Representación de los Datos y Operaciones Aritmético-Lógicas

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE
v1.0.0	2018 Octubre 1		CA

Índice

1. Introducción

1.1. Objetivos

- Programación:
  - Desarrollar programas que almacenen y procesen distintos tipos de datos como enteros con signo, caracteres, arrays, strings de distintos tamaños como 1 byte, 2 bytes, 4 bytes, etc..
  - Empleo de los sufijos de los mnemónicos: analizar el tamaño de los operandos según el sufijo del mnemónico empleado, el tamaño del operando registro y el tipo de operando en memoria.
  - Emplear distintos tipos de modos de direccionamiento (inmediato, directo, indirecto, indexado) de acceso a los operandos
  - Empleo de Macros mediante directivas.
  - Realizar operaciones aritméticas (suma,resta,multiplicación y división)
  - Concepto de llamada al sistema operativo.
- Análisis:
  - Comprobación del tipo de alineamiento *little endian* de los datos almacenados en memoria
  - Comprobar cómo afectan las operaciones lógicas y aritméticas a los flags del registro de estado EFLAGS
  - Analizar el contenido de la memoria como números con signo, caracteres, arrays y strings: tipos y tamaños de los operandos numéricos y de los operandos alfanuméricos
  - Realizar la operación de desensamblaje para comprobar el lenguaje máquina del módulo ejecutable cargado en la memoria principal

1.2. Módulos fuente

- **datos_size.s**
  - Declaración del tamaño de los operandos:
    - Mediante las directivas (.byte, .2byte, .short, etc ..)
  - Declaración de arrays de datos numéricos mediante directivas (.short,.int, etc ..)
  - Declaración de datos alfanuméricos mediante:
    - Directivas del ensamblador (.ascii, .asciiz, .string, etc)
  - Empleo de Macros
- **datos_sufijos.s**
  - Acceso a los operandos mediante instrucciones con los sufijos (b,w,l,q)
- **datos_direccionamiento.s**
- Diferentes Modos de direccionamiento de los operandos: inmediato, directo, indirecto, indexado

1.3. Requisitos

- Teoría: representación de datos, operaciones aritméticas y lógicas, formato de instrucciones y repertorio ISA de la arquitectura X86.
  - Almacenamiento con alineamiento interno de bytes "little endian"
- Práctica previa: Introducción a la programación en lenguaje ensamblador AT&T x86-32
- Conceptos del lenguaje de programación C:
  - Punteros, array, string y operación de casting.

2. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- Apuntes y Libro de Texto
- Documentación Memoria: Contenido y Formato de la Memoria
- Evaluación: sistema de evaluación
- Plataforma de Desarrollo : configuración de la computadora personal
- Programación : metodología

3. Registros internos de la CPU

- La arquitecura amd64 dispone de:
  - 16 registros de propósito general (RPG) de 64 bits cada uno: rax,rbx,rcx,rdx,rsi,rdi,rsp, etc
  - 1 registros de estado de 64 bits: rflags
- El acceso a los registros de propósito general puede ser *parcial*:
  - Registro RAX: es un registro de 64 bits
  - Registro EAX: son los 32 bits de menor peso de RAX
  - Registro AX: son los 16 bits de menor peso de RAX
  - Registro AL: son los 8 bits de menor peso de RAX
  - Registro AH: es el byte con los bits de las posiciones 8:15 de RAX
- En las hojas de referencia rápida están representados todos los nombres de los difererentes grupos de bits de cada registro de propósito general.

4. Tamaño de los datos y variables

4.1. Algoritmo

- No se desarrolla ningún algoritmo. Unicamente el bloque de código de salida.

4.2. Edición del Módulo fuente: datos_size.s

- Descargar el módulo fuente "datos_size.s" de miaulario y añadir los comentarios apropiados.

```
### Programa: datos_size.s
### Descripción: declarar y acceder a distintos tamaños de operandos
        ## MACROS
        .equ SYS_EXIT,  1
        .equ SUCCESS,   0

        ## VARIABLES LOCALES
        .data

da1:    .byte   0x0A
da2:    .2byte  0x0A0B
da4:    .4byte  0x0A0B0C0D
men1:   .ascii  "hola"
lista:  .int    1,2,3,4,5

        ## INSTRUCCIONES
        .global _start
        .text
_start:
        mov $SYS_EXIT, %eax
        mov $SUCCESS,  %ebx
        int $0x80

        .end
```

4.3. Compilación

- Seguir los pasos del proceso de compilación común a todas las sesiones.
  - `gcc -m32 -g -o datos_size datos_size.s`

4.4. Ejecución

- `./datos_size`
- `echo $?`

4.5. Análisis del módulo fuente

- Macro:
  - La construcción macro se utiliza en el programa fuente para sustituir datos utilizados en el programa fuente por símbolos de texto que faciliten la lectura del código fuente.
  - Para ello empleamos la directiva "EQU" cuya sintaxis es: `.EQU SÍMBOLO, dato`

  - El preprocesador en la primera fase de la compilación sustituirá el texto SIMBOLO que aparece a lo largo de la sección de datos e instrucciones por el dato asociado.

- Macros empleadas
  - SYS_EXIT : código de la llamada al sistema para finalizar el programa y devolver el control al Sistema Operativo. En la arquitectura i386 su valor es 1.
  - SUCCESS : código empleado por los programas para indicar que su ejecución se ha realizado con normalidad. Su valor es 0.
- LLamadas al sistema operativo
  - El programa llama al sistema operativo linux a través de una interrupción software mediante la instrucción `int 0x80`
  - Antes de realizar la llamada:
    I. almacena en el registro EAX el código de la subrutina o función de la llamada: para la función salida "EXIT" el código es 1
    II. almacena en el registro EBX el valor del argumento transferido a la subrutina o función llamada.
  - la llamada al sistema operativo para la ejecución del función EXIT es necesaria para finalizar cualquier programa y devolver el control al sistema operativo.
- Sección de datos:
  - interpretar las etiquetas y directivas de reserva de memoria e inicialización para los datos utilizando la tabla de directivas: identificar las variables ordinarias, strings y arrays.
- Sección de instrucciones:
  - determinar la instrucción de entrada al programa.
  - determinar la construcción de salida del programa.

4.6. Ejecución paso a paso

4.6.1. Operaciones

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable.
- Abrir el depurador GDB y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
- Configurar el fichero para el logging histórico de los comandos.
- Activar un punto de ruptura en la instrucción de entrada al programa.
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
- Análisis del contenido de la memoria principal mediante el depurador GDB.

```
//Alineamiento de los bytes de un dato
x /tb &da1
x /xh &da2
x /xw &da4
x /5xb &da4

//Alineamiento de los bytes de un string
x /5cb &men1
x /5xb &men1

//Volcado de un string
p /s (char *)&men1
```

```
//Volcado de un array
x /5xw &lista
p /a &lista
p /a &lista+1
p lista
p lista@5

//volcado de una instrucción
p &_start
x /i &_start

//Desensamblar: Conversión del código máquina en ensamblador
disas /r _start
```

- comando eXaminar **x**: vuelca el contenido de una **dirección** de memoria
  - formato /nvt : "t" es el *tamaño* de variable en memoria , "v" el código del *valor* del contenido de memoria a visualizar y "n" el *número* de veces que hay que volcar secuencialmente grupos de bytes en memoria de tamaño "t" comenzando en la dirección *&variable*
  - `help x` : formatos d (decimal) ,x (hexadecimal),t (binario) ,o (octal) ,c (character) ,a (address),i (instruction),etc
- operador **&** : se utiliza como prefijo de una etiqueta para evaluar la dirección de memoria a la que hace referencia una etiqueta
- operador ***** : se utiliza para evaluar el contenido de una posición de memoria mediante la indirección de un puntero
- operación de **casting**:
  - El casting consiste en definir o redifinir el tipo de variable. Se utiliza como prefijo de la variable a redefinir y va entre paréntesis.
  - Ej. (char *): el tipo `char *` es un puntero a un entero de 1 byte.
- comando Print **p**: Evalua el argumento del comando y el valor resultante lo imprime en pantalla
  - Ej. p /a &lista : evalua &lista cuyo valor resulante se imprime con formato tipo "a" (address)
  - formatos de impresión: los mismos que eXaminar: `help x`
  - operador **etiqueta@n**: evalua etiqueta y repite la evaluación con los 5 objetos en secuencia a la dirección etiqueta
- comando **disas** : desensambla el código binario traduciéndolo a código ensamblador.

4.6.2. Cuestiones

- ¿En qué orden se guardan los bytes del dato da4?
- ¿En qué orden se guardan los caracteres del string "hola"?
- ¿Cuál es el código ASCII del carácter *o*?
- ¿Cuál es la dirección de memoria principal donde se almacena el string "hola"?
- ¿Cuál es la dirección memoria principal donde se almacena el array lista?
- ¿Cuál es el contenido de los primeros 4 bytes a partir de la dirección anterior en sentido ascendente?

5. Tamaño de los Operandos

5.1. Edición del Módulo fuente: datos_sufijos.s

- Descargar el módulo fuente "datos_sufijos.s" de miaulario y añadir los comentarios apropiados.

```
### Program: datos_sufijos.s
### Descripción: utilizar distintos sufijos para los mnemónicos indicado distintos tamaños ↩
     de operandos
        ## MACROS
        .equ SYS_EXIT,  1
        .equ SUCCESS,   0

        ## VARIABLES LOCALES
        .data

da1:    .byte   0x0A
da2:    .2byte  0x0A0B
da4:    .4byte  0x0A0B0C0D
saludo: .ascii  "hola"
lista:  .int    1,2,3,4,5

        ## INSTRUCCIONES
        .global _start
        .text
_start:

        ## Reset de Registros
        xor  %eax,%eax
        xor  %ebx,%ebx
        xor  %ecx,%ecx
        xor  %edx,%edx

        ## Carga de datos
        ## mov da1,da4          ERROR: por referenciar las dos direcciones efectivas de  ↩
            los dos operandos a la memoria principal
        mov  da4,%eax
        movl da4,%ebx
        movw da4,%cx
        movb da4,%dl

        ## Reset de Registros
        xor  %eax,%eax
        xor  %ebx,%ebx
        xor  %ecx,%ecx
        xor  %edx,%edx

        ## Carga de datos
        ## movw  da4,%al        ERROR: incoherencia entre -w y AL
        mov  da4,%al            #aplica el tamaño de DL
        movb da4,%ebx           #aplica el regisro BL debido al sufijo

        mov  da1,%ecx
        mov  da4,%dx

        ## Reset de Registros
        xor  %eax,%eax
        xor  %ebx,%ebx
        xor  %ecx,%ecx
        xor  %edx,%edx
```

```
        ## Carga de datos

        mov  da1,%al

        ## inc da1              ERROR:  sin el sufijo en inc por ser la dirección efectiva ↩
             del operando una referencia a la memoria principal
        incb da1
        incw da2
        incl da4

## salida
        mov $SYS_EXIT, %eax
        mov $SUCCESS,  %ebx
        int $0x80

        .end
```

5.2. Compilación

- Seguir los pasos de la compilación de un módulo en lenguaje ensamblador.
  - `gcc -m32 -g -o datos_sufijos datos_sufijos.s`

5.3. Ejecución

- `./datos_sufijos`
- `echo $?`

5.4. Análisis del módulo fuente

- Sufijos de los mnemónicos indicando distintos tamaños de los operandos. Es el tamaño del registro quien impone el tamaño del operando.

5.5. Ejecución paso a paso

5.5.1. Operaciones

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable.

```
x /tb &da1
```

5.5.2. Cuestiones

- ¿En qué orden se guardan los bytes del dato da4?

6. Modos de Direccionamiento

6.1. Edición del Módulo fuente: datos_direccionamiento.s

- Descargar el módulo fuente "datos_direccionamiento.s" de miaulario y añadir los comentarios apropiados.

```
### Program:     datos_direccionamiento.s
### Descripción: Emplear estructuras de datos con diferentes direccionamientos,  ↩
    operaciones lógicas y aritméticas y saltos de control de flujo condicionales.

        ## MACROS
        .equ SYS_EXIT,  1
        .equ SUCCESS,   0

        ## VARIABLES LOCALES
        .data

        .align 4                                        # Alineamiento con direcciones de MP  ↩
            múltiplos de 4
da2:    .2byte  0x0A0B,0b0000111101011100,-21,0xFFFF # Array da2 de elementos de 2 bytes
        .align 4
lista:  .word    1,2,3,4,5      # Array lista de elementos de 2 bytes
        .align 8
buffer: .space  100             # Array buffer de 100 bytes
        .align 2
saludo:
        .string "Hola"          # Array saludo de elementos de 1 byte por ser caracteres

        ## INSTRUCCIONES
        .global main
        .text
main:

        ## RESET

        xor  %eax,%eax
        xor  %ebx,%ebx
        xor  %ecx,%ecx
        xor  %edx,%edx
        xor  %esi,%esi
        xor  %edi,%edi

        ## ALGORITMO sum1toN

        ## Direccionamiento inmediato
        mov $4,%si
        ## Direccionamiento indexado
bucle:  add lista(,%esi,2),%di
        ## Direccionamiento a registro
        dec %si
        ## Direccionamiento relativo al PC
        jns bucle


        ## EJERCICIOS SOBRE DIRECCIONAMIENTO

        ## Direccionamiento indirecto
        lea buffer,%eax         #inicializo el puntero EAX
        ## mov da2,(%eax) ERROR: la dirección efectiva de los dos operandos hacen  ↩
            referencia a la memoria principal
        mov da2,%bx
```

```
        mov %bx, (%eax)
        ## Direccionamiento directo
        incw da2
        ## Direccionamiento indexado
        lea  da2,%ebx
        ## inc 2(%ebx) ERROR: dirección efectiva a memoria y no hay sufijo
        incw 2(%ebx)

        mov $3,%esi
        mov da2(,%esi,2),%ebx

        ## SALTOS INCONDICIONALES

        ## Direccionamiento relativo
        jmp salto1              #salto relativo al contador de programa pc -> eip
        xor %esi,%esi
salto1:

        ## OPERACIONES ARITMETICAS

        # imul: multiplicación con signo: AX<- BL*AL
        movb $-3,%bl
        movb $5,%al
        imulb %bl
        movw $5,%ax             #dividendo
        movb $3,%bl             #divisor
        ## idiv: división con signo .   (AL=Cociente, AH=Resto) <- AX/(byte en registro o  ↩
            memoria)
        idivb %bl           # 5/3 = 1*3 + 2
        negb %bl            # negación: complemento a 2

        ## OPERACIONES LOGICAS
        mov $0xFFFF1F, %eax
        mov $0x0000F1, %ebx
        not %eax
        and %ebx,%eax
        or  %ebx,%eax
        mov %ebx,%eax           #Complemento a 2 mediante operación not()+1
        not %eax
        inc %eax
        shr $4,%eax             #desplazamiento lógico: bits a introduccir -> 0..
        sar $4,%eax             #desplazamiento aritmético: bits a introducir -> extensión ↩
             del signo



        ## SALIDA

        mov $SYS_EXIT, %eax
        mov $SUCCESS,  %ebx
        int $0x80

        .end
```

6.2. Compilación

- Seguir los pasos de la compilación de un módulo en lenguaje ensamblador.
  - `gcc -m32 -g -o datos_direccionamiento datos_direccionamiento.s`

6.3. Ejecución

- `./datos_direccionamiento`
- `echo $?`

6.4. Análisis del módulo fuente

- Alineación de datos mediante la directiva `.align`: `.align n` asigna una dirección de memoria múltiplo de *n* al siguiente dato declarado.
- Macro:

6.5. Ejecución paso a paso

6.5.1. Estructuras de datos

- Array *da2*
  - `ptype da2`: no debug info: al no tener información el debuger del tamaño de los elementos es necesario indicarlos explicitamente en los comandos posteriores.
  - Imprimir la dirección de memoria del array *da2* y el contenido del primer elemento: `x /xh &da2`
  - 4 elementos de 2bytes del array da2: `x /4xh &da2`
    - Es necesario realizar un **casting** : Array de 4 elementos de tamaño 2bytes: `p /x (short[4])da2`
    - Fijarse con el comando *eXaminar* el resultado es independiente de sí hacemos un **casting** (short *): `x /4xh (short *)&da2`
      - El tamaño y tipo de dato lo fija el argumento del comando: */4xh*
  - Comprobar la norma de almacenamiento *little endian* identificando cada dirección de memoria a un byte con su contenido.
  - Acceder a la dirección de memoria del elemento de valor -21 del array *da2*:
    - el argumento elemento de array en `p da2[2]` no es válido ya que el debugger carece de información
- Array *lista*
  - `ptype lista`
  - `p (short[5])lista`
- Array *buffer*
  - `ptype buffer`
  - Imprimir la dirección de memoria del array *buffer* y comprobar su alineamiento: `p &buffer`
- String
  - `ptype saludo` : no debug info → no admite referencia elemento array expresión *saludo[n]*
  - `p /c (char[5])saludo` :casting array
  - `x /5c (char *)&saludo` :casting puntero
  - `p /c *(char *)&saludo` :casting puntero e indirección
  - `p /s (char *)&saludo` :casting puntero y formato string
- Desensamblar
  - `disas salto1`
  - `disas /r salto1`

6.5.2. Operaciones aritméticas

- Comprobar el resultado de la división de números enteros con signo

6.5.3. Operaciones lógicas

- Comprobar los resultados de las operaciones

6.5.4. Cuestiones

- ¿En qué orden se guardan los bytes del dato da4?