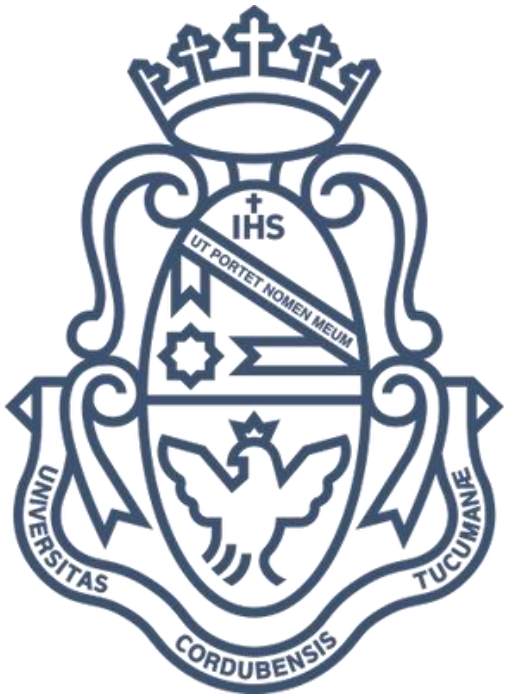
UNIVERSIDAD NACIONAL DE CÓRDOBA

Facultad de Ciencia Exactas Físicas y Naturales

Ingeniería en Computación



Trabajo Final

Pipeline Procesador MIPS

Jonathan Armando Patiño

1. Introducción

El presente trabajo tiene como objetivo implementar, validar y verificar el funcionamiento de un procesador MIPS pipeline (segmentado). La implementación del procesador MIPS se realiza en el lenguaje Verilog y la prueba de su funcionamiento se realizará en una placa Basys 3. La placa enviará los resultados de sus registros y memoria a través de una unidad UART.

2. Consignas de trabajo

Implementar el pipeline del procesador MIPS.

Para llevar a cabo la consigna del trabajo se tienen en cuenta los siguientes requerimientos.

- Implementar el procesador MIPS segmentado 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 (Excecute):*** 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.
- Instrucciones a implementar
    - ***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.
- El programa a ejecutar debe ser cargado en la memoria de 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

Los riesgos a los que se debe tener en cuenta para la realización del programa son:

- El procesador debe tener soporte para los siguientes tipos:
    - ***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.
- Unidad de Cortocircuitos.
- Unidad de Detección de Riesgos.

Una vez implementado el código de Verilog asociado al procesador MIPS se debe tener en cuenta para su prueba sobre la placa FPGA que se permiten dos modos de operación:

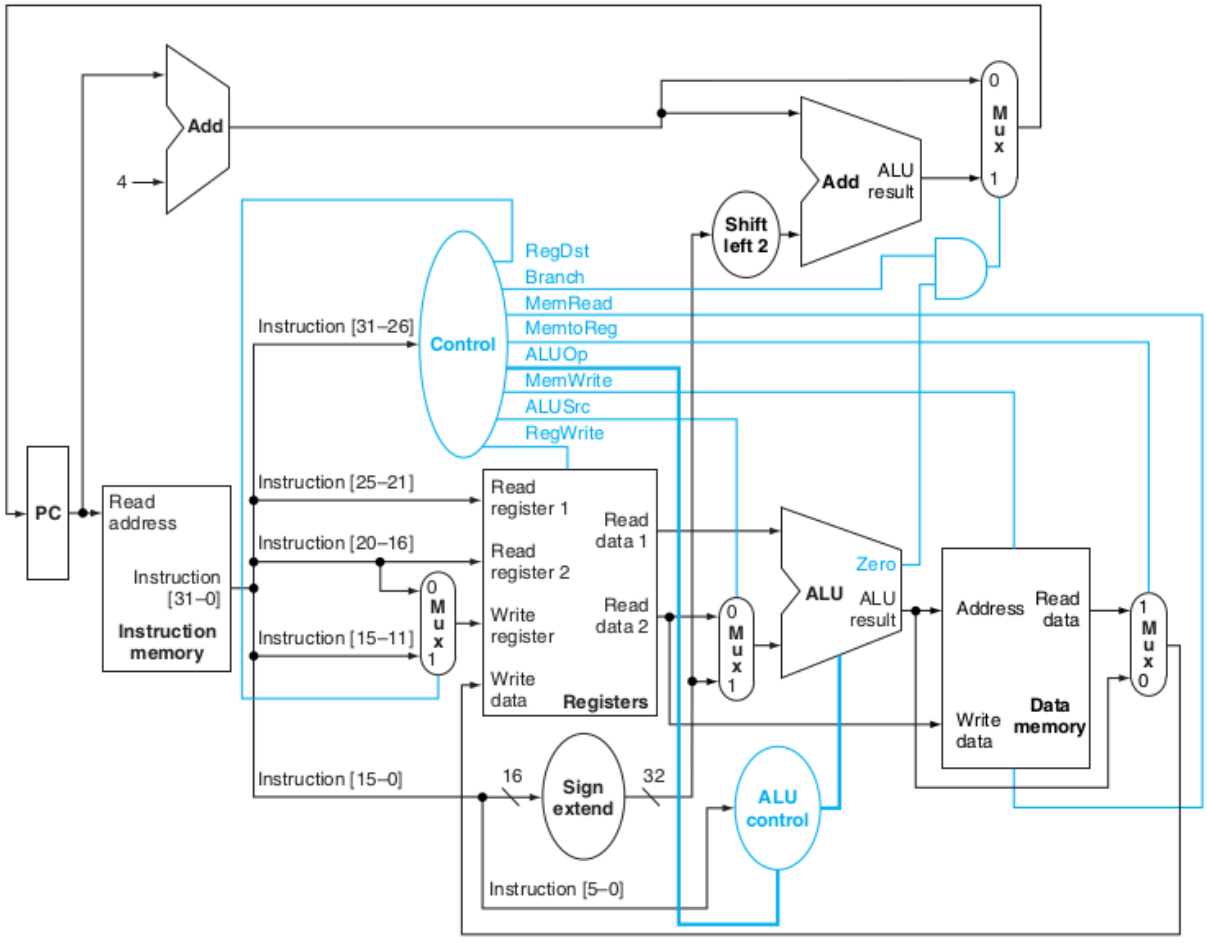
- ***Continuo:*** se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo. Llegado ese punto se muestran todos los valores indicados en pantalla.

- ***Paso a paso:*** Enviando un comando por la UART se ejecuta un ciclo de clock. Se debe mostrar a cada paso los valores indicados.

3. Marco teórico

3.1. Datapath y control del procesador MIPS monociclo

En resumen, la arquitectura del procesador monociclo en cuanto a su datapath y control con soporte para instrucciones R-type, instrucciones de carga/almacenamiento y saltos sobre igual se ve como el siguiente diagrama:

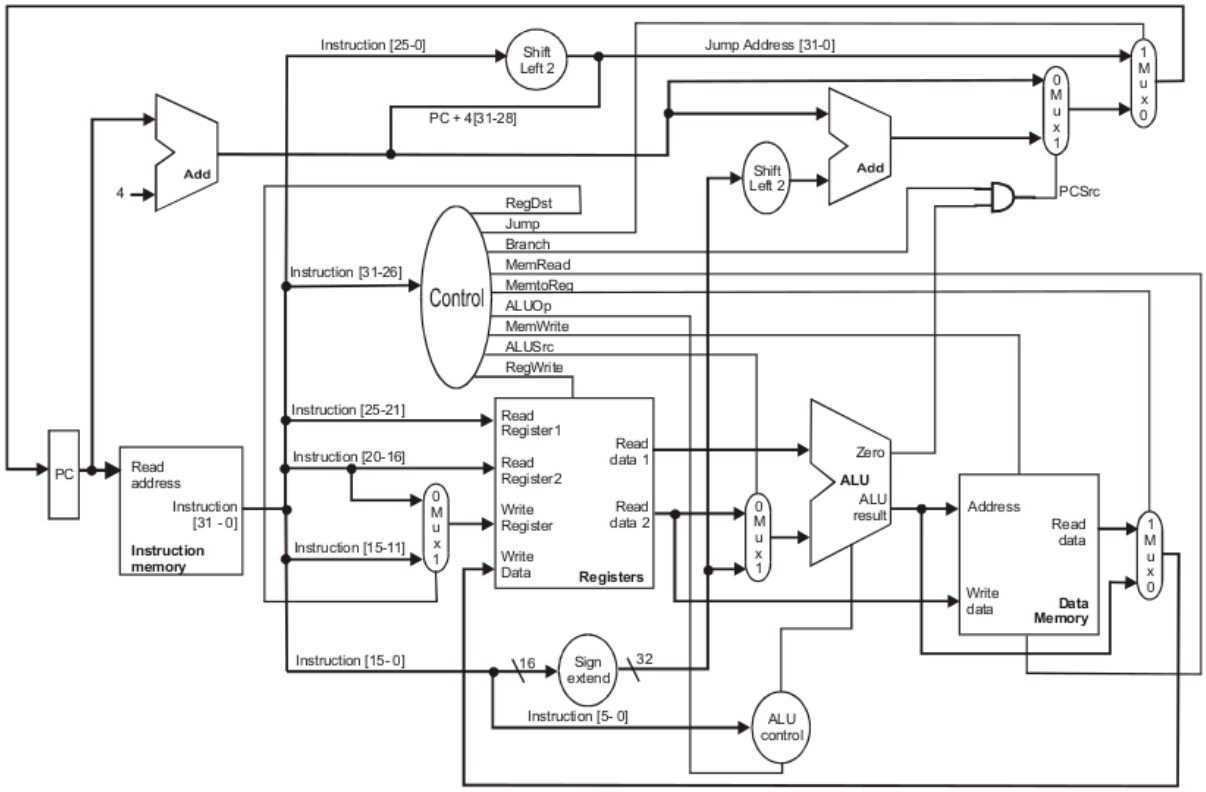


Teniendo en cuenta esta arquitectura del procesador MIPS monociclo, se puede iniciar el diseño del procesador MIPS segmentado.

Para la implementación de un procesador MIPS segmentado se requiere segmentar su ruta de datos (Datapath) y segmentar la unidad de control. Si se tuviera que implementar un procesador MIPS no segmentado se determinaría inicialmente su ruta de datos y una vez determinada se planificaría su unidad de control sobre esa ruta de datos. De la misma forma para la implementación del procesador MIPS segmentado, se inicia determinando el modelo de la ruta de datos segmentada y luego se determina la unidad de control para esa ruta de datos segmentada. En este último caso la unidad de control segmentada es equivalente a una línea de control independiente por cada etapa de segmentación de la ruta de datos.

Volviendo a la implementación monociclo del procesador MIPS, para tener en cuenta si se quisiera agregar al conjunto de instrucciones la instrucción de salto incondicional (jump), se podría agregar una bandera más para determinar el nuevo PC y el hardware necesario para la

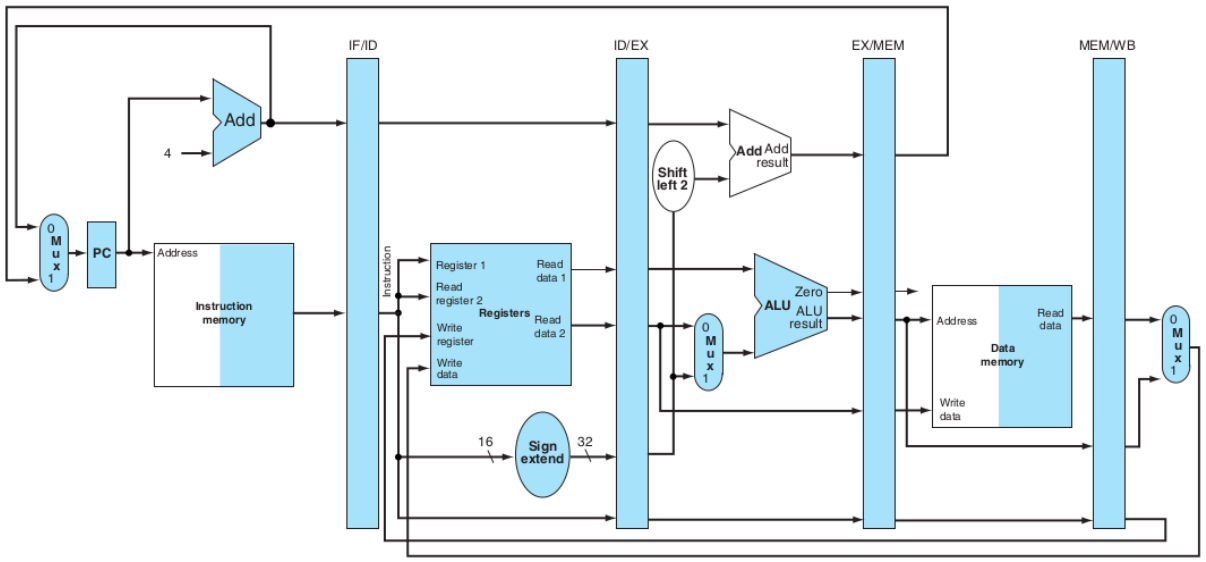
correcta elección del salto a realizar, por ejemplo, para el agregado de esta instrucción el diagrama anterior quedaría de la siguiente manera:



Y de esta forma se pueden ir agregando instrucciones en la medida que se crea conveniente y se requieran para un mayor soporte de instrucciones de ensamblador.
Retomando con la segmentación del procesador MIPS, a continuación se detallan los conceptos claves para la correcta implementación del mismo.

3.2. Datapath segmentado en el procesador MIPS

El modelo de datapath (o ruta de datos) segmentado en etapas, las cuales se describen en las 5 etapas nombradas en la consigna del presente trabajo, se representan por el siguiente diagrama:

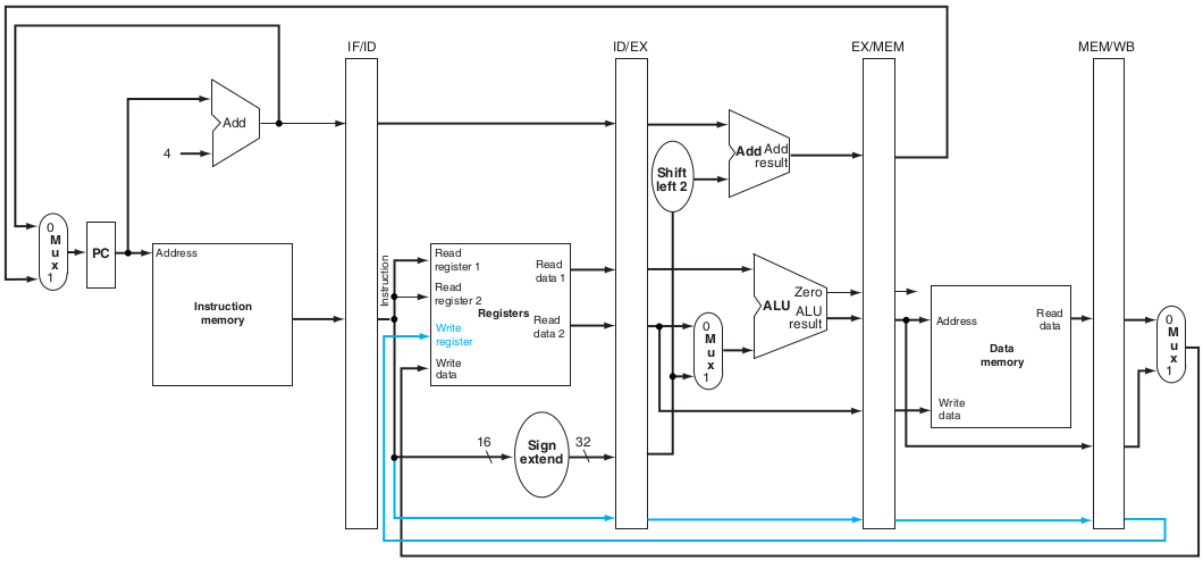


En este diagrama se observa la segmentación de la ruta de datos (Datapath), del procesador, que se logra generando los registros adicionales por cada etapa (IF/ID, ID/EX, EX/MEM y MEM/WB).

Con este segmentado aparece un inconveniente entre las etapas para las instrucciones de *carga de palabra (load word - LW)* tal como se describe a continuación:

> *El número de registro en el que se escribirá el dato obtenido en la última etapa (etapa 5 - Write Back) puede ser incorrecto ya que en ese momento se está tomando el número de registro de la instrucción que se encuentre en la etapa 2 (Instruction Decode - ID). El problema es que en la medida en que la instrucción LW avanzó en la segmentación, también debe haber avanzado el registro destino y puede que no sea así.*

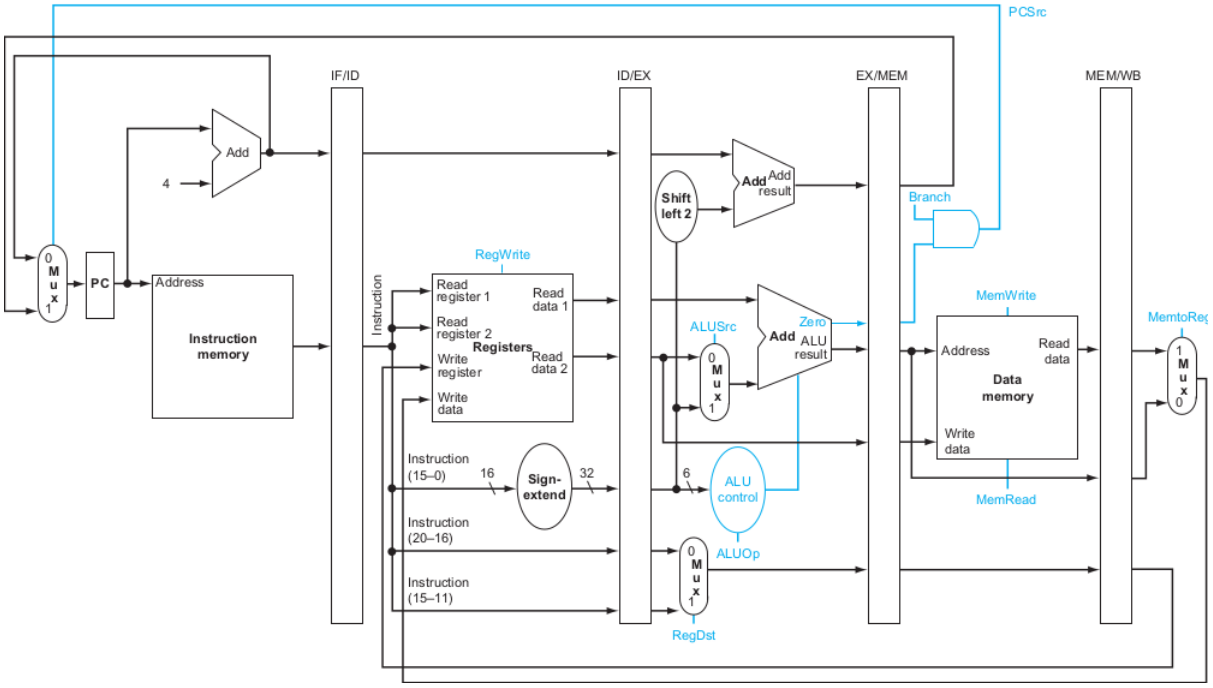
El diagrama anterior tiene en cuenta este inconveniente tal como se remarca sobre el mismo a continuación:



Por último, se destaca el sombreado en color del primer diagrama, que muestra la porción de la ruta de datos que utiliza una instrucción LW. En la memoria de instrucciones y en la memoria de datos solo se sombrea la mitad de la derecha, porque estas memorias sólo se acceden para lectura. El archivo de registros se sombrea completo, aunque es necesario recordar que la lectura se hace en la etapa 2 y la escritura en la etapa 5.

3.3. Control segmentado en el procesador MIPS

Partiendo del modelo de Datapath de la sección anterior, para las señales de control primero se deben colocar las señales de control requeridas por cada unidad funcional o multiplexor del camino de datos. Puesto que se partió de una implementación de un sólo ciclo, de ese diseño se toman las señales de control para agregarlas a la implementación segmentada. El resultado se presenta en el siguiente diagrama:



Puede notarse que los saltos condicionales se determinan en la etapa de acceso a memoria, esto provoca algunas situaciones difíciles de tratar, porque si el salto se va a realizar, habrá que eliminar de alguna manera a las instrucciones que han ingresado a la segmentación y que están en las etapas anteriores. De momento se supondrá que los saltos no se realizan, en la sección 3.6 del presente informe se analizará cómo resolver tales situaciones (riesgos de control).

Para especificar el control segmentado, es necesario especificar un conjunto de valores en cada una de las etapas de la segmentación.

Puesto que cada línea de control esta está asociada con una componente activa en solo una etapa de la segmentación, es posible dividir las líneas de control en cinco grupos de acuerdo a las etapas de segmentación:

1. ***Captura de la Instrucción.*** Las señales de control para la lectura de memoria de código y escritura del PC están siempre acertadas, porque cambiarán su valor en cada ciclo de reloj, por lo que no hay líneas de control especiales.
2. ***Decodificación de la instrucción y lectura del archivo de registro.*** Sucede una situación similar a la etapa anterior, no hay líneas de control para un ajuste opcional.
3. ***Ejecución o Cálculo de una dirección.*** En esta etapa, dependiendo del tipo de instrucción debe seleccionarse:
   - El segundo operando de la ALU (ALUSrc).
   - La operación que realizará la ALU (ALUOp).
   - El registro destino (RegDst).
4. ***Acceso a memoria.*** Aquí se determina si:
   - Se trata de un salto (branch).
   - Se escribirá en memoria (MemWrite).
   - Se leerá de memoria (MemRead).
5. ***Retro escritura.*** Son dos señales que se deben definir:
   - La que determina si se escribirá el resultado de la ALU o el dato de memoria (MemtoReg).
   - La que habilita la escritura en el archivo de registros (RegWrite).

Cabe aclarar que nuevamente se usará el control de la ALU desarrollado para la implementación de un solo ciclo, por lo que las señales ALUOp determinarán si la operación a realizar será una suma, una resta o si dependerá del campo de función. Y además se agregará para el caso de las instrucciones del tipo JAL utilizar el código 11 para no afectar a la instrucción SRA que tiene el mismo código en su campo Funct.

Valor de ALUOp	Operacion deseada en la ALU
00	Suma
01	Resta
10	Depende del campo Funct u Opcode
11	Depende del campo Opcode solo para la instruccion JAL ya que se repite su codigo de operacion en el modo 10

Sin embargo, para que el control genere esta señal necesita conocer el valor de la bandera zero. Para omitir esa entrada extra al control, se utilizó una compuerta AND de dos entradas, de manera que ahora el control genera la señal Branch cuando detecta un salto (primera entrada de la AND) y la realización del salto depende del valor de la bandera zero (segunda entrada a la AND).

Nombre de la señal	Efecto cuando es desacertada	Efecto cuando es acertada
RegDst	El número del registro destino para la escritura viene del campo rt (20-16)	El número del registro destino para la escritura viene del campo rd (15-11)
RegWrite	Ninguno	Se escribirá un dato en el archivo de registros
ALUSrc	El segundo operando de la ALU es el segundo dato leído en el archivo de registros	El segundo operando de la ALU son los 16 bits de desplazamiento tomados de la instrucción y extendidos en signo
Branch	El PC es remplazado por PC + 4	El PC será remplazado por la suma de una dirección calculada para un brinco si se generó la bandera zero
MemRead	Ninguno	Se hace la lectura de la memoria de datos
MemWrite	Ninguno	Se hace la escritura en la memoria de datos
MemtoReg	El valor del dato que se escribirá en el archivo de registros viene de la ALU	El valor del dato que se escribirá en el archivo de registros viene de la memoria de datos

La consideración importante en el diseño del control, es que las señales deben de tener el valor correcto en la etapa correcta. Basado en la descripción anterior y en las dos tablas anteriores, se construye una tabla nueva, en la que se indican las señales involucradas en cada una de las etapas y su valor para cada tipo de instrucción.

	Etapa de Ejecución/ Cálculo de dirección				Etapa de acceso a memoria			Etapa de retro escritura	
Instrucción	Reg Dst	ALUOp1	ALUOp0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
Tipo R	1	1	0	0	0	0	0	1	0
LW	0	0	0	1	0	1	0	1	1
SW	x	0	0	1	0	0	1	0	x
BEQ	x	0	1	0	1	0	0	0	x

Los valores de las señales son los mismos que los que se requerían en una implementación de un solo ciclo, la diferencia en esta implementación es que ahora esos valores se requieren en diferentes etapas.

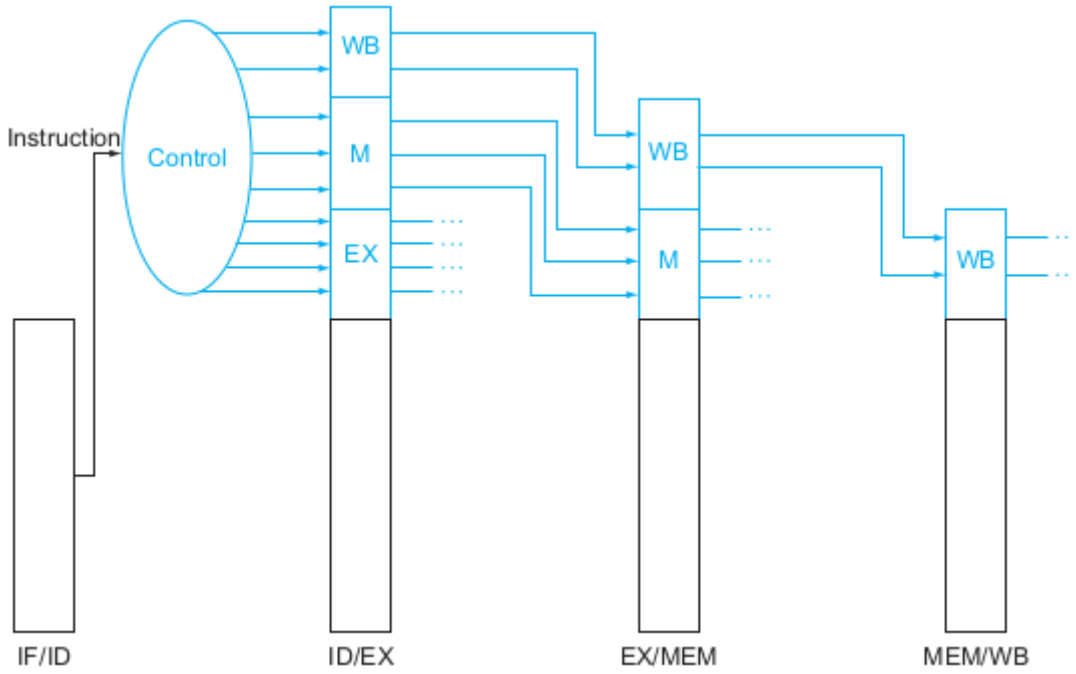
Pero como por requerimiento del presente trabajo se debe tener soporte a más instrucciones la tabla se actualiza como sigue:

	Valores de señales de control													
	Etapa IF	Etapa ID				Etapa EX					Etapa MEM		Etapa WB	
Tipo de Instruccion	End	beq	bne	j_jal	jr_jalr	RegDst	ALUSrc	ALUSrcA	ALUOP	FunctSrc	MemRead	MemWrite	RegWrite	Memtoreg
R-TYPE	0	0	0	00	00	1	0	0	10	0	0	0	1	0
SLL	0	0	0	00	00	1	0	1	10	0	0	0	1	0
SRL	0	0	0	00	00	1	0	1	10	0	0	0	1	0
SRA	0	0	0	00	10	1	0	1	10	0	0	0	1	0
JR	0	0	0	00	01	x	x	x	xx	x	0	0	0	x
JALR	0	0	0	00	00	1	0	0	10	x	0	0	1	x
ADDI	0	0	0	00	00	0	1	0	00	X	0	0	1	0
ANDI	0	0	0	00	00	0	1	0	10	1	0	0	1	0
SLTI	0	0	0	00	00	0	1	0	10	1	0	0	1	0
ORI	0	0	0	00	00	0	1	0	10	1	0	0	1	0
XORI	0	0	0	00	00	0	1	0	10	1	0	0	1	0
LUI	0	0	0	00	00	0	1	0	10	1	0	0	1	0
LW	0	0	0	00	00	0	1	0	00	x	1	0	1	1
LWU	0	0	0	00	00	0	1	0	00	x	1	0	1	1
LB	0	0	0	00	00	0	1	0	00	x	1	0	1	1
LBU	0	0	0	00	00	0	1	0	00	x	1	0	1	1
LH	0	0	0	00	00	0	1	0	00	x	1	0	1	1
LHU	0	0	0	00	00	0	1	0	00	x	1	0	1	1
SW	0	0	0	00	00	x	1	0	00	x	0	1	0	x
SB	0	0	0	00	00	x	1	0	00	x	0	1	0	x
SH	0	0	0	00	00	x	1	0	00	x	0	1	0	x
BEQ	0	1	0	00	00	x	0	0	01	x	0	0	0	x
BNE	0	0	1	00	00	x	0	0	01	x	0	0	0	x
J	0	0	0	10	00	x	x	x	xx	x	0	0	0	x
JAL	0	0	0	11	00	1	0	0	11	1	0	0	1	0
HALT	1	0	0	00	00	0	0	0	00	0	0	0	0	0

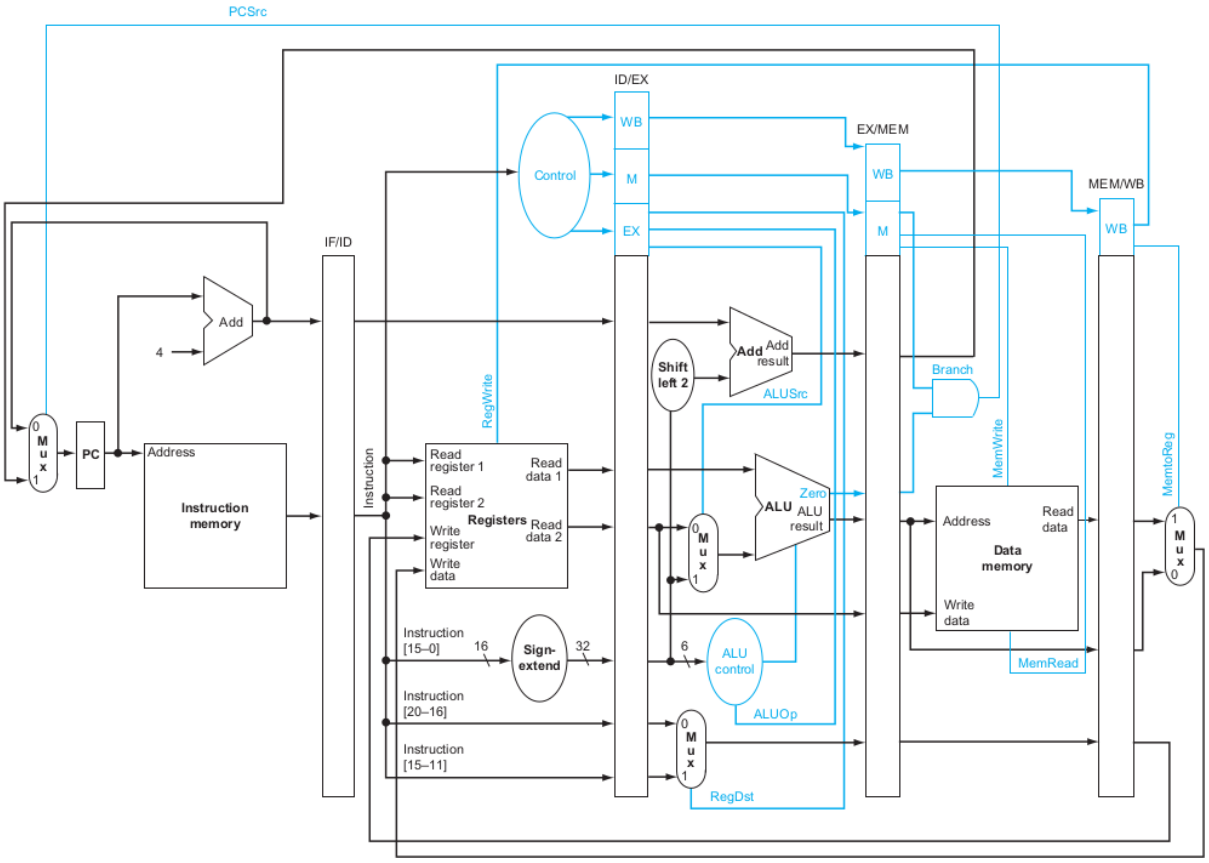
En la etapa 1 se accede a la memoria de instrucción para atrapar la instrucción a ejecutar en el registro IF/ID y se tiene en cuenta la señal de control (o flag) End para saber si finaliza la ejecución de instrucciones o debe seguir.

En la etapa 2 se toma del registro IF/ID al opcode, que determina el tipo de instrucción que se está ejecutando, los 6 bits del opcode son las entradas al circuito de control, el circuito de control es el mismo que el de una implementación de un sólo ciclo (combinacional), de manera que las señales de control toman sus valores en forma inmediata, etapa en la cual se utilizarán solo las señales de control asociadas a los saltos sobre igual y no condicionales. Luego el resto de señales deben escribirse en el registro ID/EX para que viajen junto con la instrucción en las diferentes etapas de la segmentación.

En la etapa 3, en el registro ID/EX no sólo se encuentran los datos, también están todas las señales de control, en esta etapa se usarán algunas de ellas (de acuerdo a la tabla anterior) y el resto se escribe en el registro EX/MEM para que avancen en la segmentación.
En la etapa 4 ocurre algo similar, se toman las señales de control a utilizarse del registro EX/MEM y las restantes se escriben en el registro MEM/WB.
En la etapa 5, del registro MEM/WB se toman las últimas señales de control del registro MEM/WB y se aplican sobre los datos correspondientes.
Entonces, los cuatro registros que separan a las cinco etapas de segmentación deben extenderse en tamaño para que incluyan a las señales de control. En la figura 5.23 se muestra el hardware correspondiente al control, su generación y su desplazamiento por las diferentes etapas.



En el siguiente diagrama se muestra el camino de datos y el control para una implementación segmentada, las señales de control se muestran como un bus que se divide en la etapa en la que se aplicará.



3.4. Riesgos de Control en el procesador MIPS segmentado

Los riesgos de control a tener en cuenta sobre el procesador MIPS segmentado, son:

- Riesgos estructurales.
  - Se solucionan con el adecuado diseño del *datapath* y la *unidad de control*.
- Riesgos por dependencia de datos.
  - Solución por anticipación.
  - Solución por detenciones.
- Riesgos de control (en saltos).
  - Se solucionan con la ayuda de la señal de control IFlush.

Riesgos de Estructura: Significa que el hardware no puede soportar la combinación de instrucciones que se quiere ejecutar en el mismo ciclo.
En un lavadero de ropa, por ejemplo, si la lavadora y la secadora formarán parte del mismo equipo, no sería posible trabajar estas dos etapas con dos cargas de ropa diferentes.
En un procesador, si se cuenta con una sola memoria, para datos y código, no será posible escribir o leer un dato, mientras se atrapa una instrucción. Para evitar estos riesgos, se debe definir correctamente al camino de datos.
Riesgos por dependencias de datos: Una instrucción depende del resultado de una instrucción previa que aún está en la segmentación. En las próximas secciones se revisa a detalle este tipo de estos riesgos y se muestra una técnica conocida como anticipación para resolverlos.
Riesgos de Control: Surgen de la necesidad de hacer una decisión basada en los resultados de una instrucción, mientras otras se están ejecutando.

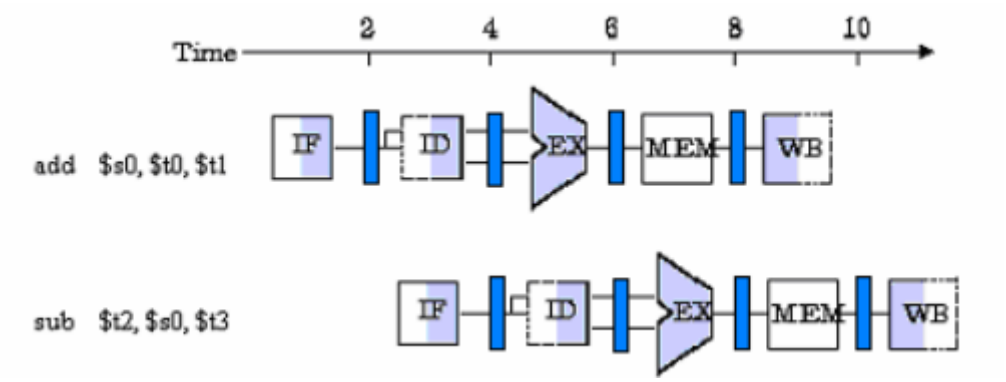
Por ejemplo, en un salto condicional, mientras se determina si el salto se hará o no, otras instrucciones ingresarán al procesador. Si se determina que el salto no se llevará a cabo, esas instrucciones prosiguen su ejecución. Pero si el salto no se realiza, habrá que evaluar cómo eliminar a las instrucciones que ya ingresaron al procesador. Estos riesgos se revisan en la última sección, resolviéndolos por otra técnica conocida como detenciones.

3.5. Riesgos por dependencia de datos

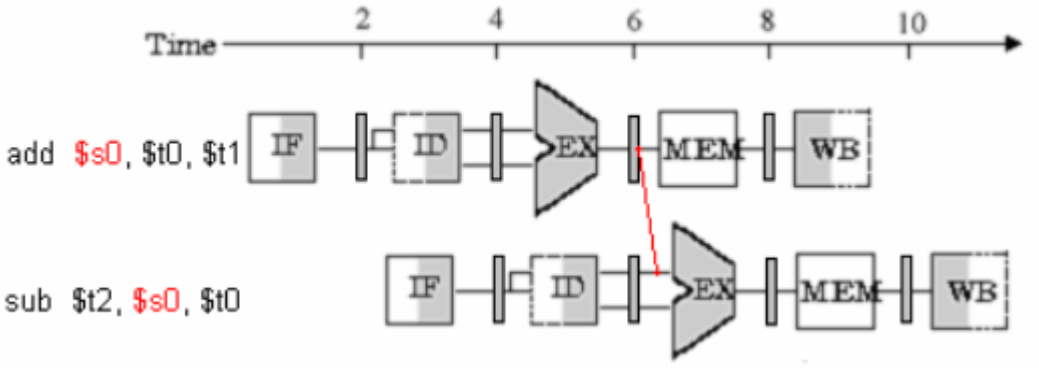
Los riesgos por dependencias de datos ocurren cuando la ejecución de una instrucción depende del resultado de una instrucción previa que aún está en la segmentación. Por ejemplo, si se ejecutan las instrucciones:

*add **$s0**, $t0, $t1*
*sub $t2, **$s0**, $t3*

De acuerdo a las etapas en las que se dividió la ejecución de instrucciones, el resultado de la suma se escribirá al final de la quinta etapa en el registro $s0 y la resta hace su lectura de registros en la segunda etapa. Suponiendo que cada etapa tarda 2 ns, el resultado de la suma se escribe a los 10 ns, y la resta esperaba leer el registro a los 4 ns, como se muestra en la siguiente figura, la resta leerá el valor anterior de $s0.



Para resolver este problema se propone capturar el dato una vez obtenido de la etapa 3 como se nota en la siguiente figura:



Al esquema mostrado en la figura anterior se le conoce como anticipación. La anticipación es una técnica para resolver los riesgos por dependencias de datos que consiste en tomar el dato de uno de los operandos de la ALU antes de que éste sea escrito en el registro destino.

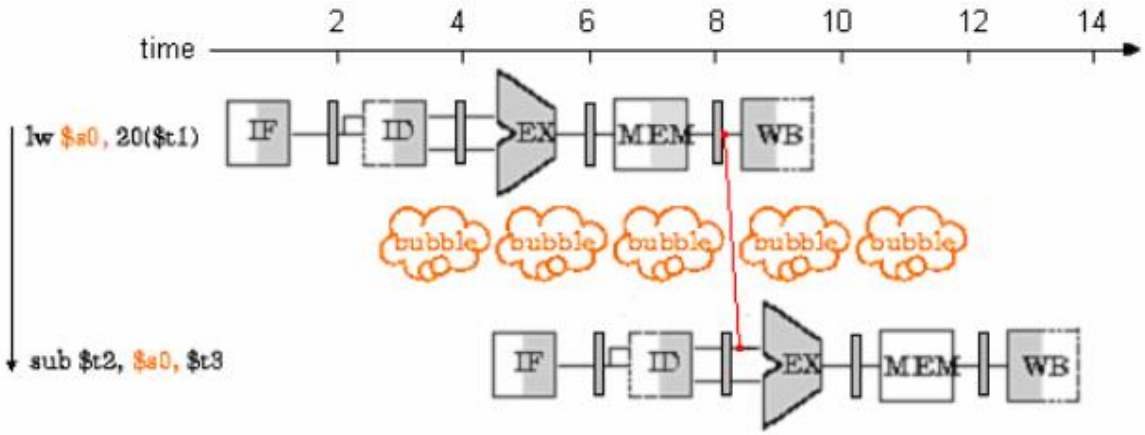
Tiene como ventaja que los riesgos se resuelven a nivel de hardware, por lo que el compilador no debe insertar instrucciones NOP y, por lo tanto, no provoca retrasos en la ejecución de un programa.

Ahora, qué ocurre cuando se intenta ejecutar la siguiente secuencia:

lw **$s0**, 20 ($t1)

sub $t2, **$s0**, $t3

En este caso, la carga obtendrá el dato de memoria hasta el final de la etapa 4, por lo que aún con la anticipación la resta no puede disponer del valor correcto para $s0, porque lo necesita un ciclo de reloj antes de que la carga lo obtenga.
Lo que es un hecho es que la resta se debe detener un ciclo de reloj, para que luego pueda usarse la anticipación. Las detenciones son necesarias en aquellos casos en que la anticipación no es suficiente. Una detención consiste en la inserción de una burbuja entre las dos instrucciones en conflicto. A una burbuja la definiremos como un conjunto de señales inofensivas, es decir, que no afectan la ejecución del programa.
En la próxima figura se muestra como después de insertar una burbuja, ya es posible anticipar el resultado esperado en $s0 para usarse en la instrucción siguiente. Las burbujas se insertan al nivel de hardware, por lo que son transparentes al compilador. Es un hecho que las burbujas producen un retraso de un ciclo de reloj, pero solo se insertarán cuando el dato a cargar en un registro será utilizado como operando de la ALU en la siguiente instrucción.



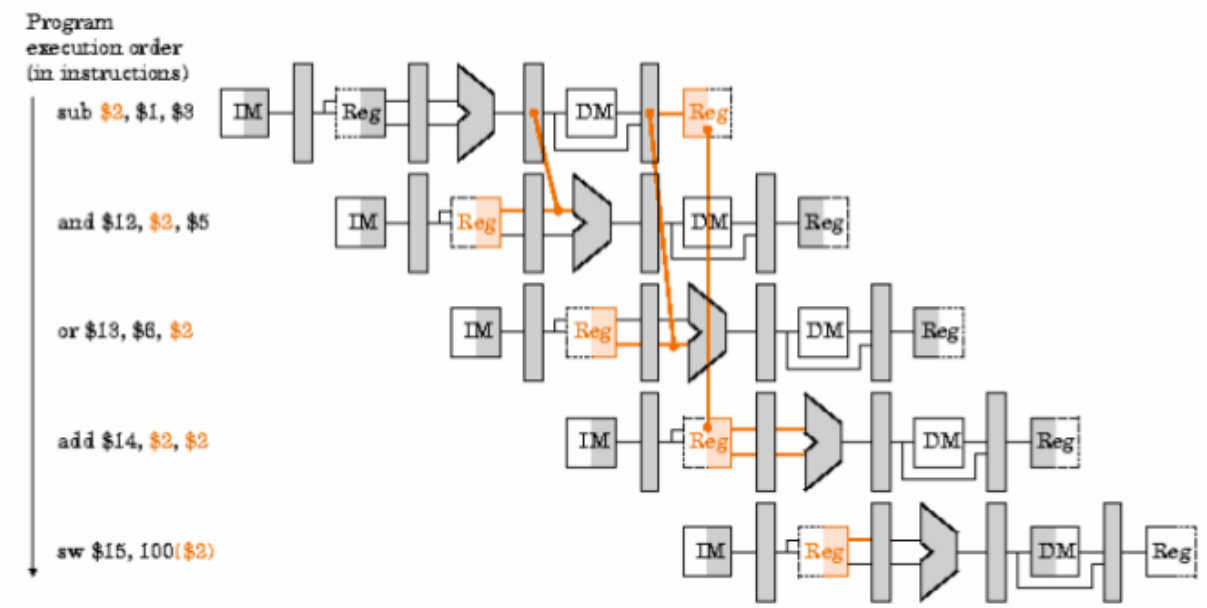
Para los riesgos por dependencia de datos se debe tomar en consideración que el camino de datos es único y que los resultados de una etapa se escriben en alguno de los registros encargados de separar etapas. De manera que las líneas rojas indican que de ahí se tomará la información y se regresará a una etapa anterior para que se utilice por las instrucciones subsecuentes.

Solución por anticipación

La anticipación tomará el dato de uno de los operandos de la ALU antes de que éste sea escrito en el registro destino. Se considera la siguiente secuencia de código para obtener los posibles casos:

```
sub $2, $1, $3          # El registro $2 es escrito por sub
and $12, $2, $5         # El 1 er operando ($2) depende de sub
or $13, $6, $2          # El 2o operando ($2) depende de sub
add $14, $2, $2         # Los 2 operandos dependen de sub
sw $15, 100( $2 )       # El registro base ($2) depende de sub
```

En la siguiente figura se muestra la representación de múltiples ciclos de la ejecución de esta secuencia. Considerando como ciclo de reloj 1 cuando la instrucción SUB entra a la segmentación, se tiene que la instrucción SUB genera con la ALU el valor que se escribirá en $2 y lo escribe en el registro EX/MEN al final del ciclo de reloj 3.



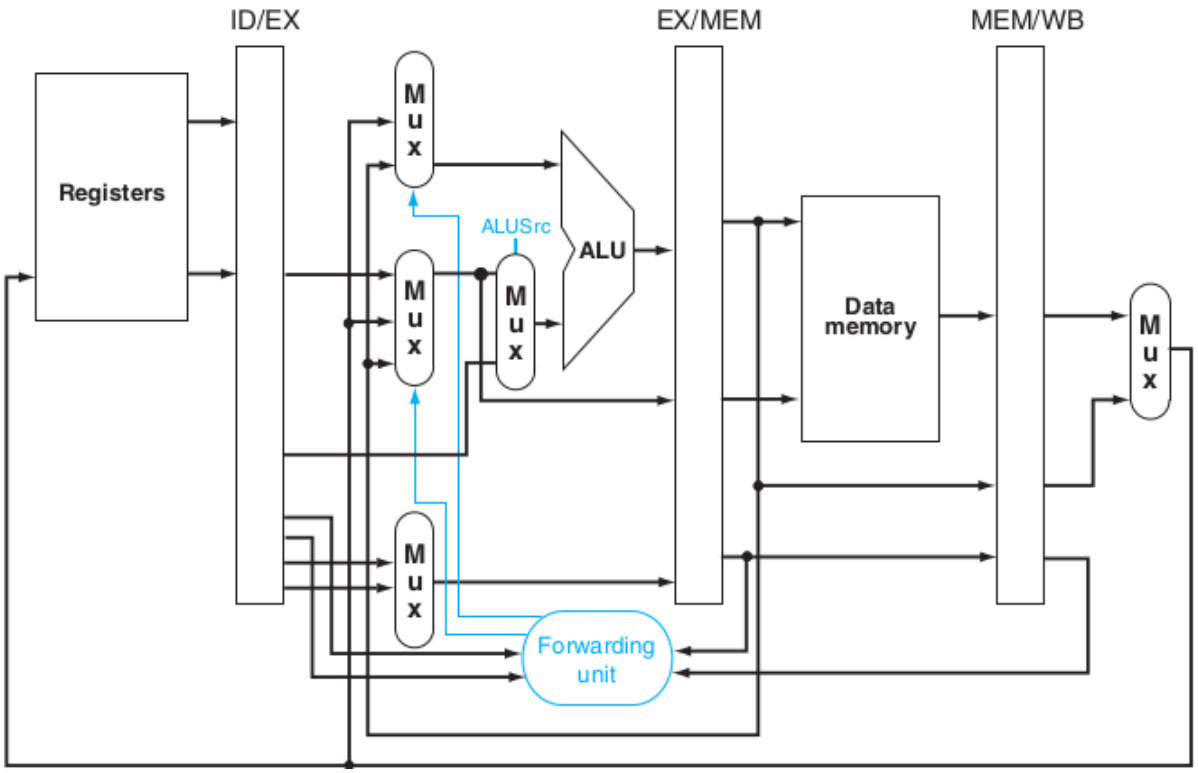
Al comienzo del ciclo de reloj 5, la OR toma el valor de $2 del registro MEM/WB y lo regresa a la etapa 3 para que sea su segundo operando de la ALU. Durante ese ciclo se espera que se haga la escritura del registro $2 para que su valor sea usado por la instrucción ADD durante el mismo ciclo.

Esto es posible si los registros que separan las etapas de la segmentación y el archivo de registros activan su escritura en flancos diferentes, así, durante el flanco de subida se puede escribir el registro MEM/WB con la información y las señales de control correspondientes, durante la primera mitad del ciclo estas señales se establecen, de manera que cuando ocurre el ciclo de bajada, el archivo de registros escribirá la información correcta (la instrucción SUB escribirá el valor de $2) y durante la segunda mitad del ciclo, se puede disponer de los datos actualizados para su lectura (la instrucción ADD leerá el nuevo valor de $2).

Entonces, la anticipación puede hacerse de diferentes lugares, puede ocurrir que el dato a anticipar se encuentre en el registro EX/MEN o en el registro MEM/WB, y en ambos casos puede tratarse del primero o del segundo operando, incluso de ambos.

En todos los casos, las anticipaciones involucran a los operandos de la ALU y la ALU está en la etapa 3, en una implementación sin anticipaciones los dos operandos se toman directamente del registro ID/EX. Al considerar las anticipaciones, se debe saber cuál será el registro destino en las instrucciones que están en las etapas MEM y WB, y si se va a escribir dicho registro. Si ese registro coincide con alguno de los operandos de la etapa EX, su valor debe regresar a la etapa EX para que sustituya a ese operando. En la próxima figura se muestra la segmentación con la incorporación de la Unidad de anticipación (Forwarding Unit).

La unidad de anticipación compara cada uno de los operandos de la ALU (rs y rt) con el registro destino de la etapa MEM (EX/MEM.RegistroRd) y con el registro destino de la etapa WB (MEM/WB.RegistroRd) si existe igualdad entre un par de ellos, y la señal de escritura en registro esta acertada en la etapas MEM (EX/MEM.RegWrite) o en la etapa WB (MEM/WB.REgWrite) , la unidad de anticipación debe colocar el valor adecuado para sustituir a algún operandos, controlando los multiplexores de anticipación.



Las decisiones que tomará la unidad de anticipación con respecto a la etapa MEM son:

if (EX/MEM.RegWrite AND EX/MEM.RegistroRd != 0 AND
EX/MEM.RegistroRd == ID/EX.RegistroRs)
ForwardA = 10

if (EX/MEM.RegWrite AND EX/MEM.RegistroRd != 0 AND
EX/MEM.RegistroRd == ID/EX.RegistroRt)
ForwardB = 10

Respecto a la etapa WB, la unidad de anticipación debe determinar:

if (MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND
MEM/WB.RegistroRd == ID/EX.RegistroRs)
ForwardA = 01

if (MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND
MEM/WB.RegistroRd == ID/EX.RegistroRt)
ForwardB = 01

ForwardA y ForwardB son señales de dos bits que controlan los multiplexores de anticipación y determinan cuáles serán los operandos de la ALU tal como se detalla en la siguiente tabla.

Control	Origen	Explicación
ForwardA = 00	ID/EX	El primer operando de la ALU viene del archivo de registros
ForwardA = 10	EX/MEM	El primer operando de la ALU es anticipado del anterior resultado de la ALU
ForwardA = 01	MEM/WB	El primer operando de la ALU es anticipado de la memoria de datos o de un previo resultado de la ALU
ForwardB = 00	ID/EX	El segundo operando de la ALU viene del archivo de registros
ForwardB = 10	EX/MEM	El segundo operando de la ALU es anticipado del anterior resultado de la ALU
ForwardB = 01	MEM/WB	El segundo operando de la ALU es anticipado de la memoria de datos o de un previo resultado de la ALU

Una complicación se presenta si la etapa WB presenta un resultado para un registro y la etapa MEM presenta un resultado diferente para el mismo registro, y ese registro es un operando de la ALU. Por ejemplo, si tenemos el siguiente código:

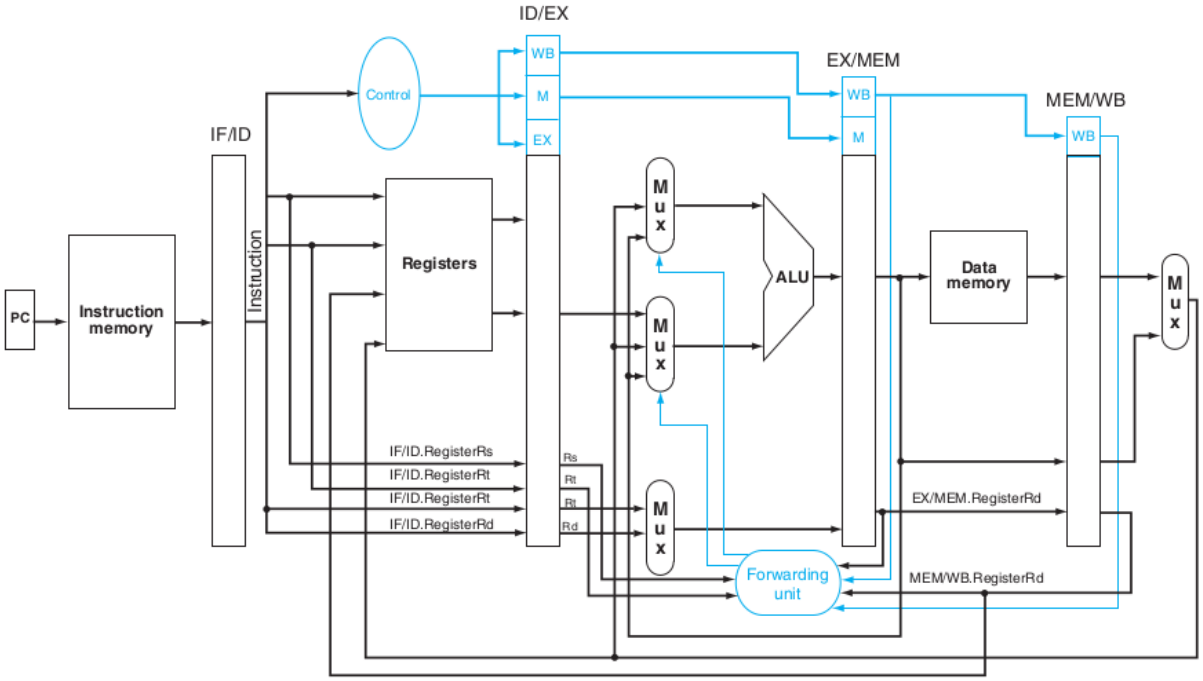
add **$1**, **$1**, $2
add **$1**, **$1**, $3
add **$1**, **$**1, $4

Cuando la segunda suma está en la etapa EXE, anticipa sin problema el valor del registro $1 de la etapa MEM (el cual fue generado por la primera suma).
Sin embargo, cuando la tercera suma alcanza la etapa EXE, tiene dos posibilidades para anticipar el valor de $1, en la etapa MEM se encuentra el valor generado por la segunda suma y en la etapa WB se encuentra el valor generado por la primera.
En ese caso, el resultado que debe anticiparse es el de la etapa MEM porque es el resultado más reciente. Por lo tanto, se deben modificar las decisiones que toma la unidad de anticipación respecto a la etapa WB, para no entrar en conflicto con la etapa MEM. Ahora se tendrá (las nuevas comparaciones se agregan en rojo):

if (MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND
EX/MEM.RegistroRd != ID/EX.RegistroRs AND
MEM/WB.RegistroRd == ID/EX.RegistroRs)
ForwardA = 01

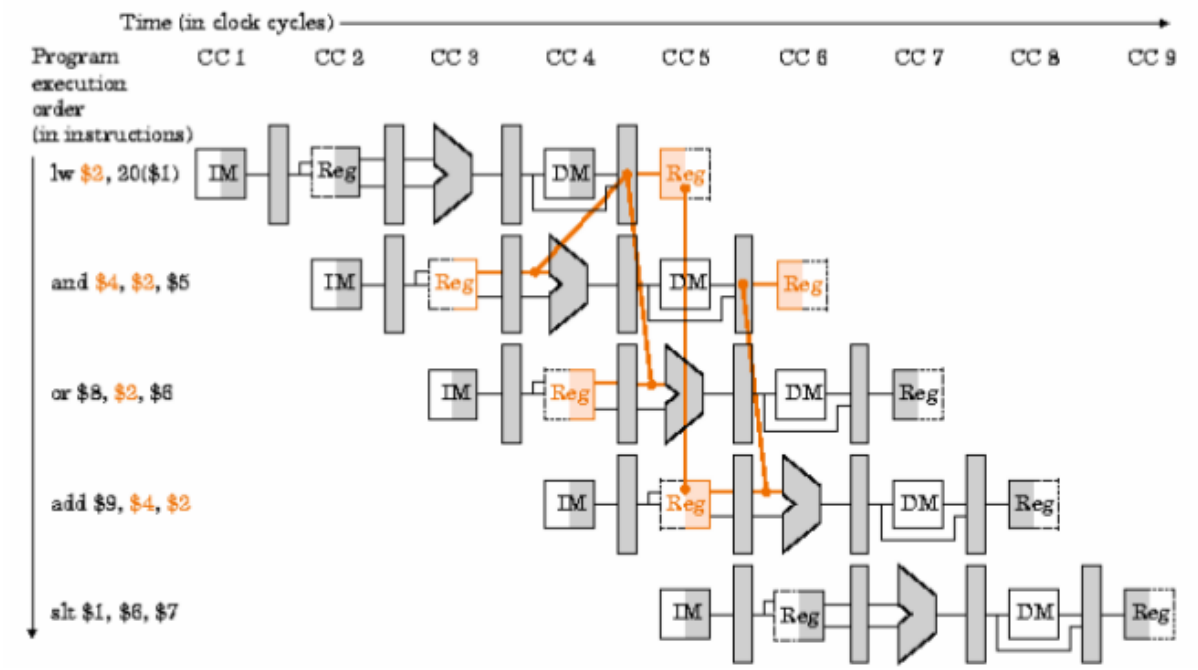
if (MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND
EX/MEM.RegistroRd != ID/EX.RegistroRt AND
MEM/WB.RegistroRd == ID/EX.RegistroRt)
ForwardB = 01

En la próxima figura se muestra el camino de los datos segmentado y el control, con la unidad de anticipación. Es importante aclarar que solo se esboza el camino de los datos, se omiten algunos detalles, como el multiplexor que se presenta en la segunda entrada de la ALU o el mismo control de la ALU, porque el punto importante en esta sección es el estudio de la unidad de anticipación.



Solución por detenciones

Como se describió en la sección anterior, la anticipación no puede resolver aquellas situaciones en las que se realizará la carga de un registro (LW) y el dato a cargar es un operando de la instrucción siguiente. En la siguiente figura se muestra un ejemplo del problema, se tiene una instrucción que cargará un dato de memoria al registro $2 seguida por una AND cuyo primer operando es $2.

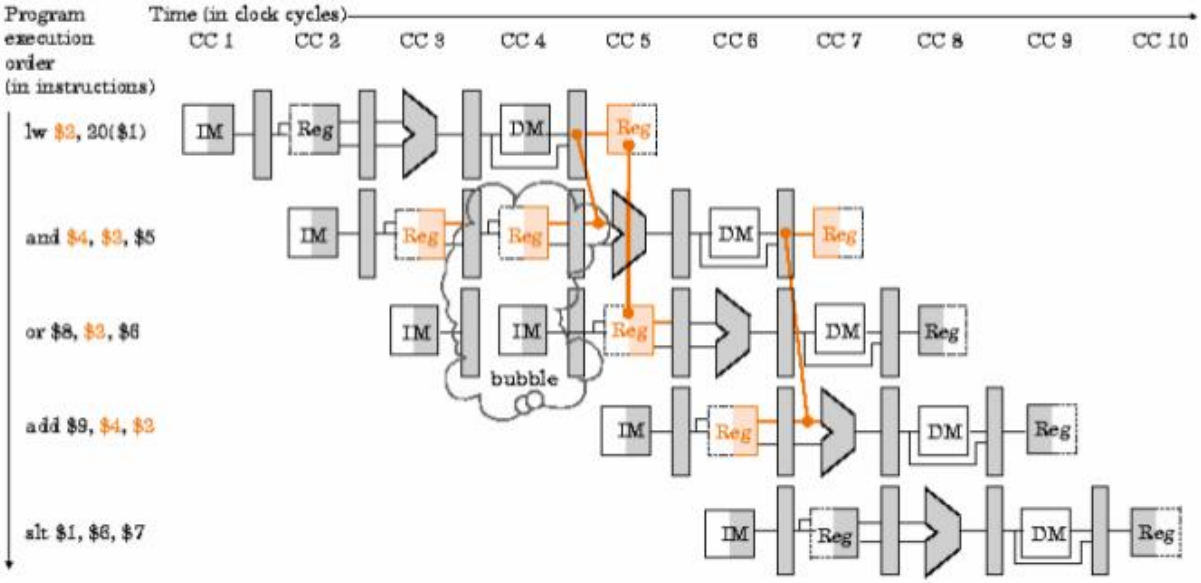


En el ciclo de reloj 4, el dato que se escribirá en $2 se está leyendo de la memoria y la ALU lo requiere en ese mismo ciclo, no es posible anticipar el valor de $2 porque se escribirá en el registro de segmentación hasta el final del ciclo y se podrá disponer de él hasta el siguiente ciclo.

Por lo tanto, se debe detener a la instrucción AND por un ciclo de reloj, para que en el ciclo de reloj 5, con ayuda de la unidad de anticipación, pueda operar sobre el valor correcto de $2.
De manera que, además de la unidad de anticipación, es necesario agregar una unidad de detección de riesgos (Hazard detection Unit), que opere en la etapa ID, de manera que pueda insertar una burbuja entra la carga y la instrucción que usará el dato a cargar. Esta nueva unidad evaluará la siguiente condición:

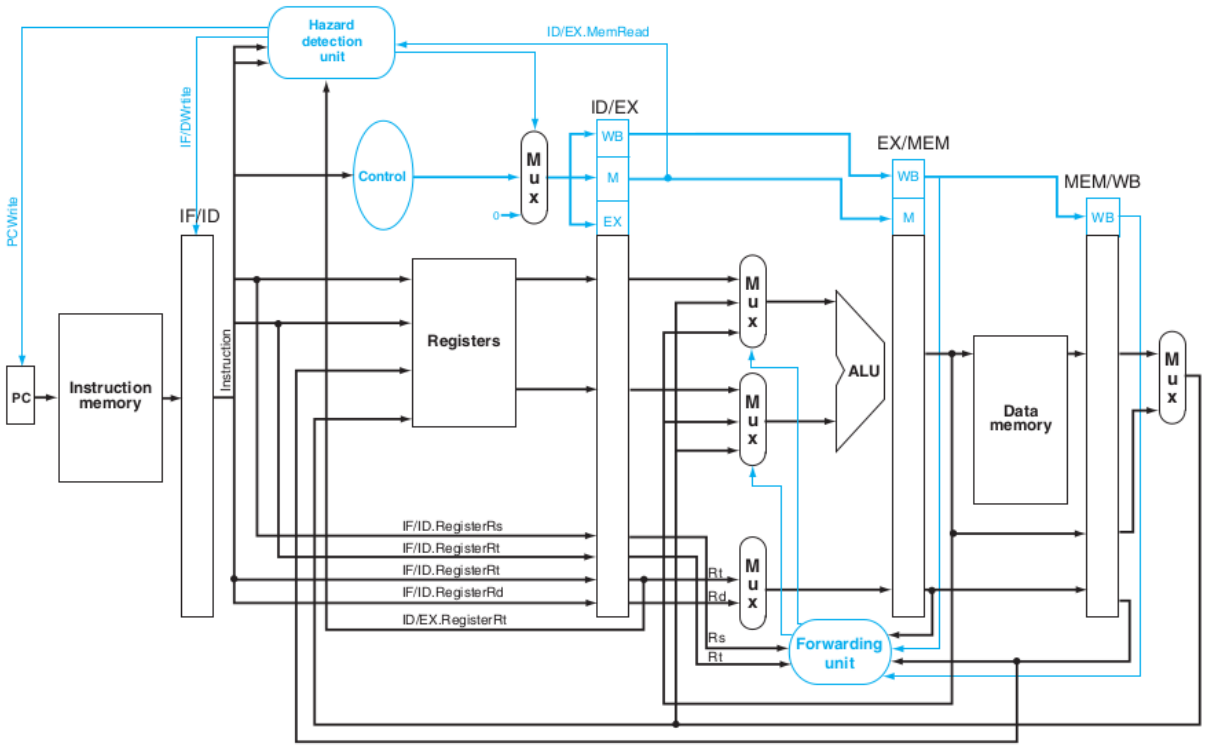
if (ID/EX.MemRead AND
(ID/EX.RegistroRt = IF/ID.RegistroRs OR
ID/EX.RegistroRt = IF/ID.RegistroRt))
Inserta una Burbuja a la segmentación

La primera condición detecta si la instrucción que está en la etapa EX es una carga; las siguientes dos condiciones evalúan si el registro a cargar (ID/EX.RegistroRt) coincide con alguno de los dos operandos de la instrucción que está en la etapa ID (IF/ID.RegistroRs o IF/ID.RegistroRt), si se cumplen las condiciones, se debe detener la ejecución de la instrucción que está en la etapa ID por un ciclo de reloj y en su lugar insertar una burbuja, esto se muestra en la próxima figura.



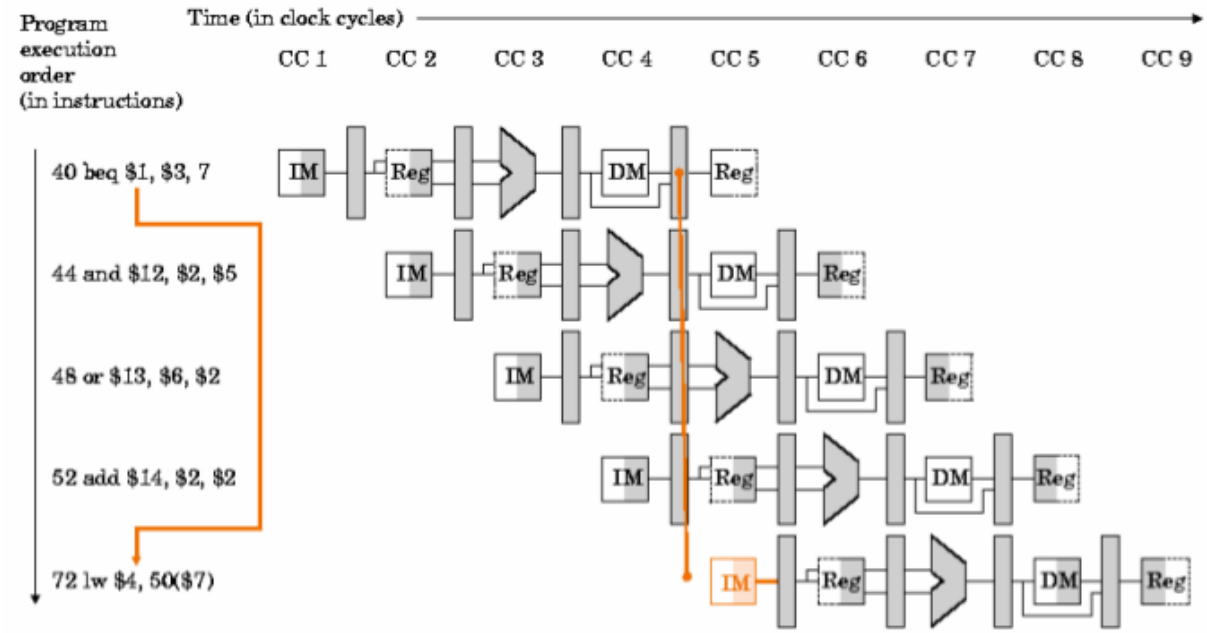
Una burbuja es un conjunto de señales que viajan sin realizar alguna operación, no escriben en registros o en memoria. Su efecto es muy similar al de una instrucción NOP, sólo que las burbujas se insertan al nivel de Hardware, retrasando a las instrucciones siguientes por un ciclo de reloj, mientras que las instrucciones NOP son parte del software.
Entonces, en la etapa ID se debe agregar un multiplexor por medio del cual la unidad de detección de riesgos pueda determinar si envía las señales de control generadas por la unidad de control o envía una burbuja (señales con valor cero).
Sin embargo, cuando se envíe una burbuja, las instrucciones que están en la etapa IF o en la etapa ID se deben detener, es decir, no deben avanzar en la segmentación. Anteriormente en estas etapas no existía alguna señal de control, puesto que no se tomaban decisiones sobre el tipo de instrucción. Ahora, es necesario agregar una señal que controle la escritura del contador del programa (PC) y del registro de segmentación IF/ID, de manera que si no se habilitan mantendrán su información actual y, por lo tanto, detendrán el avance de las dos instrucciones que están en esa etapa. La habilitación de escritura de estas señales será controlada por la Unidad de Detección de Riesgos.

En la siguiente figura se muestra la implementación segmentada con Camino de datos y control, con la unidad de detección de riesgos y la unidad de anticipación.



3.6. Riesgos de control (en instrucciones de salto)

Existe un problema en los saltos (se considerarán a los saltos sobre igual y saltos incondicionales) porque al depender de la comparación de dos registros, si se determina que el salto no se realizará, cuando eso ocurra ya habrá otras instrucciones en la segmentación.
Si se observa la figura anterior, se notará que la decisión de la ejecución de un salto se realiza en la etapa MEM, porque en la etapa anterior un sumador calculó la dirección destino del salto y la ALU hizo la resta de los dos registros para la posible generación de la bandera de zero. De manera que cuando la instrucción BEQ llega a la etapa MEM, ya se tienen los argumentos para determinar si el salto se realizará.
Hasta el momento se hizo la suposición de que los saltos no se realizaban, simplemente porque los registros bajo comparación eran diferentes. De manera que las instrucciones continuaban con su ejecución normal.
El problema principal en los saltos es que en las etapas IF, ID y EX ya hay otras instrucciones, no es complicado modificar el valor del PC para continuar en la instrucción ubicada en la dirección destino del salto, la complicación se presenta en la eliminación de las tres instrucciones que ya están en la segmentación. En la próxima figura se muestra el impacto que tienen los saltos en la segmentación.



La instrucción BEQ está en la dirección 40, y al ejecutase el PC automáticamente tomará en valor de 44. La etiqueta en BEQ corresponde a la constante 7, la cual al desplazarse a la izquierda en 2 (después de la extensión en signo) toma el valor de 28, de manera que el destino del salto corresponde a la dirección 72 (44 + 28).

Para lograr resolver este problema si en la etapa ID se detecta que se trata de un salto y se determina que éste se debe realizar, sólo se inicializaría al registro IF/ID y con ello solo se perdería un ciclo de reloj.

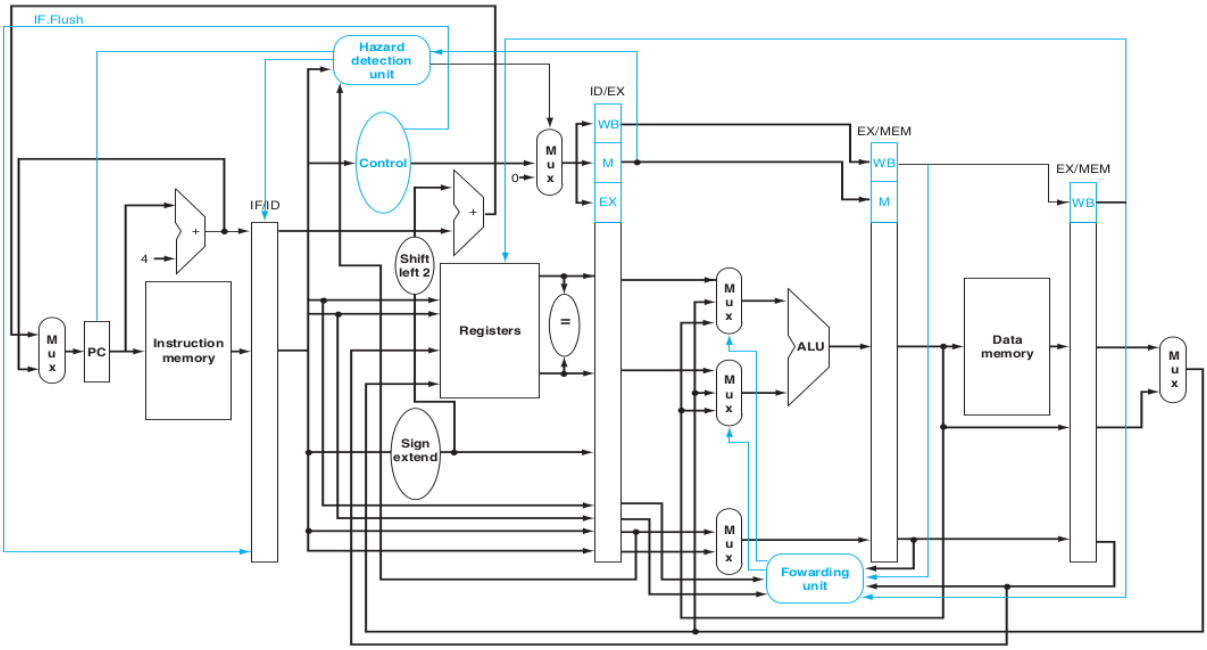
Esta idea parece congruente, pero requiere que en la etapa ID se agregue el hardware necesario para identificar a los saltos y determinar si se van a realizar. El sumador que estaba en la etapa EX, encargado de calcular la dirección destino del salto, se puede trasladar sin problema a la etapa ID, para disponer inmediatamente de esa información.

Para la comparación de los dos registros, ya no puede utilizarse a la ALU porque trasladar a la ALU a la etapa ID involucra la eliminación de una etapa y, por lo tanto, aumentaría la duración del ciclo de reloj (por que dos unidades funcionales principales trabajarían en un ciclo de reloj). De manera que se requiere de un módulo rápido dedicado a comparar el contenido de los registros, este módulo puede basarse en un conjunto de compuertas XOR, que comparen bit a bit y que sus salidas se conectan por medio de una compuerta AND. El módulo de comparación sobre igual se puede ubicar en la salida de los valores que están en los registros. Con ello, tan pronto se hizo la lectura de registros, ya se puede determinar si son iguales o diferentes.

Entonces, se agregará una señal de control adicional denominada IF.Flush, por medio de la cual se podrá limpiar el registro IF/ID. En la próxima figura se muestra el hardware resultante de trasladar la evaluación de los saltos a la etapa ID. Con estos cambios, cuando un salto se va a efectuar, habrá un retraso de un ciclo de reloj.

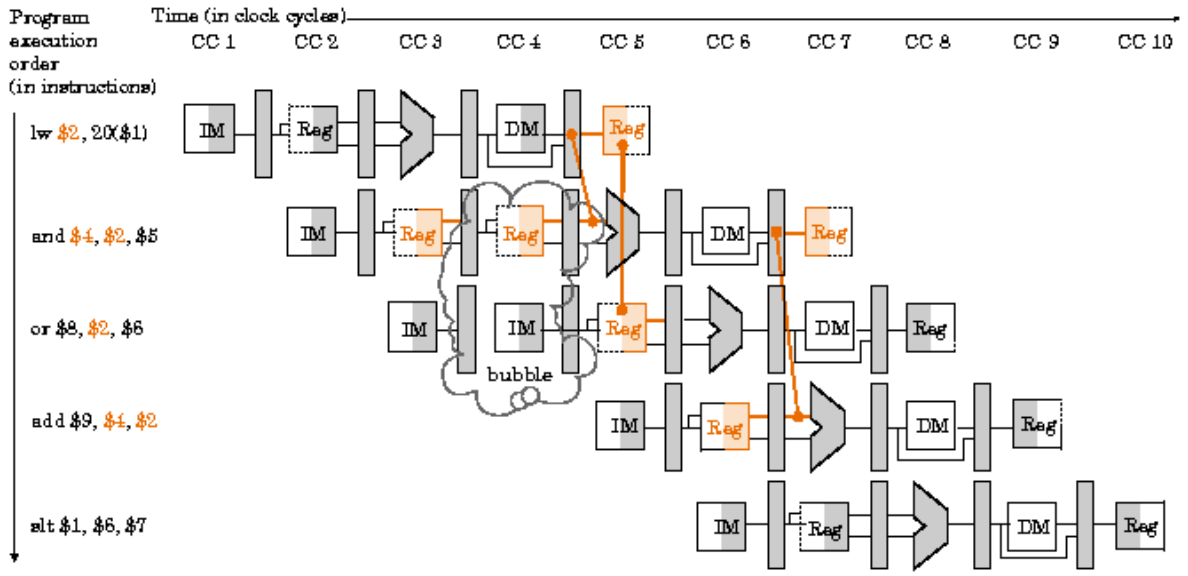
Los saltos incondicionales (J), de manera similar se pueden ejecutar en la etapa ID, una vez que se detecte el opcode de una instrucción J, el valor del PC se puede sustituir por el destino del salto y además se debe limpiar el registro IF/ID para eliminar la instrucción que está en la etapa IF.

Al trasladar la ejecución de los saltos a la etapa ID, los ciclos de reloj 3 y 4 de la secuencia mostrada en la figura anterior tendrían el comportamiento en el que sólo se inserta una burbuja (como si fuera una instrucción NOP), pero ahora lo hace la unidad de control.



3.7. Consideraciones de tiempo para la ejecución de instrucciones

Como se estuvo viendo, se destacan nuevamente las consideraciones a tener en cuenta para el correcto funcionamiento de la *unidad de detección de riesgos (Hazard detection unid)* como así también para la *unidad de anticipación (forwarding unit).* Se deben tener en cuenta un par de consideraciones con respectos a los tiempos de reloj (o clock). En un principio se remarcó que para el correcto funcionamiento de la unidad de anticipación la escritura de los registros de segmentación involucrados (EXMEM y MEMWB) se debían escribir en flancos de reloj diferentes a la escritura de los registros de propósito general (Fileregister o módulo registros) de la etapa 2. Ahora con el agregado de la unidad de detección de riesgos se hace un análisis de cómo debería funcionar el procesador MIPS con respectos a los tiempos para que las unidades de riesgos se amolden adecuadamente.

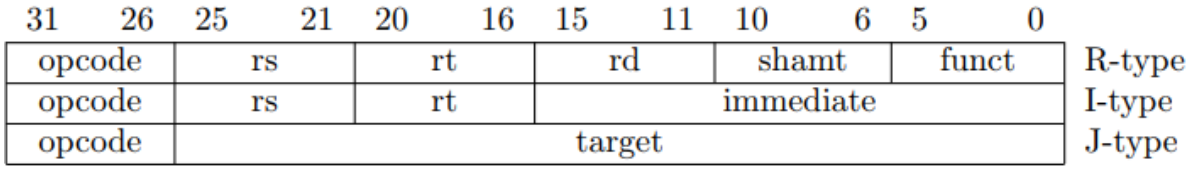


Analizando el conjunto de instrucciones anteriores, se debe suponer que si se parte de un PC que es escrito por flanco de subida (ciclo 1,) ésta es entonces la base inicial. Por lo cual se considera que en ese momento se conoce el valor del PC y luego, antes de que termine el ciclo, se aprovecha el flanco de bajada para determinar la primera instrucción, debido a esto, la memoria de programa (o memoria de instrucciones) debe operar en el flanco de bajada. En el próximo flanco de subida (ciclo 2) se registra la instrucción y el próximo PC en el reg IFID por lo que este registro es escrito en flanco de subida. Avanzando en este ciclo se tiene el flanco de bajada por lo que será utilizado para leer los registros del FileRegister, debido a esto el FileRegister opera en flanco de bajada para la lectura y opera en flanco de subida para la escritura como se verá más adelante para poder escribir y leer a la vez en un mismo ciclo.
Al comienzo del próximo ciclo de subida (ciclo 3) se registran los datos de la etapa 2 en el registro IDEX y avanzando por este ciclo durante el flanco de bajada se establecen las señales de la ALU por lo que en ese momento se registra esta información de la etapa 3 en el registro EXMEM, es por lo cual este registro opera en el flanco de bajada.
Durante el flanco de subida próximo (ciclo 4) se debe escribir o leer de la memoria de datos y es entonces por lo cual la memoria de programa se opera por flanco de subida tanto en su escritura como en su lectura, avanzando en el ciclo en su flanco de bajada es el momento en el cual se registran los datos de la etapa 4 en el registro MEMWB y es por lo cual este registro se escribe por flanco de bajada.
Por último, continúa el ciclo de subida siguiente (ciclo 5) que es el momento que se debe aprovechar para realizar la escritura del registro en el FileRegister y es por lo cual se debe utilizar el FileRegister tanto para la escritura como para la lectura con diferentes flancos de operación (escritura flanco de subida y lectura flanco de bajada en este caso) y también es por lo cual el Fileregister opera en flanco contrario a los registros EXMEM y MEMWB, como se dijo anteriormente. Todo esto es de esta forma siempre que la base inicial en la escritura del PC sea con flanco positivo como se indicó al principio, pero en caso de que la base inicial sea por flanco negativo se invertirían todos los modos de operación de acuerdo a la explicación anterior por analogía inversa.

3.8. Formato de instrucciones a implementar en el MIPS

Los formatos de instrucciones que se consideran para el procesador MIPS segmentado son:

- Instrucciones del tipo-R.
- Instrucciones del tipo-I (para carga y almacenamiento).
- Instrucciones del tipo-J (instrucciones de salto).



Las instrucciones se dividen en tres tipos: R, I y J. Todas las instrucciones empiezan con un código de operación de 6 bits. Además del código de operación, en las instrucciones tipo R se especifican tres registros (rs, rt, rd), un campo de tamaño de desplazamiento ('shamt') y otro para el código de función (funct); Las de tipo I especifican dos registros (registros fuente rs y rt) y un valor

inmediato de 16 bits; y por último en las tipo J al código de operación le siguen 26 bits de dirección destino de salto.

Instrucciones del tipo-R

Las instrucciones del tipo-R se utilizan cuando todos los valores de datos utilizados por la instrucción se encuentran en registros.

El formato de este tipo de instrucción tiene los campos que se muestran en la siguiente imagen:

opcode	rs	rt	rd	cambio (shamt)	Función
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Opcode: el opcode es la representación del código de operación de la instrucción. Varias instrucciones relacionadas pueden tener el mismo código de operación. El campo opcode tiene 6 bits de longitud (bit 26 a bit 31).

rs, rt y rd: Las representaciones numéricas de los registros de origen y el registro de destino. Estos números corresponden a la representación $ X de un registro, como $ 0 o $ 31. Cada uno de estos campos tiene 5 bits de longitud. (25 a 21, 20 a 16 y 15 a 11, respectivamente). Curiosamente, en lugar de rs y rt que se denominan r1 y r2 (para el registro de origen 1 y 2), los registros fueron llamados "rs" y "rt" porque t viene después de s en el alfabeto. Lo más probable es que se hiciera para reducir la confusión numérica.

Cambio (shamp): Utilizado con las instrucciones de cambio y rotación, esta es la cantidad por la cual el operando fuente *rs* es girado / desplazado. Este campo tiene 5 bits de longitud (6 a 10).

Función: Para las instrucciones que comparten un opcode, el parámetro **función** contiene los códigos de control necesarios para diferenciar las diferentes instrucciones. 6 bits de largo (0 a 5). Ejemplo: Opcode 0x00 accede a la ALU y el campo función selecciona la función ALU a utilizar.

Instrucciones del tipo-I

Las instrucciones del tipo-I se utilizan cuando la instrucción debe funcionar con un valor inmediato y un valor de registro. Los valores inmediatos pueden tener un máximo de 16 bits de longitud. Los números más grandes pueden no ser manipulados por instrucciones inmediatas.

El formato de este tipo de instrucción tiene los campos que se muestran en la siguiente imagen:

opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode: El código de operación de la instrucción es de 6 bits. En las instrucciones I, todos los códigos de operación tienen una correspondencia uno-a-uno con los opcodes subyacentes. Esto se debe a que no hay ningún parámetro **función** para diferenciar instrucciones con un opcode idéntico. 6 bits (26 a 31).

rs y rt: Los operandos de registro fuente y destino, respectivamente. 5 bits cada uno (21 a 25 y 16 a 20, respectivamente).

IMM: El valor inmediato es de 16 bits (0 a 15). Este valor se utiliza normalmente como el valor de compensación en varias instrucciones y, dependiendo de la instrucción, puede expresarse en complemento a dos.

Instrucciones del tipo-J

Se usan las instrucciones J cuando se necesita realizar un salto. La instrucción J tiene más espacio para un valor inmediato, porque las direcciones son números grandes.
El formato de este tipo de instrucción tiene los campos que se muestran en la siguiente imagen:

Código de operación	Pseudo-dirección

Opcode (código de operación): El código de operación es de 6 bits y correspondiente al comando de salto particular. (26 a 31).

Dirección: Una dirección abreviada de 26 bits del destino. (0 a 25). Se eliminan los dos bits menos significativos y se eliminan los 4 bits más significativos y se supone que son iguales a los de la dirección de la instrucción actual.
Algunas observaciones relacionadas con los formatos y que pueden ser importantes para el diseño del hardware son:

- Opcode (código de operación). Está siempre contenido en los bits 31-26. A este campo se lo referirá como Op[5-0].
- Los dos registros a ser leídos están siempre especificados por los campos rs y rt, en las posiciones 25-21 y 20-16. Esto para las instrucciones tipo-R y tipo-J saltos sobre igual y los almacenamientos.
- El registro base para las instrucciones de cargas y almacenamientos es rs y está en los bits 25-21.
- Los 16 bits de desplazamiento para saltos sobre igual, cargas y almacenamientos están siempre en las posiciones 15-0.
- El registro destino está en uno de dos lugares. Para las cargas está en los bits de posición 20-16 (rt), mientras que para instrucciones tipo-R su posición es 15-11 (rd). Entonces necesitamos un multiplexor para seleccionar el campo que corresponda al registro que se escribirá de acuerdo a la operación.

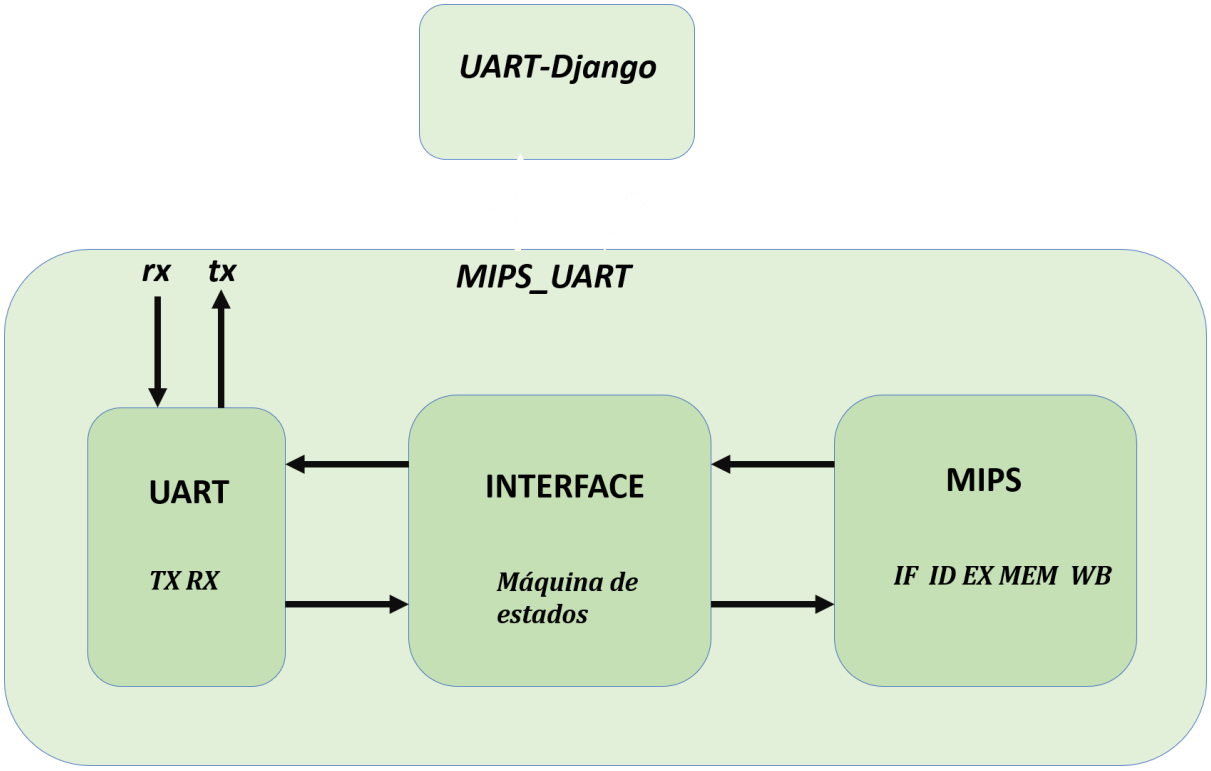
4. Desarrollo

El desarrollo para llevar a cabo el MIPS pipeline, se dividió en dos partes:

- Proyecto verilog
- Proyecto interface

Proyecto verilog

En el primer caso se desarrolló toda la codificación del pipeline en verilog según la arquitectura final vista en el marco teórico agregando una unidad UART al mismo nivel que el MIPS para poder obtener a través de una unidad de debug toda la información que se encuentra manipulando el procesador. Traducidas estas palabras en diagrama sería el siguiente:



Sobre la creación del código verilog del MIPS pipeline se tuvieron en cuentas todos los puntos claves destacados en el marco teórico pero además se agregaron las funcionalidades de cada instrucción de acuerdo a las siguientes tablas:

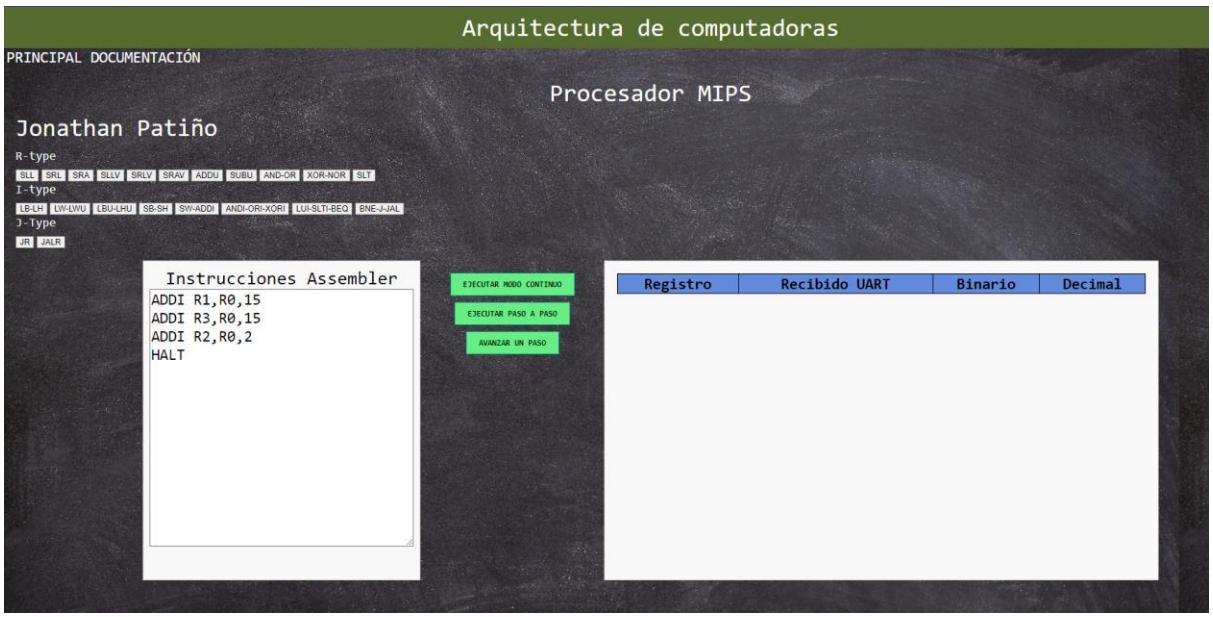
Instrcciones R-tipe						
Nemonico	**Nombre de instruccion**	**Formato**	**Operacion (en verilog)**	**opcode**	**funt**	**modo de uso**
AND	and	R	R[rd] = R[rs] & R[rt]	000000	100100	AND rd, rs, rt
OR	or	R	R[rd] = R[rs] | R[rt]	000000	100101	OR rd, rs, rt
XOR	or exclusiva	R	R[rd] = R[rs] ^ R[rt]	000000	100110	XOR rd, rs, rt
NOR	nor	R	R[rd] = ~ (R[rs] | R[rt])	000000	100111	NOR rd, rs, rt
ADDU	add unsigned	R	R[rd] = R[rs] + R[rt]	000000	100001	ADDU rd, rs, r
SUBU	sub unsigned	R	R[rd] = R[rs] - R[rt]	000000	100011	SUBU rd, rs, r
SLL	shift left logical	R	R[rd] = R[rt] << shamt	000000	000000	SLL rd, rt, shamt
SRL	shift rigth logical	R	R[rd] = R[rt] >> shamt	000000	000010	SRL rd, rt, shamt
SRA	shift rigth aritmethic	R	R[rd] = R[rt] >>> shamt	000000	000011	SRA rd, rt, shamt
SLLV	shift word left logical varible	R	R[rd] = R[rt] << R[rs]	000000	000100	SLLV rd, rt, rs
SRLV	shift word rigth logical variable	R	R[rd] = R[rt] >> R[rs]	000000	000110	SRLV rd, rt, rs
SRAV	shift word rigth aritmethic variable	R	R[rd] = R[rt] >>> R[rs]	000000	000111	SRAV rd, rt, rs
SLT	set less than	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	000000	101010	SLT rd, rs, rt
JR	jump register	R	PC = R[rs]	000000	001000	JR rs
JALR	jump and link register	R	R[rd] = PC +8; PC = R[rs]	000000	001001	JALR rd, rs

Instrcciones I-tipe					
Nemonico	**Nombre de instruccion**	**Formato**	**Operacion (en verilog)**	**opcode**	**modo de uso**
LB	load byte	I	R[rt] = M[R[rs]+SignExtImm](7:0)	100000	LB rt, offset(rs)
LH	load halfword	I	R[rt] = M[R[rs]+SignExtImm](15:0)	100001	LH rt, offset(rs)
LW	load word	I	R[rt] = M[R[rs]+SignExtImm]	100011	LW rt, offset(rs)
LBU	load byte unsigned	I	R[rt]={24'b0,M[R[rs] + SignExtImm](7:0)}	100100	LBU rt, offset(rs)
LHU	load halfword unsigned	I	R[rt]={16'b0,M[R[rs] + SignExtImm](15:0)}	100101	LHU rt, offset(rs)
LWU	load word unsigned	I	R[rt]={32'b0,M[R[rs] + SignExtImm]}	100111	
SB	store byte	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	101000	SB rt, offset(rs)
SH	store halfword	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	101001	SH rt, offset(rs)
SW	store word	I	M[R[rs]+SignExtImm] = R[rt]	101011	SW rt, offset(rs)
ADDI	add inmediate	I	R[rt] = R[rs] + SignExtImm	001000	ADDI rt, rs, zero-ext-imm.
ANDI	and inmediate	I	R[rt] = R[rs] & SignExtImm	001100	ANDI rt, rs, zero-ext-imm.
ORI	or inmediate	I	R[rt] = R[rs] | SignExtImm	001101	ORI rt, rs, zero-ext-imm.
XORI	xor inmediate	I	R[rt] = R[rs] ^ SignExtImm	001110	XORI rt, rs, zero-ext-imm.
LUI	load upper inmediate	I	R[rt] = {imm, 16'b0}	001111	LUI rt, zero-ext-imm.
SLTI	set less than inmediate	I	R[rd] = (R[rs] < SignExtImm) ? 1 : 0	001010	SLTI rt, rs, signed-imm.
BEQ	branch on equal	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	000100	BEQ rs, rt, offset
BNE	branch on no equal	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	000101	BNE rs, rt, offset

Instrcciones J-tipe					
Nemonico	**Nombre de instruccion**	**Formato**	**Operacion (en verilog)**	**opcode**	**modo de uso**
J	jump	J	PC=JumpAddr	000010	J target
JAL	jump and link	J	R[31]=PC+8;PC=JumpAddr	000011	JAL target

Proyecto interface

Por otro lado, también fue necesario armar una interfaz adecuada para poder interpretar de manera correcta cada uno de los valores obtenidos en el procesador MIPS. Para estos se utiliza como herramienta de interface el navegador a través de codificación HTML para observar los resultados y por otro lado se utiliza el programas Python para el control del MIPS y comunicación con el mismo a través de UART.



El proyecto, como se puede apreciar en la figura anterior, consta de dos paneles y tres botones en el centro. En el panel izquierdo, se debe cargar el programa que se va a ejecutar. Para llevar a cabo la carga y ejecución del programa, se debe elegir uno de los tres botones en el panel central. El primer botón realizará una ejecución continua, el segundo activará el modo paso a paso, y el tercero avanzará un paso. Los resultados se mostrarán en el panel de la derecha.
Además, hay una opción de precarga de las funciones a realizar, que se encuentra en la parte superior izquierda. Desde allí, se pueden seleccionar las instrucciones que se desean llevar a cabo. También es notable la presencia de una pestaña denominada "Documentación" donde se encuentra información relacionada con el funcionamiento y detalles del MIPS.

Conclusión:

La realización del presente trabajo ha dejado una gran enseñanza respecto al correcto diseño que deben tener en cuanto a su arquitectura cada uno de los procesadores. En este caso el MIPS pipeline es un procesador que se diseña dividiendo por un lado el camino de la instrucción, y por el otro, el control; y, además, como este procesador tiene la capacidad de explotar el paralelismo con el uso de la segmentación -lo que lo hace más rápido y eficiente en cuanto al uso de los ciclos reloj- se deberán tener en cuenta siempre los riesgos que puedan presentarse según sea la instrucción a ejecutar y los registros o recursos que utilicen cada una de esas instrucciones.

El trabajo de pruebas realizado en el presente fue un gran desafío ya que una vez realizada cada uno de las funcionalidades generadas se debería verificar si el resto ya realizado funcionaba aun correctamente y por la cantidad de instrucciones, el tiempo de verificación y validación resultó muy grande.

Por último, se puede decir que con este trabajo se tiene una muy buena base para poder crear proyectos futuros asociados a proyectos similares sobre los que se pueda trabajar buscando facilitar la enseñanza de este tipo de proyectos asociados a la arquitectura de las computadoras.