Arquitectura de Computadoras 2019

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 obtenidos.
- Implementar una versión reducida de un microprocesador con arquitectura ARMv8 en una FPGA.

Condiciones

- Realizar el trabajo práctico en grupos de **máximo 3 personas**.
- Enviar el trabajo resuelto a delfinavelez@gmail.com. La fecha límite de entrega es el **miércoles 13 de Noviembre** (inclusive). Los trabajos entregados después de esa fecha se consideran desaprobados.
- Defender en un coloquio el trabajo presentado. Deben presentarse todos los integrantes del grupo y responder preguntas acerca del trabajo realizado. El coloquio es el **viernes 15 de Noviembre**, en el horario del práctico. En caso de no poder asistir al coloquio en esa fecha, coordinar **previamente** con los docentes.

Formato de entrega

- Deben entregar un archivo comprimido (tar, zip, rar, etc.), con el nombre: “ApellidoNombre1_ApellidoNombre2_ApellidoNombre3”, respetando mayúsculas y minúsculas.
- Se deben utilizar los nombres de los módulos y señales indicados en las guías de práctico.
- Junto con el código, deben entregar un pequeño informe (**en formato pdf**) con la resolución de los ejercicios.
- No está permitido compartir código entre grupos.

Calificación

Los ejercicios 1 y 2 son obligatorios. Su resolución y presentación en coloquio deben estar aprobados para obtener la regularidad de la materia. Quienes además resuelvan el ejercicio 3 obtendrán 1 punto extra para el primer parcial y si resuelven los 4 ejercicios obtendrán 2 puntos 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 (ejemplo: inferencia de *latches* en circuitos combinacionales).

DESARROLLO

Ejercicio 1 (obligatorio)

El laboratorio está basado en la implementación de un microprocesador ARMv8 con *Pipeline*, en versión reducida. Para comenzar el diseño, descargar de Moodle el set de archivos <PipelinedProcessorPatterson-Modules> (.sv) con la descripción de los módulos del procesador con pipeline. A continuación crear un proyecto cuya *Top-level Entity* sea <processor_arm> y agregar los archivos .sv del paquete y todos los archivos de descripción desarrollados en el la guía de práctico n°1. 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.

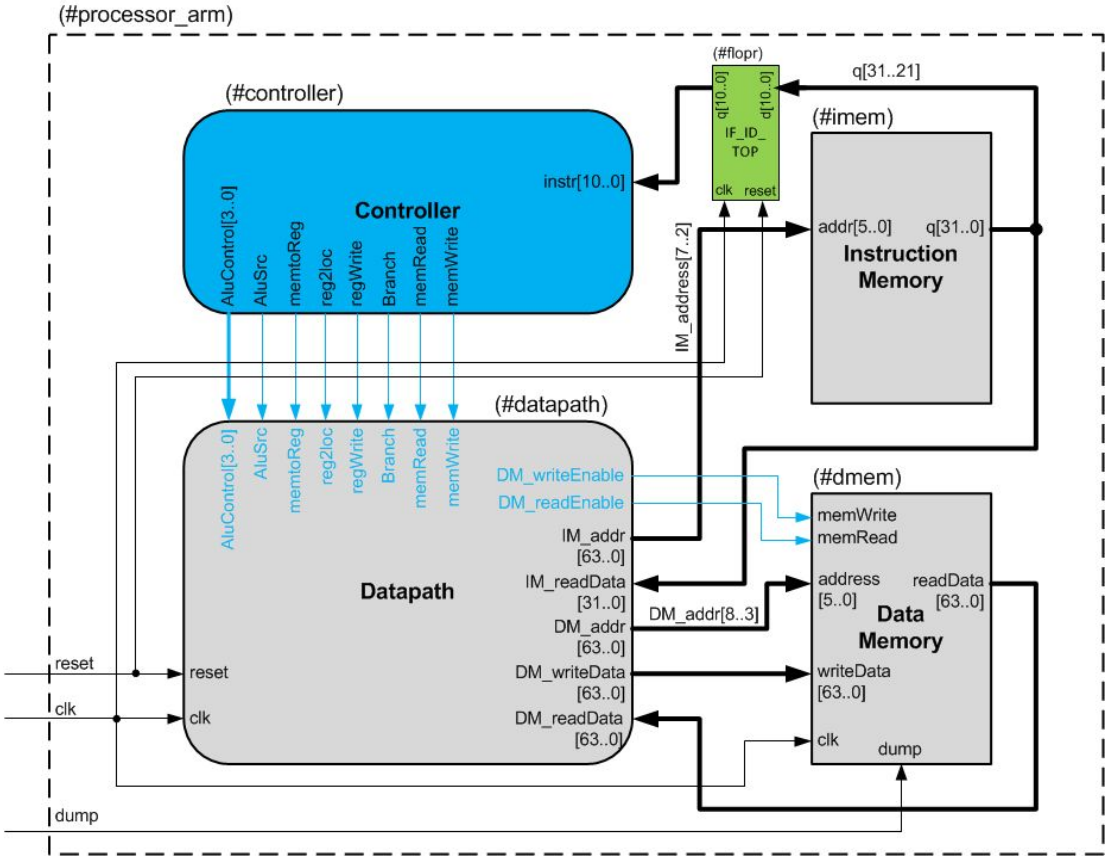

Figura 1: Esquema del *top_level_entity*

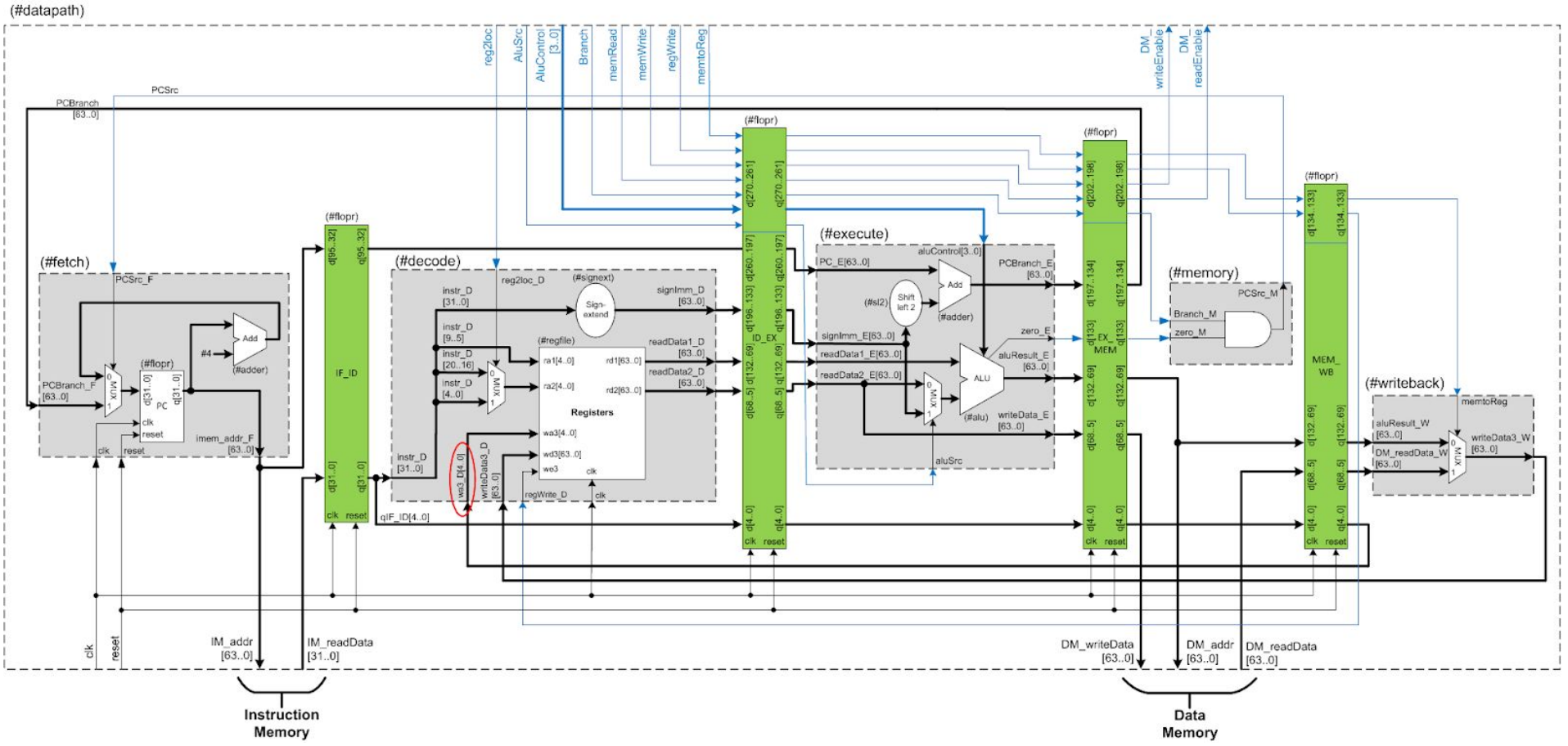

Figura 2: Esquema del *datapath*

Una vez concluida la implementación del procesador completo con pipeline, se debe verificar su correcto funcionamiento utilizando el mismo procedimiento de prueba que en el ejercicio 1 de la guía del práctico n°2.

TENER EN CONSIDERACIÓN LOS EVENTUALES PROBLEMAS DE HAZARD (de datos y de control) QUE CONTIENE EL PROGRAMA UTILIZADO Y PLANTEAR LAS MODIFICACIONES NECESARIAS PARA EVITAR SU OCURRENCIA. Para esto, se puede modificar el orden de ejecución del programa y/o agregar instrucciones “nop”, las cuales se pueden implementar mediante el agregado del OpCode 0x00000000 en lugar de un OpCode de una instrucción válida.

Para el informe:

- Mostrar el programa modificado que se utilizó para verificar el comportamiento del procesador.
- Analizar la cantidad de ciclos de CLK que se deben generar en el Test bench para la ejecución completa y correcta del código modificado y compararla con la cantidad de ciclos de CLK requeridos por el procesador de un ciclo para completar la ejecución del código original.

Ejercicio 2 (obligatorio)

Cargar el siguiente programa en su microprocesador con *pipeline* y analizar su funcionamiento utilizando *Model-Sim*.

```
ADD X0, XZR, X4
ADD X1, X0,  X4
ADD X2, X1,  X4
ADD X3, X2,  X4
STUR X0, [XZR, #0]
STUR X1, [XZR, #8]
STUR X2, [XZR, #16]
STUR X3, [XZR, #24]
```

Para el informe:

- Verificar si funciona correctamente el programa. En caso contrario, fundamentar.
- Si el programa no funciona correctamente, re-escribir el código de la manera que se considere más conveniente para solucionarlo, sin modificar el resultado final. Demostrar su correcto funcionamiento utilizando *Model-Sim*.
- Mostrar el contenido de memoria al finalizar la ejecución del código modificado.

Ejercicio 3 (+1 punto extra)

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

Una vez finalizado, correr el siguiente programa para verificar su correcta implementación:

```
     ADDI X10, X0, #15
     ORR X11, X12, X20
     AND X12, X12, X20
loop:ADD X0, X0, X10
     SUB X10, X10, X1
     STUR X11, [XZR, #0]
     STUR X12, [XZR, #8]
     CBNZ X10, loop
     STUR X0, [XZR, #16]
```

Verificar que la memoria de salida contiene:

```
Memoria RAM de Arm:
    Address Data
 0 000000000000001c
 1 0000000000000004
 2 0000000000000078
 3 0000000000000000
          …
63 0000000000000000
```

Para el informe:

- 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).

Ejercicio 4 (+1 punto extra)

Como se observa en los ejercicios anteriores, el ARMv8 desarrollado no tiene la capacidad de detectar la ocurrencia de *hazards* de ningún tipo. Es sabido que el *hazard* de ocurrencia más frecuente es el de datos, por lo que se propone la implementación de un bloque de detección de *hazards* (*Hazard Detection Unit*) a fin de insertar stalls en forma automática hasta que el *data hazard* desaparezca. Vale aclarar que esto NO IMPLICA la implementación de la lógica y recursos necesarios para la técnica del *data forwarding*.

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:
    - ID/EX y EX/MEM
    - ID/EX y MEM/WB
- La *HDU* debe, al encontrar una condición de hazard verdadera, generar la condición de *stall* en el procesador a partir del ciclo ID, esto es:
    - Evitar que el PC avance a la siguiente instrucción en el siguiente CLK (congelar su valor).

- Evitar que el registro de pipeline IF/ID cambie de valor en el siguiente CLK. 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, pág. 327).

Para el informe:

Correr el siguiente programa, el cual incluye las dependencias antes mencionadas, en el microprocesador sin *HDU* y mostrar dónde se producen los hazard y qué valores se guardan en memoria. Luego, para verificar el correcto funcionamiento de la unidad de detección de hazard, correr nuevamente el mismo programa en el microprocesador con *HDU* y mostrar dónde se agregan los stalls, el aumento del número de ciclos de ejecución y el contenido final de la memoria.

```
      ADDI X9, XZR, #10
      SUB X10, X10, X9
      CBZ X10, loop1
      ADD XZR,XZR,XZR
      ADD XZR,XZR,XZR
      ADD XZR,XZR,XZR
      STUR X6, [X0, #0]
loop1:ADDI X10, XZR, #5
loop2:STUR X2, [X0,#8]
      LDUR X11, [X0, #8]
      ADD X2, X2, X1
      OR X4, X4, XZR
      STUR X2, [X0, #16]
      ADD X0, X0, #16
      SUB X10, X10, X1
      CBNZ X10, loop2
      ADD XZR,XZR,XZR
      ADD XZR,XZR,XZR
      ADD XZR,XZR,XZR
      STUR X3, [X0, #8]
```

Si la HDU funciona correctamente, la memoria de salida debe contener:

```
Memoria RAM de Arm:
    Address Data
0 0000000000000000
1 0000000000000002
2 0000000000000003
3 0000000000000003
4 0000000000000004
5 0000000000000004
6 0000000000000005
```

```
7 0000000000000005
8 0000000000000006
9 0000000000000006
10 0000000000000007
11 0000000000000003
12 0000000000000000
          …
63 0000000000000000
```