ARQUITECTURA RISC DE ANCHO DE PALABRA DE DATOS PARAMETRIZABLE PARA IMPLEMENTACIÓN SOBRE TECNOLOGÍA FPGA

Miguel A. Sagreras , Julián U. da Silva Gillig, Nicolás Alvarez, Alberto Dams

Laboratorio de Sistemas Digitales, Departamento de Electrónica,
Facultad de Ingeniería, Universidad de Buenos Aires
Av. Paseo Colón 850, (1063) Ciudad de Buenos Aires, Argentina

msagre@fi.uba.ar, julian.dasilva@xior.org, nalvare@fi.uba.ar, adams@fi.uba.ar

RESUMEN

En este trabajo se propone la arquitectura básica de un microprocesador de tipo RISC, de ancho de palabra de datos parametrizable, diseñado para ser implementado sobre tecnología FPGA (Field Programmable Gate Array). El mismo está orientado especialmente a aplicaciones en sistemas embebidos, por lo que se pretende optimizar el uso de memoria de programa, manteniendo además un buen desempeño temporal de ejecución. Se hace hincapié en la reducción de dependencias entre instrucciones y otros aspectos que facilitan las optimizaciones de compilación y la generación de código eficiente temporal y espacialmente. Otro objetivo importante del diseño es reducir su complejidad, sobre todo teniendo en cuenta el tipo de tecnología de implementación a utilizar.
Keywords: Procesadores sobre FPGA, soft cores, RISC, Arquitectura de Computadoras, Sistemas Embebidos.

1. INTRODUCCIÓN

Las condiciones de diseño están dadas principalmente por los siguientes factores:

- El tipo de aplicaciones a las que está orientado primariamente (sistemas embebidos, aplicaciones de control, computación portátil de baja potencia).
- La tecnología de implementación (FPGA –*Field Programmable Gate Array*-).
- La información estadística de uso de instrucciones y desempeño en procesadores existentes [1].

A continuación se mencionan las principales condiciones de diseño de este procesador particular:

1) Instrucciones de largo fijo de 16 bits.
2) Tamaño de palabra de datos parametrizable (16-32 bits).
3) 16 registros (como mínimo) de uso general ortogonales, más el contador de programa.
4) Conjunto de instrucciones reducido (libre de redundancias, en lo posible) y donde se busca una alta simetría y ortogonalidad a nivel instrucción (además de la ya mencionada a nivel registro).
5) Tamaño fijo del campo de bits que identifica a cada instrucción (*opcode*), para facilitar la decodificación.
6) Arquitectura *load/store* estricta (taxonomía *<0, x>* según [1, pp.70-73], donde *0* es la cantidad de operandos en memoria y *x* la cantidad de operandos en registros internos).
7) Carga inmediata de constantes de 8 bits.

2. ASPECTOS BÁSICOS DE LA ARQUITECTURA

Los formatos de instrucción, el sistema de registros y los modos de direccionamiento son los primeros elementos a definir, por la fuerte relación existente entre ellos.

2.1. Registros

El procesador cuenta con 16 registros visibles de uso general (R0-R15). No se mencionarán en este trabajo los registros ocultos que puedan aparecer en la implementación, como los que almacenan valores temporales entre las etapas de *pipeline*. No hay restricciones en cuanto al uso de los registros generales. Esto significa que todos los modos de direccionamiento y todas las operaciones que involucran registros pueden utilizarlos de forma indistinta e independiente. La única salvedad es el uso de R0 para las direcciones de retorno de las instrucciones JAL y JRAL (*Jump And Link* y *Jump Register And Link*). El contador de programa (PC) está separado de estos registros. No hay registro de estado (ver [2], [4] y [5], por ejemplo), ya que esto genera dependencias entre instrucciones. En su lugar, cada registro posee 3 bits asociados: T (*Test*), C (*Carry*) y V (*Overflow*). En la sección 3 (saltos condicionales) se profundiza este punto. La figura 1 muestra la estructura de cada registro (con el procesador configurado en 16 bits).

Bits	15	14-0			
Designación	(N)	-	T	C	V

Fig. 1. Registros de uso general y sus bits de estado.

El bit más significativo es considerado como el flag N (*Negative*). En la figura 2 se ve un esquema de todos los registros visibles.

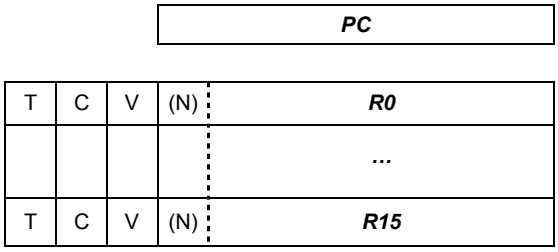

Fig. 2. Registros visibles del procesador.

2.2. Formatos de instrucción.

La tabla 1 muestra los 3 formatos de instrucción: J, A, M, S y R. El código de operación será de 4 bits, obteniéndose un espacio de 16 instrucciones. Su posición es fija, así como la de los demás campos. Esto simplifica la lógica de decodificación y la estructura interna del procesador.

Tipo	Campos (Cantidad de bits)			
J	Op (4)	Imm (12)		
A	Op (4)	Rd (4)	Imm (8)	
M	Op (4)	Rd (4)	Rs (4)	Imm (4)
S	Op (4)	Rd (4)	Imm (4)	Func (4)
R	Op (4)	Rd (4)	Rs (4)	Func (4)

Tabla 1. Formato de las instrucciones.

Todas las operaciones aritmético-lógicas entre registros son del tipo R. Por lo tanto, la cantidad de operandos es 2. Dichos operandos están representados en campos de 4 bits, para direccionar a los 16 registros. El destino de la operación está implícito, siendo uno de los operandos (Rd en la figura). El último campo (4 bits) en este tipo de instrucciones determina la operación. Las condiciones de diseño 1, 3 y 5 quedan satisfechas con esta elección de formatos. La carga de constantes inmediatas de 8 bits (condición 7) se hará con una instrucción del tipo A, lo mismo que los saltos condicionales (*branches*) y la instrucción "*jump register and link*". El tipo J permitirá hacer saltos incondicionales relativos al PC con un desplazamiento de 12 bits (código independiente de la posición absoluta en memoria). El formato de instrucción M es para las operaciones de acceso a memoria. Finalmente las instrucciones de desplazamiento inmediato (subconjunto de las instrucciones de ALU) pertenecen al tipo S.

2.3. Modos de direccionamiento

Los modos que posee son *inmediato*, *directo* e *indirecto con desplazamiento* (instrucciones LW/SW, LH/SH y SB). No existen más operaciones sobre la memoria que la carga o el almacenamiento de datos (condición 6). Los movimientos entre registros son realizados por la instrucción aritmética MOV (tipo R). Todos los registros sirven como punteros en el direccionamiento indirecto (condiciones 3 y 4). Las operaciones relacionadas con la unidad aritmético lógica (ALU) trabajan con los modos inmediato y directo (si bien en el modo inmediato se hacen algunas salvedades, se guarda una alta simetría, como se podrá ver en la sección 5).

3. EL SISTEMA DE SALTOS CONDICIONALES

Es conocido el problema de la aparición de dependencias entre instrucciones ([1, pp.80-85]) cuando el procesador cuenta con un registro de estado. Esto limita al compilador en cuanto al orden en que genera las instrucciones, lo que dificulta la optimización del código (lo cual ha sido priorizado en este diseño). En procesadores como el DLX (ver [1, p.104]), el problema se resuelve con instrucciones que colocan en 0 un registro dependiendo de la condición a evaluar (*set conditionals*), para que luego los saltos condicionales decidan en base a si dicho registro es igual o distinto de 0 (instrucciones del tipo BRZ y BRNZ). Esto puede no tener un impacto negativo apreciable cuando se cuenta con 32 o más registros generales, pero en procesadores más pequeños el problema no es menor. Instrucciones de ese tipo destruyen el contenido del registro de destino, lo que acrecienta la cantidad de operaciones necesarias. Además, con respecto al método de registro de estado, esto aumenta la cantidad de instrucciones necesarias en la evaluación de la condición de salto. En nuestro caso se optó por colocar los bits de estado asociados a cada registro (figura 1). Esto permite obtener un eficiente sistema de saltos condicionales, con sólo 2 instrucciones (la escritura condicional del bit y el salto condicional consiguiente), sin perder contenidos ni forzar el orden de ejecución. El bit sobre el que se decide es siempre el T (Test). Sin embargo, esta solución acarreó una complicación: la evaluación contra el literal 0 tras las operaciones de ALU (caso muy común –ciclos tipo *for*, por ejemplo-) requería del agregado de la instrucción de escritura condicional del bit. Por esto, se definió que todas las operaciones de ALU (tanto inmediatas como directas) modificaran al bit T de la misma forma en que lo harían si existiera el código de condición Z (*Zero*). Así, en la mayoría de los casos, tras una operación de ALU directamente puede ejecutarse el salto. Este método, en conjuntos de instrucciones más ricos (aquí las limitaciones se deben a las condiciones de diseño y sobre todo al ancho de palabra de instrucción -16 bits-) puede extenderse, agregando saltos sobre los otros bits de estado o incluso agregando nuevos bits que reflejen otras condiciones. Por último, en la instrucción de salto condicional propuesta (BRT), el campo de desplazamiento es de 8 bits (instrucción tipo A), que según las pruebas estadísticas de ejecución de [1, pp.82-84], cubre el 90% de los saltos condicionales.

4. PARAMETRIZACIÓN DEL ANCHO DE PALABRA DE DATOS

Como se estableció en la condición 2, el tamaño de la palabra de datos es parametrizable. Esto se pensó sobre todo para que el procesador sea configurado en los modos de 16 ó 32 bits (si bien otros valores son posibles, no se ha hecho un estudio aún en ese sentido). Para el diseño de la arquitectura se estableció que el código de 16 bits pueda correr sin cambios en procesadores configurados en modo 32 bits. A la inversa no se asegura compatibilidad. Las únicas instrucciones no presentes en el modo 16 son LW y SW, las cuales cargan y almacenan palabras de 32 bits entre la memoria y los registros. La aparición de sus *opcodes* en un procesador de 16 bits generará una excepción interna. Una de las ventajas principales de esta arquitectura reside justamente en que puede mantener un largo de programa reducido (debido a las instrucciones de 16 bits) y trabajar con datos de 32 bits y una capacidad de direccionamiento de 4 Giga bytes.

Transferencia de datos		
Instr.	***Tipo***	***Designación***
LW	M	Load Word (sólo modo 32 bits)
SW	M	Store Word (sólo modo 32 bits)
LH	M	Load Half word
SH	M	Store Half word
LB	M	Load Byte
SB	M	Store Byte
MOV	R	MOVe Data from register to register
Control de programa		
Instr.	***Tipo***	***Designación***
JAL	J	Jump And Link
JRAL	A	Jump Register And Link
BRT	A	Branch if register's Test bit is set
Comparación		
Instr.	***Tipo***	***Designación***
TEQ	R	set Test bit if Equals
TGTU	R	set Test bit if Greater Than (Unsigned)
TGEU	R	set Test bit if Greater or Equal than (Unsigned)
TGTS	R	set Test bit if Greater than (Signed)
TGES	R	set Test bit if Greater or Equal than (Signed)
TCF	R	set Test bit if carry flag is set
TOF	R	Set Test bit if overflow flag is set
Flags		
Instr.	***Tipo***	***Designación***
MRFR (*)	R	Move register Flags to register
MRRF (*)	R	Move register to register flags.
Aritméticas / Lógicas		
Instr.	***Tipo***	***Designación***
ADD	R	ADD registers
ADDI	A	ADD Immediate to register
SUB	R	SUBstract registers
CPI	A	ComPare Immediate
SRA	R	Shift Right Arithmetic
SRAI	S	Shift Right Arithmetic Immediate
SRL	R	Shift Right Logical
SLL	R	Shift Left Logical
SLLI	S	Shift Left Logical Immediate
SRLI	S	Shift Right Logical Immediate
AND	R	logical AND registers
ANDI	A	logical AND register with Immediate
OR	R	logical OR registers
ORI	A	logical OR register with Immediate
XOR	R	logical XOR registers
NOR	R	logical NOR registers
NEG	R	2's complement NEGate register

5. RESUMEN DEL CONJUNTO DE INSTRUCCIONES

5.1. Instrucciones

En la tabla 2 se resume el conjunto completo de instrucciones, con el formato (tipo) al que pertenecen.

Tabla 2. Conjunto de instrucciones completo.

(*) Estas instrucciones copian el flag T, C o V de todos los registros a la mitad baja de un registro, o el contenido de la mitad baja de un registro al registro del flag T, C o V de todos los registros.

5.2. Pseudo instrucciones

En la tabla 3 se ven algunas instrucciones que en realidad son mapeos realizados por el ensamblador sobre el conjunto real del procesador.

Instr.	***Sintaxis***	***Instr. real***	***Designación***
NOT	NOT Ri	NOR Ri, Ri	logical NOT register
CLR	CLR Ri	XOR Ri, Ri	CLear Register
NOP	NOP	AND Ri, Ri	No OPeration
SUBI	SUBI Ri, K8	ADDI Ri, -K8	SUBstract Immediate

Tabla 3. Algunas pseudo instrucciones posibles.

Ri representa a cualquier registro general (R0-R16); K8 y K12 son valores inmediatos de 8 y 12 bits, respectivamente.

5.3. Espacio de instrucciones

Con un *opcode* de 4 bits el espacio total de instrucciones es reducido. Por lo que fue necesario optimizar su uso para dotar al procesador de las instrucciones mínimas que se consideraron necesarias. Por ejemplo, puede notarse la falta de instrucciones para carga de valores inmediatos. Esto se soluciona, por ejemplo para una operación de carga inmediata de 32 bits, con la implementación de la siguiente secuencia de instrucciones:

```
0006  JAL  4                (1)
0008  LW  Rx, [2(R0)]       (2)
000A  16 bits altos         (3)
000C  16 bits bajos         (4)
000E
```

(1) Este salto causa que se cargue en el registro Ro la dirección de la siguiente instrucción en secuencia.
(2) Esta instrucción es ejecutada en el delay slot del salto.
(3) Bits altos del valor inmediato a ser cargado
(4) Bits bajos del valor inmediato a ser cargado

6. CONCLUSIONES Y ESTADO DEL DESARROLLO

Es cierto que el verdadero valor de una arquitectura recién se conoce al ser sometida al análisis estadístico sobre una implementación (ya sea de *hardware* o simulada) del procesador en cuestión. La Eq. (1), o ecuación de *performance* según [1], permite establecer cuantitativamente el desempeño en base a los datos obtenidos de los *benchmarks*.

$$CPU_{time} = \frac{(IC \cdot CPI)}{CR} \qquad [1]$$

donde:

- IC (*Instruction Count:* cantidad de instrucciones de un programa determinado) es un valor estadístico que se obtiene de la compilación de programas adecuadamente seleccionados (en nuestro caso *benchmarks* estándar para procesadores sin unidad de punto flotante compilados con el *gcc*).
- CR (*Clock Rate*: frecuencia del reloj) se desprende del análisis de tiempos hecho por el programa de síntesis sobre la descripción en VHDL.
- CPI (*Clock Cycles per Instruction*: ciclos de reloj por instrucción): Si bien se puede aproximar teóricamente, su valor también es estadístico y se deriva de las pruebas de ejecución en el simulador (ya que lo afectan aspectos como las dependencias entre instrucciones entre otros).
- CPU_{time} es el valor a minimizar e indica el tiempo de ejecución total del programa o conjunto de programas evaluados.

Las estimaciones que se realizaron hasta ahora, indican que para nuestro microprocesador se generará aproximadamente un 50 % más de instrucciones que las presentadas en las estadísticas de [1]. Cabe destacar que las instrucciones aquí utilizadas son de 16 bits, contra 32 de [1].

Al momento de escribir este artículo se terminaba el proceso de migración del compilador de GNU (gcc) a nuestro procesador, como así también la construcción de un simulador por software y una implementación de referencia aproximada en VHDL (*VHSIC Hardware Description Language*). Estas son herramientas imprescindibles para poder realizar las pruebas comparativas de la arquitectura y evaluar cuantitativamente su desempeño -Eq. (1)-.

Con la finalización del compilador comenzó también la migración del kernel de Linux sobre esta arquitectura. Se espera de este modo que las propuestas aquí consignadas y las herramientas desarrolladas puedan servir como base para avanzar en el desarrollo de un procesador con instrucciones de 16 bits y palabras de datos parametrizables (16-32). Creemos que puede resultar especialmente interesante la solución planteada para el sistema de saltos condicionales, muy ligado a los *flags* de estado por registro.

Otro punto en el que se hizo especial hincapié fue en el de mantener la ortogonalidad y la simpleza del procesador, de modo que no se recurrió a "desprolijidades", muy usadas incluso en procesadores que comercialmente sus fabricantes denominan RISC. Ejemplos de estas prácticas son *opcodes*, e incluso instrucciones, de largo variable, registros especiales para carga de constantes o para mantener *flags* de estado, etc.. Todo esto suele redundar en la aparición de dependencias (dificultando la optimización del código) o en ciclos de ejecución por instrucción más largos, lo cual además trae aparejada una falta de balance que complica la lógica de *pipelining* o incluso agrega tiempos muertos. Sin mencionar que además pueden complicar la lógica. La implementación del procesador funcionando con palabras de datos de 16 bits y *pipeline* de 5 etapas requiere aproximadamente 300 elementos lógicos en una ACEX1K de Altera, la cual puede operar hasta 100 MHz con el dispositivo más rápido y hasta 30 MHz con el más lento de la familia.

7. REFERENCIAS

[1] Patterson, D. A., Hannessy, J. L., con contribución de D. Goldberg. *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers, San Francisco, U.S.A., 1996.

[2] Altera Corporation. *Nios Embedded Processor: Programmer's Reference Manual*, Altera Corporation, U.S.A., Julio de 2001.

[3] Atmel Corporation, con acuerdo de licencia de Advanced RISC Machines Limited, *ARM7TDMI™: (Thumb®) Datasheet*, Atmel ES2, Francia, Enero de 1999.

[4] Altera Corporation. *Nios Embedded Processor: Programmer's Reference Manual*, Altera Corporation, U.S.A., Julio de 2001.

[5] Xilinx, Inc.. *MicroBlaze Software Reference Guide*, U.S.A., Abril de 2002.