Apuntes

Procesadores de Lenguajes I

José Tomás Tocino García

Junio de 2013

Índice

1. Estructura de un proceso

1.1. Datos

En un programa existen dos tipos de datos. Los **datos estáticos** agrupan las variables globales y las variables estáticas de las funciones. El tamaño de los datos estáticos se debe conocer en tiempo de compilación.

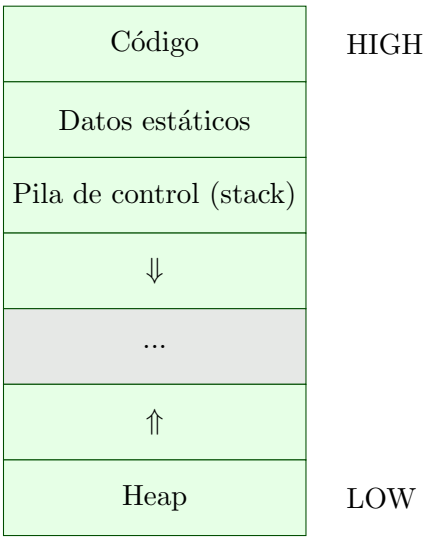
Los **datos dinámicos** son el segundo tipo. Agrupan a las variables **locales** y a las variables **dinámicas**.

Las variables locales (entre las que también se encuentran los argumentos de una función) tienen un ciclo de vida ligado al marco de ejecución de la función, por lo que cuando concluyen las instrucciones, el valor de las variables se pierde. Las variables locales se alojan en la `stack`.

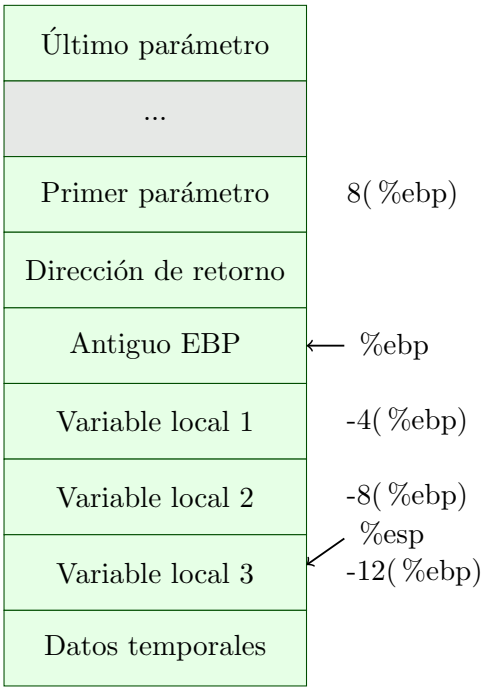
Las variables dinámicas se almacenan en el montículo o `heap`, y su tiempo de vida no está contenido en el del marco de activación, sino que deben ser manualmente gestionadas mediante elementos como `new` y `free`.

1.2. Memoria

Cuando se lanza un programa, el sistema operativo le asigna un espacio de memoria, que se organiza de la siguiente manera (de mayor a menor dirección de memoria)



En la pila de control se almacenan los **marcos de activación**, que encapsulan la información necesaria para gestionar la llamada a una subrutina. Los marcos suelen tener esta estructura:



Para gestionar la memoria en general y la pila en particular, el sistema cuenta con dos registros que mantienen direcciones de memoria y que se actualizan cuando se llevan a cabo llamadas a subrutinas:

- El registro **ESP** (Extended Stack Pointer) apunta al **tope de la pila**. Dado que la pila crece hacia direcciones de memoria más bajas, el ESP tendrá un menor valor cuanto más llena esté la pila, y viceversa.
- El registro **EBP** (Extended Base Pointer) apunta a la **base de la pila**, por lo que sirve como punto de referencia para el acceso a los parámetros y a las variables locales de la función.

Cuando un procedimiento necesita guardar información en la pila, se hace una operación `push` y se decrementa el registro `ESP`. Dado que ambos registros apuntan hacia direcciones **ocupadas**, el valor de ESP será la posición del último elemento que se ha introducido en la pila.

De igual modo, el ESP se recoloca al inicio de cada función de forma que se guarde espacio para las variables locales bajo el `EBP`, es por ello que la primera variable local se guardará en `-4(%ebp)`, la siguiente en `-8(%ebp)` y así sucesivamente.

En el siguiente ejemplo, en ambos casos se imprime lo mismo, ya que el registro `ESP` apunta al tope de la pila, que es la última variable local:

```
subl    $4, %esp        # Guardamos espacio para una variable (4 bytes)
movl    $77, -4(%ebp)   # Metemos el valor 77

pushl   -4(%ebp)
pushl   $.LC1
call    printf          # Se imprime 77
addl    $8, %esp

pushl   (%esp)
pushl   $.LC1
call    printf          # De nuevo imprime 77
addl    $8, %esp
```

2. Ensamblador

2.1. Estructura general

Un fichero de ensamblador se divide en distintas **secciones**, que en general se asocian a los diferentes segmentos del espacio de memoria del programa.

La directiva que da inicio al fichero de ensamblador es:

```
    .file   "main.c"
```

2.2. Registros de memoria

El procesador cuenta con una serie de registros de memoria de acceso muy rápido y poca capacidad. Estos registros son de 32bits = 8 bytes. En la arquitectura Intel se considera que ocupan una doble palabra (`doubleword`). Es posible direccionar los últimos 16 bits de cada registro, así como los dos bloques de 8 bits que dividen esa parte.

Los más habituales son:

EAX *Accumulator register*, usado en operaciones aritméticas.

EBX *Base register*, sirve de índice para direccionamientos.

ECX *Counter register*, usado en operaciones de bucles.

EDX *Data register*, usado en operaciones de datos (como cuando se necesita una doble palabra).

EBP *Base pointer*, apunta a la base de la pila del marco actual.

ESP *Stack pointer*, apunta al tope de la pila del marco actual.

2.3. Segmento de datos (`.data`)

El segmento de datos es una porción de la memoria asignada a un programa, que contiene las **variables globales** y las **variables estáticas** inicializadas con un valor. El tamaño de este segmento se determina en tiempo de compilación, teniendo en cuenta los valores que asigne el programador.

En este ejemplo, ambas variables se almacenarán en el segmento de datos:

```
// global
int i = 10;

int fun () {
  // static
  static int j = 10;
}
```

En ensamblador, el segmento de datos se indica mediante la directiva `.data`, tras la cual vendrán cada una de las variables señaladas con una **etiqueta**, su **tipo** y su **valor**.

```
    .data
x:
    .long  4
y:
    .byte 10
s:
    .string "pepito\n"
```

2.3.1. Datos no inicializados

Los datos **globales no inicializados** se tratan de forma distinta a los previamente comentados. Dado que no se les da valor en su declaración, son automáticamente **inicializados a cero** por el compilador. En ensamblador, las variables no inicializadas se colocan en una sección del segmento de datos denominada `.bss`, aunque los compiladores evitan usar la sección .bss y usan la directiva `.comm`, que recibe el nombre de la variable, el tamaño y opcionalmente la alineación. Una declaración global como:

```
int x;
```

produciría el siguiente código:

```
    .comm   x,4,4
```

Si se tratara de un array de 2 elementos, el código generado sería:

```
    .comm   x,8,4
```

2.4. Segmento de instrucciones (`.text`)

El segmento de instrucciones, que viene indicado por la directiva `.text`, incluye las instrucciones de las diferentes subrutinas del programa así como de la función principal. La rutina principal suele indicarse con la etiqueta `_start` y llamarse `main`. Así, el programa más pequeño que puede compilarse es el siguiente (generado con `gcc`):

```
    .file   "main.c"

    .text
    .globl  main
    .type   main, @function
main:
    ret
    .size   main, .-main
```

2.5. Funciones

A la hora de trabajar con funciones, hay que prestar atención a las instrucciones de la rutina *"que llama"* y a las instrucciones de la rutina *"llamada"*.

2.5.1. Código de llamada a función

Para llamar a una función es necesario **meter en la pila** los argumentos de la función, mediante sucesivas llamadas a la instrucción `pushl`. Una vez introducidos todos los parámetros, se usará la instrucción `call` para realizar la llamada.

Algo importante es que los argumentos se introducen en la pila **en orden inverso**, para que puedan luego sacarse en el orden correcto.

```
# Equivalente a fun(4,2)
    pushl   $2
    pushl   $4
    call    fun
```

2.5.2. Código de declaración de función

En ensamblador, las funciones son objetos globales de tipo `@function`, y tienen una etiqueta con su nombre, lo cual se define de la siguiente forma (para una supuesta función llamada *fun*):

```
    .globl  fun
    .type   fun, @function
fun:
```

Tras la etiqueta, vienen las instrucciones de la función, que comienzan con el **prólogo**. Éste suele ser siempre el mismo, y lleva a cabo las siguientes operaciones:

- Mete el antiguo registro EBP en la pila, de forma que pueda recuperarse después. Esto, a su vez, actualiza el puntero ESP
- Asigna al EBP (al puntero base) el valor de ESP (el tope de la pila), de forma que el nuevo marco de activación se creará encima del antiguo marco que se encuentra en la pila.
- Se desplaza el ESP, dependiendo del número de variables locales.

En código ensamblador, esto se traduce en:

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp  # Si hubiera 4 variables locales ( 4 * 4 = 16 )
```

Posteriormente, irán las **instrucciones propias** de la función. Dentro de la función, es importante tener en cuenta varias cosas:

- Los **argumentos** están alojados en la pila por encima del EBP y de la dirección de retorno (dos posiciones), por lo que el primer parámetro estará en `8(%ebp)`, el segundo en `12(%ebp)` y así sucesivamente.
- Las **variables locales** están alojadas por debajo del EBP, esto es, en direcciones de memoria **menores**, por lo que la primera variable se encontrará en `-4(%ebp)`, la segunda estará en `-8(%ebp)`, etcétera.

Por último, el **epílogo** de las funciones suele ser también muy común, realizando lo siguiente:

- Si la función debe devolver algún valor, éste debe colocarse en un registro, como en `EAX`.
- Se restaura el valor antiguo del `ESP`, que se encuentra en `EBP`, esto es, se hace `ESP = EBP`.
- Se restaura el valor antiguo del `EBP`, que se encontraba en la pila. Como se ha restaurado el valor del ESP, éste apuntará justo a la posición donde se guardó el antiguo EBP.
- Se devuelve el **control de la ejecución** a la función llamadora, cuya dirección se encuentra justo en el tope de la pila (recordar *estructura del marco de activación*).

En ensamblador, esto se haría de la siguiente manera:

```
movl  %ebp, %esp
popl  %ebp
ret
```

La instrucción `leave` es equivalente a las dos primeras instrucciones, por lo que el epílogo también se puede escribir así:

```
leave
ret
```

Como última línea se suele indicar el tamaño de la función mediante este código:

```
.size   fun, .-fun
```

2.5.3. Código tras la llamada a la función

Tras la llamada a la función, la función *llamadora* debe restaurar el registro `ESP` a su valor original, el que tenía antes de que se incluyesen en la pila los argumentos de la función. Por tanto, se moverá hacia arriba el `ESP` (es decir, se le sumará) 4 × el número de argumentos de la función. En nuestro ejemplo había dos argumentos, por lo que se hará:

```
    addl $8, %esp
```

Si la función devolvía algo, el valor de retorno se encontrará en el registro `EAX` para su uso en sucesivas instrucciones.

2.5.4. Ejemplo completo de función

Este es un ejemplo completo de definición y llamada a una función, con un argumento, en ensamblador.

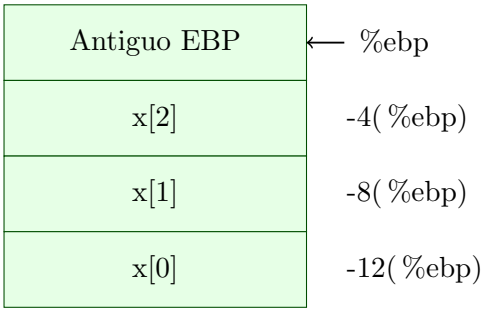
```
    .text                   # Comienza la sección de instrucciones

    .globl  fun             # Cabecera de la declaración
    .type   fun, @function
fun:                        # Comienzan las instrucciones
    pushl   %ebp            # Epílogo
    movl    %esp, %ebp

    pushl   8(%ebp)         # La función llama a printf con el primer argumento (8(%ebp))
    pushl   $CADENA
    call    printf

    leave                   # Comienza el epílogo
    ret

    .size   fun, .-fun      # Pie de la función

    .globl  main            # Cabecera de la función main
    .type   main, @function
main:
    pushl   %ebp            # Prólogo
    movl    %esp, %ebp

    pushl   $5              # Llamada a la función, primer argumento a la pila
    call    fun             # Llamada
    addl    $4, %esp        # Liberación de la memoria ocupada por los argumentos

    leave                   # Epílogo
    ret

    .size   main, .-main    # Pie de la función
```

> **Importante:** Todas las operaciones, incluso las de asignación, deben devolver el valor en el registro EAX. En el caso de las asignaciones, el valor que se devuelve es aquél que se asigna a la posición de memoria.

2.6. Arrays

Los vectores o *arrays* tienen un tratamiento especial. A la hora de reservar memoria, un array ocupará tantos bytes como los elementos que la componen: si tratamos con un array de 3 enteros, ocupará $3 \times 4 = 12$ bytes.

En **memoria**, los arrays se colocan de menor a mayor posición de memoria, por lo que si tenemos un array local de tres enteros, su direccionamiento será:



Para direccionar un array en ensamblador, primero ponemos el índice en el registro `EBX`. Para el caso de las variables locales, tomaremos la dirección base (en la que se encuentra `x[0]`), y usaremos esta sintaxis:

```
# Suponiendo que la base sea -12(%ebp)
-12(%ebp, %ebx, 4)
```

Para el caso de las variables globales, la sintaxis es:

```
# Suponiendo una global 'x'
x(, %ebx, 4)
```

2.7. Asignación

De manera general, la forma habitual de hacer una asignación es colocar el valor en un registro y moverlo a una dirección de memoria, que suele estar contenida en otro registro. El símil en C sería el siguiente:

```
int * direccion;    // Suele estar en EDX
int valor;          // Suele estar en EAX

*direccion = valor;
```

Así, en ensamblador el primer paso sería ejecutar el código que calcula la **dirección de destino**, que se almacenaría en el registro `EAX`. Este valor se metería en la pila, y posteriormente se ejecutaría el código encargado de **calcular el valor a asignar**, que también se escribiría en `EAX` (por ello la dirección se mete en la pila, para evitar perder el valor si se sobreescribe EAX). En este punto, se **recupera el valor de la dirección** de destino, copiándolo por ejemplo al registro `EDX`. El último paso es hacer la copia del valor a la dirección de memoria cuya posición está escrita en EDX.

En ensamblador:

```
# Generate memory address and store it in EAX
# ...

# Save the address in the stack for later use
pushl %eax

# Generate value to store, and place it in EAX
```

```
# ...

# Restore previously calculated address and store in EDX
popl %edx

# Make the actual copy
movl %eax, (%edx)
```

2.7.1. Asignación con operación

En las operaciones de asignación con operación (que llevan a cabo los operadores `+=`, `-=`, `*=` y `/=`), el funcionamiento es similar a la asignación tradicional, con la diferencia de que antes de hacer la asignación es necesario calcular el valor previo del destino y realizar la operación.

Suma En la suma, en lugar de hacer una copia directa del valor a la posición de memoria, se hace una suma.

```
# Restore the destination address
popl %edx

# Add the value to the one already in the destination address
addl %eax, (%edx)

# Make sure the final value is in EAX
movl (%edx), %eax
```

Resta Es básicamente igual que la suma pero usando la operación de resta.

```
# Restore the destination address
popl %edx

# Subtract the value to the one already in the destination address
subl %eax, (%edx)

# Make sure the final value is in EAX
movl (%edx), %eax
```

Producto En el caso del producto, el orden es distinto, ya que `imull` no acepta un direccionamiento como segundo parámetro:

```
# Restore the destination address
popl %edx

# Multiply the contents of the destination with EAX
imul (%edx), %eax

# Move the result back to the destination
movl %eax, (%edx)
```

División El caso de la división es algo distinto. La instrucción *idiv* utiliza como entrada (para el dividendo) el par `EDX:EAX` (una quad-word), por lo que es necesario utilizar la instrucción `cdq` (*convert double to quad*) para hacer la expansión.

```
# Restore the destination address
popl %ecx

# Move the divisor to EBX
movl %eax, %ebx

# Move the dividend to EAX
movl (%ecx), %eax

# Convert to quad word
cdq

# To the division
idiv %ebx

# Save the value in the original position
movl %eax, (%ecx)
```

2.8. Operaciones aritméticas

Los operadores aritméticos funcionan de manera similar a los operadores de operación con asignación, generando el código ensamblador de cada operando y realizando la operación final.

Suma

```
# izq -> generar();
pushl %eax

# derecha -> generar
popl %ebx
addl %ebx, %eax
```

Resta En este caso hay que tener en cuenta el orden de los factores, de forma que en `EAX` esté el factor de la izquierda y en `EBX` el factor de la derecha.

```
# izq -> generar();
# ...
pushl %eax

# derecha -> generar
# ...
movl %eax, %ebx
popl %eax
subl %ebx, %eax
```

Producto El producto es igual que la suma

```
# izq->generar()
pushl %eax

# der->generar()
popl %ebx
imul %ebx, %eax
```

División En la división ocurre lo mismo que en la resta, hay que tener en cuenta el orden de los factores. Y hay que convertir de *double word* a *quad word* con `CDQ`.

```
# izq->generar()
pushl %eax

# der->generar()
movl %eax, %ebx
popl %eax
cdq
idivl %ebx
```

2.9. Operadores lógicos

Los operadores lógicos funcionan de manera muy similar a los operadores aritméticos:

AND La operación `AND` se lleva a cabo mediante la instrucción `ANDL`:

```
# izq->generar()
pushl %eax

# der->generar()
popl %ebx
andl %ebx, %eax
```

OR La operación `OR` se lleva a cabo mediante la instrucción `ORL`:

```
# izq->generar()
pushl %eax

# der->generar()
popl %ebx
orl %ebx, %eax
```

Negación La negación lógica se consigue usando la instrucción `NOTL`:

```
# izq->generar()
notl %eax
```

2.10. Operadores relacionales

Los operadores relacionales funcionan todos basándose en la instrucción `CMP`, que activa ciertas banderas en el registro `EFLAGS`. Esas banderas son revisadas por las diferentes **instrucciones de salto**, que realizarán un salto condicional según cada caso.

Hay que recordar que al usar estos operadores, el objetivo final es colocar un 0 o 1 en el registro `EAX`, de forma que el resto de instrucciones conozcan el resultado. La forma habitual de usar estos operadores es la siguiente:

```
# izq->generar()
pushl %eax

# der->generar()
popl %ebx
```

```
  # Guardamos el primer operando en otro registro
  movl %eax, %ecx

  # Ponemos EAX a 1
  movl $1, %eax

  # Hacemos la comparacion
  cmpl %ecx, %ebx

  # Segun el caso usamos una instruccion de salto u otra
  jg etSalida

  # Esta rama se alcanza si no se hace el salto
  movl $0, %eax

  # Fin del operador
etSalida:
```

Las instrucciones que se utilizan para cada operador son:

Igual `je` (equal)

Distinto `jne` (not equal)

Mayor que `jg` (greater)

Menor que `jl` (lower)

Mayor o igual `jge` (greater or equal)

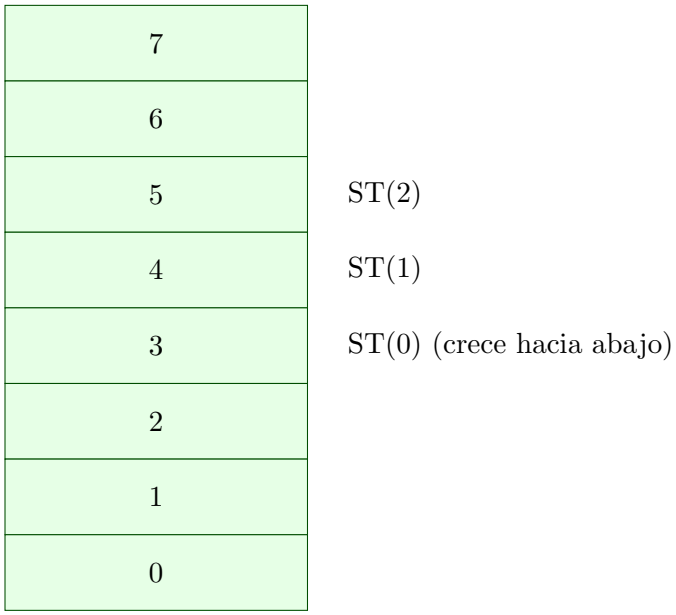
Menor o igual `jle` (lower or equal)

2.11. Flotantes

A la hora de trabajar con flotantes la cosa cambia, porque no se utilizan los registros ni las instrucciones habituales, sino que las operaciones se hacen a través de la **FPU** (*floating-point unit*), que cuenta con **sus propios registros** de control, de estado y de datos.

2.11.1. Registros de datos de la FPU

La FPU cuenta con **ocho registros** de 80 bits para almacenar números en formato de doble precisión extendida en coma flotante. Cuando **se carga** un flotante o un entero en un registro de la FPU, automáticamente **se convierte** al formato de doble precisión. Cuando es necesario **almacenar de vuelta en la memoria principal** el contenido de un registro de la FPU, el resultado puede dejarse en formato de doble precisión o **convertirse** a un formato de precisión simple, o a entero.



Pila de registros de datos de la FPU

Los ocho registros de la FPU funcionan como una pila de registros. Todo **el direccionamiento es relativo** al elemento en el tope de la pila. El número de registro que está en el tope se guarda en el campo `TOP` del registro de estado de la FPU. Las operaciones **de carga decrementan** el TOP en 1, cargando un nuevo valor en el nuevo registro en el tope. Las operaciones **de almacenamiento incrementan** el valor de TOP tras haber guardado el valor en memoria principal.

2.11.2. Carga y descarga de registros

Las operaciones de carga y descarga de registros de la FPU no permiten usar registros del procesador (`EAX`, `EBX`...) como operandos, sino direcciones de memoria principal..

La carga de datos en la FPU se hace mediante la familia de instrucciones `FLD`. Dado que los registros de la FPU funcionan como una pila, la operación de carga solo recibe como parámetro la ubicación de memoria principal.

```
fld origen
```

Según el tipo de datos que queramos cargar en la FPU, usaremos:

- `fild` - Carga de enteros. Normalmente usaremos `filds` (prefijo `s`) para cargar de una dirección de memoria de doblepalabra (32bits).
- `fld` - Carga de flotantes. Normalmente usaremos `flds` para cargar de 32 bits, y `fldl` para cargar de 64 bits

De forma similar, la instrucción `FST` se usa para **llevar a memoria** el valor del top de registros de la FPU, indicando la dirección de destino junto a la instrucción:

```
fst destino
```

Según el formato de destino que queramos usar, tenemos disponible varias instrucciones:

- `fist` - Para **enteros**. Lo más habitual es usar `fistl` (ojo con la l).
- `fst` - Para **flotantes**. Lo más habitual es usar `fsts`.

Una operación habitual es pasar un dato a la memoria principal borrándolo de la FPU – esto es, haciendo un *pop* de la pila de registros. Para ello se usa la instrucción `FSTP`, que hace lo mismo que `FST` añadiendo un *pop*. Existen las mismas variantes que en el caso anterior: `fistpl` para enteros, `fstps` y `fstpl` para flotantes.

2.11.3. Operaciones con flotantes

Para realizar operaciones con flotantes es necesario, lógicamente, que todos los operandos se encuentren en la pila de registros de la FPU.

Las instrucciones aritméticas tienen también su versión alternativa que incluye un *pop*, añadiendo el sufijo `p` al final.

Suma Utilizaremos la instrucción `fadd` o `faddp`, indicando los dos registros a sumar, siendo el segundo el destino de la suma final.

```
fadd %st(0), %st(1)
faddp  # Es lo mismo que faddp %st(0), %st(1)
```

Resta La resta es similar a la suma, pero usando `fsub` y `fsubp`.

Producto Igual que la suma y la resta, usando `fmul` y `fmulp`.

División Igual que lo anterior, pero usando `fdiv` y `fdivp`.

2.11.4. Operaciones con asignación

Para realizar una operación con asignación (como si usáramos el operador `+=` u otro similar), lo único necesario es realizar la operación aritmética de forma normal, para luego descargar el valor del registro en una dirección de memoria. Como ejemplo con la suma, si tenemos la dirección de destino en la pila:

```
faddp %st, %st(1)
popl %edx
fstps (%edx)
```

2.11.5. Comparaciones

Las comparaciones se realizan de manera similar a las operaciones. La instrucción `fcom` hace la comparación, por defecto, de `st0` y `st1`. Existen versiones que también realizan un pop (`fcomp`) y, lo más importante, existen versiones que modifican las flags del registro `EFLAGS` de la CPU, en lugar de las flags de la FPU, siendo posible utilizar las operaciones de salto condicional.

En particular, a la hora de comparar números de la FPU se usarán:

- `jb` y `jbe` en lugar de `jl` y `jle`.
- `ja` y `jae` en lugar de `jg` y `jge`.

Las comparaciones de igualdad se siguen haciendo con `je` y `jne`

3. Software de soporte

3.1. Compilación

3.1.1. Generar código ensamblador de programa C para 32bits

Se escribe en un fichero `main.s`:

```
cc -S -m32 -fno-asynchronous-unwind-tables main.c
```

O directamente en la salida estándar:

```
cc -S -m32 -fno-asynchronous-unwind-tables main.c -o -
```

La flag `-fno-asynchronous-unwind-tables` elimina las directivas adicionales que añade GCC (que comienzan con `.cfi`) para labores de depuración.

3.1.2. Compilar código ensamblador

Para generar código objeto usaremos:

```
as -o main.o main.s
```

Podemos añadir información de debug con:

```
as -gstabs -o main.o main.s
```

Finalmente, para enlazar y generar el ejecutable:

```
ld -o salida fichero.o
```

O podemos compilar y enlazar en un solo paso con gcc:

```
cc -m32 o salida -g main.c
```

3.1.3. Resaltar código ensamblador

```
cat main.s | pygmentize -l gas
```

3.2. GDB - Depurador

Tras compilar (con soporte para depuración) podemos usar el GNU Debugger (`gdb`) para seguir la ejecución de nuestro programa.

3.2.1. Lectura de código

list Listar el código del programa (opcional: indicar número de línea).

disassemble nombreFuncion Mostrar código ensamblador de una función.

3.2.2. Control de la ejecución

gdb ./ejecutable Lanzar el depurador sobre el ejecutable.

run Ejecutar el programa (opcional: indicar argumentos).

break numLínea Añadir un breakpoint en un número de línea.

delete numbreakpoint Borrar un breakpoint.

clear Borrar todos los breakpoints.

step Pasar a la siguiente instrucción.

next Pasar a la sgte. instrucción ignorando funciones.

finish Continuar hasta que la función retorne.

continue Continuar ejecución normal.

kill Finalizar ejecución.

3.2.3. Impresión de información

info registers Imprimir los valores de los registros (incluyendo eax, ebx, ecx, edx, esp y ebp).

info variables Imprimir las variables globales y estáticas.

info locals Imprimir las variables locales del marco actual.

info args Imprimir los argumentos de la función actual.

whatis nombre_variable Imprimir el tipo de una variable.

where Imprimir la pila de llamadas.

where full Imprimir la pila de llamadas con las variables locales.

print $registro Imprimir el valor de un registro o de una variable (opcionalmente `print/format elemento`).

display $registro Imprimir el valor de lo que sea tras cada instrucción.

x/Formato Dirección Inspeccionar directamente la memoria. La sintaxis del formato es la siguiente:

- Primero, un entero con la cantidad de posiciones a leer.
- Segundo, una letra con el formato, a saber:
  - **o** octal
  - **x** hexadecimal
  - **d** decimal
  - **u** unsigned
  - **t** binary
  - **f** float
  - **a** address
  - **i** instruction
  - **c** char
  - **s** string

- Tercero, una letra con el tamaño de cada posición, a saber:

  b byte

  h halfword

  w word

  g doble word

Podemos por ejemplo imprimir la stack leyendo la dirección del registro ESP (con `info registers`) y:

```
x/10xw $esp
```

3.3. Flex

Flex es un generador de **analizadores léxicos**. La estructura de un fichero `lexico.l`, con el que se define el analizador léxico, se divide en tres bloques separados por los caracteres `%%` :

```
%{
// Ficheros de cabecera y declaraciones globales
#include <iostream>

%}

// Opciones de Flex
%option noyywrap
%option yylineno

// Definición de familias usando expresiones regulares
DIGITO     [0-9]

%%

// Lista de patrones y acciones (en C)
[ \n\t]  { ; }
.        { cout << "Caracter: " << yytext << endl; }

%%

// Código C. En un compilador esto está vacío ya que el código
// del main va en el analizador sintáctico

int main ()
{
  // La llamada a yylex inicia el análisis léxico
  yylex();
}
```

El fichero con la definición léxica se procesa con:

```
flex lexico.l
```

Esto genera un fichero `lex.yy.c` con la implementación del analizador léxico. Entre otras cosas contiene la implementación de la función `yylex`, que irá analizando la entrada y ejecutando las reglas asociadas a los patrones que sean reconocidos.

Nota: es posible generar el analizador léxico en C++ usando:

```
flex --c++ lexico.l
```

Esto generará una implementación basada en clases en lugar de funciones libres, pero es una opción experimental que no funciona bien.

Para compilar, usaremos:

```
g++ lex.yy.c --o ejecutable
```

Si se produjese algún error de enlazado, habría que enlazar a la biblioteca de flex, añadiendo al comando la opción `-lfl`

3.4. Bison

Bison es un generador de analizadores sintácticos. Se nutre de una gramática libre de contexto definida en notación BNF.

El fichero `sintactico.y`, que suele albergar la definición del analizador sintáctico, tiene una estructura igual al fichero `lexico.l`, basada en tres partes divididas por los caracteres `%%`

```
%{
// Declaraciones de C/C++
#include <iostream>

// Declaración externa de la función del lexer
extern int yylex();

%}

// Declaraciones de tókens
%token CADENA
%token INT
%token CHAR

// Otras declaraciones de Bison
%start inicial

%%
// Reglas

inicial : sentencia ';' inicial { /* Código */ }
        |
;

sentencia : INT CADENA
          | CHAR CADENA
;

%%
// Código C/C++
int main () {
    yyparse();
}
```

En la primera sección se definen los tókens que el analizador léxico debe devolver. Para que Flex tenga información de esos tókens serán necesarios dos pasos. Primero, que Bison genere un fichero de cabecera con esa información, y segundo, que nuestro analizador léxico incluya esa cabecera.

El primer paso lo conseguimos con

```
bison -d sintactico.y
```

Esto genera un fichero `sintactico.tab.h` con la información de los tókens. Tendremos que añadir lo siguiente en la sección de declaraciones del léxico:

```
#include "sintactico.tab.h";
```

También se generará un fichero `sintactico.tab.c` con la implementación del analizador sintáctico, que finalmente habrá que compilar.

3.4.1. Tókens y símbolos

El analizador léxico parsea poco a poco la entrada, informando al párser (Bison) de lo que encuentra mediante bloques de información conocidos como **tókens**.

La función **yylex** irá consumiendo la entrada y ejecutando las reglas de los patrones HASTA que uno de los patrones devuelva un tóken. Ese token será el valor que devuelva la función `yylex`.

Los tókens pueden ser símbolos definidos mediante la directiva `%token` en la sección de declaración de Bison, o caracteres individuales. Internamente, Bison genera un enumerado con los tókens, asignándoles un valor entero a cada uno.

Por otro lado, la **gramática** con la que se alimenta a Bison se compone de **reglas de producción**, que se dividen en un *símbolo de cabecera* y un *cuerpo*. El cuerpo se compone de símbolos y acciones, que se ejecutarán cuando la entrada coincida con esta regla de la gramática:

```
expresion : NUMERO '+' expresion { cout << "Suma" << endl; }
          | NUMERO               { cout << "Número" << endl; }
;
```

Los símbolos que componen las reglas de la gramática pueden ser símbolos *terminales* (también conocidos como *tókens*), que representan de forma simbólica la entrada del lenguaje que estemos analizando, y símbolos *no terminales*, que son aquellos que serán reemplazados por otras reglas. En el ejemplo anterior, el símbolo `expresion` es un símbolo no terminal, mientras que `NUMERO` es un símbolo terminal o tóken.

La cabecera de una regla siempre llevará un símbolo no terminal, mientras que el cuerpo podrá incluir símbolos terminales y no terminales.

3.4.2. Valor de los símbolos

Todos los símbolos (tanto tókens como no terminales) tienen un valor asociado. Algunos símbolos, por ejemplo, necesitarán tener asociada una cadena, mientras que otros necesitarán almacenar un puntero. Para gestionar esa información, se listan los diferentes tipos de datos en una `unión` en la zona de declaraciones de Bison:

```
%union {
  string * puntero;
  int entero;
}
```

Luego, es necesario indicarle a Bison qué tipo de datos usará cada uno de los símbolos:

```
 // No terminales:
%type <puntero> expresion

 // Terminales:
%token <entero> NUMERO
```

Al realizar el **análisis léxico**, el valor de cada tóken se guardará en la variable `yylval`. Para computar ese valor es posible acceder al texto reconocido por el analizador, que se encuentra en la variable `yytext`. Será necesario hacer referencia al campo de la unión que tenga asignado el símbolo terminal:

```
[0-9]      { yylval.entero = atoi(yytext); return NUMERO }
```

Para **acceder** al valor de los símbolos de una regla se puede usar la notación del dólar.

- `$$` representa el valor de la cabecera de la regla.
- `$1, $2...` representan el valor del elemento 1, 2... de la regla. Las acciones semánticas también pueden tener valor, por lo que afectan a la numeración.
- `$0, $-1...` representan el valor de los elementos que aparecen a la izquierda de la regla. Por ejemplo:

  ```
  expresion : tipo nombre ';'
  ;

  nombre : CADENA { /* Aquí $0 es el valor del símbolo tipo */ }
  ;
  ```

3.4.3. Acciones intercaladas

Como se ha comentado, las reglas de producción pueden contener acciones semánticas, que se ejecutarán cuando el párser haga uso de esa regla gramatical.

Las acciones pueden colocarse en cualquier punto del cuerpo de una regla, ejecutándose en el orden indicado. Internamente, Bison convierte todas esas reglas de forma que las acciones semánticas estén al final

3.4.4. yylineno, yytext, yyerror y yylex

La función que el léxer provee para leer los tókens de la entrada es `yylex()`. El párser llamará a esa función periódicamente para leer tókens de la entrada. Es por ello necesario declarar su existencia en la zona de declaraciones del fichero `sintactico.y`:

```
%{
extern int yylex();
%}
```

Flex guarda en la variable global `char * yytext` la entrada que ha emparejado con las reglas léxicas, por lo que es posible acceder a ese texto desde el párser, declarando (igual que antes) la variable previamente:

```
%{
extern int yylex();
extern char * yytext;
%}
```

Además, Flex puede informarnos del número de línea de la entrada por la que está trabajando en cada momento. Lo hará si añadimos `%option yylineno` en la zona de declaraciones del `lexico.l`. Así, la variable global `int lineno` contendrá el número de línea. Igual que antes, para usarla en el párser es necesario declararla previamente.

```
%{
extern int yylex();
extern char * yytext;
extern int lineno;
%}
```

Por último, cuando se produce un error de sintaxis Bison llama a la función `yyerror (char *)`, que será necesario definir para poder gestionar los mensajes de error correctamente. Una posible implementación podría ser:

```
void yyerror(const char *msg) {
    printf("Error: %s, línea: %i, texto: '%s'\n", msg, yylineno, yytext);
}
```

4. Algo de teoría: el display de Pascal

El display es un vector de punteros a registros de activación que se suele almacenar en el propio marco de activación, y en general, es más rápido que seguir los enlaces de acceso.

Algunas instrucciones de ensamblador de la arquitectura Intel x86 son capaces de reservar bytes de espacio para variables locales y organizar un display de profundidad *"prof"*. Para ello copian, en parte, el display del marco de activación anterior y lo modifican adecuadamente.

En Pascal, cuando se establece un nuevo registro de activación para un procedimiento a profundidad de anidamiento i:

- Se guarda el vector actual de $d[i]$ dentro del nuevo registro de activación.
- Se apunta $d[i]$ al nuevo registro de activación.

Justo antes de que finalice una activación, se restablece $d[i]$ al valor guardado en el registro de activación.

En cuanto a los enlaces (dinámicos), se trata de funciones que asocian a un cierto identificador una dirección de almacenamiento. En relación al display de Pascal, cada vez que se activa un procedimiento, a sus variables locales automáticas se les liga direcciones diferentes. Esto es un enlace dinámico. Los enlaces dinámicos son los que se determinan en tiempo de compilación y se dan en otros lenguajes como Fortran o Cobol.

5. Ejemplos de código ensamblador

5.1. Cargar un flotante y devolverlo como entero

```
    .file   "main.c"
    .section    .rodata

cadena:
    .string "%i\n"

base:
    .float 28.9732

    .text

    .globl  main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp

    flds base
    fistpl -4(%esp)
    movl -4(%esp), %eax

    pushl %eax
    pushl $cadena
    call printf

    leave
    ret

    .size   main, .-main
```

5.2. Potencia de enteros

```
    .file   "main.c"
    .section    .rodata
.LC0:
    .string "%i\n"
    .text
    .globl  main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp

    # izq->generar
    # Uso 5 como ejemplo
    movl $5, %eax
    pushl %eax

    #der->generar
    # Uso 2 como ejemplo
    movl $2, %eax

    popl %edx
    movl %eax, %ecx
    movl $1, %eax

inicio:
    cmp $0, %ecx
    je final
    subl $1, %ecx
    imull %edx, %eax
    jmp inicio

final:

    # Imprimo
    pushl   %eax
    pushl   $.LC0
    call    printf

    leave
    ret
    .size   main, .-main
```

5.3. Potencia de flotantes

```
    .file   "main.c"
    .section    .rodata
.LC0:
    .string "%i\n"

base:
    .float 2.5

    .text
    .globl  main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp

    # Antes de generar meto 1 en la FPU
    fld1

    # izq->generar
    # Esto genera la base, que previamente esta en la FPU
    flds base

    # der->generar
    # Esto genera el exponente, que debe ser entero
    movl $3, %eax

    # Muevo el contador a ECX
    movl %eax, %ecx

inicio:
    # Si es 0, salto a la salida
    cmp $0, %ecx
    je final

    # Decremento el contador
    subl $1, %ecx

    # Hago el producto
    fmul %st(0), %st(1)

    # Salto al inicio
    jmp inicio

final:
    # Al final hago un pop de la pila de la FPU
    fstps -4(%esp)

    # En este punto st(0) contiene el resultado
    leave
    ret
    .size   main, .-main
```