Diseño de un microprocesador usando el lenguaje de descripción de hardware VHDL.

Sandra Milena Ovallos Gaona.

Director:
Elkim Felipe Roa Fuentes.

Codirector:
Jorge Hernando Ramón Suárez.

Septiembre, 2005

TITULO: DISEÑO DE UN MICROPROCESADOR USANDO EL LENGUAJE DE DESCRIPCIÓN DE HARDWARE VHDL[1]

AUTOR: SANDRA MILENA OVALLOS GAONA[2]

Palabras cláves.

Microprocesador, *Datapath*, Unidad de control, ALU, FPGA, Instrucciones, *Pipeline*, VHDL.

Resumen

Este trabajo muestra el diseño de un microprocesador de propósito general implementado mediante el leguaje de descripción de hardware VHDL y sintetizado en un FPGA utilizando el software ISE 6.1i de la empresa Xilinx. Este diseño fue realizado con el propósito de mostrar de forma general la estructura interna de un microprocesador.

El diseño inicia con el planteamiento de las especificaciones generales del microprocesador, discute la selección del conjunto de instrucciones, el formato de instrucción, los modos de direccionamiento. En total se seleccionaron 27 instrucciones RISC que operan sobre datos de 16 bits, el microprocesador cuenta con un formato de instrucción de 24 bits, 5 modos de direccionamiento, 8 registros de propósito general y 5 registros de propósito específico.

Una vez seleccionado el conjunto de instrucciones se diseñó una estructura que permite su correcta ejecución. Esta estructura consta de un datapath que se encarga de operar sobre los datos y una unidad de control que envía las señales de control adecuadas dependiendo de la instrucción que se esté ejecutando. El datapath cuenta con bloques funcionales como la unidad aritmético-lógica, encargada de realizar las operaciones; registros y memorias utilizados para almacenar datos; todos estos conectados mediante buses y multiplexores. Por su parte la unidad de control es una máquina de cinco estados que se encarga de interpretar las instrucciones, recibe el código de operación de la instrucción y entrega las señales de control en el momento indicado de acuerdo a la instrucción que se encuentre en ejecución.

Luego de realizar el diseño de la unidad de control y el datapath que interpreten y ejecuten correctamente el conjunto de instrucciones, se planteó el diseño de una arquitectura pipeline, buscando mejorar el rendimiento del microprocesador. El trabajo

[1]Trabajo de grado.

[2]Facultad de ingenierías físico mecánicas. Escuela de ingenierías eléctrica, electrónica y de telecomunicaciones.

muestra los cambios realizados a la estructura original.

Finalmente se sintetizó el diseño en La FPGA Spartan 2E de la empresa Xilinx, utilizando la tarjeta de desarrollo digilab. Se realizaron pruebas de simulación y síntesis tanto para la arquitectura original como para la arquitectura pipeline, obteniendo los resultados esperados.

TITLE: DESIGN OF A MICROPROCESSOR USING THE HARDWARE DESCRIPTION LANGUAGE VHDL[3]

AUTHOR: SANDRA MILENA OVALLOS GAONA[4]

Keys Words.

Microprocessor, Datapath, Control Unit, ALU, FPGA, Instructions, Pipeline, VHDL.

Abstract

This Word introduce a general purpose microprocessor design using the hardware description language VHDL and synthesize it in a FPGA using the Xilinx ISE 6.1i software. This design was realized with the purpose to show in a general way the internal structure of a microprocessor. The design starts with the the microprocessor generals specifications, it discuss the set instructions selection, the instruction format and the addressing modes. It was selected totally 27 RISC instructions whose operated over 16 bits data, the microprocessor counts with a 24 bits instructions format, 5 addressing modes, 8 general purpose register and 5 specific purpose register. Once selected the instructions set it was designed a structure whose allows its correct execution. This structure have a datapath whose principal responsibility is operate over the data and a control unit that send the correct control signal depending of the execution instructions. The datapath contain functional blocks like the arithmetic-logic unit, which is in charge of realizing the operations, registers and memories using to the data store; all of them connected through buses and multiplexors. On the other hand, the control unit is a five state machine in charge of the instructions interpretations, it receives the instruction operation code and deliver the control signs in the right moment according with the executable instruction. After realizing the control unit and datapath that interpret and realize de right executions instructions, it was planted the pipeline architecture design, looking for improve the microprocessor performance. The Word shows the realized changes to the original structure. Finally the design was synthesized in the Xilinx spartan 2E FPGA, using the digilab development target. It was realized simulations and synthesis test for the original structure and for the pipeline architecture, obtaining the expected results.

[3]Trabajo de grado.

[4]Facultad de ingenierías físico mecánicas. Escuela de ingenierías eléctrica, electrónica y de telecomunicaciones.

Contenido general

Agradecimientos **XI**

1. Introducción. **1**
1.1. Motivación. 1
1.2. Organización del documento. 2
1.3. El microprocesador. 3
1.4. Selección del conjunto de instrucciones. 3
1.4.1. Arquitecturas CISC y RISC. 4
1.4.2. Formato de instrucción. 5
1.4.3. Conjunto de instrucciones. 6
1.4.4. Modos de direccionamiento. 8

2. Diseño del microprocesador. **13**
2.1. Organización del microprocesador. 13
2.2. Descripción del *datapath*. 15
2.2.1. Memorias. 15
2.2.2. Registros. 17
2.2.3. ALU: Unidad Aritmético-Lógica. 18
2.2.4. Sumador del contador de programa ($PC + 1$). 31
2.2.5. Multiplexores. 32
2.3. Diseño de la unidad de control. 32
2.3.1. Estados de la instrucción. 33
2.3.2. Ejecución de las instrucciones. 34
2.4. Simulación. 39

3. Implementación de la arquitectura *pipeline*. **43**
3.1. Arquitectura *pipeline*. 43
3.2. Ejecución de las instrucciones en la arquitectura *pipeline*. 45
3.2.1. Instrucciones de carga. 47
3.2.2. Instrucciones de control. 48
3.2.3. Instrucciones de almacenamiento. 49

3.2.4. Instrucciones aritmético-lógicas. 49
3.3. Diseño del *datapath*. 50
3.3.1. Registros. 52
3.3.2. Multiplexores. 54
3.4. Diseño de la unidad de control. 55
3.4.1. Señales de la unidad de control. 56
3.5. Problemas en la ejecución. 59
3.5.1. Dependencia de datos. 59
3.5.2. Riesgos de control. 59
3.5.3. Solución a los conflictos en la ejecución: Esperas 60
3.6. Simulación. 61

4. Síntesis y análisis de resultados. 65
4.1. Hardware utilizado. 65
4.2. Consideraciones a nivel de *hardware*. 68
4.3. Consideraciones a nivel de *software*. 70
4.3.1. Unidad de control. 70
4.3.2. Memorias. 71
4.4. Resultados de simulación y síntesis. 71
4.4.1. Resultados de simulación. 71
4.4.2. Resultados de síntesis sobre medidas. 72
4.4.3. Resultados de síntesis sobre simulación. 73
4.4.4. Programa de prueba. 73

5. Conclusiones. 76
5.1. Especificaciones finales. 76
5.2. Observaciones y conclusiones. 77
5.3. Recomendaciones para trabajos futuros. 80

A. Gráficas de resultados. 83
A.1. Memoria de prueba arquitectura sin *pipeline*. 83
A.2. Memoria de prueba arquitectura *pipeline*. 85

B. Características del microprocesador. 88
B.1. Arquitectura sin *pipeline* 91
B.2. Arquitectura *pipeline*. 95

C. Implementación en el lenguaje VHDL. 98
C.1. Consideraciones del lenguaje VHDL. 98
C.2. Descripción del microprocesador. 100

Índice de figuras

1.1. Formatos de instrucción para 5 microprocesadores. 5
1.2. Formato de instrucción. 6
1.3. Instrucciones de tratamiento de datos. 7
1.4. Direccionamiento inmediato. 10
1.5. Direccionamiento directo. 10
1.6. Direccionamiento directo por registro. 11
1.7. Direccionamiento implícito. 11
1.8. Direccionamiento indexado. 11

2.1. Esquema general de un microprocesador. 14
2.2. Datapath. 16
2.3. Archivo de registros de propósito general. 17
2.4. Esquema general de la ALU. 19
2.5. Sumador de acarreo propagado de 4 bits. 20
2.6. Sumador de acarreo anticipado. 22
2.7. Circuito lógico implementado para calcular el complemento a2. 23
2.8. Restador. 23
2.9. Multiplicador de Pezaris para dos números de 5 bits. 26
2.10. Variación del multiplicador de Pezaris. 27
2.11. Multiplicador de Bawgh-Wooley para dos números de 5 bits. 29
2.12. Operaciones shift y rotación. 30
2.13. Bloque sumador de programa. 31
2.14. Unidad de control. 33
2.15. Carga dato inmediato en el registro result. 35
2.16. Ejecución de la multiplicación. 37
2.17. Ejecución de la instrucción salto incondicional. 37
2.18. Resultados de simulación del programa 1. 41
2.19. Resultados de simulación del programa 2. 42

3.1. Ejecución de $a + b$ en una arquitectura sin *pipeline*. 44
3.2. Ejecución de $a + b$ en una arquitectura con *pipeline*. 44
3.3. Ejemplo de la ejecución de 7 instrucciones. 46

3.4. Datapath con arquitectura pipeline. 51
3.5. Cálculo de la siguiente instrucción. 53
3.6. Registros de almacenamiento de las señales de control. 54
3.7. Conflicto con el multiplexor 4. 54
3.8. Señales de control de la instrucción suma. 56
3.9. Unidad de control para la arquitectura pipeline. 57
3.10. Conflictos en la ejecución de la arquitectura pipeline. 60
3.11. Solución a los conflictos en la ejecución. 61
3.12. Resultados de simulación del programa 1 en la arquitectura *pipeline*. . . 63
3.13. Resultados de simulación del programa 2 en la arquitectura *pipeline*. . . 64

4.1. Diagrama de bloques de la familia de FPGAs Spartan-IIE. 66
4.2. Diagrama de bloques de la tarjeta Digilab 2E. 68
4.3. Esquema de síntesis del microprocesador. 69
4.4. Diagrama de flujo. 74

B.1. Formato de las instrucciones. 90
B.2. Diagrama de bloques. 91
B.3. Diagrama de bloques para la arquitectura pipeline. 96

C.1. Entidad y arquitectura en VHDL. 99
C.2. Definición de componentes. 99
C.3. Sentencia process. 100
C.4. Bloques implementados en el lenguaje VHDL. 101
C.5. Implementación de la unidad de control. 103

Índice de tablas

1.1. Conjunto de instrucciones. 9
1.2. Especificaciones del microprocesador. 12

2.1. Tipos de sumadores. 25
2.2. Operaciones de la ALU con sus respectivas señales de control. 31
2.3. Vectores *rst_out* y *rw_reg*. 33
2.4. Señales de la unidad de control. 38
2.5. Programa 1 39
2.6. Programa 2 40

3.1. Vectores *rst_out*, *r_reg* y *w_reg*. 57
3.2. Señales de la unidad de control en la arquitectura *pipeline* 58
3.3. Programa 1 para la arquitectura *pipeline*. 62
3.4. Programa 2 62

4.1. Características generales de la FPGA Spartan 2E XC2S200E-PQ208. . 67
4.2. Pines utilizados. 70
4.3. Resultados de simulación. 73
4.4. Instrucciones máximo común divisor. 75

5.1. Especificaciones finales. 77

A.1. Memoria de prueba arquitectura sin *pipeline*. 84

B.1. Resumen del conjunto de instrucciones 89
B.2. Señales de la unidad de control. 94
B.3. Señales de la unidad de control en la arquitectura *pipeline* 97

Si el señor no construye la casa
en vano se levantan los albañiles;
si el señor no protege la ciudad
en vano vigila el centinela.

En vano te levantas tan temprano
y te acuestas tan tarde, y con
tanto sudor comes tu pan;
él lo da a sus amigos mietras duermen.
(Salmo 127)

Agradecimientos

Expreso mis sinceros agradecimientos

A Elkim Felipe Roa, Ingeniero Electricista, director del proyecto, por su respaldo, confianza y especialmente por su esfuerzo para que este trabajo cumpliera con todas las espectativas planteadas.

A Jorge Hernando Ramón Suárez, Ingeniero Electricista, codirector del proyecto, por su amabilidad y colaboración oportuna, facilitando la realización de este proyecto.

A la Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones, por la educación brindada, y por facilitar los elementos para la realización del proyecto.

A Daniel Sierra Bueno por su desinteresada colaboración.

Capítulo 1

Introducción.

En este capítulo se motiva para iniciar la tarea de diseño e implementación de un microprocesador, se comentan las razones que llevaron a la realización de este trabajo y plantea uno de los propósitos de éste libro: Mostrar la estructura básica de los microprocesadores mediante el diseño de un microprocesador de propósito general.

Por otro lado, en este capítulo se presenta el conjunto de instrucciones que será implementado, describe características como el tipo de arquitectura, el formato de instrucción, los modos de direccionamiento. Se inicia con una definición de microprocesador para indicar el tipo de dispositivo que se va a diseñar, posteriormente muestra las características que se deben tener en cuenta para seleccionar el conjunto de instrucciones y se describen las instrucciones que se implementarán en este diseño, finalmente se muestran las especificaciones básicas con que contará el microprocesador.

1.1. Motivación.

Probablemente todo estudiante de electrónica y programas afines ha tenido que realizar algún trabajo relacionado con el uso de microprocesadores, para lo cual invirtió parte de su tiempo seleccionando el dispositivo indicado, analizando su funcionamiento, realizando algunas pruebas y por último implementando la aplicación deseada. Sin embargo no siempre se dedica tiempo a analizar detenidamente la estructura interna de estos dispositivos, cómo se ejecutan las instrucciones, las decisiones que se tomaron en el diseño y construcción de los bloques funcionales, etc. El conocimiento de la estructura y funcionamiento interno de los microprocesadores facilita el proceso de selección, además reduce el tiempo de análisis del comportamiento de un microprocesador particular.

Este trabajo muestra la arquitectura interna básica de estos dispositivos mediante el diseño de un microprocesador de propósito general, para lo cual se estudia cada uno de sus componentes desde el nivel de compuertas lógicas y la manera como deben ser combinados hasta formar el microprocesador.

El análisis parte de la selección de las instrucciones que ejecutará el microprocesador y la definición de las especificaciones básicas con que contará, continúa con el diseño y construcción de los principales bloques funcionales como memorias, registros entre otros, luego se muestra una forma de mejorar el rendimiento del microprocesador mediante el diseño de una arquitectura *pipeline*, todo este proceso es implementado mediante el lenguaje de descripción de *hardware*[1] VHDL[2]. Finalmente se sintetiza el diseño en una FPGA[3]y se realizan pruebas, validando los resultados mediante medidas de las especificaciones del microprocesador resultante.

1.2. Organización del documento.

Este libro muestra el proceso llevado a cabo en el diseño, implementación y síntesis de un microprocesador, cada capítulo contiene los fundamentos generales sobre la estructura del microprocesador y posteriormente muestra la aplicación de dichos conceptos en el caso particular del diseño de un microprocesador de propósito general.
En total el libro está organizado en cinco capítulos. El capítulo 1 discute las consideraciones básicas a tener en cuenta en el diseño de un microprocesador, muestra la selección del conjunto de instrucciones así como las características generales de cada instrucción, se discuten aspectos como el tipo de arquitectura, el formato de instrucción, los modos de direccionamiento y los tipos de instrucciones.
El capítulo 2 describe el proceso de diseño del microprocesador, se especifica con detalle la forma como se ejecutan las instrucciones, las características de cada bloque funcional y la forma como se combinan hasta tener un microprocesador que interprete y ejecute las instrucciones planteadas. Este capítulo muestra con más detalle las características de cada instrucción.
El tercer capítulo esta dedicado a describir el diseño de la arquitectura *pipeline*, se discuten los cambios que deben ser implementados en el microprocesador para que esta funcione correctamente.
El capítulo cuatro muestra las pruebas realizadas y el proceso de síntesis del circuito en la FPGA.
Finalmente, en el capítulo 5 se presentan las observaciones, conclusiones y recomendaciones obtenidas durante la elaboración de este trabajo.

[1]En este trabajo se utilizaran algunos términos en inglés para ubicarlos bibliográficamente
[2]VHDL: VHSIC *Hardware Description Language.*
[3]FPGA: *Field Programmable Gate Array.*

1.3. El microprocesador.

Es un dispositivo integrado diseñado para interpretar y ejecutar un conjunto de tareas; acepta datos binarios de entrada, los procesa de acuerdo con las instrucciones y provee una salida.
Los microprocesadores pueden ser divididos en dos categorías, microprocesadores de propósito general y microprocesadores de propósito específico. Los primeros son diseñados para ejecutar una variedad de tareas, para esto cada tarea no es implementada directamente en el *hardware*, sino que es representada por una secuencia de instrucciones que se almacenan en memoria y son interpretadas por el microprocesador, si se desea ejecutar una nueva tarea, las instrucciones en memoria pueden ser cambiadas.
Por otro lado, existen microprocesadores de propósito específico diseñados para una aplicación en particular, utilizan memoria de sólo lectura, sus funciones no pueden ser modificadas [1], un ejemplo de estos microprocesadores es el procesador digital de señales (DSP -*Digital Signal Processing*-), dedicado especialmente al tratamiento de señales, su *hardware* permite un flujo de datos de alta velocidad hacia y desde la unidad de cálculo y la memoria, son usados en el tratamiento de señales en tiempo real, comunicaciones, multimedia, procesamiento de imágenes, audio, video, entre otros.

Dado que este trabajo se realiza con el propósito de mostrar las características básicas de un microprocesador, se decidió diseñar un microprocesador de propósito general con el fin de no limitar el problema a las condiciones particulares de un dispositivo. Para realizar este diseño se define un conjunto de instrucciones y se implementa una estructura que las interprete y permita su correcta ejecución. A continuación se muestra las características del conjunto de instrucciones seleccionado.

1.4. Selección del conjunto de instrucciones.

El primer paso en el diseño del microprocesador es definir el conjunto de instrucciones que serán ejecutadas, estas tienen una gran influencia en el desempeño del microprocesador dado que el *hardware* debe de algún modo implementar la semántica de las instrucciones. Definir este conjunto no solo es elegir algunas instrucciones, también implica la selección de la arquitectura, el formato de instrucción, los modos de direccionamiento, el número de bits.

1.4.1. Arquitecturas CISC y RISC.

La primera decisión tomada en la selección del conjunto de instrucciones es el tipo de arquitectura. Existen dos modos de diseño de microprocesadores, la arquitectura CISC (*Complex Instruction Set Computer*) y la arquitectura RISC (*Reduced Instruction Set Computer*) [2]. La arquitectura CISC plantea el diseño de instrucciones complicadas, acercándolas a los lenguajes de alto nivel; se basa en la idea que cuantas más posibilidades ofrezca el nivel de máquina convencional, más fácil será desarrollar lenguajes de niveles superiores, sin embargo se observa que el tiempo que demora una instrucción en ejecutarse es considerablemente alto.
Una técnica utilizada por los diseñadores de procesadores CISC es la microprogramación, esta consiste en que cada instrucción de máquina es interpretada por un microprograma localizado en una memoria dentro del circuito integrado del procesador [3]. Estos microprocesadores son desarrollados por empresas como Intel, AMD, entre otras.

Un diseño RISC por su parte busca reducir el tiempo de ejecución de cada instrucción aunque el número de instrucciones por tarea aumente. El objetivo de la arquitectura RISC es hacer más rápidas las instrucciones frecuentes, es decir, no busca ofrecer un conjunto de instrucciones de alto nivel sino proveer instrucciones básicas rápidas; las operaciones más complejas son implementadas mediante un compilador. Es importante notar que el número de instrucciones por tarea aumenta, pero el conjunto de instrucciones se reduce, además reduce los modos de direccionamiento [3, 4]. La arquitectura RISC ha permitido la aparición de microprocesadores poderosos cuya aplicación ha sido en servidores, también se ha implementado en computadores personales, maquinas de juego, entre otros. Algunas empresas que desarrollan microprocesadores RISC son *Sun Microsystems* (SPARC, MIPS), Apple, Motorola e IBM (Power PC).

Como la arquitectura RISC permite implementar instrucciones sencillas, rápidas y además facilita la implementación del *pipeline*[4] con el cual es posible mejorar el rendimiento del microprocesador; y siendo el propósito de este trabajo facilitar la comprensión de la estructura interna de un microprocesador se decidió utilizar esta arquitectura para implementar el diseño.
Una de las características de la arquitectura RISC es que permite un único formato de instrucción, esto simplifica el diseño de la estructura que interprete y ejecute las instrucciones, a continuación se describe el formato utilizado en este trabajo.

[4]Concepto ampliado en el capítulo 3.

1.4.2. Formato de instrucción.

El formato de instrucción afecta directamente la estructura interna del microprocesador pues debe definir características como el número de bits, los registros y los modos de direccionamiento que utilizará el microprocesador. Este formato puede ser diferente para cada tipo de instrucción, en forma general debe contener un código de operación que indique las acciones a realizar, y los operandos sobre los cuales trabajar [5]. La figura 1.1[5] es un ejemplo de los formatos de instrucción en cinco microprocesadores RISC para instrucciones que operan entre registros. Se observa que todos los formatos de instrucción tienen una estructura similar, sin embargo cada microprocesador interpreta los espacios del formato de manera diferente.

	31	25	20	15	10		0
DLX	Op	rs1	rs2	rd	Opx		
MIPS	Op	rs1	rs2	rd	Const.	Opx	
Power PC	Op	rd	rs1	rs2	Opx		
PA- RISC	Op	rs1	rs2	Opx		rd	

	31	29	24	18	13	12	4	0
SPARC	Op	rd	Opx	rs1	0	Opx	rs2	

Figura 1.1: Formatos de instrucción para 5 microprocesadores.

Todos los formatos de instrucción tienen una longitud de 32 bits, la casilla *Op* corresponde al código de la operación, *Opx* es una extensión del código de operación, los microprocesadores tiene un archivo de 32 registros de propósito general por lo tanto la dirección de estos registros debe tener una longitud de cinco bits, las casillas *rs1* y *rs2* corresponden a la dirección de los operandos y *rd* la dirección del registro destino, es decir donde se almacenará el resultado, la casilla *const* corresponde a una constante usada como dato inmediato o como una dirección.

[5] Tomado de: [6]

Para la elección adecuada del número de bits de la instrucción se deben tener en cuenta las especificaciones del microprocesador con respecto al bus de datos, el número de registros y el número de instrucciones. En razón a esto se seleccionaron 27 instrucciones, 8 registros de propósito general y un bus de datos de 16 bits; en total este formato cuenta con 24 bits organizados como se muestra en la figura 1.2

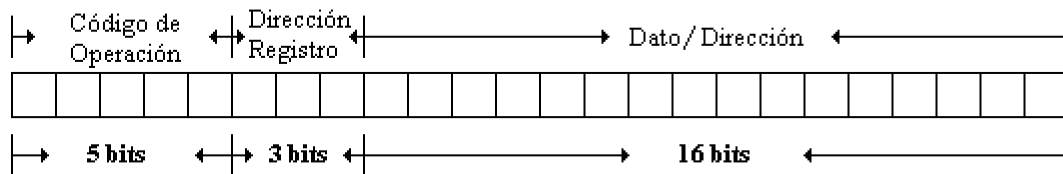


Figura 1.2: Formato de instrucción.

Código de operación: Es un identificador único para cada instrucción, este código va directamente a la unidad de control y con base en él se toman las acciones correspondientes para cada instrucción. Se seleccionó un espacio de 5 bits pues con un conjunto de hasta 32 instrucciones es suficiente para realizar diferentes programas, además el número de instrucciones en este caso está limitado por el tamaño de la FPGA donde se sintetizará el microprocesador.

Dirección de registro: Para facilitar la realización de operaciones este microprocesador cuenta con un conjunto de 8 registros de propósito general organizados en un archivo de registros[6] , por tanto el formato de instrucción debe incluir un espacio de 3 bits que permita seleccionar el registro a utilizar.

Dato/Dirección: Este segmento depende del modo de direccionamiento de la instrucción que se esté ejecutando, puede corresponder directamente al operando de la instrucción o a una dirección que indique la ubicación del operando en memoria. Un bus de datos de 16 bits permite operar sobre un rango amplio de valores sin exceder el tamaño de la FPGA.

1.4.3. Conjunto de instrucciones.

Algunas consideraciones importantes para seleccionar el conjunto de instrucciones son que las operaciones sean requeridas frecuentemente y que otras operaciones sencillas para las que no existan instrucciones puedan ser ejecutadas con facilidad mediante una combinación simple de las instrucciones existentes. Normalmente la efectividad del conjunto de instrucciones es medida en base al número de instrucciones requeridas

[6]Las decisiones tomadas en el diseño del archivo de registros se discuten en el capítulo II.

para realizar una tarea, si se utilizan pocas instrucciones significa menos espacio en la memoria de programa y menor tiempo de ejecución de la tarea, es decir el conjunto de instrucciones es eficiente [5].
A continuación se discuten los tipos de instrucciones utilizados en este trabajo, estas son una muestra representativa de las instrucciones más usadas por la mayoría de los microprocesadores.

Instrucciones de transferencia de datos.

Mueven los datos entre bloques del microprocesador sin realizar ninguna operación, su importancia radica en que le brindan flexibilidad al programador para utilizar y almacenar datos en diferentes lugares. Estas instrucciones dejan el contenido de la fuente sin modificar, en realidad se traslada una copia del dato original. En este diseño existen dos clases de instrucciones de transferencia de datos.

Carga: Permiten el manejo de los registros, cargar un dato significa llevarlo a un registro específico o de propósito general, estos datos pueden provenir de las memorias o de otro registro.

Almacenamiento: Estas instrucciones llevan los datos de los registros a la memoria de datos. Utilizar una memoria de datos permite ejecutar instrucciones sobre un amplio número de operandos sin que estos deban ser almacenados fuera del microprocesador.

Instrucciones de tratamiento de datos.

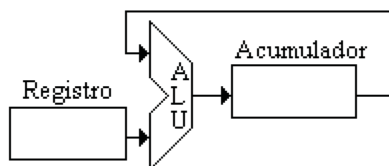


Figura 1.3: Instrucciones de tratamiento de datos.

Estas instrucciones producen nuevos resultados a través de la manipulación de los operandos. Todo microprocesador cuenta con instrucciones de suma y resta, a partir de estas puede realizar operaciones más complejas como la multiplicación o división. El tratamiento de los datos define en gran medida la capacidad del microprocesador, en este diseño se seleccionaron 14 instrucciones de tratamiento de datos divididas en instrucciones aritméticas, lógicas, de desplazamiento e instrucciones que incrementan o disminuyen en uno el valor de un registro. La figura 1.3 muestra el esquema de

las instrucciones de tratamiento de datos, las operaciones son realizadas en la unidad aritmético-lógica, en este diseño el registro de entrada y salida de la ALU es el mismo, es decir este registro actúa como acumulador.

Instrucciones de control.

Estas instrucciones le permiten al programador manipular algunas características del microprocesador, dentro de las instrucciones de control se encuentran los saltos, estos permiten alterar el orden de ejecución de las instrucciones, es decir el programa no necesariamente debe realizarse en el orden en que son almacenadas las instrucciones. Para esto cambian el contenido del registro que almacena la dirección de la memoria de programa (PC Contador de programa), lo cual causa la transferencia del control a otro lugar del programa.

La instrucción no opere es un caso particular de las instrucciones de control pues ejecuta la operación de multiplicación dentro de la ALU pero no cambia ningún valor de los registros o las memorias, esta instrucción es utilizada para dar al programador la posibilidad de utilizar esperas.

La tabla 1.1 muestra el resumen del conjunto de instrucciones seleccionado, la primera columna de la tabla muestra el tipo de instrucción, la segunda los códigos de operación definidos, luego se presenta el nombre nemotécnico utilizado para representar la instrucción, finalmente se realiza una corta descripción.

1.4.4. Modos de direccionamiento.

Son utilizados para dar flexibilidad al programador y reducir el tamaño del formato de instrucción. Como se observó anteriormente, toda instrucción está formada por un código de operación y unos operandos, sin embargo el formato de instrucción puede contener directamente estos operandos, o una dirección de memoria o de registros donde ubicarlos, los modos de direccionamiento indican la forma de interpretar el valor ubicado en el campo de operando del formato de instrucción [7].

Una vez seleccionado las instrucciones que se desean implementar, se deben definir la forma en que estas deben acceder a los operandos, se definieron los siguientes modos de direccionamiento con el fin de permitir la correcta ejecución de las instrucciones seleccionadas.

Tabla 1.1: Conjunto de instrucciones.

TIPO	COD_OP	INSTRUCCIÓN	DESCRIPCIÓN
Carga	00000	$R = di$	Carga dato inmediato en el registro *result*.
	00001	$R = F_i$	Carga un valor del archivo de registros en *result*.
	00010	$R = M[di]$	Carga un dato de memoria (de la dirección dada por un dato inmediato) en el registro *result*.
	00011	$F_i = di$	Carga un dato inmediato en el archivo de registros.
	00100	$F_i = R$	Carga el valor de *result* en el archivo de registros.
	00101	$F_i = M[R]$	Carga un valor de memoria (de la dirección dada por el registro *result*) en el archivo de registros.
	00110	$F_i = Mh$	Carga el valor de Mh en el archivo de registros.
Almacena	00111	$M[di] = R$	Almacena el valor del registro *result* en la dirección de memoria dada por un dato inmediato.
	01000	$M[R] = F_i$	Almacena un valor del archivo de registros. en la dirección dada por el registro *result*.
Aritméticas	01001	$Sum[R, F_i]$	Suma el registro *result* con el archivo de registros, el resultado lo almacena en *result*.
	01010	$Res[R, F_i]$	Resta el registro *result* con el archivo de registros, el resultado lo almacena en *result*.
	01011	$Mult[R, F_i]$	Multiplica el registro *result* con el archivo de registros, almacena en *result*, y en el registro Mh.
Lógicas	01100	C_{a2}	Realiza el complemento a2 al valor del registro *result*, almacena en *result*.
	01101	$Xor[R, F_i]$	Operación XOR entre el registro *result* y el archivo de registros, el resultado lo almacena en *result*.
	01110	$And[R, F_i]$	Operación AND entre el registro *result* y el archivo de registros, el resultado lo almacena en *result*.
	01111	$Or[R, F_i]$	Operación OR Para un valor del registro *result*.
	10000	Not	Niega el valor del registro *result*.
Desplazamiento	10001	SD	Desplaza *result* una posición hacia la derecha.
	10010	SI	Desplaza *result* una posición hacia la izquierda.
	10011	RD	Rota *result* una posición hacia la derecha.
	10100	RI	Rota *result* una posición hacia la izquierda.
INC/DIS	10101	INC	Incrementa en 1 el valor del registro *result*.
	10110	DIS	Disminuye en 1 el valor del registro *result*.
Salto	10111	JMP	Carga un dato inmediato en el registro Pc.
	11000	Jz	Si *result* es cero, carga un dato inmediato en PC.
	11001	Jn	Si *result* es negativo, carga un dato inmediato en PC.
No opere	11010	NOP	Multiplica el archivo de registros por 0, no almacena el resultado.

Direccionamiento inmediato.

Le permite al programador trabajar directamente con valores constantes, el microprocesador toma el dato directamente del formato de instrucción. La figura 1.4 muestra un ejemplo de este modo de direccionamiento, la instrucción carga un dato a un registro específico, por tanto no necesita la dirección de este registro, el dato se encuentra directamente en los 16 bits menos significativos del formato de instrucción.

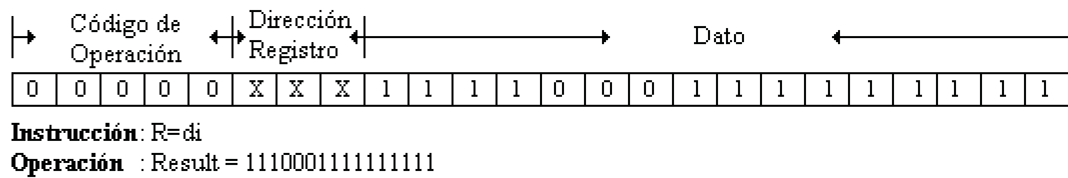


Figura 1.4: Direccionamiento inmediato.

Direccionamiento directo.

El formato de instrucción indica la dirección de la memoria de programa o de datos a la que se desea acceder, se usa en algunas instrucciones de salto para permitirle al usuario indicar la posición de memoria en que desea ubicarse y en algunas instrucciones de almacenamiento para indicar la dirección de la memoria de datos. La figura 1.5 muestra la instrucción de salto inmediato, los 16 bits menos significativos del formato de instrucción indican en este caso la dirección de la memoria de programa hacia donde se debe realizar el salto.

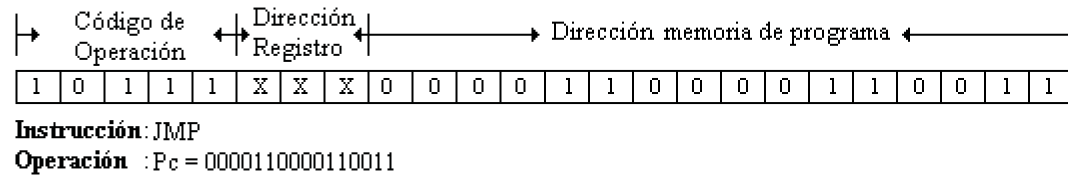


Figura 1.5: Direccionamiento directo.

Direccionamiento directo por registro.

El microprocesador contiene 8 registros de propósito general organizados como un archivo de registros, por lo tanto en algunas instrucciones se utiliza el espacio de 3 bits en el formato de instrucción para indicar el registro a utilizar. La figura 1.6 muestra la instrucción suma, el primer operando se encuentra en un registro específico por tanto no necesita dirección, el segundo operando se encuentra en el archivo de registros la ubicación está dada por los tres bits de dirección de registro, el resultado se almacena en un registro de propósito específico.

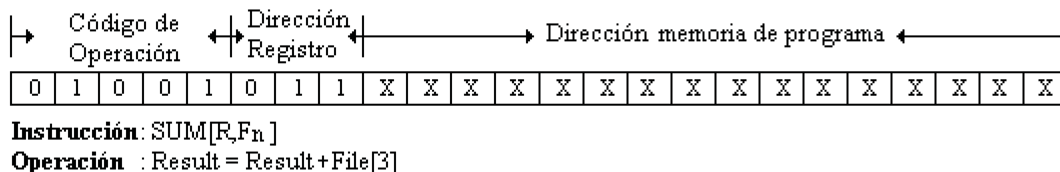


Figura 1.6: Direccionamiento directo por registro.

Direccionamiento implícito.

Algunas instrucciones sólo necesitan el código de operación para ser ejecutadas. La figura 1.7 muestra la instrucción no opere, esta realiza la multiplicación entre dos registro específicos (uno de ello es cero) y no almacena ningún valor, por tanto no necesita direcciones.

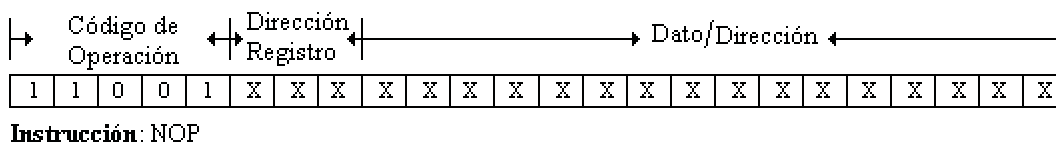


Figura 1.7: Direccionamiento implícito.

Direccionamiento indexado.

Existen instrucciones donde la dirección de la memoria de programa está dada por el valor de un registro. La figura 1.8 muestra una instrucción de almacenamiento, la dirección donde se almacenará el dato está dada por un registro del microprocesador ($result$), este modo de direccionamiento se utiliza para el manejo de vectores.

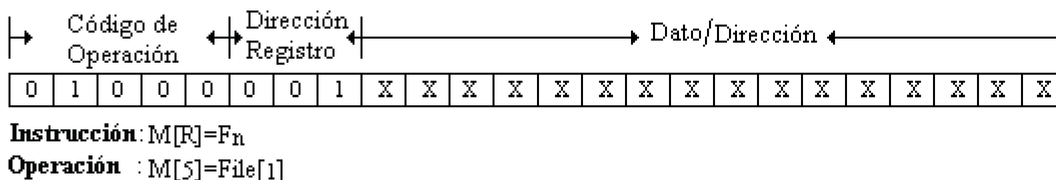


Figura 1.8: Direccionamiento indexado.

La tabla 5.1 presenta un resumen de las especificaciones deseadas del microprocesador. Resaltando que el microprocesador debe ejecutar 27 instrucciones tipo RISC y operar sobre datos de 16 bits.

Tabla 1.2: Especificaciones del microprocesador.

Especificación	Resultados esperados
Conjunto de instrucciones	Número de instrucciones: 27
	Arquitectura: RISC
Modos de direccionamiento	Direccionamiento inmediato.
	Direccionamiento directo.
	Direccionamiento directo por registro.
	Direccionamiento implícito.
	Direccionamiento indexado.
Bus de datos	16 bits.
Bus de programa	24 bits.
Registros de propósito específico	*Result*: Registro acumulador.
	MH: Almacena los MSB de la multiplicación.
	PC: Contador de programa.
	Flag: Almacena las banderas.
	Cero: Almacena el valor cero.
Registros de propósito general	8 registros de 16 bits.

Capítulo 2

Diseño del microprocesador.

Este capítulo muestra la parte central del trabajo realizado, describe el diseño e implementación de una estructura que interprete y ejecute las instrucciones seleccionadas en el capítulo anterior. Para abordar el problema de diseño se realizó un estudio de las principales características de un microprocesador, luego se analizó cada instrucción por separado para definir que componentes se necesitan para su ejecución, posteriormente se estudió el conjunto de instrucciones completo para formar la estructura del microprocesador.

Este capítulo inicia con una descripción de la organización de un microprocesador, luego muestra el esquema planteado en este trabajo y se describe cada uno de los bloques funcionales que conforman dicha estructura. Finalmente, se discute con detalle la ejecución de las instrucciones en la estructura diseñada y se muestran algunos ejemplos de simulación del microprocesador en el *software* ISE 6.1i de Xilinx.

2.1. Organización del microprocesador.

Sin importar su aplicación todo microprocesador está constituido de forma general por un *datapath* y una unidad de control [1].
El *datapath* se encarga de realizar todas las operaciones sobre los datos, es la estructura donde se ejecutan las instrucciones, cuenta con unidades funcionales como sumadores, multiplicadores, operadores lógicos, y además contiene memorias y registros utilizados para el almacenamiento de datos. Todos los bloques del *datapath* están unidos por buses y multiplexores.

La unidad de control por su parte, se encarga de interpretar las instrucciones, provee la señales de control adecuadas según la instrucción que se este ejecutando, es una máquina de estados que opera pasando de un estado a otro dependiendo del estado en que se encuentre y de las señales de entrada.

La figura 2.1[1] muestra el esquema general de un microprocesador, se observa que todos los bloques del *datapath* reciben señales de la unidad de control y a su vez el *datapath* le envía señales indicando su estado actual, para que en base a ellas la unidad de control decida cual será el próximo estado.
En las secciones 2.2 y 2.3 se discute el diseño del *datapath* y la unidad de control planteado en este trabajo.

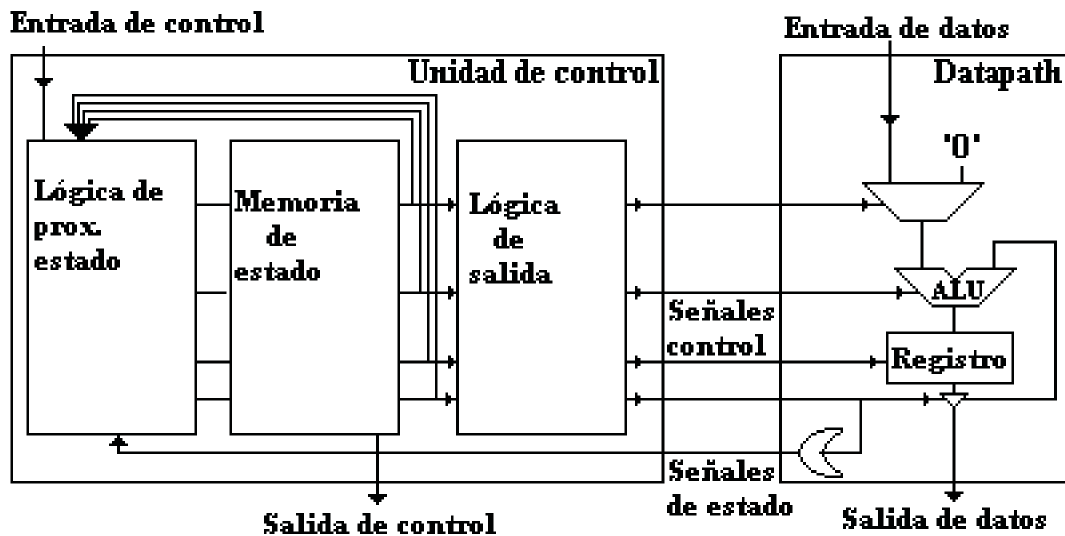


Figura 2.1: Esquema general de un microprocesador.

[1]Tomado de [1].

2.2. Descripción del *datapath*.

Una vez seleccionado el conjunto de instrucciones, se procedió a plantear una estructura que permita su correcta ejecución. Para esto se analizó cada instrucción por separado y se definieron los bloques requeridos para su implementación, luego se unieron los diferentes bloques en una única estructura que ejecute todas las instrucciones.

La figura 2.2 muestra el esquema del *datapath* diseñado en este trabajo, cada boque tiene indicadas sus señales de control, la unidad de control completa no se incluye para facilitar la comprensión de la figura.
Para que las instrucciones se ejecuten correctamente, es necesario que los componentes del *datapath* cumplan con ciertas características. A continuación se muestran las decisiones tomadas en el diseño de cada uno de los bloques funcionales que conforman este *datapath*.

2.2.1. Memorias.

El microprocesador está diseñado con base en una arquitectura *Harvard* [8], es decir, las instrucciones y los datos son almacenados por separado, implementando así en este trabajo una memoria de programa y una memoria de datos.

La memoria de programa no tiene datos de entrada y la señal de control es sólo para lectura dado que el programa almacenado en memoria no puede ser alterado por el microprocesador, sino directamente por el programador. Los datos de salida tienen una longitud de 24 bits y corresponden al formato de la instrucción, la dirección de memoria está dada por el registro contador de programa (PC) que es de 16 bits, por lo tanto se puede almacenar un programa de hasta 64K líneas.

A diferencia de la memoria de programa, la memoria de datos contiene señales de entrada y salida dado que el microprocesador debe leer o almacenar datos; estas señales de datos deben ser de 16 bits. El proceso de lectura o escritura en la memoria es controlado por la misma señal. Igualmente, existe una única dirección para lectura-escritura, esta dirección puede ser dada directamanete por el programador o por el valor del registro *result*, la longitud de la dirección es de 16 bits, por tanto es posible direccionar hasta 64K líneas de la memoria de datos.

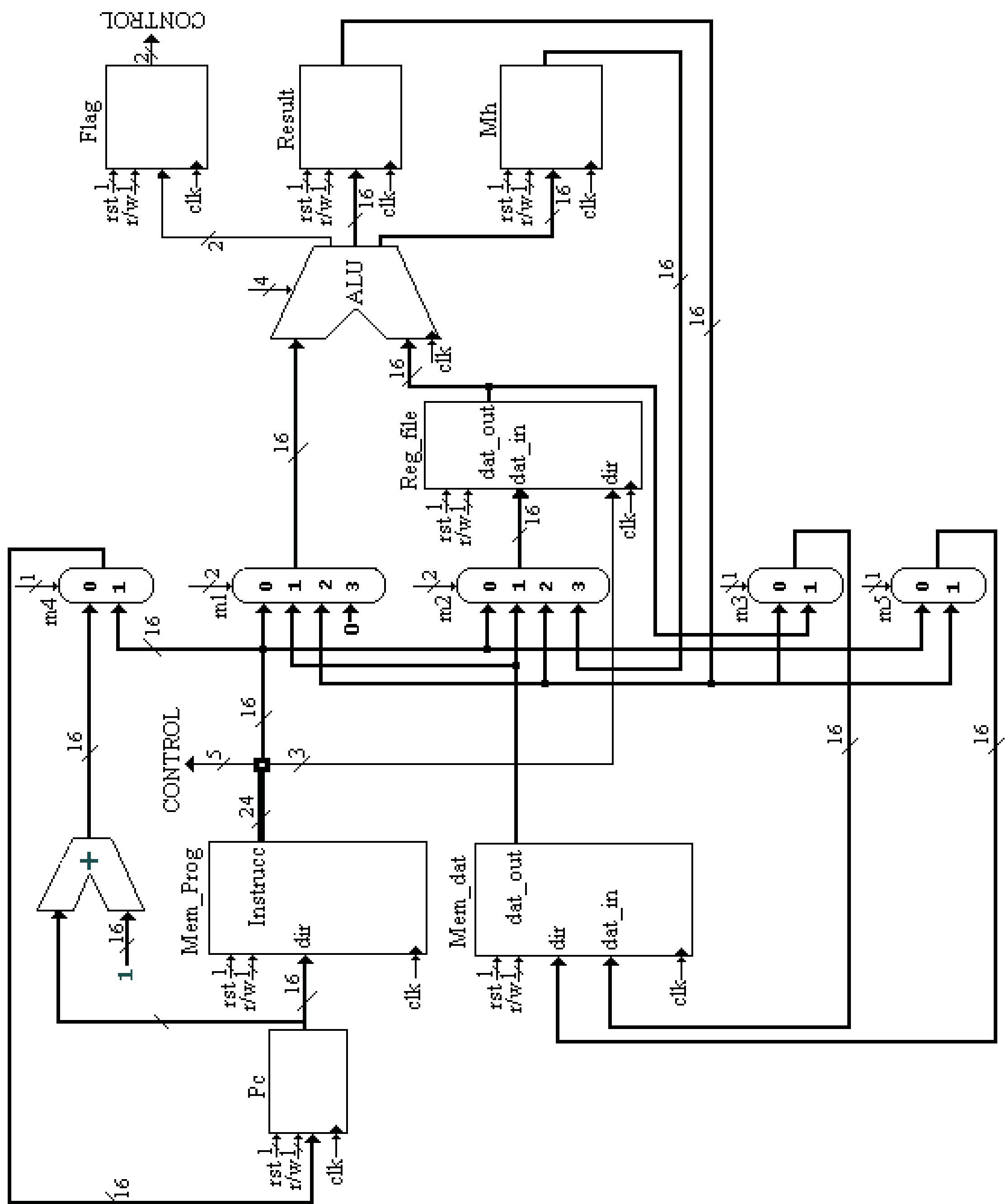


Figura 2.2: Datapath.

2.2.2. Registros.

Los registros son utilizados para almacenar datos que se necesitan durante el proceso de ejecución de las instrucciones. Un registro es un circuito con dos o mas flip-flops conectados de tal manera que todos trabajen exactamente iguales y sincronizados con la misma señal de reloj, cada flip flop es usado para almacenar un bit. En el caso particular de este trabajo los registros tienen una longitud de 16 bits, excepto el registro de banderas que sólo almacena 2 bits. Este diseño cuenta con 8 registros de propósito general y 5 registros de propósito específico.
Los registros de propósito general pueden almacenar datos de diferentes fuentes, organizados como un archivo de 8 registros, este archivo lee o escribe un dato en un ciclo de reloj. La figura 2.3 muestra un esquema del archivo de registros con sus respectivas señales de entrada y de control. Por otro lado, el *datapath* cuenta con 5 registros de

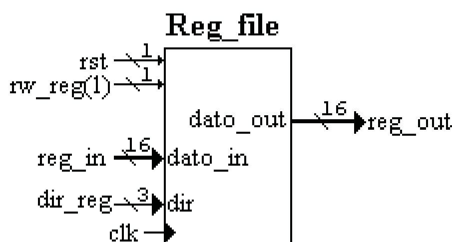


Figura 2.3: Archivo de registros de propósito general.

rst: Permite tener cero a la salida del archivo de registros.

rw_reg(1): Señal de control de lectura / escritura de datos.

dir_reg: Entrada de dirección de 3 bits para seleccionar un registro.

reg_in, reg_out: Señales de entrada y salida de datos de 16 bits.

propósito específico, es decir creados para almacenar datos provenientes de una fuente particular. Los registros son:

***PC*:** Es el contador de programa, almacena la dirección de la memoria de programa de donde se debe leer la instrucción.

***Result*:** Almacena el resultado de la ALU, este registro tiene una función de acumulador, es decir su valor puede ser llevado nuevamente a la primera entrada de la ALU para realizar otra operación.

***Mh*:** Como la salida de la multiplicación es de 32 bits, los 16 bits menos significativos se almacenan en el registro *result* y los más significativos en el registro *Mh*.

Flag: Almacena las banderas de cero y acarreo provenientes de la ALU, este registro debe ser actualizado cada vez que se almacena un nuevo dato en el registro *result*.

Cero: Almacena el valor de cero, es utilizado en la instrucción no opere.

2.2.3. ALU: Unidad Aritmético-Lógica.

La unidad aritmético-lógica es considerada un bloque neurálgico en la determinación del rendimiento del microprocesador, normalmente en este bloque se emplea el mayor tiempo de ejecución de la instrucción pues es aquí donde se realizan las operaciones sobre los datos, por tanto, parte del esfuerzo por mejorar el rendimiento del microprocesador debe ser enfocado hacia el diseño de bloques operacionales rápidos.

Existen diferentes maneras de plantear el diseño de la ALU. En este proyecto se analizaron dos alternativas de diseño de la ALU. La primera consiste en diseñar una ALU de un bit, es decir crear los bloques funcionales (suma, resta, multiplicación, etc.) para un solo bit y luego conectar 16 ALUs de 1 bit [9]. Aunque esta forma de diseño es fácil de entender, se obtienen resultados de bajo desempeño en velocidad dado que no se pude mejorar considerablemente el rendimiento de cada bloque por separado. La otra alternativa de diseño estudiada consiste en realizar por separado los bloques funcionales de 16 bits y mediante un multiplexor seleccionar el resultado adecuado según la instrucción que se este ejecutando; en este trabajo se se hace uso de esta esta forma de diseño, por permitir un mayor desempeño y flexibilidad según la selección de cada bloque funcional.

En su estructura general, la ALU cuenta con dos entradas de 16 bits (2 operandos) y 3 salidas, una corresponde al resultado de la operaciones (registro *result*), la otra a las banderas (registro *flag*) y la tercera a los bits más significativos de la multiplicación (registro *Mh*). En caso que sólo necesite un valor para operar siempre utiliza la primera entrada. La ALU propuesta realiza en total 14 operaciones aritmético-lógicas, además de permitir el paso de la primera o la segunda entrada sin realizar ninguna operación, resultando que la señal de control de la ALU debe ser de 4 bits para que permita seleccionar hasta 16 operaciones.

La figura 2.4 muestra el esquema de la ALU planteado, el primer bloque que se observa es el sumador, en este se ejecutan las instrucciones suma, resta, incrementa y disminuye; para ello la segunda entrada del sumador (*b*) debe cambiar dependiendo de la operación; *b1* es uno en caso que la operación sea incrementar o disminuir, *b2* es el complemento de *b1* en caso que la operación sea resta o disminuir; evitando repetir el bloque sumador para cada instrucción y disminuyendo el número de recursos utilizados. Este bloque genera las banderas cero (*z2*) y acarreo (*c_out*).

El diseño plantea la posibilidad de operar con sólo una entrada (a) y para ello se implementó por separado un bloque para ejecutar la instrucción complemento a2 y otro para las instrucciones de resta y disminuye.

Por otro lado, el diseño plantea que la ALU recibe las señales de control y los datos de entrada y entrega el resultado en un ciclo de reloj.

A continuación se discute con detalle el diseño de cada uno de los bloques operacionales.

Figura 2.4: Esquema general de la ALU.

Sumador.

Dentro de los bloques funcionales de la ALU el sumador es uno de los más importantes, en base a él se ejecutan las instrucciones suma, resta, incrementa y disminuye, requiriendo así el diseño de un sumador de considerable desempeño.

A lo largo del estudio de los sistemas digitales, se han planteado diferentes maneras de mejorar el diseño del sumador tratando de reducir el tiempo de cálculo sin aumentar demasiado su tamaño [10, 11], en este trabajo se analizaron dos modos de diseño, sin profundizar en sumadores especializados del estado del arte. El primero se conoce como *sumador de acarreo propagado*, este realiza la suma dígito por dígito de derecha a izquierda con acarreos pasando al siguiente dígito de la izquierda, como normalmente se hace con los números decimales. Desde el punto de vista de componentes este sumador mantiene las siguientes relaciones lógicas:

$$S_i = (A_i \quad xor \quad B_i) \quad xor \quad C_i \tag{2.1}$$

$$C_{i+1} = (A_i \quad and \quad B_i) \quad or \quad (A_i \quad xor \quad B_i) \quad and \quad C_i \tag{2.2}$$

donde S_i es la suma del bit i del operando A con el bit i del operando B; C_i es el acarreo de entrada y C_{i+1} es el acarreo de propagación. La figura 2.5[2] muestra el esquema de un sumador de acarreo propagado para operandos de cuatro bits. En este sumador se debe esperar el cálculo del acarreo para realizar la siguiente suma, limitando su velocidad por el tiempo de propagación de C_{out}. Sin embargo, tiene la ventaja de su fácil implementación y recursos reducidos.

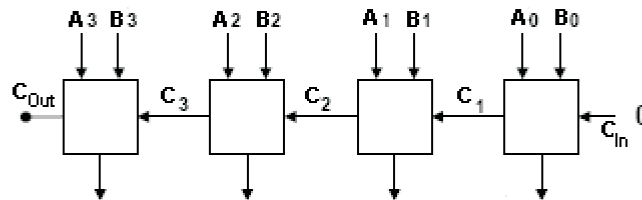


Figura 2.5: Sumador de acarreo propagado de 4 bits.

El otro sumador analizado en este trabajo se conoce como *sumador de acarreo anticipado*, como su nombre lo indica busca anticipar el cálculo del acarreo para que su resultado sea obtenido en menor tiempo.

[2]Tomado de: [10].

$$G_i = (A_i \quad and \quad B_i) \tag{2.3}$$

Si $G_i = 1$, es decir $A_i = 1$ y $B_i = 1$, una celda genera i un arrastre.

$$P_i = (A_i \quad xor \quad B_i) \tag{2.4}$$

Si $P_i = 1$, es decir $A_i = 1$ y $B_i = 1$, una celda i propaga un arrastre.

Al expresar la suma y el acarreo en función de P y G se tiene que:

$$S_i = (P_i \quad xor \quad C_i) \tag{2.5}$$

$$C_{i+1} = G_i \quad or \quad (P_i \quad and \quad C_i) \tag{2.6}$$

Por tanto:

$$\begin{aligned}
C_1 &= G_0 \quad or \quad (P_0 \quad and \quad C_0)\\
C_2 &= G_1 \quad or \quad (P_1 \quad and \quad C_1)\\
&= G_1 \quad or \quad (P_1 \quad and \quad G_0) \quad or \quad (P_1 \quad and \quad P_0 \quad and \quad C_0)\\
C_3 &= G_2 \quad or \quad (P_2 \quad and \quad C_2)\\
&= G_2 \quad or \quad (P_2 \quad and \quad G_1) \quad or \quad (P_2 \quad and \quad P_1 \quad and \quad G_0) \quad or \quad (P_2 \quad and \quad P_1 \quad and \quad P_0 \quad and \quad C_0)\\
&.\\
&.\\
&.
\end{aligned} \tag{2.7}$$

Este sumador aumenta considerablemente el *hardware*, pues conforme aumenta el número de bits el número de términos producto y el número de factores entre ellos crece demasiado. La figura 2.6[3] muestra el esquema de un sumador de *acarreo anticipado* para operandos de 4 bits, notando que cada bloque sumador de un bit no entrega directamente el acarreo, sino el valor de P y G. La unidad de acarreo anticipado se encarga de realizar el cálculo de cada acarreo.

En vista que el sumador de *acarreo anticipado* aumenta considerablemente el *hardware* se decidió no utilizarlo para los 16 bits debido a que el diseño está limitado por el tamaño de la FPGA, entonces se planteó un diseño con cuatro sumadores de cuatro bits con *acarreo propagado* y unirlos mediante *acarreo anticipado*. Por otro lado, se observa que el acarreo de entrada (C_0), para nuestro caso siempre va a ser cero es decir, se elimina el último factor del cálculo de cada acarreo, por lo tanto el tamaño del sumador disminuye.

[3] Tomado de [10].

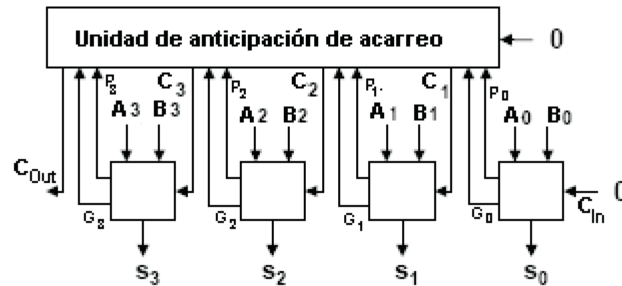


Figura 2.6: Sumador de acarreo anticipado.

Complemento a2.

Es importante que el microprocesador pueda trabajar tanto con números positivos como negativos, el complemento a2 es una forma de representación de números enteros con signo. Este módulo se diseñó pensando en la resta, sin embargo se decidió colocarlo como una instrucción dado que este resultado puede ser útil al programador. De cada número positivo se puede encontrar su negativo por medio del complemento a2, para calcular el complemento a2 se halló primero el complemento a1 y luego se le sumó uno al bit menos significativo. Las siguientes ecuaciones muestran el procedimiento utilizado:

$$Ca1_i = (A_i \quad xor \quad 1) \quad \text{Cálculo del complemento a1.} \tag{2.8}$$

$$Ca2_0 = (Ca1_0 \quad xor \quad 1) \quad \text{Primer complemento a2.} \tag{2.9}$$

$$C_0 = (Ca1_0 \quad and \quad 1) \quad \text{Acarreo inicial.} \tag{2.10}$$

$$Ca2_i = (Ca1_i \quad xor \quad C_{i-1}) \quad \text{Complemento a2.} \tag{2.11}$$

$$C_i = (Ca1_i \quad and \quad C_{i-1}) \quad \text{Acarreo.} \tag{2.12}$$

Resta.

Al igual que la suma, la resta es una operación básica por tanto debe ser incluida en el conjunto de instrucciones. La resta es una suma en la que uno de los operandos debe cambiar de signo, por lo tanto para realizar este bloque se utilizaron el sumador y el complemento a2 que se diseñaron con anterioridad. En el esquema de este *datapath* la primera entrada de la ALU conserva siempre su signo y la segunda entrada cambia de signo cuando sea necesario restar.

La figura 2.8 muestra la combinación de los bloques sumador y complemento a2 utilizados en este diseño para realizar la operación de resta.

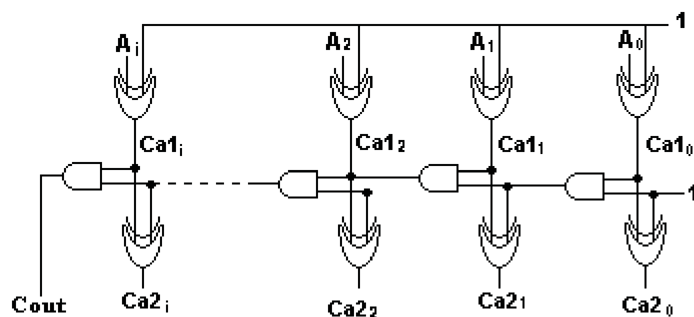


Figura 2.7: Circuito lógico implementado para calcular el complemento a2.

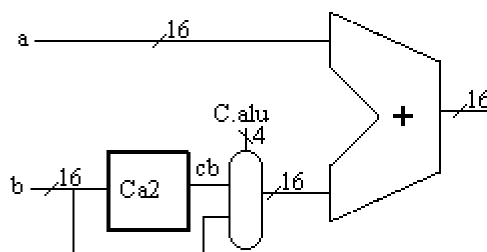


Figura 2.8: Restador.

Multiplicador.

El bloque multiplicador es de gran importancia dado que es la base de los sistemas de procesamiento de datos que se aplican en áreas como las comunicaciones, el procesamiento de imagen, control, entre otras. Por otro lado, el diseño de este procesador es un preliminar del procesador de señales que utilizará el sistema de comunicación inalámbrico que se esta diseñando en la $E3T$, por tanto se decidió incluir la multiplicación en este trabajo. Sin embargo el multiplicador es el bloque funcional que más espacio ocupa dentro del *datapath*, es por esto que normalmente no es implementado en la mayoría de microprocesadores de propósito general, además, en caso que no se necesite una alta velocidad la multiplicación puede ser obtenida a partir de otras operaciones.

Existen diferentes maneras de implementar un bloque multiplicador. En este trabajo se analizaron dos técnicas. La primera se conoce como *add-shift* [9, 11], esta se basa en el proceso de multiplicación normalmente usado para números decimales, aunque esta técnica es fácil de implementar su desempeño es reducido.

La otra técnica estudiada se conoce como *arreglo celular de multiplicadores* [12]. Se decidió implementar esta técnica debido a la velocidad de operación y a la facilidad para implementar la multiplicación con signo. Se seleccionó un tipo de multiplicador que permite desarrollar directamente la multiplicación de números en complemento a2, esto significa un aumento en la velocidad del proceso de multiplicación dado que no es necesario realizar una etapa previa para la operación complemento a2.

A continuación se describe el proceso desarrollado para realizar esta multiplicación hasta llegar al algoritmo de *Baugh-Wooley* que fue el que finalmente se implementó[4].

Para multiplicar directamente números en complemento a2 se tiene que:

Sean $A = (a_{n-1}, a_{n-2}, ..., a_1, a_0)$ y $B = (b_{m-1}, b_{m-1}, ..., b_1, b_0)$, dos números expresados en complemento a2, por tanto tienen un valor de:

$$V(A) = -a_{n-1} * 2^{n-1} + \sum_{j=0}^{n-2} a_j * 2^j \tag{2.13}$$

$$V(B) = -b_{m-1} * 2^{m-1} + \sum_{i=0}^{m-2} b_i * 2^i \tag{2.14}$$

siendo n y m la longitud de los vectores A y B respectivamente.
La multiplicación $A * B$ está dada por:

$$\begin{aligned} A * B = &\sum_{i=0}^{m-2}\left(\sum_{j=0}^{n-2} a_j * b_i * 2^{i+j}\right) - \sum_{i=0}^{m-2} a_{n-1} * b_i * 2^{n-1+i} \\ &- \sum_{j=0}^{n-2} a_j * b_{m-1} * 2^{m-1+j} + a_{n-1} * b_{m-1} * 2^{n+m-2} \end{aligned} \tag{2.15}$$

Por ejemplo para multiplicar dos números de 5 bits (a, b) se tiene que:

[4]Esta descripción se adaptó de las referencias [10, 12]

$$
\begin{array}{cccccccccc}
 & & & & & (a_4) & a_3 & a_2 & a_1 & a_0 \\
 & & & & * & (b_4) & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & (a_4b_0) & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & (a_4b_1) & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & (a_4b_2) & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & (a_4b_3) & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
 & a_4b_4 & (a_3b_4) & (a_2b_4) & (a_1b_4) & (a_0b_4) & & & & \\
\hline
P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
\tag{2.16}
$$

Donde los términos entre paréntesis aparecen con con peso negativo y P es el producto de la multiplicación.

Este modelo se conoce como *multiplicador de Pezaris*, se observa que es necesario implementar diferentes tipos de sumadores para cada suma de productos parciales.

La Tabla 2.1 define cada tipo de sumador, en forma general para multiplicar dos números de n bits, es necesario implementar $(n-2)^2$ sumadores tipo 0, $(n-2)$ sumadores tipo 1, $(2n-3)$ sumadores tipo 2 y un sumador tipo 3 [12].

Tabla 2.1: Tipos de sumadores.

Tipo	0	1	2	3
Operación	x y $+)z$ -- cs	x y $+)-z$ -------- $c(-s)$	$-x$ $-y$ $+)z$ -------- $(-c)s$	$-x$ $-y$ $+)-z$ -------- $(-c)(-s)$

* x, z son las entradas, y es el acarreo de entrada, c es el acarreo de salida y s es el resultado de la suma.

La figura 2.9[5] muestra la estructura implementada para realizar la multiplicación descrita en el arreglo 2.16.

[5]Tomado de [10]

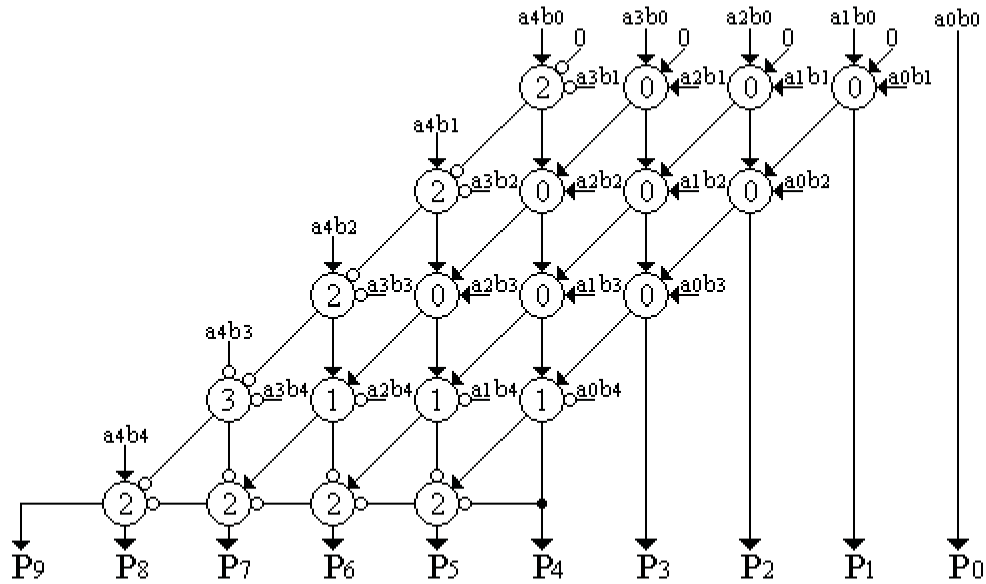


Figura 2.9: Multiplicador de Pezaris para dos números de 5 bits.

Una variación del *multiplicador de Pezaris* consiste en separar los sumandos negativos y positivos logrando un diseño más uniforme. Reagrupando el arreglo 2.16 se tiene que:

$$
\begin{array}{cccccccccc}
 & & & & (a_4) & a_3 & a_2 & a_1 & a_0 & \\
 & & & * & (b_4) & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
a_4b_4 & 0 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
 & (a_4b_3) & (a_4b_2) & (a_4b_1) & (a_4b_0) & & & & & \\
 & (a_3b_4) & (a_2b_4) & (a_1b_4) & (a_0b_4) & & & & & \\
\hline
P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
\tag{2.17}
$$

La figura 2.10[6] muestra la nueva implementación del multiplicador de *Pezaris* basado

[6]Tomado de [10]

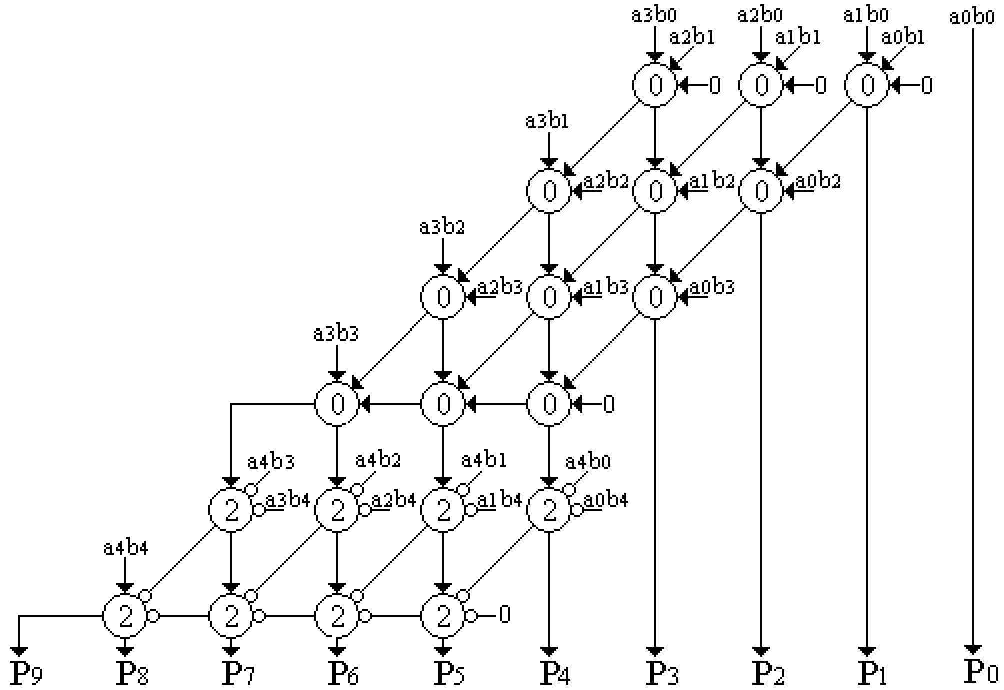


Figura 2.10: Variación del multiplicador de Pezaris.

en el arreglo 2.17, observando que sólo es necesario utilizar sumadores tipo 0 y 2.

El multiplicador *Baugh-Wooley* propone un algoritmo donde todos los sumandos son positivos, esto permite construir el multiplicador sólo con sumadores tipo 0 haciendo la estructura más uniforme.
El algoritmo *Baugh-Wooley* para la multiplicación de $n * m$ bits plantea que el vector:

$$0 \quad 0 \quad a_{m-2}b_{n-1} \quad a_{m-3}b_{n-1} \ldots a_0 b_{n-1} \tag{2.18}$$

Puede ser reemplazado por los vectores:

$$\begin{array}{cccccc} 0 & \bar{b}_{n-1} & \bar{a}_{m-2}b_{n-1} & \bar{a}_{m-3}b_{n-1} \ldots \bar{a}_0 b_{n-1} \\ 1 & 1 & 0 & 0 \quad\quad \ldots \quad b_{n-1} \end{array} \tag{2.19}$$

A su vez el vector

$$0 \quad 0 \quad a_{m-1}b_{n-2} \quad a_{m-1}b_{n-3} \ldots a_{m-1}b_0 \tag{2.20}$$

Es reemplazado por:

$$\begin{array}{cccccc} 0 & \bar{a}_{m-1} & a_{m-1}\bar{b}_{n-2} & a_{m-1}\bar{b}_{n-3} & \ldots & a_{m-1}\bar{b}_{0} \\ 1 & 1 & 0 & 0 & \ldots & a_{m-1} \end{array} \tag{2.21}$$

El segundo vector de la ecuación 2.19 puede ser combinado con el segundo vector de la ecuación 2.21 para obtener el vector

$$\begin{array}{ccccccc} 1 & 0\ldots0 & a_{m-1} & 0\ldots0 & b_{n-1} & 0\ldots0 \\ \uparrow & & \uparrow & & \uparrow & \uparrow \\ p_{m+n-1} & & p_{m-1} & & p_{n-1} & p_{0} \end{array} \tag{2.22}$$

Entonces, aplicando la ecuación 2.22 en un multiplicador de $m*n$ bits agrupado como se muestra en la ecuación 2.11, la multiplicación *Baugh-Wooley* se define como:

$$\begin{array}{ccccccccccccc}
 & & & & & & & & a_{m-1} & a_{m-2}\ldots a_{3} & a_{2} & a_{1} & a_{0} \\
 & & & & & & & & * & b_{n-1}\ \ldots & b_{2} & b_{1} & b_{0} \\
\hline
 & & & & & a_{m-2}b_{0} & & & \ldots & & a_{2}b_{0} & a_{1}b_{0} & a_{0}b_{0} \\
 & & & & a_{m-2}b_{1} & & & & \ldots & & a_{1}b_{1} & a_{0}b_{1} & \\
 & & & a_{m-2}b_{2} & & & & & \ldots & & a_{0}b_{2} & & \\
 & & & \cdot & & & & & & \cdot & & & \\
 & & & \cdot & & & & & & \cdot & & & \\
 & & & \cdot & & & & & & \cdot & & & \\
 & a_{m-1}b_{n-1} & 0 & a_{m-2}b_{n-2} & \ldots & & & a_{2}b_{n-2} & a_{1}b_{n-2} & a_{0}b_{n-2} & & & \\
 & \bar{a}_{m-1} & \bar{a}_{m-2}b_{n-1} & \bar{a}_{m-3}b_{n-1} & \ldots & & \bar{a}_{2}b_{n-1} & \bar{a}_{1}b_{n-1} & \bar{a}_{0}b_{n-1} & & & & \\
 & \bar{b}_{n-1} & a_{m-1}\bar{b}_{n-2} & a_{m-1}\bar{b}_{n-3}\ldots a_{m-1}\bar{b}_{1} & a_{m-1}\bar{b}_{0} & & & & & & & & \\
1 & & & & a_{m-1} & & & b_{n-1} & & & & & \\
\hline
p_{m+n-1} & p_{m+n-2} & p_{m+n-3} & p_{m+n-4} & \ldots & p_{m-1} & \ldots & p_{n-1} & & & p_{2} & p_{1} & p_{0}
\end{array} \tag{2.23}$$

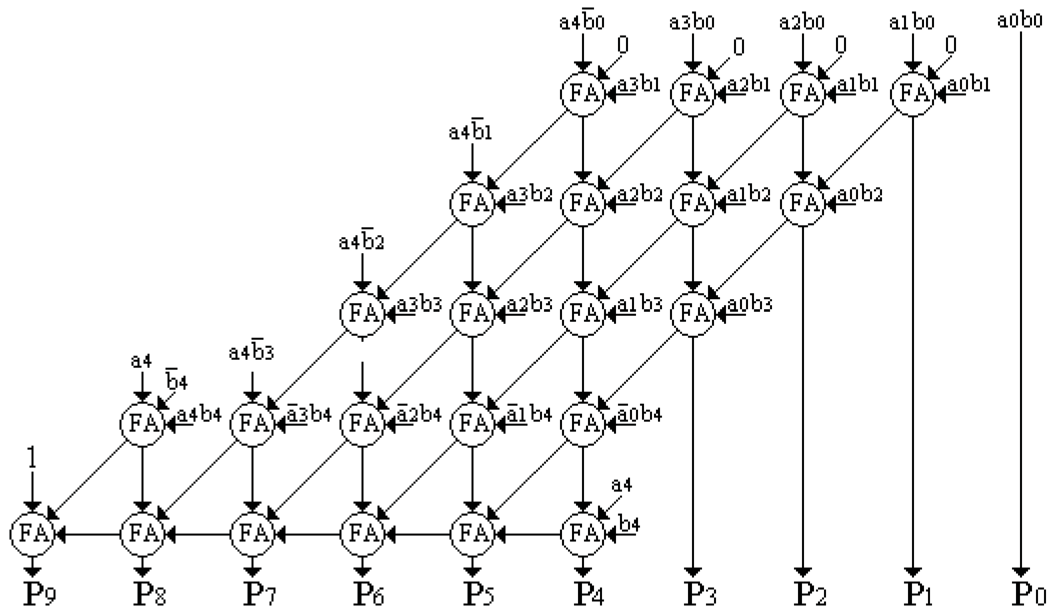


Figura 2.11: Multiplicador de Bawgh-Wooley para dos números de 5 bits.

La figura 2.11[7] basada en el arreglo 2.23 muestra el esquema del multiplicador *Baugh-Wooley* implementado para dos números de 5 bits. FA hace referencia a sumadores completos (*full-adder*), es decir sumadores con acarreo de entrada. En general un multiplicador *Baugh-Wooley* de $n * m$ bits requiere $m(n-1)+3$ *full-adders* para su implementación.
Se observa que las entradas a los sumadores son operaciones *and* y que algunos de los operandos deben ser negados.

La multiplicación es una operación requerida en diferentes aplicaciones, especialmente es la base de los sistemas de procesamiento de señales que por sus características de operación en tiempo real requieren un elevado desempeño. Es por esto que en la actualidad se siguen desarrollando una gran cantidad de diseños de multiplicadores buscando aumentar su velocidad de operación. Sin embargo, el objeto de este trabajo es mostrar la estructura general de un microprocesador sin profundizar en la elaboración de cada bloque funcional, para mas detalles sobre tipos de multiplicadores se recomienda acudir a las referencias [12].

[7]Tomado de [12]

Operaciones de desplazamiento.

Las operaciones de desplazamiento le permiten al programador manipular los bits almacenados en el registro acumulador (*result*). La ALU realiza cuatro operaciones que permiten desplazar o rotar un bit a la derecha o izquierda en un único bloque, como se observa en la figura 2.4.

El desplazamiento se realiza agregando ceros en un extremo, cada vez que se desplaza se pierde el bit del extremo opuesto. Por su parte las instrucciones de rotación no pierden el bit sino que lo insertan en la otra esquina en lugar de los ceros. Este bloque se implementó de manera comportamental, mediante la opción de concatenación (&) existente en VHDL. La figura 2.12 muestra las operaciones desplazamiento y rotación.

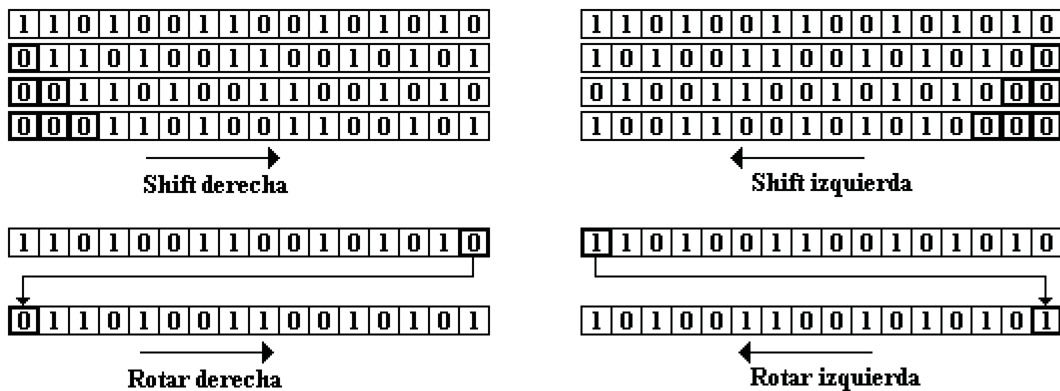


Figura 2.12: Operaciones shift y rotación.

Operaciones lógicas.

Las operaciones lógicas tienen gran importancia en el diseño del microprocesador, le brindan flexibilidad al programador permitiendo la implementación de nuevas operaciones, además de ser útiles en tareas de comparación.

EL estándar VHDL'93 [13] tiene implementadas las operaciones lógicas básicas en su paquete *standar_logic*, de esta forma la implementación no es requerida. El microprocesador cuenta con las 5 operaciones lógicas básicas: NOT, AND, OR y XOR, estos bloques reciben y generan operandos de 16 bits. Con las operaciones lógicas termina la descripción del diseño de la ALU, la tabla 2.2 muestra un resumen de las operaciones realizadas con su respectiva señal de control.

Tabla 2.2: Operaciones de la ALU con sus respectivas señales de control.

Operación	C_alu
Suma	0000
Resta	0001
Complemento a2	0010
Multiplicación	0011
Shift derecha	0100
Shift izquierda	0101
Rotar derecha	0110
Rotar derecha	0111
NOT	1000
AND	1001
OR	1010
XOR	1011
Inc 1	1100
Dis 1	1101
a	1110
b	1111

2.2.4. Sumador del contador de programa ($PC + 1$).

Además del sumador diseñado en la ALU, existe un bloque dedicado únicamente a aumentar en uno la dirección de la instrucción ($Pc + 1$) con el fin de hacer el cálculo inmediato de la dirección de la siguiente instrucción y facilitar el posterior diseño del *pipeline*. Este sumador fue diseñado con *acarreo propagado* pues utiliza menos hardware y la velocidad en este caso no es una característica crítica. La figura 2.13 muestra el proceso de cálculo de la siguiente instrucción, la primera entrada del sumador es el valor del registro Pc, la segunda entrada siempre es uno.

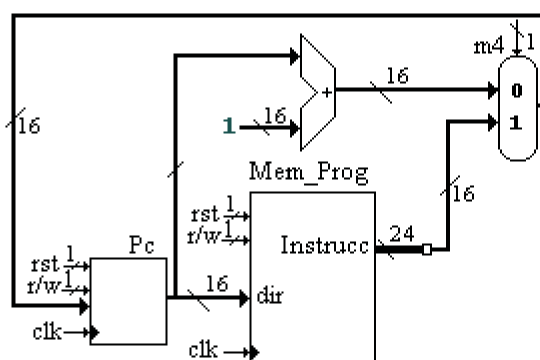


Figura 2.13: Bloque sumador de programa.

2.2.5. Multiplexores.

Si se analiza con detenimiento cada instrucción, se puede observar que existen diferentes posibilidades para la entrada de datos en algunos módulos, por ejemplo el archivo de registros puede almacenar un dato proveniente del formato de instrucción, de la memoria de datos, del registro *result* o del registro *Mh*. Para estos casos se diseñaron multiplexores que mediante una señal de control seleccionan la entrada apropiada según la instrucción que se esté ejecutando.

La figura 2.2 muestra los 5 multiplexores utilizados en este diseño, el primero de estos corresponde a primera la entrada de la ALU, esta entrada puede recibir un dato proveniente del formato de instrucción, de la memoria de datos, del registro *result* o del registro *cero*, como son cuatro entradas posibles la señal de control debe ser de 2 bits. El segundo corresponde a la entrada del archivo de registro mencionado anteriormente, también debe tener 2 bits de control. Un tercer multiplexor se encuentra a la entrada de la memoria de datos, dado que esta puede almacenar un dato proveniente del registro *result* o del archivo de registro, como son dos entradas la señal de control debe ser de un bit. El cuarto multiplexor se encuentra a la entrada del registro *PC*, este puede recibir un dato directamente del sumador de programa o en caso de saltos un dato proveniente del formato de instrucción, como son dos posibles entradas la señal de control es de un bit. Por último, existe un multiplexor que selecciona la dirección de almacenamiento en la memoria de datos, esta puede provenir del formato de instrucción o del registro *result*.

La descripción de estos multiplexores en VHDL se hizo de manera comportamental. Los multiplexores se diseñaron para que reciban la entrada y las señales de control y entreguen la salida en un ciclo de reloj.

2.3. Diseño de la unidad de control.

En la sección anterior se describieron las señales de control de cada bloque funcional, la unidad de control se encarga de entregar estas señales en el momento indicado dependiendo de la instrucción que se este ejecutando [14].

Como se muestra en la figura 2.14, la unidad de control recibe los cinco bits del código de operación y el valor de las banderas, además debe tener como entradas la señal de reloj y una señal de reset; sus salidas son las señales de lectura/escritura (r/w) y reset (rst) para cada uno de los registros y para las memorias, además la señales de control de la ALU y los multiplexores.

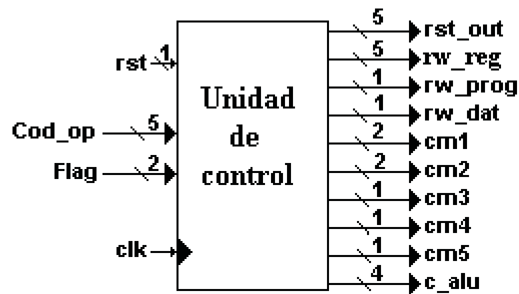


Figura 2.14: Unidad de control.

Las señales de reset y lectura/escritura de todos los registros se agruparon en dos vectores (*rst_out*, *rw_reg*) con el fin de facilitar el diseño en VHDL, la tabla 2.3 muestra la descripción de estos vectores. En el diseño se planteó que los valores de las señales de lectura/escritura (*rw_reg*) de todos los registros y las memorias sean iguales, en este caso se decidió que el valor bajo (0) corresponde a escritura y el valor alto (1) a lectura.

Tabla 2.3: Vectores *rst_out* y *rw_reg*.

Posición (*rst_out* *rw_reg*)	Registro	Descripción del registro
0	Result	Almacena los resultados de la ALU.
1	Reg_file	Archivo de registros de propósito general.
2	Mh	Almacena los bits más significativos de la multiplicación.
3	Pc	Almacena la dirección de la memoria de programa.
4	Flag	Almacena las banderas cero y acarreo.

2.3.1. Estados de la instrucción.

La unidad de control es una máquina de estados que cambia de acuerdo a la instrucción que se este ejecutando, en total se definieron cinco estados. Con el propósito de mostrar por separado cada paso de la ejecución de las instrucciones y de facilitar la implementación de la arquitectura *pipeline*, se decidió realizar una ejecución multiciclo, es decir cada uno de los estados se ejecuta en un ciclo de reloj. A continuación se definen los estados que ejecutará el microprocesador.

Reset: Este es el estado de inicio de un programa, si la señal *rst* esta activa entonces habilita las señales de reset en todos los registros (*rst_out*) del *datapath*, de lo contrario activa la lectura del contador de programa para leer la primera instrucción.

Lectura: En este estado se activan las señales de lectura de todos los registros (rw_reg)y las memorias (rw_prog, rw_dat), excepto el registro Pc que debe ser desactivado para evitar que envíe otra dirección a la memoria antes que la instrucción en curso termine de ejecutarse. Además el sumador de programa calcula la dirección de la siguiente instrucción.

Decodificación: En este estado se seleccionan los datos sobre los que se debe operar para lo cual se activan las señales de control de los multiplexores ($cm1$, $cm2$, $cm3$, $cm4$, $cm5$) dependiendo de la instrucción que se este ejecutando.

Ejecución: La ALU realiza la operación que se requiera, para esto se activan las señales de control de la ALU (c_alu).

Almacenamiento: En este estado se activan las señales de escritura para (rw_reg) los registros. La instrucción no opere no cambia el valor de ningún registro, por lo tanto no pasa por este estado.

2.3.2. Ejecución de las instrucciones.

Para comprender mejor el diseño del *datapath* y la unidad de control se debe analizar el flujo de datos de cada instrucción y determinar que señales deben estar activas en cada momento.
Las instrucciones se ejecutan siguiendo una serie de pasos (estados) determinados por la unidad de control, cada estado se ejecuta en un ciclo de reloj y utiliza diferentes bloques funcionales del microprocesador.
A continuación se analiza con detalle los pasos seguidos por cada tipo de instrucción.

Instrucciones de transferencia de datos.

La ejecución de una instrucción inicia en el estado *erst*, para lo cual se lee el registro PC ($rw_reg(3) = 1$) y este envía la dirección de la instrucción a la memoria de programa.
En el estado *e0* (lectura) es leída la instrucción de la memoria de programa ($r_prog = 1$), además se leen la memoria de datos ($r_dato = 1$) y todos los registros del *datapath* ($rw_reg = 10111$) excepto el registro PC que se prepara para escribir la dirección de la siguiente instrucción. Estos dos primeros estados son comunes a todas las instrucciones.
En el estado *e1* (decodificación) se selecciona el dato que va a ser transferido, para esto se utilizan los multiplexores $m1$, $m2$ o $m3$ y $m5$ en caso que se deba cargar un dato en el registro *result*, en el archivo de registros o almacenar en memoria respectivamente.
La figura 2.15 muestra la ejecución de la instrucción carga dato inmediato en el registro *result*, entonces la entrada del multiplexor *m1* es 0 ($cm1 = 00$).

En el estado *e1* también se debe indicar la dirección de la siguiente instrucción, excepto en las instrucciones de salto siempre se toma la siguiente dirección ($PC + 1$) por lo tanto se selecciona la entrada cero del multiplexor $m4$ ($cm4 = 0$).
En el estado *e2* (ejecución) se define la operación de la ALU, este estado solo es necesario para las instrucciones de transferencia que utilizan el registro *result*, por tanto las instrucciones de carga en el archivo de registros y de almacenamiento pasan directamente al estado *e3*. Para la instrucción particular de la figura 2.15, la unidad aritmético-lógica debe permitir que el dato pase sin ser modificado ($c_alu = 1110$).
Finalmente, en el estado *e3* (escritura) se debe almacenar el dato en el registro correspondiente. Para el ejemplo de la figura 2.15 el dato se almacena en el registro *result* ($rw_reg(0) = 0$) y los resultados de las banderas en el registro *flag* ($rw_reg(4) = 0$).
Las instrucciones carga un valor de memoria en el archivo de registros y carga un valor

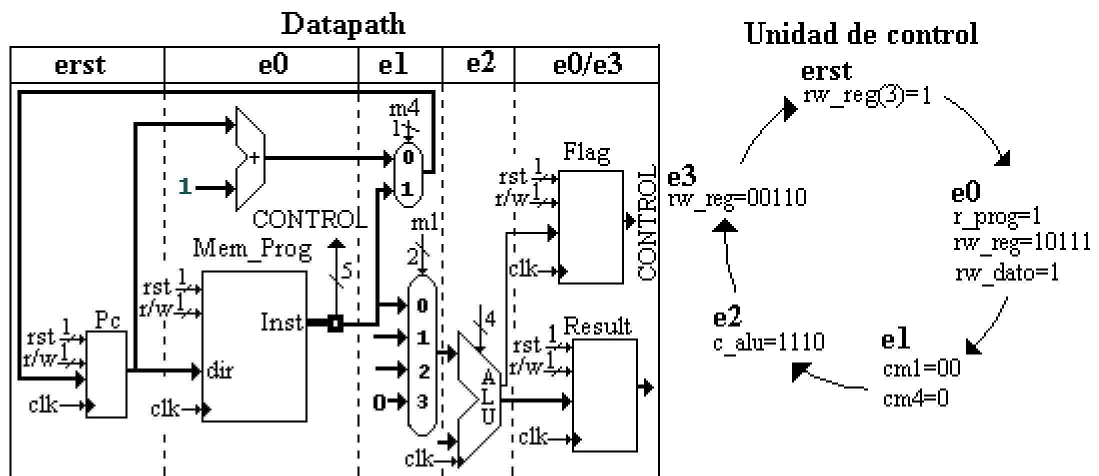


Figura 2.15: Carga dato inmediato en el registro result.

de memoria en el registro *result* tienen algunas diferencias en la forma de ejecución con respecto a las otras instrucciones de transferencia. Para estas dos instrucciones la dirección de la memoria de datos se calcula en el estado de decodificación (*e1*) cuando se activa la señal de control del multiplexor 5 ($cm5$), por lo tanto el nuevo valor de la memoria de datos se obtiene un ciclo después. Sin embargo las señales de control de los multiplexores uno y dos también se obtienen en el estado de decodificación, es decir que el dato de la memoria de programa que se obtiene en este ciclo no es correcto, en el siguiente ciclo la salida de los multiplexores será la adecuada. Esto se soluciona debido a que en el estado de escritura se almacena el valor del dato actual (no el del ciclo anterior), por tanto cuando la instrucción pase por el estado $e3$ el dato ya es correcto.

Instrucciones de tratamiento de datos.

Una vez leídos los registros y las memorias (estado *e0*), en el estado de decodificación (*e1*) las instrucciones de tratamiento de datos deben seleccionar las entradas sobre las que opera la ALU. Este diseño plantea el uso del registro *result* como acumulador, es decir que almacena el dato proveniente de la ALU y a su vez coloca el resultado nuevamente como una entrada a la ALU, por lo tanto para las instrucciones de tratamiento de datos el multiplexor *m1* debe seleccionar siempre su tercera entrada (*cm1=10*). En caso que la instrucción requiera trabajar con dos operandos, la segunda entrada de la ALU está dada por el archivo de registros. En el estado de ejecución (*e2*) la unidad de control envía las señales de control de la ALU y esta ejecuta la operación seleccionada.
La figura 2.16 muestra el proceso de ejecución de una multiplicación, para esto, la señal de control de la ALU se definió como *cm_alu=0011*. Finalmente, en el estado de escritura (*e3*) se almacenan los resultados de la ALU. La multiplicación es un caso particular de las instrucciones de tratamiento de datos pues el resultado de multiplicar dos números de 16 bits es un número de 32 bits, por lo tanto se decidió almacenar los 16 bits menos significativos en el registro *result* (*rw_reg(0)=0*) y agregar un registro (*Mh*) para almacenar los 16 bits más significativos (*rw_reg(2)=0*), por otro lado todas las instrucciones de tratamiento de datos requieren actualizar el registro banderas (*rw_reg(4)=0*).

Instrucciones de control.

Para el caso de las instrucciones de salto, su objetivo es cambiar el valor del registro *PC*, por lo tanto en el estado *e1* el multiplexor *m4* puede seleccionar su segunda entrada. La figura 2.17 muestra la ejecución de la instrucción de salto incondicional (*JMP*), por lo tanto en este estado se indica *cm4=1*, sin embargo las instrucciones de salto condicional salte si es cero (*Jz*) y salte si es negativo (*Jn*) deben tener en cuenta la lectura del registro de banderas, es decir si $flag(0)$ o $flag(1)$ respectivamente estan activas, el multiplexor m4 selecciona su segunda entrada. En el estado de escritura se almacena la salida del multiplexor m4 en el registro PC.

En el caso de la instrucción no opere (NOP), en el estado $e1$ el multiplexor $m1$ selecciona el registro $cero$, en el estado $e2$ se selecciona la multiplicación como operación de la ALU ($c_alu = 0011$), esta instrucción no almacena ningún valor, por lo tanto no para por el tercer estado.

La tabla 3.3 resume todos los estados y las señales de control para cada instrucción. Los estados reset y lectura son comunes a todas la instrucciones. Desde el estado de decodificación en adelante las señales de control dependen de la instrucción que se este ejecutando.

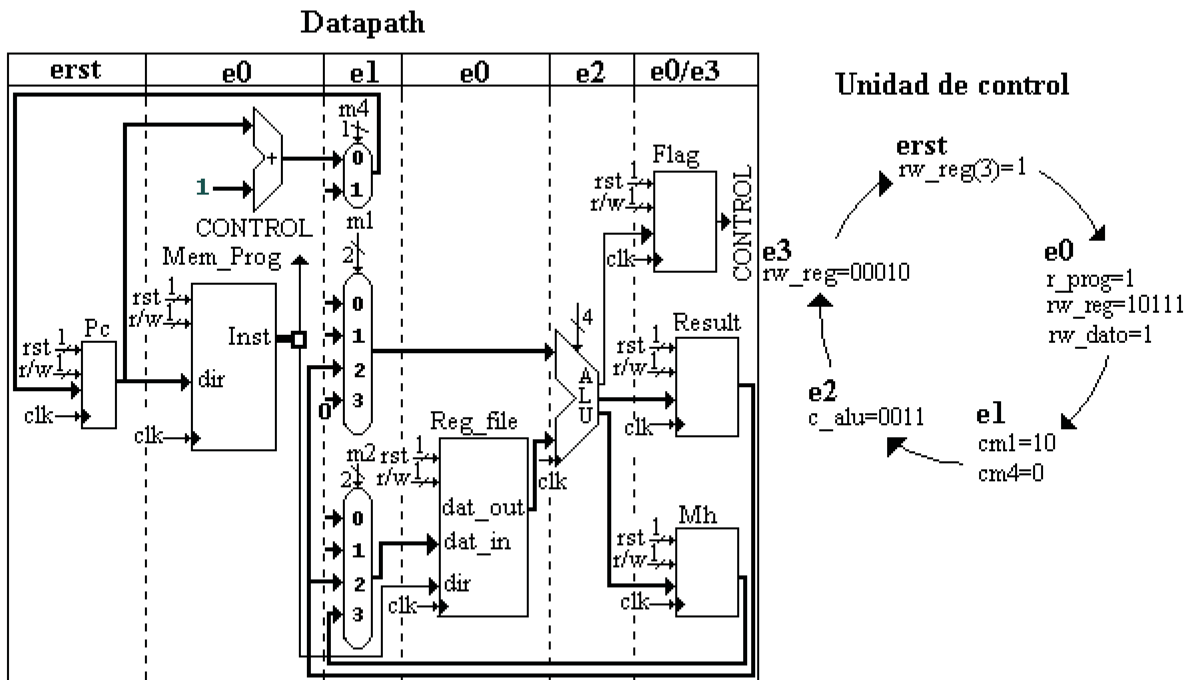


Figura 2.16: Ejecución de la multiplicación.

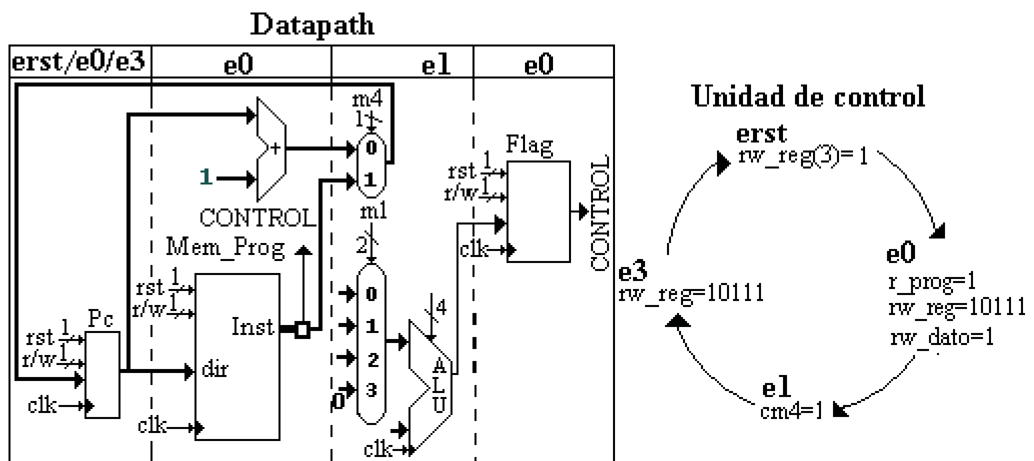


Figura 2.17: Ejecución de la instrucción salto incondicional.

Tabla 2.4: Señales de la unidad de control.

Reset ($erst$)	Lee ($e0$)	Decodifica ($e1$) Cod_op		Ejecuta ($e2$)	Almacena ($e3$)	Instrucción

if rst=1:
rst_out=11111
 	
else rst=0:
rst_out=00000
rw_reg(3)=1
	rw_prog=1
	rw_dat=1
	rw_reg=10111
		00000
		$R = di$
			cm1=00
			cm4=0
				c_alu=1110	rw_reg=00110	$R = di$

		00001 $R = F_i$	cm4=0	c_alu=1111	rw_reg=00110	$R = F_i$
		00010 $R = M[di]$	cm1=01 cm5=0 cm4=0	c_alu=1110	rw_reg=00110	$R = M[di]$
		00011 $F_i = di$	cm2=00 cm4=0		rw_reg=10101	$F_i = di$
		00100 $F_i = R$	cm2=10 cm4=0		rw_reg=10101	$F_i = R$
		00101 $F_i = M[R]$	cm2=01 cm5=1 cm4=0		rw_reg=10101	$F_i = M[R]$
		00110 $F_i = mh$	cm2=11 cm4=0		rw_reg=10101	$F_i = Mh$
		00111 $M[di] = R$	cm3=0 cm5=0 cm4=0		rw_dat=0	$M[di] = R$
		01000 $M[R] = F_i$	cm3=1 cm5=1 cm4=0		rw_dat=0	$M[R] = F_i$
		01001 $Sum[R, F_i]$	cm1=10 cm4=0	c_alu=0000	rw_reg=00110	$Sum[R, F_i]$
		01010 $Res[R, F_i]$	cm1=10 cm4=0	c_alu=0001	rw_reg=00110	$Res[R, F_i]$
		01011 $Mult[R, F_i]$	cm1=10 cm4=0	c_alu=0011	rw_reg=00010	$Mult[R, F_i]$
		01100 C_a2	cm1=10 cm4=0	c_alu=0001	rw_reg=00110	C_{a2}
		01101 $Xor[R, F_i]$	cm1=10 cm4=0	c_alu=1011	rw_reg=00110	$Xor[R, F_i]$
		01110 $And[R, F_i]$	cm1=10 cm4=0	c_alu=1001	rw_reg=00110	$And[R, F_i]$
		01111 $Or[R, F_i]$	cm1=10 cm4=0	c_alu=1010	rw_reg=00110	$Or[R, F_i]$
		10000 NOT	cm1=10 cm4=0	c_alu=1000	rw_reg=00110	Not
		10001 SD	cm1=10 cm4=0	c_alu=0100	rw_reg=00110	SD
		10010 SI	cm1=10 cm4=0	c_alu=0101	rw_reg=00110	SI
		10011 RD	cm1=10 cm4=0	c_alu=0110	rw_reg=00110	RD
		10100 RI	cm1=10 cm4=0	c_alu=0111	rw_reg=00110	RI
		10101 INC	cm1=10 cm4=0	c_alu=1100	rw_reg=00110	INC
		10110 DIS	cm1=10 cm4=0	c_alu=1101	rw_reg=00110	DIS
		10111 JMP	cm4=1		rw_reg=10111	JMP
		11000 Jz	cm4=1 (z) cm4=0 (nz)		rw_reg=10111	Jz
		11001 Jn	cm4=1 (n) cm4=0 (nn)		rw_reg=10111	Jn
		11010 NOP	cm1=11 cm4=0	c_alu=0011		NOP

2.4. Simulación.

Para comprender mejor el funcionamiento del microprocesador, en esta sección se muestra la simulación de algunas instrucciones utilizando el *software* ISE 6.1i de Xilinx. En el capítulo 4 se realiza un análisis más completo de los resultados obtenidos en simulación.

Se analizará por separado el proceso de una multiplicación y de la carga de un valor de memoria en el registro *result*.
El primer programa a simular es la multiplicación de dos números, la tabla 2.5 muestra la descripción de la memoria que contiene estas instrucciones.

Tabla 2.5: Programa 1

PC	Inst	Cod_op	Dir_reg	Dato-Dir	Decimal	Resultado
0	I_0	00000	000	0111110000100100	31780	$R = 31780$
1	I_1	00011	000	0101000110110011	1593779	$F_0 = 20915$
2	I_2	01011	000	0000000000000000	5767168	$R = 12588$ $Mh = 10142$

La figura 2.18 corresponde a la ejecución de estas instrucciones en el *software* ISE 6.1i, los resultados se muestran en notación decimal. El signo '?' corresponde a vectores que no están especificados completamente, el signo 'U' hace referencia a valores no especificados.
El programa emplea 15 ciclos de reloj en ejecutarse. En el primer ciclo se activa la señal de reset ($rst = 1$) haciendo que el microprocesador inicie la ejecución de las instrucciones mediante la activación de la unidad de control. Los cinco ciclos siguientes (2-6) corresponden a la ejecución de la instrucción carga dato inmediato en el registro *result* (I_0). El ciclo dos corresponde al estado $erst$, en este se activa la señal de lectura del registro PC ($srw =? \Rightarrow U1UUUU$), el valor de PC es cero ($spc = 0$) pues en el ciclo anterior se activó la señal rst para todos los registros. El tercer ciclo corresponde al estado de lectura ($e0$), se obtiene el valor de la memoria de programa ($M_p = 31780$) y se activa la señal de lectura de todos los registros excepto PC ($srw = 23 \Rightarrow 10111$), estos dos primeros estados son comunes a todas las instrucciones. El cuarto estado es de decodificación ($e1$), el multiplexor uno permite el paso del dato inmediato ($scm1 = 0 \Rightarrow sm1 = 31780$) y el multiplexor cuatro pasa el valor de $PC + 1$ como dirección de la siguiente instrucción ($scm4 = 0 \Rightarrow cm4 = 1$). El quinto ciclo corresponde al estado de ejecución, en este caso la ALU debe permitir el paso del dato sin ser modificado ($sc_alu = 14 \Rightarrow salu = 31780$).

Finalmente, en el sexto ciclo se escribe el resultado de la ALU en el registro *result* y se actualizan las banderas ($srw = 6 \Rightarrow 00110$), las señales de control de escritura quedan activas hasta el ciclo de lectura de la siguiente instrucción. El valor del registro se observa hasta el ciclo 8 (estado de lectura de I_1).

La instrucción carga dato inmediato en el archivo de registros (I_1), emplea cuatro ciclos de reloj (7-10) debido a que no pasa por el estado de ejecución ($e2$). Finalmente, la instrucción I_2 realiza la multiplicación de los dos operandos, esta instrucción demora 5 ciclos de reloj (11-15), los resultados de la operación se muestran en el ciclo 17, por ser el estado de lectura de la siguiente instrucción. El resultado de la multiplicación se almacena en los registros *result* ($R = 12588$) y Mh ($s_mh = 10142$).

El segundo programa consiste en almacenar un valor del archivo de registros, en la posición de memoria dada por *result*. La tabla 2.6 muestra la memoria de programa correspondiente.

Tabla 2.6: Programa 2

PC	Inst	Cod_op	Dir_reg	Dato-Dir	Decimal	Resultado
0	I_0	00000	000	0000000000001010	10	$R = 10$
1	I_1	00011	011	0010011001100001	1779297	$F_3 = 9825$
2	I_2	01000	011	0000000000000000	4390912	$M[R] = F_3$ $M[10] = 9825$

La figura 2.19 muestra la ejecución de estas instrucciones. El programa se ejecuta en 14 ciclos de reloj. Como en el ejemplo anterior, en el primer ciclo se inicia el funcionamiento del microprocesador mediante la activación de la señal *rst*. Los cinco ciclos siguientes (2-6) corresponden a la instrucción carga dato inmediato en el registro *result* (I_0). Desde el ciclo 7 hasta el 10 se ejecuta la instrucción carga dato inmediato en el archivo de registros (I_1). La instrucción almacena un dato del archivo de registros en la dirección dada por *result* (I_2) inicia en el ciclo 11, donde se ejecuta el estado *erst* en el cual se activa la señal de lectura del registro PC ($srw = 29 \Rightarrow PC = 2$), en el ciclo 12 ($eo$) se leen la memoria de programa y los registros. El ciclo 13 corresponde al estado de decodificación ($e1$), esta instrucción requiere el valor del multiplexor 5 que le indica la dirección de memoria donde se almacenará el dato ($scm5 = 1 \Rightarrow sm5 = 10$), el valor del multiplexor 3 que indica el dato que será almacenado ($scm3 = 1 \Rightarrow sm3 = 9825$), y el valor del multiplexor cuatro que indica la dirección de la siguiente instrucción ($scm4 = 0 \Rightarrow sm4 = 3$). Esta instrucción no pasa por el estado de ejecución, por lo tanto el ciclo 14 corresponde al estado de almacenamiento ($e3$), en este se activa la señal de escritura de la memoria de datos ($rw_Md = 0$). El valor de la memoria de programa se observa en el ciclo 16.

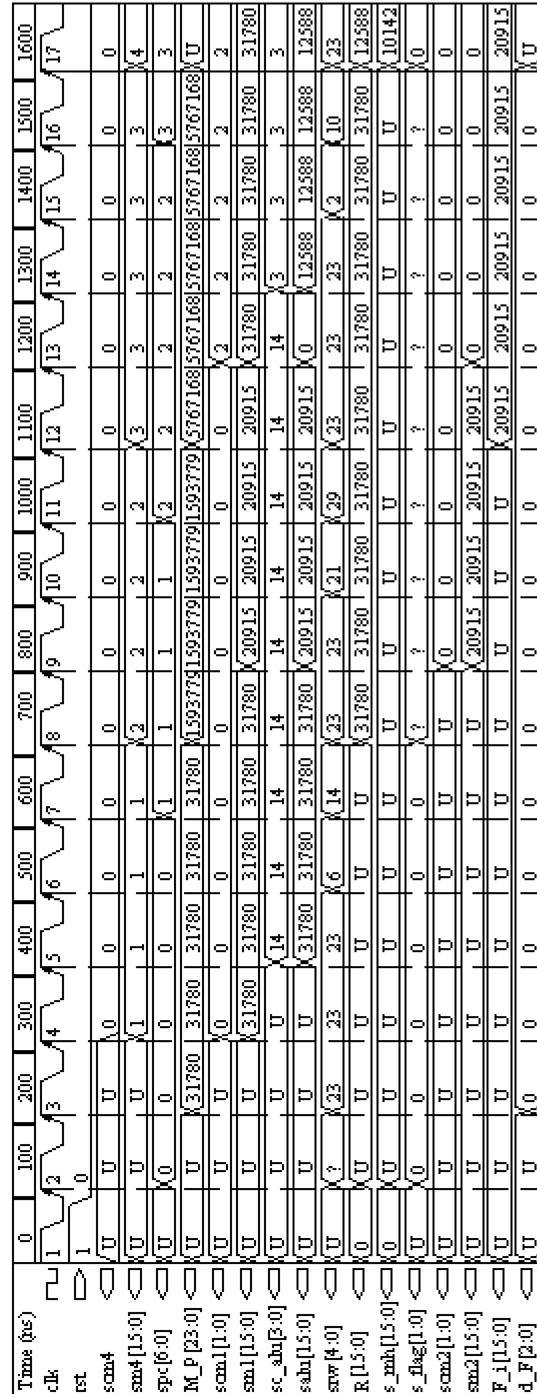


Figura 2.18: Resultados de simulación del programa 1.

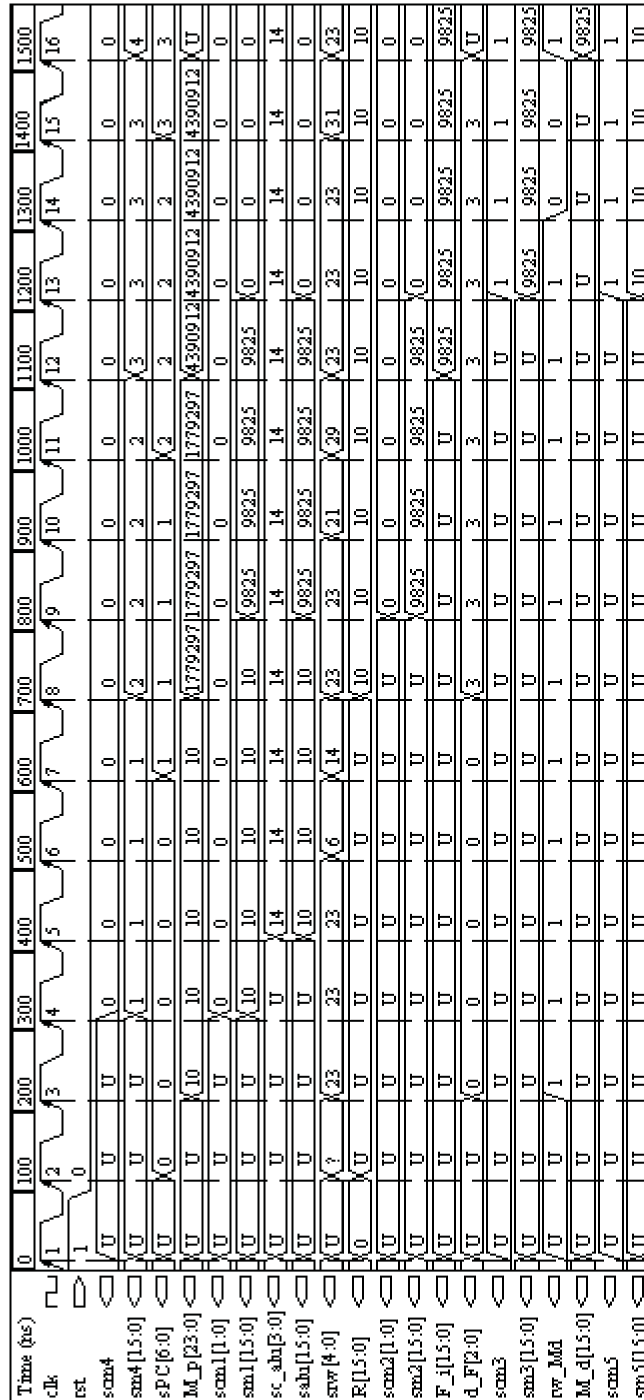


Figura 2.19: Resultados de simulación del programa 2.

Capítulo 3

Implementación de la arquitectura *pipeline.*

El *pipeline* es una técnica que busca mejorar el desempeño del procesador, consiste en descomponer la ejecución de cada instrucción en segmentos, cada uno de los cuales se ordena para formar un cauce, de esta forma una instrucción puede comenzar sin que la anterior haya terminado completamente. Con esta técnica no se reduce el tiempo en que se ejecuta cada instrucción sino que se incrementa el número de instrucciones que se ejecutan y la velocidad a la cual se empiezan y terminan las instrucciones, es decir mejora la productividad de las instrucciones [9].
En el conjunto de instrucciones RISC, cada instrucción puede pasar a través del *pipeline* en la misma forma, mientras que en un conjunto de instrucciones CISC es imposible construir un *pipeline* simple y uniforme [15].

Este capítulo muestra la adecuación del microprocesador descrito en el capítulo anterior para implementar una arquitectura *pipeline*. Primero se hace un análisis del comportamiento de las instrucciones en esta nueva arquitectura, luego se muestran los cambios implementados en cada bloque funcional, se analizan los problemas que se pueden presentar en la ejecución de las instrucciones en una arquitectura *pipeline* y por último se confrontan los resultados obtenidos en la simulación de los programas descritos en el capítulo anterior con la simulación en la arquitectura *pipeline*.

3.1. Arquitectura *pipeline.*

En el caso formal de la ejecución de un programa en el microprocesador descrito en el capítulo anterior, cada una de las instrucciones debe pasar por diferentes estados y para que una nueva instrucción comience es necesario que la anterior haya ejecutado su último estado.

Para comprender mejor la ejecución de un conjunto de instrucciones en las arquitecturas con y sin *pipeline* se analizará un programa que realice la suma de los operandos a y b, las instrucciones son:

$R = a$	Carga el dato a en el registro $result$.
$F_i = b$	Carga el dato b en el archivo de registros.
$Sum[R, F_i]$	Suma el valor de a y b ($result = a + b$).

La figura 3.1 muestra el proceso de ejecución de este programa en una arquitectura sin *pipeline*, las tres instrucciones demoran 14 ciclos de reloj en ejecutarse. Se observa que las unidades funcionales sólo trabajan un ciclo de reloj por cada instrucción, por ejemplo la memoria de programa sólo se utiliza en los ciclos 2, 7 y 11, los otros ciclos utilizan otras unidades funcionales.

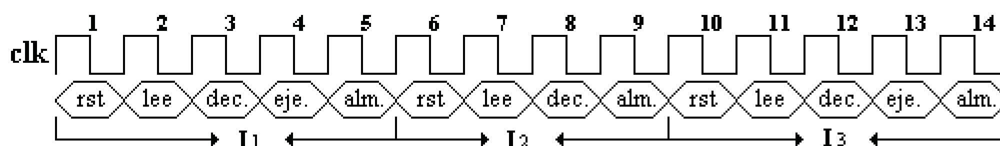


Figura 3.1: Ejecución de $a + b$ en una arquitectura sin *pipeline*.

Por otro lado, la arquitectura *pipeline* plantea que en cada ciclo de reloj se empiece a ejecutar una nueva instrucción, esto hace que las unidades funcionales se utilicen todo el tiempo en diferentes instrucciones. La figura 3.2 muestra la ejecución del programa mencionado anteriormente en una arquitectura *pipeline*, el tiempo de ejecución en este caso se reduce a 7 ciclos de reloj.

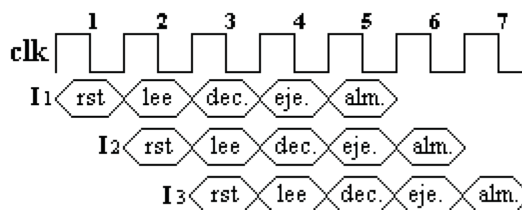


Figura 3.2: Ejecución de $a + b$ en una arquitectura con *pipeline*.

3.2. Ejecución de las instrucciones en la arquitectura *pipeline*.

A continuación se muestra la ejecución en una arquitectura *pipeline* de las instrucciones descritas en el capítulo 1, con el fin de determinar los cambios que deben hacerse en el microprocesador original.

Para analizar esta ejecución se toma como ejemplo un programa que contiene todos los tipos de instrucciones que puede manejar este microprocesador, las instrucciones son:

$$\begin{aligned} I_0 :&\quad R = M[di] \\ I_1 :&\quad F_i = di \\ I_2 :&\quad NOP \\ I_3 :&\quad JMP \\ I_4 :&\quad M[R] = F_i \\ I_5 :&\quad Sum[R, F_i] \end{aligned}$$

La figura 3.3 muestra el proceso de ejecución de estas instrucciones, a la izquierda se observan las señales de salida de los diferentes bloques funcionales con su respectivo número de bits, en la derecha se muestran los cambios de estas señales en cada ciclo de reloj dependiendo de la instrucción que se esté ejecutando. Como fue necesario agregar al microprocesador original unos registros de almacenamiento, algunas de las señales que se muestran no existen en el diseño original.

Las señales de control de los multiplexores y de la ALU cambian sólo cuando una nueva instrucción las utilice; las señales de control de escritura se especifican en cada instrucción aún cuando no se usen con el fin de evitar que se modifique inadecuadamente el valor de los registros o la memoria de datos.

Asumiendo que estas son las primeras instrucciones de un programa almacenado en memoria, en el primer ciclo de reloj el registro PC debe iniciar en cero y por defecto aumentar en uno cada ciclo. Después Que el registro PC lee la dirección, la memoria de programa (M_Prog) debe mostrar la instrucción correspondiente, esta contiene el código de operación, dato y dirección que se deben utilizar, estos dos primeros estados se ejecutan para todas las instrucciones. A continuación se analiza por separado la ejecución de cada tipo de instrucción.

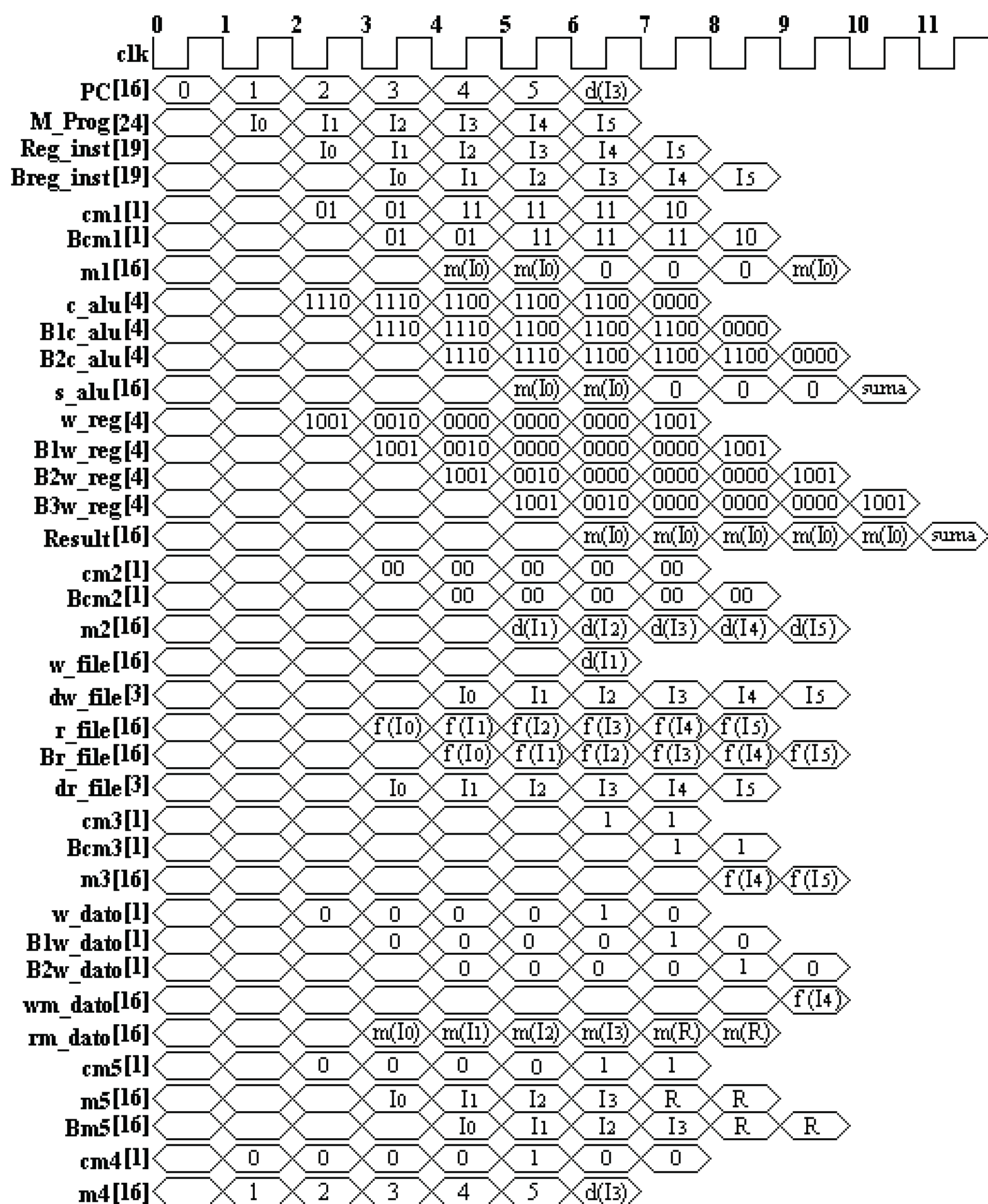


Figura 3.3: Ejemplo de la ejecución de 7 instrucciones.

3.2.1. Instrucciones de carga.

La primera instrucción (I_0) carga un valor de memoria en el registro $result$. Un ciclo después de leer la memoria de programa (clk_2) se obtienen todas las señales de control necesarias. Ésta instrucción en particular requiere las siguientes señales de control:

- Las señales de control de lectura permanecen siempre activas para todas las instrucciones, por lo tanto no se tienen en cuenta en el esquema.
- El control del multiplexor 5 ($cm5$) que indica la dirección de la memoria de datos donde se lee o escribe un dato, para este caso se debe leer un dato indicado por el formato de instrucción ($cm5 = 0 \Rightarrow m5 = I_0 \Rightarrow rm_dato = m(I_0)$), este dato de memoria se obtiene en clk_3.
- El control del multiplexor 1 ($cm1$) que debe permitir el paso del valor de la memoria de datos. Con el propósito de garantizar el paso de las instrucciones por los mismos estados que en la arquitectura sin *pipeline*, los multiplexores fueron diseñados para recibir en un ciclo de reloj la señal de control y las entradas y en el ciclo siguiente mostrar la salida. Sin embargo esto genera un problema para las instrucciones que utilizan la memoria de datos, pues el valor de ésta (rm_dato) se obtiene un ciclo después (clk_3) que la señal de control del multiplexor uno (clk_2), por tanto es necesario colocar un registro de almacenamiento a la señal de control del multiplexor uno ($Bcm1$). Para la instrucción de cargar un dato de memoria en el archivo de registros, también es necesario colocar un registro de almacenamiento a la señal de control del multiplexor 2 ($Bcm2$). En clk_4 se obtiene la salida del multiplexor 1 ($m1 = m(I_0)$).
- El control de la ALU (c_alu), que para las instrucciones de carga en el registro $result$ debe permitir el paso de la primera entrada sin modificaciones ($c_alu = 1110$). Como esta señal se obtuvo en el tercer ciclo, es necesario colocar dos registros de almacenamiento ($B1c_alu, B2c_alu$). La salida de la ALU ($s_alu = m(I_0)$) se obtiene en clk_5.
- La señales de control de escritura de los registros $result$ y $flag$ ($w_reg = 1001$), nuevamente es necesario utilizar registros de almacenamiento ($B1w_reg$, $B2w_reg$, $B3w_reg$) pues estas señales se obtuvieron en el tercer ciclo de reloj.

Este proceso lo siguen todas las señales de carga en el registro $result$, solo debe cambiar la señal $cm1$ dependiendo del origen del dato.

La segunda instrucción (I_1) corresponde a cargue un dato inmediato en el archivo de registros. La dirección de la memoria de programa (PC) se obtiene en el segundo ciclo de reloj y la instrucción se lee en el tercer ciclo ($M_Prog = I_1$).

Las señales de control correspondientes se obtienen por tanto en el cuarto ciclo de reloj. Para esta instrucción se deben tener en cuenta:

- El control del multiplexor 2 ($cm2$) que como se mencionó anteriormente, necesita un registro de almacenamiento ($Bcm2$), por tanto el dato del formato de instrucción (dato inmediato) no se puede tomar directamente de la memoria de programa sino que es necesario almacenarlo durante dos ciclos de reloj, para esto se utilizan los registros Reg_inst y $Breg_inst$. Esta instrucción debe permitir el paso del dato inmediato por tanto $cm2 = 00 \Rightarrow m2 = d(I_1)$. La salida del multiplexor dos se obtiene en clk_5.

- La señal de control de escritura del archivo de registros ($w_reg = 0010$) que se obtuvo en el tercer ciclo, por tanto necesita dos registros de almacenamiento (B1w_reg,B2w_reg).

Este procedimiento es seguido por todas las instrucciones de carga en el archivo de registros, la señal $cm2$ cambia dependiendo del origen del dato.

3.2.2. Instrucciones de control.

La tercera instrucción (I_2) corresponde a no opere. Ésta instrucción realiza la multiplicación por cero y no almacena el resultado en ningún registro. La dirección de la memoria de programa (PC) se toma en clk_2 y la memoria de programa se lee en el ciclo siguiente ($M_Prog = I_2$). Las señales de control se obtienen en clk_4, en este caso se requiere:

- El multiplexor uno debe permitir el paso del registro cero ($cm1 = 11 \Rightarrow m1 = 0$).

- EL control de la ALU que ejecuta la multiplicación ($c_alu = 1100$). Como el valor del multiplexor uno se obtiene hasta clk_6, se necesitan dos registros de almacenamiento del control de la ALU ($B1c_alu$, $B2c_alu$).

- Las señales de escritura que deben ser cero ($w_reg = 0000$, $w_d ato = 0$) pues no se almacena ningún valor.

Por otro lado la instrucción I_3 corresponde al salto incondicional (JMP), esta inicia en clk_3 con la lectura del registro PC, en clk_4 se obtiene el valor de la memoria de programa ($M_Prog = I_3$). Las señales de control se observan en clk_5, en este caso son necesarias:

- El control del multiplexor 4 que permite el paso del dato inmediato ($cm4 = 1 \Rightarrow m4 = d(I_3)$). El dato de entrada de este multiplexor proviene del registro Reg_inst.

- Las señales de control de escritura que deben ser cero debido a que esta instrucción no escribe ningún valor en los registros ($w_reg = 0000$) ni en memoria ($w_d ato = 0$). La nueva dirección de la memoria de programa se observa en clk_6.

Este proceso lo siguen las instrucciones de salto condicional, sólo que el valor de la señal $cm4$ cambia dependiendo del valor del registro de banderas ($flag$).
Con las instrucciones de salto se presenta un conflicto debido a que las dirección de la siguiente instrucción se obtiene dos ciclos después que se indica el salto, esto se analiza en la sección 3.5.

3.2.3. Instrucciones de almacenamiento.

En clk_4 se obtiene la dirección de la memoria de programa (PC) de la instrucción almacena un valor del archivo de registros en la dirección dada por el registro $result$, la memoria de programa se lee en clk_5 ($M_P rog = I_4$). En clk_6 se obtienen las señales de control correspondientes.

- El multiplexor 5 indica la dirección de la memoria donde se almacenará el dato, en este caso $m5$ debe permitir el paso del valor del registro $result$ ($cm5 = 1 \Rightarrow m5 = R$), esta dirección se obtiene el en clk_7.
- El multiplexor 3 selecciona el dato que se va a almacenar, para esta instrucción debe seleccionar el archivo de registros ($cm3 = 1$), como el valor del archivo de registros se obtiene en clk_7 es necesario almacenar la señal de control de $m3$ ($Bcm3$). En clk_8 se obtiene el valor del multiplexor 3 ($m3 = f(I_4)$)
- La señal de control de escritura de la memoria de datos necesita ser almacenada por dos ciclos de reloj ($B1w_dato, B2w_dato$), además se necesita un registro de almacenamiento para la dirección de escritura (Bm5).

3.2.4. Instrucciones aritmético-lógicas.

Finalmente, la instrucción I_5 ilustra el proceso de ejecución de las instrucciones aritmético-lógicas, el ejemplo corresponde a la suma del registro $result$ con un valor del archivo de registros. La dirección de esta instrucción ($PC = 5$) se obtiene en clk_5, la memoria de programa se lee en el ciclo siguiente ($M_Prog = I_5$). Las señales de control se generan en clk_7, para estas instrucciones se deben tener en cuenta:

- El control del multiplexor uno que permite el paso del registro $result$ ($cm1 = 10 \Rightarrow m1 = m(I_0)$), el valor de este multiplexor se muestra en clk_9.
- La señal de control de la ALU que indica la operación que se realizará, en este caso la operación corresponde a una suma $c_alu = 0000$. Como la señal de control

de la ALU se obtuvo dos ciclos antes que la salida del multiplexor 1, esta debe ser almacenada ($B1c_alu$, $B2c_alu$). La segunda entrada de la ALU es un valor del archivo de registros, como este se obtuvo en clk_8, es necesario almacenarlo durante un ciclo (Br_file). El resultado de la ALU se obtiene en clk_{10}.

- Las señales de control de escritura de los registros $result$ y $flag$ que deben ser almacenadas durante tres ciclos de reloj ($B1w_reg$, $B2w_reg$, $B3w_reg$).

Para las otras instrucciones aritmético lógicas sólo es necesario modificar la señal de control de la ALU (c_alu).

De este análisis es importante observar que el tiempo de ejecución de cada instrucción por separado aumentó en un ciclo de reloj, esto se debe a las instrucciones de acceso a memoria pues en la arquitectura sin *pipeline* las instrucciones operan sobre datos actuales, es decir los multiplexores no debe esperar la salida de la memoria de datos (ver secc. 2.3.2), mientras que en la arquitectura *pipeline* es necesario esperar un ciclo de reloj para recibir el dato de la memoria de programa. Sin embargo con la arquitectura *pipeline* es posible reducir el tiempo de ejecución de todo el programa por que cada ciclo de reloj inicia una nueva instrucción.

3.3. Diseño del *datapath*.

Una vez descrito el proceso de ejecución de las instrucciones, se muestra con detalle los cambios realizados en cada unidad funcional para que esta arquitectura *pieline* funcione correctamente. Es importante notar que estos cambios son a nivel de componentes, el microprocesador debe seguir ejecutando el mismo conjunto de instrucciones que se describió en los capítulos anteriores y por lo tanto el *datapath* conserva su estructura básica. Sin embargo es necesario agregar nuevos componentes y modificar la estructura de algunos ya existentes para garantizar la correcta ejecución de las instrucciones en esta nueva arquitectura. La figura 3.4 muestra el esquema del *datapath* para la arquitectura con *pipeline*. A continuación se mencionan los cambios que se realizaron en algunos bloques funcionales y se describen los nuevos bloques utilizados.

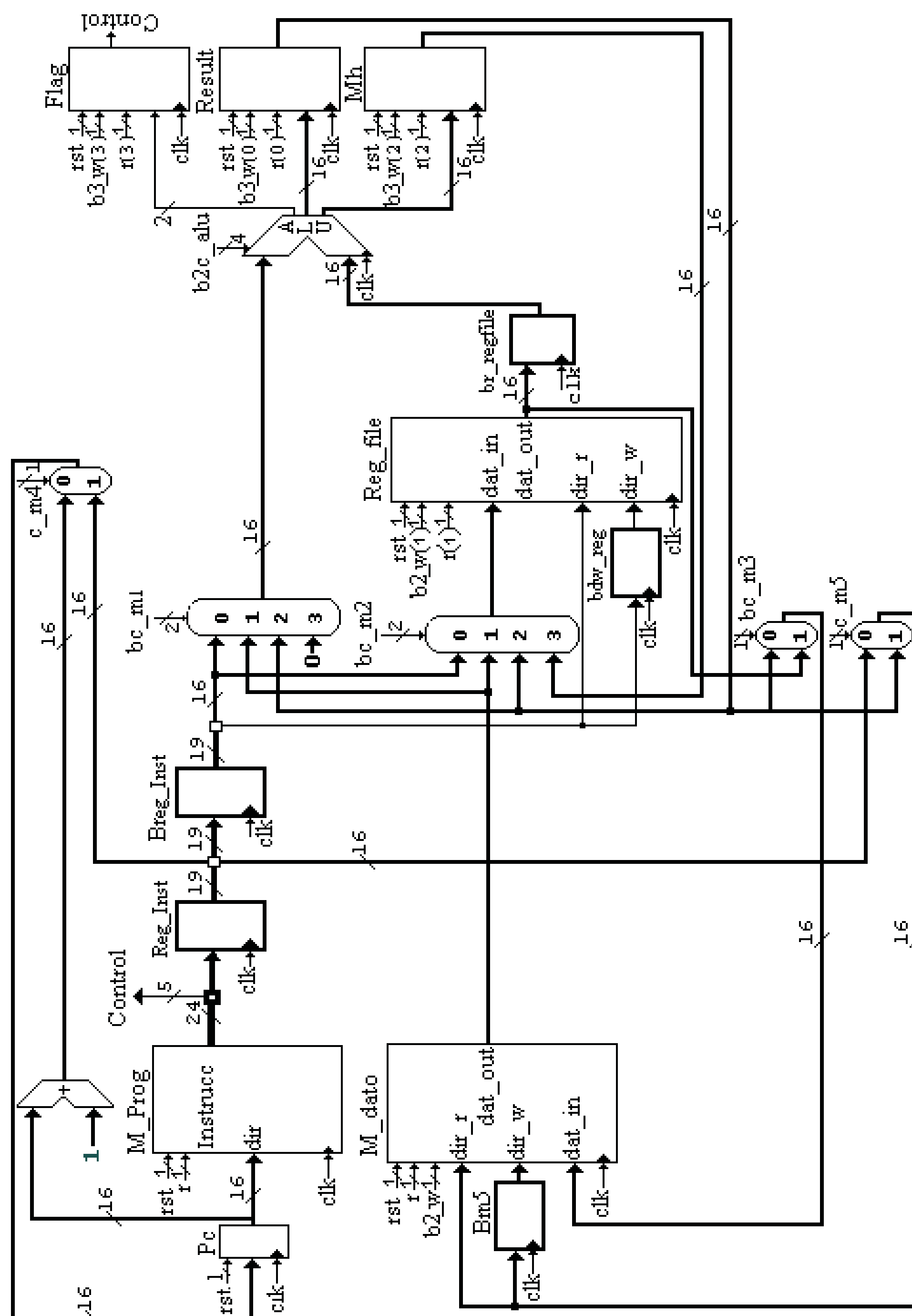


Figura 3.4: Datapath con arquitectura pipeline.

3.3.1. Registros.

Aunque el microprocesador debe tener todos registros que en la arquitectura original, la forma como estos se diseñaron cambió.
En la ejecución de las instrucciones sin *pipeline* un registro lee y almacena datos en diferentes ciclos de reloj, por tanto se utiliza una única señal que controle la lectura-escritura (rw_reg). Al utilizar la arquitectura con *pipeline* es posible que un registro necesite leer y almacenar datos en un mismo ciclo, por tanto se deben tener por separado señales de control para lectura (r_reg) y escritura (w_reg).

Archivo de registros y memoria de datos.

En el archivo de registros y la memoria de datos ocurre otro cambio importante. En la ejecución de las instrucciones sin *pipeline* la dirección de lectura y escritura es la misma, esto se debe a que cada instrucción se ejecuta por separado, sin embargo al realizar el diseño con *pipeline* es posible que una instrucción necesite leer un registro o la memoria de datos, mientras que otra este escribiendo en otro registro o en otra dirección de memoria, por lo tanto además de las señales de control se necesita por separado una dirección para lectura (dr_file, $m5$) y otra para escritura (dw_file, $Bm5$) del archivo de registro y la memoria de datos respectivamente.

Registro Pc.

En la arquitectura sin *pipeline* la dirección de memoria se lee en el estado inicial (rst), el cálculo de la siguiente dirección se realiza en el segundo estado (lee), esto no afecta las siguientes instrucciones pues la nueva dirección debe ser leída ciclos después que se calculó. Para la arquitectura con *pipeline*, la dirección de la siguiente instrucción debe ser leída en el primer ciclo de tal forma que la nueva instrucción inicie inmediatamente, para que esto sea posible el registro PC no tiene señales de lectura o escritura simplemente permite el paso de la dirección de memoria en caso que la señal rst no este activa. La figura 3.5 muestra el cálculo de la siguiente instrucción para las dos arquitecturas.

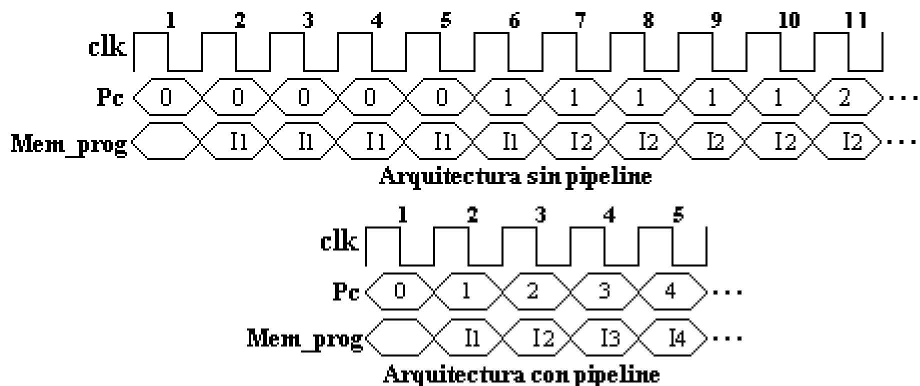


Figura 3.5: Cálculo de la siguiente instrucción.

Registros de almacenamiento.

En el análisis de la sección 3.2 se observó que para el correcto funcionamiento de la arquitectura *pipeline* es necesario incluir registros para almacenar datos que pueden ser alterados por instrucciones posteriores, cada ciclo de reloj almacena un nuevo dato, estos registros no tienen señales de control de lectura o escritura.

Registros de Instrucción (*Reg_inst*, $Breg_inst$)**:** Almacenan los 19 bits menos significativos de la instrucción (3 bits de dirección del archivo de registros y 16 bits de datos), como los bits del código de operación son utilizados inmediatamente no es necesario almacenarlos en estos registros.

Registro File ($Br_regfile$)**:** Es un registro de 16 bits que almacena la salida del archivo de registros, es utilizado en las instrucciones aritmético-lógicas.

Registros de dirección (*bdw_reg*, $Bm5$)**:** Almacenan la dirección de escritura del archivo de registro (*bdw_reg*) y la memoria de datos ($Bm5$) para utilizarla en el momento indicado.

Registros de control: La unidad de control entrega todas las señales en el tercer ciclo de la instrucción, sin embargo la señales de control de los multiplexores $m1$, $m2$, y $m3$ se utilizan un ciclo después por lo tanto necesitan un registro de almacenamiento. Las señales de control de la ALU (*c_alu*), de escritura del archivo de registros ($w_reg(1)$) y de escritura de la memoria de datos (*w_dato*) necesitan dos registros de almacenamiento. Las señales de escritura de los registros $resutl$, $flag$ y mh necesitan tres registros de almacenamiento. Todas estas señales de control son agrupadas en tres registros como se muestra en la figura 3.6.

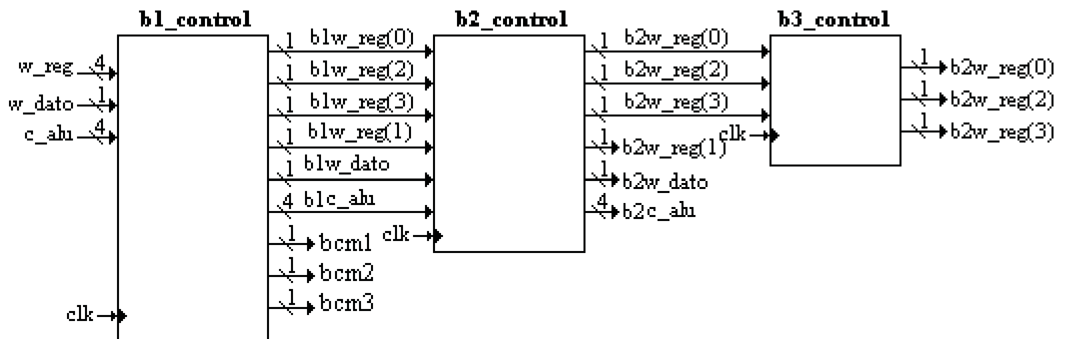


Figura 3.6: Registros de almacenamiento de las señales de control.

3.3.2. Multiplexores.

Con el propósito de garantizar que las instrucciones se ejecuten de igual forma que en la arquitectura sin *pipeline*, se diseñaron los multiplexores de modo que en un ciclo de reloj reciban la señal de control y las entradas, y en el siguiente ciclo muestran la salida.

Como el multiplexor cuatro tiene dos entradas debe tener una señal de control de un bit, en la arquitectura sin *pipeline* si esta señal es cero selecciona la primera entrada y si es uno la segunda, sin embargo al ejecutarse el *pipeline* se presenta un conflicto en este multiplexor con las instrucciones de salto.

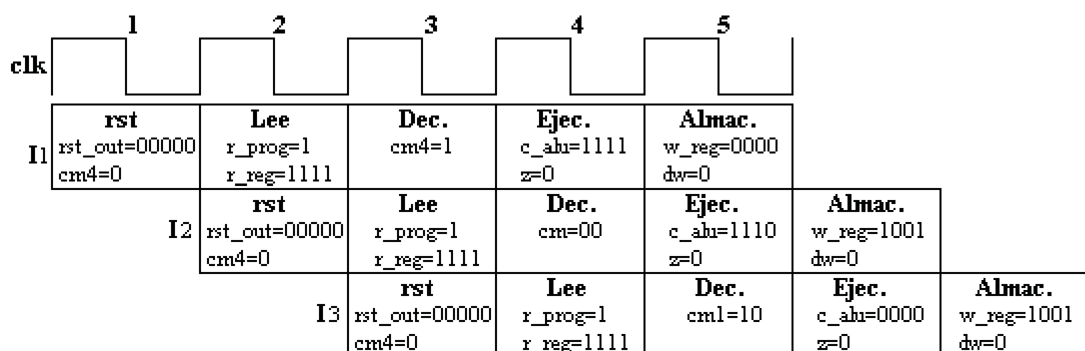


Figura 3.7: Conflicto con el multiplexor 4.

La figura 3.7 muestra la ejecución de tres instrucciones, la primera corresponde a un salto incondicional, la segunda carga un dato inmediato en el registro *result* y la tercera realiza una suma, se observa que en el tercer ciclo de reloj el valor de la señal de control del multiplexor cuatro es uno y cero, para solucionar este problema se definió en el multiplexor cuatro la primera entrada como salida por defecto, cuando la señal de control es uno la salida corresponde a la segunda entrada, es decir que en el estado *rst* no es necesario colocar la señal $cm4 = 0$.

3.4. Diseño de la unidad de control.

En una arquitectura sin *pipeline*, las instrucciones se ejecutan por estados, el microprocesador debe permitir el paso de la instrucción por cada estado e iniciar una nueva instrucción cuando la anterior haya terminado, para esto la unidad de control entrega las señales específicas de cada estado.
En la ejecución de las instrucciones con *pipeline* se debe seguir garantizando el paso de la instrucción por cada estado pero cada ciclo de reloj inicia una instrucción, para que esto se cumpla el diseño plantea que la unidad de control entregue todas las señales un ciclo después que lee la instrucción, estas señales se almacenan en registros (ver sección 3.3.1) hasta el momento que las necesite ; esto facilita considerablemente el diseño de la unidad de control y traslada el problema al diseño del *datapath*. La figura 3.9 muestra el comportamiento de la unidad de control para la instrucción suma en una arquitectura con y sin *pipeline*. Se observa que en la arquitectura sin *pipeline* como el resultado se almacena en un registro solo se entrega la señal *rw_reg*, mientras que en la arquitectura *pipeline* deben entregarse todas las señales de escritura sin importar la instrucción que se este ejecutando, por esto también se aclara la señal de control de escritura en memoria (*w_dato*) que debe ser cero para que no escriba ningún valor, esto con el fin que anteriores instrucciones no afecten la ejecución actual. La figura también muestra diferentes señales de control para las dos arquitecturas, esto se debe a las modificaciones hechas a algunos bloques funcionales del *datapath*, a continuación se muestra la descripción de la unidad de control para una arquitectura con *pipeline*.

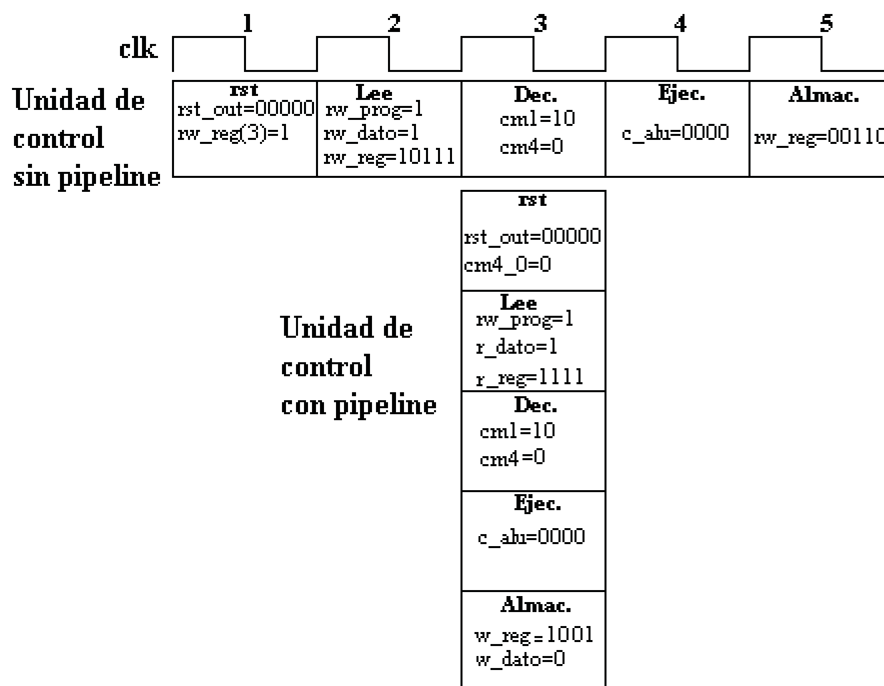


Figura 3.8: Señales de control de la instrucción suma.

3.4.1. Señales de la unidad de control.

Esta nueva unidad de control debe tener las mismas señales de entrada que la unidad original, pues estas no dependen del modo en que se ejecutan las instrucciones sino del tipo de instrucciones, sin embargo las señales de salida son diferentes. En primer lugar las señales de lectura-escritura de cada registro y de la memoria de datos deben ser separadas, es decir en el capítulo anterior se tenía una señal para lectura y escritura (1 lee - 0 escribe), sin embargo en una arquitectura con *pipeline* es posible en el mismo ciclo de reloj leer o escribir un dato en un registro o en memoria, es decir la señal valdría uno y cero en ese ciclo, para solucionar esto la unidad de control tiene por separado señales de lectura y escritura. Estas señales son agrupadas nuevamente en vectores, sin embargo como el registro PC no tiene señales de lectura o escritura, el nuevo orden de los vectores se muestra en la tabla 3.1. En esta nueva estructura, un valor alto (1) en los vectores r_reg o w_reg indica lectura o escritura respectivamente, un valor bajo (0) no permite leer o escribir. La tabla 3.2 muestra un resumen de las señales de control empleadas en la arquitectura *pipeline*.

Tabla 3.1: Vectores *rst_out*, *r_reg* y *w_reg*.

Posición (*rst_out* *w_reg* *r_reg*)	Registro	Descripción del registro
0	Result	Almacena los resultados de la ALU.
1	Reg_file	Archivo de registros de propósito general.
2	Mh	Almacena los bits más significativos de la multiplicación.
3	Flag	Almacena las banderas cero y acarreo.
4	Pc*	Almacena la dirección de la memoria de programa.

* *Pc* sólo está en el vector *rst_out*, los otros vectores tienen una longitud de 4 bits.

La figura 3.9 muestra el esquema de entradas y salidas de la unidad de control con *pipeline*; las señales reset de los registros (rst_out), lectura de la memoria de programa (*rw_prog*), control de la ALU (*c_alu*) y control de los multiplexores ($cm1$, $cm2$, $cm3$, $cm4$, $cm5$) no fueron modificadas.

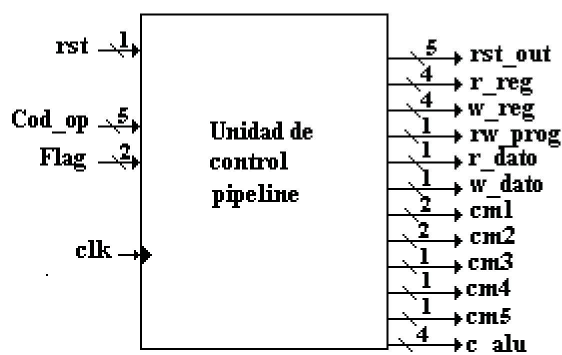


Figura 3.9: Unidad de control para la arquitectura pipeline.

La tabla 3.2 resume las nuevas señales de control.

Tabla 3.2: Señales de la unidad de control en la arquitectura *pipeline*

Reset ($erst$)	Lee ($e0$)	Decodifica ($e1$) Cod_op		Ejecuta ($e2$)	Almacena ($e3$)	Instrucción
		00000 $R = di$	cm1=00 cm4=0	c_alu=1110 z=0	w_reg=1001 w_dato=0	$R = di$
if rst=1: rst_out=11111	r_prog=1 r_dat=1	00001 $R = F_i$	cm4=0	c_alu=1111 z=0	rw_reg=1001 w_dato=0	$R = F_i$
	r_reg=1111	00010 $R = M[di]$	cm1=01 cm5=0 cm4=0	c_alu=1110 z=0	w_reg=1001 w_dato=0	$R = M[di]$
else rst=0: rst_out=00000		00011 $F_i = di$	cm2=00 cm4=0	 z=0	w_reg=0010 w_dato=0	$F_i = di$
rw_reg=11111		00100 $F_i = R$	cm2=10 cm4=0	 z=0	w_reg=0010 w_dato=0	$F_i = R$
		00101 $F_i = M[R]$	cm2=01 cm5=1 cm4=0	 z=0	w_reg=0010 w_dato=0	$F_i = M[R]$
		00110 $F_i = mh$	cm2=11 cm4=0	 z=0	w_reg=0010 w_dato=0	$F_i = Mh$
		00111 $M[di] = R$	cm3=0 cm5=0 cm4=0	 z=0	w_reg=0000 w_dato=1	$M[di] = R$
		01000 $M[R] = F_i$	cm3=1 cm5=1 cm4=0	 z=0	w_reg=0000 w_dato=1	$M[R] = F_i$
		01001 $Sum[R, F_i]$	cm1=10 cm4=0	c_alu=0000 z=0	w_reg=1001 w_dato=0	$Sum[R, F_i]$
		01010 $Res[R, F_i]$	cm1=10 cm4=0	c_alu=0001 z=0	w_reg=1001 w_dato=0	$Res[R, F_i]$
		01011 $Mult[R, F_i]$	cm1=10 cm4=0	c_alu=0011 z=0	w_reg=1101 w_dato=0	$Mult[R, F_i]$
		01100 C_a2	cm1=10 cm4=0	c_alu=0001 z=0	w_reg=1001 w_dato=0	C_{a2}
		01101 $Xor[R, F_i]$	cm1=10 cm4=0	c_alu=1011 z=0	w_reg=1001 w_dato=0	$Xor[R, F_i]$
		01110 $And[R, F_i]$	cm1=10 cm4=0	c_alu=1001 z=0	w_reg=1001 w_dato=0	$And[R, F_i]$
		01111 $Or[R, F_i]$	cm1=10 cm4=0	c_alu=1010 z=0	w_reg=1001 w_dato=0	$Or[R, F_i]$
		10000 NOT	cm1=10 cm4=0	c_alu=1000 z=0	w_reg=1001 w_dato=0	Not
		10001 SD	cm1=10 cm4=0	c_alu=0100 z=0	w_reg=1001 w_dato=0	SD
		10010 SI	cm1=10 cm4=0	c_alu=0101 z=0	w_reg=1001 w_dato=0	SI
		10011 RD	cm1=10 cm4=0	c_alu=0110 z=0	w_reg=1001 w_dato=0	RD
		10100 RI	cm1=10 cm4=0	c_alu=0111 z=0	w_reg=1001 w_dato=0	RI
		10101 INC	cm1=10 cm4=0	c_alu=1100 z=0	w_reg=1001 w_dato=0	INC
		10110 DIS	cm1=10 cm4=0	c_alu=1101 z=0	w_reg=1001 w_dato=0	DIS
		10111 JMP	cm4=1	 z=0	w_reg=0000 w_dato=0	JMP
		11000 Jz	cm4=1 (z) cm4=0 (nz)	 z=0	w_reg=0000 w_dato=0	Jz
		11001 NOP	cm1=11 cm4=0	c_alu=0011 z=0	w_reg=0000 w_dato=0	NOP

3.5. Problemas en la ejecución.

Hasta ahora se ha analizado el comportamiento ideal de una arquitectura *pipeline*, sin embargo se presentan algunos conflictos en la ejecución de las instrucciones [9, 16]. Para discutir los problemas que se pueden presentar en una arquitectura *pipeline*, se analizará la siguiente secuencia de instrucciones, su ejecución se muestra en la figura 3.10.

$$
\begin{aligned}
I_0 : &\quad R = di \\
I_1 : &\quad Sum[R, F_i] \\
I_2 : &\quad JMP \\
I_3 : &\quad Sum[R, F_i] \\
I_4 : &\quad Not \\
I_5 : &\quad F_i = di
\end{aligned}
$$

3.5.1. Dependencia de datos.

La dependencia de datos ocurre cuando una instrucción debe operar sobre datos calculados por instrucciones anteriores pero estos datos aún no están disponibles.
La primera instrucción (I_0) de la figura 3.10 carga un dato inmediato en el registro $result$, la instrucción I_1 debe sumar este dato con un valor del archivo de registros, sin embargo se observa que en clk_5 cuando el multiplexor 1 selecciona el valor del registro $result$, este aún no se ha actualizado con el valor de $d(I_0)$. En este diseño la dependencia de datos ocurre cuando una instrucción necesita datos calculados por instrucciones anteriores hasta en dos ciclos de reloj.

3.5.2. Riesgos de control.

La arquitectura *pipeline* plantea que cada ciclo de reloj inicie una instrucción, esto genera conflictos al ejecutar las instrucciones de salto.
En el programa ejemplo la instrucción I_2 corresponde a un salto incondicional, es decir que la siguiente instrucción en ejecutarse no debería ser I_3 sino la señalada por el dato de esta de I_2 ($I_{d(I_2)}$), sin embargo el registro Pc obtiene este valor dos ciclos después de leída I_2, por tanto se inicia la ejecución de las instrucciones I_3, I_4.

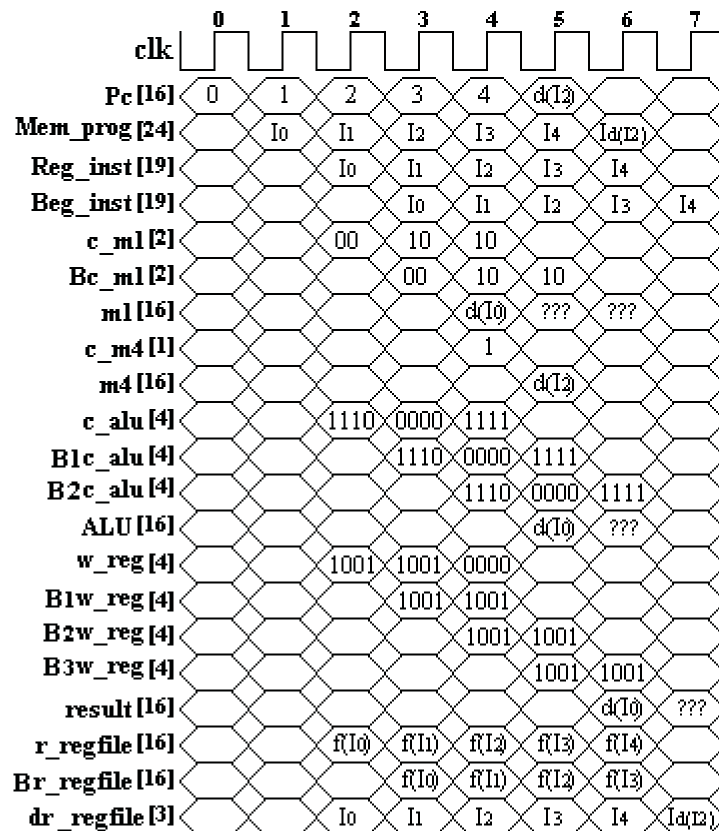


Figura 3.10: Conflictos en la ejecución de la arquitectura pipeline.

3.5.3. Solución a los conflictos en la ejecución: Esperas

Este diseño plantea una solución simple pero eficaz que permite evitar que estos problemas se presenten.
La solución a estos conflictos puede darse a nivel del programador, la idea es evitar que las instrucciones que tengan dependencias de datos estén seguidas por lo menos en dos ciclos de reloj. Para esto se puede analizar si es posible reorganizar el programa, de lo contrario se insertan esperas en la ejecución de las instrucciones hasta que el dato que se necesite esté listo. Estas esperas se insertan utilizando la instrucción NOP (no opere) pues esta demora ciclos de reloj en ejecutarse pero no modifica el valor de ningún registro.
Al insertar la instrucción NOP después de las instrucciones de salto se evita que se ejecuten instrucciones que pueden alterar los resultados del programa.

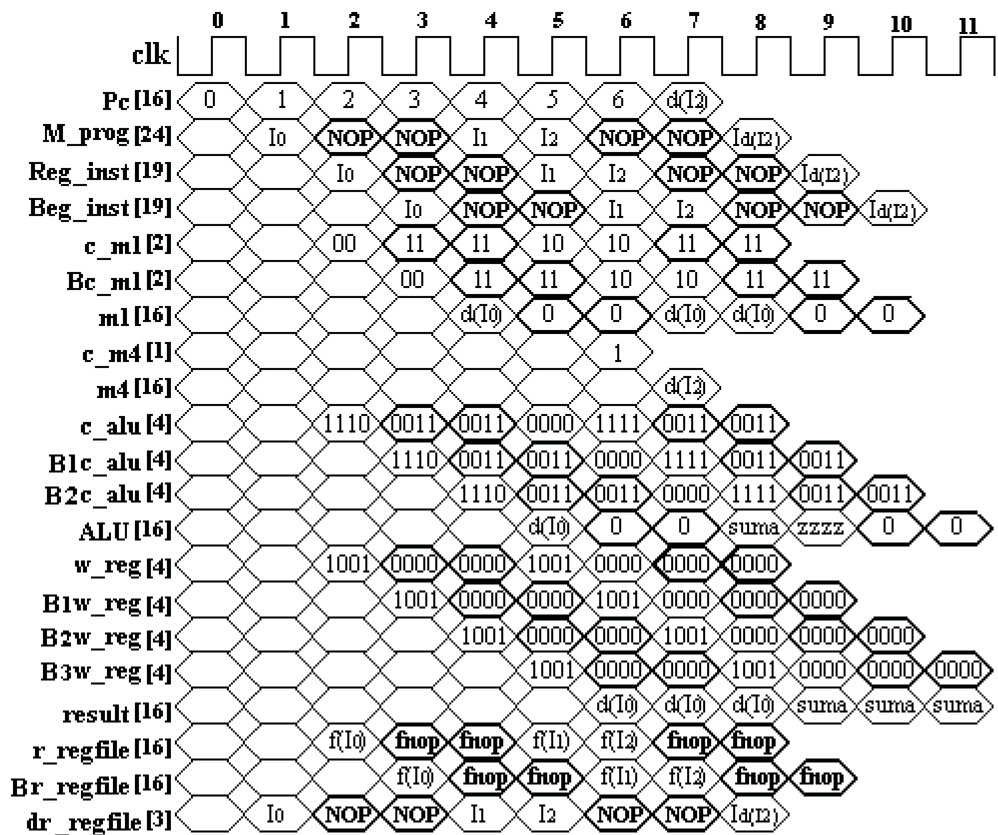


Figura 3.11: Solución a los conflictos en la ejecución.

La figura 3.11 muestra la ejecución del ejemplo anterior pero insertando esperas en donde se presentan conflictos, esta solución no requiere utilizar nuevo *hardware* en el diseño, sin embargo el tiempo de ejecución del programa aumenta.
Existen otras soluciones a estos conflictos que requieren algunas modificaciones al diseño planteado y mejoran el rendimiento del microprocesador [7].

3.6. Simulación.

Con el propósito de comprender mejor la ejecución de las instrucciones en la arquitectura *pipeline* y comparar algunos resultados de las arquitecturas con y sin *pipeline*, en esta sección se mostrará la ejecución de los programas ya simulados en el capítulo anterior.

En la arquitectura sin *pipeline* se realizó el análisis de ejecución de las instrucciones para dos programas, el primer programa se encarga de realizar una multiplicación, mientras que el segundo programa almacena un valor del archivo de registros.

El primer programa a simular es la operación de multiplicación. La tabla muestra la memoria que contiene estas instrucciones. Se observa que la instrucción I_3 depende de las instrucciones I_0 e I_1, por tanto es necesario insertar una espera mediante la instrucción NOP.

Tabla 3.3: Programa 1 para la arquitectura *pipeline*.

PC	Inst	Cod_op	Dir_reg	Dato-Dir	Decimal	Resultado
0	I_0	00000	000	0111110000100100	31780	$R = 31780$
1	I_1	00011	000	0101000110110011	1593779	$F_0 = 20915$
2	I_2	11010	000	0000000000000000	13631488	NOP
3	I_3	01011	000	0000000000000000	5767168	$R = 12588$ $Mh = 10142$

La figura 3.12 muestra el proceso de ejecución de estas instrucciones, en total el programa emplea 10 ciclos de reloj en ejecutarse, es decir la arquitectura con *pipeline* mejora el rendimiento el microprocesador para este programa en un 33 % aproximadamente.

Por otro lado, el segundo programa corresponde a el almacenamiento de un valor del archivo de registros en la memoria de programa. Las instrucciones son las siguientes. En este caso la instrucción I_4 depende del resultado de I_1, por tanto es necesario insertar dos instrucciones de espera para que el dato sea almacenado adecuadamente.

Tabla 3.4: Programa 2

PC	Inst	Cod_op	Dir_reg	Dato-Dir	Decimal	Resultado
0	I_0	00000	000	0000000000001010	10	$R = 10$
1	I_1	00011	011	0010011001100001	1779297	$F_3 = 9825$
2	I_2	11010	000	0000000000000000	13631488	NOP
3	I_3	11010	000	0000000000000000	13631488	NOP
4	I_4	01000	011	0000000000000000	4390912	$M[R] = F_3$ $M[10] = 9825$

La figura 3.13 muestra la ejecución de este programa, en total emplea 10 ciclos de reloj en ejecutarse. El rendimiento con respecto a la arquitectura sin *pipeline* para este programa mejora en un 26 % aproximadamente.

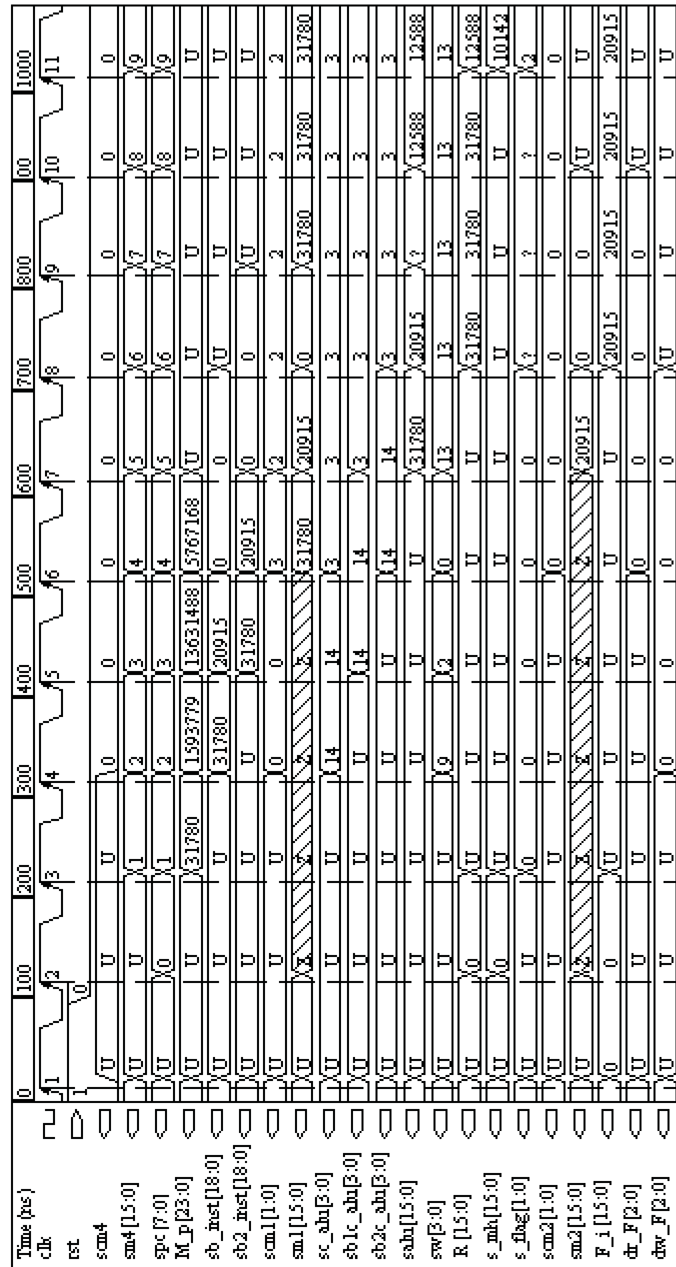


Figura 3.12: Resultados de simulación del programa 1 en la arquitectura *pipeline*.

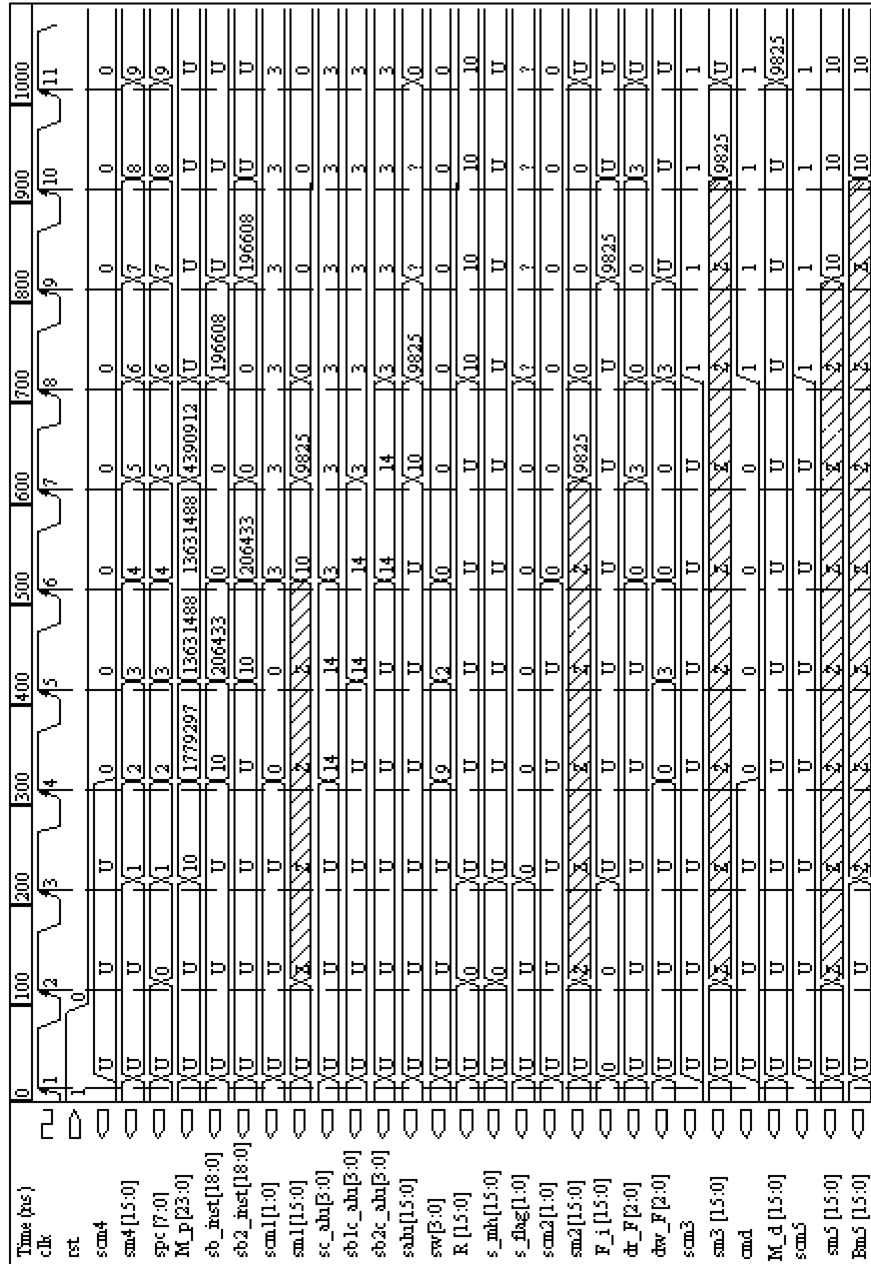


Figura 3.13: Resultados de simulación del programa 2 en la arquitectura *pipeline*.

Capítulo 4

Síntesis y análisis de resultados.

Este capítulo muestra el proceso de síntesis del microprocesador en la FPGA Spartan XC2S200E. Se inicia con una descripción del dispositivo utilizado, posteriormente se discuten las consideraciones tenidas en cuenta a nivel de *hardware* y *software*, finalmente se analizan los resultados obtenidos tanto en síntesis como en simulación.

4.1. Hardware utilizado.

Este diseño se sintetizó en un FPGA *Field Programmable Gate Array* (Arreglo de compuertas programables por campo), es un circuito con una arquitectura fija que se programa para una aplicación particular. Su estructura general contiene 3 componentes básicos [17].

- IOB (bloques de entrada salida), se encargan de la comunicación entre los pines y la lógica interna.
- CLB (bloques lógicos configurables), proveen los elementos funcionales para construir la lógica determinada, la lógica se implementa mediante bloques conocidos como generadores de funciones o LUT (Look Up Table), que cuentan con una memoria interna que permite almacenar la lógica requerida; cuando se aplica alguna combinación en las entradas de una LUT, el circuito la traduce en una dirección de memoria y envía fuera del bloque el dato almacenado en esa dirección.
- Canales de comunicación, se utilizan para comunicar los CLB entre ellos y con las terminales. de entrada salida.

Figura 4.1: Diagrama de bloques de la familia de FPGAs Spartan-IIE.

La figura 4.1[1] muestra el esquema general de un FPGA de la familia Spartan-IIE de la empresa Xilinx donde se sintetizó el microprocesador, un CLB propio de esta familia de FPGAs cuenta con 4 LUTs, cada una de las cuales con una memoria RAM de 16 x 1 bit, además 2 LUTs dentro de un CLB pueden combinarse para crear una memoria RAM de 16 x 2 bit o 32 x 1 bit [17].

Para realizar el proceso de síntesis es necesario seleccionar un dispositivo lógico programable que contenga suficientes recursos que permitan la implementación del sistema completo. En este trabajo se utilizó la FPGA Spartan 2E XC2S200E-PQ2008 de la empresa Xilinx, dado que la escuela de ingenierías eléctrica y electrónica cuenta con este dispositivo; considerando además que esta FPGA tiene suficientes recursos para implementar lógica combinacional y secuencial.
La tabla 4.1 presenta un resumen de las principales características de ésta FPGA.

[1] Tomado de [17]

Tabla 4.1: Características generales de la FPGA Spartan 2E XC2S200E-PQ208.

Recurso	Cantidad
Compuertas	200K
Celdas lógicas	5292
I/O disponibles	146
Block RAMs	56K

Para utilizar esta FPGA se contó con la tarjeta de desarrollo *Digilab2E* de la empresa Digilent que incluye [18]:

- Regulador dual de tensión (2.5V y 3.3V)
- Oscilador de 50 MHz que se utilizó como entrada de reloj del microprocesador.
- Conector de puerto paralelo DB-25 que permite programar la FPGA mediante el protocolo JTAG o mediante el puerto paralelo siguiendo el protocolo EPP contenido en el estándar IEE 1284. En este trabajo se utilizó la comunicación con el puerto paralelo.
- Conector de puerto serie usando el protocolo RS-232.
- Led de estado y pulsador usados para verificar el funcionamiento de la FPGA. El pulsador fue utilizado como señal de activación del microprocesador.
- 6 conectores de expansión, que permiten tener acceso a todos los pines de la FPGA. Para observar resultados se usaron los conectores A, C y D.

La figura 4.2 muestra el esquema general de la tarjeta Digilab 2E.

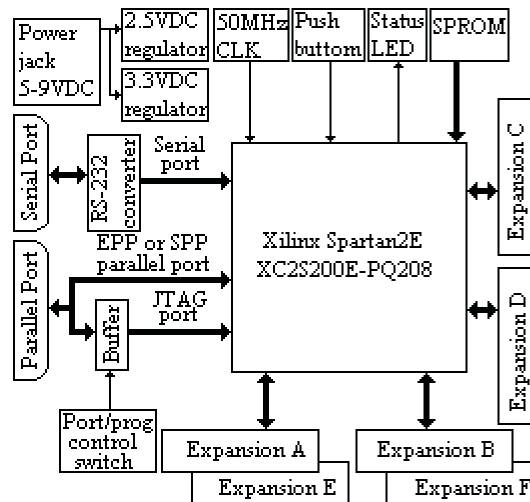


Figura 4.2: Diagrama de bloques de la tarjeta Digilab 2E.

4.2. Consideraciones a nivel de *hardware*.

Antes de realizar el proceso de síntesis es necesario adecuar las señales de entrada y salida del microprocesador a las características de la tarjeta de desarrollo utilizada. Esta sección discute las consideraciones tenidas en cuenta para adecuar estas señales.

Los datos e instrucciones de entrada en el microprocesador se insertan en la memoria de programa usando el lenguaje VHDL.
Por otro lado el microprocesador cuenta con dos señales de entrada: El reloj y la señal de reset.

Como señal de reloj se utilizó el oscilador de 50MHz que contiene la tarjeta de desarrollo. Para insertar esta señal en la FPGA es necesario utilizar un $buffer$ para mejorar la capacidad de carga de la señal

En un principio se decidió generar la señal de reset utilizando el reloj, sin embargo cuando el analizador lógico era activado para observar las señales resultantes, el microprocesador ya había ejecutado las instrucciones. Por tanto, se decidió utilizar una señal externa que junto con el reloj generan la señal de reset para activar el microprocesador y que a su vez active el analizador lógico. Esta señal es implementada usando el pulsador de la tarjeta de desarrollo, nuevamente es necesario agregar un $buffer$ para insertar esta señal en la FPGA.

Con el fin de observar los resultados de los diferentes bloques del microprocesador, se decidió que la longitud de la señal de salida sea de 24 bits, para el caso que se requiera analizar la salida de la memoria de programa.

La figura 4.3 muestra el esquema del microprocesador sintetizado.

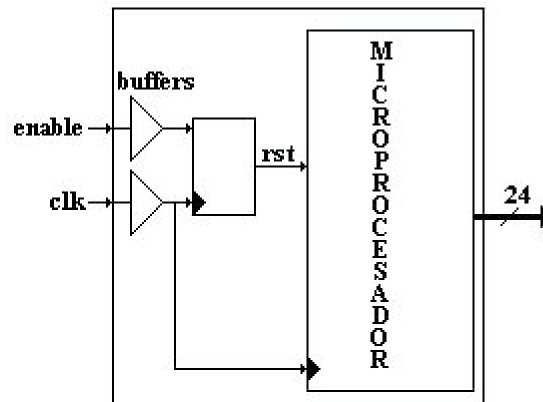


Figura 4.3: Esquema de síntesis del microprocesador.

Debido a la cantidad de señales de salida del microprocesador, se pueden presentar confusiones y errores en la conexión de estas con el analizador lógico, por tanto se decidió indicar de forma ordenada los pines de salida en la FPGA. Con el propósito de identificar fácilmente las señales de entrada y salida en la tarjeta de desarrollo se decidió utilizar las expansiones C y D para los 24 bits de salida y la expansión A para la señales de entrada.
La tabla presenta el resumen de las señales utilizadas con su ubicación en los expansores y su relación con los pines de la FPGA.

Tabla 4.2: Pines utilizados.

Señal	Tarjeta Digilab 2E	FPGA
clk (entrada)		80
clk (salida)	A39	17
enable (entrada)		77
enable (salida)	A5	64
reset	A7	62
s(0)	C5	178
s(1)	C7	175
s(2)	C9	173
s(3)	C11	168
s(4)	C13	166
s(5)	C15	164
s(6)	C17	162
s(7)	C19	160
s(8)	C21	152
s(9)	C23	150
s(10)	C25	148
s(11)	C27	146
s(12)	C29	141
s(13)	C31	139
s(14)	C33	136
s(15)	C35	134
s(16)	D5	122
s(17)	D7	120
s(18)	D9	115
s(19)	D11	113
s(20)	D13	111
s(21)	D15	109
s(22)	D17	102
s(23)	D19	100

4.3. Consideraciones a nivel de *software*.

Una vez finalizada la descripción del microprocesador en el lenguaje VHDL y verificados los resultados en el programa *Test Bench* del *software* ISE, se procedió a realizar la síntesis en el dispositivo seleccionado, sin embargo no se obtuvieron los resultados esperados y fue necesario modificar algunos bloques del microprocesador. Esto se debe a que los resultados del programa *Test Bench* son independientes del dispositivo utilizado y por tanto no asocia los posibles problemas de los recursos.
Esta sección describe la modificaciones realizadas a algunos bloques funcionales para que el microprocesador sintetizado funcione correctamente.

4.3.1. Unidad de control.

En las pruebas realizadas en el *software* para la arquitectura sin *pipeline*, las señales de la unidad de control permanecen activas hasta que un nuevo estado las modifique. Sin embargo al sintetizar el diseño en la FPGA se observó que esta señales sólo permanecen activas durante el ciclo de reloj correspondiente.

Por tanto es necesario en cada estado indicar las señales de control del estado actual y de los estados anteriores.
Esto se debe a que en la FPGA las señales son sintetizadas utilizando *flip-flops* que se actualizan con cada ciclo de reloj, es decir, un nuevo ciclo borra los resultados del ciclo anterior.

4.3.2. Memorias.

En el análisis de las memorias se mencionó que es posible direccionar hasta 64K líneas tanto para la memoria de datos como para la memoria de programa (bus de direcciones de 16 bits), sin embargo estas ocupan una cantidad considerable de recursos dentro de la FPGA, por lo tanto para efectos de síntesis se decidió acceder a las memorias utilizando un bus de direcciones de 7 bits (128 líneas).

4.4. Resultados de simulación y síntesis.

A continuación se discuten los resultados obtenidos en las pruebas realizadas tanto en simulación como en *hardware*, además se muestra el ejemplo de un programa implementado en el microprocesador.

4.4.1. Resultados de simulación.

En esta sección se analizan los resultados obtenidos en la ejecución del conjunto de instrucciones completo, utilizando el programa *test bench* del *software* ISE.

Para analizar el proceso de ejecución de las instrucciones se implementó la memoria de programa descrita en la tabla A.1[2], esta contiene las instrucciones, los resultados esperados de la ejecución de cada instrucción y los ciclos de reloj que debe emplear. La memoria de prueba contiene todas las instrucciones que puede ejecutar el microprocesador.
La figura A.1 presenta los resultados obtenidos en el programa *test bench* al ejecutarse todas las instrucciones en el microprocesador sin arquitectura *pipeline*. En ésta se puede observar los cambios de las señales de salida en cada ciclo de reloj, comprobando que todas las instrucciones se ejecutan correctamente.

Por otro lado se verificó la ejecución de las instrucciones en el microprocesador con arquitectura *pipeline*, para esto se utilizaron las instrucciones del programa anterior. Sin embargo se presentan problemas en la ejecución debido a la dependencia de datos

[2]Debido a la forma de edición del libro, el anexo A presenta las gráficas de resultados junto con la descripción de las memorias de programa utilizadas.

de algunas instrucciones y a los saltos, para solucionarlos es necesario insertar esperas entre las instrucciones en conflicto.
La tabla A.2 describe la memoria de programa utilizada para probar las instrucciones en la arquitectura *pipeline*, el número de instrucciones aumentó considerablemente.
La figura A.2 muestra la ejecución de este programa en la arquitectura *pipeline* comprobando que todas las instrucciones se ejecutan correctamente. A pesar que el número de instrucciones aumentó, el programa completo se ejecuta en menor tiempo.

4.4.2. Resultados de síntesis sobre medidas.

Para realizar la pruebas del microprocesador sintetizado se utilizó el analizador lógico TLA 7L2 de la empresa Tektronix, este permite observar señales lógicas con frecuencias de hasta 62.5 MHz. El analizador cuenta con 4 canales de entrada, cada uno con dos buses de 8 bits. La entrada de reloj del microprocesador para las pruebas de simulación es de 50MHz, esta proviene del oscilador externo de la tarjeta de desarrollo. Aunque es posible aumentar esta frecuencia programando el PLL interno de la FPGA, este proceso no se realizó, puesto que la frecuencia de medida del analizador está limitada a 62.5 MHz.

Para realizar la pruebas con el microprocesador sintetizado, se utilizaron las memorias de programa descritas en la sección anterior, nuevamente se probó el funcionamiento de las arquitecturas con y sin *pipeline*.
La figura A.3 muestra las señales de salida de algunos bloques funcionales del microprocesador sin arquitectura *pipeline*, la escala de tiempo es de 50 ns. La primera señal que se presenta es la salida del registro PC que debe aumentar en uno excepto en las instrucciones de salto. Luego se muestra la salida de la memoria de programa y finalmente se presentan las señales de salida del registro $result$, donde se comprueba que todas las instrucciones se ejecutan correctamente.

La figura A.4 por su parte, presenta los resultados de la ejecución de las instrucciones en el microprocesador con arquitectura *pipeline*, en este caso la escala de tiempo es de 10 ns. El registro PC debe aumentar cada ciclo de reloj (20 ns), excepto en las instrucciones de salto. La memoria de programa es leída cada ciclo de reloj. El registro $result$ muestra los resultados obtenidos en la ejecución de las instrucciones, observando que la ejecución en la arquitectura *pipeline* es correcta.

Es posible que se presenten alguna señales de ruido en los canales de entrada del analizador, normalmente estas señales son de una duración de $5ns$.

4.4.3. Resultados de síntesis sobre simulación.

El *software* ISE entrega un informe de los resultados obtenidos al sintetizar el diseño, presenta características como cantidad de recurso utilizado, máximo retardo, potencia, entre otras.

La tabla 4.3 resume las características obtenidas por el *software*. La frecuencia máxima se calcula teniendo en cuenta el mayor retardo, es un cálculo aproximado asociada a la ruta más larga que probablemente se encuentre en el multiplicador o las memorias por ser los bloques que más espacio ocupan dentro de la FPGA. El consumo de potencia es un valor estimado sin ejecutar instrucciones, desde aquí que el consumo de corriente se debe a la operación del reloj, pues la lógica requiere cargar y descargar capacitores asociados a las respectivas puertas de los transistores. Esta se calcula utilizando la herramienta XPower del *software* ISE.
La arquitectura *pipeline* emplea mas recursos debido a los registros de almacenamiento que utiliza.

Tabla 4.3: Resultados de simulación.

Características	Arquitectura sin *pipeline*	Arquitectura con *pipeline*
Máximo retardo	6.352 ns	5.916 ns
Frecuencia máxima*	95.419MHz	91.759MHz
Potencia estimada	24.6mW	24.6mW
Compuertas utilizadas	30.239 %	38.4245 %

* Asociada a la ruta más larga, que probablemente se encuentre en el multiplicador.

4.4.4. Programa de prueba.

Esta sección discute un programa de prueba que se implementó en el microprocesador con el fin de verificar su funcionamiento en tareas específicas.

Máximo común divisor

El programa de prueba se basa en el algoritmo de Euclides para el cálculo del máximo común divisor (MCD) entre dos números enteros.
Sean a, b dos números enteros con $a > b$, a puede ser expresada en función de b como $a = bq + r$, con $0 < r < b$; entonces se tiene que $MCD(a, b) = MCD(b, r)$.
Para la implementación del algoritmo se asume que $q = 1$, entonces $r = a - b$ con $a < b$, por tanto el cálculo del MCD se puede calcular sustituyendo el número mayor de los dos por la diferencia entre el mayor y el menor, cuando los números sean iguales es el MDC.

Para verificar la realización de este programa, se implementó la memoria de programa descrita en la tabla 4.4. Ésta calcula el MCD(94,26), se obtuvo que MCD(94,26)=2, por tanto el programa se ejecuta correctamente.
El tiempo que demora este programa en ejecutarse depende de los valores de entrada y del número de iteraciones que se deban hacer hasta llegar a $r = b$. Para el ejemplo de la tabla 4.4 el programa se ejecutó en 425 ciclos de reloj.

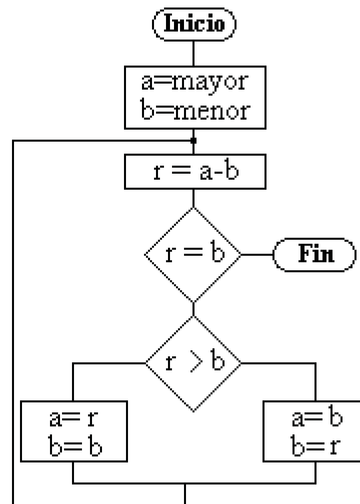


Figura 4.4: Diagrama de flujo.

Tabla 4.4: Instrucciones máximo común divisor.

PC	Inst	Cod_op	Dir_reg	Dato-Dir	Resultado
0	$R = 94$	00000	000	0000000001011110	
1	$M[0] = R$	00111	000	0000000000000000	$M[0] = a = mayor$
2	$R = 26$	00000	000	0000000000011010	
3	$M[1] = R$	00111	000	0000000000000001	$M[1] = b = menor$
4	$R = 1$	00000	000	0000000000000001	
5	$F_0 = M[R]$	00101	000	0000000000000000	
6	$R = M[0]$	00010	000	0000000000000000	$F_1 = r = a - b$
7	$Res[R, F_0]$	01010	000	0000000000000000	
8	$F_1 = R$	00100	001	0000000000000000	
9	$Res[R, F_0]$	01010	000	0000000000000000	$r = b \Rightarrow fin$
10	$Jz(22)$	11000	000	0000000000010110	
11	$Jn(17)$	11001	000	0000000000010001	
12	$R = 0$	00000	000	0000000000000000	
13	$M[R] = F_1$	01000	001	0000000000000000	$r > b \Rightarrow$
14	INC	10101	000	0000000000000000	$M[0] = a = r$
15	$M[R] = F_0$	01000	000	0000000000000000	$M[1] = b = b$
16	$JMP(4)$	10111	000	0000000000000000	
17	$R = 0$	00000	000	0000000000000000	$r < b \Rightarrow$
18	$M[R] = F_0$	01000	000	0000000000000000	$M[0] = a = b$
19	INC	10101	000	0000000000000000	$M[1] = b = r$
20	$M[R] = F_1$	01000	001	0000000000000000	
21	$JMP(4)$	10111	000	0000000000000000	
22	$R = F_1$	00001	001	0000000000000000	$R = a = MCD$

Capítulo 5

Conclusiones.

Este capítulo muestra una tabla con las especificaciones finales del microprocesador, presenta a modo de resumen las observaciones y conclusiones que se recopilaron durante el trabajo realizado y menciona las recomendaciones para trabajos futuros.

5.1. Especificaciones finales.

La tabla 5.1 resume las especificaciones obtenidas. El microprocesador está diseñado en base a una arquitectura *harvard*, en total interpreta y ejecuta 27 instrucciones RISC y opera sobre datos de 16 bits.

La frecuencia máxima estimada fue tomada de los resultados de simulación del *software* ISE 6.1i de Xiinx, basada en el cálculo del retardo de la instrucción más lenta. Igualmente la potencia corresponde a una estimación realizada por el *software* usando la herramienta XPower, esta se calcula sin ejecutar las instrucciones, la corriente consumida se debe a la lógica de distribución del reloj, la cual requiere estar cargando y descargando capacitores asociados a las respectivas puertas de los transistores.

Puede notarse que no todas las instrucciones se ejecutan en igual cantidad de ciclos de reloj, las instrucciones que utilizan el registro *result* emplean un ciclo adicional de reloj.

Tabla 5.1: Especificaciones finales.

Especificación	Resultados
Conjunto de instrucciones	Número de instrucciones: 27
	Arquitectura: RISC-*Harvard*
Modos de direccionamiento	Direccionamiento inmediato.
	Direccionamiento directo.
	Direccionamiento directo por registro.
	Direccionamiento implícito.
	Direccionamiento indexado.
Bus de datos	16 bits.
Bus de programa	24 bits.
Memoria de datos direccionable	64K palabras de 16 bits.
Memoria de programa direccionable	64K palabras de 24 bits.
Registros de propósito específico	*Result*, registro acumulador.
	MH almacena los MSB de la multiplicación.
	PC Contador de programa.
	Flag almacena las banderas.
	Cero almacena el valor cero.
Registros de propósito general	8 de 16 bits.
Fecuencia máxima estimada	95.419MHz
Potencia estimada*	26.8 mW @ 3.3V
Ciclos de reloj por instrucción	Arquitectura sin *pipeline* 5 o 4.
	Arquitectura *pipeline* 6 o 5.

Se calcula sin ejecutar las instrucciones, la corriente consumida se debe a la lógica de distribución del reloj, la cual requiere estar cargando y descargando capacitores.

5.2. Observaciones y conclusiones.

- Es posible realizar el diseño de un microprocesador partiendo de un conjunto de instrucciones, mediante la utilización de diferentes bloques funcionales que estructurados y controlados por una máquina de estados permiten la correcta interpretación y ejecución de un programa.

- Se diseñó e implementó el *datapath* y la unidad de control de un microprocesador de 16 bits que interpreta y ejecuta 27 instrucciones RISC, con un bus de programa de 24 bits, 5 modos de direccionamiento y una frecuencia máxima de operación calculada sobre la instrucción más lenta en 95.419 MHz.

- La arquitectura RISC facilitó el diseño del microprocesador y la implementación del *pipeline* debido a que todas las instrucciones se ejecutan de manera similar, en el caso del *pipeline* sólo fue necesario cambiar el momento en que unidad de control entrega las señales.

- La selección del conjunto de instrucciones define de manera general la estructura interna de un microprocesador, a partir ellas se decide los bloques funcionales que se implementarán, las conexiones que existirán entre los mismos y el modo en que cada bloque operará.

- Una vez planteado el esquema general del *datapath*, fue posible diseñar e implementar una máquina de estados que interpreta las instrucciones y envía las señales de control en el momento indicado a cada uno de los bloques del microprocesador dependiendo de la instrucción que se esté ejecutando.

- La implementación del microprocesador en el lenguaje de descripción de *hardware* VHDL se realizó siguiendo la metodología *top down*, una vez planteada una estructura general que interprete y ejecute las instrucciones seleccionadas, se procedió a diseñar e implementar cada bloque por separado desde el nivel de compuertas lógicas o de manera comportamental.

- En el diseño de los bloques funcionales debe existir un compromiso entre desempeño y recursos, es por esto que la implementación del sumador combinó dos modos de diseño tratando de mejorar el desempeño sin usar demasiados recursos.

- Se diseñó e implementó un multiplicador que puede operar sobre números con signo, para esto se partió del algoritmo general para la multiplicación hasta llegar al algoritmo de *Baugh-Wooley*, el cual permite implementar un estructura uniforme para el multiplicador.

- Una vez diseñada e implementada la estructura del microprocesador que ejecute las instrucciones, se implementó una arquitectura *pipeline* tratando de mejorar el tiempo que el microprocesador emplea en la ejecución de un programa.

- En el diseño original, la unidad de control envía las señales indicadas de acuerdo al estado en que se encuentre la instrucción, sin embargo para facilitar el diseño en la arquitectura con *pipeline*, se decidió que la unidad de control entregue todas las señales en un ciclo de reloj y mediante registros de almacenamiento estas se conservan hasta que sea necesario utilizarlas.

- La arquitectura sin *pipeline* utiliza un bloque funcional por cada ciclo de ejecución de las instrucción, mientras que en la arquitectura *pipeline* todos los bloques realizan diferentes tareas en cada ciclo de reloj, esto mejora el rendimiento del microprocesador, sin embargo se debe asegurar que el resultado de una instrucción no afecte las otras.

- Una vez diseñado e implementado el microprocesador en el lenguaje de descripción de *hardware* VHDL, se sintetizó en la FPGA Spartan 2E de la empresa Xilinx. Debido a que la tarjeta de de desarrollo usada cuenta con un reloj externo de 50MHz las pruebas de síntesis para las arquitecturas con y sin *pipeline* se realizaron a esa frecuencia.

- La ejecución de una instrucción en la arquitectura original demora 4 o 5 ciclos de reloj, al implementar la arquitectura *pipeline* este tiempo aumentó a 5 o 6 ciclos respectivamente, esto se debe a las instrucciones que utilizan la memoria de datos pues en un ciclo deben seleccionar la dirección de memoria y en el otro leerla. Lo anterior no afecta la ejecución de la instrucción en la arquitectura sin *pipeline* y el dato alcanza a escribirse correctamente; la arquitectura con *pipeline* debe esperar a que estos ciclos se ejecuten para escribir el dato. Sin embargo el tiempo de ejecución de un programa puede reducirse en la arquitectura *pipeline* por que cada ciclo de reloj inicia la ejecución de una instrucción.

- Al seleccionar el conjunto de instrucciones no se tuvo en cuenta instrucciones que permitieran direccionar la memoria desde el registro acumulador (direccionamiento indexado), sin embargo al realizar los programas de prueba, se observó la importancia de su implementación para el manejo de vectores, por tanto se decidió modificar el diseño e incluirlas.

- Se implementaron instrucciones de salto condicional e incondicional, incluyendo la instrucción "salte si es negativo, "pues esta instrucción facilita las comparaciones del tipo mayor que y menor que.

- Se efectuaron pruebas de desempeño con algunos programas con el fin de mostrar las diferencias entre la arquitectura original y la arquitectura con *pipeline*, analizando los ciclos de reloj por instrucción y el tiempo de ejecución de un programa.

- Se realizaron medidas del microprocesador sintetizado para las arquitecturas con y sin *pipeline*, utilizando el analizador lógico TLA-7L2. Se probaron cada una de las instrucciones, verificando que a 50 MHz todas se ejecutan correctamente.

- Este trabajo se realizó con el propósito de mostrar la estructura general de un microprocesador, por consiguiente el código en VHDL no se analiza con detalle, sin embargo el anexo 2 muestra algunas características de la implementación en VHDL.

5.3. Recomendaciones para trabajos futuros.

- La ALU es el bloque donde normalmente se emplea el mayor tiempo de ejecución de la instrucción, sin embargo en este trabajo se realizaron diseños de los bloques operacionales de baja complejidad para su implementación, sin profundizar en trabajos especializados. Dada la influencia de estos bloques en el rendimiento general del microprocesador se pueden estudiar topologías mejoradas para cada uno de los bloques operacionales de la ALU.

- Es posible mejorar el rendimiento de un microprocesador utilizando una arquitectura *pipeline*, sin embargo en este trabajo el rendimiento de esta arquitectura depende del número de instrucciones y el modo en que el programador las organice (tratando de evitar dependencias), un trabajo posterior debe implementar técnicas que solucionen los problemas que se pueden presentar en la ejecución de las instrucciones con *pipeline* sin comprometer el rendimiento del microprocesador.

- Un aspecto importante en el diseño de cualquier circuito es el consumo de potencia, en este trabajo se buscó reducirlo tratando de optimizar los recursos en el diseño de cada bloque funcional, sin embargo es importante analizar topologías de bloques funcionales que permitan reducir el consumo de potencia.

- El conjunto de instrucciones seleccionado es una muestra representativa de las más comunes en la mayoría de los microprocesadores, sin embargo existen otras instrucciones que no fueron tenidas en cuenta, a partir de estas se puede diseñar nuevos esquemas de microprocesadores con mayor rendimiento, o diseñar microprocesadores de propósito específico que mejoren alguna tarea en particular.

- Determinar el desempeño del microprocesador utilizando un analizador lógico que permita observar frecuencias de operación superiores a 62.5MHz, de tal forma que se utilice el reloj de la FPGA programando el PLL interno.

- Evaluar el desempeño del microprocesador mediante *benchmarks*, comparando los resultados obtenidos con otros diseños.

- Utilizar el protocolo JTAG para analizar los resultados del microprocesador sintetizado evitando los problemas que se pueden presentar conectando cada salida al analizador lógico.

- Este trabajo es un paso en el estudio del diseño de microprocesadores, se pueden analizar nuevas arquitecturas y otras formas de implementación que permitan obtener mejores prestaciones.

Bibliografía

[1] Hwang, Enoch. *Microprocessor principles and practicles with VHDL*. Books/cole. 2004

[2] Osses, Esteban. RISC/CISC. [en línea]. 66p. Disponible en web: www.inf.udec.cl/eossesa/ar-sc/t1-sisc-cisc.pdf [COnsulta: 15 Noviembre 2004.]

[3] Wilkinson, Barry. Computer Architecture design and performance. Great Britain. Prentice Hall. 1991. 375p. ISBN: 0-13-173899-2.

[4] Pollard, L. Howard. *Computer Design and Achitecture*. U.S.A. Prentice-Hall Internacional Editions. 505p. ISBN 0-13-162629-9.

[5] Protopapas, D.A. *Microcomputer Hardware Design*. U.S.A. Prentice-Hall. 1998. 510 p. ISBN 0-13-582115-0.

[6] Patterson, David; Hennesy, Jhon: et. al. Compute Architecture: A Quantitative Approach. 3 ed. U.S.A. Morgan Kuffman Pub. Junio 2002.

[7] Stallings, William. Organización y arquitectura de computadores. 5 ed. USA. Prentice Hall. 2000. 760p. ISBN 84-205-2993-1

[8] Zafra C. Juan A. Implementación en VHDL de los microcontroladores PIC-16/17. [en línea]. Proyecto para obtener el título de Ingeniero Superior de Telecomunicaciones. España. Universidad de Sevilla. 141p. Disponible en Web: www.gte.us.es/j̃on/PFCS?lang=es. [Consulta: 16 Febrero 2005].

[9] Patterson, David; Hennesy, John. Arquitectura de computadores. U.S.A. McGraw Hill Interamericana. 1993.

[10] Mozos, Daniel. Aritmética Entera. [en línea]. 60p. Disponible en Web: www.fdi.ucm.es/profesor/alfredob/AEC2004/aritm_entera.pdf [Consulta 15 Noviembre 2004].

[11] Barsten, Jeremy; Stockwell, Jeremy. *Digital Signal Processor using VLSI and FPGA's* [en línea]. USA. 2003. 47 p. Bradley University. Disponible en Web:http://cegt201.bradley.edu/projects/proj2003/dspproj/Final[Consulta 10 Diciembre 2004].

[12] Hwang, Kai. *Computer Arithmetic: Principles, architecture and design.* USA. Jhon Wile & Sons. 1979. 423p. ISBN: 0-471-06076-3.

[13] IEEE. IEEEstd 1076 *IEEE Standard VHDL Reference Manual.* IEEE Press, 1993.

[14] Mozos, Daniel. Diseño de la unidad de control. [en línea]. 66p. Disponible en Web: http://www.fdi.ucm.es/profesor/alfredob/AEC2004/unidad_control_segm_20 02.pdf. [Consulta 1 abril 2005].

[15] Rodríguez, Clemente. et. al. Microprocesadores RISC evolución y tendencias. España. Alfa omega Grupo Editor. Impreso 2000. 211p. ISBN 970-15-0505-0.

[16] Jurado, Francisco. Implementación en VHDL del microprocesador ARM9. [en línea]. Marzo 2002 Tesis. 220p Disponible en Web: www.gte.us.es/ jon/PFCS?lang=es. [Consulta 10 Noviembre 2004]

[17] XILINX. *Spartan-IIE 1.8V FPGA Family: Complete data sheet.* DS077. Julio 28, 2004. 103 p. Disponible en Web: http://direct.xilinx.com/bvdocs/publications/ds077.pdf [Consulta: 3 septiembre 2004].

[18] DIGILENT. *Digilab 2E reference manual.* D2E_rm. Marzo 24, 2004. 10p. Disponible en web: [COnsulta: 10 junio 2005].

[19] Aguirre, Miguel; Noel, Jonathan; Muños, Fernando. Diseño de sistemas digitales mediante lenguajes de descripción de *hardware*[en línea]. Octubre 2004. Disponible en Web:http://www.gte.us.es/usr/aguirre/Apuntes.pdf [Consulta: 10 febrero 2005].

[20] Alcalá, Jessica ; Maxinez, David G. VHDL El arte de programar sistemas digitales. México: continental, 2002. 352 p. ISBN 970-24-0259-X.

[21] Boluda, José; Pardo, Fernando. VHDL Lenguaje para síntesis y modelado de circuitos. México: Alfaomega Grupo Editor, impreso 1999. 238 p.ISBN 970-15-0443-7.

Apéndice A

Gráficas de resultados.

Se verificó el correcto funcionamiento de las instrucciones en las arquitecturas con y sin *pipeline*, mediante la implementación de las siguientes memorias de programa.

A.1. Memoria de prueba arquitectura sin *pipeline*.

Para probar la correcta ejecución de todas las instrucciones en la arquitectura sin *pipeline* se utilizó la memoria de programa descrita en la tabla A.1, en esta se observa la dirección de la memoria, el tipo de instrucción con su código de operación, la dirección del archivo de registros y el dato o la dirección sobre los que se debe operar, también muestra el valor de la instrucción en notación decimal, el resultado esperado y los ciclos de reloj que demora la instrucción en ejecutarse. El programa completo emplea 172 ciclos de reloj en ejecutarse.

La figura A.1 muestra ejecución de estas instrucciones en el microprocesador sin arquitectura *pipeline*. La simulación se realizó utilizando el programa *Test Bench*. Cada línea presenta las señales de salida de los diferentes bloques funcionales en cada ciclo de reloj. En esta gráfica es posible observar el paso de la instrucción por los diferentes estados. Por otro lado, la figura A.3 presenta la ejecución del programa en el microprocesador sin arquitectura *pipeline* sintetizado en la FPGA. Éstas se observaron utilizando el analizador lógico TLA 7L2. El analizador muestrea a una frecuencia de 200MHz, la resolución en este caso es de $50ns$. En la gráfica de la salida del registro PC, los cursores señalan la duración de la instrucción 6 ($R = 3$), esta se ejecuta en 5 ciclos de reloj (*Delta Time=100ns*). En la gráfica de la salida de la memoria de programa está señalado el tiempo que demora la instrucción 1 ($F_0 = 9$), la cual debe ejecutarse en 4 ciclos de reloj (*Delta Time=80ns*). Finalmente, en la figura de la salida del registro *result* se presenta la duración del ruido asociado a las entradas del analizador (*Delta Time=5ns*).

Tabla A.1: Memoria de prueba arquitectura sin *pipeline*.

Pc	Instrucción	Cod_op	dir_reg	Dato/Dir	Decimal	Resultado	Ciclos
0	$R = di$	00000	000	0010010000111000	9272	$R = 9272$	5
1	$F_i = di$	00001	000	0000000000001001	157283	$F_0 = 9$	4
2	$F_i = di$	00001	100	0000000011001000	1835208	$F_4 = 200$	4
3	$F_i = di$	00001	010	1100010011100000	1754336	$F_2 = 50400(-15136)$	4
4	$F_i = di$	00001	011	0111111100101001	1802025	$F_3 = 32553$	4
5	$M[di] = R$	00111	000	0000000000000000	3670016	$M[0] = 9272$	4
6	$R = di$	00000	000	0000000000000011	3	$R = 3$	5
7	$M[R] = F_i$	01000	011	0000000000000000	4390915	$M[3] = F_3 = 32553$	4
8	$R = F_i$	00001	000	0000000000000000	524288	$R = F_0 = 9$	5
9	$R = di$	00000	000	0000000000000000	0	$R = 0$	5
10	$F_i = M[R]$	00101	101	0000000000000000	2949120	$F_5 = M[0] = 9272$	4
11	$R = M[di]$	00010	000	0000000000000011	1048579	$R = M[3] = 32553$	5
12	$F_i = R$	00100	110	0000000000000000	2490368	$F_6 = 32553$	4
13	$Sum[R, F_i]$	01001	100	0000000000000000	4980736	$R = F + F_4 = 32753$	5
14	$Res[R, F_i]$	01010	101	0000000000000000	5570560	$R = F - F_5 = 23481$	5
15	$Mult[R, F_i]$	01011	010	0000000000000000	5898240	$R = F * F_2 = 58848$ $Mh = 60112$	5
16	$F_i = Mh$	00110	111	0000000000000000	3604480	$F_7 = 60112$	4
17	C_{a2}	01100	000	0000000000000000	6291456	$R = C_{a2}(R) = 6688$	5
18	$Xor[R, F_i]$	01101	010	0000000000000000	6946816	$R = 57024$	5
19	$And[R, F_i]$	01110	111	0000000000000000	7798784	$R = 51904$	5
20	$Or[R, F_i]$	01111	010	0000000000000000	7995392	$R = 52960$	5
21	Not	10000	000	0000000000000000	8388608	$R = Not(R) = 12575$	5
22	SD	10001	000	0000000000000000	8912896	$R = SD(R) = 6287$	5
23	SI	10010	000	0000000000000000	9437184	$R = SI(R) = 12574$	5
24	RD	10011	000	0000000000000000	9961472	$R = RD(R) = 6287$	5
25	RI	10100	000	0000000000000000	10485760	$R = RI(R) = 12574$	5
26	INC	10101	000	0000000000000000	11010048	$INC(R) = 12575$	5
27	DIS	10110	000	0000000000000000	11534336	$DIS(R) = 12574$	5
28	JMP	10111	000	0000000000011111	12058655	$Pc = 31$	4
29	INC	10101	000	0000000000000000	11010048	$INC(R) = 12575$	5
30	Not	10000	000	0000000000000000	8388608	$R = Not(R) = 52960$	5
31	Jz	11000	000	0000000000100011	12582947	$Pc = 31$ (no salta)	4
32	$R = di$	00000	000	0000000000000000	0	$R = 0$	5
33	Jz	11000	000	0000000000100011	12582947	$Pc = 35$ (si salta)	4
34	SD	10001	000	0000000000000000	8912896	$R = SD(R) = 0$	5
35	NOP	11010	000	0000000000000000	13631448	No opera	4
36	Jn	11001	000	0000000000101000	13107240	$Pc = 37$ (no salta)	4
37	$Res[R, F_i]$	01010	110	0000000000000000	5636096	$R = F - F_6 = 32983$	5
38	Jn	11001	000	0000000000101000	13107240	$Pc = 40$ (si salta)	4
39	DIS	10110	000	0000000000000000	11534336	$DIS(R) = 32982$	5
40	C_{a2}	01100	000	0000000000000000	6291456	$R = C_{a2}(R) = 32553$	5

A.2. Memoria de prueba arquitectura *pipeline*.

La tabla A.2 muestra la memoria de programa utilizada para verificar la correcta ejecución de las instrucciones en la arquitectura *pipeline*, es necesario insertar instrucciones NOP entre las instrucciones que existe algún conflicto. Este conjunto de instrucciones se ejecuta en 96 ciclos de reloj.

La figura A.2 muestra la ejecución de esta memoria de programa en el microprocesador diseñado con arquitectura *pipeline*. Los ciclos de ejecución se reducen considerablemente a pesar del aumento del número de instrucciones.

Finalmente, la figura A.4 presenta la ejecución del programa en el microprocesador con arquitectura *pipeline* sintetizado en la FPGA. Éstas se observaron utilizando el analizador lógico TLA 7L2, con una frecuencia de muestreo de 200MHz y la resolución es de 10ns.
En la gráfica de la salida del registro PC, los cursores señalan la duración de una señal de ruido (*Delta Time=5ns*).
La gráfica de la salida de la memoria de programa señala el tiempo de lectura de la instrucción 2 ($F_4 = 200$), este debe ser un ciclo de reloj (*Delta Time=20ns*).
Finalmente, en la figura de la salida del registro *result* se indica el tiempo de lectura del registro (*Delta Time=20ns*).

Tabla A.2: Memoria de prueba para la arquitectura pipeline

Pc	Instrucción	Cod_op	dir_reg	Dato/Dir	Decimal	Resultado	Ciclos
0	$R = di$	00000	000	0010010000111000	9272	$R = 9272$	6
1	$F_i = di$	00001	000	0000000000001001	157283	$F_0 = 9$	5
2	$F_i = di$	00001	100	0000000011001000	1835208	$F_4 = 200$	5
3	$F_i = di$	00001	010	1100010011100000	1754336	$F_2 = 50400(-15136)$	5
4	$F_i = di$	00001	011	0111111100101001	1802025	$F_3 = 32553$	5
5	$M[di] = R$	00111	000	0000000000000000	3670016	$M[0] = 9272$	5
6	$R = di$	00000	000	0000000000000011	3	$R = 3$	6
7	NOP	11010	000	0000000000000000	13631448	No opera	5
8	NOP	11010	000	0000000000000000	13631448	No opera	5
9	NOP	11010	000	0000000000000000	13631448	No opera	5
10	$M[R] = F_i$	01000	011	0000000000000000	4390915	$M[3] = F_3 = 32553$	5
11	$R = F_i$	00001	000	0000000000000000	524288	$R = F_0 = 9$	6
12	$R = di$	00000	000	0000000000000000	0	$R = 0$	6
13	NOP	11010	000	0000000000000000	13631448	No opera	5
14	NOP	11010	000	0000000000000000	13631448	No opera	5
15	NOP	11010	000	0000000000000000	13631448	No opera	5
16	$F_i = M[R]$	00101	101	0000000000000000	2949120	$F_5 = M[0] = 9272$	5
17	$R = M[di]$	00010	000	0000000000000011	1048579	$R = M[3] = 32553$	6
18	NOP	11010	000	0000000000000000	13631448	No opera	5

Memoria de prueba para la arquitectura *pipeline* (continuación)

Pc	Instrucción	Cod_op	dir_reg	Dato/Dir	Decimal	Resultado	Ciclos
19	NOP	11010	000	0000000000000000	13631448	No opera	5
20	$F_i = R$	00100	110	0000000000000000	2490368	$F_6 = 32553$	5
21	NOP	11010	000	0000000000000000	13631448	No opera	5
22	$Sum[R, F_i]$	01001	100	0000000000000000	4980736	$R = F + F_4 = 32753$	6
23	NOP	11010	000	0000000000000000	13631448	No opera	5
24	NOP	11010	000	0000000000000000	13631448	No opera	5
25	$Res[R, F_i]$	01010	101	0000000000000000	5570560	$R = F - F_5 = 23481$	6
26	NOP	11010	000	0000000000000000	13631448	No opera	5
27	NOP	11010	000	0000000000000000	13631448	No opera	5
28	$Mult[R, F_i]$	01011	010	0000000000000000	5898240	$R = F * F_2 = 58848$ $Mh = 60112$	6
29	NOP	11010	000	0000000000000000	13631448	No opera	5
30	NOP	11010	000	0000000000000000	13631448	No opera	5
31	$F_i = Mh$	00110	111	0000000000000000	3604480	$F_7 = 60112$	5
32	C_{a2}	01100	000	0000000000000000	6291456	$R = C_{a2}(R) = 6688$	6
33	NOP	11010	000	0000000000000000	13631448	No opera	5
34	NOP	11010	000	0000000000000000	13631448	No opera	5
35	$Xor[R, F_i]$	01101	010	0000000000000000	6946816	$R = 57024$	6
36	NOP	11010	000	0000000000000000	13631448	No opera	5
37	NOP	11010	000	0000000000000000	13631448	No opera	5
38	$And[R, F_i]$	01110	111	0000000000000000	7798784	$R = 51904$	6
39	NOP	11010	000	0000000000000000	13631448	No opera	5
40	NOP	11010	000	0000000000000000	13631448	No opera	5
41	$Or[R, F_i]$	01111	010	0000000000000000	7995392	$R = 52960$	6
42	NOP	11010	000	0000000000000000	13631448	No opera	5
43	NOP	11010	000	0000000000000000	13631448	No opera	5
44	Not	10000	000	0000000000000000	8388608	$R = Not(R) = 12575$	6
45	NOP	11010	000	0000000000000000	13631448	No opera	5
46	NOP	11010	000	0000000000000000	13631448	No opera	5
47	SD	10001	000	0000000000000000	8912896	$R = SD(R) = 6287$	6
48	NOP	11010	000	0000000000000000	13631448	No opera	5
49	NOP	11010	000	0000000000000000	13631448	No opera	5
50	SI	10010	000	0000000000000000	9437184	$R = SI(R) = 12574$	6
51	NOP	11010	000	0000000000000000	13631448	No opera	5
52	NOP	11010	000	0000000000000000	13631448	No opera	5
53	RD	10011	000	0000000000000000	9961472	$R = RD(R) = 6287$	6
54	NOP	11010	000	0000000000000000	13631448	No opera	5
55	NOP	11010	000	0000000000000000	13631448	No opera	5
56	RI	10100	000	0000000000000000	10485760	$R = RI(R) = 12574$	6
57	NOP	11010	000	0000000000000000	13631448	No opera	5
58	NOP	11010	000	0000000000000000	13631448	No opera	5
59	INC	10101	000	0000000000000000	11010048	$INC(R) = 12575$	6
60	NOP	11010	000	0000000000000000	13631448	No opera	5
61	NOP	11010	000	0000000000000000	13631448	No opera	5
62	DIS	10110	000	0000000000000000	11534336	$DIS(R) = 12574$	6

Memoria de prueba para la arquitectura *pipeline* (continuación)

Pc	Instrucción	Cod_op	dir_reg	Dato/Dir	Decimal	Resultado	Ciclos
63	NOP	11010	000	0000000000000000	13631448	No opera	5
64	NOP	11010	000	0000000000000000	13631448	No opera	5
65	JMP	10111	000	0000000001000110	12058694	$Pc = 70$	5
66	NOP	11010	000	0000000000000000	13631448	No opera	5
67	NOP	11010	000	0000000000000000	13631448	No opera	5
68	INC	10101	000	0000000000000000	11010048	$INC(R) = 12575$	6
69	Not	10000	000	0000000000000000	8388608	$R = Not(R) = 52960$	6
70	Jz	11000	000	0000000001001101	12582989	$Pc = 71$ (no salta)	5
71	NOP	11010	000	0000000000000000	13631448	No opera	5
72	$R = di$	00000	000	0000000000000000	0	$R = 0$	6
73	NOP	11010	000	0000000000000000	13631448	No opera	5
74	NOP	11010	000	0000000000000000	13631448	No opera	5
75	NOP	11010	000	0000000000000000	13631448	No opera	5
76	NOP	11010	000	0000000000000000	13631448	No opera	5
77	Jz	11000	000	0000000001010001	12582993	$Pc = 81$ (si salta)	5
78	NOP	11010	000	0000000000000000	13631448	No opera	5
79	NOP	11010	000	0000000000000000	13631448	No opera	5
80	SD	10001	000	0000000000000000	8912896	$R = SD(R) = 0$	6
81	NOP	11010	000	0000000000000000	13631448	No opera	5
82	Jn	11001	000	0000000001011110	13107294	$Pc = 83$ (no salta)	5
83	NOP	11010	000	0000000000000000	13631448	No opera	5
84	$Res[R, F_i]$	01010	110	0000000000000000	5636096	$R = F - F_6 = 32983$	6
85	NOP	11010	000	0000000000000000	13631448	No opera	5
86	NOP	11010	000	0000000000000000	13631448	No opera	5
87	NOP	11010	000	0000000000000000	13631448	No opera	5
88	NOP	11010	000	0000000000000000	13631448	No opera	5
89	Jn	11001	000	0000000001011110	13107294	$Pc = 94$ (si salta)	5
90	NOP	11010	000	0000000000000000	13631448	No opera	5
91	NOP	11010	000	0000000000000000	13631448	No opera	5
92	NOP	11010	000	0000000000000000	13631448	No opera	5
93	DIS	10110	000	0000000000000000	11534336	$DIS(R) = 32982$	6
94	C_{a2}	01100	000	0000000000000000	6291456	$R = C_{a2}(R) = 32553$	6

Apéndice B

Características del microprocesador.

En este anexo se presenta un resumen de las características del microprocesador diseñado.

- Arquitectura RISC - *Harvard*.
- 27 instrucciones.
- 5 modos de direccionamiento.
- Formato único de instrucción.
- Ejecución multiciclo
  - 5 o 4 ciclos para la arquitectura sin *pipeline*.
  - 6 o 5 ciclos para la arquitectura *pipeline*.
- 8 registros de propósito general organizados como un archivo de registros.
- 5 registros de propósito específico.
- Memoria de programa: 64K palabras de 24 bits.
- Memoria de datos: 64K palabras de 16 bits.
- Arquitectura *pipeline*.

La tabla B.1 presenta un resumen del conjunto de instrucciones seleccionado con su código de operación y nemotécnico. Todas las instrucciones tiene el mismo formato, descrito como se muestra en la figura B.1.

Tabla B.1: Resumen del conjunto de instrucciones

TIPO	COD_OP	INSTRUCCIÓN	DESCRIPCIÓN
Carga	00000	$R = di$	Carga dato inmediato en el registro *result*.
	00001	$R = F_i$	Carga un valor del archivo de registros en *result*.
	00010	$R = M[di]$	Carga un dato de memoria (de la dirección dada por un dato inmediato) en el registro *result*.
	00011	$F_i = di$	Carga un dato inmediato en el archivo de registros.
	00100	$F_i = R$	Carga el valor de *result* en el archivo de registros.
	00101	$F_i = M[R]$	Carga un valor de memoria (de la dirección dada por el registro *result*) en el archivo de registros.
Almacena	00110	$F_i = Mh$	Carga el valor de Mh en el archivo de registros.
	00111	$M[di] = R$	Almacena el valor del registro *result* en la dirección de memoria dada por un dato inmediato.
	01000	$M[R] = F_i$	Almacena un valor del archivo de registros. en la dirección dada por el registro *result*.
Aritméticas	01001	$Sum[R, F_i]$	Suma el registro *result* con el archivo de registros, el resultado lo almacena en *result*.
	01010	$Res[R, F_i]$	Resta el registro *result* con el archivo de registros, el resultado lo almacena en *result*.
	01011	$Mult[R, F_i]$	Multiplica el registro *result* con el archivo de registros, almacena en *result*, y en el registro Mh.
Lógicas	01100	C_{a2}	Realiza el complemento a2 al valor del registro *result*, almacena en *result*.
	01101	$Xor[R, F_i]$	Operación XOR entre el registro *result* y el archivo de registros, el resultado lo almacena en *result*.
	01110	$And[R, F_i]$	Operación AND entre el registro *result* y el archivo de registros, el resultado lo almacena en *result*.
	01111	$Or[R, F_i]$	Operación OR Para un valor del registro *result*.
	10000	Not	Niega el valor del registro *result*.
Desplazamiento	10001	SD	Desplaza *result* una posición hacia la derecha.
	10010	SI	Desplaza *result* una posición hacia la izquierda.
	10011	RD	Rota *result* una posición hacia la derecha.
	10100	RI	Rota *result* una posición hacia la izquierda.
INC/DIS	10101	INC	Incrementa en 1 el valor del registro *result*.
	10110	DIS	Disminuye en 1 el valor del registro *result*.
Salto	10111	JMP	Carga un dato inmediato en el registro Pc.
	11000	Jz	Si *result* es cero, carga un dato inmediato en PC.
	11001	Jn	Si *result* es negativo, carga un dato inmediato en PC.
No opere	11010	NOP	Multiplica el archivo de registros por 0, no almacena el resultado.

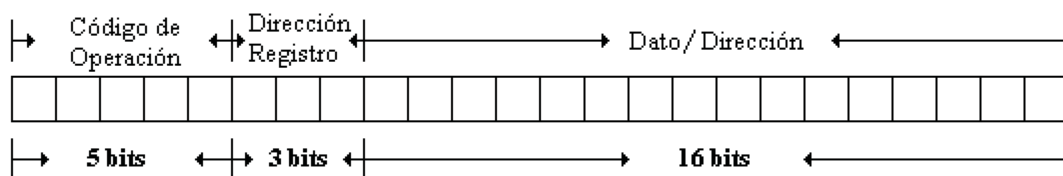


Figura B.1: Formato de las instrucciones.

Modos de direccionamiento.

El modo de direccionamiento define la forma en que el microprocesador puede interpretar el contenido de los 16 bits menos significativos del formato de instrucción. Éste microprocesador tiene los siguientes modos de direccionamiento:

- Direccionamiento Inmediato: El microprocesador toma el dato directamente del formato de instrucción.
- Direccionamiento Directo: Indica la dirección de memoria donde se encuentra el dato.
- Direccionamiento directo por registro: En algunas instrucciones se utiliza el espacio de 3 bits en el formato de instrucción para indicar el registro de propósito general que se va a utilizar.
- Direccionamiento implícito: Algunas instrucciones sólo necesitan el código de operación para ser ejecutadas.
- Direccionamiento indexado: La dirección de la memoria de datos está dada por el valor del registro $result$.

B.1. Arquitectura sin *pipeline*

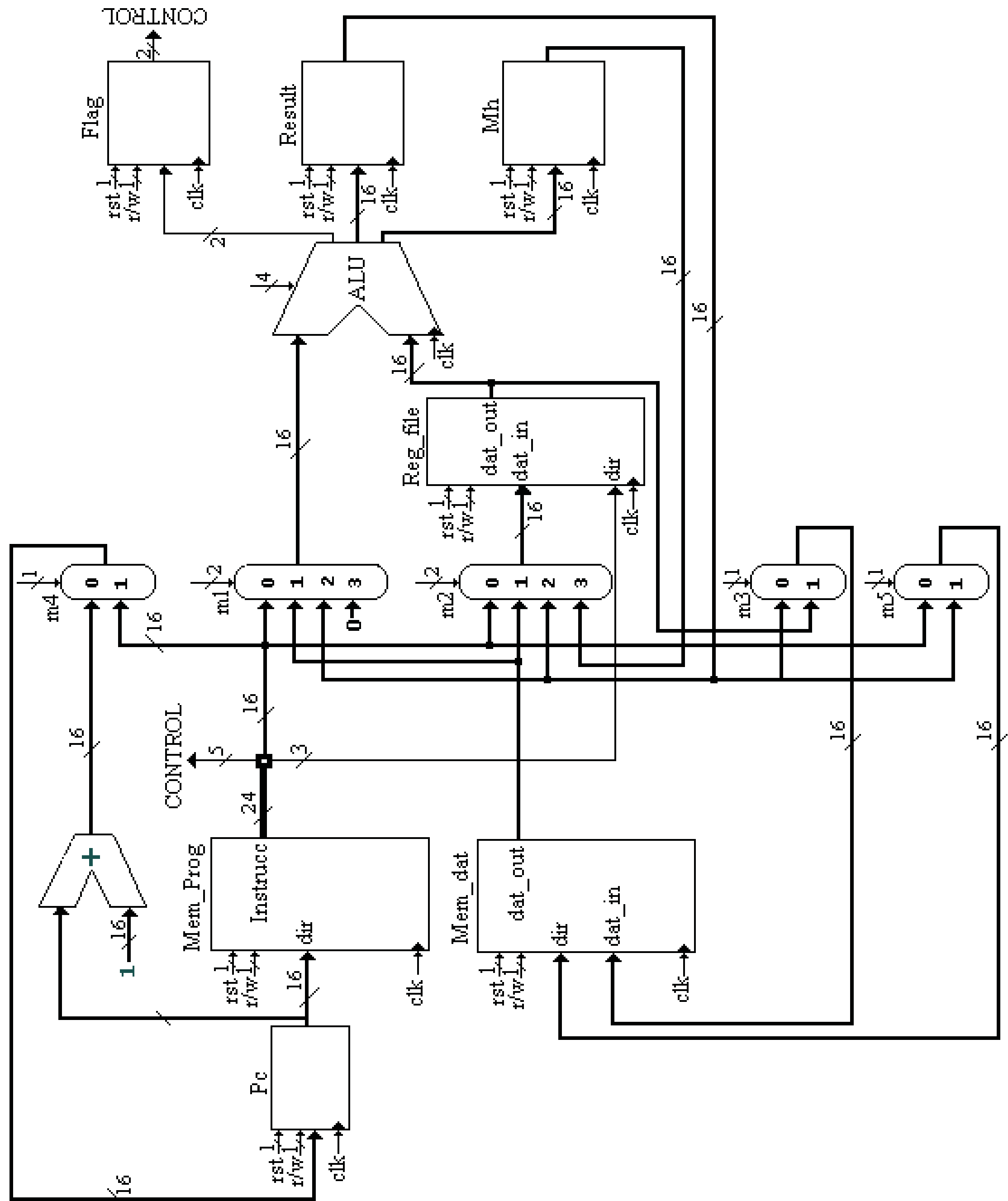


Figura B.2: Diagrama de bloques.

La figura B.2 muestra el diagrama de bloques que ejecuta las instrucciones seleccionadas en una arquitectura sin *pipeline*. A continuación se mencionan algunas características de los diferentes bloques funcionales.

Memoria.

El microprocesador se diseñó en base a una arquitectura *harvard*, es decir cuenta con una memoria de datos y una memoria de programa por separado. La memoria de programa contiene 64K líneas de 24 bits, mientras que la memoria de datos está formada por 64K líneas de 16 bits.

Bus de datos y bus de programa.

Como la estructura utilizada es *harvard* el microprocesador tiene separados el bus de datos y el bus de programa. El bus de datos define la longitud de los datos sobre los que puede operar el microprocesador, en este caso es de 16 bits. Por su parte el bus de programa define la longitud de la instrucción que es de 24 bits.

Registros.

El microprocesador cuenta con un archivo de 8 registros de propósito general y 5 registros de propósito específico que son:

- Result: Almacena los datos provenientes de la unidad aritmético lógica.
- Mh: Almacena los bits más significativos del resultado de la multiplicación.
- Flag: Almacena las banderas de cero y acarreo.
- PC: Almacena la dirección de la memoria de programa.
- Cero: Almacena el valor cero.

Todos los registros tienen una longitud de 16 bits, por ser este el tamaño del bus de datos, excepto el registro $flag$ que solo almacena dos banderas, por lo tanto su longitud es de 2 bits.

Unidad aritmético lógica.

La unidad aritmético lógica realiza las siguientes operaciones:

- Aritméticas: Suma, resta, complemento a2, multiplicación.
- Lógicas: Not, And, Or, Xor.
- Desplazamiento: Rotar derecha, rotar izquierda, shift derecha y shift izquierda.
- Incrementa-Disminuye.

Unidad de control.

La unidad de control es una máquina de 5 estados, cada ciclo de reloj cambia de estado dependiendo del estado actual y de la instrucción que esté ejecutando. La ejecución de las instrucciones es multiciclo, por tanto para la arquitectura sin *pipeline* cada ciclo de reloj la unidad de control entrega las señales adecuadas dependiendo del tipo de instrucción y del estado en que se encuentre. La tabla B.1 muestra las señales que entrega la unidad de control por cada estado según la instrucción que se esté ejecutando.

Tabla B.2: Señales de la unidad de control.

Reset (*erst*)	Lee (*e*0)	Decodifica (*e*1) Cod_op		Ejecuta (*e*2)	Almacena (*e*3)	Instrucción
		00000 $R = di$	cm1=00 cm4=0	c_alu=1110	rw_reg=00110	$R = di$
if rst=1: rst_out=11111	rw_prog=1 rw_dat=1	00001 $R = F_i$	cm4=0	c_alu=1111	rw_reg=00110	$R = F_i$
	rw_reg=10111	00010 $R = M[di]$	cm1=01 cm5=0 cm4=0	c_alu=1110	rw_reg=00110	$R = M[di]$
else rst=0: rst_out=00000		00011 $F_i = di$	cm2=00 cm4=0		rw_reg=10101	$F_i = di$
rw_reg(3)=1		00100 $F_i = R$	cm2=10 cm4=0		rw_reg=10101	$F_i = R$
		00101 $F_i = M[R]$	cm2=01 cm5=1 cm4=0		rw_reg=10101	$F_i = M[R]$
		00110 $F_i = mh$	cm2=11 cm4=0		rw_reg=10101	$F_i = Mh$
		00111 $M[di] = R$	cm3=0 cm5=0 cm4=0		rw_dat=0	$M[di] = R$
		01000 $M[R] = F_i$	cm3=1 cm5=1 cm4=0		rw_dat=0	$M[R] = F_i$
		01001 $Sum[R, F_i]$	cm1=10 cm4=0	c_alu=0000	rw_reg=00110	$Sum[R, F_i]$
		01010 $Res[R, F_i]$	cm1=10 cm4=0	c_alu=0001	rw_reg=00110	$Res[R, F_i]$
		01011 $Mult[R, F_i]$	cm1=10 cm4=0	c_alu=0011	rw_reg=00010	$Mult[R, F_i]$
		01100 C_a2	cm1=10 cm4=0	c_alu=0001	rw_reg=00110	C_{a2}
		01101 $Xor[R, F_i]$	cm1=10 cm4=0	c_alu=1011	rw_reg=00110	$Xor[R, F_i]$
		01110 $And[R, F_i]$	cm1=10 cm4=0	c_alu=1001	rw_reg=00110	$And[R, F_i]$
		01111 $Or[R, F_i]$	cm1=10 cm4=0	c_alu=1010	rw_reg=00110	$Or[R, F_i]$
		10000 NOT	cm1=10 cm4=0	c_alu=1000	rw_reg=00110	Not
		10001 SD	cm1=10 cm4=0	c_alu=0100	rw_reg=00110	SD
		10010 SI	cm1=10 cm4=0	c_alu=0101	rw_reg=00110	SI
		10011 RD	cm1=10 cm4=0	c_alu=0110	rw_reg=00110	RD
		10100 RI	cm1=10 cm4=0	c_alu=0111	rw_reg=00110	RI
		10101 INC	cm1=10 cm4=0	c_alu=1100	rw_reg=00110	INC
		10110 DIS	cm1=10 cm4=0	c_alu=1101	rw_reg=00110	DIS
		10111 JMP	cm4=1		rw_reg=10111	JMP
		11000 Jz	cm4=1 (z) cm4=0 (nz)		rw_reg=10111	Jz
		11001 Jn	cm4=1 (n) cm4=0 (nn)		rw_reg=10111	Jn
		11010 NOP	cm1=11 cm4=0	c_alu=0011		NOP

B.2. Arquitectura *pipeline*.

Para ejecución de las instrucciones en la arquitectura *pipeline* fué necesario modificar algunas características del *datapath* y de la unidad de control. La figura B.3 muestra el nuevo diagrama de bloques.
A continuación se menciona los nuevos componentes utilizados.

Registros de almacenamiento.

Para el correcto funcionamiento de la arquitectura *pipeline* es necesario incluir registros para almacenar datos que pueden ser alterados por instrucciones posteriores.

Registros de Instrucción ($Reg_inst, Breg_inst$): Almacenan los 19 bits menos significativos de la instrucción (3 bits de dirección del archivo de registros y 16 bits de datos).

Registro File ($Br_regfile$): Es un registro de 16 bits que almacena la salida del archivo de registros.

Registros de dirección (bdw_reg, $Bm5$): Almacenan la dirección de escritura del archivo de registro (bdw_reg) y la memoria de datos ($Bm5$).

Registros de control: La unidad de control entrega todas las señales en el tercer ciclo de la instrucción, sin embargo la señales de control de los multiplexores $m1$, $m2$, y $m3$ se utilizan un ciclo después por lo tanto necesitan un registro de almacenamiento. Las señales de control de la ALU (c_alu), de escritura del archivo de registros ($w_reg(1)$) y de escritura de la memoria de datos (w_dato) necesitan dos registros de almacenamiento. Las señales de escritura de los registros $resutl$, $flag$ y mh necesitan tres registros de almacenamiento. Estas señales de control son agrupadas en tres registros.

Unidad de control.

En la arquitectura *pipeline* las instrucciones deben pasar por los mismos estados que en la arquitectura original, sin embargo la unidad de control entrega todas las señales en un ciclo de reloj y estas son almacenadas hasta el momento que se necesiten. La tabla B.3 presenta las modificaciones realizadas en las señales de control. La ejecución de una instrucción por separado aumentó en un ciclo de reloj, sin embargo el *pipeline* puede mejorar considerablemente el rendimiento del microprocesador dado que cada ciclo inicia una instrucción.

Figura B.3: Diagrama de bloques para la arquitectura pipeline.

Tabla B.3: Señales de la unidad de control en la arquitectura *pipeline*

Reset ($erst$)	Lee ($e0$)	Decodifica ($e1$) Cod_op		Ejecuta ($e2$)	Almacena ($e3$)	Instrucción
		00000 $R = di$	cm1=00 cm4=0	c_alu=1110 z=0	w_reg=1001 w_dato=0	$R = di$
if rst=1: rst_out=11111	r_prog=1 r_dat=1	00001 $R = F_i$	cm4=0	c_alu=1111 z=0	rw_reg=1001 w_dato=0	$R = F_i$
	r_reg=1111	00010 $R = M[di]$	cm1=01 cm5=0 cm4=0	c_alu=1110 z=0	w_reg=1001 w_dato=0	$R = M[di]$
else rst=0: rst_out=00000		00011 $F_i = di$	cm2=00 cm4=0	z=0	w_reg=0010 w_dato=0	$F_i = di$
rw_reg=11111		00100 $F_i = R$	cm2=10 cm4=0	z=0	w_reg=0010 w_dato=0	$F_i = R$
		00101 $F_i = M[R]$	cm2=01 cm5=1 cm4=0	z=0	w_reg=0010 w_dato=0	$F_i = M[R]$
		00110 $F_i = mh$	cm2=11 cm4=0	z=0	w_reg=0010 w_dato=0	$F_i = Mh$
		00111 $M[di] = R$	cm3=0 cm5=0 cm4=0	z=0	w_reg=0000 w_dato=1	$M[di] = R$
		01000 $M[R] = F_i$	cm3=1 cm5=1 cm4=0	z=0	w_reg=0000 w_dato=1	$M[R] = F_i$
		01001 $Sum[R, F_i]$	cm1=10 cm4=0	c_alu=0000 z=0	w_reg=1001 w_dato=0	$Sum[R, F_i]$
		01010 $Res[R, F_i]$	cm1=10 cm4=0	c_alu=0001 z=0	w_reg=1001 w_dato=0	$Res[R, F_i]$
		01011 $Mult[R, F_i]$	cm1=10 cm4=0	c_alu=0011 z=0	w_reg=1101 w_dato=0	$Mult[R, F_i]$
		01100 C_a2	cm1=10 cm4=0	c_alu=0001 z=0	w_reg=1001 w_dato=0	C_{a2}
		01101 $Xor[R, F_i]$	cm1=10 cm4=0	c_alu=1011 z=0	w_reg=1001 w_dato=0	$Xor[R, F_i]$
		01110 $And[R, F_i]$	cm1=10 cm4=0	c_alu=1001 z=0	w_reg=1001 w_dato=0	$And[R, F_i]$
		01111 $Or[R, F_i]$	cm1=10 cm4=0	c_alu=1010 z=0	w_reg=1001 w_dato=0	$Or[R, F_i]$
		10000 NOT	cm1=10 cm4=0	c_alu=1000 z=0	w_reg=1001 w_dato=0	Not
		10001 SD	cm1=10 cm4=0	c_alu=0100 z=0	w_reg=1001 w_dato=0	SD
		10010 SI	cm1=10 cm4=0	c_alu=0101 z=0	w_reg=1001 w_dato=0	SI
		10011 RD	cm1=10 cm4=0	c_alu=0110 z=0	w_reg=1001 w_dato=0	RD
		10100 RI	cm1=10 cm4=0	c_alu=0111 z=0	w_reg=1001 w_dato=0	RI
		10101 INC	cm1=10 cm4=0	c_alu=1100 z=0	w_reg=1001 w_dato=0	INC
		10110 DIS	cm1=10 cm4=0	c_alu=1101 z=0	w_reg=1001 w_dato=0	DIS
		10111 JMP	cm4=1	z=0	w_reg=0000 w_dato=0	JMP
		11000 Jz	cm4=1 (z) cm4=0 (nz)	z=0	w_reg=0000 w_dato=0	Jz
		11001 NOP	cm1=11 cm4=0	c_alu=0011 z=0	w_reg=0000 w_dato=0	NOP

Apéndice C

Implementación en el lenguaje VHDL.

En este apéndice se describe de manera general la forma como se implemento el microprocesador en el lenguaje de descripción de *hardware* VHDL [1].
Se inicia mencionando unas consideraciones generales sobre el lenguaje VHDL, luego se describe los archivos .vhd utilizados para implementar cada bloque funcional.

C.1. Consideraciones del lenguaje VHDL.

Antes de describir la implementación de cada bloque funcional, y con el propósito de comprender el esquema que se siguió, es importante mencionar algunas características del lenguaje VHDL.

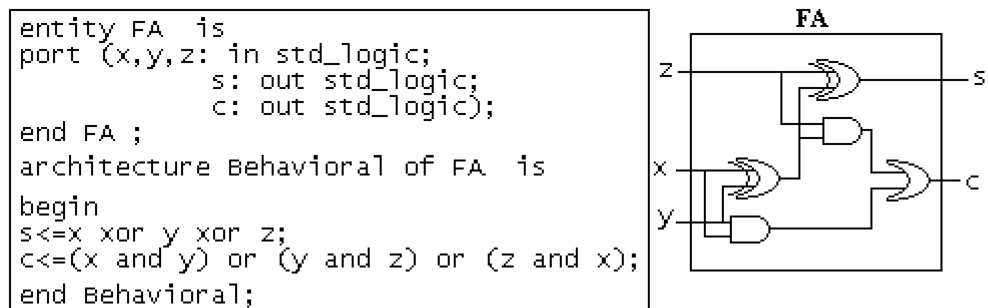
Entidad y arquitectura.

La estructura general de los porgramas en VHDL está formada por dos unidades de diseño básicas: entidad y arquitectura. La entidad es la parte del programa que define el símbolo, es decir, indica las señales de entrada y salida que tiene el sistema. La arquitectura por su parte, define el funcionamiento del módulo descrito en la entidad, describe las interconexiones entre las señales para obtener el resultado esperado. La figura C.3 muestra el ejemplo de la definición de una entidad y su arquitectura para un sumador *full-adder* a la derecha se muestra el esquema que crea esta entidad.

Definición de componentes.

Una de las ventajas que ofrece el lenguaje VHDL es que permite estructurar un diseño de forma modular, es decir subdividir un sistema en bloques menos complejos.

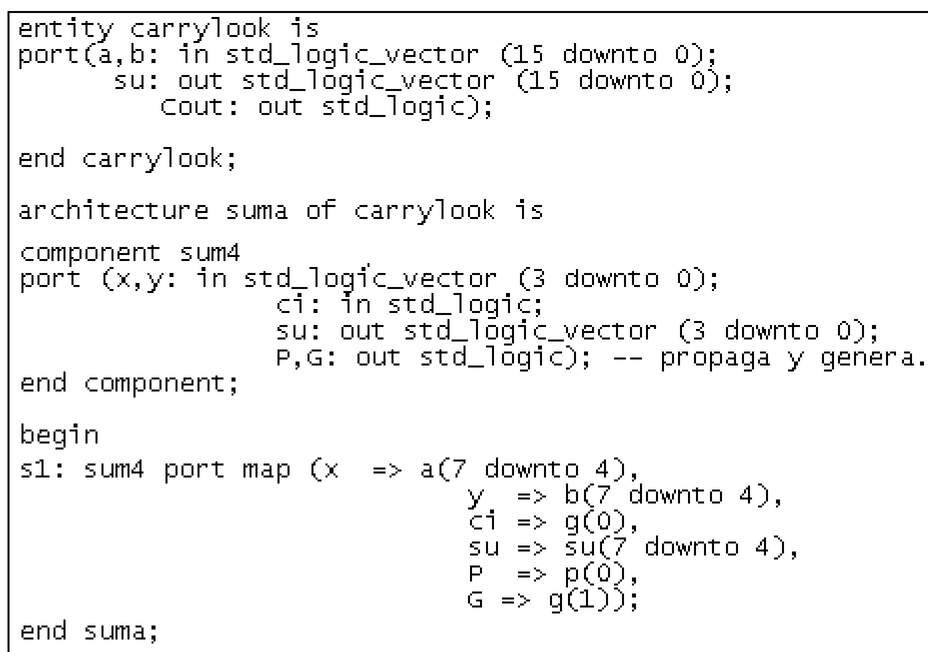
[1]Para el estudio del lenguaje VHDL se utilizaron las referencias [19, 20, 21]



```
entity FA  is
port (x,y,z: in std_logic;
            s: out std_logic;
            c: out std_logic);
end FA ;
architecture Behavioral of FA  is
begin
s<=x xor y xor z;
c<=(x and y) or (y and z) or (z and x);
end Behavioral;
```


Figura C.1: Entidad y arquitectura en VHDL.

Para esto se debe describir por separado cada bloque y luego se interconectan usando las estructuras *component* y *port map*. La figura muestra un ejemplo de la descripción de un sistema por componentes, el bloque *carrylook* contiene el bloque *sum*4.



```
entity carrylook is
port(a,b: in std_logic_vector (15 downto 0);
      su: out std_logic_vector (15 downto 0);
         Cout: out std_logic);

end carrylook;

architecture suma of carrylook is

component sum4
port (x,y: in std_logic_vector (3 downto 0);
                ci: in std_logic;
                su: out std_logic_vector (3 downto 0);
                P,G: out std_logic); -- propaga y genera.
end component;

begin
s1: sum4 port map (x  => a(7 downto 4),
                            y  => b(7 downto 4),
                            ci => g(0),
                            su => su(7 downto 4),
                            P  => p(0),
                            G => g(1));
end suma;
```

Figura C.2: Definición de componentes.

Ejecución serie: ***process*.**

El lenguaje VHDL permite manejar sentencias en serie o concurrentes; las primeras hacen referencia a sentencias que se ejecutan en el orden establecido por las líneas de comando (como en un programa normal), las sentencias concurrentes por su parte sólo se activan cuando sea necesaria su ejecución. Una de las sentencias más utilizadas es *process*, cada proceso por separado se ejecuta de manera concurrente, sin embargo las declaraciones dentro del proceso se realizan en serie. La figura muestra la sintaxis en VHDL para declarar un proceso.

```
 process (lista sensible)
    : declaraciones
begin
    :instrucciones en serie
end process;
```

Figura C.3: Sentencia process.

C.2. Descripción del microprocesador.

Para realizar esta implementación se utilizó una metodología *top down*. Una vez seleccionado el conjunto de instrucciones y planteado el diagrama de bloques del microprocesador, se implemento en el lenguaje VHDL cada uno de los bloques por separado y se comprobó su funcionamiento, finalmente se combinaron todos los componentes hasta formar la estructura completa del microprocesador.
La figura C.4 muestra el esquema general de los bloques implementados en VHDL para las dos arquitecturas, con el respectivo nombre del archivo .vhd.
A continuación se describe de forma general el contenido de los archivos.

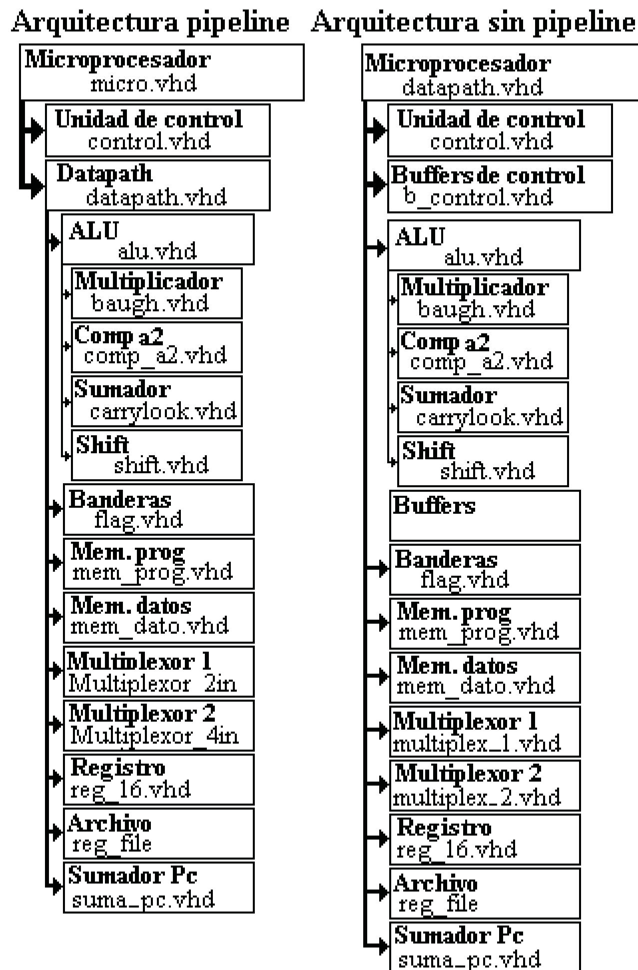


Figura C.4: Bloques implementados en el lenguaje VHDL.

ALU.

La descripción del microprocesador se inició con los bloques funcionales de la ALU, estos se implementaron desde el nivel de compuertas lógicas. Una vez implementados los bloques sumador, multiplicador, complemento a2 y shifth; el archivo *alu.vhd* mediante el uso de componentes, describe la forma como se interconectan para realizar las diferentes operaciones, además describe el multiplexor que selecciona el tipo de operación según la instrucción que se ejecute.

Memorias.

El lenguaje VHDL permite la creación de bloques de memoria RAM de cualquier tamaño mediante el comando.
type ram_type is array (127 downto 0) of std_logic_vector (15 downto 0).
El ejemplo muestra la creación de un bloque de memoria RAM de lectura y escritura. Se observa que la señal de dirección debe tener una longitud de 7 bits, para poder direccionar hasta 128 líneas de memoria, además las señales de entrada y salida de datos (*d_i*, *d_o*) son de 16 bits (15 *downto* 0), es decir que cada línea de memoria debe tener una longitud de 16 bits. La memoria tiene una señal de reloj, cada ciclo de reloj puede leer o escribir un dato.

Registros.

El archivo de registros se define como un bloque de memoria de ocho líneas de 16 bits. Un registro de propósito específico se define de manera comportamental como una señal que almacena o entrega un dato dependiendo de la señal de control y el ciclo de reloj.

Multiplexores

Los multiplexores se definieron de manera comportamental como una unidad que recibe 2 o 4 entradas y selecciona una salida dependiendo de la señal de control. Estos se implementaron usando la sentencia *case*.

Datapath.

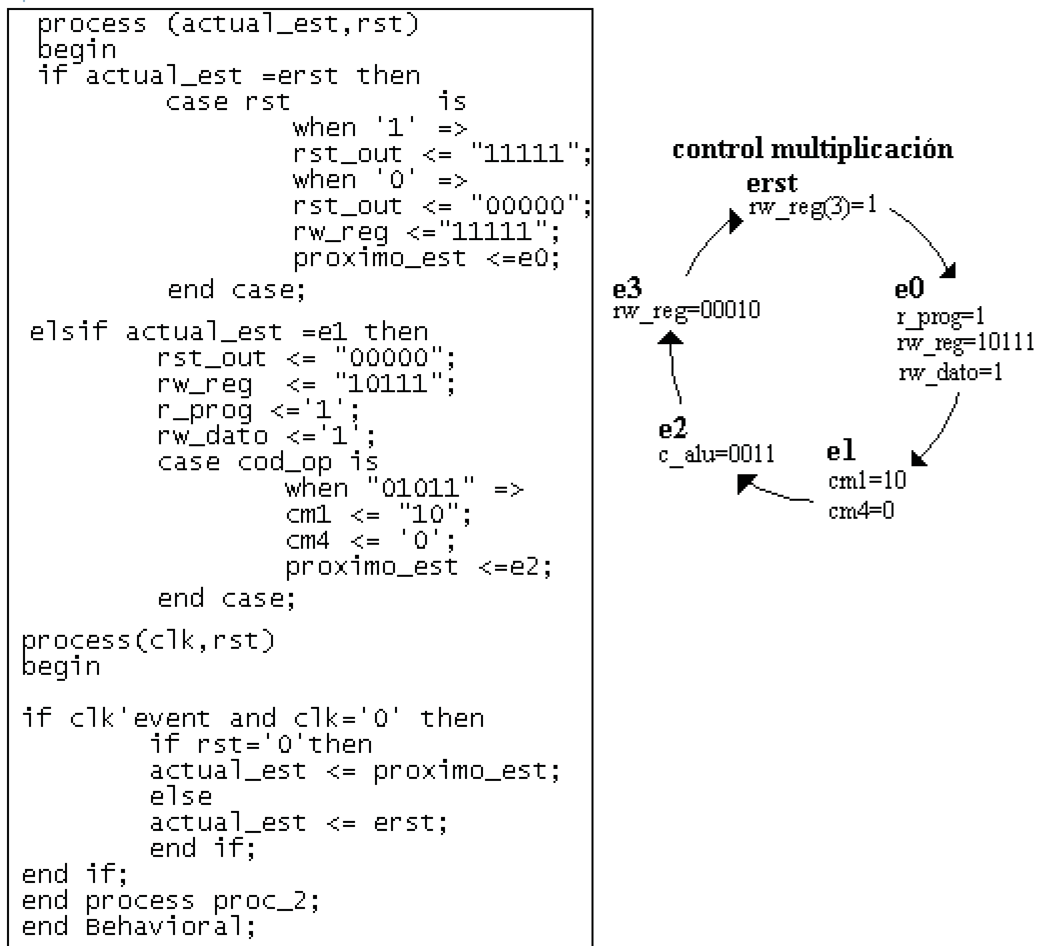
Una vez definidos los diferentes bloques funcionales, se interconectaron en el archivo *datapth.vhd* mediante la utilización de las sentencias *component* y *port map*.

Unidad de control.

La implementación de la máquina de estados que interprete las instrucciones se realizó mediante la implementación de dos porcesos, el primero indica las señales del estado actual y el segundo permite el cambio de estado por cada ciclo de reloj. La figura C.5 muestra algunos estados de la implementación de la unidad de control para la instrucción de multiplicación.

Microprocesador.

Finalmente, el archivo *micro.vhd* indica las interconexiones entre en *datapath* y la unidad de control que conforman el microprocesador completo.


```
process (actual_est,rst)
begin
if actual_est =erst then
        case rst        is
                when '1' =>
                rst_out <= "11111";
                when '0' =>
                rst_out <= "00000";
                rw_reg <="11111";
                proximo_est <=e0;
        end case;
elsif actual_est =e1 then
        rst_out <= "00000";
        rw_reg  <= "10111";
        r_prog <='1';
        rw_dato <='1';
        case cod_op is
                when "01011" =>
                cm1 <= "10";
                cm4 <= '0';
                proximo_est <=e2;
        end case;
process(clk,rst)
begin

if clk'event and clk='0' then
        if rst='0'then
        actual_est <= proximo_est;
        else
        actual_est <= erst;
        end if;
end if;
end process proc_2;
end Behavioral;
```


Figura C.5: Implementación de la unidad de control.