# Diseño de una Cola de Instrucciones para Acceso a Memoria (LSQ) Basado en la Política de Envejecimiento de Instrucciones

Eduardo Pacheco-González, César A. Hernández-Calderón, Mauricio Ontiveros-Rodríguez

Centro de Investigación en Computación, Instituto Politécnico Nacional D. F., México.

pachecoeg086@gmail.com, hdzces@outlook.com, ont.mauricio@gmail.com

**Abstract.** In this paper we present a Load/Store Queue (LSQ) design for Memory Access which operates under the aging instructions principle whit the aim of achieving memory disambiguation. This module has been developed considering its integration into a super-scalar processor´s pipeline, capable of issue two instructions per clock cycle. The LSQ has been described in Verilog on QuartusII program and the simulations were performed in ModelSim

**Resumen.** En el presente trabajo se expone el diseño de una Cola de Instrucciones para Acceso a Memoria (*Load/Store Queue* - LSQ) que opera bajo el principio de envejecimiento de instrucciones con el objetivo de conseguir la desambiguación de memoria. Este módulo ha sido desarrollado considerando su integración al *pipeline* de un procesador súper-escalar capaz de emitir dos instrucciones por ciclo de reloj. La LSQ ha sido descrita en Verilog sobre el programa QuartusII, y las simulaciones se han realizado en ModelSim.

**Keywords:** Diseño de Arquitectura de Computadoras, Acceso a Memoria, Colas FIFO, HDL, Verilog, FPGA.

## 1 Introducción

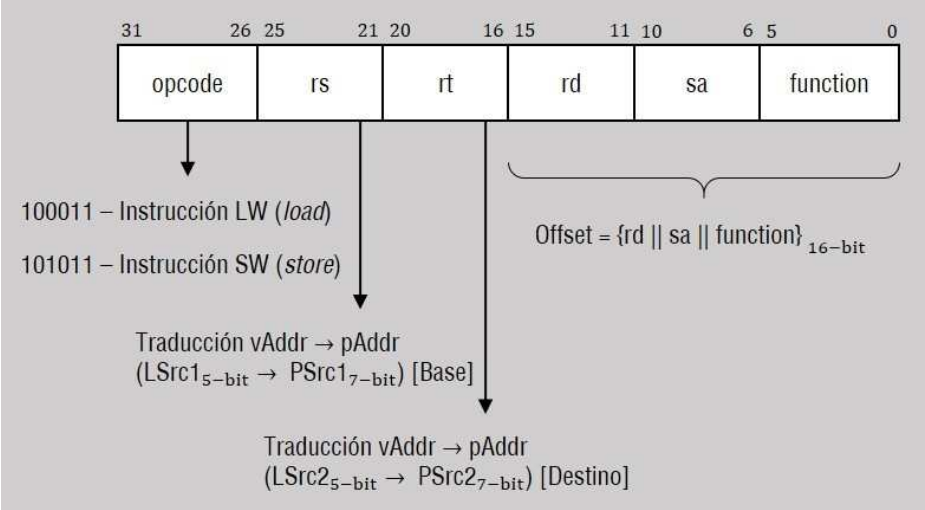
Hoy en día, los arquitectos de computadoras han mejorado el desempeño de los procesadores al implementar *pipelines* con mayor profundidad, al engrosar la emisión de instrucciones, y al ampliar la capacidad de la ventana de instrucciones con ejecución fuera de orden. En consecuencia, se han obtenido procesadores capaces de tener más de cien instrucciones en vuelo, incrementando la cantidad instrucciones de acceso a memoria. Para garantizar la ausencia de violaciones durante el acceso a memoria es necesario asegurar una desambiguación de memoria implementando una cola de ins-


Advances in Computer Science
Research in computing Science 83, 2014 pp. 37-48

trucciones (*Load/Store Queue* - LSQ) que determine, a partir de alguna política de ordenamiento, que instrucciones tienen mayor prioridad para acceder al contenido de la memoria [1][2][3].

Las instrucciones de acceso a memoria tipo MIPS se clasifican de forma general en dos grupos: *load* y *store*. Estas instrucciones, además de acceder a la Memoria de Datos, tienen acceso al Banco de Registros de Enteros (*Integer Register File* – IRF). Las instrucciones de tipo ***load*** son todas aquellas instrucciones con las que se desea actualizar algún valor del IRF, por lo que estas instrucciones requieren de una **lectura** de la Memoria de Datos para después escribir el dato en el IRF. Las instrucciones de tipo ***store*** son todas aquellas instrucciones con las que se desea almacenar un dato previamente procesado, en la Memoria de Datos, por lo que inicialmente se realiza la lectura del IRF, y posteriormente, la **escritura** en la Memoria de Datos.

Los procesadores que basan su diseño en un conjunto de instrucciones tipo MIPS trabajan con instrucciones con una longitud de 32-bits en las que se pueden definir seis campos principales al tratarse de instrucciones de CPU: *opcode*, *rs*, *rt*, *rd*, *sa* y *function* (Fig. 1)[1]. El campo *opcode* contiene el código principal para cualquier instrucción tipo MIPS, y es en donde se especifica la instrucción que se desea procesar. En el conjunto de instrucciones para MIPS64 se cuenta con veintiuna instrucciones de tipo *load* distintas, y con diecisiete instrucciones de tipo *store* [4].




[1] Dependiendo del tipo de instrucción con la que se trabaje (COP0-EH, COP1-FP, COP2, etc.) los campos en los que se divide la instrucción serán distintos, así como la operación a la que hacen referencia cada campo, sin que su longitud de 32-bits varíe. Cabe resaltar que el único campo que se encuentra presente para cualquier tipo de instrucción es el campo *opcode*.

**Fig. 1.** Formato de una instrucción de 32-bits tipo MIPS. El campo *opcode* permite identificar si se trata de una instrucción de acceso a memoria. Los campos *rs* y *rt* (5-bits cada uno) son direcciones virtuales (vAddr) que son traducidas a direcciones físicas (pAddr) para realizar la lectura/escritura del Banco de Registros. Los campos *rd*, *sa* y *function* son usados para definir el valor del *offset* (16-bit) usado para calcular la dirección de lectura/escritura de la Memoria de Datos, según sea el caso.

Los campos *rs* y *rt* son direcciones virtuales (vAdrr) de 5-bits que deben ser traducidas a direcciones físicas (pAddr) de 7-bits por una Unidad de Renombrado de Registros para realizar la lectura y escritura en el Banco de Registros, como se explicará más adelante. Por último, el valor del *offset* requerido para calcular la dirección de acceso a memoria estará conformado por los valores de los campos *rd*, *sa* y *function*[2].

El diseño que aquí se presenta ha sido desarrollado para integrarse en el procesador Lagarto II [5], un procesador súper-escalar con una arquitectura de 64-bits tipo RISC, y ejecución fuera de orden, capaz de emitir dos instrucciones tipo MIPS en cada ciclo de reloj.

## 2 Diseño y Operación de la LSQ

Para el diseño de la LSQ se debe considerar que ésta realiza un intercambio de datos con cuatro módulos del procesador: el Mapper, el Registro de Estados, el IRF y la Memoria de Datos (Fig. 2). Así mismo, recibe las señales de reloj y reset (clk y rst, respectivamente), además de un bit de control adicional denominado MemRdy, que determina si la memoria se encuentra disponible para ser escrita o leída. La LSQ ha sido descrita en Verilog sobre el programa QuartusII de la compañía Altera.

[2] Debido a que el contenido de los campos *rd*, *sa* y *function* no es considerado de forma independiente en las instrucciones de acceso a memoria, no es necesario el conocer la descripción del procesamiento de esta información durante su recorrido por el *pipeline* para nuestros fines.

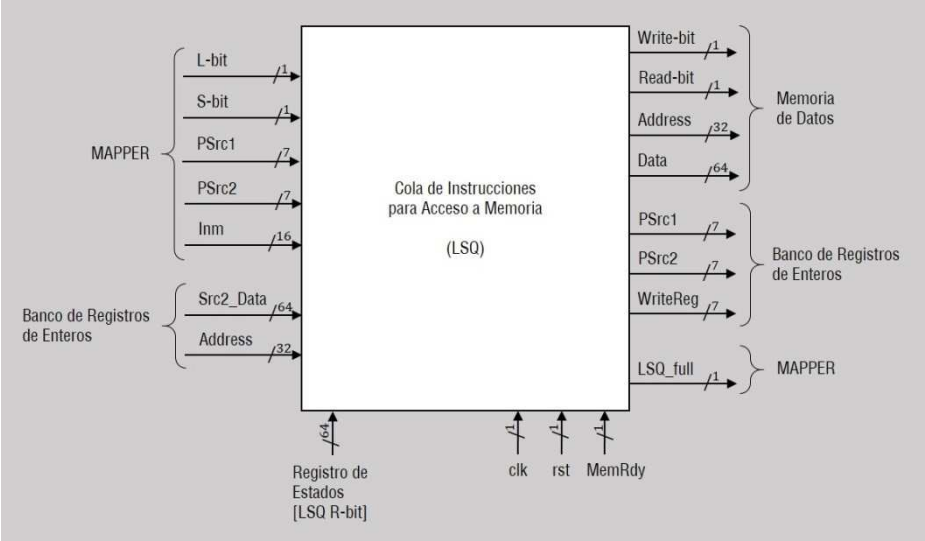



**Fig. 2.** En este esquema se muestran las señales de entrada y salida de la LSQ con las que se encuentra integrada procesador, siendo cuatro módulos con los que interactúa: Memoria de Datos, IRF, Mapper y el Registro de estados. Éste último, en combinación con el bit de control MemRdy, juegan un papel importante para el control de la operación del módulo.

En la Fig. 3 se presenta a detalle el diseño de la microarquitectura la LSQ, donde destacan los cinco módulos que forman la LSQ: Planificador de Instrucciones, Unidad de Extensión de Signo, Unidad de Cálculo de Direcciones (*Address Calculation Unit* – ACU), Buffer de Instrucciones y la Lógica de control.

## 2.1 Operación General de la LSQ

La información de las instrucciones proviene del Mapper, el cual organiza y emite dicha información hacia el Planificador de Instrucciones, que es el primer módulo que integra la LSQ, siempre y cuando el bit LSQ_full se lo permita. Este bit de control le hace saber al Mapper si el Planificador o el Buffer cuentan con espacio para poder almacenar los datos que son gestionados por la cola, este conteo es administrado por la Lógica Control.

El IRF se encuentra inmerso en el camino de datos de la LSQ. Sin importar el tipo de instrucción de la que se trate, se realiza la lectura del IRF en la dirección PSrc1 y la extensión de signo del *offset* en el momento en el que la información sale del Planificador. El dato resultante de la lectura del IRF es usado como operador en la ACU, en combinación con el valor del *offset* con signo extendido, para calcular la dirección con la que se tendrá acceso a la memoria (eq. 1). Si se trabaja con una instrucción de tipo *load*, se actualizará el registro WriteReg del IRF con el dato WriteData procedente de la memoria, mientras que al trabajar con una instrucción de tipo *store*, la dirección PSrc2 será almacenada de forma directa en el Buffer.

$$Address = base_{32} + \{signo\ extendido_{16} || offset_{16}\} \quad (1)$$

La Memoria de Datos recibe una dirección de 32-bits (Address) y los bits de control para escritura (Write-bit) o lectura (Read-bit). Al tratarse de una instrucción tipo *load*, la señal Read-bit deberá estar en alto, por lo que se indicará a la Memoria de Datos que se requiere leer la información de la dirección Address. Por otro lado, si se trata de una instrucción de tipo *store*, la señal Write-bit deberá estar en alto, y se escribirá en la dirección Address de la memoria un dato de 64-bits (Data) procedente del IRF.

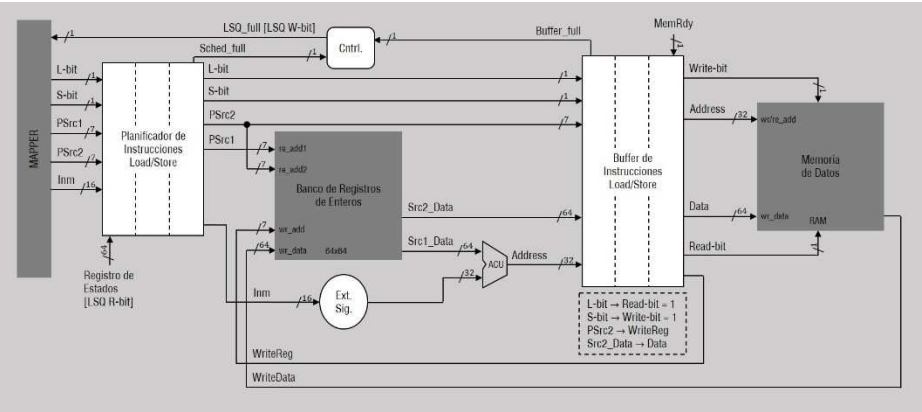



**Fig. 3.** El diseño de la LSQ comprende 5 módulos: Planificador de Instrucciones, Unidad de Extensión de Signo, Unidad de Cálculo de Direcciones (ACU), Buffer de Instrucciones y Lógica de Control. Los componentes ilustrados en gris son los elementos del procesador con los que interactúa la LSQ y no pertenecen propiamente al diseño de la misma.

## 2.2 Planificador de Instrucciones Load/Store

El Planificador de Instrucciones es el que marca la pauta a la LSQ sobre la información que será procesada ya que almacena y gestiona la emisión de los datos asociados a cuatro instrucciones de acceso a memoria (registros fuente, offset, bits de control, etc.) de forma simultánea. Estos datos procedentes del Mapper han sido previamente procesados por otros módulos incluidos en el *pipeline* del procesador.

Una vez almacenada la información, siempre la instrucción más vieja debe de ser emitida, para que lo anterior se cumpla, el vector de Registro de Estados es el encargado de indicar al Planificador que los registros fuente necesarios (PSrc1 y PSrc2) se encuentran listos, es decir, que el dato que almacena cada registro es válido para ser leído; Si se cumple finalmente, el Planificador genera la señal de control que permite la escritura de datos en el Buffer de Instrucciones, pero si lo anterior no llega a suceder el Planificador permitirá la entrada de datos hasta completar sus espacios disponibles de memoria, cuando se llegue a saturar el Planificador indicara al procesador que no alimente más instrucciones hasta tener espacios de memoria disponibles, este proceso es conocido como “Despertado de Instrucciones”.

### 2.3 Buffer de Instrucciones Load/Store

El Buffer de instrucciones, al igual que el Planificador, es de tipo *FIFO* (lo primero que entra es lo primero que sale), y está constituido por cinco ranuras para el almacenamiento de datos. Cada ranura está conformada por cinco campos para almacenar la información con la que se realizará el acceso a memoria: W-bit, R-bit, WriteReg, Address y Data.

Que la estructura sea de tipo FIFO nos permite mantener siempre el control sobre el orden en que nuestras instrucciones son emitidas, y así mantener la coherencia de las instrucciones contenidas en la LSQ.

Cuando la señal de control habilita la escritura en este módulo se almacenan los datos emitidos por el Planificador y los calculados en la LSQ. Este módulo representa el paso previo del acceso a memoria, por lo que la información que emita deberá de corresponder con el tipo de acceso que se desea realizar, ya sea lectura o escritura. Lo anterior es crítico, ya que de no tener almacenados los datos correctos en el Buffer la ejecución de instrucciones posteriores en el procesador arrojarán resultados erróneos, y en consecuencia, una mala ejecución del programa. Tomando en cuenta esto, se incluye una lógica que habilita las líneas de salida correspondientes al tipo de acceso a memoria, en las que se han almacenado datos válidos (Tabla 1).

**Table 1.** Campos válidos del Buffer de Instrucciones para el acceso a la Memria de Datos.

| Instrucción | W-bit | R-bit | WriteReg | Address | Data |
|---|---|---|---|---|---|
| Load | 0 | 1 | ✓ | ✓ | - |
| Store | 1 | 0 | - | ✓ | ✓ |

## 3 Validación

Para verificar el correcto funcionamiento del diseño y debido a que el procesador Lagarto II aún se encuentra la fase de diseño y con partes del pipeline aun en papel, la LSQ se ha integrado a la arquitectura de un Coprocesador vectorial [6] y se han realizado diversas simulaciones ejecutadas en ModelSim.

En la Fig. 4 y Fig. 5 se observa el resultado de la simulación del Coprocesador sin haber ejecutado ningún tipo de instrucción. En la Fig. 4 se muestra que el contenido inicial del IRF es de ceros, lo que indica que no hay ningún dato disponible para ser procesado por alguna instrucción. Por otra parte, en la Fig. 5, se muestra el contenido inicial de la Memoria de Datos, mismos que deben ser cargados al IRF mediante instrucciones de tipo *load* para trabajar con ellos.

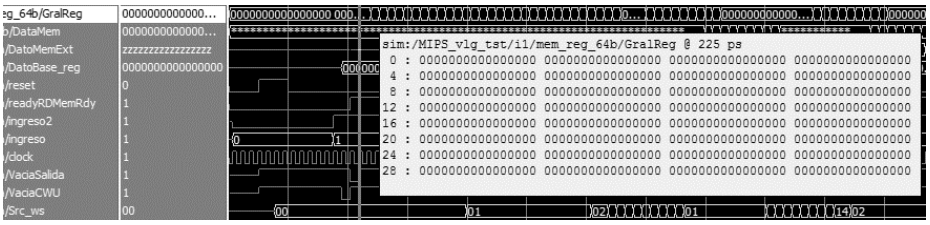
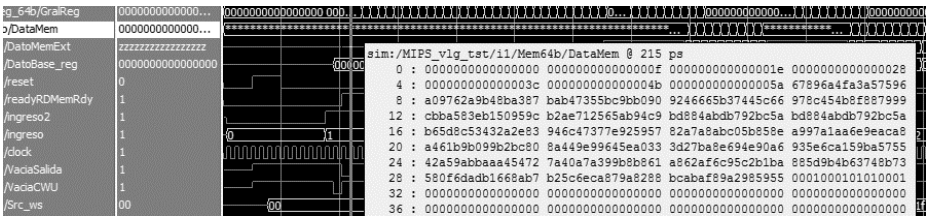



**Fig. 4.** Simulación del Coprocesador antes de ejecutar las instrucciones de tipo *load* para cargar los datos al IRF. En el recuadro de lado derecho se observa el contenido inicial del IRF.




**a)**

**b)**

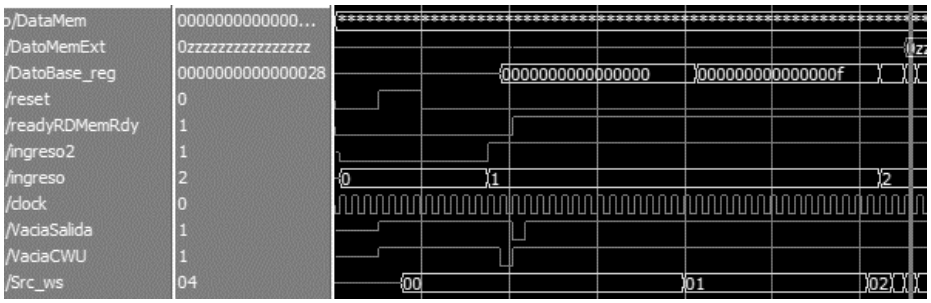
**Fig. 5.** a) Simulación del Coprocesador antes de ejecutar las instrucciones de tipo *load* para cargar los datos al IRF. b) En el recuadro se observa a detalle el contenido inicial de la Memoria de Datos.

Para las simulaciones posteriores se ha inicializado la Memoria de Instrucciones con un programa para ser procesado en el *pipeline* del Coprocesador. Las primeras diez instrucciones corresponden a la carga de información desde la Memoria de Datos hacia el IRF (Tabla 2).

En total se ejecutan treinta y dos instrucciones de tipo *load* con la finalidad de almacenar un nuevo dato en cada espacio disponible del IRF. La simulación resultante de la ejecución de esta secuencia de instrucciones tipo *load* se observa en la Fig. 6. Como se aprecia, el contenido del IRF ha dejado de ser cero, y en consecuencia, se tienen los datos necesarios para ejecutar instrucciones aritmético-lógicas en el Coprocesador.

**Table 2.** Primeras instrucciones almacenadas en la Memoria de Instrucciones

| Registro | Inst. | rs | rt | rd | HEX |
|---|---|---|---|---|---|
| 0 | LD.df | $0 | 0x0 | $0 | 4B000007 |
| 1 | LD.df | $0 | 0x1 | $0 | 4B010047 |
| 2 | LD.df | $0 | 0x2 | $0 | 4B020087 |
| 3 | LD.df | $0 | 0x3 | $0 | 4B0300C7 |
| 4 | LD.df | $0 | 0x4 | $0 | 4B040107 |
| 5 | LD.df | $0 | 0x5 | $0 | 4B050147 |
| 6 | LD.df | $0 | 0x6 | $0 | 4B060187 |
| 7 | LD.df | $0 | 0x7 | $0 | 4B0701C7 |
| 8 | LD.df | $0 | 0x8 | $0 | 4B080207 |
| 9 | LD.df | $0 | 0x9 | $0 | 4B090247 |




**a)**

```
sim:/MIPS_vlg_tst/i1/mem_reg_64b/GralReg @ 659 ps
 0 : 0000000000000000 000000000000000f 000000000000001e 0000000000000028
 4 : 000000000000003c 000000000000004b 000000000000005a 67896a4fa3a57596
 8 : a09762a9b48ba387 bab47355bc9bb090 9246665b37445c66 978c454b8f887999
12 : cbba583eb150959c b2ae712565ab94c9 bd884abdb792bc5a bd884abdb792bc5a
16 : b65d8c53432a2e83 946c47377e925957 82a7a8abc05b858e a997a1aa6e9eaca8
20 : a461b9b099b2bc80 8a449e99645ea033 3d27ba8e694e90a6 935e6ca159ba5755
24 : 42a59abbaaa45472 7a40a7a399b8b861 a862af6c95c2b1ba 885d9b4b63748b73
28 : 580f6dadb1668ab7 b25c6eca879a8288 bcabaf89a2985955 0001000101010001
```

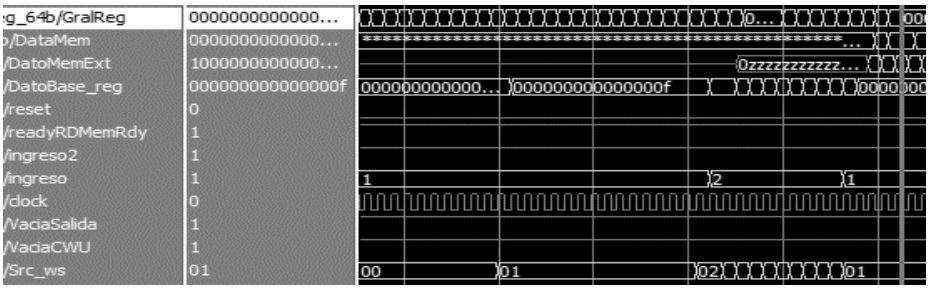
**b)**

**Fig. 6.** a) Simulación del Coprocesador después de ejecutar las instrucciones de tipo *load* para la carga de datos al IRF. b) En el recuadro se muestra a detalle el contenido del IRF después de la ejecución de las instrucciones de carga de datos.

Posteriormente, se ejecuta una serie de instrucciones de suma (addv) y resta (subv) para procesar los datos cargados anteriormente al IRF (Tabla 3). Con la ejecución de estas instrucciones se lleva a cabo una modificación en el contenido del IRF, ya que cada instrucción contiene un registro destino (*rt*) en el que será escrito el resultado.

**Table 3.** Instrucciones de suma y resta para el procesado de los datos cargados en el IRF

| Inst. | Op. | rs | rt | rd | HEX |
|---|---|---|---|---|---|
| 32 | ADDV.df | $2 | $1 | $21 | 4801154E |
| 33 | ADDV.df | $3 | $2 | $22 | 4801154E |
| 34 | ADDV.df | $4 | $3 | $23 | 4801154E |
| 35 | ADDV.df | $5 | $4 | $24 | 4801154E |
| 36 | ADDV.df | $6 | $5 | $25 | 4801154E |
| 37 | SUBVdf | $7 | $6 | $26 | 4801154E |
| 38 | SUBVdf | $8 | $7 | $27 | 4801154E |
| 39 | SUBVdf | $9 | $8 | $28 | 4801154E |
| 40 | SUBVdf | $10 | $9 | $29 | 4801154E |
| 41 | SUBVdf | $11 | $10 | $30 | 4801154E |




**a)**

```
sim:/MIPS_vlg_tst/i1/mem_reg_64b/GralReg @ 835 ps
 0 : 0000000000000000 000000000000000f 000000000000001e 0000000000000028
 4 : 000000000000003c 000000000000004b 000000000000005a 67896a4fa3a57596
 8 : a09762a9b48ba387 bab47355bc9bb090 9246665b37445c66 978c454b8f887999
12 : cbba583eb150959c b2ae712565ab94c9 bd884abdb792bc5a bd884abdb792bc5a
16 : b65d8c53432a2e83 946c47377e925957 82a7a8abc05b858e a997a1aa6e9eaca8
20 : a461b9b099b2bc80 000000000000002d 0000000000000046 0000000000000064
24 : 0000000000000087 00000000000000a5 997796b15d5b8bc4 c6f207a6ef1ad20f
28 : e5e2ef54f7eff2f7 286e0cfa8557542a faba210fa7bbe2cd 0001000101010001
```

**b)**

**Fig. 7.** a) Simulación del Coprocesador después de ejecutar las operaciones de suma y resta almacenadas en la Memoria de Instrucciones. b) En el recuadro se muestra el contenido del IRF con los resultados de cada operación ejecutada.

Una vez realizadas las operaciones requeridas, se realiza el respaldo de los datos obtenidos escribiéndolos en la Memoria de Datos mediante un conjunto de instrucciones de tipo *store*, con lo que se llevará la información del IRF a la Memoria de Datos. Este conjunto de instrucciones se muestra en la Tabla 4.

**Table 4.** Instrucciones de tipo *store* para realizar la actualización de la Memoria de Datos.

| Inst. | Op. | rs | rt | rd | HEX |
|---|---|---|---|---|---|
| 42 | ST.df | $1 | 0x6 | $21 | 4801154E |
| 43 | ST.df | $1 | 0x7 | $22 | 4801154E |
| 44 | ST.df | $1 | 0x8 | $23 | 4801154E |
| 45 | ST.df | $1 | 0x9 | $24 | 4801154E |
| 46 | ST.df | $1 | 0xa | $25 | 4801154E |
| 47 | ST.df | $1 | 0xb | $26 | 4801154E |
| 48 | ST.df | $1 | 0xc | $27 | 4801154E |
| 49 | ST.df | $1 | 0xd | $28 | 4801154E |
| 50 | ST.df | $1 | 0xe | $29 | 4801154E |
| 51 | ST.df | $1 | 0xf | $30 | 4801154E |

Los resultados de la simulación se aprecian en la Fig. 8.


**a)**

```
sim:/MIPS_vlg_tst/i1/Mem64b/DataMem @ 939 ps
    0 : 0000000000000000 000000000000000f 000000000000001e 0000000000000028
    4 : 000000000000003c 000000000000004b 000000000000005a 67896a4fa3a57596
    8 : a09762a9b48ba387 bab47355bc9bb090 9246665b37445c66 978c454b8f887999
   12 : cbba583eb150959c b2ae712565ab94c9 bd884abdb792bc5a bd884abdb792bc5a
   16 : b65d8c53432a2e83 946c47377e925957 82a7a8abc05b858e a997a1aa6e9eaca8
   20 : a461b9b099b2bc80 000000000000002d 0000000000000046 0000000000000064
   24 : 0000000000000087 00000000000000a5 997796b15d5b8bc4 c6f207a6ef1ad20f
   28 : e5e2ef54f7eff2f7 286e0cfa8557542a faba210fa7bbe2cd 0001000101010001
   32 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000
   36 : 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

**c)**

**Fig. 8.** a) Simulación del Coprocesador después de ejecutar las instrucciones de tipo *store* para el almacenamiento de datos desde el IRF a la Memoria de Datos. b) En el recuadro se aprecia contenido de la Memoria de Datos después del respaldo de los datos resultantes de las operaciones de suma y resta.

## 4 Conclusiones

El diseño de una LSQ basado en el envejecimiento de instrucciones permite realizar el ordenamiento de las instrucciones de acceso a memoria con la finalidad de mantener la coherencia durante la ejecución de las instrucciones. En el desarrollo de este trabajo se describe a detalle el diseño de la cola, la cual requirió de estructuras de tipo FIFO y diversa lógica combinacional, en el diseño se contempla el intercambio de datos con otros módulos incluidos en el *pipeline* de un procesador súper-escalar. Así mismo, se presenta la implementación de la LSQ integrada al *pipeline* de un coprocesador vectorial, con la finalidad de verificar el correcto funcionamiento del módulo.

## 5 Referencias

1. Sethumadhavan, S., Desikan, R., Burger, D., Moore, C.R. y Keckler, S.W.: *Scalable Hardware Memory Disambiguation for High ILP Processors.* En: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA. (2003)
2. Il-Park, Chong-Lian Ooi y Vijaykumar, T.N.: *Reducing Design Complexity of the Load/Store Queue.* En: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA. (2003)
3. Baugh, L. y Zilles, C.: *Decomposing the load-store queue by function for power reduction and scalability*. IBM J. Res. Dev. 50, 2/3, pp. 287-297. (2006)
4. MIPS Technologies: MIPS64 Architecture for Programmers Volume II: The MIPS64 Instructions Set. Mountain View, CA, USA. (2005)
5. Reporte Técnico. Diseño de un procesador superescalar con ejecución fuera de orden usando lenguajes de descripción de hardware (VHDL). IPN-CIC Laboratorio de Micro-tecnología y Sistemas Embebidos. (2008)
6. Pacheco G. E. "Diseño de Procesadores Morfológicos tipo SIMD". Tesis de Maestría. México. CIC-IPN. (2014)
7. Ramírez-Salinas, M.A.: *Arquitectura del Sistema de Memoria*: http://www.microse.cic.ipn.mx
8. Dreslinski, R.: *Memory Speculation*: http://www.eecs.umich.edu/eecs/courses/eecs470/