Proyecto Final: Implementación de una Micro-Arquitectura para ARMv4

1st Josué-Daniel Cubero-Montero
Instituto Tecnológico de Costa Rica
Área académica Ing. en Computadores
Taller de Diseño Digital
Cartago, Costa Rica
josuecub.mont@hotmail.com

2nd David Quesada-Calderón
Instituto Tecnológico de Costa Rica
Área académica Ing. en Computadores
Taller de Diseño Digital
Cartago, Costa Rica
davidqc05@gmail.com

3th Gerald Valverde-Mc kenzie
Instituto Tecnológico de Costa Rica
Área académica Ing. en Computadores
Taller de Diseño Digital
Cartago, Costa Rica
geraldvm167@gmail.com

***Abstract*—The project consist of an implementation to solved an engineer problem that is create an ARM processor in charge of decrypting some ASCII characters applying algoritms like adder, XOR and NOT operations. Also, after decrypt the data the system have to show the ASCII characters in a VGA monitor.**

***Index Terms*—Arquitectura, Micro-Arquitectura, ARMv4, Lenguaje Maquina.**

I. Introducción

Podría decirse que, una **arquitectura** es aquello que responde al como un programador ve a una computadora. Dicha arquitectura es definida por un conjunto de instrucciones, a menudo llamada ISA (**I**nstruction **S**et **A**rquitecture). Debido a esto, para entender una arquitectura, debemos de aprender primeramente su lenguaje, o dicho de otro modo, sus instrucciones. Estas instrucciones tienen la capacidad de hacer todo tipo de operaciones primitivas, desde operaciones aritméticas y lógicas hasta operaciones de salto(o tipo Branch), las cuales operan sobre valores inmediatos, registros o incluso sobre la memoria. La manera en la que estas instrucciones están constituidas, es tanto por la operación a ejecutarse, como por los operandos propios de la operación.

A pesar de que el hardware de una computadora solo entiende 1's y 0's, el conjunto de instrucciones de una arquitectura se presenta de manera simbólica, pues así se evita la tediosa tarea de programar mediante el uso del **lenguaje maquina**(Grupo de 1's y 0's). Esta representación simbólica de las instrucciones es llamada **Lenguaje Ensamblador**

Particularmente, la arquitectura ARM representa cada una de sus instrucciones en un formato de 32-bits [1, p. 295-296].

Como debió de notarse, hasta este punto no se ha hablado sobre como estas instrucciones son finalmente ejecutadas. La razón de esto, es que, a la Arquitectura de una computadora no le corresponde describir el proceso de ejecución de las instrucciones; por lo contrario, en este punto, los detalles del modo en el que las instrucciones son ejecutadas se ignoran, pues lo único relevante para la arquitectura es el como estas instrucciones se compartan. De este modo, si nos queremos referir a los registros, unidades aritméticas y lógicas o a las memorias -principales actores en la ejecución de una instrucción- debemos de referirnos a lo que se conoce como **micro arquitectura**, lo cual, es el tema abordado en este paper.

Para entender con claridad el porque de cada una de las distintas etapas de hardware que serán implementadas, a continuación se describirán los tipos de instrucciones de ARMv4 y su codificación a lenguaje maquina. Además, se mencionará el proceso mediante el cual se simuló y tradujo cada una de las instrucciones.

A. Lenguaje Maquina

Como se mencionó anteriormente, el **lenguaje ensamblador** es una representación simbólica del **lenguaje maquina**. Sin embargo, los circuitos digitales solo entienden 1's y 0's. Por lo tanto, todo programa en lenguaje ensamblador debe de traducirse a su representación equivalente en lenguaje maquina. En ARM se definen tres tipos principales de formatos para las instrucciones:

- Instrucciones de procesamiento de datos.
- Instrucciones de memoria.
- Instrucciones Branch.

El hecho de que en ARM cada una de sus instrucciones sean representadas con 32-bit -a pesar de que no todos lo bits sean siempre necesarios- se fundamenta en dos principios de diseño [2]:

- La regularidad fomenta la simplicidad.
- Buenos diseños demandan buenos compromisos.

Ahora bien, como ya se mencionó, cada instrucción está compuesta de 32-bits. La pregunta que debemos de plantearnos en este punto es: ¿Como identificar cada una de las partes en una instrucción en ensamblador que están siendo representas por 1's y 0s en lenguaje maquina?

Para esto, este conjunto de 32-bits se divide en grupos a lo interno de la instrucción, los cuales tienen un significado propio según el tipo instrucción que se tenga. En realidad, lo única sección que comparten los tres tipos de instrucciones de manera idéntica son sus 4-bits mas significados, donde estos representan la condición que debe de comprobarse para ejecutar o no la instrucción en cuestión. El formato que siguen estos 4-bits para algunos condicionales, se muestra en la Tabla I [1, p. 307].

TABLE I
CONDICIONES PARA LA EJECUCIÓN DE LAS INSTRUCCIONES

Condición (cond)	Mnemonico	Nombre
0000	EQ	Equal
0001	NE	Not Equal
1010	GE	Greater than or Equal
1011	LT	Less than
1100	GT	Greater than
1101	LE	Less than or Equal
1110	AL o nada	Always/Incondicional

TABLE II
COMANDOS PARA LOS DISTINTOS TIPOS DE INSTRUCCIONES EN ARM

Comando (cmd)	Instruccion	Descripcion	Operación
0000	AND Rd, Rn, Scr2	Bitwise AND	Rd <- Rn & Scr2
0001	EOR Rd, Rn, Scr2	Bitwise XOR	Rd <- Rn ^Scr2
0010	SUB Rd, Rn, Scr2	Subtract	Rd <- Rn - Scr2
0100	ADD Rd, Rn, Scr2	Add	Rd <- Rn + Scr2
1010	CMP Rn, Scr2	Compare	Set flags <- Rn - Scr2
1100	ORR Rd, Rn, Scr2	Bitwise OR	Rd <- Rn $\|\|Scr2$
1111	MVN Rd, Scr2	Bitwise NOT	Rd <- ~Scr2

Ahora bien, para describir el resto de las partes de cada tipo de instrucción es necesario analizar de manera individual cada una de estas.

B. Instrucciones de procesamiento de datos.

Este es el formato mas común. El primer operando es un registro. El segundo operando puede ser un inmediato o un registro con un corrimiento opcional. En la Figura 1 se muestra el formato de esta instrucción.

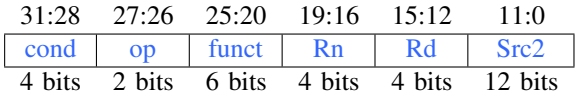

Fig. 1. Formato instrucciones procesamiento de datos

Nótese que la instrucción es de 32-bit y tiene 6 espacios: **cond**, **op**, **funct**, **Rn**, **Rd**, **Scr2**. Para instrucciones de procesamiento de datos **op = 00**. En la Figura 2 se muestra el formato del espacio **funct** y las 3 variaciones de **Scr2**.

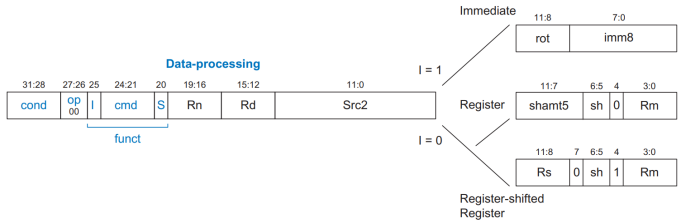

Fig. 2. Formato instrucciones de procesamiento de datos mostrando el espacio funct y Scr2.

El espacio **funct** tiene 3 divisiones: **I, cmd, S**, donde:

- **I:** El I-bit es 1 cuando Src2 es un inmediato. El I-bit es 0 cuando Src2 es un registro.
- **S:** E S-bit es un 1 cuando la instrucción settea las flags condicionales. El S-bit es un 0 cuando la instrucción no settea las flags condicionales. Por ejemplo, ADDS si settea the flags condicionales.
- **Cmd:** Indica la operación en específico a realizarse. Por ejemplo, para ADD el cmd es 4 (0100) y para SUB el cmd es 2 (0010). Un desglose mas detallado de algunos de los posibles códigos permitidos por ARM para el espacio cmd, se muestran en la Tabla II [1, p. 536].

Para el caso de **Scr2** este tiene 3 variaciones:

- **Scr2** puede ser un inmediato.
- **Scr2** puede ser un registro con un shift opcional dado por una constante. **Sh** codifica el tipo de shift a realizar. Los posibles shift se muestran en la Tabla III.
- **Scr2** puede ser un registro con un shift opcional dado por otro registro. Del mismo modo, **Sh** codifica el tipo de shift a realizar. Los posibles shift se muestran en la Tabla III.

TABLE III
CODIFICAN DEL ESPACIO SH

sh	Operation
00	Logical shift left
01	Logical shift right
10	Arithmetic shift right
11	Rotate Right

C. Instrucciones de memoria.

Estas instrucciones usan un formato similar al de las instrucciones de procesamiento de datos. Su primera similitud es que ambas tienen los 6 mismos espacios: **cond, op, funct, Rn, Rd, Scr2**. Este formato se muestra con mayor detalle en la Figura 3 [1, p. 333-334].

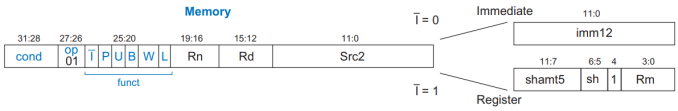

Fig. 3. Formato instrucciones de memoria.

Para las instrucciones de memoria, sus particularidades o diferencias respecto a los otros tipos son:

- Usan un espacio para **funct** dividido de diferente forma.
- Tiene dos variaciones para **Src2**.
- Usan el valor de op = 01.

Para el caso de los operandos que contiene la instruccion se tiene que:

- **Rn** contiene el registro base (Dirección de la memoria).
- **Src2** almacena el offset que puede ser un inmeadito de 12-bit sin signo o un registro **Rm** con un Shift opcional dado por una constante **shamt5**.
- **Rd** es el registro que almacenará lo que este en la memoria(para el LDR) o es el registro que contiene lo que se quiere almacenar en la memoria (para el STR).

En al caso del espacio de **funct** este se compone de 6 bits de control: **I', P, U, B, W** y **L**, donde:

- **I':** Este bit determina si el offset es un inmediato o un registro, esto de acuerdo con la Tabla IV.
- **U:** Este bit determina si el offset debe de ser sumado o restado, esto de acuerdo con la IV.
- **p y W:** Estos bits indican el modo de indexado. El P significa pre-index y el W write-back. Esto de acuerdo con la Tabla V.
- **L:** Especifica si la instrucción de memoria es un Load(L = 1) o si es un Store (L=0). Esto según la Tabla VI.
- **B:** Especifica si el valor cargado/guardado será un Byte (B=1) o un Word(B=0). Esto según la Tabla VI.

TABLE IV
TIPOS DE OFFSET

Bit	Significado	
	I'	U
0	Offset con inmediato en Src2	Restar el offset desde la base
1	Offset con registro en Src2	Sumar el offset desde la base

TABLE V
MODOS DE INDEXADO

P	W	Index Mode
0	0	Post-Index
0	1	No soportado
1	0	Offset
1	1	Pre-Index

TABLE VI
VALORES DE L Y B PARA DISTINTAS INSTRUCCIONES DE MEMORIA.

L	B	Instrucción
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

D. Instrucciones tipo branch.

En el caso de las instrucciones tipo Branch, su formato se muestra en la Figura 4.

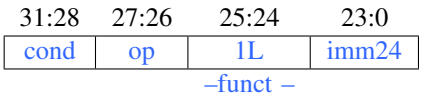

Fig. 4. Formato instrucciones tipo branch

Al igual que las instrucciones de memoria y las instrucciones de procesamiento de datos, las instrucionse tipo Branch empiezan con 4-bits para la condicion(**cond**) y 2-bit para la operacion (**op**) cuyo valor es **op = 10**. En este caso el espacio para **funct** es solo de 2 bit, donde su MSB siempre es uno para branches y el LSB(L) indica el tipo de branch: 1 para BL y 0 para B. Los 24-bit del inmeadito representan a una constante con signo y esta es usada para especificar la direccion relativa a PC+8 de una instrucción.

II. DESARROLLO

Nuestro principal objetivo fue el de desarrollar un decodificador de caracteres para su posterior despliegue gráfico. Para lograr esto recurrimos a la implementación de un procesador (una micro-arquitectura) y la reutilización de un controlador VGA. El diagrama que describe la idea anterior se muestra en la Figura 5.

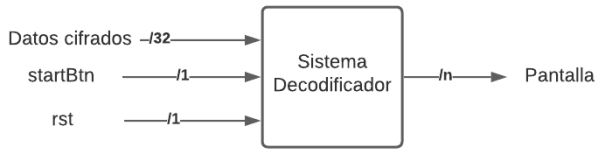

Fig. 5. Decodificador de caracteres

Sumado a esto, si contemplamos el hecho de que nuestro sistema decodificador mostrado en la Figura 5 está compuesto por el procesador y el controlador VGA, podemos plantear el diagrama mostrado en la Figura 6.

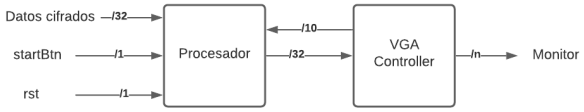

Fig. 6. Procesador junto con controlador VGA.

Ahora bien, para iniciar con el desarrollo de nuestro decodificador, se tomó primeramente la etapa del procesador y posterior a ello, se realizó lo asociado al controlador VGA.

A. Procesador - Micro arquitectura

A pesar de que el desarrollo de una micro-arquitectura parece una tarea difícil de entender y de llevar a cabo, en realidad no lo es, pues incluso su implementación no se escapa de los conceptos e ideas manejadas en la lógica de tipo combinacional y secuencial.
Visto de un modo muy superficial, un procesador se conforma mediante la combinación de distintos bloques de ambas lógicas anteriormente mencionadas. Para simplificar aun mas el desarrollo del procesador, su implementación se llevó acabo dividiendo lo en dos grandes partes:

- La ruta de los datos (Datapath).
- El controlador o unidad de control (CU).

La manera en la que estas dos partes trabajan conjuntamente se describe como sigue:
La ruta de datos opera en función de la instrucción en ejecución. La unidad de control recibe la instrucción actual desde la ruta de los datos y le dice a esta como ejecutar dicha instrucción. El ruta de los datos contiene elementos de hardware tales como como memorias, registros, la ALU, un extensor de signo y multiplexores. La unidad de control

produce señales de control tales como enables o selectores de multiplexores para habilitar los elementos de hardware anteriormente mencionados y para dejar pasar ciertos datos según sea necesario. El diagrama que muestra la interacción entre estas dos partes se muestra en la Figura 7

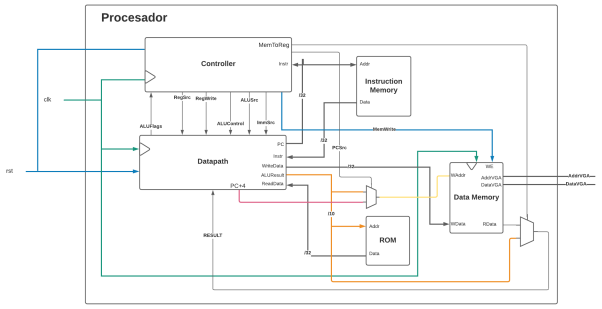

Fig. 7. Procesador formado por la ruta de los datos y la unidad de control.

1) Ruta de los datos - Datapath.

Como se puede apreciar de la imagen anterior, se hace uso de mas de una memoria. Esto se da pues el procesador que fue implementado es un procesador uni-ciclo y como existen instrucciones que acceden a la memoria ya sea para escribirla (en el caso del STR) o bien para leerla(en un LDR o al hacer fetch de la instrucción a ejecutarse) se tuvieron que separar para lograr así dichos procesos. Por esta razón, cada una de las memorias usadas tienen ciertas diferencias en su modo de implementación.

Para el caso de la **memoria de instrucciones** y de la **memoria ROM**(quien contenía los datos insumo del programa) estas fueron implementadas haciendo uso de lógica combinacional. Su interfaz fue diseñada con un único puerto de entrada de 32-bit de ancho, desde el cual se apuntada a la dirección de memoria cuyo contenido se quería acceder. De manera **simultanea**, desde el puerto de salida de 32-bit se obtenía el contenido de la dirección de memoria a la que se estaba apuntando.

Para el caso de la **memoria de datos** la manera en la que se leía su contenido era idéntica a como se hacia en las otras dos memorias ya mencionadas. La diferencia radica en que la memoria de datos también poseía un puerto para la señal de reloj, otro para la señal de enable de escritura y otro para pasar el dato que se quería escribir. Todos estos nuevos puertos presentes en la interfaz de esta memoria, fueron necesarios para soportar la escritura de los datos en las instrucciones de tipo STR. Es importante resaltar que debido a que se tenia un único ciclo de reloj para ejecutar las instrucciones, toda escritura de datos se llevó acabo en los flancos negativos del ciclo de reloj en ejecución.

Otro de los elementos de estado que fue desarrollado fue el banco de registros. Este se muestra junto con otros bloques de hardware combinacional en la Figura 8.

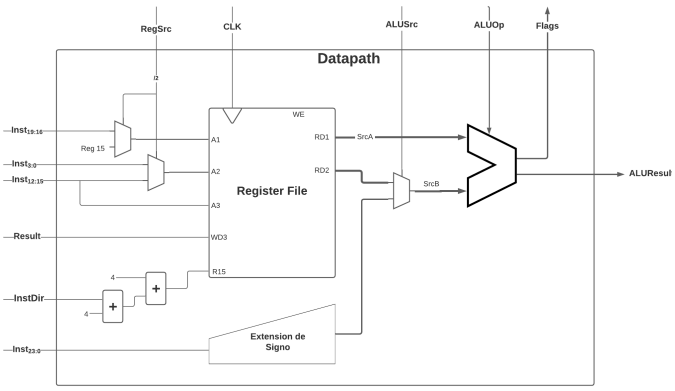

Fig. 8. Datapath

De este banco de registros se tiene la posibilidad tanto de escribir como de leer. Para el caso de la lectura se hace como en las memorias, con la diferencia de que se tienen dos puertos A_1 y A_2 para apuntar a dos direcciones y poder así leer desde RD_1 y RD_2 simultáneamente. Para el caso de la escritura se hace tal cual se hace en la memoria de datos.

Uno de los últimos módulos que se implementaron fue el de el extensor de signo donde este se encargaba de tomar los distintos inmediatos provenientes desde la instrucción en ejecución y extender su bit mas significativo hasta formar un nuevo numero de 32-bit.

Además de estos módulos anteriormente mencionados, también se diseñó una ALU con suporte para instrucciones de **Suma, Resta, XOR** y **NOT** con sus respectivas Flags. Tambien se diseñaron dos sumadores para calcular la siguiente instrucción a ejecutarse (el PC+4) y el PC+8 en caso de que se esté ejecutando una instrucción tipo branch.

2) Unidad de control.

La unidad de control genera las señales necesarias para la ruta de los datos basado en los espacios de **cond(Instr$_{31:28}$), op(Instr$_{27:26}$)** y **funct(Instr$_{25:20}$)** junto con el estado actual de las **flags**. La manera en la que la unidad de control se encuentra formada se muestra en la Figura 9.

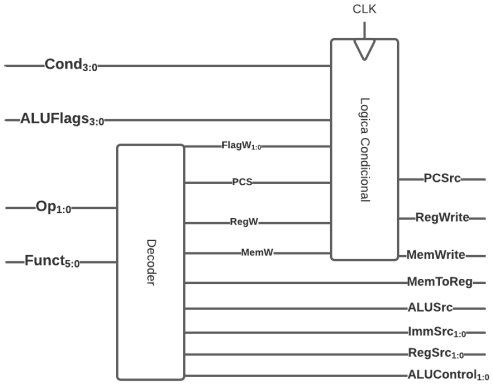

Fig. 9. Unidad de Control

Como se pudo observar la unidad de control esta conformada por las partes:

- El Decoder quien genera las señales de control basado en la instrucción.
- La lógica condicional quien mantiene el estado de las flags actualizo.

El **decoder** genera de manera directa las señales de **MemToReg, ALUSrc, ImmSrc, RegSrc** y la de **ALUControl**.
Para generar la señal de **MemToReg** se verifica si el tipo de instrucción es un LDR, si es asi, su valor será de uno, de lo contrario, su valor siempre será de cero. Esta señal conecta con el selector de un multiplexor.
Para generar la señal de **ImmSrc** se verifica el valor del campo de op de la instrucción.
Para generar la señal de **ALUSrc** y **RegSrc** se debe de verificar el campo de funct de la instrucción.
Para generar la señal de **ALUControl** se debe de verificar que se esté ejecutando una instrucción de procesamiento de datos y ademas se debe de leer el valor de cmd dentro de el campo de funct de la instrucción. Todas estas acciones mencionadas anteriormente se resumen en la Tabla VII.

TABLE VII
TABLA CON LÓGICA DE EL DECODER

Señal	**Op**				
	00		01		10
	Funct[5]	Funct[5]'	Funct[0]	Funct0]'	
RegSrc	00	00	00	10	01
ImmSrc	00	00	01	01	10
ALUSrc	1	0	1	1	1
MemToReg	0	0	1	1	0
RegW	1	1	1	0	0
MemW	0	0	0	1	0
Branch	0	0	0	0	1
ALUOp	1	1	0	0	0

Para la generación de las señales **PCSrc, RegWrite** y **MemWrite** se debe de hacer una verificación a dos pasos: primero el **decoder** genera las señales **PCS, RegW, MemW** y **flagW**, posterior a esto y como segundo paso, se verifica desde el bloque de **lógica condicional** si la instrucción en ejecución actualiza las flags, o bien, se ejecuta en función de alguna condición, generando una señal de salida de uno si las condiciones permiten la ejecución de la instrucción o un cero en caso contrario. Finalmente, es desde el bloque de **lógica condicional** desde donde se generan las señales de **PCSrc, RegWrite** y **MemWrite** contemplando los resultados obtenidos de los dos pasos anteriormente mencionados [1, p. 400]. Finalmente la manera en la que se diseña la lógica asociada a la lógica condicional es considerando el valor de los flags(NZCV) como entradas y a partir de estas se genera una única señal de salida, tal y como se muestra en la Tabla VIII.

TABLE VIII
TABLA DEL BLOQUE DE LÓGICA CONDICIONAL

cond	**Mnemonico**	**LogCondOut**
0000	EQ	Z
0001	NE	Z'
1010	GE	(N XOR Z)'
1011	LT	(N XOR Z)
1100	GT	Z'(N XOR Z)'
1101	LE	Z OR (N XOR Z)
1110	AL o nada	1

B. Controlador VGA

Como se ha planteado el sistema consiste en un procesador capaz de descifrar caracteres ASCII y representarlos por medio de un monitor. Para esta etapa de representación de datos se debe emplear el uso de un controlador VGA. Como bien se sabe VGA por sus siglas en inglés Video Graphic Array, es un estándar para visualización de vídeos en el que su resolución es de 480x640 píxeles [3]. La implementación de este controlador se realizó por medio del diseño observado en la figura 10.

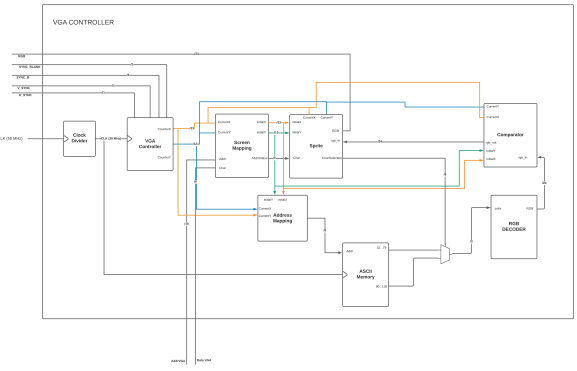

Fig. 10. Controlador VGA.

1) Señal de reloj

El controlador posee una entrada de reloj la cuál debe es calculada por medio de la ecuación 1. En donde $p = 800\frac{píxeles}{línea}$, $l = 500\frac{líneas}{pantalla}$ y $s = 60\frac{pantallas}{segundo}$.

$$pixelrate = p \cdot l \cdot s \tag{1}$$

Dando como resultado que la frecuencia de operación del estándar es de 25 MHz, como la FPGA empleada posee una señal de reloj interna se emplea la disponible en el pin AF14, al ser esta una señal de 50 MHz se procede a la elaboración del módulo clock divider el cuál se encarga de modificar el valor de frecuencia actual al deseado.

2) Señales de sincronización

Seguidamente se entra a una etapa de control por medio del submódulo VGA Controller el cuál se encarga de activar las señales de sincronización y el conteo de píxeles tanto verticales como horizontales, tomando como referencia las zonas visibles y los momentos de activación de las differentes señales como lo son V_SYNC, H_SYNC, $SYNC\_N$ y $SYNC\_BLANK$, tal como se muestra en la figura 11.

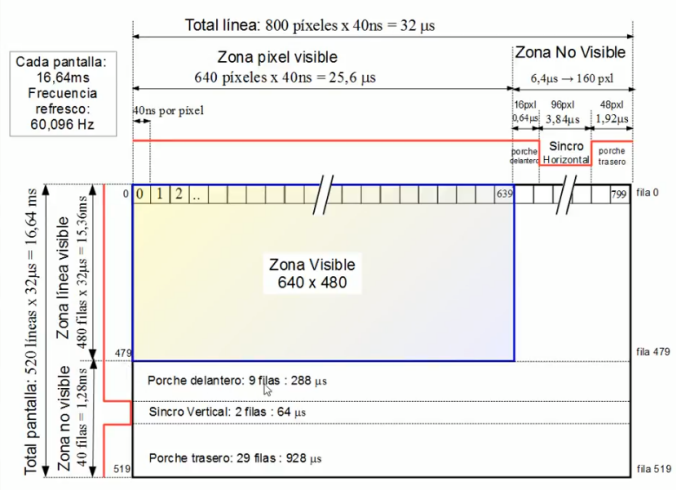

Fig. 11. Estructura lógica de la VGA.

3) Memoria ASCII

Para representar cada uno de los caracteres de ASCII estándar se procede a guardar en ROM cada uno de los caracteres para eso se diseñan por medio de imágenes de formato tipo PNG con una resolución de 32x32 píxeles. Estas imágenes son procesadas por medio de un método elaborado en Python, el cuál se encarga de convertir cada uno de los píxeles de las imágenes en datos de 3 bits, en los que 111 representan el color blanco y 000 el color negro dando como resultado la figura 12. Los datos resultantes del procesamiento

Fig. 12. Procesamiento de imágenes.

de las imágenes son almacenados en un archivo tipo MIF, para proceder a inicializar cada una de las memorias de los caracteres, siendo estos datos de un ancho de 3 bits y una profundidad de 1024 bits. Para visualizar el valor de la memoria seleccionada se emplea un MUX que por medio de la señal de selección la cuál consiste en el valor ASCII que se desea mostrar.

4) Mapeo de la memoria ASCII

Se realiza el mapeo de las memorias ASCII para saber que color mostrar según el píxel actual del contador. Como se sabe la dirección de memoria de los caracteres es de 10 bits. Para dicho mapeo se toma en cuenta que $id_x = píxelx_{inicial} - píxelx_{actual}$ y $id_y = píxely_{inicial} - píxely_{actual}$. Siendo la el valor mapeado los 5 LSB del valor de id_y concatenado de los 5 LSB de id_x.

5) Mapeo de pantalla

Como se muestra en la figura 13. Se toma la pantalla VGA y se divide el espectro visible en cuadrículas de 32 bits. En las que unas de ellas se les asignan las posiciones de memoria de la memoria de datos provenientes del resultado del operador.

	0	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480	512	544	576	608	640
0																					
32																					
64																					
96																					
128			0C	10	14	18	1C	20	24	28	2C	30	34	38	3C	40	44	48	4C		
160			50	54	58	5C	60	64	68	6C	70	74	78	7C	80	84	88	8C	90		
192			94	98	9C	A0	A4	A8	AC	B0	B4	B8	BC	C0	C4	C8	CC	D0	D4		
224			D8	DC	E0	E4	E8	EC	F0	F4	F8	FC									
256																					
288																					
320																					
352																					
384																					
416																					
448																					
480																					

Fig. 13. Mapeado de la pantalla.

III. Resultados

Para corroborar que cada uno de nuestros módulos funcionaran de acuerdo a lo esperado, se realizaron los respectivos TestBench de cada uno de estos. A continuación se muestran las formas de onda resultantes para cada caso.

A. Banco de Registro.

En este test, se realizan dos escrituras y dos lecturas. En la Figura 14 se muestra el resultado de la prueba.

Fig. 14. TestBench Banco de registros.

Observe del test que se inicia apuntando a los registros 3 y 6 para su escritura en el siguiente flanco negativo de reloj. Notese que para esto la señal de enable también fue habilitada. Los valores que fueron escritos en dichos registros corresponden a 7 y 9 respectivamente. Estos valores se aprecian al final del test, cuando mediante las variables de entrada **pointToAddress1** y **pointToAddress2** se apuntan a dichos registros.

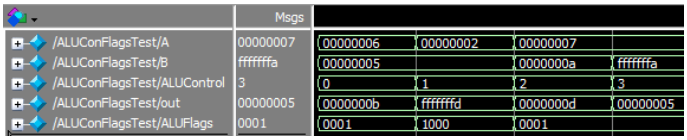

Fig. 15. TestBench ALU.

B. ALU.

En este test se realizan los cuatro tipos de instrucciones implementados en la ALU: Una suma, una resta, un xor y un not. En la Figura 15 se muestra el resultado de la prueba.

Observe del test que los operandos son:

- Suma: A = 32'h0000006 y B = 32'h0000005.
- Resta: A = 32'h0000002 y B = 32'h0000005.
- XOR: A = 32'h0000007 y B = 32'h000000A.
- NOT: B = 32'hFFFFFFFA.

C. Extensión de inmediato.

En este test se realizaron pruebas para la extensión de un inmediato proveniente de una instrucción tipo branch(lo cual implica la extensión y dos shift) y para la extensión de un inmediato proveniente de una instrucción de procesamiento de datos(DP). En la Figura 16 se muestra el resultado de la prueba.

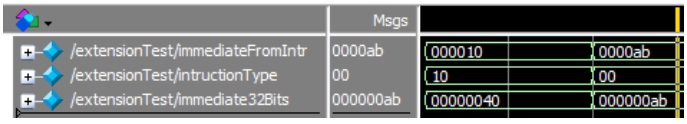

Fig. 16. TestBench Inmediato.

Observe del test que los inmediatos son:

- Para la instrucción tipo Branch: Imm = 24'h0000010.
- Para la instrucción DP: Imm = 24'h0000AB.

D. Unidad de control.

En este test se realizaron pruebas para instrucciones tipo Branch e instrucciones de procesamiento de datos. En la Figura 17 se muestra el resultado de la prueba.

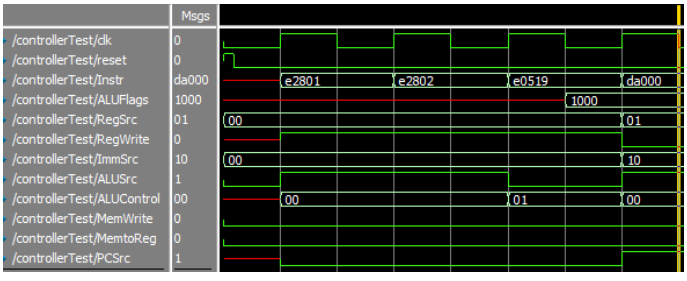

Fig. 17. TestBench Unidad de Control.

Observe que algunos de los códigos de la variable Instr corresponden a las instrucciones:

- Instr = 20'hE2801 -> ADD R1, R0, 2.
- Instr = 20'hE2802 -> ADD R2, R0, 5.
- Instr = 20'hE0519 -> SUBS R9, R1,R2
- Instr = 20'hDA000 -> BLE R2ESMAYOR

E. Procesador.

En este test se hizo la ejecución completa del siguiente código ensamblador:

```
SUB R0, R15,R15   ; R0 = 8 - 8 = 0
ADD R1, R0, #2    ; R1 = 0 + 2 = 2
ADD R2, R0, #5    ; R2 = 0 + 5 = 5
SUBS R3, R1, R2   ; R3 = 2 - 5 = -3 (NZCV = 1000)

BLT R2ESMAYOR
SUB R4,R1,R2              ; R4 = 5 - 2 = 3
EOR R7,R1,R4              ; R7 = 0101 XOR 0011 = 6 = 0110
ADD R6,R2,R7              ; R6 = 8 = 1000

B FIN
ADD R4,R1,R2              ; R4 = 7 = 0111 (R2ESMAYOR)
ADD R7,R4,#3              ; R7 = 7+3 = 10 = 1010
EOR R6,R1, R7             ; R6 = 0010 XOR 1010 = 1000 = 8

ADD R5, R4, R4            ; R5 = 14 (FIN)
```

Cuyo equivalente en lenguaje maquina corresponde al que se muestra en la Figura 18.

```
8'h00: data = 32'hE04F000F; // SUB R0, R15,R15
8'h04: data = 32'hE2801002; // ADD R1, R0, #2
8'h08: data = 32'hE2802005; // ADD R2, R0, #5
8'h0c: data = 32'hE0513002; // SUBS R3, R1, R2

8'h10: data = 32'hBA000003; // BLT R2ESMAYOR
8'h14: data = 32'hE0414002; // SUB R4,R1,R2
8'h18: data = 32'hE0217004; // EOR R7,R1,R4
8'h1c: data = 32'hE0826007; // ADD R6,R2,R7

8'h20: data = 32'hEA000002; // B FIN
8'h24: data = 32'hE0814002; // ADD R4,R1,R2
8'h28: data = 32'hE2847003; // ADD R7,R4,#3
8'h2c: data = 32'hE0216007; // EOR R6,R1, R7

8'h30: data = 32'hE0845004; // ADD R5, R4,R4
```

Fig. 18. Código maquina

Observe que el código maquina es el que está siendo asignado a la variable data. Además, notese que las instrucciones se encuentran cargadas en una memoria donde la instrucción inicial comienza en la posición 0x00, mientras que la posición de la ultima instrucción corresponde a 0x30.
Finalmente, en la Figura 19 se muestra el resultado de la prueba.

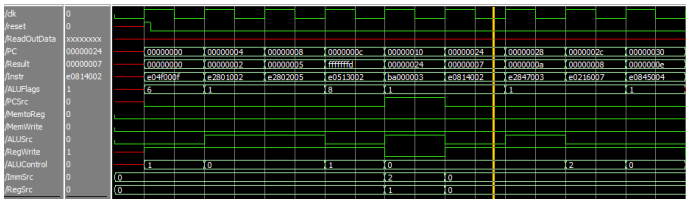

Fig. 19. Testbench Procesador

F. Monitor.

Finalmente, después de las pruebas exitosas de los distintos módulos del procesador, se comprueba el funcionamiento del

mismo en un monitor VGA. Para esto se aplica la operación ”not” al siguiente texto ”Proyecto final de Diseno de Sistemas Digitales” mediante un script de Python, posteriormente se toma el texto codificado y se ingresa al procesador para su decodificación, el cual se logra observar en la figura 20.

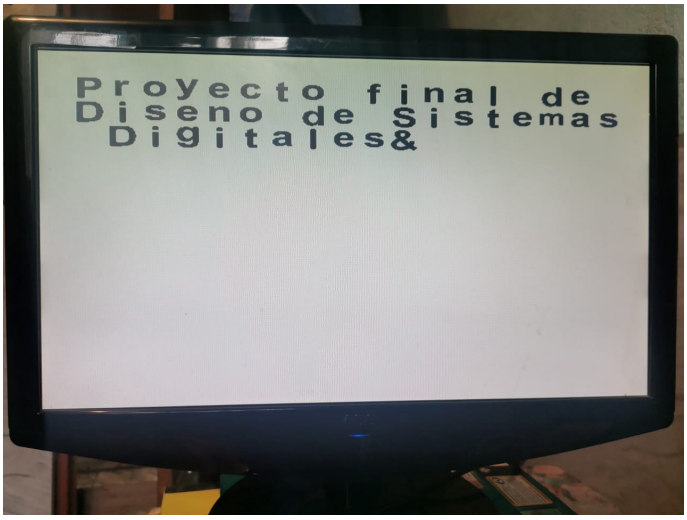

Fig. 20. Test Monitor

IV. Análisis

A. Banco de Registro.

Se sabe que en la lógica de tipo combinacional, todo cambio en las variables de entrada, afectará inmediatamente las variables de salida. Al implementarse la lectura del banco de registros con lógica combinacional, se esperaba que, al apuntar desde A_1 A_2 a la direcciones de dos de los registros se viese inmediatamente el contenido de estos en la salida. En efecto, tal y como se observa desde la Figura 14, en el momento en el que se apunta desde dichos puertos a las direcciones de los registros 3 y 6, se logra ver su contenido de manera inmediata, tal y como se esperaba.

B. ALU.

Para el caso del test de la ALU se realizaron las 4 posibles operaciones que esta soporta. Para corroborar sus resultados basta con realizar los mismos cálculos de manera manual y comparar. Para el caso de la suma se tiene que los operandos fueron:

$$A = 32'h0000006$$

$$B = 32'h0000005$$

Lo cual da como resultado:

$$C = 32'h000000B$$

Si se observa del test, este resultado coincide con el recién calculado. Para el caso de la resta se tiene que los operandos fueron:

$$A = 32'h0000002$$

$$B = 32'h0000005$$

Ahora bien, si se realiza esta operación en base diez, el resultado de la misma seria:

$$C = -3$$

Si representamos este numero a su equivalente en complemento a dos se tiene que:

$$C = 32'hFFFFFFFF = -1$$

$$C = 32'hFFFFFFFE = -2$$

$$C = 32'hFFFFFFFD = -3$$

Comparando el resultado anterior con el obtenido desde la simulación, vemos que en efecto ambos coinciden.

C. Extensión de inmediato.

De la prueba puede observarse que el resultado de aplicar una extensión al inmediato 24'h0000010 el cual proviene de una instrucción de tipo branch es el siguiente:

$$Extensión = 32'h00000040$$

Ahora bien, para corroborar el resultado anterior se tomó el inmeadiato 24'h0000010 y se aplicó el procedimiento de manera manual. Primeramente se convirtió a su equivalente en binario:

$$Imm = 24'b0000\_0000\_0000\_0001\_0000$$

Numero al cual al aplicarle dos veces shift a la derecha se obtuvo el valor de:

$$Imm = 24'b00\_0000\_0000\_0001\_000000$$

Finalmente, se le aplicó la extensión del cero 8 veces:

$$Extensión = 32'b0000\_0000\_0000\_0000\_0000\_0100\_0000$$

$$Extensión = 32'h00000040$$

Comprobando así que el resultado manual y el de la simulación coinciden.

D. Unidad de control.

El test de la unidad de control genera las señales de salida que deben de pasarse a la ruta de los datos. Si se observa con detalle, se sabe que de nuestra arquitectura toda instrucción de tipo branch que si deba de ejecutarse, debe de colocar la señal de PCSrc en uno para así permitir que la instrucción a asignarse en el PC provenga desde la instrucción de salto y no desde el PC+4. Podemos observar de la Figura 17 que en efecto esta señal es igual a uno. Además, también se puede observar que la fuente del inmediato es igual a 10, lo cual corresponde al código op de la instrucciones tipo branch. Otras señales a las que podemos prestarles atención son a las de enable para escritura tanto en el registro como en la memoria, donde se aprecia que ambas son iguales a cero, tal y como debe de ser.

E. Procesador.

Como se muestra en la figura 18, cada una de las instrucciones en lenguaje maquina fueron cargadas a partir de la dirección de memoria 0x00. Si se observan los principales detalles de este código, se puede notar que: R1 = 2 y R2 = 5. A partir de esto, en la 4ta linea de código podemos observar que se realiza la operación de resta (SUBS) que actualiza el valor de las flags (NZCV = 1000). Posteriormente en la 5ta linea de código se ejecuta un branch condicional el cual si se cumple. Por esta razón, el flujo del código debe de pasar de ejecutar la instrucción en memoria 0x10 y debe de saltar hasta la dirección de memoria 0x24, tal y como ocurre en el código en el test.

V. Conclusiones

- El empleo de buenas prácticas y metodologías estandarizadas facilitan el proceso de diseño.
- El uso de las simulaciones permiten depurar las soluciones planteadas de manera óptima.
- El empleo de micro arquitecturas para la resolución de problemas permite un control total de los sistemas, beneficiando en aspectos de optimización de recursos.
- El diseño de sistemas digitales requiere de distintas partes, apoyarse de herramientas que faciliten realizar las mismas es de gran utilidad.

Referencias

[1] Sarah Harris y David Harris. Digital design and computer architecture: arm edition. Morgan Kaufmann, 2015

[2] Patterson Hennesy. Computer Organization and Design. The Hardware/-software Interface. arm edition. Morgan Kaufmann, 2017

[3] Pong P. Chu. FPGA prototyping by verilog examples. Xilinx Spartan -3 Version. Wiley Sons, 2008