

Organización del Computador II

Gianfranco Zamboni

28 de junio de 2019

Índice

1. Introducción	2
1.1. Componentes del ISA	2
1.1.1. Registros	2
1.1.2. Arquitectura de Von Neumann	4
1.2. Microarquitectura	4
2. Pipelining	6
2.1. Pipeline Hazards	7
3. Branch Prediction	8
3.1. Predicciones estáticas	8
3.2. Predicciones dinámicas	8
4. Instruction Level Parallelism	10
4.1. Dependencias de instrucciones	10
4.2. Fetching and Decode	11
4.3. Ejecución de instrucciones	11
4.3.1. Interrupciones	11
4.3.2. In-Order Instruction Completion	12
4.3.3. Reorder-Buffer	13
4.3.4. Out of Order Execution (Tomasulo)	14

1. Introducción

La **arquitectura de computadoras** es la ciencia y arte de diseñar las interfaces hardware/software para crear un sistema computacional que posea los requerimientos funcionales, de performance, consumo (de energía) y de costo (económico) adecuados para realizar determinadas tareas.

Estas tareas son problemas que pasaron por varias transformaciones (desde su descripción en un lenguaje natural hasta convertirse en un programa) y deben ser ejecutadas por una computadora. La tarea del arquitecto consiste en diseñar el **Instruction Set Architecture (ISA)**, un conjunto de instrucciones que usarán los programas compilados para decir al micropocesor que hacer. El ISA es implementado por un conjunto de estructuras de hardware conocidas como **microarquitectura**.

El ISA y la microarquitectura sientan las bases para que el diseño del procesador que conseguir el balance adecuado de los factores mencionados para llevar a cabo ciertas tareas de la manera más óptima posible. Habrá casos en los que daremos prioridad a un subconjunto de ellos en detrimento de otros (por ejemplo, podríamos elegir mejorar performance y aumentar el costo, o quitar performance para mejorar el consumo energético).

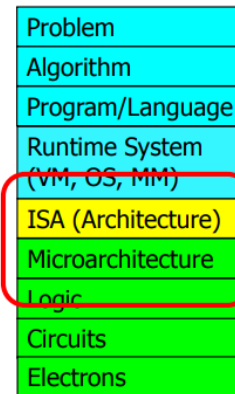


Figura 1: figure

La computadora definida en niveles de abstracción

1.1. Componentes del ISA

El ISA es la especificación completa de la interfaz entre los programas y el hardware que debe llevar a cabo las operaciones. Entre otras cosas, especifica:

1.1.1. Registros

- **Registros:** Celdas de memoria dentro del cpu que son usados para almacenar los datos necesarios para ejecutar una instrucción. Éstos son visibles al programa y se clasifican según su uso: Acumuladores, De dirección ó De Proposito General.
- **Instrucciones:** Tareas que pueden ser llevadas a cabo por la computadora. Cada una de ellas está compuesta por su **opcode** (que se espera que la computadora haga) y sus **operandos** (a que datos debe hacerlo). En una ISA, podremos encontrar tres tipos de instrucciones:

- De **Operacion**: Procesan datos.
- De **transporte**: Transportan información entre la memoria, los registros y los dispositivos de entrada salida.
- De **control (Branching)**: Modifican la secuencia de instrucciones a ser ejecutada, es decir, permiten ejecutar instrucciones que no están almacenadas secuencialmente.

Dependiendo que valores puedan modificar las instrucciones de operación, podremos clasificar las arquitecturas en: **Arquitecturas Load/Store** (solo pueden operar en registros) o **Arquitecturas memory/memory** (se pueden modificar los valores directamente en memoria)

- **Tipos de datos**: Representación deben tener ciertos valores para que puedan ser interpretados por la microarquitectura.
- **Espacio de memoria**: La cantidad de bloques univocamente distinguibles en memoria y el tamaño de cada uno de estos bloques
- **Direccionamiento**: Los mecanismos usados por la computadora para saber donde están almacenados los datos. Puede ser:
 - **Inmediato**: El operando está incluido en la instrucción.
 - **Directo o absoluto**: El operando es la dirección de memoria donde se encuentra el valor a ser utilizado.
 - **Indirecto**: El operando es una dirección de memoria donde se encuentra la dirección de memoria en la que está almacenado el valor deseado.
 - **De desplazamiento**: La instrucción toma como operandos una dirección de memoria que se toma como **base** y un **offset**, que es un número que indica cuanto hay que desplazar la base para encontrar el valor deseado, es decir $dir = base + offset$
 - **Indexado**: Lo mismo que el anterior, pero con el *offset* guardado en un registro.
 - **De Memoria Indirecta**: El operando es un registro en el que se encuentra guardada la dirección de memoria indirecta.
- **I/O Interface**: Como comunicarse con los dispositivos de entrada/salida. Puede ser por medio de instrucciones especiales o mapeos de ciertas regiones memoria para uso de esos dispositivos.
- **Modos de privilegio**: Quien puede y quien no puede ejecutar ciertas instrucciones
- **Manejo de expcepciones e interrupciones**: Qué debe suceder si una instrucción falla o cuando un dispositivo necesita usar el microprocesador.
- **Soporte de memoria virtual**: Si soporta o no el uso de **memoria virtual**, es decir, si cada programa tiene la ilusión de estar un espacio de memoria secuencial cuando en realidad el sistema operativo realiza el manejo de la memoria principal.

1.1.2. Arquitectura de Von Neumann

Como se vió en Organización del computador I, las mayoría de las ISA usadas actualmente usan el modelo de Von Neumann. Éste es un ciclo de cuatro etapas:

1. **Fetch:** Se utiliza un **program counter** para saber donde está almacenada la proxima instrucción a ser ejecutada. Y se carga desde la memoria.
2. **Decode:** Se decodifica la instrucción fetcheada y se consiguen los operandos (literales y registros) correspondientes.
3. **Execute:** En esta etapa se busca en memoria los datos requeridos (si es necesario) y se procesa los datos acorde a la instrucción.
4. **Write Back:** Se almacenan los resultados obtenidos en el lugar indicado.

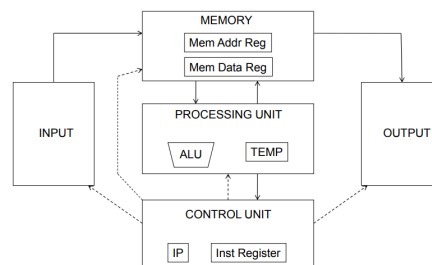


Figura 2: Arquitectura de Von Newmman

Cada instrucción es extraída de la memoria usando la dirección indicada por el **Instruction Pointer**. La unidad de control se encarga de indicar a la memoria si son necesarios otros valores para poder llevar a cabo su ejecución y luego pasa todo los datos a la unidad de procesamiento.

Si bien este es el ciclo que “ve” un programador, la implementación de las ISA (microarquitectura) usa estructuras de hardware más complejas que permiten acelerar la ejecución de cada fase.

1.2. Microarquitectura

La micoarquitectura es la implementación a nivel hardware de la ISA, es decir, es un conjunto de componentes electrónicos organizados de cierta manera para que respeten esas especificaciones. En la sección 1.1.2, vimos como el usuario ve el ciclo de instrucciones. Desde el punto de vista de la implementación (hardware), el ciclo es realizado por unidades de procesamiento que operan sobre los datos de acuerdo a ciertas señales.

Cada instrucción es una señal que usa el procesador de instrucciones para decidir que conjunto de componentes electrónicos deben ser activados para poder llevar a cabo la operación deseada. Específicamente, las instrucciones indican:

- **Datapath:** Que elementos deben manejar y transformar los datos (unidades de procesamiento, de almacenamiento y estructuras de hardware que permiten el flujo de datos)
- **Control Logic:** Que elementos de hardware determinan las señales de control que indican al datapath lo que debe hacer con los datos.

En otras palabras, la microarquitectura comprende la tecnología utilizada para construir el hardware, la organización e interconexión de la memoria, el diseño de los bloques de CPU y la implementación de distintos mecanismos de procesamiento que no son visibles para el programador. Algunas de las características encontradas en las microarquitecturas actuales son:

- Pipelining
- Ejecución de múltiples instrucciones simultáneamente.
- Ejecución fuera de orden
- y Cachés de dato e instrucciones separadas, entre otros.

2. Pipelining

El Pipeline es una técnica que permite superponer el procesamiento de múltiples instrucciones en una ejecución. En vez de procesar una instrucción completamente (realizar todas las etapas del procesamiento: Fetch, Decode, Execute, Access, Write) y luego otra, una vez que la primera instrucción fue fetcheada y pasa a la etapa de decodificación, se comienza a fetchear la siguiente instrucción.

Bajo condiciones ideales y con un gran número de instrucciones, la mejora en velocidad de ejecución es directamente proporcional a la cantidad de etapas en el pipe. Es decir, un pipeline de 5 etapas, es aproximadamente 5 veces más rápido que el procesamiento secuencial. Notemos que el pipelining no modifica el tiempo que se tarda en procesar una instrucción (**latency** / **latencia**), sino que mejora el performance aumentando la cantidad de instrucciones que se procesan por unidad de tiempo (**throughput**).

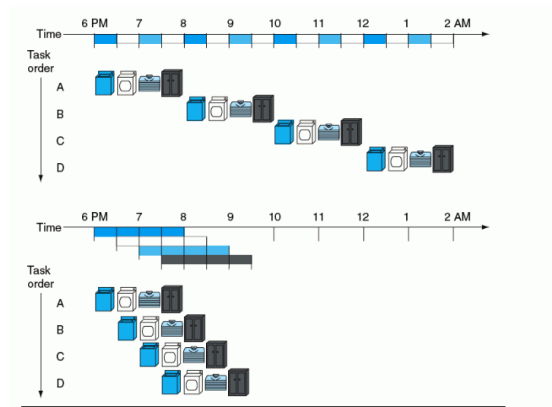


Figura 3: Analogía de la lavandería: 4 personas tienen que lavar, secar, doblar y guardar su ropa sucia. Solo disponemos de una lavadora, una secadora, un “doblador” y un ropero. Si cada parte del proceso toma 30 minutos, y lo realizamos de manera secuencial, entonces completarlo para las 4 personas tomaría ocho horas, mientras que si lo hacemos con un pipeline el tiempo se reduce a tres horas y media.

Para facilitar una implementación efectiva del pipeline, el set de instrucciones propuesto debe permitir que todas las etapas tarden el mismo tiempo en ser ejecutadas. Esto implica que hay que tener ciertas consideraciones al momento de diseñarlo:

- Todas las instrucciones deben tener la misma longitud (en la arquitectura IA-32, donde las instrucciones tienen longitud variada, esto se logra descomponiendo cada instrucción en micro-operaciones de longitud fija y a estas microoperaciones se les aplica el pipelining).
- Todas las instrucciones deben tener la misma estructura, es decir, la cantidad de parámetros y la ubicación dentro de la instrucción que los explicita deben ser las mismas (o lo más parecidas posibles).

- Los operandos deben estar alineados en memoria, esto es, las direcciones de memoria que ocupan deben ser múltiplo del tamaño de las palabras usadas para que los datos puedan ser transferidos en una única etapa del pipeline.

2.1. Pipeline Hazards

Hay situaciones en las que una instrucción no puede ser procesada en el siguiente ciclo de reloj. Estos eventos pueden llegar a detener el flujo del pipeline y generar una demora (**hazards**) en la ejecución del programa. A continuación veremos los tres tipos de obstáculos que se pueden dar:

- **Estructurales (Structural Hazards):** El hardware no soporta la combinación de instrucciones que queremos ejecutar en el mismo ciclo de reloj. Por ejemplo, si una instrucción debe acceder a memoria mientras otra debe realizar un fetch en la misma memoria. En este caso, las dos instrucciones deben utilizar los mismos recursos y debería darsele prioridad a la primera instrucción dejando a la segunda en espera.
- **De datos (Data Hazards):** Hay una instrucción en el pipeline que depende de los resultados de otra (también en el pipeline) y debe esperar a que ésta se complete para poder terminar. Esto puede bloquear el pipe durante varios ciclos de reloj ya que se debe procesar completamente la primera instrucción. Para minimizar el impacto de este obstáculo, por lo general, se agrega extra hardware que permite conseguir el valor deseado apenas sea calculado (cuando termina la etapa de ejecución) directamente de los componentes internos para no tener que esperar a que sea guardado en memoria (esta técnica se llama **forwarding** o **bypassing**).
- **De control (Control or Branch Hazards):** La instrucción que se fetcheo no es la instrucción que debe ejecutarse en este ciclo de reloj. Esto sucede cuando una de las instrucciones del pipe es una instrucción condicional. En estos casos, cuando la instrucción es fetchada, el pipeline no puede saber cuál es la próxima instrucción que debe ser ejecutada ya que esto depende del resultado de la instrucción actual. Una posible solución es parar apenas se haga el fetch del condicional y esperar hasta obtener sus resultado. Otra, es realizar predicciones (**branch prediction**) y ejecutar las instrucciones con más probabilidad de ser ejecutadas. En este último caso, el pipeline procede sin demoras si la predicción fue correcta, sino debe eliminar las instrucciones procesadas y volver a comenzar desde el lugar correcto.

3. Branch Prediction

Branching: Cuando se fetchea una instrucción condicional, se genera una ramificación en el código. Hay dos caminos posibles por el que puede seguir la ejecución y no tenemos seguro cual de los dos tomar.

Untaken Branch: Se dice cuando el camino tomado es el que ejecuta la instrucción siguiente a la del condicional dentro del programa.

Taken Branch: Cuando se ejecuta la instrucción que se encuentra en la dirección del salto de la instrucción condicional.

En la sección anterior se presentó la técnica de pipelining y los posibles problemas que se pueden llegar a tener cuando se usa. Uno de estos eran los problemas de control que bloqueaban el pipe cuando una de las instrucciones fetcheadas era de control. Hay diversos tipos de predicción que intentan disminuir el impacto de este problema, algunas de ellas son:

3.1. Predicciones estáticas

- **Assume Branch Not Taken:** Se fetchea la siguiente instrucción secuencial del programa. Si el salto no se realiza, entonces la ejecución del pipe continúa sin problemas. Si el salto se realiza, entonces se deberán descartar las instrucciones fetcheadas y decodificadas hasta el momento para poder retomar en el lugar correspondiente.
- **Assume Branch Taken:** Análogo al anterior, pero esta vez, se fetchea la instrucción en la dirección de memoria apuntada por el salto.
- **Predict by Opcode:** Se asume que el salto va a ser tomado o no dependiendo de la instrucción a ser ejecutada.

Este tipo de predicciones funcionan bien para pipelines simples. Sin embargo, la penalidad de descartar instrucciones y el tiempo que se tarda en restaurar el sistema aumenta acorde a la complejidad del mismo. Las predicciones dinámicas intentan disminuir este problema manteniendo estadísticas de uso y modificando las decisiones tomadas a medida que se realiza la ejecución.

3.2. Predicciones dinámicas

- **Branch Prediction Buffer:** Se mantiene una tabla indexada por la dirección de memoria de la instrucción del salto y 2 bits que indican si los últimos dos saltos hacia esa instrucción fueron tomados o no.

Las primeras implementaciones de esta técnica hacían uso de un único bit que se remplazaba cada vez que la predicción fallaba. En ciertos casos, la eficiencia de este método no era satisfactoria llegando a fallar completamente en otros (por ejemplo, si los saltos termina formando una secuencia intercalada de Taken y Not taken).

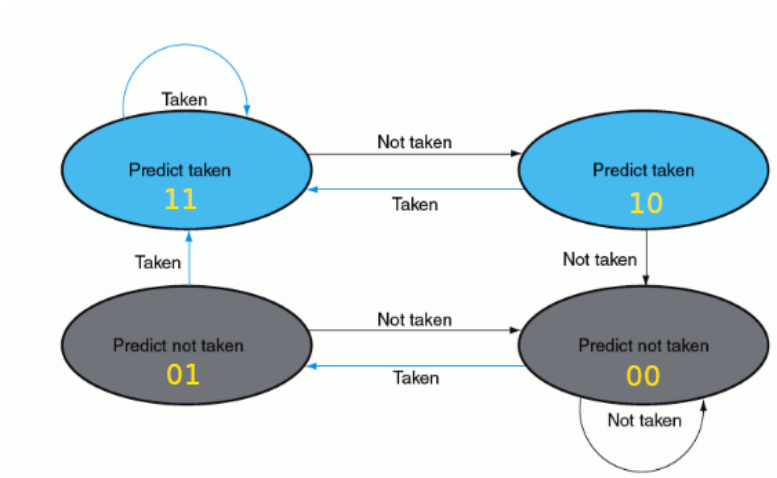


Figura 4: Maquinas de estados de predicción para un buffer de 2 bits

- **Branch Target Buffer (BTB):** Es una caché en la que se almacena en cada entrada la dirección de la instrucción de control y la dirección de la instrucción que fue ejecutada después de resolver el branching.

Cuando se fetchea una instrucción de control, se accede a esta caché usando el valor del program counter.

- Si el valor no está en el buffer, entonces se asume *taken*.
 - Si el resultado es *Non-Taken* se acepta el delay en el pipeline y no se almacena nada en el BTB.
 - Si el resultado es *taken*, se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección de target almacenado.
 - si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
 - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB, luego de la penalidad correspondiente en el pipeline.

4. Instruction Level Parallelism

Hoy en día la mayoría de los procesadores son superscalares, es decir, explotan la ejecución paralela de instrucciones para mejorar el rendimiento. Este modelo se desvía radicalmente del de ejecución secuencial, sin embargo, por razones de compatibilidad, se debe mantener/simular ciertos aspectos del mismo.

Este tipo de procesadores analizan los binarios secuenciales de los programas y lo paralelizan eliminando secuencialidad innecesaria. Por esta razón, los programas binarios deben ser vistos más como una especificación de lo que debe hacerse y no como lo que realmente sucede.

Más precisamente, un procesador superescalar implementa:

1. Estrategias de fetch que permiten fetchear múltiples instrucciones mediante la predicción de resultados y saltos.
2. Métodos para determinar dependencias de registros y mecanismos para comunicar esos valores cuando sea necesario durante la ejecución.
3. Métodos para iniciar, o resolver, múltiples instrucciones en paralelo.
4. Recursos para la ejecución en paralelo de varias instrucciones, incluyendo múltiples unidades funcionales de pipelines y jerarquías de memorias capaces de atender simultáneamente múltiples referencias a memoria.
5. Métodos para manejar datos a través de instrucciones de lectura/escritura e interfaces de memoria que tengan en cuenta el comportamiento dinámico (y muchas veces impredecibles) de las jerarquías diseñadas.
6. Métodos para actualizar los estados del proceso en el orden correcto (para mantener la apariencia de la ejecución en orden secuencial).

4.1. Dependencias de instrucciones

Por lo general, se interpreta el binario de un programa como un conjunto de bloques compuestos de instrucciones contiguas. Una vez que un bloque es fetchado, se sabe que todas sus instrucciones van a ser ejecutadas eventualmente. Cuando esto suceda, diremos que el bloque es iniciado en una **ventana de ejecución**.

Una vez que las instrucciones entran en esta ventana, son ejecutadas en paralelo teniendo en cuenta sus dependencias. Éstas, pueden ser de control o datos. Las de control son generadas por condicionales y pueden ser resueltas/optimizadas con predicciones (Sección 3).

Las **dependencias de datos** se da entre instrucciones que referencian el mismo espacio de memoria. En estos casos, si las instrucciones no se ejecutan en el orden correcto, puede haber errores en las operaciones. Pueden ser Verdaderas o Artificiales.

- **Dependencias verdaderas:** Cuando una instrucción debe leer un valor que todavía no fue generado por una instrucción previa (**Read after write hazard**).

- **Dependencias artificiales:** Resultan de instrucciones que deben escribir un nuevo valor en una posición de memoria pero debe esperar a que las instrucciones previas utilicen el valor actual (**Write after Read hazzard**) o cuando varias instrucciones deben escribir la misma posición de memoria (**Write after Write hazzard**).

Estas dependencias son producidas por código no optimizado, por escasez de registros disponibles, por el deseo de economizar el uso de la memoria principal o por ciclos donde una instrucción puede colisionar consigo misma.

4.2. Fetching and Decode

En los procesadores superescalares, una caché de instrucciones es usada para reducir la latencia y el ancho de banda del fetching. Esta caché está organizada en bloques o líneas que contienen varias instrucciones consecutivas y almacena el tipo de cada una de ellas (si es de control, de operación, de lectura de memoria, etc).

El program counter se utiliza para determinar la posición de una instrucción en la caché.

Si se produjo un hit, se fetchea el bloque de instrucciones y luego se le suma, al program counter, el tamaño del mismo. Si hay un miss, el caché debe pedir la instrucción buscada en memoria.

Una vez conseguido el bloque de instrucciones, se identifican el tipo de cada instrucción. Si alguna es de control, entonces se realizan las predicciones necesarias mediante alguno de los métodos nombrados en la sección 3. Luego, las instrucciones son decodificadas en **tuplas de ejecución** que contienen la operación a ser ejecutada, las identidad de los elementos donde se encuentran los parámetros de entrada y donde deben guardarse los resultados.

En el programa estático, las instrucciones utilizan los registros **lógicos** (los de la arquitectura). Por esta razón, cuando son decodificadas, cada uno de ellos es mapeado (o renombrado) a un registro físico y las dependencias artificiales son resueltas indicando a las instrucciones involucradas que usen distintos registros físicos (es decir, se mapean dos o mas registros físicos al mismo registro lógico).

Este mapeo se guarda en una tabla que asocia cada entrada a una instrucción y permite identificar el registro lógico que le corresponde. De esta forma, cuando sea necesario actualizar el estado visible de la arquitectura, la instrucción leerá su resultado del registro físico adecuado.

Una vez que todas las instrucciones asociadas a un registro físico son completadas (modifican el estado visible), el registro es liberado para que pueda ser usado por otro bloque de instrucciones.

4.3. Ejecución de instrucciones

4.3.1. Interrupciones

Existen dos tipos de interrupciones:

- **Excepciones** (o trampas), que se genera cuando se produce un error durante la ejecución o el fetching de ciertas instrucciones. Por ejemplo, opcodes ilegales, errores numéricos o page faults.

- **Interrupciones externas:** Son causadas por instrucciones específicas y dispositivos externos que están ejecutando algún proceso. Por ejemplo las interrupciones generadas por los dispositivos de entrada/salida (mouse, teclados, pantallas), timers, etc

Cuando ocurre una interrupción, el software o el hardware (o una combinación de ambos) guardan el estado del proceso interrumpido. Este estado consiste, generalmente, del program counter, los registros y la memoria. Si el estado guardado es consistente con la arquitectura secuencial del modelo, entonces diremos que la interrupción es **precisa**. Para ser más específicos, se debe cumplir que:

1. Todas las instrucciones previas a la indicada por el program counter, deben haber sido ejecutadas y deben haber modificado el estado del proceso correctamente.
2. Ninguna de las instrucciones siguientes a la indicada por el program counter debe haber ejecutadas ni deben haber modificado el estado del proceso.
3. Si la interrupción es causada por una excepción en una instrucción del programa, entonces el program counter guardado debe apuntar a la instrucción interrumpida.

Si el estado guardado es inconsistente con el modelo de arquitectura secuencial y no satisface estas condiciones, entonces la interrupción es **imprecisa**.

Hay varias formas de evitar las interrupciones imprecisas:

4.3.2. In-Order Instruction Completion

Las instrucciones modifican el estado del proceso solo cuando se sabe que todas las instrucciones previas están libres de excepciones. Para asegurar esto, se utiliza un registro llamado “result shift register” que contiene una tabla de n entradas (n la longitud del pipeline más largo). Una instrucción que toma i ciclos de reloj reserva la i -ésima entrada de la tabla y en cada ciclo se la desplaza una posición hacia abajo.

STAGE	FUNCTIONAL UNIT SOURCE	DESTN. REGISTER	VALID	PROGRAM COUNTER
1			0	
2	INTEGER ADD	0	1	7
3			0	
4			0	
5	FLT PT ADD	4	1	6
.
.
N			0	

↑
DIRECTION OF MOVEMENT

Fig. 2. Result shift register.

Figura 5: Result Shift Register

Si la i -ésima entrada contiene información válida, entonces la instrucción se pausa hasta el próximo ciclo de reloj y se rechequea la información de la misma.

Para evitar que una instrucción más corta se complete antes que otra de mayor longitud (cuando este es el orden deseado) se rellenan con información inválida todas las entradas anteriores que no fueron reservadas. De esta forma, la nueva instrucción es pausada hasta el próximo ciclo de reloj.

4.3.3. Reorder-Buffer

La principal desventaja del método anterior, es que instrucciones rápidas serán retenidas a pesar de no tener dependencias.

Para evitar esto, se permite que las instrucciones terminen en cualquier orden pero se les asigna un tag que indican el orden en el que deben modificar el estado de la computadora. Cuando son completadas, se almacena en la entrada indicada (por el tag) del buffer.

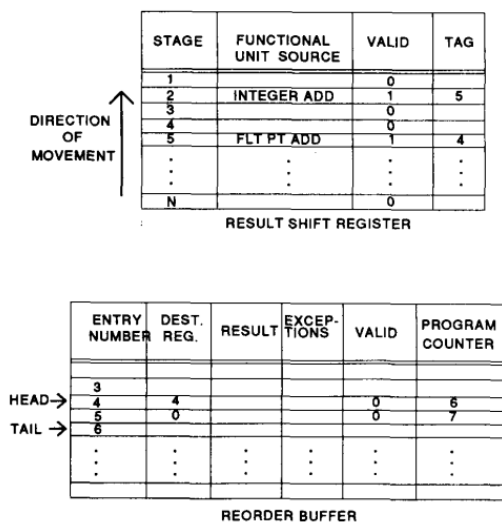


Figura 6: Reorder buffer

El buffer contiene un puntero a la entrada que corresponde a la próxima instrucción a ser ejecutada. Cuando dicha entrada contenga información válida, se chequea si hubo excepciones. Si no las hubo, se modifican los registros/memoria necesarios y se mueve el puntero a la próxima entrada. Sino, se emite la excepción y se invalidan todas las entradas posteriores.

En algunos casos, la dependencia entre instrucciones requiere el uso de valores que todavía no fueron escritos en los registros. Para conseguirlos es necesario acceder al buffer de reordenamiento, lo que agrega latencia y complejidad a la ejecución.

History Buffer: Una solución posible es dejar que las instrucciones modifiquen los registros cuando son completadas, pero que se guarden los valores previos en un buffer que permita recuperar estos datos en el caso de una excepción.

En este caso, si ocurre una excepción, se debe recorrer el history buffer para poder recuperar un estado preciso.

Future File: Otra solución es mantener dos archivos de registros. Uno que se actualice en el orden especificado por el programa (**Architectural File**). Y otro que se actualiza apenas se completa una instrucción (**Future File**).

Si ocurre una excepción simplemente se debe mostrar el Architecture File para conseguir un estado preciso. Sin embargo, hay que copiarlo al future file para restaurar el estado del sistema.

Checkpointing: Cuando se predice el resultado de una instrucción de control, se guarda el estado del future file (**checkpoint**). Si cuando se ejecuta la instrucción, la predicción resulta ser errónea, entonces solo hay que copiar la información del checkpoint en el future file para recuperar el estado del sistema y comenzar de nuevo en la instrucción correcta.

4.3.4. Out of Order Execution (Tomasulo)

El algoritmo de Tomasulo, crea un scheduling dinámico que permite ejecutar instrucciones fuera de orden y habilita el uso de múltiples unidades de ejecución.

Register Renaming: Para eliminar los WAR y WAW hazards, se utiliza una tabla llamada **Register Alias Table** que asocia un tag con cada operando de una operación.

Register Station (RS): Una vez realizado el renombre se le asigna, a la instrucción un RS. Un RS es un subsistema de hardware compuesto de bancos de registros internos que se encarga de mantener las instrucciones en espera hasta que esté lista para ser ejecutada.

Un RS chequea constantemente por la disponibilidad de los operandos de la instrucción. Para cada operando cuyo valor no esté disponible, el RS, guarda el *tag* que se le asignó en el paso anterior.

Cada vez que una unidad de ejecución pone disponible un operando, este transmite el **tag** asociado al operando junto con su valor a todas las RS.

Cuando una instrucción tiene todos sus operandos, el RS espera a que la unidad funcional correspondiente esté libre y luego despacha la instrucción hacia la misma.

Common Data Bus : Es un datapath que cruza la salida de las unidades funcionales y atraviesa las RS, los Floating Point Buffers, los Floating Points Registers y el Floating Point Operations Stack.

Pseudocódigo : Cada Registro contiene un tag que indica el último escritor en el registro.

```

if RS tiene discursos disponibles antes del renaming then
    Se inserta en la RS la instrucción y los operados renombrados (valor fuente / tag)
    Se renombra si y solo si la RS tiene recursos disponibles.
else
    stall
end if
while esté en la RS, cada instrucción debe do
    Mirar el tráfico por el Common Data Bus (CDB) en busca de tags que correspondan a
    sus Operandos fuente.
    Cuando se detecta un tag, se graba el valor de la fuente y se mantiene en RS.
    Cuando ambos operandos están disponibles, la instrucción se marca Ready para ser
    despachada
    if Unidad Funcional disponible then
        Se despacha la instrucción a esa unidad funcional
    end if
    if Finalizada la ejecución de la instrucción then
        La unidad funcional pone el valor correspondiente al tag en el CBD (tag broadcast)
        if El archivo de registros está conectado al CDB then
            if tag del Archivo de Registro == tag broadcast then
                Registro = valor broadcast
                bit de validez = '1'
                Recupera tag renombrado
                No queda copia válida del tag en el sistema
            end if
        end if
    end if
end while

```

Referencias

- [1] *18-447 Introduction to Computer Architecture - Spring 2015 Course of Carnegie Mellon University*, 2015.
- [2] James E. Smith and Andrew Pleszkun. Implementation of precise interrupts in pipelined processors. volume 13, pages 36–44, 06 1985.
- [3] Alejandro Furfaro. Organización del computador ii - clases teóricas, 2018.
- [4] Yale N. Patt. Requirements , bottlenecks , and good fortune : Agents for microprocessor evolution. 2001.

-
- [5] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.
 - [6] James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors, 1995.
 - [7] Sanjay Patel Yale Patt. *Introduction to Computing Systems: From bits and gates to C and beyond*. McGraw-Hill, 2nd international edition, 2005.