Cómo implementar memorias en Vivado. Parte 3: FIFOs

Creador: David Rubio G.

Entrada: https://soceame.wordpress.com/2024/11/17/como-implementar-memorias-en-vivado-parte-3-fifos/

Blog: https://soceame.wordpress.com/

GitHub: https://github.com/DRubioG

Fecha última modificación: 23/02/2025

Esta es la última entrada sobre cómo se generan memorias en Vivado, esta entrada habla sobre cómo se configura un FIFO en Vivado.

Un FIFO a diferencia de una memoria RAM no tiene un bus de direcciones, por lo que los datos que entran son almacenados en orden de llegada, y salen del FIFO en el mismo orden en el que han entrado, el llamado *First Input-First Output*.

Estos FIFOs pueden estar creados a partir de memorias RAM de dos puertos, donde solo hay un puerto de escritura y uno de lectura. Xilinx a esta RAM la llama, *Simple Dual Port RAM*.

Los FIFOs también pueden ser síncronos o asíncronos, dependiendo de para que lo quieras utilizar. Los FIFOs asíncronos tienen una utilidad especial en la sincronización de datos en sistemas con diferentes dominios de reloj.

Puertos

Uno de los apartados principales de los FIFOs, es entender los puertos que tienen. Los FIFOs principalmente poseen puertos de «control del estado del FIFO».

Esto es que tiene que haber puertos que garanticen que el FIFO no desborda o que el FIFO no entrega datos nulos o vacíos.

Por ello se hace prioritario tener al menos dos puertos:

- El puerto de «*full*» o lleno, que le dice al sistema que **escribe** que el FIFO está lleno y que no incorpore más datos.
- El puerto de «*empty*» o vacío, que le dice al sistema que **lee** que el FIFO está vacío y que no solicite más datos.

Estos dos puertos funcionan como semáforos, dejando el puerto a nivel alto en caso de que esté lleno.

Bloques en Vivado

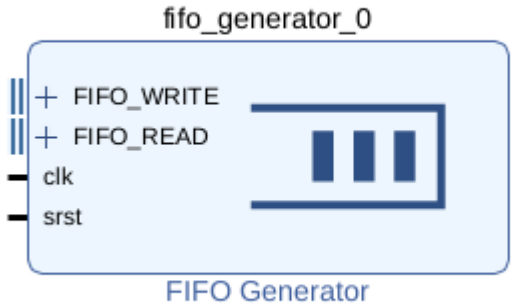
Para implementar FIFOs, Vivado tiene diferentes opciones: *AXI-Stream FIFO, AXI-Stream Data FIFO, AXI Data FIFO y el FIFO Generator.*

Todos los bloques IP anteriores funcionan sobre el protocolo AXI Stream, se utilizan para la transferencia de datos en AXIs, menos el FIFO Generator, que permite trabajar con AXI-Stream y sin AXI-Stream.

En nuestro caso, vamos a explicar solo el *FIFO Generator*, porque el resto de bloques IP es igual, solo que a través de una interfaz AXI.

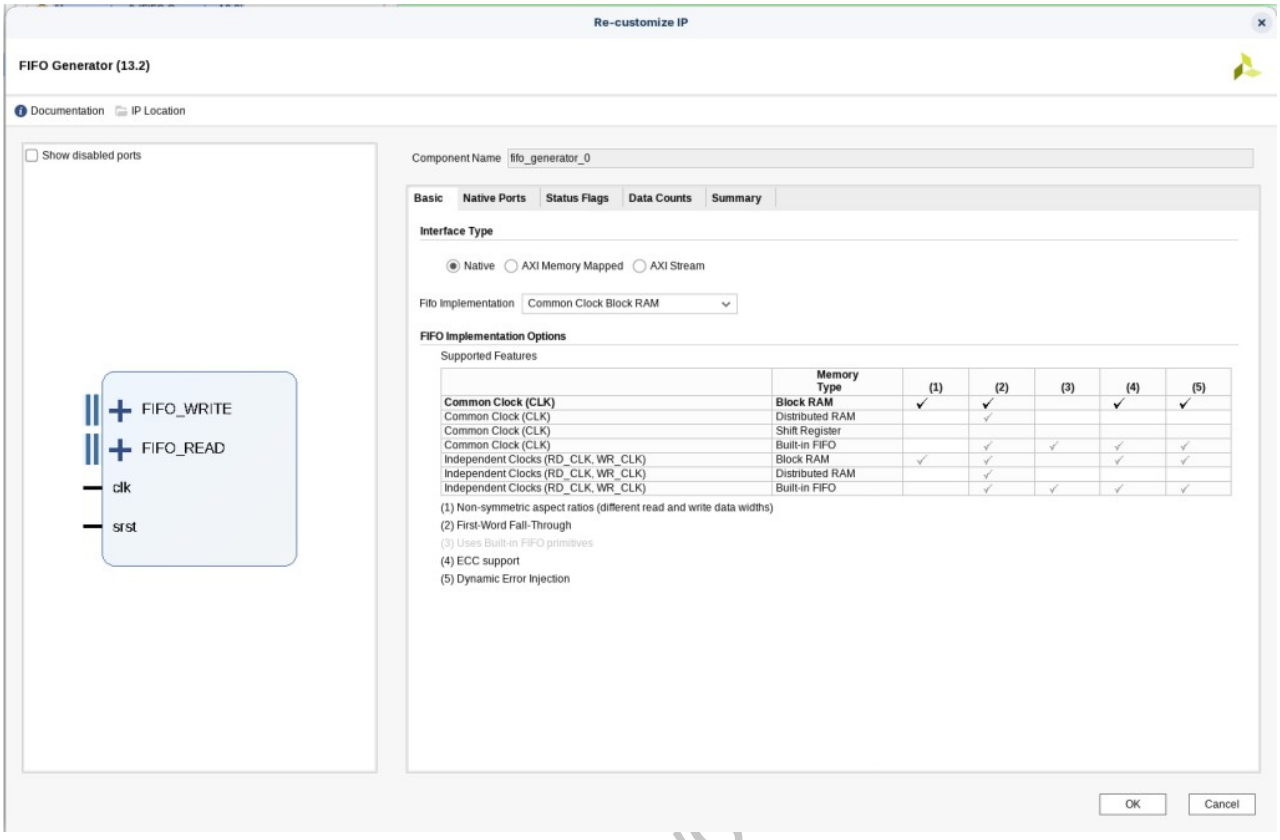
Implementación

Ahora vamos a explicar como implementar el **FIFO Generator**.



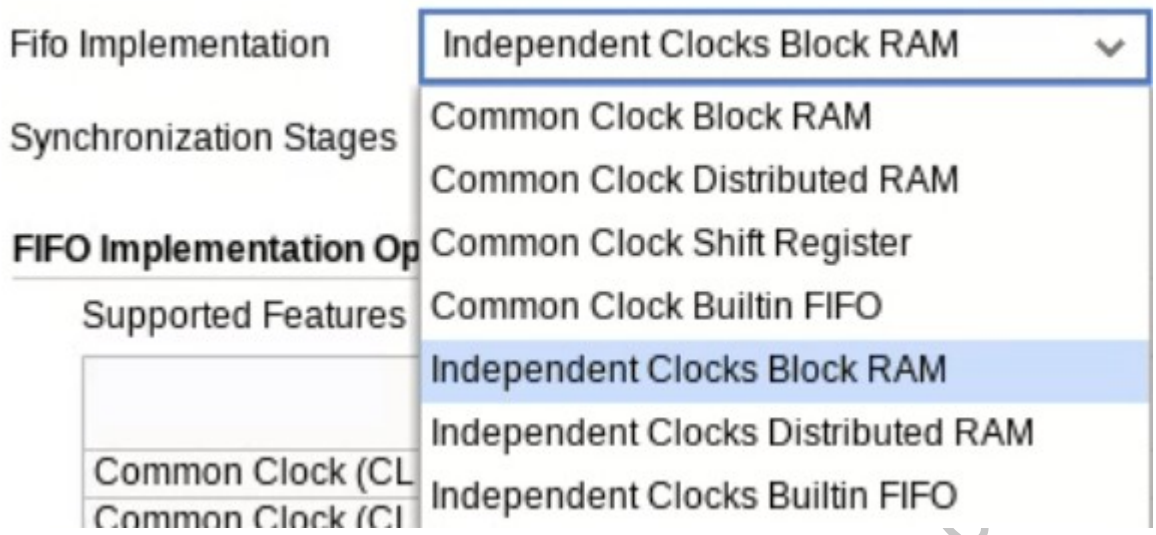
El bloque IP tiene diferentes configuraciones que modifican las pestañas, pero la mayor parte de las pestañas tienen las siguientes opciones.

Al entrar en el bloque lo primero que vemos es que tiene multitud de configuraciones (*no hay que asustarse, porque el cuadro que aparece al entrar es un cuadro resumen*). Bien, lo primero que vemos es que se puede configurar los puertos de entrada como *Native*, entrada directa de datos, por *Memory Mapped*, para que se almacene en memoria interna en vez de una BRAM, o por *AXI Stream*, para flujos de datos por AXI.



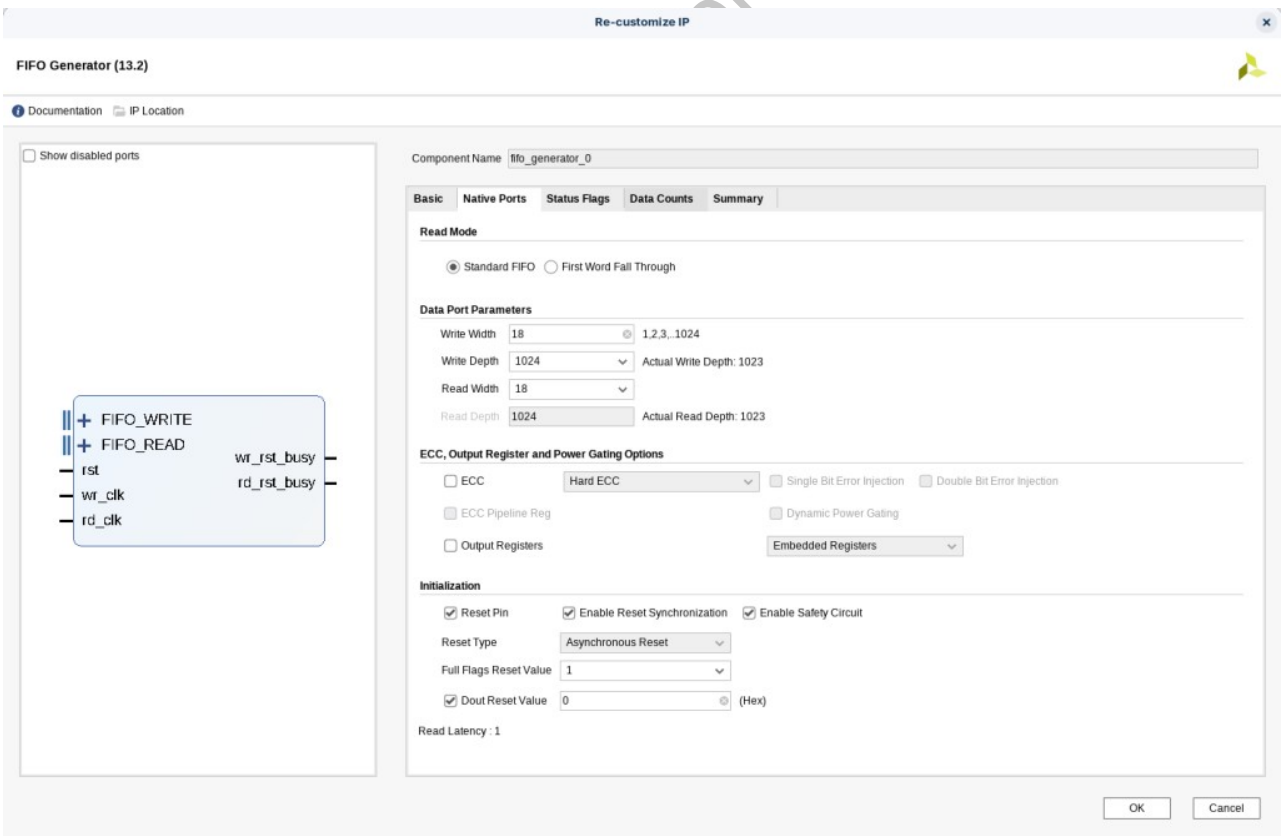
Lo siguiente que te permite es elegir cómo se construye la memoria.

- Los *Common Clock* son para hacer **FIFOs síncronos** y los *Independent Clocks* para hacer **FIFOs asíncronos**.
  Todas la memorias asíncronas, menos las Builtin, van a preguntar por cuántas etapas de sincronización quieres, el mínimo es 2. Esto último es debido a que tienes datos que vienen de dos dominios de reloj distintos y el FIFO los tiene que sincronizar con la memoria *(importante de tener en cuenta antes de escribir datos)*.
- El que pone *Block RAM*, es porque utiliza una BRAM para construirlo
- Los que pone *Distributed RAM*, es porque utiliza lógica de la FPGA para construirlo.
- El que pone *Shift Register,* es porque utiliza lógica de la FPGA en modo registro de desplazamiento.
- Y los que pone *Builtin FIFO* es porque se construyen usando primitivas *(de la librería XPM, explico al final)*



La siguiente pestaña no pregunta por el tamaño del FIFO y el tamaño de los datos que se van a guardar en él. También se puede configurar el *First Word Fall Through*, esta opción lo único que hace es escribir el próximo dato en ser leído del FIFO en la salida para acelerar la lectura.

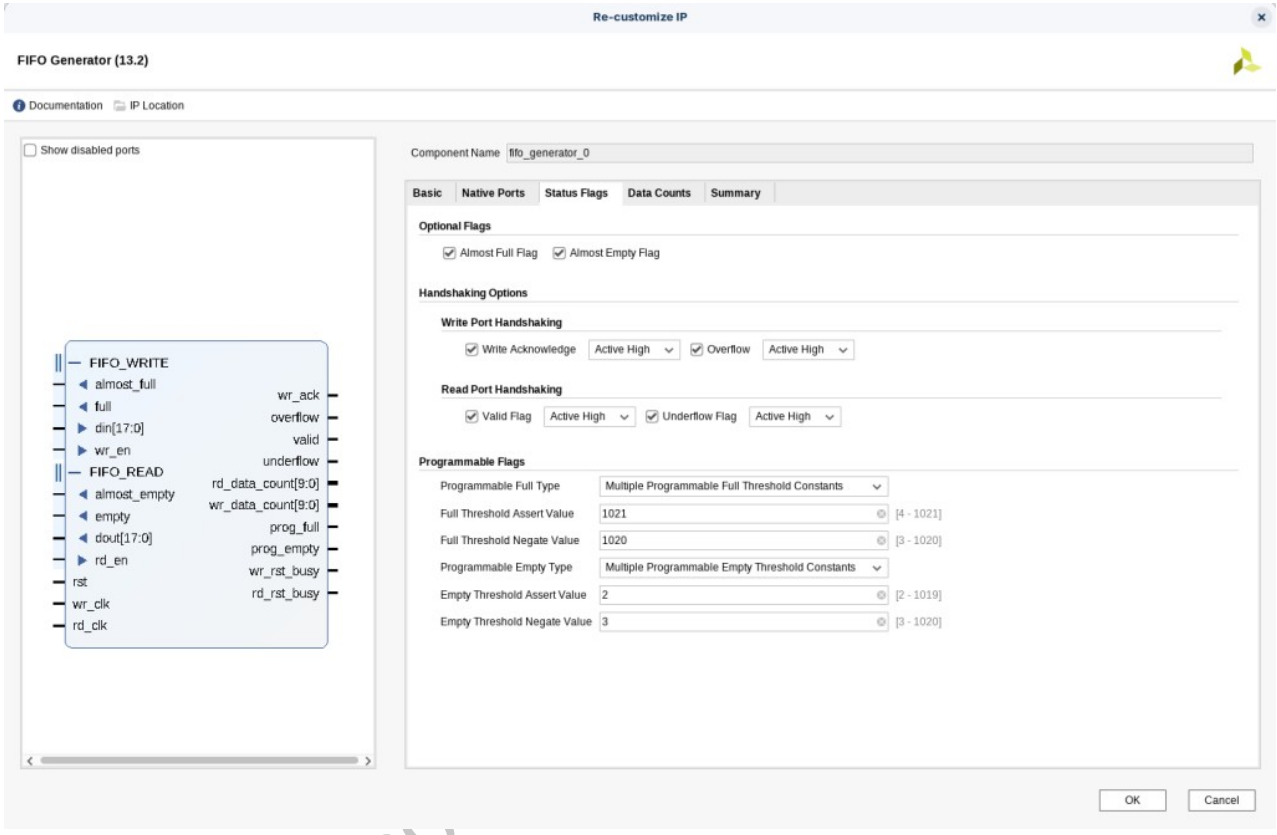
NOTA: los puertos «*_rst_busy*» se deshabilitan con la casilla *Enable Reset Synchronization*.



En la siguiente pestaña se pueden configurar puertos para conocer el estado interno.
Los puertos *Almost Full Flag* y *Almost Empty Flag*, son puertos que avisan un ciclo de reloj antes que van a estar llenos o vacíos.

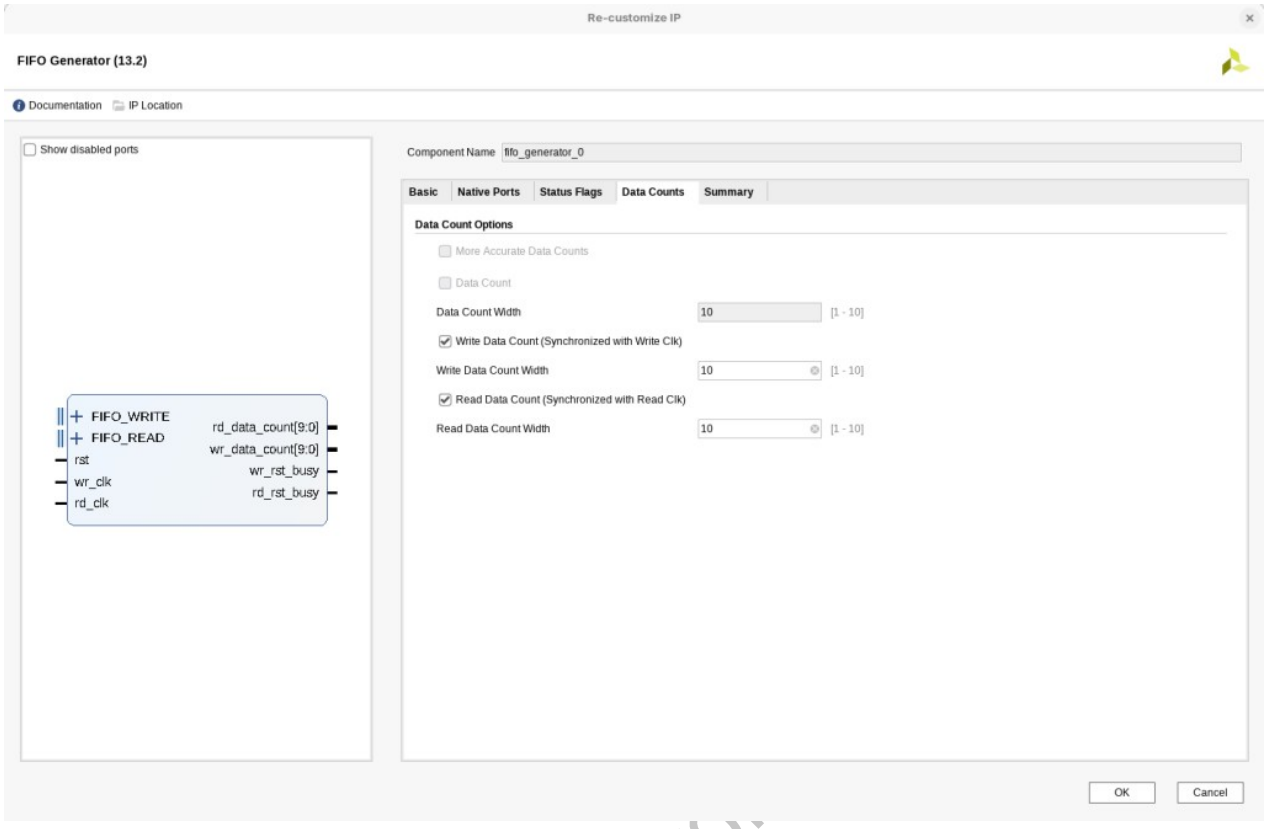
Los puertos *Write Acknowledge, Overflow, Valid Flag y Underflow Flag*, indican si la escritura o la lectura en el FIFO son correctas, o si al querer escribir o leer resulta que el FIFO está lleno o vacío *(para hacer está última función el FIFO ya cuenta con dos puertos, que son el full y el empty)*.

La última opción lo único que hace es añadir un puerto que se activa cuando: la memoria del FIFO llega a un determinado nivel (*threshold*) ya sea para escritura o para lectura. Ejemplo, cuando la memoria tenga 50 datos escritos o cuando a la memoria le queden 50 datos por ser leídos.

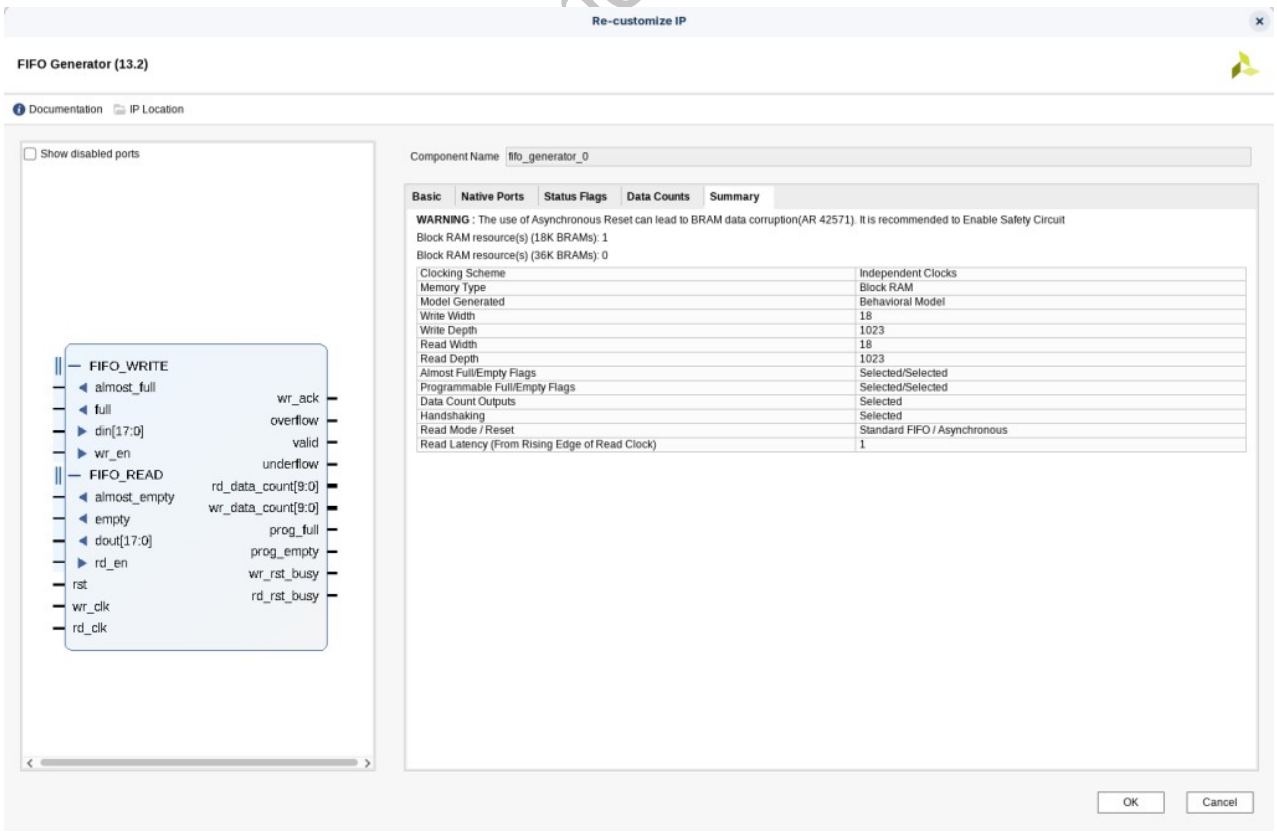


NOTA: si queremos configurar un FIFO tipo *Builtin*, nos va a preguntar por la frecuencia con la que se va a leer y a escribir para ajustar la implementación.

La siguiente pestaña te dice cuantos datos tiene el FIFO dentro, solo le tiene que decir el tamaño del contador.



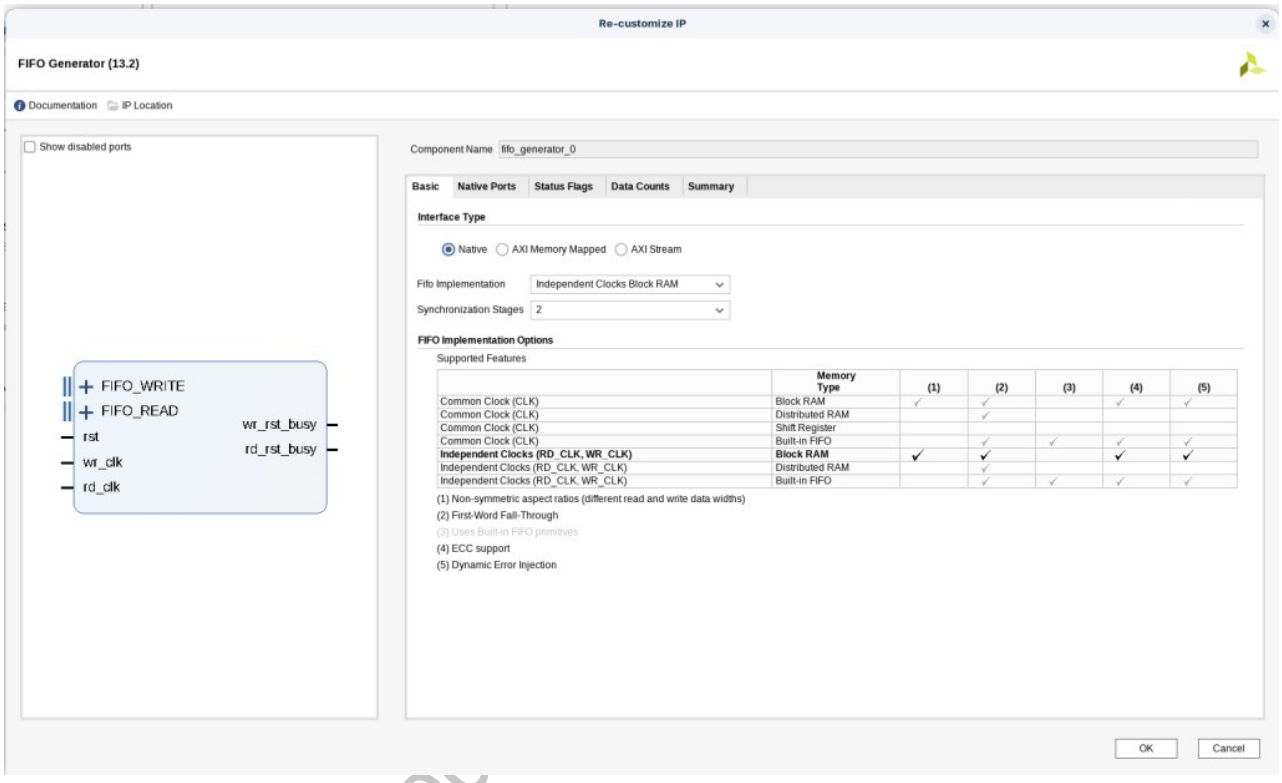
Y por último, la última pestaña te hace un resumen. De este resumen el dato más importante es el *Read Latency* que te dice que latencia tiene el FIFO en devolver un dato.



Ejemplo

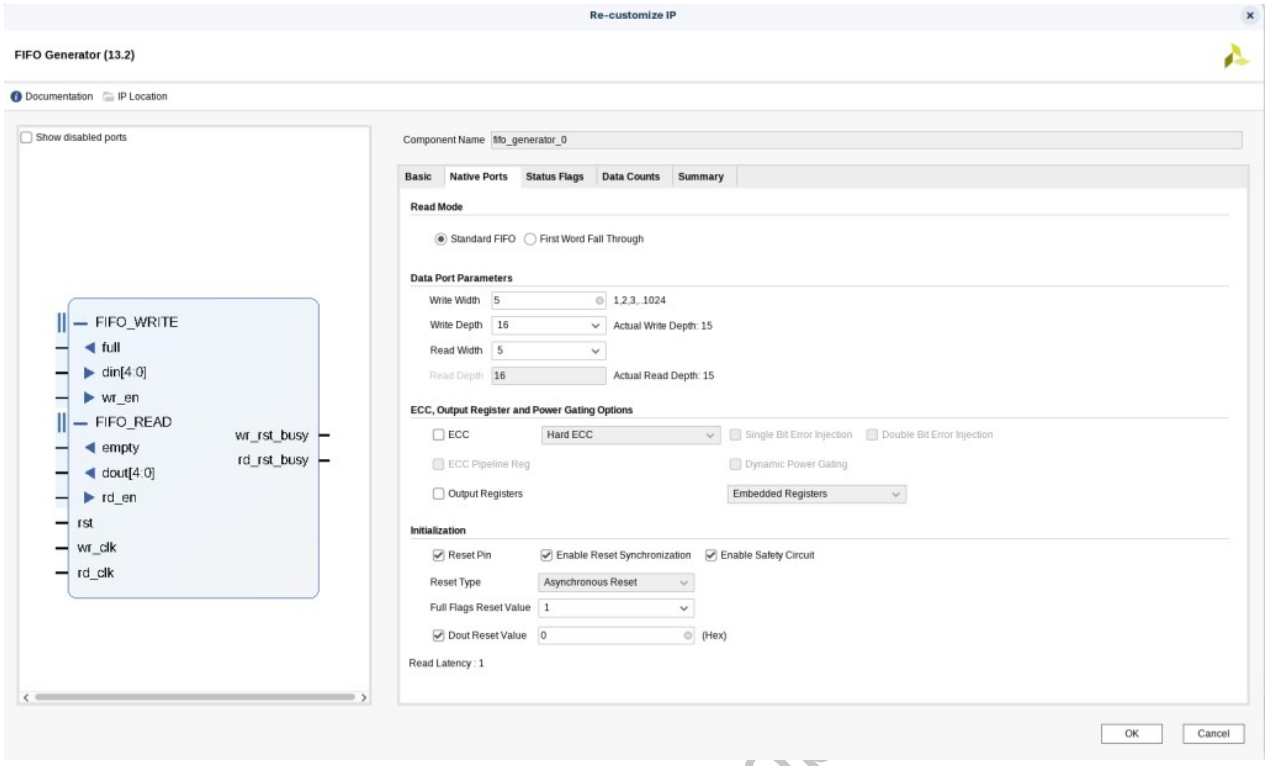
Lo próximo es hacer un ejemplo para ver más o menos el funcionamiento de un FIFO. Para ello vamos a crear un FIFO asíncronos de 5 bits de datos y de 16 espacios que vamos a rellenar hasta el final y después lo vamos a leer.

Lo primero es configurar la memoria como asíncrona, en mi caso quiero utilizar una BRAM, con 2 etapas de sincronización (2 biestables D, para que nos entendamos).

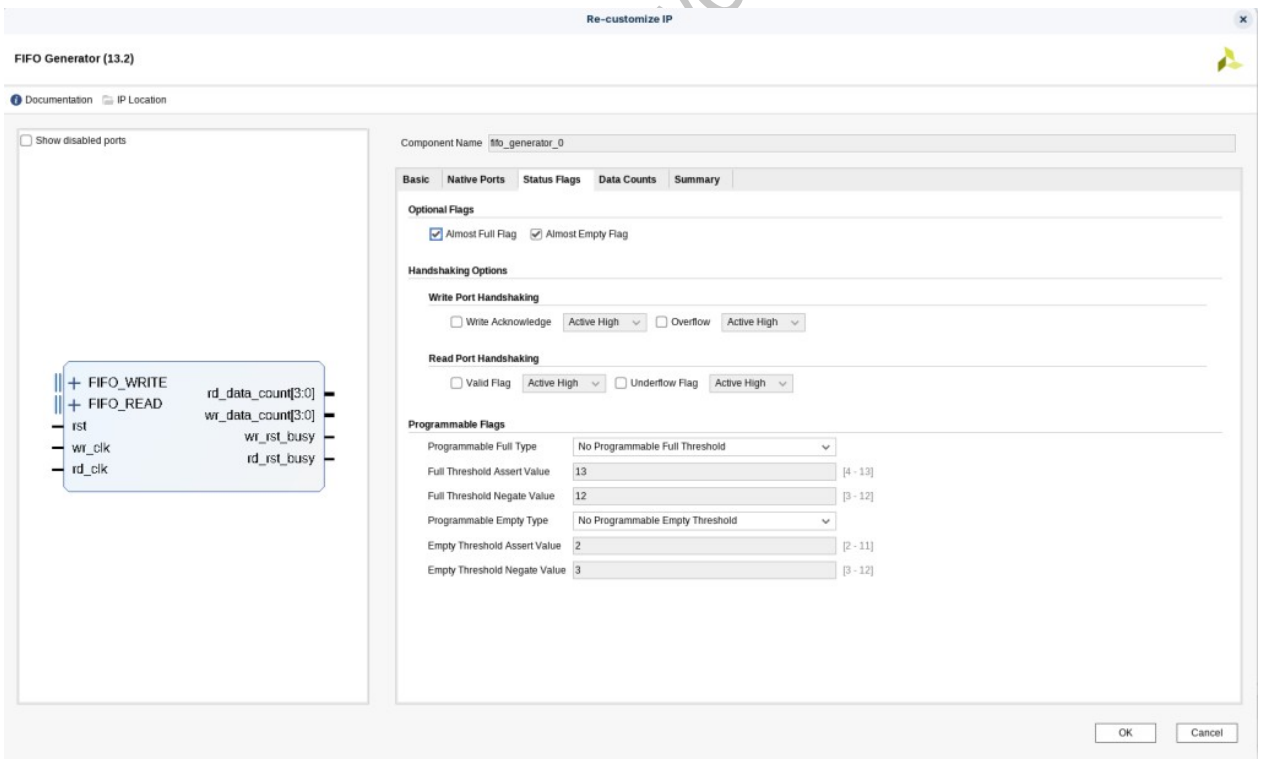


Lo siguiente es configurar la memoria a un tamaño de 16 y un tamaño de datos de 5 bits. **NOTA**: los reset es posible que sea mejor que no se sincronicen, es mejor utilizar una señal que ataque los dos puertos a la vez.

https://soceame.wordpress.com/2024/11/17/como-implementar-memorias-en-vivado-parte-3-fifos/

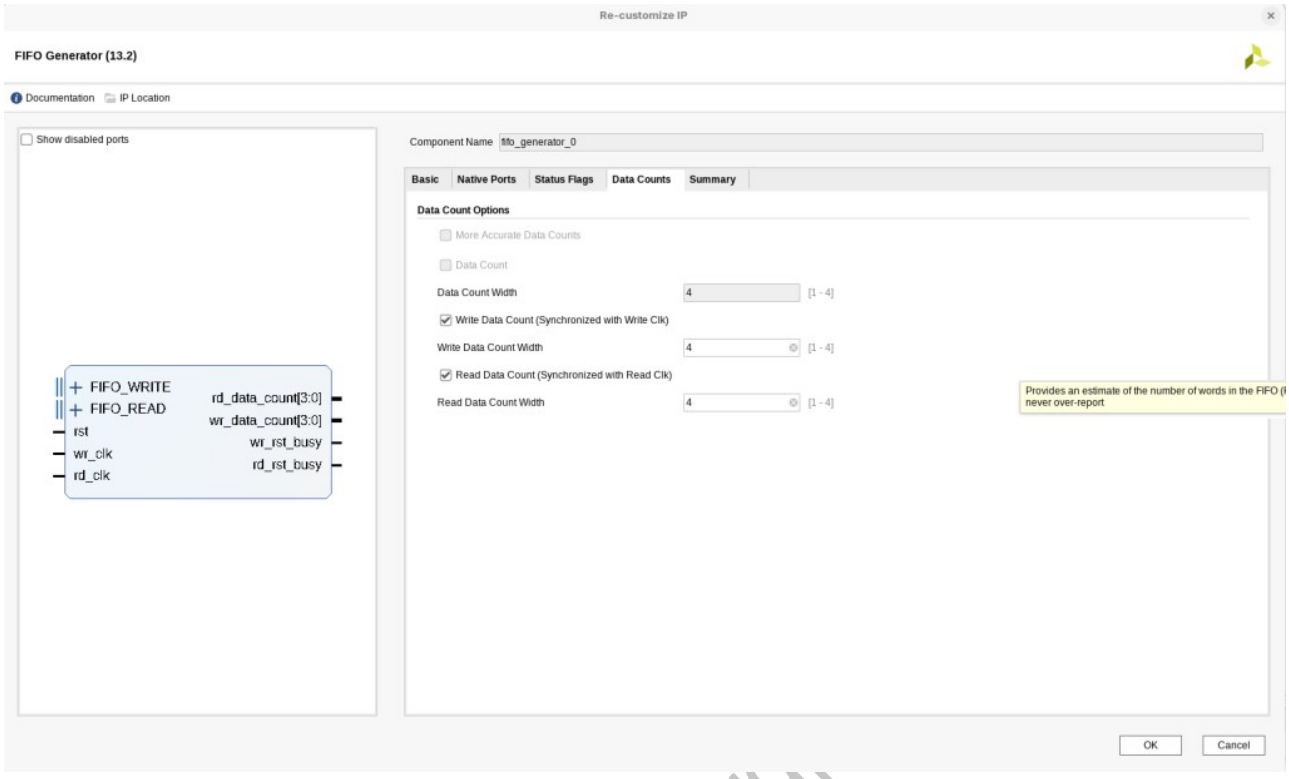


Vamos a configurar un *Almost Full* y un *Almost Empty*, solo para ver cómo se comporta.

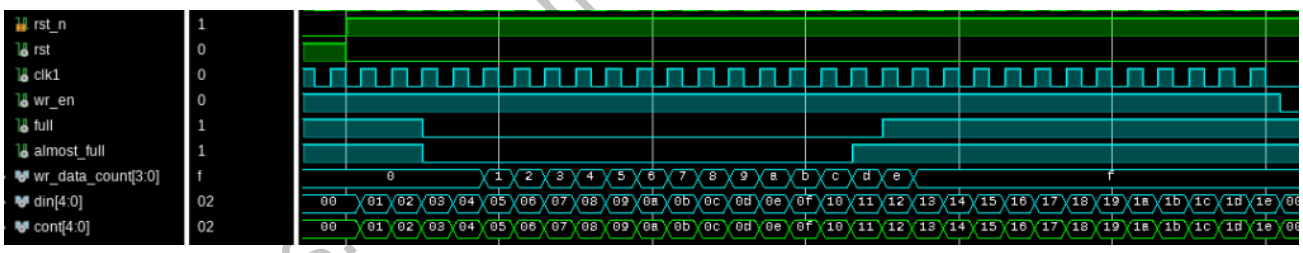


Y también el contador de datos en memoria para ver como funciona.

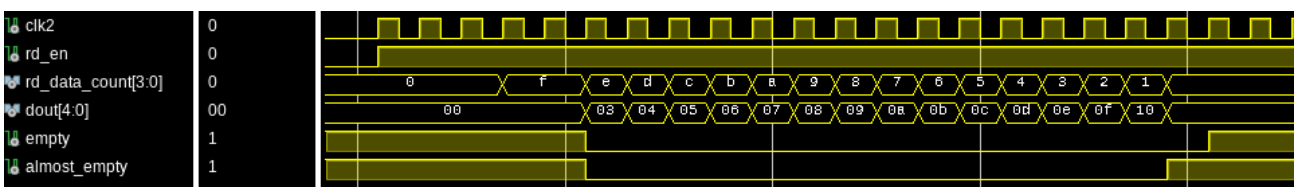
https://soceame.wordpress.com/2024/11/17/como-implementar-memorias-en-vivado-parte-3-fifos/



Con todo lo explicado anteriormente, ahora analizamos la escritura de datos. Se puede ver que el *rst* es a nivel alto, también se puede ver que el puerto *almost_full* genera un pulso antes de que lo genere el puerto *full*. También, que el contador de pulsos informa dos ciclos después de la escritura del tamaño del FIFO ocupado.

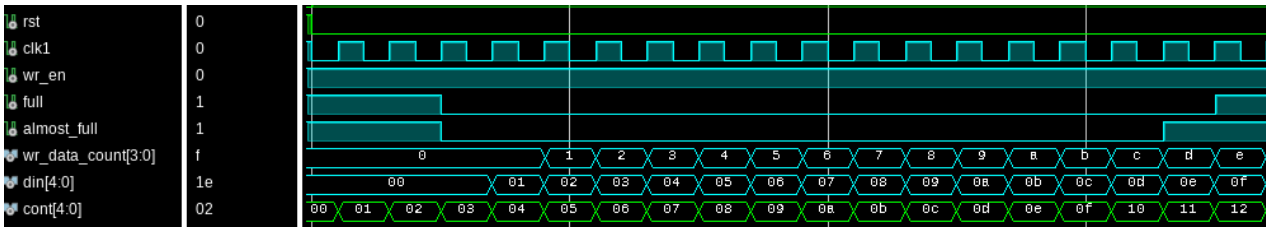


Si ahora analizamos al lectura de datos, se puede ver que el *almost_empty* genera un ciclo de reloj antes un pulso a nivel alto, con respecto a *empty.* Lo siguiente que se puede ver es que el contador de datos de lectura es decremental. Y como se puede ver es que los primeros datos que devuelve son cero y después empieza en 3, esto es debido a que en la escritura no se ha respetado el retardo que se tiene que tener entre que se habilita la escritura con *wr_en* y el dato que se quiere escribir. Esto último ocurre porque el parámetro *Synchronization Stages* para la sincronización entre datos de diferentes dominios de tiempo.

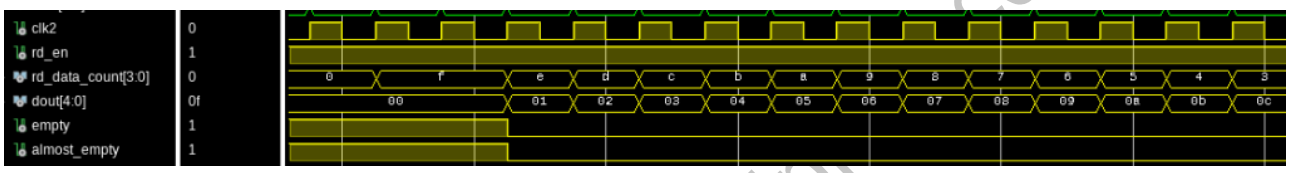


Si se decide retardar los datos de entrada manualmente para evitar empezar a escribir en el mismo instante en el que el *wr_en* pasa a nivel alto.

En la escritura se aprecia el retardo en poner el dato en el puerto *din*.



Y en la lectura ya se puede ver que se ha corregido el problema con la escritura en *dout*. Para ello solo es necesario retardar le escritura el valor que se haya puesto en *Synchronization Stages*. En mi caso son dos ciclos de reloj de retardo.



NOTA FINAL

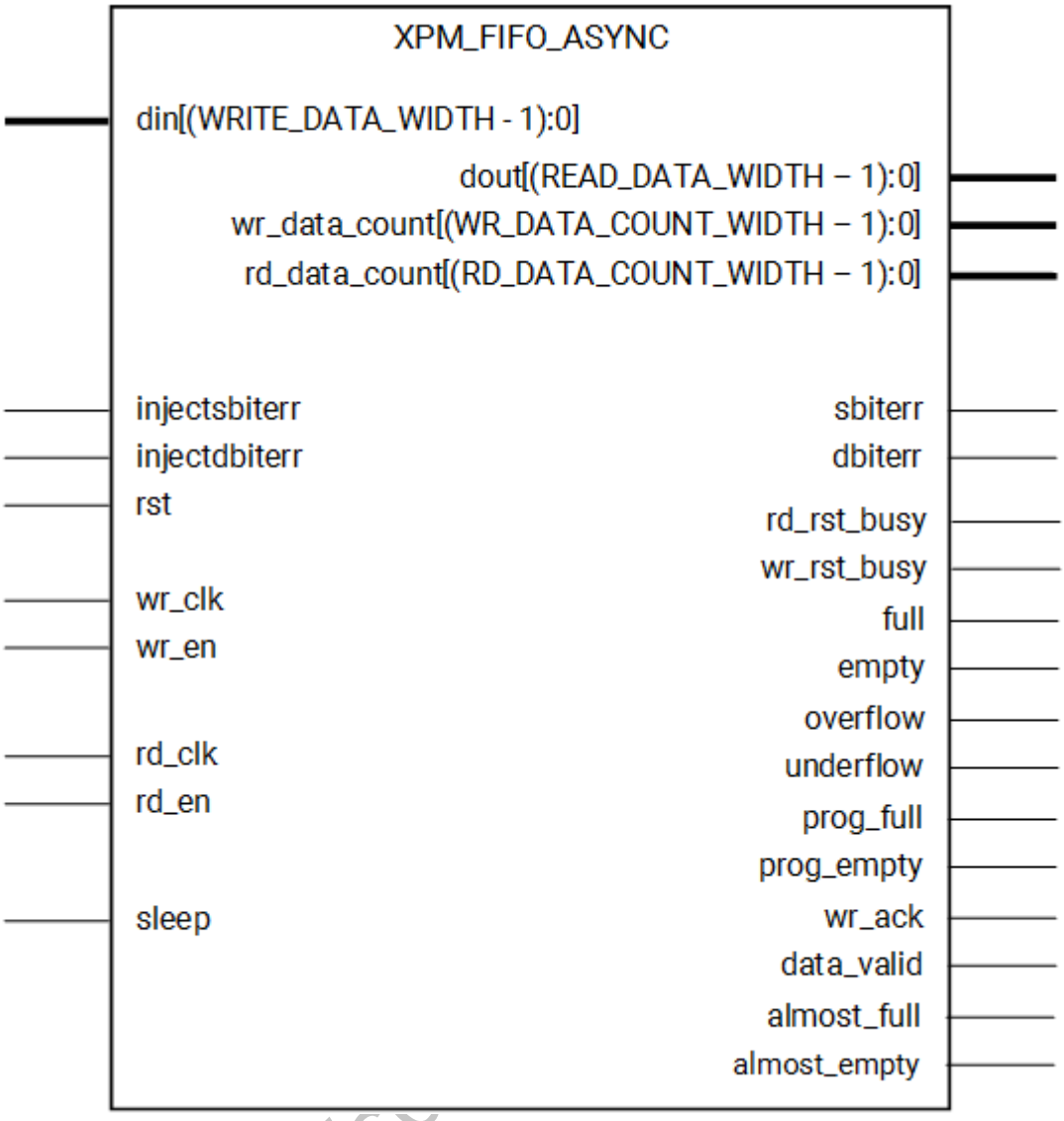
En FIFOs asíncronos hay que retardar la escritura de datos, para garantizar su escritura.

Librerías XPM

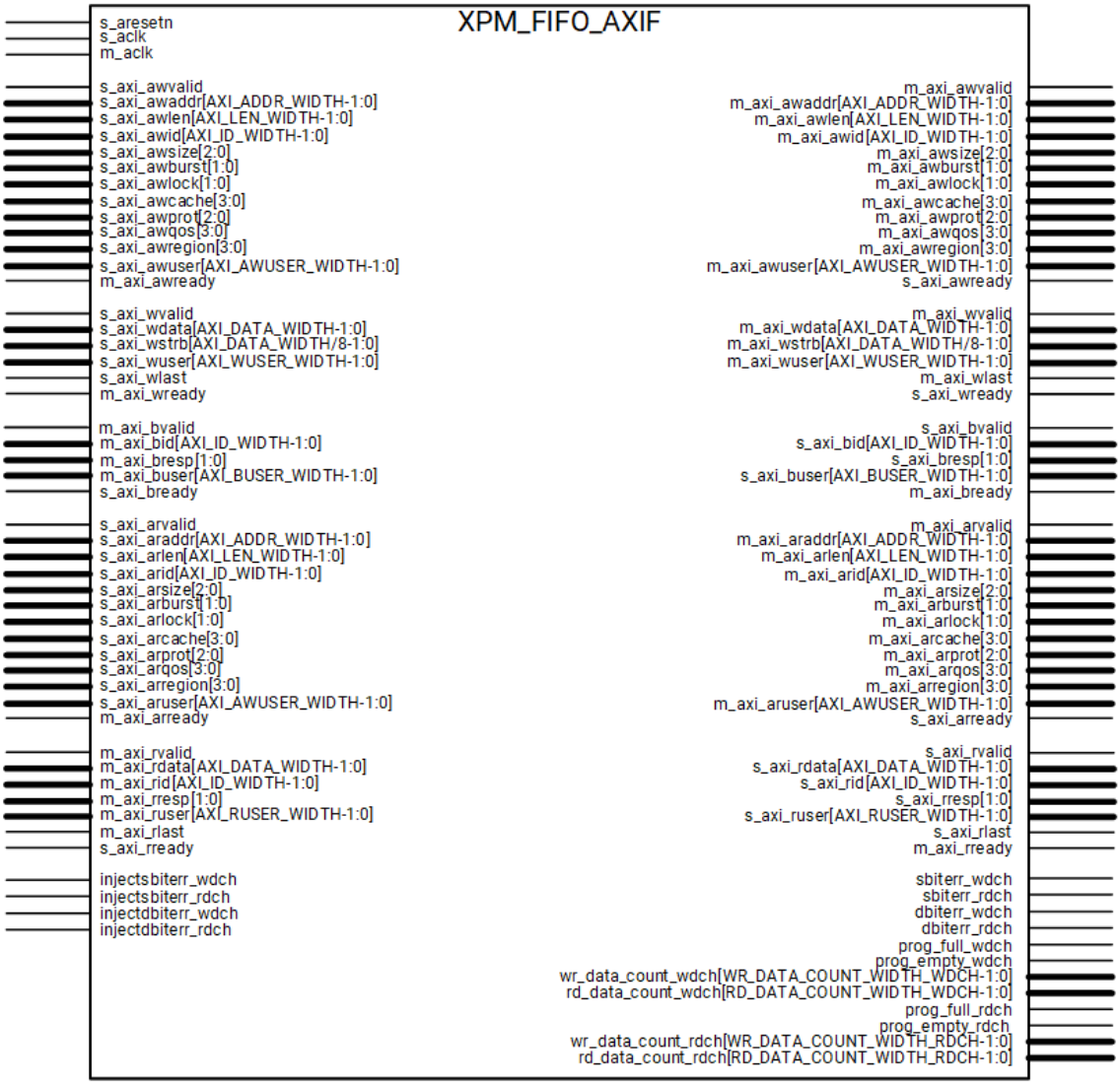
Como ya he explicado en entradas anteriores Xilinx cuenta con unas librerías llamadas XPM que permiten generar bloques FW de forma rápida. Estos bloques también son las primitivas en las que se basa Xilinx para crear sus bloques IP.

Vivado para hacer FIFOs tiene los siguientes bloques XPM:

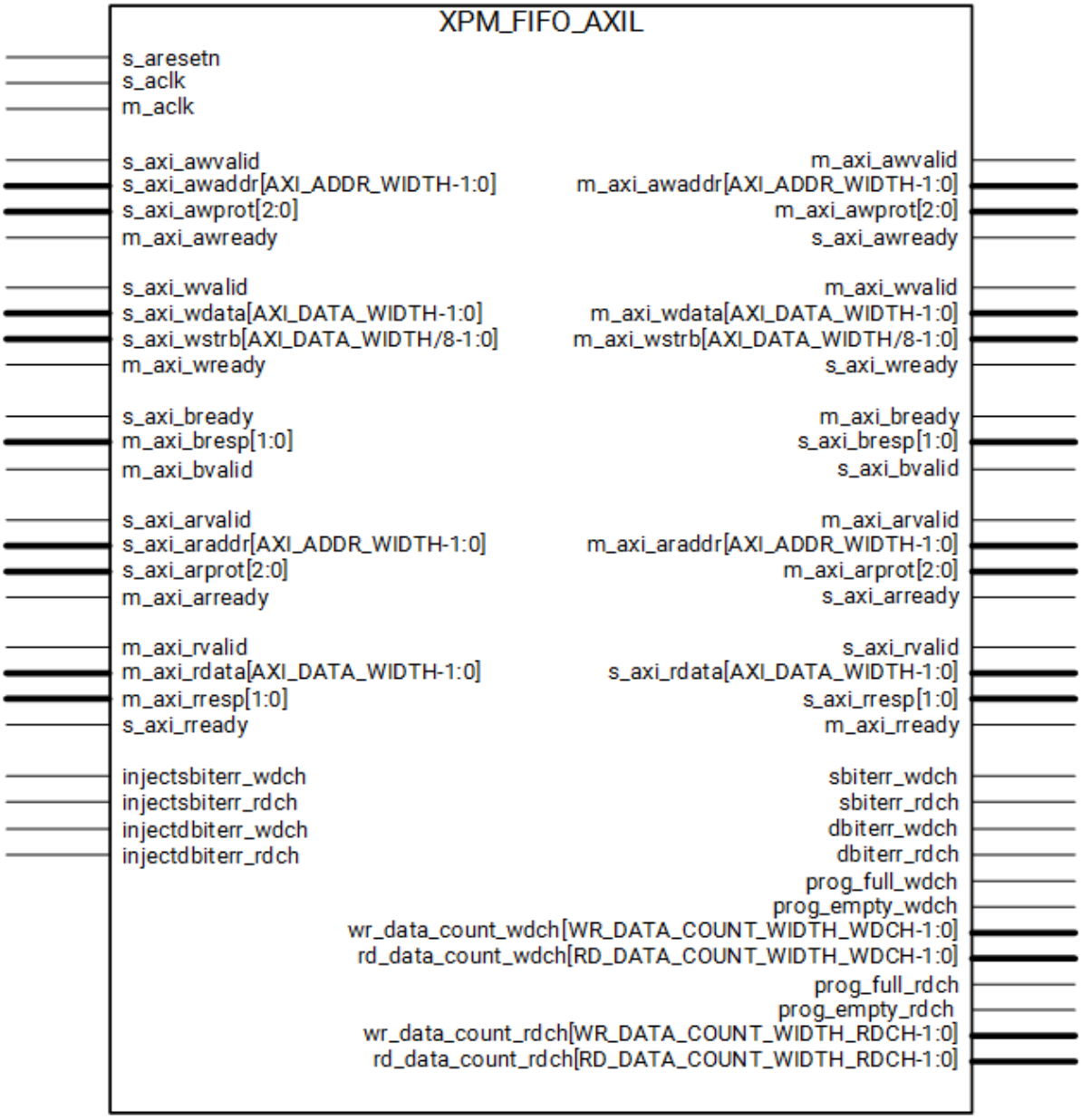
- **XPM_FIFO_ASYNC**: Este bloque permite crear *FIFOs asíncronos*



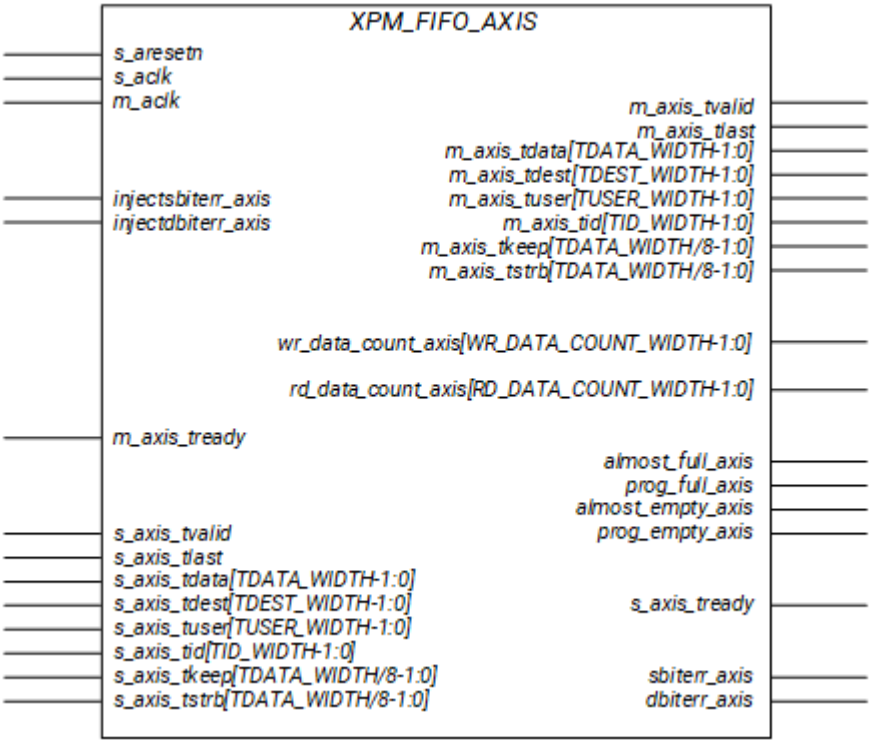
- **XPM_FIFO_AXIF**: Este bloque permite crear *FIFOs de tipo AXI Memory Mapped*



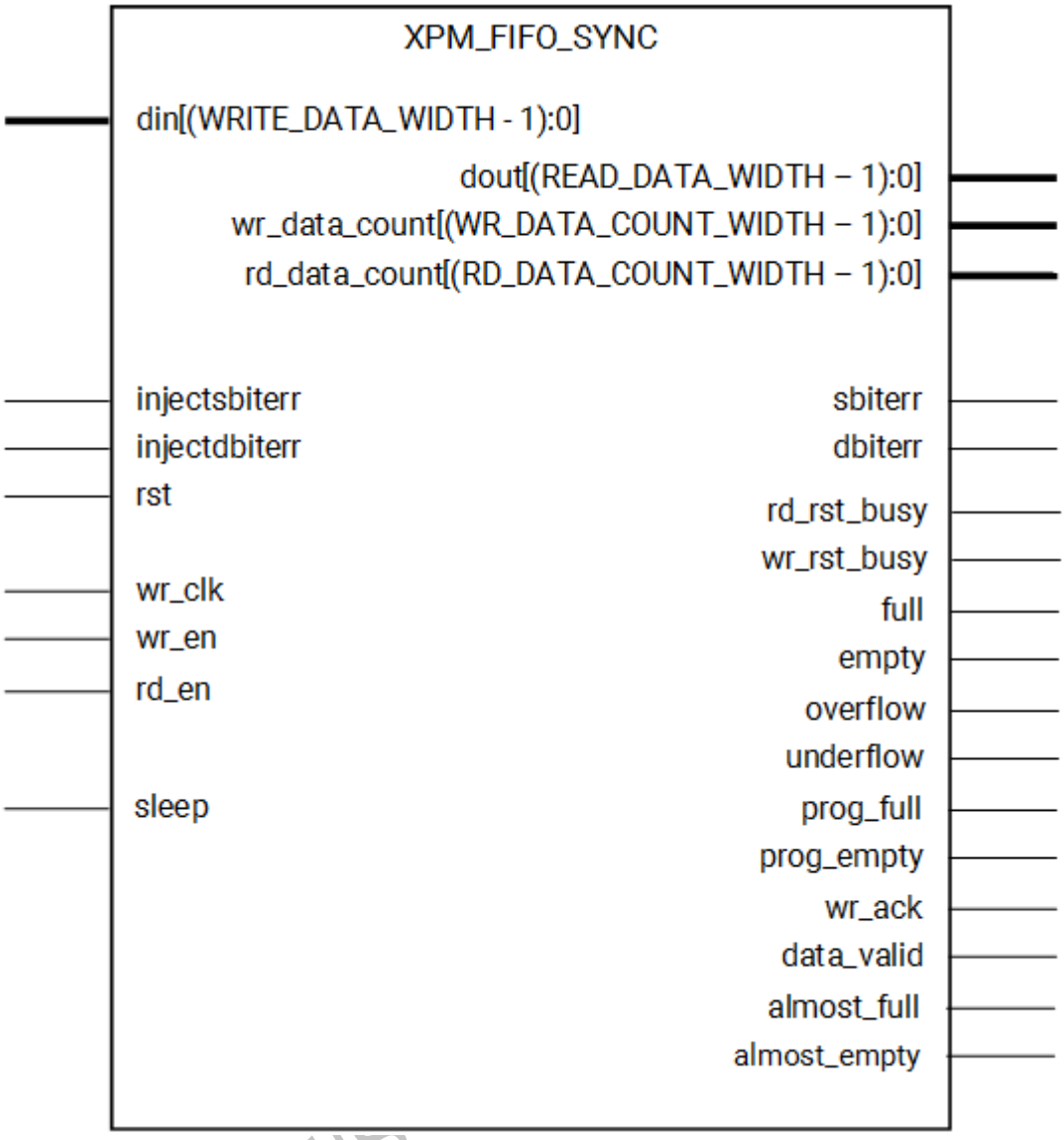
- **XPM_FIFO_AXIL**: Este bloque permite crear *FIFOs de tipo AXI Memory Mapped con AXI Lite*



- **XPM_FIFO_AXIS**: Este bloque permite crear *FIFOs con AXI Stream*



- **XPM_FIFO_SYNC**: Este bloque permite crear *FIFOs síncronos*



Documentación XPM