

DCC
DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 12 - Paralelismo a Nivel de Instrucción (ILP)

Profesor: Germán Leandro Contreras Sagredo

Objetivos de la clase

- Conocer el concepto de paralelismo a nivel de instrucción.
- Entender las modificaciones que debe tener el computador básico para permitir paralelismo a nivel de instrucción.
- Comprender los distintos problemas que pueden existir durante la ejecución paralela de un conjunto de instrucciones.
- Realizar ejercicios que consoliden los conocimientos anteriores.

Hasta ahora...

- Ya construimos un computador que tiene todo lo necesario para que pueda ser operado y programado, además de implementar lo necesario para que se comunique con otros dispositivos.

- Además, vimos cómo permitir la ejecución de múltiples procesos y cómo optimizar los accesos a memoria mediante cachés.

Ahora, veremos mejoras en lo que respecta a la ejecución de **instrucciones en la CPU**.

Paralelismo en la CPU

En multiprogramación, aplicamos un concepto de “paralelismo” que no era 100% real: dos o más procesos se turnan el uso de la CPU para la ejecución de sus instrucciones.

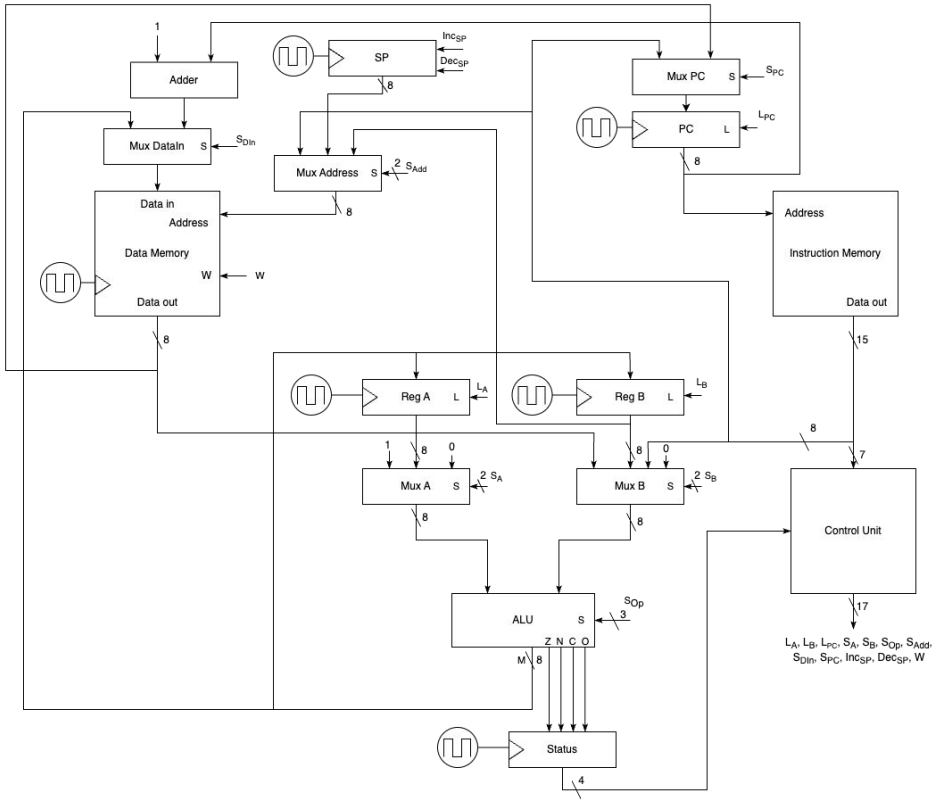
Lo que estudiaremos ahora será la posibilidad de ejecutar múltiples instrucciones **al mismo tiempo**, generando un paralelismo ***casi*** real.

Veremos cómo hacerlo a partir del computador básico de este curso.

Paralelismo en la CPU - Computador básico

Al finalizar la construcción de nuestro computador programable, obtuvimos el diagrama de la figura.

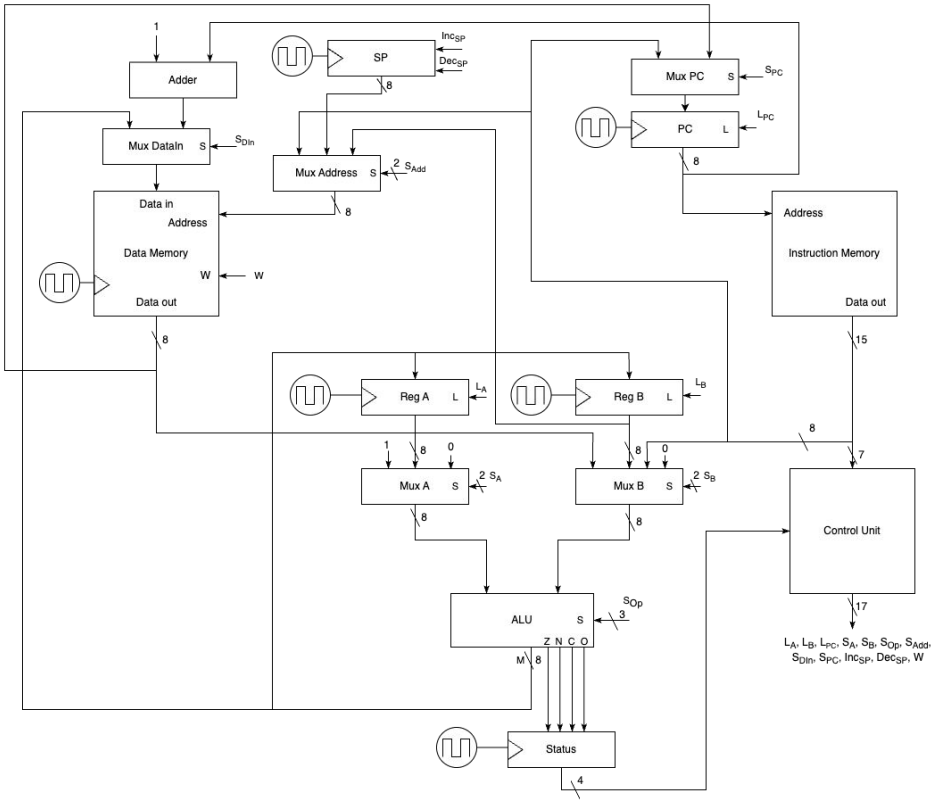
Este presenta una arquitectura ***Register-memory***, ya que permite operaciones directas entre la memoria y los registros.



Paralelismo en la CPU - Computador básico

Este esquema **complejiza** la posibilidad de implementar paralelismo, como veremos más adelante.

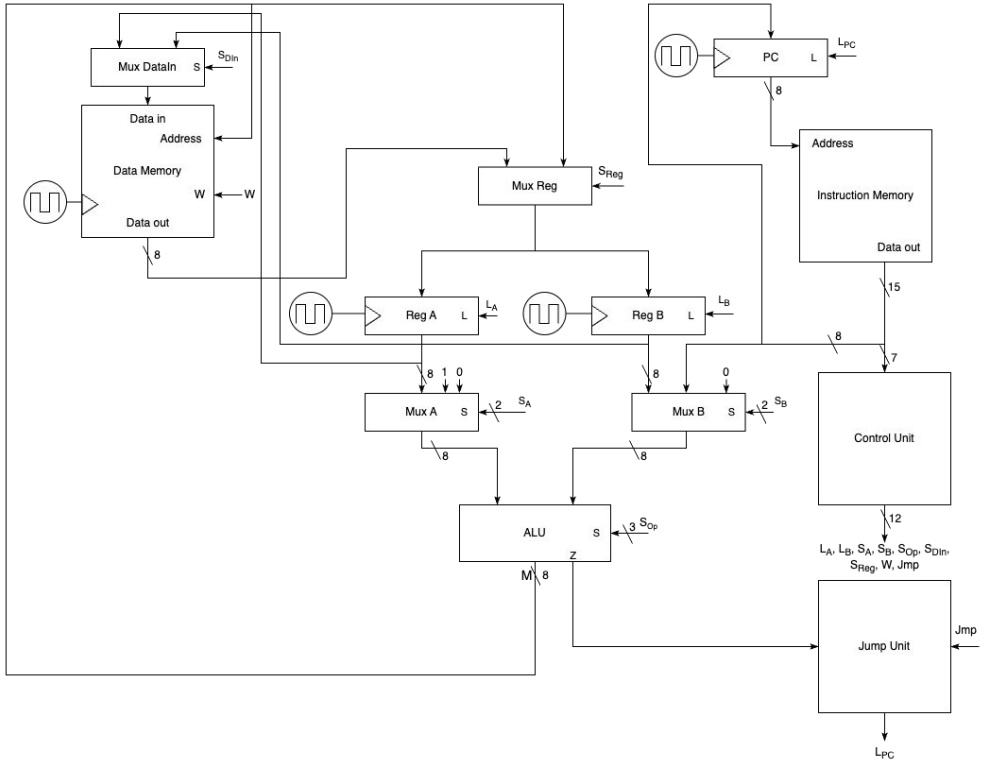
Por lo tanto, reduciremos nuestra arquitectura a una versión **simplificada** para facilitar la posterior paralelización.



Paralelismo en la CPU - Computador básico simplificado

A continuación, se muestra la arquitectura resultante de la simplificación.

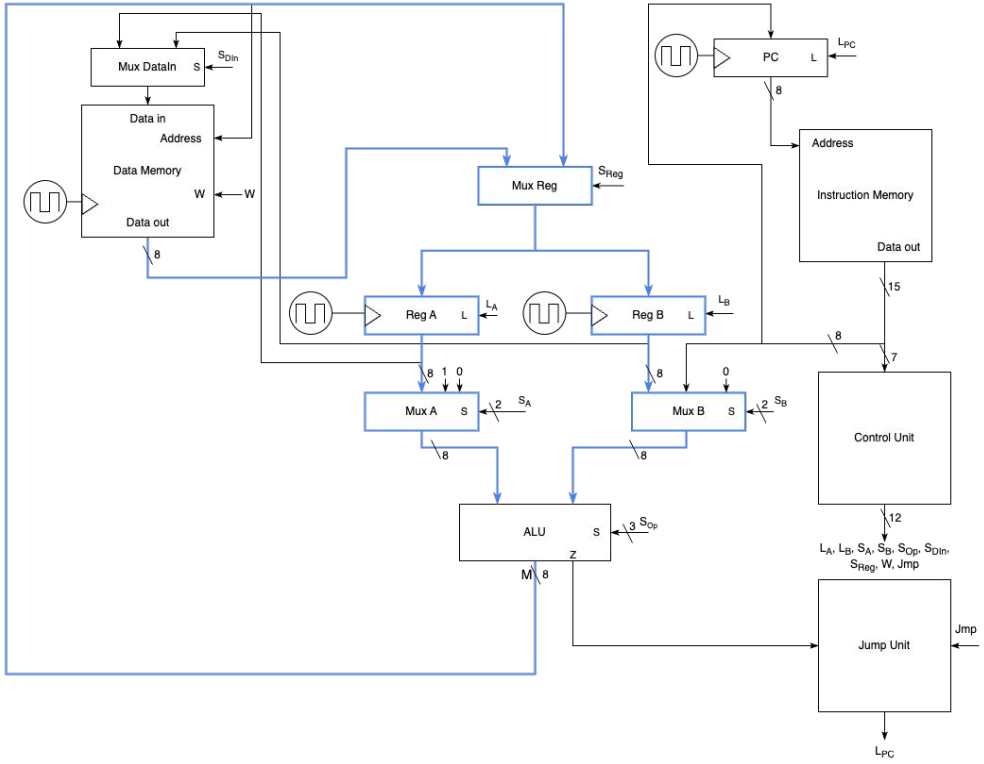
¿Qué cambios relevantes podemos observar?



Paralelismo en la CPU - Computador básico simplificado

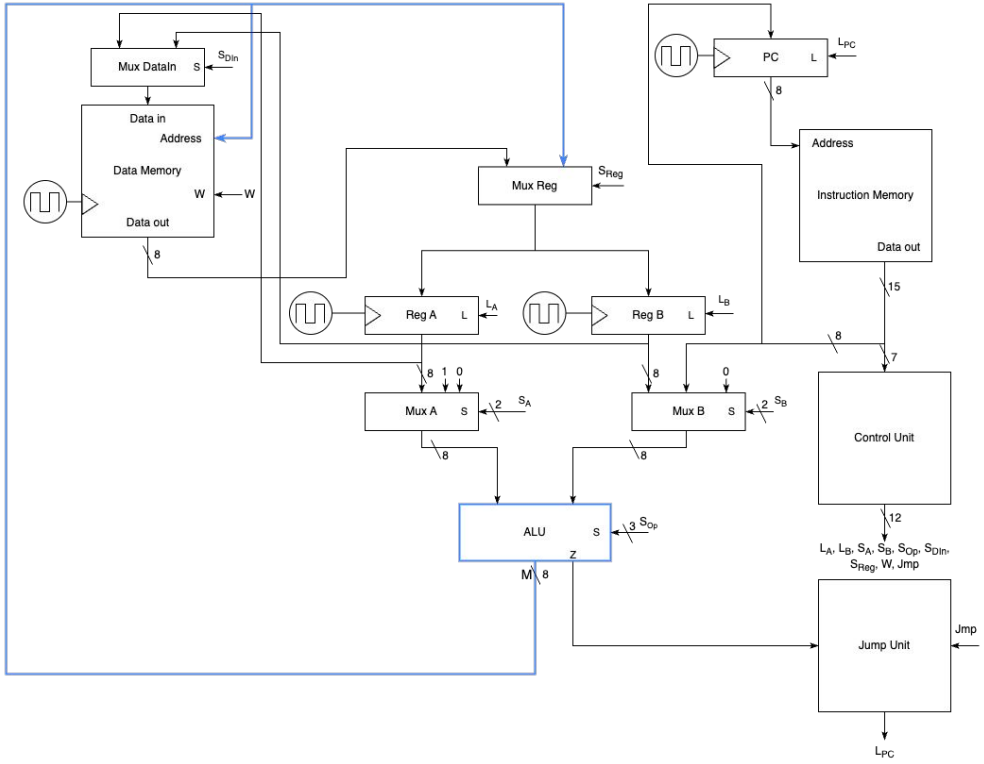
- No se trabaja directamente con la salida de memoria. La ALU recibe solo los valores de los registros, el literal, o las señales 0 y 1.

- Los registros siguen almacenando el resultado de la ALU o la salida de la memoria de datos (esto ahora de forma explícita).



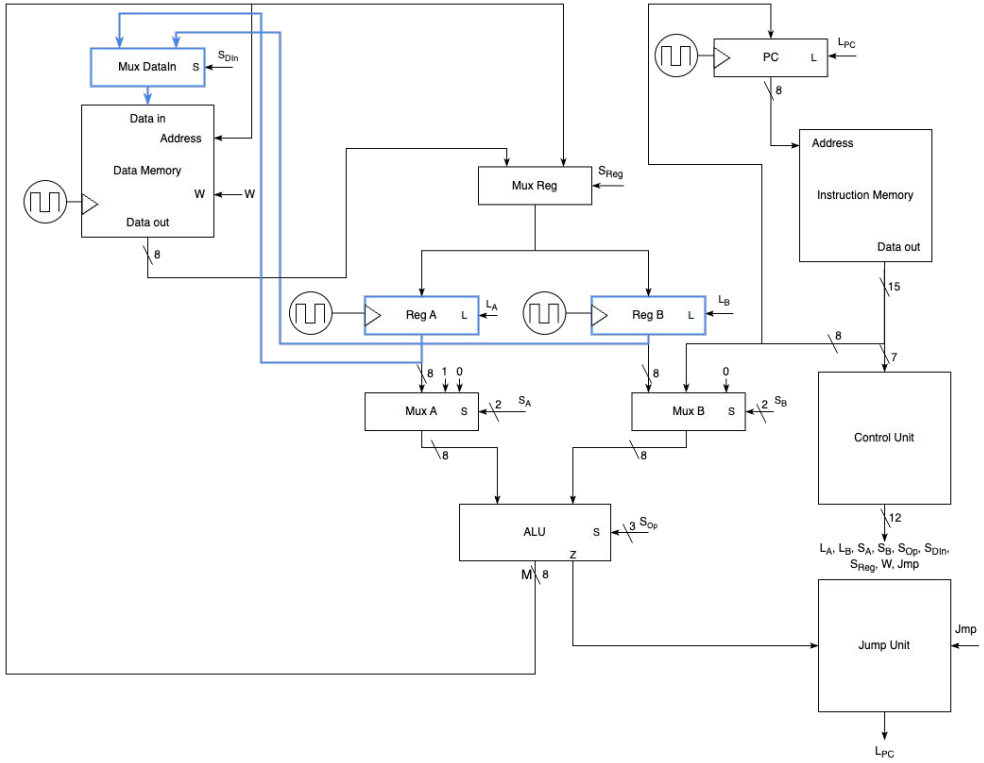
Paralelismo en la CPU - Computador básico simplificado

- El resultado de la ALU no solo se guarda en los registros, sino que se utiliza a su vez como la dirección de la memoria de datos.
- Se elimina el manejo de memoria *stack*.



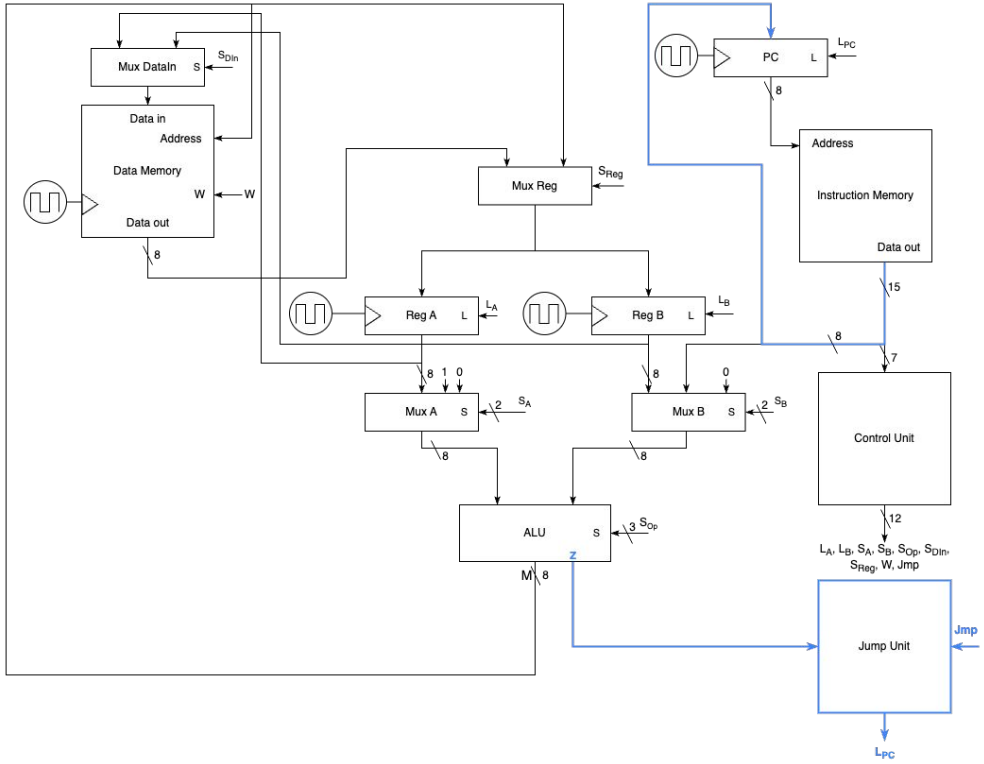
Paralelismo en la CPU - Computador básico simplificado

- En la memoria de datos solo se puede almacenar el valor del registro *A* o *B*. Por lo tanto, para almacenar resultados de la ALU, primero se deben guardar en estos.



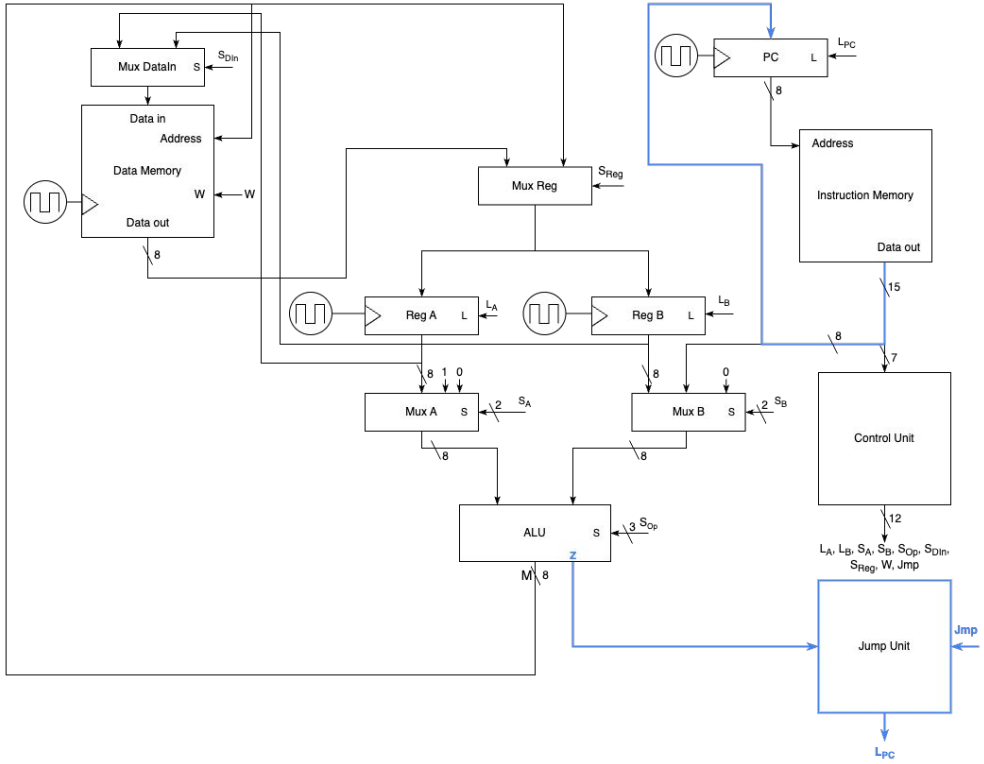
Paralelismo en la CPU - Computador básico simplificado

- Se elimina el soporte de subrutinas (ya no se almacena la dirección de retorno ni se carga en PC).
- Se implementan saltos condicionales con la *flag* Z a través de una unidad nueva: ***Jump Unit***.



Paralelismo en la CPU - Computador básico simplificado

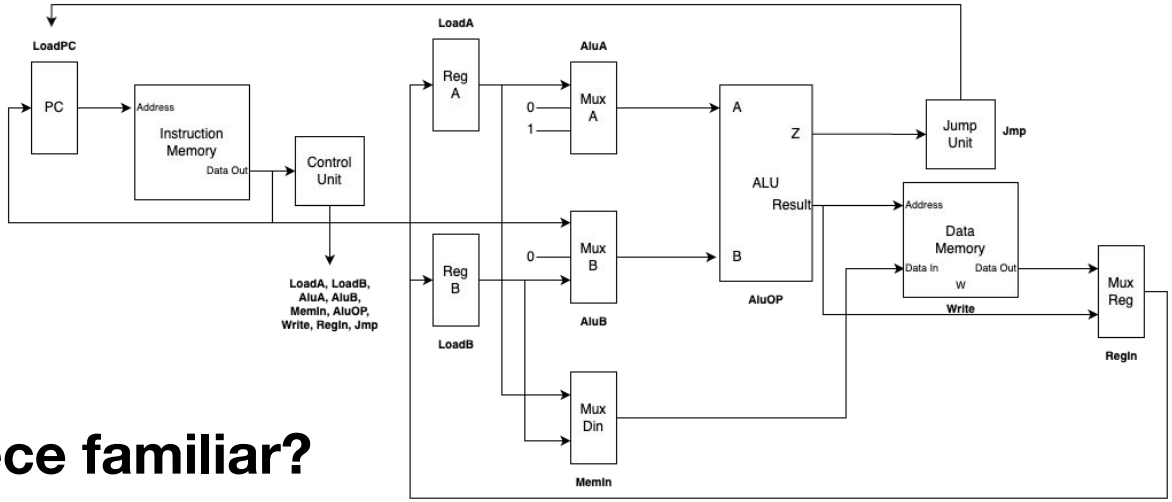
- Adicionalmente, se elimina el registro *Status*. Ahora, las instrucciones de salto deben incluir la operación de `CMP` (*i.e. A - B*) para actualizar la *flag* `Z` y enviarla a la *Jump Unit*.



Paralelismo en la CPU - Computador básico simplificado

¿Por qué hacer estas simplificaciones?

Porque nos permiten **reordenar** la arquitectura de la siguiente forma:



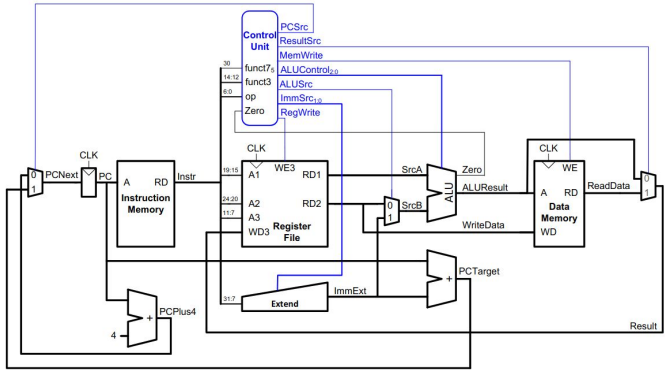
Cambiamos los nombres de las señales para que sean más explícitas, pero aplican de la misma forma.

¿Les parece familiar?

Paralelismo en la CPU - Computador básico simplificado

Recordatorio: RISC-V. Nuestra estructura simplificada es similar a la implementación de un ciclo de la arquitectura RISC-V.

Esto no es una coincidencia: nuestra arquitectura ahora también es de tipo ***Load-Store*** o ***Register-Register***, *i.e.* las operaciones de la ALU solo se guardan en registros y las operaciones de lectura y escritura solo se realizan desde y hacia ellos.

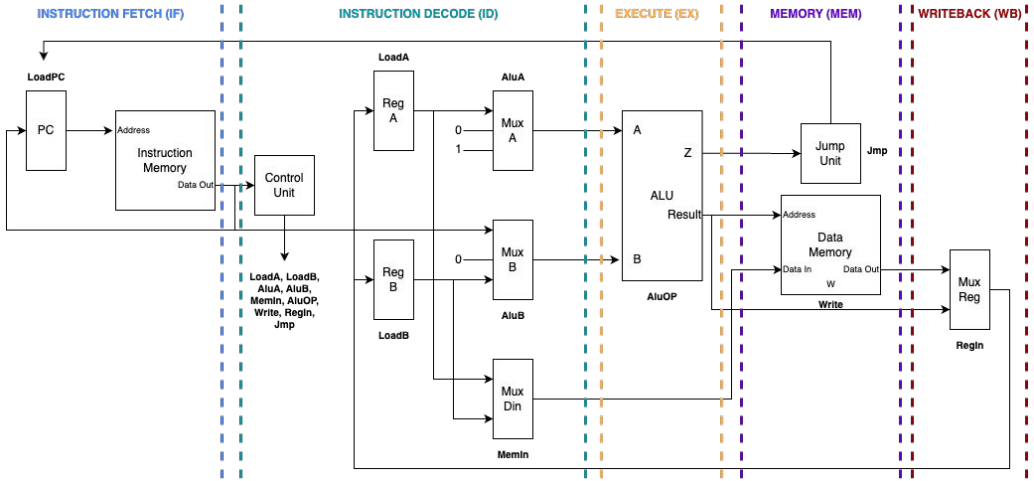


Esto permite separar la ejecución en **etapas**.

Paralelismo en la CPU - Computador básico simplificado

Etapas de ejecución: Ahora, cada instrucción puede separarse en cinco etapas distintas, cada una con distintos tiempos de ejecución:

- ***Instruction Fetch* (IF):** Obtención del *opcode* y literal.
- ***Instruction Decode* (ID):** Decodificación del *opcode* en señales de control.
- ***Execute* (EX):** Ejecución de la instrucción en la unidad (ALU).
- ***Memory* (MEM):** Lectura o escritura en memoria de datos.
- ***Writeback* (WB):** Escritura en registros.



Paralelismo en la CPU - *Pipeline*

Los procesos secuenciales con etapas **independientes** se conocen como ***pipelines***.

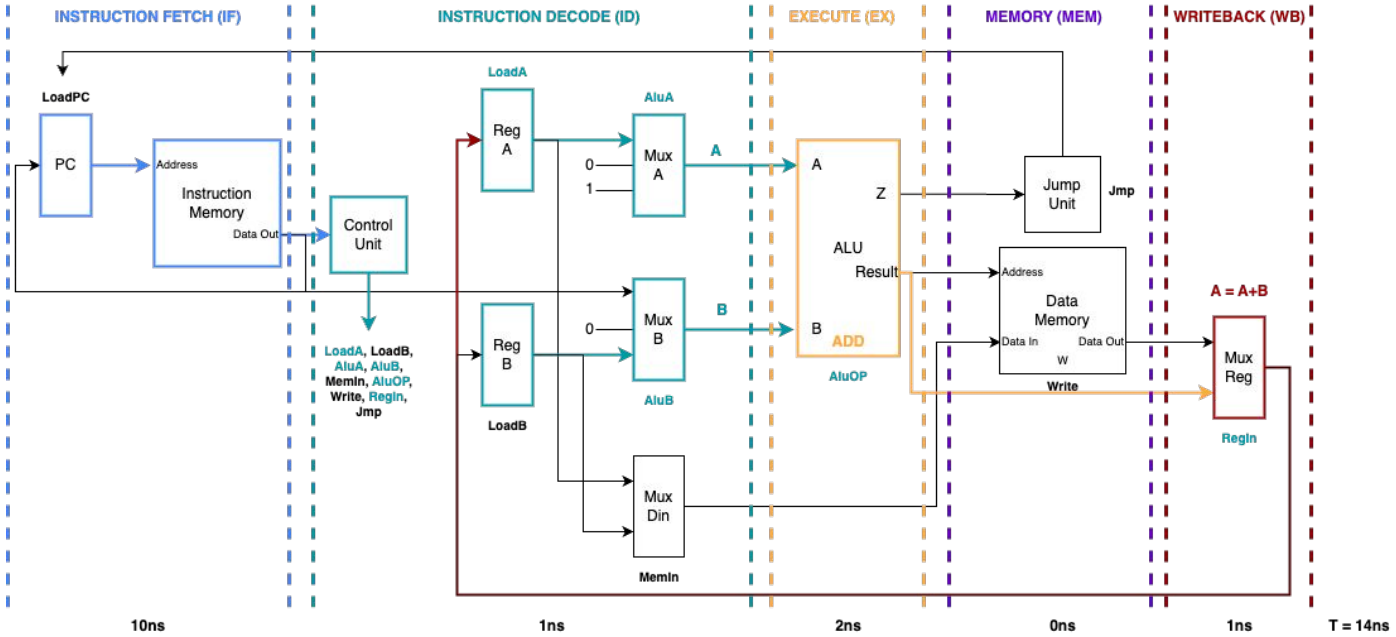
Esta nueva versión del computador básico nos permite ver el flujo de una instrucción como un *pipeline*, siendo cada etapa independiente.

Podemos basarnos en esta implementación para **ejecutar múltiples instrucciones de forma simultánea** (con un máximo igual a la cantidad de etapas de nuestro *pipeline*).

A continuación, veremos el flujo de algunas instrucciones.

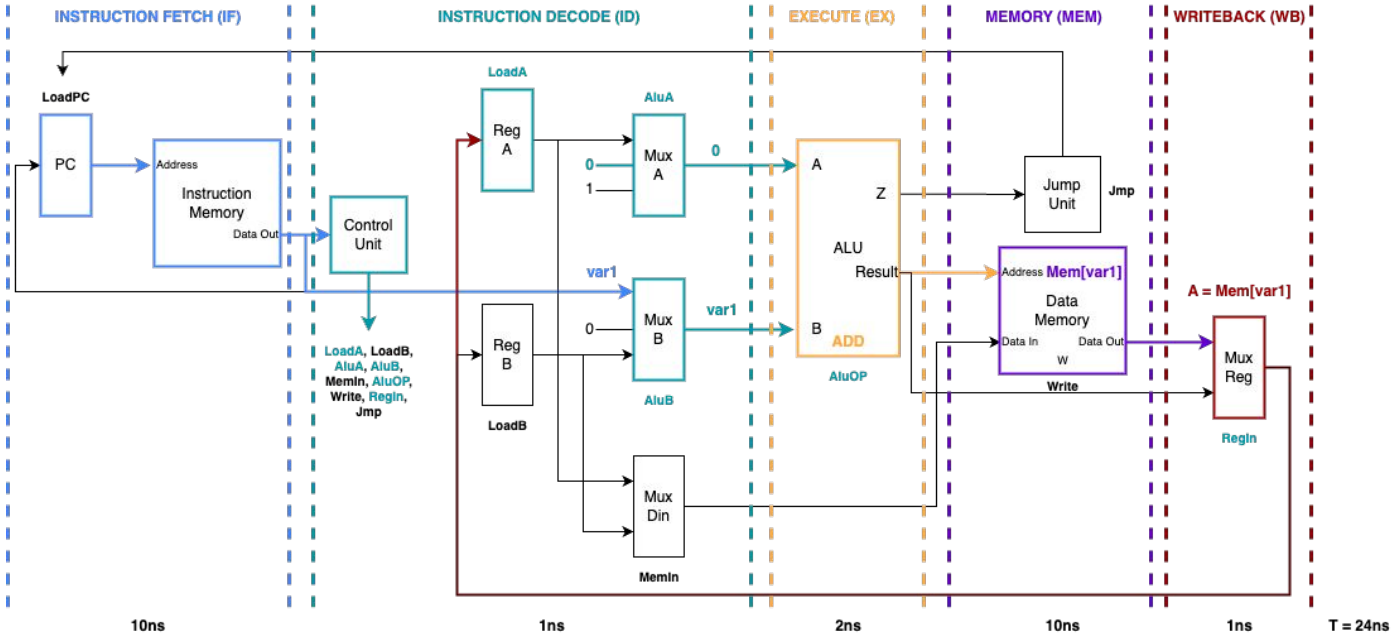
Paralelismo en la CPU - *Pipeline*

Ejemplo: ADD A,B



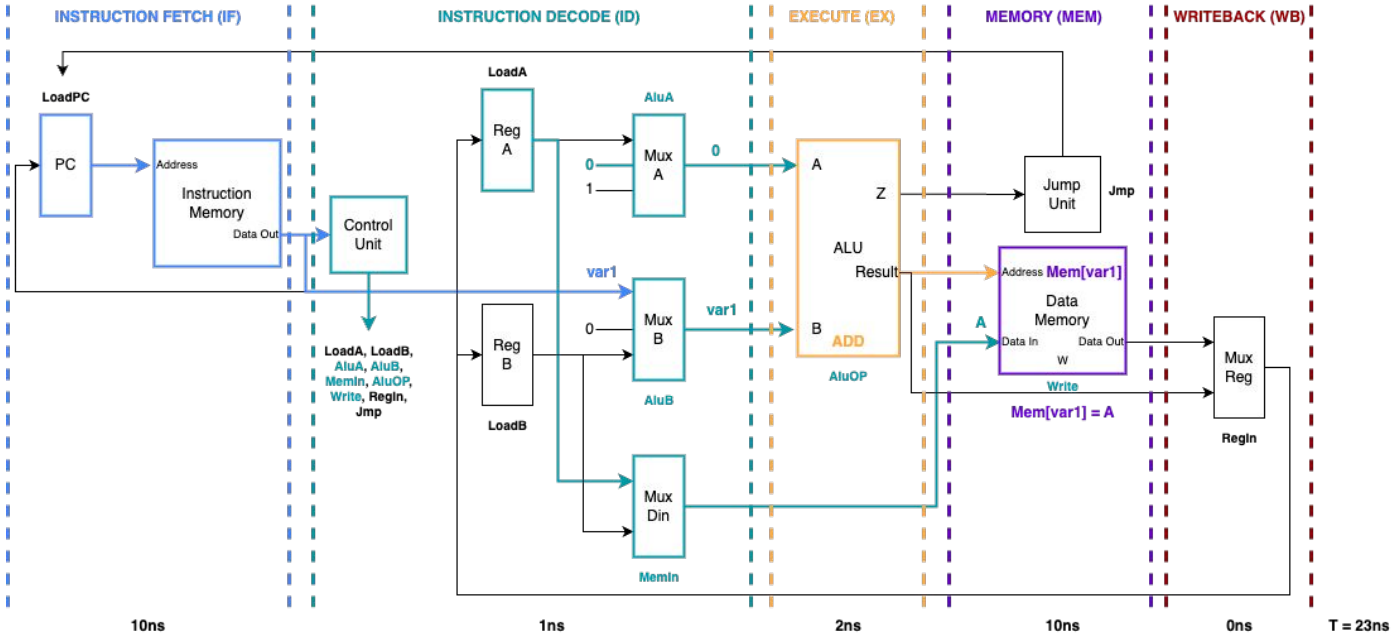
Paralelismo en la CPU - *Pipeline*

Ejemplo: `MOV A,(var1)`



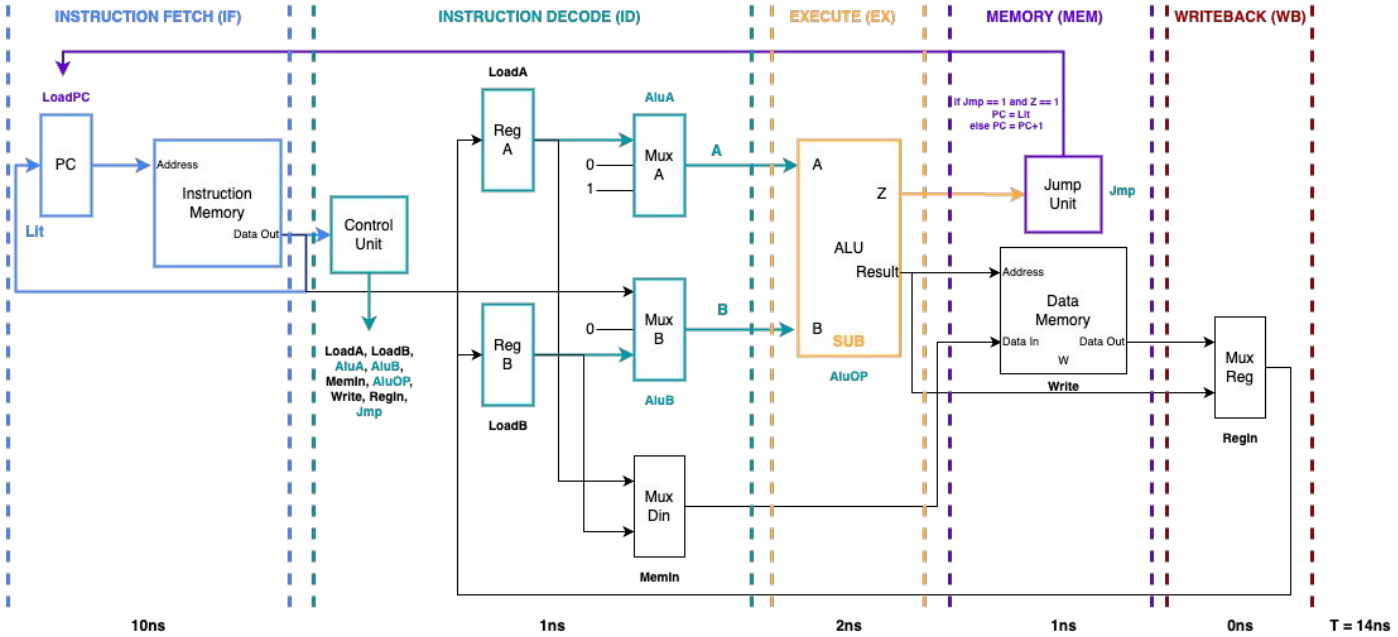
Paralelismo en la CPU - *Pipeline*

Ejemplo: `MOV (var1),A`



Paralelismo en la CPU - *Pipeline*

Ejemplo: `JEQ A,B,Lit`



Paralelismo en la CPU - *Pipeline*

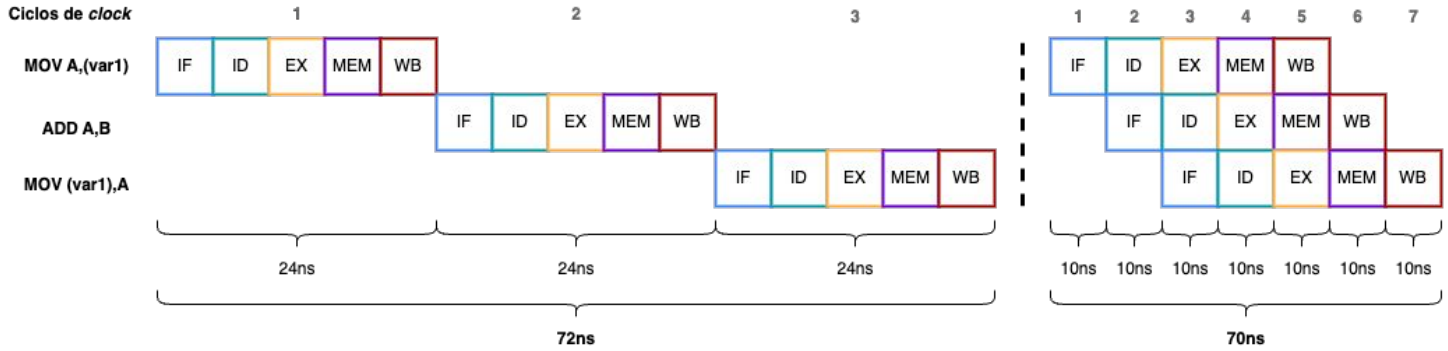
- Cada etapa puede tener una duración distinta, siendo las de mayor tiempo las que involucran lecturas/escrituras de memoria (IF, MEM). Además, no todas las instrucciones necesitan pasar por todas las etapas (por simplicidad, asumimos que sí).

- El *clock* del computador básico no puede ser más rápido que la instrucción más lenta. Con *pipeline*, **el *clock* no podrá ser más rápido que la etapa más lenta**.

A continuación, veremos la consecuencia de esto en términos de tiempo.

Paralelismo en la CPU - *Pipeline*

Ejemplo: Carga de trabajo de tres instrucciones sin/con *pipeline*



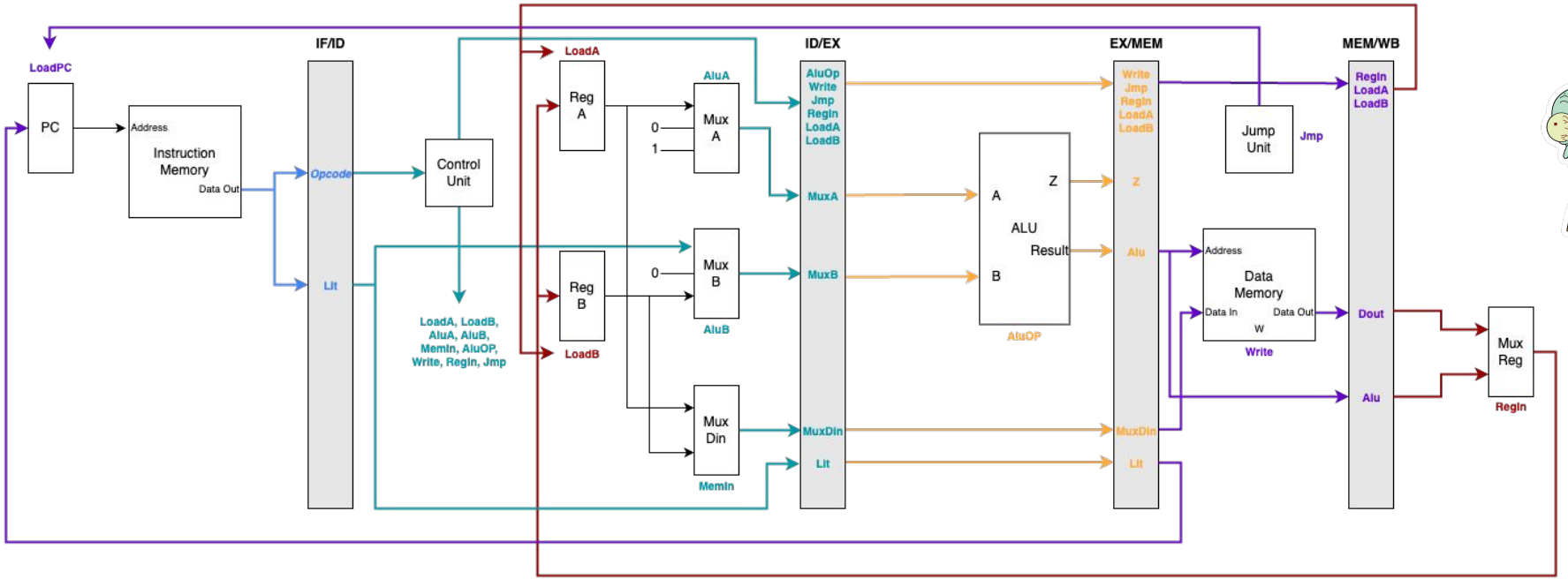
A pesar de ejecutar en más ciclos, la versión con *pipeline* es **más rápida**. ¿Cómo ocurre esto si con *pipeline* a una instrucción le toma más tiempo ejecutar?

Paralelismo en la CPU - *Pipeline*

- La ganancia real se encuentra en **la velocidad del *clock***.
- Como ahora el *clock* se adapta a la etapa más lenta y no a la suma de sus partes, es naturalmente más rápido. Así, cada instrucción espera **menos** para comenzar su ejecución.
- La mejora comienza a partir de la ejecución de 3 instrucciones en este ejemplo, pero esta crece **sustancialmente a medida que se ejecutan más instrucciones**.

¿Cómo implementamos el *pipeline* a nivel de *hardware*?

Computador básico con *pipeline*



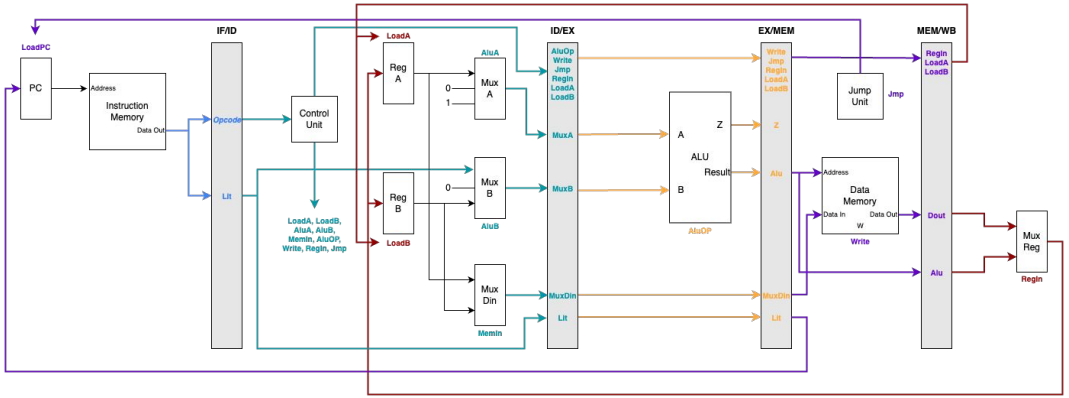


A continuación veremos cómo funciona.

Computador básico con *pipeline*

- Se agregan cuatro registros intermedios cuyos nombres definen entre qué etapas se encuentran: IF/ID; ID/EX; EX/MEM; MEM/WB.

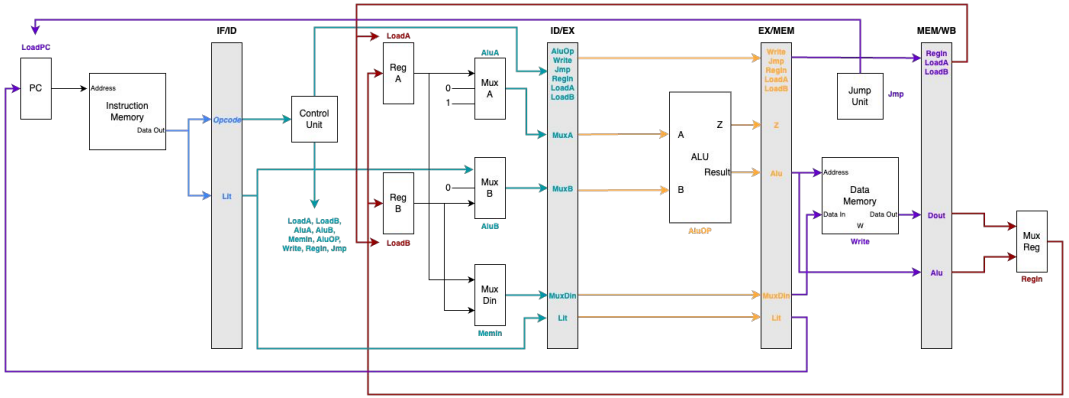
- Al ser registros, se actualizan en cada flanco de subida del *clock*.



Se asume que estos registros intermedios poseen la estructura para almacenar todas las señales indicadas, así como también las transmiten de la forma esperada en la siguiente etapa de la secuencia.

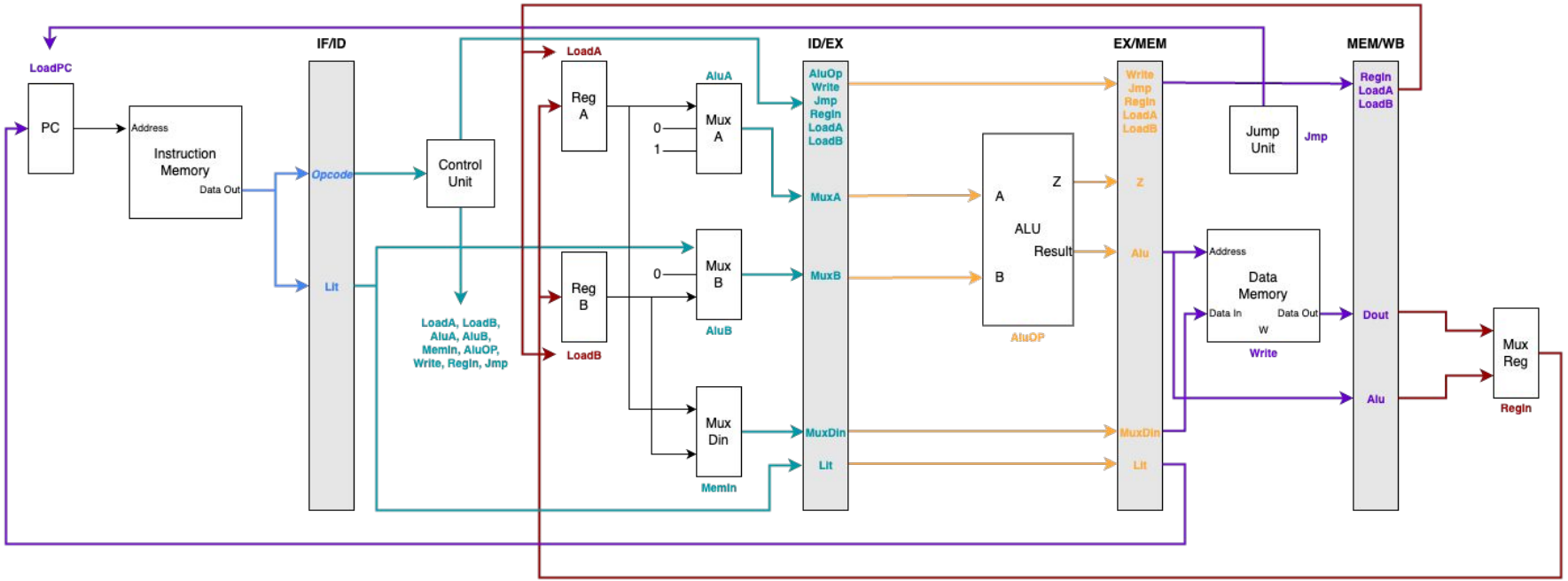
Computador básico con *pipeline*

- Cuando se ejecuta una instrucción, esta toma 5 ciclos en terminar y los “avances” de su ejecución (señales de control, salida de los multiplexores, ALU y memoria) se van propagando a los registros intermedios.



El color indica a la etapa que corresponde cada señal. Aquí destacamos, por ejemplo, que la señal `LoadPC` y el literal `Lit` que llegan al *Program Counter* vienen de la etapa MEM. Por otra parte, las cargas en los registros *A* y *B* se realizan con las señales `LoadA` y `LoadB` provenientes de la etapa WB. Estos respetan los valores de la instrucción original gracias al almacenamiento y propagación de los registros intermedios.

Computador básico con *pipeline*



Ahora que entendemos mejor la implementación, ¿funciona siempre?

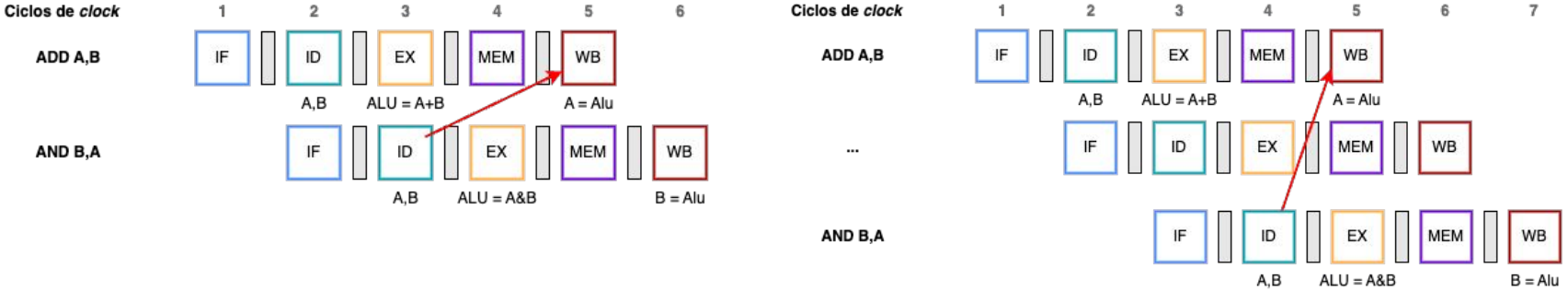
Computador básico con *pipeline* - *Hazards*

Al implementar un *pipeline*, pueden existir problemas por la **dependencia entre instrucciones**. Estos se conocen como ***hazards*** y existen tres tipos:

- *Hazards* de datos.
- *Hazards* de control.
- *Hazards* estructurales.

Computador básico con *pipeline* - *Hazards* de datos

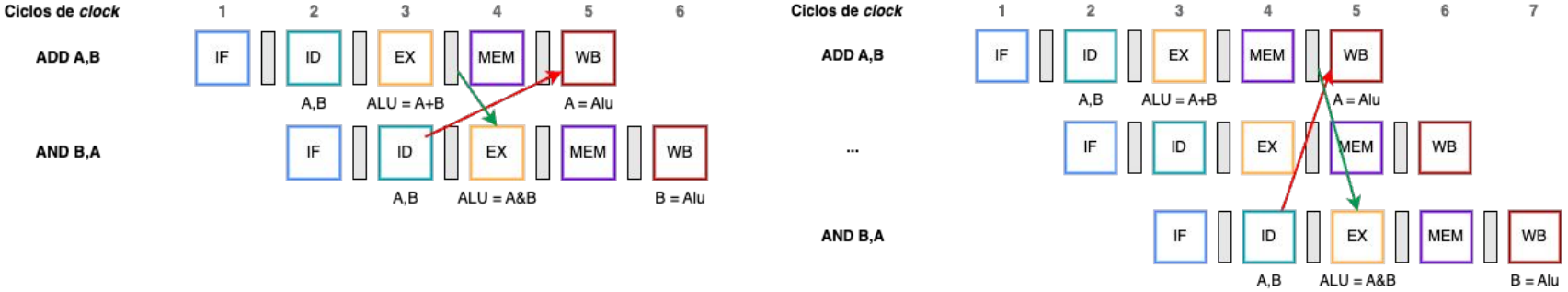
Estos ocurren cuando existen **dependencias de datos entre instrucciones**. Por ejemplo:



En ambos casos, la instrucción `AND B,A` no alcanza a utilizar el valor actualizado del registro *A* para obtener el resultado correcto. La flecha roja indica el punto en el que se selecciona el registro y apunta a la etapa en la que **recién** se actualiza al valor esperado.

Computador básico con *pipeline* - *Hazards* de datos

Solución: ***Forwarding***. Aprovechamos que el valor actualizado se calcula en la etapa EX de la primera instrucción, por lo que se encuentra almacenado en los registros intermedios.



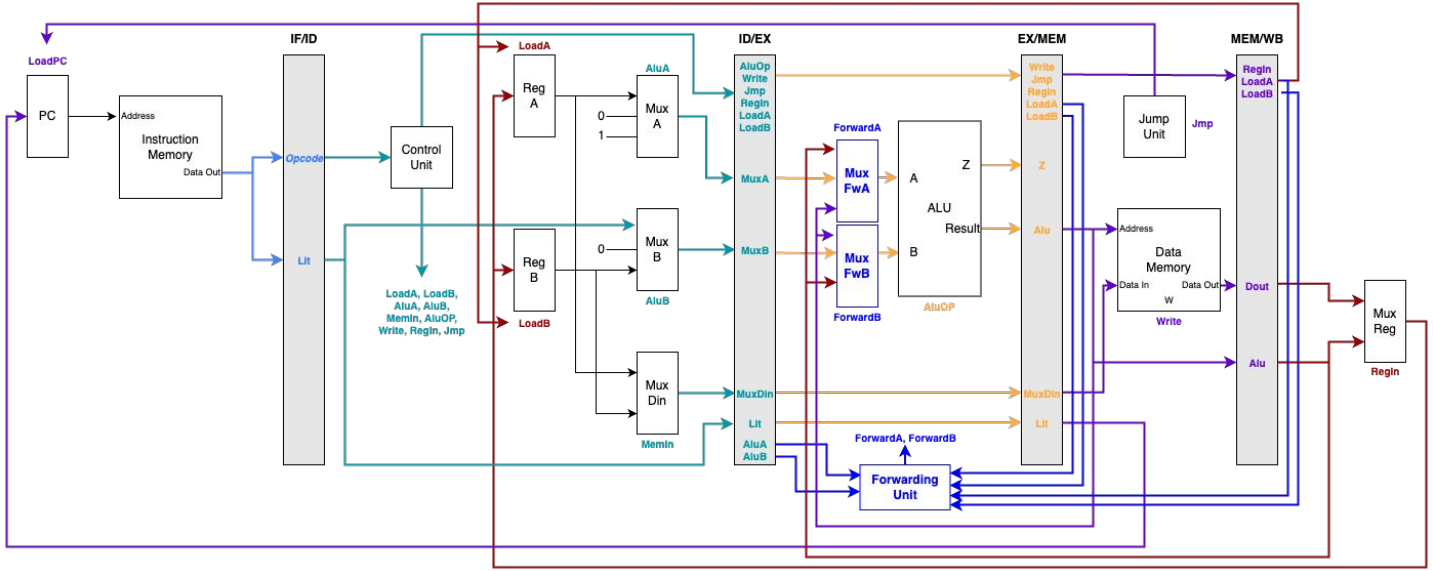
Como se requiere solo para la ejecución correcta en la ALU, se transmite (*forwarding*) desde el registro intermedio que corresponda hacia la etapa EX de la instrucción que posee la dependencia.

Computador básico con *pipeline* - *Hazards* de datos

Agregaremos una nueva unidad llamada ***Forwarding Unit*** que detectará los dos casos anteriores de la siguiente forma:

- `if (EX/MEM.LoadA == 1 and ID/EX.AluA == A)`
  `or (EX/MEM.LoadB == 1 and ID/EX.AluB == B)`
  → ***Forwarding*** **de `EX/MEM.Alu` a etapa EX según el registro.**
- `if (MEM/WB.LoadA == 1 and ID/EX.AluA == A and`
  `EX/MEM.LoadA == 0)`
  `or (MEM/WB.LoadB == 1 and ID/EX.AluB == B and`
  `EX/MEM.LoadB == 0)`
  → ***Forwarding*** **de `MEM/WB.Alu` a etapa EX según el registro.**

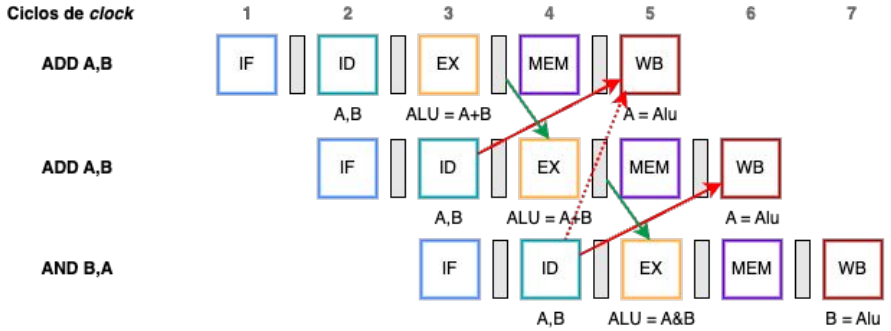
Computador básico con *pipeline* - *Hazards* de datos



Se agregan los multiplexores `FwA` y `FwB` para seleccionar entre los valores originales de la instrucción (`MuxA`, `MuxB`) o los transmitidos por la *Forwarding Unit* para resolver el *hazard* (`EX/MEM.Alu`, `MEM/WB.Alu`).

Computador básico con *pipeline* - *Hazards* de datos

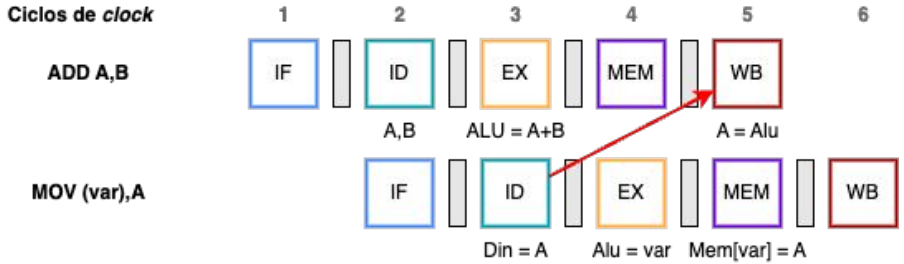
¿Y si hay dependencia entre las tres instrucciones?



Si bien existe dependencia entre la primera y la tercera instrucción, basta con hacer *forwarding* de la primera a la segunda y de la segunda a la tercera. Por lo tanto, ese *hazard* en particular no es considerado (indicado por la flecha roja punteada).

Computador básico con *pipeline* - *Hazards* de datos

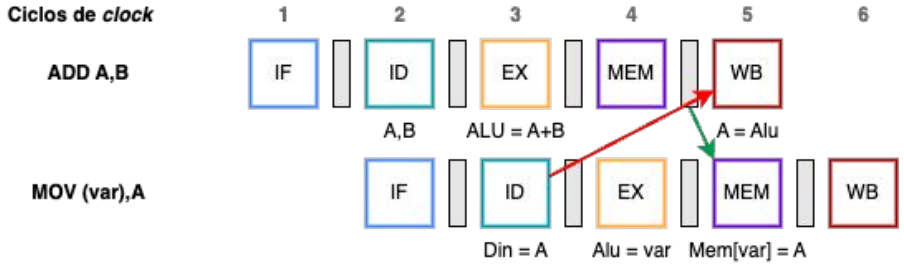
¿Es el único caso en el que tenemos dependencia de datos?



No. Falta considerar el caso en el que queremos escribir en memoria el valor de un registro actualizado recientemente. ¿Podemos resolver esto con *forwarding*?

Computador básico con *pipeline* - *Hazards* de datos

Sí, ya que poseemos el resultado calculado en `MEM/WB.Alu`.



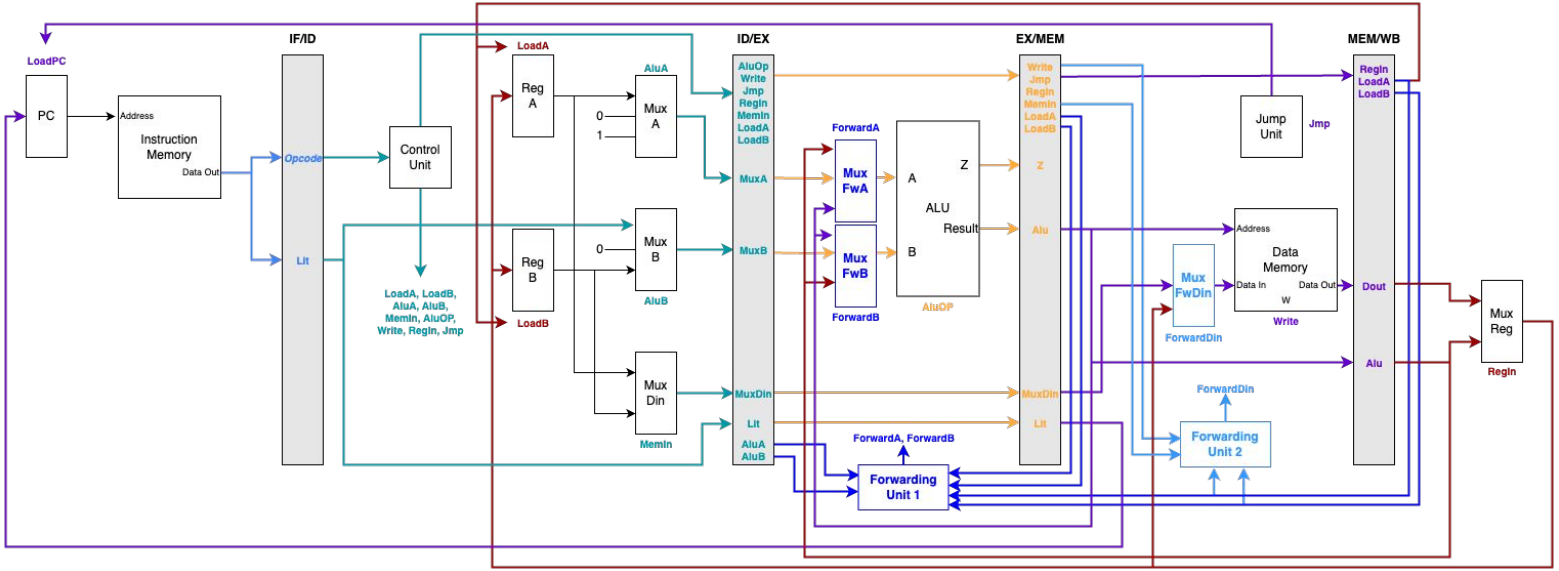
Ahora, en este caso debemos transmitir el resultado en la etapa MEM y no la etapa EX, por lo que debemos añadir **otra *Forwarding Unit***.

Computador básico con *pipeline* - *Hazards* de datos

Este caso adicional se detecta de la siguiente forma:

- `if (MEM/WB.LoadA == 1 and EX/MEM.MuxDin == A`
  `and EX/MEM.Write == 1)`
  `or (MEM/WB.LoadB == 1 and EX/MEM.MuxDin == B`
  `and EX/MEM.Write == 1)`
  → ***Forwarding*** **de `WB.RegIn` a etapa MEM según el registro.**

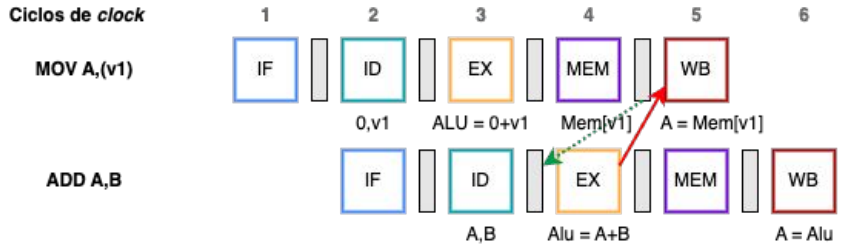
Computador básico con *pipeline* - *Hazards* de datos



Se agrega el multiplexor `FwDin` para seleccionar entre el resultado original a escribir (`MuxDin`) o el transmitido por la nueva *Forwarding Unit* para resolver el *hazard* (`WB.RegIn`).

Computador básico con *pipeline* - *Hazards* de datos

No siempre es posible resolver los *hazards* de datos con *forwarding*:

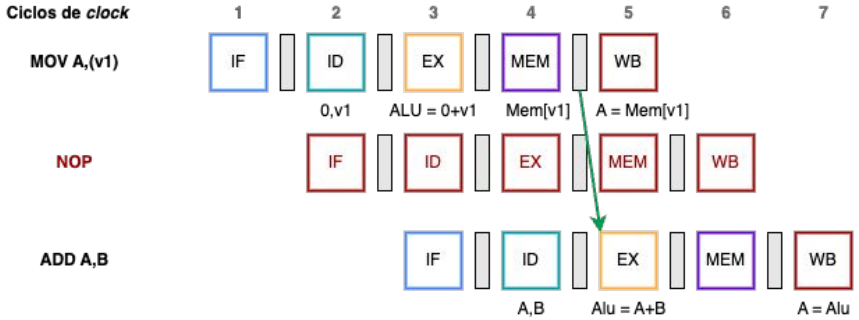


Al realizar una lectura de memoria y luego usar dicho resultado en la siguiente instrucción, se da el caso de que se termina la lectura del dato **al mismo tiempo que se realiza la ejecución de la siguiente instrucción en la ALU** (usando el valor erróneo).

¿Podemos resolver este caso?

Computador básico con *pipeline* - *Hazards* de datos

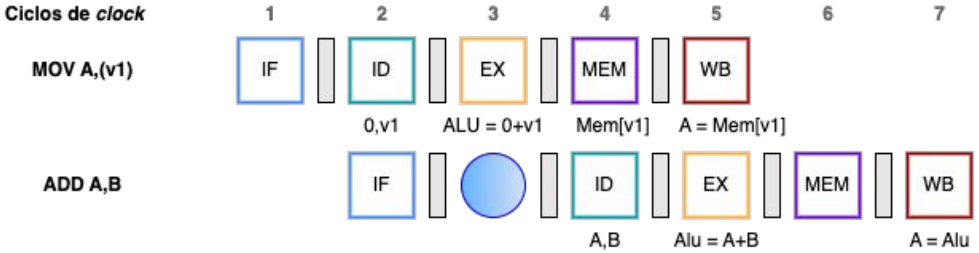
Solución 1: ***Stalling por software***. Podemos hacer que la instrucción que posee la dependencia "espere" por una iteración, para que posteriormente pueda obtener el valor por *forwarding*.



El compilador está a cargo de detectar estos casos e insertar instrucciones `NOP` entre los *hazards* de esta naturaleza para poder resolverlos.

Computador básico con *pipeline* - *Hazards* de datos

Solución 2: ***Stalling por hardware***. Sigue la misma idea, pero el *hardware* genera una "burbuja" en el *pipeline* para que, por una iteración, la instrucción con dependencia deje de avanzar.



Ahora, ¿cómo lo hacemos a través del *hardware*?

Computador básico con *pipeline* - *Hazards* de datos

Agregaremos una nueva unidad llamada ***Data Hazard Unit*** que detectará el caso anterior de la siguiente forma:

- ```
  if (ID/EX.LoadA == 1 and ID/EX.RegIn == Dout
      and ID.AluA == A)
  or (ID/EX.LoadB == 1 and ID/EX.RegIn == Dout
      and ID.AluB == B)
  ```
  → **Inserción de burbuja**

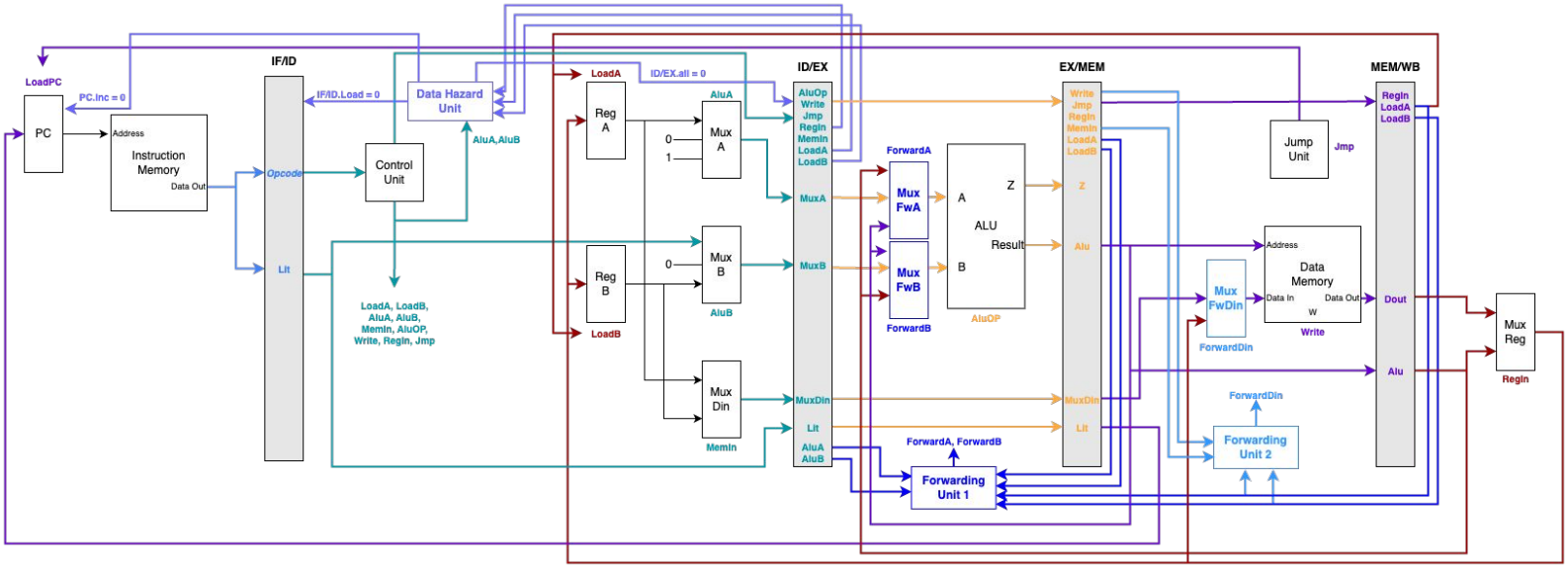
¿Qué realiza exactamente la “burbuja”?
  ```

Computador básico con *pipeline* - *Hazards* de datos

Inserción de una burbuja en *hardware*

- Se configuran en cero todas las señales de control del registro ID/EX. Esto permite que la instrucción no se propague.
- Se evita la escritura en el registro IF/ID para mantener la instrucción actual sin ser sobreescrita por la siguiente.
- Se evita que incremente el valor del *Program Counter*, deteniendo el avance del programa por una iteración.

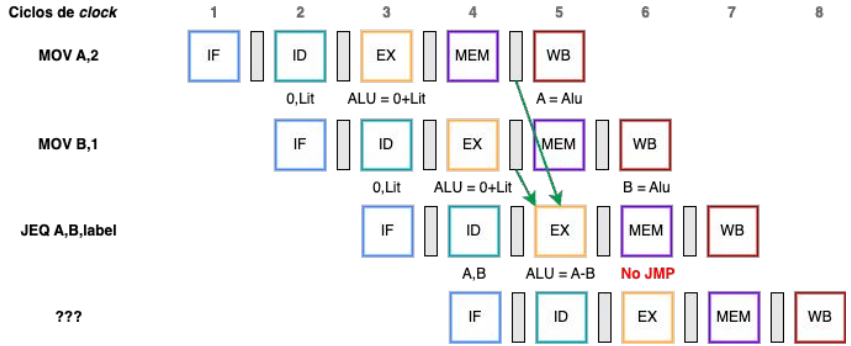
Computador básico con *pipeline* - *Hazards* de datos



Se agrega la unidad *Data Hazard Unit* y, con ella, las conexiones a los registros IF/ID, ID/EX y *Program Counter* para propagar las señales antes discutidas, generando la burbuja.

Computador básico con *pipeline* - *Hazards* de control

Estos ocurren con **las instrucciones de salto**. Por ejemplo:

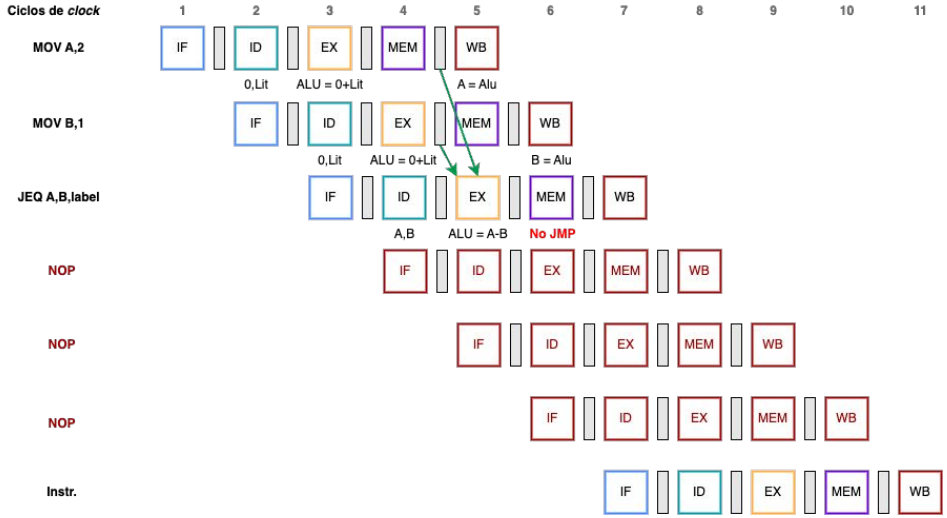


Se pueden resolver los *hazards* de datos tal como se muestra en la figura, no obstante, sabemos si corresponde el salto condicional recién en la etapa MEM de la instrucción con la unidad de salto.

¿Qué ejecutamos después y cómo lo decidimos?

Computador básico con *pipeline* - *Hazards* de control

Solución 1: *Stalling* hasta realizar o no el salto.



Si bien resuelve el problema, perdemos **siempre** tres ciclos por salto.
¿Qué pasaría si nuestro *pipeline* tuviera más etapas?

Computador básico con *pipeline* - *Hazards* de control

Agregar *stalling* siempre no es viable

- La cantidad de ciclos va en función de la profundidad del *pipeline*. Podríamos tener *pipelines* de 10 etapas y con una unidad de salto en la etapa 8. Aquí, perderíamos **7 ciclos por salto**.

- En programas reales existen muchos saltos y *pipelines* más complejos, por lo que en términos de eficiencia esta solución no es pertinente.

En la práctica, se utilizan **unidades de predicción de salto**.

Computador básico con *pipeline* - *Hazards* de control

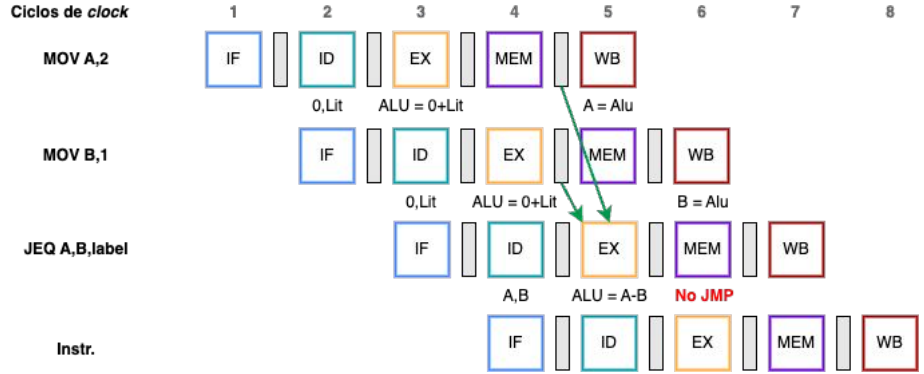
Unidad de predicción de salto

Según cierta probabilidad (en base a estudios estadísticos) el *hardware* decide si realizar el salto de inmediato (cargando el literal correspondiente en el *Program Counter*) o no (ejecutando las instrucciones siguientes, *i.e.* `PC+1`.

Esta implementación suele ser más compleja, por lo que para efectos pedagógicos implementaremos una unidad que **siempre asume que no hay salto**. Podría ser al revés (asumiendo que siempre hay salto), pero requiere de modificaciones más complejas.

Computador básico con *pipeline* - *Hazards* de control

Solución 2: Ejecución del programa asumiendo que no hay salto.

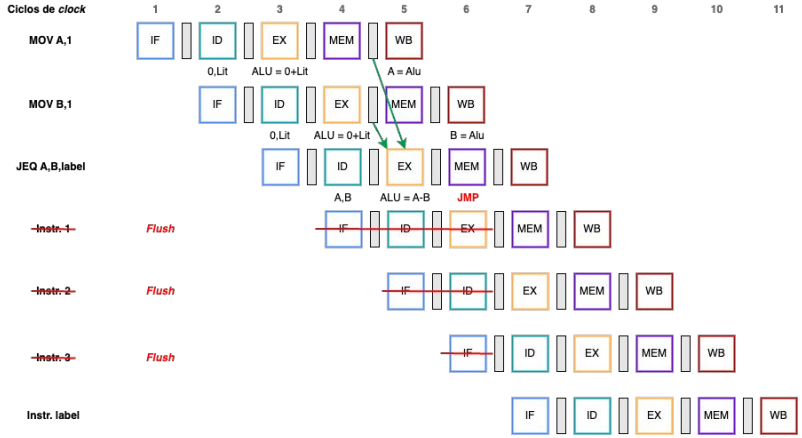


En este caso, predecimos correctamente que no hay que realizar el salto y el resto del programa se ejecuta en el *pipeline* sin problemas.

¿Qué pasa si nos equivocamos?

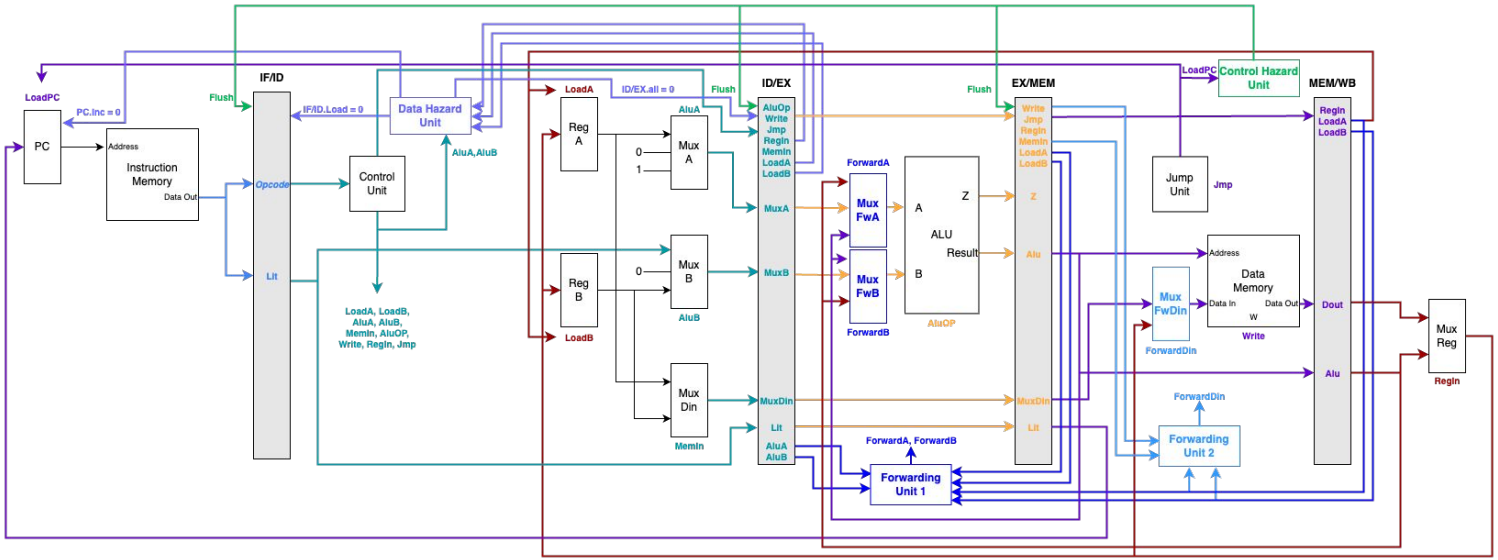
Computador básico con *pipeline* - *Hazards* de control

***Flushing*: Si nos equivocamos, "borramos" el avance de las instrucciones que alcanzaron a ejecutar y retomamos desde el salto.**



Esto sigue siendo mejor que aplicar *stalling* siempre, ya que no perdemos ciclos de ejecución. No obstante, complejiza el *hardware*: ¿Cómo lo podríamos implementar en el diagrama?

Computador básico con *pipeline* - *Hazards* de control



Se agrega la unidad *Control Hazard Unit* y, con ella, las conexiones a los registros IF/ID, ID/EX y EX/MEM para realizar *flush* y *setear* en 0 sus señales (anulando el avance de las instrucciones) si `LoadPC = 1`.

Computador básico con *pipeline* - *Hazards* de control

¿Qué pasa con los saltos incondicionales?

Por la implementación actual, estos tampoco son detectados **hasta la etapa MEM**. Por lo tanto, también caerán en la necesidad de realizar *flush* de las instrucciones ejecutadas erróneamente.

En nuestra implementación asumimos esta consecuencia, pero se podría arreglar con unidades adicionales que detecten el *opcode* de esta instrucción y, en dicho caso, carguen el literal en el *Program Counter* inmediatamente en la etapa IF.

Computador básico con *pipeline* - *Hazards* estructurales

Estos ocurren cuando dos etapas necesitan utilizar una misma unidad funcional.

- **Ejemplo:** Etapas IF y MEM en una arquitectura Von Neumann. Como en este tipo de arquitectura la memoria de datos e instrucciones se encuentran en una única unidad, estas dos etapas requieren acceder a ella en un mismo instante de tiempo.

- ¿Cómo se podría resolver este caso?

Computador básico con *pipeline* - *Hazards* estructurales

- **Solución 1:** Agregar *stalling* de forma intercalada a cada etapa para que haya un solo acceso por ciclo.

  Problema: Pérdida de 2 ciclos **por instrucción, siempre**.

- **Solución 2:** Agregar otra unidad de memoria.

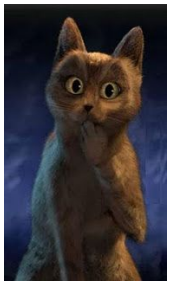
  Problema: Aumento significativo de costo, cambio estructural de la arquitectura.

Computador básico con *pipeline* - *Hazards* estructurales

Solución real: Hacer uso de una caché *split*.

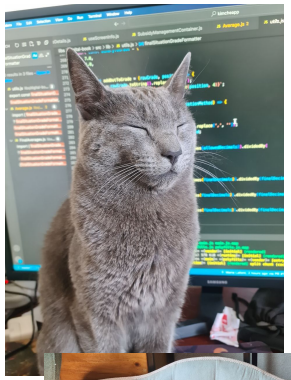
¿Por qué esto resuelve el problema?

- Las caché *split* tienen la particularidad de permitir dos accesos de memoria concurrentes: uno para la sección de datos y otra para la sección de instrucciones.
- Así, el *pipeline* trata los accesos de memoria como una arquitectura Harvard, aunque sea Von Neumann.

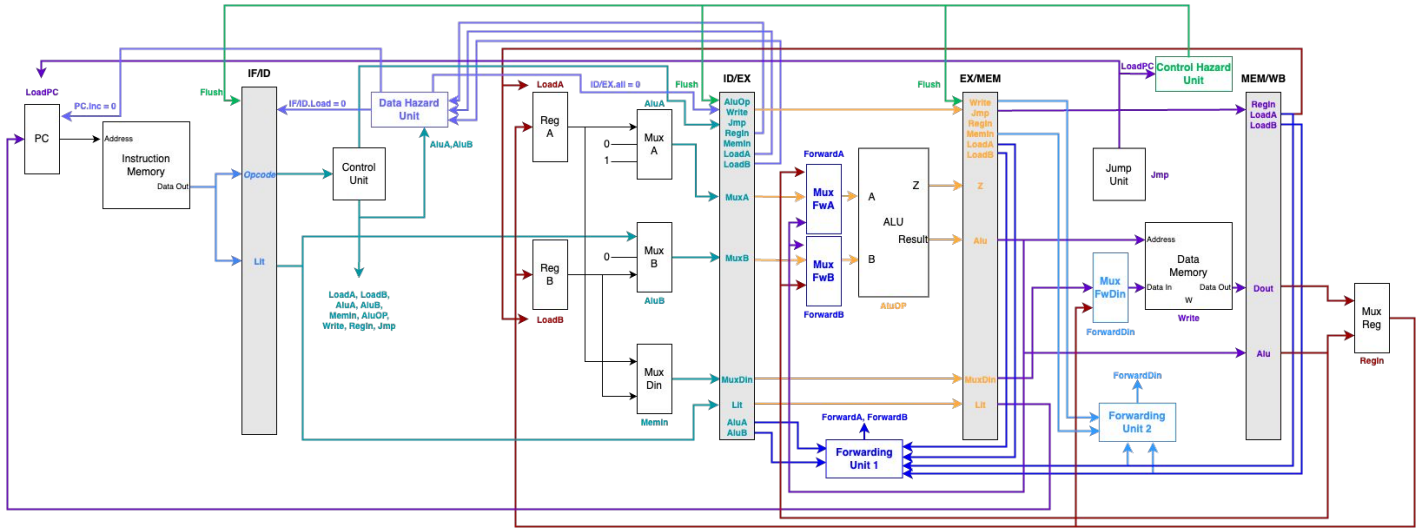


Computador básico con *pipeline*

Hemos llegado a la versión final del computador básico con *pipeline*.







Computador básico con *pipeline* - ISA

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,B	A=B		-
	B,A	B=A		-
	A,Lit	A=Lit		MOV A,15
	B,Lit	B=Lit		MOV B,15
	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,B	A=A+B		-
	B,A	B=A+B		-
	A,Lit	A=A+Lit		ADD A,5
SUB	A,B	A=A-B		-
	B,A	B=A-B		-
	A,Lit	A=A-Lit		SUB A, 2
AND	A,B	A=A and B		-
	B,A	B=A and B		-
	A,Lit	A=A and Lit		AND A,15
OR	A,B	A=A or B		-
	B,A	B=A or B		-
	A,Lit	A=A or Lit		OR A,5
NOT	A,A	A=notA		-
	B,A	B=notA		-
XOR	A,A	A=A xor B		-
	B,A	B=A xor B		-
	A,Lit	A=A xor Lit		XOR A,15
SHL	A,A	A=shift left A		-
	B,A	B=shift left A		-
SHR	A,A	A=shift right A		-
	B,A	B=shift right A		-
INC	B	B=B+1		-
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
NOP				

De este listado de instrucciones cabe destacar que:

- Ya no hay instrucciones de operación que utilicen el *output* de memoria.
- Si bien se asume que los saltos se hacen a partir de un `CMP A,B`, en los ejemplos se explicitan ya que es factible hacerlos a través de un `CMP A,Lit` a nivel de *hardware*.

Computador básico con *pipeline* - Ejercicio en clases

Considere el siguiente código escrito para el Assembly del computador básico con *pipeline*:

```
DATA:
  n 2
CODE:
  MOV A,0
  main:
    MOV B,(n)
    JEQ A,B,end
    SUB B,1
    MOV (n),B
    JMP main
  end:
    NOP
    NOP
    NOP
```

Indique con un diagrama, si es que ocurre y cuándo ocurre *forwarding*, *stalling* y *flushing* al ejecutar el código anterior y cómo lo detectan las *forwarding units* correspondientes, especificando las señales de control participantes.

Considere que el *pipeline* tiene *forwarding* entre todas sus etapas, el manejo de *stalling* es por *software* con la instrucción `NOP` y que la predicción de salto se realiza asumiendo que no ocurre.

Computador básico con *pipeline* - Ejercicio en clases

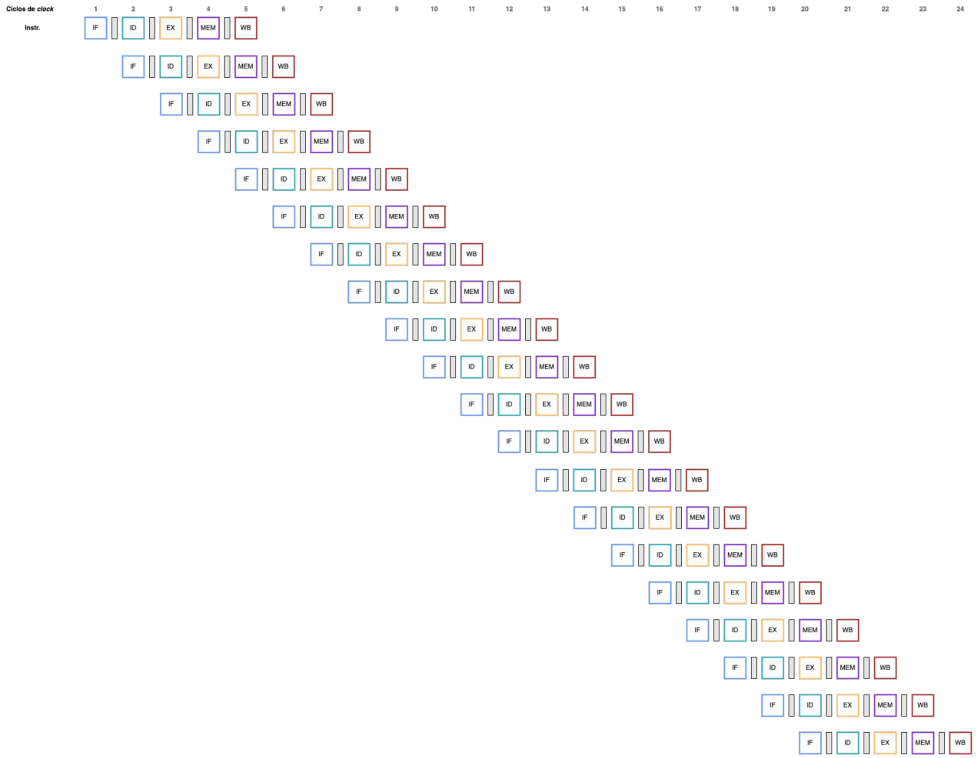
Primero, veamos el código que se ejecutará en el *pipeline*:

```
MOV A,0
MOV B,(n)
NOP              # Stalling por software
JEQ end          # Salto no realizado de forma correcta.
SUB B,1
MOV (n),B
JMP main         # Salto no realizado de forma incorrecta. Se hará flush de las siguientes tres instrucciones.
NOP              # Flushing
NOP              # Flushing
NOP              # Flushing
MOV B,(n)        # Instrucción desde el salto.
NOP              # Stalling por software
JEQ end          # Salto no realizado de forma correcta.
SUB B,1
MOV (n),B
JMP main         # Salto no realizado de forma incorrecta. Se hará flush de las siguientes tres instrucciones.
NOP              # Flushing
NOP              # Flushing
NOP              # Flushing
MOV B,(n)        # Instrucción desde el salto.
NOP              # Stalling por software
JEQ end          # Salto no realizado de forma incorrecta. Se hará flush de las siguientes tres instrucciones.
SUB B,1          # Flushing
MOV (n),B        # Flushing
JMP main         # Flushing
NOP              # Instrucción desde el salto.
NOP
NOP
```

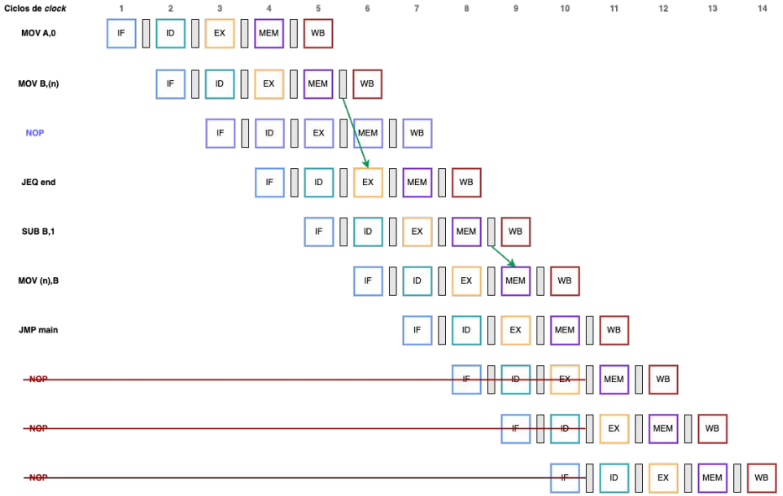
Computador básico con *pipeline* - Ejercicio en clases

Ahora, nos falta señalar los instantes en los que se realiza *forwarding*.

En este tipo de problema, nos basamos en el tipo de diagramas de la figura para visualizar de mejor manera todas las acciones de corrección realizadas.



Computador básico con *pipeline* - Ejercicio en clases



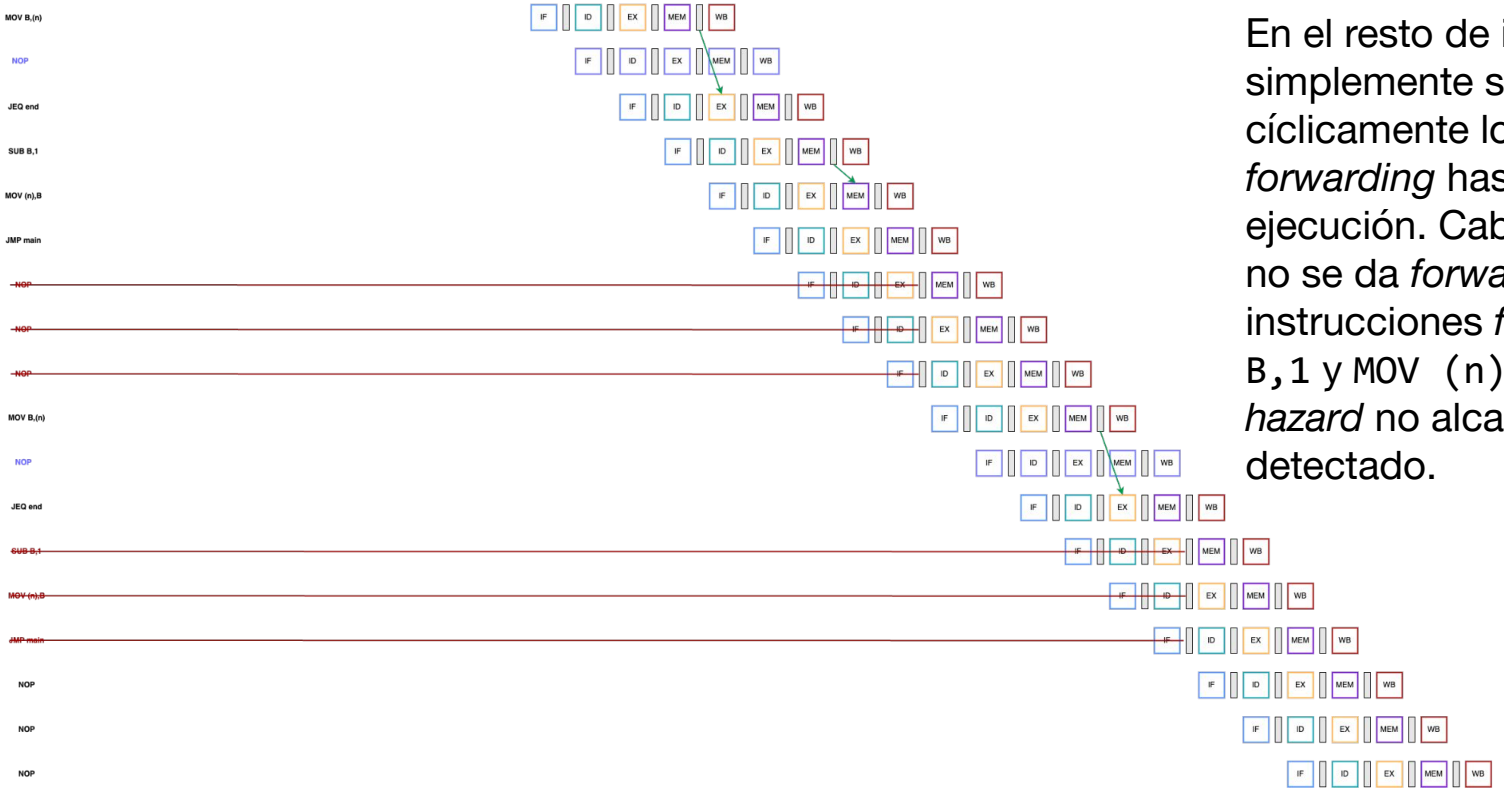
En las primeras 10 instrucciones, destacamos el primer *forwarding* entre `MOV B,(n)` y `JEQ end` detectado por la *Forwarding Unit* 1:

- `MEM/WB.LoadB == 1`
- `ID/EX.AluB == B`
- `EX/MEM.LoadB == 0`

El segundo *forwarding* se da entre `SUB B,1` y `MOV (n),B`, detectado por la *Forwarding Unit* 2:

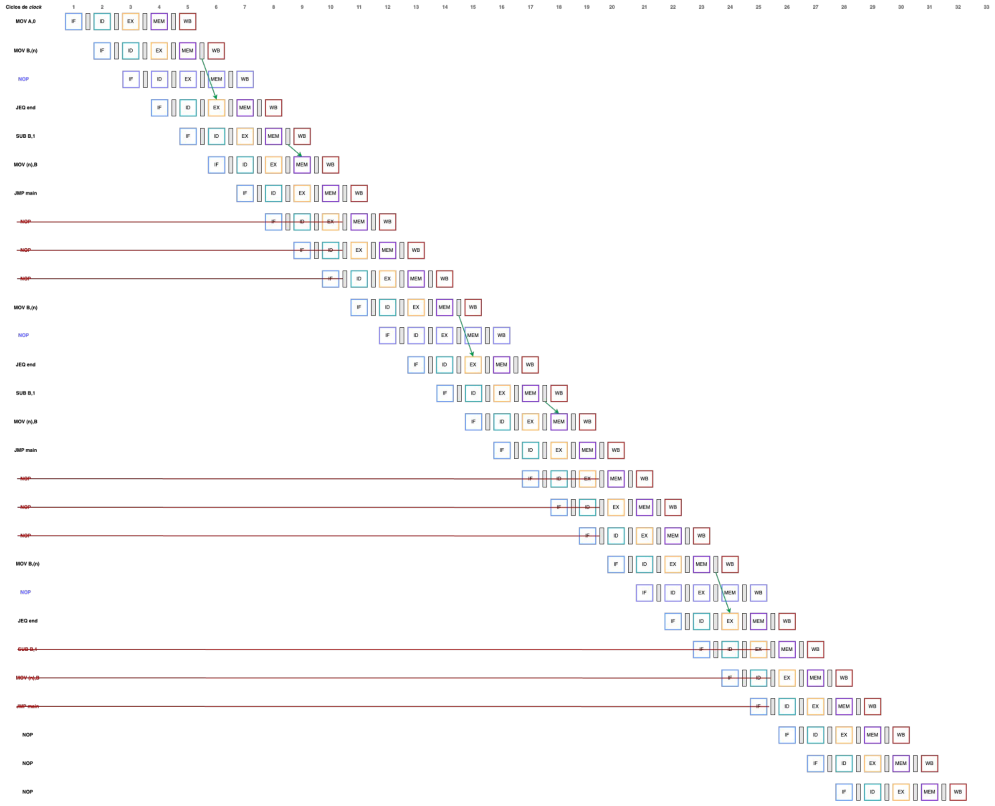
- `MEM/WB.LoadB == 1`
- `EX/MEM.MemDin == B`
- `EX/MEM.Write == 1`

Computador básico con *pipeline* - Ejercicio en clases



En el resto de instrucciones simplemente se repiten cíclicamente los mismos *forwarding* hasta el fin de la ejecución. Cabe destacar que no se da *forwarding* entre las instrucciones *flusheadas* `SUB B,1` y `MOV (n),B` ya que su *hazard* no alcanza a ser detectado.

Computador básico con *pipeline* - Ejercicio en clases



Resultado final



Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.



Ejercicios

Un computador con microarquitectura avanzada posee un *pipeline* de 30 etapas. ¿Cuál es el elemento de *hardware*, sin contar los registros entre etapas, que tiene el mayor impacto (positivo y negativo) en el rendimiento? Justifique su respuesta.

***Examen*, 2013-1**

Ejercicios

En promedio, ¿cuántos ciclos por salto pierde un computador con un *pipeline* de 12 etapas, si su unidad predictora acierta el 75 % de las veces? Asuma que los saltos se realizan en la penúltima etapa.

***Interrogación* 3, 2016-2**

Ejercicios

Estime el tiempo de ejecución de un programa que toma *N* instrucciones en el computador básico con *pipeline*, en base a las siguientes condiciones:

- El 50 % de las instrucciones realizan lecturas o escrituras en memoria.
- El 10 % de las instrucciones son de salto y en el 25 % de estas, el salto finalmente se realiza.
- En el 50% de las ocasiones, el dato obtenido desde la memoria debe ser utilizado en la instrucción siguiente.

Cualquier supuesto sobre la solución debe quedar claramente explicado.

***Interrogación* 3, 2016-1**

Ejercicios

Determine el número de ciclos que se demora el siguiente código, detallando en un diagrama los estados del *pipeline* por instrucción. El *pipeline* tiene *forwarding* entre todas sus etapas, el manejo de *stalling* es por *software* (instrucción `NOP`) y predicción de salto asumiendo que no ocurre. Indique en el diagrama cuando ocurre *forwarding*, *stalling* y *flushing*.

```
DATA:
    n 1
    index 1
    prev1 0
    prev2 1
    res 0
CODE:
    main:
        MOV A,(n)
        MOV B,(index)
        JEQ end
        ADD B,1
        MOV (index),B
        JMP main
    end:
        MOV A,(prev1)
        MOV B,(prev2)
        ADD A,B
        MOV (res),A
```

***Interrogación* 3, 2016-1**

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?





DCC
DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 12 - Paralelismo a Nivel de Instrucción (ILP)

Profesor: Germán Leandro Contreras Sagredo

Anexo - Resolución de ejercicios

¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.



Ejercicios - Respuesta

Un computador con microarquitectura avanzada posee un *pipeline* de 30 etapas. ¿Cuál es el elemento de *hardware*, sin contar los registros entre etapas, que tiene el mayor impacto (positivo y negativo) en el rendimiento? Justifique su respuesta.

Al poseer una cantidad considerable de etapas, el elemento más crítico, a la larga, será la unidad predictora de saltos. Si se implementa mal (*i.e.* posee un bajo porcentaje de acierto en las predicciones), se podrían perder muchos ciclos a medida que se avanza en las etapas.

Ejercicios - Respuesta

En promedio, ¿cuántos ciclos por salto pierde un computador con un *pipeline* de 12 etapas, si su unidad predictora acierta el 75 % de las veces? Asuma que los saltos se realizan en la penúltima etapa.

Si los saltos se realizan en la etapa 11 (correspondiente a la penúltima etapa), entonces son 10 las instrucciones que se podrían perder por una mala predicción. Entonces, si se falla en el 25 % de las veces, se pierde un total de $10 * 0{,}25 = 2{,}5$ ciclos por salto.

Ejercicios - Respuesta

Estime el tiempo de ejecución de un programa que toma *N* instrucciones en el computador básico con *pipeline*, en base a las siguientes condiciones:

- El 50 % de las instrucciones realizan lecturas o escrituras en memoria.
- El 10 % de las instrucciones son de salto y en el 25 % de estas, el salto finalmente se realiza.
- En el 50% de las ocasiones, el dato obtenido desde la memoria debe ser utilizado en la instrucción siguiente.

Cualquier supuesto sobre la solución debe quedar claramente explicado.

Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

De partida, con N instrucciones, se tendrá un total de $N + 4$ ciclos si no se consideran las condiciones anteriores. Luego, se van añadiendo más iteraciones según los siguientes criterios:

- Considerando el *pipeline* visto en clases, tenemos que se agregan 3 ciclos por cada salto realizado (correspondientes a los que se les hizo *flush*) si se asume una unidad predictora que siempre opta por no saltar. Como se tiene un 10 % de instrucciones de salto y un 25 % de estos se realiza, entonces se añade un total de $0{,}1 * 0{,}25 * 3 * N = 0{,}075N$ ciclos al total.
- Como el 50 % de las instrucciones acceden a memoria y en el 50 % de estos casos el dato obtenido de memoria se usa en la siguiente instrucción, en $0{,}5 * 0{,}5 * N = 0{,}25N$ instrucciones se necesita hacer *stalling* (por lo que se pierde una cantidad de ciclos igual a la cantidad de veces que se realiza esta espera, pues solo se pierde un ciclo por burbuja insertada).

Finalmente, tenemos que el tiempo de ejecución final es:

$$t \times ((N + 4) + (0{,}075N) + (0{,}25N)) = 4 \times t + 1{,}325N \times t$$

* ***t*** **= Tiempo de ejecución de un ciclo en el computador básico con *pipeline***

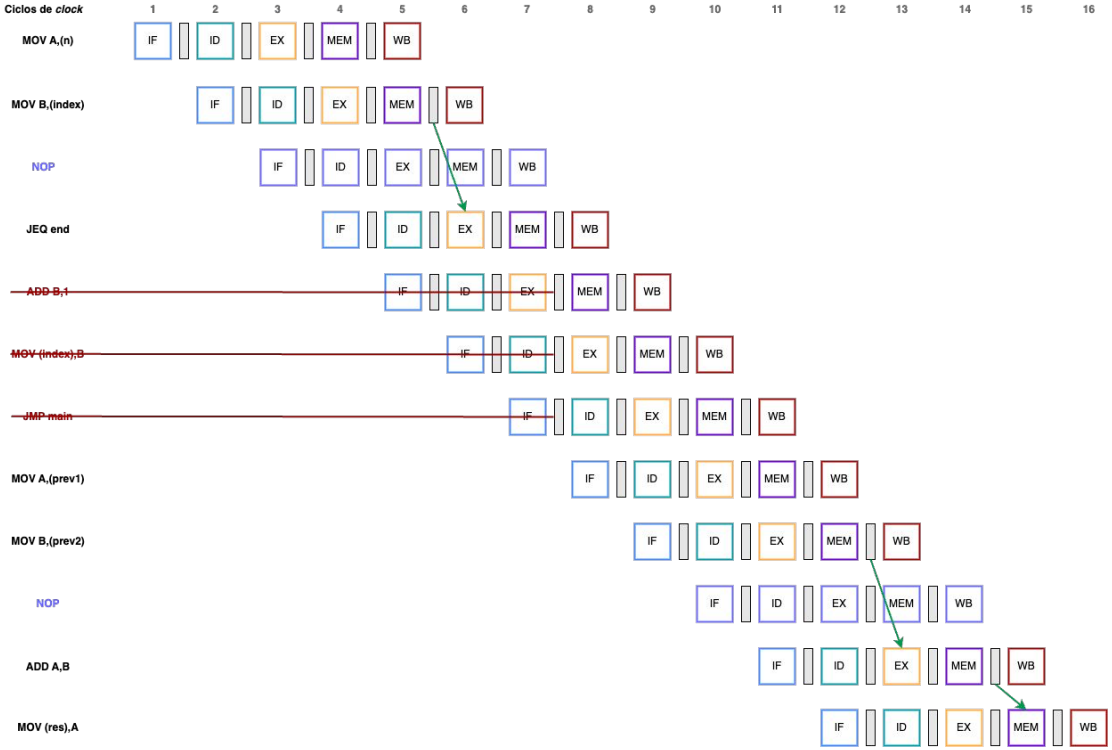
Ejercicios - Respuesta

Determine el número de ciclos que se demora el siguiente código, detallando en un diagrama los estados del *pipeline* por instrucción. El *pipeline* tiene *forwarding* entre todas sus etapas, el manejo de *stalling* es por *software* (instrucción `NOP`) y predicción de salto asumiendo que no ocurre. Indique en el diagrama cuando ocurre *forwarding*, *stalling* y *flushing*.

```
DATA:
    n 1
    index 1
    prev1 0
    prev2 1
    res 0
CODE:
    main:
        MOV A,(n)
        MOV B,(index)
        JEQ end
        ADD B,1
        MOV (index),B
        JMP main
    end:
        MOV A,(prev1)
        MOV B,(prev2)
        ADD A,B
        MOV (res),A
```

Respuesta en la siguiente diapositiva

Ejercicios - Respuesta



*** Del diagrama se deduce que la ejecución se realiza en 16 ciclos.**