



UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS FÍSICAS Y NATURALES

INFORME DE TRABAJO PRACTICO

Arquitectura de Computadoras

Estudiante:

Cabrera, Augusto Gabriel

Arquitectura de Computadoras

Córdoba,
12 de abril de 2025

Índice

1. Requerimientos	4
2. Implementación	5
2.1. Composición del Módulo MIPS	5
2.1.1. IF (Instruction Fetch)	6
2.1.2. ID (Instruction Decode)	7
2.1.3. EX (Execute)	10
2.1.4. MEM (Memory Access)	12
2.1.5. WB (Write Back)	13
2.1.6. Unidad de Cortocircuito (Forwarding)	14
2.1.7. Hazard (Hazard Unit)	14
2.2. UART	15
2.2.1. Proceso de RX y TX de Datos	15
2.3. Requisitos de Timing y Clock Wizard	16
2.3.1. Caso 60 MHz: incumplimiento de las restricciones de timing	16
2.3.2. Caso optimo de cumplimiento de las restricciones de timing	17
2.3.3. Clock Wizard	19
2.4. Unidad de Debug	20
2.4.1. Componentes Principales	20
2.4.2. Maquina de estados	20
2.4.3. STATE_IDLE	21
2.4.4. STATE_LOAD_INSTR	21
2.4.5. STATE_DEBUG_MODE	22
2.4.6. STATE_CONTINOUS_MODE	22
2.4.7. Envío de registros internos por UART	22
2.4.8. Señales de comandos por UART	22
2.4.9. Interfaz Gráfica	23
3. Estructura Evolutiva del README	25

Prólogo

Este informe presenta una detallada implementación del *pipeline* del procesador MIPS, diseñado para ilustrar las etapas clave en la ejecución de instrucciones dentro de la arquitectura de computadoras. Se describirán las cinco etapas del *pipeline*: **Instruction Fetch (IF)**, **Instruction Decode (ID)**, **Execute (EX)**, **Memory Access (MEM)** y **Write Back (WB)**, así como los riesgos asociados, como los estructurales, de datos y de control, junto con las estrategias para mitigarlos.

Adicionalmente, se abordarán los requerimientos técnicos para la implementación del sistema. Es importante destacar la inclusión de una unidad de *debug* que permite la comunicación a través de *UART*. Esta unidad enviará información crucial a una PC, como el contenido de los registros, los *latches* intermedios y la memoria de datos utilizada, facilitando el monitoreo y la depuración del sistema.

Este informe tiene como objetivo no solo explicar el funcionamiento del *pipeline* en el procesador MIPS, sino también resaltar la comunicación y el monitoreo en tiempo real a través de la *UART* en el desarrollo de este MIPS.

1. Requerimientos

- Implementar un Procesador MIPS segmentado con 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	I-type	J-type
SLL, SRL, SRA	LB, LH, LW, LWU, LBU, LHU	JR, JALR
SLLV, SRLV, SRAV	SB, SH, SW	
ADDU, SUBU	ADDI, ADDIU	
AND, OR, XOR, NOR	ANDI, ORI, XORI, LUI	
SLT, SLTU	SLTI, SLTIU	
	BEQ, BNE	
	J, JAL	

- 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. Se debe mantener el orden estricto de lecturas y escrituras.
 - **De control:** ocurren al tomar decisiones sobre una condición todavía no evaluada.
- Para dar soporte a los riesgos anteriores, se debe implementar:
 - Unidad de Cortocircuitos (*Forward*).
 - Unidad de Detección de Riesgos (*Hazard*).
- El programa a ejecutar debe ser cargado en la memoria del programa mediante un archivo ensamblado:
 - Debe implementarse un programa ensamblador que convierta código assembler MIPS a código de instrucción.
 - El programa debe transmitirse mediante una interfaz UART antes de comenzar a ejecutar.
- Se debe simular una Unidad de Debug que se comunice con el procesador mediante UART. Esta unidad debe enviar a la PC:
 - El contenido de los 32 registros.
 - El contenido de los registros intermedios (*latches*).

- El contenido de la memoria de datos utilizada.
- Antes de iniciar la ejecución, el procesador debe esperar la recepción de un programa mediante la Unidad de Debug, y debe permitir dos modos de operación:
 - **Modo continuo:** al recibir un comando por UART, la FPGA inicia la ejecución del programa hasta la instrucción HALT. Al finalizar, se muestran el ultimo estado.
 - **Modo paso a paso:** mediante comandos UART se ejecuta un ciclo de clock por paso. A cada paso, se muestran los valores indicados.

2. 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**

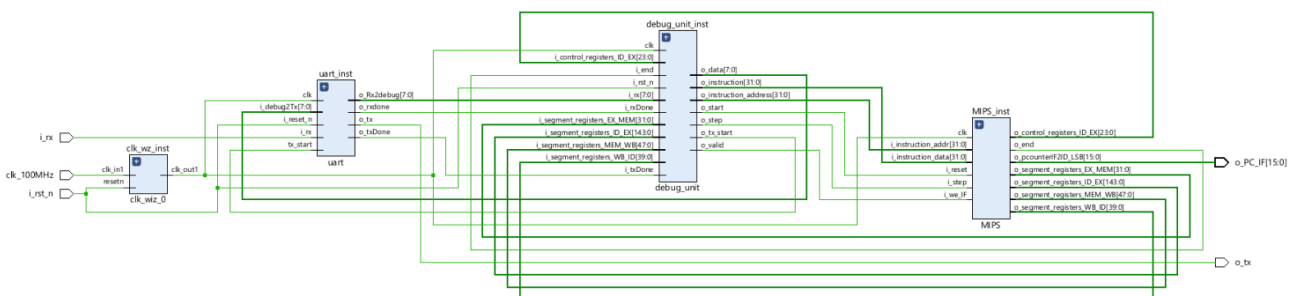


Figura 1: Vista general del sistema: conexión entre MIPS, Debug_Unit, UART y generador de clock.

La figura 1 ilustra la conexión entre los submódulos, integrados en el archivo principal `top.v`, donde se observa la interacción entre el procesador, la unidad de debug y la interfaz UART.

2.1. Composición del Módulo MIPS

El núcleo del procesador desarrollado se encapsula en el módulo MIPS, el cual integra todas las unidades funcionales necesarias para implementar un pipeline segmentado de cinco etapas. Su objetivo es coordinar la ejecución secuencial de instrucciones MIPS mediante el procesamiento en paralelo de diferentes etapas, mejorando así el rendimiento y la eficiencia del sistema.

Cada etapa del pipeline contiene internamente su correspondiente registro de segmento, responsable de mantener la información entre ciclos de reloj. Esta organización facilita la depuración y permite una clara segmentación del flujo de datos.

Además, el diseño incluye:

- **Unidad de Detección de Riesgos (Hazard):** Identifica conflictos de lectura/escritura entre instrucciones activas en distintas etapas del pipeline, e implementa mecanismos de detención (`stall`) para preservar la coherencia de los datos.

- **Unidad de Cortocircuito (Forward):** Implementa **bypassing** de datos para minimizar stalls innecesarios al reenviar resultados de etapas posteriores a anteriores.
- **Interfaz UART y Debug:** El módulo MIPS expone diversas señales internas que permiten monitorear el estado del sistema en tiempo real, incluyendo registros intermedios entre etapas (*pipeline registers*), contenido de registros y accesos a memoria. Esta información se transmite a través de la UART para depuración externa.

El módulo MIPS se divide funcionalmente en los siguientes submódulos:

2.1.1. IF (Instruction Fetch)

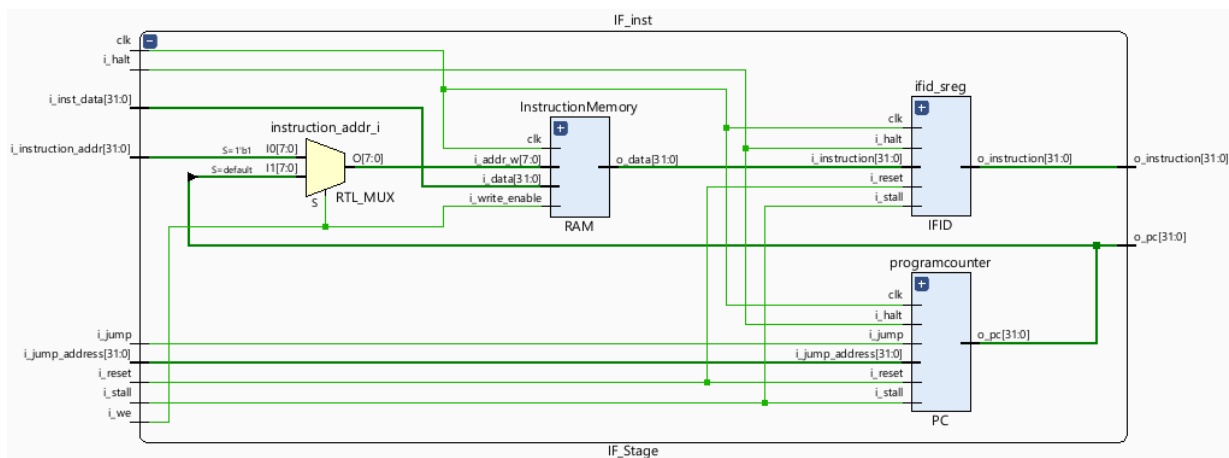


Figura 2: Estructura interna del módulo IF_Stage.

La etapa de **Instruction Fetch (IF)** es la encargada de obtener la próxima instrucción a ejecutar desde la memoria de programa. Esta etapa determina el valor del contador de programa (PC), lo actualiza en función del flujo normal o de un salto, y obtiene la instrucción correspondiente desde la memoria de instrucciones.

El módulo IF_Stage encapsula tres componentes principales:

- **PC (Program Counter):** Calcula la dirección de la próxima instrucción. Si la señal *i_jump* está activa, carga el valor de *i_jump_address*. En caso contrario, simplemente incrementa el contador en 4. Si se activa *i_halt* o *i_stall*, el contador se mantiene congelado.
- **RAM de Instrucciones:** Es una memoria implementada mediante un módulo RAM parametrizable, diseñada como una memoria asincrónica de un solo puerto y modelada internamente como un arreglo de 8 bits. Esta estructura es común en implementaciones de FPGA Xilinx.

La memoria permite tanto la escritura como la lectura de instrucciones de 32 bits. En modo escritura (cuando *i_we* está habilitado), el dato ingresado se divide en cuatro segmentos de 8 bits que se almacenan en direcciones contiguas (*i_addr_w*, *i_addr_w+1*, etc.).

Durante la ejecución normal del programa, el valor actual del contador de programa (PC) se utiliza como dirección de lectura, recuperando los cuatro bytes correspondientes y concatenándolos para formar la instrucción completa. Esto se realiza de forma asincrónica, permitiendo una respuesta inmediata al cambio de dirección.

```

1 assign o_data = {memory[i_addr_w], memory[i_addr_w + 1],
2             memory[i_addr_w + 2], memory[i_addr_w + 3]};

```

Listing 1: Lectura de datos desde la memoria RAM

- **IF/ID Register:** Este latch registra la instrucción obtenida en la etapa IF para ser procesada por la etapa siguiente (ID). Su funcionamiento está condicionado por las señales `i_stall` y `i_halt`, que impiden su actualización si están activas.

El flujo de esta etapa garantiza que la instrucción correspondiente al valor actual del PC sea correctamente propagada a la etapa de decodificación, preservando la secuencia del programa y permitiendo el soporte de instrucciones de salto. La integración del registro IF/ID dentro de la etapa contribuye a una mejor modularidad y facilita la depuración del pipeline.

2.1.2. ID (Instruction Decode)

Decodifica la instrucción obtenida, lee los registros fuente y genera las señales de control necesarias para las etapas siguientes. También almacena toda esta información en el registro de segmento ID/EX, que será utilizado por la unidad de ejecución. Este registro incluye datos de operandos, códigos de operación y señales de control.

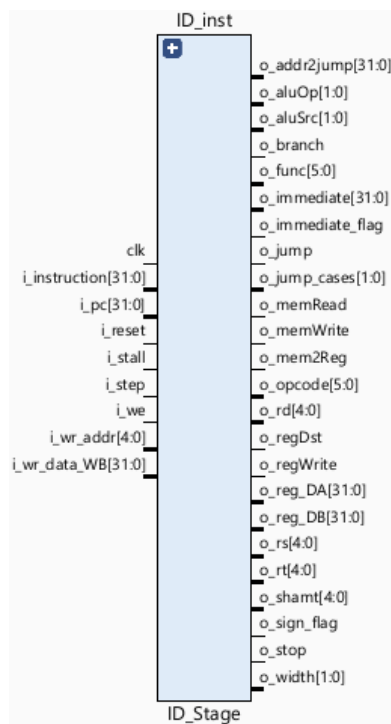


Figura 3: Señales del módulo ID_Stage.

Formatos de Instrucción en MIPS

Las instrucciones en MIPS tienen un tamaño fijo de 32 bits y se dividen en tres formatos principales:

- **Tipo R (Register):** usado para operaciones aritméticas y lógicas entre registros.
- **Tipo I (Immediate):** usado para operaciones que utilizan constantes (inmediatos) o acceso a memoria.

- **Tipo J (Jump):** usado para saltos incondicionales.

Formato Tipo R

Campo	op	rs	rt	rd	shamt	funct
Bits	6	5	5	5	5	6

Este formato se utiliza para instrucciones como **add**, **sub**, **and**, **or**, **sll**, etc.

- **op:** código de operación (generalmente 0 para instrucciones R).
- **rs, rt:** registros fuente.
- **rd:** registro destino.
- **shamt:** cantidad de desplazamiento (para instrucciones shift).
- **funct:** especifica la operación exacta (como **add** o **sub**).

Formato Tipo I

Campo	op	rs, rt	inmediato
Bits	6	5, 5	16

Usado en instrucciones como **lw**, **sw**, **beq**, **addi**, etc.

- **op:** código de operación.
- **rs:** registro fuente.
- **rt:** registro destino.
- **inmediato:** constante o desplazamiento para memoria/saltos.

Formato Tipo J

Campo	op	dirección
Bits	6	26

Usado para instrucciones como **j** (jump) y **jal** (jump and link).

- **op:** código de operación.
- **dirección:** dirección de destino (relativa a PC).

Componentes principales

- **Banco de registros (Registers):** Implementa 32 registros de 32 bits. Permite lectura de dos registros fuente y escritura condicional. La escritura se realiza en flanco de bajada, lo que evita conflictos con la lectura en flanco de subida. Los registros \$0 a \$31 están organizados según la convención estándar de MIPS (**\$zero**, **\$t0**, **\$s0**, etc.).

- **Unidad de Control (Control):** Interpreta los campos `opcode` y `funct` de la instrucción para generar todas las señales de control necesarias: selección de destino, tipo de operación ALU, lectura/escritura de memoria, extensión de inmediato, tipo de salto, entre otras. Se contemplan instrucciones tipo R, I y J, incluyendo variantes como JAL, JR, LUI, etc.

Type	Jump	Branch	RegDst	ALUSrc	M2R	RegWrite	MemRead	MemWrite	ALUOp	Width	Sign	Imm
R	0	0	0	0	0	1	0	0	10	X	0	0
JR	1	0	X	0	1	0	0	0	10	X	0	0
JARL	1	0	0	0	0	1	0	0	0	X	0	0
LW	0	0	1	1	1	1	1	0	0	10	0	1
SW	0	0	X	1	X	0	0	1	0	10	0	1
BEQ	0	1	X	0	X	0	0	0	1	X	0	1
BNE	0	1	X	0	X	0	0	0	1	X	0	1
ADDI	0	0	1	1	0	1	0	0	11	X	0	1
ADDIU	0	0	1	1	0	1	0	0	11	X	1	1
ORI	0	0	1	1	0	1	0	0	11	X	0	1
ANDI	0	0	1	1	0	1	0	0	11	X	0	1
XORI	0	0	1	1	0	1	0	0	11	X	0	1
LUI	0	0	1	1	0	1	0	0	11	X	1	1
STLI	0	0	1	1	0	1	0	0	11	X	0	1
STLIU	0	0	1	1	0	1	0	0	11	X	1	1
J	1	0	X	X	X	0	0	0	0	X	X	0

Figura 4: Tabla de señales de control para cada instrucción MIPS.

Las señales generadas se pueden clasificar de la siguiente forma:

- **Señales de control para la ALU:**
 - `o_aluOp` [1:0]: Código que define la operación a realizar por la ALU (00: suma, 01: resta, 10: función R-type, 11: operaciones lógicas/SLT).
 - `o_aluSrc` [1:0]: Selección del segundo operando para la ALU (registro o inmediato).
- **Señales de salto y ramificación:**
 - `o_branch`: Habilita la evaluación de salto condicional.
 - `o_jump`: Indica si se debe realizar un salto incondicional.
- **Señales para el acceso a memoria:**
 - `o_memRead`: Habilita lectura desde memoria.
 - `o_memWrite`: Habilita escritura en memoria.

- **o_width [1:0]:** Define el tamaño del acceso a memoria (00: byte, 01: half-word, 10: word).
- **o_sign_flag:** Define si la lectura es con o sin signo.
- **Señales de escritura en registros:**
 - **o_regWrite:** Habilita la escritura en el banco de registros.
 - **o_regDst:** Selecciona el campo **rd** o **rt** como destino.
 - **o_mem2Reg:** Selecciona la fuente de datos para escritura en el registro (ALU o Memoria).
- **Señal adicional:**
 - **o_immediate:** Indica si se debe usar el campo inmediato (para extenderlo y usarlo como operando).
- **Extensión de inmediato (SignExtension):** Convierte el campo inmediato de 16 bits en un valor de 32 bits, realizando extensión con signo o sin signo según la instrucción.
- **Lógica de salto:** En instrucciones BEQ, BNE, JR, JAL, etc., se evalúan condiciones y se calcula la dirección de destino en caso de cumplirse el salto. La lógica identifica si se trata de un salto absoluto, relativo o tipo **link**, y emite señales correspondientes como **o_jump** y **o_addr2jump**.
- **Registro de segmento ID/EX:** Almacena los datos de operandos, identificadores de registros, campo inmediato, señales de control y demás información relevante que será utilizada por la siguiente etapa. En caso de detección de **stall** o **step**, este registro detiene su actualización o fuerza su vaciado.

Consideraciones

- Se implementan mecanismos para detectar instrucciones especiales como JAL y JALR, generando ajustes adicionales en los operandos.
- El diseño está preparado para trabajar con señales de control externas como **stall** y **step**, lo que permite una ejecución controlada o pausada del pipeline.

2.1.3. EX (Execute)

Realiza operaciones aritméticas y lógicas, así como el cálculo de direcciones para operaciones de memoria. Incluye lógica de forwarding y selección de operandos. Al finalizar esta etapa, se almacenan los resultados y señales de control necesarias en el registro de segmento EX/MEM, el cual será consumido por la siguiente etapa.

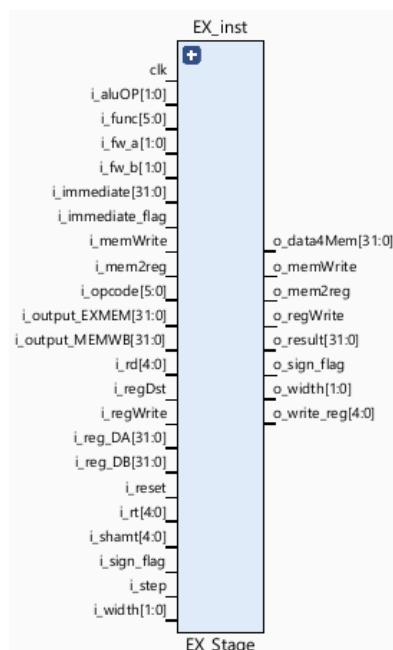


Figura 5: Señales del módulo EX_Stage.

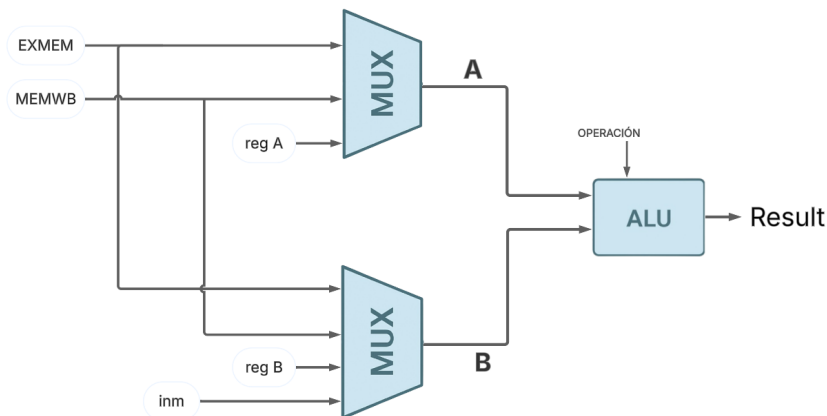


Figura 6: Operandos para la ALU con Forwarding.

Resumen de funciones:

- Selección del tipo de operación a ejecutar (por tipo R/I o carga/almacenamiento).
- Selección de operandos con forwarding y/o inmediato.
- Ejecución en la ALU.
- Determinación del registro destino (rd o rt).
- Propagación de señales a la etapa MEM.

Selección de operación ALU según tipo de instrucción:

Operación ALU	Descripción
LOAD STORE	Para instrucciones de carga y almacenamiento (como LW, SW), se usa una suma (ADD) para calcular la dirección efectiva.
BRANCH	En instrucciones de salto condicional (como BEQ, BNE), no se necesita realizar una operación aritmética significativa, por lo que se asigna una operación nula (IDLE).
R_TYPE	En instrucciones tipo R, el campo funct especifica la operación ALU a ejecutar.
I_TYPE	En instrucciones tipo I, la operación ALU se deduce directamente del opcode.

Cuadro 1: Decodificación de tipo de operación para la ALU

Salidas principales:

- **o_result**: resultado de la ALU.
- **o_data4Mem**: valor para escritura en memoria (store).
- **o_write_reg**: dirección del registro a escribir.
- Señales de control: **o_mem2reg**, **o_memWrite**, **o_regWrite**, **o_width**, **o_sign_flag**.

2.1.4. MEM (Memory Access)

Se encarga de realizar las operaciones de acceso a memoria necesarias para las instrucciones tipo **load** y **store**. En esta etapa se evalúan las señales de control relacionadas con la memoria, se enmascan correctamente los datos según el ancho y signo, y se escriben/leen los datos correspondientes. Además, esta etapa prepara los valores necesarios para la siguiente etapa WB (Write Back).

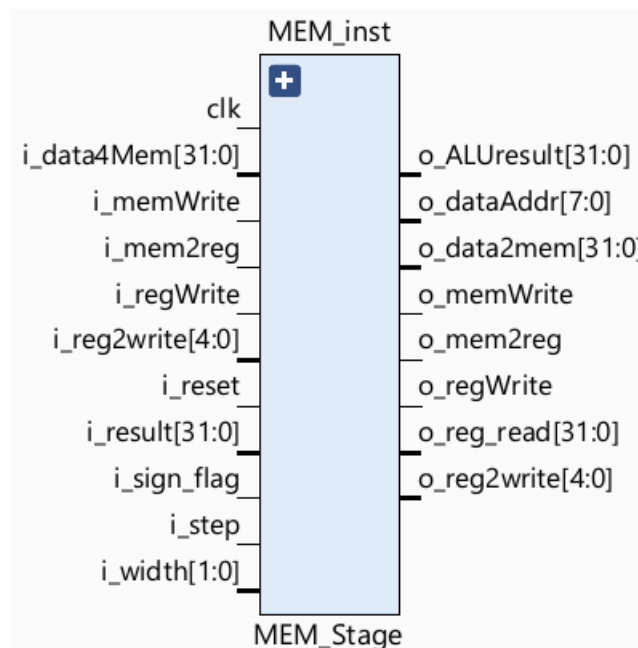


Figura 7: Señales del módulo MEM_Stage.

Funciones principales:

- Leer datos desde memoria si la instrucción lo requiere (load).
- Escribir datos en memoria cuando se detecta una instrucción store.
- Alinear y extender los datos según el ancho de acceso (**i_width**) y el signo (**i_sign_flag**).
- Propagar los resultados y control hacia la etapa WB a través del registro MEMWB.

Entradas relevantes:

- **i_result**: Dirección en memoria para acceso (proviene de la ALU).
- **i_data4Mem**: Dato a escribir en memoria (si aplica).
- **i_width**, **i_sign_flag**: Controlan tipo y formato de acceso.

- **i_memWrite, i_mem2reg, i_regWrite:** Señales de control de escritura, origen del write-back y habilitación del mismo.

Salidas relevantes:

- **o_data2mem:** Dato preparado y alineado para escritura en memoria.
- **o_dataAddr:** Dirección de acceso a memoria.
- **o_memWrite:** Habilitación de escritura en memoria.
- **o_reg_read:** Dato leído de memoria (ya alineado y con extensión correcta).
- **o_ALUresult:** Resultado de la operación ALU propagado desde la etapa EX.

Esta etapa también contiene una instancia de la memoria RAM, la cual interactúa directamente con las señales de control y datos preparados. Finalmente, los resultados se almacenan en el registro MEMWB para ser utilizados en la etapa de **Write Back**.

2.1.5. WB (Write Back)

La etapa de **Write Back (WB)** es la última en el pipeline del procesador y se encarga de decidir qué dato debe escribirse en el banco de registros como resultado de la instrucción. Este dato puede ser el valor leído desde memoria (en instrucciones **load**) o el resultado de la ALU (para operaciones aritméticas/lógicas).

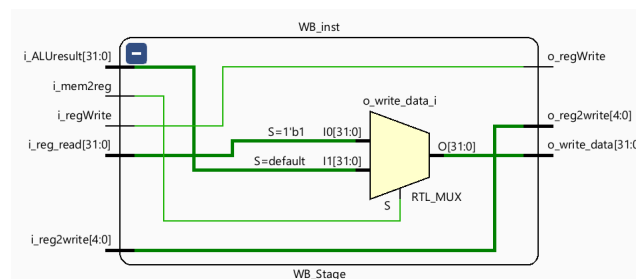


Figura 8: Señales del módulo WB_Stage.

▪ Entradas:

- **i_reg_read:** Dato proveniente de la memoria (load).
- **i_ALUresult:** Resultado de la ALU.
- **i_reg2write:** Registro destino (rd o rt).
- **i_mem2reg:** Señal de control que indica si se debe escribir lo leído de memoria o el resultado de la ALU.
- **i_regWrite:** Habilita la escritura en el banco de registros.

▪ Salidas:

- **o_write_data:** Dato que finalmente se escribirá en el banco de registros.
- **o_reg2write:** Número de registro donde se escribirá el dato.
- **o_regWrite:** Señal de control que habilita la escritura.

La lógica del módulo es simple:

- Si `i_mem2reg = 1`, se escribe el valor de memoria.
- Si `i_mem2reg = 0`, se escribe el valor de la ALU.

2.1.6. Unidad de Cortocircuito (Forwarding)

Para evitar *data hazards* en un procesador con arquitectura *pipeline*. Esta técnica permite adelantar resultados de instrucciones aún no escritas al banco de registros, evitando así el uso de *stalls* innecesarios.

▪ Entradas:

- `i_rs_IFID`, `i_rt_IFID`: Registros fuente de la instrucción en la etapa ID (decodificación).
- `i_rd_IDEX`, `i_rd_EX_MEMWB`: Registros destino de las instrucciones en etapas posteriores (WB y MEM).
- `i_wr_WB`, `i_wr_MEM`: Flags de escritura para indicar si esas etapas efectivamente escribirán en su registro destino.

▪ Salidas:

- `o_fw_a`, `o_fw_b`: Señales de control de forwarding para los operandos A (rs) y B (rt), codificadas de la siguiente manera:
 - 00: No se requiere forwarding.
 - 10: Forwarding desde la etapa MEM.
 - 11: Forwarding desde la etapa WB.

De esta manera, se evita el riesgo de dependencia de datos al reenviar resultados directamente desde las etapas MEM o WB, mejorando el rendimiento del pipeline sin introducir retardos adicionales.

2.1.7. Hazard (Hazard Unit)

Este módulo implementa la lógica para detectar riesgos (*hazards*) de datos y de control en un procesador con arquitectura *pipeline*. Su salida es una señal de control `o_stall` que se activa cuando es necesario frenar temporalmente el avance del *pipeline* para evitar comportamientos incorrectos debido a dependencias entre instrucciones.

- **Riesgos de Datos (Load-Use):** Si una instrucción en la etapa EX está realizando una carga desde memoria (`i_ID_EX_MemRead = 1`) y la siguiente instrucción (en etapa ID) depende del resultado de esa carga (usa como operandos los registros `Rs` o `Rt` que coinciden con el destino de la EX), entonces se genera un *stall*.
- **Riesgos de Control (Saltos Condicionales y JR/JALR):**
 - Si la instrucción actual es una instrucción de salto condicional como BEQ o BNE (`i_jumpType = 2'b01`) y alguno de los registros fuente (`Rs` o `Rt`) está siendo actualizado por una instrucción que aún no ha llegado a la etapa WB, se genera un *stall*.
 - Si es una instrucción de salto indirecto tipo JR o JALR (`i_jumpType = 2'b10`), se inspecciona únicamente el registro `Rs`. Si alguna instrucción posterior lo está escribiendo, o si proviene de una carga, también se activa el *stall*.

Este módulo es esencial para garantizar la correcta ejecución de instrucciones secuenciales en un pipeline sin introducir resultados incorrectos por dependencias no resueltas.

2.2. UART

Para la implementación del UART, se utilizó una versión similar a la desarrollada en el TP2, pero se descartaron las colas FIFO tanto para transmisión (TX) como para recepción (RX).

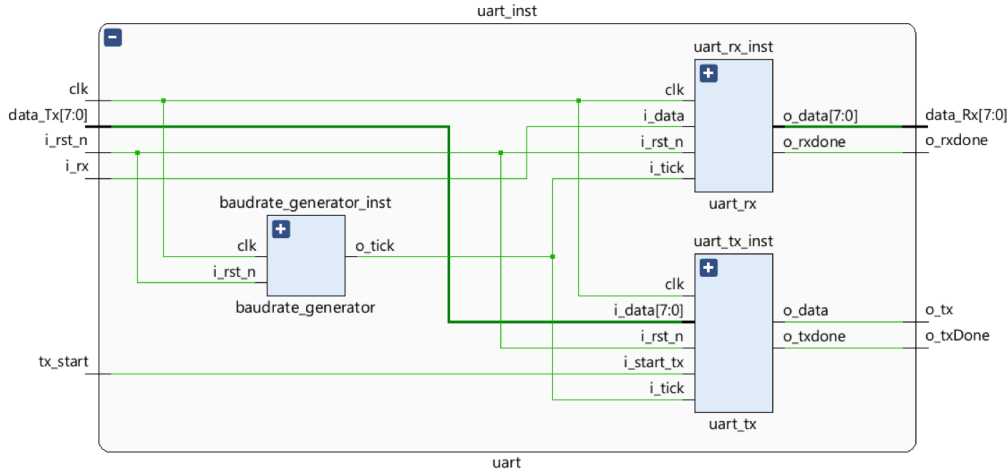


Figura 9: Esquema general de la UART

2.2.1. Proceso de RX y TX de Datos

Suponiendo un total de **N bits de datos** y **M bits de stop**, la transmisión y recepción de datos sigue el siguiente procedimiento:

1. **IDLE**: El sistema permanece en espera mientras la línea de entrada esté en 1. Una transición a 0 indica el inicio del bit de start. En ese momento se inicia el contador de ticks.
2. **START**: Cuando el contador alcanza el valor 7, la señal se encuentra en la mitad del bit de start. Se reinicia el contador.
3. **RECEIVE / DATA**: Al llegar a 15, la señal está en la mitad del primer bit de datos. Se captura ese bit y se desplaza a un registro. Luego se reinicia el contador.
4. Se repite el paso anterior un total de $N - 1$ veces para recibir los bits restantes del dato.
5. Si se utiliza bit de paridad, se repite nuevamente el paso anterior.
6. **STOP**: Se repite el mismo proceso para los M bits de stop.

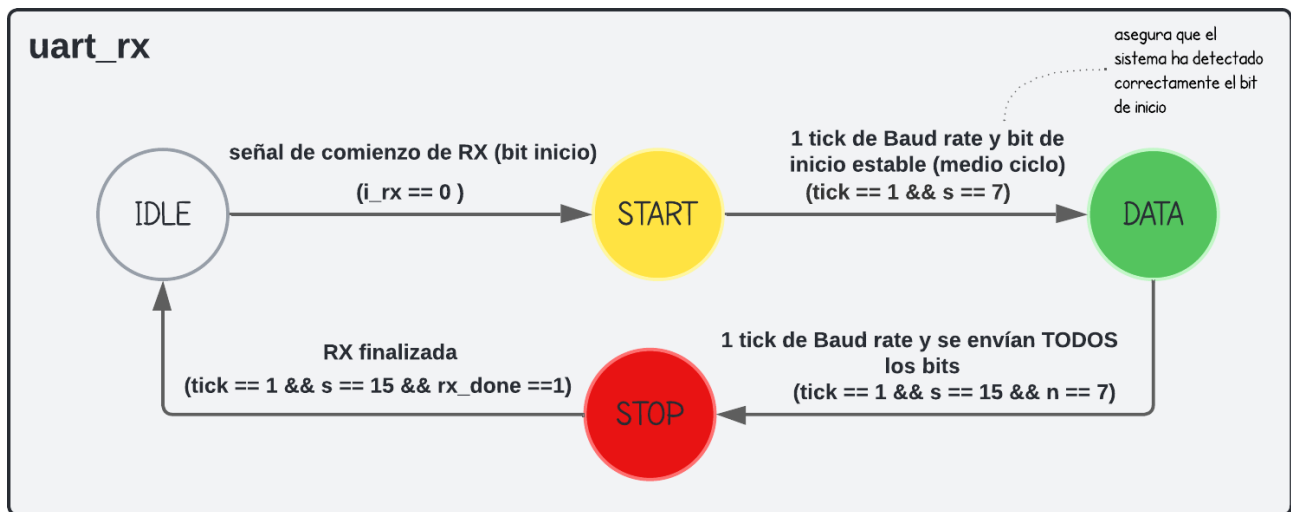


Figura 10: Máquina de estados para recepción de datos

Nota: Se utiliza la misma máquina de estados para la transmisión en `uart_tx`.

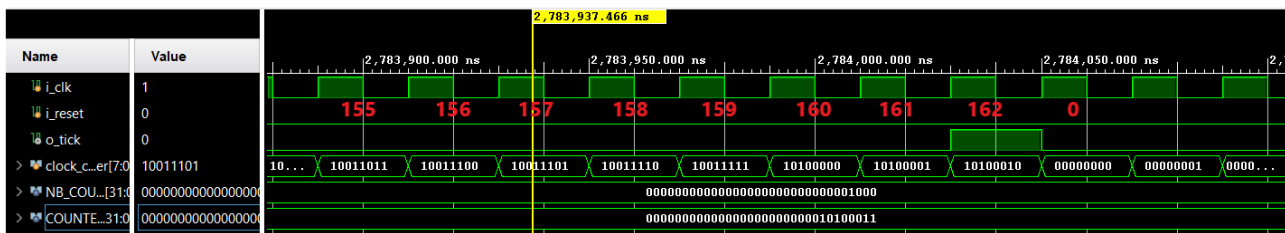


Figura 11: Configuración de baudrate: CLK = 50 MHz, BR = 19200

2.3. Requisitos de Timing y Clock Wizard

Uno de los desafíos principales al integrar el procesador MIPS en la FPGA fue cumplir con los **requisitos de timing** en todos los *paths* críticos.

2.3.1. Caso 60 MHz: incumplimiento de las restricciones de timing

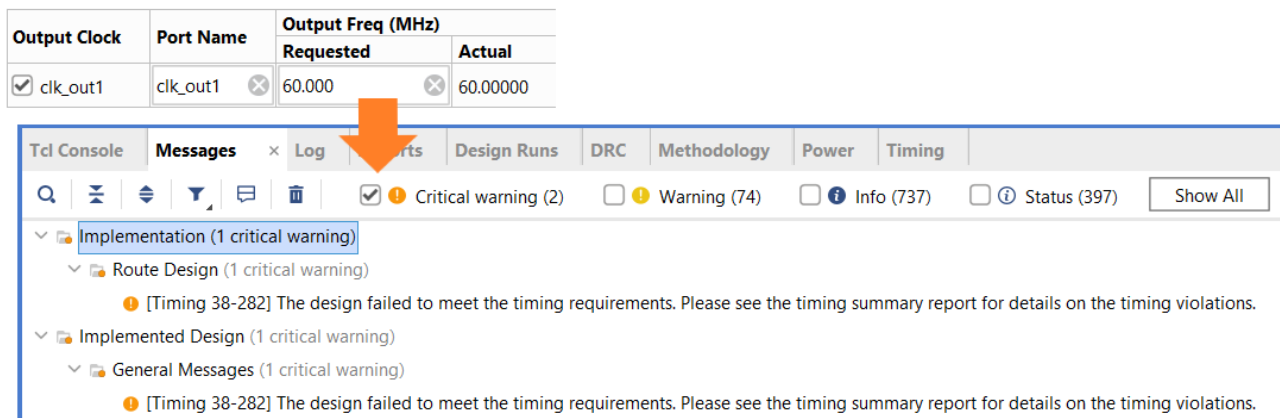


Figura 12: 60 MHz, diseño no cumple las restricciones de timing luego de la implementación.

El análisis de *Design Timing Summary* reportó un **WNS** de **-0.977 ns** y un **TNS** de **-16.309 ns**, afectando a un total de 34 endpoints. Esto indica que existen caminos críticos cuya latencia excede el período de reloj establecido, comprometiendo el cumplimiento del *setup timing*.

Por otro lado, no se observaron violaciones en términos de *hold time* ni de *pulse width*, lo que sugiere que los problemas de temporización están focalizados exclusivamente en los caminos combinacionales más largos del diseño.

Caminos críticos del diseño

El análisis de los caminos intra-clock más críticos del diseño revela que las principales violaciones de *setup timing* ocurren entre registros de la etapa de decodificación del procesador MIPS (**ID_inst**) y registros tanto del módulo de depuración (**debug_unit_inst**) como de la etapa de fetch (**IF_inst**).

El camino más crítico presenta un **slack negativo de -0.977 ns**, con un **retardo total de 9.184 ns**, superando el tiempo requerido de 8.3 ns. Este retardo está compuesto por aproximadamente **3.1 ns de lógica combinacional** y **6.1 ns de retardo de enrutamiento**.

En la Figura 13 se muestra la lista de los 10 caminos más críticos, ordenados por slack. Se observa una repetición consistente del origen en el módulo **ID_inst**, lo cual refuerza la necesidad de optimizar la lógica de salida de dicha etapa.


Name	Slack 	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	-0.977	13	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	debug_unit_inst/tx_data_reg[7]/D	9.184	3.108	6.076	8.3
Path 2	-0.848	13	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	debug_unit_inst/tx_data_reg[4]/D	9.101	3.108	5.993	8.3
Path 3	-0.821	16	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[30]/D	9.108	4.060	5.048	8.3
Path 4	-0.777	13	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	debug_unit_inst/tx_data_reg[2]/D	8.980	3.108	5.872	8.3
Path 5	-0.737	16	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[31]/D	9.024	3.976	5.048	8.3
Path 6	-0.717	16	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[29]/D	9.004	3.956	5.048	8.3
Path 7	-0.705	15	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[26]/D	8.991	3.943	5.048	8.3
Path 8	-0.697	15	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[28]/D	8.983	3.935	5.048	8.3
Path 9	-0.687	12	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	debug_unit_inst/tx_data_reg[3]/D	8.890	2.984	5.906	8.3
Path 10	-0.661	12	34	MIPS_inst/ID_inst...ers_reg[23][5]/C	debug_unit_inst/tx_data_reg[6]/D	8.917	2.984	5.933	8.3

Figura 13: Caminos críticos con violación de *setup timing*.

2.3.2. Caso optimo de cumplimiento de las restricciones de timing

Para resolver esto, se utilizó el **Clock Wizard** para generar una señal de reloj de **50 MHz**, la cual permitió que todos los caminos críticos cumplieran con el tiempo requerido. Como resultado, la métrica **WNS (Worst Negative Slack)** es positiva, lo que significa que incluso el camino más crítico tiene un margen positivo respecto al período del reloj.

Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup											
General Information	Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	
Timer Settings	Path 1	0.562	14	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[0]/D	10.312	3.300	7.012	11.1	
Design Timing Summary	Path 2	0.965	13	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[6]/D	9.909	3.176	6.733	11.1	
Clock Summary (3)	Path 3	1.026	13	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[4]/D	9.897	3.176	6.721	11.1	
Methodology Summary (34)	Path 4	1.063	13	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[3]/D	9.814	3.176	6.638	11.1	
Check Timing (153)	Path 5	1.169	13	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[5]/D	9.705	3.176	6.529	11.1	
Intra-Clock Paths	Path 6	1.425	13	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	debug_unit_inst/bx_data_reg[7]/D	9.450	3.176	6.274	11.1	
clk_100MHz	Path 7	1.446	16	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[30]/D	9.461	4.005	5.456	11.1	
clk_out1_clk_wiz_0	Path 8	1.541	16	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[31]/D	9.366	3.910	5.456	11.1	
Setup 0.562 ns (10)	Path 9	1.557	16	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[29]/D	9.350	3.894	5.456	11.1	
Hold 0.085 ns (10)	Path 10	1.560	15	32	MIPS_inst/ID_inst...ers_reg[11][5]/C	MIPS_inst/IF_inst...r/o_pc_reg[26]/D	9.347	3.891	5.456	11.1	
Pulse Width 10.611 ns (31)											

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT
synth_1	constrs_1	Synthesis Out-of-date												9643
impl_1	constrs_1	Implementation Out-of-date	0.562	0.000	0.085	0.000		0.000	0.204	0	33 Warn			9529
Out-of-Context Module Runs														
clk_wiz_0		Using cached IP results												

Figura 14: Informe de timing con frecuencia de 50 MHz

Se destaca un **WNS (Worst Negative Slack) de 0.562 ns**, lo que indica que el diseño es funcional a una frecuencia de 50 MHz. Además, no se detectaron violaciones de tiempo de *hold* ni rutas fallidas.

El diseño también presenta un bajo consumo estimado de potencia de apenas **0.204 W**, y utiliza aproximadamente **9500 LUTs**, lo cual es aceptable para un diseño de este tamaño.

Observación importante

Se realizaron pruebas adicionales para verificar el comportamiento del diseño a distintas frecuencias. Los resultados obtenidos muestran que el sistema funciona correctamente para frecuencias **menores o iguales a 50 MHz**, pero al superar dicho valor, el **reporte de timing presenta violaciones**, lo que compromete el correcto funcionamiento del procesador.

Por esta razón, se considera **50 MHz como la frecuencia máxima estable** para este diseño.

2.3.3. Clock Wizard

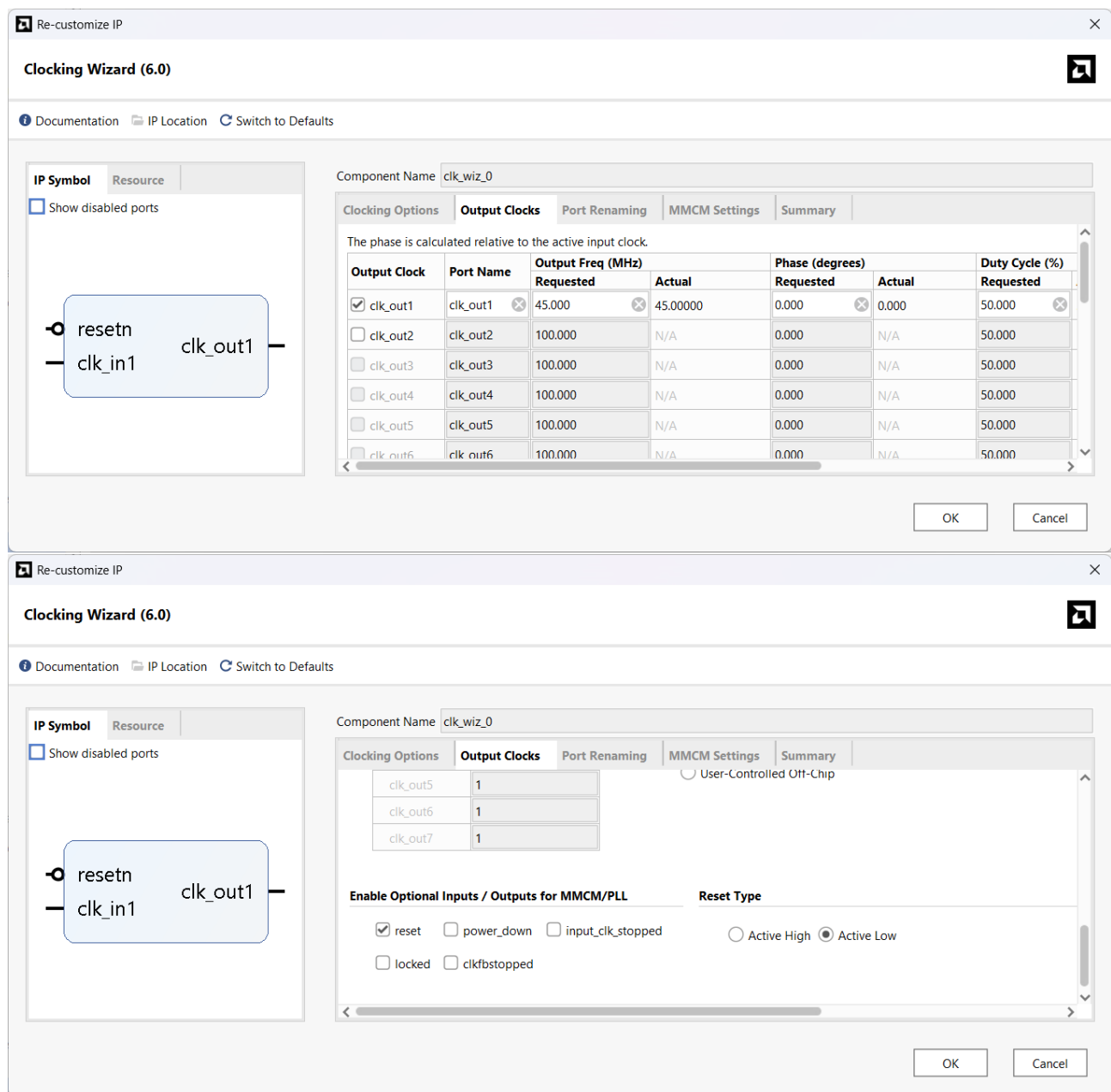


Figura 15: Clock Wizard - 45 MHz

2.4. Unidad de Debug

El módulo `debug_unit.v` es una unidad de control para debuggear el procesador. Su objetivo principal es permitir el control y monitoreo del flujo de instrucciones a través del pipeline del procesador. Esto se logra mediante la recepción de comandos desde un módulo UART (desarrollado en python) y la transmisión de información de diferentes registros internos. Además, permite cargar instrucciones manualmente, ejecutar paso a paso (debug) o en modo continuo, y visualizar los registros en cada etapa del pipeline.

2.4.1. Componentes Principales

Entre los componentes más importantes se destacan:

■ Entradas:

- `clk` y `i_rst_n`: reloj y reset, fundamentales para la sincronización del sistema.
- `i_rx` e `i_rxDone`: señales provenientes del receptor UART, que permiten recibir datos desde el entorno externo.
- `i_txDone`: indica que una transmisión UART ha finalizado.
- `i_end`: señala el fin del programa.
- `i_segment_registers_*` e `i_control_registers_ID_EX`: registros del procesador provenientes del *pipeline*, que contienen la información necesaria para depurar la ejecución.

■ Salidas:

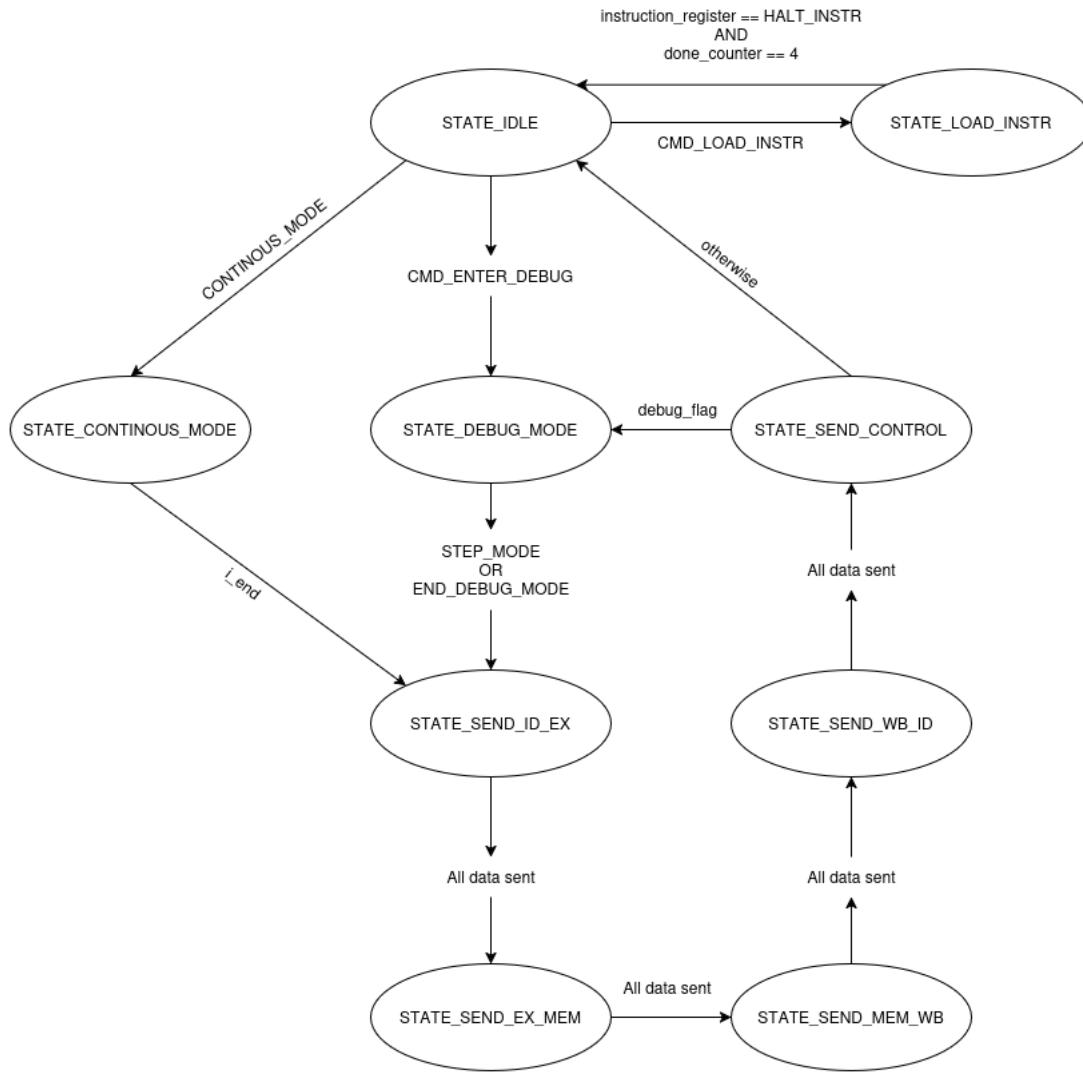
- `o_instruction` y `o_instruction_address`: instrucción recibida y dirección donde debe almacenarse.
- `o_valid`: habilita la escritura de instrucciones.
- `o_step` y `o_start`: señales de control para ejecutar en modo paso a paso o continuo.
- `o_tx_start` y `o_data`: control y datos para transmitir por UART.

■ Registros internos:

- `instruction_register` y `instruction_address`: almacenan temporalmente la instrucción y su dirección de destino.
- `valid`, `step`, `start`, `debug_flag`: registros de control que determinan el comportamiento de la unidad.
- `done_counter` y `tx_data`: permiten gestionar la transmisión de datos por UART de forma secuencial.
- `aux`: utilizado como ayuda para inicializar correctamente la dirección de instrucciones.

2.4.2. Maquina de estados

Este modulo sigue un diseño de maquina de estados finitos (FSM) que cambia de estado en función de los comandos recibidos y las condiciones internas. Cada estado representa una acción clara:

Figura 16: Diagrama de estados del `debug_unit.v`

2.4.3. STATE_IDLE

Este es el estado inicial, espera comandos desde el UART. Según el valor de `i_rx` puede transicionar:

- **STATE_LOAD_INSTR**: Si se recibe `CMD_LOAD_INSTR`.
- **STATE_DEBUG_MODE**: Si se recibe `CMD_ENTER_DEBUG`.
- **STATE_CONTINUOUS_MODE**: Si se recibe `CONTINUOUS_MODE`.

2.4.4. STATE_LOAD_INSTR

En este estado carga instrucciones manualmente de 32 bits, recibéndolas byte por byte de una a la vez, hasta llegar a la instrucción `HALT`.

Se usa un contador (`done_counter`) que incrementa con cada byte recibido (`i_rxDone`), y se van concatenando los bytes en `instruction_register`. Cuando se han recibido los 4 bytes, se habilita la señal `o_valid` y se actualiza la dirección donde se almacenará la instrucción (`instruction_address`), a menos que la instrucción sea la especial de `HALT_INSTR` (`0xFFFFFFFF`) que retornará a `STATE_IDLE`.

2.4.5. STATE_DEBUG_MODE

En este estado se permite el control paso a paso (modo debug). Según el comando recibido:

- Si se llega a `STEP_MODE`, se genera una señal `step` (inversa de `o_step`) y se empieza el envío de registros (`STATE_SEND_ID_EX`).
- Si se llega a `END_DEBUG_MODE` también se envían los registros, pero luego se vuelve al estado de `IDLE`.

Se activa la bandera `debug_flag` para indicar que se esta en modo debug.

2.4.6. STATE_CONTINUOUS_MODE

Se inicia la ejecución continua del programa, activando las señales `o_start` y `o_step`. Se permanece en este estado hasta que `i_end` se activa, lo que indica que el programa finalizó ya que la etapa ID envía la señal de que una instrucción `HALT` es la siguiente. Luego se transiciona a `STATE_SEND_ID_EX`.

2.4.7. Envío de registros internos por UART

Después de ejecutar en cualquier modo (debug o continuo), se envían por UART los registros internos del pipeline del procesador en el siguiente orden de estados:

- `STATE_SEND_ID_EX`
- `STATE_SEND_EX_MEM`
- `STATE_SEND_MEM_WB`
- `STATE_SEND_WB_ID`
- `STATE_SEND_CONTROL`

En cada uno de estos estados, los registros se serializan (de 8 en 8 bits) y se transmiten byte a byte mediante la interfaz UART, utilizando las señales `o_data` y `o_tx_start`. El contador `done_counter` y la señal `i_txDone` permiten controlar cuándo se ha completado la transmisión de un byte y cuándo enviar el siguiente. Después del último estado (`STATE_SEND_CONTROL`), la máquina de estados finita (FSM) toma una decisión sobre el siguiente estado:

- Si la señal `debug_flag` está activa, retorna a `STATE_DEBUG_MODE`.
- En caso contrario, vuelve al estado `STATE_IDLE`.

Este mecanismo permite mantener el modo de debug activo o finalizarlo, según corresponda.

2.4.8. Señales de comandos por UART

El módulo `debug_unit` recibe comandos desde una interfaz Python a través de la UART. Estos comandos se codifican en un solo byte (8 bits) y permiten controlar distintos modos de operación del sistema, como la carga de instrucciones, el debug paso a paso o la ejecución continua del programa. Las señales de comandos están definidas como constantes locales dentro del módulo y se describen a continuación:

- **CMD_LOAD_INSTR (8'b00000001):** Inicia el proceso de carga de una instrucción. El sistema espera recibir 4 bytes consecutivos que conforman una instrucción de 32 bits, los cuales serán almacenados en el registro correspondiente y enviados al procesador.

- **CMD_ENTER_DEBUG (8'b00000010):** Activa el modo de depuración. Una vez en este estado, se pueden enviar otros comandos para ejecutar instrucciones paso a paso o finalizar el proceso de depuración.
- **CONTINUOUS_MODE (8'b00000100):** Inicia la ejecución continua del programa cargado. El módulo deja de recibir comandos y ejecuta instrucciones hasta que se recibe la señal de finalización (**i_end**).
- **STEP_MODE (8'b00001000):** Ejecuta una única instrucción en modo debug y luego envía el contenido de los registros internos (ID/EX, EX/MEM, MEM/WB, WB/ID y señales de control) a través de la UART.
- **END_DEBUG_MODE (8'b00010000):** Finaliza el modo de depuración. Sin embargo, antes de retornar al estado de espera (**STATE_IDLE**), se transmiten los contenidos de los registros mencionados anteriormente, asegurando la visibilidad del estado interno del procesador.

2.4.9. Interfaz Gráfica

Esta interfaz permite al usuario cargar archivos assembler, convertirlos a formato binario, y luego enviar estos programas a la FPGA a través de una conexión UART (puerto serie). Además, proporciona controles para iniciar o finalizar un modo de debug, y visualizar el estado interno del procesador en tiempo real, incluyendo registros intermedios de la arquitectura (como ID/EX, EX/MEM), señales de control, y contenidos de memoria y registros.

- **Carga y conversión de código:** El usuario puede seleccionar un archivo **.asm**, que se convierte automáticamente a **.bin** para su envío a la FPGA.
- **Envío y recepción de datos:** Se puede cargar el binario al procesador, y recibir estructuras de datos codificadas desde la FPGA para monitorear su comportamiento.
- **Comunicación serie:** La interfaz detecta puertos disponibles y se conecta con el puerto UART del sistema embebido.
- **Visualización en tablas:** La interfaz decodifica los datos binarios recibidos y los muestra en tablas que representan registros del procesador, memoria de datos, y señales de control.
- **Modo debug:** El usuario puede activar o finalizar un modo paso a paso para observar con detalle el flujo de ejecución del procesador.

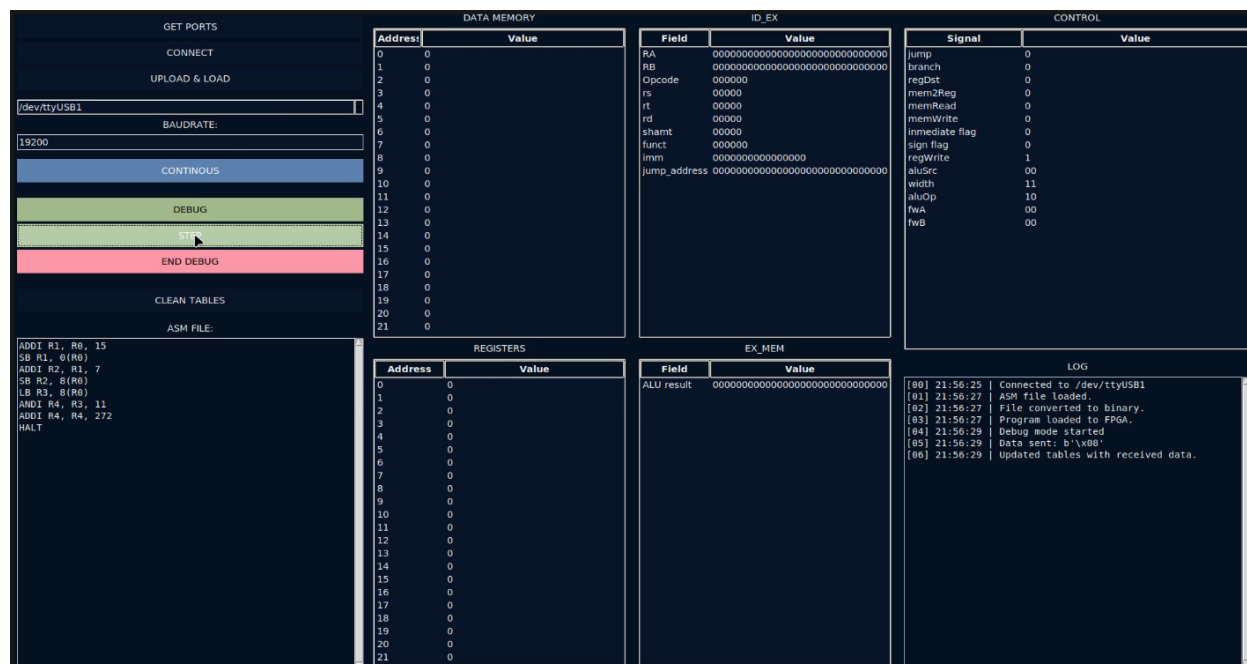


Figura 17: Interfaz gráfica en python.

Un ejemplo de uso de la interfaz puede ser el siguiente:

1. **GET PORTS** para obtener los puertos UART disponibles.
2. **CONNECT** para conectar la FPGA por el puerto serie.
3. **UPLOAD & CONNECT** para cargar un seleccionar un código assembler y cargarlo en la memoria de la FPGA.
4. **DEBUG** enviara la señal de debug a la FPGA.
5. **STEP** una vez ingresado el modo debug podremos ejecutar el código paso a paso hasta llegar al final o terminar la sesión de debug.
6. **END DEBUG** la FPGA recibe la señal de que ya no esta en modo debug.

Algo a tener en cuenta y aclarar es que el modo **CONTINUOUS** ejecutara el código de inicio a fin mostrando la información final sobre las tablas, del ultimo paso en su ejecucion. Este modo no puede ser activado mientras se tenga la flag de debug activa.

3. Estructura Evolutiva del README

Durante el desarrollo del Trabajo Práctico 3 se optó por un enfoque progresivo y basado en desafíos para estructurar la documentación del proyecto. El archivo `README.md` fue concebido no sólo como un manual del usuario o resumen de especificaciones, sino como una **bitácora técnica** que refleja cada etapa de implementación del procesador MIPS segmentado.

A medida que se fueron resolviendo distintas etapas del diseño, se documentaron los problemas enfrentados, las decisiones tomadas, y los resultados obtenidos. Esto permitió:

- Facilitar la comprensión del proceso de diseño completo.
- Trazar una línea de evolución clara desde una versión mínima funcional hasta la implementación final con UART.

Resumen de Avances

- **Avance I – MIPS Only Adder:** Implementación inicial de una ALU capaz de ejecutar `add` sin gestión de riesgos ni saltos.
- **Avance II – Hazard sin Saltos:** Se introducen técnicas de detección y solución de riesgos de datos, incluyendo cortocircuito (`forwarding`) y unidad de detección de riesgos.
- **Avance III – Saltos (Jump, JAL, JR, JALR):** Incorporación de instrucciones de salto y control del flujo mediante `flush` y `stall`. Se gestionan riesgos de control asociados.
- **Avance IV – Branch (BEQ, BNE):** Se agregan instrucciones condicionales. Se implementan estrategias para suponer salto no tomado y mitigación de riesgos de control con `forwarding` y unidades de control.
- **Avance V – Ready MIPS y UART:** Adaptación del diseño a hardware real (Basy3) con una interfaz UART simplificada para la visualización de registros. Se garantiza cumplimiento de requisitos de timing.

Resumen de Casos de Prueba

A lo largo del documento se evaluaron múltiples escenarios detallados, que se resumen a continuación:

- **Caso A–D:** Instrucciones aritméticas sin y con riesgos de datos.
- **Caso E:** Riesgo tipo Load-Use.
- **Caso F:** Combinación de Load, Store y ALU.
- **Caso G–K:** Saltos incondicionales y condicionales con y sin riesgo, incluyendo JAL y JR.
- **Caso L–O:** Instrucciones de salto condicional (BEQ/BNE) con riesgos de datos y control.
- **Caso J (final):** Riesgo por dependencia entre instrucciones `load` y `branch`, incluyendo espera de 2 ciclos.

Conclusión: Esta metodología de desarrollo documentado permitió validar de forma incremental cada componente del procesador segmentado, asegurando robustez en cada etapa del diseño.

Casos de Prueba mas importantes

ADD con riesgo RAW (Read After Write)

Este ejemplo muestra instrucciones consecutivas con dependencia de registros del tipo **RAW** (Read After Write). Se verifica que el mecanismo de *forwarding* resuelve correctamente el riesgo, evitando la necesidad de insertar **NOPs**.

Los riesgos de datos ocurren cuando, debido a la segmentación del pipeline, se altera el orden natural de lectura de operandos y escritura de resultados que impone el programa. En este contexto, un riesgo se produce si existe dependencia entre instrucciones que se ejecutan en paralelo.

Dependiendo del tipo de segmentación, el riesgo puede materializarse o no. Se distinguen tres tipos principales de riesgos de datos:

- **Lectura después de escritura (RAW / LDE):** Ocurre si una instrucción intenta leer un registro antes de que una instrucción previa termine de escribirlo.
- **Escritura después de lectura (WAR / EDL):** Ocurre si una instrucción intenta escribir en un registro antes de que otra instrucción previa termine de leerlo.
- **Escritura después de escritura (WAW / EDE):** Ocurre si una instrucción intenta escribir en un registro antes de que una instrucción anterior termine de escribir en el mismo registro.

Addres		Instrucción	M Registros			M Datos		Ciclo
decimal	hexa		REG	DECIMAL	BIN	Addres decimal	Val bin	
0	0	ADDI R1,R0,5	R1	5	101	-	-	5
4	4	ADDI R2,R0,6	R2	6	110	-	-	6
8	8	ADDI R3,R0,9	R3	9	1001	-	-	7
12	C	ADDI R4,R0,10	R4	10	1010	-	-	8
16	10	ADDI R5,R0,11	R5	11	1011	-	-	9
20	14	ADDI R6,R0,18	R6	18	10010	-	-	10
24	18	ADDI R7,R0,19	R7	19	10011	-	-	11
28	1C	ADDU R8,R6,R7	R8	37	100101	-	-	12
32	20	ADDU R9,R1,R2	R9	11	1011	-	-	13
36	24	ADDU R3,R4,R5	R3	21	10101	-	-	14
40	28	ADDU R10,R3,R4	R10	31	11111	-	-	15
44	2C	ADDU R11,R3,R4	R11	31	11111	-	-	16
48	30	ADDU R11,R3,R4	R11	31	11111	-	-	17
52	34	ADDU R11,R3,R4	R11	31	11111	-	-	18
56	38	ADDU R11,R3,R4	R11	31	11111	-	-	19
60	3C	ADDU R3,R4,R5	R3	21	10101	-	-	20
64	40	ADDU R4,R11,R5	R4	42	101010	-	-	21
68	44	ADDU R10,R3,R4	R10	63	111111	-	-	22
72	48	HALT						23

Figura 18: Ejemplo de ADD con riesgo RAW (Read After Write)

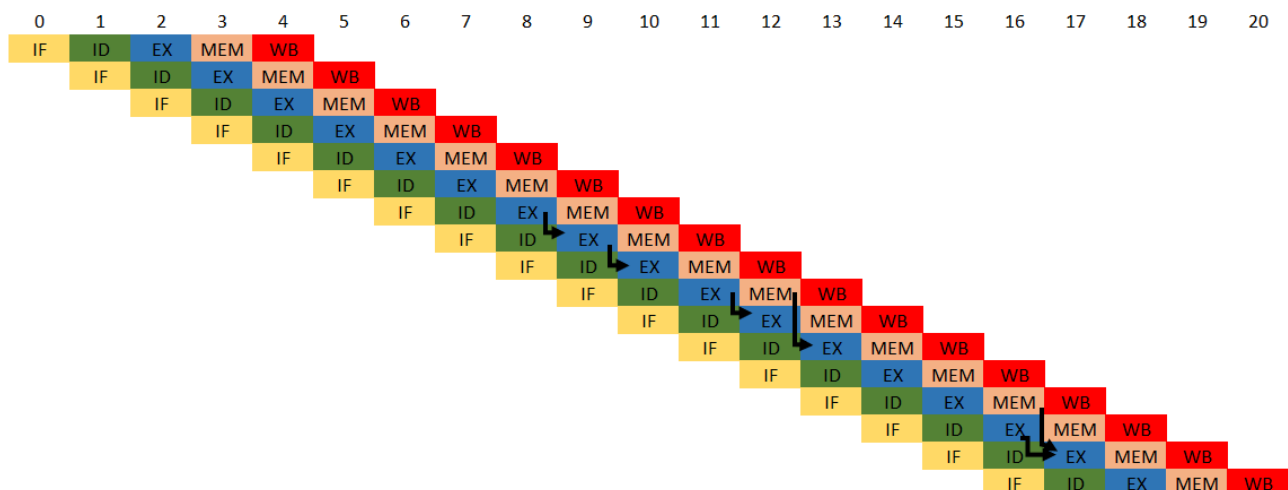


Figura 19: ejecución segmentada del conjunto de instrucciones en un procesador con pipeline

Cada fila representa una instrucción distinta, y cada columna corresponde a un ciclo de reloj. Las etapas del pipeline están codificadas por colores.

Las **flechas negras** indican puntos donde se presentan riesgos de datos del tipo **RAW (Read After Write)**. Este riesgo ocurre cuando una instrucción necesita leer un registro que aún no fue actualizado por una instrucción anterior.

Gracias al mecanismo de **forwarding** (también llamado *bypass*), el procesador puede adelantar el valor producido por una instrucción en su etapa EX o MEM hacia la etapa EX de una instrucción posterior que lo necesita, sin tener que esperar a que dicho valor llegue a la etapa WB.

Ejemplo: en el ciclo 9, una flecha conecta la etapa EX de una instrucción con la etapa EX de otra instrucción posterior. Esto indica que el resultado de la ALU se está reenviando directamente, resolviendo el riesgo sin necesidad de insertar NOPs (burbujas).

Riesgo Load-Use

Cuando una instrucción LW es seguida por una instrucción de tipo R, es necesario cortocircuitar la salida de la memoria (de la instrucción LW) con la entrada de la ALU (de la segunda instrucción).

Para que esto sea posible, la Unidad de Cortocircuito (*Forwarding Unit*) requiere la siguiente información, que se propaga a través de los registros de segmento hasta la etapa EX:

- El registro destino de la instrucción anterior (**rd** en tipo R, **rt** en tipo I).
- Los registros fuente de la nueva instrucción (**rs** y **rt**).
- La señal de control **RegWrite**, que indica si la instrucción escribe en un registro (como en una LW o una instrucción tipo R).

Como se puede observar, incluso utilizando el mecanismo de anticipación de resultados (*forwarding*), **persiste una dependencia temporal entre ambas instrucciones.**

En el caso de la instrucción LW R1, 0(R2), el valor cargado en R1 no estará disponible hasta la cuarta etapa del pipeline (MEM), ya que se recupera desde memoria. Sin embargo, la instrucción siguiente necesita ese valor en su segunda etapa (EX).

Problema: Esta dependencia no puede ser resuelta por la Unidad de Cortocircuito, ya que el dato aún no ha sido recuperado de la memoria cuando se lo necesita.

Solución: La instrucción dependiente debe esperar un ciclo de reloj adicional antes de continuar su ejecución. Esta pausa se conoce como **stall** o **burbuja**.

Unidad de Detección de Riesgos (HAZARD)

Además de la unidad de *forwarding*, se requiere una *Hazard Unit*. Esta unidad se activa durante la etapa ID, y es responsable de insertar un bloqueo cuando una instrucción de tipo `load` es seguida por una que necesita su resultado inmediato.

La detección del riesgo se basa en las siguientes condiciones:

1. La señal de control **MemRead** de la instrucción anterior.
2. El registro **rt** de la instrucción `LW` (registro destino).
3. Los registros **rs** y **rt** de la instrucción actual en la etapa ID.

Ejemplo de código

```
lw    $s2, 16($s3)  # s2 ← Mem[s3 + 16]
add   $v1, $s2, $v0  # v1 ← s2 + v0
```

- `lw` requiere 4 etapas para obtener el dato (hasta **MEM**).
- `add` necesita ese dato en su **EX**.
- Se inserta un **stall** entre ambas.

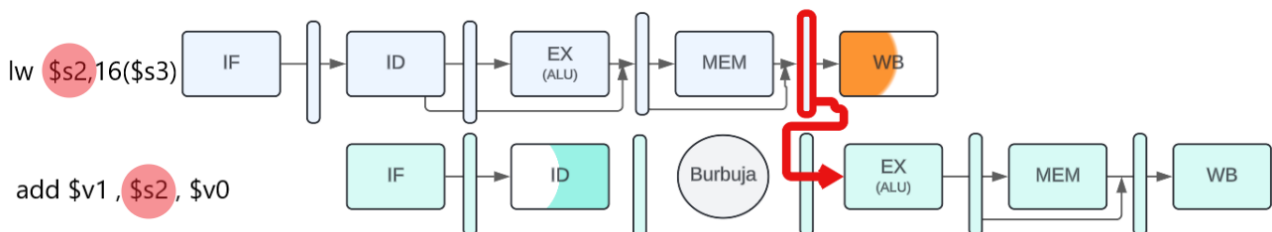


Figura 20: Instrucción `lw` seguida de `add` con dependencia de datos

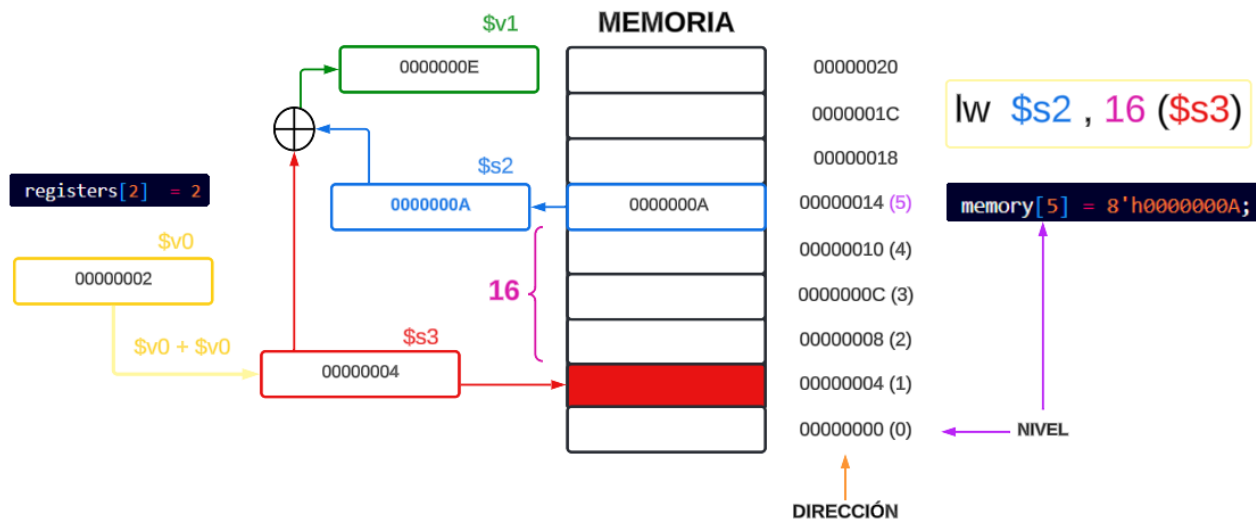


Figura 21: Detección del riesgo en la etapa ID y bloqueo del pipeline

Address		Instrucción	M Registros			M Datos		Ciclo
decimal	hexa		REG	DECIMAL	BIN	Address decimal	Val bin	
0	0	ADDI R1,R0,2	R1	2	10		-	5
4	4	ADDU R2,R1,R1	R2	4	100		-	6
8	8	ADDI R3,R0,6	R3	6	110		-	7
12	C	ADDI R4,R0,10	R4	10	1010		-	8
16	10	SW R4, 14(R3)				20	1010	9
20	14	LW R5, 16(R2)	R5	10	1010	20	1010	10
24	18	ADDU R6,R5,R2	R6	14	1110			11 12
28	1C	HALT						13

Figura 22: Inserción de burbuja (stall) para evitar el uso de datos no disponibles

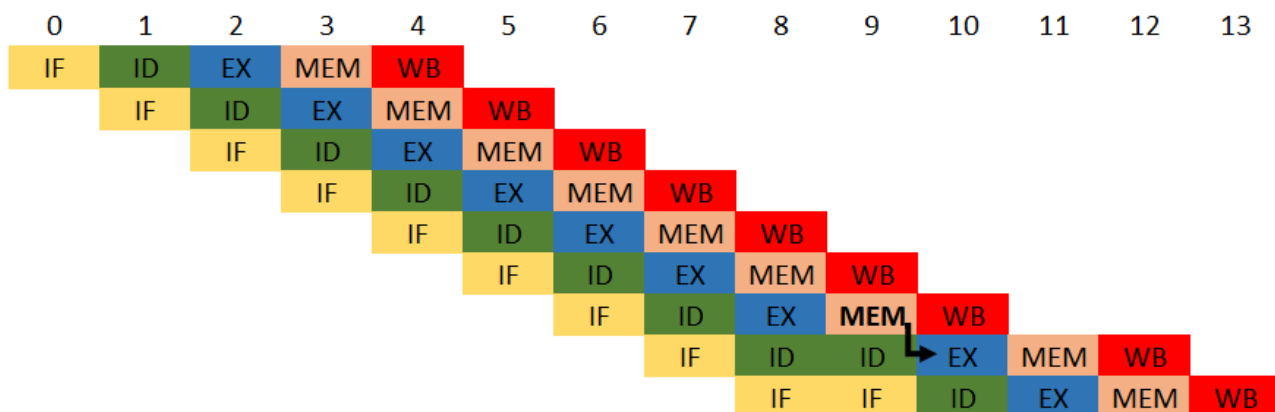


Figura 23: Inserción de burbuja (stall) para evitar el uso de datos no disponibles (2)

Combinación de Load, Store y operaciones aritméticas

Este caso prueba el correcto comportamiento del pipeline al combinar instrucciones de tipo LW, SW y operaciones aritméticas. Se validan tanto riesgos de datos como el acceso correcto a memoria, y se observa cómo interactúan múltiples tipos de instrucciones en un entorno segmentado.

```

ADDI R1, R0, 2
ADDU R2, R1, R1
ADDI R3, R0, 8
ADDI R4, R0, 6
ADDI R5, R0, 12
ADDI R6, R0, 10
SW R6, 14(R4)
SW R3, 24(R2)
LW R7, 16(R2)
SW R2, 14(R7)
LW R8, 16(R5)
ADDU R1, R2, R7
ADDU R9, R8, R1
HALT

```

- Se generan múltiples riesgos RAW por dependencia entre registros consecutivos.
- Las instrucciones SW y LW operan sobre direcciones calculadas dinámicamente.
- Se requiere detección de riesgos y forwarding para evitar stalls innecesarios.
- La instrucción HALT indica el fin del programa.

Address		Instrucción	M Registros			M Datos		Ciclo
decimal	hexa		REG	DECIMAL	BIN	Address decimal	Val bin	
0	0	ADDI R1, R0, 2	R1	2	10			5
4	4	ADDU R2, R1, R1	R2	4	100			6
8	8	ADDI R3, R0, 8	R3	8	1000			7
12	C	ADDI R4, R0, 6	R4	6	110			8
16	10	ADDI R5, R0, 12	R5	12	1100			9
20	14	ADDI R6, R0, 10	R6	10	1010			10
24	18	SW R6, 14(R4)				20	1010	11
28	1C	SW R3, 24(R2)				28	1000	12
32	20	LW R7, 16(R2)	R7	10	1010	20	1010	13
36	24	SW R2, 14(R7)				24	100	14
40	28	LW R8, 16(R5)	R8	8	1000	28	1000	15
44	2C	ADDU R1, R2, R7	R1	14	1110			16
48	30	ADDU R9, R8, R1	R9	22	10110			17
52	34	HALT						18

Figura 24: Combinación de Load, Store y operaciones aritméticas (1)

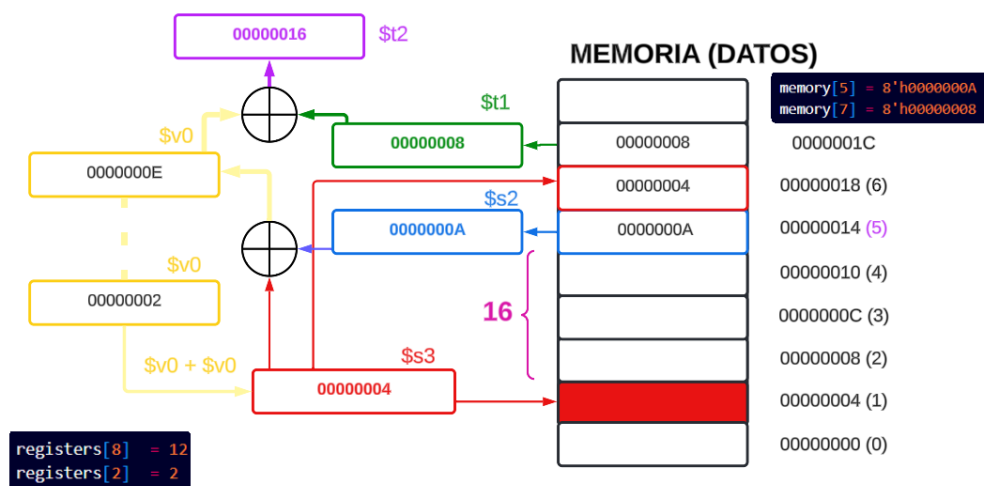


Figura 25: Combinación de Load, Store y operaciones aritméticas (2)

Salto Incondicional (J, JAL)

Una de las características fundamentales que distingue a una computadora de una calculadora es su capacidad para alterar el flujo de ejecución mediante decisiones. En este sentido, los saltos

incondicionales (J, JAL) permiten modificar el contador de programa (PC) y presentan un caso particular de lo que se conoce como **riesgo de control**.

Formato de instrucción: Las instrucciones J y JAL en MIPS tienen un formato de 32 bits con la siguiente estructura:

[opcode (6 bits) | target address (26 bits)]

Cálculo de la dirección de salto:

- Todas las instrucciones MIPS están alineadas a 4 bytes, por lo que los 2 bits menos significativos del PC siempre son cero y no se codifican.
- El campo **target address** representa los bits 27:2 de la dirección real de salto.
- La dirección de salto se reconstruye en tiempo de ejecución con la fórmula:

$$\text{Jump Address} = (PC[31 : 28] \ll 28) | (\text{target address} \ll 2)$$

Branch Delay Slot y Justificación de Diseño

En la arquitectura MIPS clásica, todas las instrucciones de salto (incluidas las incondicionales) introducen una **branch delay slot**, es decir, la instrucción inmediatamente posterior al salto se ejecuta antes de que el salto surta efecto.

Causa: Esto se debe a que la etapa de búsqueda de instrucciones (IF) ya ha cargado la instrucción siguiente antes de que la etapa de decodificación (ID) determine que se trata de un salto. Como resultado, esa instrucción siguiente continúa por el pipeline a menos que se la descarte manualmente.

Enfoque implementado: En nuestro diseño, optamos por seguir el comportamiento clásico del MIPS, manteniendo el concepto de delay slot. Para garantizar un comportamiento consistente y predecible, decidimos insertar explícitamente una instrucción NOP después de cada salto incondicional. De este modo, evitamos que se ejecute una instrucción no deseada, y a su vez simplificamos el control del pipeline.

Justificación:

- A diferencia de los saltos condicionales, cuya dirección depende del resultado de una comparación, los saltos incondicionales tienen su destino completamente definido desde la etapa IF.
- Sin embargo, redirigir el PC desde IF implicaría aumentar la complejidad lógica y romper la modularidad entre etapas del pipeline.
- Insertar un NOP es una solución simple, compatible con la semántica del MIPS, y permite mantener la resolución de saltos en ID de forma uniforme.
- Esta decisión también refleja el diseño original de MIPS.

Conclusión: Aunque técnicamente sería posible redirigir el PC desde IF en el caso de saltos incondicionales, nuestro diseño prioriza la coherencia y la simplicidad del pipeline, adoptando el uso explícito de NOP como parte del manejo de riesgos de control.

Ejemplo: salto incondicional con delay slot (NOP implícito agregado por el compilador)

El siguiente fragmento de código en ensamblador MIPS muestra el uso de saltos incondicionales. Aunque en el código fuente no se explicita, nuestro compilador introduce automáticamente una instrucción NOP (representada por un valor binario nulo) inmediatamente después de cada instrucción de salto, ocupando la **branch delay slot** y evitando la ejecución de instrucciones no deseadas.

```

1  ADDI R2, R0, 2
2  ADDI R4, R0, 4
3  ADDI R6, R0, 6
4  J salto
5  NOP # (Delay slot)
6  ADDI R3, R0, 24      # <-- NO se ejecuta
7  ADDI R5, R0, 24      # <-- NO se ejecuta
8  ADDI R7, R0, 24      # <-- NO se ejecuta
9  ADDI R9, R0, 24      # <-- NO se ejecuta
10 salto:
11 ADDI R8, R0, 8
12 ADDI R10, R0, 10
13 ADDI R12, R0, 12
14 ADDI R14, R0, 14
15 J end
16 NOP # (Delay slot)
17 ADDI R11, R0, 24      # <-- NO se ejecuta
18 ADDI R13, R0, 24      # <-- NO se ejecuta
19 ADDI R15, R0, 24      # <-- NO se ejecuta
20 ADDI R17, R0, 24      # <-- NO se ejecuta
21 end:
22 ADDI R16, R0, 16
23 ADDI R18, R0, 18
24 ADDI R20, R0, 20
25 ADDI R22, R0, 22
26 HALT

```

Listing 2: Ejemplo de código JUMP

Address	Value
0	00000000000000000000000000000000
1	0
2	00000000000000000000000000000010
3	0
4	00000000000000000000000000000100
5	0
6	000000000000000000000000000000110
7	0
8	000000000000000000000000000001000
9	0
10	0000000000000000000000000000001010
11	0
12	0000000000000000000000000000001100
13	0
14	0000000000000000000000000000001110
15	0
16	00000000000000000000000000000010000
17	0
18	00000000000000000000000000000010010
19	0
20	00000000000000000000000000000010100
21	0
22	00000000000000000000000000000010110
23	0
24	0
25	0
26	0
27	0

Figura 26: Salida GUI Jump

Observación: Nuestro compilador incorpora una política de manejo explícito de riesgos de control: al detectar un salto, inserta una instrucción NOP en la branch delay slot.

- Esto garantiza que ninguna instrucción ubicada después del salto sea ejecutada accidentalmente debido al comportamiento del pipeline.
- La NOP ocupa un ciclo y no altera ningún registro ni estado del procesador.
- Esta estrategia simplifica el diseño del pipeline y evita la necesidad de lógica de cancelación o flush en hardware.

En los procesadores MIPS reales, las instrucciones de salto son saltos retardados, los cuales no transfieren el control hasta que la instrucción que sigue al salto (su “ranura de retardo”) se ha ejecutado (véase capítulo 4).

— Patterson y Hennessy, pág. 862, Ed. Española, 4ta ed.

Conclusión: Aunque el salto incondicional se resuelve en la etapa de decodificación, al haber una instrucción ya cargada en la etapa IF, se requiere ocupar ese ciclo. Insertar una NOP de forma implícita asegura un comportamiento predecible y coherente con la semántica de MIPS.

Salto con linkeo (JAL)

```

1  ADDI R2,R0,2      # PC = 4
2  ADDI R4,R0,4      # PC = 8
3  ADDI R6,R0,6      # PC = 12
4  JAL salto         # PC = 16
5  NOP               # PC = 20
6  ADDI R3,R0,24     # PC = 24 <=
7  ADDI R5,R0,24     # PC = 28
8  ADDI R7,R0,24     # PC = 32
9  ADDI R9,R0,24     # PC = 36
10 salto:
11 ADDI R8,R0,8       # PC = 40
12 ADDI R10,R0,10     # PC = 44
13 ADDI R12,R0,12     # PC = 48
14 ADDI R14,R0,14     # PC = 52
15 HALT              # PC = 56

```

Listing 3: Ejemplo de código JAL

Address	Value
0	00000000000000000000000000000000
1	0
2	00000000000000000000000000000010
3	0
4	00000000000000000000000000000100
5	0
6	00000000000000000000000000000110
7	0
8	00000000000000000000000000001000
9	0
10	00000000000000000000000000001010
11	0
12	00000000000000000000000000001100
13	0
14	00000000000000000000000000001110
28	0
29	0
30	0
31	00000000000000000000000000001100

DEC 24

Figura 27: Salida GUI JAL

Saltar con Registro (JR)

```

1  ADDI R6,R0,40     # PC = 4
2  ADDI R4,R0,4      # PC = 8
3  ADDI R2,R0,4      # PC = 12
4  JR R6             # PC = 16
5  NOP               # PC = 20
6  ADDI R3,R0,24     # PC = 24
7  ADDI R5,R0,24     # PC = 28
8  ADDI R7,R0,24     # PC = 32
9  ADDI R9,R0,24     # PC = 36
10 ADDI R8,R0,8       # PC = 40 <=
11 ADDI R10,R0,10     # PC = 44
12 ADDI R12,R0,12     # PC = 48
13 ADDI R14,R0,14     # PC = 52
14 HALT              # PC = 56

```

Listing 4: Ejemplo de código JAL

Address	Value
0	00000000000000000000000000000000
1	0
2	000000000000000000000000000000100
3	0
4	000000000000000000000000000000100
5	0
6	0000000000000000000000000000101000
7	0
8	00000000000000000000000000001000
9	0
10	00000000000000000000000000001010
11	0
12	00000000000000000000000000001100
13	0
14	00000000000000000000000000001110
15	0
16	0

Figura 28: Salida GUI JR

Saltar con Registro (JR)

```

1  ADDI R6 ,R0 ,40      # PC = 4
2  ADDI R4 ,R0 ,4       # PC = 8
3  ADDI R2 ,R0 ,4       # PC = 12
4  JALR R1 ,R6          # PC = 16
5  NOP                 # PC = 20
6  ADDI R3 ,R0 ,24      # PC = 24
7  ADDI R5 ,R0 ,24      # PC = 28
8  ADDI R7 ,R0 ,24      # PC = 32
9  ADDI R9 ,R0 ,24      # PC = 36
10 ADDI R8 ,R0 ,8       # PC = 40  <=
11 ADDI R10,R0 ,10      # PC = 44
12 ADDI R12,R0 ,12      # PC = 48
13 ADDI R14,R0 ,14      # PC = 52
14 HALT                # PC = 56

```

Listing 5: Ejemplo de código JALR

Address	Value
0	00000000000000000000000000000000
1	00000000000000000000000000000000
2	00000000000000000000000000000000
3	0
4	00000000000000000000000000000000
5	0
6	00000000000000000000000000000000
7	0
8	00000000000000000000000000000000
9	0
10	00000000000000000000000000000000
11	0
12	00000000000000000000000000000000
13	0
14	00000000000000000000000000000000
15	0
16	0

Figura 29: Salida GUI JALR

BEQ sin riesgo

Se analiza una instrucción **BEQ** que no depende de los valores de registros que acaban de ser escritos.

```

1  ADDI R2 ,R0 ,2       # PC = 4
2  ADDI R4 ,R0 ,4       # PC = 8
3  ADDI R6 ,R0 ,6       # PC = 12
4  BEQ  R2 ,R2 ,salto   # PC = 16
5  NOP                 # PC = 20
6  ADDI R3 ,R0 ,24      # PC = 24
7  ADDI R5 ,R0 ,24      # PC = 28
8  ADDI R7 ,R0 ,24      # PC = 32
9  ADDI R9 ,R0 ,24      # PC = 36
10 salto:
11 ADDI R8 ,R0 ,8       # PC = 40
12 ADDI R10,R0 ,10      # PC = 44
13 ADDI R12,R0 ,12      # PC = 48
14 ADDI R14,R0 ,14      # PC = 52
15 HALT                # PC = 56

```

Listing 6: Ejemplo de código BEQ

Address	Value
0	00000000000000000000000000000000
1	0
2	00000000000000000000000000000000
3	0
4	00000000000000000000000000000000
5	0
6	00000000000000000000000000000000
7	0
8	00000000000000000000000000000000
9	0
10	00000000000000000000000000000000
11	0
12	00000000000000000000000000000000
13	0
14	00000000000000000000000000000000
15	0
16	0

Figura 30: Salida GUI BEQ

BEQ con riesgo

Se fuerza una situación de riesgo de datos para la instrucción **BEQ**.

```

1  ADDI R2,R0,2      # PC = 4
2  ADDI R4,R0,4      # PC = 8
3  ADDI R6,R0,4      # PC = 12
4  BEQ  R4,R6,salto  # PC = 16
5  NOP               # PC = 20
6  ADDI R3,R0,24     # PC = 24
7  ADDI R5,R0,24     # PC = 28
8  ADDI R7,R0,24     # PC = 32
9  ADDI R9,R0,24     # PC = 36
10 salto:
11 ADDI R8,R0,8       # PC = 40
12 ADDI R10,R0,10     # PC = 44
13 ADDI R12,R0,12     # PC = 48
14 ADDI R14,R0,14     # PC = 52
15 HALT               # PC = 56

```

Listing 7: BEQ verdadero

```

1  ADDI R2,R0,2      # PC = 4
2  ADDI R4,R0,4      # PC = 8
3  ADDI R6,R0,10     # PC = 12
4  BEQ  R4,R6,salto  # PC = 16
5  NOP               # PC = 20
6  ADDI R3,R0,24     # PC = 24
7  ADDI R5,R0,24     # PC = 28
8  ADDI R7,R0,24     # PC = 32
9  ADDI R9,R0,24     # PC = 36
10 salto:
11 ADDI R8,R0,8       # PC = 40
12 ADDI R10,R0,10     # PC = 44
13 ADDI R12,R0,12     # PC = 48
14 ADDI R14,R0,14     # PC = 52
15 HALT               # PC = 56

```

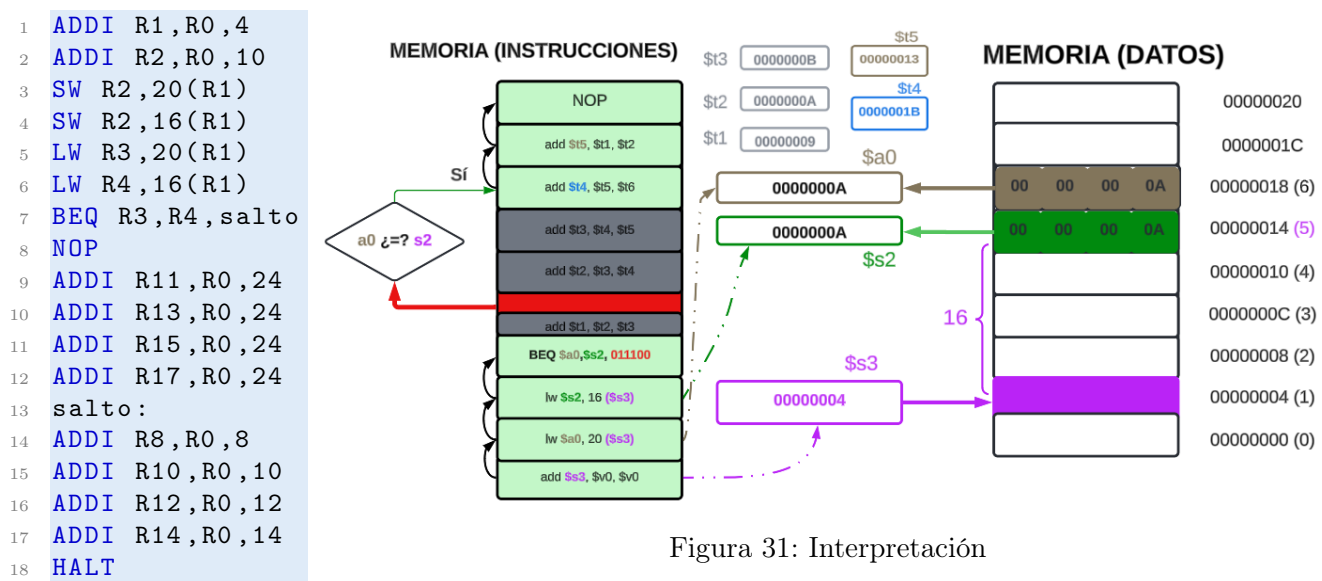
Listing 8: BEQ falso

Address	Value
0	00000000000000000000000000000000
1	0
2	00000000000000000000000000000010
3	0
4	00000000000000000000000000000100
5	0
6	00000000000000000000000000000100
7	0
8	00000000000000000000000000000100
9	0
10	000000000000000000000000000001010
11	0
12	000000000000000000000000000001100
13	0
14	000000000000000000000000000001110
15	0
16	0

Address	Value
0	00000000000000000000000000000000
1	0
2	00000000000000000000000000000010
3	0000000000000000000000000000011000
4	000000000000000000000000000000100
5	0000000000000000000000000000011000
6	000000000000000000000000000001010
7	0000000000000000000000000000011000
8	000000000000000000000000000001000
9	0000000000000000000000000000011000
10	000000000000000000000000000001010
11	0
12	000000000000000000000000000001100
13	0
14	000000000000000000000000000001110
15	0
16	0

BEQ con riesgo (load-branch)

Una instrucción de carga es seguida por una instrucción BEQ que depende del dato cargado.



DATA MEMORY		REGISTERS	
Address	Value	Address	Value
14	0	0	00000000000000000000000000000000
15	0	1	00000000000000000000000000000000
16	0	2	00000000000000000000000000000000
17	0	3	00000000000000000000000000000000
18	0	4	00000000000000000000000000000000
19	0	5	0
20	00000000000000000000000000000000	6	0
21	0	7	0
22	0	8	00000000000000000000000000000000
23	0	9	0
24	00000000000000000000000000000000	10	00000000000000000000000000000000
25	0	11	0
26	0	12	00000000000000000000000000000000
27	0	13	0
28	0	14	00000000000000000000000000000000
29	0	15	0
30	0	16	0

Figura 32: BEQ con riesgo (load-branch)

Referencias

- [1] D. A. Patterson, J. L. Hennessy, *Organización y diseño de computadores: la interfaz hardware/software*, 4^a edición en español, Editorial Reverté, 2012.
- [2] Computerphile, *Branch Hazards - Computerphile*, YouTube, 10:47 min, 2016. Disponible en: <https://www.youtube.com/watch?v=cOWxinc5oRk>
- [3] Computerphile, *Branch Hazards (minuto 19:12 recomendado)*, YouTube. Disponible en: <https://www.youtube.com/watch?v=cOWxinc5oRk&t=1152s>
- [4] Computerphile, *Data Hazards - Computerphile*, YouTube, 8:57 min, 2016. Disponible en: <https://www.youtube.com/watch?v=EW9vtuthFJY>
- [5] Neso Academy, *Branch Hazards in Pipelining*, YouTube, 11:26 min. Disponible en: <https://www.youtube.com/watch?v=2kbi-UhTHks>
- [6] Neso Academy, *Data Hazards and Forwarding*, YouTube, 12:56 min. Disponible en: <https://www.youtube.com/watch?v=CKeXzX5zv30>
- [7] Neso Academy, *Pipeline Stall (Bubble)*, YouTube, 11:09 min. Disponible en: <https://www.youtube.com/watch?v=bZwLymWvA-g>
- [8] Neso Academy, *Branch Prediction Techniques*, YouTube, 13:55 min. Disponible en: <https://www.youtube.com/watch?v=ngkpvMaNapA>
- [9] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011.
- [10] G. Kane, J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.
- [11] D. Sweetman, *See MIPS Run*, Morgan Kaufmann, 2007.