2º Grado Informática
Estructura de Computadores
2 Febrero 2015

Examen Test de Teoría (3.0p)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
b	b	d	c	a	b	a	d	c	c	c	a	c	b	c	b	a	c	b	a	b	c	a	c	c	d	b	b	b	d

Examen Test de Prácticas (4.0p)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	d	a	c	b	c	d	a	d	a	a	c	b	b	b	c	b	b	d	c

Examen de Problemas (3.0p)

1. Ensamblador (0.75 puntos).

Otras soluciones son posibles, siempre que produzcan el resultado correcto y no añadan complejidad innecesaria.
Se muestra una solución sin optimizar (a la izquierda) y otra optimizada (a la derecha).

```
max:
    push   %ebp
    mov    %esp, %ebp   # Ajuste marco
    mov    8(%ebp), %eax       # EAX=a
    mov    12(%ebp), %edx      # EDX=b
    cmp    %edx, %eax   # a:b?
    jg     .finmax      # a>b  - nada
    mov    %edx, %eax   # a<=b - return b
.finmax:
    pop    %ebp         # Destruir marco
    ret

maxV:
    push   %ebp
    mov    %esp, %ebp   # Ajuste marco
    push   %esi         # Usar todos los
    push   %edi         # salva-invocado
    push   %ebx         # porque hay un call

    mov    8(%ebp), %esi       # ESI = v1
    mov    12(%ebp), %edi      # EDI = v2
    cmp    $0, 20(%ebp) # N:0?
    jle     .finmaxV    # N<=0 - acabar
    mov    $0, %ebx     # for i=0 , EBX=i
.bucle:
    push  (%edi,%ebx,4) # arg2º v2[i] PUSH M/M
    push  (%esi,%ebx,4) # arg1º v1[i] PUSH M/M


    call   max
    add    $8, %esp     # Recuperar pila max
    mov    16(%ebp), %edx      # EDX = v3
    mov    %eax, (%edx,%ebx,4)# v3[i]=max
    inc    %ebx                # i++
    cmp    20(%ebp), %ebx      # i:N?
    jne     .bucle             # i!=N - repetir
.finmaxV:

    pop    %ebx         # Salva-invocados
    pop    %edi
    pop    %esi
    pop    %ebp         # Destruir marco
    ret
```

```
max:
    push   %ebp
    mov    %esp, %ebp   # Ajuste marco
    mov    8(%ebp), %eax       # EAX=a
    mov    12(%ebp), %edx      # EDX=b
    cmp    %edx, %eax   # a:b?
    cmovle %edx, %eax   # a<=b - return b

    pop    %ebp         # Destruir marco
    ret

maxV:
    push   %ebp
    mov    %esp, %ebp   # Ajuste marco
    push   %esi         # Usar todos los
    push   %edi         # salva-invocado
    push   %ebx         # porque hay un call
    sub    $8, %esp     # pila para max(a,b)
    mov    8(%ebp), %esi       # ESI = v1
    mov    12(%ebp), %edi      # EDI = v2
    cmp    $0, 20(%ebp) # N:0?
    jle     .finmaxV    # N<=0 - acabar
    mov    $0, %ebx     # EBX=i mejor que v3
.bucle:
    mov    (%edi,%ebx,4), %eax# v2[i]
    mov    %eax, 4(%esp)       # arg 2º
    mov    (%esi,%ebx,4), %eax# v1[i]
    mov    %eax, (%esp)        # arg 1º
    call    max

    mov    16(%ebp), %edx      # EDX = v3
    mov    %eax, (%edx,%ebx,4)# v3[i]=max
    inc    %ebx                # i++
    cmp    20(%ebp), %ebx      # i:N?
    jne     .bucle             # i!=N - repet
.finmaxV:
    add    $8, %esp     # Recuperar pila max
    pop    %ebx         # Salva-invocados
    pop    %edi
    pop    %esi
    pop    %ebp         # Destruir marco
    ret
```

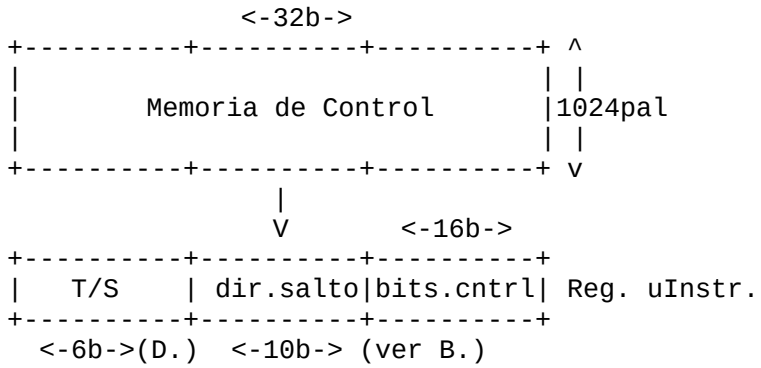
2. **Ensamblador** (0.7 puntos).

Otras soluciones son posibles, siempre que produzcan el resultado correcto y no añadan complejidad innecesaria.
Se muestra una solución sin optimizar (a la izquierda) y otra optimizada (a la derecha).
El enunciado no requería un programa completo, se ofrece sencillamente para poder comprobar su corrección.

```
.data
  array: .int 10, 20, 30, 40, 6, -5, 9, 8, 10, 35, 45, 56, 7
longarr: .int (.-array)/4
 result: .int 0

.text
_start: .global _start
    mov   $0, %esi      # ESI = indice 0..N-1
    mov   longarr, %edi # EDI = tamaño N
    mov   $0, %eax      # EAX = acumulador = 0
bucle:                  # EDX =array[i=0..N-1]
    mov   array(,%esi,4), %edx
    cmp   $30, %edx     # array[i]:30?

    jge   saltar        # >= ? no sumarlo
    add   %edx, %eax
saltar:

    inc   %esi          # i++
    cmp   %esi, %edi    # 0..N-1
    jnz   bucle         # acaba cuando i==N

    mov   %eax, result

    mov  $1, %eax       # llamada EXIT(0)
    mov  $0, %ebx
    int  $0x80
```

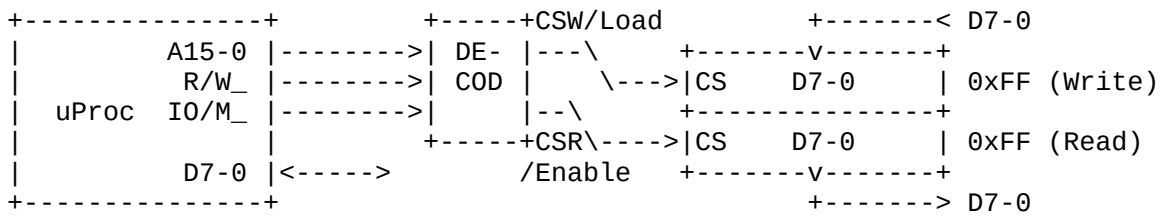
```
.data
  array: .int 10, 20, 30, 40, 6, -5, 9, 8, 10, 35, 45, 56, 7
longarr: .int (.-array)/4
 result: .int 0

.text
_start: .global _start

    mov   longarr, %edi # EDI = indice N..1
    mov   $0, %eax      # EAX = acumulador = 0
bucle:                  # EDX = array[i=N..1]
    mov   array-4(,%edi,4), %edx
    cmp   $30, %edx     # array[i]:30?

    mov   $0, %ecx      # ECX = sumando condic
    cmovl %edx, %ecx    # < ? sumarlo
    add   %ecx, %eax    # si no, suma ECX=0

    dec   %edi          # i--
                        # N..1
    jnz   bucle         # acaba cuando i==0

    mov   %eax, result

    mov  $1, %eax       # llamada EXIT(0)
    mov  $0, %ebx
    int  $0x80
```

3. **Unidad de control** (0.25 puntos).

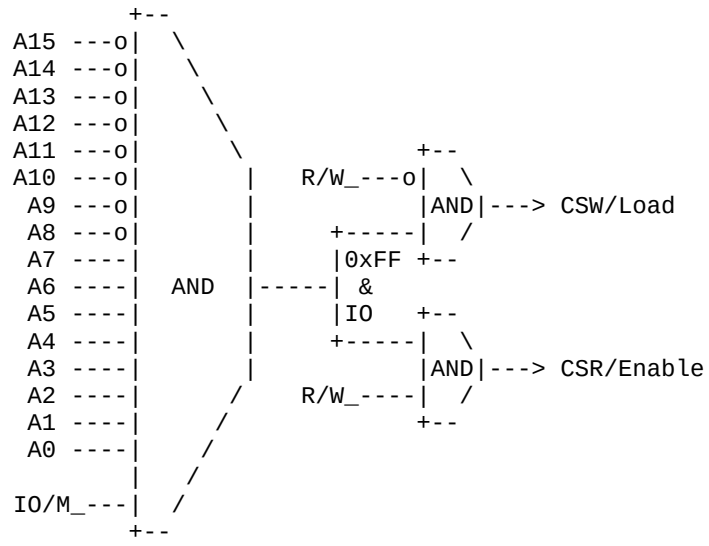


A.	32b	Registro uInstr.
B.	10b	Campo Dirección.Siguiente (indicar una de entre 1024pal=2^10)
C.	10b	Contador uProgr. (indicar una de entre 1024)
D.	6b	Campo T/S (32b-10b-16b)
E.	10líneas	Buses multiplexor uPC (pasar 10b dir.salto o uPC+1 requiere 10líneas)

4. **Entrada/Salida** (0.5 puntos).



Detalle del módulo decodificador. Notar que las entradas son A15-0, R/W_ e IO/M_, las salidas CSW y CSR.



Código de write_block()

```
void write_block(char *buffer){
     int i;

     for (i=0; i<100; i++){
           while (in(0xFF)>=0) {};   // while ( ! (in(0xFF) & 0x80) ) {};
           out (0xFF, buffer[i]);
     }
}
```

5. **Diseño del sistema de memoria** (0.4 puntos).

SRAM1:

A0-13 => 14b dir => 2^14 = 16Kpal
D0-7 => pal=Byte 16KB = **16K x 8**

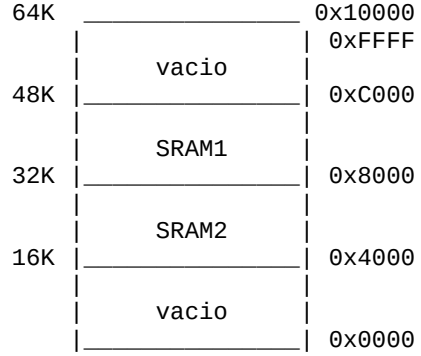
CS => A15-A14=**10** => dirs CPU desde **10**00 0000 0000 0000 hex **0x8000** 2^15=32K
hasta **10**11 1111 1111 1111 hex **0xBFFF** 2^15+2^14-1=48K-1

SRAM2:

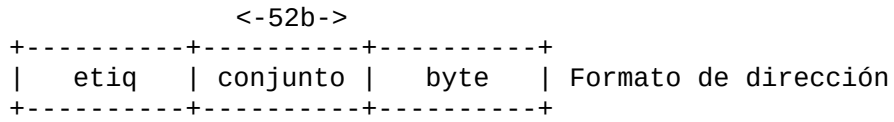
A0-13 => 14b dir => 2^14 = 16Kpal
D0-7 => pal=Byte 16KB = **16K x 8**

CS => A15-A14=**01** => dirs CPU desde **01**00 0000 0000 0000 hex **0x4000** 2^14 =16K
hasta **01**11 1111 1111 1111 hex **0x7FFF** 2^15-1=32K-1

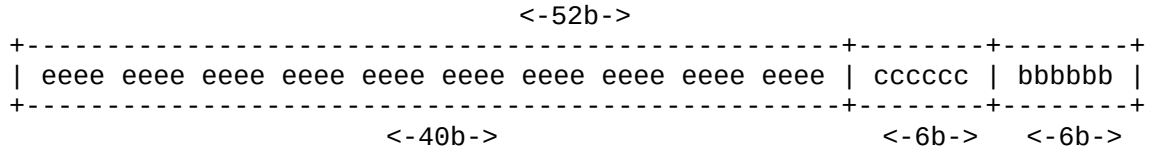
El enunciado no requería un dibujo del mapa de memoria, se ofrece sencillamente como aclaración.



6. **Memoria cache** (0.4 puntos).



L1 = 32KB => 2^15 B
líneas de 64B => **2^6 B/lin** => 2^15 B / 2^6 B/lin = 2^9 líneas en cache — 6b campo “byte”
8 vías => 8 = 2^3 lin/cjto => 2^9 lin / 2^3 lin/cjto = **2^6 conjuntos** en cache — 6b campo “conjunto”
52-6-6 = **40b** — 40b campo “etiqueta”



Dada una dirección de 52b indicada por la CPU:

- se selecciona el conjunto “cccccc” indicado por los 6b campo “conjunto”
- hay 8 vías, es decir, el conjunto tiene 8 líneas y cada línea su correspondiente etiqueta (y bit válido, etc)
- se comparan las etiquetas válidas (las 8 si los 8bits válido=1) con la etiqueta “eeee…eeee” indicada por la CPU
- si alguna línea coincide, se trata de un acierto de cache (hit), y la dirección física solicitada está en esa línea
- en concreto, está en el desplazamiento “bbbbbb” de la línea con etiqueta coincidente en el conjunto “cccccc”

El enunciado no requería un dibujo del conjunto, se ofrece sencillamente como aclaración.