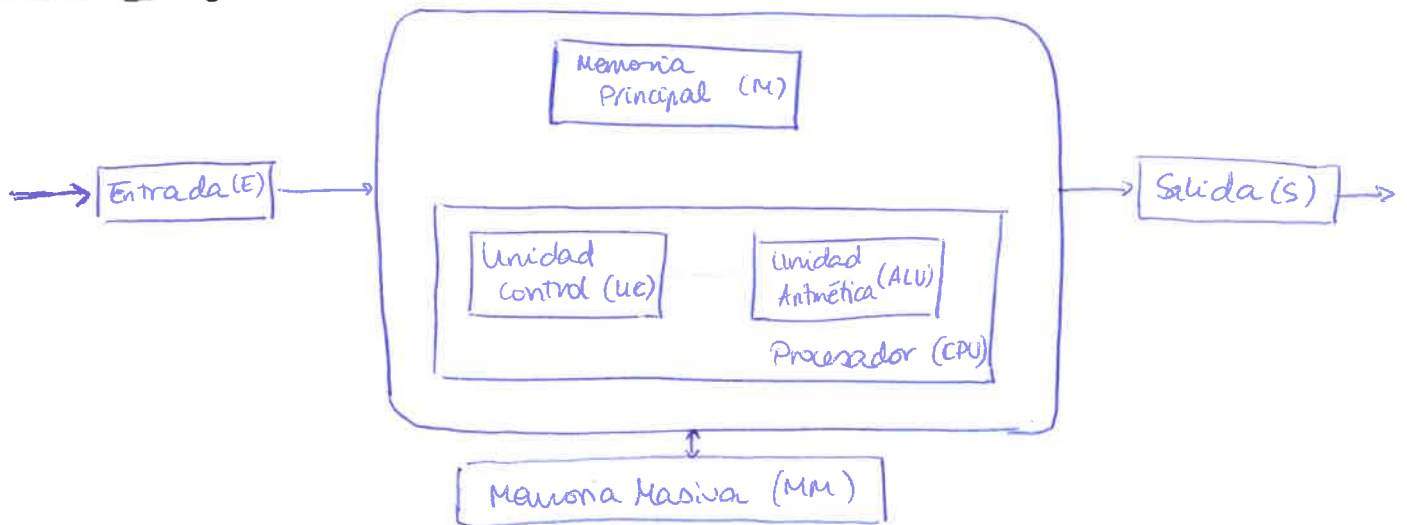


Tema 1 Ec



Registros CPU, Caché (SRAM), Memoria Principal, Discos magnéticos, Cintas Magnéticas

→ Capacidad
 ← Precio
 → Tiempo de acceso.

Big Endian

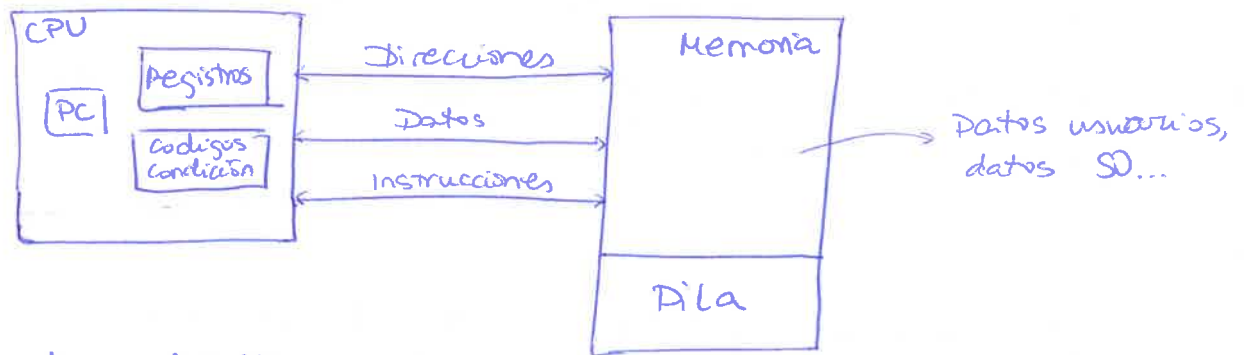
0	ab
1	75
2	23
3	56
⋮	⋮

ab752356...

Little Endian

0	75
1	ab
2	56
3	23
⋮	⋮

Perspectiva del programador en ensamblador

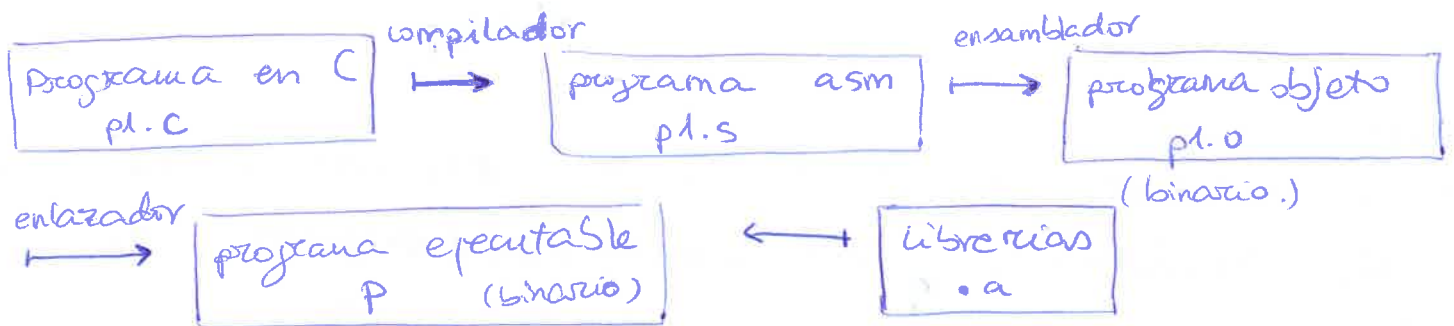


PC: contador programa : dirección de la próxima instrucción

Registros: datos del programa muy usados

Códigos condición / Flags estado : almacenan información de la op. aritmética más reciente.

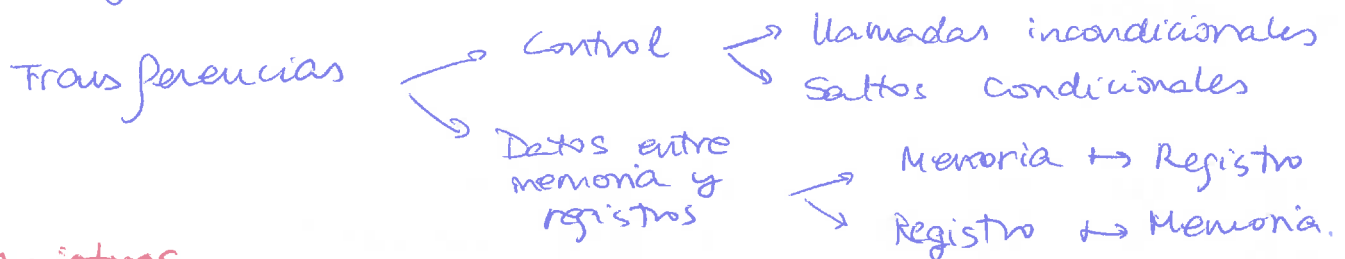
Pila: usada para llamadas a procedimientos.



Ensamblador

Enteros: 4 bytes (32 bits)

Pto flotante: 4, 8 o 10 bytes.



Registros

%eax	acumulador
%ecx	contador
%edx	datos
%ebx	base
%esi	índice fuente
%edi	índice de programa

%esp	puntero pila
%ebp	puntero base
%eip	situación actual del contador de programa.

Operandos

-) Inmediato : datos ctes. \$4
-) Registro : indica alguno de los 8 registros vistos. %eax
-) Memoria : 4 Bytes consecutivos en memoria dada por un registro. (%eax)

Combinaciones de operandos movl

movl	Inmediato:	Reg	movl \$0x4, %eax
		Mem	movl \$-147, (%eax)
	Registro:	Reg	movl %eax, %edx
		Mem	movl %eax, (%edx)
	Memoria:	Reg	movl (%edx), %eax

IMP : No podemos pasar de Mem a Mem con una sola instrucción.

Algunas instrucciones

leal (%eax, %eax, 2), %eax // $%eax + %eax \cdot 2 \rightarrow %eax$
sall \$2, %eax // realiza un desplazamiento a izq
addl suma y lo guarda en la segunda.
subl resta
imull producto
sarl \$2, %eax // realiza un desplazamiento a der
xorl potencia

Registros de un solo bit (Flags)

ZF (flag de cero)

SF (flag signo)

OF (flag de overflow)

CF (flag acarreo)

- Las instrucciones Set X se utilizan para comparar valores.
- Las instrucciones jX se utilizan para saltar a otro lugar del código si cumple una condición.

Bucle Do-while

```

Ej:  movl    $0, %ecx
     .L2:
        movl    %edx, %eax

        shrl    %edx

     jne     .L2
    
```

← programa

← comienzo "do-while"

} acciones dentro del do-while

← comprobación

Bucle Switch

long switch-eg (long x, long y, long z){

long w = 1;

switch (x){

case 1: // .L3

w = y * z;
break;

case 2: // .L4

w = y / z;

case 3; // .L5

w += z;
break;

case 5:

case 6: // .L6

w -= z;
break;

default; // .L2

w = 2;

}
return w;
}

.section .rodata

.align 4

.L7:

.long .L2 # x=0

.long .L3 # x=1

.long .L4 # x=2

.long .L5 # x=3

.long .L2 # x=4

.long .L6 # x=5

.long .L6 # x=6

ENSAMBLADOR:

.L3:
movl 16(%ebp), %eax
imull 12(%ebp), %eax
jmp .L8

.L4:
movl 12(%ebp), %edx
movl %edx, %eax
sarl \$31, %edx
idivl 16(%ebp)
jmp .L9

.L5:
mull \$1, %eax
jmp .L9


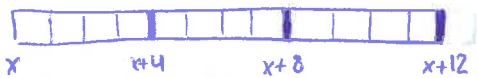

.L6:
movl \$1, %eax
subl 16(%ebp), %eax
jmp .L8

.L2:
movl \$2, %eax
jmp .L8

.L9:
addl 16(%ebp), %eax
jump .L8

.L8:
popl %ebp
ret

Tipos de datos básicos. (32 bits)

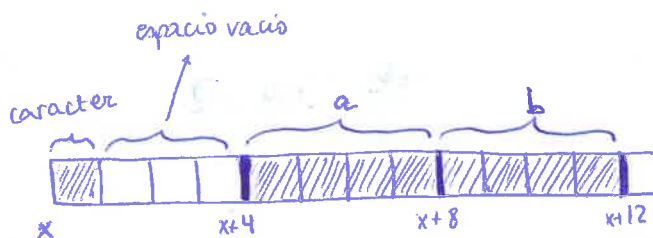
char	1 B	char val[5]	
short	2 B		
float/int	4 B	int val[3]	
double	8 B	double val[2]	
long double	12 B		
como es un puntero —→ char *p[2]			
almacena una dir. de memoria			
que al estar en un ordenador en 32 bits ocupa 4 B.			

Estructuras

La idea de las estructuras es reservar una región contigua de memoria donde los miembros pueden ser de distinto tipo.

El acceso a este tipo de datos se realiza mediante desplazamientos. Ejemplo:

```
struct Nombre {
    char caracter;
    int a, b;
};
```



Se produce espacio vacío ya que las variables deben comenzar en múltiplos suyos, esto es: en

→ int debe comenzar en $x, x+4, x+8, \dots$

→ char puede comenzar en todas las posiciones ya que 1 es múltiplo de todos los elementos.

→ etc.

Tipos de datos básicos (64 bits)

Char	1 B
Short	2 B
float/int	4 B
double	8 B
long double	16 B

Las estructuras son iguales que en los computadores de 32 bits pero teniendo en cuenta que el tamaño de los datos básicos es distinto.

Arrays de Estructuras

En este tipo de dato todos los elementos del vector provienen de un mismo struct, esto es, si el elemento

```
struct algo {
```

```
    |  
    |  
};
```

```
int main() {
```

```
    | algo a, b, c, d;
```

```
}
```

$v[4] = \{a, b, c, d\}$

No puede ser que "a" sea de un struct de dos enteros y "b" sea un struct de 2 chars.

Así, el alineamiento de cada elemento se realiza introduciendo el struct correspondiente en un múltiplo de su tamaño al igual que en las estructuras normales.

Finalmente es razonable pensar que es mejor poner los tipos de datos más grandes que formen el struct antes de aquellos que sean más pequeños. Esto es:

```
struct algo1 {
```

```
    | double d;
```

```
    | int b;
```

```
    | char a;
```

```
};
```

✓

```
struct algo2 {
```

```
    | char a;
```

```
    | int b;
```

```
    | double d;
```

```
}
```

✗

porque
gasta más
espacio.

Uniones

Algunas operaciones aritméticas

incl	Dest	$Dest = Dest + 1$
decl	Dest	$Dest = Dest - 1$
negl	Dest	$Dest = -Dest$
notl	Dest	$Dest = \sim Dest$

Pila en 64 bits

En 64 bits podemos movernos en la pila "hacia arriba" y "hacia abajo". De esta forma, podemos almacenar información por debajo del puntero pila. Ejemplo:

movq	%rbx, -16(%rsp)	# Salva %rbx	%rsp →	Dir. ret
movq	%rbp, -8(%rsp)	# salva %rbp	-8	%rbp
subq	\$16, %rsp	# Reservan marco pila	-16	%rbx
movq	(%rsp), %rbx	# restaurar %rbx	+16	Dir. ret
movq	8(%rsp), %rbp	# restaurar %rbp	+8	%rbp
addq	\$16, %rsp	# libera marco	%rsp →	%rbx

→ Al reservar marco de pila, %rsp actúa como %ebp en 32 bits.

- EIP es el registro de instrucción, almacena la dirección de memoria de la siguiente instrucción a ejecutar.

- ADC es la instrucción de suma con acarreo o realiza el salto condicional jc/jnc combinado con un incremento.

- idiv algo realiza lo siguiente:

El cociente lo lleva a %eax y el resto lo lleva a %edx.

(es como que une lo de %eax y %edx en un solo elemento)

%eax %edx | algo
 ↓
 resto cociente

- Cdq es una instrucción usada para doble precisión que extiende por defecto `%eax` a `EDX:EAX`

Práctica 2

- Asm es una instrucción que nos permite trabajar con ensamblador y con C al mismo tiempo.
- Llamamos convención de llamada al conjunto de alternativas escogidas para pasar parámetros y devolver resultados. En la convención que vamos a adoptar, que es la denominada cdecl permite mezclar ficheros objeto compilados desde fuentes C/C++ con ficheros objeto ensamblados desde fuentes ASM. Esta convención nos recita lo siguiente:

-) Los parámetros se pasan de último a primero.
-) El código de llamada es el que reserva espacio en la pila para los parámetros.
-) El resultado final se suele devolver en `%eax`.

-) Registros
 - EAX, ECX, EDX (salva-invocante). Una función les puede usar sin tener que reservarlos en pila. Es responsabilidad del invocante guardarlos en pila si no quiere perder su valor.
 - EBX, ESI, EDI (salva-invocado) Debe guardarlos en pila, usarlos y posteriormente restaurar su valor si se quiere cambiar el contenido.
 - ESP, EBP No deben manipularse.

puntero pila ↑ ↑ marco de pila

Cualquier función en la convención cdecl comienza de la siguiente manera:

```
push    %ebp.  
mov     %esp, %ebp  
:  
ret
```

TEMA 2 (Tema 3 de SWAD)

Unidad Tratamiento ALU) Unidad de procesamiento de datos y aritmético-lógica. Contiene circuitos electrónicos con los que se hacen las operaciones.

Unidad de Control (UC) detecta señales de estado procedentes de otras unidades, capta de la memoria una a una las instrucciones y genera señales de control dirigidas. Contiene además un reloj que sincroniza todas las operaciones elementales de la computadora. La UC interpreta y controla la ejecución de instrucciones leídas de memoria en dos fases:

1) Fase de captación de instrucción:

Leer la dirección de la instrucción a ejecutar.

Leerla de memoria.

Llevarla al registro adecuado para su ejecución

Incrementar PC

2) Fase ejecución

Decodificación de la instrucción

Ejecución bajo control de UC

Se realizan las operaciones del CODOP (código operación)

Se generan las señales de control oportunas.

3 2 formas de diseñar la UC

- Control cableado
- Control microprogramado

Unidad Tratamiento.

Es el conjunto de elementos del procesador no dedicados al control, por lo que incluye a la ALU, multiplexores, bistables,

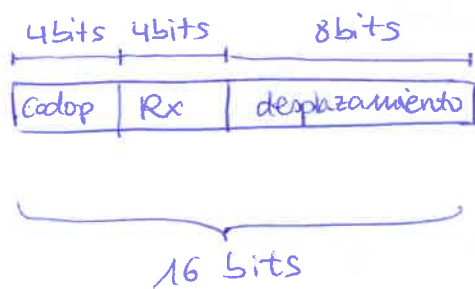
Se suelen realizar las operaciones más sencillas. Los procesadores incluyen el FPU (unidad coma flotante) que realiza operaciones más complejas.

Vamos a trabajar con un ordenador didáctico elemental CODE-2.

Mirar dispositivos 15, 16, 17, 18 suad. para entender el funcionamiento de cada elemento de la ALU.

Además la unid. tratamiento está compuesta por 2 buses que recorren los elementos del arcauto.

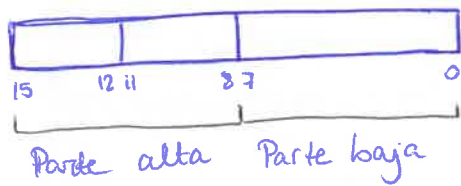
- Debemos tener claro que en el ordenador simple en el que trabajamos, solo contiene 16 registros con 16 bits cada uno y de la siguiente forma:



Cada registro almacena el cod. operación, Rx y el desplazamiento.

Así, el desplazamiento es donde se incluyen ctes en el registro, de manera que en estos registros no podemos introducir ctes con más de 8 bits.

- Se necesitan 4 bits para codificar un registro pero este almacena 16 bits. Además, se tiene:



Unidad de control cableada.

Analiza e interpreta una instrucción almacenada en IR y los valores de los registros.

Genera 29 señales de control que monitorizan el funcionamiento de los distintos elementos.

Estas señales producen microoperaciones

La unidad de control cableada de nuestra máquina se ha diseñado para ejecutar 16 órdenes que son las mostradas en la dispositivo 31.

- Para los germinos de las diapositivas 35, 36 simplemente debemos mostrar las microoperaciones que se realizan en cada ciclo de las 16 operaciones principales de las que consta el MC cableado, mirando diapo 33, 34.

Unidad de Control microprogramada

- La UC genera en cada pulso de reloj un vector de 29 microordenes. Estos vectores se pueden almacenar en la memoria de control (MC) que es una memoria ROM.
- La UC microprogramada está formada por la memoria de control y secuenciador de la memoria de control, que genera las direcciones de las posiciones de las palabras de memoria a leer.
- Podemos ver una imagen de la UC multiprogramada en la diapo 49.
- DMC es el registro que almacena la dir. de la memoria de control.
- El secuenciador va generando la dirección de memoria del siguiente vector.
- Una microinstrucción es un conjunto de bits correspondiente a las microordenes que se ejecutan al mismo tiempo junto con los bits que determinan la dirección de la microinstrucción siguiente.
- En cada palabra de la MC se almacena una microinstrucción. (1 palabra = 2B)
- Campos de una microinstrucción

{	• TD	(tipo direccionamiento)
	• BE	(bitenable que especifica la condición de salto)
	• DS	(dirección salto)
	• Microordenes	: 29 bits (señales de control.)

- Un microprograma es una secuencia de microinstrucciones que capta o interpreta una instrucción del lenguaje máquina del computador.

- Todo lo relacionado con microprogramas se denomina firmware.

- se reservan para cada micro programa de ejecución 4 microinstrucciones.

Por tanto, la UC microprogramada:

-) Lee de la memoria de control la dirección DMC.
-) carga RMC con la palabra leída de memoria.

→ Microprogramación horizontal: los bits de los microórdenes actúan directamente sobre los elementos que controlan.

→ Microprogramación vertical: para disminuir la longitud de las microinstrucciones las señales se agrupan y codifican en campos específicos.

- Unidad de control cableada es tremendamente rápida pero cualquier modificación requiere una reestructuración. En cambio la UC microprogramada es muy versátil pero más lenta.

- El tamaño de las instrucciones depende del n° de señales que tengas.

Nanoprogramación

1 0 0 0	①
1 0 0 1	②
1 1 0 0	③
1 1 0 1	④

↑ ↑
dir. memoria señales
de las instrucciones

Como en grandes programas se van a repetir múltiples señales, por ej, al sumar y restar se repite la instrucción de mover a la ALU; se hace un alias de estas instrucciones repetidas de la siguiente manera:

10 10	0 0 0
10 11	0 0 1
1 1 00	0 1 0
1 1 01	0 1 1

↑ "alias"

0 0 0	①
0 0 1	②
0 1 0	③
0 1 1	④

Nanomemoria.

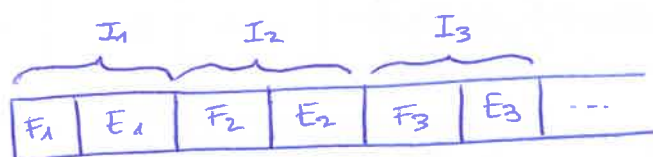
Para mayor entendimiento consultar diapo 64.

TENA 3 (Tema 4 de SWAD)

Para aumentar las prestaciones podemos:

- Mejorar tecnología.
- Reorganizar el hardware (segmentación de cauce)
- Duplicación de hardware (procesador superescalare)

Ejemplo de un procesador segmentado en el que cada instrucción está compuesta por dos fases.

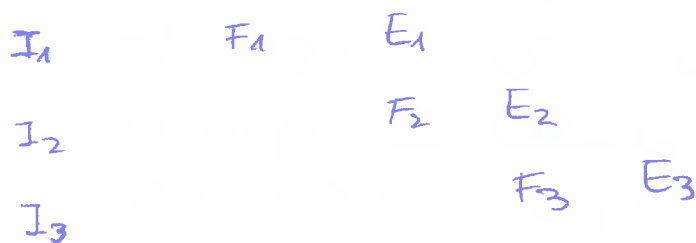


F (Fetch): captación

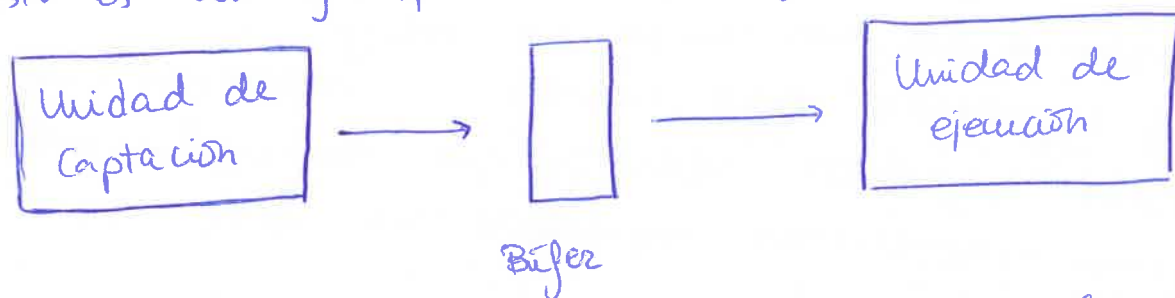
E (Execution): ejecución

En cambio, veamos un ejemplo de instrucción compuesta por dos fases pero en un procesador con segmentación de cauce:

Ciclos reloj:



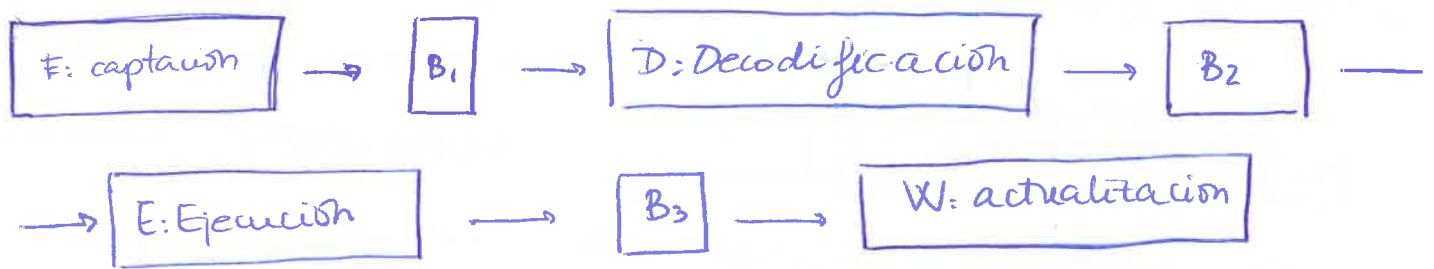
Esto es así ya que \exists el bufer entre etapas, esto es,



un elemento que recibe la información de la unidad de captación y la almacena. Esto permite que la un. captación pueda comenzar a captar una nueva instrucción y el búfer le pasará a la unidad de ejecución mientras la información que almacena.

En general vamos a trabajar con instrucciones de 4 fases. Así

Ciclos reloj:	1	2	3	4	5	6
I_1	F_1	D_1	E_1	W_1		
I_2		F_2	D_2	E_2	W_2	
I_3			F_3	D_3	E_3	W_3
\vdots						



El proceso, tal y como hemos visto antes, sería:

- ① Se lee la información en la unidad de captación.
- ② Pasa la información al búfer.
- ③ La unidad de ejecución lee la info del búfer y al mismo tiempo la unidad de captación puede comenzar a leer la siguiente instrucción.

- La memoria caché permite completar cada etapa del canal en un ciclo de reloj.

La fase de captación debe acceder a memoria que es un proceso muy lento mientras que la caché nos permite captar instrucciones empleando un único ciclo de reloj.

- Aunque debemos tener en cuenta que ante todos los beneficios de la segmentación de canal, esta tiene riesgos.

Existen numerosos problemas que impiden que una etapa se complete en un ciclo.

denotaremos como riesgo a cualquier condición que produzca un atasco.

Tipos riesgos:

- 1) Datos: operandos no disponibles
- 2) Instrucciones: la instrucción no está disponible
- 3) Estructurales: dos instrucciones necesitan el mismo recurso.

5 de ejecución de una instrucción con más de 1 ciclo.

Ciclos reloj: 2 3 4 5 6 7 8 9

Instrucciones:

	I ₁	F ₁	D ₁	E ₁	W ₁				
①	I ₂		F ₂	D ₂	E _{2A}	E _{2B}	E _{2C}	W ₂	
	I ₃		F ₃	D ₃				E ₃	W ₃
	I ₄			F ₄				D ₄	E ₄ W ₄

Etapas

	F:	F ₁	F ₂	F ₂	F ₂	F ₃			
②	D:		D ₁			D ₂	D ₃		
	E:			E ₁			E ₂	E ₃	
	W:				W ₁			W ₂ W ₃	

Instrucción:

	I ₁	F ₁	D ₁	E ₁	W ₁				
③	I ₂		F ₂	D ₂	E ₂	M ₂	W ₂		
	I ₃			F ₃	D ₃	E ₃		W ₃	
	I ₄				F ₄	D ₄		E ₄	
	I ₅					F ₅	D ₅		

• Las instrucciones independientes no tienen este problema.

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

El resultado de una no influye en otra por lo que no da fallos, en cambio.

$$A = 5$$

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

Esto puede dar fallo ya que una de ellas no se puede calcular si no se calcula la otra primero.

Por tanto si la primera de ellas tarda mucho, se produce tráfico, atasco.

Se pueden producir adelantamientos, como podemos observar en la página 15 de las diapositivas, para mejorar la eficiencia.

• Las dependencias de datos las destruye el hardware al decodificar las instrucciones.

Salto incondicionales.

Se producen cuando una instrucción no está disponible a tiempo, esto es, cuando tarda mucho.

Suele desperdiciarse tiempo como consecuencia del salto.

Lo importante es averiguar la dirección a donde se va a saltar.

Para reducir el efecto de los fallos de caché, se suelen captar instrucciones antes de que sean necesarias (precaptación) y se almacenan en una cola de instrucciones.

- D: decodificar: no solo decodifica la instrucción sino que además capta los operandos.

- Memoria principal: es muy barata ya que para cada bit se necesita un único transistor. En cambio en caché se necesitan 6 transistores por cada bit

Salto condicionales.

El riesgo originado por la dependencia entre la condición de salto y el resultado de una operación previa (comprobación de la condición).

O simplemente se puede seguir la segmentación de canal y saltar cuando se comprueba la condición, esto es:

ciclos	1	2	3	4	5	6
I_1 (comparar)	F_1	D_1	E_1	W_1		
I_2 (saltar)		F_2	D_2	E_2		
			F_3	D_3	X	
				F_4	X	
I_k					F_k	D_k

El programa trata a la primera instrucción de manera correcta y sin fallos, pero luego con la segunda instrucción al ejecutarla se da cuenta que tiene que realizar un salto. Por tanto la fase de captación y decodificación de I_3 y la fase de captación de I_4 no sirve de nada y solo ha perdido tiempo.

Por tanto los compiladores actuales intentan predecir los saltos que se van a realizar y poder anticiparse.

Hay 2 tipos de predicciones de saltos:

- Estática : es la misma decisión para cada instrucción.
- Dinámica : cambia según la ejecución del programa

La idea de esto es que en el canal entren solo aquellas instrucciones que sean estrictamente necesarias, esto es, las que van a ser ejecutadas.

- Las dependencias introducidas por los bits de condición dificultan al compilador la labor de reordenar instrucciones.

Funcionamiento Superescalar.

La idea de esto es poder realizar (captar, ejecutar...) dos instrucciones al mismo tiempo. Para ello se duplican (triplican, ... n-uplican) elementos hardware como por ej la ALU. Veamos un ejemplo.

I_1	F_1	D_1	E_{1A}	E_{1B}	E_{1C}	W_1
I_2	F_2	D_2	E_2	W_2		
I_3		F_3	D_3	E_{3A}	E_{3B}	E_{3C} W_3
I_4		F_4	D_4	E_4	W_4	

- Esto provoca un efecto negativo en cuanto a los riesgos, porque aquí un atasco es más pronunciado.
- El compilador puede reordenar instrucciones para evitar riesgos.
- Aquí la dependencia de los datos es mucho mayor ya que se realizan varias instrucciones a la vez.

Algunas consideraciones:

$$T = \frac{N \times S}{R}$$

T = tiempo de ejecución
 N = cantidad de instrucciones
 S = n° medio de ciclos que tarda una
 R = frecuencia del reloj.

Finalmente saber que los mejores procesadores se miden en MIPS (millones instrucciones por segundo).

TENA 1

Registros

- Temporales : $\%eax, \%edx, \%ecx$
S-Invocante
- Temporales : $\%ebx, \%esi, \%edi$
S-Invocado
- Especiales : $\%esp, \%ebp$ (Nos dan la situación de la pila en tiempo de ejecución)
- Hay que tener en cuenta que cada posición de memoria almacena un byte, pero una posición de memoria mide 32 bits de longitud, esto es, su dirección es de 32 bits (suponiendo un computador de 32 bits).

Variables

Variable x32	Tamaño
--------------	--------

Char	1B
Short	2B
Int	4B
Double	8B
Long Double	12B
Punteros	32bits=8B

Variable x64	Tamaño
--------------	--------

Char	1B
Short	2B
Int	4B
Double/Quad	8B
Long Double	16B
Punteros	64bits=8B

Algunos ejemplos de cálculo de direcciones

$\%edx \rightarrow 0xf000$

$\%ecx \rightarrow 0x0100$

Expresión	Cálculo	Dirección
$0x8(\%edx)$	$\%edx + 8$	$0xf008$
$(\%edx, \%ecx)$	$\%edx + \%ecx$	$0xf100$
$(\%edx, \%ecx, 4)$	$\%edx + 4\%ecx$	$0xf104$
$0x80(\%edx, 2)$	$\%edx * 2 + 80$	$0xf082$

Instrucciones

- sall \$2, %eax realiza un desplazamiento hacia la izquierda. Así, sard realiza de igual forma un desplazamiento hacia la derecha, y esta instrucción es igual a shrl.
- addl src, dest realiza la suma de estos valores y almacena el resultado en el segundo. Por el contrario, subl realiza la resta de valores.
- imull realiza el producto. En cambio, idiv es distinta, procedamos a explicarla:
- idiv algo realiza lo siguiente:

%eax %edx algo
 cociente
 :
 resto

El cociente lo lleva a %eax
y el resto lo lleva a %edx.

(Es como que une lo de %eax y %edx en un solo registro).

- xorl realiza la potencia.

- movzbl %al, %eax se encarga de extender %al hasta que ocupe por completo el registro %eax, añadiendo ceros. Pero no cambian los flags de estado.

- jne mira si está o no activado el flag de estado ZF.

- La diferencia entre cmpl y test es que cmpl compara los valores de dos registros restandolos y mirando el flag de estado SF. En cambio test realiza la operación & para comprobar cual es el mayor valor. Pero ambos nos dan el mismo resultado.

2	0	4
0	0	0
1	0	1

Diferencias entre MOV, LEAL y ADD

① `mov (%ebx, %ecx, 4), %eax`

(A) $ebx + ecx * 4 = DATO$;

En este caso DATO es un valor numérico ya que realiza la operación (A) con los valores de los registros correspondientes.

Supongamos que la dirección de memoria 0837AB contiene el valor DATO para próximos ejemplos:

② `leal (%ebx, %ecx, 4), %eax`

leal realiza la misma operación que "mov" pero la diferencia es que el resultado que obtiene es una dirección de memoria, no un valor; ya que la operación (1) ~~se~~ realiza con las direcciones de memoria de los correspondientes registros.

③ `add (%ebx, %ecx, 4), %eax` realiza lo siguiente:

•) Calcula la dirección igual que leal. En este ejemplo, la dir. es 0x0837AB.

•) Ahora cogemos ese dato en memoria, tal y como lo hace mov. $M[0x0837AB] = 25$.

•) Ahora realiza la operación:

$$25 + \%eax = \text{resultado}$$

•) Finalmente realiza la `mov resultado, %eax`.

PODEMOS CONCLUIR DE TODAS ELLAS LO SIGUIENTE:

- MOV si accede a memoria
- LEAL no accede a memoria (por lo que es más rápido).
- ADD si accede a memoria.

- Otro ejemplo de instrucciones podrían ser:
`add %eax, (%ebx, %ecx, 4)` donde la operación difíal está en el destino.

Algunas funciones Muy IMPORTANTES (DE MIANFG)

Call <dir>

- almacena en pila la dirección de retorno
- decrementa %esp (en pila)
- se va a <dir> mediante %eip

Ret <dir>

- Vuelve a la dirección de retorno
- incrementa %esp (en pila)

push <valor>

- inserta <valor> en pila
- decrementa %esp

pop <dest>

- coge el último valor de pila y lo almacena en <dest>
- incrementa %esp.

Quando decimos incrementar esp, "sube" en la pila; y lo correspondiente a decrementar

- Todos los saltos condicionales realizan alguna comprobación previa al salto de los flags de estado excepto jmp que siempre salta sin realizar comprobación alguna.

Push A { `sub $4, %esp`
`mov A, (%esp)`

Pop A { `mov (%esp), A`
`add $4, %esp`

• La instrucción int 0x80 es una instrucción que realiza una interrupción que permite el paso al modo kernel (que es el contrario al modo usuario).

• Los flags de estado son un bit que forma parte de un registro que solo almacena 0 o 1.

0 \rightarrow NO activado 1 \rightarrow Activado.

Hay 4 flags:

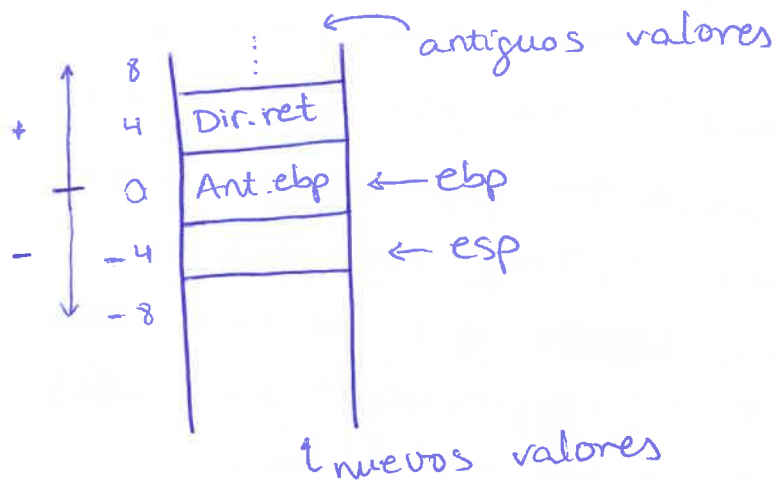
ZF : flag de cero

CF : flag de acarreo

SF : flag de signo

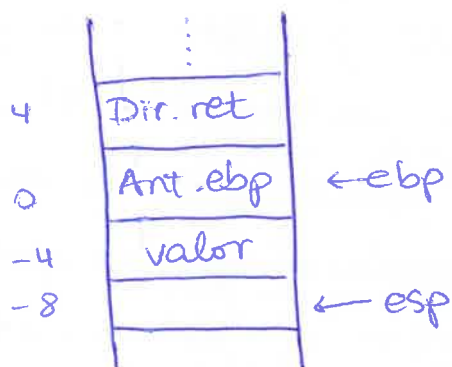
OF : flag de overflow

Funcionamiento de la pila.



La pila cuando comienza un programa tiene esta forma. Supongamos que realizamos: push valor.

Entonces pasaría esto:



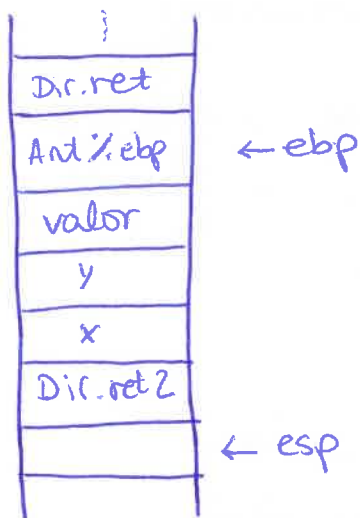
Como vemos, se ha insertado valor en la pila y ha aumentado el %esp.

Además, nuestro convenio nos dice que cuando llamamos a una función los parámetros se pasan antes que la función y además en orden inverso, esto es; supongamos que hacemos una orden como la siguiente:

call suma

suma(int x, int y)

(donde suma, en C tiene los parámetros x e y)



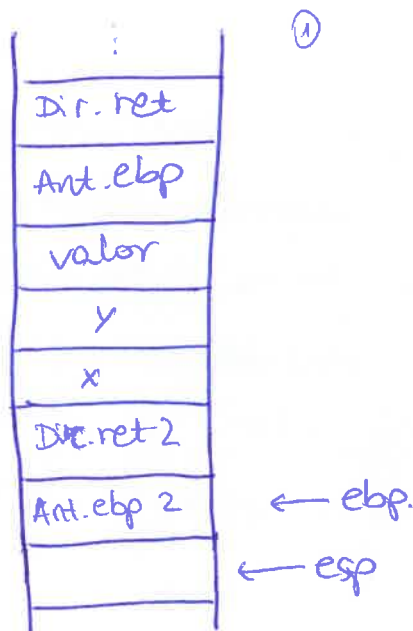
La pila quedaría así.

Por ellos es siempre recomendable (casi obligatorio) cada que se comience una función realizar estas dos instrucciones:

push %ebp

mov %esp, %ebp

Así, tras estas instrucciones el resultado nos da ①.

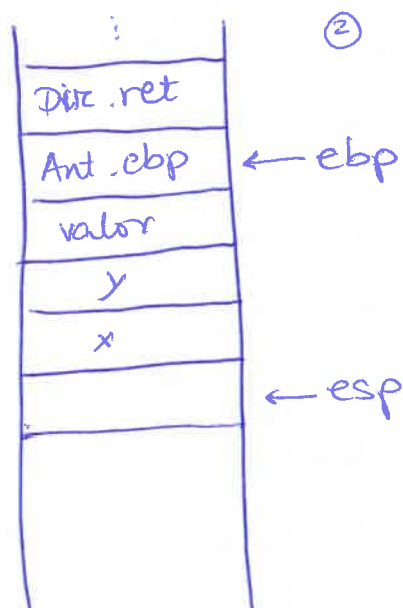


Ahora podemos realizar dentro de la función todos los pasos e introducir todas las instrucciones que queramos. Cuando terminemos debemos realizar un push de aquellas ~~variables~~ ^{valores} que se encuentran en registros temporales (5-Invocados) para no perder su valor.

Tras realizar los push adecuados es recomendable finalizar la función con la instrucción ret.

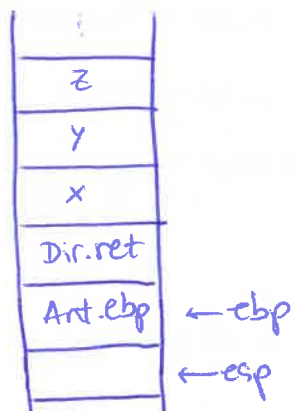
Así, volvemos a tener el esp y el ebp en las posiciones que ocupaban antes de llamar a la función.

Como podemos apreciar en ②



Por último, no se recomienda modificar el registro %esp por nuestra cuenta ya que push y pop lo modifican automáticamente.

Algunos ejemplos de expresiones aritméticas.



$$\%ebp + 16 \rightarrow z$$

$$\%ebp + 12 \rightarrow y$$

$$\%ebp + 8 \rightarrow x$$

$$\text{movl } 8(\%ebp), \%ecx \rightarrow ecx = x$$

$$\text{movl } 12(\%ebp), \%edx \rightarrow edx = y$$

$$\text{leal } (\%edx, \%edx, 2), \%eax \rightarrow eax = 3y$$

$$\text{sll } \$4, \%eax \rightarrow 2 \cdot 2 \cdot 2 \cdot 2 = 16; \quad eax = 16 \cdot 3y = 48y$$

Computadores de 64 bits.

Aquí, en vez de utilizar int se utiliza quad.

Además, en 64 bits se almacena la cifra más significativa en edx, mientras que la cifra menos significativa se almacena en eax.

En 64 bits, tenemos el doble de registros que encontramos en 32 bits. Por ej; el registro %eax de 32 bits corresponde a la parte menos significativa del registro %rax de 64 bits. Además de esto, los ordenadores de 64 bits incluyen otros registros como ahora vamos a ver.

Registros en 64 bits

%rdi Argumento 1

%rsi Argumento 2

%rdx Argumento 3

%rax valor retornado

%rcx Argumento 4

%r8 Argumento 5

%r9 Argumento 6

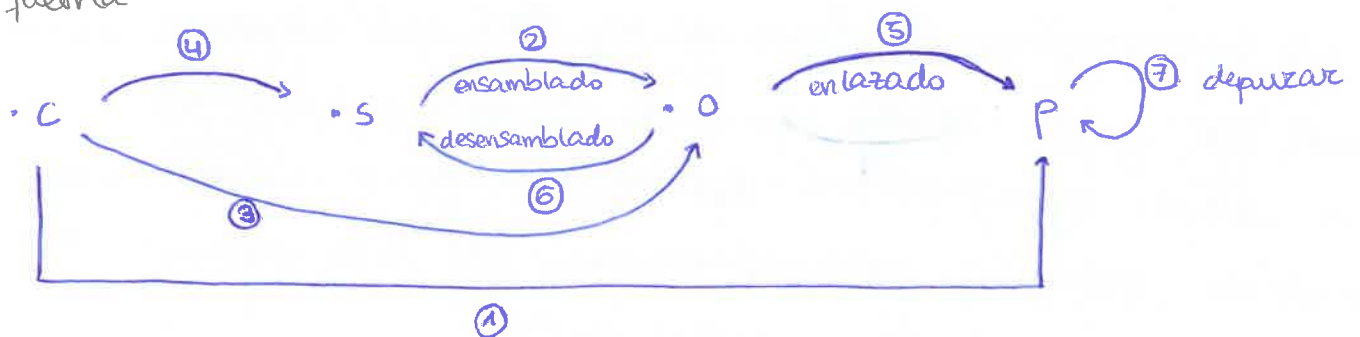
%rsp puntero pila

%r10 }
%r11 } Salva invocante

%r12 }
%r13 } Salva invocado
%r14 }
%r15 }
%r16 }
%r17 }

- En 64 bits los primeros 6 argumentos se almacenan en unos determinados registros (los vistos anteriormente). Si hay más de 6 argumentos iniciales, se almacenan en pila al igual que en 32 bits. Con la diferencia que el incremento básico en 64 bits es de 8B.
- En ensamblador si tenemos por ejemplo un vector `int val [5]` y accedemos a `val [5]`, estamos accediendo a un valor basura pero esta orden no da error.
- En ordenadores de 32 bits las ordenes van precedidas de una letra que es la "l". Esto es, `movl`, `sall`, `addl`, ... En vez de esto, en 64 bits se preceden de la letra "q". Esto es así porque en 32 bits hablamos de long y en 64 bits hablamos de quad (que ocupan 8B).

Esquema de los comandos necesarios.



- ① `gcc -m32 p1.c p2.c -o p`
- ② `as --32 -g saludo.s -o saludo.o`
- ③ `gcc -m32 -c p1.c p2.c`
- ④ `gcc -m32 -O -S p1.c`
- ⑤ `gcc -m32 p1.o p2.o -o p` → poner main global main
`ld -m elf-i386 saludo.o -o saludo`
`ld -m elf-i386 suma.o -o suma -lc -dynamic-linker /lib/ld-linux.so.2`

esto es para cuando queremos que se muestre resultado
- ⑥ `objdump -d p1.o`
- ⑦ `gdb p`