

DOCUMENTO DE DISEÑO Y ARQUITECTURA

Fecha: 08/01/2018

Versión: 2

Responsables: Danilo Alejandro Ochoa Hidalgo

ÍNDICE

1 INSTRUCCIONES MIPS (R2000)	1
2 DISEÑO	5
3 EJECUCIÓN DEL PROGRAMA EN EL SIMULADOR	12
4 BIBLIOGRAFÍA	15

1 INSTRUCCIONES MIPS (R2000)

Microprocesador MIPS (*Microprocessor without Interlocked Pipeline Stages*, microprocesador sin bloqueos en las etapas de segmentación), basado en el diseño arquitectónico RISC (*Reduced Instruction Set Computer*, Computador con Conjunto de Instrucciones Reducidas).

Para poder abordar el tema sobre el microprocesador MIPS es necesario, primeramente, conocer la arquitectura empleada en el computador RISC. La arquitectura del computador RISC, desarrollada a mediados de los años 70, nace de la necesidad de encontrar alternativas para optimizar el uso de los recursos del computador que, a diferencia de la arquitectura empleada en el computador CISC (*Complex Instruction Set Computer*, Computador con Conjunto de Instrucciones Complejas), logró incorporar una estructura que aprovecha estos recursos de manera eficiente. Realizando un análisis a las características de la arquitectura RISC, podemos destacar:

- su conjunto de instrucciones de longitud fija, lo que quiere decir que todas sus instrucciones tienen la misma cantidad de bits, de esta forma, es más sencillo para el computador procesarlas. Aunque se necesitan más instrucciones para realizar la misma operación que en un computador CISC, las instrucciones simples y de fácil interpretación hacen que la fase de decodificación de las instrucciones sea ligera y veloz.
- La implementación de la tecnología de *registro-registro* incrementó el rendimiento del computador, en donde las operaciones de carga/almacenamiento son las únicas que interactúan con la memoria, mientras que las demás se manipulan mediante registros de propósito general reduciendo el número de accesos a memoria y, en consecuencia, aumentando la velocidad del computador. Las nuevas tecnologías incrementaron la capacidad de las memorias y, esto en conjunto con la inclusión de memorias caché para separar el código de los datos, contribuyeron en gran medida a mejorar la velocidad de procesamiento. Con el desarrollo de la tecnología de memorias implementada fue posible adjuntar la metodología denominada ejecución en paralelo (*pipelining*) que consiste en la ejecución de varias instrucciones concurrentemente.
- La arquitectura RISC cuenta con pocos modos de direccionamiento debido al mejoramiento en la implementación de la circuitería interna y externa.

El desarrollo del computador RISC y toda su investigación convergen en un dispositivo capaz de procesar información de manera eficaz dando cabida a múltiples aplicaciones que hasta la actualidad se están explotando. Por lo tanto, el microprocesador MIPS trabaja bajo circunstancias parecidas siendo uno de los procesadores desarrollados tomando en cuenta la estructura del computador RISC. (Universidad Nacional de la Plata, n.d.), (Universidad Politécnica De Madrid, n.d.).

Basándonos en la conceptualización del computador RISC, en este apartado, nos adentramos en el estudio del microprocesador MIPS. Este microprocesador fue desarrollado por MIPS Technologies originalmente fundada en 1984 por un grupo de investigadores, liderados por John L. Hennessy, de la Stanford University, conocida en ese entonces como MIPS Computer Systems Inc.

Aprovechando la gran cantidad de talentos disponibles en Silicon Valley, Hennessy seleccionó un increíble equipo de diseñadores de chips, ingenieros de software e investigadores de arquitectura informática con la esperanza de crear el mejor procesador RISC. Otros fundadores principales fueron Skip Stritter, anteriormente un tecnólogo de Motorola, y John Moussouris, que vino de IBM. (Voica, 2016).

Para aquella época ya era bien conocida la tecnología de procesamiento en paralelo (procesamiento simultáneo de instrucciones divididas en fases) pero su implementación era difícil, el cual era uno de sus principales objetivos. También buscaron optimizar la ejecución de instrucción por fase para que en cada fase se tenga la tendencia de tiempo de ejecución igual a un ciclo de reloj, evitando así, los conocidos bloqueos que se dan cuando una de las fases de cierta instrucción aún no se ha completado retrasando las demás fases o las nuevas ejecuciones. (Hernández Cerezo, Alonso Iglesias, & Tejedor García, n.d.).

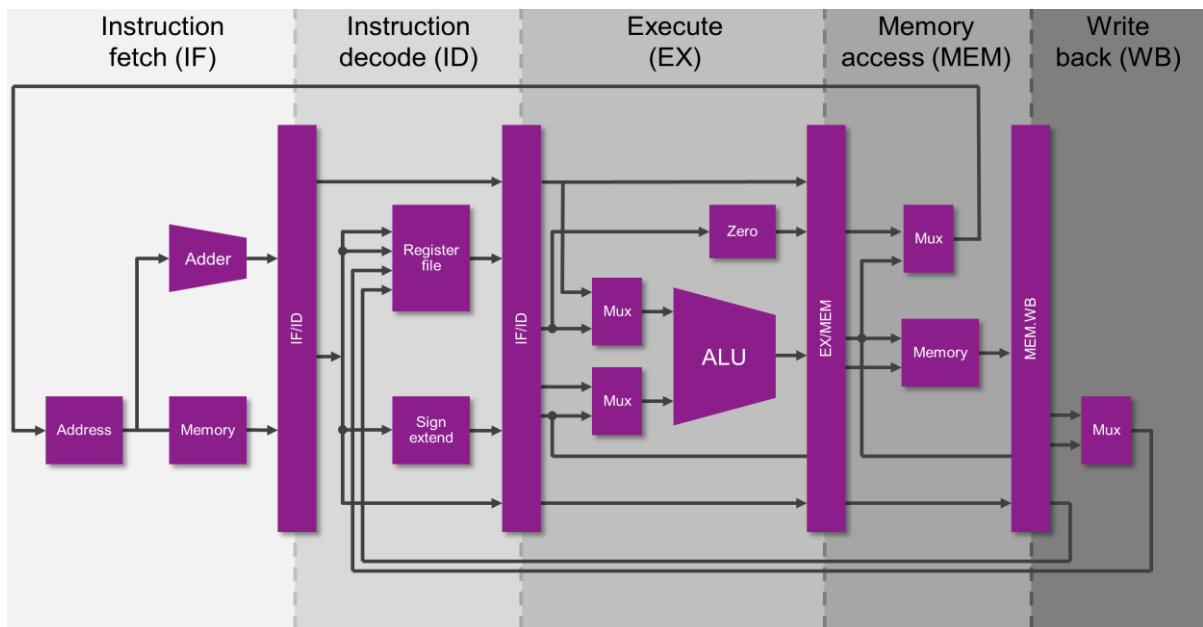
1.1 Presentación

1.1.1 Especificaciones de la estructura y organización

A continuación, se realiza el análisis del microprocesador MIPS r2000, dispositivo resultado de lograr implementar la tecnología antes mencionada. Fue el primer prototipo creado con excelentes resultados que, posteriormente, sería el primer procesador disponible comercialmente en 1986.

Analizando la estructura del MIPS r2000 podemos remarcar algunos puntos:

MIPS R2000 alcanza velocidades de hasta 15 MHz y mide 80 mm² en área de silicio; el procesador contenía aproximadamente 110,000 transistores dispuestos usando un nodo de proceso CMOS de doble metal de 2,0 µm. Para poner esto en perspectiva, una CPU basada en MIPS fabricado en 2015 usando un proceso de 28nm puede incluir de 24 a 48 de alta frecuencia, los núcleos superescalares que funcionan a hasta 2,5 GHz, grandes y altamente asociativos cachés L1 y L2, y enorme ancho de banda DRAM, que representa un aumento increíble en la velocidad de frecuencia y una merma notable en los procesos de fabricación de semiconductores. (...) Una de las principales características nuevas del chip R2000 fue el rápido tiempo de ejecución en ausencia de errores de caché; entregó una impresionante tasa de finalización de instrucción de una instrucción por ciclo ALU en una era donde los microprocesadores no RISC necesitaban varios ciclos por instrucción. El siguiente diagrama de bloques presenta el diseño de tuberías de cinco etapas que se convertiría en el punto de referencia para otros procesadores RISC (Voica, 2016):



El diseño de tubería de cinco etapas de una CPU MIPS RISC

Fuente: (Voica, 2016)

1.1.2 Especificaciones de la arquitectura

Todas las instrucciones del MIPS R2000 tienen el mismo tamaño (32 bits). Se pueden clasificar en función de los elementos que utilizan (banco de registros, memoria de datos, ALU). Cada uno de los componentes que utiliza la instrucción se debe especificar en una serie de bits. Los distintos tipos de

instrucciones constan de diferentes tamaños para los espacios reservados para esos bits (campos) es decir, utilizan diferentes formatos para codificar sus campos. (Hernández Cerezo et al., n.d., p. 10).

Los campos siguen una distribución como la que sigue (tabla 1):

- op: Operación básica de la instrucción. También es llamada opcode (código de operación).
- rs: El primer registro fuente de operandos.
- rt: Segundo registro fuente de operandos.
- rd: Registro de destino de los operandos. Se le asigna el resultado de la operación.
- shamt: Cantidad de desplazamiento. Indica el número de bits a desplazar en una instrucción de desplazamiento (Shift amount).
- funct: Función. Este campo selecciona la operación específica del campo op (function code).
- Constante o dirección: Constante no más grande de $\pm 2^{15}$, o dirección dentro de una región de $\pm 2^{15}$.

(Gómez Luis, 2010, p. 8).

Tabla 1

Tipo de instrucciones en MIPS

Nombre	Campos						Observaciones
Tamaño del campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas las instrucciones MIPS son de 32 bits
Tipo-R (Registro)	Op	rs	rt	rd	shamt	funct	Formato de instrucción aritmética
Tipo-I (Inmediato)	Op	rs	rt	Constante o dirección			Transferencia, saltos, inmediatas
Tipo-J (<i>Jump</i> , Salto)	Op	Dirección de destino					Formato de instrucción de salto

Fuente: (Gómez Luis, 2010, p. 9).

Este microprocesador cuenta con varias instrucciones para diferentes operaciones:

Lenguaje ensamblador MIPS		
Categoría	Instrucción	Observaciones
Aritmética	add (add)	Tres registros operandos
	subtract (sub)	Tres registros operandos
	add immediate (addi)	Sumar constantes
Transferencia de datos	load word (lw)	Palabra de la memoria a un registro
	store word (sw)	Palabra de un registro a la memoria
	load half (lh)	Media palabra de la memoria a un registro
	store half (sh)	Media palabra de un registro a la memoria
	load byte (lb)	Byte de la memoria a un registro
	store byte (sb)	Byte de un registro a la memoria
	load upper immediate (lui)	Cargar constante superior a 16 bits
Lógica	AND (and)	Tres registros operandos. Bit-by-bit. AND
	OR (or)	Tres registros operandos. Bit-by-bit. OR
	NOR (nor)	Tres registros operandos. Bit-by-bit. NOR
	AND immediate (andi)	Bit-by-bit AND con un registro constante.
	OR immediate (ori)	Bit-by-bit OR con un registro constante.
	shift left logical (sll)	Shift left por constante
	shift right logical (srl)	Shift right por constante
Saltos condicionales	branch on equal (beq)	Compara si es igual que.
	branch on not equal (bne)	Compara si no es igual que.
	set on less than (slt)	Compara menor que.
	set less than immediate (slti)	Compara menor que una constante
Saltos incondicionales	jump (j)	Salta a la dirección apuntada
	jump register (jr)	Salta al registro indicado
	jump and link (jal)	Para un procedimiento de llamada.

Fuente: (Gómez Luis, 2010)

Para la manipulación de los datos que se especifican en las instrucciones se utiliza una memoria con varios registros en un único componente denominado *Register File* (Tabla 2):

- 32 registros de 32 bits de propósito general para operaciones con enteros (GPRs (r0-r31), r0 tiene siempre valor 0, intentar cambiarlo no da error, pero su valor nunca cambia). Éstos se identifican por el carácter especial \$ seguido de un número de 0 a 31. Así por ejemplo el registro 1 se reconocerá cuando se vea la representación \$1.
- Los registros preservados durante una llamada son aquellos que (por convenio) no serán modificados por una llamada de sistema o a un procedimiento o función. Por ejemplo, los registros \$s deben ser almacenados en la pila por el procedimiento que los necesita, siendo siempre incrementados en constantes \$sp y \$fp, para ser después decrementados una vez finalizado el procedimiento (se marca como disponible la memoria reservada). Por el contrario, \$ra es modificado automáticamente tras una llamada a una función normal (cualquiera que utilice la instrucción jal), y los registros \$t deben ser salvados por el programa antes de llamar a cualquier función (por si el programa necesita los valores contenidos en dichos registros tras la ejecución de la subrutina)

(Hernández Cerezo, Alonso Iglesias, & Tejedor García, n.d.)

Tabla 2

Tipos de registros del componente Register File

Nombre de registro	Número	Uso
Zero	0	Constante 0
At	1	Reservado para el assembler
V0	2	Para evaluación de expresiones y retorno de resultados de una función
V1	3	
A0	4	Argumento 1
A1	5	Argumento 2
A2	6	Argumento 3
A3	7	Argumento 4
T0	8	Temporal (no se preserva a través de los llamados)
T1	9	Temporal (no se preserva a través de los llamados)
T2	10	Temporal (no se preserva a través de los llamados)
T3	11	Temporal (no se preserva a través de los llamados)
T4	12	Temporal (no se preserva a través de los llamados)
T5	13	Temporal (no se preserva a través de los llamados)
T6	14	Temporal (no se preserva a través de los llamados)
T7	15	Temporal (no se preserva a través de los llamados)
S0	16	Temporal que debe preservarse entre llamados a funciones
S1	17	Temporal que debe preservarse entre llamados a funciones
S2	18	Temporal que debe preservarse entre llamados a funciones
S3	19	Temporal que debe preservarse entre llamados a funciones
S4	20	Temporal que debe preservarse entre llamados a funciones
S5	21	Temporal que debe preservarse entre llamados a funciones
S6	22	Temporal que debe preservarse entre llamados a funciones
S7	23	Temporal que debe preservarse entre llamados a funciones
T8	24	Temporal (no se preserva a través de los llamados)
T9	25	Temporal (no se preserva a través de los llamados)
K0	26	Reservado para el núcleo del SO
K1	27	Reservado para el núcleo del SO
Gp	28	Puntero al área global de datos
Sp	29	Puntero al tope de la pila. StackPointer
Fp	30	Puntero a zona de variables en la pila. FramePointer
Ra	31	Dirección de retorno (usado en invocaciones a funciones). ReturnAddress

1.2 Justificación

El MIPS r2000 es un microprocesador de propósito general con características que lo hacen excelente para el estudio de arquitectura de computadores. No es algo de lo más básico, ni tampoco algo sumamente complejo. Además, esta arquitectura sigue siendo utilizada en muchos de los dispositivos actuales. Se puede mencionar también que se lo eligió debido a que la mayoría de las instrucciones no son captadas directamente desde memoria; además, está dotado de un gran rendimiento y bajo consumo de energía con respecto a computadores de su época. Las únicas instrucciones que interactúan de manera directa con la memoria son las de carga y almacenamiento. Este microprocesador es capaz de realizar operaciones complejas basándose en instrucciones simples de interpretar. Estas instrucciones tienen una longitud fija por lo que la fase de ejecución se torna más eficiente. La capacidad de almacenamiento con la que cuenta hace de su operar algo más fácil de llevar a cabo. Permite ejecutar instrucciones repetitivas para facilitar la programación cuando se necesita realizar bucles. Y también, se puede analizar el procesamiento en paralelo con el que cuenta y el cual le permite ejecutar varias instrucciones en un único ciclo de reloj por lo que, aunque la latencia (tiempo de ejecución de una instrucción) sea la misma por instrucción, el tiempo de ejecución total se ve reducido eficientemente.

2 DISEÑO

El programa empleado para ejecutarse en el simulador se presenta a continuación. Elaborado para lenguaje ensamblador (las instrucciones del sistema se omiten) (Tabla 3):

Tabla 3

Código ensamblador del programa para la simulación

```

1  .data
2      # Salto de Línea para impresión en filas.
3      salto: .asciiz "\n"
4  .text
5      main:
6          # $t0 <- 0 + 0
7          addi $t0, $zero, 0
8      loop:
9          # Si $t0 > 10 entonces >- salto al bloque -> exit:
10         bgt $t0, 10, exit
11         # Función del sistema para presentar un entero por consola
12         li $v0, 1
13         add $a0, $zero, $t0
14         syscall
15         # Función del sistema para presentar una cadena por consola
16         li $v0, 4
17         la $a0, salto
18         syscall
19         # $t0 <- $t0 + 1
20         addi $t0, $t0, 1
21         # Retorno al bloque loop:
22         j loop
23     exit:
24         # Función del sistema para terminar programa
25         li $v0, 10
26         syscall

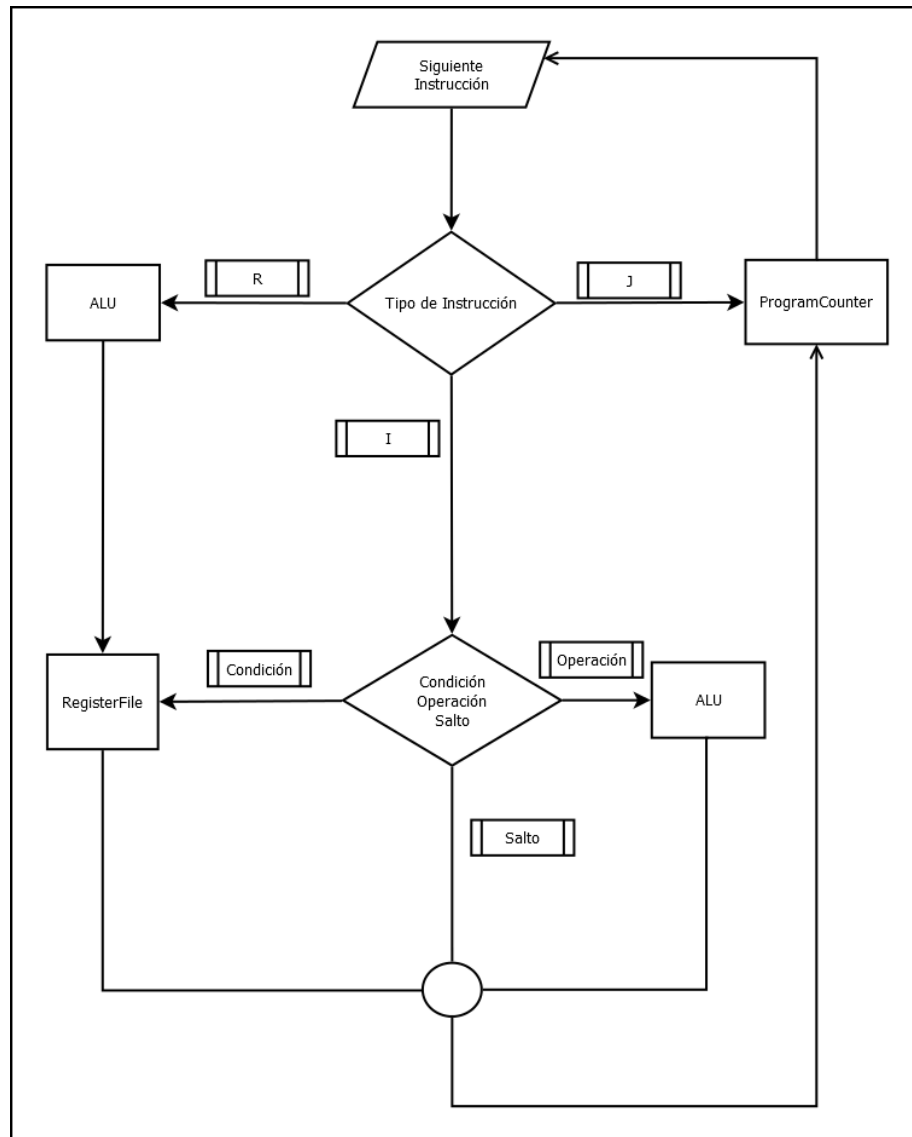
```

- El código empieza en la sección .text con el bloque main: (línea 5), en donde se tiene una suma con instrucción de tipo inmediato, por lo tanto, los campos a ocupar son op, rs, rt y la constante (en este caso). Esto con la finalidad de usar el registro \$t0 (\$8), que según las disposiciones convencionales sirve como un registro temporal.
- En el bloque loop: (línea 8) se tiene una primera instrucción condicional (línea 10) de tipo inmediato. La instrucción realiza la comparación de \$t0, el contenido del registro temporal 0, con el número diez, y si el resultado es verdadero se realiza un salto al bloque exit: (línea 23) en donde el programa termina, de lo contrario, el flujo del programa continua con la siguiente instrucción.

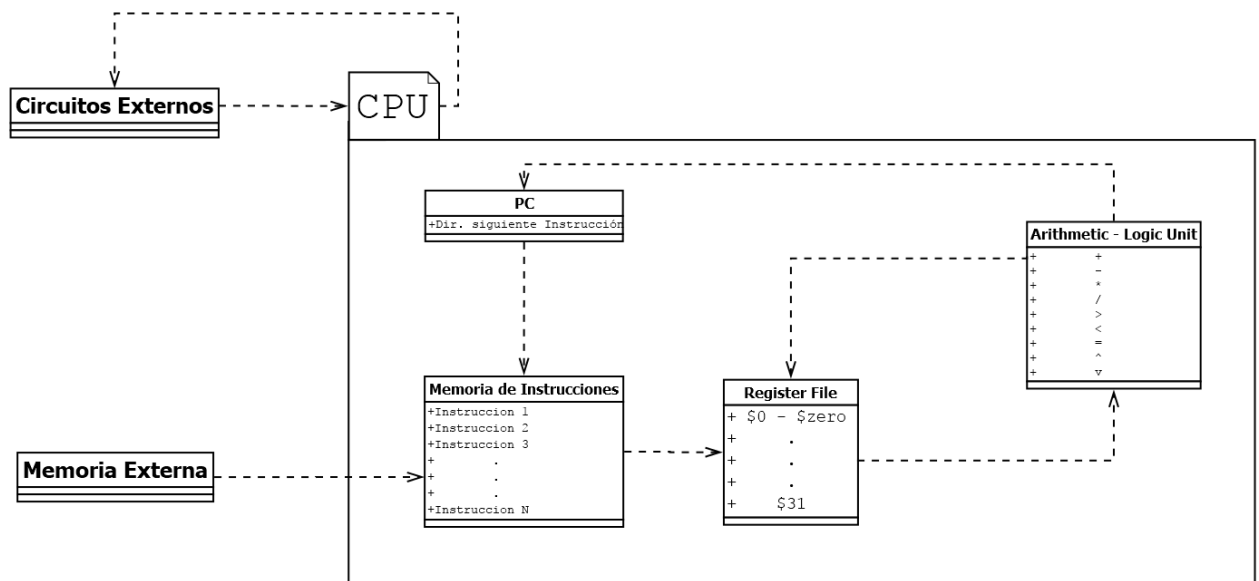
- La siguiente instrucción (línea 20) es del tipo inmediato, la cual realiza una suma entre el registro \$t0 y una constante numérica para luego almacenar el resultado en el mismo registro, \$t0.
- A continuación, se tiene una instrucción de tipo salto incondicional (línea 22) que ocupa los campos op, y la dirección de destino. La instrucción manipula el flujo del programa haciendo que la próxima ejecución sea la primera instrucción del bloque loop: (línea 8).
- El bloque loop: se repite hasta que la condición de la instrucción de la línea 10 sea verdadera en donde realiza el salto al final del programa terminando con la ejecución del mismo.

2.1 Diagrama del Sistema

En el siguiente diagrama se observan los componentes con los que interactúan los distintos tipos de instrucciones para el programa planteado.



En le diagrama a continuación, se observa la interacción de los componentes del procesador para el programa planteado.



Distribución de los componentes:

- Circuitos externos al CPU que interactúan con el mismo.
- Se debe tomar en cuenta el disco duro o memoria externa que se usa para cargar las instrucciones en la memoria de instrucciones.
- Dentro del CPU:
 - El *Program Counter* es el componente que controla el flujo de la ejecución señalando o guardando la dirección de la próxima instrucción que debe ejecutarse.
 - La Memoria de Instrucciones almacena las instrucciones que se van a ejecutar y la dirección de memoria o posición en la que se encuentran.
 - El componente *Register File* es una memoria pequeña pero veloz que permite almacenar los datos que se encuentran en las posiciones de memoria señaladas en las instrucciones, y/o para almacenar los resultados obtenidos del componente *Arithmetic Logic Unit*. Como se señaló anteriormente, este componente cuenta con un uso convencional para cada posición de memoria.
 - La *Arithmetic Logic Unit* permite realizar operaciones aritméticas, lógicas y de comparación. Este componente interactúa con el *Register File* para realizar las operaciones especificadas en las instrucciones y puede ser usado para calcular posiciones de memoria, al interactuar con el *Program Counter*, permitiendo manipular el flujo del programa facilitando bucles y saltos.

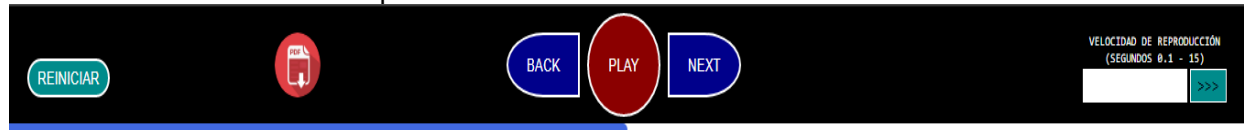
2.2 Diagrama de despliegue


En este apartado se presenta el desarrollo del simulador.

A continuación, se describen las secciones en las que se distribuyeron las funciones del procesador:



2.2.1 Barra de Funcionalidad:

La barra de funcionalidad sirve para controlar el simulador:





1. El botón  sirve para restablecer el simulador al ajuste inicial. Se puede controlar con el botón (R) del teclado o presionando en el mismo. Dependiendo de la velocidad que se encuentre en el campo de control de velocidad este adoptará el mismo, o si no existe valor alguno en dicho campo, el simulador tomara una velocidad de 2s.

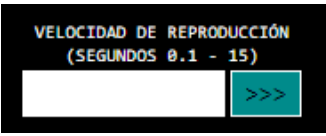
2. El apartado con el ícono  es un enlace que lleva a la documentación del simulador.

3. Los botones  y  sirven para controlar paso por paso la simulación permitiendo ver cómo se desenvuelve la misma. Se pueden controlar con las flechas del teclado o presionando en los mismos.

Cuando el botón cambia al color negro , significa que ha sido desactivado y no permite esta acción.

4. El botón  permite ejecutar el simulador de manera automática. La velocidad de simulación por defecto es de 2s pero se puede cambiar en el campo de velocidad de ejecución. Esta acción está limitada a una sola vez y se puede reactivar cuando se reinicia la simulación.

Cuando el botón está activo se torna de esta forma  en donde los botones de paso por paso se encuentran desactivados.

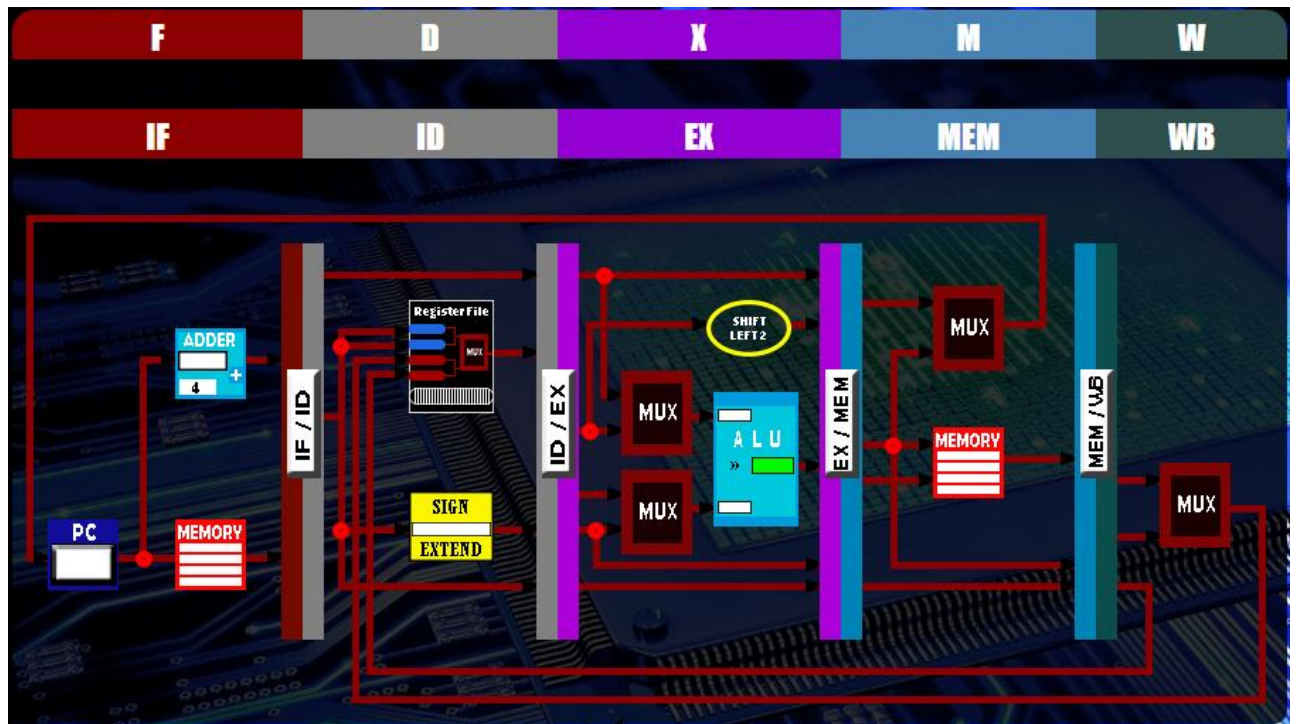
5. El apartado para controlar la velocidad de ejecución , permite cambiar el valor de la velocidad de reproducción de la simulación. Este valor se debe expresar en segundos y se permite un rango de 0.1 segundos hasta 15 segundos por instrucción.

6. La barra inferior sirve como contador de los pasos de la simulación. Son 88 pasos en total.

2.2.2 Interfaz:

Primeramente, se tiene un apartado **Interfaz** en donde se encuentra el diagrama del procesador y una tabla con las instrucciones que se van a ejecutar en el simulador.





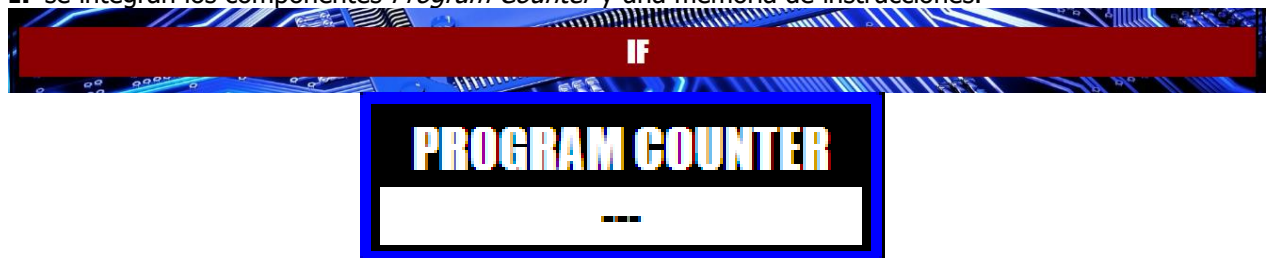
El diagrama del procesador tiene el propósito de servir de guía para comprender cómo se comporta el flujo del programa.

BLOQUE	INSTRUCCIONES				DESCRIPCIÓN
Main	addi	\$t0	\$zero	0	# \$t0 = 0
Loop	bgt	\$t0	10	exit	# si \$t0 > 10 entonces >- salto al bloque -> exit:
	addi	\$t0	\$t0	1	# \$t0 <- \$t0 + 1
	j	loop			# salto al bloque loop:
Exit	fin	programa			# final del programa

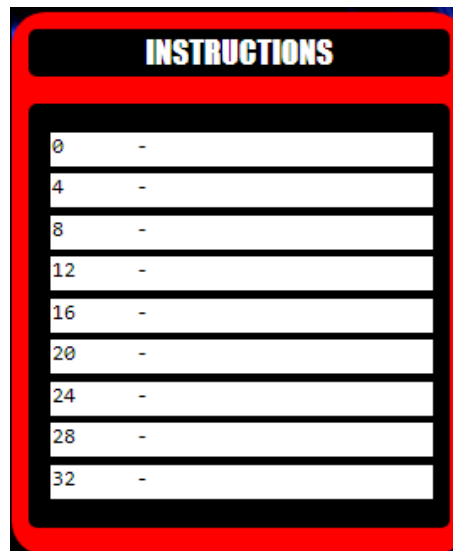
La tabla de instrucciones sirve para visualizar el código que se va a ejecutar en el simulador. El código, como se menciona antes, fue escrito para el lenguaje ensamblador del procesador.

2.2.3 Instruction Fetch (IF):

En esta sección es en donde se realiza la captura de la instrucción para ser decodificada. En la sección **IF** se integran los componentes *Program Counter* y una memoria de instrucciones.



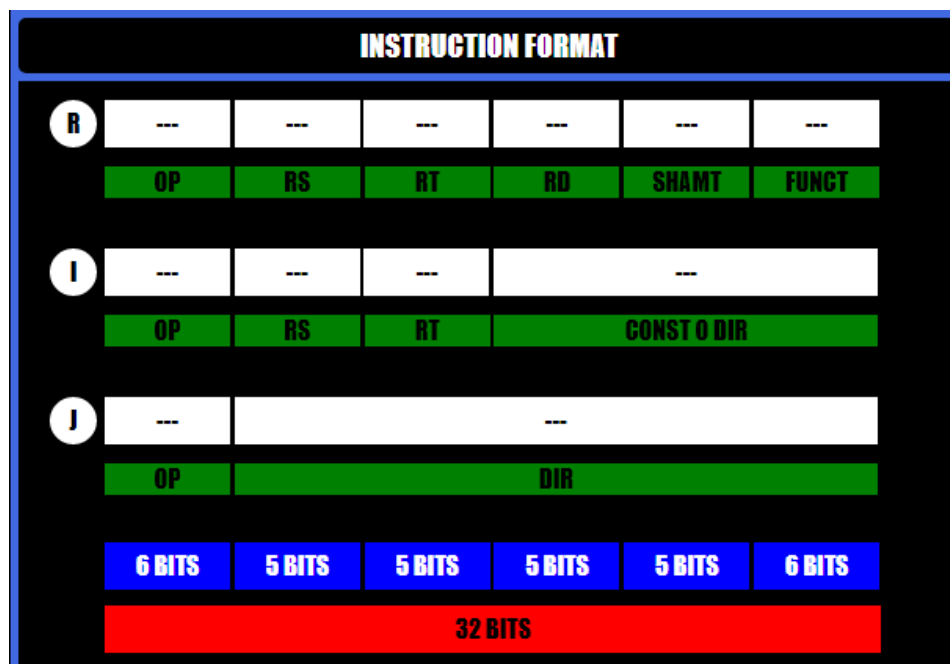
El *Program Counter* sirve para almacenar la dirección de memoria de la siguiente instrucción que debe ejecutarse.



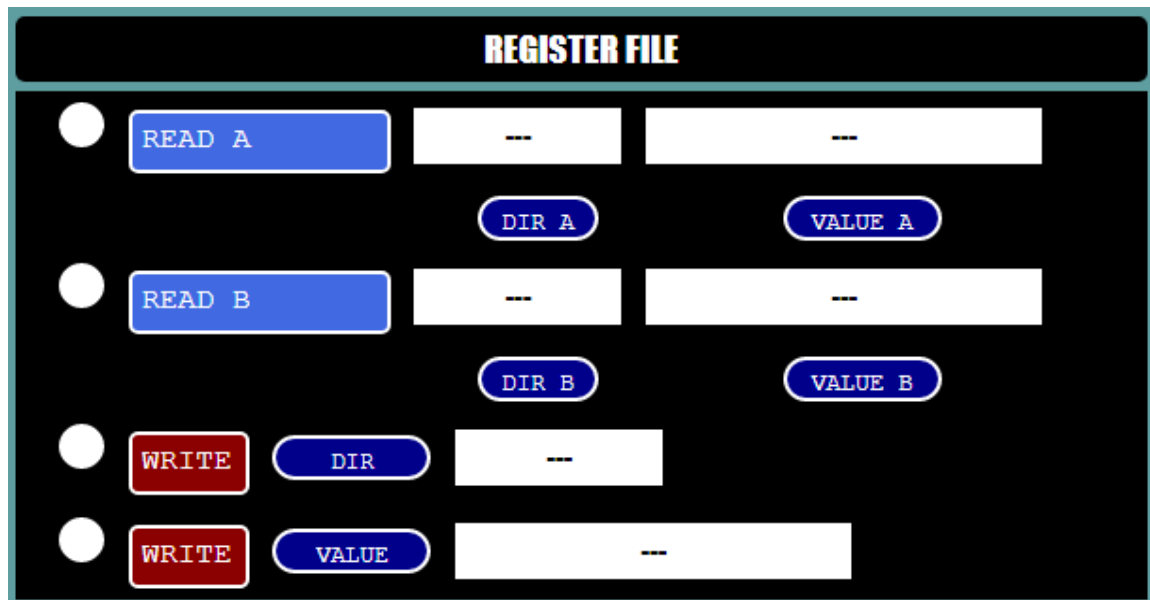
La memoria de instrucciones almacena las instrucciones a ejecutarse, las cuales deben ser previamente almacenadas en esta desde una memoria externa o el disco duro.

2.2.4 Instruction Decode (ID):

En la sección **ID** se realiza la decodificación de la instrucción para conocer el tipo de la instrucción y preparar los datos necesarios para realizar las operaciones especificadas en la misma instrucción. En esta sección se encuentran, primero: una tabla con el formato o tipo de la instrucción; y, en segundo lugar, un componente *Register File*.



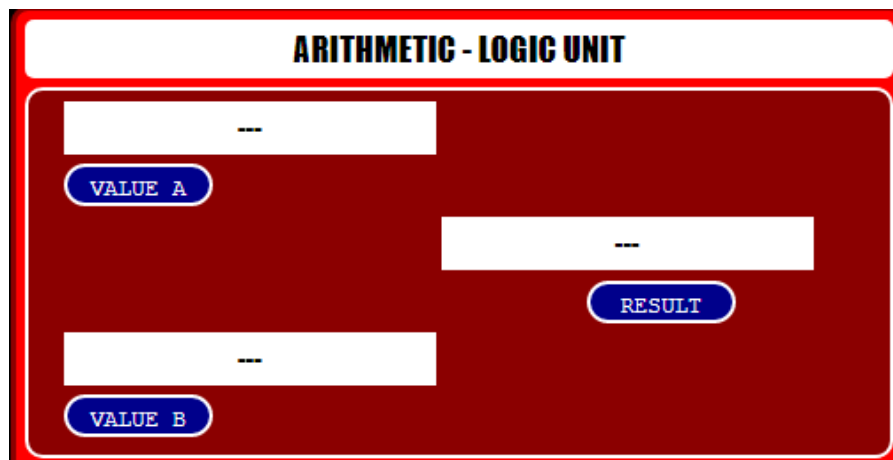
La tabla *Instruction Format* indica el tipo de instrucción a ejecutarse, así como los campos que deben emplearse para cada tipo de instrucción. Adicionalmente, en la parte inferior puede verse la cantidad de bits que componen los formatos de instrucciones, que como se analizó previamente, todas las instrucciones constan de 32 bits.



El componente *Register File* indica; cuando los círculos de la parte izquierda están de color celeste, se está realizando una lectura; si los círculos son rojo vino, el componente está realizando una escritura. La escritura se realiza a uno de los 32 registros de propósito general que contiene el mismo.

2.2.5 Execution (EX):

La sección **EX** utiliza los datos obtenidos en la anterior sección para realizar las respectivas operaciones determinadas en las mismas instrucciones. Aquí, únicamente se tiene el componente *Arithmetic Logic Unit*.



La *Arithmetic Logic Unit* permite realizar operaciones aritméticas (+, -, *, /), lógicas (and, or) y de comparación (>, <, =) con sus respectivas operaciones derivadas como resultado de la combinación de las mismas (>=, <=, etc).

3 EJECUCIÓN DEL PROGRAMA EN EL SIMULADOR

Las instrucciones tienen diferentes formatos y por eso este apartado se dedica a la explicación del flujo que toman las distintas instrucciones, omitiendo las instrucciones del tipo R ya que el programa no cuenta con ninguna de este tipo.

3.1.1 Addi (Add Immediate):

Tomando como ejemplo la instrucción `addi (op) $t0 (rs, primer operando), $t0 (destino), 1 (segundo operando)`. La **adición inmediata** utiliza los campos op, rs, rt y constante o dirección. Para decodificar la instrucción, basándose en la hoja de referencia MIPS (*MIPS Reference Sheet* o *MIPS Reference Card*), se obtiene que:

- El op representa el código de operación y el equivalente a la operación addi es 001000.
- El campo rt representa el destino y dependiendo de la distribución del *Register File* se accede a la dirección especificada para obtener la posición de memoria en donde se va a almacenar el resultado, por ejemplo 01000 equivale a \$8 el registro temporal 0 (\$t0).
- Los campos rs y constante representan los valores con los que se va a realizar la operación de adición, en donde el valor de rs debe ser tomado del *Register File* por lo que se accede a la dirección especificada para obtener el valor de esa posición de memoria, por ejemplo 01000 equivale a \$8 el registro temporal 0 (\$t0) y se debe acceder a esa posición de memoria para acarrear el valor que tiene.
- Luego de tener los datos para realizar la suma, estos son pasados a la ALU que se encarga de sumar los datos para luego pasar el resultado al destino especificado en el *Register File*.

3.1.2 BGT (Branch Greater Than):

Tomando como ejemplo la instrucción `bgt (op) $t0 (primer operando), 10 (segundo operando), exit (destino)`, y que se considera que la instrucción actual se encuentra en la posición de memoria número 4. La **comparación mayor que (salto condicional)** utiliza los campos op rs rt y constante o dirección. Para decodificar la instrucción se obtiene:

- El op tiene su equivalente a la operación de bgt igual a 000111.
- El campo rs y el campo rt son los dos operandos que deben compararse para obtener un resultado. Para el campo rs se debe acceder al *Register File* para encontrar la dirección especificada y obtener el valor de esa posición de memoria, por ejemplo 01000 equivale a \$8 el registro temporal 0 (\$t0) y se debe acceder a esa posición de memoria para extraer el valor que contiene. El segundo operando rt, puede ser una posición de memoria del *Register File* o puede ser un valor inmediato el cual va a ser comparado con rs.
- En la ALU se obtienen los valores y se realiza la comparación. En caso de que la condición sea igual a *false* el programa continúa con la siguiente instrucción sin realizar el salto.
- El campo de constante o dirección se ocupa en caso de que la condición sea igual a *true* y el programa realiza una operación con la constante para calcular la siguiente dirección. La operación se explica a continuación (de ser el caso en que $\$t0 > 10 = \text{true}$):

1. El valor inmediato pasa por el componente *Sign Extend* para completar los 32 bits, recordando que todas las instrucciones del procesador son de 32 bits. Se debe tomar en cuenta que los 16 bits de la constante son un desplazamiento relativo (2 en este caso).

Considerando el valor de la constante como:

0000 0000 0000 0010 = 2 (16 bits)

Valor con signo extendido:

0000 0000 0000 0000 0000 0000 0010 = 2 (32 bits)

2. El valor con signo extendido pasa por el componente *Shift Left 2* el cual realiza un desplazamiento a la izquierda de toda la instrucción, lo que es equivalente a multiplicar por cuatro. Esto se realiza debido a que las instrucciones son múltiplos de cuatro por lo que los 2 bits menos significativos (LSB) siempre termina en cero. Por lo tanto, la forma más eficiente de implementar un salto era tomando en cuenta esa condición.

Valor con signo extendido:

0000 0000 0000 0000 0000 0000 0000 0010 = 2 (32 bits)

Valor después del desplazamiento:

0000 0000 0000 0000 0000 0000 0000 1000 = 8 (32 bits con desplazamiento)

3. Además, se suma 4 al resultado anterior debido a que la organización en bytes siempre es de 4 bytes, por lo que para ir a la siguiente instrucción siempre se suma 4.

Valor desplazado:

0000 0000 0000 0000 0000 0000 0000 1000 = 8 (32 bits con desplazamiento)

Valor luego la suma:

0000 0000 0000 0000 0000 0000 0000 1100 = 12 (instrucción de 32 bits)

4. Y, por último (o primero), se agrega el valor de la posición actual de la instrucción (4 en este caso).

Valor suma:

0000 0000 0000 0000 0000 0000 0000 1100 = 12 (instrucción de 32 bits)

Valor luego de sumar la posición actual:

0000 0000 0000 0000 0000 0000 0001 0000 = 16 (instrucción de 32 bits)

5. Como resultado se obtiene el valor de 16 por lo que el programa realizó un salto de la instrucción 4 a la instrucción 16. En conclusión, el programa al realizar la comparación de los campos rs con rt y obtener el valor *true*, basándose en la constante, que es un valor relativo de desplazamiento, realiza un salto de 4 instrucciones (para este caso), especificado en la misma constante.

En resumen:

0000 0000 0000 0000 0000 0000 0000 0100 = 4 (posición actual)

0000 0000 0000 0000 0000 0000 0000 0100 = 4 (suma que siempre se realiza)

+ 0000 0000 0000 0000 0000 0000 0000 1000 = 8 (constante con desplazamiento)

0000 0000 0000 0000 0000 0000 0001 0000 = 16 (instrucción de 32 bits)

3.1.3 J (Jump):

Tomando como ejemplo la instrucción `j loop`, tomando en cuenta que la instrucción actual se encuentra en la posición de memoria número 12. La instrucción de **salto (salto incondicional)** utiliza los campos op y constante o dirección. Para decodificar la instrucción se obtiene:

- El op tiene su equivalente a la operación de j igual a 000010.
- El campo de constante o dirección se ocupa para calcular la siguiente dirección. La operación se explica a continuación:

1. Se añaden 2 bits menos significativos (LSB) con valor de cero.

Considerando el valor de la constante como:

0000 0000 0000 0000 0000 0000 01 = 1 (26 bits)

Valor después del desplazamiento:

0000 0000 0000 0000 0000 0000 0100 = 4 (28 bits con desplazamiento)

2. Al valor obtenido se le añaden los 4 bits más significativos (MSB) de la instrucción actual.

Valor después del desplazamiento:

0000 0000 0000 0000 0000 0000 0100 = 4 (28 bits con desplazamiento)

Valor luego de sumar los 4 bits más significativos de la posición actual:

0000 0000 0000 0000 0000 0000 0100 = 4 (32 bits con desplazamiento)

3. Como resultado se obtiene el valor de 4 por lo que el programa realizó un salto de la instrucción 12 a la instrucción 4. En conclusión, el programa calcula la dirección de la instrucción a la que debe apuntar basándose en la constante.

En resumen:

4MSB

0000 0000 0000 0000 0000 0000 1100 = 12 (posición actual)

+ 0000 0000 0000 0000 0000 0000 01 = 4 (constante con desplazamiento)

00 = *Shift Left 2*

0000 0000 0000 0000 0000 0000 0100 = 4 (instrucción de 32 bits)

4 BIBLIOGRAFÍA

- Gómez Luis, A. I. (2010). Diseño e implementación en un FPGA de un microprocesador basado en la arquitectura MIPS de un solo ciclo, 119. Retrieved from http://tesis.ipn.mx/bitstream/handle/123456789/18534/tesis_AIGL.pdf?sequence=1
- Hernández Cerezo, A., Alonso Iglesias, R., & Tejedor García, C. (n.d.). Arquitectura MIPS, 12. Retrieved from https://www.infor.uva.es/~bastida/OC/TRABAJO1_MIPS.pdf
- Universidad Nacional de la Plata. (n.d.). Arquitectura RISC, 8. Retrieved from http://electro.fisica.unlp.edu.ar/arq/transparencias/ARQII_03-RISC_A4x6.pdf
- Universidad Politécnica De Madrid. (n.d.). Arquitectura de Computadores, 12. Retrieved from http://www.dia.eui.upm.es/asignatu/arq_com/Paco/8-RISC.pdf
- Voica, A. (2016). It's been 30 years since the launch of MIPS R2000. Retrieved from <https://www.mips.com/blog/thirty-years-of-mips-r2000/>