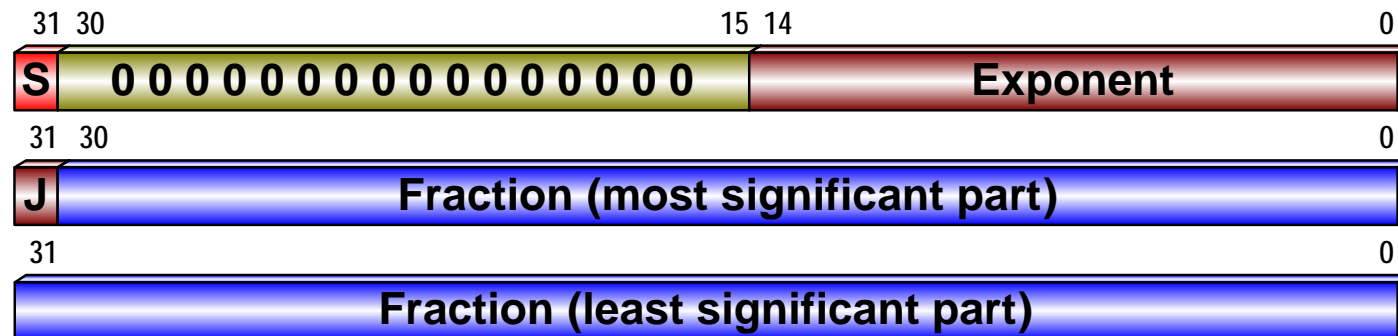




Sistemas electrónicos digitales










Tema 6:

Soporte para lenguajes de alto nivel





Índice del tema

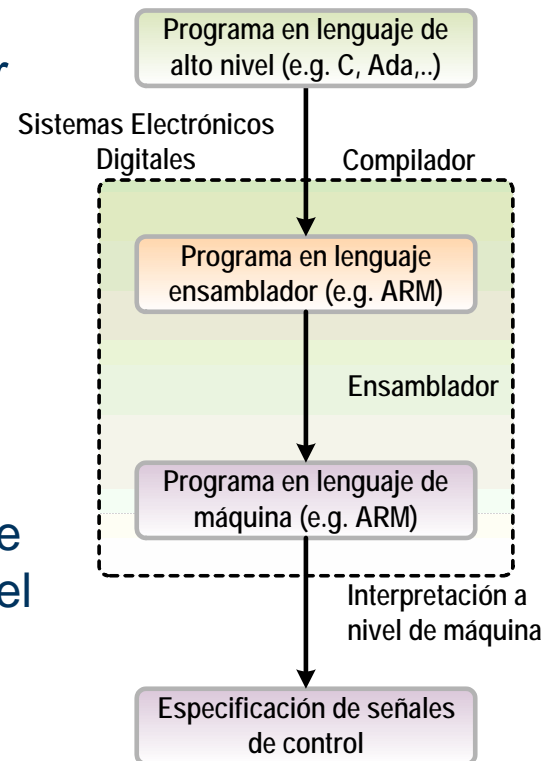
-  Introducción
-  Tipos de datos
-  Representación en punto flotante
-  Arquitectura en punto flotante en el ARM
-  Lenguajes de alto nivel, ANSI C
-  Funciones en C/Ensamblador
-  Uso de memoria
-  *Stack frame*
-  *Keil examples*



Introducción: Niveles de abstracción

■ Importancia de los niveles de abstracción en el desarrollo de software

- Posibilita la programación independiente de la arquitectura de la máquina
- Nivel ensamblador
 - Dependiente de la arquitectura del procesador
 - Abstracción → Programador
- Alto nivel (e.j. C)
 - Independiente de la arquitectura del procesador
 - Abstracción → Compilador



temp = v[k]
v[k] = v[k+1]
v[k+1] = temp

ldr r0, [r2,#0]
ldr r1, [r2,#4]
str r1, [r2,#0]
str r0, [r2,#4]

1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100

ALUOP[0:3] <= InstReg[9:11] & MASK



Introducción: Ventajas e inconvenientes

Parámetros de comparación

- Tiempo de desarrollo del software
 - A mayor nivel de abstracción más facilidad. El nivel de abstracción acerca la programación a estructuras más sencillas de utilizar por el programador
- Optimización del código
 - Los lenguajes de alto nivel no manejan muchas de las características implícitas de la arquitectura, tal y como los registros
 - La programación a bajo nivel produce códigos optimizados y, por lo tanto, más rápidos en ejecución
- *Software life cycle* y reusabilidad
 - La reusabilidad de un código a bajo nivel es más costosa siendo, en la mayor parte de las veces, más rentable la reescritura del código
- Portabilidad: Sólo en lenguajes de muy alto nivel



- Usar descripción a bajo nivel en rutinas críticas del sistema
- Ejemplo: DSP, implementación de un FIR

- 

- Necesidad de establecer mecanismo que permitan el paso de parámetros



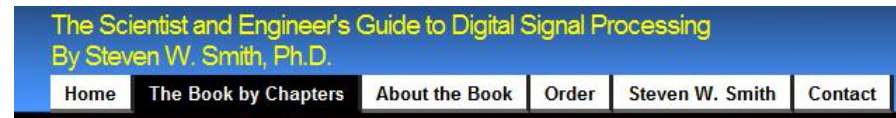


Introducción: *C* versus *Assembler*

Análisis de un ejemplo concreto

- DSP, realización de multiplicación de matrices para la implementación de un filtro FIR
- Ecuación:
 - $x[0]*y[0]+x[1]*y[1]+...$

<http://www.dspguide.com/ch28/5.htm>
Documento on-line; visitado el 18 de noviembre de 2007

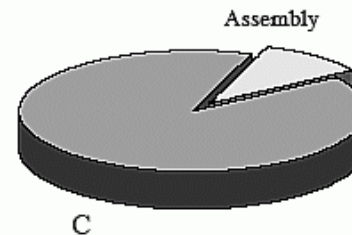


Chapter 28 - Digital Signal Processors / C versus Assembly

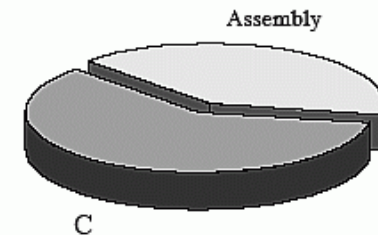
Chapter 28: Digital Signal Processors

C versus Assembly

a. Traditional Programmers



b. DSP Programmers



c. DSP Revenue

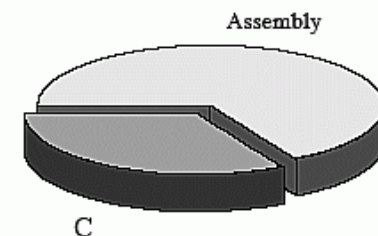


FIGURE 28-9
Programming in C versus assembly. As shown in (a), only about 10% of traditional programmers (such as those that work on personal computers and mainframes) use assembly. However, as illustrated in (b), assembly is much more common in Digital Signal Processors. This is because DSP programs must operate as fast as possible, and are usually quite short. Figure (c) shows that assembly is even more common in products that generate a high revenue.



Tipos de datos: Introducción I

 Lo importante no es cómo se almacena la información sino cómo se usa

- Un dato, una instrucción o una dirección se almacenan en una palabra de 32 bits sin “aparente” diferencia
- Niveles de abstracción en la representación de datos

 **Números:**

- Representación: MCMXCV; 1995; 0001 1001 1001 0101; 11111001011; 7CB
- Rango: Un procesador reserva un determinado número de bits para la representación de un dato
 - Unsigned 32 bits, 0 \rightarrow 4 294 967 295
 - Signed 32 bits, -2 147 483 648/+2 147 483 647
- Posibilidad de manejar números mayores. Uso de los flags de condiciones para su procesamiento



Tipos de datos: Introducción II

Números reales

- Necesidad de representar números reales (soporte a lenguajes de alto nivel)
- Tratamiento por emulación o *Floating point Coprocessor*

Caracteres

- Es uno de los tipos de datos más comunes, después de los números
- Estándar para la representación y comunicación de caracteres. ASCII
 - 1995 → ; 0x31 0x39 0x39 0x35
- ARM da soporte a este tipo de datos mediante las instrucciones de *load/store* con *unsigned byte*
- Orden de almacenamiento de bytes
 - El tratamiento de la información anterior en tamaño palabra resultará incorrecta debido al orden de almacenamiento de los bytes, *little endian*. Así, en el caso anterior la información leída en una palabra de 32 bits resultará ser “5991”



Tipos de datos: ANSI C

Los lenguajes de alto nivel definen sus propios tipos de datos, de forma independiente de la arquitectura

ANSI C, tipo de datos básicos

- *Signed and unsigned **characters***, al menos 8 bits
- *Signed and unsigned **short integers***, al menos 16 bits
- *Signed and unsigned **integers***, al menos 16 bits → 32 bits (ARM)
- *Signed and unsigned **long integers***, al menos 32 bits
- ***Floating point, double, and long double** floating point numbers*
- ***Enumerated types***
- ***Bitfields***

ANSI C, tipos derivados

- ***Arrays*** de datos del mismo tipo
- ***Functions*** que devuelven un tipo de dato
- ***Structure*** que contienen una secuencia de datos de diferente tipo
- ***Pointers*** son direcciones del sistema apuntando a un objeto
- ***Unions*** permiten a varios objetos de diferente tipo compartir el mismo espacio de memoria en tiempos diferentes



Tipos de datos: ARM, soporte para ANSI C

■ ¿Qué tipo de soporte da el ARM para la compilación de código en ANSI C?

■ Tamaño interno de datos da soporte directo a

- *Signed and unsigned integers*
- *Unsigned byte, unsigned character*
- *Long integers*
- *Pointers* corresponden a direcciones de memoria

■ Instrucciones de manipulación de bytes/*halfword*

- *Short integers*
- *Signed characters*

■ Modos de direccionamiento

- Ideal para el manejo de arrays y estructuras. Posibilidad de manejar arrays de elementos múltiplos de 2

LDR/STR{cond} {B} Rd, [Rn], offset

■ No se da soporte al procesamiento en punto flotante



Representación en punto flotante: IEEE754

■ Representación genérica $\rightarrow R = a \times b^n$

■ Estándar IEEE754

■ *Single precision*



- 1 bit de signo
- *Exponent* en exceso. El estándar define la representación del exponente en exceso 127. Con esto se consigue tener el mayor exponente como 1111 1110 y el menor como 0000 0001
 - Exponente_{,127} = E + 127
- Mantisa de 23 bits

■ Proceso de normalización, aumento de la precisión

- Ejemplo: $1995 \rightarrow 11111001011_2 \rightarrow 1.1111001011 \times 2^{10}$
Exponente_{,127} = 127 + 10 = 137





Representación en punto flotante: Consideraciones

Consideraciones

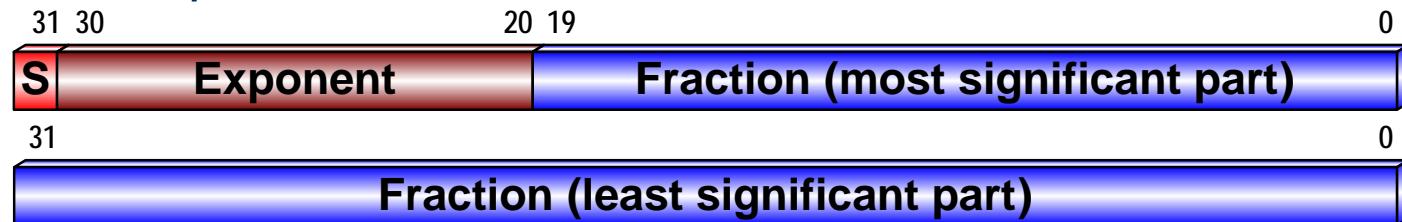
- Representación genérica
 - Fórmula: $\text{Valor}(\text{norm}) = (-1)^s \times 1.\text{fracción} \times 2^{(\text{exponente} - 127)}$
- El valor “0” tiene dos posibles representaciones
- +/- infinito se representan con el mayor exponente, mantisa a cero y el correspondiente bit de signo
- NaN (*Not a Number*). Corresponde a una representación con el mayor exponente y fracción diferente de cero
 - Utilizado para representar operaciones no válidas
- Números no normallizados
 - Se utilizan para representación de números demasiado pequeños
 - El exponente es cero y la fracción no está normalizada



Representación en punto flotante: *Double and Extended precision*

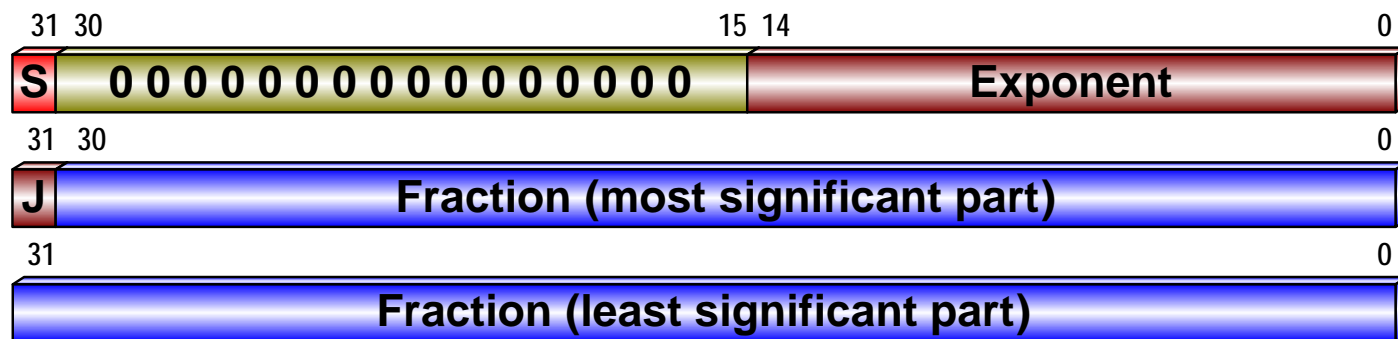
Estándar IEEE754

Double precision



- *Exponent en exceso de 1023*

Double Extended precision



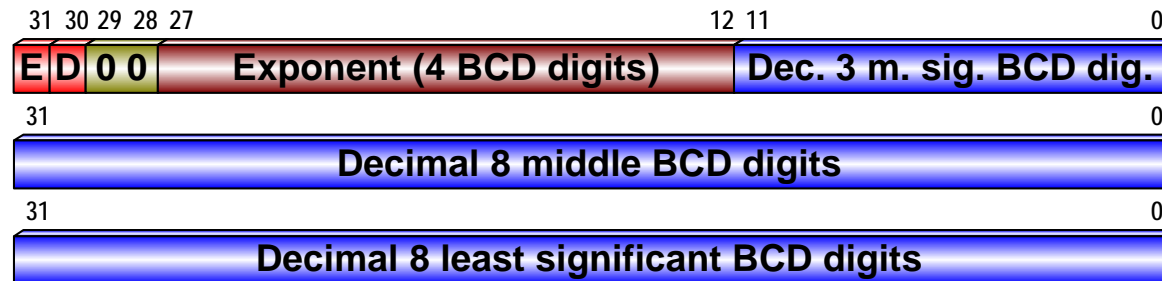
- *Exponent en exceso de 16383*
- Bit J. A "1" para números normalizados, en caso contrario a "0"



Representación en punto flotante: Representación en BCD

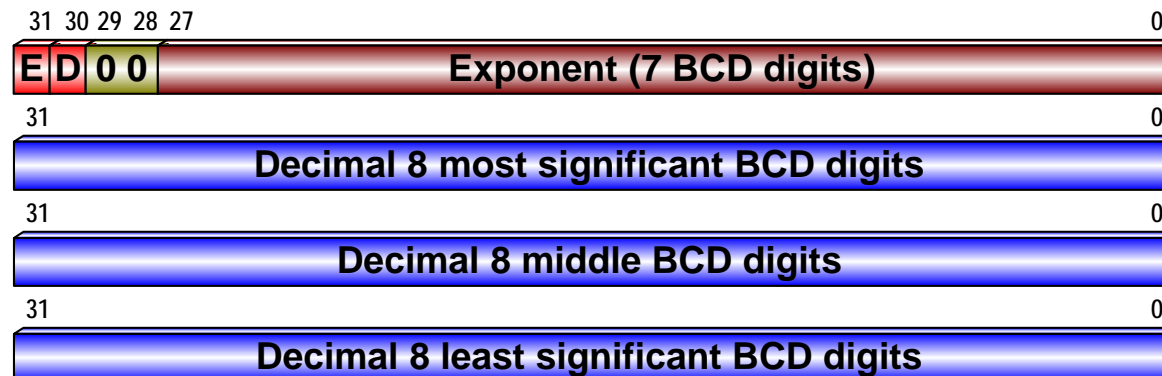
Estándar IEEE754

Packed decimal



- *E*: Signo del exponente; *D*: Signo de la parte decimal

Extended packed decimal



■ Fórmula: $\text{Valor}(\text{packed}) = (-1)^D \times \text{decimal} \times 10^{((-1)^E \times \text{exponente})}$



Representación en punto flotante: *ARM Floating point instructions*

- ARM no soporta **directamente** el procesamiento en punto flotante
- Sin embargo, el ARM ha definido un conjunto de instrucciones de procesamiento en punto flotante dentro del espacio de instrucciones de coprocesador
 - Estas instrucciones son manejadas en su gran mayoría por software a través de la excepción por instrucción no definida
 - Sin embargo, un subconjunto de estas operaciones son soportadas por coprocesadores *hardware*, tal es el caso del FPA10
- Para la programación en C, ARM ha desarrollado una librería, para el procesamiento en punto flotante, que soporta el estándar IEEE754 en simple y doble precisión



Arquitectura en punto flotante en el ARM: Introducción

ARM da soporte al procesamiento en punto flotante

- Solución hardware mediante FPA10
- Solución enteramente software mediante FPASC
- Solución mixta hardware/software FPE = FPA10/FPASC

La arquitectura hardware en punto flotante del ARM presenta

- Interpretación del juego de instrucciones cuando la unidad lógica del coprocesador es 1 o 2
- En estas unidades lógicas se dispone de 8 registros de 80 bits (los mismos para ambas unidades lógicas)
- Un registro de estado de punto flotante (FPSR), visible por el usuario, que, además, indica condiciones de error
- Opcionalmente, un registro de control de punto flotante (FPCR), no visible por el usuario, únicamente accesible al software de soporte para el coprocesador



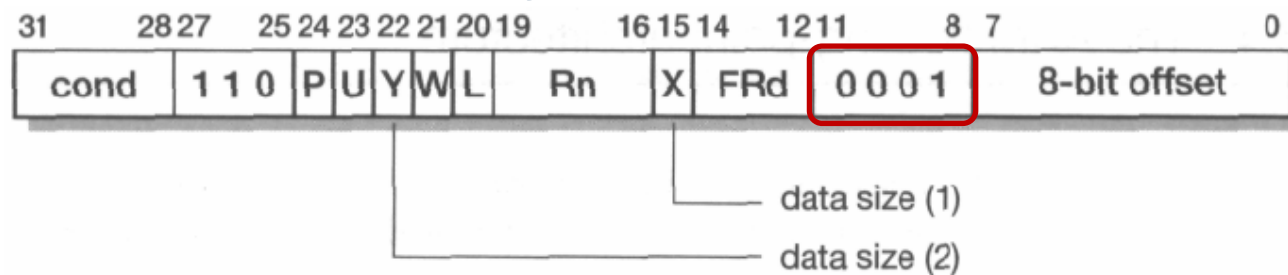
Arquitectura en punto flotante en el ARM: Instrucciones I

- En cuanto a tipos de datos, el FPA10 soporta
 - *Single, double y extended double precision*
 - El formato *packed decimal* se soporta mediante software

Registros internos

- Internamente todos los registros son *extended double precision*
- El ajuste de precisión se realiza mediante las operaciones de transferencia

Load/Store floating instructions

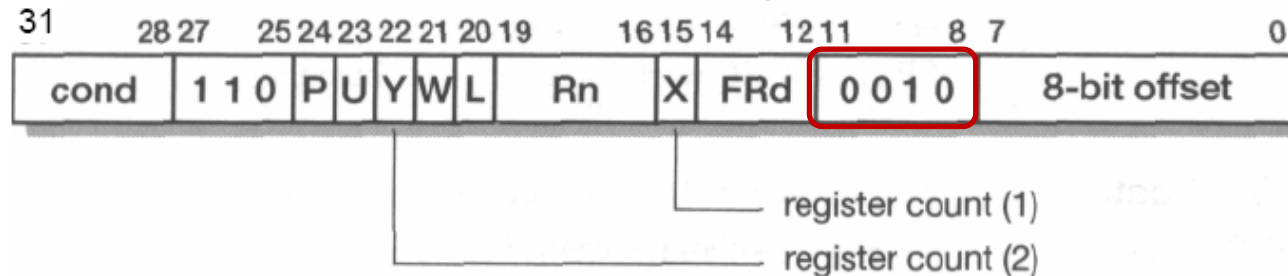


- X,Y → Codifican una de las cuatro posibles precisiones con las que trabaja el FPA10



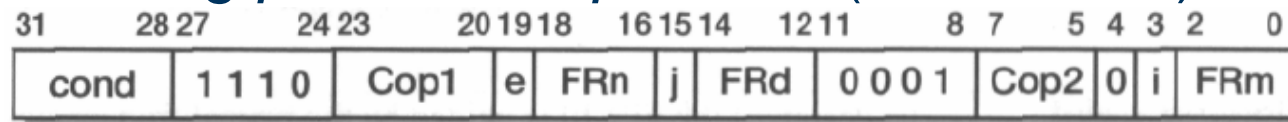
Arquitectura en punto flotante en el ARM: Instrucciones II

Load/Store multiple floating instructions



- Cada registro se almacena en tres posiciones de memoria
- *Frd* → Codifica el primer registro a transferir
- *X, Y* → Codifica el número de registros a transferir, máximo 4

Floating point data operation (handshake)

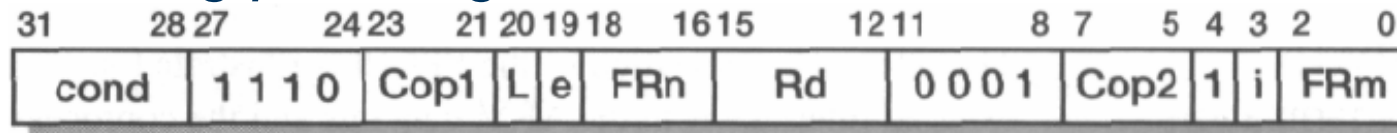


- *i* → Selecciona entre Frm o dato inmediato como 2º operando
- *e, Cop2* → Selecciona el tamaño del destino y redondeo
- *j* → Selecciona entre uno o dos operandos



Arquitectura en punto flotante en el ARM: Instrucciones III

Floating point register transfers



■ ARM → *Floating Point*

- Operaciones *Float*
- Escrituras del FPSR y FPCR

Floating Point → ARM

- Operaciones *Fix*
- Lecturas de FPSR y FPCR

■ Comparación en punto flotante

- En este caso Rd es r15 y se comparan dos registros en punto flotante
- El resultado de la comparación se devuelve en los flags de CPSR



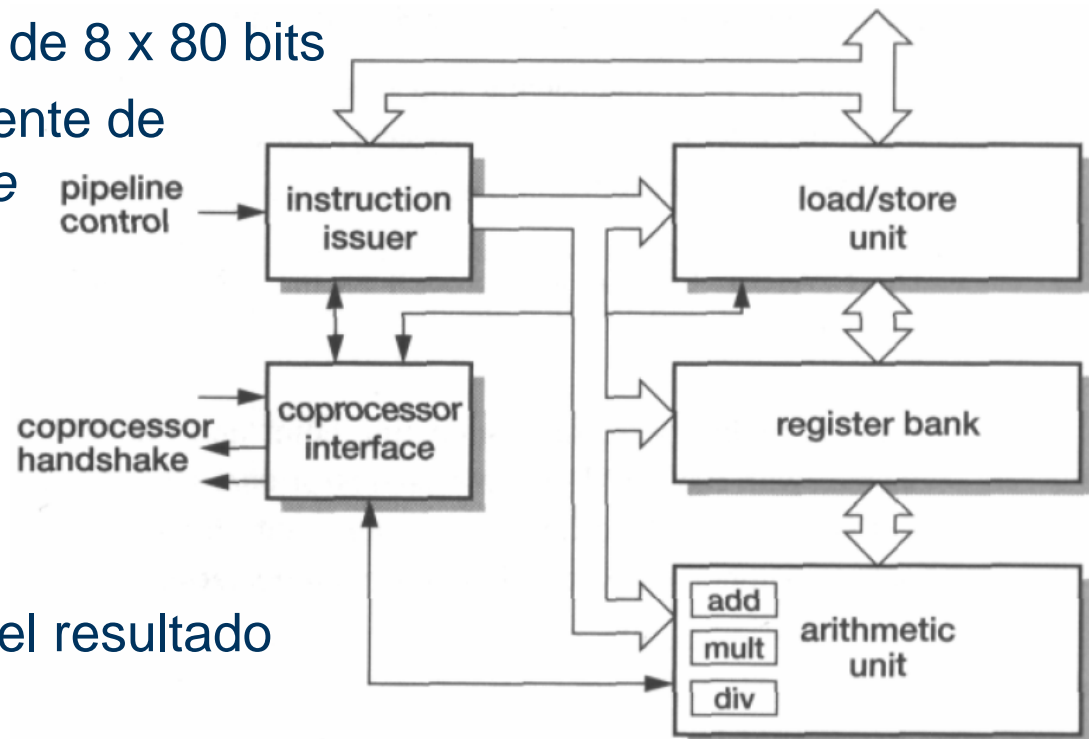
Arquitectura en punto flotante en el ARM: Arquitectura del FPA10

Organización interna

- La unidad *load/store* realiza la conversión de tipos, fijos y punto flotante
- Banco de registros de 8 x 80 bits
- Operación concurrente de la unidad *load/store* y la unidad arit.

Pipeline

- 1ª Alineamiento de los operandos
- 2ª Suma/Mul/Div
- 3ª Normalización del resultado
- 4ª Redondeo





Lenguajes de alto nivel, ANSI C: Sintaxis en C I

Ejemplo de programa: main.c


```
07  /* Include files */
08  #include <stdio.h>
09
10  /* Main program */
11  int main (void) {
12      /* Declaracion de variables */
13      unsigned int exp = 1;
14      int k;
15
16      /* Compute 2 to the 31st.*/
17      for (k=0; k<31; k++) {
18          exp = exp * 2;
19      }
20      return 0;
21  } /* End main() */
```

Sin argumentos

int main (void)

Con argumentos de llamada

int main (int argc, char *argv[])

 Reemplaza la línea por el contenido especificado en el fichero

- <file> busca *file* en el área del sistema

- “file” busca *file* en el área de usuario

 *main* is el programa llamador por el sistema (ver *Startup.s*)

 Declaración de las variables antes de utilizarse

 /* */ Para introducir comentarios

 Toda secuencia de declaraciones se introduce entre llaves

 Retorno al sistema


- Código de retorno




Lenguajes de alto nivel, ANSI C: Sintaxis en C II

Estructuras y condicional

```
07  /* Include files */
08  #include <stdio.h>
09
10  /* Main program */
11  int main (void) {
12      /* Declaracion de variables */
13      struct miTipo {
14          unsigned int exp;
15          char b;
16      };
17      struct miTipo alpha;
18      int k;
19
20      /* Compute 2 to the 31st.*/
21      alpha.exp = 1;
22      for (k=0; k<31; k++) {
23          alpha.exp = alpha.exp * 2;
24      }
25      if (alpha.exp)
26          return 0;
27      else
28          return 1;
29  } /* End main() */
```

 A diferencia de C++, en lugar de clases se definen estructuras de datos mediante struct

 Las estructuras dan la posibilidad de definir tipos propios de datos mediante la agrupación de datos relacionados, pudiendo manipular éstos como un todo

 Posibilidad de acceso a una variable de la estructura

 ¿Verdadero o Falso?

■ Condición de falso en C

- 0 (integer)
- NULL (pointer)

■ Condición de verdadero en C

- Cualquier otra



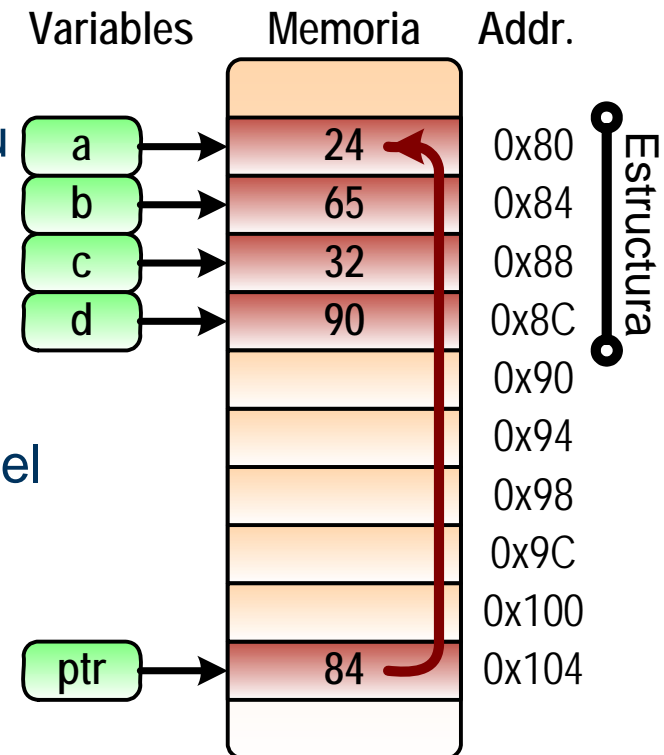
Lenguajes de alto nivel, ANSI C: Modelo de memoria

Organización de la memoria

- La memoria se considera como un simple array
- Cada celda está referenciada por su dirección
- Cada dirección contiene un valor
 - Dato
 - Dirección → Puntero
- En C la memoria es gestionada por el programador

Puntero

- Manera de representar en C una dirección de memoria
- Uso principal
 - Paso de parámetros por referencia
 - Parámetros de salida





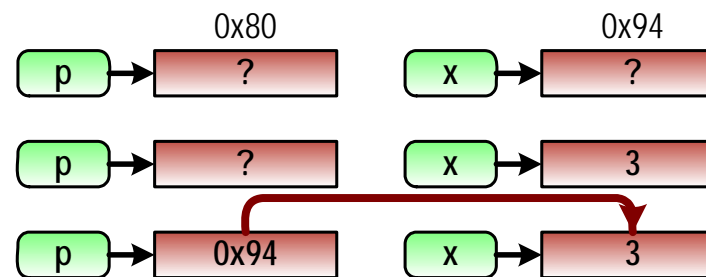
Lenguajes de alto nivel, ANSI C: Punteros en C I

¿Cómo crear un puntero en C?

- Operador &: da la dirección de una variable
- Operador *: definición de un puntero; da el valor apuntado por la variable a la que hace referencia

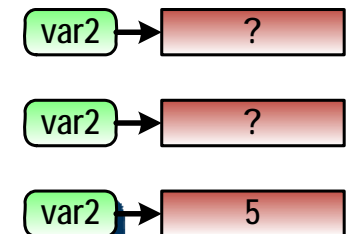
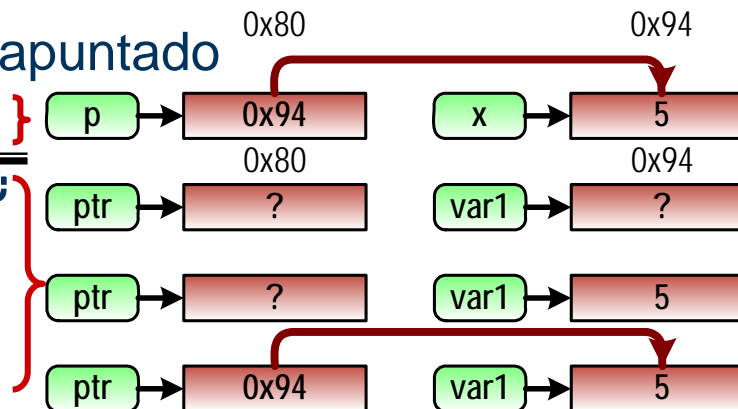
Ejemplo

```
int *p;  
int x;  
x = 3;  
p = &x;  
printf("p points to %d\n", *p);
```



■ Cambio del valor apuntado

```
*p = 5;  
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```





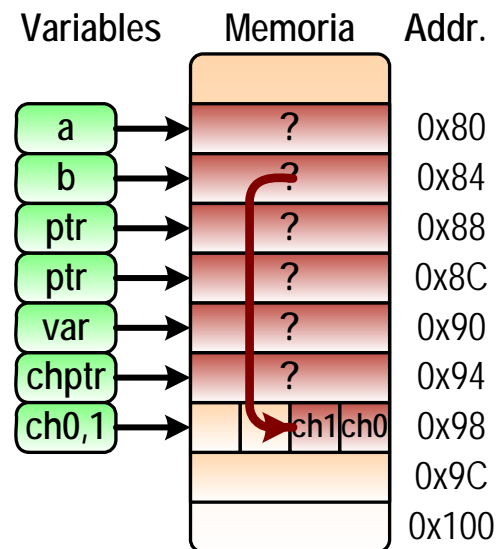
Lenguajes de alto nivel, ANSI C: Punteros en C II


Punteros, enteros, etc.


- Tamaño de entero → 1 palabra
- Tamaño de carácter → 1 byte
- Tipos de punteros, integridad de tipos
 - Un puntero sólo puede apuntar a un elemento de su tipo
 - El uso de “*” en la declaración no se extiende a todas las variables dentro de la declaración

Ejemplo

```
int    *a;
char   *b;
int    *ptr, var;
char   *chptr, ch0, ch1;
      b = &ch1;
```



 Nota: La declaración de un puntero no implica la asignación de memoria para la variable a apuntar

 Ejemplo

```
void f() {
    int    *ptr;
          *ptr = 5;
}
```



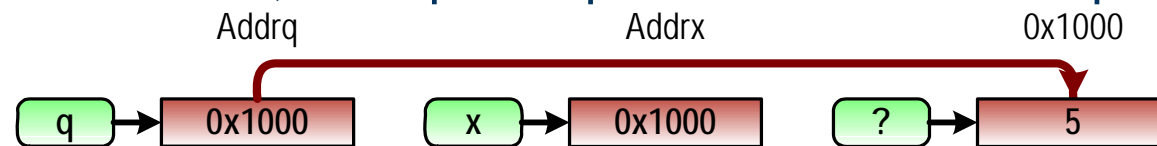

Lenguajes de alto nivel, ANSI C: Punteros en C III

■ A diferencia de otros lenguajes, C nos permite ajustar (*cast*) el tipo de una variable a cualquier otro tipo

Ejemplo

```
int    x = 0x1000;
int    *p = x;           /* Asignación no válida, no coinciden los tipos */
int    *q = (int *) x;   /* Asignación válida, ajuste (cast) de tipo */
```

■ Sin embargo, y en relación con el ejemplo anterior, el resultado no es válido, si lo que se pretende es tener un puntero a “x”



Punteros a estructuras. Operador “->”

```
struct point {
    int    x;
    int    y;
}
...
struct point *p
```



Formas equivalente de acceder a un elemento

```
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```




Lenguajes de alto nivel, ANSI C: Uso de punteros en paso de parámetros

Paso de parámetros a funciones

■ Por valor

- Se pasa una copia del dato original. El dato no resulta modificado

■ Por referencia (punteros)

- Se pasa un puntero al dato. El dato puede resultar modificado →

Peligro

Ejemplo

■ Por valor

```
int x;  
...  
    addOne(x);
```

■ Por referencia

```
int *p;  
...  
    addOne(p);
```

Ejemplo

```
/* paso por valor */  
void addOne( int x) {  
    x = x + 1;  
}  
  
int y;  
y = 3;  
addOne(y);  
/* y aún vale 3 */
```

```
/* paso por referencia */  
void addOne( int *p) {  
    *p = *p + 1;  
}  
  
int y;  
y = 3;  
addOne(&y);  
/* ahora y vale 4 */
```



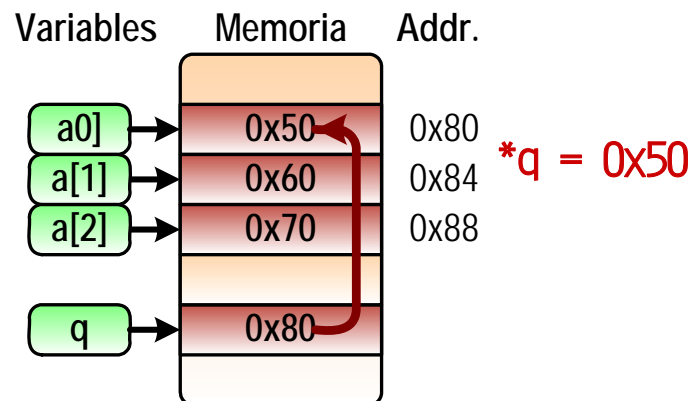
Lenguajes de alto nivel, ANSI C: Puntero a puntero I

¿Cómo poder modificar el valor de un puntero mediante un procedimiento?

Ejemplo

```
void incrementPtr( int *p) {  
    p = p + 1;  
}
```

```
...  
int  a[3] = {0x50, 0x60, 0x70};  
int  *q = a;  
    incrementPtr(q);  
    printf("**q = %d\n", *q);
```

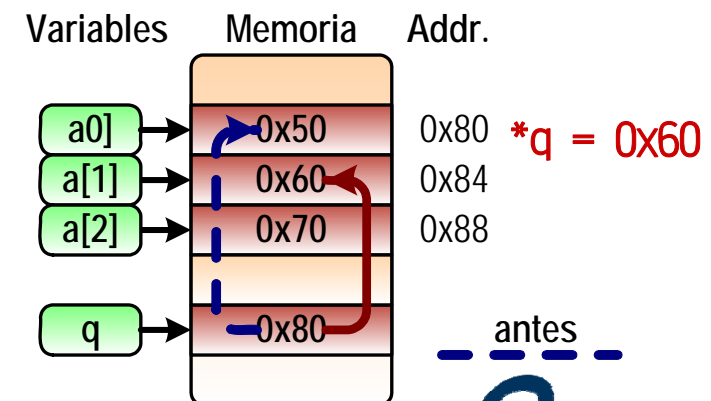


Explicación

Si se quiere manejar el puntero, habrá que pasar un puntero a puntero

```
void incrementPtr( int **h) {  
    *h = *h + 1;  
}
```

```
...  
int  a[3] = {0x50, 0x60, 0x70};  
int  *q = a;  
    incrementPtr(&q);  
    printf("**q = %d\n", *q);
```



antes



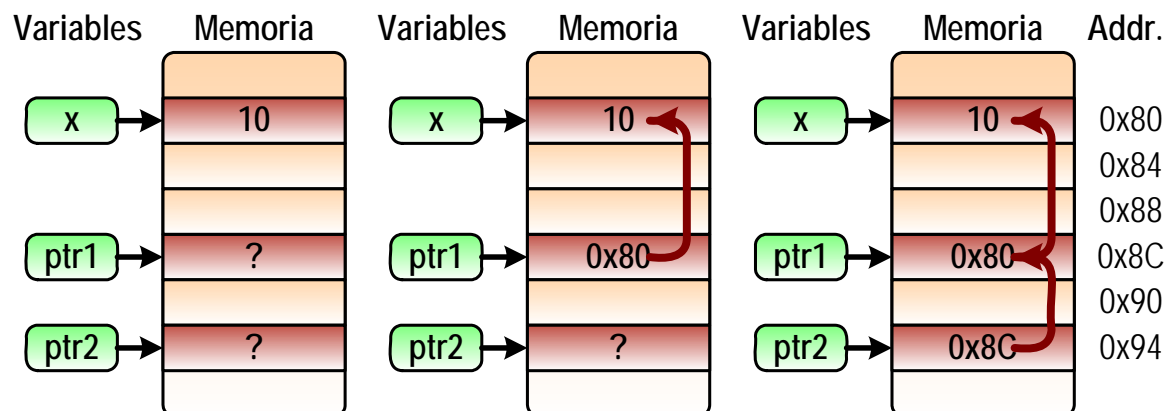
Lenguajes de alto nivel, ANSI C: Puntero a puntero II

Existe la posibilidad de definir un puntero a puntero, lo que posibilita la modificación del mismo

Ejemplo

```
11  /* Main program */
12  int main (void) {
13      int x = 10;
14      int *ptr1;
15      int **ptr2;
16      /* Main body*/
17      ptr1 = &x;
18      ptr2 = &ptr1;
19  } /* End main() */
```

Name	Value
x	0x0000000A
ptr1	0x4000045C
*ptr1	0x0000000A
ptr2	0x40000458
*ptr2	0x4000045C
**ptr2	0x0000000A



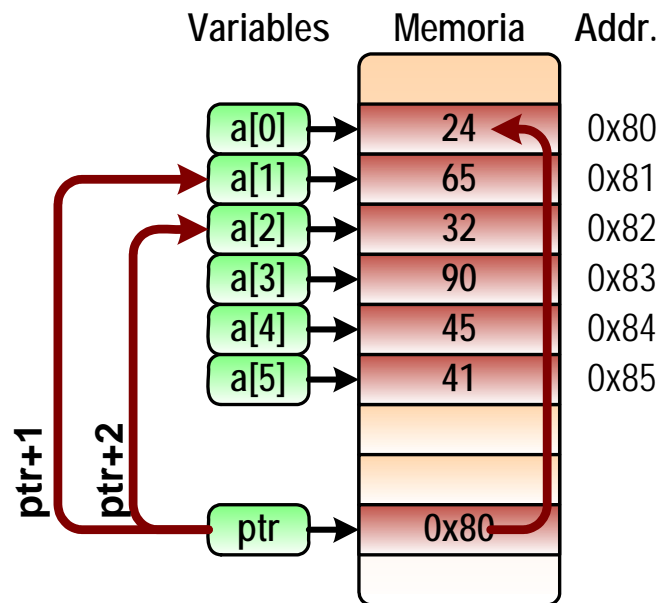


Lenguajes de alto nivel, ANSI C: Arrays y punteros I

Al ser un puntero una dirección de memoria, se puede operar con ésta

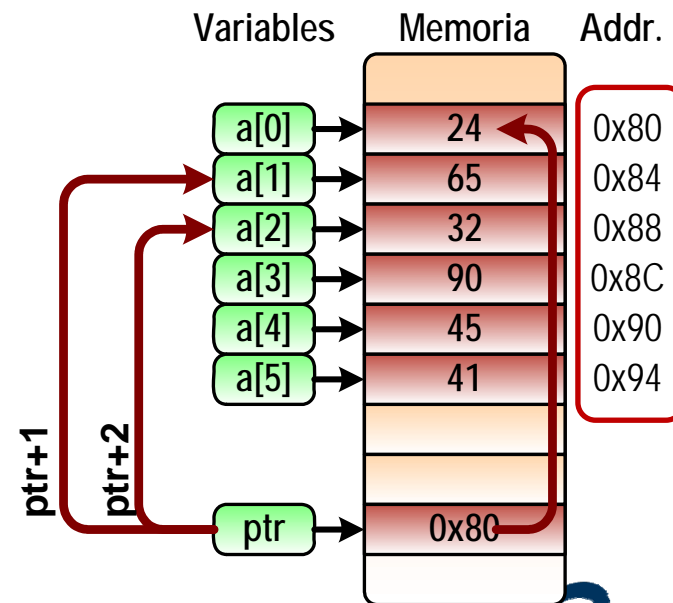
Ejemplo 1

```
char a[5];  
char *ptr;  
/* asignación de punteros */  
ptr = a;
```



Ejemplo 2

```
int a[8];  
int *ptr;  
/* asignación de punteros */  
ptr = a;
```



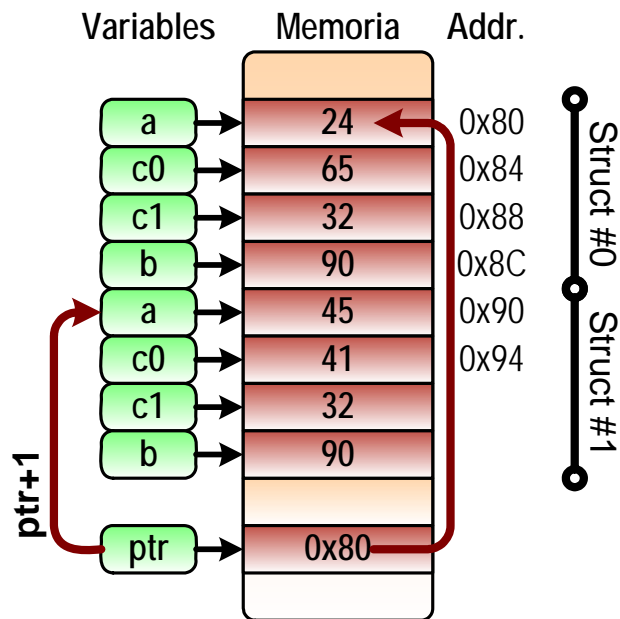


Lenguajes de alto nivel, ANSI C: Arrays y punteros II

El compilador toma nota del tamaño del elemento apuntado para realizar el cálculo de $\text{ptr} + 1$

■ Ejemplo con estructura

```
struct miTipo {  
    int  a;  
    char c0,c1;  
    int  b; }  
Variables
```



El siguiente código es equivalente al cálculo realizado, devolviendo el enésimo elemento de un array

```
int get(int array[], int n) {  
    return (array[n]);  
    /* 0 bien */  
    return *(array + n);  
}
```



Lenguajes de alto nivel, ANSI C: Arrays y punteros III

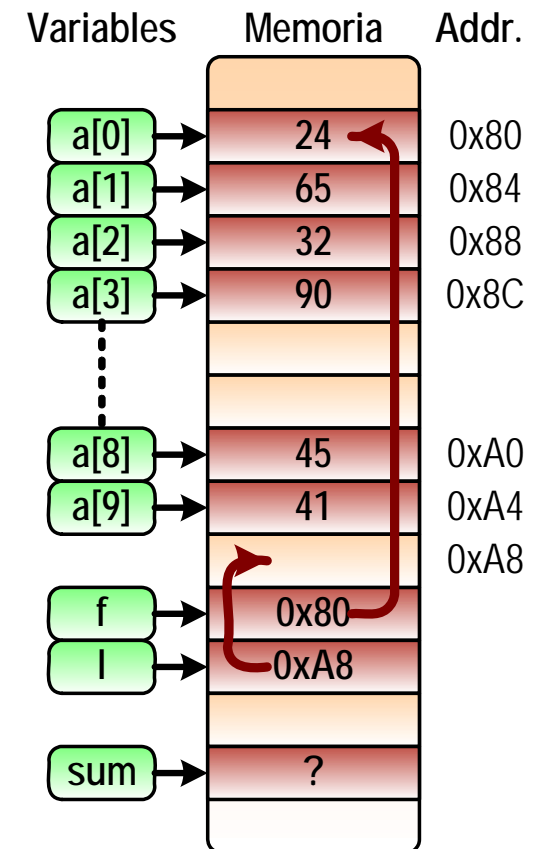
Uso de punteros para control de estructuras

Ejemplo: suma de elementos de un array

```
int    ar[10], *f, *l, sum = 0;

...
f = &ar[0];
l = &ar[10];
while (f != l) {
    /* sum = sum + *p; p = p + 1 */
    sum += *p++;
}
```

- C no controla los límites de los arrays definidos y su referencia, como dirección, es válida
- Principales consecuencias
 - *Segmentation faults*
 - *Bus error*





Lenguajes de alto nivel, ANSI C: Operaciones con punteros I

- Es posible operar aritméticamente sobre un puntero
- Únicamente se permiten aquellas operaciones que tienen sentido sobre punteros

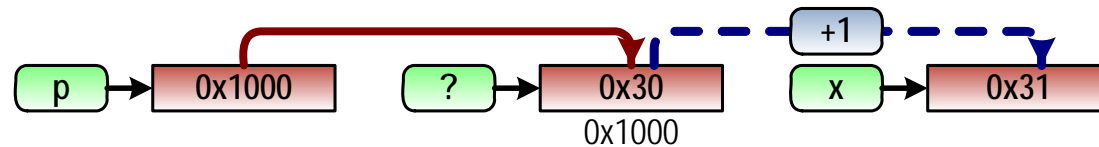
Operación	¿Válido?
Puntero + Entero	OK
Entero + Puntero	Not OK
Puntero + Puntero	Not OK
Puntero – Entero	OK
Entero – Puntero	Not OK
Puntero – Puntero	OK
Comparar puntero con puntero	OK
Comparar puntero con entero	Not OK
Comparar puntero con 0	OK



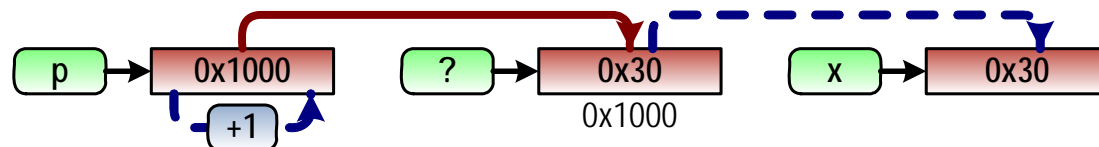
Lenguajes de alto nivel, ANSI C: Operaciones con punteros II

Operaciones: $(*p)+1$; $*p++$; $*(p+1)$; $(*p)++$

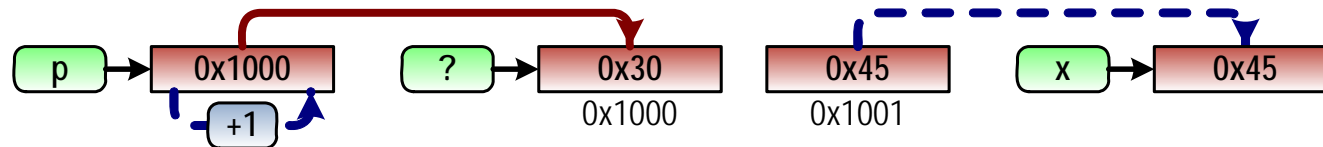
■ $(*p)+1$: $x = (*p)+1 \quad \rightarrow x = *p + 1; \quad p = p$



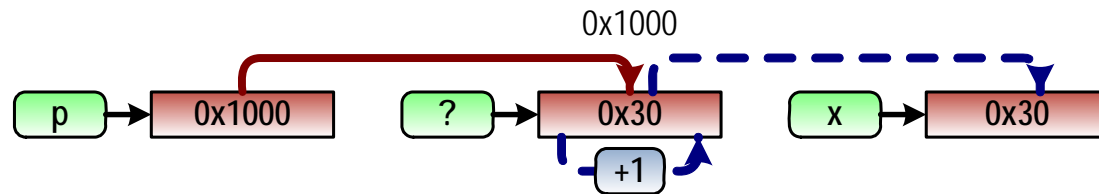
■ $*p++$: $x = *p++ \quad \rightarrow x = *p; \quad p = p + 1$



■ $*(p+1)$: $x = *(p+1) \quad \rightarrow x = *(p + 1); \quad p = p$



■ $(*p)++$: $x = (*p)++ \quad \rightarrow x = *p; \quad *p = *p + 1$





Lenguajes de alto nivel, ANSI C: *Strings*

■ Una *string* en C no es más que un array de caracteres

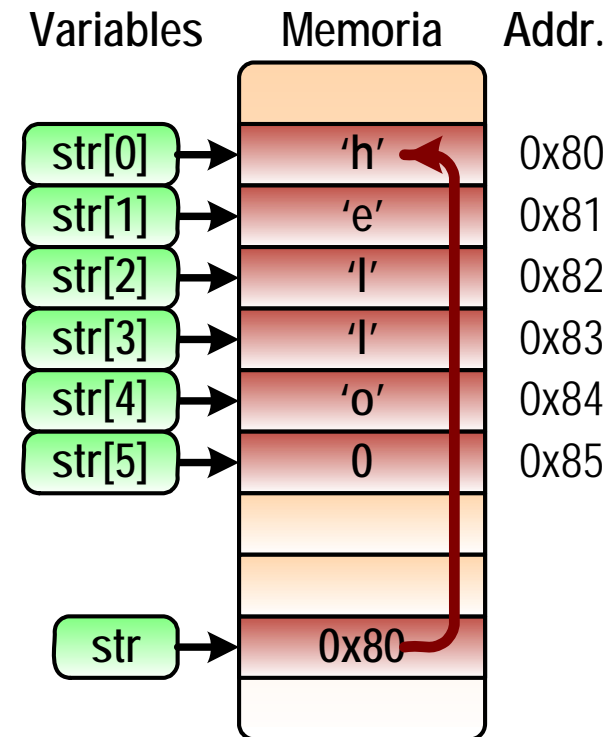
```
char *str = "hello"
```

- str apunta a la "h", le siguen los caracteres 'e', 'l', 'l', 'o' y un carácter *NULL* (\0)
- Una *string* en C siempre termina con un carácter *NULL*, por lo que a veces se denominan *null-terminated strings*

■ Ejemplo: strlen

```
int strlen (char s[]) {  
    /* Array vs Pointer → Mismo resultado */  
    int strlen (char *s) {  
        int n = 0;  
        while (s[n] != 0) n++;  
        return n;  
    }  
}
```

while (s[n])





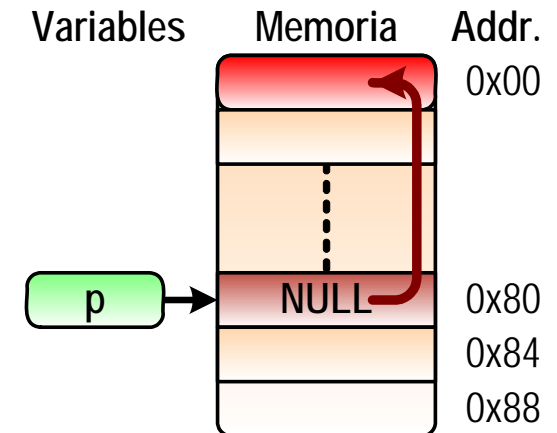
Lenguajes de alto nivel, ANSI C: Errores comunes con punteros

Asignación de un valor sin inicializar el puntero

```
int *p;  
*p = 10;
```

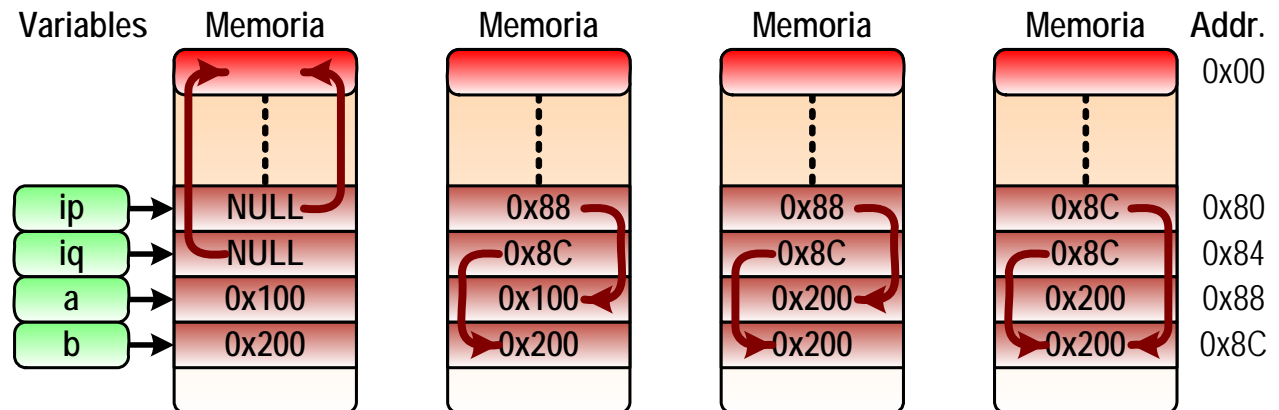
Error:

- C define un puntero con valor inicial a NULL (0)
- La dirección 0 no es una dirección válida para escribir



Copiar punteros frente a valores

```
int *ip, *iq, a = 0x100, b = 0x200;  
ip = &a; iq = &b;  
*ip = *iq;  
ip = iq;
```

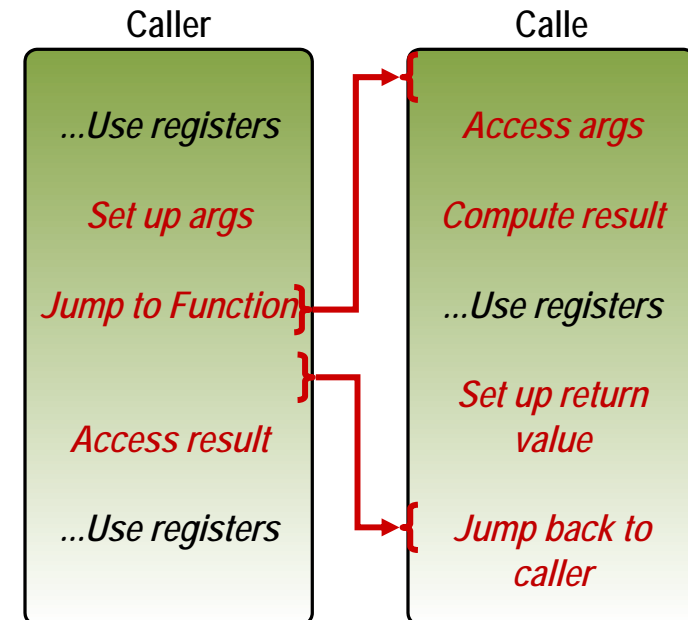




Funciones en C/Ensamblador: Fundamentos básicos

¿A qué información debe seguir la traza un compilador?

```
main (void) {  
    int    i, j, k, m;  
           i = mult(j, k);  
           m = mult(i, i);  
}  
int mult (int mcand, int mlier) {  
    int    product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier - 1;  
    }  
    return product;  
}
```





Funciones en C/Ensamblador: APCS Convenios

Convenios en ARM *Procedure Call Standard (APCS)*

- Se define el uso particular de los registros de propósito general
- Se define el tipo de pila, de entre *full/empty* y *ascending/descending*, a utilizar en los pasos de parámetros
- Se define el tipo de estructura de datos implementados en la pila, con propósito de *back-tracing*
 - Este aspecto es muy útil cuando se trata de poder acceder, desde una función/rutina a
- Uso de código re-entrante, independiente de la posición de almacenamiento en memoria, lo que permite a diferentes procesos/aplicaciones compartir código

Existen varias versiones del APCS

- *Stack limit check*
- *Floating points arguments*
- *Re-entrant vs non re-entrant code*



Funciones en C/Ensamblador: APCS Registers

ARM Procedure Call Standard (APCS) registers

Reg. Nº	APCS Nomb.	Función	
0	a1	argument 1 / integer result / scratch register	Caller Saved: no es necesario preservar sus valores; "a1" → devolución de entero
1	a2	argument 2 / scratch register	
2	a3	argument 3 / scratch register	
3	a4	argument 3 / scratch register	
4	v1	register variable	Callee Saved: No pueden ser modificados, de ser necesario hay que salvarlos en la pila para recuperarlos a la salida
5	v2	register variable	
6	v3	register variable	
7	v4	register variable	
8	v5	register variable	
9	sb/v6	static base / register variable	Registros de propósito específico: es posible su doble uso en caso de salvarlos previamente, e.g. registro lr
10	sl/v7	stack limit / stack chunk handle / register variable	
11	fp	frame pointer	
12	ip	scratch register / new sb in inter-link-unit calls	
13	sp	lower end of current stack frame	
14	lr	link address / scratch register	
15	pc	program counter	
f0	0	FP argument 1 / FP result / FP scratch register	Floating point: parámetros en punto flotante
f1	1	FP argument 2 / scratch register	
f2	2	FP argument 3 / scratch register	
f3	3	FP argument 4 / scratch register	



Funciones en C/Ensamblador: C2Asm, ejemplo *cpyTable*, uso de argumentos

 *cpyTable*: copia una tabla origen a destino de forma inversa

```
07 { void __cpyTable(char *isrc, char
08         *idest, int icnt);
09
10 int main(void) {
11     char src[10] = {0,1,2,3,4,5,6,7,8,9};
12     char dest[10];
13     int cnt = 10;
14
15     /* Llamada a la rutina */
16     __cpyTable(src+cnt-1, dest, cnt);
17 } /* end main() */
```

- Definición del interfaz de la rutina, tipos de parámetros
- Keil: uso de argumentos a1, a2 y a3
 - Posibilidad de usar los símbolos definidos en el APCS

```
07 AREA RUTINAS, CODE, READONLY
08 EXPORT __cpyTable
09
10 __cpyTable
11 LOOP0 LDRB r3, [r0], #-1
12 STRB r3, [r1], #1
13 SUBS r2, r2, #1
14 BNE LOOP0
15
16 SALIR MOV pc, lr
17 END
```

Name	Value
src	0x4000044C [...]
dest	0x40000440 [...]
cnt	0x0000000A


```
13 LOOP0 LDRB r3, [a1], #-1
14 STRB r3, [a2], #1
15 SUBS r2, r2, #1
16 BNE LOOP0
```



Funciones en C/Ensamblador: C2Asm, ejemplo *cpyTable*, uso de argumentos y variables

 *sumArray*: suma los elementos de un array y deposita la suma en el último elemento del array

- La definición de variables locales se puede referenciar mediante los símbolos v1-v7

<pre>07 /*void __sumArray(int arr[]);*/ 08 void __sumArray(int arr[]){ 09 int i, sum = 0; 10 for(i=0; i<9; i=i+1) 11 sum = sum + arr[i]; 12 arr[i] = sum; 13 } 14 int main(void) { 15 int array[10] = {1,2,3,4,5,6,7,8,9,0}; 16 17 /* Llamada a la rutina */ 18 __sumArray(array); 19 } /* end main() */</pre>		<pre>07 AREA RUTINAS, CODE, READONLY 08 EXPORT __sumArray 09 10 __sumArray 11 MOV v1,#0 12 ADD a2,a1,#40-4 13 LOOP0 CMP a1,a2 14 BGE SALIR 15 LDR a3, [a1], #4 16 ADD v1,v1,a3 17 B LOOP0 18 SALIR STR v1,[a1] 19 MOV pc,lr 20 END</pre>
---	--	---


- ¿Qué ocurre si el número de variables supera v7?
 - Uso de la pila como soporte de variables (próximamente)



Funciones en C/Ensamblador: Asm2C, ejemplo *cpyTable*, uso de argumentos

 *cpyTable*: copia una tabla origen a destino de forma inversa

<pre>07 AREA MAIN, CODE, READONLY 08 EXPORT __main 09 IMPORT cpyTable 10 11 ENTRY 12 __main ADR r0, ENTEND-1 13 ; SALIDA localizada en RAM 14 LDR r1, =SALIDA 15 ; Copiamos la Entrada en la Salida 16 ; de forma inversa 17 MOV r2, #10 18 ;LOOP0 LDRB r2, [r0], #-1 19 ; STRB r2, [r1], #1 20 ; SUBS r3, r3, #1 21 ; BNE LOOP0 22 ; 23 BL cpyTable 24 SALIR B SALIR 25 26 ENTRADA DCB 0,1,2,3,4,5,6,7,8,9 27 ENTEND 28 29 AREA DATOS, DATA, READWRITE 30 SALIDA SPACE 10 31 SALEND</pre>	<pre>07 void cpyTable(08 { char *src, 09 { char *des, 10 { int cnt) 11 { 12 while (cnt > 0) { 13 *des++ = *src--; 14 cnt--; 15 } 16 } /* end cpyTable() */</pre>
--	--



Name	Value
src	0x00000149
des	0x40000000
cnt	0x0000000A

Watches

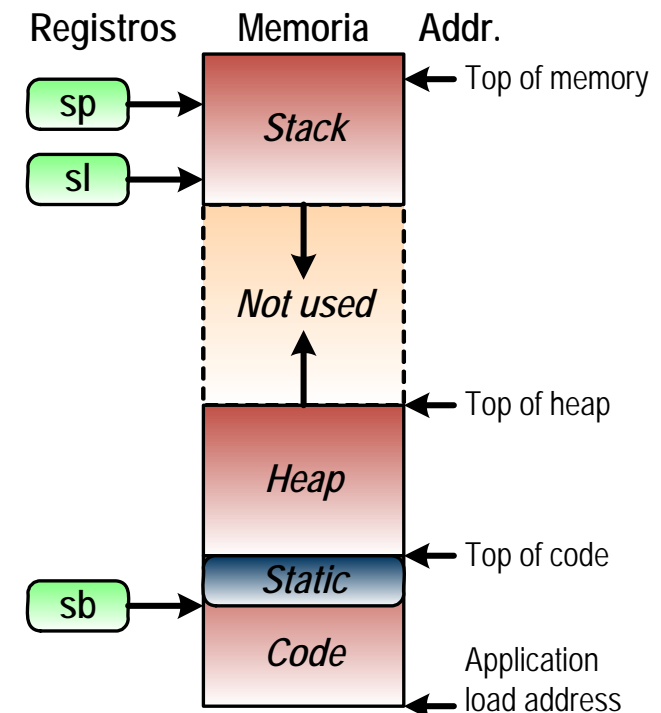
Locals Watch #1 Watch #2



Uso de memoria: modelo de espacio de direcciones

■ Áreas de memoria

- *Code*: área fija (estática) que contiene la imagen del programa/aplicación
- *Stack*: área de dato dinámica utilizada en las llamadas a funciones, crece hacia direcciones decrecientes
 - Contiene los denominados **marcos** (*frames*) de una función
 - Estos marcos se destruyen cuando se retorna de la función
- *Heap*: área de dato utilizada para la creación de nuevas variables bajo demanda de la aplicación (función *malloc*), crece hacia posiciones crecientes

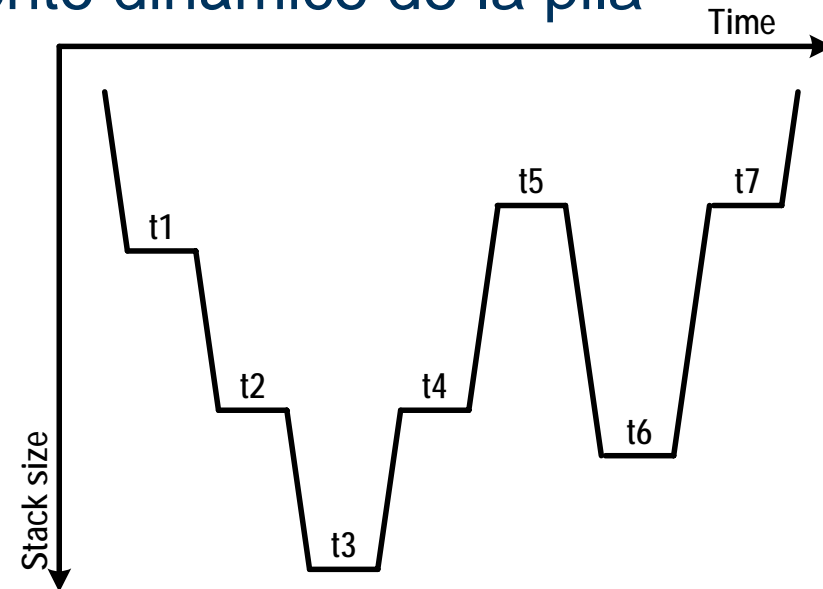




Uso de memoria: comportamiento de la pila

Ejemplo de comportamiento dinámico de la pila

```
main (void) {  
    ... /* t1 */  
    func1 ();  
    ... /* t5 */  
    func2 ();  
    ... /* t7 */  
} /* End of main */  
func1 () {  
    ... /* t2 */  
    func2 ();  
    ... /* t4 */  
} /* End of func1 */  
func2 () {  
    ... /* t3, t6 */  
} /* End of func2 */
```



 Localización de marcos para una misma función en diferentes áreas → destrucción de variables locales



Uso de memoria: variables globales

- Las variables globales son accesibles por todos los procesos/funciones
- El sistema declara las variables globales en un área de memoria
 - Esta área es referenciada mediante un puntero en memoria al final del código

```
07  /* Global variables */
08  int array[10] = {1,2,3,4,5,6,7,8,9,0};
09
10  /*void __sumArray(); */
11  void sumArray(int cnt){
12      int i, sum = 0;
13      for(i=0; i<cnt; i=i+1)
14          sum = sum + array[i];
15      array[i] = sum;
16  }
17  int main(void) {
18      /* Llamada a la rutina */
19      sumArray(9);
20  } /* end main() */
```

```
14:      sum = sum + array[i];
0x000001E4  E59F3034  LDR      R3,[PC,#0x0034]
0x000001E8  E7933101  LDR      R3,[R3,R1,LSL #2]
0x000001EC  E0822003  ADD      R2,R2,R3
0x000001F0  E2811001  ADD      R1,R1,#0x00000001
0x000001F4  E1510000  CMP      R1,R0
0x000001F8  BAF FFF9  BLT      0x000001E4
15:      array[i] = sum;
0x000001FC  E59F301C  LDR      R3,[PC,#0x001C]
0x00000200  E7832101  STR      R2,[R3,R1,LSL #2]
```

Current	
R0	0x00000009
R1	0x00000005
R2	0x0000000f
R3	0x40000000
R4	0x40000038

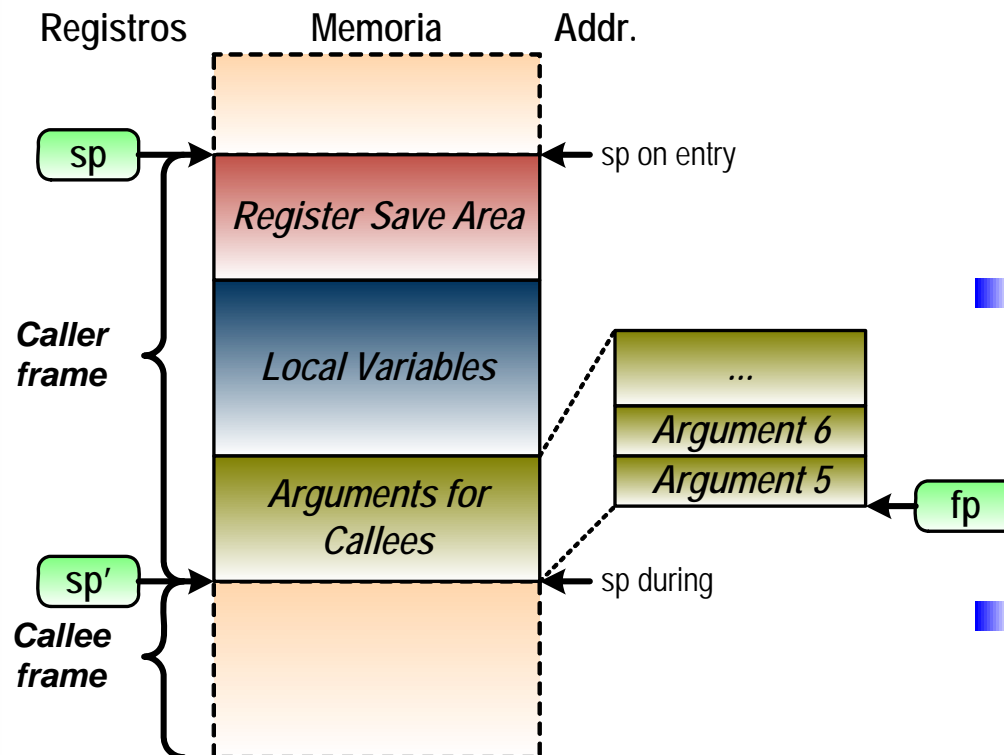


Mecanismo de llamada

- Los primeros cuatro argumentos se pasan por registros (r0-r3)

- Los primeros cuatro argumentos se pasan por registros (r0-r3)


- *sp*, *fp*, *sl/v7*, *sb/v6* y *v1-v5* deben preservar sus valores al retornar



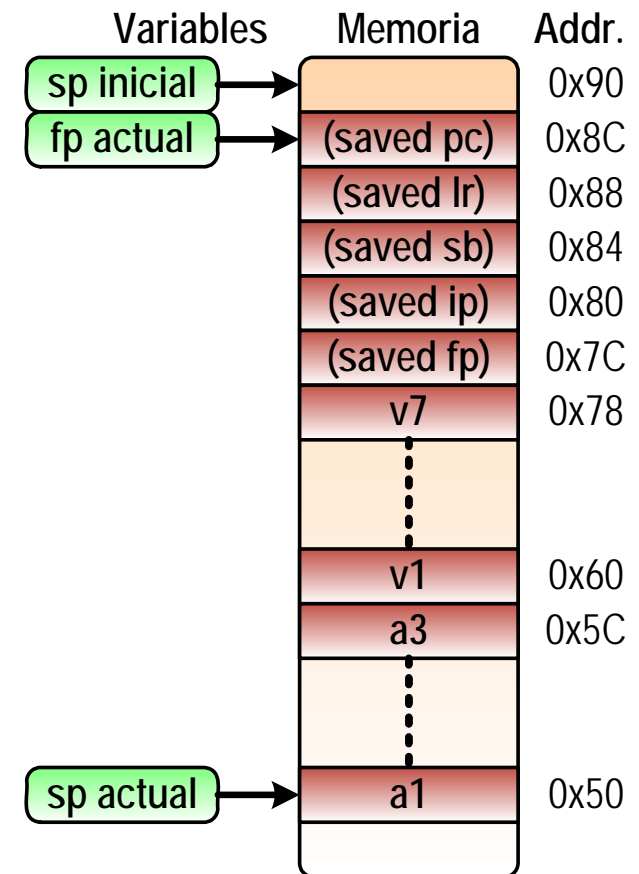


Stack frame: Registro *Frame pointer*

 *fp* apunta a la posición más alta de la pila de la función actual

 Mediante el uso del *frame pointer* y salvando éste en la pila, con el mismo desplazamiento, se consigue crear una lista encadenada de *frames*

- El registro *fp* apunta al marco de la función actual (*stack backtrace*)
- El valor del *fp* salvado apunta al marco de la función *caller*
- A su vez, el valor del *fp* salvado en el marco del *caller* apunta al marco de quién lo invocó



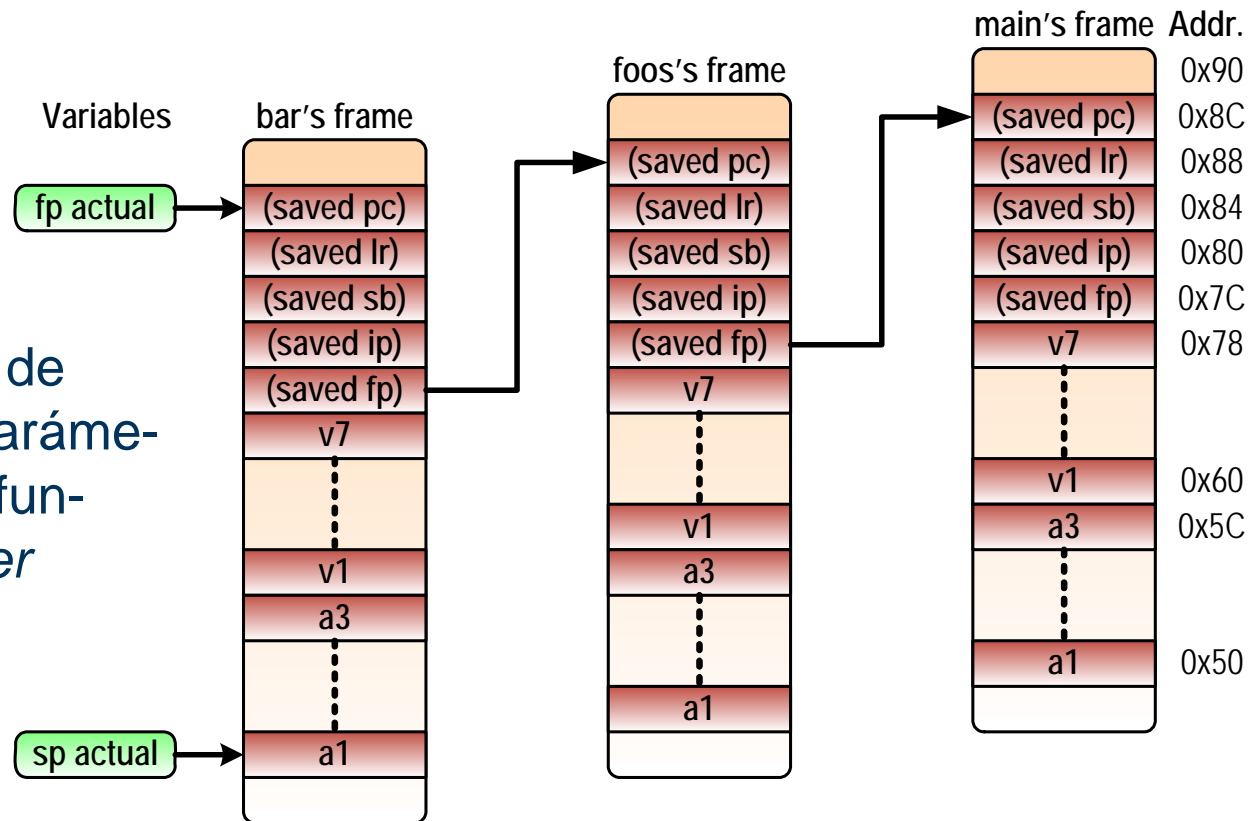


Stack frame: Ejemplo de *backtrace*

Suponer la siguiente secuencia

- main llama a la rutina foo
- foo llama a la rutina bar

- Posibilidad de acceso a parámetros de las funciones *caller*





Stack frame: Creación del *backtrace*

Pasos para crear la estructura del *backtrace*

```
MOV    ip, sp
STMFD  sp!, {a1-a4,v1-v5,sb,fp,ip,lr,pc}
SUB    fp, ip, #4
```

...

...

```
LDMFD  fp, {fp,sp,sb,pc}
```

■ Mediante el registro *ip* podemos almacenar el valor de *sp* a la entrada del procedimiento, lo que nos permitirá tener localizado el marco

■ Es necesario salvar, al menos, los registros *sb*, *fp*, *ir*, *lr*, *pc*, siempre que se desee realizar el *backtrace*

■ En caso de tratarse de rutinas *leaf*, o bien que no hagan uso de la pila, el retorno puede implementarse

```
MOV    pc, lr
/* o bien */
BL     lr
```




Stack frame: Creación del *backtrace*, ejemplo *sumSquare*

 *sumSquare*: devuelve el resultado de $x*x + y$

■ Implementación con *backtrace*

```
11 int __sumSquare(int x, int y); 07 PRESERVE8 ; 8-byte alignes stack
12                               08 AREA RUTINAS, CODE, READONLY
13 int mult (int x, int y) {      09 IMPORT mult
14     return x*y;               10 EXPORT __sumSquare
15 }                             11
16 int main(void) {             12 __sumSquare
17     int a=2, b=3, res;         13 MOV ip, sp
18     /* Llamada a la rutina */ 14 STMFD sp!, {a2, fp, ip, lr, pc}
19     res = __sumSquare(a,b);    15 SUB fp, ip, #4
20 } /* end main() */           16 MOV a2, a1
                                17 BL mult
                                18 LDR a2, [fp, #-16]
                                19 ADD a1, a1, a2
                                20 LDMEA fp, {fp, sp, pc}
                                21 SALIR END
```

- El argumento 2 se salva en la pila para recuperarlo tras la llamada a *mult*
- Uso de LDMEA al tratarse de un acceso a pila mediante *fp*

Name	Value
a	0x00000002
b	0x00000003
res	0x00000007



Stack frame: Manejo de la pila sin necesidad de *backtrace*, ejemplo *sumSquare*

En este ejemplo se muestra la estructura básica de la función *sumSquare* con manejo de la pila y sin *backtrace*

```
07      PRESERVE8 ; 8-byte alignes stack
08      AREA      RUTINAS, CODE, READONLY
09      IMPORT    mult
10      EXPORT    __sumSquare
11
12      __sumSquare
13      SUB        sp, sp, #8
14      STR        lr, [sp, #4]
15      STR        a2, [sp, #0]
16      MOV        a2, a1
17      BL         mult
18      LDR        a2, [sp, #0]
19      ADD        a1, a1, a2
20      LDR        lr, [sp, #4]
21      ADD        sp, sp, #8
22  SALIR  MOV        pc, lr
23      END
```

Prólogo: manejo de la pila para salvar la dirección de retorno y los registros necesarios

Cuerpo: gestión de las operaciones de la rutina

Epílogo: recuperar los registros de la pila, recuperar la dirección de retorno y recuperar el espacio asignado en la pila y retornar



Keil examples



Como ejemplos se recomienda la siguiente lectura:

■ <http://www.cse.cuhk.edu.hk/~phwl/teaching/ceg2400/lec9.pdf>