

Sistema operativo para SMPs basados en MicroBlaze

Huerta Pellitero P., Castillo Villar J., Sánchez de la Lama C., Martínez Torre J. I

Escuela Técnica Superior de Ingeniería Informática, URJC, Madrid, España,
{pablo.huerta, javier.castillo, carlos.delalama, joseignacio.martinez}@urjc.es

Abstract. Un tipo de sistemas que está ganando popularidad en el ámbito del diseño en FPGA son los sistemas multiprocesador basados en procesadores *soft-core*, tanto utilizando arquitecturas de propósito específico para resolver un problema concreto, como arquitecturas de propósito más general como son los sistemas de multiprocesamiento simétrico, también llamados sistemas SMP. Una de las principales carencias de este tipo de sistemas es la escasez de sistemas operativos que permitan desarrollar aplicaciones multi-hilo que aprovechen la existencia de varios procesadores en el sistema sobre el que se están ejecutando.

El presente artículo detalla la implementación de un sistema operativo para sistemas de multiprocesamiento simétrico en FPGA basados en el procesador *soft-core* MicroBlaze. También se muestra un sistema SMP sobre el que se ejecutan una serie de aplicaciones para probar el funcionamiento de dicho sistema operativo.

1 Introducción

Los procesadores *soft-core* o SCP se están popularizando mucho en los diseños en FPGA y cada fabricante ofrece su propio SCP especialmente optimizado para sus propias FPGAs, como hace Xilinx con su procesador MicroBlaze [7] o Altera con su procesador Nios II [2]. También existe una amplia variedad de SCP distribuidos en forma de hardware libre como el procesador OpenRISC [3] o el LEON2 y el LEON3 [4].

El número de procesadores que se pueden incluir en un diseño en FPGA solamente está limitado por los recursos físicos de la FPGA, permitiendo así el diseño de distintas arquitecturas multiprocesador tanto para sistemas de propósito general como para sistemas de propósito específico.

Una de las posibles arquitecturas multiprocesador que se pueden implementar en FPGA utilizando SCPs son los sistemas SMP (*symmetric multiprocessor*) [1]. Este artículo presenta un sistema operativo con interfaz POSIX para ser utilizado en sistemas SMP basados en el SCP MicroBlaze, así como un ejemplo de sistema SMP para comprobar el correcto funcionamiento del sistema operativo.

En la sección 2 se introduce la arquitectura general del sistema operativo que se ha implementado, presentando detalladamente sus principales características internas así como la capa de abstracción de hardware que permite que pueda ser portado fácilmente para funcionar en un sistema SMP concreto. En la sección 3 se presenta una aplicación multi-hilo que hace uso del sistema operativo desarrollado y que se ejecuta sobre un sistema SMP con 4 procesadores MicroBlaze. Finalmente, en la sección 4 se muestran las conclusiones y las líneas de trabajo futuro en el ámbito de los sistemas SMP en FPGA.

Este trabajo ha sido financiado por la Universidad Rey Juan Carlos y la Comunidad de Madrid mediante el programa URJC-CM-2007-CET-1550.

2 Estructura del S.O

El sistema operativo que se ha desarrollado está basado en Xilkernel [8], el *microkernel* proporcionado por Xilinx para trabajar con los procesadores MicroBlaze y PowerPC, y se ha modificado para que pueda soportar de forma transparente y eficiente sistemas SMP. En esta sección se mostrarán los requisitos hardware que son necesarios para que el S.O implementado pueda funcionar [6], y se presentarán los fundamentos de funcionamiento de dicho S.O.

2.1 Requisitos de la arquitectura hardware

Una arquitectura SMP que vaya a ejecutar el sistema operativo que se va a presentar debe disponer de una serie de características que permitan al sistema operativo realizar las operaciones necesarias para poder gestionar el reparto de tareas entre procesadores, la comunicación y la sincronización entre procesos, etc. Las características hardware imprescindibles para que el S.O pueda realizar dichas tareas son:

- Una región grande de memoria compartida, mapeada en las mismas direcciones para todos los procesadores. En esta región residirán tanto el sistema operativo como la aplicación de usuario.
- Una pequeña región de memoria no compartida en cada uno de los procesadores, que será usada como pila cuando el sistema ejecute ciertas funciones en modo *kernel*.
- Un mecanismo de identificación del procesador. El sistema operativo necesitará en ciertas ocasiones ejecutar un código u otro dependiendo de qué procesador esté ejecutando dicho código.
- Un mecanismo hardware de sincronización que permita proporcionar acceso exclusivo a secciones críticas del código.

2.2 Hardware Abstraction Layer

El sistema desarrollado puede ser utilizado por diversos tipos de arquitecturas SMP, con diferentes características o implementaciones de los requisitos mencionados en el punto anterior. Para abstraer la parte funcional del sistema operativo de la parte dependiente del hardware se ha creado una capa de abstracción del hardware o HAL (*Hardware Abstraction Layer*) donde se implementan las partes específicas del sistema SMP sobre el que se quiere utilizar el sistema operativo.

En el HAL se describe cómo y dónde están implementadas para una arquitectura SMP concreta las características mencionadas en el punto anterior. Con esto se consigue que adaptar el sistema operativo para una arquitectura concreta sea una tarea sencilla, ya que solamente se debe modificar el HAL para que recoja el modo en que están implementadas las partes específicas del sistema.

Las características que se deben definir en esta capa de abstracción son los siguientes:

- NUM_CPUS: número de procesadores que tiene la arquitectura concreta sobre la que se va a ejecutar el S.O.
- SHARED_MEMORY_BEGIN: dirección de la zona de memoria que comparten todos los procesadores.

- SHARED_MEMORY_SIZE: tamaño de la zona de memoria compartida.
- PRIVATE_MEMORY_BEGIN: inicio de las zonas de memoria privadas de cada procesador.
- PRIVATE_MEMORY_SIZE: tamaño de la zona de memoria privada.
- GET_PROCESSOR_ID(VAR): con esta identificación se describe la instrucción o conjunto de instrucciones necesarias para leer el mecanismo de identificación del procesador que utilice la arquitectura para la que se va a utilizar el S.O.
- HARDWARE_MUTEX_LOCK / HARDWARE_MUTEX_UNLOCK(NAME): con estas dos definiciones se debe indicar la instrucción o conjunto de instrucciones necesarias para tomar / ceder el control del mecanismo de sincronización hardware que implemente la arquitectura sobre la que va a correr el S.O.

A continuación se muestra un ejemplo de cómo se debería escribir el HAL para una arquitectura concreta como la mostrada en la figura 1, que consta de:

- 4 procesadores MicroBlaze.
- Una zona de memoria compartida a través del bus OPB.
- Una zona de memoria privada accesible a través del bus LMB.
- Un mecanismo de sincronización, hardware_mutex, como el presentado en trabajos anteriores [5].
- Un registro accesible a través del bus FSL con el identificador del procesador.

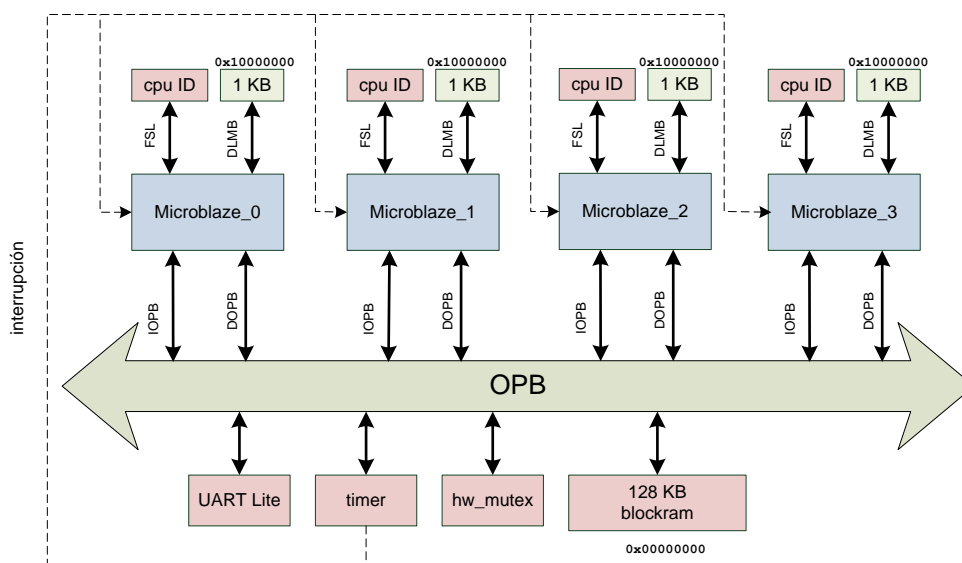


Figura 1: sistema SMP utilizando procesadores soft-core MicroBlaze

Para este sistema concreto el HAL tendrá las siguientes definiciones:

```

#define NUM_CPUS 4
#define SHARED_MEMORY_BEGIN 0x00000000
#define SHARED_MEMORY_SIZE 0x00200000 // (128 KB)
#define PRIVATE_MEMORY_BEGIN 0x10000000
#define PRIVATE_MEMORY_SIZE 0x00000400 // (1 KB)
#define GET_PROCESSOR_ID (VAR) \
    microblaze_bread_datafs1 (VAR, 0)
#define HARDWARE_MUTEX_LOCK \
    hw_mutex_lock(0x10010000)
#define HARDWARE_MUTEX_UNLOCK \
    hw_mutex_unlock(0x10010000)

```

Tabla 1: ejemplo de HAL para el sistema de la figura 1

Con esas definiciones el S.O ya sabe todo lo necesario sobre las características de la plataforma, y usará esta información y estas funciones para realizar de forma correcta las operaciones de planificación de procesos en los 4 procesadores, comunicación y sincronización entre los distintos procesos que se estén ejecutando en el sistema, y el resto de operaciones específicas de un sistema SMP.

2.3 Estructuras de datos que utiliza el S.O

El sistema operativo utiliza distintas estructuras de datos para almacenar distinta información sobre los procesos que están en el sistema, así como de los procesadores.

La estructura de datos que utiliza el S.O para albergar las características de los procesos que se ejecutan en el sistema se llama *process_struct* y contiene la siguiente información acerca de cada proceso:

- *process_context*: almacena el estado del procesador para cada proceso (el contenido de los 32 registros de propósito general, más el registro de estado del procesador MSR)
- *state*: estado del proceso (*new*, *run*, *ready*, *wait*, *dead*, . . .).
- *pid*: identificador del proceso.
- *priority*: prioridad del proceso. Sólo va a ser útil en el caso de que se haya escogido un esquema de planificación basado en prioridades.
- *is_allocated*: indica si este *process_struct* está asignado a un proceso o está libre para ser utilizado por un proceso de nueva creación.
- *block_q*: si el proceso está bloqueado, esta variable indica en que cola está bloqueado.

El sistema operativo va a mantener una tabla de procesos llamada *ptable[]*, que consiste en un array de *process_structs* con tantas entradas como procesos simultáneos pueda soportar el S.O, lo cual se puede configurar a través de un parámetro del S.O llamado *MAX_PROCS*.

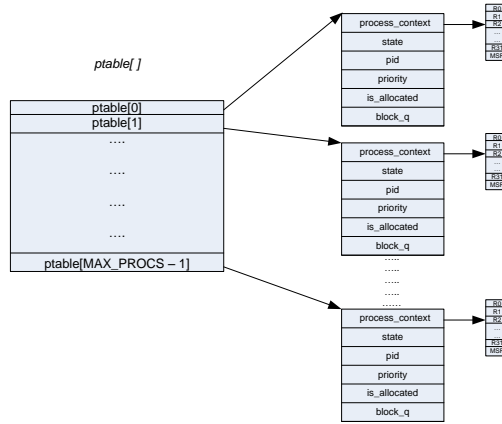


Figura 2: tabla de procesos del sistema

Además de estas dos estructuras de datos para almacenar el estado de los procesos, se hace necesario saber qué proceso se está ejecutando en cada procesador. Para ello se utiliza un array de punteros a `process_struct`, llamado `current_process[]`. El array tendrá tantas entradas como procesadores haya en el sistema, y cada elemento apuntará al `process_struct` del proceso que esté ejecutando ese procesador. Así, `current_process[0]` apuntará en todo momento al proceso que esté ejecutando el procesador número 0, `current_process[1]` apuntará al proceso que esté ejecutando el procesador número 1, etc.

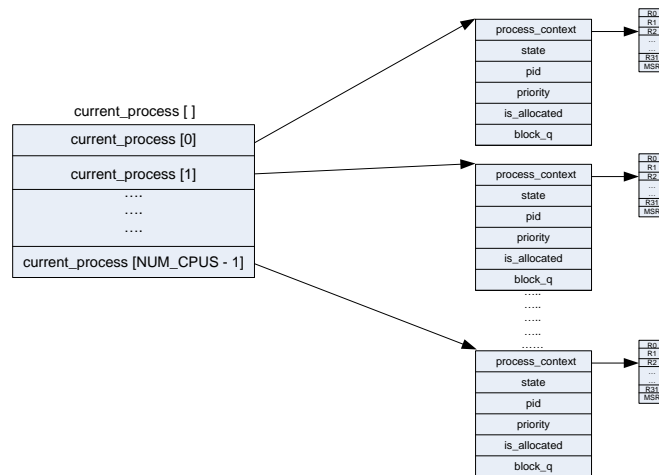


Figura 3: punteros al proceso activo en cada procesador

2.4 Inicialización del sistema

Durante la inicialización del sistema, el sistema operativo tiene que configurar varios elementos. Por una parte tiene que configurar los dispositivos hardware que utiliza para realizar ciertas tareas, como son el controlador de interrupciones y el *timer* del sistema. Por otra parte debe inicializar una serie de estructuras de datos y colas para dar soporte a los distintos módulos del S.O: estructuras de datos para los procesos, la cola del *scheduler* y otras estructuras y colas para los módulos opcionales que tiene el S.O (*threads*, semáforos, colas de mensajes, etc).

En concreto la secuencia de acciones que realiza el sistema al iniciarse son:

- Configurar el *timer*.
- Inicializar las estructuras de datos para los procesos
- Inicializar la cola del *scheduler*
- Inicializar el módulo *pthread*, para dar soporte a procesos ligeros con interfaz POSIX
- Crear una tarea *dummy*, llamada *idle_task* que no hace nada útil y que se utilizará para mantener ocupados a los procesadores si no hay procesos listos para ejecutar.
- Crear los hilos estáticos que haya definido el usuario, y ponerlos en la cola del *scheduler*.
- Activar las interrupciones
- Ejecutar el proceso *idle_task* en cada procesador

Una vez realizado este proceso de inicialización, cuando llegue la primera interrupción del *timer* comenzarán a planificarse los distintos procesos que están listos para ejecutar en los procesadores disponibles en el sistema.

El código de inicialización del sistema operativo no debe ser ejecutado íntegramente por todos los procesadores, si no que sólo uno de ellos debe realizar ciertas tareas. El procesador encargado de las principales tareas de inicialización recibe el nombre de *boot processor*, y será el que tenga como identificador de procesador el número '0'.

2.5 Interrupción del timer y planificación de procesos

El funcionamiento de la rutina de atención a la interrupción es similar al tratamiento de interrupciones clásico: deshabilitar las interrupciones, salvar el estado del procesador, cambiar a la pila del *kernel*, atender la interrupción, restaurar el estado del procesador y habilitar las interrupciones.

El hecho de tener varios procesadores hace que se complique un poco el diseño, ya que todos son interrumpidos simultáneamente, y por tanto no todos deben atender todas las interrupciones. La única interrupción que debe ser atendida por todos los procesadores es la interrupción del *timer*, que es la que se utiliza para lanzar el *scheduler* y planificar nuevos procesos que estuvieran en espera. En esta primera versión del S.O no se da soporte a otras fuentes de interrupción además del *timer*, y se deja ese aspecto para una versión futura.

Otro problema a la hora de tratar las interrupciones surge a la hora de cambiar a la pila del *kernel*. Esta pila se utiliza cuando se ejecutan funciones del *kernel*, como llamadas al sistema o rutinas de atención a la interrupción. El hecho de tener varios procesadores hace necesario que exista una pila por procesador, ya que al atender todos los procesadores a la vez la interrupción del *timer*, o si por ejemplo dos procesadores realizan simultáneamente una llamada al sistema, si sólo existe una pila compartida por todos los procesadores se

podrían alterar los datos de forma no controlada dando lugar a situaciones que podrían llevar a un comportamiento no esperado del sistema completo.

Para tener una pila por procesador se ha optado por exigir que cada procesador tenga una región de memoria no compartida, y que todos la tengan mapeada en la misma dirección. Esta forma de resolver el problema de pilas separadas no es la única posible, pero se ha escogido como primera opción por que simplifica el diseño de las rutinas de inicialización del S.O así como de los scripts de linkado de las aplicaciones de usuario. En un futuro se pretende añadir la opción de que en lugar de tener esta región no compartida de memoria, se puedan tener en la zona de memoria principal varias pilas separadas, una para cada procesador.

La figura 4 muestra las acciones que se realizan durante la atención a la interrupción del *timer*. A continuación se detalla el funcionamiento de dichas acciones:

- Primero se salva el estado del procesador en el *process_struct* que corresponda. Para ello es necesario saber qué procesador está ejecutando la rutina de atención a la interrupción, y obtener un puntero al *process_struct* del proceso que está ejecutando dicho procesador.
- Cambiar el puntero de pila para que apunte a la pila del *kernel*.
- Llamar a la rutina del *scheduler*. Después de que finalice la rutina del *scheduler*, ya se tendrá un puntero apuntando al *process_struct* del nuevo proceso planificado.
- Restaurar el estado del procesador a partir de la información del *process_struct*.
- Retornar de la interrupción.

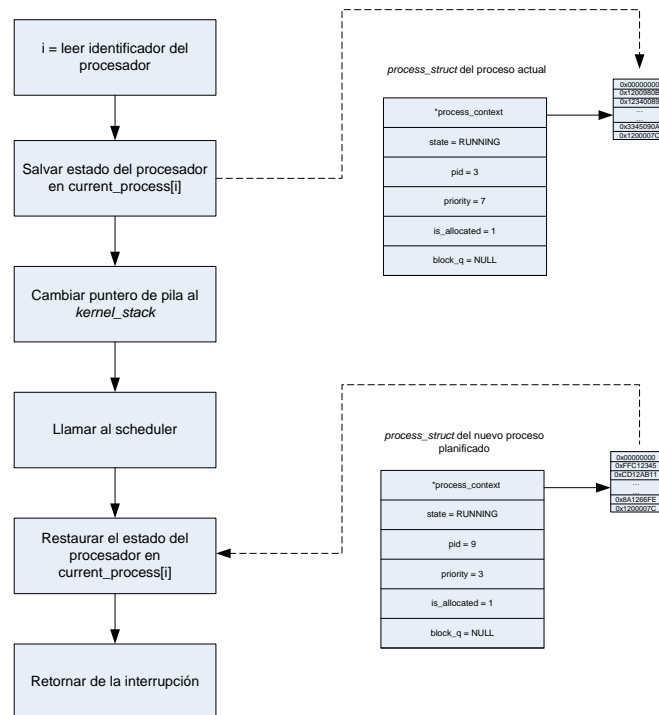


Figura 4: funcionamiento de la rutina de atención a la interrupción

El modelo de procesos que implementa el sistema operativo distingue entre seis posibles estados en los que puede estar un proceso:

- PROC_NEW: un proceso nuevo, recién creado.
- PROC_READY: un proceso listo para ser ejecutado.
- PROC_RUN: un proceso actualmente en ejecución.
- PROC_WAIT: un proceso bloqueado en algún recurso.
- PROC_DELAY: un proceso que está esperando por un *timeout*.
- PROC_TIMED_WAIT: un proceso bloqueado en un recurso, y con un *timeout* asociado.

Utilizando esta información sobre los procesos el *scheduler* se encarga de ir planificando qué procesos deben estar en estado PROC_RUN y por tanto siendo ejecutados por alguno de los procesadores del sistema siguiendo una política de planificación del tipo *round-robin*. Como el *scheduler* se activa con la interrupción del *timer* en todos los procesadores simultáneamente se debe proteger el acceso a la cola del *scheduler* para que solamente un procesador acceda a ella en un determinado instante de tiempo. Para ello se utilizará el dispositivo hardware que se haya escogido a la hora de implementar el sistema SMP y que se habrá configurado en el HAL para poder ser manejado por el sistema operativo.

La figura 5 muestra las distintas acciones que realiza la rutina del *scheduler* a la hora de planificar los procesos que se deben ejecutar.

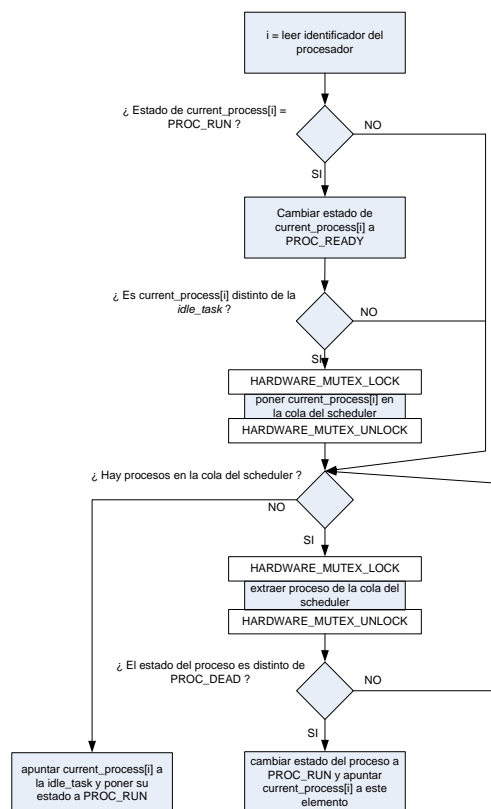


Figura 5: planificación de procesos por el *scheduler*

3 Sistema SMP basado en MicroBlaze

El sistema propuesto como ejemplo en el punto 2.2 se implementó sobre una FPGA XC2V6000-FF1152-4 de la familia Virtex 2 de Xilinx y sobre el sistema se ejecutó una aplicación multi-hilo que utiliza diversos servicios del sistema operativo implementado para comprobar su correcto funcionamiento.

La aplicación consiste en un hilo principal que crea varios *threads* idénticos que imprimen a través del puerto serie mensajes indicando el identificador del *thread* junto con el identificador del procesador que lo está ejecutando. A continuación se muestra parte del código del programa principal así como de los *threads* que se lanzan.

```
#define NUM_THREADS 8

pthread_t p_th[NUM_THREADS];
sem_t semaforo;

void main_thread(void)
{
    sem_init(&semaforo, NULL, 1);
    sem_wait(&semaforo);
    print("Lanzando threads\r\n");
    sem_post(&semaforo);
    for(i = 0; i < NUM_THREADS; i++)
        pthread_create(&p_th[i], NULL, (void*)thread_1, NULL);
    for(i = 0; i < NUM_THREADS; i++)
        pthread_join(p_th[i], NULL);
    sem_wait(&semaforo);
    print("Todos los threads finalizaron\r\n");
    sem_post(&semaforo);
}

void thread_1(void)
{
    int i, processor_id, thread_id;
    thread_id = pthread_self();
    sem_wait(&semaforo);
    GET_PROCESSOR_ID(processor_id);
    xil_printf("Al entrar: thread_id = %d\t
               processor_id = %d\r\n", thread_id, processor_id);
    sem_post(&semaforo);

    for(i = 0; i < 5000000; i++);

    sem_wait(&semaforo);
    GET_PROCESSOR_ID(processor_id);
    xil_printf("Al salir: thread_id = %d\t
               processor_id = %d\r\n", thread_id, processor_id);
    sem_post(&semaforo);

    pthread_exit(NULL);
}
```

Tabla 2: aplicación multi-hilo para testear el sistema operativo

La figura 6 muestra el resultado de la ejecución del programa de prueba, donde se puede observar como los distintos *threads* son ejecutados por los 4 procesadores presentes en el sistema SMP.

```

Lanzando threads
Al entrar: thread_id = 1      processor_id = 0
Al entrar: thread_id = 0     processor_id = 1
Al entrar: thread_id = 2     processor_id = 2
Al entrar: thread_id = 4     processor_id = 0
Al entrar: thread_id = 5     processor_id = 0
Al entrar: thread_id = 7     processor_id = 1
Al entrar: thread_id = 6     processor_id = 3
Al entrar: thread_id = 8     processor_id = 0
Al salir: thread_id = 1      processor_id = 0
Al salir: thread_id = 0     processor_id = 1
Al salir: thread_id = 2     processor_id = 2
Al salir: thread_id = 4     processor_id = 0
Al salir: thread_id = 5     processor_id = 0
Al salir: thread_id = 7     processor_id = 1
Al salir: thread_id = 6     processor_id = 3
Al salir: thread_id = 8     processor_id = 0
Todos los threads finalizaron

```

Figura 6: salida de la aplicación de prueba

4 Conclusiones y trabajo futuro

Se ha presentado un sistema operativo que permite ejecutar aplicaciones multi-hilo en sistemas SMP y se ha probado sobre un sistema con 4 procesadores MicroBlaze. El sistema operativo que se ha desarrollado está pensado para ser fácilmente portable a una arquitectura SMP concreta basada en MicroBlaze mediante el uso de una capa de abstracción de hardware que implementa la parte del sistema operativo dependiente del hardware.

Aunque el sistema operativo se ha desarrollado para funcionar con MicroBlaze, se está trabajando en portarlo para funcionar con los dos procesadores PowerPC que incluyen algunas FPGAs de las familias Virtex 2 Pro y Virtex 4 FX de Xilinx. Otra línea en la que se está trabajando actualmente es en la evaluación de rendimiento de diferentes arquitecturas SMP basadas en MicroBlaze utilizando el sistema operativo que se ha presentado en este artículo para ejecutar diferentes aplicaciones paralelizables.

Referencias

1. A.Hung, W.Bishop, A.Kunnings, "Symmetric Multiprocessing on Programmable Chips Made Easy", Proceedings of the Design Automation and Test in Europe Conference and Exhibition, 2005
2. Altera, "Nios II Processor Reference Handbook", 2005, available at <http://www.altera.com>
3. D. Lampret, "OpenRISC 1200 IP Core Specification", 2001, available at www.opencores.org
4. Gaisler Research, "Leon2 Processor User's Manual", 2005, available at <http://www.gaisler.com>
5. Huerta P., Castillo J., Martinez J.I., Pedraza C., "Exploring FPGA capabilities for building symmetric multiprocessor systems", Proceedings of the 3rd Southern Conf. on Programmable Logic, 2007
6. S. Dharmasnam, "Multiprocessing with real-time operating systems", Embedded.com eMagazine, 2003
7. Xilinx, "MicroBlaze Processor Reference Guide", 2005, available at <http://www.xilinx.com>
8. Xilinx, "OS and Libraries Document Collection", 2005, available at <http://www.xilinx.com>