Unidad 4 y 5 - Microarquitectura Segmentada y Riesgos

1. Implementación Segmentada

Antes vimos la implementación **monociclo**, la más simple en cuanto a diseño e implementación circuital. El problema de esta es que todas las instrucciones se ejecutan en un sólo ciclo de reloj, por lo que el periodo del reloj debe ser igual al tiempo de ejecución de **la instrucción más lenta**, despediciando tiempo en las instrucciones más rápidas.

Las instrucciones más lentas son las de carga y almacenamiento de datos, que involucran todas las etapas del ciclo de instrucción (fetch, decode, op fetch, execute, write-back), además del acceso a la memoria. Por el otro lado, las más rápidas son las instrucciones de salto y operación con datos en registros, ya que solo involucran la decodificación y ejecución.

En la **implementación segmentada** se divide la ejecución de la instrucciones en etapas más pequeñas a nivel circuital, tal que cada instrucción utilice solo las etapas necesarias y el período del reloj se determine por la **etapa** más lenta. En principio las etapas en las que se dividirá la ejecución serán Fetch, Decode, Operand Fetch, Execute y Write-Back, pero hay implementaciones que usan más o menos divisiones.
De esta forma, una instrucción tipo R o tipo B (Fetch, Decode, Execute) tardará tres ciclos de reloj, una tipo S (Fetch, Decode, Execute, Data Memory, Write-Back) tardará 5 ciclos.

Cada bloque del procesador segmentado se "desocupa" al ejecutar una etapa de la instrucción, por lo que puede utilizarse para iniciar la ejecución de la próxima instrucción. De esta forma, aprovechamos los recursos del procesador para paralelizar la ejecución de las instrucciones.

A nivel circuito, la segmentación requiere elementos para almacenar los valores generados por los diferentes bloques

1.1. Supersegmentación

Si quisieramos calcular el tiempo necesario para ejecutar una instrucción, podríamos expresarlo como:

$$t = t_{fetch} + t_{decode} + t_{op\_fetch} + t_{execute} + t_{write\_back}$$

En un caso ideal en el que todas las etapas requieren el mismo tiempo para ejecutarse (microarquitectura balanceada), el tiempo necesario para ejecutar una etapa sería igual a `t/n`, donde `n` es el número de etapas en las que se divide el ciclo de instrucción. En este caso, la frecuencia de reloj entonces sería `f = 1/(t/n) = n/t`, por lo que podríamos decir que cuanto más segmentemos el procesador, mayor potencia tendrá, TEÓRICAMENTE.

A la segmentación del procesador en muchas etapas se le llama **supersegmentación**, y puede llevar al deterioro del desempeño global, ya que

- El pipeline requiere más tiempo para llenarse (que todas las etapas estén "ocupadas").
- Se requiere más espacio para alojar la circuitería necesaria.

2. Riesgos

Los riesgos son situaciones que imposibilitan la ejecución de la próxima instrucción en el próximo ciclo de reloj. Se clasifican en:

- **Riegos de Datos**: hay que esperar que se termine de ejecutar una instrucción anterior para disponer del dato necesario para la próxima actual.
- **Riesgos de Control**: una acción de control de flujo de programa (salto condicional, por ejemplo) depende de una instrucción previa.
- **Riegos Estructurales**: un recurso requerido para la ejecución se encuentra ocupado.

2.1. Riesgos de Datos

Ocurren porque una instrucción refiere a un resultado que aún no se calculó o no fue recuperado.

```
sub  x2, x1, x3
and  x12, x2, x5    // necesita el resultado de la instrucción anterior
or   x13, x6, x2    // necesita el resultado de la primer instrucción
add  x14, x2, x2    // necesita el resultado de la primer instrucción
sd x15, 100(x2)     // necesita el resultado de la primer instrucción
```

Las instrucciones muestran una dependencia de los datos en distintas etapas del datapath.

Las situaciones en las que puede ocurrir un riesgo de datos son:

- Lectura despues de Escritura (Read-After-Write): una instrucción refiere a un resultado que no se ha caculado o recuperado. Aunque una instrucción se ejecute después de la anterior, al momento de recuperar el operando puede que la instrucción anterior solo se haya procesado parcialmente en el datapath (**dependencias de datos**).
- Escritura despues de Lectura (Write-After-Read): una instrucción escribe antes de que una instrucción anterior la lea (**dependencias de nombre**). Como te podrás imaginar, es dificil que suceda esta situación ya que la etapa de op-fetch es anterior a la de write-back, pero existen arquitecturas en las que puede suceder.
- Escritura después de Escritura (Write-After-Write): dos instrucciones sucesivas escriben el mismo registro (**reutilización de registros**).

Existen diferentes estrategias para resolver estas situaciones.

2.1.1. Adelanto de Datos

Una estrategia para salvar los riesgos de *lectura despues de escritura* es la llamada **adelanto de datos**.

El dato que quiere leer una instrucción próxima puede estar disponible antes de la etapa de escritura en registro. Puede agregarse una **unidad de adelanto** en la etapa de ejecución (a la salida de la ALU), que se encargará de enviar los datos de la salida de esta etapa directamente a la instrucción próxima en los casos que lo necesite.

2.1.2. Detención

Los riesgos generados por una *escritura después de lectura* no se puede salvar adelantando datos porque el dato simplemente no está disponible. En estos casos el riesgo de elimina deteniendo la etapa que requiere escribir, para que la instrucción anterior pueda leer el dato.

Existen dos tipos de detenciones:

- Burbujas: se inserta una instrucción `nop` entre las dos instrucciones en el datapath. De esta forma todas las instrucciones anteriores no progresan en el datapath durante un ciclo, pero las instrucciones posteriores progresan normalmente.
- Vaciado: en este caso todas las instrucciones del datapath se sustituyen por `nop`.

2.2. Riesgos de Control

Suceden al alterarse el orden de ejecución del programa, el orden en el que deben ejecutarse las instrucciones. Lo importantes a la hora de manejar estos riesgo es **preservar el resultado del programa**, por lo que las dependencias de control (el orden del programa) pueden violarse bajo esta condición. Puede pensarse como que desde el punto de vista del programador el programa debe comportarse como si fuera monociclo.
Además de esto, debe prestarse atención a

- Dependencias de Control y Excepciones: un cambio en el orden de ejecución no debe cambiar cómo son atendidas las excepciones.
- Dependencias de Control y Flujo de Datos: un cambio en el orden de ejecución no debe cambiar el flujo de datos entre instrucciones productoras y consumidoras de datos.

2.2.1. Flushing

Suponiendo que se de un salto condicional, todas las instrucciones "salteadas" que ya esten dentro del pipeline deberían ser borradas (reset del datapath), a esto se le llama flushing.

2.2.2. Saltos

Cuando una instrucción de salto es detectadas, se determina el tipo de salto, la información necesaria para ejecutarlo y la dirección del salto.
Los tipos de saltos que nos podemos encontrar son

- **Salto Incondicional**: como podemos imaginar, un salto incondicional se va tomar **siempre**, tal es el caso de una instrucción `j` o `jal`, por ejemplo. En estos saltos se debe determinar la fuente de información necesaria para calcular la dirección final del salto. Existen dos casos de estos saltos
  - Salto absoluto: la dirección final se encuentra en la instrucción misma.
  - Salto relativo: la dirección final se obtiene componiendo parte de la instrucción con la información de algún registro.
- **Salto Condicional**: cuando se detecta un salto condicional, primero se determina si el salto se ejecuta o no para luego, si se ejecuta, determinar la fuente de información necesaria para calcular la dirección final del salto.
  - Se predice como tomado: se calcula la dirección destino y la ejecución continua de forma **especulativa** desde dicha dirección.
  - Se predice como no tomado: la ejecución continua en el curso normal de forma especulativa.

Además, en los casos distintos casos luego de predecirse se resuelve de distintas formas:

- Predicción correcta: se confirma la ejecución y continua normalmente.
- Predicción incorrecta: se descartan las instrucciones ejecutadas **especulativamente** (flushing) y se reanuda por el camino correcto.

El problema es que no se conoce el tipo de instrucción hasta finalizar la etapa de decodificación (decode), por lo que existe un retardo en la detección del salto. Para esto existen tres alternativas:

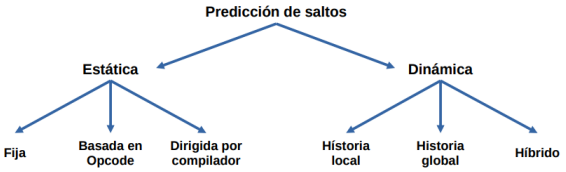
- Detección de salto paralelo a Decodificación: se introduce un decodificador dedicado a detectar saltos antes del final de la etapa de decodificación.
- Detección despues de Búsqueda: se detectan las instrucciones de salto en el buffer de instrucciones antes de ser decodificadas.
- Detección durante la Búsqueda: se detecta el salto al tiempo que se leen la memoria de instrucciones.

Una primera aproximación a resolver este problema consiste en ejecutar en la etapa de búsqueda de instrucción (fetch) los saltos con información completa, mientras que ejecutamos en la etapa de búsqueda de datos (operand fetch) aquellos saltos con información incompleta y permitimos vaciar el datapath en caso de tomar el salto.

Para esto se introduce un **buffer de destino de saltos** (Branch Target Buffer - BTB), que almacena las direcciones en las que ha habido un salto anteriormente y así disminuir al mínimo el tiempo que se requiere al obtener la dirección destino, sobretodo, como dijimos, en los casos donde la dirección se debe calcular.

Es importante no confundir la BTB con la BHT (mas adelante).

2.2.3. Predicción de Saltos



Clasificacion de técnicas de predicción

Como ya dijimos, cuando la información de la dirección final no se encuentra completa en la instrucción, la ejecución de la instrucción de salto se retrasa al menos dos ciclos. Para superar esto, se introducen unidades de predicción de saltos que predigan (duh) si es más probable tomar o no el salto. En caso de fallar la predicción, se pierde tanto tiempo como números de etapas entre la etapa de búsqueda y la ejecución. En los procesadores modernos el pipeline es bastante largo, por lo que esta pérdida es considerable y aumenta la necesidad de predictores más sofisticados.
Los algorítmos de predicción funcionan porque los algorítmos y los datos que utilizan presentan regularidades que éstos pueden detectar. Además de estos, pueden existir redundancias fruto de la forma de pensar de los humanos.

2.2.3.1. Predicción Estática

Con esta técnica sencilla evitamos almacenar un historial de ejecución del código (como ya veremos en la predicción dinámica), prediciendo el resultado basándonos únicamente en la instrucción. Para cada tipo de predicción estableceremos reglas fijas para indicar cuándo se toma el salto y cuándo no:

- **Predicción Fija**: es la más simple de todas, solo establece si se toma SIEMPRE o NUNCA. Puede establecerse también según la dirección relativa, es decir, tomar el salto siempre que sea "hacia atrás", muy tipico en bucles.
- **Predicción Basada en OpCode**: se establece si tomar o no el salto dependiendo de la instrucción específica. Podríamos tomarlo siempre en las instrucciones `j` y no tomarlo nunca en las instrucciones `beqz`, por ejemplo.
- **Predicción Dirigida por Compilador**: en esta técnica el compilador analiza el código y genera información con "sugerencias" de predicción basadas en este análisis.

2.2.3.2. Predicción Dinámica

Un predictor dinámico trabaja en tiempo de ejecución intentando "aprender" el comportamiento del programa para predecir, con la mayor tasa de acierto, si un salto será o no tomado.
Un predictor de saltos suele manejarse llevando un historial de los saltos tomados, junto con las direcciones.

- **Branch History Table** (BHT): guarda la información de las etapas de Fetch, Decode y Execute junto con el resultado. Esta tabla es direccionada por los bits más bajos del contador de programa. Es una tabla de doble puerto que
  - **Lee**, durante la etapa de Fetch, si hay registro de un salto en la dirección de salto de programa actual, y en tal caso recupera la dirección destino asociada (de la BTB, por ejemplo).
  - **Actualiza**, durante la etapa de Decode, la dirección asociada según si la predicción fue correcta o no.
  - **Escribe** una entrada en la tabla durante la etapa de Execute, en caso de encontrarse con un salto sin entradas en la tabla.

Como se puede ver, es lógico combinar la BHT con la BTB. La tabla de historia (BHT) contiene información sobre el comportamiento de los saltos, mientras que la tabla de destino (BTB) contiene información sobre el origen y destino de los mismos. La ventaja esta implementación es que podemos predecir salto que no se encuentren en la BTB, aunque se necesita más hardware para su implementación.