Índice

Marina de la Cruz
Alfonso Ortega

Esquema básico de un programa NASM

```
segment .data
...
...
segment .bss
...
...
```

Sección de datos (opcional)

```
segment .text
global main
extern scan_int, scan_float ...
_fun1:
...
...
ret
_fun2:
...
...
ret
main:
...
...
ret
```

Sección de código

La sección de datos (I)

Introducción

- La sección de datos contiene las declaraciones de las variables del programa.
- Dentro de la sección de datos se distinguen dos subsecciones:
  - **segment .data:** en esta subsección se declaran variables inicializadas.
  - **segment .bss**: reservado para la declaración de variables no inicializadas.
- En un programa no es necesario que aparezcan ambas subsecciones. Incluso puede no aparecer ninguna de ellas.
- La declaración de las variables se ajusta a una sintaxis diferente en cada una de las subsecciones.

Variables inicializadas

- La variables inicializadas se declaran en la subsección **"segment .data"** según la siguiente sintaxis:

<nombre variable> <tamaño> <valor inicial>

Donde:

- **<nombre variable>** es el identificador de la variable
- **<tamaño>** es el tamaño de la variable según la siguiente notación:
  - db: 1 byte
  - dw: 2 bytes
  - dd: 4 bytes
  - ...
- **<valor inicial>** es el valor de inicialización de la variable

- Ejemplos:
  - unbyte db 0
  - unword dw 10
  - undoubleword dd 1000
  - Mensaje1 db "División por cero", 0

```
section .data
    var1 bb 0
    var2 bb 1
```

Variables no inicializadas

- La variables no inicializadas se declaran en la subsección **"segment .bss"** según la siguiente sintaxis:

  <nombre variable> <tamaño> <cantidad>

  Donde:
  - **<nombre variable>** es el identificador de la variable
  - **<tamaño>** es el tamaño de la variable según la siguiente notación:
    - resb: byte
    - resw: 2 bytes
    - resd: 4 bytes (este es el tamaño de datos que vamos a utilizar)
    - ...
  - **<cantidad>** es la cantidad de posiciones de tamaño indicado por el campo <tamaño> que se reservan para la variable

- Ejemplos:
  - unbyte resb 1
  - doswords resw 2
  - ochodoublewords resd 8 (indicado para vectores)

La sección de datos (IV)

Uso de los segmentos de datos en el desarrollo del compilador

- Se utilizará el segmento de datos **.bss** para la declaración de las **variables** del programa (escalares y vectoriales).
- Todas las variables escalares (lógicas y enteras) se declararán de 32 bits (***resd***).
- Los nombres de las variables ensamblador irán precedidos del símbolo "_" para evitar errores.
- Se utilizará el segmento de datos **.data** para la inicialización de los **mensajes de error** en tiempo de ejecución.

La sección de código (I)

Estructura

- La sección de código contiene:
  - Declaración de símbolos globales. 1
  - Declaración de símbolos externos. 2
  - Instrucciones correspondientes a la traducción a NASM del código **o**. 3

```
segment .text
global main          1
extern scan_int, scan_float ...   2
_fun1:
...
...
ret
_fun2:
...
...                  3
ret
main:
...
...
ret
```

Símbolos externos

- Son aquellos que no se definen en el fichero, pero que se utilizan y por lo tanto se asume que están definidos en otro fichero. Por ejemplo, las funciones de la librería **olib.o** que se proporcionan al alumno para las operaciones de entrada/salida.

- Para generar correctamente el ejecutable hay que enlazar con el fichero que contenga la definición de los símbolos.

- La definición de símbolos externos se realiza con la palabra reservada **extern** seguida del símbolo o símbolos separados por comas. Por ejemplo:

```
extern simbolo1, simbolo2, simbolo3
```

- Los símbolos externos que tienen que aparecer son las siguientes funciones de la librería **olib.o**:
  - scan_int
  - scan_boolean
  - print_int
  - print_boolean
  - print_blank
  - print_endofline
  - print_string

Símbolos globales

- Son aquellos que se definen en el módulo y que pueden ser utilizados desde el exterior.

- La definición se realiza con la palabra reservada **global** seguida del símbolo o símbolos separados por comas. Por ejemplo:

```
global    main
```

Instrucciones

- Registros de propósito general de 32 bits: **eax**, **ebx**, **ecx**, **edx**
- Puntero de pila: **esp**
- Cuando aparezca en una instrucción el nombre de una variable, representa la dirección de memoria de dicha variable. Para acceder al contenido hay que utilizar corchetes alrededor del nombre (diferencia con TASM).
- Los **códigos de operación** se pueden consultar en el manual de NASM (algunos se verán en los ejemplos de NASM y otros cuando se implemente el generador de código).
- Los **comentarios** son de una línea y comienzan con el carácter ";".
- **Convenio de llamadas a funciones**: se sigue el convenio de llamadas C:
  a. El llamador inserta en la pila los parámetros de la llamada.
  b. Se invoca a la función.
  c. Al terminar, se restaura la pila (eliminamos los parámetros con `add esp,4`).
  d. La función deja el resultado (si es entero) en el registro `eax`.

Instrucciones

- Registros de propósito general de 32 bits: **eax**, **ebx**, **ecx**, **edx**
- Puntero de pila: **esp**
- Cuando aparezca en una instrucción el nombre de una variable, representa la dirección de memoria de dicha variable. Para acceder al contenido hay que utilizar corchetes alrededor del nombre (diferencia con TASM).
- Los **códigos de operación** se pueden consultar en el manual de NASM (algunos se verán en los ejemplos de NASM y otros cuando se implemente el generador de código).
- Los **comentarios** son de una línea y comienzan con el carácter ";".
- **Convenio de llamadas a funciones**: se sigue el convenio de llamadas C:
  a. El llamador inserta en la pila los parámetros de la llamada.
  b. Se invoca a la función.
  c. Al terminar, se restaura la pila (eliminamos los parámetros con `add esp,4`).
  d. La función deja el resultado (si es entero) en el registro `eax`.

Instrucciones

- En general, la mayoría de las instrucciones permiten:
  - accesos a registro
  - accesos a memoria
  - accesos inmediatos (a valores explícitos)

- No obstante (cuando se implemente el compilador se verá), **los operandos de las operaciones aritmético-lógicas estarán, en la mayoría de los casos, almacenados en los registros de carácter general.**

- **Movimiento de datos**
  - Código de operación: **mov**
  - Copia el contenido del segundo operando en el primero.
  - nasm no guarda el tipo de las variables, por ello hay instrucciones de movimiento que son ambiguas, por ejemplo mov [_x], 8. Como nasm desconoce el tamaño de la variable _x no puede deducir con cuántos bytes representar el valor 8. En estos casos se especifica el tamaño de los datos a mover utilizando una palabra reservada. En el caso del compilador, todos los datos van a ser de 32 bits, y se utilizará la palabra reservada **dword** en todos los movimientos de datos (incluso en los que no sean ambiguos).
  - Ejemplo con acceso a registros: mov dword eax, edx
  - Ejemplo con acceso a registro y memoria: mov dword eax, [_x]

Instrucciones

- **Operaciones con la pila**
  - Código de operación: **push**
  - Apila el operando
  - Uso del cualificador **dword**
  - Ejemplo con acceso a memoria: push dword [_x]
  - Ejemplo con acceso a registro: push dword eax (dword no es necesario)

- **Operaciones con la pila**
  - Código de operación: **pop**
  - Desapila al operando
  - Uso del cualificador **dword**
  - Ejemplo con acceso a registro: pop dword eax
  - Ejemplo con acceso a memoria: pop dword [_y]

Instrucciones

```
section .text
   global main
main:
   mov eax, 100
   mov edx, 5
   add eax, edx
   ret
```

- **Suma de enteros**
  - Código de operación: **add**
  - Realiza la suma de los dos operandos y deja el resultado en el primero.
  - Ejemplo con acceso a registros: add eax, edx
- **Resta de enteros**
  - Código de operación: **sub**
  - Realiza la resta de los dos operandos y deja el resultado en el primero.
  - Ejemplo con acceso a registros: sub eax, edx
- **Menos unario**
  - Código de operación: **neg**
  - Niega el contenido del operando.
  - Ejemplo con acceso a registros: neg eax

Instrucciones

- **División de enteros**
  - Código de operación: **idiv**
  - Ejemplo con acceso a registros: `idiv ecx`
  - Funcionamiento: esta instrucción trabaja de la siguiente manera,
    - Asume que el dividendo está en la composición de registros `edx:eax`.
    - El divisor es el registro que aparece en la instrucción.
    - El resultado de la división entera se ubica en el registro `eax`.
    - El resto de la división entera se ubica en el registro `edx`.

    Para que el dividendo esté en la composición de registros edx:eax y realizar correctamente la división se tiene que realizar la siguiente secuencia:
    - Cargar el dividendo en `eax`.
    - Extender el dividendo a la composición de registros `edx:eax` utilizando el código de operación **`cdq`**.
    - Cargar el divisor en `ecx`.
    - Realizar la división: `idiv ecx`.

- **Multiplicación de enteros**
  - Código de operación: **imul**
  - Ejemplo con acceso a registros: `imul ecx`.
  - Funcionamiento: esta instrucción trabaja de la siguiente manera,
    - Asume que uno de los operandos está en el registro `eax` y otro en el registro que aparece en la instrucción.
    - El resultado se carga en `edx:eax` y consideraremos como resultado de la operación el valor contenido en `eax`.

Instrucciones

- **Conjunción**
  - Código de operación: **and**
  - Realiza la conjunción bit a bit de los dos operandos y deja el resultado en el primero. Internamente en el compilador se representará el valor lógico *true* con un 1 y el valor lógico *false* con un 0.
  - Ejemplo con acceso a registros: and eax, edx

- **Disyunción**
  - Código de operación: **or**
  - Realiza la disyunción bit a bit de los dos operandos y deja el resultado en el primero. Internamente en el compilador se representará el valor lógico *true* con un 1 y el valor lógico *false* con un 0.
  - Ejemplo con acceso a registros: or eax, edx

- **Negación lógica**
  - El código de operación not no es válido porque no realiza la negación booleana (hace el complemento a 1)
  - La negación lógica se hará con un código diseñado específicamente para ello (antes de verlo hay que conocer el código de operación de la comparación entera, **cmp**, y los saltos).

Instrucciones

Comparación entera

- Código de operación: **cmp**
- Realiza la resta de los operandos y dependiendo del resultado actualiza los flags del sistema (el resultado de la resta no se almacena)
- Ejemplo con acceso a registros: cmp eax, edx

Saltos condicionales

- Las instrucciones de salto se ubican inmediatamente después de una instrucción cmp ya que el salto se realiza en función del estado de los flags del sistema.
- Las instrucciones de salto disponibles, para comparaciones de enteros con signo) son las siguientes:

je <etiqueta>	salta a etiqueta si eax == edx	(también **jz**)
jne <etiqueta>	salta a etiqueta si eax != edx	(también **jnz**)
jle <etiqueta>	salta a etiqueta si eax <= edx	(también **jng**)
jge <etiqueta>	salta a etiqueta si eax >= edx	(también **jnl**)
jl <etiqueta>	salta a etiqueta si eax < edx	(también **jnge**)
jg <etiqueta>	salta a etiqueta si eax > edx	(también **jnle**)

Creación de ejecutables

- Para crear un ejecutable a partir de un fichero fuente escrito en NASM, se utilizan los siguientes comandos:

```
nasm  -g  -o  [<fichero objeto (.o)>]  -f elf  <fichero fuente (.asm)>
gcc  -o  <fichero ejecutable>  <fichero objeto 1>  <fichero objeto 2> ...
```

Por ejemplo, si se escribe en NASM un programa que se llama **ej1.asm**, y que utiliza funciones de la librería **olib.o** para generar el ejecutable se hace:

```
nasm  -g  -o  ej1.o  -f elf  ej1.asm
gcc  -o  ej1  ej1.o  olib.o
```

Posibles problemas en arquitectura de 64 bits

Problema: Mensajes del estilo:

```
skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/6/libgcc.a when searching for -lgcc
cannot find -lgcc
```

Solución: Instalar librerías del sistema de 32 bits (sólo necesario una vez)

```
$ sudo apt-get install g++-multilib libc6-dev-i386
```

Problema:

```
i386 architecture of input file 'ej1.o' is incompatible with i386:x86-64 output
```

Solución:

```
nasm -g -o ej1.o -f elf32 ej1.asm
gcc -m32 -o ej1 ej1.o olib.o
```

¡CUIDADO!

No copiar las líneas anteriores de estas diapositivas, pues puede dar el siguiente error:

```
nasm: error: more than one input file specified
```

Es por ello importante escribir el comando de nuevo.

Resumen de los pasos para conseguir un ejecutable

1. Compilar las funciones creadas en generacion.c junto al ejemplo en cuestión (**ej1.c**)
   ```
   gcc -Wall -g -c ej1_asm generacion.c ej1.c
   ```

1. Ejecutar el ejecutable obtenido para sacar el fichero .asm (como argumento)
   ```
   ./ej1_asm ej1.asm
   ```

1. Compilar el fichero nasm obtenido
   ```
   nasm -g -o $ejemplo.o -f elf32 $ejemplo.asm
   ```

1. Compilar con gcc sobre el objeto obtenido en 3
   ```
   gcc -Wall -g -m32 -o ej1 ej1.o olib.o
   ```

1. Ejecutar el ejecutable obtenido
   ```
   ./ej1
   ```

Se recomienda usar **Valgrind** para comprobar errores.