Instituto Tecnológico de Buenos Aires

22.15 - Electrónica V

Trabajo Práctico 2

Grupo 2

Chiocci, Ramiro Antonio	45132
Figueroa, Maximiliano Antonio	56406
Laguinge, Juan	57430
Lin, Benjamín Carlos	57242
Scala, Tobias	55391

Profesores

Rodriguez, Andrés

Wundes, Pablo

Contents

1 Introducción

En el presente trabajo práctico se ha desarrollado y diseñado un procesador cuya arquitectura se encuentra caracterizado como SISD (single instruction, single data). En otras palabras, el presente procesador se encuentra inspirado por la arquitectura básica propuesta por el matemático Von Neumann. El mismo cuenta con una pipeline de 5 etapas. Dichas etapas son: Fetch, Decode, Execute, Operand y Retire.

Este procesador es denominado EV21 y cuenta con 51 instrucciones los cuales se encuentran detallados en el apéndice junto con las microinstrucciones. Estas instrucciones cuentan con opcode expadido ya que no todas las mismas tienen una misma cantidad de bits asignadas para identificarse (opcode). El mismo es capaz de ejecutar instrucciones matemáticas como multiplicar, dividr y módulo. También ejecuta instrucciones de salto tanto condicionales como no condicionales.

Cabe mencionar que el presente procesador no cuenta con predictores. Por lo que, cuando hay una instrucción de salto condicional, se espera el resultado proveniente del CCR de la ALU para poder continuar. Dicho registro (CCR), contiene los siguientes flags: Negative, Zero, Overflow y Carry.

El procesador EV21 funciona con solo 4 clocks de 1 MHz cada uno. Cada bloque de la CPU ha sido asignado a uno de estos clocks de forma cuidadosa para que el mismo funcione correctamente evitando la pérdida de datos por el motivo de escribir antes de leer y evitando resultados erroneos por el motivos de leer antes de escribir. En las próximas secciones se detallan las operaciones que se realizan en los flancos de cada clock.

Para cargar un programa al procesador, el mismo debe estar programado en lenguaje de máquina debido a que no se cuenta con un nivel de ensamblaje de código (assembler). Este procesador cuenta con 35 registros los cuales se encuentran dentro del Register Bank. De los cuales, 28 son registros de propósito general, 2 son de entrada, 2 son de salida, 2 son auxiliares y el último es el working register el cual será utilizado para leer y escrbir a memoria. Si bien la cátedra ha provisto de un enunciado para este trabajo, el resultado tiene varios cambios con respecto al mismo. Sin embargo se mantiene una referencia constante en el diseño hacia el enunciado.

2 Clocks

El presente procesador diseñado cuenta con 4 clocks para su funcionamiento. Los mismos derivan de un clock genérico de 50 MHz el cual ingresa a un PLL donde su frecuencia, duty cycle y fase es modificado para generar distintas salidas. Todos los clocks que salen del PLL tienen una frecuencia de 1 MHz. Cada uno acciona múltiples bloques pero no comparten un bloque de destino. En la siguiente figura se muestran las señales de cada clock durante un período.

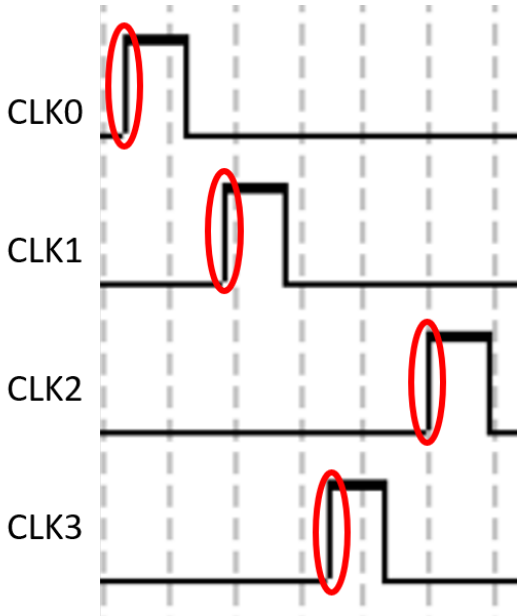

Figure 1: Señales de clock que utiliza la CPU EV21.

A continuación se detallan las distintas operaciones que se llevan a cabo en los flancos ascendentes de cada clock.

- Flanco ascendente del CLK0: Con este flanco de clock se avanza una etapa en el pipeline. A su vez, los buses A y B son cargados desde el Register Bank con el valor del registro indicado en el flip flop que contiene el MIR (etapa 3 de la pipeline).
- Flanco ascendente del CLK1: Con este flanco de clock el bloque UC (encargado de relaciones de dependencias) analiza la dependencia de la nueva instrucción con los demás y, en caso afirmativo, se encarga de pausar el flujo de el pipeline. A su vez, el valor cargado en el bus C ingresa al Register Bank escribiéndose en el registro indicado en el flip-flop que contiene el MIR (etapa 5 de la pipeline).
- Flanco ascendente del CLK2: Con este flanco de clock el bloque de memoria "PROGRAM" (donde se encuentra la secuencia de instrucciones del programa) es accionado para su lectura, obteniéndose así la próxima instrucción. A su vez, en este flanco, el valor cargado tanto en el bus A como B son guardados en el latch para luego poder acceder a la ALU.
- Flanco ascendente del CLK3: Con este flanco de clock el PC se actualiza sumando 1 o saltando a la posición de memoria que se indica en la instrucción. A su vez, el bloque de memoria "DATA" (donde se encuentran los datos) es accionado tanto para su lectura y escritura en la dirección especificada en la instrucción. También, en este flanco, el valor cargado en el BUS C es guardado en el latch para luego poder acceder al Register Bank.

3 Program Counter (PC)

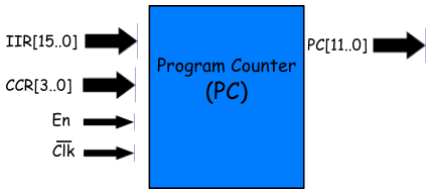

Figure 2: Módulo y señales de entrada y salida

La función fundamental de este módulo es controlar el puntero de direccionamiento hacia la memoria donde se almacenan las microinstrucciones del programa a ejecutar en la arquitectura presentada.

- Detecta las instrucciones de salto JMP mediante los primeros bits IIR[12..9] y modifica la posición del contador.
- Detecta la instrucción de salto si el resultado de la ALU es cero, negativo o ante la existencia de bit de "carry". Verificando la señal de CCR proveniente de la ALU, manipulamos la condición de detención del módulo (HOLD).
- Controla el acceso a sub-rutina mediante el comando BSR, a lo cual graba la posición del puntero en un registro temporal y, modifica el puntero actual con la nueva dirección que acompaña al comando.
- Detecta los retornos de subrutina mediante el monitoreo de los bits de la microinstrucción IIR[8..5]. En el caso de que se genere una operación de RET, el contador se carga con el valor previamente guardado al efectuar el salto.

Cuando tenemos comandos que pueden generar conflictos de dependencia (RAW, WAR y WAW), el módulo verifica la señal de ENABLED que es manipulado por el bloque UC de la arquitectura. Este procedimiento evita la modificación del puntero hasta que el conflicto no este resuelto.

4 Micro-Instruction Pipeline

Esta etapa tiene como objetivo el manejo correcto de las instrucciones y propagaciones de señales de control al resto de los módulos. Para ello se hace uso de los MIR (Micro-Instruction Register) y otros bloques de control para el manejo del pipeline durante la ejecución de instrucciones.

4.1 MIR

Este bloque se encarga de retener la información necesaria de cada bloque en distintos flip-flops del tipo D:

- ALUC: el código de la operación aritmética que puede involucrar la modificación del estado del Carry.
- SH: maneja el Shifter del la unidad aritmética.
- KMx: selector de datos.
- MR: estado de activación de lectura en memoria.
- MW: estado de activación de escritura en memoria.
- Bus A, B, C: bus que selecciona el registro o puertos a utilizar para la operación.
- Type: indica el tipo de instrucción.

4.2 Bloques de Control

Con el objetivo de manipular el flujo del pipeline es necesario bloques de control, que son los mediadores durante las ejecuciones.

- MIR Reset: vacía el tipo de instrucción e informa que los datos siguientes en el pipeline son irrelevantes.
- MIR Dependency: verifica si hay relación de dependencia entre las operaciones, específicamente un WAR (Write After Read) o un WAW (Write After Write).
- MIR Delay: en caso de haber dependencia, se pausa el ciclo de ejecución para que termine de realizar la tarea, evitando errores durante la operación.

5 Gestor de Datos

Se gestionan los datos utilizando una memoria integrada que provee el Quartus y un Banco de Registro. Cada registro del banco es de 16 bits, y el banco se integra con 28 registros de propósito general, 2 puertos de entrada, 2 puertos de salida y 2 auxiliares inaccesibles por el programador. Además de los mencionados, se le debe agregar el 'Working Register' (Registro de Trabajo), es decir que en total son 35 registros pero solo 33 registros accesibles por el programador.

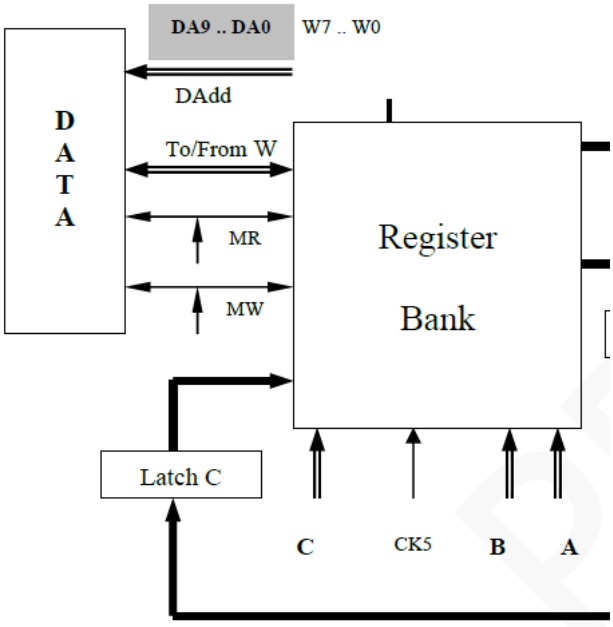

Figure 3: Banco de Registros

5.1 Memoria Integrada

Para la aplicación, se utilizo la memoria integrada RAM: 1-Port de la figura 4. Para guardar en memoria el dato de la entrada, se debe establecer una señal de enable en alto por 'wren', o sea el 'MW' (Memory Write) del codigo MIR, y la dirección del registro a transcribir. Intuitivamente, se lee la dirección de memoria cuando 'wren' es LOW.

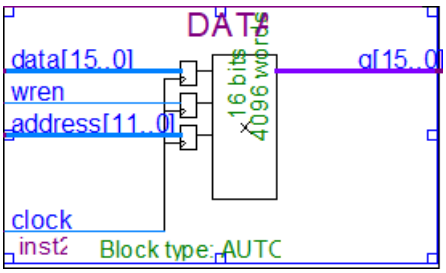

Figure 4: Memoria Integrada

5.2 Banco de Registros

Se transcribió en verilog el banco de registros para simplificar el manejo de flip-flops y las entradas y salidas del banco de registros.

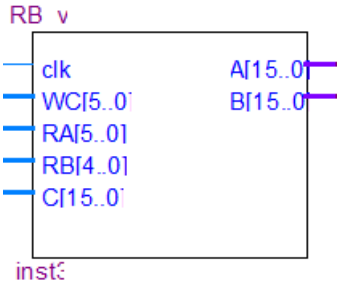

Figure 5: Register Bank

En este bloque el bus 'WC' selecciona donde se guarda el contenido del bus C en flancos ascendentes del CLK, es decir que se almacena en el registro numero 'WC' la información de 'C'. Por ejemplo: si WC = 21 → se retiene el dato en el registro 21 de propósito general, o si WC = 28 → se retiene el dato en el primer puerto de entrada. Existe un caso partícula en donde no se accede un registro accesible, este caso se da cuando WC = 35, ya que esta pertenece a uno de los 2 registros auxiliares que el programador no puede comunicarse, sin embargo se conserva el dato de C en ella temporalmente.

Por otro lado, el bloque selecciona los dos registros de salida 'A' y 'B' mediante los selectores 'RA' y 'RB'. Cabe destacar que para nuestro diseño, el selector RA es el encargado de acceder a todos los 33 registros accesibles, sin embargo el selector RB no es capaz de acceder al 'working register'.

5.3 Etapa de Salida

Para que los datos sean relevantes con la operación en ejecución se coloco a la salida de ambos un MUX para coordinar. A la salida de 'A', se selecciona el contenido entrante a la ALU dependiendo si la tarea necesita un 'MR' (Memory Read). A su vez, la salida 'B' puede ser una constante dada por el programa, por ello en caso de que sea así se enviara desde el código MIR la señal 'KMx' en alto y con una máscara filtra la constante que se le ha dado.

6 ALU

La unidad lógica aritmética es donde se van a realizar todas las operaciones básicas de nuestra CPU tales como las operaciones lógicas AND, OR, XOR y NOT además de las operaciones aritméticas suma, resta, división, modulo y multiplicación. También se realizan operaciones con los flags, como por ejemplo el carry en 1 o en 0 según corresponda.
Todas las operaciones son realizadas con entradas y salidas de 16 bits. A continuación un diagrama de la implementación:

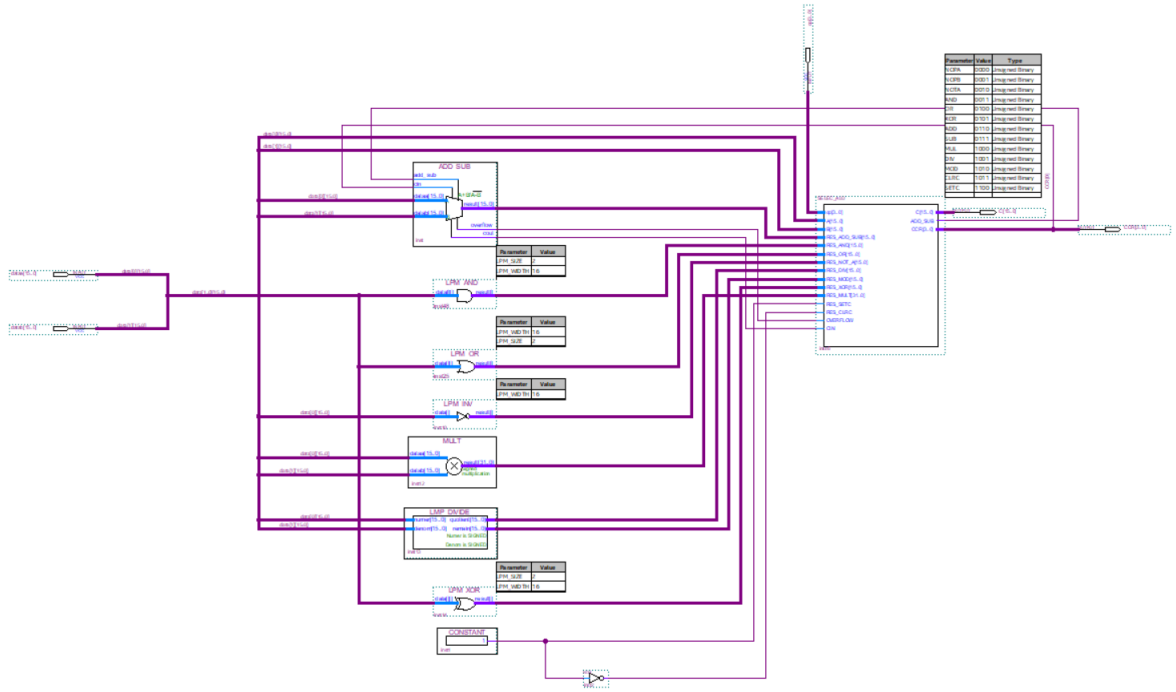

Figure 6: Diagrama de bloques de la ALU

Podemos dividir la misma en 2 sub-bloques: uno donde se realizan todas las operaciones, y otro donde se selecciona la operación a realizar.

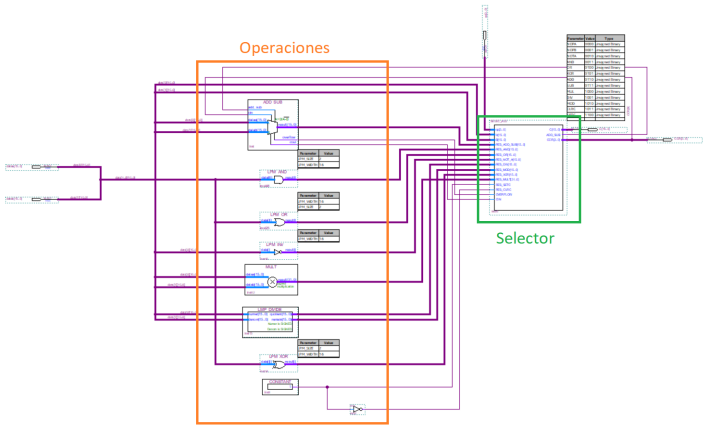

Figure 7: Diagrama de bloques de la ALU

6.1 Bloque de Operaciones

Las operaciones realizadas son todas las mencionadas anteriormente donde cada una fue aplicada con la utilización de bloques predefinidos por el programa Quartus utilizado para realizar el código de la FPGA.
Las operaciones de suma y restas son las únicas que cambian los flags de carry y overflow mientras que los flags de zero y negative son cambiados por todas las operaciones disponibles. Existe también la posibilidad de cambiar el flag de carry con una operación directa, sin pasar por una suma o una resta. Esto se logró mediante un bloque constate que transmite un bit y su salida y salida negada se conectaron al selector.
Cabe agregar que la operación multiplicación a diferencia de las demás tiene una salida de 32 bits en vez de 16 bits la cual es enviada al selector para realizar los ajustes necesarios para recortar el resultado a 16 bits.

6.2 Bloque del Selector

Para finalmente obtener el resultado de la operación deseada, se realizó el selector que recibe los resultados dados de todas las operaciones y en caso de ser necesario ajusta su salida a 16; además de realizar los cambios necesarios en los flags.
A continuación se muestra un cuadro con el opcode de 4 bits necesario para elegir la operación, así como la operación que se realiza acorde al mismo:

Opcode	Operación
0000	No operation A, devuelve el registro A
0001	No operation B, devuelve el registro B
0010	Negación del registro A
0011	AND lógico
0100	OR lógico
0101	XOR lógico
0110	Suma del registro A con B
0111	Resta del registro A con B
1000	Multiplicación del registro A con B
1001	División del registro A con B
1010	Modulo del registro A con B
1011	Poner en 0 el flag del carry
1100	Poner en 1 el flag del carry

Se puede apreciar que quedaron dos valores sin utilizar, lo que da lugar a posibles mejoras futuras y ampliar la funcionalidad del bloque en general con mayor cantidad de funciones.

La implementación del mismo fue realizada por medio de código Verilog el cual se encuentra descripto con más detalles en el apéndice. Cabe agregar que aunque el selector recibe los 32 bits de la multiplicación solo tenemos como salida de dicha operación 16 bits, limitando a nuestro usuario a trabajar solo con multiplicaciones que no superas los 16 bits de representación.

7 Memorias

En el diseño de este procesador se encuentran 2 tipos de memorias distintas e independientes: una ROM y una RAM. Ambas son bloques propios de Quartus que cumplen la funcionalidad de poder guardar datos y leerlos.

La memoria ROM es utilizada para poder cargarle un programa en lenguaje de máquina antes de que la CPU esté en funcionamiento. Luego la CPU leerá instrucción por instrucción a través de esta memoria. Es por esto que esta memoria se llama "PROGRAM".

La memoria RAM es utilizada para poder leer y escribir datos a la misma. Asimismo, es posible cargarle datos como estado inicial antes de que la CPU esté en funcionamiento. Es por esto que esta memoria se llama "DATA". En la siguiente figura se muestra la conexión de esta memoria con el data path.

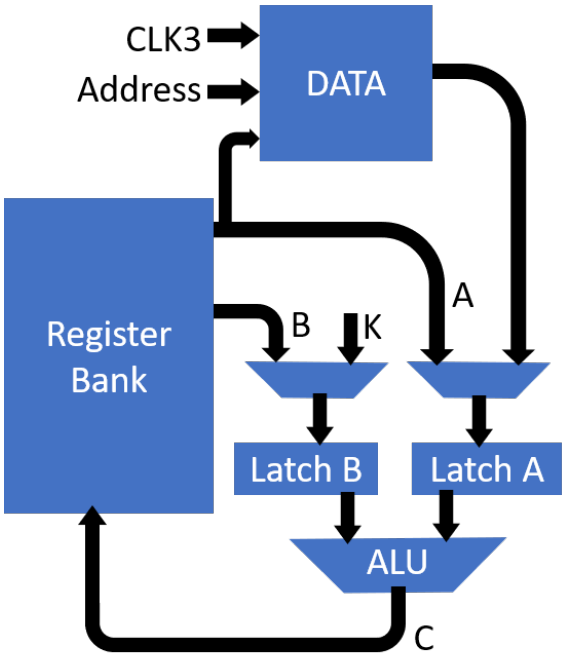

Figure 8: Conexionado de la memoria RAM con el data path.

Como se puede ver, la memoria no es un bloque independiente al flujo de la pipeline, sino que forma parte del mismo respetándolo. A continuación se explican los pasos cuando se quiere leer y escribir de la memoria "DATA".

- Lectura de la memoria "DATA": cuando se quiere leer de la memoria, primero se carga el bus Address con la posición de memoria correspondiente (el cual se extrae de la instrucción). Luego, en el flanco ascendente del clock 3, la memoria carga su bus de salida llevando el valor al MUX. Dicho MUX dejará pasar el valor proveniente de la memoria al latch. A partir de este momento, el valor ingresa al data path hasta que finalmente es guardado en el working register dentro del Register Bank.
- Escritura a la memoria "DATA": cuando se quiere escribir de la memoria, primero se cargan el bus Adress y los buses A y B. Luego, en el flanco ascendente del clock 3, la memoria escribe el valor que está en el bus A en la posición de memoria especificada en la instrucción.

8 Unidad Lógica para Relación de Dependencia

El procesador EV21 cuenta con un bloque de unidad lógica (llamada "UC") el cual tiene múltiples entradas recibiendo así información de distintas etapas de el pipeline. El mismo analiza la dependencia que hay entre la nueva instrucción y las siguientes. Dependiendo del caso, esta unidad pondrá en pausa el flujo de la pipeline o no. Por ejemplo, si la nueva instrucción necesita leer un registro, el cual está por ser escrito por otra instrucción, el bloque "UC" detiene a la instrucción nueva hasta que la otra instrucción termine su tarea escribiendo el resultado en el registro mencionado.

Ya que el bloque "UC" tiene la posibilidad de detener el flujo de el pipeline, se decidió ejecutar ciertas instrucciones de manera real. Se sabe que instrucciones como lectura a memoria, multiplicar, dividir y modulo tardan más de un ciclo de clock en ejecutarse por completo. Por lo que se optó por fijar un tiempo de espera de 3 ciclos de clock cuando una de las instrucciones mencionadas se está ejecutando. El programa en lenguaje Verilog del bloque UC se encuentra en el apéndice.

9 Apéndice

9.1 Set de Instrucciones

1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	JMP X	PC = X
1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	JZE X	IF CCR[Z] = 1, THEN PC = X
1	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	JNE X	IF CCR[N] = 1, THEN PC = X
1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	JCY X	IF CCR[C] = 1, THEN PC = X
1	1	0	0	y	y	y	y	y	y	y	y	y	y	y	y	MER W,Y	W = M[Y]
1	1	0	1	y	y	y	y	y	y	y	y	y	y	y	y	MEW Y,W	M[Y] = W
1	1	1	0	s	s	s	s	s	s	s	s	s	s	s	s	BSR S	SAVE PC, PC = PC + S
0	1	0	0	0	0	i	i	i	i	i	j	j	j	j	j	EQR Ri,Rj	Ri = Rj
0	1	0	0	0	0	1	1	1	1	i	j	j	j	j	j	EQR POi,Rj	POi = Rj
0	1	0	0	0	0	i	i	I	I	i	1	1	1	0	j	EQR Ri, PIj	Ri = PIj
0	1	0	0	0	0	1	1	1	1	i	1	1	1	0	j	EQR POi,PIj	POi = PIj
0	1	0	0	0	1	i	i	i	i	i	0	0	0	0	0	EQR Ri,W	Ri = W
0	1	0	0	0	1	1	1	1	1	i	0	0	0	0	0	EQR POi,W	POi = W
0	1	0	0	1	0	i	i	i	i	i	J	j	j	j	J	ANR Ri,Rj	Ri = W & Rj
0	1	0	0	1	1	i	i	i	i	i	J	j	j	j	J	ORR Ri,Rj	Ri = W | Rj
0	1	0	1	0	0	i	i	i	i	i	J	j	j	j	J	XOR Ri,Rj	Ri = W ^ Rj
0	1	0	1	0	1	i	i	i	i	i	J	j	j	j	J	ADR Ri,Rj	Ri = W + Rj + CY
0	1	0	1	1	0	i	i	i	i	i	J	j	j	j	J	SUR Ri,Rj	Ri = W - Rj - CY
0	1	0	1	1	1	i	i	i	i	i	J	j	j	j	J	MUR Ri,Rj	Ri = W * Rj
0	1	1	0	0	0	i	i	i	i	i	J	j	j	j	J	DIR Ri,Rj	Ri = W / Rj
0	1	1	0	0	1	i	i	i	i	i	j	j	j	j	J	MOR Ri,Rj	Ri = W % Rj
0	1	1	0	1	0	i	i	i	i	i	0	0	0	0	0	SLR Ri,W	Ri = W << 1
0	1	1	0	1	1	i	i	i	i	i	0	0	0	0	0	SRR Ri,W	Ri = W >> 1
0	0	1	0	0	0	0	k	k	k	k	k	k	k	k	k	EQK W,K	W = K
0	0	1	0	0	0	1	K	k	k	k	k	k	k	k	k	ANK W,K	W = W & K
0	0	1	0	0	1	0	k	k	k	k	k	K	k	k	k	ORK W,K	W = W | K
0	0	1	0	0	1	1	k	k	k	k	k	K	k	k	k	XOK W,K	W = W ^ K
0	0	1	0	1	0	0	k	k	k	k	k	K	k	k	k	ADK W,K	W = W + K + CY
0	0	1	0	1	0	1	K	K	k	k	k	K	k	k	k	SUK W,K	W = W - K - CY
0	0	1	0	1	1	0	K	K	k	k	k	K	k	k	k	MUK W,K	W = W * K
0	0	1	0	1	1	1	K	k	k	k	k	K	k	k	k	DIK W,K	W = W / K
0	0	1	1	0	0	0	K	k	k	k	k	K	k	k	k	MOK W,K	W = W % K
0	0	0	1	0	0	0	0	0	0	0	J	J	j	j	j	EQW W,Rj	W = Rj
0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	j	EQW W,PIj	W = PIj
0	0	0	1	0	0	0	1	0	0	0	j	j	j	j	j	ANW W,Rj	W = W & Rj
0	0	0	1	0	0	1	0	0	0	0	j	j	j	j	j	ORW W,Rj	W = W | Rj
0	0	0	1	0	0	1	1	0	0	0	j	j	j	j	j	XOW W,Rj	W = W ^ Rj
0	0	0	1	0	1	0	0	0	0	0	j	j	j	j	j	ADW W,Rj	W = W + Rj + CY
0	0	0	1	0	1	0	1	0	0	0	j	j	j	j	j	SUW W,Rj	W = W - Rj - CY
0	0	0	1	0	1	1	0	0	0	0	j	j	j	j	j	MUW W,Rj	W = W * Rj
0	0	0	1	0	1	1	1	0	0	0	j	j	j	j	j	DIW W,Rj	W = W / Rj
0	0	0	1	1	0	0	0	0	0	0	j	j	j	j	J	MOW W,Rj	W = W % Rj
0	0	0	1	1	0	0	1	0	0	0	J	j	j	j	J	SLW W,Rj	W = Rj << 1
0	0	0	1	1	0	1	0	0	0	0	J	j	j	j	j	SRW W,Rj	W = Rj >> 1
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	NOT W	W = ~W
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	SHL W	W = W << 1
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	SHR W	W = W >> 1
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	CLR CY	CY = 0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	SET CY	CY = 1
0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	RET	PC = SAVED PC + 1
0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	NOP	

Figure 9: Set de instrucciones.

9.2 Registro de Microinstrucciones (MIR)

ABUS	BBUS	KM	MW	MR	ALU	SH	CBUS	TYPE		
000000	00000	0	0	0	0000	00	100011	0000000	JMP X	PC = X
000000	00000	0	0	0	0000	00	100011	1000000	JZE X	IF CCR[Z] = 0, THEN PC = X
000000	00000	0	0	0	0000	00	100011	1000000	JNE X	IF CCR[N] = 1, THEN PC = X
000000	00000	0	0	0	0000	00	100011	1010000	JCY X	IF CCR[C] = 1, THEN PC = X
000000	00000	0	0	1	0000	00	100010	0000010	MER W,Y	W = M[Y]
100010	00000	0	1	0	0000	00	100011	0000001	MEW Y,W	M[Y] = W
000000	00000	0	0	0	0000	00	100011	0000000	BSR S	SAVE PC, PC = PC + S
000000	Rj	0	0	0	0001	00	Ri	0001100	EQR Ri,Rj	Ri = Rj
000000	Rj	0	0	0	0001	00	POi	0001100	EQR POi,Rj	POi = Rj
000000	Rj	0	0	0	0001	00	Ri	0001100	EQR Ri, Plj	Ri = Plj
000000	Rj	0	0	0	0001	00	POi	0001100	EQR POi,Plj	POi = Plj
100010	00000	0	0	0	0000	00	Ri	0001001	EQR Ri,W	Ri = W
100010	00000	0	0	0	0000	00	POi	0001001	EQR POi,W	POi = W
100010	Rj	0	0	0	0011	00	Ri	0001101	ANR Ri,Rj	Ri = W & Rj
100010	Rj	0	0	0	0100	00	Ri	0001101	ORR Ri,Rj	Ri = W | Rj
100010	Rj	0	0	0	0101	00	Ri	0001101	XOR Ri,Rj	Ri = W ^ Rj
100010	Rj	0	0	0	0110	00	Ri	0111101	ADR Ri,Rj	Ri = W + Rj + CY
100010	Rj	0	0	0	0111	00	Ri	0111101	SUR Ri,Rj	Ri = W - Rj - CY
100010	Rj	0	0	0	1000	00	Ri	0001101	MUR Ri,Rj	Ri = W * Rj
100010	Rj	0	0	0	1001	00	Ri	0001101	DIR Ri,Rj	Ri = W / Rj
100010	Rj	0	0	0	1010	00	Ri	0001101	MOR Ri,Rj	Ri = W % Rj
100010	00000	0	0	0	0000	01	Ri	0001001	SLR Ri,W	Ri = W << 1
100010	00000	0	0	0	0000	10	Ri	0001001	SRR Ri,W	Ri = W >> 1
000000	00000	1	0	0	0001	00	100010	0000010	EQK W,K	W = K
100010	00000	1	0	0	0011	00	100010	0000011	ANK W,K	W = W & K
100010	00000	1	0	0	0100	00	100010	0000011	ORK W,K	W = W | K
100010	00000	1	0	0	0101	00	100010	0000011	XOK W,K	W = W ^ K
100010	00000	1	0	0	0110	00	100010	0110011	ADK W,K	W = W + K + CY
100010	00000	1	0	0	0111	00	100010	0110011	SUK W,K	W = W - K - CY
100010	00000	1	0	0	1000	00	100010	0000011	MUK W,K	W = W * K
100010	00000	1	0	0	1001	00	100010	0000011	DIK W,K	W = W / K
100010	00000	1	0	0	1010	00	100010	0000011	MOK W,K	W = W % K
000000	Rj	0	0	0	0001	00	100010	0000110	EQW W,Rj	W = Rj
000000	Rj	0	0	0	0001	00	100010	0000110	EQW W,Plj	W = Plj
100010	Rj	0	0	0	0011	00	100010	0000111	ANW W,Rj	W = W & Rj
100010	Rj	0	0	0	0100	00	100010	0000111	ORW W,Rj	W = W | Rj
100010	Rj	0	0	0	0101	00	100010	0000111	XOW W,Rj	W = W ^ Rj
100010	Rj	0	0	0	0110	00	100010	0110111	ADW W,Rj	W = W + Rj + CY
100010	Rj	0	0	0	0111	00	100010	0110111	SUW W,Rj	W = W - Rj - CY
100010	Rj	0	0	0	1000	00	100010	0000111	MUW W,Rj	W = W * Rj
100010	Rj	0	0	0	1001	00	100010	0000111	DIW W,Rj	W = W / Rj
100010	Rj	0	0	0	1010	00	100010	0000111	MOW W,Rj	W = W % Rj
000000	Rj	0	0	0	0001	01	100010	0000110	SLW W,Rj	W = Rj << 1
000000	Rj	0	0	0	0001	10	100010	0000110	SRW W,Rj	W = Rj >> 1
100010	00000	0	0	0	0010	00	100010	0000011	NOT W	W = ~W
100010	00000	0	0	0	0000	01	100010	0000011	SHL W	W = W << 1
100010	00000	0	0	0	0000	10	100010	0000011	SHR W	W = W >> 1
000000	00000	0	0	0	1011	00	100011	0100000	CLR CY	CY = 0
000000	00000	0	0	0	1100	00	100011	0100000	SET CY	CY = 1
000000	00000	0	0	0	0000	00	100011	0000000	RET	PC = SAVED PC + 1
000000	00000	0	0	0	0000	00	100011	0000000	NOP	

Figure 10: Registro de microinstrucciones.

9.3 Gestor de Datos

9.3.1 Banco de Registros

```
module RB_v (input clk,
                                    input [5:0] WC,
                                    input [5:0] RA,
                                    input [4:0] RB,
                                    input [15:0] C,
                                    output reg [15:0] A,
                                    output reg [15:0] B);
        reg [15:0] R [0:34]; // GPR+IO+W+MAR+AUX

        always@ (posedge clk) begin ///
                R[WC] <= C;
        end
        always@ (RA or RB) begin ///
                A <= R[RA];
                B <= R[RB];
        end
endmodule
```

9.3.2 MUX

```
module MUX_v (input sel,
                                    input [15:0] D0,
                                    input [15:0] D1,
                                    output reg [15:0] Q);
        always@(D0 or D1 or sel) begin
                if(sel)
                        Q <= D1;
                else
                        Q <= D0;
        end
endmodule

module MUXK_v (input sel,
                                        input [15:0] D0,
                                        input [15:0] K,
                                        output reg [15:0] Q);
        always@(D0 or K or sel) begin
                if(sel)
                        Q <= K & 16'h01ff;
                else
                        Q <= D0;
        end
endmodule
```

9.4 Código ALU

9.4.1 Bloque del selector

```
module SELEC_ALU (input [3:0] op,
                                    input [15:0] A, B, RES_ADD_SUB, RES_AND, RES_OR, RES_NOT
                                    input [31:0] RES_MULT,
```

```
                                    input RES_SETC,RES_CLRC, OVERFLOW, CIN,
                                    output reg signed [15:0] C,
                                    output reg ADD_SUB,
                                    output reg [3:0] CCR);

    parameter NOPA = 4'b0000,
                                NOPB = 4'b0001,
                                NOTA = 4'b0010,
                                AND = 4'b0011,
                                OR = 4'b0100,
                                XOR = 4'b0101,
                                ADD = 4'b0110,
                                SUB = 4'b0111,
                                MUL = 4'b1000,
                                DIV = 4'b1001,
                                MOD = 4'b1010,
                                CLRC = 4'b1011,
                                SETC = 4'b1100;

    always@ (A, B, op) begin
            CCR = 0;
            case (op)
               NOPA: C <= A;
               NOPB: C <= B;
                     NOTA: C <= RES_NOT_A;
                     AND : C <= RES_AND;
                     OR : C <= RES_OR;
                     XOR : C <= RES_XOR;
                     ADD : begin
                              ADD_SUB <= 1;
                              C <= RES_ADD_SUB;
                              CCR = (CCR | CIN);
                              CCR =  CCR | ( OVERFLOW << 1 );
                     end
                     SUB : begin
                              ADD_SUB <= 0;
                              C <= RES_ADD_SUB;
                              CCR = (CCR | CIN);
                              CCR =  CCR | ( OVERFLOW << 1 );
                     end
                     MUL : begin
                              C <=  16'b1111111111111111 & RES_MULT;
                     end
                     DIV : C <= RES_DIV;
               MOD : C <= RES_MOD;
               CLRC: CCR = (CCR & RES_CLRC);
                     SETC: CCR = (CCR | RES_SETC);
                     default : C <= 0;
            endcase
            if(C == 0)
                     CCR = CCR | 4'b0100;
            else if(C < 0)
                     CCR = CCR | 4'b1000;
    end
endmodule
```

9.4.2 Bloque de Unidad Lógica UC

```
module UC_v (input clk,
                                input [5:0] C2,
                                input [6:0] T2,
                                input [3:0] ALU1,
                                input [5:0] C1,
                                input [6:0] T1,
                                input [5:0] A0,
                                input [4:0] B0,
                                input [6:0] T0,
                                input MR0, //Memory read.
                                output reg en0,
                                output reg en1);
        reg [2:0] hold;
        reg [2:0] hold_;
        initial begin
                hold = 0;
                hold_ = 0;
                en0 <= 1;
                en1 <= 1;
        end
        always@ (posedge clk) begin
                if (hold > 0 || hold_ > 0) begin
                        if(hold_) begin
                                hold_ = hold_ >> 1;
                                if (hold_ == 0) begin
                                        if(!hold)
                                                en0 <= 1;
                                        en1 <= 1;
                                end
                        end
                        else begin
                                hold = hold >> 1;
                                if (hold == 0)
                                        en0 <= 1;
                        end
                end
                else begin
                        if (MR0) begin //MER W,Y
                                hold = 3'b100;
                                en0 <= 0;
                        end
                        if (!hold) begin
                                if (T0 & 7'b0000001) begin //read W when W is to be writte
                                        if (T1 & 7'b0000010) begin
                                                hold = 2'b10;
                                                en0 <= 0;
                                        end
                                end
                                if (T0 & 7'b0000100) begin //read Ri when Ri is to be writ
                                        if (T1 & 7'b0001000) begin
                                                if (A0 == C1 || B0 == C1) begin
                                                        hold = 2'b10;
                                                        en0 <= 0;
                                                end
                                        end
                                end
                                if (T0 & 7'b0010000) begin //read CY when CY is to be writ
```

```
                    if (T1 & 7'b0100000) begin
                        hold = 2'b10;
                        en0 <= 0;
                    end
                end
                case (ALU1)
                    4'b1000 : begin //MUL
                        hold_ = 3'b100;
                        en0 <= 0;
                        en1 <= 0;
                    end
                    4'b1001 : begin //DIV
                        hold_ = 3'b100;
                        en0 <= 0;
                        en1 <= 0;
                    end
                    4'b1010 : begin //MOD
                        hold_ = 3'b100;
                        en0 <= 0;
                        en1 <= 0;
                    end
                endcase
                if (!hold && !hold_/* && en_*/) begin
                    if (T0 & 7'b0000001) begin //read W when W is to b
                        if (T2 & 7'b0000010) begin
                            //if (en_) begin
                                hold = 2'b1;
                                en0 <= 0;
                            //end
                        end
                    end
                    if (T0 & 7'b0000100) begin //read Ri when Ri is to
                        if (T2 & 7'b0001000) begin
                            if (A0 == C2 || B0 == C2) begin
                                //if (en_) begin
                                    hold = 2'b1;
                                    en0 <= 0;
                                //end
                            end
                        end
                    end
                    if (T0 & 7'b0010000) begin //read CY when CY is to
                        if (T2 & 7'b0100000) begin
                            //if (en_) begin
                                hold = 2'b1;
                                en0 <= 0;
                            //end
                        end
                    end
                end
            end
        end
    end
endmodule
```