Trabajo final

Pipeline MIPS

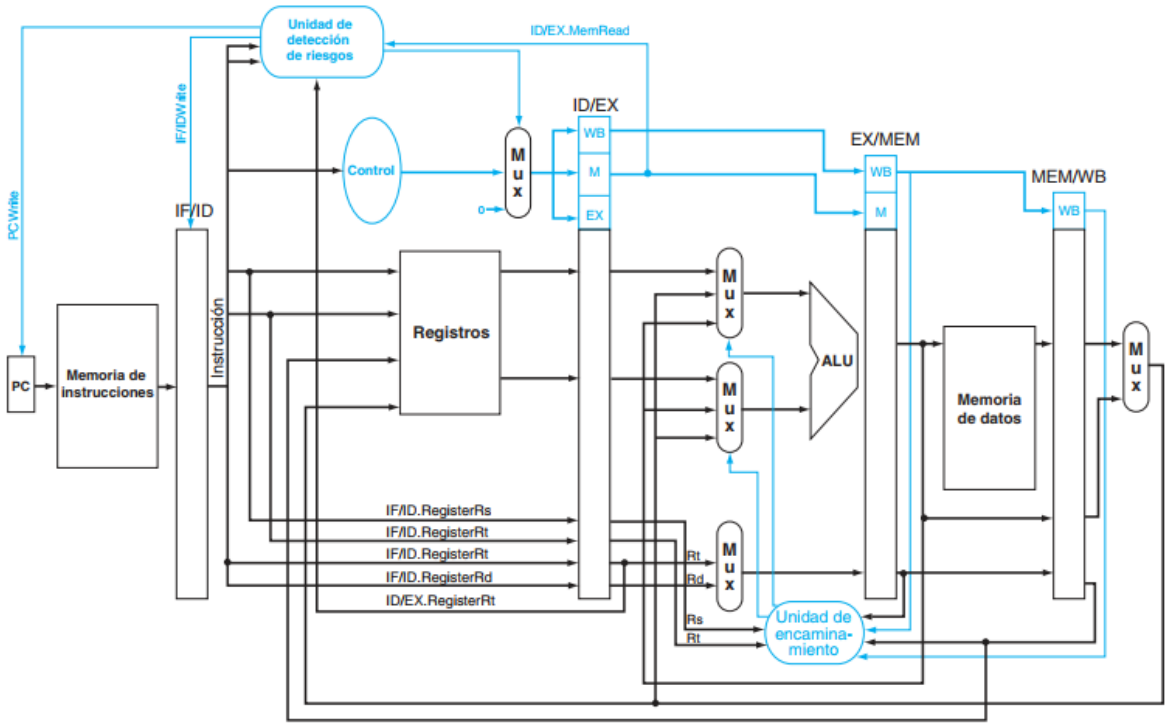
Arquitectura de Computadoras

Orionte Andres
Peschiutta Luciano

07 / 2023

Indice

Introducción



El trabajo consiste en el diseño e implementación de un procesador MIPS en una placa FPGA. El mismo debe contar con las siguientes características:

- Segmentación del procesador en las siguientes etapas:
  - IF (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
  - ID (Instruction Decode): Decodificación de la instrucción y lectura de registros.
  - EX (Execute): Ejecución de la instrucción propiamente dicha.
  - MEM (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
  - WB (Write back): Escritura de resultados en los registros.
- Implementación de las siguientes operaciones:
  - R-type
    - SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT.
  - I-Type
    - LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL.
  - J-Type

        - JR, JALR.
- Contar con soporte para los siguientes tipos de riesgo:
    - Estructurales: Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
    - De datos: Se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
    - De control: Intentar tomar una decisión sobre una condición todavía no evaluada.
- Implementación de las siguientes unidades:
    - Unidad de cortocircuitos.
    - Unidad de detección de riesgos.
- Cumplir con los siguientes requerimientos:
    - El programa a ejecutar debe ser cargado en la memoria del programa mediante un archivo ensamblado.
    - Debe implementarse un programa ensamblador.
    - Debe transmitirse ese programa mediante interfaz UART antes de comenzar a ejecutar.
    - Se debe incluir una unidad de Debug que envíe información hacia y desde la PC mediante la UART. La misma debe poder enviar mediante UART:
        - PC.
        - Contenido de los registros usados.
        - Contenido de la memoria de datos usada.
- Contar con el siguiente modo de operación:
    - Antes de estar disponible para ejecutar, el procesador está a la espera para recibir un programa mediante la Debug Unit.
    - Una vez cargado el programa, debe permitir dos modos de operación:
        - Continuo, se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo (Instrucción HALT). Llegado ese punto se muestran todos los valores indicados en pantalla.
        - Paso a paso: Enviando un comando por la UART se ejecuta un ciclo de Clock. Se debe mostrar a cada paso los valores indicados.

Diseño y desarrollo

Debido a la complejidad y el tamaño del diseño, se decidió dividirlo en dos etapas.

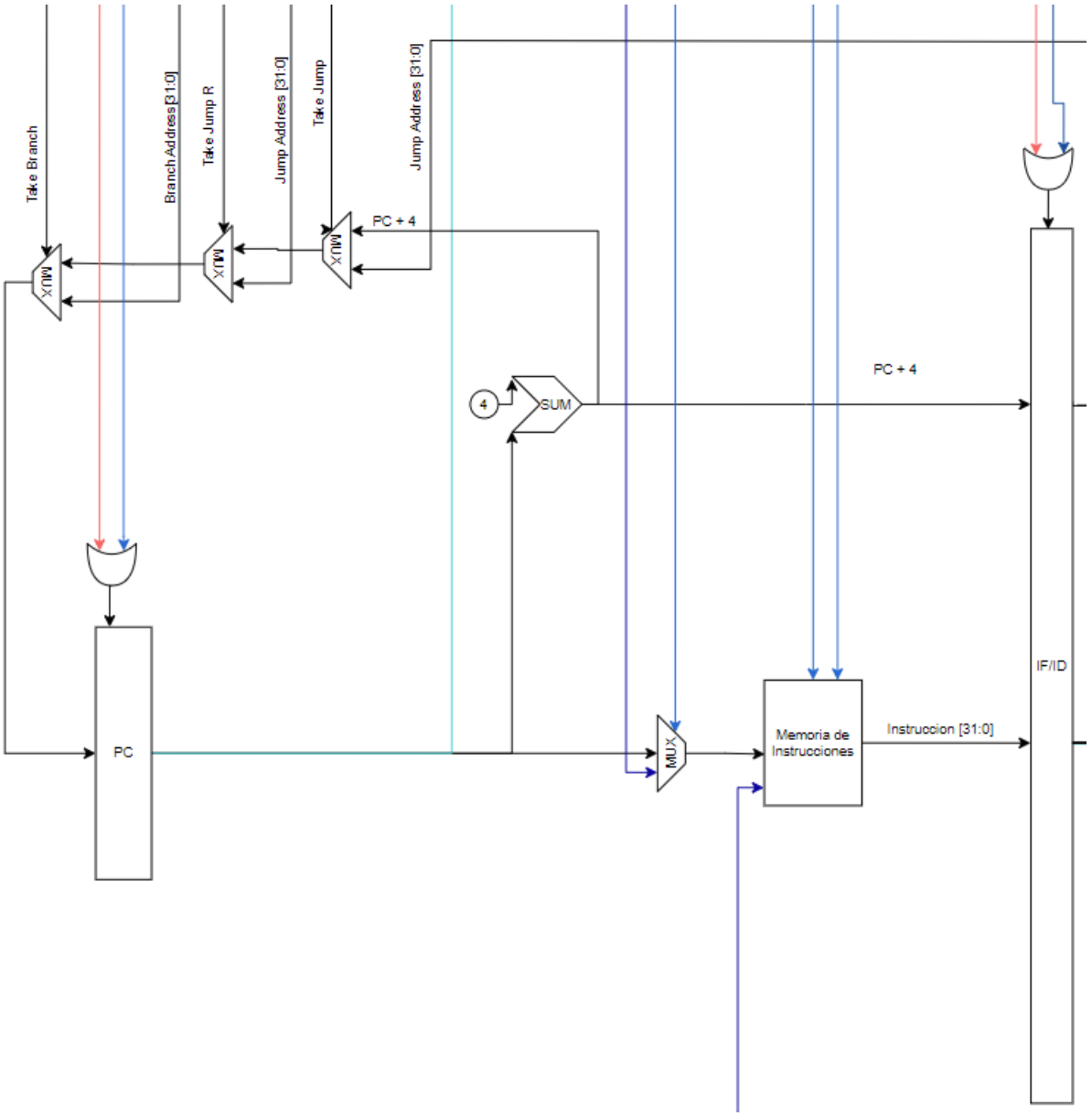
Durante la primera etapa, se hizo enfoque en conseguir un procesador segmentado, centrándonos en el modelado de las etapas y los módulos que las componen, definiendo las operaciones y algunos de los elementos. Se hicieron a un lado las unidades de debug, cortocircuito y detección de riesgos, por agregar una complejidad que podría dificultar conseguir un procesador funcional. Al final de esta etapa, se consiguió un procesador segmentado que puede ejecutar cualquier operación del set, pero de manera aislada (por no contar con herramientas para sortear los riesgos).

Durante la segunda etapa y habiendo comprobado el funcionamiento base del procesador, se modelaron y diseñaron las unidades excluidas durante la primera. Se adaptaron las partes necesarias y también se agregó el módulo necesario para la comunicación UART con el dispositivo.

A continuación, se detalla la composición final del sistema. En cada apartado se describen los módulos, así como las decisiones de diseño que se tomaron para cada uno.

Como la etapa que se encarga de tratar con las instrucciones crudas, decodificandolas para generar las respuestas deseadas es la etapa EX, dentro de ella se contemplan las cuestiones de diseño que se refieren a la decisión de los códigos de operación y el formato de las instrucciones.

Etapa Instruction Fetch (IF)



Módulos

Program Counter (PC)

Entradas	• Clock • Reset • Dirección siguiente [32] • Block
Salidas	• Dirección [32]
Sincronismo	• Negedge

Como su nombre lo indica, este módulo se encarga de indicar en qué posición del programa nos encontramos. Su salida apunta a la dirección de la memoria de instrucciones donde se aloja la próxima instrucción a ejecutar.
Durante el flanco indicado en el cuadro, pone en su salida el valor que se encuentra en la entrada, siempre que la señal Block no se encuentre activada. Si a la señal del clock se encuentra activado el reset, entonces pone su salida a cero.

El PC funciona por flanco de bajada para darle tiempo a los circuitos de actualizarse y estar en valores estables. Esto evita ingresar una burbuja en caso de, por ejemplo, un jump (ya que si bien hay una demora en reconocer la instrucción, la actualización del pc se realiza en el segundo semiciclo).

Memoria de Instrucciones

Entradas	• Clock • Reset • Dirección [32] • Dato de escritura [32] • Flag de escritura • Flag bloqueo lectura
Salidas	• Instrucción leída [32]
Sincronismo	• Posedge

Por una cuestión de capacidad de la placa de prueba la memoria consta de 256 posiciones válidas (desde la h00000000). En la realidad esto significa que la dirección ingresada será determinada por los 8 LSB.

Cada dirección de memoria corresponde a un byte, pero la salida del módulo entrega una palabra de 32 bits que consta del byte solicitado como el de menor peso de la palabra y los 3 bytes siguientes. Por lo tanto los accesos deben ser alineados y avanzar de 4 en 4 (lo que equivale a dejar los últimos dos bits de la dirección siempre en cero). De esta manera si pedimos la posición 32, se entrega una palabra conformada por [32+3, 32+2, 32+1, 32].

Se seleccionó el flanco de clock de subida, para que haya un correcto sincronismo con el PC. También es lo adecuado porque el procesador cambia de ciclo en dicho flanco, haciendo oportuna la actualización de la instrucción en ese momento.

Durante el accionamiento del clock, si está activado el flag de escritura, se escribe el dato de escritura en la posición indicada. Si no está activado, entonces se pone a la salida el dato almacenado en la posición indicada por la dirección. Si la señal de bloqueo de lectura está activada, entonces no se actualiza la salida.
Si en el accionamiento del clock, se activa la señal de reset, entonces todas las posiciones de memoria se ponen a cero, al igual que las salidas.

Latch 1

Entrada	• Clock • Reset • Block • PC+4 • Instrucción
Salidas	• PC+4 • Instrucción
Sincronismo	• Posedge

En este sistema los latches son los que se encargan de dividir las etapas unas de otras, sincronizando el avance. Como las etapas avanzan en los flancos positivos de clock, estos también lo harán.

Cuando el clock da la señal, el latch refleja en la salida lo que se encontraba en la entrada, excepto que se encuentre activa la señal de bloqueo, en cuyo caso no habrá cambio en la salida. Si en el momento de la señal de clock la señal de reset se encuentra activada, entonces las salidas se pondrán a cero. Este reset tiene prioridad sobre el resto de la lógica.

Normalmente todos los latches cumplen con su función para todas las entradas que no sean clock, reset y bloqueo, pero en este caso hay una excepción. Dado que la memoria de instrucciones es un módulo síncrono, que se activa en posedge, una

señal que pase de ella a este latch, demorará un clock más en verse reflejada a la salida del latch. Como este es un comportamiento no deseado, se optó por hacer que atraviese el latch directamente (un cable que atraviesa el módulo sin modificaciones), por lo cual quien se encargará de actualizarla a tiempo es dicha memoria. Para compensar la pérdida de capacidad de bloquear este avance a través del latch, se produjo la señal de bloqueo de lectura de la memoria, que cumple la misma función y por tanto la señal es la misma para ambas.

Elementos adicionales

Existen algunos elementos adicionales que intervienen para ayudar a proveer a la etapa de todos los comportamientos necesarios.

Mux

Hay 4 muxes en esta etapa.

Uno de ellos permite tomar control de la dirección de entrada de la memoria de instrucciones, para que la unidad de debug pueda barrer las posiciones de memoria que necesita escribir durante la carga del programa.

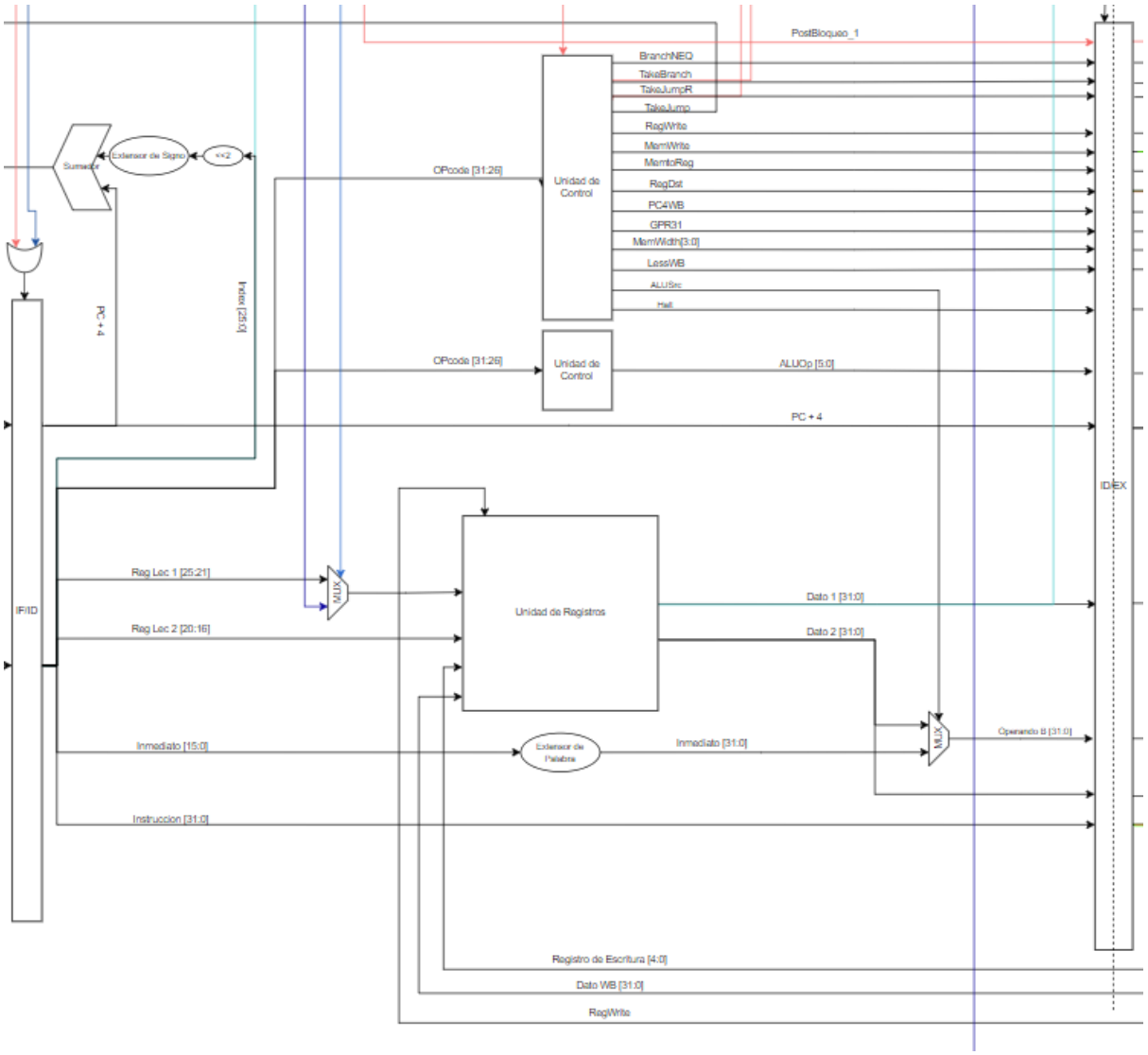
Los otros tres determinan si en el siguiente ciclo, el contador tendrá el valor del contador actual más cuatro posiciones (comportamiento por defecto (PC = PC+4)), o una dirección de memoria diferente, dada por alguna instrucción de salto (como branch o jump).

Si bien por la implementación se genera una aparente jerarquía en la prioridad de los mux encargados de seleccionar el nuevo valor del PC, en realidad solo uno de ellos puede estar activo por vez.

Sumador

En esta etapa hay un sumador, que se encarga de generar el nuevo valor que hará avanzar normalmente el PC. Sencillamente le suma 4 al valor del PC actual.

Etapa Instruction Decode (ID)



Definición de Instrucciones y códigos de operación

El código de operación de las instrucciones consta de 6 bits. Como muchas de estas operaciones comparten un gran abanico de características similares, se agrupó todo lo posible las funciones que comparten estas características, asignándoles códigos de operación con una lógica que facilite la decodificación posteriormente.

Lógica de la división de instrucciones en códigos de instrucciones

La nomenclatura utilizada para determinar los rangos de códigos de operación para cada grupo es la siguiente.

Cada operación cuenta con 6 bits, dentro de los cuales, aquellos cuya posición reciba un uno (1) o un cero (0), indican que son valores característicos (y por tanto obligatorios) del grupo, mientras que aquellos que reciban una A, indican que es un valor variable, que se utilizará para identificar entre sí a los elementos del grupo.
Luego de agrupar las operaciones, la asignación de códigos resultó de la siguiente manera:

- NOP: 000000
- R-type: 00AAAA (No incluye al 000000)
- I-type(Aritmética/Lógica): 01AAAA
- I-type(Load/Store): 10AAAA
  - Loads: 100AAA
  - Stores: 101AAA
- Jumps y Branchs: 110AAA
  - Jumps: 1100AA
  - Branchs: 1101XA
- HALT: 111111

Dada que la definición de los códigos de operación exactos genera un cierto impacto en la forma en cómo se decodifica por parte de la unidad de control, al momento del desarrollo se tomó la decisión final en base a tablas confeccionadas que ayudan a representar las relaciones entre las operaciones, sus códigos y las correspondientes señales que desencadenan.
Por ese motivo, la disposición final se encuentra en la sección dedicada al desarrollo de la unidad de control.

Módulos

Unidad de Registros

Entradas	• Clock • Reset • Registro de lectura 1 [5] • Registro de lectura 2 [5] • Registro de escritura [5] • Dato de escritura [32] • Flag de escritura • Bloqueo de escritura
Salidas	• Dato leído 1 [32] • Dato leído 2 [32]
Sincronismo	• Posedge • Negedge

Dentro de este módulo hay 32 registros, todos ellos pueden leerse y escribirse, a excepción del registro 0, que siempre valdrá cero.

Cada registro se escribe durante su flanco designado, seleccionandolo mediante su dirección en Registro de escritura e ingresando el dato en Dato de escritura. Para que la escritura se realice, también debe estar activado el Flag de escritura y no debe estar activado el Bloqueo de escritura.

El registro 0 se utiliza como registro nulo (necesario para un correcto funcionamiento de las burbujas introducidas para salvar riesgos). Esto significa que siempre vale 0 y no se puede escribir (Intentarlo no causará ningún error, solo no se verá reflejado resultado alguno). Si bien no es posible escribirlo, está prohibido utilizarlo como registro de escritura porque podría causar inconsistencias en los resultados. Este caso se desarrolla en la sección dedicada a la unidad de cortocircuito.

La lectura de los registros se hace durante el flanco de clock correspondiente, ingresando las direcciones de los registros que se desean obtener en las entradas Registro de lectura 1 y 2. Los datos leídos estarán disponibles en Dato leído 1 y 2.

Dada la necesidad de que en un mismo ciclo de clock se realicen lecturas y escrituras, fue necesario llevar a cabo las acciones en dos semiciclos diferentes. Se escogió realizar la escritura en el flanco positivo porque de esta manera un dato escrito en un ciclo puede ser leído en el mismo ciclo de reloj, en lugar de esperar al siguiente.

Unidad de Control

Entradas	• Reset • OPcode, 6 bits
Salidas	• RegDst • RegWrite • ALUSrc • MemWrite • MemtoReg • PC4Wb • GPR31 • MemWidth [4] • LessWB • TakeJump • TakeJumpR • TakeBranch • BranchNEQ • HALT • ALUOP
Sincronismo	• Asíncrona

Dada la necesidad de que la unidad de control genere las señales de control, así como la de operación de la ALU, basándose en el código de operación de la instrucción, se tomaron dos decisiones:

En primer lugar, para no generar un circuito de decodificación de lógica muy compleja, se separa la unidad de control en dos módulos, uno encargado de la generación de señales control y otro solo de la generación del ALUOP.

En segunda instancia, se desarrollan en este apartado tanto la lógica como la definición de los códigos de operación de las instrucciones, así como la sintaxis de la instrucción completa.

Señales de control

Nombre de la señal	Descripción	Señal desactivada (0)	Señal activada (1)
RegDst	Controla la procedencia del registro de destino. También aprovechada para desactivar Forward B de Unidad de Cortocircuito.	El destino de escritura proviene del campo **rt** (bits 20:16 de instrucción).	El destino de escritura proviene del campo **rd** (bits 15:11 de instrucción).
RegWrite	Controla si habrá o no escritura.	No hay escritura.	Hay escritura
ALUSrc	Controla de donde proviene el segundo operando de la ALU.	Proviene del segundo registro leído del banco de registros.	Proviene de la parte extendida de la instrucción.
MemWrite	Controla si habrá escritura o lectura de memoria.	La dirección indicada será leída.	La dirección indicada será escrita.
MemtoReg	Controla de donde viene el valor a escribir en los registros.	Proviene de la ALU.	Proviene de la Memoria de Datos.
PC4WB	Controla de donde viene el valor a escribir en los registros.	Determinado por MemtoReg.	Toma el valor de PC+4 traído por los latches.
GPR31	Determina que el registro seleccionado para escritura es el 31 (solo útil para la instrucción JAL).	Toma el valor determinado por el código de instrucción	Toma el valor de GPR31.

MemWidth [4]	Bus de 4 señales que controlan la cantidad de bytes que se leen o guardaran para cada alineación de memoria (3 bits) y un bit extra que indica si la operación es sobre un palabra signada o no.	• Bit [3]:
		  ○ 0: no signado.
		  ○ 1: signado.
		• Bits [2:0]:
		  ○ 001: 8b.
		  ○ 010: 16b.
		  ○ 100: 32b.	

LessWB	Selecciona si el valor útil de la operación es el resultado o el flag de carry de la ALU (útil para SLT y SLTI).	Toma el resultado de la ALU.	Toma el valor carry de la ALU.
Añadidas para jumps y branchs:			
TakeJump	Indica si hay un Jump o JAL.	No hay jump.	Hay Jump.
TakeJumpR	Indica si hay un Jump Register o JALR.	No hay Jump R.	Hay Jump R.
TakeBranch	Indica si hay BEQ o BNQ.	Hay un branch.	No hay un branch.
BranchNEQ	Indica el tipo de branch.	Es EQ (equal).	Es NEQ (non-equal).
Añadida para detener el procesador:			
HALT	Indica la presencia de la señal HALT.	No se encontró un HALT	Se encontró un HALT
Añadidas para ALU:			
ALUOP (6 bits):	Selector de operación de la ALU.	Sus valores posibles están detallados en el módulo ALU, así como la función de cada uno.	

Códigos de operación, salida de ALUOP y formato de instrucción

Tipo	Nombre	OPCode	ALUOP	Formato de instrucción	Comportamiento
NULO	NOP	000000	xxxxxx	OP(6)+**F**(26)	NULO
R	ADDU	000001	000001	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs + rt
R	SUBU	000010	000010	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs - rt
R	AND	000011	000011	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs AND rt
R	OR	000100	000100	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs OR rt
R	XOR	000101	000101	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs XOR rt
R	NOR	000110	000110	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs NOR rt
R	SRAV	001001	001000	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs >> rt (Aritmético)
R	SRLV	001010	010000	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs >> rt (Logico)
R	SLLV	001100	100000	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< rs << rt (Logico)
R	SLT	001000	000010	OP(6)+rs(5)+rt(5)+rd(5)+**F**(11)	rd <<< (rs < rt)
I(AL)	ADDI	010001	000001	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs + inmediato
I(AL)	SUBI	010010	000010	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs - inmediato
I(AL)	ANDI	010011	000011	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs AND inmediato
I(AL)	ORI	010100	000100	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs OR inmediato
I(AL)	XORI	010101	000101	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs XOR inmediato
I(AL)	LUI	010111	000111	OP(6)+**F**(5)+rt(5)+inmed(16)	rt <<< (inmediato << 16)
I(AL)	SRA	011001	001000	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs >> inmed (Aritmético)
I(AL)	SRL	011010	010000	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs >> inmed (Logico)
I(AL)	SLL	011100	100000	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< rs << inmed (Logico)
I(AL)	SLTI	011000	000010	OP(6)+rs(5)+rt(5)+inmed(16)	rt <<< (rs < inmediato)
I(LS)	LB	100000	000001	OP(6)+base(5)+rt(5)+offset(16)	rt <<< memoria[base+offset] (**8+ExSi**)
I(LS)	LBU	100001	000001	OP(6)+base(5)+rt(5)+offset(16)	rt <<< memoria[base+offset] (**8**)
I(LS)	LH	100010	000001	OP(6)+base(5)+rt(5)+offset(16)	rt <<< memoria[base+offset] (**16+ExSi**)
I(LS)	LHU	100011	000001	OP(6)+base(5)+rt(5)+offset(16)	rt <<< memoria[base+offset] (**16**)
I(LS)	LW	100100	000001	OP(6)+base(5)+rt(5)+offset(16)	rt <<< memoria[base+offset] (**32**)
I(LS)	SB	101001	000001	OP(6)+base(5)+rt(5)+offset(16)	memoria[base+offset] (**8**) <<< rt
I(LS)	SH	101010	000001	OP(6)+base(5)+rt(5)+offset(16)	memoria[base+offset] (**16**) <<< rt
I(LS)	SW	101100	000001	OP(6)+base(5)+rt(5)+offset(16)	memoria[base+offset] (**32**) <<< rt
J	J	110000	000000	OP(6)+dirección(26)	PC <<< (dirección<<2)
J	JAL	110001	000000	OP(6)+dirección(26)	GPR[31] <<< PC+4 PC <<< (dirección<<2)
J	JR	110010	000000	OP(6)+rs(5)+**F**(21)	PC <<< rs
J	JALR	110011	000000	OP(6)+rs(5)+**F**(5)+rd(5)+**F**(11)	rd <<< PC + 4 PC <<< rs
B	BEQ	110100	000010	OP(6)+rs(5)+rt(5)+offset(16)	if (rs == rt) then branch
B	BNE	110101	000010	OP(6)+rs(5)+rt(5)+offset(16)	if (rs != rt) then branch
STOP	HALT	111111	xxxxxx	OP(6)+**F**(26)	STOP

Referencia: *__F__:Bits libres. *__ExSi__: Extensión de signo.

Salida de señales de la unidad de control

Instrucción	OPCode	Reg Dst	Reg Write	ALU Src	Mem Write	Mem to Reg	PC 4 WB	GPR 31	Mem Width	Less WB	Take Jump	Take Jump R	Take Branch	Branch NEQ
NOP	000000	0	0	0	0	0	0	0	xxxx	0	0	0	0	0
Tipo R (SLT)	001000	1	1	0	0	0	0	0	xxxx	1	0	0	0	0
Tipo R	-(-1) 00XXXX	1	1	0	0	0	0	0	xxxx	0	0	0	0	0
Tipo I (Inm)(SLT)	011000	0	1	1	0	0	0	0	xxxx	1	0	0	0	0
Tipo I (Inm)	-(-1) 01XXXX	0	1	1	0	0	0	0	xxxx	0	0	0	0	0
Tipo I: LBU	100000	0	1	1	0	1	0	0	0001	0	0	0	0	0
Tipo I: LB	100001	0	1	1	0	1	0	0	1001	0	0	0	0	0
Tipo I: LHU	100010	0	1	1	0	1	0	0	0010	0	0	0	0	0
Tipo I: LH	100011	0	1	1	0	1	0	0	1010	0	0	0	0	0
Tipo I: LW	100100	0	1	1	0	1	0	0	1100	0	0	0	0	0
Tipo I: SB	101001	0	0	1	1	0	X	X	1001	0	0	0	0	0
Tipo I: SH	101010	0	0	1	1	0	X	X	1010	0	0	0	0	0
Tipo I: SW	101100	0	0	1	1	0	X	X	1100	0	0	0	0	0
Jump: J	110000	X	0	X	0	0	X	X	xxxx	X	1	0	0	0
Jump: JAL	110001	X	1	X	0	0	1	1	xxxx	X	1	0	0	0
Jump: JR	110010	X	0	X	0	0	X	X	xxxx	X	0	1	0	0
Jump: JALR	110011	1	1	X	0	0	1	0	xxxx	X	0	1	0	0
Branch: BEQ	1101X0	1	0	0	0	0	X	X	xxxx	X	0	0	1	0
Branch: BNE	1101X1	1	0	0	0	0	X	X	xxxx	X	0	0	1	1

Dentro de las tablas se encuentran identificadas, mediante coloreado de casillas, resaltado de caracteres y coloreado de caracteres, relaciones muy importantes entre las salidas de las señales y los códigos de operación de las instrucciones y el ALUOP. Estas relaciones son útiles para simplificar y optimizar el circuito decodificador.

El resaltado verde de algunos elementos de la señal RegDst se debe a que se le da ese valor para aprovechar esta señal en la desactivación del Forward B en la unidad de cortocircuitos (en lugar de dejarlo como indistinto).

El resaltado marrón de algunos elementos de la señal MemToReg se debe a que se le da ese valor para aprovechar esta señal en reconocimiento de instrucciones load para la Unidad De Detección de Riesgos (en lugar de dejarlo como indistinto).

Latch 2

Entradas	• Clock • Reset • Block • PC+4 [32] • Dato 1 [32] • Dato 2 [32] • Operando B [32] • Instrucción [32] • ALUOP [6] • RegDst • RegWrite • MemWrite • MemtoReg • PC4WB • GPR31 • MemWidth [4] • LessWB • TakeJumpR • TakeBranch • BranchNEQ • HALT • PostBloqueo
Entrada	• PC+4 [32] • Dato 1 [32] • Dato 2 [32] • Operando B [32] • Instrucción [32] • ALUOP [6] • RegDst • RegWrite • MemWrite • MemtoReg • PC4WB • GPR31 • MemWidth [4] • LessWB • TakeJumpR • TakeBranch • BranchNEQ • HALT • PostBloqueo
Sincronismo	• Posedge

Ante la presencia de un flanco positivo de clock, las entradas se reflejan en su contraparte a la salida, excepto en el caso en que el bloqueo esté activado. Frente a un reset, todas las salidas se establecen en cero (con prioridad ante bloqueo).

Elementos adicionales

Mux

En esta etapa nos encontramos con 2 muxes.

Uno, propio del funcionamiento del MIPS, se encuentra anterior a la entrada del latch. Este selecciona si el operando B, que pasará a la ALU en el siguiente ciclo, proviene de la lectura de un registro o de un inmediato obtenido directamente de la instrucción.

El segundo se ubica ante la entrada de direcciones de la unidad de registros, y permite que en determinados momentos la unidad de debug sea la que indique las direcciones a ser leídas, para poder hacer una lectura de los registros y enviarlos mediante UART.

Sumador

El sumador de esta etapa se encarga de sumar la dirección PC+4 con el valor provisto por una instrucción, este resultado será la dirección de salto en caso de que haya un jump.

Shift

Este elemento hace un shift de dos posiciones hacia la izquierda para alinear la dirección indicada con el jump. Permitiendo que ante la orden de un jump de 1 instrucción, en realidad la posición se cambie 4 lugares, manteniendo el alineamiento del PC, que siempre debe estar en múltiplos de 4.

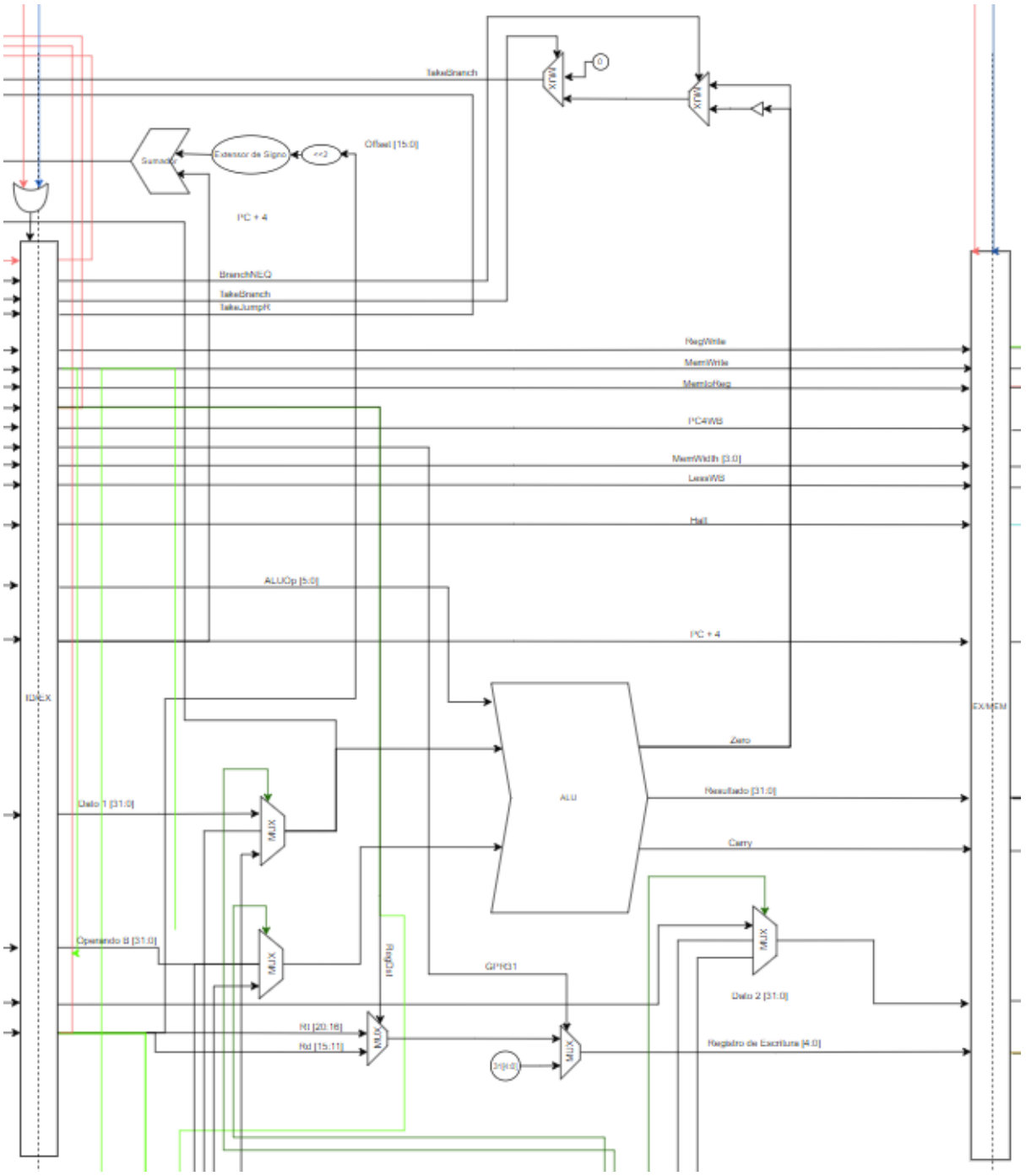
Extensor de palabra

Convierte un bus de x elementos a x+n elementos, sencillamente completando los bits agregados (los MSB) con ceros. En este caso el bus de 16 pasa a uno de 32.

Extensor de signo

También amplía el tamaño del bus, pero en este caso, los bits completados dependen del signo de la palabra original, para mantener el mismo valor.

Etapa Execution (EX)



Módulos

ALU

Entradas	• Operando A [32] • Operando B [32] • ALUOp [6]
Salidas	• Resultado [32] • Carry flag • Zero flag
Sincronismo	• Asíncrona

La ALU soporta un conjunto de operaciones diferentes, para seleccionar cual de ellas queremos llevar a cabo debemos ingresar el ALUOP correspondiente.

Operaciones y códigos de operación de la ALU

- ADD 000001
- SUB 000010
- AND 000011
- OR 000100
- XOR 000101
- NOR 000110
- SLL16 000111
- SRA 001000
- SRL 010000
- SLL 100000

A excepción de SLL16, todas las operaciones son conocidas, así que no las detallaremos.

Tomamos la decisión de implementar la operación SLL16 en la ALU para poder realizar la operación LUI sin demasiadas complicaciones. Su función es tomar un solo operando a la entrada y hacerle un SLL de 16 bits independientemente del operando 2.

Tipos de operación de ALU para cada tipo de instrucción

Aquí detallamos qué tipo de operación de la ALU se requiere para cada una de las instrucciones:

- ADD:
    - R: ADDU
    - I: LB; LBU; LH; LHU; LW; SB; SH; SW; ADDI
- SUB:
    - R: SUBU; SLT
    - I: SUBI; SLTI
    - Branch: BEQ; BNQ
- AND:
    - R: AND
    - I: ANDI
- OR:
    - R: OR
    - I: ORI
- XOR:
    - R: XOR
    - I: XORI
- NOR:
    - R: NOR
- SRA:
    - R: SRAV
    - I: SRA
- SRL:
    - R: SRLV
    - I: SRL
- SLL:
    - R: SLLV
    - I: SLL
- SLL16:
    - I: LUI
- Sin intervención de la ALU:
    - J; JAL; JR; JARL

Latch 3

Entradas	• Clock • Reset • Reset (sin prioridad) • Block • PC+4 [32] • Resultado [32] • Carry • Dato 2 [32] • Registro de Escritura [5] • RegWrite • MemWrite • MemtoReg • PC4WB • MemWidth [4] • LessWB • Halt
Salidas	• PC+4 [32] • Resultado [32] • Carry • Dato 2 [32] • Registro de Escritura [5] • RegWrite • MemWrite • MemtoReg • PC4WB • MemWidth [4] • LessWB • Halt
Sincronismo	• Posedge

Este latch funciona igual al resto de latches. Al llegar la señal de clock, todas las señales de entrada a excepción del clock, el bloqueo y los dos reset se ven reflejados en su contraparte a la salida. Posee una señal de bloqueo que impide el cambio de valores a la salida, y una señal de reset que pone todas las señales a cero, con prioridad sobre el resto de comportamientos.

Pero además de lo anterior, este latch posee una segunda señal de entrada de reset, que manifiesta el mismo comportamiento, pero no tiene prioridad sobre la señal de bloqueo. Este reset se utiliza para lograr una correcta implementación de las burbujas, permitiendo que el latch se resetee en un momento específico, por lo que si el procesador (y por tanto el latch) está bloqueado, no se reseteará sino hasta que el bloqueo se desactive.

Elementos adicionales

Mux

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.

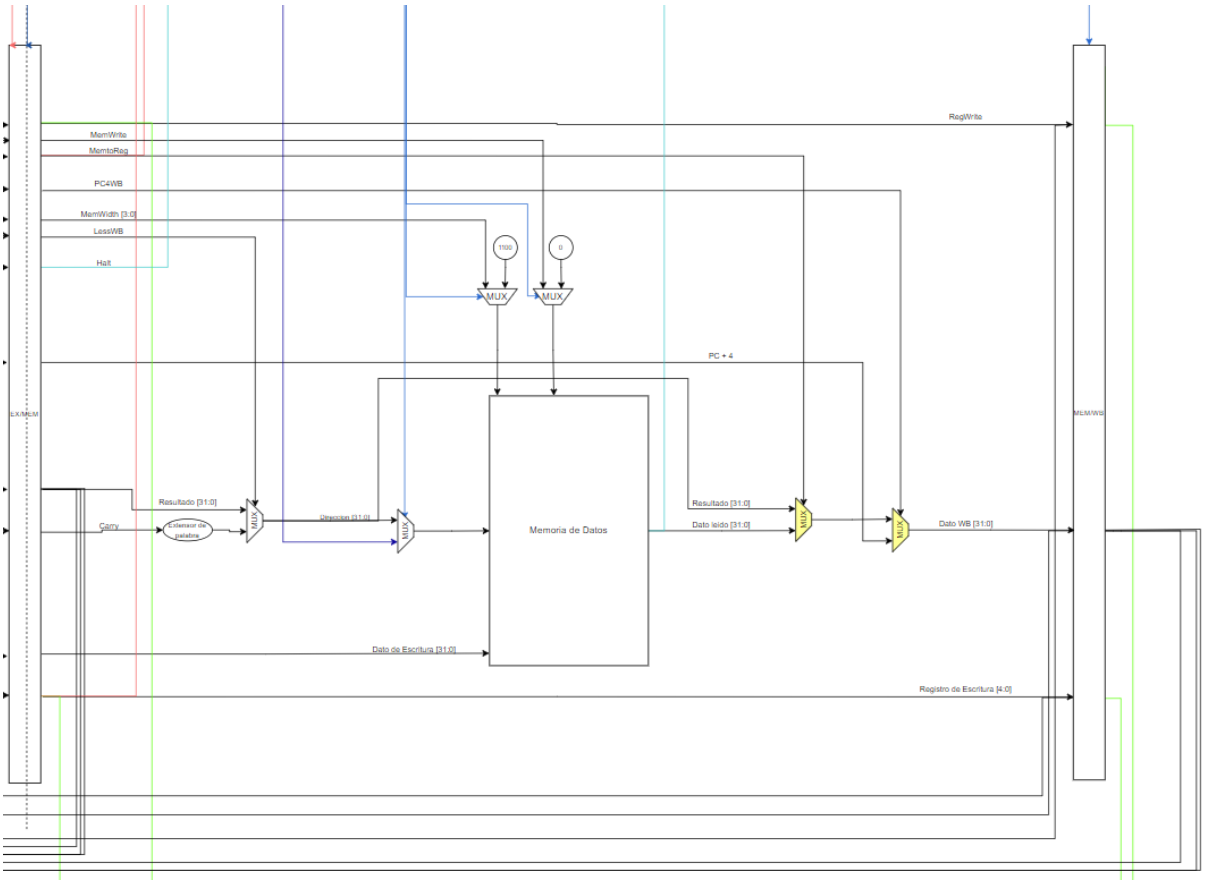
Dos de los muxes de 2 entradas que se encuentran en la parte superior del diagrama y están conectados en serie a la salida del flag de zero de la ALU, son los encargados de determinar si hubo un branch, y en caso que lo haya, si la condición se cumplió o no.

Finalmente nos encontramos con otros dos muxes de dos entradas en la parte inferior del diagrama. Estos 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 Memory Access (MEM)



Módulos

Memoria de Datos

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

Por la limitación en la capacidad de la placa, la memoria consta de 128 posiciones válidas, desde la 00000000h. Cada dirección de memoria corresponde a un byte, pero la salida del módulo entrega una palabra de 32 bits.

Existen múltiples modos de operación:

- 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].
- Podemos seleccionar si las lecturas serán signadas o no signadas (para los casos de 8 y 16 bits). En caso de ser signada, entonces si la palabra es un número negativo, los bits sobrantes de la salida se completarán con unos, para que la palabra de 32 siga interpretándose como un número negativo (por el modo de funcionamiento de la representación numérica del complemento a 2).

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.

Mediante los bits de la señal de entrada MemWidth se selecciona el modo de operación:

- MemWidth [3]:
    - 0: no signado.
    - 1: signado.
- MemWidth [2:0]:
    - 001: 8b.
    - 010: 16b.
    - 100: 32b.

Mediante la señal MemWrite se selecciona si la operación a llevar a cabo será una lectura o una escritura.

Latch 4

Entradas	• Clock • Reset • Block • DatoWB [32] • RegEsc [5] • RegWrite
Salidas	• DatoWB [32] • RegEsc [5] • RegWrite
Sincronismo	• Posedge

Ante la presencia de un flanco positivo de clock, las entradas se reflejan en su contraparte a la salida, excepto en el caso en que el bloqueo esté activado. Frente a un reset, todas las salidas se establecen en cero (con prioridad ante bloqueo).

Elementos adicionales

Mux

En esta etapa nos encontramos con 6 muxes.

Tres de ellos, el primero de la izquierda en el diagrama y los dos más a la derecha, se encargan de seleccionar la fuente de la señal que llegará a la salida de la etapa, dependiendo del tipo de instrucción que se encuentre en la etapa en el momento.

Los dos que se encuentran sobre la memoria 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.

Extensor de palabra

Se encarga de hacer que la salida carry de la ALU se convierta en un bus de 32 bits para adecuarse al tamaño de palabra que recorre el circuito.

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.

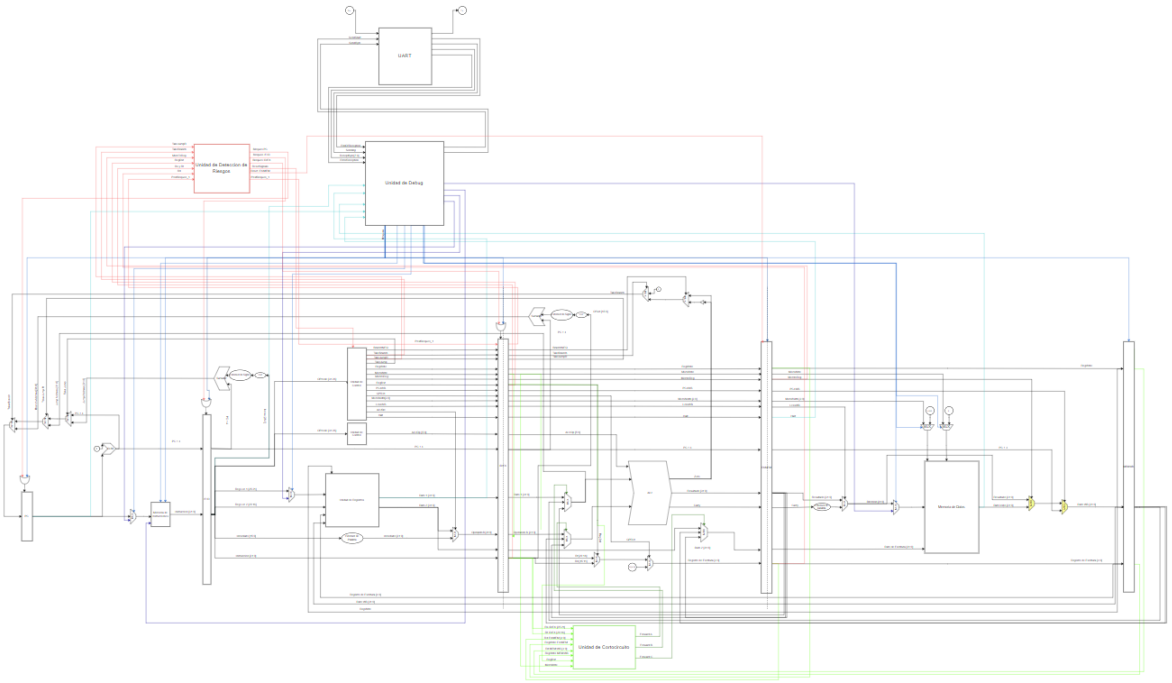
En esta etapa, la salida del latch 4 retorna hacia atrás, e indica si habrá escritura en los registros y cuál será el dato y el registro escrito en caso de que lo haya.

Como la salida del latch 4 y la entrada de escritura de la unidad de registros funcionan ambos en posedge, ocurre un problema de sincronización si los conectamos. Debido a esto, la unidad de registros consume los datos a la entrada del latch 4, evitando este retardo indeseado.

Dado que el latch cumple también la función de detener el procesador ante la señal de bloqueo, esto nos trae un nuevo problema, que solucionamos agregando a la unidad de registros un bloqueo de escritura, que se acciona en conjunto con el bloqueo del latch, y suple su función.

El latch 4 sigue estando presente en el diseño porque algunas otras unidades aún necesitan obtener los datos a su salida.

MIPS



El diseño completo del MIPS integra e interconecta todas las etapas desarrolladas anteriormente, a la vez que añade módulos que son transversales a muchas de ellas.

Módulos

Unidad de cortocircuito

Entradas	• Rs (ID/EX) • Rt (ID/EX) • Rd (EX/MEM) • Rd (MEM/WB) • RegWrite (EX/MEM) • RegWrite (MEM/WB) • RegDst • MemWrite (ID/EX)
Salidas	• Forward A [1:0] • Forward B [1:0] • Forward C [1:0]
Sincronismo	• 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.

El comportamiento anterior es la causa de porque no debe usarse el registro 0 como registro de escritura. Si bien, al saber que ese registro no puede ser escrito, sería esperable que usarlo como escritura no traería ninguna consecuencia, es cierto que la unidad de cortocircuito puede interceptarlo cuando avanza por el pipeline, momento en el cual su valor puede ser cualquiera.

Lógica de control de la unidad de cortocircuito:

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 (UDR)

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	• Asíncrona

Esta unidad se encarga de solucionar riesgos de control, y también ayuda a la unidad de cortocircuito a solucionar el riesgo de lectura después de escritura del Load.

Dado que el PC funciona durante el segundo semiciclo del clock, tanto la instrucción J como JAL no necesitarán control de riesgos, porque en el medio ciclo entre el cual la instrucción llega a ID y el PC cambia, hay tiempo suficiente para procesar el nuevo valor del PC.

Esto nos deja con la necesidad de detectar los riesgos de:

- Loads
- Branches
- Jump registers

Para los riesgos de jump registers y branches siempre se mete una burbuja en la etapa id (en los jump para dar tiempo a que se lean los registros, y en los branchs para dar tiempo que se lean los registros y se evalúe la condición).

Para los loads se debe meter una burbuja en la etapa de EX, siempre que se cumpla que el rd de la operación que transcurre EX sea igual a rs o rt de la operación que transcurre por ID.
Para poder decidir si una instrucción load es riesgosa necesitamos saber que una instrucción load atraviesa la etapa MEM (mediante la señal MemToReg de dicha etapa), saber qué registros utilizará como fuente la operación siguiente (mediante la

señal RegDst sabemos si rs y rt, o solo rs), y compararlos con el registro que está utilizando el load.

Lógica de control de la unidad de control de riesgos:

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) || (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 latch1 y el latch2. En el ciclo siguiente debo desbloquearlos y mantener un reset sobre latch3. Para el correcto funcionamiento debo también implementar el R0 como 0 permanente.

Extras:

La necesidad de hacer un reset en el momento posterior al bloqueo radica en que, debido a que solo una parte del procesador se detiene, al siguiente ciclo, la etapa que ya procesó lo que debía, recibirá nuevamente los mismos, por lo que repetirá el procedimiento. El reset evita esto, generando la burbuja propiamente dicha.

La razón por la cual es necesario que R0 siempre valga cero es que al hacer un reset al latch3, se pone el RegDst a cero y el resultado a cero, por lo cual si alguna operación hace uso de ese registro y la unidad de cortocircuitos lo intercepta, tomará como cero el valor de R0. Como no hay escape sencillo de esta situación, es preferible hacer que R0 solo pueda tomar el valor de 0. Esto aporta otra utilidad, y es que podemos usar R0 como base 0 para los cálculos de direcciones con total seguridad de que su valor siempre será ese. Evitando la necesidad de poner a cero un registro en el caso de necesitar tal valor.

Dado que durante las burbujas para branches y jumps, la UDR bloquea el latch1 del cual consume la información para detectar dichos riesgos, se produciría un bloqueo permanente. Para evitar esto, utilizaremos la señal Post Bloqueo 1, que sale de la unidad y la volvemos a consumir en ciclo de reloj siguiente (haciéndola pasar para eso por el latch2). Recibir esta señal significará que ya hubo un bloqueo en el ciclo anterior, y por tanto ahora se debe llevar a cabo la acción siguiente (desactivar todas las señales de la etapa y desbloquear PC y latch 1).

Para el caso de los loads, no necesitamos hacer uso de una señal de Post Bloqueo porque el módulo que nos indica la presencia del riesgo (latch 3) es el mismo que debe ser reseteado y es síncrono. Esta sincronicidad nos permite sacar la señal de reset en el momento de detectar el riesgo, sabiendo que por su sincronicidad el módulo la detectará en el ciclo siguiente.

Tomamos la decisión de implementar esta solución, en lugar de ponerle estado a la unidad de detección de riesgos y que se desbloquee tras el transcurso de un clock, porque podría ocurrir que otra unidad bloquee ciertas partes del circuito (como la unidad de debug) y se genere una desincronización. Otra opción es permitir que las otras unidades bloqueen a esta, tal como hacen con los latches, pero dado que complicaría la lógica a medida que escala el procesador, nos parece más correcto que toda sincronía esté provista por el accionar de los latches.

Jerarquía:

Dado que puede darse una situación donde entren simultáneamente un riesgo de tipo branch o jump r, en conjunto con uno de tipo load, es importante aclarar que los del tipo load tienen prioridad porque se dan en etapas más avanzadas del pipeline y deben resolverse antes.

Dado el modo en que están implementadas las burbujas, la jerarquía se consigue naturalmente, sin necesidad de agregar nada.

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.

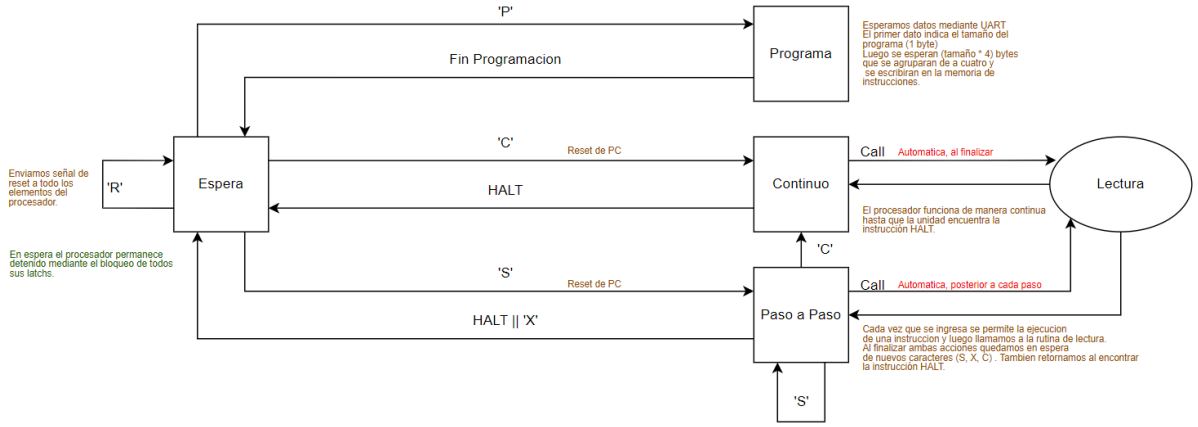
Tiene 4 modos:

- 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.

Diagrama de estados



UART

Módulo que proveé comunicación UART. Puede ajustarse para alcanzar diversos Baud Rates.

Su funcionamiento fue detallado en el Trabajo Práctico 2.

Compilador

Esta pieza de software, escrita en python, recibe como entrada un archivo con el programa escrito en assembler y hace una traducción.

La traducción originalmente era a binario, pero dado que el software utilizado para comunicarse mediante UART con la placa (RealTerm) provee un soporte más adecuado para el envío de ASCII y números decimales, finalmente se optó por añadir una salida donde la traducción sea a decimal.

Sintaxis del assembler

La sintaxis para las operaciones las encuentran debajo, en el listado de instrucciones, y se deben respetar las comas.
Cuando se encuentra R-x, significa que en dicha posición se debe indicar un registro (entre R0 y R31).
Cuando no se encuentra el prefijo R-, se debe colocar el número en decimal, por ejemplo para un inmediato en ADDI o el destino en un J (jump).

Listado de Instrucciones

Instrucciones Matemáticas y Lógicas:

- Suma:
  - ADDU: **ADDU R-Destino, R-Operando, R-Operando**
  - ADDI: **ADDI R-Destino, R-Operando, Inmediato**
- Resta:
  - SUBU: **SUBU R-Destino, R-Operando, R-Operando**
  - SUBI: **SUB R-Destino, R-Operando, Inmediato**
- And:
  - AND: **AND R-Destino, R-Operando, R-Operando**
- Or:
  - OR: **OR R-Destino, R-Operando, R-Operando**
- Xor:
  - XOR: **XOR R-Destino, R-Operando, R-Operando**
- Nor:
  - NOR: **NOR R-Destino, R-Operando, R-Operando**
- SRA:
  - SRAV: **SRA R-Destino, R-Palabra, R-Desplazamiento**
  - SRA: **SRA R-Destino, R-Palabra, Desplazamiento**
- SRL:
  - SRLV: **SRLV R-Destino, R-Palabra, R-Desplazamiento**

    - SRL: **SRL R-Destino, R-Palabra, Desplazamiento**
- SLL:
    - SLLV: **SLLV R-Destino, R-Palabra, R-Desplazamiento**
    - SLL: **SLL R-Destino, R-Palabra, Desplazamiento**
- SLT:
    - SLT: **SLT R-Destino, R-Operando, R-Operando**
    - SLTI: **SLTI R-Destino, R-Operando, Operando**
- LUI:
    - LUI: **LUI R-Destino, Palabra(16bits)**

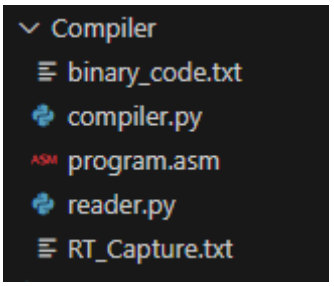
Instrucciones Load y Store:

- Load:
    - LB: **LB R-Destino, R-Base, Offset(16bits)**
    - LBU: **LBU R-Destino, R-Base, Offset(16bits)**
    - LH: **LH R-Destino, R-Base, Offset(16bits)**
    - LHU: **LHU R-Destino, R-Base, Offset(16bits)**
    - LW: **LW R-Destino, R-Base, Offset(16bits)**
- Store:
    - SB: **SB R-Fuente, R-Base, Offset(16bits)**
    - SH: **SH R-Fuente, R-Base, Offset(16bits)**
    - SW: **SW R-Fuente, R-Base, Offset(16bits)**

Instrucciones Branch y Jump:

- Jump:
    - J: **J Destino(26bits)**
    - JAL: **JL Destino(26bits)**
    - JR: **J R-Destino**
    - JALR: **JL R-Destino, R-Link**
- Branch:
    - BEQ: **BEQ Destino(16b), R-Minuendo, R-Sustraendo**
    - BNQ: **BNQ Destino(16b), R-Minuendo, R-Sustraendo**

Modo de uso



Para realizar una traducción, debemos escribir el código assembler en el archivo *program.asm*, y lo guardamos. Luego ejecutamos el compilador *"python3 compiler.py"* y este nos generará el archivo *binary_code.txt* con la traducción y también la imprimirá en la consola.
El compilador también genera la salida en hexadecimal y decimal, se pueden activar y desactivar fácilmente comentando y descomentando sus respectivas funciones ubicadas al final del código.

Extras

Con el fin de facilitar la lectura de los registros recibidos por UART a la hora de usar la placa de manera física, decidimos realizar el programa *readr.py* el cual recibe los 520 caracteres en el archivo RT_Capture.txt y por consola los muestra formateados en hexadecimal junto a su etiqueta correspondiente.
Los 520 caracteres corresponden al PC, el banco de registros, y la memoria de datos.

Ejemplo:

Programa:

- LW 25 2 #0
- ADD 26 25 #100
- SW 26 2 #1

Salida del compilador:

- 100100 11001 00010 0000000000000000
- 010001 11010 11001 0000000100000000
- 101100 11010 00010 0000000000000001

Implementación y desempeño

Implementación en FPGA

Hardware

La implementación se llevó a cabo en una placa de desarrollo Basys 3, que cuenta con el chip FPGA que contendrá el procesador y también entradas o salidas (en forma de switches, botones y leds) que usados de la manera adecuada, nos permiten seguir de manera externa el comportamiento interno de la placa.

Para obtener noción de lo que sucedía dentro de la placa, decidimos hacer uso de algunos leds, que indican externamente el estado actual transitado por la unidad de debug.

El resto del comportamiento se observó mediante las devoluciones de la placa a través de la interfaz UART.

Software

Luego de definir todos los módulos y su funcionamiento, fué momento de escribir todo el código necesario para llevar a cabo su implementación física.

Para este fin, utilizamos el software Vivado, y el lenguaje de descripción de hardware Verilog.

La herramienta Vivado ofrece la posibilidad de hacer simulaciones, y con el fin de conseguir un buen resultado, se escribieron pruebas para casi todo los módulos de manera aislada, y también una vez integrados.

Clock

Debido a que es un circuito de gran tamaño, que involucra un gran número de recursos que necesitan estar sincronizados y actuar en conjunto, la frecuencia máxima de clock estará limitada.

Dado que todas las etapas deben actuar bajo el mismo dominio de clock para una correcta sincronización, la frecuencia máxima del procesador será igual a la frecuencia máxima de la etapa más lenta.

Afortunadamente, el software vivado, tras hacer una síntesis e implementación del proyecto, permite hacer análisis de timing, que nos indican si las restricciones de tiempo se cumplen para la frecuencia de clock utilizada.

Por defecto, la placa se configura a 100 MHZ, frecuencia a la cual las restricciones no se cumplieron. La metodología usada para determinar una opción viable fue ir reduciendola sistemáticamente hasta alcanzar una en la cual funcionase correctamente.

Para reducir la frecuencia de clock se utilizó la herramienta clock wizard de Vivado, que automatiza el proceso, permitiendo generar el módulo necesario solo ingresando los parámetros de entrada y salida deseados.

Configuración

El conjunto de elementos utilizados para llevar a cabo la síntesis e implementación del proyecto fueron:

- Una IP (propiedad intelectual) de clock wizard encargada de generar el clock requerido.
- El archivo top de verilog, que contenga el MIPS, así como el módulo de clock, para que queden definidas todas las conexiones.
- Un archivo de constraints, que mapee las entradas y salidas del procesador a pines específicos de la placa.

Desempeño

Funcionamiento general

El procesador cumplió con todas las especificaciones y se comunicó correctamente mediante el uso de UART.

Fueron probados diversos programas, que hicieron uso de un gran abanico de instrucciones y el comportamiento siempre fue el esperado.

A continuación se deja uno, escogido por hacer una comprobación intensa de las unidades y comportamientos más complejos.

1. ORI R10, R0, 112
2. OR R11, R0, R10
3. BEQ 1, R10, R11
4. HALT
5. SW R11, R0, 112
6. LW R12, R11, 0
7. SUBI R12, R12, 104
8. JALR R11, R12

En este, podemos apreciar el uso de la unidad de cortocircuitos, asi como de la unidad de detección de riesgos, que en un momento incluso debe meter una burbuja doble, para responder ante un riesgo de Load y de Jump Register que ingresan en el mismo ciclo.

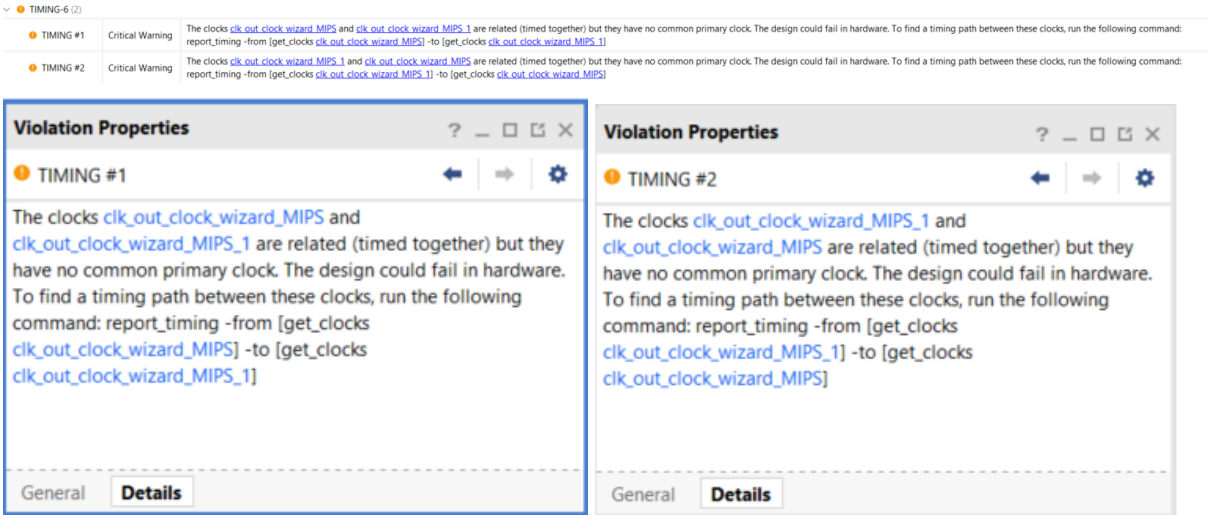
Resultados de la selección de frecuencia

La mejor frecuencia conseguida para la implementación fue de 40 MHz, ya que al elevarla a 50 MHz, el programa acusó que las restricciones de tiempo dejaban de cumplirse.

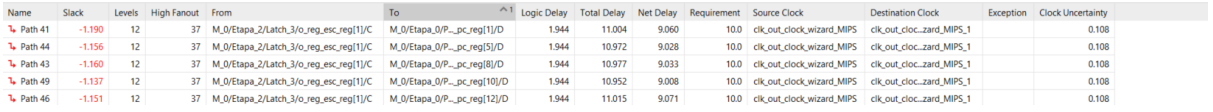
Estamos al tanto de que el valor es un poco bajo, pero al hacer un análisis un poco más profundo de los reportes entregados por Vivado nos encontramos con lo siguiente:

- Por un lado el Methodology Report nos arroja dos criticals Warnings, en los cuales se observa que al parecer hay dos clocks funcionando en el sistema. Ambos comparten el mismo nombre y solo difieren en la numeración añadida al final. Lo extraño del caso es que solo hay una IP generada en todo el

proyecto, y aquí vemos otra, que no hemos podido rastrear a ningún archivo del proyecto.



- Por otro lado, los caminos críticos indicados en el reporte de timing no se corresponden con ningún camino real, ya que los registros que se indican en el mismo no están relacionados ni conectados de manera alguna. También se observa que estos caminos críticos van de un clock a otro.



Name	Slack	Levels	High Fanout	From	To	Logic Delay	Total Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 41	-1.190	12	37	M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[1]/D	1.944	11.004	9.060	10.0	clk_out_clock_wizard_MIPS	clk_out_cloc...zard_MIPS_1		0.108
Path 44	-1.156	12	37	M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[5]/D	1.944	10.972	9.028	10.0	clk_out_clock_wizard_MIPS	clk_out_cloc...zard_MIPS_1		0.108
Path 43	-1.160	12	37	M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[8]/D	1.944	10.977	9.033	10.0	clk_out_clock_wizard_MIPS	clk_out_cloc...zard_MIPS_1		0.108
Path 49	-1.137	12	37	M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[10]/D	1.944	10.952	9.008	10.0	clk_out_clock_wizard_MIPS	clk_out_cloc...zard_MIPS_1		0.108
Path 46	-1.151	12	37	M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[12]/D	1.944	11.015	9.071	10.0	clk_out_clock_wizard_MIPS	clk_out_cloc...zard_MIPS_1		0.108

From	To
M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[1]/D
M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[5]/D
M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[8]/D
M_0/Etapa_2/Latch_3/o_reg_esc_reg[1]/C	M_0/Etapa_0/P..._pc_reg[10]/D

Source Clock	Destination Clock
clk_out_clock_wizard_MIPS	clk_out_clock_wizard_MIPS_1
clk_out_clock_wizard_MIPS	clk_out_clock_wizard_MIPS_1
clk_out_clock_wizard_MIPS	clk_out_clock_wizard_MIPS_1
clk_out_clock_wizard_MIPS	clk_out_clock_wizard_MIPS_1

Tras diversos intentos de solucionar el error sin resultados, y suponiendo que la limitación en la frecuencia surge de esta situación, en la cual dos clocks relacionados sin un clock primario común deben estar sincronizados, se optó por dejarlo de esta manera.

Conclusiones

El diseño del procesador cumple con todos los requerimientos indicados al inicio y funciona de manera sumamente satisfactoria. Su realización ha permitido observar de cerca todos los emergentes de la implementación de un procesador segmentado, así como relacionar el costo de esta implementación frente a sus beneficios.

Si bien por cuestiones que creemos más relacionadas al funcionamiento de la herramienta de software, que a la forma de implementación, no pudimos hacer que la frecuencia de funcionamiento de la placa alcance los 50 MHz, el verdadero objetivo era el desarrollo del procesador y creemos que se cumplió con creces.

La observación del resultado final y su funcionamiento, nos ha hecho reflexionar sobre la necesidad de implementar una unidad de excepciones si quisiéramos que el modelo fuera más funcional, ya que hay problemáticas que no se logran abordar con los módulos implementados.

De dichas problemáticas, y por citar una que se nos hizo muy evidente, rescatamos el caso de los Jump Registers, que no fuerzan ni corroboran de ninguna manera el alineamiento del PC.

Recursos

- https://github.com/AndresOriente/Arqui2022-TP3.git

Referencias

- MIPS IV Instruction Set - MIPS Technologies, Inc.