

Práctica 4

Diseño de un procesador segmentado de 5 etapas



JAVIER MENGOD Y ÁLVARO FRAIDIAS

Contenido

[1. Trabajo previo 2](#_Toc71223093)

[1.1 Imagen 2](#_Toc71223094)

[1.2 Formato de instrucciones 2](#_Toc71223095)

[1.2.1 Código de operación 2](#_Toc71223096)

[1.2.2 ADD 2](#_Toc71223097)

[1.2.3 LOAD, STORE Y SALTO CONDICIONAL 2](#_Toc71223098)

[1.2.4 MOV 3](#_Toc71223099)

[1.2.5 Nop 4](#_Toc71223100)

[1.3 Camino de datos de las nuevas instrucciones 4](#_Toc71223101)

[1.3.1 Load y store 4](#_Toc71223102)

[1.3.2 Beq 5](#_Toc71223103)

[1.4 Programa en ensamblador 5](#_Toc71223104)

[2. Procesador Multiciclo 7](#_Toc71223105)

[2.1 Autómata de estados 7](#_Toc71223106)

[2.2 Tabla de señales de control 7](#_Toc71223107)

[3. Procesador Segmentado 8](#_Toc71223108)

[3.1 Unidad de control 9](#_Toc71223109)

[3.3 Programa en ensamblador 10](#_Toc71223110)

[4.Análisis temporal 13](#_Toc71223111)

[4.1 Máquina multiciclo 13](#_Toc71223112)

[4.1.1 Número de ciclos necesarios 13](#_Toc71223113)

[4.1.2 Tiempo de ciclo 13](#_Toc71223114)

[4.2 Máquina segmentada 15](#_Toc71223115)

[4.2.1 Número de ciclos sin reorganizar el código 15](#_Toc71223116)

[4.2.2 Número de reorganizando el código 16](#_Toc71223117)

[4.2.3 Tiempo de ciclo 17](#_Toc71223118)

[5. Problemas durante la práctica 20](#_Toc71223119)

[5.1 Instrucción Beq 20](#_Toc71223120)

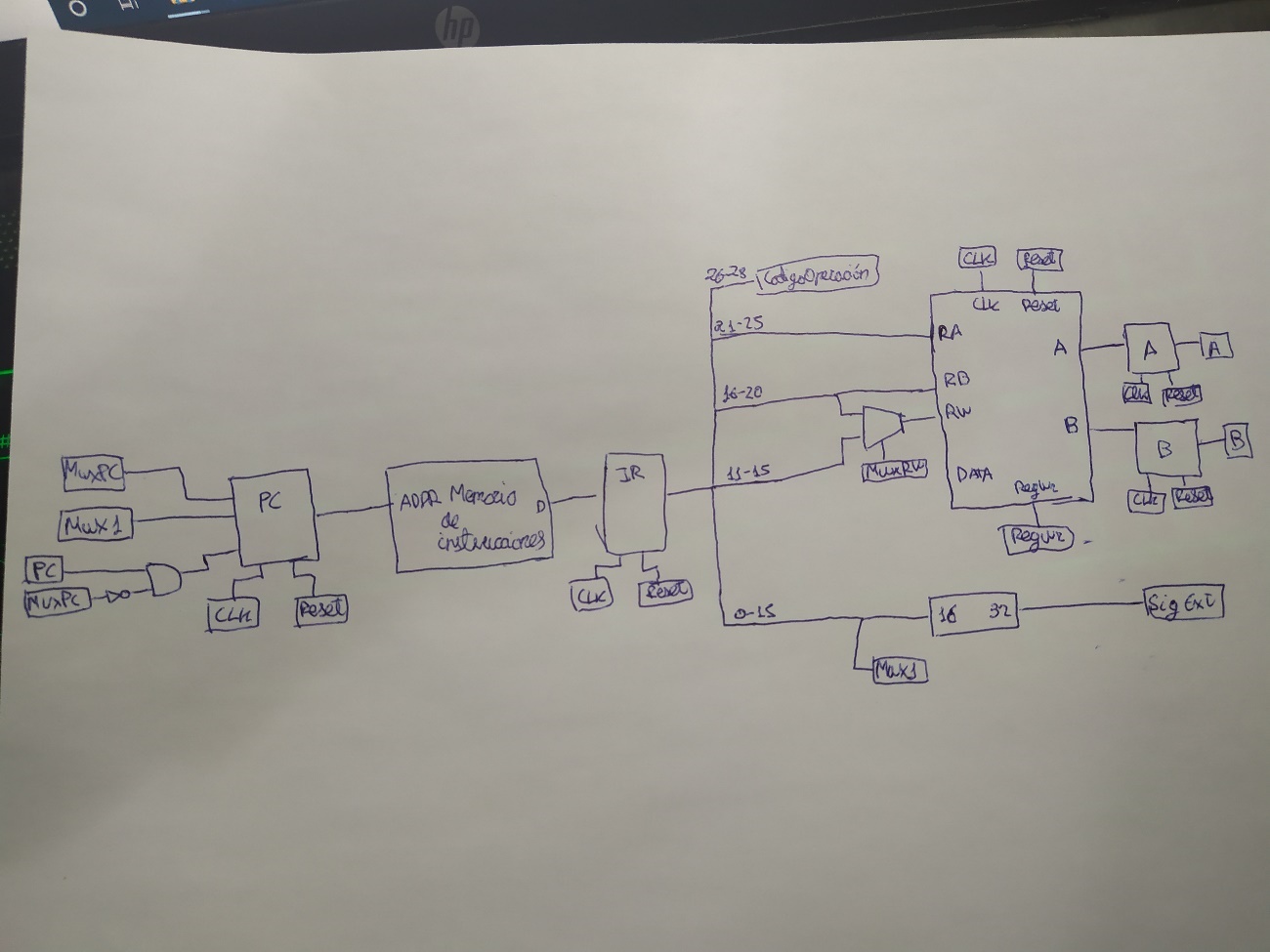
[5.2 Memoria RAM 21](#_Toc71223121)

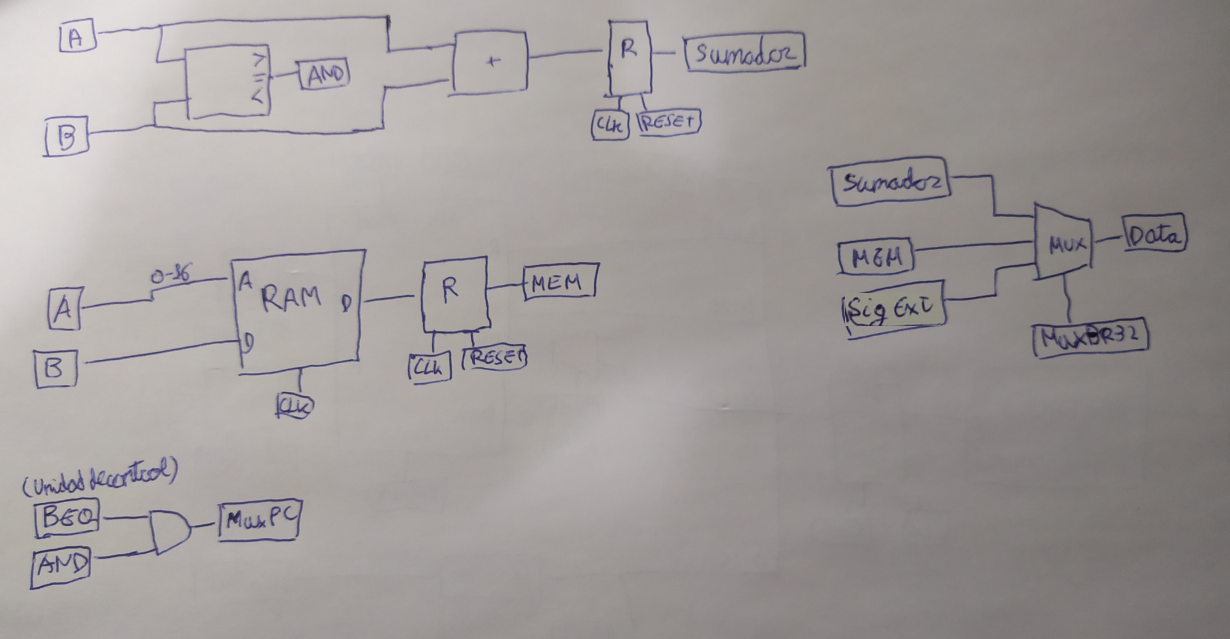
[5.3 Banco de registros con lectura-escritura 22](#_Toc71223122)

# 

# 1. Trabajo previo

## 1.1 Imagen





## 1.2 Formato de instrucciones

### 1.2.1 Código de operación

En este caso, al ejecutar 6 instrucciones, necesitamos 3 bits para codificar dichas instrucciones ya que con 1 bit codificaríamos 2 instrucciones y con 2 bits 4 instrucciones. El código de operación de estas instrucciones son los siguientes:

* Mov = 001
* ADD = 101
* LOAD = 010
* STORE = 011
* BEQ = 100
* Nop = 110

### 1.2.2 ADD

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Código operación | RA | RB | RW | NADA |

31 26 25 21 20 16 15 11 10 0

Esta operación suma dos registros del BR32 (RA y RB) y lo guarda en un tercer registro del banco de registros (RW).

* Del bit 0 al 11 (12 bits) la instrucción no nos dará ninguna información.
* Del bit 11 al 15 (5 bits) estará codificado el registro destino del banco de registros donde se guardará la suma.
* Del bit 16 al 20 (5 bits) irá codificado el segundo operando del sumador proveniente de los 32 registros del banco de registros.
* Del bit 21 al 25 (5 bits) vendrá codificado el primer operando del sumador proveniente de los 32 registros del banco de registros.
* Del bit 26 al 31 (5 bits) vendrá dado el código de operación.

### 1.2.3 LOAD, STORE Y SALTO CONDICIONAL

|  |  |  |  |
| --- | --- | --- | --- |
| Código operación | RA | RB | INMEDIATO |

31 26 25 21 20 16 15 0

RA nunca es destino en BR32, pero sí en la memoria en la store. En cambio, RB sí que es destino en el BR32 en la load.

#### 1.2.3.1 LOAD

Esta operación carga en un registro del BR32 (RB) un dato de la memoria RAM proveniente de la dirección almacenada por otro registro del BR32 (RA).

* En este caso los bits del 0-15 (16 bits) no nos aporta ninguna información.
* En los bits del 16-20 (5 bits) el BR32 nos dará el registro donde se guardará el dato proveniente de memoria.
* En los bits del 21-25 (5 bits) el BR32 nos dará el registro donde se obtendrá la dirección de memoria de la RAM cuyo dato queremos cargar.
* Del bit 26-31 (6 bits) se encuentra el código de operación.

#### 1.2.3.2 STORE

Esta operación escribe un dato del BR32 (RB) en una dirección de la memoria RAM dada por otro registro del BR32 (RA).

* Igual que en el caso de la instrucción load, los bits del 0-15 (16 bits) no nos aporta ninguna información.
* En los bits del 16-20 (5 bits) se encuentra el registro del BR32 cuyo dato que almacena queremos guardar en la memoria RAM.
* En los bits del 21-25 (5 bits) se encuentra el registro del BR32 cuyo dato es la dirección de memoria donde queremos guardar el dato de RB.
* Del bit 26-31 (6 bits) se encuentra el código de operación

#### 1.2.3.3 BEQ

Esta operación compara dos registros del BR32 (RA y RB) y en caso de que sean iguales la memoria de instrucciones salta a la dirección dada por el número inmediato.

* Del bit 0-15(16 bits) estará codificada la dirección a la que se actualizará la memoria de instrucciones en caso de que se cumpla la instrucción.
* Del bit 16-20 (5 bits) vendrá el segundo operando que se compara.
* Del bit 21-25 (5 bits) vendrá el primer operando que se compara.
* Del bit 26-31 (6 bits) se encuentra el código de operación.

### 1.2.4 MOV

|  |  |  |  |
| --- | --- | --- | --- |
| Código operación | NADA | RW | INMEDIATO |

31 26 25 21 20 16 15 0

Esta instrucción carga en el registro destino del BR32 (RW) el número codificado como inmediato.

* Del bit 0 al 15 (16 bits) irá codificado el número que se pretende cargar en el registro.
* Del bit 16 al 20 (5 bits) estará codificado el registro destino donde se guardará el número inmediato.
* Del bit 21 al 25 (5 bits) nos dará igual los bits que haya.
* Del bit 26 al 31 (6 bits) irá codificado el código de operación que al pasar por la UC determinará el valor de las señales de control.

### 1.2.5 Nop

|  |  |
| --- | --- |
| Código operación | Nada |

31 26 25 0

Esta instrucción se usa para bloquear la pipeline del procesador segmentado para evitarnos riesgos en los datos.

* Del bit 26 al 31 (6 bits) irá codificado el código de operación que al pasar por la UC generará todas las señales de control a 0 para que no se ejecute ninguna instrucción.
* Desde el bit 0 al 25 (26 bits) no nos aportará ninguna información ya que al poner todas las señales a 0 daría igual que registros estuvieran codificados entre estos bits.

## 1.3 Camino de datos de las nuevas instrucciones

### 1.3.1 Load y store

Para añadir estas instrucciones debemos conectar una memoria RAM a las salidas de los registros temporales que guardan las salidas del banco de registros ya que uno de esos registros nos dará la dirección de memoria donde leer o escribir el dato y el otro el registro donde quedemos guardar el dato en el caso de la load o el dato que queremos escribir en la RAM en caso de la store:

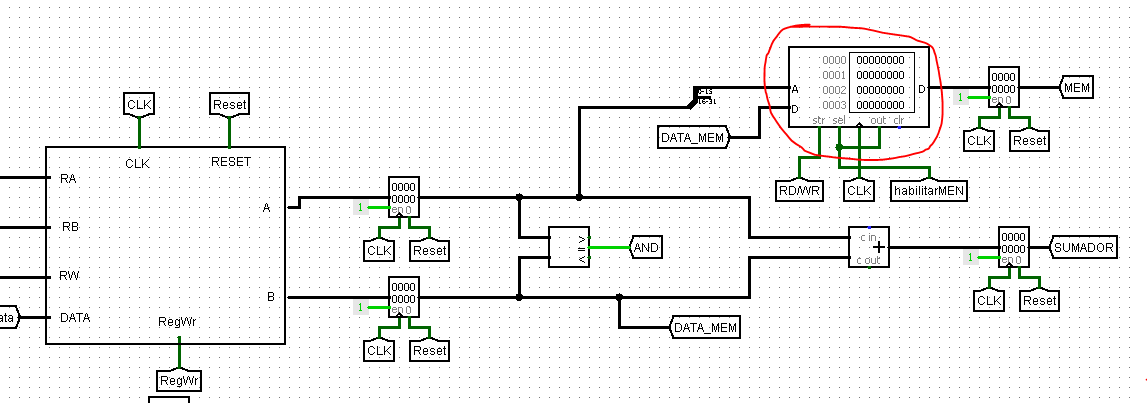


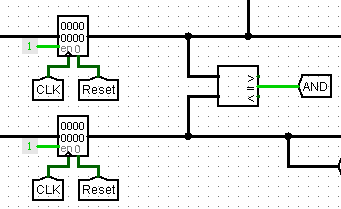
Ilustración : Camino de datos instrucciones LOAD y STORE

En la teoría dada en clase se implementa el procesador multiciclo con una única memoria unificada para instrucciones y datos, pero en nuestro caso la hemos dividido en una memoria ROM para las instrucciones y una memoria RAM para los datos.

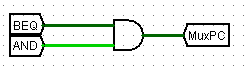
### 1.3.2 Beq

Para implementar esta instrucción en nuestro procesador debemos añadir 3 apartados diferentes:

* Un comparador para comprobar si los dos registros de la instrucción son iguales. En caso de que sean iguales generará una señal que se pondrá a 1

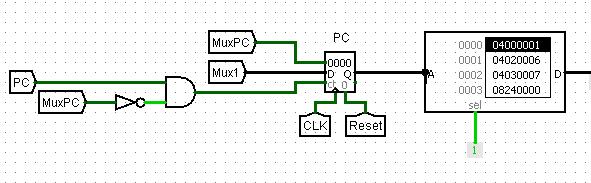


* Una puerta and cuyas entradas sean la salida del comparador mencionado anteriormente y una señal beq ya que si no hiciéramos esto el procesador saltaría si los dos registros fuesen iguales sin que necesariamente sea una instrucción Beq.



La señal resultante de la comparación y la señal Beq sirve para decidir si se escribirá en el contador la dirección actual incrementada en uno para pasar a la siguente dirección o la dirección codificada en los bits 0-15 de la instrucción Beq.

* Debemos modificar las entradas al contador PC ya que antiguamente era un contador incremental y ahora puede incrementarse o saltar a una dirección completamente diferente:



## 1.4 Programa en ensamblador

Hemos creado un programa que mediante un bucle carga en el banco de registros uno a uno los datos de la memoria RAM para sumarlos y el resultado total guardarlo en dicha memoria RAM. El código a nivel ensamblador quedaría de la siguiente manera:

mov $0, #1 Iterador

mov $2, #6 Cantidad de veces que se ejecuta el bucle

mov $3, #7 Dir.MEN donde se guarda el resultado

INICIO

ld $4, ($1) Carga en $4 el dato de memoria

add $1, $1, $0 Itera la dirección de memoria

add $5, $5, $4 Suma el dato

beq $1, $2, FIN Si se cumple va a la etiqueta “FIN”

beq $0, $0, INICIO Si se cumple va a la etiqueta “INICIO”

FIN

St $3, ($5) Guarda en la dirMen $3 lo que hay en $5 EEEEEEEEEEEEEEEEEEEEE(el resultado de la suma)

Escrito en binario y en hexadecimal quedaría de la siguiente manera:

mov $0, #1

**Binario:**

00000100000000000000000000000001

**Hexadecimal:**

04000001

mov $2, #6

**Binario:**

00000100000000100000000000000110

**Hexadecimal:**

04020006

mov $3, #7

**Binario:**

00000100000000110000000000000111

**Hexadecimal:**

04030007

INICIO

ld $4, ($1)

**Binario:**

00001000001001000000000000000000

**Hexadecimal:**

08240000

add $1, $1, $0

**Binario:**

00010100001000000000100000000000

**Hexadecimal:**

14200800

add $5, $5, $4

**Binario:**

00010100101001000010100000000000

**Hexadecimal:**

14A42800

beq $1, $2, FIN

**Binario:**

00010000001000100000000000001000

**Hexadecimal:**

10220008

beq $0, $0, INICIO

**Binario:**

00010000000000000000000000000011

**Hexadecimal:**

10000003

FIN

St $3, ($5)

**Binario:**

00001100011001010000000000000000

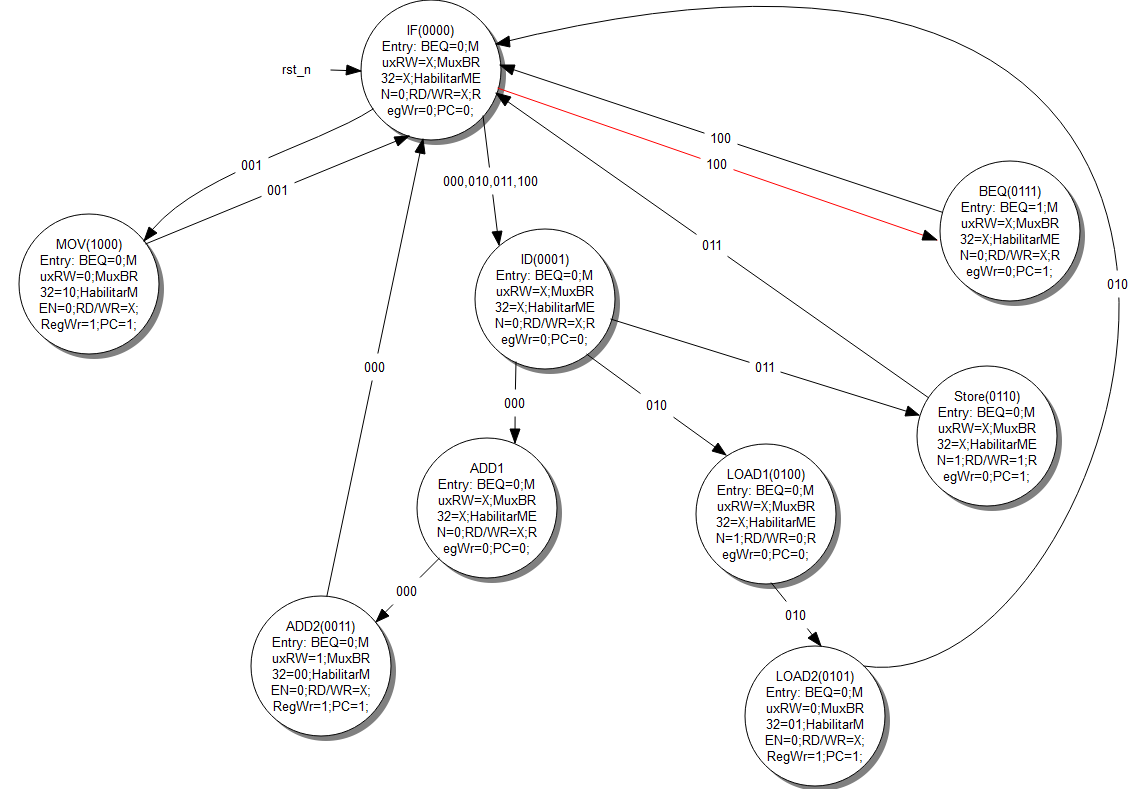
**Hexadecimal:**

0C650000

# 2. Procesador Multiciclo

## 2.1 Autómata de estados

Este autómata tendrá más estados ya que la nueva máquina soportará más instrucciones. Por ello los estados se codificarán con 4 bits y se saltará entre un estado u otro mediante los 3 bits que componen el código de operación:



## 2.2 Tabla de señales de control

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | IF  (0000) | ID  (0001) | ADD1 (0010) | ADD2 (0011) | LOAD1  (0100) | LOAD2  (0101) | STORE  (0110) | BEQ  (0111) | MOV  (1000) |
| BEQ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| MuxRW | X | X | X | 1 | X | 0 | X | X | 0 |
| MuxBR32 | X | X | X | 00 | X | 01 | X | X | 10 |
| HabilitarMEN | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| RD/WR | X | X | X | X | 1 | X | 0 | X | X |
| RegWr | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| PC | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ENTRADAS | | SALIDAS | | | | | | | |
|  | Codigo operacion | Estado | BEQ | MuxRW | MuxBR32 | HabilitarMEN | RD/WR | REGWR | PC | SigEstado |
| IF | 001 | 0000 | 0 | X | XX | 0 | X | 0 | 0 | 1000 |
| MOV | 001 | 1000 | 0 | 0 | 10 | 0 | X | 1 | 1 | 0000 |
|  |  |  |  |  |  |  |  |  |  |  |
| IF | 101 | 0000 | 0 | X | XX | 0 | X | 0 | 0 | 0001 |
| ID | 101 | 0001 | 0 | X | XX | 0 | X | 0 | 0 | 0010 |
| ADD1 | 101 | 0010 | 0 | X | XX | 0 | X | 0 | 0 | 0011 |
| ADD2 | 101 | 0011 | 0 | 1 | 00 | 0 | X | 1 | 1 | 0000 |
|  |  |  |  |  |  |  |  |  |  |  |
| IF | 010 | 0000 | 0 | X | XX | 0 | X | 0 | 0 | 0001 |
| ID | 010 | 0001 | 0 | X | XX | 0 | X | 0 | 0 | 0100 |
| LOAD1 | 010 | 0100 | 0 | X | XX | 1 | 0 | 0 | 0 | 0101 |
| LOAD2 | 010 | 0101 | 0 | 0 | 01 | 0 | X | 1 | 1 | 0000 |
|  |  |  |  |  |  |  |  |  |  |  |
| IF | 011 | 0000 | 0 | X | XX | 0 | X | 0 | 0 | 0001 |
| ID | 011 | 0001 | 0 | X | XX | 0 | X | 0 | 0 | 0110 |
| STORE1 | 011 | 0110 | 0 | X | XX | 1 | 1 | 0 | 1 | 0000 |
|  |  |  |  |  |  |  |  |  |  |  |
| IF | 100 | 0000 | 0 | X | XX | 0 | X | 0 | 0 | 0001 |
| ID | 100 | 0001 | 0 | X | XX | 0 | X | 0 | 0 | 0111 |
| BEQ | 100 | 0111 | 1 | X | XX | 0 | X | 0 | 1 | 0000 |
|  |  |  |  |  |  |  |  |  |  |  |

# 3. Procesador Segmentado

Para realizar un procesador segmentado en 5 etapas debemos saber que los datos que no se guarden en un registro de un ciclo a otro se perderán para siempre por lo que es necesario guardar todos los datos necesarios en registros de la siguiente forma:

* Registro IF/ID:
  + Instrucción a ejecutar (32 bits)
* Registro ID/EX:
  + Señales de control de la unidad de control (8 bits)
  + Primer operando del banco de registros (A) (32 bits)
  + Segundo operando del banco de registros (B) (32 bits)
  + Inmediato extendido de signo de la instrucción MOV (32 bits)
  + Dirección salto condicional (16 bits)
  + Los 10 bits que deciden el registro destino usado en la etapa WB
* Registro EX/MEN:
  + Señales de control de la unidad de control (7 bits)
  + Resultado sumador (32 bits)
  + Resultado comparador (1 bit)
  + Registro destino del BR32 (5 bits)
  + Dirección de memoria (A) (32 bits)
  + Dato a escribir en la memoria (B) (32 bits)
  + Inmediato extendido de signo de la instrucción MOV (32 bits)
  + Dirección salto condicional (16 bits)
* Registro MEN/WB:
  + Señales de control de la unidad de control (4 bits)
  + Dato de memoria (32 bits)
  + Resultado sumador (32 bits)
  + Inmediato extendido de signo de la instrucción MOV (32 bits)
  + Registro destino del BR32 (5 bits)

## 3.1 Unidad de control

La unida de control de la máquina segmentada es más simple que la máquina multiciclo ya que no precisa de estados por lo que funciona de la siguiente manera: Tiene como entradas los bits del código de operación y en base a ellos genera a la vez todas las señales de control necesarias. Puesto en una tabla sería de la siguiente manera:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Señales de control | | | | | |
| Codigo operación | MuxRW | Beq | HabilitarMEN | RD/WR | MuxBR32 | RegWr |
| ADD 101 | 1 | 0 | 0 | 0 | 00 | 1 |
| MOV 001 | 0 | 0 | 0 | 0 | 10 | 1 |
| LOAD 010 | 0 | 0 | 1 | 0 | 01 | 1 |
| STORE 011 | 0 | 0 | 1 | 1 | 00 | 0 |
| BEQ 100 | 0 | 1 | 0 | 0 | 00 | 0 |
| Nop 110 | 0 | 0 | 0 | 0 | 00 | 0 |

## 3.3 Programa en ensamblador

El programa propuesto en el punto 1.4 en el caso del procesador segmentado no nos sirve ya que habría varios riesgos de datos y de control. La solución propuesta es codificar instrucciones nop para bloquear el pipeline y que se ejecute todo correctamente. El código quedaría de la siguiente manera

mov $0, #1

mov $1, #0

mov $2, #6

mov $3, #7

**nop**

INICIO

ld $4, ($1)

add $1, $1, $0

**nop**

**nop**

add $5, $5, $4

beq $1, $2, FIN

**nop**

**nop**

**nop**

beq $0, $0, INICIO

**nop**

**nop**

**nop**

FIN

St $3, ($5)

Ahora traducimos el programa a binario y más tarde a hexadecimal ya que es así como se guarda en la memoria ROM

mov $0, #1

**Binario:**

00000100000000000000000000000001

**Hexadecimal:**

04000001

mov $2, #6

**Binario:**

00000100000000100000000000000110

**Hexadecimal:**

04020006

mov $3, #7

**Binario:**

00000100000000110000000000000111

**Hexadecimal:**

04030007

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

INICIO

ld $4, ($1)

**Binario:**

00001000001001000000000000000000

**Hexadecimal:**

08240000

add $1, $1, $0

**Binario:**

00010100001000000000100000000000

**Hexadecimal:**

14200800

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

add $5, $5, $4

**Binario:**

00010100101001000010100000000000

**Hexadecimal:**

14A42800

beq $1, $2, FIN

**Binario:**

00010000001000100000000000001000

**Hexadecimal:**

10220008

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

beq $0, $0, INICIO

**Binario:**

00010000000000000000000000000011

**Hexadecimal:**

10000003

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

nop

**Binario:**

00011000000000000000000000000000

**Hexadecimal:**

18000000

FIN

St $3, ($5)

**Binario:**

00001100011001010000000000000000

**Hexadecimal:**

0C650000

Como se puede comprobar en la siguiente imagen se han solucionado todos los riesgos:

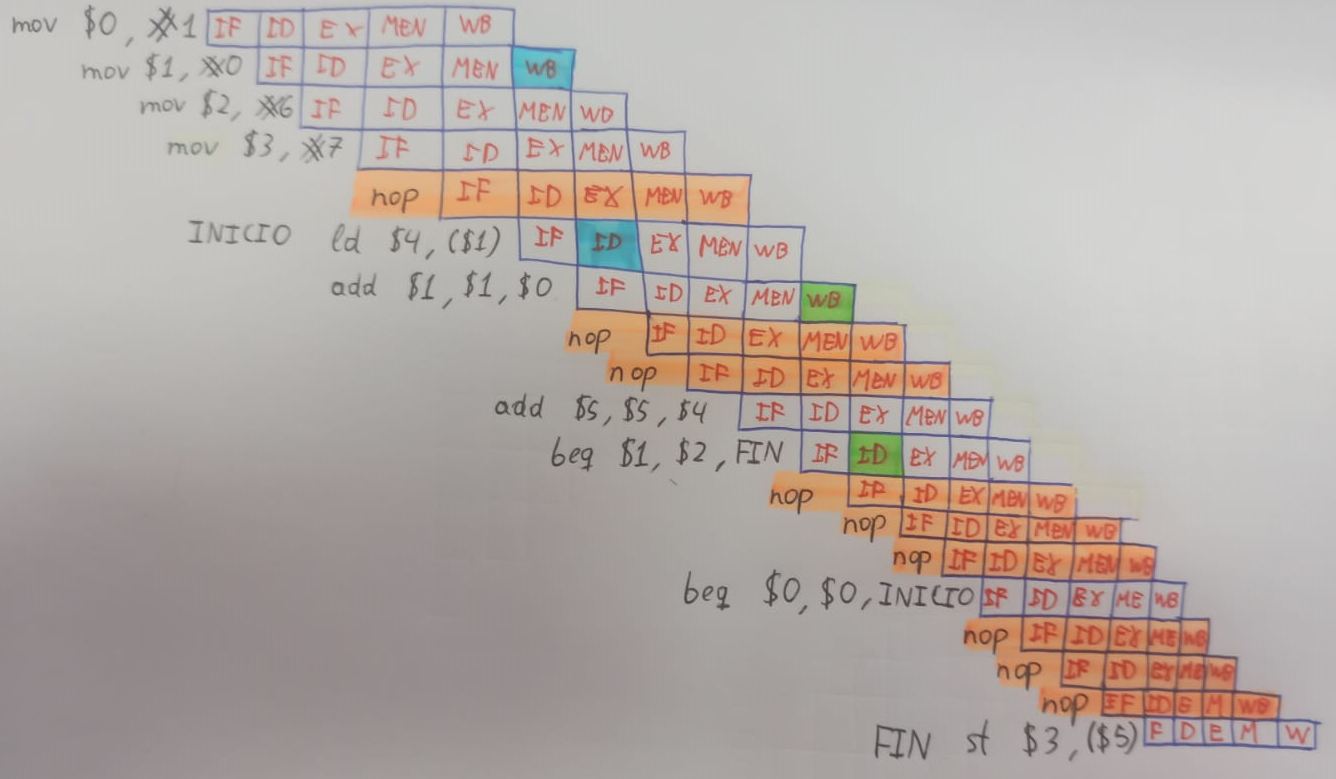


Ilustración : Nuestro programa mostrando todas las etapas

# 4.Análisis temporal

## 4.1 Máquina multiciclo

### 4.1.1 Número de ciclos necesarios

El numero de ciclos necesarios para ejecutar el programa será igual al numero de instrucciones del programa, por el número de etapas que tiene cada instrucción (recordamos que el tiempo de ciclo de una máquina multiciclo será igual al tiempo que tarda en hacer su etapa más larga).

Las instrucciones mov tienen dos etapas (IF,MOV), las add tienen 4 etapas (IF,ID,ADD1,ADD2), las store tienen 3 etapas (IF, ID, STORE1), las load tienen 4 etapas (IF,ID,LOAD1,LOAD2) y las beq tienen 3 etapas (IF, ID, BEQ)

Dicho esto, vamos a proceder al cálculo:

Las instrucciones de antes del bucle son tres mov: 3 \* 2 = 6 ciclos

El bucle está compuesto por una ld, dos add y dos beq: 4 + 2 \* 4+ 2\*3 = 18 ciclos

Como el bucle se itera 6 veces: 18 \* 6 = 108 ciclos

La última instrucción es la store: 3 ciclos

6+108+3=117 ciclos

Sumando los resultados nos queda que el programa se ejecuta en 117 ciclos.

### 4.1.2 Tiempo de ciclo

Como en la etapa WB hay 3 caminos diferentes para escribir en el banco de registros (ADD, LOAD y MOV) ya no nos vale el multiplexor2:1 por lo que hay que crear un nuevo multiplexor y estudiar su temporización:

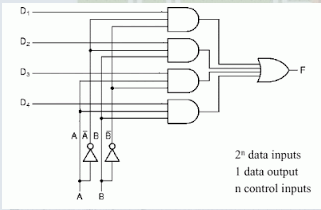


Ilustración : Multiplexor 4:1

Una vez definido y estudiado el nuevo componente debemos de suponer los siguientes tiempos en los diferentes componentes:

* Delay registro = 70ps
* Tiempo setup registro = 30ps
* Delay banco de registros = 184ps
* Tiempo setup banco de registros = 101ps
* Delay Mux2:1 (Datos) = 60ps
* Delay Mux2:1 (Control) = 65ps
* Delay AND = 30ps
* Delay NOT = 5ps
* Delay Contador = 75ps
* Tiempo setup Contador = 60ps
* Delay RAM = 95ps
* Tiempo setup RAM = 80ps
* Delay ROM = 95ps
* Tiempo setup ROM = 80ps
* Delay Mux4:1 (Datos) = 60ps
* Delay Mux4:1 (Control) = 65ps
* Delay extensor signo = 10ps
* Delay sumador = 180ps
* Delay comparador = 180ps

TIEMPO EN LAS INSTRUCCIONES

* Mov:
  + Delay reg(IR) + delay sigextend + mux4:1 (Datos) + mux 2:1 (Datos) + tiempo setup banco de registros = 70 pc + 10 ps + 60 ps + 60 ps + 101 ps = 301 ps
  + Delay ROM + mux4:1 (control) + mux 2:1 (Control) =95 ps + 65 ps + 65 ps= 225ps
* Load
  + ID: Delay reg(IR) + Delay banco de registros + tiempo de setup registro= 70 ps + 184 ps + 30 ps = 284 ps
  + MEN:
    - Delay registro + delay RAM + tiempo setup registro = 70 ps + 95 ps + 30 ps =195 ps
    - Delay ROM + tiempo setup RAM = 95ps + 80 ps= 175 ps
* WB:
  + Delay reg + Mux4:1(Datos) + tiempo setup banco de registros= 70 ps +60 ps + 30 ps= 160 ps
  + Delay ROM + Mux4:1(control) = 95 ps + 65 ps = 160 ps
* Store
  + ID: Delay reg(IR) + Delay banco de registros + tiempo de setup registro 70 ps + 184 ps + 30 ps =284 ps
  + MEN: Delay reg + Tiempo setup RAM = 70 ps + 80 ps = 150 ps
* BEQ
  + ID: Delay reg(IR) + Delay banco de registros + tiempo de setup registro = 70 ps + 184 ps + 30 ps =284 ps
  + EX: Delay reg + delay comparador + delay AND + Delaly NOT + Delay AND + tiempo setup PC = 70 ps + 180 ps + 30 ps + 5 ps + 30 ps + 30 ps =345 ps
* ADD
  + ID: Delay reg(IR) + Delay banco de registros + tiempo de setup registro = 70 ps + 184 ps + 30 ps = 284 ps
  + EX: Delay reg + delay sumador + tiempo setup reg = 70 ps + 180 ps + 30 ps = 280 ps
  + WB
    - Delay reg + mux4:1(Datos) + Tiempo setup banco de registros = 70 ps + 60 ps + 101 ps = 161 ps
    - Delay ROM + mux4:1(control) = 95 ps + 65 ps = 160 ps

La etapa mas larga la podemos encontrar en la EX de la beq con 345 ps, por lo que el tiempo de ciclo será de 345 ps.

En cuanto al tiempo tardado por el procesador multiciclo en ejecutar el programa es:

Tiempo ciclo \* número de ciclos \* número instrucciones = 345 ps \* 1 \* 83 ciclos

= 28 635 ps

## 4.2 Máquina segmentada

Primero hemos calculado el tiempo que tardaría el procesador segmentado en ejecutar el programa escrito en el apartado 3.3 y más tarde lo analizaremos reorganizando el código para evitarnos las máximas instrucciones nops posibles:

### 4.2.1 Número de ciclos sin reorganizar el código

Para calcular el número de ciclos debemos dividir el código en tres partes:

* Código pre-blucle (instrucciones mov y una nop) = 5 ciclos
* Bucle = 6 iteraciones \* 13 ciclos
* Código post-bucle (instrucción store) = 1 ciclo

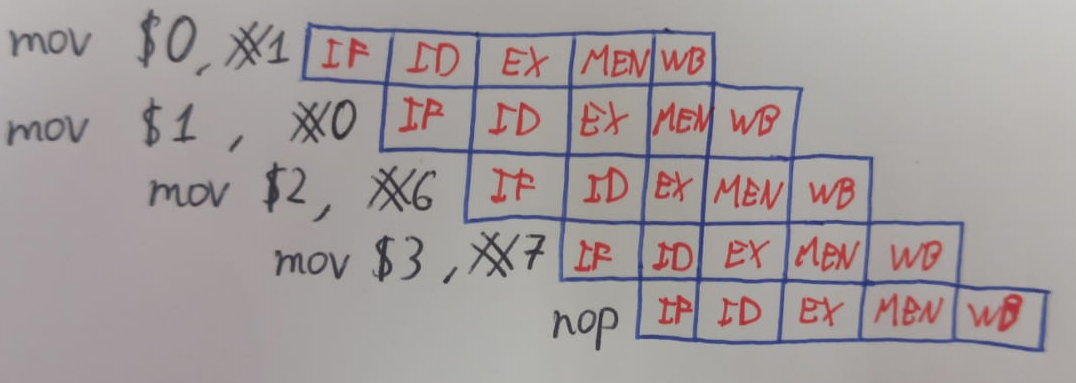


Ilustración : Código pre-bucle con 5 instrucciones

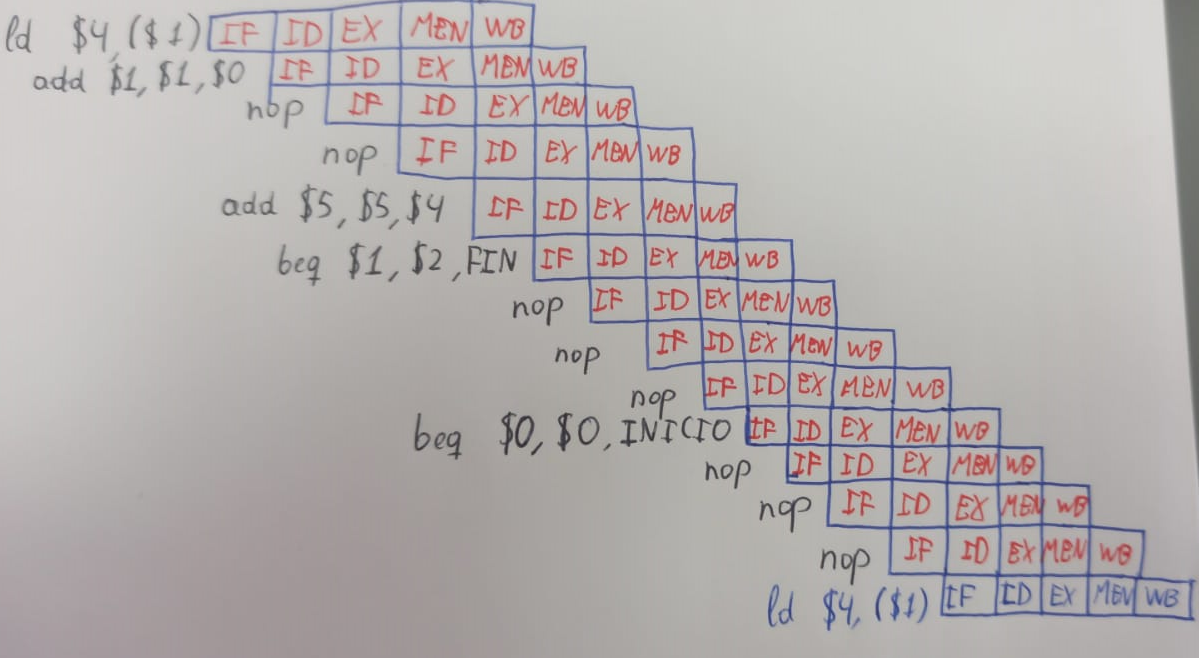


Ilustración : Bucle con 13 instrucciones

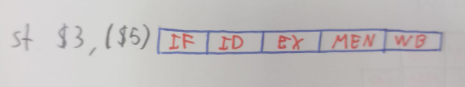


Ilustración : Código post-bucle con 1 instrucciónca

Por consiguiente, obtenemos que los ciclos necesarios son: 5 + (6\*13) + 1 = 84 ciclos

### 4.2.2 Número de reorganizando el código

Como podemos comprobar hay un riesgo entre la instrucción “mov $1, #0” y “ld $4, ($1)”. Para solucionarlo podemos subir la instrucción “mov $1, #0” y ponerla como la primera instrucción. Gracias a esto nos ahorraríamos un ciclo ya que no sería necesario ejecutar la primera nop. Las demás instrucciones no podemos colocarlas en otra posición diferente.

mov $1, #0

mov $0, #1

mov $2, #6

mov $3, #7

INICIO

ld $4, ($1)

add $1, $1, $0

**nop**

**nop**

add $5, $5, $4

beq $1, $2, FIN

**nop**

**nop**

**nop**

beq $0, $0, INICIO

**nop**

**nop**

**nop**

FIN

St $3, ($5)

El número de ciclos sería = 4 + (6\*13) + 1 = 83 ciclos

La mejora de reorganizar el código de este programa sería:

### 4.2.3 Tiempo de ciclo

Para calcular el tiempo de ciclo debemos calcular el tiempo de cada una de las diferentes etapas (IF,ID,EX,MEN,WB) en las diferentes instrucciones y quedarnos con la etapa más larga (camino crítico):

* IF: El camino de datos de esta etapa es común para todos:
  + El tiempo es el siguiente: Delay PC + Delay ROM + Tiempo setup registro IF/ID = 75ps + 95ps + 30ps = 200ps
* ID: En esta etapa hay 5 caminos diferentes:
  + **Instrucciones LOAD, STORE, BEQ, ADD**: Delay registro IF/ID + Delay banco de registros + Tiempo Setup registro ID/EX = 70ps + 184ps + 30ps = 284ps
  + **Unidad de control**: Delay registro IF/ID + Delay ROM + Tiempo Setup registro ID/EX = 70ps + 95ps + 30ps = 195ps
  + **Instrucción Mov**: Delay registro IF/ID + Delay extensor de signo + Tiempo Setup registro ID/EX = 70ps + 10ps + 30ps = 110ps
  + **Registro destino (RW):** Delay registro IF/ID + Tiempo Setup registro ID/EX

= 70ps + 30ps = 100ps

* + **Dirección salto**: Delay registro IF/ID + Tiempo Setup registro ID/EX

= 70ps + 30ps = 100ps

* EX: En esta etapa hay 7 caminos diferentes:
  + **ADD**: Delay registro ID/EX + Delay sumador + Tiempo Setup registro EX/MEN

= 70 ps + 180ps + 30ps = 280ps

* + **LOAD Y** **STORE**: Delay registro ID/EX + Tiempo Setup registro EX/MEN

= 70ps + 30ps = 100ps

* + **BEQ**: Delay registro ID/EX + Delay comparador + Tiempo Setup registro EX/MEN = 70ps + 180ps + 30ps = 280ps
  + **MOV**: Delay registro ID/EX + Tiempo Setup registro EX/MEN

= 70ps + 30ps = 100ps

* + **Registro destino (RW):** Delay registro ID/EX + Delay mux2:1 (Datos) + Tiempo Setup registro EX/MEN = 70ps + 60ps + 30ps = 160ps
  + **Dirección** **salto**: Delay registro ID/EX + Tiempo Setup registro EX/MEN

= 70ps + 30ps = 100ps

* + **Señal** **de** **control** **MuxRW**: Delay registro ID/EX + Mux2:1 (Control)

= 70ps + 75ps = 145ps

* MEN: En esta etapa hay 10 caminos diferentes:
  + **ADD**: Delay registro EX/MEN + Tiempo setup registro MEN/WB

= 70ps + 30ps = 100ps

**LOAD**: Delay registro EX/MEN + Delay RAM + Tiempo setup registro MEN/WB = 70ps + 95ps + 30ps = 195ps

* + **STORE**: Delay registro EX/MEN + Tiempo setup RAM

= 70ps + 80ps = 150ps

* + **BEQ**: Delay registro EX/MEN + Delay AND + Delay NOT + Delay AND + Tiempo setup PC = 70ps + 30ps + 5ps + 30ps + 30ps = 165ps

**MOV**: Delay registro EX/MEN + Tiempo setup registro MEN/WB

= 70ps + 30ps = 100ps

* + **Registro** **destino (RW):** Delay registro EX/MEN + Tiempo setup registro MEN/WB = 70ps + 30ps = 100ps
  + **Dirección** **salto**: Delay registro EX/MEN + Tiempo setup PC

= 70ps + 60ps = 130ps

* + **Señal de control BEQ**: Delay registro EX/MEN + Delay AND + Tiempo setup PC =70ps + 30ps + 60ps = 160ps
  + **Señal de control HabilitarMEN**: Delay registro EX/MEN + Tiempo setup RAM

= 70ps + 80ps = 150ps

* + **Señal de control RD/WR**: Delay registro EX/MEN + Tiempo setup RAM

= 70ps + 80ps = 150ps

* WB: En esta etapa hay 6 caminos diferentes:
  + **ADD**: Delay registro MEN/WB + Delay mux4:1 (Datos) + Tiempo setup banco registros = 70ps + 60ps + 101ps = 231ps
  + **MOV**: Delay registro MEN/WB + Delay mux4:1 (Datos) + Tiempo setup banco registros = 70ps + 60ps + 101ps = 231ps
  + **LOAD**: Delay registro MEN/WB + Delay mux4:1 (Datos) + Tiempo setup banco registros = 70ps + 60ps + 101ps = 231ps
  + **Registro destino (RW):** Delay registro MEN/WB + Tiempo setup banco registros = 70ps + 101ps = 171ps
  + **Señal de control MuxBR32**: Delay registro MEN/WB + Delay mux4:1 (Control)

= 70ps + 65ps = 135ps

* + **Señal de control RegWr**: Delay registro MEN/WB

= 70ps

Una vez calculados todos los caminos posibles nos damos cuenta que el camino crítico lo protagoniza la etapa ID con 284ps por lo que la frecuencia máxima de este procesador es:

3.521.126.760,56 Hz = 3,5Ghz

En cuanto al tiempo tardado por el procesador segmentado en ejecutar el programa es:

Tiempo ciclo \* número de ciclos \* número instrucciones = 284ps \* 1 \* 83 ciclos

= 23.572 ps

La mejora del procesador segmentado frente al procesador multiciclo es la siguiente:

P(segmentado) =

El número de instrucciones de la segmentada son 83 debido a:

* + 4 instrucciones antes del bucle
  + 13 instrucciones del bucle ejecutadas 6 veces
  + 1 instrucción despues del bucle

P(multiciclo) =

N =

El procesador segmentado es 21,11 veces mejor que el procesador multiciclo.

# 5. Problemas durante la práctica

## 5.1 Instrucción Beq

Durante la implementación de la instrucción Beq tuvimos el siguiente problema, con el siguiente contador de programa:

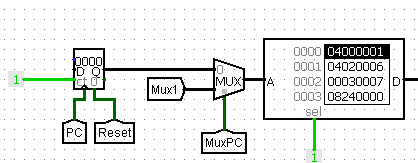


Ilustración . Contador de programa inicial.

Cuando la condición de la instrucción Beq se cumplía saltaba a la dirección destino ya que el control del multiplexor que decidía la siguiente instrucción a ejecutar (MuxPC) se ponía a 1 y la memoria de instrucciones era direccionada por el número inmediato de la instrucción Beq, pero en la siguiente instrucción si no era una Beq con una condición verdadera, el controlador de dicho multiplexor era 0 por lo cual se volvía a la dirección que había antes de que se ejecutara la beq por lo que se ejecutaba mal el programa.

Tras varios intentos decidimos eliminar el multiplexor y actuar con las señales de control del contador de programa en un contador. Tras realizar varias tablas de la verdad y varios intentos conseguimos obtener la solución:

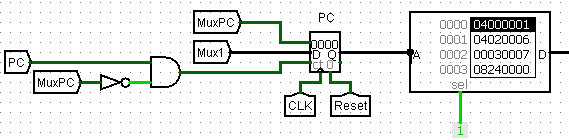


Ilustración : Solución del direccionamiento de la ROM para aguantar la instrucción BEQ

## 5.2 Memoria RAM

Un problema que no parecía tan grande a priori, que hemos encontrado en la realización de la práctica es que el programa logisim ofrece las descripciones de los pines incorrectas en las memorias RAM con puertos de carga y almacenamiento separados.

Si ponemos el cursor encima de cada pin, podemos ver que en el pin str aparece una descripción como mínimo extraña.

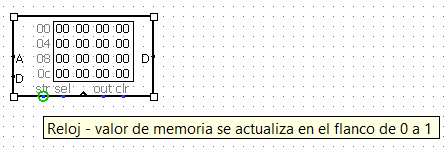


Ilustración . Descripción del pin str.

Si vamos al pin del reloj para contrastar, podemos ver que está en inglés. Lo que nos hizo sospechar que podría ser un error del programa.

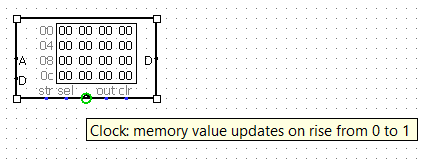


Ilustración . Descripción del pin reloj.

La descripción que salía del pin out es la siguiente:

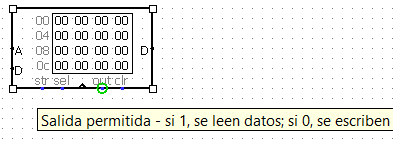


Ilustración . Descripción del pin out.

Por lo que decidimos implementar la memoria RAM de la siguiente forma.

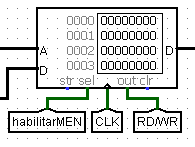


Ilustración . Memoria RAM incorrecta.

Las señales de control se activarían cuando se tuviera que hacer uso de la memoria RAM y según las descripciones no habría ningún problema con no usar str.

El programa hacía lo que tenía que hacer con la diferencia de que los dígitos que leía de la RAM los desplazaba uno a la derecha. Esto es debido a que utilizábamos un registro para cargar cada dato. El programa iba con ese registro (cuando el banco de registros estaba todo a 0) y escribía ese dato a la vez que sacaba al registro de fuera de la RAM el dato que estaba antes de escribir. Por lo que en el siguiente ciclo escribía en el mismo registro en el banco de registros para posteriormente escribirlo en la RAM cuando cargara el siguiente registro. Y así sucesivamente hasta que dejaran de ejecutarse las load en el código y perdiendo la última cifra en cada ejecución del programa.

Descubrir el problema seguramente fue lo más complicado. La solución pasa por hacer una tabla de la verdad con las entradas str out en la que pudimos ver lo siguiente:

* Si ambas están a cero ni lee ni cargan ningún dato
* Si str está a uno y out a cero escribirá en el banco de registros, pero no devolverá nada
* Si str esta cero y out a uno devolverá el dato que ha leído.
* Si str esta a uno y out a uno escribirá en el banco de registros y devolverá el valor escrito (hemos utilizado esta opción ya que es más cómodo tal y como lo teníamos planteado y es más visual al verse en el registro rápidamente si las operaciones han sido las correctas, además no nos dará ningún problema porque ese valor será desechado con las señales de control posteriores)

El diseño final de nuestra memoria RAM es la siguiente:

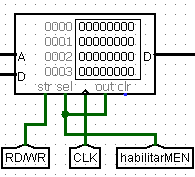


Ilustración . Memoria RAM correcta.

Cabe resaltar que RD/WR han sido modificadas en la unidad de control a efectos como si fuera una not de la anterior señal de control (al principio lo hicimos poniendo una not en esa señal para comprobar si funcionaba correctamente).

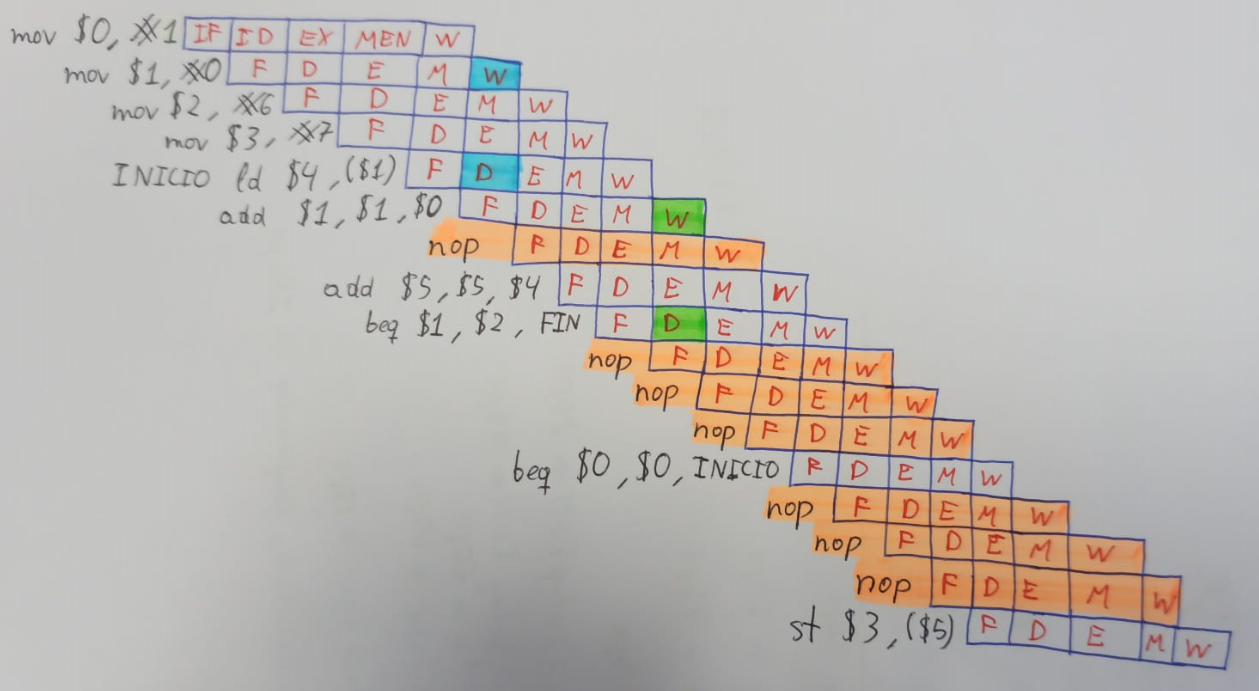
## 5.3 Banco de registros con lectura-escritura

Otro de los problemas que hemos tenido durante la práctica ha sido no saber si en nuestro banco de registros se puede leer y escribir en el mismo ciclo de reloj. Para solucionar este problema diseñamos el programa de dos formas:

* Suponiendo que el banco de registros lee y escribe en el mismo ciclo

Esta solución no fue la correcta ya que el banco no escribe y lee en el mismo ciclo.

Si el banco de registros escribiera y leyera en el mismo ciclo nos simplificaría mucho todo ya que no necesitaríamos muchas instrucciones nop como se puede comprobar en la siguiente ilustración:



* Suponiendo que el banco de registros lee y escribe en ciclos diferentes.

Actualmente no disponemos de un banco de registros que escriba y lea en el mismo ciclo por lo que debemos añadir varias instrucciones nop para que el programa funcione correctamente:

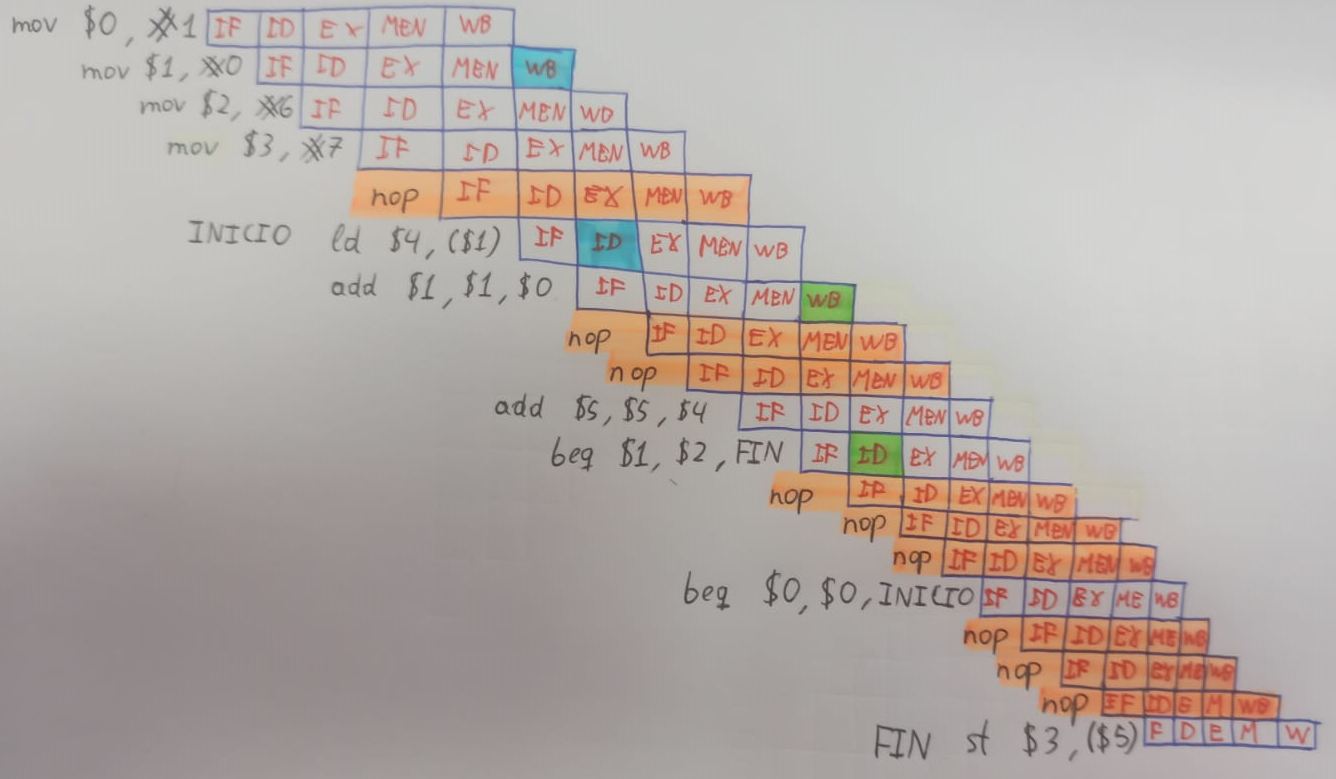


Ilustración :Ejecución del programa sin lectura-escritura en el mismo ciclo