Arquitectura de Computadores II: Práctica 3

25 de septiembre de 2024

1. Autómata y Estructuras de Datos

Los objetivos de esta sesión son por un lado rematar vuestros conocimientos sobre circuitos secuenciales, para ello introduciremos el diseño de autómatas (también conocidos como máquinas de estados finitas, 'FSM' en ingles) y estructuras de datos hardware. Estas herramientas son fundamentales para el modelado de algoritmos software utilizando circuitos hardware, y son ampliamente utilizadas en el diseño de CPUs y de aceleradores como las GPUs (Graphic Processing Units), etc.

Utilizando estos conocimientos, definiremos el núcldeo del camino de datos de un procesador. Dicho nucleo estará compuesto por un banco de registos, un sumador y una cola circular.

2. Máquinas de Estado

Un autómata, o máquina de estados finitos, es un modelo matemático compuesto por por estados y transiciones entre estados. Un autómata hardware es la tradución o modelado de un autómata utilizando circuito secuenciales y combinacionales. Por lo tanto en cada ciclo se evaluará el estádo del circuito y se realizará una transición entre estados si corresponde. Por regla general utilizaremos circuitos secuenciales para almacenar el estado del autómata, el cual codificaremos utilizando un vector de bits registrado (p. ej. un automata de 3 estados utilizará dos bits y los estados 00, 01, 10). Las transiciones entre estados las modelaremos utilizando un circuito combinacional independientemente de si es un autómata de Moore o de Mealy. A continuación se muestra un autómata de ejemplo, que codifica para una estructura de datos finita cuantas entradas están ocupadas. Para reservar una entrada se utiliza la señal 'insert' y para eliminar una entrada la señal 'delete'. Supongamos que la estructura únicamente tiene 3 entradas y que no se puede insertar y eliminar en un mismo ciclo.

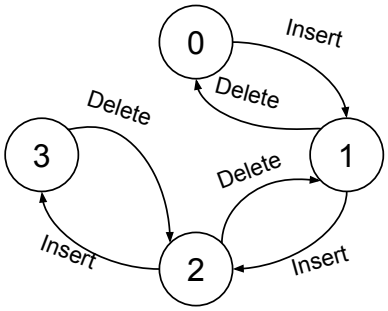

Figura 1: Diagrama conceptual de un autómata simple que calula cuantas entradas hay ocupadas para una estructura de datos de 3 entradas.

La traducción de dicho autómata a un módulo sería la siguiente:

```
module contador_entradas #(
    parameter ENTRADAS = 3
) (
```

```
    input logic clk_i,    // Señal del reloj
    input logic rstn_i,   // Señal de reset
    input logic insert_i, // Señal de adquisición de entrada
    input logic delete_i, // Señal de eliminación de entrada
);

logic [$clog2(ENTRADAS):0] estado_d, estado_q;
logic insert_enable;
logic delete_enable;

// El usuario puede insertar una entrada
assign insert_enable = (estado < ENTRADAS) & insert_i;

// El usuario puede eliminar una entrada
assign delete_enable = (estado > 0) && delete_i;

// Lógica Combinacional de siguiente estado
always_comb begin
    estado_d = estado_q;
    if (insert_enable) begin
        estado_d = estado_q + {'h0, 1'b1};
    end else if (delete_enable) begin
        estado_d = estado_q - {'h0, 1'b1}
    end
end

// Lógica Secuencial de estado actual
always_ff @(posedge clk_i, rstn_i) begin
    if (~rstn_i) begin
        estado_q <= 'h0;
    end else begin
        estado_q <= estado_d;
    end
end

endmodule
```

Para calcular cuantos bits necesitamos de estado, utilizaremos la función $clog2 que devuelve un entero que representa la potencia de 2 más pequeña que es mayor o igual al parametro de entrada. En el ejemplo anterior para 3 nos devolverá 2, ya que $2^2 = 4$.

3. Estructuras de Datos Hardware

En este apartado repasaremos las principales elementos basicos de memorización. Estos elementos se emplean en la elaboración de estructuras de datos hardware. Además repasaremos las tres principales estructuras de datos utilizadas por los arquitectos en sus diseños.

Como ya vimos en la práctica anterior, un registro es un elemento básico de almacenamiento que actualiza su estado en el flanco ascendente de la señal de reloj. Si un registro elemental almacena un bit, para construir un registro de n bits se agrupan n registros elementales y se accederá a ellos como una unidad. Sin embargo, existen otro tipo de elementos de almacenamiento como son las celdas de memoria de acceso aleatorio (Static Random Access Memory, SRAM; en inglés). Una SRAM es un conjunto de elementos de almacenamiento a los que se accede por grupos (filas o rows) y que tiene una densidad elevada, a diferencia de los registros. Es por ello que son muy utiles para la especificación de estructuras de almacenamiento. Durante el curso unicamente utilizaremos registros, pero tendréis la posibilidad de estudiarl las SRAMs en futuros cursos.

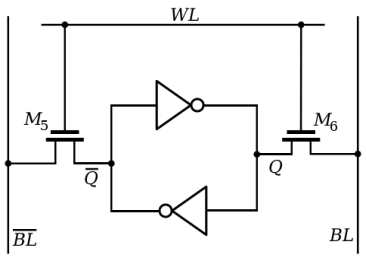

Figura 2: Celda de memoria SRAM usando dos inversores.

Estos elementos básicos almacenamiento nos permiten definir estructuras más complejas de almacenamiento, que solemos llamar estructuras de datos. Cada tipo de estructura se caracteriza por su forma de insertar, recuperar y organizar los datos internamente. A continuación veremos las tres estructuras de datos más sencillas y sus posibles usos.

3.1. Banco de Registros

Un banco de registros es un conjunto de registros de almacenamiento a los que se accede individualmente. Cada registro tiene un identificador único que será utilizado para determinar cuando un registro es leido o escrito. Además el banco de registros tiene otros elementos como el permiso de escritura (Write Enable; WE), el reset o el reloj.

La lectura de un registro no modifica su estado, por lo tanto para leer un registro es suficiente con seleccionar cual se quiere leer. Esta selección se efectúa mediante un multiplexor cuyas entradas son las señales de salida de los registros y la señal de selección debe determinarse en base al el identificador del registro. En la siguiente figura se muestra un esquema de un banco de registros con 32 registros y dos caminos de lectura.

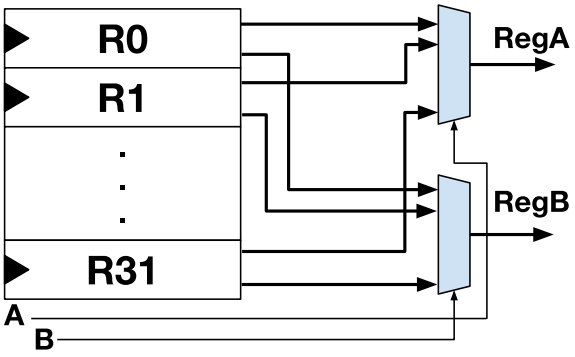

Figura 3: Esquema del camino de lectura de un banco de registros.

La escritura de un registro sí actualiza el estado de este. Por lo que además del identificador de registro, el valor que se quiere almacenar y el permiso de escritura, es necesaria una señal que indique el instante de tiempo en que se actualiza, la señal de reloj. La siguiente figura muestra los elementos del banco de registros que componen el camino de datos para la escritura. Para determinar el registro que se escribe se utiliza el decodificador (dec), el cual tiene como entrada el identificador de registro (ID). El flanco ascendente de la señal de reloj indica el instante en el que se actualiza el registro, si la señal permiso de escritura (WE) está activada.

Notese que las funciones de lectura y escritura son totalmente compatibles y que por lo tanto en un mismo ciclo podemos leer y escribir en el banco de registros. Sin embargo, la lectura del banco de registros es asíncrona mientras que la escritura es sincrona. Como ya explicamos los registros deben esperar al flanco

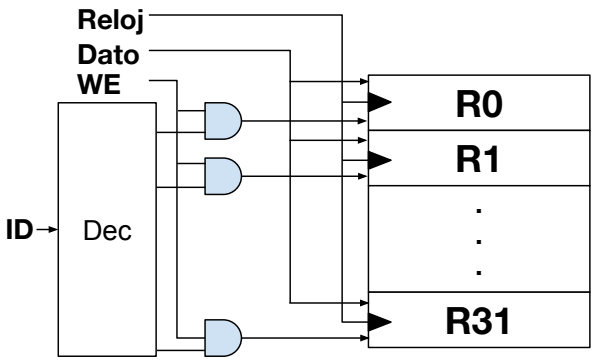

Figura 4: Esquema del camino de escritura de un banco de registros.

de subida del reloj para consolidar la escritura, por lo tanto el valor escrito no puede leerse hasta el ciclo siguiente. Por ello la lógica de control debe detectar estos casos y esperar al ciclo siguiente para poder leer el valor escrito.

El comportamiento descrito para el banco de registros puede generalizarse para cualquier tipo de memoria de acceso aleatorio (RAM en ingles por sus siglas Random Access Memory). Las memorias de acceso aleatorio son aquellas en las que el usuario introduce una clave o dirección de memoria y la RAM se devuelve los datos almacenados para esa dirección. Se llaman de acceso aleatorio porque todos los accesos deben realizar el mismo proceso y no dependen de otros accesos (memoria de acceso secuencial).

A continuación se muestra el código en System Verilog para la implementación de un banco de registros de 'NUM' registers 'WIDTH' bits. Notese, que esta implementación sólo funciona para valores de 'NUM' iguales a una potencia de 2. Si no se cumpliese dicha condición y se estimulase una de las entradas'A_i', 'B_i o 'wid_i' con un valor mayor que NUM el banco de registros tendrá un comportamiento indeterminado.

```
module BancoRegistros #(
    parameter   WIDTH      = 64,
    parameter   NUM        = 8,
    localparam  INDEX_SIZE = $clog2(NUM)

)(
input logic                   clk_i,  // Senyal de reloj
input logic                   rstn_i, // Senyal negada de reset
input logic [INDEX_SIZE-1:0]  A_i,    // ID Registro Lectura 1
input logic [INDEX_SIZE-1:0]  B_i,    // ID Registro Lectura 2
input logic                   we_i,   // Permiso de escritura
input logic [WIDTH-1:0]       dato_i, // Dato a escribir
input logic [INDEX_SIZE-1:0]  wid_i,  // ID registro escritura

output logic [WIDTH-1:0]      RegA_o, // Registro A
output logic [WIDTH-1:0]      RegB_o  // Registro B
);

// Instanciamos los registros WIDTHxNUM
logic [WIDTH-1:0]  registers_q  [NUM-1:0];

// Lógica de Lectura
always_comb begin
    RegA_o = registers_q[A_i];
    RegB_o = registers_q[B_i];
end
```

```
// Lógica de Escritura
always_ff @(posedge clk_i, negedge rstn_i) begin
    if (!rstn_i) begin // Reset
        for(int i=0; i<NUM; ++i) begin
            registers_q[i] <= 'h0;
        end
    end else if (we_i) begin
        registers_q[wid_i] <=  dato_i;
    end
end

endmodule
```

3.2. Buffer Circular

En multitud de ocasiones debemos integrar dos componentes que tienen una relación de productor (p. ej. instrucciones a ejecutar) y consumidor (p. ej. unidad funcional que realiza las operaciones). Habitualmente esta comunicación entre un productor y un consumidor se resuelve utilizando un buffer FIFO (First-In First-Out). El buffer se utiliza para almacenar los datos que no pueden ser consumidos durante un bloqueo del consumidor, si no está lleno. Por lo que el buffer permite absorver ráfagas del productor antes de que las procese el consumidor, incrementando la productividad del sistema. Alternativamente, el consumidor puede extraer datos del buffer mientras el productor está produciendo un dato, si no está vacío.

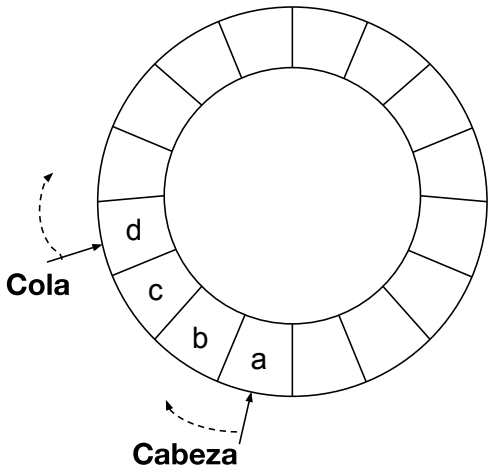

Figura 5: Esquema de un buffer/cola circular.

Una forma usual de construir una FIFO es utilizar un conjunto de posiciones de almacenamiento con dos puertos y reutilizar una posición cuando es consumida. Este tipo de cola FIFO se denomina buffer circular. Uno de los puertos del buffer circular es utilizado por el productor (escritura) y el otro puerto es utilizado por el consumidor (lectura). Cada uno de los puertos tiene asociado un contador (o puntero) para conocer la posición de almacenamiento que se escribe (productor) o que se lee (consumidor).

Al implementar dicho buffer circular en hardware, la estructura de almacenamiento utilizada no es circular, sino un banco de registros o memoria RAM. El puntero es el encargado de dotar de comportamiento 'circular' ya que utilizaremos contadores binarios que cuando un contador llegan al máximo valor, el siguiente valor es cero. El contador asociado al puerto de escritura o inserción (el puntero cola), indica la posición de almacenamiento donde el productor almacena la información. El contador asociado al puerto de lectura o deleción, al cual denominamos cabeza, indica la posición de almacenamiento de donde el consumidor extrae información (lectura).

Cuando la cola recorre las posiciones de almacenamiento y encuentra la cabeza, el buffer está lleno y

no se pueden almacenar datos. Al contrario, cuando la cabeza recorre las posiciones de almacenamiento y encuentra la cola, el buffer está vacío y no se puede extraer información.

A continuación se muestra el escleto de código de un buffer circular. Para esta práctica, deberéis completar su implementación

```
module BancoRegistros #(
    parameter   WIDTH      = 64,
    parameter   NUM        = 8,
    localparam  INDEX_SIZE = $clog2(NUM)

)(
input logic              clk_i,       // Senyal de reloj
input logic              rstn_i,      // Senyal negada de reset
input logic              insercion_i, // Senyal de inserción
input logic [WIDTH-1:0]  dato_i,      // Dato a insertar
input logic              deleción_i,  // Senyal de deleción

output logic [WIDTH-1:0] dato_o,      // Dato leído
output logic             vacia_o,     // Senyal de cola vacía
output logic             llena_o      // Senyal de cola llena
);

// Instanciamos los registros WIDTHxNUM
logic [WIDTH-1:0]  registers_q  [NUM-1:0];
// Instanciamos punteros de control
logic [INDEX_SIZE-1:0] cola_q, cabeza_q;
// Instanciamos senyales de control
logic permiso_insercion, permiso_delecion;
logic [INDEX_SIZE:0] num_q;

// Lógica de inserción
assign permiso_insercion = (num_q <  ) & ( );      // Codigo a completar

// Lógica de deleción
assign permiso_delecion =  (num_q >  ) & ( );      // Codigo a completar

// Lógica de Lectura. Sólo lee si hay un dato
assign dato_o = (vacia_o) ? 'h0 : registers_q[ ];  // Codigo a completar

// Lógica de Escritura.
always_ff @(posedge clk_i, negedge rstn_i)
begin
    if (~rstn_i) begin
        for (int i=0; i<NUM; ++i) begin
            registers_q[i] <= 'h0;
        end
    end else if (permiso_insercion) begin
        registers_q[ ] <= ;                  // Codigo a completar
    end
end

// Lógica de Control
always_ff @(posedge clk_i, negedge rstn_i) begin
    if (!rstn_i) begin
```

```
        cabeza_q <= 'h0;
        cola_q   <= 'h0;
        num_q    <= 'h0;
    end else begin
        cabeza_q <= cabeza_q + {2'b0, permiso_...};  // Codigo a completar
        cola_q   <= cola_q   + {2'b0, permiso_...};  // Codigo a completar
        num_q    <= num_q    + {3'b0, permiso_...}
                             - {3'b0, permiso_...};  // Codigo a completar
    end
end

assign vacia_o = (num_q == 0);
assign llena_o = ((num_q == NUM) | !(rstn_i));

endmodule
```

3.3. Memoria de contenido direccionable

La ultima estructura de datos que estudiaremos en esta práctica es la memoria de contenido direccionable (CAM por sus siglas en ingles Content-Addressable Memory). Al contrario de las memorias de acceso aleatorio en las que el usuario introduce una clave y se devuelve los datos almacenados, una CAM devuelve la clave que pertenece a los datos proporcionados por el usuario. Esta operación es equivalente a la busqueda en una lista, pero en lugar de acceder a cada elemento uno por uno, la CAM busca en toda la memoria al mismo tiempo para ver si los datos están almacenados en alguna posición y devolver esa posición como la clave. La siguiente figura muestra el camino de lectura de una CAM.

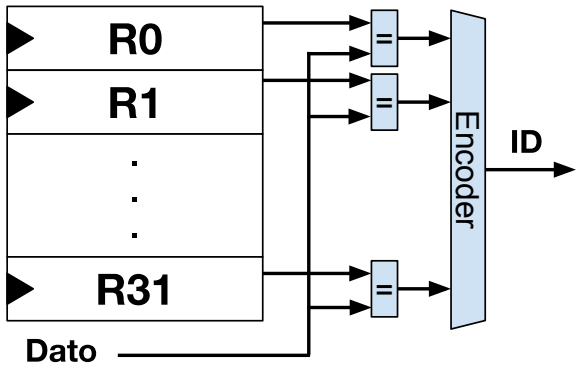

Figura 6: Esquema del camino de escritura de un banco de registros.

De nuevo es la estructura de datos está conformada por un conjunto de registros que son accedidos en paralelo. La salida de cada uno de estos registros está conectada a un comparador (puerta XNOR), al que también se conecta el dato de busqueda. El comparador devuelve un bit que codifica si la clave y el registro son iguales (1'b1) o distintos (1'b0). Cada una de las salidas de los comparadores está conectada a un codificador (encoder) que devolverá en qué registro está almacenado el dato de búsqueda. Existe la posibilidad de que más de un comparador detecte que el valor almacenado por un registro concuerda con el dato buscado. En muchos de los casos de uso de una CAM, este tipo de esceneario no es posible ya que antes de insertar un nuevo valor en la CAM se comprueba si este ya está almacenado. Sin embargo, dependendiendo de la utilidad dada a la CAM, puede ser necesario almacenar más de una vez un mismo dato, en esos casos se puede utilizar un codificador con prioridad (devolverá el id más alto o más bajo).

El camino de escritura de una CAM es exactamente igual al de un banco de registros. Para escribir son necesarias la señal de reloj, el dato de escritura y una señal de permiso.

A continuación se describe el código de una CAM en SystemVerilog.

```
module CAM #(
    parameter   WIDTH      = 64,
    parameter   NUM        = 8,
    localparam  INDEX_SIZE = $clog2(NUM)

)(
input logic                    clk_i,        // Senyal de reloj
input logic                    rstn_i,       // Senyal negada de reset
input logic [WIDTH-1:0]        dato_cmp_i,   // Dato de comparación
input logic                    we_i,         // Permiso de escritura
input logic [WIDTH-1:0]        dato_esc_i,   // Dato a escribir
input logic [INDEX_SIZE-1:0]   wid_i,        // ID registro escritura

output logic [INDEX_SIZE-1:0] id_o,          // Indice encontrado
output logic                   encontrado_o  // Senyal de encontrado
);

// Declaramos un nuevo tipo de datos
typedef logic [INDEX_SIZE-1:0] index_t;

// Instanciamos los registros WIDTHxNUM
logic [WIDTH-1:0]  registers_q  [NUM-1:0];
// Instanciamos las señales de match de salida de los comparadores
logic              match        [NUM-1:0];

// Generador de comparadores
generate
    for (genvar j = 0; j < NUM; j++) begin
        assign match[j] = (registers_q == dato_cmp_i);
    end
endgenerate

// Lóigica de match
always_comb begin
    encontrado_o = 'h0;
    id_o         = 'h0;

    for(int i = NUM-1; i >= 0; i--) begin
        if (match[i]) begin
            encontrado_o |= 1'b1;
            id_o          = index_t'(i);
        end
    end
end

// Lógica de Escritura
always_ff @(posedge clk_i, negedge rstn_i) begin
    if (!rstn_i) begin // Reset
        for(int i=0; i<NUM; ++i) begin
            registers_q[i] <= 'h0;
        end
    end else if (we_i) begin
        registers_q[wid_i] <=  dato_esc_i;
    end
```

```
end

endmodule
```

4. Camino de datos

En la siguiente figura se muestra un esquema del núcleo de un camino de datos de un procesador el cual contiene un banco de registros (circuito secuencial) y un sumador (circuito combinacional).

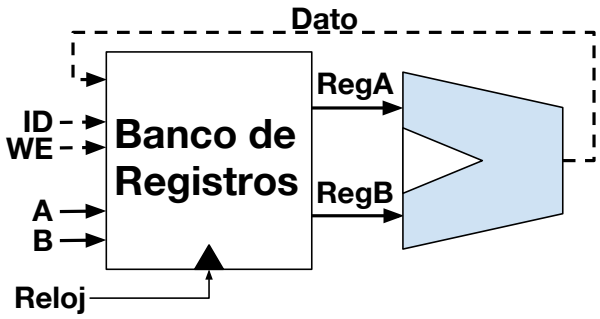

Figura 7: Esquema del nucleo del camino de datos de procesador.

Por lo tanto, el camino de datos de un procesador es una máquina de estados finitos que incluye lógica combinacional y registros que almacenan el estado del sistema. El banco de registros, que utilizaremos, tiene dos caminos de lectura y uno de escritura. El camino de datos de escritura, incluyendo la señal de permiso de escritura (WE), se muestran en trazo discontinuo y los caminos de datos de lectura en trazo continuo. Así mismo, se muestra la señal reloj que sincroniza el funcionamiento del camino de datos. Una operación típica en un camino de datos lee dos registros del banco de registros, utiliza los valores leídos como entradas del sumador y almacena el resultado en un registro del banco de registros.

Sin embargo dicho camino de datos necesita de una unidad de control o controlador, que especifique en cada ciclo los identificadores de registro que see leerán y escribirán. Y cuando se debe activar el permiso de escritura en el banco de registros. Dicho controlador suele implementarse como un automata. En general un camino de datos manipula y procesa datos.

4.1. Controlador del camino de datos

La siguiente figura muestra un diagrama del camino de datos junto a su controlador. El controlador es el encargado de generar las salidas correspondientes para seleccionar los registros adecuados de lectura (A y B) y escritura (ID). Para inicializar el circuito disponemos de la señal 'Inicio' que inicializa el estado del controlador. A si mismo, el controlador tiene una señal de 'Fin' que indica cuando ha finalizado la operación.

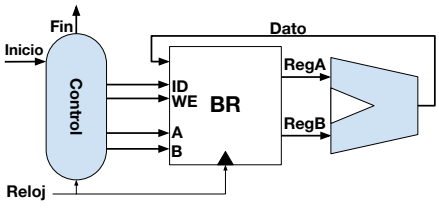

Figura 8: Esquema del controlador y el camino de datos de procesador .

Existen muchos diseños de controlador diferentes dependiendo de las operaciones que realizará el camino de datos. Para esta práctica veremos dos funcionalidades sencillas que podemos realizar con este camino de datos sencillo.

La primera funcionalidad es la suma de dos listas de enteros. Imaginemos que hemos cargado dos listas de enteros una en los registros 0 al 7 y otra en los registros 8 al 15. Nuestra aplicación necesita sumar cada elemento de la primera lista al elemento correspondiente de la segunda lista y almacenar dicho resultado en los registros 16 al 23. La siguiente tabla resume la secuencia de registros que debemos enviar al camino de datos para que realice dicha operación.

Señal	Ciclo 0	Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4	Ciclo 5	Ciclo 6	Ciclo 7
A	0	1	2	3	4	5	6	7
B	8	9	10	11	12	13	14	15
ID	16	17	18	19	20	21	22	23

La segunda funcionalidad consiste en acumular la suma de todos los elementos de una suma. Recuperando el ejemplo anterior queremos acumular el resultado de la suma de listas que está almacenado en los registros 16 al 23. Deberemos acumular el resultado en el registro 32. La siguiente tabla muestra la secuencia de registros que deberemos utilizar. Asumiremos que el registro 32 está inicializado a 0.

Señal	Ciclo 8	Ciclo 9	Ciclo 10	Ciclo 11	Ciclo 12	Ciclo 13	Ciclo 14	Ciclo 15
A	16	17	18	19	20	21	22	23
B	32	32	32	32	32	32	32	32
ID	32	32	32	32	32	32	32	32

Vease como el registro 32 es utilizado como registro fuente y registro resultado de todas las operaciones. Y por lo tanto existe una dependencia entre cada una de las operaciones. Al final del ciclo 7 se escribe el resultado de sumar los 8 registros en el registro 32.

Para la implementación del primer comportamiento descrito (suma de listas) anteriormente utilizaremos un autómata sencillo de dos estados. Este autómata tendrá varios contadores (autómatas) subordinados al autómata principal. A continuación se muestra el grafo de estados del autómata principal.

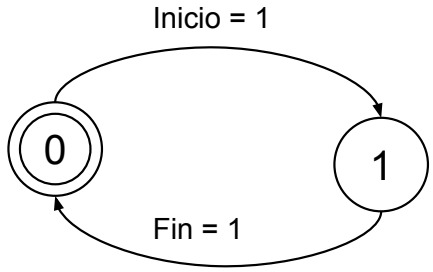

Figura 9: Autómata principal del módulo de control.

Los autómatas subordinados (contadores) están especificados de forma similar al primer automata descrito en la práctica. A continuación se muestra la descripción del módulo de control.

```
module Controlador (
input logic         clk_i,        // Senyal de reloj
input logic         rstn_i,       // Senyal negada de reset

input  logic        inicio_i,     // Senyal de inicio
output logic        fin_o,        // Senyal de fin

output logic [4:0]  id_o,         // Id de registro de escritura
output logic        we_o,         // Senyal de permiso de escritura
output logic [4:0]  A_o,          // Id de registro de lectura A
```

```
output logic [4:0]  B_o           // Id de registro de lectura B

);

// Instanciamos el estado del automata
logic estado_q, proximo_estado_d;

// Instanciamos la senyal interna de finalización
logic next_fin;

// Lógica de próximo estado
always_comb begin
    proximo_estado_d = 'b0; // Valor por defecto

    if (!rstn_i) begin  // Transición a 0 en reset
        proximo_estado_d = 'b0;
    end else if ((estado_q == 1'b0) & (inicio_i)) begin // Condición de inicio
        proximo_estado_d = 'b1;
    end else if (next_fin) begin // Condición de finalización
        proximo_estado_d = 'b0;
    end else if (estado_q == 1'b1) begin  // Estado de operación
        proximo_estado_d = 'b1;
    end // else toma el valor defecto 'b0
end

// Actualización de automata principal
always_ff @(posedge clk_i, negedge rstn_i) begin
    if (!rstn_i) begin // Reset
        estado_q <= 'h0;
    end begin
        estado_q <= proximo_estado_d;
    end
end

// Instanciamos los contadores
logic [4:0] cnt_q, proximo_cnt_d;   // contador de pasos
logic [4:0] idA_q, proximo_idA_d;   // contador de id A
logic [4:0] idB_q, proximo_idB_d;   // contador de id B
logic [4:0] id_q,  proximo_id_d;     // contador de id escritura

// Lógica de proximo estaod para los contadores
always_comb begin
    // Completar
end

// Actualización de los automatas secundarios
always_ff @(posedge clk_i, negedge rstn_i) begin
    if (!rstn_i) begin // Reset
        cnt_q <= 'h0;
        idA_q <= 'h0;
        idB_q <= 'h0;
        id_q  <= 'h0;
    end begin
        cnt_q <= proximo_cnt_d;
```

```
        idA_q <= proximo_idA_d;
        idB_q <= proximo_idB_d;
        id_q  <= proximo_id_d;
    end
end

// Lógica de finalización. (Se puede simplificar)
always_comb begin
    next_fin = 'b0; // Valor por defecto

    if (!rstn_i) begin // Valor en reset
        next_fin = 'b0;
    end else if (estado_q == 1'b0) begin // Valor en el estado 0
        next_fin = 'b0;
    end else if ((estado_q == 1'b0) & (cnt_q == 4'b0000)) begin // Condición de finalización
        next_fin = 'b1;
    end // Cualquier otra combinación
end

// Asignación de condición de fin
assign fin_o = next_fin;

// Asignación de registro de escritura
assign id_o = id_q;

// Asignacion de permiso de escritura
assign we_o = estado_q;

// Asignación de registros de lectura
assign A_o = idA_q;
assign B_o = idB_q;

endmodule
```

5. Test de Verificacion

En esta práctica no dispondréis de un test de verificación para la cola FIFO. Para poder comprobar el correcto funcionamiento de la cola FIFO deberéis implementar vosotros mismos un test que compruebe los diferentes casos. En cambio, sí que disponeis de un banco de pruebas para comprobar el correcto funcionamiento del camino de datos y de vuestra unidad de control.

6. Trabajo a realizar

1) Completad el esquema RTL de la cola FIFO. Para facilitaros la práctica os proveemos del fichero 'buffer_circular.sv'. **Debéis entregar dicho módulo al final de la práctica.**

2) Implementad un banco de pruebas para comprobar el correcto funcionamiento del buffer circular implementado en el apartado anterior. Para ello, deberéis llamar al banco de pruebas '*tb_buffer_circular.sv*'. **Debéis entregar dicho módulo al final de la práctica.**

3) Implementad un módulo de control para un camino de datos. La secuencia de operaciones realizará la acumulación de la suma de los registros 1 al 6 del banco de registros en el registro 0 que incialmente contiene un 0. Para ello, deberéis completar el módulo '*controlador.sv*'. **Debéis entregar dicho módulo al final de la práctica.**

4) Implementad el camino de datos instanciando los módulos del banco de registros, sumador y controlador. La secuencia de operaciones realizará la acumulación de la suma de los 12 primeros registros del banco de registros. Para ello, deberéis crear el módulo '*controlador.sv*'. Disponéis de los ficheros 'tb_camino_datos', 'banco_registros.sv', 'sumador.sv' y 'camino_datos'. **Debéis entregar dicho módulo al final de la práctica.**