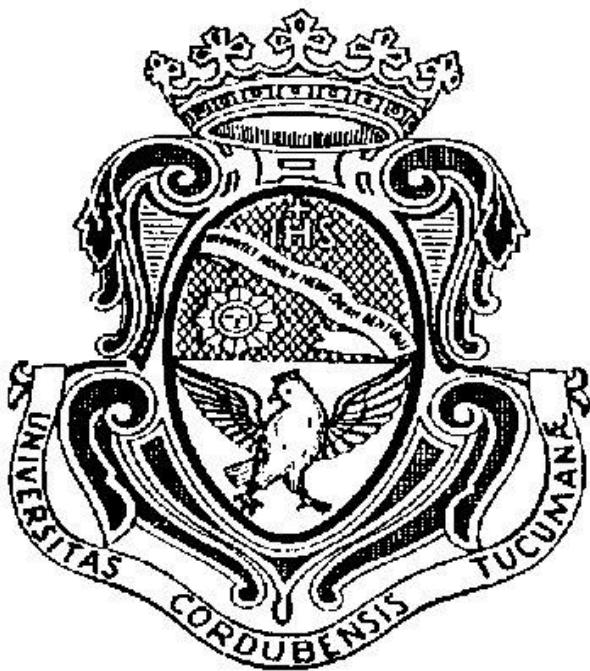


Universidad Nacional de Córdoba

Facultad de Cs. Exactas, Físicas y Naturales



Arquitectura de Computadoras

Trabajo Práctico Final - Pipeline MIPS

- Auet, Luca 42.336.357
- Ojeda, Gastón 40.248.481

Introducción

El siguiente informe trata la implementación del procesador segmentado de 5 etapas del procesador MIPS, que corresponde al trabajo final de la materia de Arquitectura de Computadoras.

Consigna

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

Instrucciones a implementar

- 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

Riesgos

El procesador debe tener soporte para los siguientes tipos:

- 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.

Unidades de Riesgos

Para dar soporte a los riesgos nombrados se debe implementar:

- Unidad de Cortocircuitos
- Unidad de Detección de Riesgos

Otros Requerimientos

- 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.

Debug unit

Se debe enviar a la PC a través de la UART:

- Contenido de los 32 registros
- PC (Program Counter)
- Contenido de la memoria de datos

Modos de operación

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.

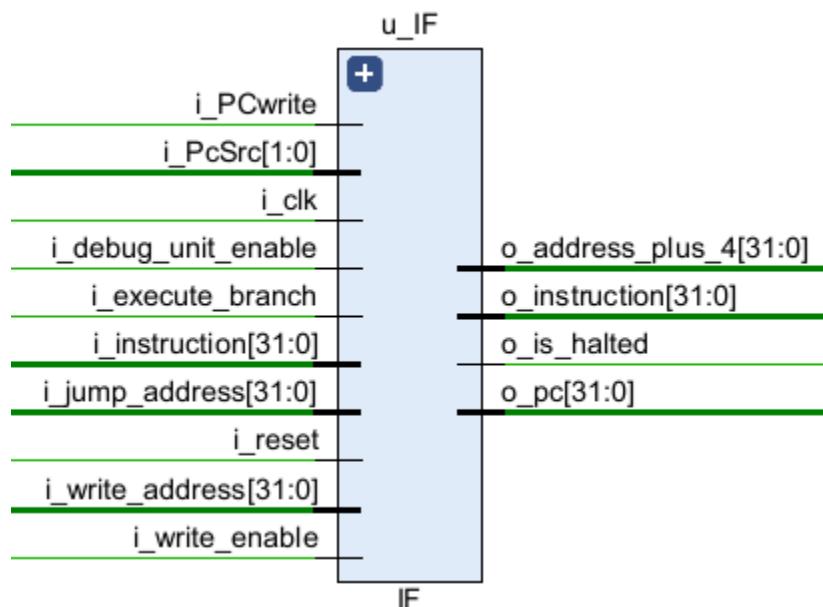
Desarrollo

IF (Instruction Fetch)

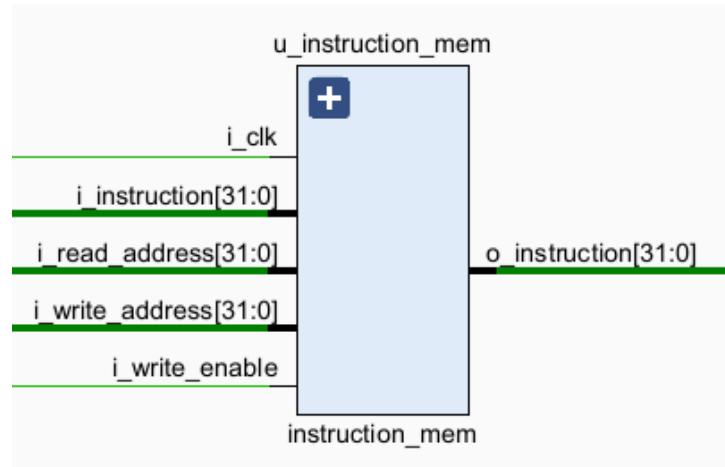
En esta etapa el procesador accede a la memoria de instrucciones mediante un contador de programa (PC) y recupera la siguiente instrucción a ejecutar.

El valor del PC se obtiene de `i_PCSrc` el cual puede ser 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.

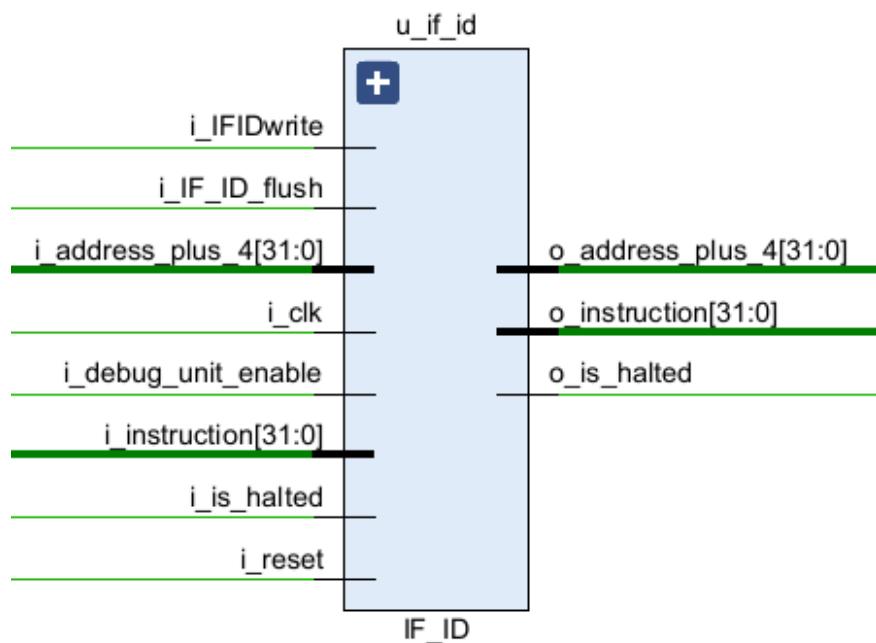


Memoria de instrucciones



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_is_halted** el cual indica si el programa llegó a su fin.



ID (Instruction Decode)

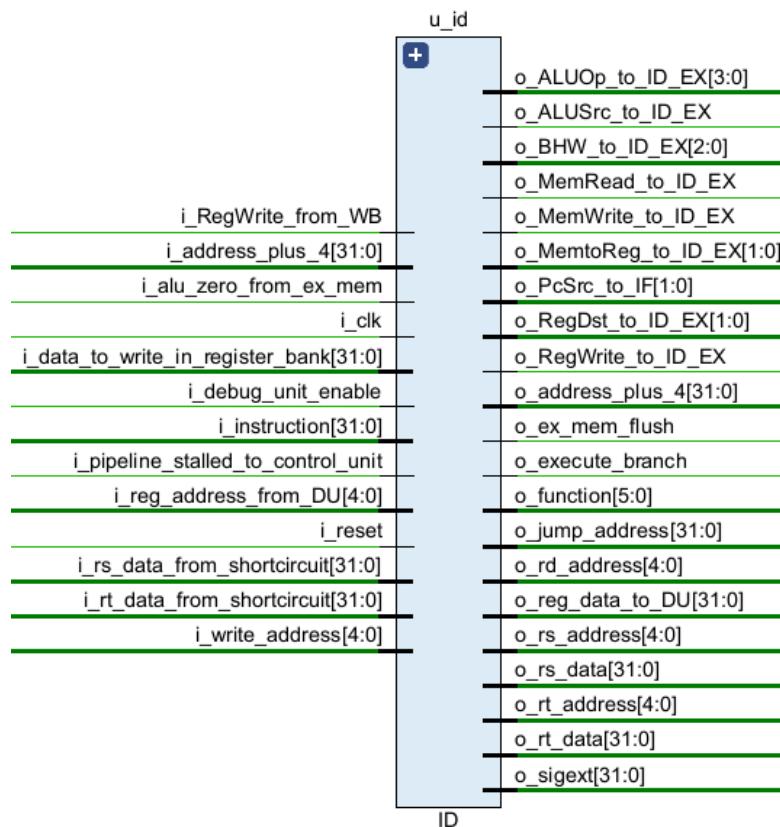
Este módulo lleva a cabo la decodificación de la instrucción y lectura del banco de registros, representando la segunda etapa del pipeline. Los 3 tipos de instrucciones de 32 bits que se decodificaron son los siguientes.

R-type	op	rs	rt	rd	shamt	funct
Arithmetic instruction format						

I-type	op	rs	rt	address/immediate		
Transfer, branch, immediate.						

J-type	op	target address				
Jump instruction						

Field size	6 bits	5bits	5bits	5bits	5bits	6 bits
------------	--------	-------	-------	-------	-------	--------



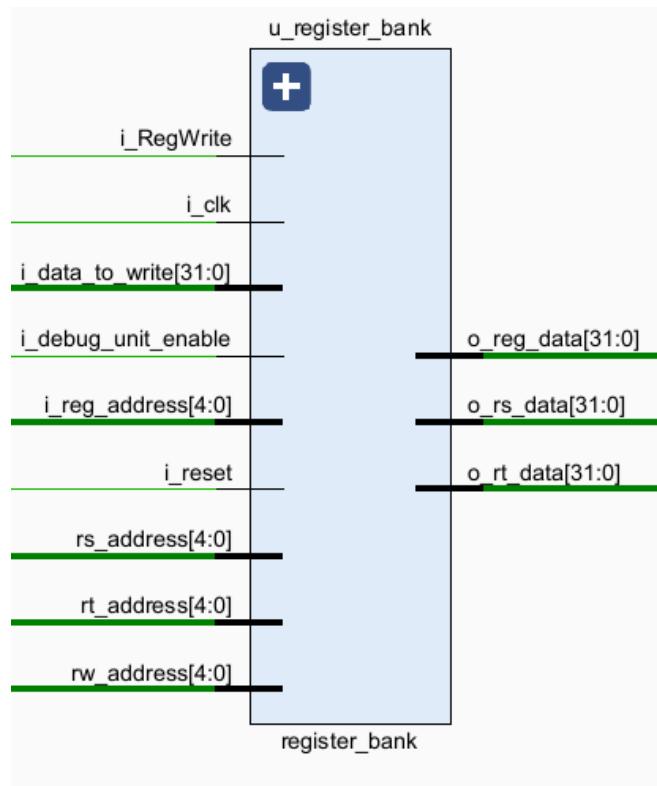
En la etapa de decodificación, la instrucción recuperada se interpreta para determinar qué acciones se deben llevar a cabo.

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

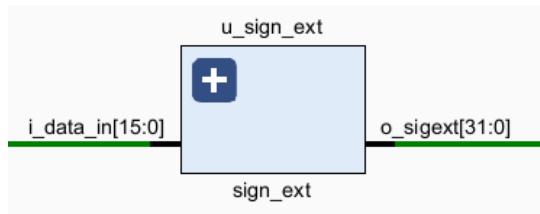
Banco de Registros

Se implementó un banco de registros donde se están los 32 registros del procesador MIPS y de los cuales se extraen los operandos



Extensor de Signo

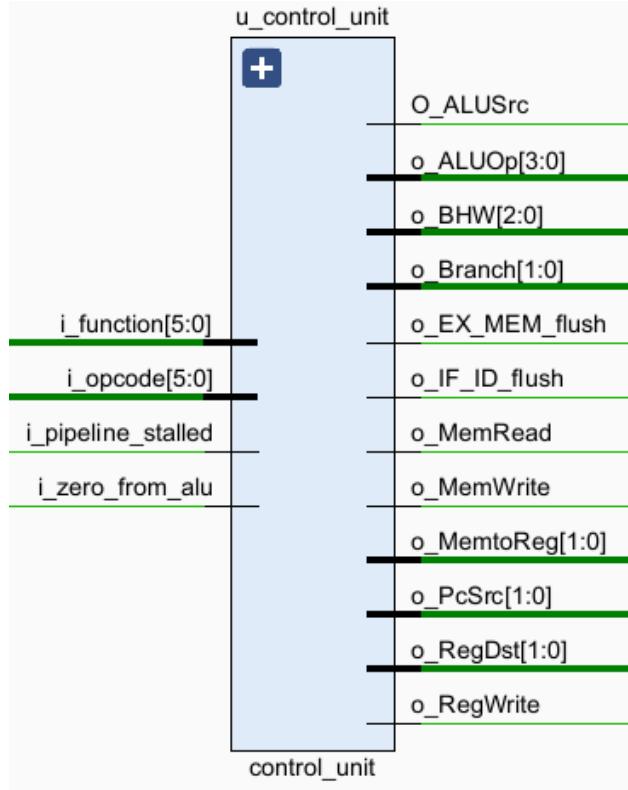
También se implementa un extensor de signo para los casos de las instrucciones tipo I en los que se trabaja con un valor inmediato de 16 bits y se necesita extender a 32 bits



Unidad de Control

Además, desde la unidad de control, se preparan las señales de control necesarias para las etapas siguientes:

- **PcSrc**: indica que entrada del MUX será el nuevo PC.
- **RegDst**: indica cuál será el registro de destino, rt o rd.
- **ALUsrc**: indica cuáles serán las entradas de la ALU.
- **ALUOp**: define el tipo de operación como entrada al control de la ALU.
- **MemRead**: habilita la lectura de memoria.
- **MemWrite**: habilita la escritura de memoria.
- **Branch**: indica si la instrucción es un branch.
- **RegWrite**: habilita la escritura en el banco de registros.
- **MemtoReg**: indica la fuente en la escritura de registro, ya sea ALU, memoria o dirección de retorno.



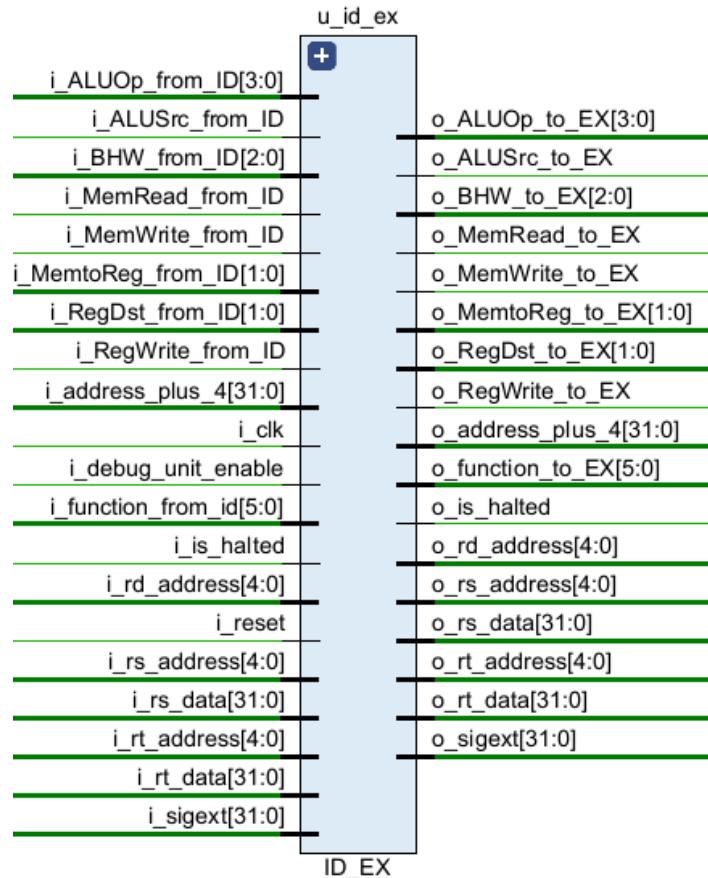
La unidad de control recibe de la instrucción a decodificar el campo OP y function (para las instrucciones tipo R).

En primer lugar se analiza el campo OP para determinar qué tipo de operación es. En caso de que este sea 000000, se deberá analizar el campo funct, a través del cual se determina de qué operación se trata, y a partir de esto se generan las señales de control correspondientes para ejecutar esa instrucción.

Para las tipo I, cada una cuenta con un opcode diferente. Así, en cada caso y para cada instrucción, se setean las distintas señales de control necesarias para el correcto funcionamiento del procesador.

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)

Durante la fase de ejecución, la instrucción se lleva a cabo utilizando la Unidad Aritmética Lógica (ALU). Dependiendo del tipo de instrucción, en esta etapa se pueden realizar:

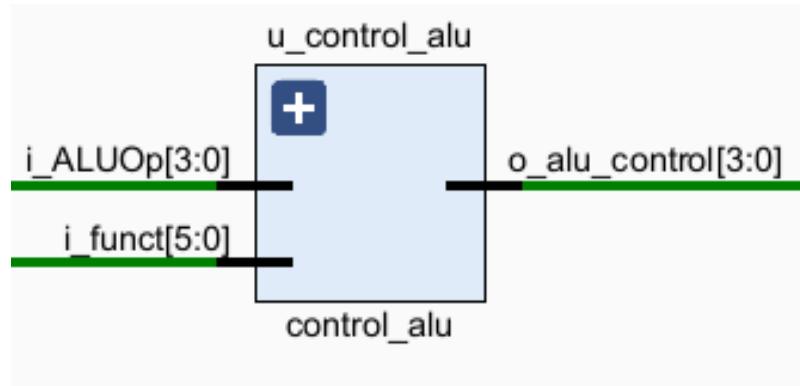
- Operaciones aritméticas y lógicas (suma, resta, AND, OR, etc.).
- Cálculo de direcciones para instrucciones de acceso a memoria (Load y Store).
- Las entradas de esta etapa son los operandos y codigos de operacion extraídos en la etapa anterior

En esta etapa también se encuentran los multiplexores de la unidad de cortocircuito los cuales son controlados por *i_forward_a* e *i_forward_b* para evitar conflictos por dependencia de datos



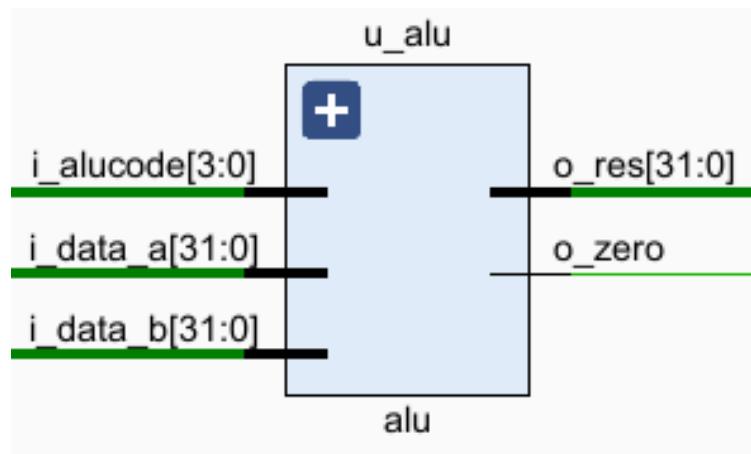
Control ALU

Se encarga de generar la señal de control que especifica la operación a realizar por la ALU.

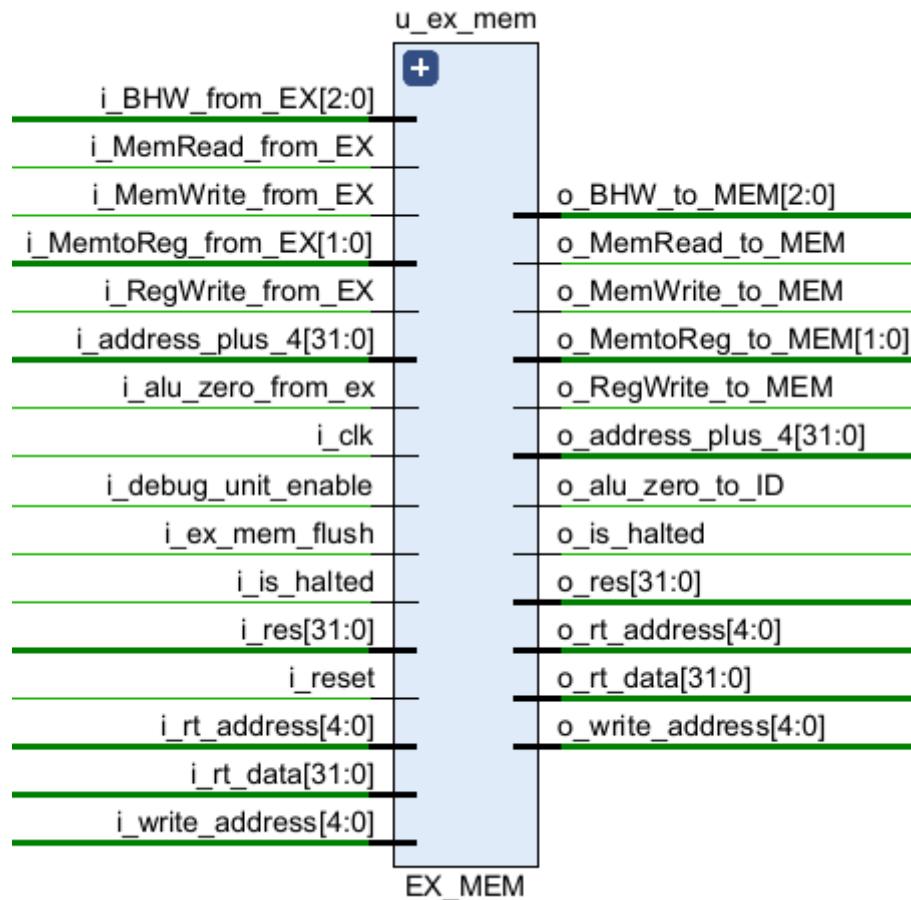


ALU

Unidad Aritmética Lógica combinacional que realiza la operación indicada por el ALU code, a partir de los dos operandos que se encuentran en su entrada.



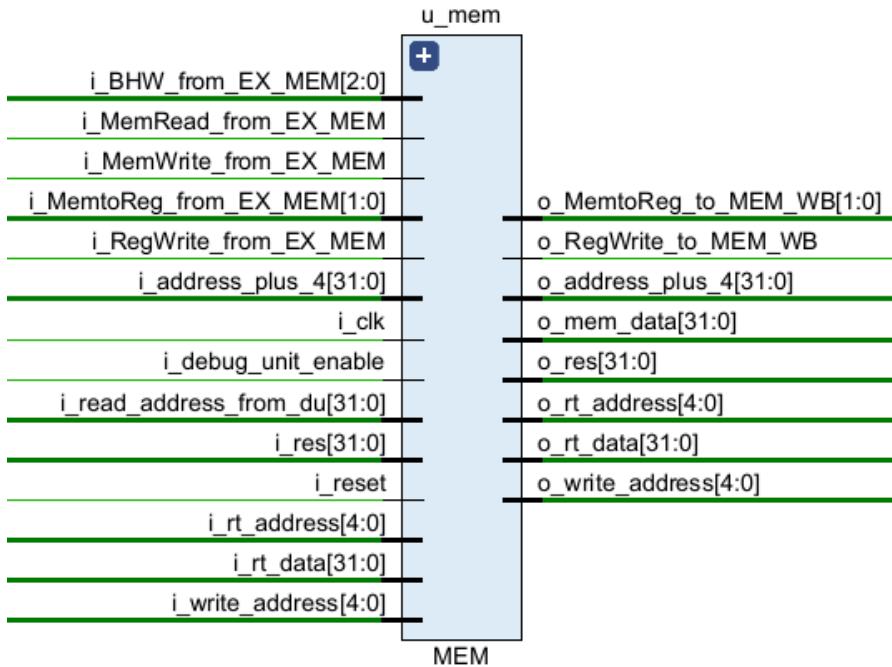
EX/MEM



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.

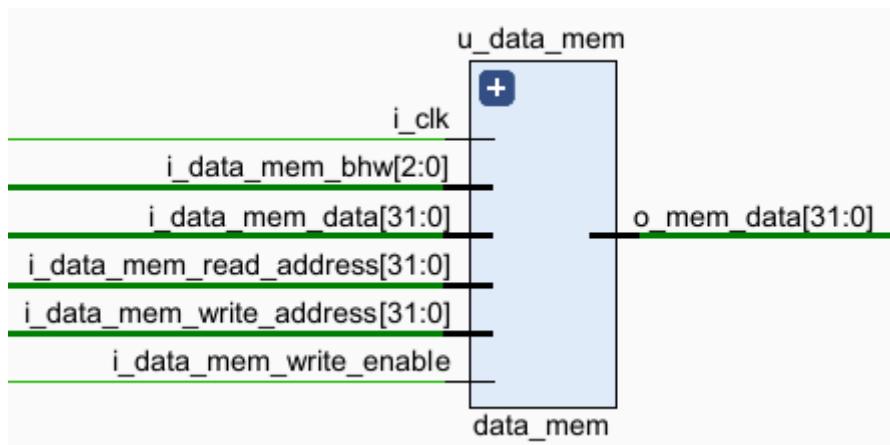
MEM (Memory Access)

En la etapa MEM del MIPS, se realiza un acceso a memoria de datos indicado por la actual instrucción, de ser requerido. Este acceso puede ser para escritura o lectura de memoria, y para esto se tendrán en cuenta diferentes señales de control. Este módulo está integrado por un submódulo denominado `data_mem`.



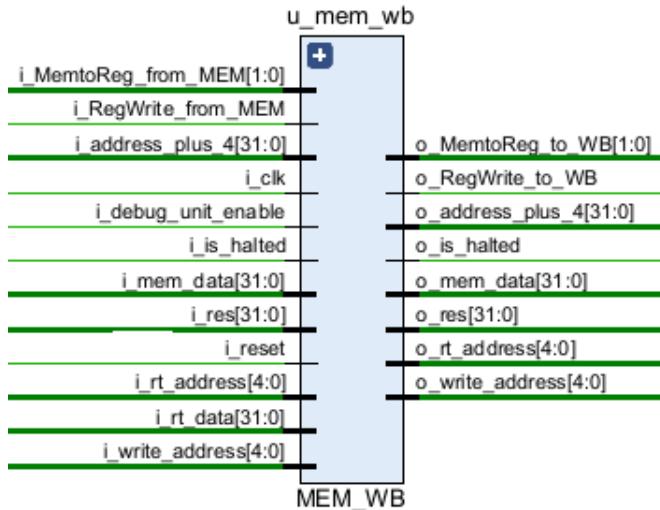
Memoria de Datos

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

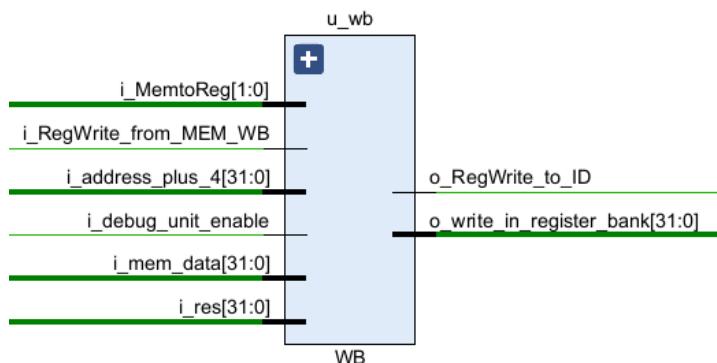
En este módulo se registran tanto la salida de ALU como la salida de la memoria. También se registran las últimas señales de control que llegan hasta la última etapa



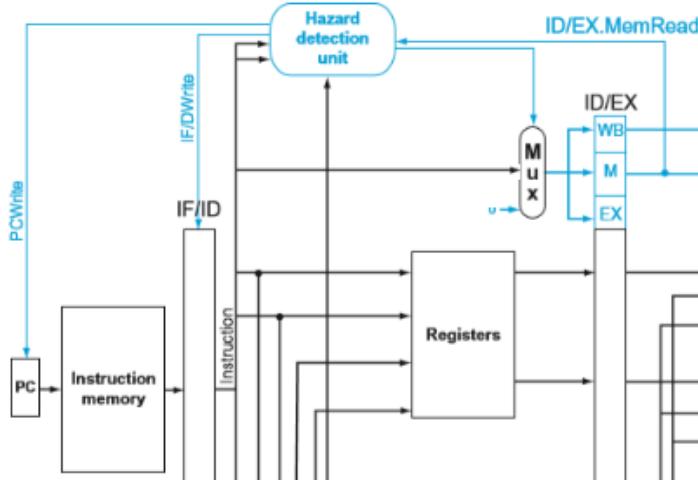
WB (Write Back)

La última etapa del pipeline es la escritura de resultados, donde se actualiza el valor de los registros con los resultados finales de la instrucción. En esta etapa:

- Se implementa un multiplexor el cual mediante la señal de control MemToReg indica si el resultado a escribir proviene de la memoria de datos o de la alu directamente.
- Este resultado se escribe en el registro de destino especificado.
- Se completa el ciclo de la instrucción.

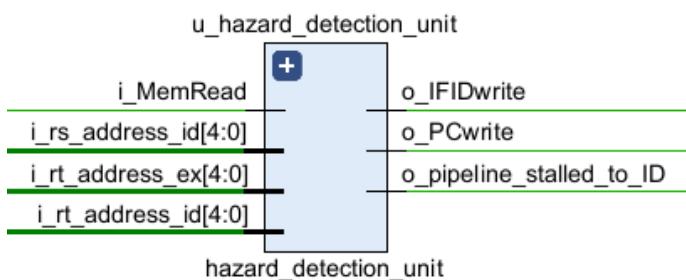


Unidad de Detección de Riesgos



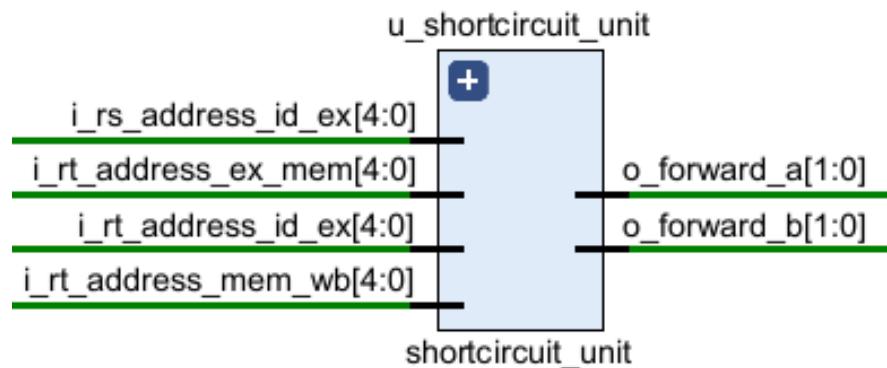
Este módulo tiene como objetivo detectar el riesgo de dependencia de datos, donde es necesario introducir un retardo de un ciclo de reloj (stall). Esto se da cuando la instrucción seguida de un load necesita de la lectura del mismo registro que el load debe escribir en el banco de registros, es decir, que todavía no se encuentra disponible en la memoria de datos ni registros.

Cuando el riesgo es detectado se introduce el retardo impidiendo que durante un ciclo se actualice el PC y se registren los valores en IF/ID, manteniendo así la primera mitad del pipeline frenado, el valor leído de la memoria de datos se multiplexa hacia el operando de la ALU, con el objetivo de que la instrucción que necesita dicho dato pueda realizar la operación.

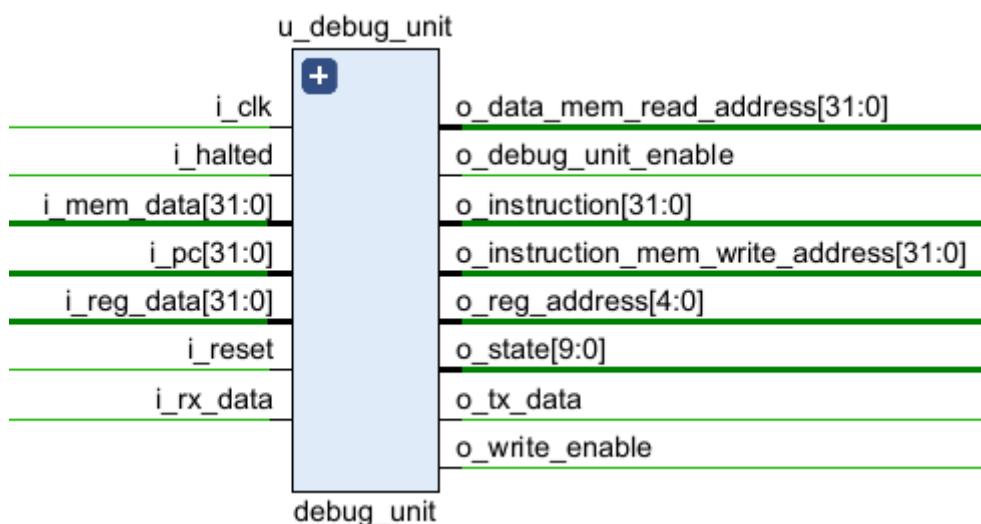


Unidad de Cortocircuito

La unidad de cortocircuito es responsable de verificar las dependencias de datos entre instrucciones al comparar los registros que utilizan dichas instrucciones. Para gestionar esto, se emplean multiplexores que son controlados por diversas señales. Estas señales determinan si es necesario realizar un cortocircuito del dato, permitiendo así que el módulo EX (Ejecución) pueda acceder al dato actualizado directamente.

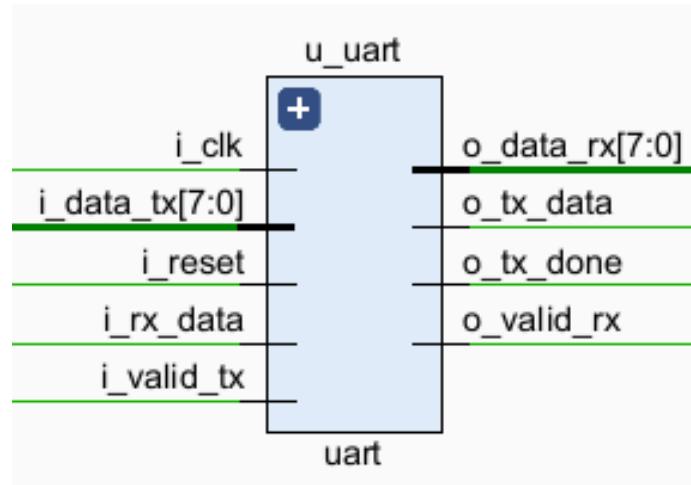


Unidad de Debug



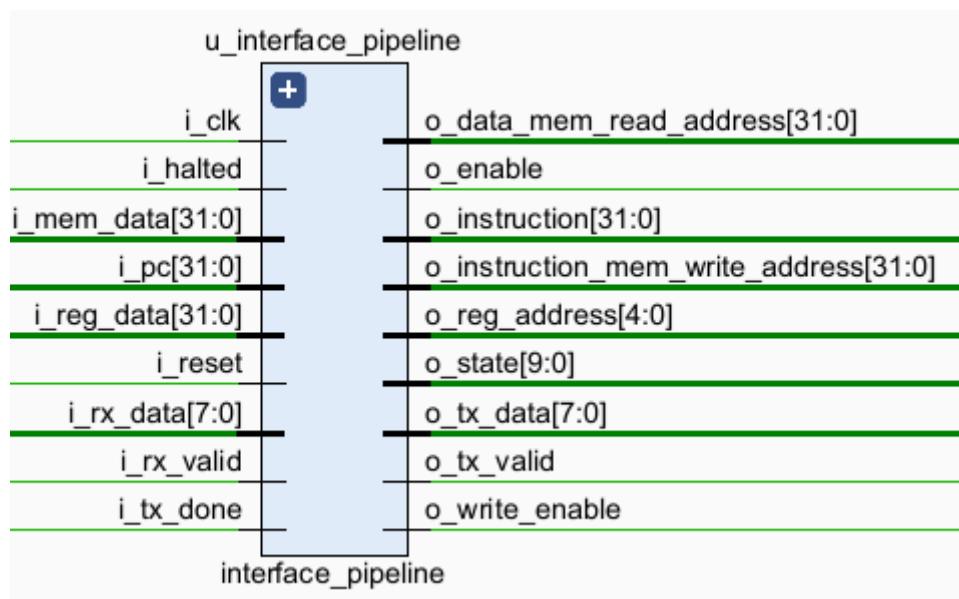
UART

Este módulo permite la comunicación del usuario con el procesador. Recibe desde el módulo UART comandos para ejecutar determinadas acciones.



Interfaz con Pipeline

A su vez la uart se conecta al submódulo **interface_pipeline**, el cual posee una máquina de estados que se encarga de recibir los comandos enviados por el usuario para manejar el funcionamiento del pipeline.



Los comandos implementados son los siguientes:

CMD_SET_INST

Setea el modo de carga de instrucciones. Posterior a este comando se envía un byte con la cantidad de instrucciones a cargar, y luego las instrucciones desde un archivo binario las cuales se cargan en la memoria.

CMD_SET_CONTINUOUS

Setea el modo de ejecución continua. Se ejecutan instrucciones hasta llegar a una instrucción de HALT, donde comienza a imprimir el valor del PC, los registros y la memoria de datos

CMD_SET_STEP_BY_STEP

Setea el modo paso a paso. Se ejecuta el avance de un ciclo del pipeline.

CMD_RUN_STEP

Ejecuta el avance de 1 ciclo en el pipeline

CMD_RESET

Resetea la debug unit

Modo de carga de instrucciones

La implementación de la carga de instrucciones se realizó a través de una conexión de escritura y una dirección con la memoria de instrucciones. Se reciben byte a byte las mismas, cuando se reciben los 4 bytes de una instrucción, se escribe la misma en memoria y se incrementa el puntero de direccionamiento. Así sucesivamente hasta recibir una instrucción de HALT

Modo de ejecución continua y modo paso a paso

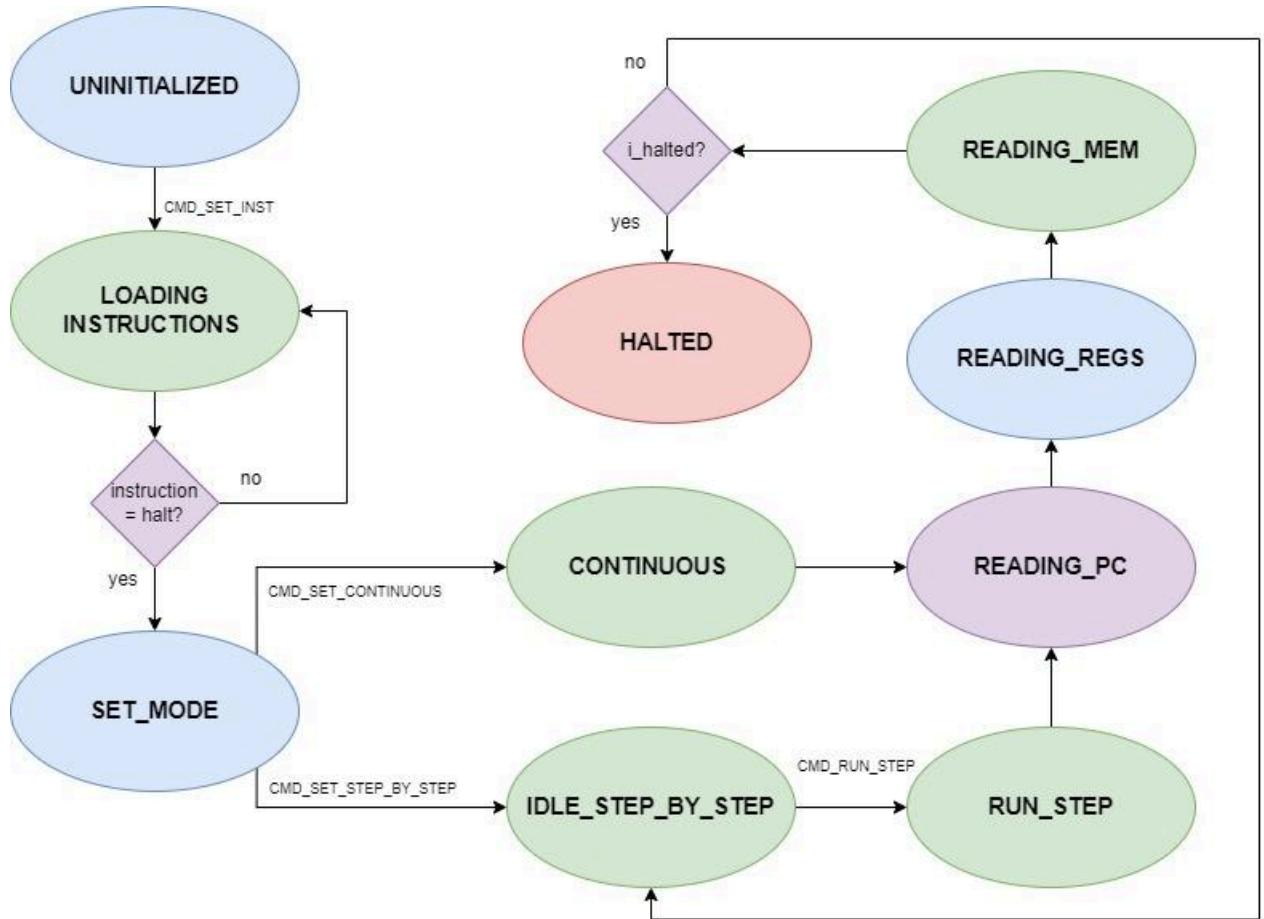
La implementación del modo de ejecución continua y del modo paso a paso se realizó de la siguiente manera.

Desde la debug unit, se envía al pipeline una señal enable, la cual es la encargada de permitir o no el avance de un ciclo del mismo. Solo cuando esta está activa, al darse un ciclo de clock se actualizan los valores del PC y de los registros de segmentación, caso contrario los mismos conservan su valor. De esta manera, es que puede conservarse el estado del pipeline durante los ciclos de clock que se desee.

En el caso del modo de ejecución continua, esta señal permanece activa hasta que se dirccione en la memoria una instrucción de HALT. Cuando esta condición se da,

se imprime el contenido del contador de programa, el valor de los registros y de la memoria.

En el caso del modo paso a paso, la señal habilitadora solo se activa durante un ciclo de clock al recibir el comando de “**CMD_RUN_STEP**”, permitiendo así el avance de un ciclo del pipeline e imprimiendo el contenido del contador de programa, el valor de los registros y de la memoria en cada ciclo.



Estados del módulo de debug

UNINITIALIZED

La unidad de debug está esperando el comando **CMD_SET_INST** para comenzar a ejecutarse

LOADING_INSTRUCTIONS

La unidad de debug está cargando las instrucciones y calculando la cantidad ingresada hasta encontrar una instrucción **HALT**

SET_MODE

En este estado se está esperando el comando **CMD_SET_CONTINUOUS** o **CMD_SET_STEP_BY_STEP** dependiendo el tipo de ejecución que se quiera se pasa al estado **CONTINUOUS** o **IDLE_STEP_BY_STEP**

RUN_STEP

Si se está en el estado IDLE_STEP_BY_STEP este estado espera el comando CMD_RUN_STEP para llevar a cabo otro ciclo de reloj en el pipeline

READING_PC, READING_REGS, READING_MEM

Una vez que la ejecución del estado de ejecución llegó a una instrucción HALT se pasa a este estado donde se lee el valor del contador de programa, luego el valor de los registros y por último los valores de la memoria

HALTED

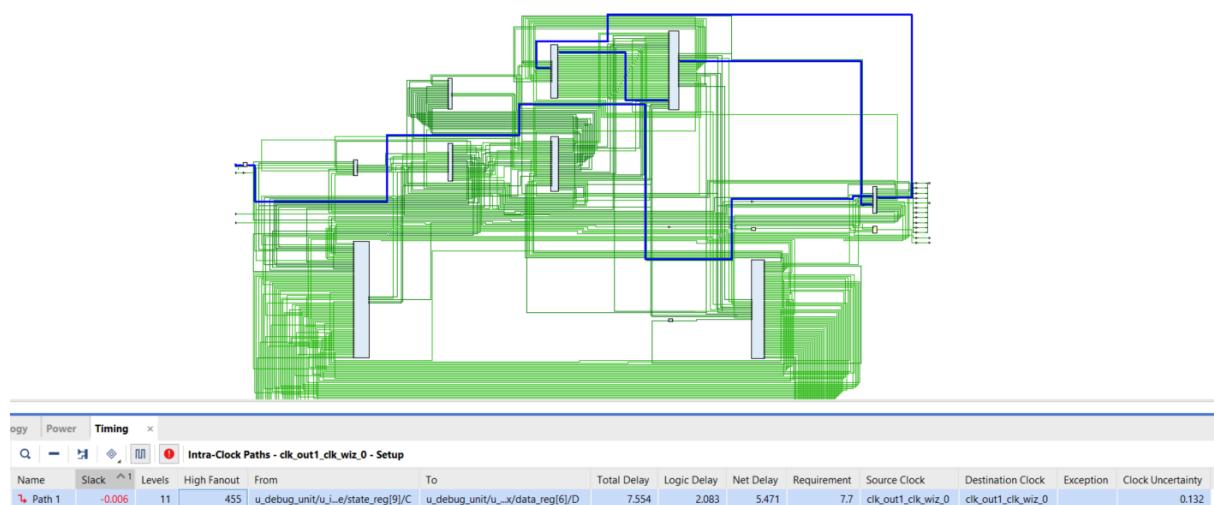
Estado final de la ejecución donde la debug unit queda detenida

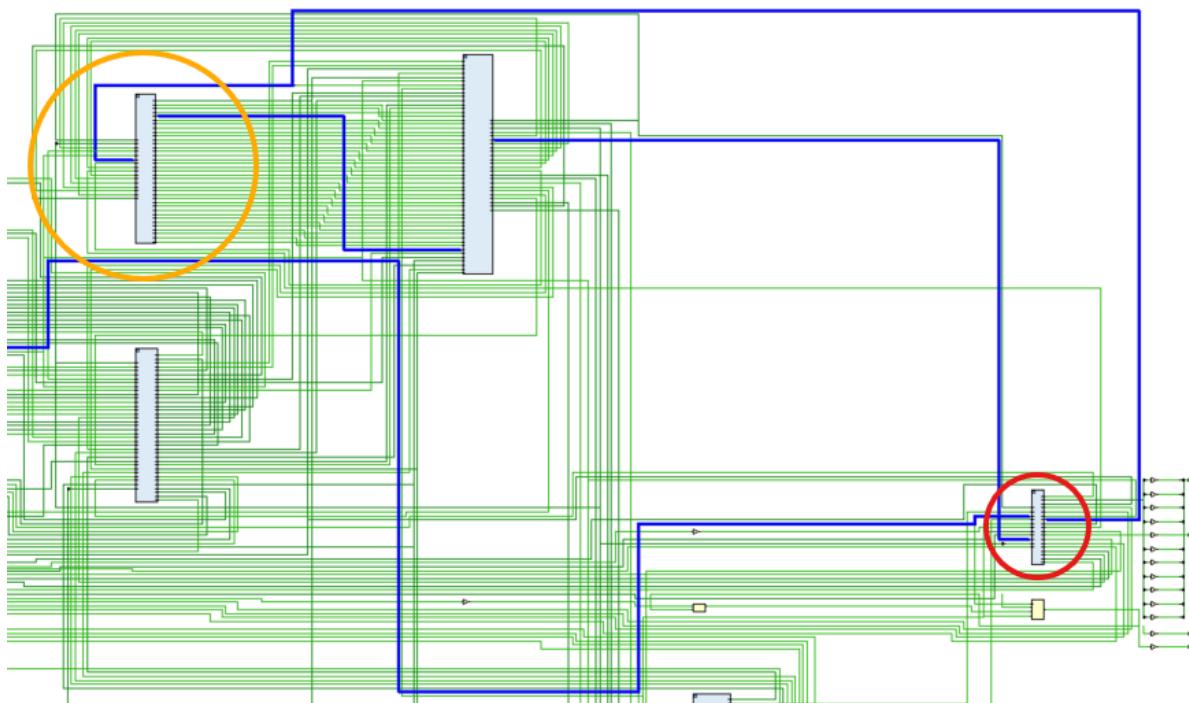
Módulo Clock Wizard

Un punto importante de la implementación es determinar cuál es la frecuencia máxima a la cual puede operar el diseño. Para reducir la frecuencia del clock se implementó un módulo de Clock Wizard. Este, permite generar nuevas salidas de clock de diferentes frecuencias. En nuestro caso, utilizamos una salida de clock de 45MHz la cual se obtuvo mediante análisis de los reportes de timing de Vivado y pruebas en la placa.

Análisis de tiempo

En la imagen se muestra el análisis de tiempo del camino crítico del sistema. Con un clock de 65 MHz, se identifica un camino crítico que impide cumplir con los requisitos de timing, lo que podría afectar el rendimiento y estabilidad del sistema.

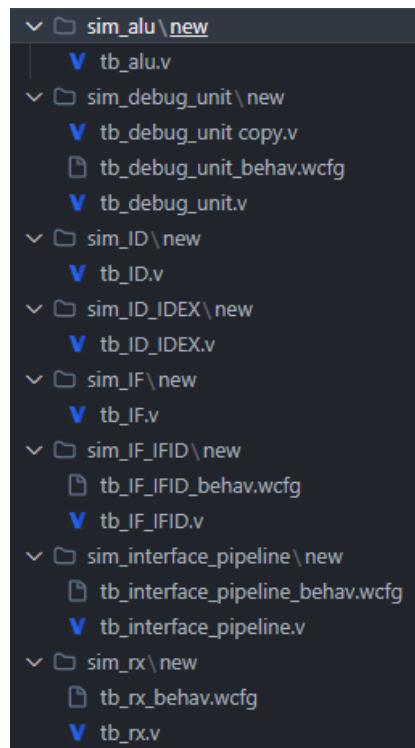




En la imagen, se señala un módulo con un círculo rojo, que corresponde a la unidad de debug. Este módulo tiene como salida una dirección de memoria que se conecta al módulo señalado con un círculo naranja, el módulo de memoria. El problema identificado es que la operación desde la salida de la unidad de debug, pasando por el módulo de memoria, accediendo a la memoria de datos para obtener el dato almacenado en la dirección especificada, y volviendo a utilizar ese dato en la unidad de depuración, se realiza de manera combinacional dentro de un solo ciclo de reloj. Esta configuración resulta en un fanout considerable y un camino de datos extensamente largo, lo que contribuye a que no se cumplan los requisitos de timing del sistema.

Testbenches

Durante el desarrollo del proyecto se llevaron a cabo testbenches individuales de cada módulo para validar el funcionamiento de los mismos y así encerrar posibles problemas y facilitar su resolución.



Una vez validados todos los módulos por separado se realizó un testbench de todos conectados entre sí para probar el funcionamiento completo del pipeline

```
module tb_top;
reg clk;
reg reset;

always #0.5 clk = ~clk;

initial
begin
    clk = 0;
    reset = 1;

    #10
    reset = 0;
end

```

```
top
#(
    .NB_PC(32),
    .NB_IN5(32),
    .N_REG(32),
    .NB_DATA(32),
    .NB_DATA_IN(16),
    .NB_DATA_OUT(32),
    .NB_OP(4),
    .NB_ADDR(32),
    .NB_FUNCTION(6),
    .NB_OPS(6),
    .NB_ALUCODE(4)
)
u_top
(
    .i_clk(clk),
    .i_reset(reset)
);
endmodule
```

Testbench de Integración

Finalmente con el pipeline funcionando se realizó un testbench con la debug unit conectada para probar el funcionamiento completo del proyecto.

El testbench que se ejecutará tiene como objetivo enviar los comandos necesarios para cargar y ejecutar un conjunto de instrucciones en el procesador. Primero, enviará el comando SET_INSTR seguido de la cantidad de instrucciones a cargar, que en este caso son 2. A continuación, enviará ambas instrucciones: LUI r1,1 y HALT. Luego, configurará el modo de ejecución en continuo y permitirá que el procesador execute las instrucciones hasta llegar a la instrucción HALT. Finalmente, el testbench recibirá los datos correspondientes al program counter (PC), los registros (regs), y la memoria de datos (mem) del procesador para su verificación.

```
initial begin
    #0
    clk = 0;
    reset = 1;
    rx_data = 1; // Idle state for UART

    #300;
    reset = 0;

    // Send CMD_SET_INST
    #200;
    send_byte(8'b00000001);

    // Send number of instructions = 2
    #5;
    send_byte(8'b00000010);

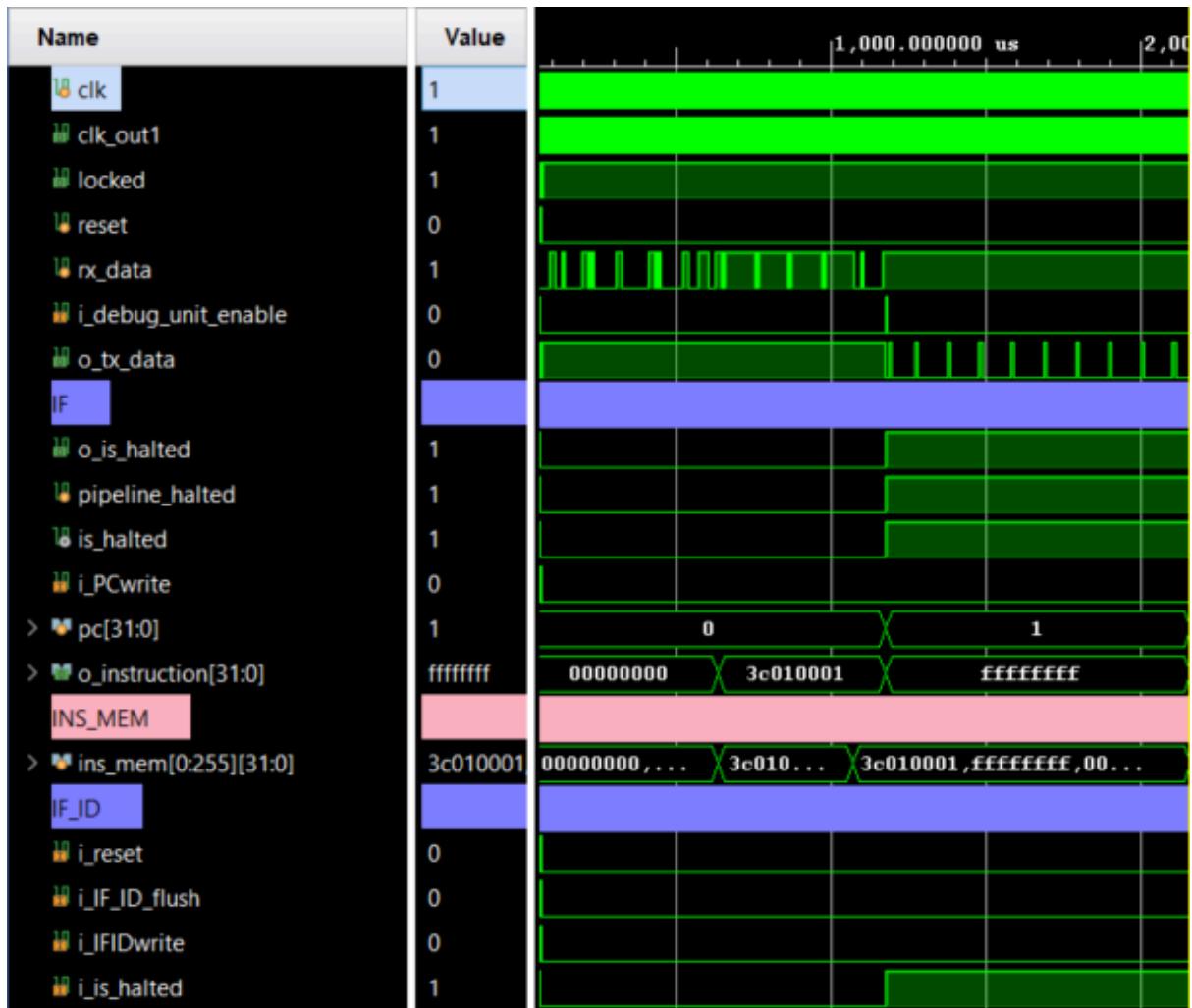
    // Send instruction
    #5;
    send_word(32'h3c010001);

    // Send instruction
    #5;
    send_word(32'hffffffff);

    // Set Mode Continuous
    #20;
    send_byte(8'b00000010);

    #100000;
    $finish;
end | You, last week • Connect du a
always #0.5 clk = ~clk;
```

En el gráfico se observa cómo el pin `rx_data` recibe todos los comandos e instrucciones. Una vez que `rx_data` se pone en 1, indicando que se ha completado la recepción de la información, la unidad de debug activa el enable del pipeline, lo cual se refleja en el pin `i_debug_unit_enable`. Finalmente, después de la ejecución del pipeline, los resultados son enviados de vuelta a la PC, lo que se puede ver en la actividad del pin `o_tx_data`.

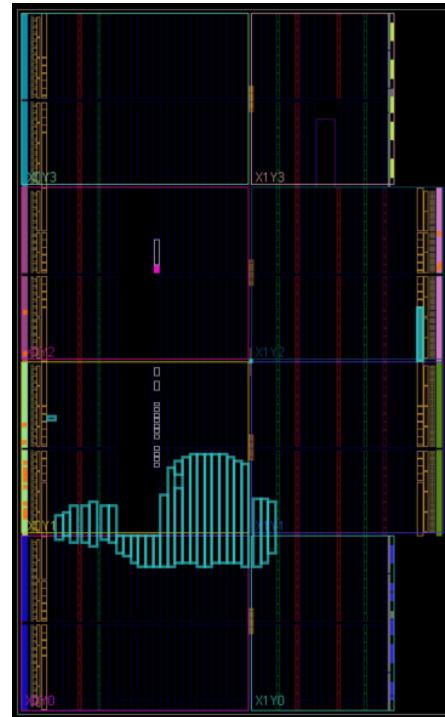
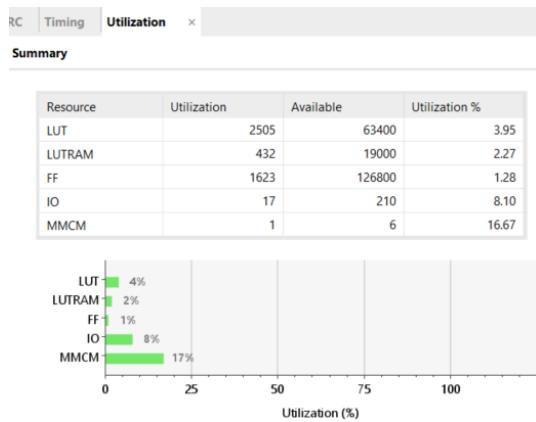


En la imagen del banco de registros, se puede observar que la instrucción LUI ha cargado correctamente el valor 1 desplazado 16 bits en el registro 1, tal como se esperaba.

Implementacion en NEXYS 4 DDR

Reporte de Utilización

Resumen



Detalle

Hierarchy

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
N top	2505	1623	542	40	974	2073	432	17	2	1
u_clk_wiz_0 (clk_wiz_0)	0	0	0	0	0	0	0	0	2	1
inst (clk_wiz_0_clk_wiz)	0	0	0	0	0	0	0	0	2	1
u_debug_unit (debug_unit)	145	130	0	0	73	145	0	0	0	0
inter	75	78	0	0	49	75	0	0	0	0
u_uart (uart)	70	52	0	0	33	70	0	0	0	0
u_baud_rate_generator	9	9	0	0	3	9	0	0	0	0
u_rx (rx)	42	23	0	0	23	42	0	0	0	0
u_tx (tx)	20	20	0	0	9	20	0	0	0	0
u_ex (EX)	0	0	0	0	15	0	0	0	0	0
u_alu (alu)	0	0	0	0	15	0	0	0	0	0
u_ex_mem (EX_MEM)	36	121	0	0	93	36	0	0	0	0
u_id (ID)	864	1025	384	40	495	864	0	0	0	0
u_register_bank (register_l)	864	1024	384	40	488	864	0	0	0	0
u_id_ex (ID_EX)	751	138	30	0	271	751	0	0	0	0
u_IF (IF)	222	33	0	0	86	46	176	0	0	0
u_instruction_mem (instru)	179	0	0	0	46	3	176	0	0	0
u_if_id (IF_ID)	122	65	0	0	67	122	0	0	0	0
u_mem (MEM)	273	0	128	0	78	17	256	0	0	0
u_data_mem (data_mem)	273	0	128	0	78	17	256	0	0	0
u_mem_wb (MEM_WB)	95	110	0	0	92	95	0	0	0	0
u_wb (WB)	0	0	0	0	8	0	0	0	0	0

Reporte de Timing

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,897 ns	Worst Hold Slack (WHS): 0,052 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: 7830	Total Number of Endpoints: 7830	Total Number of Endpoints: 2061

All user specified timing constraints are met.

Ejecución de un Programa

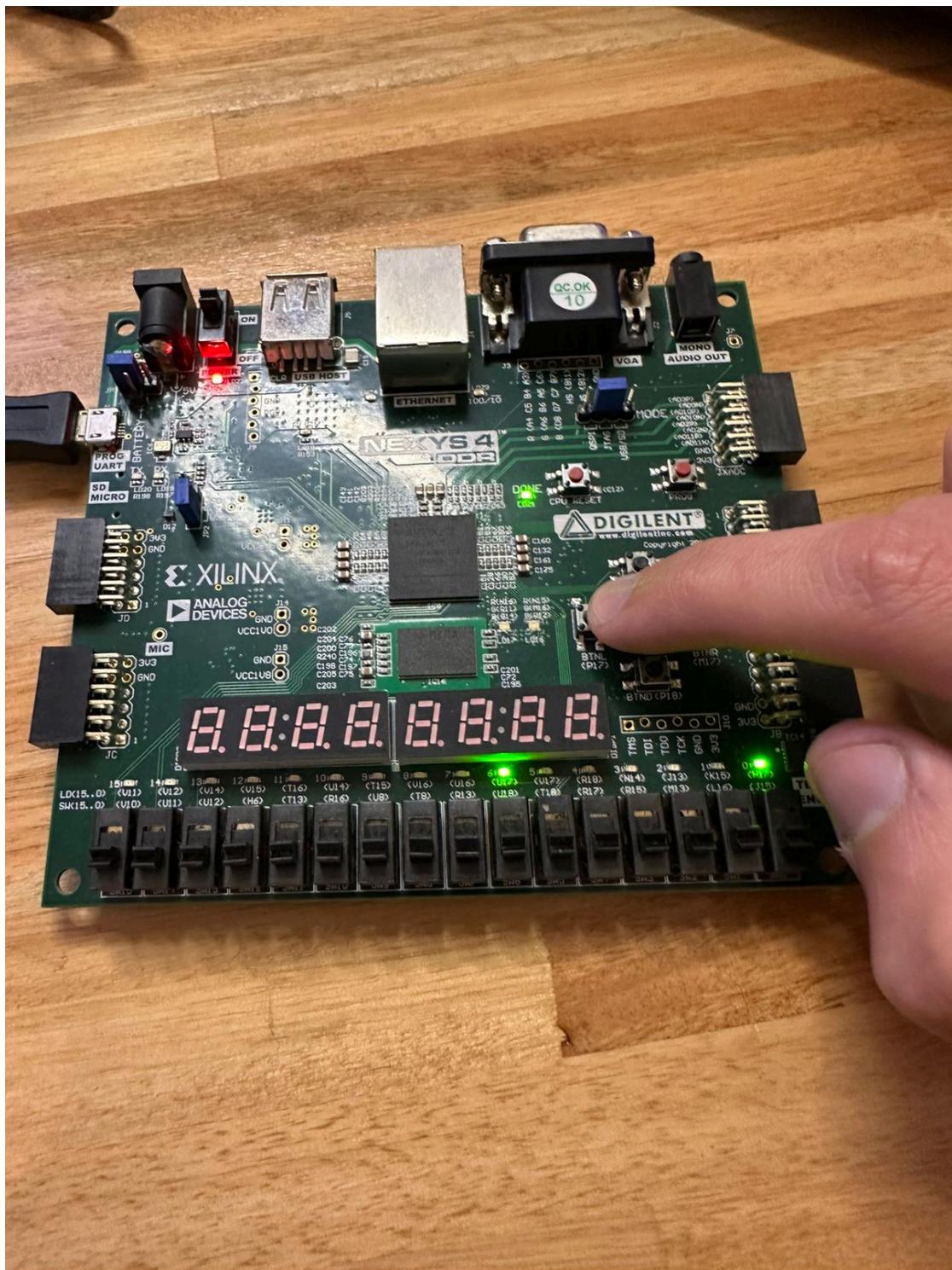
1. Descripción del programa a ejecutar

El programa que vamos a ejecutar en nuestro procesador segmentado tipo MIPS está compuesto por las siguientes instrucciones en lenguaje ensamblador (.asm). Este programa será compilado utilizando un script de Python ubicado en utils/compiler.py, que convertirá el código ensamblador en un formato adecuado para ser cargado en la memoria de instrucciones del procesador:

```
ADDI r13 r13 13
ADDI r1 r1 1
ADDI r3 r3 3
ADDI r5 r5 5
JAL 7
ADDI r7 r7 7
ADDI r9 r9 9
LUI r11 10
LUI r12 10
ADDI r2 r2 2
JALR r12 r13
ADDI r4 r4 4
ADDI r6 r6 6
ADDI r8 r8 6
HALT      You,
```

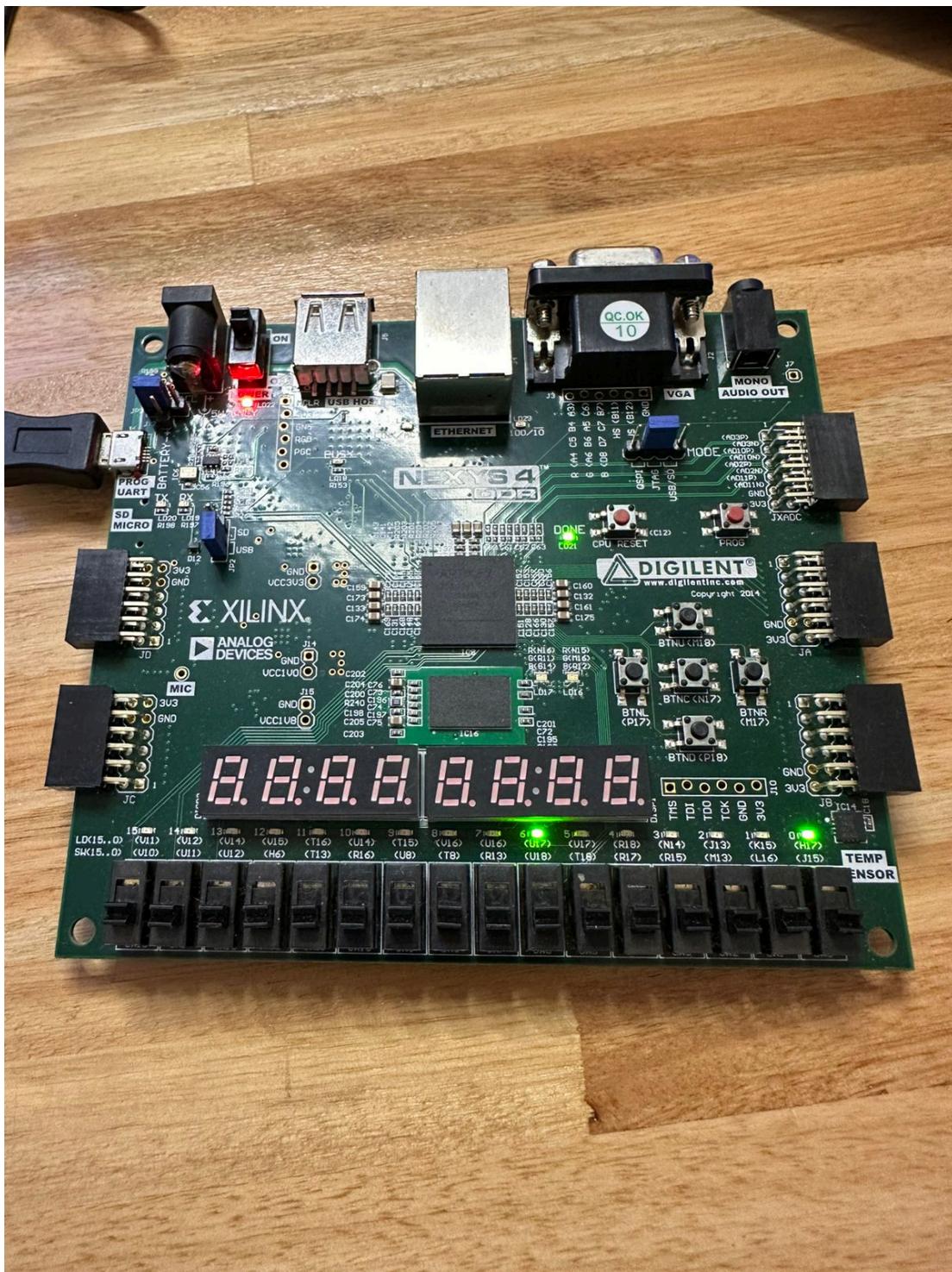
2. Reinicio del sistema

Para iniciar la ejecución, presionamos el botón P17, que activa el proceso de reinicio del sistema. Este reset garantiza que todos los registros, memorias y componentes del procesador se encuentren en su estado inicial, permitiendo una ejecución limpia del programa.



3. Verificación del estado inicial

Una vez realizado el reinicio, la placa se encuentra en su estado inicial (UNINITIALIZED). En este estado, todos los registros y memorias están sin inicializar



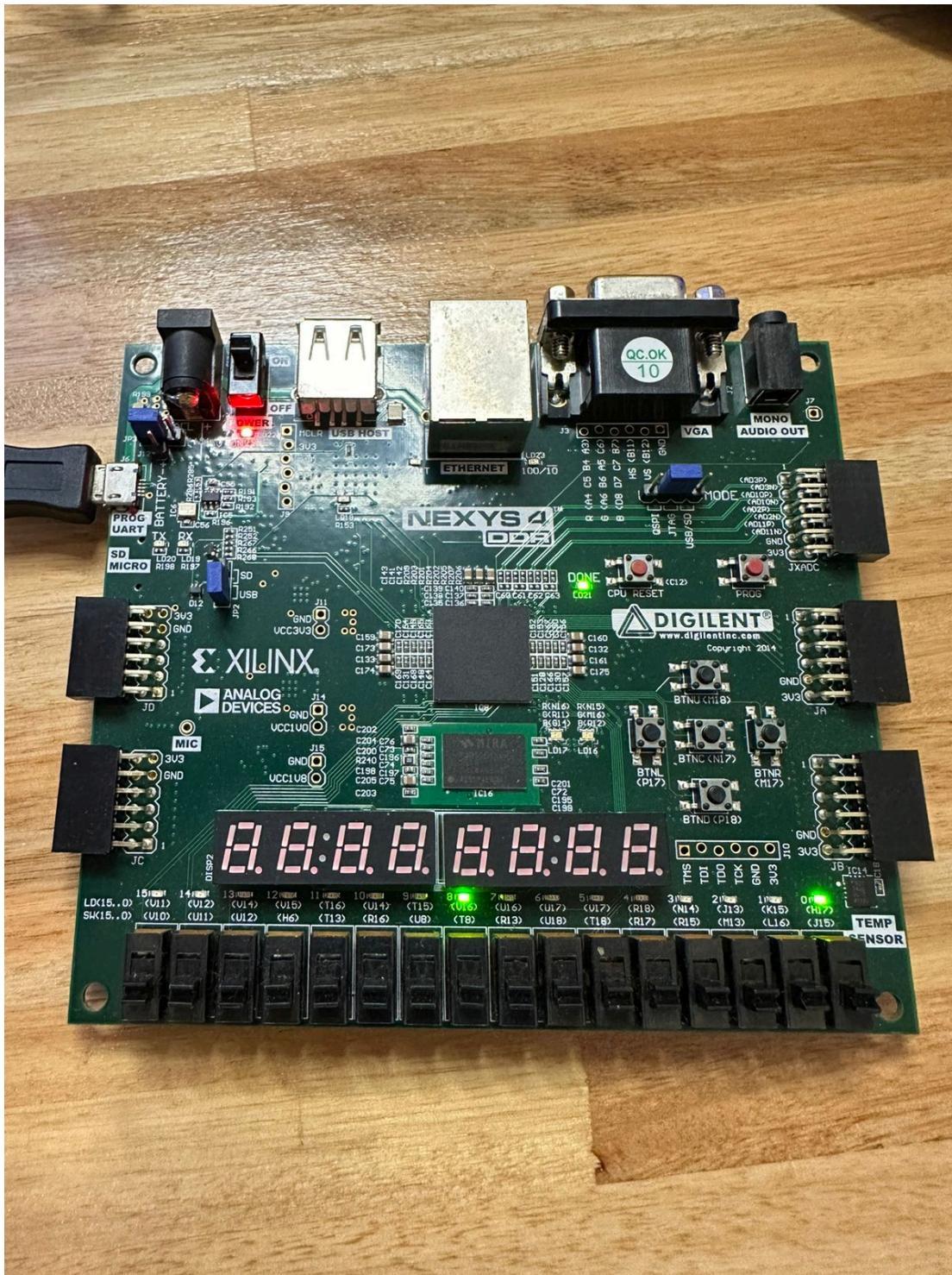
4. Carga del programa

Procedemos a cargar el programa en la memoria de instrucciones del procesador. Para ello, utilizamos la interfaz UART y enviamos el comando CMD_SET_INST. Primero, enviamos un byte que indica la cantidad de instrucciones que se van a cargar, seguido de las instrucciones del programa, una a una. Para facilitar esta comunicación con la placa, utilizaremos otro script ubicado en `utils/serial-port.py`, que gestiona la transferencia de datos entre la PC y la placa a través de la UART.

```
PS D:\Facultad\ArquitecturaComputadoras\TpFinal> python .\utils\serial-port.py
Using port: COM8
Enter command (type 'exit' to quit): CMD_SET_INST
Enter the path to the binary program file: .\jal-jalr_bin.txt
Sent line count: 15
Sent: b'\r\x00\xad!'
Sent: b'\x01\x00!'
Sent: b'\x03\x00c'
Sent: b'\x05\x00\xa5'
Sent: b'\x07\x00\x00\x0c'
Sent: b'\x07\x00\xe7'
Sent: b'\t\x00)!'
Sent: b'\n\x00\x0b<'
Sent: b'\n\x00\x0c<'
Sent: b'\x02\x00B'
Sent: b'\t`\xa0\x01'
Sent: b'\x04\x00\x84'
Sent: b'\x06\x00\xc6'
Sent: b'\x06\x00\x08!'
Sent: b'\xff\xff\xff\xff'
Enter command (type 'exit' to quit): $
```

5. Espera del modo de ejecución

Una vez que la placa recibe todas las instrucciones, esta entra en el estado SET_MODE. En este estado, el procesador espera la indicación del modo de ejecución, que puede ser en modo continuo o paso a paso.



6. Configuración del modo continuo y ejecución

Decidimos ejecutar el programa en modo continuo, por lo que enviamos el comando CMD_SET_CONTINUOUS a través de la UART. En este modo, el procesador ejecuta el programa cargado hasta encontrar la instrucción HALT, la cual detiene la ejecución. Tras finalizar, la Debug Unit envía a la PC, nuevamente por UART, el estado final del program counter, los registros, y el contenido de la memoria de datos, permitiendo así una completa verificación del estado final del sistema.

```
Enter command (type 'exit' to quit): Program Counter (Hex): 0000000E
Registers:
R0: 00000000
R1: 00000001
R2: 00000002
R3: 00000003
R4: 00000000
R5: 00000005
R6: 00000000
R7: 00000000
R8: 00000006
R9: 00000000
R10: 00000000
R11: 000a0000
R12: 0000000c
R13: 0000000d
R14: 00000000
R15: 00000000
R16: 00000000
R17: 00000000
R18: 00000000
R19: 00000000
R20: 00000000
R21: 00000000
R22: 00000000
R23: 00000000
R24: 00000000
R25: 00000000
R26: 00000000
R27: 00000000
R28: 00000000
R29: 00000000
R30: 00000000
R31: 00000006
Memory:
Mem[0]: 00000000
Mem[1]: 00000000
Mem[2]: 00000000
Mem[3]: 00000000
Mem[4]: 00000000
Mem[5]: 00000000
Mem[6]: 00000000
```

Al finalizar la ejecución, ciertos registros se modificarán conforme a las instrucciones ejecutadas. Específicamente, las instrucciones ADDI y LUI afectarán los registros según lo establecido, configurando sus valores según los operandos proporcionados.

Sin embargo, debido a la correcta ejecución de las instrucciones de salto JAL y JALR, algunas de las instrucciones ubicadas después de estos saltos no se ejecutarán. Como resultado, los registros involucrados en esas instrucciones conservarán sus valores iniciales y no reflejarán cambios.

Como resultado, la placa quedará en estado HALTED, indicando que el programa ha terminado su ejecución de manera exitosa.

