Arq. Lóg. 1

Arquitectura Lógica

Es la visión que tiene un programador de la máquina, sin importar su implementación. Se preocupa de:

- La representación de los números y datos
- Las operaciones realizables
- El lenguaje de máquina o conjunto de instrucciones.

Representación de números

La máquina representa los números en palabras de 8, 16, 32 o 64 bits. Una palabra es una secuencia de bits y sirve para almacenar enteros sin signo, enteros con signo, caracteres, nºs reales, etc.

Usaremos la siguiente notación:

$$x = \boxed{x_{n-1}\, x_{n-2} \ldots x_1\, x_0}$$

x es una palabra de n bits.

El valor numérico almacenado en la palabra x depende de la representación usada:

Arq. Lóg. 2

(i) Enteros sin signo:

$$[\![x]\!]_u = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

Observe que $[\![\ ]\!]_u$ es una función que recibe una secuencia de bits (una palabra) y entrega un $n^o \in [\phi, 2^n - 1]$

(ii) Enteros con signo

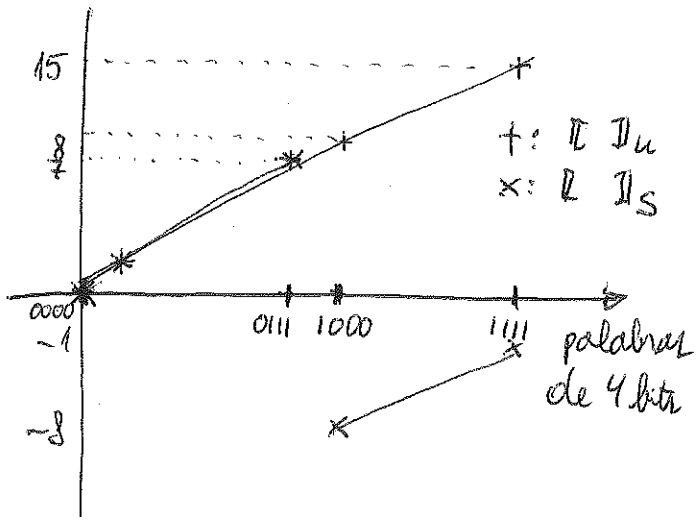
$$[\![x]\!]_s = \begin{cases} [\![x]\!]_u & \text{si } x = \boxed{\phi\, x_{n-2} \ldots x_\phi} \\ [\![x]\!]_u - 2^n & \text{si } x = \boxed{1\, x_{n-2} \ldots x_\phi} \end{cases}$$

con $[\![\ ]\!]_s$: palabras de n bits $\longrightarrow [-2^{n-1}, 2^{n-1} - 1[$.

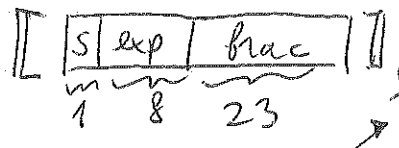
En el siguiente gráfico se observa la diferencia entre $[\![\ ]\!]_u$ y $[\![\ ]\!]_s$ para $n = 4$

Ej:

$[\![0000]\!]_s = \phi$
$[\![\phi 111]\!]_s = 2^{4-1} - 1 = 7$
$[\![1000]\!]_s = -2^{4-1} - 2^4 = -8$
$[\![1111]\!] = -1$
$\vdots$



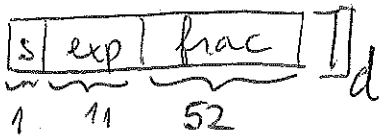
(iii) n^{or} reales de precisión simple (norma IEEE)



$$[\![\ s\ |\ exp\ |\ frac\ ]\!]_f = 1.frac \cdot 2^{exp-127} \cdot \begin{cases} 1 \text{ si } s=0 \\ -1 \text{ si } s=1 \end{cases}$$

(si $exp \neq 0$ y $exp \neq 127$) 255

(iv) n^{or} reales de precisión doble (norma IEEE)

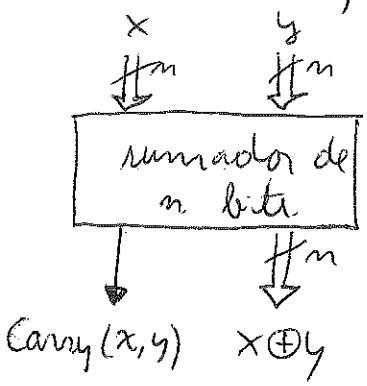


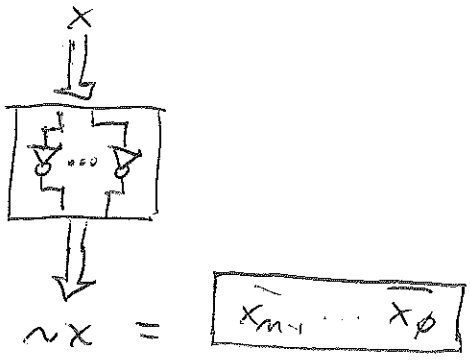
$$[\![\ s\ |\ exp\ |\ frac\ ]\!]_d = \text{similar pero} \cdot 2^{exp-1023}$$

No todas las arquitecturas ofrecen n^{or} reales por Hardware. M32 no los ofrece.

Operaciones entre enteros: Suma de enteros

Def: Sea x, y palabras de n bits.





$\oplus$ es la salida del sumador visto en clase truncado a n bits y corresponde a la suma de palabras

Arq. Lóg. 4

En general se cumple que el número representado por la suma de las palabras x e y es la suma de los números representados por x e y. Es decir:

$$[\![x \oplus y]\!]_u \stackrel{?}{=} [\![x]\!]_u + [\![y]\!]_u$$

Pero dado que el resultado de $\oplus$ se trunca a n bits, la suma no coincide cuando hay desborde.

Ej:

$$\begin{array}{rl} 0101 & \xrightarrow{u} 5 \\ \oplus\ 0011 & \xrightarrow{u} +3 \\ \hline 1000 & \xrightarrow{u} 8 \end{array}$$

✓✓ sí coincide

pero

$$\begin{array}{rl} 0101 & \xrightarrow{u} 5 \\ \oplus\ 1111 & \xrightarrow{u} +15 \\ \hline 0100 & \xrightarrow{u} 4 \end{array}$$

XX no coincide.

Observe que 20 no es representable en 4 bits. Veamos qué ocurre cuando vemos las mismas palabras pero como enteros con signo:

$$\begin{array}{rl} 0101 & \xrightarrow{s} 5 \\ \oplus\ 0011 & \xrightarrow{s} +3 \\ \hline 1000 & \xrightarrow{s} -8 \end{array}$$

XX ¡no coincide!

pero observe que $5+3=8$ no es representable en 4 bits con signo.

Arq. Lóg. 5

$$\begin{array}{rcl} \phi 1 \phi 1 & \xrightarrow{S} & 5 \\ \oplus\ 1\ 1\ 1\ 1 & \xrightarrow{S} & -1 \\ \hline \phi 1 \phi \phi & \xrightarrow{S} & 4 \end{array}$$ ✓✓ sí coincide.

La idea fundamental es que $\oplus$ coincide con la suma si el resultado de la suma es representable en n bits (el tamaño de las palabras). Matemáticamente escribimos:

$$\boxed{\begin{aligned} [\![x \oplus y]\!]_u &\overset{\text{mod } 2^n}{=} [\![x]\!]_u + [\![y]\!]_u \\ [\![x \oplus y]\!]_s &\overset{\text{mod } 2^n}{=} [\![x]\!]_s + [\![y]\!]_s \end{aligned}}$$

con $a \overset{\text{mod } k}{=} b \iff a \bmod k = b \bmod k$

Conclusión fantástica: el sumador visto en clase sirve para sumar números sin signo y números con signo. De ahí el interés de la representación usada para los enteros con signo.

Resta de enteros

Def: $x \ominus y = x \oplus \sim y \oplus 1$

Prop:

$$\boxed{\begin{aligned} [\![x \ominus y]\!]_u &\overset{\text{mod } 2^n}{=} [\![x]\!]_u - [\![y]\!]_u \\ [\![x \ominus y]\!]_s &\overset{\text{mod } 2^n}{=} [\![x]\!]_s - [\![y]\!]_s \end{aligned}}$$

Arq. Lóg. 6

Eg:

13	$\xleftarrow{u}$	1101	$\xrightarrow{s}$	-3
-6	$\xleftarrow{u}$	0110	$\xrightarrow{s}$	-6
7	$\xleftarrow{u}$	0111	$\xrightarrow{s}$	-9
			$\searrow$	7

$$\begin{array}{r} 1101 \\ \oplus\ 1001 \\ \hline 0111 \end{array}$$

Conclusión más fantástica: el sumador visto en clases nos sirve también para restar números, con signo o sin signo.

En realidad, la representación de los números con signo se escogió cuidadosamente de manera que el mismo sumador sirviera para realizar todas las operaciones. Esto se debe a que en los primeros computadores colocar circuitos distintos para realizar estas 4 operaciones (2 sumar y 2 restar) era demasiado caro.

Conversión entre palabras de distinto tamaño

Reducción: se eliminan los bits más significativos.

$$Trunc^{n\to m}\left(\boxed{x_{n-1}\cdots x_{m-1}\cdots x_0}\right) = \boxed{x_{m-1}\cdots x_0}$$

Prop:

$$\boxed{\begin{array}{l} [\![Trunc^{n\to m}(x)]\!]_u \overset{mod\ 2^m}{=} [\![x]\!]_u \\ [\![Trunc^{n\to m}(x)]\!]_s \overset{mod\ 2^m}{=} [\![x]\!]_s \end{array}}$$

Arq. Lóg. 7

Si una palabra de n bits contiene un número representable en m bits, la operación de truncación no altera el número representado.

Ej:

$5 \xleftarrow{u} \emptyset\emptyset1\emptyset1 \xrightarrow{s} 5$

$\downarrow \text{Trunc}^{5\to4}$

$5 \xleftarrow{u} \emptyset1\emptyset1 \xrightarrow{s} 5$ ✓

$26 \xleftarrow{u} 11\emptyset1\emptyset \xrightarrow{s} -6$

$\downarrow \text{Trunc}^{5\to4}$

~~10~~ $\xleftarrow{u} 1\emptyset1\emptyset \xrightarrow{s} -6$ ✓

(porque 26 no es representable en 4 bits, en cambio -6 sí).

Extensión sin signo: Se agregan $\emptyset$s a la izquierda

Def: $\text{Ext}^{u}_{n\to m}(\boxed{x_{n-1}\dots x_\emptyset}) = \boxed{\underbrace{\emptyset\emptyset\dots\emptyset}_{m-n} x_{n-1}\dots x_\emptyset}$

Prop.: $\boxed{[\![\ \text{Ext}^{u}_{n\to m}\ ]\!]_u = [\![\ x\ ]\!]_u}$

Ej.

$13 \xleftarrow{u} 11\emptyset1 \xrightarrow{s} -3$

$\downarrow \text{Ext}^{u}_{4\to5}$

✓ $13 \xleftarrow{u} \emptyset11\emptyset1 \xrightarrow{s}$ ~~13~~

Arq. Lóg. 8

Observe que al extender una palabra con ceros su valor sin signo se preserva. Sin embargo su valor con signo no necesariamente se mantiene, a pesar de que el número -3 es representable en 5 bits.

Extensión con signo: Se repite el bit de signo.

Def: $Ext^{S}_{n\to m}(\boxed{x_{n-1}\cdots x_0}) = \boxed{\underbrace{x_{n-1}\cdots x_{n-1}}_{m-n}\, x_{n-1}\cdots x_0}$

Ahora sí: $\boxed{[\![Ext^{S}_{n\to m}(x)]\!]_S = [\![x]\!]_S}$

Ej: $13 \xleftarrow{4} 1101 \xrightarrow{S} -3$

$\downarrow Ext^{S}_{4\to 5}$

~~20~~ $\xleftarrow{4} 11101 \xrightarrow{S} -3$

Conclusión: La truncación es la misma para números con signo y sin signo, pero la extensión es distinta.

Esto no es grave, porque sigue siendo trivial hacer un circuito que extienda con o sin signo de acuerdo a lo que diga una de sus entradas.

Arq. Lóg 9

Arquitectura Lógica de M32

M32 es un procesador diseñado para propósitos docentes. No existe ninguna implementación real de M32 en un chip. Pero es lo suficientemente simple como para poder explicar en clase cómo se puede implementar en circuitos.

M32 sigue la filosofía RISC: Reduced Instruction Set Computer. Las líneas generales de esta filosofía son:

- Todas las instrucciones se codifican en palabras de 32 bits.
- Pocos formatos de instrucción: 3
- Muchos registros: 31
- Todas las operaciones manipulan sus operandos en registros.
- Instrucciones load/store para transferir datos entre memoria y registros.
- Instrucciones simples: es decir la operación realizada no debe ser compleja.

Estos principios permiten realizar implementaciones eficientes de los procesadores RISC.

Arq. Lóg. 10

La filosofía RISC se fundamenta en el estudio de los procesadores VAX. Estos poseen muchas instrucciones que se codifican en 1, 2, 3, hasta 20 y más bytes. Poseen pocos registros (16), pero las instrucciones pueden tener operandos directamente en memoria, de modo que los registros sólo se usan con fines de optimización. El set de instrucciones implementa operaciones complejas como cálculo de raíces, acceso a arreglos y matrices, etc. Por esta razón este tipo de procesadores se denomina CISC o Complex Instruction Set Computer.

Los procesadores CISC están orientados a la programación en assembler, predominante en los años 60 y 70. Pero en los 80 el assembler es reemplazado por los lenguajes de alto nivel. Los compiladores generan programas que usan un reducido número de instrucciones haciendo que sólo se use el 10% de la circuitería de los VAX.

Por otra parte, implementar eficientemente el set de instrucciones de un VAX resulta difícil y caro, puesto que existe un compromiso entre la eficiencia de ejecución de una instrucción (i.e. el nº de ciclos que tarda en ejecutarse) y la cantidad de transistores que se necesita para implementarla. Con tantas instrucciones,

quedaban pocos transistores para c/u de ellos y por lo tanto los VAXes eran lentos para ejecutar todas sus instrucciones.

De ahí surge la filosofía RISC: la arquitectura debe poseer pocas instrucciones para que éstas puedan implementarse eficientemente. El criterio para dejar fuera una instrucción es:

- Los compiladores no generan la instrucción.
- La operación es demasiado compleja y su uso no es frecuente.

Las operaciones complejas se implementan en software, es decir a partir de una secuencia de instrucciones simples. A menudo esta secuencia se ejecuta más rápido que cuando la misma operación se implementa en una sola instrucción. Esto gracias a la eficiencia con que se logra implementar las instrucciones simples.

Con el paso del tiempo, hoy en día se dispone de un mayor número de transistores para implementar un procesador. Por ello los últimos procesadores RISC incorporan muchas instrucciones que no incluían los primeros procesadores RISC, como la multiplicación, división y las operaciones en punto flotante.

— o —

El set de instrucciones de M32

M32 posee 3 grupos de instrucciones:

- instrucciones de lectura/escritura de memoria
- operaciones aritméticas y lógicas
- instrucciones de control para realizar saltos.

Instrucciones aritmético/lógicas.

Estas instrucciones tienen la forma

(i) <operador> <reg-op1>, <reg-op2>, <reg-dest>
(ii) <operador> <reg-op>, <imm>, <reg-dest>

Los operandos de la forma <reg-?> deben ser alguno de los registros R0, R1, ..., R31,
<imm> es un valor inmediato que debe ser representable en 13 bits con signo, es decir debe estar en el rango $[-4096, 4096[$.

Ej:
```
add  R17, R2, R2 ;   R2 = R17 ⊕ R2
sll  R3 , 3 , R5 ;   R5 = R3 << 3
```

La diferencia entre SRA y SRL es que SRA es el desplazamiento a la derecha para enteros con signo y SRL para enteros sin signo. SRL completa los bits más significativos de la izquierda con 0's.

$$SRL(\boxed{x_{31}\, x_{30} \ldots x_0}, b) = \boxed{0 \ldots 0\, x_{31} \ldots x_b}$$

b veces.

Arq. Lóg. 13

y $SRA(\boxed{x_{31}\, x_{30} \ldots x_0}, b_5) = \boxed{\underbrace{x_{31} \ldots x_{31}}_{b_5 \text{ veces}}\; x_{31} \ldots x_{b_5}}$

SRL equivale a dividir entero sin signo por 2^{b_5}
SRA " " " " " con " " 2^{b_5}

Obs: el valor de b_5 es siempre positivo.

Para realizar aritmética de 64 o más bits se usan las instrucciones ADDX y SUBX. Estas instrucciones trabajan con el bit C del registro de estado SR.

Si en R1 y R2 tenemos un entero de 64 bits
R3 y R4 " " " " " "
y queremos dejar en
R5 y R6 la suma de $\boxed{R1|R2} \oplus \boxed{R3|R4}$

basta ejecutar:

```
add  R2, R4, R6
addx R1, R3, R5
```

El bit de acarreo al sumar R2 con R4 queda almacenado en C (carry).
La instrucción addx suma R1, R3 y el acarreo almacenado en C.

Instrucciones de salto

Los saltos condicionales se realizan con:

b<cond> <label> (label = etiqueta)

Esta instrucción se usa normalmente en conjunto con sub para realizar comparaciones. Por ejemplo si deseamos saltar a la etiqueta MAYOR si R3 > R4 (datos con signo):

```
    sub  R3, R4, R0
    bg   MAYOR
    :
MAYOR:
    :
```

R0 → se usa para descartar el resultado de una operación cuando se quiere sólo modificar SR.

R0 siempre vale 0

La instrucción SUB modifica el contenido de SR para describir las características del resultado:

SR = | Z | V | S | C |

Z = 1 si el resultado R3 ⊖ R4 fue 0.
S = 1 " " " " fue < 0
C = 1 " hubo acarreo al calcular R3 ⊕ ~R4 ⊕ 1
V = 1 " " desborde al calcular R3 ⊖ R4, es decir si R3 − R4 es representable con signo en 32 bits.

La instrucción bgt mira el valor de SR para decidir si salta o no. El criterio es si el resultado de la resta fue $> \emptyset$, es decir cuando $Z = \emptyset$ y $S = \emptyset$ (el resultado no fue $\emptyset$ y sí fue positivo).

— o —

Los llamados a procedimientos y retornos se efectúan con la instrucción:

jmpl <reg-proc>, <reg-ret>

<reg-proc> es un registro que contiene la dirección del procedimiento que se quiere invocar.
<reg-ret> es el registro en donde se coloca la dirección de retorno, es decir la instrucción siguiente a jmpl.

Para retornar se usa la misma instrucción:

jmpl <reg-ret>, R$\emptyset$

<reg-ret> es el registro en donde quedó la dirección de retorno. Se usa R$\emptyset$ para descartar la dirección siguiente a jmpl.

Instrucciones de lectura/escritura de memoria:

Lectura: <load> [<reg-base> + <reg-desp>], <reg-dest>
<load> [<reg-base> + <imm>], <reg-dest>

especifica la dirección.

La instrucción <load> depende del tamaño del dato a leer y de su representación con o sin signo. Como el dato se deja en un registro de 32 bits, es necesario indicar el tipo de conversión cuando se lee un byte o una media palabra.

tamaño \ rep.	sin signo	con signo	
palabra	ldw	ldw	no hay conversión
halfword	ldruh	ldrsh	
byte	ldrub	ldrsb	

Ej:

ldw [R1+12], R2 ; $R2 = M_{em}^{w}[R1+12]$.

ldrsb [R1+R4], R1 ; $R1 = Ext_{s}^{8\to32}(M_{em}^{b}[R1+R4])$

ldrub [R1+R4], R1 ; $R1 = Ext_{u}^{8\to32}(M_{em}^{b}[R1+R4])$

ldrsh [R1+R4], R1 ; $R1 = Ext_{s}^{16\to32}(M_{em}^{h}[R1+R4])$

Escritura `<store> <reg>, [<reg-base> + <reg-desp>]`
`<store> <reg>, [<reg-base> + <imm>]`

dirección en memoria.

En donde `<store>` puede ser stw, sth o stb dependiendo de si se escribe en una palabra en memoria, una media palabra o un byte.

Ej: stb R1, [R2 + 5] ; $Mem^{b}[R2+5] = Trunc^{32\to 8}(R1)$

Observe que como la operación Trunc es la misma para enteros con o sin signo no es necesario especificar el tipo de representación.

Alineamiento: M32 exige que todas las palabras estén alineadas a direcciones múltiplos de 4 y todas las medias palabras deben tener dirección múltiplo de 2.

Por lo tanto es un error escribir una palabra en una dirección que no sea múltiplo de 4. Ej:

stw R1, [R0 + 1] ; R0 + 1 ≡ la dirección 1
¡0!
⟹ error.

Este tipo de errores genera una interrupción en los sistemas Unix. Cuando un proceso incurre en este error, el shell despliega el mensaje:

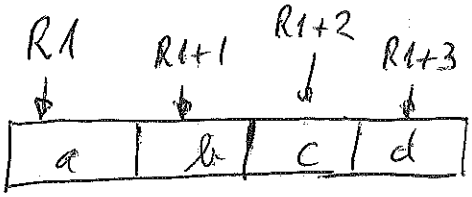
bus error

Y se detiene la ejecución del procesador.

Little Endian y Big Endian

M32 es una arquitectura big-endian. Esto significa que la dirección que se calcula en un load o store corresponde a la del byte más significativo:

Ej: Sea R1 tq



Entonces ldw [R1+0], R2

$$\Rightarrow \boxed{R2 = a \cdot 256^3 + b \cdot 256^2 + c \cdot 256 + d}$$

En cambio en las arquitecturas little-endian

$$\boxed{R2 = a + b \cdot 256 + c \cdot 256^2 + d \cdot 256^3}$$

Es decir R1 apunta al byte menos significativo dentro de la palabra.

El siguiente programa determina si una arquitectura es little o big-endian.

```
main()
{   int w = 1;
    if (*(char*)&w == 1) printf("little endian\n");
    else printf("big endian\n");
}
```

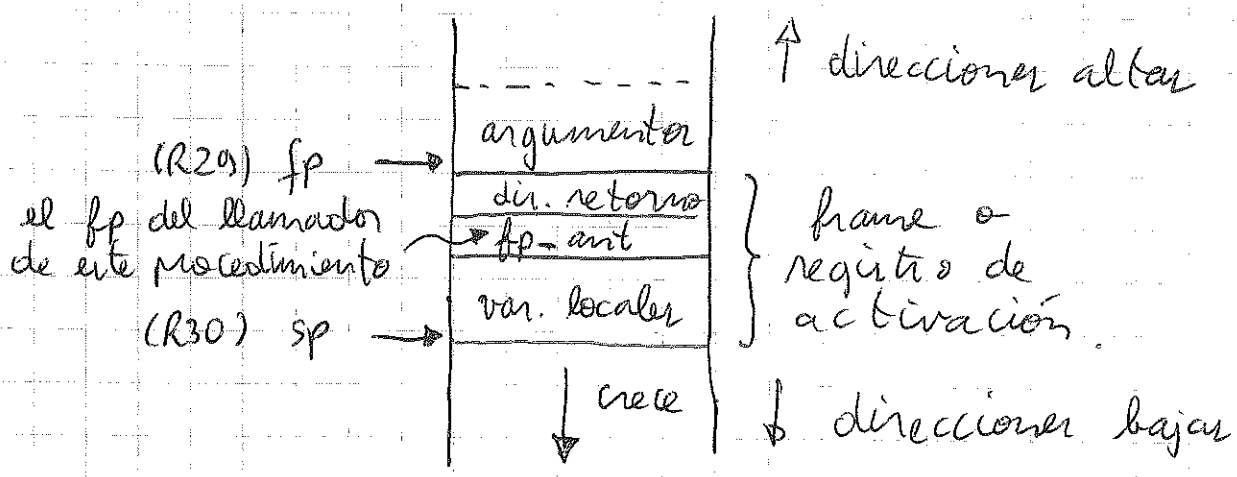
¿Por qué? &w es la dirección de la palabra w que contiene un 1. Al colocar *(char*)&w, se lee el primer byte de w. Si la arquitectura es little endian en ese byte se encuentra el 1. Si la arquitectura es big endian, ese byte contiene un 0.

Compilación de programas C a assembler M32.

Para implementar la recursividad, C utiliza una pila. En la pila se almacenan los registros de activación de los procedimientos. En éste, se colocan las variables locales, la dirección de retorno, etc. En la pila se colocan también los argumentos al invocar procedimientos. Para implementar este esquema es necesario destinar algunos registros para que apunten a la pila:

fp (frame pointer): apunta al registro de activación (o frame) del procedimiento en ejecución. Por convención es el registro R29.

sp (stack pointer): apunta al tope de la pila. Por convención es el registro R30



La pila crece normalmente hacia las direcciones bajas de la memoria.

Veamos como se traduce el siguiente procedimiento a instrucciones de máquina:

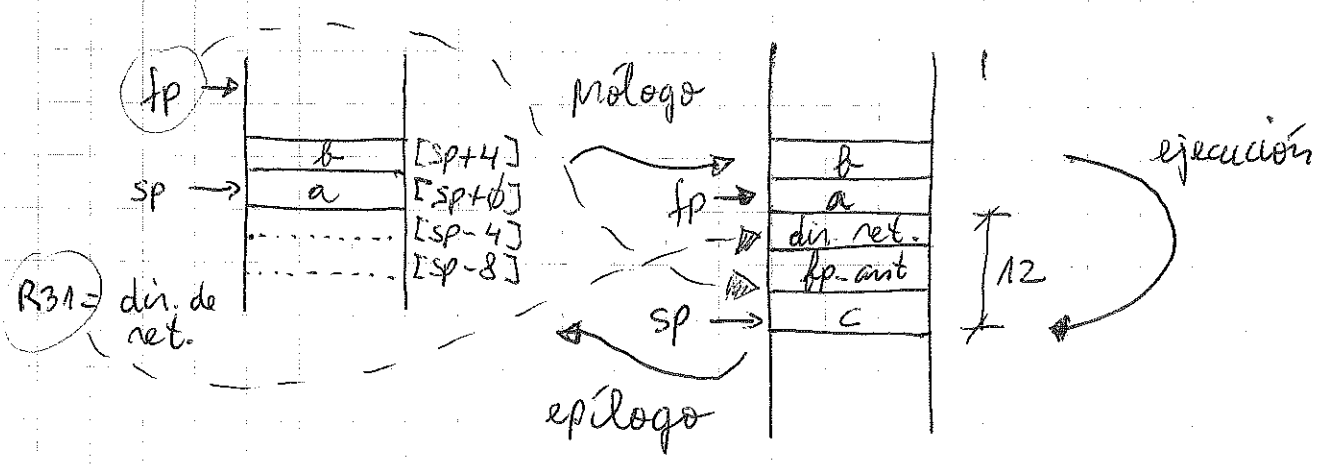
Ej:
```
int P (int a, int b)
{ int c= a+b;
  return c;
}
```

Al ingresar a un procedimiento, fp apunta al registro de activación del llamador y sp apunta al 1er argumento. Además la dirección de retorno se recibe en un registro que será R31 para M32. La forma general de un procedimiento será:

- prólogo para crear el registro de activación
- ejecución del procedimiento
- epílogo para destruir el registro de activación.

Arq. Lóg. 21

El estado de la pila y los registros antes, durante y después de la ejecución será entonces:



En el prólogo se resguardan la dir. de retorno y el fp del llamador para poder restaurar los antes de retornar:

```
                P:
prólogo  {  stw   R31, [R30-4]   ; SP[-1] = dir. ret.
            stw   R29, [R30-8]   ; SP[-2] = fp
            add   R30, 0, R29    ; fp = sp
            sub   R29, 12, R30   ; sp -= sizeof(<frame>)
ejecución {ldw   [R29+0], R1     ; R1 = SP[0]  (a)
c = a+b     ldw   [R29+4], R2    ; R2 = SP[1]  (b)
            add   R1, R2, R3
            stw   R3, [R29-12]   ; c = R1+R2
return c    ldw   [R29-12], R1   ; R1 = c
epílogo  {  add   R29, 0, R30    ; sp = fp
            ldw   [R30-8], R29   ; fp = SP[-2]
            ldw   [R30-4], R31   ; dir.ret. = SP[-1]
            jmpl  R31, R0        ; goto dir. ret.
```

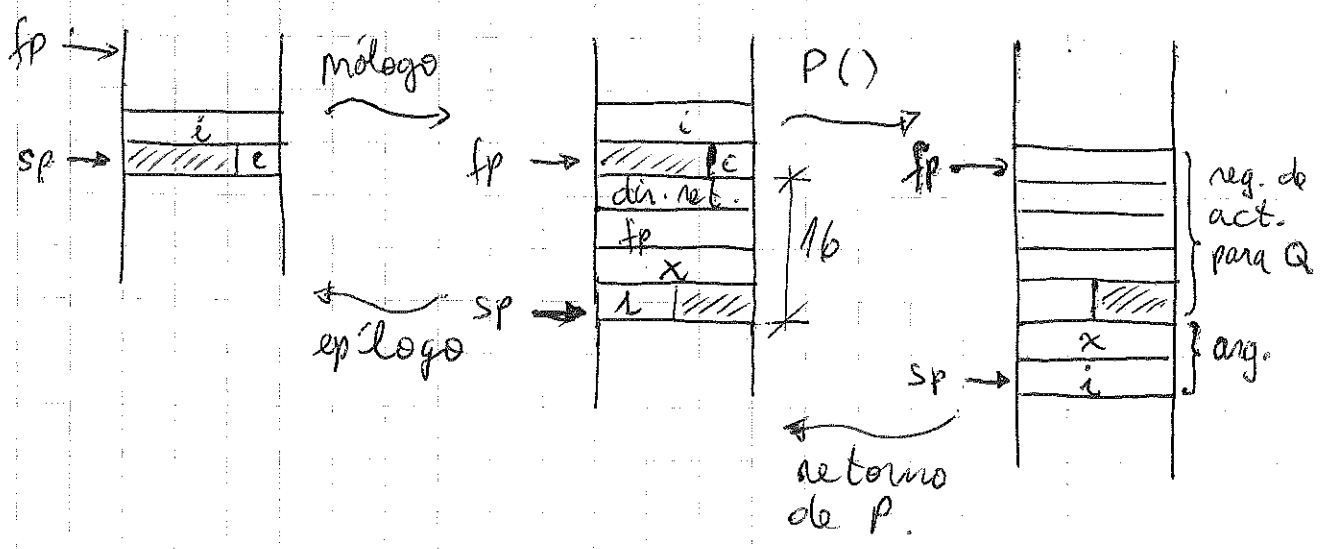
Otro ejemplo:

```
int Q ( unsigned char c, int i)
{   int x = c+1
    short r = P(i,x);
    if (r >= 5)  x >>= 1
    return x & c;
}
```

Pila antes y después de ejecutar Q

Pila durante la ejecución de Q

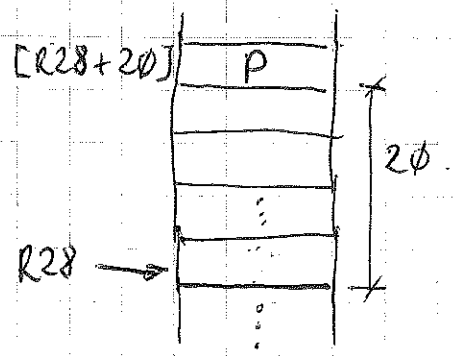
Pila antes de invocar a P.



Para invocar P se colocan los argumentos i y x en el tope de la pila. Luego se transfiere el control a P usando la instrucción jmpl que permite colocar la dirección de retorno en R31. Cuando P retorna, la pila queda exactamente en el estado en que estaba antes de invocar P, por lo que hay que desapilar los argumentos.

Arq. Lóg. 23

Para obtener la dirección de P suponemos que existe una tabla de punteros a todos los procedimientos del programa en ejecución. Esta tabla será referenciada por R28.



prólogo
```
Q: stw R31, [R30-4]
   stw R29, [R30-8]
   add R30, 0, R29     tamaño del reg. de
   sub R29, 16, R30    activación de Q.
```

```
x = c+1
   ldub [R29+3], R1 ; R1 = Ext^u_{b→w}(c)
   add  R1, 1, R2   ; R2 = R1+1
   stw  R2, [R29-12]; x = R2
```

```
push x
   ldw [R29-12], R1 ; R1 = x
   sub R30, 4, R30  } *--sp = R1
   stw R1, [R30+0]  }
push i
   ldw [R29+4], R1  ; R1 = i
   sub R30, 4, R30  } *--sp = R1
   stw R1, [R30+0]  }
P(...)
   ldw [R28+20], R1 } R1 = dir. de P
   jmpl R1, R31     } R31 = PC+4, PC = R1
desapila argumentos
   add R30, 8, R30  ; pop ; pop.
x = P(...)
   sth R1, [R29-16] ; x = Trunc_{w→s}(R1)
```

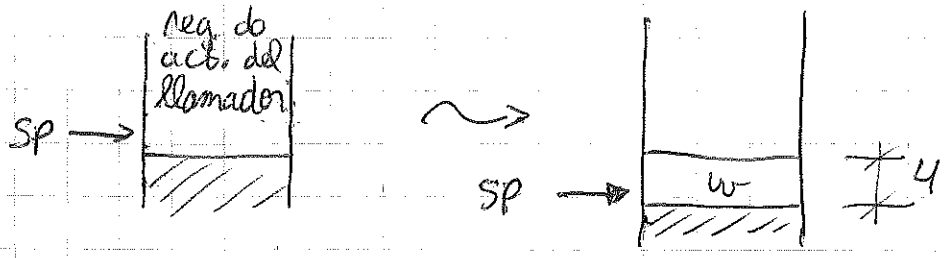
```
if(x>=5) { ldsh [R29-16], R1   ; R1 = x
           sub  R1, 5, R0      ; R1-5 ?
           bl   endif.         ; if R1-5<0 goto ...
-------------------------------------------------
x >>= 1  { ldsw [R29-12], R1   ; R1 = x
           sra  R1, 1, R2      ; R2 = x>>1
           stw  R2, [R29-12]   ; x = R2.
-------------------------------------------------
endif:
return x&c { ldsw [R29-12], R1   ; R1 = x
             ldub [R29+3], R2    ; R2 = Ext_{u b->w}(c)
             and  R1, R2, R1     ; R1 = R1 & R2.
-------------------------------------------------
epílogo { add  R29, 0, R30
          ldw  [R30-8], R29
          ldw  [R30-4], R31
          jmpl R31, R0
```

Hagamos ahora un procedimiento que retorna
1 si la arquitectura es little-endian y
0 si " " " big-endian.

```
int LittleEndian()
{
    int w = 1;
    return *(char *)&w;
}
```

Este procedimiento no recibe argumentos, pero necesita crear un registro de activación para w.

antes de invocar Little Endian



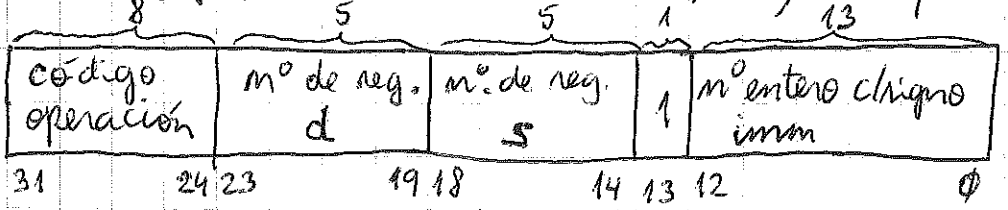
Como este procedimiento no invoca otros procedimientos, no es necesario guardar fp ni la dirección de retorno en el registro de activación, cuyo tamaño será entonces de solo 4 bytes. Esta es una optimización que no todos los compiladores hacen:

Little Endian:

```
prólogo   { sub  R30, 4, R30 ; crea el reg. de activación
w = 1     { add  RØ, 1, R1   ; R1 = 1
          { stw  R1, [R3Ø+Ø] ; w = R1
return
*(char*)&w { ldsb [R3Ø+Ø], R1 ; nótese que R3Ø = &w
epílogo   { add  R3Ø, 4, R3Ø
          { jmpl R31, RØ
```

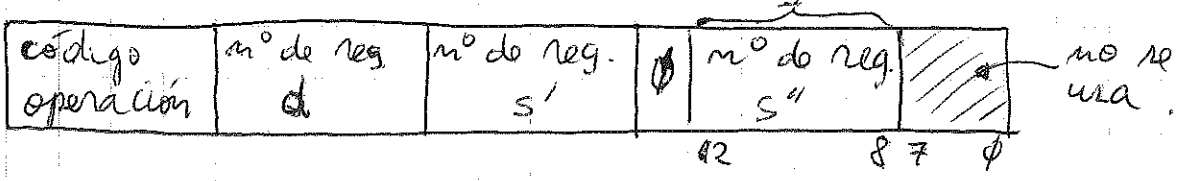
Formatos de instrucción

Dijimos que en M32 todas las instrucciones se codifican en palabras de 32 bits. Además la ubicación de la información en las instrucción se denomina formato de la instrucción. En M32 sólo hay 3 formatos:



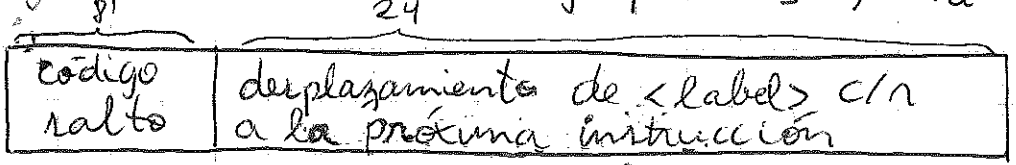
para las instrucciones:

aritmético lógicas: <op> Rs, imm, Rd
lectura de memoria: <load> [Rs + imm], Rd
escritura en memoria: <store> Rd, [Rs + im]



para instrucciones:

aritmético lógicas: <op> Rs', Rs'', Rd
lectura en memoria: <load> [Rs' + Rs''], Rd
escritura en memoria: <store> Rd, [Rs' + Rs'']
jmpl : jmpl Rs', Rd



para saltos condicionales: b-<cond> <label>