



ARQUITECTURA DE COMPUTADORAS

Trabajo Práctico 3

AÑO 2024

Profesores:

- Santiago Rodriguez
- Martin Pereyra

Alumnos:

- | | |
|-------------------------|----------|
| ➤ Sebastian Klinevitzky | 41158451 |
| ➤ Ivo Ferrari | 40326730 |

Índice

1 - Introducción	3
2 - Objetivos	4
3 - Desarrollo	5
Instruction Fetch	5
Latch - IF/ID_reg	7
Instruction Decode	7
Latch - ID/EX_reg	10
Execute	10
Latch - EX/MEM_reg	14
Memory	14
Latch -MEM/WB_reg	15
Write Back	15
Forwarding Unit	16
Unit Stall	16
4 - Unidad de Debug	17
Diagrama de estados debug unit	17
Detalle en ejecución PASO A PASO:	17
En ejecución CONTINUA:	18
Ejecución PASO A PASO que pasa a CONTINUA:	19
5- Uso del Clock	19
6 - Interfaz de Usuario	20
Requisitos	20
Descripción de la interfaz	20
Uso paso a paso	21
7 - Ejecución con la placa	25
8 - Implementation y análisis de la frecuencia de clock	28

1 - Introducción

La arquitectura MIPS (Microprocessor without Interlocked Pipeline Stages) es una familia de microprocesadores diseñada con un tipo de arquitectura RISC.

Esta arquitectura tiene en mente los siguientes principios

- Segmentación: Divide la ejecución en varias etapas
- Número fijo de registros: Cuenta con 32 registros de propósito general de 32 bits
- Operaciones en memoria: Carga y Almacenamiento
- Hay múltiples instruction sets, conocidos como MIPS I, MIPS II, MIPS III, MIPS IV y MIPS 32/64
- Existen 3 formatos principales, los de tipo **R**, **I**, **J**

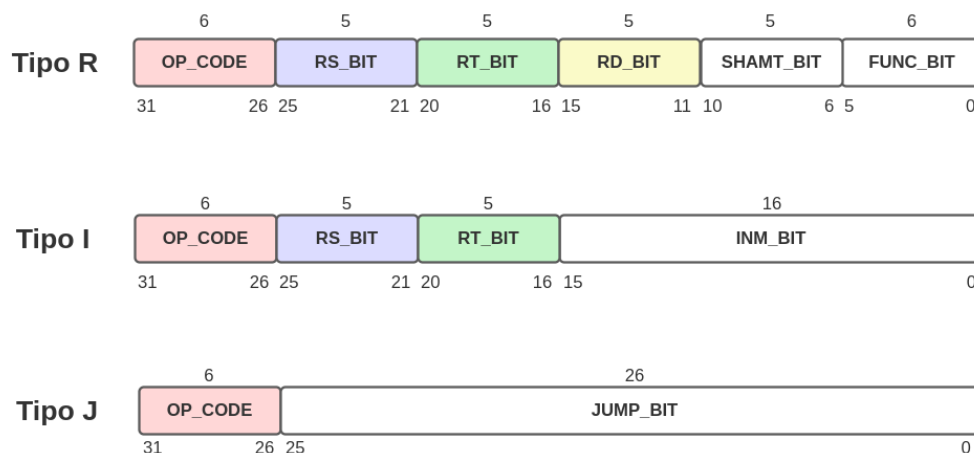
2 - Objetivos

Se debe implementar el procesador MIPS segmentado en las siguientes etapas:

- **Instruction Fetch**
- **Instruction Decode**
- **Execute**
- **Memory Access**
- **Write back**

Las instrucciones a implementar son un subconjunto del set de instrucciones del MIPS:

- 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.



La pipeline debe implementar detección y control de riesgos. Se debe implementar una unidad de debug que envíe información hacia y desde el procesador mediante UART. La información es la siguientes:

- Contador de programa (PC).
- Contenido de los 32 registros.
- Contenido de la memoria de datos usada.

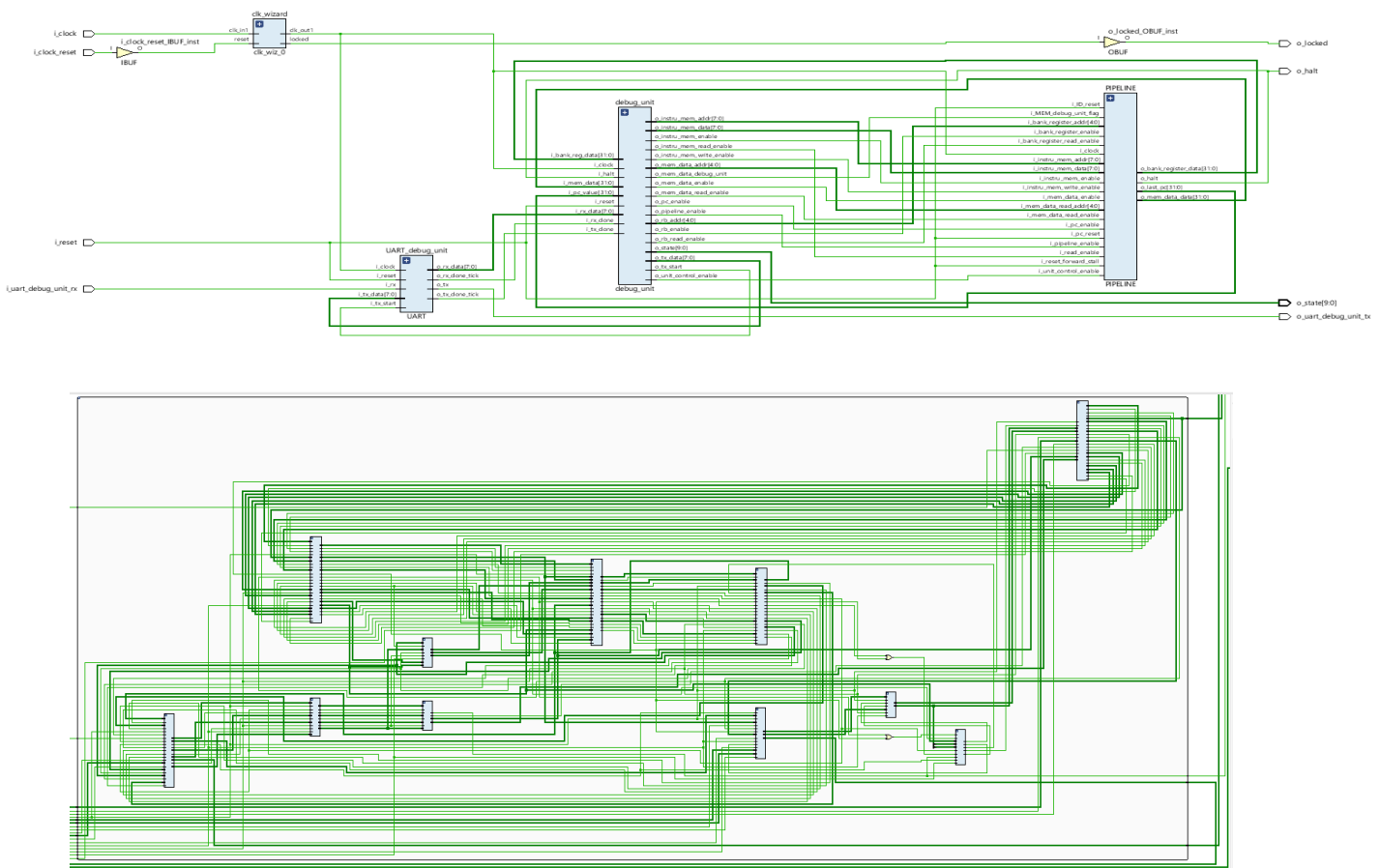
Una vez cargado el programa a ejecutar, el procesador debe permitir dos modos de operación:

- **Paso a paso:** se envía un comando a la FPGA por la UART, se ejecuta un ciclo de clock y se muestra la información pedida. Seguido de recibir nuevamente un comando ya sea un ciclo más o en modo continuo hasta que termine de ejecutarse todo el programa.

- **Continuo:** se envía un comando a la FPGA por la UART y se inicia la ejecución del programa hasta llegar al final del mismo (instrucción HALT). Luego, se muestra la información pedida en la pantalla.

Se debe incluir un ensamblador para convertir el lenguaje Assembly a código de máquina y se debe transmitir mediante UART.

3 - Desarrollo



Las imágenes que se ven son los módulos del trabajo completo en la primer imagen, y del Pipeline en la segunda.

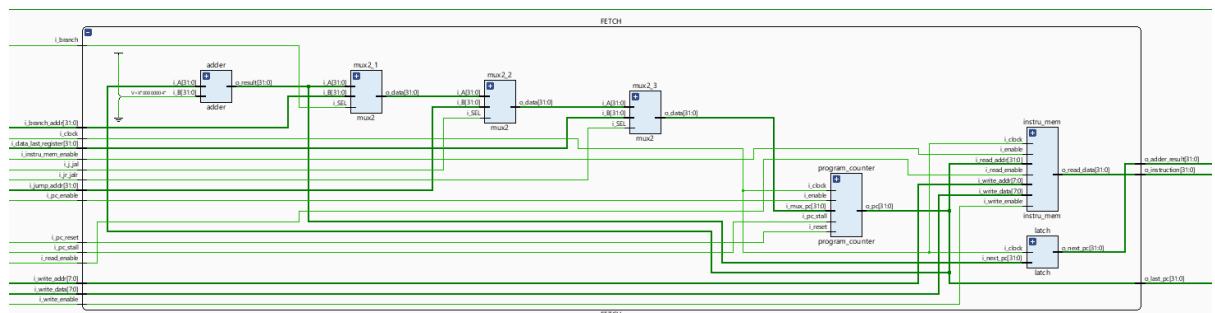
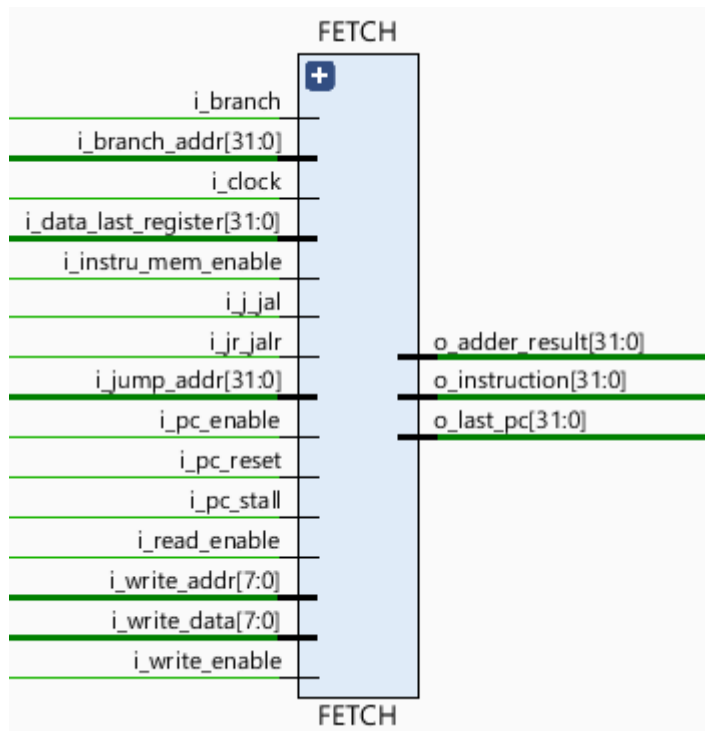
A continuación vamos a proceder a explicar con más detalle alguna de sus partes.

Instruction Fetch

Esta etapa es la encargada de leer las instrucciones de la memoria de instrucciones. Para ello utiliza la dirección almacenada en el Contador del Programa (PC).

La dirección se incrementa en 4 y se escribe de nuevo en el PC para prepararse para el siguiente ciclo de reloj. En caso de haber algún salto, existen señales de control que a través de multiplexores indican la siguiente dirección de memoria de otro origen (por ejemplo de la ALU).

Tanto la instrucción como la dirección del PC se almacenan en el registro de IF/ID por si alguna instrucción, por ejemplo BEQ, la necesita con posterioridad.



Consta de distintos submódulos:

- Adder: Suma el pc+4
- 3 multiplexores que envían la data a el program counter
 - Los multiplexores seleccionan entre la salida del adder, o un valor del branch, o algún salto.
- Program counter: Contador de programa que almacena la dirección de la siguiente instrucción, retorna la dirección de la siguiente instrucción o se mantiene el valor si recibe un stall (burbuja)
- Instruction Memory: Es una implementación de la memoria de instrucciones, devuelve los datos leídos desde la memoria de instrucciones.

- Esta memoria está hecha de 8×256 , dado que los datos que le llegan de la UART están limitados a 8 bits. Luego dentro del módulo, se concatenan de a 4, para tener una instrucción de 32.
- Latch: Registro que captura y retiene el valor de la señal de entrada en el flanco de subida del reloj, retorna el valor capturado. El valor que se guarda es $pc+4$ para tener un registro de cual es en caso de que se fetchee otra cosa por los mux

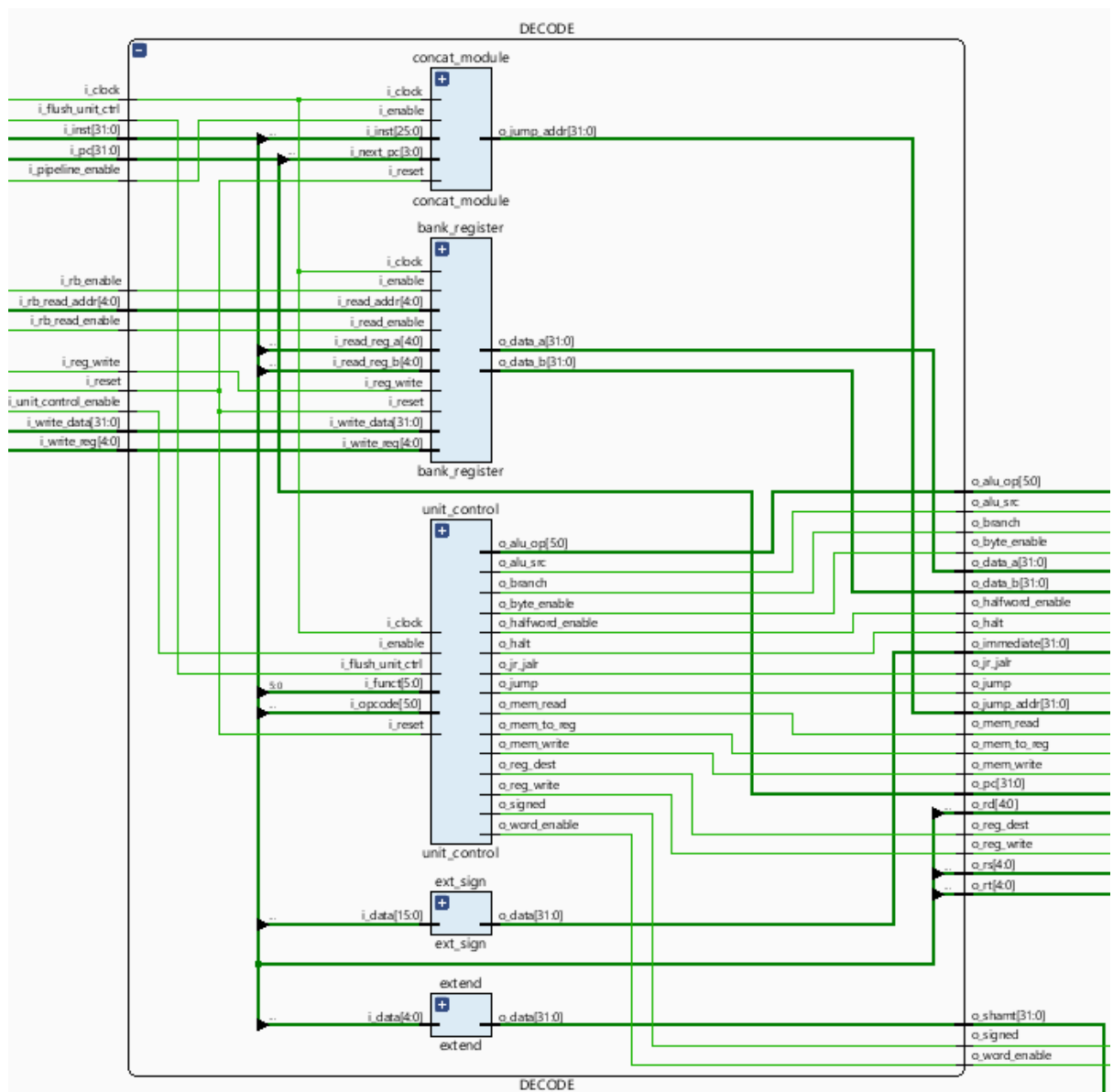
Latch - IF/ID_reg

- Actúa como registro intermedio entre las etapas de Instruction Fetch e Instruction Decode.
- Retiene la salida de la etapa de IF y la pasa a ID en el siguiente ciclo de clock.
- Es necesario para hacer cumplir las burbujas o las medidas de control de riesgo.

Instruction Decode

Esta etapa es la encargada de leer las instrucciones provistas por la etapa de Instruction Fetch y decodificar lo que la función debe hacer obteniendo operandos y señales de control. Utiliza el banco de registros junto a unidades funcionales que proporcionan el control de flujo y de datos del procesador.





Tiene sub módulos como:

- Concat module: concatena los valores necesarios para determinar la dirección de salto. El valor de salida está determinado por como indica *MIPS IV Instruction set* para el jump:
 - $pc = next_pc$ (los 4 bits más significativos) + instruction + 00
- Register Bank: representa el banco de registros del procesador. Permite leer 2 registros (rt y rs) y escribir en uno (rd).
 - Es de 32x32.
 - Al iniciar el programa, llenamos todos los registros con 0.
 - Además de permitir la lectura y escritura de los registros, tiene condiciones para evitar riesgos (escritura después de lectura).
- Control Unit: se encarga de generar señales de control en función del tipo de instrucción. Recibe el opcode y el function code como entradas. Está atento a el valor que pueda recibir de reset o flush para borrar el pipeline

- Extend_sign: se utiliza para extender la instrucción, transforma el número de 2 bytes a 4, manteniendo el valor del signo.
- Extend_bus: recibe 5 bits y le agrega 27 0s sin cambiar el valor, para hacerlo así de 32

Latch - ID/EX_reg

- Actúa como registro intermedio entre las etapas del Instruction Decode y Execute.
- Toma múltiples señales de entrada provenientes de la etapa de ID, como los valores de los registros, señales de control para operaciones ALU, control de memoria, control de salto, etc., y las retiene hasta que el reloj realiza una transición negativa, lo que permite que estas señales se pasen a la etapa de EX

Execute

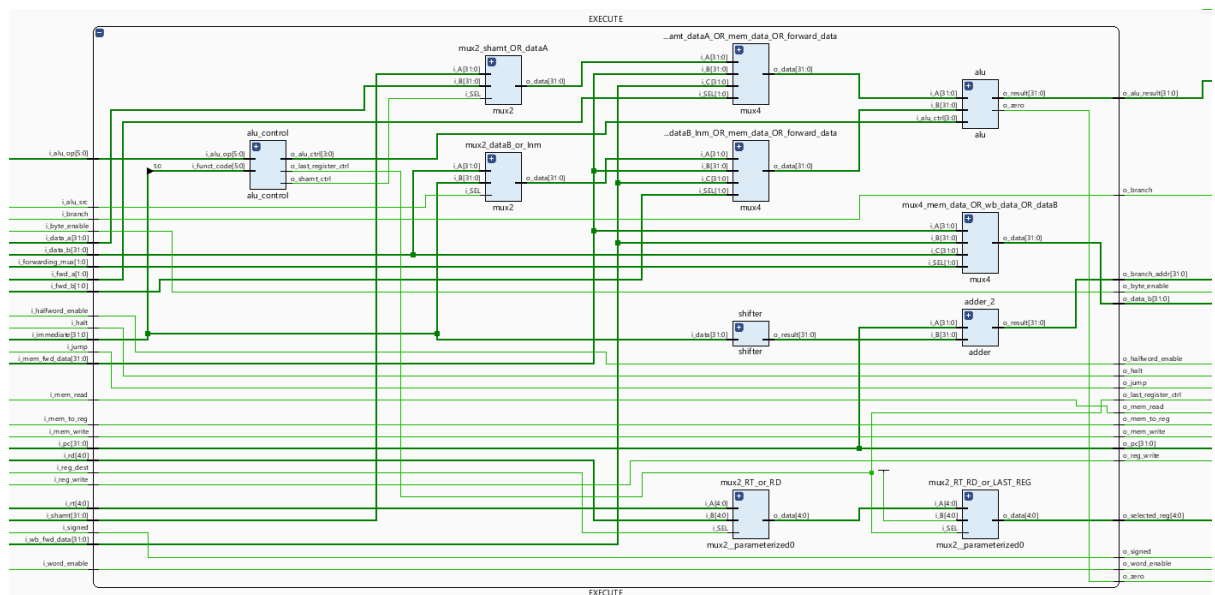
La etapa de ejecución del procesador es gestionada por el módulo EXECUTE, que se encarga de realizar operaciones aritméticas, lógicas y de control de flujo necesarias para ejecutar una instrucción dada. Es una parte crucial del procesador, encargada de ejecutar instrucciones y determinar los resultados de las operaciones. Durante esta etapa, se interactúa con otros módulos del procesador, como el módulo de memoria (MEM) y el módulo de escritura en registros (WB), para completar la ejecución de instrucciones.

Esta etapa es quien más utiliza junto al Instruction Decode, los opcodes y los function code, aquellos usados en este trabajo, se pueden observar en la imagen siguiente.

R-Type		
OPERATION	OPCODE	FUNCTION
AND	000000	100100
OR	000000	100101
XOR	000000	100110
NOR	000000	100111
ADDU	000000	100001
SUBU	000000	100011
SLT	000000	101010
SRL	000000	000010
SLL	000000	000000
SRA	000000	000011
SLLV	000000	000100
SRLV	000000	000110
SRAV	000000	000111

J-Type		
OPERATION	OPCODE	FUNCTION
JR	000000	001000
JALR	000000	001001

I-Type	
OPERATION	OPCODE
LB	100000
LH	100001
LW	100011
LWU	100111
LBU	100100
LHU	100101
SB	101000
SH	101001
SW	101011
ADDI	001000
ANDI	001100
ORI	001101
XORI	001110
LUI	001111
SLTI	001010
BEQ	000100
BNE	000101
J	000010
JAL	000011



Posee los distintos submódulos:

- **Alu control:** Se encarga de generar señales de control que indican a la ALU qué operación debe realizar, según el tipo de instrucción que se esté ejecutando.
- **Shifter:** Hace un shift a la izquierda en 2 (desplazamiento a la izquierda)
- **Adder:** Suma el PC+4, está en este módulo para calcular la dirección de salto cuando hay un branch
- **Alu:** componente utilizado en la etapa de ejecución de un procesador para llevar a cabo operaciones aritméticas y lógicas en datos. Este submódulo toma dos operadores, realiza una operación basada en una señal de control y produce un resultado.
 - En la tabla siguiente se pueden ver operaciones que realiza junto al código que lo representa.
- **Multiplexores** que terminan eligiendo de donde vienen los datos que usará la Alu
 - Para una de las entradas, los multiplexores deciden si toma el dato normal (puede ser un valor como puede ser el valor de desplazamiento), o si lo toma por una redirección del memory o del write back.
 - **Entrada A de la Alu**
 - La otra de las entradas elige el valor normal (valor del dato o el immediate), o una redirección del memory o del write back.
 - **Entrada B de la Alu**
 - Tenemos uno que se encarga de, de ser necesario, transmitir el valor que utilizara JAL o JALR.
 - El último nos da el dato a escribir en memoria, si es que hiciera falta.

OPERACIÓN	OPCODE
SLL/SLLV	hx0
SRL/SRLV	hx1
SRA/SRAV	hx2
ADD	hx3
SUB	hx4
AND	hx5
OR	hx6
XOR	hx7
NOR	hx8
SLT	hx9

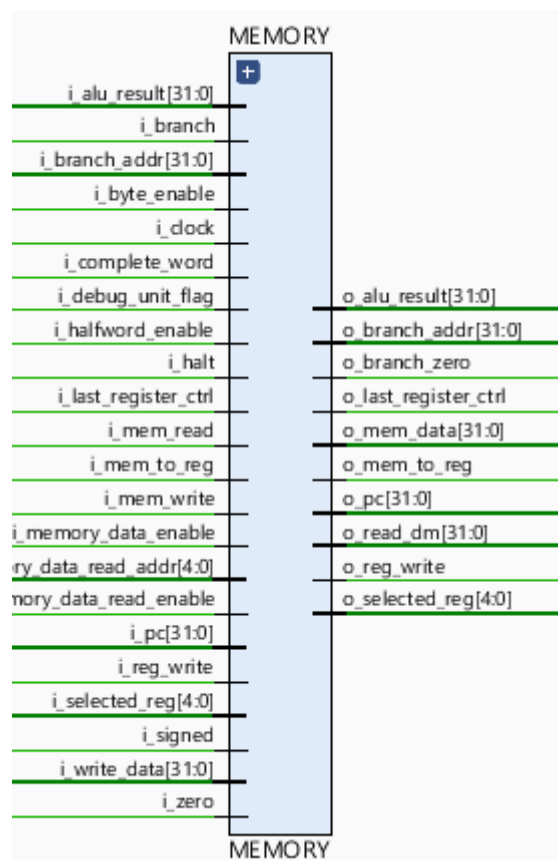
SLL16	hxa
BEQ	hxb
BNEQ	hxc

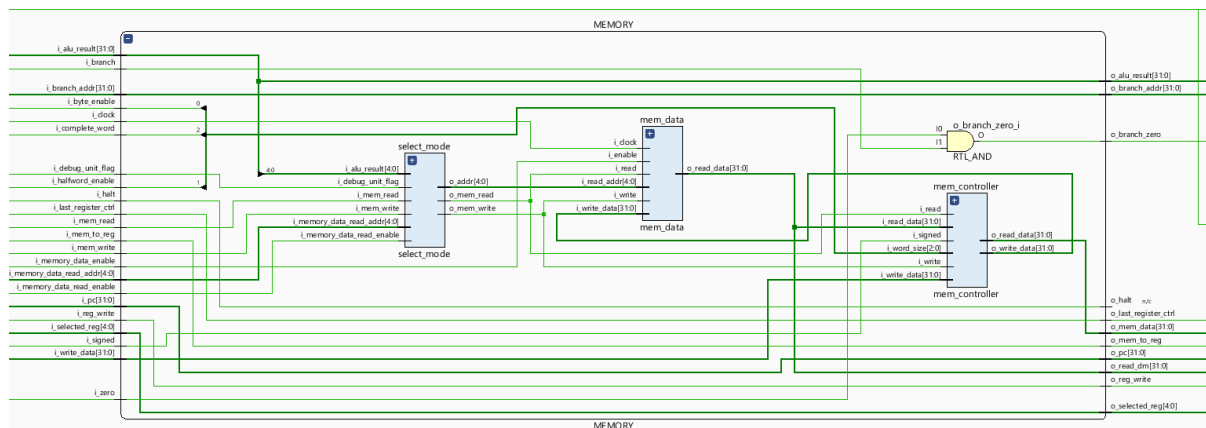
Latch - EX/MEM_reg

- Actúa como registro intermedio entre las etapas de Execute y Memory
- Actúa como un buffer que asegura que los resultados generados durante la etapa de ejecución se mantengan estables y se propaguen a la siguiente etapa.
- Garantiza que los resultados y las señales de control de la etapa de ejecución se pasen de manera correcta y sin alteraciones a la etapa de acceso a memoria, ayudando a mantener la integridad del flujo de datos en el pipeline del procesador.

Memory

La etapa de memory tiene como función principal permitir la escritura y lectura de datos en la memoria, además de proporcionar salidas y controlar el flujo de datos. Puede habilitar la lectura y escritura de datos en la memoria.





Tiene distintos submódulos:

- **Memory controller:** Controla las operaciones de lectura y escritura en la memoria. Maneja ajustes de datos en función de las señales habilitadas de ancho de palabra y si la operación de lectura se realiza con signo o sin signo.
- **Memory data:** Gestiona la manipulación de datos dentro de la memoria. Maneja operaciones de lectura y escritura y controla las operaciones habilitadas.
 - Este módulo es quien contiene a la memoria de 32x32
- **Select mode:** Este módulo sirve para elegir entre la debug unit para leer o el funcionamiento normal de la pipeline.
- También se encuentra una AND entre señales de entrada de un zero y un branch que se dirige hacia el Instruction Fetch (como `o_branch_zero`).

Latch -MEM/WB_reg

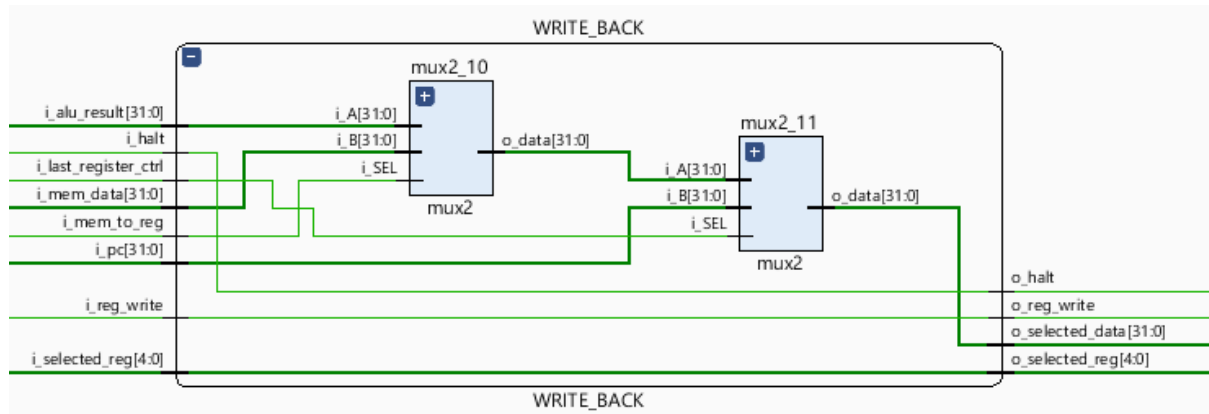
- Actúa como registro intermedio entre las etapas del Memory y Write Back.
- Es el último latch en el pipeline, y su función principal es almacenar los resultados de las operaciones de memoria y ALU para escribirlos de vuelta en los registros del procesador.
- Asegura que los datos y señales de control de la etapa de acceso a memoria (MEM) sean correctamente almacenados y transferidos a la etapa de escritura (WB), donde los resultados finales de las operaciones de memoria o ALU serán escritos en los registros del procesador.

Write Back

Es una parte esencial de la etapa de escritura de un pipeline en un procesador. Su función principal es determinar si se realizará una escritura en los registros del procesador y seleccionar los datos que se escribirán.

Toma varias entradas, incluida la decisión de escribir en los registros y la selección de datos para la escritura.

Las salidas de este módulo incluyen señales para controlar la escritura en los registros, los datos seleccionados para escribir en los registros, el número de registro seleccionado y una señal de detención opcional.

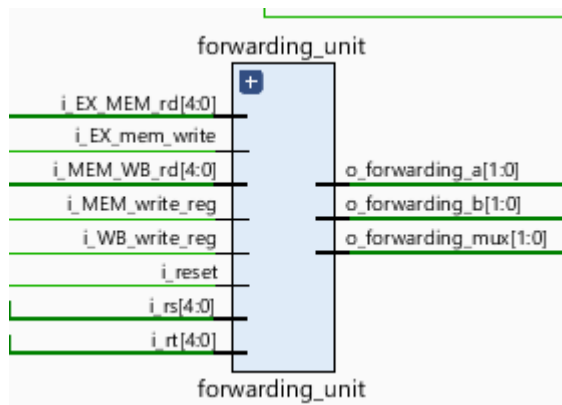


Tiene los distintos multiplexores:

- Un multiplexor (**mux2_10**) que decide si los datos de la memoria o el resultado de la ALU se deben escribir en los registros.
- Otro multiplexor (**mux2_11**) que selecciona los datos que se escribirán en los registros entre los datos de salida del primer multiplexor y el contador de programa.

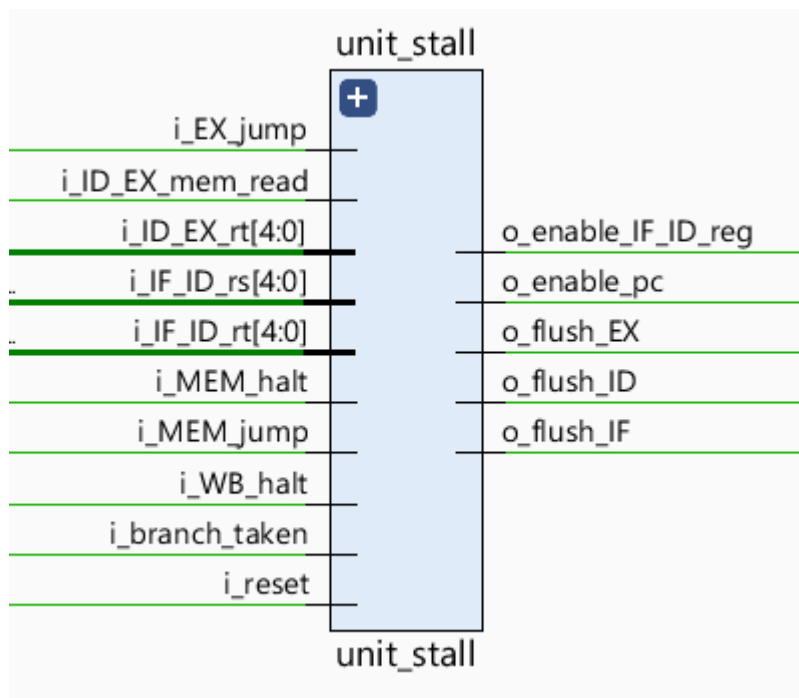
Forwarding Unit

Este módulo recibe la información necesaria para determinar si es necesario para una instrucción que haya un cortocircuito de alguna fuente para evitar riesgos de datos.



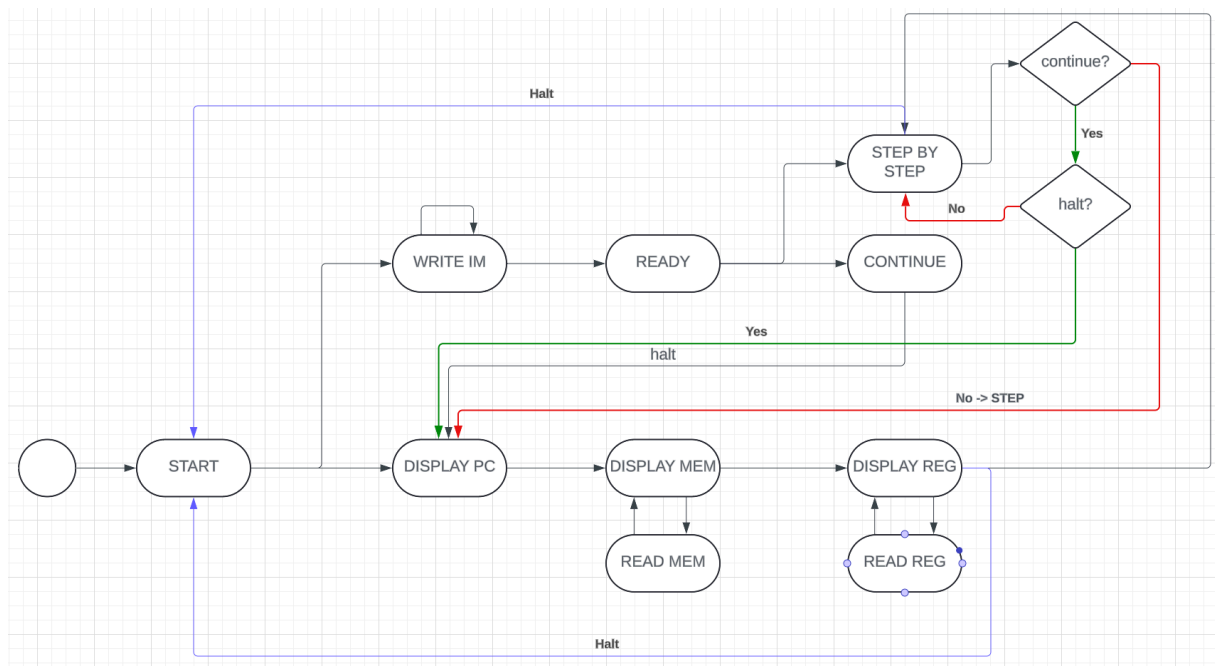
Unit Stall

Este módulo recibe la información necesaria para determinar si es necesario para una instrucción que haya una burbuja para evitar riesgos de datos y control.



4 - Unidad de Debug

Diagrama de estados debug unit



Detalle en ejecución **PASO A PASO:**

Estado	Descripción
START	Estado inicial donde la debug unit espera comandos. Entra en este estado al cargarse el programa en la placa. Cuando hacemos click en Iniciar en la aplicación, enviamos un 1 lógico por la UART y pasamos al estado de WRITE_IM
WRITE_IM	En este estado se carga todo el programa compilado por la aplicación a la memoria de instrucciones y una vez cargado pasa al estado READY
READY	Se espera el modo de ejecución del programa, en este caso, se selecciona en el programa el botón de STEP, que provoca el envío de un 3, que lo hace pasar al estado STEP BY STEP
STEP BY STEP	Este estado espera la llegada de algún comando por UART (puede ser CMD_STEP o CMD_CONTINUE). Cuando llega un dato, habilita el pipeline y en ese caso pasamos un 4 (STEP), y pasamos a DISPLAY PC . En caso de recibir la señal de que la instrucción es un halt, de aquí vuelve a START .
DISPLAY PC	Enviamos el valor del PC el cual es visualizado en la aplicación (se visualiza arriba en Decimal y Hexadecimal), para posteriormente ir al estado DISPLAY MEM . Este estado deshabilita al resto del pipeline para evitar que siga su ejecución, y activa la UART.
DISPLAY MEM	Envía la memoria de datos para que se visualice en la aplicación. Después de haber enviado toda la memoria, pasa el estado DISPLAY REG , luego de haber estado iterando entre DISPLAY MEM y READ MEM
READ MEM	Durante este estado es cuando se lee la memoria para poder pasarla en el anterior estado.
DISPLAY REG/READ REG	Igual que DISPLAY MEM y READ MEM , pero con la memoria de registros. Como en este ejemplo estamos siguiendo STEP BY STEP volvemos a dicho estado.

En ejecución **CONTINUA**:

Estado	Descripción
START	Estado inicial donde la debug unit espera comandos. Entra en este estado al cargarse el programa en la placa. Cuando hacemos click en Iniciar en la aplicación, enviamos un 1 lógico por la UART y pasamos al estado de WRITE_IM
WRITE IM	En este estado se carga todo el programa compilado por la aplicación a la memoria de instrucciones y una vez cargado pasa al estado READY

READY	Se espera el modo de ejecución del programa, en este caso, se selecciona en el programa el botón de CONTINUE , que provoca el envío de un 2, que lo hace pasar al estado CONTINUE
CONTINUE	Habilita todas las etapas del pipeline y se queda a la espera de la señal de que se llegó al halt, en ese caso, se procede a DISPLAY_PC
DISPLAY PC	Enviamos el valor del PC el cual es visualizado en la aplicación (se visualiza arriba en Decimal y Hexadecimal), para posteriormente ir al estado DISPLAY MEM . Este estado deshabilita al resto del pipeline para evitar que siga su ejecución, y activa la UART.
DISPLAY MEM	Envía la memoria de datos para que se visualice en la aplicación. Después de haber enviado toda la memoria, pasa el estado DISPLAY REG , luego de haber estado iterando entre DISPLAY MEM y READ MEM
READ MEM	Durante este estado es cuando se lee la memoria para poder pasarla en el anterior estado.
DISPLAY REG/READ REG	Igual que DISPLAY MEM y READ MEM , pero con la memoria de registros. Como en este ejemplo estamos siguiendo la ejecución continúa, de aquí vuelve a START .

Ejecución **PASO A PASO** que pasa a **CONTINUA**:

(Nos saltamos las partes iguales que en el **PASO A PASO**, para explicar la parte que cambia)

Estado	Descripción
STEP BY STEP	<p>Este estado espera la llegada de algún comando por UART (puede ser CMD_STEP o CMD_CONTINUE). Cuando llega un dato, habilita el pipeline y en ese caso pasamos un 2 (CONTINUA).</p> <p>Cuando esto sucede, entra en un bucle dentro del mismo estado de STEP BY STEP, que no se rompe, hasta que llega la señal de la instrucción del halt. El pipeline funciona hasta el final, dado que nunca se deshabilita el pipeline activado.</p> <p>Cuando llega el halt, pasamos a DISPLAY PC, y al igual que en la ejecución continua, vuelve a START desde DISPLAY REG</p>

5- Uso del Clock

En el trabajo, algunos módulos trabajan en Posedge, y otros en Nedge.

El término posedge indica que el bloque always debe activarse cuando la señal de reloj o cualquier otra señal hace una transición de nivel bajo (0) a nivel alto (1). En el caso del reloj, esta transición corresponde a la "subida" del ciclo de reloj, lo que comúnmente se denomina "flanco ascendente".

El término negedge indica que el bloque always debe activarse cuando la señal de reloj o cualquier otra señal hace una transición de nivel alto (1) a nivel bajo (0). En el caso del reloj, esta transición corresponde a la "bajada" del ciclo de reloj, también conocida como "flanco descendente".

En general los latches intermedios y la debug unit trabajan en nededge, mientras que el resto en posedge. Esto sucede porque en el pipeline la escritura y lectura de datos no se hacen en el mismo momento, se hace primero la escritura y después la lectura y los latches son quienes se encargan de escribir la señales de control y los valores a utilizar por las etapas posteriores (al igual que la debug unit), mientras que el resto de la pipeline hace uso de esos valores actualizados.

Por su lado, la debug unit además de funcionar en nededge, tiene un bloque funcionando de manera secuencial, que constan principalmente de las señales de control, asegurándose del funcionamiento deseado del MIPS.

6 - Interfaz de Usuario

Se diseñó un programa en Python para interactuar con la placa a través de una conexión UART (Universal Asynchronous Receiver/Transmitter) de una manera más sencilla y que sea de más fácil entendimiento que si fuera por vía consola. Este programa contiene funcionalidades como seleccionar un puerto UART (que son sacados de los puertos que tiene el usuario en uso), cargar un archivo de instrucciones, compilar el código assembly y al mismo tiempo enviar el programa al sistema y por último ejecutar las instrucciones cargadas previamente.

Requisitos

Necesitamos tener instalado previamente:

```
python3 -m pip install tkinter  
python3 -m pip install  
pyserialpython3 -m pip install numpy
```

Descripción de la interfaz

Selección de Puerto y Baudrate:

La ventana principal muestra un menú desplegable para seleccionar el puerto UART y otro para seleccionar la velocidad de baudios (baudrate).

Ventana de Selección de Archivo:

Si se aprieta seleccionar archivo, se abrirá una ventana que te permitirá seleccionar un archivo de instrucciones en lenguaje assembly.

En el recuadro central se muestra el código de Assembly que se pasará a la placa.

Este código será enviado a la placa al apretar el botón de iniciar, que primero enviará la señal de control a la debug unit para que pase al estado de WRITE_IM, y posteriormente le envía el programa completo.

Posteriormente pasa a una nueva ventana

Ejecución del programa:

Se abre una nueva ventana mostrando en la parte superior, el PC y Ciclo de Clock en el que se encuentra, y tres recuadros con las instrucciones del programa, la memoria de registros y la memoria de datos.

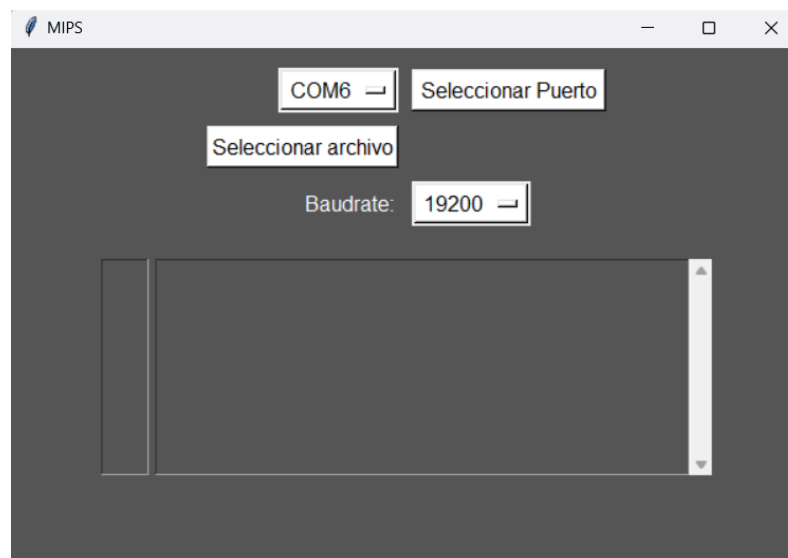
En la parte inferior de la ventana hay varios botones de acción:

- **Ejecución CONTINUA:** Inicia la ejecución continua del programa.
- **Ejecución STEP:** Ejecuta el programa paso a paso.
- **Volver:** Vuelve a la pestaña en la que te permite cargar nuevamente el programa.

Uso paso a paso

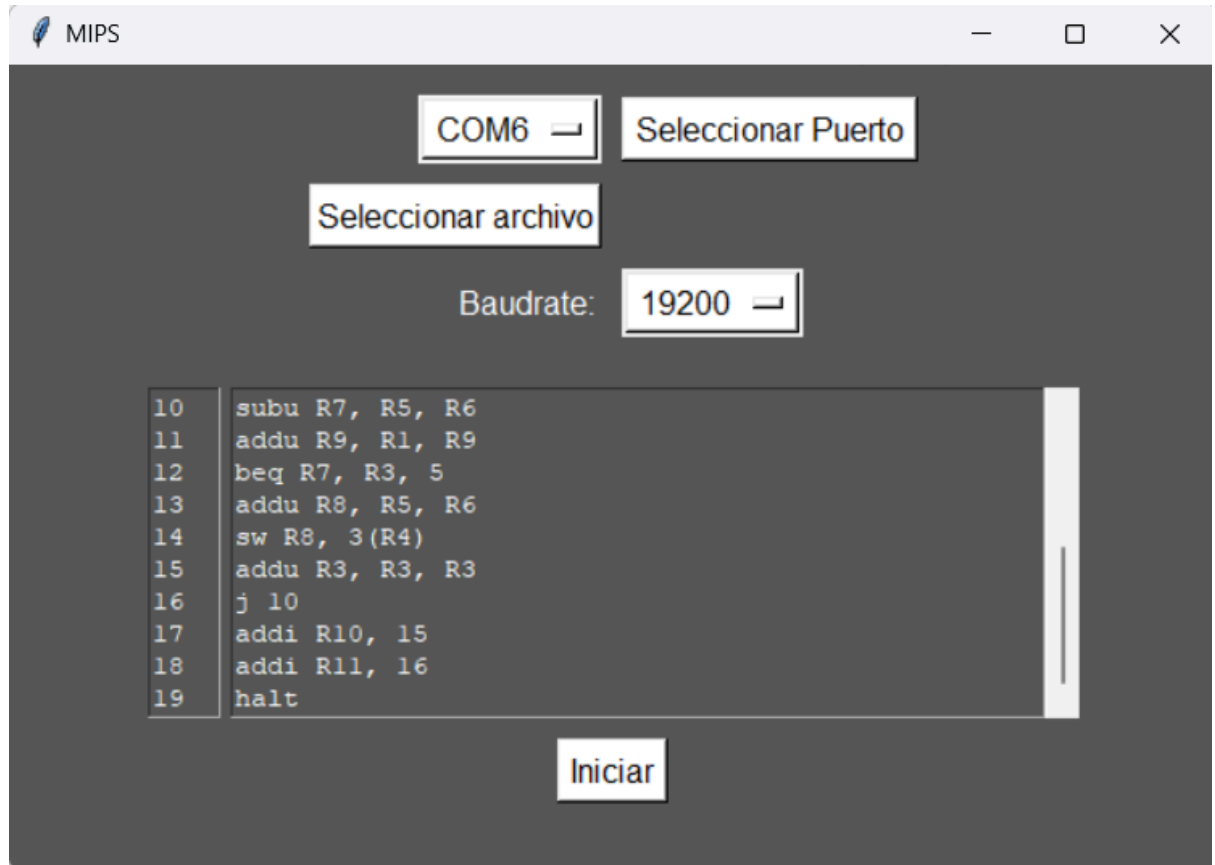
Seleccionar Puerto:

Abre la aplicación y selecciona el puerto UART y la velocidad de baudios. Haz clic en "Seleccionar Puerto".



Seleccionar Archivo de Instrucciones:

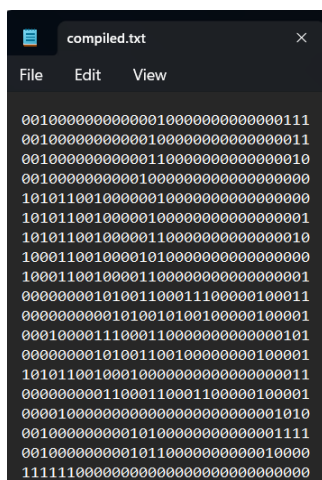
Seleccionamos aquí el archivo .txt que queremos cargar y el mismo aparecerá en la pantalla.



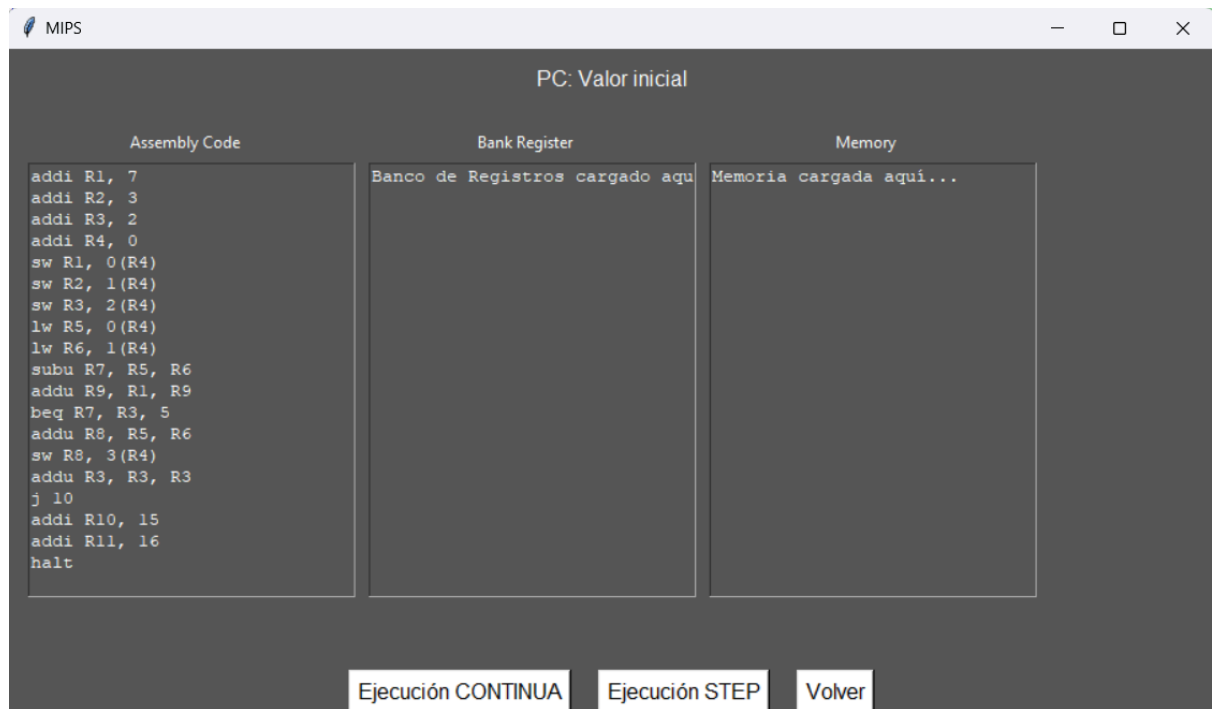
Iniciar:

Luego de cargar el archivo aparecerá el botón de **Iniciar**.

Envía las instrucciones al sistema a través del puerto UART luego de haberlas convertido de assembly en un código de instrucciones de máquina.

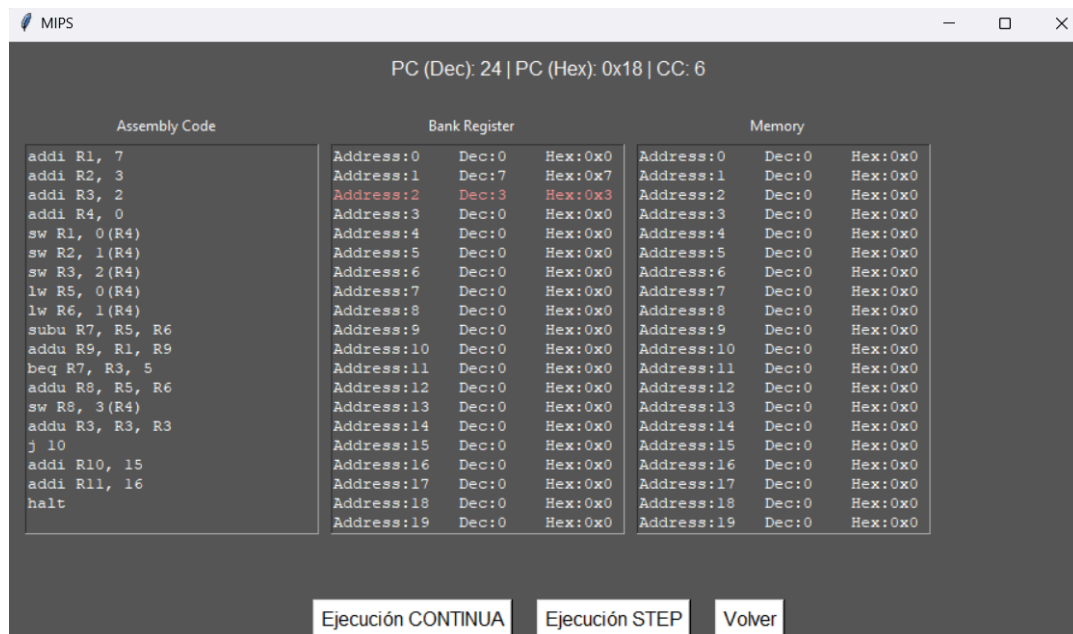


Una vez hecho esto, se abrirá la nueva ventana donde podremos seguir el funcionamiento del programa.



Ejecución STEP:

Ejecutar el programa paso a paso en el sistema. Los valores mostrados son reportados por la placa, excepto por las instrucciones que son los tomados desde el txt, y CC, que es simplemente un contador que va sumando de a 1.



Al haber un cambio en algún registro, este se resalta con rojo.

Ejecución Continua:

Ejecuta el programa de manera continua en el sistema.

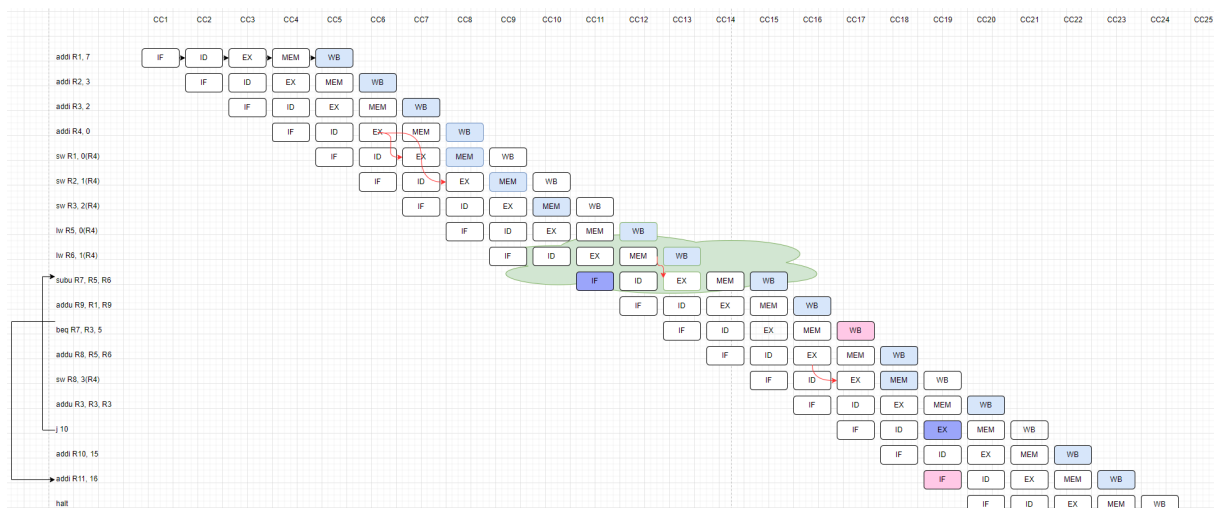
MIPS

PC (Dec): 96 | PC (Hex): 0x60 | CC: 7

Assembly Code	Bank Register	Memory
addi R1, 7	Address:0 Dec:0 Hex:0x0	Address:0 Dec:7 Hex:0x7
addi R2, 3	Address:1 Dec:7 Hex:0x7	Address:1 Dec:3 Hex:0x3
addi R3, 2	Address:2 Dec:3 Hex:0x3	Address:2 Dec:2 Hex:0x2
addi R4, 0	Address:3 Dec:4 Hex:0x4	Address:3 Dec:10 Hex:0xa
sw R1, 0(R4)	Address:4 Dec:0 Hex:0x0	Address:4 Dec:0 Hex:0x0
sw R2, 1(R4)	Address:5 Dec:7 Hex:0x7	Address:5 Dec:0 Hex:0x0
sw R3, 2(R4)	Address:6 Dec:3 Hex:0x3	Address:6 Dec:0 Hex:0x0
lw R5, 0(R4)	Address:7 Dec:4 Hex:0x4	Address:7 Dec:0 Hex:0x0
lw R6, 1(R4)	Address:8 Dec:10 Hex:0xa	Address:8 Dec:0 Hex:0x0
subu R7, R5, R6	Address:9 Dec:14 Hex:0xe	Address:9 Dec:0 Hex:0x0
addu R9, R1, R9	Address:10 Dec:0 Hex:0x0	Address:10 Dec:0 Hex:0x0
beq R7, R3, 5	Address:11 Dec:16 Hex:0x10	Address:11 Dec:0 Hex:0x0
addu R8, R5, R6	Address:12 Dec:0 Hex:0x0	Address:12 Dec:0 Hex:0x0
sw R8, 3(R4)	Address:13 Dec:0 Hex:0x0	Address:13 Dec:0 Hex:0x0
addu R3, R3, R3	Address:14 Dec:0 Hex:0x0	Address:14 Dec:0 Hex:0x0
j 10	Address:15 Dec:0 Hex:0x0	Address:15 Dec:0 Hex:0x0
addi R10, 15	Address:16 Dec:0 Hex:0x0	Address:16 Dec:0 Hex:0x0
addi R11, 16	Address:17 Dec:0 Hex:0x0	Address:17 Dec:0 Hex:0x0
halt	Address:18 Dec:0 Hex:0x0	Address:18 Dec:0 Hex:0x0
	Address:19 Dec:0 Hex:0x0	Address:19 Dec:0 Hex:0x0

Ejecución CONTINUA Ejecución STEP Volver

A continuación, una captura de como evoluciona el programa mostrado en las capturas anteriores.

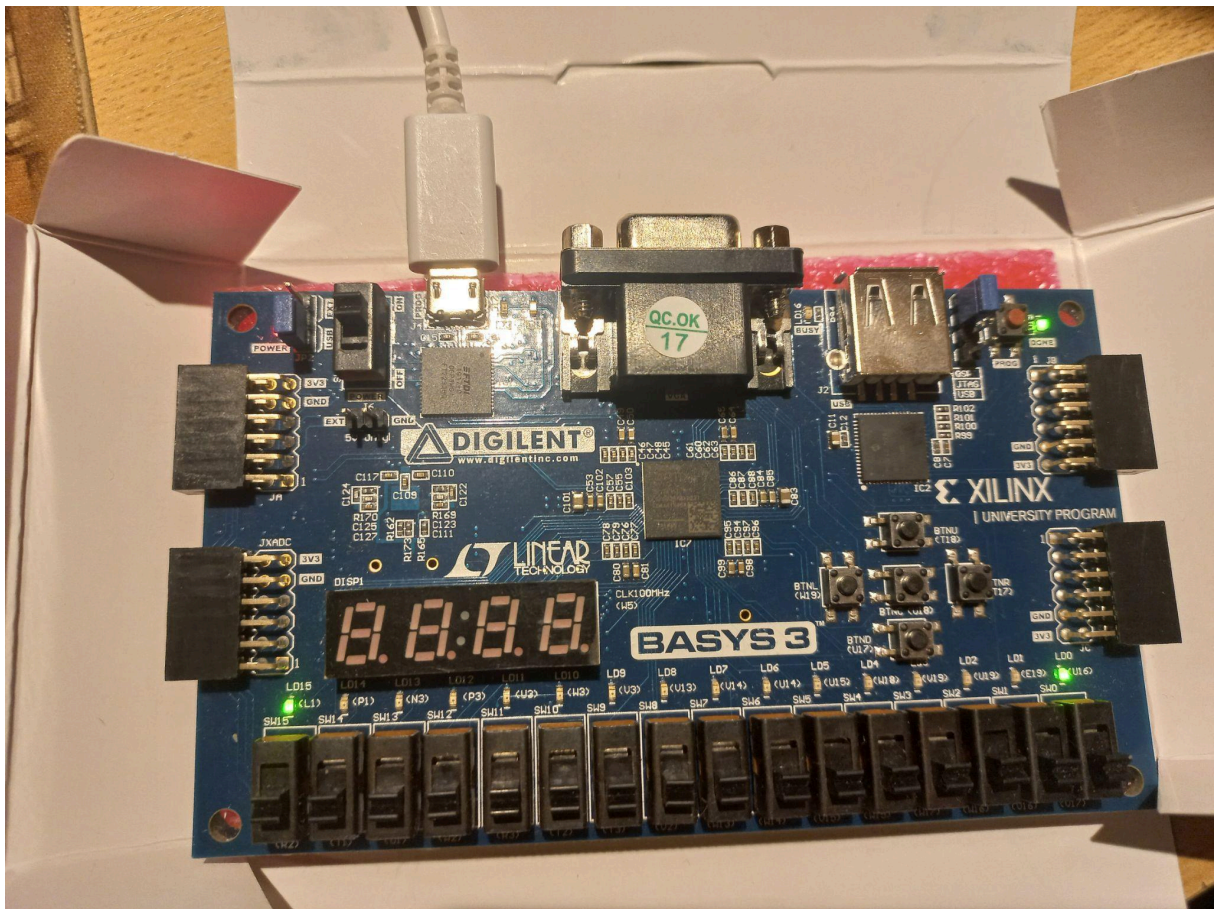


Las celdas fueron pintadas de celestes, en los momentos que se puede apreciar un cambio en el programa, con líneas rojas están marcados los cortocircuitos que deben suceder en el programa para

evitar riesgos, la gran nube verde representa una burbuja, y las celdas pintadas de morado y rosa, son aquellas que muestran cómo cambia el programa debido a los saltos que ocurren.

7 - Ejecución con la placa

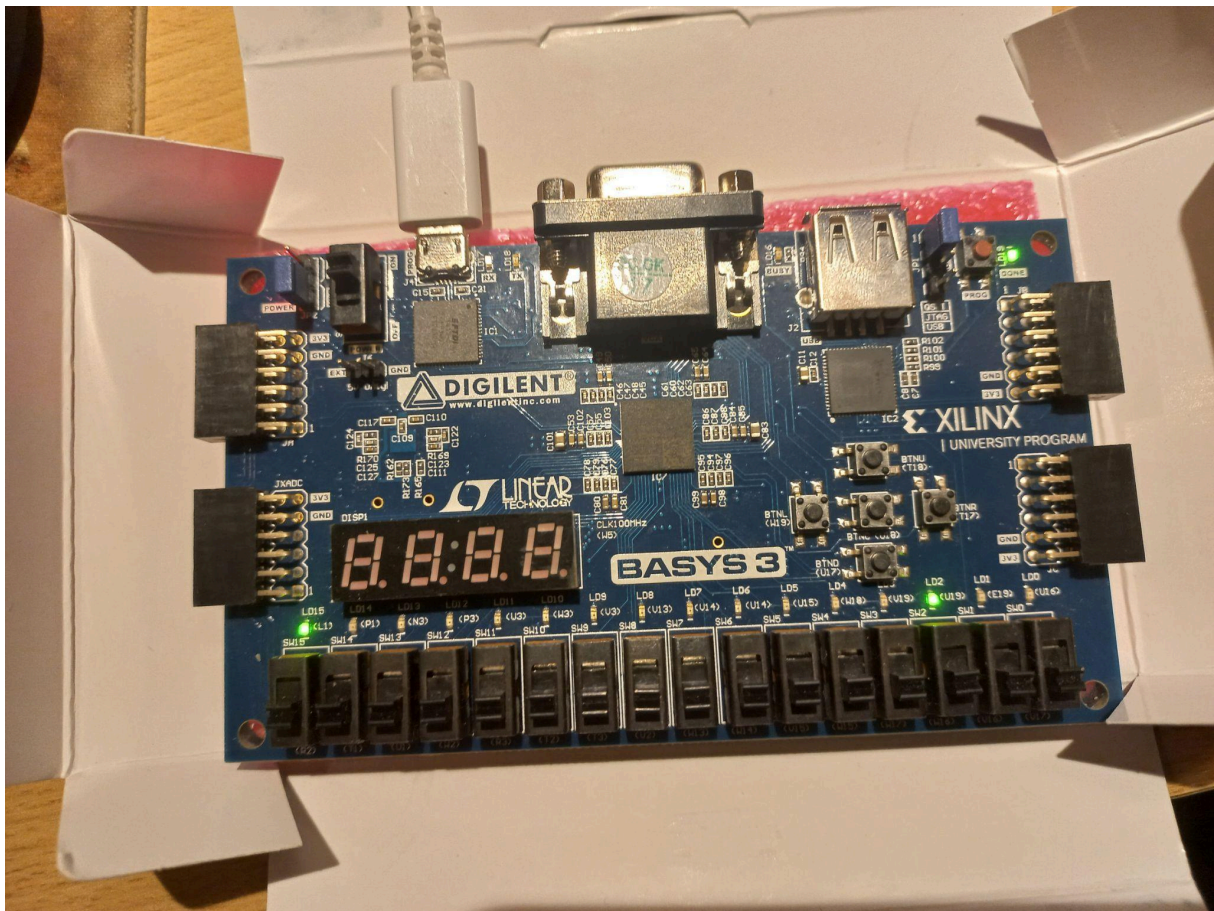
Una vez conectada la placa a la computadora, haber generado el bitstream, vinculado vivado a la placa, y cargado el programa, al observar la placa se la podrá ver en el siguiente estado:



En el archivo de constraints, hay algunos leds que están vinculados, y eso es lo que podemos ver:

- El primer LED a la izquierda, etiquetado como L1, corresponde a la variable `o_locked` del clock wizard.
- El último LED a la derecha, etiquetado como U16, está vinculado al primer estado, es decir la máquina de estados se encuentra en START, por lo que podemos comprobar que está esperando la señal por UART para avanzar.

Una vez apretado iniciar en la app, podremos ver que la placa se encuentra así:

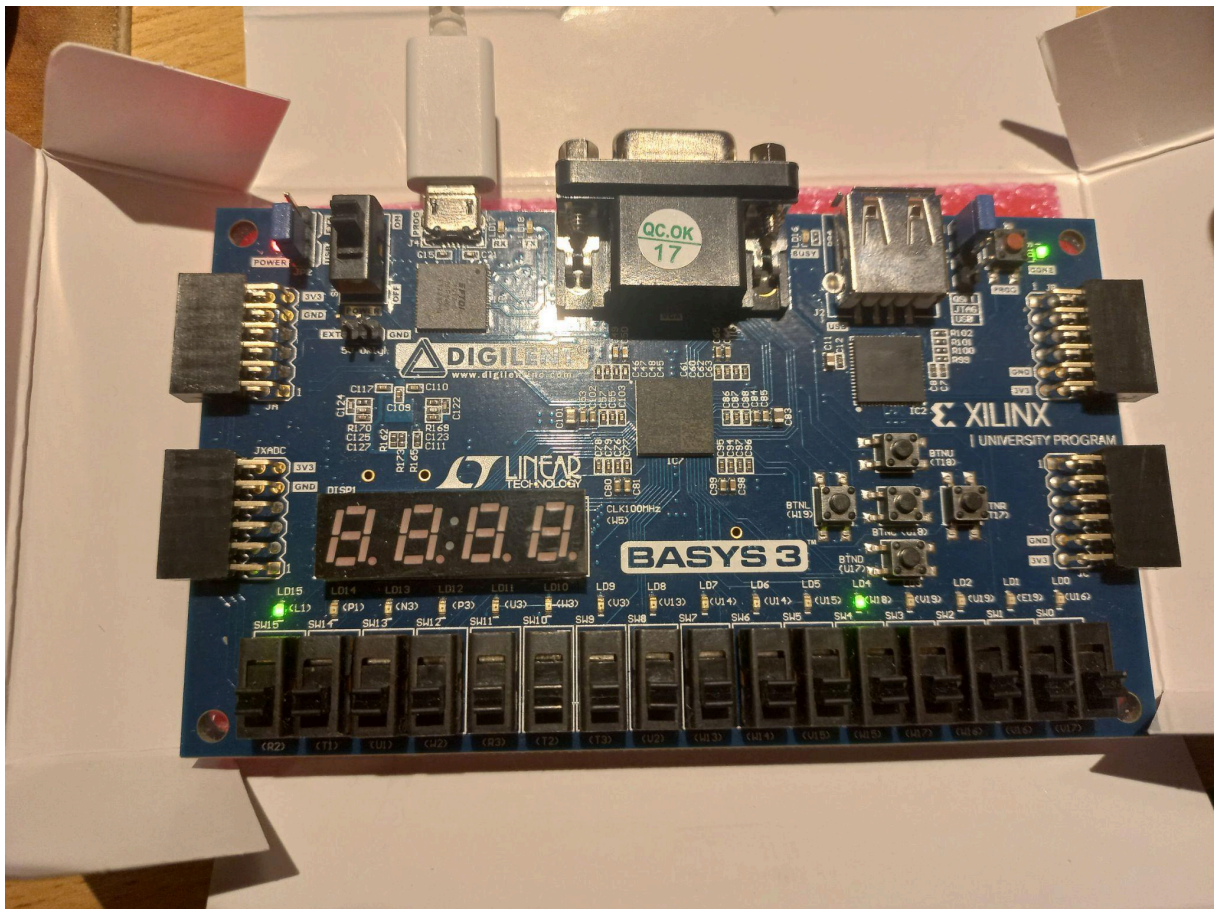


Donde:

- El LED de *o_locked* se encuentra nuevamente encendido.
- Se encuentra prendido el tercer LED desde la derecha, etiquetado como U19, el mismo representa que la máquina de estados se encuentra READY, el tercer estado.
- Esto no significa que se haya saltado el segundo estado, WRITE_IM, el mismo está vinculado al segundo LED, lo que sucede es que al ser automáticamente ejecutado por la aplicación, tan solo se puede apreciar cómo parpadea mientras se carga el programa, y finalmente se estabiliza en el tercer estado.

Todos los estados de la máquina de estados están vinculados a un LED, sin embargo, al igual que sucede con WRITE_IM, al avanzar automáticamente sin intervención del usuario, tan solo se puede ver cómo parpadean, hasta que finalmente quede fijo en un estado que se quede a la espera de intervención.

A continuación, un ejemplo de como se ve, una vez que se apreta el boton de Step

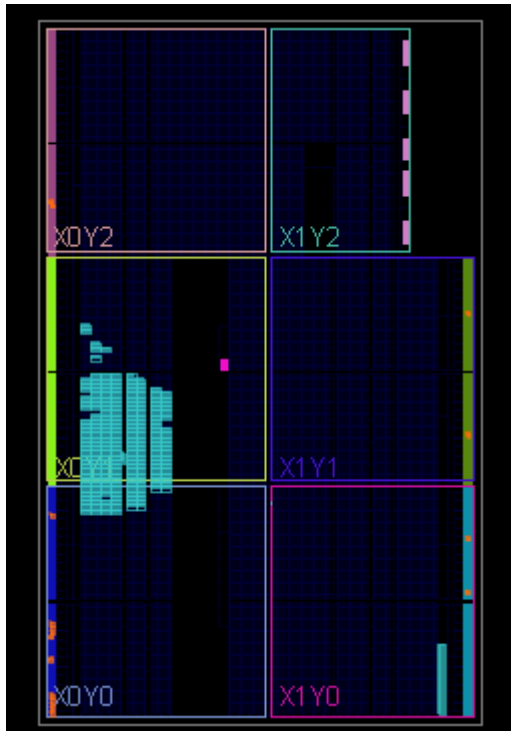


- El LED de *o_locked* se encuentra nuevamente encendido.
- Se encuentra prendido el quinto LED desde la derecha, etiquetado como W18, el mismo representa que la máquina de estados se encuentra STEP_BY_STEP, el quinto estado, entendiendo que el cuarto es CONTINUE, esperando a nuevamente recibir un comando.

Si en algún momento, se alcanza el final del programa, ya sea porque se usó la función continua, o se llegó de a pasos hasta al final, la máquina de estados vuelve al principio, y se puede visualizar como en la primera foto.

Como dato adicional, el botón a la izquierda de entre los 5 que forman un símbolo de suma, está vinculada a la entrada *i_reset* del sistema, que puede ser utilizada para cargar un nuevo programa utilizando el botón de volver en la aplicación sin tener que reconfigurar la placa (el reset no borra la memoria de datos ni registros).

8 - Implementation y análisis de la frecuencia de clock



Al seleccionar *implementation* en vivo, se nos ofrece mucha información sobre el hardware programado para ser utilizado en la placa, siendo la imagen de arriba la primera en abrirse, y luego en las pestañas de abajo, detalles de las mismas, por ejemplo la de energía:

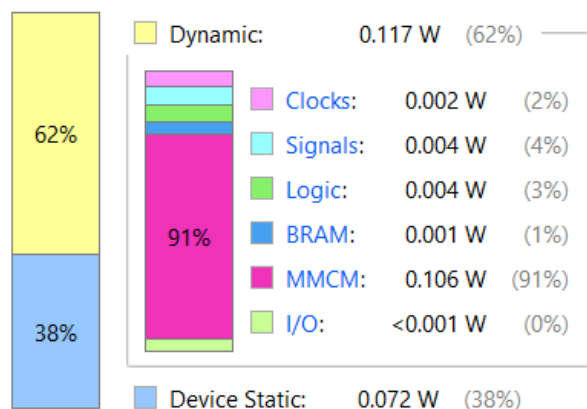
Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	0.189 W
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	25.9°C
Thermal Margin:	59.1°C (11.7 W)
Effective θ_{JA} :	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power



Otra de las pestañas, corresponde al del uso del clock:

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.216 ns	Worst Hold Slack (WHS): 0.060 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: 4001	Total Number of Endpoints: 4001	Total Number of Endpoints: 1087

All user specified timing constraints are met.

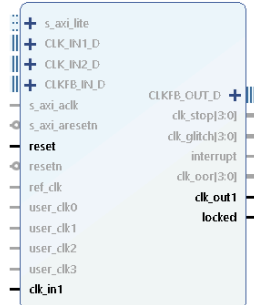
En el caso de nuestro trabajo, utilizamos 50Mhz para que vaya de la mano con el de la UART, el mismo fue configurado agregando un clockin wizard, como sysclock, y luego detallado su salida:

Clocking Wizard (6.0)

Documentation IP Location Switch to Defaults

IP Symbol Resource

☒ Show disabled ports



Component Name clk_wiz_0

Board Clocking Options Output Clocks Port Renaming MMCM Settings Summary





Associate IP interface with board interface

IP Interface	Board Interface
CLK_IN1	sys clock
CLK_IN2	Custom
EXT_RESET_IN	Custom

Clear Board Parameters

Board	Clocking Options	Output Clocks	Port Renaming	MMCM Settings	Summary
-------	------------------	---------------	---------------	---------------	---------


The phase is calculated relative to the active input clock.


Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)
		Requested	Actual	Requested	Actual	Requested
<input checked="" type="checkbox"/> clk_out1	clk_out1 	50.000 	50.00000	0.000 	0.000	50.000 

Volviendo al análisis del tiempo, en el resumen mostrado, es importante notar WNS (Worst Negative Slack) el cual aproximadamente significa el peor caso de retraso del sistema, por lo que si fuera negativo, implicaría que en el peor caso, el retraso que existe en el sistema es mayor que el ciclo del clock, con valores positivos como se ha mostrado, podríamos decir que es una buena frecuencia para trabajar.


Para encontrar el máximo valor posible, se procede a probar con distintas frecuencias de clock, a continuación algunos ejemplos:

100 MHz

 Critical Messages
✕

 There was one critical warning message while opening this design.

Messages

 [Timing 38-282] The design failed to meet the timing requirements. Please see the timing summary report for details on the timing violations.

Design Timing Summary

Setup	Hold
Worst Negative Slack (WNS): -0.201 ns	Worst Hold Slack (WHS): 0.110 ns
Total Negative Slack (TNS): -0.201 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 1	Number of Failing Endpoints: 0
Total Number of Endpoints: 4001	Total Number of Endpoints: 4001

Timing constraints are not met.

Como se puede ver, incluso muestra un mensaje de error, y la leyenda en negrita marcando el problema, con el WNS en negativo

93 MHz

Design Timing Summary

Setup	Hold
Worst Negative Slack (WNS): 0.211 ns	Worst Hold Slack (WHS): 0.052 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 4001	Total Number of Endpoints: 4001

All user specified timing constraints are met.

94MHz

Design Timing Summary

Setup

Worst Negative Slack (WNS): **-0.254 ns**

Total Negative Slack (TNS): **-0.254 ns**

Number of Failing Endpoints: **1**

Total Number of Endpoints: 4001

Hold

Worst Hold Slack (WHS): **0.109 ns**

Total Hold Slack (THS): **0.000 ns**

Number of Failing Endpoints: **0**

Total Number of Endpoints: 4001

Timing constraints are not met.

Con esto podemos ver que el máximo valor de frecuencia de clock que se puede utilizar en este trabajo es de 93 MHz.

Adicionalmente, podemos observar como se ven afectados los módulos:

Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup											
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	
Path 1	-0.254	11	132	PIPELINE/ME...g_reg[0]/C	PIPELINE/EX...zero_reg/D	10.791	1.823	8.968	10.6	clk_out1_clk_wiz_0	
Path 2	0.286	5	36	PIPELINE/IF...on_reg[18]/C	PIPELINE/DEC...halt_reg/R	4.450	1.079	3.371	5.3	clk_out1_clk_wiz_0	
Path 3	0.286	5	36	PIPELINE/IF...on_reg[18]/C	PIPELINE/DE...jump_reg/R	4.450	1.079	3.371	5.3	clk_out1_clk_wiz_0	
Path 4	0.286	5	36	PIPELINE/IF...on_reg[18]/C	PIPELINE/DEC...dest_reg/R	4.450	1.079	3.371	5.3	clk_out1_clk_wiz_0	
Path 5	0.286	5	36	PIPELINE/IF...on_reg[18]/C	PIPELINE/DE...write_reg/R	4.450	1.079	3.371	5.3	clk_out1_clk_wiz_0	
Path 6	0.286	5	36	PIPELINE/IF...on_reg[18]/C	PIPELINE/DE...gnded_reg/R	4.450	1.079	3.371	5.3	clk_out1_clk_wiz_0	
Path 7	0.310	0	262	debug_unit/i...unt_reg[3]/C	PIPELINE/F...RAMA/WADR3	4.783	0.459	4.324	5.3	clk_out1_clk_wiz_0	
Path 8	0.310	0	262	debug_unit/i...unt_reg[3]/C	PIPELINE/F...RAMB/WADR3	4.783	0.459	4.324	5.3	clk_out1_clk_wiz_0	
Path 9	0.310	0	262	debug_unit/i...unt_reg[3]/C	PIPELINE/F...RAMC/WADR3	4.783	0.459	4.324	5.3	clk_out1_clk_wiz_0	
Path 10	0.310	0	262	debug_unit/i...unt_reg[3]/C	PIPELINE/_AMD/WADR3	4.783	0.459	4.324	5.3	clk_out1_clk_wiz_0	

En este caso, siendo el latch entre el execute y memory quien ya no puede trabajar más a esta frecuencia.