***Arquitectura de Computadoras***

******

**TEMA**

Trabajo Práctico N°4 MIPS (Microprocessor without Interlocked Pipeline Stages)

**Profesor:**

Ing. Rodriguez Santiago

<[santiagoarodriguez@gmail.com](mailto:santiagoarodriguez@gmail.com)>

**INTEGRANTES:**

Tissot, Esteban

<[egtissot@gmail.com](mailto:egtissot@gmail.com)>

Manero, Matias

<[matiasmanero@gmail.com](mailto:matiasmanero@gmail.com)>

**CARRERA:**

Ingeniería en Computación

**AÑO:**

2018

**INDICE**

[**Introducción**](#_lrvr6megd6uv) **2**

[**Diseño**](#_sojyu7a6h3c5) **3**

[Pipeline](#_mj8tu04yf3fp) 3

[Structural Hazards](#_e9jdx0p9xtq8) 4

[Data Hazards](#_mplvzc8oll4y) 4

[Control Hazards](#_9eti9s9f9cyh) 5

[Módulo InstructionFetch](#_1bi5iri6cl1c) 7

[Módulo InstructionDecode](#_z20tt3agyur9) 7

[Módulo Execute](#_locfkputde57) 7

[Módulo MemoryAccess](#_cf6ly8v6k75) 8

[Módulo WriteBack](#_59ar8zrt8xv8) 8

[Módulo DebugUnit](#_fpsf873ayff8) 8

[**Implementación**](#_nchxnemgtd82) **10**

[Execute](#_9vr4hzx3k634) 13

[MemoryAccess](#_biy37lzaqo8u) 13

[WriteBack](#_asj0nn2d4vsz) 14

[Summary (Nexys 4 DDR)](#_rohnbi2rn5w) 19

[Reporte de Timing](#_hgde5xosj4t1) 20

[**Programa Ensamblador**](#_tqnonyf8dhro) **22**

[**Testing**](#_3z6z8g2pxzqm) **23**

[**Conclusión**](#_bw240xn953ei) **25**

[**Anexo**](#_qtxigmlagb8) **26**

# 

# 

# Introducción

Las siglas MIPS (Microprocessor without Interlocked Pipelines Stages) hacen referencia a la gama de microprocesadores desarrollados por MIPS Technologies, de arquitectura RISC y registros de tipo propósito general, de clasificación registro-registro, siendo las únicas instrucciones que acceden a memoria son los load y store.

Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI, en muchos sistemas embebidos, en dispositivos para Windows CE, routers Cisco, y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable. Más recientemente, la NASA usó uno de ellos en la sonda New Horizons1.

La idea básica de estos microprocesadores es mejorar el rendimiento mediante el uso de la segmentación, una técnica muy conocida pero difícil de implementar. Su funcionamiento puede resumirse en que la ejecución de una instrucción es dividida en varias etapas, comenzando la "etapa 1" de una instrucción antes de que haya finalizado la ejecución de la instrucción anterior. En contraste, los diseños tradicionales esperaban la finalización por completo de una instrucción antes de pasar a la siguiente, de modo que grandes áreas de la CPU permanecían inactivas mientras el proceso continuaba. Además, la frecuencia de reloj de toda la CPU venía dictada por la latencia del ciclo completo, en lugar de por el llamado camino crítico, latencia de la etapa de segmentación que más tarda en completarse. Uno de los principales aspectos del diseño del MIPS es el marcarse como objetivo que todas las subfases (incluyendo el acceso a memoria) de todas las instrucciones tarden un único ciclo en completarse, dejando así de ser necesarios los bloqueos, y permitiendo un rendimiento de un solo ciclo. Aunque esta idea de diseño eliminó numerosas instrucciones útiles, destacando el hecho de que la multiplicación y la división necesitan varias instrucciones, en conjunto el rendimiento general del sistema es drásticamente mejorado al poder funcionar los chips a frecuencias mucho mayores. Este ascenso de la velocidad era difícil con la intervención de los bloqueos, pues el tiempo necesario es función del tamaño del chip y de la frecuencia de reloj: añadir el hardware necesario para evitarlos reduce notablemente la velocidad del sistema.

En un primer momento varios fabricantes basaron el diseño de sus Workstations en procesadores MIPS (como ACER, NEC, etc), sin embargo su uso ha decaído, enfocándose su empleo en sistemas embebidos, gracias a su bajo consumo energético entre otras cosas.

Debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades y escuelas técnicas a menudo se basan en la arquitectura MIPS. El diseño de la familia de CPU's MIPS influiría de manera importante en otras arquitecturas RISC posteriores como los DEC Alpha.

# Diseño

En el presente trabajo se desarrolla un microprocesador MIPS de 5 etapas siguiendo como guía, el capítulo 4 del libro “Computer Organization and Design”.

El primer paso es definir cada etapa y su debida función para lograr una independencia entre cada etapa y evitar cierto riesgos estructurales(De esto se hablará más adelante), por lo que procedemos a dividir en las siguientes etapas:

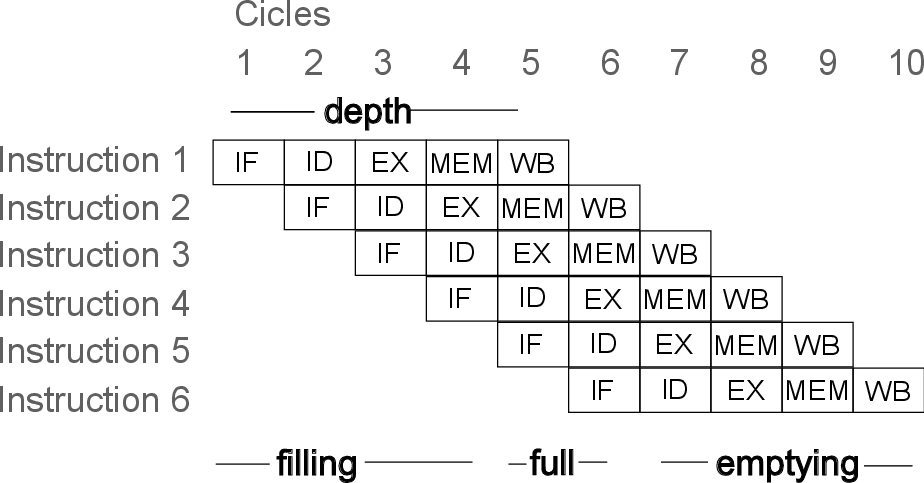
1. Instrucción Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write Brack

Estas etapas constituyen lo que llamamos pipeline, luego se agrega la unidad de debugging, la cual contiene dos máquinas de estados y un módulo de UART.

Para realizar un correcto diseño del MIPS hay que tener en cuenta ciertos conceptos claves, ya sea el correcto comportamiento del pipeline como mantener un control sobre riesgos, a continuación se explicaran brevemente.

## Pipeline

Uno de los conceptos en el cual tenemos que enfocar nuestro diseño es el funcionamiento de pipeline, el mismo mejora el rendimiento incrementando el throughput sin decrementar el tiempo de ejecución de una instrucción individual. En nuestro caso utilizaremos una Arquitectura RISC, esto significa que las instrucciones estoy tiene importancia por dos razones, la primera es que nuestro diseño se va a basar en que todas las instrucciones tiene el mismo tamaño siendo este de 32bits y la segundo es que se trata de un set reducido de instrucciones. A su vez no se lograran implementar todas las instrucciones que componen los procesadores con arquitectura MIPS, sino que se eligió un set más reducido de instrucciones para implementar.



Hay situaciones en un pipeline donde una instrucción no puede ejecutarse en el siguiente ciclo de clock. Estas situaciones son problemas potenciales (hazards) y se clasifican en 3 tipos: riesgos estructurales (**structural hazard**), riesgo de datos (**data hazards**) y riesgos de salto o de control (**control hazard**).

## Structural Hazards

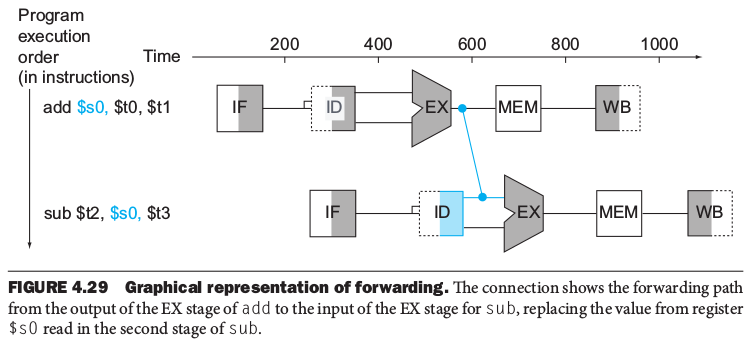
Estos problemas se encuentran cuando no se puede ejecutar una instrucción en el correspondiente ciclo de clock porque el hardware no soporta la combinación de instrucciones planificada. Un ejemplo sería si en vez de dos memorias se tiene una sola, mientras que la primer instrucción está accediendo a un dato en la etapa mem, se está buscando la cuarta instrucción en la misma memoria.

## Data Hazards

Los riesgos de datos ocurren cuando éstos son modificados. El ignorar riesgos de datos potenciales puede resultar en race condition (condiciones de carrera). Hay tres situaciones en las que puede aparecer un riesgo de datos:

* Read after Write (RAW) o dependencia verdadera: Un operando es modificado para ser leído posteriormente. Si la primera instrucción no ha terminado de escribir el operando, la segunda estará utilizando datos incorrectos.
* Write after Read (WAR) o anti-dependencia: Leer un operando y escribir en él en poco tiempo. Si la escritura finaliza antes que la lectura, la instrucción de lectura utilizará el nuevo valor y no el antiguo.
* Write after Write (WAW) o dependencia de salida: Dos instrucciones que escriben en un mismo operando. La primera en ser emitida puede que finalice en segundo lugar, de modo que el operando final no tenga el valor adecuado.

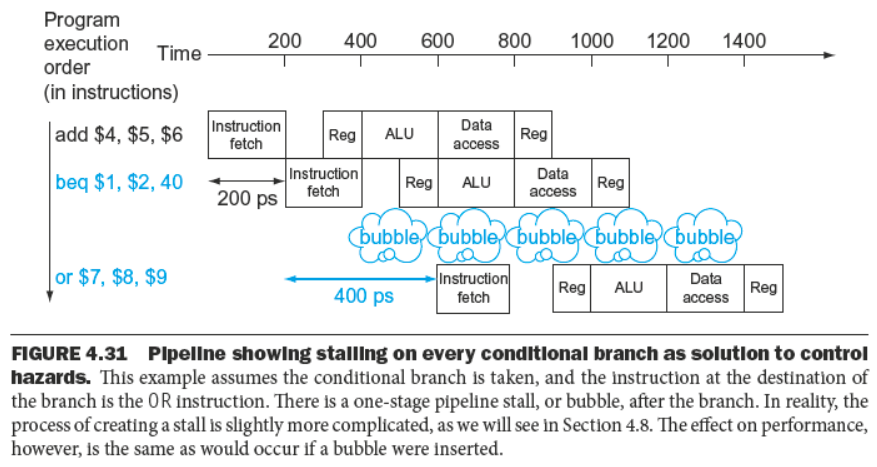
En un pipeline si hay alguna de las dependencia antes mencionadas una instrucción deberá esperar a que la anterior termine, frenando toda la cadena. Una solución sería que el compilador se encargue de eliminar estas dependencias, pero estas situaciones se repiten muy a menudo y el tiempo de espera ocioso sería demasiado. Otra, se basa en la observación de que una instrucción no necesita esperar a que la anterior termine. Si se añade hardware extra se puede obtener el dato en una etapa más temprana, esta solución se llama **forwarding** o **bypassing**.

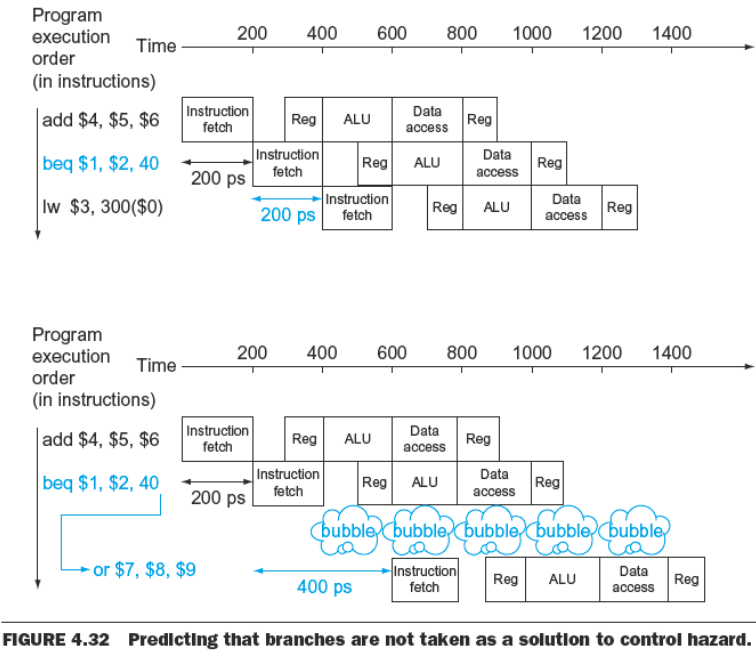


## Control Hazards

Los riesgos de salto o de control ocurren cuando el procesador se ve obligado a saltar a una instrucción que no tiene por qué ser necesariamente la inmediatamente siguiente en el código. En ese caso, el procesador no puede saber por adelantado si debería ejecutar la siguiente instrucción u otra situada en otra parte del código. Esto puede resultar en acciones no deseadas por parte de la CPU.

Para solucionar este problema, se podría detectar la instrucción branch y esperar hasta saber si hay que saltar o no, se deberá agregar hardware extra y esperar hasta que la instrucción branch esté en la segunda etapa para saber qué instrucción buscar.





Teniendo en consideración todos los conceptos que se trataron se pasa a definir las funcionalidades y alcances de las 5 etapas que hemos definido del pipeline. Las misma las ordenaremos desde la primer etapa donde se busca la instrucción hasta la última etapa donde se escriben los registros.

## Módulo InstructionFetch

Esta etapa posee una memoria de ancho de 32 bits donde se almacenan las direcciones del programa.

La funcionalidad es buscar, en dicha memoria, la instrucción a ejecutar, para ello se utilizará un registro denominado Program Counter o PC. El mismo contiene la dirección de memoria que se debe buscar para ser ejecutada. En este bloque se debe tener en consideración que cuando ocurre un salto sea condicional o incodicional el PC debe indicar la dirección de salto proveniente de las demás etapas.

**Submódulos**:

* InstructionMemory. Es una memoria que contiene el programa de usuario.

## Módulo InstructionDecode

En esta etapa se decodifica la instrucción del programa, como hemos dicho anteriormente vamos a tener un set de instrucciones reducido. Estas instrucciones las podemos dividir en tres grandes grupos que llamaremos:

Tipo-R

Tipo-Y

Tipo-J

Cada una de estas pueden tener uno, dos o tres registros asociados y diferentes señales de control. Este bloque se encargará de diferenciar cada una de ellas, buscar los registros asociados que son necesarios y setear las señales de control que correspondan en cada caso.

También se encuentra una unidad de detección de hazards, encargada de incorporar burbujas al flujo del programa.

**Submódulos**:

* HazardDetectionUnit. Esta unidad se encarga de detectar los hazard provocados por las instrucciones de load, detener el PC e incorporar una burbuja en el pipe.
* ControlBlock. Este bloque setea las señales de control de acuerdo al opcode de la instrucción.
* FileRegister. Es una memoria que contiene los 32 registros internos del microprocesador.

## Módulo Execute

En este módulo se realizan las operaciones que involucren a la unidad de ALU, entre ellas las operaciones entre registros o cálculo de direcciones, y posee un submódulo “ForwardingUnit” el mismo se encarga de realizar los “cortocircuitos” necesarios para evitar frenar el pipeline si dos instrucciones consecutivas hacen uso del mismo registro.

Este a su vez contiene multiplexores para definir diferentes entradas a la ALU, estos mismos se manejan con los bits de control que salen del módulo anterior

**Submódulos**:

* ALU. Unidad Aritmética Lógica
* AluControl. Este módulo se encarga de setear las señales de control para la ALU.
* ForwardingUnit. Este módulo se encarga de solucionar los data hazard (cortocicuitos).

## Módulo MemoryAccess

En esta etapa es en el único momento en que se accede a memoria a través de las instrucciones de Load y Store. El resto de las instrucciones en esta etapa no realizan ninguna tarea.

**Submódulos**:

* DataMemory. Es la memoria principal.

## Módulo WriteBack

Este módulo es el encargado de regresar el dato a escribir en los registros, el dato puede provenir de múltiples lugares. Por lo que posee un multiplexor que es manejado por las señales de control para informar que dato va a ser guardado en el registro y si va a ser guardado

**Submódulos:**

* No tiene.

## Módulo DebugUnit

En este módulo se realizan dos funciones fundamentales, en un primer momento todas las memorias que forman parte del MIPS se encuentra “limpias”, es decir todos su registros son ceros. Por lo que será función de esta unidad cargar el respectivo programa en la memoria que se encuentra en la etapa de InstructionFetch, para hacer esto debe recibir una por una las instrucciones por medio de la comunicación serial y almacenar dichos datos en direcciones ascendentes empezando por la dirección 0.

Los datos recibidos se van a cargar hasta recibir una instrucción de Halt, luego de esto el Módulo DebugUnit pasa a cumplir con su segunda función.

Se trata de enviar por medio de comunicación serial la información parcial o total del pipeline, con esto nos referimos a valores de memoria, latch intermedios, pc, etc. Dependiendo de la configuración que el usuario ha elegido se podrá ver en cada ciclo del pipeline los valores de interés o se puede recibir el valor de todas las señales al final del programa.

**Submódulos**:

* Top\_UART. Este módulo instancia 5 archivos (baud\_rate\_generator.v, receptor.v, interfaz\_rx.v, interfaz\_tx.v, transmisor.v) que se encargan de recibir y decodificar los datos recibidos y de hacer el proceso inverso.

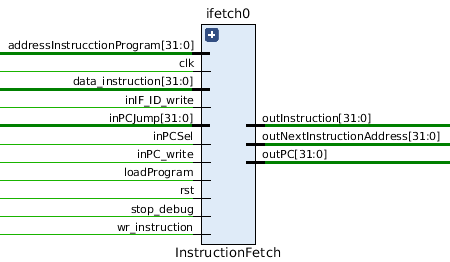
# 

# Implementación

Para la implementación se codificó en lenguaje Verilog las distintas etapas del pipeline como así también el debugunit, para la codificación se utilizó el framework Vivado y se probó y testeo en una placa Xilinxs Nexys 4 DDR.

A continuación se muestra como quedaron terminada cada etapa por medio de su vista RTL, con esta vista queremos mostrar claramente cuáles son las entradas y salidas de las etapas y sus correspondiente ancho de bus.

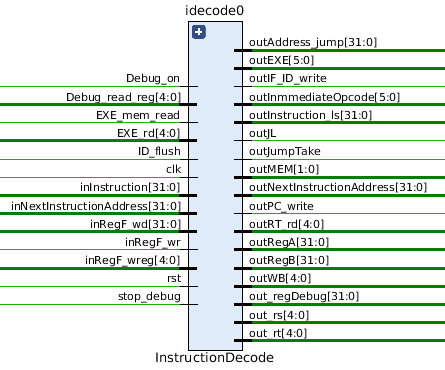
InstructionFetch



Como podemos observar las salidas se encuentra de lado derecho y esta etapa solo posee dos de ellas, la instrucción(outInstruction) y la dirección de memoria de la próxima instrucción.

La mayoría de las entradas a este módulo provienen del DebugUnit debido que este carga el programa en memoria y detiene el Pc cuando es necesario. También posee entradas relacionadas a los jump, brach y sus direcciones de salto.

InstructionDecode



En esta etapa nos centraremos en la señal de control, las mismas pueden ser observadas a la salida y están divididas en 4 partes, donde el nombre de las partes indica en qué etapa van a ser usadas, estas son:

* outEXE: 6bits
* outMEM: 2 bits
* outWB: 5bits
* outJL : 1 bit

1. Instrucción Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write Brack

Cada bits de estas salidas tiene su respectivo uso que se explicaran en la tabla a continuación.

**Control Block:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ControlBlock.v | | | | | |
| OPCODE | JUMP | EXE | MEM | WB | INSTRUCCION |
| 6'd0  6'd35  6'd20  6'h20  6'h24  6'h21  6'h25  6'h28  6'h29  6'd43  6'd4  6'd5 6'd8,6'hc,6'hd,6'he  6'd2  6'd3  default: | 00  00  00  00  00  00  00  00  00  00  00  00  00  10  11  00 | 001100  000001  000001  000001  000001  000001  000001  000001  000001  000001  000001  110000  000111  000000  000000  000000 | 00  10  10  10  10  10  10  01  01  01  00  00  00  00  00  00 | 000\_10  000\_11  000\_11  101\_11  001\_11  110\_11  010\_11  001\_00  010\_00  000\_00  000\_00  000\_00  000\_10  000\_01  000\_11  000\_00 | R-Type  Load word  Load byte  Load signed byte  Load unsigned byte  Load signed halfword  Load unsigned halfword  Store byte  Store halfword  Store word  Branch equal  Branch not equal  Inmediato  Jump  Jump and Link  Default |

**Referencias:**

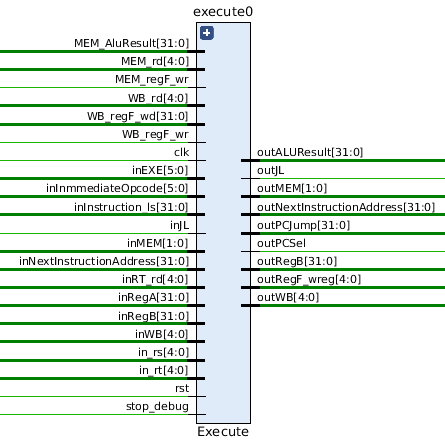
|  |  |
| --- | --- |
| JUMP | |
| bit\_1 | bit\_0 |
| JUMP | LINK |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Execute | | | | | |
| bit\_5 | bit\_4 | bit\_3 | bit\_2 | bit\_1 | bit\_0 |
| equal or  not equal | Branch | Select rd | Instruction Type | | MUX\_B |

|  |  |
| --- | --- |
| MEMORY | |
| bit\_1 | bit\_0 |
| Read Mem | Write Mem |

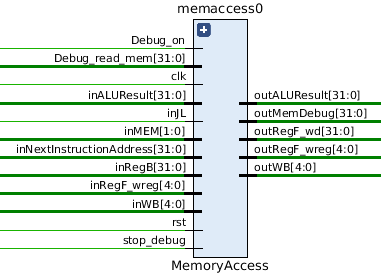
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WriteBack | | | | |
| bit\_4 | bit\_3 | bit\_2 | bit\_1 | bit\_0 |
| Control Load Byte/Load Halfword | | | Write Register | Source Write Register |

## Execute



En este módulo se decidió implementar las instrucciones de salto condicional y de tipo J ya que para estas instrucciones necesitan hacer uso de los registros internos del micro.

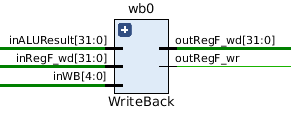
## MemoryAccess



En esta etapa se almacenan datos en la memoria y se los trae de ella. Para implementar los “load” y “store” de halfword y de byte, se debe visualizar la memoria como una matriz de 2 columnas y de 4 columnas respectivamente.

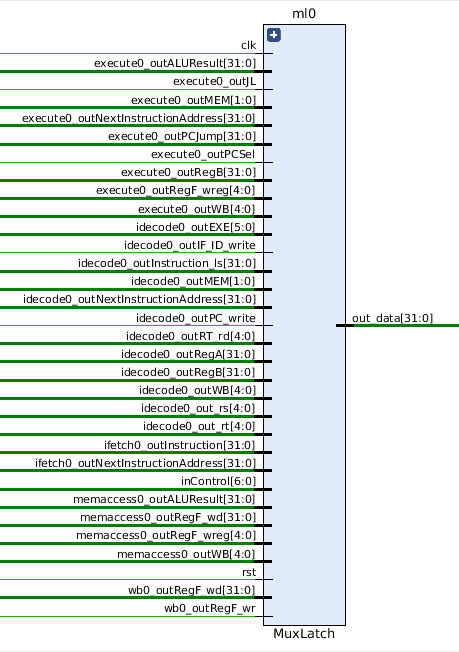
Para saber cual de los 4 octetos de una posición de memoria se va a escribir o leer, se utilizan los últimos dos bits de la instrucción de load/store byte, y para el caso de load/store halfword el último bit.

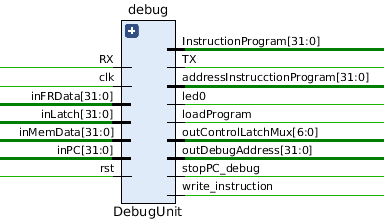
## WriteBack



Este módulo tiene la particularidad que no está asociado a un clock, es decir que es un circuito combinacional que se encarga de escribir en los registros internos del MIPS. Por conveniencia y para evitar riesgos al acceso de registro se debe optar por escribir en el flanco de clock de subida así el módulo InstruccionDecode que lee registro puede hacer uso de la memoria de registro en el flanco de bajada.

DebugUnit





Para el correcto funcionamiento de este módulo se necesita hacer uso de un multiplexor (MuxLach) que se encargue de elegir el dato correcto para ser enviado por UART. Su funcionamiento es el siguiente, se envía la información de memorias y latch, es decir el estado del pipeline en un momento dado y para eso se necesitan los correspondientes registros, el MuxLatch agrupa los datos en registros de 32bits y los envía al debugUnit a pedido. Esto permite mayor sencillez y modularización.

A continuación se va a explicar el comportamiento del módulo en sí, como ya se mencionó, en el diseño, tiene dos funciones muy importantes la recepción y la emisión de datos desde la PC y hacia ella, por lo que para realizar esto consta de dos máquinas de estados separadas que manejan estos dos aspectos.

|  |
| --- |
|  |
| *Máquina de estado Receptora* |

En el primer estado de la máquina receptora, se espera hasta recibir el modo en que debe trabajar el sistema, como ya se aclaró anteriormente existen dos modos de operar, se puede ir viendo la información a medida que las instrucciones avanzan en el pipeline (modo paso a paso) o solo al final del programa (modo continuo). Esta configuracion se guarda en una variable llamada “debug\_mode”.

En el segundo estado se reciben, una por una, las instrucciones del programa de usuario y se cargan los datos recibidos en la memoria de instrucciones. Cuando termina el envío del programa por UART, se recibe una instrucción de Halt que fue definida por los programadores.

Luego de esto la máquina pasa al último estado dando por finalizada la carga de datos en la memoria.

|  |
| --- |
|  |
| *Funcionamiento Maquina de Estado transmisora, envía el programa una vez finalizado (debug\_mode=0)* |

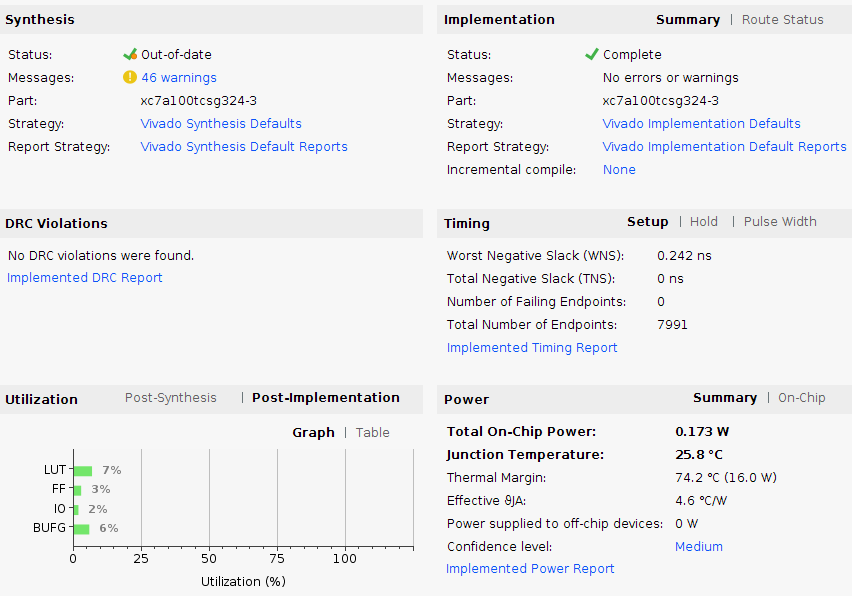
|  |
| --- |
|  |
| *Funcionamiento Maquina de Estado transmisora, envía el programa ciclo por ciclo de instrucción(debug\_mode=0)* |

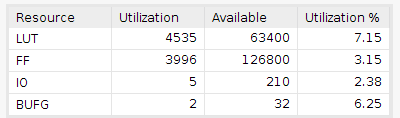
En cuanto a la máquina de estados que maneja la emisión de datos, está a la espera de la finalización de la máquina receptora. Cuando esto sucede, se avanza al siguiente estado dependiendo del funcionamiento seteado por el usuario. En caso de la variable debug\_mode valga cero(el programa se envia al final), se permanece en el estado inicial hasta que el PC sea igual a la cantidad de instrucciones recibidas más cuatro, esto es para poder limpiar el pipeline. Luego se avanza por los distintos estados donde se envía la información correspondiente (PC, memoria de registros y latchs intermedios de cada etapa).

El estado llamado send\_waitFinish se utiliza para esperar hasta que el dato sea enviado por completo. Esto es necesarios debido a dos cuestiones importantes, el protocolo UART trabaja a una frecuencia menor a la del clock del sistema y, además, la cantidad de bits que se envía en cada transmisión es de 8bits mientras que los registros que se deben enviar son de 32. Por lo tanto se debe parsear el dato y enviarlo en 4 tramos.

Por último, en el estado final si se desea enviar la información al final del programa permanece en este estado dando por finalizada la máquina, en cambio del otro modo tanto el estado inicial como el final van a permitir que se recorra toda la máquina de estado cada vez que se sume una dirección nueva al registro PC, para lograr esto se debe detener todos los registros intermedios para que las instruccione no avancen en el pipeline, esto se logra con una señal al cual denominamos “stop\_debug”. Este cable funciona como un enable del módulo, esto quiere decir que el PC y las salidas se mantendrán constantes mientras que esté en 1.

## Summary (**Nexys 4 DDR)**





## 

## Reporte de Timing

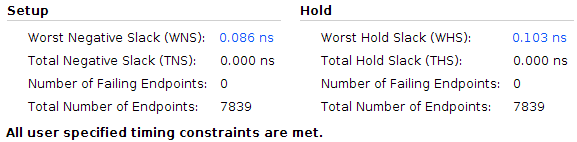
**System Clock**

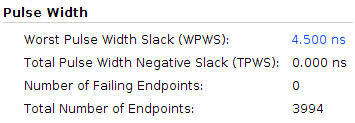
El system clock es el clock del sistema con el que se realizó la implementación. En las siguientes imágenes se muestran más información que aseguran el correcto funcionamiento en la FPGA.



**Design Timing Summary**

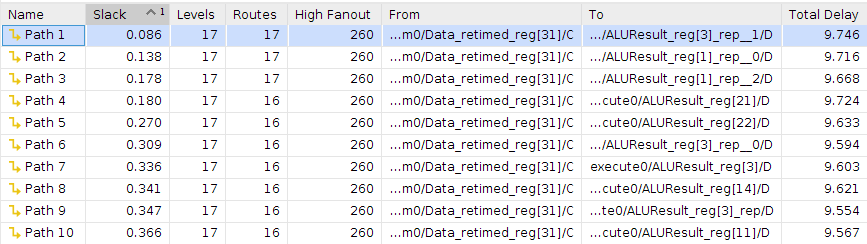
En esta sección se muestra un resumen del reporte de timing, se puede observar que el peor tiempo de propagación (WNS) es de 0.086ns, y que el tiempo estable que se necesita para que los datos sean capturados por el componente receptor (WHS) es de 0.103ns. Otro análisis que podemos observar es que el ancho mínimo del pulso (WPWS) es de 4.5ns, lo que significa que el límite de frecuencia del clock según este parámetro es de 222MHz. Esta frecuencia, en este caso, es restringida aún más por el camino crítico.





**Worst Negative Slack (WNS)**

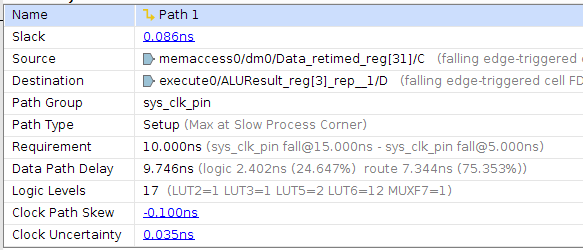
Si se entra más en detalle del WNS se puede obtener una lista en orden descendente de los caminos (paths) que tienen el mayor delay.



**Path Properties**

En la siguiente imagen se muestra las características del path1 que va desde el registro Data[31] del módulo memaccess0 al registro AluResult[3] del módulo execute0. Se impuso como requisito un periodo de 10ns(100MHz) y se obtuvo un delay de 9.746ns y un slack de 0.086ns.

Por ser este path el de mayor delay se convierte en el camino crítico de todo el sistema, por lo que limita a un clock máximo de 102.6MHz. Si bien este trabajo se ejecuto con una frecuencia de 100MHz se recomienda, si es posible, trabajar a una frecuencia menor.



# 

# Programa Ensamblador

Como parte del trabajo se desarrolla un programa que se encargue de traducir instrucciones para convertirlas a binario y enviarlas por UART.

Para esto se codifica un programa en python y se carga manualmente el set de instrucciones que se va a utilizar, en una estructura tipo key:value siendo la key el nombre de la instrucción por ejemplo ADDU. Luego como value se asoció un formato determinado, esto quiere decir que para las instrucciones de tipo- R como por ejemplo una ADDU el formato que se cargo es el siguiente:

*000000\_RS\_RT\_RD\_00000\_100001*

El funcionamiento del programa es el siguiente, cuando se pone en ejecución lo que primero que se muestra es la pregunta de en qué modo de debugging va a funcionar el sistema en la fpga, luego se pregunta de donde el usuario carga las instrucciones. Aquí existen dos modos, el por defecto es de un archivo con formato txt o el usuario puede ir ingresando instrucción por instrucción en la consola que esté corriendo el programa en python.

Cuando el programa tiene la instrucción en formato assembler lo primero que hace es pasar todas las letras a minúsculas y quitar la letra “r” o “$t” que se utiliza para indicar la presencia de un registro, luego se busca el formato de la instrucción por medio del key y se reemplazan las variables que aparecen en la estructura por los datos del assembler. Este procedimiento se va imprimiendo en la consola.

Por ejemplo en el formato de la ADDU se reemplaza las variables RS, RT y RD para luego proceder a enviarlos por UART . Al concluir con la última instrucción en el archivo o la ingresada por el usuario, se envia un codigo unico “01111111111111111111111111111110” para indicar la finalización del envío de datos y acto seguido se prepara para recibir la información que va provenir de la FPGA.

La transmisión de datos por UART se realiza en código ascii, lo que significa que cuando se reciben los datos, el python calcula el número decimal asociado al carácter ascii y le resta 48 para obtener el número puro y luego lo traduce a binario. En el caso inverso se pondera el binario y se manda el caracter ascii del binario, por lo que en la FPGA no es necesario restarle las 48 unidades decimales.

Por otro lado, es necesario aclarar que el programa envía datos cada un segundo aproximadamente, esto es porque la unidad de recepción de datos de la FPGA aunque está preparada para recibir 4 tramos de 8 bits seguidos no cuenta con un buffer, por lo que es necesario darle tiempo a que procese el dato antes de volver a enviarle otro.

# Testing

A continuación se muestran los dos programas creados por los integrantes del grupo para verificar el correcto funcionamiento de MIPS. Estos son los test de integración (Black Box) pero cabe aclarar que a medida que se fue desarrollando el trabajo se realizaron distintas simulaciones y pruebas unitarias que no se incorporan en el informe.

**Test1:** Este test prueba las instrucciones de TIPO-R, Store Word, Load Word, Load Byte, Branch Equals y Jumps.

|  |  |  |
| --- | --- | --- |
| Programa de testing 1 | | |
| 00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | ADDI R1 R0 8  ADDI R2 R0 3  ADDI R6 R0 1  ADDI R9 R0 14  ADDI R11 R0 127  ADDI R13 R0 6  SW R1 6 2  SW R6 2 4  SW R9 6 3  SW R11 1 0  ADDU R3 R13 R6  AND R4 R1 R9  OR R5 R9 R2  XOR R6 R1 R9  BEQ R1 R3 17  ADDU R3 R6 R3  J 11  SLL R1 R3 6  SUBU R4 R5 R1  LW R8 2 0  LB R12 0 1024  SLL R6 R9 1  SLL R6 R9 1  SLL R6 R9 1  SLL R6 R9 1  SRA R10 R9 1 | R1 <- R0(0) + 8  R2 <- R0(0) + 3  R6 <- R0(0) + 1  R9 <- R0(0) + 14  R11 <- R0(0) + 127  R13 <- R0(0) + 6  M3 <- R1(8)  M7 <- R6(1)  M4 <- R9(14)  M8 <- R11(127)  R3 <- R13(6) + R6(1)  R4 <- R1(8) & R9(14)  R5 <- R9(14) | R2(3)  R6 <- R1(8) XOR R9(14)  PC <- 17 SI R3(7) SEA IGUAL A R1(8)  R3 <- R6(1) + R3(7)  PC <- 11  R1[8:0](0000\_0110) << 6 = R3[8:0](1000\_0000)  (8)R4 - (15)R5 = (-7)R1  R8 <- M2(3)  R12 <- M8\_1(?)  SLL R6 R9 1  SLL R6 R9 1  SLL R6 R9 1  SLL R6 R9 1  R10 <- R9(1110) >> 1 |

**Test 2:**

Este test prueba las instrucciones Tipo-I, Store Halfword, Jump and Link, Branch Not Equal y las unidades de Forwarding y Hazard.

|  |  |  |
| --- | --- | --- |
| Programa de testing 2 | | |
| 00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | ORI R1 R0 1  ORI R2 R0 2  ORI R3 R2 12  ANDI R4 R3 10  ADDI R10 R0 34  ADDU R5 R1 R2  SUBU R6 R1 R2  ADDU R7 R4 R6  AND R8 R1 R6  SLL R9 R3 6  SLLV R3 R6 R4  SH R4 R1 0  ADDU R11 R1 R2  ADDU R12 R1 R2  SH R3 R1 1  SB R9 R1 2  SW R3 R1 3  ANDI R8 R0 3  LW R14 R1 0  ADDU R2 R1 R14  LH R16 R1 8  LHU R17 R1 8  LB R18 R1 16  JAL 36  ADDU R1 R2 R11  ADDU R1 R2 R12  ADDU R1 R2 R11  ADDU R1 R2 R12  ADDU R1 R2 R11  ADDU R1 R2 R12 | R1 <- R0(0) | 1  ***forwarding***  R2 <- R0(0) | 2  R3 <- R2(2) | 12  R4 <- R3(14) | 10  R10 <- R0(0) + 34  R5 <- R1(1) - R2(2)  R6 <- R1(1) - R2(2)  R7 <- R4(10) + R6(-1)  R8 <- R1(1) & R6(-1)  ***Shift y Store***  R9 <- R3(‘b0000\_1110) << 6  R3 <- R6('hFFFFFFFF) << 10  M1 <- (10)R4 (halfword)  R11 <- R1(1) + R2(2)  R12 <- R1(1) + R2(2)  M2 <- R3('hFC00) (halword)  M3 <- R9('b1100\_0000) (byte)  M4 <- R3('hFFFFFC00) (word)  R8 <- R0(0) & 3  ***Hazard***  R14 <- M1(10)  R2 <- R1(1) + R14(10)  R6 <- ('hFFFF)  R16 <- ('hFFFF)  R16 <- ('hFC)  R31 <- PC ; PC <- 36  R11 <- R1(1) + R2(2)  R12 <- R1(1) + R2(2)  R11 <- R1(1) + R2(2)  R12 <- R1(1) + R2(2)  R11 <- R1(1) + R2(2)  R12 <- R1(1) + R2(2) |

# Conclusión

Se logró implementar un procesador MIPS de 5 etapas con los controles necesarios de hazard y forwarding, si bien el registro de instrucciones desarrollados en hardware es más reducido que un RISC se logró la implementación de las más comunes incluyendo diferentes tipos de saltos condiciones y no condicionales.

Si bien el desarrollo de la memoria de datos fue con ancho de palabra de 32bits, con señales de control se logró que el usuario cuando ingrese instrucciones de load o store, de byte o halfword ,pueda ver la memoria dividía en palabras de 8 bits o 16 bits.

Con la información obtenida del reporte de timing podemos concluir que el clock que utilizamos es muy elevado 100MHz, por lo que para usos más prácticos se recomienda utilizar una frecuencia de clock menor.

En cuanto el programa ensamblador si bien está acotado a las instrucciones cargadas manualmente en él, se ha desarrollado de manera que sea muy sencillo agregar más instrucciones en caso de ser necesario. Siempre siendo coherentes con las instrucciones que se encuentran desarrolladas en hardware.

# 

# Anexo

|  |  |  |  |
| --- | --- | --- | --- |
| Set de Instrucciones | | | |
| Tipo | Nombre | Estructura | Ejemplo |
| Tipo R | Shift L logical  Shift R logical  Shift R arithmetic  Shift L logical variable  Shift R logical variable  Shift R arithmetic variable  Addition (without overflow)  Subtract (without overflow)  AND  OR  Exclusive OR  NOR  Set less than  Jump register  Jump and link register | sll rd rt shamt  srl rd rt  sra rd rt  sllv rd rt rs  srlv rd rt rs  srav rd rt rs  addu rd rt rs  subu rd rt rs  and rd rt rs  or rd rt rs  xor rd rt rs  nor rd rt rs  slt rd rt rs  jr rt  jalr rd rs | sll $t10 $t6 1  srl $t11 $t6 2  sra $t12 $t4 3  sllv $t20 $t6 $t1  srlv $t21 $t6 $t1  srav $t22 $t6 $t1  addu $t23 $t6 $t1  subu $t24 $t6 $t1  and $t25 $t6 $t1  or $t26 $t6 $t1  xor $t27 $t6 $t1  nor $t28 $t6 $t1  slt $t29 $t6 $t1  jr $t50  jalr $t51 $t16 |
| Tipo J | Jump  Jump and link | j target  jal target | j 26  jal 26 |
| Tipo I | Load byte  Load halfword  Load word  Load unsigned byte  Load unsigned halfword    Store byte  Store halfword  Store word    ADD immediate (with overflow)  AND immediate (with overflow)  OR immediate  XOR immediate  Set less than immediate  Branch on equal  Branch on not equal    Load upper immediate | lb rt offset  lh rt offset  lw rt offset  lbu rt offset  lhu rt offset  sb rt offset  sh rt offset  sw rt offset  addi rt rs imm  andi rt rs imm  ori rt rs imm  xori rt rs imm  slti rt rs imm  beq rt rs imm  bne rt rs imm  lui rt imm | lb $t30 16  h $t31 16  lw $t32 16  bu $t33 16  lhu $t34 16  sb $t35 16  sh $t36 16  sw $t37 16  addi $t40 $t6 16  andi $t41 $t6 16  ori $t42 $t6 16  xori $t43 $t6 16  slti $t44 $t6 16  beq $t45 $t6 16  bne $t46 $t6 16  lui $t47 16 |
| - | Finish the program | halt | halt |

|  |
| --- |
| user\_interfaz.py |
| import serial  import time  # Esta tabla contiene el set de instrucciones implementado.  # Se utiliza para mostrar informacion al usuario.  def instruction\_set(argument):  instr = {  #KEY : (instruction, example, description)  1 : ('Shift L logical ' ,'sll rd rt shamt' , 'sll $t10 $t6 1 '),  2 : ('Shift R logical ' ,'srl rd rt ' , 'srl $t11 $t6 2 '),  3 : ('Shift R arithmetic ' ,'sra rd rt ' , 'sra $t12 $t4 3 '),  4 : ('Shift L logical variable ' ,'sllv rd rt rs ' , 'sllv $t20 $t6 $t1'),  5 : ('Shift R logical variable ' ,'srlv rd rt rs ' , 'srlv $t21 $t6 $t1'),  6 : ('Shift R arithmetic variable' ,'srav rd rt rs ' , 'srav $t22 $t6 $t1'),  7 : ('Addition (without overflow)' ,'addu rd rt rs ' , 'addu $t23 $t6 $t1'),  8 : ('Subtract (without overflow)' ,'subu rd rt rs ' , 'subu $t24 $t6 $t1'),  9 : ('AND ' ,'and rd rt rs ' , 'and $t25 $t6 $t1'),  10 : ('OR ' ,'or rd rt rs ' , 'or $t26 $t6 $t1'),  11 : ('Exclusive OR ' ,'xor rd rt rs ' , 'xor $t27 $t6 $t1'),  12 : ('NOR ' ,'nor rd rt rs ' , 'nor $t28 $t6 $t1'),  13 : ('Set less than ' ,'slt rd rt rs ' , 'slt $t29 $t6 $t1'),  14 : ('Load byte ' ,'lb rt offset ' , 'lb $t30 16 '),  15 : ('Load halfword ' ,'lh rt offset ' , 'lh $t31 16 '),  16 : ('Load word ' ,'lw rt offset ' , 'lw $t32 16 '),  17 : ('Load unsigned byte ' ,'lbu rt offset ' , 'lbu $t33 16 '),  18 : ('Load unsigned halfword ' ,'lhu rt offset ' , 'lhu $t34 16 '),  19 : ('Store byte ' ,'sb rt offset ' , 'sb $t35 16 '),  20 : ('Store halfword ' ,'sh rt offset ' , 'sh $t36 16 '),  21 : ('Store word ' ,'sw rt offset ' , 'sw $t37 16 '),  22 : ('ADD immediate (with overflow)','addi rt rs imm ' , 'addi $t40 $t6 16 '),  23 : ('AND immediate (with overflow)','andi rt rs imm ' , 'andi $t41 $t6 16 '),  24 : ('OR immediate ' ,'ori rt rs imm ' , 'ori $t42 $t6 16 '),  25 : ('XOR immediate ' ,'xori rt rs imm ' , 'xori $t43 $t6 16 '),  26 : ('Set less than immediate ' ,'slti rt rs imm ' , 'slti $t44 $t6 16 '),  27 : ('Branch on equal ' ,'beq rt rs imm ' , 'beq $t45 $t6 16 '),  28 : ('Branch on not equal ' ,'bne rt rs imm ' , 'bne $t46 $t6 16 '),  29 : ('Load upper immediate ' ,'lui rt imm ' , 'lui $t47 16 '),  30 : ('Jump ' ,'j target ' , 'j 26 '),  31 : ('Jump and link ' ,'jal target ' , 'jal 26 '),  32 : ('Jump register ' ,'jr rt ' , 'jr $t50 '),  33 : ('Jump and link register ' ,'jalr rd rs ' , 'jalr $t51 $t16 '),  34 : ('Finish the program ' ,'halt ' , 'halt '),  }  return instr.get(argument)  # Esta tabla contiene la estructura de las instrucciones implementadas en el MIPS.  # Se utiliza para codificar el assembler escrito por el usuario.  def opcode(argument):  op = {  #KEY : (estructura,cant\_param, param1,param2,..)  'sll' : ('000000\_00000\_RT\_RD\_SHAMT\_000000',4,'REG\_D','REG\_T','SHAMT'),  'srl' : ('000000\_00000\_RT\_RD\_SHAMT\_000010',4,'REG\_D','REG\_T','SHAMT'),  'sra' : ('000000\_00000\_RT\_RD\_SHAMT\_000011',4,'REG\_D','REG\_T','SHAMT'),  'sllv' : ('000000\_RS\_RT\_RD\_00000\_000100',4,'REG\_D','REG\_T','REG\_S'),  'srlv' : ('000000\_RS\_RT\_RD\_00000\_000110',4,'REG\_D','REG\_T','REG\_S'),  'srav' : ('000000\_RS\_RT\_RD\_00000\_000111',4,'REG\_D','REG\_T','REG\_S'),  'addu' : ('000000\_RS\_RT\_RD\_00000\_100001',4,'REG\_D','REG\_S','REG\_T'),  'subu' : ('000000\_RS\_RT\_RD\_00000\_100011',4,'REG\_D','REG\_S','REG\_T'),  'and' : ('000000\_RS\_RT\_RD\_00000\_100100',4,'REG\_D','REG\_S','REG\_T'),  'or' : ('000000\_RS\_RT\_RD\_00000\_100101',4,'REG\_D','REG\_S','REG\_T'),  'xor' : ('000000\_RS\_RT\_RD\_00000\_100110',4,'REG\_D','REG\_S','REG\_T'),  'nor' : ('000000\_RS\_RT\_RD\_00000\_100111',4,'REG\_D','REG\_S','REG\_T'),  'slt' : ('000000\_RS\_RT\_RD\_00000\_101010',4,'REG\_D','REG\_S','REG\_T'),  'lb' : ('100000\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Load byte -> lb rt address  'lh' : ('100001\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Load halfword -> lh rt address  'lw' : ('100011\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Load word -> lw rt address  'lbu' : ('100100\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Load unsigned byte -> lbu rt address  'lhu' : ('100101\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Load unsigned halfword -> lhu rt address  'sb' : ('101000\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Store byte -> sb rt address  'sh' : ('101001\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Store halfword -> sh rt address  'sw' : ('101011\_RS\_RT\_OFFSET',4,'REG\_T','REG\_S','OFFSET'), #Store word -> sw rt address  'addi' : ('001000\_RS\_RT\_IMMEDIATE',4,'REG\_T','REG\_S','IMMEDIATE'), #Addition immediate -> addi rt rs im  'andi' : ('001100\_RS\_RT\_IMMEDIATE',4,'REG\_T','REG\_S','IMMEDIATE'), #AND immediate -> andi rt rs im  'ori' : ('001110\_RS\_RT\_IMMEDIATE',4,'REG\_T','REG\_S','IMMEDIATE'), #OR immediate -> ori rt rs im  'xori' : ('001111\_RS\_RT\_IMMEDIATE',4,'REG\_T','REG\_S','IMMEDIATE'), #XOR immediate -> xori rt rs im  'lui' : ('010000\_00000\_RT\_IMMEDIATE',3,'REG\_T','IMMEDIATE'), #Load upper immediate -> lui im  'slti' : ('001010\_RS\_RT\_IMMEDIATE',4,'REG\_T','REG\_S','IMMEDIATE'), #Set less than imm -> sltirt rs im  'beq' : ('000100\_RS\_RT\_OFFSET',4,'REG\_S','REG\_T','OFFSET'), #Branch on equal -> beq rs rt label  'bne' : ('000101\_RS\_RT\_OFFSET',4,'REG\_S','REG\_T','OFFSET'), #Branch on not equal -> bne rs rt label  'j' : ('000010\_TARGET',2,'TARGET'), #jump -> j target  'jal' : ('000011\_TARGET',2,'TARGET'), #jump and link -> jal target  'jr' : ('000000\_RS\_000000000000000\_001000',2,'REG\_S'), #jump register -> jr rs  'jalr' : ('000000\_RS\_00000\_RD\_00000\_001001',3,'REG\_S','REG\_D'), #jump and link register -> jalr rs,rd  'halt' : ('01111111111111111111111111111110') # HALT  }  return op.get(argument)  # Esta tabla contiene el orden y etiqueta de los datos que se reciben por UART  # Se utiliza para presentar al usuario los datos recibidos.  def parseo(argument,byte1,byte2,byte3,byte4):  switcher = {  0: '\n\t\t\t Program Counter\nPC : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  53: '\nStage : Instruction Fetch\nNextInstructionAddress : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  54: 'Instruction \t: 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  55: '\nStage : Instruction Decode\nEXE : 0x{:0>2x} MEM : 0x{:0>2x} WB : 0x{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte2,byte3,byte4,byte1,byte2,byte3,byte4),  56: 'NextInstructionAddress : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  57: 'RegA \t\t : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  58: 'RegB \t\t : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  59: 'Instruction\_ls : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  60: 'dir\_Rs : 0x{:0>2x} dir\_Rt : 0x{:0>2x} dir\_Rd : 0x{:0>2x} PC\_write[1] IF\_ID\_write[0] : 0x{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  61: '\nStage : Execute\nMEM : 0x{:0>2x} WB : 0x{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte3,byte4,byte1,byte2,byte3,byte4),  62: 'NextInstructionAddress : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  63: 'PCJump : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  64: 'ALUResult : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  65: 'RegB : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  66: 'JL : 0x{:0>2x} PC\_Sel : {:0>2x} RegF\_wreg : {:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte2,byte3,byte4,byte1,byte2,byte3,byte4),  67: '\nStage : Memory\nWB : 0x{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte4,byte1,byte2,byte3,byte4),  68: 'RegF\_wd : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  69: 'ALUResult : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  70: 'RegF\_wreg : {:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte4,byte1,byte2,byte3,byte4),  71: '\nStage : Write Back\nRegF\_wd : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4),  72: 'RegF\_wr : {:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte4,byte1,byte2,byte3,byte4)  }  print switcher.get(argument, "Error!!")  # Esta funcion se encarga de enviar los datos por UART  # Parsea el argumento de 32bits en 4 grupos de 8bits, con 8 bits se representa un caracter ascii.  # Suma el peso de cada bit y concatena el ascii correspondiente a ese valor. Luego envia los 4 caracteres ascii  def send(argument):  print(argument)  send = ''  for j in range (0,32,8): # parseo el string en 4 grupos de 8 caracteres  a = argument[j:j+8]  cont = 0  for i in range (0,8,1): # tomo cada uno de los caracteres del grupo  cont = cont + int(a[i])\*(2\*\*(7-i)) # calculo el peso y lo sumo  send = send + '{:c}'.format(cont)  print('send: {} len: {}'.format(send, len(send)))  ser.write(send)  # Esta funcion se encarga de desarmar la instruccion de assembler y transformarla en 32 caracteres (1 o 0)  # Es necesario reordenar y adicionar datos segun la instruccion.  def decode(cmd):  print(cmd)  try:  cmd = cmd.lower()  instruction = cmd.split(' ')  print(instruction)  binary\_instruction = ''  struct = opcode(instruction[0])[0]  binary\_instruction = binary\_instruction + '{}'.format(struct)  print("")  print(binary\_instruction + ' <- estructura')  cant\_param = opcode(instruction[0])[1]  print(cant\_param)  for i in range(2,cant\_param+1,1):  instruction[i-1]=instruction[i-1].replace("$t","")  instruction[i-1]=instruction[i-1].replace("r","")  if((opcode(instruction[0])[i]) == 'REG\_D'):  #instruction[i-1]=instruction[i-1].replace("$t","")  binary\_instruction = binary\_instruction.replace('RD','{:0>5b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} RD'.format(i-1))  elif((opcode(instruction[0])[i]) == 'REG\_T'):  #instruction[i-1]=instruction[i-1].replace("$t","")  binary\_instruction = binary\_instruction.replace('RT','{:0>5b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} RT'.format(i-1))  elif((opcode(instruction[0])[i]) == 'REG\_S'):  #instruction[i-1]=instruction[i-1].replace("$t","")  binary\_instruction = binary\_instruction.replace('RS','{:0>5b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} RS'.format(i-1))  elif((opcode(instruction[0])[i]) == 'SHAMT'):  binary\_instruction = binary\_instruction.replace('SHAMT','{:0>5b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} SHAMT'.format(i-1))  elif((opcode(instruction[0])[i]) == 'OFFSET'):  binary\_instruction = binary\_instruction.replace('OFFSET','{:0>16b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} OFFSET'.format(i-1))  elif((opcode(instruction[0])[i]) == 'IMMEDIATE'):  binary\_instruction = binary\_instruction.replace('IMMEDIATE','{:0>16b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} IMMEDIATE'.format(i-1))  elif((opcode(instruction[0])[i]) == 'TARGET'):  binary\_instruction = binary\_instruction.replace('TARGET','{:0>26b}'.format(int(instruction[i-1])))  print(binary\_instruction + ' <- param{} TARGET'.format(i-1))  print(binary\_instruction)  binary\_instruction = binary\_instruction.replace("\_","")  print('Instruccion: {}\nLongitud: {}'.format(binary\_instruction, len(binary\_instruction)))  return binary\_instruction  except:  print("\nERROR - INSTRUCCION INVALIDA\n\n")  return -1  ################################## INTERACCION CON EL USUARIO #################################  print ('\t\t\t\tMIPS - UNIDAD DE DEBUGGING')  # Configuracion de la comunicacion serial  try:  ser = serial.Serial('/dev/ttyUSB1',19200,timeout=1)  print ('\t\tSerialPort: {} , BaudRate: {} , ByteSize: {}\n'.format(ser.name,ser.baudrate, ser.bytesize))  except:  print('ERROR - Asegurese de conectar el dispositivo ')  exit()  # Seleccion del modo de debugging  print('Debugging Mode: \n\t 1) Debug at the end of the program (default)\n\t 2) Debug for each clock')  debug\_mode = raw\_input("Select the number of the mode: ")  if (debug\_mode == '2'):  send("00000000000000000000000000000001")  else:  send("00000000000000000000000000000010")  # Se imprime el set de instrucciones soportado por el MIPS  print (' ------------------------------ INSTRUCTION SET ------------------------------\n')  print ('| \t\tNAME \t\t|\tINSTRUCTION \t|\tEXAMPLE \t|')  for i in range (1,35):  print ('| {}\t| {}\t| {}\t|'.format(instruction\_set(i)[0], instruction\_set(i)[1], instruction\_set(i)[2] ))  print (' -----------------------------------------------------------------------------')  # Seleccion del modo de carga del programa de usuario  print('Load Program Mode: \n\t 1) Load from file (default) \n\t 2) Manual Loading')  load\_mode = raw\_input("Select the number of the mode: ")  if (load\_mode != '2'):  print (' ---------------------------------- PROGRAM ----------------------------------\n')  try:  f = open ('TEST2.txt','r') #program\_addi  while True:  line = f.readline()  if not line:  print("LOAD SUCCESSFULLY")  break  line = line[:-1]  print('{:^30}'.format(line))  b\_instruction=decode(line)  send(b\_instruction)  time.sleep(1)  except:  print ("ERROR")  mode = 1  pass  f.close  send(opcode('halt')) #instruccion HALT  print ('\n -----------------------------------------------------------------------------')    if (load\_mode == '2'):  print("\n\t\tManual Loading")  while (1):  in\_comando = raw\_input("Write one instruction: ")  in\_comando = in\_comando.lower()  if (in\_comando == "exit"): # comando de salida  print("\n\tGOODBYE !!\n")  exit()  if (in\_comando == 'halt'):  send(opcode(in\_comando)) #instruccion HALT  break  b\_instruction=decode(in\_comando)  #print(b\_instruction)  send(b\_instruction)  print ("\n\nWait for debug data ...")  while 1:  # Si el modo de debug es en cada ciclo necesito enviar un dato para recibir insformacion.  if (debug\_mode == '2'):  var = raw\_input("Press a keyboard for debug ")  ser.write("1")  inline = ""  inline = ser.readline() # Lee hasta que se vacie el buffer.  #inline = ser.read(296) #Lee 296 bytes del buffer.  if (inline != ""):  #print (inline)  j=0  # Se separa el string recibido cada 4 caracteres (32bits), luego se obtiene el numero entero restandole 48 al valor  # del caracter ascii y se pone la etiqueta correspondiente al dato.  for i in range (4 , len(inline)+1 , 4):  reg = inline[i-4:i]  byte1 = ord(reg[0]) - 48 & 0xff  byte2 = ord(reg[1]) - 48 & 0xff  byte3 = ord(reg[2]) - 48 & 0xff  byte4 = ord(reg[3]) - 48 & 0xff  if (j == 0 ):  parseo(j,byte1,byte2,byte3,byte4)  elif (j <= 32 ):  if j==1:  print('\n\t\t\t REGISTROS')  print('Reg[{:0>2d}] : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(j-1,byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4))  elif (j <= 52 ):  if j==33:  print('\n\t\t\t MEMORIA')  print('Mem[{:0>2d}] : 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(j-33,byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4))  elif (j <= 73 ):  if j == 53:  print('\n\t\t\t LATCHS')  parseo(j,byte1,byte2,byte3,byte4)  else:  print('Default: 0x{:0>2x}{:0>2x}{:0>2x}{:0>2x} : 0b{:0>8b}\_{:0>8b}\_{:0>8b}\_{:0>8b}'.format(byte1,byte2,byte3,byte4,byte1,byte2,byte3,byte4))  j=j+1 |