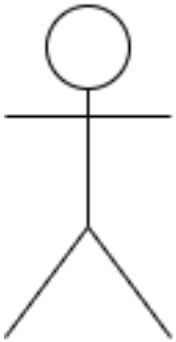
Programación en ASM

Programando con la Arquitectura ORGA1

Organización del Computador I
DC - UBA

1er Cuatrimestre 2022

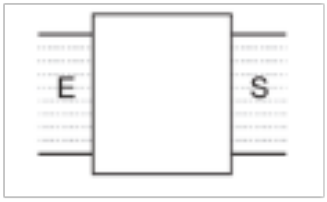
¿Dónde estamos?



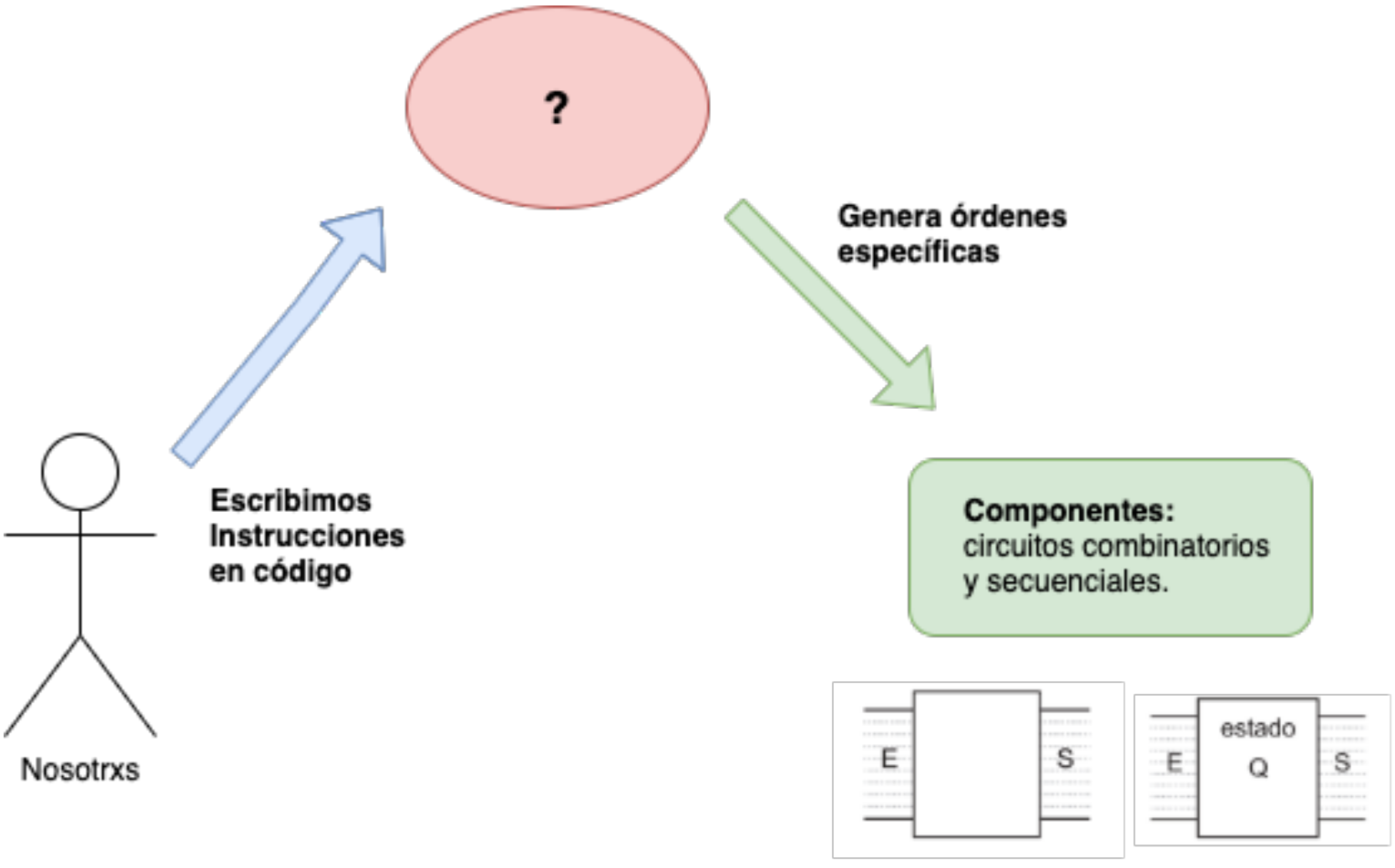
Nosotrxs

Ordenes manuales

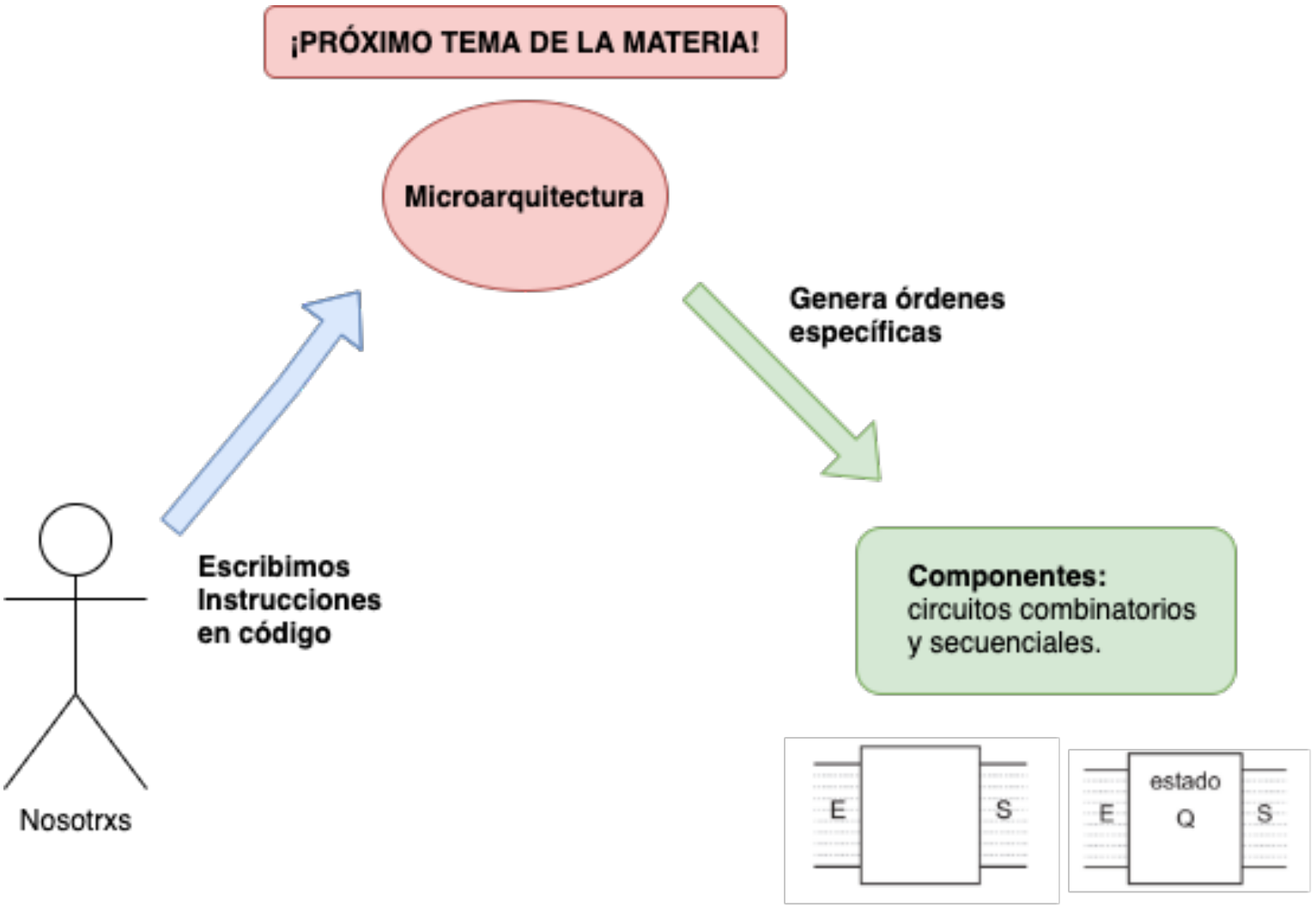
Componentes:
circuitos combinatorios
y secuenciales.



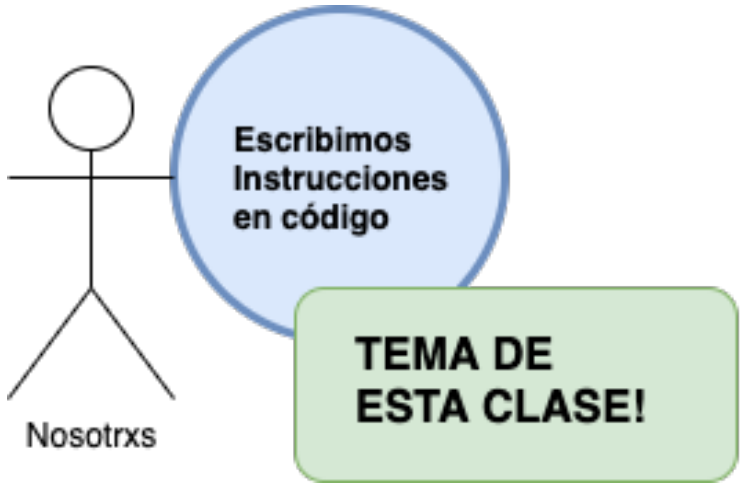
¿Para dónde vamos?



¿Para dónde vamos?



Menú de hoy!

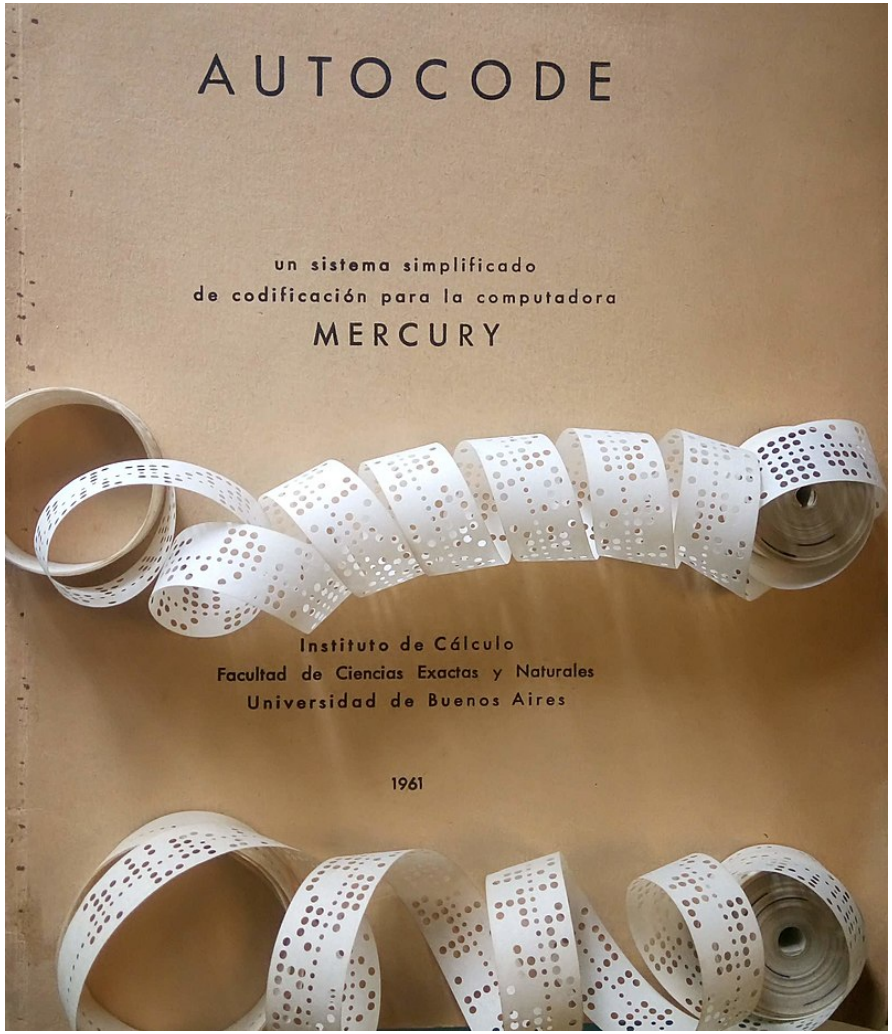

Escribimos
Instrucciones
en código
TEMA DE
ESTA CLASE!
Nosotrxs

Motivación

Código de la computadora del Apollo XI

https://github.com/chrislgarry/Apollo-11/

Motivación



Descripción del Lenguaje AUTOCODE

https://bit.ly/3u2ue9f

Algunas aclaraciones

- Se utilizará el lenguaje Assembler de la máquina ’Orga 1’.

Algunas aclaraciones

- Se utilizará el lenguaje Assembler de la máquina ’Orga 1’.
- Este es un lenguaje ensamblador particular de esta arquitectura, que es una simplificación de una arquitectura Intel x86.

Algunas aclaraciones

- Se utilizará el lenguaje Assembler de la máquina ’Orga 1’.
- Este es un lenguaje ensamblador particular de esta arquitectura, que es una simplificación de una arquitectura Intel x86.
- Posee instrucciones con 2 operandos, 1 operando, 0 operandos y saltos.

Algunas aclaraciones

- Se utilizará el lenguaje Assembler de la máquina ’Orga 1’.
- Este es un lenguaje ensamblador particular de esta arquitectura, que es una simplificación de una arquitectura Intel x86.
- Posee instrucciones con 2 operandos, 1 operando, 0 operandos y saltos.
- Además posee diferentes modos de direccionamiento.

Arquitectura ORGA1

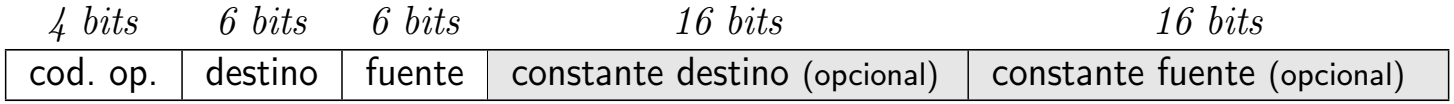
Descripción General

- Palabras de *16 bits*.
- Direccionamiento a palabra.
- Espacio direccionable de *65536* palabras.
- Espacio de direcciones dedicado a entrada/salida en las direcciones 0xFFF0 - 0xFFFF.
- Ocho registros de propósito general de *16 bits*: R0..R7.
- *Program counter* (PC) de *16 bits*.
- *Stack pointer* (SP) de *16 bits* inicializado en la dirección 0xFFEF.
- Los valores de los *flags* se calculan interpretando los operandos en complemento a 2. *Flags*: Z (*zero*), N (*negative*), C (*carry*), V (*overflow*).
- Todas las instrucciones alteran los *flags*, excepto MOV, CALL, RET, JMP y Jxx.
- De las que alteran los *flags*, todas dejan C y V en cero, excepto ADD, ADDC, SUB, CMP y NEG.

Arquitectura ORGA1

Formato de instrucción

Tipo 1: Instrucciones de dos operandos



operación	cod. op.	efecto
MOV d, f	0001	$d \leftarrow f$
ADD d, f	0010	$d \leftarrow d + f$ (suma binaria)
SUB d, f	0011	$d \leftarrow d - f$ (resta binaria)
AND d, f	0100	$d \leftarrow d$ and f
OR d, f	0101	$d \leftarrow d$ or f
CMP d, f	0110	Modifica los *flags* según el resultado de $d - f$ (resta binaria)
ADDC d, f	1101	$d \leftarrow d + f +$ *carry* (suma binaria)

Formato de operandos destino y fuente.

Modo	Codificación	Resultado
Inmediato	000000	c16
Directo	001000	[c16]
Indirecto	011000	[[c16]]
Registro	100rrr	Rrrr
Indirecto registro	110rrr	[Rrrr]
Indexado	111rrr	[Rrrr + c16]

c16 es una constante de *16 bits*.

Rrrr es el registro indicado por los últimos tres *bits* del código de operando.

Las instrucciones que tienen como destino un operando de tipo *inmediato* son consideradas como inválidas por el procesador, excepto el CMP.

***Tipo 2*: Instrucciones de un operando**

Tipo 2a: Instrucciones de un operando destino.

4 bits	*6 bits*	*6 bits*	*16 bits*
cod. op.	destino	000000	constante destino (opcional)

operación	cod. op.	efecto
NEG d	1000	$d \leftarrow 0 - d$ (resta binaria)
NOT d	1001	$d \leftarrow$ not d (bit a bit)

El formato del operando *destino* responde a la tabla de formatos de operando mostrada más arriba.

Tipo 2b: Instrucciones de un operando fuente.

4 bits	*6 bits*	*6 bits*	*16 bits*
cod. op.	000000	fuente	constante fuente (opcional)

operación	cod. op.	efecto
JMP f	1010	PC $\leftarrow f$
CALL f	1011	[SP] $\leftarrow$ PC, SP $\leftarrow$ SP $-$ 1, PC $\leftarrow f$

El formato del operando *fuente* responde a la tabla de formatos de operando mostrada más arriba.

Tipo 3: Instrucciones sin operandos

4 bits	*6 bits*	*6 bits*
cod. op.	000000	000000

operación	cod. op.	efecto
RET	1100	PC ← [SP+1], SP ← SP + 1

Tipo 4: Saltos condicionales

Las instrucciones en este formato son de la forma *Jxx* (salto relativo condicional). Si al evaluar la condición de salto en los *flags* el resultado es *1*, el efecto es incrementar el PC con el valor de los *8 bits* de desplazamiento, representado en *complemento a 2* de *8 bits*. En caso contrario, la instrucción no produce efectos.

8 bits	*8 bits*
cod. op.	desplazamiento

Codop	Operación	Descripción	Condición de Salto
1111 0001	JE	Igual / Cero	Z
1111 1001	JNE	Distinto	not Z
1111 0010	JLE	Menor o igual	Z or (N xor V)
1111 1010	JG	Mayor	not (Z or (N xor V))
1111 0011	JL	Menor	N xor V
1111 1011	JGE	Mayor o igual	not (N xor V)
1111 0100	JLEU	Menor o igual sin signo	C or Z
1111 1100	JGU	Mayor sin signo	not (C or Z)
1111 0101	JCS	Carry / Menor sin signo	C
1111 0110	JNEG	Negativo	N
1111 0111	JVS	Overflow	V

¿Cómo alterar los flags?

Sea r el resultado de una instrucción que modifica los flags, el nuevo valor es el que sigue:

- `Z=1` $\leftrightarrow$ $r = 0x0000$.
- `N=1` $\leftrightarrow$ el bit más significativo de r es igual a 1.
- `C=1` $\leftrightarrow$ se produjo *carry* durante una suma binaria o *borrow* durante una resta binaria.
- `V=1` $\leftrightarrow$ la suma de dos números con signo produce un número *sin* signo ($S + S = \overline{S}$) ó la suma de dos números *sin* signo produce un número con signo ($\overline{S} + \overline{S} = S$) ó alguna analogía con la resta ($S - \overline{S} = \overline{S}$ ó $\overline{S} - S = S$)

El ensamblador

Directivas

El ensamblador de código tiene una única directiva.

Directiva	Efecto
DW c16	Asigna en la posición correspondiente la constante c16

Ejemplos de direccionamiento

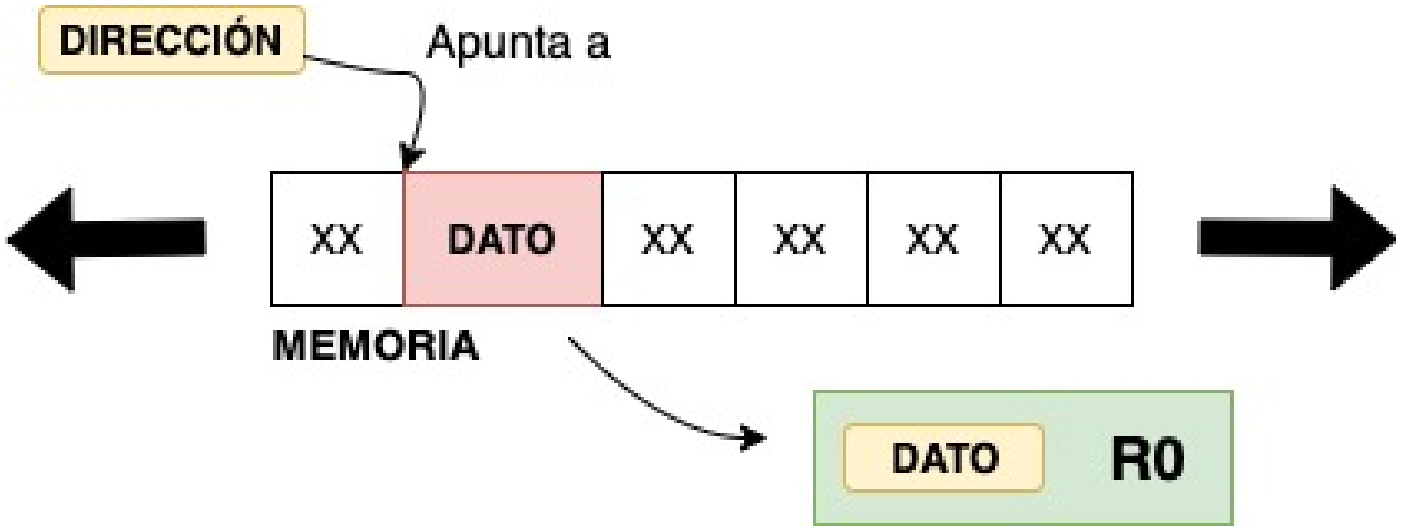
Direccionamiento directo

MOV R0, [DIRECCION]

Ejemplos de direccionamiento

Direccionamiento directo

MOV R0, [DIRECCION]



Ejemplos de direccionamiento

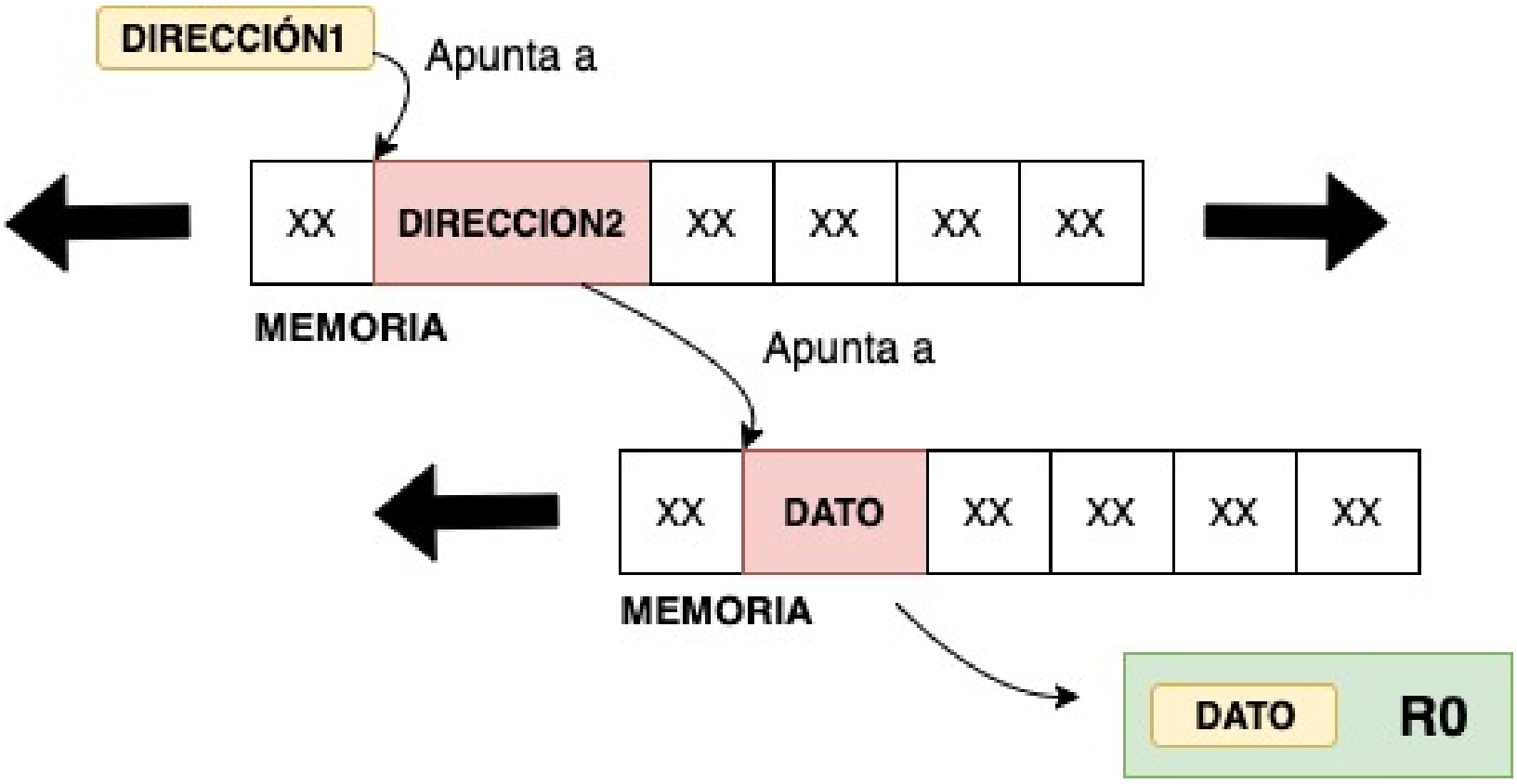
Direccionamiento indirecto

MOV R0, [[DIRECCION1]]

Ejemplos de direccionamiento

Direccionamiento indirecto

MOV R0, [[DIRECCION1]]



Ejemplos de direccionamiento

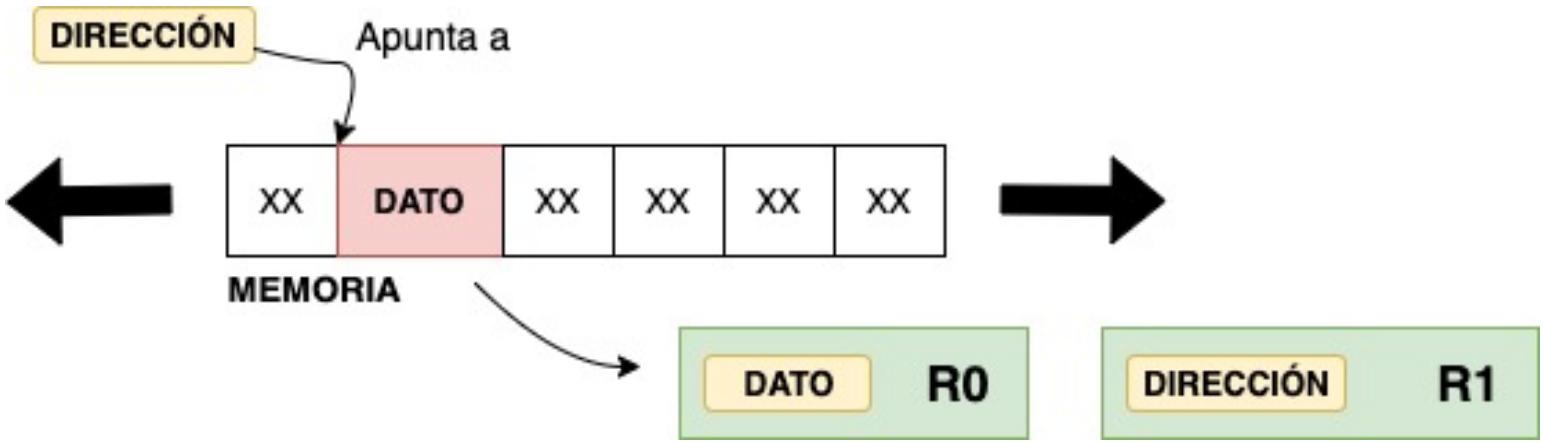
Direccionamiento indirecto con registro

MOV R0, [R1]

Ejemplos de direccionamiento

Direccionamiento indirecto con registro

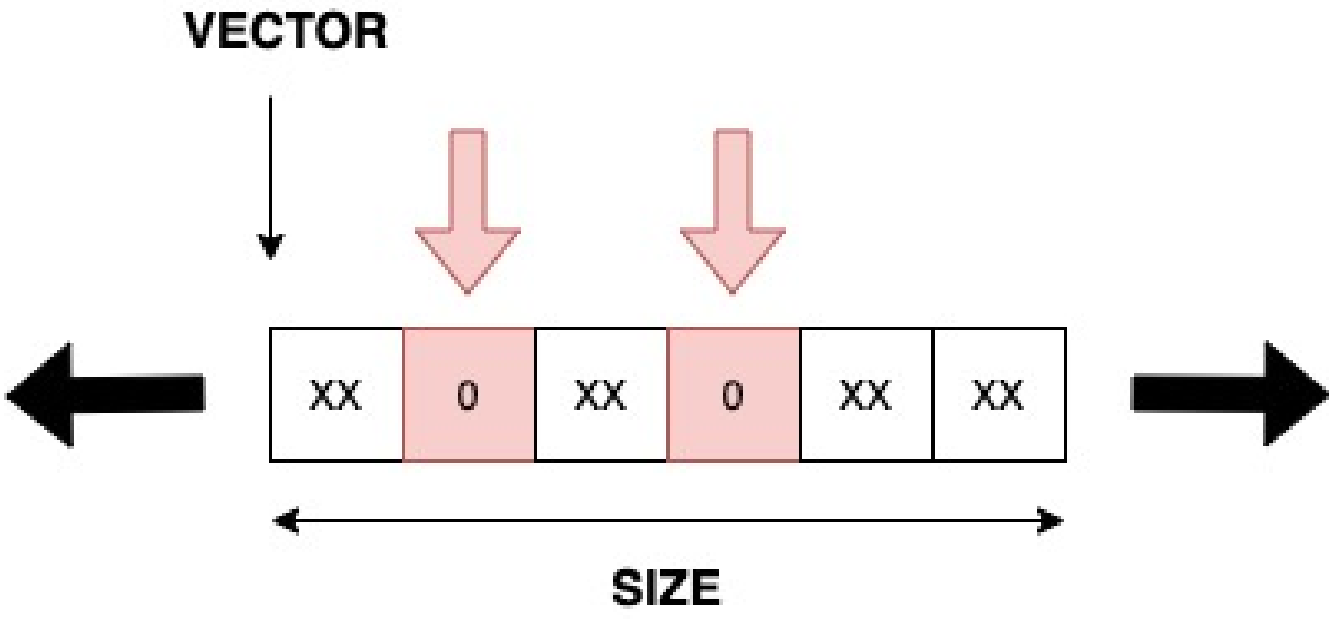
MOV R0, [R1]



Ejercicio 1

Podemos pensar la memoria como una *tira* de bits. Dentro de ella tenemos almacenado un **vector de enteros de 16 bits** guardado en posiciones de memoria consecutivas a partir de la dirección identificada por la etiqueta **VECTOR**. Queremos hacer una rutina que calcule la cantidad de *ceros* que hay almacenados en el vector. El tamaño del vector se encuentra en la posición de memoria identificada por la etiqueta **SIZE**. Buscamos retornar el resultado (la cantidad de *ceros* que hay en el vector) en la posición de memoria identificada por la etiqueta **CEROS**.

Analizando el enunciado



Armando el pseudocódigo

Armando el pseudocódigo

Inicializar Registros
para cada elemento del vector
    si vector(i) == 0 **entonces**
        incrementar el contador de ceros
    fin si
fin para
Colocar el resultado en la etiqueta **CEROS**

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
        JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
        JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
        ADD R1, 0x0001 ; Encontré un cero.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
        JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
        ADD R1, 0x0001 ; Encontré un cero.
seguir: ADD R0, 0x0001 ; Avanzo una posición del vector.
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
        JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
        ADD R1, 0x0001 ; Encontré un cero.
seguir: ADD R0, 0x0001 ; Avanzo una posición del vector.
        SUB R2, 0X0001 ; Decremento tamaño del vector (un elemento
                       menos por recorrer).
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
         JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
         ADD R1, 0x0001 ; Encontré un cero.
seguir:  ADD R0, 0x0001 ; Avanzo una posición del vector.
         SUB R2, 0X0001 ; Decremento tamaño del vector (un elemento
                        menos por recorrer).
         JNE ciclo ; Si quedan elementos por recorrer retoma el ciclo
```

Assembler!

```
; Vamos a usar los siguientes registros para guardar:
; R0 –> Dirección del comienzo del vector.
; R1 –> Cantidad de ceros que voy encontrando en el vector.
; R2 –> Tamaño del vector a recorrer.
  main: MOV R0, VECTOR ; R0 = dirección donde comienza el vector
        MOV R1, 0x0000 ; R1 inicializado en 0
        MOV R2, [SIZE] ; R2 = tamaño del vector.
; Chequeo si el elemento del vector (apuntado por R0) es 0.
  ciclo: CMP [R0], 0x0000
         JNE seguir ; Salto a la etiqueta seguir si el elemento no era cero.
         ADD R1, 0x0001 ; Encontré un cero.
seguir:  ADD R0, 0x0001 ; Avanzo una posición del vector.
         SUB R2, 0X0001 ; Decremento tamaño del vector (un elemento
                        menos por recorrer).
         JNE ciclo ; Si quedan elementos por recorrer retoma el ciclo
         MOV [CEROS], R1 ; Terminé de recorrer el vector y almaceno el
                         resultado
```

Enunciado Ejercicio Nro 2

Queremos implementar en Assembler una función que nos devuelva el valor del doceavo (12) término de la sucesión de Fibonacci.

Sucesión de Fibonacci

$$f_0 = 1$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Pseudocódigo

```
terminosXCalcular = 10; //(12-2) me da el término 12 de la suc.
anteUltTermino = 1;
ultTermino = 1;
mientras terminosXCalcular > 0
    valor = ultTermino;
    valor = valor + anteUltTermino;
    anteUltTermino = ultTermino;
    ultTermino = valor;
    terminosXCalcular = terminosXCalcular - 1;
fin mientras
```

Assembler!

```
; R0 –> terminosXCalcular
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
      MOV R1, R2 ; anteUltTermino = ultTermino;
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
      MOV R1, R2 ; anteUltTermino = ultTermino;
      MOV R2, R3 ; ultTermino = valor;
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
      MOV R1, R2 ; anteUltTermino = ultTermino;
      MOV R2, R3 ; ultTermino = valor;
      SUB R0, 0x0001 ; terminosXCalcular = terminosXCalcular - 1;
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
      MOV R1, R2 ; anteUltTermino = ultTermino;
      MOV R2, R3 ; ultTermino = valor;
      SUB R0, 0x0001 ; terminosXCalcular = terminosXCalcular - 1;
      JG ciclo
```

Assembler!

```
; R0 –> terminosXCalcular
; R1 –> anteUltTermino
; R2 –> ultTermino
; R3 –> valor
main: MOV R0, 0x000A ; terminosXCalcular = 10
      MOV R1, 0x0001 ; anteUltTermino = 1
      MOV R2, 0x0001 ; ultTermino = 1
ciclo: MOV R3, R2 ; valor = ultTermino;
      ADD R3, R1 ; valor += anteUltTermino;
      MOV R1, R2 ; anteUltTermino = ultTermino;
      MOV R2, R3 ; ultTermino = valor;
      SUB R0, 0x0001 ; terminosXCalcular = terminosXCalcular - 1;
      JG ciclo
  fin:
```

Queda de tarea ..

Queremos escribir un programa que calcule la división entera entre dos enteros sin signo de 16 bits en ASM.

- R1 contiene la dirección de memoria donde se aloja el dividendo.
- R2 contiene la dirección de memoria donde se aloja el divisor.
- R3 debe ser el registro en el que se devuelva el cociente (el resultado de la división).

Para que no explote nada, si el divisor es 0, vamos a retornar un 0 como resultado directamente.

Sugerencia: Pueden usar el pseudocódigo de la siguiente slide, aunque también es una buena práctica elaborar nuestros propios pseudocódigos.

Pseudocódigo posible del Ej. de tarea

```
resultado = 0
si divisor == 0 entonces
    listo
sino
    mientras dividendo >= divisor hacer
        dividendo = dividendo - divisor
        resultado = resultado + 1
    fin mientras
fin si
```

Conclusiones

- Vimos como realizar pequeños programas en Assembler utilizando la Arquitectura ’Orga 1’

Conclusiones

- Vimos como realizar pequeños programas en Assembler utilizando la Arquitectura ’Orga 1’
- Utilizar pseudocódigo es una buena herramienta para aproximarnos al problema antes de *codear* en ASM.

Conclusiones

- Vimos como realizar pequeños programas en Assembler utilizando la Arquitectura ’Orga 1’
- Utilizar pseudocódigo es una buena herramienta para aproximarnos al problema antes de *codear* en ASM.
- Pueden utilizar un pseudocódigo informal, que pueda comprenderse y no sea ambiguo.

Conclusiones

- Vimos como realizar pequeños programas en Assembler utilizando la Arquitectura ’Orga 1’
- Utilizar pseudocódigo es una buena herramienta para aproximarnos al problema antes de *codear* en ASM.
- Pueden utilizar un pseudocódigo informal, que pueda comprenderse y no sea ambiguo.
- Una vez verificada la solución, ya podemos hacer el código en el lenguaje ensamblador de la materia.