U.B.A. - Facultad de Ingeniería

66.20 Organización de Computadoras
Guía de estudio

Práctica jueves

Índice general

Prólogo

Durante varios cuatrimestres estuvo el interrogante si la deserción temprana y baja concurrencia no se debía a no contar con una guía fija y actualizada de ejercicios que ordene y estructure el curso.

La intención de este apunte es guiar al alumno a atravesar los temas de la materia con un breve resumen que sirve como introducción para luego si, atacar la bibliografía y poder resolver los ejercicios propuestos al final de cada capitulo. Se agregaron capturas de imágenes del libro de aquellas secciones que valen la pena resaltar, se resumieron las ideas principales y muchas de las ecuaciones y términos quedaron en inglés.

De ninguna manera se buscó que este apunte actúe como reemplazo de la basta bibliografía ni, menos como una traducción por más somera que sea, si bien muchas veces nos preguntamos que tanta incidencia traía esta barrera idiomática que extiende el material que utilizamos. Esperamos que sirva como motivación a los alumnos y que sirva como herramienta facilitadora para enfrentar esta materia.

Unidad 1

Principios fundamentales

¿Cómo pasa un programa escrito en un lenguaje de alto nivel como C o Java a traducirse a lenguaje de máquina, y cómo el hardware ejecuta el programa resultante?

¿Cuál es la interface entre el software y el hardware, y cómo el software controla el hardware para realizar las funciones requeridas?

¿Qué determina la performance de un programa, y como un desarrollador puede mejorar dicha performance?

¿Qué técnicas pueden ser utilizadas por ingenieros para mejorar la performance?

¿Cuáles son los motivos y las consecuencias del cambio de procesamiento secuencial al procesamiento paralelo?

Sin entendimiento acabado de las respuestas a estas preguntas, mejorar la performance de un programa en un sistema de cómputo moderno o evaluar que características podrían hacer una computadora mejor que otra para una aplicación particular, terminará siendo un proceso demasiado complejo de prueba y error en lugar de ser un procedimiento científico canalizado por el entendimiento y el análisis.

1.1. Desempeño

Cuando se trata de elegir entre distintas computadoras, el rendimiento o la performance es un atributo importante. Medir la performance en forma precisa y correcta para comparar distintas computadoras es crítica para los compradores y por lo tanto para los diseñadores.

Para maximizar la performance vamos a querer minimizar el tiempo de respuesta o el tiempo de ejecución de una determinada tarea. Con lo cual, podemos relacionar la performance y el tiempo de ejecución para una computadora X.

$$\text{Performance}_x = \frac{1}{\text{Tiempo de ejecución}_{tx}}$$

Si la performance de X es mayor que la performance de Y, tenemos que:

$$\text{Performance}_x > \text{Performance}_y$$

$$\frac{1}{\text{Tiempo de ejecución}_x} > \frac{1}{\text{Tiempo de ejecución}_y}$$

$$\text{Tiempo de ejecución}_y > \text{Tiempo de ejecución}_x$$

El tiempo de ejecución de Y es mayor que el de X, es decir, X es más rápida que Y.

1.2. Benchmarks

La mejor opción para comparar o medir performance (benchmark) son las aplicaciones reales, aquellas que se van a ejecutar, por ejemplo un compilador. El intento de correr programas que son mucho más simples que una aplicación real lleva a resultados de performace no del todo certeros. Ejemplos de estos pueden ser:

- kernels, partes pequeñas y significativas de una aplicación real.
- programas de juguete, son programas de no más de 100 líneas, como el quicksort.
- benchmarks sintéticos, son programas inventados que aprovechan ciertas características de una determinada computadora, como Dhrystone.

1.3. Ley de Amdahl

La ley de Amdahl determina que la mejora en la performance obtenida de usar un modo más rápido de ejecución está limitada por la fracción de tiempo donde puede ser utilizado dicho modo más rápido.

La ley de Amdahl define el speedup que puede ser obtenido al utilizar una mejora particular.

El speedup está determinado por la relación:

$$\text{speedup} = \frac{\text{Performance de toda la tarea utilizando la mejora cuando es posible}}{\text{Performance de toda la tarea sin ninguna mejora}}$$

O de otro modo,

$$\text{speedup} = \frac{\text{Tiempo de ejecución de toda la tarea sin utilizar la mejora}}{\text{Tiempo de ejecución de toda la tarea utilizando la mejora cuando es posible}}$$

El speedup indica que tan rápido se puede correr una tarea al utilizar una mejora en contraposición al tiempo original sin mejora.

La ley de Amdahl entonces nos brinda una forma rápida de encontrar el speedup de una mejora que depende de dos factores:

1. La fracción de tiempo del tiempo original que se puede aplicar una mejora.
2. La mejora obtenida al aplicar el nuevo modo, esto es, que tan rápido correrá el sistema si se aplica la mejora.

$$\text{Tiempo de ejecución}_{\text{nuevo}} = \text{Tiempo de ejecución}_{\text{viejo}} \times \left((1 - \text{Fraccion}_{\text{mejorable}} + \frac{\text{Fraccion}_{\text{mejorable}}}{\text{speedup}_{\text{local}}} \right)$$

1.4. Ecuación del desempeño de CPU

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

Se puede ver como estos factores son combinados para indicar el tiempo de ejecución en segundos por programa:

Componente de performance	Unidad de medida
Tiempo de ejecución de CPU para un programa	Segundos por programa
Instruction count (Cantidad de instrucciones)	Instrucciones ejecutadas por el programa
Ciclos de reloj por instrucción (CPI)	Promedio de ciclos de reloj por instrucción
Tiempo de ciclo de reloj (Clock cycle time)	Segundos por ciclo de reloj

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycles}}$$

La única medida confiable y completa para medir la performance de una computadora es el tiempo. Por ejemplo, al cambiar el set de instrucciones por uno más chico puede llevar a una organización con un tiempo de ciclo de reloj más lento o con un CPI más alto que hace que la mejora en el I_C se vea opacada.

De forma similar y porque el CPI depende del tipo de instrucciones ejecutadas, el código que ejecuta un número menor de instrucciones puede no ser el más rápido.

Es decir, que es muy complicado cambiar un sólo parámetro en forma aislada, dada la interdependencia que existe en cada uno por la tecnología que los involucra.

- Clock cycle time. Tecnología del hardware y la organización.
- CPI. Organización y la arquitectura del set de instrucciones.
- Instruction count. Arquitectura del set de instrucciones y la tecnología del compilador.

1.5. Medición de desempeño

El tiempo es la medida de performance: la computadora que realiza la misma cantidad de tareas en el menor tiempo es la más rápida. El tiempo de ejecución es medido en segundos por programa. Sin embargo, el tiempo puede ser definido de distintas maneras, dependiendo en como lo contamos. La definición más directa de tiempo es llamada tiempo de reloj de pared, tiempo de respuesta o tiempo transcurrido. Estos términos implican el tiempo total en concluir una tarea, incluyendo acceso a disco, accesos a memoria, actividades de I/O, overhead del sistema operativo, etc.

- CPU execution time o CPU time. El tiempo total que toma la CPU en computar una tarea específica.
- user CPU time. El tiempo de CPU que toma el programa en sí.
- system CPU time. El tiempo de CPU que toma el sistema operativo realizando tareas para el programa.

Una alternativa a medir tiempo es MIPS (millones de instrucciones por segundo). Para un dado programa, MIPS se calcula:

$$\text{MIPS} = \frac{\text{Intruction count}}{\text{Execution time} \times 10^6}$$

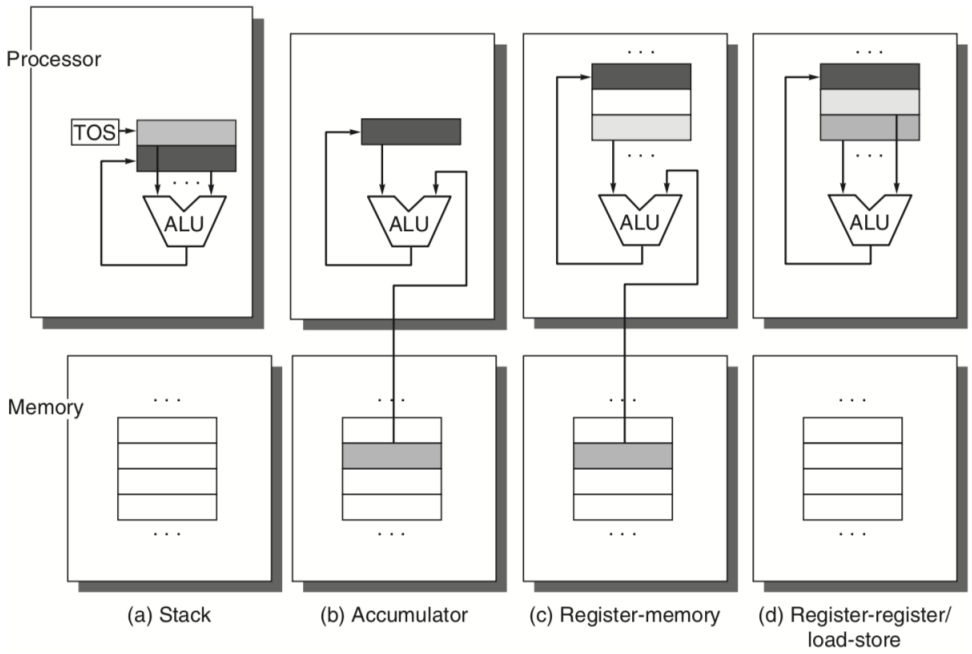
Vale destacar que no podemos comparar computadoras con diferentes set de instrucciones usando MIPS, dado que el IC será ciertamente diferente. Además, MIPS varía entre programas ejecutados en la misma computadora, por ejemplo, substituyendo el tiempo de ejecución, se ve la relación entre MIPS, clock rate y CPI:

$$\text{MIPS} = \frac{\text{Intruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Unidad 2

Arquitectura de programación

El término arquitectura del conjunto de instrucciones (instruction set architecture) (ISA) se refiere al conjunto de instrucciones visibles al programador. Funciona como la frontera entre el software y el hardware.



2.1. Modelo de pila

Los operandos en una arquitectura de stack están implícitamente en el tope del stack y el hardware debe evaluar la expresión en un solo orden y cargar un operando múltiples veces.

2.2. Modelo de acumulador

En una arquitectura de acumulador un operando está implícito en el acumulador.

2.3. Modelo de registros de propósito general

En la arquitectura de registros de propósito general se tienen únicamente operandos explícitos, ya sea que estén ubicados en registros o en memoria.

Los operandos explícitos podrían ser accedidos directamente de memoria o necesitar ser cargados de memoria a un registro temporal dependiendo de la arquitectura y de la instrucción específica.

2.4. Modelo de carga y almacenamiento

En este tipo de arquitectura, la memoria solo puede ser accedida a través de instrucciones de carga y almacenamiento (load/store).

2.4.1. Conjunto de instrucciones RISC

Las arquitecturas RISC se caracterizan por tener ciertas propiedades que simplifican su implementación.

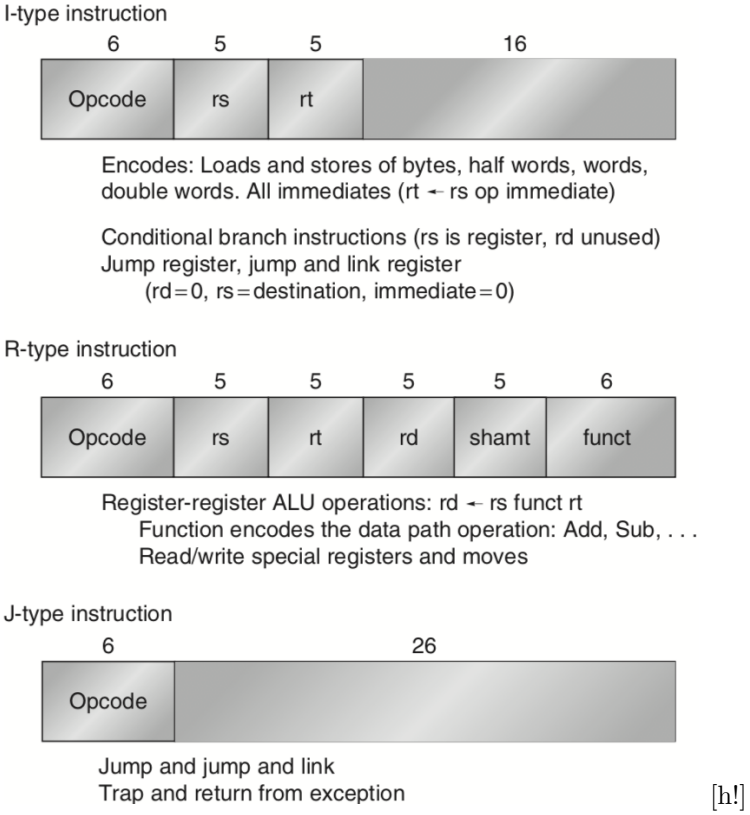
- Todas las operaciones de datos aplican a datos en registros y típicamente modifican todo el registro (32 o 64 bits por registro).
- Las únicas operaciones que afectan a la memoria son de carga y almacenamiento (load y stores) que mueven datos de la memoria a un registro o viceversa. Estas operaciones pueden mover datos menores que la capacidad de un registro (por ejemplo 16 bits).
- El número de formatos de instrucciones son pocos con todas las instrucciones del mismo tamaño.

Como otras arquitecturas RISC, el conjunto de instrucciones de MIPS tiene 32 registros, aunque el registro 0 siempre tiene el valor 0.

Hay tres tipos de instrucciones:

1. Instrucciones ALU. Estas instrucciones toman dos registros o bien un registro y un inmediato de signo extendido, realiza la operación y almacena el resultado en otro registro.
2. Instrucciones Load / Store. Estas instrucciones toman un registro base, y un inmediato como offset. La suma resultante del contenido del registro y el offset da la dirección efectiva y es usada para direccionar la memoria. En el caso de un store, el segundo registro opera como fuente del dato que se almacena en memoria.
3. Branches y jumps. Branches son transferencias condicionales del flujo de control. La forma de especificar la condición del branch, MIPS compara entre un par de registros o entre un registro y cero.

   El destino del branch se obtiene sumando un offset con extensión de signo (16 bits) al valor actual del PC.



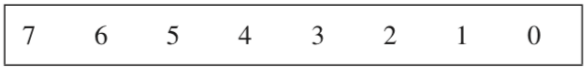
[h!]

2.4.2. Direccionamiento de memoria

Independientemente de la arquitectura, si es load-store o una que permita a cualquier operación tener una referencia a memoria, se debe definir como son interpretadas las direcciones de memoria y como se especifican.

Todo el conjunto de instrucciones es direccionado por bytes y se provee acceso por bytes (8 bits), mitad de palabra (half words) (16 bits), por palabra (words) (32 bits).

Hay dos convenciones para ordenar los bytes en un objeto. **Little Endian** coloca el byte cuya dirección "x . . . x000" en la posición menos significativa. Los bytes entonces son numerados:



[h!]

El orden **Big Endian** coloca el byte cuya dirección es "x . . . x000" en la posición más significativa. Los bytes entonces son numerados:

[h!]

El orden de los bytes es un problema cuando se intercambia datos entre computadoras con diferentes ordenamientos. Por ejemplo cuando se comparan strings, en Little Endian aparece "SDRAWKCAB" (backwards) en los registros.

2.4.3. Registros

MIPS64 tiene 32 registros de propósito general de 64-bit (GPRs) llamados R0, R1,..., R31. Además tiene 32 registros de punto flotante, F0, F1,..., F31, que pueden almacenar 32 valores de

simple precisión o 32 valores de doble precisión (64-bit). Cuando se almacena un valor de precisión simple, la otra mitad del registro no se utiliza.

El valor del registro R0 es siempre cero.

2.4.4. Tipo de datos

Los tipo de datos son bytes de 8-bit, half words de 16-bit, words de 32-bit y double words de 64-bit para enteros y datos de 32-bit de precisión simple y 64-bit de doble precisión para punto flotante.

2.4.5. Operaciones

Las operaciones en MIPS64 trabajan en enteros de 64-bit o punto flotante de 32 o 64-bit. Bytes, half words, y words son cargados en los registros de propósito general con ceros o con el bit de signo para llenar los 64 bits.

2.4.6. Modos de direccionamiento

Los únicos modos de direccionamiento son inmediato o por desplazamiento, ambos con campos de 16-bit. Por registro indirecto se logra al ubicar 0 en los 16-bit del campo de desplazamiento y direccionamiento absoluto se logra al usar el registro cero como registro base.

2.5. Application Binary Interface

Esta sección no pretende ser una descripción completa de la ABI presentada en [7], sino un detalle de los puntos que se consideran confusos, más importantes o que varían respecto al documento mencionado.

Esta explicación se basa en [7] y en las observaciones realizadas sobre código assembly generado con GCC, con nivel de optimización 0 ($-$O0).

Por lo tanto, se recomienda primero leer la sección "Function Calling Sequence" de [7], para luego aclarar y/o modificar con el contenido de esta sección.

2.5.1. Caller y Callee Saved Registers

Los siguientes registros deben ser salvados por la función llamada si ésta los debe modificar:

- ra, sp, fp (o s8), gp, s0...s7
- f20 a f30 (floating point)

El resto de los registros no se garantiza que su valor se preserve entre llamadas a funciones. Si la función caller necesita preservarlos, debe salvarlos ella en su stack frame.

2.5.2. Stack Frame

Cada función crea su Stack Frame siempre.

El Stack Frame se compone de areas de un tamaño múltiplo de 8 bytes, alineadas a 8 bytes.

Los registros en un area se almacenan de abajo hacia arriba, en orden según el número de registro.

Las áreas son, en orden de aparición (de arriba hacia abajo en memoria):

- General Register Save Area (obligatoria). Registros salvados:
  - Siempre: fp, gp;
  - Cuando la función es non-leaf: ra;
  - Si es necesario: el resto de los general purpose callee-saved regs.
- Floating-Point Registers Save Area: f20 a f30 si es necesario (ver 3-15 de [7] para más detalle)
- Local and Temporary Variables Area;
- Argument Building Area.
  - Se crea cuando la función es non-leaf
  - Al menos es de 16 bytes, aún cuando los argumentos del callee requieren menos.
  - Cuando los cuatro primeros argumentos son enteros o punteros, se pasan en a0 a a3, y el resto se guarda en esta área a partir de los primeros 16 bytes.
  - Los argumentos pasados en a0 a a3 son almacenados en esta área por el callee siempre.

Saved Regs Area (*)
Float Regs Area
Local Area
Arg Building Area $\leq$ 16 bytes

Cuadro 2.1: Layout genérico del stack frame. Las áreas marcadas con (*) son obligatorias.

Debugging: algunas reglas de creación de stack frame existen sólo para garantizar que las herramientas de debugging (ej: gdb) puedan puedan realizar un stack backtrace (es decir, algunas reglas pueden no seguirse y el programa funcionará correctamente, pero no se garantiza un correcto análisis post-mortem en caso de que el proceso termine de forma anormal).

fp ($30)
gp ($28)

Cuadro 2.2: Ejemplo de Saved Regs Area mínima - función leaf.

alignment [4]
ra ($31)
fp ($30)
gp ($28)

Cuadro 2.3: Ejemplo de Saved Regs Area mínima para una función non-leaf. También puede salvar el resto de los callee-saved registers.

Unidad 3

Jerarquía de memoria

3.1. Latencia y ancho de banda

El tiempo requerido para resolver un miss en el cache depende de la latencia y del ancho de banda de la memoria.

La latencia determina el tiempo en traer el primer word del bloque y el ancho de banda determina el tiempo en traer el resto del bloque.

3.2. Principio de localidad

El principio de localidad temporal nos indica que es muy probable requerir el mismo word en el futuro cercano, con lo cual es útil ubicar este bloque de manera tal que sea rápidamente accesible.

Por la localidad espacial también, hay altas probabilidades que otro dato en el bloque se requiera pronto.

3.3. Memoria cache

Cuando el procesador encuentra el dato requerido en el cache, se llama *cache hit*. Cuando el procesador no lo encuentra, ocurre lo que se conoce como un *cache miss*.

Un miss en el cache es manejado por el hardware y hace que el procesador que utiliza ejecución in-order se frene (stall) hasta que el dato se encuentre disponible.

3.3.1. ¿Dónde se ubica un bloque en el cache?

Si un bloque tiene un solo lugar para ubicarse en el cache, el cache se dice que es de mapeo directo, de correspondencia directa o direct mapped (DM). El mapeo se puede calcular:

(Block address) MOD (Number of blocks in cache)

Si un bloque puede ubicarse en cualquier lugar en el cache, el cache se dice que es completamente asociativo o fully associative (FA).

Si un bloque puede ubicarse en conjuntos restrictivos dentro del cache, el cache se dice que es asociativo por conjunto. Un conjunto es un grupo de bloques en el cache. Un bloque primero se mapea a un conjunto y luego ese bloque se puede ubicar en cualquier en cualquier vía para ese conjunto. El conjunto se elije con un bit de selección.

(Block address) MOD (Number of sets in cache)

Si hay n bloques en un conjunto, el cache se llama n-way set associative o de asociativo por conjuntos de n-vías.

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

3.3.2. ¿Cómo se encuentra un bloque en el cache?

En cada frame, el cache tiene un tag que proporciona la dirección del bloque. Este tag se verifica para cada bloque de cache para saber si coincide con la dirección que emite el procesador. Obviamente como la velocidad es crucial, cada bloque del cache se verifica en paralelo.

Esta figura muestra que partes conforman una dirección. La primera división es entre la dirección del bloque y el offset. Luego vienen los campos de tag y de index.

El offset selecciona el dato buscado dentro del bloque, el index selecciona el conjunto, y el tag es comparado para determinar si es un hit.

3.3.3. ¿Qué bloque debe reemplazarse ante un miss?

Cuando ocurre un miss, el controlador del cache debe seleccionar el bloque a ser reemplazado con los datos deseados. Una de las ventajas del cache de mapeo directo es que la decisión del hardware es muy simple, de hecho, no hay elección, solo se verifica un bloque si es hit y sino solo ese bloque se puede reemplazar. En la cache completamente asociativa o asociativa por conjuntos, se pueden elegir varios bloques para reemplazarse ante un miss.

- Random. Para distribuir uniformemente la carga en el cache, los bloques candidatos a ser reemplazados se seleccionan aleatoriamente.
- Least recently used (LRU). Menos recientemente usado. Para reducir la chance de descartar información que se va a requerir pronto, se registran los bloques accedidos.
- First in, first out (FIFO). Dado que LRU es muy complejo de calcular, una aproximación para determinar el bloque más antiguo puede ser esta estrategia.

3.3.4. ¿Qué ocurre en una escritura?

Hay dos opciones ante una escritura en el cache:

- Write-through. La información es escrita en ambos lados, tanto en el cache como en el sistema más bajo de memoria.
- Write-back. La información es escrita únicamente en el cache. El bloque de cache modificado es escrito en los niveles más bajos solo cuando es reemplazado.

Dado que los datos no son necesitados en una escritura hay dos opciones ante un miss de escritura:

- Write allocate. El bloque es almacenado ante un miss de escritura, seguido de las acciones ante un hit descritas arriba. Esta sería la opción más lógica dado que un miss de escritura se trata igual que un miss de lectura.
- No-write allocate. Esta alternativa hace que la cache no se vea afectada ante un miss de escritura, solo se modifica los niveles más bajos de memoria.

Es decir, si es no-write allocate el bloque permanece fuera del cache hasta que se trate de leer, pero en write-allocate por más que el bloque solo se escriba, este quedará en cache.

3.3.5. Tasa de desacierto

Una forma de medir las ventajas de las distintas organizaciones de cache es la tasa de miss o miss rate. El miss rate es simplemente la relación entre los accesos al cache que resultan en miss y el número total de accesos.

Los misses en el cache se pueden clasificar en tres categorías (clasificación de las tres C):

- Compulsivos. El primer acceso a un bloque siempre será un miss. Los misses compulsivos son aquellos que ocurrirían incluso si la cache fuese infinita.
- Capacidad. Si la cache no puede contener todos los bloques requeridos durante la ejecución del programa, ocurrirán misses debido al reemplazo de bloques que luego serán requeridos.
- Conflicto. Si la estrategia para ubicar un bloque en el cache no es completamente asociativa, van a a ocurrir conflictos que obligarán a reemplazar un bloque cuando se trae otro y este mapea a la misma ubicación por más que exista lugar disponible en la cache.

3.3.6. Tiempo promedio de acceso a memoria

Ahora podemos considerar el número de ciclos del procesador en el cual esta esperando los accesos a memoria, estos se llaman ciclos de stall.

La performance se puede calcular como el producto entre tiempo de los ciclos de reloj y la suma de los ciclos de procesar que está esperando a la memoria (memory stall cycles):

CPU execution = (CPU clock cycles + Memory stall cycles) × Clock cycle time

Esta ecuación asume que los ciclos de CPU incluyen el tiempo de hit en el cache y el procesador está parado durante un miss en el cache.

El numero de ciclos de stall depende de el número de misses y del costo por miss, que llamamos penalidad de miss o miss penalty.

$$\begin{aligned}\text{Memory Stall cycles} &= \text{Number of misses} \times \text{Miss Penalty} \\ &= IC \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss Penalty} \\ &= IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss Penalty}\end{aligned}$$

3.3.7. Cache Performance

Average memory access time = Hit time + Miss rate × Miss penalty

Tener en cuenta que el tiempo de acceso promedio a memoria es una medida indirecta de la performance, es mejor que la tasa de miss pero no es de ninguna forma un reemplazo al tiempo de ejecución.

3.4. Memoria virtual

3.4.1. ¿Donde se ubica un bloque en memoria principal?

La penalidad de miss para la memoria virtual es considerablemente alta porque involucra acceso a dispositivos de almacenamiento mecánicos de cinta magnética por ejemplo. Entre elegir bajas tasas de miss o algoritmos de ubicación más simple, los programadores y diseñadores de sistemas operativos eligen la baja tasa de miss dada la alta penalidad de miss. Con lo cual, los sistemas operativos permiten que los bloques se ubiquen en cualquier lugar en memoria principal.

3.4.2. ¿Cómo se encuentra un bloque si está en memoria principal?

La paginación se apoya en una estructura que es indexada por el número de página o por el segmento. Esta estructura contiene la dirección física del bloque. Luego, el offset simplemente se concatena a la dirección física de la página.

La estructura de datos que contiene la dirección física de la página usualmente toma forma de tabla de páginas y es indexada por el número de página virtual. El tamaño de la tabla es entonces el espacio de número de páginas en la dirección virtual.

3.4.3. ¿Qué bloque debe reemplazarse ante un miss en Memoria Virtual?

Casi todos los sistemas operativos tratan de reemplazar el bloque menos recientemente usado (LRU). Los procesadores tienen un bit de referencia que se setea cuando se accede a la página para facilitarle la tarea al sistema operativo.

3.4.4. ¿Qué sucede en una escritura?

La estrategia de escritura siempre es write-back dado que los niveles más bajos de memoria podrían involucrar discos magnéticos que toman millones de ciclos de reloj para ser accedidos. Es por esto que se incluye un bit de dirty para escribir bloques a disco únicamente si han sido modificados desde que fueron leídos.

3.4.5. TLB

Las tablas de página por lo general son muy grandes y están almacenadas en memoria, incluso, algunas veces se paginan.

El hecho de paginar significa que cada acceso a memoria requiere al menos dos accesos, uno para obtener la dirección física y otro para obtener el dato en sí.

Para reducir el tiempo de traducción, se utiliza una cache dedicada llamada *translation lookaside buffer*.

Una entrada en la TLB es como una entrada al cache donde el tag contiene una parte de la dirección virtual y la parte de datos contiene el número de página física, además del bit de protección, validez, de uso y de dirty.

Para cambiar el número de página física o el bit de protección de una entrada de la tabla de página, el sistema operativo debe asegurarse que la vieja entrada no se encuentre en la TLB, de otra forma el comportamiento será errático.

El bit de dirty significa que la página correspondiente está dirty, no que la dirección en la TLB o que el dato en el cache lo están. El sistema operativo resetea estos bits cambiando sus valores en la tabla de página y luego invalida la entrada correspondiente en la TLB. cuando se recarga la entrada desde la tabla de página, la TLB toma los valores adecuados de estos bits.

La TLB utiliza un esquema de ubicación completamente asociativa, con lo cual la traducción empieza enviando la dirección virtual a todos los tags. Por supuesto, el tag tiene que estar marcado como válido.

Unidad 4

Pipeline

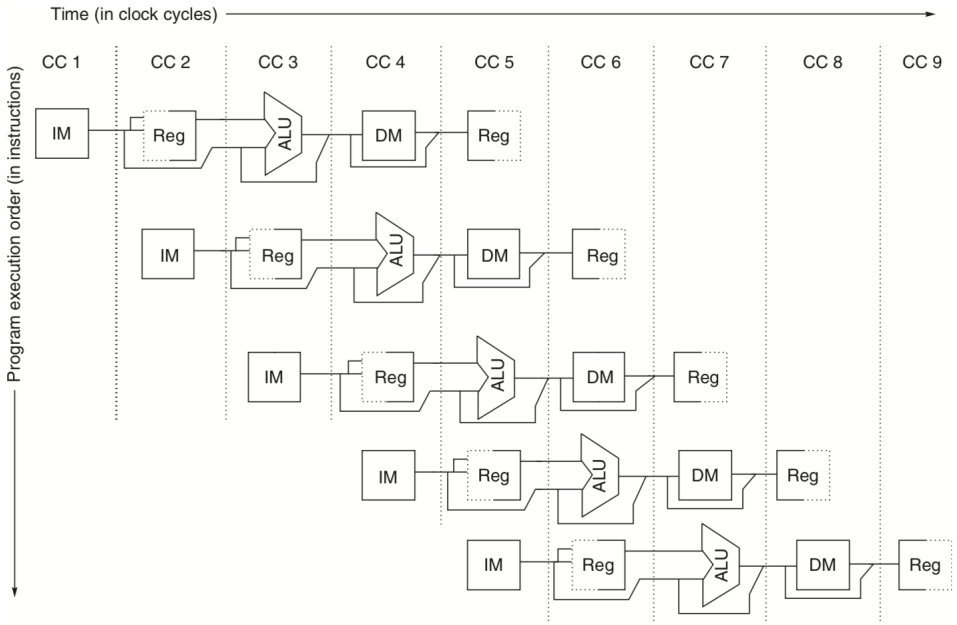
El pipelining es una técnica de implementación donde múltiples instrucciones son solapadas en ejecución; toma ventaja del paralelismo que existe entre las acciones requeridas para ejecutar una instrucción.

Un pipeline es como una línea de ensamblado. Por ejemplo, en una línea de ensamblado automotriz, hay muchas etapas, cada una contribuye a la construcción del auto. Cada una opera en paralelo de las otras para diferentes autos. En el pipeline de una computadora, cada etapa completa una parte de la instrucción.

Se define el *throughput* en una línea de ensamblado de autos como el número de autos por hora y se determina por que tan seguido se termina un auto en la línea de montaje. En las computadoras, esto es análogo, y se determina por que tan seguido se completa una instrucción en el pipeline.

Esta técnica de pipelining reduce el tiempo de ejecución promedio por instrucción. Dependiendo de que se considera como base, esta reducción puede ser vista como una reducción en el número de ciclos de reloj por instrucción (CPI) o como una reducción en el tiempo del ciclo de clock o como una combinación. Si se considera un procesador que toma múltiples ciclos de clock por instrucción, entonces pipelining es usualmente visto como una reducción del CPI.

Cada instrucción RISC puede ser implementada como mucho en 5 ciclos de reloj.



1. Instruction fetch cycle (IF):

   Se manda el program counter (PC) a memoria y trae la instrucción actual de memoria. Se actualiza el PC para la próxima instrucción secuencial sumando 4 dado que cada instrucción

es de cuatro bytes.

2. Instruction decode/register fetch cycle (ID):

   Se decodifica la instrucción y se leen los registros especificados del register file. Se verifica la igualdad mientras se leen los registros por un posible branch. Se extiende el signo en caso de ser necesario y se calcula la dirección destino del salto sumando el offset de la extensión de signo al PC.

3. Execution/effective address cycle (EX):

   La ALU ejecuta en los operandos dependiendo el tipo de instrucción:

   - Referencia a memoria. La ALU suma al registro base el offset para formar la dirección efectiva.
   - Registro - Registro. La ALU realiza la operación especificada en el opcode sobre los valores leídos del register file.
   - Registro - Inmediato. La ALU realiza la operación especificada en el opcode sobre el primer valor leído del register file y el inmediato con signo extendido.

4. Memory access (MEM):

   Si la instrucción es un load, se hace una lectura a memoria usando la dirección efectiva calculada en el ciclo anterior. Si es un store, se escribe a memoria el dato del segundo registro leído del register file usando la dirección efectiva.

5. Write Back cycle (WB):

   Escribe el resultado en el register file, ya sea que viene de memoria (load) o de la ALU.

En esta implementación, las instrucciones de salto requieren dos ciclos, las instrucciones de store requieren 4 y todas las demás 5 ciclos.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

4.1. Pipeline Hazards

Los riesgos (hazards) en el pipeline se pueden clasificar de tres formas:

1. Estructurales. Estos hazards ocurren por conflicto de recursos cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones simultáneamente.
2. De datos. Ocurren cuando una instrucción depende del resultado de una instrucción previa de forma que se solapan en el pipeline.
3. De control. Ocurren por los saltos y otras instrucciones que modifican al PC.

4.2. Data Hazards

4.2.1. Minimizando los Hazards de datos con forwarding

La técnica de forwarding funciona de la siguiente manera:

1. El resultado de la ALU ya sea de la etapa EX/MEM o MEM/WB se conecta de vuelta a las entradas de la ALU.

2. Si el hardware de forwarding detecta que la operación previa de la ALU ha escrito al registro correspondiente como fuente de la operación actual, se toma el resultado de las entradas de la ALU en vez del valor leído del register file.

Por ejemplo en la siguiente secuencia de instrucciones, no es posible aplicar forwarding.

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

4.3. Branch Hazards

Recordar que si un salto cambia al PC a la dirección target, el salto es tomado, si continua, entonces, es no tomado. Cuando la instrucción siguiente es de un salto tomado, en realidad, el PC no se modifica hasta el final de la etapa de ID.

El esquema más simple para manejar los saltos es freezar o vaciar el pipeline, manteniendo o eliminando las instrucciones después del salto hasta conocer el destino. En este caso, la penalidad de salto es fija y no puede minimizarse por software.

Otro esquema apenas más complejo es tratar cada salto como no tomado y dejar que se continúe como si el salto no se ejecutase. La precaución que hay que tener en cuenta es no cambiar el estado del procesador hasta que se conozca realmente la salida del salto.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

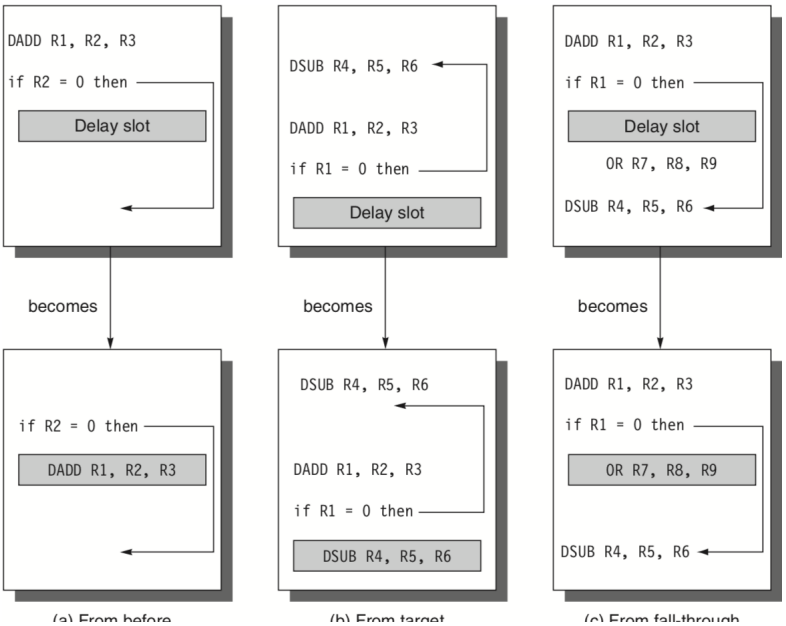
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Si el salto es tomado, hay que cambiar la instrucción que se hizo el fetch y convertirá a no-op y reiniciar el fetch a la dirección target.

Otro esquema para lidiar con los salto es el del salto demorado,

```
branch  instruction
sequential   successor
branch target if taken
```

La secuencia elegida se coloca en el branch delay slot. Esta instrucción es ejecutada de todas formas se tome o no se tome el salto.



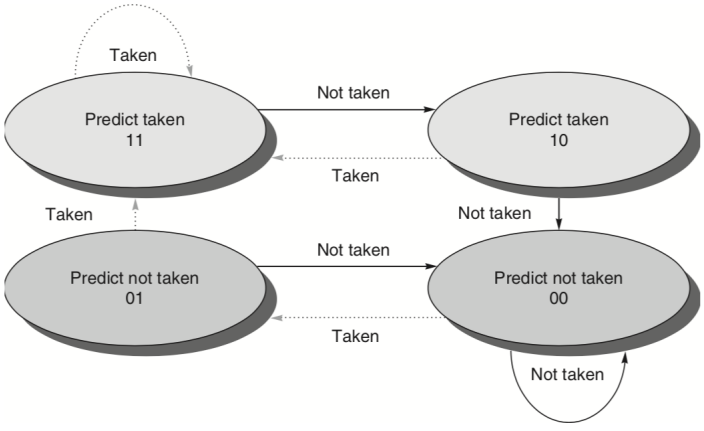
4.3.1. Dynamic Branch Prediction y Branch-Prediction Buffers

Un branch-prediction buffer es una memoria pequeña indexada por la parte mas baja de la dirección de la instrucción de salto. Esta memoria contiene un bit que indica si el salto fue recientemente tomado o no.

Este buffer efectivamente es una cache donde cada acceso es un hit y la performance depende de la predicción del salto y que tan bien se predijo.

El esquema de predicción de un bit no tiene tan buena performance ya que incluso si el salto siempre se toma se estaría prediciendo dos veces en forma incorrecta en lugar de una, cuando el salto no se toma el haber errado provoca que el bit se cambie.

Para solucionar este inconveniente se utiliza un esquema de 2-bit donde hay que predecir mal dos veces para cambiar la decisión.



Bibliografía

[1] David Patterson, John Hennessy, *Computer Architecture a Quantitative Approach*, Elsevier, 3rd edition. ISBN: 1-55860-596-7. May 2002.

[2] David Patterson, John Hennessy, *Computer Organization and Design, the Hardware/Software Interface*, Elsevier, 3rd edition. ISBN: 1-55860-604-1. Aug. 2004.

[3] B.L. Jacob and T.N. Mudge, *Virtual Memory: Issues of Implementation*, Computer, Vol. 31, No. 6, June 1998, pp. 33-43.

[4] B.L. Jacob and T.N. Mudge, *Virtual Memory in Contemporary Microprocessors*, IEEE Micro, Aug. 1998.

[5] Jean-Loup Baer, *Microprocessor Architecture. From Simple Pipelines to Chip Multiprocessors*, Cambridge University Press. ISBN-13 978-0-521-76992-1. 2010

[6] Rajeev Balasubramonian and Norman P. Jouppi and Naveen Muralimanohar, *Multi-Core Cache Hierarchies*, Morgan and Claypool Publishers, 2011.

[7] System V Application Binary Interface, MIPS RISC Processor, 3rd Edition, The Santa Cruz Operation, February 1996 (http://www.sco.com/developers/devspecs/mipsabi.pdf).