Introducción al assembler de AT&T

Horacio Goetendía Bonilla

13 de Octubre de 2002

Capítulo 1

Generalidades y cuestiones preliminares

1.1. Significancia de los bits

La memoria de los computadores no guardan la información en base decimal, lo hace en base 2 por que simplifica de manera considerable la manera en como almacena la información en la memoria

Al primer dígito del número 10101110 es decimal al 1 se le llama el bit más significativo (msb) por sus siglas en inglés y al último se le llama bit menos significativo o (lsb).

La unidad básica de la memoria es el byte que es la agrupación de 8 bits.

1.2. La base 16 o hexadecimal *(hex)*

Los elementos de numeración en base 2 son {0,1} en la base 10 son {0,1,2,3,4,5,6,7,8,9} en base hexadecimal es {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} donde A es equivalente a 10_{10} lo mismo que para F_{16} su equivalente es 15_{10}.

La razón de la inclusión de la numeración hexadecimal en el assembler es la facilidad de transformación que existe entre la base 2 y la base 16, por ejemplo $23E_{16}$ transformándolo a base 2 es 0010 0011 1110_2, con este ejemplo vemos que cada dígito de la base 16 se descompone a base 2 pero dentro de 4 bits equivalente a un nible por ejemplo el 2_{16} se descompone en 0010_2, pero tengan cuidado de completar el nible con ceros a la izquierda como se hizo en el ejemplo anterior.

1.3. El orden dentro de una computadora

Como sabemos para entender desempeño o funcionamiento de un sistema debemos entender de que manera está organizada y que orden rige dentro de él.

0	1	2	3	4	5	6	7
2A	12	2F	3C	14	5F	3A	4B

Cuadro 1.1: Dirección de memoria

palabra	2 bytes
doble palabra	4 bytes
cuádruple	8 bytes
párrafo	16 bytes

Cuadro 1.2: Unidades de memoria

1.3.1. La memoria

Como habíamos mencionado la unidad básica de la memoria es el byte entonces una computadora con 32 Megas de memoria RAM puede almacenar 32768 bytes de información. Cada byte en la memoria es etiquetado con un único número denominado *dirección* como se ve en la tabla 1.1.

Todos los datos que son guardados en memoria son numéricos, los caracteres son usados usando unas equivalencias estamos hablando de los códigos *ASCII*.

Otras denominaciones o convenciones en las unidades de la memoria son las palabras y se muestran en el cuadro 1.2

1.3.2. La CPU

La CPU es la encargada de la ejecución de las instrucciones, las instrucciones que realiza la CPU son simples y requieren de datos, estos datos se guardan en sitios de la CPU llamados *registros*, la razón de por que la CPU guarda los datos en sus registros y no en la memoria es por que el acceso a los registros es mas rápido que el acceso a memoria, sin embargo el número de registros en la CPU son limitados, de modo que el programador debe tener cuidado de mantener en registros solo los datos que actualmente se usan.

1.3.3. La 8086 y sus registros de 16 bits

La 8086 provee cuatro registros de 16 bits cada uno para propósitos generales estos registros son denominados: AX, BX, CX y DX, cada uno de estos registros puede ser descompuesto en dos registros de un byte cada uno como se muestra en la Figura 1.1 en la cual se muestra la descomposición del registro AX.

El registro AH de la Figura 1.1 contiene los 8 bits más significativos mientras que el AL contiene los restantes 8 bits menos significativos, con

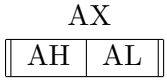

Figura 1.1: Descomposición de registros

frecuencia AH y AL son usados independientemente sin embargo es necesario e importante recalcar que ellos no son independientes de AX, es decir cambiar el valor en AX cambia el valor en AH y AL y viceversa, el propósito general de los registros son mover datos e instrucciones aritméticas.

También existen dos registros índice el SI y el DI ellos con frecuencia son usados como punteros, pero pueden ser utilizados para muchos propósitos como los registros generales con la excepción de que nos se pueden dividir en sub-registros.

Los registros BP y SP de 16 bits son usados para apuntar a datos en la pila del lenguaje de máquina.

Los registros CS, DS, SS, ES de 16 bits cada uno son los denominados registros de segmento. Estos registros denotan que memoria es usada para diferentes partes de un programa. CS denota el segmento de código (Code Segment), DS para segmento de datos (Data Segment), SS para el segmento de pila (Stack Segment) y ES para el segmento extra (Extra Segment), El registro ES es usado como un registro de segmento temporal. Más adelante explicaremos en detalle estos registros.

El registro IP (Instruction Pointer) es usado con el registro CS (Segmento de Código) para secuenciar la dirección de la siguiente instrucción a ser ejecutada por la CPU.

Los registros bandera o (Flags registers) guardan información importante acerca de resultados de la instrucción anterior. Los resultados de las instrucciones son almacenados a manera de bits en estos registros

1.3.4. la 80386 y sus registros de 32 bits

El 80386 y los procesadores subsecuentes han extendido sus registros. Por ejemplo los registros AX de 16 bits fueron extendidos a registros de 32 bits. Pero para mantener la compatibilidad AX permanece refiriéndose al registro de 16 bits y EAX es usado para referirse al registro extendido de 32 bits. Como sucede esto, la respuesta es que AX se refiere solo a los 16 bits menos significativos de EAX como sucedía anteriormente con AL que se refería y los 8 bits menos significativos del registro AX. Como referencia mencionamos que no se pueden accesar directamente a los 16 bits más significativos de EAX.

Con lo que respecta a los registros de segmento continúan siendo de 16 bits pero se adhieren dos nuevos registros: FS y GS, esta nomenclatura no determina algún comportamiento o función especial, estos registros aparecen

también como registros temporales como el ES.

1.3.5. El modo real

En el modo real (Real Mode), la memoria está limitada a solo un Megabyte 2^{20} bytes por lo tanto las direcciones validas van desde 0000_{16} a $FFFF_{16}$, estas direcciones requieren de números de 20 bits. Obviamente un numero de 20 bits no cabría en un registro de 16 bits de la 8086. Intel solucionó este problema usando dos valores de 16 bits para determinar una dirección. Siendo los primeros 16 bits el *selector* y su valor sería almacenado en los registros de segmento. Los siguientes 16 valores son denominados *desplazamiento*. Por lo tanto la memoria física de 32 bits nos lo da el binomio *selector:desplazamiento* con la siguiente formula:

$$16 * selector + desplazamiento$$

Por ejemplo la dirección física de 023B:0018 es

$$\begin{aligned} 023B_{16} * 16_{10} &= 023B0_{16} \\ 023B0_{16} + 0018_{16} &= 023C8_{16} \end{aligned}$$

Representemos lo anterior en base 2 para un mayor entendimiento.

$$\begin{aligned} 023C_{16} &= 0000\ 0010\ 0011\ 1100_2 \\ 0018_{16} &= 0000\ 0000\ 0001\ 1000_2 \\ 023C_{16} * 16_{10} &= 0000\ 0010\ 0011\ 1100\ 0000_2 \end{aligned}$$

Sumando tenemos:

$$\begin{array}{r} 0000\ 0010\ 0011\ 1100\ 0000 \\ +\ 0000\ 0000\ 0001\ 1000 \\ \hline 0000\ 0010\ 0011\ 1101\ 1000 \end{array}$$

El valor del selector es el número de párrafo.

La segmentación de las direcciones trae con sigo ciertas desventajas como son:

- Un simple selector solo referencia 64k de memoria ($2^16 = 65536bytes = 64k$) entonces ¿que pasaría con un programa que tiene más de 64k de código? un simple valor en el registro de segmento de código CS no bastaría para la ejecución del programa entero (sabemos de antemano que CS establece el área donde se halla el programa en ejecución). Entonces el programa debe ser partido en piezas (llamados segmentos) menores que 64k de tamaño. Entonces cuando la ejecución se mueve de segmento a segmento el valor de CS cambia. Problemas similares ocurren con el registro de segmento de datos DS con programas con grandes cantidades de datos.

- Cada byte en memoria no tiene una única dirección de segmento por ejemplo la referencia 047C:0048 y 047D:0038 se refieren a la dirección física 04808. Esto puede complicar las comparaciones de las direcciones de segmento.

1.3.6. Modo protegido de 16 bits

En el modo protegido de 286 el valor del selector es interpretado completamente diferente que en el modo real, En el modo real el valor de un selector es el número de párrafo de la memoria física. En modo protegido un selector es un índice dentro de una tabla de descriptores. En ambos modos los programas son divididos en segmentos. En el modo real estos segmentos están en posiciones fijas de la memoria física.

El modo protegido usa una técnica llamada *memoria virtual*. La idea base del sistema de memoria virtual es de solo mantener en memoria la data y el código de los programas que están actualmente usándolo. Otra data y código son guardados temporalmente en el disco hasta que sean necesitados nuevamente. En el modo protegido de 16 bits, los segmentos son movidos del disco a la memoria o viceversa como sea necesario. Cuando un segmento retorna del disco a la memoria es muy probable que vaya a otra posición de la memoria a la que estaba antes de ser descargado al disco todo esto es realizado de modo transparente por el sistema operativo. El programa no tiene que ser escrito de una manera especial para un sistema con memoria virtual como dijimos de esto se encarga el sistema operativo.

En el modo protegido, cada segmento es asignado a una entrada en una tabla de descriptores. Esta entrada tiene toda la información que el sistema necesita para conocer acerca del segmento. Esta información incluye: ¿este segmento esta actualmente en memoria?, si lo está ¿donde está?, ¿que permisos de acceso tiene?. El índice de la entrada del segmento es el valor del selector que es guardado en los registros de segmento.

Una gran desventaja del modo protegido de 16 bits es que los desplazamientos permanecen en cantidades de 16 bits. Como consecuencia de esto, las medidas de los segmentos permanecen limitados a los 64k. Esto obliga el uso de grandes y problemáticos arrays.

1.3.7. El modo protegido de 32 bits

El 80386 introdujo el modo modo protegido de 32 bits, hay dos grandes diferencias entre el modo protegido de 80286 de 16 bits y el 80386 de 32 bits y estas son:

1. Los desplazamientos son expandidos a 32 bits. Esto permite desplazamientos en el rango de los 2^{32} que es equivalente a los 4 billones Estos segmentos pueden tener medidas al rededor de los 4 Gigabytes.

2. Los segmentos pueden ser divididos en piezas de mucho más pequeña de 4k de tamaño llamados *páginas*. La memoria virtual trabaja con páginas en lugar de segmentos. Esto significa que partes del segmento podrían estar en memoria. En cambio en el 80286 modo 16 bit, el segmento entero podría estar en memoria o no. Los sistemas operativos Linux, Windows 9X, Windows NT, OS/2 corren en modo protegido y paginado de 32 bits

1.3.8. Interrupciones

Algunas veces el flujo ordinario de un programa debe ser interrumpido para procesar eventos que requieren pronta respuesta. El hardware de la computadora provee un mecanismo llamado *interrupciones* para manejar estos eventos. Por ejemplo cuando el mouse es movido, el hardware del mouse interrumpe el programa actual para manejar el movimiento del mouse. La interrupción causa que el control será pasado a un *manejador de interrupciones*. Los manejadores de interrupciones son rutinas que procesan la interrupción. Cada tipo de interrupción es asignado a un número entero. En el inicio de la memoria física reside una tabla de *vectores de interrupciones* y contienen las direcciones segmentadas de los manejadores de interrupciones, el número de interrupción es esencialmente un índice en esa tabla.

Interrupciones externas son generadas fuera de la CPU (el mouse es un ejemplo de esto). Muchos dispositivos de entrada y salida generan interrupciones (teclado, timer, drivers de disco, CD-ROM, tarjetas de sonido, etc). Las interrupciones internas son generadas dentro del CPU y estas pueden ser por algún error o por la instrucción de la interrupción. Las interrupciones de error son llamados *traps*. Las interrupciones generadas de la instrucción de la interrupción son llamadas *interrupciones de software*. El DOS usa estos tipos de interrupciones para implementar sus API's (Application Programming Interface), otros sistemas operativos mucho más modernos usan una interfaz basada en C [1].

Muchos manejadores de interrupciones retornan el control al programa interrumpido cuando acaban. Ellos recuperan todos los valores de los registros a los mismos valores antes de que los interrumpieran, es como si la interrupción no hubiera sucedido, los traps generalmente no retornan si no que abortan el programa.

[1] Sin embargo, ellos podrían usar una interfase de bajo nivel en el nivel del kernel.

Capítulo 2

El assembler de AT&T

2.1. El lenguaje ensamblador

2.1.1. El lenguaje de máquina

Cada tipo de CPU entiende su propio lenguaje de máquina. Las instrucciones en el lenguaje de máquina son números guardados como bytes en la memoria. Cada instrucción tiene su propia y única codificación numérica llamado *Código de operación* o *opcode*. La opcode siempre está al inicio de la instrucción. Muchas instrucciones también incluyen data (constantes o direcciones) usados por la instrucción.

Es dificultoso manejar el lenguaje de máquina directamente. Esto implicaría descifrar el código numérico de cada instrucción que realmente para un humano e tedioso. Un ejemplo de esto es: La instrucción que dice suma los registros EAX y EBX y el resultado que lo almacene en EAX en código de máquina es:

03 C3

Como vemos todo esto es en codificación hexadecimal. Es muy tediosa pero afortunadamente existe un programa llamado *assembler*o ensamblador que puede hacer este tedioso trabajo por el programador.

2.1.2. El lenguaje assembler

En el lenguaje assembler el programa es almacenado como texto (como los lenguajes de alto nivel). Cada instrucción en assembler representa exactamente una instrucción de máquina. Por ejemplo: Aprovechemos el ejemplo pasado y representemos lo en el lenguaje assembler. Entonces esto sería:

add eax,ebx

Acá el significado de la instrucción es mucho más comprensible para el humano. La palabra *add* es un *mnemónico* para representar la adición. La forma general de una instrucción en assembler es:

mnemónico operando(s)

El assembler es un programa que lee un archivo de texto que contiene instrucciones de assembler convirtiéndolos en código de máquina. El assembler es mucho más sencillo que un compilador porque cada oración en código assembler directamente representa una simple instrucción de maquina. Los ejemplos de este artículo usan el NASM (Netwide Assembler) que se puede obtener gratuitamente en Internet en http://www.web-sites.com.uk/nasm/.

Existen otros ensambladores más comunes como el MASM (Miscrosoft Assembler) o la de borland (TASM) (Turbo Assembler), claro que existe diferencias en la sintaxis de un lenguaje ensamblador a otro.

2.1.3. Instrucciones y operandos

Los operandos pueden ser de los siguientes tipos:

registro Estos operandos se refieren directamente a el contenido de los registros de la CPU.

memoria Estos se refieren a datos en memoria.

inmediato Estos son valores estacionarios que son listados en la instrucción por si misma. Es decir que estos están guardados en la instrucción por si misma (en el segmento de código) no en el segmento de datos.

implícitos Hay operandos que no son mostrados explícitamente. Por ejemplo, la instrucción de incremento que suma uno al registro o memoria. Ese uno que se le suma ya es implícito

2.1.4. Instrucciones básicas

La instrucción básica en assembler es la instrucción *MOV*. Esta instrucción mueve datos de un lugar a otro y toma dos operandos para esto de la siguiente forma:

mov destino,origen

El dato especificado como origen es copiado al destino. Una restricción de esta instrucción es que ambos operandos no deben ser del tipo memoria. Otra regla en el assembler es que los operandos no deben ser de diferente tamaño como ejemplo tenemos que no se puede guardar el valor de AX en BL.

A continuación mostramos más ejemplos de la instrucción *mov*(el texto que sigue a un ; es considerado como comentario por el lenguaje assembler).

```
mov   eax,3   ;   Guarda el valor de 3 en el registro AX
mov   bx,ax   ;   Guarda el valor de ax en el registro BX
```

El operando 3 es un operando del tipo inmediato en el ejemplo anterior.

La instrucción *ADD* se usa para sumar enteros.

```
add   eax,4   ;   eax=eax + 4
add   al,ah   ;   al=al+ah
```

La instrucción *SUB* es el mnemónico de la sustracción.

```
sub   bx,10     ;   bx=bx - 10
sub   ebx,edi   ;   ebx=ebx - edi
```

Las instrucciones *INC* y *DEC* incrementa o decrementan valores uno por uno. El uno que se incrementa o decrementa es un operando implícito, el código de máquina que genera INC y DEC es menor que el de las instrucciones ADD y SUB.

```
inc   ecx   ;   ecx++
dec   dl    ;   dl– –
```

2.1.5. Directivas

Una directiva es un artificio de assembler no de la CPU y es usado por las instrucciones de assembler para hacer algo o informar el ensamblado de algo. Las directivas no se transforman en código de máquina. Los usos comunes de las directivas son para:

- Definir constantes
- Definir memoria para guardar datos en él.
- Agrupar la memoria en segmentos.
- Para incluir códigos fuente.
- Para incluir otros archivos.

NASM pasa el código a un preprocesador, a diferencia del C la directiva para el preprocesador es el símbolo % y no #.

La directiva *equ*: La directiva equ es usada para definir un símbolo. Los símbolos son las llamadas constantes que pueden ser usadas en el programa en assembler. El formato es el siguiente:

símbolo equ *valor*

El valor símbolo no debe ser redefinido después.

Unidad	Letra
byte	B
palabra	W
doble palabra	D
cuádruple palabra	Q
diez bytes	T

Cuadro 2.1: Tamaños de los objetos

La directiva *% define*: Esta directiva es similar a la directiva *#define* del C acá un ejemplo:

%define TAM 100 mov AX,TAM

Directiva de datos Las directivas de datos son usadas en los segmentos de datos para definir espacios de memoria. Hay dos maneras en la que se puede reservar la memoria. La primera manera solo define el espacio para el dato. La segunda forma hace lo mismo pero definiendo el valor inicial de esta. La primera forma se logra empleando la directiva *RES X*, dónde X es reemplazado por una letra que determina el tamaño del objeto u objetos que será almacenado. En el cuadro 2.1 mostramos sus posibles valores.

El segundo método (el método que define además su valor inicial) usa una de la directiva *DX* donde X representa la mismas equivalencias que el cuadro 2.1.

Es muy común marcar locaciones de memoria con etiquetas. Las etiquetas nos permiten referirnos de una manera fácil locaciones de memoria en el código que estamos desarrollando. A continuación unos cuantos ejemplos:

```
L1   db     0       ;  byte etiquetado con L1 y su valor inicial es 0.
L2   dw     1000    ;  palabra etiquetada con L2 y su valor inicial es 1000.
L3   db     11101b  ;  byte etiquetada con L3 y su valor inicial es 11101_2.
L4   dd     1A9h    ;  doble palabra y su valor inicial es 1A9_16.
L5   resb   1       ;  un byte no inicializado.
L6   dd     10Ah    ;  doble palabra y su valor inicial es 10A_16
L7   db     "A"     ;  byte inicializado al código ASCII para A=65.
```

Las comillas simples o dobles comillas son tratadas de igual forma. *La definición consecutiva de datos son guardadas secuencialmente en memoria.*Las secuencias de memoria también pueden ser definidas.

```
L8   db   0,1,2,3                    ;  define 4 bytes.
L9   db   "M","U","N","D","O",0      ;  Define la cadena = "MUNDO".
L10  db   'MUNDO',0                  ;  Igual que L9.
```

Para secuencias largas, NASM tiene la directiva *TIMES* que es muy usada. Esta directiva repite su operando un número especificado de veces. Por ejemplo.

```
L11   times db   0     ;  Equivalente a 100 veces db 0.
L12   resw       100   ;  Reserva espacio para 100 palabras.
```

Recordemos que podemos utilizar las etiquetas para referirnos a datos en el código. Existen dos manera en que las etiquetas pueden ser usadas. Si una etiqueta plana es usada, esta es interpretada como una dirección (o desplazamiento) de el dato. Si la etiqueta está entre corchetes, es interpretada como el dato de la dirección. En otras palabras, uno debería pensar de una etiqueta como un puntero a el dato y los corchetes son como el uso del * en C es decir hace referencia al contenido de la referencia (MASM y TASM siguen diferentes convenciones). En el modo de 36 bits las direcciones son de 36 bits. Acá algunos ejemplos.

```
1  mov   al,[L1]     ;  Copia el byte de L1 a AL.
2  mov   eax,L1      ;  Asigna la dirección de L1 a EAX.
3  mov   [L1],ah     ;  Copia el byte de AH a L1
4  mov   eax,[L6]    ;  Copia la doble palabra de L6 a EAX.
5  add   eax,[L6]    ;  EAX=EAX+la doble palabra de L6.
6  add   [L6],eax    ;  La doble palabra de L6 += EAX.
7  mov   al,[L6]     ;  Copia el primer byte de la doble palabra de L6 a AL.
```

La linea 7 muestra una propiedad importante del NASM. Este ensamblador no mantiene el tipo de dato a la cual se refiere la etiqueta. Es recomendable que el programador se asegure que se usa la etiqueta correcta. Después será común guardar direcciones de datos en en registros y usar los registros como punteros al igual que en C. Recalco que, el compilador no verifica si el puntero es usado correctamente. De esta manera el assembler es más propenso a generar errores que el C.

Consideremos la siguiente instrucción:

```
mov   [L6],1   ;  Guarda 1 en L6.
```

El ejemplo anterior generará un error ("operation size not specified") este error sucede por que assembler no sabe si guardará 1 como byte, palabra o doble palabra. Para superar este error adherimos el especificador de tamaño:

```
mov   dword [L6],1   ;  Guarda 1 en L6.
```

Esto le dice al assembler guarda 1 como doble palabra en L6. Otros especificadores de tamaños son: Byte, Word, Qword y Tword.

2.1.6. Entrada y Salida I/O

La entrada y la salida son actividades muy dependientes del sistema. Estas operaciones implican una interfaz con el hardware del s sistema. Los lenguajes de alto nivel como el C proveen librerías estándar que provee una forma simple y uniforme de programación con los dispositivos de entrada y salida. Los lenguajes ensambladores proveen librerías no estándares, si no que directamente acceden al hardware (que es una operación privilegiada en el modo protegido) o usan rutinas de bajo nivel que el sistema operativo provee.

Es muy común para las rutinas del ensamblador ser interfaceadas con el C. Una ventaja de esto es que el código en assembler puede las librerías estándar del C para rutinas de I/O. Sin embargo uno debe conocer las reglas para el paso de información entre las rutinas que el C usa. Las reglas para el paso de información son muy complicadas (mas adelante se explicarán con detenimiento). Para simplificar las operaciones de entrada y salida el autor ha desarrollado sus propias rutinas que esconden las complejas reglas del C y provee una interfaz mas simple a continuación mostramos las rutinas proveidas.

print_int Imprime en la pantalla el valor del entero en EAX.

print_char Imprime en la pantalla el caracter ASCII del registro AL.

print_string Imprime en la pantalla el contenido de la cadena de la dirección almacenada en EAX (Esta cadena debe tener la terminación NULL);

print_nl Imprime en la pantalla el caracter nueva linea.

read_int Lee un entero del teclado y lo guarda en el registro EAX.

read_char Lee un entero del teclado y guarda su código ASCII en el registro EAX.

Todas estas rutinas preservan el valor de todos los registros excepto para las rutinas de lectura. Estas rutinas modifican el valor del registro EAX. Para usar estas rutinas, uno debe incluir un archivo con información que el ensamblador necesita para usarlos.

Para incluir el archivo en el NASM se usa la directiva del preprocesador %include. Las siguientes lineas incluyen el archivo necesitado por las rutinas del autor.

%include “asm_io.inc”

Para usar una de las rutinas de impresión, uno debe cargar el registro EAX con el valor correcto y usar la instrucción *CALL* que es equivalente al

llamado de una función en los lenguajes de alto nivel. Esta llamada genera un salto en la ejecución a otra sección del código después retornando al origen después que termine la rutina.

2.1.7. El debug

La librería del autor también contiene algunas rutinas útiles para "debugear"los programas. Estas rutinas muestran información acerca del estado de la computadora sin modificar su estado. Estas rutinas son realmente *macros* que preservan el estado actual de la CPU y es entonces donde hace un llamado a la subrutina. Las macros están definidas en el archivo *asm_io.inc*. Los macros son usados igual que instrucciones ordinarias. Los operandos de los macros deben de estar separados por comas.

Existen tres rutinas para el debuggeo estas son:*dump_regs, dump_mem* y *dump_math*; estas rutinas muestran los valores de los registros, memoria y la del coprocesador matemático respectivamente.

2.2. Creando un programa

Hoy en día, es inusual crear un solo programa estándar escrito completamente en assembler. El ensamblador es usualmente usado para realizar ciertas rutinas críticas. Esto es por que es más fácil programar en un lenguaje de alto nivel que en ensamblador. También, usar assembler hace que un programa sea muy difícil de portar a otras plataformas. En conclusión es raro usar assembler para todo.

Entonces ¿Por que debo aprender assembler es decir que ventajas tiene?

1. Algunas veces el código escrito en ensamblador puede ser mas pequeño y veloz que otros códigos generados por otros compiladores.

2. Ensamblador permite el acceso al hardware con ciertas bondades directamente que podría ser dificultoso o imposible de usar en los lenguajes de alto nivel.

3. Aprender a programar en ensamblador ayuda a uno a ganar mayor entendimiento del trabajo de la computadora.

4. Aprender a programar en ensamblador ayuda a entender mejor como los compiladores y los lenguajes de alto nivel trabajan.

2.2.1. El primer programa

El siguientes programas en este artículo comenzarán del simple código en C llamado *driver.c* del cuadro 2.2

```
int main()
{
int ret_status;
ret_status = asm_main();
return ret_status ;
}
```

Cuadro 2.2: El programa driver.c

Este simple programa llama a otra función llamada *asm_main*. La función *asm_main* es realmente la rutina que será escrita en ensamblador. Hay muchas ventajas con el uso del programa *driver.c*. Primero esto deja al C setear el programa para que corra correctamente en el modo protegido. Todo los segmentos y sus correspondientes registros de segmento serán inicializados por el C. El código en ensamblador no se preocupará por hacer esto. Segundo, la librería del C será disponible para ser usada por el código de ensamblador. Las rutinas de I/O del autor toma ventaja de esto, usando las I/O rutinas del C (printf, etc).