Cátedra de Arquitectura de Computadoras

Trabajo Práctico Final

Procesador MIPS simplificado con pipeline

Ing. En Computación

Blanco Lucas
Machado Matias

2023

Índice

Enunciado

El objetivo de este trabajo es implementar en FPGA, programando en Verilog, un procesador MIPS simplificado con pipeline.

Requerimientos

Los requerimientos del trabajo son los siguientes:
Se debe implementar el procesador MIPS segmentado en las siguientes etapas:

- **Instruction Fetch:** busca la instrucción en la memoria de programa.
- **Instruction Decode:** decodifica la instrucción y lee los registros.
- **Execute:** ejecuta la instrucción.
- **Memory Access:** lee o escribe la memoria de datos.
- **Write back:** escribe los resultados en los registros.

El procesador debe tener soporte para los riesgos estructurales, de datos y de control mediante:

- **Unidad de cortocircuitos;**
- **Unidad de detección de riesgos.**

Además:

- El programa a ejecutar debe ser cargado en la memoria de programa mediante un archivo ensamblado.

- El archivo de ensamblado debe convertirse a código de instrucción y transmitirse a través de una interfaz UART antes de comenzar su ejecución.

- Se debe simular una unidad de debug que envíe información hacia y desde el proce-sador mediante UART. La información es la siguientes:

  1. Contenido de los 32 registros;
  2. Contador de programa (PC);
  3. Contenido de la memoria de datos usada;
  4. Cantidad de ciclos de reloj desde el inicio.

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

  1. 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;

  2. 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 hasta terminar el programa.

Etapas del MIPS

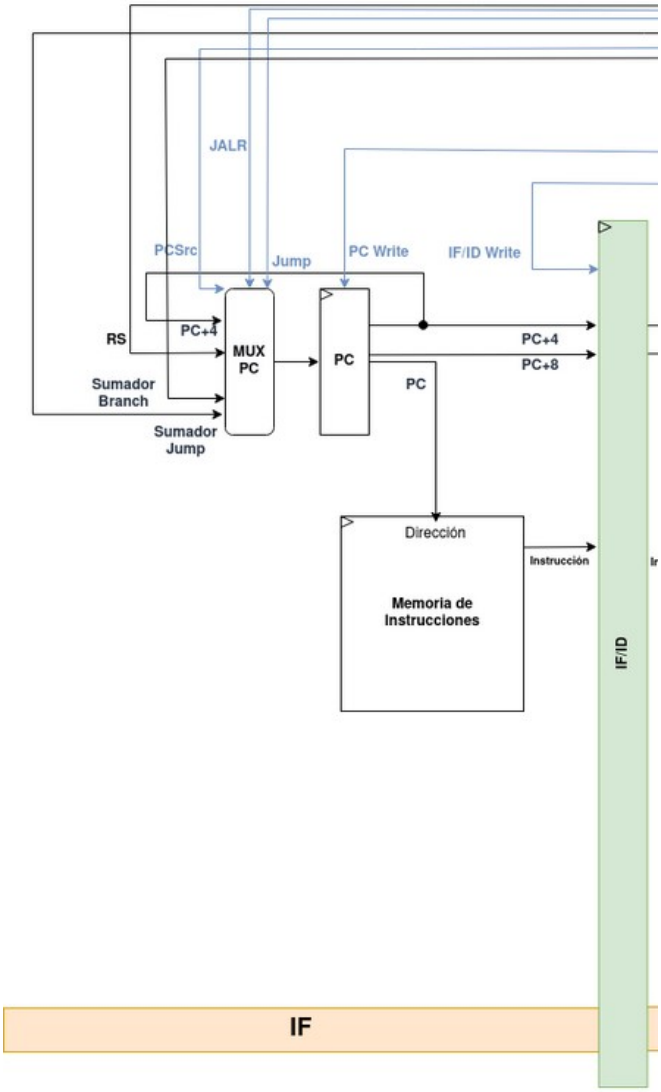
El procesador MIPS resuelve x cantidad de instrucciones a partir de un pipeline de 5 etapas. Las etapas son las siguientes:

IF(Instruction Fetch)

Durante esta etapa se lee la instrucción ha ejecutar de la memoria de instrucciones y se actualiza el **Program Counter (PC)**.

Para actualizar el **PC** se puede hacer:

- Sumando 4 para avanzar a la siguiente instrucción en memoria.
- Saltar a una dirección en memoria particular en base a las instrucciones que se ejecutaron anteriormente.



Modulos

- **MuxPC:** este módulo se encarga de enviarle al PC la siguiente dirección de instrucción eligiendo entre:
  - **PC+4**
  - **Instrucción resultado de la ALU de Branch**
  - **Instrucción resultado de Jump**
  - **RS**

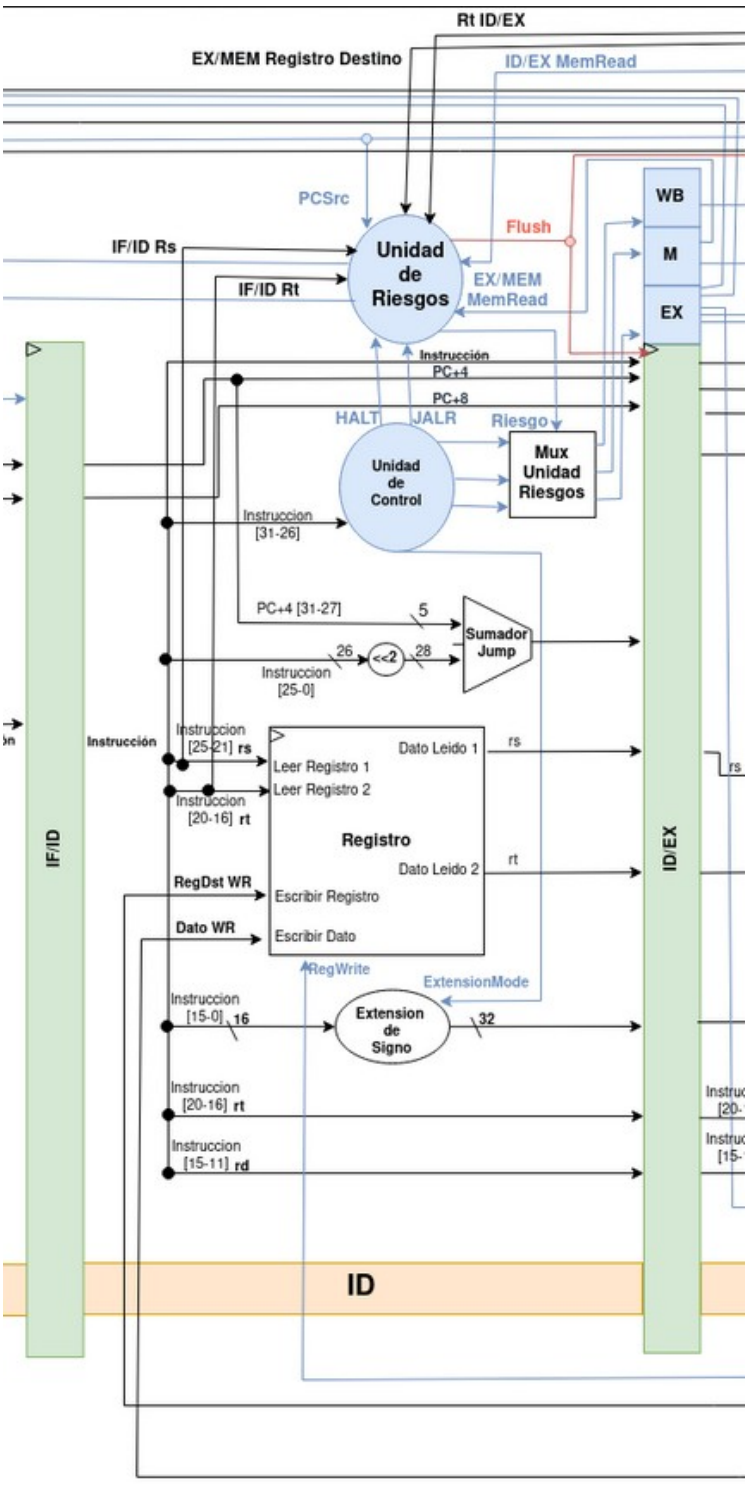
- **PC:** este módulo se encarga de entregar la nueva dirección de instrucción, la instrucción+4 y la instrucción+8.

- **Memoria de Instrucciones:** recibe la dirección de memoria del program counter y devuelve la siguiente instrucción.

- **IF/ID:** este módulo se encarga de recibir y entregar los datos generados en la etapa de **fetch** a la etapa de **decode** cuando hay un pulso positivo de clock.

ID (Instruction Decode)

Durante esta etapa se decodifica la instrucción a resolver, se buscan los registros a utilizar y se calcula la extensión de signo.



Modulos

- **Sumador Jump:** este módulo se encarga de sumar la dirección de Jump con el PC+4. Da como resultado la dirección de Jump.

- **Registro:** este módulo se encarga de:
  - Recibir la dirección **rs, rt** y leer los registros correspondientes para presentar los datos correspondientes a la salida.
  - Recibir la dirección **rd** del registro donde se va a guardar el **dato a escribir**.

- **Extensor de Signo:** este módulo se encarga de extender el registro de 16 bits a 32 bits. Lo puede hacer de tres formas:
  - Extender a 32 con los 16 más significativos respetando el signo.
  - Extender a 32 con los 16 más significativos usando un cero.
  - Colocar los 16 bits como más significativos y agregar ceros en los 16 menos significativos.

- **Unidad de Control:** este módulo recibe los 6 bits más altos de la instrucción y setea los señal de control necesarios en base a la instrucción decodificada.

- **Unidad de Riesgos:** este módulo se encarga de agregar burbujas cuando hay un riesgo como una lectura después de una escritura para una instrucción de Load o para el caso de Branch.

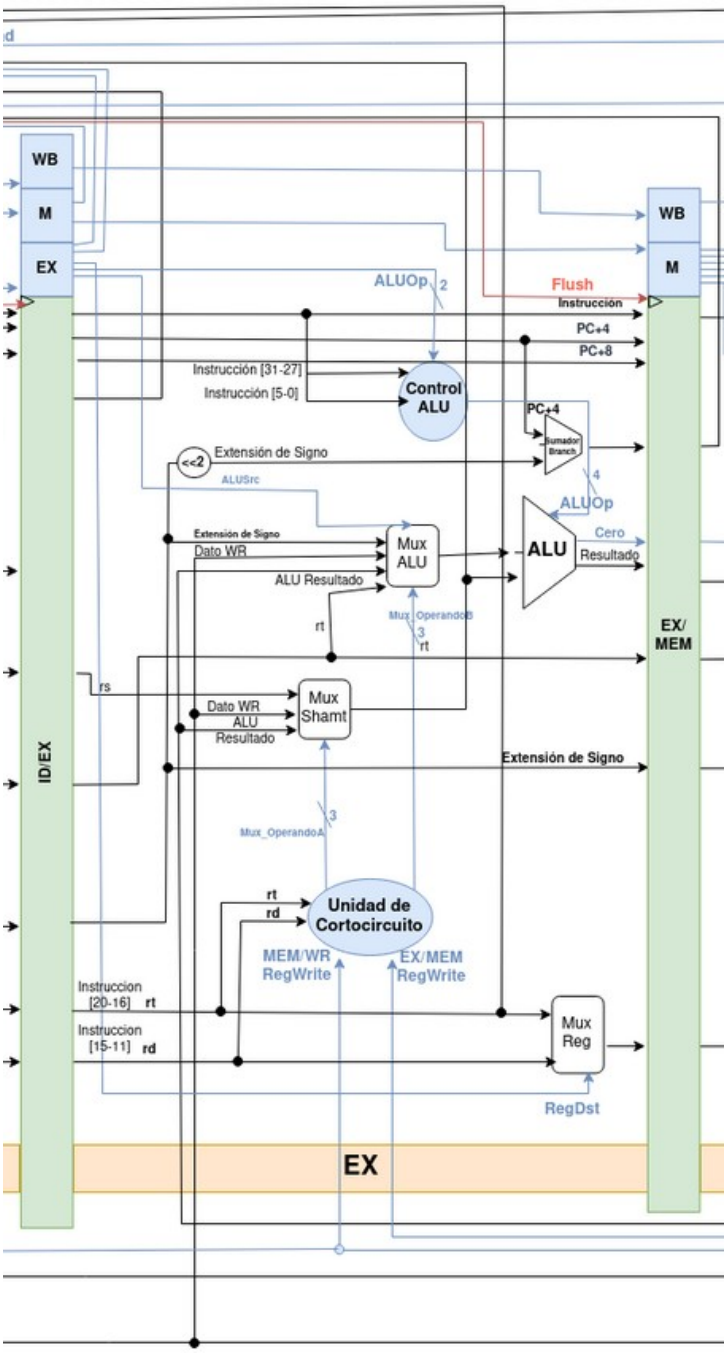
- **Mux Unidad de Riesgos:** este módulo se encarga de limpiar las señales de control cuando hay un riesgo.

- **ID/EX:** este módulo se encarga de recibir y entregar los datos generados en la etapa de decode a la etapa de execute cuando hay un pulso positivo de clock. Además envía las señales de control generadas por la Unidad de Control.

EX(Execute)

Durante esta etapa:

- Se executa la instrucción en la ALU
- Se calcula la dirección de instrucción de Branch
- Se calcula la dirección de instrucción de Jump
- Se selecciona el registro destino



Modulos

- **Sumador Branch:** este módulo se encarga de sumar la dirección de signo extendido (multiplicado x 4) con el PC+4. Da como resultado la dirección de Branch.

- **Mux ALU Shamt:** este módulo se encarga de enviar el valor del **operando A** a la ALU dependiendo de la señal de control de la Unidad de Cortocircuito.

- **Mux ALU:** este módulo se encarga de darle el **operando B** a la ALU dependiendo de:
  - Si la instrucción debe usar el inmediato o no.
  - Si se debe usar el dato del EX/MEM o MEM/WR según la unidad de Cortocircuito.

- **ALU:** este módulo:
  - Resuelve una de 10 instrucciones con dos registros.
  - Resuelve la instrucción Shamt recibiendo la cantidad de lugares a mover.
  - Devuelve el resultado de la operación.
  - Devuelve un flag con cero si el resultado es este.

Op	Código
AND	0000
OR	0001
ADD	0010
SLL	0011
SRL	0100
SRA	0101
SUB	0110
SLT	0111
NOR	1100
XOR	1101

- **Unidad de Control ALU:** este módulo sirve para interpretar la instrucción y enviarle una señal a la ALU sobre que operación debe resolver.
  - Recibe una señal de control a partir de la cuál sabe de que tipo va a ser la instrucción.
    - **(00):** Si la instrucción accede a **memoria** se debe hacer una **suma**.
    - **(01):** Si la instrucción es un **salto condicional** se debe hacer una **resta**.
    - **(10):** Si la instrucción es de **tipo R** se va a evaluar el código de instrucción [5:0] y se va a seleccionar la operación específica.
    - **(11):** Si la instrucción utiliza un **inmediato** se va a evaluar el código de instrucción [31:27].

- **Unidad de Cortocircuito:** este módulo verifica si hay un riesgo de **lectura antes de escritura** por lo que valida que valor de registro debe utilizarse comparando con los valores resultado de EX/MEM y MEM/WB y sus registro a guardar. Genera una señal de salida de dato a utilizar para los operandos de la ALU.

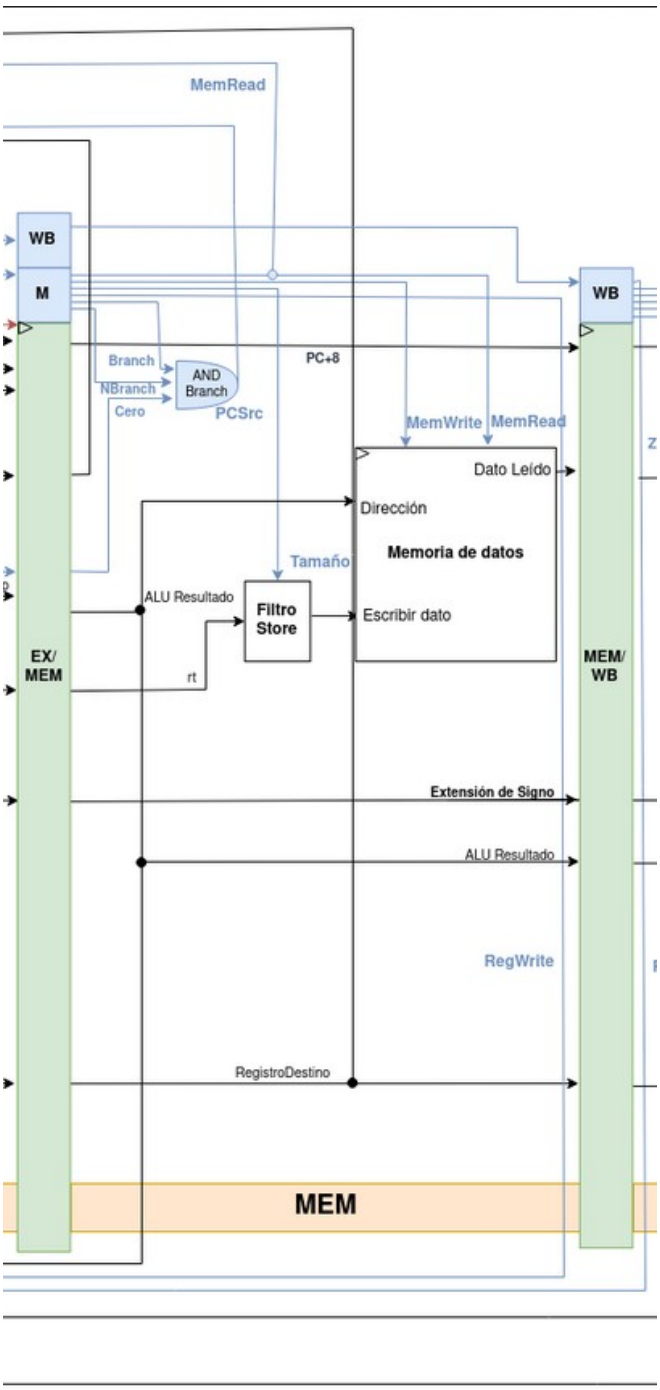
- **Mux Registro:** este módulo elige el registro destino entre rd y rt en base a una señal de control.

- **EX/MEM:** este módulo se encarga de recibir y entregar los datos generados en la etapa de **execute** a la etapa de **memory access** cuando hay un pulso positivo de clock. Además envía las señales de control generadas por la Unidad de Control.

MEM(Memory Access)

Durante esta etapa:

- Se administra el flag de salto de Branch hacia el PC.
- Se aplica el filtro de datos para las instrucciones de Store.
- Se guardan los datos en memoria de datos.



Modulos

- **AND Branch:** este módulo se encarga de validar si el resultado de la ALU fue cero y si la instrucción a ejecutar es Branch On Equal o Branch On Not Equal.

- **Filtro Store:** este módulo se encarga de implementar un filtro de valores a guardar en la Memoria de Datos. En base a señales de control guarda el dato entero, el primer byte o la mitad del dato.

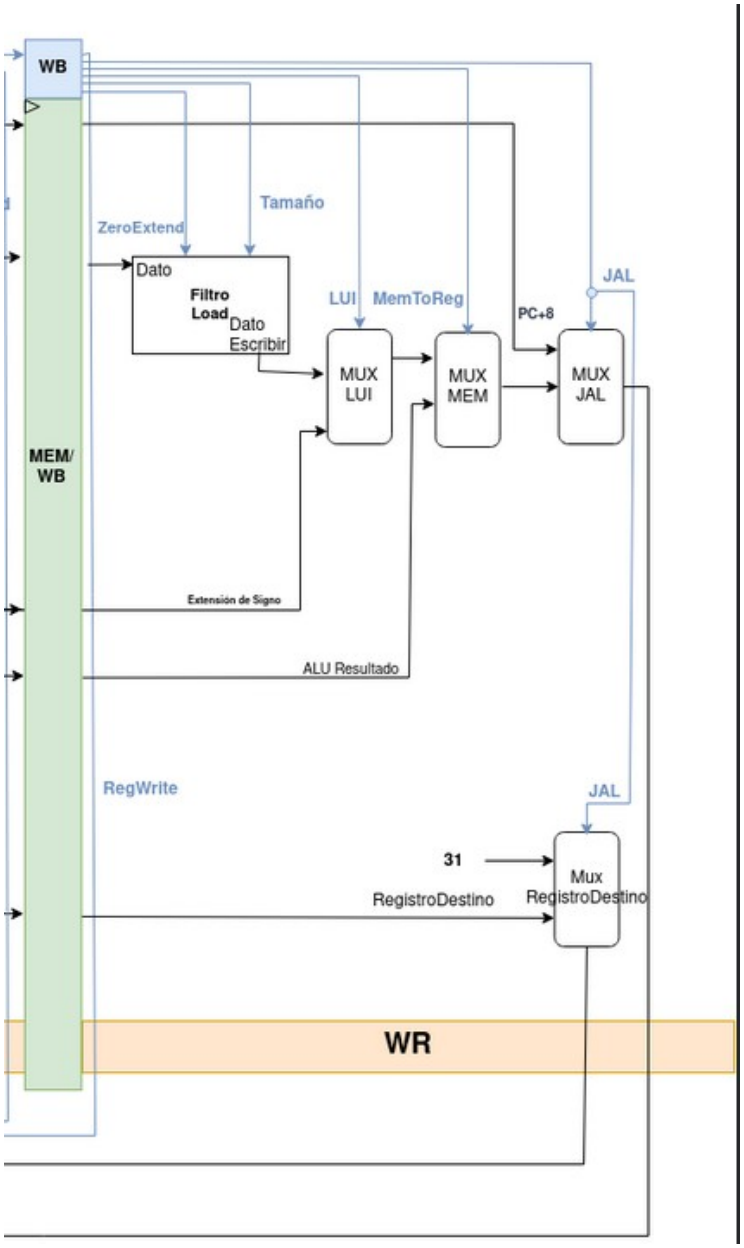
- **Memoria de Datos:** este módulo se encarga de implementar una memoria de datos que lee un dato en memoria y guarda un dato en memoria en base a el pulso del clock y señales de control.

- **MEM/WR:** este módulo se encarga de recibir y entregar los datos generados en la etapa de memory access a la etapa de write back cuando hay un pulso positivo de clock. Además envía las señales de control generadas por la Unidad de Control.

WR(Write Back)

Durante esta etapa:

- Se aplica el filtro de datos para las instrucciones de Load.
- Se aplica el multiplexor de instrucción LUI.
- Se aplica el multiplexor de instrucción JAL.
- Se aplica el multiplexor de registro destino.



Modulos

- **Filtro Load:** este módulo se encarga de filtrar cuantos bits y de que forma (signed/unsigned) pasan hacia los registros.

- **Mux LUI:** este módulo se encarga de evaluar la señal de control de la **instrucción LUI** y enviar el dato correspondiente.

- **Mux Memoria:** este módulo se encarga de evaluar la señal de control y enviar el dato correspondiente entre la ALU y Memoria.

- **Mux Registro Destino:** este módulo se encarga de evaluar la señal de control de la **instrucción JAL** y elegir entre Rd y el **registro 31.**

- **Mux Escribir Dato:** este módulo se encarga de evaluar la señal de control de la **instrucción JAL** y elegir si el dato que se guarda en el registro es el PC+8 o no.

Riesgos

Los riesgos que se tuvieron en cuenta y se resolvieron tener en la implementación del MIPS son:

- **Estructurales:** el riesgo existe cuando distintas instrucciones desean utilizar un mismo recurso. Para resolverlo se decidió crear distintos módulos:

  - **Memoria de Instrucciones:** en un ciclo la instrucción que se encuentra en la etapa de IF utiliza este recurso.
  - **Memoria de Datos:** en un ciclo la instrucción que se encuentra en la etapa de MEM utiliza este recurso.
  - **ALU:** unicamente la instrucción en la etapa de EX va a utilizar este recurso.
  - **Sumador Branch:** cuando hay una instrucción de Branch en EX se puede calcular la dirección a saltar en este sumador y en la ALU la condición.
  - **Sumador Jump:** sirve para que se pueda calcular la dirección de salto de Jump en la etapa de ID.

- **De datos:** el riesgo existe cuando distintas instrucciones utilizan los mismos recursos y la modificación de uno genera un riesgo en futuras instrucciones. Los riesgos son de dos tipos:
  - **Lectura despues de escritura:** se da cuando una instruccion anterior escribe en un registro que la siguiente instrucción lee. Este riesgo se resuelve con la **Unidad de Cortocircuito.**

Por ejemplo, en este caso el registro r1 va a estar listo al final de la ejecución del pipeline (WR) por lo que tenemos un riesgo.

    - add r1,r2,r3
    - sub r4,r0,r1

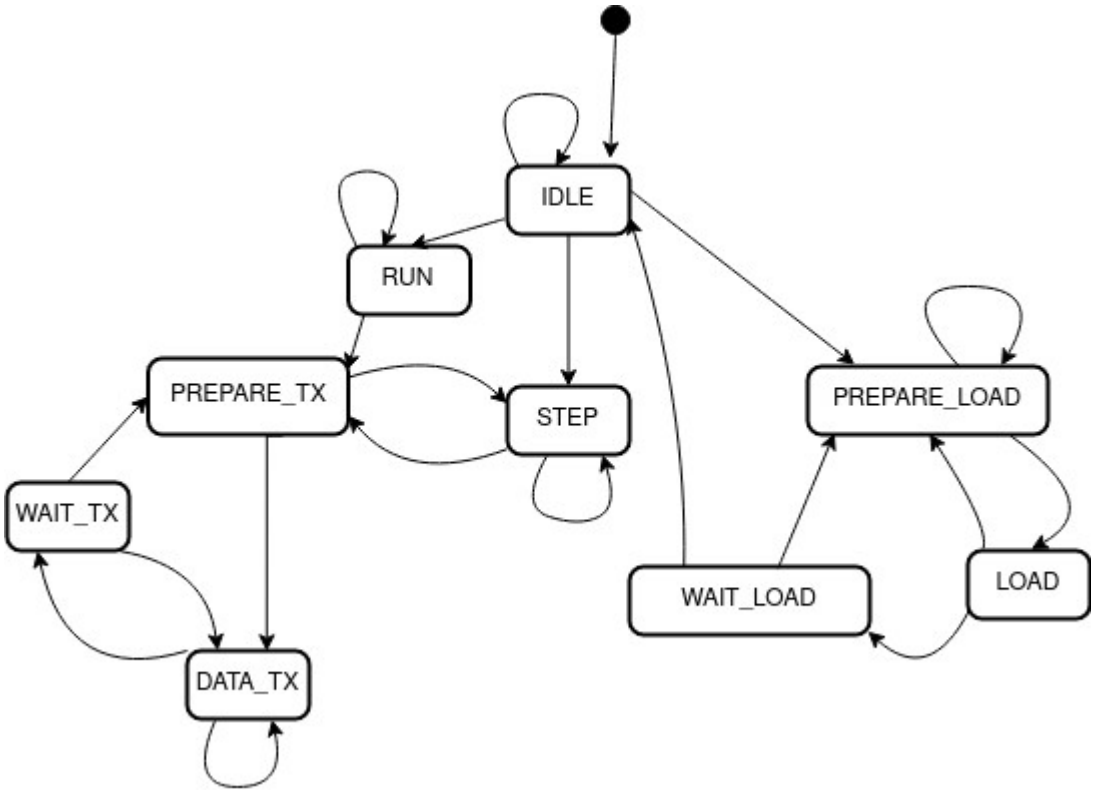
  - Otro problema que surje con la lectura después de escritura es si se carga un valor de memoria en un registro. En ese caso se debe esperar un ciclo hasta que el valor sea obtenido en WR. Para resolver este riesgo se agrega la **Unidad de Riesgos** que agrega una burbuja hasta que este listo el resultado.

  Para el caso de los siguientes riesgos se solucionan separando los momentos en los que se leen y se escriben los registros:
  - **Escritura después de escritura.**
  - **Escritura después de lectura.**

- **De control:** el riesgo existe cuando existe un salto condicional (Branch). En este caso vamos a tener el resultado en la etapa de MEM cuando se sepa la condición.
  Para resolver este inconveniente se decidió ejecutar todas las instrucciones siguientes a la Branch hasta la etapa de MEM. Luego se agregó lógica a la **Unidad de Riesgos** para que al momento de saber si se debe tomar el salto se limpien las etapas borrando las instrucciones que se ejecutaron que no debian (**Flush**).

Unidad de Debug

La unidad de debug es el punto de comunicacion entre el MIPS y el usuario. Esta unidad permite cargar el codigo del programa a la placa, iniciar el programa y obtener informacion de PC, cantidad de ciclos de reloj y contenido de las memorias de datos y registros.
La unidad de debug se implementa por sobre el MIPS como otro modulo y mantiene la comunicacion via UART entre la placa y la PC.
El modulo de la unidad de debug corresponde a una maquina de estados que sigue el siguiente diseño:



- **IDLE:** Este es el estado inicial y final de la maquina de estados, en este estado la unidad de debug queda a la espera de un comando via UART. Este comando puede ser:
  - **R:** Comando ‘run’ inicia el clock del procesador en modo continuo, es decir, que el programa correra de inicio a fin.
  - **S:** Comando ‘step’ inicia el clock del procesador en modo paso a paso, es decir, que el programa correra un ciclo de clock y esperara por el comando ‘next’ para continuar.
  - **L:** Comando ‘load’, inicia el modo de carga de programa. Se recibiran las instrucciones a grabar en la memoria de instrucciones en bloques de 8 bits, hasta recibir una instruccion de HALT.

- **RUN:** Este es el modo continuo del procesador, el clock comenzara a correr y al finalizar la ejecucion del mismo (hasta que llegue una instruccion HALT), pasara al estado de envio de datos (PREPARE_TX), donde se enviara la informacion via UART hacia la PC.

- **STEP:** Este es el modo paso a paso del procesador, el clock avanzara en un paso y enviara via UART la informacion hacia la PC, luego quedara en espera de un comando ‘n’ para realizar nuevamente el mismo proceso, hasta que la instruccion que se ejecuta sea un HALT, en cuyo caso volvera a IDLE.

- **PREPARE_TX:** En este estado se prepara la linea (4 palabras, 32 bits) a enviar via UART. Cuando se envien todas las palabras necesarias (32 registros, 16 lineas de memoria , 1 linea para el PC y 1 linea para la cuenta de los ciclos de reloj.

- **DATA_TX:** Aqui se enviara cada chunk de 8bits de la linea a enviar.

- **WAIT_TX:** Se espera a que se termine de enviar la palabra solicitada y se valida si se termino de enviar la linea, en cuyo caso volvera a PREPARE_TX para comenzar con la siguiente linea o se volvera a DATA_TX para enviar la siguiente palabra de la linea actual.

- **PREPARE_LOAD:** En este estado se recibe una linea completa via UART (repartida en 4 chunks de 8bits).

- **LOAD:** Aqui se carga la instruccion recibida en la memoria de registros, guardando las instrucciones en orden de llegada.

- **WAIT_LOAD:** Se valida si la instruccion cargada es un HALT en cuyo caso se termina la etapa de carga de programa volviendo a IDLE, sino se vuelve a PREPARE_LOAD para esperar la siguiente linea a escribir.

Para la comunicacion desde la computadora se provee un programa en Python que funciona como interfaz con la placa. Este programa tiene dos funciones principales.

- Convertir un programa ASM (desde un archivo) a binario para ser enviado via UART a la placa.
- Enviar comandos y recibir informacion desde la placa y mostrarla al usuario.

El programa cuenta con el modulo principal llamado **mips-cli.py** que al ejecutarlo, convertira el programa **.asm a binario** y comenzara a cargarlo a la placa via UART.

El programa recibe ciertos argumentos para configurar el proceso:

- **-f:** determina el archivo .asm a subir a la placa
- **-p:** especifica el puerto COM a utilizar para comunicarse con la placa.
- **-m:** especifica la cantidad de entradas de memoria a mostrar.
- **-r:** especifica la cantidad de entradas de registros a mostrar.

```
❯ python3 mips-cli.py -f program.asm -p /dev/ttyUSB1 -m 10 -r 32
MIPS PROCESSORS DEBUG UNIT

Program file: program.asm - SerialPort: /dev/ttyUSB1 - SerialConfig: 9600-8-N-1
Loading...
```

Luego espera por comandos introducidos por el usuario para ser enviados a la placa.

```
MIPS PROCESSORS DEBUG UNIT

Program file: program.asm - SerialPort: /dev/ttyUSB1 - SerialConfig: 9600-8-N-1
Loading...
Enter a character to send over serial port:
```

Desde este punto puede enviarse el comando ‘r’ o ‘s’ segun el modo de procesamiento requerido.

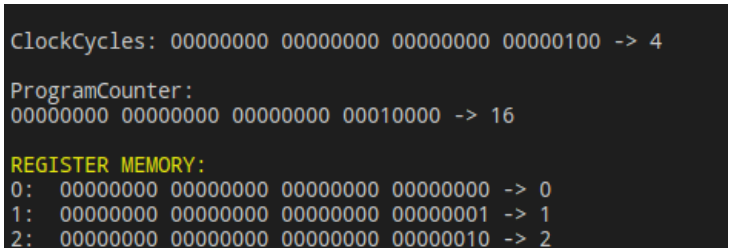
Al iniciar el modo continuo, el programa correra en la placa y devolvera el estado final de la memoria y los registros.

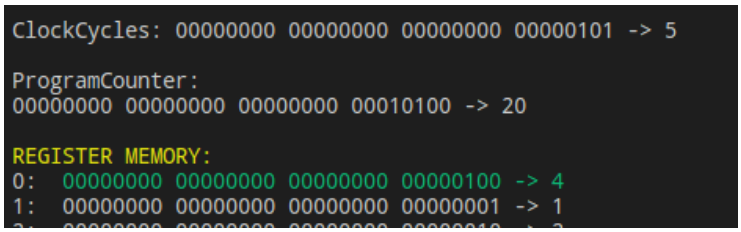
```
Program file: program.asm - SerialPort: /dev/ttyUSB1 - SerialConfig: 9600-8-N-1
Loading...
Enter a character to send over serial port: r
MIPS Processor Continuous Mode

ClockCycles: 00000000 00000000 00000000 00100111 -> 39

ProgramCounter: 00000000 00000000 00000000 10011100 -> 156

REGISTER MEMORY:
0:  00000000 00000000 00000000 00000000 -> 0
1:  00000000 00000000 00000000 00000001 -> 1
2:  00000000 00000000 00000000 00000010 -> 2
3:  00000000 00000000 00000000 00000001 -> 1
4:  00000000 00000000 00000000 00000100 -> 4
5:  00000000 00000000 00000000 00000101 -> 5
6:  00000000 00000000 00000000 00000110 -> 6
7:  00000000 00000000 00000000 00000111 -> 7
8:  00000000 00000000 00000000 00001000 -> 8
9:  00000000 00000000 00000000 00001001 -> 9
10:  00000000 00000000 00000000 00001010 -> 10
11:  00000000 00000000 00000000 00001011 -> 11
12:  00000000 00000000 00000000 00001100 -> 12
13:  00000000 00000000 00000000 00001101 -> 13
14:  00000000 00000000 00000000 00001110 -> 14
15:  00000000 00000000 00000000 00001111 -> 15
16:  00000000 00000000 00000000 00010000 -> 16
17:  00000000 00000000 00000000 00010001 -> 17
18:  00000000 00000000 00000000 00010010 -> 18
19:  00000000 00000000 00000000 00010011 -> 19
20:  00000000 00000000 00000000 00010100 -> 20
21:  00000000 00000000 00000000 00010101 -> 21
22:  00000000 00000000 00000000 00010110 -> 22
23:  00000000 00000000 00000000 00010111 -> 23
24:  00000000 00000000 00000000 00011000 -> 24
25:  00000000 00000000 00000000 00011001 -> 25
26:  00000000 00000000 00000000 00011010 -> 26
27:  00000000 00000000 00000000 00011011 -> 27
28:  00000000 00000000 00000000 00011100 -> 28
29:  00000000 00000000 00000000 00011111 -> 31
30:  00000000 00000000 00000000 00011110 -> 30
31:  00000000 00000000 00000000 00011101 -> 29

DATA MEMORY:
0:  00000000 00000000 00000000 00000000 -> 0
1:  00000000 00000000 00000000 00000001 -> 1
2:  00000000 00000000 00000000 00000010 -> 2
3:  00000000 00000000 00000000 00000011 -> 3
4:  00000000 00000000 00000000 00000011 -> 3
5:  00000000 00000000 00000000 00000101 -> 5
6:  00000000 00000000 00000000 00000110 -> 6
7:  00000000 00000000 00000000 00000010 -> 2
8:  00000000 00000000 00000000 00001000 -> 8
9:  00000000 00000000 00000000 00001001 -> 9
```

Al iniciar el modo paso a paso, el programa correra un solo ciclo de reloj y devolvera el estado actual de la memoria y registros para ese ciclo:

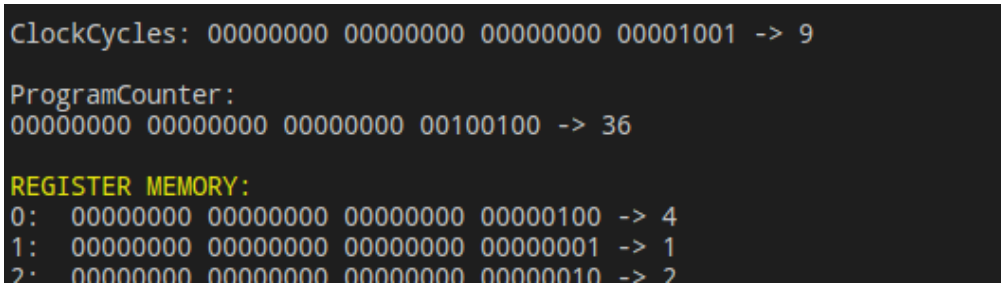
```
Enter a character to send over serial port: s
MIPS Processor Step Mode
Step Mode - press n for a new step n
-----------------------------------------------------

ClockCycles: 00000000 00000000 00000000 00000001 -> 1

ProgramCounter:
00000000 00000000 00000000 00000100 -> 4

REGISTER MEMORY:
0:  00000000 00000000 00000000 00000000 -> 0
1:  00000000 00000000 00000000 00000001 -> 1
2:  00000000 00000000 00000000 00000010 -> 2
3:  00000000 00000000 00000000 00000011 -> 3
4:  00000000 00000000 00000000 00000100 -> 4
5:  00000000 00000000 00000000 00000101 -> 5
6:  00000000 00000000 00000000 00000110 -> 6
7:  00000000 00000000 00000000 00000111 -> 7
8:  00000000 00000000 00000000 00001000 -> 8
9:  00000000 00000000 00000000 00001001 -> 9
10:  00000000 00000000 00000000 00001010 -> 10
11:  00000000 00000000 00000000 00001011 -> 11
12:  00000000 00000000 00000000 00001100 -> 12
13:  00000000 00000000 00000000 00001101 -> 13
14:  00000000 00000000 00000000 00001110 -> 14
15:  00000000 00000000 00000000 00001111 -> 15
16:  00000000 00000000 00000000 00010000 -> 16
17:  00000000 00000000 00000000 00010001 -> 17
18:  00000000 00000000 00000000 00010010 -> 18
19:  00000000 00000000 00000000 00010011 -> 19
20:  00000000 00000000 00000000 00010100 -> 20
21:  00000000 00000000 00000000 00010101 -> 21
22:  00000000 00000000 00000000 00010110 -> 22
23:  00000000 00000000 00000000 00010111 -> 23
24:  00000000 00000000 00000000 00011000 -> 24
25:  00000000 00000000 00000000 00011001 -> 25
26:  00000000 00000000 00000000 00011010 -> 26
27:  00000000 00000000 00000000 00011011 -> 27
28:  00000000 00000000 00000000 00011100 -> 28
29:  00000000 00000000 00000000 00011101 -> 29
30:  00000000 00000000 00000000 00011110 -> 30
31:  00000000 00000000 00000000 00011111 -> 31

DATA MEMORY:
0:  00000000 00000000 00000000 00000000 -> 0
1:  00000000 00000000 00000000 00000001 -> 1
2:  00000000 00000000 00000000 00000010 -> 2
3:  00000000 00000000 00000000 00000011 -> 3
4:  00000000 00000000 00000000 00000100 -> 4
5:  00000000 00000000 00000000 00000101 -> 5
6:  00000000 00000000 00000000 00000110 -> 6
7:  00000000 00000000 00000000 00000111 -> 7
8:  00000000 00000000 00000000 00001000 -> 8
9:  00000000 00000000 00000000 00001001 -> 9
Step Mode - press n for a new step
```

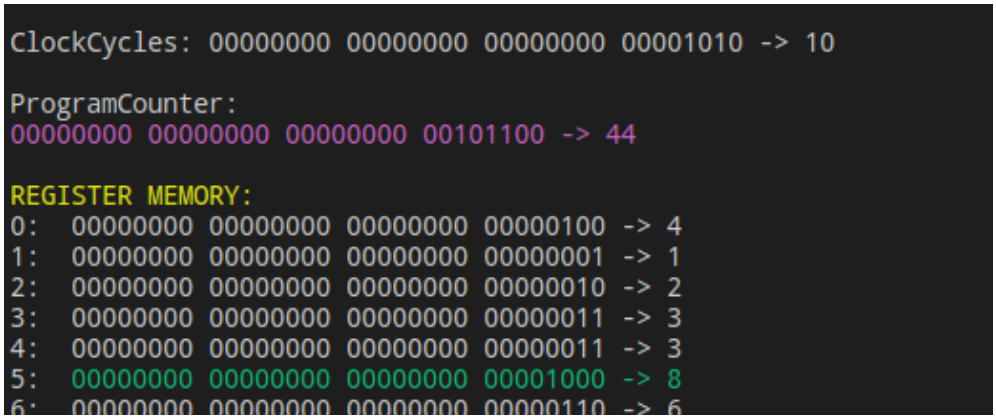
Si en el paso ejecutado se modifica algun registro o memoria estas apareceran en verde en el output del programa.





Si hubiese un salto en el PC se mostrara en color magenta y en el caso de una burbuja (mantener el PC) se mostrara en color azul:





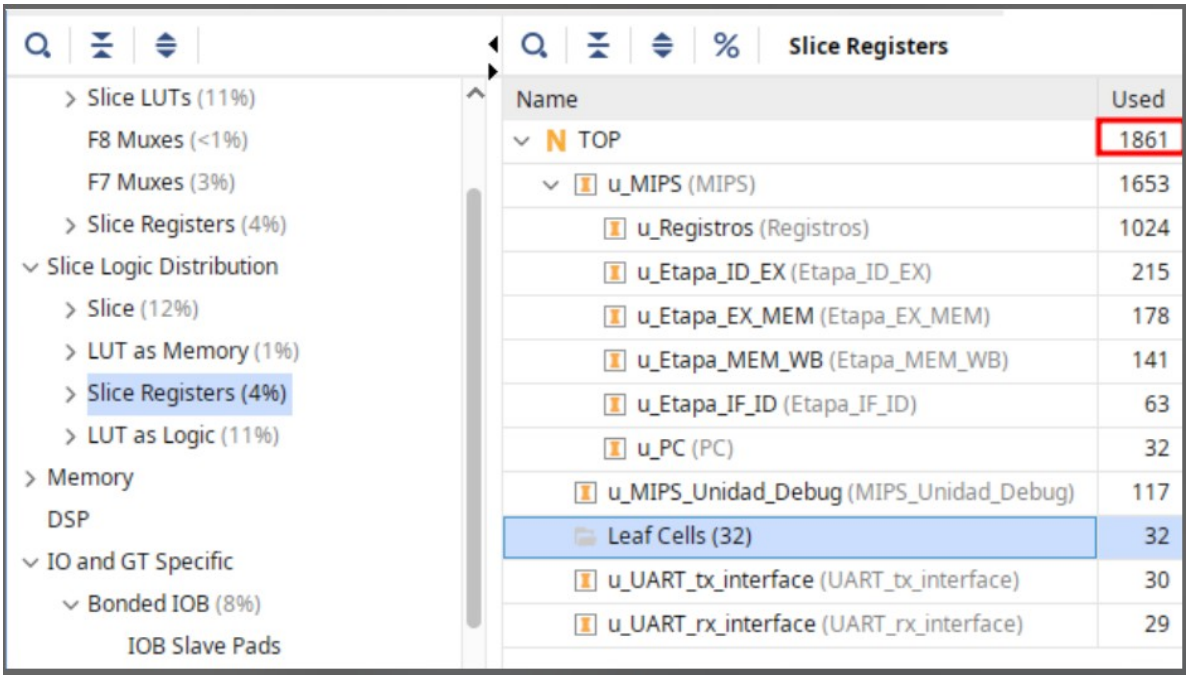
```
ClockCycles: 00000000 00000000 00000000 00001100 -> 12

ProgramCounter:
00000000 00000000 00000000 00110100 -> 52

REGISTER MEMORY:
0:  00000000 00000000 00000000 00000100 -> 4
1:  00000000 00000000 00000000 00000001 -> 1
2:  00000000 00000000 00000000 00000010 -> 2
```

```
ClockCycles: 00000000 00000000 00000000 00001101 -> 13

ProgramCounter:
00000000 00000000 00000000 00110100 -> 52

REGISTER MEMORY:
0:  00000000 00000000 00000000 00000100 -> 4
1:  00000000 00000000 00000000 00000000 -> 0
2:  00000000 00000000 00000000 00000010 -> 2
```

Utilization report

Para obtener una primera aproximacion de la cantidad de flip flops utilizados en el proyecto se listo la cantidad de registros y se calcula la cantidad de celdas utilizadas en todos estos registros, esta cuenta nos dio como resultado 5583, contando la totalidad de los registros asi como el numero maximos de celdas declaradas para las memorias.
Este numero se reducira en gran medida ya que no utilizamos las memorias en su totalidad, sin tener en cuenta la optimizacion de diseño que aplica Vivado al lanzar la implementacion.

A continuacion podemos ver el reporte de cantidad de Slice Registers (flip flops registers), en post implementacion:



Luego de la implementacion vemos que la cantidad de FFs disminuye un 60% aproximadamente.