Core USB y software asociado

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

En este trabajo presentamos un *core* USB que implementa la mayor parte de la funcionalidad del estándar 2.0, excluyendo el modo isócrono, así como también las herramientas desarrolladas para utilizar y verificar el *core*.

El mismo fue verificado utilizando FPGAs y ofrece una amplia variedad de configuraciones.

I. Introducción

Las computadoras personales (PC) actuales han dejado de poseer 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 muchos casos funcionan como periféricos de una PC. Debido a que en la actualidad el USB es el mecanismo de comunicación más usado se planteó el desarrollo de un *core* USB.

En este trabajo presentamos las características del *core* desarrollado, así como las herramientas auxiliares desarrolladas para su verificación y uso. En la sección II exponemos los objetivos de dicho desarrollo. Las partes que componen el *core* se explican en la sección III. La sección IV explica las herramientas desarrolladas para su verificación y uso así como también las metodologías utilizadas para la verificación. Los detalles de implemantación y resultados obtenidos se discuten en la sección V y finalmente se exponen las conclusiones en la sección VI.

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

II. Objetivos

Los objetivos del proyecto fueron desarrollar un *core* con las siguientes características:

- Sintetizable para la mayor parte de las FPGAs disponibles en el mercado.
- Bajo costo comparado con la opción de utilizar una solución externa.
- Ofrecer configuraciones muy compactas que permitan su uso con mínimos requerimientos de área y componentes externos.

Para lograr un *core* sintetizable en una amplia gama de FPGAs, así como dejar abierta la posibilidad de usar el *core* en un ASIC, se utilizó el lenguaje VHDL 93 estándar.

Para cumplir con el segundo objetivo se debió tener en cuenta que el costo de soluciones USB completas, como las de la línea EX-USB FX2 de Cypress, es comparable al de una FPGA equivalente a 200.000 gates (*Spartan 3* 200, 3.840 LUTs+FFs). Esto impuso una primer restricción en área a ser consumida y al costo de los componentes externos necesarios para la implementación.

El tercer objetivo responde a que uno de los usos previstos para este *core* fue el de utilizarlo para verificar periféricos implementados en FPGAs de bajo costo. En nuestro laboratorio poseemos varias placas de desarrollo con *Spartan II* 100 [1] (2.400 LUTs+FFs). Esto impuso una segunda restricción al área disponible.

III. Capas o niveles

A los fines de dividir el problema en partes menores que pudieran ser intercambiables y poder comparar nuestro *core* con otros ya existentes se adoptó la siguiente división en capas o niveles:

- **Nivel eléctrico**: adaptación de niveles de tensión y/o corriente.
- **Nivel físico o PHY**: se adoptó lo especificado por *USB 2.0 Transceiver Macrocell Interface* (UTMI). Convierte de paralelo a serie, codifica los bits para que sea posible sincronizar los relojes y señaliza el comienzo y fin de los paquetes. En la recepción realiza las tareas inversas.
- **Nivel de *handshake* o SIE**: normalmente conocido como *Serial Interface Engine* (SIE). Se encarga de realizar las transacciones básicas del protocolo USB.
- **Nivel de unión con la SIE**: se encarga de comunicar la SIE con el nivel superior.
- **Nivel de protocolo**: encargado de implementar la parte más alta del protocolo USB, incluyendo la funcionalidad de *plug and play*.
- **Nivel de aplicación**: donde se encuentra la aplicación propiamente dicha, es decir el periférico.

III-A. Nivel eléctrico

Los niveles de tensión especificados por el estándar USB para las velocidades más bajas (baja o LS 1,5 Mb/s y completa o FS 12 Mb/s) son compatibles con los entregados por la mayor parte de las FPGAs cuando se evalúa desde el punto

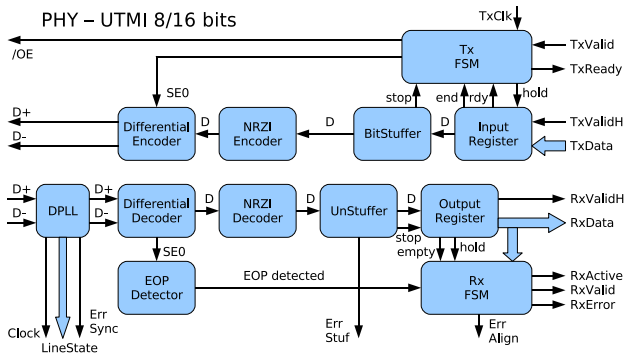

Figura 1. Diagrama en bloques del PHY.

de vista del extremo que transmite, pero no del que recibe. En condiciones ideales, como las de un laboratorio, es posible implementar un periférico USB con sólo unas pocas resistencias externas a la FPGA. Pero cuando se desea cumplir con la especificación es necesario utilizar un driver externo.

En el caso de alta velocidad (HS 480 Mb/s) la señalización es por corriente y por lo tanto es indispensable un driver externo.

III-B. Nivel físico o PHY

Se implementó un PHY compatible con UTMI cuyo diagrama en bloques se encuentra en la Fig. 1. Las características del mismo son: soporte para LS y FS, interfaz de 8/16 bits y frecuencia de reloj de 48 MHz para FS (6 MHz para LS). Para HS consideramos que no era posible implementar el nivel físico de la especificación utilizando FPGAs de bajo costo.

Este bloque se encarga de dos tareas distintas, la recepción y la transmisión de paquetes. La sección de recepción se encarga de la recuperación de reloj, decodificación de los bits (NRZI y bitstuffing), conversión serie a paralelo y detección de inicio/fin de paquete. La de transmisión se encarga de la conversión paralelo a serie, señalización de inicio y fin de paquete y de la codificación de los bits.

El bloque de recepción detecta errores de bitstuffing, alineación y sincronismo.

III-C. Nivel de handshake o SIE

Este nivel implementa las transacciones básicas del protocolo USB. Las mismas se componen de dos o más paquetes intercambiados entre el *host* (maestro) y el dispositivo (esclavo). En nuestra implementación dejamos de lado el modo isócrono e implementamos los modos de trasmisión de *control, interrupt* y *bulk*.

La implementación se dividió como se muestra en la Fig. 2. El packer se encarga de ensamblar los paquetes a ser transmitidos. Los paquetes soportados son los de datos DATA0 y DATA1 y los de *handshake* ACK, NAK, STALL y NYET. En los primeros es necesario computar un CRC (Código de Redundancia Cíclica) de 16 bits. El unpacker se encarga de desensamblar los paquetes recibidos. Los paquetes soportados son los *tokens* IN, OUT y SETUP, que incluyen un CRC de 5 bits, los de datos DATA0 y DATA1, que incluyen un CRC de 16 bits y los de *handshake* ACK, NAK y STALL. Cualquier otro tipo de paquete es descartado. Este bloque reporta errores de: paquete incompleto, falla de CRC e identificador de paquete erróneo, no mostrados en el diagrama en bloques por cuestiones de espacio. La máquina de estados principal se encarga de asegurar que las transacciones sean completadas y de seleccionar las contestaciones adecuadas para cada requerimiento. Una máquina de estados separada se encarga de detectar las señalizaciones de *reset* y *suspend* en el *bus*. Durante el *reset*, y en el caso de haber sido configurado para HS, esta máquina de estados se encarga de la negociación para pasar de FS a HS. Cuando el *host* indica que enviará datos (SETUP u OUT), o cuando se espera una respuesta ACK del *host*, la especificación pone un límite al tiempo para esperar dichos paquetes. El bloque *Bus Turn-around T-O* se encarga de monitorear este tiempo. El bloque de *Timers* se encarga de generar varias temporizaciones: 2,5 μs (*reset*), 100 μs y 1 ms (negociación) y 3 ms (*suspend* y negociación).

El *bus Token Info* incluye información sobre el *token* recibido (número de *endpoint*, tipo de paquete, etc.), *EP Control* incluye señales para colocar/sacar el *endpoint* en/del modo STALL, *Bus Status* señaliza el estado actual del *bus* (*idle, reset, active*), *EP Status* permite indicar el estado actual del *endpoint* (disponible, STALL, aún no disponible) y que tipo de paquetes de datos se espera o se desea enviar (DATA0/1), finalmente *EP Mode* le indica a este bloque que modos soporta el *endpoint* (IN, OUT, SETUP).

La interfaz con el PHY y con el nivel superior son configurables para 8 y 16 bits de datos. El reloj de este bloque puede ser el mismo usado en el PHY o la mitad.

III-D. Nivel de unión con la SIE

En la mayor parte de los *cores* USB disponibles [2] [3] [4] [5] [6] este nivel se compone de registros de control y estado junto con los *buffers* utilizados para los datos recibidos o a ser transmitidos. Este esquema es muy versatil y se acomoda perfectamente al uso del *core* junto con una CPU.

En nuestro caso buscamos lograr optimizar el uso de recursos y determinamos que en muchos casos no es necesario utilizar una CPU completa para los niveles superiores. También determinamos que en muchos casos no es necesario utilizar *buffers* ya que los datos son volátiles. Por estas razones decidimos simplificar este nivel de manera tal que se componga sólo de multiplexores que permiten enrutar los datos a los distintos *enpoints*.

III-E. Nivel de protocolo

Una de las características más importantes de USB es el soporte de *plug and play* y de múltiples configuraciones para un mismo dispositivo, así como más de una funcionalidad en el mismo. Para todo esto el *host* debe poder dialogar con el dispositivo. Este diálogo se realiza a través de los llamados *requests*. La información del dispositivo se alberga en estructuras llamadas *descriptors*.

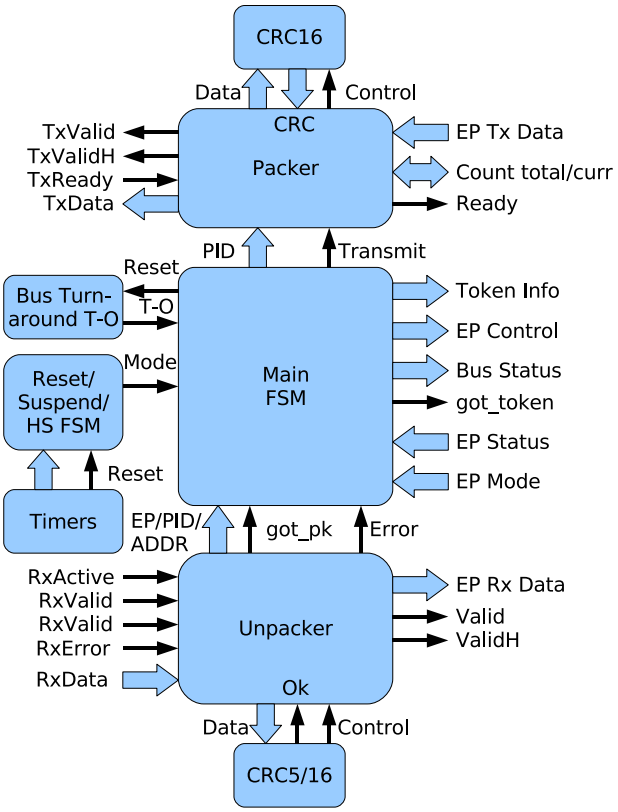

Figura 2. Diagrama en bloques de la SIE.

La implementación de las contestaciones a los *requests* y el manejo de los *descriptors* se puede realizar en *software* [4] [2]. Esta opción es muy flexible ya que no es necesario modificar el *core* para cada dispositivo. En otros casos se ofrece un número fijo de *enpoints* y este nivel se une con el anterior [3] [6]. Esta opción permite simplificar el *software*, pero requiere de cambios en el *core* si los *enpoints* ofrecidos no son los adecuados para el dispositivo en cuestión.

Para lograr implementaciones USB compactas decidimos implementar este nivel en *hardware*. Debido a que los *requests* que soporta cada dispositivo son variables y, a que optamos por evitar imponer limitaciones a la cantidad y tipo de *enpoints* usados, diseñamos una arquitectura modular que permite agregar módulos que implementan cada *request* y cada *endpoint*.

Para facilitar la tarea de desarrollo identificamos un conjunto de *requests* básicos que son normalmente soportados por dispositivos USB y los encapsulamos en un *core* que ofrece esta funcionalidad básica (*EP0 Base*). La Fig. 3 muestra un diagrama en bloques simplificado del esquema modular implementado.

III-F. Nivel de aplicación

En este nivel es donde se coloca la funcionalidad de nuestro dispositivo. Gracias a una implementación en *hardware* del nivel inferior es posible realizar una implementación compacta de este nivel. En muchos casos este nivel se reduce a unos pocos regisros junto con el *hardware* específico de nuestra aplicación. En caso de que un *endpoint* necesite *buffers* de datos es posible implementarlos en este nivel, pero en el caso de *enpoints* simples es posible obviarlos.

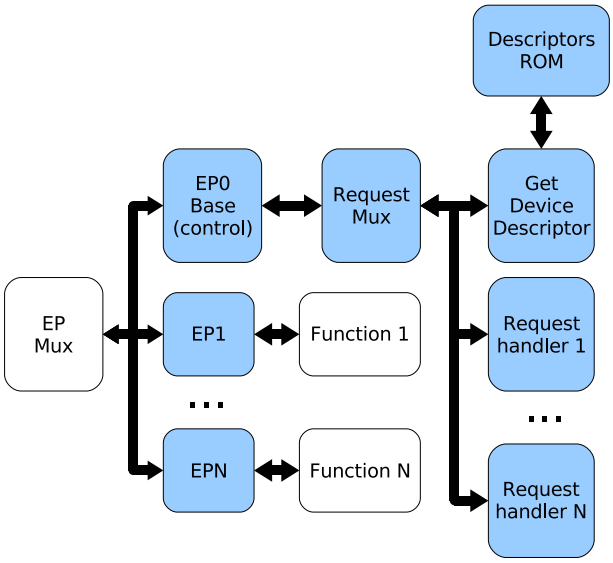

Figura 3. Arquitectura modular de *enpoints* y *requests*.

IV. Validación y herramientas auxiliares

IV-A. Herramientas de validación

El USB es complejo y, como se explicó anteriormente, se compone de varios niveles. Para su validación utilizamos bancos de prueba automáticos que verifican cada parte del *core* por separado y luego el funcionamiento en conjunto. Debido a este esquema modular se decidió utilizar la estrategia utilizada en UNIX, es decir desarrollar pequeñas herramientas que luego pudieran ser utilizadas en conjunto. Las herramientas se desarrollaron en C y C++ y se utilizaron *pipes* como mecanismo de comunicación.

Para la validación del PHY desarrollamos las siguientes herramientas:

- **gen-dpll-diff**: genera una sequencia aleatoria de bits, utilizando un reloj que posee un corrimiento en el tiempo. La máxima desviación del reloj es la indicada por la especificación. Se utiliza para verificar el *DPLL* y el *Differential Decoder*.
- **nrzi_en**: codifica los bits en formato NRZI (*Non return to zero, inverted*). Utilizado para verificar el *NRZI Encoder*.
- **nrzi_de**: decodifica bits en formato NRZI. Utilizado para verificar el *NRZI Decoder*.
- **bitstuffer**: agrega los bits indicados por la especificación. Utilizado para verificar el *BitStuffer*.
- **bitunstuffer**: elimina los bits agregados por el bitstuffer Utilizado para verificar el *UnStuffer*.
- **usb-make-packet**: permite ensamblar y desensamblar paquetes USB. Se encarga de agregar o remover el inicio de paquete (SYNC) y el fin de paquete (EOP). Para

de/codificar los bits adecuadamente hace uso de nrzi_en, nrzi_de, bitstuffer y bitunstuffer. Permite verificar el funcionamiento del PHY completo.

Para la validación de la SIE:

- **crc**: computa el CRC de 5 y 16 de USB.
- **usb_to_bits**: permite generar transacciones USB completas. Interpreta un archivo de texto donde se dice que transacciones se desea generar y opcionalmente si las mismas contendrán algún error. Utiliza usb-make-packet y crc para generar los paquetes USB. Utilizado para verificar el *Unpacker* y la SIE completa.

IV-B. Herramientas de validación y desarrollo

Se desarrollaron otras dos herramientas que no sólo fueron útiles para la verificación sino que también lo son para el desarrollo de dispositivos USB:

- **hid-report**: los dispositivos HID (Human Interface Device) pueden indicarle al *host* que formato de datos utilizan. Ejemplos de dispositivos HID son el teclado, el ratón y el *joystick*. La especificación HID utiliza en los ejemplos un pseudo-lenguaje. Este programa es capaz de interpretar dicho pseudo-lenguaje y compilarlo al *descriptor* equivalente.
- **usb-descriptor**: una parte compleja del desarrollo de dispositivos USB es la creación de los *descriptors* que le permiten al *host* saber como comunicarse con nuestro dispositivo. La creación manual de dichos *descriptors* es una tarea pesada y propensa a errores. El problema es aún mayor cuando se desea modificar algo, por ejemplo agregar un nuevo *endpoint*, en este caso es posible que olvidemos corregir todos los datos que se verán afectados. Para simplificar esta tarea usb-descriptor interpreta una descripción de alto nivel de nuestro dispositivo y crea los *descriptors* adecuados. En el caso de tratarse de un dispositivo HID utiliza hid-report para los *descriptors* HID. Adicionalmente, este programa genera el código VHDL necesario para crear la ROM que conten-drá los *descriptors*, así como un *package* con las posiciones de memoria donde se encuentra cada *descriptor*.

Utilizando usb-descriptor y usb_to_bits fue posible verificar dispositivos completos utilizando transacciones de inicialización reales. Para conocer un mecanismo de inicialización real se observó una inicialización completa con un osciloscopio digital. Luego se utilizaron herramientas como usb-make-packet para desensamblar los paquetes observados y finalmente se escribió dicha incialización en los lenguajes soportados por usb-to-bits y usb-descriptor. Para esto se utilizó como base un teclado USB. Estas transacciones permitieron crear secuencias reales para nuestros dispositivos. Luego se crearon secuencias con errores para verificar el comportamiento ante los mismos.

V. Implementación y resultados obtenidos

V-A. Implementación

Nuestro *core* se implementó en VHDL93 estándar, evitando utilizar detalles específicos de algún dispositivo en particular. Para la simulación y validación se utilizó GHDL [7] 0.27, así como otras herramientas recomendadas por el proyecto FPGALibre [8]. 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.

Para LS y FS se utilizó como driver el ISP1106 cuyo costo es de aproximadamente un 5 % del de una solución completa. Se verificó que en condiciones de laboratorio y con un cable de 1,5 m es posible conectar una *Spartan II* directamente al *bus* USB. Para HS se utilizó una UTMI CY7C68000 de Cypress conectada a una *Spartan 3* con un *bus* de 16 bits a 30 MHz. Este componente tiene un costo de aproximadamente el 15 % del de una solución completa. En ambos casos el área de circuito impreso necesaria es mucho menor al de un chip que implementa una solución completa.

Para el caso de LS y FS se utilizó una *Spartan II* y para HS una *Spartan 3*. Esto sólo se debió a que la UTMI externa se encontraba en una placa de desarrollo basada en *Spartan 3*. El *host* utilizado fue una computadora personal corriendo el sistema operativo Debian [9] GNU [10] /Linux. El software para controlar nuestros dispositivos se escribió en C++ utilizando dos APIs diferentes ofrecidas por Linux: hiddev y libusb.

V-B. Resultados obtenidos

Se desarrollaron los siguientes dispositivos de prueba:

- **Adaptador de *joystick* analógico a USB**: el mismo se reporta a la PC como un *joystick* USB y permite usar un viejo *joystick* analógico de PC. Este dispositivo implementa HID y posee un único *endpoint*. Se implementaron dos conversores A/D utilizando un doble comparador y midiendo el tiempo de carga de capacitores a través de los potenciómetros del *joystick* analógico. Este dispositivo insumió 708 LUTs y 427 FFs (493 *slices*) junto con un BRAM. Se verificó el funcionamiento del *joystick* en GNU/Linux y Windows XP.
- **Dispositivo HID genérico**: a los fines de experimentar con la API HID de Linux se desarrolló un *core* que permitiera encender los LEDs de la placa de desarrollo, así como encuestar el estado de los botones de la misma. Dicho dispositivo se reporta como HID pero con una funcionalidad definida por el fabricante. El mismo posee dos *enpoints*. Este dispositivo insumió 692 LUTs y 402 FFs (488 *slices*) junto con un BRAM.
- **Dispositivo USB genérico**: a los fines de experimentar con la API libusb de Linux se desarrolló un *core* similar al anterior pero que no implementara HID. Adicionalmente, se agregó un *endpoint* con modo de transferencia *bulk* que permitiera leer el contenido de la ROM de *descriptors*. Se implementaron tres versiones de este *core*, dos trabajando en FS y una en HS. La versión FS se implementó utilizando *buses* de 8 y 16 bits. Se obtuvieron los siguientes resultados: FS/8 bits 584 LUTs y 349 FFs (418 *slices*), FS/16 bits 925 LUTs y 466 FFs (591 *slices*) y HS 860 LUTs y 347 FFs (511 *slices*).

- **Puente USB a WISHBONE**: debido a que habitualmente usamos el *bus* WISHBONE se identificó a este *core* como muy interesante. El mismo permite controlar el *bus* WISHBONE desde una PC utilizando USB. Esto permite controlar cualquier periférico WISHBONE desde la PC y resulta muy útil a la hora de verificarlos. A modo de prueba se conectó al *bus* WISHBONE un pequeño periférico que permitiera encender los LEDs de la placa de desarrollo, así como encuestar el estado de los botones de la misma. Se implementaron dos versiones del *core*: FS 716 LUTs y 404 FFs (496 *slices*) y HS 927 LUTs y 396 FFs (571 *slices*).

V-C. Otras implementaciones

No es simple comparar estos resultados con los recursos insumidos por otras implementaciones USB ya que sería necesario comparar las mismas aplicaciones. Otra forma sería comparar bloques con igual funcionalidad, pero la mayor parte de las implementaciones realizan separaciones diferentes. Por estas razones sólo es posible realizar una comparación aproximada.

El *core* usbhostslave [5] de OpenCores incluye un ejemplo de implementación de un ratón. El mismo es comparable a la implementación de nuestro *joystick*. La síntesis de usbhostslave arrojó un consumo de 2511 LUTs y 1631 FFs (1704 *slices*) para una *Spartan 3*. Estos valores fueron corregidos para descontar 512 LUTs utilizadas como memoria de doble puerto que estimamos podría mapearse a un BRAM. Esta implementación incluye el PHY, SIE, una interfaz con registros y FIFOs y el ratón propiamente dicho.

En otros casos no se disponía de una aplicación completa, por lo que medimos el área insumida por la SIE y la interfaz con el nivel superior. A esto debería agregarse la implementación de alto nivel del protocolo, el PHY y la aplicación propiamente dicha: Xilinx [2] 2340 LUTs y 610 FFs (1235 *slices*) 4 BRAMs (8 *enpoints* usando *Spartan 3* o superior, interfaz con *bus* OPB), usb [4] de OpenCores 2899 LUTs y 1712 FFs (1985 *slices*) con memoria externa (16 *enpoints* usando *Spartan 3*, interfaz con *bus* WISHBONE) y GRUSBDC [11] de Aeroflex/Gaisler 3500 LUTs y 2 BRAMs en su configuración mínima (hasta 32 *enpoints* usando *Spartan 3*, interfaz con *bus* Amba). Otro ejemplo es la implementación de ComBlock [3] que implementa el nivel de protocolo en *hardware* de una manera muy rígida y ocupa 1449 LUTs y 620 FFs (854 *slices*) 5 BRAMs (4 *enpoints* usando Virtex-2). En estos tres casos se trata de *cores* con soporte para HS.

Otro proyecto que se puede tomar como referencia es el usb1_fun [6] de OpenCores que incluye un PHY, la SIE y un nivel superior de protocolo implementado en *hardware* que reporta 6 *enpoints* fijos. El mismo no incluye ninguna aplicación y necesitaría de cambios muy importantes para implementar un dispositivo HID. No soporta HS e insume 978 LUTs y 518 FFs (670 *slices*) y un BRAM.

VI. Conclusiones

Se obtuvo una implementación de USB compacta que permite el uso de FPGAs pequeñas como la *Spartan II*. En el caso de una aplicación USB simple se consumió sólo el 35 % de una *Spartan II* 100.

El número de componentes externos se pudo reducir al mínimo mediante la implementación de un PHY que soporta LS y FS. Sólo es necesario agregar un chip económico como el ISP1106 que ocupa muy poca área de impreso. En condiciones de laboratorio se verificó que es posible conectar la FPGA directamente al *bus* USB.

El costo del área insumida es bajo, aprox. 22 % de una *Spartan 3* 200, por lo que es mucho más conveniente que el uso de una solución externa completa. La utilización del estándar UTMI permitió utilizar un PHY externo que soportara HS.

Las herramientas hid-report y usb-descriptor mostraron ser de gran utilidad a la hora de desarrollar aplicaciones demostrativas. Por otro lado, la estrategia de crear pequeñas herramientas modulares que pudieran ser reusadas por otras herramientas mayores permitió el reuso de código de una manera adecuada.

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

Si bien no es posible realizar una comparación totalmente justa con otras implementaciones, es posible afirmar que nuestra implementación es mucho más compacta. Si se compara el ratón [5] con nuestro *joystick* se ve que el nuestro ocupa menos de una tercera parte. Incluso cuando se compara con *cores* comerciales como el de ComBlock y Xilinx se ve que, aún cuando no implementan un dispositivo completo, ocupan más área que cualquiera de nuestros dispositivos demostrativos. Concluimos que la principal diferencia se encuentra en haber eliminado la interfaz entre la SIE y el nivel superior. Esto no sólo permite un ahorro importante sino que también simplifica la implementación en *hardware* de los niveles superiores.

Referencias

[1] D. J. Brengi, S. E. Tropea, and J. P. D. Borgna, "Tarjeta de diseño abierto para desarrollo y educación," in *2007 3rd Southern Conference on Programmable Logic Designer Forum Proceedings*, Mar del Plata, 2007, pp. 57–60.

[2] (2008, Dec.) OPB universal serial bus device (v1.00a). Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/usb_ds591.pdf

[3] (2008, Dec.) USB 2.0 interface user manual. ComBlock. [Online]. Available: http://www.comblock.com/ download/USB20_UserManual.pdf

[4] R. Usselmann. (2008, Dec.) USB 2.0 function core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/usb

[5] S. Fielding. (2008, Dec.) USB 1.1 host and function ip core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/usbhostslave

[6] R. Usselmann. (2008, Dec.) USB 1.1 function ip core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/usb1_funct/overview

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

[8] 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.

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

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

[11] (2008, Dec.) USB 2.0 device controller. Aeroflex / Gaisler. [Online]. Available: http://www.gaisler.com/