Índice

- Introducción
- Funciones para la generación de estructuras fijas de programas NASM
- Funciones para el control de errores en tiempo de ejecución
- Generación de código para las operaciones aritméticas

Introducción

- En este material, se adelanta parte de la documentación que se entregará de forma completa en la práctica de Generación de Código.
- Se trata de la documentación necesaria para realizar los ejercicios que se le reclaman al estudiante y que persiguen un doble objetivo:
  - Familiarizarse con el lenguaje ensamblador NASM.
  - Utilizar este lenguaje como lenguaje objeto de un compilador.
- No se trata de utilizar NASM de la forma "tradicional" como se puede utilizar en las prácticas de otras asignaturas de programación en ensamblador, sino de utilizarlo como lenguaje objetivo. Esto implicará el uso de un número mucho menor de instrucciones de entre todas las posibles en NASM. El estudiante debe aprender una serie de sentencias básicas, no se trata de buscar el máximo nivel de complejidad del lenguaje.
- Dicho de otra manera, el lenguaje objetivo NASM no es sino un paso más del compilador, y se ha elegido este lenguaje por tener algunas ventajas (compiladores de libre distribución en muchos sistemas y plataformas, fácil legibilidad…) como podría haberse elegido cualquier otro. Si bien se considera necesario que el estudiante que va a desarrollar un compilador tenga agilidad y conocimiento de la parte final de su proceso de compilación.

Funciones para la generación de estructuras fijas de programas NASM

- Inicialización de segmentos y Finalización de Programas NASM:
  - Inicializar segmentos (debe subdividirse en varias funciones): Se trata de una serie de rutinas que escribirá la parte inicial de los segmentos de un programa NASM. Estos son:
    - Segmento "bss": Esta rutina escribirá la línea "segment .bss". Más adelante también se registrarán las variables globales contenidas en la tabla de símbolos. Se debe guardar en una variable el puntero de la pila al inicio del programa. Para ello se debe definir una variable en este segmento con una instrucción tal como: "pila resd 1".
      - En esta práctica únicamente se escribirán estas líneas dejando un comentario "//TO-DO" para incluir las variables de la tabla de símbolos en prácticas posteriores.
    - Segmento "data": Además de escribir la instrucción "segment .data" escribirá los mensajes fijos de error que se mostrarán en caso de error.
    - Sección de código (Segmento "text"): Se incluirá, además de la línea "segment .text" y la llamada "global main", las de habilitar con "extern" las llamadas a funciones en "alfalib.o".
      - Tras la sección de código se mostrarían las funciones, fuera del alcance de esta práctica.
    - Etiqueta "main": Además de escribir la línea "main:", debe guardar el valor del puntero de pila al inicio, para restaurarlo al final del programa. (con una línea como "mov dword[pila], esp" o similar).
  - Finalizar: La finalización de programas debe contemplar la posibilidad que haya habido algún error controlado de ejecución. Se explica detalladamente en la sección "Funciones para el control de errores en tiempo de ejecución" de este material

- Según lo visto anteriormente, el contenido del fichero ensamblador generado sería el siguiente:

```
segment .bss
pila resd 1
; declaración de variables
;
segment .data
; declaración de variables inicializadas (ver transparencias siguientes)
;
mensaje_2   db   "División por cero", 0
segment .text
global main
extern scan_int,  scan_boolean
extern print_int,  print_boolean, print_string, print_blank, print_endofline
;
; código correspondiente a las funciones
;
main:
mov dword [pila], esp
```

Alcance Práctica Introducción NASM

Funciones para el control de errores en tiempo de ejecución

- La gestión de errores en tiempo de ejecución requiere que en el proceso de compilación se genere el código ensamblador que comprueba dichos errores.

- El código ensamblador asociado a un control de error puede funcionar de la siguiente manera:
  - comprueba la condición de error
  - si se cumple la condición de error, se transfiere el flujo de ejecución a una etiqueta en la que se realizan dos acciones:
    - se informa al usuario del error ocurrido
    - se salta de manera incondicional al final del programa

- Para poder informar de los errores, se tienen que declarar los mensajes en el segmento de datos inicializados .data, por ejemplo:

```
segment   .data
.......
mensaje_2   db   "División por cero" , 0
..........
```

Funciones para el control de errores en tiempo de ejecución

- En la parte final del programa se debe dedicar una zona para las etiquetas en las que se informa al usuario de un error de ejecución. La estructura de esta zona sería similar a la siguiente:

```
error_1:  push dword mensaje_1
          call print_string
          add esp, 4
          mov dword esp, [pila]
          jmp near fin
error_2:  push dword mensaje_2
          call print_string
          add esp, 4
          mov dword esp, [pila]
          jmp near fin
fin:      ret
```

NOTA: la estructura del código ensamblador podría ser diferente.

IMPORTANTE: Restaurar el valor de la pila antes de llegar a fin en caso de pasar por las sentencias de error con “mov dword esp, [pila]"

Generación de código para las operaciones aritméticas

- El código NASM que el compilador tiene que generar para la ejecución de una operación aritmética sigue la siguiente secuencia:
  - Antes de realizar cualquier operación aritmética, los operandos están en la pila (esto se recomienda por el diseño que se explicará más adelante en la etapa de generación de código).
  - Se extraen los operandos de la pila a los registros.
  - Se realiza la operación aritmética con el código de operación adecuado.
  - Se deposita en la pila el resultado de la operación.
- Los **registros** que se utilizan para aritmética entera, son los de **propósito general**: eax, edx, etc.
- Los códigos de operación son, entre otros: add, imul, sub, etc. Y se detallarán más adelante.

Generación de código para la operación suma

resultado = $operando_1$ **'+'** $operando_2$

GENERACIÓN DE CÓDIGO

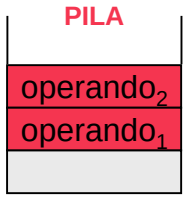
```
; cargar el segundo operando en edx
pop dword edx

; cargar el primer operando en eax
pop dword eax

; realizar la suma y dejar el resultado en eax
add eax, edx

; apilar el resultado
push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA OPERACIÓN

DESPUÉS DE LA OPERACIÓN

Generación de código para la operación suma

- Para la suma de expresiones de tipo entero, la función tendría el siguiente aspecto:

```
void gc_suma_enteros(FILE* f)
{
        fprintf(f, "; cargar el segundo operando en edx\n");
        fprintf(f, "pop dword edx\n");
        fprintf(f, "; cargar el primer operando en eax\n");
        fprintf(f,"pop dword eax\n");
        fprintf(f, "; realizar la suma y dejar el resultado en eax \n");
        fprintf(f, "add eax,edx\n);
        fprintf(f, "; apilar el resultado\n");
        fprintf(f, "push dword eax\n");
        return;
}
```

Generación de código para el resto de las operaciones aritméticas binarias

- El código ensamblador correspondiente a las otras operaciones de resta, multiplicación y división anteriores es similar al de la regla de la suma visto anteriormente. La única diferencia es la instrucción ensamblador que se utilice y las posibles peculiaridades de la misma.

- Para la **resta** se puede utilizar la instrucción ensamblador siguiente:

```
sub eax,edx
```

que resta al contenido del registro `eax` el contenido del registro `edx` y deja el resultado en `eax`.

Generación de código para el resto de las operaciones aritméticas binarias

- Para la **división** se puede utilizar la instrucción ensamblador de división entera **`idiv`**. 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`

- Para que el dividendo esté en la composición de registros `edx:eax` y realizar correctamente la división se tiene que hacer:
  - Cargar el divisor en `ecx`
  - Cargar el dividendo en `eax`
  - Extender el dividendo a la composición de registros `edx:eax` (`cdq`)
  - Realizar la división (`idiv ecx`)

- **Se debe controlar la división por cero según se explicó en el apartado "Control de errores en tiempo de ejecución"**

Generación de código para el resto de las operaciones aritméticas binarias

- Para el **producto** se puede utilizar la instrucción ensamblador de multiplicación entera **`imul`**. Esta instrucción asume que uno de los operandos está en el registro `eax`, y el otro operando está en el registro que aparece en la instrucción. El resultado se carga en `edx:eax`, y consideraremos el valor de `eax` como resultado de la operación.