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

IIC2343 – Arquitectura de Computadores

Arquitectura x86

Profesor: Hans Löbel

Arquitectura x86 es de las más utilizadas en la actualidad

- El conocimiento de esta arquitectura resulta fundamental para aplicaciones complejas.
- Presenta diferencias claves con la arquitectura del computadores básico.
- Puede definirse como Von Neuman + CISC.
- Veremos la versión de 16 bits, usada hasta el 286 (386 es de 32 bits).
- Actualmente se utiliza la ISA x86-64, propuesta y popularizada por:

AMD

Veamos primero la microarquitectura

- 4 registros de propósito general de 16 bits (AX, BX, CX, DX), divisibles en sectores altos y bajos (AX = AH | AL) (¿Por qué?).
- BX se utiliza además para el direccionamiento indirecto (registro base).
- 2 registros de 16 bits para uso general y direccionamiento indirecto (SI, DI), usados como registro índice.
- Instruction pointer (IP), stack pointer (SP) y base pointer (BP), todos de 16 bits.

Veamos primero la microarquitectura

- Unidad de control microcode (CISC).
- Sólo una ALU como unidad de ejecución (FPU se introdujo en el 486).
- 6 condition codes: Z, S, C, O, P, A.
- Direcciones de memoria de 16 bits.
- Palabras de memoria de 8 bits.
- Stack en memoria, SP apunta al último elemento ingresado al stack.

ISA x86 tiene algunas similitudes con la del computador básico

- Instrucciones de transferencia, aritméticas, lógicas, saltos, subrutinas.
- Tipos de datos nativos de 8 y 16 bits, con y sin signo.
- Múltiples tipos de direccionamiento:
  - directo
  - indirecto por reg.
  - indirecto por reg. base y offset
  - indirecto con reg. base y reg. Índice
  - indirecto con reg. base, reg. índice y offset

Definición y uso de variables es distinto en assembly x86 16 bits

- Soporta 2 tipos: *byte* de 8 bits y *word* de 16 bits
- Son representados por db (byte) y dw (word)
- Arreglos también pueden ser de estos tipos
- Datos son almacenados en *little endian.*
- Todo esto implica que manejo de memoria requiere mayor cuidado
- Instrucción LEA *reg, var* nos permite almacenar en el registro *reg* la dirección de la variable var

¿Cómo queda la memoria luego de declarar variables?

- Existen 4 variables declaradas a partir de la dirección 0:
  - var1 db 0x0A y var2 dw 0x07D0
  - arr1 db 0x01, 0x02, 0x03 y arr2 dw 0x0A0B, 0x0C0D

Variable	Dirección (16 bits)	Palabra (8 bits)
var1	00	0x0A
var2	01	0xD0
	02	0x07
arr1	03	0x01
	04	0x02
	05	0x03
arr2	06	0x0B
	07	0x0A
	08	0x0D
	09	0x0C

Ejemplo Multiplicación: Código 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
```

ISA x86 aprovecha ventajas de CISC

- Arquitectura CISC permite incluir instrucciones aritméticas complejas como MUL y DIV, que utilizan varios ciclos:

MUL *op* => AX = AL × *op*

MUL *op* => DX|AX = AX × *op*

DIV *op* => AL = AX ÷ *op*

DIV *op* => AX = DX|AX ÷ *op*

```
;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
```

Uso de subrutinas en x86 presenta mayor complejidad que en el computador básico

- En nuestro computador, el stack almacenaba la dirección de retorno.
- No había una convención sobre donde almacenar los parámetros y valores de retorno.
- En x86, utilizaremos el stack de manera más explícita: parámetros, retorno, variables locales.
- Uso del registro BP (base pointer) es fundamental para facilitar el manejo de todos estos datos.

Uso de subrutinas requiere la definición de convenciones de llamada (*calling conventions*)

- Las convenciones definen la interfaz sobre la cual trabajará el código de la subrutina.
- Una convención de llamada debe especificar lo siguiente:
  - Donde se encuentran los parámetros (stack, registros o una mezcla de ambos).
  - Si se usa el stack, el orden en que los parámetros son entregados.
  - Definición de responsabilidades de restauración del stack, entre la subrutina y el código que la llama.

Uso de subrutinas requiere la definición de convenciones de llamada (*calling conventions*)

- Existen múltiples convenciones: *stdcall*, *cdecl*, *fastcall, safecall, syscall, thiscall,…*
- Se dividen entre las que asignan la responsabilidad de limpieza del stack a la subrutina, y las que se la asignan al código que llama a la subrutina.
- Ocuparemos la convención *stdcall*, que es la usada por la API Win32 de Microsoft.

stdcall deposita en la subrutina la responsabilidad de limpiar el stack

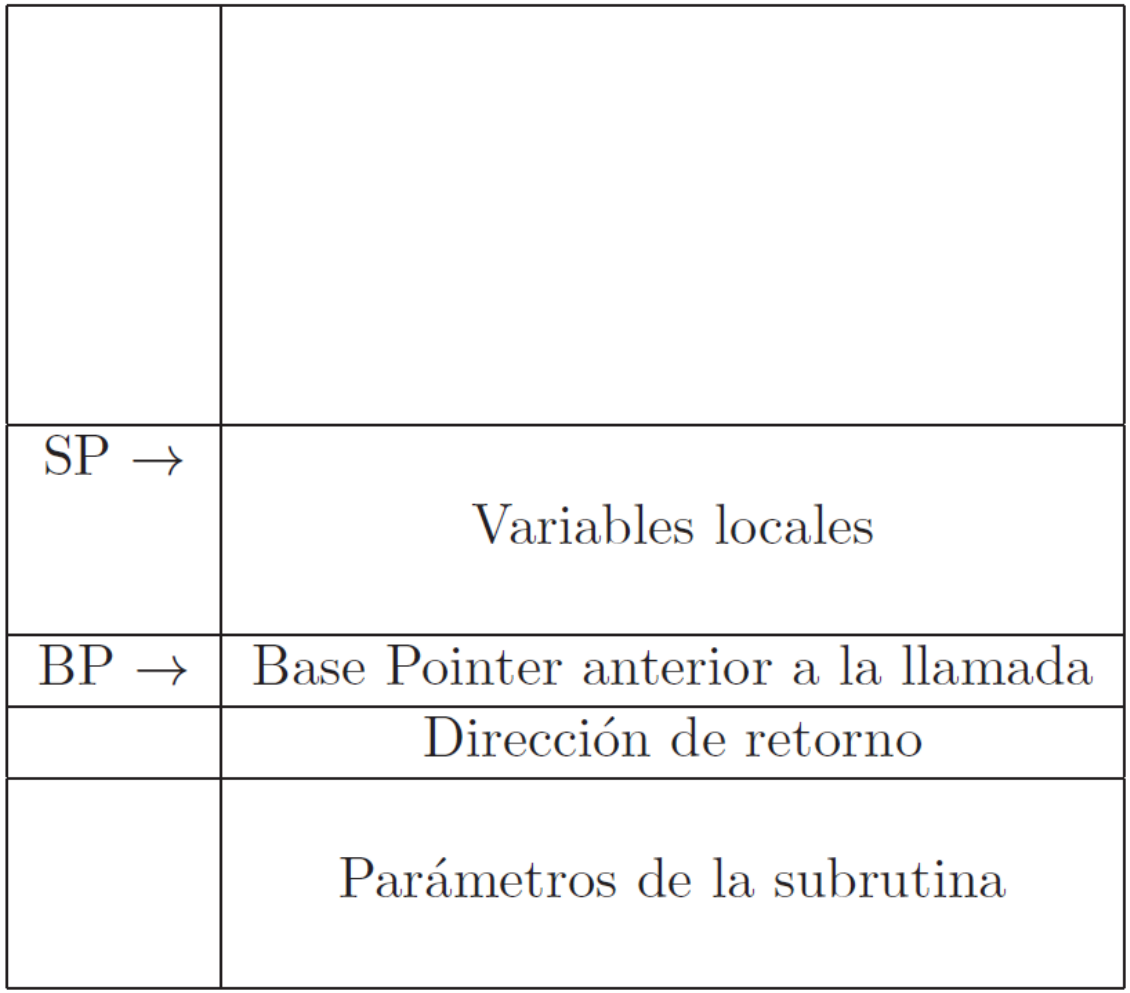
La convención *stdcall* especifica los siguientes 3 puntos:

1. Los parámetros son pasados de derecha a izquierda, usando el stack.

2. El retorno se almacenará en el registro AX

3. La subrutina se debe encargar de dejar SP apuntando en la misma posición que estaba antes de pasar los parámetros.

En *stdcall*, SP y BP permiten tener llamadas anidadas de subrutinas (recursión) y variables locales



Se deben ejecutar 2 pasos al momento de llamar a una subrutina

1. Agregar los parámetros al stack usando la instrucción PUSH. Los valores agregados sólo pueden ser de 16 bits.

2. Llamar a la subrutina con la instrucción CALL, lo que almacena en el stack la dirección de retorno y ejecuta el salto a la dirección de la subrutina.

Dentro de la subrutina, son 5 los pasos a ejecutar

1. Guardar el valor actual de BP en el stack y cargar el valor de SP en BP:

   ```
   PUSH BP
   MOV BP, SP
   ```

   En caso de usar variables locales, se debe reservar espacio para estas, moviendo SP *n* posiciones hacia arriba, donde *n* es el número de palabras de memoria que usan las variables:

   ```
   SUB SP, n
   ```

Dentro de la subrutina, son 5 los pasos a ejecutar

2. Ejecutar la subrutina. Para acceder a los parámetros se usa direccionamiento mediante BP. Así, el primer parámetro estará en BP+4, el segundo en BP+6, etc.

   De la misma manera, las variables locales se acceden usando BP, pero con offset negativo, BP-2, etc.

3. Al finalizar la subrutina se debe recuperar el espacio de las variables locales:

   ADD SP, n

Dentro de la subrutina, son 5 los pasos a ejecutar

4. Rescatar el valor previo de BP:

   POP BP

5. Mover SP al valor previo al paso de los parámetros:

   RET *n*

   donde *n* indica la cantidad de palabras de memoria usadas por los parámetros.

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

IIC2343 – Arquitectura de Computadores

Arquitectura x86

Profesor: Hans Löbel