

Índice

Representaciones	2
Representación de enteros	2
Representación de punto flotante	2
Lenguaje de máquina	3
Jerarquía de memorias	4
Jerarquía de dispositivos	4
Tecnologías de almacenamiento y tipos de acceso	5
RAM (Random Access Memory)	5
Memoria caché	6
Memoria virtual	7
Localidad (en este contexto)	9
Gestión de memoria	9
Herramienta de protección	9
Traducción de direcciones	10
Translation Lookaside Buffer	11
Tablas de paginación multinivel	12
Arquitectura	13
Diseño Lógico	14
Implementación secuencial	16
Implementación segmentada	19
Implementación con paralelismo	21
Consultas	25

1. Representaciones

Word size: cantidad máxima de bits que pueden ser procesados al mismo tiempo. Esto determinará el máximo tamaño del virtual address space.

Byte ordering o *endianness*: formato en que se almacenan los datos de más de un byte en un ordenador.

Little Endian: byte menos significativo. Ej.: Windows, Linux.

Big Endian: byte más significativo primero. Ej.: Sun.

Representación de enteros

Unsigned: $U_{\min} < u_x < U_{\max} \rightarrow 0 < u_x < 2^w - 1$

Signed: $T_{\min} < x < T_{\max} \rightarrow -2^{(w-1)} < x < 2^{(w-1)} - 1$

Rango asimétrico: $|T_{\min}| = T_{\max} + 1$

$U_{\max} = 2 * T_{\max} + 1$

Conversión de signed a unsigned (T2U(x))

si $x \geq 0 \rightarrow u_x = x$

si $x < 0 \rightarrow u_x = x + 2^w$

Conversión de unsigned a signed (U2T(u_x))

si $u_x \leq T_{\max} \rightarrow x = u_x$

si $u_x > T_{\max} \rightarrow x = u_x - 2^w$

Representación de punto flotante

$V = (-1)^s \times M \times 2^E$

s = 1 bit de signo

M = n bits que representan un número fraccional binario

E = k bits de exponente

2. Lenguaje de máquina

Traducción de programas en lenguaje C

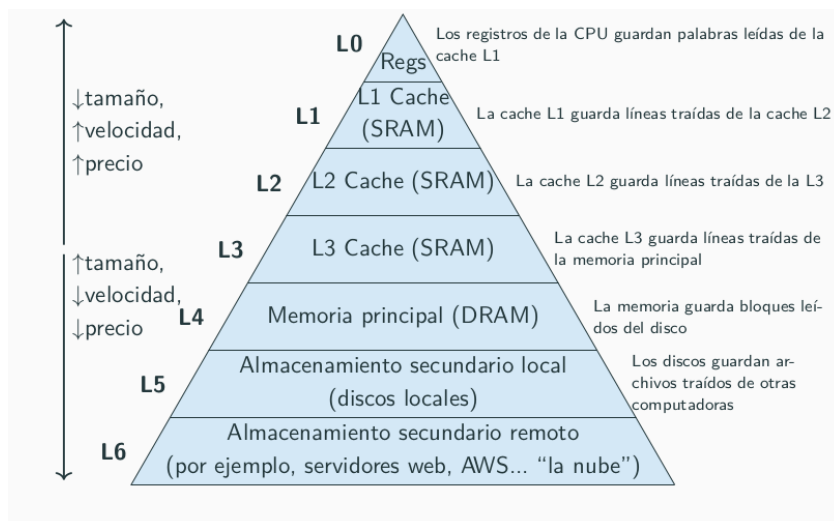
1. Se escribe un programa en un archivo de texto que contiene sólo caracteres ASCII cuyo formato es **.c**.
2. El **.c** pasa por el pre-procesador (cpp) que devuelve una versión modificada también en formato C pero con el sufijo **.i**.
3. El compilador (cc1) traduce el **.i** a lenguaje Assembly, obteniéndose un **.s**.
4. El ensamblador (as) traduce el **.s** en instrucciones de lenguaje de máquina y las empaqueta en un programa objeto reubicable. El resultado se guarda en un archivo objeto **.o**.
5. El enlazador (ld) fusiona el archivo objeto **.o** con los **.o** necesarios de las librerías de los compiladores de C para devolver un archivo ejecutable que puede ser cargado en la memoria y ejecutado por el sistema.

3. Jerarquía de memorias

Jerarquía de dispositivos

Los sistemas modernos de memoria funcionan como una jerarquía de dispositivos de almacenamiento con diferentes capacidades, costos y tiempos de acceso.

- **Registros de CPU:** datos usados más frecuentemente.
- **Memoria caché:** Dispositivo de almacenamiento de datos más rápido y de menor capacidad que actúa como staging area de un subconjunto de datos almacenados en un dispositivo más lento y de mayor capacidad
- **Memoria principal:** Los datos se guardan en grandes y lentos discos que funcionan como etapas intermedias para los datos guardados en discos.
- **Discos:** Dispositivos que guardan miles de gigabytes pero cuya lectura de datos es mucho más lenta que de las DRAMs o SRAMs.



Localidad

Los programas con buena localidad tienden a acceder a datos de los niveles más altos de la jerarquía de memorias que los programas con baja localidad, por lo tanto, son más rápidos.

Las memorias caché son las que mayor impacto tienen en las performances de programas.

Principios de localidad

- **Localidad espacial:** Los programas tienden a acceder a direcciones de memoria cercanas a las direcciones previamente accedidas.

- **Localidad temporal:** Los programas tienden a acceder a direcciones de memoria **iguales** a las direcciones previamente accedidas.

Tecnologías de almacenamiento y tipos de acceso

Almacenamientos no volátiles: Memoria de solo lectura (ROM), ROM eléctricamente borrrable y programable (EEPROM), memorias flash. Usos:

- Los firmwares se suelen almacenar en ROMs (BIOS, controladores de discos, placas de red, televisores, consolas de videojuegos, etc.).
- Discos de estado sólidos (SSD, solid state disks)—reemplazan a los discos rotativos.
- Caches de otros discos más lentos.

SSD

- Los datos se leen/escriben por unidades de páginas.
- Es necesario borrar el bloque antes de escribir una página.
- Un bloque se gasta (wears out) después de aproximadamente 100000 escrituras.
- El acceso secuencial es más rápido que el aleatorio.
- Las escrituras aleatorias son un tanto más lentas (para modificar una página es necesario copiar todas las demás a un nuevo bloque).
- No tiene partes que se muevan (como los discos rotativos) ⇒ más rápido, menor energía, más resistente.
- Se desgastan.
- En 2020, aproximadamente 4 veces más caros (por GB).

RAM (Random Access Memory)

- Permite la lectura y escritura de cualquier dato con el mismo costo temporal, no siendo necesario seguir un orden de acceso secuencial.
- Está compuesta por muchos chips.
- El almacenamiento básico es una celda con un bit por celda.

SRAM (Static)

- Usado para memorias caché
- Los bits se almacenan en estados estables, prevalecen mientras haya energía, no hay necesidad de refrescar.
- 6 transistores por bit.


DRAM (Dynamic)

- Usado para memoria principal
- Los bits son almacenados como carga de un capacitor y deben ser actualizados continuamente por el sistema de memoria.
- 1 transistor por bit.

Observación: Las DRAM/SRAM analizadas son volátiles, es decir, pierden su información si se corta la provisión de energía. (Parece que existen variedades de SRAM no volátiles).

4. Memoria caché

Características

- Dispositivo de almacenamiento.
- Chica y rápida. Actúan como etapas intermedias para subsets de datos e instrucciones almacenadas en memorias de nivel superior (ver pirámide de Ghiza).
- Aprovechan la localidad.
- Se almacena memoria de a bloques, reemplazando lo que había antes (ver políticas de reemplazo).
- Tipos de fallos:
 - **En frío:** la caché comienza vacía. Falla siempre en la primera referenciación.
 - **Por capacidad:** maaaa se me lleno la cache ya me rompi .
 - **Por conflictos:** cuando, siendo la caché lo suficientemente grande, se busca ubicar más de un bloque en la misma posición.

Organización de la caché

- Tamaño de la caché: $C = S * E * B$
 - $S = 2^s$ (número de sets en caché).
 - $E = 2^e$ (número de líneas por set).
 - $B = 2^b$ (número de bytes por bloque).
- Direcciones de memoria:
 - [Cache Tag (t bits) | Cache Index (s bits) | Cache offset (b bits)]
- Estructura de línea:
 - [valid bit | dirty bit | tag | block [0,1,2,3,4...]]

Optimización de cálculos con matrices.

Métricas de desempeño de la caché.

Memory mountain.

Blocking.

AHRE



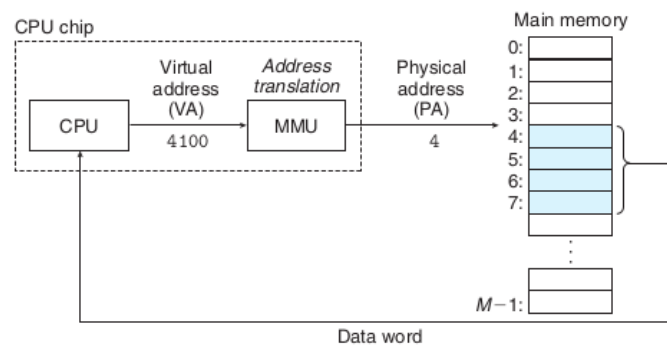
5. Memoria virtual

Contexto

La memoria principal puede verse como un array de M celdas contiguas de un byte y cada byte tiene una dirección física única (PA).

Con memoria virtual la CPU accede a la memoria principal generando una dirección virtual (VA) que es convertida a la dirección física apropiada antes de ser enviada a memoria principal (address translation).

Esto requiere de cooperación entre el hardware de la CPU y el sistema operativo. La MMU (hardware dentro del CPU chip) traduce las direcciones usando una tabla alojada en la memoria principal cuyos contenidos son manejados por el sistema operativo.



Organización de cachés

- **Caché SRAM:** cachés L1, L2, L3 (entre la CPU y la memoria principal).
- **Caché DRAM:** caché del sistema de memoria virtual (que cachea las páginas virtuales en la memoria principal).

Los misses de DRAM son más costosos que los de la SRAM.

Espacio de direcciones virtual (VAS)

- Puede verse como un array de N células contiguas de 1 byte cada una almacenadas en el disco.
- Cada byte tiene un único VA (dirección virtual) que sirve como índice dentro del array.
- Los contenidos del VAS se *cachean* en memoria principal (DRAM); los datos en discos son particionados en bloques que sirven como unidades de transferencia entre el disco y la memoria principal.
- Los bloques son llamados páginas virtuales (VPs) y su tamaño es $P = 2^p$ bytes. (Los bloques de direcciones físicas también son agrupados en páginas físicas PPs también de tamaño P).
- El set de páginas virtuales es particionado en 3 subsets disjuntos:

- **Unallocated:** páginas no *alocadas* o creadas por la VM, que no tienen asociados datos y no ocupan espacio en el disco.
- **Cached:** páginas *alocadas* que están *cacheadas* en la memoria física.
- **Uncached:** Páginas *alocadas* que no están *cacheadas* en la memoria física.

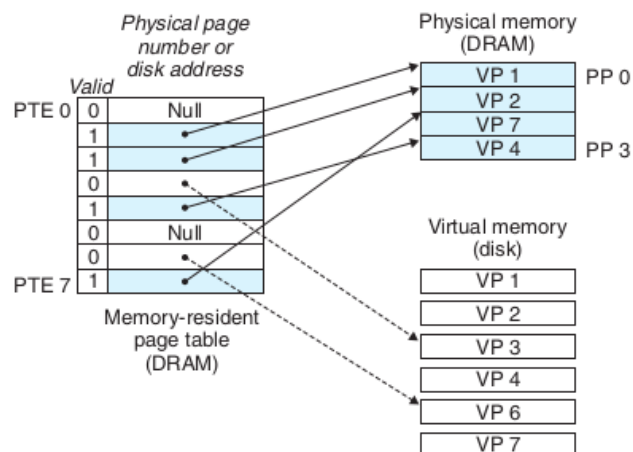
Motivos de su uso

- Hace uso eficiente de la memoria principal.
 - Usa la DRAM como caché de partes del Virtual Address Space
- Simplifica la gestión de la memoria.
 - Cada proceso tiene un espacio de direccionamiento lineal.
- Aísla los espacios de direcciones
 - Un proceso no interfiere con la memoria de otro.
 - Un programa de usuario no accede a información del kernel.

La DRAM se usa como caché de cosas que están en el disco entonces no es necesario acceder siempre a él, pueden ir guardándose los datos que se van necesitando directamente de la memoria principal.

Page Table

Se encuentra en la memoria principal (DRAM), y esta se beneficia del principio de localidad. Los programas con buena localidad temporal mantienen conjuntos activos de páginas (working sets) reducidos. Si el working set es mayor a la memoria principal, el desempeño cae abruptamente debido a que las páginas se copian continuamente entre el disco y la DRAM.



- Formada por Page Table Entries (PTE) que mapean páginas virtuales (VA) a páginas físicas (PA) (el Kernel almacena por proceso en la DRAM).

- **Page Hit:** se referencia un dato usando memoria virtual correspondiente a una página física que **está** en memoria principal (caché DRAM **hit**).
- **Page Fault:** se referencia un dato usando memoria virtual correspondiente a una página física que **no está** en memoria principal (caché DRAM **miss**).
 - La MMU lanza una excepción por page fault, manejada por el Kernel.
 - El Kernel elige una página víctima para desalojar, y la sobrescribe con una VP correcta.
 - El Handler ejecuta la instrucción **iret** para retorno desde interrupciones.

Localidad (en este contexto)

- Los programas tienden a acceder a un conjunto activo de páginas llamado **Working Set**.
 - Los programas con localidad temporal mantienen un working set reducido.
- Si working set > memoria principal: El desempeño cae abruptamente porque las páginas se copian continuamente entre el disco y DRAM.

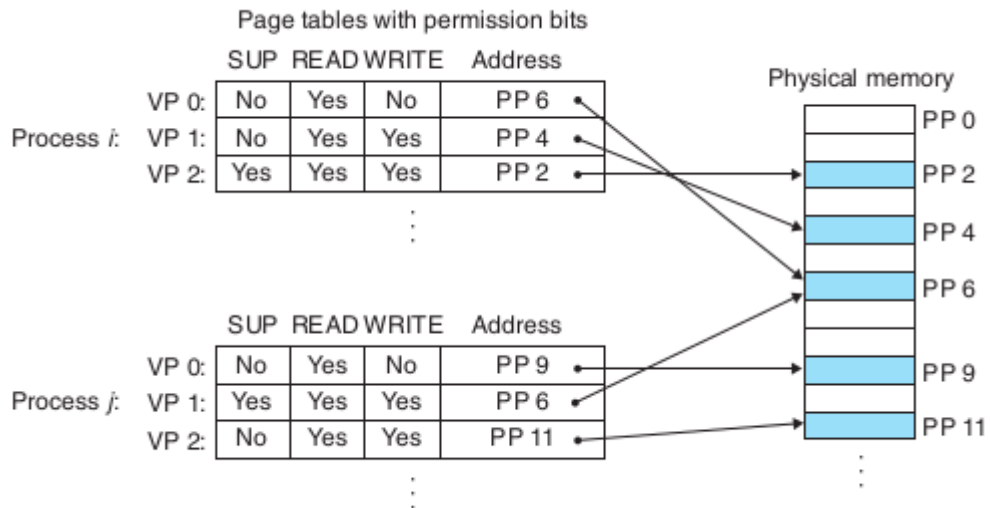
Gestión de memoria

- La memoria virtual simplifica la reserva de memoria.
 - Una VP puede mapear cualquier PP.
 - Una VP se puede almacenar en distintas PP (¿será así como funcionan los punteros?).
 - Múltiples VP pueden mapear una misma PP (compartición de datos y código entre procesos).
- Cada proceso tiene su propio Virtual Address Space.

LA MEMORIA VIRTUAL IMPLEMENTA EN LA RAM ALGO QUE SIRVE COMO CACHÉ AL DISCO, PERO NO ES MEMORIA CACHÉ.

Herramienta de protección

- Se extienden los PTEs con bits de permisos.
- La MMU comprueba estos bits en cada acceso



- **SUP:** Se debe estar en modo kernel para leer la PP
- **READ:** Se puede leer la página.
- **WRITE:** Se puede escribir en la página.
- **EXEC:** (no se encuentra en el gráfico) se puede extraer código ejecutable de la PP.

Traducción de direcciones

- Mapeo entre elementos de un VAS de N elementos y un PAS de M elementos, realizado por la MMU por medio de seleccionar la correcta PTE de la page table.
- $VA = VPN + VPO$
- $PA = PPN \text{ (de la PTE)} + VPO \text{ (del VA)}$
- Parámetros importantes:

Elemento	Descripción
$N = 2^n$	Cantidad de direcciones en un VAS
$M = 2^m$	Cantidad de direcciones en un PAS
$P = 2^p$	Tamaño de una página en bytes
VPO	Offset de una VP (p bits)
VPN	Número de una VP (n-p bits)
TLBI	Índice de la TLB
TLBT	Tag de la TLB
PPO	Offset de una PP (p bits)

PPN	Número de una PP
CO	Offset en un bloque de la caché
CI	Índice de la caché
CT	Tag de la caché

Translation Lookaside Buffer

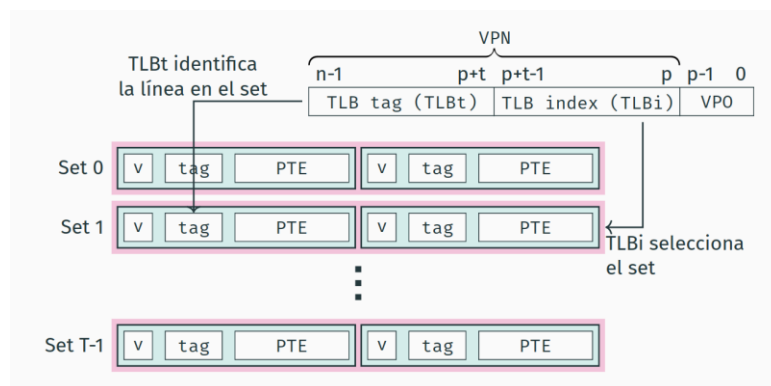
Contexto

Cuando el CPU genera una VA, la MMU debe recurrir a una PTE para traducir la VA a una PA. Si la PTE no está *cacheada* en alguna caché el costo en ciclos es mayor, para minimizar esto, se incluye una pequeña caché de PTEs en la MMU (la TLB), una caché con addressing virtual cuyas líneas contienen un bloque con una única PTE.

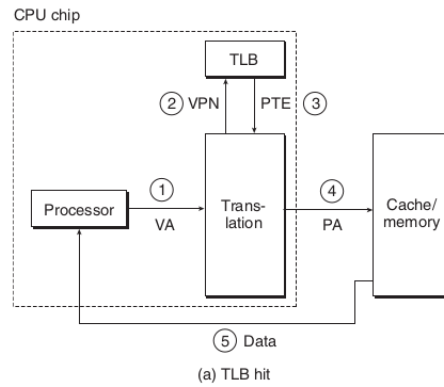
Características

- Es una caché asociativa por conjuntos de la tabla de paginación, en hardware, en el chip.
- Mapea números de páginas virtuales (**VPN**) con números de páginas físicas (**PPN**). Funciona como una caché de las VPN ya vistas.
- Contiene PTE 's completas para un subconjunto pequeños de páginas.

Acceso a la TLB: La MMU usa la VPN de la VA para acceder a la TBL.

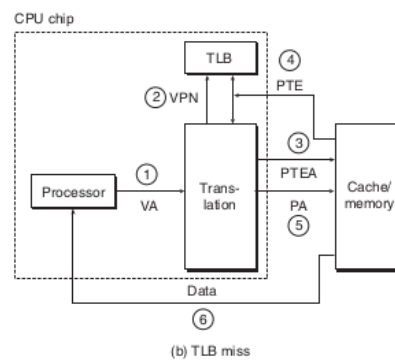


TLB hit: Elimina un acceso a memoria (y a la tabla de paginación).



1. La CPU genera un VA
2. La MMU obtiene el PTE apropiado de la TLB
3. La MMU traduce la VA a una PA y la envía a la caché/memoria principal.
4. La caché/memoria devuelve el dato a la CPU.

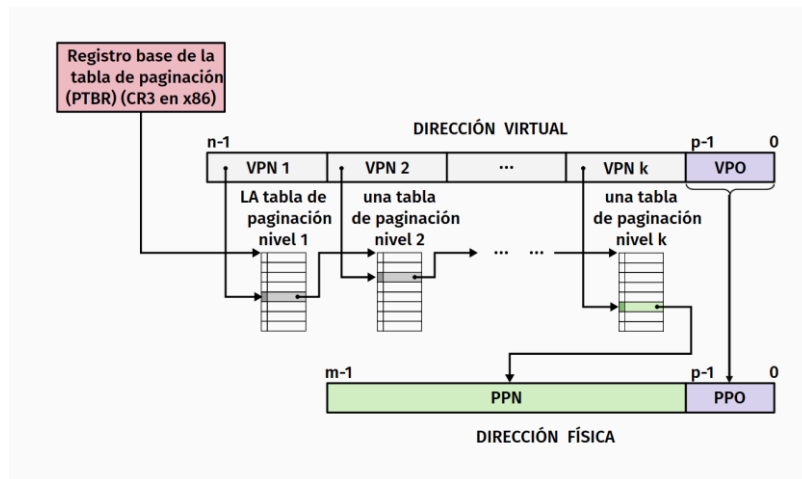
TLB miss: Añade un acceso a memoria (estos misses son raros).



1. La CPU genera un VA
2. La MMU no obtiene el PTE de la TLB, y debe buscarlo de la caché/memoria principal.
3. De la caché/memoria principal se obtiene el PTE apropiado y se pasa a la TLB, donde se lo guarda, posiblemente sobrescribiendo una entrada existente.
4. Finalmente la MMU traduce la VA a una PA y la envía a la caché/memoria principal.
5. La caché/memoria devuelve el dato a la CPU.

Tablas de paginación multinivel

- En una tabla de paginación de k niveles:
 - En los niveles 1 a $(k - 1)$, cada PTE apunta a tablas de paginación de su siguiente nivel,
 - En las tablas de paginación de nivel k , los PTE apuntan a VP.
- **Motivo:** Bajo ciertos requisitos, puede ocurrir que nos surja la necesidad de utilizar tablas de paginación de tamaños extremos.



6. Arquitectura

	Tipos de arquitecturas	
	CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)
Cantidad de instrucciones	Gran cantidad de instrucciones	Menor cantidad de instrucciones (codificaciones binarias más largas debido a esto)
Tiempo de ejecución de instrucciones	Instrucciones con largos tiempos de ejecución	Instrucciones con cortos tiempos de ejecución
Tamaño de codificación de variables	Codificación de las instrucciones de tamaño variable.	Codificación de las instrucciones de tamaño fijo.
Complejidad de desplazamientos	Hay múltiples formatos para especificar operandos.	Sólo se especifica base y desplazamiento de los operandos.
Operandos aritméticos	Los operandos de operaciones aritméticas y lógicas sólo pueden ser memoria o registros.	Los operandos de operaciones aritméticas y lógicas sólo pueden ser registros. Las referencias a memoria sólo se permiten en instrucciones de lectura o escritura.
	Implementation programs are hidden from machine level programs. The ISA provides a	Implementación expuesta a los programadas de nivel máquina.

	clean abstraction between programs and how they get executed.	
Condition Codes	Hay condition codes.	No hay condition codes (los resultados de instrucciones test son almacenados en registros normales para su uso en evaluaciones condicionales).
Argumentos de funciones	Se usa una pila (stack) para los argumentos de procedimientos y para los valores de retorno.	Se usan registros para los argumentos de procedimientos y para los valores de retorno.
Variables locales	Las variables locales generalmente son alocadas en el stack y los registros son reservados para valores intermedios.	Las variables locales se alocan en registros temporales (callee-saved registers).

Observación: Y 86-64 tiene atributos de ambos sets.

Diseño Lógico

Los tres principales componentes para implementar un sistema digital son:

- Lógica combinacional para computar funciones sobre los bits.
- Memoria para almacenar bits.
- Señales de clock para regular la actualización de la memoria.

Compuertas lógicas

- Computan un output que corresponde con alguna función booleana de los bits del input.
- Siempre están activas (responden constantemente a los cambios de los inputs).

Circuitos combinacionales

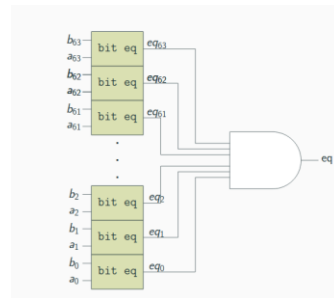
- Compuertas lógicas ensambladas en redes que forman bloques computacionales.
- La red no debe ser cíclica.
- Ejemplo: ALU. Tiene dos data inputs (A y B) y una condición de control que dependiendo de cómo se setee el circuito producirá una operación aritmética o lógica distinta.

Igualdad de bits

- $\text{bool eq} = (a \&\& b) \parallel (!a \&\& !b)$
 - Genera 1 si $a = b$.

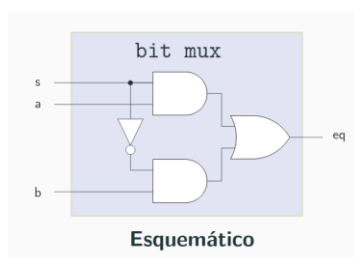
Igualdad de palabras (64 bits)

- $\text{bool eq} = (A == B)$

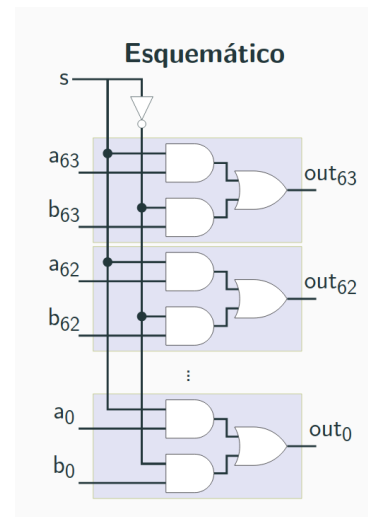


Multiplexor: Selecciona un valor entre un set de distintas señales dependiendo del valor de una señal de control recibida por input.

- Señal de control: s
- Señales de datos: a y b
- s selecciona entre a y b

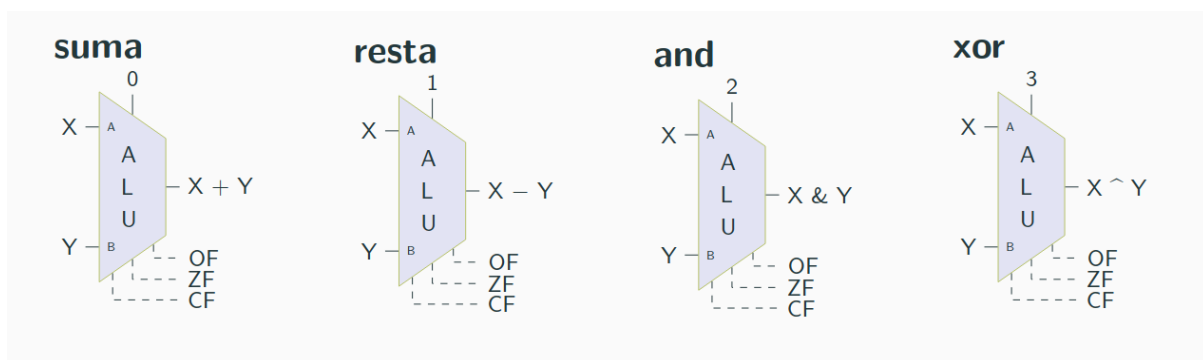


Multiplexores (palabras)



Unidad Aritmética - Lógica (ALU)

- Logica combinacional.
- La señal de control selecciona una operación.
 - 0: suma
 - 1: resta
 - 2: and
 - 3: xor
- Genera y modifica códigos de condición: OF, ZF, CF.



Registros (Distintos a los registros de Assembly)

- Almacenan palabras.
- Es un conjunto de Latches disparados por flancos.

- Cargan datos en el clock ascendente.

Register File (Banco de registros)

- Almacena los distintos registros, es decir, almacena múltiples palabras.
- Guarda los valores de los registros (%rax, %rdi, etc.).
- Los identificadores de registros sirven de direcciones.
- Posee múltiples puertos:
 - Tiene varias entradas y salidas (para direcciones y datos) separadas.

Implementación secuencial

- En cada ciclo de reloj se ejecutan todas las etapas requeridas para procesar una instrucción completa, por lo tanto, tiene un clock rate **inaceptablemente lento**.
- Consiste de lógica combinacional (mediante la que se propagan las señales) y dos formas de almacenamiento: clocked registers (PC y CC) y RAMs (register file, la memoria de las instrucciones y la memoria de los datos), que se controlan con una señal única de reloj que dispara la carga de nuevos valores en los registros y la escritura de datos en la RAM.
- Cada vez que el clock se mueve de abajo arriba el procesador comienza a ejecutar una nueva instrucción.
- El PC es guardado en un registro.
- La información fluye por medio de cables.
- Los procesos nunca requieren leer un estado actualizado por una instrucción para terminar su procesamiento.
- Ninguna instrucción debe setear y leer condition codes.

Set de instrucciones Y 86- 64

	icode	ifun	rA	rB	extra
halt	0	0			
nop	1	0			
cmovxx	2	fn	rA	rA	
irmovq	3	0	F	rB	Valor
rmmovq	4	0	rA	rB	Desplazamiento
mrmmovq	5	0	rA	rB	Desplazamiento
opq	6	fn	rA	rB	
jxx	7	fn	Destino		
call	8	0	Destino		

ret	9	0			
push	A	0	rA	F	
pop	B	0	rA	F	

Código	Registro	Código	Registro
0x0	%rax	0x8	%r8
0x1	%rcx	0x9	%r9
0x2	%rdx	0xA	%r10
0x3	%rbx	0xB	%r11
0x4	%rsp	0xC	%r12
0x5	%rbp	0xD	%r13
0x6	%rsi	0xE	%r14
0x7	%rdi	0xF	No register

Operaciones en la ALU	Código
Suma	0x60
Resta	0x61
And	0x62
Xor	0x63

Operaciones de movimiento	Descripción	Código
cmovxx	mueve un dato de rA a rB de manera condicional	0x2(fn)
irmovq	mueve un dato de forma inmediata a un rB	0x30
rmmovq	mueve un dato de rA a memoria D(rB)	0x40
mrmovq	mueve un dato de memoria D(rA) a rB	0x50

ifun (para jxx y cmovxx)	Descripción	Código
-----	No hay condición (jmp, rrmovq)	0x0
le	lower/equal	0x1
l	lower	0x2
e	equal	0x3
ne	not equal	0x4
ge	greater/equal	0x5
g	greater	0x6

Etapas:

1. **Fetch:** Se interpretan los bytes de la instrucción de memoria extrayéndose operación, registros, etc. necesarios para la traducción.
2. **Decode:** Se leen los datos designados en la etapa Fetch del banco registros.
3. **Execute:** La ALU realiza operaciones con los datos obtenidos en la etapa Decode. Se setean los condition codes (CC).
4. **Memory:** Se realizan las lecturas o escrituras en memoria requeridas por la instrucción.
5. **Write Back:** Se actualizan los datos del banco de registros.
6. **PC update:** Se setea al PC con la dirección de la siguiente instrucción.

	Cómputo	Op rA, rB	Descripción
Fetch	icode, ifun rA, rB valC valP	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valP <- PC + 2	Lee el byte de instrucción Lee el byte de registros Lee el valor constante Calcula el siguiente PC
Decode	valA, srcA valB, srcB	valA <- R[rA] valB <- R[rB]	Lee el operando A Lee el operando B
Execute	valE Cond. Code	valE <- valB OP valA Establece CC	Realiza una operación en la ALU Usa/Modifica el registro CC
Memory	valM	valM <- M[valX]	Lectura/Escritura de la memoria

Write Back	dstE dstM	$R[rB] \leftarrow valE$	Guarda/Usa el resultado de la ALU Guarda el resultado de la memoria
PC Update	PC	$PC \leftarrow valP$	Actualiza el PC

Implementación segmentada

- Se basa en la división del proceso en etapas que se hacen de manera independiente; se mueve una instrucción a través de etapas y en todo momento se están procesando varias instrucciones.
- Parámetros:
 - **Latencia (ps, ns, s):** Tiempo requerido para ejecutar una sólo instrucción de principio a fin
 - **Delay (ps, ns, s):** Tiempo requerido en el procesamiento de una etapa determinada.
 - **Throughput (GIPS):** Cantidad de instrucciones procesadas por unidad de tiempo.
 - **Frecuencia del reloj (Hz, GHz):** Cantidad de ciclos por unidad de tiempo.
 - **Periodo del reloj:** duración de un ciclo del reloj.

Para realizar una instrucción requiero de 1 ciclos de reloj. Un ciclo del reloj son 1 ms.

- La latencia queda limitada por la etapa más lenta de la instrucción.
- Se requiere segmentar en etapas balanceadas.
- Si se fragmenta en bloques muy chicos, el delay por actualización de registro se torna un factor limitante.
- Las etapas de la implementación segmentada son iguales a las de la implementación secuencial, excepto que:
 - En este caso son físicas (no conceptuales).
 - La última etapa es la de Write back, y la actualización del PC se realiza en la etapa de Fetch que se caracteriza ahora por:
 - 1. Selección del PC actual**
 - 2. Lectura de instrucción**
 - 3. Cálculo del PC incrementado con predicción**
- Se introducen registros de pipeline que guardan valores intermedios; reciben valores de una etapa previa y con el clock se actualizan para ser pasados a la siguiente etapa.
- Los valores se propagan de una etapa a la siguiente sin saltarse etapas.
- Si hay error, se propaga hasta la etapa WB (última etapa) dónde se actualiza el estado del procesador.

Riesgos

- **Riesgos estructurales:** Surgen de conflictos en los recursos, cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones en ejecuciones simultáneas.
- **Riesgos por dependencia de datos:**
 - **RAW:** Cuando se quiere guardar en un registro un dato sea un valor constante (obtenido en etapa Decode) o calculado (obtenido en Execute). El dato se requiere en la etapa Write Back para actualizar el registro, y puede obtenerse mediante realimentación desde Decode o Execute.
 - **Load/Use:** Cuando se quiere guardar en un registro un dato que debe obtenerse de la etapa Memory (en Execute sólo tengo disponible el registro donde voy a guardar ese valor). El dato se requiere en la etapa Write Back para actualizar el registro y no puede obtenerse mediante realimentación. No puede salvarse la dependencia porque el dato se requiere en la etapa Decode, por lo que, deben utilizarse burbujas para retrasar la ejecución hasta poder obtenerse el dato ya sea mediante realimentación o hasta que la instrucción que obtendrá el dato de memoria termine su ejecución.
- **Riesgos de control:** Se producen de la segmentación de los saltos y otras instrucciones que cambian el PC de modo que la pipeline no predice correctamente. Esto ocurre debido a que el predictor del pipeline ordena que siempre se salte.
 Un **salto condicional** posee una condición que es propiciada por los condition codes (CC) que provee la ALU cuando la instrucción anterior finaliza su etapa Execute, lo cual provoca que el salto también debe confirmar si el salto fue dado correctamente en su pasaje por la etapa Execute. En consecuencia, el pipeline tomará inevitablemente dos instrucciones y las procesará hasta que la condición de salto sea verificada (o no). En caso de ocurrir un fallo en la predicción, se insertan burbujas en la etapa decode y execute, provocando el borrado de dichas instrucciones, e ingresando la nueva instrucción correcta al pipeline, producto de un fall through.
 Otro caso es el de **ret**, ya que se requiere de la dirección de memoria a la que realizar el salto, y ésta recién se obtiene en la etapa Memory, es decir, tampoco se sabe en Decode. En este caso, la instrucción siguiente al ret (que entra por fall through) es demorada en la etapa fetch y se insertan burbujas en las siguientes etapas, hasta que ret llega a la etapa memory, y la nueva dirección de memoria obtenida en dicha etapa actualiza el PC, permitiendo el ingreso de la instrucción correcta.

Técnicas para resolver riesgos

- **Stall:** Se mantiene la instrucción en la etapa Decode, insertando burbujas (retrasando la ejecución) para “esperar” a la obtención del dato requerido.
 - **Caso error de predicción en salto condicional:** Si se detecta un error en la etapa Execute se reemplazan las instrucciones que se deben cancelar en Execute y Decode por burbujas.

- **Caso return:** Recién en la etapa Memory se sabrá que hay que cortar la ejecución, por lo que, se utilizan tantas burbujas como para alcanzar la etapa Memory de ret, donde puede realizarse un envío de datos hacia atrás.
- **Nops:** Se insertan instrucciones que “no hacen nada” permitiendo retrasar la necesidad de obtención de un dato, lográndose que cuando se llegue a la instrucción que requiere cierto dato, éste ya pueda obtenerse.
- **Data forwarding:** Se envía el valor de la etapa de generación directamente a la etapa Decode o a la que lo requiera mediante circuitos de realimentación. Esta técnica **no** resuelve las dependencias Load/Use.
- **Load forwarding:** Ocurre en muy pocos casos. Dado que en ciertos casos es posible que una instrucción no requiera de un dato (decodeado) hasta la etapa memory, se agrega una realimentación entre la etapa Memory y Execute que permite reenviar un dato cargado desde memoria a la instrucción anterior que lo precise.

Implementación con paralelismo

Ejecución de más de una instrucción a la vez.

Instruction-Level Parallelism (**ILP**) se refiere a ejecutar más de una instrucción a la vez, en paralelo.

El mejor CPI (Ciclo Por Instrucción) que podemos obtener es 1, un ciclo por instrucción. En definitiva, queremos mejorar el rendimiento del procesador, que medimos con el tiempo de respuesta. Si bien hay varias técnicas, nos centraremos en 2:

Pipelines más profundos:

más etapas en la segmentación → más instrucciones en el datapath a la vez
 menos trabajo por etapa → ciclo de reloj más corto
 disminuye el tiempo de respuesta, manteniendo el CPI ideal en 1

Emisión múltiple (multiple issue):

replicando el hardware de la pipeline → múltiples pipelines
 comienzo de múltiples instrucciones por ciclo
 $CPI (ideal) < 1 \rightarrow$ se usa IPC

Por ejemplo: 4 GHz 4-way multiple-issue

16 BIPS, mejor CPI = 0.25, mejor IPC = 4

pero dependencias terminan reduciendo estos valores en la práctica

Emisión Múltiple

si queremos reducir todavía más el CPI, necesitamos que por cada ciclo de reloj se termine más de una instrucción, y los procesadores que lo logran se llaman de emisión múltiple (**multiple issue**).

Supongamos que la cantidad máxima de instrucciones que se pueden emitir en un ciclo de reloj es m , entonces se dice que el procesador es de m -vías, o en inglés, ***m-issue wide***.

Hay dos categorías de procesadores de emisión múltiple, que si bien difieren mucho en el hardware, y vamos a ver varias diferencias, parten de una idea fundamental **en la división del trabajo**. Esta división, entre el compilador y el hardware, está dada por **cómo se toma la decisión para la emisión de las instrucciones** y define las categorías de emisión múltiple.

- Si la decisión se hace en tiempo de compilación, es estática, y el procesador es de *emisión múltiple estática* (**static multiple issue**).
- Si la decisión se hace mientras se ejecuta el programa, es dinámica (la está tomando el hardware), y el procesador es de *emisión múltiple dinámica* (**dynamic multiple issue**).
- **Emisión estática:**
 - El compilador organiza las instrucciones para ser emitidas juntas
 - Se agrupan las instrucciones en *espacios de emisión* (**issue slots**)
 - Es el compilador quien detecta y evita los riesgos del pipeline
 - Todo lo que puede
- **Emisión dinámica:**
 - La CPU examina el flujo de instrucciones y decide qué instrucciones se emiten en cada ciclo
 - El compilador puede haber ayudado organizando las instrucciones de manera propicia
 - La CPU resuelve los riesgos usando técnicas más avanzadas en tiempo de ejecución

Especulación

la *especulación* (**speculation**) les permite al procesador o al compilador “adivinar” sobre las propiedades de una instrucción. Esta especulación permite que comiencen instrucciones que en caso contrario tendrían que esperar los resultados, ciertos, de instrucciones previas

- La idea básica detrás del mecanismo **es comenzar con la siguiente instrucción lo antes posible**. Una vez que se realiza la especulación, es necesario considerar que la misma puede no ser correcta. Si fue correcta, se completa la operación. Si fue incorrecta, se debe deshacer o descartar lo procesado y hacer lo correcto

Tanto los procesadores de emisión múltiple dinámica como los de emisión múltiple estática pueden hacer uso de la especulación, y nuevamente la diferencia está en dónde se hace (y cómo se resuelve)

Emisión Múltiple Estática

Estos procesadores trasladan el trabajo al compilador. Es este último quien se encarga de empaquetar las instrucciones y analizar los riesgos, en función de los recursos disponibles en el pipeline

Se puede pensar que las instrucciones de estos procesadores, para un determinado ciclo de reloj, forman una única gran instrucción, este conjunto de instrucciones empaquetadas se llama *paquete de emisión* (**issue packet**). Además, el conjunto de instrucciones que se pueden empaquetar juntas es **limitado** por lo que se puede analizar pensando en una única instrucción con varios campos para las distintas operaciones que se enviarán al pipeline.

Very Long Instruction Word (VLIW). También se los conoce como procesadores de instrucciones de paralelismo explícito (**EPIC, Explicitly Parallel Instruction Computing**), a pesar de no ser iguales.

planificación (scheduling)

- el compilador es responsable de los riesgos y debe eliminarlos (tal vez no completamente)
- reordena las instrucciones y las agrupa en paquetes.

Los paquetes **no** pueden tener dependencias internas se las debe haber eliminado por reordenamiento y/o la inserción de uno o más nop

Sin embargo, el procesador puede tener capacidad para detectar riesgos y retener paquetes, pero lo hace entre paquetes. Puede estar implementado, como no, en caso de que no el peso recae en el compilador. El compilador debe saber para quién compila.

Como las instrucciones se ejecutan **en el mismo momento** puede haber problemas con las dependencias *Load/Store*. Ya que se realizan en simultaneo no se puede esperar un ciclo

- **Antidependencia o dependencia por nombres:** se da cuando se fuerza el código a tener cierto orden debido a la reutilización de algún nombre, típicamente, un registro.
- **Renombrado de registros (*register renaming*):** es el uso de registros adicionales para evitar las antidependencias

Emisión Múltiple Dinámica

Los procesadores de emisión múltiple dinámica también son llamados (procesadores) **superescalares**. En cada ciclo de reloj es el procesador quien decide cuántas instrucciones se emiten, entre cero, una o más. Al hacerlo, puede evitar riesgos estructurales o por dependencias de datos

A diferencia de los procesadores VLIW, el hardware garantiza que el código se ejecuta correctamente

ORAL

Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration

donde cada iteración

calcula dos elementos de la suma del prefijo, reduciendo a la mitad el número total de

iteraciones requeridas

First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation

Existe una técnica interesante llamada blocking que puede mejorar la ubicación temporal de los bucles internos.

La idea general del bloqueo es organizar las estructuras de datos de un programa en grandes fragmentos llamados bloques

El programa está estructurado para que cargue un fragmento en la caché L1, haga todas las lecturas y escrituras que

necesita en ese fragmento, luego descarta el fragmento, carga en el siguiente fragmento, y así sucesivamente.

A translation lookaside buffer (TLB) is a memory [cache](#) that is used to reduce the time taken to access a user memory location. It is a part of the chip's [memory-management unit](#) (MMU). The TLB stores the recent translations of [virtual memory](#) to [physical memory](#) and can be called an address-translation cache

