Repaso Examen Parcial

Donada la següent sentència escrita en alt nivell en C:

```
if (((a & 0xFFFF) != 0) && ((a ^ 0xAAAA) > 0))
    a = 5;
else
    a = 1;
```

Completa el següent fragment de codi en MIPS, que tradueix l'anterior sentència, escrivint en cada calaix un mnemònic d'instrucció o macro, etiqueta, registre o immediat. La variable `a` és de tipus `int` i està inicialitzada i guardada al registre `$t0`.

```
        [          ] $t1, $t0, 0xFFFF
        [          ] $t1, $zero, [          ]
etq1:   [          ] $t1, $t0, 0xAAAA
etq2:   [          ] $t1, $zero, [          ]

etq3:   li      $t0, 5

etq4:   b       et6

etq5:   li      $t0, 1

etq6:
```

Donada la següent funció `foo` en llenguatge C:

```
void foo(short M[][64], unsigned int k) {
        int i;
        short aux = M[k][8];
        for (i=5; i<45; i+=5){
                M[45-i][i] = aux;
        }
}
```

Completa el següent codi MIPS omplint les caselles en blanc perquè sigui equivalent a l'anterior codi en alt nivell, tenint en compte que els elements de la matriu M s'accedeixen utilitzant la tècnica **d'accés seqüencial** sempre que es pot, usant el registre `$t1` com a punter. Aquest punter `$t1` s'inicialitza amb l'adreça de l'element `M[40][5]`. Al codi se li ha aplicat l'optimització de conversió d'un bucle `for` en un `do_while` i l'eliminació de la variable d'inducció.

```
        sll     $t0, $a1, [          ]
        addu    $t0, $t0, $a0
        lh      $t2, [          ]($t0)      # aux = M[k][8];
        addiu   $t1, $a0, [          ]      # @M[40][5]
        addiu   $t3, $a0, [          ]      # adreça final del punter $t1
        b       cond
bucle:  sh      $t2, [          ]($t1)
        addiu   $t1, $t1, [          ]
cond:   bgtu    $t1, $t3, bucle
        jr      $ra
```

Donat el següent programa en C:

```
int v[100];
main() {
int i, pos=0;
      for (i=0; i<100; i++) {
        if (v[i] >= 0)
          pos += v[i];
      }
}
```

Hem traduït i optimitzat el codi en assemblador MIPS de la següent manera:

```
main:     move    $t5, $zero          # pos=0
          la      $t0, v              # macro que s'expandeix en 2 instruccions
          li      $t1, 100
          move    $t2, $zero          # i=0
for:
          sll     $t3, $t2, 2
          addu    $t3, $t0, $t3
          lw      $t4, 0($t3)         # $t4 = v[i]
          blt     $t4, $zero, fiif    # macro que s'expandeix en 2 instruccions
          addu    $t5, $t5, $t4       # pos += v[i]
fiif:
          addiu   $t2, $t2, 1         # i++
          blt     $t2, $t1, for       # macro que s'expandeix en 2 instruccions
fifor:
          jr      $ra                 # aquest salt sempre salta!
```

La taula següent mostra els CPI de cada tipus d'instruccions en un computador MIPS, quan executa l'anterior programa en assemblador:

Tipus	salts que salten	salts que no salten	load/store	les altres
CPI	3	2	10	1

Sabent que el vector `v` conté 50 elements positius, completa la següent taula indicant, per a cada tipus d'instrucció, el nombre total d'instruccions executades i el nombre total de cicles de rellotge corresponents. Atenció a les macros (en negreta), ja que l'expansió d'algunes d'elles pot comportar instruccions addicionals. Calcula també el temps d'execució total del programa, expressat en cicles i també en nanosegons, tenint en compte que la freqüència de rellotge és de 2GHz.

Tipus	salts que salten	salts que no salten	load/store	les altres
Instruccions				
Cicles				

Total cicles		Total temps (ns)	

¿Verdadero o falso?

1. Si faig un producte d'enters amb operadors i resultat de 32 bits per mitjà de la instrucció mult, puc assegurar que si $hi és diferent de zero s'ha produït overflow.

Donades les següents declaracions en C:

```
int vec[100];
void f(short *p) {
        *(p+5) = *p + 3;
}

int *g(int i) {
        return &vec[i];
}
```

a) Tradueix a MIPS la funció f

b) Tradueix a MIPS la funció g

Donades les següents declaracions de funcions en C:

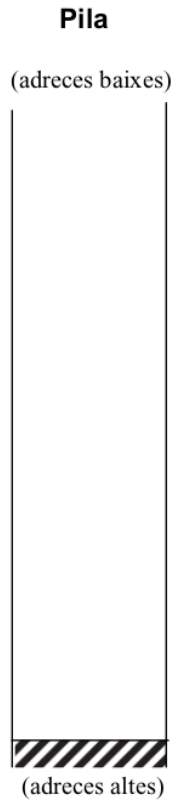
```
int f1(char *a, int *b);
int f2(short *a) {
        char cv[9];
        short sv[14];
        int s,t;
        s = 0;
        while(*a != sv[s])
                s = s + f1(cv, &t);
        return s;
}
```

Contesta els següents apartats que hi fan referència:

a) Quins elements de `f2` (paràmetres, variables locals i càlculs intermedis) s'han de guardar en registres de tipus segur `$s`? Especifica el registre segur concret que tries en aquells casos que en cal un. (Omple tantes entrades de la taula següent com calgui).

element de `f2 (en C)`	registre `$s`

b) Dibuixa el bloc d'activació de `f2`, especificant-hi la posició on apunta el registre `$sp` un cop reservat l'espai corresponent a la pila, així com el nom de cada registre i/o variable, i la seva posició (desplaçament relatiu al `$sp`).



c) Tradueix a llenguatge assemblador de MIPS la següent sentència del cos de la subrutina `f2`:

```
s = s + f1(cv, &t);
```

Donada la següent declaració de variables globals d'un programa escrit en llenguatge C:

```
char a[5]    = "DADA";
char *b      = &a[2];
long long c = -4;
char d[2]    = {14,16};
unsigned int e[100];
```

a) Tradueix-la al llenguatge assemblador del MIPS

```
.data
```

b) Completa la següent taula amb el contingut de les 48 posicions de memòria representades, en hexadecimal (sense el prefix "0x"). Les variables globals s'emmagatzemen a partir de l'adreça 0x10010000. Recorda que el codi ASCII de la 'A' és el 0x41, i que les variables globals no inicialitzades valen 0x00. Les posicions de memòria no ocupades es deixen en blanc.

c) Donat el següent codi en MIPS, indica quin és el valor final en hexadecimal del registre `$t1`:

```
la      $t1, d+1
lb      $t1, 0($t1)
lui     $t2, 0x1001
or      $t1, $t2, $t1
lhu     $t1, 0($t1)
```