

Edgar Aguadé Nadal

Quadern de laboratori Estructura de Computadors

Emilio Castillo
José María Cela
Montse Fernández
David López
Joan Manuel Parcerisa
Angel Toribio
Rubèn Tous
Jordi Tubella
Gladys Utrera

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Quadrimestre de Primavera - Curs 2014/15



Aquest document es troba sota una llicència Creative Commons

Licencia Creative Commons

Esta obra está bajo una licencia Reconocimiento-No comercial-Compartir bajo la misma licencia 2.5 España de Creative Commons. Para ver una copia de esta licencia, visite

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/>

o envíe una carta a

Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

- **Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- **No comercial.** No puede utilizar esta obra para fines comerciales.
- **Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Advertencia: Este resumen no es una licencia. Es simplemente una referencia práctica para entender el Texto Legal (la licencia completa).

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Sessió 1: Assemblador MIPS i tipus bàsics de dades

Objectiu: En aquesta sessió es programaran petits codis que treballen amb els tipus bàsics de dades del llenguatge assemblador MIPS: números naturals, enters i caràcters. En la primera part de la sessió pràctica s'estudiarà la representació d'aquests tipus de dades i les seves principals instruccions d'accés a memòria. En la segona part s'estudiarà el concepte de punter i es treballaran els accisos a les variables de tipus estructurats més senzills: vectors i strings (o vectors de caràcters).

Lectura prèvia

Tipus elementals de dades del MIPS

L'arquitectura del processador MIPS és de 32 bits (4 bytes), però permet accedir a dades de memòria de 8, 16 i 32 bits. No obstant el llenguatge permet declarar dades de fins a 64 bits. A continuació podem observar una correspondència entre els tipus bàsics de dades en llenguatge C, la seva declaració en assemblador del MIPS i l'espai que ocupen a memòria:

En C	Tipus	MIPS .data	#bits
char c; unsigned char c;	enter o caràcter ASCII natural	.byte 0	8
short s; unsigned short s;	enter natural	.half 0	16
int i; unsigned int i;	enter natural	.word 0	32
char *p; int *p; short *p; long long *p;	punter (natural)	.word 0	32
long long l; unsigned long long l;	enter natural	.dword 0	64

Instruccions d'accés a memòria

Les instruccions que utilitza l'assemblador MIPS per accedir als tipus bàsics de dades en memòria son:

Mida dades	Instruccions d'accés	Significat	Restriccions d'alignement
8 bits	lb \$1, -100(\$2)	Load byte	Cap restricció
	lbu \$1, -100(\$2)	Load byte unsigned	
	sb \$1, -100(\$2)	Store byte	
16 bits	lh \$1, -100(\$2)	Load halfword	Adreces múltiples de 2
	lhu \$1, -100(\$2)	Load halfword unsigned	
	sh \$1, -100(\$2)	Store halfword	
32 bits	lw \$1, -100(\$2)	Load word	Adreces múltiples de 4
	sw \$1, -100(\$2)	Store word	

Punters

Un punter és una variable que conté l'adreça de memòria on es troba ubicada una altra variable. Si la variable *p* conté l'adreça de la variable *v* diem que *p* "apunta" a *v*. En MIPS aquesta adreça ocupa 32 bits i es pot declarar en assemblador MIPS de la següent manera:

```
.data
p: .word 0
```

La variable punter *p* es pot inicialitzar en la seva pròpia declaració, indicant l'adreça de la variable a la que ha d'apuntar:

```
.data
var: .byte 'e'
p: .word var
# p 'apunta' a la variable "var" ja que conté la seva adreça
```

O bé, es pot inicialitzar en el codi:

```
.data
var: .byte 'e'
punter:.word 0

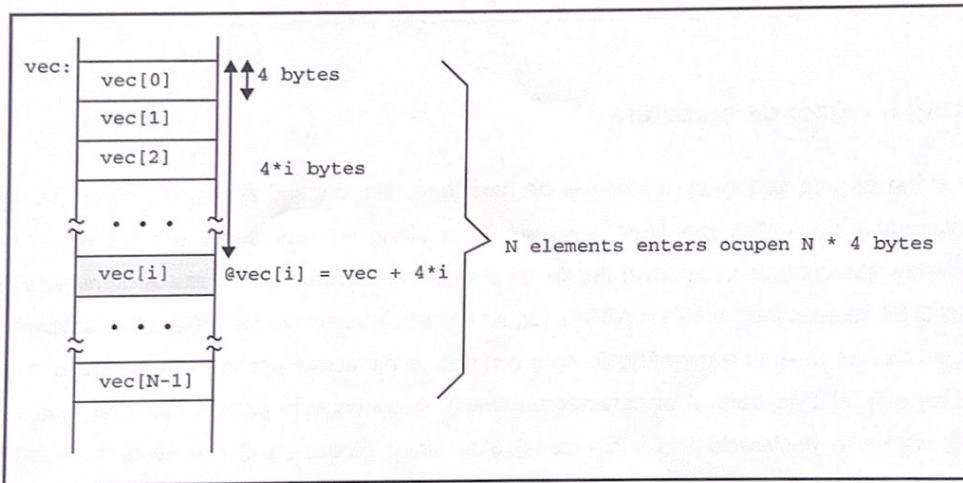
.text
la    $s0, var
la    $s1, punter
sw    $s0, 0($s1)
```

Vectors

Un vector és un conjunt unidimensional d'elements del mateix tipus, els quals s'emmagatzemem en memòria consecutivament a partir de l'adreça inicial del vector, respectant les regles d'alignació dels elements. Cada element s'identifica amb un índex, i en llenguatge C aquest es numera començant sempre des de 0 per al primer element fins a N-1 per a l'últim (essent N el nombre d'elements del vector). Per accedir a un element d'un vector, l'índex indica quants elements s'han de saltar des de l'adreça inicial per trobar l'element que es busca. Així, per calcular l'adreça de l'element amb índex i , hem de fer el següent càlcul:

```
@v[i] = @v + (i * mida_d'un_element)
```

Per exemple, a la següent figura es pot observar la representació en memòria d'un vector `vec` de N elements enters.



En assemblador MIPS es pot declarar una variable global del tipus vector inicialitzant els seus elements un per un:

```
nomvector:    directiva valor_element_0, valor_element_1, ...
```

on la *directiva* d'assemblador pot ser `.byte`, `.word`, `.half` o bé `.dword`, segons la mida dels elements. Però si el vector té un elevat nombre d'elements i volem inicialitzar-los amb el valor 0, llavors podem usar la directiva `.space`, la qual reserva un determinat espai a memòria:

```
nomvector:    .space  num_elements * bytes_per_element
```

Caràcters

Existeixen moltes codificacions que associen un caràcter amb una representació numèrica binària. En els nostres programes farem servir la codificació ASCII de 7 bits (el bit 7 a zero), per codificar els caràcters alfanumèrics més representatius ('a', 'b',..., 'z', 'A', 'B',..., 'Z', '0', '1',..., '9'), els signes de puntuació (coma, punt, punt i coma, dos punts, parèntesi,...) i altres símbols (TAB, LF, CR,...), tal i com podem observar en la següent taula:

valor	símbol	valor	símbol
0x00	null	0x30	'0'
...	...	0x31	'1'
0x09	TAB
0x0A	LF	0x41	'A'
...	...	0x42	'B'
0x0D	CR
...	...	0x61	'a'
0x20	'' (espai)	0x62	'b'
...

String o vector de caràcters

Un string és una seqüència ordenada de caràcters, de longitud arbitrària i finita. Es pot implementar de moltes maneres, però en C un string es representa per un vector de caràcters que conté a continuació del darrer caràcter de la cadena un caràcter-sentinella de valor 0 (el caràcter 'null' = '\0' en ASCII). Els strings es declaren com a vectors de caràcters, i l'espai que es reserva a memòria és com a mínim el necessari per ubicar els caràcters de l'string més el byte dedicat al caràcter-sentinella. A continuació podem observar diverses variants de la declaració i la inicialització d'un string global en C i la seva traducció a assemblador:

C	MIPS
<pre>/* Dues declaracions equivalents */ char cadena1[20] = {'u','n','a','f','r','a','s','e','\0'}; char cadena1[20] = "una frase";</pre>	<pre># Dues declaracions equivalents .data cadena1: .ascii "una frase" # 9 caràcters, sense sentinel·la .space 11 # Completa la resta amb zeros cadena1: .asciiz "una frase" # 9 caràcters més el sentinel·la .space 10 # Completa la resta amb zeros</pre>

Pseudo-instruccions del MIPS

El llenguatge assemblador MIPS permet definir noves pseudo-instruccions (també anomenades macros) a partir d'una o més instruccions MIPS. L'assemblador de MIPS les reconeix i les expandeix a les corresponents instruccions abans de traduir-les a codi màquina. Les macros serveixen per facilitar l'escriptura i la lectura del codi.

Un exemple clar són les macros **la** (load address) i **li** (load immediate). La macro **la** permet carregar l'adreça de memòria d'una variable global en un registre a partir de la seva etiqueta. Com que una adreça de memòria no hi cap completament en una instrucció (tenen la mateixa mida 32 bits), la macro **la** sempre s'expandirà en dues instruccions, una que carrega la part alta de l'adreça i l'altra que carrega la part baixa. D'altra banda, la macro **li** permet carregar un operand immediat en un registre, però en funció del valor d'aquest operand s'expandirà en una o dues instruccions (s'estudia en detall a l'activitat 1.B).

Crides al sistema operatiu del MIPS (Syscall)

Una crida al sistema és el mecanisme usat per una aplicació o un processador per sol·licitar un servei al sistema operatiu. MIPS posa a disposició del programador un determinat conjunt de crides al serveis del sistema operatiu, principalment per a operacions d'entrada/sortida. En aquesta pràctica usarem aquestes crides per mostrar en pantalla resultats de l'execució.

Per fer servir les crides al sistema, el programa ha de seguir els següents passos:

- Pas 1. Copiar l'identificador del servei al registre \$v0.
- Pas 2. Copiar els arguments als registres especificats. En les crides que farem servir únicament s'usa \$a0.
- Pas 3. Executar la instrucció *syscall*.
- Pas 4. Recuperar el valors de retorn, si és el cas, en els registres específicats.

Les principals crides que usarem en aquestes pràctiques i que l'alumne ha de saber són¹:

Syscall	Identificador	Arguments	Exemples
print integer	\$v0 = 1	\$a0 = Enter a imprimir	li \$v0, 1 li \$a0, 18 syscall # Mostra l'enter 18 en pantalla
print string	\$v0 = 4	\$a0 = Adreça inicial del string a imprimir	li \$v0, 4 la \$a0, string syscall # Mostra el string en pantalla
print character	\$v0 = 11	\$a0 = Caràcter a imprimir	li \$v0, 11 li \$a0, 'E' syscall # Mostra el caràcter 'E' en pantalla

1. Per a més informació podeu consultar l'ajuda del simulador MARS (Opció help, pestanya syscalls)

Enunciats de la sessió

Activitat 1.A: Declaracions amb alineació en memòria automàtica

Exercici 1.1: Tradueix a assemblador la següent declaració de variables globals en C:

C	Assemblador MIPS
.data	.data
char aa = -5;	aa: .byte -5
short bb = -344;	bb: .half -344
long long cc = -3;	cc: .dword -3
unsigned char dd = 0xA0;	dd: .byte 0xA0
int ee = 5799;	ee: .word 5799
short ff = -1;	ff: .half -1

Exercici 1.2: Sabent que les dades globals s'emmagatzemem a partir de l'adreça 0x10010000, escriviu el contingut de memòria de la declaració de l'exercici anterior, byte per byte, en ordre little-endian, i escriviu cada etiqueta a la posició que correspongui. Indiqueu amb una 'X' les posicions de memòria que el compilador deixa sense ocupar a fi d'alinear les dades (per defecte l'alineació és automàtica):

Etiqueta	@Memòria	Contingut	Etiqueta	@Memòria	Contingut
aa:	0x10010000	0xFB		0x1001000E	0xFF
	0x10010001	X		0x1001000F	0xFF
bb:	0x10010002	0xA8	dd:	0x10010010	0xA0
	0x10010003	0xFE		0x10010011	X
	0x10010004	X		0x10010012	X
	0x10010005	X		0x10010013	X
	0x10010006	X	ee:	0x10010014	0xA7
	0x10010007	X		0x10010015	0x16
cc:	0x10010008	0xFD		0x10010016	0x00
	0x10010009	0xFF		0x10010017	0x00
	0x1001000A	0xFF	ff:	0x10010018	0xFF
	0x1001000B	0xFF		0x10010019	0xFF
	0x1001000C	0xFF		0x1001001A	
	0x1001000D	0xFF		0x1001001B	

Comprovació pràctica

Engegueu el simulador MARS i carregueu el fitxer **s1a.s**. Copieu la declaració de les variables de l'exercici 1.1 i premeu F3 per assemblar el programa. Comproveu els continguts de memòria de l'exercici 1.2 amb els valors que apareixen a la vista de dades. Recordeu que aquesta vista mostra el contingut de memòria en format word (paraules de 4 bytes ordenades en little-endian).

Activitat 1.B: Inicialització de registres amb immediats i adreces

Feu una còpia del fitxer **s1a.s** amb el nom **s1b.s**. Inserteu en aquest fitxer el següent codi en assemblador MIPS i assembleu-lo. Observant la vista de Codi desassemblat del MARS, indiqueu en quines instruccions s'expandeixen cada una de les següents macros:

Macros MIPS	Instruccions MIPS
la \$s3, aa	lui \$1, 0x1001 ore \$19,\$1, 0x0000
li \$s4, 65535	ore \$20,\$0, 0xffff
li \$s5, 65536	lui \$1, 0x0001 ore \$21,\$1, 0x0000
move \$s0, \$s1	addu \$16,\$0,\$17

Activitat 1.C: Accés a variables de tipus elemental en memòria

Exercici 1.3: Les instruccions en negreta del següent codi accedeixen a memòria per llegir (o escriure) les variables globals de l'exercici 1.1. Escriviu, per a cada variable del programa l'adreça i mida. També escriviu el valor final dels registres destinació de les instruccions de load que hi accedeixen (ressaltades en negreta) o el contingut de memòria (en cas d'escriptura) a partir dels resultats calculats a l'exercici 1.2:

Codi assemblador MIPS	Adreça efectiva d'accés a memòria	Núm bytes accedits	Valor llegit/escrit (hex 32/64 bits)
main:	0x10010000	1	0xfffffffffb
	0x10010002	2	0xffffffeaa8
	0x10010008 0x1001000C	4 + 4	0xfffffffffffffd
	0x10010010	1	0x0000000a0
	0x10010018	2	0xffffffffff
	0x10010018	4	0x0000ffffb

Comprovació pràctica

Feu una còpia del fitxer **s1a.s** amb el nom **s1c.s**. Afegiu-hi el codi anterior i executeu el programa pas a pas (tecla F7), tot comprovant que les respostes anteriors són correctes.

Activitat 1.D: Operacions amb punters a variables globals

Exercici 1.4: Donada la següent declaració de dades en assemblador MIPS (un punter inicialitzat amb l'adreça d'una altra variable global), i suposant que les variables estan emmagatzemades en memòria a partir de l'adreça 0x10010000, escriviu el valor en hexadecimal de cada una de les següents expressions en C:

```
dada:    .data
          .half 3
pdada:   .word dada
```

&pdada	0x10010004	&dada	0x10010000
pdada	0x10010000	dada	0x00003
*pdada	0x0003		

Activitat 1.E: Accés indirecte a una variable a través d'un punter

Exercici 1.5: Traduïu a assemblador MIPS el següent programa escrit en C, omplint les caselles en blanc. Considereu que la variable `temp` es guardarà al registre \$s0:

C	Assemblador MIPS
<pre> int A[3] = {3, 5, 7}; int *punter = 0; void main() { int temp; punter = &A[2]; temp = *punter + 2; temp = *(punter-2) + temp; A[1] = temp; print_integer(temp); // Consultar lectura prèvia // main retorna al codi de startup } </pre>	<pre> .data A: .word 3,5,7 Punter: .word 0 .text .globl main main: la \$t0, punter la \$t1, A addiu \$t1,\$t1,8 sw \$t1,0(\$t0) lw \$s0,0(\$t1) addiu \$s0,\$s0,2 lw \$t2,-8(\$t0) addu \$s0,\$s0,\$t2 la \$t1,A sw \$s0,4(\$t1) li \$v0,1 move \$a0,\$s0 syscall jr \$ra </pre>

Comprovació pràctica

Copieu el codi anterior al fitxer `s1e.s`. Salveu-lo, assembleu-lo i executeu-lo. Comproveu que el programa mostra per la consola d'entrada/sortida del simulador MARS el número 12, valor de la variable temporal `temp`. Comproveu també a la vista de dades que `A[1]` val 12.

Activitat 1.F: Tipus estructurats de dades: el vector

Exercici 1.6:

Donat el següent vector global vec de 10 elements de tipus enter:

```
int vec[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

A continuació, escriviu la declaració del vector vec en assemblador MIPS:

```
vec: .word 9,8,7,6,5,4,3,2,1,0
```

Escriviu també la fórmula per al càlcul de l'adreça de l'element `vec[i]`, en funció de l'adreça inicial de `vec` i del valor de l'índex `i`:

```
@vec[i] = &vec + i*4
```

A partir de la fórmula anterior, escriviu un fragment de codi en assemblador MIPS tal que copii en el registre \$s1 el valor de `vec[i]`, és a dir: `$s1 <- vec[i]`, suposant que el valor de `i` es troba al registre \$s2.

```
la sto,vec
sll s1,$s2,2 #i*4
addu sto,st0,$s1+vec($s2) (adresa)
lw s1,0($sto)
```

Activitat 1.G: Accés aleatori als elements d'un vector

Suposem un vector global de 10 elements enters fib. El codi en C mostrat a continuació escriu en els 10 elements del vector els 10 primers valors de la sèrie de Fibonacci

```
int fib[10];

void main() {
    int i = 2;
    fib[0] = 0;
    fib[1] = 1;
    while (i < 10){
        fib[i] = fib[i-1] + fib[i-2];
        i++;
    }
}
```

Completeu a continuació l'exercici 1.7.

Exercici 1.7: Traduïu a assemblador MIPS el següent programa escrit en C, omplint les caselles en blanc. Considereu que la variable *i* es guardarà al registre \$s0:

C	Assemblador MIPS
<pre> int fib[10]; void main() { int i = 2; fib[0] = 0; fib[1] = 1; while (i < 10) { fib[i] = fib[i-1] + fib[i-2]; i++; } // main retorna al codi de startup } </pre>	<pre> .data fib: .space 40 .text .globl main main: li \$s0, 2 la \$t1, fib sw \$zero, 0(\$t1) li \$t2, 1 sw \$t2, 4(\$t1) while: slti \$t0, \$s0, 10 beq \$t0, \$zero, fi sll \$t2, \$s0, 2 # i*4 addu \$t2, \$t1, \$t2 lw \$t3, -4(\$t2) lw \$t4, -8(\$t2) addu \$t0, \$t3, \$t4 sw \$t0, 0(\$t2) addiu \$s0, \$s0, 1 b while fi: jr \$ra </pre>

Comprovació pràctica

Copieu el codi de l'exercici 1.7 a l'arxiu **s1g.s**. Salveu-lo, assembleu-lo i executeu-lo. Comproveu en la zona de memòria que el contingut del vector **fib** és 0 1 1 2 3 5 8 13 21 34.

Activitat 1.H: Cadenes de caràcters (strings)

Sigui el vector de naturals `vec`, el qual conté els díigits (números del 0 al 9) de la representació en decimal del número natural `num=19865`. El primer element del vector representa el dígit de menor pes. El següent programa en C converteix cada un dels elements del vector `vec` a la seva representació ASCII i els emmagatzema en el string `cadena`:

```
char cadena[6];
unsigned int vec[5] = {5, 6, 8, 9, 1};

void main() {
    int i=0;
    while (i<5)
    {
        cadena[i] = vec[4-i] + '0';
        i++;
    }
    cadena[5]=0;      // posa la marca de final de string
    print_string(cadena);
}
```

Observeu que el programa escriu a l'string `cadena` els díigits decimals de `num` començant pel de major pes, ja que volem que es pugui llegir el número correctament quan imprimim el string per pantalla. Per aquesta raó, a cada iteració del bucle es converteix el dígit `vec[5-1-i]` en comptes de convertir el dígit `vec[i]`. Noteu que la conversió a ASCII es fa sumant 48 al dígit en decimal, o el que és el mateix, el codi ASCII de '0'. Fixeu-vos també que quan es declara el vector de caràcters `cadena` es reserva espai per a 5+1 elements, per tal de poder guardar el valor *sentinella* 0, que assenyala el final del string.

Completeu a continuació l'exercici 1.8

Exercici 1.8: Traduïu a assemblador MIPS el següent programa escrit en C, omplint les caselles en blanc. Considereu que la variable *i* es guardarà al registre \$s0:

C	Assemblador MIPS
<pre> char cadena[6]={-1,-1,-1,-1,-1,-1}; unsigned int vec[5]={5, 6, 8, 9, 1}; void main() { int i=0; while (i < 5) { cadena[i]=vec[4-i] + '0'; i++; } cadena[5]=0; print_string(cadena); // consulteu lectura prèvia // main retorna al codi de startup } </pre>	<pre> .data Cadena: .byte -1,-1,-1,-1,-1,-1 Vec: .word 5,6,8,9,1 .text .globl main main: li \$s0,0 la \$t7, cadena la \$t6, Vec +16 while: li \$t0,5 bge \$s0, \$t0, fi lbu \$t0, 0(\$t6) addiu \$t0, \$t0, 0x30 sb \$t0, 0(\$t7) addiu \$t7, \$t7, 1 addiu \$t6, \$t6, -4 addiu \$s0, \$s0, 1 b while fi: sb \$zero, 0(\$t7) li \$s0,4 addiu \$s0, \$t7, -5 syscall jr \$ra </pre>

Comprovació pràctica

Copieu el codi de l'anterior exercici a l'arxiu **s1h.s**. Verifiqueu el correcte funcionament del programa de manera que imprimeixi la cadena de caràcters: "19865". Comproveu també a la vista de dades que al final del programa la variable *cadena* representa aquest mateix número.

Alerta, perquè la vista de dades mostra la memòria en format word:

Per exemple, suposem el string "0123456". Estaria format pels 8 elements 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x00 (sentinella), i es guardarien en memòria en aquest mateix ordre. Però MARS mostra a la vista de Dades tot el contingut de la memòria suposant que tot són words (agrupant els bytes de 4 en 4). Així doncs, el primer word del string estaria format pels bytes 0x30, 0x31, 0x32, 0x33, amb el byte 0x30 guardat en primer lloc. El primer byte és el de menys pes dels quatre si els interpretem com un word, de manera que en l'escriptura normal en hexadecimal, tal com ho mostra MARS a la vista de dades, apareix escrit a la dreta: 0x33323130. De la mateixa manera, el segon word apareixeria escrit: 0x00363534.

Anàlogament, la variable cadena del nostre exercici hauria de contenir la seqüència "19865" que està formada pels bytes: 0x31, 0x39, 0x38, 0x36, 0x35, 0x00, guardats en memòria en aquest ordre. Així doncs, ¿com s'hauria de mostrar la cadena, en format word hexadecimal, a la vista de dades?

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000			...
0x10010020			...