Funciones y Datos en Intel 8086

Departamento de Arquitectura[1]

[1]Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Arquitectura de Computadoras, 2016

Temas

1. Variables
2. Tipos de datos
   - Tipos atómicos
   - Tipos estructurados
   - Tipos compuestos
3. Funciones
   - Instrucciones y directivas
   - Invocación
   - Contexto de Ejecución
   - Pasaje de Parámetros
4. Recursión

Introducción

Por definición, una variable es *un espacio en memoria con un nombre asignado*.
En alto nivel todos conocemos el concepto de variable. En assembler, esto se mapea a posiciones de memoria con una etiqueta.
El tipo de una variable de alto nivel se refleja en el tamaño que ocupan en memoria.
Tampoco hay new o delete, son funciones de alto nivel.

Tipos en memoria

Memoria:

0x00000	
…	
0x3f302	0x37
0x3f303	0x24
0x3f304	0x7A
0x3f305	0x0B
0x3f006	0x43
0x3f307	0x23
0x3f308	0xBA
0x3f309	0x11
0x3f00A	0x19
…	
0xFFFFF	

Interpretada como		
CHAR	SHORT	FLOAT
“7”	9271	4.82E-32
“$”		
“z”	2938	
VT		
“C”	9027	2.94E-28
“#”		
“º”	4538	
DC1		
EM	…	…

¿Cómo se compila?

Las variables siempre tendrán una posición de memoria asignada. En base a su alcance, decidiremos si almacenarla en memoria, stack o un registro.

Alcance

Una variable con un alcance (*scope*) corto, podremos ubicarla únicamente en un registro.
Una variable global, sí o sí debe tener un espacio en memoria principal.
Veremos más sobre esto al ver funciones.

Programación bajo nivel

Impasse
Hay "malas prácticas"de programación en alto nivel que son **necesarias** en bajo nivel.

- while (true)
- Variables globales
- goto/jmp

Es importante tener en cuenta la diferencia de herramientas y objetivos!

Introducción

A pesar de que varios de los conceptos vertidos aquí son de alcance general, nos restringiremos a la arquitectura Intel 8086.

Tipos Básicos

- char (1 byte)
- short y unsigned short (2 bytes)

Punteros

Los punteros en Intel 8086 pueden ser:

- en el mismo segmento (2 bytes para el desplazamiento)
- en otro segmento (4 bytes, 2 desplazamiento y 2 segmento)

Usualmente trabajaremos con punteros en el mismo segemento.

Compilar Arreglos

Para compilar un arreglo en Intel, disponemos un elemento del arreglo a continuación del otro.
Cada elemento del arreglo ocupa lo que ocupa el tipo de datos del arreglo.

Manipular arreglos

Sumar 10 elementos de un arreglo de enteros que comienza en `[BX]`

Ejemplo

```
  MOV SI, 0
  MOV AX, 0
  MOV CX, 0
evaluar:
  CMP CX, 10
  JE fin
  ADD AX, [BX+SI]
  ADD SI, 2 ; Tamaño elem
  INC CX ; Posición
  JMP evaluar
fin: ...
```

Arreglo en memoria

Memoria:

0x00000	
...	
0x3f302	0x37
0x3f303	0x24
0x3f304	0x7A
0x3f305	0x0B
0x3f006	0x43
0x3f307	0xFE
0x3f308	...
0x3f309	...
0x3f00A	...
...	
0xFFFFF	

short arreglo[3]	
arreglo[0]	9271
arreglo[1]	2938
arreglo[2]	-445

Compilar Structs

Para compilar un struct en Intel, disponemos un elemento del struct continuación del otro en el orden que son declarados.
El struct ocupa la suma de lo que ocupan sus elementos.

Manipular Structs

Acceso a los componentes de un struct ubicado en [BX]

```
struct ejemplo{
  short x, y;
  char a, b;
};
// ocupa 6 bytes
```

```
; si BX apunta al incio
  mov AX, [BX] ; x
  mov AX, [BX+2] ; y
  mov AX, [BX+4] ; a
  mov AX, [BX+5] ; b
```

Manipular Structs

Acceso a los componentes de un struct ubicado en [BX]

```
struct ejemplo{
short x, y;
char a, b;
};
// ocupa 6 bytes
```

```
; si BX apunta al incio
mov AX, [BX] ; x
mov AX, [BX+2] ; y


mov AX, [BX+4] ; a
mov AX, [BX+5] ; b
```

Struct en memoria

Memoria:

0x00000	
...	
0x3f302	0x37
0x3f303	0x24
0x3f304	0x7A
0x3f305	0x0B
0x3f006	0x43
0x3f307	0x40
0x3f308	...
0x3f309	...
0x3f00A	...
...	
0xFFFFF	

struct ejemplo{ short x, y; Char a,b ; }	
ejemplo.x	9271
ejemplo.y	2938
ejemplo.a	'C'
ejemplo.b	'@'

Compilar Tipos Compuestos

Nos referimos como tipos compuestos aquellos que utilizan más de un tipo estructurado en su definición.
Para compilarlos, seguiremos las pautas planteadas para cada uno de los tipos, teniendo en cuenta únicamente que sus componentes no son tipos básicos.

Manipular Tipos Compuestos

Sumar 10 componentes enteros de un arreglo de structs que comienza en [BX]

```
typedef struct{
  char a;
  short x;
} ejemplo;
ejemplo arreglo[10];

; arreglo
  MOV SI, 0
  MOV AX, 0
  MOV CX, 0
  evaluar:
  CMP CX, 10
  JE fin
  ADD AX, [BX+SI+1]
  ADD SI, 3 ; Tamaño elem
  INC CX ; Posición
  JMP evaluar
fin: ...
```

Ejemplo en memoria

Memoria:

0x00000	
...	
0x3f302	0x43
0x3f303	0x37
0x3f304	0x24
0x3f305	0x40
0x3f006	0x7A
0x3f307	0x0B
0x3f308	...
...	
0xFFFFF	

```
typedef struct{
char a;
short x;
} ejemplo;
ejemplo arreglo[10];
```

Arreglo[0].a	'C'
Arreglo[0].x	9271
Arreglo[1].a	'@'
Arreglo[1].x	2938
...	...

Introducción

Como la mayoría de las arquitecturas, Intel 8086 incluye un mecanismo para implementar funciones y llamados.
Difiere de un flujo condicional en cuanto a que se retorna luego de un llamado.

Llamado Funciones

¿Cómo es la secuencia de pasos en este programa?

Ejemplo

```
int sumar(int a, int b){
  return a+b;
}

int main(){
  ...
  int dato = sumar(4,5);
  ...
}
```

Llamado Funciones

1. Pasar parámetros 4 y 5
2. Saltar a la función suma
3. Ejecutar la función suma
4. Pasar resultado a devolver
5. Volver al punto de invocación
6. Asignar a dato el resultado
7. ...

Pasaje de parámetros

Al llamar una función, es posible pasar los parámetros mediante dos formas:

1. Por REGISTRO. Se cargan los parámetros en registros antes de hacer el llamado.
2. Por STACK. Se hace PUSH de los parámetros antes de hacer el llamado.

El retorno del **resultado** es análogo. Veremos esto más adelante.

Stack

El mecanismo para el llamado de funciones en Intel es naturalmente **el stack**.
Es la herramienta que provee la arquitectura para guardar el contexto del llamador (en particular, la posición actual).

Instrucciones

Las dos instrucciones para implementar llamados a funciones son:

1. CALL: Realiza el llamado.
2. RET: Devuelve el control al llamador.

CALL

CALL permite invocar una rutina definida mediante la directiva PROC.
Para esto, sencillamente realiza el PUSH de la posición actual en la ejecución y salta a la etiqueta.

1. PUSH IP
2. JMP funcion

CALL FAR

Si para invocar la rutina se debe cambiar el valor de CS, el call modifica su comportamiento.
Hace PUSH además del IP, del CS.

1. PUSH CS
2. PUSH IP
3. JMP funcion (nuevo CS e IP)

RET

RET nos permite volver al punto anterior de ejecución.
Para esto, toma del tope del stack la dirección de retorno y vuelve hacia allí.

1. POP IP

Notar que el RET asume que lo que se encuentra en el tope del stack, es la dirección de retorno. Si es *otra cosa* el sistema tendrá un comportamiento desconocido.
¿Qué pasa si hay que retornar de un CALL FAR?

Ejemplo Invocación

Ejemplo

```
int sumar(int a, int b){
  return a+b;
}

int main(){
  ...
  int dato = sumar(4,5);
  ...
}
```

Compilar funcion

Definimos el protocolo de recibir los parámetros en AX y BX respectivamente, devolviendo el resultado en AX

Ejemplo

```
proc SUMAR:
  add AX, BX
  ret
endproc

proc main:
  ...
  mov AX, 4
  mov BX, 5
  call SUMAR
; AX es 9
```

Contexto de Ejecución

¿Qué es el contexto de ejecución?
El contexto de ejecución consiste en el valor de los registros del CPU. Esto incluye los registros de datos así como los registros de estado.
Este concepto también existe en alto nivel y se relaciona con el alcance de una variable.

Alto Nivel

Cuando llamamos una función en código alto nivel, al retornar, nuestras variables valen lo mismo antes y después del llamado.

Ejemplo

```
int sumar(int a, int b){
  // Nada de lo que se haga aqui
  // modifica previoDato
  return a+b;
}
  int dato = sumar(4,5);
  // previoDato == 8
  return;
}
```

¿Qué pasa en bajo nivel?

En Intel, al ejecutar una función el valor de los registros (contexto) no se preserva automáticamente como en alto nivel.

Bajo nivel

Cuando llamamos una función en Intel, al retornar, los registros pueden cambiar.

Ejemplo

```
proc SUMAR:
  xor CX, CX
  add AX, BX
  ret
endproc
```

```
proc main:
  mov CX, 30
  mov AX, 4
  mov BX, 5
  call SUMAR
; AX es 9 y CX es 0!
```

Firmas de funciones en Intel y pasaje de parámetros

Veremos esto en detalle más adelante, pero una función debe especificar como recibe parámetros, cuál es su comportamiento respecto al contexto y dónde devuelve su resultado.

Preservar el contexto

En Intel, el mecanismo para preservar el contexto es el stack.
Intel preserva el contexto mínimo de la función "padre", esto es, el punto de ejecución previo a la llamada (a dónde volver luego).
El programador **debe** preservar manualmente el contexto y restaurarlo, pero es una tarea *opcional*.

Cómo preservar el contexto?

En términos generales, el procedimiento es:

1. Al comenzar la función, almacenar en el stack el valor de todos los registros que se utilizan.
2. Ejecutar el cuerpo de la función
3. Restaurar desde el stack *en orden inverso* los registros.
4. Retornar (el tope del stack debe contener el valor del IP anterior!)

Existen excepciones, por ejemplo si un registro se utiliza para devolver el resultado (como AX en la suma).

No preserva contexto

Ejemplo

```
proc SUMAR:
  xor CX, CX
  add DX, DX
  add AX, BX
  ret
endproc
```

```
proc main:
  mov CX, 30
  mov DX, 40
  mov AX, 4
  mov BX, 5
  call SUMAR
; AX = 9, BX = 5,
; CX = 0, DX = 80
```

Preserva contexto

Ejemplo

```
proc SUMAR:
  push CX
  push DX
  xor CX, CX
  xor DX, DX
  add AX, BX
  pop DX
  pop CX
  ret
endproc
```

```
proc main:
  mov CX, 30
  mov DX, 40
  mov AX, 4
  mov BX, 5
  call SUMAR
; AX = 9, BX = 5,
; CX = 30, DX = 40
```

Introducción

Como ya se mencionó, el pasaje de parámetros y resultados puede hacerse tanto por stack como por registros.
Las funciones pueden utilizar simultáneamente ambos mecanismos.

Pasaje por registros

Al definir una función, podemos especificar los registros utilizados para el intercambio de información.
Para el caso de *sumar* del último ejemplo anterior:

Ejemplo

SUMAR es una función que preserva el contexto, recibe en AX y BX los operandos a sumar y retorna en AX el valor de la suma.

Inconvenientes

El pasaje de parámetros y resultados por registros tiene varios inconvenientes:

1. **Pocos Registros** La cantidad de parámetros a pasar es muy limitada.
2. **Liberar registros para resultado**Para retornar resultados, el llamador debe liberar registros.

Pasaje por stack

El mecanismo más flexible para el pasaje de parámetros y retorno de resultados es el stack.
El esquema básico para el llamado de una función que recibe y devuelve los parámetros por stack es:

1. **PUSH de parámetros** *Antes* de invocar la función, se realiza el push de los parámetros.
2. **Invocación** La rutina invocada puede o no preservar el contexto.
3. **Retrono y POP de resultados** Al retornar al llamador, en el tope del stack se encuentran los resultados.

Especificación pasaje por stack

Al especificar la firma de la función, en el caso de utilizar el stack para el pasaje de parámetros se debe especificar **la cantidad y el orden de los mismos**. Aplica lo mismo para el resultado.
Notar que la cantidad de parámetros y resultados son independientes.

Ejemplo

DIV-ENTERA es una función que preserva el contexto, recibe en el stack DIVIDENDO y DIVISOR con el último en el tope. Devuelve RESTO y COCIENTE con el último en el tope.

Especificación pasaje por stack

Otra manera de expresar lo mismo es abusar un poco la notación...

Ejemplo

```
push DIVIDENDO ; push op1
push DIVISOR ; push op2
call DIV_ENTERA
pop COCIENTE ; pop res1
pop RESTO ; pop res2
```

Accediendo a los parámetros

Al pasar los parámetros por stack, surge la incógnita de cómo los podemos acceder. Notar que **la dirección de retorno** es lo que la rutina tiene en el tope del stack.

Ejemplo

```
...
mov CX, 1705
push CX
mov CX, 6
push CX
call DIV_ENTERA
pop AX ; AX = 284
pop BX ; BX = 1
...
```

Accediendo a los parámetros

1

Instrucción:

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
	0x3f006	...
	0x3f007	
	0x3f008	...
	0x3f009	
SP→	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

SP	0x100A
SS	0x3E00

2

Instrucción: mov CX, 1705

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
	0x3f006	...
	0x3f007	
	0x3f008	...
	0x3f009	
SP→	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

CX	**0x06A9**
SS	0x3E00

3

Instrucción: push CX

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
	0x3f006	...
	0x3f007	
SP→	0x3f008	0x06A9
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

CX	0x06A9
SP	**0x1008**

Accediendo a los parámetros

4

Instrucción: mov CX, 6

Resultado:

	0x00000	
	…	
	0x3f002	…
	0x3f003	
	0x3f004	…
	0x3f005	
	0x3f006	…
	0x3f007	
SP→	0x3f008	0x06A9
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	…	
	0xFFFFF	

CX	**0x0006**
SP	0x1008

5

Instrucción: push CX

Resultado:

	0x00000	
	…	
	0x3f002	…
	0x3f003	
	0x3f004	…
	0x3f005	
SP→	0x3f006	0x0006
	0x3f007	
	0x3f008	0x06A9
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	…	
	0xFFFFF	

CX	0x0006
SP	**0x1006**

6

Instrucción: call DIV_ENTERA

Resultado:

	0x00000	
	…	
	0x3f002	…
	0x3f003	
SP→	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x06A9
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	…	
	0xFFFFF	

IP	**DIV_ENTER**
SP	**0x1004**

Accediendo a los parámetros

DIV-ENTERA ejecuta su contenido y . . .

Accediendo a los parámetros

7

Instrucción: ret (de DIV_ENTERA)

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
SP→	0x3f006	0x011C
	0x3f007	
	0x3f008	0x0001
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

IP	**IP_retorno**
SP	**0x1006**

8

Instrucción: pop AX

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
	0x3f006	...
	0x3f007	
SP→	0x3f008	0x0001
	0x3f009	
	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

AX	0x011C
SP	**0x1008**

9

Instrucción: pop BX

Resultado:

	0x00000	
	...	
	0x3f002	...
	0x3f003	
	0x3f004	...
	0x3f005	
	0x3f006	...
	0x3f007	
	0x3f008	...
	0x3f009	
SP→	0x3f00A	0xAAC3
	0x3f00B	
	...	
	0xFFFFF	

BX	0x0001
SP	**0x100A**

Accediendo a los parámetros con BP

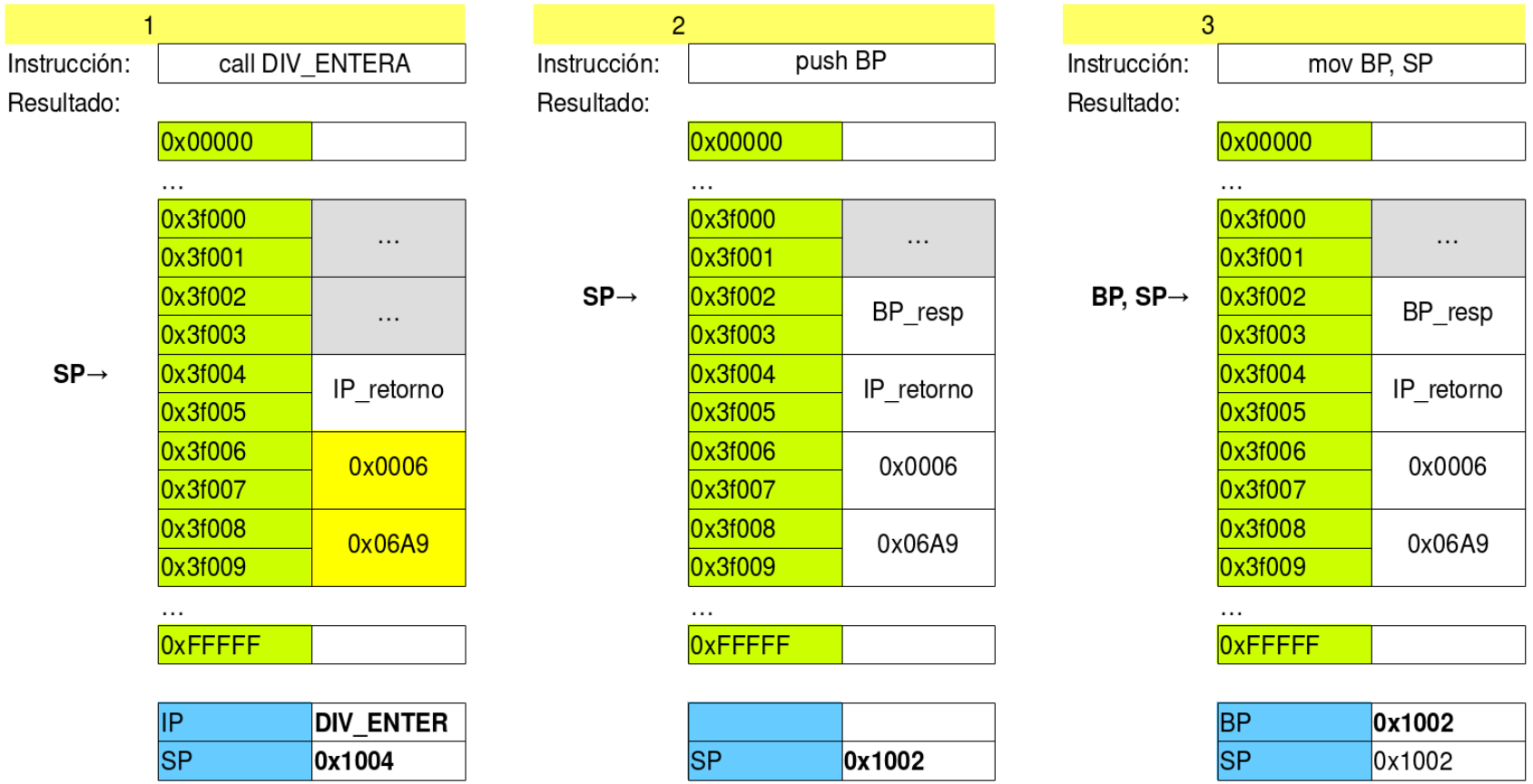
Para poder manipular el stack y sus parámetros, utilizaremos BP. Cargaremos BP con un valor conocido (SP) que nos permita acceder a otras posiciones del stack.

Accediendo a los parámetros con BP

Ejemplo

```
PROC DIV-ENTERA                    ; SS:[BP+4] -> Divisor
  push BP ;respaldo BP             ; SS:[BP+6] -> Dividendo
  mov BP, SP ; BP=SP               ...
  ; SS:[BP] -> BP anterior
  ; SS:[BP+2] -> IP_retorno
```

Accediendo a los parámetros con BP



Accediendo a los parámetros con BP

Ejemplo

```
PROC DIV-ENTERA
  push BP ;respaldo BP
  mov BP, SP ; BP=SP
  push AX
  push DX
  push CX
  mov DX, 0
  mov AX, [BP+6]
  mov CX, [BP+4]
  div CX ; ver cartilla
  ; DX =0x1, AX=0x11C
  mov [BP+6], DX
  mov [BP+4], AX
  pop CX
  pop DX
  pop AX
  pop BP
  ret
ENDPROC
```

Accediendo a los parámetros con BP

Accediendo a los parámetros con BP

7

Instrucción: `mov [BP+6], DX`

Resultado:

	0x00000	
	…	
	0x3f000	AX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	**0x0001**
	0x3f009	
	…	
	0xFFFFF	

DX	0x0001
AX	0x011C

8

Instrucción: `mov [BP+4], AX`

Resultado:

	0x00000	
	…	
	0x3f000	AX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	**0x011C**
	0x3f007	
	0x3f008	0x0001
	0x3f009	
	…	
	0xFFFFF	

DX	0x0001
AX	0x011C

9

Instrucción: `pop CX`

Resultado:

	0x00000	
	…	
	0x3effc	…
	0x3effd	
SP→	0x3effe	DX_resp
	0x3efff	
	0x3f000	AX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	…	
	0xFFFFF	

CX	**CX_resp**
SP	**0x0FFE**

Accediendo a los parámetros con BP

10

Instrucción: pop DX

Resultado:

	0x00000	
	…	
	0x3effc	…
	0x3effd	
	0x3effe	…
	0x3efff	
SP→	0x3f000	AX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	…	
	0xFFFFF	

DX	**DX_resp**
SP	**0x1000**

11

Instrucción: pop AX

Resultado:

	0x00000	
	…	
	0x3f000	…
	0x3f001	
BP, SP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x011C
	0x3f007	
	0x3f008	0x0001
	0x3f009	
	…	
	0xFFFFF	

AX	**AX_resp**
SP	**0x1002**

12

Instrucción: pop BP

Resultado:

	0x00000	
	…	
	0x3f000	…
	0x3f001	
	0x3f002	…
	0x3f003	
SP→	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x011C
	0x3f007	
	0x3f008	0x0001
	0x3f009	
	…	
	0xFFFFF	

BP	**BP_resp**
SP	**0x1004**

Ajuste de stack

Cuando la cantidad de parámetros y de resultados son diferentes, es necesario manipular el stack para ajustarlo, de forma tal que al ejecutar RET se encuentre la dirección de retorno en el tope del stack y debajo estén los resultados.

Ejemplo ajuste stack

Veamos un ejemplo con más parámetros que resultados.

Ejemplo

```
push MINUENDO ; push op1
push SUSTRAENDO ; push op2
call RESTA ; preserva contexto
pop DIFERENCIA ; pop res1 (op1-op2)
```

Ejemplo ajuste stack

Ejemplo

```
PROC RESTA
  push BP ;respaldo BP
  mov BP, SP ; BP=SP
  push CX
  mov CX, [BP+6]
  sub CX, [BP+4]
  mov [BP+6], CX
  mov CX, [BP+2]; CX = dir
  mov [BP+4], CX; [BP+4] = dir
  pop CX
  pop BP
  add SP, 2
  ret
ENDPROC

ret
```

Ejemplo ajuste stack

1

Instrucción: call RESTA

Resultado:

	0x00000	
	...	
	0x3f000	...
	0x3f001	
	0x3f002	...
	0x3f003	
SP→	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

IP	**RESTA**
SP	**0x1004**

2

Instrucción: push BP

Resultado:

	0x00000	
	...	
	0x3f000	...
	0x3f001	
SP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

SP	**0x1002**

3

Instrucción: mov BP, SP

Resultado:

	0x00000	
	...	
	0x3f000	...
	0x3f001	
BP, SP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

BP	**0x1002**
SP	0x1002

Ejemplo ajuste stack

4

Instrucción: push CX

Resultado:

	0x00000	
	...	
SP→	0x3f000	**CX_resp**
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

SP	**0x1000**

5

Instrucción: mov CX, [BP+6]

Resultado:

	0x00000	
	...	
SP→	0x3f000	CX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

CX	**0x0005**
SP	0x1000

6

Instrucción: sub CX, [BP+4]

Resultado:

	0x00000	
	...	
SP→	0x3f000	CX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0x0005
	0x3f009	
	...	
	0xFFFFF	

CX	**0xFFFF**
SP	0x1000

Ejemplo ajuste stack

7

Instrucción: `mov [BP+6], CX`

Resultado:

	0x00000	
	...	
SP→	0x3f000	CX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	**0xFFFF**
	0x3f009	
	...	
	0xFFFFF	

CX	0xFFFF
SP	0x1000

8

Instrucción: `mov CX, [BP+2]`

Resultado:

	0x00000	
	...	
SP→	0x3f000	CX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	0x0006
	0x3f007	
	0x3f008	0xFFFF
	0x3f009	
	...	
	0xFFFFF	

CX	**IP_retorno**
SP	0x1000

9

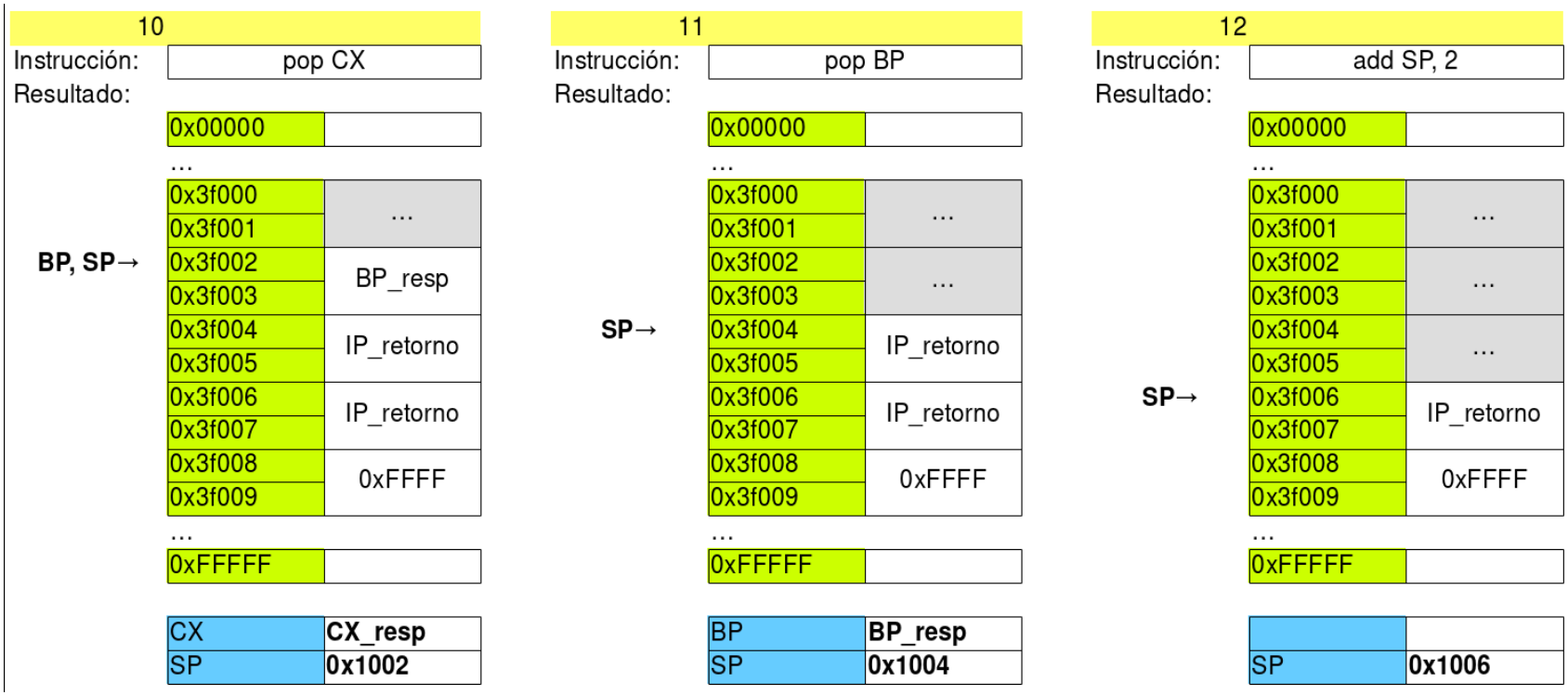
Instrucción: `mov [BP+4], CX`

Resultado:

	0x00000	
	...	
SP→	0x3f000	CX_resp
	0x3f001	
BP→	0x3f002	BP_resp
	0x3f003	
	0x3f004	IP_retorno
	0x3f005	
	0x3f006	**IP_retorno**
	0x3f007	
	0x3f008	0xFFFF
	0x3f009	
	...	
	0xFFFFF	

CX	IP_retorno
SP	0x1000

Ejemplo ajuste stack



Ejemplo ajuste stack

En caso de que se tengan más resultados que parámetros se deben reservar posiciones de stack al inicio de la rutina
Ver examen Febrero 2016

Introducción

Un tipo especial de rutinas, son las rutinas recursivas. Nos interesará sobre las mismas su comportamiento y su consumo de stack.
En particular nos interesa conocer su consumo máximo posible de stack.

Ejemplo recursión

Para estudiar las rutinas recursivas, utilizaremos la función factorial como caso de estudio.

Ejemplo

```
short factorial(short n){
  if (n == 0){
    return 1;
  }
  else {
    return n * factorial (n-1);
  }
}
```

Factorial - Ejemplo 1

Veamos una compilación posible para la función factorial:

Ejemplo

FACT es una función que recibe en AX un número positivo mayor o igual a 0 y devuelve en BX su factorial. No preserva registros.

Factorial - Ejemplo 1

Assembler

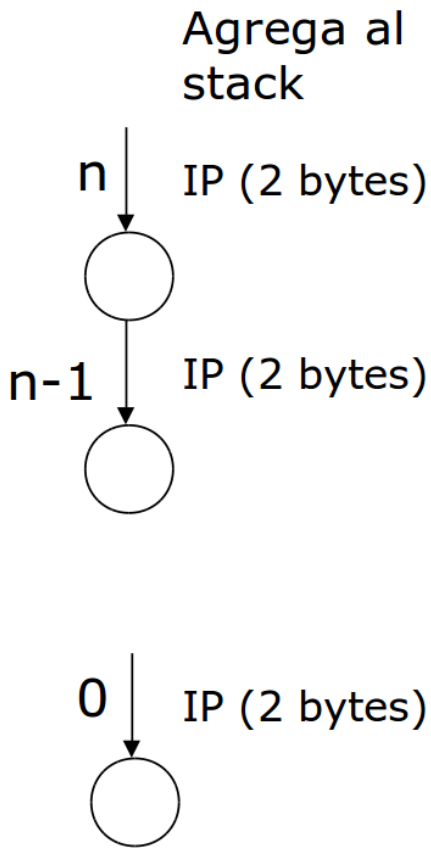
Llamada	Asembler

`mov ax,n`
`call fact`
`mov resultado,bx`
	`fact proc`
	`  cmp ax,0  ; comparo n con cero`
	`  je esCero`
	`  dec ax  ; ajusto parámetro para la invocación`
	`  call fact ; realizo la llamada recursiva`
	`  inc ax`
	`  mov cx,ax ; guardo ax pues mul lo modifica`
	`  mul bx  ; calculo el paso recursivo`
	`  mov bx,ax ; asigno el resultado del paso`
	`  ; recursivo`
	`  mov ax,cx ; restauro ax`
	`  jmp fin`
	`esCero:`
	`  mov bx,1  ; asigno el resultado del paso base`
	`fin:`
	`  ret`
	`fact endp`

Factorial - Ejemplo 1

Consumo

- Planteo de la recurrencia
  - $consumo(0) = 2$
  - $consumo(n) = 2 + consumo(n-1)$
- Resolviendo la recurrencia
  - $consumo(n) = 2 * (n + 1)$



Factorial - Ejemplo 2

Veamos otra compilación posible para la función factorial que utiliza el stack durante la ejecución.

Factorial - Ejemplo 2

Assembler

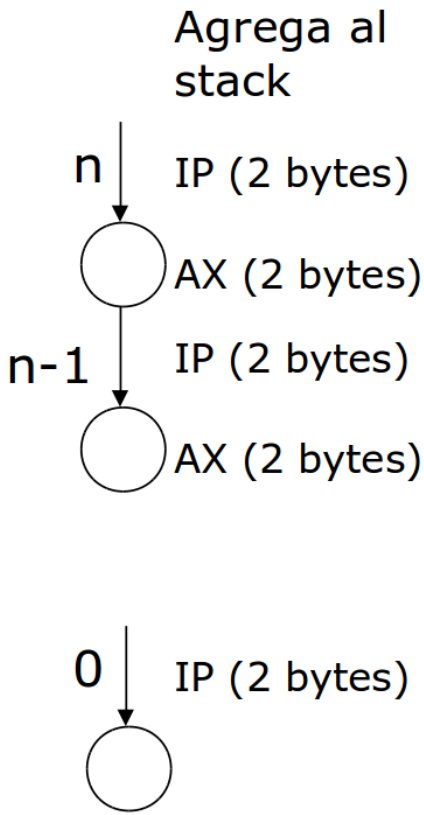
Llamada	Asembler

`mov ax,n`
`call fact`
`mov resultado,bx`
	`fact proc`
	`  cmp ax,0  ; comparo n con cero`
	`  je esCero`
	`  push ax  ; salvo contexto`
	`  dec ax  ; ajusto parámetro para la invocación`
	`  call fact ; realizo la llamada recursiva`
	`  pop ax  ; restauro el contexto`
	`  mul bx  ; calculo el paso recursivo`
	`  mov bx,ax ; asigno el resultado del paso`
	`  ; recursivo`
	`  jmp fin`
	`esCero:`
	`  mov bx,1  ; asigno el resultado del paso base`
	`fin:`
	`  ret`
	`fact endp`

Factorial - Ejemplo 2

Consumo

- Planteo de la recurrencia
  - $consumo(0) = 2$
  - $consumo(n) = 4 + consumo(n-1)$
- Resolviendo la recurrencia
  - $consumo(n) = 4 * n + 2$



Factorial - Ejemplo 3

Veamos otra compilación posible para la función factorial que utiliza el stack para el pasaje de parámetros.

Factorial - Ejemplo 3

Assembler

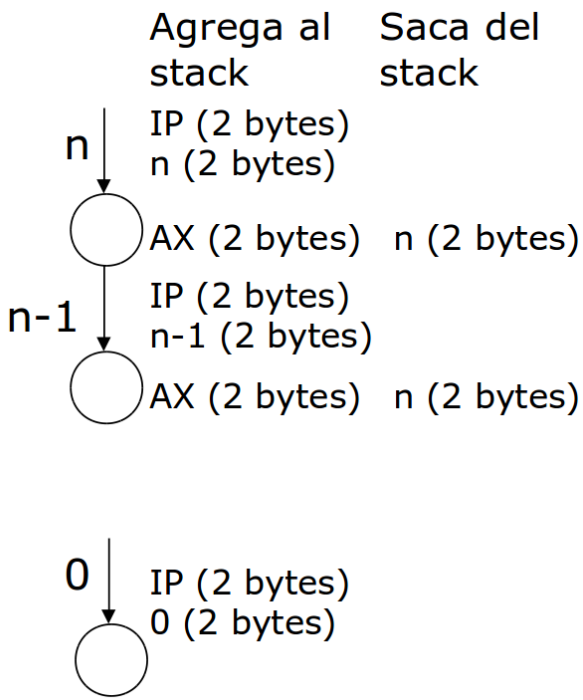
Llamada	Asembler

push n
call fact
pop resultado
	fact proc
	  pop dx ; saco dirección de retorno del stack
	  pop ax ; saco parámetro del stack
	  push dx ; coloco la dirección de retorno en el stack
	  cmp ax,0 ; comparo n con cero
	  je esCero
	  push ax ; guardo el contexto
	  dec ax ; ajusto parámetro para la invocación
	  push ax ; coloco parámetro en el stack
	  call fact ; realizo la llamada recursiva
	  pop bx ; retiro del stack el resultado de fact(n-1)
	  pop ax ; restauro el contexto
	  mul bx ; calculo el paso recursivo
	  mov bx,ax ; asigno el resultado del paso recursivo
	  jmp fin
	esCero:
	  mov bx,1 ; asigno el resultado del paso base
	fin: ; acomodo el stack para el retorno
	  pop dx ; saco dirección de retorno del stack
	  push bx ; coloco el resultado en el stack
	  push dx ; coloco la dirección de retorno en el stack
	  ret
	fact endp

Factorial - Ejemplo 3

Consumo

- Planteo de la recurrencia
  - $consumo(0) = 4$
  - $consumo(n) = 4 + consumo(n-1)$
- Resolviendo la recurrencia
  - $consumo(n) = 4 * n + 4$



Factorial - Ejemplo 4

Veamos otra compilación posible para la función factorial que utiliza el stack para el pasaje de parámetros y preserva el contexto.

Factorial - Ejemplo 4

Assembler

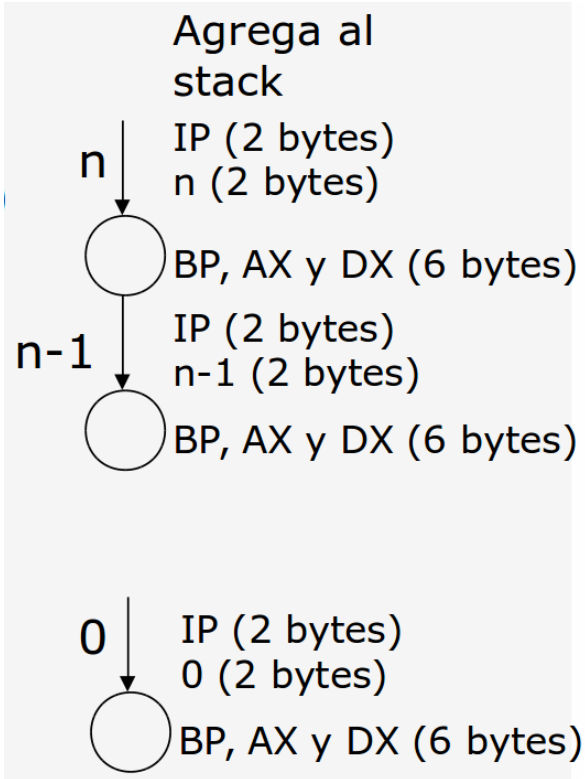
Llamada	Asembler

push n
call fact
pop resultado
	fact proc
	  push bp ; guardo el registro bp
	  mov bp,sp ; apunto con bp al tope de la pila
	  push ax ; conservo registros
	  push dx
	  cmp word ptr [bp+4],0 ; comparo n con cero
	  je esCero
	  mov ax,[bp+4] ; obtengo n
	  dec ax ; ajusto parámetro para la invocación
	  push ax ; coloco parámetro en el stack
	  call fact ; realizo la llamada recursiva
	  pop ax ; obtengo el resultado
	  mul word ptr [bp+4] ; calculo el paso recursivo
	  jmp fin
	esCero:
	  mov ax,1 ; asigno el resultado del paso base
	fin: ; acomodo el stack para el retorno
	  ;piso el parámetro de entrada con el resultado
	  mov [bp+4],ax
	  pop dx ; restauro registros
	  pop ax
	  pop bp
	  ret
	fact endp

Factorial - Ejemplo 4

Consumo

- Planteo de la recurrencia
  - $consumo(0) = 10$
  - $consumo(n) = 10 + consumo(n-1)$
- Resolviendo la recurrencia
  - $consumo(n) = 10 * (n+1)$



Repaso

1. Tipos de datos. Básicos, estructurados.
2. Funciones. Contexto.
3. Pasaje de parámetros. Ajuste de stack.
4. Recursión.

¿Preguntas?