

Laboratorio ARMv8 en SystemVerilog

Objetivos

- Desarrollar códigos en lenguaje SystemVerilog para describir circuitos secuenciales y combinacionales vistos en el teórico y el práctico.
- Utilizar la herramienta Quartus para analizar y sintetizar el código SystemVerilog.
- Aprender a reutilizar código SystemVerilog mediante módulos estructurales.
- Mediante el uso de test bench, analizar las formas de onda y testear los resultados.
- Aplicar los conceptos aprendidos sobre microprocesadores y la técnica de mejora de rendimiento: segmentación de cauce (*pipeline*).

Condiciones

- Realizar el trabajo práctico en los grupos de **3 personas** ya creados.
- Crear un tag en el repositorio que se llame EntregaLab1. La fecha límite de entrega es el **martes 12 de noviembre** (inclusive). No se considerarán modificaciones después de esa fecha.

Formato de entrega

- Utilizar los nombres de los módulos y señales indicados en las guías del práctico.
- Al repositorio se debe subir el proyecto de Quartus completo y el readme que funcionará de informe (**en formato md**). Éste último debe incluir el nombre de los integrantes del laboratorio, una aclaración de qué ejercicios resolvieron y los requerimientos descriptos al final de cada ejercicio.
- El trabajo es incremental, se pide que se entregue un único procesador con todas las modificaciones correspondientes a los ejercicios que se decidieron resolver.
- No está permitido compartir código entre grupos.
- No está permitido subir el código en repositorios públicos.

Calificación

El ejercicio 1 es obligatorio. Su resolución debe estar aprobada para obtener la regularidad de la materia. Quienes resuelvan el ejercicio 2 (partiendo del procesador modificado en el ejercicio 1) estarán habilitados para promocionar, si cumplen con los demás requisitos. Quienes resuelvan los 3 ejercicios obtendrán 1 punto extra para el primer parcial.

***Importante:** verificar que no se produzcan advertencias de la herramienta durante el proceso de síntesis relacionadas con una mala interpretación del circuito a implementar.

DESARROLLO

El laboratorio está basado en la implementación del microprocesador ARMv8 en versión reducida obtenida en el trabajo práctico 2, antes de comenzar el diseño crear un tag que se

llame "uP-SingleCycle". Luego, descargar de Moodle el set de archivos <PipelinedProcessorPatterson-Modules> con la descripción de algunos módulos del procesador con pipeline y reemplazar los archivos con el mismo nombre del repositorio. Finalmente verificar las conexiones resultantes según los diagramas de las figuras 1 y 2.

Recomendaciones importantes:

- Recordar inicializar los registros X0 a X30 con los valores 0 a 30 respectivamente en la implementación del bloque *#regfile*.
- Se debe modificar el bloque *#decode* a fin de agregar el puerto de entrada resaltado con un círculo rojo en la Fig. 2.
- Se debe modificar el bloque *#regfile* a fin de que si alguno, o ambos registros leídos por una instrucción en la etapa *#decode*, están siendo escritos como resultado de una instrucción anterior en la etapa *#writeback* se obtenga a la salida de *#regfile* el valor actualizado del registro.

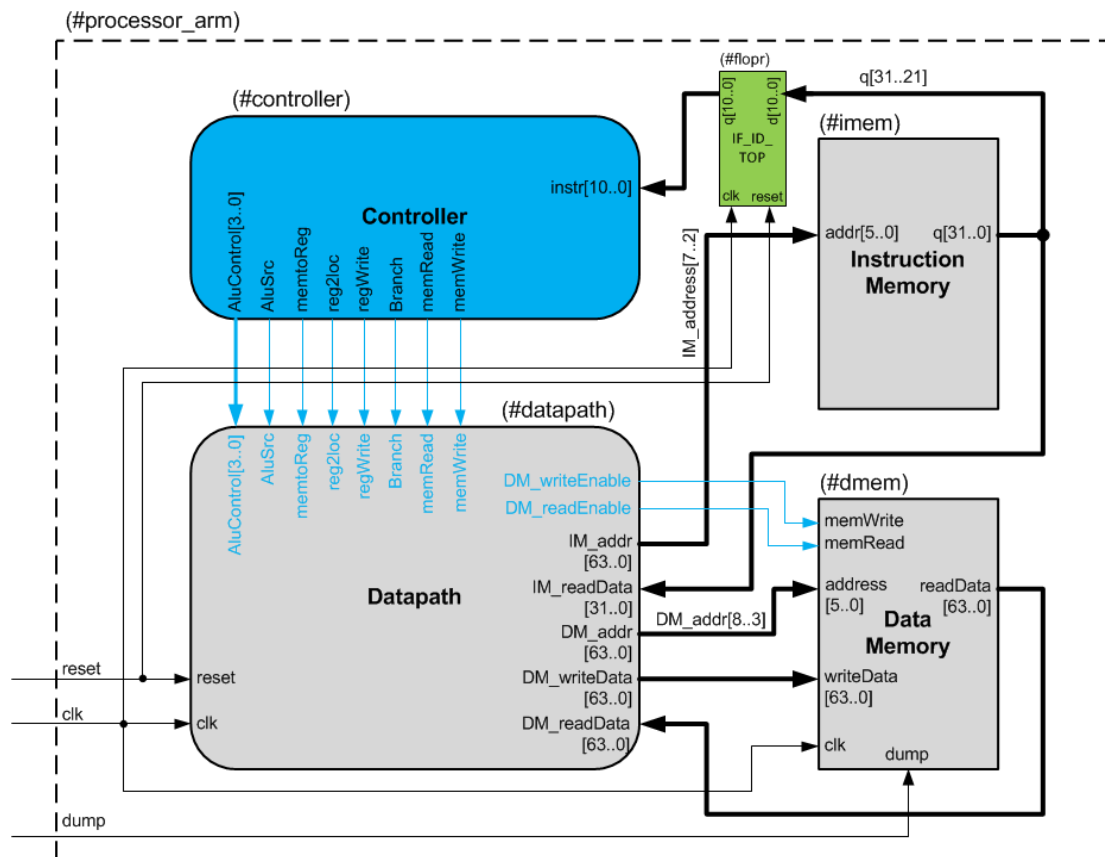


Figura 1: Esquema del *top_level_entity*

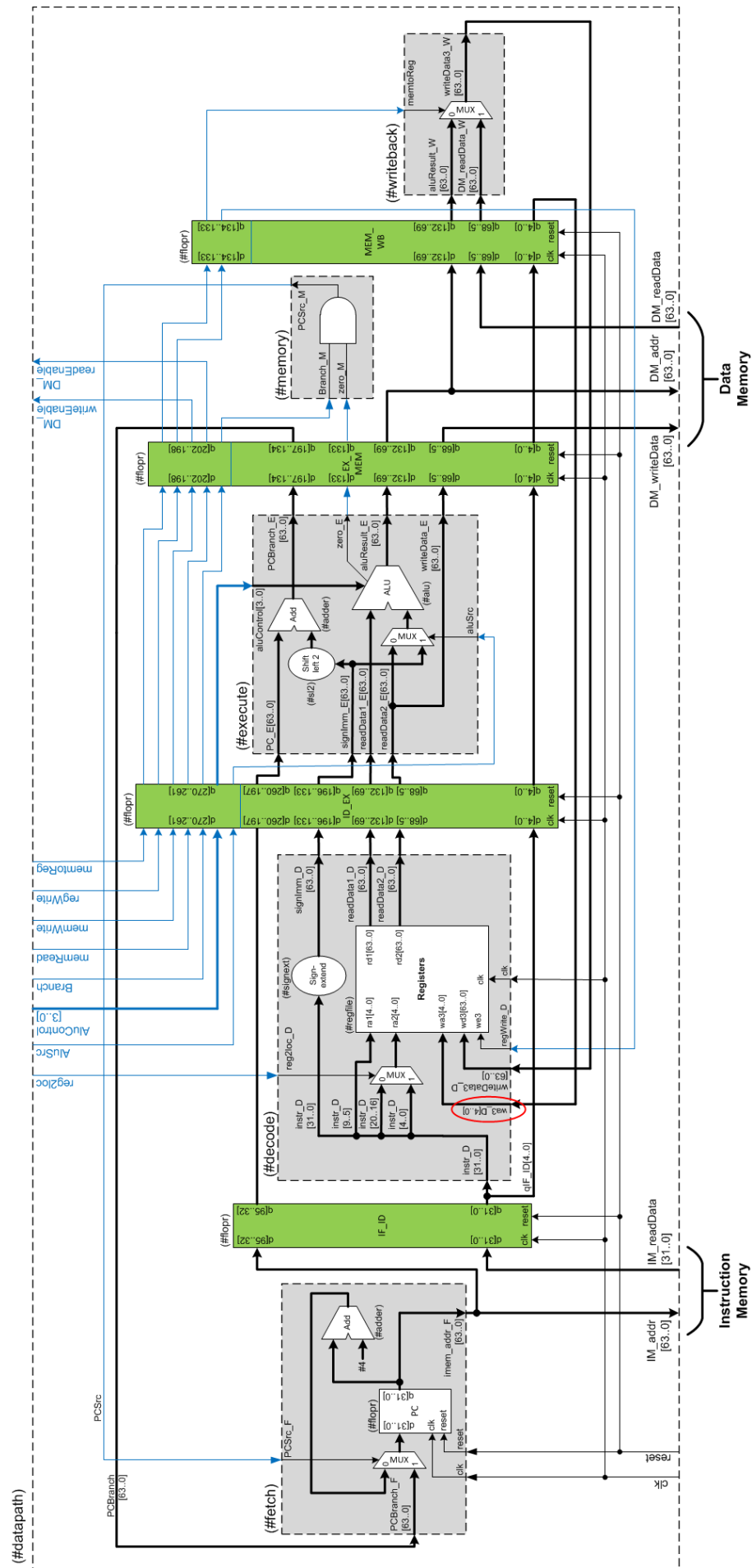


Figura 2: Esquema del datapath

Una vez concluida la implementación del procesador completo con pipeline, se debe verificar su correcto funcionamiento utilizando el siguiente código:

```

// Dirección:valor
STUR X1, [X0, #0] // MEM 0:0x1
STUR X2, [X0, #8] // MEM 1:0x2
STUR X3, [X16, #0] // MEM 2:0x3
ADD X3, X4, X5
STUR X3, [X0, #24] // MEM 3:0x9
SUB X3, X4, X5
STUR X3, [X0, #32] // MEM 4:0xFFFFFFFFFFFFFFFF
SUB X4, XZR, X10
STUR X4, [X0, #40] // MEM 5:0xFFFFFFFFFFFFFFFF6
ADD X4, X3, X4
STUR X4, [X0, #48] // MEM 6:0xFFFFFFFFFFFFFFFF5
SUB X5, X1, X3
STUR X5, [X0, #56] // MEM 7:0x2
AND X5, X10, XZR
STUR X5, [X0, #64] // MEM 8:0x0
AND X5, X10, X3
STUR X5, [X0, #72] // MEM 9:0xA
AND X20, X20, X20
STUR X20, [X0, #80] // MEM 10:0x14
ORR X6, X11, XZR
STUR X6, [X0, #88] // MEM 11:0xB
ORR X6, X11, X3
STUR X6, [X0, #96] // MEM 12:0xFFFFFFFFFFFFFFFF
LDUR X12, [X0, #0]
ADD X7, X12, XZR
STUR X7, [X0, #104] // MEM 13:0x1
STUR X12, [X0, #112] // MEM 14:0x1
ADD XZR, X13, X14
STUR XZR, [X0, #120] // MEM 15:0x0
CBZ X0, L1
STUR X21, [X0, #128] // MEM 16:0x0 (si falla CBZ =21)
L1: STUR X21, [X0, #136] // MEM 17:0x15
ADD X2, XZR, X1
L2: SUB X2, X2, X1
ADD X24, XZR, X1
STUR X24, [X0, #144] // MEM 18:0x1 y MEM 19:0x1
ADD X0, X0, X8
CBZ X2, L2
STUR X30, [X0, #144] // MEM 20:0x1E
ADD X30, X30, X30
SUB X21, XZR, X21
ADD X30, X30, X20
LDUR X25, [X30, #-8]
ADD X30, X30, X30
ADD X30, X30, X16
STUR X25, [X30, #-8] // MEM 21:0xA (= MEM 9)
finloop: CBZ XZR, finloop

```

TENER EN CONSIDERACIÓN LOS EVENTUALES PROBLEMAS DE HAZARD (de datos y de control) QUE CONTIENE EL PROGRAMA UTILIZADO Y PLANTEAR LAS MODIFICACIONES DE CÓDIGO NECESARIAS PARA EVITAR SU OCURRENCIA. Para esto, agregar instrucciones tipo “nop”, las cuales se pueden implementar como ADD XZR, XZR, XZR.

Ejercicio 1 (obligatorio)

Sin afectar el funcionamiento de las instrucciones ya implementadas en la versión reducida del microprocesador con pipeline, **agregar** las instrucciones **ADDI** y **SUBI**. Introducir en el procesador todas las modificaciones necesarias, tanto en el datapath como en las señales de control.

Una vez finalizadas las modificaciones, agregar al código dado (en la página anterior) instrucciones **ADDI** y **SUBI** con distintos valores numéricos y verificar su correcta implementación. Es decir, se debe analizar que todo el set de instrucciones continúe funcionando correctamente y también las instrucciones tipo I agregadas (para esto pueden escribir los resultados en memoria y verificar si obtienen los valores correctos en “mem.dump” al finalizar la ejecución del código). No olvidar resolver los eventuales problemas de hazard agregando instrucciones “nop”.

Para el readme:

- Describir brevemente qué modificaciones se introdujeron (en qué entidades y con qué finalidad). Mostrar el diagrama del nuevo microprocesador, indicando las señales y entidades agregadas (de ser necesario).
- Mostrar el programa en assembler LEGv8 modificado que se utilizó para verificar el comportamiento del procesador.

Ejercicio 2 (para promocionar)

Sin afectar el funcionamiento de las instrucciones ya implementadas en la versión reducida del **microprocesador con pipeline** del ejercicio 1, agregar las instrucciones de saltos condicionales **B.cond** y las instrucciones aritméticas que configuran las banderas del microprocesador: **ADDS**, **SUBS**, **ADDIS** y **SUBIS**. Estas últimas instrucciones son necesarias para que los saltos condicionales (**B.cond**) puedan determinar si deben saltar o no.

Para que este ejercicio sea considerado correctamente implementado, el microprocesador debe cumplir los siguientes requerimientos:

- La ALU debe generar las 4 banderas existentes en el microprocesador estudiado: Zero, Negative, Carry y oVerflow. Además debe generar la señal *write_flags* que indica si se deben actualizar las banderas o no.
- En las nuevas instrucciones **ADDS**, **SUBS**, **ADDIS** y **SUBIS** la señal *alucontrol* debe ser similar al de las **ADD**, **SUB**, **ADDI** y **SUBI** (respectivamente) pero con un ‘1’ en el bit más significativo.

- Debe existir un registro de flags llamado *CPSR_flags*, de cuatro bits, que almacene el estado de las banderas (Z, N, C, V). Este registro debe escribirse únicamente cuando se ejecuta una instrucción que setee flags (*ADDS*, *SUBS*, *ADDIS* o *SUBIS*).
- Debe crearse una nueva señal de control llamada *condBranch* que indica si se debe realizar un salto condicional.
- Se deben agregar las 14 condiciones de saltos existentes en el set de instrucciones LEGv8. El salto condicional es una instrucción tipo CB. En los cinco bits menos significativos se indica qué condición se debe cumplir para que se tome el salto de la forma en que se muestra en la siguiente tabla:

Instruction	Rt [4:0]	Instruction	Rt [4:0]
B.EQ	00000	B.VC	00111
B.NE	00001	B.HI	01000
B.HS	00010	B.LS	01001
B.LO	00011	B.GE	01010
B.MI	00100	B.LT	01011
B.PL	00101	B.GT	01100
B.VS	00110	B.LE	01101

- Todas las demás instrucciones, incluida la de salto CBZ, deben continuar funcionando normalmente.

Recomendaciones:

- Crear un nuevo bloque llamado *bCondCheck* en la etapa de memory que verifique la condición de salto.
- En caso de que en la ALU hayan implementado la resta utilizando el operador de la resta de system verilog ($a - b$), se recomienda modificarlo a la siguiente forma: complementar el operando b y sumar: $a + \text{complemento}_2(b)$. Esto se debe a que la bandera de Carry se genera de forma distinta en ambos casos y es necesario que la bandera de carry sea generada de esta forma en particular para que puedan utilizarse las condiciones de salto impuestas en las siguientes tablas:

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	$\sim(Z=0 \ \& \ N=V)$	B.LS	$\sim(Z=0 \ \& \ C=1)$
>	B.GT	$(Z=0 \ \& \ N=V)$	B.HI	$(Z=0 \ \& \ C=1)$
≥	B.GE	N=V	B.HS	C=1

Signed and Unsigned numbers	
Instruction	CC Test
Branch on minus (B.MI)	N= 1
Branch on plus (B.PL)	N= 0
Branch on overflow set (B.VS)	V= 1
Branch on overflow clear (B.VC)	V= 0

Importante!! Utilizar una señal interna para calcular el complemento a 2 de “b” y luego utilizar esa señal para operar con “a”.

Para el readme:

- Describir brevemente qué modificaciones se introdujeron (en qué módulos y con qué finalidad). Mostrar el diagrama del nuevo microprocesador, indicando las señales y módulos agregados.
- Escribir una sección de código assembler, donde se pruebe el funcionamiento de las nuevas instrucciones y se verifique que las que ya estaban implementadas continúen funcionando correctamente. Los resultados obtenidos al ejecutar cada una de las instrucciones deben guardarse en la memoria de datos. Mostrar el programa en assembler LEGv8 modificado que se utilizó para verificar el comportamiento del procesador y el contenido de memoria al finalizar su ejecución.

Ejercicio 3 (+1 punto)

El procesador LEGv8 desarrollado no tiene la capacidad de detectar la ocurrencia de *hazards* de ningún tipo. En este ejercicio se propone la implementación de un bloque de detección de *hazards* (*Hazard Detection Unit*) y otro de *forwarding* (*Forwarding Unit*), a fin de aplicar la técnica de *forwarding-stall* en caso de la ocurrencia de un **data hazard**, hasta que el mismo desaparezca.

Algunas aclaraciones respecto a la implementación:

- La *HDU* debe implementarse en la instancia del *Instruction decode* (ID).
- Las diversas condiciones para la detección de un *data hazard* se analizan en el capítulo 4.7- “Data Hazards: Forwarding vs Stalling” del libro “Computer Organization and Design - ARM Edition” de D.Patterson y J. Hennessy. Se deben considerar TODAS las condiciones para todos los tipos de dependencias de datos.
- Para generar la condición de *stall* en el procesador es necesario:
 - Evitar que el PC avance a la siguiente instrucción en el siguiente CLK y evitar que el registro de pipeline IF/ID cambie de valor en el siguiente CLK (congelar su valor). Para esto deberán diseñar una nueva entidad **FLOPRE** similar al **FLOPR**, pero agregando una señal de *enable* (habilitación). Funcionamiento: *enable* = 1 el funcionamiento es normal, *enable* = 0 no altera el valor de salida al detectar un flanco de CLK (síncrono).
 - Forzar que todas las señales de control a partir del ciclo EX en adelante tomen el valor “0” (Ver implementación de referencia en Fig 4.59).
- ¡No olvidar que una parte del registro IF/ID está en la entidad **#processor_arm!**

Para el readme:

- Mostrar un diagrama de bloques general del nuevo procesador, indicando las señales y módulos agregados.
- Correr nuevamente el código desarrollado para el ejercicio 1, sin el agregado de instrucciones “nop”, y tomar una captura de la pantalla “Wave” de ModelSim donde se vea un caso de *stall* de instrucciones y otra donde se vea el *forwarding* de datos entre instrucciones. Explicar brevemente qué se observa en la imagen.