IP core Puente USB a WISHBONE

Rodrigo A. Melo, Salvador E. Tropea
Electrónica e Informática
Instituto Nacional de Tecnología Industrial
Buenos Aires, Argentina
Email: rmelo@inti.gob.ar, salvador@inti.gob.ar

***Resumen*—El *Universal Serial Bus* (USB) es actualmente el mecanismo de comunicación más usado para periféricos de computadoras personales. El mismo ha desplazado a los tradicionales puertos serie (RS-232) y paralelo (IEEE 1284).**

Uno de los estándares de interconexión ampliamente usado para SoCs (*System-on-Chip*) es el *WISHBONE*. En este trabajo presentamos un *core* que implementa un puente entre USB y *WISHBONE*. El *core* es un maestro *WISHBONE* capaz de controlar a cualquier esclavo del bus.

Este *core* fue verificado utilizando FPGAs y ofrece diversas configuraciones.

I. INTRODUCCIÓN

En la actualidad, las computadoras personales (PC) han dejado de contar con puertos de comunicación serie y paralelos. Los mismos han sido desplazados por el USB, que ofrece una amplia variedad de velocidades de comunicación, *plug and play* y otras ventajas.

Nuestro equipo de trabajo desarrolla sistemas embebidos que en la mayoría de los casos necesitan de validación en *hardware*, haciendo necesaria la comunicación con una PC. Por otra parte, utilizamos el bus *WISHBONE* [1] como medio de interconexión de nuestros *cores*.

Por lo expuesto, se planteó el desarrollo del *core* puente entre USB y *WISHBONE* (**usb2wb**). El mismo es un maestro *WISHBONE* controlado mediante el puerto USB. Esto nos permite validar en *hardware* descripciones que se comuniquen a través del bus *WISHBONE* o realizar sencillos periféricos abstrayéndonos de las dificultades inherentes a USB, las cuales ya están resueltas por el *core*.

Es importante destacar que este documento asume que el lector se encuentra familiarizado con la terminología usada en USB.

II. ARQUITECTURA

El diagrama en bloques de la implementación se encuentra en la Fig. 1.

El *core* usb2wb utiliza un *core* USB. La comunicación entre ambos se da por medio del *endpoint* 1, el cual se conecta a una *dual* FIFO. La dirección *out* del *endpoint* recibe comandos a ejecutar provenientes del *host* y los almacena en la FIFO *out*. Las respuestas se recogen en la FIFO *in* y se envían a la dirección *in* del *endpoint*. La máquina de estados es la encargada de traducir los comandos almacenados en la FIFO *out* a transacciones del bus *WISHBONE* y almacenar en la FIFO *in* las respuestas de los esclavos.

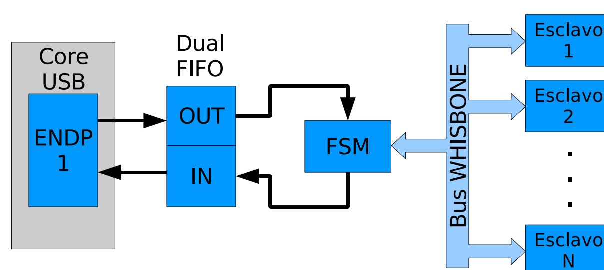

Figura 1. Diagrama en bloques de usb2wb.

II-A. *core USB*

Es un desarrollo propio de nuestro laboratorio, el cual en el *core* usb2wb resuelve la comunicación USB (*FS* o *HS*, seleccionable mediante un *generic*) brindando como interfaz el *endpoint* 1.

II-B. *dual FIFO*

Consta de dos *buffers* asociados a las direcciones *out* e *in* del *endpoint* 1. En un principio la FIFO *out* está libre y acepta recibir datos. Una vez llena y confirmados sus datos, pueden ser consumidos por el puente el cual los procesa y coloca los resultados en la FIFO *in*. Al terminar de procesar todos los datos el *endpoint in* está disponible. El *host* retira los datos y con esto la FIFO *out* vuelve a estar disponible para recibir nuevos datos.

II-C. *Máquina de estados*

La FSM (*Finite State Machine*) se compone de la **unidad de procesamiento** y la **unidad de ejecución**.

La **unidad de procesamiento** es la FSM en sí y es la encargada del diálogo con la *dual* FIFO y de la interpretación de los comandos. Los mismos son de 8 bits y se codifican de la siguiente manera:

Bits 7 y 6: especifican si el comando tiene 0, 1, 2 o 3 operandos.

Bit 5: especifica que se trata de una operación de escritura.

Bit 4: especifica que se trata de una operación de lectura.

Bits 3 a 0: código de la operación a realizar.

Así, la FSM pasa por estados de lectura de comando, lectura de operandos y ejecución de la operación.

La **unidad de ejecución** lee los operandos y ejecuta los comandos sobre el bus *WISHBONE*.

III. Comandos implementados

Se implementaron comandos de lectura, escritura y control, sobre el bus *WISHBONE*.

Algunos de estos comandos son combinaciones de otros comandos más básicos y se pueden habilitar y deshabilitar mediante *generics* para hacer el *core* más compacto.

A continuación, agrupados considerando que se habilitan/deshabilitan juntos, se presenta una pequeña descripción de los comandos implementados.

III-A. Comandos básicos

Estos comandos están siempre habilitados. Proveen la funcionalidad mínima del *core*.

- WBB_RESET: reset del puente usb2wb.
- WBB_WB_RESET: reset del bus *WISHBONE*.
- WBB_SET_ADDRESS: recibe como operando la dirección *WISHBONE* a utilizar y la configura en el bus.
- WBB_SET_VALUE: recibe como operando el valor a escribir sobre un registro utilizado por otros comandos.
- WBB_READ: lee un dato de la dirección *WISHBONE* actual.
- WBB_WRITE: utiliza el valor del registro escrito por WBB_SET_VALUE en la dirección *WISHBONE* actual.

III-B. Comandos de autoincremento

Comandos para habilitar/deshabilitar el incremento automático de la dirección *WISHBONE* luego de cada operación de lectura/escritura.

- WBB_AUTO_INC: habilita el incremento automático.
- WBB_NO_INC: deshabilita el incremento automático.

III-C. Comando con demora

Comando que permite implementar demoras.

- WBB_DELAY: recibe como operando el número de cuentas de la habilitación de entrada de datos, con lo cual genera demoras.

III-D. Comandos directos

Comandos que utilizan directamente los operandos que recibe como dirección del bus a utilizar o valor a escribir en el mismo.

- WBB_READ_ADR: lee de la dirección recibida como operando.
- WBB_WRITE_VALUE: escribe en la dirección actual el valor que recibe como operando.
- WBB_WRITE_ADR_VAL: escribe en la dirección que recibe como primer operando el valor que recibe como segundo operando.

III-E. Comandos de repetición

Comandos utilizados para repetir operaciones.

- WBB_SET_REPEAT: recibe como operando el valor de las repeticiones a utilizar con los comandos de repetición y de repetición con demoras (si se habilitan).

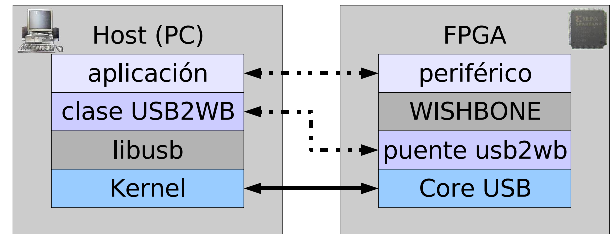

Figura 2. Comunicación entre una aplicación y el dispositivo *WISHBONE*.

- WBB_READ_REPEAT: utiliza el valor fijado por WBB_SET_REPEAT para realizar repeticiones de lecturas en la dirección *WISHBONE* actual.
- WBB_WRITE_REPEAT: utiliza el valor fijado por el comando WBB_SET_REPEAT para realizar repeticiones de escritura en la dirección *WISHBONE* actual.

III-F. Comandos de repetición con demoras

Se habilitan cuando tanto el comando con demoras como los comandos de repetición están habilitados.

- WBB_SET_DELAY: recibe como operando el valor de la demora a utilizar con los comandos de repetición con demoras.
- WBB_READ_REP_DLY: los valores fijados por los comandos WBB_SET_REPEAT y WBB_SET_DELAY son utilizados por este comando para realizar repeticiones de lecturas con demoras en la dirección *WISHBONE* actual.
- WBB_WRITE_REP_DLY: los valores fijados por los comandos WBB_SET_REPEAT y WBB_SET_DELAY son utilizados por este comando para realizar repeticiones de escrituras con demoras en la dirección *WISHBONE* actual.

IV. Uso desde la PC

IV-A. Clase USB2WB

Para simplificar el desarrollo de programas que se comunicarán con el puente USB a *WISHBONE*, se abstrajo la funcionalidad en una clase escrita en C++.

Un diagrama que indica las capas de la comunicación entre una aplicación corriendo en la PC y el dispositivo *WISHBONE* puede verse en la Fig. 2.

En la misma se puede observar que por el lado de la PC, una aplicación corriendo en ella utiliza la clase USB2WB, la cual internamente trabaja con la libusb (API ofrecida por Linux) que es la encargada de dialogar con el Kernel del sistema operativo, quien manejará al *host* para llevar a cabo las transacciones en el bus USB.

Por el lado de la FPGA, el *core* USB recibirá las transacciones del bus y dejará/tomará los datos del *endpoint* 1 con el cual se comunica con el puente de USB a *WISHBONE* que maneja el bus *WISHBONE* para dialogar con el/los esclavos, en este caso representado como un periférico.

IV-B. API USB2WB

El API (*Application Programming Interface*) de la clase USB2WB pone a disposición de la aplicación cuatro miembros, a saber:

- ***void USB2WB::BPush(unsigned v)***: encola el comando v en el *buffer* de transmisión de datos.
- ***void USB2WB::Send()***: envía al dispositivo los comandos almacenados en el *buffer*.
- ***void USB2WB::Send(int v, ...)***: envía al dispositivo los comandos indicados.
- ***const char *USB2WB::Get(unsigned size)***: recibe datos hasta un tamaño máximo *size* y los deposita en un *buffer*. Devuelve la dirección del *buffer* en caso de éxito y *NULL* en otro caso.

Todos los miembros implementan reintento en caso de errores por *time-out* y utilizan *signals* en el caso de errores que no pueden solucionarse.

V. Implementación y resultados obtenidos

V-A. Implementación

Nuestro *core* se implementó en VHDL93 estándar.

Para la simulación y validación se utilizó GHDL [2] 0.27, así como otras herramientas recomendadas por el proyecto FPGALibre [3].

La validación en *hardware* se llevó a cabo utilizando FPGAs *Spartan II* y *Spartan 3* de Xilinx y el *software* ISE WebPack 9.2.03i J.39.

El *host* utilizado fue una computadora personal corriendo el sistema operativo Debian [4] GNU [5] /Linux.

V-B. Resultados obtenidos

Se comparó el área ocupada por el core, para diversas configuraciones de comandos habilitados, corriendo a FS:

- Comandos básicos: 677 LUTs y 385 FFs (476 *slices*).
- Comandos de autoincremento: 681 LUTs y 386 FFs (478 *slices*).
- Comando con demora: 691 LUTs y 393 FFs (485 *slices*).
- Comandos directos: 708 LUTs y 396 FFs (490 *slices*).
- Comandos de repetición: 727 LUTs y 401 FFs (502 *slices*).
- Todos los comandos: 791 LUTs y 429 FFs (537 *slices*).

A modo de prueba se conectó al WISHBONE un pequeño periférico que permitiera encender los LEDs de una placa de desarrollo, así como encuestar el estado de los botones de la misma.

Se implementaron dos versiones del *core*, habilitando sólo los comandos de autoincremento:

- FS 716 LUTs y 404 FFs (496 *slices*)
- HS 927 LUTs y 396 FFs (571 *slices*).

VI. Conclusiones

Se obtuvo una implementación del *core* compacta que permite el uso de FPGAs pequeñas como la *Spartan II* 100.

La utilización de las herramientas propuestas por el proyecto FPGALibre demostró ser adecuada para un proyecto de estas características.

El puente permite validar en *hardware* de manera rápida y sencilla *cores* con interfaz *WISHBONE* utilizando una PC y *software* de alto nivel (C++).

El *core* es también una alternativa interesante para el rápido desarrollo de dispositivos USB de pequeña y mediana envergadura.

El uso de la clase USB2WB desarrollada, resume la realización de un *software* de aplicación, al uso de simples funciones de envío y recepción, haciendo innecesario por parte del programador conocimientos sobre el protocolo USB.

Referencias

[1] Silicore and OpenCores.Org, "Wishbone system-on-chip (soc) interconnection architecture for portable ip cores," http://prdownloads.sf.net/-fpgalibre/wbspec_b3-2.pdf?download.

[2] T. Gingold. (2008, Dec.) A complete VHDL simulator. [Online]. Available: http://ghdl.free.fr/

[3] S. E. Tropea, D. J. Brengi, and J. P. D. Borgna, "FPGAlibre: Herramientas de software libre para diseño con FPGAs," in *FPGA Based Systems*. Mar del Plata: Surlabs Project, II SPL, 2006, pp. 173–180.

[4] I. Murdock *et al.* Debian gnu/linux operating system. [Online]. Available: http://www.debian.org/

[5] R. M. Stallman *et al.* The GNU project. [Online]. Available: http://www.gnu.org/