Trabajo Práctico Especial: Microprocesador MIPS uniciclo

Grupo: 7

Integrantes: Avila Agustin
Lobo Ignacio

Materia: Arquitectura de Computadoras I

Profesores: Vázquez Martín
Toloza Juan
Leiva Lucas

Introducción

En el presente informe se va a detallar el desarrollo del Microprocesador MIPS uniciclo, así como las decisiones tomadas en su desarrollo, los conflictos atravesados y los resultados finales.
El proyecto tuvo como objetivo no solo terminar de comprender teóricamente el funcionamiento del Microprocesador MIPS, sino que también sumergirse en la comprensión práctica de su funcionamiento. Todo esto acompañado de los conceptos aprendidos durante la cursada y nuestros conocimientos de VHDL.
Para comenzar a desarrollar el trabajo partimos de los archivos facilitados por la cátedra:
Archivos VHDL

- Processor
- ProcessorTB
- ProgramMemory
- DataMemory

Archivos contenidos de memorias (Datos y Programa)

- data
- program1
- program1.s
- registers

Luego de revisar y analizar a detalle los archivos nombrados comenzamos con el desarrollo de nuestro propio código.

Desarrollo

Con respecto al trabajo grupal utilizamos distintas herramientas que facilitaron nuestra colaboración en paralelo:

-Para la comunicación utilizamos ***Discord***, lo cual nos permitió compartir nuestras pantallas y opinar sobre la realización del trabajo en tiempo real.

-Con respecto al código utilizamos ***GitHub***, ya que lo consideramos la manera más cómoda de tener un seguimiento sobre los cambios que se realizaban y llevar un control sobre los mismos.
Link del repositorio:
https://github.com/Nacho1810/unicycle_processor_Untref.git

-Por otro lado, para tener un recurso más visual, nos ayudó tener a disposición un gráfico con los nombres de las variables utilizadas en el código. Para ello utilizamos ***app diagrams***.
Link del diagrama:
https://viewer.diagrams.net/?tags=%7B%7D&lightbox=1&highlight=0000ff&edit=_blank&layers=1&nav=1&title=Microprocesador%20MIPS#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1uinx10jlawnzsKLl56Ucq-kUYX-Fp6JW%26export%3Ddownload
(En él se encuentra el diagrama del Microprocesador MIPS con los nombres de señales y puertos utilizados en nuestro código)

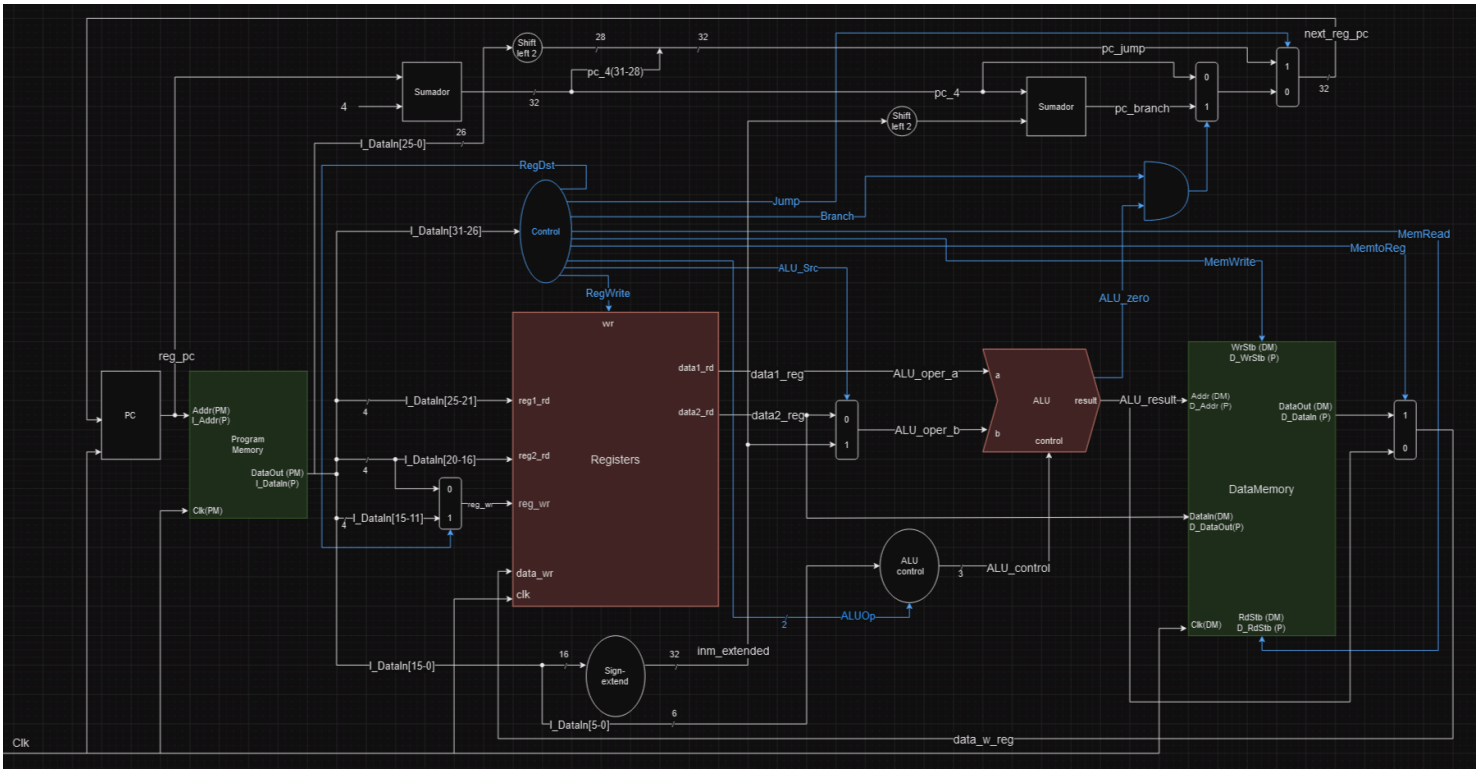
- Realizamos también la mayor parte del desarrollo en ***Visual Studio Code***, proveyéndonos de extensiones visuales y de formateo que aligeren el codeo . La mayor parte del trabajo fue desarrollado sincrónicamente y en contacto para ayudarnos a comprender mejor el código y sobreponernos a equivocaciones y errores de tipeos.

En un primer momento decidimos crear más entidades de las dadas por la cátedra y utilizarlas como componentes en nuestro procesador. Finalmente, decidimos que lo simple era lo más elegante y legible, terminamos utilizando solo las entidades que nos proporcionó la cátedra e implementamos los multiplexores y unidades lógicas por sentencias.

Resultados y pruebas

Gracias a la prolijidad del desarrollo del trabajo es que, a la hora final de probarlo en ***EDA Playground***, el compilador no detectó más que unos pocos errores y la simulación fue completamente exitosa.
Además, creamos un programa adicional, llamado ***program2***, buscando un testeo adicional con un orden distinto de instrucciones. Fue añadido en su forma hexadecimal y assembler.

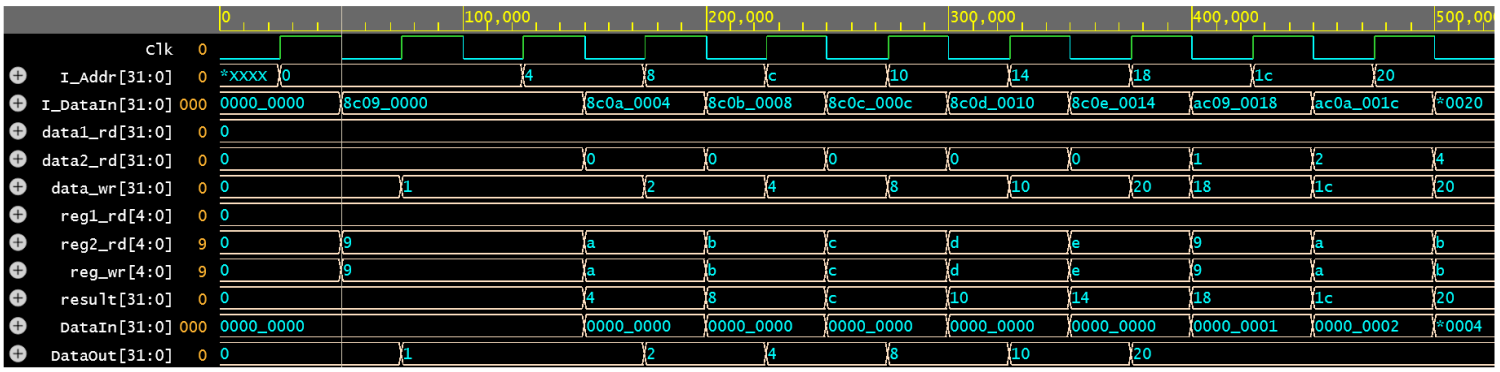
Les compartimos un diagrama que realizamos para nuestro mayor entendimiento del funcionamiento concreto de cada señal y puerto.



La siguiente secuencia es una simulación del programa, propiciado por la cátedra, ejecutado:
I_Addr[31:0] : puerto de salida del *Processor* a la dirección de memoria del *Program Memory*. Recibe la señal reg_pc.
I_DataIn[31:0] : puerto de entrada del *Processor* de la instrucción leída del *Program Memory*.
data1_rd[31:0] : puerto de salida del componente *Registers*. Ingresa el primer dato leído del banco de registro en la señal data1_reg.
data2_rd[31:0] : puerto de salida del componente *Registers*. Ingresa el segundo dato leído del banco de registro en la señal data2_reg.
data_wr[31:0] : puerto de entrada del componente *Registers*. Es posible que reciba el dato a escribir en el banco de registros por la señal data_w_reg.
reg1_rd[4:0] : puerto de entrada del componente *Registers*. Recibe la señal I_DataIn[25:21].
reg2_rd[4:0] : puerto de entrada del componente *Registers*. Recibe la señal I_DataIn[20:16].
reg_wr[4:0] : puerto de entrada del componente *Registers*. Recibe la señal I_DataIn[20:16] o I_DataIn[15:11].
result[31:0] : puerto de salida del componente *ALU*. Recibe la señal I_DataIn[20:16] o I_DataIn[15:11].
DataIn[31:0] : puerto de entrada del *Data Memory*. Es posible que reciba la señal data2_reg.
DataOut[31:0] : puerto de salida del *Data Memory*. Puede ingresar a la señal data_w_reg el dato leído del *Data Memory*.

From: 0ps To: 2,000,000ps

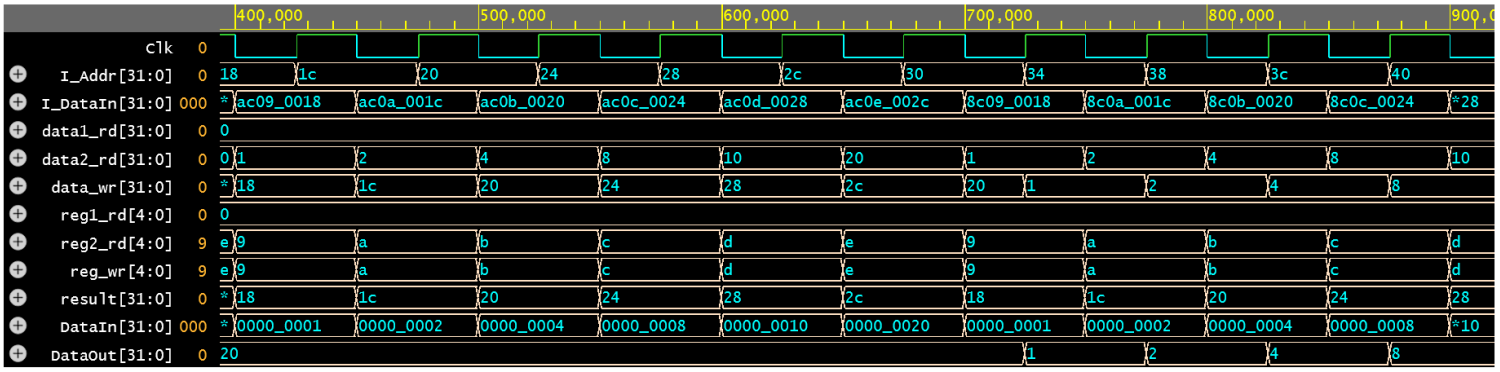
Get Signals | Radix | 100% | Δ 0ps



Note: To revert to EPWave opening in a new browser window, set that option on your profile page.

From: 0ps To: 2,000,000ps

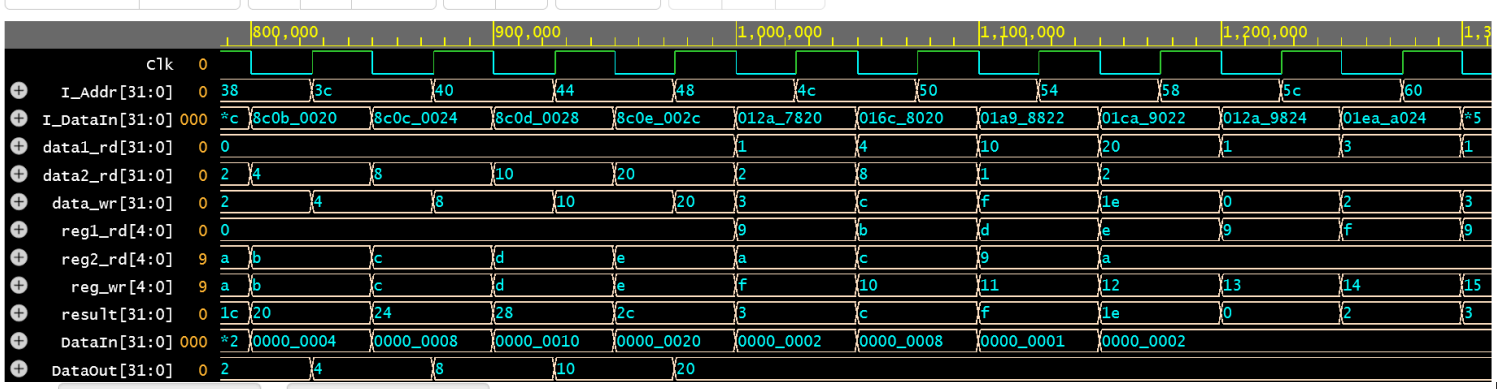
Get Signals | Radix | 100% | Δ 0ps



Note: To revert to EPWave opening in a new browser window, set that option on your profile page.

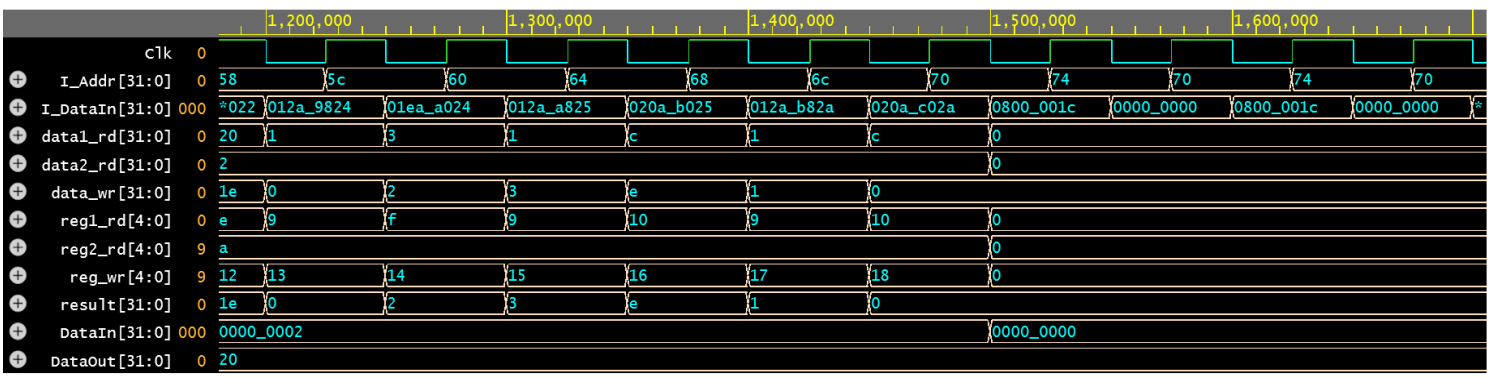
From: 0ps To: 2,000,000ps

Get Signals | Radix | 100% | Δ 0ps



From: 0ps To: 2,000,000ps

Get Signals | Radix | 100% | Δ 0ps



Note: To revert to EPWave opening in a new browser window, set that option on your profile page.

Consignas del tp

1) Material adjunto con el email. Le compartimos también el link al repositorio en github: https://github.com/Nacho1810/unicycle_processor_Untref.git

2) Analizamos por qué el banco de registros y la *Program Memory* trabajan en flanco descendente y el *Program Counter* junto a la *Data Memory* en flanco ascendente. Matizamos exactamente que la Program Memory y la Data Memory *leen* y *escriben* en los flancos de reloj descendente y ascendente respectivamente. El banco de registros es sensible a las dirección de lectura, así que lee constantemente pero *escribe* solo en flanco descendente. Y que el registro del PC se *escribe* en flanco ascendente. Esta configuración permite realizar todas las instrucciones en un ciclo de reloj, excepto la primera a la que le damos 2 ciclos para que se estabilice.
   En un principio, cuando tenemos un flanco ascendente, se actualiza el PC y se determina el valor de la señal pc_4, es decir, la próxima instrucción en caso de que no haya salto.
   Posterior a eso, en el próximo flanco descendente, se lee la instrucción de la Program Memory. Es durante ese lapso, cuando el clock está en 0, que se decodifica la instrucción: Se lee desde la dirección en el banco de registro, se ejecuta la ALU, se determina el salto si es que lo hay y se termina definiendo el valor del Program Counter.
   En el próximo flanco ascendente se almacena información en la Memoria de Datos o se lee uno de sus registros, esto dependerá de si tiene valor 1 RdStb o WrStb. Así se conoce si el valor de la señal data_w_reg es el dato leído, o queda el resultado de la ALU. Por último se actualiza el nuevo valor del PC. Lo importante es saber que aquí no se leyó de la Program Memory la nueva instrucción, seguimos trabajando con la misma instrucción.
   En el siguiente flanco descendente puede que se guarde en el banco de registro la información de la señal data_w_reg en caso de que el valor de RegWrite sea 1. Al mismo tiempo, se comienza a decodificar y procesar la nueva instrucción recién leída en la Memoria de Programa, esta nueva instrucción no interfiere con el dato a guardar porqué este se guardará en la dirección determinada por la dirección anterior, cuyo valor es la señal reg_wr.

Diagrama de instrucciones *lw*

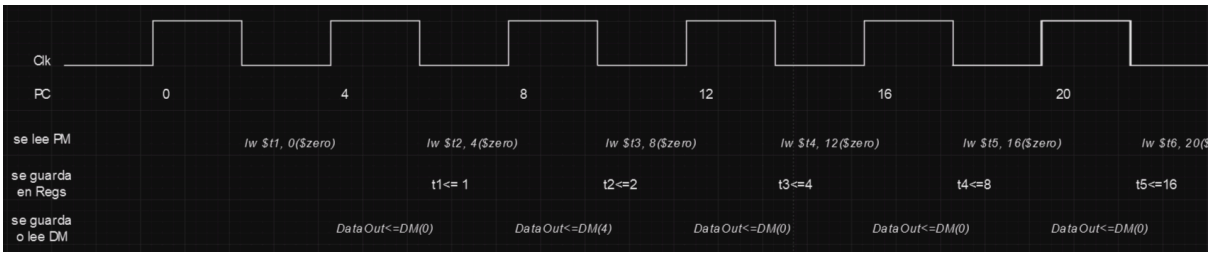


Diagrama de instrucciones *sw*

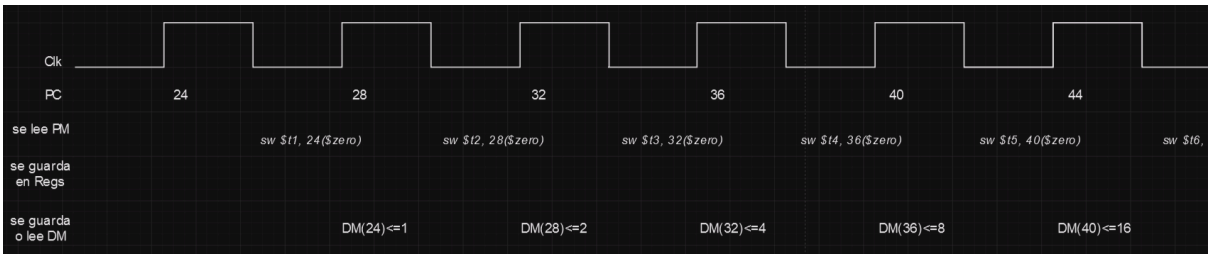


Diagrama de instrucciones tipo *R*