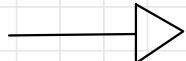




Como pasar variables de C a MIPS:

Siempre hacer la declaración de las variables en el mismo orden que en el código de alto nivel.

```
unsigned char a;  
short x = 13;  
char b = -1, c = 10;  
int y = 0x13;  
long long d = 0x12;
```



```
.data  
a: .byte 0  
x: .half 13  
b: .byte -1  
c: .byte 10  
y: .word 0x13  
d: .dword 0x12
```

Cuando almacenamos en memoria hay que tener en cuenta que van al revés (Little endian) i hay que tener en cuenta que tienen que estar en la posición de memoria que les corresponde en función del tipo de dato que és.

byte multiple de 1

half multiple de 2

word multiple de 4

double word multiple de 8

Al usar las directivas half, word i dword la información se alinea automáticamente
La directiva .space se usa para cuando queremos reservar espacio en la memoria
n posiciones, però esta no sabe alinear. Entonces usamos .align
.align ubica el próximo dato a 2^n

2.6

100100000	10	9	00	12	43	1B		24	
1	XX	A	00	13	00	1C		25	
A:	2	F B	B	00	14	XX	1D	26	
3	FF	C:	C	FC	15	XX	1E	27	
B:	4	FF	D	3S	16	XX	1F	28	
5	FF	E	66	17	XX	20		29	
6	FF	F	XX	E:	18		21	2A	
7	FF	D: 10	42		19		22	2B	
8	43	11	00		1A		23	2C	

Acabar

36	0010 0100	0x24
-97	1001 1111	0x9F
109	0110 1101	0x6D
132	NR	NR
-128	1000 0000	0x80
43	0010 1011	0x2B
-34	1101 1110	0xD E
-131	NR	UN
-1	1111 1111	0xFF

.data

a: .byte 0

x: half 0xFF

b: .byte -109 0x93

c: .byte 47 0x2F

y: .word 356 0x00000164

s: .half 119 0x0077

d: .dword -519

a: 0	00	9	01	12	FF	1B		24	
1	xx	A	00	13	FF	1C		25	
x: 2	FF	B	00	14	FF	1D		26	
3	FF	C	??	15	FF	1E		27	
b: 4	93	D	00	16	FF	1F		28	
c: 5	2F	E	xx	17	FF	20		29	
6	xx	F	xx	18		21		2A	
7	xx	d: 10	FA	19		22		2B	
y: 8	64	11	FD	1A		23		2C	

Instrucciones MIPS

La mayoría de las instrucciones tienen 3 operandos

INST op1, op2, op3 # op1 ← op2 (INST) op3

addu (add unsigned) - addu \$t1, \$t2, \$t3 #\$t1 = \$t2 + \$t3

addiu (add immediate unsigned) - addiu \$t1, \$t2, SE(N16) #\$t1 = \$t2 + 29

subu (subtract unsigned) - subu \$t1, \$t2, \$t3 #\$t1 = \$t2 - \$t3

move \$t1, \$t2 # equivale a addu \$t1, \$t2, \$zero

li (load immediate) - li \$t1, N32

Si N > 16 bits

Si N ≤ 16 bits

lui \$at, hi(N16)

addiu \$t1, \$zero, N16

ori \$t1, \$at, lo(N16)

la (load address) - la \$t1, etiqueta

Esta instrucción guarda la dirección donde se encuentra el valor

lui \$at, hi(@etiqueta)

ori \$t1, \$at, lo(@etiqueta)

Para definir una constante en MIPS se usa .eqv

Para saber la dirección de una posición de un vector declarado, tenemos que multiplicar la posición a la que queremos acceder por los bites que ocupe el tipo de variable con la que estamos trabajando

La instrucción lb (load byte), extiende signo. Pero la instrucción lbu (load byte unsigned) extiende siempre 0

Caracteres:

Para hacer operaciones con caracteres en MIPS podemos declarar directamente las variables como en C ('A').

Strings

Cuando trabajamos con strings hay que tener en cuenta que siempre al final va el centinela que define que se ha acabado el string ('\0' o null).

En MIPS eso se usa con

.ascii (este no incluye centinela)
.asciiz (este si)

Punteros

Un puntero es una variable que contiene la dirección de otra variable

En MIPS todos los punteros ocupan 32 bits y se definen como .word

Para inicializar un puntero usamos el operador &:

int v[100];

char a = 'E';

int *q = &a; (este puntero apunta a la dirección de la variable a)

int *p = &v[12]; (este puntero apunta a la dirección de la posición 12 del vector)

En MIPS se usa de la siguiente manera:

.data

v: .space 400

.align 2

a: .byte 'E'

q: .word a # q = &a

p: .word v+48 #p = &v[12]

Para leer la dirección de memoria apuntada por el puntero usamos *

Un ejemplo de esto es lo siguiente:

int *pgl;

int temp = *pgl

*pgl = 17;

Aritmética de punteros

Al sumar posiciones a punteros que apuntan a diferentes tipos de variables, tenemos que tener en cuenta las posiciones que ocupa el tipo de variable que hemos utilizado

MIPS:

```
char *p1;  
short *p2;  
int *p3;  
long long *p4;
```

```
p1: .word 0  
p2: .word 0  
p3: .word 0  
p4: .word 0
```



```
p1 += 3;           addiu $t1, $t1, 3    #tamaño de char = 1 byte  
p2 = p2 + 3;     addiu $t2, t2, 3*2  # tamaño de short = 2 byte  
p3 += 3;          addiu $t3, $t3, 3*4 #tamaño de int = 4 byte  
p4 = p4 + 3;     addiu $t4, $t4, 3*8 #tamaño de long long = 8 byte
```

sll (shift left logical) rd, rt, shamt # rd = rt << shamt

shamt = shift amount,

srl (shift right logical) rd, rt, shamt # rd = rt >> shamt

natural de 5 bits

La multiplicación siempre sirve, sea sin o con signo.

Mientras que la división solo sirve con naturales (sin signo)

sra (shift right arithmetic) rd, rt, shamt # rd = rt >> shamt

Esta division extiende signo al hacer la operación

Si queremos hacer las mismas instrucciones con un registro tenemos que añadir simplemente una v al final de la instrucción (usa los 5 bits de menor peso)

sllv rd, rt, rs # rd = rt << rs

srlv rd, rt, rs # rd = rt >> rs

Puertas lógicas MIPS (bit a bit)

and \$t0, \$t0, \$t1

si queremos usar las instrucciones

xor \$t0, \$t0, \$t1

con inmediatos es solo añadir una i

or \$t0, \$t0, \$t1

nor \$t0, \$t0, \$zero

al final (solo 16 bits)

Usos de las puertas bit a bit

Seleccionar bits dejando el resto a 0 andi \$t0,\$t0, 0xF

XXXX XXXX XXXX 0101 → resultado 0000 0000 0000 0101
0000 0000 0000 1111

También sirve para calcular residuos al dividir por potencias de 2

Activar determinados bits a 1 sin modificar el resto ori \$t0, \$t0, 0x80

Complementar bits xori \$t0, \$t0, 0xFF

Comparaciones MIPS

En MIPS solo existe una operación para poder comparar 2 registros o un registro y un inmediato

slt rd, rs, rt rd = rs < rt existe para unsigned y también para inmediatos

Para hacer una negación booleana tenemos que usar la siguiente instrucción

sltiu \$t2, \$t0, 1 de esta manera si el contenido del registro es:
0, tenemos $0 < 1$ que es cierto (1)
 < 0 , tenemos que $1 < 1$ es falso (0)

De esta manera podemos implementar todas las comparaciones posibles

Normalizar un booleano es considerado pasar un valor a 0 o 1, dependiendo si ya es 0, o si es mayor que 0

Para hacer esto en MIPS usamos la siguiente instrucción:

sltu \$t0, \$zero, \$t0 #si \$t0 es falso (0) ($0 < 0$) \$t0 = 0
 #si \$t0 es cierto ($!= 0$) ($0 <= 0$) \$t0 = 1

AND (&&) booleana

sltu

sltu

and

OR (||) booleana

Saltos condicionales MIPS (branch)

beq \$t1, \$t2, etiq #salta a etiq si \$t1 == \$t2

bne \$t1, \$t2, etiq #salta a etiq si \$t1 != \$t2

b etiq

blt rs, rt, etiq #salta a etiq si rs < rt

bgt rs, rt, etiq #salta a etiq si rs > rt

bge rs, rt, etiq #salta a etiq si rs >= rt

ble rs, rt, etiq #salta a etiq si rs <= rt

$$d = a + b + c$$

l suma

$$c = a + b$$

sin sumar

Saltos incondicionales lejanos (jump)

j target #PC = target

jr rs #PC = rs

slti \$t1, \$t0, 1
li \$t3, 0xAAAAAAAAAA
and \$t1, \$t1, \$t3
srli \$t2, \$t0, 1
li \$t4, 0xSSSSSSSS
and \$t2, \$t2, \$t4
or \$t1, \$t2, \$t4

Traducción sentencia IF-THEN-ELSE

evaluar condición

saltar si es falso a else

if:

CUERPO-IF

saltar a endif

else:

CUERPO-ELSE

endif:

Tema 2: 10, 14, 20, 23, 30, 32

Tema 3: 3, 4, 5, 16, 22, 27, 28

33, 36, 13

Traducción sentencia WHILE

while:

evaluar condición
saltar a end si es FALSO
CUERPO WHILE
saltar a while

end:

Traducción sentencia FOR

INI

for:
evaluar COND
saltar si es FALSO a end
CUERPO FOR
INC
saltar a for

end:

Traducción sentencia DO WHILE

do:

CUERPO DO
evaluar condición
saltar si CIERTO a do

end:

Subrutinas (Importante)

Es un conjunto de instrucciones que realizan una tarea específica y que puede ser llamada desde cualquier punto del programa (también existen subrutinas recursivas que se llaman a si mismas)

La instrucción que se usa para llamar a una subrutina es jal (jump and link) esta guarda en el registro \$ra la dirección de la próxima instrucción.

Al final de la subrutina usamos el jr \$ra, que es un jump register que salta de vuelta a la dirección guardada en \$ra

El paso de parámetros en MIPS se pasan por \$a0-\$a3, en orden de derecha a izquierda (máximo 4 parámetros)

El resultado de cualquier subrutina se devuelve siempre en \$v0 (\$f0 si es float)

Las variables locales de una función se guardan en los siguientes registros:

\$t0-\$t9, \$s0-\$s7, \$v0-\$v1

Si tenemos que inicializar un vector como variable local (vectores, matrices, tuplas...), tenemos que usar una pila (stack) y tenemos un registro que apunta al primer elemento de la pila que es el \$sp (stack pointer)

Cuando hacemos subrutinas multinivel tenemos que guardar el valor de \$ra, para poder volver a la rutina anterior o main. Esa pila en MIPS se guarda en la dirección 0xFFFF FFFC, la pila crece hacia direcciones bajas quiere decir que cada vez la dirección es más pequeña.

Para poder ir entre las diferentes subrutinas se usa una estructura de datos llamada Bloque de Activación, está usa la pila.

Esta sigue unas reglas:

- Tienen que estar en el mismo orden en el que están escritas en el código
- Tiene que estar alineadas a su tamaño y si hace falta incluimos espacios en blanco (tiene que apuntar a una dirección múltiple de 4)

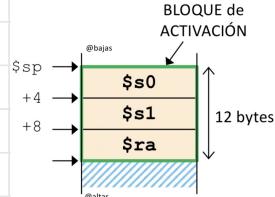
En MIPS, en primer lugar deberíamos reservar el espacio para poder ejecutar la subrutina con addiu \$sp, \$sp, -(Espacio necesario) y al final justo lo contrario sumar el espacio necesario a \$sp

ABI MIPS

Registros Temporales: \$t0-\$t9, \$v0-\$v1, \$a0-\$a3. Estos registros se pueden modificar en cualquier momento

Registros Seguros: \$s0-\$s7, \$ra, \$sp. Si queremos modificar alguno de estos registros los tenemos que guardar en la pila

Este es el dibujo que siempre se va a usar para representar el Bloque de Activación



En el bloque de activación, arriba siempre van las variables locales y luego los registros que queremos guardar (\$s0-\$s7)

Si hay alguna llamada a otra subrutina dentro de una rutina és seguro que el registro \$ra tiene que ser guardado en memoria

Matrices

Para poder calcular la dirección de algún elemento de la matriz, en primer lugar hace falta multiplicar dos valores. Però el problema esta en que la multiplicación nos devuelve un valor de 64 bits.

Para ello tenemos instrucciones especiales junto con 2 registros \$hi y \$lo.

La instrucción és **mult rs, rt # \$hi, \$lo = rs*rt**

Estos dos registros són especiales, no se pueden usar en ninguna operación mas, tenemos instrucciones para pasar el valor a algún registro:

mflo rd # rd = \$lo mfhi rd # rd = \$hi

En C las matrices se almacenan por filas, primero la fila 0 como un vector, luego la fila 1 y asi consecutivamente. Siempre és Matriz[Fila][Columna].

Si queremos acceder al elemento [i][j] de una matriz, tenemos que coger en primer lugar la dirección de la

Matriz[0][0] + tamaño de la variable(i*numero de columnas+j)

#@Mat[i][j]

la \$t3, Mat	#t3 ← @Mat[0][0]
li \$t4, nCol	
mult \$t0, \$t4	#i*nCol
mflo \$t4	#t4 ← i*nCol
addu \$t4, \$t4, \$t1	#t4 ← i*nCol + j
sll \$t4, \$t4, 2	#t4 ← (i*nCol+j)*4
addu \$t4, \$t4, \$t3	#t4 ← @Mat[i][j]
lw \$t2, 0(\$t4)	#t2 ← Mat[i][j]

#@Mat[i][5]

```
la $t3, Mat+20      #t3 <- @inicioMat+20
li $t4, nCol*4
mult $t0, $t4        #i*nCol*4
mflo $t4             #t4 <- i*nCol*4
addu $t4, $t4, $t3   #t4 <- @Mat[ i ][ 5 ]
lw $t2, 0($t4)       #t2 <- Mat[ i ][ 5 ]
```

#@Mat[3][j]

```
la $t3, Mat          #t3 <- @Mat[0][0]
sll $t4, $t1, 2       #t4 <- j*4
addu $t4, $t4, $t3   #t4 <- @Mat[0][ j ]
lw $t2, nCol*12($t4) #t2 <- Mat[3][ j ]
```

Acceso Secuencial

Cuando tenemos un acceso a todo un vector (como inicializarlo a 0), si podemos saber fácilmente el tamaño del vector podemos facilitar el acceso al vector.
Movemos la variable a un registro y le sumamos el tamaño de la variable.

```
move $t1, $a0
...
addiu $t1, $t1, 4
```

Para poder hacer comparaciones entre direcciones es importante usar las comparaciones unsigned.

El **stride** es lo que tenemos que aumentar para pasar a la siguiente posición deseada, si nos movemos por filas de una matriz de int sera 4. Si nos movemos entre columnas sera 6*4 (numero de filas*tamaño del elemento) 24 en este caso.

Estructura de la memoria MIPS

Almacenamiento estático:

La sección .data tiene un tamaño fijo para cada programa y se usa principalmente para guardar las variables globales de C.

Para acceder a una variable que esta en .data usamos 3 instrucciones:

la \$t0, etiquetaVarGlobal

lw \$t1, 0(\$t0)

Almacenamiento dinámico:

Esto es principalmente la pila, guarda en el bloque de activación variables locales y registros seguros.

.heap (memoria que pedimos al OS)

Tenemos ciertas instrucciones en C que nos dan espacio en memoria (prt = malloc()) esta instrucción nos va a dar un puntero que apunta a esa dirección. Si hacemos un malloc() pidiendo demasiada memoria nos dará un puntero con el valor NULL, así que hay que comprobar que no valga NULL.

Lo contrario sería free(prt), esta libera la memoria en función del puntero y siempre después de un free(prt) hay que poner el puntero a NULL

Pasos a seguir para ejecutar un programa:

Compilación:

Traduce un código fuente escrito en un Lenguaje de alto nivel a Ensamblador

Ensamblado:

Traduce de un código escrito en ensamblador a lenguaje maquina (fichero objeto)

Enlazado:

Normalmente en una aplicación tenemos múltiples ficheros objeto. El enlazador (linker) se encarga de generar un fichero único.

Rendimiento

Para saber cuánto mejor es un hardware que otro hay principalmente 2 criterios:

El tiempo de ejecución:

$$T(exe) = T(CPU) + T(E/S) + T(resto)$$

T(CPU) El tiempo que consumimos ejecutando el programa

T(E/S) Tiempo que pasamos esperando al disco o la red

Speedup (Aceleración, Ganancia de rendimiento)

Tiempo original/Tiempo mejorado = Rendimiento mejorado/Rendimiento original

Maneras de mejorar:

Optimizar el programa

En el compilador

En la micro arquitectura

En el hardware

Productividad (tareas completadas por unidad de tiempo)

Se puede expresar el tiempo de ejecución de la siguiente forma:

$$T(EXE) = n^{\circ} \text{ ciclos} * T(\text{ciclo}) = n^{\circ} \text{ ciclos} / \text{frecuencia clock}$$

Para reducir el tiempo de ejecución podemos reducir el número de ciclos o aumentar la frecuencia

$$T(EXE) = N * CPI * T(C)$$

Reducción del número de ciclos

Tiempo de ejecución de un programa

n° ciclos = $N * CPI$ (Número total de instrucciones * Promedio de ciclos por instrucción)

Para reducir los ciclos podemos:

Reducir las instrucciones → Mejorando el compilador

Reducir el CPI → Mejorando la microarquitectura o usando instrucciones mas rápidas

$$T_{\text{EXE}} = \frac{\text{instrucciones}}{\text{programa}} \cdot \frac{\text{ciclos}}{\text{instrucción}} \cdot \frac{\text{segundos}}{\text{ciclo}} = \frac{\text{segundos}}{\text{programa}}$$

Como medir el consumo de una CPU

$$P = P(d) + P(s)$$

Potencia dinámica (causada por la comutación de los transistores)

Potencia estática (corrientes de fuga)

Potencia dinámica:

$$P_d = \alpha \cdot C \cdot V^2 \cdot f_{\text{clock}}$$

($\alpha * C$) normalmente aparecen como un factor C

siendo,

- P_d : Potencia Dinámica (Watios, W)
- α : Fracción de transistores que comutan por ciclo
- C: Capacitancia (Faradios, F)
- V: Voltaje (Voltios, V)
- f_{clock} : Frecuencia de reloj (ciclos/s)

Problemas de rendimiento

$$\text{CPI} = \text{nºciclos/instrucciones}$$

Tenim un codi en MIPS que treballa sobre matrius. Hem fet dues versions del codi, una usant accés aleatori (AA) i una altra usant accés seqüencial (AS). El total d'instruccions utilitzades a cada versió les podeu trobar a la següent taula:

	Load / Store	Mult	Salts que salten	Salts que no salten	Altres instruccions
AA	500.000	250.000	250.000	250.000	3.750.000
AS	500.000	0	500.000	500.000	6.000.000

Suposem que a la nostra arquitectura, cada instrucció de memòria costa 5 cicles, cada multiplicació 16 cicles, els salts 1 o 2 cicles depenen de si salta (2) o no salta (1) i la resta d'instruccions 1 cicle.

- Quin seria el CPI de cada versió del programa? Quin seria el speed-up de la versió d'accés seqüencial respecte a la d'accés aleatori?
- Quant de temps, en segons, triga en executar-se cada programa si el rellotge va a una freqüència de 2GHz?
- Si el processador dissipa 3 watts (Joules per segon) sigui quina sigui la instrucció executada, quin ha estat el consum total del processador en Joules de cada versió del programa?
- Ens diuen que es pot dissenyar una arquitectura alternativa on podríem reduir el temps de cicle de les operacions de multiplicació. Quin és el màxim nombre de cicles de multiplicació admissible per a la nova arquitectura per tal que el codi AA sigui al menys igual de ràpid que el codi AS?

a)

AA:

$$5 \cdot 10^6 \text{ instrucciones} (0,5 \cdot 5 + 0,25 \cdot 16 + 0,25 \cdot 2 + 0,25 + 3,75) \cdot 10^6 = \\ (2,5 + 4 + 0,5 + 0,25 + 3,75) = 11 \cdot 10^6 \text{ ciclos}$$

$$\text{CPI} = (11/5) = 2,2 \text{ ciclos/inst}$$

AS:

$$\#\text{instrucciones} = 7,5 \cdot 10^6$$

$$\#\text{ciclos} = 10 \cdot 10^6$$

$$\text{CPI} = 10/7,5 = 1,33 \text{ ciclos/inst}$$

b)

$$\text{Speedup} = 11/10 = 1,1$$

$$T = \#\text{ciclos}/f \rightarrow AA = 11/2 \cdot 10^3 = 5,5 \text{ ms}$$

$$\rightarrow AS = 10/2 \cdot 10^3 = 5 \text{ ms}$$

c)

$$W/t = 3/5 \cdot 10^{-3}$$

d)

$$0,25 \cdot 10^6 \cdot 16 = 4 \cdot 10^6 \text{ ciclos}$$

$$0,25 \cdot 10^6 \times = 3 \cdot 10^6 \text{ ciclos} \rightarrow x = 12$$

Problema 2:

a)

$$60 = C \cdot 1 \cdot 6/3 \cdot 10^9$$

$$\frac{60 \cdot 3}{6 \cdot 10^9} = 30 \cdot 10^{-9} F$$

$$P_2 = 30 \cdot 10^{-9} \cdot 1,1^2 \cdot 7/3 \cdot 10^9 \\ = 84,7 W$$

$$P_3 = " 1,2^2 \cdot 8/3 \cdot 10^9 = 115,2 W$$

$$P_4 = " 1,3^2 \cdot 2 \cdot 10^{-9} = 152,1 W$$

Un processador ha estat dissenyat per poder funcionar correctament sols a les següents combinacions de freqüències i voltatges d'alimentació (les freqüències estan expressades en forma de fracció per facilitar la simplificació dels càlculs, sense la calculadora):

combinació	voltatge (V)	freqüència (GHz)
1	1,0	6/3
2	1,1	7/3
3	1,2	8/3
4	1,3	9/3

Suposem que la potència estàtica consumida és zero. Sabent que la potència dinàmica consumida amb la combinació número 1 és de $P_1 = 60W$, es demana:

a) Quines són les potències dinàmiques consumides en les combinacions 2, 3 i 4, en watts? Quina és la combinació amb màxima freqüència i voltatge a la qual podrà funcionar la CPU sense sobrepassar el màxim consum permès pel dissipador, que són 120W?

b) (0,5 pts) Quin és el guany de rendiment (o speedup) que s'obté amb la combinació número 4 respecte de la combinació número 1, executant el mateix programa?

c) (0,5 pts) Amb la combinació número 1, quin és el temps d'execució (en segons) d'un programa que executa 10^{10} instruccions i que té un CPI promig de 4?

b)

$$\frac{4}{1} \text{ speed up? } \frac{\frac{1}{F_1}}{\frac{1}{F_4}} = \frac{F_4}{F_1} \quad \frac{9/3}{6/3} = \frac{9}{6} = 1,5$$

$$T_{exe} = N \cdot \text{CPI} / F$$

c)

$$= 10^{10} \cdot 4 \cdot 1/2 \cdot 10^9 = 20s$$

Tema 5: Aritmética de enteros y coma flotante

Instrucciones que generan excepción overflow

add, addi, sub

Ignora overflow

addu, addiu, subu

Código para ver si hay overflow

addiu \$t2, \$t1, \$t0

xor \$t3, \$t1, \$t0 # a^b

nor \$t3, \$t3, \$zero # $-(a^b)$

xor \$t4, \$t2, \$t0 # a^c

and \$t3, \$t3, \$t4 # $-(a^b) \& (a^c)$

srl \$t3, \$t3, 31 # Ponemos V en el bit 0

Para detectar el overflow en números naturales:

addiu \$t2, \$t0, \$t1 # $c = a+b$

sltu \$t3, \$t2, \$t0 # $\$t3 = \text{carry}$

Se puede detectar:

$c = a+b$ si $c < a \rightarrow \text{carry}$

$c = a-b$ si $a < b \rightarrow \text{carry}$

Algoritmo para multiplicar naturales

P = 0;

MD = X;

MR = Y;

for (int i = 0; i < n; ++i) {

 if (MR == 1)

 P += MD;

 MD = MD << 1;

 MR = MR >> 1;

Overflow al multiplicar MIPS

mult tiene overflow si $\$hi \neq 0$

mult tiene overflow si $\$hi$ no es la extension de signo de $\$lo$

- Si bit de 31 de $\$lo$ es 1, $\$hi$ tiene que ser -1
- Si el bit 31 de $\$lo$ es 0, $\$hi$ tiene que ser 0

Como pasar números decimales a binario (coma flotante)

Primero pasamos la parte entera a binario

Luego cogemos la parte decimal i la multiplicamos por dos

Ejemplo: $-1029,68 \rightarrow 1029 = 100000000101 \quad 0,68 * 2 = 1,36 \text{ (1)}$

$$\begin{array}{r} 0111 \\ 1000 \\ 1000 \\ 1000 \\ 1001 \\ 1010 \\ 1011 \\ 1100 \\ 1101 \\ 1110 \\ 1111 \end{array} \quad \begin{array}{l} 0,36 * 2 = 0,72 \text{ (0)} \\ 0,72 * 2 = 1,44 \text{ (1)} \\ \dots \end{array}$$

Tema 6: Memoria cache

La memoria cache es una memoria muy pequeña pero a la vez muy rápida también.

Cuando el procesador lee algo, primero lo lee de la cache si está allí es un acierto. Pero en cambio si no está es un fallo y tendrá que acceder a memoria para poder llevar a cabo la instrucción.

Un ejemplo es el acceso de datos es un vector, en la primera vez será un fallo porque no tenemos el vector en la cache.

Entonces esta coge un bloque de 16 (como es un vector de enteros ocupa 4) entonces nos traemos $v[0] \dots v[3]$.

El próximo fallo se producirá en el $v[4]$ y entonces volverá a leer de memoria 4 posiciones así en bucle.

Para las instrucciones es lo mismo ya que en MIPS las instrucciones ocupan 4 bytes, pero en este caso cuando entramos en un bucle y leemos una vez las instrucciones ya las tenemos en memoria y no hace falta leerlas otra vez.

Tasa de aciertos

Tasa de fallo

$$h = \# \text{aciertos} / \# \text{referencias}$$

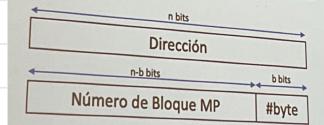
$$m = \# \text{fallos} / \# \text{referencias}$$

$$m = 1 - h$$

Organización de la Memoria Principal

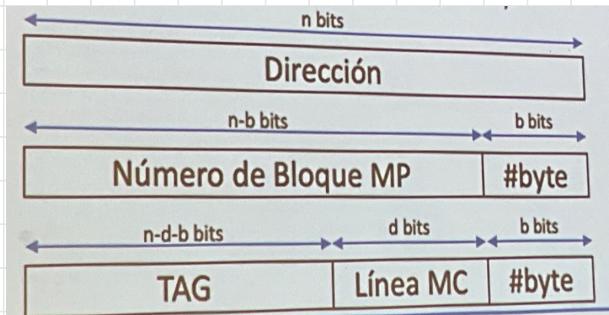
La MP se organiza en bloques de tamaño fijo ($B = 2^b$ bytes)

Si tenemos una dirección cualquiera los b bits bajos es el número de byte y lo restante es el número de bloque.



Cache Directa

En una memoria cache, podemos almacenar D bloques de MP y es potencia de 2



Para poder saber que línea de la memoria principal esta en la posición 00 tenemos que almacenar el TAG de la linea

Podemos ver una Memoria cache directa como una tabla

En cada entrada de la tabla tenemos 1 linea de cache

La información que almacenamos de cada línea es:

Datos: contiene 2^b bytes del bloque

TAG: información necesaria para identificar el bloque de MP

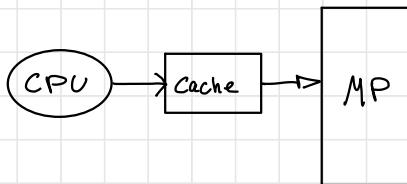
Bit de validez: indica si al línea esta ocupada

Tiempo de acceso a la cache

$$\text{Solo de lectura} \quad t_{\text{acceso}} = t_h \quad / \quad t_{\text{acceso}} = t_h + t_{\text{bloq}} + t_h$$

Write through / no alocate Lectura | Escritura

Presumen cache



$$m = 1 - h$$

$$h = \frac{\text{aciertos}}{\# \text{ referencias}}$$

3 tipos de cache

- Cache directa

$$T_{\text{cache}}: 2^b + 2^D = 2^T$$

$$T_{\text{lnea}}: 2^b$$

$$N^o \text{ lneas: } 2^D$$

- Cache completamente
associativa

$$T_{\text{cache}}: 2^b \cdot x$$

$$T_{\text{lnea}}: 2^b$$

$$N^o \text{ lneas} = \text{grado ass.} = x$$

- Cache n-way

$$T_{\text{cache}}: 2^S \cdot 2^b \cdot x$$

$$T_{\text{lnea}} = 2^b$$

$$N^o \text{ Conjuntos} = 2^S$$

grado de
asociatividad: x

Dada una dirección

$n-b$	b
# Bloque MP	# byte

MC Directa

$n-D-b$	D	b
TAG	# línea MC	#byte

MC Conjuntos

$n-S-b$	S	b
TAG	# set	#byte

MC Completamente asociativa

$n-b$	b
TAG	#byte

Algoritmos de reemplazo

- Aleatorio
- LRU
 - FIFO \rightarrow El mas antiguo
 - ↓ - El menos usado

Políticas de escritura

- WT (escribir siempre)
- CB (escribir en cache y escribo en MP cuando el dirty es 1)
- WA / asignación (copiamos el bloque de MP)
- WnA / no traer nada (escribir en MP)

Evaluación

$$T_{am} = t_h + t_p$$
$$T_{am} = \% \text{ escrituras} \cdot t_h + \% \text{ lecturas}$$
$$T_{am} = h \cdot t_h + m \cdot t_p$$

$$T_{exe} = N \cdot CPI \cdot T_c$$

\downarrow

accesos a mem

\downarrow

$N_r \cdot m \cdot t_p$

\nwarrow tiempo de penalización

\uparrow

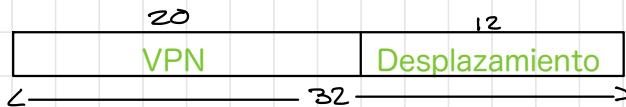
tasa de fallo

$CPI_{ideal} + CPI_{mem}$

Tema 8 - Memòria virtual

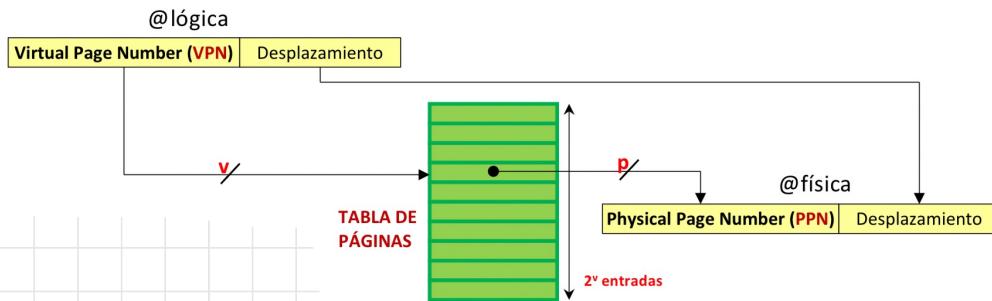
Cuando tenemos una @lògica (de 32 bits) y páginas de 4KB = 2^{12} B

Entonces en la dirección tenemos 12 bits para el desplazamiento y la resta és el VPN (Virtual page number)



En el caso que la dirección sea @física tenemos los mismos bits de desplazamiento 12 y lo siguiente és el PPN (physical page number)

El esquema és el siguiente:



La tabla de páginas se indexa con el VPN:

- PPN (Physical page number) que és la dirección del disco
- D (dirty bit) que dice si la línea esta modificada
- P (bit de presencia) que dice si la entrada és valida y esta en MP

La solución para implementar una tabla de páginas, la manera és tener una cache “especial” llamada TLB.

Esta cache solo tiene información para acelerar la traducción de direcciones.

El TLB:

- Es muy similar a la MC
- Tiene muy pocas entradas
- Es completamente asociativo o asociatividad muy alta
- La tasa de fallo es muy baja
- Muy rápido → 0,5 o 1 ciclo
- Algoritmo de remplazo (LRU o pseudoLRU)

Tenemos páginas de 4KB

La entrada es: (38B)

Dirección lógica de 32B (20B VPN)

Dirección física de 28B (16B PPN)

V	VPN	M	PPN
---	-----	---	-----

Como funciona la memoria virtual?

Esta permite ejecutar programas más grandes que la MP que tenemos actualmente.

También permite proteger el espacio de dirección de los programas de ser accedido por otros programas

La tabla de páginas tiene un bit de presencia que:
if (fallos página) {

Reemplazar página: MP → Disco

Cargar la página solicitada: Disco → MP

}

Hay que reducir al máximo los fallos de página porque son muy costosos

El sistema operativo gestiona totalmente la memoria virtual.

Se trae una página de disco a MP cuando hay algún fallo de página.

La página se ubica en cualquier marco (política totalmente asociativa)

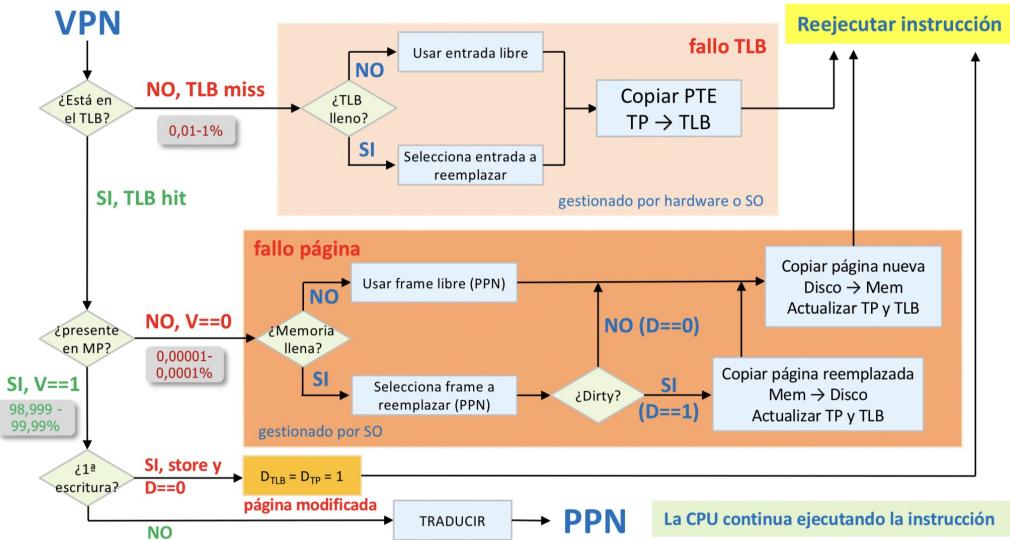
En caso de fallo se substituye en función de un algoritmo de remplazo muy sofisticado.

Las páginas modificadas hay que escribir las en disco

La tasa de fallos es 0,00001% - 0,001%

La política de escritura es CB+WA

Como funciona la memoria virtual en MIPS



Resumen coma flotante

Grandes $4,17 \cdot 10^3$ Pequeños $17,39 \cdot 10^{-15}$

El 0,2 no se puede representar con bits finitos

Todos los números en coma flotante tienen:

Un dígito para el signo

Una fracción

Una base y exponente



El exponente se codifica en exceso

El exceso depende del número de bits del exponente

Normalmente se usa la de 32bits en este curso

1 bit para el signo, 8 para el exponente y 23 para la fracción



Para calcular el valor del exponente hacemos: $255 - \text{exceso o } X_e = 127 + x$

Para sumar dos números en C.F.

1) Igualar los exponentes

Normalmente lo mas fácil es pasar el pequeño al grande, desplazando la fracción

2) Siempre se pone el mayor arriba entonces dependiendo del signo se resta o se suma el pequeño.

Signos iguales -> Sumar

Signos diferentes -> Restar mayor - menor

3) Normalizar

Para normalizar el numero hay que desplazar la fracción hasta que en la primera posición haya un 1

4) Redondear

Para redondear miramos los 3 últimos bits G R S

Si G es 0 se queda igual

Si G es 1 y R o S es 1 redondeamos hacia arriba

Si G es 1 y, R y S son 0 redondeamos al par

Ejemplo:

Convertir 49,175 en coma flotante

Primero cogemos el 49 y lo pasamos a binario

$$49 \rightarrow 110001$$

Para lo decimal multiplicamos por 2

Como movemos la coma 5 veces

$$\begin{array}{rcl} 0,175 \cdot 2 = 0,350 & 0 \\ 0,350 \cdot 2 = 0,7 & 0 \\ 0,7 \cdot 2 = 1,4 & 1 \\ 0,4 \cdot 2 = 0,8 & 0 \\ 0,8 \cdot 2 = 1,6 & 1 \\ 0,6 \cdot 2 = 1,2 & 1 \\ 0,2 \cdot 2 = 0,4 & 0 \end{array}$$

$$11\ 0001,001\overbrace{0110}^{\text{Fraccion}}$$

↓ Fraccion

$$1,10001001\overbrace{0110}^{\text{Fraccion}} \times 2^5$$

$$\leq \text{en exceso} \rightarrow xe = 127 + 5 = 132$$

↓

Se repite

$$10000\ 100$$

En la Fraccion hay que usar 23 bits

$$1,10001\ 0010110\ 0110\ 0110\ 0110\ 0110$$

12

↑

Como es 0
no redondeamos

$$\text{Ahora el } 13,5 \rightarrow 1101,1 \quad 1 \cdot 1011 \times 2^3$$

Se mueve la coma 3 veces
(Normalizar)

Para sumarlo con el anterior tenemos que poner el exponente igual

$$1 \cdot 1011 \times 2^3 \rightarrow$$

$$\begin{aligned} & 1 \cdot 100010010110011001100110011 \cdot 2^5 \\ + & 0.011010000000000000000000 \cdot 2^5 \\ & 1.111101010101100011001100110011 \cdot 2^5 \end{aligned}$$

Como calculamos el error:

Cojemos el ultimo bit $1 \cdot 2^{-23} \cdot 2^{\text{expo}}$

Repasso subrutinas:

Llamada a una subrutina (jal subr)

x = subr(par1, par2, par3);

Hay que poner los parámetros de izquierda a derecha, en los registros, \$a0, \$a1, \$a2 y el resultado esta en el registro \$v0.

Cuando pasamos un parámetro por valor, ponemos directamente el valor en el registro, però cuando és un puntero o un vector/matriz pasamos la dirección.

Cuando hay rutinas que llaman a otras, hay que guardar el registro \$ra para así poder volver a la anterior (dirección de retorno)

Para gestionar las subrutinas hay 2 cosas:

Tenemos las variables locales, estas se crean y destruyen.

Dentro de estas podemos tener escalares, vectores, matrices, etc.

Las variables escalares se guardan en registros.

Però los vectores o matrices se guardan en memoria (la pila)

Al entrar tenemos un BA (bloque de activación) que se apunta con el registro \$sp.

En este bloque se guardan primero las variables locales en el mismo orden que están escritos y SIEMPRE alineados a su tamaño y al final alineado a 4.

En el BA también se guardan las variables que se use su dirección (&x).

El tamaño SIEMPRE tiene que ser múltiplo de 4.

En los registros seguros (\$s0-\$s9), se guardan los registros que són vitales para el funcionamiento de la subrutina actual.

Estos se guardan en la pila (BA) junto al bloque \$ra.



Resumen memoria virtual

Usamos la memoria virtual principalmente para 3 cosas

- Proteger el acceso a lugares de memoria que no son del código
- Ejecutar un programa mas grande que la memoria física disponible
- Reubicar código de manera sencilla

La CPU lanza la dirección lógica esta va a parar a la TLB que es un "traductor", y esta envia a la MP la dirección física.

En el espacio lógico hay las páginas y en el espacio físico hay los frames

La traducción hace lo siguiente

Tenemos una @lógica que tiene: VPN y el desplazamiento dentro de la página

VPN	Desplazamiento
-----	----------------

Y la @física tiene: PPN y el mismo desplazamiento

PPN	Desplazamiento
-----	----------------

Entonces la TLB pasa de VPN a PPN para indexar el lugar de la MP

A parte del PPN en la TLB también tenemos un bit que nos representa si está en MP y también un bit que nos dice si la línea está sucia (si la hemos modificado)

Cuando el bit de presencia está a 0, e intentamos acceder se provoca un fallo de página, entonces tenemos que llevar desde disco la información a la MP.

Para reemplazar una página tendremos que mirar si el dirty bit está a 1, si es el caso no solo tenemos que acceder al disco para leer la información, si no que tenemos que también escribir esa página en disco

Cuando traemos una página la podemos poner en cualquier lugar de la MP

La memoria virtual principalmente la gestiona el OS, por lo que el porcentaje de fallo es muy bajo, y los algoritmos son muy sofisticados

El TLB es una especie de cache de páginas, esta guarda las últimas páginas que ha accedido un proceso, entonces tenemos un acceso mucho más rápido a las páginas más recientes.

Ejercicio 7.5

Esta en KB

i	0	0	0	0	2	0	4	0	6	1	8
8192	1	8	1	10	1	12	1	14	z	16	
8192	1	8	1	10	1	12	1	14	z	16	
16384	z	16	z	18	z	20	z	22	z	24	

0	0	2	4	6	8
8192	8	10	12	14	16
8192	8	10	12	14	16
16384	16	18	20	22	24
	2	2	2	2	3

Páginas de 8KB

De la dirección 0-7 pág 1

Total de fallos 4, una vez por página

8-15 pág 2

16-23 pág 3

24-32 pág 4

Páginas de 4KB Tendría un total de 9 fallos