Ejercicio Práctico 0

Trabajar con un programa ensamblador.

Trabajar con las herramientas para desarrollar un programa en lenguaje ensamblador.

1) Revisar el punto 3 del documento Entorno de trabajo que se ha publicado en el Tablón.

2) Editar el archivo div.asm que os damos utilizando el editor *geany*. Para abrir el archivo podéis hacer lo siguiente:
   - Abrir el editor utilizando menú, *Aplicaciones → Programación*
   - Seleccionar el archivo div.asm, pulsar el botón derecho i seleccionar la opción Abrir con Geany.

3) A continuación, haced el ensamblaje del código fuente, el enlazado del código objeto y la generación del código ejecutable utilizando los comandos explicados en los puntos 3.2 y 3.3 del documento Entorno de trabajo desde el terminal del mismo editor.

4) Con el comando (ls div*) podréis ver que tenéis los archivos siguientes: div.asm, div.o, div.

5) El archivo div (sin extensión) es el archivo que podréis ejecutar directamente como se explica en el punto 3.4 del documento Entorno de trabajo.

   En este caso, ejecutando directamente el programa no visualiza nada por pantalla, y por lo tanto, no veremos nada y no podréis comprobar que hace, ni validar si lo hace correctamente.

6) Para ver que hace y como lo hace tenéis que utilizar el depurador que nos permite ejecutar el programa instrucción a instrucción visualizando en cada paso las variables y registros que queramos.

   Abrid el programa con el depurador y haced algunas pruebas para comprobar que funciona correctamente. Consultad el punto 3.5 del documento Entorno de trabajo mientras seguís los pasos siguientes:

   a) Definid un punto de interrupción en la línea 25.

   b) Haced una primera ejecución paso a paso apuntando el número de línea de las instrucciones que se ejecutan.

25	27	31	33	35	36	31	33	35	36	31	33	35	36	38	40	45	47	49

   c) Añadid nuevos puntos de interrupción en las líneas 27, 35 y 45. Añadid los registros rax y rcx y las variables dividendo, divisor, cociente y residuo en la ventana de 'Expresiones Vigiladas'. Ejecutad el programa con la opción Ejecutar o con la tecla F5 y anotad los valores de los registros y variables cada vez que se detenga la ejecución.

Línea	rax	rcx	dividendo	divisor	cociente	residuo
25	- - - - -	- - - - - -	14	4	0	0
27	14	- - - - - -	14	4	0	0
35	10	1	14	4	0	0

e) Ver el contenido de la memoria: dividendo, divisor, cociente y residuo.
Iniciad nuevamente la ejecución del programa utilizando la opción *Restart* del menú *Execution* quedando la ejecución detenida en la línea 25. Si ha finalizado la ejecución del código, seleccionar nuevamente la opción *'Ejecutar' (Run)*.

Seleccionad la pestaña *Memoria* (parte inferior izquierda de la pantalla), escribid en el cuadro de texto: &dividendo (nombre de la primera variable declarada en la sección de datos) y pulsad *Enter*.
Haced clic con el botón derecho encima la ventana *Dirección* y seleccionad primero la opción *Bytes* y después la opción *Unsigned decimal*.
Ejecutad el programa hasta el siguiente punto de interrupción (línea 45). Anotad la dirección y el valor de las variables que aparecen en la ventana *Dirección*.

	Dirección	Valor de los 8 bytes en decimal (unsigned decimal)							
dividendo	0x601018	14	0	0	0	0	0	0	0
divisor	0x601020	4	0	0	0	0	0	0	0
cociente	0x601028	3	0	0	0	0	0	0	0
residuo	0x601030	2	0	0	0	0	0	0	0

Haced otras pruebas con el depurador hasta que os sintáis seguros con el entorno, de esta forma podréis continuar trabajando con más comodidad. En los materiales hay explicadas otras funcionalidades del depurador, revisadlas y probadlas. Tener cierta agilidad en la utilización del depurador os ayudará mucho a detectar errores en los programas que vayáis desarrollando.

Trabajar con un programa en C

Trabajar con las herramientas para desarrollar un programa en lenguaje C.

7) Revisamos el punto 4 del documento Entorno de trabajo.

8) Editad el archivo divc.c utilizando el editor *geany p*ara abrir el archivo podéis hacer lo siguiente:
   - Abrir el editor utilizando menú, *Aplicaciones → Programación*
   - Seleccionar el archivo divc.c, pulsar el botón derecho del ratón i seleccionar la opción Abrir con Geany.

9) A continuación, haced la compilación del código fuente y la generación del código ejecutable utilizando los comandos explicados en el punto 4.4 del documento Entorno de trabajo desde el terminal del mismo editor (no indicaremos ningún código objeto ensamblador, por lo tanto, sólo hay que especificar el archivo .c).

```
$ gcc –o divc –g divc.c
```

10) Con la orden (ls divc*) podréis ver que tenéis los archivos siguientes: divc.c, divc.

11) El archivo divc (sin extensión) es el archivo que podréis ejecutar directamente como se explica en el punto 4.5 del documento Entorno de trabajo.

a) Definid un punto de interrupción a la línea 18.

b) Haced una primera ejecución paso a paso apuntando el número de línea de las instrucciones que se ejecutan.

18	20	24	26	28	24	26	28	24	26	28	30

c) Añadid nuevos puntos de interrupción en las líneas 20, 28 y 30. Añadid las variables dividendo, divisor, cociente y residuo a la ventana de ‘‘Expresiones vigiladas’ (watches)’. Ejecutad el programa con la opción ‘Ejecutar’ (Run) o con la tecla F5 y anotad los valores de las variables cada vez que se detenga la ejecución.

Línea	dividendo	divisor	cociente	residuo
18	14	4	0	0
20	14	4	0	0
28	14	4	1	10
28	14	4	2	6
28	14	4	3	2
30	14	4	3	2

Notar que los valores obtenidos en esta tabla son los mismos que los obtenidos en la tabla del apartado c) del punto 6).

d) Desplegad el código c para ver el código ensamblador que genera el compilador.
Haced clic en el signo + que hay a la izquierda del número de línea y observad las instrucciones ensamblador asociadas a cada instrucción c.

Indicad el código ensamblador asociado a cada una de las siguientes instrucciones:
Marcar en negrita las instrucciones equivalentes al código ensamblador anterior.

```
    residuo = residuo - divisor;
```

```
0x4004c6 mov    rdx,QWORD PTR [rip+0x200b73]        # 0x601040 <residuo>
0x4004cd mov    rax,QWORD PTR [rip+0x200b4c]        # 0x601020 <divisor>
0x4004d4 mov    rcx,rdx
0x4004d7 sub      rcx,rax      ;SUB RAX, [DIVISOR]
0x4004da mov    rax,rcx
0x4004dd mov    QWORD PTR [rip+0x200b5c],rax        # 0x601040 <residuo>
```

```
    cociente++;
```

```
0x4004e4 mov    rax,QWORD PTR [rip+0x200b4d]        # 0x601038 <cociente>
0x4004eb add      rax,0x1      ;INC RCX
0x4004ef mov    QWORD PTR [rip+0x200b42],rax        # 0x601038 <cociente>
```

```
    while (residuo >= divisor);
```

Trabajar con un programa en C i en ensamblador

Trabajar con las herramientas para desarrollar un programa en lenguaje C que llama a un programa en lenguaje ensamblador.

13) Revisar el punto 4 del documento Entorno de trabajo.

14) Editar el archivo div2.asm i div2c.c que os damos utilizando el editor *geany*.
Para abrir el archivo podéis hacer lo siguiente:
- Abrir el editor utilizando menú, *Aplicaciones → Programación*
- Seleccionar los archivo div2.asm i div2c.c, pulsar el botón derecho i seleccionar la opción Abrir con Geany.

15) A continuación, haced el ensamblaje del código fuente, el enlazado del código objeto y la generación del código ejecutable utilizando los comandos explicados en los puntos 4.2 y 4.4 del documento Entorno de trabajo desde el terminal del mismo editor.

16) Con el comando (ls div2*) podréis ver que tenéis los archivos siguientes: div2.asm, div2.o, div2c.c i div2.

17) El archivo div2 (sin extensión) es el archivo que podréis ejecutar directamente como se explica en el punto 4.5 del documento Entorno de trabajo.

En este caso, ejecutando directamente el programa no visualiza nada por pantalla, y por lo tanto, no veremos nada y no podréis comprobar que hace, ni validar si lo hace correctamente.

18) Para ver que hace y como lo hace tenéis que utilizar el depurador que nos permite ejecutar el programa instrucción a instrucción visualizando en cada paso las variables y registros que queramos.

Abrid el programa con el depurador y haced algunas pruebas para comprobar que funciona correctamente. Consultad el punto 3.5 i 4.6 del documento Entorno de trabajo mientras seguís los pasos siguientes:

a) Definid un punto de interrupción en la línea 20. (del archivo div2c.c, codi C).

a) Haced una primera ejecución paso a paso, utilizando la opción ‘‘Entrar en la función’ (Step into)’ o pulsando la tecla F8. Anotar el número de línea de las instrucciones que se ejecutan indicando en azul las instrucciones de código C y en verde las instrucciones de código ensamblador.

20	18	20	24	26	28	29	24	26	28	29	24	26	28	29	31	33	35	22

b) Añadid nuevos puntos de interrupción en las líneas en las líneas 24, 31 y 35. Añadid los registros rax y rcx y las variables dividendo, divisor, cociente y residuo en la ventana de ‘Expresiones Vigiladas’. Ejecutad el programa con la opción Ejecutar o con la tecla F5 y anotad los valores de los registros y variables cada vez que se detenga la ejecución.

Línea	rax	rcx	dividendo	divisor	cociente	residuo
20	- - - - -	- - - - - -	14	4	0	0
24	14	- - - - - -	14	4	0	0

Trabajar con diferentes tipos de datos

Trabajar con las herramientas para desarrollar un programa en lenguaje C que llama un programa en lenguaje ensamblador.

19) Revisamos el punto 4 del documento Entorno de trabajo.

20) Editad el archivo dataTypes.asm y dataTypesc.c que os damos utilizando el editor geany.
Para abrir el archivo podéis hacer lo siguiente:
- Abrir el editor utilizando menú, Aplicaciones → Programación
- Seleccionar los archivos dataTypes.asm y dataTypesc.c, pulsar el botón derecho y seleccionar la opción Abrir con Geany.

21) A continuación, haced la compilación del código fuente, el enlazado del código objeto y la generación del código ejecutable utilizando las órdenes explicadas en los puntos 4.2 y 4.4 del documento Entorno de trabajo desde el terminal del mismo editor.

22) Con el comando (ls dataTypes*) podréis ver que tenéis los archivos siguientes: dataTypes.asm dataTypes.o, dataTypesc.c, dataTypes.

23) El archivo dataTypes (sin extensión) es el archivo que podréis ejecutar directamente como se explica en el punto 4.5 del documento Entorno de trabajo.

En este caso, ejecutando directamente el programa, tal como pasaba anteriormente, no visualiza nada por pantalla y por lo tanto no veréis nada, no podréis comprobar que hace ni validar si lo hace correctamente.

24) Para ver que hace y como lo hace tenéis que utilizar el depurador, que nos permite ejecutar el programa instrucción a instrucción visualizando a cada paso las variables que queremos.

Abrid el programa con el depurador y haced algunas pruebas para comprobar que funciona correctamente. Consultad los puntos 3.5 y 4.6 del documento Entorno de trabajo mientras seguís los pasos siguientes:

a) Definid un punto de ruptura en la línea 22 (del archivo dataTypesc.c, código C).

b) Haced una primera ejecución con la opción ‘Ejecutar’ (Run) o pulsando la tecla F5. El programa se para en la línea 20 (del código C), utilizando la opción ‘Entrar en la función’ (Step into) o pulsando la tecla F8, pasamos al código ensamblador y el programa se para en la línea 30 (del código ensamblador).

c) Añadid las variables varChar, varShortInt, varInt, varLongInt y los registros al, ah, ax, eax y rax en la ventana ‘Expresiones vigiladas’ (watches). Mostrad los valores en formato hexadecimal utilizando el prefijo ‘/x’ ante las variables y de los registros. (/x varChar y /x $al). En el punto 5) del apartado 3.5 del documento Entorno de trabajo podéis consultar como hacerlo.

d) Ver el contenido de la memoria: varChar, varShortInt, varInt, varLongInt.
Seleccionad la pestaña *Memoria* (parte inferior izquierda de la pantalla), escribid en el cuadro de texto: &varChar y pulsad Enter. Haced clic con el botón derecho encima de la ventana Dirección y seleccionad primero la opción Bytes y después la opción Hexadecimal. Haced lo mismo con el resto

0x001030	varLongInt	0xef	0xcd	0xab	0x89	0x67	0x45	0x23	0x01

Los valores que corresponden a cada variable están del mismo color, en rojo el valor de la variable varChar, en gris varChar2, a continuación tenemos varShortInt, varInt y varLongInt.

e) Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Anotad el valor asignado a los registros después de ejecutar cada una las siguientes instrucciones:

	rax	eax	ax	ah	al
mov rax, 0	0x0000000000000000	0x00000000	0x0000	0x00	0x00
mov rax, [varLongInt]	0x0123456789abcdef	0x89abcdef	0xcdef	0xcd	0xef
mov eax, [varInt]	0x0000000012345678	0x12345678	0x5678	0x56	0x78
mov ax, [varShortInt]	0x0000000012340123	0x12340123	0x0123	0x01	0x23
mov al, [varChar]	0x0000000012340101	0x12340101	0x0101	0x01	0x01

¿Por qué en cada instrucción se han modificado los valores de rax, eax, ax, ah y al?

Porque eax, ax, ah y al, son parte del registro rax. No son registros independientes. El nombre es para asignar valores al registro según el tipo de dato que utilizamos.

f) Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Observad que después de ejecutar la instrucción mov [varLongInt], rax se modifica la variable varLongInt de la ventana Expresiones vigiladas (watches) y 8 posiciones de memoria en la pestaña Memoria.

Expresión	**Valor**
/x varLongInt	0x0000000012340101

Dirección	**Variable**	**Valor de cada dirección de memoria (Byte/Hexadecimal)**							
0x601050	varLongInt	0x01	0x01	0x34	0x12	0x00	0x00	0x00	0x00

Dirección	**Variable**	**Valor**	
0x601050	varLongInt	0x01	varLongInt = 0x0000000012340101 Aquí se puede observar que los datos se guardan en formato little-endian. El byte menos significativo en la dirección de memoria más pequeña.
0x601051		0x01	
0x601052		0x34	
0x601053		0x12	
0x601054		0x00	
0x601055		0x00	
0x601056		0x00	
0x601057		0x00	

	rax	eax	ax	ah	al
	0x0000000012340101	0x12340101	0x0101	0x01	0x01
mov al, [varDB]	0x0000000012340101	0x12340101	0x0101	0x01	0x01
mov ax, [varDW]	0x0000000012340123	0x12340123	0x0123	0x01	0x23
mov eax, [varDD]	0x0000000001023567	0x01023567	0x3567	0x35	0x67
mov rax, [varDQ]	0x0123456789abcdef	0x89abcdef	0xcdef	0xcd	0xef

Observad que pasa lo mismo que en el punto anterior, donde se ha accedido a variables declaradas en C.

h) Seleccionad la pestaña Memoria (parte inferior izquierda de la pantalla), aseguraos de tener escrito el texto: &varChar, sino, lo escribís y pulsáis Enter. Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Observad que pasa después de ejecutar la instrucción mov [varChar], rax.

Expresión	Valor
/x varChar	0xef
/x varShortInt	0x89ab
/x varInt	0x01234567

Dirección	Variable	Valor de cada dirección de memoria (Byte/Hexadecimal)							
0x601048	varChar	0xef	0xcd	0xab	0x89	0x67	0x45	0x23	0x01

Se modifican las variables varChar, varShorInt y varInt de la ventana 'Expresiones vigiladas' (watches) y 8 posiciones de memoria en la pestaña Memoria, correspondientes a las variables varChar, varChar2, varShorInt y varInt. Esto es debido a que estamos asignando un valor de tipo Long Int (el registro rax es de 8 bytes – tipo QWORD) a una variable de tipo char (1 byte – tipo BYTE).

A diferencia de los lenguajes de alto nivel, en ensamblador el compilador no avisa que se está haciendo una asignación de forma incorrecta, te lo deja hacer, y por lo tanto, es responsabilidad del programador elegir los tipos y hacer las asignaciones correctamente. Hay que estar muy atento con esto cuando estéis desarrollando los ejercicios de ensamblador, como podéis ver, se pueden modificar datos que no esperábamos y el programa no funcionará de la forma deseada.

i) Añadid los registros rbx y rcx a la ventana 'Expresiones vigiladas' (watches). Mostrar los valores en formato hexadecimal utilizando el prefijo '/x' delante de los registros. (/x $rbx y /x $rax). En el punto 5) del apartado 3.5 del documento entorno de trabajo podéis consultar como hacerlo.

j) Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Anotad los valores de los registros rax, rbx y rcx después de ejecutar estas 3 instrucciones:

```
mov rax, [varLongInt]  ;
```


¿Qué diferencia hay en estas 3 instrucciones?

La primera y la tercera son equivalentes, puesto que el nombre de la variable varLongInt, se utiliza como dirección de memoria, esta dirección es la misma que hemos puesto en la tercera instrucción, y en los dos casos, tanto el nombre de la variable como la dirección de memoria están entre corchetes.
En la segunda instrucción, no estamos accediendo al dato, sino que obtenemos la dirección de memoria correspondiente a esta variable, el valor utilizado como dirección en la tercera instrucción.
Si hicierais mov rcx, [rbx] , estaríais haciendo lo mismo que en la primera y la tercera instrucción, cogiendo el dato (0x0000000012341212) que hay en la dirección de memoria que tenemos guardada enl registro rbx (0x601050).
Ahora todo esto os puede parecer un poco complicado, pero más adelante trabajaremos los modos de direccionamiento que podemos utilizar en una instrucción en ensamblador y bien seguro os quedará mucho más claro. Ahora mismo es sólo para indicar que hay que estar atento a la sintaxis de los operandos utilizados en las instrucciones de ensamblador.

k) Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Anotad el valor del registro eax después de ejecutar la instrucción mov eax, [varChar+2]:

Expression	Value
/x $eax	0x456789ab

¿De qué variable o de qué variables estamos cogiendo el valor que ponemos en el registro eax?

La dirección varChar+2 (0x601048+2=0x60104a) corresponde a la variable varShortInt, pero como el registro eax es de 4 bytes (tipo Int – QWORD), leemos 4 bytes a partir de esta dirección, y por lo tanto, cogemos los 2 bytes de la variable varShortInt y 2 bytes de la variable varInt.
Otra vez, estamos accediendo incorrectamente a los datos de memoria.

En este ejemplo sólo hemos utilizado instrucciones de transferencia (mov), pero los problemas expuestos los podemos tener con cualquier tipo de instrucción si no referenciamos correctamente las variables o no utilizamos el tipo de registro que corresponde.

l) Parad la ejecución del programa con la opción 'Matar' (Kill) del menú 'Ejecución'. Definid un punto de ruptura en la línea 51 (del archivo dataTypes.asm, código ensamblador). Ejecutad nuevamente el código con la opción 'Ejecutar' (Run) o pulsando la tecla F5. El programa se para en la línea 20 (del código C), volved a ejecutar nuevamente el código con la opción 'Ejecutar' (Run) o pulsando la tecla F5, pasamos al código ensamblador y el programa se para en la línea 51 (del código ensamblador) dónde hemos puesto el punto de ruptura.

m) Continuamos ejecutando el programa con la opción 'Entrar en la función' (Step into) o pulsando la tecla F8. Anotad el valor del registro 'al' después de ejecutar cada una de les 4 instrucciones siguientes:

Instrucción	Expresión	Valor
	/x $al	0xab
mov al, valueBinary	/x $al	0x3c
mov al, valueOctal	/x $al	0x3c

siempre trabaja en binario utilizando la representación de complemento a 2.

Llegados a este punto, ya tenéis instalado el entorno a trabajo, habéis comprobado que funciona correctamente, habéis empezado a utilizar el depurador (kdbg) y hemos revisado algunos conceptos importante a tener en cuenta a la hora de trabajar en ensamblador.

Todo a punto para empezar los ejercicios de ensamblador.