Capítulo 3: MIPS

SEGMENTACIÓN EN LA EJECUCIÓN DE INSTRUCCIONES

Parte del material fue desarrollado en la Escuela Politécnica Superior de la Universidad Autónoma de Madrid.

Arqui1-UNICEN

Introducción

- Para el estudio de procesadores segmentados se parte de un sencillo procesador RISC denominado MIPS (Microprocessor without Interlocked Pipeline Stages)
  - Procesador de 32 bits (datos, memoria)
  - 32 registros de propósito general
  - Memoria de datos y código separadas

Características de las arquitecturas RISC

- Juego de instrucciones reducido.
- Acceso a memoria limitado a instrucciones de carga/almacenamiento.
- Muchos registros de propósito general.
- Pocos modos de direccionamiento (inmediato, directo, indexado).
- Formato de instrucción homogéneo (misma longitud y distribución de campos).
- Todas las instrucciones se ejecutan en un ciclo de reloj.

El juego de instrucciones

- Los diseñadores de computadoras tienen como objetivo encontrar un juego de instrucciones tal que
  - Sea sencillo construir el **hardware** que materialice ese juego de instrucciones y
  - El **compilador** sea sencillo y eficiente,
  - Maximizando el **rendimiento** de la computadora y
  - Minimizando su consumo de **energía**.

Procesadores MIPS

- Los procesadores MIPS tienen un juego de instrucciones elegante desarrollado desde la década de 1980.
- Otro ejemplo de juego de instrucciones es el de los procesadores ARM.
  - Similar a las instrucciones MIPS.
  - Vendió más de 3.000 millones de chips para aplicaciones embebidas en 2008.
- Un juego de instrucciones diferente es el Intel x86.
  - Se vendieron 330 millones de PCs Intel en 2008.

Procesadores MIPS

- En 1981, un equipo liderado por John L. Hennessy en la Univ. de Stanford comenzó a trabajar en el primer procesador MIPS.
- A principios de los '90 MIPS Technologies comenzó a otorgar licencias de sus diseños a terceros, de ahí procedían más de la mitad de los ingresos de MIPS.
- Los procesadores MIPS se utilizaron por ejemplo en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2, etc.
- Debido a que su conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades a menudo se basan en la arquitectura MIPS.

R10000 (Toshiba TC86R10000-200,1996)

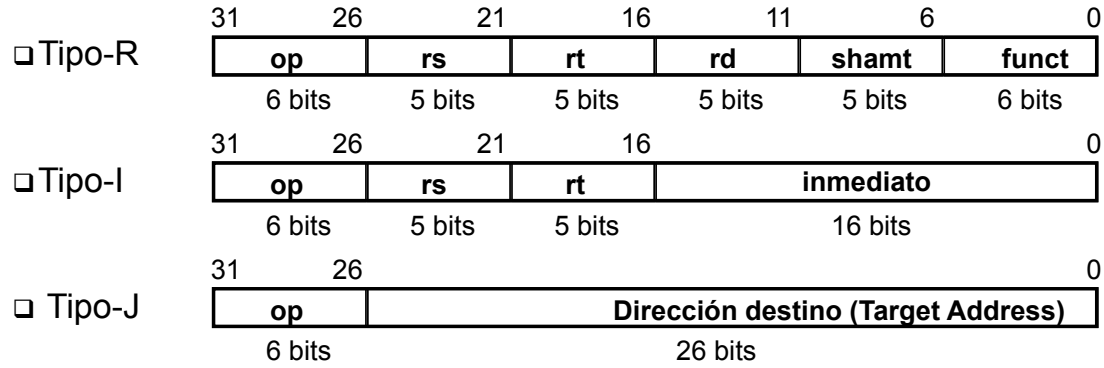
Emotion Engine (Sony, 2000) MIPS-IV (R4000) de 128 Bits

Procesador MIPS

- 32 registros de uso general: $0 .. $31 (excepto $0 siempre igual a 0).
- 2^{30} palabras de memoria (32 bits c/u).
- Instrucciones de 1 palabra (32 bits) de longitud.
- Acceso a memoria limitado a 2 tipos de instrucciones:
  - LOAD (carga una palabra de memoria en registro)
  - STORE (almacena un registro en memoria)

El repertorio de instrucciones: características y tipos

- Conjunto de instrucciones MIPS (32 bits). Tres formatos de instrucciones:

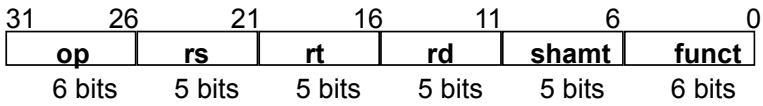


- Campos de cada instrucción:
  - **op: Código de operación de la instrucción**
  - **rs, rt, rd: identificador de los registros fuente y destino**
  - **shamt: desplazamiento deseado**
  - **funct: selección de la variante de función asociada**
  - **inmediato: dato inmediato en 16 bits**
  - **Dirección destino**

Convención para el banco de registros MIPS

Nombre	Número	Uso	Se preserva en call?
$zero	0	Constante cero	-
$at	1	Reservado assembler	No
$v0-$v1	2-3	Resultados de expresiones	No
$a0-$a3	4-7	Argumentos	No
$t0-$t7	8-15	Temporarios	No
$s0-$s7	16-23	*Saved*	Si
$t8-$t9	24-25	Temporarios	No
$k0-$k1	26-27	Reservados SO	No
$gp	28	Puntero global	Si
$sp	29	Puntero pila	Si
$fp	30	Puntero frame	Si
$ra	31	Dir retorno sub-rutina	Si

El repertorio de instrucciones: Tipo-R



Ejemplos: ADD y SUB

add rd, rs, rt # rd=rs+rt

sub rd, rs, rt # rd=rs-rt

Ejemplos: Código C

f = (g + h) – (i + j);

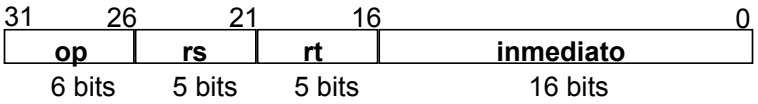
Si las variables f, g, h, i, j están en los registros $s0 a $s4 el compilador puede generar:

add $t0, $s1, $s2

add $t1, $s3, $s4

sub $s0, $t0, $t1

El repertorio de instrucciones: Tipo-I



ADD inmediato

addi rt, rs, inm # rt=rs+inm

LOAD y STORE word

lw rt, rs, inm # rt=mem[rs+inm]

sw rt, rs, inm # mem[rs+inm]=rt

SALTOS

beq rs, rt, inm # si rs=rt,

entonces PC=PC+4+inm*4

el lenguaje ensamblador admite etiquetas y calcula inm

El repertorio de instrucciones: Ejemplo

Ejemplos: Código C

if (i == j) f = (g + h); else f = g – h;

Si las variables f, g, h, i, j están en los registros $s0 a $s4 el compilador puede generar:

```
        bne $s3, $s4, CasoElse
        add $s0, $s1, $s2          # si i==j
        j Exit                     # salto incondicional
CasoElse: sub $s0, $s1, $s2
Exit:
```


El repertorio de instrucciones: Ejemplo

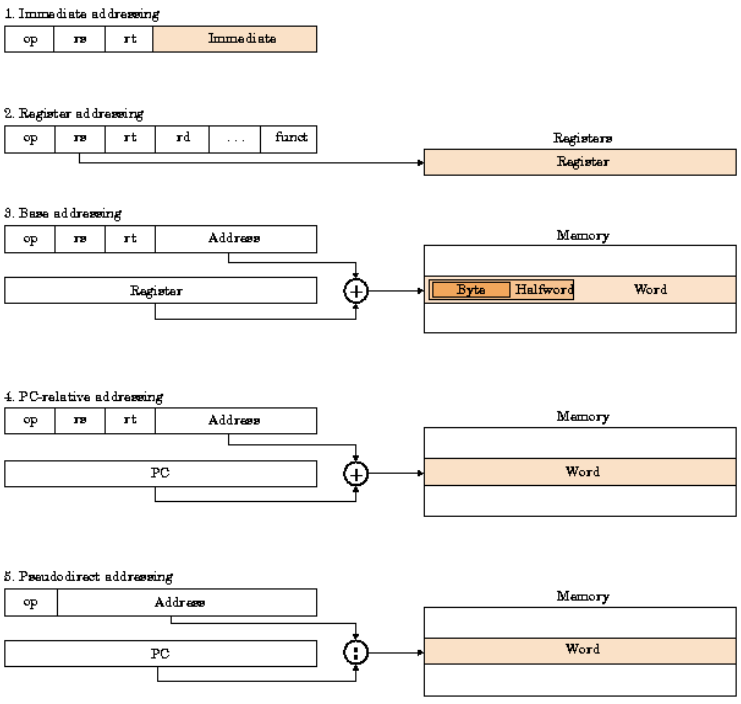
Ejemplos: Código C

while (save[i] == k) i+=1;

Si las variables *i* y *k* están en los registros $s3 y $s5 y el registro base del arreglo *save* esta en $s6, el compilador puede generar:

```
Loop: sll $t1, $s3, 2       # sll con 2 equivale a multip x4
      add $t1, $t1, $s6     # calcula dir de save[i]
      lw $t0, 0($t1)        # carga save[i]
      bne $t0, $s5, Exit
      addi $s3, $s3, 1      # i+=1
      j Loop                # salto incondicional
Exit:
```


Modos de direccionamiento MIPS



1. El operando es una cte dentro de la instrucción:

   lui $s0, 61

2. El operando es un registro: ejemplos 1, 3 y 4.
3. La dir del operando es la suma de un reg y una cte en la instrución:

   lw $t0, 8($t1)

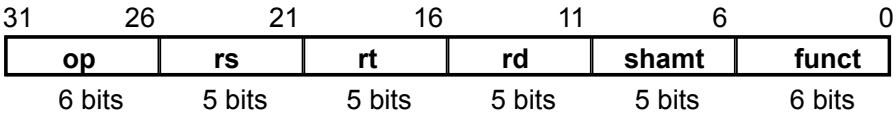
4. La dir de salto es la suma del PC mas una cte. en la instr.:

   beq $s0, $s1, L1

5. La dir de salto es la concat. de los 4 bits más altos del PC con la cte de 26 bits en la instrucción (más los dos LSB que son cero):

   jal printf

Ejemplo ejecución RTL de una instrucción



Instrucción add rd, rs, rt

Descripción de la ejecución (RTL)

IR ←Mem[PC]	Carga de la instrucción desde memoria
R[rd] ← R[rs] + R[rt]	Realiza la operación (SUMAR)
PC ← PC + 4	Calcula la dirección de la siguiente instrucción

Descripción RTL del procesador MIPS

Descripción RTL (Register Transfer Level) de las instrucciones

a. Fase inicial: carga desde Memoria (Fetch)

IR <= MEM[PC] ; IR = op & rs & rt & rd & shamt & funct

; IR = op & rs & rt & Imm16

; IR = op & Inm26

b. Transferencia entre Registros (ejemplos: Instrucc y transf entre registros)

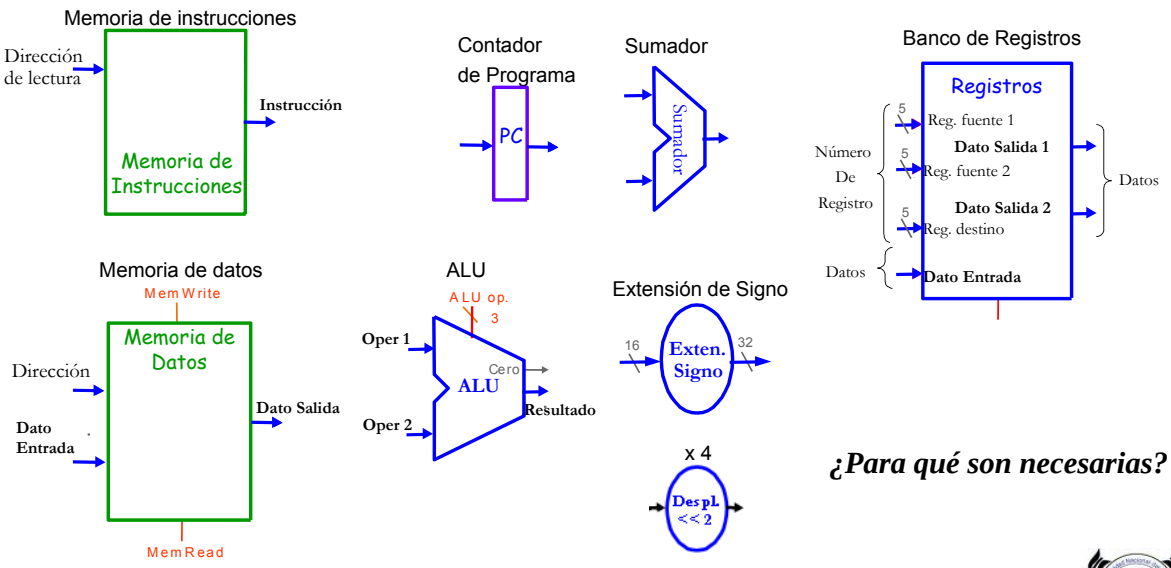
ADD	**R[rd] <= R[rs] + R[rt];**	**PC <= PC + 4**
ADDI	**R[rt] <= R[rs] + Ext_signo(Inm);**	**PC <= PC + 4**
LOAD	**R[rt] <= MEM[R[rs] + Ext_signo(Inm)];**	**PC <= PC + 4**
STORE	**MEM[R[rs] + Ext_signo(Inm)] <= R[rt];**	**PC <= PC + 4**
BEQ	**if (R[rs] == R[rt]) then**	**PC <= PC + 4 + (Ext_signo(Inm) & 00)**
else		**PC <= PC + 4**

Generalidades para el diseño de un procesador

1. Analizar el conjunto de instrucciones => requisitos para la ruta de datos (*datapath*).
   - El significado de cada instrucción viene dado por su funcionamiento a nivel de transferencia de registros (RTL).
   - El *datapath* debe incluir elementos de almacenamiento para los registros accesibles en el modelo de programación del procesador.
   - El *datapath* debe soportar todas las transferencias entre registros definidas en el conjunto de instrucciones.
2. Selección de los componentes y de la metodología de reloj.
3. Implementación del *datapath* cumpliendo los requisitos.
4. Análisis de cada instrucción para determinar el mecanismo de control que efectúe la transferencia entre registros.
5. Implementación de la lógica de control.

Diseño del procesador: elementos básicos

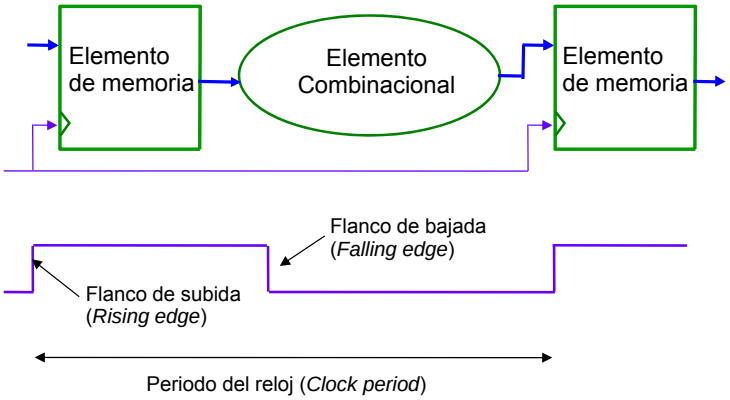
- Unidades funcionales necesarias para las instrucciones



¿Para qué son necesarias?

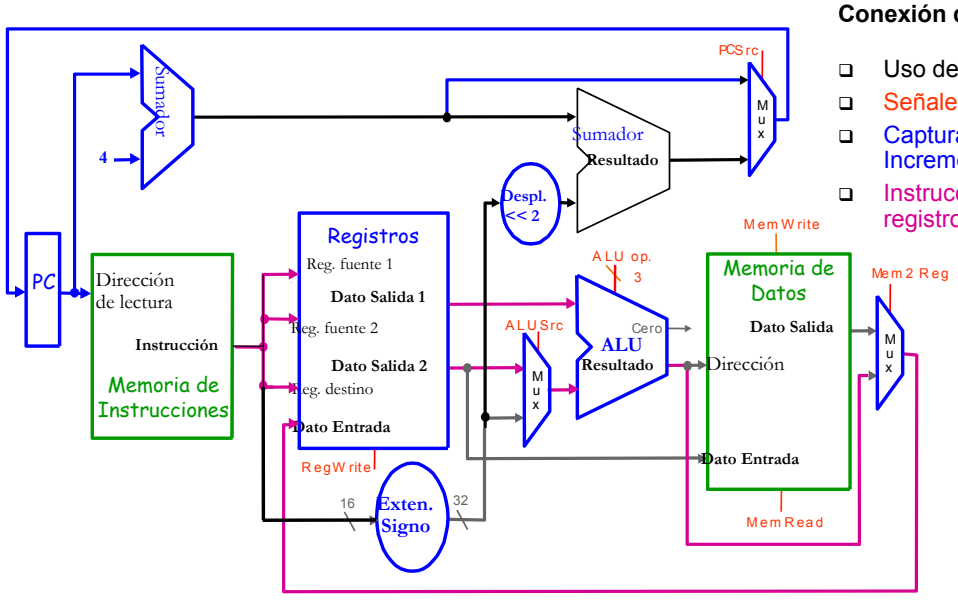
Diseño del procesador: sincronización

- La metodología de *sincronización* indica cuándo pueden leerse y escribirse las diferentes señales.
- En este procesador (MIPS) los ciclos de reloj comienzan en flanco de subida



Conexiones en la ruta de datos (*Datapath*)

Operaciones entre registros, Tipo-R (ADD, SUB, OR, AND, etc)

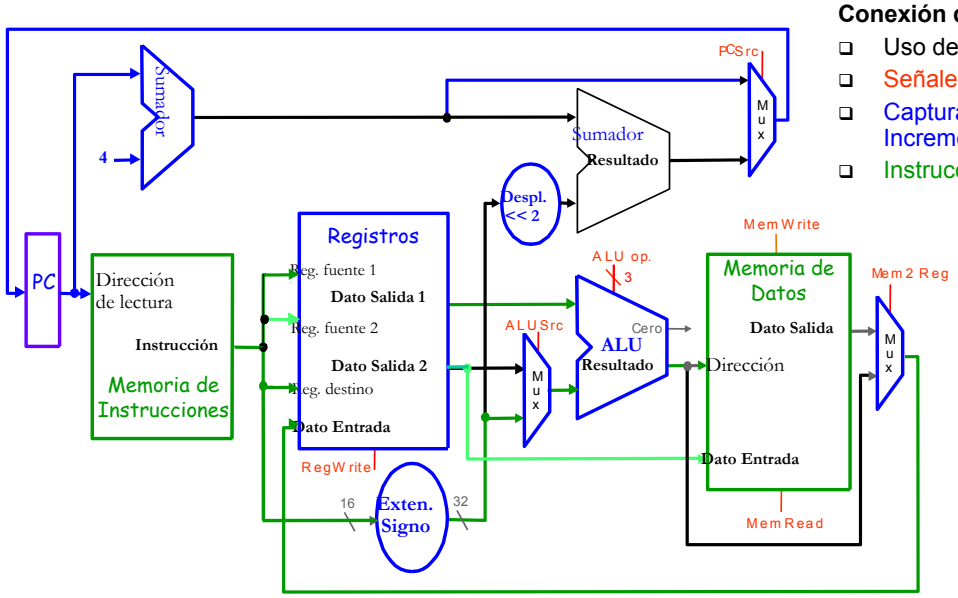


Conexión de los elementos:

- Uso de multiplexores.
- Señales de control.
- Captura de la siguiente instruc. Incremento de PC.
- Instrucciones de la ALU entre registros.

Conexiones en la ruta de datos (*Datapath*)

Operaciones de carga y almacenamiento, Tipo-I (Load / Store)

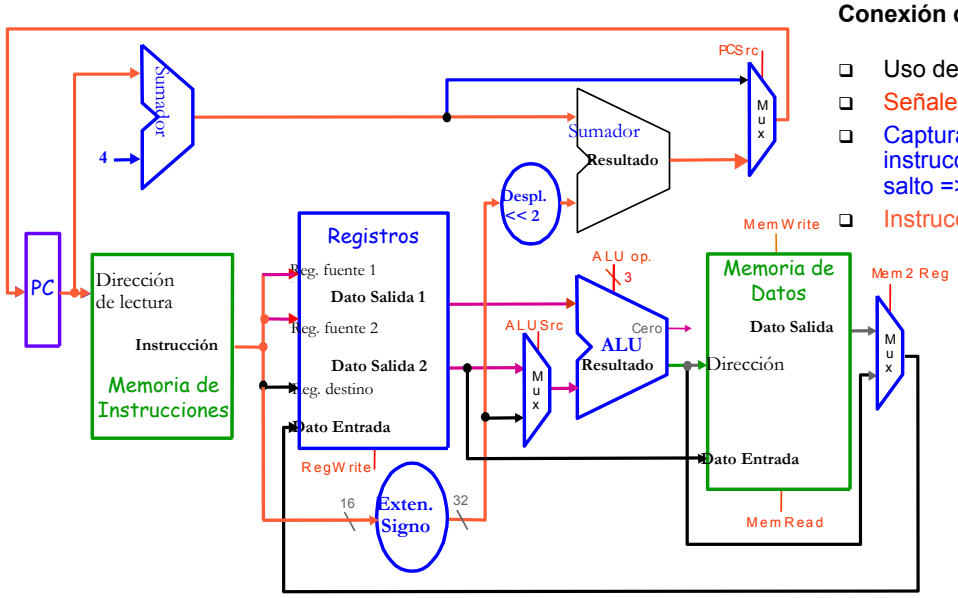


Conexión de los elementos:

- Uso de multiplexores.
- Señales de control.
- Captura de la siguiente instruc. Incremento de PC.
- Instrucciones LOAD/STORE.

Conexiones en la ruta de datos (*Datapath*)

Operaciones de salto condicional (Beq)

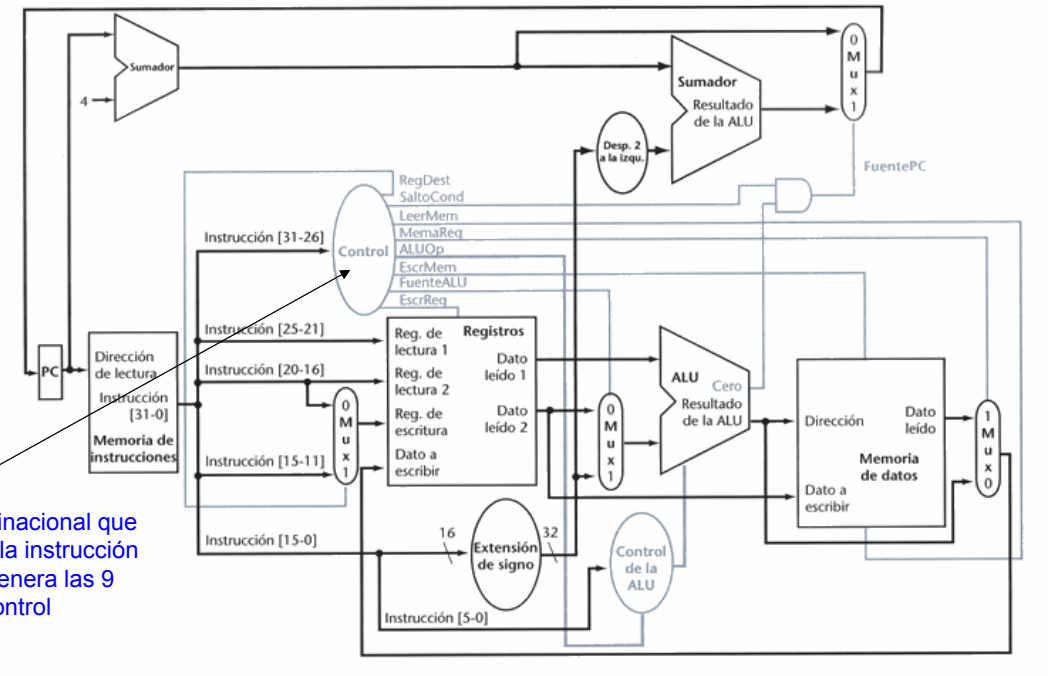


Conexión de los elementos:

- Uso de multiplexores.
- Señales de control.
- Captura de la siguiente instrucción o del destino de salto =>Incremento de PC.
- Instrucciones de Salto.

Ruta de Datos con Control Uniciclo

Instrucción	RegDest	FuenteALU	MemaReg	EscrReg	LeerMem	EscrMen	SaltoCond	ALUOp1	ALUOp0
Reg a Reg	1	0	0	1	0	0	0	1	0
LOAD	0	1	1	1	1	0	0	0	0
STORE	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1



Control:

Bloque combinacional que con 6 bits de la instrucción (bits 31:26) genera las 9 señales de control

¿Cómo agregar Jumps?

Jump	2	address
	31:26	25:0

Jump usa direccionamiento pseudo-directo

El nuevo valor del PC se forma concatenando

- Los 4 bits más altos del PC
- El operando de 26 bits
- 00

¿Se necesita alguna nueva señal de control?

Ruta de Datos agregando Jumps

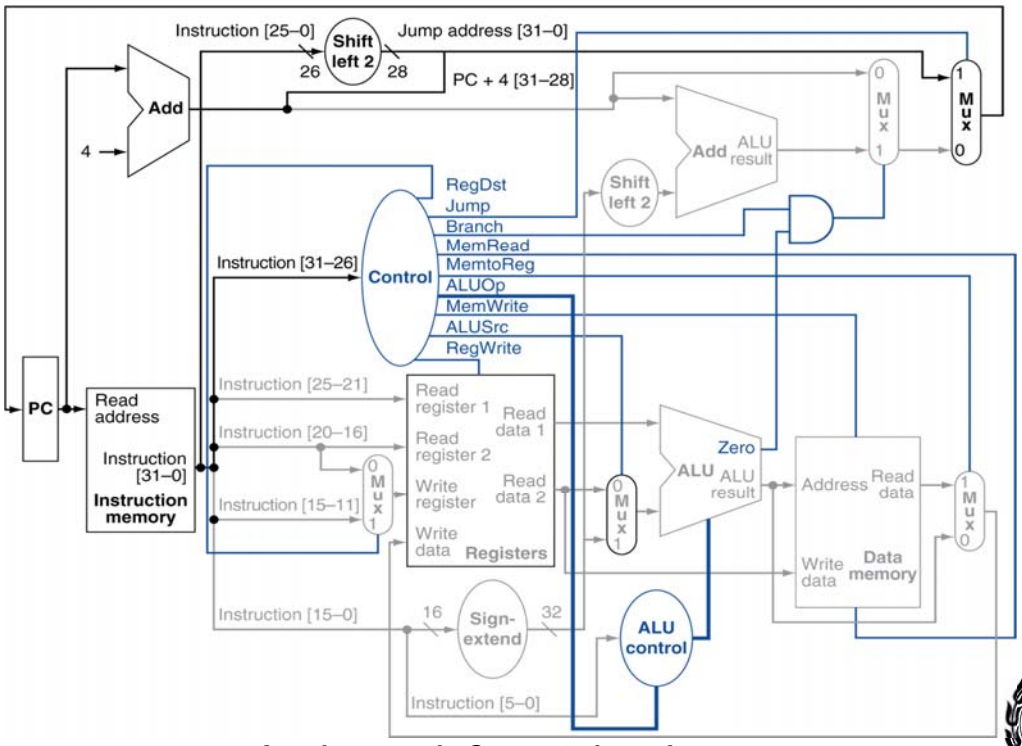
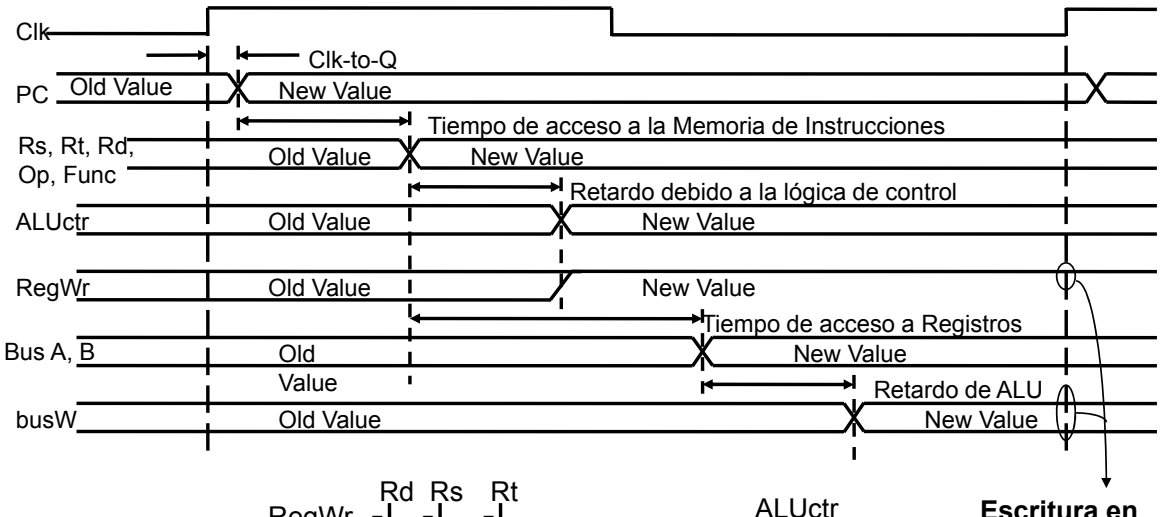
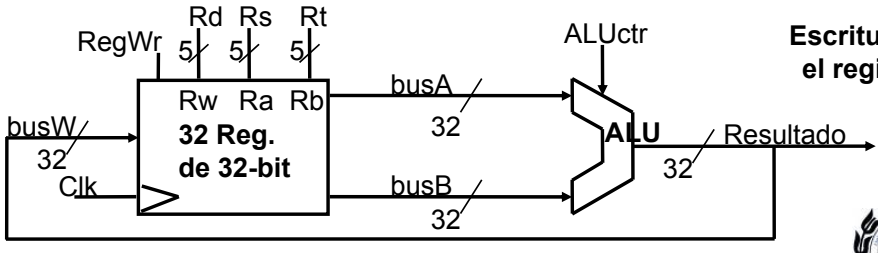
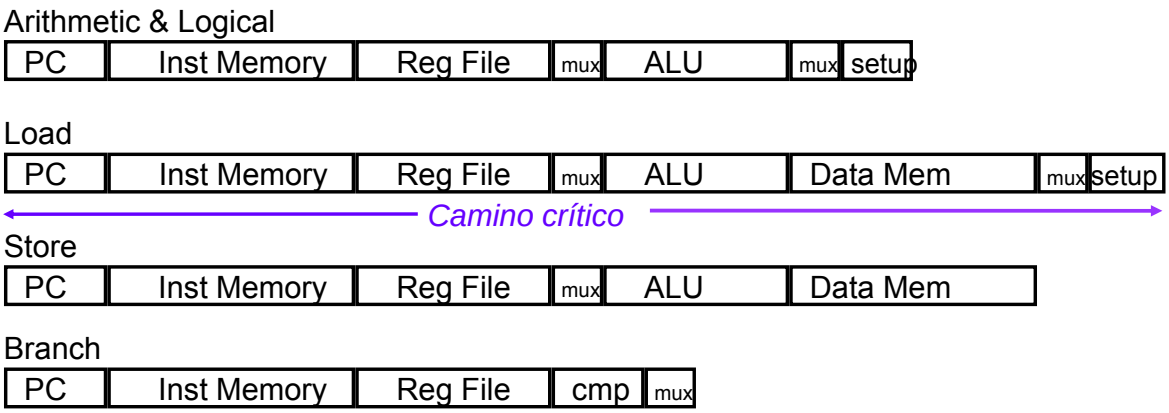


Diagrama de tiempo de operación Reg-Reg





Desventajas del diseño uniciclo (CPI=1)



- Tiempo de ciclo muy largo (el peor de todos).
- Casi todas las instrucciones utilizan, sin necesidad, tanto tiempo como la instrucción más lenta.
- Se viola el principio de diseño: Hacer que el caso común sea rápido

Desventaja en la ejecución uniciclo: ejemplo

- Tiempos hipotéticos para ejecutar cada instrucción

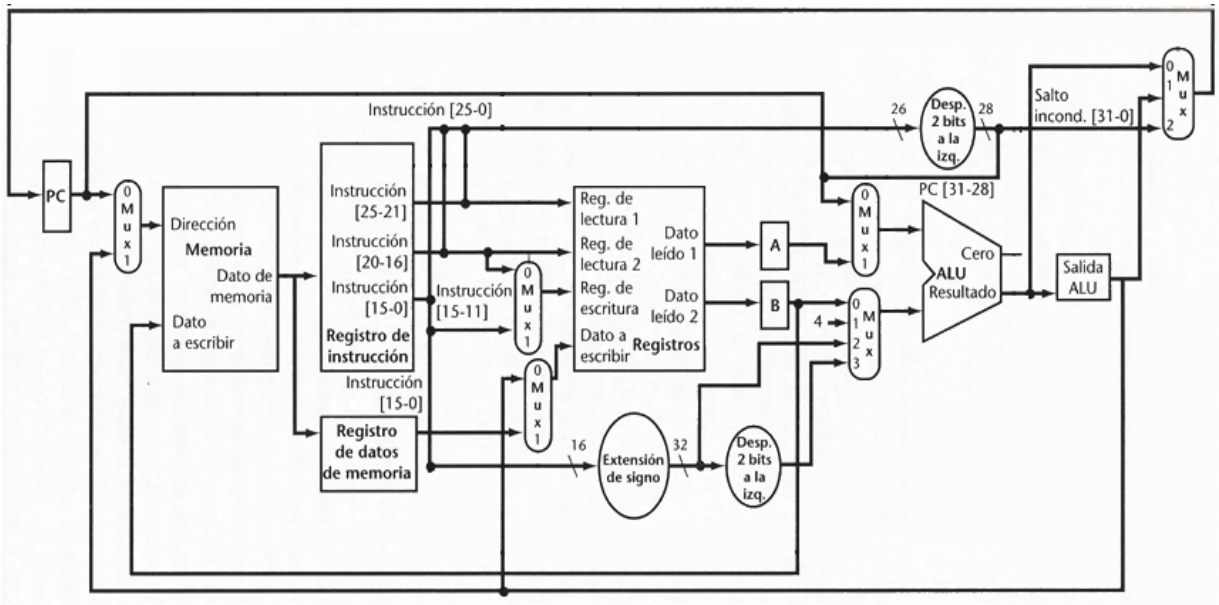
Clase de Instrucción	Mem instr.	Lect de Reg	Operac. ALU	Mem de datos	Escrit en Reg	Total
Formato R	2	1	2	0	1	6 ns
Load (LW)	2	1	2	2	1	8 ns
Store (SW)	2	1	2	2		7 ns
Salto (BEQ)	2	1	2			5 ns
Jump	2					2 ns

- Sea un programa que utiliza 24% de cargas, 12% de almacenamientos, 44% de operaciones entre registros en la ALU, 18% de saltos condicionales y 2 % de saltos incondicionales
- Si fuese simple (NO LO ES) tener un reloj variable
  - ¿Cuál es tiempo medio de ciclo?
  - ¿Cuál sería la aceleración respecto de una ejecución uniciclo?

Ejecución multiciclo

- Las instrucciones pueden tardar diferente número de ciclos.
- Un “datapath” con ligeras modificaciones:
  - Dividir las instrucciones en pasos, cada paso tarda un ciclo.
  - Balancear la cantidad de trabajo a realizar en un paso.
  - En cada ciclo sólo se utiliza una unidad funcional (se reduce el número de U.F.)
- Al final del ciclo
  - Almacenar los valores para su uso en posteriores pasos.
  - Añadir registros internos adicionales.

Cambios para una aproximación multiciclo



Ejecución en cinco pasos (5 ciclos)

1. Carga la instrucción (todas igual)

La ALU actualiza el contador de programa:

- Todas: IR = Memoria[PC];
  PC = PC + 4;

2. Decodificación y lectura de operandos (todas igual)

Todavía se sigue el mismo cauce para cualquier instrucción porque se están decodificando.

- Lee registros rs y rt por si son necesarios
- Calcula la dirección de salto en la ALU por si fuera necesaria (branch)

- Todas: A = Reg[IR[25-21];
  B = Reg[IR[20-16];
  SalidaALU = PC + (extension_signo(IR[15-0]<<2);

Ejecución en cinco pasos (5 ciclos)

3. Ejecución. Calculo de la dirección de memoria. Finalización del salto.

La ALU, dependiendo del tipo de instrucción, realiza una operación u otra.

- Referencia a memoria: SalidaALU = A + extension_signo(IR[15-0]);
- Operación entre registros: SalidaALU = A op B;
- Saltos: if (A==B) PC = SalidaALU;

4. Acceso a memoria o final instrucción tipo R.

Acceso a memoria en Load's y Store's.

- Load: MDR = Memoria[SalidaALU];
- Store: Memoria[SalidaALU] = B;

Escritura del registro destino en instrucciones entre registros.

- Operación entre registros: Reg[IR[15-11]] = SalidaALU;

(La escritura tiene lugar en el flanco al final del ciclo)

Ejecución en cinco pasos (5 ciclos)

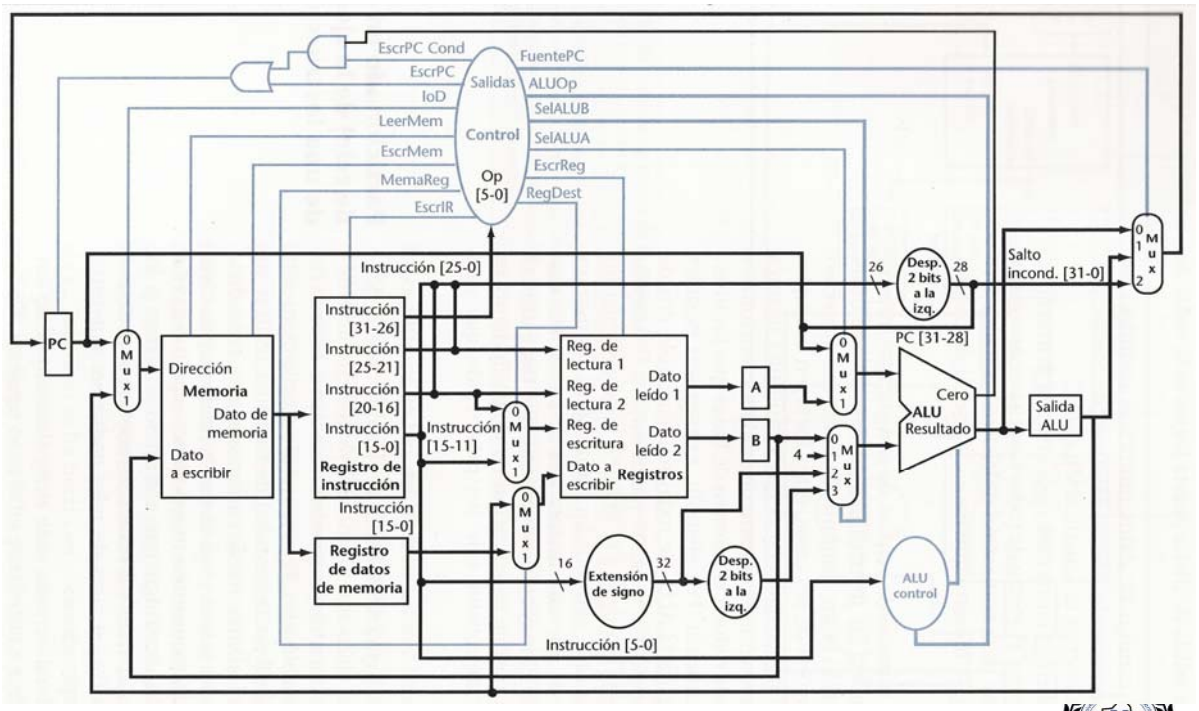
5. Escritura del valor leído de memoria en el registro destino (Write-back).
   - Load: Reg[IR[20-16]]= MDR;

"UNA INSTRUCCIÓN TARDA DE 3 A 5 CICLOS"

Control multiciclo: resumen de etapas

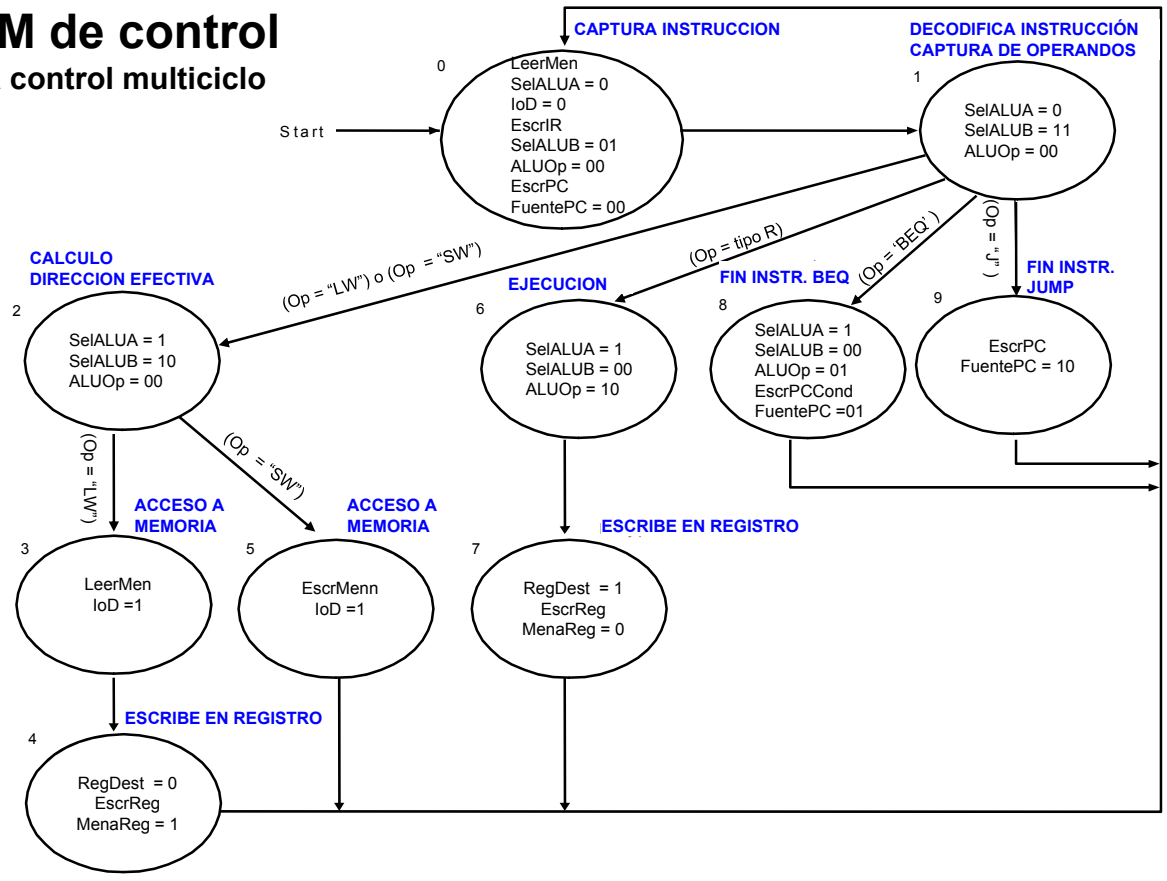
Nombre de la etapa	Tipo R	acceso a memoria	Saltos condicionales	Saltos incondic(jump)
Carga Instrucciones	IR = Memoria[PC] PC = PC + 4			
Decodific de instrucc / carga Reg	A = Reg [IR[25-21]] B = Reg [IR[20-16]] SalidaALU = PC + (extension-signo (IR[15-0] << 2)			
Ejecución, cálculo de direcc y fin de saltos condicionales	SalidaAlu = A op B	SalidaALU = A + (extension-signo (IR[15-0])	si (A = B) entonces PC = SalidaALU	PC = PC[31-28] || (IR[25-0] << 2)
Acc. a MEM y Fin de instrucc tipo R	Reg [IR[15-11]] = SalidaALU	*Load*: MDR = Memoria[SalidaALU] ó *Store:* Memoria[SalidaALU] = B		
Fin lectura MEM		*Load*: Reg [IR[20-16]] = MDR		

Control Multiciclo



FSM de control

Para control multiciclo



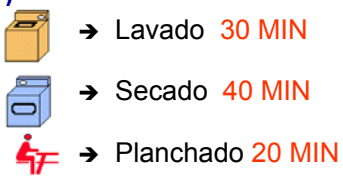
Segmentación: Perspectiva General

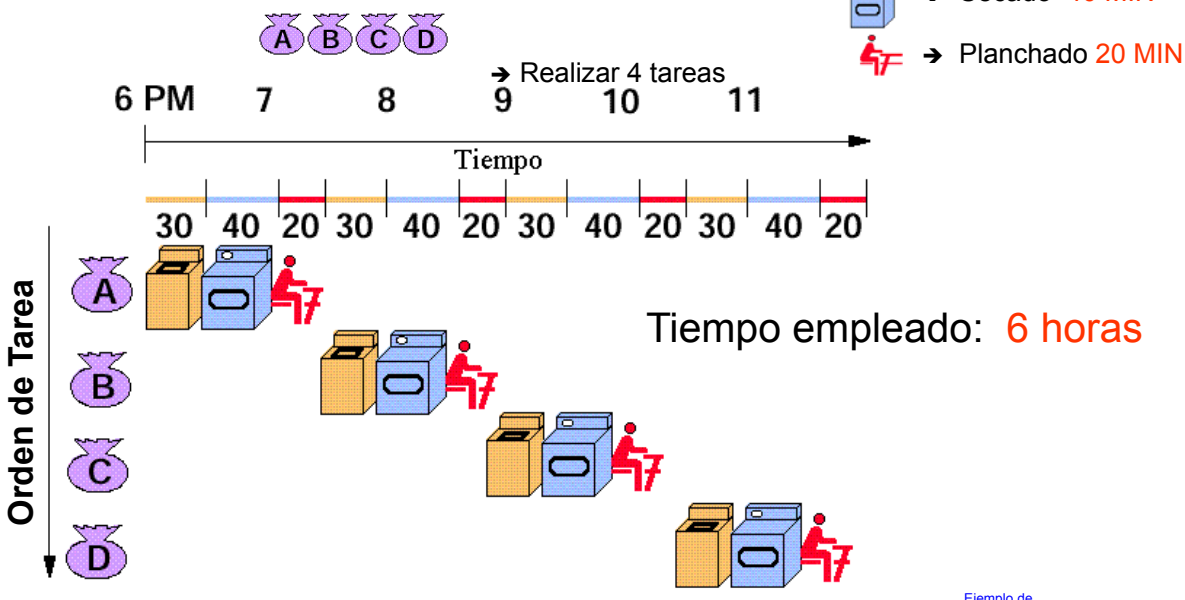
Técnica utilizada para optimizar el tiempo de ejecución de procesos que se realizan mediante la repetición de una secuencia de pasos básicos.

Permite la ejecución de procesos concurrentemente.

- ***Fundamento:*** Separar el proceso en etapas y ejecutar cada etapa en un recurso independiente.
- ***Objetivo:*** Mejorar la productividad, aumentando el número de instrucciones ejecutadas por unidad de tiempo.
- ***Funcionamiento:*** Cuando una etapa del proceso termina, el recurso liberado puede empezar a ejecutar la misma etapa del siguiente proceso.
  - Se consigue la ejecución de varios procesos en paralelo cada uno en una etapa diferente. ILP: Instruction Level Paralelism
  - Las etapas son ejecutadas secuencialmente.

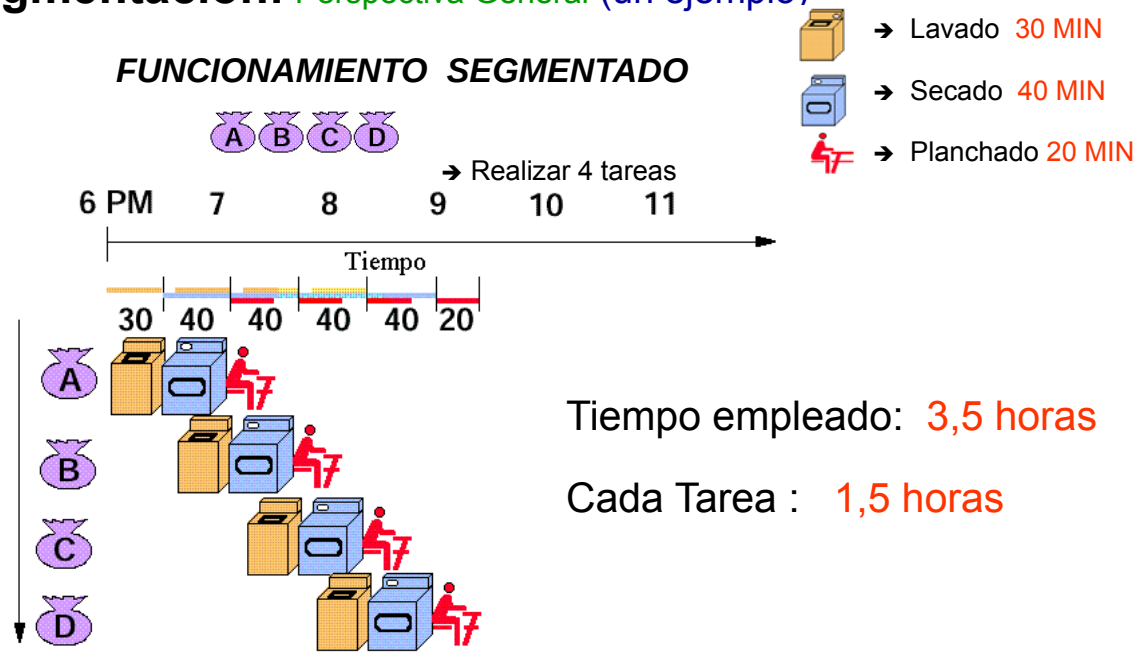
Segmentación: Perspectiva General (un ejemplo)





Ejemplo de Hennesy - Patterson

Segmentación: Perspectiva General (un ejemplo)



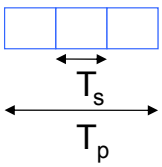
Tiempo empleado: 3,5 horas

Cada Tarea : 1,5 horas

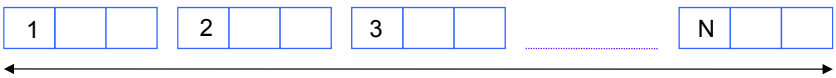
Segmentación: Funcionamiento Ideal

T_p es el tiempo de ejecución de un proceso.

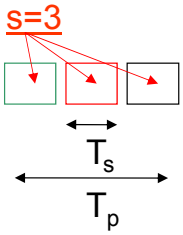
Se puede descomponer en s (s=3) etapas de duración T_s (T_p=s·T_s)



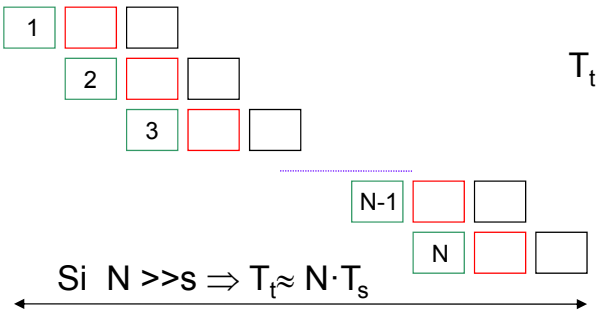
SECUENCIALMENTE (1 unidad de proceso para todas las etapas)



Ejecutar N procesos $T_t = N \cdot T_p = N \cdot s \cdot T_s$



Con SEGMENTACIÓN (unidades independientes para cada etapa)



$T_t = T_p + (N-1)T_s$
$= sT_s + (N-1)T_s$
$= (N+s-1)T_s$

Si $N >> s \Rightarrow T_t \approx N \cdot T_s$

Segmentación: Funcionamiento Ideal

Proceso segmentado vs ***Proceso secuencial***

VENTAJAS

- La segmentación, aunque no mejora la *latencia* de un solo proceso, mejora el *rendimiento o productividad (throughput)* de una tarea con muchos procesos.
- Varios procesos se ejecutan "en paralelo".

RESTRICCIONES

- La razón de segmentación está limitada por la etapa más lenta.
- La aceleración máxima posible = Número de etapas de segmentación.
- Etapas de segmentación desequilibradas ⇒ Reducción de productividad.

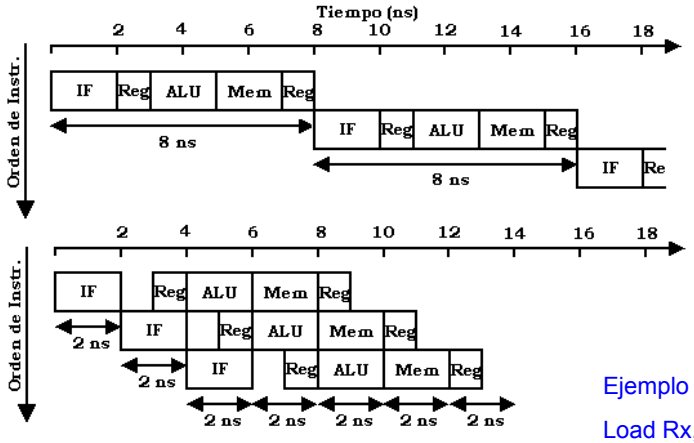
Segmentación: Funcionamiento Ideal

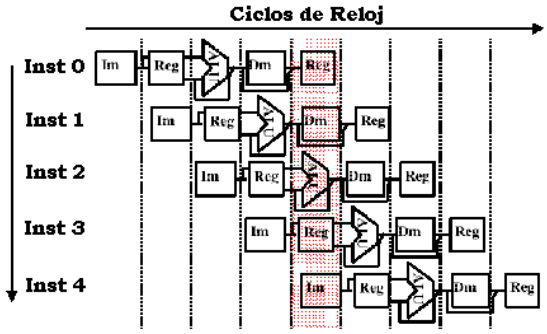
- Un procesador segmentado ***perfecto*** consigue ejecutar una instrucción por ciclo.
- La segmentación más evidente consta de tres etapas:
  - Obtener instrucción (*Fetch*)
  - Decodificar instrucción (*Decode*)
  - Ejecutar instrucción (*Execute*)
- La frecuencia de funcionamiento es mayor si el número de etapas de segmentación se incrementa. Aunque:
  - La segmentación fina es muy difícil
  - Cada nueva etapa añade el retardo de un registro
  - La independencia entre etapas es más difícil de conseguir

Segmentación: Segmentación de instrucciones.

- EJEMPLO: Segmentación de instrucciones con 5 etapas (MIPS)

IMem	Reg	ALU	DMem	Reg
Etapa IF	Etapa ID	Etapa EX	Etapa MEM	Etapa WB
➔ Obtener instrucción ➔ Acceso a la memoria de instrucciones	➔ Decodificar instrucción ➔ Lectura de operandos, carga de registros	➔ Ejecutar instrucción o bien ➔ Calcular dirección efectiva memoria.	➔ Acceso a Memoria o bien ➔ Escribir en PC la dirección de salto.	➔ Escribir en un registro el resultado de la operación.



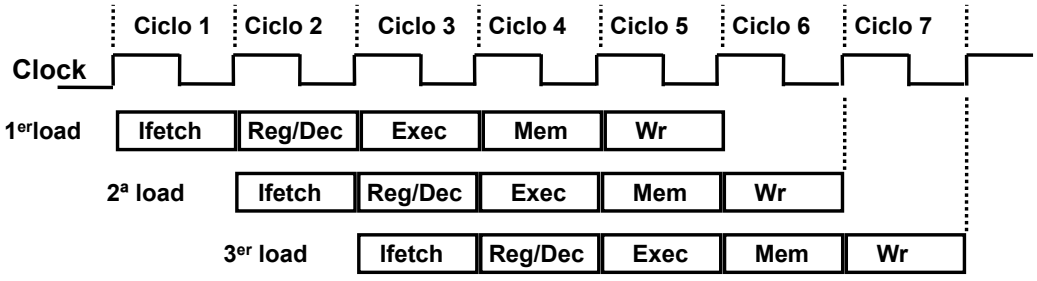


Ejemplo para tres instrucciones tipo

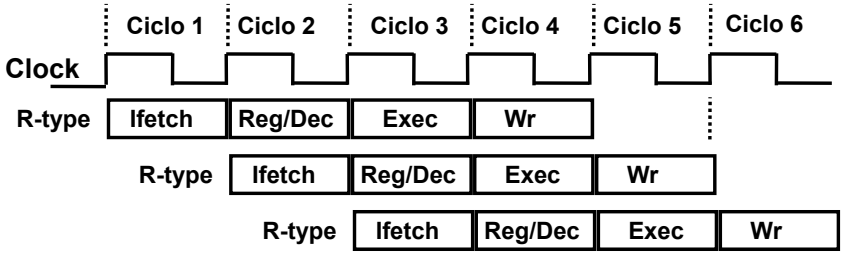
Load Rx, 100(Ry)

Segmentación: Load vs Operación entre registros

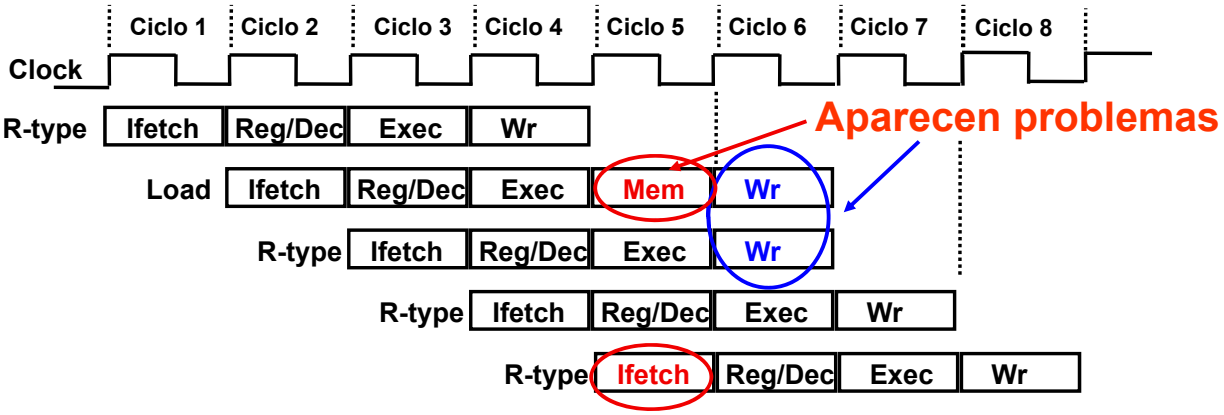
Las cinco etapas de Load



Las cuatro etapas en operaciones entre registros



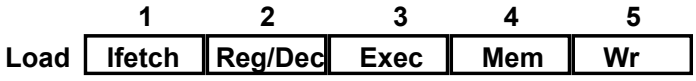
Segmentación: Load vs Operación entre registros



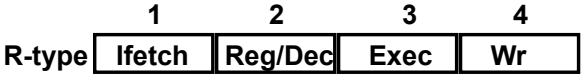
- Existen conflictos estructurales
  - Hay dos instrucciones que intentan acceder a memoria al tiempo.
  - Hay dos instrucciones que intentan escribir en el banco de registros al mismo tiempo y sólo existe un puerto de escritura.

Segmentación: Consideraciones de diseño

- Cada unidad funcional pueda usarse sólo una vez por instrucción. Deben aparecer dos unidades de memoria.
- Cada unidad funcional se utiliza en la misma etapa para todas las instrucciones:
  - Load usa el puerto de escritura en Registros durante su 5ª etapa.



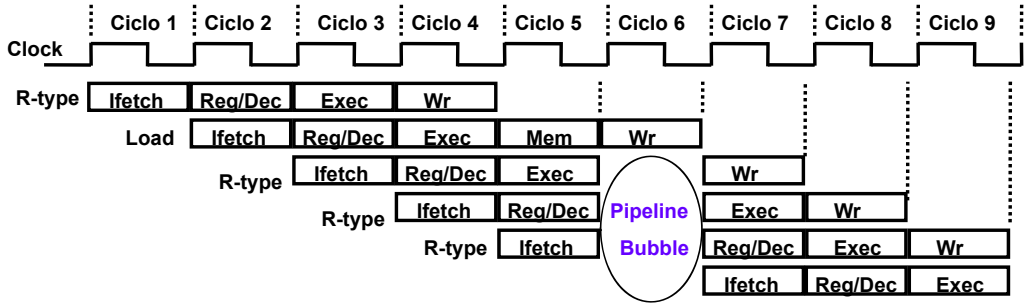
  - Las operaciones entre Registros usan el puerto de escritura en Registros durante su 4ª etapa



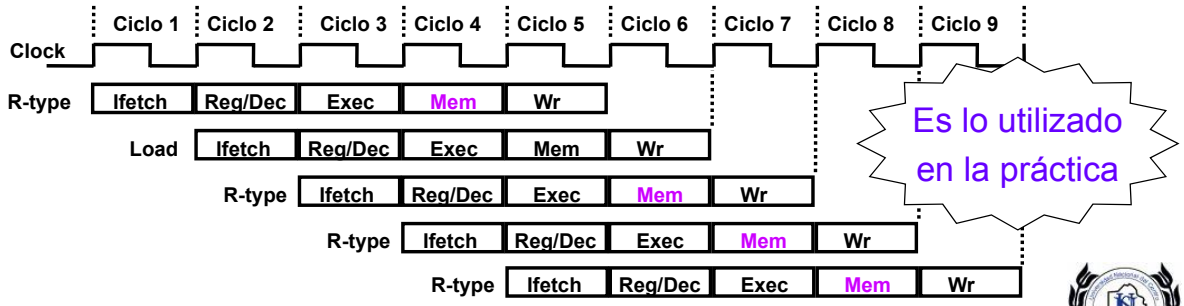
Soluciones posibles: paradas entre etapas, retraso de la escritura en registro, ...

Segmentación: Consideraciones de diseño

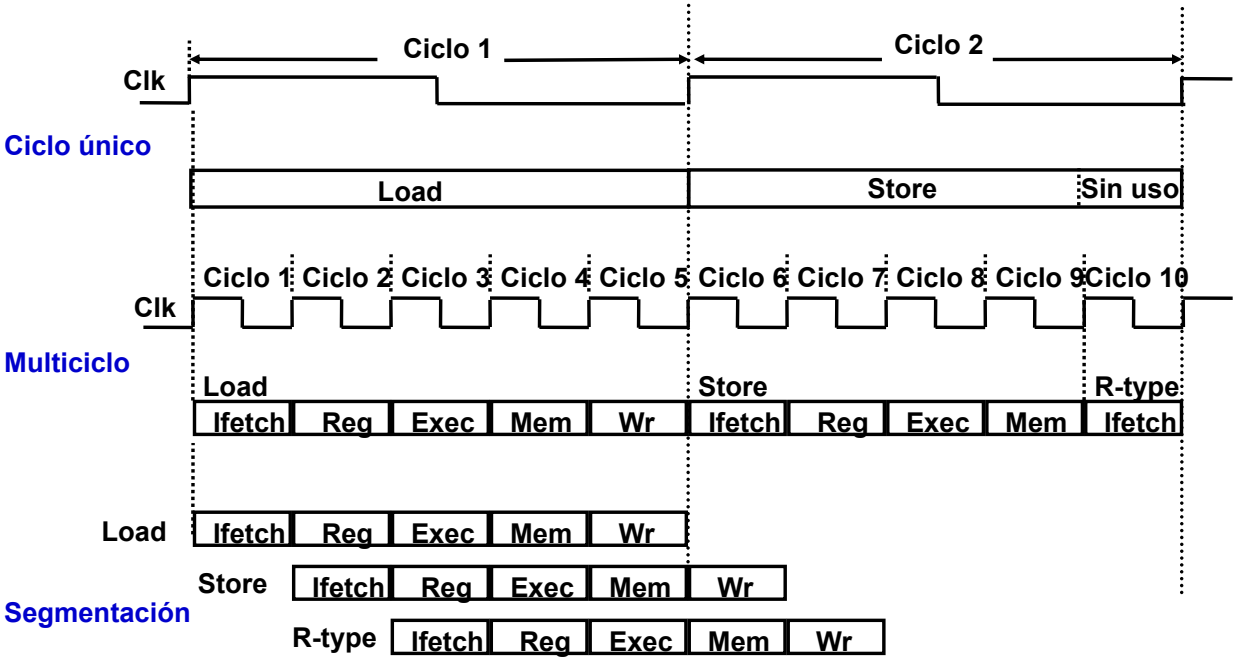
- Solución1: Parar el cauce de instrucciones (*Pipeline bubble*)



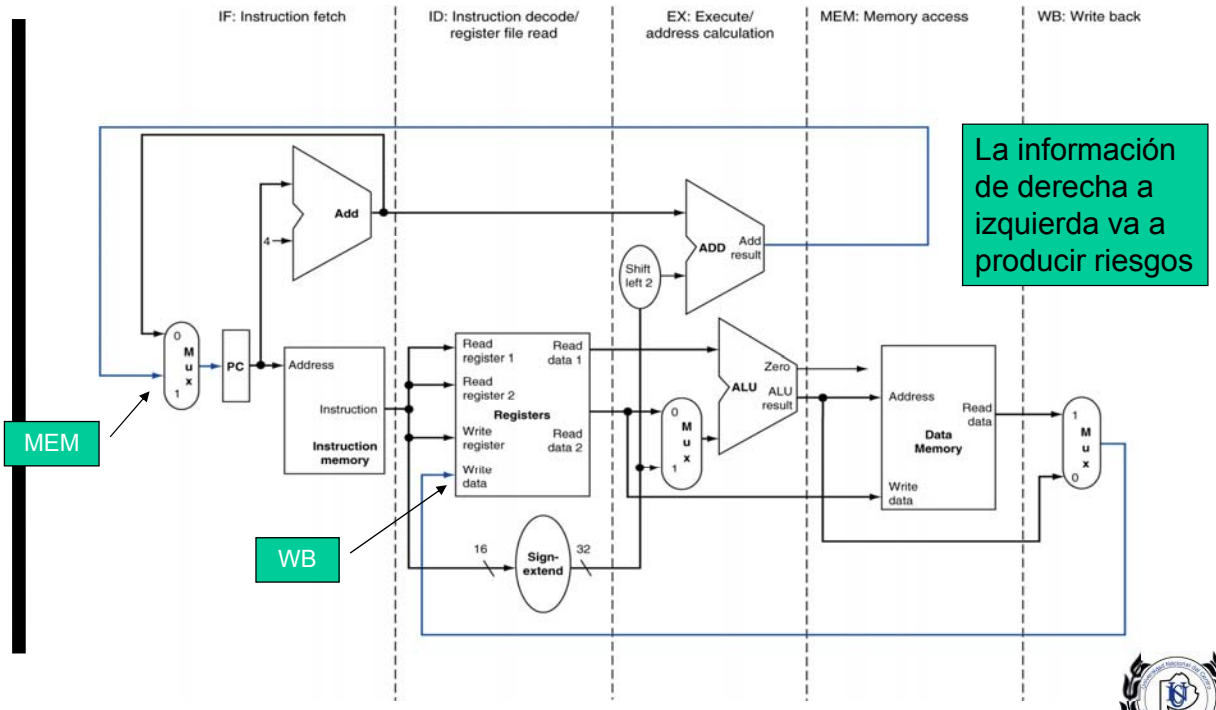
- Solución 2: Retrasar la escritura en registro



Resumen: Ciclo único vs multiciclo vs segmentación



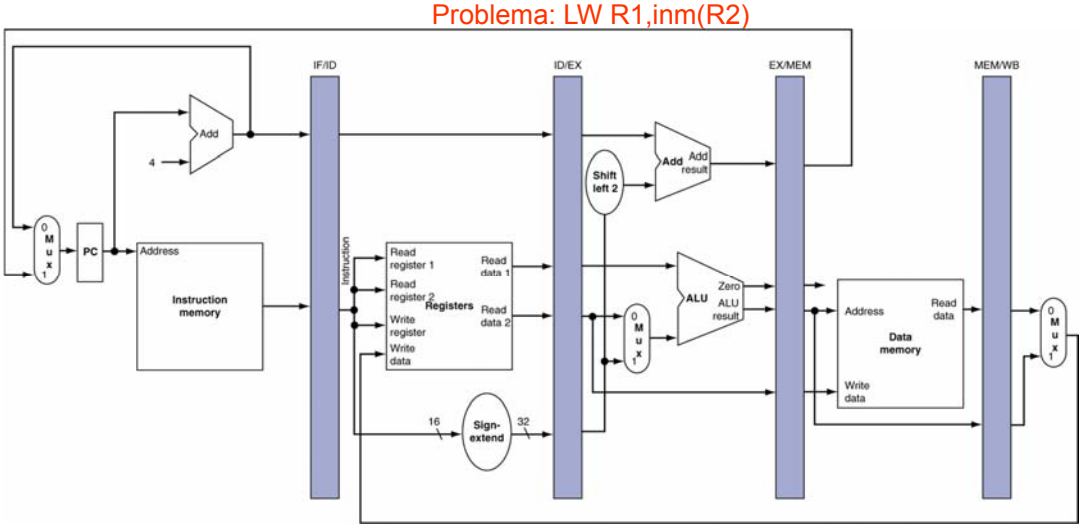
Ruta de Datos de MIPS segmentado



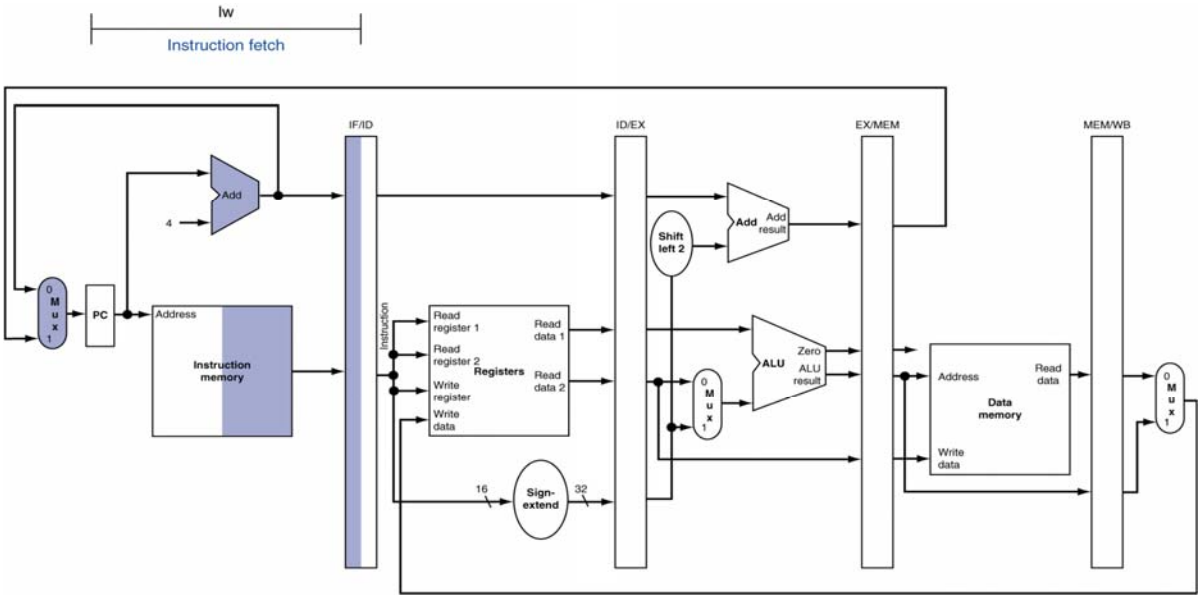
Segmentación: Un diseño con 5 etapas

- La base es el camino de datos de un ciclo.
- Se añaden registros entre etapas.
- Hay que analizar si todas las instrucciones funcionan.

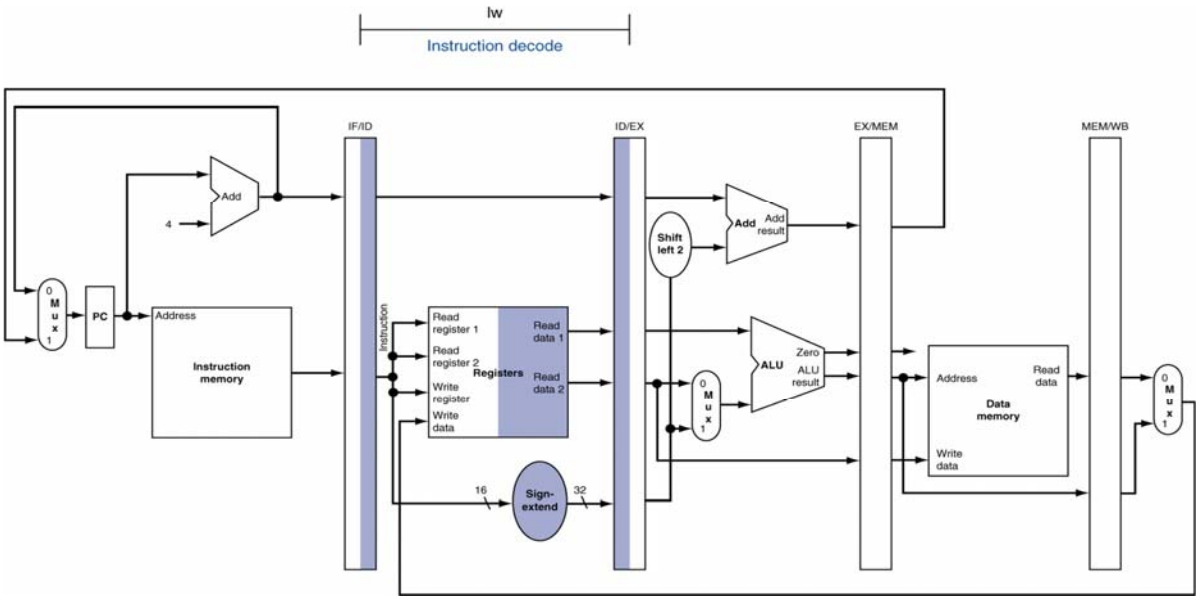
Problema: LW R1,inm(R2)



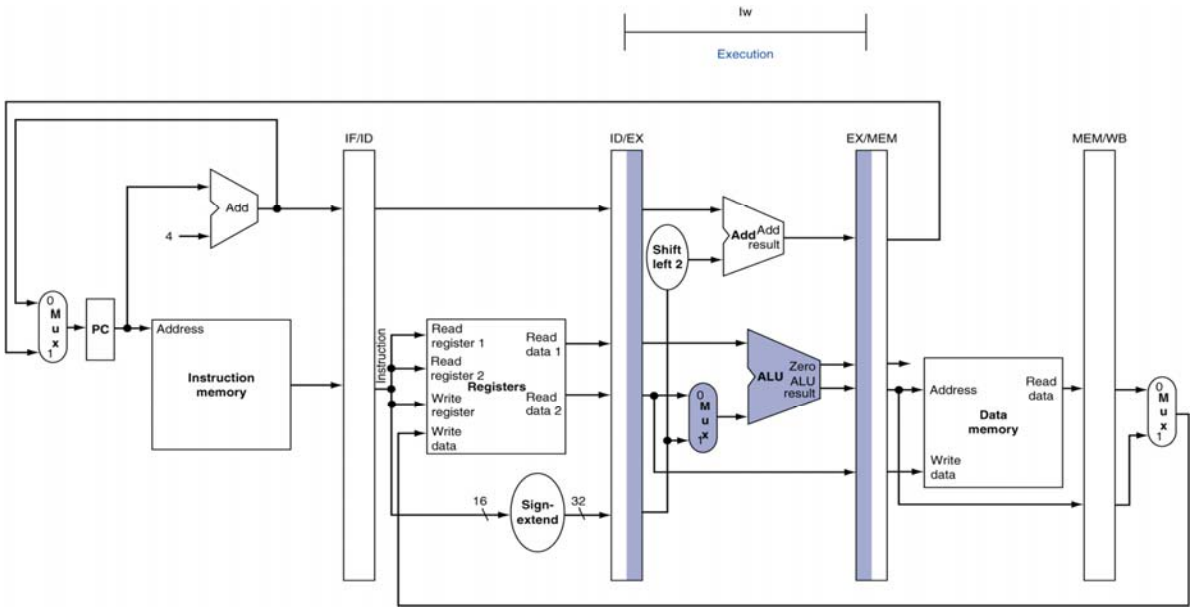
Etapa IF



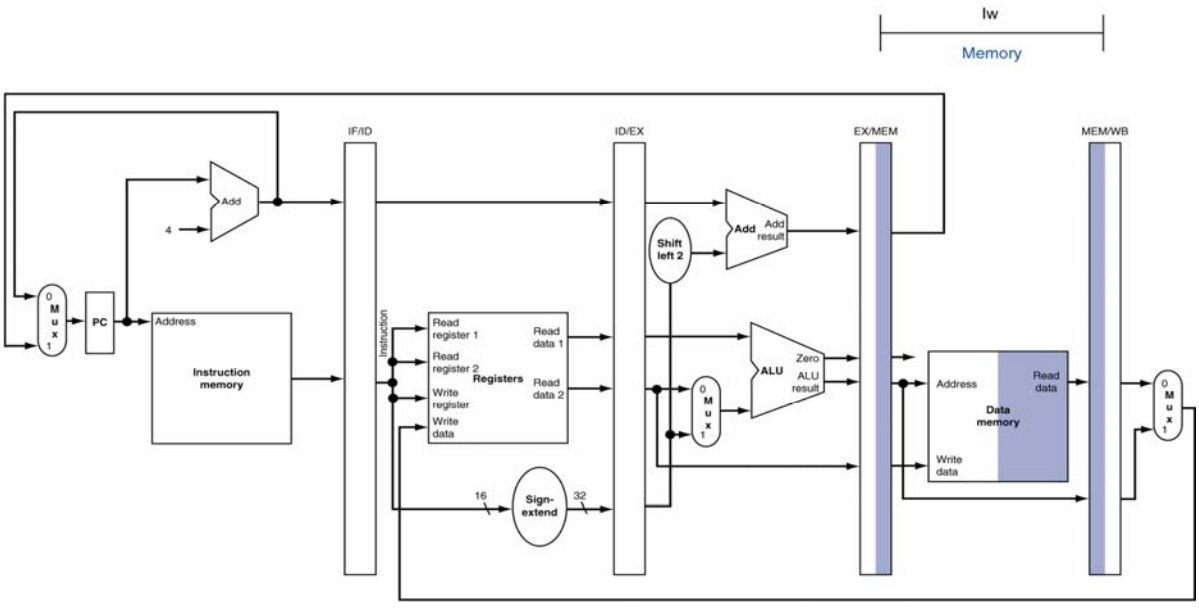
Etapa ID



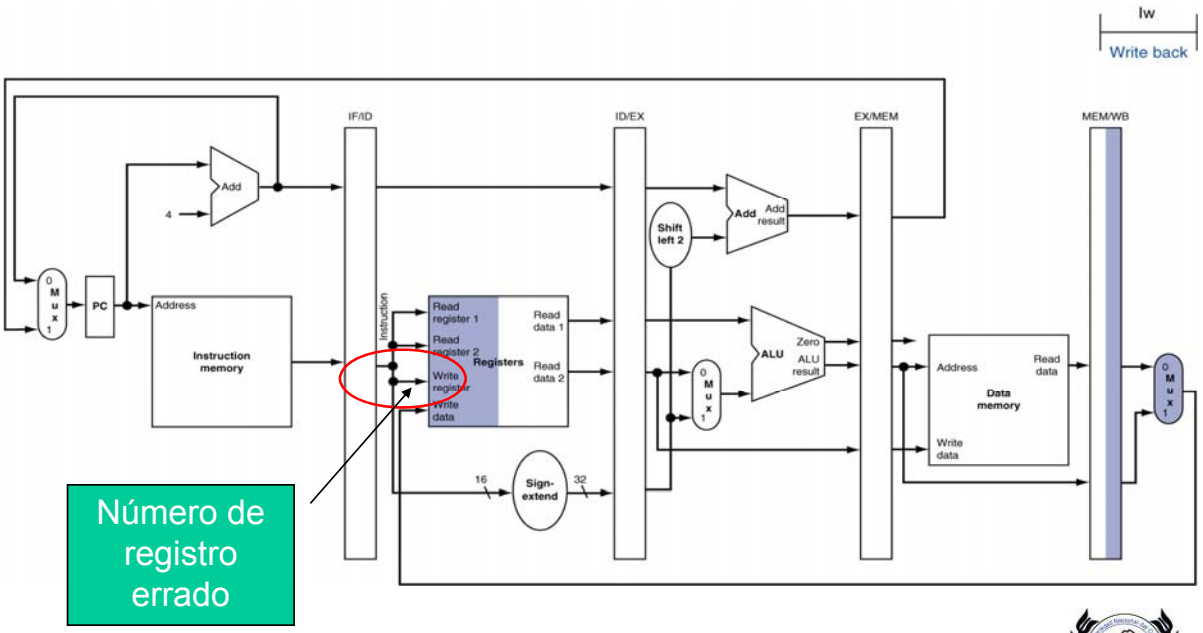
Etapa EX para Load



Etapa MEM para Load

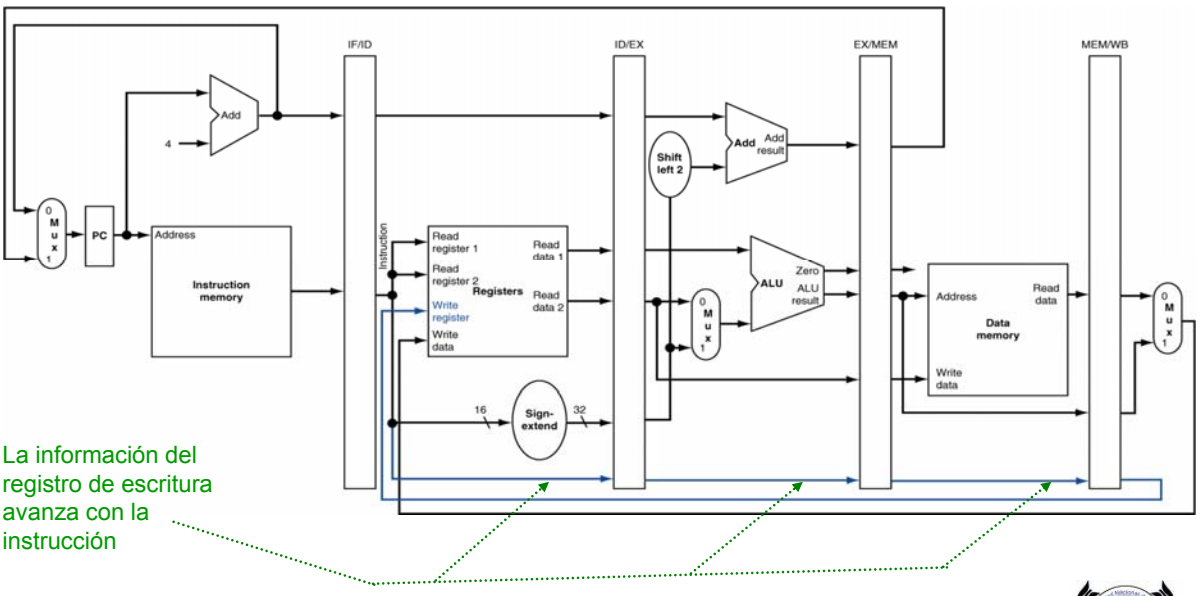


Etapa WB para Load



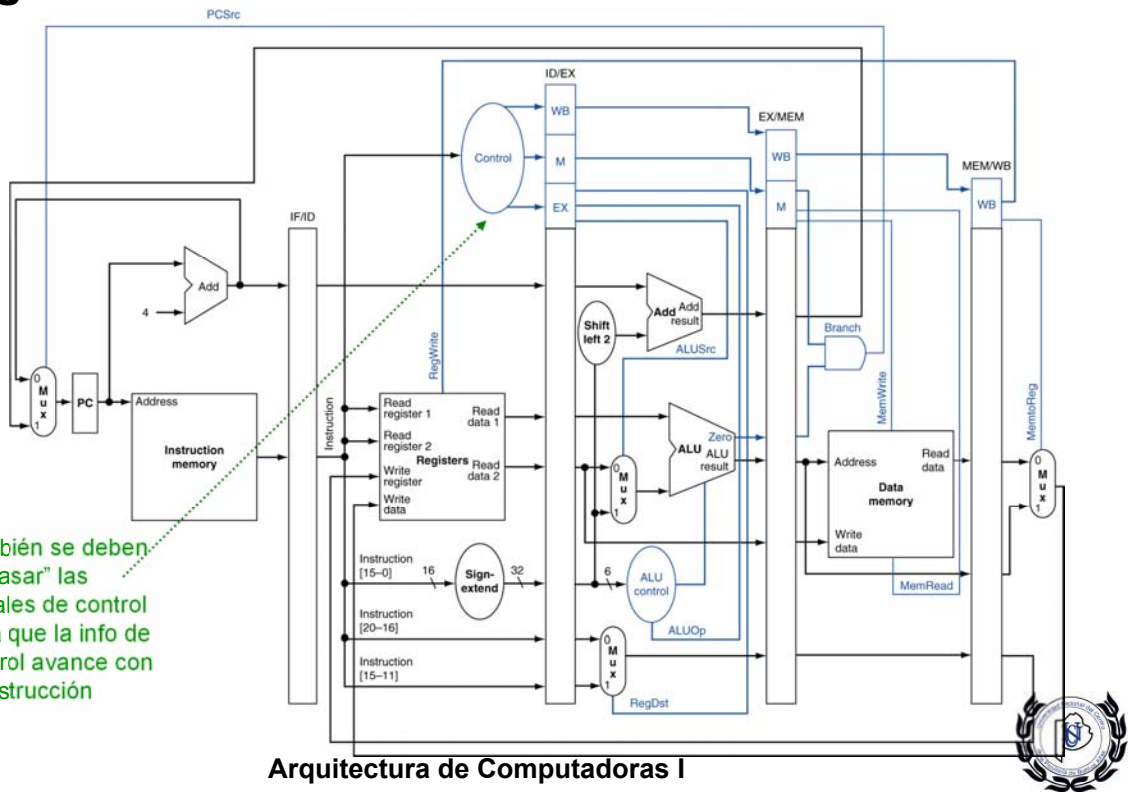
Número de registro errado

Corrección en la Ruta de Datos



La información del registro de escritura avanza con la instrucción

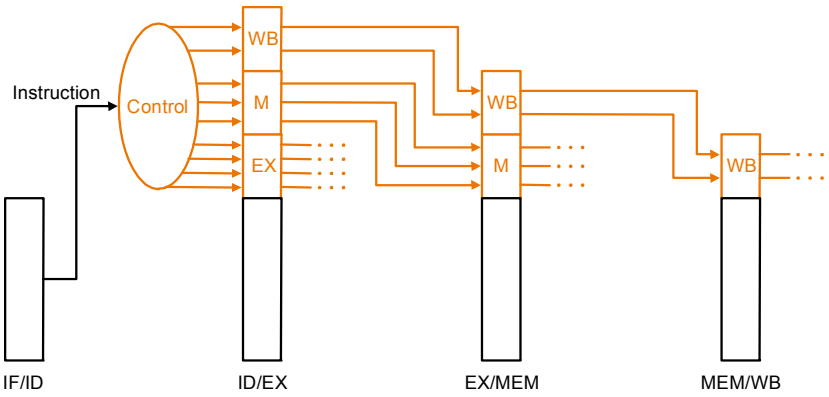
Segmentación: Añadir el control



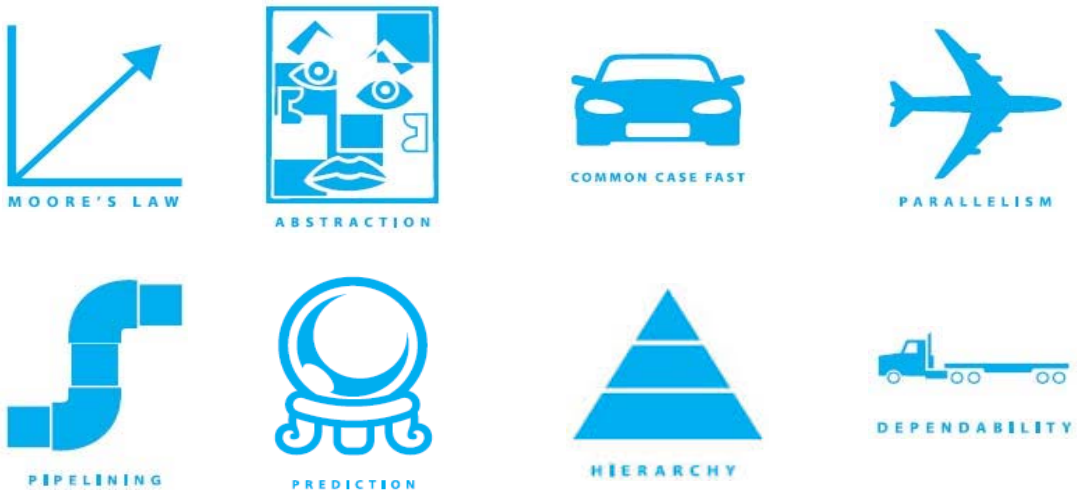
También se deben "retrasar" las señales de control para que la info de control avance con la instrucción

Segmentación: Añadir el control

- Todas las instrucciones tardan los mismos ciclos de reloj.
  - La secuenciación de la instrucción está implícita en la estructura del *pipeline*.
  - No hay un control especial para la duración de la instrucción (no hay FSM).
- Toda la información de control se calcula durante la decodificación, y se envía hacia delante a través de los registros de segmentación.
- Los valores de las líneas de control son los mismos que los calculados en el control uniciclo.



Las 8 grandes ideas en Arquitectura de Computadoras



Segmentación: Conflictos en funcionamiento real

- Las causas que pueden reducir el rendimiento en un procesador segmentado de instrucciones son tres:

- ***Riesgos estructurales***:
  - Se intenta usar el mismo recurso de dos maneras diferentes al mismo tiempo.
  - El hardware impide una cierta combinación de operaciones.

- ***Riesgos por dependencia de datos***:
  - Se intenta usar un dato antes de que esté disponible.
  - El operando de una instrucción depende del resultado de otra instrucción precedente que todavía no se ha obtenido.

- ***Riesgos de control***:
  - Se intenta tomar una decisión antes de evaluarse la condición.
  - Si se salta, las instrucciones posteriores no deben ejecutarse (o al menos, no deben finalizar).

Segmentación: Conflictos en funcionamiento real

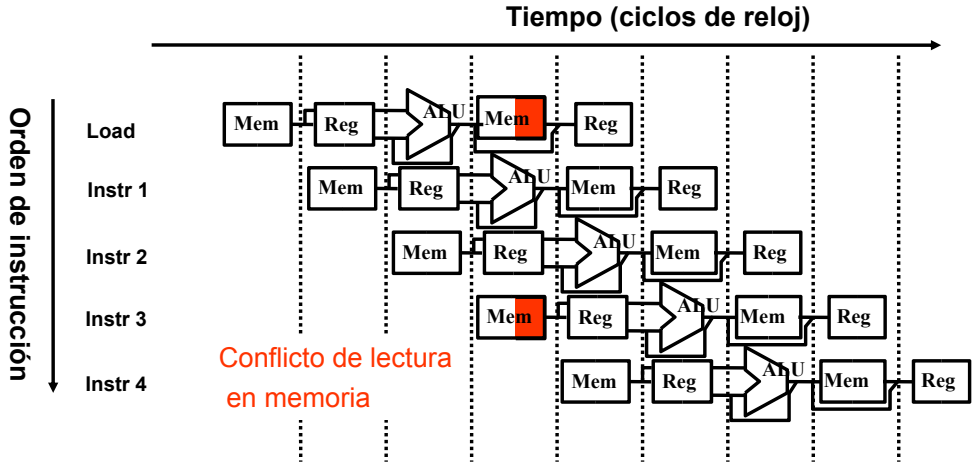
¿Todos estos riesgos se pueden solucionar?.....Sí

¿Cómo?.....Esperando

- Cuando se detecta un riesgo, la solución más simple es parar la segmentación (*stall the pipeline*) hasta que desaparezca el riesgo.
  - Las instrucciones que preceden a la causante del riesgo pueden continuar.
  - La instrucción que causa el riesgo y siguientes no continúan hasta que desaparece el riesgo.
- Se necesita que el control de la segmentación (*pipeline)* sea capaz de:
  - Detectar las causas de riesgo.
  - Decidir acciones que resuelvan el riesgo (por ejemplo, esperar).

Segmentación: Riesgos estructurales

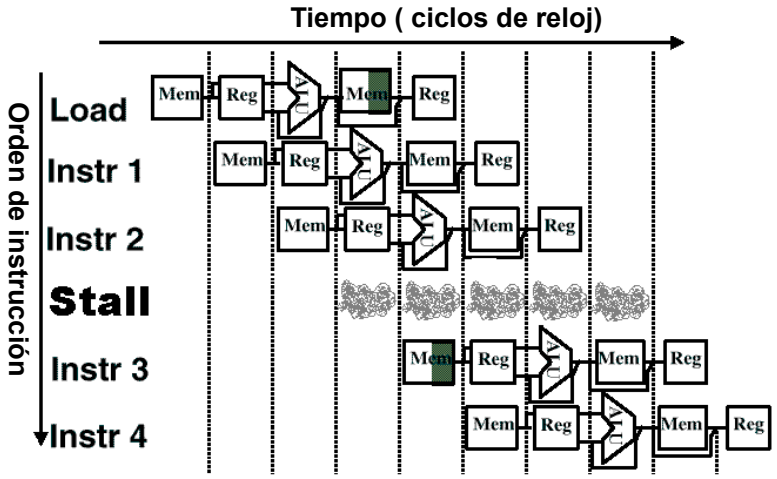
- Casos que se pueden presentar. Accesos simultáneos a:
  - Memoria (si es Von Neuman, única para datos e instrucciones).
  - Unidades funcionales.
  - Registros internos.



(La mitad izquierda coloreada indica escritura y la mitad derecha lectura)

Segmentación: Riesgos estructurales

- Soluciones:
  - Introducir esperas.
  - Duplicar recursos o separar memoria de datos de la memoria de instrucciones (Harvard en lugar de Von Neuman).



Segmentación: Riesgos por dependencia de datos

- Dependencias que se presentan para 2 instrucciones i y j, con i ejecutándose antes que j.
  - RAW (Read After Write): la instrucción posterior j intenta leer una fuente antes de que la instrucción anterior i la haya modificado.
  - WAR (Write After Read): la instrucción j intenta modificar un destino antes de que la instrucción i lo haya leído como fuente.
  - WAW (Write After Write): la instrucción j intenta modificar un destino antes de que la instrucción i lo haya hecho (se modifica el orden normal de escritura).

✓ Ejemplos:

RAW

```
ADD  r1, r2, r3
SUB  r5, r1, r6
AND  r6, r5, r1
ADD  r4, r1, r3
SW   r10, 100(r1)
```

WAR

```
ADD r1, r2, r3
OR  r3,r4, r5
```

WAW

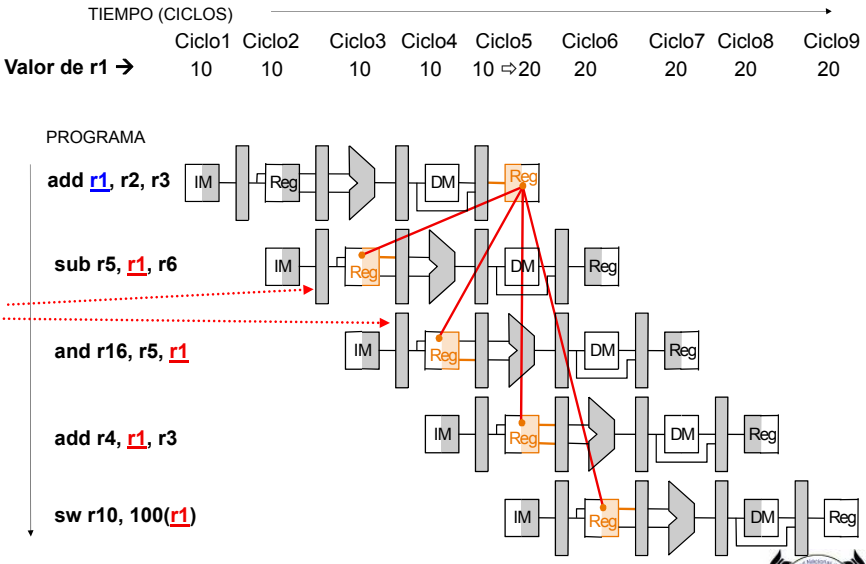
```
DIV  r1, r2, r3
AND r1,r4, r5
```

En micros segmentados con ejecución en orden sólo son problema los RAW

Segmentación: Riesgos por dependencia de datos

- Aparecen problemas al poder empezar la siguiente instrucción antes de que la primera haya terminado.

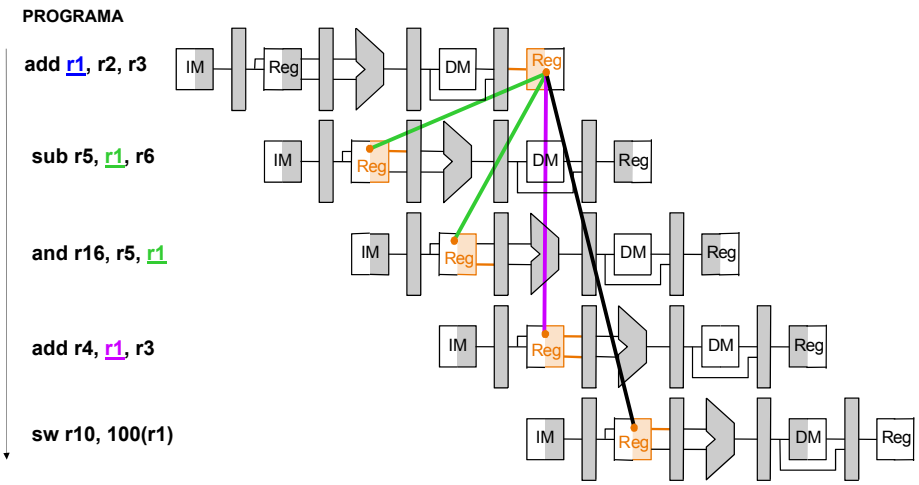
- Ejemplo (RAW):



Dependencias problemáticas son aquellas que necesitan datos que hay que buscar "**hacia atrás**" en el esquema de tiempos.

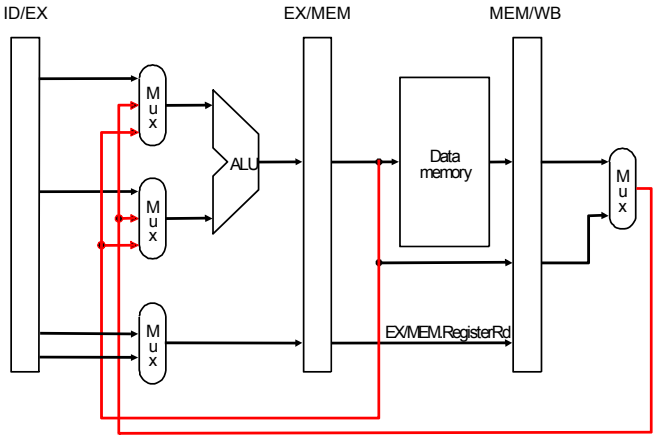
Segmentación: Riesgos por dependencia de datos

- Soluciones:
  - "Adelantar" (*forward*) el resultado de una etapa a las siguientes.
  - Definir adecuadamente la secuencia Read/Write (la instrucción "add r4,r1,r3" funciona correctamente si en la etapa WB, Write se realiza en la 1ª mitad del ciclo y Read en la 2ª).



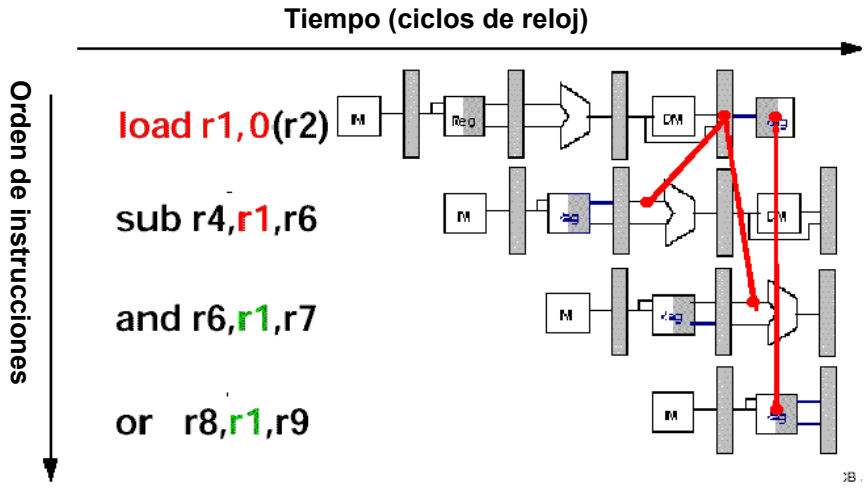
Segmentación: Riesgos por dependencia de datos

- Necesidades hardware para adelantar resultados:
  - Multiplexores adicionales donde se vaya a recibir el dato (p. ej. en las entradas de datos de la ALU).
  - Buses extra entre registros internos y multiplexores.
  - Comparadores entre los operandos de una instrucción y los operandos destino de instrucciones previas.



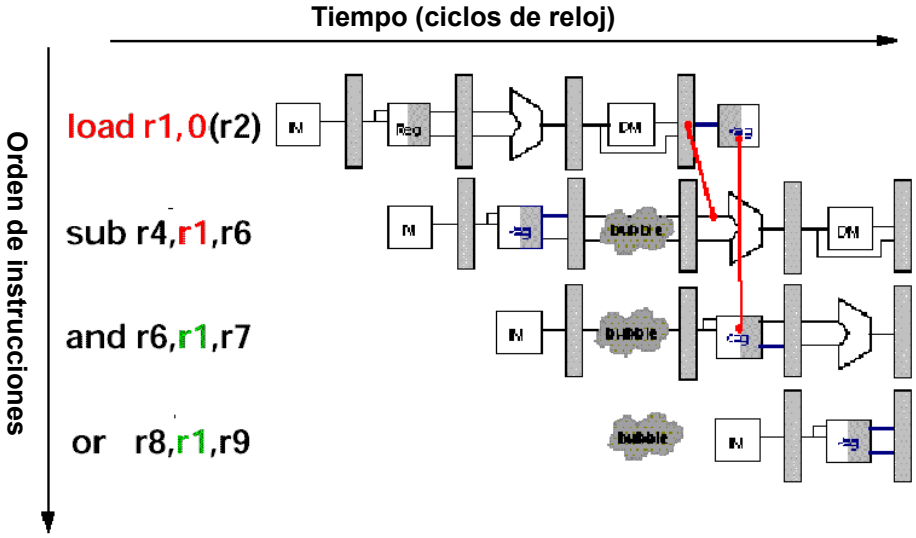
Segmentación: Riesgos por dependencia de datos

- Los riesgos pueden persistir incluso con adelantamiento de datos
  - Ej: tras la instrucción LOAD se pueden evitar los riesgos en AND y en OR con adelantamiento de datos, pero no de SUB (no puede adelantar resultados a etapas que son de tiempos anteriores)

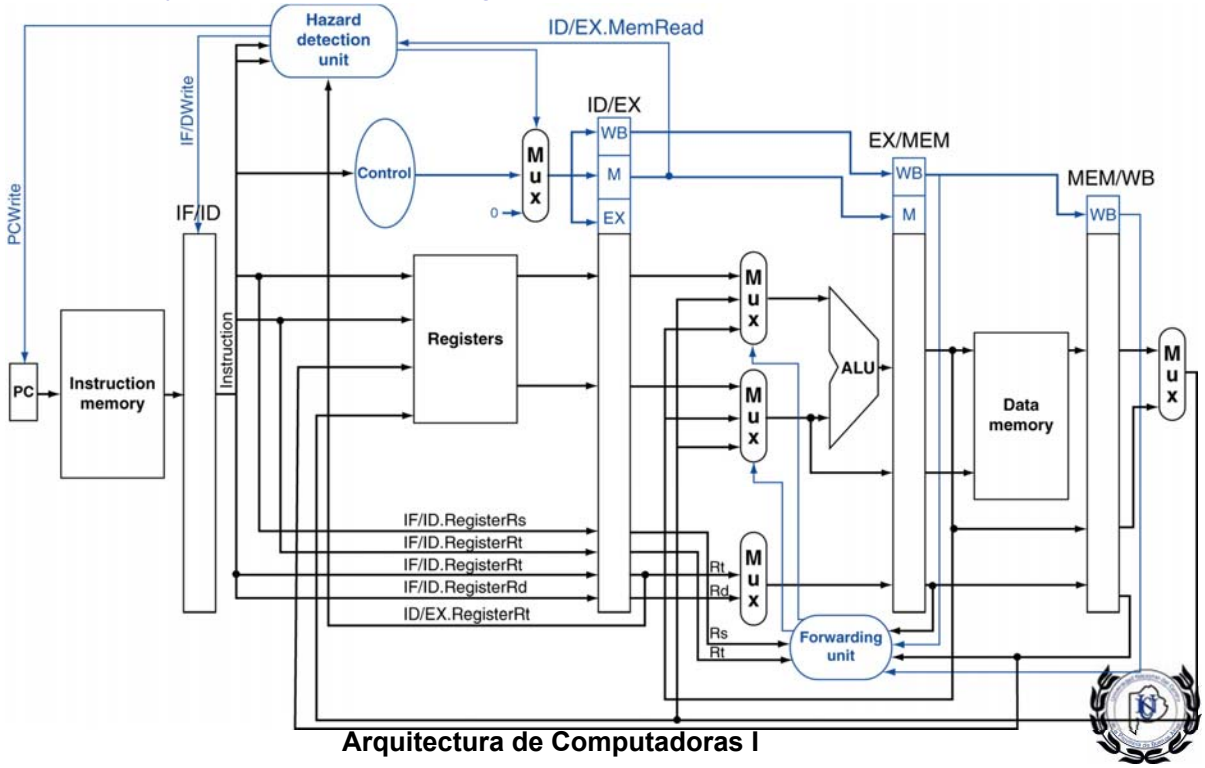


Segmentación: Riesgos por dependencia de datos

- Soluciones:
  - Insertar un ciclo de espera (*stall*) en el ciclo 3º, para la instrucción SUB y siguientes
  - Insertar una operación NOP detrás del LOAD (es lo más utilizado, lo puede automatizar el compilador sin necesidad de más hardware)

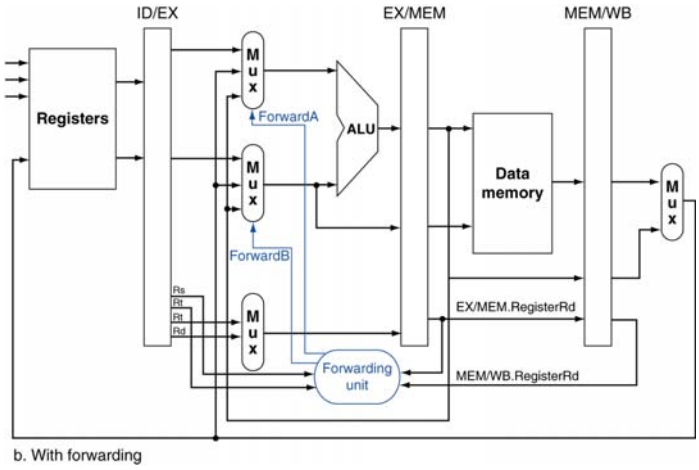


Segmentación: Ruta de datos con control para adelantamiento de datos y detección de riesgos



Segmentación: Control para adelantamiento de datos

- Unidad de adelantamiento de datos (*forwarding*)
  - Se debe detectar el riesgo y luego “anticipar” el valor a su destino
  - Se agrega un bloque combinacional para detectar el riesgo y multiplexores para adelantar los datos oportunamente

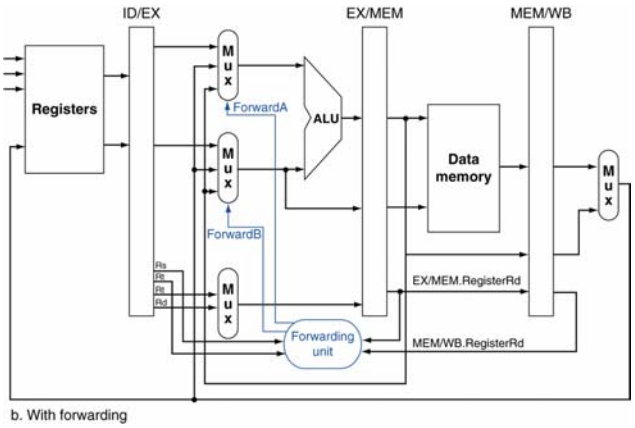


Segmentación: Control para adelantamiento de datos

- Unidad de adelantamiento de datos (*forwarding*)
  - Se debe detectar el riesgo y luego “anticipar” el valor a la ALU
  - Existen 4 riesgos potenciales:

1a. EX/MEM.Registro.Rd = ID/EX.Registro.Rs
1b. EX/MEM.Registro.Rd = ID/EX.Registro.Rt
2a. MEM/WB.Registro.Rd = ID/EX.Registro.Rs
2b. MEM/WB.Registro.Rd = ID/EX.Registro.Rt

Analizar por ejemplo con:
Add r1, r2, r3
Sub r5, r1, r6
And r6, r5, r1
Add r4, r1, r3



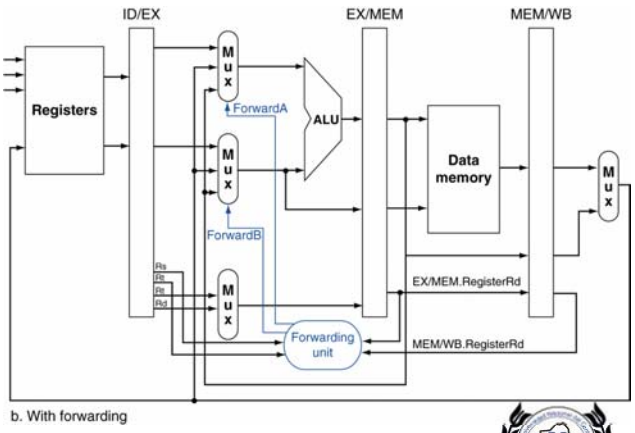
Segmentación: Control para adelantamiento de datos

- Unidad de adelantamiento de datos (*forwarding*)
  - Pseudo-código del funcionamiento:

(Riesgos en EX)

if (EX/MEM.EscrReg **and**
EX/MEM.RegistroRd ≠ 0 **and**
EX/MEM.Registro.Rd = ID/EX.Registro.Rs)
then AnticiparA = 10
else AnticiparA = 00

if (EX/MEM.EscrReg **and**
EX/MEM.RegistroRd ≠ 0 **and**
EX/MEM.Registro.Rd = ID/EX.Registro.Rt)
Then AnticiparB = 10
else AnticiparB = 00



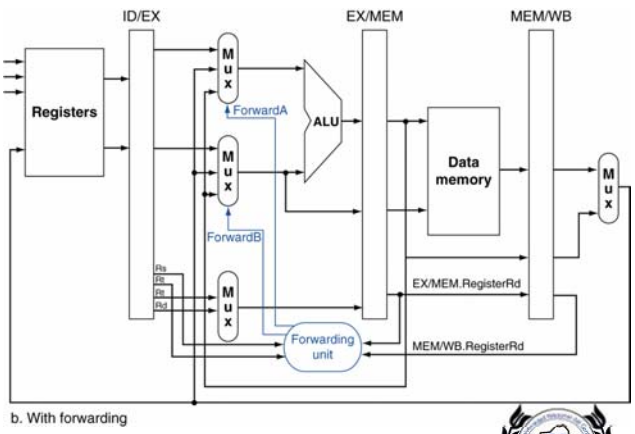
Segmentación: Control para adelantamiento de datos

- Unidad de adelantamiento de datos (*forwarding*)
  - Pseudo-código del funcionamiento:

(Riesgos en MEM)

if (MEM/WB.EscrReg **and**
MEM/WB.RegistroRd ≠ 0 **and**
EX/MEM.Registro.Rd ≠ ID/EX.RegistroRs
and
MEM/WB.Registro.Rd =
ID/EX.RegistroRs)
then AnticiparA = 01
else AnticiparA = 00

if (MEM/WB.EscrReg **and**
MEM/WB.RegistroRd ≠ 0 **and**
EX/MEM.Registro.Rd ≠ ID/EX.RegistroRt
and
MEM/WB.Registro.Rd = ID/EX.Registro.Rt)
then AnticiparB = 01
else AnticiparB = 00



Segmentación: Detección de riesgos insalvables

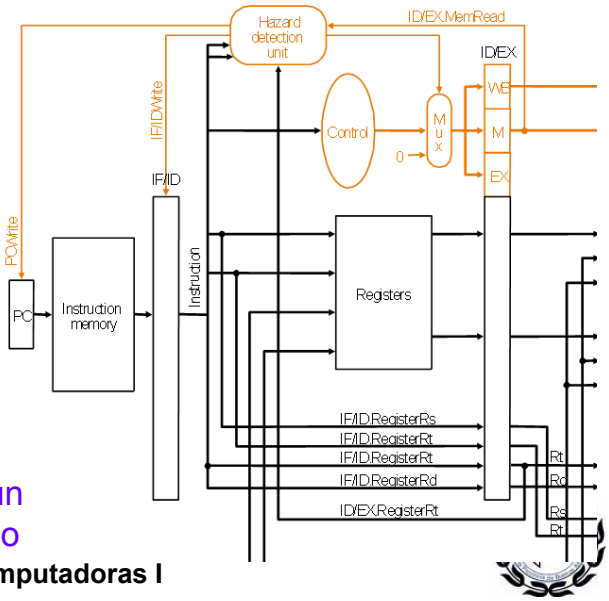
- Unidad de detección de riesgos (*hazard detection unit*)
  - Para cuando el adelantamiento no resuelve los riesgos (caso de load y uso del registro destino en la siguiente instrucción)

Pseudo-código del funcionamiento:

If (ID/EX.LeerMem **and**
((ID/EX.Registro.Rt = IF/ID.Registro.Rs) **or**
(ID/EX.Registro.Rt = IF/ID.Registro.Rt)))
then *Bloquear el pipeline*

Bloquear el pipeline:

- ✓ **PCWrite** = 0
- ✓ **IF/IDWrite** = 0
- ✓ **MuxNOP** = 1



Todo esto no es necesario si el compilador / ensamblador añade un NOP después del LOAD con riesgo

Una solución SW para el caso del lw

El compilador puede "reacomodar" el código para evitar *stalls*

Ejemplo en C: `A = B + E; C = B + F;`

stall

stall

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 ciclos

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 ciclos

Riesgos de control (instrucciones de salto)

- Las instrucciones de salto pueden suponer una alteración del orden secuencial de ejecución.

Etapa IF	Etapa ID	Etapa EX	Etapa MEM	Etapa WB
➔ Obtener instrucción ➔ Acceso a la memoria de instrucciones	➔ Decodificar instrucción. ➔ Lectura de operandos, carga de registros	➔ Ejecutar instrucción ➔ Calcular la dirección efectiva salto (PC+Inm.)	➔ Acceso a Memoria o bien ➔ Resolver la condición y escribir en PC la dirección de salto.	➔ Escribir en un registro el resultado de la operación

- No se sabe si el salto es efectivo hasta la etapa de ejecución y no se actualiza la dirección destino (caso de que sea efectivo) hasta la cuarta etapa => Pérdida de 3 ciclos
- Mejora: Adelantar el cálculo del destino (*target*) PC+Inmediato y adelantar la resolución de la condición (y actualizar el PC) a la 2ª etapa.

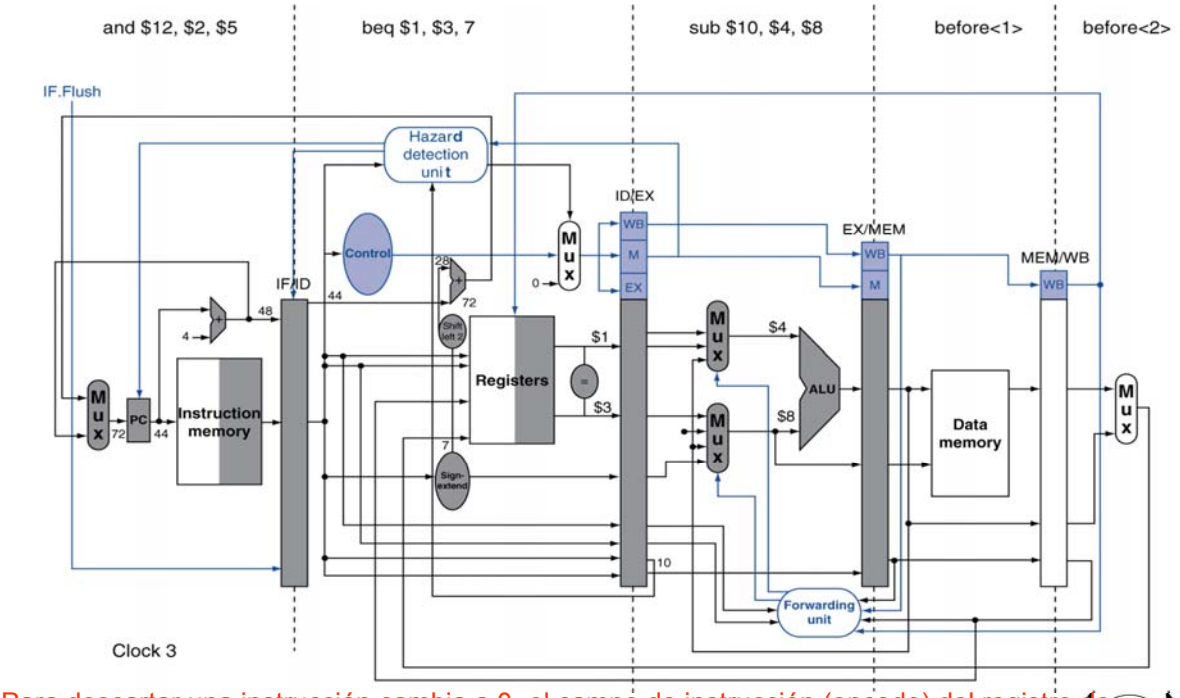
Riesgos de control (instrucciones de salto)

Cuando se decide saltar, ya se están ejecutando otras instrucciones en el cauce segmentado => Se necesita incluir hardware para vaciar (*flushing*) el *pipeline*

Ejemplo: salto efectivo

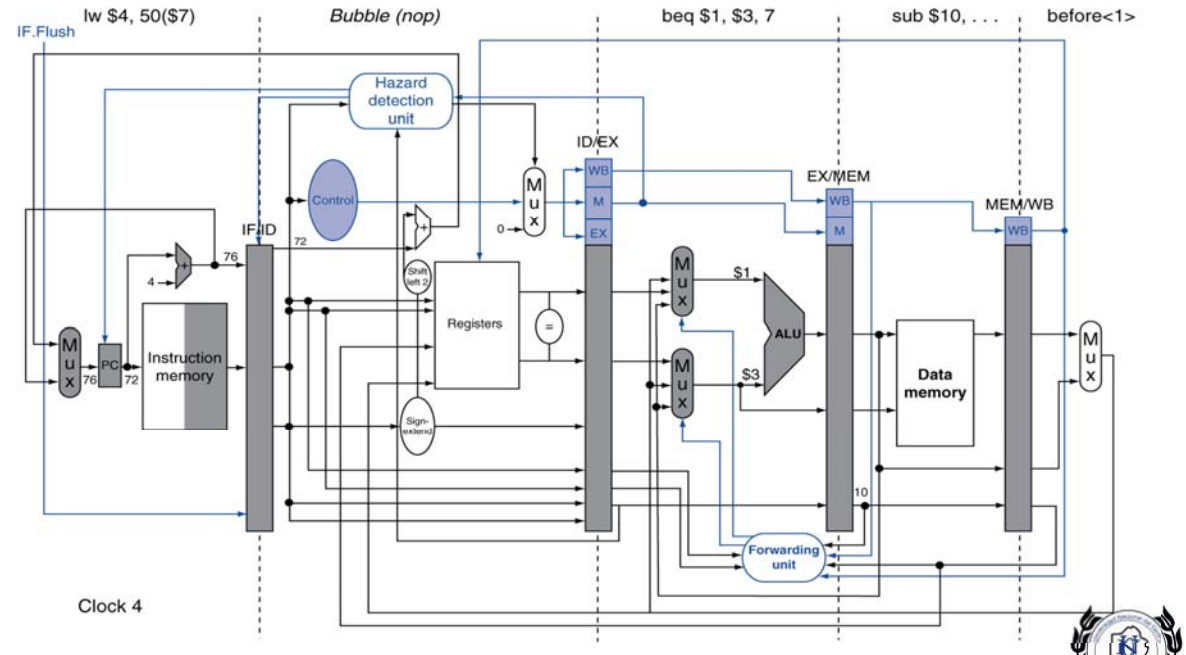
```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

Ejemplo: Salto Efectivo



Para descartar una instrucción cambia a 0 el campo de instrucción (opcode) del registro de segmentación IF/ID ⇒ **NOP**

Ejemplo: Salto Efectivo



Riesgos de Control (Instrucciones de salto)

¿QUÉ HACER CON LAS SIGUIENTES INSTRUCCIONES A LA DE SALTO CONDICIONAL?

1. Esperar hasta que la dirección y condición del salto estén definidas.
   - ✓ Conviene conocer la dirección de salto y la condición tan pronto como sea posible.
2. Salto retardado, la(s) instrucción(es) posterior(es) siempre se ejecuta(n).
   - ✓ El compilador rellena con instrucciones válidas los huecos de retardo.
3. Predecir si se va a saltar o no.
   - ✓ Se ejecuta especulativamente, en caso de error se debe "vaciar" el procesador.
4. Anticipar la dirección más probable para el salto (BTB).
   - ✓ Se ejecuta especulativamente, se almacena en una caché la dirección del último salto.
5. Ejecutar todos los caminos.
   - ✓ Implica la duplicación del hardware.
6. Ejecución con predicados.

Riesgos de Control. Salto retardado

La siguiente instrucción al salto siempre se termina de ejecutar, se salte o no.
El compilador utiliza tres estrategias para buscar una/varias instrucciones de relleno:

	■ DEL BLOQUE BÁSICO	■ SI SALTO PROBABLE, DEL BLOQUE DESTINO	■ SI SALTO NO PROBABLE, DEL BLOQUE SECUENCIAL

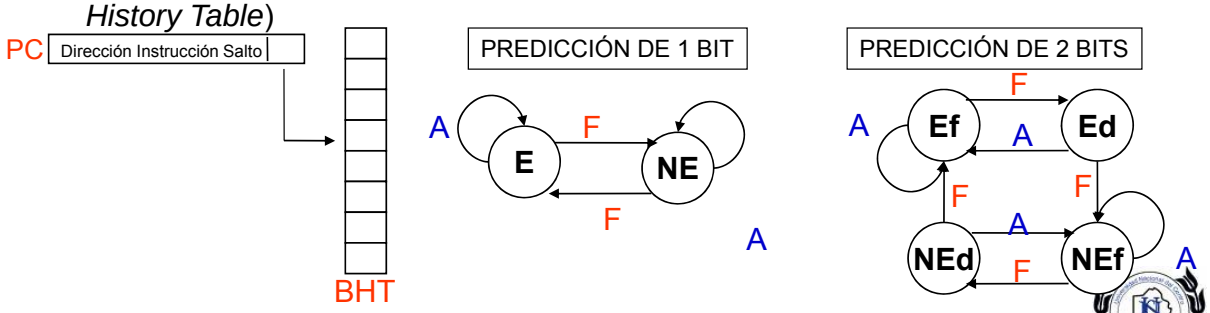
add r1, r2, r3
bnz r2, L1
nop
	.....
	bnz r2, L1
	add r1, r2, r3
		add r1, r2, r3
		bnz r2, L1
		and r2, r3, r2
			add r1, r2, r3
			bnz r2, L1
			sub r6,r7,r6

sub r6,r7,r6
mul r2, r3, r8
.....
	sub r6,r7,r6
	mul r2, r3, r8
	.....
		sub r6,r7,r6
		mul r2, r3, r8
		.....
			.....
			mul r2, r3, r8
			.....

L1: and r2, r3, r2 andi r5,r6,inm	L1: and r2, r3, r2 andi r5,r6,inm	L1: andi r5,r6,inm	L1: and r2, r3, r2 andi r5,r6,inm
	■ Operación válida si la instrucción no afecta a la condición del salto. Siempre se realiza trabajo útil.	■ Operación válida siempre siempre que r2 no se utilice como fuente y sea modificada como destino en el en bloque secuencial. (**CORRECTO**)	■ Operación válida siempre que r6 no se utilice como fuente y sea modificada como destino en el en bloque destino. (**INCORRECTO**)

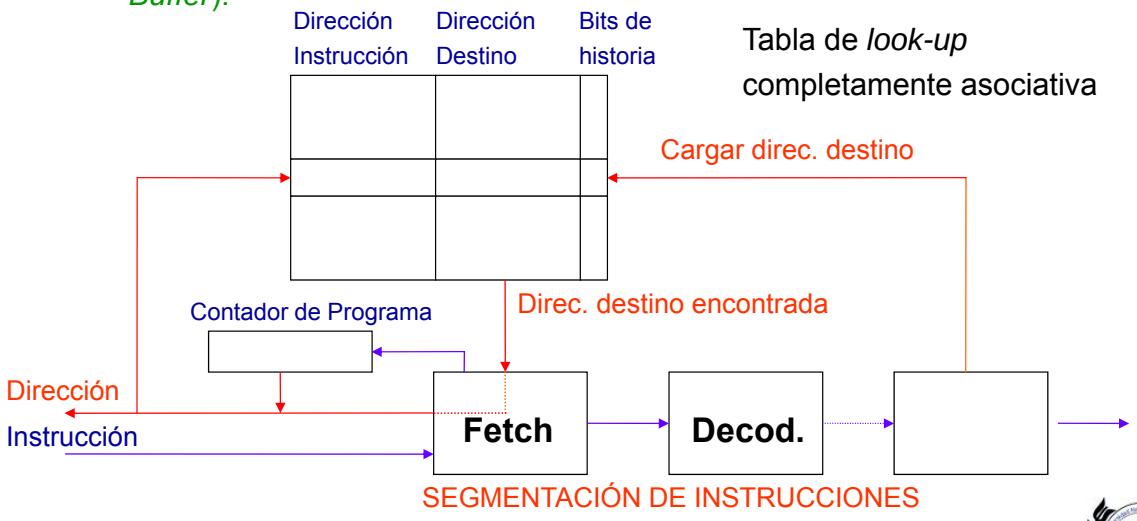
Riesgos de Control. Predecir el salto

- Predicción estática: siempre predice el mismo sentido del salto.
  - Ejecuta especulativo hasta que se resuelve la condición (normalmente, ciclo 3-4)
  - En caso de error debe eliminar los resultados especulativos
    - Predicción efectiva (E), el salto se realiza
    - Predicción no efectiva (NE), el salto no se realiza
    - Predicción NE si el salto es adelante y E si es hacia atrás
- Predicción dinámica: cambia la predicción en función de la historia del salto.
  - Utiliza una pequeña memoria asociada a cada dirección de salto (BHT, *Branch History Table*)



Riesgos de Control. Anticipar la dirección más probable

- Utiliza la técnica de predicción histórica anterior
  - ✓ Utiliza una tabla asociativa (tipo caché) que incorpora, para cada instrucción de salto, la dirección de destino de la predicción anterior.
  - ✓ A la tabla se la conoce como buffer de destino de saltos o BTB (*Branch Target Buffer*).



Segmentación: Tratamiento de excepciones y mejoras

- ¿Cómo afecta a la segmentación el tratamiento de excepciones (interrupciones, desbordamientos aritméticos, peticiones de E/S, servicios del sistema operativo, uso de instrucciones no definidas, mal funcionamiento de la circuitería, etc)?
  - Básicamente detectarlo y vaciar el *pipeline* para dar el control a alguna rutina de tratamiento de excepciones
- ¿Cómo mejorar aún más la segmentación?
  - Super-segmentación (implica segmentación en operaciones aritméticas)
  - Superescalares (replicar rutas de datos)
  - Planificación dinámica del *pipeline*

Las 8 grandes ideas en Arquitectura de Computadoras