Arquitectura de Computadoras 2022

TP Final: Pipeline procesador MIPS simplificado

Integrantes:

Gaston Emilio Sartori - 42337350 - gaston.sartori@mi.unc.edu.ar

Julian Tantera - 41994377 - julian.tantera@mi.unc.edu.ar

Introducción

En el presente informe se detalla la implementación del pipeline de 5 etapas del procesador MIPS simplificado, que corresponde al trabajo final de la materia de Arquitectura de Computadoras.

Consigna

Implementar el pipeline del procesador MIPS.

Requerimientos

1. Implementar el Procesador MIPS Segmentado en las siguientes etapas

- **IF** (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
- **ID** (Instruction Decode): Decodificación de la instrucción y lectura de registros.
- **EX** (Execute): Ejecución de la instrucción.
- **MEM** (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
- **WB** (Write back): Escritura de resultados en los registros.

2. Instrucciones a implementar

- **R-type:**
  - SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT
- **I-Type:**
  - LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL
- **J-Type:**
  - JR, JALR

3. Riesgos

El procesador debe tener soporte para los siguientes tipos:

- **Estructurales:** Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
- **De datos:** Se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
- **De control:** Intentar tomar una decisión sobre una condición todavía no evaluada.

4. Unidades de Riesgos

Para dar soporte a los riesgos nombrados se debe implementar:

- **Unidad de Cortocircuitos**

- **Unidad de Detección de Riesgos**

5. Otros Requerimientos

- El programa a ejecutar debe ser cargado en la memoria del programa mediante un archivo ensamblado.
    - Debe implementarse un programa ensamblador que convierte código assembler de MIPS a código de instrucción.
    - Debe transmitirse ese programa mediante interfaz UART antes de comenzar a ejecutar.

- Se debe simular una unidad de Debug que envíe información hacia y desde el procesador mediante UART.

6. Debug unit

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

- Contenido de los 32 registros
- PC
- Contenido de la memoria de datos usada

7. Modos de operación

Antes de estar disponible para ejecutar, el procesador está a la espera para recibir un programa mediante la Debug Unit.

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

- **Continuo:** se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo (Instrucción HALT). Llegado ese punto se muestran todos los valores indicados en pantalla.

- **Paso a paso:** Enviando un comando por la UART se ejecuta un ciclo de Clock. Se debe mostrar a cada paso los valores indicados.

Desarrollo

Para la implementación del procesador, realizamos una modularización de cada etapa del pipeline y de la debug unit. Cada una de ellas cuenta con registros, entradas y salidas que lo componen. En la siguiente imagen, se puede observar un diagrama con las 5 etapas del pipeline que conforman el procesador, los registros de segmentación y también las unidades de control, cortocircuito y debug.

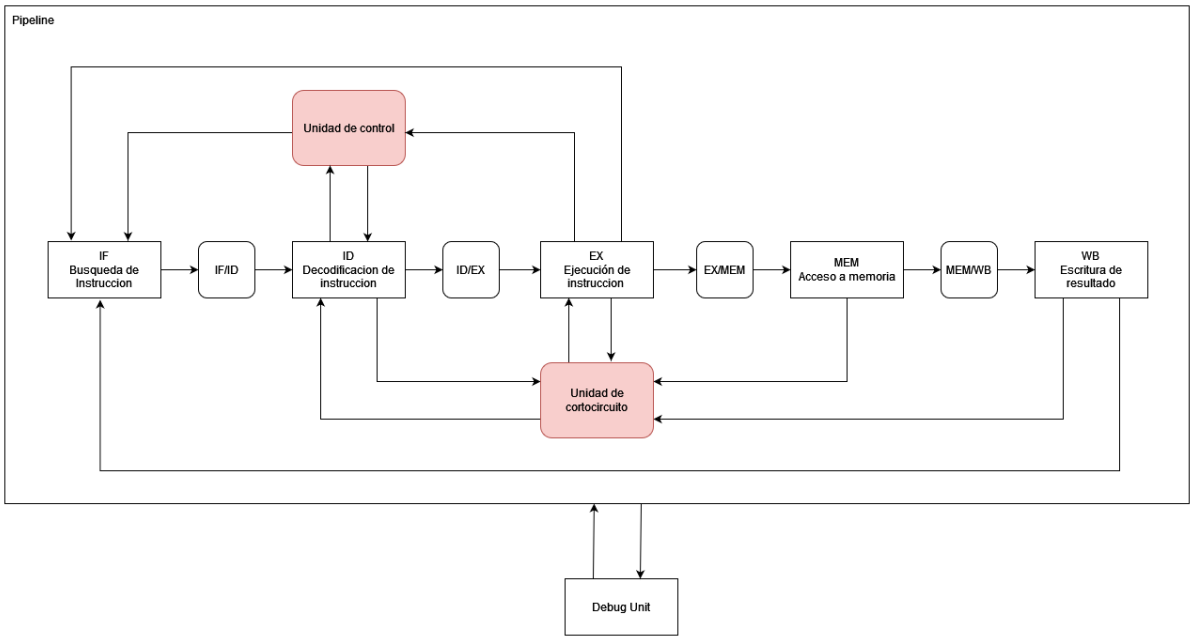

Figura 1: Diagrama general de modulos

Módulo IF

Este módulo realiza la tarea de Búsqueda de instrucción en la memoria de programa (Instruction Fetch). Los 4 componentes que lo integran son:

1. Multiplexor: es el encargado de seleccionar las posibles entradas del Program Counter.
2. Program Counter: representa el contador de programa y es la entrada de la memoria de instrucciones, contiene la dirección que se buscará en la misma.
3. Memoria de Instrucciones: se utiliza para almacenar las instrucciones del programa.
4. Sumador: es utilizado para actualizar el valor del Program Counter, obteniendo así la dirección de la próxima instrucción.

En el siguiente diagrama, se pueden observar los diferentes inputs y outputs de la etapa, obteniendo finalmente como salida del módulo la instrucción de 32 bits obtenida en memoria.

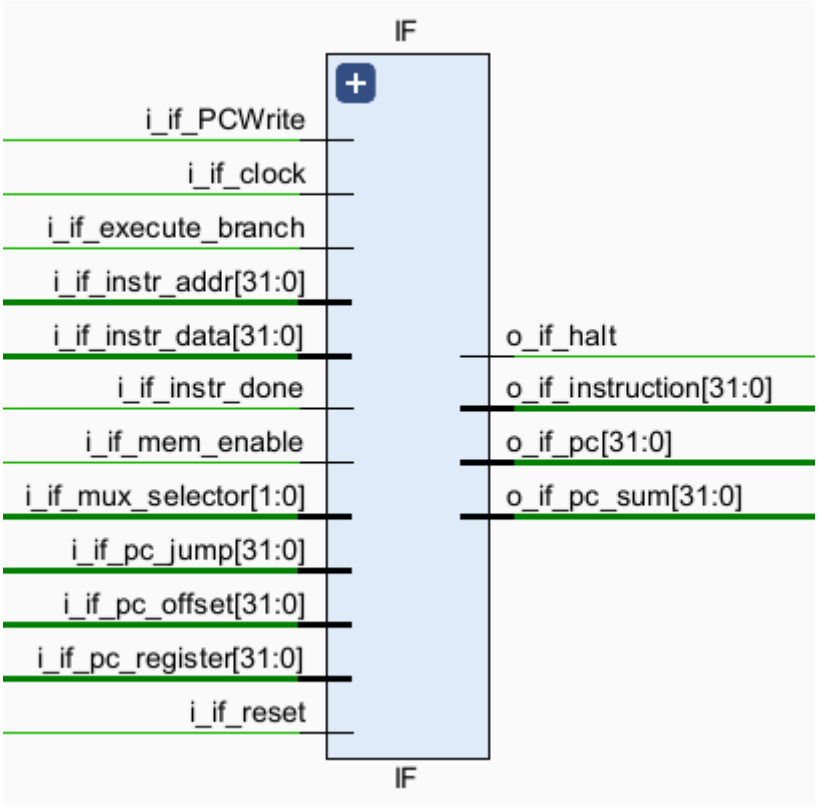

Figura 2: Entradas y salidas de la etapa IF

Módulo IF/ID

Este módulo contiene los registros de segmentación, cuya finalidad es mantener las salidas de la etapa de Búsqueda de instrucción, almacenando en registros el Program Counter incrementado, como también la instrucción obtenida en memoria, que luego serán utilizadas en la etapa de decodificación de instrucción.

En la figura 3 se pueden observar las entradas y salidas de este módulo.

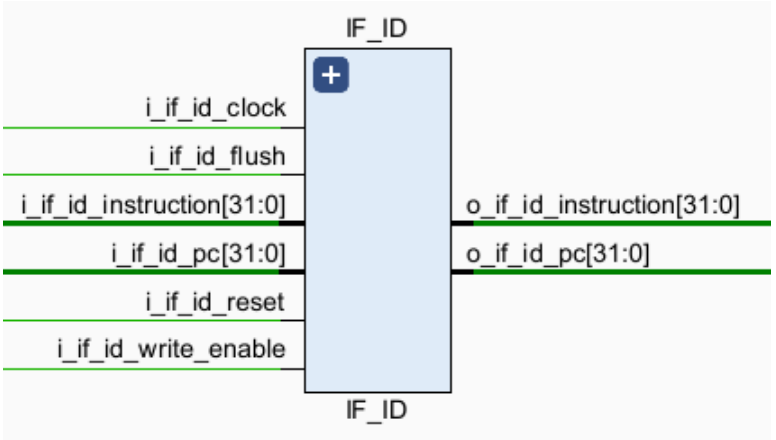

Figura 3: Entradas y salidas de la etapa IF/ID

Módulo ID

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

- R-Type: aritmetico-logicas.
- I-Type: salto condicional.
- J-Type: salto incondicional.

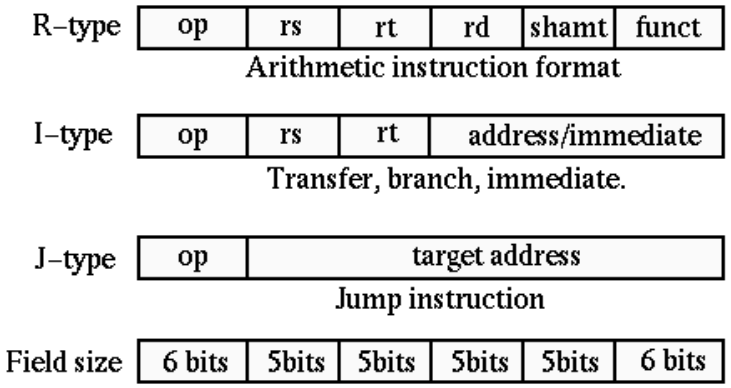

Figura 4: Instrucciones y sus tipos

Cuando la instrucción es decodificada, el procesador recibe desde la unidad de control las señales que determinan la operación a realizar. Para esto, contaremos con los siguientes submódulos.

- Banco de registros: en este se podrá leer y escribir, teniendo en cuenta diferentes señales provenientes de la Stall Unit y la Unidad de control.
- Extensión de Signo: extenderá el campo inmediato de las instrucciones de tipo I, a un tamaño de 32 bits, para luego poder operar con el Program Counter.
- Unidad de Control: será quien se encargue de decodificar las instrucciones, generando las señales de control de las etapas EX, MEM, WB. Estas son:
    - PcSrc: indica que entrada del MUX será el nuevo PC.
    - RegDst: indica cuál será el registro de destino, rt o rd.
    - ALUsrc: esta señal consta de 2 bits que nos indican cuáles serán las entradas de la ALU, ya sean rs/rt o rt/offset.
    - ALUOp: selecciona el tipo de instrucción que luego ingresará al control de la ALU.
    - MemRead: habilita la lectura de memoria.
    - MemWrite: habilita la escritura de memoria.
    - Branch: indica si la instrucción es un branch.
    - RegWrite: habilita la escritura en el banco de registros.
    - MemtoReg: indica la fuente en la escritura de registro, ya sea ALU, memoria o dirección de retorno.

Para determinar cuales señales generar para cada instruccion, se implemento una tabla de verdad. La misma podra encontrarse en el repositorio del proyecto.

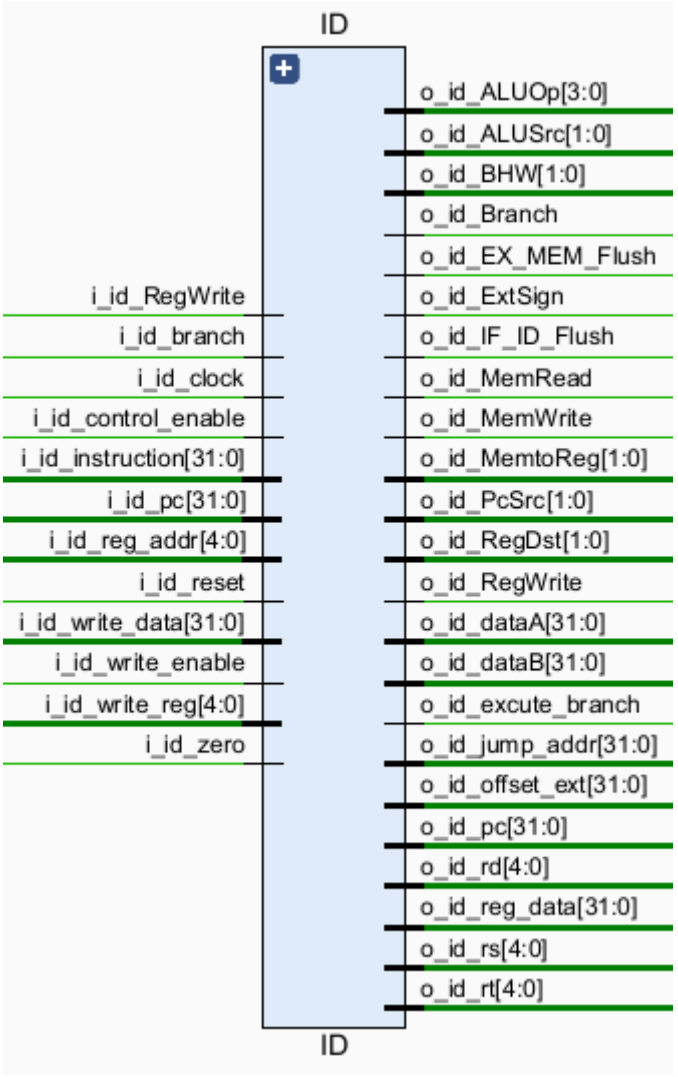

Figura 5: Entradas y salidas de la etapa ID

Funcionamiento de la unidad de control: este submódulo recibe de la instrucción a decodificar el campo OP y function (se utiliza solo en caso de ser necesario por las instrucciones tipo R y tipo J).
En primer lugar se analiza el campo OP para determinar qué tipo de operación es. En caso de que este sea 000000, se deberá analizar el campo funct, a través del cual se determina de qué operación se trata, y a partir de esto se generan las señales de control correspondientes para ejecutar esa instrucción. Para las tipo I, cada una cuenta con un opcode diferente. Así, en cada caso y para cada instrucción, se setean las distintas señales de control necesarias para el correcto funcionamiento del procesador.

Módulo ID/EX

Al igual que el módulo IF/ID, se almacenan en este los registros de segmentación de las etapas ID/EX, los cuales tienen la tarea de mantener las salidas de ID. Estas son las diferentes señales de control correspondientes a las etapas siguientes, Program Counter,

los datos A y B provenientes del banco de registros, el offset con signo extendido y los campos rs, rt y rd.

En la figura 6 se observan las entradas y salidas de este módulo.

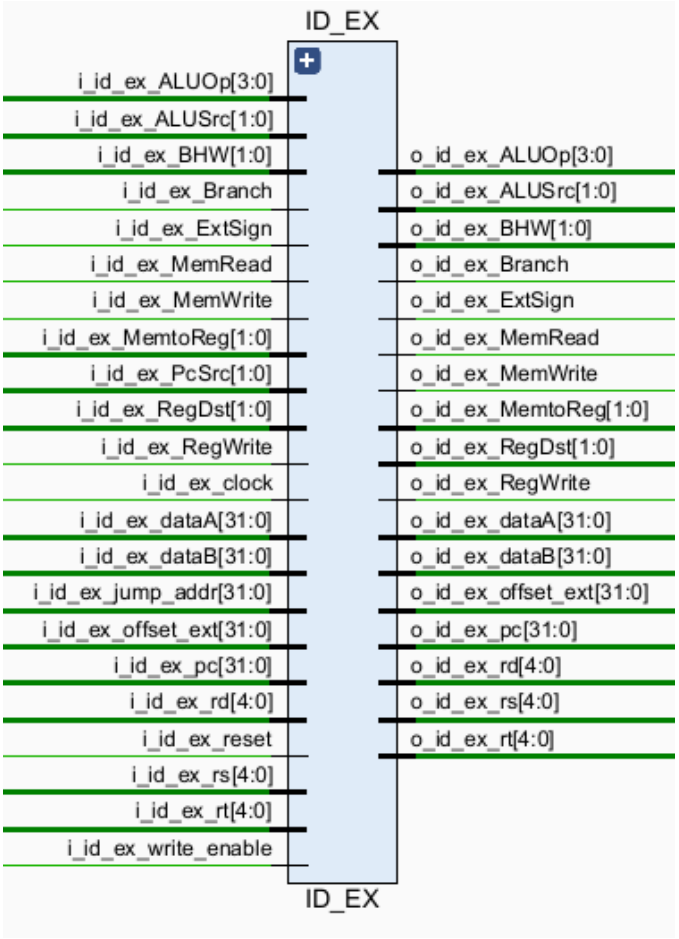

Figura 6: Entradas y salidas de la etapa ID/EX

Módulo EX

En este módulo, se lleva a cabo la ejecución de la instrucción. Una vez decodificada la instrucción en la etapa previa, se obtienen los operandos (rt, rd de la instrucción), como también el código de operación y los datos A y B.

Submodulos que componen la etapa EX:

- Sumador: realiza el incremento del program counter en una cantidad indicada por el offset, para instrucciones de salto condicional o branch.
- Multiplexores: tienen el objetivo de seleccionar las entradas de la ALU, como también determinar el registro de destino, a partir de diferentes señales de control.
- Control_ALU: se encarga de generar la señal de control que especifica la operación a realizar por la ALU.
- ALU: Unidad Aritmética Lógica combinacional que realiza la operación indicada por el ALU code, a partir de los dos operandos que se encuentran en su entrada.

A partir de esto, la lógica de los multiplexores y teniendo en cuenta la instrucción y la unidad de cortocircuitos, la ALU obtendrá dos operandos que serán utilizados en la operación combinacional indicada por la unidad de control de la ALU, arrojando el resultado a la salida de la misma, como se observa en uno de los outputs de la siguiente imagen (figura 7).

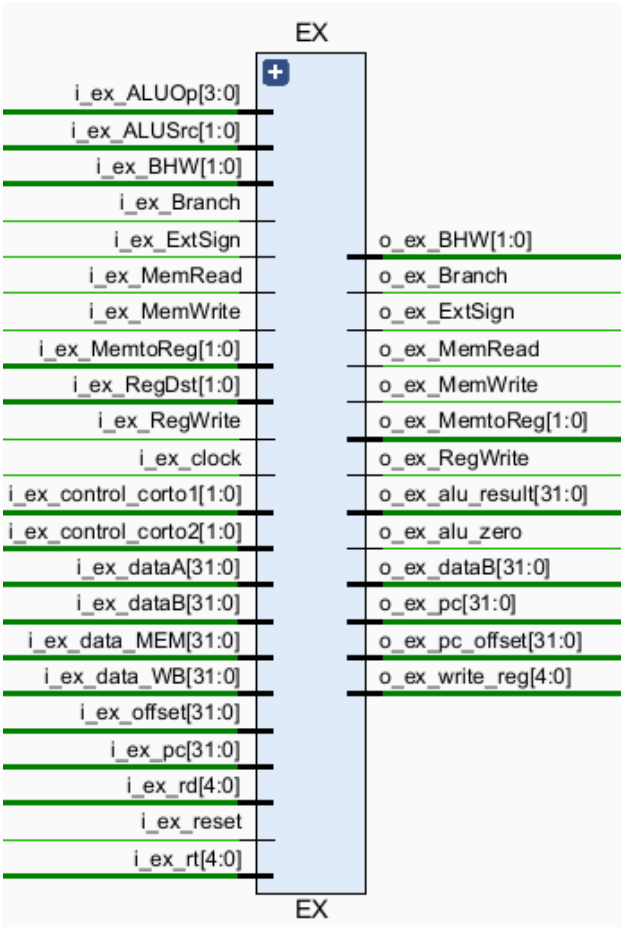

Figura 7: Entradas y salidas de la etapa EX

Módulo EX/MEM

En este módulo, se almacenan los registros de segmentación de las etapas EX/MEM, manteniendo las salidas de la etapa EX. Además, nos encontramos con una señal de flush, que será controlada por la Unidad de Control y seteada en caso de tener una instrucción de branch, donde se pondrán en 0 las salidas correspondientes al módulo.

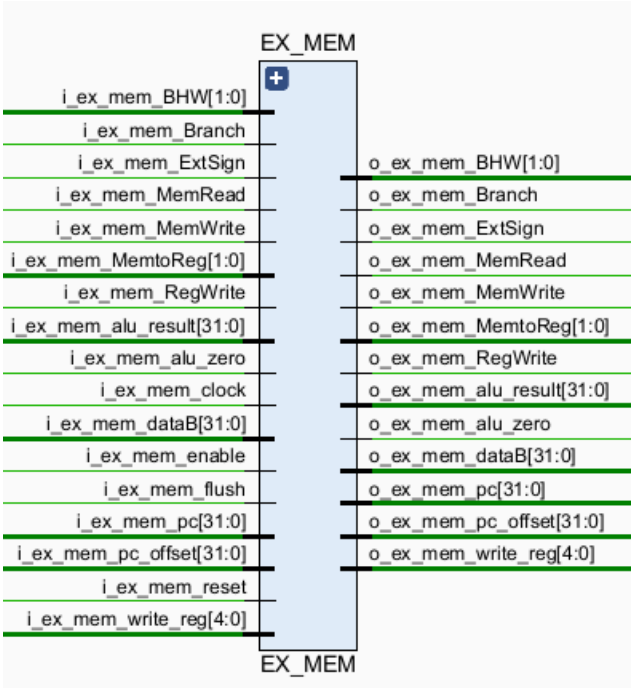

Figura 8: Entradas y salidas de la etapa EX/MEM

Módulo MEM

En la etapa número 4 del MIPS, se realiza un acceso a memoria de datos indicado por la actual instrucción, de ser requerido. Este acceso puede ser para escritura o lectura de memoria, y para esto se tendrán en cuenta diferentes señales de control, indicadas en el diagrama de entradas y salidas.
Este módulo está integrado por un submódulo denominado Memoria_datos, el cual indica la forma en que está organizada la memoria y la manera en que será escrita o leída por las instrucciones, ya sea un tamaño de byte (8 bits), half word (16 bits) o word (32 bits).

Las entradas y salidas se pueden ver en la siguiente Figura (figura 9).

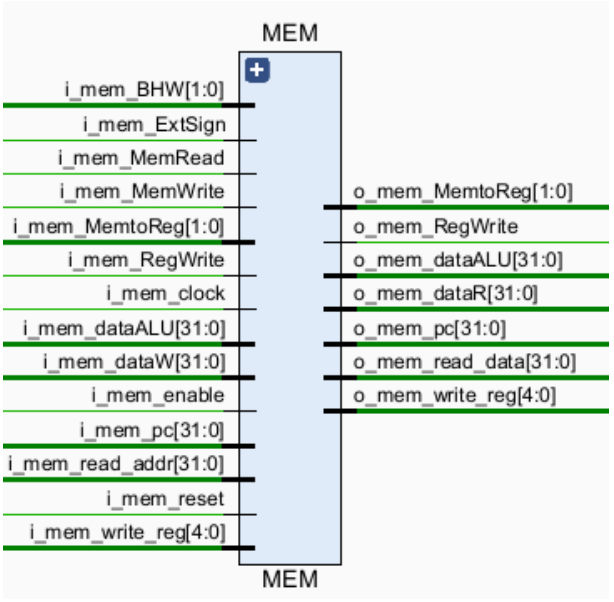

Figura 9: Entradas y salidas de la etapa MEM

Módulo MEM/WB

Este módulo está integrado por los registros de segmentación de las etapas MEM/WB, en los cuales se mantendrán las salidas de la etapa de acceso a memoria que luego ingresarán a la etapa de WB y cuenta con los siguientes inputs/outputs (Figura 10).

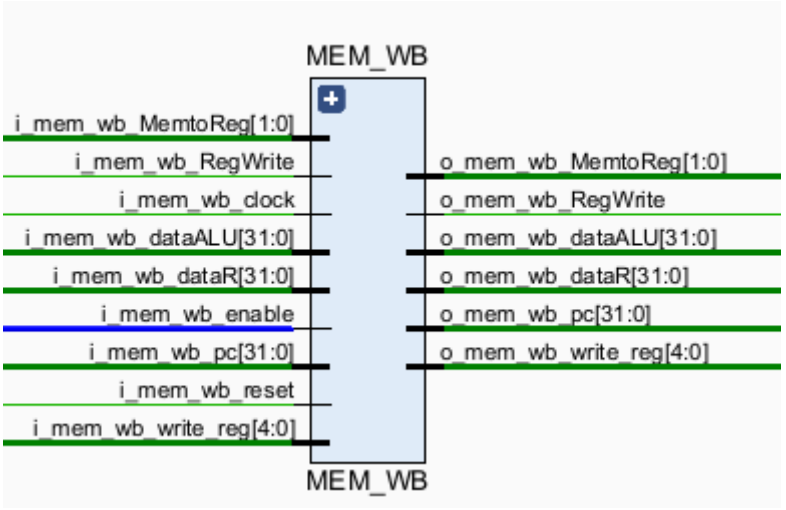

Figura 10: Entradas y salidas de la etapa MEM/WB

Módulo WB

En esta última etapa del pipeline, se realiza una actualización de registros, es decir, se escriben los resultados de las instrucciones correspondientes en el archivo de registro.

El módulo WB está compuesto por un submódulo multiplexor controlado por una señal seteada por la Unidad de Control, que especifica cual es la fuente al escribir los registros. Las entradas de este multiplexor corresponden a si el dato proviene de memoria de datos (instrucción store), al resultado de la ALU (para el caso de instrucciones aritméticas) o dirección de retorno.

En la figura 11 se pueden observar las entradas y salidas de este módulo.

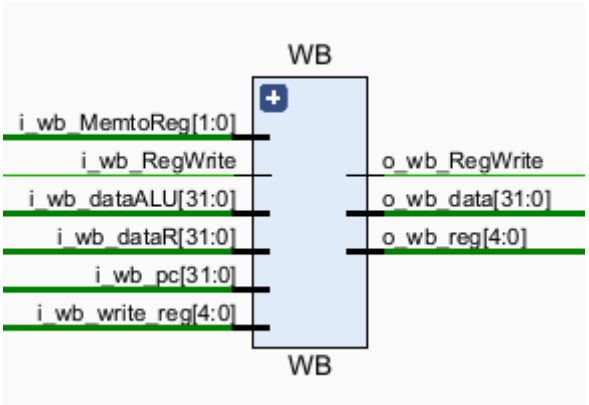

Figura 11: Entradas y salidas de la etapa WB

Módulo Unidad de detección de riesgos

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

Cuando el riesgo es detectado y se introduce el retardo, el valor leído de la memoria de datos se multiplica hacia el operando de la ALU, con el objetivo de que la instrucción que necesita dicho dato pueda realizar la operación.

En la siguiente figura se pueden observar las entradas y salidas de la unidad de detección de riesgos.

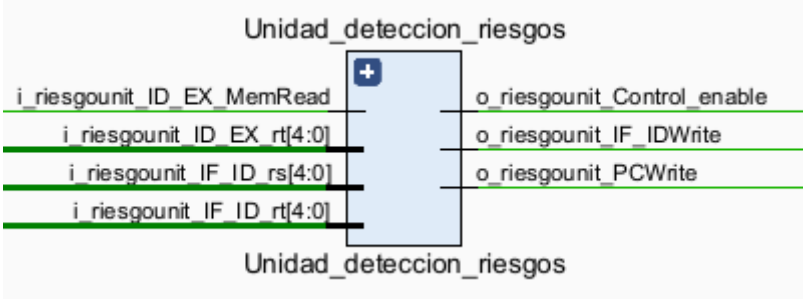

Figura 12: Entradas y salidas de la Unidad de detección de riesgos

Módulo Unidad de cortocircuito

La unidad de cortocircuito es quien se encarga de chequear la dependencia de datos entre instrucciones, verificando los registros que se utilizan en estas. Se utilizan multiplexores que serán controlados por diferentes señales que nos indicarán si el dato necesita ser cortocircuitado, de esta manera el módulo EX obtiene el dato actualizado.

A continuación, se puede observar en el diagrama, las entradas y salidas del módulo, que luego serán utilizadas en la etapa EX como se mencionó anteriormente.

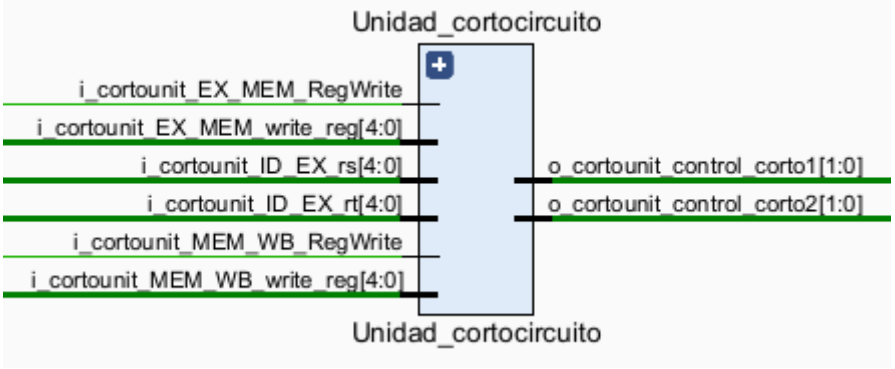

Figura 13: Entradas y salidas de la Unidad de Cortocircuito

Módulo Debug Unit

Este módulo permite la comunicación del usuario con el procesador. Recibe desde el módulo UART comandos para ejecutar determinadas acciones. Los comandos implementados son los siguientes:

- Código 0x01: setea el modo de **carga de instrucciones**. Posterior a este comando se envía el archivo binario de instrucciones las cuales se cargan en la memoria. Recibe hasta que se cargue una instrucción de HALT.
- Código 0x02: setea el modo de **ejecución continua**. Se ejecutan instrucciones hasta llegar a una instrucción de HALT.
- Código 0x04: setea el modo **paso a paso**. Se ejecuta el avance de un ciclo del pipeline.
- Código 0x08: retorna el valor del **program counter**.
- Código 0x01: retorna el valor de los **32 registros**.
- Código 0x02: retorna el valor de la **memoria de datos**.

Modo de carga de instrucciones

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

Modo de ejecución continua y modo paso a paso

La implementación del modo de ejecución continua y del modo paso a paso se realizó de la siguiente manera. Desde la debug unit, se envía al pipeline una señal habilitadora (o_debugunit_enable en el diagrama), la cual es la encargada de permitir o no el avance de un ciclo del mismo. Solo cuando esta está activa, al darse un ciclo de clock se actualizan los valores del PC y de los registros de segmentación, caso contrario los mismos conservan su valor. De esta manera, es que puede conservarse el estado del pipeline durante los ciclos de clock que se desee.
En el caso del modo de ejecución continua, esta señal permanece activa hasta que se direccione en la memoria una instrucción de HALT. Cuando esta condición se de, se recibe una señal (i_debugunit_halt en el diagrama) y permite que la debug unit salga del estado de ejecución continua y desactiva la señal habilitadora.
En el caso del modo paso a paso, la señal habilitadora solo se activa durante un ciclo de clock, permitiendo así el avance de un ciclo del pipeline.
Al direccionarse una instrucción HALT, la ejecucion se bloquea y no se permite el avance de la misma, en ninguno de los dos modos de ejecucion.

Modos de lectura de PC, registros y memoria

La lectura de PC, registros y memoria de datos, se implementó a través de conexiones directas con los módulos correspondientes. Estas conexiones permiten leer los datos deseados (i_degubunit_pc, i_degubunit_reg_data, i_degubunit_read_data) y posteriormente

enviarlos a través de UART al usuario. Para el caso de los registros y la memoria, fue necesario también una conexión de direccionamiento (o_degubunit_reg_addr, o_degubunit_read_addr), para determinar cuál valor se desea leer en ese momento.

Diagrama de estados

La implementación de la debug unit está basada en una máquina de estados, que cambia de estados según el comando recibido y realiza la acción correspondiente. A continuación se muestra un diagrama de la máquina de estados:

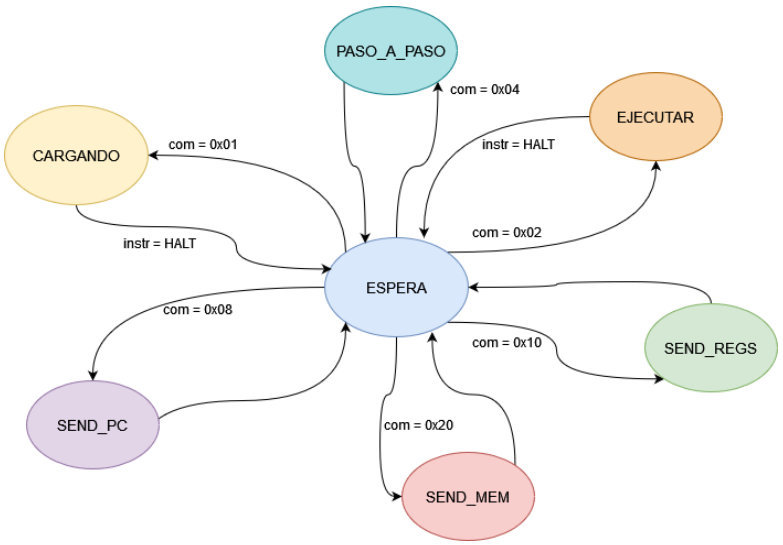

Figura 14: Diagrama de estados de la Debug Unit

Además, podemos observar en el siguiente diagrama, las entradas y salidas del módulo.

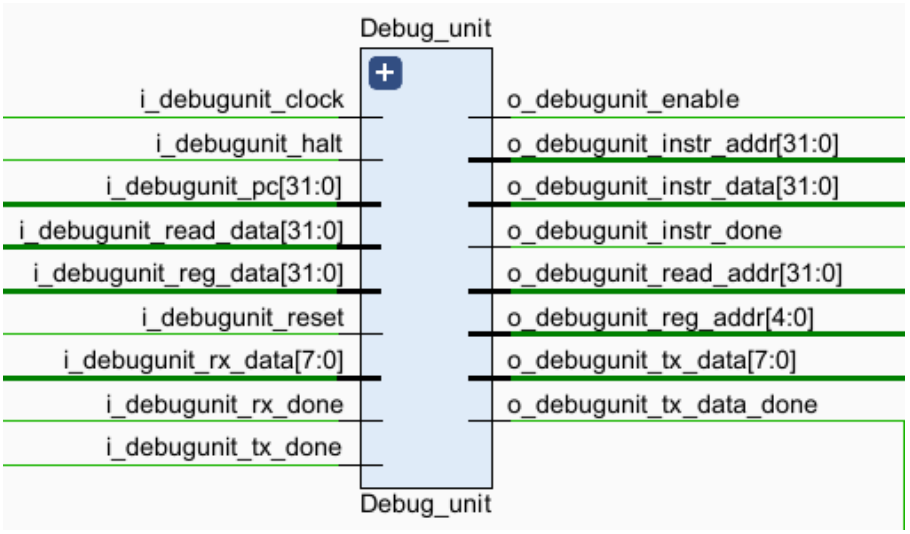

Figura 15: Entradas y salidas de la Debug Unit

Módulo Clock Wizard

Un punto importante de la implementación es determinar cuál es la frecuencia máxima a la cual puede operar el diseño. Para reducir la frecuencia del clock se implementó un módulo de Clock Wizard. El cual permite generar nuevas salidas de clock de diferentes frecuencias. En nuestro caso, utilizamos una salida de clock de 50MHz, y se analizaron los reportes de timing de Vivado:

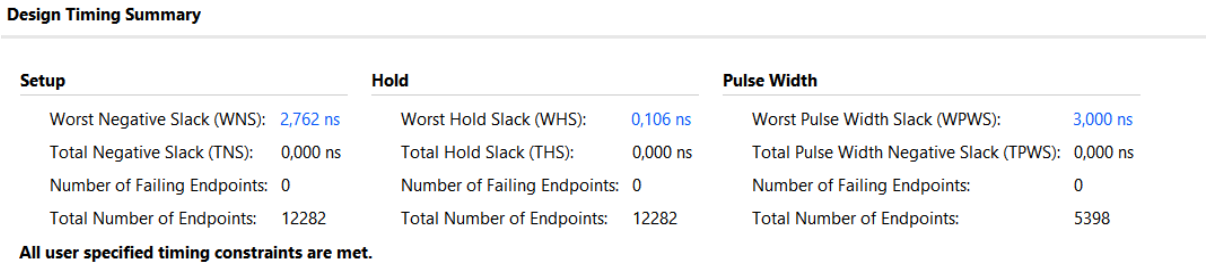

Figura 16: Reporte de timing.

Como podemos observar, con 50MHz, es decir con un periodo de clock de 20ns, el diseño podría funcionar perfectamente, ya que el Worst Negative Slack da un valor positivo. La máxima frecuencia a la que podría operar según este informe, es a 58MHz, lo que equivale a un periodo de clock de 17,23ns. Sin embargo, como estos resultados son obtenidos mediante simulación, decidimos darle un margen de frecuencia para estar seguros del buen funcionamiento del diseño, por lo que se mantuvo el clock de 50MHz.

Conversion assembler a formato binario

Con el objetivo de poder probar la implementación realizada, fue necesario poder convertir instrucciones assembler a su equivalente en formato binario. Para lo cual se desarrollo un programa Python, el cual cumple con esta funcionalidad. Dado un archivo .asm, para cada instruccion en el mismo, se convierte la misma a su formato binario, y se genera un archivo .bin con las mismas. Este programa puede encontrarse en el repositorio del proyecto, llamado parser.py.

Simulaciones a traves de testbench

Para probar las funcionalidades de los modulos desarrollados, se implementaron diferentes testbenchs. Los mismos permitieron probar a cada modulo por separado y tambien la integracion entre los mismos, hasta poder comprobar el correcto funcionamiento de la totalidad del proyecto. Los testbenchs desarrollados se muestran a continuacion y pueden encontrarse en el repositorio del proyecto.

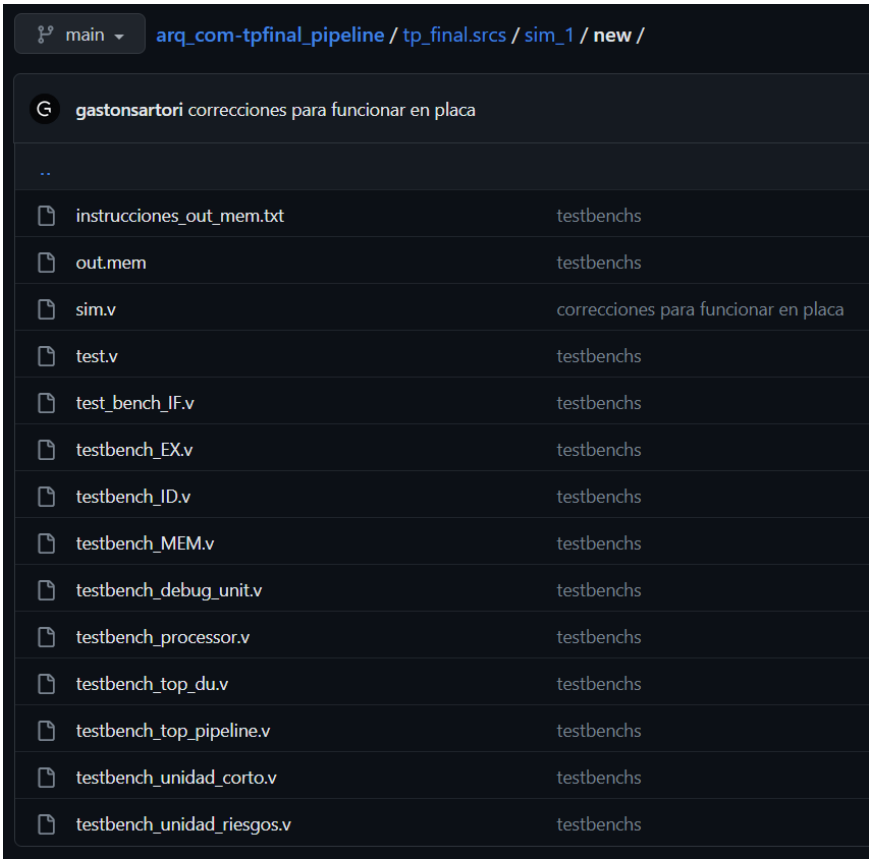

Figura 17: Testbenchs implementados.

Implementacion en NEXYS 4 DDR

La implementación en placa fisica se realizo en una fpga NEXYS 4 DDR. A continuacion se muestran imagenes de las pruebas realizadas en la misma. La misma se programo con el diseño realizado, fue conectada a una PC y se utilizo RealTerm para enviar y recibir informacion con la misma.

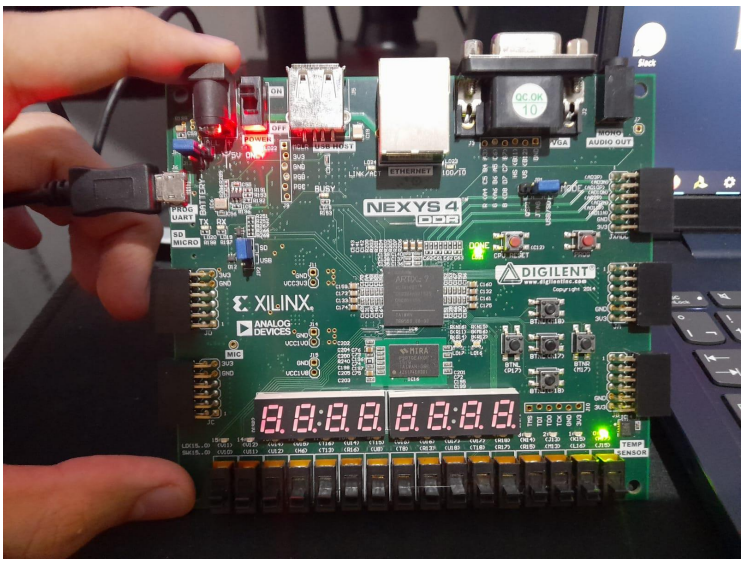

Figura 18: FPGA NEXYS 4 DDR.

Figura 19: FPGA NEXYS 4 DDR conectada a PC a traves de USB.

Luego de programar la misma, se cargo el codigo assembler que se muestra a continuacion, el cual previamente fue convertido a su equivalente en formato binario.

test_completo.txt

```
1   ADDI R4, R0, 7123       //R4 = 7123=1BD3h
2   ADDI R3, R0, 85         //R3 = 85=55 h
3   ADDU R5, R3, R4         //R5 = R4 + R3 = 7123 + 85 = 7208=1C28h - cortocircuito desde MEM y WB
4   SUBU R6, R4, R3         //R6 = R4 - R3 = 7123 - 85 = 7038=1B7Eh - cortocircuito desde WB
5   NOR  R10, R4, R3        //R10 = R4 NOR R3 = FFFFE428h
6   SLL  R12, R10, 2        //R12 = R10 <<L 2 = FFFF90A0h
7   SRL  R13, R10, 2        //R13 = R10 >>L 2 = 3FFFF90Ah
8   SRA  R14, R10, 2        //R14 = R10 >>A 2 = FFFFF90Ah
9   SB   R13, 4(R0)         //M4 = 0Ah - store por byte
10  SH   R13, 8(R0)         //M8 = F90Ah- store por halfword
11  SW   R13, 12(R0)        //M12 = 3FFFF90Ah - store por word
12  LB   R18, 12(R0)        //R18 = M12B = 0Ah - load por byte
13  ANDI R19, R18, 6230     //R19 = R18 & 6230 = 0010 = 2h- use dsp de load, stall
14  LH   R20, 12(R0)        //R20 = M12W = FFFF F90Ah - load por 2byte
15  ORI  R21, R20, 6230     //R21 = R20 | 6230 = FFFFF95Eh - use dsp de load, stall
16  LW   R22, 12(R0)        //R22 = M12 = 3FFFF90Ah - load por 4byte
17  XORI R23, R22, 6230     //R23 = R22 XOR 6230 = 3FFFE15Ch  - use dsp de load, stall
18  JAL     22              //Salta ADDI R7, R0, 15 PC=88=58h, R31=76=4Ch
19  ADDI    R2, R0, 95      //R2=95=5Fh
20  ADDI    R1, R0, 85      //no se ejecuta por el jump
21  NOP
22  NOP
23  ADDI    R7, R0, 255     //R7=255=FFh
24  ADDI    R8, R0, 15      //R8=15=Fh inicialmente, termina en 32=20h
25  ADDI    R9, R0, 32      //R9=32=20h
26  ADDI    R8, R8, 1       //incrementa R8
27  BNE R8, R9, -2          //salta a ADDI  R8,R8,1 cuando R8!=R9
28  SB  R7, 0(8)            //guarda FFh donde apunta R8 -> se guarda FFh desde M16 a M32
29  BEQ R8, R9, 2           //salta a ADDI  R27,R0,8 cuando R8==R9
30  ADDI    R25, R0, 8      //R25=8=8h
31  ADDI    R26, R0, 8      //no se ejecuta por el branch
32  ADDI    R27, R0, 8      //R27=8=8h
33  NOP
34  NOP
35  NOP
36  HALT
```

Figura 20: Codigo assembler de prueba.

Antes de comenzar la ejecución, se observaron los valores del PC, registros y memoria:

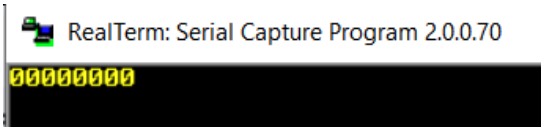

Figura 21: Valor de PC antes de la ejecución.

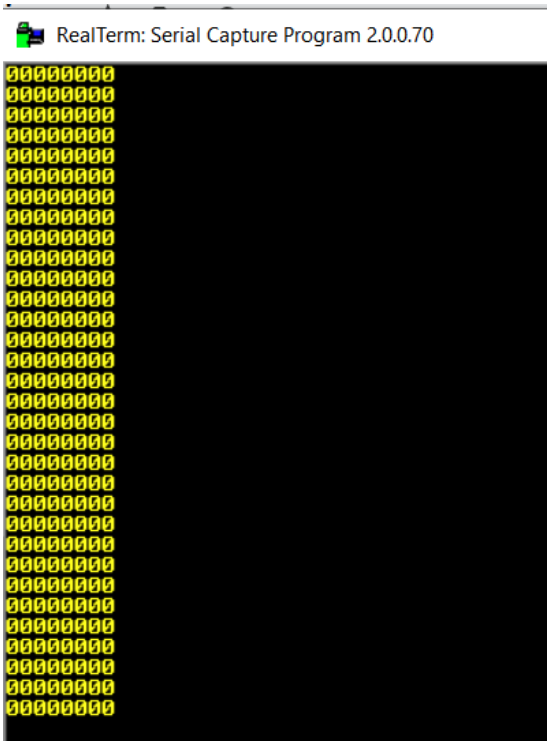

Figura 22: Valores de registros antes de la ejecución.

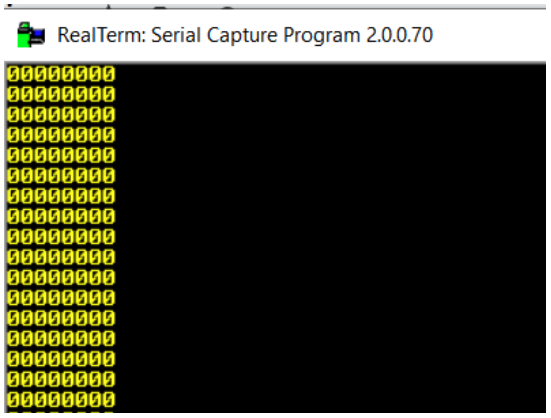

Figura 23: Valores de memoria antes de la ejecución.

Luego se ejecuto el codigo y al finalizar la ejecucion, los valores de PC, registros y memoria fueron los siguientes:

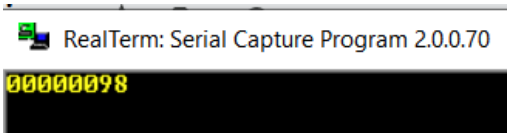

Figura 24: Valor de PC al finalizar la ejecución.

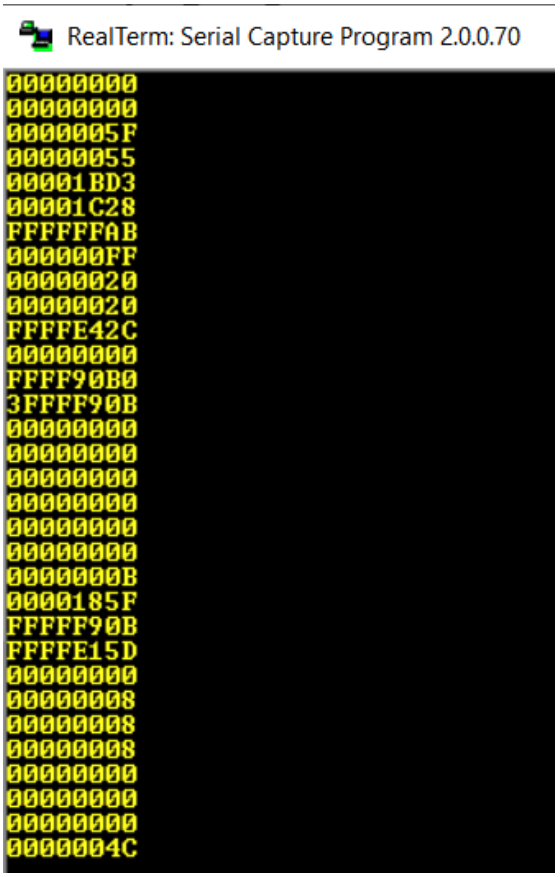

Figura 25: Valores de registros al finalizar la ejecución.

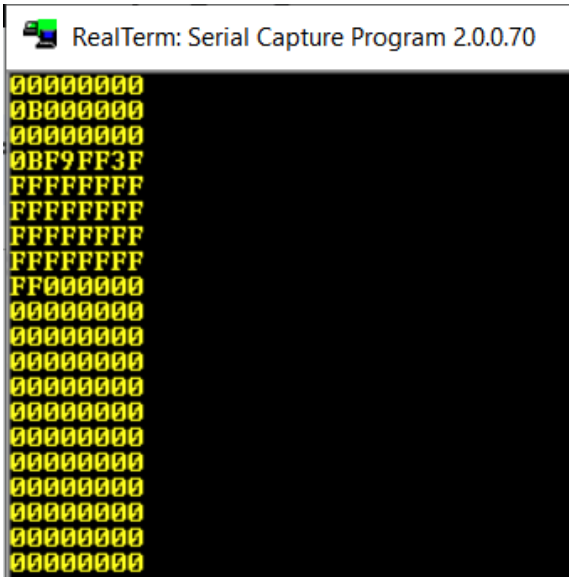

Figura 26: Valores de memoria al finalizar la ejecución.

Analizando los valores finales obtenidos, se pudo comprobar que los mismos fueron los esperados y la ejecucion se realizó correctamente.

Conclusiones

A lo largo de este trabajo, tuvimos la posibilidad de afianzar y ampliar nuestros conocimientos sobre el procesador MIPS y sus diferentes etapas. Con el objetivo de mantener un orden, consideramos que fue de gran importancia la acción de realizar un diagrama que simplifique el hecho de trabajar con numerosas variables, lo cual muchas veces fue de ayuda a la hora de pasarlo a código.
En el desarrollo, surgieron dudas y dificultades con respecto a los tiempos de clock a utilizar, lo cual pudimos solucionar al comprender el funcionamiento del clock wizard y el Worst Negative Slack.
También, fue de gran ayuda la herramienta de simulación de Vivado, ya que nos permitió mitigar errores que no sabíamos de donde provenían o que variables los estaban ocasionando, lo cual fue llevado a cabo testeando de un módulo a la vez para finalmente poder llevar la implementación al hardware.
Finalmente, luego de realizar una breve investigación sobre la placa de desarrollo Nexys 4, pudimos comprender la lógica de cada una de sus entradas y pulsadores, logrando que cada una de las funcionalidades desarrolladas previamente se puedan ejecutar sin problema alguno.

Repositorio

En el siguiente link se puede encontrar los códigos de todos los módulos implementados, un diagrama completo de la implementación y la tabla de verdad implementada para decodificación de las instrucciones:

Repositorio-TPFinal-Sartori-Tantera