Lab kern2

Juan Pablo Capurro - 98194

29/9/2017

Contents

asm-inc

Explicar los parámetros de la instrucción `asm` mostrada.

La instrucción hará un incremento(por la instrucción `inc`) de la variable `x`, dándole la libertad al compilador de usar para ello cualquier registro de uso general o hacer acceso directo a memoria(restricción `g`).

Describir, usando para ello la salida de objdump -d, la correspondencia entre el código original y las instrucciones assembler generadas tras la compilación

La primer línea es parte del preámbulo de gcc para ejecutar las instrucciones pedidas, que carga el valor de `x` al registro `eax`.
La segunda es la instrucción explicitada, con los parámetros reemplazados, y la tercera es el `return x` de la línea siguiente. GCC prefiere el uso del registro `eax` para ahorrarse una instrucción extra para retornar el valor de `x`

```
00000000 <inc>:
   0:   8b 44 24 04             mov    0x4(%esp),%eax
   4:   40                      inc    %eax
   5:   c3                      ret
```

Mostrar, de nuevo usando gcc -c y objdump -d, en qué cambia el código generado para la siguiente versión, y razonar el por qué de las diferencias

El código generado para `int2` fuerza que el incremento se haga en memoria, y que luego el valor incrementado se mueva a la variable `ret`. GCC optimiza el retorno de dicho valor al compilar con `-O1`, no así con `-O0`, caso en el cual copia el resultado del incremento a la posición de memoria que ocupaba `ret`.
compilando con `-O1`

```
00000006 <inc2>:
   6:   ff 44 24 04             incl   0x4(%esp)
   a:   8b 44 24 04             mov    0x4(%esp),%eax
   e:   c3                      ret
```

compilando con `-O0`

```
0000000f <inc2>:
   f:   55                      push   %ebp
  10:   89 e5                   mov    %esp,%ebp
  12:   83 ec 10                sub    $0x10,%esp
  15:   ff 45 08                incl   0x8(%ebp)
  18:   8b 45 08                mov    0x8(%ebp),%eax
```

```
  1b:   89 45 fc                mov    %eax,-0x4(%ebp)
  1e:   8b 45 fc                mov    -0x4(%ebp),%eax
  21:   c9                      leave
  22:   c3                      ret
```

¿Sería válido especificar `"m"(x)` si `inc` solo funcionase con un registro como operando?

No, no sería válido ya que a GCC lesería imposible formar una instrucción correcta.

Recompilar ambas versiones cambiando incl por inc y, si alguna de las dos versiones no compila, explicar el porqué de la diferencia.

`inc1` compila haciendo el cambio pedido, `inc2` no, con el mensaje de error `Error: no instruction mnemonic suffix given and no register operands; can't size instruction`
`inc1`, al indicársele con `=g` que puede usar cualquier registro, puede saber que el operando en cuestión ocupa una palabra, mientras que `inc2` pide que la instrucción no use registros, por lo que a la hora de generar el código objeto, no hay forma de deducir de qué largo es el operando.

asm-ptr

Se proponen las siguientes tres implementaciones de inc_p(); indicar cuáles funcionan y cuáles no y, examinando el código generado por gcc, explicar por qué

```
0804846b <inc_p_0>:
 804846b:       55                      push   %ebp
 804846c:       89 e5                   mov    %esp,%ebp
 804846e:       ff 45 08                incl   0x8(%ebp)
 8048471:       90                      nop
 8048472:       5d                      pop    %ebp
 8048473:       c3                      ret

08048474 <inc_p_1>:
 8048474:       55                      push   %ebp
 8048475:       89 e5                   mov    %esp,%ebp
 8048477:       8b 45 08                mov    0x8(%ebp),%eax
 804847a:       ff 00                   incl   (%eax)
 804847c:       90                      nop
 804847d:       5d                      pop    %ebp
 804847e:       c3                      ret

0804847f <inc_p_2>:
 804847f:       55                      push   %ebp
 8048480:       89 e5                   mov    %esp,%ebp
 8048482:       8b 45 08                mov    0x8(%ebp),%eax
 8048485:       ff 00                   incl   (%eax)
 8048487:       90                      nop
 8048488:       5d                      pop    %ebp
 8048489:       c3                      ret
```

Sólo funcionan las implementaciones 1 y 2, ya que la 0 no modifica el valor apuntado por `p` sino el valor del puntero `p`, desplazándolo un byte hacia adelante.

Opciones gcc	::“m”(*p*) / :“=*m*”(p)	
`-O2`	n=0	n = -8013875
`-O2 -fno-inline`	n=1	n=1

Al permitirle a GCC hacer inline automático de las funciones, la variante *1* se rompe porque no considera a `*p` un operando

de salida, y por ende no se guarda en `n` el resultado del incremento. La variante *2* se rompe porque no considera a `*p` un operando de entrada, por lo que no lee el valor de `n` de memoria antes de incrementarlo. El código correcto sería `asm("incl %0" : "=m"(*p) : "m"(*p));`. Es posible hacerlo con matching constraints (`asm("incl %0" : "=m"(*p) : "0"(*p));`), pero gcc da warnings de `matching constraint does not allow a register`.

asm-regs

Completar los constraints en la función my_write, de manera que la llamada al sistema funcione.

```
asm("int $0x80" : : "a"(SYS_write), "b"(1), "c"(msg), "d"(count));
```

¿En qué cambia el código generado (incluyendo el código de main) si se declara my_write con atributo fastcall?

con *fastcall*:

```
$ gcc -o my_write -O2 -fno-inline -m32 my_write.c && ./my_write
Hello, world!

080482e0 <main>:
 80482e0:       ba 0e 00 00 00          mov    $0xe,%edx
 80482e5:       b9 80 84 04 08          mov    $0x8048480,%ecx
 80482ea:       e8 01 01 00 00          call   80483f0 <my_write>
 80482ef:       31 c0                   xor    %eax,%eax
 80482f1:       c3                      ret

080483f0 <my_write>:
 80483f0:       53                      push   %ebx
 80483f1:       b8 04 00 00 00          mov    $0x4,%eax
 80483f6:       bb 01 00 00 00          mov    $0x1,%ebx
 80483fb:       cd 80                   int    $0x80
 80483fd:       5b                      pop    %ebx
 80483fe:       c3                      ret
 80483ff:       90                      nop
```

sin *fastcall*:

```
$ gcc -o my_write -O2 -fno-inline -m32 my_write.c && ./my_write
Hello, world!

080483f0 <my_write>:
 80483f0:       53                      push   %ebx
 80483f1:       b8 04 00 00 00          mov    $0x4,%eax
 80483f6:       bb 01 00 00 00          mov    $0x1,%ebx
 80483fb:       8b 54 24 0c             mov    0xc(%esp),%edx
 80483ff:       8b 4c 24 08             mov    0x8(%esp),%ecx
 8048403:       cd 80                   int    $0x80
 8048405:       5b                      pop    %ebx
 8048406:       c3                      ret
 8048407:       66 90                   xchg   %ax,%ax
 8048409:       66 90                   xchg   %ax,%ax
 804840b:       66 90                   xchg   %ax,%ax
 804840d:       66 90                   xchg   %ax,%ax
 804840f:       90                      nop

080482e0 <main>:
 80482e0:       6a 0e                   push   $0xe
 80482e2:       68 90 84 04 08          push   $0x8048490
 80482e7:       e8 04 01 00 00          call   80483f0 <my_write>
 80482ec:       58                      pop    %eax
```

```
80482ed:       31 c0                   xor    %eax,%eax
80482ef:       5a                      pop    %edx
80482f0:       c3                      ret
```

Con niveles mayores de optimización y *fastcall*, el pasaje de parámetros se hace con registros cuando es posible.

asm-volatile

Indicar qué ocurre al ejecutar el programa con las siguientes modificaciones, explicando por qué funciona, o no:

Eliminando el modificador volatile de la instrucción assembler.

El programa imprime `Hello, write2!` como se espera.

Con volatile de vuelta en su lugar, eliminando el if (!result) ... de la función main.

El programa imprime `Hello, write2! \n write2 falló`, como se espera, ya que *volatile* le indica al compilador que las instrucciones dentro del *asm* tienen *side effects*, por lo que deben ser ejecutadas independientemente de si sus parámetros de salida son usados o no.

Eliminando tanto volatile como la comprobación del booleano. ¿Qué código genera gcc para main en este caso? ¿Cambió el código generado para my_write2?

El programa solo imprime `write2 falló`, porque el compilador no ejecuta el contenido de *asm* si considera que sus parámetros de salida no son relevantes (es decir, no son usados).

¿Por qué no se hizo necesario *volatile* en el ejercicio asm-regs, ni tampoco en asm-inc?

En `asm-inc`, no es necesario *volatile* ya que los parámetros de salida de asm son usados (se retornan de la función y luego se imprimen por pantalla más adelante en el código, es decir, son parámetro de entrada de otra operación).
En `asm-regs`, no es necesario *volatile* ya que si un statement asm no tiene operandos de salida es implícitamente *volatile*.