Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Programación en Arquitectura x86 (16 bits)

1. Motivación

La arquitectura x86 es la más usada hoy en día en los computadores personales, y por tanto es relevante entender como se desarrollan programas en el assembly de esta.

2. Programación en Arquitectura x86 de 16 bits

Para poder desarrollar programas en el assembly de la arquitectura x86, es necesario entender los elementos básicos de esta que pueden ser usados en los programas, en particular: registros, variables e instrucciones avanzadas.

2.1. Registros

A continuación se listan los registros de la arquitectura que pueden ser usados en un determinado programa:

- Registro AX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

AH	AL
8 bits	8 bits

- Registro BX: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro base). Puede ser usado como dos subregistros de 8 bits:

BH	BL
8 bits	8 bits

- Registro CX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

CH	CL
8 bits	8 bits

- Registro DX: registro de 16 bits, de propósito general. Puede ser usado como dos subregistros de 8 bits:

DH	DL
8 bits	8 bits

- Registro SI: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro índice)
- Registro DI: registro de 16 bits, de propósito general y direccionamiento indirecto por registros (registro índice)
- Registro SP: (stack pointer) registro de 16 bits, indica la posición tope del stack.
- Registro BP: (base pointer) registro de 16 bits, indica la posición base del frame de la subrutina. Puede ser usado para direccionar indirectamente el stack.

2.2. Variables

El assembly de la arquitectura x86 de 16 bits soporta dos tipos de datos principales: tipo byte, de 8 bits representado por el símbolo `db` y tipo word, de 16 bits representado por el símbolo `dw`. La declaración de variables en el assembly tiene la siguiente sintaxis:
Identificador Tipo Valor.

Adicionalmente el assembly soporta definición de arreglos, los cuales pueden ser tanto de tipo byte como de tipo word. La sintaxis para declarar arreglos es la siguiente:
Identificador Tipo Valor1, Valor2, Valor3,...

Es importante analizar la relación entre las variables y su almacenamiento en memoria. Suponiendo que se han declarado dos variables: `var1 db 0x0A` y `var2 dw 0x07D0`, y que las direcciones asociadas a las variables son: var1 = 100 y var2 = 101, y se han declarado dos arreglos: `arr1 db 0x01, 0x02, 0x03` y `arr2 dw 0x0A0B, 0x0C0D` , y que las direcciones asociadas a los arreglos son: arr1 = 103 y arr2 = 106, el estado de la memoria sería el siguiente

Variable	Dirección (16 bits)	Palabra (8 bits)
var1	100	0x0A
var2	101	0xD0
	102	0x07
arr1	103	0x01
	104	0x02
	105	0x03
arr2	106	0x0B
	107	0x0A
	108	0x0D
	109	0x0C

De esta tabla de memoria se pueden observar varios elementos relevantes:

- El tamaño de las palabras de memoria es de 8 bits (1 byte).
- Las variables de tipo word se guardan en dos palabras de memoria en orden **little endian**.
- El tamaño de las direcciones de memoria es de 16 bits (2 bytes).

A continuación se presenta un ejemplo de programación en assembly ocupando los registros y variables antes descritos. Es importante tomar en cuenta que las instrucciones de copia entre registro y memoria permiten copiar variables tanto de tipo byte como de tipo word de manera directa, pero se debe tener en consideración el registro que se utiliza para la carga: registros de 8 bits para variables de tipo byte (como AH, AL) y registros de 16 bits para variables de tipo word (como AX).

Ejemplo Multiplicación: Código Java

```java
public static void mult()
{
    int a = 10;
    int b = 200;
    int res = 0;
    while(a > 0)
    {
        res += b;
        a--;
    }
    System.out.println(res);
}
```

Ejemplo Multiplicación: Código Assembly x86

```
;Calculo de la multiplicacion res = a*b

MOV AX, 0
MOV CX, 0
MOV DX, 0

MOV CL, a          ;CL guarda el valor de a
MOV DL, b          ;DL guarda el valor de b

start:
CMP CL, 0            ;IF a <= 0 GOTO end
JLE endprog

ADD AX, DX           ;AX += b

DEC CL               ;a--
JMP start

endprog:

MOV res, AX        ;res = AX

RET

a         db 10
b         db 200
res       dw 0
```

2.3. Instrucciones avanzadas

La arquitectura x86 tiene un ISA CISC, y por tanto implementa instrucciones más complejas que pueden simplificar la programación. En particular en la arquitectura de 16 bits vienen implementadas las instrucciones `MUL op` y `DIV op` las cuales implementan las operaciones de multiplicación y división entera respectivamente, ocupando el registro AX como operando y resultado, de la siguiente forma: `MUL op =>AX = AL*op` y `DIV op =>AL = AX/op`.

Con estas instrucciones se puede reimplementar el código de la multiplicación visto previamente de una manera mucho más simple:

```
;Calculo de la multiplicacion res = a*b

MOV AX, 0

MOV AL, a           ;AL = a
MUL b               ;AX = AL*b

MOV res, AX         ;res = AX

RET

a         db 10
b         db 200
res       dw 0
```

Otra instrucción que provee el ISA es la instrucción `LEA` Load Effective Address. El propósito de esta instrucción es poder obtener la dirección en donde efectivamente está almacenada una determinada variable o arreglo. Esto es más complejo de determinar en la arquitectura x86 que en el computador básico debido a que esta arquitectura soporta la presencia de múltiples programas, y por tanto la ubicación de un determinado programa no se sabe necesariamente antes de ejecutarlo.

La sintaxis de la instrucción es la siguiente `LEA reg, var`, la cual está copiando la dirección de la variable `var` en el registro de 16 bits `reg`. A continuación se muestra un ejemplo en que se usa la instrucción para recorrer un arreglo al calcular el promedio:

Ejemplo Promedio: Código Java

```
public static void promedio()
{
    int[] arreglo = new int[]{6,4,2,3,5};
    int n = 5;
    int i = 0;
    int promedio = 0;

    while(i < n)
    {
        promedio += arreglo[i];
        i++;
    }
    promedio /= n;
    System.out.println(promedio);
}
```

Ejemplo Promedio: Código Assembly x86

```
;Calculo del promedio de un arreglo

MOV CL, n           ;CL tendra el valor del tamano completo del arreglo
MOV SI, 0           ;SI se usara para iterar sobre el arreglo

MOV AX, 0           ;AL se usara para guardar el promedio
```

```
    MOV DX, 0           ;DL se usara para guardar los valores del arreglo

    start:              ;Loop principal: recorrer el arreglo completo
    CMP SI, CX
    JGE endprog

    LEA BX, arreglo     ;En BX se almacena posicion inicial del
                        ;arreglo
    MOV DL, [BX + SI]   ;Se almacena en AL el primer valor que indica
                        ;las repeticiones
    ADD AL, DL          ;Se acumula el valor del arreglo

    INC SI              ;Se incrementa el contador del loop principal
    JMP start
    endprog:

    DIV n
    MOV promedio, AL

    RET

    n        db 5
    arreglo  db 6,4,2,3,5
    promedio db 0
```

2.4. Subrutinas

El manejo de subrutinas en la arquitectura x86 es más complejo que en el computador básico, principalmente porque el stack (i.e. el segmento inferior de la memoria) tiene una mayor relevancia, y es usado para guardar más que solamente la dirección de retorno (como es el caso del computador básico).

La estructura general de un stack luego de llamar e inicializar una subrutina en la arquitectura x86 es la siguiente:

SP →	Variables locales
BP →	Base Pointer anterior a la llamada
	Dirección de retorno
	Parámetros de la subrutina

Se puede observar que en el stack se está almacenando:

- Los parámetros de la subrutina: a diferencia del computador básico en que se ocupaban variables para el paso de parámetros, en la arquitectura x86 se ocupa el stack, ocupando la instrucción `PUSH` antes del llamado para agregarlos.
- La dirección de retorno: al igual que en el computador básico, en la arquitectura x86 al momento de ejecutar la instrucción `CALL` se almacena la dirección de retorno (la que viene después de la llamada) en el stack, para al momento de retornar de la subrutina saber adónde se debe ir.
- El valor anterior del base pointer: es importante en el caso de una secuencia de llamadas anidadas (más adelante se verá por qué).
- Las variables locales: la arquitectura x86 agrega este concepto de variables locales, variables que sólo viven en el scope de la subrutina, las cuales son definidas en el stack, y por tanto dejarán de ser accesibles luego del retorno de la subrutina.

Adicionalmente, la forma en que son manejados los parámetros y el retorno en la arquitectura x86 es especial, y tiene distintas variaciones. Cada una de estas variaciones se denomina una **convención de llamada**, en nuestro caso ocuparemos la convención de Windows, que se denomina **stdcall**. Esta convención especifica lo siguiente:

- Los parámetros son pasados de derecha e izquierda
- El retorno se almacenará en el registro AX
- La subrutina se debe encargar de dejar el SP apuntando en la misma posición que estaba antes de pasar los parámetros.

Considerando estos dos elementos, la secuencia completa de pasos que se debe realizar para llamar a una subrutina es la siguiente:

1. Estando en la parte del programa que va a llamar a la subrutina:

   a) Paso de parámetros: se deben agregar al stack los parámetros que se necesiten en la subrutina, ocupando la instrucción `PUSH`. Es importante señalar que esta instrucción sólo permite agregar valores de 16 bits, es decir 16 bits es la unidad mínima de dato del stack.

   b) Llamada a la subrutina: ocupando la instrucción `CALL` se llama a la subrutina, lo que almacena en el stack la dirección de retorno, y ejecuta el salto a la dirección de la subrutina.

2. Estando dentro de la subrutina:

   a) Guardar el valor actual del base pointer (BP) en el stack, y cargar el valor del stack pointer en el base pointer, lo que se implementa con la secuencia de instrucciones: `PUSH BP` y `MOV BP, SP`

Opcional En caso de ocupar variables locales, se debe reservar el espacio para estas, moviendo el SP n posiciones hacia arriba, donde n es el número de bytes que se ocupará para las variables. Para esto se utiliza la instrucción `SUB SP, n`

   b) Ejecución de la subrutina: se ejecuta lo que corresponda. Para acceder a los parámetros se ocupa direccionamiento mediante el registro BP: el primer parámetro estará en la dirección BP + 4 (dirección del base pointer + 4 bytes), el segundo parámetro en la dirección BP + 6, y así. Para acceder a las variables locales, también se ocupa direccionamiento con el registro BP pero con offsets negativos (las variables locales están «arriba» de donde está apuntando el BP). Por ejemplo se puede acceder a una primera variable de 16 bits, apuntando a la dirección BP - 2.

   c) Al finalizar la ejecución de la subrutina se debe primero recuperar el espacio asignado a las variables locales, bajando el SP con un llamado a la instrucción `SUB SP, n` donde n es el mismo número de bytes asignados al comienzo.

   d) Luego se debe rescatar el valor previo del BP, con la instrucción `POP BP`

   e) Finalmente se debe retornar, pero para que el SP quede ubicado al final de la memoria, es necesario indicarle que se mueva al espacio ocupado por los parámetros, para lo cual se le agrega un parámetro a la instrucción de retorno de la siguiente forma: `RET n` donde n indica el número de bytes ocupados por los parámetros.

A continuación se muestran dos ejemplos de subrutinas. Ambas implementan una subrutina de potencia ($base^{exponente}$), la primera sin variables locales, mientras que la segunda lo hace con variables locales:

Ejemplo Potencia sin variables locales: Código Java

```java
public static int potencia(int base, int exp)
{
  int res = 1;
  while(exp > 0)
  {
    res *= base;
    exp--;
  }
  return res;
}
```

Ejemplo Potencia sin variables locales: Código Assembly x86

```
;Calculo de la potencia pow = base*exp

MOV BL, exp
MOV CL, base
PUSH BX             ;Paso de parametros (de derecha a izquierda)
PUSH CX
CALL potencia       ;potencia(base,exp)
MOV pow, AL         ;Retorno viene en AX
RET

potencia:           ;Subrutina para el calculo de la potencia
PUSH BP
MOV BP, SP          ;Actualizamos BP con valor del SP

MOV CL, [BP + 4]    ;Recuperamos los dos parametros
MOV BL, [BP + 6]

MOV AX, 1           ;AX = 1

start:
CMP BL,0            ;if exp <= 0 goto endpotencia
JLE endpotencia

MUL CL              ;AX = AL * base

DEC BL              ;exp--
JMP start
endpotencia:

POP BP
RET 4               ;Retornar, desplazando el SP en 4 bytes

base      db 2
exp       db 7
pow       db 0
```

Ejemplo Potencia con variables locales: Código Java

```java
public static int potencia(int base, int exp)
{
  int res = 1;
  int i = 0;
  while(i < exp)
  {
    res *= base;
    i++;
  }
  return res;
}
```

Ejemplo Potencia con variables locales: Código Assembly x86

```
;Calculo de la potencia pow = base*exp

MOV BL, exp
MOV CL, base

PUSH BX           ;Paso de parametros (de derecha a izquierda)
PUSH CX

CALL potencia     ;potencia(base,exp)

MOV pow, AL     ;Retorno viene en AX

RET

potencia:         ;Subrutina para el calculo de la potencia

PUSH BP
MOV BP, SP        ;Actualizamos BP con valor del SP
SUB SP, 2         ;Reservamos espacio para variable i

MOV CL, [BP + 4]  ;Recuperamos los dos parametros
MOV BL, [BP + 6]

MOV AX, 1         ;AX = 1
MOV [BP - 2],0

start:
CMP [BP - 2],BL   ;if i >= n goto endpotencia
JGE endpotencia

MUL CL            ;AX = AL * base

INC [BP - 2]      ;i++
JMP start

endpotencia:

ADD SP, 2
```

```
    POP BP
    RET 4               ;Retornar, desplazando el SP en 4 bytes

    base      db 2
    exp       db 7
    pow       db 0
```

Paso de parámetros por valor y por referencia

Otro elemento importante que se debe considerar al momento de ocupar subrutinas es la idea de paso de valores por referencia o por valor. En el ejemplo anterior el paso de parámetros es por valor: se está guardando una copia del valor de la variable en el stack. Para implementar el paso por referencia debemos entregarle como parámetro a la subrutina no el valor, sino la dirección de la variable, la cual se puede obtener con la instrucción `LEA`. Para trabajar con referencias, es necesario además que la subrutina considere esto, y acceda a los valores direccionando de manera indirecta, como se observa en el siguiente ejemplo de la misma subrutina anterior que se muestra a continuación.

```
;Calculo de la potencia pow = base*exp

LEA BX, exp         ;Paso por referencia: cargamos las direcciones
LEA CX, base        ;de ambas variables

PUSH BX             ;Paso de parametros (de derecha a izquierda)
PUSH CX

CALL potencia       ;potencia(&base,&exp)

MOV pow, AL      ;Retorno viene en AX

RET

potencia:           ;Subrutina para el calculo de la potencia

PUSH BP
MOV BP, SP          ;Actualizamos BP con valor del SP

MOV SI, [BP + 4]    ;Recuperamos los dos parametros, que
MOV BX, [BP + 6]    ;almacenan direcciones, no valores

MOV AX, 1           ;AX = 1

start:
CMP [BX],0          ;if exp <= 0 goto endpotencia
JLE endpotencia

MUL [SI]            ;AX = AL * base

DEC [BX]            ;exp--
JMP start

endpotencia:

POP BP
```

```
    RET 4               ;Retornar, desplazando el SP en 4 bytes

    base      db 2
    exp       db 7
    pow       db 0
```

Al ejecutar esta subrutina la diferencia principal es que ahora el valor que se está decrementando en el ciclo de la potencia es directamente el valor de la variable `exp` y no una copia.

Recursión

Esta estructuración de las llamadas a subrutinas nos permite directamente implementar subrutinas recursivas. Si se cumple con la secuencia de operaciones antes descritas, la recursión funcionará correctamente. Lo único a tener en cuenta es como ir acumulando los resultados en el registro AX que será el retorno de cada paso recursivo. A continuación se muestran dos ejemplos de subrutinas recursivas, implementando la función factorial y la serie de Fibonacci:

Ejemplo Factorial: Código Java

```
public static int factorial(int n)
{
  if(n==0)
    return 1;
  return n*factorial(n-1);
}
```

Ejemplo Factorial: Código Assembly x86

```
  MOV BL, n
  PUSH BX             ;Paso de parametros (de derecha a izquierda)
  CALL factorial
  MOV fact, AL      ;Retorno
  RET

  factorial:
  PUSH BP
  MOV BP, SP
  MOV BL, [BP + 4]    ;Recuperamos parametro

  CMP BL, 0           ;if n== 0 goto set1
  JE  set1
  MOV CL, BL          ;Parametro para llamada fact(n-1)
  DEC CL
  PUSH CX
  CALL factorial      ;Paso recursivo
  MOV BL, [BP + 4]    ;Recuperamos el parametro n original
  MUL BL              ;AX = AL*n
  JMP endfact

  set1:               ;Caso base de recursion
  MOV AX,1
  endfact:

  POP BP
  RET 2               ;Retorno desplazando 2 bytes (1 parametro)

  n         db 3
  fact      db 0
```

Ejemplo Fibonacci: Código Java

```
public static int fibonacci(int n)
{
  if(n==0)
```

```
    return 1;
  if(n==1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Ejemplo Fibonacci: Código Assembly x86

```
MOV BL, n
PUSH BX              ;Paso de parametros (de derecha a izquierda)
CALL fibonnaci
MOV fib, AL        ;Retorno
RET

fibonnaci:
PUSH BP
MOV BP, SP

MOV BL, [BP + 4]     ;Recuperamos parametro

CMP BL, 0            ;if n== 0 goto set1
JE  set1

CMP BL, 1            ;if n== 1 goto set1
JE  set1

MOV CL, BL           ;Parametro para llamada fib(n-1)
DEC CL
PUSH CX
CALL fibonnaci       ;Primera llamada recursiva
MOV BL, [BP + 4]     ;Recuperamos valor n original
MOV CL, BL           ;Parametro para llamada fib(n-2)
SUB CL, 2
PUSH CX
CALL fibonnaci       ;Segunda llamada recursiva
JMP endfib
set1:                ;Caso base de recursion
ADD AL, 1            ;Acumulamos 1 en AL por cada vez que llegamos
endfib:              ;al caso base

POP BP
RET 2                ;Retorno desplazando en 2 bytes (1 parametro)

n         db 3
fib       db 0
```