Universidad de Costa Rica

Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0323 – Circuitos Digitales II
II ciclo 2021

Proyecto 2

QoS en la capa de transmisión PCIE

Estudiante:
Meybelle Castro Valverde - B91887
Adrián Montero Bonilla - B85092
Sofía Villalta Jinesta - B88565
Sinaí Zamora Vega - B88719
Grupo 07

Profesor: Jorge Soto

27 de noviembre, 2021

Índice

1. Descripción arquitectónica

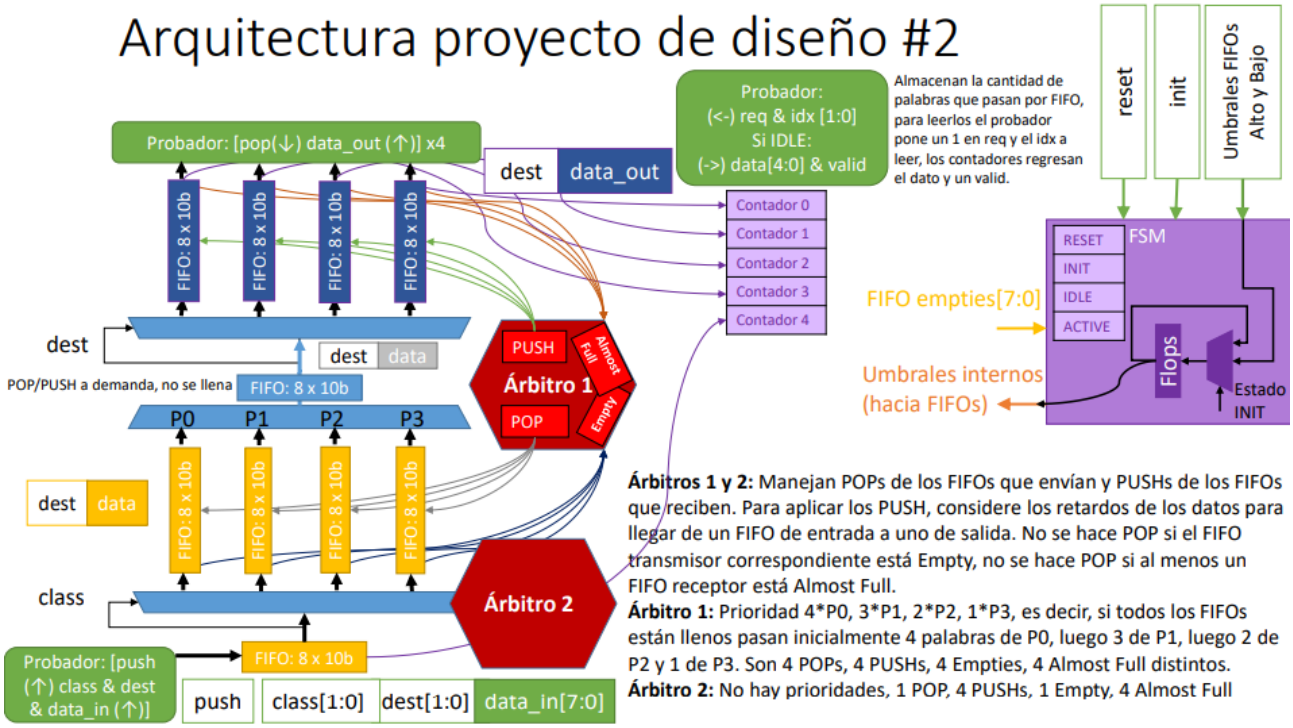

Figura 1: Esquemático del diseño.

2. Investigación

2.1. QoS (Calidad de Servicio)

El QoS o Quality of Service es un ”proceso” de control el cual regula los mecanismos de gestión de tráfico en una red para que esta cumpla con las necesidades de servicio de aplicaciones específicas y usuarios sujetos a las políticas de la red. Estas redes QoS necesitan poseer una lógica para manipular y/o controlar la asignación de recursos entre las aplicaciones y los usuarios. En la actualidad, se poseen algunos parámetros que pueden ayudar a cuantizar qué tan bueno es el QoS de una cierta red [1]:

- Packet Delay
- Packet Loss
- Bandwidth
- Jitter

2.2. Arbitraje en sistemas digitales

En una unidad de hardware pueden llegar varios buses de datos y lo más óptimo en estas situaciones es usar sistemas de arbitraje. Esto ayuda en la transmisión de los datos debido a que

evitan que entren muchos al mismo tiempo y se pierda información, los protocolos de arbitraje mantienen la organización del bus siguiendo un orden de prioridad y garantizando que el acceso del bus se realiza solo por un master. Existen principalmente dos tipos de arbitraje de datos.

- Centralizados: utilizan una unidad que se llama árbitro de bus que se encarga de transmitir de forma centralizada los datos desde una unidad a otra sección.
- Distribuidos: Las gestiones se realizan para cada bus de datos de forma distribuida desde las unidades de acceso, no hay árbitro.

Un ejemplo de protocolos de arbitraje es el Encadenamiento, este se basa en utilizar señales para manejar todos los buses, estas señales serían: concesión, petición, ocupación y reconocimiento. La primera va al árbitro mientras las demás van dirigidas hacia el master.

2.3. Priority Flow Control

El Priority flow control (PFC), que también es llamado Class-based flow Control (CBFC) consiste en un mecanismo que previene la perdida de datos debido a la congestión. [2]

Este mecanismo a diferencia del link-level flow control (LFC), funciona en una base per class-of-service (CoS), sin embargo sus funcionamientos son muy similares. En el caso del LFC, cuando un umbral de un buffer es excedido debido a congestión, el LFC envía una pausa de datos por un periodo especifico de tiempo, así, cuando la congestión se mitiga, es decir, cuando el trafico esta por debajo del valor umbral, se envía una señal de continuación para seguir la transmisión de datos.[2]

En el caso del PFC, durante la congestión, el Priority flow control envía una pausa que va a indicar cual valor de CoS necesita ser pausado. La señal de pausa del PFC contiene 2 octetos del tiempo del valor para cada CoS que indica el periodo de tiempo durante el cual el trafico necesita ser pausado.[2]

2.4. ¿Cómo se relacionan los créditos con Flow Control?

El flow control lo que hace es que se ocupa de prevenir que los recibidores lentos o los recibidores que están haciendo otras tareas se llenen y causen un “overflow“ a causa de que los transistores sacan datos de manera muy rápida. A pesar de que existen los mecanismos de flow control estáticos y dinámicos, ambos se basan en el uso de Créditos para así mantener marca de la cantidad de casillas disponibles en los determinados recibidores.

Este tipo de mecanismo asegura que los emisores tengan determinado espacio del lado de los receptores. En cuanto a la cantidad de espacio que poseen los buffers, está directamente determinado por el número de créditos que recibe del emisor al inico del sistema.

A parte se tiene que un mismo emisor puede enviar datos a distintos receptores y estos datos no se combinan, esto debido a su lógica de distribución. Y los créditos se pueden cambiar durante la ejecución, por lo cual se implementó la máquina de estados con un estado para cambiar estos valores de umbral.

3. Instrucciones de simulación

Primero, se debe descargar el .zip adjunto a este reporte y descomprimirlo en algún lugar. Una vez hecho esto, se abre dicha carpeta y dentro de esta se encuentran otras carpetas en las cuales se tienen guardados los módulos del proyecto. Para simular algún módulo, se abre la terminal con la dirección de la carpeta deseada y se escribe ”make”; automáticamente se desplegará la ventanilla de GTKWave para verificar el módulo.

Nota: Para poder visualizar las pruebas 1-7 del módulo QoS final, simplemente se abre la carpeta ”QoS-PCIE” y dentro de esta se encuentra otra carpeta llamada ”Pruebas-QoS”; la carpeta contiene todos los .vcd y .gtkw necesarios. Si se quiere hacer un ”make”para realizar una cierta prueba y verificar que todo compila, se debe eliminar el probador que está en la carpeta ”QoS-PCIEz cambiarlo por el probador deseado. Asimismo, se debe cambiar el makefile para seleccionar el .vcd correspondiente.

4. Resultados y análisis

4.1. Memoria

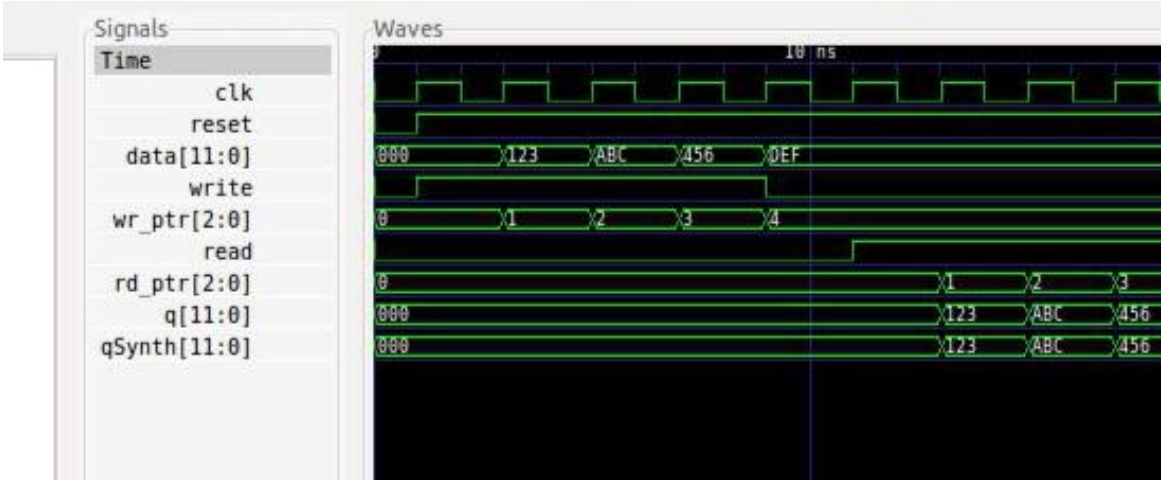

Figura 2: Resultados obtenidos para la memoria

Para la creación del módulo de la memoria, primeramente se buscó en internet un módulo de memoria en Verilog ya existente (en este caso, se basó en una memoria de Intel) y se modificó de forma que la lectura de datos fuese de modo asíncrona (no depende del reloj). La idea es que, cuando write está en 1, se escribe en la memoria y no se recibe ningún output, y, en caso contrario, si read está en 1, se lee de la memoria y se tiene un valor en la salida q. Además, nótese que se puede escribir y leer al mismo tiempo.

Como se puede observar en la figura 2 esta funciona correctamente porque en los primeros ciclos del reloj write está en alto y no hay ninguna salida, pero cuando read se pone en alto se empiezan a transmitir los valores de la entrada, también se puede observar que el archivo sintetizó bien porque las salidas son iguales.

4.2. FIFO

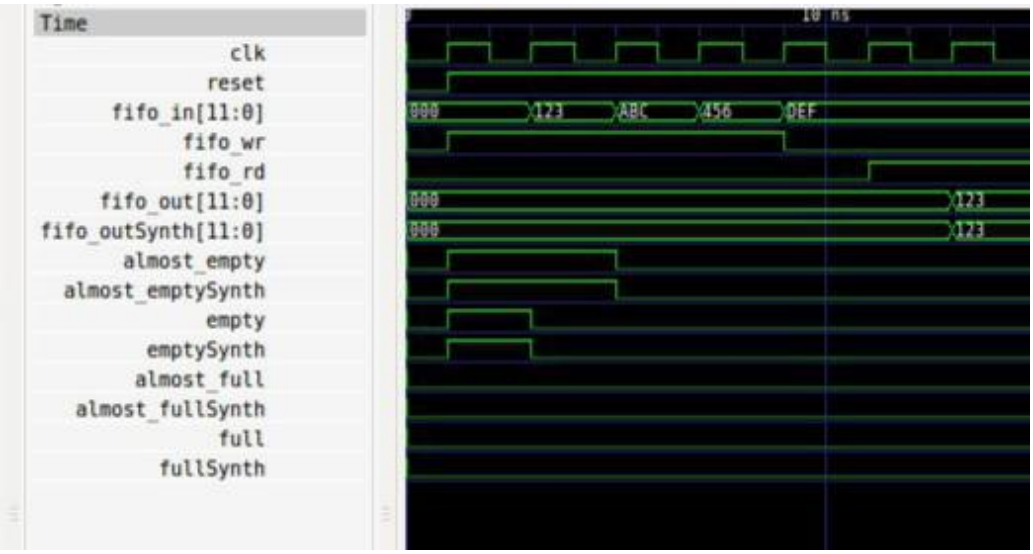

Figura 3: Resultado del módulo para el FIFO

Un FIFO es una ”array” de memorias que escribe o borra datos de estas en un cierto orden de prioridad. Para este proyecto se utilizaron 10 FIFOs y estos se encargan del manejo de la memoria en la arquitectura (tienen una lógica de escritura, lectura y de control). Como se puede observar en la figura 3, el sistema implementa la prioridad de los FIFOs de manera correcta; guarda en memoria los datos de entrada en caso que sen tenga una instrucción de write y, en caso contrario, lee el primer dato que se guardó. Asimismo, cuando se recibe un write/read, los punteros wrptr y rdptr incrementan en 1, avanzando al siguiente elemento de memoria. Además, nótese que la lógica de control implementada (control de señales almost-full, empty, etc) funciona correctamente.

4.3. Árbitros

4.3.1. Árbitro 1

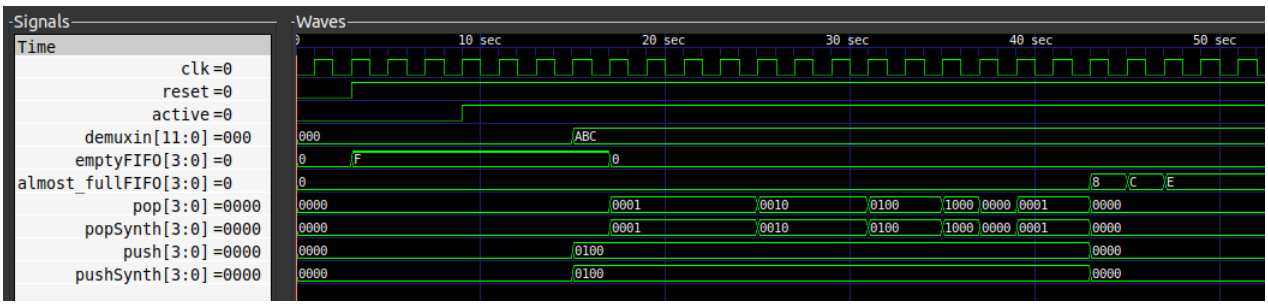

Figura 4: Resultados obtenidos para el árbitro 1

El árbitro 1 tiene todas sus salidas en 0 cuando se tiene reset o active en bajo. Una vez que dichas señales se ponen en 1, se revisan los almost-full y en caso que alguna esté en alto, tanto los PUSH como los POP son 0. Por otro lado, también se revisan las señales de empty

y en caso que se tenga que un cierto FIFO está empty, no se le hará POP a dicho FIFO. Si se cumplen todas las condiciones de operación, nótese de la figura 4 que se va siguiendo la prioridad (Round-Robin con pesos). En el caso de los PUSH, estos solamente dependen de la señal demuxin; si lo que se recibe del segundo demultiplexor es un 0, los PUSH son 0, caso contrario, se hace PUSH basándose en la entrada de selección del DEMUX (dest). Tanto las salidas conductuales como las estructurales dan igual.

4.3.2. Árbitro 2

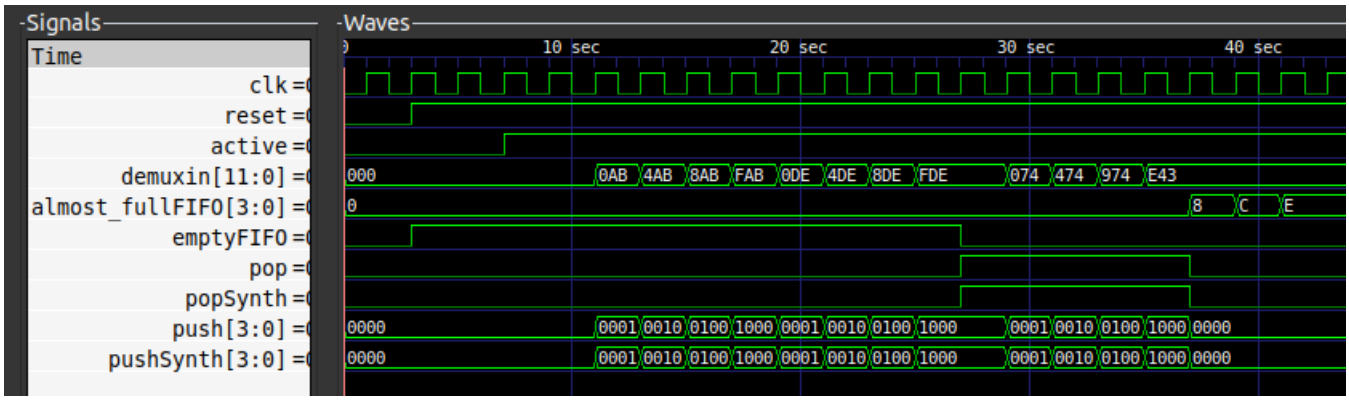

Figura 5: Resultados obtenidos para el árbitro 2

El árbitro 2 funciona de forma similar al primero, la única diferencia es que este no sigue una prioridad; va entregando las palabras que saca el FIFO amarillo solo a como van llegando. Nótese de la figura 5 que la funcionalidad es la correcta y que la síntesis fue exitosa.

4.4. Demux

Los demultiplexores están conformados por una entrada y cuatro salidas. Su función es la de direccionar las palabras que los FIFOs aislados sacan a los 4 FIFOs correspondientes (sean los amarillos o azules). El primer demultiplexor elige la salida de datos en base a los dos bits más significativos de la palabra de entrada (class), mientras que el segundo demultiplexor escoge la salida de datos según los tercer y cuartos bits más significativos de la palabra de entrada (dest).

Dicho módulo no lleva probador ni testbench ya que es muy básico y no se considera pertinente.

4.5. Contador

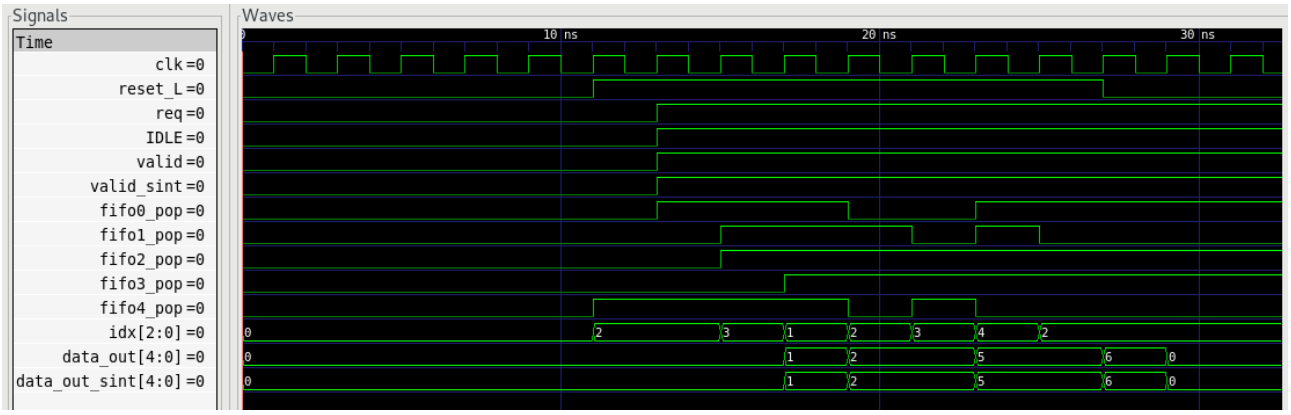

Figura 6: Resultado de módulo del contador.

Este módulo se usa para corroborar que la cantidad de palabras de la entrada y la salida sean iguales, con el objetivo de verificar que no se pierdan datos a través del proceso. En la figura 6 se observa que cuando reset está en cero, los FIFO y las salidas también están en cero.

4.6. Mux

El multiplexor recibe como entradas las salidas de los FIFOS y los POP (como un bus de datos, siendo 1 pop para cada FIFO) de estos. De esta manera, se diseña la lógica de selección del MUX a partir de los POP de los 4 FIFOs amarillos; la salida del multiplexor va a reflejar la salida del FIFO acorde al pop de cual FIFO reciba. En la figura 7 se puede apreciar el código que permite la lógica descrita.

```
module muxY(
    output reg [11:0] mux_out,
    input [11:0] fifo_out_0,
    input [11:0] fifo_out_1,
    input [11:0] fifo_out_2,
    input [11:0] fifo_out_3,
    input [3:0] pop);

    // dependiendo del pop recibido pasa fifo a la salida
    always @(*)begin
            mux_out = 0;;

            if (pop[0] == 1)
                mux_out = fifo_out_0[11:0];

            else if (pop[1] == 1)
                mux_out = fifo_out_1[11:0];

            else if (pop[2] == 1)
                mux_out = fifo_out_2[11:0];

            else if (pop[3] == 1)
                mux_out = fifo_out_3[11:0];


        end

endmodule
```

Figura 7: Código del modulo Mux.

4.7. Máquina de estados

La máquina de estados es la encargada de determinar el estado en que se encuentra el PCIE. Entre los estados que se tienen están el **RESET**, **INIT**, **IDLE** y el **ACTIVE**. Cuando se encuentra en RESET, esta lista para cambiar a INIT, por lo que su próximo estado sera INIT. En INIT se pueden modificar los umbrales y su próximo estado sera IDLE. En IDLE se encuentra inactivo, pues mientras todos los 9 FIFOS están vacíos (empty = 111111111) no hay actividad, de lo contrario, cuando al menos un FIFO no se encuentra vacío, se pasa a un estado de ACTIVE donde se transmiten los datos. En la figura 8 se encuentra el diagrama de tiempos de la maquina de estados, donde podemos observar que cuando se tiene el bus de empty lleno, el próximo estado pasa a ser IDLE, y esta señal se activa, y en cuanto haya por lo menos un FIFO que no esté vacío (algún cero en el bus empty), el estado se encuentra en ACTIVE ya que hay transmisión de datos.

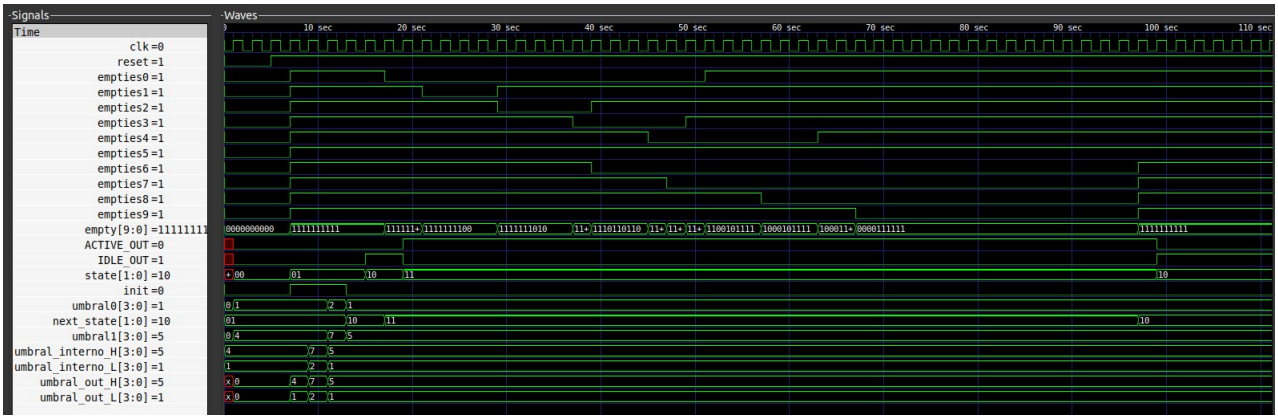

Figura 8: Resultados del módulo FSM

4.8. Plan de pruebas

4.8.1. Pruebas 1-3

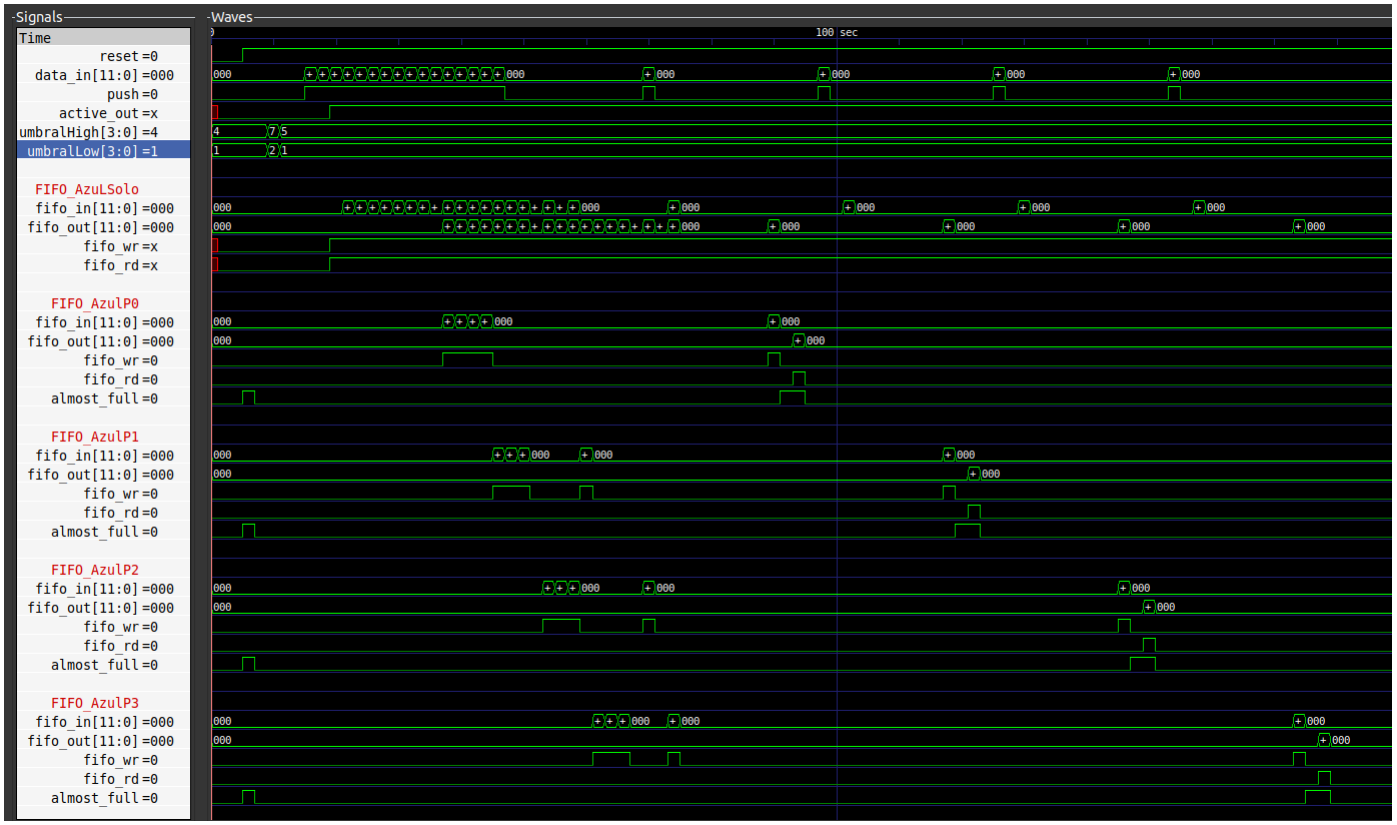

Figura 9: Resultados de las pruebas 1-3

Nótese de la figura 9 que se satisfacen los 3 primeros puntos del plan de pruebas; se saca al bloque de la señal de reset dejando la señal de init y luego se modifican los umbrales para todos los FIFOs 2 veces y se pone en bajo la señal de init. Una vez hecho esto, se provoca un almost-full en todos los FIFOs azules y para lograr esto se necesitó hacer POPs desde el probador hacia dichos FIFOs azules. Tanto las salidas conductuales como las estructurales son exactamente iguales (por temas de espacio no se incluyeron dentro de la figura 9, pero se pueden confirmar utilizando el .vcd de la prueba 1-3).

4.8.2. Prueba 4

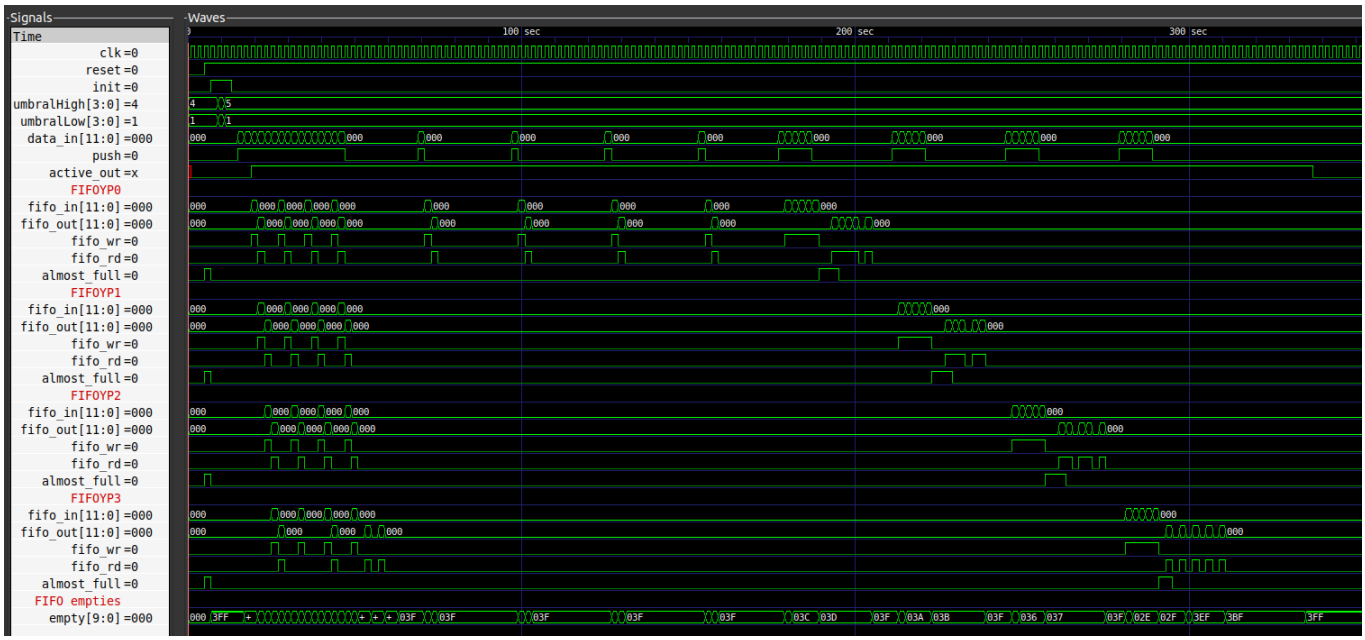

Figura 10: Resultados de las prueba 4

Siguiendo con los resultados de las pruebas anteriores, ahora se provoca un almost-full en todos los FIFOs amarillos, como se puede ver en la figura 10. De misma manera que con las pruebas anteriores, se van sacando poco a poco palabras de los FIFOs de salida para poder seguir llenando hasta almost-full a todos los FIFOs de entrada. Una vez que se cumple con el punto anterior, se realizan POPs desde el probador para vaciar a todos los 10 FIFOs de la arquitectura.

4.8.3. Prueba 5

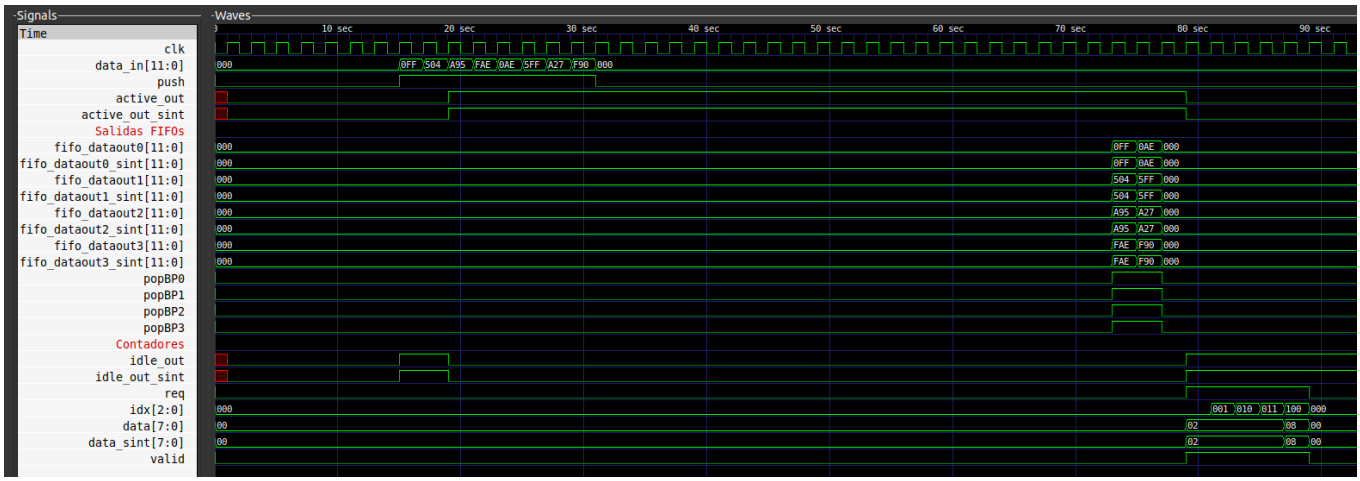

Figura 11: Resultados de las prueba 5

Para términos de simplificación, se realizó una prueba a parte para cumplir con el punto 5 del plan de pruebas. Nótese que en este caso metemos al sistema solamente 8 palabras; al final

se tendrán 2 palabras en cada FIFO de salida, obteniendo en total las 8 palabras que ingresaron. Después de aplicar los POPs a los FIFOs azules, nótese que todos los FIFOs quedan en empty, provocando que se alce la señal de IDLE de la máquina de estado. En este estado es permitido leer la cantidad de palabras que se sacaron de cada FIFO azul mediante las señales de req y data. La figura 11 muestra los resultados esperados.

4.8.4. Pruebas 6 y 7

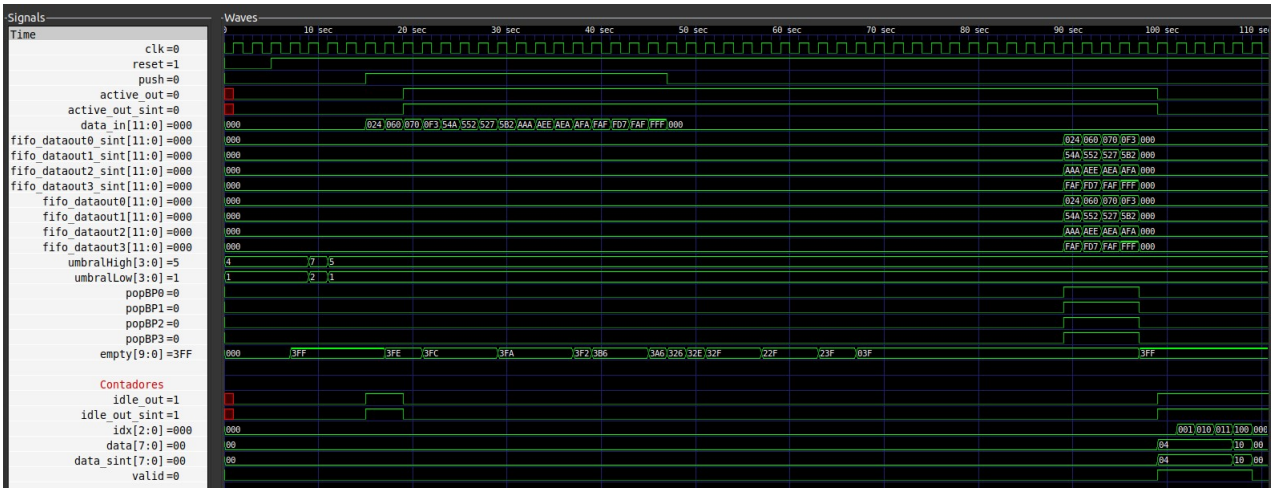

Figura 12: Resultados de las pruebas 6 y 7

En la figura 12 se observa que la entrada data_in recibe 16 palabras conformadas por 12 bits, de las cuales éstas se pasan a cada FIFO en sets de 4, donde la señal Fifo_dataout0 recibe las cuatro palabras que contengan 00 en los bits más significativos, mientras que Fifo_dataout1 recibe las palabras que contengan 01 en los bits más significativos, en cambio Fifo_dataout2 recibe las palabras que contengan 10 en los bits más significativos, y por último Fifo_dataout3 recibe las palabras que contengan 11 en los bits más significativos.

Como se puede observar de la figura 12, los contadores de salida están funcionando bien debido a que estos marcan que se sacaron 4 palabras de cada FIFO de salida, para un total de 16 palabras.

5. Conclusiones

- Se logró entender el mecanismo de un sistema de arbitraje y su importancia a la hora de transmitir información, como con los POPS se fueron enviando los datos en el orden de prioridad establecido para que fueran recibidos por los otros cuatro FIFOS.
- Un error que se presentó fue el de un latch por no tener cuidado dentro de la lógica de un always(*) por lo que siempre es recomendable revisar que se inicialicen todos los valores de manera correcta.
- Es importante revisar cuidadosamente las conexiones que existen entre cada módulo y en su capa final ya que se tuvo problemas con distintas señales debido a que se estaba conectando de manera errónea los módulos en el QoS, no obstante, al revisar este modulo, se pudo corregir donde existían conexiones incorrectas en los demuxes y en el contador.

6. Bitácora

- Avance 1: Para el primer avance se subdividieron las tareas de la primera entrega en parejas, donde Sofía y Adrián trabajaron en conjunto con la Memoria y Sinai y Meybelle trabajaron en los FIFOS. Cada pareja se encargo de hacer la síntesis, los probadores y los bancos de pruebas respectivos para visualizar si se tenia un correcto funcionamiento. Ambos grupos acabaron entre el 7 y 8 de noviembre.
- Avance 2: Para la entrega de este avance se mantuvieron las parejas del avance uno, donde Adrián y Sofía se encargaron de trabajar con los árbitros y Sinai y Meybelle con los contadores. Luego Sofía se unió a Sinai y Meybelle en contadores y Adrián se encargó de los árbitros. Cada uno se encargo de los probadores y bancos de pruebas respectivos. Todos los módulos de esta entrega se finalizaron el 15/16 de noviembre.
- Avance 3: Para esta última entrega se invirtieron las parejas, esta vez Sofía y Sinai trabajaron juntas en la lógica miscelánea del mux y la máquina de estados y Adrián y Meybelle se encargaron de los demuxes. Luego todos nos reunimos y nos encargamos de unir todos los módulos para el QoS. Esta entrega se finalizó el 26 de noviembre.

Referencias

[1] M. Kaur, "An overview of quality of service computer network," *Indian Journal of Computer Science and Engineering*, vol. 2, 06 2011.

[2] Cisco, "Configuring Priority Flow Control," accesado 26 de Noviembre, 2021. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/sw/5_x/$nx-os$/qos/$configuration$/guide/$nx-os_q os_b ook$/$qos_p fc.pd$