**ARQUITECTURA DE COMPUTADORAS (E1213)**
**CIRCUITOS DIGITALES Y MICROPROCESADORES (E0213)**
**Curso 2021**

# GUIA PARA EL MÓDULO 1

## Objetivo del documento

Este es un recorrido guiado por el repertorio de instrucciones de la **arquitectura didáctica MARIE**. Se utilizan ejemplos y ejercicios simples que no requieren mucho tiempo para su resolución. Algunos ejercicios están resueltos en esta guía y otros fueron presentados en las clases teórico-prácticas. Se utilizan para resaltar los aspectos más importantes de un repertorio de instrucciones, los cuales servirán como base del diseño de los procesadores que serán presentados en los próximos módulos.

Se recomienda al alumno analizar los ejercicios propuestos y reflexionar sobre las preguntas que se dejan abiertas. Se recomiendan la discusión en grupo, las consultas con los docentes y la comparación de resultados. Sin embargo se espera que cada alumno arribe a sus propias conclusiones. Este tipo de problemática se utilizará para evaluar al final del semestre.

Se recomienda la lectura del anexo **Resumen de la Arquitectura MARIE**, donde se presentan las características principales del repertorio de instrucciones y la organización. La arquitectura fue presentada en detalle en las clases teóricas, utilizando como base los capítulos 4 y 5 del texto “The Essentials of Computer Organization and Architecture” de Linda Null y Julia Lobur (ver en Google Books).

## El repertorio de instrucciones de MARIE

El repertorio de instrucciones de MARIE es muy reducido, sobre todo si se lo compara con procesadores comerciales que disponen de cientos de instrucciones. MARIE tiene un único registro de usuario (acumulador) y sólo 16 instrucciones, las cuales se encuentran detalladas en el apunte Resumen de la Arquitectura MARIE, de este módulo. El ensamblador/simulador es bastante limitado y el depurado de programas resulta trabajoso. Por este motivo esta práctica se limita a plantear sólo problemas simples.

El objetivo del módulo es que el alumno comprenda el funcionamiento de un microprocesador y reflexione acerca de las limitaciones que se presentan en el momento de programar una arquitectura con un único registro y limitados modos de direccionamiento.

En el siguiente módulo se utilizará un procesador con un repertorio de instrucciones más potente, lo que permitirá explorar con comodidad algoritmos más complicados.

Por ahora analicemos lo que nos ofrece MARIE.

**a) Estructura del programa**

Recordemos la definición de programación estructurada.

> (Wikipedia) "La *programación estructurada* es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a *subrutinas* y tres estructuras básicas: *secuencia*, *selección* (if y switch) e *iteración* (bucles for y while); asimismo, se considera innecesario y contraproducente el uso de la instrucción de transferencia incondicional (GOTO), que podría conducir a código mucho más difícil de seguir y de mantener, y fuente de numerosos errores de programación."

i) Un programa escrito con el repertorio de instrucciones de MARIE no es necesariamente *estructurado*. ¿Por qué? ¿Tiene salto incondicional? ¿Tiene repeticiones y bifurcaciones?

Ahora bien, si pudiéramos implementar las estructuras básicas (secuencia, selección, iteración y subrutina) podríamos traducir un programa escrito en un lenguaje de programación estructurado (por ejemplo C) al código de MARIE. En ese caso serían implementables todos los algoritmos que tengan implementación en C (o sea, todos?).

Veamos si pueden armarse las estructuras básicas.

La *secuencia* es natural. El PC se incrementa de a uno y recorre la memoria de programa en orden.

ii) ¿Cuáles son las instrucciones de MARIE que pueden alterar la secuencia?

iii) ¿Cómo puede implementarse una *iteración* infinita? ¿Y una iteración de un número finito de veces? Por ejemplo en C:

```
for(i=0;i<10;i++)
    x=x+1;
```

iv) ¿Cómo puede implementarse una *selección* en el código? Por ejemplo en C:

```
if(x==y)
    x=x*2;
else
    x=x-1;
```

v) Finalmente, ¿qué instrucciones de MARIE son útiles para la implementación de *subrutinas*? Por ejemplo,

```
int duplicar(int a){
    return a+a;
}
```

Las resoluciones de estos problemas fueron planteadas en la teoría y también están en el libro. Se recomienda simularlos para familiarizarse con el funcionamiento del procesador y de las herramientas disponibles.

```
      / Example 4.1
```

```
        ORG 100
        Load    Addr    / Load address of first number to be added
        Store   Next    / Store this address is our Next pointer
        Load    Num     / Load the number of items to be added
        Subt    One     / Decrement
        Store   Ctr     / Store this value in Ctr to control looping

Loop,   Load    Sum     / Load the Sum into AC
        AddI    Next    / Add the value pointed to by location Next
        Store   Sum     / Store this sum
        Load    Next    / Load Next
        Add     One     / Increment by one to point to next address
        Store   Next    / Store in our pointer Next
        Load    Ctr     / Load the loop control variable
        Subt    One     / Subtract one from the loop control variable

        Store   Ctr     / Store this new value in loop control variable

        Skipcond 000    / If control variable < 0, skip next instruction
        Jump    Loop    / Otherwise, go to Loop
        Halt            / Terminate program
Addr,   Hex     117     / Numbers to be summed start at location 118

Next,   Hex     0       / A pointer to the next number to add
Num,    Dec     5       / The number of values to add
Sum,    Dec     0       / The sum
Ctr,    Hex     0       / The loop control variable
One,    Dec     1       / Used to increment and decrement by 1
        Dec     10      / The values to be added together
        Dec     15
        Dec     20
        Dec     25
        Dec     30
```

**/ Example 4.2**

```
        ORG 100
If,     Load    X       / Load the first value
        Subt    Y       / Subtract the value of Y, store result in AC
        Skipcond 0x400      / If AC=0, skip the next instruction
        Jump    Else    / Jump to Else part if AC is not equal to 0
Then,   Load    X       / Reload X so it can be doubled
        Add     X       / Double X
        Store X         / Store the new value
        Jump    Endif   / Skip over the false part (else)
Else,   Load    Y       / Start the else part by loading Y
        Subt    X       / Subtract X from Y
        Store Y         / Store Y-X in Y
Endif,  Halt            / Terminate program
X,      Dec     12
Y,      Dec     20
        END
```

```
/ Example 4.3
      ORG    100
      Load   X         / Load the first number to be doubled.
      Store  Temp     / Use Temp as parameter to pass value to Subr.
      JnS    Subr      / Store return address, jump to the procedure.
      Store  X         / Store the first number, doubled
      Load   Y         / Load the second number to be doubled.
      Store  Temp
      JnS    Subr      / Store return address, jump to the procedure.
      Store Y           / Store the second number doubled.
      Halt              / End program.
X,    DEC    20
Y,    DEC    48
Temp, DEC    0
Subr, HEX    0       / Store return address here.
      Load   Temp     / Actual subroutine to double numbers.
      Add    Temp    / AC now holds double the value of Temp.
      JumpI  Subr    / Return to calling code.
      END
```

Analice la secuencia de instrucciones con que se implementa en los ejemplos un llamado a subrutina, utilizando la instrucción JnS para hacer el llamado y JumpI para retornar. ¿Cuáles son las limitaciones que presenta este método? ¿Qué método utilizan los procesadores comerciales más simples?

En Marie, ¿el programa principal podría utilizar el acumulador para pasarle a la subrutina un argumento? ¿Puede la subrutina devolver el resultado en el acumulador?

¿Cómo sería la implementación por software de una pila? Cómo puede utilizarse para trabajar con funciones? Los procesadores comerciales ¿disponen de alguna herramienta adicional para trabajar con este tipo de estructura de datos?

**b) Tipos de datos y tipos de operaciones aritmético/lógicas**

MARIE sólo dispone de operaciones aritméticas de suma y resta en formato Ca2 de 16 bits.

i) Experimente con las operaciones de suma y resta. Familiarícese con el rango de números representables [-32768,32767]. ¿Qué sucede si se almacena el número decimal positivo 40000 en una posición de memoria (Y, DEC 40000)? ¿Cómo se puede cambiar de signo un número? ¿Cómo se verifica el desborde (overflow)?

ii)¿Es posible realizar sumas de enteros en 32 bits?

iii) Analice el ejemplo de multiplicación iterativa entre enteros de 16 bits que se ofrece como ejemplo en el simulador. Observe que su duración es variable. Además la duración no es la misma al hacer x*y que al hacer y*x.

```
/ Multiplication Calculator
/ by the MARIE.js Team
```

```
/ Copyright (C) 2016. Licensed under the MIT License
/ Prompt user to type in integers
INPUT
Store X
INPUT
Store Y
/ Loop for performing iterative addition
loop,    Load num
         Add X
         Store num
         Load Y
         Subt one
         Store Y
         Skipcond 400 / have we completed the multiplication?
         Jump loop / no; repeat loop
         / yes, so exit the loop
/ Output result to user then halt program
Load num
Output
Halt
X, DEC 0
Y, DEC 0
num, DEC 0
one, DEC 1
```

Suponga que desea multiplicar un número grande (por ejemplo 1000) por 2. Calcule la cantidad de instrucciones que se ejecutan si:

- Utiliza la rutina con X=2 e Y=1000
- Idem con X=1000 e Y=2
- Si en lugar de multiplicar X por 2, hago X+X
- Si en lugar de multiplicar X por 2, desplazo X a la izquierda una posición

iv) Reflexione sobre las operaciones en punto flotante. ¿Cuál sería la secuencia de instrucciones MARIE que permita sumar dos números representados en formato IEEE 32 bits? No hace falta escribir un programa, la idea es que trate de dimensionar el problema.

v) ¿De qué facilidades se dispone para el procesamiento de texto como números (codificación ASCII). ¿Cómo convertiría números binarios a ASCII?

vi) MARIE no tiene las operaciones lógicas de que disponen los procesadores comerciales. ¿Hay lugar en el ISA para agregarlas? ¿Pueden reemplazarse con operaciones aritméticas?

**c) Modos de direccionamiento**

Los modos de direccionamiento disponibles en MARIE son *directo* e indirecto a través de la memoria.

i) ¿Cómo describiría en RTN las instrucciones Add y AddI?

ii) ¿Es posible utilizar eficientemente punteros en MARIE?

iii) Siendo que no se dispone de direccionamiento inmediato, ¿cómo se puede hacer para incorporar constantes a un programa? ¿Cómo se codificaría y e implementaría en RTN una nueva instrucción de suma que admita direccionamiento inmediato, por ejemplo AddM? ¿Y si quisiera incorporar también el modo de direccionamiento indexado (indirecto via registro con offset)?

iv) Si pudiera elegir, ¿cuál sería el modo de direccionamiento más eficiente para procesar un vector en forma secuencial? ¿Cómo luciría el supuesto código, suponiendo que se desea operar en orden sobre todos los elementos del vector?

Estos ejercicios fueron resueltos en la teoría.

**d) Registros**

MARIE dispone de un único registro de usuario. Esto no es necesariamente una desventaja, ya que simplifica la codificación de las instrucciones. Todas las instrucciones utilizan dicho registro, porque es el único, entonces no hace falta nombrarlo en el código de la instrucción (esto suele llamarse modo de direccionamiento implícito). En consecuencia las instrucciones son cortas, ya que no requieren de un campo para nombrar los registros que se utilizan.

La desventaja es que se genera un tráfico importante de datos con la memoria, ya que, al no haber registros, la memoria es el único lugar donde se puede almacenar un resultado intermedio.

Por ejemplo, considere la secuencia de instrucciones que realiza el siguiente cálculo:

$$f = (g + h) - (i + j)$$

```
Load i
Add j
Store temp    / ***
Load g
Add h
Subt temp
Store f
```

El programa realiza 3 operaciones de la ALU y 7 accesos a memoria. El resultado intermedio (resultado de la suma i + j) debe ser almacenado en memoria para ser utilizado luego. La instrucción indicada con *** sería innecesaria si se dispusiera de un registro más en la CPU. Se evitarían dos accesos a memoria (uno para guardar y otro para recuperar el intermedio).

¿Cómo modificaría el repertorio de instrucciones de MARIE y su codificación para incorporar el nuevo registro? Suponga que los nombra A y B, como en la arquitectura Intel. ¿Qué deberá sacrificar si desea mantener el largo de las instrucciones en 16 bits?

¿Y si en lugar de modificar la arquitectura cambio el algoritmo? Me paso a "modo álgebra" y

$$f = (g + h) - (i + j) = g + h - i - j$$

```
Load g
Add h
Subt i
Subt j
Store f
```

Se resuelve en MARIE con 3 operaciones de la ALU y 5 accesos a memoria (el mínimo). En lugar de modificar la arquitectura, busqué la forma de resolver el problema que mejor se adecúa a los recursos disponibles.

Resolví el mismo problema con 5 instrucciones en vez de 7, obteniendo una mejora del 40%!

## ANEXO I: Acerca de la correcta redacción de código

Es muy recomendable redactar código ordenado por secciones, con un encabezado informativo y comentarios precisos, no obvios. También es útil nombrar las variables en forma que ayude a la comprensión del código.

```
//
// A + B = C
// Autor, fecha, version
//
     Load A  // Sección de instrucciones (code)
     Add B
     Store C
     Halt
//
A,   DEC 35  // Sección de constantes y variables (data)
B,   DEC -23
C,   DEC 0
```

## ANEXO II: Resumen de la arquitectura MARIE

MARIE es una arquitectura diseñada con fines didácticos, presentada en el texto “The Essentials of Computer Organization and Architecture” de Linda Null y Julia Lobur. En las clases teóricas se describió en detalle su diseño y operación. A continuación se realiza un resumen de sus características principales con el objetivo de que pueda visualizarse la arquitectura en su totalidad. La información está ordenada tal que pueda utilizarse para realizar una comparación ordenada con otra arquitectura. Nótese la diferencia que se hace entre la funcionalidad de la máquina (el repertorio de instrucciones) y su implementación (el circuito digital que se utiliza para ello).

**Principales características del repertorio de instrucciones de MARIE**

- Representación de datos en complemento a dos en 16 bits
- Programa almacenado, largo de instrucción fijo de 16 bits
- 4K palabras de memoria principal (12 bits por dirección)
- Program counter de 12 bits (PC)
- Un registro de usuario de 16 bits (AC, acumulador)
- Las instrucciones tienen un formato uniforme:
    - Los bits 15-12 de la instrucción componen el código de operación. Con 4 bits de código de operación pueden implementarse 16 instrucciones diferentes (24).
    - Los bits 11-0 son el argumento, que en la mayoría de las instrucciones se trata de una dirección de memoria. Con 12 bits pueden direccionarse 4 Kbytes de memoria (212=4096)

La entrada/salida se realiza a través de dos registros de 8 bits (en el simulador son de 16 bits) que se acceden utilizando instrucciones específicas (no están mapeados en la memoria).

- Input register
- Output register

Recordar que se trata de un procesador del tipo (1,1): todas las instrucciones son referidas al único registro disponible (acumulador). Las instrucciones que requieren con dos argumentos utilizan el acumulador y la posición de memoria indicada en el cuerpo de la instrucción (argumento). El resultado de las instrucciones es almacenado en el acumulador.

## MARIE ISA

| Opcode | Instruction | Description |
|---|---|---|
| 0 | JnS X | Store the PC at address X and jump to X+1 |
| 1 | Load X | Load contents of address X into AC |
| 2 | Store X | Store the contents of AC at address X |
| 3 | Add X | Add the contents of address X to AC |
| 4 | Subt X | Subtract the contents of address X from AC |
| 5 | Input | Input a value from the keyboard into AC |
| 6 | Output | Output the value in AC to the display |
| 7 | Halt | Terminate program |
| 8 | Skipcond X | Skip next instruction on condition (See note below.)<br><br>The two address bits closest to the opcode field, bits 10 and 11 specify the condition to be tested. If the two address bits are 00, this translates to "skip if the AC is negative". If the two address bits are 01, this translates to "skip if the AC is equal to 0". Finally, if the two address bits are 10 (or 2), this translates to "skip if the AC is greater than 0".<br><br>Example: the instruction Skipcond 800 will skip the instruction that follows if the AC is greater than 0. |
| 9 | Jump X | Load the value of X into PC |
| A | Clear | Put all zeros in AC |
| B | AddI X | Add indirect: Use the value at X as the actual address of the data operand to add to AC |
| C | JumpI X | Use the value at X as the address to jump to |
| D | LoadI X | Load indirect: Use the value at X as the address of the value to load. |
| E | StoreI X | Store indirect: Use X the value at X as the address of where to store the value. |

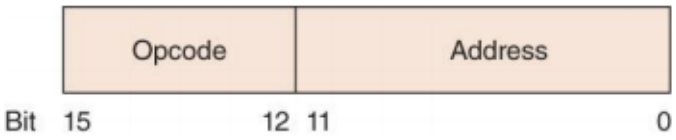


*Figura 1: Repertorio de instrucciones completo de MARIE. La ejecución de programas puede simularse utilizando la herramienta online https://marie.js.org/.*

**Un ejemplo simple de programación**

Se analiza una secuencia de instrucciones para sumar dos números que están almacenados en las posiciones de memoria A y B, y almacenar el resultado en la posición de memoria C. Utilizaremos sólo las tres primeras instrucciones de MARIE:

- OPCODE=0001) Load X: Cargar el acumulador con el contenido de la posición de memoria X

- OPCODE=0010) Store X: Guardar el contenido del acumulador en la posición de memoria X
- OPCODE=0011) Add X: Sumar el contenido del acumulador con el contenido de la posición de memoria X y guardar el resultado en el acumulador
- OPCODE=0111) Halt: Terminar el programa

| Dirección (HEX) | Dirección (BIN) | Contenido (HEX) | Contenido (BIN) | Instrucción (ASM) | |
|---|---|---|---|---|---|
| 000 | 000000000000 | 1004 | 0001000000000100 | | Load A |
| 001 | 000000000001 | 3005 | 0011000000000101 | | Add B |
| 002 | 000000000010 | 2006 | 0010000000000110 | | Store C |
| 003 | 000000000011 | 7000 | 0111000000000000 | | Halt |
| 004 | 000000000100 | 0023 | 0000000000100011 | A, | DEC 35 |
| 005 | 000000000101 | FFE9 | 1111111111101001 | B, | DEC -23 |
| 006 | 000000000110 | 0000 | 0000000000000000 | C, | DEC 0 |

Luego de ejecutarse el programa, la posición de memoria 006 contiene el resultado 35 + (-23) = 12 = 0x000C

RECORDAR: el *assembler* es un programa que toma el código de la 5ta columna escrito en *assembly language* y lo convierte a la 4ta columna en código máquina binario. El *loader* transfiere el código de la 4ta columna a las direcciones de memoria de la máquina de destino indicadas por la 2a columna. Al reiniciar la máquina de destino comienza la ejecución del programa.

**La organización de MARIE**

La Figura 1 muestra una posible implementación del repertorio de instrucciones de MARIE, utilizando una organización de von Neumann sin segmentación, que requiere 7 ciclos de reloj por instrucción. Esta es la organización detallada en el texto “The Essentials of Computer Organization and Architecture” de Linda Null.

Utiliza los siguientes registros adicionales, necesarios para la operación, pero que no son accesibles a través del repertorio de instrucciones:

- Un registro de instrucción de 16 bits (IR)
- Memory address register de 12 bits (MAR)
- Memory buffer register de 16 bits (MBR)

MAR y MBR son la interfaz con la memoria principal, la cual almacena tanto instrucciones como datos. IR almacena la instrucción en proceso que fue recuperada de la dirección de memoria apuntada por el PC.

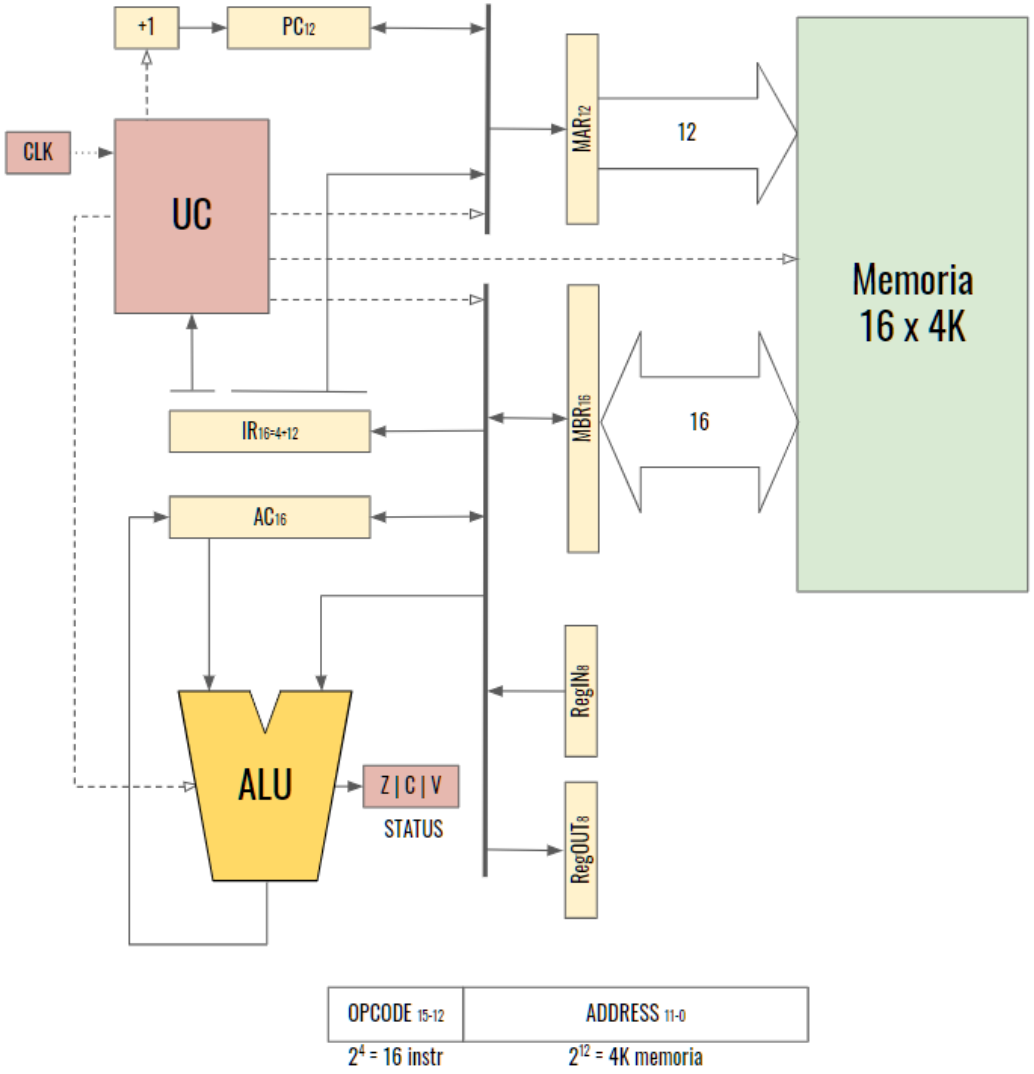


*Figura 1: Organización básica de MARIE: memoria única con un solo bus de acceso que se utiliza para almacenar tanto los datos como las instrucciones (arquitectura de von Neumann).*

Utilizando Register Transfer Notation (RTN) puede describirse en detalle el funcionamiento de la organización:

a) Todas las instrucciones tienen la misma secuencia inicial (FETCH/DECODE):

```
MAR <- PC
MBR <- M[MAR]
IR  <- MBR          UC decodifica IR[15-12]
PC  <- PC + 1
```

b) La etapa de ejecución (EXECUTE) se diferencia según del código de operación decodificado:

```
IR[15-12] == 0001 (Load)
```

```
MAR <- IR[11-0]
MBR <- M[MAR]
AC  <- MBR
IR[15-12] == 0010 (Store)
MAR <- IR[11-0]
MBR <- AC
M[MAR] <- MBR
IR[15-12] == 0011 (Add)
MAR <- IR[11-0]
MBR <- M[MAR]
AC  <- AC + MBR
```

Utilizando https://marie.js.org/ puede simularse el flujo en RTN. Habilitando en el menú View -> Datapath, y avanzando con [Microstep] puede verse una representación gráfica bastante didáctica.

En esta organización todas las instrucciones utilizan 7 “pasos” que pueden ser implementados en la Unidad de Control (UC) utilizando una máquina de estados simple que tomaría 7 ciclos de reloj por instrucción.

**Organización mejorada**

Para mejorar la organización, ¿se podría adelantar alguno de esos pasos? ¿O realizar dos en paralelo? ¿Se pueden superponer las distintas etapas de diferentes instrucciones? En las clases de teoría se discutieron algunas posibles mejoras en la organización de MARIE. Por ejemplo, en la Figura 2 se muestra una implementación Harvard segmentada del mismo repertorio de instrucciones. En la teoría mostramos cómo, en determinadas condiciones, esta organización podría llegar a ejecutar una instrucción MARIE por ciclo de reloj, excepto los saltos que tomarían dos ciclos.

Es importante notar que se trata del mismo repertorio de instrucciones, implementado con una organización más elaborada: más rápida pero a la vez más costosa. La misma funcionalidad, diferente circuito.

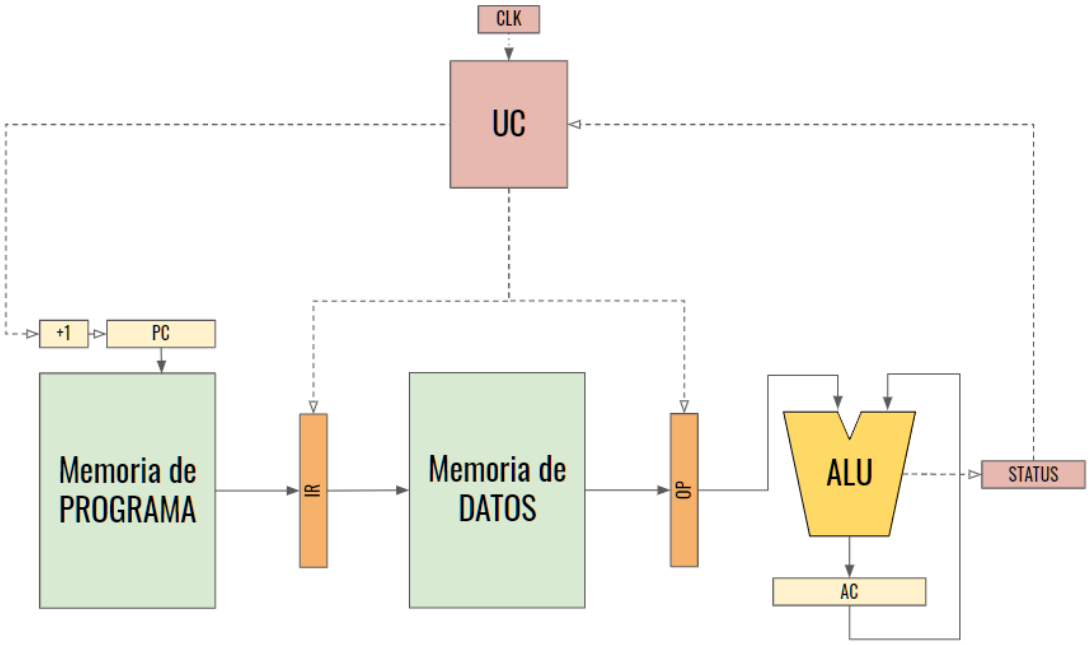


*Figura 2: Una posible organización avanzada de MARIE: acceso Harvard a la memoria con segmentación del ciclo de instrucción.*

**Expresión de la mejora**

¿Cómo se podría expresar la mejora que se logra con la modificación de la organización?

Primero es importante notar que la mejora no será igual para todos los programas. Depende de la proporción de instrucciones de salto.

Por ejemplo considere el siguiente programa:

```
/ Repetición: decrementar Y hasta cero
loop,    Load Y
         Subt one
         Store Y
         Skipcond 400  / completamos el loop?
         Jump loop     / no: repetir
Halt                   / si: terminar
/
Y, DEC 10
one, DEC 1
```

Este programa repite 10 veces una secuencia de 5 instrucciones, de las cuales 1 es un salto.

En la versión von Neumann de MARIE (Figura 1) todas las instrucciones se realizan en 7 ciclos de reloj. Por lo tanto el programa tomará

*t1 = 10 iteraciones * 5 instrucciones * 7 ciclos/instrucción = 350 ciclos*

En la versión Harvard de MARIE (Figura 2) el programa se completará en

*t2 = 10 * (4*1+1*2) = 60 ciclos*

Entonces, ¿cuánto mejor es la versión mejorada?