**RESUMEN DE ARQUITECTURA DE COMPUTADORAS**

**Capítulo 2: Procesadores Segmentados**

**Resumen de arquitecturas MIPS**

Las arquitectura de los procesadores pueden ser de diversas formas dentro de las más comunes se tiene:

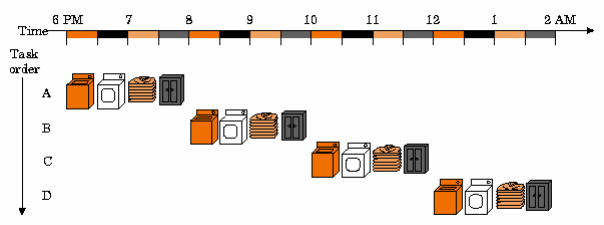
* ***Arquitectura monociclo:*** En este caso el procesador utiliza un solo bloque de lógica para resolver las instrucciones en un solo ciclo. Es decir se ejecuta una instrucción por ciclo. Pero esta arquitectura tiene el problema de que su circuito digital (bloque de lógica) tiene muchas partes inactivas.
* ***Arquitectura multiciclo:*** Esta idea se basa en resolver el problema anterior, identificando distintas funciones (etapas) con el fin de dividir los ciclos de un bloque de lógica entero en un conjunto de varios bloques de lógica más pequeños, es decir cada ciclo hace menos cosas en comparación del procesador monociclo. Si cada etapa se realiza en un ciclo reloj entonces la ejecución de una instrucción requerirá de múltiples ciclos. De esta manera el número de ciclos dependerá de cada instrucción. De esta forma cada unidad funcional (bloque de lógica pequeños) puede ser utilizado más de una vez por cada instrucción en diferentes ciclos. Esta implementación tiene el problema de que cada etapa no la utiliza en un mismo tiempo (en paralelo) a pesar de que se repiten por cada instrucción.
* ***Arquitectura pipeline:*** Esta arquitectura permite que cada etapa formada por diferentes módulos lógicos puedan ser ejecutadas en un mismo ciclo reloj, es decir ejecutar todas las etapas de un flujo de instrucciones en un mismo ciclo (en paralelo). Esta técnica es conocida como segmentación (pipelining) por media de la cual se solapa la ejecución de múltiples instrucciones y hoy en día es una técnica clave para hacer CPUs rápidas. De esta manera la idea es que el pipeline esté siempre lleno, es decir siempre ejecutando un flujo de instrucciones por cada ciclo reloj. Esta arquitectura a pesar de ser una de las más eficientes, presenta un montón de problemas cuyas soluciones las analizaremos a lo largo del curso.

**Concepto de Segmentación**

Para introducir los conceptos de segmentación, se iniciará con una analogía con una lavandería. En una lavandería, para una carga de ropa (x cantidad de kilos), las tareas que se realizan pueden dividirse en las siguientes etapas:

1. Colocar la carga de ropa sucia en la lavadora.
2. Sacar la ropa de la lavadora para pasarla a la secadora.
3. Doblar la ropa seca.
4. Acomodar la ropa limpia y seca en su lugar correspondiente.

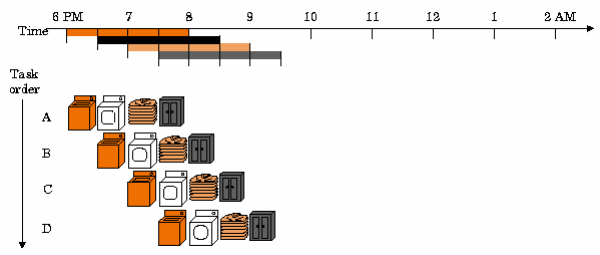
Si para una carga de ropa cada tarea requiere de media hora para su ejecución, al trabajar sobre cuatro cargas de ropa: A, B, C y D, se requerirán de 8 horas para concluir con el lavado de las 4 cargas. Como se muestra en la figura siguiente:



**figura 1.** Lavado de 4 cargas sin segmentación (analogía procesador monociclo)

Ahora, si contamos con los recursos suficientes, tal vez sea posible traslapar algunas tareas, de manera que después de que la carga A se extraiga de la lavadora y pase a la secadora, la carga B puede introducirse en la lavadora. De manera que al mismo tiempo se tendría trabajando a la lavadora y a la secadora con dos cargas diferentes.

En la media hora siguiente, puede doblarse la ropa que corresponde a la carga A, pasar la carga B a la secadora e introducir la carga C en la lavadora. De esta manera se traslaparían diferentes cargas de ropa durante el proceso de lavado y para el lavado de 4 cargas se requeriría solo de 3.5 horas, esto se muestra en la figura 2. Lo cual es la aplicación de segmentación en el proceso de lavado. Y cada uno de los pasos en los que se dividió el proceso se conoce como una etapa de la segmentación o un segmento.



**figura 2.** Lavado de 4 cargas con segmentación (analogía procesador pipeline)

Puede notarse en la figura 2 que el tiempo invertido para una carga de ropa sigue siendo de dos horas, lo que se disminuyó fue el tiempo invertido para las cuatro cargas. En otras palabras, la segmentación no disminuye el tiempo de ejecución pero si aumenta la productividad, y de esta manera se aumenta el rendimiento.

**Segmentación (pipelining) en el procesador**

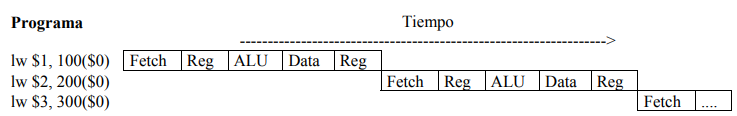
Un proceso similar se aplicará a la ejecución de las instrucciones. La ejecución de cada instrucción se dividirá en diferentes etapas, conectando cada una a la siguiente, para formar una especie de cauce - las instrucciones entrarán por un extremo, se procesarán a través de las etapas y saldrán por el otro extremo.

La ***productividad (throughput)*** de la segmentación está determinada por la frecuencia con que una instrucción salga del cauce. Como las etapas están conectadas entre sí, todas las etapas deben estar listas para proceder al mismo tiempo. El tiempo requerido para desplazar una instrucción, un paso, a lo largo del cauce es un ciclo de reloj. La duración del ciclo está determinada por el tiempo que necesita la etapa más lenta (porque todas las etapas progresan a la vez).

Las etapas en las que se puede dividir la ejecución de las instrucciones MIPS son:

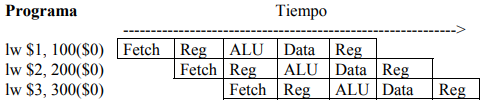
1. ***Instruction Fetch:*** Atrapar las instrucciones de la memoria de instrucciones (o memoria de programa).
2. ***Instruction Decode:*** Decodifica la instrucción y lee los registro.
3. ***Execute:*** Ejecutar la instrucción o calcular una dirección a través de la ALU.
4. ***Memory Access:*** Accesar un operando en memoria de datos.
5. ***Write Back:*** Escribir el resultado en un registro.

Algunas instrucciones no requerirían de todas las etapas de segmentación, a excepción de las instrucciones de carga, las cuales si utilizan todas las etapas (5 etapas). Si por ejemplo, ejecutamos 3 instrucciones de carga en una implementación no segmentada (similar a una implementación multiciclos), la ejecución se comportaría como:



**figura 3.** Tres instrucciones de carga sin segmentación.

Hasta que concluye la primer carga, se continúa con la segunda; y cuando concluya la segunda, se continuará con la tercera. Si utilizamos segmentación, podemos traslapar el acceso a registros de la primera carga con la etapa de captura de la segunda; la operación de la ALU de la primera carga, con el acceso a registros de la segunda y con la captura de la tercera, y así sucesivamente. De manera que la ejecución de las instrucciones tendría el siguiente comportamiento mostrado en la figura 5.4, que evidentemente requiere de menos tiempo.



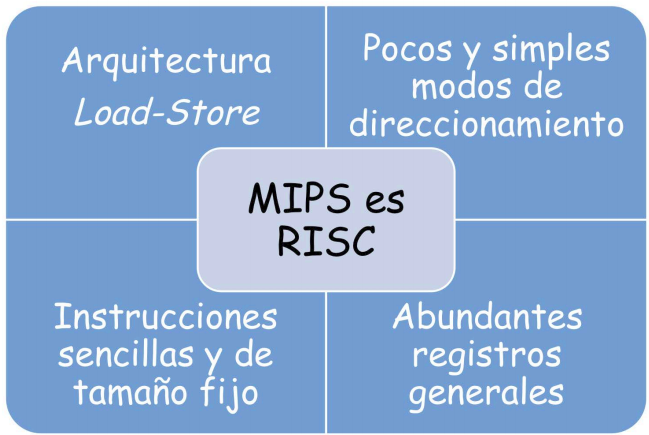
**figura 4.** Tres instrucciones de carga con segmentación.

Para implementaciones segmentadas, el objetivo del diseñador es equilibrar la duración de las diferentes etapas. Si las etapas están perfectamente equilibradas, entonces el tiempo por instrucción de la máquina segmentada - suponiendo condiciones ideales es igual a:

Bajo estas condiciones, la mejora de velocidad debida a la segmentación es igual al número de etapas. Sin embargo, habitualmente, las etapas no están perfectamente equilibradas; además, la segmentación involucró algún gasto. Así, el tiempo por instrucción en la máquina segmentada no tendrá su valor mínimo posible, aunque pueda estar cercano.

***Filmina 3:***

El procesador MIPS, es un procesador RISC (del inglés Reduced Instruction Set Computer, en español Computador con Conjunto de Instrucciones Reducidas).

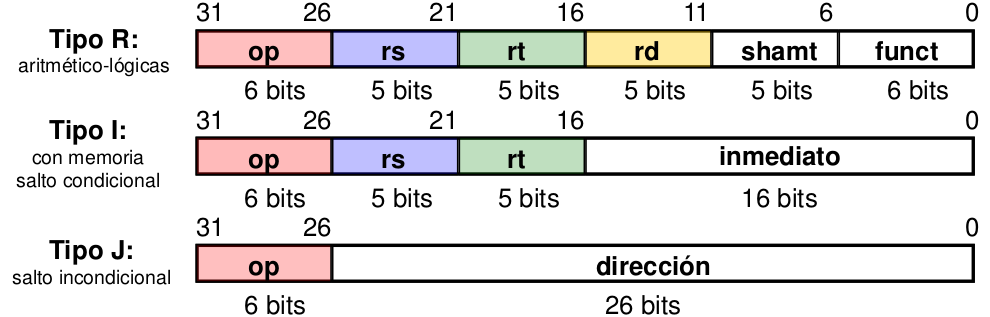


Estos procesadores cumple con las siguientes condiciones:

* Es una ***arquitectura Load-Store***, es decir, para operar con datos, estos deben cargarse desde la memoria principal en registros internos de la CPU, donde, quedan disponibles para operar con ellos. Así, las únicas instrucciones que acceden a memoria principal son las de carga-almacenamiento, esto es, las que mueven los operandos entre memoria principal y los registros de la CPU. Estando los datos en registros de la CPU, el acceso a ellos es muchísimo más rápido que si estuvieran en memoria principal.
* Ya que las únicas instrucciones de acceso a memoria principal son las de carga-almacenamiento, se dispone de ***pocos y sencillos modos de direccionamiento***, lo que facilita enormemente la decodificación de las instrucciones y la obtención de sus operandos.
* La organización del formato de las instrucciones también es muy sencillo, lo que facilita su decodificación, pues dispone de ***pocos formatos***, compartiendo todos la misma ***longitud fija de instrucción***.
* Ya que no se opera con los datos en memoria principal, se hace necesario disponer de un ***generoso conjunto de registros generales*** para albergar los distintos datos del programa.

**Instrucciones del repertorio MIPS**

Todas las instrucciones del repertorio MIPS tiene 32 bits de anchura. Estos 32 bits se utilizan para tres formatos de instrucciones diferentes:



En donde:

***op:*** identificador de instrucción

***rs, rt, rd:*** identificadores de los registros fuentes y destino

***shamt:*** cantidad a desplazar (en operaciones de desplazamiento)

***funct:*** selecciona la operación aritmética a realizar

***inmediato:*** operando inmediato o desplazamiento en direccionamiento a registro-base

***dirección:*** dirección destino del salto

Con este reducido número de formatos y una longitud fija de instrucción, se consigue minimizar el tiempo de decodificación de la instrucción y del cálculo de la dirección de los operandos.

***Instrucciones R-type:*** Las instrucciones con formato de tipo R corresponden a las operaciones de registro a registro en las que interviene la Unidad Aritmético-Lógica. en este caso, el campo Función establece la operación a realizar en la ALU (suma, resta, …, por ejemplo ADD, ADDU, SUB).

***Instrucciones I-type:*** El formato de tipo I es el utilizado por las instrucciones de carga y almacenamiento (load-store, por ejemplo LW, SW) y las instrucciones de salto condicional (ejemplo BEQ, BNE).

***Instrucciones J-type:*** Las instrucciones de tipo J se utilizan exclusivamente para instrucciones de salto incondicional (por ejemplo J, JAL) y retorno de excepciones.

Dentro del conjunto de instrucciones hay dos factores a tener en cuenta:

* Se trata de un conjunto reducido de instrucciones.
* Y cuantos registros le ponemos al procesador.

El ***conjunto reducido de instrucciones*** quiere decir que tiene que ser ortogonal, es decir que con la combinación de dos o tres instrucciones del conjunto no se pueda generar otra, dicho de otra forma todas las instrucciones son tan simples y están tan elegidas que con la combinación de ellas no se puede hacer otra. Como con una combinación de ellas no se puede hacer otra, no hay instrucciones de operaciones contra la memoria principal (osea no existe una instrucción que opere con datos de memoria directamente sino que se utiliza los registros de la CPU para operar con ellos). Por ejemplo si queremos hacer una suma y guardar el dato en memoria principal, no existe una instrucción que haga la suma y guarde el resultado directamente en memoria principal. En lugar de esto lo que se hace es ejecutar la instruccions de suma sobre los registros y luego se almacena el resultado, con los registros, sobre memoria principal. Es decir las instrucciones serán:

|  |
| --- |
| add $t0, $a0, $a1 # carga en reg $t0 la suma entre reg $a0 y reg $a1  sw $t0, 4($zero) # almacenar el word de reg $t0 en mem[$zero+4] |

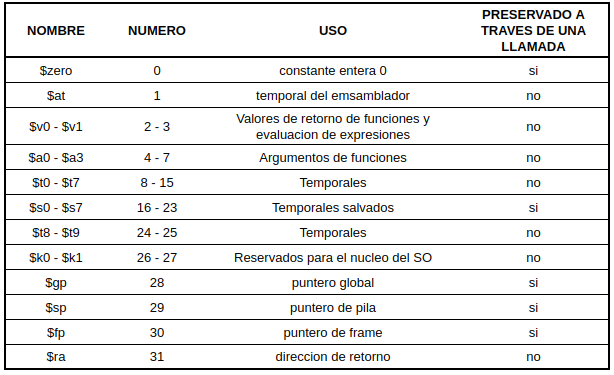
Lo mismo sucede si queremos sumar un valor desde memoria principal. Primero cargamos el valor de memoria en un registro luego operamos con este a través de los registros y por último se carga este en la memoria principal.

|  |
| --- |
| lw $a0, 8($zero) # carga en reg $a0 el valor de memoria[$zero+8]  add $t0, $a0, $a1 # carga en reg $t0 la suma entre reg $a0 y reg $a1  sw $t0, 4($zero) # almacenar el word de reg $t0 en mem[$zero+4] |

Por el lado de la ***cantidad de registros*** que contiene el procesador depende de las instrucciones porque estas instrucciones son de 32 bits, sabiendo la instrucción que más registros utiliza, podemos determinar su tamaño de dirección y con el mismo determinar el direccionamiento. Para el procesador MIPS las instrucciones que más registros usa son las instrucciones R-type, que usan tres registros en una sola instrucción, ya que utilizan un registro destina (rd) y dos registros operandos (rs y rt). Por ejemplo:

|  |
| --- |
| add $t0, $a0, $a1 # carga en reg $t0 la suma entre reg $a0 y reg $a1 |

Por lo tanto observando el tamaño de la dirección de los registros vemos que se reparte en 5 bits para cada una (dentro de los 32 que tiene la instrucción) y de esta manera se puede determinar que con 5 bits solo se puede direccionar hasta 32 combinaciones diferentes (iniciando con la 0 y finalizando en la 31). Es por lo cual el procesador MIPS contiene ***32 registros de CPU***. Estos registros en los procesadores MIPS se reparten de la siguiente manera:



Los bancos de registros del procesador MIPS son 2 y ambos son de 32 registros, el que mostramos arriba, que se trata del banco de registro para enteros. Y además se tiene el banco de registros de punto flotante que también contiene 32 registros.

***Filmina 4:*** En el diagrama de la filmina observamos el ***camino o la ruta de datos*** de una instrucción como también las ***señales de control*** (líneas de puntos) sobre esa ruta de datos (o recursos del procesador).

Este diagrama de la arquitectura MIPS multiciclo define las diferentes funciones (o etapas) que debe tener el procesador.

Lo primero sería buscar la instrucción en memoria principal (podemos considerar que esta tiene las instrucciones y los datos). Para esto se requiere de un registro que me indique en qué dirección sacar la instrucción, esto lo hace el registro PC (contador de programa), es el primer registro del diagrama y su escritura depende de la señal de control ***PCWrite***.



Entonces en memoria tenemos las instrucciones y los datos, por lo tanto en esta memoria se puede escribir, para esto el control de la memoria principal se logra a través de las señales ***MemWrite*** y ***MemRead***. Como tenemos por un lado datos y por otro lado instrucciones debemos saber cuando se accede a un dato y cuando a una instrucción. Para esto se coloca el MUX entre el registro PC y la memoria principal, este MUX será controlado por la señal ***IorD*** que nos indicará si se trata de una lectura de instrucción o una lectura/escritura de datos, en caso de ser una escritura de datos el dato vendrá desde la línea que conecta DW (Write data, que se traduce como dato a escribir, el mismo será sacado de el bus del FileRegister-Banco de Registros, en este caso busB).



Cuando leo desde memoria una instrucción, esta se almacena en el registro de instrucción IR (Instruction Register), en caso de leer un dato este se almacena en el registro de datos MDR (Memory Data Register), ambos registros son escritos si lo permite su señal de control ***IRWrite*** y ***MDRWrite***.

El registro IR es de 32 bits, y contiene la instrucción que será de alguno de los formatos vistos (R-Type, I-Type o J-Type), com vemos en el diagrama los bits se reparten para direccionar sobre los registros del *Banco de Registros*, RA lo direcciona rs (que se encuentra en los bits 25-21 de la instrucción), luego RB lo direcciona rt (bits 20-16 de la instrucción), RW (Write Register, que se traduce como escritura de registro) lo direcciona rt (bits 20-16 de la instrucción) o rd (bits 11-15 de la instrucción) por eso se coloca el MUX controlado por la señal de control ***RegDst*** y *busW* es el dato a escribir en el registro de escritura RW, que puede venir del resultado de la ALU (ALUOut) o directamente desde memoria principal a través del registro MDR que es un dato de 32 bits, es por lo cual se coloca un MUX entre ambos registros controlado por la señal de control ***MemToReg***. El dato del registro indicado en RA se muestra en busA y el dato del registro RB se muestra en busB, ambos valores se almacenan en los registros A y B respectivamente que serán escritos si se activan las señales de control ***AWrite*** y ***BWrite***. Con los registros *busA* y *busB* se realiza una operación a través de la ALU, cuyo operación será determinada a través del *control de la ALU* que depende de la señal de control ***ALUOp***. El resultado de la ALU es almacenado en el registro *ALUOut* cuya escritura es posible si se activa la señal de control ***OutWrite***.



Este registro será un resultado de una dirección de lectura en la memoria principal o bien será un resultado de una operación para almacenar en el *Banco de Registros*, todo esto va depender de la instrucción en ejecución.

La ALU puede operar con valores inmediatos los cuales, como lo indican las instrucciones I-type, vienen de los bits 15-0 de la instrucción, como se tratan de 16 bits se debe extender este dato a 32 bits, esta tarea es llevada a cabo por el bloque de lógica *Extensión de Señal*, también si se trata de un salto el valor inmediato debe multiplicarse por 4 ya que debe ser múltiplo de 4 debido a que el PC incrementa de 4 en 4, es por lo cual esta tarea es llevada a cabo por el bloque de lógica *Shift Left 2 (<<2)* que es otra de las posibilidades para el uso de la ALU, otra alternativa es que solo se requiere calcular el proximo PC por la cual esto se logra adicionando 4 al valor del PC actual, de esta manera vimos como según sea la instrucción puede variar la decisión de la ALU sobre cuál va ser su segundo operando, esto es controlado a través del MUX de 4 entradas con la señal de control ***ALUSrcB***.

***Filmina 5:***

El diagrama de estado del procesador multiciclo me muestra cada una de los estados que va teniendo el procesador por cada una de las instrucciones a medida que va avanzando por el datapath (ruta de datos). También podemos observar como cada uno de los estado se asocia a cada una de las etapas, que más adelante veremos que son las etapas en las que se segmenta el procesador.

***Función Búsqueda de instrucción (IF)***: Los estados de esta función son que con el valor actual del PC busco la instrucción desde memoria principal y la carga en el registro IR, luego se incrementa el contador a la próxima instrucción (PC+4).

***Función decodificación de la instrucción (ID):*** Desde el registro IR podemos conocer cuales son los las direcciones de los registros fuentes rs y rt y a través del Bano de Registro de esta manera podemos obtener cuales son los datos de los registros almacendolos en los reg A y B asociados a las direcciones de los registros mencionados.

***Función Ejecución (EX):*** Según sea el tipo de instrucción en esta función la ALU se encargará de obtener un resultado diferente de acuerdo a la instrucción que lo solicite.

R-type: si se trata de una instrucción de tipo R la salida de la ALU ALUOut será el resultado de A funt B.

I-type(LW-SW): si se trata de una instrucción de tipo I la salida de la ALU ALUOut será el resultado de A + SignExt(inmed). Es decir opera con valores inmediatos.

I-type(LW-SW): si se trata de una instrucción de tipo I, por ejemplo BEQ la salida de la ALU ALUOut no me interesa pero si su flag Zero por lo cual es útil el resultado, que depende del resultado de los operandos A - B, si Zero =0 sigo con el próximo PC, si Zero=1 se calcula el número de saltos con el valor del PC.

***Función de acceso a memoria (MEM):*** En esta función de acceso a memoria, los estados se basan en cargar en memoria principal o escribir sobre la misma. En el primer caso La dirección será indicada por el registro ALUOut y el dato de la dirección se almacena en registro MDR. Si se trata de una escritura el dato proviene del registro B, por lo cual el mismo es cargado en memoria principal según la dirección que me indique el registro de salida de la ALU ALUOut.

***Función Write-Back (WB):*** Los estados de esta función, según sea el tipo de instrucción se tendrán estados diferentes.

R-type: Para el caso de las instrucciones de tipo R el estado es escribir el registro ALUOut en el Banco de Registros sobre la dirección destino rd.

I-type: Para el caso de las instrucciones de tipo I el estado es escribir el registro MDR en el Banco de Registros sobre la dirección destino rt.

I-type: Para el caso de las instrucciones de tipo I instrucción BEQ, el estado es escribir el registro PC en PC+4\*SignExt(inmediato). Caso de determinar la próxima dirección a través del número de saltos solicitados por la instrucción y el Pc actual.

Conclusiones:

* El diagrama de estado me muestra en que partes de cada tipo de instrucción debo accionar las señales de control.
* De este diagrama se concluye también que puedo ejecutar en un solo ciclo todas las funciones (o etapas) sobre diferentes instrucciones, esta es la clave de la segmentación o técnica pipelining.
* Cada ciclo del procesador multiciclo se asocia a una función diferente, por ejemplo el ciclo uno se encarga de La función Búsqueda de Instrucción, de esta forma nos podemos dar cuenta que señales de control debemos accionar o utilizar por cada uno de los ciclos que se correspondan diferentes funciones. Siguiendo el mismo ejemplo las señales a accionar en el ciclo 1 serán:
  + Primero: IorD, para indicar la lectura de una instrucción.
  + También primero: MemRead, para sacar la instrucción de memoria.
  + Segundo: IRWrite para tomar la instrucción asociada a esa dirección de memoria (PC).
  + Y por último PCWrite para determinar la próxima dirección de instrucción.
* Con las señales de control algunas instrucciones cambian las direcciones de la memoria o el banco de registro pero no cambian los registros (IR, MDR, A, B) y otras si los cambian.
* El procesador multiciclo es un circuito que en cinco ciclos de reloj logra ejecutar todas las funciones (IF, ID, EX, MEM, WB) que yo necesito realizar por cada instrucción.

***¿Que es mejor, un procesador monociclo, un multiciclo o un pipeline?***

En primera medida podríamos decir que el procesador monociclo ejecuta una solo instrucción en un solo ciclo mientras que el multiciclo lo hace en cinco ciclos, pero el procesador monociclo al realizar todo en un solo ciclo tiene la desventaja que su ciclo es muy largo y queda determinada por la instrucción más larga. Mientras que por el lado del procesador multiciclo esto se evita haciendo que el ciclo no sea tan largo y de esta manera se podría decir que sería más eficiente, ya que su arquitectura permite utilizar ciclos más pequeños y de esta manera es mejor en cuanto a rendimiento.

Por el lado del pipeline se podría decir que es mejor aún ya que su arquitectura me permitiría, aparte de tener un ciclo corto, ejecutar las diferentes etapas de un flujo de instrucción. Es decir una instrucción ba estar en la etapa IF, mientras otra en la ID, otra en la EX, otra en la MEM y una última en la WB todo en el mismo ciclo reloj. Pero el único problema es que se deberán tener en cuenta los riesgos (estructurales, de datos y de control) que se presentan en un pipeline para que esto funcione correctamente.

***Filmina 6:*** Además de los riesgos, el pipeline debe soportar interrupciones y excepciones.

***¿Cuando atendemos las interrupciones y cuando las excepciones ?***

***Excepciones:***

La atención de una excepción depende de cuándo se produce. Estas se producen cuando se produce una división por cero, por overflow, o por tener un acceso a una posición de memoria no permitida o por un fallo de página.

Entonces la excepción se produce por alguna de esas causas por lo cual debo resolverla y una vez resuelta volver a la instrucción para continuar con el código normalmente. Por ejemplo si tengo un fallo de página, me di cuenta durante la ejecución de una parte de la instrucción , ya que calcule la dirección a donde quiero buscar la pagina pero resulta que cuando la busca esta no esta en memoria, por eso debo ejecutar la excepción que traiga la página y una vez que está en memoria continuar con la instrucción.

***¿Cuantas excepciones juntas (anidadas) pueden producirse en un programa?***

Por ejemplo si ejecuto un programa y se produce una excepción por overflow, o división por cero, (si la capturó por try y catch) entonces desde la captura me manda a la rutina de excepción, y por ejemplo el código de esa rutina no está en memoria entonces con otra excepción, por fallo de página lo busco en memoria a través de otra excepción que me trae la página en memoria. De esta forma las excepciones que pueden producirse son dos, o mejor dicho a dos niveles (usuario y kernel). Es decir en la aplicación puedo tener varias excepciones pero si desde el nivel de aplicación/usuario salto a una excepción a nivel de kernel, como lo son los fallos de página no puede superarse este nivel de excepción ya que es el de mayor privilegio. Dicho de otro forma si se provoca una excepción a nivel de kernel no puede producirse otra excepción en este nivel ya que llegan hasta ese nivel, resolviendo la excepción a este nivel se retomará con la instrucción que lanzó la excepción y luego podrán tomarse más excepciones si se producen.

***Interrupciones:*** Las interrupciones se producen en cualquier momento desde el exterior o interrupciones por software como las de timer y serán atendidas luego de que se termine de ejecutar la instrucción en curso, es por esto que se las denomina asíncronas a la ejecución del programa.

Los programa en ejecución deben mandar ***donde se produjo la interrupción***, esto lo determinan a través del ***PCE*** (contador de programa de excepciones) se trata de otro contador de programa que contendrá la dirección de la instrucción que causó la instrucción. Por el lado del registro ***PC*** será en donde se cargará la dirección base del conjunto de instrucciones encargadas de atender la excepción o interrupción.

Luego hay un registro denominado ***causa*** que contendrá algún código asociado a la causa en concreto.

***Filmina 7:*** El nuevo diagrama de estado del procesador multiciclo nos muestra, cómo se pueden tener en cuenta las excepciones e interrupciones. Para esto vemos como se agregan nuevos estados asociados a las mismas. Con este diagrama de estado podemos saber donde se producen las excepciones o interrupciones y podemos saber qué estados se llevan a cabo como también que señales de control agregar. Por ejemplo durante el ciclo de la función de Decodificación de la instrucción (IF) podemos darnos cuenta de que detectamos una *instrucción no definida*, entonces dentro de este estado se carga el registro PC con la dirección que atiende la excepción, se indica qué instrucción causó la excepción con PCE=PC-4 y se completa el registro causa que indicara algún código que se asocia a la causa en concreto.

Lo mismo sucede cuando se produce un *overflow*, será un nuevo estado detectado en el ciclo de la función EX, ya que el overflow puede darse a partir de una operación de la ALU. En este estado se deberá cargar de la misma forma el registro PCE, el registro PC y el registro causa.

***Filmina 8:*** Explicación de segmentación. Esto lo explicamos anteriormente en los títulos del presente resumen *Concepto de segmentación* y *segmentación en los procesadores*. Se destaca que la segmentación nos proporciona en el procesador un paralelismo temporal sobre las instrucciones, es por lo cual se ejecutan flujos de instrucciones por ciclo.

***Filmina 9:*** Esta filmina nos muestra las diferencias que hay entre cada uno de las diferentes arquitectura de procesadores, monociclo, multiciclo y segmentado (o pipelining).

Como sabemos la instrucción que consume todos los recursos del procesador para ejecutarse adecuadamente es una Load por ejemplo LW.

**Monociclo:**

En el monociclo esta instrucción se ejecuta en un solo ciclo, pero su arquitectura tiene tanta lógica en un solo módulo que su ciclo es muy grande, como consecuencia las frecuencia a la que pueden ser utilizados estos procesadores son muy bajas, dichos con otras palabras la arquitectura de estos procesadores se adapta a una frecuencia muy baja.

**Multiciclo:**

Por el lado del multiciclo, vimos que la arquitectura divide en ciclos cada una de las funciones para ejecutar cada instrucción . De esta manera cada ciclo utiliza una pequeña porción de lógica lo que hace que el ciclo sea más pequeño y con múltiples ciclos ejecutará una instrucción completa. Para el caso de una LW demandará el máximo de ciclos por instrucción que sería de cinco ciclos. A pesar de ser 5 ciclos su rendimientos es mejor ya que la arquitectura permite que se adapten frecuencia mucho más altas debido a los pequeños bloques de lógica que hay por función. De esta forma prácticamente los ciclos son casi cinco veces más chicos que los del monociclo.

**Segmentado:**

Por último con la arquitectura segmentada nos damos cuenta que cada etapa se asocia a una función y en un mismo ciclo cada una de estas puede ser ejecutada sobre un flujo de instrucciones. Esto nos brinda un paralelismo temporal, es decir mientras una instrucción está en la etapa 1, la anterior está en la etapa 2, la anterior a esta ultima esta en la etapa 3, la otra en la etapa 4 y por último la primer instrucción que ingresó está en la etapa 5 y de esta manera continúa avanzando mientras más instrucciones ingresen. Por lo cual se tiene un ciclo pequeño y durante este ciclo se aplica a los diferentes bloques pequeños de lógica asociados a las etapas de cada una de las diferentes instrucciones del flujo de instrucciones. Esto permite que la arquitectura del procesador se adapte a frecuencias altas y obtener un alto rendimiento por el paralelismo temporal.

**Riesgos en la segmentación**

Filmina 10: Se presentan los riesgo de la segmentación.

***Conceptos de Riesgo en la segmentación:*** Situaciones que impiden que cada ciclo se inicie la ejecución de una nueva instrucción. Hay tres tipos de riesgos:

**Riesgos de Estructura:** Significa que el hardware no puede soportar la combinación de instrucciones que se quiere ejecutar en el mismo ciclo.

En un lavadero de ropa, por ejemplo, si la lavadora y la secadora formaran parte del mismo equipo, no sería posible trabajar estas dos etapas con dos cargas de ropa diferentes.

En un procesador, si contamos con una sola memoria, para datos y código, no será posible escribir o leer un dato, mientras se atrapa una instrucción. Para evitar estos riesgos, se debe definir correctamente al camino de datos, en este caso un banco de memoria para datos y otro para instrucciones solucionaría el problema, caso contrario se presenta el riesgo.

**Riesgos por dependencias de datos:** Una instrucción depende del resultado de una instrucción previa que aún está en la segmentación. Esto es porque tiene prioridad la lectura de los datos por sobre su escritura, es decir su escritura se realiza en la etapa 5, mientras que su lectura en la etapa 2, con esto determinadas situaciones de lecturas requerirán antes el dato que se escribe en la etapa 5.

**Riesgos de Control:** Surgen de la necesidad de hacer una decisión basada en los resultados de una instrucción, mientras otras se están ejecutando.

Por ejemplo, en un salto condicional, mientras se determina si el brinco se hará o no, otras instrucciones ingresarán al procesador. Si se determina que el brinco no se llevará a cabo, esas instrucciones prosiguen su ejecución. Pero si el brinco no se realiza, habrá que evaluar cómo eliminar a las instrucciones que ya ingresaron al procesador. Estos riesgos se revisan en la última sección, resolviendolos por otra técnica conocida como detenciones.

Los riesgos deben ***detectarse*** y ***resolverse***. La solución a los riesgos son:

* Riesgos estructurales.
  + Se solucionan con el adecuado diseño del *datapath* y la *unidad de control*.
* Riesgos por dependencia de datos.
  + Solución por anticipación.
  + Solución por detenciones.
* Riesgos de control (en saltos).
  + Se solucionan con la ayuda de la señal de control IFlush.

***Filmina 11:***

En esta filmina observamos que para evitar los riesgos estructurales el objetivo sería: Ejecutar sin conflicto cualquier combinación de instrucciones.

De esta manera notamos que en relación a:

***Memoria:***

Si esta fuera una sola memoria tanto para datos como para instrucciones, se presentaría el riesgo porque como vemos en un flujo de instrucciones se daría el caso de que una instrucción 2 escribe en memoria mientras la instrucción 5 estaría leyendo en memoria un nueva instrucción y chocaron las decisiones. Es por lo cual, la arquitectura del MIPS segmentado, tiene en su Datapath (ruta de datos) un banco de memoria para las instrucciones y un banco de memoria para los datos. Y mejor aún si se utilizan dos memorias caché una para instrucciones y otra para datos, lo que permitiría mejores tiempos de acceso.

***Registros:*** En este caso la etapa ID de la instrucción 1 lee un registro y la etapa WB de la instrucción 4 escribe un registro. Para evitar que choquen las decisiones la solución se basa en utilizar un flanco para la lectura y el flanco opuesto dentro del ciclo para la escritura.

***ALU:*** Para este recurso, distintas etapa pueden requerir de la ALU, por lo cual se nota en este caso que la instrucción 3 requerirá de la ALU para calcular una función sobre los operandos y la instrucción 5 requiere de la ALU para determinar el valor del próximo PC, con una sola ALU chocarian las decisiones, para evitar esto se colocan sumadores de manera separada para la determinación del próximo PC en la etapa 1 (IF), dejando exclusivamente la ALU para las operaciones aritméticas lógicas de la etapa 3 (EX).

En este caso distintas etapas requieren del uso de la ALU para:

- Realizar un cálculo aritmético lógico.

- Sumar 4 al PC.

- Sumar direcciones para el cálculo de un salto.

Veremos como el datapath diseñado correctamente con los sumadores en su lugar y la ALU en el suyo resuelven este inconveniente.

***Filmina 12:***

En esta filmina vemos el diseño inicial que debería tener nuestro MIPS segmentado. La idea es que los choques de decisiones que puedan producirse por el uso de la ALU la memoria o los registros, se solucionan con la duplicación de recursos, de esta manera los riesgos estructurales desaparecen. En este caso vemos como utilizando sumadores independientes para la etapa IF y para el cálculo de saltos incondicionales se evita tener riesgos estructurales. Dicho con otras palabras:

* Para lograr que se incremente el PC en PC+4, lo realiza independientemente un sumador directamente en la etapa 1 (IF).
* Para lograr sumar direcciones para el cálculo de un salto incondicional se utiliza un sumador independiente a la ALU para que haga el cálculo.

Y por último para lograr que el uso de memoria sea correcto se utiliza una memoria para instrucciones y otra exclusiva para datos, de esta manera no hay riesgos estructurales.

Para el caso de los registros la solución es como dijimos teniendo en cuenta utilizar los dos flancos de un ciclo, uno para la lectura y el otro para la escritura, se busca siempre que primero sea la lectura.

***Filmina 13:***

Los riesgos de datos se producen si existe dependencia de datos entre instrucciones que se ejecutan concurrentemente. Como se ejecutan 5 instrucciones paralelamente es en donde se debe resolver el problema.

La idea es la siguiente, si tenemos las siguientes operaciones de datos por instrucción:

|  |
| --- |
| Si a = b + c  Sj c = a + e  Sk a = d + e |

***Donde la instrucción i precede a j***. Es decir la instrucción i está una instrucción antes que la instrucción j.

Lo que debería suceder en el juego de instrucciones es que en la ***instrucción Si*** se calcule el dato a con los datos b y c, luego teniendo este dato la ***instrucción Sj*** hará uso del dato a para calcular el nuevo dato c a través del dato a, previamente calculado y el dato e. Por último en la ***instrucción Sk*** se actualiza nuevamente el dato a con los datos d y e. El problema es que la segmentación hace que esto no se realice como debería ser, debido a que como una instrucción se realiza en paralelo con otras, por etapas, luego de la 5ta etapa se tendría los datos correctos de una instrucción, pero entre instrucciones precedentes se requiere utilizar los datos antes de la escritura de los mismos, ya que hay cinco instrucciones en un mismo ciclo.

Para determinar los riesgos se define los siguientes conceptos:

***Dominio:*** operandos de la instrucción (rs y rt).

***Rango:*** Resultado de la instrucción (rd).

Las situaciones que pueden plantearse si i precede a j son:

**:** Dominio de la *instrucción i* intersección con dominio de la *instrucción j* no produce ningún riesgo. Es decir uno o todos los operandos de la instrucción i es igual a los operandos de la instrucción j, esto no provoca riesgos. Por ejemplo:

|  |
| --- |
| Si a = **b** + **c**  Sj d = **c** + **b** |

**Conclusión:** No hay riesgo.

: Dominio de la *instrucción i* intersección con rango de la *instrucción j* produce riesgo EDL (Escritura después de Lectura). Es decir uno o todos los operandos de la instrucción i son iguales al resultado de la instrucción j existe riesgo EDL, ya que si escribo antes el resultado de la instrucción j, después la lectura de los operandos de la instrucción i será incorrecta. Por ejemplo:

|  |
| --- |
| Si a = **b** + **c**  Sj **b** = e + f |

**Conclusión:** Una instrucción posterior trata de escribir un resultado antes de que haya leído la instrucción anterior.

**:** Rango de la *instrucción i* intersección con dominio de la *instrucción j* produce riesgo LDE (Lectura después de Escritura). Es decir si el resultado de la instrucción i coincide con uno o todos los operandos de la instrucción j existe riesgo LDE, ya que si escribo después que la instrucción j el resultado de la instrucción i, la lectura de los operandos de la instrucción j serán incorrectos. Por ejemplo:

|  |
| --- |
| Si **a** = b + c  Sj d= **a** + **f** |

**Conclusión:** Una instrucción posterior trata de leer un operando antes de que lo escriba la instrucción anterior.

**:** Rango de la *instrucción i* intersección con rango de la *instrucción j* produce riesgo EDE (Escritura después de Escritura). Es decir si el resultado de la instrucción i coincide el resultado de la instrucción j existe riesgo EDE, ya que si escribo el resultado de la instrucción i, después que la instrucción j, será incorrecto porque el último resultado debería ser el de la instrucción j. Por ejemplo:

|  |
| --- |
| Si **a** = b + c  Sj **a** = e + f |

**Conclusión:** Una instrucción posterior trata de escribir un resultado antes de que lo escriba la instrucción anterior.

***Filmina 14:*** El diagrama nos muestra como los riesgos EDL y EDE están resueltos por la arquitectura que tiene la ruta de datos del pipe, ya que una escritura de una instrucción j nunca podrá realizarse antes que una lectura de una instrucción precedente (anterior, instrucción i) también una escritura de una instrucción j nunca podrá realizarse antes que la escritura de una instrucción i precedente. Es decir que la arquitectura garantiza en el pipe :

* Que se lean los registros en el final de la segunda etapa.
* Todas las instrucciones escriben en la última etapa.
* Todas las instrucciones tienen igual duración.

En conclusión por estos puntos, están resueltos los riesgos EDL y EDE.

***Filmina 15:*** El diagrama nos muestra la cantidad de ciclo que deberían esperar las instrucciones siguientes a la instrucción i para que sus lecturas escrituras sean correctas, cuando se presentan riesgos de datos. Por ejemplo la instrucción siguiente a i, es decir i+1, debería esperar después de su segunda etapa dos ciclos más para que recién en el tercer ciclo pueda obtener el correcto valor que escribió en REG la instrucción i.

Se plantea que sobre los riesgos de datos si la instrucción i precede al resto de instrucciones y como las siguientes instrucciones requieren de los registros en la segunda etapa, las instrucciones que serán afectadas serán ***i+1*** e ***i+2*** el resto se libran de los riesgos. Estos valores representan la ***distancia en ciclos*** a la que serán afectados por los riesgos de datos desde la instrucción i, en este caso es de 2 ciclos. Luego se nota que desde la instrucción i a tres ciclos más adelante (i+3) las instrucciones están libres de riesgo de datos.

También se el diagrama muestra cómo las líneas que van para atrás, es decir que las instrucciones posteriores requieren los datos antes de tiempo (líneas rojas) y son las que muestran el inconveniente por estar afectadas al riesgo de datos. Mientras que líneas hacia adelante (líneas verdes) muestran que no hay problema en el tiempo ya que en ese tiempo el dato está actualizado.

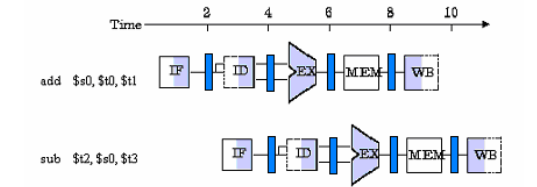
## **Riesgos por dependencia de datos (solucionados por cortocircuito)**

Los Riesgos por dependencias de datos ocurren cuando la ejecución de una instrucción depende del resultado de una instrucción previa que aún está en la segmentación. Por ejemplo, si se ejecutan las instrucciones:

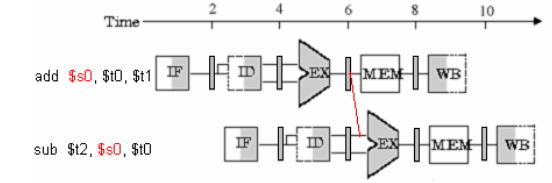
*add* ***$s0****, $t0, $t1*

*sub $t2,* ***$s0****, $t3*

De acuerdo a las etapas en las que se dividió la ejecución de instrucciones, el resultado de la suma se escribirá al final de la quinta etapa en el registro $s0 y la resta hace su lectura de registros en la segunda etapa. Suponiendo que cada etapa tarda 2 ns, el resultado de la suma se escribe a los 10 ns, y la resta esperaba leer el registro a los 4 ns, como se muestra en la siguiente figura, la resta leerá el valor anterior de $s0.



Para resolver este problema se propone capturar el dato una vez obtenido de la etapa 3 como se nota en la siguiente figura:



Al esquema mostrado en la figura anterior se le conoce como ***anticipación*** o también ***cortocircuito***. La anticipación es una técnica para resolver los riesgos por dependencias de datos que consiste en tomar el dato de uno de los operandos de la ALU antes de que éste sea escrito en el registro destino.

Tiene como ventaja que los riesgos se resuelven a nivel de hardware, por lo que el compilador no debe insertar instrucciones NOP y por lo tanto, no provoca retrasos en la ejecución de un programa.

## **Solución por anticipación**

La anticipación tomará el dato de uno de los operandos de la ALU antes de que éste sea escrito en el registro destino. Se considera la siguiente secuencia de código para obtener los posibles casos:

sub **$2**, $1, $3 # El registro $2 es escrito por sub

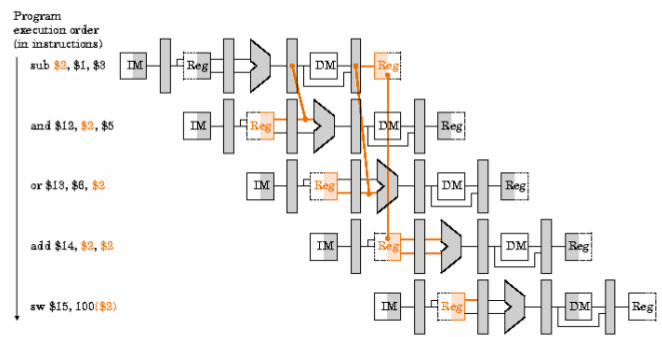
and $12, **$2**, $5 # El 1 er operando ($2) depende de sub

or $13, $6, **$2** # El 2o operando ($2) depende de sub

add $14, **$2**, **$2** # Los 2 operandos dependen de sub

sw $15, 100( **$2** ) # El registro base ($2) depende de sub

En la siguiente figura se muestra la representación de múltiples ciclos de la ejecución de esta secuencia. Considerando como ciclo de reloj 1 cuando la instrucción SUB entra a la segmentación, se tiene que la instrucción SUB genera con la ALU el valor que se escribirá en $2 y lo escribe en el registro EX/MEN al final del ciclo de reloj 3.



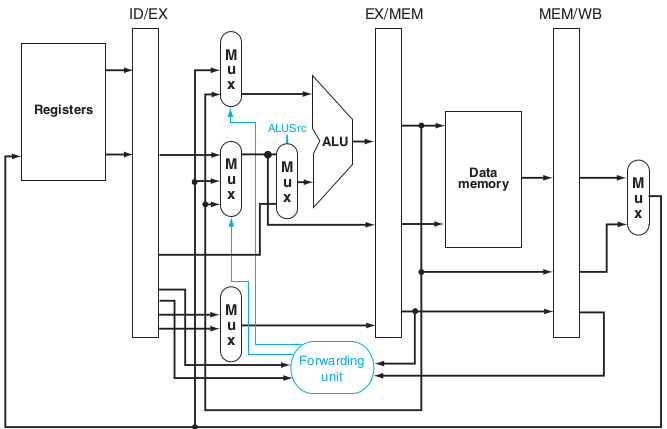
Al comienzo del ciclo de reloj 5, la OR toma el valor de $2 del registro MEM/WB y lo regresa a la etapa 3 para que sea su segundo operando de la ALU. Durante ese ciclo se espera que se haga la escritura del registro $2 para que su valor sea usado por la instrucción ADD durante el mismo ciclo.

Esto es posible si los registros que separan las etapas de la segmentación y el Banco de registros (también denominado *archivo de registros*) activan su escritura en flancos diferentes, así, durante el flanco de subida se puede escribir el registro MEM/WB con la información y las señales de control correspondientes, durante la primera mitad del ciclo estas señales se establecen, de manera que cuando ocurre el ciclo de bajada, el banco de registros escribirá la información correcta (la instrucción SUB escribirá el valor de $2) y durante la segunda mitad del ciclo, se puede disponer de los datos actualizados para su lectura (la instrucción ADD leerá el nuevo valor de $2).

Entonces, la anticipación puede hacerse de diferentes lugares, puede ocurrir que el dato a anticipar se encuentre en el registro EX/MEN o en el registro MEM/WB, y en ambos casos puede tratarse del primero o del segundo operando, incluso de ambos.

En todos los casos, las anticipaciones involucran a los operandos de la ALU y la ALU está en la etapa 3, en una implementación sin anticipaciones los dos operandos se toman directamente del registro ID/EX. Al considerar las anticipaciones, se debe saber cuál será el registro destino en las instrucciones que están en las etapas MEM y WB, y si se va a escribir dicho registro. Si ese registro coincide con alguno de los operandos de la etapa EX, su valor debe regresar a la etapa EX para que sustituya a ese operando. En la próxima figura se muestra la segmentación con la incorporación de la ***Unidad de anticipación (Forwarding Unit)***.

La unidad de anticipación compara cada uno de los operandos de la ALU (rs y rt) con el registro destino de la etapa MEM (EX/MEM.RegistroRd) y con el registro destino de la etapa WB (MEM/WB.RegistroRd) si existe igualdad entre un par de ellos, y la señal de escritura en registro esta acertada en la etapas MEM (EX/MEM.RegWrite) o en la etapa WB (MEM/WB.REgWrite) , la unidad de anticipación debe colocar el valor adecuado para sustituir a algún operandos, controlando los ***multiplexores de anticipación***.



Las ***decisiones*** que tomará la unidad de anticipación con respecto a la etapa MEM son:

if ( EX/MEM.RegWrite AND EX/MEM.RegistroRd != 0 AND

EX/MEM.RegistroRd == ID/EX.RegistroRs )

**ForwardA = 10**

if ( EX/MEM.RegWrite AND EX/MEM.RegistroRd != 0 AND

EX/MEM.RegistroRd == ID/EX.RegistroRt )

**ForwardB = 10**

Respecto a la etapa WB, la unidad de anticipación debe ***determinar***:

if ( MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND

MEM/WB.RegistroRd == ID/EX.RegistroRs )

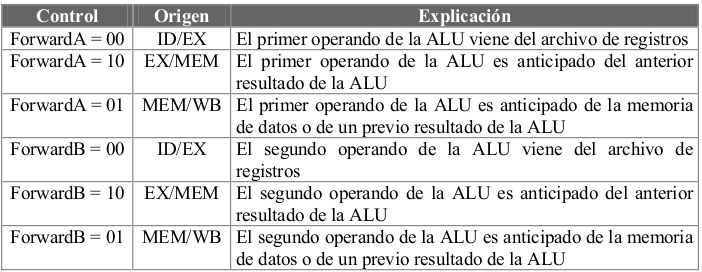
**ForwardA = 01**

if ( MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND

MEM/WB.RegistroRd == ID/EX.RegistroRt )

**ForwardB = 01**

***ForwardA*** y ***ForwardB*** son señales de dos bits que controlan los multiplexores de anticipación y determinan cuáles serán los operandos de la ALU tal como se detalla en la siguiente tabla.



Una complicación se presenta si la etapa WB presenta un resultado para un registro y la etapa MEM presenta un resultado diferente para el mismo registro, y ese registro es un operando de la ALU. Por ejemplo, si tenemos el siguiente código:

add **$1**, **$1**, $2

add **$1**, **$1**, $3

add **$1**, **$**1, $4

Cuando la segunda suma está en la etapa EXE, anticipa sin problema el valor del registro $1 de la etapa MEM (el cual fue generado por la primera suma).

Sin embargo, cuando la tercera suma alcanza la etapa EXE, tiene dos posibilidades para anticipar el valor de $1, en la etapa MEM se encuentra el valor generado por la segunda suma y en la etapa WB se encuentra el valor generado por la primera.

En ese caso, el resultado que debe anticiparse es el de la etapa MEM por que es el resultado más reciente. Por lo tanto, se deben modificar las decisiones que toma la unidad de anticipación respecto a la etapa WB, para no entrar en conflicto con la etapa MEM. Ahora se tendrá (las nuevas comparaciones se agregan en rojo):

if ( MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND

***EX/MEM.RegistroRd != ID/EX.RegistroRs*** AND

MEM/WB.RegistroRd == ID/EX.RegistroRs )

**ForwardA = 01**

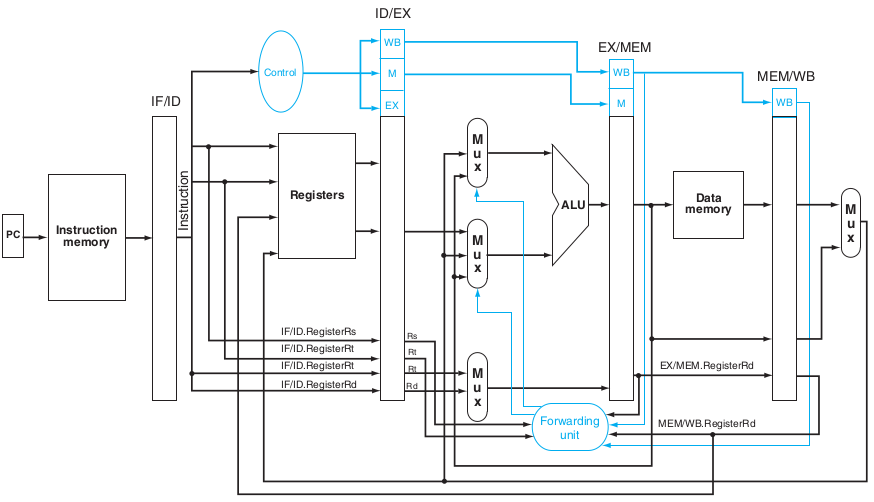
if ( MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0 AND

***EX/MEM.RegistroRd != ID/EX.RegistroRt*** AND

MEM/WB.RegistroRd == ID/EX.RegistroRt )

**ForwardB = 01**

En la próxima figura se muestra el camino de los datos segmentado y el control, con la unidad de anticipación. Es importante aclarar que solo se esboza el camino de los datos, se omiten algunos detalles, como el multiplexor que se presenta en la segunda entrada de la ALU o el mismo control de la ALU, por que el punto importante en esta sección es el estudio de la unidad de anticipación.



***Filmina 16:*** Esta filmina muestra un diagrama en donde observamos parte de la que plantea la teoría anterior. Como con la técnica de la anticipación o cortocircuito, podemos obtener los datos desde las registros que manejan las etapas y de esta forma detectando el caso se los conecta directamente a los operando que los requieran. En este caso vemos como las líneas que eran rojas ahora se reemplazan por líneas verdes ya que los datos son actualizados al tiempo que se los requiere. Algunos se los obtiene del registro *EXMEM* mientras que otros del registro *MEMWB* el resto vemos que se obtiene desde la etapa WB es decir cuando ya no se presenta el riesgo. Todo esto lo ***detecta*** y ***soluciona*** la ***Unidad de anticipación o cortocircuito (Forwarding Unit)*** una unidad de control dedicada especialmente a detectar este tipo de riesgos. Además como se nota en la arquitectura la unidad utiliza un conjunto de multiplexores para determinar cuándo se presenta cada uno de los casos. Con esta unidad se resuelven todos los problemas por dependencia de datos salvo para las instrucciones load (por ejemplo LW).

## **Riesgos por dependencia de datos (solucionados por detenciones)**

Vimos que para la mayoría de las instrucciones la unidad de cortocircuito soluciona el problema. Pero ahora, que ocurre cuando se intenta ejecutar la siguiente secuencia:

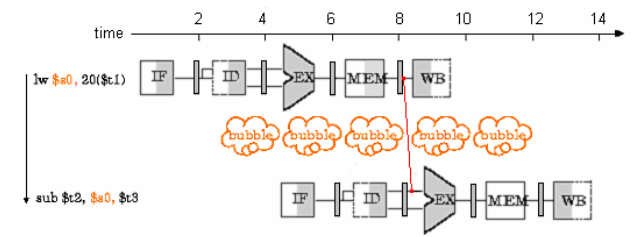
lw **$s0**, 20 ( $t1 )

sub $t2, **$s0**, $t3

En este caso (en instrucciones de carga), la carga obtendrá el dato de memoria hasta el final de la etapa 4, por lo que aún con la anticipación la resta no puede disponer del valor correcto para $s0, porque lo necesita un ciclo de reloj antes de que la carga lo obtenga.

Lo que es un hecho es que la resta se debe detener un ciclo de reloj, para que luego pueda usarse la anticipación, esta técnica se denomina ***detención***. Las detenciones son necesarias en aquellos casos en que la anticipación no es suficiente. Una detención consiste en la ***inserción de una burbuja*** entre las dos instrucciones en conflicto. A una burbuja la definiremos como un conjunto de señales inofensivas, es decir, que no afectan la ejecución del programa, dicho de otra forma las señales de control se establecen en valores de manera tal que durante un ciclo no se hace nada.

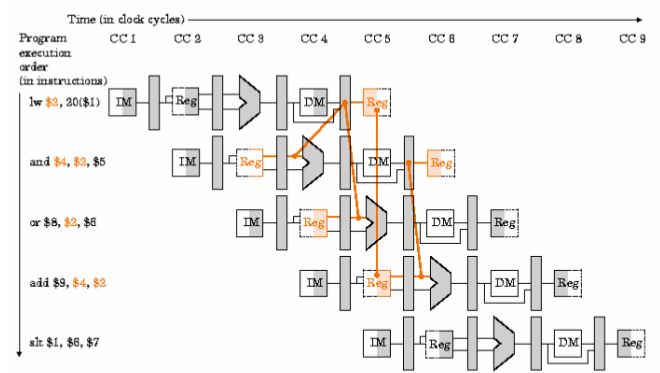
En la próxima figura se muestra como después de insertar una burbuja, ya es posible anticipar el resultado esperado en $s0 para usarse en la instrucción siguiente. Las burbujas se insertan al nivel de hardware, por lo que son transparentes al compilador. Es un hecho que las burbujas producen un retraso de un ciclo de reloj, pero solo se insertarán cuando el dato a cargar en un registro será utilizado como operando de la ALU en la siguiente instrucción.



Para los riesgos por dependencia de datos se debe tomarse en consideración que el camino de datos es único y que los resultados de una etapa se escriben en alguno de los registros encargados de separar etapas. De manera que las líneas rojas indican que de ahí se tomará la información y se regresará a una etapa anterior para que se utilice por las instrucciones subsecuentes.

## **Solución por detenciones**

Como se describió en la sección anterior, la anticipación no puede resolver aquellas situaciones en las que se realizará la carga de un registro (LW) y el dato a cargar es un operando de la instrucción siguiente. En la siguiente figura se muestra un ejemplo del problema, se tiene una instrucción que cargará un dato de memoria al registro $2 seguida por una AND cuyo primer operando es $2.



En el ciclo de reloj 4, el dato que se escribirá en $2 se está leyendo de la memoria y la ALU lo requiere en ese mismo ciclo, no es posible anticipar el valor de $2 porque se escribirá en el registro de segmentación hasta el final del ciclo y se podrá disponer de él hasta el siguiente ciclo. Por lo tanto, se debe detener a la instrucción AND por un ciclo de reloj, para que en el ciclo de reloj 5, con ayuda de la unidad de anticipación, pueda operar sobre el valor correcto de $2.

De manera que, además de la unidad de anticipación, es necesario agregar una unidad de detección de riesgos (Hazard detection Unit), que opere en la etapa ID, de manera que pueda insertar una burbuja entra la carga y la instrucción que usará el dato a cargar. Esta nueva unidad evaluará la siguiente condición:

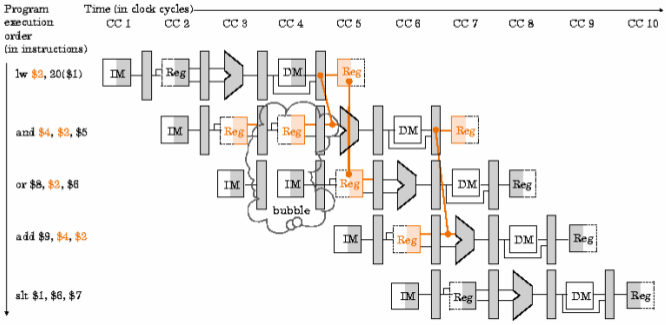
**if** ( ID/EX.MemRead AND

( ID/EX.RegistroRt = IF/ID.RegistroRs OR

ID/EX.RegistroRt = IF/ID.RegistroRt ) )

**Inserta una Burbuja a la segmentación**

La primera condición detecta si la instrucción que está en la etapa EX es una carga; las siguientes dos condiciones evalúan si el registro a cargar (ID/EX.RegistroRt) coincide con alguno de los dos operandos de la instrucción que está en la etapa ID (IF/ID.RegistroRs o IF/ID.RegistroRt ), si se cumplen las condiciones, se debe detener la ejecución de la instrucción que está en la etapa ID por un ciclo de reloj y en su lugar insertar una burbuja, esto se muestra en la próxima figura.



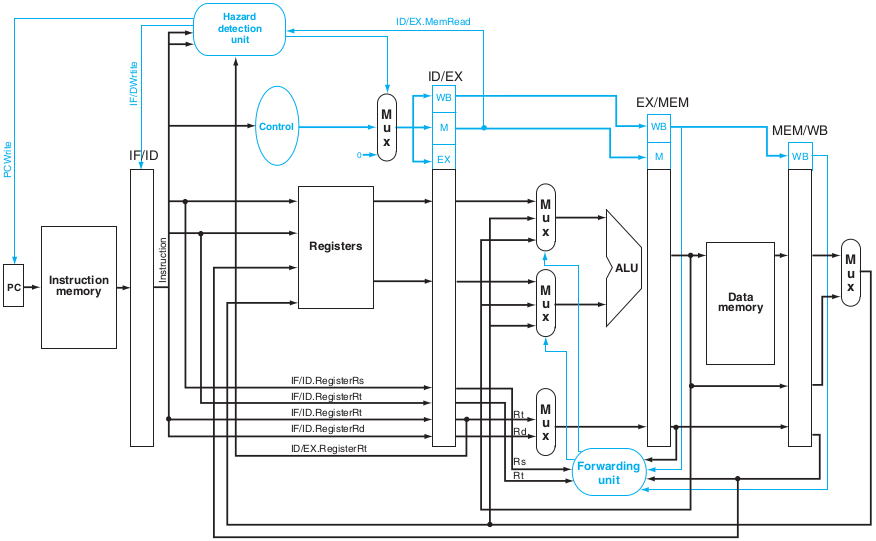
Una burbuja es un conjunto de señales que viajan sin realizar alguna operación, no escriben en registros o en memoria. Su efecto es muy similar al de una instrucción NOP, sólo que las burbujas se insertan al nivel de Hardware, retrazando a las instrucciones siguientes por un ciclo de reloj, mientras que las instrucciones NOP son parte del software.

Entonces, en la etapa ID se debe agregar un multiplexor por medio del cual la unidad de detección de riesgos pueda determinar si envía las señales de control generadas por la unidad de control o envía una burbuja (señales con valor cero).

Sin embargo, cuando se envíe una burbuja, las instrucciones que están en la etapa IF o en la etapa ID se deben detener, es decir, no deben avanzar en la segmentación. Anteriormente en estas etapas no existía alguna señal de control, puesto que no se tomaban decisiones sobre el tipo de instrucción. Ahora, es necesario agregar una señal que controle la escritura del contador del programa (PC) y del registro de segmentación IF/ID, de manera que si no se habilitan mantendrán su información actual y por lo tanto, detendrán el avance de las dos instrucciones que están en esa etapa. La habilitación de escritura de estas señales será controlada por la Unidad de Detección de Riesgos.

En la siguiente figura se muestra la implementación segmentada con Camino de datos y control,

con la unidad de detección de riesgos y la unidad de anticipación.



***Filmina 17:*** Se muestra el caso de la instrucción LW (load) en donde por más que la unidad de cortocircuito este presente hay fallas porque no se puede lograr obtener el dato desde alguna de las etapas antes que la MEM, es decir recién en la etapa MEM obtengo el dato y cortocircuitando a la próxima instrucción el tiempo en que lo requiere es anterior, como lo indica la línea roja que vuelve hacia atrás. En este caso no queda otro remedio que perder un ciclo reloj por hardware, a esta técnica se la denomina ***detención***. Se nota que por más que se tenga la unidad de cortocircuito el dato se puede obtener desde un punto (etapa MEM p 4ta etapa del pipeline) en donde la línea va hacia atrás (línea rojo) la cual indicaba que el dato se requería antes de tiempo en que se obtuviese. Con la unidad de riesgos por Detenciones (Hazard Unit) se logra resolver este tipo de riesgos, incorporando las burbujas por hardware, quiénes son las encargadas de demorar un ciclo reloj sin hacer nada siendo todo transparente al compilador.

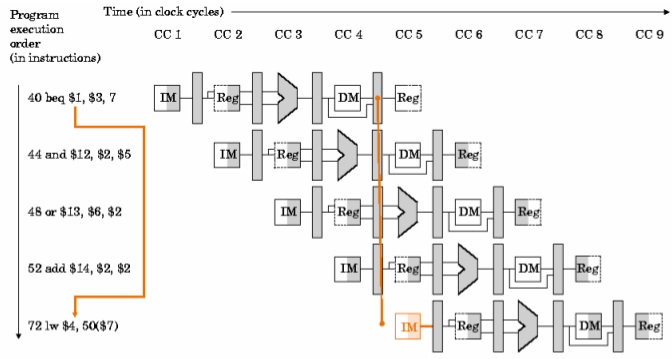
## **Riesgos de control (en instrucciones de salto)**

Existe un problema en los saltos (consideraremos a los saltos sobre igual y saltos incondicionales) porque al depender de la comparación de dos registros, si se determina que el salto no se realizará, cuando eso ocurra ya habrá otras instrucciones en la segmentación.

Si observamos la figura anterior, notaremos que la decisión de la ejecución de un salto se realiza en la etapa MEM, por que en la etapa anterior un sumador calculó la dirección destino del salto y la ALU hizo la resta de los dos registros para la posible generación de la bandera de zero. De manera que cuando la instrucción BEQ llega a la etapa MEM, ya se tienen los argumentos para determinar si el brinco se realizará.

Hasta el momento se hizo la suposición de que los saltos no se realizaban, simplemente por que los registros bajo comparación eran diferentes. De manera que las instrucciones continuaban con su ejecución normal.

El problema principal en los saltos es que en las etapas IF, ID y EX ya hay otras instrucciones, no es complicado modificar el valor del PC para continuar en la instrucción ubicada en la dirección destino del salto, la complicación se presenta en la eliminación de las tres instrucciones que ya están en la segmentación. En la próxima figura se muestra el impacto que tienen los saltos en la segmentación.



La instrucción BEQ está en la dirección 40, y al ejecutase el PC automáticamente tomará en valor de 44. La etiqueta en BEQ corresponde a la constante 7, la cual al desplazarse a la izquierda en 2 (después de la extensión en signo) toma el valor de 28, de manera que el destino del salto corresponde a la dirección 72 (44 + 28).

Para lograr resolver este problema si en la etapa ID se detecta que se trata de un salto y se determina que éste se debe realizar, sólo se reinicializaría al registro IF/ID y con ello solo se perdería un ciclo de reloj.

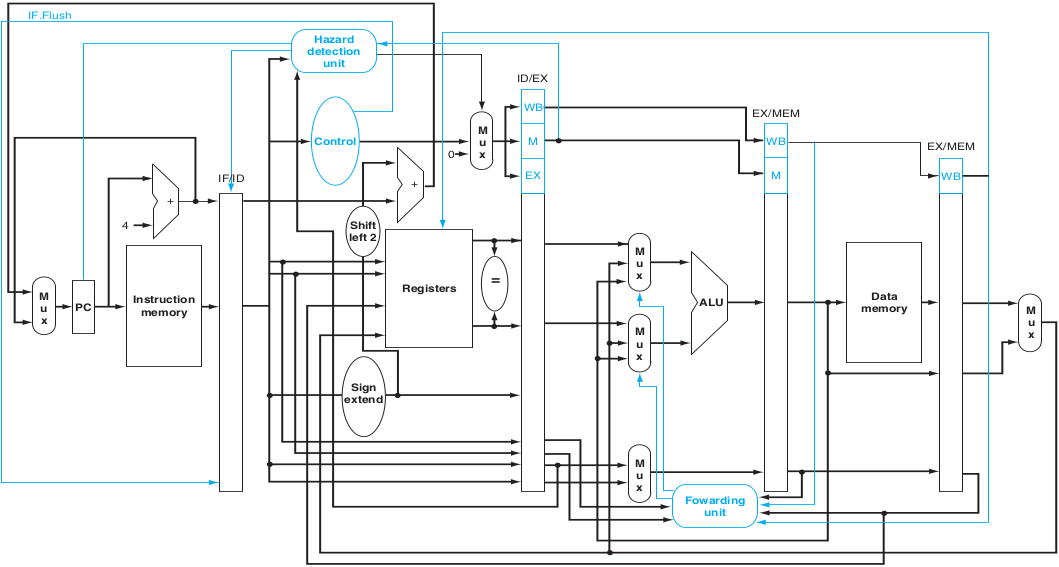
Esta idea parece congruente, pero requiere que en la etapa ID se agregue el hardware necesario para identificar a los saltos y determinar si se van a realizar. El sumador que estaba en la etapa EX, encargado de calcular la dirección destino del salto, se puede trasladar sin problema a la etapa ID, para disponer inmediatamente de esa información.

Para la comparación de los dos registros, ya no puede utilizarse a la ALU por que trasladar a la ALU a la etapa ID involucra la eliminación de una etapa y por lo tanto, aumentaría la duración del ciclo de reloj (por que dos unidades funcionales principales trabajarían en un ciclo de reloj). De manera que se requiere de un módulo rápido dedicado a comparar el contenido de los registros, este módulo puede basarse en un conjunto de compuertas XOR, que comparen bit a bit y que sus salidas se conectan por medio de una compuerta AND. El ***módulo de comparación sobre igual*** se puede ubicar en la salida de los valores que están en los registros. Con ello, tan pronto se hizo la lectura de registros, ya se puede determinar si son iguales o diferentes.

Entonces, se agregará una señal de control adicional denominada ***IF.Flush***, por medio de la cual se podrá ***limpiar el registro IF/ID***, lo que equivale a descartar la instrucción que seguía previamente al salto. En la próxima figura se muestra el hardware resultante de trasladar la evaluación de los brincos a la etapa ID. Con estos cambios, cuando un salto se va a efectuar, habrá un retraso de un ciclo de reloj. En caso de no adelantar la información de una salto dejando el sumador en la etapa 3 y dejando la determinación de la condición de un salto en la etapa 4 (MEM), es decir sin el módulo de comparación sobre igual, la solución es la misma solo que en este caso se perderían 3 ciclos reloj por cada salto condicional realizado.

Los saltos incondicionales (J), de manera similar se pueden ejecutar en la etapa ID, una vez que se detecte el opcode de una instrucción J, el valor del PC se puede sustituir por el destino del salto y además se debe limpiar el registro IF/ID para eliminar la instrucción que está en la etapa IF, para estos casos se pierde solo un ciclo reloj.

Al trasladar la ejecución de los saltos a la etapa ID, los ciclos de reloj 3 y 4 de la secuencia mostrada en la figura anterior tendrían el comportamiento en el que sólo se inserta una burbuja (como si fuera una instrucción NOP), pero ahora lo hace la unidad de control.



***Filmina 18:*** Esta filmina muestra un diagrama de instrucciones donde se produce un salto condicional, como lo dice su teoría en estos casos, si la condición se detecta recién en la 4ta etapa (MEM) y se produce el salto, las 3 instrucciones cargadas en los ciclos que se demoró en determinar la condición, deberán ser descartados por lo cual se pierden 3 ciclos reloj, ya que estas instrucciones no deberían haberse cargado. Con esta arquitectura si por estudios se sabe que entre 5 a 8 instrucciones se produce un salto podemos decir que cada 8 instrucciones estaríamos perdiendo 3 ciclos instrucción, alrededor de un 30% de instrucciones pérdidas o de demora.

***Filmina 19:*** Esta filmina explica que desplazar el cálculo del próximo PC de la etapa EX a la etapa ID acelera obtener información de la dirección inmediatamente al reconocer la instrucción de salto y además si se coloca en la dicha etapa un módulo de comparación sobre igual inmediatamente al detectar la instrucción se si debo saltar o no a la dirección de salto, de esta manera solo se perdería un solo ciclo de reloj que es el que lleva para determinar la instrucción de salto, de esta manera que se descarta una solo instrucción que se carga en el pipe hasta que se determina que hay un salto, por lo cual con estos valores las demoras se reducen a un 10% en lugar de un 30% que se tenía sin modificar la arquitectura.

***Filmina 20:***

Esta filmina explica que si se pudiera reinterpretar los saltos, en lugar de perder un ciclo esperando hasta que se detecte, sería que cambiando el orden de la ejecución de las instrucciones, en ese ciclo de espera se meta otra instrucción totalmente independiente del salt, para luego de esta si tomar el salto. De esta manera lo que se está haciendo es que en lugar de esperar un ciclo sin hacer nada se realiza otra instrucción en ese ciclo para que no se pierda tiempo esperando. En conclusión no hay ciclos de espera con esta mejora si se encuentra una instrucción para poner en el hueco (0 ciclos de penalización). Si no se encuentra una instrucción para poner en el hueco el ciclo se perderia, dando una probabilidad de un 50% que si con un 50% que no, sólo se perderian un 5% de instrucciones para los casos en los que no pueda ser posible la mejora. Para lograr esta mejora se requiere de la ayuda de dos jugadores importantes, estos son:

* El compilador.
* El propio procesador debería saber que hay un hueco de espera, para poner en ese hueco una instrucción independiente del salto. Es decir cuando se inserta una burbuja, en lugar de meter una burbuja meteriamos otra instrucción.
* La idea sería cambiar el orden de ejecución de las instrucciones para que de esta manera sea posible detectar estos casos y en los huecos de espera colocar las instrucciones, casi siempre da resultado pero hay casos en donde puede fallar por ese hay un 5% de demoras o ciclos de espera de cada el 100% de instrucciones de un programa.

***Filmina 21:*** Teniendo en cuenta todos los riesgo que debe soportar nuestro procesador por la incorporación de la segmentación cambia su ruta de datos (Datapath) tal como se observa en el diagrama.

* Por cada segmento que separa una etapa de la otra hay registros de unión (IFID, IDEX, EXMEM, MEMWB).
* Además vimos que para resolver los riesgos estructurales se tiene la incorporación de las memoria de datos e instrucción.
* Y los sumadores independientes para cálculo del próximo PC. Otro para la determinación de la dirección del salto. Todo esto libera la ALU solo para las instrucciones que la requieren.

Filmina 23: En esta filmina se plantea que las instrucciones deben tener la misma cantidad de etapas por más que algunas utilicen menos, como por ejemplo las R-type, que solo usan 4 etapas ya que no usan MEM. Pero si se diferencia por etapas voy a tener conclicto con el resto de las instrucciones.

Filmina 24: Esta filmina plantea que las instrucciones Store y Branch solo tienen activa 4 etapas ya que no escriben en el Banco de registros, es decir no utilizan la etapa 5, en este caso también se dejara que no hagan nada durante esa etapa.

***Filmina 26***: Nos muestra cual sería la cantidad de bits de control que tenemos sobre nuestro procesador segmentado. Se puede observar que todas las señales que salen de los recursos son señales que voy a manejar por cada instrucción para darle el correcto funcionamiento a los recursos en la medida que se lo requiera. Se nota también en el diagrama que los registros que hay entre etapa y etapa son cargados en cada uno de los ciclos para de esta manera no perder información a medida que se avanza por los ciclos.

También se observa cómo avanzan cada una de las señales necesarias de la instrucción a partir de los registros intermedios para que no exista pérdida de información en cada una de los ciclos que se ejecuta la instrucción. Por ejemplo vemos como IR se obtiene en la etapa 2 (IF) a partir del registro intermedio IFID, luego pasa lo mismo para el resto de señales como A, B, SX, S, S1, B1, M, PCSalto. Estas señales pasan por los ciclos es por lo cual se usa B y luego B1, porque se trata de la misma señal pero en diferentes ciclos reloj, en otras palabras se corresponden a señales iguales pero sobre diferentes etapas.

Las señales verdes muestra como avanza la dirección de escritura en el Banco de Memoria a medida que avanza la instrucción por los ciclos, cuando se llega a la etapa 5 (WB) esta dirección es utilizada en el banco de registro para guardar datos en caso de ser necesario por una instrucción, como est señal puede venir desde el registro rt o el registro rd es por lo cual hay un MUX que decide sobre la señal a través del control con RegDst, de esta forma es varia de acuerdo a la instrucción, por ejemplo sobre las instrucciones R-type esta dirección viene en rd ya que se utiliza 3 registros pero en las instrucciones I-type esta dirección viene desde el registro rt ya que solo utiliza 2 registro y un literal.

***Filmina 27:*** Esta filmina nos indica que por cada una de las instrucciones tengo un ***flujo de control*** diferente. Como este flujo de control no se utiliza directamente sino que se utiliza en las etapas que se corresponden. El flujo de control va viajando a medida que avanza la instrucción por la etapas hasta que sea el momento de su uso. Podemos observar como en la etapa 2 (ID) a la detectar la instrucción en ejecución se determina cuál es su flujo de control, este flujo de control se almacena en los registros que hay entre etapas y de esta manera a medida que avanza se va utilizando o continúa avanzando la parte que se usa más adelante.

El flujo de control son todas las señales de control para cada uno de los recurso que componen el procesador. Por ejemplo para la etapa 3 se requiere usar EX=[RegDst, ALUSrc, ALUOp] en esta etapa se utiliza y el resto de las señales de control siguen avanzando. El flujo de control se conforma por:

* EX=[RegDst, ALUSrc, ALUOp]: señales necesarias por una instrucción en la etapa EX.
* MEM=[MemRead, MemWrite, Branch]: señales necesarias por una instrucción en la etapa MEM.
* WB=[RegWrite, MemtoReg]: señales necesarias por una instrucción en la etapa WB.

Esta señales vemos que se ponen en verde cuando se corresponden a la etapa en que deben ser utilizadas. Se remarca que quizás puedan faltar señales de control pero la lógica sería la misma, si falta una señal de control de la etapa 4 (MEM) está al agregarse avanzara por los registros hasta la etapa 4.

Filmina 28: Vemos un diagrama en donde se observa cómo se utilizan las señales de control por cada una de las etapas

***Filmina 30:*** Esta filmina nos remarca cómo influye el control del procesador cuando se presentan los riesgos de datos. Nos indica:

* Cuales son los dos caminos de datos. En ambos caso se nota que desde etapas posteriores se vuelve hacia la ALU.
* Cual es la información necesaria.

Es decir, se nota que se tiene dos tipos de cortocircuito, y qué información es necesaria por cada caso.

Y con esto es con lo que se conforma la unidad de cortocircuito o anticipación.

Como se nota la información necesaria, por la unidad de cortocircuito, serían los registros del rango de la instrucción en curso (instrucción i) y los registros del dominio de las instrucciones en los instantes posteriores (instrucciones i+1, i+2, etc).

El fin de la unidad de cortocircuito es detectar el riesgo y luego corregirlo.

***Filmina 31:*** En esta filmina se puede observar cómo funciona la unidad de cortocircuito (Forwarding Unit) a través del diagrama del datapath. Se nota como la unidad controla toda la información necesaria para luego detectando los diferentes casos controlar de esta manera el correcto funcionamiento de los multiplexores que darán los operandos con los que debe actuar la ALU. Se nota que en los MUX controlados por la unidad ingresan los datos directos de las direcciones rs y rt, sino la otra posibilidad es el nuevo valor obtenido para el registro destino que viene desde el resultado de la ALU que como se registra en EXMEM vuelve la etapa EX y por último la otra posibilidad sería el mismo dato desde el registro intermedio MEMWB que sale desde el MUX WB en caso de requerir el dato en un ciclo posterior. El control para cada una de estas opciones se realiza con las señales CortoA y CortoB (vistas en el teórico como FowardA y ForwardB). Entonces se detecta si los datos de los registros rs y rt de la instrucción i con la instrucción i+1 o i+2 son iguales y en caso de serlo entonces se usa el cortocircuito, si no hay detección se deja los valores por defecto que vienen desde el registro IDEX.

***La filmina 32:*** Remarca el hecho del caso de la instrucciones load con respecto a su riesgo de datos si la instrucción siguiente tiene un mismo operando que los resultados de la anterior (caso LDE). Como vimos en esto casos no basta con la unidad de cortocircuito por lo cual se debe esperar un ciclo, agregado de burbujas. Para esto vemos cómo se modifica el datapath y como es su control en la siguiente filmina.

***Filmina 34:*** En esta filmina se nota cómo actúa la unidad de detección de riesgos (Hazard Unit). Se puede observar como esta unidad influye en los registros de la etapa IF, como asi también en los registros de la etapa ID tanto para cargar o no los registros como también para el agregado de las burbujas, mandando un cero en caso de requerir una burbuja sobre el flujo de control, esto se controla mediante un MUX. Pero además cuando se presente este tipo de riesgo si tengo un ciclo que no hace nada debo parar el contador de programa así también el registro IFID, es por lo cual estas señales de escritura de los registros PC e IFID es controlada por esta unidad (PCWirte y IFIDWrite). Además podemos notar que la unidad para detectar el riesgo en la instrucción load utiliza el registro destino del load (que viene en el registro rt) y lo compara con los registros fuentes rs y rt de la instrucción siguiente analizando si MEMRead está en uno. La condición es como la determina la teoría:

|  |
| --- |
| **if** ( ID/EX.MemRead AND  ( ID/EX.RegistroRt = IF/ID.RegistroRs OR  ID/EX.RegistroRt = IF/ID.RegistroRt ) )  **Inserta una Burbuja a la segmentación** |

Podemos notar que esto se hace cuando una instrucción i en curso en la etapa ID se detecta que su instrucción precedente (o anterior que es la instrucción LW de la filmina 32) es una load (por ejemplo LW) y es por lo cual como en este momento la instrucción precedente de carga va estar en la etapa 3 (EX), entonces se debe traer desde esta etapa a la etapa ID de la instrucción en curso, la flag MemRead y la dirección destino de la instrucción load (como vimos está el registro rt de la misma, ya que se trata de una instrucción I-type).

En conclusión con todo esto lo que sucede es que cuando la unidad detecta el riesgo, manda una burbuja, se detiene las escrituras de los registros PC y IFID, para no seguir avanzando y durante el ciclo de espera la instrucción precedente avanzó un ciclo extra mientras la instrucción en curso espero en la etapa ID.

Filmina 33: Esta filmina muestra que la ejecución de las instrucciones cambiando el orden y sin cambiar el orden de las mismas, donde el número de ciclos que le lleva la ejecución de un programa se corresponde a los diferentes casos sobre distintos compiladores. La conclusión es que cuando hay un planificador de por medio prácticamente la cantidad de ciclos de ejecución se divide a la mitad que cuando no hay planificador, es decir ordenar las instrucciones para que se ejecuten de la menor manera mejora el rendimientos de nuestro procesador segmentado.

Filmina 36-37: Estas filminas explican que para el caso de los riesgos de control que presentan las instrucciones de salto condicional/incondicional, uno de los requisitos era obtener la información lo antes posible (inmediatamente). Para esto es que desde la etapa EX se traslada el cálculo de la dirección del salto hacia la etapa ID, moviendo el sumador que calcula la dirección del salto, de esta manera inmediatamente al detectar una instrucción de salto ya se tiene la dirección y solo se perdería un solo ciclo reloj, es por lo cual se utiliza la señal de control ***IFlush*** que lo que hace es colocar las señales de control a cero y además de limpiar el registro IFID ya que pertenece a una instrucción que debe ser descartada porque se realiza un salto. Cuando se trata de un salto incondicional el salto se realiza inmediatamente como se explicó. En caso de tratarse de una instrucción de salto condicional además de determinar la dirección de salto inmediatamente, se debe determinar la condición (de si provoca o no el salto) inmediatamente, para esto, es que se se agrega el ***módulo de comparación sobre igual*** quién será el encargado de determinar si los registros son iguales o no inmediatamente en esta etapa, determinando las condiciones de las instrucciones BEQ y BNE. En conclusión sea un salto condicional o incondicional solo se perdería un ciclo reloj de demora. Recordamos que si se utiliza un compilador con planificación de instrucciones (reordenamiento de la ejecución de instrucciones) se logra reducir estas demoras a un 5% en lugar de un 10% cuando no se utiliza un planificador.

Filmina 39: Se habla que las pérdidas de rendimiento del procesador segmentado por la penalización promedio de demoras provocadas por los saltos. Suponiendo que se pierde un 30% de las instrucciones de un programa el SpeedUp sería en lugar de 5 (lo ideal) es de 3.84 (por las pérdidas del 30%).

El SpeedUp es la aceleración del procesador por la cantidad de etapas introducidas. Como se meten 5 etapas lo ideal sería que sean un total de 5 de SpeedUp pero por las pérdidas es reducido al 3.84.

Respecto del SpeedUp ideal se pierde un 24% ya que su relación da un total de 76%.

Filmina 40: Las excepciones se producirían en el pipeline del procesador.

Ejecutar una instrucción (entre comillas) significa cuando se modifican los registros (banco de registros, registro de estado, etc) y los valores de memoria. Cuando no se escribe en estos recursos se puede decir que la instrucción fue descartada o no se ejecutó.

Filmina 41: Se muestra que tipo de excepción puede producirse por cada una de las diferentes etapas del pipeline.

Se destaca que cuando se produce un fallo de página este se produce en la etapa MEM, es por lo cual se debe salvar las instrucciones hasta esas etapas el resto se pueden descartar.

Las excepciones precisas todas las instrucciones anteriores (que se cargaron primeras en el pipe) se terminan de ejecutar completamente el resto son descartada para luego atender la excepción.

Filmina 42: Esta filmina muestra como en el tiempo se van produciendo las excepciones dentro de un flujo de instrucciones sobre el pipe. Por ejemplo se nota que sobre la 2da instrucción como también la 3ra al producirse la excepción en una de sus primeras etapas, están serán atendidas antes que la que se produce en la primer instrucción ya que se produciría recién en la 4ta etapa de la misma. Este es un problema que se debería evitar ya que se tendría que resolver primero la excepción de las instrucciones que ingresan primero al pipe.

La idea es que cuando se produce una excepción lo dejo marcado en un registro (vector de estado) y luego cuando se llega a la etapa de WB será atendida ya que es el momento en el que se chequea el vector de estado. El vector de estado tiene un bit disponible por cada etapa donde es posible una excepción.

***¿Cuando se produce una excepción que es lo que yo necesito para atender la excepción?***

Me hace falta saber en donde se produjo. Y me hace falta saber donde tengo que volver luego de resolver la excepción.

***¿Cuando se produce muchas juntas que hago cual atiendo?***

Si tengo las instrucciones en orden lo importante sería atender la excepción de la primer instrucción que estaba en el pipe. El resto de instrucciones son desechadas ya que al volverse a ejecutarse, luego de que ya se haya atendido la excepción de la primer instrucción, con el mismo procedimiento serán siendo atendidas luego las excepciones de cada instrucción. Por ejemplo cuando se produce un fallo de página en una instrucción debo guardar la dirección de esa instrucción y las instrucciones siguientes a esta deberán ser descartadas, luego se resuelve la excepción asociada al fallo de página y luego se retorna a la misma dirección para continuar sobre el conjunto de instrucciones que se estaba ejecutando.

Si las instrucciones no están en orden es importante que se deba tener una tabla que nos indique cual es ese orden para de esta manera saber cual es la excepción de las instrucciones fuera de orden que se debe atender primeramente.

**Mejoras del rendimiento en riesgos de control**

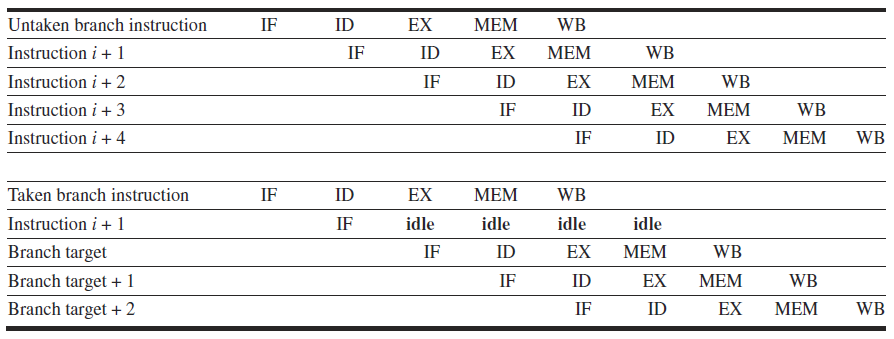
***Predicción estática***

Una solución un poco más compleja pero más eficiente implica predecir de alguna manera si el salto será tomado o no. Se tiene una “predicción de salto siempre tomado” o una “predicción de salto nunca tomado”. La complejidad de esta propuesta radica en saber determinar cuando la predicción falla y que se debe hacer para revertir los cambios.

Si se elije una predicción de salto nunca tomado se continua la búsqueda de la siguiente instrucción como si la instrucción de salto no tuviera nada en especial. Luego, si la etapa ID determina que la predicción fallo, es decir, que el salto debería haberse tomado, se debe convertir la instrucción buscada en una NOP y realizar la búsqueda de la instrucción indicada por el salto.

Por el contrario, si se predice que el salto siempre es tomado, como no se puede conocer la dirección de salto antes de la etapa ID, no se tiene ventajas y se genera la misma situación que con las detenciones del pipeline.

En este esquema, el compilador puede mejorar el rendimiento organizando el código de manera que la decisión más frecuente coincida con la decisión implementada en el hardware.



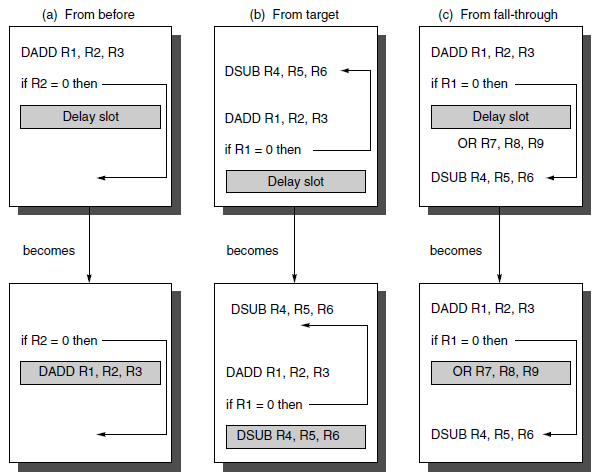
***Predicción estática, salto nunca tomado.***

***Salto retardado***

Otro esquema de manejo de saltos consiste en retardar el salto una instrucción. Esto implica que luego del salto se genera un ciclo de retardo (delay slot) consiguiendo el tiempo para determinar si el salto es tomado o no y efectuar la acción correspondiente. Este delay slot, generalmente es llenado con la instrucción siguiente al salto, por ende, la instrucción siguiente al salto se ejecuta siempre independientemente de si el salto es tomado o no.

El trabajo del compilador es buscar instrucciones válidas y útiles para rellenar estos delay slots. De no llegar a encontrar instrucciones útiles se utilizan NOP. Existen tres tipos de decisiones que el compilador puede tomar respecto a qué instrucción utilizar para rellenar el delay slot:

1. Rellenar con alguna instrucción independiente anterior al salto. Esta es la mejor opción para rellenar un delay slot.
2. Rellenar con alguna instrucción ubicada en el destino del salto. Sería como suponer salto tomado. Útil en bucles.
3. Rellenar con alguna instrucción del camino secuencia que sigue al salto. Sería como suponer que el salto no fue tomado. Útil en condiciones IF-ELSE.



***Filmina 43:*** Esta filmina explica de l mejora del rendimiento introduciendo una instrucción no dependientes del salto en la burbuja. Teniendo en cuenta el concepto de los saltos retardados.

**Apéndice A:** ***Segmentación (pipelining): conceptos básicos e intermedios***

**Ampliación de la tubería MIPS para manejar operaciones multiciclo**

Ahora queremos explorar cómo se puede extender nuestra tubería MIPS para manejar operaciones de coma flotante. Esta sección se concentra en el enfoque básico y las alternativas de diseño, cerrándose con algunas mediciones de rendimiento de una tubería de punto flotante MIPS.

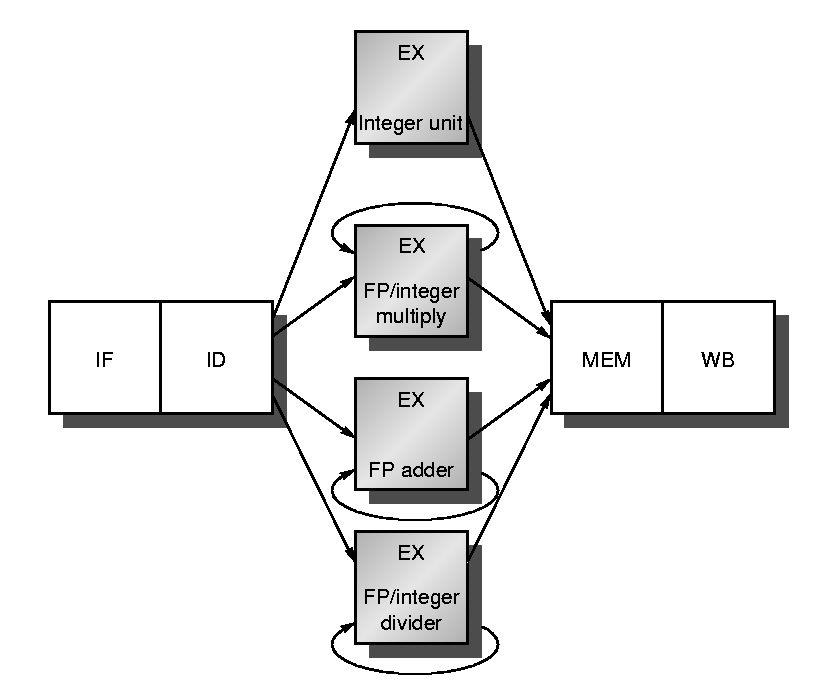
No es práctico exigir que todas las operaciones de coma flotante de MIPS se completen en un ciclo de reloj, o incluso en dos. Hacerlo significaría aceptar un reloj lento, o usar enormes cantidades de lógica en las unidades de coma flotante, o ambas cosas. En cambio, la tubería de coma flotante permitirá una latencia más larga para las operaciones. Esto es más fácil de entender si imaginamos que las instrucciones de punto flotante tienen la misma interconexión que las instrucciones enteras, con dos cambios importantes. Primero, el ciclo EX puede repetirse tantas veces como sea necesario para completar la operación; el número de repeticiones puede variar para diferentes operaciones. En segundo lugar, puede haber múltiples unidades funcionales de coma flotante. Se producirá una pérdida si la instrucción que se emitirá causará un peligro estructural para la unidad funcional que utiliza o puede causar un peligro en los datos.

Para esta sección, supongamos que hay cuatro unidades funcionales separadas en nuestra implementación de MIPS:

1. La unidad entera principal que maneja cargas y almacena, operaciones enteras de ALU y ramas.
2. FP y multiplicador de enteros.
3. Separador FP que maneja sumar, restar y convertir para FP.
4. FP y divisor entero.

Si también asumimos que las etapas de ejecución de estas unidades funcionales no están canalizadas, la Figura A.29 muestra la estructura de la tubería resultante. Como EX no está canalizado, ninguna otra instrucción que use esa unidad funcional puede emitirse hasta que la instrucción anterior deje EX. Además, si una instrucción no puede pasar a la etapa EX, toda la tubería detrás de esa instrucción se detendrá.

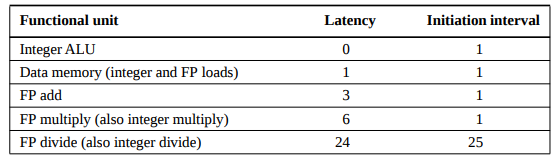
En realidad, los resultados intermedios probablemente no se ciclen alrededor de la unidad EX, como sugiere la Figura A.29; en su lugar, la etapa de la tubería EX tiene un número de retrasos de reloj mayor que 1. Podemos generalizar la estructura de la tubería FP que se muestra en la Figura A.29 para permitir la canalización de algunas etapas y múltiples operaciones continuas. Para describir tal canalización, debemos definir tanto la latencia de las unidades funcionales como también el intervalo de inicio o el intervalo de repetición. Definimos la latencia de la misma manera que la definimos anteriormente: el número de ciclos intermedios entre una instrucción que produce un resultado y una instrucción que usa el resultado.



**Figura A.29 La tubería MIPS con tres adicionantes las unidades funcionales, punto flotantes, sin segmentado. Debido a que solo una instrucción emite en cada ciclo de reloj, todas las instrucciones pasan por la tubería estándar para operaciones enteras. Las operaciones de punto flotante simplemente se repiten cuando alcanzan la etapa EX. Después de que hayan terminado la etapa EX, proceden a MEM y WB para completar la ejecución.**

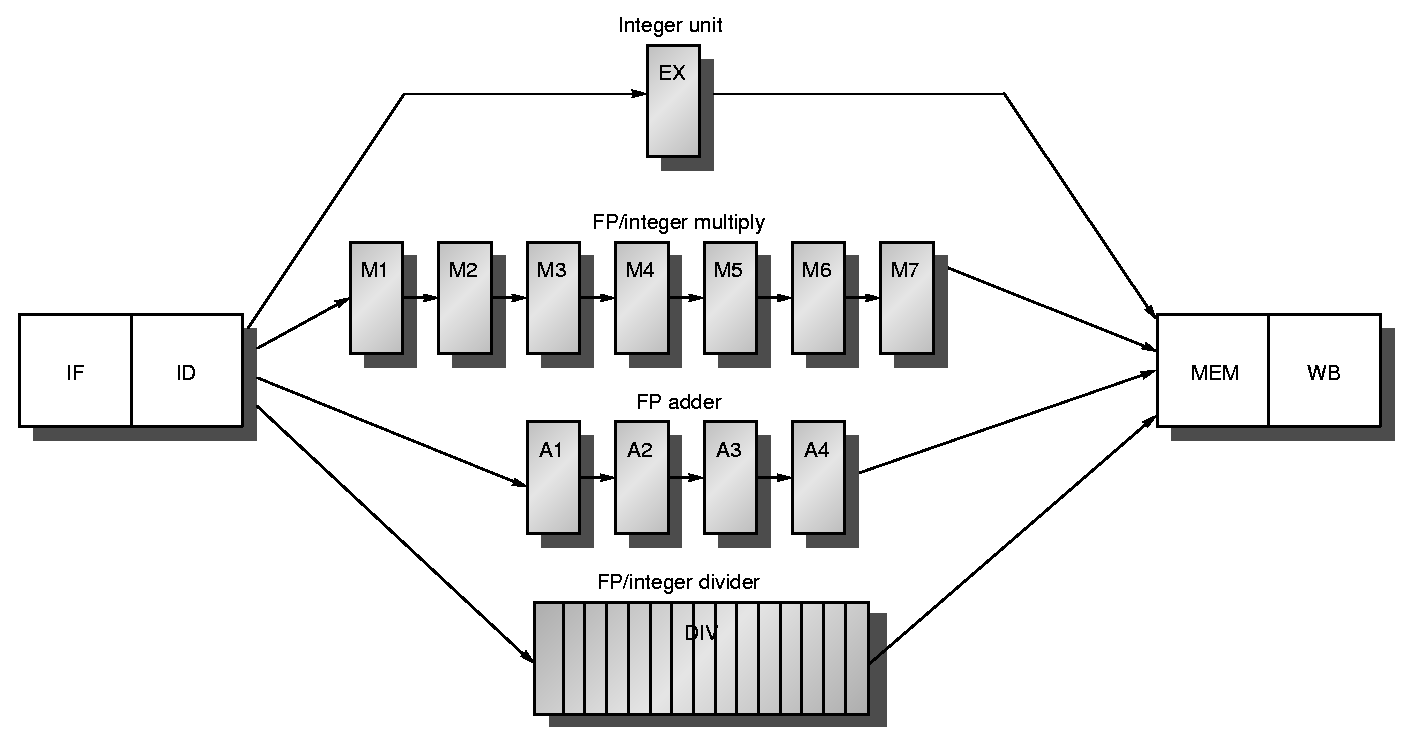
El intervalo de iniciación o repetición es la cantidad de ciclos que deben transcurrir entre la emisión de dos operaciones de un tipo determinado. Por ejemplo, usaremos las latencias y los intervalos de iniciación que se muestran en la Figura A.30.

Con esta definición de latencia, las operaciones ALU enteras tienen una latencia de 0, ya que los resultados se pueden usar en el siguiente ciclo de reloj, y las cargas tienen una latencia de 1, ya que sus resultados se pueden usar después de un ciclo intermedio. Como la mayoría de las operaciones consumen sus operandos al comienzo de EX, la latencia generalmente es el número de etapas después de EX que una instrucción produce un resultado, por ejemplo, cero etapas para las operaciones de ALU y una etapa para las cargas. La principal excepción son las stored, que consumen el valor almacenado un ciclo más tarde. Por lo tanto, la latencia de una store por el valor almacenado, pero no por el registro de dirección base, será de un ciclo menos. La latencia de la tubería es esencialmente igual a un ciclo menor que la profundidad de la tubería de ejecución, que es el número de etapas desde la etapa EX hasta la etapa que produce el resultado. Por lo tanto, para la tubería de ejemplo justo arriba, el número de etapas en un agregado FP es cuatro, mientras que el número de etapas en una multiplicación FP es de siete. Para lograr una mayor velocidad de reloj, los diseñadores necesitan poner menos niveles lógicos en cada etapa de tubería, lo que hace que el número de etapas de tubería necesarias para operaciones más complejas sea más grande. La penalización para una velocidad de reloj más rápida es, por lo tanto, una latencia más larga para las operaciones.

****

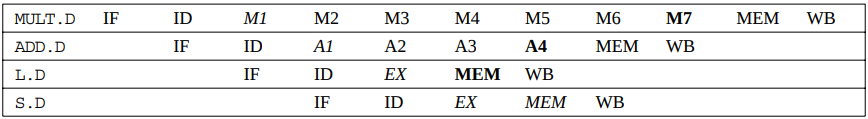
**Figura A.30 Latencias e intervalos de iniciación para unidades funcionales.**

La estructura de la tubería de ejemplo en la Figura A.30 permite hasta cuatro adiciones pendientes de FP, siete FP increíbles / multiplicaciones enteras, y una división FP. La Figura A.31 muestra cómo se puede dibujar esta tubería extendiendo la Figura A.29. El intervalo de repetición se implementa en la Figura A.31 agregando etapas de canalización adicionales, que estarán separadas por registros de canalización adicionales. Debido a que las unidades son independientes, nombramos las etapas de manera diferente. Las etapas de la tubería que toman múltiples ciclos de reloj, como la unidad de división, se subdividen para mostrar la latencia de esas etapas. Debido a que no son etapas completas, solo una operación puede estar activa.



**Figura A.31 Una canalización que admite múltiples operaciones de FP pendientes. El multiplicador FP y el sumador están totalmente canalizados y tienen una profundidad de siete y cuatro etapas, respectivamente. El divisor FP no está canalizado, pero requiere 24 ciclos de reloj para completarse. La latencia en las instrucciones entre la emisión de una operación FP y el uso del resultado de esa operación sin incurrir en un bloqueo RAW está determinada por el número de ciclos gastados en las etapas de ejecución. Por ejemplo, la cuarta instrucción después de un agregado FP puede usar el resultado del complemento FP. Para operaciones enteras de ALU, la profundidad de la canalización de ejecución es siempre una y la siguiente instrucción puede usar los resultados. Tanto las cargas FP como las cargas enteras se completan durante MEM, lo que significa que el sistema de memoria debe proporcionar 32 o 64 bits en un solo reloj.**

La estructura de la tubería también se puede mostrar utilizando los diagramas familiares de anteriores en el apéndice, como muestra la Figura A.32 para un conjunto de operaciones de FP independientes y cargas y almacenes de FP. Naturalmente, cuanto mayor es la latencia de las operaciones de FP, aumenta la frecuencia de los riesgos RAW y los puestos resultantes, como veremos más adelante en esta sección.



***Figura A.32 La sincronización de la tubería de un conjunto de operaciones de FP independientes.*** Las etapas en cursiva muestran dónde se necesitan los datos, mientras que las etapas en negrita muestran dónde está disponible un resultado. La extensión ".D" en la instrucción mnemónica indica operaciones de punto flotante de doble precisión (64 bits). Las cargas y los almacenes de FP utilizan una ruta de acceso de 64 bits a la memoria para que el tiempo de canalización sea exactamente igual a una carga o almacenamiento de entero.

La estructura de la tubería en la figura A.31 requiere la introducción de los registros adicionales de la tubería (por ejemplo, A1 / A2, A2 / A3, A3 / A4) y la modificación de las conexiones a esos registros. El registro ID / EX debe expandirse para conectar ID a EX, DIV, M1 y A1; podemos referirnos a la parte del registro asociada con una de las siguientes etapas con la notación ID / EX, ID / DIV, ID / M1 o ID / A1. El registro de tuberías entre ID y todas las demás etapas puede considerarse como registros lógicamente separados y, de hecho, puede implementarse como registros separados. Debido a que solo una operación puede estar en una etapa de tubería a la vez, la información de control puede asociarse con el registro en la cabecera del escenario.

Filmina 46: Esta filmina muestra cómo sería etapa EX para obtener en el pipe operaciones con datos de punto flotante. Se nota que existe hardware o lógica adicional en la etapa de ejecución. Donde se observa que la etapa de ejecución cuenta con cuatro unidades de ejecución independiente según sea el tipo de operación que se atienda.

Todo este agregado de lógica trae un conjunto de problemas:

* Se presentan riesgos estructurales.
* Mayor penalización de los riesgos LDE (riesgos de datos).
* Problemas con la finalización fuera de orden.
* También existe un solapamiento de operaciones enteras con las de punto flotante.
  + Pero no hay inconvenientes si utilizamos operandos independientes.

Filmina 47: Esta filmina muestra en detalle cómo sería la etapa Ex teniendo en cuenta las operaciones con punto flotante y cómo se evita los riesgos estructurales con la replicación de las unidades.

Además se observa cómo entran en juego los conceptos de  ***latencia de uso*** y el ***intervalo de inicialización***, de esta forma las instrucciones terminan fuera de orden.

**Riesgos de datos en operaciones multiciclo**

Ya habíamos visto los problemas que se pueden producir debidos a dependencias entre operaciones cercanas en el tiempo. Ahora, estos problemas se pueden complicar con las operaciones multiciclo y, además, veremos que aparecen nuevos tipos de riesgos de datos. Las dependencias entre instrucciones pueden ser de datos y de nombre.

Las ***dependencias de datos*** son las que existen entre dos instrucciones que comparten datos.

Las ***dependencias de nombre*** se producen cuando dos instrucciones utilizan el mismo registro o dirección de memoria (comparten el nombre), pero realmente no hay flujo de datos entre tales instrucciones (no comparten el dato). Hay dos tipos de dependencias de nombre: *antidependencias y dependencias de salida.*

Los distintos tipos de dependencias pueden dan lugar a otros tantos tipos de “riesgos de

datos” (RAW, WAR y WAW). La verdadera dependencia de datos (RAW) viene dada por la propia lógica del programa, mientras que las otras dos pueden producirse dependiendo de las características del cauce del procesador.

* Verdadera dependencia de datos → RAW (Read After Write)
* Dependencia de salida → WAW (Write After Write)
* Antidependencia → WAR (Write After Read)

En los cauces con operaciones multiciclo, además de que las dependencias RAW se pueden producir más fácilmente, también aparecen los otros dos tipos de dependencias.

**Riesgos RAW-LDE (Read after Write - Lectura despues de Escritura)**

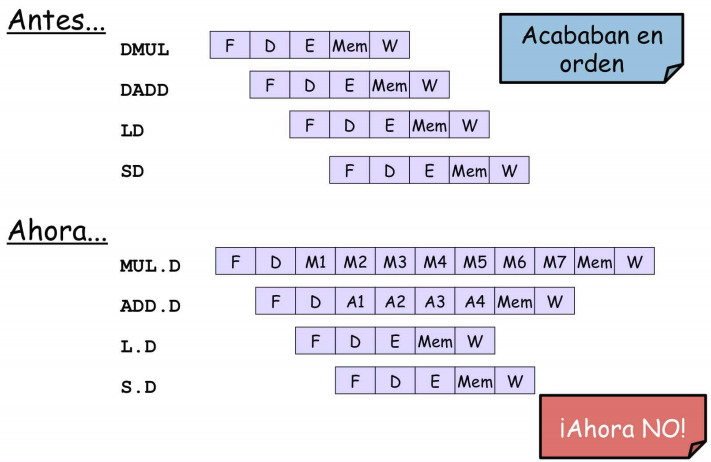


Los riesgos RAW de las dependencias entre instrucciones ya los teníamos en los cauces convencionales, pero ahora, con las operaciones multiciclo, se van a producir con mayor facilidad.

En el ejemplo de arriba, las dos operaciones de ***suma y almacenamiento de enteros*** pueden ejecutarse sin ningún problema pese a que tienen la dependencia de R1, ya que, como ya sabemos, aunque la escritura y la lectura de R1 coinciden en el mismo ciclo, la suma puede escribir el resultado en R1 en el primer subciclo, y el almacenamiento puede leer R1 en el segundo subciclo.

En el caso de que estas operaciones fueran con datos en ***coma flotante***, las cosas cambian. La etapa de ejecución de la suma ahora requiere más tiempo, por lo que es multiciclo; así que dura 3 ciclos más que con los enteros. Ahora nos encontramos que cuando la operación de almacenamiento llega a la etapa D para leer el operando R1, la suma todavía no ha llegado a la etapa W de escritura del resultado en R1.

Ante este escenario, la solución de MIPS es parar la instrucción de almacenamiento en la etapa D, hasta que el operando R1 quede actualizado en la etapa W de la suma.



En el pipeline que habíamos visto hasta hace poco, cuando solamente operábamos con enteros, varias instrucciones se ejecutaban simultáneamente en el cauce, e iban finalizando en el mismo orden en el que comenzaban.

Con la incorporación de la coma flotante y la necesidad de incluir unidades funcionales multiciclo, nos encontramos con el escenario de la parte inferior de la figura, en el que ¡las instrucciones no finalizan en el mismo orden en el que se emiten!

La multiplicación en coma flotante es una operación multiciclo muy larga, por lo que finaliza después de su sucesora, una suma, también en coma flotante, pero 2 ciclos más corta. La suma en coma flotante también es unos ciclos más larga que la carga y el almacenamiento que le siguen, por lo que, igualmente, también finaliza más tarde que éstas.

Esto va a generar unas nuevas formas de dependencias de datos que veremos en las siguientes páginas.

***Preguntas Coloquio***

***Tema 2: Procesadores Segmentados***

1. **Fil 3 – FAQs coloquio**

Explicación muy por encima de los tipos de instrucciones

1. **Fil 4 – FAQs coloquio**

Explicación general del pipe.

1. **Fil 7 – FAQs coloquio**

Explicar los círculos en rojo.

Son excepciones e interrupciones

¿Qué diferencia hay entre excepciones e interrupciones?

¿Por qué las instrucciones de tipo R hacían el WB en la quinta etapa y no en la cuarta?

Porque si hay un LW seguido de una de tipo R, se superponen las escrituras en los registros de las dos instrucciones, así que hay que poner una etapa donde no se haga nada en las instrucciones tipo R.

¿Qué es cada columna?

1. **Fil 9 – FAQs coloquio**

Explicar los 3 tipos: Monociclo, multiciclo y segmentado. Decirle que se desaprovecha tiempo en los primeros dos.

1. **Fil 12 – FAQs coloquio**

Qué tipos de riesgo había. Cuales estaba solucionando en esta filmina?

1. **Fil 13 – FAQs coloquio**

Explicar los riesgos.

1. **Fil 15-16 – FAQs coloquio**

Explicar todo lo referido a cortocircuito.

Que ocurría ahí. Cual era la solución?

1. **Fil 18 – FAQs coloquio**

Branches

Problema el Load. Solución.

1. **Fil 19 – FAQs coloquio**

Problema de los saltos. Solución.

1. **Fil 26-27 – FAQs coloquio**

Ver señal roja y verde, son señales de cada etapa.

1. **Fil 28 – FAQs coloquio**

Qué son las flechitas rojas?

1. **Fil 31 – FAQs coloquio**

¿Dónde pongo la unidad de cortocircuito? ¿Cómo funciona?

Lo importante que Mico quiere escuchar: La unidad de cortocircuito toma como entrada los operandos fuentes de la instrucción que se está por ejecutar y los operandos destino de las instrucciones anteriores, para verificar si son iguales y en caso de serlo, adelantar el resultado para no tener que esperar el WB.

Se cortocircuita si se quiere escribir, para eso son las señales de control que ingresan a la unidad.

1. **Fil 34 – FAQs coloquio**

¿Qué es la unidad de detección de riesgos? ¿Cómo funciona?

Lo importante que Mico quiere escuchar: La unidad de detección de riesgos toma las mismas entradas que las de cortocircuito, pero evita que se cargue la instrucción en caso de haber un riesgo.

Las señales de control dependen de la instrucción completa, y se utilizan para controlar los multiplexores y elegir la forma en que se controlan.

Hay 3 bloques de control, uno para cada etapa (EX, MEM y WB) y los bloques de control se van cortocircuitando a las siguientes etapas si no se utilizan.

Que hace la unidad de control

Que hace la unidad de cortocircuito

Que hace la unidad de detección de riesgos.

1. **Fil 43 – FAQs coloquio**

¿Qué se hace acá?

Explicar lo del delay slot del branch.

¿Por qué surgía?

Si bien se adelanta el cálculo de la dirección de destino y el cálculo del resultado del salto, lo mismo se pierde un ciclo que habría que esperar, entonces se pone una instrucción independiente del salto que se va a ejecutar siempre, y se retarda un ciclo el salto del programa.

Explicar los delays. ¿Quién los rellena?

Hay que modificar el procesador para que deje terminar la instrucción que sigue.

Quien se encarga de hacer esto?

El compilador pero el microprocesador debe modificarse.

1. **Fil 47 – FAQs coloquio**

¿Qué son las unidades de ejecución?

¿Por qué la de división es más grande?

Porque no está segmentada, por eso el intervalo de iniciación.

¿Qué entendés por las 2 columnas de la tabla?

Ver que por ejemplo en la división PF en usarla de nuevo demora los 24 ciclos, no así las otras unidades funcionales.