PROBLEMAS: El repertorio de instrucciones

7. Sea un computador de instrucción única que ejecuta la operación `restar_y_saltar_si_negativo` (`rsn`) tomando como operandos tres posiciones de memoria `a`, `b` y `c`. La funcionalidad se describe a continuación:

```
rsn a,b,c      mem[a] = mem[a] - mem[b]
               si (mem[a] < 0) entonces ir a c
```

   Como vemos, la instrucción resta el valor contenido en la posición `b` del valor contenido en la posición `a` salvando el resultado en la posición `a`. Si el resultado es menor que 0, se ejecuta la instrucción contenida en la posición `c` y en caso contrario la siguiente.

   Al objeto de cumplir con la funcionalidad de saltar si la diferencia es negativa, los valores contenidos en la memoria se interpretan como enteros codificados en complemento a dos.

   Asumimos que disponemos de una posición de memoria de sólo lectura llamada `uno` que contiene el valor 1.

   Sabiendo todo esto, escriba las secuencias de código que realicen las siguientes operaciones e identifique cada una con un nombre para poderlas usar como macros. Se pueden usar las posiciones de memoria temporales o auxiliares que se desee.

   a) Escriba un 0 en la posición de memoria `destino`.

   b) Copie el contenido de la posición de memoria `fuente` en la posición de memoria `destino`.

   c) Copie en la posición de memoria `neg-fuente` el negativo del valor contenido en `fuente`.

   d) Copie en la posición de memoria `not-fuente` el complemento a uno del valor contenido en `fuente`.

   e) Detecte el signo y el cero del operando `fuente`. Si es negativo cargue -1 en el operando de memoria que hace de bandera de signo `sf-fuente` y 0 en caso contrario. Si el operando es cero cargue -1 en el operando que hace de bandera de cero `zf-fuente` y 0 en caso contrario.

   f) Escriba una secuencia de código que implemente un salto incondicional a la posición de memoria `#destino`.

   g) Escriba una secuencia de código que salte a `#destino` si la bandera de cero `zf-fuente` está activada. La bandera no se puede destruir.

   h) Realice el incremento (suma de la unidad) al valor contenido en el operando `fuente`. Asumimos que el operando no tiene signo y no escribimos banderas de estado.

   i) Realice el decremento (resta de la unidad) al valor contenido en el operando `fuente`. Asumimos que el operando no tiene signo y no escribimos banderas de estado.

   j) Realice la suma de los valores contenidos en las posiciones de memoria `sumando-a` y `sumando-b` salvando el resultado en la primera de ellas. Asumimos que los operandos no tienen signo y no escribimos banderas de estado.

   k) Realice la resta del valor contenido en la posición de memoria `sustraendo` del de la posición `minuendo` salvando el resultado en la segunda de ellas. Asumimos que los operandos no tienen signo y no escribimos banderas de estado.

   l) Escriba la secuencia de código que realiza el producto de un operando llamado `multiplicando` por otro operando llamado `multiplicador` salvando el resultado en un tercer operando llamado `producto`.

SOLUCIÓN:

a) Esta macro la llamaremos `@setzero destino`. Para poner a 0 la posición de memoria `destino` basta con ejecutar una instrucción:

```
rsn destino, destino, pc+1
```

b) La macro `@mov destino,fuente,auxiliar` copia el contenido de la posición de memoria `fuente` a la posición `destino`:

```
@setzero auxiliar
@setzero destino
rsn auxiliar, fuente, pc+1
rsn destino, auxiliar, pc+1
```

Independientemente de que el resultado sea negativo, se pasa a ejecutar la siguiente instrucción.

c) La macro `@neg neg-fuente,fuente` copia el negativo del operando `fuente` en el operando `neg-fuente`:

```
@setzero neg-fuente
rsn neg-fuente, fuente, pc+1
```

Independientemente de que el resultado sea negativo, se pasa a ejecutar la siguiente instrucción.

d) La macro `@not not-fuente,fuente` copia el complemento a uno del operando `fuente` en el operando `not-fuente`:

```
@setzero not-fuente
rsn not-fuente, uno, pc+1
rsn not-fuente, fuente, pc+1
```

Si a 0 le restamos 1 obtenemos una representación con todos los bits a '1', es decir, $1 \cdots 1$ y si a dicha representación le restamos cualquier valor obtenemos el complemento a uno de ese valor. El resultado de restar 1 a 0 es negativo pero forzamos a ejecutar la siguiente instrucción haciendo que el destino sea `pc+1`.

e) La macro `@set_sz sf-fuente,zf-fuente,fuente,auxiliar` carga las posiciones de memoria que hacen de banderas de signo y cero del operando `fuente` con los valores que describen el estado de dicho operando:

```
@setzero auxiliar
@setzero sf-fuente
@setzero zf-fuente
rsn fuente, auxiliar, #negativo
rsn auxiliar, fuente, #fin
nulo:
rsn zf-fuente, uno, #fin
#negativo:
rsn sf-fuente, uno, pc+1
#fin:
```

f) La macro `@jmp auxiliar,#destino` salta a la posición de memoria dada por la dirección (o posición) `#destino`.

```
@setzero auxiliar
rsn auxiliar, uno, #destino
```

g) La macro `@bz zf-fuente,#destino,auxiliar` salta a `#destino` si la bandera `zf-fuente` está a -1, es decir activada. Cuando no está activada es 0.

```
@setzero auxiliar
rsn zf-fuente, auxiliar, #destino
```

h) La macro `@inc fuente,auxiliar` suma uno al operando `fuente`. Trata el valor contenido en el operando como número sin signo y por esa razón ejecuta la siguiente instrucción en la secuencia independientemente de que el resultado pudiera ser considerado negativo. No actualiza banderas de estado.

```
@neg auxiliar, uno
rsn fuente, auxiliar, pc+1
```

i) La macro `@dec fuente` resta uno al operando `fuente`. Trata el valor contenido en el operando como número sin signo y por esa razón ejecuta la siguiente instrucción en la secuencia independientemente del signo del resultado. No actualiza banderas de estado.

```
rsn fuente, uno, pc+1
```

j) La macro `@add sumando-a,sumando-b,auxiliar` calcula la suma de los operandos `sumando-a` y `sumando-b` dejando el resultado en `sumando-a`. Trata los valores contenidos en los operandos como números sin signo y por esa razón ejecuta la siguiente instrucción en la secuencia independientemente del signo del resultado. No actualiza banderas de estado.

```
@neg auxiliar, sumando-b
rsn sumando-a, auxiliar, pc+1
```

k) La macro `@sub minuendo,sustraendo` calcula la diferencia de los operandos `minuendo` y `sustraendo` dejando el resultado en `minuendo`. Trata los valores contenidos en los operandos como números sin signo y por esa razón ejecuta la siguiente instrucción en la secuencia independientemente del signo del resultado. No actualiza banderas de estado.

```
rsn minuendo, sustraendo, pc+1
```

l) La secuencia consiste en hacer tantas sumas del `multiplicando` como indique el `multiplicador`. Dado que sólo podemos restar, realizamos tantas restas como indique el `multiplicador` y luego salvamos el resultado en `producto` restándolo de 0.

```
rsn producto, producto, pc+1
rsn auxiliar, auxiliar, pc+1
#bucle:
rsn multiplicador, uno, #fin
rsn auxiliar, multiplicando, #bucle
#fin:
rsn producto, auxiliar, pc+1
```

8. Contamos con máquinas de diferentes arquitecturas que podemos agrupar, estudiando sus juegos de instrucciones, de la siguiente forma:

	ARQUITECTURAS				
Modo de ejecución	0 direcciones pila	1 dirección acumulador	2 direcciones registro-registro	3 direcciones memoria-memoria	3 direcciones registro-registro

Juego de instrucciones	`push M`
	`pop M`
	`add`
	`sub`
	`mul`
	`div`
		`load M`
		`store M`
		`add M`
		`sub M`
		`mul M`
		`div M`
			`load X, M`
			`store M, X`
			`move X, Y`
			`add X, Y`
			`sub X, Y`
			`mul X, Y`
			`div X, Y`
				`add M1, M2, M3`
				`sub M1, M2, M3`
				`mul M1, M2, M3`
				`div M1, M2, M3`
					`load X, M`
					`store M, X`
					`move X, Y`
					`add X, Y, Z`
					`sub X, Y, Z`
					`mul X, Y, Z`
					`div X, Y, Z`

Los operandos designados con `M` son posiciones de memoria mientras que los designados con `X`, `Y` o `Z` se refieren a registros del procesador.

La máquina de 0 direcciones tiene un modo de ejecución a pila de forma que todas las operaciones se realizan en la cima de la pila. Las operaciones de proceso toman sus operandos de la cima de la pila y los eliminan sustituyéndolos por el resultado. Las operaciones de transferencia (`push` y `pop`) son las únicas que realizan accesos a memoria.

La máquina de 1 dirección tiene un modo de ejecución basado en acumulador. Las operaciones de proceso se realizan con un operando de memoria y otro implícito que es siempre el acumulador. El juego incluye dos instrucciones de transferencia para cargar el registro acumulador (`load`) o para almacenar su contenido en memoria (`store`).

La máquina de 2 direcciones realiza las operaciones de proceso entre dos operandos residentes en alguno de los 16 registros con que cuenta. Las instrucciones de transferencia son las encargadas de mover información entre los registros y memoria (`load`, `store`) o entre los propios registros (`move`).

Las máquinas de 3 direcciones especifican tres operandos por instrucción. Se proponen en la tabla dos casos extremos: uno de ellos trabaja en modo de ejecución memoria-memoria mientras que el otro es registro-registro (también llamado de carga/almacenamiento).

Sabiendo todo esto, escribir los programas correspondientes a cada juego de instrucciones propuesto de forma que realicen el siguiente cálculo:

$$A = \frac{(B + C) \cdot D}{E - F \cdot G - H \cdot I}$$

Partir de la situación en la que todas las variables están en memoria.

SOLUCIÓN:

Máquina mem-mem de 3 direcciones:

```
add B, C, AUX1
mul AUX1, D, AUX1
mul F, G, AUX2
mul H, I, AUX3
sub E, AUX2, AUX2
sub AUX2, AUX3, AUX2
div AUX1, AUX2, A
```

Máquina de
0 direcciones:

```
push B
push C
add
push D
mul
push E
push F
push G
mul
sub
push H
push I
mul
sub
div
pop A
```

Máquina de
1 dirección:

```
load F
mul G
store AUX1
load H
mul I
store AUX2
load E
sub AUX1
sub AUX2
store AUX3
load B
add C
mul D
div AUX3
store A
```

Máquina de
2 direcciones:

```
load R1, B
load R2, C
load R3, D
add R1, R2
mul R1, R3
load R2, E
load R3, F
load R4, G
mul R3, R4
sub R2, R3
load R3, H
load R4, I
mul R3, R4
sub R2, R3
div R1, R2
store A, R1
```

Máquina reg-reg
de 3 direcciones:

```
load R1, B
load R2, C
load R3, D
add R1, R2, R4
mul R4, R3, R1
load R2, F
load R3, I
mul R2, R3, R4
load R2, H
load R3, G
mul R2, R3, R5
load R2, E
sub R2, R4, R3
sub R3, R5, R2
div R1, R2, R3
store A, R3
```

9. Mida la eficiencia de memoria de las diferentes arquitecturas propuestas en el problema anterior. Se harán las siguientes suposiciones sobre los repertorios de instrucciones:

   - los códigos de operación son de 1 byte
   - el direccionamiento de los operandos de memoria ocupa 2 bytes
   - los operandos son de 2 bytes
   - los registros se especifican mediante campos de 4 bits (16 registros)

   Calcule para cada programa dado en la solución del problema anterior el número de bytes que ocupa y el número de bytes que se transfieren a través del bus de datos. Determinar cual es la arquitectura más eficiente si solo se tiene en cuenta el tamaño del código ejecutable. Determinar la eficiencia si se evalúa desde el punto de vista del ancho de banda total de memoria que se necesita, es decir, la suma de bytes de código más datos que se han de mover.

SOLUCIÓN:

Máquina mem-mem de 3 direcciones:

tamaño del código:	
7 códigos de operación	7 bytes
21 direcciones de memoria	42 bytes
total flujo de instrucciones	49 bytes
tamaño datos en memoria:	
14 lecturas	28 bytes
7 escrituras	14 bytes
total flujo de datos	42 bytes
tráfico con memoria:	
código	49 bytes
datos	42 bytes
TOTAL	91 bytes

Máquina de 0 direcciones:

tamaño del código:	
16 códigos de operación	16 bytes
9 direcciones de memoria	18 bytes
total flujo de instrucciones	34 bytes
tamaño datos en memoria:	
8 lecturas	16 bytes
1 escrituras	2 bytes
total flujo de datos	18 bytes
tráfico con memoria:	
código	34 bytes
datos	18 bytes
TOTAL	52 bytes

Máquina de 1 dirección:

tamaño del código:	
15 códigos de operación	15 bytes
15 direcciones de memoria	30 bytes
total flujo de instrucciones	45 bytes
tamaño datos en memoria:	
11 lecturas	22 bytes
4 escrituras	8 bytes
total flujo de datos	30 bytes
tráfico con memoria:	
código	45 bytes
datos	30 bytes
TOTAL	75 bytes

Máquina de 2 direcciones:

tamaño del código:	
16 códigos de operación	16 bytes
9 direcciones de memoria	18 bytes
23 registros	12 bytes
total flujo de instrucciones	46 bytes
tamaño datos en memoria:	
8 lecturas	16 bytes
1 escrituras	2 bytes
total flujo de datos	18 bytes
tráfico con memoria:	
código	46 bytes
datos	18 bytes
TOTAL	64 bytes

Máquina reg-reg de 3 direcciones:

tamaño del código:	
16 códigos de operación	16 bytes
9 direcciones de memoria	18 bytes
30 registros	15 bytes
total flujo de instrucciones	49 bytes
tamaño datos en memoria:	
8 lecturas	16 bytes
1 escrituras	2 bytes
total flujo de datos	18 bytes
tráfico con memoria:	
código	49 bytes
datos	18 bytes
TOTAL	67 bytes

La máquina con el menor tráfico con memoria (flujo de instrucciones y flujo de datos en conjunto) para la tarea computacional propuesta resulta ser la máquina a pila o máquina de 0 direcciones. No es de extrañar, pues muchas calculadoras se basan en una arquitectura a pila dado que el uso de la notación polaca inversa para especificar la operación es connatural al ordenamiento en pila.

10. Los accesos a datos en memoria se pueden implementar mediante direccionamientos directos, relativos a registro o indirectos. Describa cada uno de ellos y discuta sus ventajas e inconvenientes.

SOLUCIÓN:

Los **directos** consisten en incluir en el formato de la instrucción el puntero de memoria al que accede para realizar la operación. Cuanto mayor es el mapa de memoria mayor es el puntero haciendo crecer el tamaño de codificación de la instrucción hasta límites impracticables. El tamaño de instrucción más frecuente en formatos regulares es de 32 bits. Si tomamos como tamaño más frecuente del *opcode* 6 bits, esto nos deja en el caso más favorable 24 bits de puntero lo cual es claramente insuficiente para los mapas de memoria habituales hoy en día.

Una manera de aliviar este inconveniente es utilizar el direccionamiento **directo paginado**. En este caso, no se emite el puntero completo sino un desplazamiento respecto a un puntero previamente fijado. Esto es lo que *Intel* llamó segmentación de memoria. Evidentemente, el desplazamiento puede tener el tamaño que nos convenga pero el problema es que el modo directo paginado no permite ver el mapa de memoria *plano* o en su conjunto y cuando hay que referenciar un puntero fuera de página hay que modificar el puntero base de página.

El modo **relativo a registro** consiste en guardar el puntero en un registro con lo que el formato de instrucción solamente tiene que indicar el registro cuya codificación es notablemente más pequeña que un puntero de memoria. Este modo admite la suma de un desplazamiento sobre el valor del puntero contenido en dicho registro hacíendolo más versatil. Como contrapartida tenemos que el puntero de memoria referenciado (dirección efectiva) no se conoce hasta el instante mismo de la ejecución de la instrucción impidiendo que el compilador pueda realizar ninguna optimización del proceso.

Finalmente, los direccionamientos **indirectos** consisten en referencar un puntero de memoria en el que se encuentra otro puntero que apunta al dato. La primera referencia podría llevarse a cabo mediante cualquiere modo de direccionamiento de memoria con sus ventajas e inconvenientes. Este modo supone una doble referencia a memoria para alcanzar un dato y eso redunda en un coste temporal que no lo hace atractivo por lo que ha terminado desapareciendo en los repertorios de instrucciones modernos.

11. El modo de direccionamiento indexado con autoincremento o autodecremento es similar al relativo a registro índice pero con la particularidad de que incrementa o decrementa el índice automaticamente antes o después de calcular la dirección efectiva de memoria. Discuta las ventajas e inconvenientes que presenta este modo.

SOLUCIÓN:

La secuencia siguiente muestra un caso de direccionamiento relativo a registro índice (`SI`) cuyo valor se incrementa después de acceder a la posición de memoria mediante la ejecución de una instrucción específica (`INC SI`).

```
MOV AX, [SI]
INC SI
```

El modo de direccionamiento indexado con autoincremento sería el equivalente pero condensado en una única instrucción. Algo así como esto:

```
MOV AX, [SI]++
```

o esto si primero se incrementa y luego se accede a la dirección efectiva:

```
MOV AX, ++[SI]
```

El modo con autodecremento sería similar.

La ventaja estriba en que disminuye el recuento de instrucciones además de liberar al programador de ocuparse del cómputo de punteros.

La principal desventaja podría estar en que la instrucción con este modo de direccionamiento debe hacer más tareas que una convencional pero lo cierto es que un incremento (o decremento) es una tarea rápida y sencilla.

El modo de direccionamiento indexado con autoincremento o autodecremento estaba presente en el VAX con la nomemclatura `(m)+` y `-(m)` respectivamente.

12. En muchos casos, el modo de direccionamiento indexado con autoincremento o autodecremento suma o resta un valor diferente en función del objeto referenciado. Explique por qué se hace esto así y hasta qué punto es interesante.

SOLUCIÓN:

El valor usado en las actualizaciones automáticas del índice tiene que ver con el tamaño de los datos declarados en memoria. Si son de tamaño byte, el valor será 1 pero si son de 32 bits, el valor será 4. Es muy interesante que el valor se escale de acuerdo a la declaración de los datos para que el modo de direccionamiento sea eficiente.

Por otra parte, este incremento o decremento escalado sigue siendo muy sencillo y rápido pues el *hardware* del incrementador-decrementador escalado es muy poco más complejo que el unitario.

13. Sabemos que el modo de direccionamiento absoluto a memoria no es útil en máquinas con un mapa de memoria muy grande ya que incluir los punteros en el formato de las instrucciones las hace desmasiado largas. Por ello se creó el direccionamiento paginado a memoria que divide la memoria en páginas de tamaño manejable referenciadas por un puntero *base* y luego se direcciona la memoria mediante un puntero pequeño a la página llamado *desplazamiento*. Discuta las ventajas e inconvenientes de este modo de direccionamiento.

SOLUCIÓN:

La principal ventaja estriba en que alivia considerablemente el tamaño requerido para la codificación de las instrucciones. Sin embargo, no permite "ver" el recurso memoria de manera "plana", es decir, no permite un acceso uniforme a todo el mapa de memoria. Siempre que sea necesario cambiar de página se va a perder un tiempo en referenciar la nueva página. Además, la manera de referenciar un puntero efectivo de memoria no tiene una construcción unívoca sino que diferentes conbinaciones de *base* y *desplazamiento* pueden alcanzar el mismo puntero lo cual contribuye a provocar ciertos efectos laterales: compromete la seguridad, dificulta la optimización del código, etc. No obstante, al final, su eficiencia depende mucho de la huella en memoria que produce cada proceso. Un proceso cuya huella en memoria se circunscriba a una página será un candidato muy adecuado para usar este modo mientras que un proceso cuya huella en memoria sea dispersa o muy cambiante será todo lo contrario.

14. Sea una máquina de acumulador (`Acc`) de la que se quieren diseñar los formatos de representación de sus instrucciones teniendo en cuenta las siguientes características del computador:

- el tamaño de palabra es de 16 bits;
- el modelo de ejecución es registro-registro;
- la memoria principal es de 2 Kpalabras;
- tiene 32 registros, entre los cuales se encuentra el acumulador (`Acc`), el `PC` y el `SP`;
- los modos de direccionamiento que puede tener el procesador son: inmediato, directo a registro y a memoria, relativo a registro base y relativo a `PC`;
- las operaciones que ejecuta son:
  - instrucciones de una dirección:
    - `STA`: almacena el contenido del `Acc` en un registro o en una dirección de memoria.
    - `LDA`: carga en el Acc un inmediato, el contenido de una posición de memoria o el de un registro.
    - `Bcc`: salto condicional (las posibles condiciones son `Z`, `NZ`, `C`, `NC` y el único modo de direccionamiento para esta instrucción es relativo a `PC`).
    - `BR`, `CALL`, `PUSH`, `POP`, `ADD`, `SUB`, `MUL`, `AND`, `OR`, `XOR`, `CMP`
  - instrucciones de cero direcciones:
    - `RET`, `INC`, `DEC`, `NEG`, `NOT`, `SHL`, `SHR`, `ROL`, `ROR`, `HALT`, `WAIT`

Sabiendo todo esto, diseñe el/los formatos de representación de instrucciones, indicando los modos de direccionamiento que puede admitir cada instrucción (o grupo de instrucciones), y minimizando el espacio de representación.

SOLUCIÓN:

Minimizar el espacio de representación significa que hemos de diseñar el formato del repertorio de instrucciones procurando que la imagen del ejecutable sea lo menor posible. Para alcanzar este objetivo tenemos varias opciones. Una podría ser asignar a las operaciones más frecuentes la codificación más pequeña posible. Otra manera de abordar el problema podría ser asignar códigos de operación pequeños a operaciones que requieren modos de direccionamiento que necesitan muchos bits para codificarse (punteros, por ejemplo). Ambas aproximaciones se pueden combinar y siempre van a estar basadas en la técnica de los campos de extensión.

Por lo general, el criterio de diseño orientado a la minimización del espacio de representación suele contraponerse al criterio orientado a la codificación regular.

Es importante hacer notar que, como cualquier problema de diseño, existen multiples soluciones.

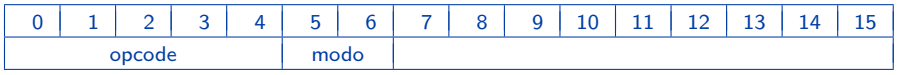
Antes de comenzar vamos a cuantificar el número total de operaciones, modos de direccionamiento y la cantidad de bits que se requieren para codificar una variedad de datos:

1. número total de operaciones: 25 operaciones
2. bits necesarios para codificar las operaciones: 5 bits
3. número de modos de direccionamiento: 4 modos asumiendo que relativo a registro y relativo a `PC` son el mismo y son excluyentes
4. bits necesarios para codificar los modos: 2 bits en caso de que una operación pudiera usar todos los modos
5. bits necesarios para codificar los registros: 5 bits
6. tamaño de un puntero de memoria: 11 bits

Con estos datos podemos intentar construir un formato regular sabiendo que no es el criterio pedido pero es un ejercicio que nos puede dar pistas de cómo llevar a cabo la tarea.

Sabemos que la codificación de un repertorio de instrucciones se ha de hacer sobre un formato de tamaño multiplo de byte. Tomaremos como tamaño regular 2 bytes, es decir, 16 bits.

Como hemos visto, necesitaremos un código de operación u opcode de 5 bits y un campo de modo de direccionamiento de 2 bits. El resto (9 bits) sería la dirección.



Pues bien, una dirección de 9 bits no soporta punteros de 11 bits con lo que no permite codificar completamente todas las posibilidades de la instrucción `STA`, por ejemplo. Sería posible si aumentáramos el tamaño del formato a 3 bytes pero entonces es evidente que no vamos a minimizar el tamaño de representación.

Nos damos cuenta de que el nudo del problema está en la codificación de todos los modos de direccionamiento que utiliza cada instrucción. Por eso, vamos a hacer una tabla en la que relacionamos cada instrucción con sus modos empezando por las más exigentes (más modos). Tendremos:

instrucciones de 1 dirección:

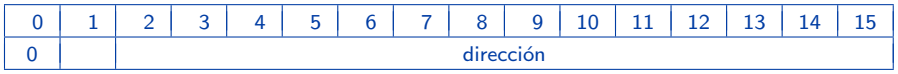
instrucción	**modos de direccionamiento**
`LDA`	inmediato
	directo a registro
	directo a memoria
	relativo a registro
`STA`	directo a registro
	directo a memoria
	relativo a registro
`Bcc`	relativo a registro (`PC`)
`BR`, `CALL`	directo a memoria
	relativo a registro
`ADD`, `SUB`, `MUL`	inmediato
`AND`, `OR`, `XOR`	directo a registro
`CMP`, `PUSH`	
`POP`	directo a registro

instrucciones de 0 direcciones:

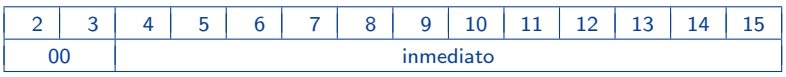
`NEG`, `INC`, `DEC`, `NOT`, `SHL`, `SHR`, `ROL`, `ROR`, `HALT`, `WAIT`

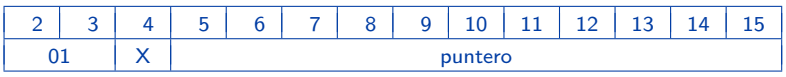
Teniendo en cuenta esta clasificación proponemos los siguientes formatos:

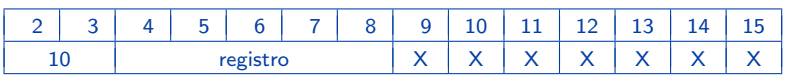
Formato I: para las instrucciones `LDA` y `SDA`. Campo de extensión `0`.

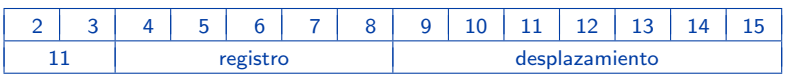


El bit `0` es el campo de extensión (`0`) y el bit `1` es el opcode (`0` para `STA` y `1` para `LDA`). El resto es el campo dirección sobre el que se deben codificar 4 posibles modos de direccionamiento según los siguientes formatos:



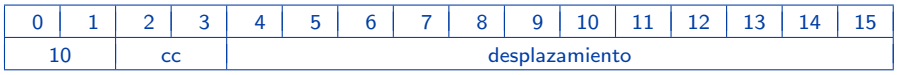






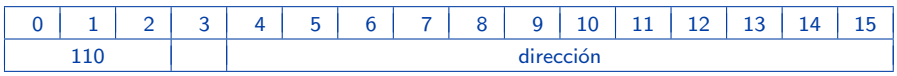
Como vemos, los bits 2 y 3 codifican cada uno de los cuatro modos posibles (la instrucción `STA` no admite el inmediato).

Formato II: para la instrucción `Bcc`. Campo de extensión 10.

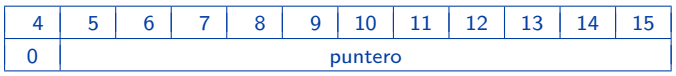


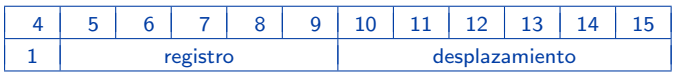
En este formato solamente hay una instrucción y, por tanto, no necesita opcode. Los cuatro códigos de condición se codifican sobre los bits 2 y 3. El desplazamiento representa un número entero en complemento a dos que se suma al `PC`. Como tiene 12 bits permite alcanzar cualquier posición del mapa de memoria completo.

Formato III: para las instrucciones `BR` y `CALL`. Campo de extensión 110.



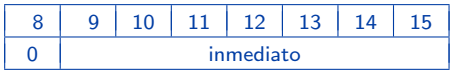
Como tenemos dos instrucciones diferentes en este formato necesitamos un opcode de un bit. Los restantes 12 bits se reservan para la dirección cuyo formato se muestra seguidamente para codificar el modo directo a memoria y el relativo a registro más desplazamiento.



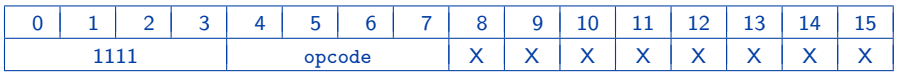


Formato IV: para el resto de instrucciones de 1 dirección. Campo de extensión 1110.

Codificamos las 9 instrucciones restantes sobre un opcode de 4 bits y nos queda un campo dirección de 8 bits sobre el que hemos de codificar dos posibles modos de direccionamiento (en el caso de la instrucción `POP` solamente directo a registro).



Formato V: para las instrucciones de 0 direcciones. Campo de extensión 1111.



Vemos que en este caso se desperdician 8 bits que no contienen codificación alguna.

El formato propuesto para el repertorio de instrucciones utiliza un tamaño fijo de 2 bytes para todas las instrucciones consiguiendo representar modos de direccionamiento de gran tamaño junto con las operaciones que los usan sin aumentar el tamaño de codificación.

No es un formato regular ya que, aunque el tamaño de codificación es común a todas las instrucciones, la ubicación de los campos opcode y dirección cambia de unas instrucciones a otras.

En algunos casos quedan bits sin utilizar lo que nos hace pensar que muy probablemente podríamos conseguir un formato más compacto.