Cátedra de Arquitectura de Computadoras

Trabajo Práctico Nº IV - PROCESADOR MIPS SIMPLIFICADO

Integrantes:
López Gastón (gopezlaston@gmail.com)
Kleiner Matías (kleiner.matias@gmail.com)

Fecha:
17 de abril de 2019

Introducción

Para el Trabajo Práctico N° 4, se realizó la implementación en Verilog del pipeline del procesador **MIPS**. El mismo se trata de un procesador de 32 bits, con 32 registros, una memoria de instrucciones de 2048 posiciones, y una memoria de datos de 1024 posiciones, ambos valores parametrizables. El MIPS se conectó a un módulo UART, el cual había sido diseñado e implementado en el Trabajo Práctico N° 2. El sistema se completa con una PC, a partir de la cual se envían y se reciben comandos, ACKs y datos de monitoreo. Además es de destacar que el pipeline del procesador se conecta al mencionado módulo UART vía otro módulo, denominado Debug Unit. Para la implementación se usará una FPGA, la cual es un dispositivo programable que contiene bloques lógicos que, al interconectarlos, se genera una funcionalidad. La mencionada interconexión y funcionalidad se configuran mediante un lenguaje de descripción de hardware (Verilog).

Resumen de la consigna

El procesador MIPS debe estar implementado con las siguientes etapas de pipeline:

- **IF (Instruction Fetch):** búsqueda de la instrucción en la memoria de programa.
- **ID (Instruction Decode):** decodificación de la instrucción y lectura de los 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.

Las instrucciones a implementar del set de instrucciones del MIPS son las siguientes:

R-Type:

1. SLL: Shift Word Left Logical.
2. SRL: Shift Word Right Logical.
3. SRA: Shift Word Right Arithmetic.
4. SLLV: Shift Word Left Logical Variable.
5. SRLV: Shift Word Right Logical Variable.
6. SRAV: Shift Word Right Arithmetic Variable.
7. ADDU: Add Unsigned Word.
8. SUBU: Subtract Unsigned Word.
9. AND: and.
10. OR: or.
11. XOR: exclusive or.
12. NOR: nor.
13. SLT: Set on Less Than.

J-Type:

1. JR: Jump Register.
2. JALR: Jump and Link Register.

I-Type:

1. LB: Load byte.
2. LH: Load Halfword.
3. LW: Load Word.
4. LWU: Load Word unsigned.
5. LBU: Load byte unsigned.
6. LHU: Load Halfword Unsigned.
7. SB: Store byte.
8. SH: Store HalfWord.
9. SW: Store Word.
10. ADDI: Add Immediate Word.
11. ANDI: And Immediate.
12. ORI: Or Immediate.
13. XORI: Exclusive Or Immediate.

14. LUI: Load Upper Immediate.
15. SLTI: Set on Less Than Immediate.
16. BEQ: Branch on Equal.
17. BNE: Branch on Not Equal.
18. J: Jump.
19. JAL: Jump and Link.

El procesador MIPS debe poseer soporte para los siguientes tipos de riesgos:

- **Estructurales:** se genera cuando dos instrucciones en el mismo ciclo intentan utilizar el mismo recurso.
- **De datos:** se genera cuando se intenta utilizar un dato antes de que esté preparado. Se debe mantener el orden estricto de lecturas y escrituras.
- **De control:** se genera cuando se intenta tomar una decisión sobre una condición no evaluada todavía.

Dos soluciones para los riesgos de datos son:

- Unidad de cortocircuitos.
- Unidad de detección de riesgos.

Otros requerimientos:

- El programa a ejecutar debe ser cargado en la memoria de programa mediante un archivo ensamblado.
- Se debe implementar un programa ensamblador.
- Debe transmitirse ese programa mediante interfaz UART antes de comenzar a ejecutar.

- Se debe incluir una unidad de Debug (denominada en el presente trabajo Debug Unit). Dicha unidad debe enviar información hacia y desde la PC mediante la UART.
- Se deben enviar a la PC a través de la UART:
    - Contenido de los 32 registros.
    - Contenido de los latches intermedios.
    - PC.
    - Contenido de la memoria de datos usada.
    - Cantidad de ciclos de clock desde el inicio.
- Antes de estar disponible para ejecutar, el procesador está a la espera para recibir un programa mediante la Debug Unit.
- Una vez que se carga el programa en la memoria de programa, se deben permitir dos modos de operación:
    - Continuo: se envía un comando a la FPGA por la UART y ésta inicia la ejecución del programa hasta llegar al final del mismo (instrucción HALT). Llegado a ese punto, se muestran todos los valores mencionados anteriormente.
    - Paso a paso (se denomina modo debug en el presente trabajo práctico): se envía un comando por la UART y se ejecuta un solo ciclo de clock. Mostrar en PC en cada paso los valores anteriormente mencionados.

Herramientas y elementos utilizados

Entre los dispositivos disponibles para la implementación del MIPS integrado con la UART, se seleccionó la placa de desarrollo ARTY, que puede verse en la siguiente figura.

La principal razón de la selección fue que esta placa es una de las que soportan la interfaz de desarrollo Vivado, sobre la cual poseemos algo de experiencia previa. Cabe destacar que la FPGA que contiene la placa es una Artix-7 de Xilinx.

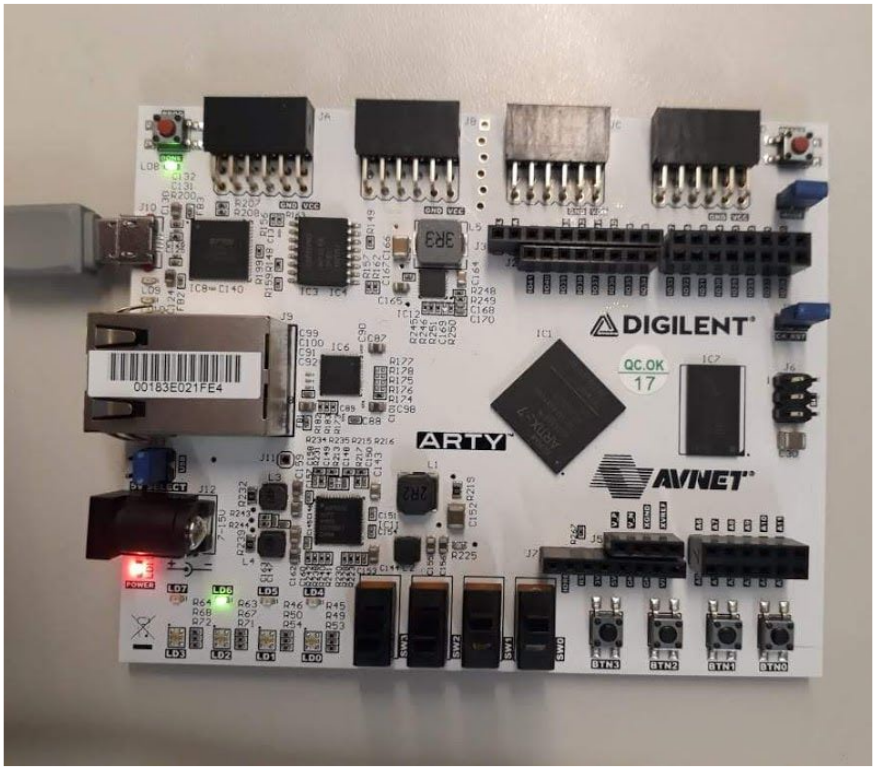


Figura 1 - FPGA Utilizada para el desarrollo del proyecto.

Diseño - Módulo TOP.

Se definió un módulo Top, que contiene la instanciación de los siguientes módulos:

- Tx.
- Rx.
- Baud Rate Generator.
- Contador de ciclos.
- Clock Wizard.
- Debug Unit.
- Top IF.
- Top ID.
- Top Ejecución.
- Top Memoria de Datos.
- Top Write Back.
- Database.
- Forwarding Unit.
- Hazard Detection Unit.

Las entradas y salidas de este módulo son:

- **input i_clock_top:** clock (100 MHZ).
- **input i_reset:** reset por hardware.
- **input i_switches:** switches de la placa para reset del IP core Clock Wizard.
- **input uart_txd_in:** transmisor de PC.
- **output uart_rxd_out:** receptor de PC.
- **output o_leds:** leds de la placa (usados para pruebas y test).
- **output led0:** leds RGB de la placa (usados para pruebas y test).

El baud rate (parametrizable) utilizado en este trabajo práctico es de 9600 bps, con una frecuencia del clock de 50 MHz. La cantidad de bits de stop es 2 y la cantidad de bits de datos es 8, generando con el bit de start una trama con longitud igual a 11 bits.

Durante el desarrollo del presente trabajo, se tuvo un problema con el timing, debido a la gran cantidad de registros instanciados. La señal de clock llega al registro antes de que se estabilice el valor correspondiente en la entrada de dicho registro.

Para solucionar este inconveniente existen diversas formas, como por ejemplo, usar la herramienta de ruteo que brinda Vivado para posicionar de forma diferente el registro con problemas de timing. Otra forma (que fue la utilizada por el grupo para resolver el problema) es reducir la frecuencia del clock, generando así que se logre estabilizar el valor en la entrada del registro para cuando llegue el flanco del mencionado clock. Para lograr esto, se instanció un bloque IP core que brinda la herramienta (Clock Wizard), en la que se especificó una frecuencia de entrada de 100 MHz y una de salida de 50Mhz. La única utilidad de este bloque es generar una señal de reloj con la frecuencia deseada a partir de la frecuencia base de la placa (100Mhz). Esta nueva señal de reloj de 50Mhz es la que sirve de clock a todos los demás módulos del proyecto. Finalmente, se logró resolver el problema de timing.

Por otra parte, es importante destacar que para lograr que el MIPS posea un rendimiento de una instrucción por ciclo, fue necesario diseñarlo e implementarlo teniendo en cuenta lo siguiente:

- Con el flanco descendente del clock, se actualizan los latches intermedios que separan las etapas del pipeline.
- Con el flanco ascendente del clock, todos los registros presentes en el interior de las etapas del pipeline se actualizan.
- Las memorias trabajan con el flanco ascendente del clock.
- La escritura de registros en el banco de registros se efectúa en el flanco ascendente del clock, con el fin de evitar riesgos estructurales, ya que la lectura se realiza con el flanco descendente.
- Todos los demás módulos que se encuentran fuera de las etapas del pipeline trabajan con el flanco ascendente del clock.
- Con el flanco descendente del clock, se actualiza el contador de programa.

Esquemático general del proyecto.

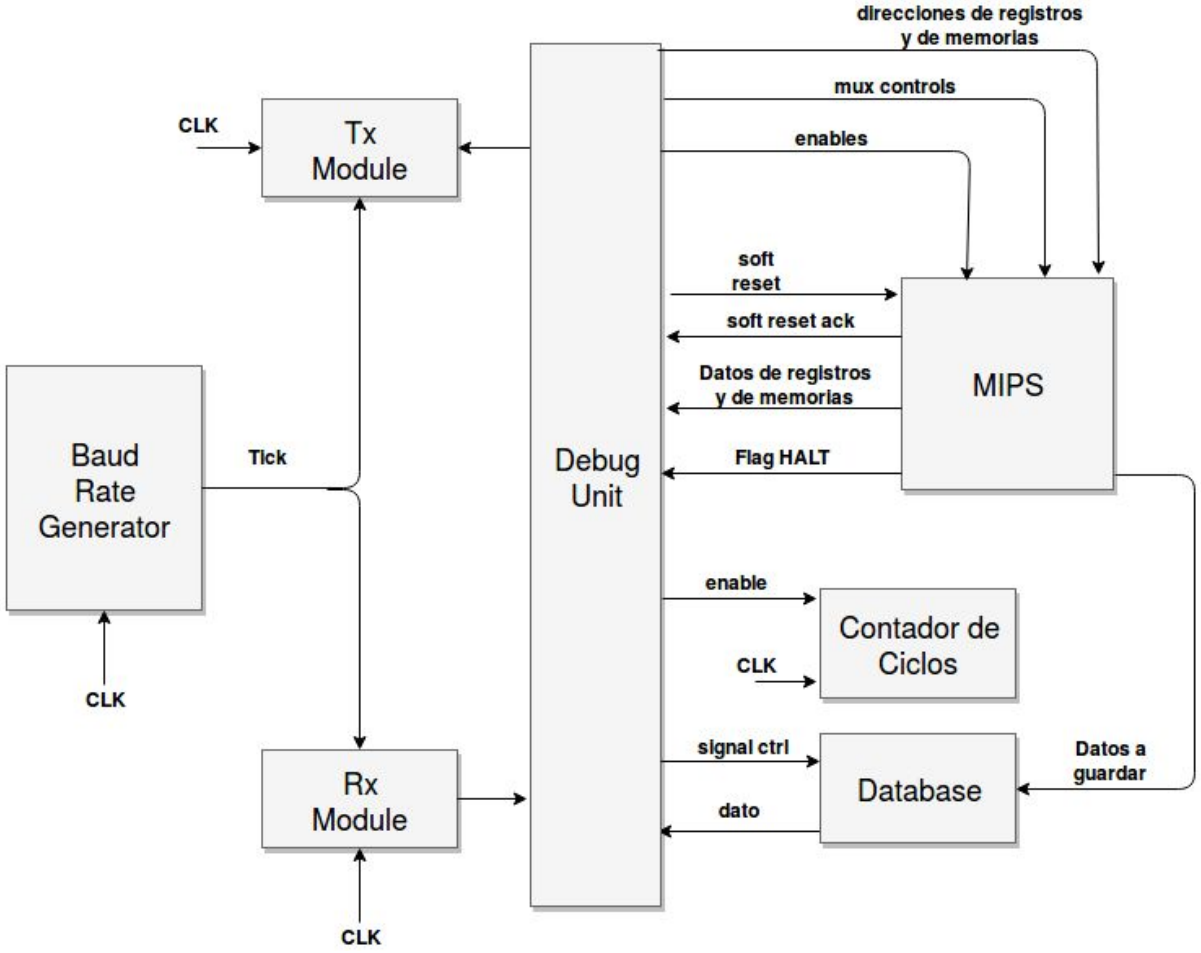


Figura 2 - Esquema general del proyecto.

Esquemático general del bloque MIPS.

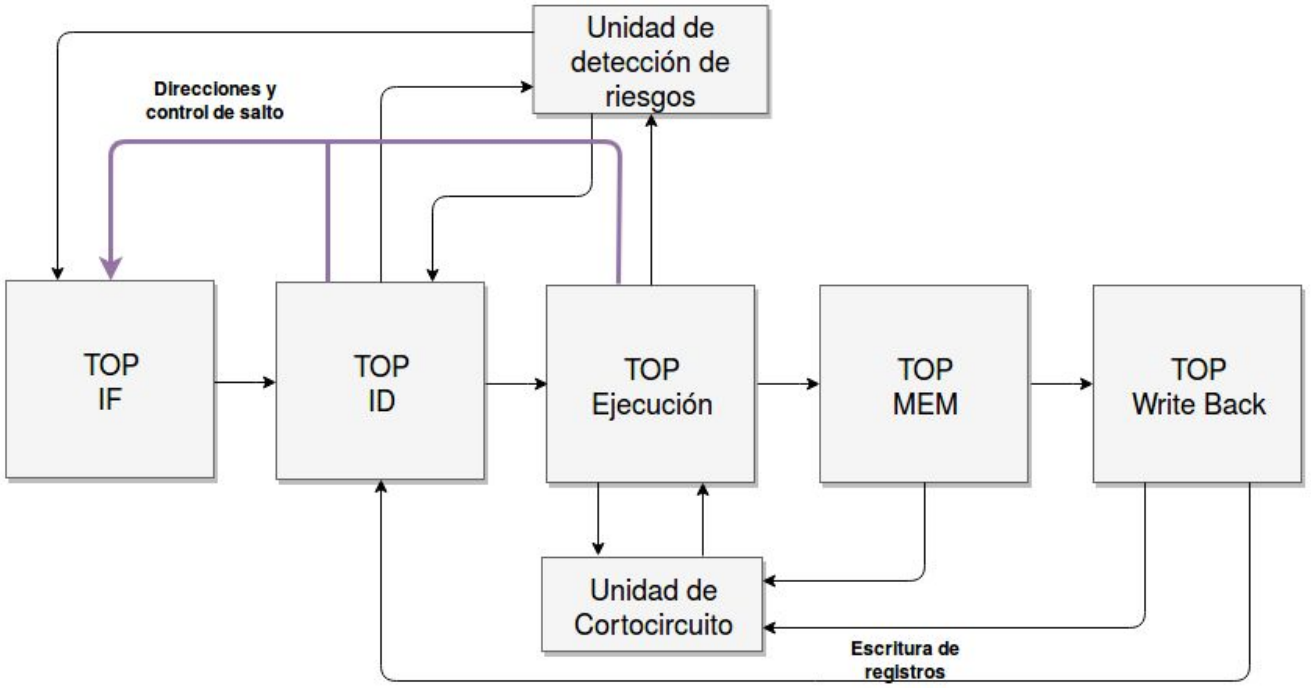


Figura 3 - Esquema general del bloque MIPS.

Diseño - Módulo Baud Rate Generator.

Este módulo genera ticks con los que alimenta a los módulos Tx y Rx para coordinar las señales recibidas o transmitidas. La generación de ticks es 16 veces por baud rate. Consiste en un contador, que cuando alcanza un límite, genera un tick. Para calcular dicho límite la fórmula es la siguiente:

$$Límite = \frac{Clock}{Baud\ Rate * 16}$$

Diseño - Módulo RX.

Consiste en una máquina de estados. La representación de estados utilizada es la one-hot. Observando la Figura 4, se tiene que para pasar del estado de Espera al estado de Start se requiere la llegada del bit de start (bit en nivel bajo). Cuando se cuentan 8 ticks se está en la mitad de dicho bit de start que se recibió, por lo que se cambia al estado Read. En este estado, se cuentan los 8 bits de datos (cada 16 ticks se toma el valor recibido debido a que se está en el medio del bit transmitido desde la PC). Luego de contar esos 8 bits (valor parametrizable) se pasa al estado de Stop. Allí se espera una cantidad de bits de stop (en nivel alto) igual al valor definido en la parametrización del módulo. Cuando llegan dichos bits de stop, se coloca el bit Rx Done en alto y en la salida del módulo Rx también se inserta el dato recibido (los 8 bits de dato). Luego se pasa al estado de Espera. En caso de que se necesiten 2 bits de stop, si el primero que llega es un 1 y el segundo un cero, se desincroniza todo el sistema debido a que faltan 8 ticks para pasar el segundo bit de stop. Esta tarea de dejar pasar dicha cantidad de ticks se realiza en el estado de Error.

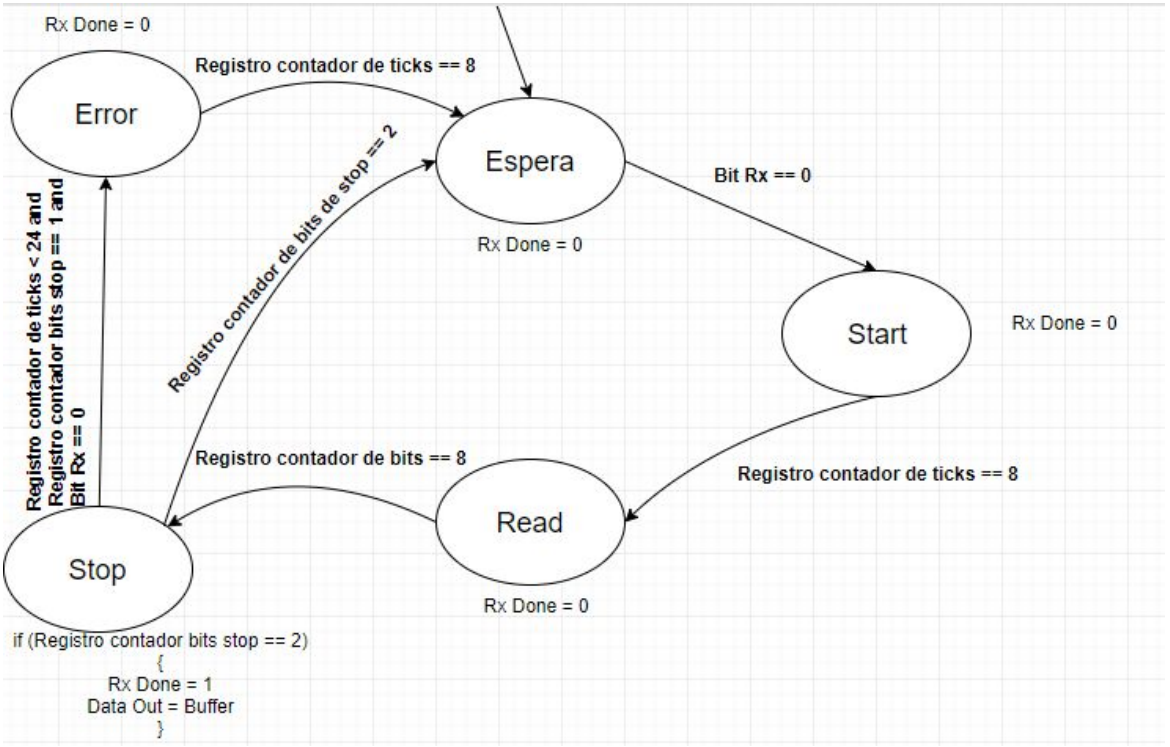


Figura 4 - Máquina de Estados del módulo Rx.

Diseño - Módulo TX.

Consiste en una máquina de estados similar a la anterior. La representación de estados utilizada es la one-hot. Observando la Figura 5, se tiene que para pasar del estado de Espera al estado de Start se requiere un nivel en alto de la señal Tx Start. Luego el pasaje entre los demás estados se debe a la finalización en la generación de los distintos bits de la trama. Mientras se están transmitiendo dichos bits la salida Tx Done se encuentra en un nivel bajo para que el módulo Interface Circuit no envíe otro dato a transmitir. En los estados Read y Stop las salidas están sujetas a condiciones para prevenir errores, colocándose en el diagrama anterior aquellas más representativas.

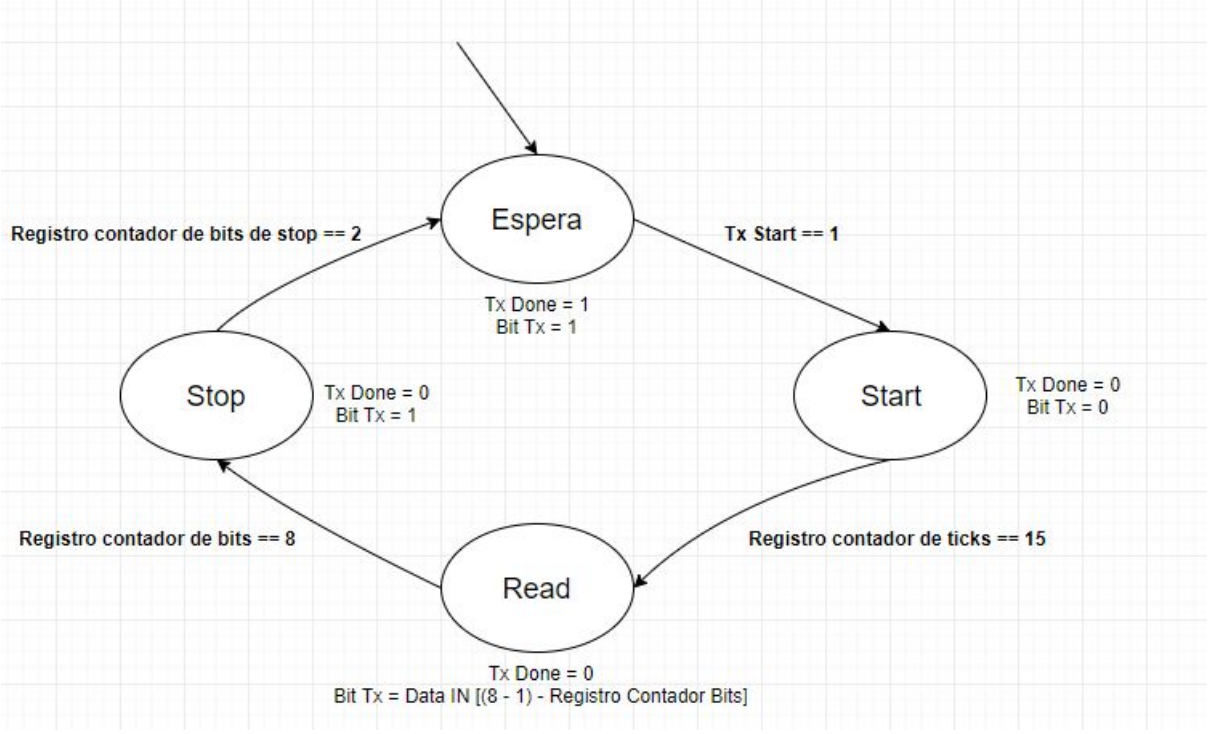


Figura 5 - Máquina de Estados del módulo Tx.

Diseño - Módulo Contador de Ciclos.

Su función es la de llevar la cuenta de la cantidad de pulsos de clock (incrementando la misma en los flancos ascendentes del mismo), siempre y cuando la entrada i_enable del mismo se encuentre en alto. De lo contrario la cuenta se mantiene.

Las entradas y salidas de este módulo son:

- **input *i_enable*:** dicha entrada habilita la cuenta si se encuentra en 1, de lo contrario la mantiene.
- **input *i_clock*:** entrada de clock al módulo, utilizada para actualizar los registros.
- **input *i_soft_reset*:** resetea el registro que lleva la cuenta de los pulsos de clock cuando se encuentra en 0 (activo por bajo).
- **output *o_cuenta*:** registro de 10 bits que lleva la cuenta de la cantidad de pulsos de clock. (Son 10 bits para igualar su tamaño al del contador de programa).

Diseño - Módulo Clock Wizard.

Solución utilizada para bajar la frecuencia base de la placa de 100Mhz a 50Mhz. El módulo es un IP Core perteneciente a Xilinx y es el que brinda el clock a todos los demás módulos del proyecto.

Diseño - Módulo Debug Unit.

Consiste en una máquina de 13 estados. Tiene las tareas de:

- Sincronizar el estado del MIPS.
- Permitir llevar a cabo los dos modos de ejecución, el continuo y el paso o paso.

- Comunicar el módulo UART con el procesador MIPS. De esta forma desde la PC es posible enviar datos a dicho procesador y viceversa.

Estados:

- **ESPERA:** estado inicial. Funciona como punto de partida y sincronización con la GUI. Espera un dato enviado (8'b00000000) desde la PC para pasar al siguiente estado.

- **SOFT_RESET:** se pone en bajo la señal de soft_reset (reset por software) y todos los módulos del MIPS resetean sus registros. Las memorias de programa y de datos resetean todo su contenido y cuando finalizan ambas dicho reseteo envían en forma conjunta un bit en bajo de reset ACK al Debug Unit. Cuando la máquina de estados recibe dicho ACK, pasa al siguiente estado.

- **ESPERA_PC_ACK:** dicho estado pone en uno el soft_reset y se informa a la PC que ya se terminaron de resetear todos los registros a través del envío del byte 8'b00000000 a la misma. Permanece en este estado hasta que la mencionada PC envíe al Debug Unit el dato 8'b00000001 que indica que se va a comenzar a cargar las instrucciones en la memoria de programa.

- **READ_PROGRAMA:** recibe las instrucciones enviadas desde la PC y mediante un contador las va cargando en la memoria de programa. Cuando la instrucción enviada es el HALT, pasa al siguiente estado. De lo contrario permanece en READ_PROGRAMA, moviendo el puntero de la dirección de memoria de programa, como se mencionó anteriormente.

- **ESPERA_START:** en este estado se especifica qué modo de ejecución debe efectuar el MIPS y se da la orden de comienzo de ejecución al procesador. Se admiten dos posibles datos enviados desde la PC para pasar al siguiente estado (EJECUCIÓN):

8'b00000011 para modo continuo o 8'b00000111 para modo debug. En el modo debug a este estado se llega cada vez que se completa un ciclo de clock de ejecución de la instrucción y se envían a la PC los datos correspondientes a dicho paso de ejecución.

- **EJECUCIÓN:** ejecuta las instrucciones según el modo de ejecución dado en el estado anterior. Si el modo fue indicado como continuo, pasará al siguiente estado si la instrucción HALT llegó a la última etapa, es decir, la de Write-Back. Por otra parte, si el modo de ejecución seteado fue el modo debug, pasará al siguiente estado en el segundo flanco ascendente de clock independientemente de la instrucción ejecutada. Lo mencionado sobre el segundo flanco es debido a lo siguiente:

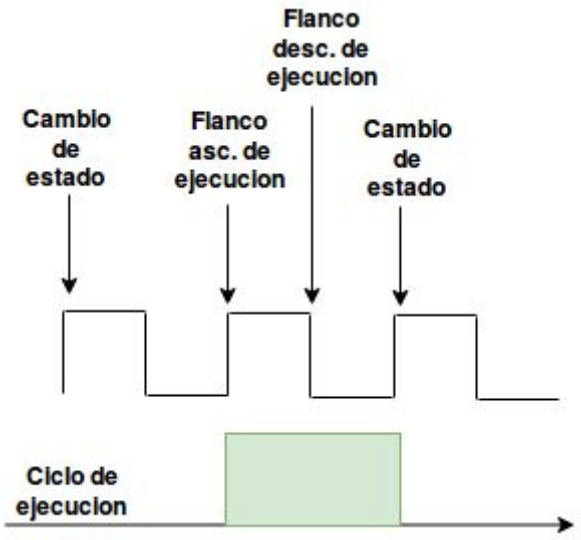


Figura 6 - Ciclo de ejecución en modo Debug.

Si no se respeta el orden de los flancos y, por ejemplo, se comienza con el flanco descendente, se pierde la sincronización en cuanto a los valores que se esperan leer desde la PC, debido a que solamente la instrucción se debe ejecutar durante un solo ciclo de clock en cada paso del modo debug.

- **SEND_PART3:** Estado de envío de datos de monitoreo. Se envían los bits 31-24 de los 32 bits a enviar a la PC. Pasa al siguiente estado si la PC envía 8'b00001000. Dependiendo de

los contadores internos del módulo puede enviar datos del módulo database, de la memoria de datos o del banco de registros.

- **SEND_PART2:** Estado de envío de datos de monitoreo. Se envían los bits 23-16 de los 32 bits a enviar a la PC. Pasa al siguiente estado si la PC envía 8'b00010000. Dependiendo de los contadores internos del módulo puede enviar datos del módulo database, de la memoria de datos o del banco de registros.

- **SEND_PART1:** Estado de envío de datos de monitoreo. Se envían los bits 15-8 de los 32 bits a enviar a la PC. Pasa al siguiente estado si la PC envía 8'b00011000. Dependiendo de los contadores internos del módulo puede enviar datos del módulo database, de la memoria de datos o del banco de registros.

- **SEND_PART0:** Estado de envío de datos de monitoreo. Se envían los bits 7-0 de los 32 bits a enviar a la PC. Pasa al siguiente estado si la PC envía 8'b00100000. Dependiendo de los contadores internos del módulo puede enviar datos del módulo database, de la memoria de datos o del banco de registros. Presenta una lógica de cambio de estado que consiste en lo siguiente:
    - Si el contador de datos enviados del Database posee un valor menor a la cantidad de datos que debe mandar realmente desde dicho módulo, pasa al estado SEND_PART3.
    - Si el contador de datos enviados del Database posee un valor igual a la cantidad de datos que debe mandar realmente desde dicho módulo y el contador de envío de datos del banco de registros es menor a la cantidad de registros, entonces pasa al estado REGISTROS_DATA_CHECK.

    - Si el contador de datos enviados del Database posee un valor igual a la cantidad de datos que debe mandar realmente desde dicho módulo y el contador de envío de datos del banco de registros es igual a la cantidad de registros, entonces pasa al estado MEM_DATOS_CHECK.

- **REGISTROS_DATA_CHECK:** actualiza el contador de datos enviados del banco de registros y pasa al estado SEND_PART3.

- **MEM_DATOS_CHECK:** controla que se envíen tanto los datos como las direcciones de la memoria de datos que modificó el MIPS. La última dirección de memoria es siempre enviada junto con el valor que almacena, por más que no se haya modificado. Pasa al estado de ESPERA_MEM_DATOS_CHECK_ACK si se envió la última dirección de la memoria de datos y el dato correspondiente a dicha dirección. Por otra parte, pasa al estado SEND_PART_3 si:
    - Se llegó a la última dirección de la memoria de datos y todavía no se terminó de completar el envío del dato y de la dirección correspondiente.
    - En una dirección anterior, el bit de sucio está en alto y todavía no se terminó de completar el envío del dato y de la dirección correspondiente.

- **ESPERA_MEM_DATOS_CHECK_ACK:** si la PC envía el dato 8'b00101000 puede ocurrir lo siguiente:
    - Pasa al estado de ESPERA si se encuentra el procesador en modo continuo de ejecución o en modo debug pero con la instrucción de HALT en la última etapa del pipeline.
    - Pasa al estado ESPERA_START si se encuentra el procesador en modo debug pero sin la instrucción HALT en la última etapa del pipeline.

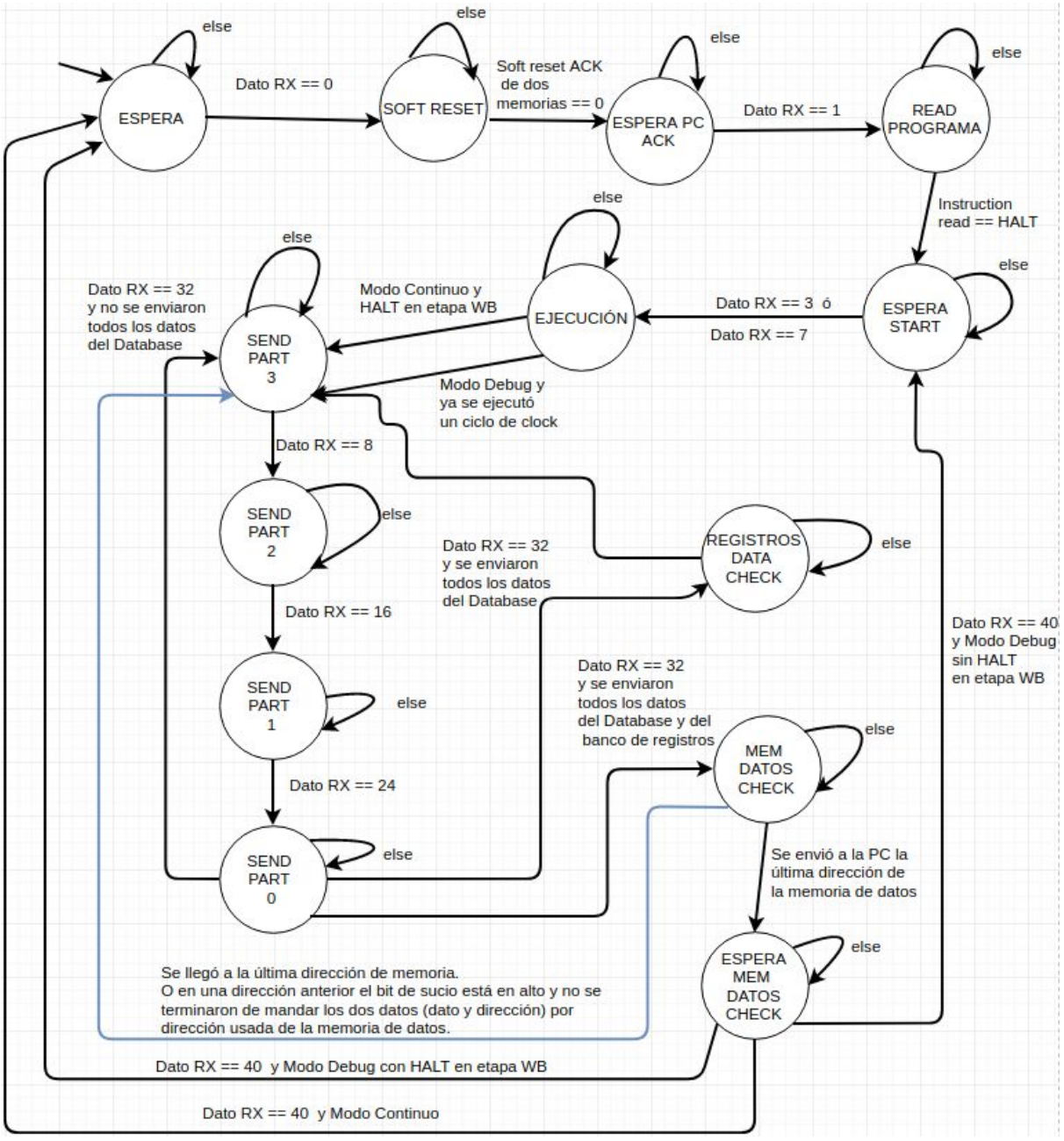


Figura 7 - Máquina de estados del Debug Unit.

Diseño - Módulo Top IF

En este módulo, la memoria de programa trabaja con el flanco ascendente del clock. Por otra parte, el contador de programa y los registros de salida, como por ejemplo, el IR y el Adder PC Register, utilizan el otro flanco del ciclo de reloj (flanco descendente).

Cabe destacar que las siguientes señales provienen del módulo Debug Unit:

- Soft reset.
- Enables.
- Señales de control de la memoria de programa.
- Control MUX Address.

A su vez, es importante mencionar que para la memoria de programa sintetizada como Block RAM por la herramienta Vivado, se utilizó el template Xilinx Single Port Byte-Write Read First RAM.

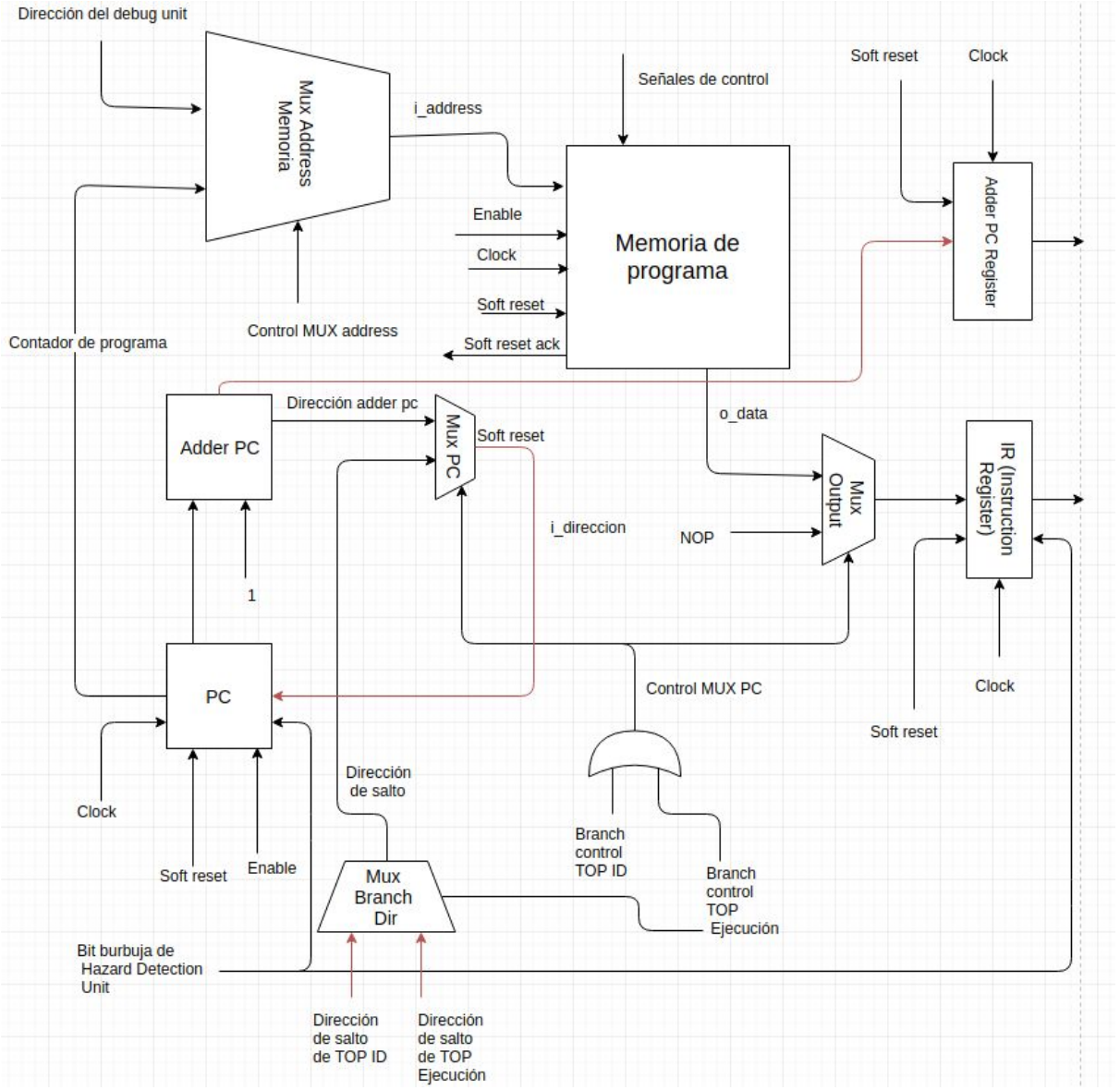


Figura 8 - Esquema del TOP IF.

Diseño - Módulo Top ID.

En este módulo, los distintos submódulos cumplen la siguiente función:

- **Decoder:** módulo combinacional que decodifica la instrucción que proviene de la etapa Instruction Fetch. De este módulo salen las direcciones de registros rs, rt, rd, reg A y reg B. A su vez, otras salidas importantes son el reconocimiento de los saltos, el valor inmediato de la instrucción, el valor del instruction index branch para los saltos J y JAL y el reconocimiento de la instrucción HALT.
- **Register File:** es el banco de 32 registros de 32 bits cada uno. En el flanco ascendente del clock, se escriben con los valores y las señales de control que provienen de la etapa Write Back. Por otra parte, en el flanco descendente se colocan en la salida los valores que contienen los registros apuntados por las direcciones reg A y reg B. Contiene además una función auxiliar que le otorga al Debug Unit el poder de extraer el valor que contiene un registro colocando su puntero en una entrada específica para tal fin.
- **Control:** prácticamente combinacional, es un módulo que genera las señales de control para las etapas EX, MEM y WB para la instrucción correspondiente. Dichas señales de Control son las siguientes:
    - AluOP.
    - AluSrc.
    - AluCtrl.
    - RegDst.
    - MemtoReg.
    - RegWrite.
    - MemRead.
    - MemWrite.
    - Select Bytes Mem Datos: esta señal de control no se encuentra en la bibliografía y se utilizó en el presente trabajo para enviar información a la etapa de la memoria de datos sobre el tamaño de los datos que manejan las

operaciones de load y store (byte, halfword o word) y sobre si dichos datos que se utilizan son signados o no signados.

- **Branch Address Calculator Low Latency:** si la instrucción es un J o JAL, con este módulo combinacional se logra perder un solo ciclo de clock. Además debido a que el J o JAL no utilizan registros, este módulo deshabilita por precaución mediante un bit de control el módulo de detección de riesgos.

Cabe destacar que las siguientes señales provienen del módulo Debug Unit:

- Soft reset.
- Enables.
- Dirección de lectura de registro en el banco de registros para enviar el dato correspondiente hacia la PC.

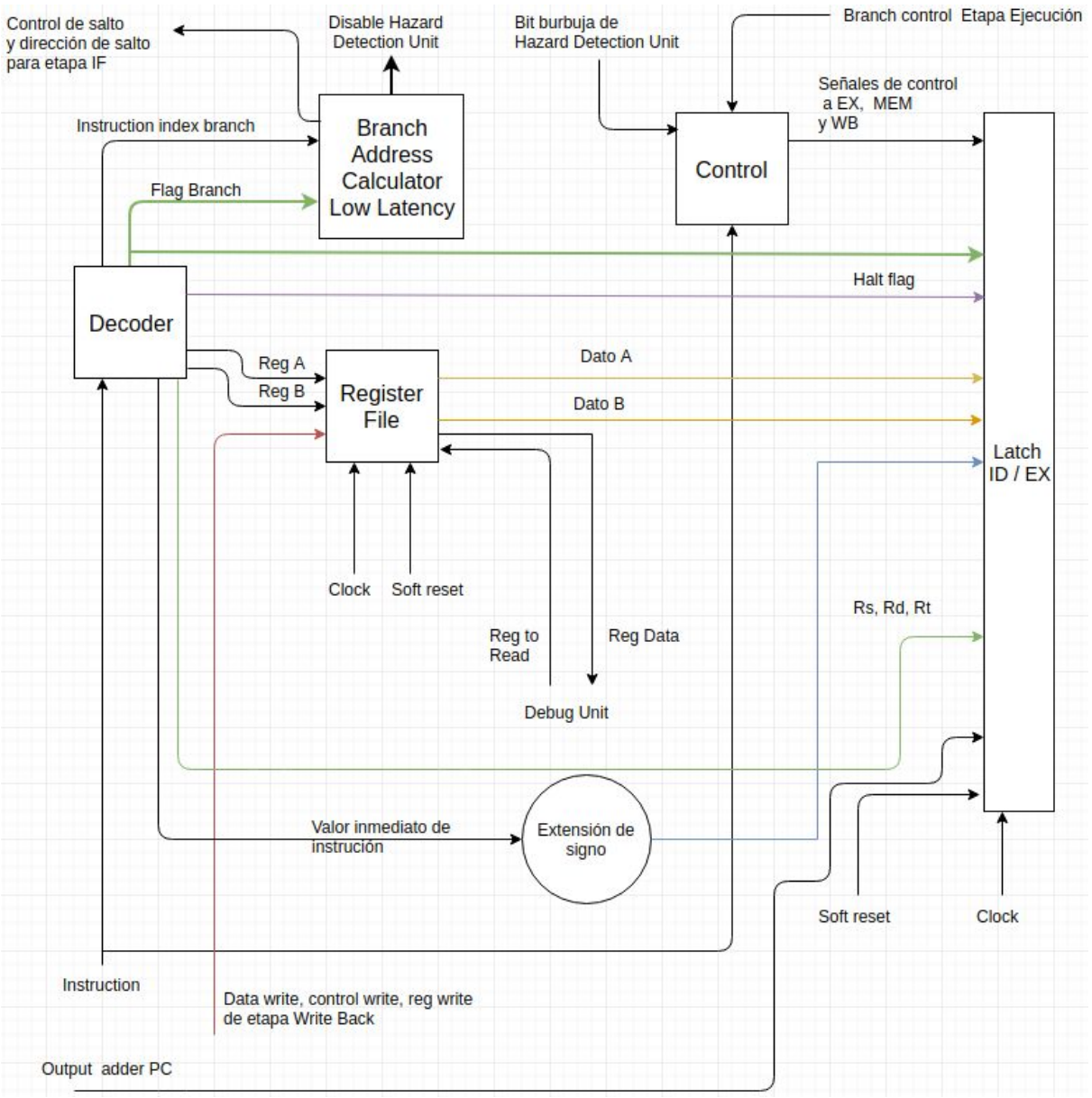


Figura 9 - Esquema del TOP ID.

Diseño - Módulo Top Ejecución.

En este módulo, los distintos submódulos cumplen la siguiente función:

- **Muxs:** generan la lógica adecuada para colocar en la entrada de la ALU los operandos correctos y/o más actualizados (en caso de requerir cortocircuito). A diferencia de la bibliografía, se agregó un multiplexor más para el primer operando, debido a que para

algunas instrucciones de salto, lo que se debe escribir en los registros es la dirección de retorno (PC + 1).

- **ALU:** es un módulo combinacional que efectúa la operación solicitada entre los dos operandos de sus entradas en función de la señal de control ALU Ctrl.
- **Branch Address Calculator High Performance:** este módulo combinacional genera las señales de branch control y branch dir correspondientes a las instrucciones JR, JALR, BNE y BEQ. Para la dos primeras instrucciones, se pierden dos ciclos de clock. Por otra parte, para las dos últimas se pierde la misma cantidad si la condición resulta verdadera, no perdiéndose ninguno en caso de que se evalúe falsa.

Cabe destacar que las siguientes señales provienen del módulo Debug Unit:

- Soft reset.
- Enables.

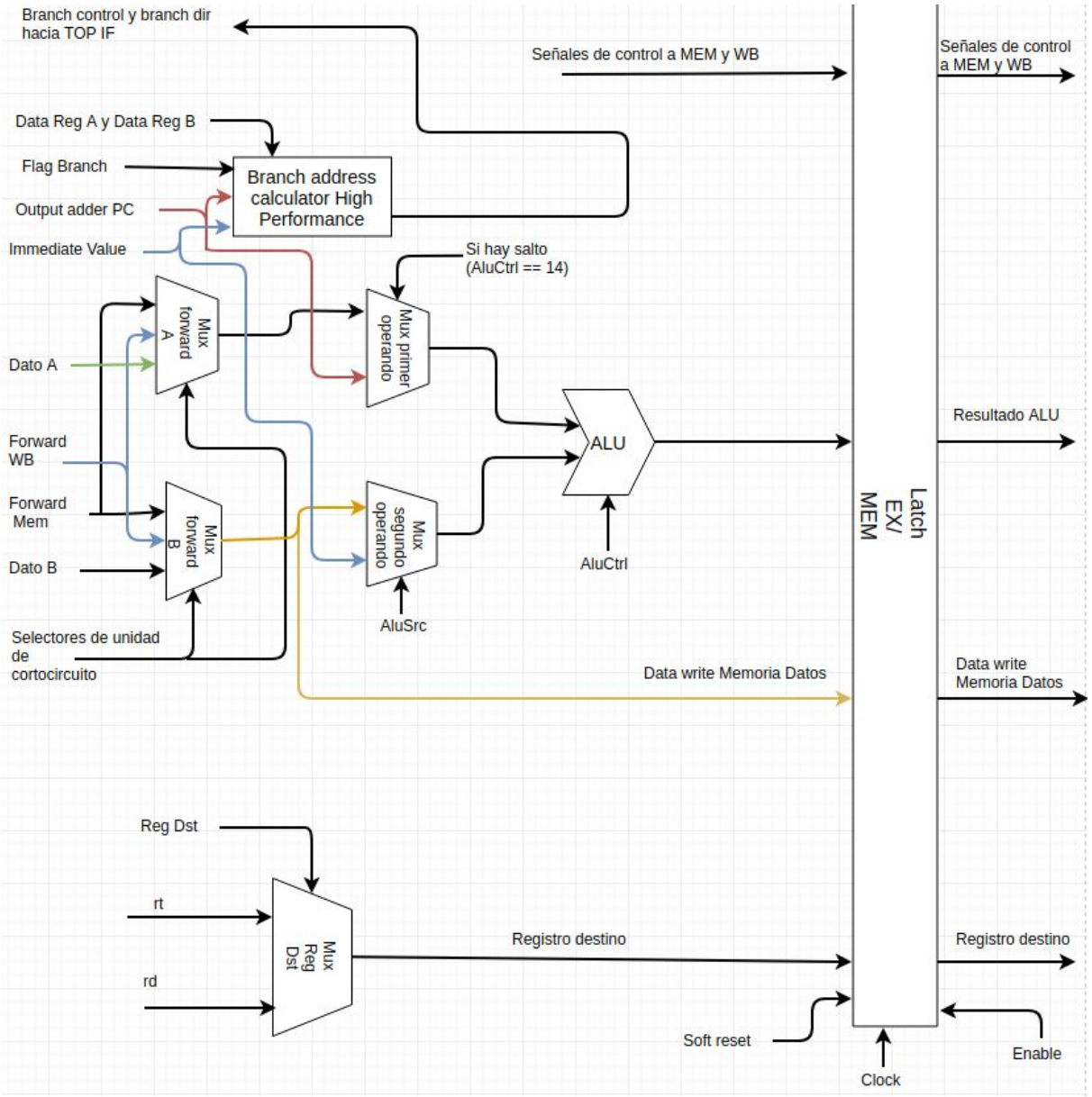


Figura 10 - Esquema del TOP Ejecución.

Diseño - Módulo Top Memoria de Datos.

En este módulo, los submódulos memoria de datos y control de bit de sucio trabajan con el flanco ascendente del clock. Por otra parte los registros de salida utilizan el otro flanco del ciclo de reloj (flanco descendente).

Cabe destacar que las siguientes señales provienen del módulo Debug Unit:

- Soft reset.
- Enables.

Además, el bit de sucio se envía como salida al Debug Unit.

Las lógicas combinacionales de entrada y de salida de este módulo se utilizan para cumplir con los requisitos que demandan las instrucciones de tipo Load o Store, en cuanto a:

- Si el valor devuelto de la memoria debe ser signed o unsigned.
- Si la instrucción señala a un determinado byte, a un determinado halfword o a la palabra completa de 32 bits (word).

Es importante mencionar que los 2 bits menos significativos del bus de direcciones permiten señalar a un byte dentro de los 4 bytes que conforman una word.

Por último, para la memoria de datos sintetizada como Block RAM por la herramienta Vivado, se utilizó el template Xilinx Single Port Byte-Write Read First RAM.

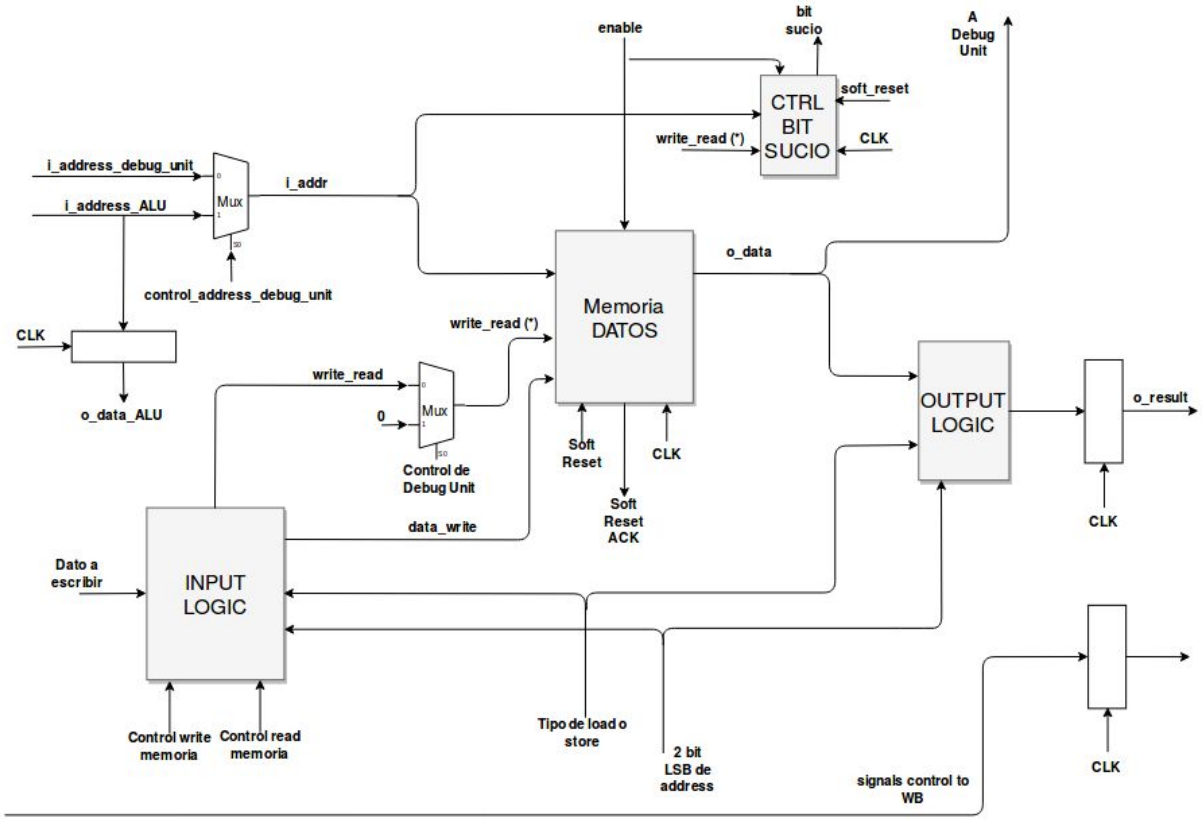


Figura 11 - Esquema del TOP Memoria de Datos.

Diseño - Módulo Top Write Back

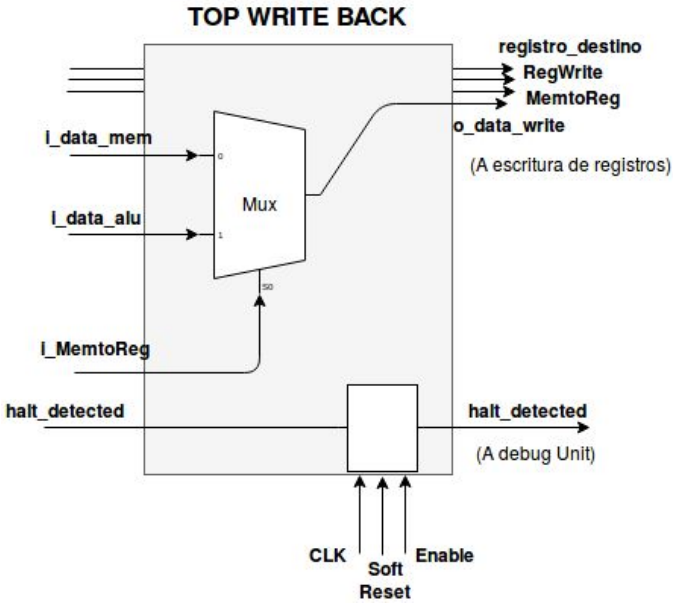


Figura 12 - Esquema del TOP Write Back.

Diseño - Módulo Database.

El módulo Database tiene la función de almacenar en registros la mayoría de los datos que se deben enviar a la PC, tal como lo especifica la consigna. Es controlado por el módulo Debug Unit, quien le indica qué dato necesita enviar a la PC en un momento dado, siendo el Database quien se lo proporciona. A su vez, el Debug Unit es el encargado de indicarle al mencionado módulo Database en qué momento debe actualizar los valores de los registros.

Diseño - Módulo Forwarding Unit.

Permite proveer a la etapa de Execute del pipeline, los argumentos necesarios que hayan sido calculados uno o dos ciclos antes. Los extrae de las etapas Memory Access o Write-Back, respectivamente. Dicho módulo funciona de manera combinacional y soluciona problemas de dependencia de datos.

Las entradas y salidas de este módulo son:

- **input i_rs_ex:** dirección del registro en donde se encuentra el primer operando a usar en la etapa de ejecución.
- **input i_rt_ex:** dirección del registro en donde se encuentra el segundo operando a usar en la etapa de ejecución.
- **input i_registro_destino_mem:** registro destino donde se escribirá el resultado de la instrucción que se encuentra en la etapa Memory Access.
- **input i_registro_destino_wb:** registro destino donde se escribirá el resultado de la instrucción que se encuentra en la etapa Write-Back.
- **input i_reg_write_mem:** señal de control de la etapa Memory Access que indica, cuando se encuentra en alto, que se escribirá un registro del banco de registros en la etapa Write-Back.
- **input i_reg_write_wb:** señal de control de la etapa Write-Back que indica, cuando se encuentra en alto, que se escribirá un registro del banco de registros.

- **output o_selector_mux_A:** selector de un multiplexor que permite introducir el valor cortocircuitado como primer operando de la ALU.
- **output o_selector_mux_B:** selector de un multiplexor que permite introducir el valor cortocircuitado como segundo operando de la ALU.

Condiciones para que se produzca un cortocircuito:

- El registro destino en la etapa de Memory Access o de Write-Back es igual a la dirección dada por rs y/o rt en la etapa Execute.
- La señal de control reg_write en la etapa de Memory Access o de Write-Back debe estar en alto. Debe ser igual a uno en la misma etapa en donde se produce igualdad del registro destino con las direcciones rs y/o rt de la etapa Execute.

Es de destacar que si existe una dependencia de datos simultánea entre la etapa Execute y la de Memory Access y entre la Execute y la de Write-Back, el cortocircuito se hace desde la etapa de Memory Access debido a que allí estará la última instrucción que tocará el registro que genera la dependencia.

Diseño - Módulo Hazard Detection Unit.

Funcionando de manera combinacional, la tarea de este módulo es introducir un stall o burbuja en el pipeline debido al riesgo que se introduce cuando a una instrucción load le sigue una instrucción de tipo R que intenta utilizar el valor leído de memoria. Este módulo impide la carga y actualización del instruction register y del program counter durante un ciclo de clock. Por otra parte, coloca las señales de control en cero con el fin de introducir la mencionada burbuja o stall en el pipeline.

Las entradas y salidas de este módulo son:

- **input i_rs_id:** dirección del registro en donde se encuentra el primer operando a usar. Esta señal parte de la etapa de Instruction Decode.
- **input i_rt_id:** dirección del registro en donde se encuentra el segundo operando a usar. Esta señal parte de la etapa de Instruction Decode.
- **input i_registro_destino_ex:** dirección del registro en donde se escribirá el resultado obtenido de la instrucción load. Esta señal parte de la etapa Execute.
- **input i_read_mem_ex:** esta señal de control, que parte de la etapa Execute, indica con un valor en alto si la instrucción que se encuentra en dicha etapa, leerá el contenido de la memoria de datos. Las instrucciones de tipo load son las únicas que poseen ese bit en uno.
- **input o_bit_burbuja:** si se detecta un riesgo, esta salida se pone en alto. De lo contrario, se encuentra en bajo.
- **input i_disable_for_exception**: se encuentra en alto en caso de que en la etapa de *instruction decode* se encuentre una instrucción J o JAL, en cuyo caso el riesgo no debería detectarse debido a que dichas instrucciones no hacen uso de ningún registro.

Condiciones para la detección de un riesgo:

- ***i_read_mem_ex*** en alto.
- ***i_registro_destino_ex*** igual a:
  - ***i_rs_id*** ó (inclusivo)
  - ***i_rt_id***.
- **i_disable_for_exception** en bajo.

Traducción de assembler a código máquina.

Es importante mencionar que la memoria de programa se carga con instrucciones binarias que se encuentran en un archivo de nombre *init_ram_file.txt.* Para hacer más fácil la tarea de escribir dichas instrucciones binarias, se generó un script en python que traduce el archivo *assembler_MIPS.txt*, que se encuentra escrito en assembler, al archivo *init_ram_file.txt*, escrito en código máquina.

Casos de test.

Además de los test benches realizados para cada módulo que permitieron corroborar el correcto funcionamiento de la descripción del hardware RTL, se confeccionó una hoja de cálculo donde se encuentra el estado de las señales de control frente a cada instrucción que presentamos como test y los valores finales de la memoria de datos y de los registros del banco de registros. Los resultados obtenidos se observaron mediante la GUI en Python, y se corroboraron con los test benches anteriormente descritos. A continuación se describen los casos de test que ponen a prueba el correcto funcionamiento de este proyecto:

Link de la hoja de cálculo mencionada:

https://docs.google.com/spreadsheets/d/1HWEvU7EXdF_0cged2QVcRQxrmoc_oxP6kq4C7UMtupM/edit?usp=sharing

1. Caso de test 1: load y salto.

Se prueban de forma general 3 instrucciones. La primera es una instrucción de load seguida de una instrucción jump a la dirección 8 de memoria de programa. Luego, se prueba la instrucción HALT. Todos los

valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la primer hoja.

Archivo de assembler utilizado:

```
#A 16
#B 8
#C 20
#D -4
#E b101
LWU R1,B{R2}
J 8
HLT
```

2. Caso de test 2: halt.

Se prueba la instrucción HALT. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la segunda hoja.

Archivo de assembler utilizado:

```
HLT
```

3. Caso de test 3: load, store, sumas.

Se prueban 9 instrucciones. Estas instrucciones cumplen como objetivo guardar en una posición de memoria la suma de los valores que se encuentran en otras dos posiciones de memoria. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la tercer hoja.

Archivo de assembler utilizado:

```
#A 10
#B 5
#C 8
#D 0
#E 4
#F 8
//Cargo en R1 un 10
ADDI R1,R2,A
//Cargo en R2 un 5
ADDI R2,R2,B
//Cargo en posicion cero de memoria un 10
SW R1,D{R3}
//Cargo en posicion uno de memoria un 5
SW R2,E{R3}
//Cargo en registro 3 el valor de la posicion cero de memoria
LWU R3,D{R3}
//Cargo en registro 4 el valor de la posicion uno de memoria
LWU R4,E{R4}
// R3 + R4 = 15 (lo guardo en R5)
ADDU R5,R3,R4
//Cargo en posicion dos de memoria el valor de R5
SW R5,F{R7}
HLT
```

4. Caso de test 4: forwarding.

Se prueban 4 instrucciones. En la ejecución de dichas instrucciones es necesario hacer un cortocircuito entre las etapas del pipeline debido a la dependencia de datos entre las mencionadas instrucciones. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la cuarta hoja.

Archivo de assembler utilizado:

```
#A 10
#B 5
#C 8
#D 0
#E 1
#F 2
//Cargo en R1 un 10
ADDI R1,R2,A
//Cargo en R2 un 15 (forward en mem)
ADDI R2,R1,B
//Cargo en R3 un 15 (forward en wb)
ADDI R3,R1,B
HLT
```

5. Caso de test 5: prueba de control de riesgos frente a load e instrucción de tipo R (ORI).

Se prueban 5 instrucciones. En la ejecución de dichas instrucciones es necesario lanzar una burbuja (stall) entre la instrucción de load y la de la operación or debido a la dependencia de datos. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la quinta hoja.

Archivo de assembler utilizado:

```
#A 10
#B 5
#C 8
#D 0
#E 1
#F 2
//Cargo en R1 un 10
ADDI R1,R1,A
//Cargo en posicion cero de memoria el contenido de R1
SW R1,D{R0}
//Cargo en R3 el contenido de la posicion cero de memoria
LWU R3,D{R0}
//Cargo en R4 el resultado de R3 | C
ORI R4,R3,C
HLT
```

6. Caso de test 6: prueba general de instrucciones tipo R.

Se prueban todas las instrucciones de tipo R. Todos los resultados que se obtienen son guardados en diferentes posiciones de memoria con el fin de facilitar el chequeo de los valores obtenidos. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la sexta hoja.

Archivo de assembler utilizado:

```
#A 10
#B 5
#C 8
#D 0
#E 1
#F 2
#G -11
//Cargo en R1 un 10
ADDI R1,R1,A
//Cargo en R2 un 8
ADDI R2,R2,C
//Cargo en R3 un 5
ADDI R3,R3,B
//Cargo en R4 un 2
ADDI R4,R4,F
//Cargo en R5 un 0
ADDI R5,R5,D
//Cargo en R6 el resultado de R1 & R2
AND R6,R1,R2
//Cargo en R7 el resultado de R1 | R2
OR R7,R1,R2
//Cargo en R8 el resultado de R1 ^ R2
XOR R8,R1,R2
//Cargo en R9 el resultado de R1 ~| R2
NOR R9,R1,R2
//Cargo en R10 el resultado de R1 & 2
ANDI R10,R1,F
//Cargo en R11 el resultado de R1 ^ 2
XORI R11,R1,F


//Cargo en R12 el resultado de R1 - R2
SUBU R12,R1,R2
//Cargo en R13 el resultado de R2 - R1
SUBU R13,R2,R1
//Cargo en R14 el resultado de R2 < R1
SLT R14,R2,R1
//Cargo en R15 el resultado de R1 < R2
SLT R15,R1,R2
//Cargo en R16 el resultado de R1 < R13
SLT R16,R1,R13
//Cargo en R17 el resultado de R2 < A
SLTI R17,R2,A
//Cargo en R18 el resultado de R1 < C
SLTI R18,R1,C
//Cargo en R19 el resultado de R1 < G
SLTI R19,R1,G
//Cargo en R20 el resultado de R1 >> 3
SRL R20,R1,3
//Cargo en R21 el resultado de R1 << 2
SLL R21,R1,F
//Cargo en R22 el resultado de R1 >> R4
SRLV R22,R1,R4
//Cargo en R23 el resultado de R1 << R4
SLLV R23,R1,R4
//Cargo en R24 el resultado de R1 >>> 1
SRA R24,R1,E
//Cargo en R25 el resultado de R13 >>> R4
SRAV R25,R13,R4
```

```
//Almaceno datos en memoria
SW R1,0{R0}
SW R2,4{R0}
SW R3,8{R0}
SW R4,12{R0}
SW R5,16{R0}
SW R6,20{R0}
SW R7,24{R0}
SW R8,28{R0}
SW R9,32{R0}
SW R10,36{R0}
SW R11,40{R0}
SW R12,44{R0}
SW R13,48{R0}
SW R14,52{R0}
SW R15,56{R0}
SW R16,60{R0}
SW R17,64{R0}
SW R18,68{R0}
SW R19,72{R0}
SW R20,76{R0}
SW R21,80{R0}
SW R22,84{R0}
SW R23,88{R0}
SW R24,92{R0}
SW R25,96{R0}
HLT
```

7. Caso de test 7: prueba general de instrucciones tipo load y store.

Se prueban todas las instrucciones de tipo load y store. Todos los resultados que se obtienen son guardados en diferentes posiciones de memoria con el fin de facilitar el chequeo de los valores obtenidos. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la séptima hoja.

Archivo de assembler utilizado:

```
#A 10
#B 1024
#C 20024
#D 0
#E 1
#F 255
#G -1
//Cargo en R1 un 10
ADDI R1,R1,A
//Cargo en R2 un 20024
ADDI R2,R2,C
//Cargo en R3 un 1024
ADDI R3,R3,B
//Cargo en R4 un -1
ADDI R4,R4,G
//Almaceno datos en memoria
SW R1,0{R0}
SW R4,12{R0}
SH R2,6{R0}
SH R4,16{R0}
SB R3,8{R0}
SB R1,9{R0}
//Cargo datos en registros
LW R5,8{R0}
LWU R6,12{R0}
LH R7,12{R0}
LHU R8,14{R0}
LB R9,0{R0}
LB R10,9{R0}
LB R11,17{R0}
LBU R12,17{R0}
LUI R13,F
HLT
```

8. Caso de test 8: prueba general de instrucciones de salto.

Se prueban todas las instrucciones de salto. Todos los valores esperados para dicho caso de test se encuentran en la planilla de cálculo compartida anteriormente, en la octava hoja.

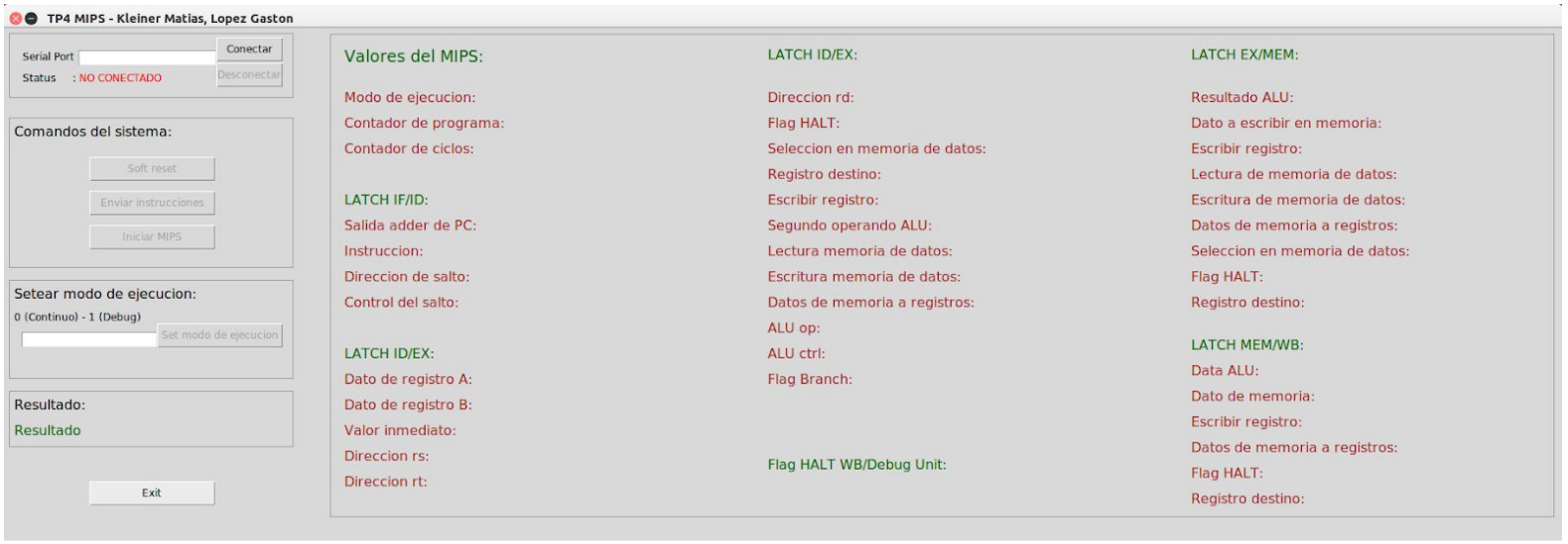
Archivo de assembler utilizado:

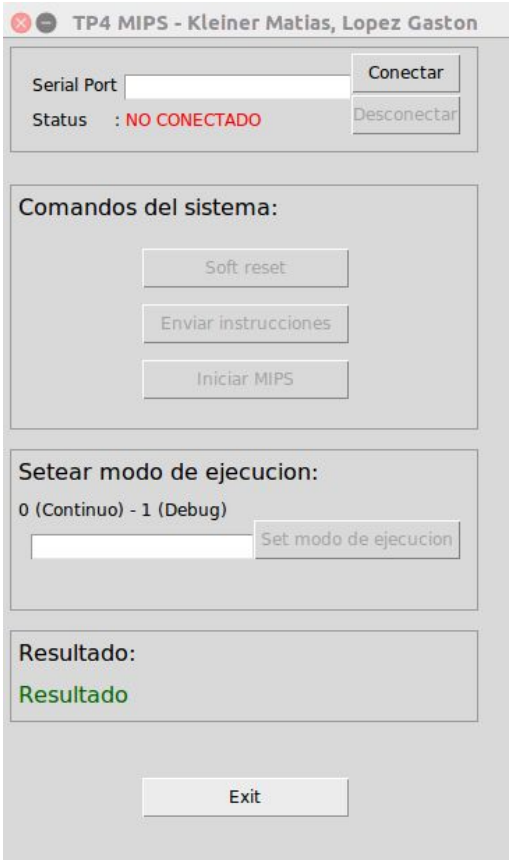
```
#A 10
#B 1024
#C 20024
#D 0
#E 1
#F 2
#G -1
//Cargo en R0 un 10
ADDI R0,R0,A
//Cargo en R1 un 2
ADDI R1,R1,F
//Cargo en R2 un 1
ADDI R2,R2,E
//Cargo en R3 un 21
ADDI R3,R3,21
//Decremento contador
SUBU R1,R1,R2
BEQ R1,R5,2
BNE R1,R5,-3
AND R1,R1,R1
JR R0
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
JALR R6,R3
JAL 27
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
AND R1,R1,R1
JALR R4,R6
AND R1,R1,R1
HLT
```

Interfaz en la PC.

Se generó una GUI en Python con la librería Tkinter como se muestra en la Figura 8.

1) **Conectarse/Desconectarse:** se brinda la opción de conectarse a la placa por medio del campo “Serial Port”, como así también la opción para desconectarse de la misma.
2) **Indicaciones para el MIPS:** el campo “Comandos del sistema” es gobernado por una máquina de estados interna de la GUI, donde permite realizar el Soft Reset de la placa, efectuar el envío de las instrucciones a la misma y también iniciar la ejecución de las instrucciones previamente cargadas.
3) **Alternar modo de ejecución:** en el campo “Setear modo de ejecución” se posibilita la elección entre modo continuo o modo debug (modo paso a paso). Siempre y cuando la instrucción HALT no llegue a la última etapa, se puede alternar entre alguno de los dos modos en cualquier momento.
4) **Valores obtenidos del MIPS:** en esta sección se arrojan los resultados obtenidos de los diferentes registros y etapas implementados en el MIPS.
5) **Indicaciones varias:** el apartado “Resultado” brinda información como, por ejemplo, detalles de la operación efectuada, muestra de errores, informacion de escritura en archivos, etc.
6) **Abandonar la GUI:** se brinda además una opción para salir de la interfaz gráfica en el botón “Exit”.


TP4 MIPS - Kleiner Matias, Lopez Gaston
Serial Port
Conectar
Status : NO CONECTADO
Desconectar
Comandos del sistema:
Soft reset
Enviar instrucciones
Iniciar MIPS
Setear modo de ejecucion:
0 (Continuo) - 1 (Debug)
Set modo de ejecucion
Resultado:
Resultado
Exit
Valores del MIPS:
Modo de ejecucion:
Contador de programa:
Contador de ciclos:
LATCH IF/ID:
Salida adder de PC:
Instruccion:
Direccion de salto:
Control del salto:
LATCH ID/EX:
Dato de registro A:
Dato de registro B:
Valor inmediato:
Direccion rs:
Direccion rt:
LATCH ID/EX:
Direccion rd:
Flag HALT:
Seleccion en memoria de datos:
Registro destino:
Escribir registro:
Segundo operando ALU:
Lectura memoria de datos:
Escritura memoria de datos:
Datos de memoria a registros:
ALU op:
ALU ctrl:
Flag Branch:
Flag HALT WB/Debug Unit:
LATCH EX/MEM:
Resultado ALU:
Dato a escribir en memoria:
Escribir registro:
Lectura de memoria de datos:
Escritura de memoria de datos:
Datos de memoria a registros:
Seleccion en memoria de datos:
Flag HALT:
Registro destino:
LATCH MEM/WB:
Data ALU:
Dato de memoria:
Escribir registro:
Datos de memoria a registros:
Flag HALT:
Registro destino:


TP4 MIPS - Kleiner Matias, Lopez Gaston
Serial Port
Conectar
Status : NO CONECTADO
Desconectar
Comandos del sistema:
Soft reset
Enviar instrucciones
Iniciar MIPS
Setear modo de ejecucion:
0 (Continuo) - 1 (Debug)
Set modo de ejecucion
Resultado:
Resultado
Exit

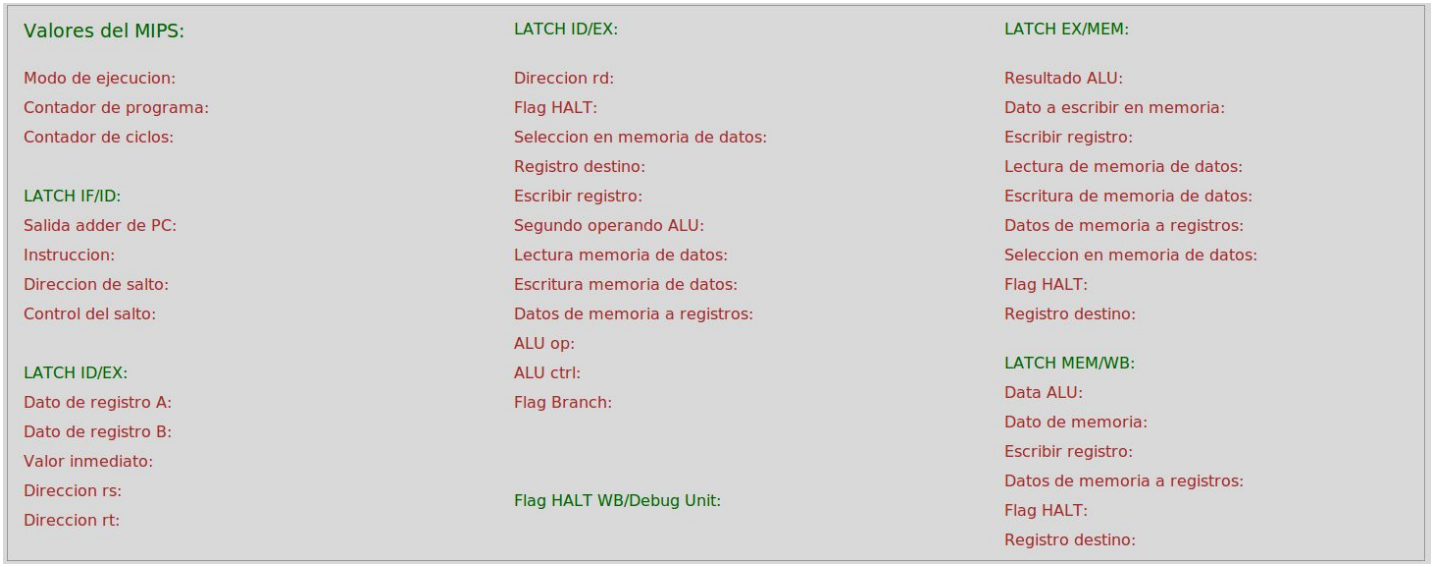


Figura 13 - Imágenes de GUI efectuada en Python.

Archivos a tener en cuenta

A fines de cumplir el requerimiento dado por la consigna en donde se pide traer todos los datos que se escribieron en la memoria de datos, la GUI almacena los mismos en un archivo denominado "*datamem.txt*". Dicho archivo de texto tiene 2 columnas, la primera indica el dato en cuestión y la segunda la dirección en donde se encuentra ese dato en la memoria. Ambas columnas están dadas en formato hexadecimal.

De forma similar, existe un archivo denominado "datareg.txt" en donde la GUI almacena el contenido de los 32 registros del MIPS.

Conclusión.

Una vez terminado el práctico, a través de la herramienta Vivado se obtuvieron las especificaciones en la utilización de los recursos de la FPGA y en la potencia que consume dicho circuito instanciado. Además, en este práctico se afianzaron los conceptos en cuanto al pipeline del procesador MIPS, visto durante las clases teóricas de la materia.

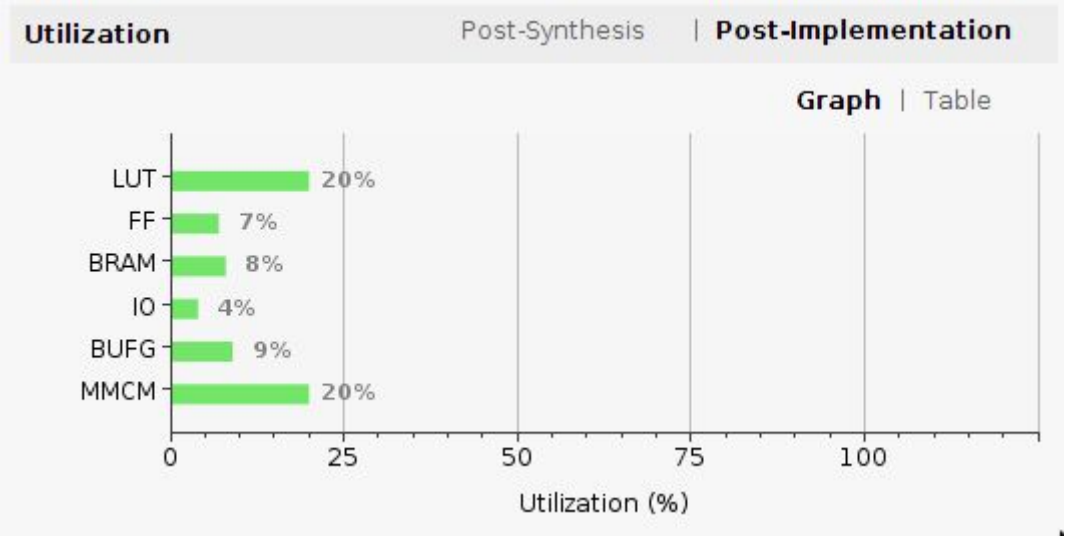


Figura 14 - Recursos utilizados para el desarrollo del proyecto.

Figura 15 - Potencia.

Bibliografía

- **Instrucciones:**
  - MIPS IV Instruction Set
- **Pipeline:**
  - Computer Organization and Design 3rd Edition. Chapter 6. Hennessy - Patterson.
  - Filminas de clases teóricas y prácticas de la materia.