FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Resumen Organización Del Computador

Curso Santi
Primer cuatrimestre de 2021

Alumno:	VAZQUEZ LAREU, Román
Número de padrón:	100815
Email:	rlareu@fi.uba.ar

Índice

1. Programación MIPS y performance

Priorizar favorecer el caso frecuente.

1.1. Ley de Amdahl

SpeedUp: ganancia en tiempo de ejecución.

$$SPup_G = \frac{1}{1 - f_L + \frac{f_L}{SPup_L}} = \frac{t_v}{t_n} \tag{1}$$

Generalizada:

$$SPup_G = \frac{1}{1 - \sum_i f_L i + \sum_i \frac{f_{Li}}{SPup_{Li}}} \tag{2}$$

Para este caso tener en cuenta que no se usan en simultáneo ni solapadas

Tiempo de la CPU: ciclos de reloj del CPU para el programa dado por la cantidad de tiempo que conlleva cada ciclo (tiempo de reloj).

IC: instrucciones del procesador dinamicamente ejecutadas.

CPI: ciclos por instruccion.

$$CPI = \frac{\text{Ciclos}}{IC} \tag{3}$$

$$\text{tiempo de CPU} = IC \times CPI \times T_{CLK} \tag{4}$$

$$Frec_{CLK} = \frac{1}{T_{CLK}}[Hz] = [\frac{1}{\text{s}}] \tag{5}$$

$$\frac{\text{Instrucciones}}{\text{Programa}} \times \frac{\text{Ciclos de reloj}}{\text{Instruccion}} \times \frac{\text{Segundos}}{\text{Ciclo de reloj}} = \frac{\text{Segundos}}{\text{programa}} = \text{Tiempo de CPU} \tag{6}$$

MIPS: millones de instrucciones por segundo

$$MIPS = \frac{IC}{T_e \times 10^6} = \frac{Frec_{CLK}}{CPI \times 10^6} = \frac{\text{IC} \times Frec_{CLK}}{\text{ciclos} \times 10^6} \tag{7}$$

CPI efectivo es un promedio ponderado, donde se tiene en cuenta el CPI promedio por instruccion o por conjunto de instrucciones ponderado/pesado por el porcentaje de tiempo de ejecucion

$$CPI_{\text{conjuntos}} = \sum_i f_i \times CPI_i \tag{8}$$

$$CPI_{\text{efectivo}} = CPI_{\text{ideal}} + \frac{\text{referencias a memoria}}{IC} \times \frac{1}{T_{CLK}} \times t_{acceso} \tag{9}$$

De derecha a izquierda: paso de tiempos de acceso a ciclos de acceso, luego a ciclos promedio para accesos.

Siempre

$$\frac{\#\text{mem ref instr.}}{IC} = 1$$

, el resto es load store. De esta manera,

$$\frac{\#\text{mem ref}}{IC} = \frac{\#\text{mem ref instr}}{IC} + \frac{\#\text{mem ref dato}}{IC}$$

$$T_e = \text{ciclos} \times T_{CLK} = \frac{CPI \times IC}{Frec_{CLK}} \tag{10}$$

MFLOPS: millones de operaciones de punto flotante por segundo.

Ecuación de desempeño de CPU con Memoria Cache

$$\text{Tiempo de CPU} = \text{IC} \times (CPI_{ejecucion} + \text{Miss Rate} \times \frac{\text{Accesos a memoria}}{\text{Instruccion}} \times \text{Penalidad de miss}) \times T_{CLK} \tag{11}$$

1.2. ABI

Los siguientes registros **deben** ser salvados por la funcion llamada si esta los va a modificar:

- ra,sp,fp (o s8), gp, s0...s7
- f2 a f30

No se garantiza la preservación del resto de los registros entre llamados. Si la función llamadora los quiere preservarlos, debe salvarlos en su stack frame. Los valores se devuelven por v0-v1

1.3. Stack Frame

Cada funcion crea su stack Frame. Este se compone de areas de un tamaño multiplo de 8 bytes, alineadas a 8 bytes. Las áreas, de abajo hacia arriba:

- Argument Building Area (ABA): al menos 16 bytes. Los primeros 4 argumentos se guardan en a0-a3, el resto a partir de los primeros 16 bytes. Los argumentos pasados en a0-a03 son almacenados por la función llamada.
- Local and Temporary Variables Area (LTVA)
- Floating Point Registers Save Area (FPRSA)
- General Register Save Area(GRSA, es obligatoria): siempre se salvan el fp y gp. Si es non-leaf, también el ra.

2. Jerarquia de memorias

Las memorias cache almacenan la info en **bloques o líneas**. El **tamaño de bloque** es la minima unidad de transferencia desde el siguiente nivel.La memoria cache contiene **copias** de los bloques de memoria principal.

Si lo buscado por CPU está en cache, entonces se considera un **Hit**. Si no está, es **Miss** y se produce una **penalidad**

$$\textbf{Capacidad de memoria Cache} = \text{Tamaño de bloque} \times \text{Cantidad de bloques} \tag{12}$$

La asociatividad define cómo se oraniza la memoria y cómo se mapean las distintas direcciones a cada bloque. Las direcciones se dividen en dos campos:

- **Offset**: permite elegir el byte dentro del bloque, por lo tanto la determina el tamaño del bloque.
- **Memory Block Addres**: determina, de acuerdo a la asociatividad, que bloque puede cachear dicho MB.

Dirección de memoria de **N bits** y un tamaño de bloque de 2^F **bytes**.

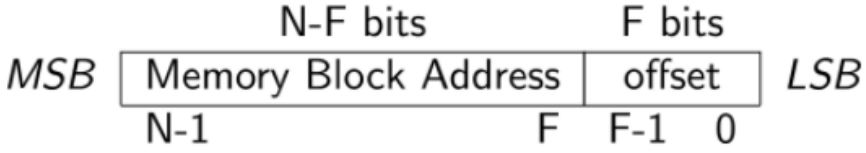


Figura 1: Direccion de memoria

El Tamaño de bloque define la cantidad de bits del offset y, en consecuencia, la cantidad de bits restantes para el MBA.

Relación entre bloques y conjuntos segun asociatividad:

- **DM**: un bloque por conjunto, solo puede ser cacheado en un bloque. El mapeo es 1 a 1 entre bloques y conjuntos
- **N-WSA**: N bloques por conjunto, cada direccion de memoria puede ser cacheada en N lugares distintos dentro del conjunto.
- **FA**: todos los bloques pertenecen a un unico conjunto, no es necesario el indice ya que no discrimino donde caigo

2.1. Cache totalmente asociativa (*Fully associative: FA)*

El bloque de memoria principal puede almacenarse en cualquier bloque de memoria cache

En este caso el **TAG** es el numero de bloque correspondiente en memoria principal. Todos los bloques pertenecen a un unico conjunto, no es necesario un índice. Todas las direcciones mapean a cualquier bloque.

Memory Block Address	Offset
TAG	Offset
30	2

Figura 2: Direccion de memoria para cache FA con lineas 8 de 32 bits (4 bytes)

TAG	Data			
TAG 0	B0	B1	B2	B3
TAG 1	B0	B1	B2	B3
TAG 2	B0	B1	B2	B3
TAG 3	B0	B1	B2	B3
TAG 4	B0	B1	B2	B3
TAG 5	B0	B1	B2	B3
TAG 6	B0	B1	B2	B3
TAG 7	B0	B1	B2	B3

Figura 3: Estructura de FA descripta en figura 2

2.2. Cache por Correspondencia Directa (*Directed Map: DM)*

El bloque de memoria principal puede alojarse en un solo bloque de memoria cache. De esta manera se tiene un bloque por conjunto, quedando la cantidad de bloques igual a la cantidad de conjuntos

$$\text{Nro. de bloque en cache} = \text{Nro de bloque en principal} \quad mod \quad \#\text{bloques en cache} \tag{13}$$

El **MBA** se divide en dos partes:

- **TAG**
- **INDICE**

IDX	TAG	Data			
0	TAG 0	B0	B1	B2	B3
1	TAG 1	B0	B1	B2	B3
2	TAG 2	B0	B1	B2	B3
3	TAG 3	B0	B1	B2	B3
4	TAG 4	B0	B1	B2	B3
5	TAG 5	B0	B1	B2	B3
6	TAG 6	B0	B1	B2	B3
7	TAG 7	B0	B1	B2	B3

Figura 4: Direccion de memoria para cache DM con 8 lineas de 32 bits (4 bytes) y 8 conjuntos

La cantidad de bits del indice se corresponden con la cantidad de conjuntos a direccinar. De esta manera quedan definidos los bits del TAG por descarte. Una vez en la memoria cache, comparo por TAG.

Memory Block Address		Offset
TAG	IDX	Offset
27	3	2

Figura 5: Estructura de DM descripta en figura 3

2.3. Cache asociativa por conjuntos (*N-WSA)*

El bloque de memoria principal puede alojarse en un solo conjunto de memoria cache. Los conjuntos agrupan los bloques dentro de la cache según asociatividad.

$$\text{Nro. de conjunto en cache} = \text{Nro. de bloque en principal} \quad mod \quad \#\text{Conjuntos en cache} \tag{14}$$

$$\#\text{conjuntos} = \frac{\#\text{bloques}}{\#\text{vias}} \tag{15}$$

Memory Block Address		Offset
TAG	IDX	Offset
28	2	2

Figura 6: Direccion de 2WSA de 8 lineas de 32 bits (4 bytes)

	WAY A					WAY B				
IDX	TAG	Data				TAG	Data			
0	TAG 0WA	B0	B1	B2	B3	TAG 0WB	B0	B1	B2	B3
1	TAG 1WA	B0	B1	B2	B3	TAG 1WB	B0	B1	B2	B3
2	TAG 2WA	B0	B1	B2	B3	TAG 2WB	B0	B1	B2	B3
3	TAG 3WA	B0	B1	B2	B3	TAG 3WB	B0	B1	B2	B3

Figura 7: Estructura de 2WSA de figura 6

2.4. Politicas de escritura

- **Write-Through**: la información se escribe al cache y al siguiente nivel de la jerarquía
- **Write-Back**: la información se escribe solo al cache. El bloque modificado se escribe al siguiente nivel solo ante un reemplazo.Antes de que un bloque sea desalojado de memoria cache debido a un miss, este se copia a memoria principal.
  - **Dirty Bit**: flag que indica que el bloque de cache fue modificado por un write (sucio, su valor es 1) o no (limpio, su valor es 0). Ante un reemplazo, previamente se transfiere el bloque de cache al siguiente nivel jerárquico solo cuando está sucio. Ahorra transferencia al momento del reemplazo

2.5. Estrategias ante write miss

Al escribir en cache, pueden darse dos situaciones:

- la direccion a la que queremos escribir esta cargada en cache (***Write Hit***, entonces escribimos el dato en cache. Quedarian la cache y la principal con valores distintos. Si es WT, a continuacion se escribe en memoria principal. Si es WB, se coloca el DB en 1, y unicamente se modificará en memoria principal ante un reemplazo.
- la dirección a la que queremos escribir no se encuentra cargada en cache. En este caso se da un ***write miss***. Para resolver esta cuestion existen diversas estrategias:
  - **Write Allocate**: el bloque faltante es asignado y se continua como un write hit. Se comporta de la misma manera que un read miss. De esta manera se escribe en memoria principal y se carga el valor actualizado en cache.
  - **Write No Allocate**: no se modifica el cache. El bloque se modifica en el siguiente nivel jerárquico. De esta manera, se escribe el valor en memoria principal y no se modifica el cache. Solo es aplicable bajo políticas write through.

2.6. Write Back, Write Allocate *WB-WA*

Si **Write Hit**: Escribe unicamente en cache Si **Write Miss**:

2.7. Write Through, Write Allocate *WT-WA*

Si **Write Hit**: Escribe en cache y en siguiente nivel de jerarquia Si **Write Miss**:

2.8. Write Back, Write No Allocate *WB-WNA*

Si **Write Hit**: Escribe unicamente en cache Si **Write Miss**:

2.9. Write Through, Write No Allocate *WT-WNA*

Si **Write Hit**: Escribe en cache y en siguiente nivel de jerarquia Si **Write Miss**:

3. Memoria Virtual

3.1. Virtual Address, Physical Address y Page Table

Esta memoria hace referencia a lo que ve el programa, mientras que la memoria fisica es el hardware. Entonces lo que usa el programa es Virtual Addresses (en MIPS serían de 32 bits). Physical Addres (PA) es la direccion final usada por el hardware. De esta manera el programa carga una dirección virtual, la CPU la traduce a una PA (Physical address). Esto lo hace a través de una tabla de paginacion que mapea direcciones virtuales a físicas. Si la dirección buscada no está en memoria, la carga desde el disco. Una vez leida desde memoria, devuelve la data al programa. Si la tabla es lineal, preciso una PTE (Page Table Entry) por cada direccion virtual. Esto ocupa mucho espacio, entonces se pueden mapear rango de VA a rangos de PA. Entonces cada PTE cubre el tamaño de pagina virtual (i.e 4kb).

La *Virtual Address* se encuentra compuesta por la *VPN (Virtual page number)* y el *Offset*. La *VPN* se traduce, y el *Offset* se concatena a la traducción. Obteniendo el *Physical page number (PPN)* y el *Offset*. Si se supone un tamaño de pagina virtual de 4kb, enttonces cada PTE maneja 4096 direcciones. De esta manera se necesitarían 12 bits para desplazarse dentro de la pagina. Esos 12 bits conformarán el offset y no serán traducidos.

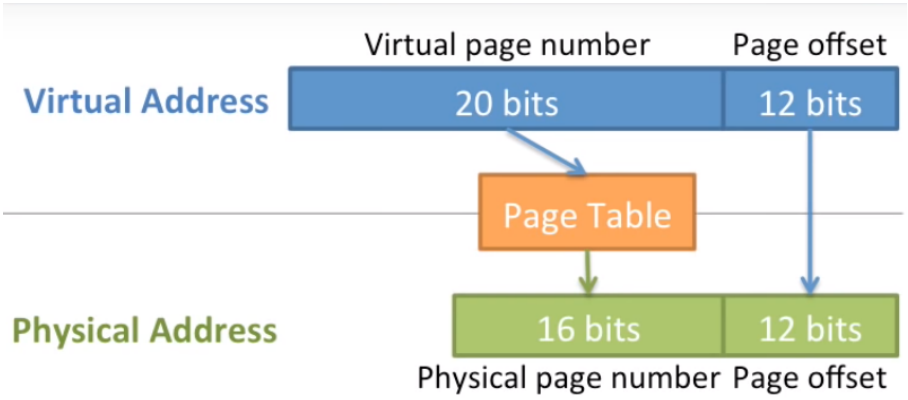


Figura 8: Traduccion de VA a PA, para paginas de 4kb (12 bits de offset para direccionar) y 256MB de RAM (28 bits para direccionar, 12 de offset entonces 16 de PPN)

El tamaño de la VA está determinado por la ISA. Mientras que el de la PA por la cantidad de RAM. Para determinar el offset, se tiene en cuenta el tamaño de pagina virtual a direccionar. El tamaño del offset es el mismo para la PA y VA. De esta manera, se ingresa a la PT con la VPN, y se sale con una PPN que luego se concatena con el offset.

Si la VPN no está en memoria y está en disco, nos damos cuenta porque la PTE apunta al disco. En este caso, la CPU genera una **Page Fault Exception**, la cual es tomada el **OS Page Fault**

Handler. Asi, el sistema elimina una entrada de RAM y carga la página para luego actualizar el valor en la PT.

Cada programa ejecutandose precisa su propia Page Table, lo cual ocuparía muchisimo espacio en memoria. Asi surgen las PT jerárquicas.

3.2. TLB

Para evitar ir a la tabla de traducciones, lo que implica varios acceso a memoria, está la *Translation Lookaside Buffer (TLB)*, la cual contiene las traducciones de algunas paginas virtuales. De hacer Hit en la TLB, no es necesario ir a la tabla de paginas.

Si las direcciones que llegan a la memoria cache ya han sido traducidas, entonces se tiene una cache direccionada por direcciones físicas

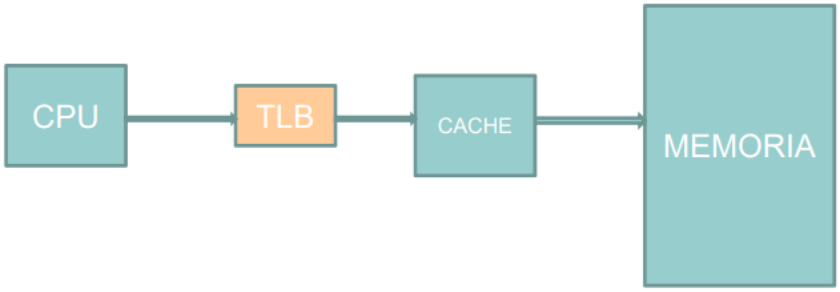


Figura 9: Cache fisicamente direccionada

Si las direcciones que llegan a la memoria cache no han sido traducidas, entonces se tiene una cache direccionada por direcciones virtuales

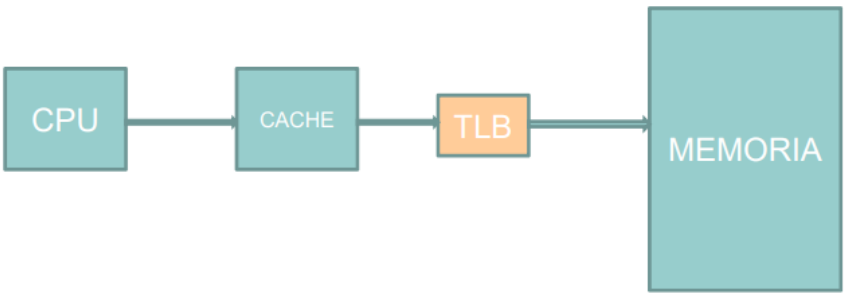


Figura 10: cache virtualmente direccionada

Ante una Direccion virtual. Tomo la VPN y la comparo con el TAG de la TLB. Si es hit, leo la PPN y la concateno con el offset, obteniendo la PA (Physical Address). Si es miss en TLB, la voy a buscar a la PT (page table), que implica más accesos a memoria principal. Suponiendo una tabla de paginacion lineal, esto no es más que acceder con la VPN como índice y extraer la PPN. Si es una tabla de paginación jerárquica, este procedimiento cambia.

4. Ejercicios de parcial

4.1. Cache

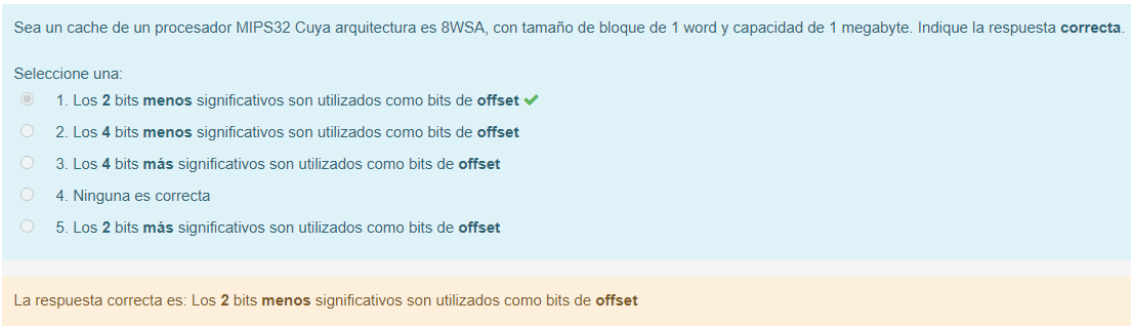


Figura 11: Ejercicio Cache

La cache es MIPS 32, por lo tanto maneja direcciones de 32 bits. La asociatividad es de 8WSA (8 vias), el tamaño de bloque es de 1 word (en MIPS32 un word son 32 bits, es decir 4 bytes) y la capacidad es de 1 megabyte (10^6 bytes). El offset se utiliza para determinar el byte dentro del bloque. Cada bloque es de 4 bytes, entonces preciso 2 bits para direccionarlos. Estos serán el bit 0-1, es decir los 2 LSB.

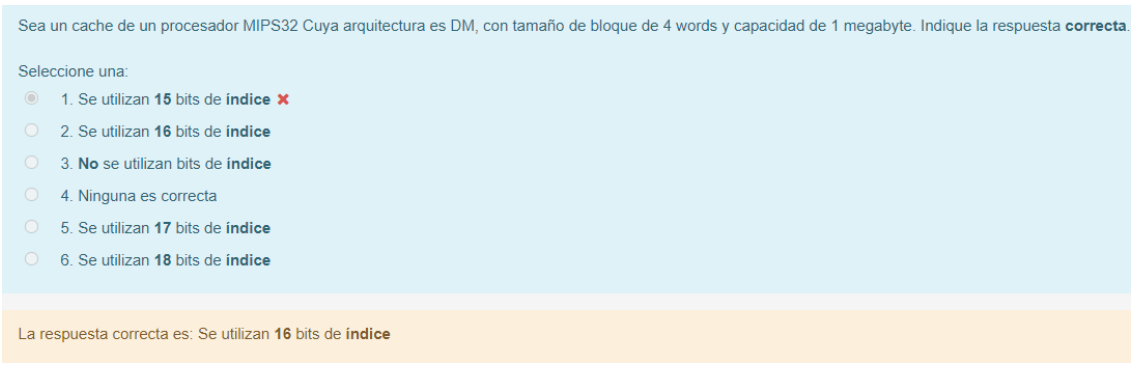


Figura 12: Ejercicio Cache

La cache es MIPS 32, por lo tanto maneja direcciones de 32 bits. Es DM, sabemos que la cantidad de bloques es igual a la cantidad de conjuntos. Ademas el MBA se encuentra dividido en TAG e INDICE. Esto concatenado al offset serán los 32 bits. El índice sirve para direccionar los conjuntos. Tamaño de bloque de 4 words, en MIPS32 cada word es de 32 bits (4 bytes). Entonces $4\text{bytes} \times 4 = 16\text{bytes}$ a direccionar en cada bloque. La capacidad es de 1 megabyte (10^6 bytes). Para todas las caches sabemos que

$$\text{Capacidad} = \text{Tamaño bloque} \times \#\text{bloques}$$

Entonces

$$10^6\text{bytes} = 16\text{bytes} \times \#\text{bloques}$$

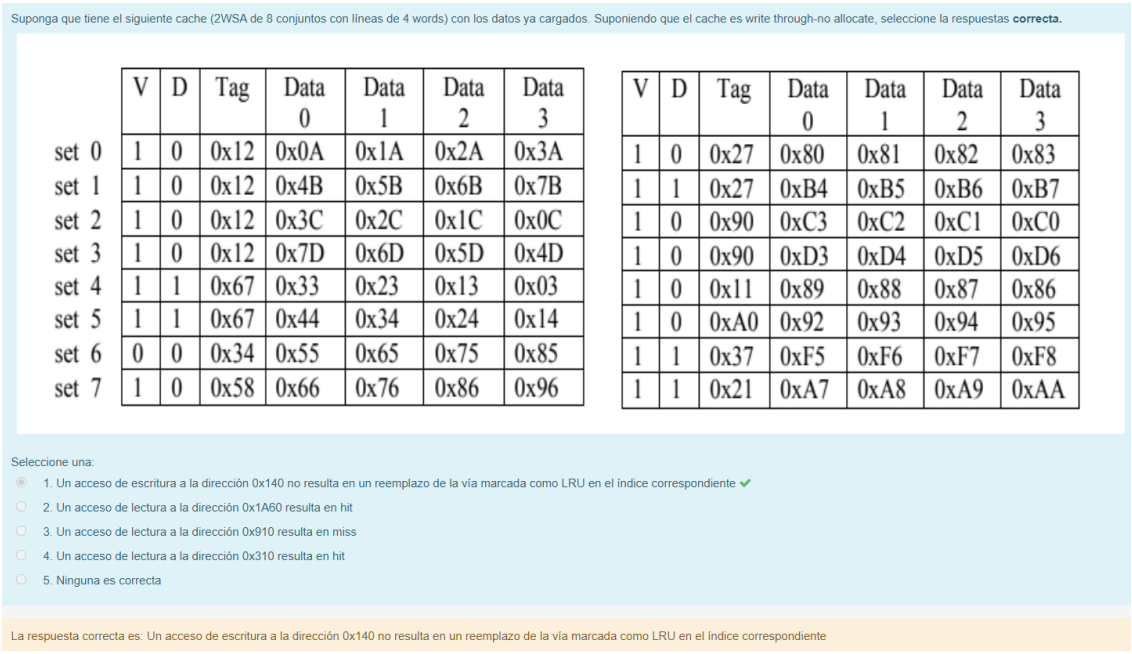
De esta manera

$$\#\text{bloques} = 62500$$

En las cache DM

$$\#\text{bloques} = \#\text{conjuntos} = 62500$$

Para direccionar 62500 conjuntos se precisan 16 bits ya que $2^{16} = 65536$



Suponga que tiene el siguiente cache (2WSA de 8 conjuntos con líneas de 4 words) con los datos ya cargados. Suponiendo que el cache es write through-no allocate, seleccione la respuestas **correcta.**

	V	D	Tag	Data 0	Data 1	Data 2	Data 3
set 0	1	0	0x12	0x0A	0x1A	0x2A	0x3A
set 1	1	0	0x12	0x4B	0x5B	0x6B	0x7B
set 2	1	0	0x12	0x3C	0x2C	0x1C	0x0C
set 3	1	0	0x12	0x7D	0x6D	0x5D	0x4D
set 4	1	1	0x67	0x33	0x23	0x13	0x03
set 5	1	1	0x67	0x44	0x34	0x24	0x14
set 6	0	0	0x34	0x55	0x65	0x75	0x85
set 7	1	0	0x58	0x66	0x76	0x86	0x96

V	D	Tag	Data 0	Data 1	Data 2	Data 3
1	0	0x27	0x80	0x81	0x82	0x83
1	1	0x27	0xB4	0xB5	0xB6	0xB7
1	0	0x90	0xC3	0xC2	0xC1	0xC0
1	0	0x90	0xD3	0xD4	0xD5	0xD6
1	0	0x11	0x89	0x88	0x87	0x86
1	0	0xA0	0x92	0x93	0x94	0x95
1	1	0x37	0xF5	0xF6	0xF7	0xF8
1	1	0x21	0xA7	0xA8	0xA9	0xAA

Seleccione una:

1. Un acceso de escritura a la dirección 0x140 no resulta en un reemplazo de la vía marcada como LRU en el índice correspondiente ✔
2. Un acceso de lectura a la dirección 0x1A60 resulta en hit
3. Un acceso de lectura a la dirección 0x910 resulta en miss
4. Un acceso de lectura a la dirección 0x310 resulta en hit
5. Ninguna es correcta

La respuesta correcta es: Un acceso de escritura a la dirección 0x140 no resulta en un reemplazo de la vía marcada como LRU en el índice correspondiente

Figura 13: Ejercicio Cache

Una Cache 2WSA de 8 conjuntos de 4 words. En las NWSA sabemos que

$$\#\text{conjuntos} = \frac{\#\text{bloques}}{\#vias}$$

Entonces

$$8 = \frac{\#\text{bloques}}{2}$$

Obteniendo

$$\#\text{bloques} = 16$$

. Las 4 words implican 16 bytes, es decir preciso 4 bits de offset para direccionarlos. A su vez preciso 3 bits de indice para direccionar los 8 conjuntos. El TAG entonces queda de 25 bits. Si el cache es WT-WNA entonces, la informacion se escribe simultaneamente en cache y en memoria si el bloque está en memoria (WT). Si es Write miss, por ser WNA no se modifica el cache, sino que se escribe en memoria principal unicamente.

- 0x140 =>**10 100** 0000 =>Indice = 100 y TAG = 10 =>Indice = 4 y TAG = 0x2. Si busco por este Set/TAG, obtengo un write miss.Acceso Escritura da como resultado un miss, por ser WNA escribo en principal y no modifico la cache, por lo tanto no hay un reemplazo
- 0x1A60 =>**110100 110** 0000 =>Indice = 110 y TAG = 11 0100 =>Indice = 6 y TAG = 0x34. Si busco por este Set/TAG, el bit Valid es 0, por lo tanto resulta en un miss ya que en esa entrada hay basura. Acceso Lectura
- 0x910 =>**10010 001** 0000 =>Indice = 001 y TAG = 1 0010 =>Indice = 1 y TAG = 0x12. Si busco por este Set/TAG, obtengo un hit.Acceso Lectura

- 0x310 =>**110 001** 0000 =>Indice = 001 y TAG = 110 =>Indice = 1 y TAG = 0x6. Si busco por este Set/TAG, obtengo un miss.Acceso Lectura

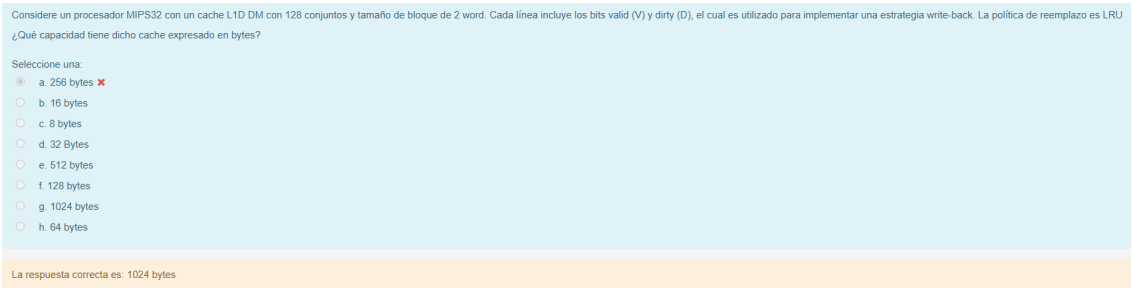


Figura 14: Ejercicio Cache

MIPS32 entonces direcciones de 32 bits, cache DM con 128 conjuntos y tamaño de bloque 2 word (8 bytes). En DM la cantidad de bloques es igual a la cantidad de conjuntos, teniendo un bloque por conjunto. Preciso 3 bits de offset para direccionar los 8 bytes. Preciso 7 bits para direccionar los 128 conjuntos. La cache es WB, entonces la información se escribe solo al cache. El bloque modificado se escribe al siguiente nivel solo ante un reemplazo. Sabemos que

$$\text{Capacidad} = \text{Tamaño bloque} \times \#\text{bloques}$$

entonces

$$\text{Capacidad} = 8 \times 128$$

obteniendo

$$\text{Capacidad} = 1024\text{bytes}$$

. No caer en el cazabobos de los bits de validez y WB, no son necesarios para este ejercicio

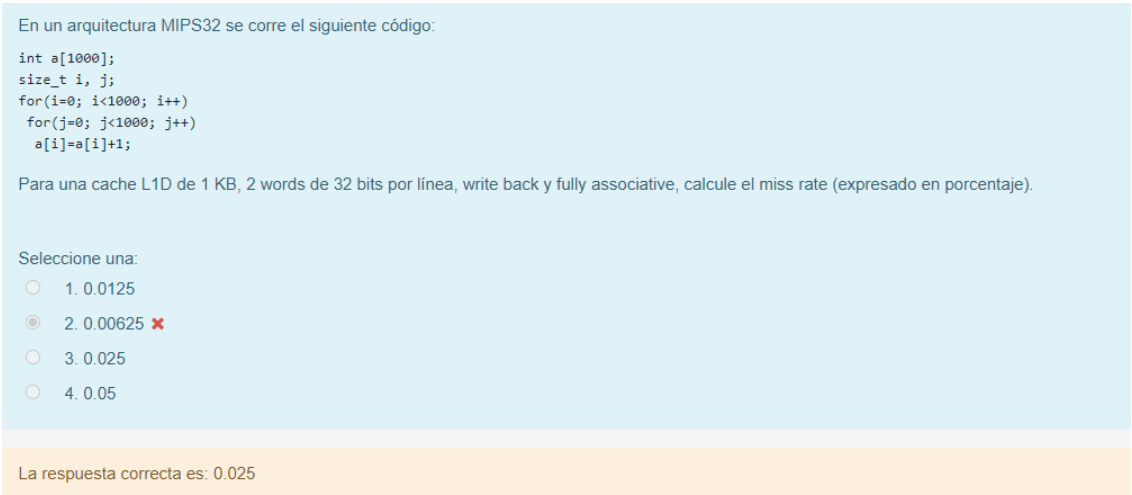
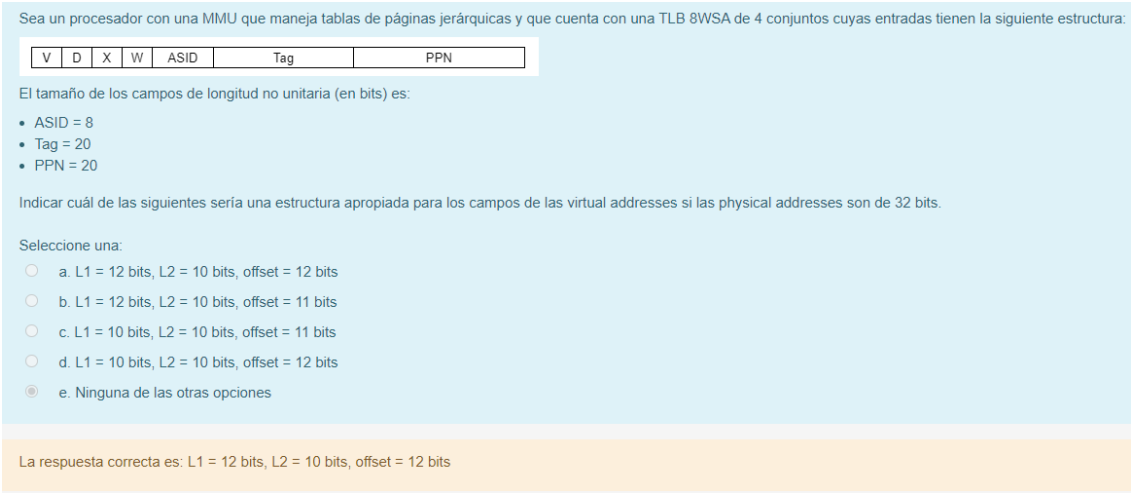


Figura 15: Ejercicio Cache

4.2. Memoria Virtual


Sea un procesador con una MMU que maneja tablas de páginas jerárquicas y que cuenta con una TLB 8WSA de 4 conjuntos cuyas entradas tienen la siguiente estructura:

V	D	X	W	ASID	Tag	PPN

El tamaño de los campos de longitud no unitaria (en bits) es:

- ASID = 8
- Tag = 20
- PPN = 20

Indicar cuál de las siguientes sería una estructura apropiada para los campos de las virtual addresses si las physical addresses son de 32 bits.

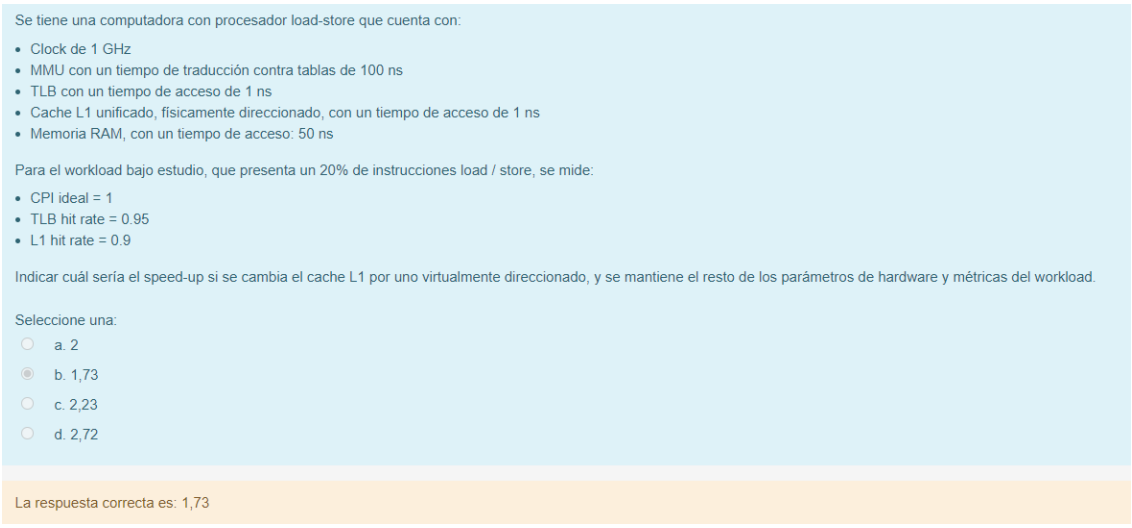
Seleccione una:

- a. L1 = 12 bits, L2 = 10 bits, offset = 12 bits
- b. L1 = 12 bits, L2 = 10 bits, offset = 11 bits
- c. L1 = 10 bits, L2 = 10 bits, offset = 11 bits
- d. L1 = 10 bits, L2 = 10 bits, offset = 12 bits
- e. Ninguna de las otras opciones

La respuesta correcta es: L1 = 12 bits, L2 = 10 bits, offset = 12 bits

Figura 16: Ejercicio Memoria Virtual

La TLB es un cache de traducciones. La PA es de 32 bits. PPN es de 20 bits. Entonces Offset es de 12 bits, por lo tanto tengo páginas de 4kb. La traduccion es de VPN a PPN. De los bits de la VPN, tienen que salir para L1 y L2. El TAG de la entrada de la TLB debe ser comparado contra una parte o toda la VPN, por lo que como minimo la VPN será de 20 bits. Si tengo 4 conjuntos, de los bits de la VPN, voy a usar 2 para direccionar los conjuntos (indice). Entonces si el TAG de la entrada es de 20 bits, la VPN deberá ser de 22 bits, ya que 2 serán el indice y los otros 20 para comparar con el TAG. De esta manera, la suma de el tamaño de L1 y L2 debe ser de 22 bits, y el offset de 12 bits.


Se tiene una computadora con procesador load-store que cuenta con:

- Clock de 1 GHz
- MMU con un tiempo de traducción contra tablas de 100 ns
- TLB con un tiempo de acceso de 1 ns
- Cache L1 unificado, físicamente direccionado, con un tiempo de acceso de 1 ns
- Memoria RAM, con un tiempo de acceso: 50 ns

Para el workload bajo estudio, que presenta un 20% de instrucciones load / store, se mide:

- CPI ideal = 1
- TLB hit rate = 0.95
- L1 hit rate = 0.9

Indicar cuál sería el speed-up si se cambia el cache L1 por uno virtualmente direccionado, y se mantiene el resto de los parámetros de hardware y métricas del workload.

Seleccione una:

- a. 2
- b. 1,73
- c. 2,23
- d. 2,72

La respuesta correcta es: 1,73

Figura 17: Ejercicio Memoria Virtual

20 % load/store, entonces $\frac{\text{referencias a memoria}}{IC} = 1,2$.

$$CPI_{\text{efectivo}} = CPI_{\text{ideal}} + \frac{\text{referencias a memoria}}{IC} \times \frac{1}{T_{CLK}} \times t_{acceso}$$

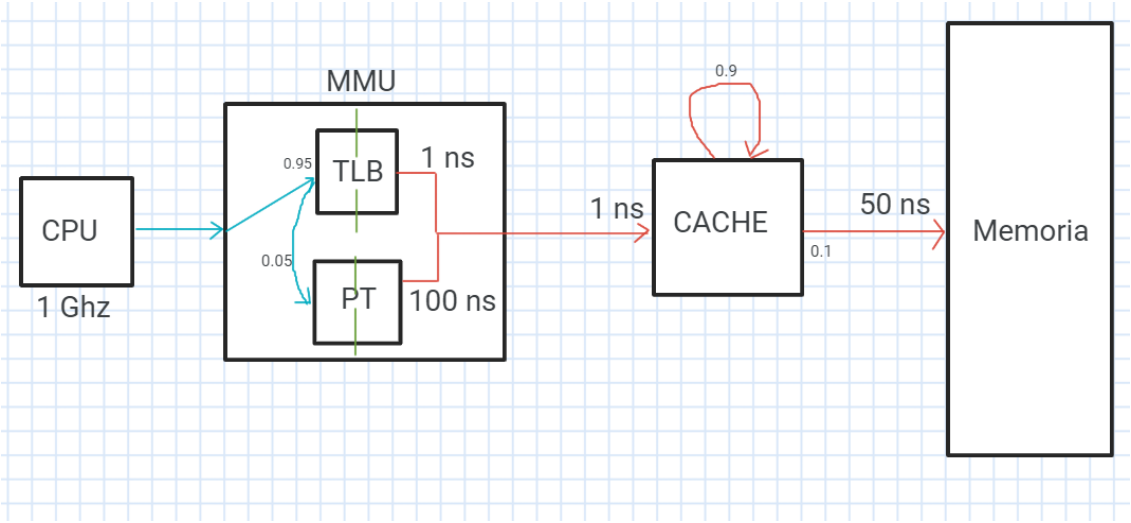


Figura 18: Disposicion para cache fisicamente direccionada

t_{acceso} será el tiempo de acceso promedio a memoria. Siempre voy a acceder a la TLB (despues puede ser hit o miss, pero siempre estará el tiempo de acceso). El 0.05 de las veces voy a tener que ir a la PT. Luego, una vez con la PA siempre voy a tener el tiempo de cache (ya sea hit o miss). De ser miss (0.1 de las veces), voy a tener que sumarle el tiempo de acceso a memoria.

$$t_{acceso} = t_{TLB} + mr_{TLB} * t_{PT} + t_{L1} + mr_{L1} * t_{t_memoria}$$
$$t_{acceso} = 1ns + 0{,}05 * 100ns + 1ns + 0{,}1 * 50ns = 12ns = 12 \cdot 10^{-9}s$$
$$CPI_{\text{efectivo}} = 1 + 1{,}2 \times 1 \cdot 10^{9}s^{-1} \times 12 \cdot 10^{-9}s = 15,4ciclos$$

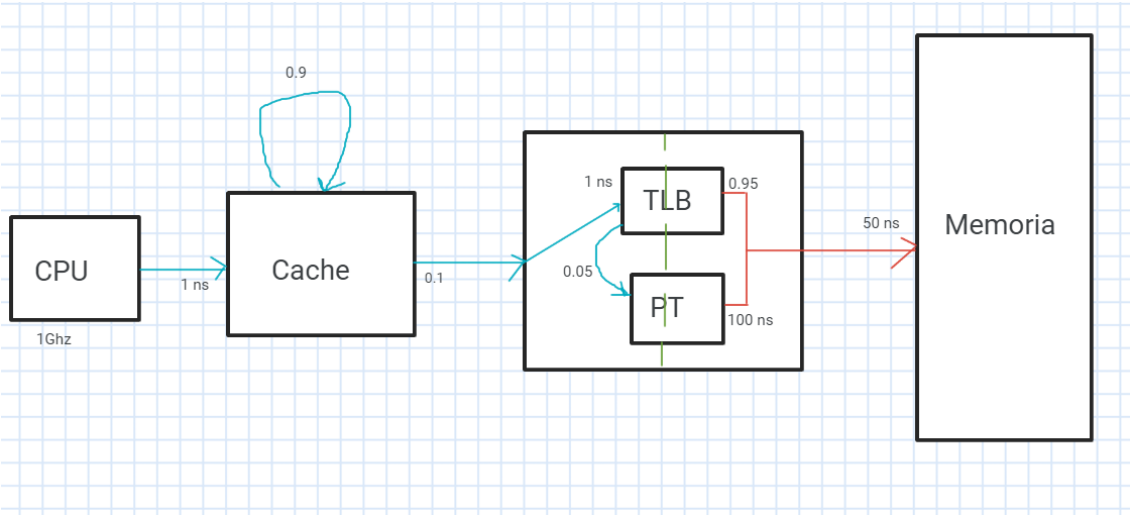
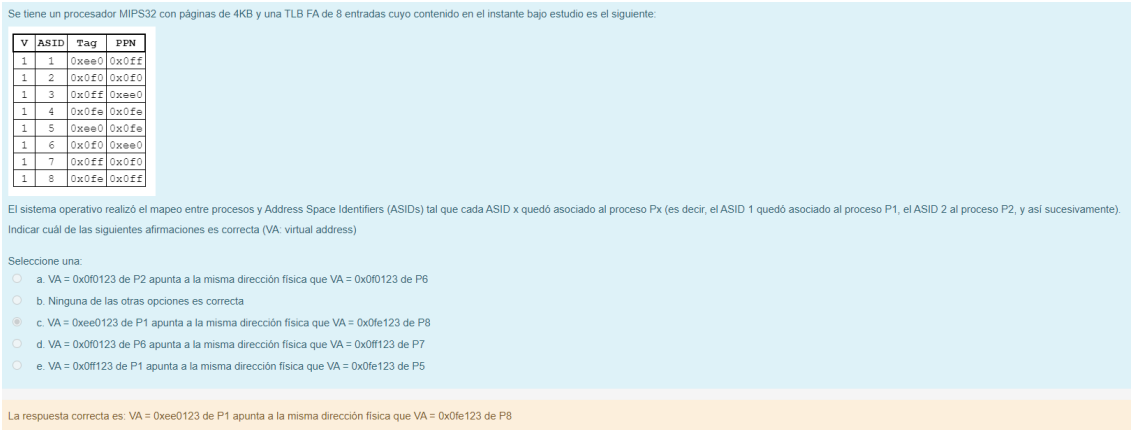


Figura 19: Disposicion para cache virtualmente direccionada

$$t_{acceso} = t_{Cache} + mr_{Cache} * [t_{TLB} + mr_{TLB} * t_{PT} + t_{memoria}]$$
$$t_{acceso} = 1ns + 0{,}1 * [1ns + 0{,}05 * 100ns + 50ns] = 6,6ns = 6,6 \cdot 10^{-9}s$$
$$CPI_{\text{efectivo}} = 1 + 1{,}2 \times 1 \cdot 10^{9}s^{-1} \times 6,6 \cdot 10^{-9}s = 8{,}92 ciclos$$

Finalmente

$$SP_{up} = \frac{CPI_{\text{efectivoViejo}}}{CPI_{\text{efectivoNuevo}}} = \frac{15{,}4}{8{,}92} = 1{,}73$$



Se tiene un procesador MIPS32 con páginas de 4KB y una TLB FA de 8 entradas cuyo contenido en el instante bajo estudio es el siguiente:

V	ASID	Tag	PPN
1	1	0xee0	0x0ff
1	2	0x0f0	0x0f0
1	3	0x0ff	0xee0
1	4	0x0fe	0x0fe
1	5	0xee0	0x0fe
1	6	0x0f0	0xee0
1	7	0x0ff	0x0f0
1	8	0x0fe	0x0ff

El sistema operativo realizó el mapeo entre procesos y Address Space Identifiers (ASIDs) tal que cada ASID x quedó asociado al proceso Px (es decir, el ASID 1 quedó asociado al proceso P1, el ASID 2 al proceso P2, y así sucesivamente).
Indicar cuál de las siguientes afirmaciones es correcta (VA: virtual address)

Seleccione una:

a. VA = 0x0f0123 de P2 apunta a la misma dirección física que VA = 0x0f0123 de P6
b. Ninguna de las otras opciones es correcta
c. VA = 0xee0123 de P1 apunta a la misma dirección física que VA = 0x0fe123 de P8
d. VA = 0x0f0123 de P6 apunta a la misma dirección física que VA = 0x0ff123 de P7
e. VA = 0x0ff123 de P1 apunta a la misma dirección física que VA = 0x0fe123 de P5

La respuesta correcta es: VA = 0xee0123 de P1 apunta a la misma dirección física que VA = 0x0fe123 de P8

Figura 20: Ejercicio Memoria Virtual

MIPS32 paginas de 4kb (12 bits de offset) y TLB FA (20 bits de TAG por descarte) de 8 entradas.

$$ASID_i <=> P_i$$

- $VA = 0 \times 0F0123 => TAG = 0 \times 0F0$ de $P_2 <=> ASID_2 => PPN = 0 \times 0F0$
- $VA = 0 \times 0F0123 => TAG = 0 \times 0F0$ de $P_6 <=> ASID_6 => PPN = 0 \times EE0$

- $VA = 0 \times EE0123 => TAG = 0 \times EE0$ de $P_1 <=> ASID_1 => PPN = 0 \times 0FF$
- $VA = 0 \times 0FE123 => TAG = 0 \times 0FE$ de $P_8 <=> ASID_8 => PPN = 0 \times 0FF$

- $VA = 0 \times 0F0123 => TAG = 0 \times 0F0$ de $P_6 <=> ASID_6 => PPN = 0 \times EE0$
- $VA = 0 \times 0FF123 => TAG = 0 \times 0F0$ de $P_7 <=> ASID_7 => PPN = 0 \times 0F0$

- $VA = 0 \times 0FF123 => TAG = 0 \times 0F0$ de $P_1 <=> ASID_1 => PPN = 0\times?$
- $VA = 0 \times 0FE123 => TAG = 0 \times 0F0$ de $P_5 <=> ASID_5 => PPN = 0\times?$

Se tiene un procesador con direcciones virtuales y físicas de 32 bits, páginas de 4 KBytes y TLB unificada cuyo contenido es:

V	Tag	PPN
1	0xf053a	0x22dfe
1	0x528a0	0xf053a
0	0x9dd01	0xdd8a0
0	0x0088d	0x0022b
1	0xd8db2	0xf053a
1	0x0088d	0x0a122
1	0x45a22	0xd8db2
0	0xd3da1	0x528a0

El programa realiza un acceso a la dirección virtual 0xf053a888: ¿cuál sería la dirección física correspondiente?

Seleccione una:

- a. Segmentation fault
- b. 0x528a0888
- c. TLB fault
- d. Ninguna de las otras respuestas es correcta
- e. 0x22dfe888
- f. 0x0022b888
- g. TLB miss
- h. No es posible determinar la dirección física con los datos suministrados
- i. Page fault

La respuesta correcta es: 0x22dfe888

Figura 21: Ejercicio Memoria Virtual

VA y PA de 32 bits. Paginas de 4kb, entonces preciso 12 bits de offset para direccionarlas.El offset no se traduce y es el mismo para PA y VA. Tag entonces de 20 bits.

Acceso a

$$0 \times F053A88 => TAG = \mathbf{11110000010100111010} \quad OFFSET = \mathbf{100010001000}$$

=>

$$TAG = 11110000010100111010 \quad y \quad OFFSET = 100010001000$$

=>

$$TAG = 0 \times F053A \quad y \quad OFFSET = 0 \times 888$$

=>

$$PPN = 0 \times 22DFE$$

=>

$$PPN \oplus OFFSET => 0 \times 22DFE888$$

Sea el siguiente extracto de la tabla de paginación lineal de un procesador que maneja páginas de tamaño 4KB:

Idx	V	D	P	PPN
0	1	1	1	0x10
1	1	1	0	0x00
2	0	0	0	0x00
3	1	1	1	0x20
4	0	0	0	0x00
5	1	0	1	0x30
6	1	0	1	0x40
7	1	0	0	0x00

La dirección virtual 0x35a0:

Seleccione una:

- a. Se traduce en 0x10F
- b. Ninguna de las otras opciones es correcta
- c. Page fault
- d. TLB fault
- e. Segmentation fault
- f. Se traduce en 0x205a0

La respuesta correcta es: Se traduce en 0x205a0

Figura 22: Ejercicio Memoria Virtual

Tamaño de página de 4kb, entonces 12 bits de offset para direccionar. Tabla de paginacion lineal, entonces indexo con la VPN. VA

$$0 \times 35A0 = 11010110100000 => VPN = \mathbf{11} \quad OFFSET = \mathbf{010110100000}$$

=>

$$VPN = 0 \times 3 \quad y \quad OFFSET = 0 \times 5A0$$

=>

$$IDX = 0 \times 3$$

=>

$$PPN = 0 \times 20$$

=>

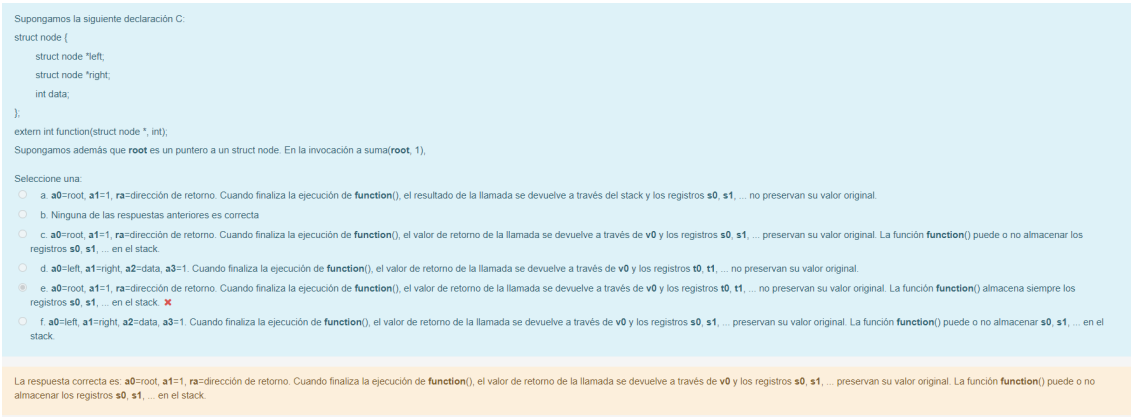
$$PPN \oplus OFFSET => 0 \times 205A0$$

.

- No poseemos la TLB, por lo tanto sería imposible determinar si es TLB fault.

- $V = 1$ y $P = 1$ por lo que no puede ser un Page Fault. Para que se de este caso deberia estar el bit de Presente en cero, $P = 0$. Un Page Fault quiere decir que la traducción es válida, pero que la pagina no está presente. Tengo que ir a levantarla a disco en ese caso.
- Para que se de el caso del segmentation fault, deberia estar el bit de Valido en 0, $V = 0$. Segmentation Fault implica un acceso ilegal a memoria, eso en la PT se indica con un bit de validez.

4.3. MIPS y performance



Supongamos la siguiente declaración C:

```
struct node {
    struct node *left;
    struct node *right;
    int data;
};
extern int function(struct node *, int);
```

Supongamos además que **root** es un puntero a un struct node. En la invocación a suma(**root**, 1),

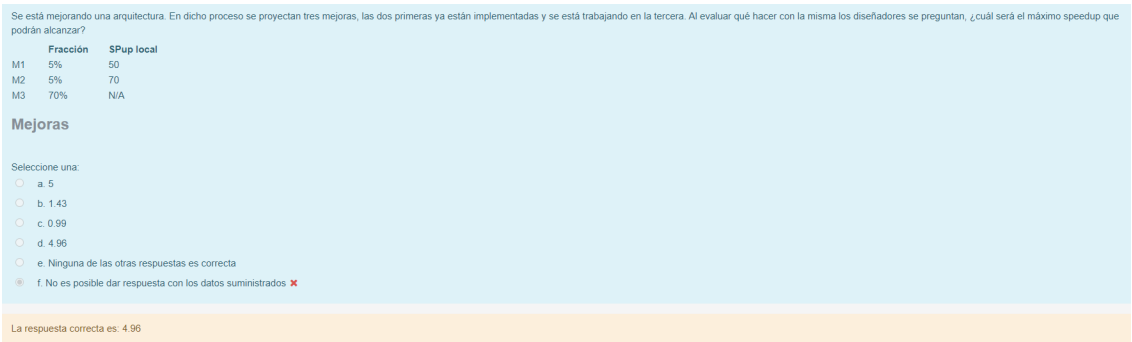
Seleccione una:

- a. **a0**=root, **a1**=1, **ra**=dirección de retorno. Cuando finaliza la ejecución de **function**(), el resultado de la llamada se devuelve a través del stack y los registros **s0**, **s1**, ... no preservan su valor original.
- b. Ninguna de las respuestas anteriores es correcta
- c. **a0**=root, **a1**=1, **ra**=dirección de retorno. Cuando finaliza la ejecución de **function**(), el valor de retorno de la llamada se devuelve a través de **v0** y los registros **s0**, **s1**, ... preservan su valor original. La función **function**() puede o no almacenar los registros **s0**, **s1**, ... en el stack.
- d. **a0**=left, **a1**=right, **a2**=data, **a3**=1. Cuando finaliza la ejecución de **function**(), el valor de retorno de la llamada se devuelve a través de **v0** y los registros **t0**, **t1**, ... no preservan su valor original.
- e. **a0**=root, **a1**=1, **ra**=dirección de retorno. Cuando finaliza la ejecución de **function**(), el valor de retorno de la llamada se devuelve a través de **v0** y los registros **t0**, **t1**, ... no preservan su valor original. La función **function**() almacena siempre los registros **s0**, **s1**, ... en el stack. ✖
- f. **a0**=left, **a1**=right, **a2**=data, **a3**=1. Cuando finaliza la ejecución de **function**(), el valor de retorno de la llamada se devuelve a través de **v0** y los registros **s0**, **s1**, ... preservan su valor original. La función **function**() puede o no almacenar **s0**, **s1**, ... en el stack.

La respuesta correcta es: **a0**=root, **a1**=1, **ra**=dirección de retorno. Cuando finaliza la ejecución de **function**(), el valor de retorno de la llamada se devuelve a través de **v0** y los registros **s0**, **s1**, ... preservan su valor original. La función **function**() puede o no almacenar los registros **s0**, **s1**, ... en el stack.

Figura 23: Ejercicio MIPS y Performance

La invocacion en $suma(root, 1)$, por lo que $a_0 = root$ y $a_1 = 1$. Al finalizar la ejecución de una función, siguiendo la ABI, los resultados se devuelven por v_0 y v_1. Los registros t_0 y t_1 son temporales, por lo que no preservan su valor original. Si los va a modificar, los registros s_0 y s_1 deben ser salvados en el stack, de lo contrario no es necesario. Sin embargo se garantiza que preservan su valor (ya sea salvandolos o no modificandolos).



Se está mejorando una arquitectura. En dicho proceso se proyectan tres mejoras, las dos primeras ya están implementadas y se está trabajando en la tercera. Al evaluar qué hacer con la misma los diseñadores se preguntan, ¿cuál será el máximo speedup que podrán alcanzar?

	Fracción	SPup local
M1	5%	50
M2	5%	70
M3	70%	N/A

Mejoras

Seleccione una:

- a. 5
- b. 1.43
- c. 0.99
- d. 4.96
- e. Ninguna de las otras respuestas es correcta
- f. No es posible dar respuesta con los datos suministrados ✖

La respuesta correcta es: 4.96

Figura 24: Ejercicio MIPS y Performance

$$SPup_G = \frac{1}{1 - \sum_i f_L i + \sum_i \frac{f_{Li}}{SPup_{Li}}}$$

$$SPup_G = \frac{1}{1 - 0{,}05 - 0{,}05 - 0{,}7 + \frac{0{,}05}{50} + \frac{0{,}05}{70} + \frac{0{,}7}{X}}$$

El máximo Speedup se alcanzará cuando el Speedup local de la mejora 3 sea máximo. Haciendo tender este valor a ∞, el cociente $\frac{0,7}{X}$ tenderá a cero, quedando la ecuación

$$SPup_G = \frac{1}{1 - 0{,}05 - 0{,}05 - 0{,}7 + \frac{0{,}05}{50} + \frac{0{,}05}{70}} = 4{,}957$$


Supongamos un programa con la siguiente declaración para manejar árboles binarios representado con nodos:

```
struct node {
    struct node *left;
    struct node *right;
    int data;
    char balance;
};
```

Asumiendo que el registro **t0** contiene la dirección de memoria de el primer byte de un nodo, cuál de las siguientes instrucciones puede usarse para leer el atributo **balance** del árbol?

Seleccione una:

- a. move t1, 16(t0)
- b. lw t1, 12(t0) ✖
- c. Ninguna de las otras opciones es correcta
- d. lhu t1, 16(t0)
- e. lw t1, 0(t0)
- f. lw t1, 4(t0)
- g. lbu t1, 12(t0)

La respuesta correcta es: lbu t1, 12(t0)

Figura 25: Ejercicio MIPS y Performance

En lo primero 4 bytes tengo left, la siguiente palabra es right, la 3ra es data y en el cuarto campo se encuentra balance. Balance es un solo char, por lo que es un byte, el resto es padding.

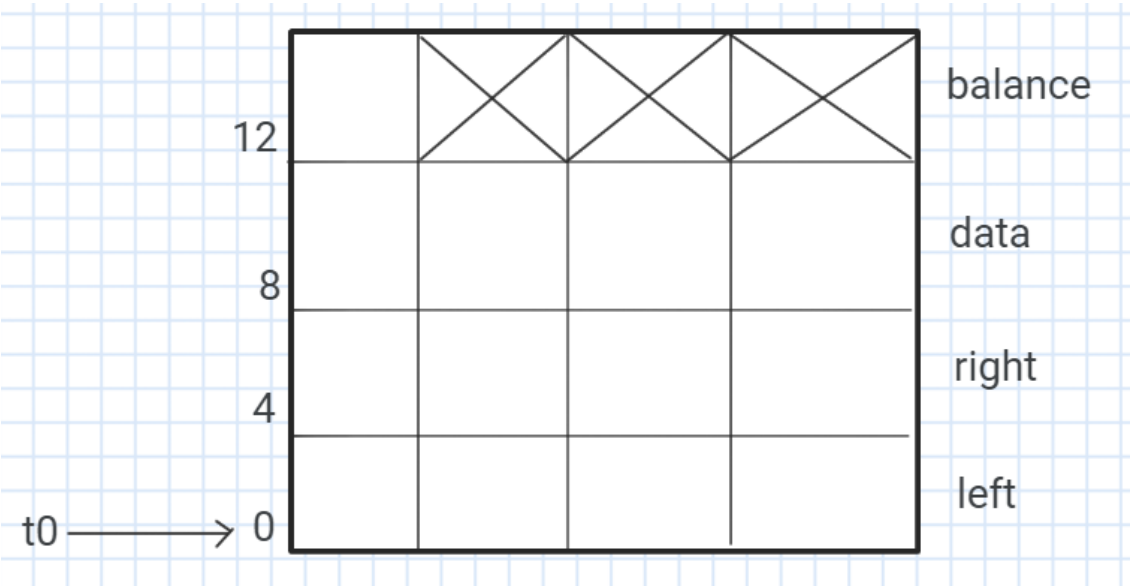


Figura 26: Estructura de memoria

- *move* $t_1, 16(t_0)$, el move se da entre registros, no implica acceso a memoria, por lo que es falso
- *lbu* $t_1, 12(t_0)$, accedo el byte en $t_0 + 12$, y como es load byte unsigned, accedo a un solo byte que es lo que queria. Es verdadera

- $lw \quad t_1, 4(t_0)$ en este caso el offset está mal ya que apunta a right. Además es un load word, carga 4 bytes y nosotros queremos acceder a unos solo.
- $lw \quad t_1, 0(t_0)$ caso igual al anterior pero apunta a left.
- $lw \quad t_1, 12(t_0)$ El offset es correcto pero carga toda la palabra, incluyendo el padding. En este caso leo el dato y basura
- $lhu \quad t_1, 16(t_0)$ El tamaño está mal mal porque load half unsigned carga 2 bytes y además el offset me coloca por encima de la estructura a la que quiero acceder.

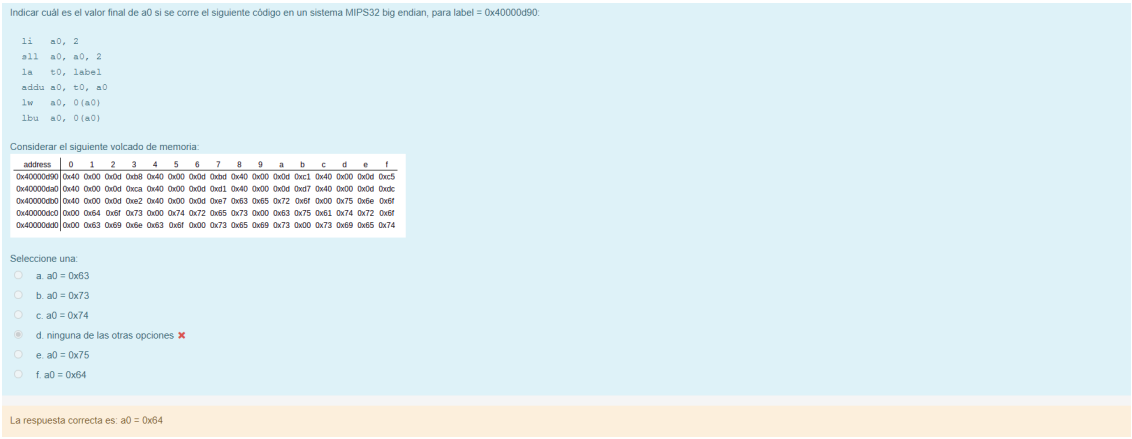


Figura 27: Ejercicio MIPS y Performance

Al ser Big Endian, el byte mas significativo es el primero que leo en memoria. De ser Little Endian, seria al revés. En Little Endian lo primero que aparece es el LSB.

$$Label = 0 \times 40000d90$$

Cargo un inmediato en a0

$$li \quad a_0, 2$$

$$a_0 = 2 \qquad t_0 = ?$$

Shift a la izquierda en 2 posiciones, que es lo mismo que multiplicarlo por 4

$$sll \quad a_0, a_0, 2$$

$$a_0 = 8 \qquad t_0 = ?$$

cargo en t0 una direccion etiquetada como label

$$la \quad t_0, label$$

$$a_0 = 8 \qquad t_0 = 0 \times 40000d90$$

Sumamos a0 a esa dirección, usando a a0 como índice. $a_0 = 4 \times 2 = 8$ entonces

$$addu \quad a0, t0, a0$$

$$a_0 = 0 \times 40000d90 + 8 = 0 \times 40000d98 \qquad t_0 = 0 \times 40000d90$$

Accedo al word en esa posicion

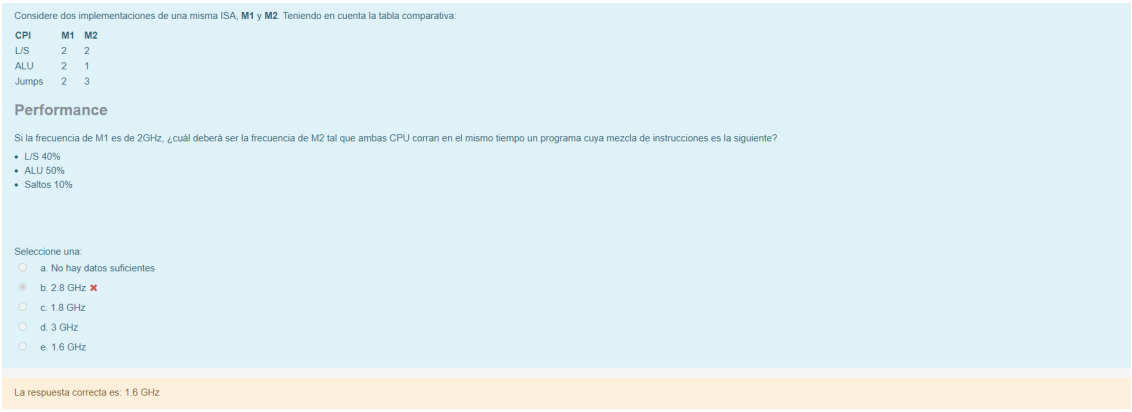
$$lw \quad a_0, 0(a_0)$$

$$a_0 = 0 \times 40000dc1 \qquad t_0 = 0 \times 40000d90$$

El contenido de ese word lo usamos como una direccion para acceder a un byte

$$lbu \quad a_0, 0(a_0)$$

$$a_0 = 0 \times 64 \qquad t_0 = 0 \times 40000d90$$


Considere dos implementaciones de una misma ISA, **M1** y **M2**. Teniendo en cuenta la tabla comparativa:

CPI	M1	M2
L/S	2	2
ALU	2	1
Jumps	2	3

Performance

Si la frecuencia de M1 es de 2GHz, ¿cuál deberá ser la frecuencia de M2 tal que ambas CPU corran en el mismo tiempo un programa cuya mezcla de instrucciones es la siguiente?

- L/S 40%
- ALU 50%
- Saltos 10%

Seleccione una.

- a. No hay datos suficientes
- b. 2.8 GHz ✖
- c. 1.8 GHz
- d. 3 GHz
- e. 1.6 GHz

La respuesta correcta es: 1.6 GHz

Figura 28: Ejercicio MIPS y Performance

$$CPI_{\text{conjuntos}} = \sum_i f_i \times CPI_i$$

$$CPI_{\text{M1}} = 2 \times 0{,}4 + 2 \times {,}5 + 2 \times 0{,}1 = 2$$
$$CPI_{\text{M2}} = 2 \times 0{,}4 + 1 \times {,}5 + 3 \times 0{,}1 = 1{,}6$$

Considero IC igual para ambos casos

$$\text{tiempo de CPU} = IC \times CPI \times T_{CLK}$$

$$t_{M1} = IC \times CPI \times T_{CLK} = IC \times 2 \times \frac{1}{2GHz}$$
$$t_{M2} = IC \times CPI \times T_{CLK} = IC \times 1{,}6 \times \frac{1}{Frec_{CLKM2}}$$

$$\frac{t_{M1}}{t_{M2}} = \frac{IC \times 2 \times \frac{1}{2GHz}}{IC \times 1{,}6 \times \frac{1}{Frec_{CLKM2}}} = 1$$

$$1{,}25 \times \frac{Frec_{CLKM2}}{2GHz} = 1$$

$$Frec_{CLKM2} = 1{,}6GHz$$