Arquitectura: implementación segmentada

95.57/75.03 Organización del computador

Docentes: Patricio Moreno y Adeodato Simó

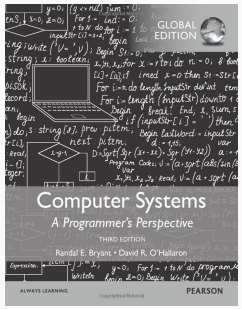
1.er cuatrimestre de 2020

Última modificación: Mon Aug 3 20:06:39 2020 -0300

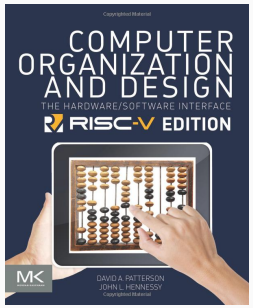
Facultad de Ingeniería (UBA)

Créditos

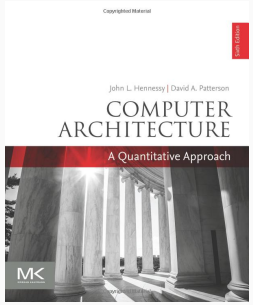
Para armar las presentaciones del curso nos basamos en:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2019.

Tabla de contenidos

Tabla de contenidos

Segmentación: lavado de autos

Secuencial

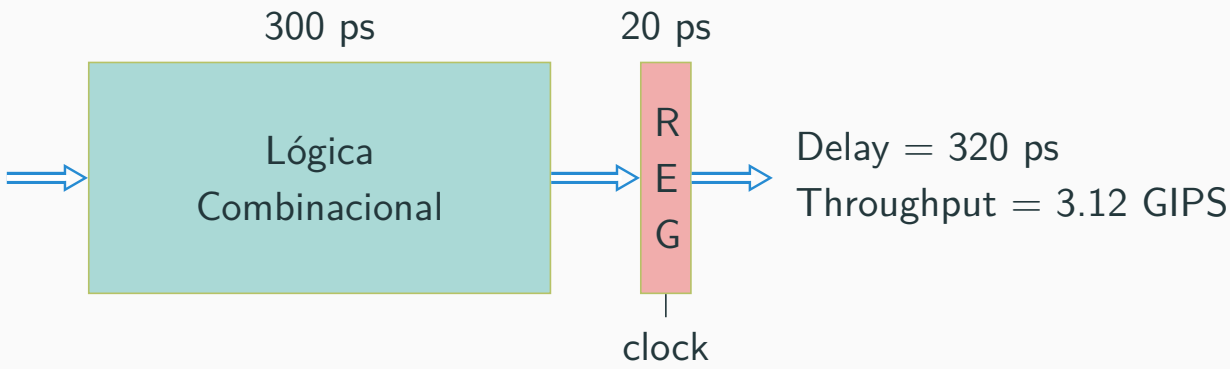
Paralelo

Segmentado

Idea:

- Dividir el proceso en etapas independientes
- Mover objetos a través de las etapas
- En todo momento, múltiples objetos se están procesando

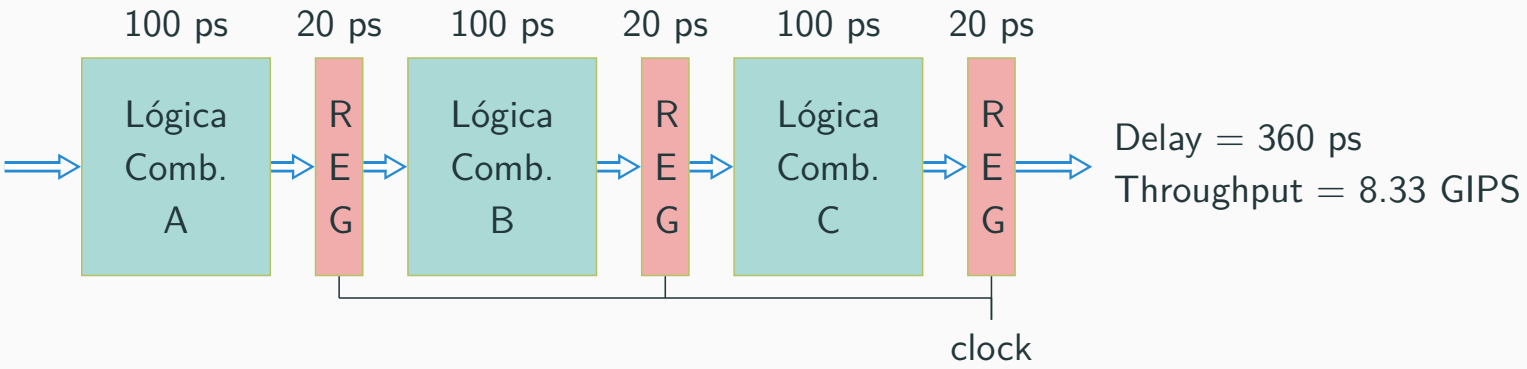
Ejemplo con lógica computacional



Sistema

- El resultado se obtiene en 300 picosegundos
- Requiere 20 ps más para guardar el resultado en el registro
- Necesita un ciclo de clock mínimo de 320 ps

Ejemplo con lógica computacional



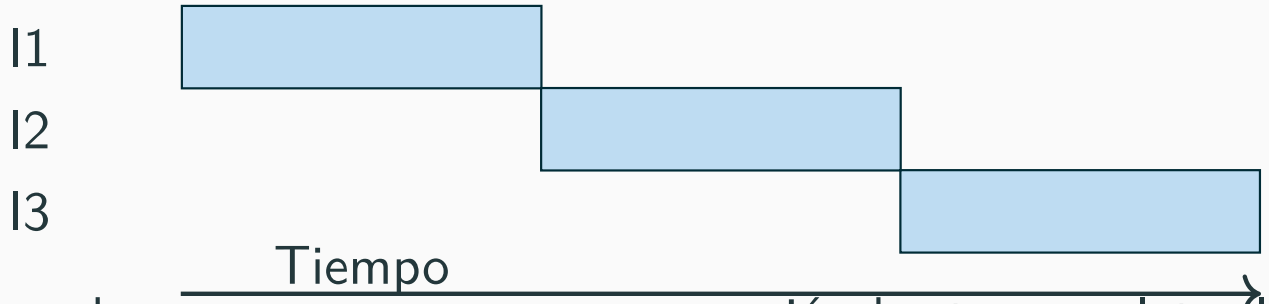
Sistema

- Divide la lógica combinacional en 3 bloques de 100 ps cada uno
- Puede comenzar una nueva operación cuando la anterior supera la etapa A
  - Puede empezar una operación cada 120 ps
- Aumenta la latencia total
  - 360 ps desde el inicio hasta el final (vs. 320 ps)

Tabla de contenidos

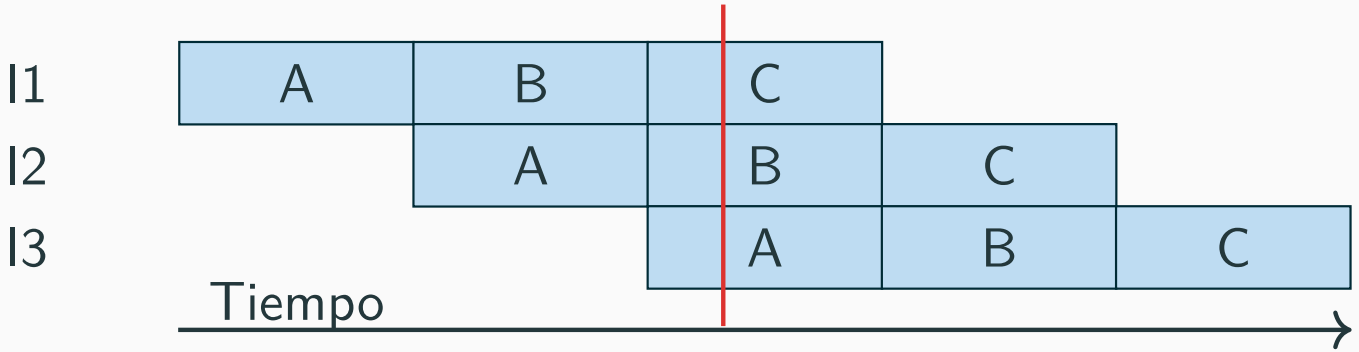
Diagramas de segmentación

Sin segmentar



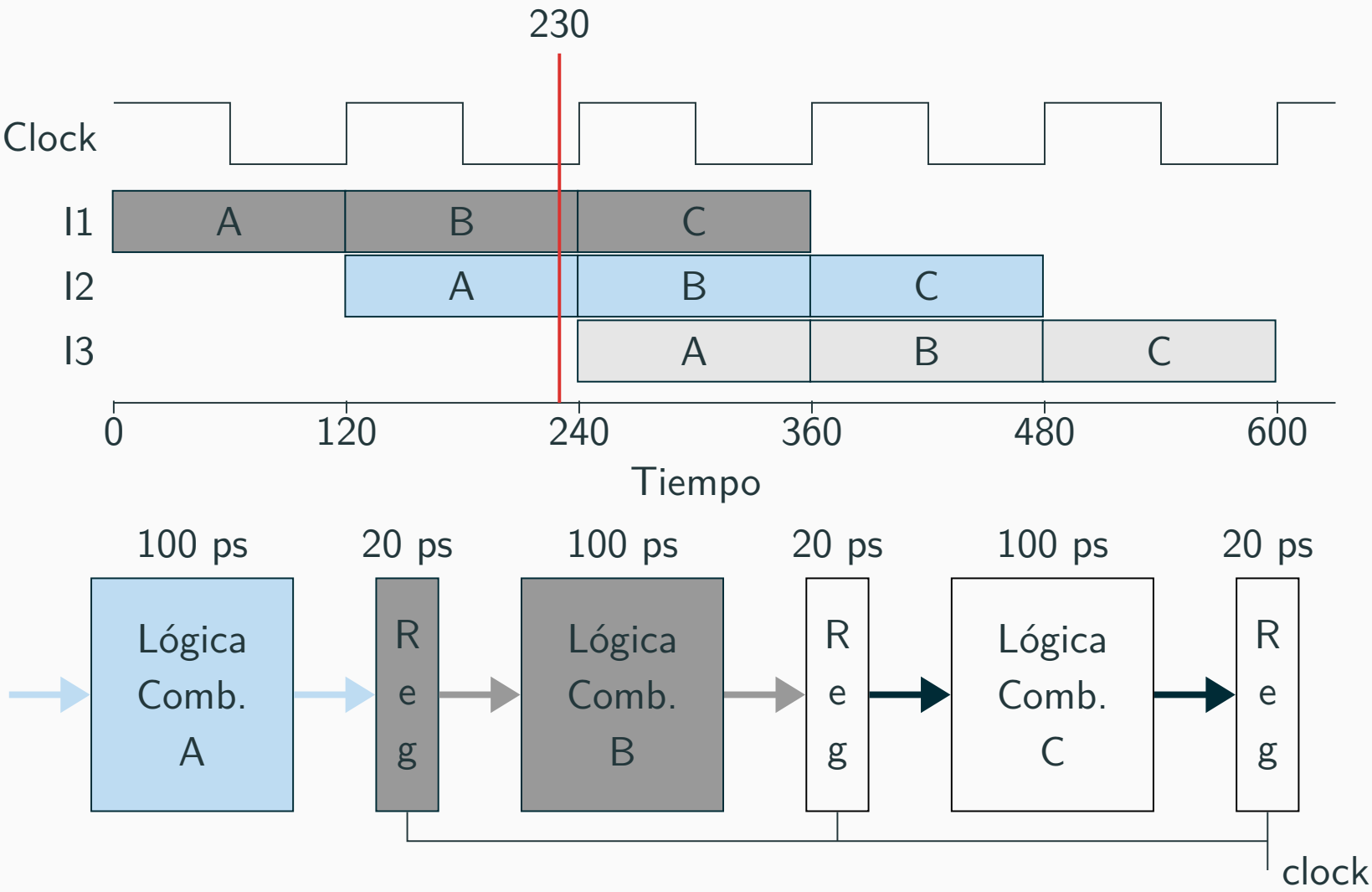
- No puede comenzar una nueva operación hasta completar la anterior

Segmentado (*3-stage pipelined*)

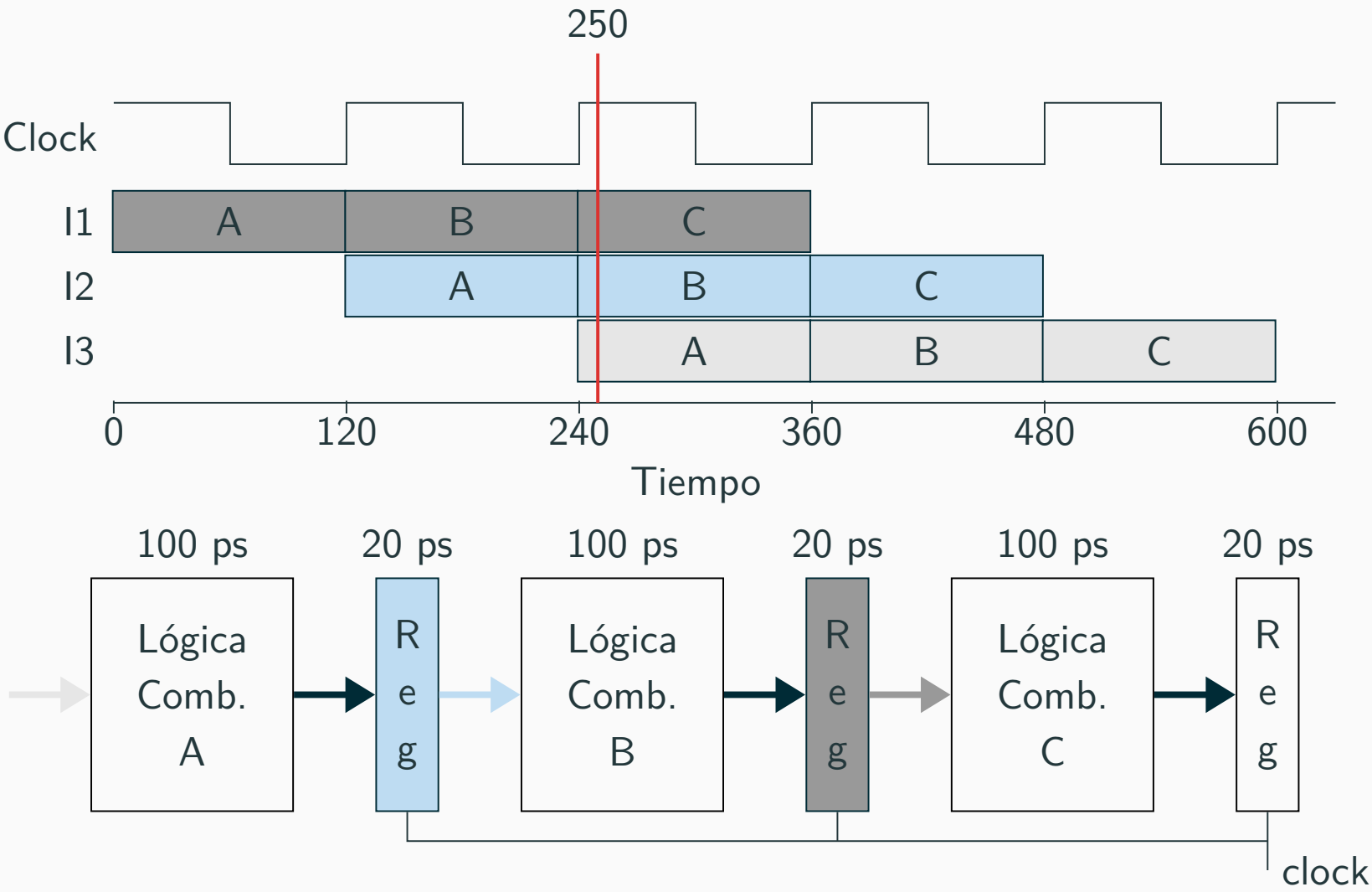


- Hasta 3 operaciones procesándose en simultáneo

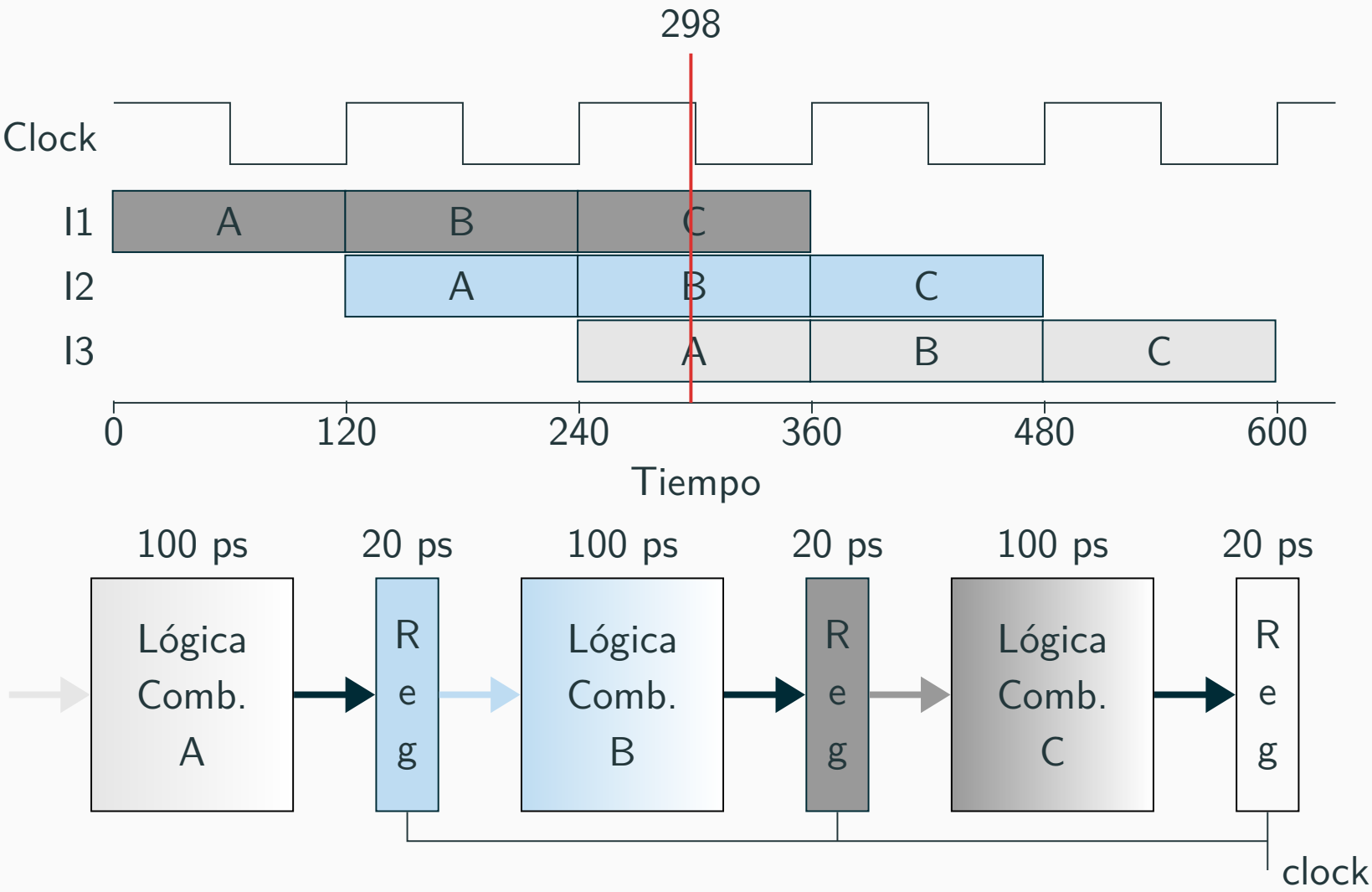
Operación a través del *pipeline*



Operación a través del *pipeline*



Operación a través del *pipeline*



Operación a través del *pipeline*

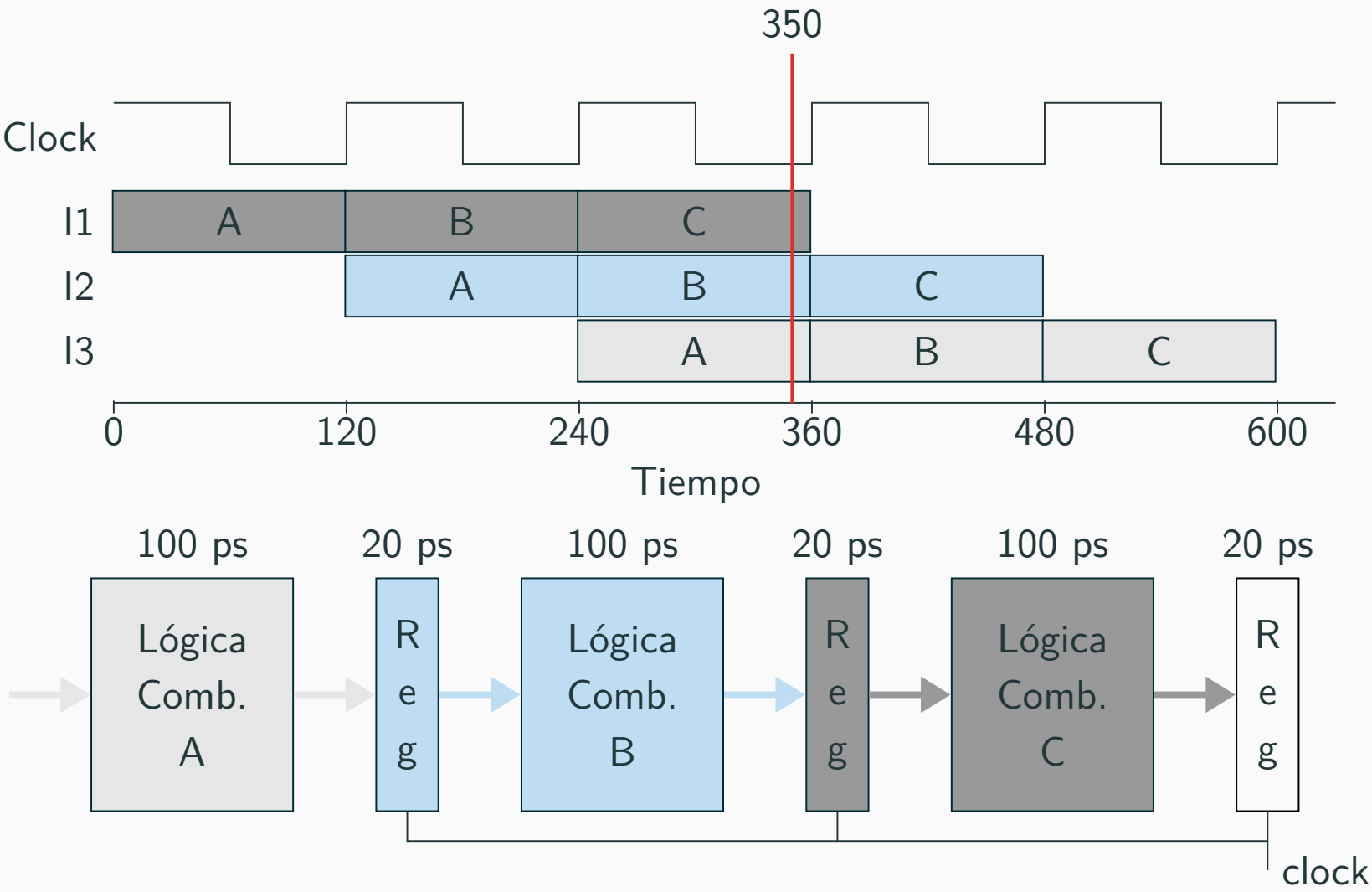
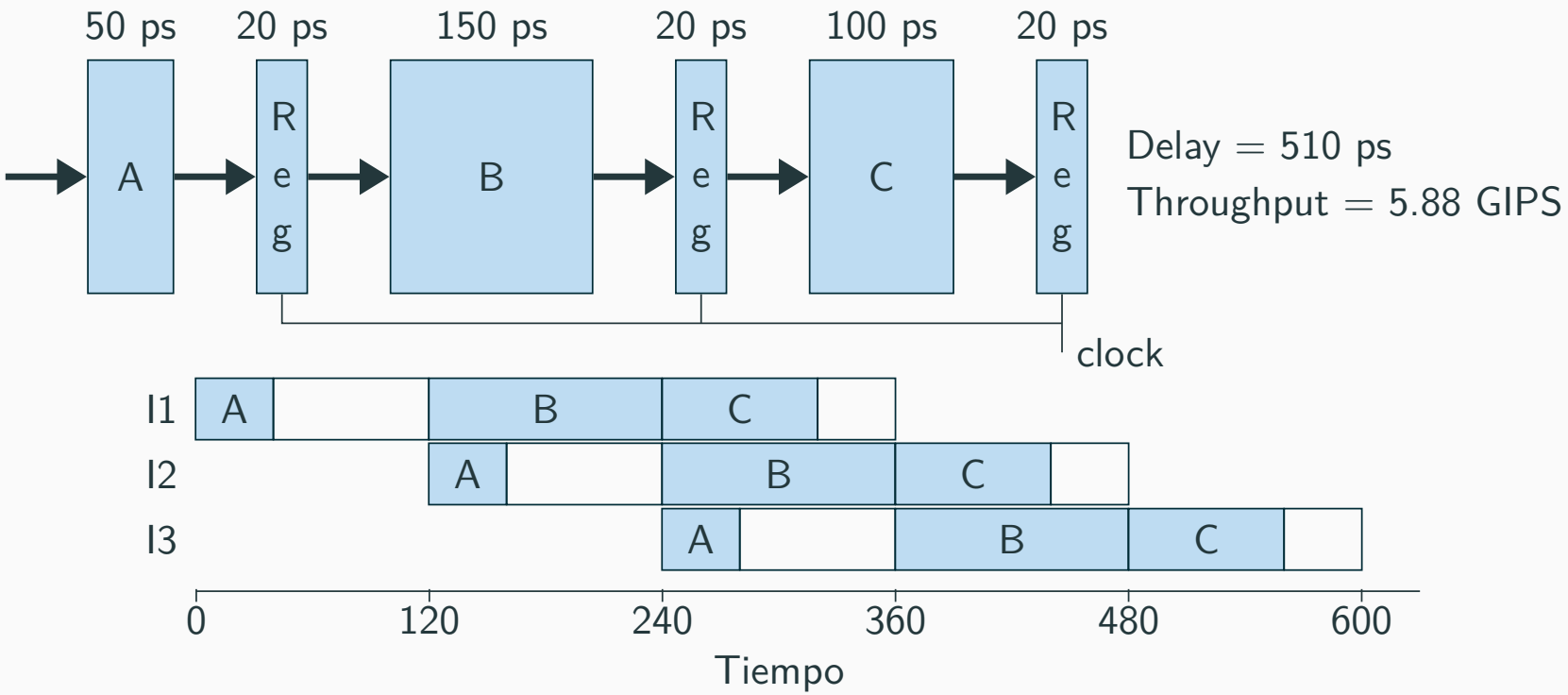


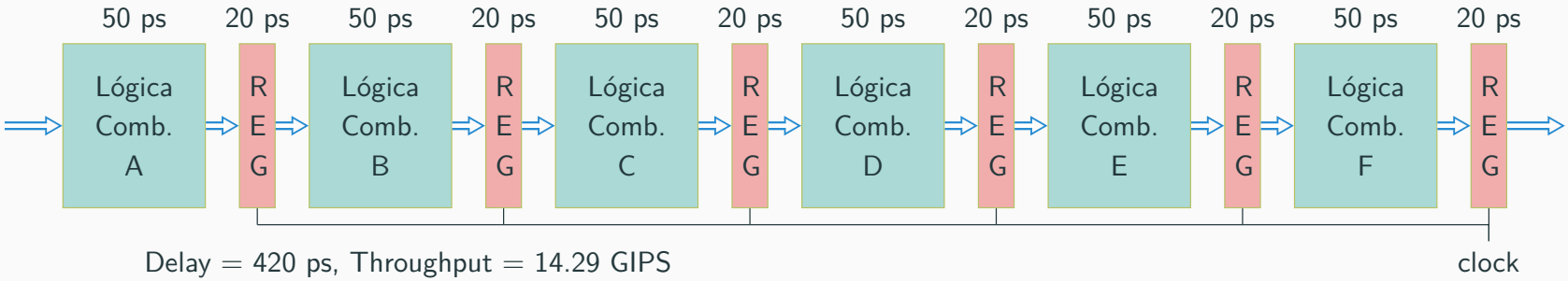
Tabla de contenidos

Limitaciones: delays no uniformes



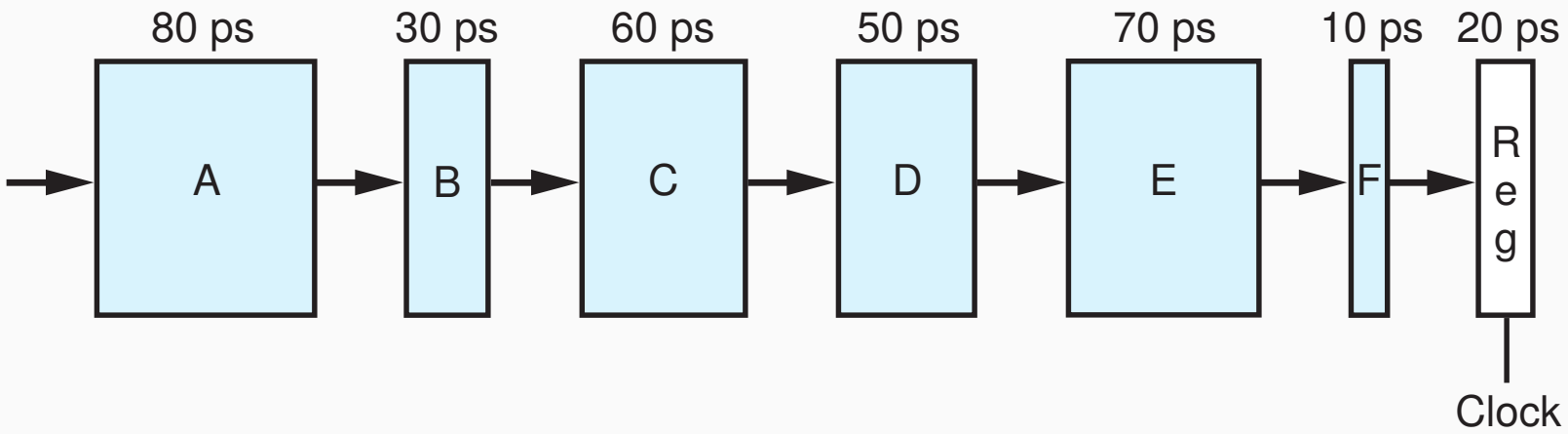
- El throughput es limitado por la etapa más lenta
- Las demás etapas permanecen ociosas
- Desafío: segmentar el sistema en etapas balanceadas

Limitaciones: overhead de los registros



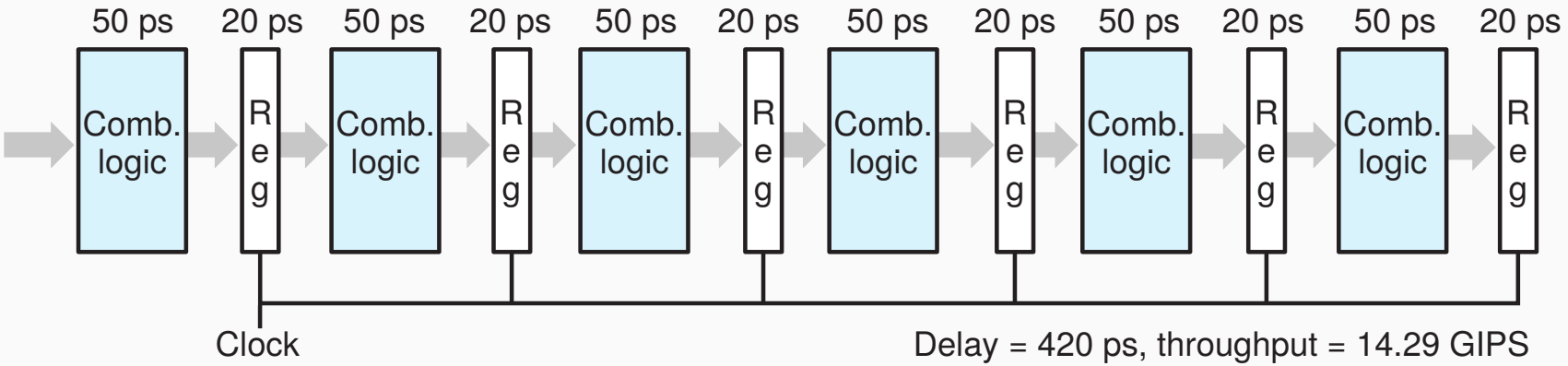
- Cuanto más se segmenta, más pesa la carga de los registros
- Porcentaje del ciclo de clock utilizado en cargar los registros:
  - 1-stage pipeline: 6,25 %
  - 3-stage pipeline: 16,67 %
  - 6-stage pipeline: 28,57 %

Ejercicio 4.28



- 2-stage pipeline ¿throughput? ¿latencia?
- 3-stage pipeline ¿throughput? ¿latencia?
- 4-stage pipeline ¿throughput? ¿latencia?
- ¿cuál es la mejor segmentación? ¿throughput? ¿latencia?

Ejercicio 4.29



- k-stage pipeline ¿throughput? ¿latencia?
- ¿límites?

Tabla de contenidos

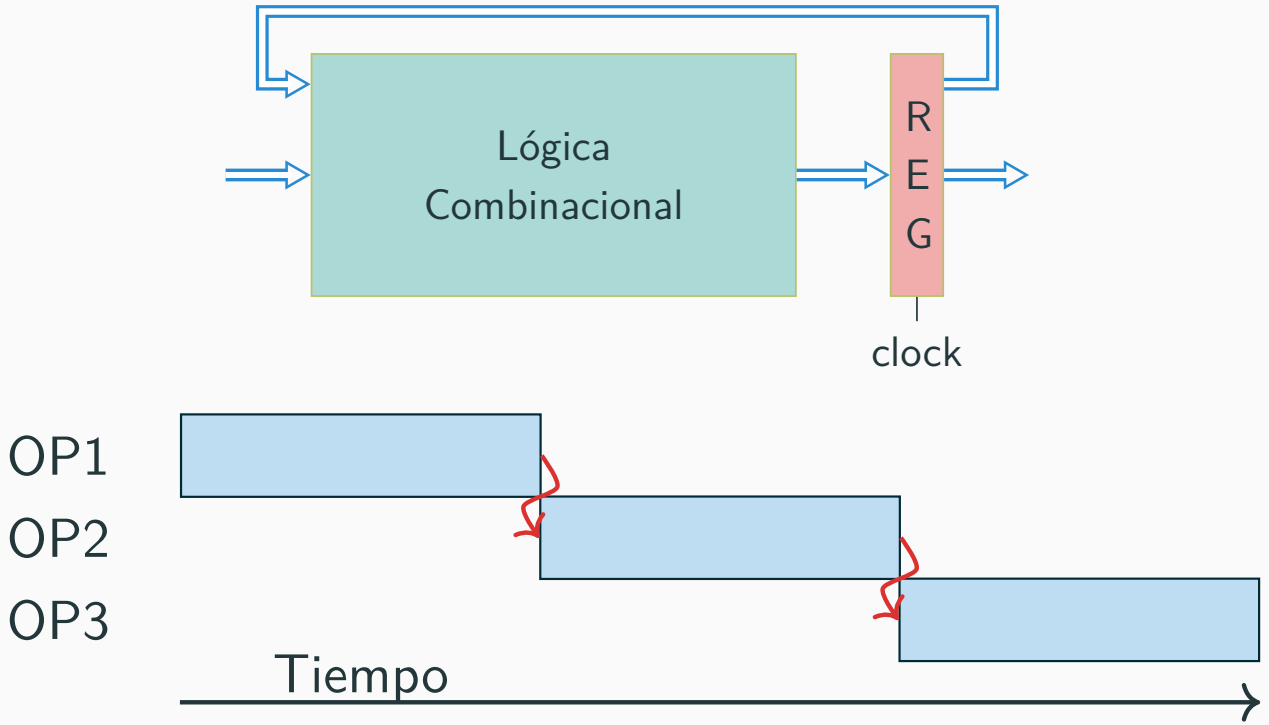
Riesgos de segmentación

Riesgos estructurales surgen de conflictos de los recursos, cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones en ejecuciones solapadas simultáneamente.

Riesgos por dependencia de datos surgen cuando una instrucción depende de los resultados de una instrucción anterior, de forma que, ambas, podrían llegar a ejecutarse de forma solapada.

Riesgos de control (de saltos) surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.

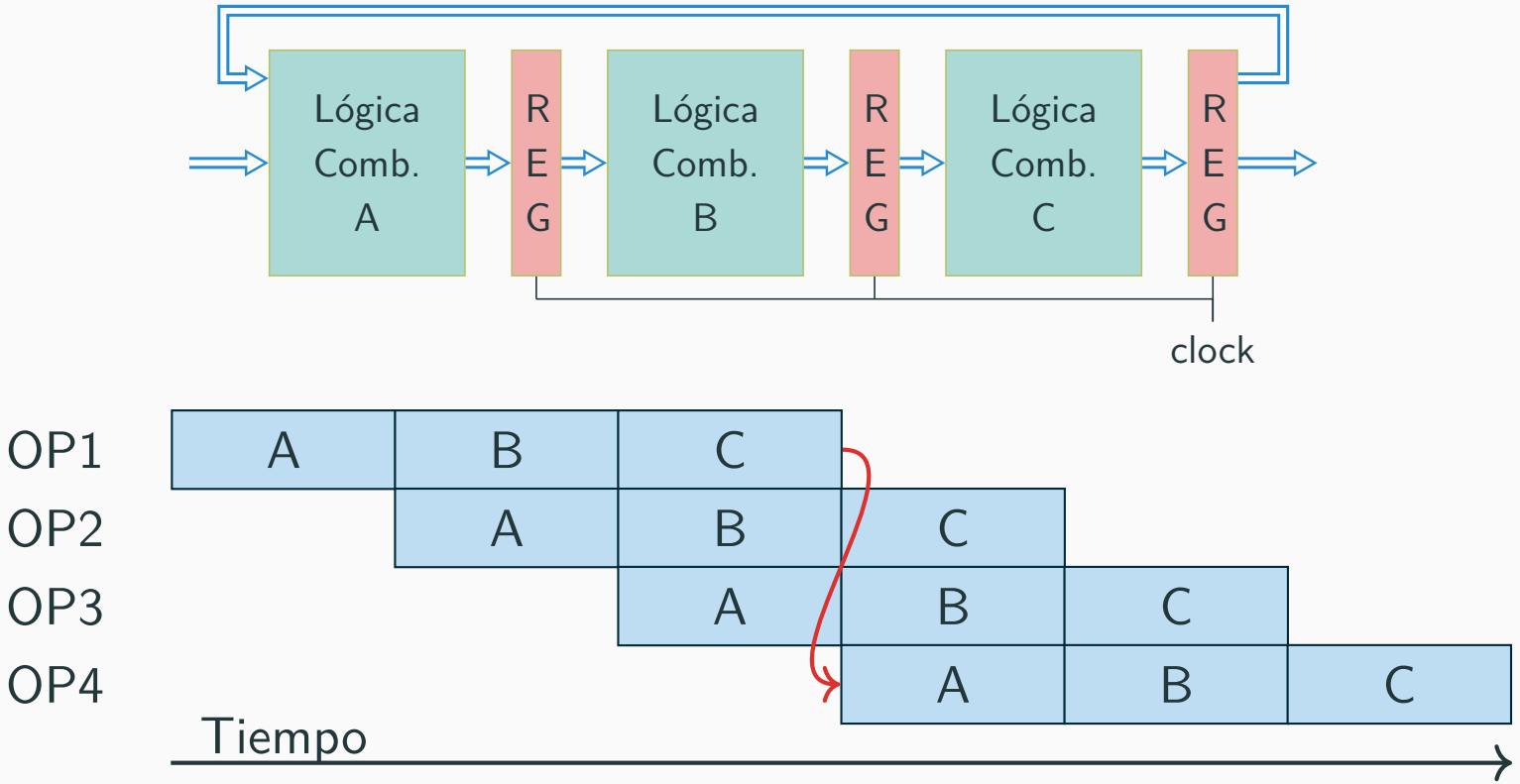
Dependencia de datos



Sistema

- Cada operación depende del resultado de una operación anterior

Riesgos de la dependencia de datos



- El resultado no se realimenta a tiempo
- La segmentación *cambió* el comportamiento del sistema

Dependencias en los procesadores

```
1  irmovq $50, %rax
2  addq   %rax, %rbx
3  mrmovq 100(%rbx), %rdx
```

- El resultado de una operación se usa como operando en la siguiente
  - dependencia **R**ead-**A**fter-**W**rite (RAW)
- *Muy* común (típico)
- Hay que mejarlas bien
  - Obtener el resultado correcto
  - Minimizar el impacto en el desempeño

Dependencias en los procesadores

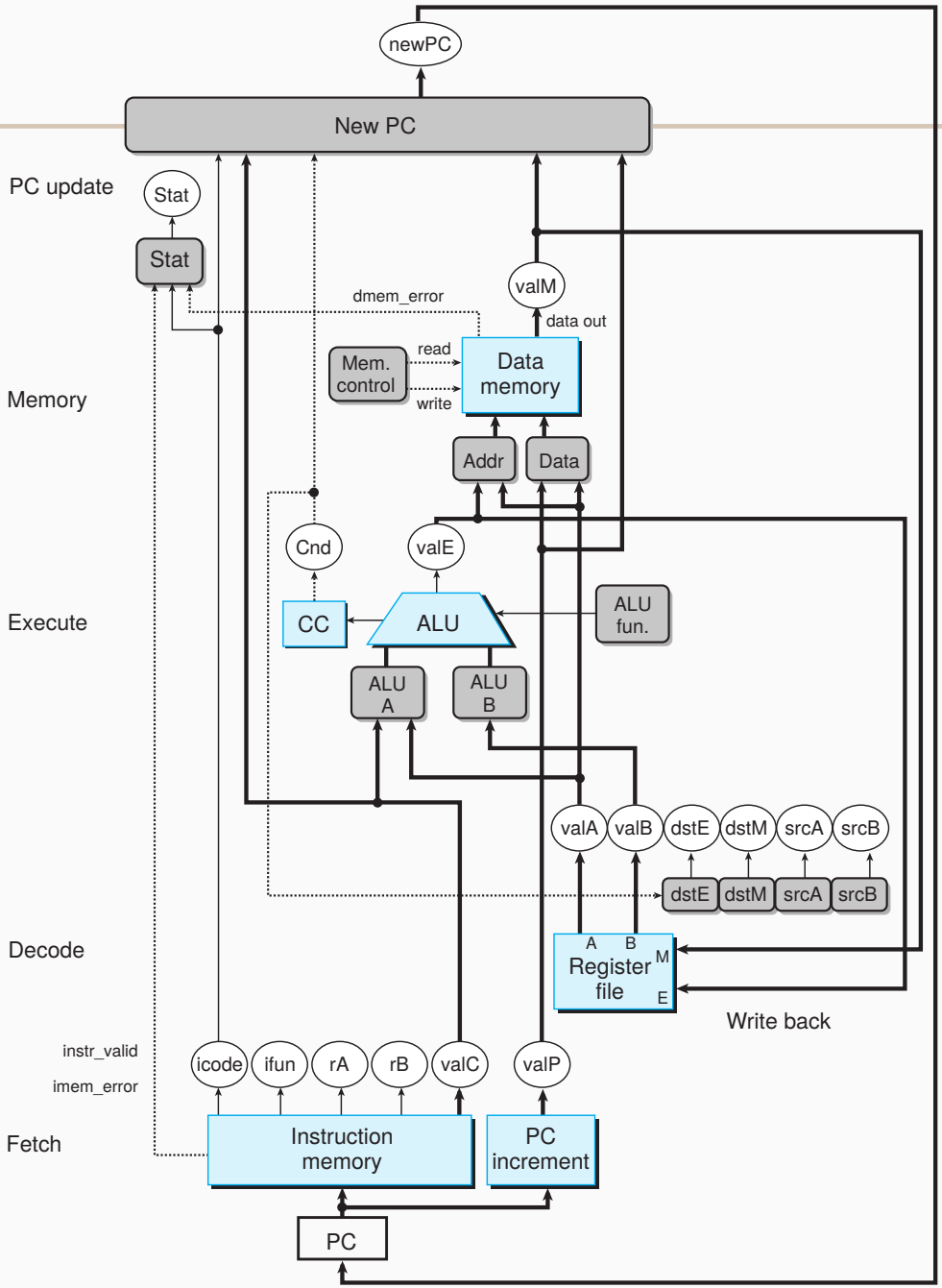
```
1  loop:
2      subq   %rdx, %rbx
3      jne    targ
4      irmovq $10, %rdx
5      jmp    loop
6  targ:
7      halt
```

- La siguiente instrucción a ejecutar depende del resultado de la actual
  - dependencia en el control

Tabla de contenidos

SEQ Hardware

- Las etapas se ejecutan secuencialmente
- Se procesa de a una operación por vez



SEQ+ Hardware

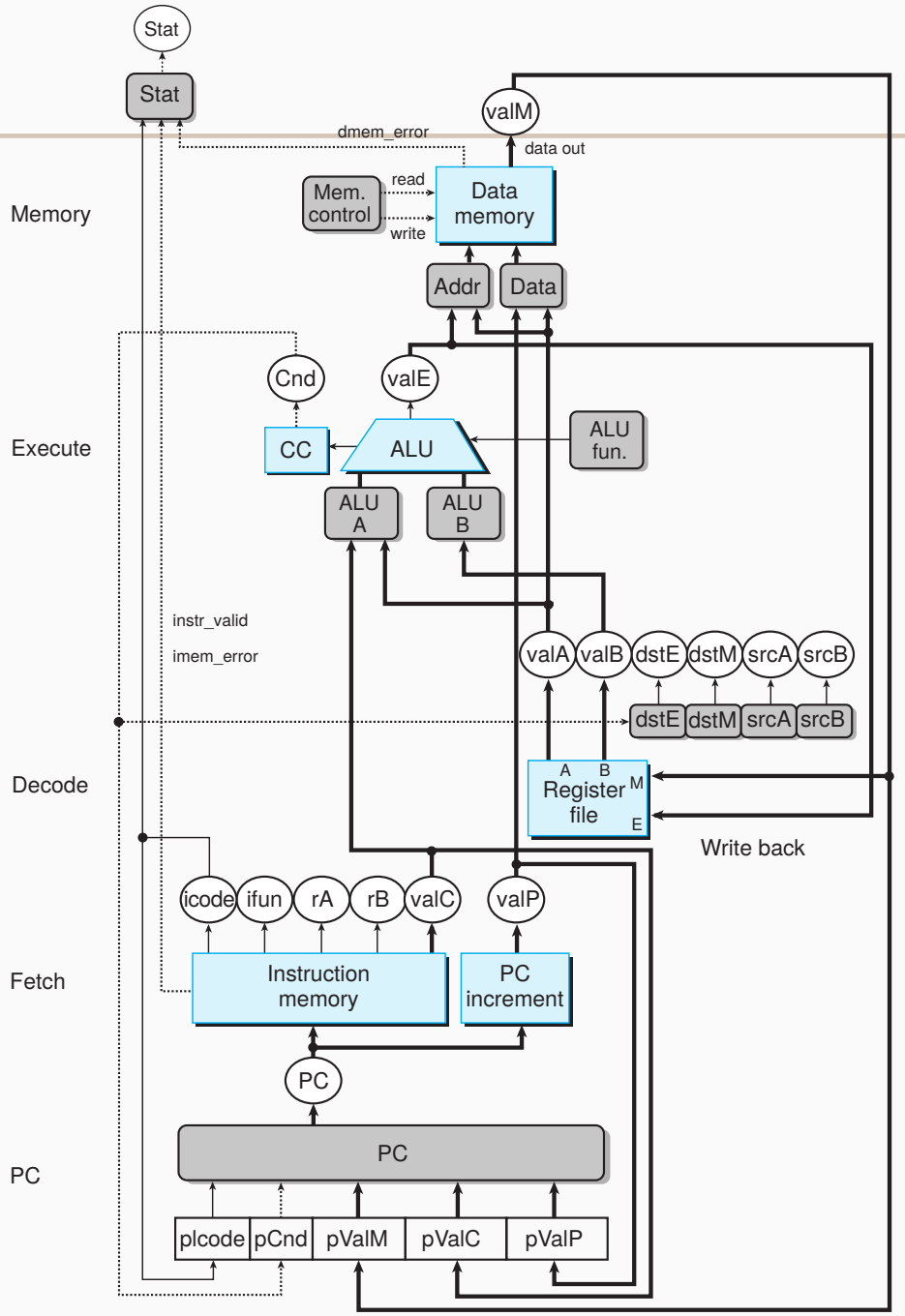
- sigue siendo secuencial
- la etapa del PC es, ahora, la primera

Etapa del PC

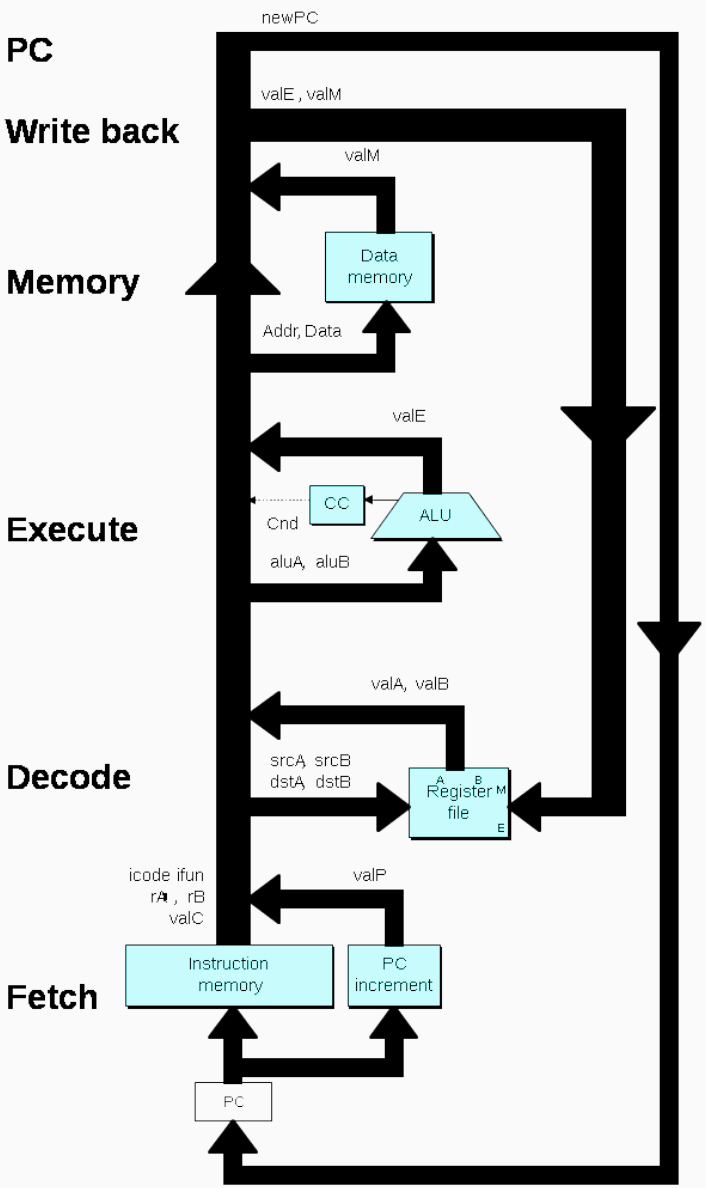
- Debe seleccionar el valor de PC para la instrucción *actual*
- Se basa en los resultados de la instrucción anterior

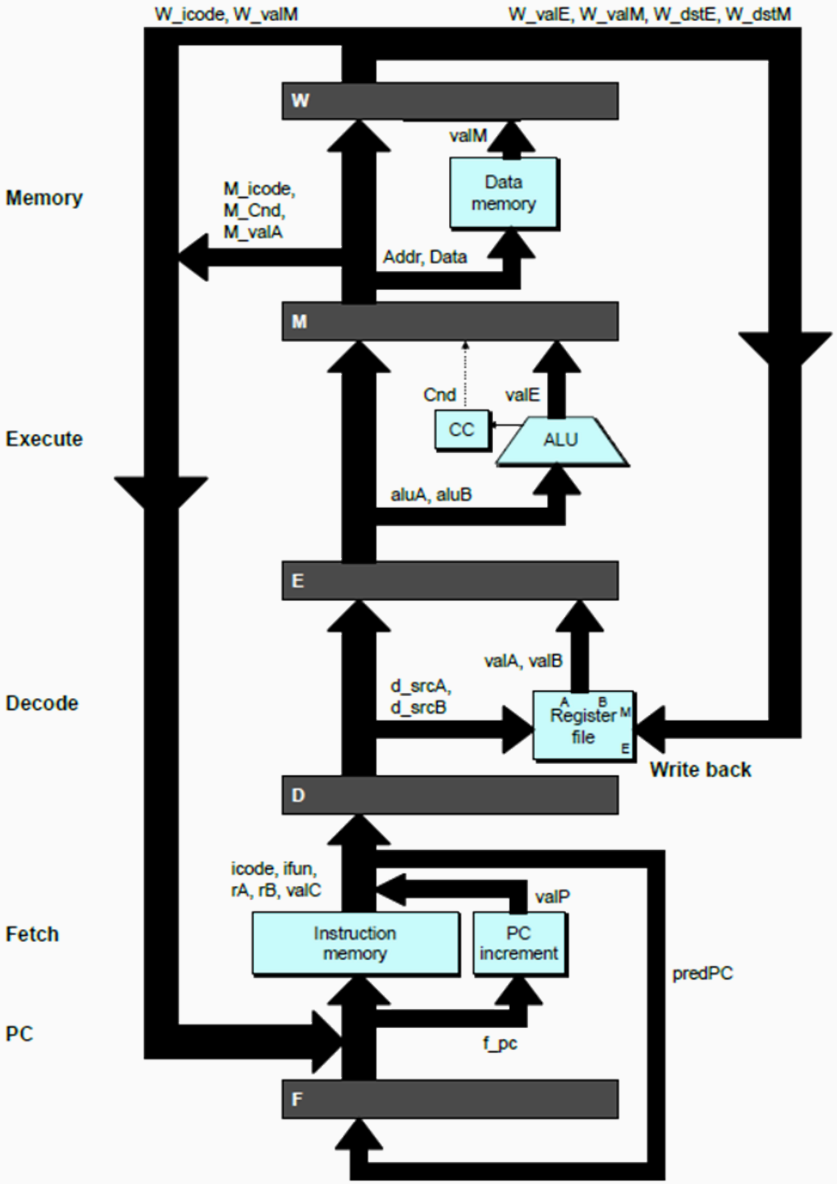
Estado del procesador

- El PC ya no está en un registro
  - Se lo determina en base a información almacenada



Pipeline registers





Etapas segmentadas

Fetch

- Seleccionar el PC actual
- Leer instrucción
- Calcular el PC incrementado

Decode

- Leer registros

Execute

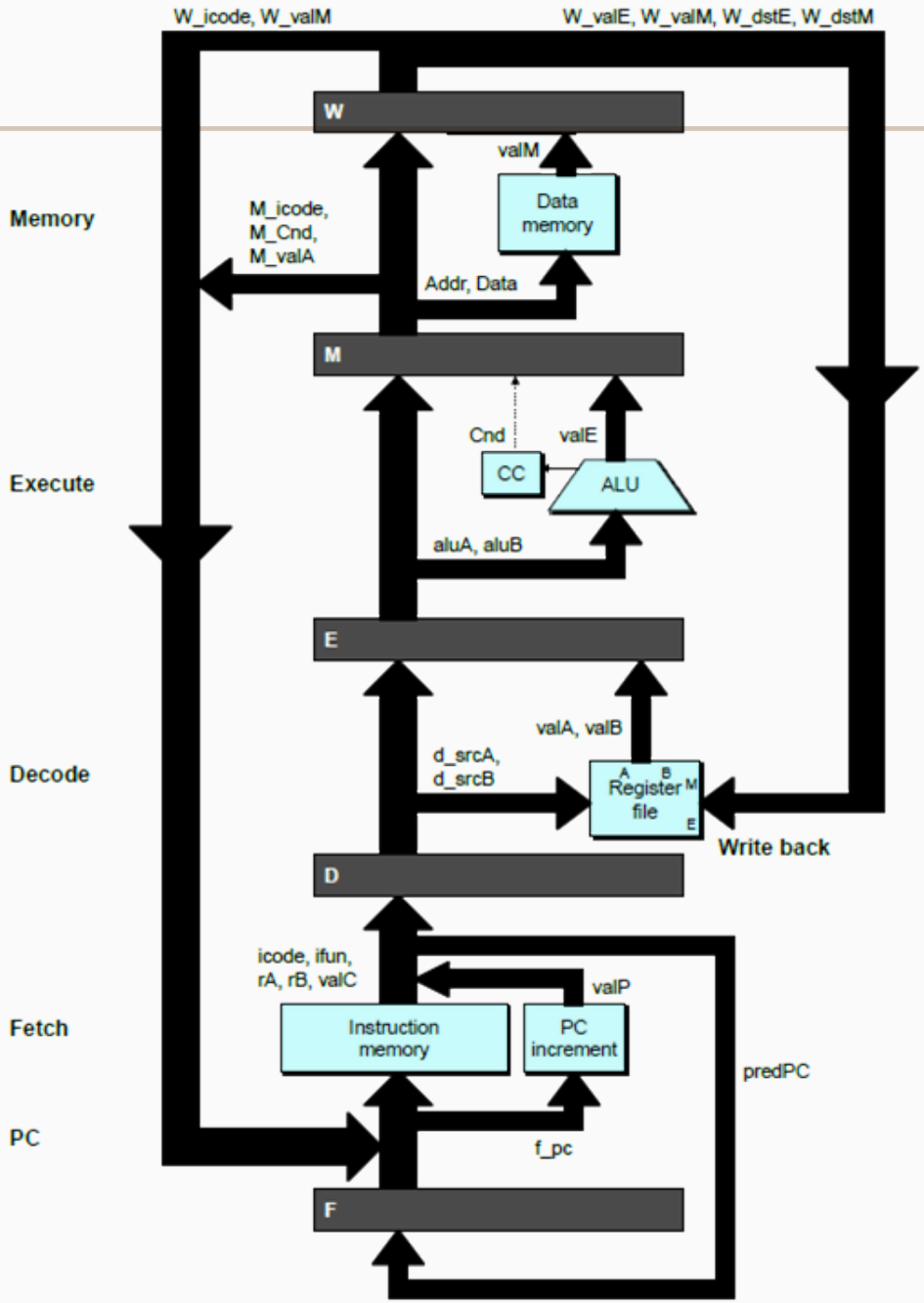
- Usar la ALU

Memory

- Leer/escribir datos de/en la memoria

Write-Back

- Actualizar registros



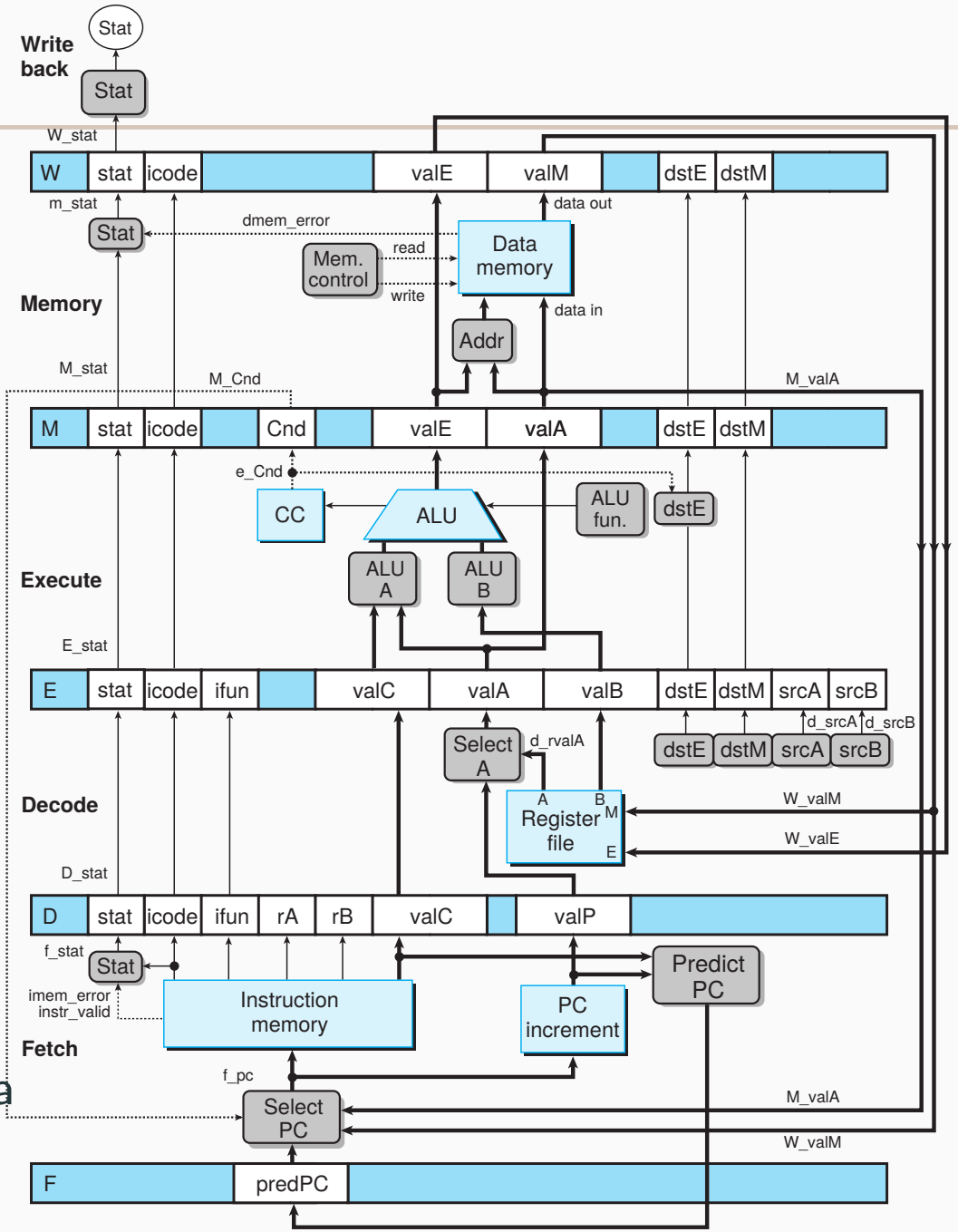
PIPE– Hardware

- Los registros de segmentación (*pipeline register*) guardan valores intermedios
- Las instrucciones avanzan en sentido ascendente en el dibujo

Nombres de las señales

S_field valor del campo **almacenado** en el registro de segmentación en la etapa S

s_field valor del campo **calculado** en la etapa S



Realimentación

Predicted PC

- Valor supuesto para el siguiente PC

Información de saltos

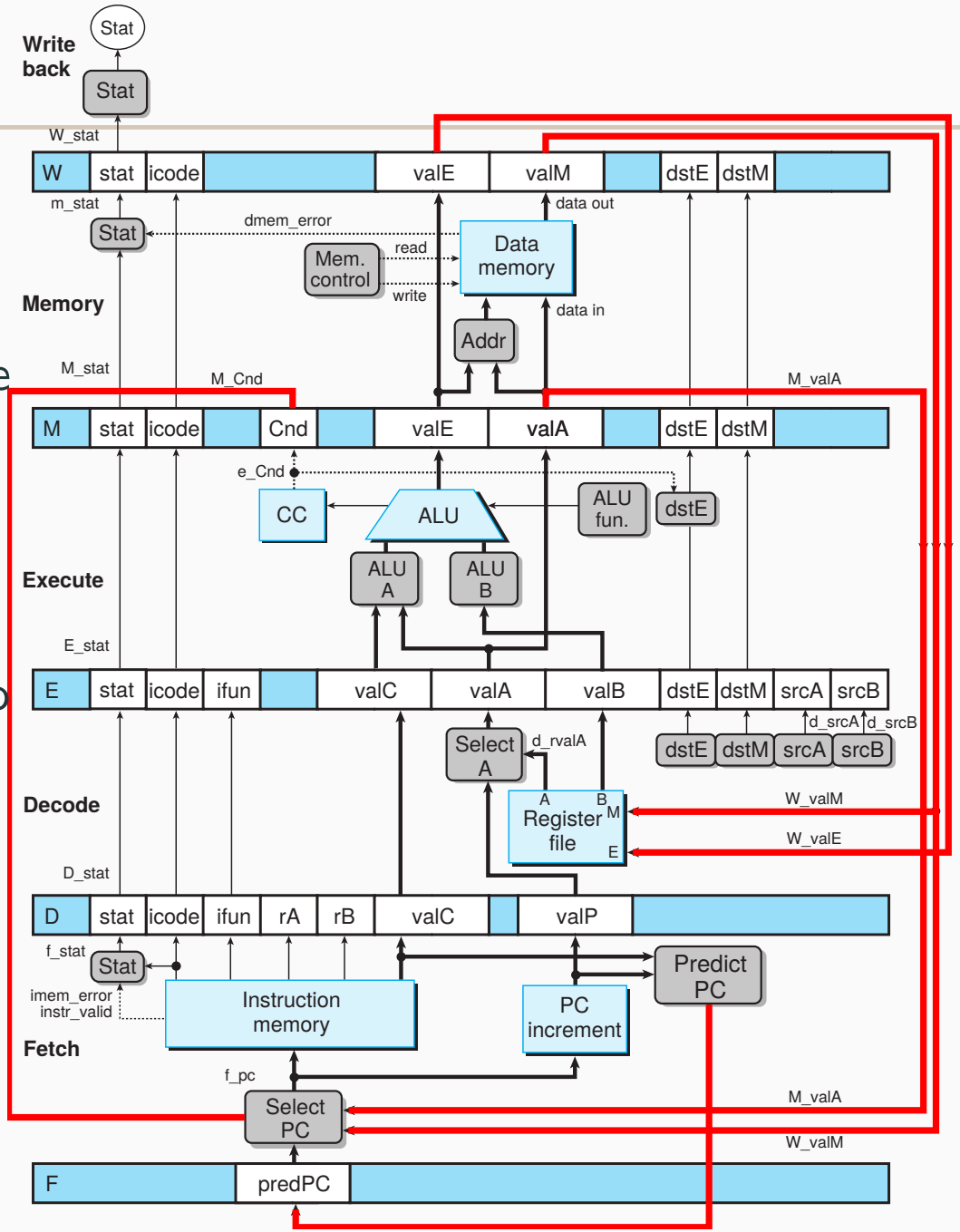
- Salto tomando o no
- *Fall-through* o dirección destino

Punto de retorno

- Leer de memoria

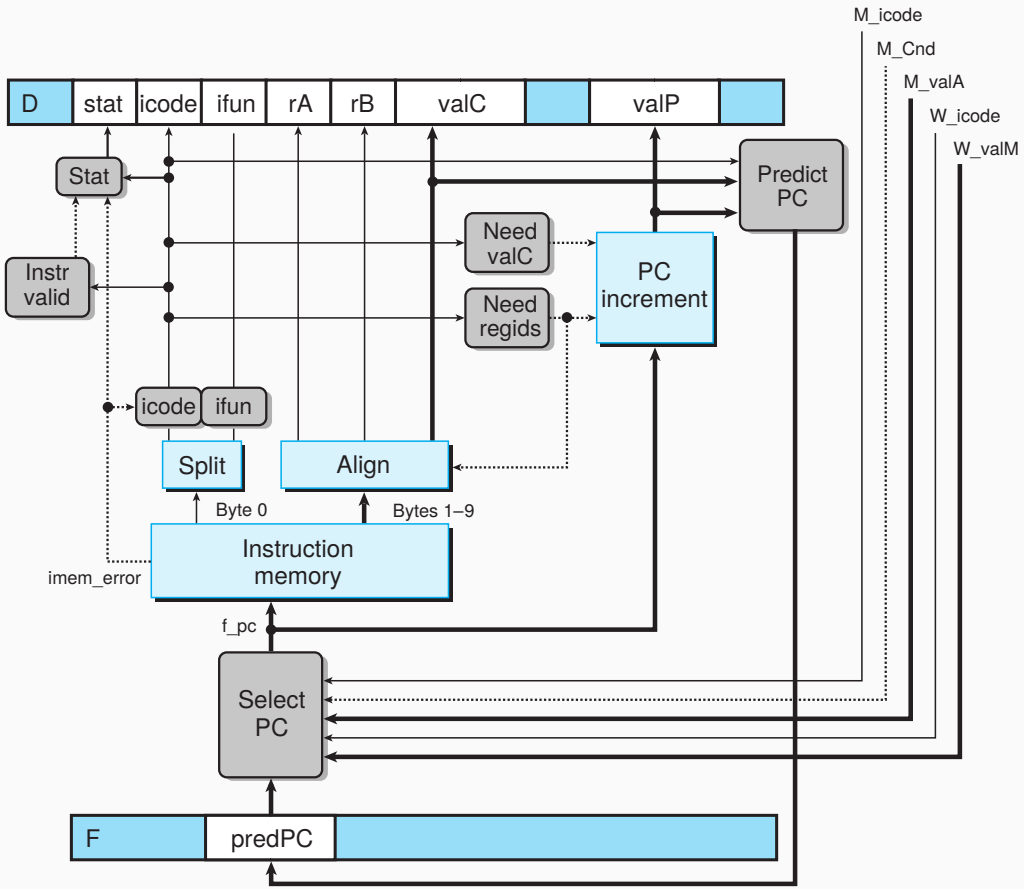
Actualización de registros

- A los puertos de entrada del archivo de registros



Predicción del PC

- Comenzar la lectura de la siguiente instrucción una vez que termina la etapa *fetch* de la actual
  - No se puede determinar a ciencia cierta la siguiente instrucción
- Adivinar qué instrucción va a seguir
  - Y recuperarse correctamente si la predicción fue incorrecta



Estrategia

Instrucciones que no hacen transferencia de control

- Predicción: el siguiente PC es el valor de valP
- No falla

Call y saltos incondicionales

- Predicción: el siguiente PC es el valor de valC
- Siempre es fiable

Saltos condicionales

- Predicción: el siguiente PC es el valor de valC
- Sólo es correcto si se realiza el salto (se cambia de rama)
  - Típicamente es correcto el 60 % de las veces (hay mejores estrategias)

Instrucción de retorno (`ret`)

- No intentar predecir

Fallas en la predicción

- **Salto incorrecto**
  - El *flag* que indica si se debía realizar el salto se obtiene cuando la instrucción llega a la etapa de memoria
  - Se puede obtener el valor correcto del PC (la dirección correcta donde saltar) del valor de valA (M_valA en la versión segmentada)
- **Instrucción de retorno (`ret`)**
  - No hubo predicción
  - El valor de retorno del PC se obtiene cuando la instrucción `ret` llega a la etapa de memoria

Resumen de PIPE–

Concepto

- Separar la ejecución de la instrucción en etapas (5 en este caso)
- Ejecutar las instrucciones en pipeline

Limitaciones

- No opera correctamente cuando hay dependencia entre instrucciones que se hallan muy cercanas
- *Dependencia de datos*
  - Una instrucción escribe en un registro, luego otra instrucción lo lee (Read-After-Write, RAW)
- *Dependencia en el control*
  - Una instrucción establece el PC de forma que la pipeline no predice correctamente
  - Predicción incorrecta de una rama y retorno de la instrucción

Implementación PIPE

Riesgos de datos

- Una instrucción que tiene al registro R como destino es seguida por una instrucción que tiene al registro R como origen.
- Es una condición común
- No queremos reducir el *throughput* de instrucciones

Riesgos de control (de saltos)

- Predicción incorrecta de un salto condicional
  - Predice que siempre se salta
  - Siempre se ejecutan 2 instrucciones de más
- Obtención de la dirección de retorno para la instrucción `ret`
  - Siempre se ejecutan 3 instrucciones de más

Pipeline stages

Fetch

- Seleccionar el PC actual
- Leer instrucción
- Calcular el PC incrementado

Decode

- Leer registros

Execute

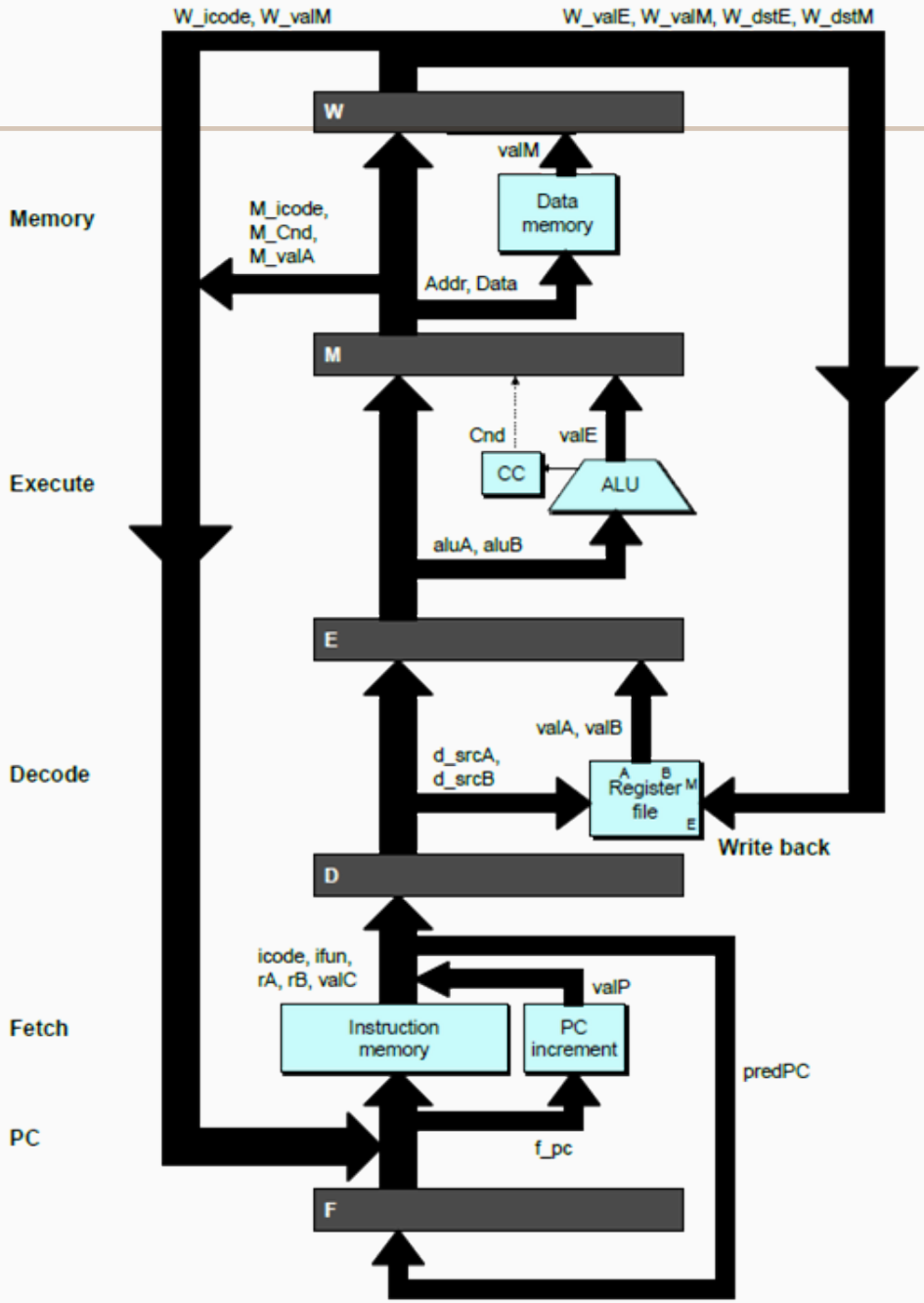
- Usar la ALU

Memory

- Leer/escribir datos de/en la memoria

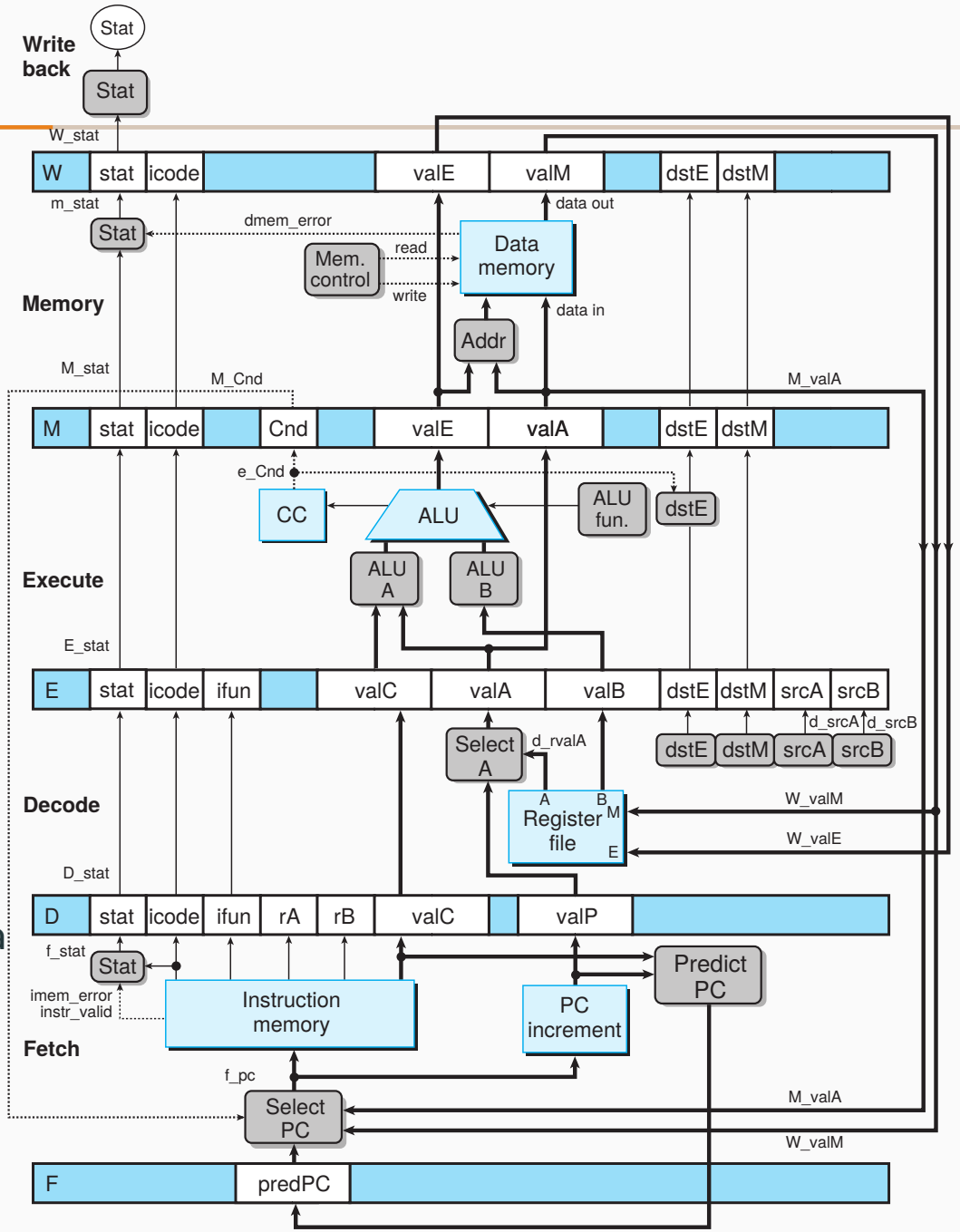
Write-Back

- Actualizar registros



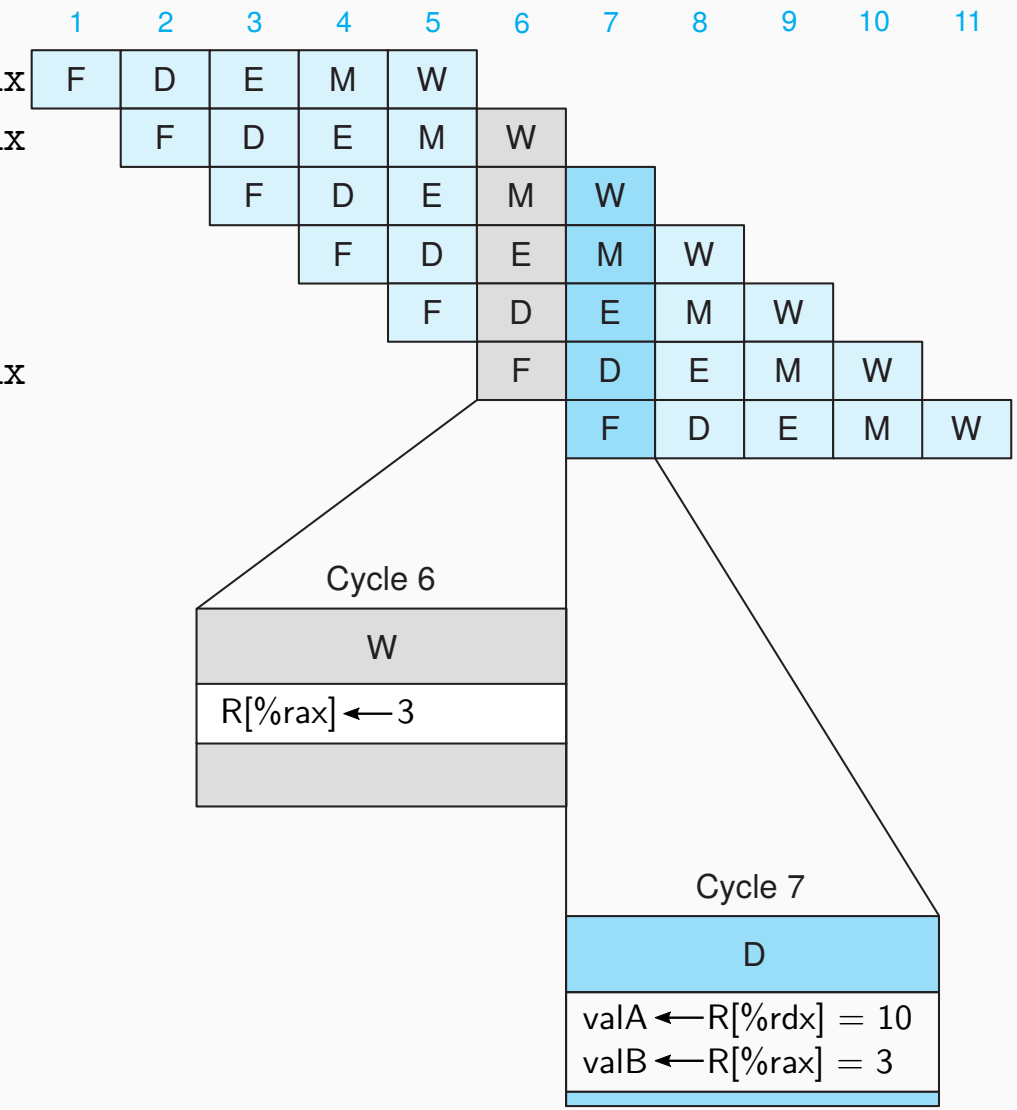
PIPE– Hardware

- Los registros de segmentación (*pipeline register*) guardan valores intermedios
- Las instrucciones avanzan en sentido ascendente en el dibujo
- Los valores se pasan de una etapa a la siguiente
- No se pueden saltear etapas
  - Por ejemplo, `valC` no se necesita en la etapa *decode* pero dicho valor se almacena en el registro, no pasa directamente a la etapa *execute*



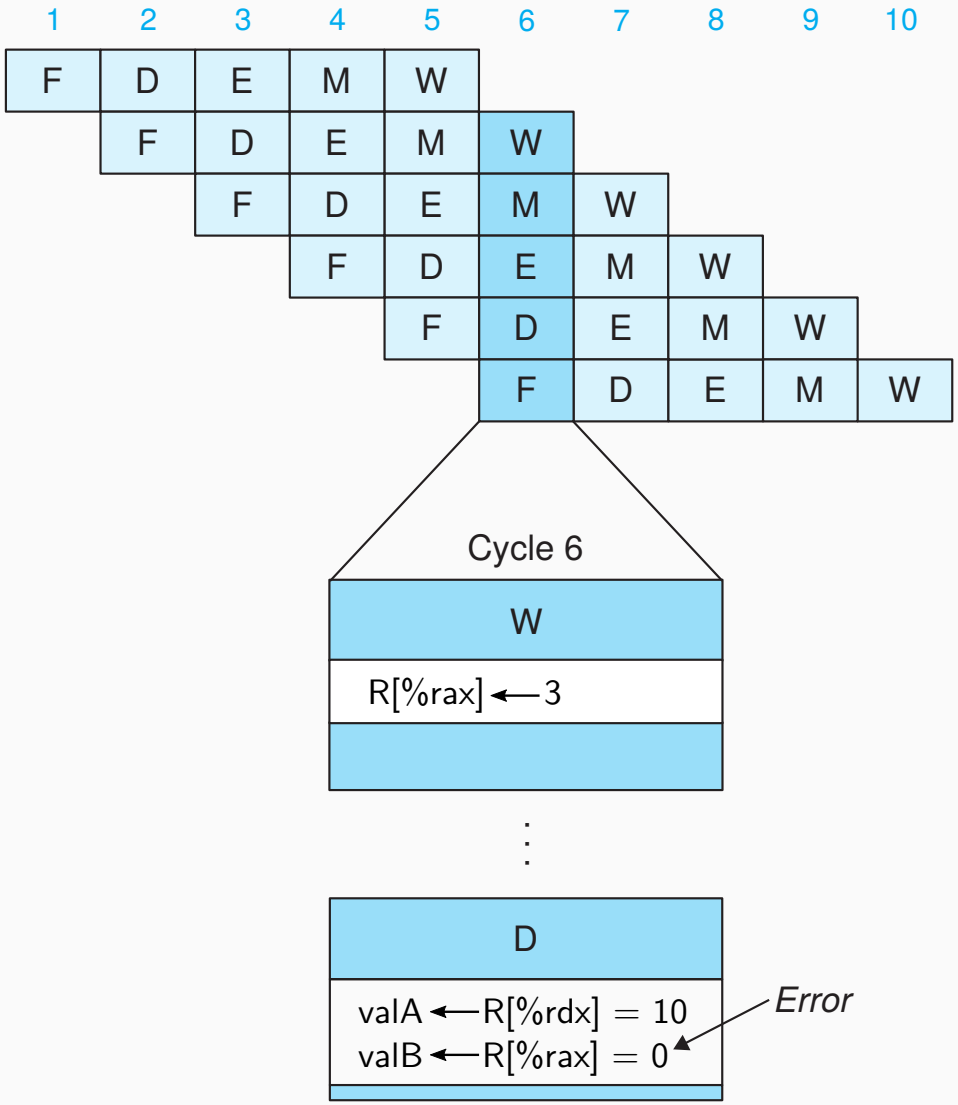
Dependencia de datos: 3 nops

```
#prog1
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq   %rdx, %rax
0x019: halt
```



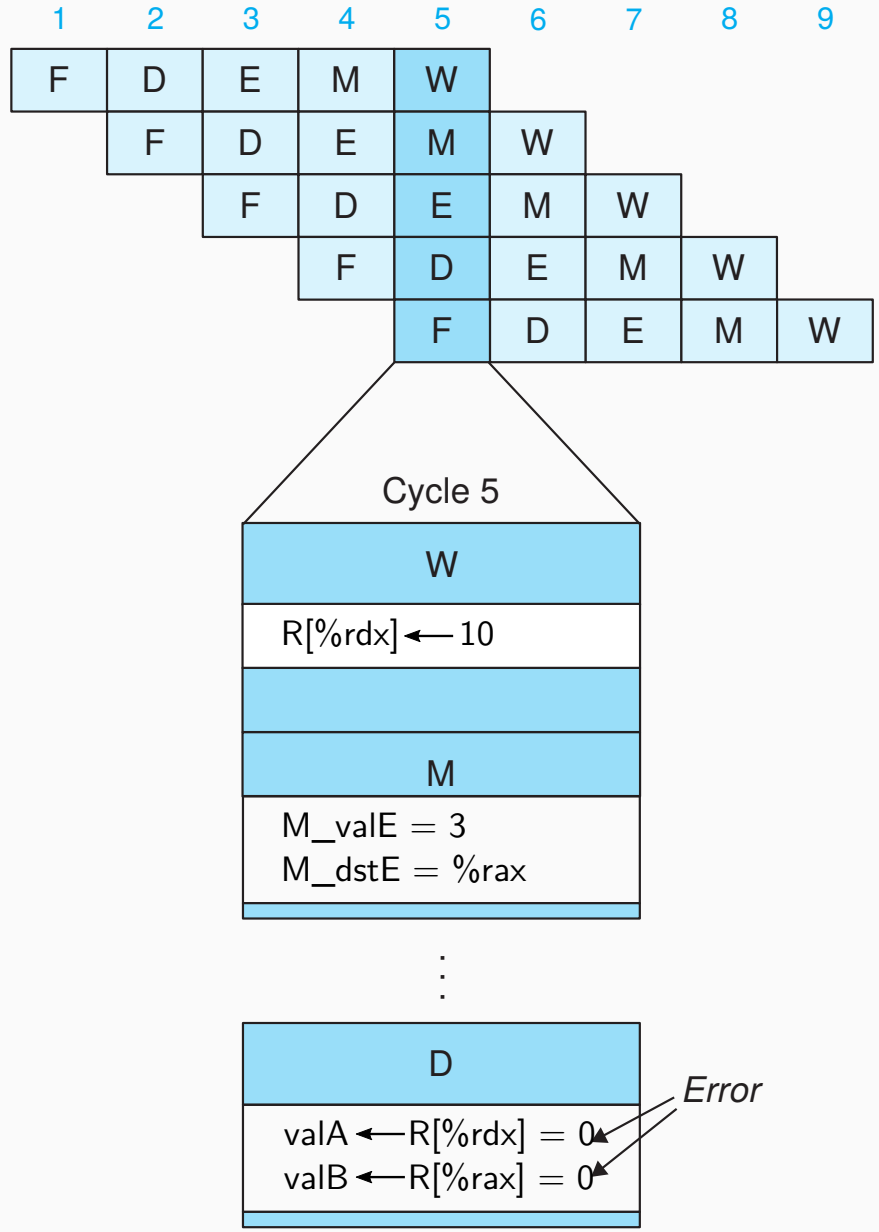
Dependencia de datos: 2 nops

```
#prog2
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
0x014: nop
0x015: nop
0x016: addq   %rdx, %rax
0x018: halt
```



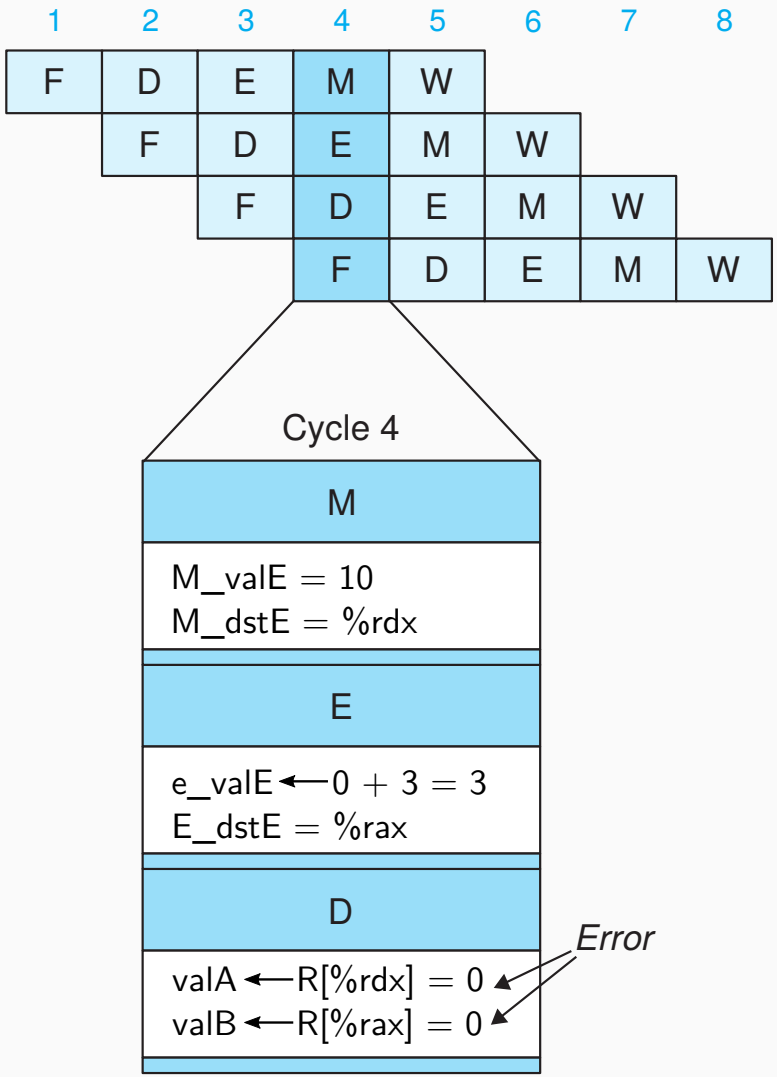
Dependencia de datos: 1 nops

```
#prog3
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
0x014: nop
0x015: addq   %rdx, %rax
0x017: halt
```



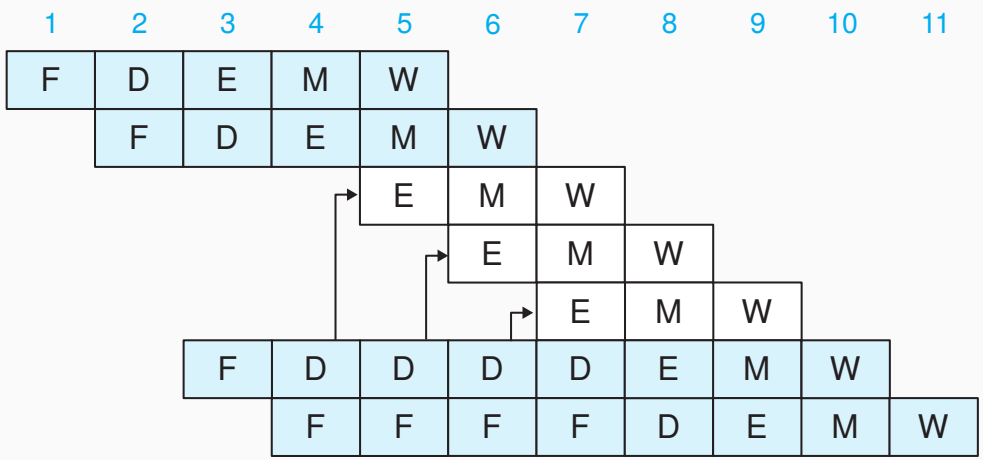
Dependencia de datos: 0 nops

```
#prog4
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq   %rdx, %rax
0x016: halt
```



Dependencia de datos: inserción de burbujas

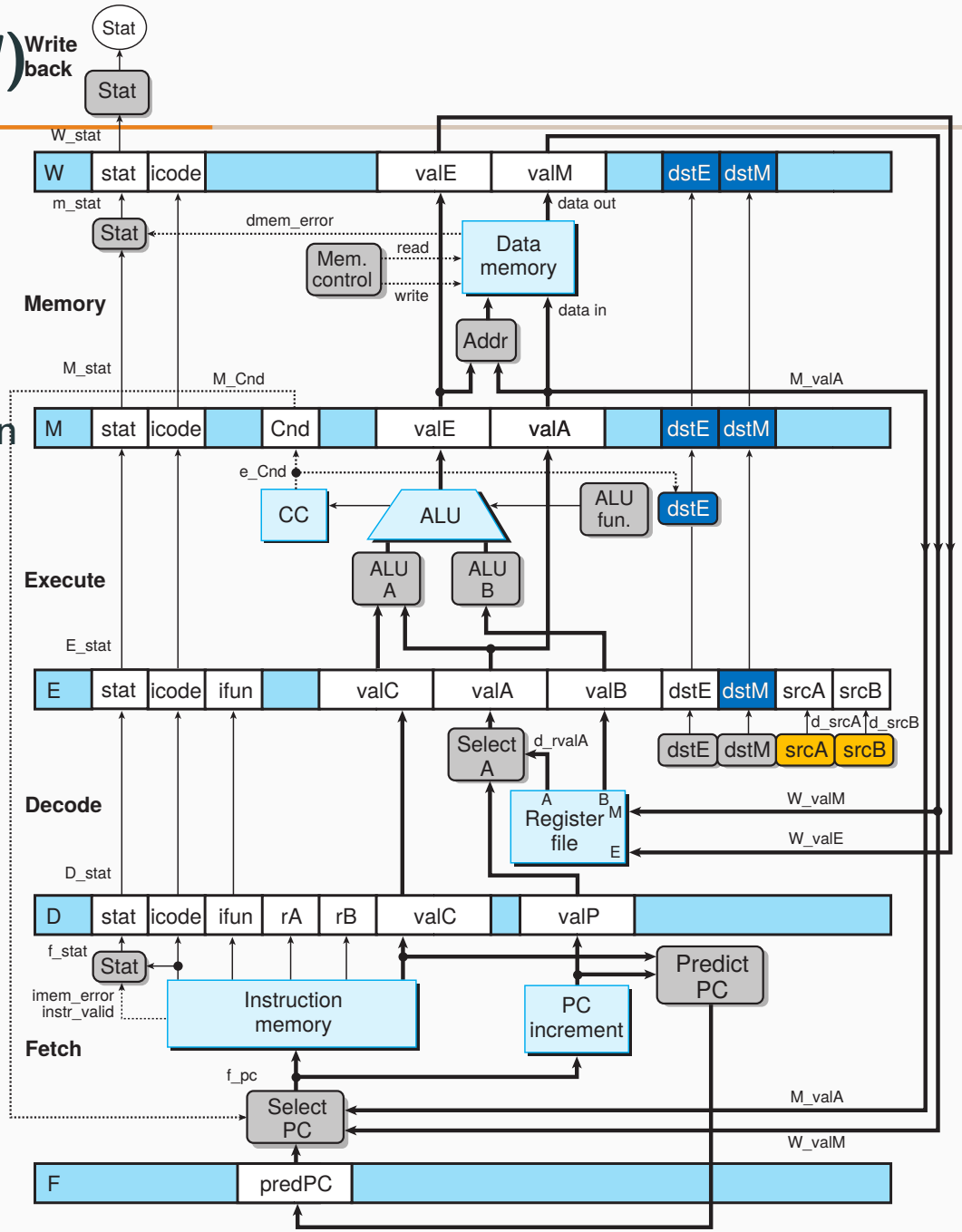
```
#prog4
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
       bubble
       bubble
       bubble
0x014: addq    %rdx, %rax
0x016: halt
```



- Si una instrucción genera un riesgo de datos, se la demora un poco
- Al frenarla, se la mantiene en la etapa decode
- Dinámicamente se insertan instrucciones `nop` a la etapa de ejecución

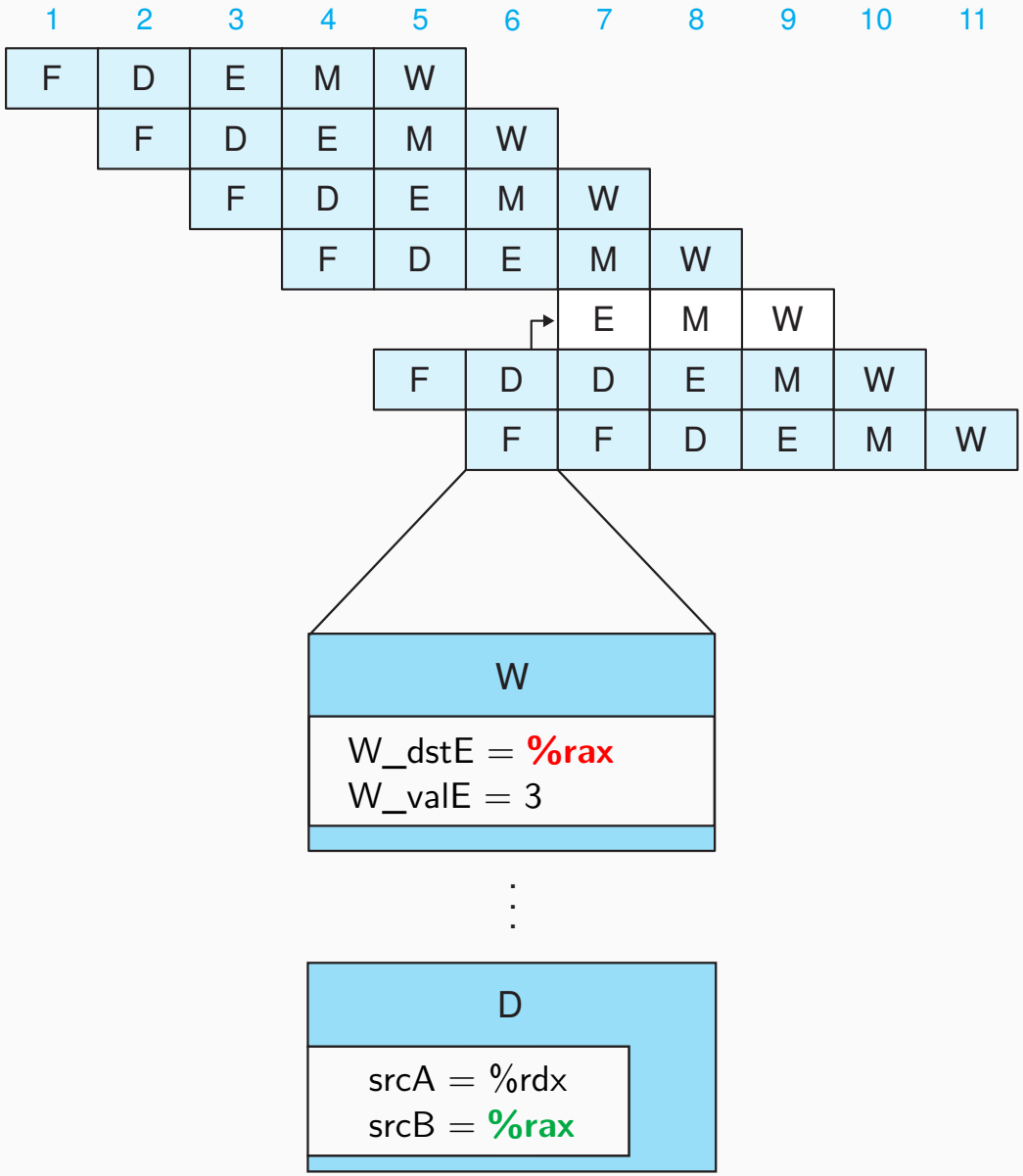
Condiciones de demora (*stall*)

- **Registro de origen**
  - `srcA` y `srcB` de la instrucción en la etapa decode
- **Registro de destino**
  - Campos `dstE` y `dstM`
  - Instrucciones en las etapas `execute`, `memory`, y `write-back`
- **Caso especial**
  - No realizar ninguna demora para el registro 15 (`0xF`)



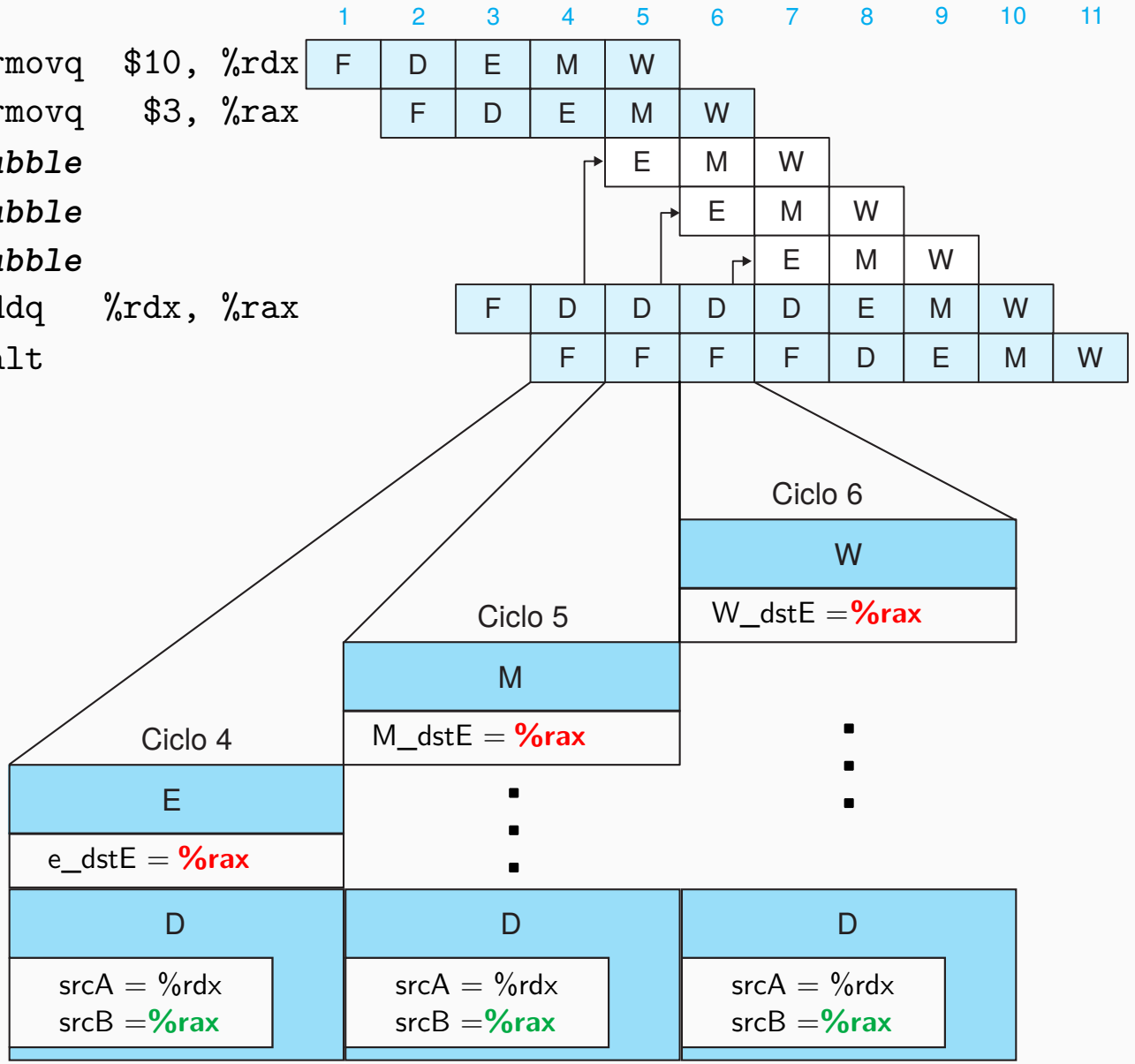
Detección de la condición de demora

```
#prog2
0x000: irmovq $10, %rdx
0x00a: irmovq  $3, %rax
0x014: nop
0x015: nop
       bubble
0x016: addq  %rdx, %rax
0x018: halt
```



Detección de la condición de demora

```
#prog4
0x000: irmovq  $10, %rdx
0x00a: irmovq   $3, %rax
       bubble
       bubble
       bubble
0x014: addq    %rdx, %rax
0x016: halt
```



¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

Ciclo 4

Write-Back
Memory
Execute
Decode
Fetch

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq%rdx,%rax
0x016: halt
```

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

	Ciclo 5
Write-Back	`0x000: irmovq $10,%rdx`
Memory	`0x00a: irmovq $3,%rax`
Execute	*burbuja*
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

Ciclo 6

Write-Back	`0x00a: irmovq $3,%rax`
Memory	*burbuja*
Execute	*burbuja*
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

Ciclo 7

Write-Back	*burbuja*
Memory	*burbuja*
Execute	*burbuja*
Decode	`0x014: addq %rdx, %rax`
Fetch	`0x016: halt`

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

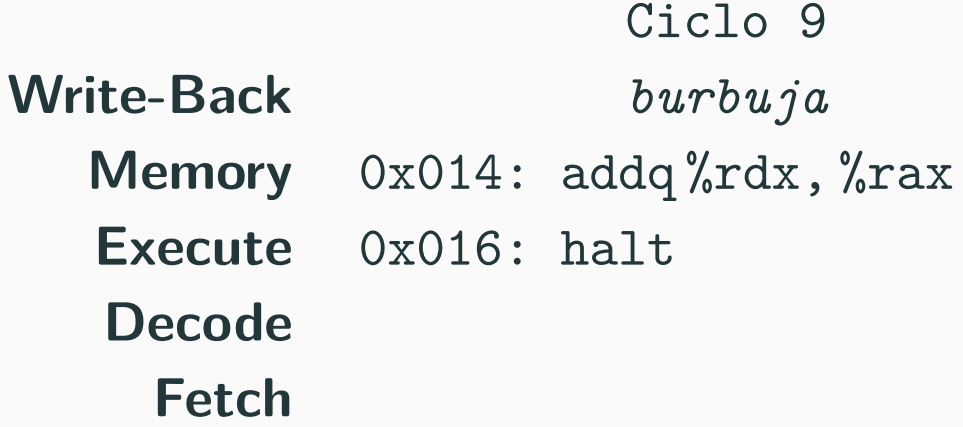
```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

	Ciclo 8
Write-Back	*burbuja*
Memory	*burbuja*
Execute	`0x014: addq %rdx, %rax`
Decode	`0x016: halt`
Fetch	

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```



- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

Ciclo 10

Write-Back `0x014: addq %rdx, %rax`

Memory `0x016: halt`

Execute

Decode

Fetch

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

¿Qué ocurre al retener las instrucciones?

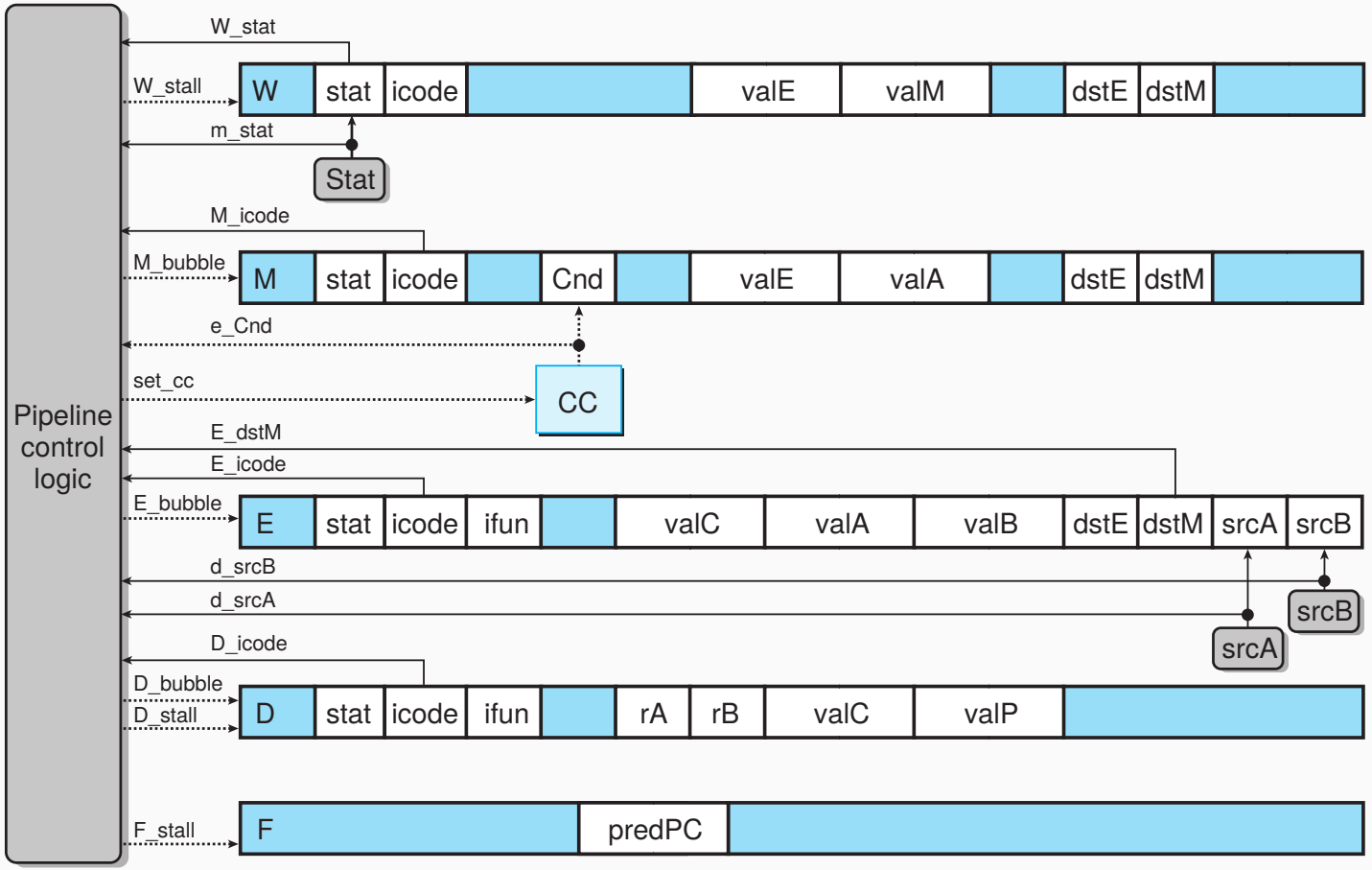
```
# prog4.ys
0x000: irmovq   $10, %rdx
0x00a: irmovq   $3, %rax
0x014: addq     %rdx, %rax
0x016: halt
```

Ciclo 11

Write-Back `0x016: halt`
Memory
Execute
Decode
Fetch

- La instrucción demorada se retiene en la etapa *decode*
- La siguiente instrucción se mantiene en la etapa *fetch*
- Se inyectan burbujas en la etapa *execute*
  - Como `nop`s generados dinámicamente
  - Los `nop` se propagan a través de las siguientes etapas

Implementación de la rentención

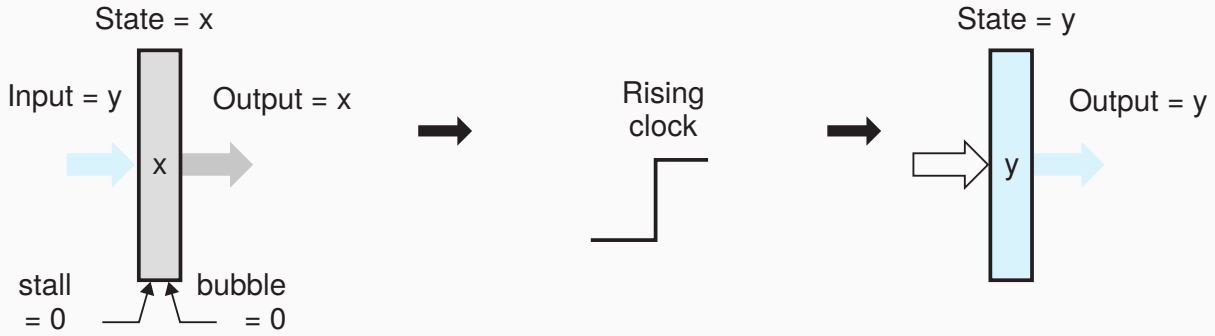


Lógica de Control

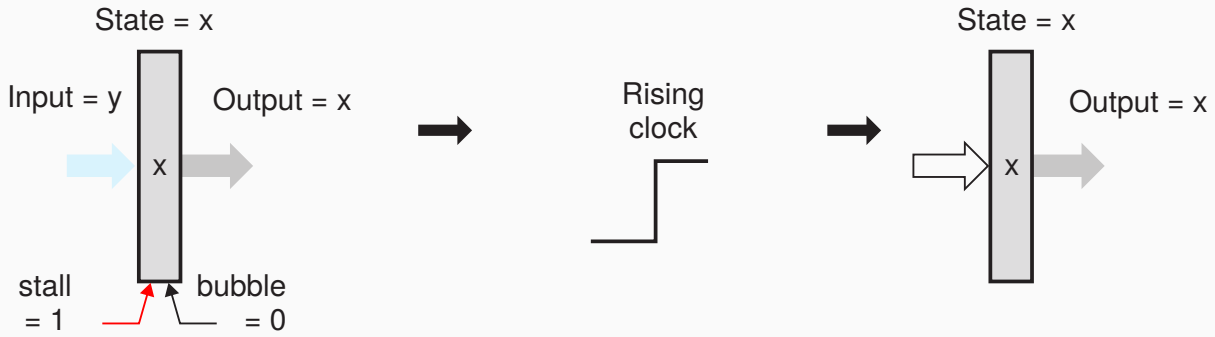
- La lógica de control detecta la condición de demora
- Estable las señales indicando a los registros cómo se deben actualizar

Modos de actualización de los registros de pipeline

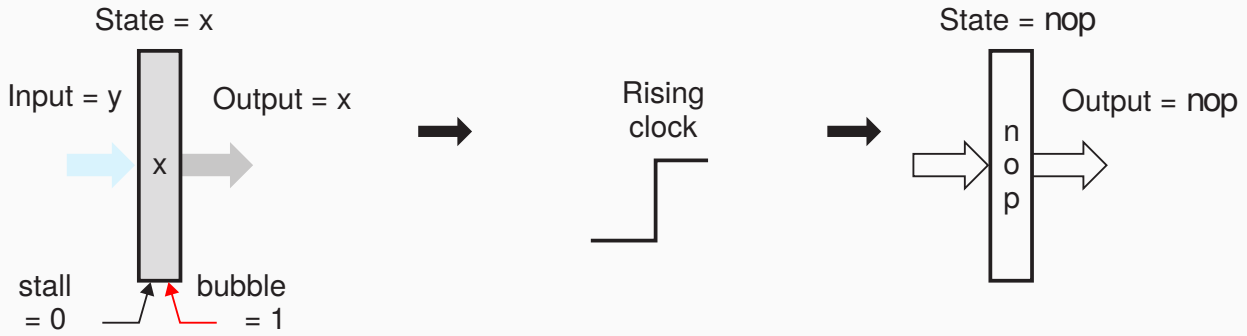
Normal



Retención



Burbuja



Envío de datos

Pipeline Naïve

- Los registros no se actualizan hasta el final de la etapa *write-back*
- Los operandos origen leen del banco de registros en la etapa *decode*
  - Es necesario que el valor esté guardado al comienzo de la etapa

Observación

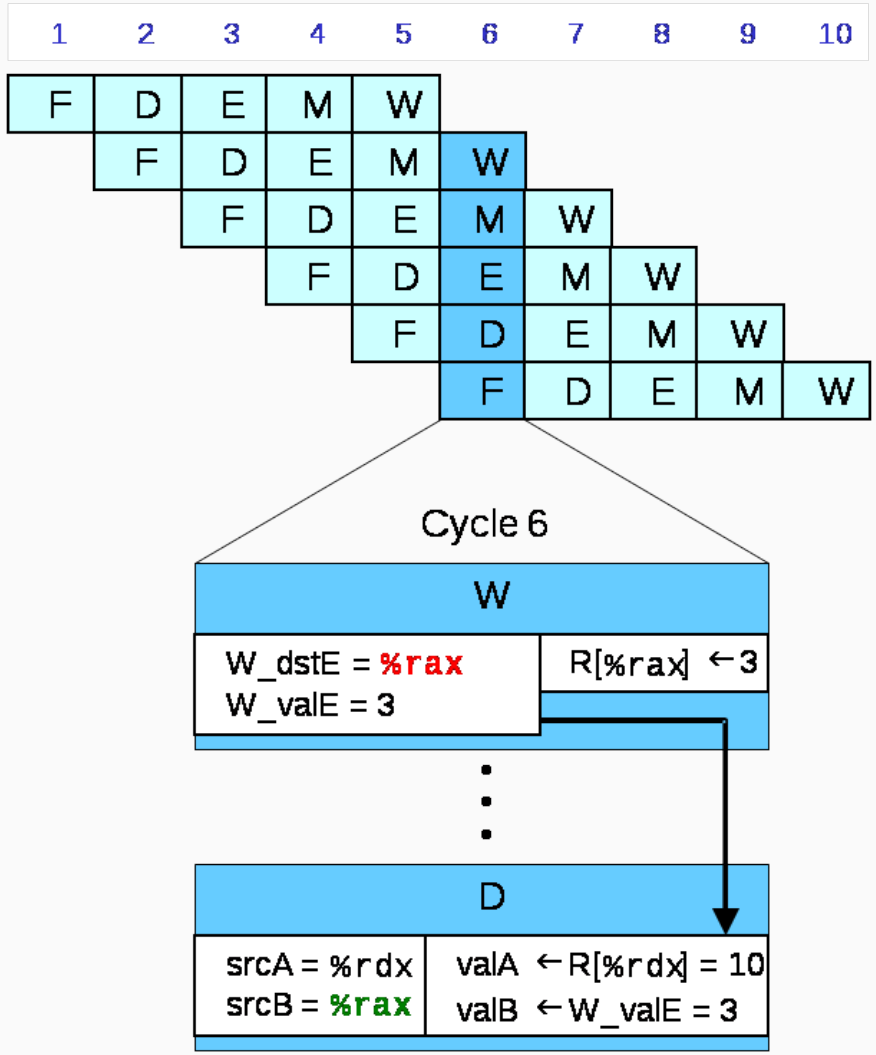
- El valor es generado en la etapa *execute* o en la etapa *memory*

Solución

- Pasar el valor de la etapa de generación directamente a la etapa *decode*
- Es necesario que esté disponible al final de la etapa *decode*

Ejemplo de envío de datos

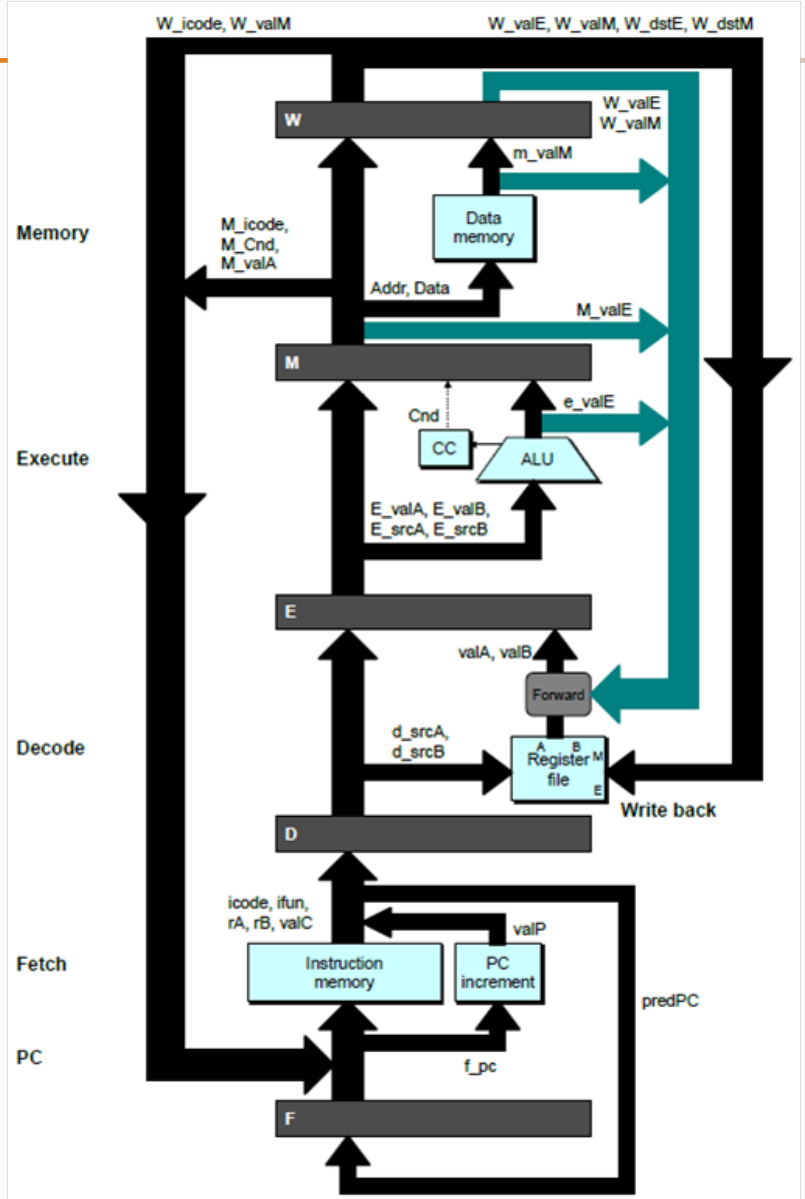
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `irmovq` en la etapa *write-back*
- El valor de destino en el registro W del pipeline
- Enviar como `valB` hacia la etapa decode

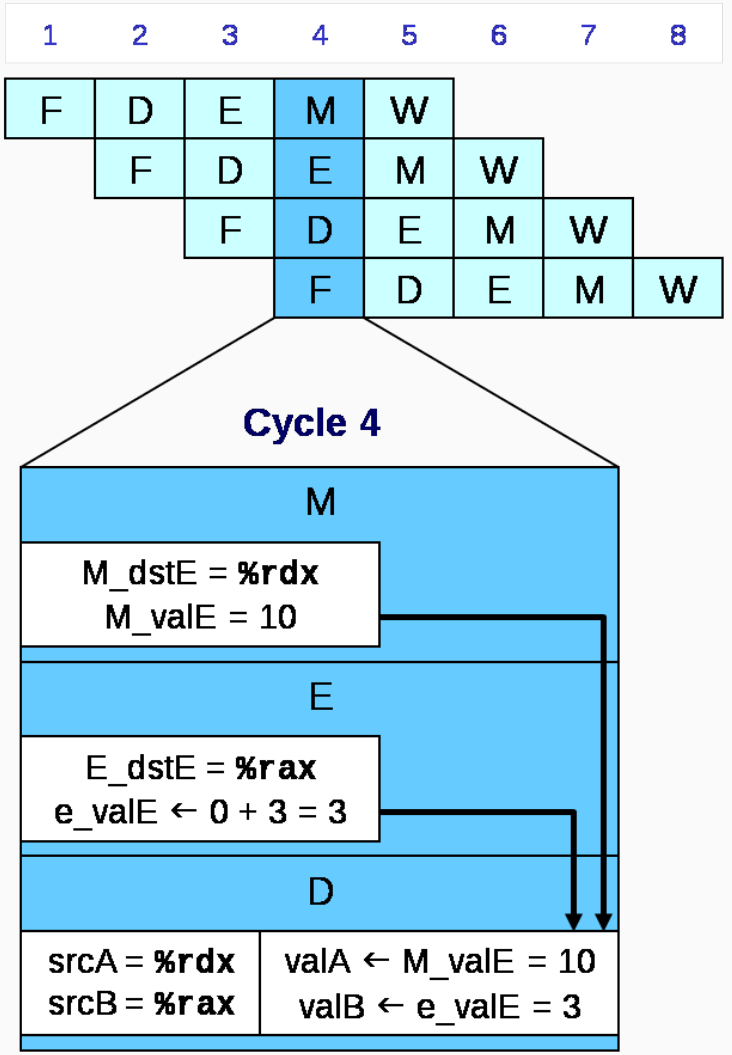
Caminos de envío

- **Etapa decode**
  - La lógica de reenvío selecciona `valA` y `valB`
  - Típicamente del banco de registros
  - Reenvío: obtiene `valA` o `valB` de una etapa posterior del pipeline
- **Fuentes de reenvío**
  - *Execute:* `valE`
  - *Memory:* `valE`, `valM`
  - *Write back:* `valE`, `valM`



Ejemplo 2 de envío de datos

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Registro %rdx

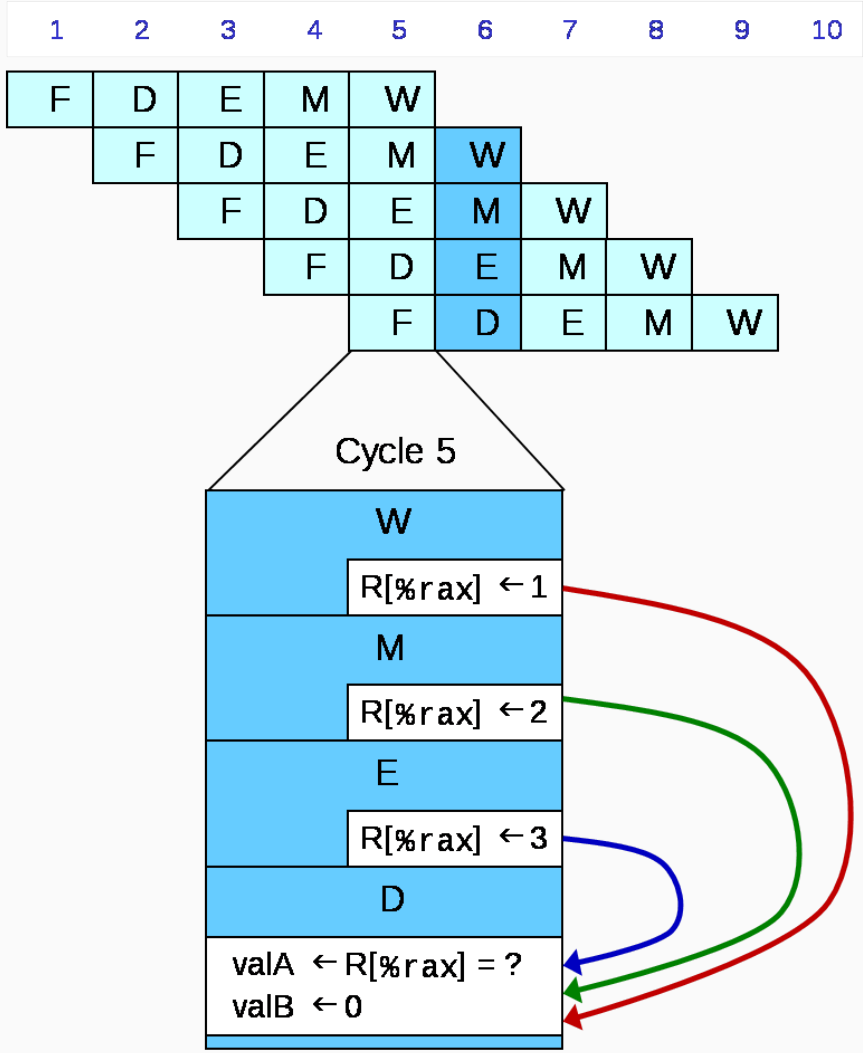
- La ALU genera el valor en el ciclo anterior
- Enviado desde la etapa *memory* como `valA`

Registro %rax

- Su valor fue recién generado por la ALU
- Se envía desde la etapa *execute* como `valB`

Prioridad de envío

```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```

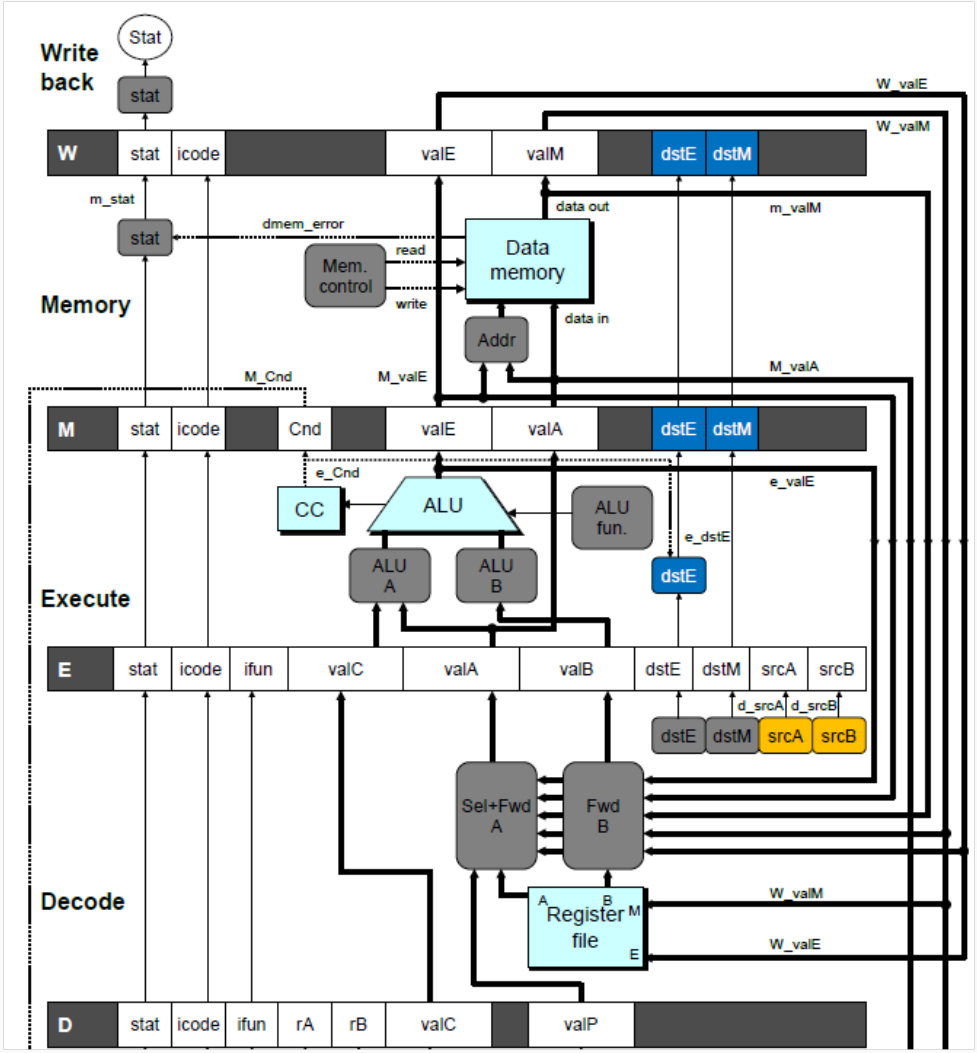


Opciones al tener múltiples orígenes

- ¿Cuál debería tener prioridad sobre las demás?
- Usar el valor de la etapa más cercana

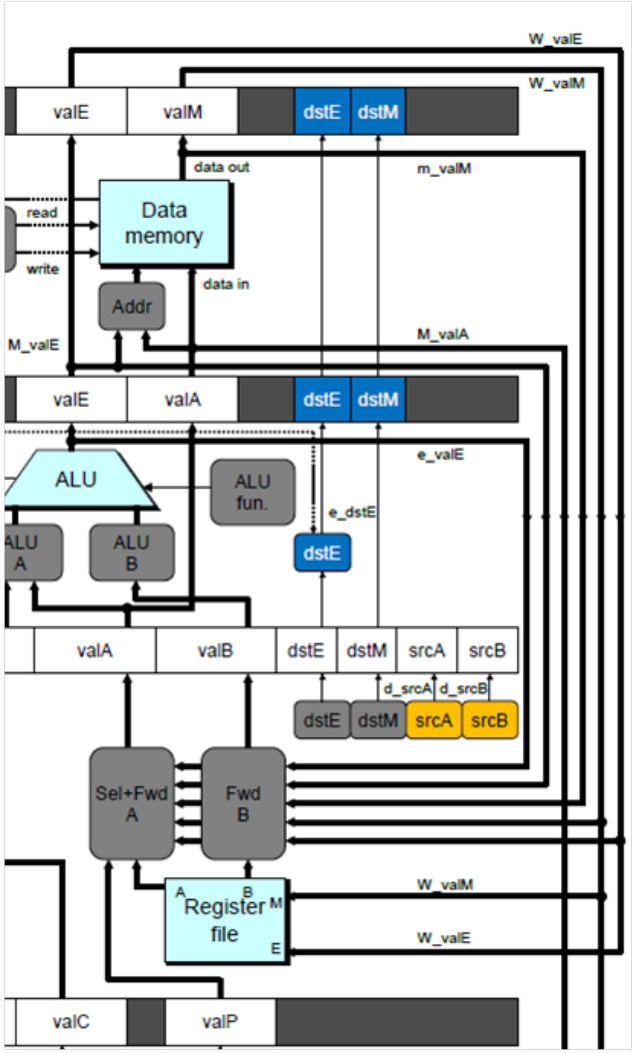
Implementación del reenvío de operandos

- Agrega circuitos de realimentación desde los registros de pipeline E, M, y W a la etapa decode
- Agrega lógica de control para seleccionar entre múltiples opciones los valores para `valA` y `valB`

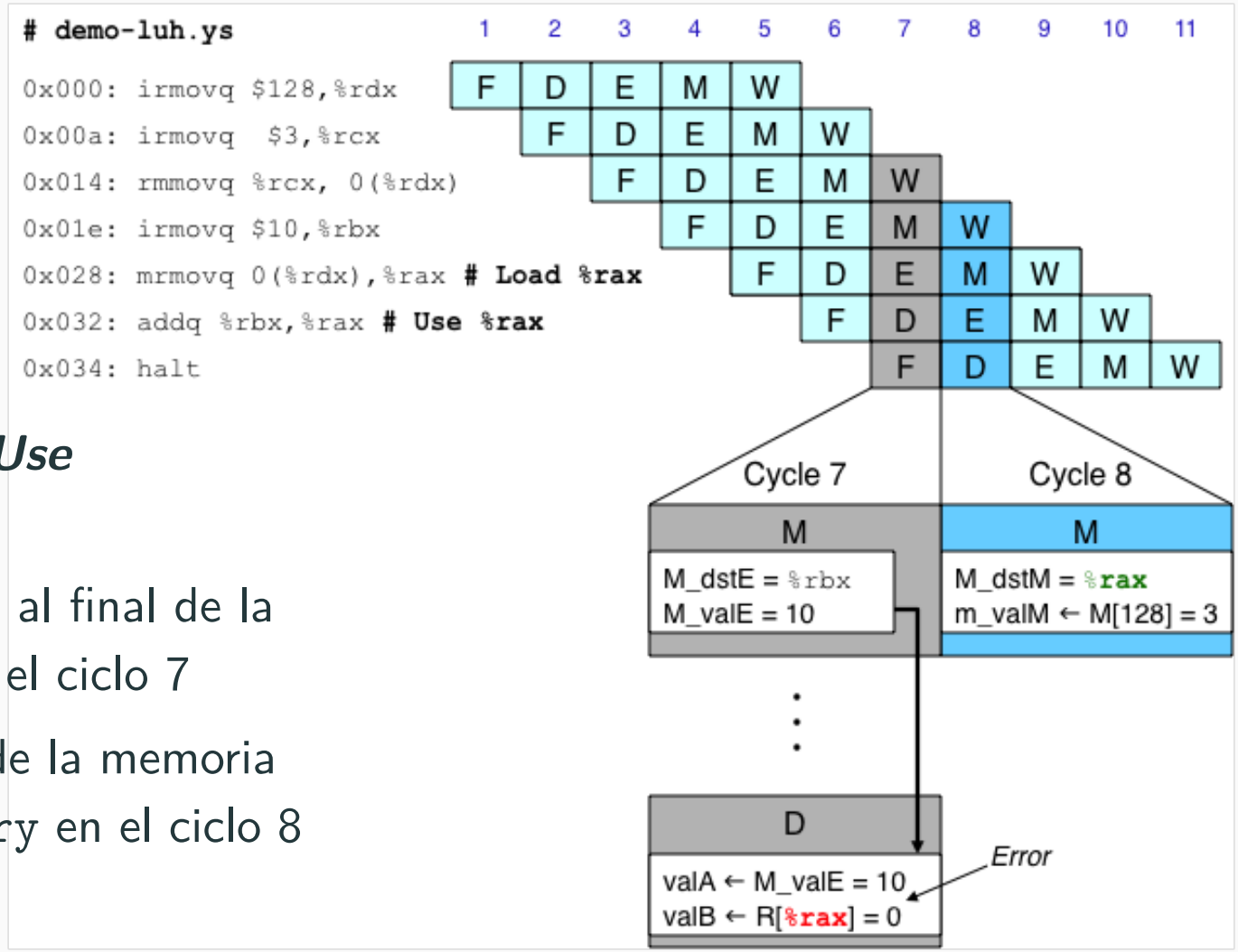


Implementación del reenvío de operandos

```
1  ## ¿Cuál debebería ser el valor de A?
2  int d_valA = [
3    # Use incremented PC
4    D_icode in { ICALL, IJXX } : D_valP;
5    # Forward valE from execute
6    d_srcA == e_dstE : e_valE;
7    # Forward valM from memory
8    d_srcA == M_dstM : m_valM;
9    # Forward valE from memory
10   d_srcA == M_dstE : M_valE;
11   # Forward valM from write back
12   d_srcA == W_dstM : W_valM;
13   # Forward valE from write back
14   d_srcA == W_dstE : W_valE;
15   # Use value read from register file
16   1 : d_rvalA;
17 ];
```



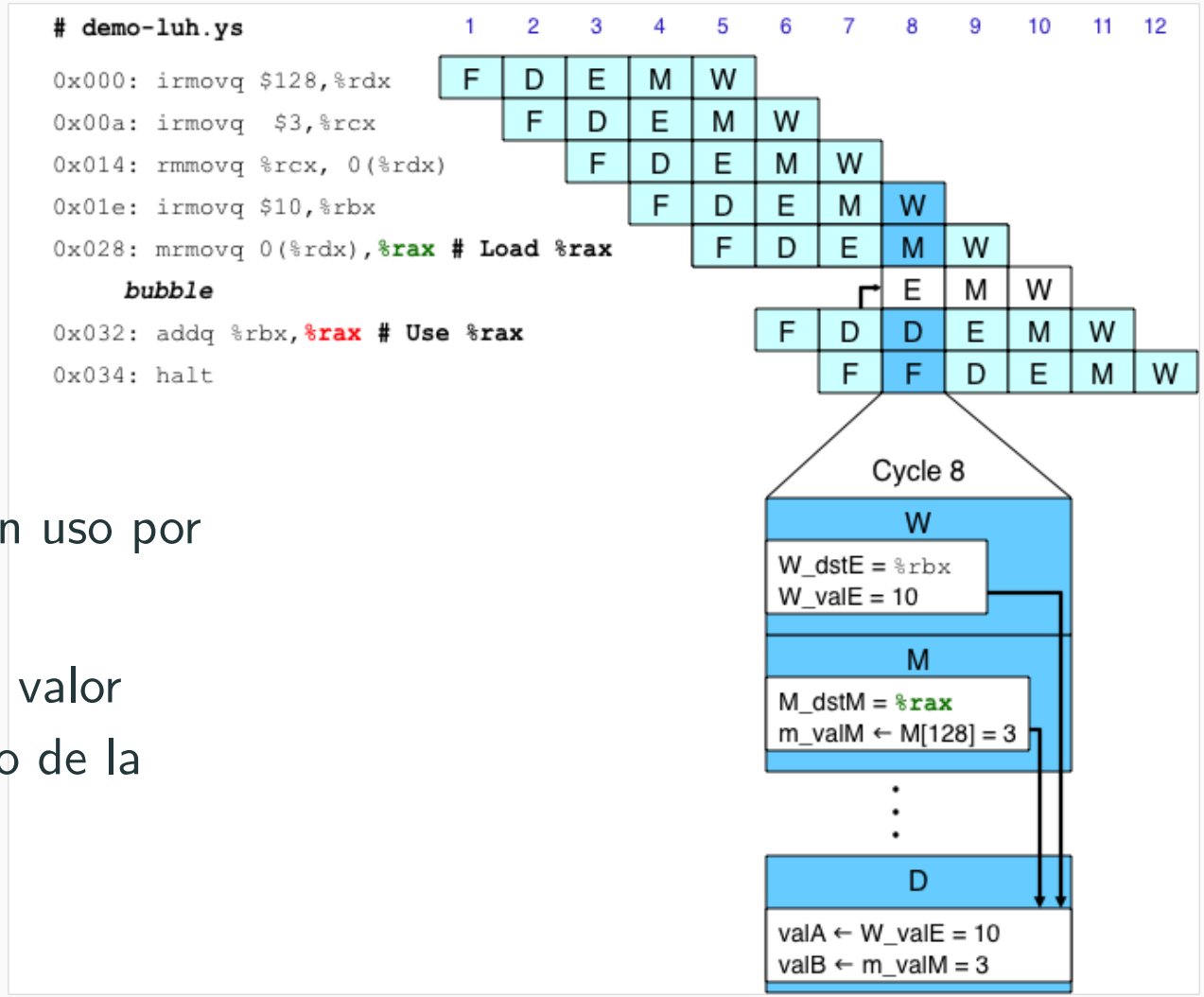
Limitaciones de la realimentación



Dependencia *Load/Use* (Carga/Uso)

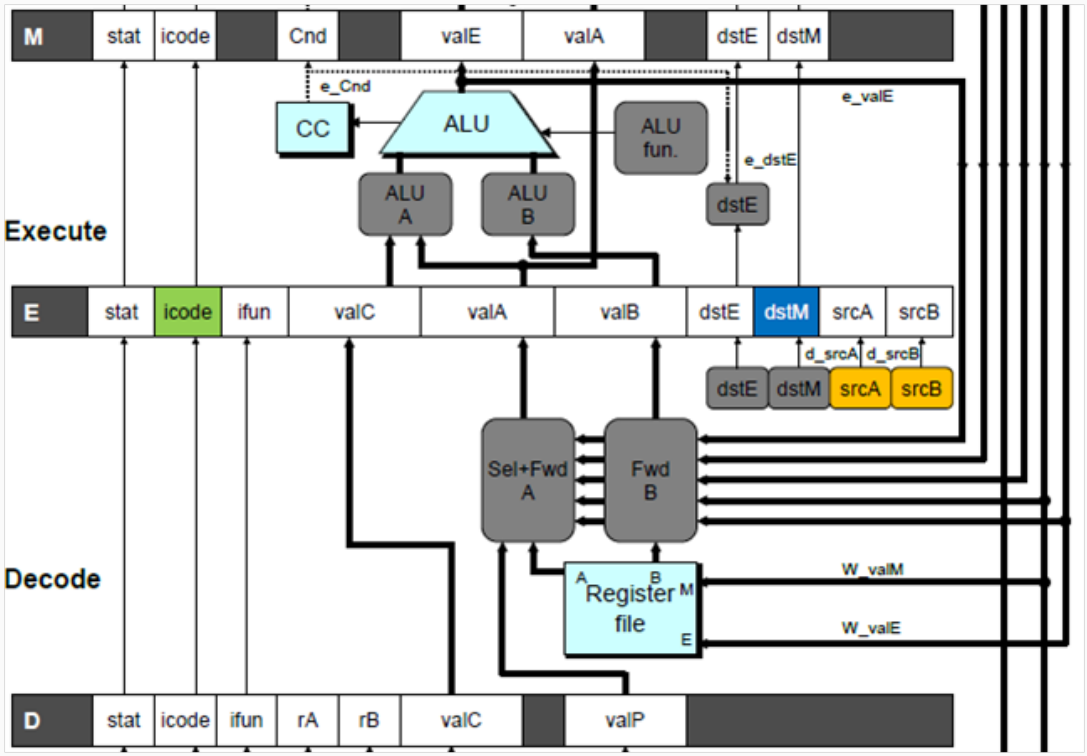
- Requiere un valor al final de la etapa `decode` en el ciclo 7
- El valor es leído de la memoria en la etapa `memory` en el ciclo 8

¿Cómo se evita el riesgo de carga/uso?



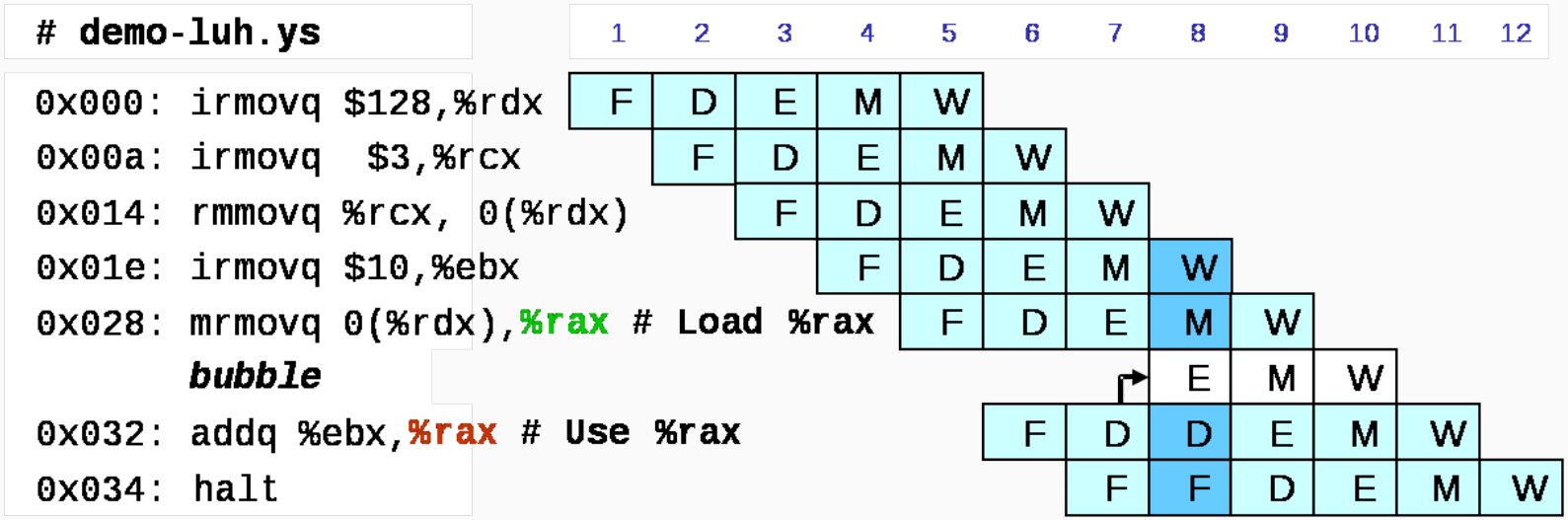
- Frena la instrucción en uso por un ciclo
- Luego puede tomar el valor cargado realimentando de la etapa `memory`

Detección del riesgo de carga/uso



Evento	Condición
Riesgo *Load/Use*	`E_icode in {IMRMOVQ, IPOPQ } && E_dstM in {d_srcA, d_srcB}`

Control del riesgo de carga/uso



- Frenar las instrucciones en las etapas `fetch` y `decode`
- Inyectar una burbuja en la etapa `execute`

Evento	F	D	E	M	W
Riesgo *Load/Use*	stall	stall	burbuja	normal	normal

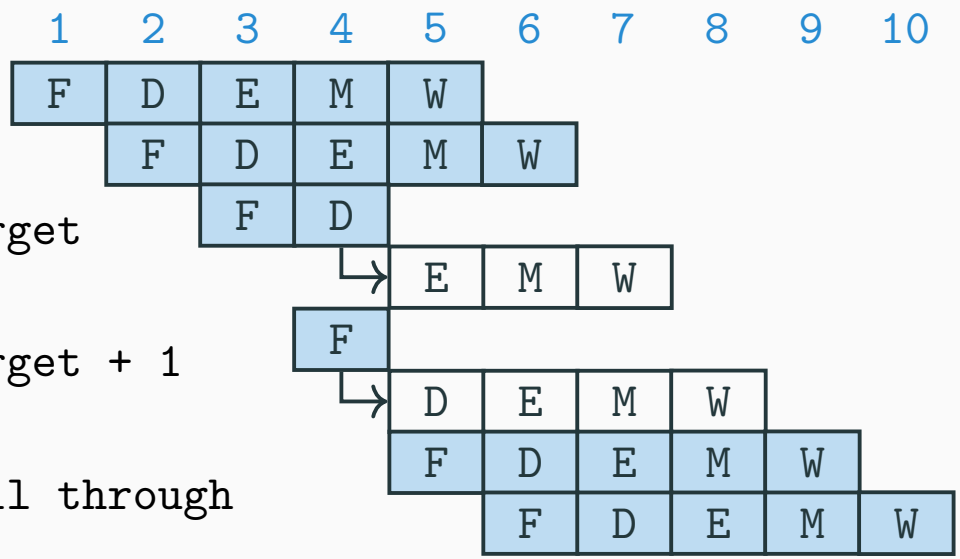
Ejemplo: error en la predicción

```
# prog7
0x000:     xorq    %rax,%rax
0x002:     jne     target     # No debería saltar
0x00b:     irmovq $1, %rax    # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx    # No debe ejecutarse
0x020:     irmovq $3, %rbx    # No debe ejecutarse
0x02a:     halt
```

- Sólo debe ejecutar las primeras 4 instrucciones (¿por qué?)
- Con predictor "*salta siempre*" ¿qué hace?

Procesamiento de errores de predicción

```
# prog7
0x000: xorq    %rax,%rax
0x002: jne     target
0x016: irmovq $2,%rdx # target
       burbuja
0x020: irmovq $3,%rbx # target + 1
       burbuja
0x00b: irmovq $1,%rax # fall through
0x015: halt
```



Se predice que la rama se toma (o no)

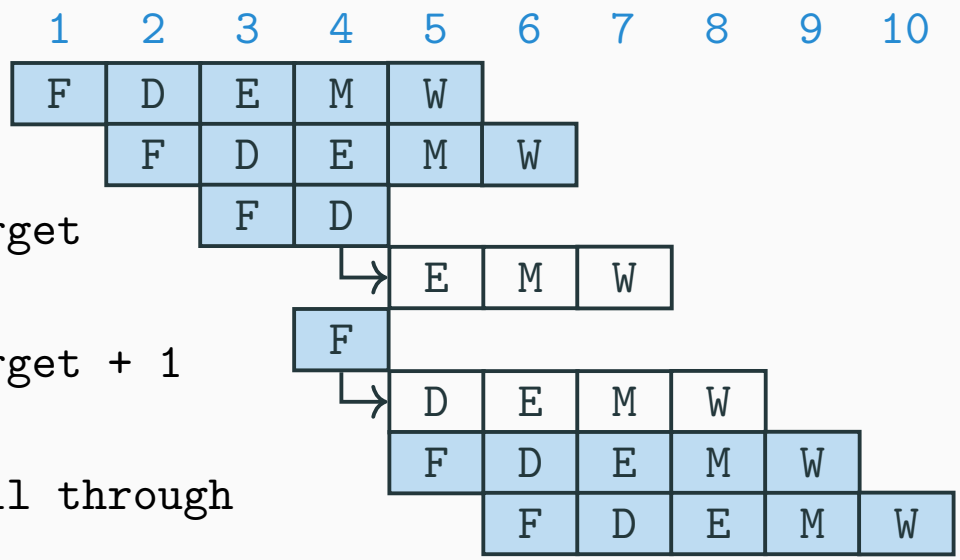
- La unidad de generación de instrucciones tomará 2 instrucciones.

Cancelar al fallar en la predicción

- Se detecta el error en la etapa de ejecución
- Reemplazar las instrucciones de las etapas *execute* y *decode* por burbujas
- Las instrucciones reemplazadas no generaron cambios en el estado del pipeline

Control de errores de predicción

```
# prog7
0x000: xorq    %rax,%rax
0x002: jne     target
0x016: irmovq $2,%rdx # target
       burbuja
0x020: irmovq $3,%rbx # target + 1
       burbuja
0x00b: irmovq $1,%rax # fall through
0x015: halt
```



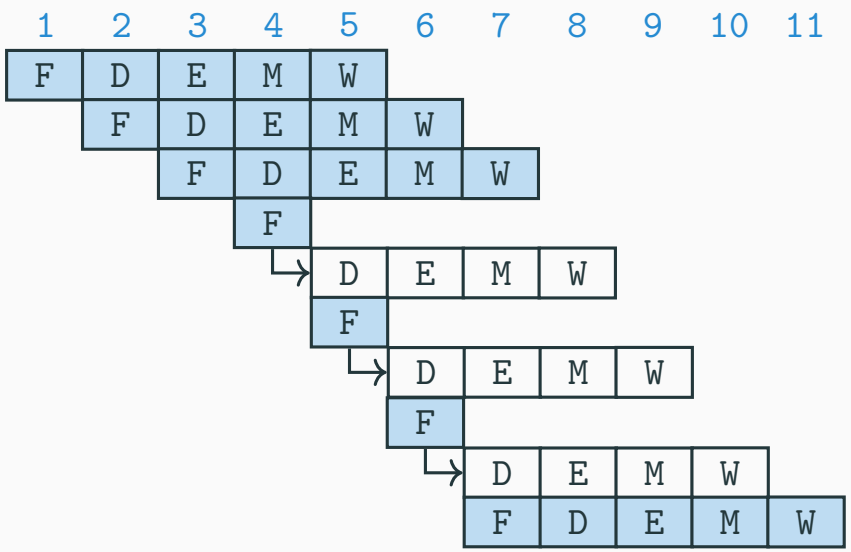
Evento	F	D	E	M	W
Error de predicción	normal	burbuja	burbuja	normal	normal

Ejemplo: *return*

```
0x000:    irmovq stack,%rsp #    Inicializar SP
0x00a:    call proc         #    Call
0x013:    irmovq $10,%rdx   #    Punto de retorno
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                # proc:
0x020:    ret               #    Retorno imediato
0x021:    rrmovq %rdx,%rbx  #    No ejecutar
0x030: .pos 0x30
0x030: stack:               # stack: Stack pointer
```

Procesamiento del *return*

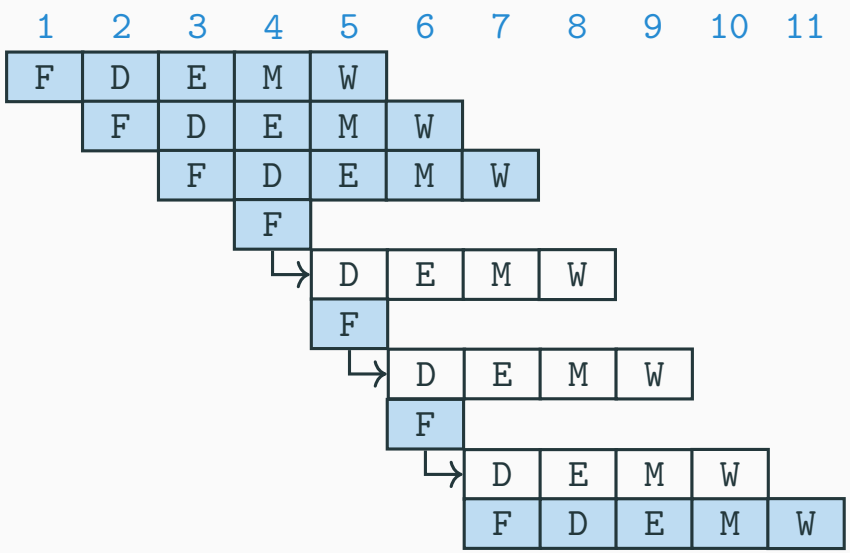
```
# prog6
0x000: irmovq stack,%rsp
0x00a: call   proc
0x020: ret
0x021: rrmovq %rdx,%rbx
       burbuja
0x021: rrmovq %rdx,%rbx
       burbuja
0x021: rrmovq %rdx,%rbx
       burbuja
0x00b: irmovq  $10,%rdx
```



- Mientras `ret` pasa por el *pipeline*, la etapa *fetch* se mantiene parada
- Se insertan burbujas en la etapa *decode*
- Se continúa con la ejecución cuando `ret` llega a la etapa de escritura

Control del *return*

```
# prog6
0x000: irmovq stack,%rsp
0x00a: call   proc
0x020: ret
0x021: rrmovq %rdx,%rbx
       burbuja
0x021: rrmovq %rdx,%rbx
       burbuja
0x021: rrmovq %rdx,%rbx
       burbuja
0x00b: irmovq  $10,%rdx
```



Evento	F	D	E	M	W
Detección del *return*	stall	burbuja	normal	normal	normal

Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`