Facultad de Informática
Universidad Complutense de Madrid

Fundamentos de Computadores II - Práctica 4
C y ensamblador

Daniel Báscones (danibasc@ucm.es)

27 de marzo de 2023

1. Objetivos

En esta práctica profundizaremos en nuestros conocimientos sobre la programación en en ensamblador sobre RISC-V creando proyectos con varios ficheros fuente, algunos de ellos escritos en lenguaje C y otros escritos en lenguaje ensamblador. Los objetivos concretos son:

- Comprender la diferencia entre variables locales y variables globales, en su almacenamiento a bajo nivel.
- Comprender el significado de símbolos estáticos (variables y funciones).
- Analizar los problemas que surgen cuando queremos utilizar varios ficheros fuente y comprender cómo se realiza la resolución de símbolos.
- Comprender la relación entre el código C que escribimos y el código máquina que se ejecuta.
- Saber utilizar desde un programa escrito en C variables y rutinas definidas en ensamblador, y viceversa.
- Comprender el código generado por el compilador gcc.
- Conocer la representación de los tipos estructurados propios de los lenguajes de alto nivel.
- En esta práctica el alumno tendrá que crear un programa, escribiendo unas partes en C y otras partes en ensamblador, de forma que las rutinas escritas en C puedan invocar rutinas escritas en ensamblador y viceversa.

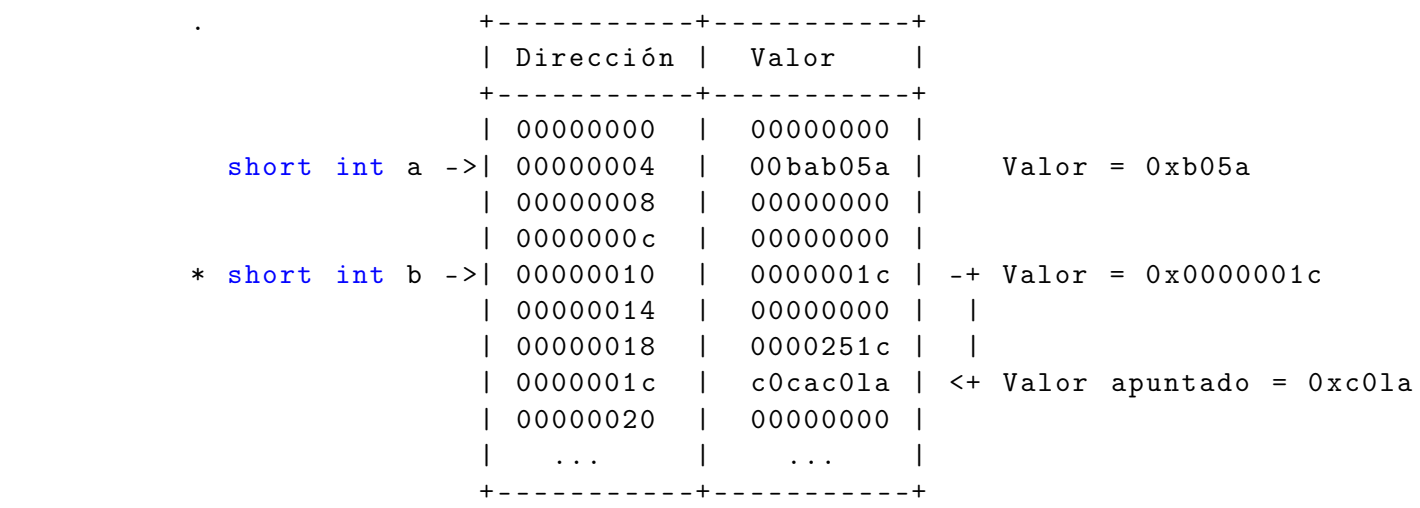
2. Tratando con diferentes tipos de datos

Hasta ahora, los accesos a memoria realizados han sido mediante las instrucciones `lw` y `sw`, que operan con 4 bytes a la vez. Es importante saber que existen instrucciones para manejar diferentes tamaños de dato, con y sin signo. Estas son:

- `lb`: Carga un byte (el menos significativo), extendiendo el signo a anchura de 32 bits.
- `lbu`: Como la anterior, pero completando con ceros a anchura de 32 bits.
- `lh`: Carga dos bytes (los menos significativos), extendiendo el signo.
- `lhu`: Como la anterior, completando con ceros.
- `sb`: Guarda un único byte (el menos significativo).
- `sh`: Guarda dos bytes (los dos menos significativos).

Estas instrucciones se utilizan para tratar, a nivel de ensamblador, con los tamaños que tienen los diferentes tipos de variables de `C`:

- `bool` / `char`: Se representa con un byte con signo, utiliza las instrucciones `lb`, `sb`.
- `short int`: Un entero corto de dos bytes, utiliza `lh`, `sh`.
- `int`: Un entero normal de cuatro bytes, que utiliza `lw`, `sw`.
- `unsigned char` / `unsigned short int`: Tipos sin signo que utilizan, respectivamente, las instrucciones `lbu`, `lhu`. Nótese que para los *stores*, utilizan la versión normal `sb`, `sh`.
- `* <type>`: Los punteros son un caso especial. Representan direcciones de memoria de 32 bits, y su verdadero valor se encuentra al acceder a dicha dirección de memoria. Al ser de 32 bits, utilizan `lw` y `sw` para su manejo independientemente del tipo al que apuntan. Ej: `* int`, `* char` y `* short int` son todos de 32 bits.



3. Pasando de C a ensamblador

Un código escrito en un lenguaje de alto nivel como C no puede ejecutarse directamente. Antes de ello, debe pasar por un compilador, a fin de generar las instrucciones en ensamblador que sí entiende el procesador. El compilador cuenta con varias herramientas:

- **Preprocesador:** Realiza una primera iteración sobre el código C, eliminando comentarios y resolviendo directivas de preprocesador (como `#define`) precedidas por `#`.
- **Compilador:** El compilador propiamente dicho. Procesa el código C y genera un código equivalente en lenguaje ensamblador (RISC-V en nuestro caso)
- **Ensamblador:** Partiendo del código ensamblador, genera un fichero objeto, que ya contiene las instrucciones en código máquina (binario). Las direcciones de memoria de funciones y símbolos aún no están asignadas, por lo que este fichero no es ejecutable.
- **Enlazador:** Utilizando uno o varios ficheros objeto, además de un `ld_script`, el enlazador toma las instrucciones y símbolos y genera un fichero ejecutable, donde los diferentes símbolos (variables, funciones, etc) tienen una dirección de memoria asignada. Este fichero ya puede cargarse en la memoria indicada y ejecutarse.

NOTA: Para utilizar todas estas herramientas, debemos abrir una terminal (Escribiendo `cmd` en el menú de inicio de windows) y luego colocarla en el directorio de las herramientas de eclipse con `cd <ruta a EclipseRV>\SysGCC\bin`. Desde ahí podremos ejecutar las herramientas sobre nuestro código (el cual debemos copiar a la misma carpeta).

Veamos un ejemplo sencillo. Empezamos con el siguiente código:

Código C (`testc.c`)

```
#define OP_A 20
#define OP_B 30
int a = OP_A;
int b = OP_B;

int main(void) {
    int c = a + b;
    return c;
}
```

Linker script (`ld_script.ld`)

```
SECTIONS
{
    . = 0x0;
    .text : { *(.text) }
    . = 0x10000;
    .data : { *(.data) }
    .bss  : { *(.bss)  }
    _stack = 0x20000;
}
```

Podemos ejecutar el preprocesador con la sentencia:

```
> riscv64-unknown-elf-cpp testc.c -o out_prep.c
```

Y el compilador, ejecutando:

```
> riscv64-unknown-elf-gcc out_prep.c -S -o out_asm.s
```

Preprocesado (`out_prep.c`)

```
# 1 "testc.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "testc.c"

int a = 20;
int b = 30;

int main(void) {
    int c = a + b;
    return c;
}
```

Código ensamblado (`out_asm.s`)

```
.   .file   "out_prep.c"
    .option nopic
    .attribute arch,
    "rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"
    .attribute unaligned_access, 0
    .attribute stack_align, 16
    .text
    .globl  a
    .section    .sdata,"aw"
    .align  2
    .type   a, @object
    .size   a, 4
a:
    .word   20
    .globl  b
    .align  2
    .type   b, @object
    .size   b, 4
b:
    .word   30
    .text
    .align  1
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-32
    sw      s0,24(sp)
    addi    s0,sp,32
    lui     a5,%hi(a)
    lw      a4,%lo(a)(a5)
    lui     a5,%hi(b)
    lw      a5,%lo(b)(a5)
    add     a5,a4,a5
    sw      a5,-20(s0)
    lw      a5,-20(s0)
    mv      a0,a5
    lw      s0,24(sp)
    addi    sp,sp,32
    jr      ra
    .size   main, .-main
    .ident  "GCC:␣(GNU)␣10.1.0"
```

Observamos que, el código preprocesado, ha visto resueltas las directivas de preprocesador `#define`, y se le han añadido algunos detalles de procedencia en forma de comentarios. El código ya ensamblado indica diferentes atributos del programa y de sus símbolos, además de contener el programa RISC-V equivalente al código C escrito. Algunas de las construcciones que utiliza pueden resultarnos desconocidas, y es que este código ensamblado no utiliza pseudoinstrucciones. Por ejemplo, tenemos las siguientes equivalencias:

Ensamblador	Equivalencia	Ensamblador	Equivalencia
`lui a5, %hi(a)` `lw a4, %lo(a)(a5)`	`la a5, a` `lw a4, 0(a5)`	`jr ra`	`ret`

Nos falta únicamente generar el código máquina desde este fichero y después enlazarlo junto con el `ld_script.ld` para conseguir el ejecutable. Lo hacemos con:

```
> riscv64-unknown-elf-gcc out_asm.s -c -o out_comp.o
```

```
> riscv64-unknown-elf-ld out_comp.o -T ld_script.ld -o out_linked.elf
```

Este fichero tiene un formato que no podemos leer en texto plano, por lo que necesitamos una herramienta que nos ayude a ver el contenido:

```
> riscv64-unknown-elf-objdump -xD out_linked.elf
```

```
out_linked.elf:     file format elf64-littleriscv
architecture: riscv:rv64
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File
off  Algn
  0 .text         00000024  0000000000000000  0000000000000000
  00001000  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .sdata        00000008  0000000000010000  0000000000010000
  00002000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000010000 l    d  .sdata 0000000000000000 .sdata
0000000000000000 l    df *ABS*  0000000000000000 out_prep.c
0000000000010004 g     O .sdata 0000000000000004 b
0000000000000000 g     F .text  0000000000000024 main
0000000000010000 g     O .sdata 0000000000000004 a

Disassembly of section .text:
0000000000000000 <main>:
```

```
   0:   1101                    addi    sp,sp,-32
   2:   ec22                    sd      s0,24(sp)
   4:   1000                    addi    s0,sp,32
   6:   67c1                    lui     a5,0x10
   8:   0007a703                lw      a4,0(a5) # 10000 <a>
   c:   67c1                    lui     a5,0x10
   e:   0047a783                lw      a5,4(a5) # 10004 <b>
  12:   9fb9                    addw    a5,a5,a4
  14:   fef42623                sw      a5,-20(s0)
  18:   fec42783                lw      a5,-20(s0)
  1c:   853e                    mv      a0,a5
  1e:   6462                    ld      s0,24(sp)
  20:   6105                    addi    sp,sp,32
  22:   8082                    ret

Disassembly of section .sdata:
0000000000010000 <a>:
   10000:       0014                    0x14
0000000000010004 <b>:
   10004:       001e                    c.slli  zero,0x7
```

La salida del comando ha sido filtrada por razones de visualización, pero es importante saber que aparece bastante más información de la aquí presentada. Si nos fijamos, el código continúa siendo el mismo, si bien ahora ya vemos las instrucciones correspondientes en binario a la izquierda. Además, tenemos el código ubicado en la dirección `0`, así como los datos ubicados en la dirección `10000`, gracias a la labor del `ld_script`. El programa está listo para ejecutarse.

4. Símbolos globales

Cuando utilizamos varios ficheros de código fuente (ya sea en C o en ensamblador RISC-V), lo normal es que unos ficheros utilicen funciones y variables de otros. Aunque los ficheros se enlazan todos juntos para generar el ejecutable final, la compilación se realiza individualmente (para no tener que recompilar todo ante un cambio en únicamente un archivo). En esta etapa, los ficheros desconocen la existencia de funciones o variables externas, a menos que se lo indiquemos explícitamente. Aunque no es obligatorio, sí es recomendable en ambos lenguajes, importando o exportando **símbolos globales**.

```
//en ensamblador:
//.extern es recomendable aunque no obligatorio
.extern fun_1 //fun_1 está en otro archivo
//.global es obligatorio si queremos usar la función en otro archivo
.global fun_2 //fun_2 puede ser utilizada por otro archivo
.extern var_1 //var_1 está en otro archivo
.global var_2 //var_2 puede ser utilizada por otro archivo
```

```
//en C:
extern void fun_2( void ); //función definida en otro archivo
                           //extern recomendable, no obligatorio
void fun_1( void );        //por defecto las funciones son globales
                           //fun_1 puede ser vista por otro archivo
extern int var_2;          //variable definida en otro archivo
int var_1;                 //por defecto las variables son globales
                           //var_1 puede ser vista por otro archivo
static void fun_3( void ); //función forzada a ser local
static int var_3;          //variable forzada a ser local
```

Nótese que tanto en C como en ensamblador RISC-V, la sintaxis para definir variables o funciones globales es la misma. Esto es porque, a nivel de compilador, **ambas son símbolos**, y como tales se tratan igual. Un símbolo es un nombre con una dirección de memoria asociada. En caso de una variable, dicha dirección contendrá el valor de la variable. En caso de una función, dicha dirección será la de la primera instrucción de la función.

Al compilar un programa, los símbolos que aún no se conocen (por ser externos), se quedan a la espera de que el enlazador los encuentre en otro fichero. Si no fuera así, la etapa de enlazado generará un error. Todos los símbolos deben estar resueltos para generar el ejecutable final. Veamos un ejemplo:

Archivo `main.c`

```
extern int suma(int a, int b);

int main(void) {
    return suma(5, 7);
}
```

Archivo `suma.s`

```
.text
.global suma
suma:
    add a0, a0, a1
    ret
```

Si compilamos `main.c` y observamos sus contenidos, vemos lo siguiente:

```
> riscv64-unknown-elf-gcc main.c -c -o obj_main.o
> riscv64-unknown-elf-objdump -x obj_main.o
//contenido del archivo obj_main.o
obj_main.o:     file format elf64-littleriscv
//[...]
SYMBOL TABLE:
//[...]
0000000000000000          *UND* 0000000000000000 suma
```

Es decir, el símbolo (en este caso función) `suma` no está definido (`UNDefined`). Haciendo lo mismo con el fichero `suma.s`, obtenemos:

```
> riscv64-unknown-elf-gcc suma.s -c -o obj_suma.o
```

```
> riscv64-unknown-elf-objdump -xD obj_suma.o

//contenido del archivo obj_suma.o
obj_suma.o:     file format elf64-littleriscv
//[...]
SYMBOL TABLE:
//[...]
0000000000000000          .text 0000000000000000 suma
//[...]
Dissasembly of section .text:
0000000000000000 <suma>:
    0: 952e                 add     a0,a0,a1
    2: 8082                 ret
//[...]
```

Vemos que en este caso, el símbolo para la función `suma` sí está definido. Ahora solo queda juntar los dos con el enlazador, que utilizará todos los símbolos existentes para terminar de generar el programa:

```
> riscv64-unknown-elf-ld out_main.o out_suma.o -T ld_script.ld -o
out_linked.elf

> riscv64-unknown-elf-objdump -xD out_linked.o
```

Tras lo cual obtenemos el ejecutable final, donde ambos archivos han sido unidos, y el símbolo para `suma` está ubicado en el ejecutable, y contiene el código pertinente:

```
//contenido del archivo out_linked.elf
SYMBOL TABLE:
//[...]
0000000000000000 g     F .text  000000000000001c main
000000000000001c g       .text  0000000000000000 suma
//[...]
Disassembly of section .text:
0000000000000000 <main>:
   0:   1141                    addi    sp,sp,-16
   2:   e406                    sd      ra,8(sp)
   4:   e022                    sd      s0,0(sp)
   6:   0800                    addi    s0,sp,16
   8:   459d                    li      a1,7
   a:   4515                    li      a0,5
   c:   010000ef                jal     ra,1c <suma>
  10:   87aa                    mv      a5,a0
  12:   853e                    mv      a0,a5
  14:   60a2                    ld      ra,8(sp)
  16:   6402                    ld      s0,0(sp)
  18:   0141                    addi    sp,sp,16
  1a:   8082                    ret
000000000000001c <suma>:
  1c:   952e                    add     a0,a0,a1
  1e:   8082                    ret
```

5. Desarrollo de la práctica

En esta práctica vamos a experimentar mezclando código de alto nivel C, y ensamblador RISC-V. Este tipo de programación se da habitualmente en la práctica cuando programamos a bajo nivel: Interrupciones hardware, drivers, microcontroladores... Hay ciertas funcionalidades específicas de la plataforma con que trabajamos que no tienen representación en lenguajes de alto nivel.

En esta práctica, vamos a implementar el siguiente programa, capaz de calcular qué vector está más lejos del origen.

```
#define N 5

//nuestros dos vectores de N componentes
int U[N] = {5, 2, -3, 7, 6};
int V[N] = {6, -1, 1, 0, 3};
//variable que almacenará el resultado: 1 si U es mayor, 0 si es V
char mayor_u;

/**
 * Función que guarda un valor en el puntero proporcionado
 */
void guardar(char valor, char * ubicación) {
    *ubicación = valor;
}

/**
 * Función simple que multiplica dos números iterativamente
 * con signo
 */
int mul(int a, int b) {
    int res = 0, sign = 0;
    if (a < 0) {
        sign = 1;
        a = -a;
    }
    while (a--) res += b;
    if (sign)
        return -res;
    else
        return res;
}

/**
 * Función simple que obtiene la raíz cuadrada iterativamente
 */
int i_sqrt(int a) {
    int root = 0;
```

```
    while (mul(root, root) < a) {
        root++;
    }
    return root;
}

/**
 * Calculamos distancia euclídea. Sumamos todos los cuadrados
 * y terminamos sacando la raíz cuadrada (entera)
 */
int eucl_dist(int w [], int size) {
    int acc = 0;
    for (int i = 0; i < size; i++) {
        acc += mul(w[i], w[i]);
    }
    return i_sqrt(acc);
}

/**
 * Punto de entrada al programa
 */
void main() {
    //calculamos la distancia euclídea al origen
    int d_u = eucl_dist(U, N);
    int d_v = eucl_dist(V, N);
    //vemos si U estaba más lejos
    char mayor = d_u > d_v;
    guardar(mayor, &mayor_u);

    while(1); //quedo en un bucle infinito
}
```

1. Probar la práctica en su estado actual. Crear un fichero `main.c` donde se pegue el código anterior, compilar y ejecutar en eclipse y ver que da el resultado esperado (U es mayor que V).

2. Para probar el código mixto y llamadas de C a ensamblador RISC-V y viceversa, traducir las funciones `eucl_dist` y `guardar` a ensamblador. Para ello, crear un archivo `fun_asm.asm` con la siguiente forma:

```
//rellenar con directivas .extern y .global
//con las funciones apropiadas

eucl_dist:
    //recibo dirección de W en a0, y tamaño N en a1
    //realizo los cálculos pertinentes
    //devuelvo el resultado en a0
```

```
guardar:
    //recibo el valor en a0, y la dirección destino en a1
    //asegurarse que sólo se guarda UN BYTE!!
```

Es de **extremada importancia** respetar el estándar y convenio de llamadas, ya que la parte del programa en C estará esperando que todo siga el estándar de modo estricto.

No hay que olvidarse de declarar en `main.c` las funciones `eucl_dist` y `guardar` como `extern` (y borrar las declaraciones originales), ya que ahora estarán en el fichero `fun_asm.asm`. Tampoco debemos olvidarnos de declarar en el archivo `fun_asm.asm` las funciones `mul` y `i_sqrt` como `.extern`, ya que vienen del archivo `main.c`, además de exportar las funciones `eucl_dist` y `guardar` mediante `.global`.