Relación de problemas - Estructura Computadores

Rubén Calvo Villazón → 2ºB

3.1)

Operando	Valor	Comentario
%eax	0x100	Registro
0x104	0xAB	Dir. Absoluta
$0x108	0x108	Inmediato
(%eax)	0xFF	Dir. 0x100
4(%eax)	0xAB	Dir. 0x104
9(%eax, %edx)	0x11	Dir. 0x10C
260(%ecx, %edx)	0x13	Dir. 0x108
0xFC(, %ecx, 4)	0xFF	Dir. 0x100
(%eax, %edx, 4)	0x11	Dir. 0x10C

3.2) El código generado por el GCC incluye sufijos en las instrucciones mientras que el de ensamblador no. El código en sufijos es el siguiente:

```
movl %eax, (%esp)
movb (%ecx), %dx
movb $0xFF, %bl
movb (%esp, %edx, 4), %dh
pushl $0xFF
movw %dx, (%eax)
popl %edi
```

3.3) Código con las explicaciones de los diferentes errores:

movb $0xF, (%bl) No se puede usar %bl como reg. de direcciones

movl %ecx, (%esp) Desajuste entre el registro de identificación y el sufijo de instrucciones

movw (%eax), 4(%esp) No se puede tener como destino y origen las referencias a memoria

movb %ah, %sh → No hay registro %sh

movl %eax, %dx Operando de destino con tamaño incorrecto

movb %cl, 8(%ebp) Desajuste entre el registro de identificación y el sufijo de instrucciones

3.4)

src_t	dest_t	Instrucción
int	int	movl %eax, (%edx)
char	int	movsbl %cl, (%edx)
char	unsigned	movsbl %cl, (%edx)
unsigned char	int	movzbl %cl, (%edx)
int	char	movb %cl, (%edx)
unsigned	unsigned char	movb %al, (%edx)
unsigned	int	movl %eax, (%edx)

3.5) Iniciamos una simulación con los valores x, y, z en las localizaciones xp, yp, zp. Obtenemos el siguiente comportamiento

xp en 8(%ebp), yp en 12(%ebp), zp en 16(%ebp)

```
movl 8(%ebp), %edi ; obtener xp
movl 12(%ebp), %edx ; obtener yp
movl 16(%ebp), %ecx ; obtener zp
movl (%edx), %ebx ; y
movl (%ecx), %esi ; z
movl (%edi), %ecx ; obtener x
movl %ecx, (%edx) ; x -> yp
movl %ebx, (%ecx) ; y -> zp
movl %esi, (%edi) ; z -> xp
```

Se puede generar el equivalente C:

```
void decode (int *xp, int *yp, int *zp){
    int tx = *xp;
    int ty = *yp;
    int tz = *zp;
    *yp = tx;
    *zp = ty;
    *xp = tz;
}
```

3.6) Versatilidad de la orden leal

Inst.	Res:
leal 6(%eax), %edx	6+x
leal (%eax, %ecx), %edx	x+y
leal (%ecx, %ecx, 4), %edx	x+4y
leal 7(%ecx, %eax, 8), %edx	7+9x
leal 0xA(, %ecx, 4), %edx	10+4y
leal 9(%eax, %ecx, 2), %edx	9+x+2y

3.7) Inst. Aritméticas:

Inst.	Dest.	Val.
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull $10, (%eax, %edx, 4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD

3.8) En ensamblador:

```
movl 8(%ebp), %eax    ; x
sall $2, %eax         ; x<<2
movl 12(%ebp), %ecx   ; n
sarl %cl, %eax        ; x>>n
```

3.9) Cada una de las expresiones se implementa por una única instrucción

```
int1 t1 = x^y;
     t2 = t1>>t3;
     t3 = ~t2;
     t4 = t3-t;
```

3.10) 1°) La inst. xor sirve para indicar que %edx esté = 0

2°) Una forma de poner %edx = 0 es con movl $0, %edx

3°) El montaje y desmontaje del código, sin embargo encontramos que la versión con xorl requiere 2b y la de movl 5b

3.11) Se reemplaza la inst. cltd con una de los registros %edx a 0 y se usa con divl en vez de idivl, produciendo:

```
movl 8(%ebp), %eax   ; %eax = x
movl $0, %edx        ; conjunto de bits de orden sup.
divl 12(%ebp)        ; dividido por y
movl %eax, 4(%esp)   ; x/y
movl %edx, (%esp)    ; x % y
```

3.12) 1°) Podemos ver que el programa está realizando operaciones de multiplicación de datos en 64b. También se puede ver que la operación de multiplicación de 64b se usa extrayendo su signo y se llega a la conclusión de que mult es long long unsigned.

2) Sea x el valor de la variable x y y el valor de y. Se puede escribir como $y = y_h \cdot 2^{32} + y_l$ Donde y_h y y_l son valores representados por el orden alto y bajo de 32 bits respectivamente. La representación completa del producto sería 96 bits de longitud, pero se requiere solo el orden bajo de 32 bits con x, y, y t el total de 64 bits producto de $x \cdot y_l$ que se puede dividir en el orden alto y bajo de partes de t_h y t_l.
Finalmente el resultado de t_l en el orden bajo y $s+t_h$ es el orden superior

El código es:

```
movl 12(%ebp), %eax     ; x
movl 20(%ebp), %ecx     ; y_h
imull %eax, %ecx        ; s = x * y_h
leal (%ecx, %edx), %edx ; s = t_h
movl 8(%ebp), %ecx      ; destino
movl %eax, (%ecx)       ; t_l
movl %edx, 4(%ecx)      ; s + t_h
```

3.13) Es importante entender el código de montaje y no perder de vista el tipo de valor del programa. En su lugar, las diferentes instr. determinan el tamaño del operando y se llevan una forma.
Para hacer el mapa de instr. de secuencias hay que volver a C y deducir los datos de los diferentes valores del programa

1°) El sufijo 'l' y el registro identificador indica operandos de 32 bits mientras que la comparación es un complemento a 2. Podemos inferir que data_t debe ser int

2°) 'w' y los identificadores indican operandos 16 bits, mientras que la comparación es un complemento a 2 >= 1. Podemos inferir que data_t debe ser short

3º) El sufijo 'b' y el registro identificador indica operandos de 8 bits, mientras que la comparación es un compl a 2 'c' Podemos inferir que dato_t es char

4º) El sufijo 'l' y el registro identificador indica operandos de 32 bits mientras que la comparación es con '!=' que es lo mismo que si los argumentos son son signed, unsigned o punteros. Podemos inferir que dato_t podría ser int, unsigned o puntero. Pero el primero de los dos casos, se puede descartar ~~unsigned short~~. long

3.14) El problema es similar al anterior, la diferencia es que en este se trata las instrucciones de test:

1º) El sufijo 'l' y ~~los~~ identificadores de registro indican operandos de 32 b. mientras que la comparación es con !=, que es lo mismo para signed y unsigned. Podemos inferir que dato_t debe ser int, unsigned o otro tipo de puntero. Pero los del primero de los dos casos se puede tener un descartar de tipo long

2º) El sufijo 'w' y los identificadores de registro indican operandos de 16 b. mientras que la comparación es un complemento a 2 'g'. Podemos inferir que dato_t debe ser char

4º) El sufijo 'w' y los identificadores de registro indican 16 b. mientras que la comparación es unsigned >, Podemos inferir que dato_t debe ser unsigned short

3.15)

1º) La inst. je tiene como objetivo 0x80482917 0x05 el código desensamblado es 0x80482c96:

80482f5: 74 05 je 80482c96

80482f7: e8 1e 00 0000 call 804825d

2º) La inst. jb tiene como objetivo 0x8048359 el código desensamblado orig. es: 0x8048340:

8048357: 72 e7 jb 8048340

8048359: c6 05 10 c0 04 08 01 movb 8048254

3º) De acuerdo con la anterior producida por el desensamblador el objetivo de salto está en 0x8048391 dire. absoluta De acuerdo con la codificación del byte esto debe estar en 0x12 más allá de mov.

Si se resta todo esto nos da 0x804837f

804837d: 74 12 je 8048391

804837f: b8 00 00 00 00 mov $0x0, %eax

4º) Si se leen los bytes en orden inverso se puede ver un desfase en 0xffffffe0 o 32 (dec). La dirección de este es 0x80482c4 de

0x80482c4:

80482bf: e9 e0 ff ff ff jmp 80482c4

80482c4: 90 nop

5º) Un salto indirecto denota por código de instrucción a ff 25. La dirección desde donde se da el salto se ha de leer esta codificada por 4b. Desde que la máquina es little endian, orden inverso:

es fc ff 04 08

3.16) Código de control sobre el flujo de control

```
void goto_cond(int c, int *p){
    if(p==0)
        goto done;
    if(c<=0)
        goto done;
    *p += c
done:
    return;
}
```

- La primera rama condicional es parte de la aplicación si la prueba a p no nula falla, el código salta a c>0

3.17) 1º La conversión alternativa implica cambiar algunas líneas de código:

```
int gotodiff-dt(int x, int y){
    int result;
    if(x<y)
        goto twe;
    result = x-y;
    goto done;
twe:
    result = y-x;
done:
    return result;
}
```

2º) En la mayoría de casos la elección es arbitraria. La regla original funciona mejor para el caso común donde no hay ninguna declaración más. En este caso basta con modificar la regla de conversión.

```
E = test_expr;
if(!E)
    goto done;

then_statement;

done:
```

3.18) Este problema requiere trabajar a través de una estructura de ramas anidada, donde se aplica la regla de traducción de sentencias if. En su mayor parte el código máquina es una traducción directa de C, la única diferencia es la exp. de inicialización que es mandada hacia abajo de manera que solo se calcula cuando este será el valor devuelto

```
int test(int x, int y) {
  int val = x^y;
  if( x < -3) {
      if( y < x)
          val = x * y;
      else
          val = x + y;
  } else if( x > 2)
      val = x - y;

  return val
}
```

3.22)
```
int foo (unsigned x) {
    int val = 0;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0x1;
}
```

El código computa la paridad en el argumento x, este devuelve 1 si es impar en x y 0 si es par

3.23)
```
int fun_b(unsigned x){
    int val = 0;
    int i;
    for(i=0; i<32; i++) {
        val = (val<<1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

Este código invierte los bits de x creando un espejo. Para ello se cambian los bits de x de izquierda a derecha y luego se llenan, ya que val se desplaza de derecha a izquierda

3.24)

```
int sum = 0;
int i = 0;
while (i < 10) {
    if(i & 1)
        continue;
    sum += i;
    i++;
}
```

Es un bucle infinito actualizando la variable i

Reemplazando goto por el continue (goto update),

```
update:
    i++;
```

3.25)

Este problema refuerza el metodo de calculo de una prediccion errónea,

1° Podemos aplicar nuestra formula para obtener tmp = 2(31-10) = 30

2° Cuando se produce prediccion erronea, la funcion requiere alrededor de 16+30 = 46 ciclos

3.27)

```
int test(int x, int y){
    int val = 4*x;
    if(y > 0){
        if(x < y)
            val = x - y;
        else
            val = x ^ y;
    }else if(y < -2)
        val = x + y
    return val;
}
```

3.41

A. struct P1 { int i; char c; int j; char d; };

c	e	j	d	t	Al.
0	4	8	12	16	4

B. struct P2 { int i; char c; char d; int j; };

i	c	j	d	t	Al
0	4	5	8	12	4

C. struct P3 { short w[3]; char c[3] };

w	c	t	Al
0	6	10	2

D. struct P4 { short w[3]; char * c[3] };

w	c	T	Al
0	8	20	4

E. struct P3 { struct P1 c[2]; struct P2 *p };

c	p	T	Al
0	32	36	4

3.53

A. struct P1 { int i; char c; long j; char d; };

i	c	j	d	t	Al
0	4	8	16	24	8

B. struct P2 { long i; char c; char d; int j; }

i	c	d	j	t	Al
0	8	9	12	16	8

C. struct P3 { short w[3]; char c[3]; };

w	c	t	Al
0	6	10	2

D. struct P4 { short w[3]; char *c[3]; };

w	c	t	Al
0	8	32	8

E. struct P5 { struct P1 c[2]; struct P2 *p; };

c	p	t	Al
0	48	56	8