Universidad de Costa Rica

Facultad de Ingeniería

Escuela de Ingeniería Eléctrica

IE-0523 Circuitos Digitales II

SD Host Controller

SD Host Controller

Profesor:

Enrique Coen Alfaro

Estudiantes:

Alberto Alfaro Degracia, B40167
Juan José Delgado Quesada, B42250
Jose Pablo Delgado Chinchilla, B32241
Leonardo Hernández Chacón, B43262

Ciudad Universitaria Rodrigo Facio
II-2016

Índice de contenidos

Índice de figuras

Índice de cuadros

1. Introducción

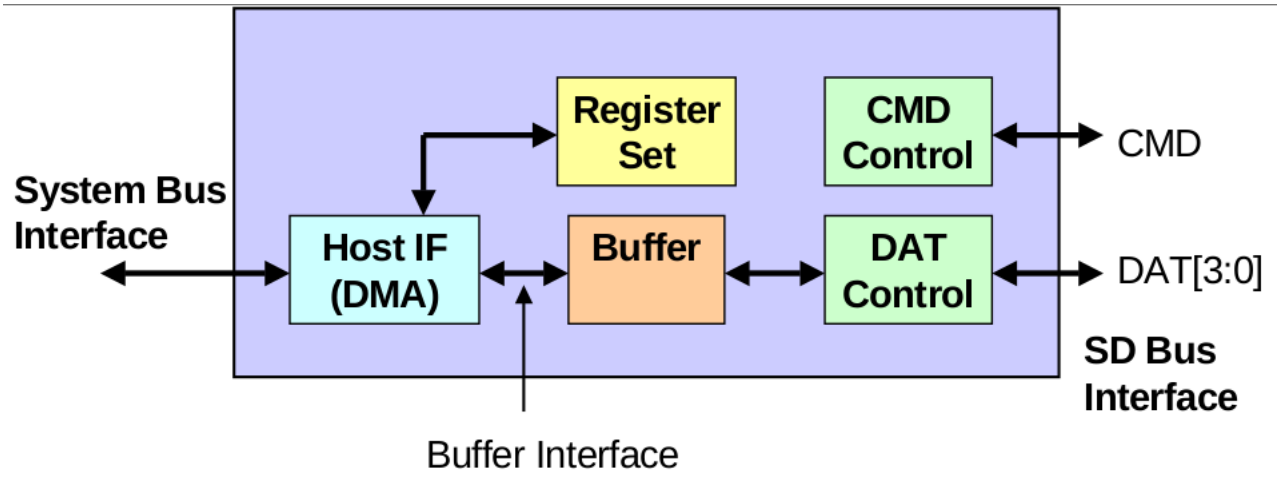

Figura 1: Partes del SD Host

2. CMD Control

2.1. Arquitectura

Para el diseño del CMD Control se crearon varios sub-bloques que realizaran tareas pequeñas del controlador para luego instanciarlos todos en un único bloque llamado "CMD CONTROL", por lo cual para cada sub-bloque se desarrollo un probador y un testbench que verificara el correcto funcionamiento de este. En la figura 2 se observa la división en sub-bloques utilizada. La función principal de cada uno de los sub-bloques es :

- **Control CMD** : Control interno del CMD Control que se encarga de administrar la ejecución de todo el proceso de transmisión desde que se da un comando del CPU hasta que se escribe la respuesta en registros.
- **Control Capa Física** : Controlador que trabaja en el dominio del reloj de la SD Card. Se encarga de controlar el correcto funcionamiento de los wrappers y del PAD.
- **Paralelo a Serie** : Wrapper encargado de recibir la el comando por enviar de CPU (en paralelo) y serializar los datos para poder transmitirlos a la SD Card.
- **Serie a Paralelo** : Wrapper encargado de recibir la respuesta proveniente de la SD Card (en serie) y deserializar los datos para poder transmitir la respuesta a los registros.
- **PAD** : Buffer de tercer estado que permite seleccionar entre activar la salida del CMD Control o activar su entrada, colocando la que no esta en uso en alta impedancia.

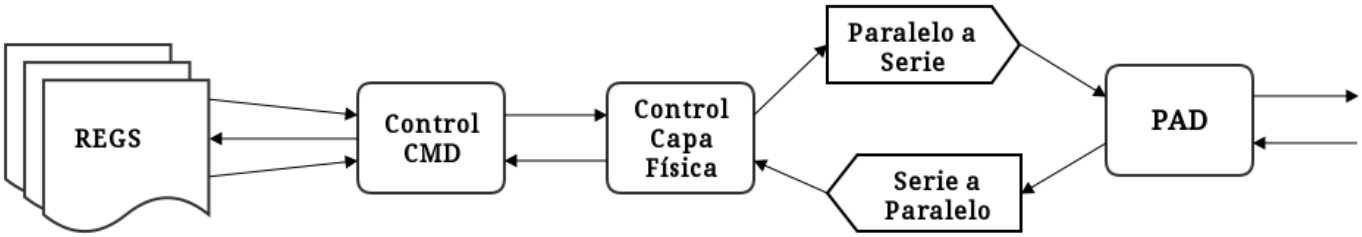

Figura 2: Partes del CMD Control

En la figura 3 se muestra la máquina de transición que describe el funcionamiento del bloque "Control CMD", el funcionamiento de cada uno de ellos es el siguiente :

1. **Reset** :
   - Es el estado por defecto.
   - Coloca todas las salidas en bajo.
   - Automáticamente prosigue al estado IDLE.
2. **IDLE** :
   - Espera a que "New command" se afirme para iniciar el proceso y pasar a "Setting Outputs".
   - Estado por defecto al ocurrir un error por lo que "idle out" se encuentra afirmada.
3. **Setting Outputs** :
   - Se activa "Strobe out" para indicar el uso de la capa física.
   - Se forma el "cmd out" utilizando las entradas "cmd argument" y "cmd index".
   - Automáticamente se prosigue al estado processing.
4. **Processing** :
   - Constantemente esperando la señal que indica que la capa física terminó su proceso (strobe in).
   - Colocar la respuesta de la SD Card a la salida "Response".
   - Esperar la afirmación de ack in para pasar al estado "IDLE".

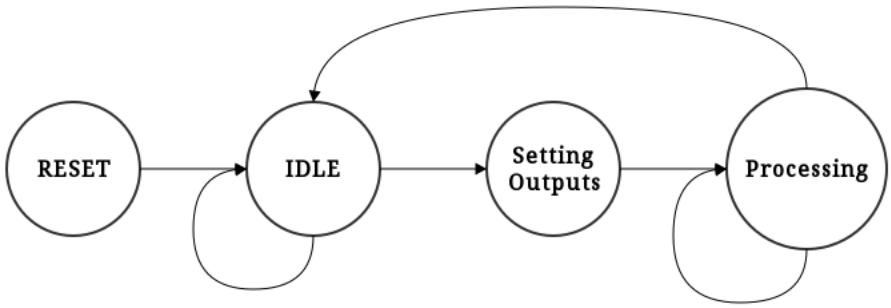

Figura 3: Transición de estados del control interno del CMD

En la figura 4 se muestra la máquina de transición que describe el funcionamiento del bloque "Control Capa Física", el funcionamiento de cada uno de ellos es el siguiente :

1. **Reset** :
   - Es el estado por defecto.
   - Coloca todas las salidas en bajo.
   - Automáticamente prosigue al estado IDLE.
2. **IDLE** :
   - Se reinician los wrappers (reset wrapper = 1).
   - Se espera a que la señal strobe in sea afirmada lo que indica que inicia un nuevo proceso de la capa física pasando al estado Load Command.
   - Estado por defecto cuando idle in esta afirmada.
3. **Load Command** :
   - Se habilita el wrapper paralelo a serie.
   - Se carga en el wrapper la trama por enviar.
   - Se afirman la salidas pad state y pad enable para activar el pad y ponerlo en modo salida.
   - Automáticamente pasa al estado Send Command.
4. **Send Command** :
   - Se afirma la señal load send para iniciar la transmisión del wrapper.
   - Se continua en este estado hasta que transmission complete sea afirmada por el wrapper pasando al estado Wait Response.
5. **Wait Response** :
   - Se coloca el pad en modo entrada.
   - Se habilita el wrapper de serie a paralelo.
   - Se pasa al estado Send Response cuando se afirma reception complete o no response.
6. **Send Response** :
   - Se afirma la salida strobe out para indicar que se ha completado el proceso de la capa física.
   - Se coloca en response los contenidos de pad response.
   - Automáticamente pasa al estado Wait Ack.
7. **Wait Ack** :
   - Se colocan las salidas en estado bajo.
   - Se espera que ack in sea afirmada para avanzar al estado send ack.

8. **Send Ack** :
   - Se afirma la señal ack out.
   - Automáticamente se pasa al estado IDLE.

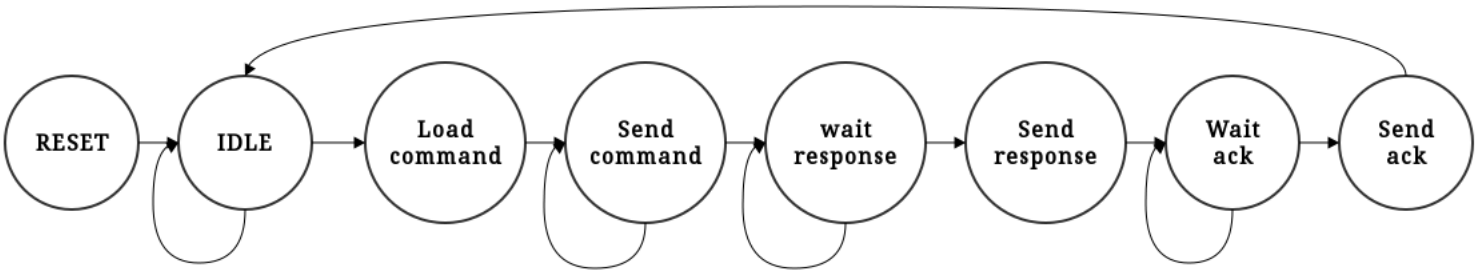

Figura 4: Transición de estados del control capa física

2.2. Resultados

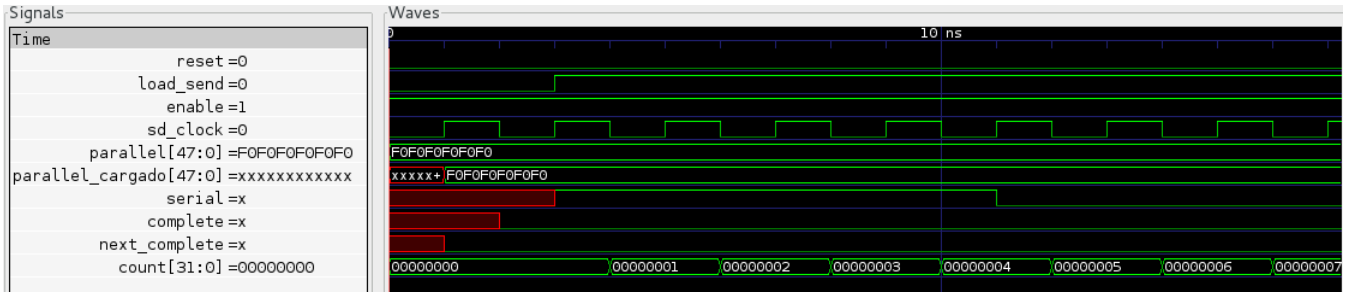

Figura 5: Funcionamiento del wrapper paralelo a serie

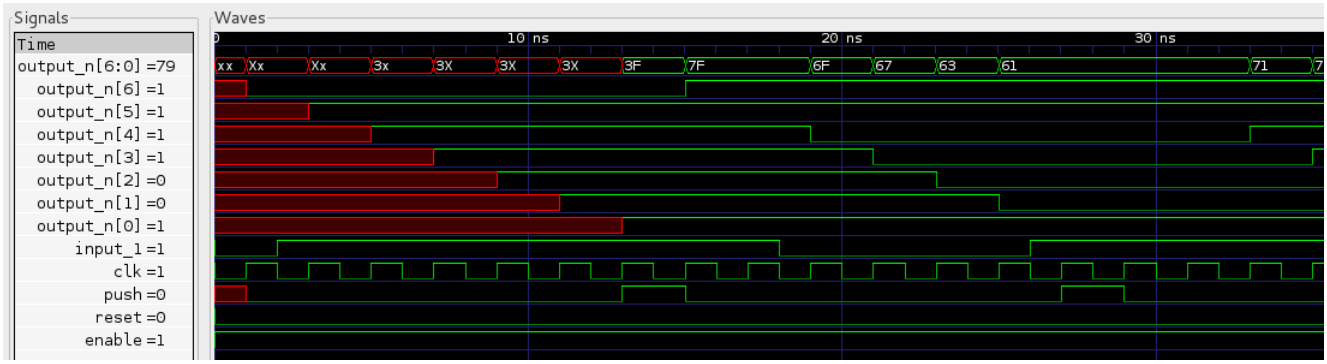

Figura 6: Funcionamiento del wrapper serie a paralelo

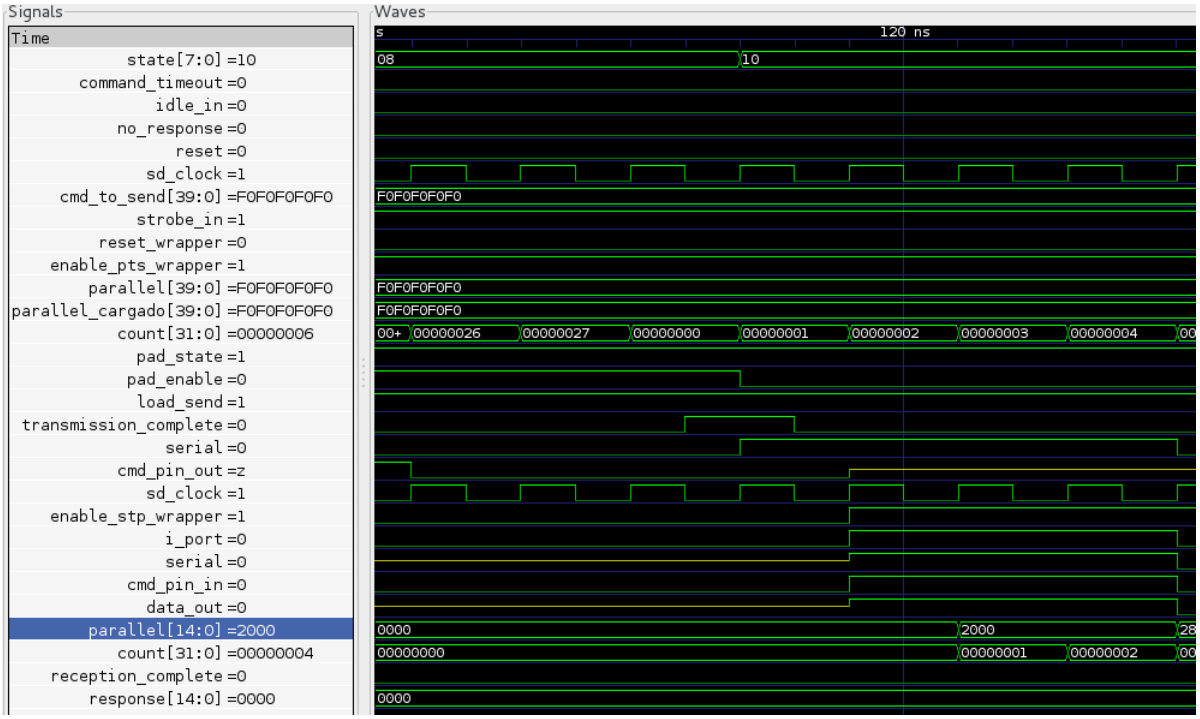

Figura 7: Funcionamiento del control de la capa física

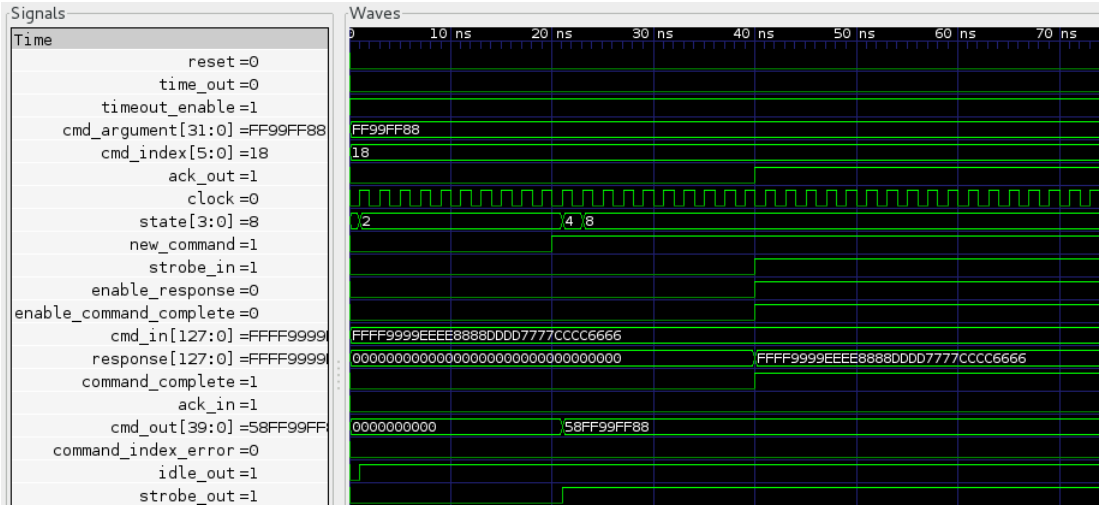

Figura 8: Funcionamiento del control interno

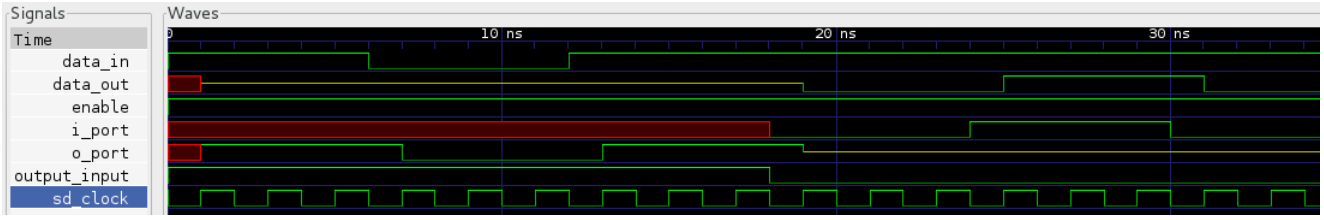

Figura 9: Funcionamiento del PAD

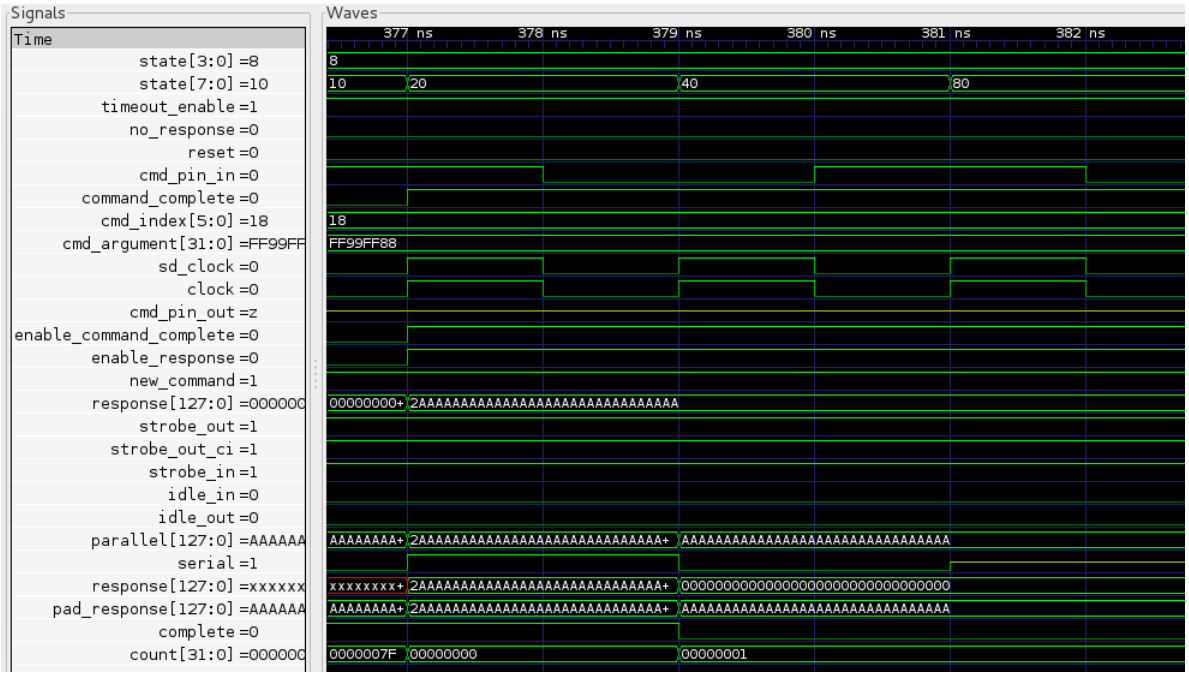

Figura 10: Funcionamiento del bloque final CMD Control

3. SD Host Standard Register

3.1. Standard Implementacion

En la implementacion del SD Host se encuentra el bloque de registros, en este se guarda toda la información pertinente para el estado y control de bloques internos de este.
Este consta en una memoria de 256 bytes la cual va a estar principalmente dividida en bloque de 32 bits, 16 bits y 8bits trabajando así con el standard de escrituras y lecturas que puede realizar el CPU al controlador, en cada uno de estos espacios de memoria contiene tanto información especifica del controlador así como espacios asignados para las características de la tarjeta SD esto con el objetivo de modular todas las variables necesarias y tener los limites claros de donde llegar.

Offset	15-08 bit	07-00 bit	Offset	15-08 bit	07-00 bit
002h	SDMA System Address (High)		000h	SDMA System Address (Low)	
006h	Block Count		004h	Block Size	
00Ah	Argument1		008h	Argument0	
00Eh	Command		00Ch	Transfer Mode	
012h	Response1		010h	Response0	
016h	Response3		014h	Response2	
01Ah	Response5		018h	Response4	
01Eh	Response7		01Ch	Response6	
022h	Buffer Data Port1		020h	Buffer Data Port0	
026h	Present State		024h	Present State	
02Ah	Wakeup Control	Block Gap Control	028h	Power Control	Host Control
02Eh	Software Reset	Timeout Control	02Ch	Clock Control	
032h	Error Interrupt Status		030h	Normal Interrupt Status	
036h	Error Interrupt Status Enable		034h	Normal Interrupt Status Enable	
03Ah	Error Interrupt Signal Enable		038h	Normal Interrupt Signal Enable	
03Eh	---		03Ch	Auto CMD12 Error Status	
042h	Capabilities		040h	Capabilities	
046h	Capabilities (Reserved)		044h	Capabilities (Reserved)	
04Ah	Maximum Current Capabilities		048h	Maximum Current Capabilities	
04Eh	Maximum Current Capabilities (Reserved)		04Ch	Maximum Current Capabilities (Reserved)	
052h	Force Event for Error Interrupt Status		050h	Force Event for Auto CMD12 Error Status	
---	---		054h	---	ADMA Error Status
05Ah	ADMA System Address [31:16]		058h	ADMA System Address [15:00]	
05Eh	ADMA System Address [63:48]		05Ch	ADMA System Address [47:32]	
---	---		---	---	
0F2h	---		0F0h	---	
---	---		---	---	
0FEh	Host Controller Version		0FCh	Slot Interrupt Status	

Register map

Register Attribute	Description
RO	Read-only register: Register bits are read-only and cannot be altered by software or any reset operation. Writes to these bits are ignored.
ROC	Read-only status: These bits are initialized to zero at reset. Writes to these bits are ignored.
RW	Read-Write register: Register bits are read-write and may be either set or cleared by software to the desired state.
RW1C	Read-only status, Write-1-to-clear status: Register bits indicate status when read, a set bit indicating a status event may be cleared by writing a 1. Writing a 0 to RW1C bits has no effect.
RWAC	Read-Write, automatic clear register: The Host Driver requests a Host Controller operation by setting the bit. The Host Controllers shall clear the bit automatically when the operation of complete. Writing a 0 to RWAC bits has no effect.
HwInit	Hardware Initialized: Register bits are initialized by firmware or hardware mechanisms such as pin strapping or serial EEPROM. Bits are read-only after initialization, and writes to these bits are ignored.
Rsvd	Reserved. These bits are initialized to zero, and writes to them are ignored.
WO	Write-only register. It is not physically implemented register. Rather, it is an address at which registers can be written.

Register Types

Figura 11: Register File Structure

Como se ve anteriormente en la figura 11 notamos la distribución de los registros asi como

los atributos que tiene cada uno cabe destacar que cada registro por dentro se compone de bits separados que tienen un significado especifico y solo es escrito por un bloque.

3.2. Implementacion

Teniendo todo esto en mente se implementaron los registros individuales esto tanto porque ciertos registros para el fin de esta prueba eran irrelevantes, la comunicaciones especifica con los bloques mejora de manera exponencial ya que se realiza de manera transparente para el bloque que este haciendo el acceso este lo nota como un espacio donde tiene cierta información guardada que puede accesar siempre y puede escribir cuando este listo. De manera que para realizar los 64 registros se realizo un template a seguir que se puede ver en la figura 12

```
module reg_template(
    clk,
    rst,

    ack,
    enb_block0,
    enb_block1,
    enb_block2,

    CommandIndex_in,
    CommandType_in,
    DataPresentState_in,
    CommandIndezCheckEnable_in,
    CommandCRCCheckEnable_in,
    ResponseTypeSelect_in,

    CommandIndex_out,
    CommandType_out,
    DataPresentState_out,
    CommandIndezCheckEnable_out,
    CommandCRCCheckEnable_out,
    ResponseTypeSelect_out
    );
//Parameters before
parameter width = 16;
```

Instancia del template

```
//REG SIZE
wire [(width-1):0] data_in;

//WIRES
    //Regular Blocks
    wire rst;
    wire clk;
    wire enb_block0;
    wire enb_block1;
    wire enb_block2;
    reg ack;
    //INPUTS
    wire    [4:0] CommandIndex_in;
    wire    [1:0] CommandType_in;
    wire    DataPresentState_in;
    wire    CommandIndezCheckEnable_in;
    wire    CommandCRCCheckEnable_in;
    wire    [1:0] ResponseTypeSelect_in;

    //OUTPUTS
    wire    [4:0] CommandIndex_out;
    wire    [1:0] CommandType_out;
    wire    DataPresentState_out;
    wire    CommandIndezCheckEnable_out;
    wire    CommandCRCCheckEnable_out;
    wire    [1:0] ResponseTypeSelect_out;

//REG
    //OUTPUTS
    reg [(width-1):0] data_out;
```

Entradas del registro

Figura 12: Implementacion del Registro 1

En este se ve como cada bloque empieza con un CLK, un Reset, un ack, enables, y las entradas y salidas de cada registro especifico . Se declaran las variables necesarias y llegar al puerto de data in cuyo tamaño depende de un parámetro por lo que en teoría se pueden realizar registros de cualquier tamaño, después de esto tal y como se ve en la figura 13 notamos como se da la asignación al puerto para formar un tipo de mascara que separe la salida completa en salidas mas pequeñas que sea mas funcional para los otros bloques con que se va a comunicar.

```
//REG INPUT OUTPUT ASSIGNS
    //RESERVED
    assign data_in [15:14] = 2'b00 ;
    assign data_in [2] = 2'b00 ;

    //INPUTS
    assign data_in [13:8] = CommandIndex_in;
    assign data_in [7:6] =  CommandType_in;
    assign data_in [5] =    DataPresentState_in;
    assign data_in [4] =    CommandIndezCheckEnable_in;
    assign data_in [3] =    CommandCRCCheckEnable_in;
    assign data_in [1:0] =  ResponseTypeSelect_in;

    //OUTPUTS
    assign  CommandIndex_out    = data_out [13:8];
    assign  CommandType_out = data_out [7:6];
    assign  DataPresentState_out    = data_out [5];
    assign  CommandIndezCheckEnable_out = data_out [4];
    assign  CommandCRCCheckEnable_out   = data_out [3];
    assign  ResponseTypeSelect_out  = data_out [1:0];
```

Asignación de las entradas al template

```
//Bloque secuencial
always @(posedge clk) begin
    if (rst) begin
        // reset
        data_out <= 32'b0;
    end
    else if (enb_block0 || enb_block1 || enb_block2) begin
        data_out <= data_in;
    end
    else begin
        data_out <= data_out;
    end
end


//Bloque combinacional
always @(*) begin
    if (data_in == data_out) begin
        ack = 1'b1;
    end
    else begin
        ack = 1'b0;
    end
end
```

Código para el ack y Flip Flop

Figura 13: Implementacion del Registro 2

Finalmente el código que representa el funcionamiento del registro se basa en un flipflop que toma la entrada y la pasa a la salida dependiendo del flanco del reloj así como los enables activados y el ack el cual corresponde a una bandera que se levanta cuando la salida es igual a la entrada esto con el objetivo de informarle al bloque que ya se escribió el dato.

3.3. Resultados

El probador fue bastante simple de implementar todos los registros se basan del template por lo que se opto por la prueba de escritura y lectura de este con esto en mente podemos ver en la figura 14

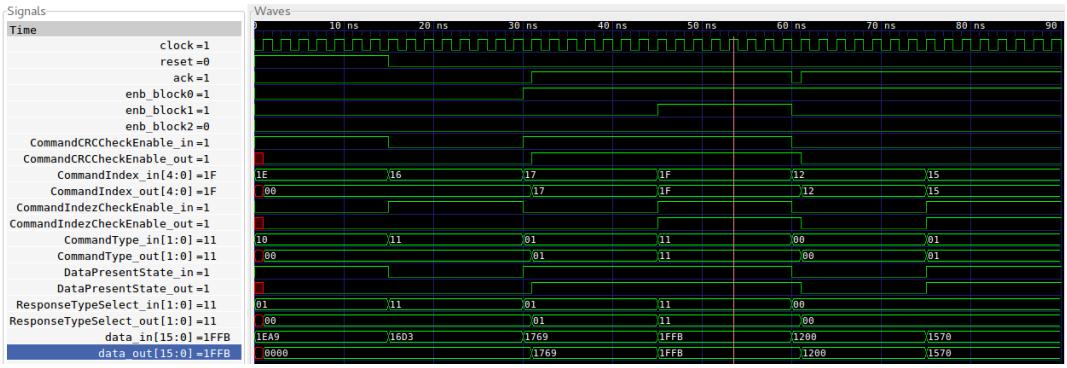

Figura 14: Resultado comprobación de registros

Se notan retrasos pero estos son causador por que las senales llegaban en el flanco negativo esto para simular cualquier tipo de de ingreso de datos pero notamos que los datos pasan bien a la salida, de la misma manera el comportamiento del ack y los enables funciona como esta estipulado.

3.4. Síntesis

Se realizo una serie de commandos con el objetivo de sintetizar los registros de manera que se sintetizo el archivo completo que tiene todos los registros para de una vez estar seguro que no existiera ningun error y el hecho de hacer cada registro por separado ayudo de manera que si habia un problema que corregir todos los demas eran exactamente iguales.

De la misma manera se utilizo el probador ya implementado al reg sintetizado y como podemos ver por la figura 16 los resultados concuerdan con los del conductual sin ningún problema solo existen unos retrasos presentes pero esto lo soluciona el ack con el protocolo de comunicación

```
read_verilog -sv regs.v
hierarchy; proc; fsm; opt; memory; opt;
techmap; opt
dfflibmap -liberty cmos_cells.lib
abc -liberty cmos_cells.lib
clean
write_verilog -noattr sintetizado.v
```

Figura 15: Yosys archivo

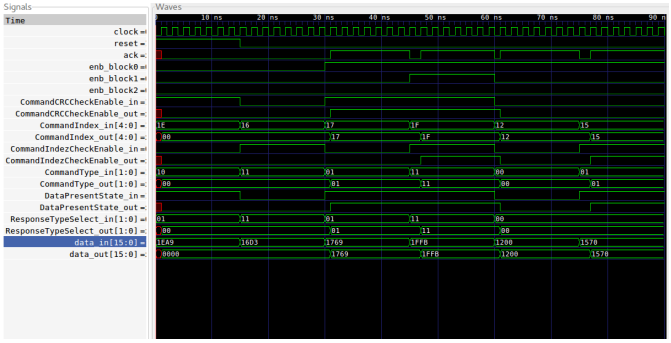

Figura 16: Resultados Sintetizados

4. DMA

4.1. Descripción

DMA se traduce de sus siglas en inglés como acceso directo a memoria (*Direct Memory Access*) y es básicamente un algorítmo de transferencia de datos. Existen diferentes protocolos de DMA, el más sencillo es SDMA (*Single Operation DMA*) sin embargo este tiene como desventaja que tras cada acceso se genera un *DMA_Interrupt* y se interrumpe al CPU, y a la larga esto termina siendo ineficiente. Como solución a este problema, nace el ADMA (*Advanced DMA*) a partir de la implementación por parte del Host Driver de una tabla en memoria denominada Descriptor Table, que incluye la información necesaria para cada acceso en sus líneas, y así se evita la interacción con el CPU. Finalmente, existen dos tipos de ADMA: ADMA1 y ADMA2, cuya diferencia se da en la manera de realizar la transferencia. DMA1 solamente soporta transferencias de datos de 4KB, mientras que DMA2 es más flexible pues soporta transferencias de datos de cualquier tamaño, siempre y cuando se cumpla con el alineamiento en memoria. El estándar del SD Host recomienda utilizar ADMA2, y fue el protocolo utilizado en este proyecto.

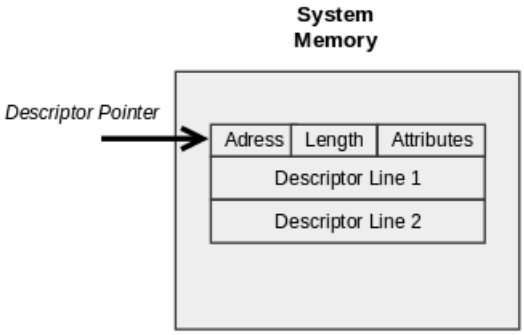

Figura 17: Descriptor Table.

En la figura 17, se muestra una ilustración del Descriptor Table, básicamente este bloque recibe como entrada la dirección de memoria donde debe buscar, que además va a estar contenida en un registro interno del DMA (*System _Address_Register*) de 64 bits, luego este bloque devuelve como salida el Descriptor Line correspondiente. Como se observa en la ilustración el Descriptor Line esta conformado por un conjunto de bits para la dirección del dato para la transferencia (que se guarda en el registro interno *Data_Address_Register*), bits para el tamaño del dato (*Data_Length_Register*) y un grupo de atributos, veremos que definen las transiciones de estados. Cabe señalar que la cantidad de bits para atributos y el tamaño del dato son fijos, sin embargo se pueden dar direcciones del dato de 32 o 64 bits, para este proyecto se escogió direcciones de 64 bits dado también soporta direcciones de 32 bits (utilizando los bits menos significativos), lo cual condicionó el tamaño de los Descriptor Line a 96 bits.

La distribución del Descriptor Line se muestra a continuación:

Address Field		Length		Reserved		Attribute					
95	32	31	16	15	06	05	04	03	02	01	00
64-bit Address		16-bit Length		000000		Act2	Act1	0	Int	End	Valid

Figura 18: Descriptor Line.

Donde:

Valid	Valid=1 Indica que la línea del descriptor es efectiva. Valid=0 Activa señales de ADMA Error Interrupt y stop ADMA.
End	End=1 indica el final del Descriptor Line. Cuando se completa el descriptor Line, Transfer Complete Interrupt se activa.
Int	Int=1 genera DMA Interrupt cuando el Descriptor Line se completa.

Cuadro 1: Valid, End y Int.

Act2	Act1	Símbolo	Comentario	Operación
0	0	Nop	No Operation	No ejecuta la línea, salta a la siguiente.
0	1	rsv	reserved	No ejecuta la línea, salta a la siguiente.
1	0	Tran	Transfer Data	Transferencia de un Descriptor Line
1	1	Link	Link Descriptor	Enlaza otra dirección.

Cuadro 2: Act.

Esto se implementa en Verilog con un case donde dependiendo del Act se define el SYS_ADR y Tran como sigue, en seguida se verá que este paso ocurre en el estado de cambio de dirección:

```
if(Present_State==ST_CADR)  begin
        case(Act)
                2'b00:  begin   //NOP
                        SYS_ADR <= SYS_ADR+ 8;
                        Tran=0;
                        end

                2'b01:  begin  //RSV
                        SYS_ADR <= SYS_ADR+ 8;
                        Tran=0;
                        end

                2'b10:  begin  //TRAN
                        SYS_ADR <= SYS_ADR+ 8;
                        Tran=1;
                        end

                2'b11:  begin  //LINK
                        SYS_ADR <= DAT_ADR;
                        Tran=0;
                        end
                default: begin
                        SYS_ADR <= SYS_ADR;
                        Tran<=Tran;
                         end

                endcase
end
```

Figura 19: Act en Verilog.

Luego, se implementó la siguiente máquina de estados para la transferencia:

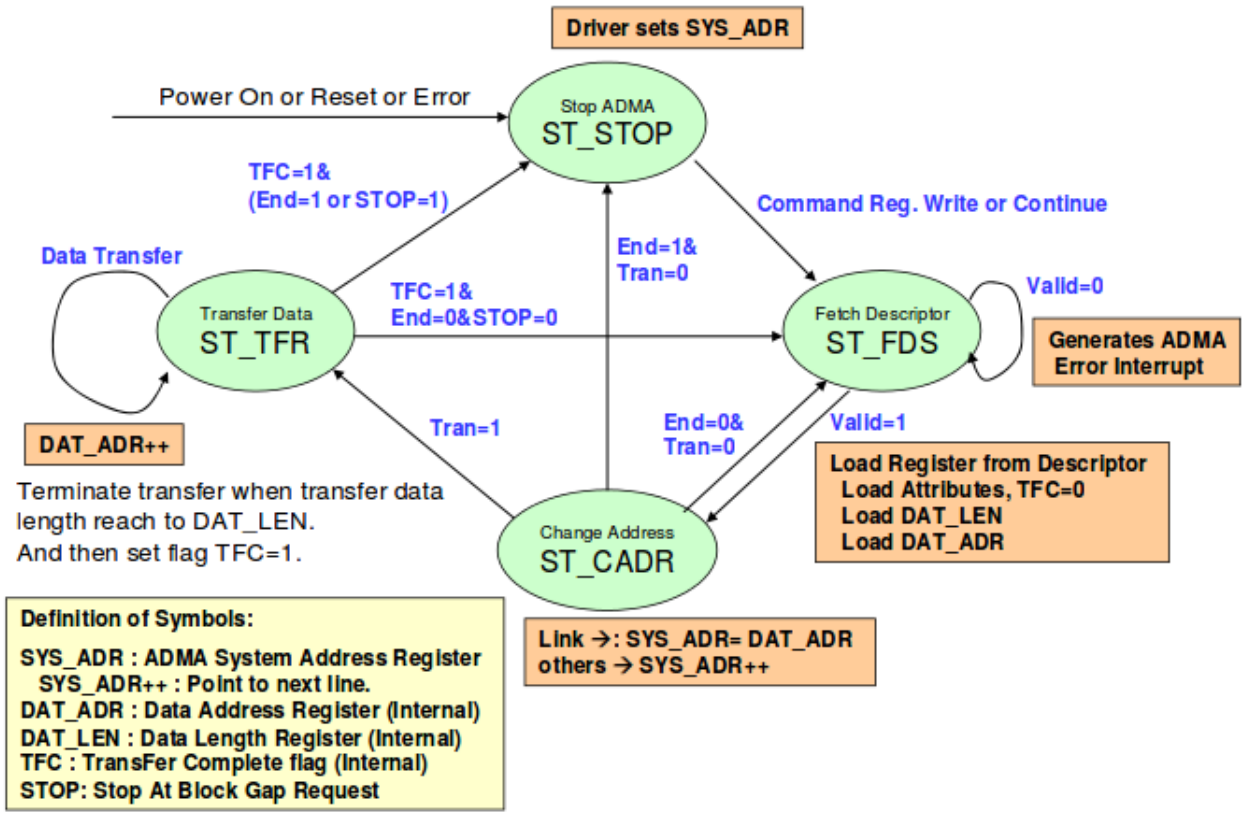

Figura 20: Estados y Transición.

Donde:

Nombre del Estado	Operación
ST_FDS (Fetch Descriptor)	Se accesa el Descriptor Table y se carga el Descriptor Line, con ello los atributos y se actualizan los registros internos del DMA (DAT_ADR y DAT_LEN).
ST_CADR (Change Address)	De acuerdo a Act, se define el SYS_ADR. ADMA2 no puede detenerse en este estado, incluso ante señales de interrupción.
ST_TFR (Transfer Data)	Se da la transferencia entre System_Memory y la SD Card.
ST_STOP (Stop DMA)	En caso de que se detenga el DMA. Es el estado de inicio de la máquina.

Cuadro 3: Estados.

En seguida se muestra la designación de estados utilizados en el bloque ADMA, para la transición de estados, dirección de transferencia y tipos de transferencia. Los estados de dirección de transferencia y tipo de transferencia serán explicados más adelante. Se designó de forma One Hot.

```
parameter ST_STOP   = 4'b0001; //Estados
parameter ST_FDS    = 4'b0010;
parameter ST_CADR   = 4'b0100;
parameter ST_TFR    = 4'b1000;

parameter WAIT= 4'b0001; //DirecciónTransferencia
parameter CARD_TO_HOST= 4'b0010;
parameter HOST_TO_CARD= 4'b0100;

parameter Single_Transfer   = 2'b00; //Tipos de Transferencia
parameter Infinite_Transfer   = 2'b01;
parameter Multiple_Transfer   = 2'b10;
parameter Stop_Multiple_Transfer   = 2'b11;
```

Figura 21: Estados oh.

Se implementó los siguientes módulos, cuyo funcionamiento se procede a explicar.

- ADMA: Es el bloque principal, dado que este bloque se encarga de la comunicación con los registros que necesita el DMA(tanto lectura como escritura del bloque de registros), define la transición de estados y además maneja las banderas y salidas del DMA, e instancia otros bloques dentro de sí.
- Descriptor_Table: Simula una RAM, es muy simple dado que consiste en que ante una entrada SYS_ADR, retorna una salida Descriptor Line.
- TipoDeTransferencia: Lógica que indica si la transferencia se da en modo de trama o multitrama.
- Transferencia: Se encarga de la transferencia tanto del SD al Card como del Card al Sd.
- aFIFO: FIFO asincrónico, necesario para la comunicación con el bloque de DAT.

4.2. Pruebas

Inicialmente procedemos a probar la transición de estados.

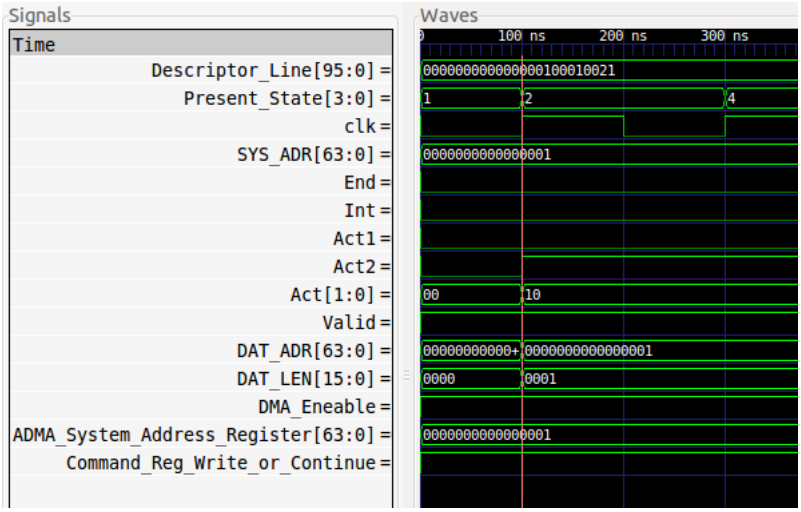

Figura 22: Transición de estados, primera prueba.

Para facilitar la explicación, en seguida mostramos los bits del Descriptor_Line, según como los da el módulo de Descriptor_Table.

```
Descriptor_Line[0]<=1;          //Valid
Descriptor_Line[1]<=0;          //End
Descriptor_Line[2]<=0;          //Int
Descriptor_Line[3]<=0;
Descriptor_Line[4]<=0;          //Act1
Descriptor_Line[5]<=1;          //Act2
Descriptor_Line[15:6]<=0;       //Rsv
Descriptor_Line[31:16]<= 1;     //Length
Descriptor_Line[95:32]<= 1;     //Address
```

Figura 23: Bits del Descriptor Line, primera prueba.

Nótese que los atributos y las señales de Data_Address y Data_Length cambian hasta que el Present_State es el estado de Fetch, como indica el estándar, además puede verse que se asignó los valores esperados a cada atributo.

Luego como Command_Reg_Write_or_Continue está en alto puedo pasar del estado ST_STOP a ST_FDS. Como Valid está en 1 paso al estado ST_CADR y como Act es 10, paso al estado de transferencia.

Ahora procedemos a cambiar algunos valores del Descriptor Line, para ver el comportamiento de las señales. Inicialmente ponemos el bit de valid=0

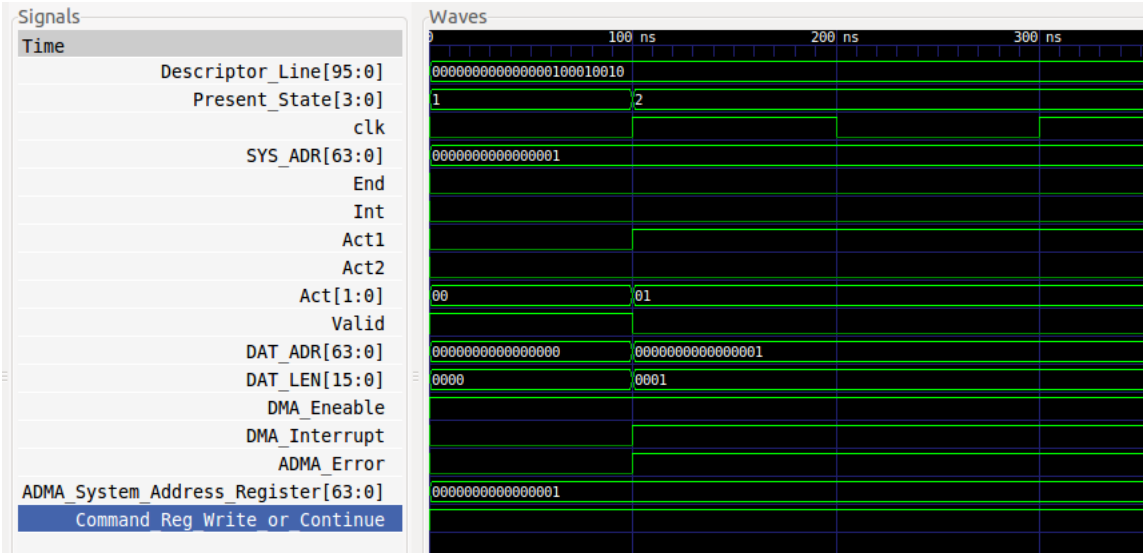

Figura 24: Transición de estados, segunda prueba.

para esta prueba se nota que como valid=0, entonces el estado ST_FDS se mantiene en vez de pasar al estado de ST_CADR, pero además se ponen en alto DMA_Interrupt y ADMA_Error (nótese que no bajan), como lo indica el estándar. Luego de nuevo valid=1 pero ahora Act= 00 de forma que ahora cuando está en el estado de ST_CADR pasa al estado ST_FDS, como se ve en la gráfica y nótese que hay un cambio en SYS_ADR, dado que se le suman 8bits para acceder a la siguiente línea del Descriptor Table.

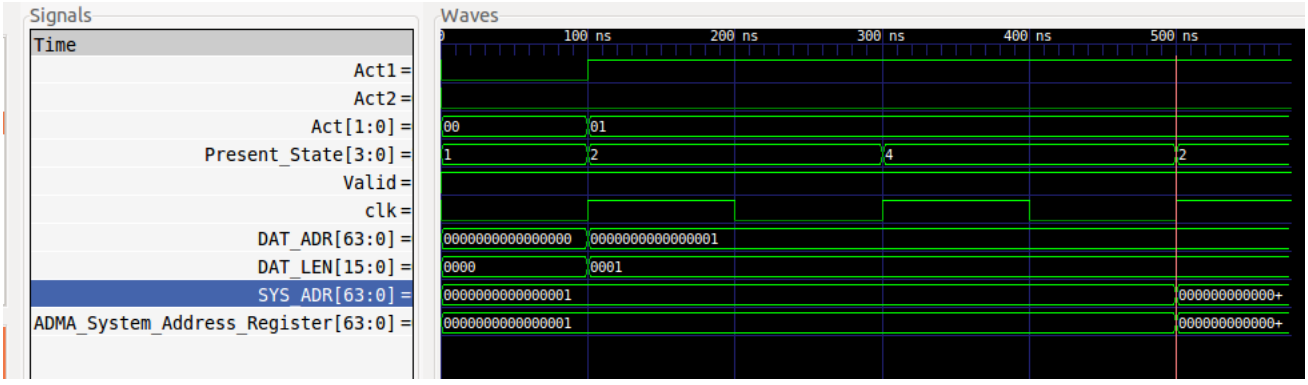

Figura 25: Transición de estados, tercera prueba.

Además, para la comunicación con el bloque de registros se implementó entradas del tipo ack y salidas enb para los registros para ser consistentes con el protocolo de comunicación. Así existe un bloque de lógica que determina el próximo estado, pero la transición se da hasta que se cumpla la transferencia, véase un ejemplo

```
ST_FDS:  begin
         TFC<=0;
                    if(Valid==1)
                                    begin
                                    Next_State <= ST_CADR;
                                    end
                     else
                                    begin
                                    Next_State <= ST_FDS;
                                   end
          end
```

Figura 26: Próximo Estado.

Por ejemplo para el estado ST_FDS se determina el próximo estado y se guarda en Next_State. Luego Para cada estado hay una condición que cumplir para que se proceda a asignar el valor de Next_State a Presente_State. (Esto se da en case diferentes)

```
ST_STOP: begin
                SYS_ADR<=Initial_ADMA_System_Address;
                ADMA_System_Address_Register<=SYS_ADR;
                enb_ADMA_System_Address_Register<=1;

                if(ack_ADMA_System_Address_Register==1)begin

                  Present_State <= Next_State;
                  enb_ADMA_System_Address_Register<=0;
                 end

                else begin
                  Present_State <= Present_State;
                     end
          end
```

Figura 27: Transición de estados, tercera prueba.

En síntesis vemos que si por ejemplo necesito escribir el registro SYS_ADR, levanto la señal de enb y reviso la señal de ack para ese registro(recuerdese que son un enb y un ack por cada registro). Si ack está en alto entonces puedo proseguir con la transición de estados (y bajo el eneable), sino me quedo esperando a que el bloque de registros me avise que ya leyó con éxito el dato.

Luego, uno de los aspectos más importantes es la transmisión con el bloque de datos, para esto se utilizó un buffer asincrónico tipo FIFO, que para más detalles se puede revisar el módulo Buffer. Se utilizó un buffer para lectura(solamente) y uno para escritura(solamente). A continuación se muestra una prueba de los 2 buffers instanciados independientemente uno del otro.

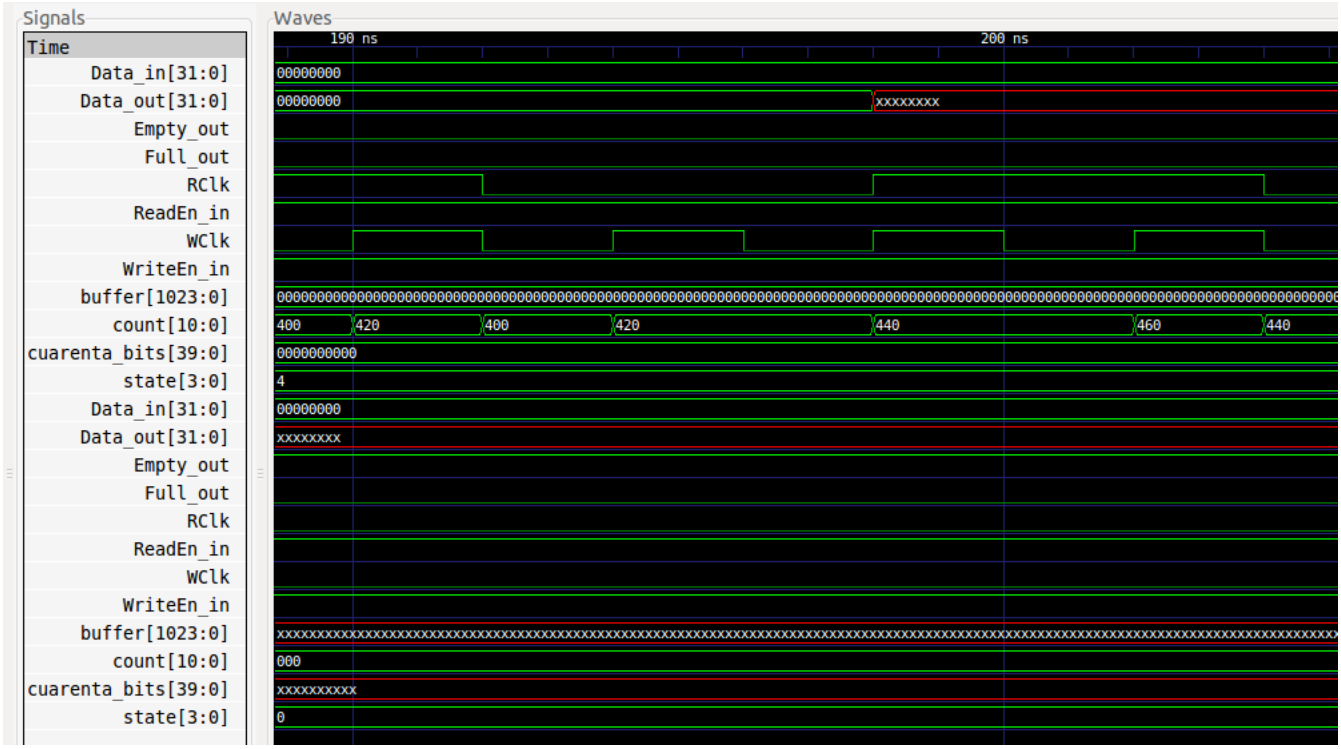

Figura 28: Buffers instanciados.

Se insiste en dar énfasis al módulo Buffer para una explicación más exhaustiva. Ya con esto el siguiente paso del proyecto era sintetizar con yosis.

5. Módulo Buffer

El módulo de buffer corresponde a una estructura tipo FIFO, que permite lectura y escritura a diferentes frecuencias de reloj. Este se implementó con un registro desplazante y un contador, operando como puntero. Al escribir datos, se van desplazando desde el puerto de escritura a la derecha y se aumenta el valor del puntero, que parte de la posición inicial del registro desplazante. En la figura 29 se muestra la estructura a grandes rasgos del buffer, con registros desplazantes que almacenan una cantidad de datos igual al ancho (width) del buffer, para el reloj de escritura y un puntero que hace lectura del dato de salida usando el reloj de lectura.

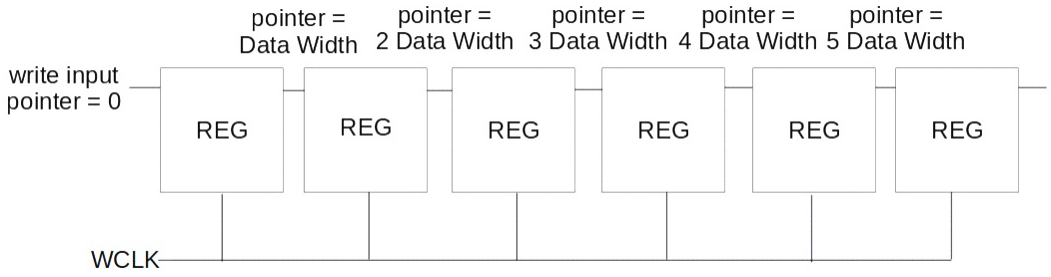

Figura 29: Estructura del módulo de buffer

Se requiere de una salida de lleno y casi lleno para asegurar que se produzcan lecturas y escrituras validas. Se generan varias banderas a partir del valor del puntero, que actúa como un contador,

ademas de señales producidas a partir de las condiciones de la maquina de estados del Buffer.

En la figura **??** se muestra una breve simulación del comportamiento del buffer y lectura y escritura con relojes distintos.

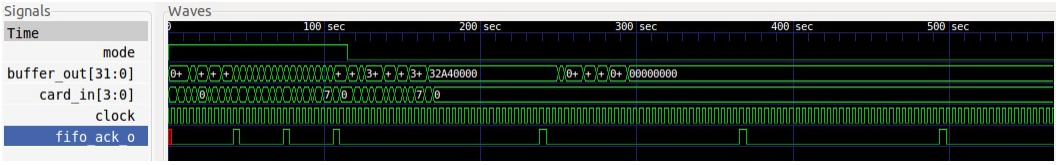

Figura 30: Simulación de lectura y escritura del buffer con relojes distintos

5.1. Prueba

Se hizo una pequeña prueba con dos dominios de reloj distintos para el módulo. Se muestra en la siguiente imagen:

6. Módulo DAT

El módulo de DAT es muy similar al módulo de CMD. Se distingue porque se encarga de enviar información a la tarjeta SD a través de 1 o 4 terminales de 1 bit, para modo de operación de trama y multitrama.
Se presenta en la figura 31 la transición de estados del módulo DAT. Su principal función es la de generar salidas para comunicación con el FIFO y banderas de habilitación y selección para los módulos que reciben la señal del FIFO y la tarjeta.

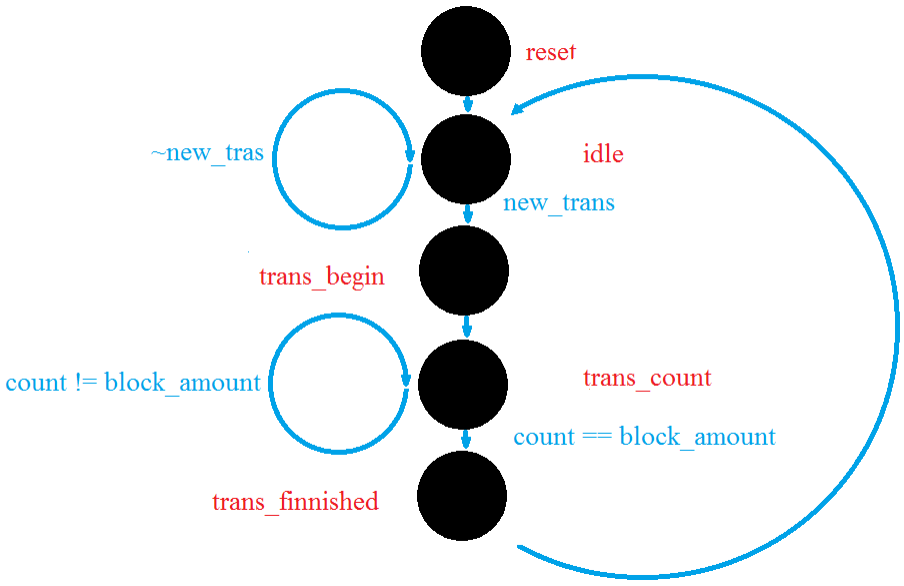

Figura 31: Transición de estados del módulo DAT

La estructura del módulo DAT se basa en el el control del un módulo de comunicación, que cuenta con instancias de módulos capaces de pasar de un canal de un número "n" de bits a

un número "m", lo cual incluye conversión serie a paralelo y paralelo serie, requeridos para la transferencia en modo de trama y multitrama. Esta estructura se muestra en la figura **??**.

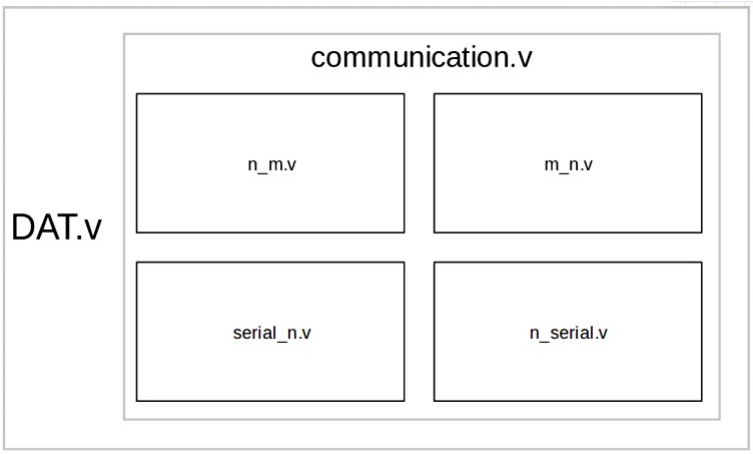

Figura 32: Estructura del módulo DAT

6.1. Simulaciones de transferencia de datos

Se realizaron simulaciones para cada uno de los posibles modos de transferencia, de la tarjeta SD al módulo de DMA y viceversa, utilizando el modo de operación de trama y multitrama. Se puede obtener del contenido del módulo de registros la dirección de escritura, además del modo de lectura y escritura de la tarjeta para 1 o 4 terminales.

En la siguiente imagen se muestra la transferencia de datos producto de una escritura de los datos de la tarjeta SD. Se observa la entrada de datos al módulo de DAT, tanto en modo de trama como de multitrama.

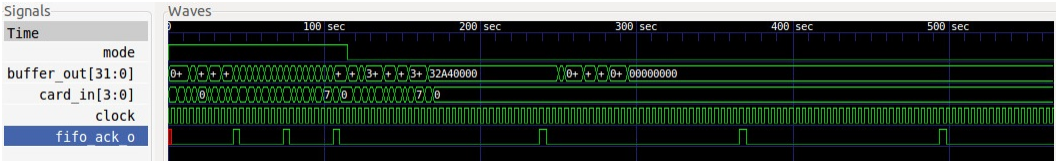

Figura 33: Lectura de datos de la tarjeta, señales obtenidas en el buffer para modo de operación de trama y multitrama

La siguiente sería una escritura que viene mediante el módulo DMA. Se muestra la salida del buffer y la transmisión a la tarjeta en modo de trama y multitrama.

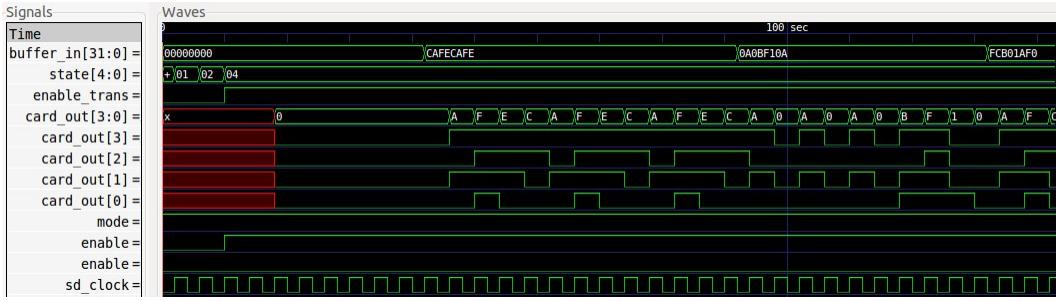

Figura 34: Transferencia hacia tarjeta SD, modo multitrama

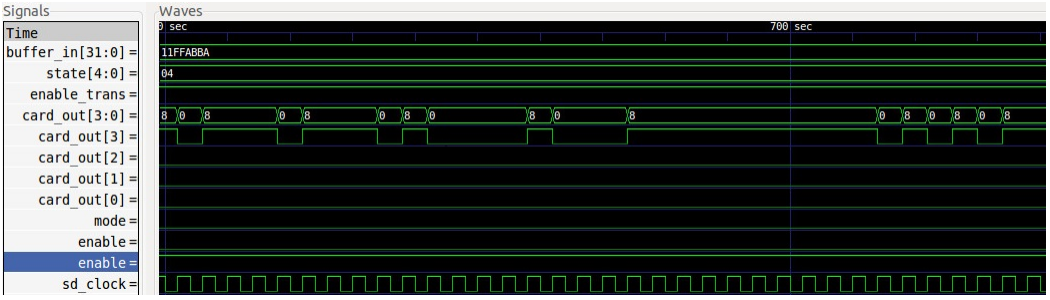

Figura 35: Transferencia hacia tarjeta SD, modo de trama

Se pone en evidencia que al utilizar más bits para leer y escribir datos de la tarjeta, aumenta significativamente la velocidad de transferencia de datos.

La siguiente sería la entrada de datos que recibe el DMA, al introducir datos leídos a DAT provenientes de la tarjeta.

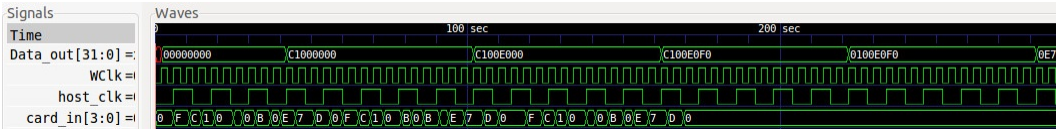

Figura 36: Lectura de tarjeta SD, datos recibidos por el módulo DMA

Referencias

[1] Rojas, José David & Espinoza Mauricio. (2016) *Modelado y análisis de sistemas lineales*. Universidad de Costa Rica.