Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE0521 – Estructuras de Computadoras II
I ciclo 2015

Reporte 4

Laboratorio 4: Procesador segmentado con *pipelining*

Natalia Araya Campos, B00448
JeanCarlos Chavarría Hughes, B11814
Alejandro Masís Castillo, B13960
Grupo 01
Profesor: Erick Carvajal

2 de julio de 2015

Índice

1. Objetivos **4**
1.1. Objetivo general . . . 4
1.2. Objetivos especfíficos . . . 4

2. Introducción **5**

3. Manejo del proyecto **6**
3.1. División de los roles . . . 6
3.2. Cronogramas del proyecto . . . 6

4. Diseño **10**
4.1. Generalidades . . . 10
4.2. Etapa Instruction Fetch (IF) . . . 10
4.2.1. Módulo PC . . . 10
4.2.2. Instruction Memory . . . 10
4.3. Etapa Instruction Decoding (ID) . . . 11
4.3.1. Instruction Decoder . . . 11
4.3.2. Registros A y B . . . 11
4.3.3. Módulo de cálculo de branch . . . 11
4.3.4. Módulo de branch taken . . . 12
4.4. Etapa de Ejecución (EX) . . . 12
4.4.1. Módulo de controlador de ALU . . . 12
4.4.2. Módulo ALU . . . 12
4.5. Etapa de Almacenamiento en Memoria (MEM) . . . 12
4.6. Etapa de Write Back (WB) . . . 13

5. Estrategia de pruebas **14**

6. Evaluación **16**
6.1. Banco de pruebas del decodificador de instrucciones . . . 16
6.2. Banco de pruebas de la ALU . . . 16
6.3. Banco de pruebas del controlador de ALU . . . 17
6.4. Banco de pruebas del módulo *branch_ taken* . . . 17
6.5. Banco de pruebas del módulo *branch_ calc* . . . 18
6.6. Banco de pruebas de los saltos relativos (branches) . . . 19
6.7. Banco de pruebas de las memorias . . . 19
6.8. Banco de pruebas de la unidad de forwarding . . . 20
6.9. Banco de pruebas de los registros . . . 21
6.10. Banco de pruebas finales para el CPU . . . 21

7. Conclusiones **23**

A. Anexos **25**

Índice de figuras

1. Diagrama de Gannt Propuesto 8
2. Diagrama de Gannt Real 9
3. Diagrama inicial que ejemplifica el diseño 10
4. Simulación del decodificador de instrucciones 16
5. Simulación de módulo alu 17
6. Simulación de módulo controlador de alu 17
7. Simulación del módulo *branch_taken* 18
8. Simulación del módulo *branch_calc* 19
9. Evaluación de los saltos relativos 19
10. Evaluación de la memoria RAM 20
11. Evaluación del módulo Forwarding Unit 21
12. Evaluación del módulo de los registros 21
13. Evaluación del módulo CPU 22

índice de tablas

1. Esta tabla muestra la codificación del controlador de ALU para las instrucciones en las que se debe realizar una operación 13
3. Estrategia de pruebas 14
5. Estrategia de pruebas, continuación 15
6. Saltos relativos y condiciones para ser tomados 18

1. Objetivos

1.1. Objetivo general

Desarrollar conocimientos de un procesador con pipeline mediante la implementación de un módulo en un lenguaje de descripción de hardware.

1.2. Objetivos especfíficos

- Desarrollar habilidades de implementación de circuitos combinacionales y secuenciales en Verilog.
- Desarrollar habilidades de diseño de un procesador con pipeline.
- Lograr entender el funcionamiento de la segmentación de un procesador mediante su implementación en verilog.
- Determinar las dependiencias de instrucciones y como resolverlo mediante el uso de una unidad de *forwarding*.

2. Introducción

El propósito de este laboratorio consiste en brindar una metodología de trabajo en la que se permita segmentar el proceso de diseño e implementación de un proyecto de software en diferentes etapas, y además, delegar diferentes responsabilidades entre cada uno de los autores para simplificar el desarrollo de las subtareas.

Se busca diseñar e implementar un procesador con pipeline de cinco etapas para dos registros de 16 bits, utilizando el lenguaje de descripción de hardware Verilog. Las etapas definidas para el procesador son: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) y Write Back (WB).

Finalmente, se busca dar a conocer una posible opción de organización temporal para que el proyecto cumpla con especificaciones de fechas de entrega y diferentes hitos, para lo cual se diseña un esquema propuesto y luego se contrasta con el desarrollo real del proyecto.

A modo de resumen, se logró concluir con el laboratorio de manera satifasctoria mediante la implementación de diferentes módulos en Verilog, y su respectivo plan de pruebas y cronograma de tareas. Además, se desarrollaron efectivamente habilidades en las áreas de diseño, modelado y verificación de hardware en Verilog de circuitos digitales.

3. Manejo del proyecto

3.1. División de los roles

Con respecto a la metodología empleada en el presente laboratorio, se realizó una divisón de tareas o roles que cada uno de los integrantes va a tomar, tal y como se puede observar en la siguiente lista:

- **Natalia Araya Campos**: *Diseñadora*. Esta persona participará activamente en todas las tareas que se requieran, sin embargo, tendrá un interés enfocado hacia las labores diseño y desarrollo del proyecto. Entre sus principales labores destacan que debe tener una alta fluidez en la codificación y descripción de hardware mediante el lenguaje de programación utlizado, por lo que se recomienda que sea capaz de tener una visión general del proyecto para identificar los puntos claves y los puntos más complicados de desarrollar y dividirlos entre los integrantes. Será el encargado de iniciar a trabajar en la implementación del diseño en las etapas tempranas del laboratorio y también deberá delegar tareas de implementación de RTL adicionales al arquitecto y al verificador conforme el laboratorio avanza. Para las etapas finales del laboratorio se espera que los tres estudiantes estén trabajando juntos en la implementación del RTL, pero el diseñador será el encargado de asegurar que la implementación esté completa para la fecha asignada.
- **JeanCarlos Chavarría Hughes**: *Verificador*. Esta persona participará activamente en todas las tareas que se requieran, sin embargo, tendrá un interés enfocado hacia las labores de verificación y estrategias de prueba del RTL implementado. Además será el encargado de iniciar el trabajo en casos de prueba usando los diseños desarrollados y también deberá delegar tareas de verificación adicionales conforme avance el laboratorio. Finalmente está a cargo de asegurar que la verificación esté completa para la fecha asignada.
- **José Alejandro Masís Castillo**: *Arquitecto*. Esta persona participará activamente en todas las tareas que se requieran, sin embargo, tendrá un interés enfocado hacia las labores de planeamiento y arquitectura de los módulos que se tendrán que desarrollar. Tiene una responsabilidad extra pues su labor debe ser realizada los primeros días del proyecto y debe ser concluida de la misma manera. El arquitecto le dará seguimiento al progreso del proyecto y si se están cumpliendo o no las expectativas iniciales. El arquitecto estará encargado de iniciar el reporte del laboratorio, delegar tareas de escritura al diseñador y al verificador, y asegurar que todas las partes del reporte formen un documento coherente. Para las etapas finales del laboratorio, se espera que todos los estudiantes estén trabajando juntos en el reporte, pero el arquitecto estará a cargo de asegurar que el reporte esté completo para la fecha asignada.

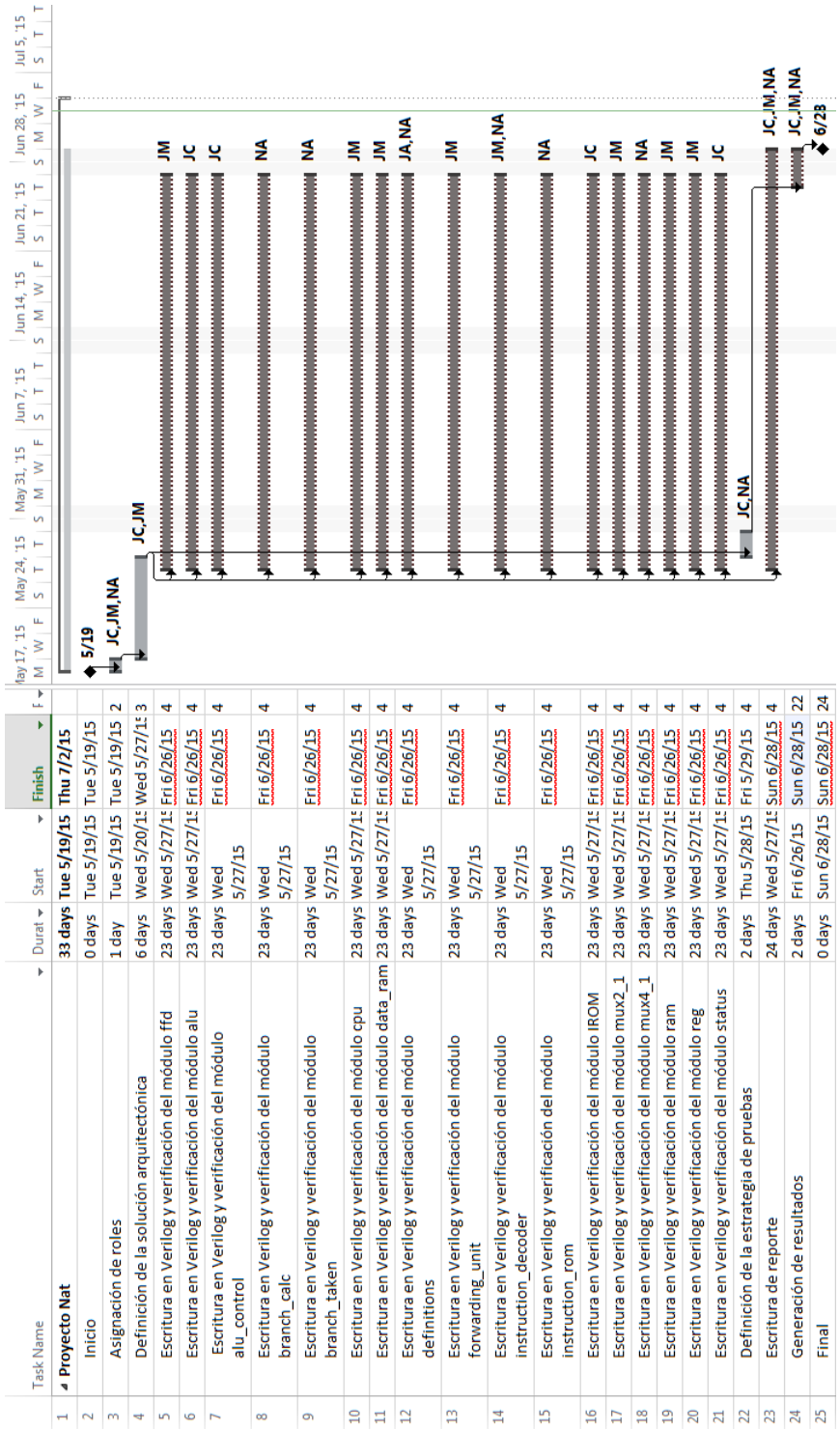
Como se plantea en las descripciones de los roles, efectivamente todos los integrantes participaron en todas las tareas, esto fue necesario ya que la complejidad de algunas tareas era mayor, como para ser asignadas a un solo integrante.

3.2. Cronogramas del proyecto

En la figura 1 se puede observar el cronograma popuesto para realizar el proyecto en diagrama de Gannt, donde se puede resaltar que debido a la naturaleza del proyecto, existe una distribución

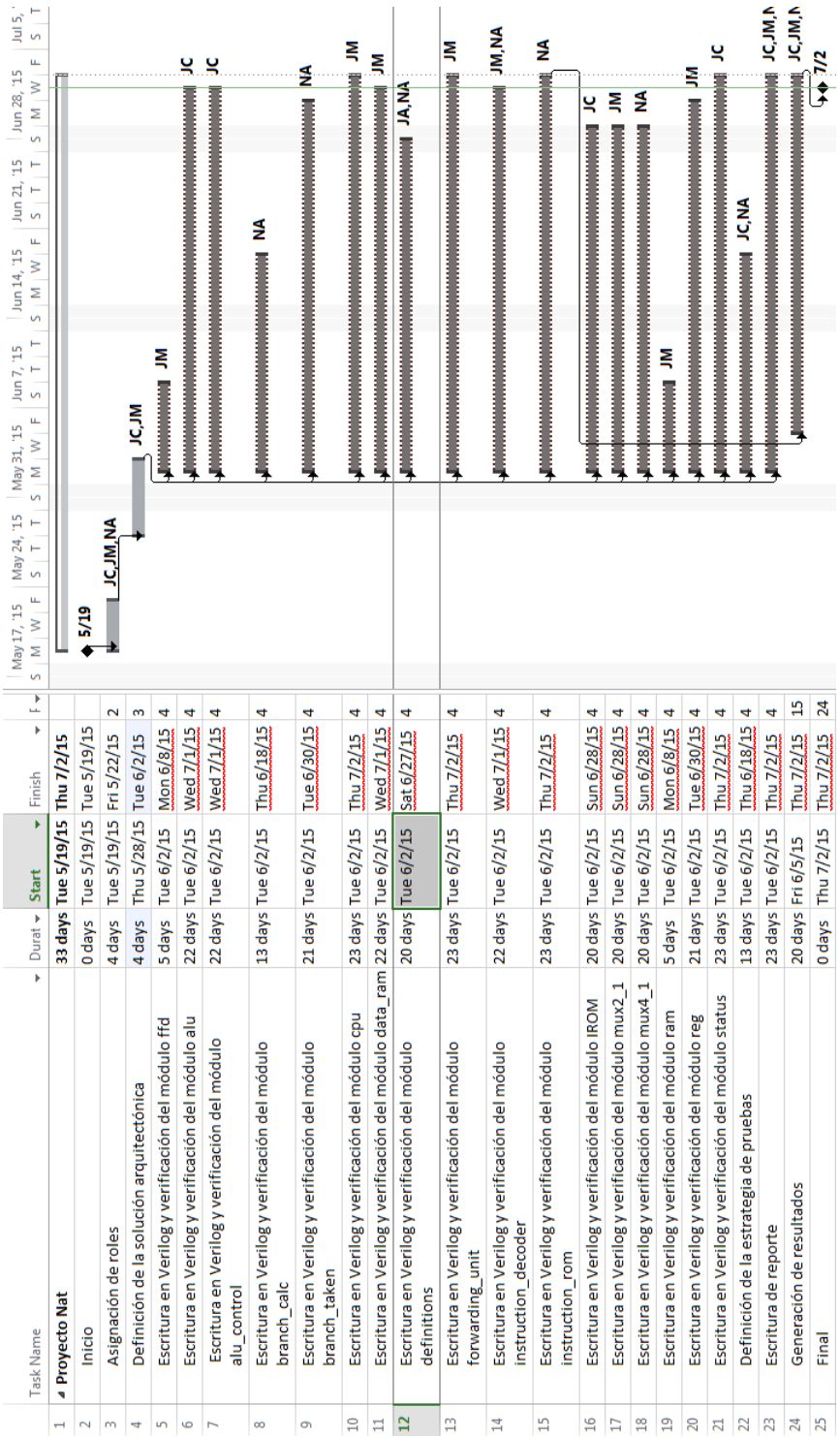
de tiempo un poco mayor en la redacción del reporte que en la implementación del diseño y la arquitectura del mismo, ya que se pretende que el reporte se escriba desde que se inician las tareas de definición de la solución arquitectónica. Por otra parte en la figura 2 se puede observar como se realizó el proyecto y en donde se observa que el tiempo utilizado para realizar la implementación del RTL fue similar al planteado, con la diferencia de que se empezaron las tareas más tarde de lo propuesto. Con esto se puede concluir que basados en la experiencia se hizo una distribución más aceertada del tiempo, aunque se atrasó el inicio y la ejecución de algunas tareas.

Figura 1: Diagrama de Gannt Propuesto



	Task Name	Durat	Start	Finish	Pred
1	**Proyecto Nat**	**33 days**	**Tue 5/19/15**	**Thu 7/2/15**	
2	Inicio	0 days	Tue 5/19/15	Tue 5/19/15	
3	Asignación de roles	1 day	Tue 5/19/15	Tue 5/19/15	2
4	Definición de la solución arquitectónica	6 days	Wed 5/20/15	Wed 5/27/15	3
5	Escritura en Verilog y verificación del módulo ffd	23 days	Wed 5/27/15	Fri 6/26/15	4
6	Escritura en Verilog y verificación del módulo alu	23 days	Wed 5/27/15	Fri 6/26/15	4
7	Escritura en Verilog y verificación del módulo alu_control	23 days	Wed 5/27/15	Fri 6/26/15	4
8	Escritura en Verilog y verificación del módulo branch_calc	23 days	Wed 5/27/15	Fri 6/26/15	4
9	Escritura en Verilog y verificación del módulo branch_taken	23 days	Wed 5/27/15	Fri 6/26/15	4
10	Escritura en Verilog y verificación del módulo cpu	23 days	Wed 5/27/15	Fri 6/26/15	4
11	Escritura en Verilog y verificación del módulo data_ram	23 days	Wed 5/27/15	Fri 6/26/15	4
12	Escritura en Verilog y verificación del módulo definitions	23 days	Wed 5/27/15	Fri 6/26/15	4
13	Escritura en Verilog y verificación del módulo forwarding_unit	23 days	Wed 5/27/15	Fri 6/26/15	4
14	Escritura en Verilog y verificación del módulo instruction_decoder	23 days	Wed 5/27/15	Fri 6/26/15	4
15	Escritura en Verilog y verificación del módulo instruction_rom	23 days	Wed 5/27/15	Fri 6/26/15	4
16	Escritura en Verilog y verificación del módulo IROM	23 days	Wed 5/27/15	Fri 6/26/15	4
17	Escritura en Verilog y verificación del módulo mux2_1	23 days	Wed 5/27/15	Fri 6/26/15	4
18	Escritura en Verilog y verificación del módulo mux4_1	23 days	Wed 5/27/15	Fri 6/26/15	4
19	Escritura en Verilog y verificación del módulo ram	23 days	Wed 5/27/15	Fri 6/26/15	4
20	Escritura en Verilog y verificación del módulo reg	23 days	Wed 5/27/15	Fri 6/26/15	4
21	Escritura en Verilog y verificación del módulo status	23 days	Wed 5/27/15	Fri 6/26/15	4
22	Definición de la estrategia de pruebas	2 days	Thu 5/28/15	Fri 5/29/15	4
23	Escritura de reporte	24 days	Wed 5/27/15	Sun 6/28/15	4
24	Generación de resultados	2 days	Fri 6/26/15	Sun 6/28/15	22
25	Final	0 days	Sun 6/28/15	Sun 6/28/15	24

Figura 2: Diagrama de Gannt Real



	Task Name	Durat	Start	Finish	F
1	**Proyecto Nat**	**33 days**	**Tue 5/19/15**	**Thu 7/2/15**	
2	Inicio	0 days	Tue 5/19/15	Tue 5/19/15	
3	Asignación de roles	4 days	Tue 5/19/15	Fri 5/22/15	2
4	Definición de la solución arquitectónica	4 days	Thu 5/28/15	Tue 6/2/15	3
5	Escritura en Verilog y verificación del módulo ffd	5 days	Tue 6/2/15	Mon 6/8/15	4
6	Escritura en Verilog y verificación del módulo alu	22 days	Tue 6/2/15	Wed 7/1/15	4
7	Escritura en Verilog y verificación del módulo alu_control	22 days	Tue 6/2/15	Wed 7/1/15	4
8	Escritura en Verilog y verificación del módulo branch_calc	13 days	Tue 6/2/15	Thu 6/18/15	4
9	Escritura en Verilog y verificación del módulo branch_taken	21 days	Tue 6/2/15	Tue 6/30/15	4
10	Escritura en Verilog y verificación del módulo cpu	23 days	Tue 6/2/15	Thu 7/2/15	4
11	Escritura en Verilog y verificación del módulo data_ram	22 days	Tue 6/2/15	Wed 7/1/15	4
12	Escritura en Verilog y verificación del módulo definitions	20 days	Tue 6/2/15	Sat 6/27/15	4
13	Escritura en Verilog y verificación del módulo forwarding_unit	23 days	Tue 6/2/15	Thu 7/2/15	4
14	Escritura en Verilog y verificación del módulo instruction_decoder	22 days	Tue 6/2/15	Wed 7/1/15	4
15	Escritura en Verilog y verificación del módulo instruction_rom	23 days	Tue 6/2/15	Thu 7/2/15	4
16	Escritura en Verilog y verificación del módulo IROM	20 days	Tue 6/2/15	Sun 6/28/15	4
17	Escritura en Verilog y verificación del módulo mux2_1	20 days	Tue 6/2/15	Sun 6/28/15	4
18	Escritura en Verilog y verificación del módulo mux4_1	20 days	Tue 6/2/15	Sun 6/28/15	4
19	Escritura en Verilog y verificación del módulo ram	5 days	Tue 6/2/15	Mon 6/8/15	4
20	Escritura en Verilog y verificación del módulo reg	21 days	Tue 6/2/15	Tue 6/30/15	4
21	Escritura en Verilog y verificación del módulo status	23 days	Tue 6/2/15	Thu 7/2/15	4
22	Definición de la estrategia de pruebas	13 days	Tue 6/2/15	Thu 6/18/15	4
23	Escritura de reporte	23 days	Tue 6/2/15	Thu 7/2/15	4
24	Generación de resultados	20 days	Fri 6/5/15	Thu 7/2/15	15
25	Final	0 days	Thu 7/2/15	Thu 7/2/15	24

4. Diseño

4.1. Generalidades

La figura 3 muestra el diseño inicial para el procesador de 5 etapas de pipelining con todas las especificaciones dadas.

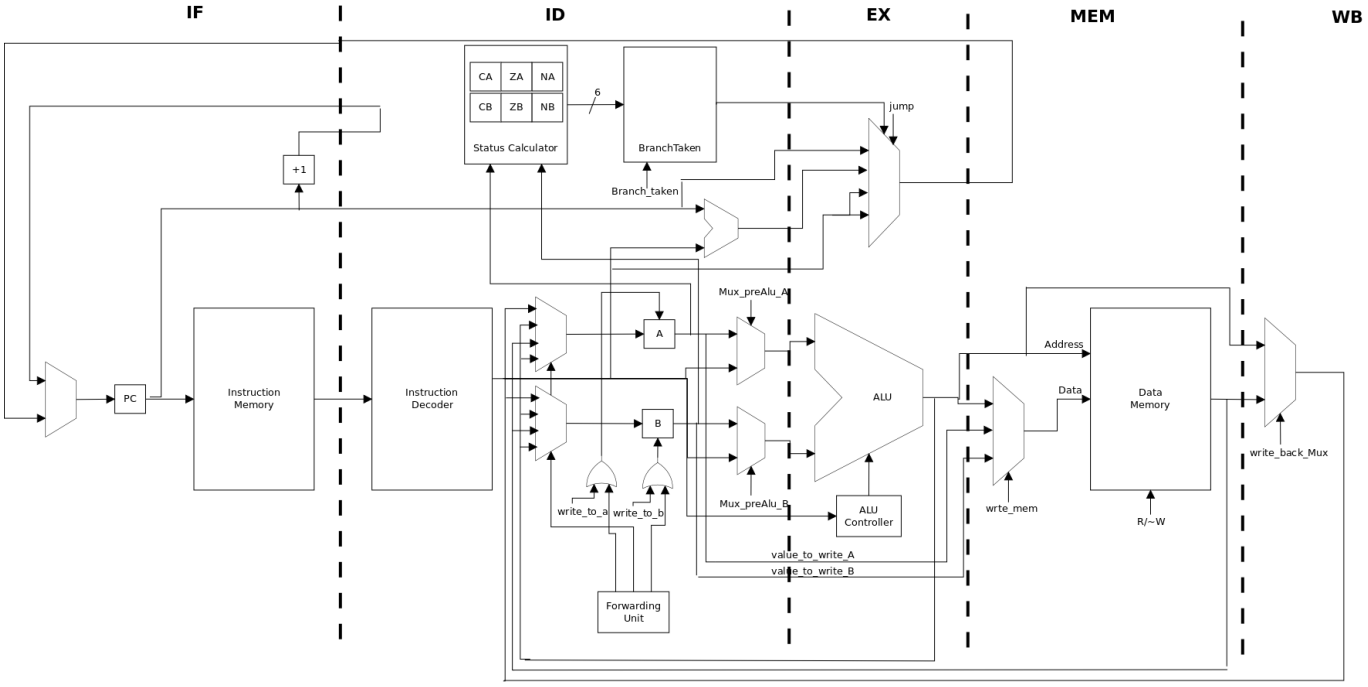


Figura 3: Diagrama inicial que ejemplifica el diseño

A continuación se va a explicar la función de cada uno de los bloques, dividido en cada una de las etapas del pipelining.

4.2. Etapa Instruction Fetch (IF)

En esta etapa se obtiene de la memoria de instrucciones la instrucción que se va a procesar en ese siguiente ciclo.

4.2.1. Módulo PC

El módulo PC es un registro sencillo, encargado de almacenar la dirección de memoria de la que se quiere obtener la dirección. Esta, en forma normal, incrementa uno cada ciclo, para apuntar a la siguiente posición de memoria, y leer la instrucción siguiente; sin embargo, con los branches y los jumps, existe una lógica para calcular la nueva dirección.

4.2.2. Instruction Memory

Es una memoria asincrónica sencilla, con 1024 posiciones de memoria (10 bits de bus de direcciones) y sus datos son de 16 bits, los cuales, el decodificador los va a interpretar acorde a lo especificado en el enunciado, según sea la instrucción.

4.3. Etapa Instruction Decoding (ID)

En esta etapa, principalmente, se decodifica la instrucción recibida, y se configuran todas las señales de control acorde a lo necesitado dependiendo de la instrucción.

4.3.1. Instruction Decoder

Se encarga de decodificar la instrucción y de definir todas las señales de control. Las señales de control manejadas por este módulo son:

- write_to_a y write_to_b: Cuando se da llega a la etapa de Write Back (explicada posteriormente), si esta señal está en alto, permite que se realice una escritura en alguno de los dos registros, colocando la señal respectiva en alto.
- Mux_preAlu_A y Mux_preAlu_B: Señales de control que manejan los multiplexores colocados previos a la ALU. Estos definen qué dato se ingresa a la ALU, si el proveniente del registro A (o del B, según el caso), o el proveniente directamente del Decoder. Esta elección varía dependiendo del tipo de instrucción que se debe ejecutar. Por ejemplo: para un acceso a memoria, se requiere de una dirección, proveniente directamente del Instruction Decoder; en cambio, para una suma, se necesita del dato contenido en A o en B.
- Branch_taken: Esta señal se encarga de comunicarle al Demux qué tipo de branch es el que se quiere evaluar, y dependiendo del que sea, se analiza el bit de status correspondiente.
- jump: Esta señal se coloca en alto únicamente en los casos en que la instrucción que se debe ejecutar sea un JMP (Jump). Esto coloca al sistema en un modo en el que se puede modificar el contador del programa a un nuevo valor, definido por la instrucción leída.
- write_mem: Esta señal define cual dato es el que se quiere ingresar en la memoria de datos: si el dato contenido en A, en B, o el proveniente de la ALU. Esto se utiliza para los casos en que se realiza un Store.
- R/$\tilde{\text{W}}$: Esta señal controla si la memoria de datos está en modo de lectura o de escritura.
- write_back_mux: Define si el dato que se escribe de vuelta en los registros es el proveniente de la ALU o de la memoria de Datos.

4.3.2. Registros A y B

Funcionan como Acumuladores. Son registros que almacenan los datos provenientes de la etapa de Write Back. Se controlan por las señales write_to_a y write_to_b, respectivamente, como se explicó anteriormente.

4.3.3. Módulo de cálculo de branch

Este módulo se encarga del cálculo del nuevo valor del contador de programa, en caso de que el branch sea tomado finalmente. Este módulo consiste en una suma del contador de programa actual, con el valor indicado en la instrucción que se quiere desplazar relativamente ya sea hacia arriba o hacia abajo, dependiendo del valor del sétimo bit.

4.3.4. Módulo de branch taken

Este módulo se encarga de definir si el branch es tomado o no, dependiendo de los 6 bits de status que se tienen especificados: C_A, Z_A, N_A, C_B, Z_B, y N_B. Dependiendo del tipo de instrucción que sea, se referencia alguno de los bits de status y se define si el branch se toma o no. Además se encarga de efectuar el cálculo del nuevo PC, por lo que debe ser capaz de tomar en cuenta el tipo de instrucción que esta ejecutándose, y el valor de las banderas antes mencionadas. Se trata de usar una señal de control que elija en un mux 4:1, donde las entradas son PC+1, Branch Magnitude, JMP dirección, JMP dirección.

4.4. Etapa de Ejecución (EX)

Se realiza el cálculo necesario según sea la instrucción especificada. Las diferentes operaciones se realizan en la ALU.

4.4.1. Módulo de controlador de ALU

Este es el módulo del procesador que se encarga de manejar todas las señales de control que se utilizan para controlar la unidad lógico arimética. Su función principal se puede resumir en que debe ser capaz de determinar la operación que se debe efectuar a partir de los 6 bits más significativos de la instrucción decodificada, por lo tanto tiene una entrada de 6 bits y una salida de 3 bits.

4.4.2. Módulo ALU

Este es el módulo del procesador que se encarga de efecuar los cálculos aritméticos y la ejecución del *pipeline* de manera que se cumpla con la etapa (EX). En términos generales es un módulo que recibe dos entradas de 10bits cada una y tiene una salida de 10bits y una entrada de señal de control proveniente del módulo controlador de alu de acuerdo a como se puede observar en la tabla 1. Cualquier operación que no está presente en la tabla se asigna con el valor de $7 en cual es usado para decir que la alu no debe realizar ninguna operación con los operandos, lo cual es así para todos los saltos condicionales e incondicionales, así como tambien para la **NOP**.

4.5. Etapa de Almacenamiento en Memoria (MEM)

La etapa de MEM es en la cual se realiza la escritura de datos en la memoria principal de datos, se puede observar que cuando se ocupa realizar una instruccion de escribir desde los registros hasta la memoria, los datos se almacena en el registro EX/MEM, luego para el siguiente ciclo de reloj, los datos se transportan hacia la memoria.

De igual manera la dirección de memoria calculada por la ALU se almacena en el registro EX/-MEM y se ingresa a la memoria principal para el siguiente ciclo de reloj. Así como tambien cuando toca almacenar resultados de operaciones aritméticas desde la ALU hasta la memoria de datos.

La única ocación en que la etapa EX/MEM no utiliza la memoria principal es cuando se realiza una instrucción de salto, o cuando se debe almacenar un dato en uno de los registros en lugar de la memoria principal.

Controlador de ALU, valores en octal

Instrucción	Código	Decodificado	Instrucción	Código	Decodificado
LDA	0	0	SUBA	12	3
LDB	1	0	SUBB	13	3
LDCA	2	0	SUBCA	14	3
LDCB	3	0	SUBCB	15	3
STA	4	0	ANDA	16	2
STB	5	0	ANDB	17	2
ADDA	6	1	ANDCA	20	2
ADDB	7	1	ANDCB	21	2
ADDCA	10	1	ORA	22	5
ADDCB	11	1	ORB	23	5
ASLA	26	4	ORCA	24	5
ASRA	27	6	ORCB	25	5

Tabla 1: Esta tabla muestra la codificación del controlador de ALU para las instrucciones en las que se debe realizar una operación

4.6. Etapa de Write Back (WB)

La etapa del procesador que se encarga de escribir desde los registros hasta la memoria principal o de la ALU hacia los registros deben pasar por esta etapa por lo tanto cuenta con ciertas señales de control como se puede observar en el diagrama arquitectónico.

Para esta etapa realmente la duración del ciclo no esta relacionada con el manejo de la instrucción, sino más bien con la duración de la memoria principal para las lecturas y escrituras. En este caso dado que la simualación es conductual y no presenta retardos de propagación, no se tiene en cuenta este factor, sin embargo, si es importante hacer la aclaración de que en un caso real, la mayor parte del tiempo que se encuentre en este ciclo es por acceso de la memoria principal.

5. Estrategia de pruebas

Con el fin de comprobar el adecuado comportamiento del procesador con etapas segmentadas, se define el siguiente plan de pruebas el cual busca comprobrar que además de que cada módulo funciona de manera adecuanda por separado, también que todo el procesador funciona bien en cualquier caso, con solo evaluar los puntos críticos.

Se debe comprobar que cada módulo diseñado cumple con lo establecido en el diagrama arquitectónico y en la definición dada en la sección de Diseño, donde los resultados de la simulación obteniddos deben mostar lo que se observa en la Tabla 3 y 5.

Módulo	**Resultado Esperado**
Contador de programa	Debe mostrar un diagrama temporal como el PC funciona de manera sincrónica de acuerdo a una señal de reloj del sistema. Además debbe mostrar que la salida corresponde a direcciones de memoria sin signo y de 10 bits en formato decimal.
Memoria de instrucciones y de datos	Se debe mostrar el acceso de al menos 10 posiciones de memoria diferentes, incluyendo la primer posición y la última, y se debe mostrar el contenido de cada posición. Además se debe intentar ingresar a una posicióon inválida (negativa y mayor a 1024) y comprobar que no se puede acceder
Decodificador de instrucciones	Se debe mostrar la decodificación de todas las señales de control para todas las instrucciones posibles. Recuerde que todas las señales deben estar definidas y cualquier *switch case* debe contener su respectivo *default state*, esto para que el compilador infiera un RTL. Las señales de control a mostrar necesariamente son: *write to a*, *write to b*, *Mux Pre ALU A*, *Mux Pre ALU B*, *branch taken*, *jump*, *write mem*, *R/W* y *write back mux*.

Tabla 3: Estrategia de pruebas

Registros A y B	Se debe mostrar de manera independiente que se puede almacenar valores distintos y colocar su valor en la salida del módulo dependiento de la combinación de las señales de control: *write to a* y *write to b*. Por lo tanto, se debe hacer una simulación con todas las posibles combinaciones de las señales de control.
Módulo de salto (branch)	Se debe mostrar el funcionamiento de este módulo para todas las posibles intrucciones de salto, tomando en cuenta las combinaciones de banderas que hacen que el *branch* se tome y no se tome, incluyendo las instrucciones JUMP y NOP. Es importante destacar que se debe mostrar que el PC nuevo es la suma de PC actual más la magnitud del salto cuando es suma, o resta cuando es substraccióon y además se debe mostrar que PC nuevo es PC actual + 1 cuando el salto no se toma. Así, por cada instrucción hay que simular 3 combinaciones posibles, tomando el salto y sumando dirección, tomando el salto y restando dirección, o no tomando el salto.
Módulo Controlador de ALU	Se debe mostrar la decodificación para todas las instrucciones del procesador, su respectivo valor de entrada de control a la ALU y su respectiva salida en un diagrama temporal. Recuerde que es un módulo completamente combinacional por lo que las señales no están relacionadas con ningún reloj del sistema.
Módulo ALU	Se debe mostrar un un diagrama temporal todas las posibles operaciones que se puedan realizar en la ALU, así como también el caso en el que la ALU no desarrolla ninguna tarea. Dado que las entradas siempre son positivas, todos los 10 bits de entrada representan la magnitud de un valor, si el resultado de la substracción es menor que cero, se debe indiciar de alguna manera, que la bandera correspondiente debe ser ajustada.

Tabla 5: Estrategia de pruebas, continuación

6. Evaluación

Para evaluar las pruebas planteadas en el apartado anterior, se ejecutaron los bancos de pruebas y se graficaron las señales resultantes (archivo signals.vcd), para analizar los resultados obtenidos.

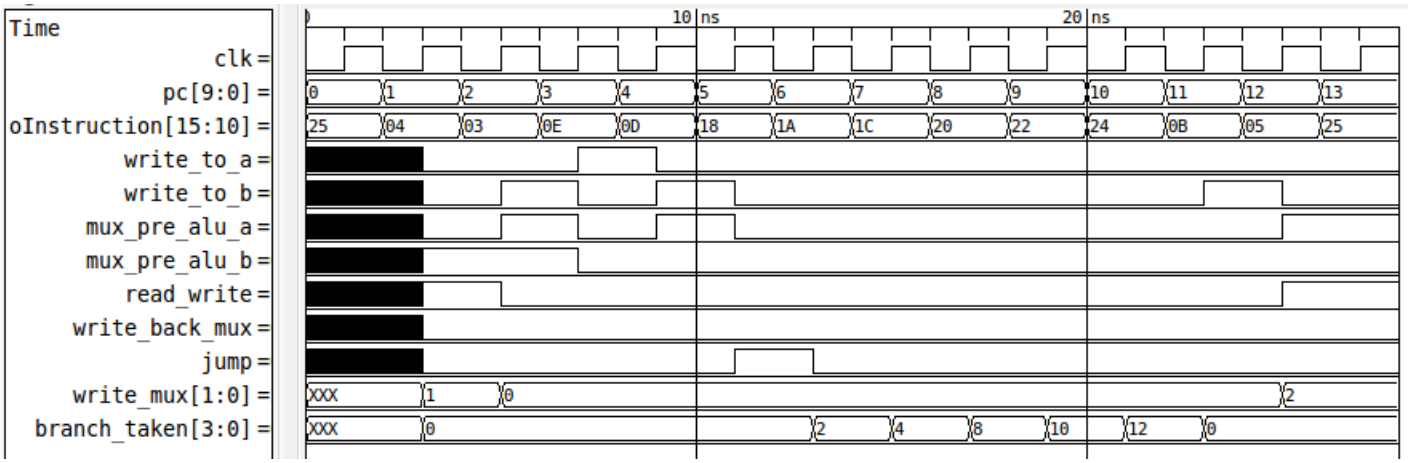
En los siguientes apartados se muestran los resultados obtenidos para cada uno de los multiplicadores.

6.1. Banco de pruebas del decodificador de instrucciones

El decodificador de instrucciones es el módulo que se encarga de ajustar todas las señales de control del procesador de acuerdo a la instrucción que se está procesando en cada ciclo de reloj, de manera tal que debe ser capaz de obtener la decodificación de la instrucción, las señales de control: *write to a, write to b, mux pre alu a, mux pre alu b, read write, write back mux, jump, write mux y branch taken* de acuerdo al avance del contador de programa *pc*.

Es de esta manera como se realizó la simulación de las instrucciones que modifican las señales de control y se puede observar en la Figura 4 como efectivamente se tienen los resultados esperados de acuerdo con el diseño del procesador.

Figura 4: Simulación del decodificador de instrucciones

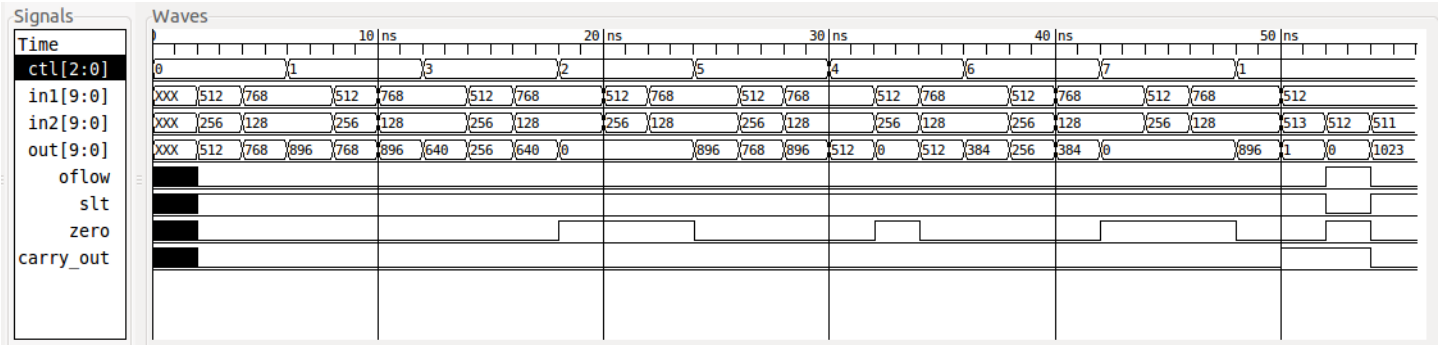


6.2. Banco de pruebas de la ALU

El módulo ALU es el que se encarga de realizar el cálculo aritmético de ciertas instrucciones que lo requieren y además obtener el valor de la señal de carry y la señal de cero. Tambien se encarga de realizar el cálculo de las direcciones de memoria para instrucciones que tienen que ver con el acceso o ingreso a la memoria principal.

De acuerdo con lo anterior y con la descripción proporcionada en la sección de diseño se puede observar los resultados de la simulación efectuada en la Figura 5 donde cabe destacar que el valor de los operandos *in1, in2* y el resultado final *out* se observa en formato decimal, mientras que las señales de *carry* y *zero* son bits.

Figura 5: Simulación de módulo alu

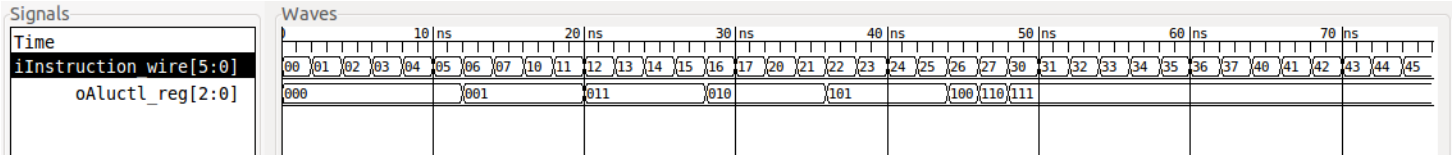


6.3. Banco de pruebas del controlador de ALU

El módulo controlador de la ALU es bastante importante en tanto que se encarga de obtener los bits de control que se encargan de decirle al módulo ALU la tarea que debe desarrollar, por lo que en la evaluación de este módulo se debe de tener como entrada todas las posibles instrucciones que se pueden ejecutar en el procesador y obtener a partir del código de instrucción, el código de operando que debe implementar la ALU.

El resultado de la simulación se puede observar en la Figura 6 donde se puede destacar que los códigos de presentan en formato octal y además en orden de acuerdo con la descripción presentada en la sección de diseño.

Figura 6: Simulación de módulo controlador de alu



6.4. Banco de pruebas del módulo *branch_ taken*

El módulo *branch_ taken* se encarga de revisar cual branch se desea tomar y si se cumplen las condiciones para tomarlo, esto por medio de los bits de status especificados: C_A, N_A, Z_A, C_B, N_B, Z_B.

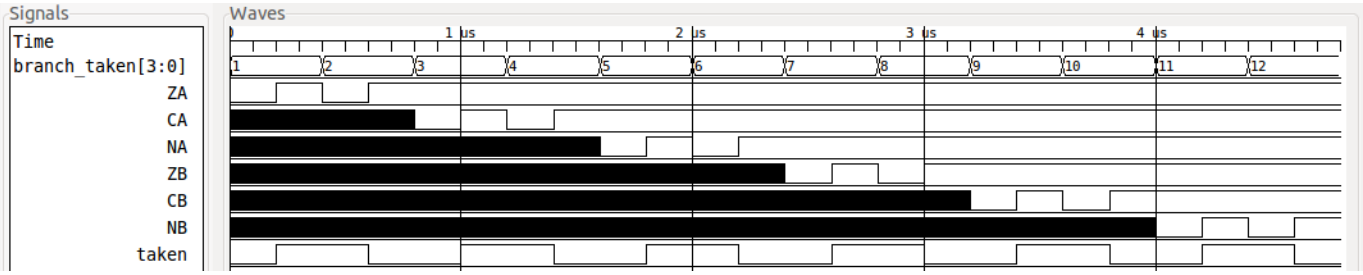
En la tabla 6 se muestran los valores asociados a cada uno de los saltos relativos (branches) definidos, así como la condición que permite que estos sea tomados. Basado en esta tabla, se escribió un banco de pruebas con el objetivo de probar el correcto funcionamiento del módulo, los resultados obtenidos de la ejecución de este banco de pruebas se muestran en la figura 7.

En la figura 7, se puede notar que los saltos relativos definidos por la señal *branch_ taken[3:0]*, se están tomando, es decir la señal *taken* está en 1, cuando se cumple la condición necesaria dada por las señales *ZA*, *CA*, *NA*, *ZB*, *CB* y *NB*, con lo que se puede concluir que el módulo tiene el funcionamiento esperado.

Tabla 6: Saltos relativos y condiciones para ser tomados

Branch	**Valor**	**Condición para ser tomado**
BAEQ	1	$Z_A = 1$
BANE	2	$Z_A = 0$
BACS	3	$C_A = 1$
BACC	4	$C_A = 0$
BAMI	5	$N_A = 1$
BAPL	6	$N_A = 0$
BBEQ	7	$Z_B = 1$
BBNE	8	$Z_B = 0$
BBCS	9	$C_B = 1$
BBCC	10	$C_B = 0$
BBMI	11	$N_B = 1$
BBPL	12	$N_B = 0$

Figura 7: Simulación del módulo *branch_ taken*



6.5. Banco de pruebas del módulo *branch_ calc*

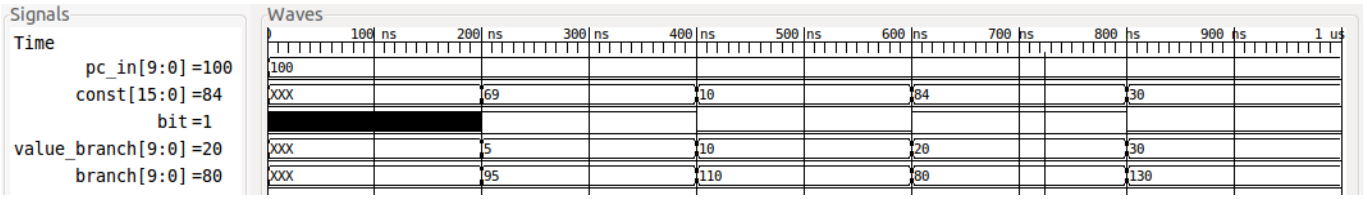
El módulo *branch_ calc* se encarga de calcular el nuevo valor del contador del programa si el salto relativo debe ser tomado. Se espera que el módulo sume o reste, según el valor de sétimo bit de la instrucción 0 u 1 respectivamente, al valor del contador de programa el valor indicado en la instrucción.

Para probar el adecuado funcionamiento de este módulo se escribió un banco de pruebas en el que se probaron diferentes valores de la constante que acompaña a la instrucción, buscando probar casos con el sétimo bit de la instrucción en 1 y en 0.

En la figura 8 se muestran los resultados tras la ejecución de dicho banco de pruebas, donde se puede ver que efectivamente a la señal *pc_ in* que se definió con un valor de 100 se le suma el valor del salto (señal *value_ branch*) cuando el sétimo bit (señal *bit*) está en 0, ambas tomadas del valor de la señal *const*, y se resta cuando el sétimo bit está en 1.

De forma que se puede concluir que el módulo cumple adecuadamente con sus especificaciones.

Figura 8: Simulación del módulo *branch_ calc*



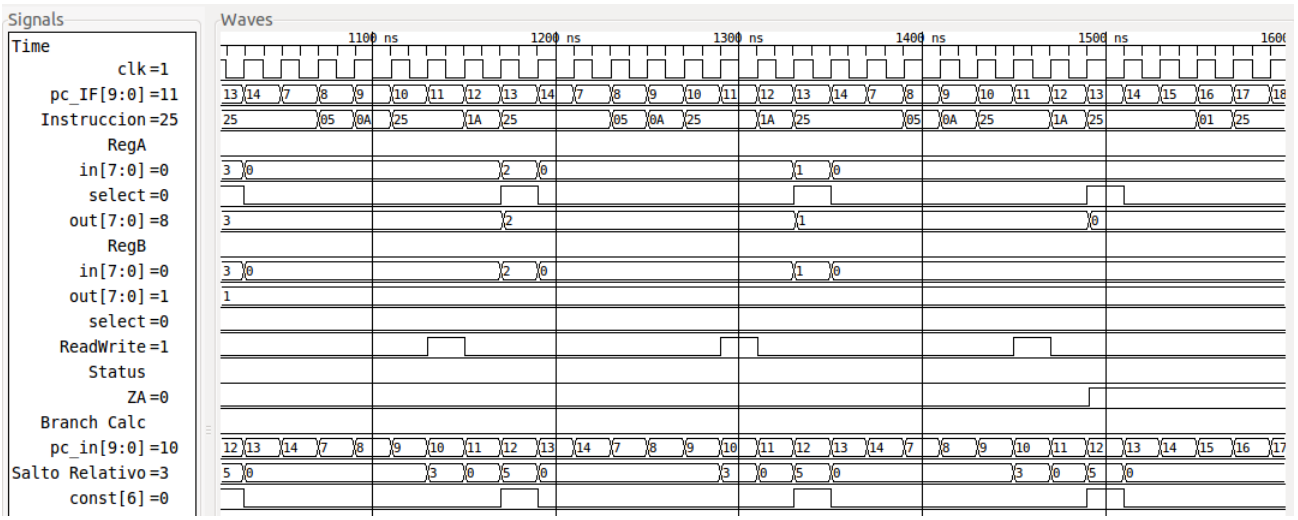
6.6. Banco de pruebas de los saltos relativos (branches)

Para comprobar que los saltos relativos (branches) se ejecutan de la forma correcta se escribió una secuencia de instrucciones, en la cual se creó un lazo utilizando el branch $BANE$ $(1A)$, el cual se ejecuta si se cumple que $Z_A = 0$, es decir si el registro A tiene un valor distinto a cero.

En la figura 9 se muestran los resultados obtenidos al ejecutar dicha secuencia, se puede ver que una vez que el branch ingresa, pasan dos ciclos de reloj para que la instrucción llegue a MEM y posteriormente es ejecutado según los valores especificados en las instrucciones. En este caso, el valor del salto es de 5, de forma que el program counter se devuelve a la instrucción STB (05) en la secuencia de instrucciones.

Con esto se concluye que efectivamente los saltos relativos tienen el comportamiento esperado.

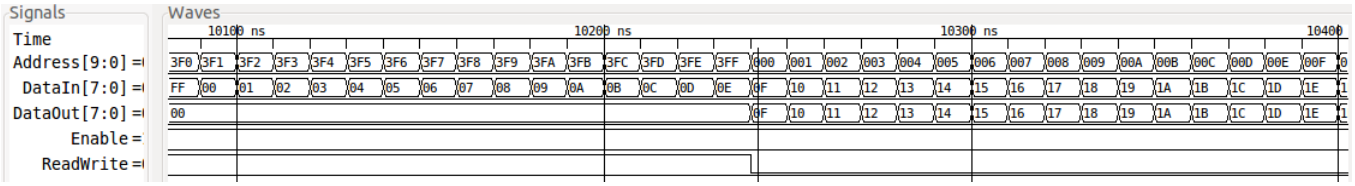
Figura 9: Evaluación de los saltos relativos



6.7. Banco de pruebas de las memorias

Se realizó una evaluación de la memoria RAM tal com se puede observar en la Figura 10 en donde se realizan 1024 iteraciones de escritura a la memoria para que tenga valores almacenados, y luego se realizan 1024 lecturas a todas las posiciones de memoria. En la Figura 10 se muestra donde hace el cambio de escritura a lectura para mostrar el comportamiento de las dos posibles acciones que se pueden llevar a cabo en este módulo.

Figura 10: Evaluación de la memoria RAM



6.8. Banco de pruebas de la unidad de forwarding

La unidad de *forwarding o bypassing* es un módulo bastante separado del resto del pipeline y básicamente se agrega como una mejora al diseño ya funcional con el fin de mejorar el desempeño de ciertos programas que ejecutan instrucciones de manera tal que producen alertas o *hazards* en el funcionamiento.

Se realizó una prueba a este módulo en donde se ejecute una secuencia de instrucciones para tratar de comprobrar que se afecta un registro una o más veces consecutivas. En términos generales se puede observar en la Figura 11 que primero viene una instrucción que afecta el registro A, luego una instrucción que no lo afecta. Luego se tienen dos instrucciones seguidas que afectan al registro A y finalmente tres instrucciones seguidas que afectan el registro A.

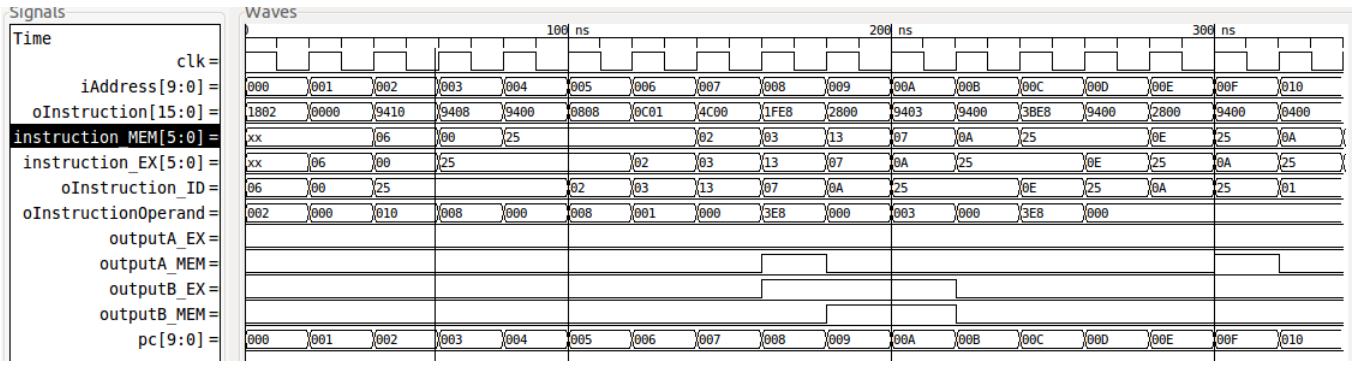
La secuencia de instrucciones implementada es tal que en principio debe comprobrar todos lo posibles casos, tal como se describe el comienzo de las instrucciones a continuación:

- ADDA: Se tiene una instruccíon de suma sin dependencias por lo cual no levanta ninguna de las señales de control.
- LDA: No hay dependencias por lo tanto no levanta ninguna señal de control.
- NOP: No hace nada. No levanta ninguna señal de control. Aparece 3 veces seguidas.
- LDCA: No tiene dependencia con ninguna instrucción. No levanta ninguna señal de control.
- LDCB: No tiene dependencia con ninguna instrucción. No levanta ninguna señal de control.
- ORB: Tiene una dependencia directa con la instrucción anterior, por lo tanto debe activar la señal de control outputB EX, porque el operando se encuentra actualizado en el registro EX MEM. Además con la instrucción transanterior por lo que debe activar la señal outputA MEM, ya que el operando actual se encuentra actualizado en MEM WB
- ADDB: Es una dependencia directa con la instruccíon anterior, por lo que debe activar la señal de control outputB MEM y la transanterior con outputB EX
- SUBA: Tiene una dependencia directa con la instrucción anterior y la transanterior por eso se levantan las dos señales de control correspondientes.
- NOP dos veces. No hace nada por lo tanto no activa ninguna señal de control.
- ANDA. No posee dependencia pues la NOP anterior se ejecuta dos veces y elimina el problema de la posible dependencia.
- NOP. No hace nada por lo tanto no activa ninguna señal de control.

- SUBA. Esta instrucción posee dependencia con la instrucción del ciclo transanterior, pero solo para uno de los registros, por eso solo debe activar outputA MEM.
- NOP. No hace nada por lo tanto no activa ninguna señal de control.
- LDB. No posee dependencia pues hace lectura de memoria principal, por lo tanto no activa ninguna señal de control.

Para mayor entendimiento se puede observar la definición de las instrucciones en el apéndice A

Figura 11: Evaluación del módulo Forwarding Unit

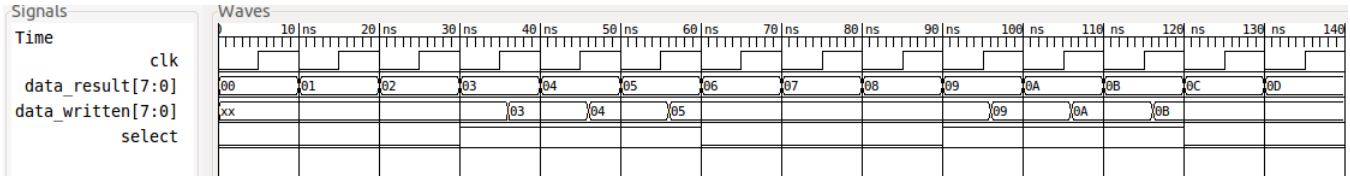


6.9. Banco de pruebas de los registros

Se realizaron pruebas al módulo de los registro para comprobrar que efectivamente realizan lo esperado. Aunque en términos generales es un módulo bastante pequeño en comparación con el resto de los módulos, se definió dentro de la estrategia de pruebas para evitar errores innecesarios a la hora de compilar, y pues el resultado obtenido se puede observar en la Figura 12 en donde claramente se puede percibir que los registros trabajan tal y como se esperaba.

Las señales *data result* y *data written* representan la entrada y la salida respectivamente mientras que la señal *select* es la señal de control que dice cuando hay que escribir en el registro y cuando no. Entonces cuando la señal de escritura *select* está en alto, el valor se almacena en el registro tal como se puede observar en el diagrama temporal.

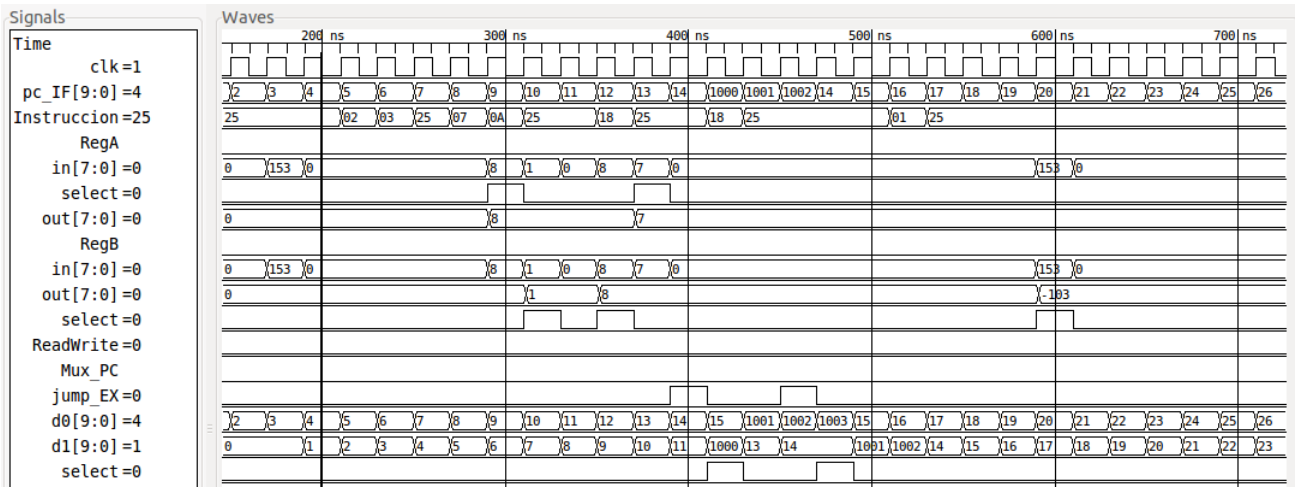
Figura 12: Evaluación del módulo de los registros



6.10. Banco de pruebas finales para el CPU

Se realizó una simulación de un conjunto de instrucciones variadas para el procesador, tratando de efectuar varios saltos incondicionales, tal como se puede observar en la Figura 13.

Figura 13: Evaluación del módulo CPU



En la figura 13 se puede apreciar cómo en la posición número 12 de la memoria se lee que se va a realizar un Jump (Instrucción 6'h18), y tres ciclos de reloj después el contador de programa realiza un salto a la posición de memoria 1000. Esto muestra el correcto funcionamiento de los saltos de memoria absolutos (Jump). Adicionalmente en esta prueba, en la posición 1000 se programó también un Jump que regresara a las primeras posiciones de la memoria y continuar con la ejecución de las instrucciones en forma secuencial.

6.10.0.1. Instrucción SUBB Para la implementación de la instrucción **SUBB** se debe aclarar que en el procesador desarrollado la operación que ejecuta es $B = A$ - B. Esto se realizó así dado que se encontró un error en la implementaci on pues se diseño el caso para que fuera así y no $B = B$ - A. Una posible solución futura para realizar la instrucción tal y como se solicita es agregar una nueva salida al módulo ALU que tenga el mismo resultado de la salida actual pero complementado, y luego agregar una señal de control al esquema que sea capaz de conocer si la instrucción es *SUBB*, debe tomar la segunda salida de la ALU, de lo contrario debe procesar la instrucción con normalidad.

7. Conclusiones

- Una de las pricipales conclusiones que se obtienen al realizar el presente proyecto es comprender la importancia del uso de segmentación en los procesadores actuales en tanto que se puede obtener un desempeño mucho mayor por el simple hecho de utilizar cada uno de los segmentos del procesador para tareas independientes y que de esta manera no se desperdicie tiempo de procesamiento, como si ocurre en los procesadores de ciclo simple.

- Además de comprender el funcionamiento de un procesador segmentado mediante su implementación y diseño en verilog, también se logró entender y comprender el efecto de los diferentes *hazards* que se presentan aquí y complican la labor de su desarrollo. De esta manera, el uso de unidades de *forwarding* para procesadores segmentados es un deber que debería de tener siempre y cuando se busqué mejorar el desempeño del mismo.

- Al igual que los puntos anteriores, el uso de más de un registro en el procesador hace que el funcionamiento sea considerablemente más complicado pues es necesario tomar en cuenta dependencia de instrucciones que normalmente no producen *hazards*, tanto por el operando 1 como el operando 2.

- Para lograr implementar una unidad lógico arimética, es necesario contar con un controlador de la misma, por lo cual una conclusión que se obtuvo al respecto es que debe de existir un diagrama arquitectónico y un conjunto de instrucciones definidas mucho antes de realizar la implementación, y si se desean agregar nuevas instrucciones puede resultar en un cambio considerable del código de programación.

- Las instrucciones de ensamblador propuestas para el procesador son limitantes para diferentes aplicaciones y usos de las mismas, principalmente por no tener direccionamiento indirecto, interrupciones ni pila.

- Por medio del uso de banderas se puede efectivamente determinar cuando un branch debe ser tomado.

- Con respecto a la distribución de trabajo en grupo de logró determinar que la distribución de las tareas debe ir de la mano con la modulación del esquema principal, lo cual simplifica mucho la ejecución del trabajo tanto aquí como en cualquier otro proyecto pues permite a cada uno de los integrantes trabajar de manera independiente y tener un molde al cual apegarse para que el tiempo utilizado en la etapa de desarrollo sea lo menor posible.

El formato recomendado para la bibliografía es el APA. El siguiente es un ejemplo:

Referencias

[Morris y Ciletti, 2013] M. Morris Mano y Michael D. Ciletti (2013). *Diseño Digital*. México: Pearson Always Learning, 5th Edition.

A. Anexos

Se presenta un resumen del código de las instrucciones en hexadecimal para su rápida referencia.

```
`timescale 1ns / 100ps
`ifndef DEFINTIONS_V
`define DEFINTIONS_V

`default_nettype none
`define LDA   6'h00
`define LDB   6'h01
`define LDCA  6'h02
`define LDCB  6'h03
`define STA   6'h04
`define STB   6'h05

`define ADDA  6'h06
`define ADDB  6'h07
`define ADDCA 6'h08
`define ADDCB 6'h09
`define SUBA  6'h0A
`define SUBB  6'h0B

`define SUBCA 6'h0C
`define SUBCB 6'h0D
`define ANDA  6'h0E
`define ANDB  6'h0F
`define ANDCA 6'h10
`define ANDCB 6'h11

`define ORA   6'h12
`define ORB   6'h13
`define ORCA  6'h14
`define ORCB  6'h15
`define ASLA  6'h16
`define ASRA  6'h17

`define JMP   6'h18
`define BAEQ  6'h19
`define BANE  6'h1A
`define BACS  6'h1B
`define BACC  6'h1C
`define BAMI  6'h1D

`define BAPL  6'h1E
`define BBEQ  6'h1F
`define BBNE  6'h20
```

```
`define BBCS  6'h21
`define BBCC  6'h22
`define BBMI  6'h23

`define BBPL  6'h24
`define NOP   6'h25

`endif
```