Microarquitectura (uArch)

David Alejandro González Márquez

Programación de softcores en FPGAs
Programa de Profesoras/es Visitantes
Departamento de computación
Universidad de Buenos Aires

Clase disponible en: `https://github.com/fokerman/fpgaSoftcoreProgrammingCourse`

Microarquitectura

La **microarquitectura** de un procesador hace referencia a su diseño interno.
Usualmente denominada la **organización** del procesador.

En esta clase vamos a presentar una **implementación de la arquitectura MIPS**.

Primero proponiendo un diseño simple donde se presenten las técnicas utilizadas para pensar y construir un *datapath*. El objetivo es diseñar un sistema que responda al procesamiento de instrucciones, buscando ejecutarlas lo **más rápido posible**. **En un solo ciclo de reloj**.

Para luego extender está solución aplicando la **técnica de segmentación** o ***pipeline***.
Donde nos vamos a encontrar con múltiples problemas derivados de separar en partes las instrucciones.

Arquitectura MIPS

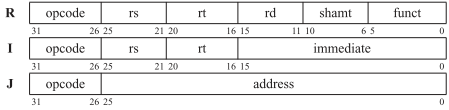
Tamaño de palabra de 32 bits, con instrucciones de 32 bits.
32 registros en total entre dedicados y propósito general.

Tres tipos de codificación de instrucciones.

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

BASIC INSTRUCTION FORMATS



CORE INSTRUCTION SET

NAME	MNEMONIC	FORMAT	OPERATION
Add	add	R	R[rd] = R[rs] + R[rt]
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]
And	and	R	R[rd] = R[rs] & R[rt]
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr
Jump	j	J	PC=JumpAddr
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr
Jump Register	jr	R	PC=R[rs]
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])
Or	or	R	R[rd] = R[rs] | R[rt]
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0
Set Less Than Unsig	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]
Subtract	sub	R	R[rd] = R[rs] - R[rt]
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]

Implementación básica de un MIPS

Vamos a estudiar una implementación limitada del conjunto de instrucciones de la arquitectura MIPS.

- **Instrucciones de acceso a memoria:** *load word* (`lw`), *store word* (`sw`)
- **Instrucciones artiméticas y lógicas:** `add`, `sub`, `AND`, `OR`, `slt`
- **Instrucciones de control:** *branch equal* (`beq`), *jump* (`j`)

Estas subconjunto de instrucciones no incluye todas soportadas en la ISA.
Pero ilustra los principios de funcionamiento usados para crear y diseñar el *datapath*.

Todos los conceptos que se ilustran en está implementación son las ideas básicas que se utilizan para diseñar desde procesadores de alto rendimiento, o proposito general, hasta dispositivos embebidos.

Descripción general de la implementación

En MIPS, en general todas las instrucciones requieren pasos similares.

Los primeros pasos son siempre los mismos:

1. Enviar el PC a la memoria y leer el contenido (*instruction fetch*).
2. Leer uno o dos registros usando campos en la instrucción.

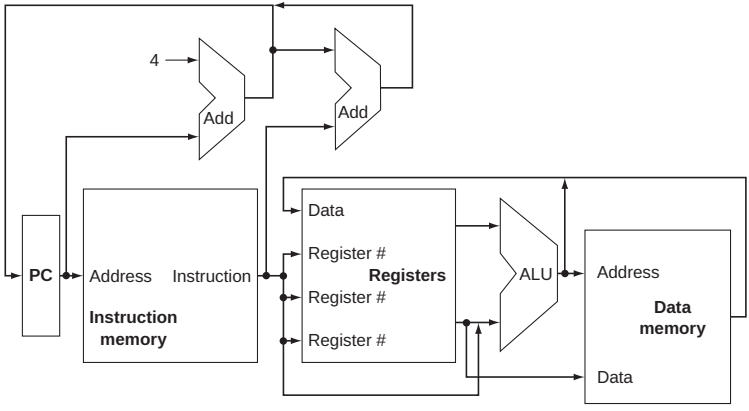
Luego de estos dos pasos, las acciones dependen del tipo de instrucción.

Existen tres tipos de instrucciones:

1. memory-reference
2. arithmetic-logical
3. branches

La simplicidad del conjunto de instrucciones permite que la operatoria de cada uno de los tipos de instrucciones sea similar.

Vista general de la implementación



Observaciones

- De la **instrucción** se toman datos para el banco de registros, el PC y la ALU.
- El **nuevo PC** puede provenir de dos flujos diferentes.
- Escribir al **banco de registros** llega desde la memoria o desde la ALU.
- A la **memoria** solo llegan datos del banco de registros.

La ilustración indica los componentes básicos del *datapath* y como se **interconectan**.
La idea es identificar especificamente el **flujo de los datos** dentro del *datapat*.

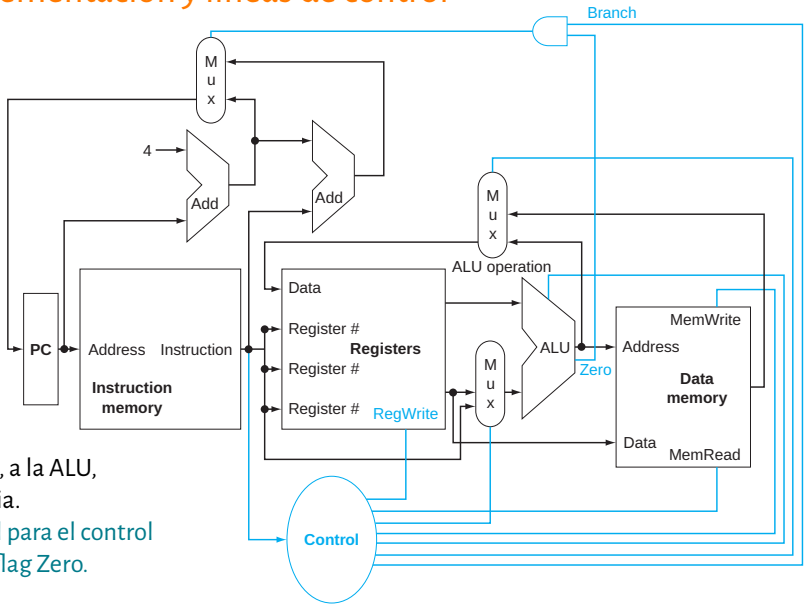
Vista general de la implementación y líneas de control

Se agregan **líneas de control**, junto con tres multiplexores.

1. Escritura en el PC.
2. Escritura en el banco de registros.
3. Entrada a la ALU.

Las líneas de control toman como entrada la **instrucción**.

Llegan a los tres multiplexores, a la ALU, al banco de registros y memoria.
Además de una línea adicional para el control de saltos condicionales por el flag Zero.



Diseño Lógico

El *datapath* contiene dos tipos de componentes:

- Componentes que dependen **solo de sus entradas** (Operación combinacional).
- Componentes que **mantienen un estado** (Operación secuencial).

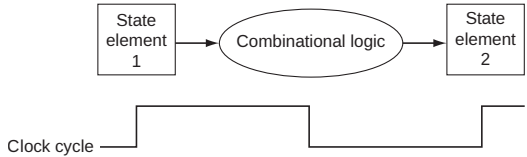
Los elementos con estado soportan el comportamiento de **lectura** y **escritura**.
Contienen al menos dos entradas y una salida.

Una de las entradas indica el dato que será escrito, y otra el clock para sincronizar el momento que de debe ser escrito. La salida permite obtener el dato escrito en el ciclo anterior.

Además de los flip-flops que respetan esté comportamiento tenemos dos tipos de elementos con estado: **Memorias** y **Registros**.

Metodología de sincronización

La sincronización define cuando las señales serán leídas y cuando serán escritas en función del reloj.



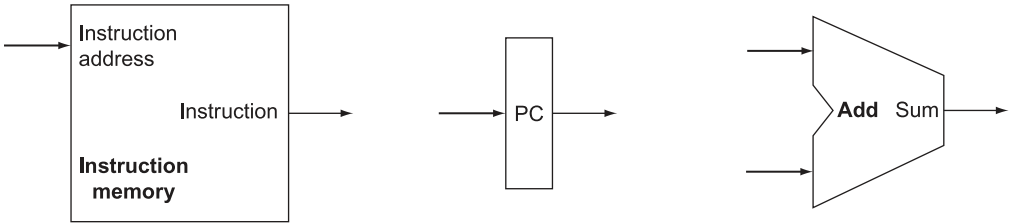
Por simplicidad asumimos que los cambios serán por flanco ascendente de reloj.

Como solo los elementos con estado pueden guardar información, **cualquer circuito combinacional tiene como entradas datos provenientes de elementos con estado**. Luego, las salidas son nuevamente guardadas en otros elementos con estado.

Esta metodología nos permite **realimentar** la escritura de los elementos con estado por medio de un circuito combinacional.

Construyendo un Datapath

Usando como guía el ciclo de instrucción. Vamos a resolver la **etapa de *fetch***.

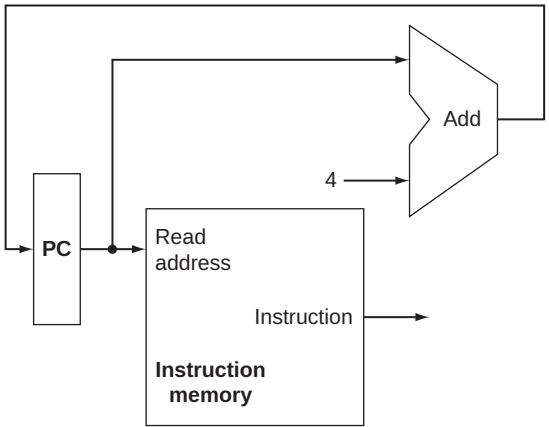


La memoria contiene las instrucciones que **leemos** según indique la dirección en el PC.

Para pasar a la próxima instrucción debemos **incrementar** el PC usando el sumador.

Ambas acciones, tanto leer de memoria y actualizar el PC se pueden realizar **en el mismo ciclo**.

Construyendo un Datapath: *Fetch*



El PC llega tanto al puerto de direcciones de la **memoria** como al **sumador**.

El sumador es un circuito combinacional que genera el nuevo PC sumandole 4 unidades.

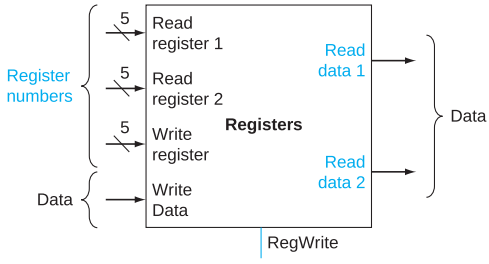
La instrucción leída de la memoria se presenta en la **salida de la memoria** luego del ciclo de reloj.

Simultáneamente el PC cambia su estado pasando a la **próxima dirección**.

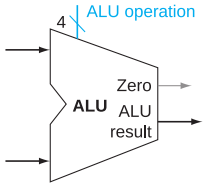
Construyendo un Datapath: *R-format instructions*

Las instrucciones *R-format* operan con tres registros.
Dos como entrada y uno como salida.
En un banco de registros se puede seleccionar cualquiera de los registros para ser **escrito o leído**.

Los puertos de lectura permiten seleccionar simultáneamente dos registros para ser leídos.



Los datos leídos **entran** como parámetros directamente en la ALU que realiza la operación seleccionada y genera un resultado que es almacenado nuevamente en el banco de registros.



MIPS cuenta con 32 registros (2^5), y cada registro es de 32 bits.

Construyendo un Datapath: *load/store instructions*

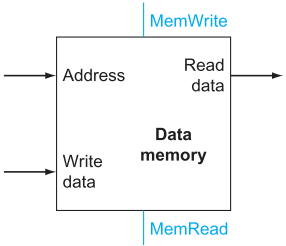
Las instrucciones *load/store* son de la forma:

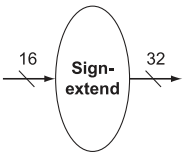
- LOAD: `lw $t1, offset_value($t2)`
- STORE: `sw $t1, offset_value($t2)`

El primer parámetro es un registro fuente o destino del dato.
El segundo parámetro es un registro más un offset de 16 bits.

Para poder operar con el offset de manera signada se tiene un componente que extiende el signo a 32 bits.

La codificación de los registros en cada caso es la misma.
Las señales de control indican si se trata de una lectura o escritura.





Construyendo un Datapath: *branch instructions*

La instrucción salto por igualdad es de la forma:

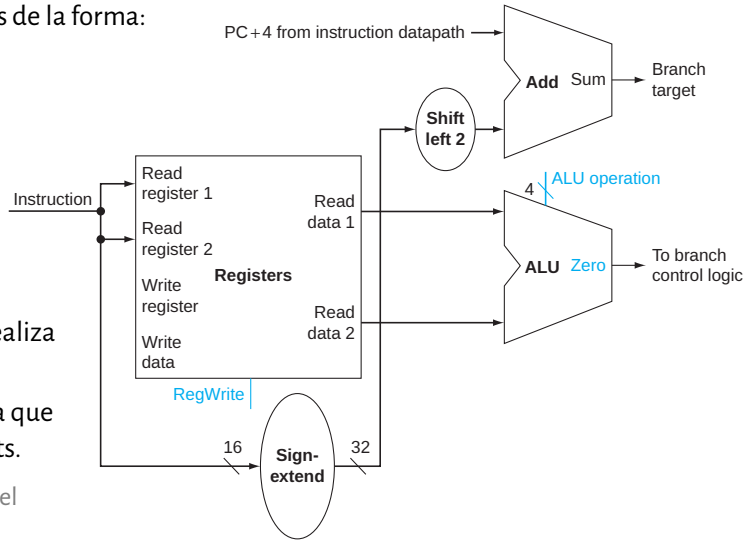
```
beq $t1, $t2, offset
```

Toma tres operandos, dos registros que serán comparados por igualdad y un offset de 16 bits utilizado para calcular la dirección destino del salto.

Observaciones

- El calculo de la dirección se realiza sobre el valor del PC + 4.
- El offset se multiplica por 4 ya que las instrucciones son de 32 bits.

La ALU provee una señal que indica si el resultado de la operación es cero.



Construyendo un Datapath: Observaciones

Examinamos los distintos caminos posibles para resolver las diferentes instrucciones.
Ahora buscamos combinar todo en un solo *datapath* y agregar las señales de control necesarias.

Esta solución permitirá **ejecutar una instrucción por ciclo de reloj**.
Esto significa que **ningún recurso puede ser usado más de una vez por instrucción**.

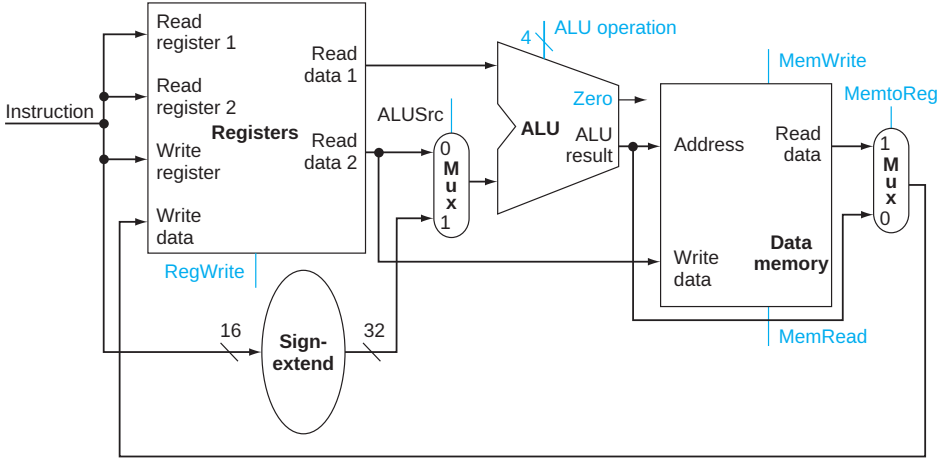
De ser necesario, los recursos deben ser:

- **Duplicados**: Como sucede con los sumadores para calcular direcciones y operaciones.
- **Compartidos**: Como sucede con el banco de registros y sus puertos.
- **Divididos**: Como sucede con las memorias de datos y código.

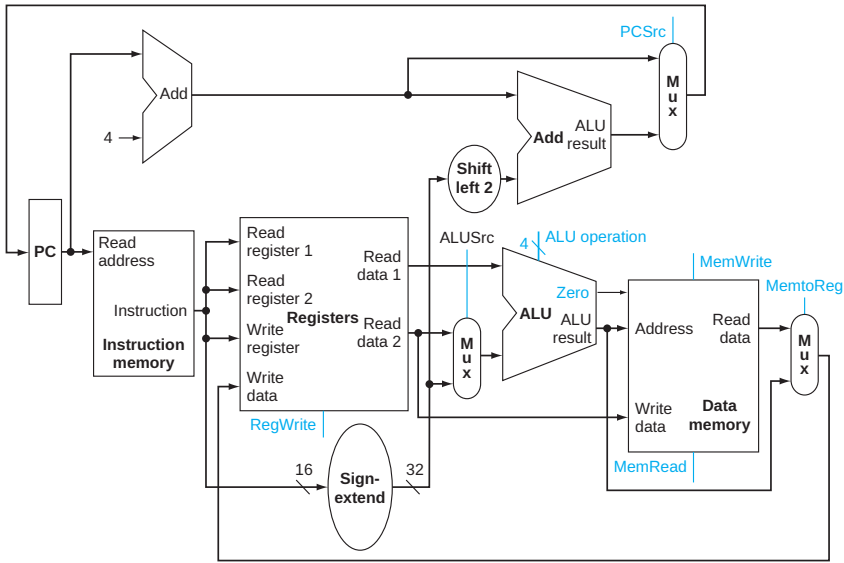
Es fundamental como parte del diseño de un *datapath* entender
como las instrucciones utilizarán los componentes en cada ciclo.

Construyendo un Datapath

Colocando todos los componentes juntos para resolver las instrucciones *R-format*



Simple Datapath



Control de la ALU

Supongamos una ALU limitada a 5 operaciones. Dependiendo del tipo de instrucción, la ALU deberá realizar diferentes operaciones.

Para simplificar el control podemos identificar la operación a realizar utilizando los bits de función y 2 bits de control para seleccionar la operación.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Los bits de control (ALU0p) indican:

- `00` Operación ADD. Para *load/store*.
- `01` Operación SUB. Para *branch equal*.
- `10` Determinado por los bits de función. Para *R-type*

Instruction opcode	ALU0p	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Control de la ALU

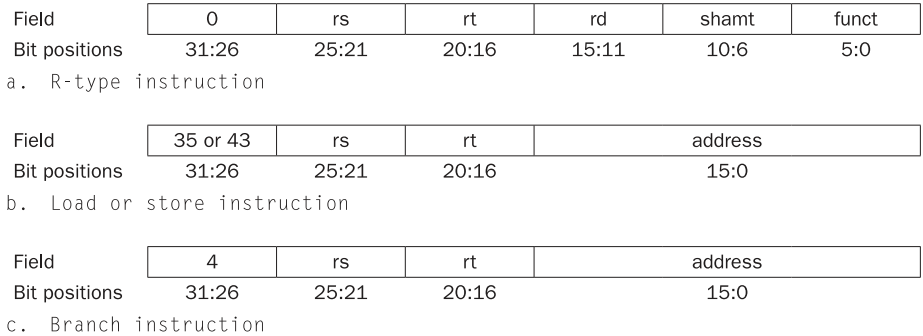
Considerando todos los bits obtenemos la siguiente tabla de verdad.

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Observar que la tabla contiene x (don't-care) bits.
Como ALUOp no puede valer 11, se considera x1 y 1x en la tabla de verdad.
Los bits de F5 y F4 siempre son siempre 10 para los últimos 5 casos, por lo tanto se ignoran.

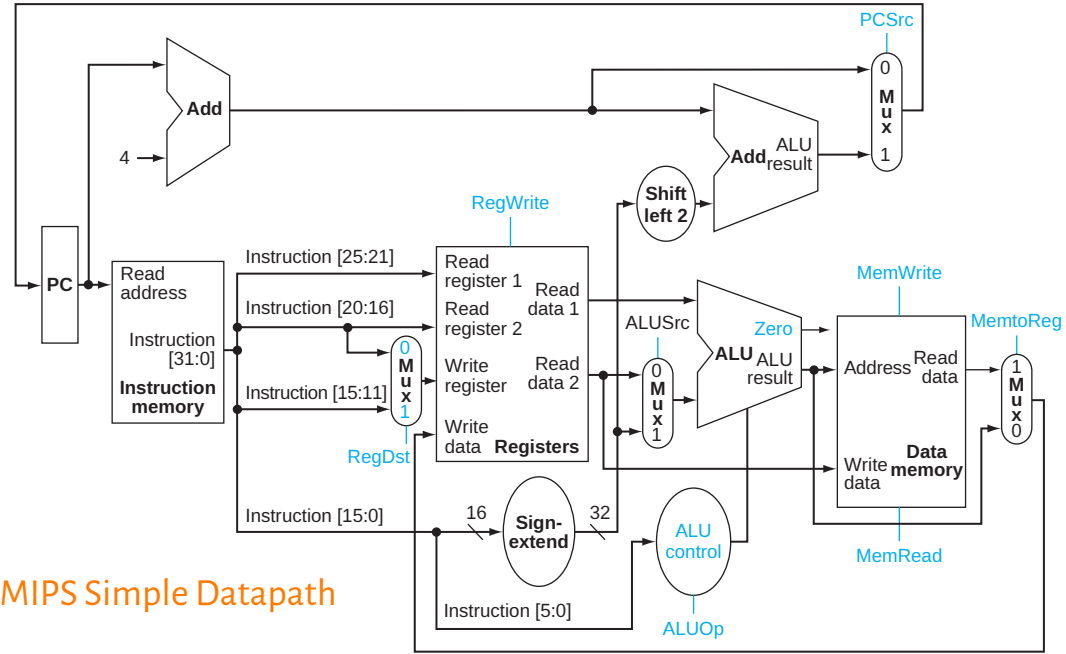
Esta tabla ahora puede ser optimizada al momento de ser convertida en compuertas.

Formatos de instrucción

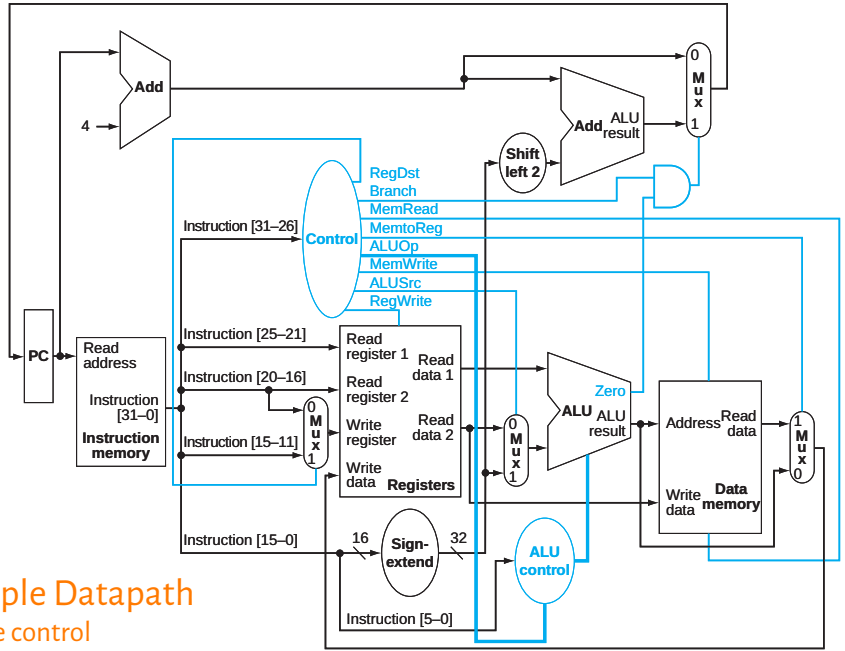


- Los bits 31:26 refieren al Opcode (`Op[5:0]`). Los registros a leer `rs` (bits 25:21) y `rt` (bits 20:16).
- El registro base para *load/store* es `rs` en bits 25:21.
- El offset de 16-bit para *branch equal* y *load/store* se toma de los bits 15:0.
- El registro destino puede ser `rt` para *load* y `rd` para instrucciones *R-type*.

Vamos a requerir un multiplexor adicional para seleccionar el registro donde escribir.



MIPS Simple Datapath



MIPS Simple Datapath

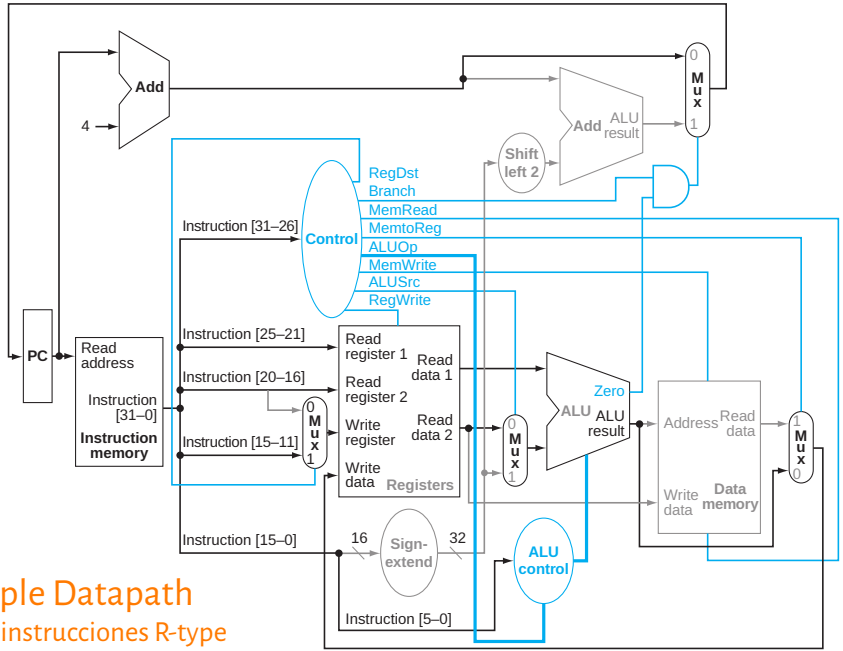
Con lineas de control

Unidad de control

La unidad de control funciona como una máquina de estados, sin embargo,
en este caso cada instrucción que se realiza **en un solo ciclo**.
Por lo tanto la codificación de cada instrucción corresponde a un conjunto de señales de control.

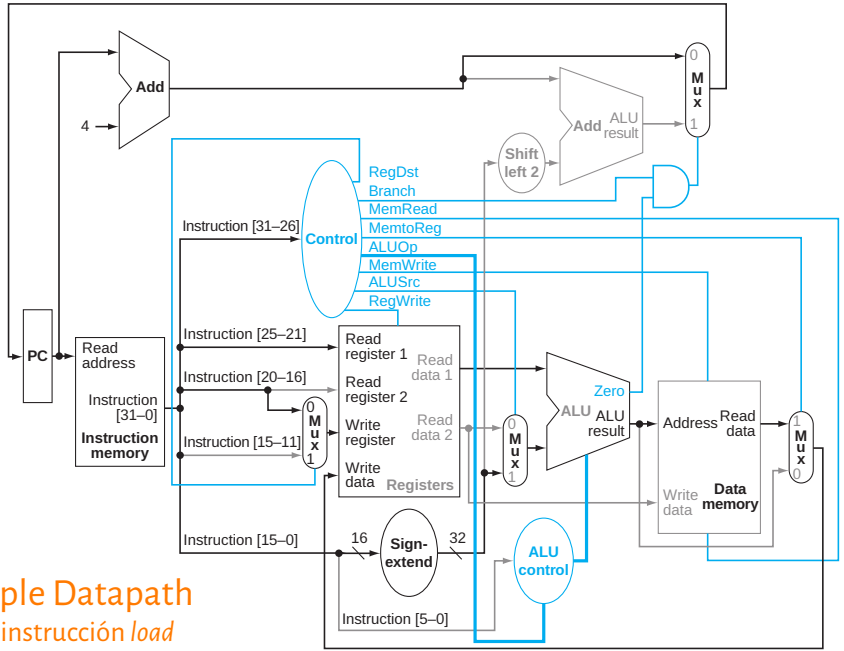
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

1. Instrucciones *R-format*. Los registros `rs` y `rt` se utilizan como fuente, y `rd` como destino (`RegDst=1`). Los bits de `ALUOp` seleccionan `10` para usar los bits de función.
2. Los bits de `ALUOp=00` para sumar con la ALU. Las señales para leer de memoria y escribir en `rt` (`RegDst=0`).
3. Los bits de `ALUOp=00` para sumar con la ALU. Las señales para escribir en memoria y leer de `rt`.
4. Los bits de `ALUOp=01` para restar con la ALU. La señal de `branch=1` para saltar condicionalmente.



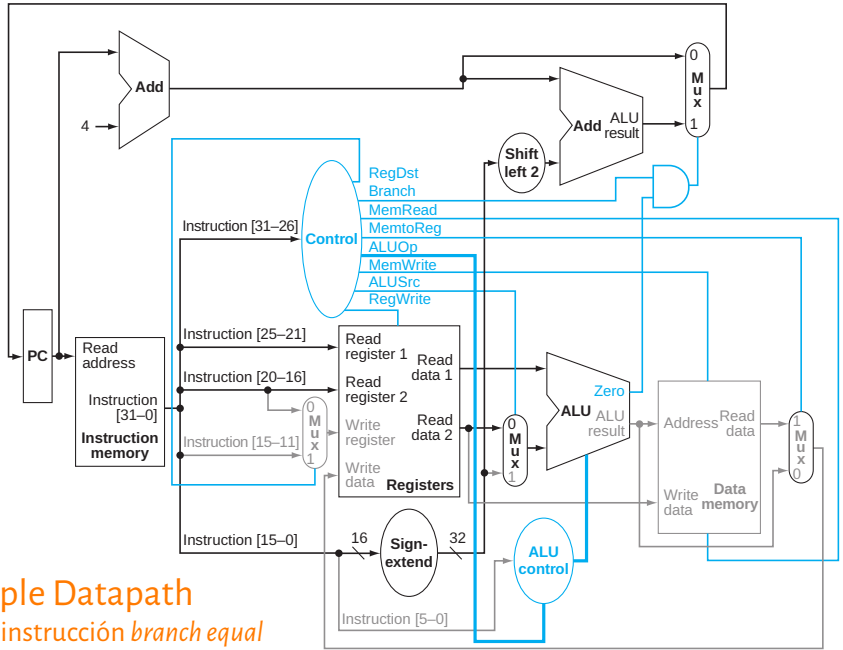
MIPS Simple Datapath

Control para instrucciones R-type



MIPS Simple Datapath

Control para instrucción *load*



MIPS Simple Datapath

Control para instrucción *branch equal*

Unidad de control

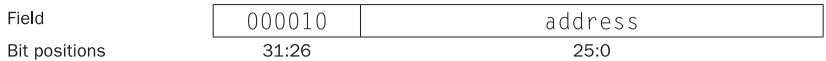
Luego queda construir los estados de la unidad de control.

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

La tabla considera como entradas el opcode y salidas las señales de control de nuestro *datapath*.

Instrucción *jump*

Nos resta agregar en nuestro *datapath* la instrucción para saltos incondicionales.



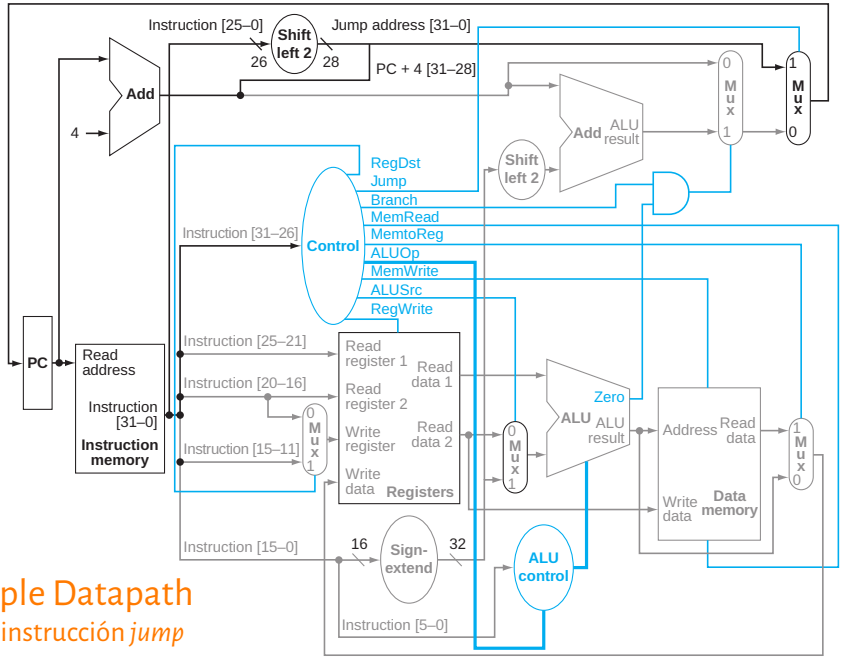
En MIPS está instrucción calcula la dirección de la siguiente forma:

- Se toma un inmediato de 26 bits de la instrucción.
- Los dos bits menos significativos se los completan con `0`.
- Los cuatro bits más significativos corresponden al valor del PC+4.

Con esto tenemos un valor de 32 bits para usar como dirección destino.

Al *datapath* debemos agregar:

- Tomar los 26 bits como parte de la decodificación de la instrucción.
- Un componente que realice el shift de 2 bits.
- Un componente que combine los bits entre el PC+4 y el valor calculado.



MIPS Simple Datapath

Control para instrucción *jump*

¿Podemos hacer algo mejor?

Si bien el diseño en un solo ciclo es funciona, no es utilizado ya que es **ineficiente**.

Notar que cada instrucción debe demorar **lo mismo** en un diseño *single-cycle*.
Esto implica que todas van a demorar lo mismo que **el camino más costoso**.
Este camino es una instrucción de *load*, que utiliza **todas las unidades funcionales**.

La penalidad de un diseño con **el tiempo de reloj fijo** es muy alta y solo puede ser considerada para conjuntos de instrucciones pequeños.

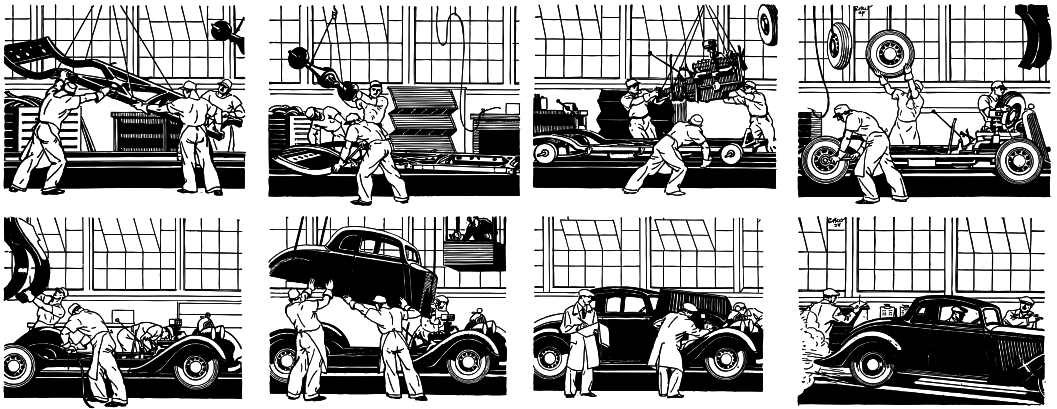
Si nuestro conjunto de instrucciones es complejo, o incluye instrucciones de punto flotante, **debemos asumir el peor retardo para todas las instrucciones**, cualquier esfuerzo por reducir el retardo para el caso común no va a mejorar el peor caso.

Un diseño *single-cycle* no respeta el objetivo de hacer el caso común lo más rápido posible.

Segmentación (*Pipelining*)

Pipelining es una técnica de implementación de procesadores en la cual **se superpone la ejecución** de múltiples instrucciones. La idea es **dividir el trabajo** en etapas (*stages*), donde en cada etapa se puede ejecutar una sola instrucción a la vez.

Veamos un ejemplo con una fabrica de autos.



Segmentación (*Pipelining*)

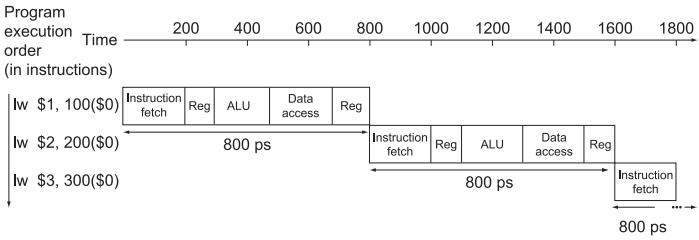
Usando la misma idea aplicada a las instrucciones de un procesador podemos construir un *pipeline* o un procesador segmentado.

Un diseño posible para MIPS es el ***pipeline* de cinco etapas**.

1. Leer la instrucción de memoria (*Fetch*).
2. Leer registros mientras se decodifica la instrucción.
3. Ejecutar la operación o calcular una dirección.
4. Acceder a los operandos en memoria.
5. Escribir el resultado en un registro.

Single-Cycle vs *Pipelined*

Implementación *single-cycle*: Cada instrucción demora un ciclo de reloj.

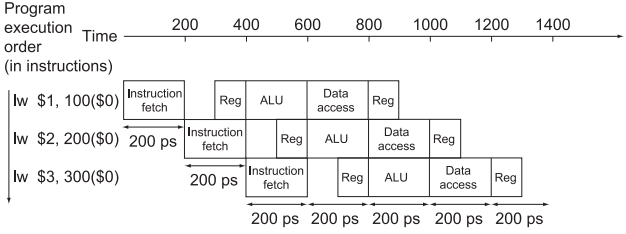


Ejemplo:

200 ps para accesos a memoria.
200 ps para operaciones (ALU).
100 ps para acceder a registros.

El ciclo se debe acomodar al tiempo de la instrucción más lenta.

Implementación *pipelined*: Cada etapa esta acotada al tiempo de la etapa más costosa.



Cada etapa se debe diseñar para el peor caso de 200 ps, incluso si algunas etapas pueden demorar menos.

Diseño de un ISA para una implementación Segmentada

El diseño de un ISA debe respetar **máximas** que faciliten su implementación segmentada.
En particular, MIPS las respeta.

- **Todas las instrucciones de la misma longitud.**
  Esto facilita la decodificación y simplifica la etapa de *fetch*. En un x86 donde el tamaño de las instrucciones es variable y está tarea es mucho más desafiante.
- **Tener pocos formatos de instrucción.**
  Con campos de parámetros en los mismos lugares para cada instrucción. Está regularidad permite acceder al banco de registros al mismo tiempo que se determina la operación de la instrucción. Si no fueran regulares deberíamos generar más etapas del *pipeline* para decodificar y acceder a los registros.
- **Los operandos de memoria solo aparecen para escrituras y lecturas a memoria.**
  Podemos usar la etapa de ejecución para calcular la dirección de memoria y luego acceder a la memoria en la siguiente etapa. Si pudiéramos operar con operandos en memoria, como en el x86, se deberían crear más etapas.
- **Los operandos deben estar alineados a memoria.**
  Nos permite asegurarnos que se requiere una sola transferencia entre el procesador y la memoria.

Pipeline Hazards

Existen situaciones en las que la próxima instrucción **no puede ser ejecutada** en el siguiente ciclo. Estos eventos se denominan ***hazards***.

Structural Hazard

Sucede cuando el hardware no puede soportar una determinada combinación de instrucciones en el mismo ciclo. Por ejemplo, si la memoria de instrucciones y datos fuera la misma, no se podría en el mismo ciclo hacer una escritura a memoria y una lectura de la nueva instrucción.

Data Hazard

Ocurre cuando el *pipeline* debe ser detenido (*stall*) porque se requiere que la instrucción anterior esté completa. Por ejemplo, las instrucciones `add $s0, $t0, $t1`; `sub $t2, $s0, $t3`; dependen que `$s0` tenga el valor de la suma antes de ejecutar la resta. Las instrucciones dependen entre sí.

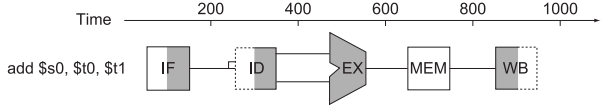
Control Hazard

Aparece cuando se debe tomar una decisión basada en el resultado de una operación. Luego del *fetch* debemos calcular cual es la próxima instrucción. Pero si esta es un salto condicional, debemos esperar a que la misma se ejecute, esperando el calculo del nuevo PC.

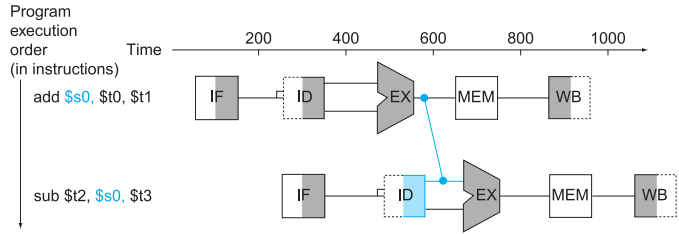
Data Hazard - *Forwarding*

La solución está basada en la observación:
no es necesario esperar a que se termine la instrucción para resolver el problema.

Por ejemplo, para la secuencia de suma y resta, podemos, una vez terminada la suma en la ALU, usar ese resultado en la resta.



Forwarding o ***Bypassing***
Hardware adicional para identificar el dato faltante desde los recursos internos y proveerlo para usarlo en la próxima instrucción.

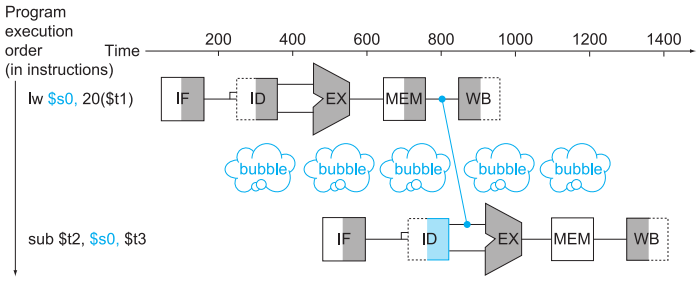


Data Hazard - *Forwarding*

No siempre es posible disponer del dato para la próxima instrucción.

Por ejemplo,

- Si la primer instrucción es un `load` al registro `$s0`.
- El dato para operar con la siguiente instrucción recién lo vamos a obtener luego de la etapa 4 donde leemos a memoria.
- Este dato lo vamos a tener muy tarde para llegar a hacer un *forwarding*.



En este caso estamos obligados a agregar un *stall* en el *pipeline*. Usualmente la etapa sin operación se denomina *bubble*.

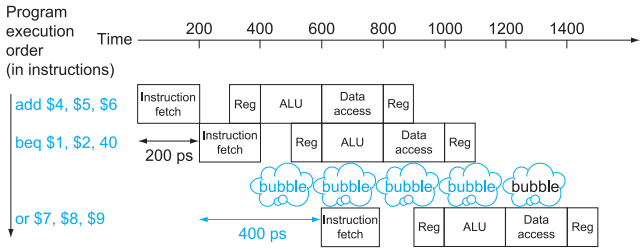
Hardware especial permite detectar estos casos y generar los *bubble* artificialmente. Incluso podemos reordenar el código para reducir la necesidad de tener *pipeline stalls*.

Control Hazard - *Branch Prediction*

La dirección del salto recién será resuelta en la segunda etapa.

Por lo tanto vamos a tener que esperar un ciclo más antes de conocer la nueva instrucción.

Esto obliga a agregar un ciclo *stall*.



Las instrucciones de salto representan en promedio un 17 %,
por lo tanto para todas estas instrucciones **desperdiciariamos un ciclo**. **¿Podemos hacer algo mejor?**

¡Predecir!

La idea es simple, podemos asumir que los saltos nunca se toman, y si se llegan a tomar, pagamos el costo de un *stall*. La instrucción se ejecutará especulativamente, sin escribir nada y luego se confirma.

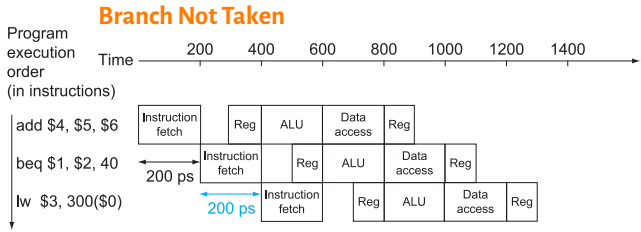
Control Hazard - *Branch Prediction*

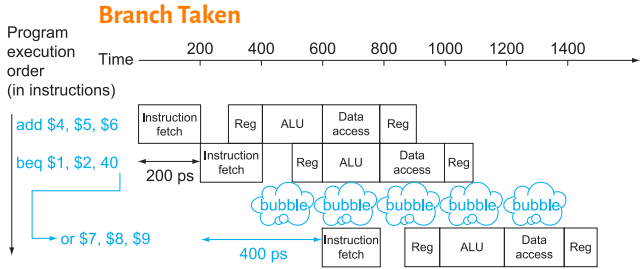
Procesadores modernos, utilizan sofisticados predictores de saltos.

La predicción dinámica utiliza información historica del programa y cambia su predicción durante la vida de este.

La precisión de los predictores de saltos es mayor al 90 %

Otra técnica es *delayed branch*, donde se demora la decisión de tomar el salto, agregando una instrucción luego del salto que se ejecuta siempre.





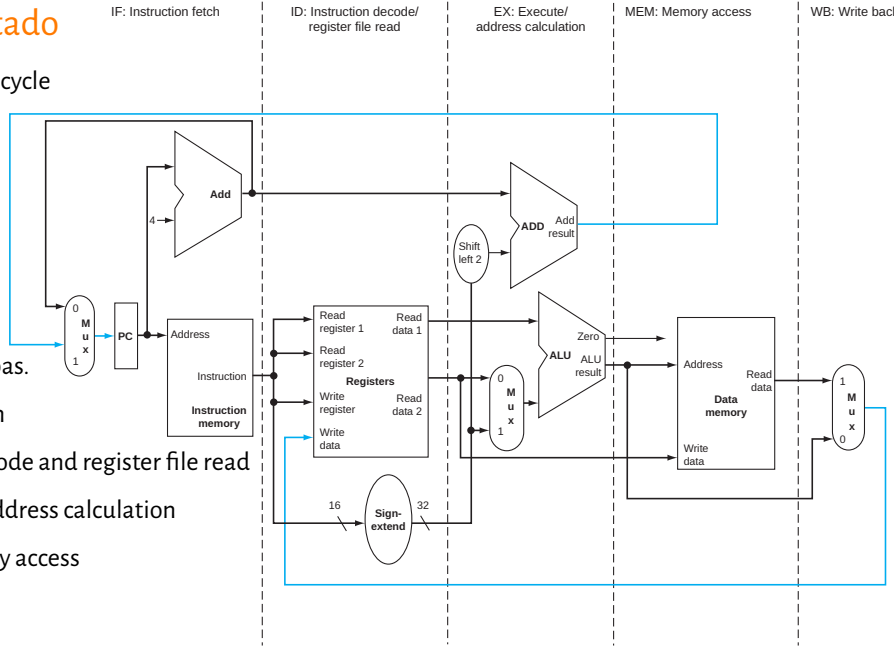
Datapath Segmentado

Sobre el datapath single-cycle identificamos las etapas del *pipeline*.

En un *pipeline* de cinco etapas, 5 instrucciones serán ejecutadas al mismo tiempo.

Debemos separar el datapath en cinco etapas.

1. **IF**: Instruction Fetch
2. **ID**: Instruction Decode and register file read
3. **EX**: EXecution or address calculation
4. **MEM**: Data MEMory access
5. **WB**: Write Back

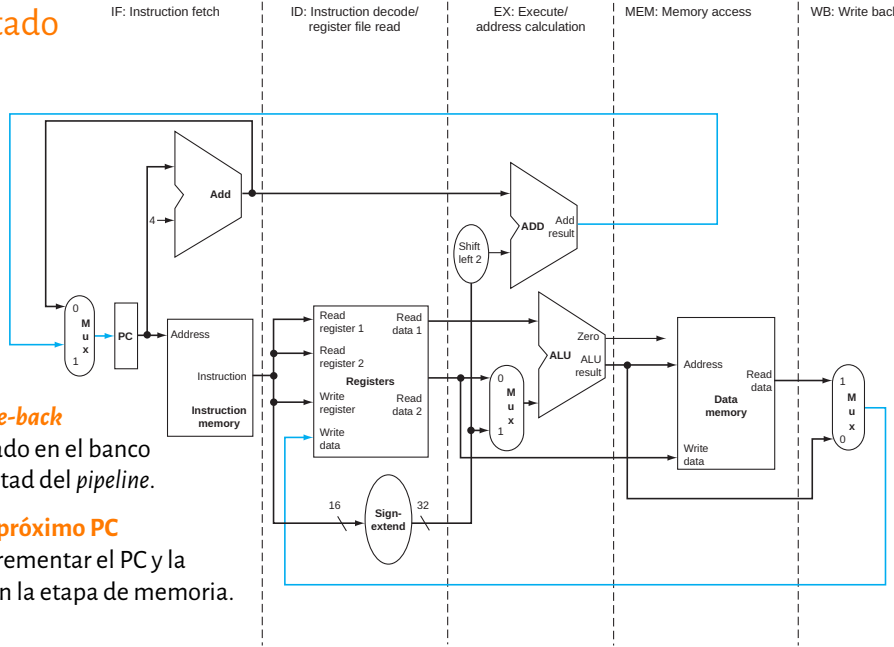


Datapath Segmentado

Los datos en el datapath se mueven de **izquierda a derecha** siguiendo las etapas del *pipeline*.

Con dos excepciones:

- **En la etapa de *write-back***
  Se escribe el resultado en el banco de registros, a la mitad del *pipeline*.
- **En la selección del próximo PC**
  Eligiendo entre incrementar el PC y la dirección de salto en la etapa de memoria.

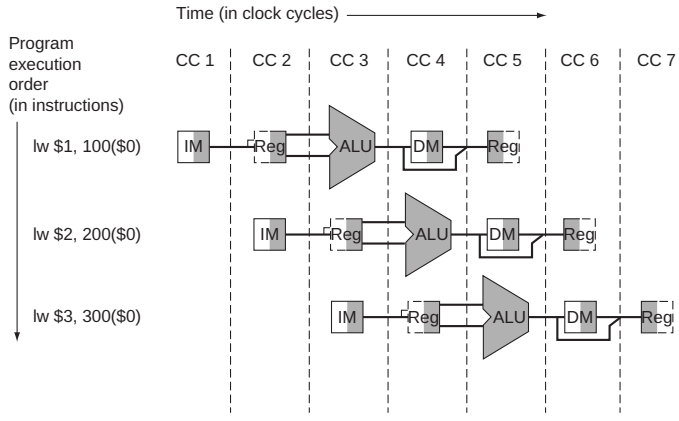


Datapath Segmentado

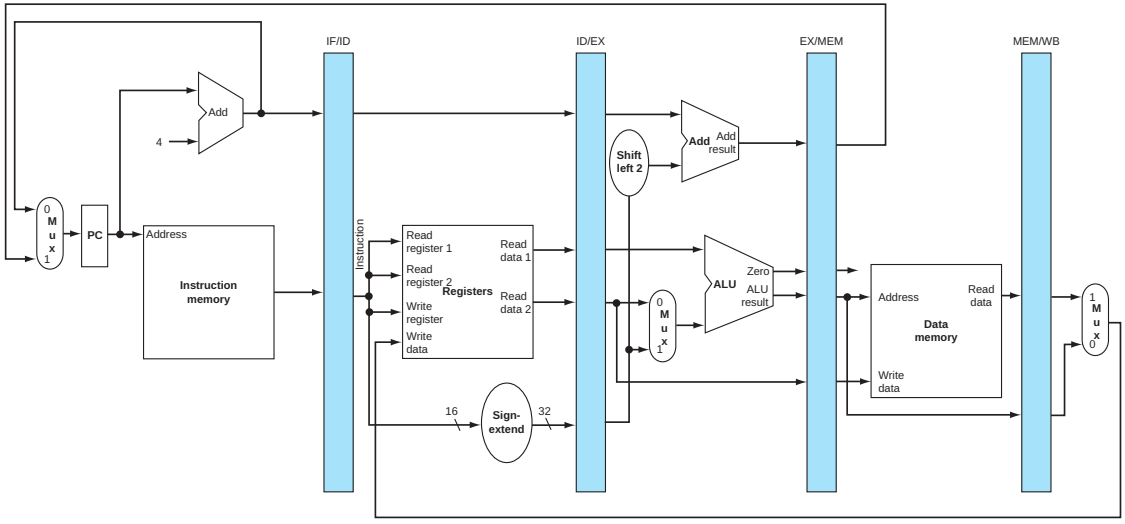
Una forma de ver la ejecución segmentada de instrucciones es pretender que existe **un datapath exclusivo para cada instrucción**. Luego, agrupar esos datapath en el tiempo y así ver sus relaciones.

Para compartir el datapath debemos agregar registros que **mantengan la información** a medida que la instrucción recorre el *pipeline*.

Cada una de las etapas es utilizada por solo una instrucción por ciclo. **Entonces podemos compartirlas en los cuatro ciclos restantes**.



Datapath Segmentado



Datapath Segmentado

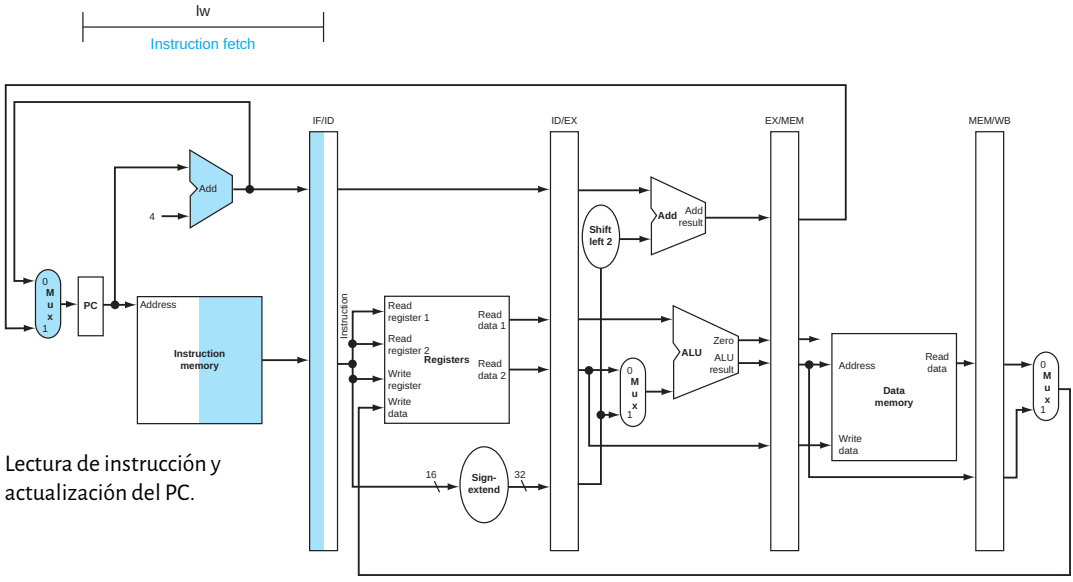
En el diseño se observa:

- Las instrucciones **avanzan** de etapa por cada ciclo de reloj.
- Se identifican un **conjunto de registros** para cada par de etapas.
  Estos se nombran como las dos etapas que los comparten.
- No existe un registro para la **última etapa**, siempre escribe.
- Todas las instrucciones **deben modificar** el estado del procesador, banco de registros, memoria, PC.

Ahora resta entender que parte del datapath se **utiliza** en cada etapa.

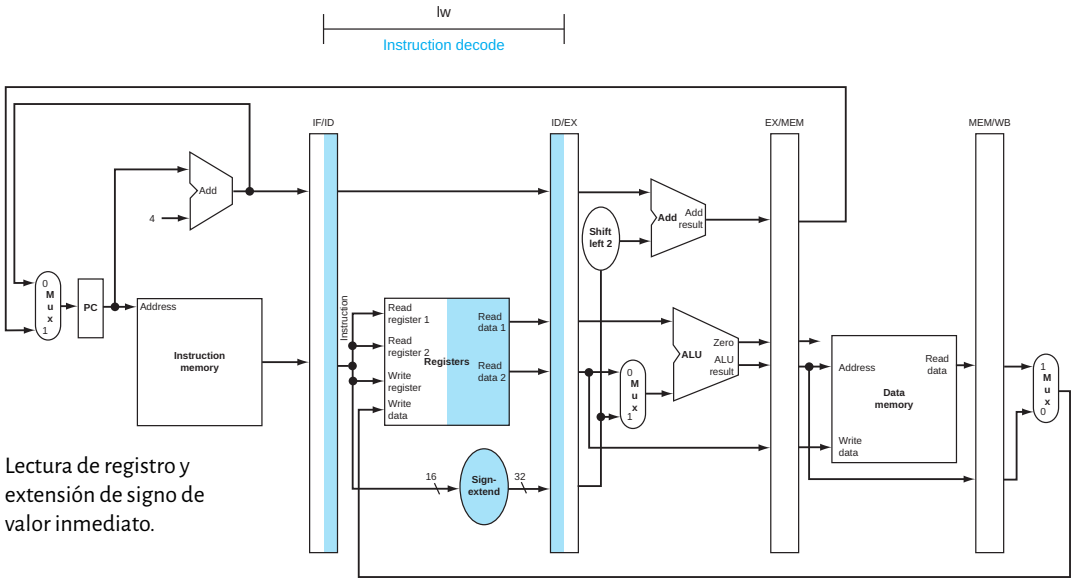
Vamos a estudiar las instrucciones `lw` (*load*) y `sw` (*store*)

Datapath Segmentado - Load (Instruction fetch)



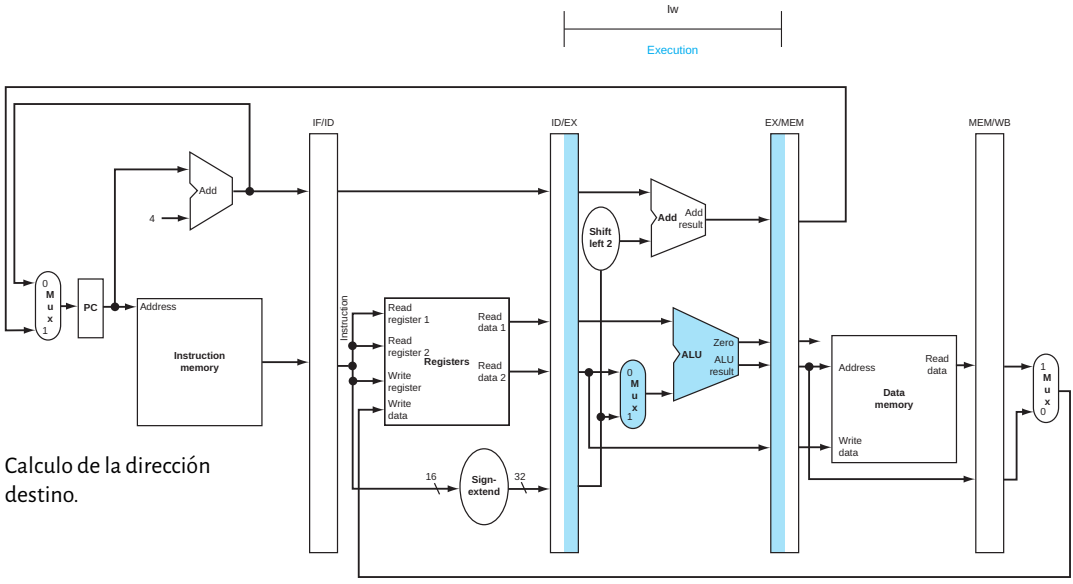
Lectura de instrucción y actualización del PC.

Datapath Segmentado - Load (Instruction decode and register file read)



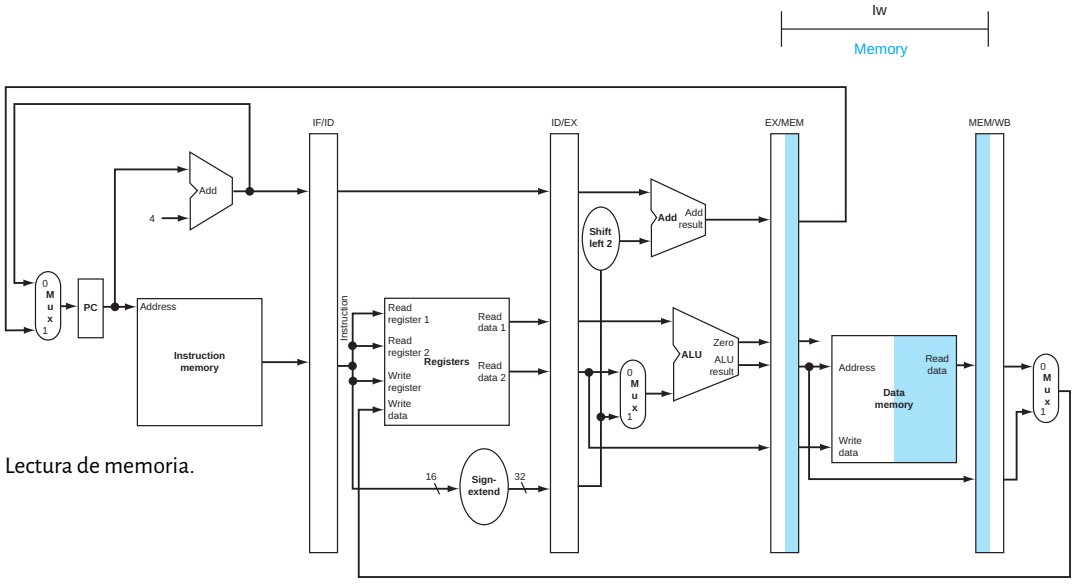
Lectura de registro y extensión de signo de valor inmediato.

Datapath Segmentado (Address calculation)



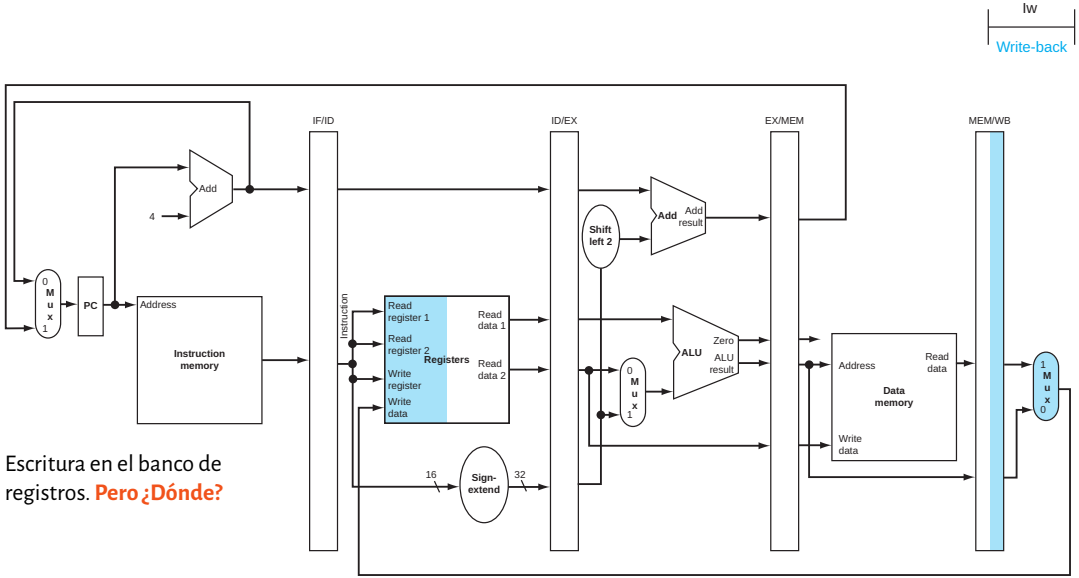
Calculo de la dirección destino.

Datapath Segmentado - Load (Memory access)



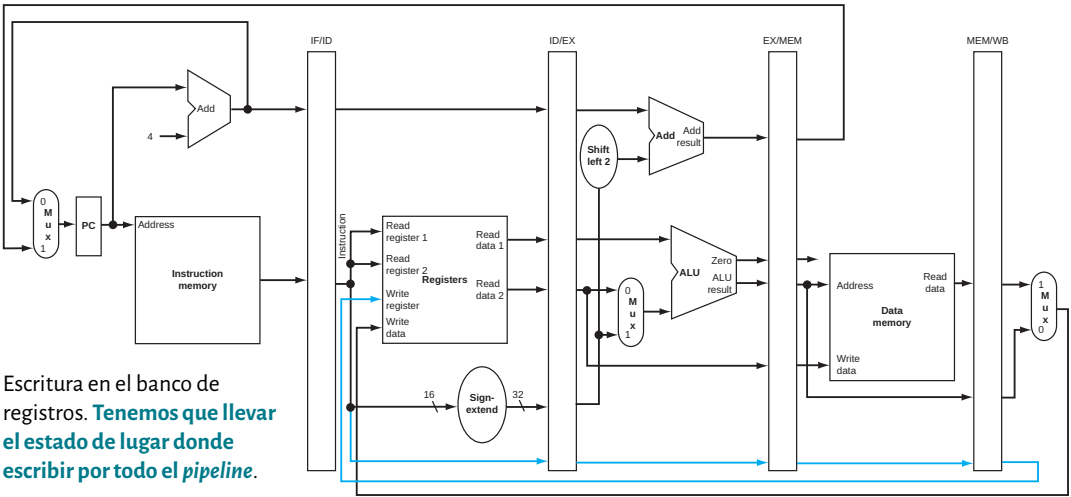
Lectura de memoria.

Datapath Segmentado - Load (Write-back)



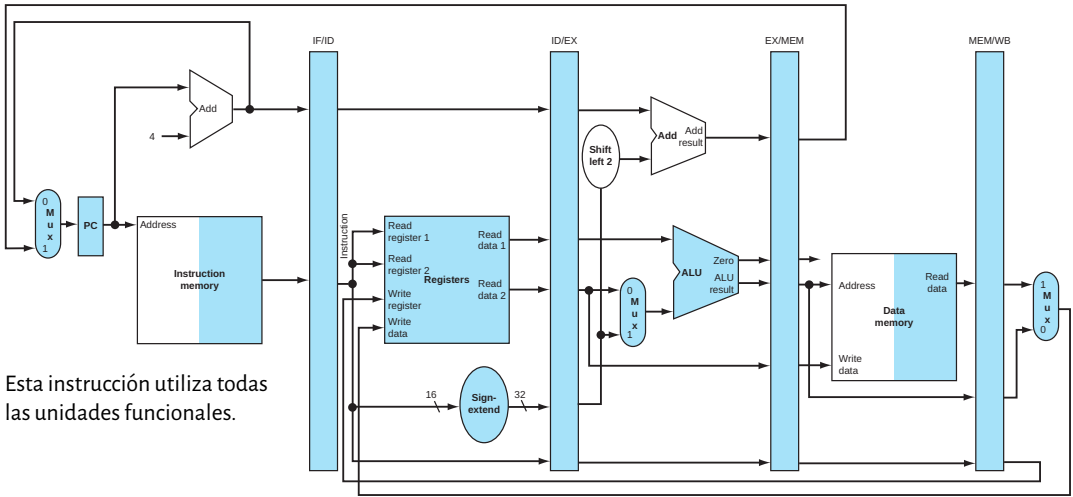
Escritura en el banco de registros. **Pero ¿Dónde?**

Datapath Segmentado - Load (Write-back) → **Arreglado**



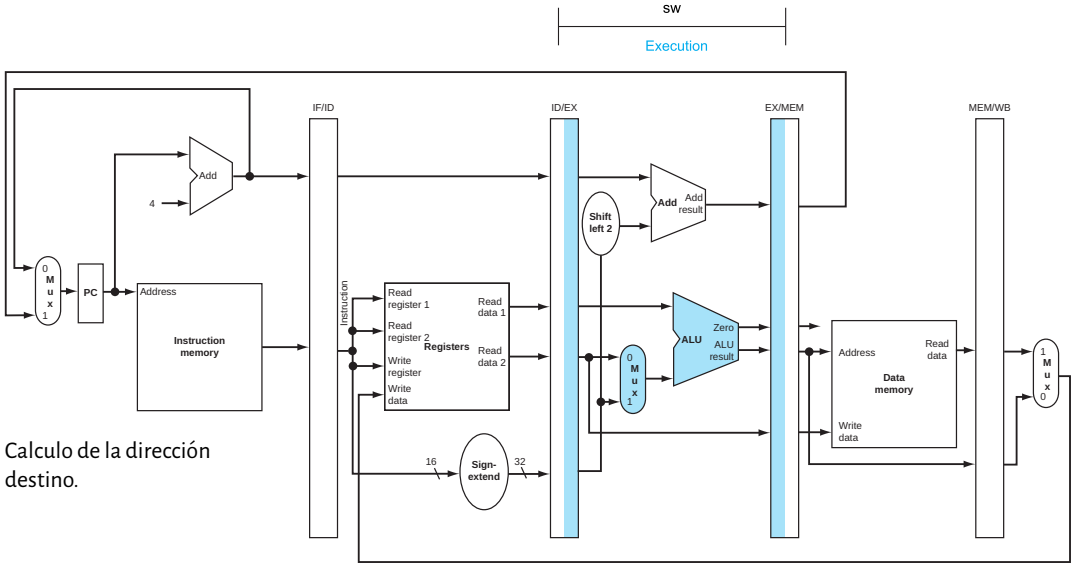
Escritura en el banco de registros. **Tenemos que llevar el estado de lugar donde escribir por todo el *pipeline*.**

Datapath Segmentado - Load



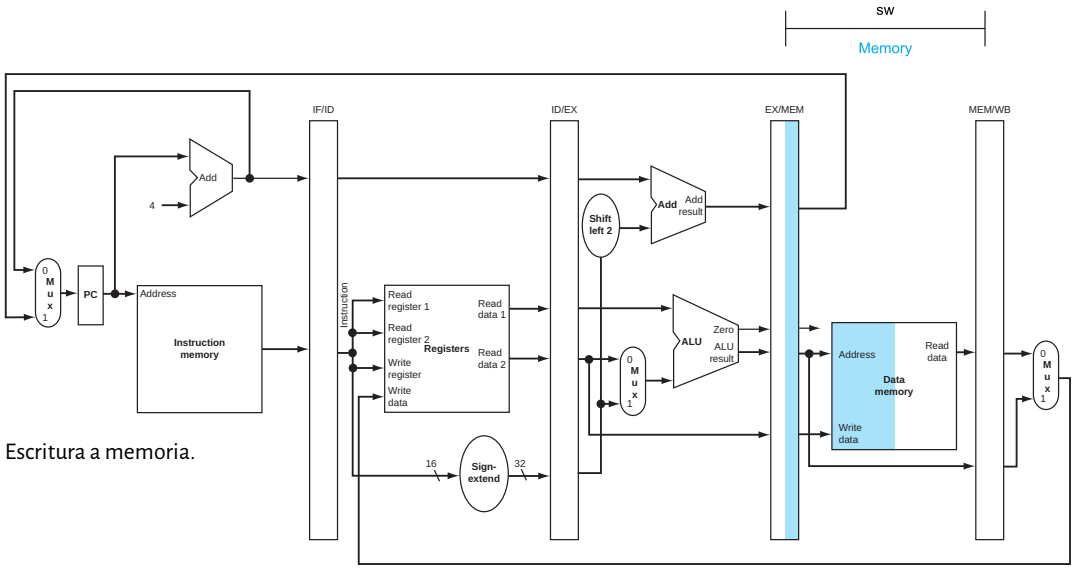
Esta instrucción utiliza todas las unidades funcionales.

Datapath Segmentado - Store (Address calculation)



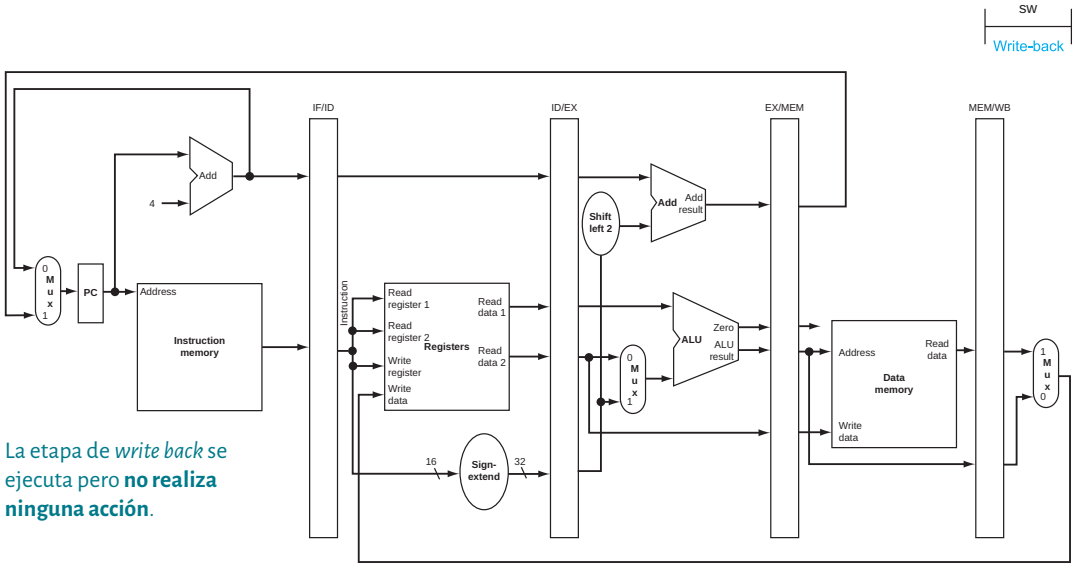
Calculo de la dirección destino.

Datapath Segmentado - Store (Memory access)



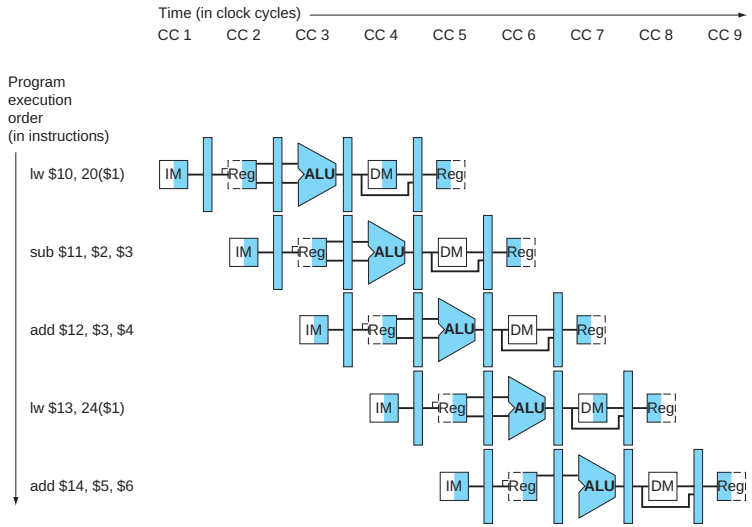
Escritura a memoria.

Datapath Segmentado - Store (Write-back)



La etapa de *write back* se ejecuta pero **no realiza ninguna acción**.

Representación gráfica de un *pipeline*



Supongamos las siguientes instrucciones: `lw $10, 20($1)`

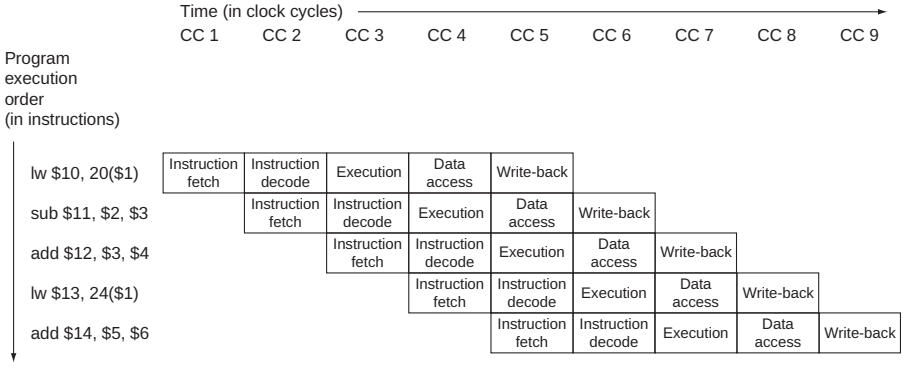
```
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

En el diagrama se puede **identificar que unidad está en uso** en cada ciclo de reloj para cada instrucción en ejecución.

Notar que el banco de registros se utiliza por dos instrucciones simultaneamente.

Representación gráfica de un *pipeline*

Otra forma de representar el *pipeline* es indicando en que unidad se encuentra cada instrucción.

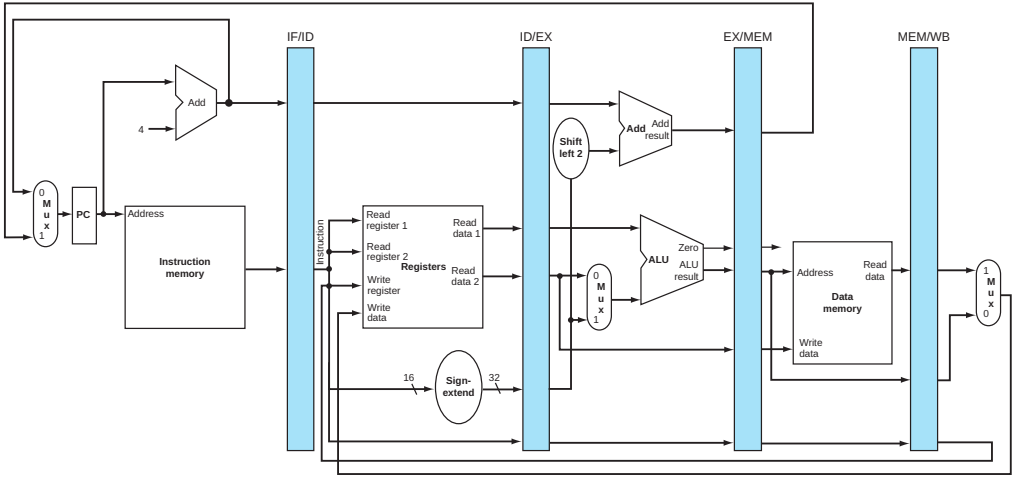


En este caso se pierde que tarea está realizando cada unidad, o incluso si está en uso.

Veamos entonces el estado de todo el *datapath* durante el quinto ciclo.

Datapath Segmentado - Quinto Ciclo

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



Señales de control en un diseño segmentado

Usando las señales de control del diseño *single-cycle* y el funcionamiento de la unidad de control.

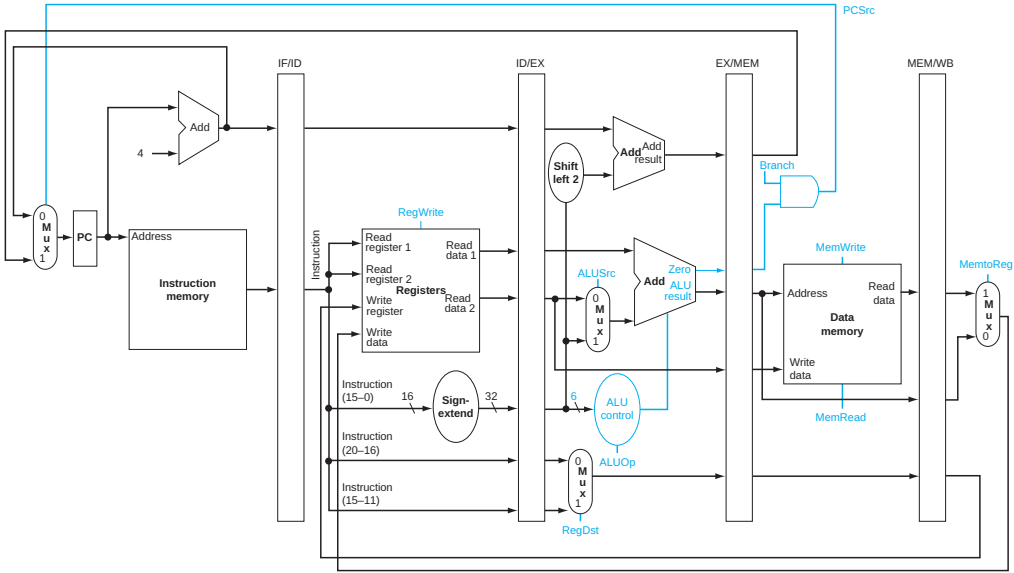
Vamos a asumir que el PC se escribe en todos los ciclos al igual que los registros de *pipeline* (`IFID`, `IDEX`, `EXMEM` y `MEMWB`), por lo tanto **no requieren una señal de control** para su escritura.

Necesitamos especificar las señales de control. Los valores de cada señal dependerán de la etapa del *pipeline*, **ya que las señales de control dependen de cada componente activo en la etapa**.

Vamos a dividir las líneas de control en cinco grupos.

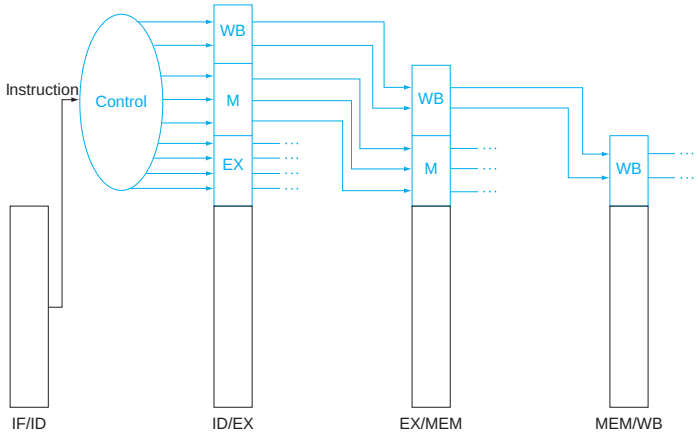
1. *Instruction fetch*: Las señales siempre están activas, no hay nada que hacer.
2. *Instruction decoderegister file read*: Las señales siempre están activas, no hay nada que hacer.
3. *Executionaddress calculation*: Las señales a configurar son `RegDst`, `ALUOp`, y `ALUSrc`.
   Las señales seleccionan el registro resultado, la operación ALU, y una de las entradas de la ALU.
4. *Memory access*: Las líneas de control configuradas en esta etapa son `Branch`, `MemRead` y `MemWrite`.
   Las instrucciones `branch equal`, `load` y `store` establecen estas señales, respectivamente.
5. *Write-back*: Las dos líneas de control son `MemtoReg`, que decide entre enviar el resultado de la ALU o el valor de la memoria al banco de registros, y el registro destino.

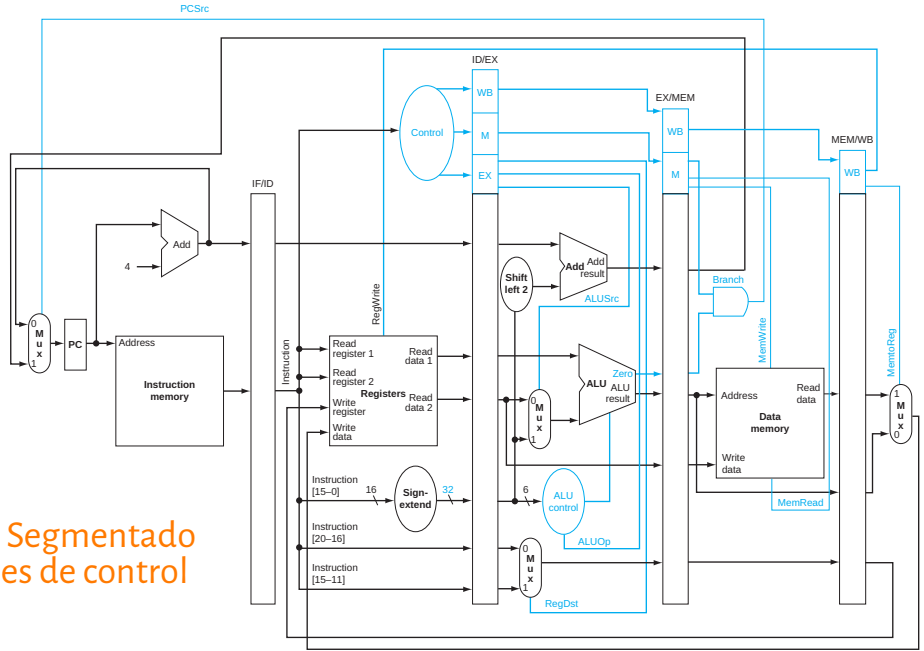
Datapath Segmentado - Indentificación de señales de control



Datapath Segmentado - Con señales de control

La forma más simple de implementar que cada instrucción traslade sus señales de control en el *pipeline*, es extendiendo los registros del *pipeline* con las señales a mantener.





Datapath Segmentado
Con señales de control

Datapath Segmentado - Continuación

Hasta aquí logramos introducir el funcionamiento básico de un diseño segmentado.
Sin embargo resta indagar sobre los *Hazards*.

Los *data hazards* se mitigan usando la **técnica de *forwarding***. Para esto se debe diseñar una *Forwarding unit*, que identifica cuando existe una dependencia y decide cuando reenviar un dato antes de terminar la instrucción que lo debería calcular.

Los *control hazards* se resuelven implementando **la *hazard detection unit***. Esta determina cuando una instrucción no puede ser ejecutada en el próximo cico y genera un *stall* en el *pipeline*.

Adicionalmente se utilizan **técnicas de ejecución especulativa**, gracias a predictores de saltos dinámicos. Estos conocen el contexto de ejecución y su historia, para determinar si un salto debe o no ser tomado.

Bibliografía

- **“Computer Organization and Design: The Hardware/Software Interface”**, Fifth Edition
  David A. Patterson, John L. Hennessy - Morgan Kaufmann - 2014
  - Chapter 4 - The Processor. Pag. 242-303

- **“MIPS Reference Data Card (“Green Card”)”**
  From Patterson and Hennessy, Computer Organization and Design, 4th ed.
  Elsevier, Inc.

¡Gracias!