Arquitectura de Computadoras
Trabajo Final: MIPS 5 Etapas

2024

Alumna: GINA COMMISSO

MIPS

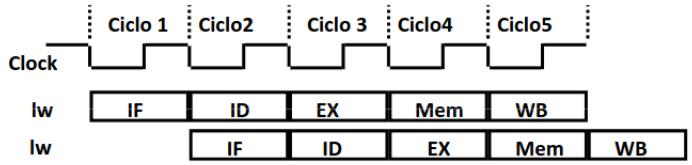
Las siglas MIPS (Microprocessor without Interlocked Pipelines Stages, o lo que es lo mismo, microprocesador sin enclavamiento de estados de tuberías) hacen referencia a la gama de microprocesadores desarrollados por MIPS Technologies, de arquitectura RISC y registros tipo propósito general de clasificación registro-registro, en los que la mayoría de las instrucciones no acceden a memoria (salvo las instrucciones de carga/descarga) y las instrucciones de los procesadores presentan dos operandos, el fuente y el resultado.

En un primer momento bastantes fabricantes basaron el diseño de sus Workstation en procesadores MIPS (como ACER, NEC, Siemens-Nixdorf, etc), sin embargo su uso ha decaído, enfocándose su empleo en sistemas embebidos, gracias a sus características de implementación (un bajo consumo energético y gran disponibilidad de herramientas de desarrollo y expertos conocedores de la arquitectura), y desarrollo de iteraciones de alto rendimiento de esta arquitectura a favor de procesadores basados en la tecnología Intel IA64.

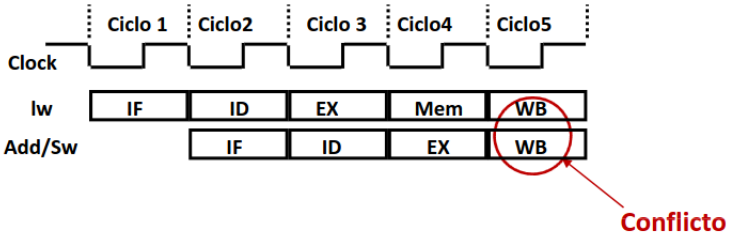
Hablando un poco de su historia, el origen de la arquitectura MIPS se remonta al año 1981. En la universidad de Standford, un equipo liderado por John L. Hennessy (fundador de MIPS Technologies) comienza a trabajar en lo que sería posteriormente el primer procesador MIPS. Su idea era mejorar a gran escala el rendimiento de la máquina a través del uso de la segmentación.

La segmentación es una técnica de implementación en la que múltiples instrucciones se solapan en ejecución. Cada etapa opera en paralelo con otras etapas pero sobre instrucciones diferentes. Se puede aplicar porque las fases por la que pasa una instrucción no usan todas las componentes de la ruta de datos componentes de la ruta de datos. Explota el paralelismo a nivel de instrucciones.

- Ejemplo 1



- Ejemplo 2



Señales de control

Señal de control	Descripción	0	1
AluSrc	Controla de donde proviene el segundo operando de la ALU	El segundo operando de la ALU viene de Read Data 2 (archivo de registros)	El segundo operando de la ALU viene de los 16 bits más bajos de la instrucción luego de pasar por el sign-extender.
MemToReg	Especifica de donde viene el dato a escribir en la entrada de escritura del archivo de registros	El valor que se escribe en la entrada Write data del Archivo de registros viene de la ALU	El valor que se escribe en la entrada Write data del Archivo de registros viene de la memoria de datos
TakeBranch	Indica si habrá BE o BNE	No hay Branch	Hay Branch
TypeBranch	Indica si el branch es BE o BNE	BEQ	BNE
TakeJump	Indica si habrá un Jump o JAL	No hay salto	Hay salto
TakeJumpR	Indica si hay un Jump Register o Jump and Link Register	No hay salto	Hay Salto
RegDst	Controla qué registro se utilizará como destino para escribir el resultado de la instrucción	El registro destino para el Write Register viene del campo rt [20:16]	El registro destino para el Write Register viene del campo rd [15:11]
MemRead	Controla si se leerá o no la memoria	Nada	El contenido de la memoria en la dirección especificada en la entrada, se colocará en la salida Read Data
MemWrite	Controla si se escribe o no en la memoria de datos	Nada	El contenido de la memoria en la dirección especificada por la entrada será reemplazado por el valor presente en la entrada de Write data.
RegWrite	Indica si se va a escribir en un registro.	Nada	El registro en la entrada de Write Register es escrito con el valor en la entrada de Write Data WD3
GPR31	Indica si la instrucción es JAL y se almacena la dirección de retorno en GPR31	Toma el valor de retorno de la instrucción (rd)	Toma el valor de retorno de GPR31

Instrucción	RegDst	RegWrite	ALUSrc	MemToReg	MemWrite	MemRead	TakeBranch	TypeBranch	TakeJump	TakeJumpR	GPR31
NOP	0	0	0	0	0	0	0	0	0	0	0
Tipo R	1	1	0	0	0	0	0	0	0	0	0
TIPO I Básica	0	1	1		0	0	0	0	0	0	0

s (ALUI)											
LB	0	1	1	1	0	1	0	0	0	0	0
LH	0	1	1	1	0	1	0	0	0	0	0
LW	0	1	1	1	0	1	0	0	0	0	0
LWU	0	1	1	1	0	1	0	0	0	0	0
LBU	0	1	1	1	0	1	0	0	0	0	0
LHU	0	1	1	1	0	1	0	0	0	0	0
SB	X	0	1	X	1	0	0	0	0	0	0
SH	X	0	1	X	1	0	0	0	0	0	0
SW	X	0	1	X	1	0	0	0	0	0	0
SLTI	0	1	1	0	0	0	0	0	0	0	0
BEQ	X	0	0	0	0	0	1	0	0	0	0
BNE	X	0	0	0	0	0	1	1	0	0	0
J	X	0	x	0	0	0	0	0	1	0	0
JAL	X	1	x	0	0	0	0	0	1	0	1
JR	X	0	x	0	0	0	0	0	1	1	0
JALR	1	1	X	0	0	0	0	0	1	1	0

Operaciones de la ALU

ADD	000001	00010100001, 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 100000
SUB	000010	000010,100101,110000,110001
AND	000011	000011, 100001
OR	000100	000100, 100010
XOR	000101	000101, 100011
NOR	000110	000110
SRL	000111	001000
SRA	001000	000111
SLL	001001	001001
SLL16	001010	100100

Lógica de los códigos de operación

Para los códigos de operación se tiene en cuenta que hay varios tipos de instrucciones más el NOP y el HALT que son instrucciones particulares.
Como son 8 grupos en total, se necesitan 3 bits para diferenciar entre grupos. El resto de los bits se usarán para representar las diferencias entre las instrucciones que pertenecen al mismo grupo.

- NOP 000000
- TIPO R Aritmeticas y Logicas 000*** (excepto el 000000) y 001***: Si los tres MSB valen 0, el decodificador sabrá que se trata de una operación tipo R y pondrá regDst en 1 y regWrite en 1 y el resto de señales en 0.
- Tipo I ALU 100*** : Si los tres MSB valen 1, el decodificador sabrá que se trata de las operaciones típicas de la ALU pero con valores inmediatos, por lo tanto regWrite y AluSrc estaran en 1 y el resto en 0.
- LOAD 010*** : Si los tres MSB valen 2, se trata de operaciones de carga de datos de la memoria a registros, por lo tanto MemRead, MemToReg, AluSrc y RegWrite estarán en 1.
- STORE 011*** : Si lo s tres MSB valen 3, se trata de operaciones de almacenamiento de datos en memoria, por lo tanto MemWrite y AluSrc estarán en 1.
- BEQ Y BNE 100***: Estas instrucciones tienen en común el TakeBranch y se diferencian en TypeBranch.
- JUMPS 111*** : Si los tres MSB valen 5, se tratara de una instrucción de salto que tendrá en 1 el TakeJump. Sin embargo se diferencian en varias se;ales:
    - Jump 111000 : solo TakeJump en 1.
    - Jump and Link 111001 : salta a una dirección pero guarda la dirección de retorno en GPR31 por lo tanto GPR31, RegWrite y TakeJump estarán en 1.
    - Jump Register 111010 : TakeJump y TakeJumpR en 1
    - Jump and Link Register 111011 : RegDst, RegWrite, TakeJump y TakeJumpR en 1.
- HALT 111111

Tipo	Nombre	Opcode	AluOP	Formato
Nulo	Nop	000000	xxxxxx	OP(6)+F(26)
R	ADDU	000001	000001	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	SUBU	000010	000010	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	AND	000011	000011	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	OR	000100	000100	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	XOR	000101	000101	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	NOR	000110	000110	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	SRAV	000111	001000	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	SRLV	001000	000111	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	SLLV	001001	001001	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
R	SLT	001010	001000	OP(6)+rs(5)+rt(5)+rd(5)+F(11)
I	LB	010000	000001	OP(6)+base(5)+rt(5)+offset(16)
I	LH	010001	000001	OP(6)+base(5)+rt(5)+offset(16)
I	LW	010010	000001	OP(6)+base(5)+rt(5)+offset(16)
I	LWU	010011	000001	OP(6)+base(5)+rt(5)+offset(16)
I	LBU	010100	000001	OP(6)+base(5)+rt(5)+offset(16)
I	LHU	010101	000001	OP(6)+base(5)+rt(5)+offset(16)
I	SB	011110	000001	OP(6)+base(5)+rt(5)+offset(16)
I	SH	011111	000001	OP(6)+base(5)+rt(5)+offset(16)
I	SW	011000	000001	OP(6)+base(5)+rt(5)+offset(16)
I	ADDI	100000	000001	OP(6)+rs(5)+rt(5)+inmed(16)
I	ANDI	100001	000011	OP(6)+rs(5)+rt(5)+inmed(16)
I	ORI	100010	000100	OP(6)+rs(5)+rt(5)+inmed(16)
I	XORI	100011	000101	OP(6)+rs(5)+rt(5)+inmed(16)
I	LUI	100100	001010	OP(6)+F(5)+rt(5)+inmed(16)
I	SLTI	100101	000010	OP(6)+rs(5)+rt(5)+inmed(16)
I	BEQ	110000	000010	OP(6)+rs(5)+rt(5)+offset(16)
I	BNE	110001	000010	OP(6)+rs(5)+rt(5)+offset(16)
I	J	111000	000000	OP(6)+dirección(26)
I	JAL	111001	000000	OP(6)+dirección(26)

Tipo	Nombre	Opcode	AluOP	Formato
J	JR	111010	000000	OP(6)+rs(5)+F(21)
J	JALR	111011	000000	OP(6)+rs(5)+F(5)+rd(5)+F(11)
HALT	HALT	1111111	xxxxxx	OP(6)+F(26)

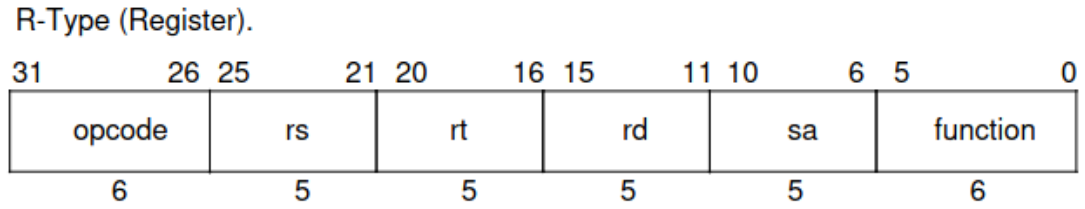
F significa que son bits libres, es decir pueden tener cualquier valor.

Instrucciones

Estos tipos de procesadores ejecutan tres tipos de instrucciones.

Tipo R

Son instrucciones aritmeticas y logicas



Un código de operación, primer registro operando fuente, segundo registro operando fuente, registro destino donde se almacena el resultado de la operación, shamt o tamaño de desplazamiento y función que selecciona la variante específica de la operación del campo opcode.

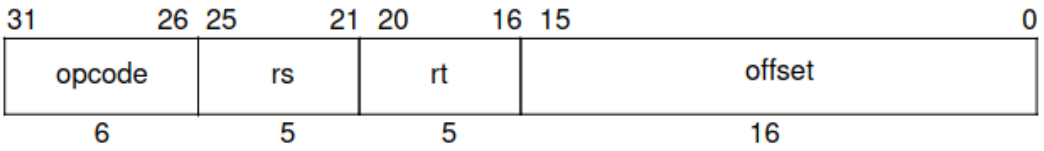
Instrucción	Descripción	Operación de la ALU	Ejemplo
SLL	Desplaza los bits a la izquierda por una cantidad específica y rellena con ceros. Almacena el resultado en un registro.	SLL	sll $t0, $t1, 2 (desplaza el contenido de $t1 dos bits a la izquierda y almacena el resultado en $t0).
SRL	Desplaza los bits a la derecha por una cantidad específica y rellena con ceros. Almacena el resultado en un registro.	SRL	srl $t0, $t1, 3 (desplaza el contenido de $t1 tres bits a la derecha y almacena el resultado en $t0).
SRA	Desplaza los bits del valor del registro fuente a la derecha por una cantidad específica y llena los bits vacíos con el bit de signo original.	SRA	sra $t0, $t1, 4 (desplaza el contenido de $t1 cuatro bits a la derecha con extensión de signo y almacena el resultado en $t0).

SLLV	Similar a SLL, pero la cantidad de bits a desplazar se toma del valor en el registro rs.	SLL	sllv $t0, $t1, $t2 (desplaza el contenido de $t1 por la cantidad de bits especificada en $t2 y almacena el resultado en $t0)
SRLV	Similar a SRL, pero la cantidad de bits a desplazar se toma del valor en el registro rs.	SRL	srlv $t0, $t1, $t2 (desplaza el contenido de $t1 por la cantidad de bits especificada en $t2 y almacena el resultado en $t0)
SRAV	Similar a SRA, pero la cantidad de bits a desplazar se toma del valor en el registro rs.	SRA	srav $t0, $t1, $t2 (desplaza el contenido de $t1 por la cantidad de bits especificada en $t2 con extensión de signo y almacena el resultado en $t0).
ADDU	Realiza la suma de dos registros sin considerar el desbordamiento.	ADD	addu $t0, $t1, $t2 (suma el contenido de $t1 y $t2 y almacena el resultado en $t0).
SUBU	Realiza la resta de dos registros sin considerar el desbordamiento.	SUB	subu $t0, $t1, $t2 (resta el contenido de $t2 de $t1 y almacena el resultado en $t0).
AND		AND	
OR		OR	
XOR		XOR	
NOR		NOR	
SLT	Establece el valor del registro de destino en 1 si el valor del registro fuente es menor que el valor del registro de destino; de lo contrario, se establece en 0.	SUB	slt $t0, $t1, $t2 (establece $t0 en 1 si el contenido de $t1 es menor que $t2; de lo contrario, establece $t0 en 0)

Tipo I (inmediatas)

Son instrucciones de transferencia, salto condicional y con operandos inmediatos (valores constantes). Suelen realizar operaciones que involucran la transferencia de datos entre registros y memoria.

I-Type (Immediate).



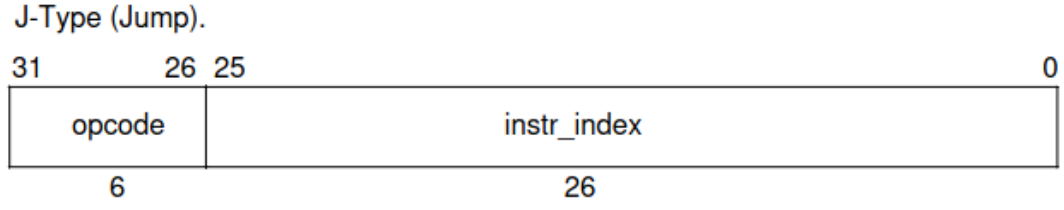
LB	Carga un byte de memoria en un registro con extensión de signo.	ADD	lb $t0, 100($t1) (carga el byte en la dirección de memoria $t1 + 100 en $t0 con extensión de signo).
LH	Carga una palabra de memoria en un registro con extensión de signo.	ADD	lh $t0, 200($t1) (carga la media palabra en la dirección de memoria $t1 + 200 en $t0 con extensión de signo).
LW	Carga una palabra de memoria en un registro.	ADD	lw $t0, 300($t1) (carga la palabra en la dirección de memoria $t1 + 300 en $t0).
LWU	Carga una palabra de memoria en un registro sin extensión de signo.	ADD	lwu $t0, 400($t1) (carga la palabra en la dirección de memoria $t1 + 400 en $t0 sin extensión de signo).
LBU	Almacena la media palabra más baja de un registro en memoria.	ADD	lhu $t0, 600($t1) (carga la media palabra en la dirección de memoria $t1 + 600 en $t0 sin extensión de signo).
LHU	Carga una media palabra de memoria en un registro sin extensión de signo.	ADD	

SB	Almacena el byte más bajo de un registro en memoria.	ADD	sb $t0, 700($t1) (almacena el byte más bajo de $t0 en la dirección de memoria $t1 + 700).
SH		ADD	sh $t0, 800($t1) (almacena la media palabra más baja de $t0 en la dirección de memoria$t1 + 800).
SW	Almacena la palabra completa de un registro en memoria.	ADD	sw $t0, 900($t1) (almacena la palabra completa de $t0 en la dirección de memoria $t1 + 900).
ADDI	Suma un valor inmediato a un registro.	ADD	addi $t0, $t1, 100 (suma 100 al contenido de $t1 y almacena el resultado en $t0).
ANDI	Realiza la operación AND entre un registro y un valor inmediato.	AND	Ejemplo: andi $t0, $t1, 255 (realiza AND entre el contenido de $t1 y 255 y almacena el resultado en $t0).
ORI	Realiza la operación OR entre un registro y un valor inmediato.	OR	ori $t0, $t1, 65535 (realiza OR entre el contenido de $t1 y 65535 y almacena el resultado en $t0).
XORI	Realiza la operación XOR entre un registro y un valor inmediato.	XOR	xori $t0, $t1, 128 (realiza XOR entre el contenido de $t1 y 128 y almacena el resultado en $t0).
LUI	Carga un valor inmediato en los bits superiores de un registro.	SLL16	Ejemplo: lui $t0, 0xFFFF0000 (carga 0xFFFF0000 en los bits superiores de $t0).

SLTI	Establece el valor del registro de destino en 1 si el valor del registro fuente es menor que el valor inmediato; de lo contrario, se establece en 0.	SUB	slti $t0, $t1, 50 (establece $t0 en 1 si el contenido de $t1 es menor que 50; de lo contrario, establece $t0 en 0).
BEQ	Salta a una dirección específica si dos registros son iguales.	SUB	beq $t0, $t1, etiqueta (salta a la etiqueta si el contenido de $t0 es igual al contenido de $t1).
BNE	Salta a una dirección específica si dos registros no son iguales.	SUB	bne $t0, $t1, etiqueta (salta a la etiqueta si el contenido de $t0 no es igual al contenido de $t1).
J	Salto incondicional a una dirección específica.	/	j etiqueta (salta a la dirección de memoria especificada por la etiqueta).
JAL	Salto a una dirección específica y guarda la dirección de retorno en el registro $ra.	/	jal etiqueta (salta a la dirección de memoria especificada por la etiqueta y guarda la dirección de retorno en $ra).

Tipo J

Estas instrucciones están relacionadas con operaciones de salto (branching) y saltos incondicionales. El único campo significativo en una instrucción J-type es la dirección de destino, que es el objetivo del salto. Las instrucciones J-type se utilizan para implementar saltos incondicionales y bifurcaciones en programas.



JR	Salta a la dirección almacenada en un registro.	/	jr $ra (salta a la dirección almacenada en el registro de dirección de retorno $ra).
JALR	Salta a la dirección almacenada en un registro y guarda la dirección de retorno en un registro específico.	/	Ejemplo: jalr $t0, $t1 (salta a la dirección almacenada en $t1 y guarda la dirección de retorno en $t0).

Módulos

Etapa INSTRUCTION FETCH

Program Counter

El PC (Program Counter) en un procesador MIPS almacena la dirección de la próxima instrucción que se va a ejecutar. Esta dirección se utiliza para buscar la instrucción correspondiente en la memoria de instrucciones. Después de que se ha accedido a la instrucción en la memoria de instrucciones, la dirección almacenada en el PC se incrementa en 4 para apuntar a la siguiente instrucción, ya que las instrucciones en MIPS son de longitud fija de 32 bits y se alinean en memoria.

El valor del PC incrementado en 4 (que representa la dirección de la siguiente instrucción) pasa a través de un multiplexor. El propósito de este multiplexor es permitir que el flujo del programa pueda ser controlado, ya sea por una secuencia de instrucciones secuencial (cuando no hay saltos o ramificaciones) o por una dirección alterna cuando ocurre un salto o una rama condicional. El controlador del multiplexor seleccionará entre la dirección incrementada en 4 (para ejecución secuencial) o una dirección diferente (cuando se toma una decisión de salto o rama) y esa dirección seleccionada se carga de vuelta en el PC, actualizando así la secuencia de ejecución del programa.

El **multiplexor** que se encuentra antes del Program Counter (PC) en el diseño de un procesador MIPS es crucial para determinar la próxima dirección que el PC deberá cargar. Este multiplexor selecciona entre dos posibles fuentes de dirección:

1. La dirección calculada para la siguiente instrucción (PC + 4).
2. La dirección de salto proveniente de una instrucción de salto (como un salto condicional, una instrucción de salto directo, una llamada a subrutina, etc.).

El controlador del multiplexor toma una decisión basada en la operación actual del procesador y el resultado de las etapas anteriores del pipeline. Por ejemplo:

- Si no se detecta ningún salto o rama condicional, el controlador del multiplexor seleccionará la dirección calculada (PC + 4) como la próxima dirección a cargar en el PC. Esto asegura que el procesador avance secuencialmente a través del programa.
- Si se detecta un salto o una rama condicional y se toma el salto, el controlador del multiplexor seleccionará la dirección de salto como la próxima dirección a cargar en el PC. Esto desvía el flujo del programa hacia la dirección de destino del salto.

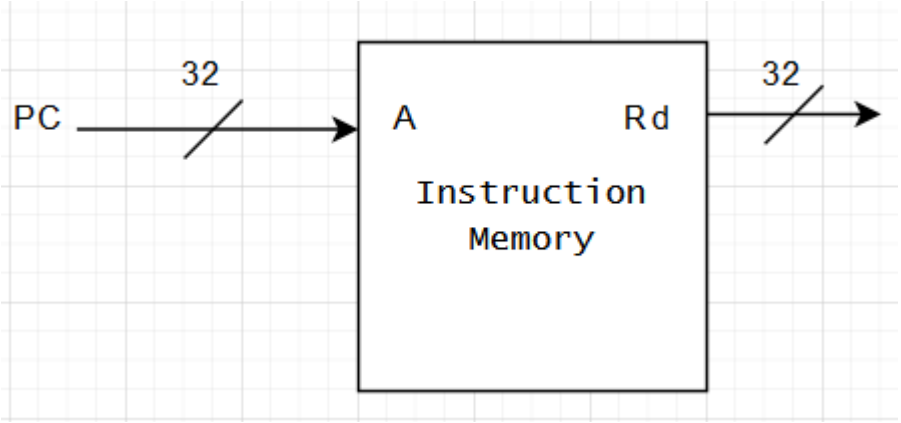
El controlador del multiplexor recibe señales de control desde las etapas anteriores del pipeline, como la etapa de decodificación (ID) donde se decodifican las instrucciones y se determina si alguna instrucción de salto debe tomarse. Además, puede recibir señales de otros bloques del procesador que detecten eventos como interrupciones o excepciones que podrían cambiar el flujo del programa.

Entradas	Clock Reset Próxima dirección [32]
Salidas	Dirección de la próxima instrucción [32]
Sincronismo	posedge

Memoria de Instrucciones

En este módulo residen las instrucciones del procesador. Recibe una dirección A y devuelve el contenido almacenado en esa dirección. Según el valor del PC, el procesador selecciona una instrucción de esta memoria para luego decodificarla.

Entradas	Clock Reset Dirección de la próxima instrucción [32]
Salidas	Instrucción [32]
Sincronismo	



Latch IF/ID

El propósito principal del latch IF/ID es mantener la integridad de los datos mientras se propagan a través del pipeline, asegurando que los datos necesarios para la siguiente etapa estén disponibles y estables.

Entradas	Clock Reset Instrucción [32] PC+4 [32]
Salidas	Instrucción [32] PC+4 [32]
Sincronismo	

Multiplexores

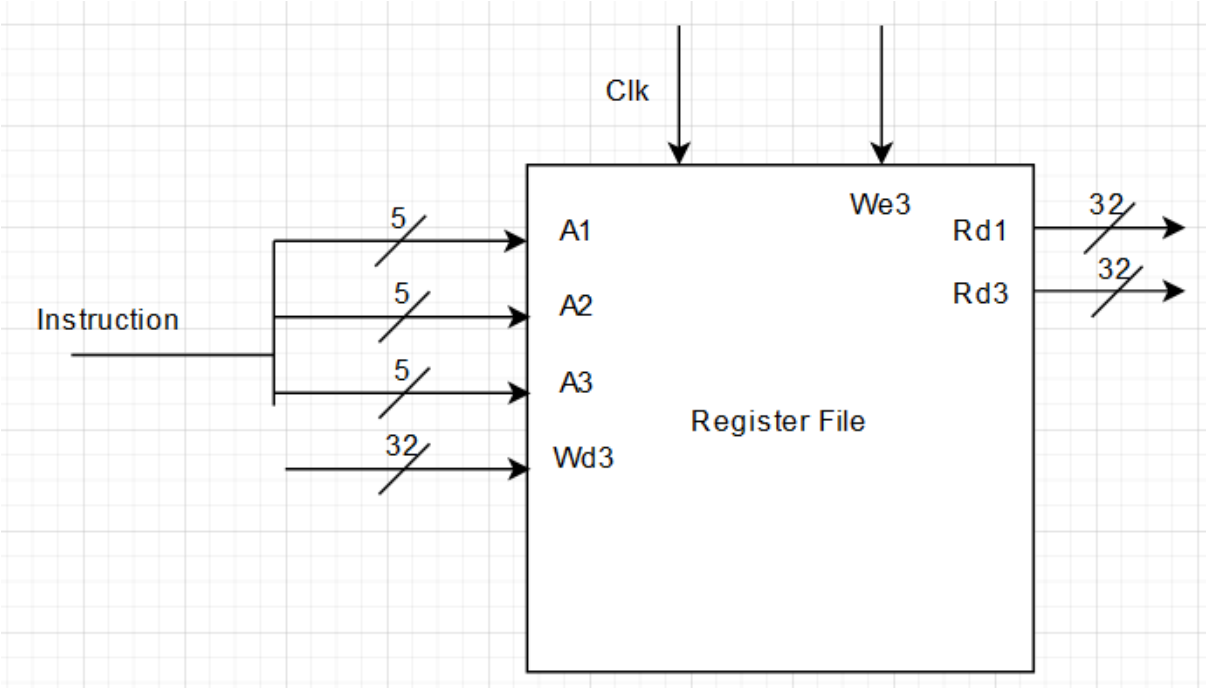
1. **Multiplexor para instrucciones jal y jalr:**
   - Entradas:
     - PC + 4: Dirección de la siguiente instrucción en el flujo secuencial.
     - Dirección calculada por la instrucción jal o jalr: Esta dirección puede ser la dirección de retorno, almacenada en el registro de enlace.
   - Selección: Una señal de control activada por las instrucciones jal y jalr.
   - Función: Selecciona la dirección calculada por la instrucción jal o jalr si están activas, de lo contrario, selecciona PC + 4.
2. **Multiplexor para instrucciones de salto (branch):**
   - Entradas:
     - PC + 4: Dirección de la siguiente instrucción en el flujo secuencial.
     - Dirección calculada por la ALU si la condición de salto es verdadera: Esta dirección se utiliza para saltar a una nueva ubicación si se satisface una condición de salto.
   - Selección: Una señal de control activada por las instrucciones de salto condicional.

    - Función: Selecciona la dirección calculada por la ALU si se cumple la condición de salto, de lo contrario, selecciona PC + 4.
3. **Multiplexor para instrucción jr (jump register):**
    - Entradas:
        - PC + 4: Dirección de la siguiente instrucción en el flujo secuencial.
        - Dirección almacenada en el registro específico (registro de enlace o similar): Esta dirección es cargada desde un registro específico.
    - Selección: Una señal de control activada por la instrucción jr.
    - Función: Selecciona la dirección almacenada en el registro específico para la instrucción jr, de lo contrario, selecciona PC + 4.
4. **Multiplexor para instrucción de salto (jump):**
    - Entradas:
        - Los primeros 4 bits del PC: Parte más significativa de la dirección actual del PC.
        - 26 bits de la instrucción jump, desplazados a la izquierda y concatenados con los primeros 4 bits del PC: Estos bits forman la dirección de destino de la instrucción jump.
    - Selección: Una señal de control activada por la instrucción de salto (j).
    - Función: Selecciona la dirección calculada por la instrucción j como la próxima dirección de PC.

Etapa INSTRUCTION DECODE

Archivo de Registros

Este módulo se lo puede pensar como un “Notepad” donde se almacenan valores temporales, operaciones intermedias y resultados. A nivel de hardware, representa una parte de la memoria integrada del procesador. Puede recibir dos direcciones A1 y A2 en y retornar el contenido de dos registros, todo en simultáneo. Uno de los registros puede escribirse si la señal de habilitación está en alto y se proporciona una dirección correcta y un dato válido.



Entradas	Clock Reset Instrucción [25:21] (rs) instrucción [20:16] (rt) RegDir[4:0] WBData[31:0] Regwrite
Salidas	srcA[31:0] srcB[31:0]
Sincronismo	

Latch ID/EX

Entradas	Clock Reset PC+4 PcSrc MemToReg TakeBranch TakeJump TakeJumpr

	RegDst AluOp MemRead MemWrite srcA[31:0] srcB[31:0] signExtend[31:0]
Salidas	PC+4 PcSrc MemToReg TakeBranch TakeJump TakeJumpr RegDst AluOp MemRead MemWrite srcA[31:0] srcB[31:0] signExtend[31:0]
Sincronismo	

Unidad de Control

Entradas	opCode[31:26]
Salidas	RegWrite PcSrc MemToReg TakeBranch TakeJump TakeJumpr RegDst AluOp MemRead MemWrite AluSrc GPR31
Sincronismo	

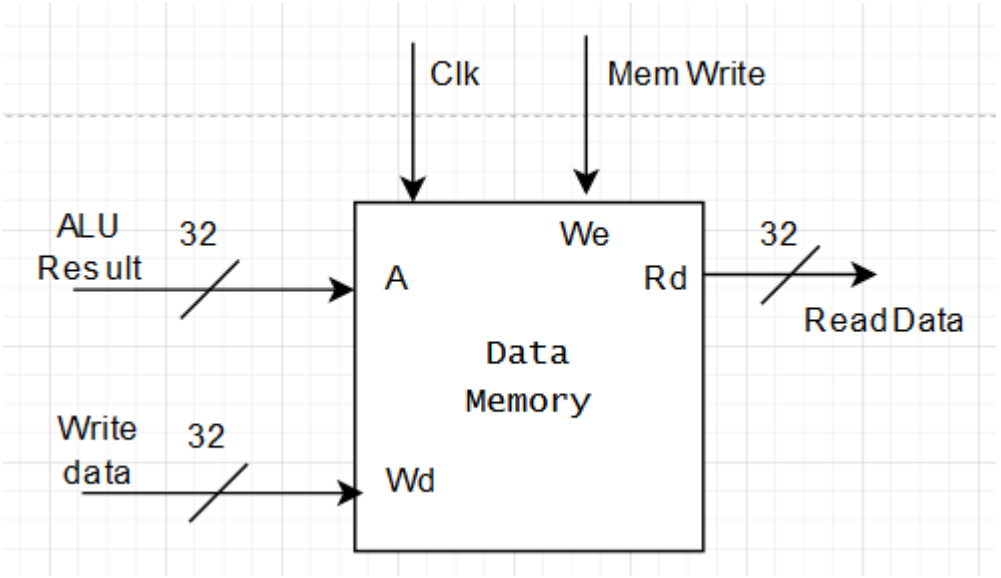
Multiplexores

- Multiplexor que selecciona entre srcB o signExtend (para instrucciones BEQ y BNQ)

Etapa EXECUTE

Memoria de Datos

Similar a la memoria de instrucciones excepto que puede ser escrita y requiere un clock. Solo reciben una dirección y los datos pueden ser leídos o escritos en un ciclo de clock. El write enable debe estar en alto para poder escribir.



Extensor de signo

Las instrucciones leídas de la memoria de instrucciones pueden contener offsets de 16 bits (para direccionar la memoria de datos) sin embargo la memoria de datos recibe direcciones de 32 bits. Este offset se agrega a la dirección como una manera de atravesar el espacio de memoria. Puede ser positivo o negativo. El extensor de signo toma el bit más significativo de los 16 bits y lo copia en el resto de bits para tener un offset de 32 bits.

ALU

Entradas	srcA[32] srcB[32] ALUO[6]
Salidas	Zero Carry aluResult[32]
Sincronismo	Asíncrona

Para seleccionar que operando de la alu se va a ejecutar, se utiliza la señal de control ALUOp.

Códigos de operación

- ADD 000001
- SUB 000010
- AND 000011
- OR 000100
- XOR 000101
- NOR 000110
- SLL16 001010
- SRA 001000
- SRL 000111
- SLL 001001

La decisión de implementar una operación SLL16 fue para hacer más simple la ejecución de la instrucción LUI

Entradas	Clock Reset Block i_halt pcplus4 result carry dato2 writeRegister memToReg regWrite MemWrite memRead memWidth

Salidas	halt pcplus4 carry writeRegister dato2 memToReg regWrite memWrite memWidth
Sincronismo	Posedge

Multiplexores

En esta etapa podemos encontrar una gran variedad de muxes.
Tres de ellos poseen 3 entradas. Estos los utiliza la unidad de cortocircuito para traer desde una etapa posterior del pipeline un dato que aún no se ha actualizado en los registros.
Dos de los muxes de 2 entradas son los encargados de determinar si hubo un branch, y en caso que lo haya, si la condición se cumplió o no.
Finalmente nos encontramos con otros dos muxes de dos entradas que se encargan de determinar de dónde provendrá el valor que indica en qué registro debe almacenarse el resultado, en caso de que deba
almacenarse.

Sumador, extensor de signo y shift

Estos elementos están configurados de la misma manera que se describe para la etapa anterior, y de manera conjunta determinan la dirección a la que se saltará en caso de un branch

Etapa MEM

Entradas	Clock Reset Dirección [32] Dato de escritura [32] MemWrite MemWidth [4]
Salidas	Dato leido [32]
Sincronismo	Negedge

La memoria consta de 128 posiciones. Cada dirección es un byte pero la salida entrega una palabra de 32 bits.

Podemos seleccionar si la escritura será de 8, 16 o 32 bits. Dependiendo de esta selección, a la hora de hacer un store se guardarán en la posición indicada y hacia las posteriores el dato de escritura. Por ejemplo, si queremos guardar una palabra de 16 bits en la posición 64, entonces los 8 LSB se guardarán en la posición 64 y los 8 MSB en la posición 65. Podemos seleccionar si las lecturas serán de 8, 16 o 32 bits. El mecanismo es idéntico al anterior. Por ejemplo, si hacemos una lectura de 32 bits a la posición 32 se entrega una palabra conformada por [32+3, 32+2, 32+1, 32].

Su funcionamiento en negedge se debe a que tiene a la entrada y salida latches que funcionan en posedge. Por lo tanto, para que haya una sincronización correcta, esta funciona en el flanco opuesto.

Frente a la señal de reset, todas sus salidas y posiciones de memoria se ponen a cero.

Multiplexores

El primer multiplexor se encarga de seleccionar cual sera la señal que entre al LATCH MEM/WB, si el resultado de la ALU o el dato que sea lea de la memoria de datos en caso de que sea una operación LOAD por ejemplo.

Hay dos multiplexores que se encargan de seleccionar el modo de operación correcto para poder recorrer la memoria en modo lectura de 32 bits, durante la lectura de los registros para ser enviados por UART.

El mux restante se encarga de darle control a la unidad de debug sobre las direcciones, para que pueda llevar a cabo el recorrido de memoria.

Etapa Write Back (WB)

Esta etapa no posee módulos en sí mismos, ya que si bien es la que se encarga de la escritura en registro, la unidad de registros pertenece a la etapa ID. La salida de este latch retorna hacia atrás, e indica si habrá escritura en los registros y cuál será el dato y el registro escrito en caso de que lo haya. Sin embargo como ambos tienen sincronización en posedge, para evitar el retardo en la unidad de registros directamente conecte las entradas del latch en vez de las salidas.

Unidad de Forwarding

Entradas	i_rs, i_rt i_rd_exmem i_reg_write_exmem- i_rd_memwb- i_reg_write_memwb- i_reg_dst i_mem_write_idex
Salidas	o_forward_a

	o_forward_b o_forward_c
sincronización	Asíncrona

Uno de los riesgos de datos que surgen de la segmentación del procesador es la lectura después de escritura. Este riesgo se produce porque, debido a que las instrucciones pueden comenzar antes de la finalización de sus predecesoras, en algunas ocasiones podemos necesitar hacer uso de un dato que aún no ha llegado a la etapa de escritura, y por tanto no está disponible en la unidad de registros. Sin embargo, en la mayoría de los casos, si bien el dato no ha sido escrito, si ha sido ya calculado u obtenido, y por lo tanto va viajando por el pipeline. La función de la unidad de cortocircuitos es detectar cuando ocurren estas situaciones, y traer desde otra etapa posterior el dato a la etapa EX, que es donde es necesario. Si bien esto soluciona casi todas las manifestaciones de este riesgo, no resuelve por sí misma el caso en el cual el dato a cortocircuitar procede de un Load, y es necesario en la operación inmediatamente siguiente. Para poder llevar a cabo su función, este módulo tiene el control de los tres muxes de 3 entradas de la etapa EX. Estos están conectados a las etapas posteriores, y pueden conectar con el dato que va viajando.

Control de los muxes:
- 00: No hay cortocircuito.
- 01: Dato traído desde latch MEM/WB.
- 10: Dato traído desde latch EX/MEM.

La lógica de funcionamiento de la unidad se detalla en el cuadro siguiente, donde quedan expresadas qué condiciones acusan la presencia de un riesgo, y cómo se modifican las salidas para responder a ellos. Como puede observarse en él, cuando un dato de un determinado registro X es necesario, y al mismo tiempo un dato destinado a ese registro avanza por las etapas posteriores, entonces se hace un cortocircuito y se trae el dato a la posición necesaria.

ForwardA = 00	Default
ForwardA = 01	if (RW(MEM/WB) && (RD(MEM/WB) == RS) && !(RW(EX/MEM) && (RD(EX/MEM) == RS))
ForwardA = 10	if (RW(EX/MEM) && (RD(EX/MEM) == RS))
ForwardB = 00	Default
ForwardB = 01	if ((RW(MEM/WB) && (RD(MEM/WB) == RT) && !(RW(EX/MEM) && (RD(EX/MEM) == RT)) && RegDst)
ForwardB = 10	if ((RW(EX/MEM) && (RD(EX/MEM) == RT)) && RegDst)

ForwardC = 00	Default
ForwardC = 01	if ((MW(ID/EX) && RD(MEM/WB) == RT) && ! (MW(ID/EX) && RD(EX/MEM) == RT)
ForwardC = 10	if (MW(ID/EX) && RD(EX/MEM) == RT)

Unidad de Detección de Riesgos

Entradas	Reset TakeJumpR (UnidadDeControl) TakeBranch (UnidadDeControl) MemToReg (LatchEXMEM) RegDst (LatchIDEX) Rs (LatchIDEX) Rt (LatchIDEX) Rd (LatchEXMEM) Señal de Post Bloqueo 1
Salidas	Bit de bloqueo (PC) Bit de bloqueo (IF/ID) Bit de bloqueo (ID/EX) Reset Unidad Control Reset LatchEXMEM Señal de Post Bloqueo 1
Sincronismo	Asincrono

La Unidad de Detección de Riesgos tiene como objetivo identificar y gestionar situaciones de riesgo que pueden ocurrir en un pipeline de procesamiento, como cuando una instrucción depende de los resultados de una instrucción anterior que aún no ha finalizado su ejecución. Esta unidad toma decisiones para detener (bloquear) o reiniciar (resetear) ciertas etapas del pipeline con el fin de evitar errores en la ejecución de las instrucciones.

La unidad monitorea varias señales de entrada, como las que indican si se está tomando un salto (jump) o un salto condicional (branch), o si se está realizando una operación que involucra la memoria. Si detecta que una instrucción en la etapa de ID/EX (Instruction Decode/Execute) necesita un dato que aún está siendo procesado en una etapa posterior, la unidad activa las señales de bloqueo (o_bloqueo_pc, o_bloqueo_latch_ifid, o_bloqueo_latch_idex) para detener temporalmente el avance del pipeline. Esto permite que la instrucción que está esperando el dato pueda obtenerlo correctamente en el siguiente ciclo.

Además, si en un ciclo anterior hubo un riesgo por un salto o un branch, la unidad utiliza una señal interna (i_post_bloqueo_1) para resetear ciertas señales en el ciclo actual, asegurando que el pipeline pueda continuar su operación normal sin riesgos. En resumen, esta unidad es crucial para

mantener la integridad de la ejecución en un pipeline, evitando conflictos de datos y asegurando que cada instrucción reciba los datos correctos en el momento adecuado

Dado que el PC se actualiza durante el segundo semiciclo del clock, las instrucciones de salto (J y JAL) no requieren un control de riesgos adicional. Esto es porque, una vez que estas instrucciones llegan a la etapa de Instruction Decode (ID), el PC aún no se ha actualizado, lo que permite que se procese el nuevo valor del PC sin necesidad de detener o alterar el flujo del pipeline.

Riesgos de Datos (Data Hazards)

1. **Riesgo en Instrucciones Load (MemToReg Activo)**:
    - Ocurre cuando una instrucción de carga (load) está en la etapa **EX/MEM** y su registro de destino (rd) es el mismo que el registro fuente (rs o rt) de una instrucción que está en la etapa **ID/EX**.
    - **Problema**: El valor que se necesita en la etapa **ID/EX** (por ejemplo, rs) aún no se ha actualizado, porque la instrucción load en la etapa **EX/MEM** todavía no ha terminado de cargar el dato desde la memoria. Esto significa que si se sigue adelante sin hacer nada, la instrucción en **ID/EX** utilizará un valor desactualizado, lo que puede causar errores en la ejecución.
    - **Solución**: Se bloquea el PC (bloqueo_pc_2 = 1) y el latch IF/ID (bloqueo_ifid_2 = 1), lo que esencialmente inserta una burbuja en el pipeline. Esto permite que la instrucción load complete la carga de datos y actualice el registro antes de que la instrucción siguiente avance, evitando así que use un valor incorrecto.

Riesgos de Control (Control Hazards)

2. **Riesgo en Instrucciones de Salto o Rama (Branches y Jumps)**:
    - Ocurre cuando hay una instrucción de salto (jump) o de rama (branch) en el pipeline que puede cambiar la secuencia de ejecución del programa.
    - **Problema**: Antes de que el procesador determine si se debe tomar o no el salto, el pipeline podría continuar cargando instrucciones que no deberían ejecutarse si el salto se toma. Esto puede llevar a la ejecución incorrecta de instrucciones.
    - **Solución**: Se bloquea el PC (bloqueo_pc_1 = 1) y el latch IF/ID (bloqueo_ifid_1 = 1), lo que permite al procesador determinar si el salto o la rama debe tomarse antes de que el pipeline avance. Si se toma el salto o la rama, se actualiza el PC con la nueva dirección y se asegura de que no se ejecuten las instrucciones incorrectas que se habrían cargado en el pipeline.

Resumen

- **bloqueo_pc_2 y bloqueo_ifid_2**: Se utilizan para manejar **riesgos de datos**, bloqueando el avance del PC y el latch IF/ID para evitar que una instrucción en la etapa **ID/EX** utilice un dato desactualizado de una instrucción load que aún no ha terminado.
- **bloqueo_pc_1 y bloqueo_ifid_1**: Se utilizan para manejar **riesgos de control**, bloqueando el PC y el latch IF/ID cuando hay una instrucción de salto o rama que podría cambiar la secuencia de ejecución, asegurando que el pipeline procese las instrucciones correctas en la secuencia adecuada.

Para los jump register:
Condición: if(TakeJumpR)
Acción: Insertar burbuja en ID. Lo que implica que en el instante que ingresa la operación a ID, debo bloquear PC y latch1. En el ciclo siguiente debo desbloquearlos y mantener un reset sobre la unidad de control.

Para los branchs:
Condición: if(TakeBranch)
Acción: Insertar burbuja en ID. Lo que implica que en el instante que ingresa la operación a ID, debo bloquear PC y latch1. En el ciclo siguiente debo desbloquearlos y mantener un reset sobre la unidad de control.

Para los loads:
Condición: if(MemToReg(EX/MEM) && (rd(EX/MEM) == rs(ID/EX) II (rd(EX/MEM) == rt(ID/EX) && RegDst(ID/EX))))
Acción: Insertar burbuja en EX. Lo que implica que en el instante en que ingresa la operación a la etapa MEM, debo bloquear el PC, el latch IFID y el IDEX. En el ciclo siguiente debo desbloquearlos y mantener un reset sobre latch EXMEM. Para el correcto funcionamiento debo también implementar el R0 como 0 permanente.

El bloqueo de los latches IFID e IDEX se utiliza para manejar riesgos de control y datos, asegurando que el pipeline no avance con información incorrecta. El reset del latch EXMEM se realiza para evitar que se utilicen resultados de ejecuciones previas que no son válidos debido al riesgo detectado.

Burbuja para branches y jumps

Cuando se generan burbujas en instrucciones de salto (branches y jumps), la Unidad de Detección de Riesgos (UDR) puede bloquear un latch (latch IF/ID) que consume información para detectar riesgos. Si esto se dejara así, podría provocar un bloqueo permanente. Para solucionarlo, se utiliza la señal "Post Bloqueo 1", que se genera en el ciclo de reloj anterior y se consume en el siguiente. Al recibir esta señal, el sistema sabe que hubo un bloqueo y puede proceder a desactivar las señales correspondientes y desbloquear el Program Counter (PC) y latch IF/ID.

Para las instrucciones de carga, no es necesario implementar una señal de Post Bloqueo, porque el latch EX/MEM, que indica la presencia de riesgos, es el mismo que debe ser reseteado y funciona de manera síncrona. Esto significa que, al detectar un riesgo, se puede emitir una señal de reset que será reconocida en el siguiente ciclo. Esto simplifica el diseño.

Unidad de Debug

Entradas	● Clock ● Reset ● End of reception ● Error flag ● Recept byte [8] ● Sending flag

	● PC [32] ● Dato Reg [32] ● Dato Mem Datos [32] ● HALT signal
Salidas	Send start ● Send byte [8] ● Reset ● Reset PC ● Block ● Selector Mem Instrucciones ● Flag escritura Mem Instrucciones ● Dirección Mem Instrucciones [32] ● Dato Mem Instrucciones [32] ● Selector Unidad Registros ● Dirección Unidad Registros [5] ● Selector Mem Datos ● Dirección Mem Datos [32] ● State [4]
Sincronismo	Negedge

Esta unidad es transversal a todas las etapas y no aporta al funcionamiento como tal del procesador. Sino que funciona como una unidad que permite el control y la depuración del mismo y sus programas. Se comunica con un ordenador externo mediante UART. Funciona en negedge para que se sincronice correctamente con los latches del procesador, que trabajan en posedge.

Entre sus funciones se encuentran:
● Agregar datos a la memoria de instrucciones para poder cargar programas.
● Pausar el procesador para poder interactuar con sus memorias y registros.
● Durante el pausado puede leer el PC, los registros y memorias.
● Controlar el flujo de funcionamiento del procesador, dejándolo continuar libremente hasta el final del programa o parandolo instrucción a instrucción.

Tiene 4 modos:
● Modo espera: Mantiene el procesador bloqueado.
● Modo programa: Si al estar en espera recibe una P (ASCII), pasa al modo programa. El primer dato recibido posterior al cambio de estado será la cantidad de instrucciones a poner en la memoria de instrucciones. Los datos posteriores serán dichas instrucciones. Al completar el modo programa se retorna al modo espera.
● Modo continuo: Al estar en modo espera y recibir una C (ASCII), entramos a este modo. Ejecuta el programa hasta el final y luego imprime todos los datos de memoria, los registros y el PC.
● Modo paso a paso: Al estar en modo espera y recibir una S (ASCII), entramos a este modo. Ejecuta el programa un paso a la vez y luego de cada paso imprime todos los datos de memoria, los registros y el PC. Se avanza un nuevo paso al enviar otra S.
○ Se puede cancelar el modo paso a paso enviando una X (ASCII), o hacerlo continuó enviando una C (ASCII).

Cuenta con:

● Una señal que bloquea todos los latches para detener el procesador.
● Tiene lectura directa de la salida del latch PC.
● Tiene lectura directa de la salida 1 de la unidad de registros, así como el
control de un mux que le permite tomar control de la entrada de direcciones de la misma.
● Tiene lectura directa de la salida de la memoria de datos, así como el control de un mux que le permite tomar control de la entrada de direcciones de la misma.
● Tiene control de un mux a la entrada de la memoria de instrucciones, que controla si la dirección la dicta el PC o la unidad de debug. Controla un flag de escritura de dicha unidad y una entrada de 32b para indicar el valor a cargar.