

**Investigación**

# Procesador MIPS de 32 bits

El procesador MIPS (Microprocessor without Interlocked Pipeline Stages) de 32 bits es una arquitectura de procesador basada en RISC (Reduced Instruction Set Computing). A continuación se detallan algunos de sus elementos y características generales:

1. Unidad de control (Control Unit): es la parte del procesador que se encarga de controlar el flujo de datos y de instrucciones a través de las diferentes unidades del procesador.
2. Unidad aritmético-lógica (Arithmetic Logic Unit, ALU): es la unidad encargada de realizar las operaciones aritméticas y lógicas básicas, como sumar, restar, multiplicar, dividir, comparar y realizar operaciones booleanas
3. Arquitectura RISC: El procesador MIPS utiliza una arquitectura RISC (Reduced Instruction Set Computing) que se caracteriza por tener un conjunto de instrucciones reducido y simple. Esto permite que el procesador pueda ejecutar instrucciones de manera más rápida y eficiente
4. Memoria caché: el procesador MIPS utiliza una memoria caché para acelerar el acceso a los datos almacenados en memoria. La memoria caché es una memoria de acceso rápido que almacena copias de los datos más utilizados en la memoria principal.
5. FPU (Floating Point Unit): es la unidad encargada de realizar operaciones aritméticas en punto flotante, como sumas, restas, multiplicaciones y divisiones.
6. Arquitectura de canalización: El procesador MIPS utiliza una arquitectura de canalización para aumentar la eficiencia en la ejecución de instrucciones. La canalización permite que varias instrucciones se ejecuten al mismo tiempo, 1 reduciendo el tiempo de espera entre instrucciones.
7. Registro de propósito general: El procesador MIPS tiene 32 registros de propósito general de 32 bits, que se utilizan para almacenar datos y direcciones de memoria.

# Set de instrucciones

INSTRUCCION ADD: La instrucción "add" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "add" se utiliza para sumar dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION SUB: La instrucción "sub" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "sub" se utiliza para restar dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION OR: La instrucción "or" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "or" se utiliza para realizar una operación lógica OR a nivel de bit entre dos operandos y almacenar el resultado en un registro de destino

INSTRUCCION AND: La instrucción "and" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "and" se utiliza para realizar una operación lógica AND a nivel de bit entre dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION SLT: La instrucción "slt" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "slt" se utiliza para comparar dos valores y almacenar el resultado de la comparación en un registro de destino.

INSTRUCCION NOP: La instrucción "nop" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "nop" (abreviatura de "no operation" o "sin operación" en español) se utiliza para indicar que no se realizará ninguna operación en la ejecución de una instrucción y se utiliza comúnmente como un espacio reservado para futuras expansiones del programa

# Personajes

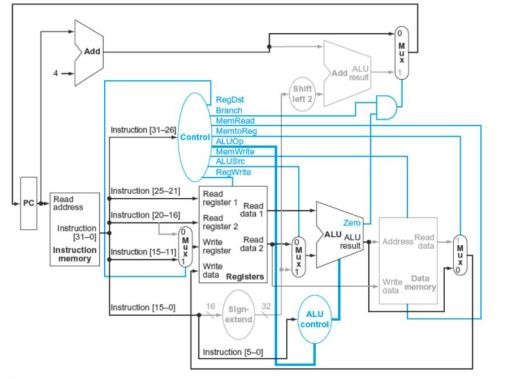
LJUBISA BAJIC: Ljubisa Bajic es un ingeniero en computación que ha trabajado en el diseño de microprocesadores durante más de 30 años. Una de sus contribuciones más importantes en microarquitectura ha sido en la técnica de pipeline en paralelo, que permite que el procesador ejecute múltiples instrucciones simultáneamente. Bajic también ha trabajado en la mejora de la eficiencia energética de los procesadores y ha desarrollado técnicas para reducir el consumo de energía de los circuitos de procesamiento de datos.

JIM KELLER: Jim Keller es otro ingeniero en computación que ha sido fundamental en la evolución de los microprocesadores. Keller ha trabajado en el diseño de procesadores de alta gama durante más de 20 años y ha liderado equipos de diseño en empresas como AMD, Apple e Intel. Una de sus contribuciones más importantes ha sido en la arquitectura de los procesadores x86 de 64 bits, que se utilizan en la mayoría de los ordenadores personales modernos. Keller también ha trabajado en la optimización de la eficiencia energética de los procesadores, la mejora de la velocidad de reloj y la reducción del tamaño de los transistores en los circuitos integrados

**Desarrollo**

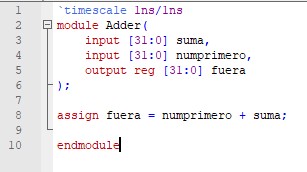
Se implementaron módulos correspondientes a la siguiente imagen:

F

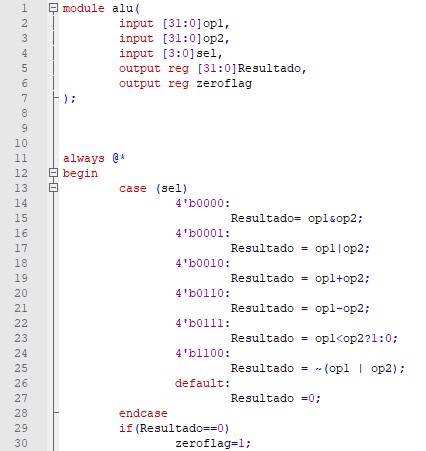


En código se ve de la siguiente manera:

Adder:



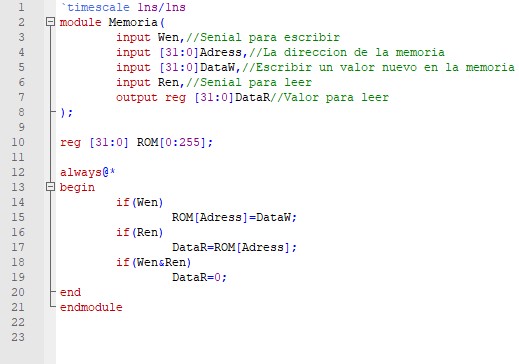
ALU:



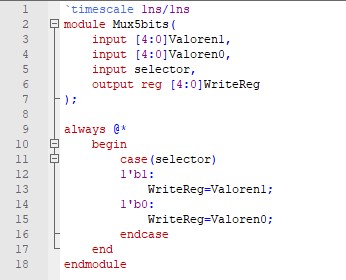
Alu control:



Memoria:



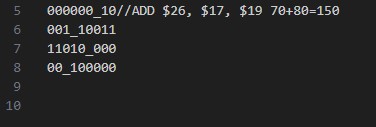
Mux 5 bits:



Y así se creó modulo por modulo hasta que cada uno de los módulos se instancio en un TestBench.

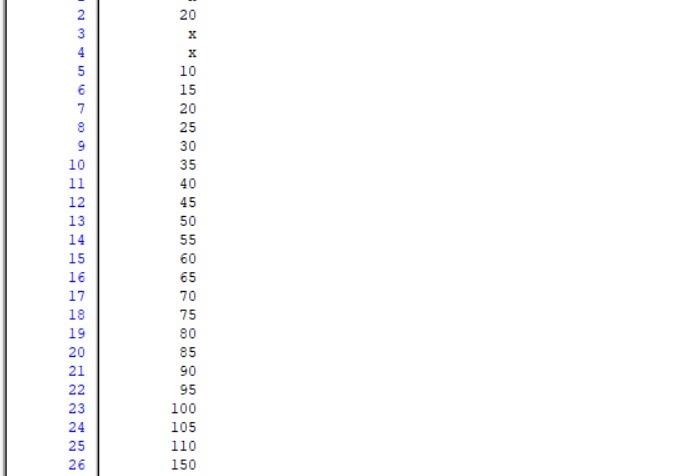
**Resultados:**

Una vez creado el TestBench e instanciado los módulos se corrió la simulación y lo primero fue cargar la memoria de instrucciones ADD:



Diciéndole que vamos a sumar lo que este en la posición 17,19 y se lo vamos a agregar a la posición 26.

Corroboramos los resultados de revisando nuestro banco de registros:



**Referencias:**

MIPS Architecture. (n.d.). Retrieved from<https://mips.com/>

LjubisaBajic, Founder and CEO, Tenstorrent - Topio Networks. <https://www.topionetworks.com/people/ljubisa-bajic-588b09eb2c537740e300002a>

Cutress, I. (2021, May 27). An Interview with Tenstorrent: CEO Ljubisa Bajic and CTO Jim Keller. AnandTech.

[https://www.anandtech.com/show/16709/an-interview-with-tenstorrent-ceo-ljubisa-bajic-and-ctojim-keller](https://www.anandtech.com/show/16709/an-interview-with-tenstorrent-ceo-ljubisa-bajic-and-cto-jim-keller)

# Fase 2-Instrucciones tipo I

# Logotipo Descripción generada automáticamente

* Hermosillo Guizar Irvin Yael
* Jauregui Rabelero Hugo Johnathan
* Cruz Sanchez Juan Pablo
* Covarrubias Becerra Juan Pablo

Arquitectura de Computadoras

Jorge Ernesto Lopez Arce-Proyecto Final

# Introducción Fase 2-Instrucciones tipo I.

La implementación del proyecto en la primera fase es capaz de realizar operaciones aritméticas y lógicas simples (suma, resta, OR, AND, SLT y NOP), esta segunda fase permitirá la ejecución de instrucciones de tipo I. Este tipo de operaciones son **de transferencia, salto condicional e instrucciones con operandos inmediatos**.

Este tipo de operaciones (en especial transferencia) serán de gran ayuda para comenzar a integrar la memoría de datos en algoritmos cargados en la memoría de instrucciones y para comenzar a implementar estructuras cíclicas y condicionales.

# Objetivos Fase 2-Instrucciones tipo I.

**General:**

Agregar los módulos necesarios al datapath para poder ejecutar las instrucciones tipo I de la tabla 1 y tabla 2. Vea el archivo Fase2.PDF para más detalles.Tabla

Descripción generada automáticamente

# Desarrollo de Verilog:

# Incluir 2 nuevos módulos para implementación de tipo I (signExtend 32 y shift2left), con ellas podremos realizar saltos condicionales, referente a instrucciones inmediatas y carga o lectura de datos en memoría.

# Desarrollo del decodificador:

# Traducir las instrucciones tipo I incluidas en *Tabla 1* y *Tabla 2* a código ensamblador.

# Desarrollo Fase 2-Instrucciones tipo I.

Esta fase requiere la incorporación de 2 módulos nuevos (**sign extend 32** y **shift left 2**) estas operaciones nos serán de ayuda para las instrucciones de salto y para las instrucciones inmediatas, por su parte, las instrucciones de transferencia (**SW**, **LW**) necesitan sobre todo cambios en la unidad de control. El resultado de las nuevas implementaciones debe ser el descrito en la siguiente imagen.

Diagrama

Descripción generada automáticamente

La extensión de signo de 32 bits tomará una entrada de 16 bits. A esta entrada se le añadirán 16 bits adicionales en los espacios más significativos (a la izquierda). Estos bits adicionales se asignarán con un valor de “1” o “0”, dependiendo del bit más significativo de la entrada proporcionada. Esta salida ampliada de 32 bits puede tener dos posibles destinos:

* La Unidad Lógica Aritmética (ALU): Esto ocurre si es una instrucción inmediata o para instrucciones como Beq o similares.
* Desplazamiento a la Izquierda 2 (Shift 2 Left): Esto ocurre si es una instrucción de salto (Beq o Bnq).

Además, se requerirá un segundo sumador en el ciclo de búsqueda (fetch) para las instrucciones de tipo J (que no corresponden a esta fase).

Volviendo a las instrucciones de tipo I, el valor de este sumador solo se considerará si la salida del módulo ALU resulta en un zeroFlag de “1” (lo que significa que el resultado de la ALU es “0”) y si la unidad de control identificó la instrucción como una de salto (branch).

Cuando verificamos que el resultado de la ALU es “0”, estamos hablando de una instrucción de salto de tipo beq (branch). Su objetivo es comparar dos entradas numéricas proporcionadas. Esta comparación se realiza a través de una operación de resta, y si el zeroFlag resulta en 1 en este caso, indica que los dos valores numéricos son iguales.

Hemos descrito la instrucción de salto condicional, que puede ser utilizada para instrucciones condicionales, pero no para instrucciones cíclicas. Esto se debe a que solo puede realizar saltos de instrucciones en adiciones, es decir, solo puede saltar hacia adelante en el documento de instrucciones y no puede volver atrás. Para ello, se implementa una instancia de desplazamiento a la izquierda 2 (shift left 2) en el ciclo de búsqueda que se desarrollará en la siguiente fase.

**Módulos Implementados en esta fase:**

El módulo PC es el encargado de gestionar la salida de la dirección de lectura de la “memoria de instrucciones”. Su función es parecida a la de un almacenador temporal o búfer.

Por otro lado, el Add o Adder lleva a cabo una operación de adición con las dos entradas que se le proporcionan, y emite el resultado de dicha operación como salida.

**Módulos de Utilidad en esta fase:**

**Multiplexores:**

Los multiplexores son dispositivos que reciben dos entradas y, según el valor de un "selector" controlado principalmente por la unidad de control, emiten una de las dos entradas como salida. La dimensión de las entradas y salidas puede variar según su implementación, pero en general, la salida tiene la misma longitud que la entrada. Es posible que se requieran múltiples multiplexores, diferenciándose solo en la longitud de la palabra.

**Módulos Nuevos de Utilidad:**

**1.Sign Extend:**

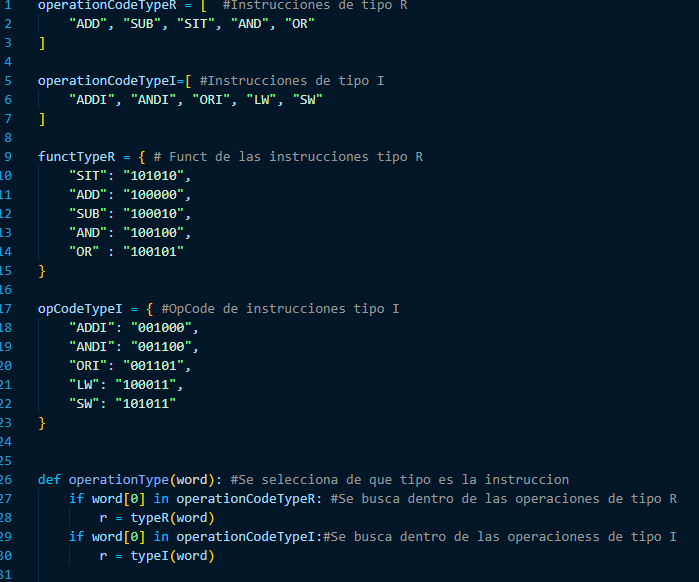
Este módulo toma una entrada de 16 bits. En términos generales, extrae los 16 bits menos significativos de la instrucción y los extiende a 32 bits. El proceso de extensión implica replicar el valor del bit más significativo de la entrada en los 16 bits de extensión. En otras palabras, si el bit más significativo es 1, los 16 bits de relleno también serán 1; lo mismo se aplica si el bit más significativo es 0.

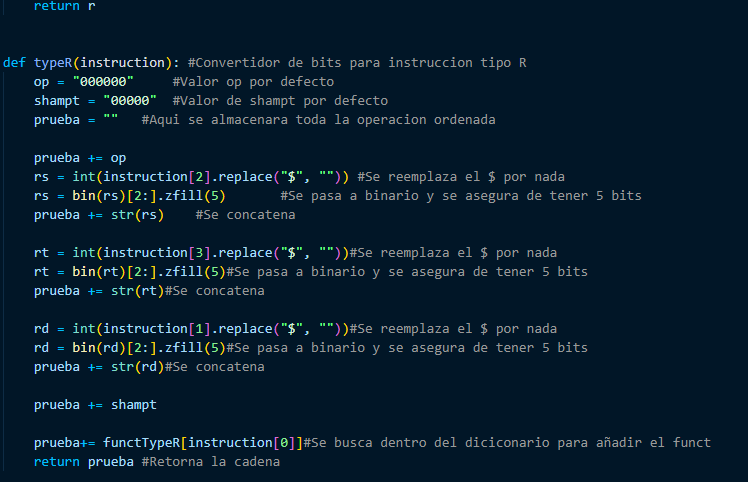
**2.Shift Left 2:**

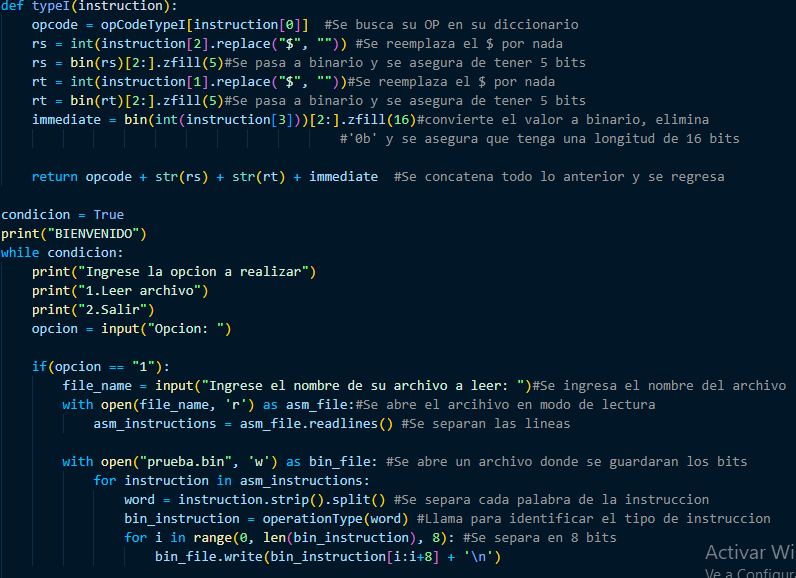
Este módulo realiza un desplazamiento de 2 bits a la izquierda en la entrada recibida. La entrada puede tener diferentes tamaños, similar a los multiplexores, ya que también se implementará en la fase 3. Por ejemplo, si la entrada es 0011001, la salida será 1100100.

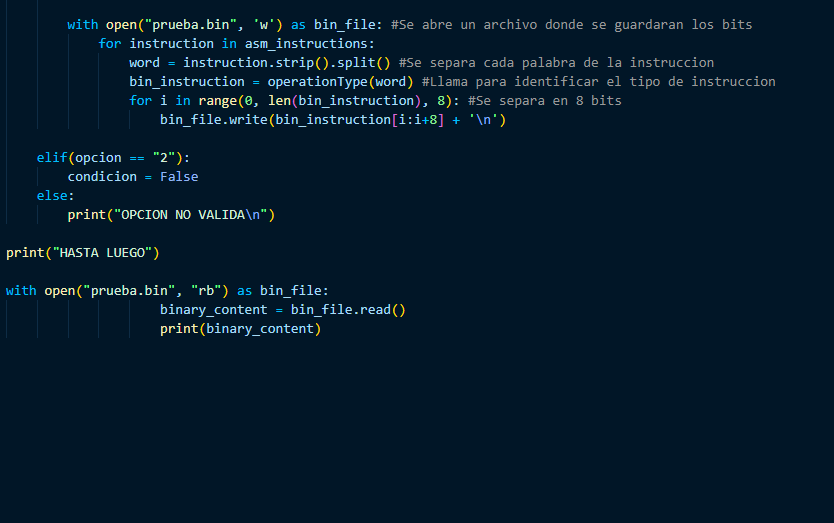
Estos módulos y multiplexores son componentes esenciales en la arquitectura de un sistema, permitiendo la manipulación y extensión de datos, así como la selección de entradas en función de un controlador.

**Desarrollo del decodificador Fase 2-Instrucciones tipo I.**

****

****

****

****

**Funcionamiento Detallado de Instrucciones Tipo I:**

**def typeI(instruction):**

**opcode = opCodeTypeI[instruction[0]]**

Se busca el OpCode correspondiente en el diccionario para la instrucción tipo I.

**rs = int(instruction[2].replace("$", ""))**

Se obtiene el valor del registro rs, eliminando el símbolo '$'.

**rs = bin(rs)[2:].zfill(5)**

Se convierte rs a binario y se asegura de que tenga 5 bits.

**rt = int(instruction[1].replace("$", ""))**

Se obtiene el valor del registro rt, eliminando el símbolo '$'.

**rt = bin(rt)[2:].zfill(5)**

Se convierte rt a binario y se asegura de que tenga 5 bits.

**immediate = bin(int(instruction[3]))[2:].zfill(16)**

Se convierte el valor inmediato a binario, se elimina '0b' y se asegura de tener 16 bits.

**return opcode + str(rs) + str(rt) + immediate**

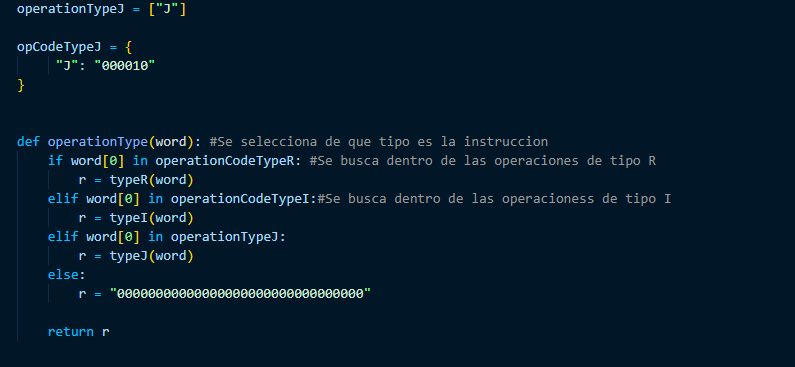
Se concatenan todos los elementos para formar la representación binaria de la instrucción tipo I.

**Explicación línea por línea:**

1. **opcode = opCodeTypeI[instruction[0]]**: Se busca en el diccionario **opCodeTypeI** el valor correspondiente al OpCode de la instrucción actual (**instruction[0]** es el mnemónico de la instrucción, por ejemplo, "ADDI"). El resultado se almacena en la variable **opcode**.
2. **rs = int(instruction[2].replace("$", ""))**: Se obtiene el valor del registro rs desde la instrucción. Se elimina el símbolo '$' si está presente, y el resultado se almacena en la variable **rs**.
3. **rs = bin(rs)[2:].zfill(5)**: Se convierte el valor de **rs** a binario (**bin(rs)**), se eliminan los dos primeros caracteres ('0b') y se asegura de que tenga una longitud de 5 bits utilizando **zfill(5)**.
4. **rt = int(instruction[1].replace("$", ""))**: Similar a la línea 2, se obtiene el valor del registro rt desde la instrucción, eliminando el símbolo '$'. El resultado se almacena en la variable **rt**.
5. **rt = bin(rt)[2:].zfill(5)**: Similar a la línea 3, se convierte el valor de **rt** a binario y se asegura de que tenga una longitud de 5 bits.
6. **immediate = bin(int(instruction[3]))[2:].zfill(16)**: Se convierte el valor inmediato a binario, se eliminan los dos primeros caracteres ('0b') y se asegura de que tenga una longitud de 16 bits.
7. **return opcode + str(rs) + str(rt) + immediate**: Se concatenan los elementos (**opcode**, **rs**, **rt**, y **immediate**) para formar la representación binaria completa de la instrucción tipo I. El resultado se devuelve.

**(Solo es la explicación de las instrucciones de Tipo I del decodificador)**.

**A continuación, una breve explicación de las instrucciones tipo J, recién agregadas:**

****

Las instrucciones de tipo J (Jump) son utilizadas en arquitecturas de computadoras para realizar saltos incondicionales en el flujo de ejecución de un programa. Estas instrucciones permiten alterar la secuencia normal de ejecución y dirigirse a una dirección de memoria específica.

En el código que proporcionaste, la instrucción de tipo J es representada por la operación "J". A continuación, te explico brevemente cada parte de la instrucción de tipo J:

**- Operación (Opcode):** En el caso de las instrucciones de tipo J, el opcode específico para la operación "J" es "000010". Esto se almacena en el diccionario `opCodeTypeJ`.

**- Dirección de destino:** La dirección de destino es la dirección de memoria a la que se realizará el salto. En tu implementación, la dirección de destino se ingresa como un valor inmediato (por ejemplo, `"10"`), y se convierte a binario con `bin(int(instruction[1]))[2:].zfill(26)` para asegurar que tenga una longitud de 26 bits.

**En resumen**, una instrucción de tipo J consta del opcode correspondiente a la operación de salto (en este caso, "000010") y la dirección de destino de 26 bits.

**Por ejemplo**, si tu instrucción es `"J 10"`, en binario podría ser algo como: `"000010000000000000000000001010"`. Esta instrucción provocaría un salto incondicional a la dirección de memoria 10.

**Conclusiones:**

El desarrollo de esta fase dentro de la implementación en verilog fue relativamente sencilla, al considerar solo 2 módulos adicionales y cambios en la unidad de control. Su implementación nos hará capaces de aprovechar mejor los componentes ya incluidos en la fase 1 para realizar tareas un poco más elaboradas que puedan contener saltos de instrucciones condicionados, carga y almacenamiento de datos e instrucciones inmediatas cuando no se necesite utilizar datos de memoria.

**Referencias:**

(n.d.). ARQUITECTURA MIPS. Retrieved 2023, from https://www.infor.uva.es/~bastida/OC/TRABAJO2\_MIPS.pdf