

Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales



Trabajo Práctico 3 - Arquitectura de Computadoras

MIPS Pipeline - Procesador segmentado de cinco etapas

Integrantes del grupo:

Casanueva, María Constanza

Riba, Franco

Estudiantes de Ingeniería en Computación

Profesores:

Rodriguez, Santiago

Pereyra, Martín

Año de cursado: 2023

Índice

Introducción	3
Requerimientos	3
Implementación	4
IF (Instruction Fetch):	4
IF/ID	5
ID (Instruction Decode)	6
Banco de Registros	7
Unidad de Control Principal	7
Módulos Auxiliares	8
ID/EX	8
EX (Execute)	9
Unidad de control ALU	10
ALU	11
EX/MEM	12
MEM (Memory Access)	12
Memoria de Datos	13
MEM/WB	14
WB (Write Back)	14
Unidad de detección de Hazards	15
Unidad de Cortocircuito (Fowarding)	16
Unidad de Debug	17
Interface Unit	18
UART	20
Generador de Baud Rate	20
Interfaz Python	21
Testbenches	22
Herramienta Clock Wizarding	23
Paths Críticos	24
Análisis de frecuencia	25
Explicación Programa	26
Bibliografía y recursos	27

Repositorio del proyecto en [Github](#)

Introducción

Durante el desarrollo de este informe se reportan los detalles de la implementación de un procesador MIPS, el cuál consiste de un pipeline segmentado de cinco etapas.

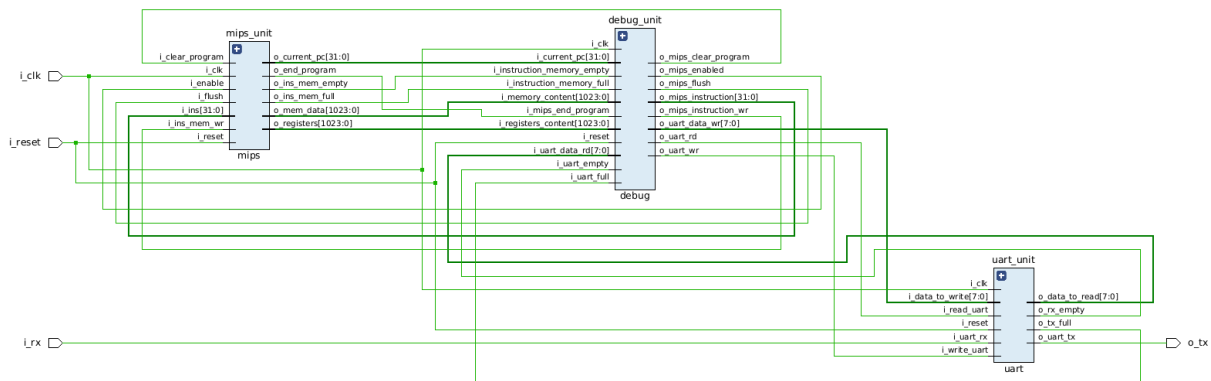
Requerimientos

- Implementar el Procesador MIPS Segmentado en las siguientes etapas:
 - **IF** (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
 - **ID** (Instruction Decode): Decodificación de la instrucción y lectura de registros.
 - **EX** (Execute): Ejecución de la instrucción propiamente dicha.
 - **MEM** (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
 - **WB** (Write back): Escritura de resultados en los registros.
- El procesador debe tener soporte para las siguientes instrucciones:
 - **R-type**: SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT
 - **I-Type**: LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL
 - **J-Type**: JR, JALR
- El procesador debe tener soporte para los siguientes tipos de riesgos:
 - **Estructurales**: Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
 - **De datos**: Se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
 - **De control**: Intentar tomar una decisión sobre una condición todavía no evaluada.
- Para dar soporte a los diversos riesgos se debe implementar:
 - Unidad de Cortocircuitos (short-circuit unit)
 - Unidad de Detección de Riesgos (Hazard detection unit)
- El programa a ejecutar debe ser cargado en la memoria del programa mediante un archivo ensamblado.
 - Debe implementarse un programa ensamblador que convierte código assembler de MIPS a código de instrucción.
 - Debe transmitirse ese programa mediante interfaz UART antes de comenzar a ejecutar.
- Se debe simular una unidad de Debug que envíe información hacia y desde el procesador mediante UART. Se debe enviar a la PC a través de la UART:
 - Contenido de los 32 registros del MIPS
 - PC (Program Counter)
 - Contenido de la memoria de datos
- Antes de estar disponible para ejecutar, el procesador está a la espera para recibir un programa mediante la Debug Unit. Debe permitir dos modos de operación:
 - Continuo: se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo (Instrucción HALT). Llegado ese punto se muestran todos los valores indicados en pantalla.
 - Paso a paso: Enviando un comando por la UART se ejecuta un ciclo de Clock. Se debe mostrar a cada paso los valores indicados.

Implementación

La solución propuesta se basa en la interacción entre tres grandes módulos dedicados a:

- Implementación del Pipeline del MIPS
- Implementación de la Unidad de Debug
- Implementación de la comunicación UART.

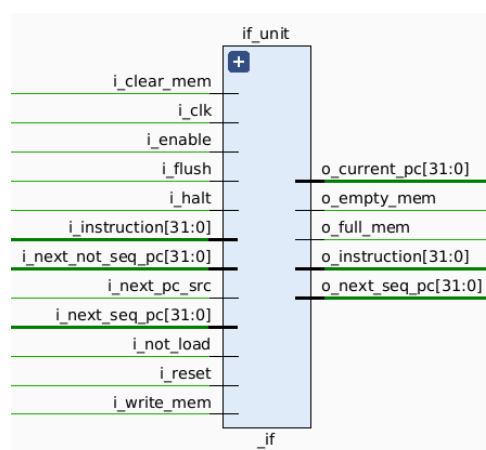


IF (Instruction Fetch):

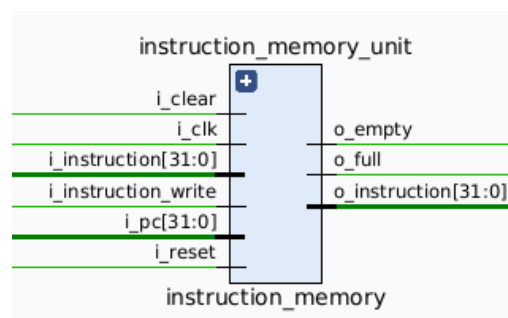
En esta etapa permite al procesador acceder a la memoria de instrucciones mediante el Program Counter y recupera la siguiente instrucción a ejecutar.

El valor del PC se obtiene de **i_next_seq_pc** o de **i_next_not_seq_pc** dependiendo del valor de selección en **i_next_pc_src**. Puede que se debe usar la instrucción siguiente del PC anterior o una dirección de salto, el valor del mismo se envía a la memoria de instrucciones.

La instrucción recuperada de la memoria de instrucciones es enviada a la etapa de decodificación a través de **o_instruction**. El PC se incrementa para apuntar a la dirección de la siguiente instrucción.

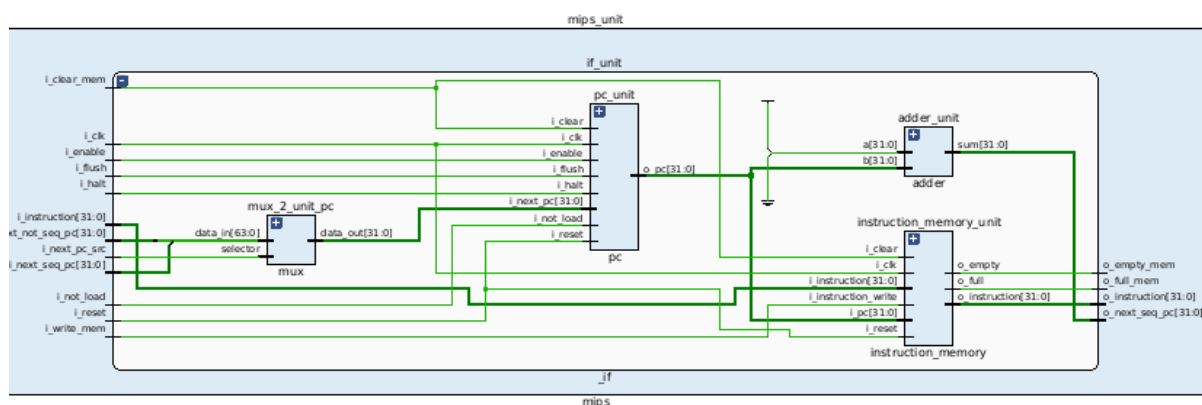


Dentro del módulo de la etapa de IF se encuentra implementada la memoria de instrucciones, con capacidades para ser leída o escrita:



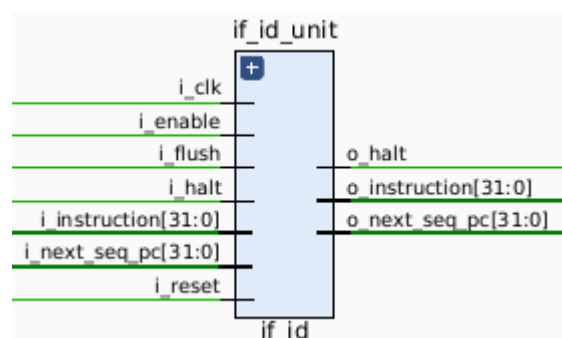
El módulo IF cuenta con otros submódulos útiles para distintas funciones:

- Multiplexor 2:1: para seleccionar la fuente del PC.
- PC (Program Counter): contador para mantener referencias a las direcciones de la siguiente instrucción a ejecutar.



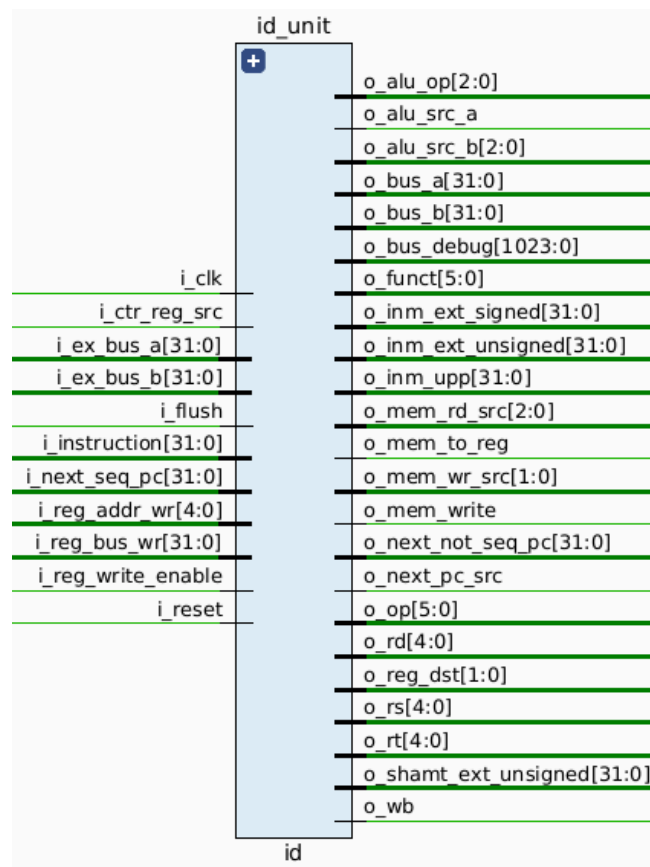
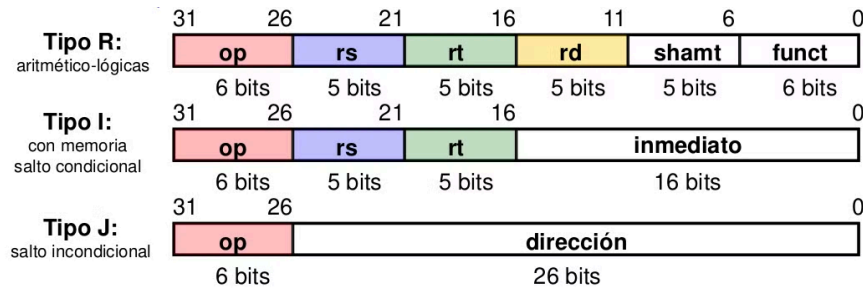
IF/ID

Este módulo contiene los registros de segmentación, cuya finalidad es mantener las salidas de la etapa IF, almacenando en registros la dirección de la instrucción siguiente en el program counter, así como la instrucción obtenida de la etapa IF y el flag `o_halt` el cual indica si el programa llegó a su fin.



ID (Instruction Decode)

Este módulo lleva a cabo la decodificación de la instrucción y lectura del banco de registros, representando la segunda etapa del pipeline. Los 3 tipos de instrucciones de 32 bits que se decodificaron son los siguientes:

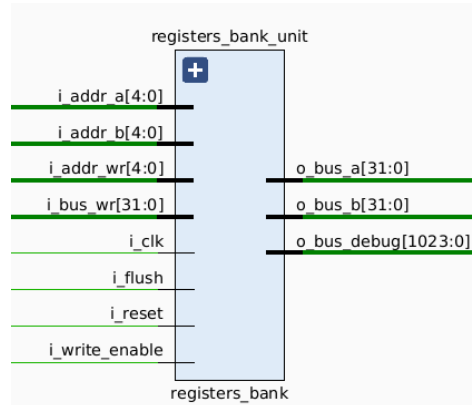


En la etapa de decodificación, la instrucción recuperada se interpreta para determinar qué acciones se deben llevar a cabo. Se extraen los campos de la instrucción, como los códigos de operación, código de función y dirección de registros fuente y destino.

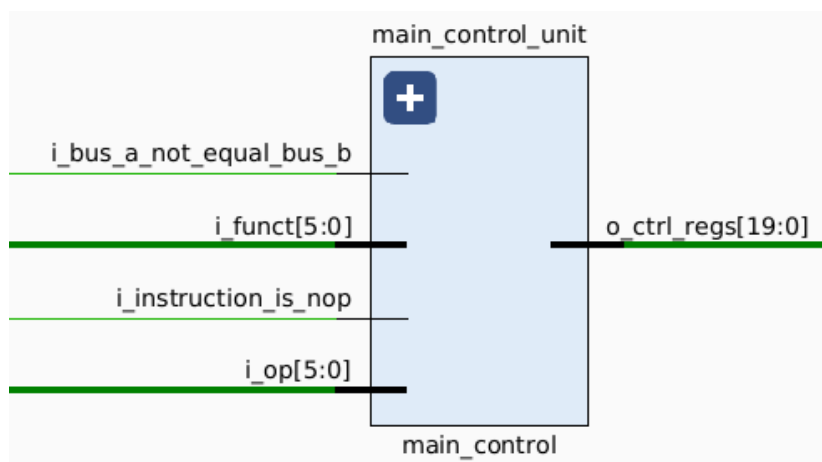
En esta etapa también se realiza el cálculo de la dirección de salto en caso de encontrarse ante una instrucción que lo requiera para no perder ciclos (Jump y Branch). En el caso de los branch también calcula la igualdad de los operandos para las instrucciones BEQ y BNE.

Banco de Registros

Dentro del módulo de la etapa de ID se encuentra implementado el banco de registros, aquí se ubican los 32 registros del MIPS que serán utilizados principalmente para la extracción de los operandos. El banco de registros permite la lectura, escritura y reinicio de registros.



Unidad de Control Principal




- **i_instruction:** Instrucción que se recupera desde la memoria de instrucciones y se utiliza para decodificar las señales de control. Los diferentes campos de la instrucción determinan las acciones a realizar.
- **i_bus_a_not_equal_bus_b:** Señal que llega del comparador que verifica si los dos registros fuente (bus_a y bus_b) son iguales. Se utiliza, por ejemplo, en instrucciones de salto condicional como BNE (Branch Not Equal).
- **i_instruction_is_nop:** Indica si la instrucción actual es un NOP (una instrucción vacía).
- **i_funct:** el código de función

A partir de estas entradas, la unidad de control determina el valor de las señales de control que se utilizan para controlar diferentes unidades funcionales del procesador:

- **next_pc_src:** Controla la fuente del próximo valor del contador de programa (PC).
- **jmp_ctrl:** Controla el tipo de salto (si es un salto y su dirección).
- **reg_dst:** Selecciona el destino del resultado de la ALU para escribir en el registro.
- **alu_src_a** y **alu_src_b:** Seleccionan las fuentes de entrada para la ALU.
- **code_alu_op:** Especifica la operación de la ALU.

- **mem_rd_src**: Selecciona la fuente de datos de lectura de memoria.
- **mem_wr_src**: Selecciona la fuente de datos de escritura en memoria.
- **mem_write**: Controla si se realiza una operación de escritura en memoria.
- **wb**: Controla si se habilita la escritura de nuevo valor en el registro.
- **mem_to_reg**: Controla si el resultado debe escribirse en el registro desde la memoria.

Todas estas señales se transmiten concatenadas en el bus **o_ctrl_regs** que se utiliza para controlar las diferentes unidades funcionales del MIPS.

Tabla de verdad de la Unidad de Control Principal (acceso con mail UNC):  [TablasDeVerdadTP3](#)

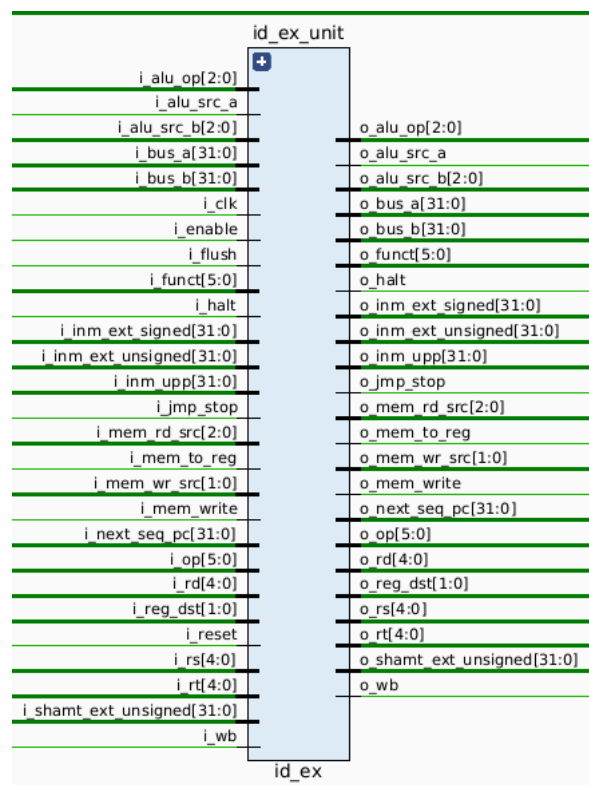
Módulos Auxiliares

El módulo ID cuenta también con otros submódulos útiles para distintas funciones:

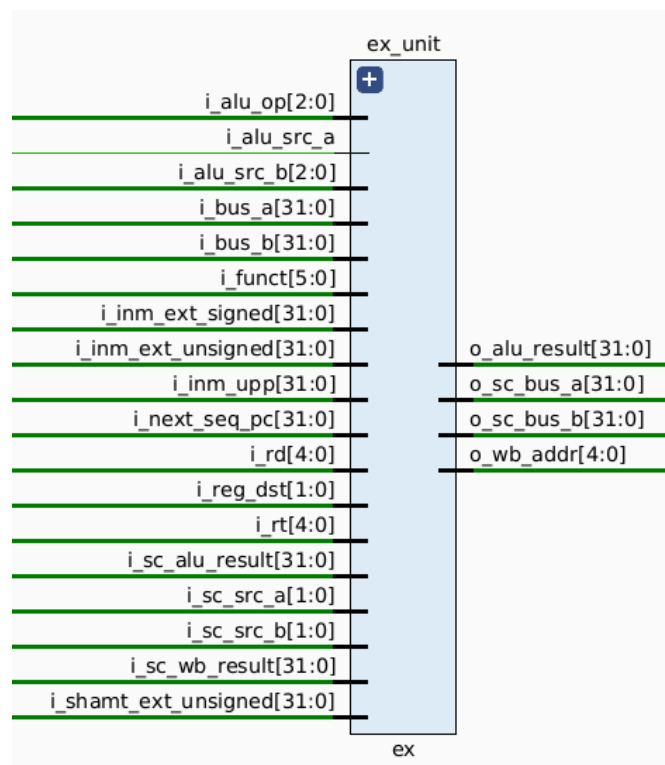
- **Is Not Equal Unit**: Este módulo compara los dos operandos provenientes del banco de registros para determinar si son diferentes, lo cual es útil para instrucciones como BNE (Branch if Not Equal).
- **Is Nop Unit**: Detecta si la instrucción actual es un NOP (instrucción vacía), lo que permite gestionar correctamente los ciclos sin acción en el pipeline.
- **Adder Unit**: Realiza sumas de 32 bits, generalmente utilizada para calcular direcciones de salto o branch, sumando la dirección base con un valor de desplazamiento.
- **Mux Units**: Existen varios multiplexores (como mux_ctr_regs_unit, mux_jump_src_unit) que se encargan de seleccionar entre diferentes entradas para dirigir el flujo de datos correcto hacia los siguientes módulos.
- **Shift Left Units**: Estos submódulos realizan operaciones de desplazamiento lógico de los operandos, útiles para calcular las direcciones de salto o hacer cambios de representación.
- **Unsigned Extend Units**: Extienden los valores inmediatos de la instrucción de manera adecuada (signo o sin signo) para que se puedan usar en operaciones de 32 bits.

ID/EX

En este módulo se registran los operandos y otros campos relevantes extraídos de la instrucción, así como también las señales de control generadas por la unidad de control de la etapa ID y el valor del extensor de signo.



EX (Execute)

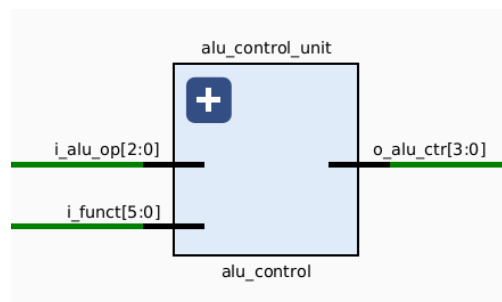


Esta es la tercera etapa del pipeline en el procesador MIPS. Aquí, se realizan las operaciones aritméticas y lógicas especificadas por la instrucción decodificada.

- Las entradas del módulo incluyen señales de control como **i_alu_src_a**, **i_alu_src_b**, **i_reg_dst**, **i_alu_op**, **i_sc_src_a**, **i_sc_src_b**, **i_rt**, **i_rd**, **i_funct**, y buses de datos como **i_sc_alu_result**, **i_sc_wb_result**, **i_bus_a**, **i_bus_b**, **i_shamt_ext_unsigned**, **i_inm_ext_signed**, **i_inm_upp**, **i_inm_ext_unsigned**, **i_next_seq_pc**. Las salidas del módulo incluyen **o_wb_addr**, **o_alu_result**, **o_sc_bus_b**, **o_sc_bus_a**.

El módulo posee múltiples submódulos para realizar las operaciones necesarias. Entre ellos una unidad ALU para realizar las operaciones aritméticas y lógicas, una unidad de control ALU (**alu_control_unit**) para generar las señales de control para la ALU, y varios multiplexores (**mux_alu_src_data_a_unit**, **mux_alu_src_data_b_unit**, **mux_sc_src_a_unit**, **mux_sc_src_b_unit**, **mux_reg_dst_unit**) para seleccionar los datos de entrada para la ALU y la dirección del registro de destino para la etapa WB.

Unidad de control ALU



Esta unidad genera las señales de control para la ALU en función de los códigos de operación y función de la instrucción.

Las entradas del módulo son **i_funct** (el campo funct de la instrucción, que especifica la operación a realizar en caso de instrucciones de tipo R) e **i_alu_op** (el código de operación de la ALU, que especifica la operación a realizar). Esta última viene de la unidad de control principal.

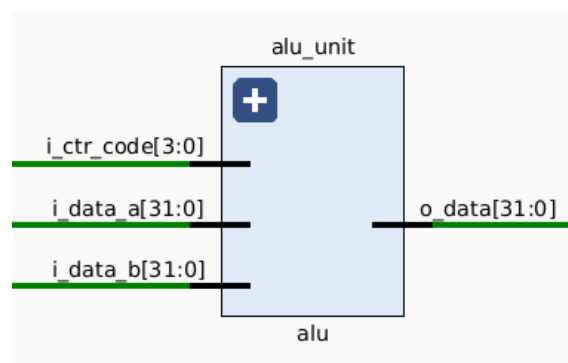
La salida del módulo es **o_alu_ctr** (la señal de control para la ALU, que indica la operación que debe realizar).

Tabla de control de la ALU control unit: [+ TablasDeVerdadTP3](#)

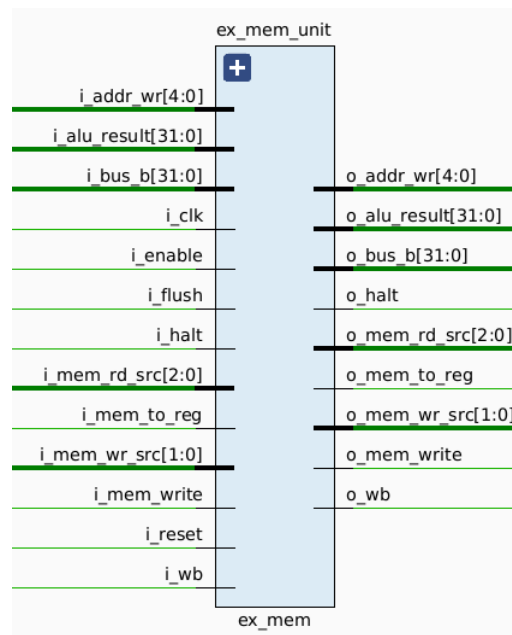
i_alu_op	i_func	o_alu_ctr	operación
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SLL`	`0000`	`b << a`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SRL`	`0001`	`b >> a`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SRA`	`0010`	`Signed(b) >>> a`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_ADD`	`0011`	`Signed(a) + Signed(b)`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_ADDU`	`0100`	`a + b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SUB`	`0101`	`Signed(a) - Signed(b)`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SUBU`	`0110`	`a - b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_AND`	`0111`	`a & b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_OR`	`1000`	`a b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_XOR`	`1001`	`a ^ b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_NOR`	`1010`	`~(a b)`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SLT`	`1011`	`Signed(a) < Signed(b)`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SLLV`	`1100`	`a << b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SRLV`	`1101`	`a >> b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_SRAV`	`1110`	`Signed(a) >>> b`
`CODE_ALU_CTR_R_TYPE`	`CODE_FUNC_JALR`	`1111`	`b`
`CODE_ALU_CTR_LOAD_TYPE`	`*****`	`0011`	`Signed(a) + Signed(b)`
`CODE_ALU_CTR_STORE_TYPE`	`*****`	`0011`	`Signed(a) + Signed(b)`
`CODE_ALU_CTR_JUMP_TYPE`	`*****`	`1111`	`b`
`CODE_ALU_CTR_ADDI`	`*****`	`0011`	`Signed(a) + Signed(b)`
`CODE_ALU_CTR_ANDI`	`*****`	`0111`	`a & b`
`CODE_ALU_CTR_ORI`	`*****`	`1000`	`a b`
`CODE_ALU_CTR_XORI`	`*****`	`1001`	`a ^ b`
`CODE_ALU_CTR_SLTI`	`*****`	`1011`	`Signed(a) < Signed(b)`

ALU

La implementación de este módulo sigue la misma lógica y diseño planteados y utilizados para los dos trabajos prácticos previos. Realiza la operación indicada por el ALU code, a partir de los dos operandos que se encuentran en su entrada.

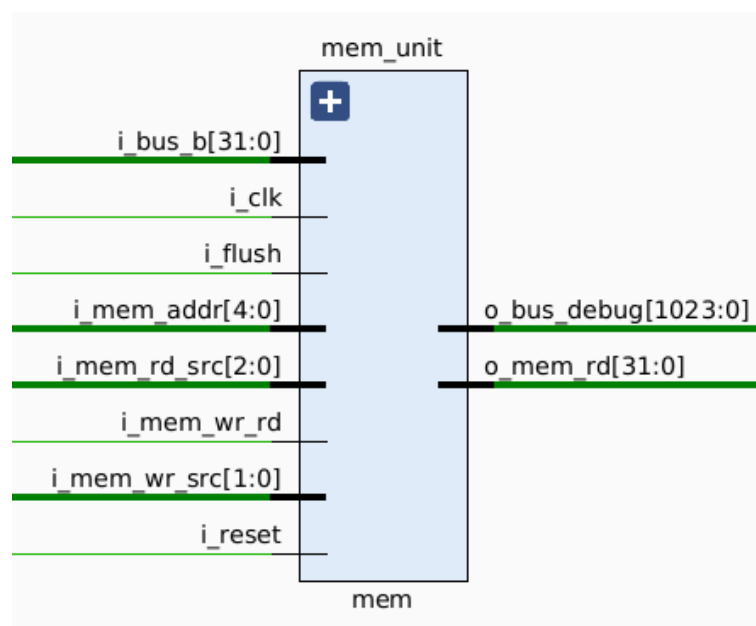


EX/MEM



En este módulo se registra el resultado de la ALU obtenido en la etapa EX para ser usado en la siguiente etapa. También se registra la dirección de memoria calculada en caso de instrucciones de acceso a memoria y las señales de control que se utilizarán más adelante.

MEM (Memory Access)



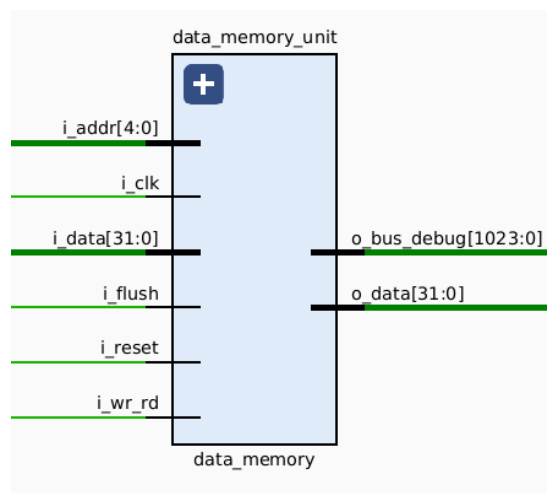
En esta etapa, si la instrucción es una operación de carga o almacenamiento, se accede a la memoria de datos. El módulo `mem` en el código Verilog representa esta etapa. Este módulo tiene varias entradas y salidas que permiten su interacción con otras etapas del pipeline y con la memoria de datos.

Las entradas del módulo incluyen señales de control como **i_clk** (la señal de reloj), **i_reset** (para reiniciar el módulo), **i_flush** (para limpiar la memoria de datos), **i_mem_wr_rd** (para indicar si se debe realizar una operación de escritura o lectura en la memoria), **i_mem_wr_src** (para seleccionar la fuente de los datos a escribir en la memoria), **i_mem_rd_src** (para seleccionar cómo se deben leer los datos de la memoria), y **i_mem_addr** (la dirección de la memoria a la que se debe acceder). También incluye una entrada para los datos a escribir en la memoria (**i_bus_b**).

Las salidas del módulo son **o_mem_rd** (los datos leídos de la memoria) y **o_bus_debug** (bus para depuración que almacena los datos de la memoria).

Dentro del módulo, se utilizan varios componentes para realizar las operaciones necesarias. Entre ellos un módulo `data_memory` para representar la memoria de datos, varios multiplexores (`mux_in_mem_unit`, `mux_out_mem_unit`) para la selección de los datos a escribir en la memoria y cómo se deben leer los datos de la memoria, y varias instancias de módulos `unsig_extend` y `sig_extend` para extender los datos leídos de la memoria a la longitud correcta, ya sea con o sin extensión de signo.

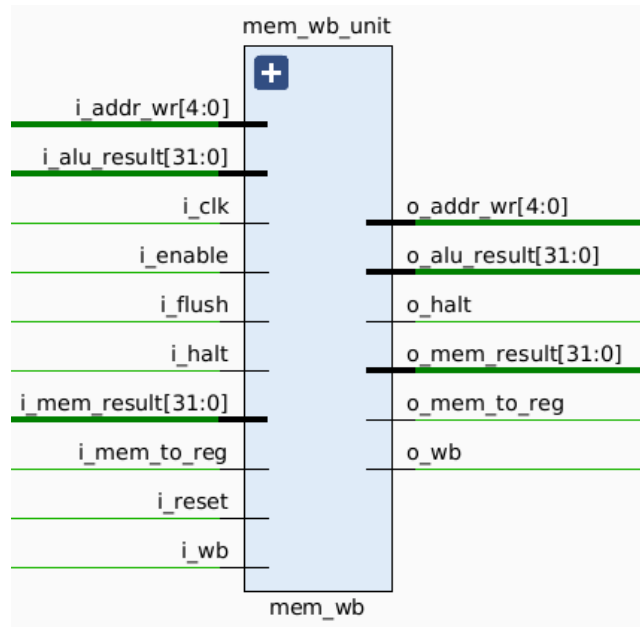
Memoria de Datos



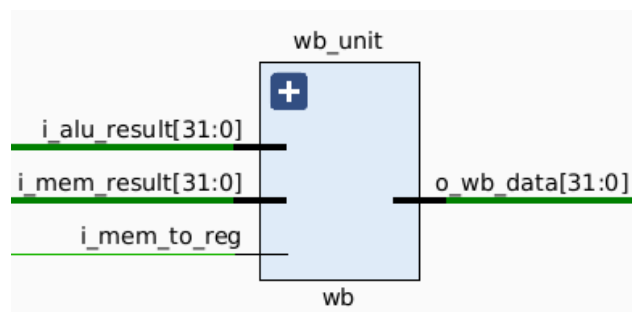
Es una memoria de 255 bytes a la cual se puede acceder tanto para escritura en instrucciones de tipo store, como para lectura en instrucciones de tipo load. Se puede acceder a la misma en tamaño de byte (8 bits), half word (16 bits) o word (32 bits) dependiendo el tipo de instrucción.

MEM/WB

En este módulo se registran tanto la salida de ALU como la salida de la memoria. También se registran las últimas señales de control que llegan hasta la última etapa



WB (Write Back)



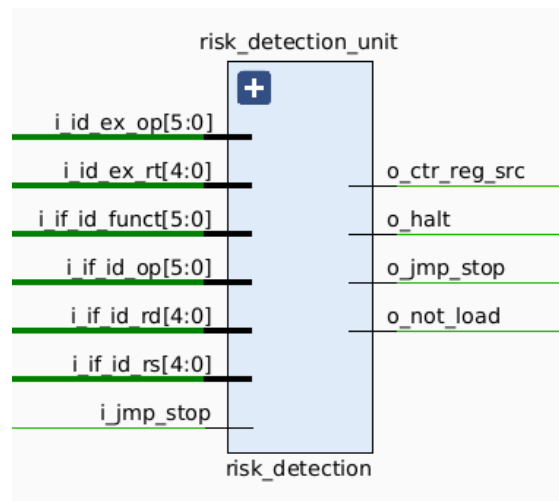
Es la quinta y última etapa del pipeline en el procesador. En esta etapa, los resultados de la ejecución de la instrucción se escriben de vuelta en los registros. Este módulo tiene varias entradas y una salida que permiten su interacción con otras etapas del pipeline.

Las entradas del módulo incluye **i_mem_to_reg** (una señal de control que indica si el resultado de la operación de memoria debe ser escrito en los registros), **i_alu_result** (el resultado de la operación de la ALU) e **i_mem_result** (el resultado de la operación de memoria).

La salida del módulo es **o_wb_data** (los datos que se deben escribir en los registros).

Dentro del módulo, se utiliza un multiplexor (**mux_wb_data**) para seleccionar entre **i_alu_result** e **i_mem_result** en función de la señal **i_mem_to_reg**. El resultado de esta selección se envía a la salida **o_wb_data**.

Unidad de detección de Hazards



Esta unidad se utiliza para detectar y manejar riesgos de datos y control. Las entradas del módulo son: **i_jump_stop**, **i_if_id_rs**, **i_if_id_rd**, **i_if_id_op**, **i_if_id_func**, **i_id_ex_rt**, **i_id_ex_op**: Estas señales representan varias partes de las instrucciones en las etapas IF/ID y ID/EX del pipeline, incluyendo los códigos de operación y función, los registros fuente y destino, y una señal que indica si se debe detener el salto.

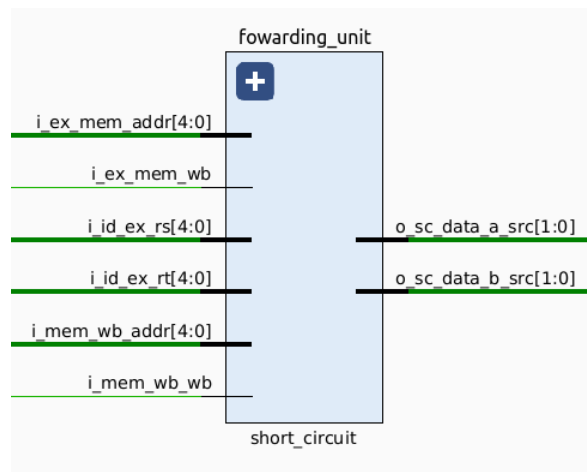
Las salidas del módulo son:

- **o_jump_stop**: Se activa si se debe detener el procesador un ciclo debido a un salto, lo cual ocurre si la instrucción en la etapa IF/ID es un salto y la señal **i_jump_stop** no está activa.
- **o_halt**: Se activa si la instrucción en la etapa IF/ID es una instrucción de finalización de programa.
- **o_not_load**: Se activa si la instrucción en la etapa ID/EX es una instrucción de carga y el registro destino coincide con uno de los registros fuente de la instrucción en la etapa IF/ID, o si la señal **o_jump_stop** está activa. Esto indica un riesgo de datos.
- **o_ctr_reg_src**: Igual a la señal **o_not_load** y se utiliza para indicar si las señales de control se propagan a las siguientes etapas o no.

La lógica del módulo verifica las instrucciones en las etapas IF/ID y ID/EX del pipeline y activa las señales de salida correspondientes si detecta un riesgo o si se da una instrucción de salto que haga uso de los registros del procesador (potencial riesgo).

La tabla de verdad para la unidad de detección de riesgos se basa en las señales de entrada y determina las señales de salida. Aquí está la tabla de verdad simplificada: [📄 TablasDeVerdadTP3](#)

Unidad de Cortocircuito (Fowarding)



Esta unidad se utiliza para implementar el adelanto (forwarding) de datos, una técnica que ayuda a minimizar los riesgos de datos en el pipeline del procesador.

Las entradas del módulo son:

- **i_ex_mem_wb** y **i_mem_wb_wb**: Estas señales indican si hay datos válidos en las etapas EX/MEM y MEM/WB del pipeline, respectivamente.
- **i_id_ex_rs** y **i_id_ex_rt**: Estos son los registros fuente de la etapa ID/EX del pipeline.
- **i_ex_mem_addr** y **i_mem_wb_addr**: Estas son las direcciones de memoria de las etapas EX/MEM y MEM/WB del pipeline, respectivamente.

Las salidas del módulo son:

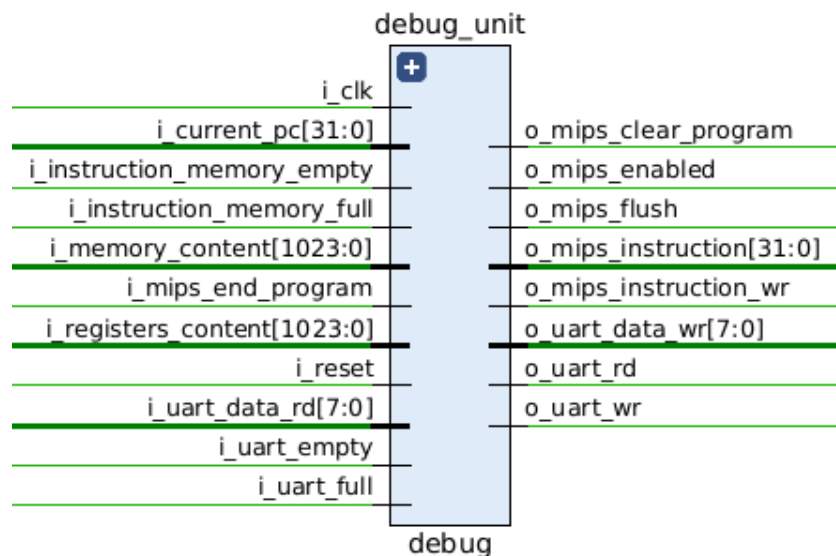
- **o_sc_data_a_src** y **o_sc_data_b_src**: Estas señales indican la fuente de los datos para los registros fuente rs y rt, respectivamente.

La lógica del módulo verifica si las direcciones de memoria de las etapas EX/MEM y MEM/WB coinciden con los registros fuente de la etapa ID/EX. Si es así, y si hay datos válidos en las etapas EX/MEM y MEM/WB, entonces los datos se adelantan desde estas etapas. Si no, los datos provienen de la etapa ID/EX.

Esto ayuda a minimizar los riesgos de datos al permitir que las instrucciones utilicen los resultados de las instrucciones anteriores tan pronto como estén disponibles, en lugar de esperar a que las instrucciones anteriores completen su ejecución y los resultados se escriban en los registros.

La tabla de verdad de la unidad de cortocircuito: [+ TablasDeVerdadTP3](#)

Unidad de Debug



Este módulo actúa como una unidad de depuración para el procesador MIPS, gestionando la comunicación con la UART y controlando la impresión de datos del procesador, incluidos los registros y la memoria. Proporciona señales de control para realizar diferentes operaciones de depuración y comunicarse con el módulo interface, que se encarga de coordinar las acciones de depuración.

Parámetros:

- Define diferentes tamaños de buses para la UART, los registros, la memoria y otras estructuras de datos.

Entradas:

- Señales de reloj y reset.
- Indicadores de vaciado y llenado de memoria e instrucciones.
- Contenidos de los registros, la memoria y el PC del procesador MIPS.

Salidas:

- Señales de control para escribir y leer desde la UART, habilitar el procesador MIPS, borrar su memoria, y enviar instrucciones al procesador.

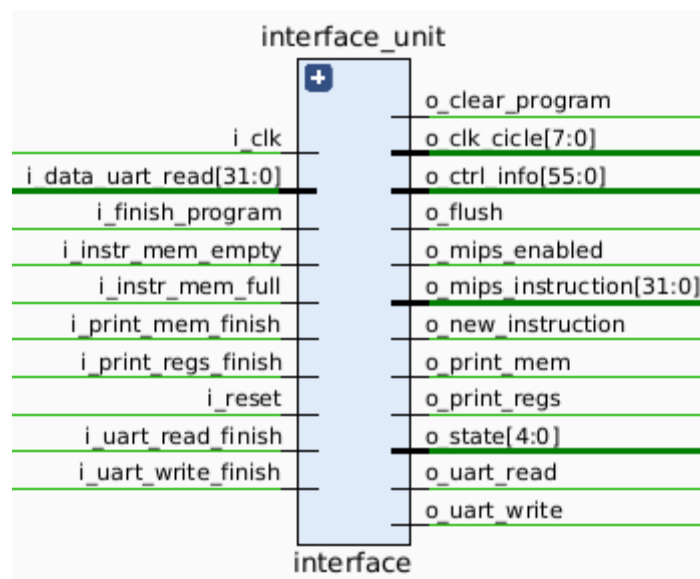
Interacción con otros módulos:

- Instancia los módulos **buffer_reader**, **buffer_writer**, y dos **reg_printer** para manejar la comunicación con la UART y la impresión de datos (de registros y memoria).
- **Interface:** Este es el submódulo clave que gestiona la interacción entre el módulo debug y el procesador MIPS. Controla el flujo de depuración, ejecutando comandos como la carga de nuevas instrucciones y la impresión de datos del procesador.
- En el debug, las señales internas (**start_uart_wr**, **start_register_print**, etc.) definen cuándo comenzar a leer/escribir desde/hacia la UART, cuándo imprimir los registros o la memoria, y más.

Interface Unit

La unidad de debug admite los modos de ejecución continuo y paso a paso (coordinados por el módulo interface que forma parte de la debug unit):

- **Continuo:** Se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo (Instrucción HALT). Llegado ese punto se muestran todos los valores en pantalla.
- **Paso a paso:** Enviando un comando por la UART se ejecuta un ciclo de clock. Se debe mostrar a cada paso los valores.



El módulo interface es el encargado de controlar el flujo de la depuración. Este módulo se asegura de que las diferentes etapas de depuración (como la lectura/escritura desde/hacia la UART, y la impresión de registros/memoria) ocurran en el orden correcto, controlando así los diferentes estados de la depuración.

Entradas:

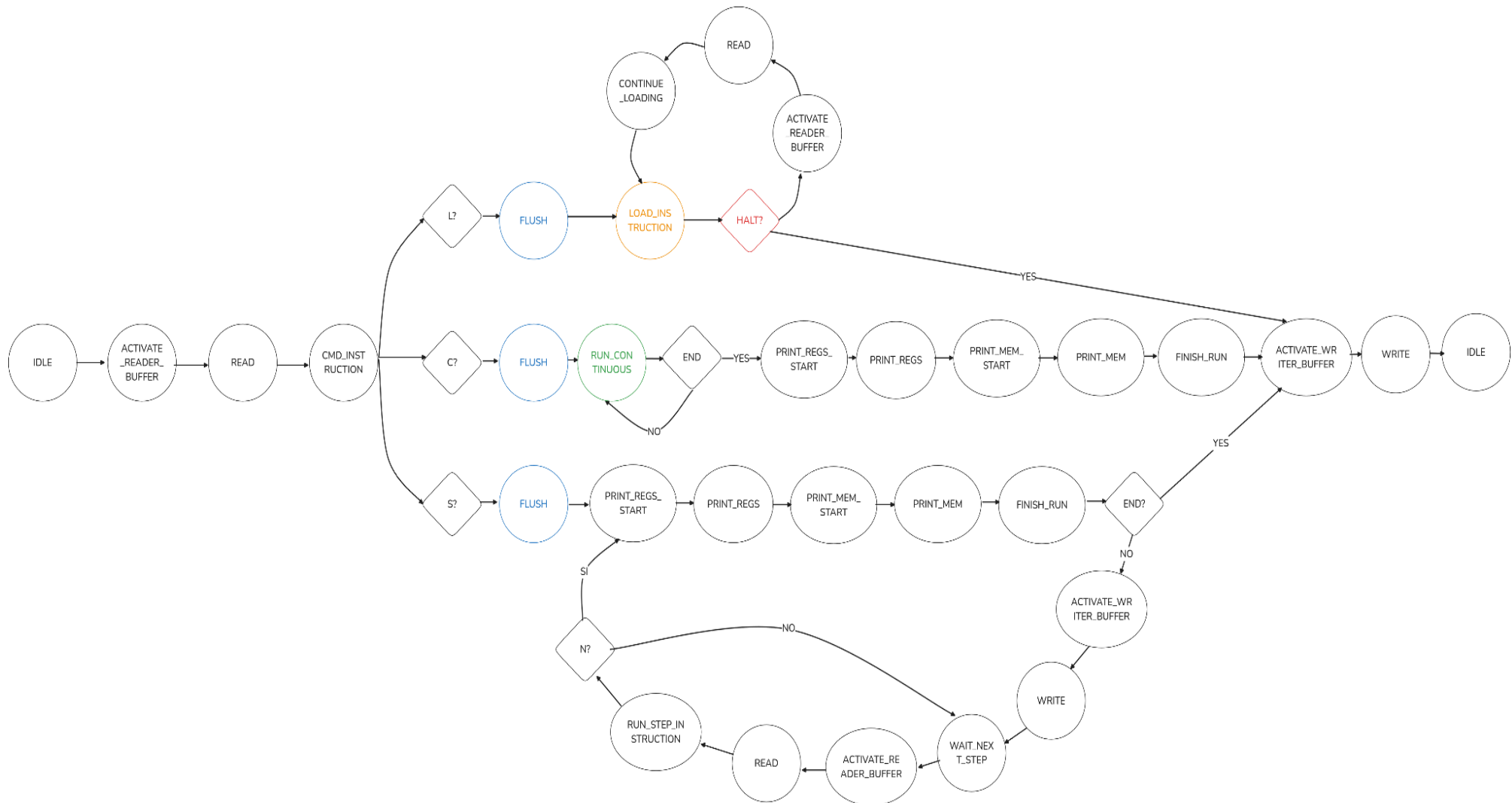
- Recibe señales del procesador MIPS y del módulo debug (finalización de programas, lectura de UART completada, etc.).

Salidas:

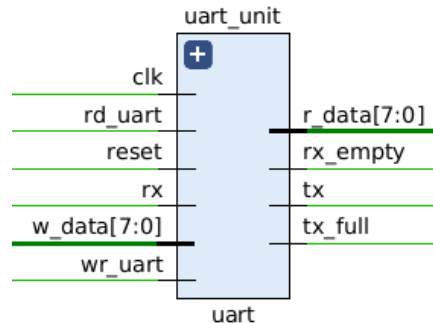
- Emite señales que activan la lectura/escritura desde la UART y comandos como la impresión de registros, la ejecución paso a paso, y el flush del pipeline.

Máquina de estados:

- El módulo interface utiliza una máquina de estados para gestionar las acciones de depuración. Los estados controlan cuándo leer o escribir en la UART, cuándo imprimir datos de los registros o memoria, cuándo ejecutar una instrucción MIPS, etc.



UART



El módulo `uart` tiene varios parámetros configurables, incluyendo el número de bits de datos (**DATA_BITS**), el número de ticks para el bit de inicio (**SB_TICKS**), el bit de precisión de la tasa de baudios (**DVSR_BIT**), el divisor de la tasa de baudios (**DVSR**) y el tamaño de la FIFO (**FIFO_SIZE**).

Las entradas del módulo incluyen la señal de reloj (`clk`), la señal de reset (`reset`), las señales de lectura y escritura de la UART (`rd_uart` y `wr_uart`), la señal de recepción (`rx`) y los datos a escribir (`w_data`). Las salidas del módulo incluyen las señales de llenado completo y vacío de la transmisión y recepción (`tx_full` y `rx_empty`), la señal de transmisión (`tx`) y los datos leídos (`r_data`).

El módulo `uart` consta de varias subunidades:

- **uart_brg**: Genera la señal de tick para la tasa de baudios.
- **uart_rx**: Maneja la recepción de datos.
- **uart_tx**: Maneja la transmisión de datos.
- **fifo_rx_unit** y **fifo_tx_unit**: Son colas FIFO para almacenar los datos recibidos y a transmitir, respectivamente.

Cada subunidad tiene su propia configuración y señales de entrada y salida, que se conectan a las entradas y salidas del módulo `uart` para formar un sistema de comunicación UART completo.

Además la comunicación `uart` implementada tiene las siguientes características

- Baudrate: 19200.
- Bytesize: 8 bytes.
- No parity bits
- Stop bit: 1

Generador de Baud Rate

Este módulo genera un pulso cada vez que se desborda un contador, dado que se tiene como requerimiento que la transmisión UART se realice a una velocidad de 19200 baudios, y la basys 3 estará operando a una frecuencia de 100 MHz, resulta necesario generar un tick cada 326 ciclos del clock.

$$COUNTER\ LIMIT = \frac{CLK_{freq}}{16 * Baudrate}$$

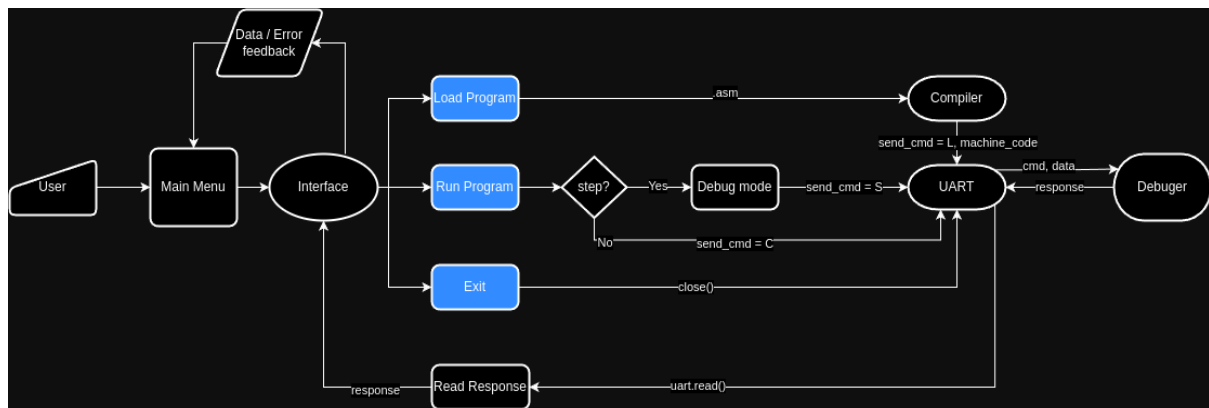
$$COUNTER\ LIMIT = 326$$

Interfaz Python

Los comandos para la selección de los modos de ejecución se envían a la Basys 3 por medio de un programa en Python por medio del cuál el usuario puede:

- Cargar un programa en la memoria de instrucciones
- Ejecutar el programa en modo continuo o en modo paso a paso recibiendo información de la evolución de los registros, la memoria y el PC durante la ejecución.

Aquí el diagrama de flujo del script de Python que permite esto:



Testbenches

tb_alu_ctrl: con este testbench verificamos el módulo de alu_ctrl, en donde simulamos diferentes casos para la ALU y comparamos resultados con valores esperados y nos aseguramos que el diseño de control de la ALU sea preciso. En esta instancia probamos instrucciones R-Type como las de salto y nos aseguramos que todo funcione como lo esperamos.

tb_ctrl_register : con este test bench comprobamos que el módulo ctrl_register funcione correctamente con todos los tipos de operaciones aritméticas, de carga de memoria, saltos. Cada vez que se prueba una operación imprimimos el valor y observamos si el modulo esta funcionando correctamente.

tb_ex: en este testbench lo que hacemos es verificar el modulo de ejecucion ex este funcionando correctamente donde evaluamos los resultados de la ALU y las direcciones de escritura donde se guardan y luego comparamos para ver que la ejecución fue la correcta

tb_extend: aca verificamos el módulo extend donde vemos que los valores pasan de 16 bits a 32 bits con los distintos tipos de pruebas, numero positivo con extension firmada, negativo con extension firmada, cero con extensión no firmada, máximo número positivo con extensión firmada, máximo negativo.

tb_id: verificamos la etapa de decodificación de instrucciones, el procesador interpreta las instrucciones de la memoria y prepara las señales que se necesitan para ejecutar la instrucción, incluimos las instrucciones aritméticas, de salto condicional e incondicional. Si las señales generadas por el módulo al compararlas con los valores esperados consideramos que el módulo esta correctamente diseñado.

tb_is_equal: aca verificamos que el módulo is_equal este comparando correctamente los valores, en esta instancia comparamos varias situaciones y verificamos que la salida sea la esperada asi podemos ver que este módulo está funcionando correctamente

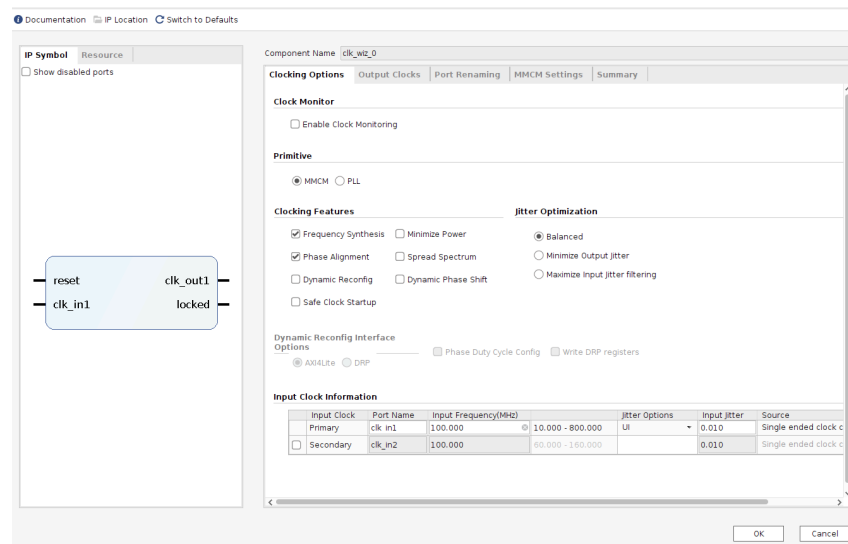
tb_is_not: verificamos si el módulo puede identificar correctamente cuando una instrucción es un nop diseñamos el testbench para que cada 10 unidades de tiempo el módulo procesa la entrada, después imprime el valor de entrada i_opp y su respectiva salida o_is_nop si el resultado es el que esperábamos la prueba paso y es un test ok si no es un test failed.

tb_shift_left: comprobamos que el desplazamiento lógico hacia la izquierda funcione correctamente, en cada caso de prueba evaluamos una entrada distinta para poder asi verificar que el comportamiento del módulo sea el esperado bajo diferentes condiciones.

tb_wb: en este módulo simulamos para la fase de write back, en este testbench verificamos que sea capaz de elegir correctamente.

Herramienta Clock Wizarding

Utilizando la herramienta de Clock Wizarding que provee Vivado pudimos determinar el rango de frecuencias en el cual nuestra implementación cumple con los requisitos de timing. Se puede acceder a la herramienta desde el menú lateral de Vivado, en la sección de “IP Catalog”. Al usar la herramienta es posible seleccionar la frecuencia de trabajo, la herramienta se ocupa de generar dicha frecuencia a partir del clock disponible, que en nuestro caso, por usar la Basys 3 es de 100MHz.



Component Name: clk_wiz_0

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)	Requested	Actual	Phase (degrees)	Requested	Actual	Duty Cycle (%)	Requested	Actual
<input checked="" type="checkbox"/> clk_out1	clk_out1	100.000	100.000	100.00000	0.000	0.000	50.000	50.000	50.000	50.000
<input type="checkbox"/> clk_out2	clk_out2	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A
<input type="checkbox"/> clk_out3	clk_out3	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	N/A	0.000	N/A	50.000	N/A	N/A	N/A

De esta forma fuimos modificando la frecuencia hasta encontrar una en la que los requisitos de Timing sean cumplidos para todos los paths críticos de nuestro diseño.

Paths Críticos

Con una frecuencia de trabajo de 100MHz nuestro diseño tiene 10 paths críticos para los que no se cumplen los requisitos de timing.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0.402 ns	Worst Hold Slack (WHS): 0.064 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): -11.233 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 48	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 14888	Total Number of Endpoints: 14888	Total Number of Endpoints: 5022

Timing constraints are not met.

Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic De
Path 1	-0.402	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[1]_repD	5.279	1.1
Path 2	-0.400	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[4]_rep_3D	5.313	1.1
Path 3	-0.397	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[1]D	5.309	1.1
Path 4	-0.393	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[4]_rep_1D	5.306	1.1
Path 5	-0.391	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[2]_rep_0D	5.306	1.1
Path 6	-0.379	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[4]_repD	5.249	1.1
Path 7	-0.369	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[1]7D	5.345	1.1
Path 8	-0.367	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[2]_repD	5.245	1.1
Path 9	-0.347	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[2]_rep_2D	5.261	1.1
Path 10	-0.338	9	50	mips_unitmem...[4]_replicaC	mips_unit/uf_unit/pc_unit/pc_reg[4]_rep_0D	5.254	1.1

Al bajar la frecuencia a 95MHz pasamos a tener tan solo dos paths críticos que siguen sin cumplir los requisitos de timing.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0.022 ns	Worst Hold Slack (WHS): 0.045 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): -0.025 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 2	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 14882	Total Number of Endpoints: 14882	Total Number of Endpoints: 5020

Timing constraints are not met.

Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination C
Path 1	-0.022	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...g[4]_rep_4D	5.149	1.806	3.343	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 2	-0.004	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...g[4]_rep_2D	5.134	1.806	3.328	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 3	0.000	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...g[4]_rep_0D	5.159	1.816	3.343	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 4	0.001	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...t/pc_reg[0]D	5.128	1.806	3.322	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 5	0.061	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...g[3]_rep_3D	5.068	1.806	3.262	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 6	0.064	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...pc_reg[2]1D	4.987	1.806	3.181	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 7	0.068	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...pc_reg[4]_repD	5.060	1.806	3.254	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 8	0.070	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...g[4]_rep_3D	5.057	1.806	3.251	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 9	0.078	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...t/pc_reg[1]D	5.085	1.806	3.279	5.3	clk_out1_clk_wiz_0	clk_out1_clk_
Path 10	0.082	9	51	mips_unit/ld_e...itrs_reg[1]C	mips_unit/uf...reg[3]_repD	4.969	1.806	3.163	5.3	clk_out1_clk_wiz_0	clk_out1_clk_

Finalmente tras bajar la frecuencia a 90 MHz, todos los paths críticos cumplen los requisitos de timing, esto se traduce en que la métrica del WNS (Worst Negative Slack) es positiva, lo cual significa que incluso para el path más crítico del diseño “sobró” tiempo del periodo de la señal de reloj usada.

Tcl Console	Messages	Log	Reports	Design Runs	Timing	Power	Methodology	Package Pins	I/O Ports	?	
Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup											
<div>● Methodology Summary (4) Check Timing (5014) Intra-Clock Paths clk_out1_clk_wiz_0 Setup 0.059 ns (10) Hold 0.182 ns (10) Pulse Width 4.506 ns (31) clk_out1_clk_wiz_0_1 clkfbout_clk_wiz_0 clkfbout_clk_wiz_0_1 l_clk sys_clk_pin Inter-Clock Paths Other Path Groups</div>											
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination
Path 1	0.059	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...t/pc_reg[5]/D	5.288	1.789	3.499	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 2	0.075	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[3]/rep_3/D	5.292	1.789	3.503	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 3	0.196	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...pc_reg[29]/D	5.277	1.789	3.488	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 4	0.205	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...reg[1]/rep/D	5.213	1.789	3.424	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 5	0.212	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...pc_reg[28]/D	5.261	1.789	3.472	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 6	0.227	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[4]/rep_1/D	5.226	1.789	3.437	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 7	0.235	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[2]/rep_2/D	5.219	1.789	3.430	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 8	0.236	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[4]/rep_2/D	5.219	1.789	3.430	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 9	0.244	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[4]/rep_4/D	5.237	1.796	3.441	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Path 10	0.244	9	51	mips_unit/ld_e...t/rt_reg[3]/C	mips_unit/lf_u...g[2]/rep_0/D	5.173	1.789	3.384	5.6	clk_out1_clk_wiz_0	clk_out1_clk
Timing Summary - impl_1 (saved)											

Tcl ConsoleMessagesLogReportsDesign RunsTiming xPowerMethodologyPackage PinsI/O Ports

Design Timing Summary

General Information

Timer Settings

Design Timing Summary

Clock Summary (6)

Methodology Summary (4)

Check Timing (5014)

Intra-Clock Paths

- clk_out1_clk_wiz_0
 - Setup 0.059 ns (10)
 - Hold 0.123 ns (10)

Setup

Hold

Pulse Width

Worst Negative Slack (WNS): 0.059 ns

Worst Hold Slack (WHS): 0.054 ns

Worst Pulse Width Slack (WPWS): 3.000 ns

Total Negative Slack (TNS): 0.000 ns

Total Hold Slack (THS): 0.000 ns

Total Pulse Width Negative Slack (TPWS): 0.000 ns

Number of Failing Endpoints: 0

Number of Failing Endpoints: 0

Number of Failing Endpoints: 0

Total Number of Endpoints: 14872

Total Number of Endpoints: 14872

Total Number of Endpoints: 5017

All user specified timing constraints are met.

Análisis de frecuencia

Realizamos un análisis de frecuencia para determinar el rango de frecuencias en que nuestra implementación funciona de forma esperada, a modo de resumen presentamos los siguientes resultados obtenidos en base a los timing reports mostrados previamente y las pruebas físicas en la Basys 3.

		Resultados de ejecución incorrectos
175 MHz		
170 MHz		No se cumplen los time constrains pero se observa funcionamiento o correcto en el hardware
Fmax = 90 MHz		
		Se cumplen los time constrains y los resultados de ejecución son correctos
10 MHz		

Ejecución de Programas

Para interactuar con la placa desarrollamos una interfaz en Python. Aquí podemos cargar un programa en código ensamblador y luego ejecutarlo en modo continuo o paso a paso, en caso de usar el modo paso a paso, en cada step se resaltan con colores los registros o posiciones de memorias que fueron modificados durante ese ciclo, además de mostrar el valor del Program Counter en cada ciclo.



Dentro del directorio `asm_examples` del proyecto se pueden encontrar distintos programas en ensamblador, algunos de ellos para testear instrucciones específicas y otros son ejemplos más realistas de programas que combinan distintos tipos de instrucciones.

Programa 1

Propósito: Comparar dos números y ejecutar diferentes instrucciones dependiendo de si son iguales o no.

```

ADDI r1,r0,5      # r1 = 5
ADDI r2,r0,10     # r2 = 10
ADDI r3,r0,7      # r3 = 7
ADDI r4,r0,12     # r4 = 12
BEQ r1,r2,equal   # Si son iguales, salta a "equal" donde hace una and y termina
OR r5,r3,r4       # Si no son iguales, r5 = r3 | r4
J end             # Salta al final
equal: AND r5,r3,r4 # Si son iguales, r5 = r3 & r4
end: NOP          # No Operation
HALT              # Termina la ejecución
  
```

			0	0	4	8	C	10	10	14	18	20	24	24	24	24	24
			Cycle														
IM		Intruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		ADDI r1,r0,5	F	D	E	M	W										
4		ADDI r2,r0,10		F	D	E	M	W									
8		ADDI r3,r0,7			F	D	E	M	W								
C		ADDI r4,r0,12				F	D	E	M	W							
10		BEQ r1,r2,equal					F	D	E	M	W						
14		OR r5,r3,r4						(F)	F	D	E	M	W				
18		J end								F	D	E	M	W			
1C	equal	and r5,r3,r4															
20	end	nop									(F)	F	-	-	-	-	
24		halt											F	D	E	M	W

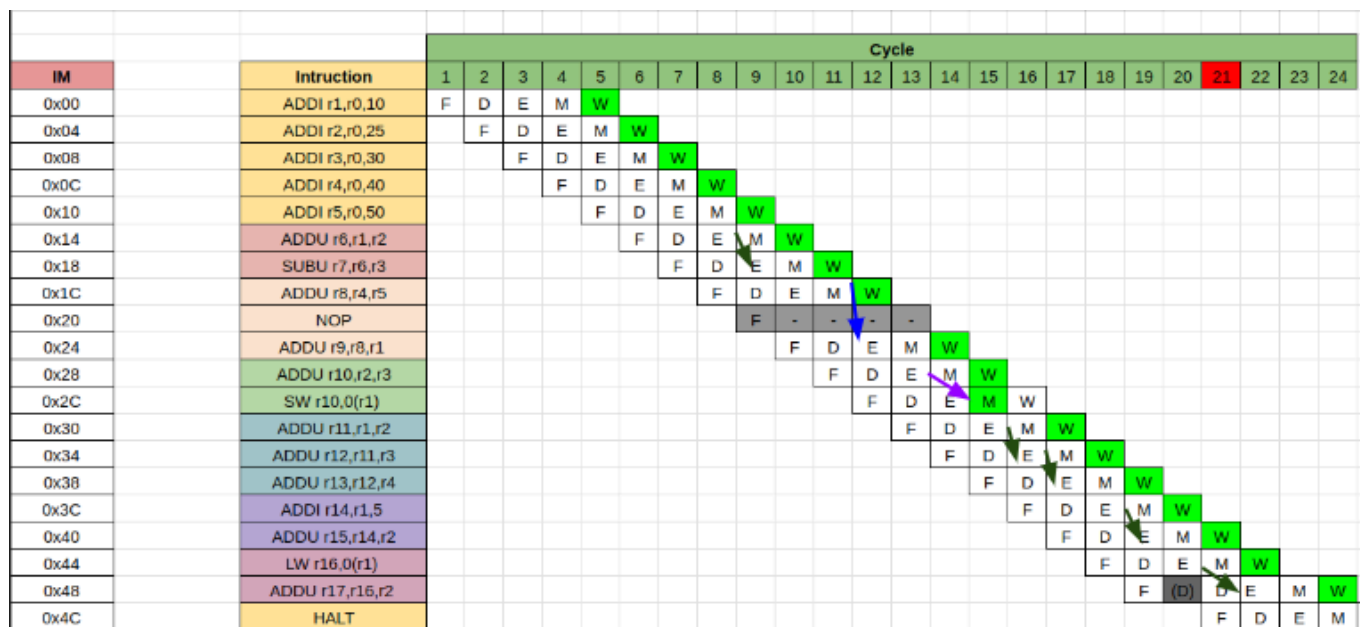
Notar el stall en IF de la instrucción posterior al salto condicional

Programa 2

Propósito: Poner a prueba todos los tipos de forwarding soportados por el diseño

```

ADDI r1,r0,10
ADDI r2,r0,25
ADDI r3,r0,30
ADDI r4,r0,40
ADDI r5,r0,50
# EX/MEM to EX Forwarding
ADDU r6,r1,r2    # r6 = 10 + 25 = 35
SUBU r7,r6,r3    # Forward r6 from EX/MEM, r7 = 35 - 30 = 5
# MEM/WB to EX Forwarding
ADDU r8,r4,r5    # r8 = 40 + 50 = 90
NOP              # No operation, creates a cycle gap
ADDU r9,r8,r1    # Forward r8 from MEM/WB, r9 = 90 + 10 = 100
# EX/MEM to MEM Forwarding (for store instructions)
ADDU r10,r2,r3   # r10 = 25 + 30 = 55
SW r10,0(r1)     # Store r10 to memory address in r1, forward r10 from EX/MEM
# Multiple forwarding in a sequence
ADDU r11,r1,r2   # r11 = 10 + 25 = 35
ADDU r12,r11,r3  # Forward r11 from EX/MEM, r12 = 30 + 30 = 60
ADDU r13,r12,r4  # Forward r12 from EX/MEM, r13 = 60 + 40 = 100
# Forwarding with immediate values
ADDI r14,r1,5    # r14 = 10 + 5 = 15
ADDU r15,r14,r2  # Forward r14 from EX/MEM, r15 = 15 + 20 = 35
# Load-use hazard with forwarding
LW r16,0(r1)     # Load value from memory address in r1 to r16
ADDU r17,r16,r2  # Forward r16 from MEM/WB (after stall), r17 = mem[r1] + 25
HALT
  
```



Programa 3

ADDI r13,r13,0x20

LUI r11,10

JAL 5 # salta a la dirección de 5 y guarda la dir de retorno (PC + 1) en r31

ADDI r7,r7,7

5: ADDI r9,r9,9

ADDI r2,r2,2

JALR r31,r13 # salta a la dirección en r13 y guarda la dir de retorno (PC + 1) en r31





ADDI r4,r4,4

ADDI r6,r6,6

HALT

IM	Instruction	Cycle																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0x00	ADDI r13,r13,0x20	F	D	E	M	W												
0x04	LUI r11,10		F	D	E	M	W											
0x08	JAL 7			F	D	E	M	W										
0x0C	ADDI r7,r7,7				F	-	-	-	-									
0x10	7: ADDI r9,r9,9					F	D	E	M	W								
0x14	ADDI r2,r2,2						F	D	E	M	W							
0x18	JALR r13,r16							F	D	E	M	W						
0x1C	ADDI r4,r4,4								F	-	-	-	-					
0x20	ADDI r6,r6,0x2c									F	D	E	M	W				
0x24	JR r6										F	D	E	M	W			
0x28	ADDI r7,r6,7											F	-	-	-	-		
0x2C	HALT												F	D	E	M	W	

Bibliografía y recursos

- [Pipeline](#)
-  Lecture 26 - Forwarding
-  Lecture 27 - Stalling, Control Hazards
-  Timing report and RTL schematic interpretation
-  65 - Generating Different Clocks Using Vivado's Clocking Wizard
- Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design MIPS Edition: The Hardware/Software Interface. Morgan Kaufmann.
- Hennessy, J. L., & Patterson, D. A. (2011). Computer Architecture: A Quantitative Approach. Morgan Kaufmann.
- Kane, G., & Heinrich, J. (1992). MIPS RISC Architecture. Prentice Hall.
- Sweetman, D. (2007). See MIPS Run. Morgan Kaufmann.