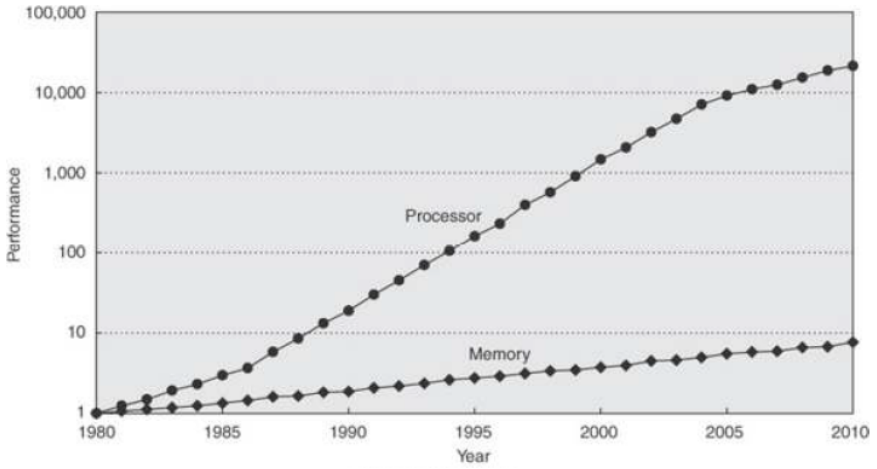
Resumen organización de computadoras 66.20

Tendencias tecnológicas:

- Ley de Moore: la densidad de circuitos integrados se duplica cada 2 años.

Brecha CPU memoria:
El procesador busca desempeño y los chips de ram capacidad de almacenamiento. (menor costo por bit y grandes capacidades de almacenamiento).



Mercado de computadoras: requerimientos

- Computadora de escritorio:
  - Precio- desempeño óptimo.
  - Tiempo de respuesta óptimo.
- Servidor:
  - **Disponibilidad:** quiere decir que la máquina esté encendida dando su servicio porque detrás del servidor hay un servicio que tiene que estar dando una respuesta. Para lograrlo, como el hardware puede fallar (fuentes de alimentación, discos magnéticos que están sometidos a movimientos), la idea es que no implique que el servidor tenga que apagarse sino que tenga elementos redundantes para que pueda seguir dando su servicio y que pueda repararse manteniéndose prendido. También que si por ejemplo se quema un disco que no se pierda información.
  - **Escalabilidad:** se va haciendo crecer el servidor en la medida que crece el negocio.
  - **Desde el punto de vista del desempeño, trabajos por unidad de tiempo:** maximizar la cantidad de tareas por unidad de tiempo porque va a haber muchas personas/computadoras que están solicitando muchas cosas al mismo tiempo y tiene que poder atenderlos.
- Procesadores embarcados: sistema de cómputo que está presente en algún aparato.
  - Desempeño a precio mínimo: que logre el desempeño que requiere para la aplicación a precio mínimo.

    - Mínimo consumo de potencia: importante para los dispositivos que funcionan con una batería. Los que consumen poca potencia son más caros.

Procesadores para aplicaciones embarcadas DSP

Procesador embarcado para señales digitales (audio, video, etc). Procesa un flujo de datos continuo de datos al cual generalmente hay que aplicarle el mismo algoritmo. El tipo de dato que viene de las señales suele ser más chico que el tamaño de palabra convencional de una máquina de escritorio. Suelen incorporar características que son propias para hacer más eficiente el procesamiento de estos datos.
Implementa un buffer circular para no chequear por los límites, el flujo de datos es continuo y se reusa la zona del buffer. En este buffer es en el que entran los datos.
Por el lado de la aritmética, tiene una aritmética de saturación. Si te queres pasar del máximo de representación no falla, tiene un chequeo para que se mantenga en el máximo de volumen por ejemplo.
Implementa la multiplicación por acumulación porque casi todo el tiempo se están haciendo operaciones con matrices o vectores.

Medida de desempeño

Para medir el desempeño vamos a usar el tiempo de ejecución. Es la medida segura. Este no es el tiempo de reloj de pared, sino que es el tiempo que la CPU dedico a la aplicación, ya que la CPU no está trabajando todo el tiempo en el programa.

Programas para evaluar desempeño

Para testear se puede utilizar las aplicaciones que usará el usuario (aplicaciones reales). A veces por una cuestión de incompatibilidad, etc no se puede utilizar las aplicaciones reales y se las puede modificar. Estas modificaciones deben ser pequeñas y se llaman *aplicaciones modificadas.*

- kernel: se puede utilizar para evaluar una porción aislada de una aplicación real).
- benchmarks de juguete: son pequeños algoritmos que no ponen en evidencia bien el desempeño de una máquina, con lo cual no sirven para evaluar desempeño.
- benchmarks sintéticos: son programas no reales del usuario que tratan de ser representativos de la mezcla de instrucciones y de trabajo del usuario.

SPEC: organización que se encargó de reunir benchmarks y agruparlos para utilizarlos como testers.

Principios cuantitativos

La idea es implementar por hardware lo que más frecuentemente se utiliza y hacerlo eficiente. A menudo el caso más frecuente es el simple.

Ley de amdahl

Calcula la mejora del desempeño que se puede obtener al mejorar alguna parte de la máquina. Esto está limitado al tiempo en el que se puede utilizar dicha mejora.

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

CPI

Se define a los CPIi como los ciclos por instrucción promedio para cada grupo de instrucciones 'i'. Se definen también los CPI individuales para cada instrucción como los ciclos requeridos para la ejecución de cada una de ellas.

Los manuales de los procesadores los suelen especificar y siempre serán dados en condiciones óptimas de ejecución. Es decir serán los mejores valores de CPI posibles de obtener durante la ejecución.

Ecuación de desempeño del CPU

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

Parámetros interrelacionados:

- IC: depende del programa, del compilador, de los flags, de la arquitectura del conjunto de instrucciones.
- CPI: depende del IC y de la organización de la CPU. Todo lo que cambie el IC cambia el CPI. El tipo de implementación de la arquitectura del conjunto de instrucciones va a afectar al CPI.
- Periodo de reloj: de la organización de la CPU y la tecnología del hardware.

Otras medidas de desempeño: MIPS

Fuertemente dependiente de la arquitectura del conjunto de instrucciones. No es adecuada para evaluar desempeño. Que un procesador ejecute instrucciones a una tasa muy alta no significa que sea rápido porque hay que ver cual es la complejidad de ese tipo de instrucciones que está ejecutando. La que tiene un MIPS más alto quizás tiene que ejecutar más instrucciones (tiene un IC mayor).

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

Clasificación de las arquitecturas del conjunto de instrucciones

Criterio de Clasificación: según dónde se encuentran los operandos para una operación de ALU.

STACK

Los operandos están en una estructura interna de CPU que se llama stack y también hay un puntero que apunta al tope del stack. Es un stack hardware que tiene que estar en CPU, no

es el stack software de las funciones. Toma los operandos del stack y almacena el resultado en el stack. Suponemos que ya buscamos los operandos de memoria.

En el tope apunta al último lugar ocupado. Si tengo que hacer una operación ALU toma las dos posiciones superiores del stack y para guardar el resultado disminuye el tope en uno y guarda el resultado pisando uno de los operandos.

No hay transferencia de bytes de datos entre CPU y memoria durante la operación de ALU.

ACUMULADOR
Un operando es un registro (único) del CPU llamado acumulador y el otro operando fuente está en memoria. El resultado se guarda en el acumulador (se pierde el valor fuente del acumulador). No son prácticos ni eficientes para una pc de uso de propósito general.

Se transfieren los bytes del dato de memoria a la CPU durante la operación de ALU.

REGISTRO MEMORIA
Varios registros. Un operando fuente es un registro y el otro está en memoria. El resultado va en un registro. Se disponen en CPU de varios registros que se pueden usar como un fuente y un destino, el otro fuente viene de memoria.

Se transfieren los bytes del dato en memoria a la CPU durante la operación de ALU.

Ventajas:
- Se pueden acceder datos sin un load por separado primero.
- Tendencia a ser fácilmente codificado.

Desventajas:
- Los operandos no son equivalentes.
- Pocos bits disponibles para codificar el registro.

REGISTRO REGISTRO (LOAD STORE)
Los operandos están en registros y el resultado se escribe en un registro. No hay restricción en cuanto a fuentes y destino.

No hay transferencia de bytes de datos entre CPU y memoria durante la operación de ALU.

Ventajas:
- Simple
- Codificación con instrucciones de tamaño fijo.
- Modelo de generación de código simple.
- Las instrucciones toman similares ciclos de reloj para su ejecución.

Desventajas:
- IC más alto.
- Programas más grandes.

MEMORIA MEMORIA

Los operandos y el destino están en memoria. La CPU no necesita usar registros. Los especificadores del operando en memoria son más grandes que en una load store porque las direcciones de memoria son más largas que lo que almacenan los registros.

Se transfieren los bytes del dato en memoria a la CPU durante la operación de ALU.

Ventajas:

- Código compacto.
- No desperdicia registros como temporarios.

Desventajas:

- Gran variación en el tamaño de instrucción.
- CPI muy variable.
- Cuello de botella en la memoria.

Lo que se busca es implementar lo que más frecuentemente se usa, hacer favorable el caso frecuente y hacerlo muy eficientemente. Para hacerlo eficiente se debe implementar con pipeline, y el pipeline necesita que las instrucciones sean de tamaño fijo. Para esto se necesita implementar las operaciones entre registros.

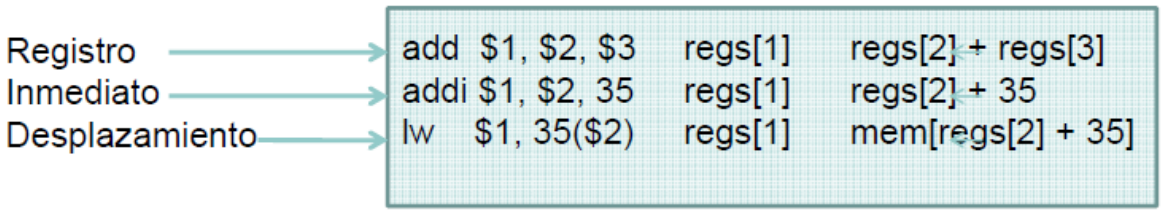
Interpretación de las direcciones de memoria:

Little endian guarda el byte *menos* significativo del word en la dirección más baja.
Big endian guarda el byte *más* significativo del word en la dirección más baja.

Direccionamiento de memoria

Alineación: se dice que un dato está alineado si la dirección del dato modulo tamaño del dato da cero. Si el dato está desalineado tiene que acceder dos veces a la memoria a agarrar cachitos del dato.

Modos de direccionamiento en MIPS 32:



- Inmediato: el valor inmediato viene incorporado en la instrucción, no hay que buscarlo en memoria.
- Registro: leer y escribir en los registros de CPU, no va a memoria.
- Desplazamiento: el único modo que va a la memoria. Usa un registro base que le suma un offset o desplazamiento para formar la dirección.

Soporte multimedia para desktop RISC

Aparece con intel mmx (SIMD) la extensión de instrucciones multimedia. El dato multimedia es un dato de tamaño pequeño (pixel, sonido) y los procesadores tenían un tamaño de 32 bits. Los algoritmos tienen que hacer cuentas etc con datos de tamaño menor y ocupa la

ALU y se desperdicia tamaño. Se puede dividir una ALU de 64 bits en 4 de 16 bits para ejecutar en paralelo.
Si se logra dividir así se logra ejecutar en paralelo. El compilador cuando ve que dispone de esta extensión puede ejecutar instrucciones en paralelo y acelerar la ejecución del programa.

Instrucciones de flujo de control de flujo de programa
Los que más abundan son los saltos condicionales. Los que van a causar más dificultad para el desempeño debido al pipeline son los saltos y los condicionales son los que más trabajo dan. Se van a implementar las condiciones que más se utilizan.

Modos de direccionamiento para instrucciones de control:

- Relativos al pc: los branch. Donde vamos a especificar una cantidad de instrucciones a saltar para atrás relativas al pc.
- Directo, pseudodirecto: no podemos especificar los 32 bits porque necesitamos bits para el opcode.
- registro indirecto: cuando decis que queres saltar a un label, pero cuando retornas en tiempo de ejecución podes retornar a distintos lugares. Cuando vuelve, el lugar debe estar guardado en un registro que se guarda en tiempo de ejecución. Para saltar al registro usamos jump register.

Evaluación de las condiciones de los branch
Tenemos CPU que tiene un registro de condición que mediante bits individuales guardan como fue el estado de la última operación. Entonces las instrucciones de salto saltan en base al bit que chequean del registro de código de condición. Con lo cual, está desacoplado el salto de la operación propiamente dicha sino que el pasaje de información es indirecto vía el registro de ejecución. Un salto lo vas a programar. Saltar en base a la condición de un registro. La contra que puede tener: estado extra porque tiene el registro de código de condición y limita el paralelismo porque tal vez la condición para el salto la establece la instrucción anterior.

Mips tiene la comparación de dos registros como condición. Pero esto puede enlentecer el tiempo de reloj de la máquina. También se puede comparar con una constante. Comparaciones simples → hardware menos complejo.

El tiempo de CPU está determinado por el tiempo del trabajo más lento. Las instrucciones de tamaño fijo benefician al pipeline

Programando en lenguaje ensamblador

Las directivas al assembler no son instrucciones, son directivas. Las pseudo instrucciones van a ser reemplazadas por el assembler por una o más instrucciones, no son instrucciones realmente. Mips 32 tendrá pocas instrucciones y simples pero son super importantes como para poder resolver cosas en pocos pasos.

Proceso de producir un ejecutable
De los fuentes que son archivos de texto en lenguaje ensamblador son tomados por una pieza de software, que es el assembler, que genera el programa objeto. A este programa

objeto le quedan por resolver información de funciones que está en otros archivos fuentes que no están contenidas en el archivo que se está ensamblando sino otros archivos fuentes. Por eso no es un ejecutable. Se necesita entonces un paso más que es alguien que los una con otros objetos, con funciones de biblioteca y esto lo hace el linker. Ahí tenemos variantes de cómo se generan estos archivos.
Antes se llamaban monolíticos porque incluía a las funciones y a la biblioteca en el mismo ejecutable. La contra es que eran más grandes los ejecutables. Después aparecieron las bibliotecas de enlace dinámico.
El compilador genera un lenguaje ensamblado que luego es traducido.
Tener varios registros no restringidos a ciertas operaciones se traduce en eficiencia para el compilador. Cuando opera con palabras de 64 bits usa registros de a pares.

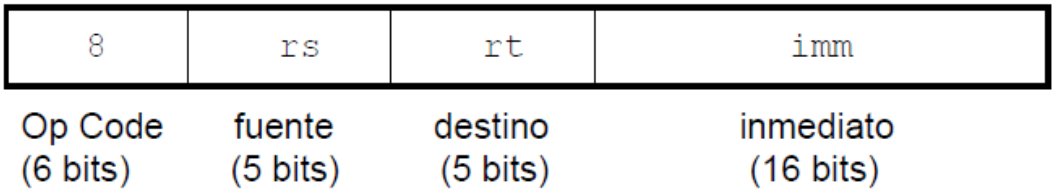
Registro lo y registro hi: En la multiplicación se guarda el resultado en HI y LO. En división se guarda el resultado en uno y el resto en el otro.

Coprocesador: si no se tiene un coprocesador de punto flotante, hay bibliotecas de emulación de instrucciones de punto flotante.

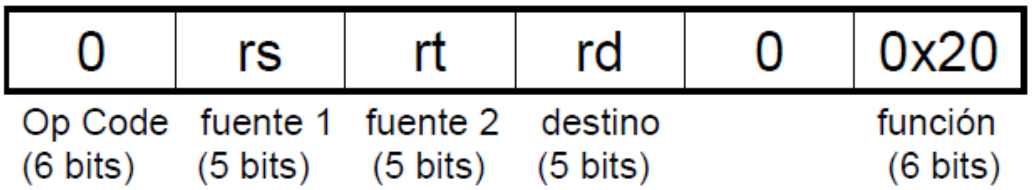
Excepciones: se detiene la ejecución ante un evento.

Formatos de instrucciones:

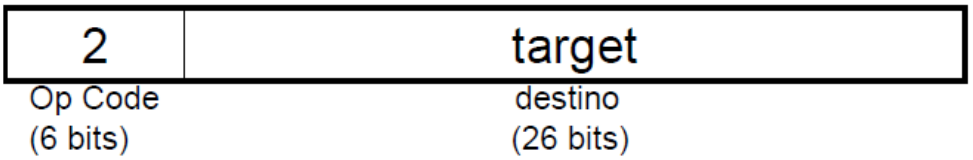
- Tipo I



- Tipo R



- Tipo J



Jerarquia de memorias

Los tres niveles de jerarquía de caché. Si un nivel figura compartido quiere decir que se comparte entre los núcleos.

En el caso de la memoria el desempeño se mide en función del acceso. No tiene sentido tener un CPU que es rápido pero tiene que esperar a que la memoria le entregue un dato por miles de ciclos.
Para la cpu se busca un mayor desempeño pero en el caso de la memoria se busca el menor costo por bit, de gran tamaño.

El principio de localidad para las referencias a memoria
Podemos arreglar este desbalance por el principio de localidad a las referencias a memoria.
• **Localidad espacial:** si el programa referencia una dada dirección de memoria, es muy probable que se referencie a las direcciones vecinas en un futuro inmediato. Idea de vecindad en cuanto a la dirección en una o varias posiciones de memoria.
• **Localidad temporal:** si el programa referencia una dada dirección de memoria, es muy probable que la vuelva a referenciar en un futuro inmediato. Da la idea de que los programas suelen revisitar a la instrucción ejecutada o un dato utilizado y en un tiempo corto.

Composición secuencial del código: si se ejecuta una instrucción a nivel procesador lo más probable es que se ejecute la siguiente en memoria salvo que haya algún salto.
Variables locales vecinas: generalmente se ubican vecinas en memoria. Las variables locales en las funciones se van a estar referenciando mientras la función esté activa.

Reuso de variables: variables para indexar, o comparar.
Reuso de funciones: para código, reuso la misma zona de memoria.
Bucles: revisito el código tantas veces se ejecute este y también accedo a las variables.

Memoria caché
Entre la CPU y MP se coloca una memoria caché de manera tal que pueda capturar la localidad presente del programa y se le pueda dar a la cpu las instrucciones y datos que pide de la memoria caché sin ir a la MP. Dado que la localidad es alta, la memoria caché puede ser pequeña y como esta es pequeña (tiene menos niveles de decodificación, etc) puede ser rápida. Normalmente está integrada en la cpu. La caché se implementa con memoria estática para que la velocidad pueda estar bien de acuerdo con la cpu porque es más pequeña y por el tipo de tecnología que utiliza.

Con el paso del tiempo y la brecha creciente la memoria empieza a ser lenta, mismo para la caché. Cuando las cosas no se encuentran en caché se accede a MP. Luego se agregó un segundo nivel y hoy en día hay 3. El nivel 1 está más cerca de la cpu. Los niveles van aumentando en tiempo de acceso a medida que crece el nivel.

Los tamaños se hacen más grandes hacia abajo y más lentos, y hacia arriba el costo por bit es mayor. Lo que está integrado en el procesador es más caro y la memoria estática no logra una buena densidad, requiere más componentes por celda (5 o 6 transistores por celda para la estática y la dinámica 1).

Una de las cosas que vamos a tener presente es que la caché va a ser transparente al software salvo en algún momento que aparezca alguna optimización o esquema que en alguna medida podrá ser visible. Pero en general no lo sabe. Los programas siguen funcionando de la misma manera.

Cuando se indica el tamaño de caché en el procesador generalmente es la L3. Un 75% del área del procesador es la caché.

Funcionamiento básico

La cpu pide palabras o bytes. El fetch busca la instrucción. Por otro lado, la caché tiene que obtener las instrucciones y los datos para guardarlos/capturarlos, capturar la localidad, de la MP. Tiene una interfaz con la MP a la derecha. Si bien es lenta en cuanto al tiempo de acceso al dato de la memoria, luego de esa latencia, los datos que siguen se obtienen con una tasa. Tiene una latencia y un ancho de banda. Dado que al buscar una palabra las que vienen a continuación se pueden obtener muy rápido, para aprovecharlo se organiza la caché no de una manera que haya palabras individuales, sino que en bloques para poder aprovechar las características del ancho de banda de la memoria principal. Nos conviene pensar lógicamente tanto a la caché como a la MP como una *secuencia ordenada de bloques.*

Si la cpu busca una instrucción que no está en cache, hay que traerla a la cache para tenerla (porque se asume que se la va a volver a pedir) y traigo a los vecinos porque gano por el ancho de banda, capturo localidad espacial.

Bloques:

- captura localidad espacial
- aprovecha ancho de banda (tasa a la cual puede transmitir la MP después de una latencia)
- interfaz entre caché y memoria principal.

Solo trae el bloque que contiene el dato que está pidiendo el procesador. Si trae más bloques es una optimización (se ve más adelante, prefetching).

El ancho de banda es la tasa a la que después de una latencia se accede a words. La organización por bloques hace que la brecha entre cpu y memoria se allane bastante.

Funcionamiento básico

La caché tiene copias de la memoria principal. Es para guardar información redundante, y es redundante porque es de acceso rápido.

La cpu busca una instrucción/dato. Si la caché tiene lo que busca la cpu es hit y se lo da en un tiempo rápido. Si no está en cache es miss y la cpu tiene que parar (sin hacer nada en principio) esperando que se traiga la instrucción/dato a la caché.

Trabaja sobre demanda. Cuando arranca el programa, la primera instrucción no va a estar en cache y es un miss. Ahí se trae el bloque, hasta que cambia de bloque y se repite. "Arranca en frío". Los arranques de los programas son lentos en ese sentido.

Cache split o unificada

En la mayoría de los procesadores de alto desempeño son split. Logran duplicar el ancho de banda porque al ser memorias independientes se acceden en paralelo. Y además se

pueden diseñar los parámetros por separado para que sea óptimo para cada tipo de info que guarda cada tipo. Las L2 y L3 son unificadas.

Las caches de datos suelen tener más misses y las caches de instrucciones tienen más alta tasa de hit porque el código normalmente es más local por ser secuencial.

Organizaciones de caché

La cpu le pide la dirección de un byte. La cpu no ve nada. Si cambia la organización de caché la dirección sigue siendo la misma.

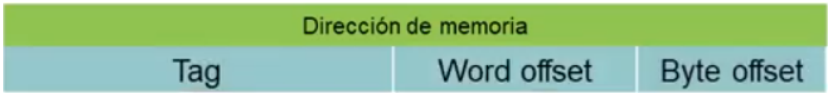
- Cache totalmente asociativa.

El ancho de la memoria es de un bloque y la caché también. El mapeo o política de ubicación es a nivel de bloque. No hay restricción de ubicación del bloque.

¿Cómo se busca?
La cpu pide (no va a buscar) y pide por dirección. *El hardware controlador de caché* va a tomar la dirección y va a buscar en la caché para ver si tengo un hit. El controlador interpreta la dirección en dos campos (pero uno lo podemos subdividir en dos partes).
Tag y offset de byte (dentro de un bloque).

En esta organización el tag coincide con el número de bloques. Se divide una dirección en 32 para ver en qué bloque cae. Con un bloque de 16 bytes tengo 4 words.

Tiene que haber un tag asociado a cada bloque, entonces el tag que coincide con el número de bloque en la tot asociativa ya me define quien es el bloque de memoria que tengo guardado. ¿Por qué guardo el tag y no la dirección completa? Porque la dirección de comienzo del bloque tiene los bits de offset en cero.
El bloque que da hit lo tengo decodificado, y tengo que tener un esquema general de que bloque da hit. Se codifica devuelta y el encoder elige en el mux el bloque que dio hit, pero solo si se activa la señal de hit.



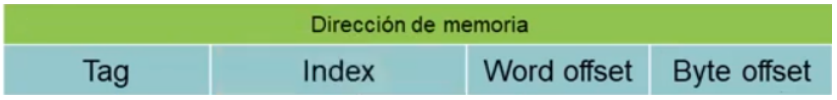
El encoder genera el número de bloque que contiene el tag pedido. El byte offset en 00 indica que se quiere el byte de inicio, cuando queremos pedir words estos van a estar en cero siempre.
El circuito controlador de caché está en el sistema de memoria caché. El hardware separa los bits de tag y offset. Hay un comparador para cada bloque.

- Caché por correspondencia directa:

Opuesto al anterior. Con una cuenta (operación de módulo) se obtiene el número de bloque en memoria caché en el cual este puede alojarse.
Bloque en caché = nro de bloque % cantidad de bloques en caché.
Varios bloques comparten o pueden alojarse solo en un bloque de memoria caché.

La tasa de hit en general aumenta cuando no hay restricciones en la ubicación.

¿Cómo se busca?
La dirección se interpreta con un campo más. Con la hipótesis de que la cantidad de bloques de cache es una potencia de dos, mirando los dos bits más bajos del número de bloque entonces esos bits lo llamamos índice y lo usamos para indexar de manera directa. El número de bloque es *tag* + *index* y *word offset* + *byte offset* es offset de byte dentro del bloque. Pero como necesito hacer el modulo lo divido. Con los bits de índice indexo la memoria de tags para ver si tengo hit, porque la operación de módulo me lleva directo al bloque que tengo que comparar. En este caso necesito un solo comparador para toda la caché. Leo lo que hay guardado y si coincide tengo un hit. Para acceder a la info el arreglo de words es más grande porque tengo un bloque, indexo con la concatenación de *índice* + *offset*.

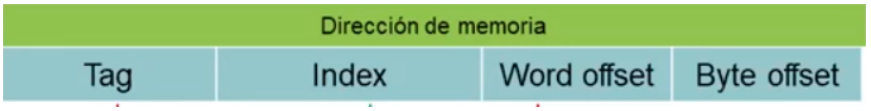


Con el índice acceder al arreglo de tags. Tiene un solo comparador y no hay multiplexor.

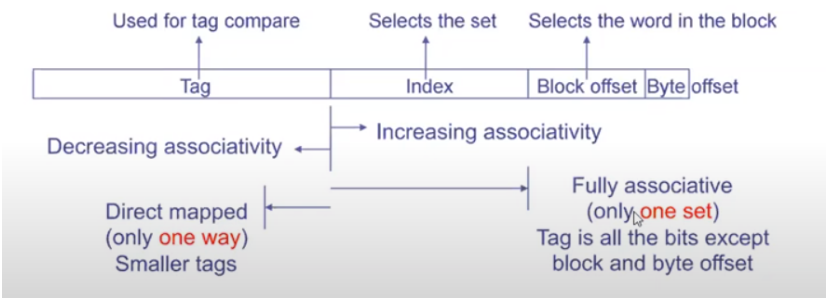
- Caché asociativa por conjuntos:

Se divide a la memoria caché en conjuntos. El grado de asociatividad es la cantidad de bloques por conjunto.
Por mapeo elijo el conjunto, y dentro del conjunto es totalmente asociativa. La política de ubicación es con operación de módulo pero eligiendo el conjunto.
Nro de conjunto en caché = nro de bloque en MP % cantidad de conjuntos.
La cantidad de bloques por conjuntos se llama grado de asociatividad o nro de vías.

¿Cómo se busca?



El conjunto es horizontal (la parte asociativa). Con los bits de índice elijo el conjunto. Indexo el área de tags con el índice. Con el índice elijo el conjunto y después comparó el tag buscado con los tags del conjunto.



A medida que aumenta la asociatividad decrecen los bits de índice.
La caché totalmente asociativa para la mayoría de los programas es la que *da la mejor estadística* (mejor tasa de miss) y la de correspondencia directa es la de peor estadística. La otra es intermedia. Desde el punto de vista de tiempo la de correspondencia directa es la mas rapida porque no tiene comparador ni encoder ni multiplexor con lo cual tiene hardware más simple, y la totalmente asociativa es la más compleja, tiene mucha actividad paralela, encoder mas grande y multiplexor más grande. Se elige más que nada para el nivel uno la asociativa de dos vías. Las caches más cercanas al procesador tienen menos vías.

Políticas de reemplazo

Cuando hay un miss hay que ver qué bloque se reemplaza por el bloque que está produciendo el miss porque necesito que ese este si o si en caché. Si la caché es totalmente asociativa, todos son candidatos a salir. En el caso de la caché asociativa por conjuntos, los candidatos están dentro del conjunto. Para la de mapeo directo no hay política de reemplazo.

- LRU: elige el bloque que hace más tiempo no se usa porque es el menos probable a ser utilizado. Tiene la propiedad de inclusión.

- FIFO: saca el bloque que se agregó hace más tiempo. El primero que entró. Ignora la info de reuso del bloque. No tiene la propiedad de inclusión.

- RANDOM: elige al azar. Evita elegir mal de manera sistemática.

- BASADAS EN DISTANCIAS DE PILA LRU: la distancia de pila se establece según la cantidad de bloques distintos que fueron referenciados desde la última vez que se referenció dicho bloque. Serían referencias intermedias a bloques distintos. La idea es estimar la próxima distancia de pila LRU para sacar la que tiene la más alta para alcanzar la política de reemplazo de belady.
  Modelado de la pila LRU: podemos hacer el modelado de la pila. Los bloques que se van referenciados se ponen en el tope. Cuando pido un bloque que ya estaba, lo saco de donde estaba y lo pongo en el tope y los demás caen un lugar.
  La distancia de pila LRU de esa referencia es menor que el tamaño de la caché en bloques es HIT, sino miss.

- OPTIMA (belady): el bloque óptimo a desalojar de la caché es el bloque que va a ser referenciado más lejos en el futuro. Necesita información del futuro. No es implementable desde el punto de vista práctico.

Propiedad de inclusión

Significa que una caché más grande que otra sometidas las dos al *mismo patrón de referencias* con el mismo bloque, la más grande contiene todo lo de la más pequeña y más porque es más grande.

Pila LRU: tener las referencias que se van apilando y cuando una referencia ya está en la pila la saco y la pongo en el tope, entonces en el tope tengo la más recientemente usada y

al final la menos. Claramente una pila más grande contiene lo de una más chica con lo cual se ve bastante directa la propiedad de inclusión.

Políticas de escritura

Modifico un bloque y luego por políticas de reemplazo necesito desalojar el bloque modificado.

- WRITE THROUGH: actualizar la caché y memoria principal cuando hay una escritura. La contra es que la memoria principal es lenta y si las escrituras tienen que esperar a que se actualice la memoria principal tenemos una penalidad en el desempeño para las escrituras. Una posibilidad para no tener tanta penalidad es escribir en caché y escribir en un buffer y que después el *controlador de memoria* copie del buffer a la memoria principal (en paralelo) y libera a la cpu y actualiza la memoria principal. Le pasa el trabajo a otro hardware. Estos buffers implican una complejidad extra y puede que se acumulen las escrituras. Usa mucho ancho de banda.
  Lo bueno es que hay una consistencia entre la caché y memoria principal. La memoria principal siempre está actualizada. Se escribe solo la palabra que está escribiendo la cpu porque es la cpu la que tiene la dirección del dato.

- WRITE BACK: inconsistencia entre memoria y caché porque se modifica la caché. Se escribe solo en caché. Para recordar que el bloque se modificó se incluye un bit de dirty porque tiene que actualizarse en algún momento. Ese momento es antes de que el bloque sea desalojado porque hay un miss y el bloque que viene a cache va a reemplazar al bloque modificado. Estas escrituras son rápidas pero los misses producen dos penalidades distintas: si el miss se produce sobre un bloque no modificado, es normal. Si fue modificado, se actualiza el bloque y luego se trae el otro. Antes de desalojar actualizo, después piso con el bloque que quiero traer.
  Se actualiza el bloque completo porque no se pone un bit de dirty individual por palabra.

Miss de escritura

Cuando hay un miss de escritura y solo para el caso de write through podría llegar a evaluar no traer el bloque de la memoria principal a la memoria caché porque como siempre escribo en memoria principal y caché, podría decidir escribir directo en memoria principal.
WRITE ALLOCATE: traigo el bloque a cache.
NO WRITE ALLOCATE: no traigo el bloque a cache.

Ecuación de desempeño de CPU con Memoria caché

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

Agregamos la cantidad de ciclos de parada del cpu (ciclos de stall) por instrucción. ¿Por qué? porque la caché va a producir misses *donde la cpu va a tener que parar*, para que se escriba el bloque en caché por ej. Se agrega un promedio de ciclos por instrucción le agregamos al cpi por situaciones de jerarquía de memorias.

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

Otra medida de desempeño

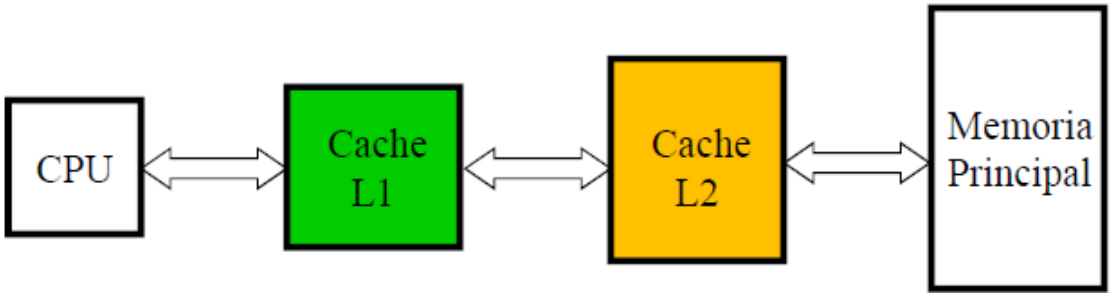
$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Reducir el Tiempo Promedio de Acceso a Memoria lleva a una reducción prácticamente igual en el tiempo de ejecución.

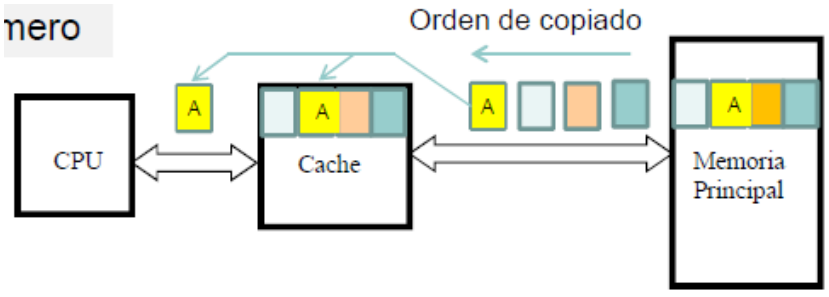
La penalidad de miss engloba los eventos que producen que se pierda tiempo para cumplir con algo del funcionamiento. Si quiero mejorar el tiempo de acceso a memoria que es equivalente a mejorar el tiempo de ejecución del programa, entonces podemos ver cómo mejorar el tiempo de hit, la tasa de miss y la penalidad de miss. Generalmente mejorar uno empeora los otros.

Reducción de la penalidad de miss

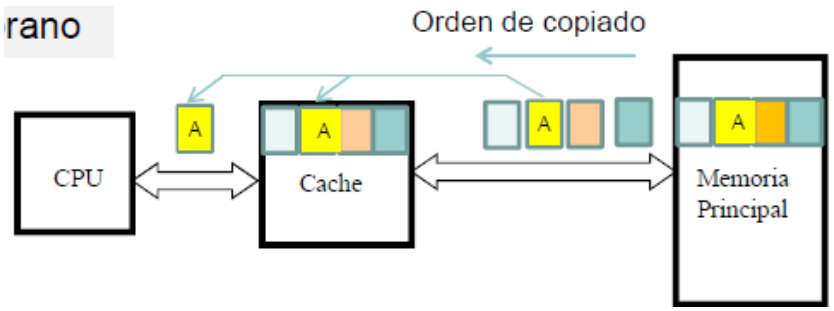
- CACHE MULTINIVEL: la L1 cuando pide un bloque y le baja la penalidad de miss si tengo una L2. Las referencias que llegan a L2 son referencias difíciles. Para tener una tasa de hit razonable la L2 tiene que ser aprox 10 veces el tamaño de L1, sino serían todos misses. El controlador de memoria sabe donde buscar los datos.



- Optimizaciones para adelantar el envío del dato a CPU
  - PALABRA CRÍTICA PRIMERO: primero se copia la palabra que se necesita y luego lo demás.



  - RECOMENZAR TEMPRANO: la búsqueda se hace normal pero cuando le toca el turno a lo que está pidiendo la cpu que también se lo copie a cpu, entonces así la cpu se libera y lo demás se sigue copiando en background.



- CACHE DE VÍCTIMAS: los bloques que son desalojados de la caché son alojados en esta caché, mientras se procede a ir a buscar en memoria, en *paralelo* se busca en esta caché de víctimas. También tiene su política de reemplazo.
- Prioridad a lecturas sobre las escrituras
- MERGING WRITE BUFFERS: combina escrituras que tengan direcciones destino consecutivas en una sola entrada del buffer. Sino, ocuparían entradas separadas lo que podría incrementar las chances de tener stalls en el pipeline.

Clasificación de misses en memoria caché

Modelo de las “tres C”

Clasifica los misses en tres tipos

- Obligatorios: miss en la caché infinita (debido a accesos de primera vez).
- Capacidad: misses que no son obligatorios y son misses en la caché FA LRU de igual tamaño que la caché bajo estudio.
- Conflicto: el resto.

No clasifica individualmente, tiene anomalías (las caché que usa son (1) FA LRU infinita y (2) FA LRU del mismo tamaño que la caché bajo estudio). Ignora las políticas de reemplazo.

Se modificó por el modelo "D3C" que no tiene anomalías.

Modelo D3C

Este nuevo modelo se basa en las distancias LRU D. B es la cantidad de bloques en la memoria caché. Es un modelo determinístico.

- Obligatorios: D no computable. Si es un miss en la caché bajo estudio y en la caché infinita.
- Capacidad: D >= B. Miss no obligatorio que es miss en la caché FA del mismo tamaño y en la caché bajo estudio.
- Conflicto: D < B.

Simulación manejadas por traza: consta en instrumentar a una compu para que modifique los códigos del programa que se ejecutan para que durante la ejecución *generen archivos que contienen las referencias a memoria*. Y luego esos archivos altamente comprimidos con técnicas especiales que aprovechan del saber del tipo de dato que se está comprimiendo.

Los usan como entradas a los simuladores *para probar distintas organizaciones de memoria caché*.

Modelo del lazo simple
Quiero estudiar una memoria caché para ver cómo me afecta el desempeño cambiar los parámetros.
Es un modelo de referencias a memoria simple. Los programas tienen lazos. Puedo ver cómo se comportan las organizaciones de memoria con respecto al lazo.
Consta de referenciar una zona de memoria de manera cíclica formando un lazo. Se accede a bloques consecutivos en memoria.

Reducción de la tasa de miss
Aumentar el tamaño del bloque en una cache de tamaño fijo captura más localidad espacial. Traes más instrucciones. Entonces me disminuye la localidad temporal. Puedo ganar localidad temporal y perder espacial y viceversa.

Reducción de los ciclos de stall vía paralelismo

- Caches no bloqueantes
  Cuando hay un miss, la cpu puede seguir ejecutando otras instrucciones mientras queda pendiente la ejecución que está con el miss. (cpu moderna para que el código se ejecute fuera de orden, no es el caso de mips).

- Hardware prefetching
  Trata de predecir qué va a necesitar el programa y lo trae de forma automática. ¿Dónde lo guardo? Generalmente en caché. Pero por ahí saco un bloque que me sirve. Otra alternativa es meter otro buffer aparte.

- Software prefetching
  Por programa el programador lo indica con directivas e instrucciones.

Reducción del tiempo de hit
Caches más simples y pequeñas, menos asociativas. Pero esto va en contra de la estadística de la tasa de hit. La mejora en un parámetro suele empeorar otro. Hay que estar viendo el equilibrio. Evitar la traducción de páginas, cache pipeline.

Memoria virtual

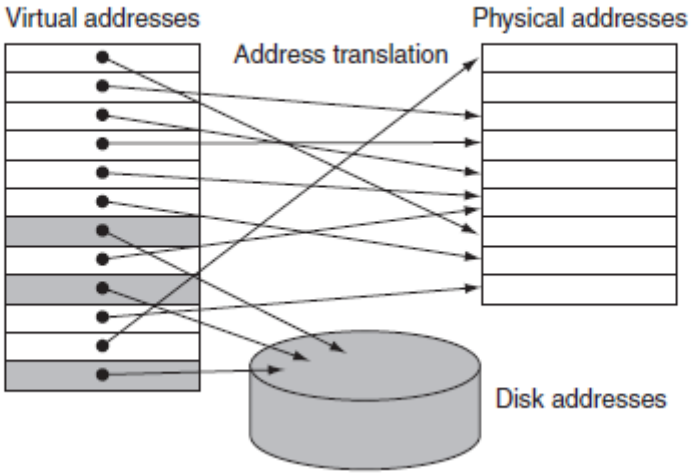
Antecedentes

- overlays: La administración del uso de memoria recaia en el programador y no en el sistema operativo, cuando los programas superan el tamaño de memoria física.
  Problema: falta de memoria.
- registro cerco: Contiene un valor que es sumado a las direcciones generadas por el programa para no poder acceder a cierta parte de la memoria. Y se le pone una dirección límite para que no se exceda.
  Problema: protección.

Desde el punto de vista del programa, los dos sistemas memoria caché y memoria virtual son transparentes.

La memoria virtual es una abstracción. El software ve que puede usar del byte 0 a x, todo contiguo. El hardware se encarga de ubicarlo en ram y demás.
El sistema operativo es quien dice que pagina sale de memoria física (la política de reemplazo se implementa por software). Hasta se puede hacer una detección de trashing.

Memoria virtual paginada
La idea es dividir a la memoria en bloques de tamaño fijo que se denominan páginas. La memoria física también es un espacio lineal de bytes pero real.
Lo que se divide en páginas es el espacio de direcciones, el cual es un concepto, abstracción. Este es un conjunto lineal de bytes continuo de una sola dimensión. Este espacio se divide en páginas. Mediante un mecanismo de asignación estas páginas, que desde la app (software) se llaman páginas virtuales, van a ser mapeadas por un mecanismo a memoria física. A la memoria física también se la divide en bloques de igual tamaño pero desde el lado de memoria física se los llama marco de página. La política de ubicación es totalmente asociativa (no hay restricciones de ubicación).



El sistema de memoria virtual paginado hace jugar al disco como parte del sistema de memoria. La zona de disco destinada al sistema de memoria virtual se llama área de swap. Las páginas pueden estar en memoria física o área de swap. Están en área de swap cuando no entran en memoria física.
El área de swap está relacionada con el tamaño de la memoria física, también se configura. Suele ser del 50% del tamaño de la memoria física. No es un reemplazo sin consecuencias esta área de swap. Se lo tiene que pensar como un recurso de memoria de emergencia **para poder ejecutar programas que no entran en memoria**. Es tan alta la penalidad que debe hacerse con baja frecuencia.
Cuando en una pc esta la memoria llena y se está ejecutando procesos y empieza el proceso de swapping de manera frecuente, se llama trashing y la pc deja de andar (se empiezan a copiar las páginas).
Se tiene otro tipo de hit y miss al buscar una instrucción o dato. Si está la página en memoria física hay hit y si está en área de swap es un fallo de página. La penalidad es la de acceder a un dispositivo de entrada/salida.

Los espacios de direcciones virtuales están dados por la capacidad de direccionamiento del procesador.

Y el tamaño de la memoria física puede estar determinado por la implementación de la cantidad de la memoria máxima de esa computadora. Usualmente desde hace mucho como los procesadores saltaron de capacidad de direccionamiento en escalones muy grandes, el espacio de direccionamiento virtual puede ser muy grande.
Lo importante es que no se está implementando todo eso, sino que se está implementando a nivel de página.
La memoria virtual paginada es totalmente transparente al software. La cpu va a generar direcciones y va a obtener como respuesta instrucciones o datos que está pidiendo y no se va a enterar que está en otras direcciones.

Proceso de traducción

Al dividir los espacios de memoria en bloques, vamos a tener un número de bloque y un offset dentro del bloque.
Tenemos un número de página virtual y un offset de página. El offset de página es el mismo en una dirección virtual y en una página física. No hay un reordenamiento de bytes dentro de la página. Los bits de offset de página no se traducen, son los mismos. Vía la traducción se obtiene el número de marco de página a partir del número de página virtual.

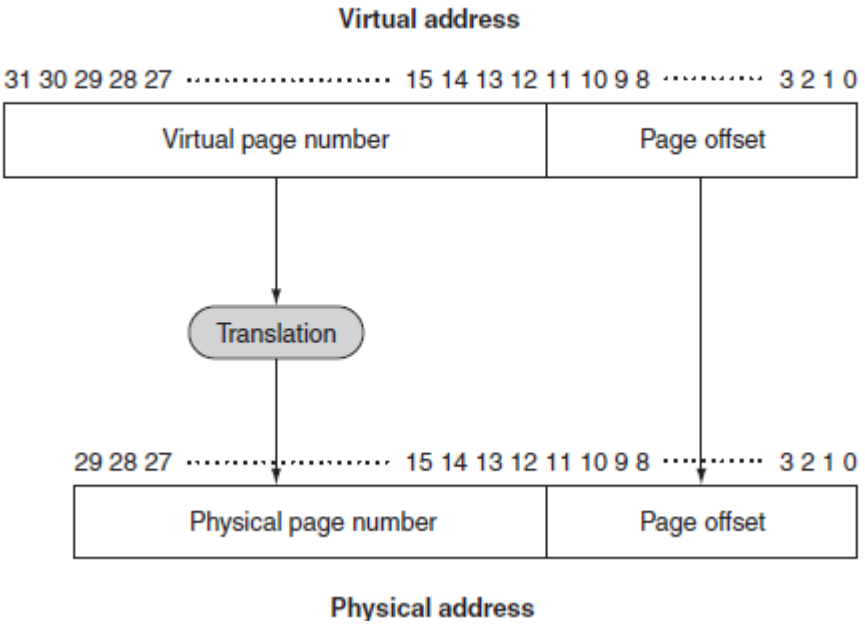
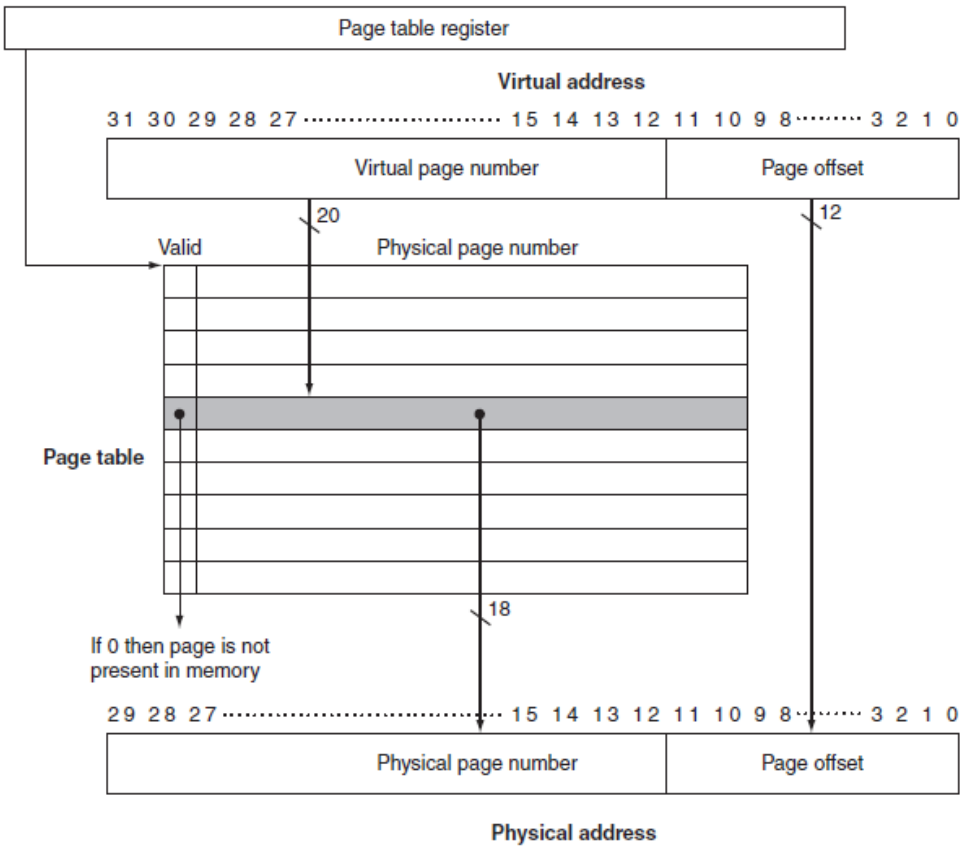


Tabla de traducción

Se entra en la página con el número de página virtual y se lee en la entrada de la tabla el número de marco de página, se concatenan los bits de offset de páginas y obtengo la dirección física para ir a la memoria principal.

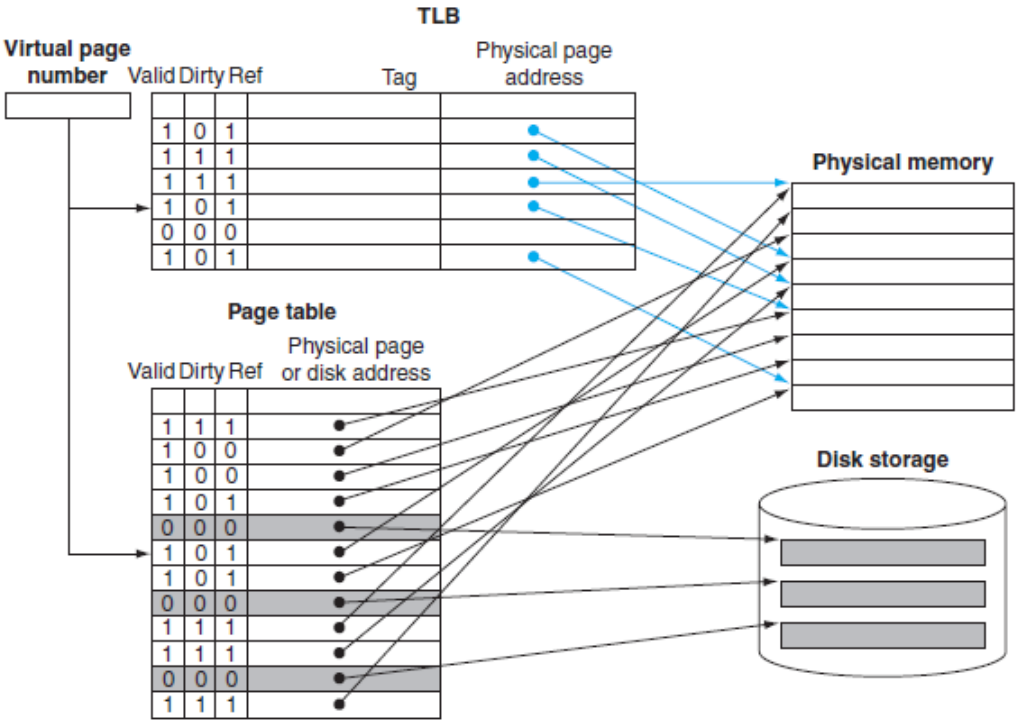


El mecanismo por indexado directo esta bueno para la explicación teórica pero no para la práctica. Otra es mediante una estructura de directorios. Otros son con algoritmos de hashing.

Proceso de traducción: podemos contar con un registro que contenga la dirección de comienzo de la tabla.
Las páginas pueden o no estar presentes en la memoria física, o podrían estar en el área de swap. Tenemos un bit de validez para ver si la traducción es válida porque la página está en memoria física o no. Esta información que es necesaria para el funcionamiento de todo esto está guardada en la tabla de páginas además de la traducción.

<u>Técnica de traducción rápida: TLB</u>
Por cada direccionamiento (para cada fetch, load y store) en definitiva tengo que acceder a memoria para buscar esa traducción. Pase a tener dos accesos a memoria por cada acceso. Esto crea un problema en el desempeño enorme. *Para evitar ir a la tabla*, para evitar ir a memoria, se utiliza una caché pequeña llamada TLB que contiene la traducción de las últimas páginas o conjunto de páginas que esté utilizando el programa. Cuando la cpu genera direcciones para ir a buscar las instrucciones o los datos, se direccionan a la tlb (el hardware) para ver si tiene la traducción (y es un hit en la tlb) con lo cual me evito ir a la tabla para hacer la traducción y lo obtengo en un tiempo rápido, implementada en la cpu, en el propio integrado. Se genera la traducción de una manera rápida.
Cuando la dirección que genera la cpu no está en la tlb, tenemos un miss en tlb. Un miss en tlb tiene una penalidad baja. Si es una penalidad alta un miss en la tabla de páginas porque es un page fault.

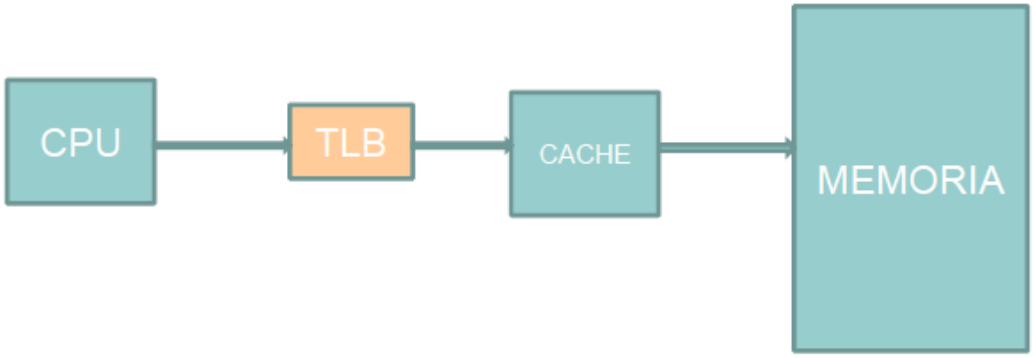
La tlb contiene el mismo tipo de entrada que está en la tabla. El bit de dirty es importante porque cuando una página va a ser desalojada de la memoria física, si no fue modificada no hace falta actualizarla y copiarla en el área de swap. Nos ahorramos la penalidad de actualizar la página. Solo copiamos la página que está provocando el page fault desde swap a memoria física.



Número de proceso: indica quien es el que está generando esta dirección.
Los bits de protección: asn (address space number) fundamentales. Es lo que los sisop necesitan para darle protección a los programas entre sí y el sistema operativo del programa. Cuando un programa se ejecuta con su num de proceso o su num de espacio de direcciones, va a acceder a alguna dirección y entonces el hardware lo que va a hacer es chequear estos bits de protección si el proceso tiene el permiso para acceder a esa página. Permiso de lectura, lectura y escritura, permiso para ejecutar.

Caches direccionadas por direcciones físicas

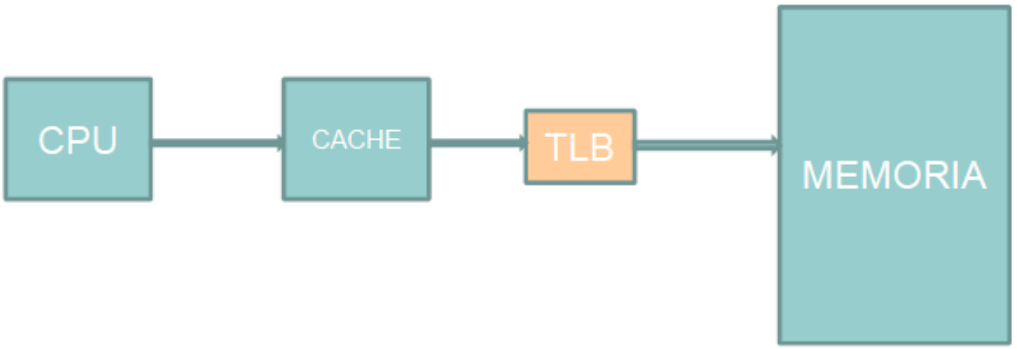
La cpu siempre va a generar direcciones virtuales y necesito traducirlas para ir a mi sistema de memoria. Entonces viene la búsqueda en tlb. Si la traducción está, ok, y sino se actualiza la tlb. Con la dirección física se accede al sistema de memoria que tiene caché.
Contra: el acceso a tlb por mas que sea más rápido que ir a la tabla, es una penalidad en tiempo porque no deja de ser otra caché. Podría llegar a ser un problema. Estiro el ciclo de reloj o busco otro mecanismo.



Caso especial: los bits de offset no se traducen. Podría hacer el indexado de la caché (con los bits que permanecen igual) en paralelo con la búsqueda de la traducción. Entonces, se solapan la traducción con el acceso a la caché. Con lo cual la caché entrega su tag y la tlb entrega su número de página y hace la comparación sin hacerlo en un paso tras de otro. Tiene su limitación por sus números. Esquema híbrido (index virtual y tag físico).

Caches direccionadas por direcciones virtuales

Las direcciones que salen de la cpu van directamente a la caché sin traducir. Solo en el caso en que haya miss en caché vas a lo que sigue en el sistema de memoria. Como máximo es la L1 que tiene este esquema. Si hay miss en la L1 vas a la TLB. Entonces, la ventaja que tiene es que me despreocupo de traducir para acceder a cache. La contra que tiene es que podría haber un desperdicio de espacio en caché por los alias, que es que con distintos nombres me estoy refiriendo a la misma posición de memoria. Y los distintos nombres implican por ahí distintas direcciones lógicas. Entonces como estoy direccionando a la cache por direcciones lógicas podría tener copiado lo mismo en la cache con diferente nombre.



Memoria virtual segmentada

En memoria virtual paginada dividimos a los espacios de direcciones virtual y físico en bloques del mismo tamaño pero en la segmentada estos bloques son de tamaño variable y se lo llama segmentos. La idea es tener por un mecanismo de traducción tener una asignación de los segmentos virtuales donde se ubican en la memoria física. Dado que los segmentos son de tamaño variable, al offset de segmento se le tiene que dar una palabra completa, porque puede llegar a tener que especificar a todos los bits de la memoria para especificar los bits dentro del segmento. Por esto se compone de dos partes: número de segmento y un offset dentro del segmento.
Si se tiene una máquina con memoria virtual segmentada es *visible al software*.
El más limpio es el paginado.
(puede haber una combinación de los dos)

Paginado vs segmentado
External fragmentation: espacio que quedan entre segmentos en memoria física que no los puede aprovechar nadie porque no entra ningún otro segmento en esos espacios.

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

Tecnologías de memoria

Memoria ram dinámica y memoria ram estática. Ambas memorias son volátiles.

Dinámica: implementa la menor cantidad de hardware para guardar un bit. Es un capacitor con un transistor que funciona con una llave para acceder. Se busca la mayor cantidad de bits para almacenar para tener la menor cantidad de costo por bit por eso se elige. Esto lleva a que la carga del capacitor se vaya perdiendo porque tiene corrientes de fuga. Las celdas son tan chiquitas que se van descargando. Por lo tanto hay que refrescar la carga de esos bits.
Hay mucha circuiteria para que la información no se pierda. Se usa para la memoria principal en los módulos dimm. Al ser más complicada la ram dinámica también es más lenta por el cuidado que hay que tener en el acceso.

Estática: dos inversores enganchados con su transistores de accesos para grabar un cero o uno y es mayor la cantidad de hardware que se requiere por bit. Se utiliza para las caches de todos los niveles y la tlb y están integrados en el procesador.

DRAM convencional
El parámetro RAS es la latencia (row access strobe). La estructura de almacenamiento de la memoria se ordena de manera matricial para disminuir el tamaño de los decodificadores. Si se tiene un arreglo tendría muchas líneas. Con una de 1MB si es lineal tengo 1MB de líneas, si lo divido en matriz lo divido en 1KB de fila y 1KB de columna. Se decodifica la fila y se decodifica la columna.
Ventaja: tengo la dirección en dos partes: direc de fila y direc de columna.

DRAM modo página rápida
Deja fijo el número de fila y lo deja un tiempo largo. Cambia el número de columna y lo deja un tiempo largo. Luego para buscar otro dato deja el número de fila fijo también y cambia la columna.

Si guardo datos consecutivos en columnas consecutivas me viene bien porque es normal que la cpu pida datos consecutivos, la caché pide bloques. La tasa de acceso a datos mejora.

SRAM

Para qué tener un cas y un número de columna cada vez, y no asumimos que el número de columna que viene es la siguiente, le sumamos uno internamente en el chip, y con un juego inicial de fila y columna.
Con fila y columna accedo al primer dato y después vía una señal de reloj que cambie de columna accede a los otros datos consecutivos.

DDR: accede a datos a una tasa doble que una normal.

Arquitectura ram ddr simplificada

La estructura del chip está compuesta por 4 matrices (banco) y pasamos a una estructura 3D. Señales: RAS, CAS, WE (write enable), CS (chip select), etc.

Si tengo dos memorias ddr equivalentes, me quedo con la que tiene el primer parámetro más chico de timing.

Arquitectura módulo DIMM

Módulo dimm más simple: cada uno tiene una interfaz de info de 8 bytes que típicamente cada byte lo implementa un chip.
ecc: código de corrección de error.
A medida de que se van agregando conexiones a la salida de una compuerta, la señal se deteriora. Cuando hay poca carga el flanco es bastante vertical.
Para evitar esto se pone un registro a la entrada de chip. Entonces cada entrada se carga con una compuerta.
La ram es susceptible a errores: cambia el valor de la celda. Por rayos cósmicos que tienen una probabilidad dada. Esa probabilidad depende de la cantidad de RAM que tenes y el tiempo que tiene prendida.
Los servidores: están prendidas mucho tiempo y tienen mucha ram. Por eso tienen que tener la capacidad de usar módulos de memoria que contengan un bit adicional que sea capaz de detectar y corregir errores (errores soft). En las pc normales no es tan común.
Este esquema entonces no es necesario para las computadoras normales. Tienen un dimm más que es para verificar errores.

Banco único y banco doble

Para aumentar la capacidad del módulo dimm se puede poner chips de los dos lados con lo cual tenes doble banco a nivel del dimm. Y estos son más densos.

Arquitectura del sistema

El sistema de memoria se conectaba a la cpu vía un chip que se llamaba north bridge (chipset) (viejo). Hoy en día esta electronica esta en el cpu, la memoria se conecta directo a la cpu también con el video y los periféricos se conectan con el chipset.

Arquitectura del sistema, memoria de doble canal
Surge la idea de utilizar dos canales de memoria de manera de poder accederlos de manera paralela. Uso en paralelo dos módulos de memoria, y aumento el ancho de banda. Estoy intercalando el acceso a los dos canales. Reduce la latencia para la CPU para la recepción de los datos de memoria. Palabras consecutivas se guardan en canales consecutivos.

Camino de datos monociclo

Se va a suponer que se implementan pocas instrucciones del mips que establecen el camino de datos para casi todas las instrucciones. Cuando va a ejecutar las instrucciones el cpu tiene que realizar una determinada cantidad de trabajo, que se puede considerar como pasos y estos pasos son arbitrarios en el sentido de cómo agrupo el trabajo que hago en cada paso, y lo que es cierto tengo que hacerlo todo y depende de cada instrucción.
Metodología de clocking: define cuando las señales se leen y cuando se escriben. La metodología de diseño de sistema de máquina de estados finitos de una cpu es sincrónica, y los distintos elementos están gobernados por una señal de reloj y los cambios de los elementos de estado (elementos de memoria de la pc). Van a ser leídos/escritos en un determinado flanco de la señal de reloj. La salida combinacional que es a partir de un estado puede volver a ser entrada de un mismo elemento de memoria porque la metodología de clocking evita que haya un bucle o realimentación de bucle.
Memoria de instrucciones: va a haber una memoria para las instrucciones y una para los datos.

Esta organización es la de una implementación monociclo, que ejecuta una instrucción en un único ciclo de reloj, lo que dice que *no se puede reutilizar ningún elemento del hardware*. Por eso hay memoria separada. Para actualizar el pc hay un sumador, no se está usando la ALU ppal, porque esta se va a reservar para el cálculo principal de la instrucción. No puedo en un único ciclo actualizar el pc y también hacer el cálculo ppal de la instrucción, con lo cual los recursos van a estar replicados y ajustados a lo mínimo para lo que se la utiliza porque no se pueden reusar.
El fetch de la instrucción involucra leer la instrucción de memoria, actualizar el pc.

Register file: va a necesitar tener la cantidad de puertos para poder ejecutar la instrucción en un ciclo, con un puerto de lectura y uno de escritura. Tenemos entonces un puerto de lectura con un especificador de uno de los fuentes en 5 bits lo voy a buscar y me va a devolver el contenido en 32 bits. Una escritura que eventualmente se puede necesitar. Tiene una señal de write explícita porque no todas tienen registros destino (jump por ejemplo).
Para la parte de lectura, con 5 bits elegimos el registro que queremos leer por ejemplo en uno de los puertos, entonces estos 5 bits van a ser las líneas de selección de un multiplexor de 32 a uno. Los registros son de 32 bits. Los dos registros de lectura se seleccionan en paralelo.
Para la escritura necesitamos un deco. Cuando llegue la señal de write con su flanco, se selecciona el registro que se escribe.

Decodificación de las instrucciones: la unidad de control tiene que recibir el opcode y los bits de función en el caso de las instrucciones del tipo r. La decodificación se puede hacer una

vez que la instrucción llegó a la cpu después de leerla de memoria. Una vez que está en la cpu, la podemos decodificar y leer los operandos fuente de la instrucción. Muchas cosas que se van a hacer son especulativas, se van a hacer aun cuando todavía no se sepa lo que se va a hacer. La decodificación y la lectura de los fuentes se va a hacer al mismo tiempo.

Va a haber caminos más cortos y caminos más largos para atravesar los elementos del hardware porque hay un retraso en algunos de estos elementos debido a los accesos. Entonces como la máquina va a ser de un solo ciclo y este es fijo, el camino más largo de ejecución tiene que ser tal que le dé tiempo a la más larga de ejecutarse para que las señales se establezcan.
Hay instrucciones donde el rd es el rt y otros en el que rd es rd. Entonces el mux de write address selecciona según si es de ALU o no.

Este camino de datos nadie lo va a implementar porque el hardware es complejo porque necesita duplicar recursos, necesita memorias separadas, sumadores repetidos. También es lenta, porque el periodo de reloj es lento porque tiene que adaptarse a la instrucción más lenta. Las instrucciones que se ejecutarán más rápido se ejecutan lentamente. El load seguro es la más lenta porque usa todos los recursos del camino de datos.

Camino de datos multiciclo cableado

Los buses únicos son de bajo desempeño porque generan un cuello de botella. Se va a tratar de reusar los elementos del camino de datos. Va a tener una implementación con la menor cantidad de recursos posibles, pero va a aparecer una máquina de estados que va a ser difícil de seguir porque se ejecutan pasos.

Aparecen registros de la implementación. Como una instrucción se va a ejecutar en distintos ciclos, necesito tener un lugar de almacenamiento interno para que el dato se pueda utilizar más adelante. Es almacenamiento temporal. Estos registros son *transparentes a la implementación de la arquitectura* porque solo son necesarios en esta implementación.

Los datos usados por instrucciones subsecuentes se almacenan en registros visibles al programador. Estos registros de implementación se necesitan para llevar información de un paso a otro siempre dentro de una instrucción.

Ventaja: usa más eficientemente el clock. Este se adapta al paso más lento. También permite que las unidades funcionales se reutilicen mientras que sea en pasos diferentes.
Requiere los reg internos, más multiplexores y una máquina de estado finitos más compleja.
La unidad de control es más compleja.

Pipeline

Idea del pipeline: aumentar la frecuencia del ciclo de reloj.
Pipeline: tubería (más parecido a una línea de montaje en realidad).
Registros interetapa: hacen que la instrucción se ejecute en pasos.
Si una instrucción tiene que volver para atrás se choca con otra instrucción.

Con cada clock el registro interetapa copia lo que está a la izquierda que es necesario para la derecha, y luego escribe en la derecha. Estos registros interetapa tienen los nombres de la etapa de la izquierda y de la derecha. Y la cantidad de entradas es a medida.
Cuando un elemento de memoria es leído se pinta la mitad derecha, y cuando es escrito se pinta la mitad izquierda y los circuitos combinacionales se pintan completos.
Diagramas pipeline: de izquierda a derecha tenemos el paso del tiempo en ciclos de reloj. Es un diagrama de transición. De arriba a abajo, las instrucciones ejecutadas dinámicamente.
En el pipeline de 5 etapas, hay un paralelismo que es un solapamiento temporal en la ejecución de las instrucciones.
El pipeline se puede aplicar en cualquier elemento de la cpu (ALU por ejemplo).
La unidad de control genera todas las señales, se generan en la etapa de id y se van copiando en los registros interetapa.
En el mejor de los casos estamos acelerando por 5 veces.
Puede haber una dependencia entre las instrucciones o competencias por recursos.

Si no están listos los resultados, tenemos otro problema.
Soluciones:

- parar el pipeline (ciclos de stall = bubble)

La unidad de control detecta una dependencia con un registro que todavía no está actualizado entonces para la ejecución de esa instrucción hasta que se actualice. Estás pagando un hardware más complejo porque el programador se desentiende y el hardware tiene que verificarlo.

- por software (NOP)

En este caso se desentiende el hardware. Pero el programa es más grande, el ejecutable ocupa más lugar y la caché se hace más ineficiente porque está capturando nops que no hacen nada.

Estas dos soluciones son iguales en términos de espera.

- forwarding

En vez de esperar a escribir en el ciclo correspondiente, se conecta la salida de la ALU con otras entradas.
La unidad de forwarding va leyendo los destinos y fuentes de las instrucciones para detectar dependencias y settear a los mux de manera correcta para que a la ALU le llegue el dato correcto de acuerdo a la lógica del programa.
No es aplicable al caso de load delay slot, es como que necesito un dato del futuro. No puedo evitar perder este ciclo.

Pipeline: taxonomía de los riesgos

- Riesgos estructurales
  Las instrucciones no se pueden ejecutar por no haber suficientes recursos: dos o más instrucciones compiten por el mismo recurso. Generalmente la que lo va a usar es la que está más avanzada en el pipeline y la otra tiene que esperar. Se soluciona agregando un ciclo de stall.

- memoria unificada:
  Tener una caché separada, por un lado para el programa y otro para datos es muy conveniente desde el punto de vista del desempeño porque de tenerla unificada podría haber un problema de estos.

- register file sin suficiente cantidad de puertos:
  Si un register file para una implementación no tiene la suficiente cantidad de puertos, va a haber un riesgo estructural. Las de tipo R tienen dos registros de operando y si el register file tiene un solo puerto hay que leerlos de a uno.

- unidad funcional requiere más de un ciclo:
  Hay instrucciones que son complejas y puede que necesiten del uso de una etapa por más de un ciclo. En este caso la unidad funcional no estaría implementada en sintonía con avanzar un paso. Entonces puede ser que el pipeline deba pararse.

- Riesgos de datos
  Hay una dependencia entre un dato i y un dato i + 1.
  - RAW: el riesgo es que se lea antes de tiempo, con el *dato que todavía no se actualizó*. Se resuelve con **forwarding**, el load delay slot es el que no se puede resolver.
    Load delay slot: como la etapa de memoria viene después de la ejecución, si una instrucción quiere usar un dato que se está trayendo de memoria por la instrucción anterior no hay forma de que esté en cpu al comienzo del ciclo. Es porque hay un orden temporal (caso de load y después add de ese dato que cargue).
    Se solucionan con forwarding, o stalls/NOPs si es el caso del load delay slot
  - WAR: el problema sería que la i + 1 se adelante antes de que la i lea. *Que la i lea un valor del futuro*, el valor que lee es un valor calculado por una instrucción posterior.
  - WAW: por algún motivo el orden en el que queda la variable compartida queda invertido. Quiero que en la variable quede lo que indique la instrucción i + 1. *Es un riesgo en una arquitectura superescalar.* Hay un único punto de escritura entonces no se desordena, por eso no es un riesgo en mips 32 con pipeline de 5 etapas.
  - RAR: no son riesgos porque son lecturas, no hay nadie que esté modificando un registro.

- Riesgos de control
  Son los que se producen cuando se ejecutan instrucciones que alteran el contenido del PC. Relacionados con branch. Son los que meten la mayor cantidad de ciclos perdidos porque el pipeline anda bien cuando todo se va ejecutando de manera secuencial.

Soluciones a los riesgos de control

- Parar el pipeline

No se hace ningún cálculo especulativo. Para el pipeline hasta resolver el salto (entra en stall, cancelando el fetch de la siguiente instrucción al decodificarse la instrucción). Se pierden 3 ciclos por cada ciclo (es mucho).

- Asumir salto no tomado
  Llega el salto y continua con el fetch de las instrucciones siguientes. Cuando llega al final del ciclo cuatro, en la etapa donde se resuelve, se continúa con la instrucción siguiente o se toma el salto.
  Si no se toma el salto, no hay pérdida de ciclos. Si el salto se toma hay que anular lo hecho y estar seguros de que no alteramos el programa. Se pierden tres ciclos cuando se toma el salto porque se estuvo ejecutando cosas que no se debían ejecutar.

  <u>Mejora: mover la señal que gobierna el multiplexor del pc a la etapa de ex</u>
  Ahora el salto se resuelve después de la etapa de ejecución. Si predigo mal solo pierdo dos ciclos. Cuando resuelvo ejercicios de calcular cuántos ciclos se pierden se debe tener en cuenta en qué etapa se resuelve el salto.

  <u>Mover aún más hacia adelante para mejorar el desempeño</u>
  Se resuelve el salto en la etapa ID. Agrego un comparador especial para los saltos (hardware adicional). Ahora pierdo un solo ciclo si el salto no se toma.

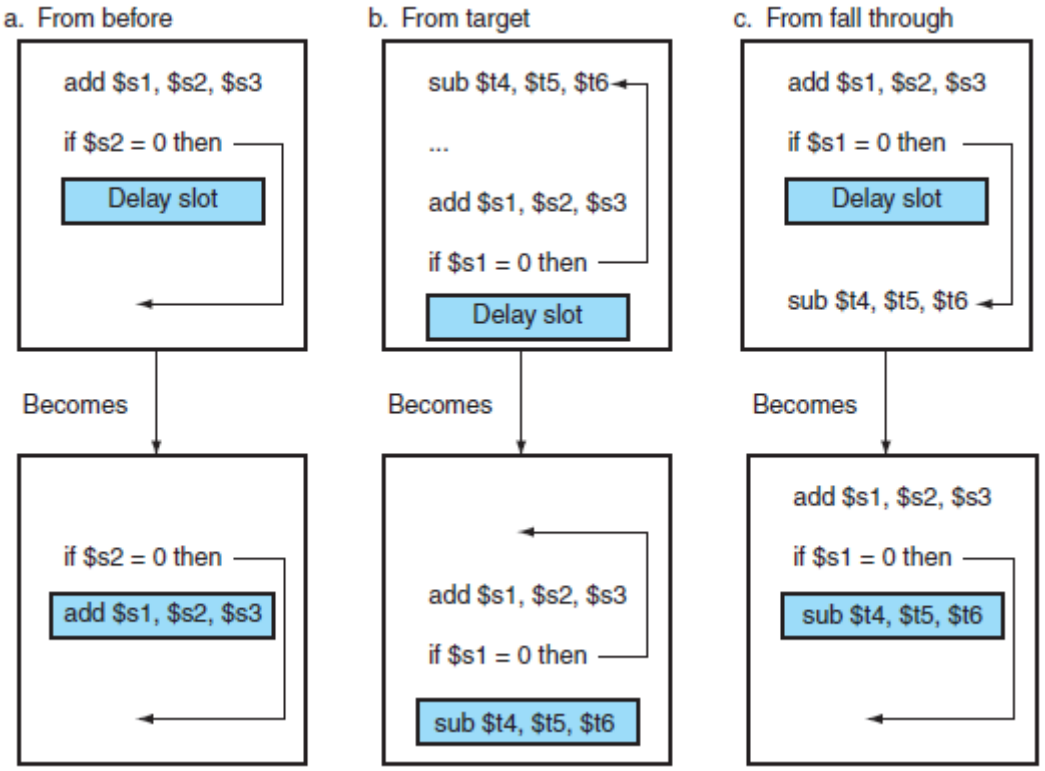
- Predicción de saltos
  Predicción estática: por hardware o software. Significa que no van a cambiar en ningún programa, predicciones fijas en cpu.

  <u>Esquema del salto demorado</u>
  El *hardware se desentiende* por completo del tema de los saltos. Cuando le llega un salto no chequea nada, como si fuera una instrucción más. **Es el software el que tiene que garantizar que eso funcione.**
  Que haya un delay slot significa que el salto se resuelve en la etapa ID.
  Si el salto se resuelve en la etapa de ejecución es de dos delay slot y si es en la etapa de memoria tres delay slot.
  Este es un esquema que se implementó, se hizo visible al software y tiene como consecuencia que después no se puede cambiar. Es el software el que tiene que garantizar que esa instrucción que va a ejecutar no altere la lógica del programa y que no se pierda el ciclo.
  El **branch delay slot** es la que está a continuación del branch.
  Como este esquema siempre ejecuta la instrucción a continuación del salto, se garantiza que no se se altere el programa poniendo una nop luego de cada salto. El beneficio de este esquema es tener un hardware más simple porque no tiene que estar detectando estos saltos, y no altera la lógica del programa. La desventaja es que vas a perder un ciclo ejecutando una NOP y el programa ocupa más memoria.
  Esta solución es software.
  Para cada salto pierdo un ciclo ejecutando una nop para no perder la lógica. Hay un segundo paso que es que en lugar de poner una nop pongo una instrucción útil que forme parte del programa, que había que ejecutar pero moviendo las instrucciones la pongo en el delay slot sin que altere la lógica, el funcionamiento del salto, y

adelantando ese cálculo simple. Dos instrucciones que son independientes se estarían ejecutando de manera desordenada y no altera la lógica del programa. Lo malo es que es visible al software.

Mover instrucciones

a. Instrucción previa al salto: Se cambia el orden de ejecución de manera que no afecta la lógica del programa. Se *demoró* la ejecución de la instrucción. La pase al delay slot.
b. del target.
c. del fall through: pones una instrucción en el delay slot que si se ejecuta y modifica un dato que no debía modificarse, luego se debe corregir eso (se debe pisar ese resultado mal calculado por otra instrucción). Si la predicción es incorrecta, si no tenes garantizado que el error se auto corrige no podes hacer el movimiento.

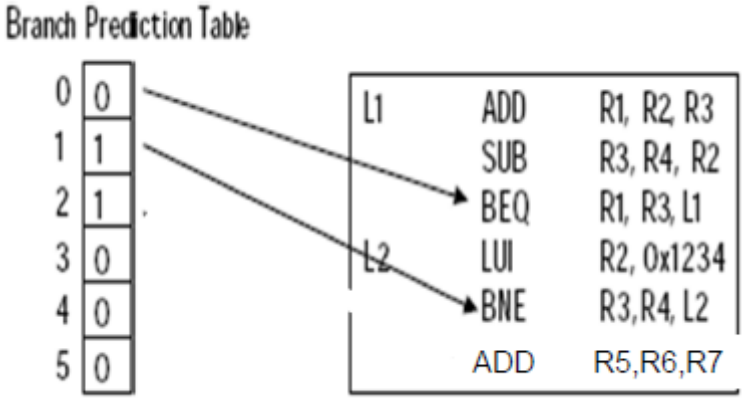


Predicción dinámica: puede cambiar con cada ejecución dinámica del salto. Porque están dentro de un loop o en una función reutilizada.

Predictor local de un bit

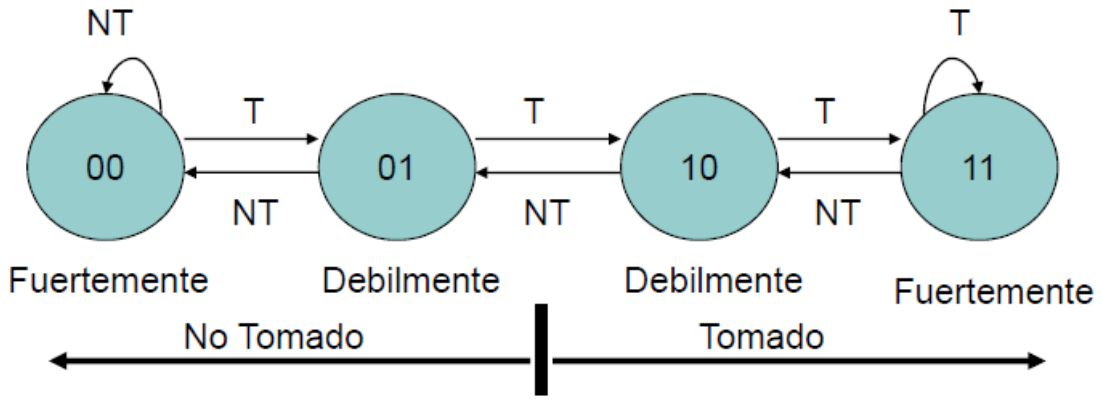
Cada salto va a tener una entrada en la tabla. Para predecir lo que hago es ir a la tabla para ver qué entrada le toca a este salto que se tiene que ejecutar. Si leo un 1, predigo que el salto se toma y busco la instrucción del target. Si lo que leo es un cero, sigo con instrucciones del fall through.
Después hay que verificar que el salto efectivamente se toma o no se toma. Por otro lado, el procesador sigue evaluando la condición y determina si había que saltar o no. Si no se toma se debe garantizar que no se altere la lógica del estado del programa.

Una vez que el salto ya se evaluo, lo que tengo que hacer es actualizar la tabla para que la prox que le toque tomar este salto, tome una decisión basada en el comportamiento que tuvo.
Que la tabla sea de un bit y que se actualice así, quiere decir que el salto se va a comportar como la última vez. Si el salto se toma, se actualiza con 1, sino con 0.
Los saltos predichos en bucles van a tener una buena tasa de aciertos.



Predictor local de dos bits
Ya tenemos un diagrama de estado para hacer la predicción. Tengo 4 estados posibles.
00: va a predecir que no se va a tomar fuertemente.
01: no se va a tomar débilmente.
10: predice tomado débilmente.
11: predice tomado fuertemente.

Clipping digital: cuando te pasas del máximo y te quedas en el techo (cuando estoy en 00 se mantiene ahí si no se toma y lo mismo para 11).



Interferencia de predictores
Si quiero que cada branch tenga una entrada exclusiva en la tabla, debería tener una tabla por cada bits tenga el pc (2^bits) (seria una tabla bastante grande).
Pueden haber dos branchs con los bits que se utilizan para acceder a la tabla. Con una tabla de 1KB es bastante buena para que no ocurra.
No siempre es algo negativo. Hay que ver la estadística.
Si ambos branch tienen la misma lógica, l**a interferencia es neutra**.
Si su ejecución es alternada, es **negativa.** Es tipo trashing.
Si un salto estableció la predicción para la siguiente y acierto gracias a otro branch,

es una **interferencia positiva**.

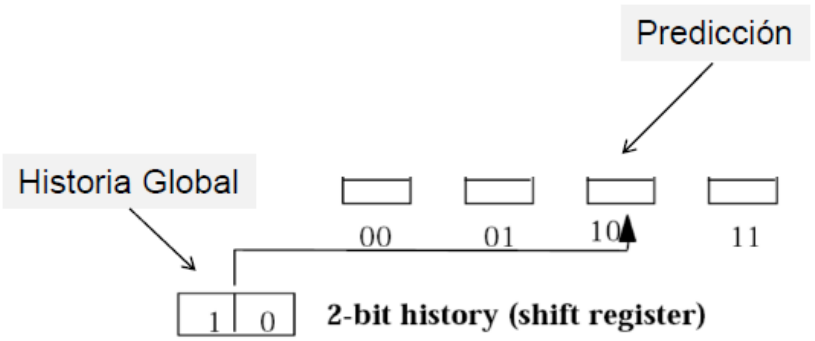
Predictor global

Voy a predecir en base al resultado de los últimos n saltos dinámicamente ejecutados, no importa cuales. Con los 0 y 1 de esos últimos n saltos voy a indexar una tabla.

Predecir con dos bits de historia global: voy guardando el resultado de los saltos en el shift register. Como hay 10, entro a la entrada 10 y leo la predicción, y luego actualizo.

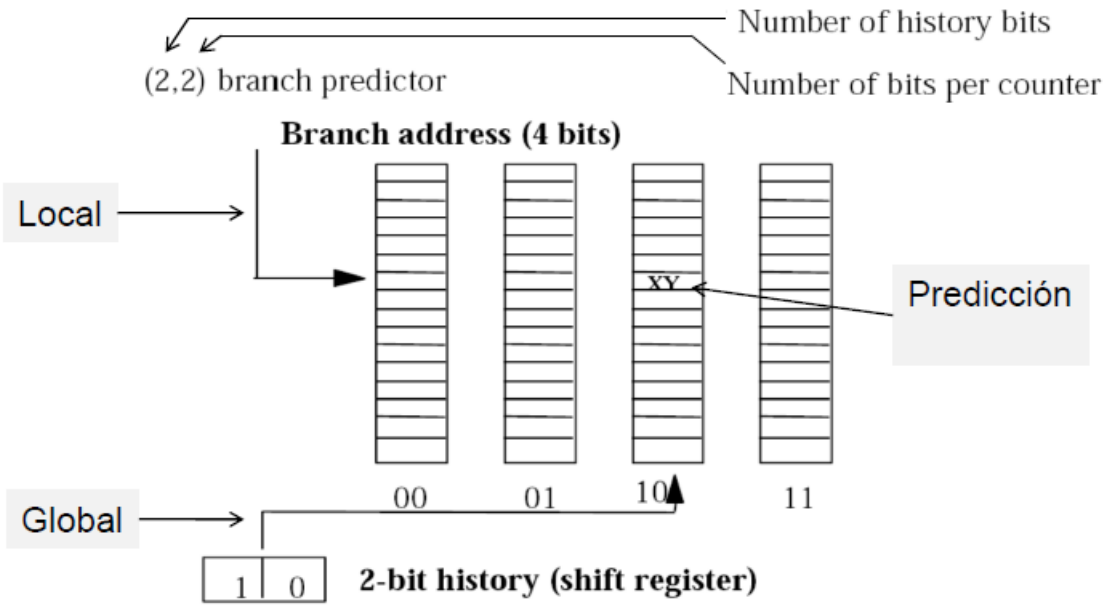
Al actualizar, el resultado de la predicción se actualiza la entrada y luego el shift register (estos pasos son muy importantes).

No es tan bueno como el local. Se puede usar un combinado (local y global).



Predictores de dos niveles: historia global + historia local

En este ejemplo hay 2 bits de historia global y dos de historia local. Con la historia global elijo una tabla completa.



Localmente indexo con los n bits más bajos del pc para este branch con el XY historia en particular elijo la predicción. Después actualizo primero el xy y luego el shift register.

- BTB: branch target buffer

Este esquema que necesitan los procesadores que tienen un pipeline profundo un paralelismo a nivel de instrucciones como los superescalares.
Es una **caché para predecir saltos**. Lo que guardo en el área de TAG son los pc completos de los branches que voy a predecir tomados. Entonces cuando el procesador está buscando una instrucción, cuando se busca la instrucción se compara la instrucción que está buscando el procesador con los pcs almacenados en esta caché.
En la parte de data está el target.
En este esquema si busco un branch que predigo no tomado, es un miss. Miss en la BTB es PC + 4. Un hit en la BTB procedo a saltar.
Los pcs que se van a guardar son solo los de los branch. Si mis predictores me dicen que no va a saltar, no lo guardo en la BTB.

Eliminación de saltos
Si los saltos son complicados, se puede minimizar la cantidad. Instrucciones de aritmética de ejecución condicional (las tiene ARM). Si se cumple la condición, hacer la aritmética. Si no se cumple, no hagas nada. (por ejemplo tengo un if que de ser true, me pone una variable en 1).
Estaría reemplazando una de salto y una de aritmética por una sola, que tiene la parte del cálculo condicional a que se de la condición a evaluar. El pc siempre va a ser PC + 4. El salto desaparece porque no cambia el valor a uno que no sea pc + 4.

Desenrollado de lazos
Para acelerar la ejecución del programa. Reducción del número de saltos: reducir el número de bucles.

Hiperpipeline y superescalares

Extensión del pipeline para manejar operaciones multiciclos: unpipelined
Van a haber elementos hardware que no se va a ejecutar en un ciclo. Es común más que nada en la ALU que tiene unidades funcionales separadas.
Algunas unidades van a requerir más de un ciclo cuando les llegue una operación a alguna de ellas, con lo cual *hay un riesgo estructural* y no es bueno para el desempeño.

Extensión del pipeline para manejar operaciones multiciclos: pipelined
La etapa de división sigue siendo una unidad funcional que requiere múltiples ciclos, no fue implementado con pipeline.
Se separa en etapas cada unidad funcional. La latencia es el tiempo que tarda una unidad funcional en realizar la operación desde que entra y sale (en ciclos).
El intervalo de iniciación me dice el grado de implementación en pipeline que tiene la unidad funcional.
Si las instrucciones no están separadas quizás no conviene que las unidades funcionales trabajen con pipeline.
La latencia es el tiempo que la unidad funcional tarda en realizar su trabajo desde que empieza hasta que termina.

Cuando puedo decidir no implementar una unidad funcional con pipeline: cuando son poco frecuentes o cuando las operaciones de ella están espaciadas. En el caso de la división, es muy complicado dividir en etapas la unidad de división.

1- Aún pueden existir riesgos estructurales dado que la unidad de división no está implementada con pipeline.
2- Debido a que las instrucciones tienen variados tiempos de ejecución, el número de escrituras requeridos en un ciclo puede ser mayor que 1.
3- Son factibles los riesgos WAW, dado que las instrucciones alcanzan la etapa WB fuera del orden de iniciación.
4- Las instrucciones se completan en un orden distinto al de iniciación.
5- Debido a la latencia de operaciones más largas los riesgos RAW son más frecuentes.

Hiperpipeline

Cuando tiene una cantidad respetable de etapas. 8 etapas o más.
Se incorporaron más unidades de saltos, más transistores para manejo de memoria virtual, predictor de saltos. Se dividen en etapas el acceso a memoria.

IF-Primera mitad de búsqueda de instrucción. Selección de Pc, inicio de acceso
IS-segunda mitad de búsqueda de instrucción. Completa acceso.
RF-Decodificación de instrucción. Chequeo de hit.
EX-Ejecución.
DF-Busqueda de dato. Primera mitad de acceso a cache de datos.
DS-Segunda mitad de búsqueda de datos. Compleción de acceso a cache de datos.
TC-Chequeo de Tag. Determinación de hit.
WB-Write Back.

Las dependencias de datos reales son las raw porque hay una instrucción previa termina después que una posterior y escriben en el mismo registro destino, me queda mal el valor. Las dependencias que empiezan con W son **dependencias de nombre**. No estás obligado a usar un registro en particular, si cambias el nombre del registro eliminas el riesgo con w. Los riesgos raw no se pueden eliminar, hay que resolverlos en el sentido de que no se produzca un cálculo erróneo. Los de W se pueden eliminar porque no es necesario que haya una dependencia real de datos.

Renombrado de registros

Para solucionar los riesgos que empiezan con W.
Va a haber una tabla la cual va a atar la correspondencia entre el nombre del registro software con registros hardware internos del procesador. Los registros software pasan a ser "virtuales". Cuando una instrucción necesita leer un registro, entra con el número de registro a la tabla y sale con el registro hardware. Y cada vez que una instrucción tenga un registro destino para escribir lo que va a hacer es asignar un nuevo registro hardware a una locación a ese registro destino. Si logra hacer esto no va a haber riesgos que empiecen con W porque cada vez que una instrucción escriba lo va a hacer en un registro físico *que no va a estar siendo utilizado por nadie.*
Cada vez que una instrucción tiene un registro destino le asigna una nueva locación, elige un registro nuevo.

También los registros que ya no están siendo usados pasan a la lista de registros libres, esto es cuando todas las instrucciones que lo tenían como fuente ya lo leyeron.

Superescalares

Se parte de un pipeline escalar. Para pasar al super escalar el hardware tiene que ser capaz de buscar decodificar y ejecutar más de una instrucción por ciclo. No tiene que garantizarse que esas instrucciones se puedan ejecutar siempre, sino que los riesgos van a marcar un límite. Aca hay un nivel de paralelismo distinto al del pipeline, el del pipeline era temporal (distinto grado de avance) y es un paralelismo real donde un par de instrucciones tienen caminos paralelos para poder avanzar que pueden no introducir riesgos estructurales para ejecutarse varias instrucciones a la vez.
Cuando tenemos un superescalar, hablamos de ipc (instrucciones por ciclo) y es mayor a 1. El caso ideal, si es de dos vías el ipc ideal es 2. Pero por los distintos tipos de riesgos y los programas a bajo nivel no se va a poder llegar al ipc ideal.
Único pc, único cpu. Sólo ahora se buscan dos instrucciones por ciclo.

Algoritmo de Tomasulo

Para el camino de datos de la ejecución superescalar. La idea es que no va a haber una unidad de control sino que el avance del cálculo va a ser de manera descentralizada y distribuida. Va a haber buffers llamados estaciones de reserva que van a tener un bus de datos común y van a estar pendientes de él para observar y colocar resultados sobre él. El acceso a registros es reemplazado por valores o punteros a las estaciones de reserva.

Etapas

- issue
- execute
- write result

Riesgos RAW: resueltos por copia de operandos o punteros a estaciones de reserva a espera de valores a calcular.
Riesgos WAW y WAR: eliminados por renombrado de registros.

Información en las estaciones de reserva

op→ operación a realizar sobre los operandos fuentes.

Vj,Vi→ campos valid, los dos fuentes. Cuando una instrucción se encola o se pone en una estación de reserva, si los fuentes ya están calculados y están en el register file, se los lee del register file y se los copia aca. Entonces en la estación de reserva quedan el opcode y los operandos.

Qj,Qk → lugares donde pongo el nombre de la estación de reserva que tiene el calculo que necesito de manera que cuando esa RS está produciendo el cálculo con su U Fun correspondiente y ponga el resultado en el bus común, entonces la estación de reserva que tiene guardado este link (Q) va a capturarlo de manera autónoma, y va a eliminar el campo Q y copiar en el campo V. Acá es donde está la parte distribuida porque cada RS que se esté ocupando, es la que va a producir el resultado que se está esperando y cuando esté listo lo va a publicar. Si el resultado está listo, se ponen en cero estos campos.

A → campo de address del dato.

Busy → bit para indicar si la estación de reserva está ocupada

El register file tiene el siguiente campo:
Qi→ un valor va a ir a un reg destino. El valor de ese reg destino en el reg file ya no es válido. En el campo Q se pone el nombre de la estación de reserva que va a producir el valor que tiene que ir en ese registro. Cuando se ponga el dato resultado de la operación, si la operación tenía un reg destino el reg file va a copiar ese resultado.

Arquitecturas de very long instruction word y arquitecturas multithreading

Clasificación de arquitecturas de **computadoras de flynn**
Clasifica a las computadoras de acuerdo a si el flujo de instrucciones es simple (uno) o múltiple y si el flujo de datos que utilizan dichas instrucciones es simple o múltiple.

Procesadores sisd
único flujo de datos → único pc, único hilo de ejecución. Cada instrucción opera sobre un único juego de datos (un fuente y un destino por ejemplo). Es el caso de los que estuvimos estudiando. Mips con pipeline es, mips con superescalar también. (su flujo viene de un único pc)

Procesadores simd
Hay un programa que tiene el algoritmo y la ejecución del programa dispara una cuenta para múltiples juegos de datos.
Cada elemento del procesador tiene su a,b,c local. Hay un pc que tiene el algoritmo.
Por ejemplo para procesar multimedia. Cualquier reproductor que esté procesando el audio tiene que hacer seguramente la misma operación sobre el flujo de datos que está viniendo y poder aprovechar la ALU acelera el **procesamiento multimedia**. Por eso esta extensión se llama extensión simd.
Lo de la extensión de instrucciones multimedia.

Procesadores misd
no existe aunque se intentó

Procesadores mimd
Estos son los multiprocesadores verdaderos.
Divididos en dos categorías:

- los fuertemente acoplados: comparten la info a través de un mismo espacio de direcciones, espacio común. No hay necesidad de ir pidiendo datos por otro sistema (el de pasaje de mensajes). Todas las cpus ven la misma memoria.
- los débilmente acoplados: cuando un sistema tiene multiprocesadores donde para saber cuánto vale la variable que tiene otro procesador hay que mandarle un mensaje. Es visible al software. Si hay que tratar con protocolos de comunicación al programar. Hay un sistema que se llama cluster que es un agrupamiento de computadoras estándar y es el más débilmente acoplado, con distintos sistemas operativos que operan una aplicación.

Multiprocesadores conectados por un único bus
El único bus es lo más fácil y lo más barato.
Ventaja: economía y practicidad.

Desventaja: no escala, se genera un cuello de botella cuando se conectan muchos procesadores a ese bus.

Multiprocesadores conectados por una red

Cuando no podemos usar bus y queremos tener conectados varios procesadores, se tiene una red de interconexión vía una topología.

Topologías de red

Crossbar: permite que **cualquier par de procesadores compartan** información al mismo tiempo. Pero puede ser que al paso siguiente los interlocutores cambien. Puede conectar cualquier par sin ningún tipo de restricción. Desde el punto de vista del algoritmo es la más flexible pero desde el punto de vista del hardware de la red es muy complejo, entonces la limita en cuanto a su *escalabilidad* cuando se tienen muchos procesadores.
Omega network: tiene elementos de conmutación que tiene dos entradas y dos salidas. Conecta directo o cruza. Es muy popular porque está buena para algoritmos importantes. Permite que tu algoritmo funcione bien y que escala bien. Pero limita la eficiencia de ciertos algoritmos.
Anillo: el que mejor escala. Permite que el mensaje tenga dos caminos.
Cada elemento del procesador tiene que tener dos links, uno para el vecino de la derecha y uno para el de la izquierda.
Si vos queres aumentar la cantidad de procesadores, no aumenta la complejidad de la red. El tema es si le queres mandar un mensaje al vecino llega rápido, pero si esta mas lejos tiene la demora de pasar por los pueblitos intermedios. Brinda una cierta tolerancia a la falla, siempre tenes otro link para llegar.

Coherencia en multiprocesadores

Si hay otro procesador en el sistema que me está modificando la memoria y tengo cacheada la posición de memoria hay una inconsistencia, tiene que haber una coherencia. Hay que mantener consistente la memoria. Para que no se estén resolviendo operaciones con información desactualizada.
Por ejemplo si los procesadores comparten un bus, la técnica snoop es que cada caché envía si están modificando alguna memoria compartida para invalidar, etc.

Arquitectura VLIW: very long instruction word

Un procesador superescalar lo que hace es buscar en cada ciclo una cierta cantidad de instrucciones, las decodifica etc. Al poder hacerlo en paralelo acelera la ejecución. *Lo hace transparente al software.*
La idea de VLIW es que el compilador va a empaquetar en una palabra muy ancha hasta 4 instrucciones, y garantiza el hardware que estas 4 se van a poder ejecutar porque *no va a haber riesgos estructurales ni dependencias de datos*. Entonces el hardware se saca de encima la complejidad de andar haciendo el análisis de riesgo y dependencia de datos.
Hipótesis: todas las instrucciones empaquetadas en una instrucción de palabra ancha se pueden ejecutar en paralelo.
El compilador tiene que conocer el hardware. Se libera al hardware de hacer los chequeos de riesgos, ya que lo que le mandaron lo puede ejecutar en paralelo.

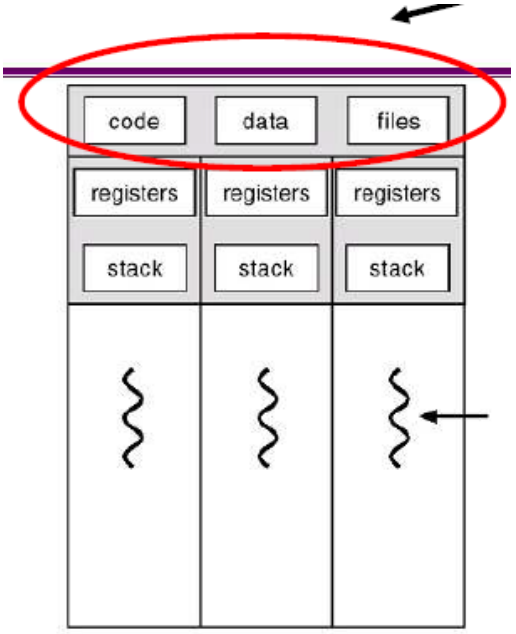
Procesamiento multihilo

Ejecución concurrente de múltiples hilos (threads) de procesamiento, pertenecientes a un mismo programa o a diferentes programas. NO estamos hablando de procesos.

CMT (grano grueso): el procesador cambia de hilo de ejecución cuando el hilo incurre en algún evento que tiene una penalidad relativamente alta. Por ejemplo un miss en caché. Entonces se conmuta a otro hilo ya que este puede que esté parado por muchos ciclos (ciclos de stall).

FMT (grano fino): Puede cambiar de hilo en cada ciclo. La conmutación se puede hacer a nivel de instrucción con instrucción.

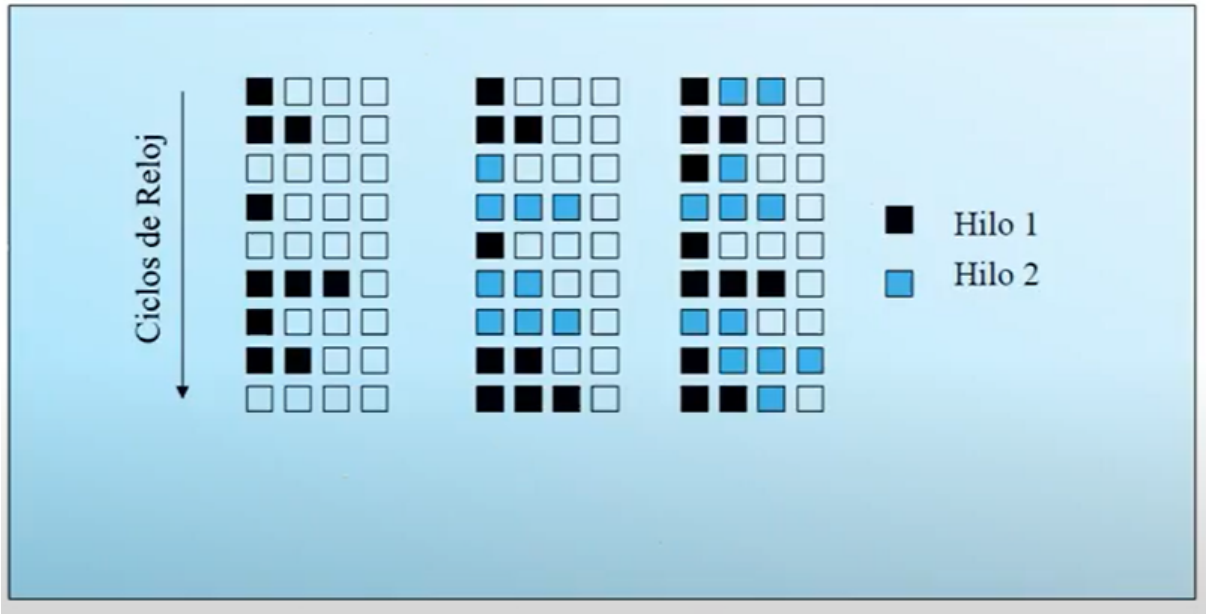
SMT (multihilo simultáneo): multihilo simultáneo, el procesador puede tomar en paralelo instrucciones de más de un hilo. El que utiliza intel es HT (si el procesador dice HT cada cpu es de multihilo simultáneo).



Desperdicios vertical y horizontal

Cuando podes intercalar hilos, por ejemplo con un fmt, conmutó de hilo.

En un superescalar de 4 vías:



Desperdicio horizontal: no se pudo ejecutar en paralelo.
Desperdicio vertical: no pudo ejecutar nada tal vez por una falla de la caché.

Pipeline SMT
Para que un hilo sea smt o ht hay que duplicarlo o agregarle lo que es propio de cada hilo.

Optimizaciones de software

Nos vamos a enfocar en disminuir el tiempo de ejecución.

Quien optimiza? El compilador o el usuario.

Optimizaciones -o0,-o1
El compilador tiene varios flags que son para optimizar.

Tipos de optimizaciones

- Locales: aplican localmente a un procedimiento.
- globales: aplican a varios procedimientos.

Optimizaciones locales

Asignación de registros

La idea es mantener una variable en un registro. Me evito el acceso a memoria cuando la quiero leer y escribir. Si tengo cache. Me evito un posible miss y ciclos de stall. Podría decrementarse la cantidad de instrucciones (evito los load store).

Eliminación de código

El código inalcanzable son instrucciones que nunca se ejecutan. Código muerto es el que no produce efectos en el resultado. Es código que ocupa espacio en memoria.

Plegado y propagación de constantes
Algunas cuentas que deberían hacerse en tiempo de ejecución se pueden hacer en tiempo de compilación (plegado de constantes). Si hay sumas de distintas variables se junta el resultado total en tiempo de compilación (propagación de constantes)

Reducción del costo de operación
Reemplazar divisiones y multiplicaciones por shifts y reemplazar multiplicaciones por sumas.

Optimización de bucles
Desenrollado de bucles: Alivia el costo relacionado al control del bucle. En el bucle no desenrollado voy a tener un branch cada 4 instrucciones aprox. En el de la derecha, hace varias sumas, ocupa más el código pero tiene un branch cada 40 instrucciones aprox. Si ese branch metía 3 ciclos de stall, se disminuyen los ciclos en el de la derecha.

Fusión de bucles: se aplica por el programador.

Optimizaciones globales
Expansión en línea
El uso de funciones es ventajoso para encapsular, prolijidad, etc. El llamado a una función tiene un costo. El costo de llamar y volver puede ser mucho más que la operación que está haciendo la función. En tal caso en vez de llamar a la función, se pega el cuerpo de esa función en el lugar del llamado.

Optimizaciones de la jerarquía de memoria
Lectura adelantada (prefetching)
La idea es que cuando vayamos a buscar un dato, que ya esté cargado.
Trata de adelantarse a los misses en caché. El hardware trata de predecir quizás por localidad espacial, y hace un fetch del próximo bloque. Hay que ver donde se guarda. Cuando se tienen buffers aparte de la caché, no hay problema de cache pero el hardware es más complejo.
Lo tengo que pedir con una anticipación suficiente para que esté ahí para cuando lo necesite.
Siempre que termino de procesar un bloque termina la prebúsqueda de uno que lance 3 bloques atrás. Hay que tener los datos de cuánto tarda la prebúsqueda y cuanto la iteración.

Reemplazo de elementos de arreglo
En vez de acceder a un elemento del arreglo, tratar de acceder a una variable.

Intercambio de bucles
Es importante saber cómo se guardan las matrices en el lenguaje porque siempre conviene recorrer en el orden que se guarden consecutivamente en memoria (c en filas). De esta manera se aprovecha la localidad espacial.

Operación en bloques
Operar en bloques más pequeños para aprovechar la localidad temporal.

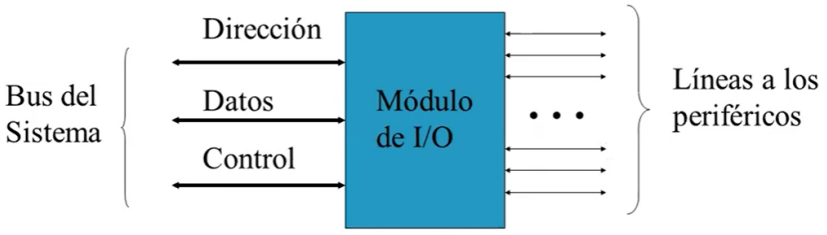
Rellenado de arreglos (padding)
Cuando se trabaja con arreglos, suele pasar que las posiciones simultáneas de los distintos arreglos que usa el algoritmo mapean a los mismos conjuntos. Hay misses de conflicto en caché. Si eso pasa, se declara un arreglo de relleno entre cada par de arreglos.

Reducción de solapamiento
Un dato en memoria puede ser accedido de diferentes maneras. Es importante reducir el aliasing para lograr optimizaciones más agresivas.

Entrada/salida

La transferencia de datos es muy similar a acceder a memoria desde el punto de vista de la CPU. Cada dispositivo con un único identificador.

Módulo entrada/salida: es la electrónica que trata en la computadora con el periférico, es el nexo entre el periférico y nuestro sistema CPU memoria.



Funciones del módulo

- Comunicación con la CPU
- Comunicación con el dispositivo
- Buffering de datos
- Control & Timing
- Detección de error

Registros del módulo

- Registro de datos que se comparten entre el entorno y nuestro sistema. Por acá pasan los datos que el sistema genera y los queres mandar afuera y viceversa.
- Registro de control/estado. Está destinando bits para reflejar el estado de módulo y de dispositivo y otros bits que puede escribir la cpu de manera tal para indicarle al módulo lo que tiene que hacer. Si el módulo es más complejo, este registro se separa en reg de control y reg de estado.

Técnicas de entrada y salida: Programada, por interrupciones, dma.

Entrada/salida programada
La cpu es la encargada de llevar a cabo el proceso de entrada y salida. Es la encargada de iniciarlo, terminarlo, etc pero la cpu es la encargada de hacerlo paso por paso. Hace todo, con lo cual *se pierde tiempo de cpu*.

Se utiliza en procesadores embarcados donde la cpu no es potente, en cpus donde no hay nada más para hacer. En las computadoras de uso de propósito general no sirve porque se trabaría la pc y no respondería.
La CPU pide una operación y el módulo setea los bits de estado. La cpu debe chequearlos periódicamente porque el módulo no le indica explícitamente que terminó de realizar la operación.

Mapeo de entrada/salida
Dos tipos de mapeo. Los dispositivos tienen una posición de memoria. Si la dirección se ubica en el espacio de direcciones de la memoria del sistema, se dice que esta E/S mapeado en memoria, si esos registros están en un espacio de direcciones *diferente* se dice E/S separada.
En el caso de memoria mapeada no distingo, y estoy ocupando espacio de memoria (desventaja) pero todas las instrucciones que acceden a la memoria acceden a los registros de los módulos porque la cpu no los distingue (ventaja).
En el caso de entrada salida separado como es un bus separado (siempre separada desde el punto de vista lógico, no siempre físicamente separado).

Entrada/salida manejada por interrupciones
Soluciona la espera de la CPU. No se requiere el chequeo repetido por parte de la CPU del dispositivo. El módulo de E/S interrumpe cuando está listo.
Si el módulo va a interrumpir a la cpu, tiene que estar chequeando constantemente si llego una interrupción. Este chequeo debe ser casi inmediato. El módulo le va a avisar a la cpu que terminó de enviar el dato o que ya puede recibir otro, etc.
Cada vez que la cpu ejecuta una instrucción se verifica si algún módulo interrumpió. Ahí se salva el contexto y se procesa la interrupción. Procesar la interrupción significa saltar a ejecutar un código que atienda el procesamiento de ese dato (saltar a ejecutar un handler).
El chequeo no implica que hay un clock más, sino que hay una acción más que se hace al final de la ejecución de la instrucción. No se mira si hay interrupciones durante la ejecución.

Ciclo de interrupción
Se agrega al ciclo de instrucción. El procesador chequea si hay interrupciones. Si no las hay, busca la próxima instrucción. Si las hay, suspende la ejecución actual, salva el contexto, pone el pc al comienzo del handler, procesa la interrupción y restablece el contexto para continuar con la ejecución.

Clasificación de las interrupciones
- internas (excepciones): cálculos erróneos, etc.
- externas: dispositivos.
- de software: saltar a ejecutar un handler.

Identificación del módulo que interrumpe
Una línea de interrupción diferente para cada módulo (vectorizada)
Los módulos lo que hacen en realidad es mandar sus líneas de interrupción individuales a un *controlador de interrupciones* que está en el chipset. Es el controlador de interrupciones el que sabe quién está interrumpiendo. Como no es práctico que la cpu lo maneje, está este controlador de interrupciones que concentra todas las líneas de los módulos, hace como la decodificación e interrumpe a la cpu y ya resolvió el tema de la identificación del módulo.

zona del vector de interrupciones: cada dirección está asociada a un periférico. Se puede grabar la dirección del handler que maneja esa interrupción y permite al software ubicar los handler en la pos de memoria que quede cómodo y lo único que está fijo por hardware es el número de interrupción que es el número de entrada en el vector de interrupciones.

Una sola línea de interrupción compartida entre todos los módulos
Software poll: llega una interrupción pero no sabe de quién es porque todas comparten la misma línea. Es más barato pero más lento. La cpu tarda tiempo en ir módulo por módulo chequeando el *bit de estado* para ver si tiene una interrupción.

Hardware poll: los módulos comparten la línea de interrupción pero la identificación es por hardware. El módulo responsable ubica un “vector” en el bus. Si el módulo no fue cierra la llave y el pulso continua. Cuando se envía el pulso para verificar si interrumpio se envía a todos y frena cuando llega al que interrumpio. La cpu usa el vector para identificar el handler a usarse.

<u>Múltiples interrupciones</u>
Secuenciales: se ejecuta un programa, se produce una interrupción y se atiende el handler x. Cuando termina la del handler X, se hace la del handler Y.

Anidadas: se produce una interrupción. Si se produce otra que tiene más prioridad, se atiende esa y después se sigue con la anterior. Hay una jerarquía de prioridades.

En el caso de software o hardware pull la prioridad la estableces según el orden en que preguntas porque al primero que te dice que interrumpió lo atendes.
Las interrupciones requieren una atención de a lo mejor 10 instrucciones. Puede ocurrir un cuello de botella si no es rápido el dispositivo.

<u>DMA</u>
Utiliza el esquema de interrupciones como un esquema auxiliar, pero el mecanismo central de la *transferencia* de la información entre el dispositivo externo y la memoria lo hace el controlador de dma y no la cpu.
La cpu programa al controlador dma para q haga una transferencia, el controlador dma hace la transferencia y cuando termina la transferencia, le avisa a la cpu que terminó mediante una interrupción para que la cpu pueda ir a atenderlo pero no va a ir a buscar un dato, sino un bloque completo.

Registros de DMA: dos registros. Uno que contiene el número de bytes o palabras a transferir y otro para almacenar la dirección de memoria.