmente realizar una comparativa con diferente número de canales virtuales y justificar adecuadamente los resultados obtenidos.

5. Encaminamiento determinista y adaptativo

5.1 Objetivo

5.2 Desarrollo

Ejercicio 1. Extraer la topología de red a un fichero ”topologia.topo”, y dibujarla utilizando “InfiniBand-Graphviz”:

Para hacer este ejercicio, lo primero que hay que hacer es conectarse a CELLIA usando `slurm`

Ejercicio 2. ¿Qué topología de las vistas en clase hay construida en CELLIA? Anotar sus propiedades (nº de nodos, nº de switches, grado del switch, etc.)

Ejercicio 3. Modificar el fichero “device_list.c” para que ofrezca más información acerca del HCA de un nodo.

Ejercicio 4. Modificar un programa con verbs para añadir información acerca del uso de la red.

6. La dimensión temporal

Verilog permite especificar un *tiempo de demora* en *asignaciones continuas*. Esto se hace incluyendo el carácter almohadilla (#) y un número real después de la palabra clave `assign`. El número real indica la demora en las unidades indicadas por la *escala temporal*. También se puede especificar una demora en asignaciones procedurales. En este caso, la almohadilla y el número se colocan tras los símbolos < o <=. Para especificar la *escala temporal*, se usa la directiva del compilador `‘timescale` con la siguiente sintaxis:

```
`timescale time-unit / time-precision
```

time-unit indica la unidad que estará asociada con las demoras[1]. Por otro lado, *time-precision* indica la granularidad con la que operará el simulador (sueles tener un valor de *ps*). Estos valores serán tenidos en cuenta también por *tareas y funciones del sistema* como `$time`.

Otra manera de invocar una demora temporal dentro del *código procedural* es usando la *sentencia de demora*, que es un # seguido de un número. Es importante tener en cuenta que las demoras temporales serán tenidas en cuenta durante la simulación, pero no durante la síntesis.

[1] e.g. si *time-unit* es igual a *ns*, una demora de 2 se traducira a 2*ns*.

7. Simulación

Una vez que la tenemos un modelo de Verilog con sintaxis y semántica correcta, se puede y debe utilizar un simulador para observar su operación.

El simulador opera comenzando en un *tiempo de simulación* de cero. En este punto, el simulador inicializa todas las señales a un valor por defecto de `x`. A continuación, el simulador inicializa las señales o variables cuyos valores iniciales han sido declarados explícitamente. Tras esto, el simulador comienza la ejecución de las sentencias concurrentes del diseño.

Aclaración

El simulador en realidad no puede ejecutar las sentencias concurrentes simultáneamente. Sin embargo, crea esa ilusión mediante el uso de una *lista de eventos* basados en tiempo. y una matriz de sensibilidad basada en todas las listas de sensibilidad individuales.

Cada sentencia concurrente se da lugar a un procesos software, mientras que una instanciación de módulo puede dar lugar a uno o más procesos. En tiempo de simulación cero, todos los procesos generados se planifican para ejecución, de los cuales se selecciona uno. La ejecución del código procedural se lleva a cabo secuencialmente. Si se ha especificado una demora, la ejecución del proceso se suspende. Cuando esto ocurre (o cuando el proceso se ha terminado de ejecutar) se selecciona otro proceso para su ejecución.

Cuando todos los procesos se han ejecutado, se completa lo que se denomina un *ciclo de simulación*.

Cuando el simulador se encuentra una asignación no bloqueante sin demora, ésta se supone que se ejecuta en tiempo de simulación cero. No obstante, lo que ocurre realmente es que se planifica su ejecución para el tiempo de simulación actual al que se le suma una *demora delta*.

Cada vez que un ciclo de simulación es completado, la lista de eventos es escaneada para buscar las señales que antes cambien. Debido a que algunos procesos pueden ser sensibles al cambio de señales, la matriz de sensibilidad indica, para cada señal, qué procesos tienen dicha señal en su lista de sensibilidad. Cada proceso que sea sensible a una señal, será planificado para que se ejecute en el siguiente ciclo de simulación. Este proceso se repite hasta que la lista de eventos queda vacía, momento en el cual la simulación termina.

8. Bancos de pruebas

Un *banco de prueba* especifica una secuencia de entradas que serán aplicadas por el simulador al modelo HDL que será probado. Dicho modelo puede ser un módulo Verilog o un diseño más grande. La entidad sujeta a las pruebas se denomina *unidad bajo prueba (UBP)*.

En bancos de pruebas es frecuente emplear otro tipo de sentencia concurrente, el bloque `initial`. Esta sentencia se ejecuta una sola vez, en tiempo de simulación cero. Al igual que el bloque `always`, se puede incluir un bloque `begin-end` para declarar el código procedural.

Listing 7. Banco de prueba para un circuito detector de numeros primos.

```
1  `timescale 1 ns / 100 ps
2
3  module Vrprime_tb1();
4  reg [3:0] Num;
5  wire Prime;
6
7  Vrprimed UUT (.N(Num), .F(Prime));
8
9      initial begin: TB
10         integer i;
11         for (i = 0; i <= 15; i = i+1) begin #10
12             Num = i;
13         end
14 endmodule
```

Los simuladores suelen generar un fichero que especifica las formas de onda que cada señal toma durante la simulación. Dicho fichero puede ser leído por aplicaciones específicas para su visualización. El correcto funcionamiento la UBP quedará a cargo del usuario, cuya función es interpretar si las formas de onda tienen sentido. Debido a que este proceso es tedioso, es conveniente formatear la salida del simulador para que sea más amigable. Para ello se pueden usar tareas del sistema como `$write` o `$display` para mostrar información que se considere útil por consola o, incluso, redirigirla a un fichero.

Hay situaciones en las que la combinación de entradas que una UBP puede recibir es tan elevada. En estos casos, puede no resultar factible para el diseñador examinar los resultados obtenidos para cada combinación de entradas. En estos casos, conviene escribir un banco de prueba autoevaluado

9. Circuitos combinacionales

9.1 Decodificadores

A pesar de que los decodificadores en Verilog suelen formar parte de diseños mayores, también se pueden definir y probar de manera aislada. Para diseñar un decodificador, existen las siguientes aproximaciones:

- *Estilo estructural.* Consistiría en definir la operación del decodificador mediante puertas lógicas. Esta aproximación daría lugar a un código difícil de leer y de mantener.
- *Estilo de flujo de datos.* Se define el comportamiento del circuito mediante sentencias de asignación contínua. El Listing 8 muestra un posible diseño.

Listing 8. Estilo de flujo de datos.

```
module decodificador_2a4(input a0,
                         input a1,
                         input en,
                         output y0,
                         output y1,
                         output y2,
                         output y3);
    assign y0 = en ? ({a1,a0} == 2'b00) : 0;
    assign y1 = en ? ({a1,a0} == 2'b01) : 0;
    assign y2 = en ? ({a1,a0} == 2'b10) : 0;
    assign y3 = en ? ({a1,a0} == 2'b11) : 0;
endmodule
```

- *Estilo conductual.* Se modela el circuito por medio de código prodedural. El Listing 9 muestra un posible diseño.

Listing 9. Estilo conductual.

```
module decodificador_2a4(input a0,
                         input a1,
                         input en,
                         output reg y0,
                         output reg y1,
                         output reg y2,
                         output reg y3);

    always @(*) begin
        if(en) begin
            {y3,y2,y1,y0} = 0
            case ({a1,a0})
```

```
                2'b00: y0 = 1;
                2'b00: y1 = 1;
                2'b00: y2 = 1;
                2'b00: y3 = 1;
                default: {y3,y2,y1,y0} = 0;
            endcase
        end
    end

endmodule
```

Una vez diseñado el circuito, es hora de probarlo.

Listing 10. Banco de pruebas del decodificador.

```
`timescale 1 ns / 1 ps
module tb_decodificador_2a4();

    // declaracion de tipos de datos
    reg a0_p, a1_p, en_p;
    wire y0_p, y1_p, y2_p, y3_p;
    integer i, errores;
    reg [3:0] resultado_esperado;

    decodificador_2a4 UBP (.a0(a0_p),
                           .a1(a1_p),
                           .en(en_p),
                           .y0(y0_p),
                           .y1(y1_p),
                           .y2(y2_p),
                           .y3(y3_p)
                          );
    initial begin
        errors = 0;
        for(i = 0; i < 8; i++) begin
            {en, a0, a1} = i;
            #10;
            resultado_esperado = 4'b0000;
            if(en) resultado_esperado[{a1,a0}] = 1'b1;
            if ({y3,y2,y1,y0} !== resultado_esperado)
               begin
                errores = errores + 1;
            end
        end
        $display("Test completado con %d errores", errores
           );
```

```
30      end
31
32  endmodule
```

Si la compilación con `iverilog` ha sido correcta, el banco de pruebas debería mostrar 0 errores.

9.1.1 Tipos de decodificadores

- *Decodificadores binarios.* Un decodificador binario n-a-2^n es un circuito con una entrada de n bits y una salida activa de las 2^n totales.
- *Decodificadores de siete segmentos.* Estos circuitos tienen una entrada de 4 bits (Decimal Codificado en Binario) y una salida de 1 bit para cada segmento del LCD.

9.2 Codificadores binarios

Un codificador binario es un circuito que hace la acción opuesta al decodificador binario. Si un decodificador binario convierte un código de n bits en otro de m bits, el codificador binario correspondiente debe ser capaz de transformar el código de m bit en el de n bits.

9.3 Multiplexores

Una operación muy común en diseño digital es escoger una fuente de datos entre varias para transferirla a su destino a través de un medio compartido, como un bus. Esta operación es tan común que tiene su propio nombre, *multiplexación* o *mux* para abreviar.

Una definición más precisa de multiplexador es la de *conmutador digital que conecta los datos de 1 entre n fuentes a una sola salida*. Para ello, al multiplexor hay que conectarle una entrada que permita hacer una selección entre n fuentes. La anchura de este entrada de selección tendra que ser $log_2(n)$, donde n es el número de fuentes.

9.4 Demultiplexores

Los demultiplexores son los dispositivos digitales que realizan la operación opuesta a los multiplexores, es decir, conectar una entrada (de b bits) a una de entre n destinos/salidas.

10. Dispositivos triestado

Los *dispositivos triestado* son muy importantes porque, a nivel de la placa de circuito impreso son una alternativa ampliamente utilizada a los multiplexores. Existen diferentes tipos de dispositivos triestado.

10.0.1 Buffers triestado

Es el dispositivo triestado más sencillo. También se le puede llamar *conductor triestado*. El buffer triestado es como un buffer normal al que se le conecta una tercera señal llamada *habilitación triestado*. Cuando esta señal está activa, el buffer actúa como un buffer corriente, conectando la entrada a la salida. Por el contrario, cuando habilitación triestado es cero, la salida pasa a un estado de alta impedancia y el buffer se comporta como si ni siquiera estuviese ahí.

Los buffers triestado son raramente usados en chip, es decir, dentro de ASICs o FPGAs, ya que los multiplexores suelen ofrecer un mayor desempeño.

10.1 Codificación con prioridad

En la sección de codificadores binarios (REFERENCIAR) se vio que estos dispositivos seleccionan la entrada activa de entre 2^n para codificarla en la salida pero, ¿qué ocurre si varias entradas están activas al mismo tiempo? En este caso convendría asignar una *prioridad* a cada una para que la entrada con más prioridad sea la seleccionada por el codificador. Los dispositivos que hacen esto se denominan *codificadores con prioridad*.

Los codificadores con prioridad tienen una salida *OCIOSA* que se activa si ninguna de las entradas está activa.

11. Glosario

- Sentencia de demora *(delay statement)*.
- Tiempo de simulación *(simulation time)*.
- Lista de eventos *(event list)*.
- Matriz de sensibilidad *(sensitivity matrix)*.
- Ciclo de simulación *(simulation cycle)*.
- Banco de prueba *(test bench)*.
- Unidad bajo prueba *(unit under test (UUT))*.
- Decodificador de siete segmentos *(seven-segment decoder)*.
- Multiplexación *(multiplexing)*.
-
- Dispositivo triestado *(three-state device)*.
-
-
- Conductor triestado *(three-state driver)*.
-
-
- Habilitación triestado *(three-state enable)*.
-
-
- Codificador con prioridad *(priority encoder)*.
-
-
- OCIOSA *(IDLE)*.

Acknowledgments

Delete if not applicable

References

Appendix A: Supplemental Materials

Brief description of supplemental files

Appendix B: Change Log

If updating document with errata, detail changes made to document – delete if not applicable.