Universidad de Granada — ETSIIT — ATC

**2º Grado Informática**
**Estructura de Computadores**
**19 Septiembre 2012**

# Examen Test (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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | d | b | c | a | b | b | b | d | a | c | c | c | a | b | b | b | a | c | a | d | c | a | b | d | a | b | b | a | d |

# Examen de Prácticas (4.0p)

**1. Funcionamiento de la Pila.** (1 punto). (en gris respuestas adicionales opcionales)

A. CALL guarda en pila (push) la dirección de retorno (%eip) (y salta %eip <- función invocada)
B. LEAVE (deshace el marco de pila (mov %ebp, %esp) y…) recupera de pila el marco anterior (pop %ebp)
C. RET recupera de pila (pop) la dirección de retorno (%eip anteriormente salvado)
D. Args se meten en pila en orden inverso (…, push arg3, push arg2, push arg1)
E. Valor retorno se devuelve en el registro %eax (y %edx, si 64-bit)
F.

| marco pila invocante |
|---|
| 3er arg 0x4 |
| 2º arg 0xbabecafe |
| 1er arg 0xbeefbabe |
| dirección retorno |
| antiguo EBP |
| marco pila de printf |

Pila crece hacia abajo ↓ (posiciones inferiores)

**2. Funciones aritméticas sencillas.** (1 punto).

A. **twice** –ret está mal, antes de leave (se retorna a dirección basura %ebp antiguo, leave nunca llega a ejecutarse).
B. **second** – no hay error.
C. **constant** – 0x2 debería ser $0x2 (inmediato). 0x2 (direccionamiento directo) intenta accede a la posición de memoria 0x2 (que seguramente causará segmentation fault).

**3. Programación mixta C-asm.** (1 punto).

A. while ( r = A % B ) { A = B; B = r; }
B. resultado= **3**
C. Descripción: calcula el máximo común divisor (mcd) de A y B (algoritmo de Euclides).
- En cada iteración se calcula el resto de la división A/B en r (r=A%B). En la siguiente iteración se repite el cálculo, usando B y r como nuevos valores de A y B (porque mcd(A,B)=mcd(B,A%B)). Eventualmente r=0 porque A será múltiplo de B, y ese B es el resultado (mcd).

**4. Segmentación de cauce.** (1 punto).

A. Bloqueos:

| #ciclo | instrucción | retardo | tipo de bloqueo-motivo |
|---|---|---|---|
| 8,17,26 | bnez r2,loop | 1 | RAW de R2 con instrucción anterior: sub r2,r1,poli-8 |
| 15,24 | addd f2,f4,f2 | 1 | RAW de D4 con instrucción -3 antes: multd f4,f2,f0<br>hay otra dependencia con instr. anterior: ld f2,(r1), pero no causa bloqueo |
| (otros bloqueos no son relevantes para ahorrar ciclos con la reordenación que se pide) | | | |

**B.** Resultado: Se guarda en la variable poli.
Poli= 25 (3x4 + 4x2 + 5)

**C.** Calcula coefs[0] x^2 + coefs[1] x + coefs[2]. Si coefs contiene los coeficientes de un polinomio desde Cn-1 a C0, este programa evalúa el polinomio en x. Lo hace sacando factores comunes x todo lo posible: ax^2+bx+c = ((ax)+b)x+c.

**D.** Como los bloqueos están entre sub-bnez y entre multd-addd (también hay dependencia ld-addd, ver 2 flechas verdes), se puede adelantar sub entre ld y addd (así se rellena el ciclo del bloqueo), pero entonces hay que repetirla antes de "j test" (o antes de ld f2, pero siempre después de add r1, que es donde se fija el valor de r1) para que se siga calculando bien la condición de salto en la primera iteración

Original

```
start:
        ld      f0,x
        add     r1,r0,coefs
        ld      f2,(r1)

        j       test
loop:
        multd   f4,f2,f0
        add     r1,r1,8
        ld      f2,(r1)

        addd    f2,f4,f2
test:
        sub     r2,r1,poli-8
        bnez    r2,loop

        sd      poli,f2
        trap    0
```

Modificado

```
start:
        ld      f0,x
        add     r1,r0,coefs
        ld      f2,(r1)
        sub     r2,r1,poli-8
        j       test
loop:
        multd   f4,f2,f0
        add     r1,r1,8
        ld      f2,(r1)
        sub     r2,r1,poli-8
        addd    f2,f4,f2
test:

        bnez    r2,loop

        sd      poli,f2
        trap    0
```

# Examen de Problemas (3.0p)

**1. Acceso a arrays** (0.5 puntos).

```
H = 7
J = 5
Posibles comentarios que podrían añadirse al código para demostrar su comprensión
copy_array:
        movslq  %esi,%rsi               # índice y
        movslq  %edi,%rdi               # índice x
        movq    %rsi, %rdx              # rdx = y
        salq    $3, %rdx                #       8y
        subq    %rsi, %rdx              #             - y
        addq    %rdi, %rdx              #             + x -> rdx = 7y+x
        leaq    (%rdi,%rdi,4), %rax     # rax = 5x
        addq    %rsi, %rax              #             + y -> rax = 5x+y
        movl    array1(,%rax,4), %eax # &array1[x][y] = array1+ 5x+y -> J = 5
        movl    %eax, array2(,%rdx,4) # &array2[y][x] = array2+ 7y+x -> H = 7
        ret
```

**2. Representación y acceso a estructuras** (0.5 puntos).

```
type1_t:  [unsigned] short (tamaño 2B)
type2_t:  char
CNT:      14
Posibles comentarios que podrían añadirse al código para demostrar su comprensión
p1:
        pushl   %ebp
        movl    %esp, %ebp
```

```
movl    12(%ebp), %eax          # eax = ap
movsbw  28(%eax),%cx            # cx = ap->x, pasar de 1B con signo a 2B (para y[i]=x)
                                #      => type2_t = 1B con signo = char
                                #      => y[CNT] ocupa 28B, saltarlo para llegar a x
movl    8(%ebp), %edx           # edx = i, notar cómo siguiente instr. indexa *2
movw    %cx, (%eax,%edx,2)      # ap->y[i]=x => elementos y[] son de tamaño 2B
popl    %ebp                    #      => type1_t = cualquiera de 2B
ret                             #      => 28/2 = 14 = CNT
```

**3.** **Memoria cache** (0.5 puntos).
Bloques de 64 palabras = 2^6 pal/blq
-> 6 bits para direccionar la palabra dentro de cada bloque

MP 32K = 2^5 2^10 = 2^15 palabras
-> 15 bits para direccionar en memoria
-> 2^15 pal / 2^6 pal/blq = 2^9 bloques en MP (512 bloques)

Cache 4K = 2^2 2^10 = 2^12 palabras de cache
-> 2^12 pal / 2^6 pal/blq = 2^6 marcos en cache (64 marcos)

Totalmente Asociativa:
Cualquier bloque puede ir a cualquier marco, se identifica bloque por etiqueta

| 15 bits | |
|---|---|
| 15-6 = 9 bits | 6 bits |
| etiqueta | palabra |

Correspondencia directa:
Bloques sucesivos van a marcos sucesivos. Al acabarse la cache, vuelta a empezar
-> 6 bits inferiores palabra, 6 bits siguientes marco, resto etiqueta

| 15 bits | | |
|---|---|---|
| 15-6-6=3 | 6 bits | 6 bits |
| etiqueta | marco (bloque) | palabra |

Asociativa 16 vías:
Bloques sucesivos van a conjuntos sucesivos, pero cada conj.tiene varios marcos (vías)
16 = 2^4 blq/conj, cache de 2^6 marcos
-> 2^6 marcos / 2^4 blq/conj = 2^2 conjuntos -> 2 bits para direccionar conjunto
-> 6 bits inferiores palabra, 2 bits siguientes conjunto, resto etiqueta

| 15 bits | | |
|---|---|---|
| 15-2-6= 7 bits | 2bits | 6 bits |
| etiqueta | conj. | palabra |

**4.** **Diseño del sistema de memoria** (0.5 puntos).
Ver imágenes adjuntas

**5.** **Entrada/Salida** (0.5 puntos).
Ver imágenes adjuntas

**6.** **Unidad de control microprogramada** (0.5 puntos).
Ver imágenes adjuntas