DISEÑO E IMPLEMENTACIÓN DE UN MICROPROCESADOR DE PROPÓSITO ESPECÍFICO

Design and implementation of a special-purpose microprocessor

RESUMEN
Este documento pretende compartir la experiencia del diseño de un procesador y su implementación en una FPGA (Spartan 3E – 100E), mostrando las características generales que debe tener el procesador para que sea lo suficientemente flexible, y que herramientas adicionales se deben crear para facilitar sus pruebas e implementación.

PALABRAS CLAVES: Procesador, FPGA, VHDL, Ensamblador, Java

ABSTRACT
This paper aims to share the experience of designing a microprocessor and its implementation on a FPGA device (Spartan 3E - 100E). Showing the general features that the processor must have in order to be flexible enough , and the general tools that must be created to make easier the test and implementation stages.

KEYWORDS: *Processor, FPGA, VHDL, Assembler, Java*

JUAN DAVID HINCAPIE ZEA
Ingeniero Electrónico
Docente Facultad de Ingenierías
Universidad Tecnológica de Pereira
judaz@sirius.utp.edu.co

JOSE ALFREDO JARAMILLO
Ingeniero Electrónico
Docente Facultad de Ingenierías
Universidad Tecnológica de Pereira
jj@sirius.utp.edu.co

SEBASTIÁN GÓMEZ
Ingeniero de Sistemas.
Estudiante
Universidad Tecnológica de Pereira
sgomez_gonzalez@yahoo.es

1. INTRODUCCIÓN

Cuando nos enfrentamos con un proyecto que requiere el desarrollo de un sistema embebido, tenemos que tomar la decisión de que tecnología usar. El problema es que de esta decisión puede depender el éxito o fracaso del proyecto, ya que cambiar la tecnología implica realizar de nuevo gran parte del trabajo, por la programación a bajo nivel (lenguaje ensamblador y/ó código maquina)[1]. En el mercado nos encontramos con muchas alternativas que varían enormemente en capacidad, tamaño físico, consumo de corriente y precio; Todas estas variables son sumamente importantes al escoger el procesador indicado para nuestra necesidad particular. En muchas de las aplicaciones sabemos cuales son los requerimientos y podemos asumir que no van a cambiar mucho en el tiempo, por tanto podemos elegir fácilmente entre las alternativas que nos ofrece el mercado. Pero es en los casos donde los requerimientos pueden variar significativamente en el tiempo, en el que cobra importancia trabajar con tecnologías reconfigurables como las FPGA's.

En la comunidad científica se muestran numerosos ejemplos de diseños de procesadores usando FPGA's y lenguajes de descripción de Hardware. Morimoto et al [5] plantean un lenguaje de descripción de Hardware de rápido desarrollo llamado AIDL. En cuanto a desarrollo de procesadores R.D Witting[6] plantea como se puede mejorar el desempeño de un sistema de cómputo mezclando procesadores duros con lógica reconfigurable, [7] plantea como utilizar una FPGA para hacer un coprocesador matemático para una arquitectura MIPS y [8] muestra como incrementar la velocidad de algoritmos de encriptación con el uso de lógica reconfigurable.

2. PLANTEMANIENTO DEL PROBLEMA

Como ya se dijo en la introducción, hay proyectos en los que no se conoce de antemano si los requerimientos pueden cambiar con el tiempo, como por ejemplo diseñar robots que el usuario pueda programar para alguna labor, y no sabemos para qué labores en particular.
En estos casos, es recomendable trabajar con tecnologías reconfigurables, en estas tecnologías también se cuenta con una gran variedad de precios, capacidades, consumos de corriente y tamaños físicos. La principal diferencia entre los dispositivos reconfigurables y los microprocesadores convencionales es que si solo se utiliza un pequeño porcentaje del dispositivo reconfigurable, se puede agregar hardware adicional si es necesario. En este caso particular se pretende crear un procesador para manejar un robot programable por el usuario, esta implementación se debe hacer sobre una tarjeta que ocupa poco espacio físico, por lo que se escogió una tarjeta Basys la cual tiene una FPGA Spartan 3E-100E, que tiene una capacidad suficientemente para la solución del proyecto.

Así como existe el Software libre, también hay sitios de Hardware libre de donde se pueden descargar descripciones en VHDL de procesadores (www.opencores.org), sin embargo existen algunas

Fecha de Recepción: 9 de Septiembre de 2010
Fecha de Aceptación: 24 de Abril de 2011

ventajas en crear una arquitectura propia[2]; la primera es que muchos de los diseños de Hardware que se pueden descargar son de gran tamaño, por tanto ocupan un mayor número de compuertas lógicas, utilizando muchos recursos del dispositivo. Esto hace necesario usar una FPGA de más alta capacidad, y por tanto más costosa; Otra razón es que no es fácil entender su funcionamiento ni su diseño interno, dificultando hacer cambios más adelante. Por último, si diseñamos una arquitectura pensando en nuestras necesidades, seguro obtendremos un mejor resultado, ya que podremos realizar un hardware mucho mas eficiente.

3. EL PROCESADOR

Hablemos primero un poco de la arquitectura, recordemos que la idea es diseñar un procesador suficientemente flexible pero eficiente en cuanto al consumo de compuertas lógicas. Para esto trabajamos con un conjunto de instrucciones reducido tipo RISC (Reduced Instruction Set Computer)[2], la pregunta ahora es de cuantos bits debe ser la arquitectura.

Con una arquitectura de 8 bits, solo se pueden direccionar 256 (2^8) Bytes de memoria RAM, además hay que descontar las direcciones que se usan en la Entrada/Salida. Suponiendo que se usaran 16 de estas direcciones para la E/S y que se usaran 48 direcciones mas en variables globales a la aplicación, nos quedarían 192 Bytes para la pila de llamados. Y suponiendo que cada función necesite en promedio 32 Bytes (Para almacenar la dirección de retorno, parámetros y variables locales) solo se podrían hacer 192/32 = 6 llamados anidados a funciones. Esta cantidad de llamados posibles es inaceptable para la mayoría de las aplicaciones, especialmente si se deben implementar algoritmos recursivos. Una arquitectura de 32 bits en cambio, aunque es muy apropiada para desarrollar Software sin muchas restricciones, utilizaría muchos recursos físicos de la FPGA, y por tanto tampoco es apropiada. Se eligió implementar una arquitectura de 16 bits, en esta se pueden direccionar 65536 (2^{16}) posiciones de memoria RAM, que nos da mucha más libertad y nos permite usar toda la RAM disponible en la Spartan 3E.

Para mantener las instrucciones lo más simples posible, estas solo recibiran un solo operando y se toma como el otro operando al acumulador, el resultado también se guarda en el acumulador. La siguiente expresión explica la lógica de este tipo de instrucciones:

$$[3.1]\ \textit{Acumulador} = \textit{Acumulador}\ \ \textit{Operador}\ \ \textit{Operando}$$

Los "operadores" soportados por el hardware son la suma, AND, OR, XOR, desplazamiento de bits a la izquierda y derecha, comparación y resta.

Los "Operandos" pueden ser registros del procesador o constantes numéricas (que de aquí en adelante llamaremos inmediatos).

Para el control de flujo de los programas tenemos instrucciones de salto, en estas tenemos: Salto condicional, salto incondicional, llamado a subrutina y retorno de subrutina.

Además están las instrucciones que permiten mover datos desde y hacia el acumulador de la memoria o los registros.

Registro	Uso
0 (cero)	Contiene el valor 0.
1 (PP)	Puntero de la pila
2,3 (v0, v1)	Valores retornados por funciones
4,5,6,7	Parámetros de funciones (a0-a3)
8 → 15	Temporales (t0 - t7)
16 → 23	Guardados (s0 - s7)
24 → 29	Temporales del compilador (c0 - c5)
30 (RI)	Dirección de retorno de interrupción
31 (DR)	Dirección de retorno

Tabla 1. Distribución de los registros en el Archivo de Registros

Se tienen 32 registros, cada uno de 16 bits. Se distribuyen de manera específica (tabla 1).

Con las pruebas realizadas, podríamos concluir que aunque este conjunto de instrucciones es muy reducido, podemos realizar operaciones complejas como las de otros procesadores del mercado para sistemas embebidos.

Para lo anterior se hace necesario las siguientes unidades de hardware (Figura 1) dentro del procesador :

- **Memoria de programa**: La memoria donde se almacena el programa a ejecutar.
- **Archivo de registros**: Memoria donde se guardan los registros del procesador.
- **Memoria de datos**: Memoria RAM, se puede implementar en VHDL creando un arreglo.
- **ALU**: Unidad aritmética-lógica, aquí se ejecutan las operaciones matemáticas y lógicas, se puede implementar calculando todas las operaciones para las 2 entradas y multiplexarlas escogiendo cual poner en la salida dependiendo de la operación deseada (que es otra entrada)
- **Unidad de control**: Por todo el procesador, hay multiplexores para escoger el camino de los datos, la tarea de la unidad de control es escoger que señales generarle a estos multiplexores dependiendo de la instrucción a ejecutar.

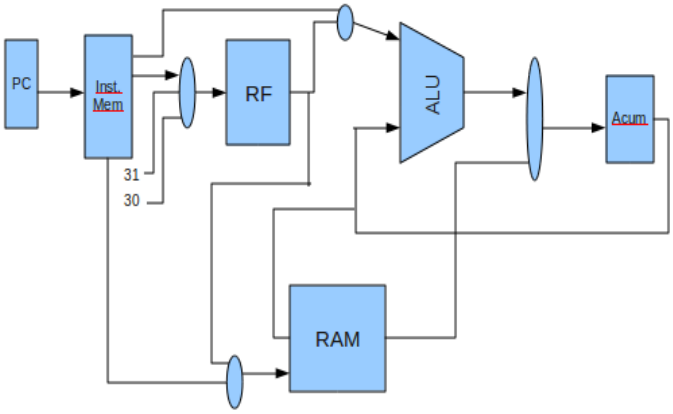

Figura 1. Diagrama de bloques del procesador.

Además de tener el hardware que nos permita correr estas instrucciones, es necesario agregar algunas características que ayuden a la multitarea, como las interrupciones. Las interrupciones son muy útiles ya que la la E/S (Entrada/Salida) es generalmente más lenta que el procesador, y lo ideal es que el procesador se dedique a realizar otras tareas o ejecutar otras instrucciones mientras la E/S termina. Una vez la E/S a terminado, el controlador del dispositivo de E/S le genera una interrupción al procesador[4].
Para este diseño tenemos un registro que indica si las interrupciones están habilitadas o no. Se creó una entrada de interrupción síncrona al procesador, esta entrada se conecta a la unidad de control. Si esta entrada se activa cuando estaban habilitadas las interrupciones, se espera al siguiente ciclo de reloj y se asigna al contador de programa (registro que indica cual es la siguiente instrucción a ejecutar) un valor fijo llamado dirección de rutina de interrupción (Posición 2 de la memoria de programa). En este mismo ciclo de reloj se guarda el contador de programa actual (PC) en el registro 30 del archivo de registros. Así el procesador puede retornar lo que estaba haciendo antes de la interrupción, una vez la interrupción ha sido procesada.
Otra tarea importante en el diseño del procesador es escoger como codificar las instrucciones. Siguiendo la filosofía de las arquitecturas RISC, las instrucciones se deben codificar de tal forma que todas tengan el mismo tamaño (Todas deben ocupar la misma cantidad de bits).

4. HERRAMIENTAS

Para el desarrollo de este proyecto, se necesitan unas herramientas básicas, entre estas se hace obligatorio tener un sintetizador para la descripción del hardware que se realice en VHDL. Para esta implementación se uso el ISE de Xilinx que se puede descargar gratuitamente[3].
Los procesos de ingeniería de Software establecen además que una de las etapas mas importantes del proyecto es la de simulación y pruebas. Por esta razón se hace necesario escribir algún programa que se pueda ejecutar en el procesador diseñado. Dicho programa debe estar escrito en lenguaje de máquina. Dado que escribir un programa en lenguaje de máquina es una labor compleja para un programador, se diseño un lenguaje ensamblador, que aunque no brinda las facilidades de un lenguaje de alto nivel, es mas amigable con el programador que el lenguaje de máquina. Para hacer la traducción de lenguaje ensamblador a lenguaje de máquina, se utiliza usualmente un Software llamado ensamblador. Dado que se está creando una arquitectura nueva, se debe crear un software ensamblador para esta arquitectura en particular.

4.1 Ensamblador

Esta es una de las etapas que usualmente queremos automatizar, en este caso se diseñó e implementó un programa en Java. Este programa toma de un archivo de texto las instrucciones en el ensamblador que se definieron para el procesador y las traduce a lenguaje de máquina. Recordemos que las instrucciones se pueden dividir en 3 tipos[1][2][4], las de registros (Tipo R: Instrucciones que reciben un registro como parámetro), las inmediatas (Tipo I: las que reciben una constante como parámetro) y las de saltos (Tipo J).
Se creó una base de datos en MySql con una tabla de instrucciones con los campos “OPCODE” (Código de operación de la instrucción), “FUNCTION” (Código de apoyo en caso de ser tipo R) y “INST” (Instrucción en lenguaje Ensamblador).
En compiladores o ensambladores, usualmente se hace una lectura secuencial del código fuente. A una única lectura secuencial de inicio a fin se le denomina una pasada. En ocasiones los lenguajes de programación se definen de tal forma que en una única pasada no se puede obtener toda la información requerida para hacer la compilación. Por ejemplo, si el lenguaje permite llamar a una función que se declara mas abajo de donde se está llamando. En este caso el lenguaje no puede saber en la primera pasada donde se encuentra la función llamada o si siquiera está declarada, y por tanto, se requiere una segunda pasada por el código resolviendo los nombres no resueltos en la primera pasada.
En el caso del lenguaje ensamblador, las instrucciones tipo R y tipo I se pueden convertir a lenguaje de máquina en una sola pasada. Las instrucciones tipo J en cambio, necesitan una segunda pasada, ya que en ensamblador se hacen saltos a etiquetas, pero en lenguaje de máquina se debe poner la dirección física del salto. En este caso se puede hacer un llamado a una función que se define mas abajo o saltar a una etiqueta definida mas abajo. Entonces de deben hacer 2 pasadas. Una pasada ensamblando las instrucciones tipo R e I y guardando en una tabla HASH las direcciones físicas que le corresponden a las etiquetas lógicas y una segunda pasada ensamblando las instrucciones tipo J con estas direcciones físicas.

4.2 Simulador

Aunque tener el ensamblador es de mucha ayuda, ya que no se programa en lenguaje de máquina sino con

instrucciones nemónicas fáciles de entender y recordar, aun se es propenso a cometer muchos errores lógicos. Por ejemplo que en un ciclo el programador olvide incrementar la variable o apilar un valor importante.
Este tipo de errores son difíciles de detectar sin las herramientas adecuadas, especialmente porque el procesador está en etapa de pruebas, y cuando algo no funcione correctamente es difícil determinar si la falla está en el procesador o en el programa escrito en ensamblador. Por nuestra experiencia podríamos decir que 1 de cada 10 errores esta en el hardware, los otros 9 son errores lógicos del lenguaje ensamblador.
Por esta razón es importante contar con herramientas que permitan depurar el programa antes de montarlo en la FPGA. Para hacer esto se escribió un programa en Java, que simula la arquitectura, con la misma cantidad memoria y registros que el procesador diseñado. La idea de este programa no es que simule el hardware de la arquitectura sino la lógica de las instrucciones.
La implementación del simulador, mejoró los tiempos de desarrollo logrando que la mayoría del software funcionara al primer intento al ser ejecutado en la FPGA.
Lo más complicado de simular son las interrupciones y la instrucción "dormir", ya que el hardware real se queda esperando a que llegue la interrupción para despertarse, pero en el simulador necesitamos una condición de salida para saber cuándo se ha terminado la ejecución del programa, así que la instrucción "dormir" interrumpe la ejecución de la simulación y al enviar una interrupción no se puede reanudar.

5. CONCLUSIONES

Este procesador ocupó un 38% de la lógica de la FPGA mientras que algunos de los diseños libres disponibles en Internet ocupaban el 120% del mismo dispositivo, así que se puede lograr reducir mucho el consumo de la lógica disponible en la FPGA eliminando características que son poco necesarias para el problema que se pretende resolver. Algunas de estas características se pueden hacer necesarias, pero se puede evaluar si se pueden implementar por software (Por ejemplo, este Hardware no soporta multiplicación y división, pero se implementaron rutinas que pueden realizan estas operaciones en 13 µs).
Sin embargo también hay algunas desventajas al crear una arquitectura propia, la que mas salta a la vista es la falta de herramientas preexistentes. Si el tiempo de que se dispone para el desarrollo es poco, resulta poco viable desarrollar todas las herramientas mencionadas a lo largo de este artículo. Si bien un ensamblador y un simulador ayudan mucho, hace falta también un compilador de un lenguaje de alto nivel como C o C++, incluso cuando se trabajan con sistemas embebidos.

6. ANEXO (CONJUNTO DE INSTRUCCIONES)

Conjunto de instrucciones que soporta el procesador:

SUMA	Suma el contenido de un registro al acumulador.
Sintaxis:	SUMA R
Operación:	A = A + REG[R]
Y	Operación AND bit a bit de un registro al acumulador.
Sintaxis:	Y R
Operación:	A = A & REG[R]
O	Operación OR bit a bit de un registro al acumulador.
Sintaxis:	Y R
Operación:	A = A | REG[R]
XOR	Operación XOR bit a bit de un registro al acumulador.
Sintaxis:	XOR R
Operación:	A = A ^ REG[R]
ROTI	Rota al acumulador N bits a la izquierda, siendo N los 3 bits menos significativos del registro dado.
Sintaxis:	ROTI R
Operación:	A = A << (REG[R] & 0xF)
ROTD	Rota al acumulador N bits a la derecha, siendo N los 3 bits menos significativos del registro dado.
Sintaxis:	ROTD R
Operación:	A = A >> (REG[R] & 0xF)
RESTA	Resta el contenido de un registro al acumulador.
Sintaxis:	RESTA R
Operación:	A = A - REG[R]
UMQ	(Uno Menor Que) Pone al acumulador en 1 si es menor que el registro.
Sintaxis:	UMQ R
Operación:	A = (A < REG[R]) ? 1 : 0
MOVRA	Mover un registro al acumulador.
Sintaxis:	MOVRA R
Operación:	A = REG[R]
MOVIA	Mover un inmediato al acumulador.
Sintaxis:	MOVIA I
Operación:	A = I
MOVAR	Mover el acumulador a un registro.
Sintaxis:	MOVAR R
Operación:	REG[R] = A

GUARDA	Guarda el acumulador en memoria.
Sintaxis: Operación:	GUARDA M MEM[M] = A
CARGA	Carga una posición de memoria en el acumulador
Sintaxis: Operación:	CARGA M A = MEM[M]
CP	Carga una posición de memoria en el acumulador, solo que lee esta dirección de un registro que actúa como puntero.
Sintaxis: Operación:	CP R A = MEM[REG[R]]
GP	Guarda el acumulador en una posición de memoria, solo que lee esta dirección de un registro que actúa como puntero.
Sintaxis: Operación:	GP R MEM[REG[R]] = A
SC	Salta a la etiqueta si el acumulador es cero
Sintaxis: Operación:	SC Etiqueta PC = (A==0) ? Etiqueta : PC+1;
SNC	Salta a la etiqueta si el acumulador no es cero
Sintaxis: Operación:	SNC Etiqueta PC = (A!=0) ? Etiqueta : PC+1;
IR	Va a la etiqueta
Sintaxis: Operación:	IR Etiqueta PC = Etiqueta
CALL	Se llama al procedimiento y se guarda la dirección de retorno.
Sintaxis: Operación:	CALL Etiqueta REG[31]=PC+1; PC = Etiqueta;
RIR	Salto a la dirección almacenada en el registro, si el registro es el 31 equivale a un retorno de la función.
Sintaxis: Operación:	RIR R PC = REG[R];
RIRHI	Salto a la dirección almacenada en el registro, si el registro es el 31 equivale a un retorno de la función. Además habilita interrupciones.
Sintaxis: Operación:	RIRH R PC = REG[R]; Interrupciones=1;
DI	Deshabilita interrupciones.
Sintaxis: Operación:	DI Interrupciones=0;
HI	Habilita interrupciones.
Sintaxis: Operación:	DI Interrupciones=1;
DORMIR	Pausa el contador de programa hasta que se reciba una interrupción.
Sintaxis: Operación:	DORMIR Deshabilita cambios en el PC hasta la siguiente interrupción;

Tabla 2. Conjunto de instrucciones del procesador diseñado.

Las primeras 7 instrucciones tienen su versión en inmediatos.

7. BIBLIOGRAFÍA

[1]. William H. Murray, Chris H. Pappas. "80386/80286 Programacion en lenguaje ensamblador", 1ª Edicion, Editorial McGraw-Hill, 1986.

[2]. David A. Patterson, Jhon L. Hennessy. "Computer Organization Design", 3rd Edition, Editorial Kauffmann, 2005.

[3]. Thomas L. Floyd. "Digital Fundamentals With VHDL", 1st Edition, Editorial Prentice Hall, 2003.

[4]. Edison Duque. "Curso avanzado de microcontroladores PIC", 1er Edición, Editorial CEKIT, 1998.

[5]. Takayuki Morimoto et al. "Advanced processor design using Hardware description Language AIDL". *Design Automation Conference 1997*. Proceedings of the ASP-DAC'97. Asia and South Pacific. P 387-390.

[6]. Wittig, R.D. and Chow, P. "FPGAs for Custom Computing Machines". *Proceedings IEEE Symposium*. P 126-135, 1996.

[7]. Hauser, J.R. and Wawrzynek, J. "FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium". 1997. *IEEE*. P 12-21.

[8]. Hodjat, A. and Verbauwhede, I. "A 21.54 Gbits/s fully pipelined AES processor on FPGA". 2004. *IEEE Computer Society*.