Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Paralelismo a nivel de instrucción (ILP)

1 Motivación

Las principales mejoras en eficiencia de los computadores no ocurren sólo por avances en la tecnología de construcción de estos, sino también por el desarrollo de técnicas que permiten aprovechar procesamiento paralelo en el computador. Existen distintos niveles a los cuales se puede aprovechar el paralelismo, siendo el más básico el paralelismo a nivel de la instrucción.

2 Ciclo de la instrucción

Cada instrucción que se ejecuta un computador pasa por un ciclo, que va desde que es seleccionada desde memoria hasta que completa su objetivo. Este ciclo, aunque es similar en todas las arquitecturas de computadores, presenta algunas variaciones, dependiendo de la complejidad de la microarquitectura y de decisiones de diseño. El computador básico Harvard (figura 1), por ejemplo, tendrá un cierto ciclo, pero este no será exactamente igual al de una arquitectura x86 o de un PIC16F87AA.

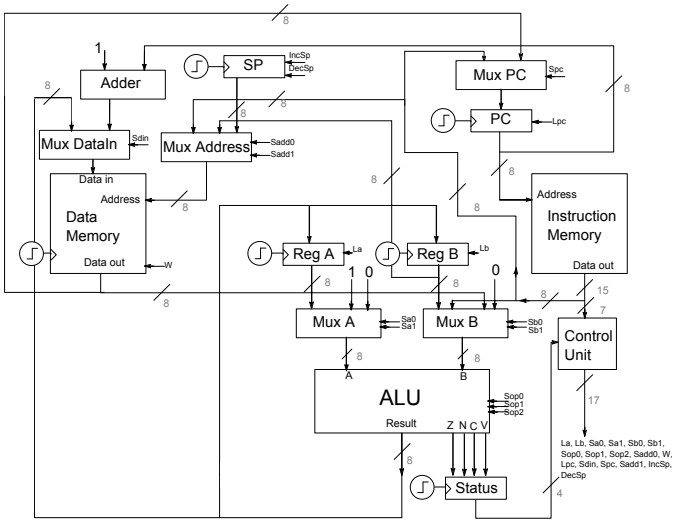


Figura 1: Diagrama computador básico.

El ciclo actual de una instrucción en el computador básico Harvard que se ha estudiado, aunque es simple, presenta algunos elementos que harán innecesariamente difícil el análisis que realizaremos es este capítulo. Para evitar estos problemas y trabajar con un ciclo más simple, se usará una versión simplificada del computador básico, la cual se muestra en la figura 2.

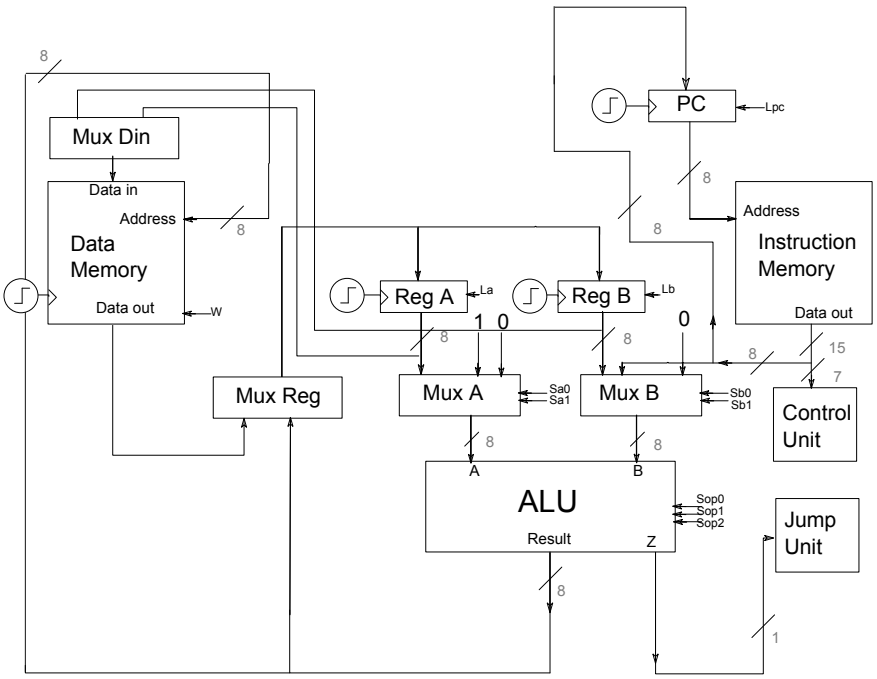


Figura 2: Diagrama computador básico modificado.

Las principales diferencias son las siguientes:

- Se elimina el soporte de stack, eliminando el Stack Pointer y las conexión del PC con memoria.
- Se simplifican los saltos condicionales, soportando ahora solamente el condition code Z, y las instrucciones `JEQ` y `JNE`. Adicionalmente, esta instrucción realizará la comparación $(A - B)$ y el salto en un mismo ciclo, por lo que se elimina también el Status Register.
- Se elimina la conexión entre la salida de memoria y el MUX B. Ahora la salida de memoria no se puede ocupar como parámetro en la ALU, y solo puede ser usada para cargar los registros A y B, es decir sólo se soportan instrucciones de transferencia desde la memoria.
- La entrada de datos de la memoria ahora sólo puede provenir de los registros A y B, y no de la ALU, es decir sólo se soportan instrucciones de transferencia hacia la memoria
- La dirección de la memoria ahora proviene de la ALU, lo que permite los mismos direccionamientos que antes (directo, indirecto por registro B), y agrega otros tipos de direccionamiento (indirecto por registro A, indirecto con índice, etc.)
- La unidad de salto, originalmente parte de la unidad de control encargada de determinar si corresponde un salto condicional, se separó de la unidad de control.

La idea detrás de estas modificaciones es, por un lado, simplificar el computador, pero también lograr que el ciclo de las instrucciones sea más uniforme que en la versión original. Con estas modificaciones es posible reorganizar las partes del computador, para mostrar con más claridad las distintas partes del ciclo, lo que se observa en la figura 3.

El ciclo de la instrucción para este computador contará de 5 etapas: **instruction fetch**, **instruction decode**, **execute**, **memory** y **writeback**.

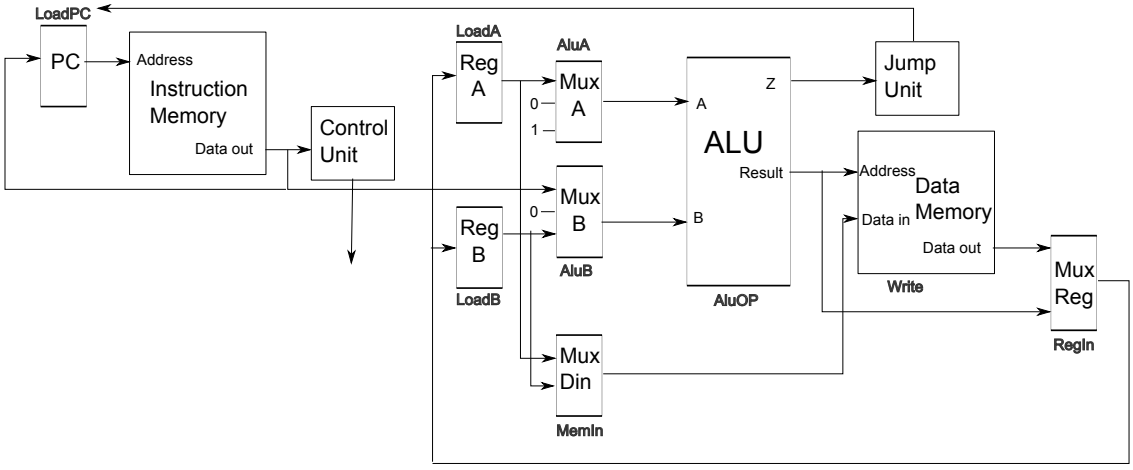


Figura 3: Diagrama computador básico modificado reordenado.

Instruction fetch (IF)

Corresponde a la primera etapa del ciclo, en la cual se va a buscar a la memoria de instrucciones la siguiente instrucción, apuntada por el valor actual del Program Counter. La salida de esta etapa es la instrucción obtenida desde memoria, la cual se separará en un opcode y en el parámetro.

Instruction decode (ID)

La segunda etapa del ciclo corresponde al instruction decode, es decir la decodificación de la instrucción y la generación de las señales de control. La unidad de control se encargará de transformar el opcode de la instrucción obtenida en las señales de control específicas que le indicarán al procesador que tarea ejecutar.

Execute (EX)

La tercera etapa corresponde a la ejecución, la cual es realizada mediante la ALU. Esta, como única unidad de ejecución en este computador, se encargará de realizar la operación seleccionada en la etapa previa, con los parámetros seleccionados también por las señales de control, obteniendo un resultado. Adicionalmente, la ALU también generará el condition code Z, dependiendo del resultado de la operación obtenida.

Memory (MEM)

La cuarta etapa corresponde a la lectura o escritura en memoria de datos. Esta etapa solo estará presente si a la instrucción ejecutada le correspondía transferencia hacia o desde memoria (por ejemplo la instrucción `MOV (var1), A`). En esta etapa se podrá haber escrito en memoria u obtenido un dato de esta para luego ser almacenado en un registro.

Adicionalmente al manejo de memoria, en esta etapa también se incluirá la definición si corresponde saltar o no, por parte de la unidad de salto. Aunque este proceso no tiene relación con la memoria de datos, se incluye en esta etapa debido a que solo luego de ejecutar la operación en la ALU se podrá tener la información necesaria para esta decisión.

Writeback (WB)

La última etapa del ciclo también estará presente sólo en algunas instrucciones. En esta etapa de writeback, corresponderá escribir en los registros, ya sea un resultado de la ALU o un dato obtenido desde memoria (por ejemplo las instrucciones `ADD A,B` y `MOV A, (var1)`

El resumen de las 5 etapas, indicando las unidades funcionales asociadas a cada una, se observa en la figura 4.

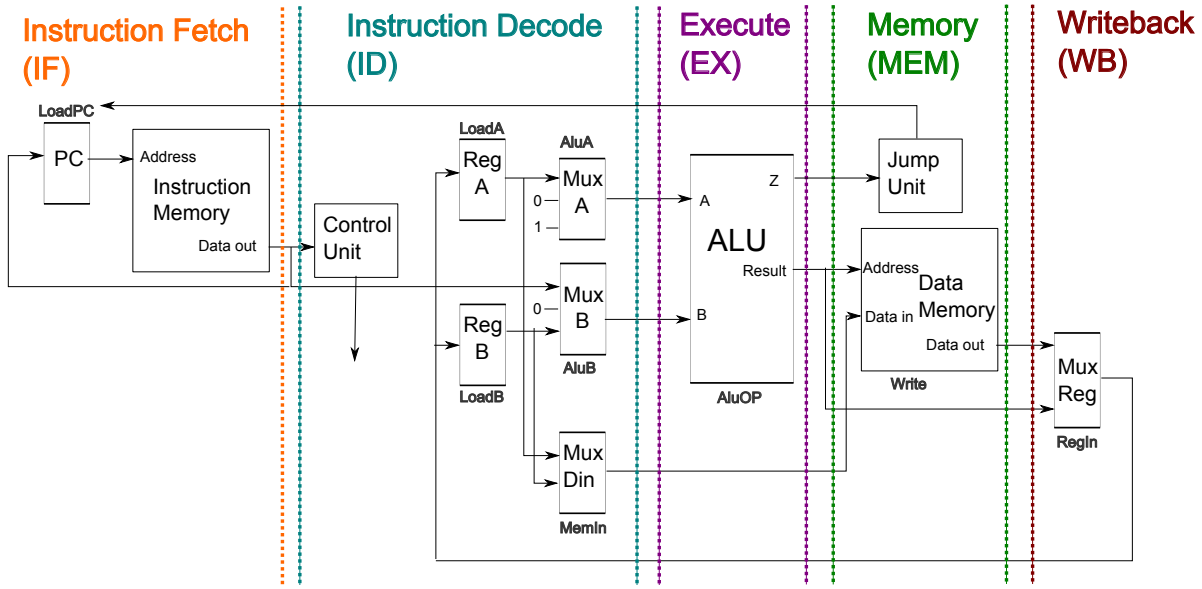


Figura 4: Ciclo de la instrucción.

A continuación se presentan una serie de ejemplos con los ciclos de distintas instrucciones. En cada caso se muestra el tiempo que tomaría hipotéticamente en completarse cada fase, y el tiempo total que tomaría la instrucción:

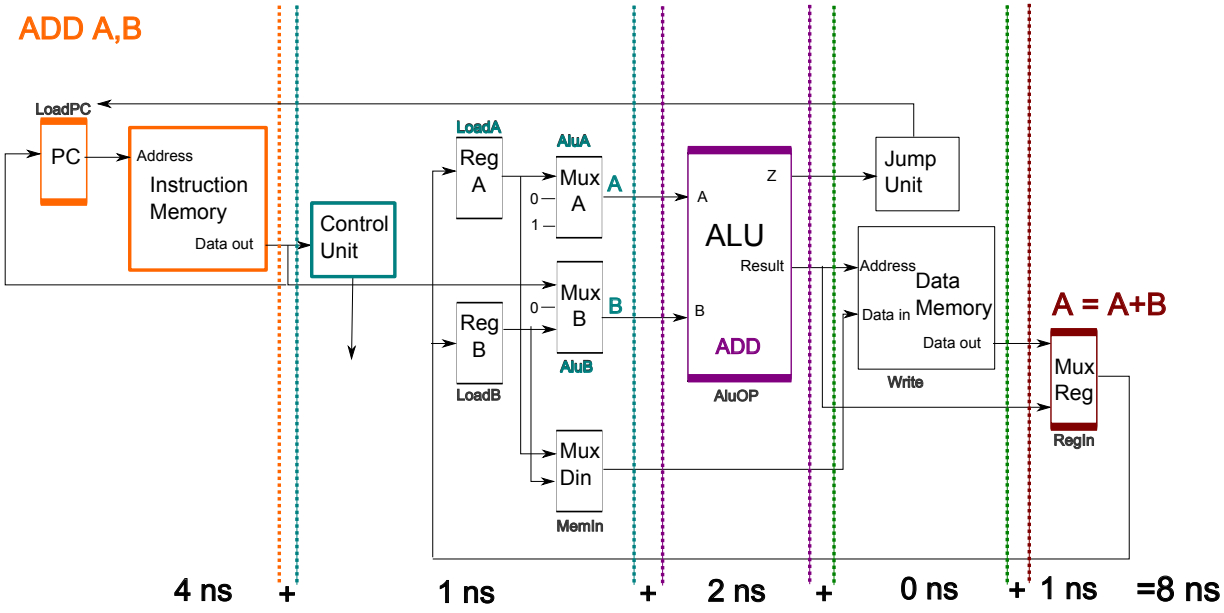


Figura 5: Ciclo de instrucción ADD A,B

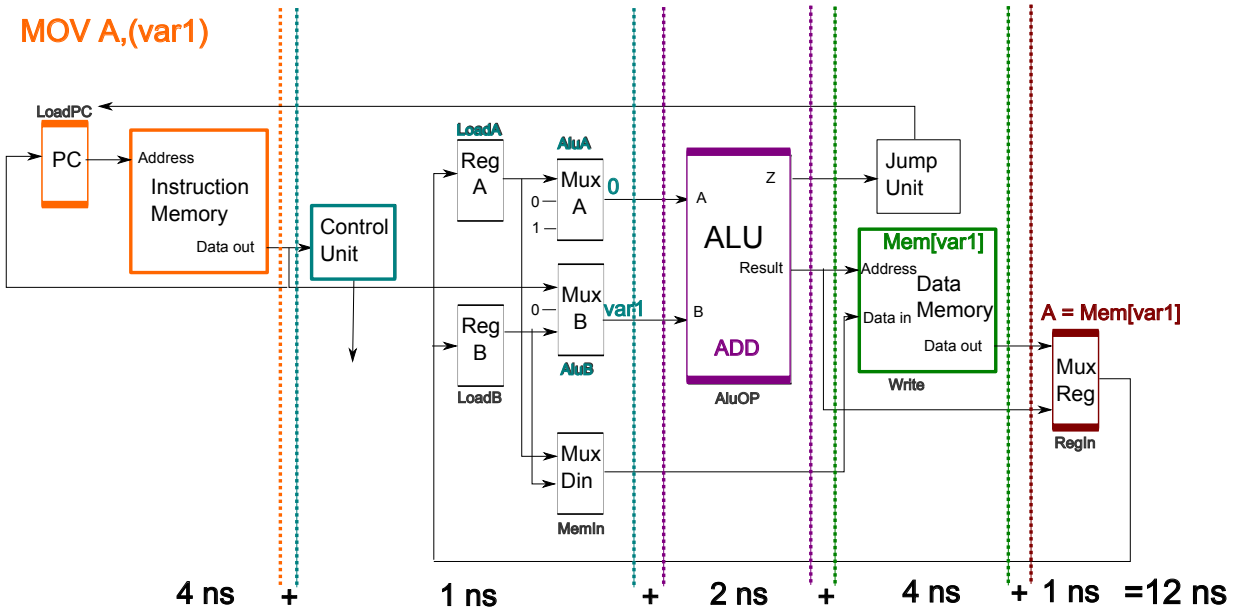


Figura 6: Ciclo de instrucción MOV A, (var1)

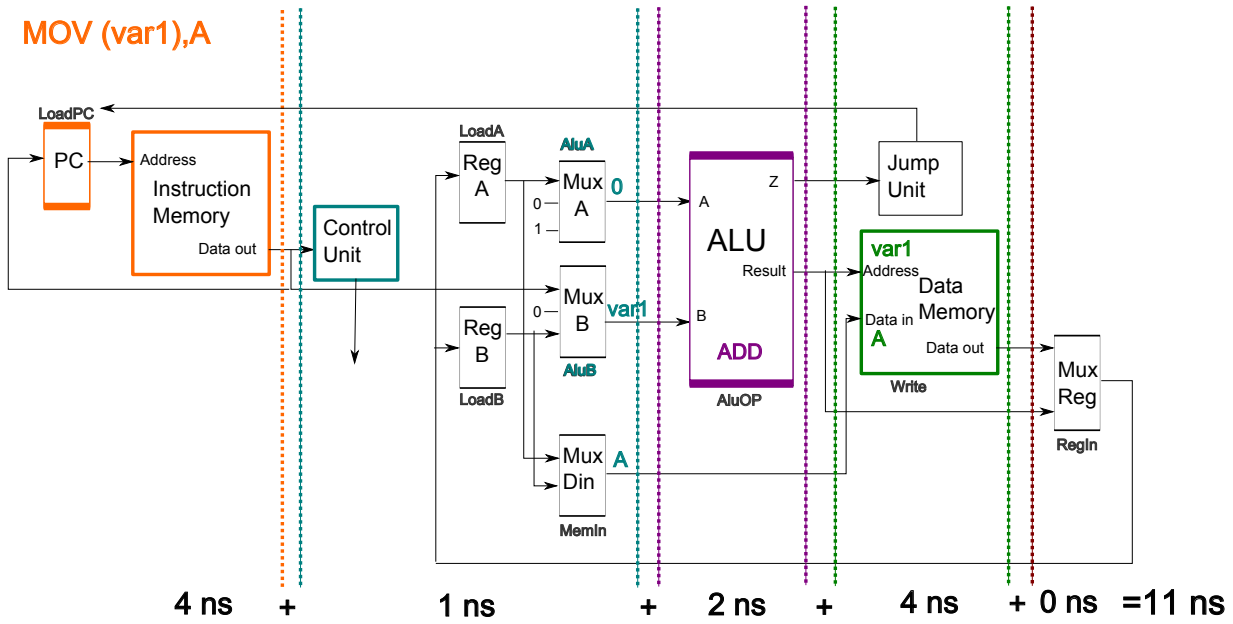


Figura 7: Ciclo de instrucción MOV (var1), A

Como se observa en los ejemplos anteriores, cada una de las etapas del ciclo de la instrucción tendrá una duración determinada, dependiendo de que instrucción se ejecute. De esta forma el tiempo total de ejecución de una instrucción variará de acuerdo a que etapas estén involucrados. En un computador de un ciclo por instrucción, como el computador básico, se debe cumplir que el tiempo que dura el ciclo (indicado por la frecuencia del clock) no puede ser menor que el tiempo que toma la instrucción más lenta (en este ejemplo `MOV A, (var1)`). Debido a esto en un computador de una instrucción por ciclo, la frecuencia del clock queda limitada por el tiempo que se demora en procesar la instrucción más lenta.

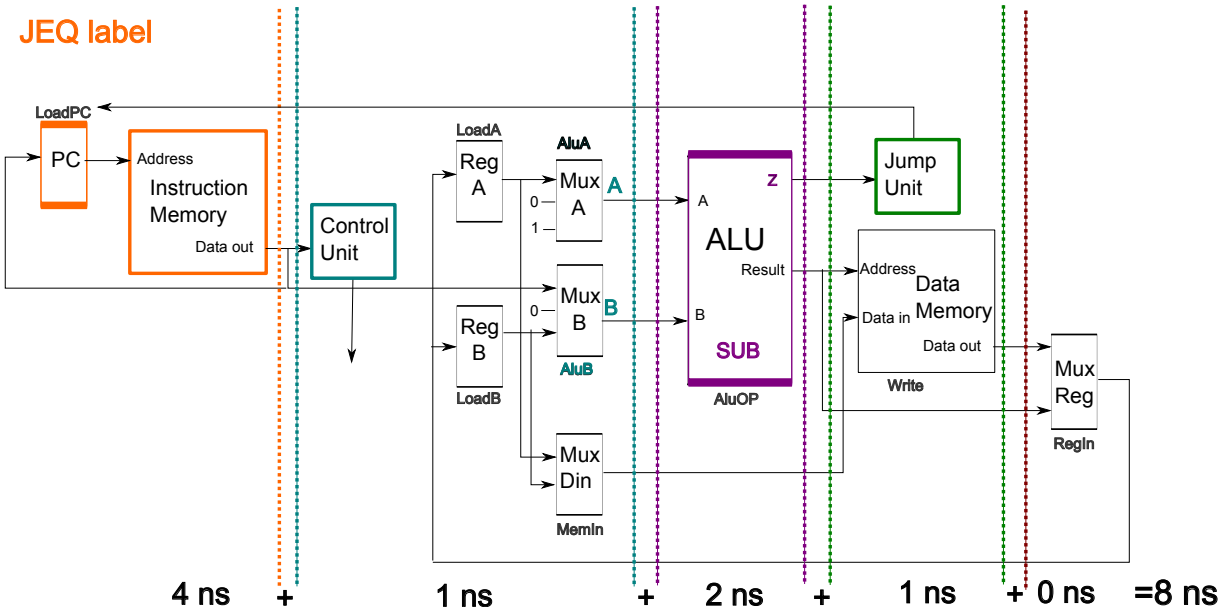


Figura 8: Ciclo de instrucción JEQ label

3 Instruction pipeline

3.1 Fundamentos

El ciclo de una instrucción se caracteriza por corresponder a una secuencia de pasos que debe recorrer esta, cada uno de los cuales estará a cargo de una unidad funcional del computador distinta. Este tipo de procesos secuenciales con etapas independientes se conoce como **pipeline**. Existen diversos procesos que pueden ser modelados como un pipeline, siendo los más comunes las líneas de ensamblaje industriales, en las cuales distintas tareas son ejecutadas sobre un producto (e.g. automóvil) y luego de ejecutar todas las tareas se obtiene un producto final terminado.

En el caso del procesamiento de la instrucción, esta secuencia de etapas se conoce como **instruction pipeline**, que contiene las distintas partes del ciclo de la instrucción antes descrito. A diferencia de las líneas de ensamblaje, en el **instruction pipeline** no se obtiene un producto final, sino que el resultado corresponde a lo que haya ocurrido a lo largo del proceso.

La gran ventaja de un pipeline de ensamblaje está en que es posible paralelizar la producción de un producto (e.g auto), dado que al mismo tiempo que se le esté aplicando un proceso a un elemento (e.g agregar las puertas), es posible estar ejecutando otra tarea sobre otro elemento (e.g. construir el chasis). De esta forma, al dividir la producción completa en tareas menores se logra reducir el tiempo, ya que se aprovecha que cada parte del proceso es independiente y que por tanto queda "'libre"' una vez que un determinado elemento ha pasado por esta.

En el **instruction pipeline** ocurre lo mismo que en el pipeline de ensamblaje: la idea es poder reutilizar cada una de las subetapas para ir procesando otras instrucciones de manera simultánea. Supongamos el siguiente ejemplo: se quieren ejecutar 3 instrucciones seguidas en el computador básico modificado. Como señalamos anteriormente, dado que la instrucción más lenta toma 12 nanosegundos (ns), el ciclo del clock debe tardar ese tiempo como mínimo siempre. Considerando esto, las tres instrucciones tomarían 36 ns (figura 9).

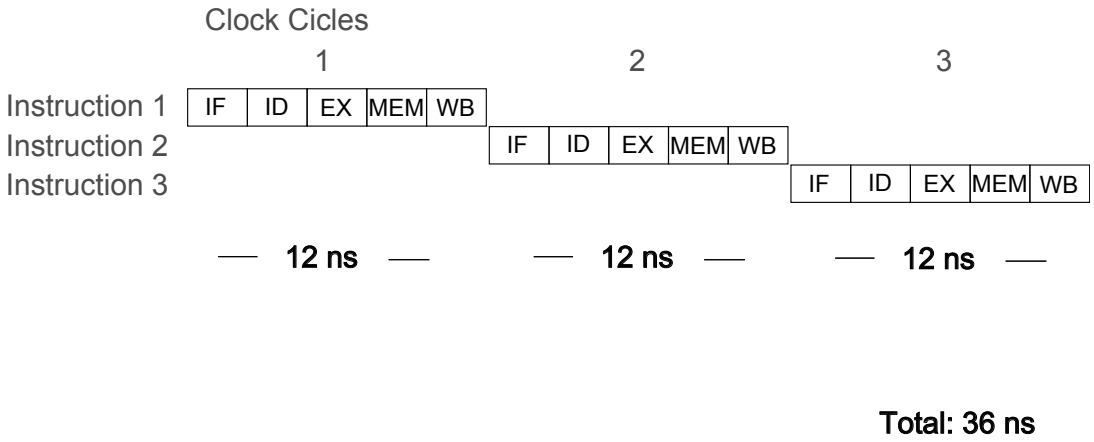


Figura 9: Diagrama ejecución sin pipeline

Para poder aprovechar el instruction pipeline, hay que considerar que cada instrucción se ejecuta en una secuencia de etapas, cada una de las cuales ocupa una unidad funcional distinta (al igual que en la línea de ensamblaje) y por tanto, una vez que se desocupa una etapa de procesar una instrucción, puede inmediatamente comenzar a procesar la siguiente. Para lograr esto, es necesario que en cada ciclo del clock no se procese toda una instrucción, sino sólo una etapa del ciclo, con lo que podemos tener ciclos de clocks más rápidos. El tiempo del ciclo estará limitado ahora no por la instrucción mas lenta, sino por la etapa más lenta, lo que continuando con el ejemplo anterior serían 4 ns para los etapas de acceso a memoria.

Con esta modificación, el procesamiento de las 3 instrucciones quedaría de la siguiente manera:

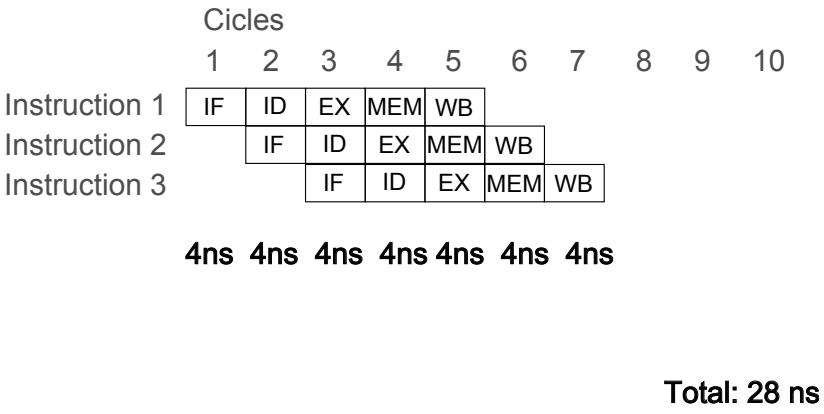


Figura 10: Diagrama ejecución con pipeline

Se observa una reducción de tiempo de los 36ns originales a 28ns. Esta reducción puede parecer menor, pero hay que considerar que un programa contiene millones de instrucciones. Si se considera ahora un ejemplo con un millón de instrucciones, el tiempo que tomaría procesarlas sin pipeline sería de 12 millones de nanosegundos (12×1 millon). En el caso del procesador con pipeline, el cálculo no es tan directo. Hay que observar que pasa con más instrucciones en el diagrama:

Se puede observar que a partir de la 5ta instrucción, el pipeline está "'lleno"' es decir, todas las etapas están procesando algo. Con esto se logra que a partir de esta instrucción se logra estar procesando **una instrucción por ciclo**, pero un ciclo mucho más rápido que en el caso anterior 4ns. Sólo en las 4 primeras instrucciones el pipeline está funcionando parcialmente, por lo que la regla para determinar cuanto se demoraría para este pipeline de 5 etapas es $Tiempociclo \times Ninstrucciones + Tiempociclo \times 4$. En este ejemplo serían entonces 4 millones 16 nanosegundos, es decir se reduce prácticamente un tercio del tiempo

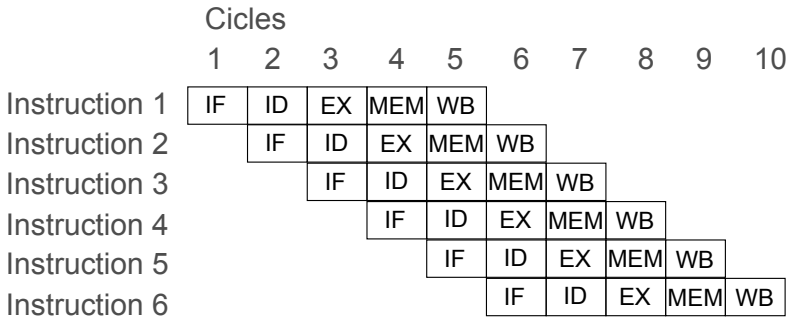


Figura 11: Diagrama ejecución con pipeline y 6 instrucciones

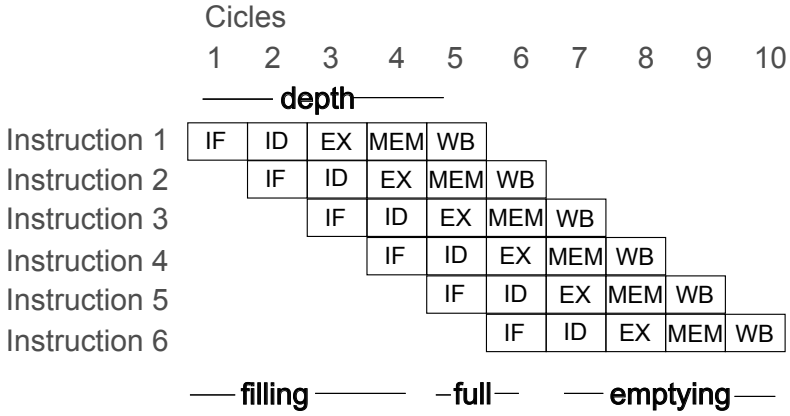


Figura 12: Terminología de pipeline

original.

La gran paradoja del instruction pipeline es que, el tiempo para procesar cada instrucción no disminuye, sino que aumenta: en este caso, el tiempo de procesar cada instrucción es de 20 ns, mayor que los 12 ns anteriores. Esto se debe a que la etapa más lenta indicará la velocidad del clock y por tanto ese tiempo se multiplicará por todas las etapas. Lo que mejora considerablemente con un sistema que ocupa pipeline es el **throughput** de las instrucciones de un programa, es decir, cuantas instrucciones son procesadas en un cierto tiempo. En el ejemplo anterior, en los mismos 12 millones de nanosegundos que toma el computador sin pipeline en ejecutar 1 millón de instrucciones, el computador con pipeline ejecutará 3 millones de instrucciones.

La siguiente es una lista de términos relevantes en el contexto de pipeline (figura 12):

- La **profundidad (depth)** del pipeline corresponde al número de etapas de este. En nuestro ejemplo son 5 etapas.
- El período al comenzar el procesamiento de una secuencia de instrucciones, en el cual no todas las etapas del pipeline están ocupadas, se conoce como **llenado o filling**.
- El período en el cual todas las etapas están ocupadas se dice que el pipeline está **lleno o full**.
- El período al terminar el procesamiento de una secuencia de instrucciones, en el cual no todas las etapas del pipeline están ocupadas, se conoce como **vaciado o emptying**.

3.2 Soporte de Hardware

Para implementar un procesador con pipeline es necesario ir almacenando los resultados de las distintas etapas y propagándolos, de manera que en el siguiente ciclo se puedan utilizar para la siguiente etapa. Para lograr esto, se utilizan una serie de registros que se ubican entre las distintas etapas. Los conjuntos de registros se denominan de acuerdo al par de etapas que comuniquen: registros IF/ID, ID/EX, EX/MEM y MEM/WB (figura 13). Para la etapa de write back no se requieren registros adicionales, ya que el resultado de esta etapa se almacena en los registros propios de la CPU.

En cada uno de estos registros se almacenará los distintos resultados de cada etapa, para el computador básico estos serían:

- IF/ID: almacena la instrucción obtenida desde la memoria de instrucciones, incluyendo el opcode y el parámetro
- ID/EX: almacena los dos parámetros de la ALU, el valor del registro seleccionado para ser copiado a memoria (si corresponde) y el parámetro de la instrucción. Este último se debe ir propagando para el caso de las instrucciones de salto, ya que es necesario que sea cuando se revise si va a haber salto (en la etapa MEM) que se cargue el PC con la dirección de memoria a saltar.
- EX/MEM: resultado de la ALU, el valor del registro seleccionado para escribir en memoria y el parámetro de la instrucción ambos que se propagaron de la etapa anterior.
- MEM/WB: resultado de la ALU y el valor leído de memoria, los cuales serán utilizados en WB para escribir (si corresponde) en los registros.
- El resultado de la etapa WB se almacena en los registros A o B, por lo que no se requieren registros especiales.

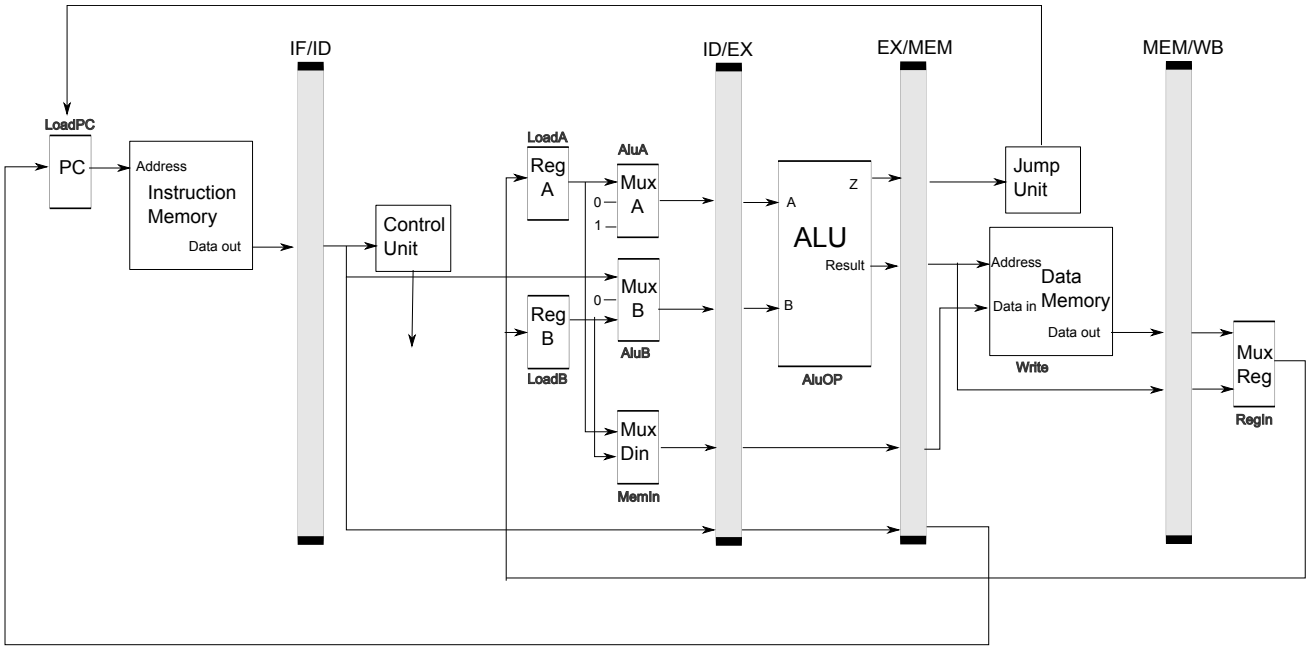


Figura 13: Soporte de hardware para propagación de datos

Además de almacenar datos, los registros intermedios deben almacenar las señales de control, generadas en la etapa ID, que son necesarias en etapas siguientes (figura 14):

- ID/EX: almacena todas las señales de control necesarias para todas las siguientes etapas: operación de la ALU (AluOP), señal de escritura en Memoria (W), señales de carga en los registros (LoadA y LoadB), señal de selección de que se va a escribir en los registros (RegIn) y señal que indica si la instrucción actual es de salto (Jmp).
- EX/MEM: en la etapa EX se utilizó la señal AluOp, por lo que no es necesario propagarla, se propagan todas las demás.
- MEM/WB: En la etapa MEM se utilizan las señales Jmp y W, solo se requiere propagar la señal RegIn y las señales LoadA y LoadB.

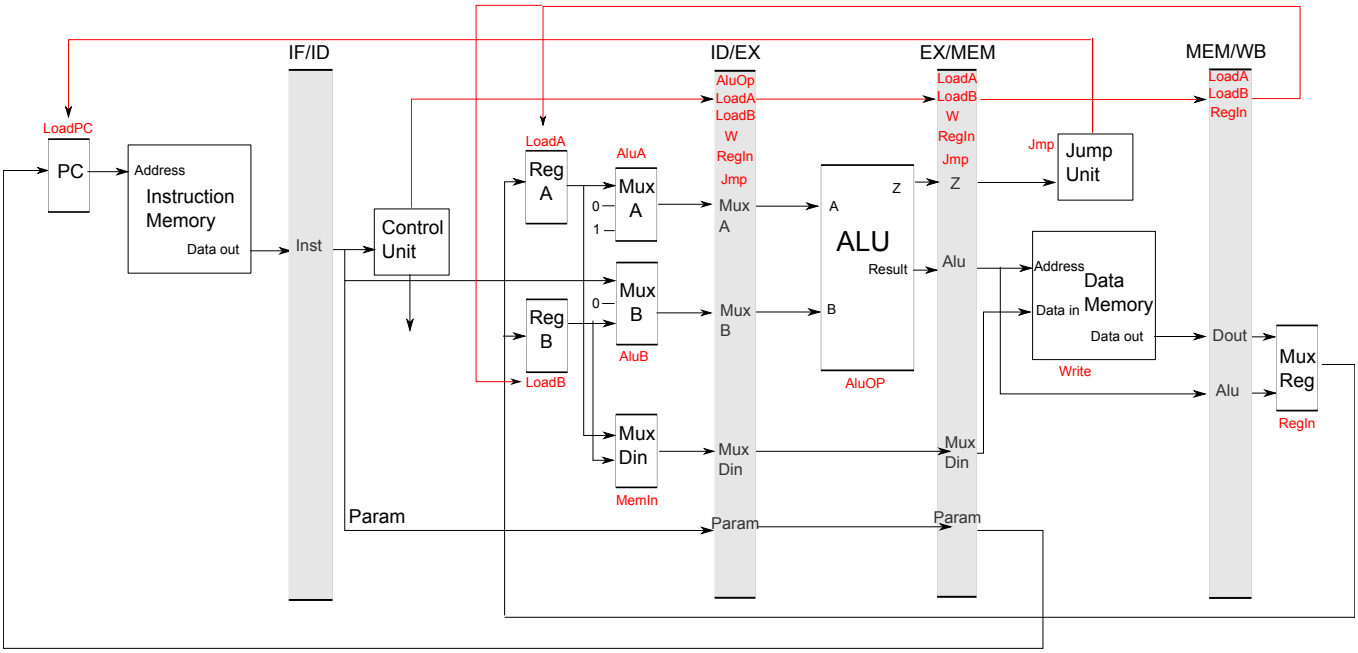


Figura 14: Soporte de hardware para propagación de control

3.3 Hazards

Al incorporar un pipeline en el computador, se generan una serie de problemas que son conocidos como **hazards**. Estos problemas ocurren por distintas razones, pero en general se deben al hecho de empezar a procesar una instrucción antes de que se haya procesado la anterior. Existen tres tipos de hazards: hazards de datos (data hazards), hazards de control (control hazards) y hazards estructurales (structural hazards).

3.3.1 Hazards de datos

Para entender los data hazards, revisemos el siguiente ejemplo de un programa en el que se ejecutan la instrucción `ADD A,B` y luego la instrucción `AND B,A`. Se puede observar que la segunda instrucción ocupa como parámetro el registro A. La primera instrucción va a modificar el valor de A, por lo que es necesario que haya terminado de ejecutarse para que las siguientes instrucciones tengan el valor correcto. El diagrama de pipeline de estas dos instrucciones es el siguiente:

Como se observa en el diagrama, la primera instrucción `ADD A,B` va a actualizar el valor del registro A en su etapa de **WB**, es decir en el ciclo 5. El problema es que la siguiente instrucción `AND B,A`, necesita obtener el valor de A en su etapa **ID**, la que ocurre en el ciclo 3. La causa de este problema es que la

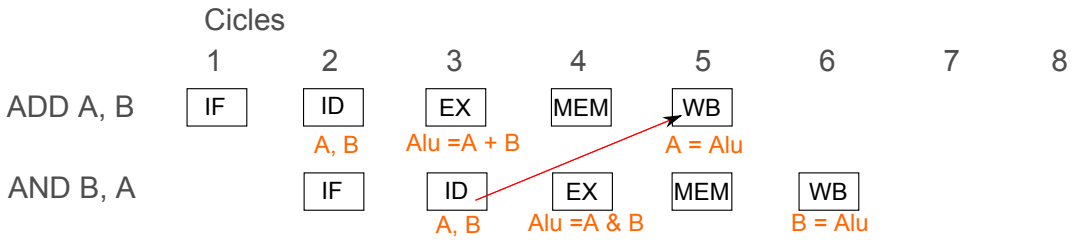


Figura 15: Ejemplo de data hazard, las flechas rojas indican las dependencias de datos

secuencia de instrucciones del ejemplo presenta **dependencia entre los datos** que utiliza. Si la instrucción 2 no ocuparán el registro A, no habría problema, pero como depende de tener el valor que se genera en la primera instrucción, ocurre un error.

Si observamos el diagrama de la figura 16 observamos que aunque efectivamente es en la etapa WB de la instrucción 1 cuando el registro A se actualiza, el nuevo valor de A se generó antes, en la etapa EX, la cual ocurre en el ciclo 3 de ejecución. Para la instrucción 2 además, tenemos que aunque el valor de A se capta en la etapa ID, este se necesitará en la etapa EX, es decir en el ciclo 4. De esta forma, cuando el nuevo valor de A se necesita (ciclo 4), este ya se generó (ciclo 3), y está almacenado en los registros de la etapa ID/EX. Lo que necesitamos entonces es una forma de propagar el valor que se generó en la ALU al procesar la primera instrucción luego de la ejecución para que pueda ser ocupado como parámetro en la ejecución de las siguientes etapas EX, de la segunda instrucción (figura 16). Este mecanismo de propagación que se agrega a un pipeline se conoce como **forwarding**.

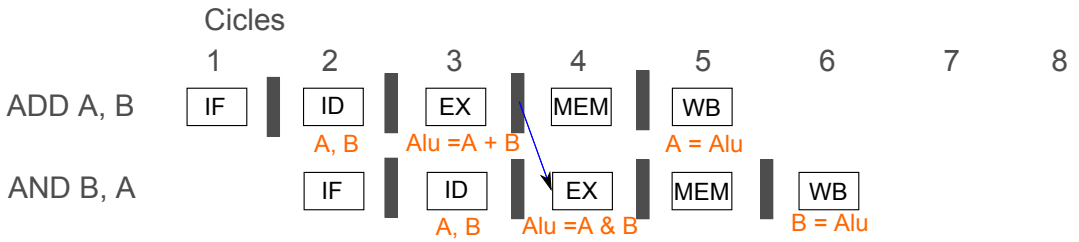


Figura 16: Ejemplo de data hazard, las flechas azules indican cuando hay que hacer forwarding

La idea de forwarding es, al estar ejecutando una instrucción, detectar si va a ocurrir un data hazard, y en ese caso realizar la propagación correspondiente. Esta detección debe realizarse en la etapa EX de la instrucción actual, antes de que se ejecute la operación de la ALU, que es lo primero que va a necesitar algún parámetro.

Existen varias situaciones en las cuales puede ocurrir esta detección, dependiendo de que instrucción previa genera dependencia, y en que etapa se genera:

1. Instrucción previa escribe en registro; instrucción actual utiliza registro en la ALU:

   La primera situación que requiere forwarding es en el caso de que una instrucción vaya a realizar una operación con la ALU, y requiera como parámetro un registro que está siendo modificado por la instrucción anterior. En el ejemplo anterior esto es lo que ocurre con la segunda instrucción, `AND B,A` .

   Para detectar este tipo de hazards hay que obtener la siguiente información:

   (a) La instrucción previa tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción previa esta

en la etapa MEM en este momento, las señales de control correspondientes estarán en el registro EX/MEM. La nomenclatura para referirse a estas señales es EX/MEM.LoadA y EX/MEM.Loadb.

(b) La instrucción actual ocupa como uno de sus parámetros el mismo registro que se escribió previamente. Para saber esto necesitamos saber que valores se eligieron como parámetros de la ALU en los MuxA y MuxB, lo cual estará almacenado en las señales de control AluA y AluB asociados a la instrucción actual, las cuales ahora deben ser propagadas entre las etapas ID y EX. Como la instrucción actual está en la etapa EX, las señales de control que nos interesan están en los registros ID/EX, y por tanto la nomenclatura para estas señales es ID/EX.AluA y ID/EX.AluB.

En base a esta información se puede decidir si corresponde hacer forwarding a la entrada A o B de la ALU. Lo que se tiene que propagar es el resultado de la ALU que se tenga almacenado en la etapa MEM. Para saber si ese resultado debe ingresar a la entrada A o B, se revisa lo siguiente:

- Si EX/MEM.LoadA == 1 y ID/EX.AluA == A (se eligió el registro A como parámetro), corresponde hacer forwarding del resultado de la ALU de la etapa MEM a la entrada A de la ALU.
- Si EX/MEM.LoadB == 1 y ID/EX.AluB == B (se eligió el registro B como parámetro), corresponde hacer forwarding del resultado de la ALU de la etapa MEM a la entrada B de la ALU.

2. Instrucción previa a la anterior escribe en registro; instrucción actual utiliza registro en la ALU:

La segunda situación en que es necesario propagar ocurre cuando una instrucción que va a operar con la ALU requiere como parámetro un registro que está siendo modificado por la instrucción de dos ciclos atrás:

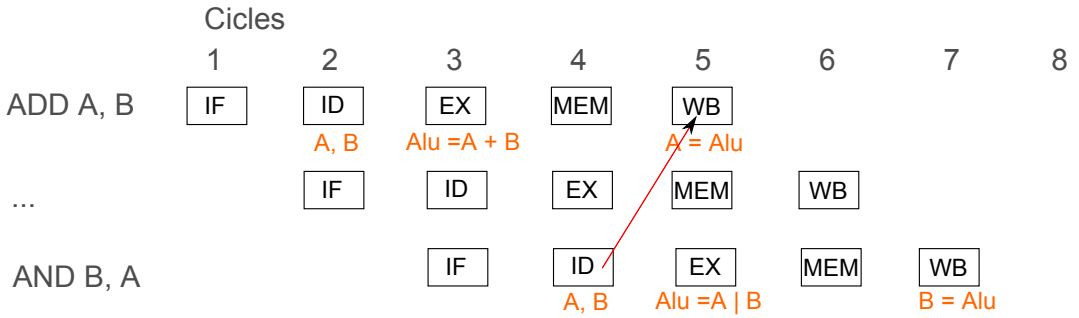


Figura 17: Ejemplo de data hazard en que una instrucción afecta a la subsiguiente.

Aunque este caso es similar al anterior, y también es posible detectarlo ocupando el mismo tipo de condiciones antes descritas, existen un par de diferencias. En primer lugar, cuando la instrucción que va a sufrir un data hazard esté en la etapa EX (`AND B,A` en el ejemplo), la instrucción que generará el hazard (`AND B,A`) estará en la etapa WB, y por tanto las señales LoadA y LoadB que interesa revisar estarán en los registros MEM/WB (MEM/WB.LoadA y MEM/WB.LoadB). Lo segundo es qué ocurre si la instrucción intermedia genera también dependencia de datos con la tercera instrucción, como en el ejemplo de la figura 18. En este caso, no es necesario hacer el forwarding desde la instrucción 1 a la 3, dado que para la instrucción 3 lo que importa es el último valor de A, que será obtenido en la instrucción 2.

Considerando esto, la información para detectar este tipo de hazards es:

(a) La instrucción anterior a la previa tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción

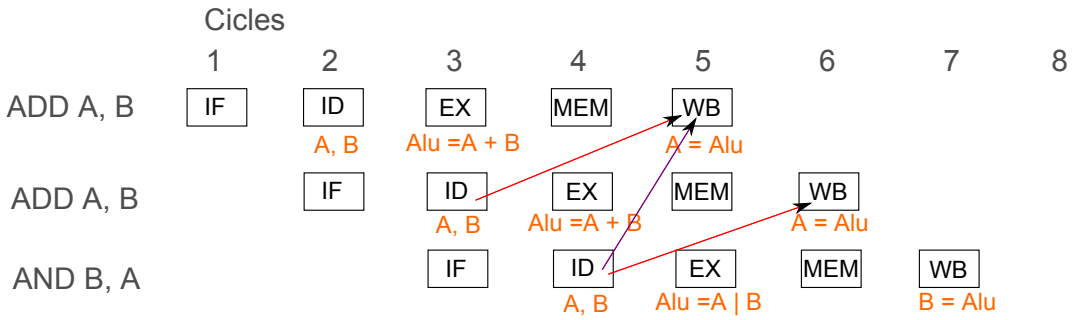


Figura 18: Ejemplo de data hazard: para la tercera instrucción sólo importa la última modificación sobre el registro A, por lo que la dependencia entre la 1era y 3era instrucción (flecha morada) no necesita forwarding.

anterior a la previa esta en la etapa WB en este momento, las señales de control correspondientes estarán en el registro MEM/WB: MEM/WB.LoadA y MEM/WB.Loadb.

(b) La instrucción actual ocupa como uno de sus parámetros el mismo registro que se escribió previamente. Para saber esto necesitamos saber que valores se eligieron como parámetros de la ALU en los MuxA y MuxB, lo cual estará almacenado en las señales de control AluA y AluB asociados a la instrucción actual: ID/EX.AluA y ID/EX.AluB.

(c) Si la instrucción previa también modificó los registros A o B, esta información estará en los registros EX/MEM: EX/MEM.LoadA y EX/MEM.LoadB.

En base a esta información se puede decidir si corresponde hacer forwarding a la entrada A o B de la ALU. Lo que se tiene que propagar es el resultado de la ALU que se tenga almacenado en la etapa WB. Para saber si ese resultado debe ingresar a la entrada A o B, se revisa lo siguiente:

- Si MEM/WB.LoadA == 1, ID/EX.AluA == A y EX/MEM.LoadA == 0 (es decir, no hubo escritura en la instrucción anterior), corresponde hacer forwarding del resultado de la ALU de la etapa WB a la entrada A de la ALU.
- Si MEM/WB.LoadB == 1, ID/EX.AluB == B y EX/MEM.LoadB == 0 (es decir, no hubo escritura en la instrucción anterior), corresponde hacer forwarding del resultado de la ALU de la etapa WB a la entrada B de la ALU.

3. Instrucción previa escribe en registro; instrucción actual escribe en memoria valor del registro modificado:

Los últimos casos en que es necesario propagar son cuando la instrucción actual está escribiendo en memoria el valor de un registro que está siendo modificado por la instrucción previa:

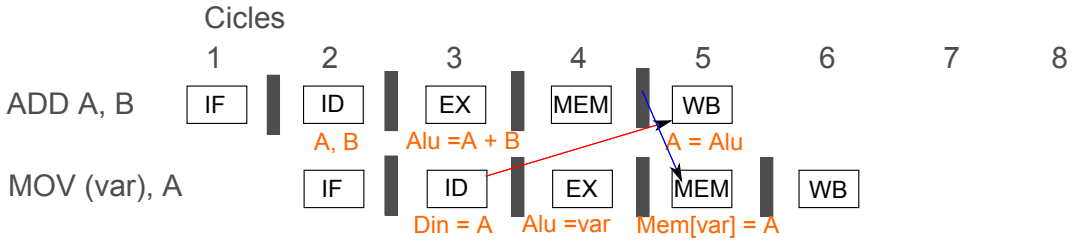


Figura 19: Ejemplo de data hazard: la segunda instrucción necesita el nuevo valor de A para escribir en memoria. Se debe realizar forwarding de la salida de la etapa WB previa a la etapa MEM actual.

En este caso, la dependencia de datos está entre las etapas ID de la instrucción actual, donde se elige el registro que será escrito en memoria, y la etapa WB de la instrucción anterior, donde se está modificando ese registro. A diferencia de los casos anteriores, la etapa a la cual es necesario propagarle datos es la etapa MEM. La información para detectar este tipo de hazards es:

(a) La instrucción anterior tiene que estar escribiendo en registro, es decir las señales LoadA o LoadB deben haberse activadas en el decode de dicha instrucción. Como la instrucción anterior está en la etapa WB en este momento, las señales de control correspondientes estarán en el registro MEM/WB: MEM/WB.LoadA y MEM/WB.Loadb.

(b) La instrucción actual está escribiendo en memoria (EX/MEM.W = 1) y ocupa como dato de entrada en memoria el mismo registro que fue modificado (EX/MEM.MuxDin = A o EX/MEM.MuxDin = B).

En base a esta información se puede decidir si corresponde hacer forwarding al valor de entrada de la memoria. Lo que se tiene que propagar en este caso es la salida del Mux RegIn de la etapa WB, ya que esta situación puede ocurrir tanto si la instrucción previa fue ocupando la ALU o leyendo de memoria. Las codiciones para revisar este caso son:

- Si MEM/WB.LoadA == 1, EX/MEM.MemDin == A, EX/MEM.W == 1 corresponde hacer forwarding del RegIn de la etapa WB a la entrada de memoria.
- Si MEM/WB.LoadB == 1, EX/MEM.MemDin == B, EX/MEM.W == 1 corresponde hacer forwarding del RegIn de la etapa WB a la entrada de memoria.

Para lograr detectar el forwarding e implementarlo cuando corresponda, se agrega una pieza de hardware denominada **forwarding unit**, la cual se encarga de revisar si ocurrió un data hazard y en caso de haber ocurrido uno, propaga los valores a las entradas de la ALU correspondiente, a través de dos multiplexores que se agregan a estas entradas (figura 20).

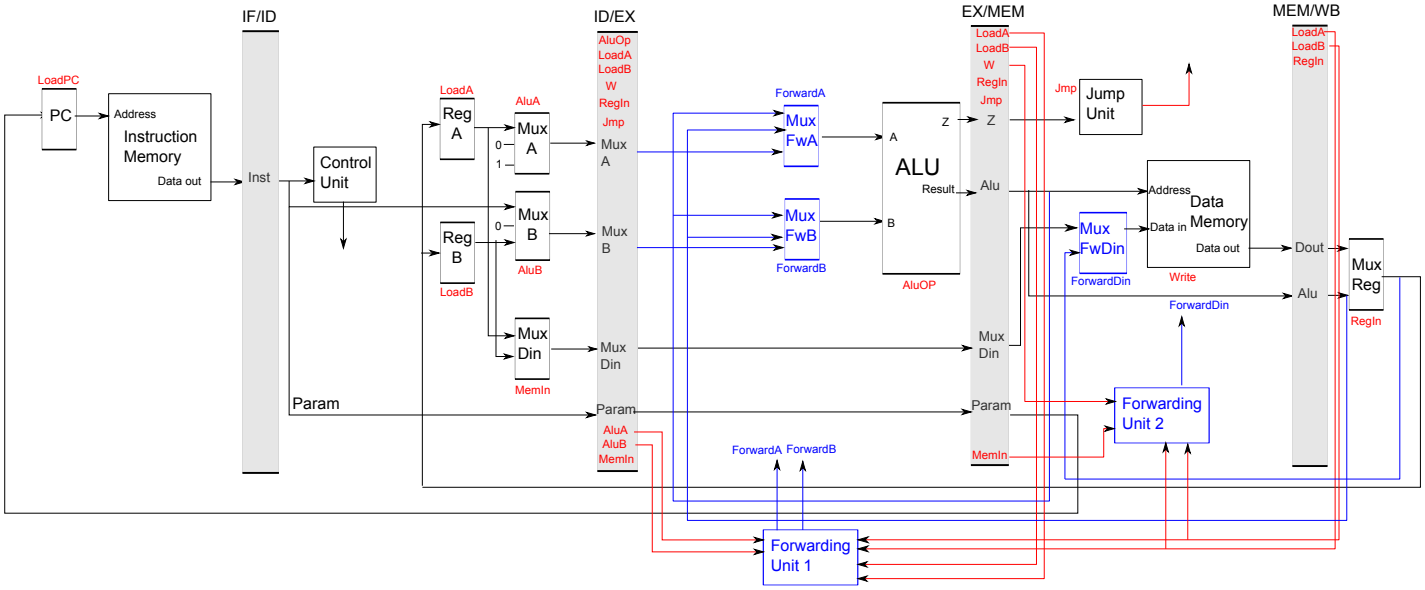


Figura 20: Soporte de hardware para forwarding.

Al agregar forwarding al procesador se solucionan los casos de data hazard antes descritos, pero no es suficiente para manejar todos los posibles casos. En particular, existe otro tipo de data hazard, que ocurre

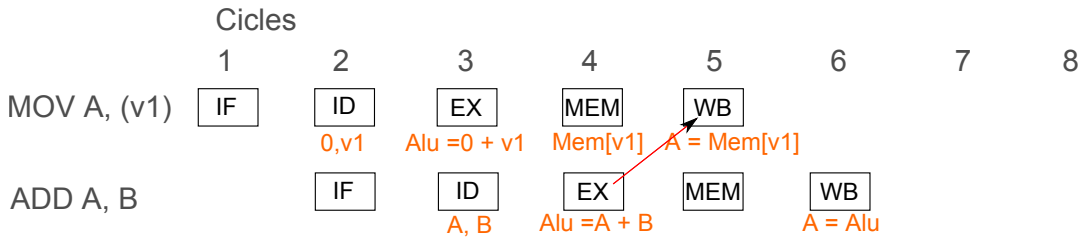


Figura 21: Ejemplo de data hazard, producido por una lectura de memoria.

cuando un valor se lee de memoria en una instrucción, y luego se utiliza como parámetro en la ALU de la instrucción siguiente, como se observa en el ejemplo de la figura 21.

El problema en este caso es que a diferencia de los hazards anteriores, acá el valor modificado no se obtiene en la etapa EX, sino en la etapa MEM. Si intentamos hacer forwarding ocurre lo que se observa en la figura 22: el valor que requerimos propagar solo se obtendrá al final del ciclo en que se ejecuta MEM, pero es requerido al principio de ese mismo ciclo por la etapa EX de la siguiente instrucción. Esto hace imposible lograr el forwarding ya que en el momento que tenemos el valor ya es muy tarde y se ejecutó la operación siguiente con el valor erróneo. En estos casos se requiere agregar un mecanismo que permita que la siguiente instrucción "'espere"' mientras se obtiene el valor necesario, lo que se conoce como **stalling**.

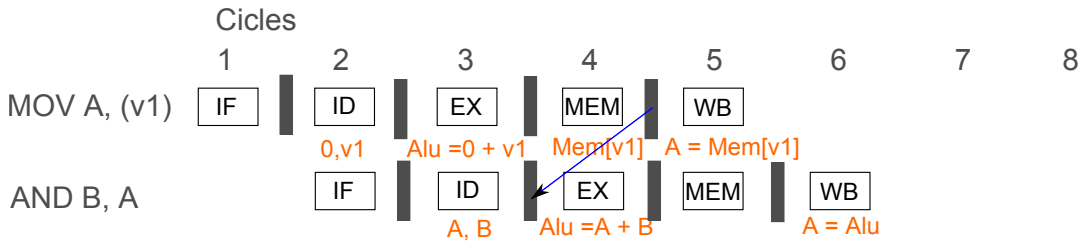


Figura 22: Ejemplo de data hazard, producido por una lectura de memoria.

En el mecanismo de stalling, la idea es hacer que la instrucción siguiente no entré en la etapa EX hasta que se haya obtenido el dato desde memoria. Para lograr esto hay dos mecanismos: uno implementado por hardware y otro por software. En el mecanismo por hardware, el procesador debe detectar esta situación y detener el avance del pipeline por un ciclo, insertando lo que se conoce como una "'burbuja"' en el pipeline (figura 23). Esta detención permite que se mantenga la consistencia de los datos, pero a costo de perder un ciclo de CPU.

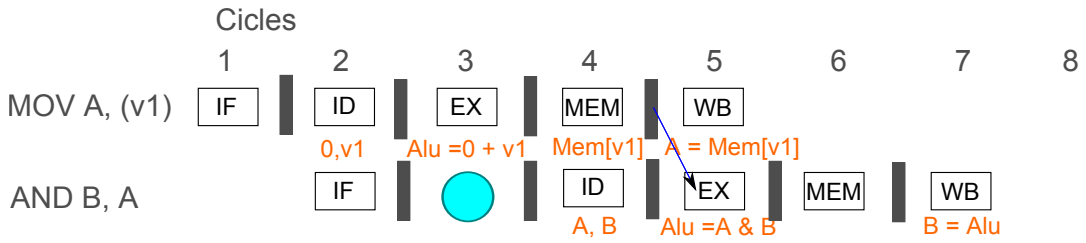


Figura 23: Stalling del pipeline para solucionar data hazard de lectura. Se agrega una "'burbuja"' al pipeline.

Las condiciones que se deben dar para que haya stalling para este computador, son las siguientes:

1. La instrucción anterior tiene que estar escribiendo en registro desde memoria, lo que se puede observar en la señal de control RegIn.

2. La instrucción actual ocupa como parámetro de la ALU el registro escrito.

En este caso, la idea es detectar lo antes posible el hazard para perder la menor cantidad de ciclos. Debido a esto, la detección de este tipo de hazards se realiza en la etapa ID, cuando son generadas las señales de control de la instrucción actual, la cual debemos detener en caso de que haya hazard. Los valores que se deben revisar para estos casos son:

- Si ID/EX.LoadA == 1 y ID/EX.RegIn == Dout (se va a leer de memoria), y la instrucción actual tienen AluA == A, corresponde hacer stalling para evitar inconsistencias con el registro A.
- Si ID/EX.LoadB == 1 y ID/EX.RegIn == Dout (se va a leer de memoria), y la instrucción actual tienen AluB == B, corresponde hacer stalling para evitar inconsistencias con el registro B.

Para detectar estos hazards, se agrega una unidad de detección de hazards en la etapa ID. Una vez detectado un caso, esta unidad realiza tres acciones:

1. Setea en cero todas las señales de control almacenadas en el registro ID/EX, para no propagar la instrucción actual.
2. Envía una señal de no escritura al registro IF/ID, para que mantenga la instrucción actual almacenada y no sobreescriba con la siguiente.
3. Envía una señal de no avanzar al PC, para que se mantenga apuntando a la instrucción siguiente, y no avance.

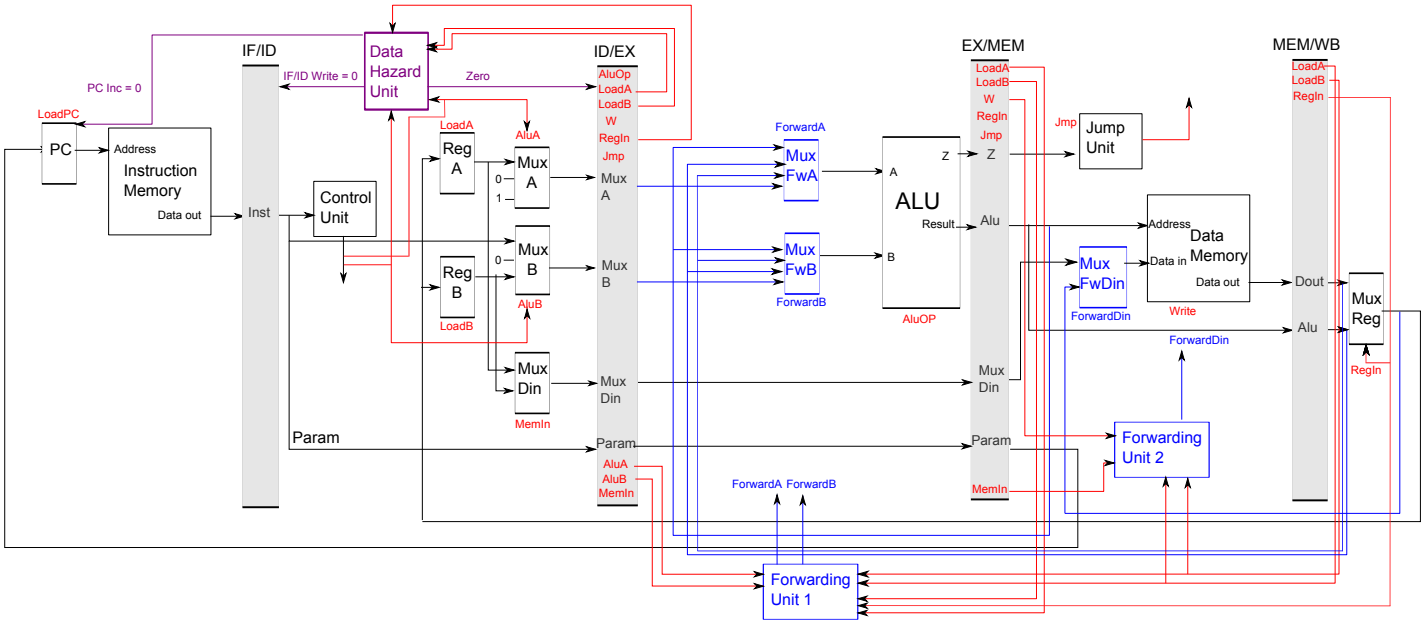


Figura 24: Soporte de hardware para stalling.

Un mecanismo alternativo para manejar stalling se implementa por software: la idea es que el compilador cuando genere el código assembly a partir del código alto nivel, detecte los casos posibles de hazard que requieran stalling e inserte entre las dos instrucciones una instrucción que no hace nada, la que se conoce como `NOP` (figura 25). Con este mecanismo se logra el mismo resultado que en el caso anterior, es decir se retrasa la CPU en un ciclo para lograr la consistencia en los datos. La ventaja es que no requiere modificar la CPU, ya que el trabajo lo realiza el compilador.

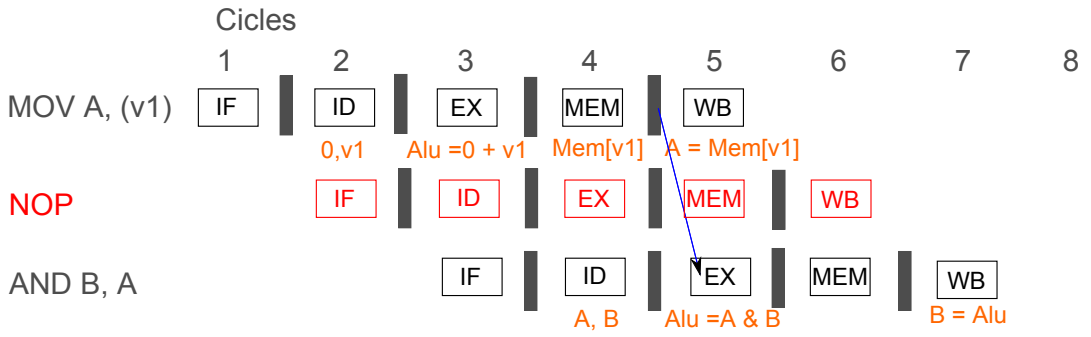


Figura 25: Inserción de un NOP para solucionar data hazard de lectura.

3.3.2 Hazards de Control

Un segundo tipo de hazard que ocurre en un procesador con pipeline corresponde a los hazards de control. Este tipo de hazard ocurre cuando hay una instrucción de salto: el problema con estas instrucciones es que dependiendo de si hay o no salto cambiará cual es la siguiente instrucción, pero con el diseño actual del procesador esto se sabe recién en la etapa MEM de la instrucción de salto, y por tanto a esas alturas ya entraron las siguientes instrucciones al pipeline (figura 26).

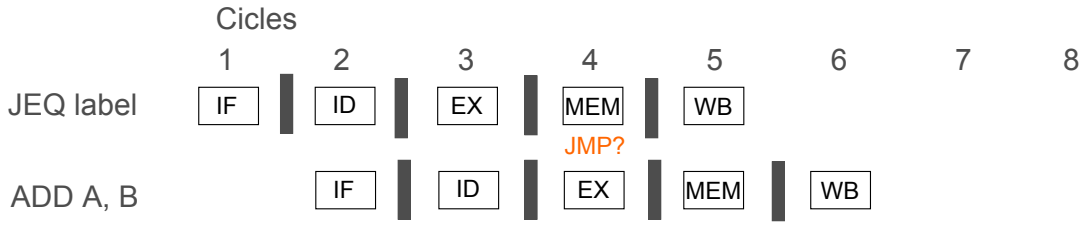


Figura 26: Hazard de control producido por una instrucción de salto: si el salto se efectúa la segunda instrucción no se debe ejecutar.

Para solucionar este tipo de hazards existen distintos mecanismos. Lo más simple es realizar stalling del pipeline los ciclos que sean necesarios para que la siguiente instrucción no entre hasta que se sepa si hay o no salto, y por tanto se sepa que instrucción corresponde. En el caso de este pipeline, esto involucra detener la CPU por 3 ciclos, es decir perder 3 ciclos por cada salto (figura 27).

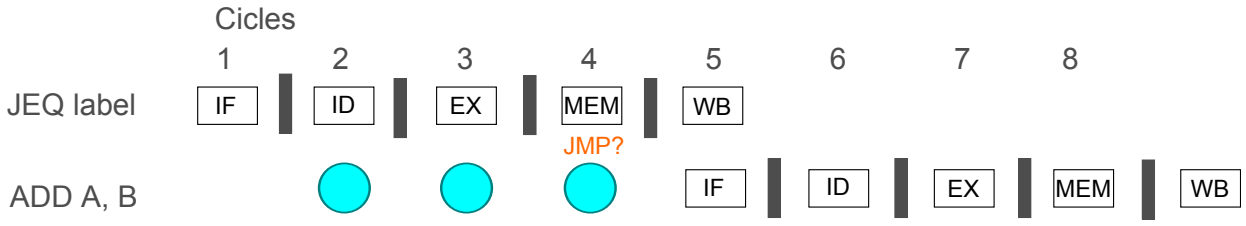


Figura 27: Stalling de 3 ciclos para prevenir control hazard

Es posible mejorar el manejo de los saltos, ocupando un mecanismo de **predicción de salto**. La idea de este mecanismo es que uno puede predecir si el salto va a ocurrir o no, y en caso de que la predicción sea correcta, no se van a perder ciclos de la CPU. La predicción más simple consiste en predecir que nunca va a ocurrir salto, es decir, que el flujo de las instrucciones siempre seguirá de manera secuencial. Cuando esta predicción sea correcta, no se perderán ciclos de la CPU (figura 28).

En caso de que la predicción sea incorrecta y efectivamente ocurra el salto, las siguientes tres instrucciones que ya entraron al pipeline no debe ser ejecutada. Para lograr esto se agrega una unidad de hazard de control

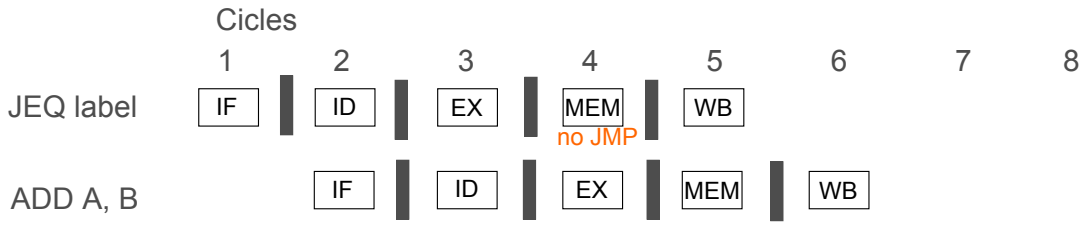


Figura 28: Manejo de hazard de control ocupando predicción de salto.

en la etapa IF (figura 29), la cual cuando detecta que hay salto envía una señal de **flush** que avisa a las siguientes etapas del pipeline que no deben ejecutar las señales de control que se habían enviado (figura 30).

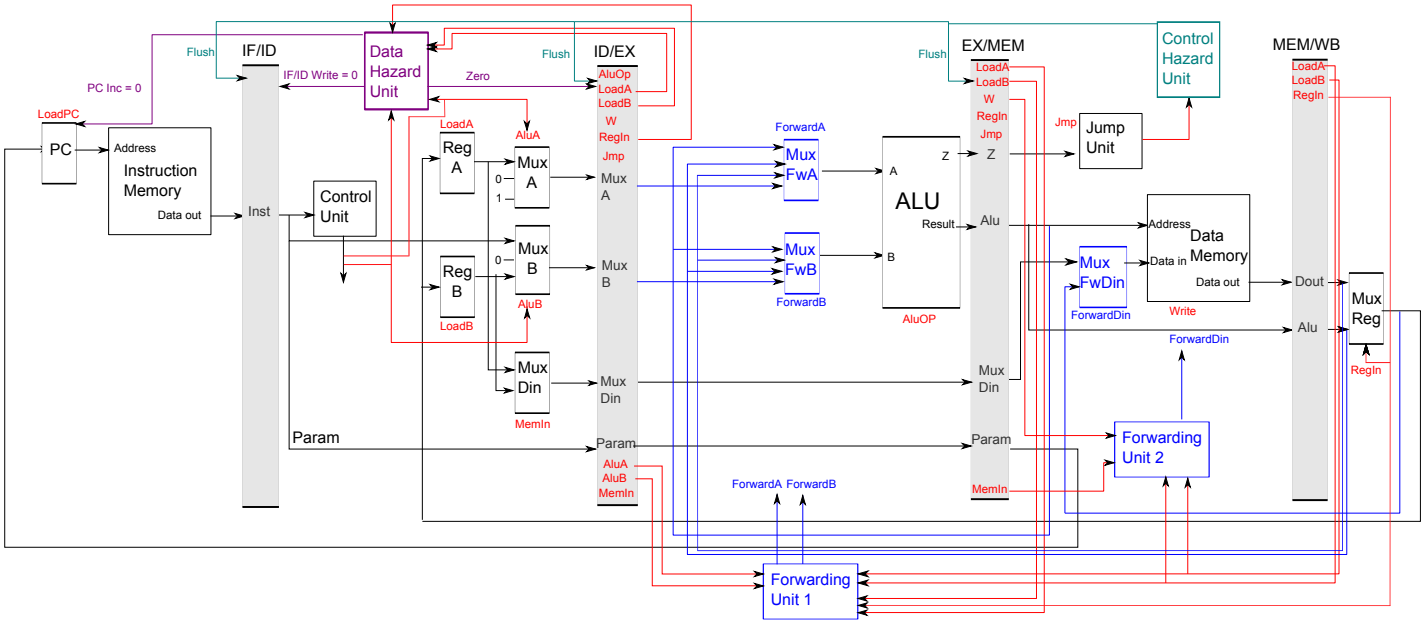


Figura 29: Soporte de hardware para control hazards.

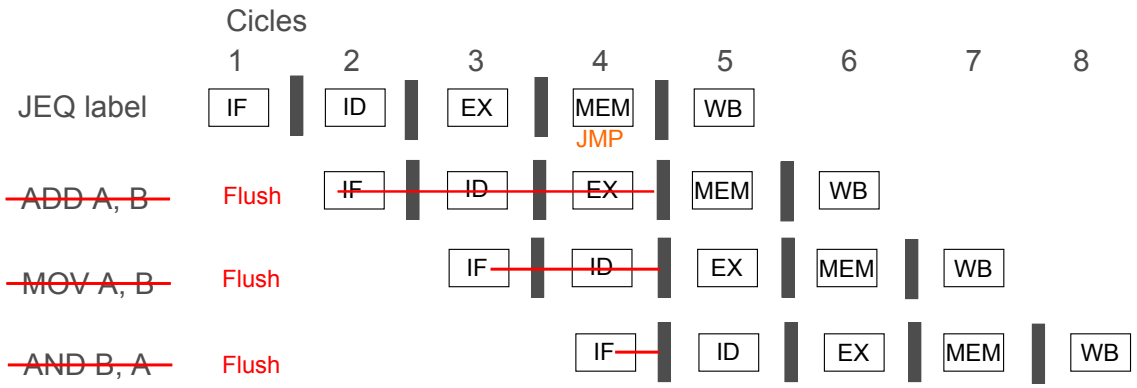


Figura 30: La unidad de control de hazard detecta que ocurrió un salto y envía una señal de flush para no ejecutar las siguientes etapas.

Es posible mejorar aún más el manejo de estos hazards, ocupando sistemas de **predicción dinámica** en los cuales se predice a medida que se ejecuta el programa si un salto va a ocurrir o no, en base a información estadística obtenida de cada instrucción. Este tipo de predicción requiere hardware especializado, pero con esto es posible mejorar aún más el desempeño del procesador en estos casos.

3.3.3 Hazards Estructurales

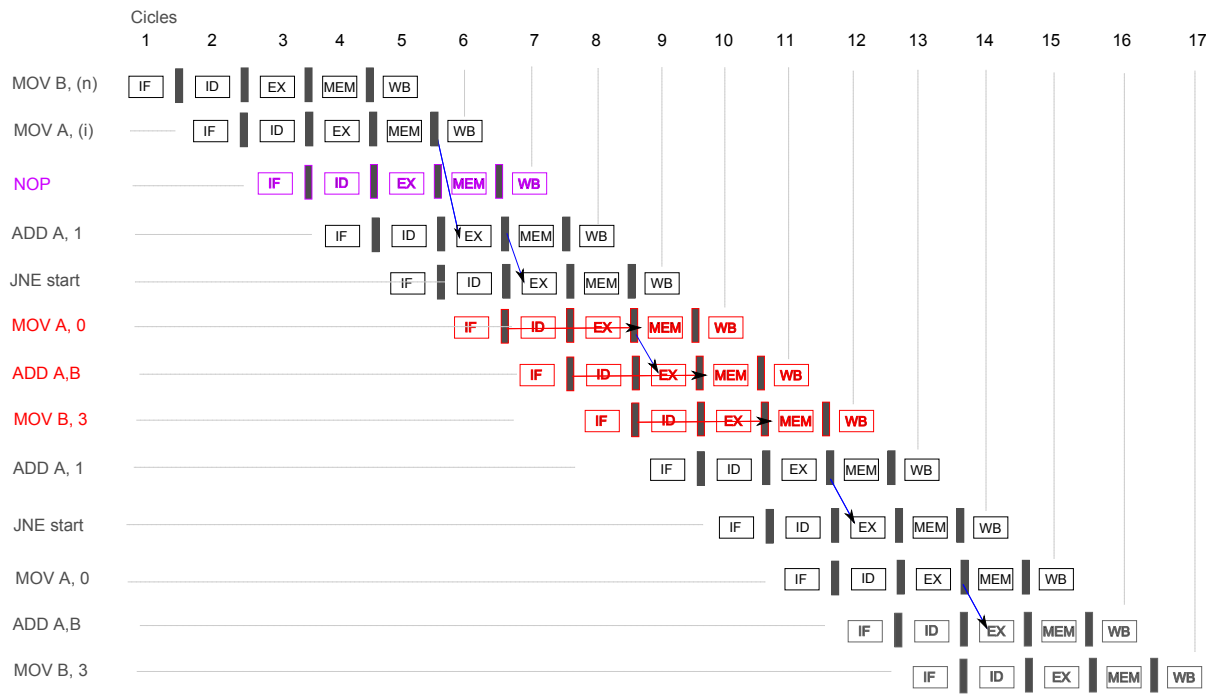
Los hazards estructurales ocurren cuando dos etapas del pipeline necesitan acceder a la misma unidad funcional. Un ejemplo ocurre en un computador Von Neumann, donde la memoria única de instrucciones y datos sería accedida tanto en la etapa IF como en la etapa MEM. Existen distintas formas de solucionar este tipo de hazards:

- Agregar unidades funcionales extras que permitan atender las dos etapas del ciclo sin contención. En el caso de las memorias, tener una caché split de primer nivel transforma la arquitectura del computador en Harvard y se soluciona el problema.
- Agregar "'burbujas"' mediante stalling que hagan que una de las instrucciones que quería ocupar la unidad funcional en un cierto ciclo espere hasta el siguiente.

Ejemplo

El siguiente ejemplo de código muestra la secuencia efectiva del pipeline de un programa en un computador con pipeline, que tiene forwarding (flechas azules en el diagrama), stalling por software (instrucciones NOP), predicción de salto asumiendo salto no realizado y flushing en caso de predicción errónea (flecha roja).

```
   DATA:
   n       2
   i       0
   CODE:
           MOV B, (n)
           MOV A, (i)
start:     ADD  A, 1
            JNE start
            MOV  A,5
            ADD  A,B
            MOV B,3
```



4 Apéndice: Set de instrucciones de Computador Básico con Pipeline

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				

5 Ejercicios

Para el siguiente código del computador básico con pipeline, determine el número de ciclos que se demora detallando los estados del pipeline para cada instrucción. Considere que el pipeline tiene forwarding implementado entre todas sus etapas, el manejo de stalling se realiza por software y se tiene predicción de salto asumiendo que no ocurre salto. En su diagrama de pipeline detalle cuando ocurre forwarding de datos y flushing de instrucciones.

```
    DATA:
    res     0
    i       0
    CODE:
           MOV B,2
    mult:  MOV A,2
           ADD  (res)  //res = A + B
           MOV A, (i)
           ADD  A,1
           MOV (i), A
           JNE  mult
```

6 Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 4: The Processor.
- Huang, H. : CS232: Computer Architecture II, University of Illinois at Urbana-Champaign, http://howardhuang.u