

NASM

- Esquema básico de un programa NASM
- La sección data
 - Introducción
 - Variables inicializadas
 - Variables no inicializadas
 - Utilización de los segmentos de datos durante el desarrollo del compilador
- La sección de código
 - Estructura
 - Símbolos extern
 - Símbolos global
 - Comandos
- Creación de los binarios ejecutables

Esquema básico de un programa NASM

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

data section (optional)

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

code section

Sección data I

- La sección data incluye la declaración de las variables del programa
- Dentro de la sección data se distinguen dos subsecciones:
 - `segment .data`: Declaración de las variables inicializadas
 - `segment .bss`: Declaración de las variables no inicializadas
- Incluir ambas secciones dentro de un programa no es obligatorio. Incluso se puede omitir ambas
- La declaración de variables se expresa con una sintaxis diferente en cada una de las subsecciones

Sección data II

- Las variables inicializadas se declaran en la subsección “segment .data” de acuerdo a la siguiente sintaxis:
 - `<variable's name> <size> <initial value>`
- donde:
 - **<variable's name>** es el identificador de la variable
 - **<size>** es el tamaño de la variable de acuerdo a su notación
 - db: 1 byte
 - dw: 2 bytes
 - dd: 4 bytes
 - ...
 - **<initial value>** es el valor inicial de la variable

Sección data III

- Ejemplos de declaración de variables inicializadas:
 - `abyte db 0`
 - `aword dw 10`
 - `adoubleword dd 1000`
 - `div_error_message db "Division by zero", 0`

Sección data IV

- Las variables no inicializadas se declaran en la subsección “segment .bss” de acuerdo a la siguiente sintaxis:
 - `<variable's name> <size> <quantity>`
- donde:
 - **<variable's name>** es el nombre del identificador de la variable
 - **<size>** es el tamaño de la variable de acuerdo a su notación
 - resb: byte
 - resw: 2 bytes
 - resd: 4 bytes
 - ...
 - **<quantity>** es la cantidad de posiciones **<size>** de memoria que se reservarán para almacenar la variable especificada

Sección data V

- Ejemplos de la declaración de variables no inicializadas:
 - `abyte resb 1`
 - `twowords resw 2`
 - `eightdoublewords resd 8` (utilizaremos una cantidad múltiple para los vectores)

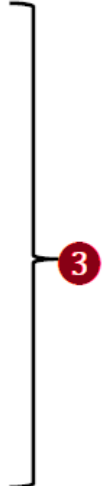
Sección data VI

- Utilizaremos el segmento **bss.** para declarar las **variables del programa** (escalares y vectores)
- Todas las variables escalares (booleanas y enteras) se declararán como de 32 bit (**resd**)
- Los nombres de las variables estarán precedidos de un símbolo “_” para evitar errores
- Utilizaremos el segmento **.data** para inicializar los **mensajes de error en tiempo de ejecución**

Sección de código I

- La sección de código contiene:
 - La declaración de símbolos globales ❶
 - La declaración de símbolos externos ❷
 - Instrucciones correspondientes a la compilación de los programas ALFA en programas NASM ❸

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



Sección de código II

- Los símbolos externos no están definidos en el fichero actual, pero dado que se utilizan, se asume que están definidos en un fichero diferente. Por ejemplo, las funciones proporcionadas por la librería **alfalib.o** para las operaciones de entrada y salida
- Debemos enlazar el fichero que contiene la definición de estos símbolos para generar correctamente el ejecutable
- La definición de estos símbolos externos se alcanza utilizando la palabra reservada **extern** seguida de un símbolo o un conjunto de símbolos separados por comas. Por ejemplo:
 - `extern symbol1, symbol2, symbol3`

Sección de código III

- Las funciones de la librería **alfalib.o** deben declararse como símbolos externos en nuestros programas NASM:
 - `scan_int`
 - `scan_boolean`
 - `print_int`
 - `print_boolean`
 - `print_blank`
 - `print_endofline`
 - `print_string`

Sección de código IV

- Los símbolos globales son aquellos definidos dentro de un programa NASM que pueden utilizarse desde el exterior
- La definición de los símbolos globales se alcanza utilizando la palabra reservada **global** seguida de un símbolo o de un conjunto de símbolos separados por comas. Por ejemplo:
 - `global main`

Sección de código V

- Registros de propósito general de 32 bit: **eax**, **ebx**, **ecx**, **edx**
- Puntero de pila: **esp**
- Cuando se incluye el nombre de una variable en una instrucción es la dirección de memoria de dicha variable. Se necesita utilizar los corchetes, es decir, “[_a]”, para acceder a su contenido
- Los **códigos de operación** se pueden consultar en la especificación de NASM (algunos de ellos se presentarán en los ejemplos de NASM y veremos otros cuando implementemos el generador de código)

Sección de código VI

- Los comentarios son de una línea y empiezan por el símbolo “;”
- Se seguirá la convención del lenguaje C para **llamadas de funciones**:
 - El “caller” almacena en la pila los parámetros de la llamada
 - El “caller” llama a la función
 - Cuando la función termina, restauramos la pila (se eliminan los parámetros con `add esp, 4`)
 - La función almacena el valor de retorno (si es un entero) en el registro `eax`

Sección de código VII

- En general, la mayoría de las instrucciones NASM permiten
 - Acceso a registros
 - Acceso a memoria
 - Acceso directo (a valores explícitos)
- No obstante, (se verá cuando implementemos el compilador), normalmente, los operandos de las operaciones aritmético-lógicas se almacenarán en los registros de propósito general: **eax**, **ebx**, **ecx**, **edx**

Sección de código VIII

- Movimientos de datos

- Código de la operación: **mov**
- **mov** copia los contenidos del segundo operando en el primero
- NASM no almacena el tipo de las variables, por tanto existen algunas instrucciones de movimiento ambiguas, por ejemplo:

```
mov [_x], 8
```

- En este caso, NASM no sabe el tamaño de la variable `_x`, y no puede deducir cuántos bytes representan el valor 8. En estos casos debemos especificar el tamaño del dato a mover utilizando una palabra clave. En el caso de nuestro compilador, todos los datos que se utilizan son de un tamaño de 32 bit, por tanto utilizaremos la palabra clave **dword** en todos nuestros movimientos de datos (incluyendo aquellos que no son ambiguos)
- Ejemplo con acceso a registros: `mov dword eax, edx`
- Ejemplo con acceso a memoria y a registros: `mov dword eax, [_x]`

Sección de código IX

- Operaciones de pila I
 - Código de operación: **push**
 - Almacena el operando
 - Utilización del calificador **dword**
 - Ejemplo con acceso a memoria: `push dword [_x]`
 - Ejemplo con acceso a registro: `push dword eax`
- Operaciones de pila II
 - Código de operación: **pop**
 - Extrae el operando
 - Utilización del calificador **dword**
 - Ejemplo con acceso a registro: `pop dword eax`
 - Ejemplo con acceso a memoria: `pop dword [_x]`

Sección de código X

- Suma de enteros
 - Código de operación: **add**
 - Realiza la suma de dos operandos y almacena el resultado en el primero
 - Ejemplo con acceso a registros: `add eax, edx`
- Resta de enteros
 - Código de operación: **sub**
 - Realiza la resta de dos operandos y almacena el resultado en el primero
 - Ejemplo con acceso a registros: `sub eax, edx`

Sección de código XI

- Inversión de signo
 - Código de operación: **neg**
 - Invierte el valor de un operando
 - Ejemplo con acceso a registro: `neg eax`
- Multiplicación de enteros
 - Código de operación: **imul**
 - Ejemplo con acceso a registro: `imul ecx`
 - Esta función asume que uno de los operandos está en el registro `eax` y el otro está en el registro incluido en la instrucción
 - El resultado se almacena en `edx:eax` y se puede considerar que el valor almacenado en el registro `eax` como el resultado de la operación

Sección de código XII

- División de enteros
 - Código de operación: **idiv**
 - Ejemplo con acceso a registro: `idiv ecx`
 - Esta función asume que el numerador está almacenado en los registros `edx:eax`
 - El denominador está almacenado en el registro incluido en la instrucción
 - El resultado de la división de enteros se almacena en el registro `eax`
 - El resto de la división de enteros se almacena en el registro `edx`
- Para especificar que el numerador está almacenado en los registros `edx:eax` y realizar correctamente la división debemos realizar las siguientes acciones
 - Cargar el numerador en `eax`
 - Extender el numerador a los registros `edx:eax` utilizando la instrucción NASM `cdq`
 - Cargar el denominador en `ecx`
 - Realizar la división: `idiv ecx`

Sección de código XIII

- Conjunción
 - Código de operación: **and**
 - Realiza la conjunción bit a bit de los dos operandos dados y almacena el resultado en el primero. Internamente en el compilador, se representa el valor booleano *true* como un 1 y el valor booleano *false* como un 0
 - Ejemplo con acceso a registros: `and eax, edx`
- Disyunción
 - Código de operación: **or**
 - Realiza la disyunción bit a bit para los dos operandos dados y almacena el resultado en el primero. It will perform a disjunction bit by bit for the two given operands and will store the result in the first one. Internamente en el compilador, se representa el valor booleano *true* como un 1 y el valor booleano *false* como un 0
 - Ejemplo con acceso a registros: `or eax, edx`

Sección de código XIV

- Negación lógica
 - El código de operación **not** no es válido porque no realiza la negación booleana (realiza el complemento al uno)
 - Se implementará la operación de negación lógica con un código específico (antes de verlo, se debe conocer el código de operación para la comparación de enteros, **cmp**, y los saltos)
- Comparación de enteros
 - Código de operación: **cmp**
 - Realiza la resta de los operandos y dependiendo del resultado actualiza las flags del sistema (el resultado de la resta no se almacena)
 - Ejemplo con acceso a registros: `cmp eax, edx`

Sección de código XV

- Saltos condicionales
 - Las instrucciones de saltos condicionales se incluyen inmediatamente después de la instrucción `cmp` porque se ejecutarán dependiendo del estado de las flags del sistema
 - Las instrucciones de salto proporcionadas, para las comparaciones de enteros con signo, son las siguientes:
 - `je <label>` jumps to <label> if `eax == edx` (also **jz**)
 - `jne <label>` jumps to <label> if `eax != edx` (also **jnz**)
 - `jle <label>` jumps to <label> if `eax <= edx` (also **jng**)
 - `jge <label>` jumps to <label> if `eax >= edx` (also **jnl**)
 - `j1 <label>` jumps to <label> if `eax < edx` (also **jnge**)
 - `jg <label>` jumps to <label> if `eax > edx` (also **jnle**)

Creación del ejecutable

- Para generar el ejecutable a partir de un fichero fuente de código NASM se utilizan los siguientes comandos:
 - `nasm -g -o [<object file (.o)>] -f elf <source file (.asm)>`
 - `gcc -o <executable file> <object file 1> <object file 2> ...`
- Por ejemplo, si se escribe un programa NASM con el nombre `ej1.asm` y se utiliza la librería `alfalib.o`, ejecutaremos estas instrucciones para generar el ejecutable:
 - `nasm -g -o ej1.o -f elf ej1.asm`
 - `gcc -o ej1 ej1.o alfalib.o`

Problemas posibles de compatibilidad I

- Mensajes de error como los siguientes:
 - `skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/6/libgcc.a
when searching for lgcc`
 - `cannot find lgcc`
- Solución: instalar librerías del sistema para compatibilidad de 32 bits:
 - `sudo apt-get install g++-multilib libc6-dev-i386`

Problemas posibles de compatibilidad II

- Mensajes de error de este tipo:
- `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 alfablib.o`

Resumen para obtener el fichero ejecutable

- Compilar las funciones creadas en el fichero generacion.c junto al programa del ejemplo (ej1.c)
- `gcc -Wall -g -c ej1_asm generacion.c ej1.c`
- Ejecutar este programa ej1_asm para obtener el fichero .asm (como argumento)
- `./ej1_asm ej1.asm`
- Compilar el fichero nasm obtenido
- `nasm -g -o ej1.o -f elf32 ej1.asm`
- Compilar con gcc el fichero objeto generado en el paso anterior
- `gcc -Wall -g -m32 -o ej1 ej1.o alfablib.o`
- Recomendamos la utilización de **Valgrind** para comprobar posibles errores