TP3: MIPS

- Luna, Lihué Leandro
- Bonino, Francisco Ignacio

Introducción

Este trabajo tiene como objetivo la implementación de un procesador con arquitectura MIPS en un lenguaje de descripción de hardware para FPGA. Este trabajo se realizó reciclando los trabajos prácticos anteriores (ALU + UART).

Pipeline

La idea de un pipeline es la de separar las partes de procesamiento de una instrucción en diferentes etapas, de tal manera que cuando una etapa de procesamiento de una instrucción se libera, automáticamente es ocupada por otra instrucción para ser procesada, y así aprovechar al máximo los ciclos de clock de la CPU. Cada una de estas etapas está conectada con la siguiente y la anterior mediante latches que almacenan los datos a transferir de una etapa a la otra, los cuales se sincronizan y habilitan mediante el clock del sistema. En la arquitectura MIPS tenemos un pipeline canónico de 5 etapas:

Instruction fetch (IF)

La instrucción a ejecutar se busca accediendo a la memoria de instrucciones usando el program counter (PC, de 8 bits) como puntero a la dirección de memoria donde está la misma.

Instruction decode (ID)

En esta etapa se decodifica la instrucción leída y se accede al register file (RF, que consta de 32 registros de 4 bytes cada uno) para buscar aquellos registros que sean utilizados en la instrucción.

Execute (EX)

En esta etapa se ejecuta la instrucción propiamente dicha. Los datos a utilizar como operandos y operador se definen mediante señales de control provenientes de la unidad de control.

Memory access (MEM)

En esta etapa se accede a memoria de datos para realizar operaciones de lectura/escritura según sea necesario en base a la instrucción ejecutada. Se accede a memoria si la instrucción es de tipo load/store. Para el resto de instrucciones, en esta etapa no pasa nada, pero deben respetar el paso por esta etapa para mantener la consistencia y sincronización del pipeline

Write-back (WB)

Los resultados finales se escriben en los registros del procesador para que estén disponibles para la siguiente instrucción. Por ejemplo, si se tiene una instrucción que suma dos valores y almacena el resultado en un registro, en la etapa de write back se escribirá el resultado final en el registro correspondiente. Si la instrucción es una carga de memoria, la etapa de write back escribirá el valor cargado desde la memoria en

el registro correspondiente. Una vez que se completa la etapa de write back, la instrucción actual se ha ejecutado por completo y se elimina del pipeline. Cuando una instrucción "sale" del pipeline (termina la etapa de write-back) se la considera como finalizada.

Instrucciones soportadas

En la arquitectura MIPS, las instrucciones en assembly son todas de 32 bits y se dividen en 3 tipos:

- Tipo R: Instrucciones aritmético-lógicas
- Tipo I: Instrucciones de salto condicional
- Tipo J: Instrucciones de salto incondicional

Las instrucciones soportadas en este trabajo son las siguientes:

- Tipo R:
    - ADDU
    - SUBU
    - AND
    - OR
    - XOR
    - NOR
    - SLT
    - SLL
    - SRL
    - SRA
    - SLLV
    - SRLV
    - SRAV
- Tipo I:
    - ADDI
    - ANDI
    - ORI
    - XORI
    - SLTI
    - LB
    - LH
    - LW
    - LBU
    - LHU
    - LWU
    - SB
    - SH
    - SW
    - LUI
    - BEQ
    - BNE
    - J
    - JAL

- Tipo J:
    - JALR
    - JR

Riesgos

A la hora de trabajar con un pipeline hay que tener en cuenta que pueden presentarse los tres tipos de riesgos explicados en el teórico:

- Riesgos estructurales: Dos instrucciones intentan utilizar un mismo recurso en el mismo ciclo de ejecución.
- Riesgos de datos: Una instrucción intenta utilizar un dato antes de que esté disponible (un resultado de una instrucción inmediatamente anterior que aún no llegó a la etapa de write-back, por ejemplo). Pueden darse tres tipos de riesgos de datos:
    - Escritura después de lectura (WAR)
    - Escritura después de escritura (WAW)
    - Lectura después de escritura (RAW)
- Riesgos de control: Se intenta tomar una decisión en base a una condición que aún no se ha evaluado. Estos riesgos se dan cuando la CPU no tiene el resultado de una branch a la hora de insertar una nueva instrucción en el pipeline en la etapa IF. En este trabajo se tomó la decisión de considerar que todas las instrucciones de tipo branch no son tomadas.

Unidad de detección de riesgos

Esta unidad debe detectar si se esta dando una instrucción de tipo load para generar un stall, es decir, introducir una operación vacía entre operaciones ("frenar" el pipeline en las etapas anteriores) para lograr resolver algunas dependencias de datos.

Unidad de cortocircuito

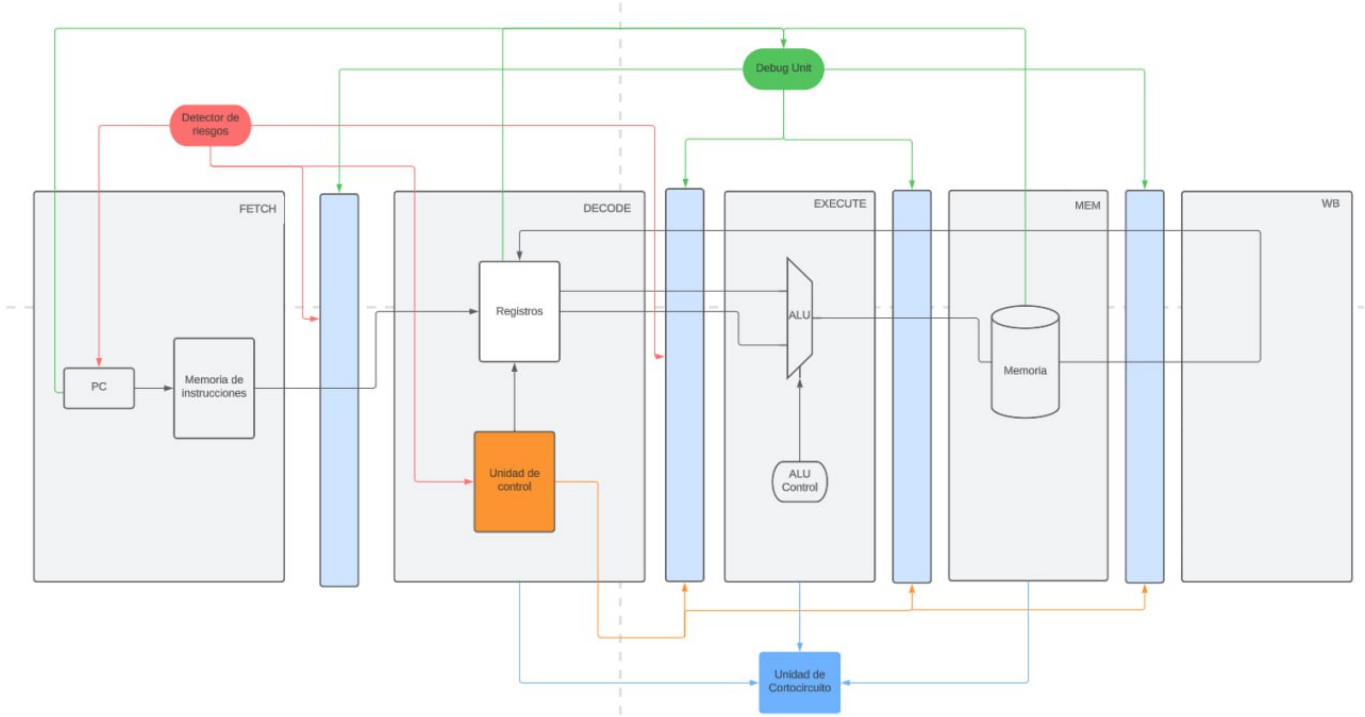
Esta unidad busca cortocircuitar los conectores latcheados de las etapas con las entradas de la ALU para disponer de los datos antes de que sean escritos en memoria y así poder solucionar los riesgos de datos.

Unidad de control

Esta unidad está implementada sobre la etapa de decodificación de instrucciones, y está encargada de generar señales de control que se envían de etapa en etapa para controlar qué se debe hacer. Por ejemplo, algunas señales involucran definir qué operación de la ALU debe llevarse a cabo, si hay que escribir o leer en memoria, o si hay que escribir en el RF en la etapa de WB.

Diagrama

El diagrama que muestra la esencia de cada etapa y sus conexiones mediante las unidades y módulos mencionados se muestra a continuación:



Unidad de debug

El propósito de implementar una unidad de debug es la de proveer una interfaz entre el usuario y el procesador mediante la UART desarrollada en el trabajo práctico anterior para implementar funcionalidades útiles para analizar la ejecución de un programa

Funcionalidades

Esta unidad de debug ofrece las siguientes funcionalidades:

- Cargar un programa en memoria de instrucciones mediante comunicación serie
  - El programa se carga instrucción por instrucción, byte a byte, por puerto serie
  - El programa debe terminar sí o sí en una instrucción HALT
  - Al hallar una instrucción HALT, la carga se detiene (aunque el archivo .asm tenga más instrucciones luego del HALT, esas no se van a cargar en memoria)
- Obtener datos tanto de la memoria de datos como de los registros
  - Mediante comandos de tipo "get" y punteros, se puede acceder a una posición de memoria de datos o del RF para ver los datos almacenados
- Divisar el valor actual del program counter
- Permitir ejecutar de dos formas distintas un programa cargado en memoria
  - Continuo: Ejecutar el programa completo de principio a fin sin interrupciones
  - Paso a paso: Ejecutar el programa ciclo a ciclo mediante un comando de avance (sólo se avanza de a un ciclo cuando se recibe el comando de avance por puerto serie)

Módulos principales de la implementación

El procesador MIPS se implementa como un conjunto de módulos interconectados que representan las diferentes etapas del pipeline y las unidades de control y detección de riesgos. A continuación se describen los módulos principales de este trabajo:

MIPS

Módulo principal que representa el procesador completo. Este módulo incluye las cinco etapas del pipeline (IF, ID, EX, MEM, WB) y también se conecta con las unidades de control y detección de riesgos. Es responsable de orquestar todas las etapas del pipeline y garantizar que las instrucciones se ejecuten correctamente.

ALU

Este módulo representa la unidad aritmético-lógica (ALU) del procesador. Es responsable de realizar las operaciones aritméticas y lógicas que requieren las instrucciones del tipo R, I o J, y se comunica con las etapas pertinentes del pipeline para transferir los datos adecuadamente.

Separador de bytes

Se encarga de dividir una palabra de 32 bits en bytes individuales y enviarlos a través de la UART para la carga del programa en memoria de instrucciones.

UART

El módulo UART es responsable de la comunicación serie entre la interfaz gráfica (que se explicará más adelante) y el procesador MIPS. Permite enviar datos (por ejemplo, el programa en formato .hex) desde la interfaz gráfica al procesador, y también permite enviar información de depuración y estado del procesador desde el procesador a la interfaz gráfica.

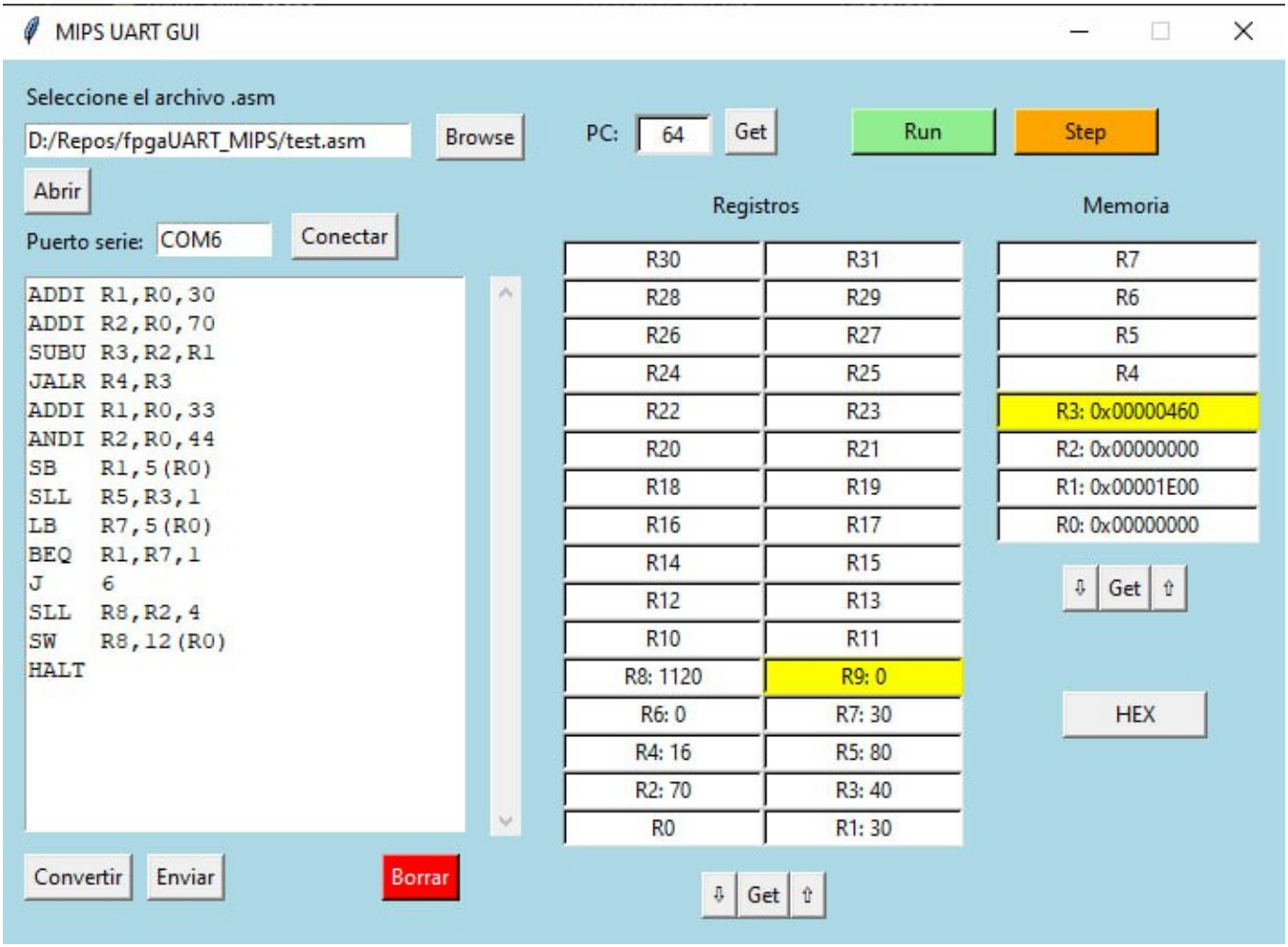
Interfaz gráfica

De igual manera que en el trabajo práctico anterior, se propuso una abstracción mayor sobre la conexión e interacción entre la unidad de debug y la implementación del procesador, por lo que se desarrolló una interfaz gráfica en Python que facilita las funcionalidades implementadas mediante los siguientes componentes:

- Barra de búsqueda y botón "browse": Permite ingresar una ruta de un archivo en formato .asm manualmente, o buscarlo con un navegador de archivos del sistema
- Botón "abrir": Abre el archivo seleccionado y muestra su contenido en un área de texto
- Area de texto: Permite visualizar el contenido del archivo abierto (programa en assembly)
- Campo de puerto serie: Permite ingresar el nombre del puerto serie utilizado para conectarse con la placa
- Botón "conectar": Verifica que el puerto ingresado sea válido y esté disponible, y se conecta
- Botón "convertir": Convierte cada instrucción del archivo .asm en su opcode equivalente y genera un archivo .hex
- Botón "enviar": Carga el programa en formato .hex por puerto serie a la memoria de instrucciones implementada
- Botón "borrar": Borra el programa cargado
- Campo PC y botón "get" asociado: Permite visualizar el valor del program counter al presionar el botón "get"
- Campos de registros del RF (R0-R31) y botones asociados: Mediante los botones de flechas el usuario puede desplazarse entre los registros del RF, y puede solicitar ver su contenido mediante el botón "get" (el registro solicitado se pinta para distinguirlo)

- Campos de registros de la memoria de datos (R0-R7) y botones asociados: Mediante los botones de flechas el usuario puede desplazarse entre los registros de la memoria de datos, y puede solicitar ver su contenido mediante el botón "get" (el registro solicitado se pinta para distinguirlo)
- Botón "hex": Este botón togglea el formato en el que se muestran los datos de los registros. Permite elegir entre formato hexadecimal (hex) o decimal (dec)
- Botón "run": Ejecuta el programa completo en el modo de ejecución continuo
- Botón "step": Ejecuta el programa en el modo de ejecución paso a paso

La interfaz gráfica desarrollada puede verse en funcionamiento a continuación:



Programas de ejemplo

Para validar el funcionamiento de la implementación completa, se desarrollaron 4 programas:

- Uno que utiliza principalmente instrucciones de tipo R (r-type-test.asm) con su respectivo análisis
- Uno que utiliza principalmente instrucciones de tipo I (i-type-test.asm) con su respectivo análisis
- Uno que utiliza principalmente instrucciones de tipo J (i-type-test.asm) con su respectivo análisis
- Uno que utiliza instrucciones de todo tipo (test.asm)

Conclusiones

La implementación del procesador MIPS en FPGA fue un desafío complejo pero interesante. Este trabajo permitió ver reflejada la teoría sobre cómo el uso de un pipeline y las etapas de control y detección de riesgos permiten optimizar el rendimiento y evitar problemas de dependencias entre instrucciones. Además, ayudó mucho a reforzar la estrategia de diseño y desarrollo a la hora de trabajar con lenguajes de

descripción de hardware. Por otra parte, el programa desarrollado en Python facilitó la tarea de carga de programas y la visualización de los datos de depuración, añadiendo una capa más de abstracción al trabajo para tener un camino de stack completo desde la interfaz gráfica de usuario hasta la implementación propia del procesador MIPS en la placa.

Si bien este informe proporciona una visión general de la implementación del procesador MIPS y los detalles específicos pueden variar según la implementación, esta descripción es suficiente para entender el funcionamiento general de un procesador con arquitectura MIPS.