ESTRUCTURA DE COMPUTADORS
Grau en Enginyeria Informàtica

Sessió de laboratori número 9

VARIABLES I PAS DE PARÀMETRES

Objectius

- Entendre i usar les instruccions i pseudoinstruccions per a la lectura i l'escriptura en la memòria principal.
- Fer servir les funcions del sistema que permeten entrada i eixida de cadenes de caràcters.
- Manipular adreces i recórrer vectors.
- Fer procediments amb arguments de tipus punter.

Bibliografia

- D.A. Patterson i J. L. Hennessy, *Estructura y diseño de computadores*, Reverté, capítol 2, 2011.

Introducció teòrica

Variables estàtiques i directives relacionades

(Aquesta secció és un repàs de la pràctica 1) L'assemblador del MIPS ofereix aquests recursos per a declarar les variables estàtiques d'un programa en la memòria:

- El segment `.data` on ubicar les dades en la memòria.
- La directiva `.space` permet reservar memòria del segment de dades. Útil per a declarar variables sense inicialitzar.
- Les directives `.byte`, `.half`, `.word`, `.ascii` i `.asciiz` permeten definir variables i inicialitzar-les.

Instruccions i pseudoinstruccions d'accés a la memòria de dades

(Aquesta secció és un repàs de la pràctica 1) El joc d'instruccions del MIPS per a lectura i escriptura de dades en la memòria comprèn vuit instruccions:

unitat	restriccions sobre l'adreça	lectura amb extensió de signe	lectura sense extensió de signe	Escriptura
byte	cap	`lb`	`lbu`	`sb`
halfword	múltiple de 2	`lh`	`lhu`	`sh`
word	múltiple de 4	`lw`		`sw`

Taula 1. Les instruccions de lectura i escriptura d'enters

Totes elles són del format I i en assemblador s'escriuen de la forma `op rt,D(rs)`. `D` és un **desplaçament** de 16 bits (amb signe) que se suma al contingut del registre **base** `rs` per tal de formar l'adreça de la memòria on es llig o s'escriu. Aquesta manera d'especificar l'adreça permet accedir a la memòria amb diverses intencions que referim tot seguit.

L'**adreçament absolut** es fa a una paraula fixa en la memòria de la què es coneix la posició *A*: en principi només caldria considerar la constant *A* com desplaçament i el registre `$zero` com a base. Amb aquest propòsit us convé utilitzar les pseudoinstruccions de la forma `op rs,A`, que es descomponen en les instruccions de màquina adients quan *A* ocupa més de 16 bits ($A \geq 2^{16}$).

Per exemple, la pseudoinstrucció `lw $rt,A` permet expressar la càrrega d'un registre amb el valor d'una variable en memòria ubicada en la posició que s'ha etiquetat com `"A"`. Aquesta línia pot traduir-se en una o més instruccions màquina, depenent del valor de l'etiqueta:

- Si `A` és un nombre expressable en 16 bits, la traducció és `lw $rt,A($0)`.
- Si `A` és massa gran per això, l'assemblador la descompon en la part alta `Ah` i la part baixa `Al`. Una traducció pot ser:

```
lui $at,Ah
lw $rt,Al($at)
```

Adreçament indirecte, quan la posició de la variable està en un registre. És la visió del programador quan ha d'accedir a una adreça calculada pel programa, o per a seguir un punter (en parlarem més avall) o per a recórrer variables estructurades.

Adreçament relatiu a registre, quan un registre conté una adreça de referència i el programador pensa en desplaçaments respecte d'ella. Aquest adreçament també s'utilitza per a accedir a variables estructurades: el registre conté l'adreça de la variable i el desplaçament és el corresponent al camps a què s'accedeix.

Punters

Es diu punter a qualsevol variable (en un registre o en la memòria) que continga una adreça de la memòria principal. El comptador de programa, per exemple, és el punter que conté l'adreça de la pròxima instrucció que s'ha de llegir i executar —de fet, hi ha processadors on el comptador de programa rep el nom de *Instruction Pointer* (IP).

Els programadors que utilitzen punters se'n fan la imatge mental de sagetes que assenyalen o *apunten a* (d'ahí el nom de punter) un lloc de la memòria i diuen, per exemple, que el PC *apunta a* la instrucció que el processador descodificarà i executarà tot seguit; o que, després d'executar-se la instrucció `jal F`, el registre `$ra` *apunta a* la instrucció (la següent a la crida) on tornarà el flux d'execució al final de la funció `F`, en executar-se `jr $ra` que salta a la instrucció *on apunta* `$ra`.

La pseudoinstrucció `la` (*load address*) és similar a `li`, però el seu nom fa explícit que **assigna** una adreça a un registre. És, per tant, la pseudoinstrucció que permet que un registre apunte a una posició de memòria representada per una etiqueta.

També els punters tenen la seua pròpia **aritmètica**. Sumar o restar constants al valor d'un punter es visualitza com que el punter *es desplaça* a una adreça més amunt o més avall en la memòria. Per exemple, en cada cicle d'instrucció del processador, en sumar-li 4, el PC *avança* per a apuntar a la següent instrucció, i si s'executa una bifurcació, que suma un valor amb signe al PC, aquest *es desplaça* cap amunt o cap avall tantes paraules com està codificat en la instrucció.

L'aritmètica amb els punters s'ha de fer sense signe ja que les adreces s'expressen en codi binari natural (no té sentit parlar-ne de signe ja que són nombres naturals); per tant, l'actualització dels punters es fa mitjançant les instruccions `addu` i `addiu`.

Els punters en **alt nivell**: mentre que en el llenguatge Java els punters s'amaguen, en C els punters se declaren i manipulen explícitament. A continuació teniu algunes equivalències entre codi C i assemblador de fragments de codi que manegen punters.

`int A = 4;`
`int * p;`
	`        .data`
	`A:      .word 4`
	`p:      .space 4`

`p = &A; /* p apunta a A */`	`        .text`
	`        la $s0,A`
	`        sw $s0,p`
	o bé:
	`        la $s0,A`
	`        la $s1,p`
	`        sw $s0,0($s1)`

`*p = *p + 1; /* incrementa l'enter`
`                a què apunta p */`
	`        la $s0,p`
	`        lw $s1,0($s0)`
	`        lw $s2,0($s1)`
	`        addi $s2,$s2,1`
	`        sw $s2,0($s1)`

Paràmetres per referència

Els paràmetres de les funcions d'usuari poden ser de dues classes:

- Els paràmetres **per valor** són dades, com els que hem vist fins ara. Per a fer servir una funció, el programa que la crida ha d'assignar un valor al paràmetre.
- Els paràmetres **per referència** són adreces. Per fer servir la funció, el programa que la crida ha d'assignar una adreça al paràmetre.

En els llenguatges d'alt nivell també pot donar-se aquesta distinció. En C, els paràmetres de tipus no estructurat són sempre per valor. Com el llenguatge permet manejar explícitament els punters, els programadors poden passar el punter a una variable. Els paràmetres estructurats (vectors, estructures) es passen sempre per referència.

Les funcions del sistema que processen cadenes prenen sempre com a un dels arguments d'entrada l'adreça de memòria on es troba la cadena. A continuació referim les crides al sistema del simulador PCSpim per a llegir i imprimir cadenes de caràcters.

$v0	Nom	Descripció	Arguments	Resultat
4	*print_string*	Imprimeix una cadena de caràcters acabada en nul ('\0')	`$a0` = punter a la cadena	—
8	*read_string*	Llig una cadena de caràcters (de llargària limitada) fins trobar un '\n' i la desa en el buffer acabada en nul ('\0')	`$a0` = punter al buffer d'entrada `$a1` = nombre màxim de caràcters de la cadena	—

Taula 2. Funcions del sistema per a l'entrada/eixida de cadenes de caràcters

Variables estructurades en assemblador

Per a declarar variables estructurades, tenim dues alternatives, depenent si la variable està inicialitzada o no.

- Ací teniu un vector de quatre components de tipus enter inicialitzat:

```
          .data 0x10000000
vector:   .word 3, -9, 2, 7
```

- Un vector de la mateixa talla, però no inicialitzat, podría declarar-se així:

```
          .data 0x10000000
vector:   .space 16
```

Noteu que, en ambdós casos, només s'hi defineix una etiqueta, que indica l'adreça on comença la variable. En el cas d'un vector `v` qualsevol, això correspon a l'element `v[0]`.

Accés a variables estructurades en assemblador

Per a accedir a les variables estructurades en assemblador s'empra una adreça base (punter a la primera component de la variable) i un desplaçament per tal d'accedir a les seues components. A tall d'exemple, a continuació mostrem un programa en assemblador que recorre un vector d'enters i incrementa cada component en una unitat. L'accés a les

components del vector es fa mitjançant adreçament **indirecte**. Fixeu-vos que l'aritmètica de punters es fa amb aritmètica sense signe.

```
          .data 0x10000000
vector:   .word 3, -9, 2, 7
          .globl __start
          .text 0x00400000

__start:  la $s0, vector         # Punter a vector[0]
          li $s1, 4              # Dimensió del vector
bucle:    lw $t0, 0($s0)         # Llegeix vector[i]
          addi $t0, $t0, 1       # Incrementa vector[i]
          sw $t0, 0($s0)         # Escriu vector[i]
          addi $s1, $s1, -1      # Decrementa comptador
          addiu $s0, $s0, 4      # Actualitza punter a vector[i+1]
          bgtz $s1, bucle
```

També podríem haver emprat l'adreçament **relatiu** per tal d'accedir al vector, a partir d'una adreça base i un índex que se li suma o resta en lloc d'usar només un registre punter. El codi següent ho mostra. En ell, la lectura de cada component s'hi fa amb la pseudoinstrucció `lw $t0,vector($s0)`, que es tradueix en instruccions màquina en funció del valor numèric de l'etiqueta `vector`. En aquest exemple, `vector` és l'adreça base i el valor en $s0 l'índex que se suma a la base. Noteu que ara, com no manipulem directament l'adreça de memòria, el desplaçament s'hi calcula amb aritmètica amb signe, atès que la base o l'índex poden ser negatius.

```
          .data 0x10000000
vector:   .word 3, -9, 2, 7
          .globl __start
          .text 0x00400000

__start:  li $s0, 0              # Desplaçament inicial
          li $s1, 4              # Dimensió del vector
bucle:    lw $t0, vector($s0)    # Llegeix vector[i]
          addi $t0, $t0, 1       # Incrementa vector[i]
          sw $t0, vector($s0)    # Escriu vector[i]
          addi $s1, $s1, -1      # Decrementa comptador
          addi $s0, $s0, 4       # Actualitza l'índex a vector[i+1]
          bgtz $s1, bucle
```

Registres o memòria? Detalls a no perdre de vista...

Les variables poden estar en un registre del banc o en la memòria principal.

Quan una variable està en la memòria, la seua identitat és una adreça en comptes d'un identificador de registre.

El nombre de registres és molt limitat comparat amb la capacitat de la memòria principal.

Amb variables en la memòria no es pot fer directament ni aritmètica ni control de flux, cal dur-les al banc de registres per poder manipular-les.
Les variables en memòria poden ser més grans que 32 bits.
Les variables en memòria poden tenir un valor inicial en carregar el programa; tanmateix, les variables en un registre necessiten una instrucció per a inicialitzar-les.

Exercicis de laboratori

Exercici 1: Paràmetres per referència

En aquest primer exercici anem a considerar el programa que vam tractar a la pràctica 3 que calculava el producte de dos nombres enters introduïts pel teclat, `M` i `Q`, i imprimia el resultat `R`. Aquella pràctica partia d'un arxiu font que contenia el programa principal i la funció `Mult` i calia afegir-li dues noves funcions, `Input` i `Output`, per a millorar el diàleg amb l'usuari a través de la consola. El pseudocodi de les tres funcions, a grans trets, era el següent:

```
int Input(char $a0) {
        print_char($a0);
        print_char('=');
        $v0=read_int();
        return($v0); }
```

```
void Output(char $a0, int $a1) {
        print_char($a0);
        print_char('=');
        print_int($a1);
        print_char('\n');
        return; }
```

```
int Mult (int $a0, $a1) {
        $v0 = $a0 * $a1;
        return($v0); }
```

El nostre objectiu és reproduir de nou un diàleg como el següent: (en cursiva apareix el text teclejat per l'usuari):

```
M=215
Q=875
R=188125
```

En aquest cas, tanmateix, les variables `M`, `Q` i `R` van a ubicar-se en el segment de dades de la memòria; recordeu que en la pràctica 3 aquestes variables estaven emmagatzemades en registres. Així doncs, aneu a construir tres funcions noves, anomenades `InputV`, `OutputV` i `MultV`. Les dues primeres aprofiten, ja sabeu, per a introduir valors pel teclat i imprimir resultats en pantalla, respectivament; la tercera s'encarrega de calcular el producte de dos enters. A diferència de les funcions dissenyades en la pràctica 3 (`Input`, `Output` i `Mult`) que rebien els paràmetres per valor a través de registres, les noves reben els paràmetres també en els registres però ara ho fan per **referència**, això és, en el registre hi ha el *punter* o adreça de memòria de la variable amb què treballen.

Vegeu un exemple en pseudocodi del programa principal `main()` i de la funció per a introduir valors pel teclat `void InputV(char lletra, int *var)` aquesta rep el caràcter amb què retolarem l'entrada del teclat (per valor) i l'adreça de memòria on s'emmagatzemarà el valor llegit (és a dir, que passa el resultat per referència).

```
main() {
        int M;
        $a0 = 'M';
        $a1 = &M;
        InputV($a0, $a1);
        exit; }
```

```
void InputV(char $a0, int *$a1) {
        print_char($a0);
        print_char('=');
        *$a1 = read_int();
        return; }
```

Com a punt de partida d'aquest exercici us proporcionem en el fitxer "09_exer_01.s" el codi assemblador següent equivalent al pseudocodi anterior que il·lustra la declaració d'una variable `M` localitzada en memòria i la seua inicialització des del programa principal amb l'ajuda de la funció `InputV`:

```
          .globl __start
          .data 0x10000000
M:        .space 4

          .text 0x00400000
__start:  li $a0,'M'
          la $a1, M
          jal InputV
          li $v0,10
          syscall

InputV:   li $v0, 11
          syscall
          li $v0, 11
          li $a0, '='
          syscall
          li $v0, 5
          syscall
          sw $v0, 0($a1)
          jr $ra
```

Després de comprendre el programa de partida, carregueu-lo i executeu-lo amb el simulador PCSpim. No dubteu a formatar-lo i afegir-hi comentaris per fer més clar el seu propòsit.

- En acabar l'execució del programa, on està el valor de la variable que heu llegit? **Tècnica experimental:** interpreteu la finestra *data segment* del simulador.
- Si en el programa principal volguéreu sumar 1 a la variable `M` tot just l'heu llegida amb `InputV`, quines opcions de les següents serien correctes?

a)

```
…
jal InputV
addi $a1,$a1,1
```

b)

```
…
jal InputV
lw $s0,M
addi $s0,$s0,1
```

c)

```
…
jal InputV
lw $s0,M
addi $s0,$s0,1
sw $s0,M
```

d)

```
…
jal InputV
lw $s0,0($a1)
addi $s0,$s0,1
sw $s0,0($a1)
```

e)

```
…
jal InputV
addi $v0,$v0,1
```

f)

```
…
jal InputV
li $s0,M
addi $s0,$s0,1
```

Ara estem en condicions de reescriure les funcions `OutputV` i `MultV`, el pseudocodi de les quals mostrem tot seguit. El resultat de la multiplicació és de 32 bits, així que només interessa el valor del registre LO.

```
void OutputV(char $a0, int *$a1) {
    print_char($a0);
    print_char('=');
    print_int(*$a1);
    return; }
```

```
void MultV(int *$a0, int *$a1, int *$a2) {
    $t0 = *$a0;
    $t1 = *$a1;
    $t0 = $t0+$t1;
    *$a2=$t0 ;
    return; }
```

El programa principal haurà de cridar les funcions dissenyades de la manera indicada en el pseudocodi següent (el símbol "*&*" representa l'adreça de la variable que el segueix):

```
main() {
    int M, Q, R;
    InputV('M', &M);
    InputV('Q', &Q);
    MultV(&M, &Q, &R);
    OutputV('R', &R);
    exit; }
```

Una vegada comprovat que el programa funciona correctament, contesteu les qüestions següents:

- En quina adreça de memòria es troba emmagatzemada la variable `R`?
- Executeu el programa amb els valors `M=5` i `Q=-5`. Consulteu el segment de dades del programa i localitzeu-hi els valors de les variables `M`, `Q` i `R` emmagatzemats en la memòria.

Exercici 2. Paràmetres de tipus cadena de caràcters

Una cadena no és més que un vector on cada component emmagatzema un caràcter. Si s'empra la codificació ASCII, aleshores cada caràcter ocuparà un byte. Atès que una cadena d'uns pocs caràcters superaria fàcilment el llarg de paraula del processador, els paràmetres del

tipus cadena (i els vectors en general) es passen per referència a les funcions dels programes i també a les funcions del sistema (vegeu per exemple `print_string`).

En aquest exercici anem a treballar amb cadenes de caràcters. Partim del programa següent emmagatzemat en el fitxer "09_exer_02.s", el funcionament del qual haureu d'esbrinar.

```
          .globl __start
          .data 0x10000000
demana:   .asciiz "Escriviu-me alguna cosa: "
cadena:   .space 80

          .text 0x00400000
__start:  la $a0, demana
          la $a1, cadena
          li $a2, 80
          jal InputS
          li $v0,10
          syscall

InputS:   li $v0, 4
          syscall
          li $v0, 8
          move $a0, $a1
          move $a1, $a2
          syscall
          jr $ra
```

En particular, estudieu la funció anomenada `InputS` i digueu què fa el programa complet. Noteu que el perfil de la funció és `void InputS(char *$a0, char *$a1, int $a2)`. Compileu el programa i executeu-lo.

- On està la cadena que heu teclejat? Busqueu-la en la finestra *data segment* del simulador.

Ara heu de completar el programa anterior per tal que imprimisca la cadena que heu introduït pel teclat, reproduint el comportament reflectit tot seguit:

```
main() {
        char[] t1 = "Escriviu-me alguna cosa: "
        char[] t2 = "Heu escrit: "
        char[80] cadena;

        InputS(&t1, &cadena, 80);
        OutputS(&t2, &cadena);
        exit;
}
```

```
Escriviu-me alguna cosa: Estic retallat del tot
Heu escrit: Estic retallat del tot
```

Paseudocodi del programa demanat. Havent cridat InputS, el programa crida OutputS per imprimir la cadena llegida. En la part de baix, en teniu un exemple d'execució, on la part teclejada per l'usuari està en cursiva i la part escrita pel programa en negreta

Amb aquest fi implementareu la funció **`void OutputS(char *$a0, char *$a1)`**, que imprimeix seguides en la consola les dues cadenes a què apunten **`$a0`** i **`$a1`**.

Exercici 3. Recorregut de cadenes de caràcters

Com a complement de l'exercici anterior anem a escriure una nova funció que calcule la longitud d'una cadena de caràcters que se li passarà per referència. La declaració de la funció és **`int StrLength(char *c)`** i retorna el nombre de caràcters de la cadena. Suposarem que la cadena acaba amb el caràcter NUL (valor zero del codi ASCII). D'altra banda, teniu en compte que, mentre no s'òmpliga completament el buffer, la crida al sistema **`read_string`** introdueix el caràcter LF (*line feed,* valor 10 del codi ASCII) abans del caràcter NUL.

Després d'implementar-la podeu fer-la servir amb el programa de l'exercici anterior per tal de calcular la longitud de la cadena introduïda pel teclat i mostrar la longitud en la consola. Per exemple, un possible diàleg del programa tindrà ara aquest aspecte:

```
Escriviu-me alguna cosa: Estic retallat del tot
Heu escrit: Estic retallat del tot
La longitud és: 24
```

Qüestions diverses

1. Digueu en quina o quines instruccions de màquina es podria traduir la pseudoinstrucció **`lw $t0,var`** si l'adreça de la variable **`var`** (o siga, el valor de l'etiqueta **`var`**) és:
   - **`0x1000`**
   - **`0x100000`**
   - **`0x101000`**

2. Suposeu que l'adreça de la variable **`A`** és **`0x10000000`**. Compareu aquests dos fragments de codi equivalents:

```
lw $t0, A
addi $t0, $t0, 1
sw $t0, A
```

```
la $t0, A
lw $t1, 0($t0)
addi $t1, $t1, 1
sw $t1, 0($t0)
```

Quin dels dos codis màquina resultants és el més curt?

3. Considereu el fragment de codi següent:

```
alpha:    .asciiz "á"
          lb $t0, alpha
```

   Quin valor tindrà el registre `$t0` després de la seua execució? Quin valor hauria contingut si en comptes de `lb` s'hagués emprat `lbu`? Quina de les dues instruccions és més correcta per a emprar en aquest cas?

4. Feu aquesta prova amb el simulador: afegiu la instrucció `addi $ra,$ra,-4` al final del cos de la funció `InputS`, justament abans de la instrucció `jr $ra`, i feu que un programa la cride. Què passa? Expliqueu-ne el comportament.

Exercicis addicionals amb el simulador

Exercici 4: Més recorregut de cadenes

Escriviu el codi per a la funció `char StrChar(char *c, int n)`, que retorna el caràcter *n*-èssim de la cadena `*c`. Per no complicar massa el codi, suposeu que *n* mai no serà major que el llarg de la cadena.

Exercici 5: Vectors d'enters

Volem dissenyar un programa que calcule la suma de dos vectors d'enters A i B i deixe el resultat en un vector C. La dimensió dels vectors i els seus valors seran introduïts per teclat. Un exemple de diàleg del programa per a vectors de dimensió 4 és el següent:

```
D=4
A[0]=100
…
A[3]=130
B[0]=200
…
B[3]=230
C[0]=300
…
C[3]=360
```

El programa principal farà servir les funcions que referim tot seguit. Noteu que no partiu des de zero, ja que podeu inspirar-vos en algunes de les funcions ja dissenyades en programes anteriors.

- `void InputVector(char L, int D, word *V)`
- `void OutputVector(char L, int D, word *V)`
- `void AddVector(word dim, word *V1,word *V2,word *V3)`

Si haguereu d'escriure una nova versió d'aquestes tres mateixes funcions per a vectors de halfword (paraules de 16 bits) o de tipus byte (paraules de 8 bits), què caldria canviar en el perfil? I en el cos?

Apèndix

Funcions del sistema

Nom	$v0	Descripció	Arguments	Resultat
print_int	1	Imprimeix el valor d'un enter	`$a0` = enter a imprimir	—
print_float	2	Imprimeix el valor d'un *float*	`$a0` = *float* a imprimir	—
print_double	3	Imprimeix el valor d'un *double*	`$a0` = *double* a imprimir	—
print_string	4	Imprimeix una cadena de caracters	`$a0` = adreça on comença la cadena	—
read_int	5	Llig el valor d'un enter	—	`$v0` = enter llegit
read_float	6	Llig el valor d'un *float*	—	`$f0` = *float* llegit
read_double	7	Llig el valor d'un *double*	—	`$f0` = *double* llegit
read_string	8	Llig una cadena de caracters	`$a0` = adreça on comença la cadena	—
exit	10	Acaba el procés	—	—
print_char	11	Imprimeix un caràcter	`$a0` = caràcter a imprimir	—

Codi ascii

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	'	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	¦
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂