Práctica 0: Conceptos de Generación de Código. El lenguaje NASM.

Fecha de entrega: Semana del 9 de Octubre, antes del inicio de la clase.

Objetivo de la práctica:

El objetivo de la práctica es introducir la etapa de Generación de Código y escribir una librería de funciones C para la generación de código NASM. Esto ayudará al estudiante a familiarizarse con el lenguaje en el que se generará el código objeto en el compilador y agilizará la solución de la generación de código del compilador en la fase final del desarrollo del mismo.

Desarrollo de la práctica:

0. Decisiones de diseño para la generación de código

Se resumen a continuación las decisiones de diseño más importantes que tu profesor te explicará a lo largo del laboratorio respecto a la generación de código. En general son sugerencias que tú puedes seguir; si adoptaras otras decisiones debes consensuarlo con el profesor y asegurarte de que se cumplen las condiciones que en cada momento se describan tanto de funcionalidad como de interfaz (nombre de funciones, tipo, número y posición de argumentos, etc.)

Tamaño y tipo de datos en memoria

La aritmética de nuestro compilador se reducirá siempre al tipo de dato entero y a tamaño de 32 bits. Eso implica que el conjunto de registros sea el extendido, así por ejemplo el acumulador es `eax`, el puntero de pila es `esp`, etc...

Los tipos de datos básicos de nuestro lenguaje de programación son entero y booleano (que será internamente manipulado como entero).

Pila como almacenamiento auxiliar

Cuando se necesite espacio auxiliar (por ejemplo para gestionar las expresiones aritmético-lógicas) se utilizará la pila del sistema.

La manera en la que se gestionarán las **expresiones aritmético-lógicas** será la siguiente:

- Se supondrá que los operandos están siempre en la cima de la pila (dos posiciones contiguas en el caso de operaciones binarias y una en monádicas).
- Por lo tanto, el proceso de cualquier operación implicará:
    - Introducir, cuando se disponga de ellos, los operandos en la pila.
    - Cuando se sepa qué operación realizar:
        - Extraer de la cima de la pila los operandos guardándolos en registros,
        - realizar la operación y
        - copiar el resultado desde el registro que lo tenga a la pila.
    - Cada operación tiene sus peculiaridades y sus códigos de operación NASM que debes consultar en la documentación que se te proporciona al respecto.

Este esquema también determina cómo gestionar las **asignaciones de valor a las variables de alto nivel**.
Por ejemplo `x=4+y;`

- Como se ha indicado previamente, tras gestionar una expresión, su resultado se deja en la cima de la pila.
- Por lo tanto, dado que la asignación considera que el valor que se asigna a la variable está expresado mediante una expresión aritmético-lógica, en las asignaciones se toma el valor que hay que asignar a la variable de la cima de la pila.

Operaciones de entrada salida
Tu profesor te explicará las funciones contenidas en la librería alfalib.o que se te proporciona como material del laboratorio.
También te explicará cómo se invocan.
Esta librería contiene, entre otras, todas las operaciones de lectura y escritura que puedes necesitar invocar en NASM cuando generes código.
Tu profesor te explicará con detalle cómo se invocan funciones en ensamblador y el convenio de llamada que usamos y que es el utilizado en la librería.

Nombres internos correspondientes a los símbolos de alto nivel
En general se precederá del carácter "_" los nombres de alto nivel para ser convertidos en nombres correctos NASM.
En algunos casos (como la variable para guardar el puntero de pila) se utilizarán "_" adicionales para asegurar que son nombres correctos (en ese caso se te recomienda que utilices un nombre como `__esp`).

Información semántica que mantiene el compilador.
A lo largo del proceso de compilación, se mantiene cierta información que facilita tanto el análisis semántico como la generación de código.
Una de esas informaciones sirve para distinguir si lo que hay en la cima de la pila (cuando se está procesando una expresión aritmético-lógica) es un variable (se suele llamar referencia) o no (no es lo mismo encontrar en el programa fuente un `3` que un `b1`, cuando se introduce este último operando en la pila por primera vez se introduce como (`_b1`) y el compilador debe recordar que es referencia (es decir, que en la pila no está el valor del operando sino una referencia al lugar donde está, en este caso la posición de memoria `_b1`), sin embargo, el operando `3` es ya un valor explícito.

Gestión de las etiquetas
A lo largo de un programa NASM hay múltiples etiquetas necesarias para implementar las estructuras de control de flujo del programa.
Cuando el compilador genere código deberá articular algún mecanismo para poder distinguir unas etiquetas de otras. A lo largo del curso se sugiere el uso de un contador que se incremente cada vez que se utilice una etiqueta y que aparezca explícitamente en ella. Así, por ejemplo, una etiqueta de fin de if podría ser `fin_if_99:`. Si es 99 el contador que le corresponda. La siguiente etiqueta que se necesite (por ejemplo una etiqueta para una rama then) podría ser `then_100:`. Ya que se ha incrementado el contador de etiquetas.

1. Librería de Funciones de Generación de Código.

Partiendo de la información aportada por el profesor y accesible en el curso de *Moodle* de la asignatura, se van a desarrollar una serie de funciones en lenguaje C, que irán generando código NASM. Se generará una librería "generacion.c" y su correspondiente fichero de encabezados "generacion.h" en los que se van a codificar una serie de funciones que resolverán parcialmente la

generación de código. Todas las funciones recibirán como parámetro, entre otros, el fichero en el que se debe escribir.
Más adelante en este curso se trabajará en el proceso completo de generación de código. A continuación se enumera la funcionalidad que tu librería debe cubrir.

- **Inicialización y Finalización de Programas NASM**: Todos los programas generados comenzarán y finalizarán de la misma manera y seguirán la estructura descrita en el material de la asignatura. El inicio y fin de programa implica las siguientes funciones:
    - Inicializar (debe subdividirse en varias funciones): Se trata de una serie de rutinas que escribirán la parte inicial de la estructura de un programa NASM. Esto es:
        - Cabecera de compatibilidad (si se usa).
        - Segmento "bss".
        - Segmento "data".
        - Sección de código (Segmento "text").
            - Tras la sección de código se mostrarían las funciones del programa fuente de partida, la generación de su código está fuera del alcance de esta práctica.
        - Etiqueta "main" y acciones iniciales del programa.

    - Finalizar: La finalización de programas debe contemplar la posibilidad de que haya habido algún error controlado de ejecución.

- **Operaciones Aritméticas y Lógicas**: con la recepción correspondiente de parámetros, según lo explicado en clase, se codificarán las funciones necesarias para la realización de:
    - Suma
    - Cambio de signo
    - Resta
    - Multiplicación
    - División
    - Negación (lógica)
    - O (disyunción lógica)
    - Y (conjunción lógica)

- Funciones **auxiliares para la manipulación de los operandos** de las operaciones aritmético-lógicas.
- Funciones para la **declaración de variables.**
- Funciones de **entrada / salida.**
- Funciones para **asignación de expresiones a variables.**

2 Cabeceras de las funciones pedidas

A continuación se muestran las cabeceras que debes utilizar.

Es imprescindible mantener este convenio de funciones porque es un requisito de diseño que se pueda utilizar la librería con cualquier programa principal que cumpla el convenio.

En cada función tienes un comentario que resume lo que debe hacer. Puedes encontrar con más detalle esta misma información en el material dedicado a la generación de código.

```
#ifndef GENERACION_H
#define GENERACION_H

#include <stdio.h>
```

```
/* Declaraciones de tipos de datos del compilador */
#define ENTERO      0
#define BOOLEANO    1


/* OBSERVACIÓN GENERAL A TODAS LAS FUNCIONES:
   Todas ellas escriben el código NASM a un FILE* proporcionado como primer
argumento.
*/

void escribir_cabecera_compatibilidad(FILE* fpasm);
/*
   Función para imprimir el código NASM necesario para que sea multiplataforma.
*/

void escribir_subseccion_data(FILE* fpasm);
/*
   Declaración (con directiva db) de las variables que contienen el texto de los
mensajes para la identificación de errores en tiempo de ejecución.
   En este punto, al menos, debes ser capaz de detectar la división por 0.
*/

void escribir_cabecera_bss(FILE* fpasm);
/*
   Código para el principio de la sección .bss.
   Con seguridad sabes que deberás reservar una variable entera para guardar el
puntero de pila extendido (esp). Se te sugiere el nombre __esp para esta
variable.
*/

void declarar_variable(FILE* fpasm, char * nombre,  int tipo,  int tamano);
/*
   Para ser invocada en la sección .bss cada vez que se quiera declarar una
variable:
   - El argumento nombre es el de la variable.
   - tipo puede ser ENTERO o BOOLEANO (observa la declaración de las constantes
     del principio del fichero).
   - Esta misma función se invocará cuando en el compilador se declaren
     vectores, por eso se adjunta un argumento final (tamano) que para esta
     primera práctica siempre recibirá el valor 1.
*/

void escribir_segmento_codigo(FILE* fpasm);
/*
   Para escribir el comienzo del segmento .text, básicamente se indica que se
exporta la etiqueta main y que se usarán las funciones declaradas en la librería
alfalib.o
*/

void escribir_inicio_main(FILE* fpasm);
/*
   En este punto se debe escribir, al menos, la etiqueta main y la sentencia que
guarda el puntero de pila en su variable (se recomienda usar __esp).
*/

void escribir_fin(FILE* fpasm);
/*
   Al final del programa se escribe:
   - El código NASM para salir de manera controlada cuando se detecta un error
     en tiempo de ejecución (cada error saltará a una etiqueta situada en esta
     zona en la que se imprimirá el correspondiente mensaje y se saltará a la
     zona de finalización del programa).
   - En el final del programa se debe:
```

```
         ·Restaurar el valor del puntero de pila (a partir de su variable __esp)
         ·Salir del programa (ret).
*/

void escribir_operando(FILE* fpasm, char* nombre, int es_var);
/*
   Función que debe ser invocada cuando se sabe un operando de una operación
aritmético-lógica y se necesita introducirlo en la pila.
   - nombre es la cadena de caracteres del operando tal y como debería aparecer
     en el fuente NASM
   - es_var indica si este operando es una variable (como por ejemplo b1) con un
     1 u otra cosa (como por ejemplo 34) con un 0. Recuerda que en el primer
     caso internamente se representará como _b1 y, sin embargo, en el segundo se
     representará tal y como esté en el argumento (34).
*/

void asignar(FILE* fpasm, char* nombre, int es_referencia);
/*
   - Genera el código para asignar valor a la variable de nombre nombre.
   - Se toma el valor de la cima de la pila.
   - El último argumento es el que indica si lo que hay en la cima de la pila es
     una referencia (1) o ya un valor explícito (0).
*/


/* FUNCIONES ARITMÉTICO-LÓGICAS BINARIAS */
/*
   En todas ellas se realiza la operación como se ha resumido anteriormente:
   - Se extrae de la pila los operandos
   - Se realiza la operación
   - Se guarda el resultado en la pila
   Los dos últimos argumentos indican respectivamente si lo que hay en la pila es
una referencia a un valor o un valor explícito.
   Deben tenerse en cuenta las peculiaridades de cada operación. En este sentido
sí hay que mencionar explícitamente que, en el caso de la división, se debe
controlar si el divisor es "0" y en ese caso se debe saltar a la rutina de error
controlado (restaurando el puntero de pila en ese caso y comprobando en el
retorno que no se produce "Segmentation Fault")
*/
void sumar(FILE* fpasm, int es_referencia_1, int es_referencia_2);
void restar(FILE* fpasm, int es_referencia_1, int es_referencia_2);
void multiplicar(FILE* fpasm, int es_referencia_1, int es_referencia_2);
void dividir(FILE* fpasm, int es_referencia_1, int es_referencia_2);
void o(FILE* fpasm, int es_referencia_1, int es_referencia_2);
void y(FILE* fpasm, int es_referencia_1, int es_referencia_2);

void cambiar_signo(FILE* fpasm, int es_referencia);
/*
   Función aritmética de cambio de signo.
   Es análoga a las binarias, excepto que sólo requiere de un acceso a la pila ya
que sólo usa un operando.
*/

void no(FILE* fpasm, int es_referencia, int cuantos_no);
/*
   Función monádica lógica de negación. No hay un código de operación de la ALU
que realice esta operación por lo que se debe codificar un algoritmo que, si
encuentra en la cima de la pila un 0 deja en la cima un 1 y al contrario.
   El último argumento es el valor de etiqueta que corresponde (sin lugar a
dudas, la implementación del algoritmo requerirá etiquetas). Véase en los
ejemplos de programa principal como puede gestionarse el número de etiquetas
cuantos_no.
*/

/* FUNCIONES DE ESCRITURA Y LECTURA */
```

```
/*
   Se necesita saber el tipo de datos que se va a procesar (ENTERO o BOOLEANO) ya
que hay diferentes funciones de librería para la lectura (idem. escritura) de
cada tipo.
   Se deben insertar en la pila los argumentos necesarios, realizar la llamada
(call) a la función de librería correspondiente y limpiar la pila.
*/
void leer(FILE* fpasm, char* nombre, int tipo);
void escribir(FILE* fpasm, int es_referencia, int tipo);

#endif
```

3. Ejemplos.

A continuación se muestra un ejemplo de un programa principal que realiza lo mismo que el siguiente programa de alto nivel.

Observa que el programa principal toma como argumento al ser invocado el nombre del fichero que contendrá la salida.
El programa también declara variables de tipo entero y realiza operaciones aritméticas con ellas antes de mostrar resultados.

3.1 Ejemplo 1

Código fuente alto nivel para ejemplo 1

```
main
{
	int x;
	int y;
	int z;

	x=8;
	scanf y;
	z = x + y;
	printf z;
}
```

Posible programa princpial para ejemplo 1

```
#include <stdio.h>
#include "generacion.h"

int main (int argc, char** argv)
{
	FILE * salida;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}


	salida = fopen(argv[1],"w");


	escribir_cabecera_compatibilidad(salida);
	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
```

```
	declarar_variable(salida, "x", ENTERO, 1);
	declarar_variable(salida, "y", ENTERO, 1);
	declarar_variable(salida, "z", ENTERO, 1);


	escribir_segmento_codigo(salida);
	escribir_inicio_main(salida);

	/* x=8; */

	escribir_operando(salida,"8",0);
	asignar(salida,"x",0);

	/* scanf y; */

	leer(salida,"y",ENTERO);

	/* z = x + y */

	escribir_operando(salida,"y",1);
	escribir_operando(salida,"x",1);
	sumar(salida,1,1);

	asignar(salida,"z",0);


	/* printf z; */

	escribir_operando(salida,"z",1);
	escribir(salida,1,ENTERO);



	escribir_fin(salida);

	fclose(salida);
	return 0;


}
```

Salida esperada

Supondremos que el ejecutable se llama `ej1`.

```
> ej1   /* La entrada es -9 y la salida -1*/
-9
-1

> ej1 /* La entrada es 10 y la salida 18*/
10
18
```

3.2 Ejemplo 2

Código fuente alto nivel para ejemplo 2

El siguiente programa manipula una variable de tipo booleano y realiza alguna operación lógica antes de mostrar resultados.

```
main
{
	boolean b1;

	scanf b1;
	 printf !b1;
	printf !!b1;

}
```

Posible programa principal

```
#include <stdio.h>
#include "generacion.h"

int main (int argc, char** argv)
{
	FILE * salida;
	int num_nots = 0;

	if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}


	salida = fopen(argv[1],"w");


	escribir_cabecera_compatibilidad(salida);
	escribir_subseccion_data(salida);
	escribir_cabecera_bss(salida);
	declarar_variable(salida, "b1", BOOLEANO, 1);


	escribir_segmento_codigo(salida);
	escribir_inicio_main(salida);

	leer(salida,"b1",BOOLEANO);
	escribir_operando(salida,"b1",1);

	no(salida,1,num_nots++);
	escribir(salida,0,BOOLEANO);


	escribir_operando(salida,"b1",1);
	no(salida,1,num_nots++);
	no(salida,0,num_nots++);
	escribir(salida,0,BOOLEANO);

	escribir_fin(salida);

	fclose(salida);
	return 0;

}
```

Salida esperada

Supondremos que el ejecutable se llama `ej2`.

```
> ej2   /* La entrada es 0 y la salida true false*/
0
true
false

> ej2 /* La entrada es 1 y la salida false true */
1
false
true
```

3.3 Ejemplo 3

Código fuente alto nivel

Este último programa maneja tanto datos enteros como booleanos y realiza algunas operaciones con ellos antes de mostrar resultados.

```
main
{
       int x;
       int y;
       int z;
       boolean b1;
       int j;

       scanf x;
       scanf z;
       scanf b1;
       printf !b1;
       j = - x;
       printf j;
       printf x+z;
       printf z;

}
```

Posible programa principal

```
#include <stdio.h>
#include "generacion.h"

int main (int argc, char** argv)
{
       FILE * salida;
       int cuantos_no = 0;


       if (argc != 2) {fprintf (stdout, "ERROR POCOS ARGUMENTOS\n"); return -1;}


       salida = fopen(argv[1],"w");


       escribir_cabecera_compatibilidad(salida);
       escribir_subseccion_data(salida);
       escribir_cabecera_bss(salida);
```

```
        declarar_variable(salida, "x", ENTERO, 1);
        declarar_variable(salida, "y", ENTERO, 1);
        declarar_variable(salida, "z", ENTERO, 1);
        declarar_variable(salida, "b1", BOOLEANO, 1);
        declarar_variable(salida, "j", ENTERO, 1);


        escribir_segmento_codigo(salida);
        escribir_inicio_main(salida);

        leer(salida,"x",ENTERO);
        leer(salida,"z",ENTERO);
        leer(salida,"b1",BOOLEANO);

        escribir_operando(salida,"b1",1);
        no(salida,1,cuantos_no++);
        escribir(salida,0,BOOLEANO);


        escribir_operando(salida,"x",1);
        cambiar_signo(salida,1);
        asignar(salida, "j",0);
        escribir_operando(salida,"j",1);
        escribir(salida,1,ENTERO);

        escribir_operando(salida,"x",1);
        escribir_operando(salida,"z",1);

        sumar(salida, 1, 1);
        escribir(salida,0,ENTERO);
        escribir_operando(salida,"z",1);
        escribir(salida,1,ENTERO);

        escribir_fin(salida);

        fclose(salida);
        return 0;


}
```

Salida esperada

Supondremos que el ejecutable se llama `ej3`.

```
> ej3   /* La entrada es -3 3 0  y la salida true 3 0 3*/
-3
3
0
true
3
0
3

> ej3 /* La entrada es 10 -9 1 y la salida false -10 1 -9 */
10
-9
1
false
-10
1
```

```
-9
```

Entrega de la práctica:

Se entregará a través de *Moodle* un único fichero comprimido (*.zip*) que deberá cumplir los siguientes requisitos:

- Deberá contener todos los fuentes (ficheros *.h* y *.c*) necesarios para resolver el enunciado propuesto.
- Deberá contener un fichero *Makefile* compatible con la herramienta *make* que para el objetivo ***all*** genere la librería de nombre ***generacion_codigo.o***
- El nombre del fichero *.zip* seguirá el siguiente formato***:***

  Apellido1Estudiante1_Apellido2Estudiante2_ NASM.zip

  Los apellidos de los integrantes de la pareja aparecerán en orden alfabético. Los nombres no deben contener espacios, acentos, ni eñes.

- De forma explícita se tendrá en cuenta en la evaluación que se mantengan las cabeceras de las funciones, nombres de ficheros y módulos y cualquier otra consideración que tenga que ver con el interfaz de la librería de tal forma que, dependiendo de la gravedad de los errores, las prácticas alcanzarán puntuaciones en cualquier caso inferiores a la máxima.

Se recuerda al alumno que **las prácticas son incrementales por lo que una práctica aprobada pudiera conllevar errores importantes en las siguientes fases. Es responsabilidad del alumno subsanar totalmente los errores en cada fase, dando correcto cumplimiento a los requerimientos de los enunciados.**