2. **[2.5 ptos]** Escribe una **función** en ensamblador MIPS que dado un vector de números enteros devuelva la diferencia entre el valor máximo y el valor mínimo almacenados (máximo - mínimo).
La función ha de llamarse **dif**, y aceptará los siguientes parámetros de entrada:
   - Dirección del vector de datos enteros – se le pasará por el registro **$a0**
   - Tamaño del vector – se le pasará por el registro **$a1**

   El resultado lo tiene que devolver por el registro **$v0** (resultado = diferencia entre el valor máximo y valor mínimo almacenados en el vector de números cuya dirección se ha pasado por **$a0**).
   Recuerda que tienes que hacer una **función** y que debes seguir el convenio de llamadas a función.

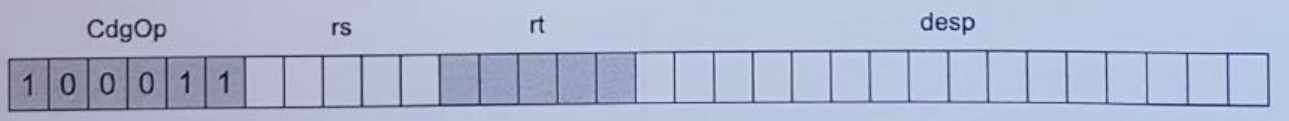
5. [2.5 ptos] La figura muestra una implementación MIPS monociclo que soporta el repertorio de instrucciones básico visto en las prácticas.

(a) Supóngase que se está ejecutando la instrucción **lw $7, 8($8)** (el funcionamiento y formato de la instrucción aparecen después de la tabla) y que conocemos los valores de las **direcciones** de entrada a la Memoria de Instrucciones (0X0000301C) y a la Memoria de Datos (0X000007AC). Responde y completa la siguiente tabla:

Buses /señales	Valor
Señal K3	
Señal K1	
Señal K9	
Valor contenido en registro $7	
Valor contenido en registro $8	
Bus SUM2	
Bus entrada a PC	
Flag Zero de la ALU	

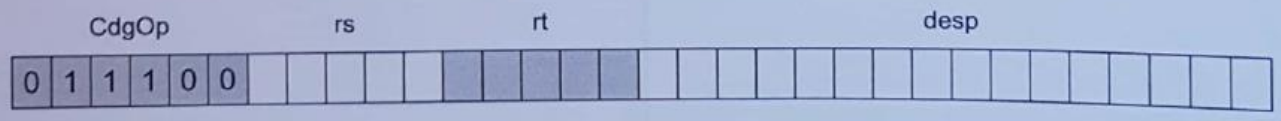
Caso que consideres que se desconocen alguno de los valores pedidos por faltar datos, debes indicarlo en la correspondiente casilla de la tabla escribiendo FD (faltan datos).

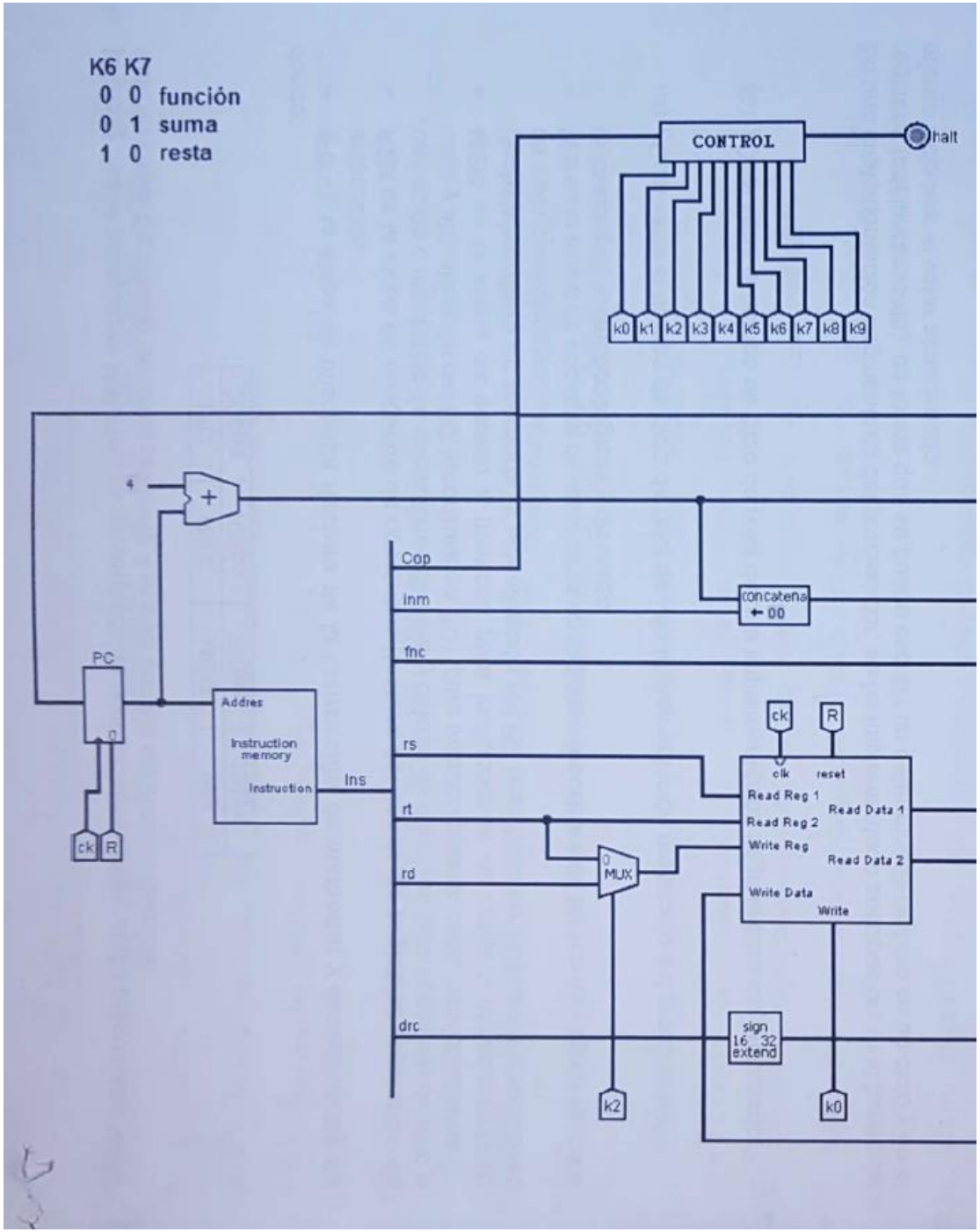
lw rt, desp(rs) **rt ← MEM(rs + ExtSig(desp))**

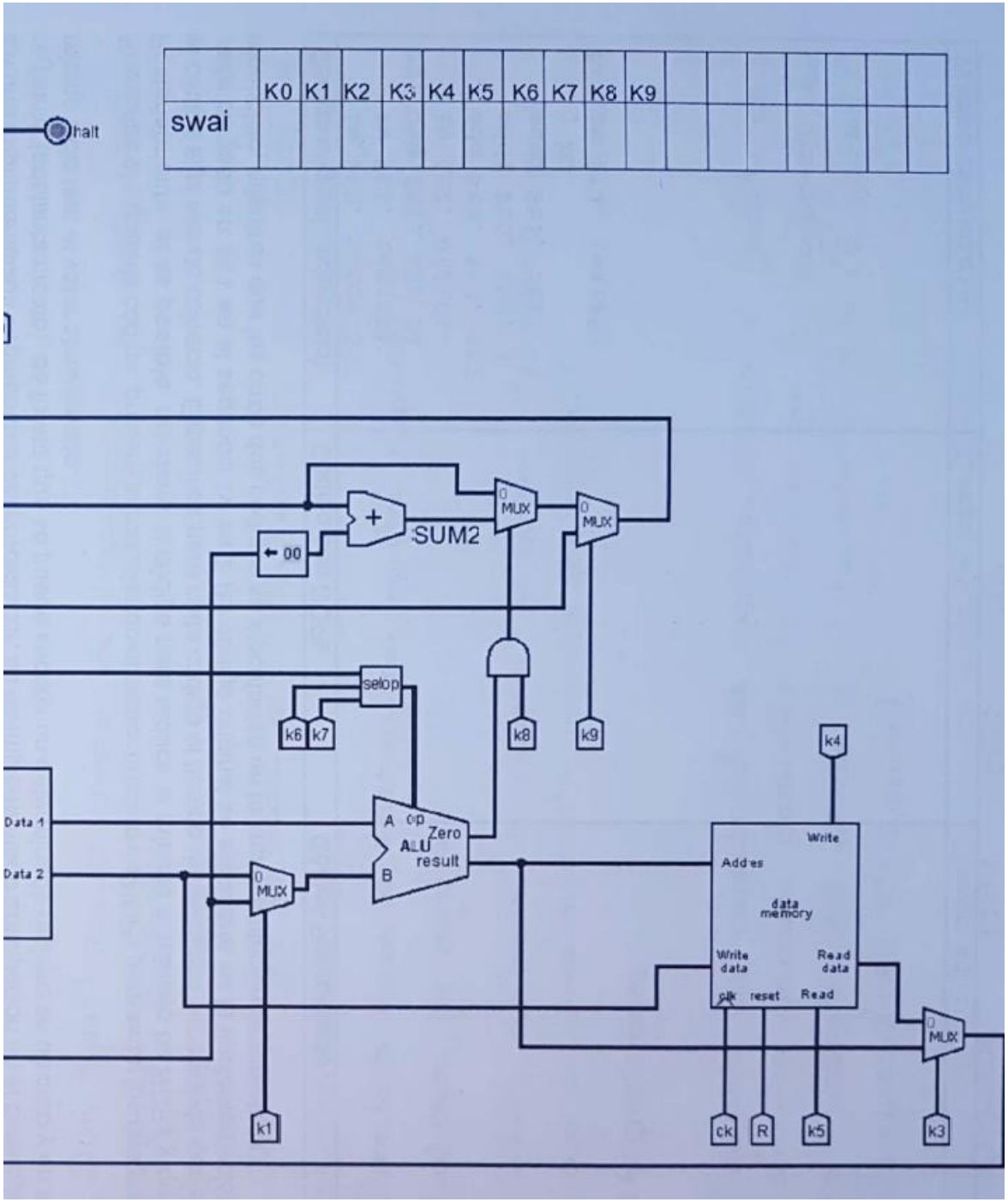


(b) Modifica el diseño del procesador para que soporte la siguiente instrucción. Dibuja sobre el esquema las modificaciones, elementos hardware y nuevos puntos de control necesarios en la vía de datos. Especifica los valores de todos los puntos de control para la nueva instrucción.

swai rt, desp(rs) **MEM(rs + ExtSig(desp)) ← rt**
rs ← rs + 4




K6 K7
0 0 función
0 1 suma
1 0 resta
CONTROL
halt
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9
4
+
Cop
inm
concatena
← 00
fnc
PC
Addres
Instruction memory
Instruction
Ins
ck R
rs
rt
rd
drc
0
MUX
ck R
clk reset
Read Reg 1
Read Reg 2
Write Reg
Write Data
Read Data 1
Read Data 2
Write
sign
16 32
extend
k2
k0


halt
K0 K1 K2 K3 K4 K5 K6 K7 K8 K9
swai
SUM2
00
MUX
MUX
selop
k6
k7
k8
k9
k4
Data 1
Data 2
A
op
Zero
ALU
result
B
MUX
Write
Addres
data
memory
Write
data
Read
data
clk
reset
Read
MUX
k1
ck
R
k5
k3

(5) La figura muestra una implementación MIPS monociclo que soporta el repertorio de instrucciones básico visto en las prácticas

a) Supóngase que se está ejecutando la instrucción lw $7, 8($8) (el funcionamiento y formato de la instrucción aparecen después de la tabla) y que conocemos los valores de las direcciones de entrada a la Memoria de Instrucciones (0x00003010) y a la Memoria de Datos (0x000007AC). Responde y completa la siguiente tabla

Buses/Señales	Valor
Señal K3	
Señal K1	
Señal K9	
Valor contenido en registro $7	
Valor contenido en registro $8	
Bus SUM2	
Bus entrada a PC	
Flag Zero de la ALU	

* Caso que consideres que se desconocen algunos de los valores pedidos por faltar datos, debes indicarlo en la correspondiente casilla de la tabla escribiendo FD (faltan datos). *

lw rt, desp(rs) rt ← MEM(rs + ExtSig(desp))

OPCODE | rs | rt | desp.

3 ✓

6. [2 ptos] Sea el procesador MIPS16 con un conjunto de instrucciones similar al del procesador MIPS visto en clase. Su camino de datos presenta 4 etapas con las siguientes latencias:

IFD	REX	MEM	WB
200ps	100ps	250ps	125ps

donde:

- **IFD** es la etapa de búsqueda (**lectura de la instrucción de memoria**) y decodificación de instrucción.
- **REX** es la etapa de búsqueda de operandos (**lectura del banco de registros**), ejecución de operación o resolución de condiciones de salto y cálculo de dirección para operando destino o salto y actualización del PC (**escritura en PC**) para instrucciones de salto incondicionales.
- **MEM** es la etapa de acceso a memoria para instrucciones de carga o almacenamiento (**lectura/escritura en memoria**) y actualización del PC (**escritura en PC**) para instrucciones de salto condicionales.
- **WB** es la etapa de escritura de resultados en registros (**escritura en banco de registros**) para instrucciones aritmético/lógicas y de carga.

(a) ¿Cuál sería el tiempo de ciclo de reloj de una implementación monociclo del procesador?

(b) ¿Cuál sería el tiempo de ciclo de reloj de una implementación segmentada del procesador?

a) ¿Cuál sería el tiempo de ciclo de reloj de una implementación monociclo del procesador?

$TC_{mon} = (200ps + 100ps + 250ps + 125ps) = 675ps$

b) ¿Cuál sería el tiempo de ciclo de reloj de una implementación segmentada del procesador?

$TC_{seg} = MAX(200ps, 100ps, 250ps, 125ps) = 250ps$

4 En una implementación segmentada del procesador, se ha implementado anticipación en el banco de registros (adelantamiento), de forma que se puede escribir un determinado registro en un ciclo y en el mismo ciclo leer el valor almacenado.

Reescribe el siguiente código, primero introduciendo el menor número de NOPs para evitar los riesgos. Posteriormente, **si es posible**, reordena el código para reducir al mínimo el número de NOPs y que el código siga siendo correcto. Especifica para cada código el tiempo de ejecución suponiendo que el valor cargado en `$t1` en el segundo `lw` es 2 (El código original se ejecutaría en la implementación monociclo, mientras que los otros dos códigos se ejecutarían en la implementación segmentada).

Código original (monociclo)	Código con NOPs	Código Reordenado

`lw $t0, datos($0)`
`lw $t1, tam($0)`
`bu: beq $t1, $0, fi`
`lw $t2, 0($t0)`
`add $t3, $t3, $t2`
`addi $t0, $t0, 4`
`addi $t1, $t1, -1`
`j bu`
`fi: sw $t3, res($0)`		

Tiempo monociclo:	Tiempo ej.:	Tiempo ej.:

Código Original (wounded)	Código con NOP's	Código Reordenado
lw $t0, datos($0) 1	lw $t0, datos($0)	lw $t1, tam ($0)
lw $t1, tam($0) 2	lw $t1, tam($0)	lw $t0, datos ($0)
bu: beq $t1, $0, fi 3	nop	bu: beq $t1, $0, fi
lw $t2, 0($t0) 4	bu: beq $t1, $0, fi	nop
add $t3, $t3, $t2 5	nop	nop
addi $t0, $t0, 4 6	nop	lw $t2, 0($t0)
addi $t1, $t1, -1 7	lw $t2, 0($t0)	addi $t0, $t0, 4
j bu 8	nop	add $t3, $t3, $t2
fi: sw $t3, res($0) 9	add $t3, $t3, $t2	addi $t1, $t1, -1
	addi $t0, $t0, 4	j bu
	addi $t1, $t1, -1	nop
	nop	fi: sw $t3, res ($0)
	j bu	
	nop	
	fi: sw $t3, res ($0)	
Tiempo ejecución (2 + 6N + 1) × 675 = = 2025 + 4050N	Tiempo ejecución (3 + 11N + 1 + 3) × 250 = = 1750 + 2750N	Tiempo ejecución (2 + 9N + 1 + 3) × 250 = 1500 + 2250N

IFD	REX	MEM	WB	
	IFD	REX	MEM	WB
		IFD	REX	MEM	WB
			IFD	REX	MEM	WB

5

1.- Escribe una **función** en ensamblador MIPS que dado un vector **A** de números enteros y un número entero **x**, devuelva el número de elementos del vector que son mayores que **x**.
La función ha de llamarse **gtx**, y aceptará los siguientes parámetros de entrada:

- Dirección del vector de datos enteros **A** – se le pasará por el registro **$a0**
- Tamaño del vector **A** – se le pasará por el registro **$a1**
- Número entero **x** – se le pasará por el registro **$a2**

El resultado lo tiene que devolver por el registro **$v0** (resultado = número de elementos del vector que son mayores que **x**).
Recuerda que tienes que hacer una **función** y que debes seguir el convenio de llamadas a función.

6

3.- Sea el procesador MIPS64 con un conjunto de instrucciones similar al del procesador MIPS visto en clase. Su camino de datos presenta una segmentación en 4 etapas. Esto hace que las instrucciones se ejecuten según se muestra a continuación:

Instrucción	1	2	3	4	5	6	7
Instr. 1	IFD	REX	MEM	WB			
Instr. 2		IFD	REX	MEM	WB		
Instr. 3			IFD	REX	MEM	WB	

donde:

- **IFD** es la etapa de búsqueda (**lectura de la instrucción de memoria**) y decodificación de instrucción.
- **REX** es la etapa de búsqueda de operandos (**lectura del banco de registros**), ejecución de operación aritmético/lógica ó resolución de condiciones de salto y cálculo de dirección para operando destino ó salto y actualización del PC (**escritura en PC**) para instrucciones de salto incondicionales.
- **MEM** es la etapa de acceso a memoria para instrucciones de carga o almacenamiento (**lectura/escritura en memoria**) y actualización del PC (**escritura en PC**) para instrucciones de salto condicionales.
- **WB** es la etapa de escritura de resultados en registros (**escritura en banco de registros**) para instrucciones aritmético/lógicas y de carga.

7

No hay anticipación (adelantamiento) en el banco de registros, de forma que no se puede escribir un determinado registro en un ciclo y en el mismo ciclo leer el valor almacenado.
Reescribe el siguiente código, primero introduciendo el menor número de NOPs para evitar los riesgos (a). Posteriormente, si es posible, reordena el código para reducir el número de NOPs y que el código siga siendo correcto (b).

Código original	(a) Código con NOPs	(b) CódigoReordenado

`add $4, $3, $2`
`sub $6, $3, $2`
`loop: beq $5, $6, fin`
`lw $6, 10($5)`
`lw $7, 10($3)`
`addi $5, $5, 2`
`sub $10, $7, $6`
`sw $10, 10($6)`
`addi $6, $6, 2`
`j loop`
`fin: sw $7, 10($6)`
`subi $5, $7, 2`		

Código Original

```
      add $4,$3,$2
      sub $6,$3,$2
loop: beq $5,$6,fin
      lw $6,10($5)
      lw $7,10($3)
      addi $5,$5,2
      sub $10,$7,$6
      sw $10,10($6)
      addi $6,$6,2
      j loop
fin:  sw $7,10($6)
      subi $5,$7,2
```

2 - 3 ($6)
3 -> beq
5 - 7 ($7)
7 - 8 ($6)
9 - 3 ($6)
10 -> J

a) Código con NOP's

```
add $4,$3,$3
sub $6,$3,$2
nop
nop
beq: $5,$6,fin <- loop
nop
nop
lw $6,10($5)
lw $7,10($3)
nop
addi $5,$5,2
sub $10,$7,$6
nop
nop
sw $10,10($6)
addi $6,$6,2
j loop
nop
sw $7,10($6) <- fin
subi $5,$7,2
```

b) Código Reordenado

```
sub $6,$3,$2
add $4,$3,$2
nop
beq $5,$6,fin <- loop
nop
nop
lw $7,10($3)
lw $6,10($5)
addi $5,$5,2
nop
sub $10,$7,$6
addi $6,$6,2
nop
j loop
sw $10,10($6)
sw $7,10($6) <- fin
subi $5,$7,2
```

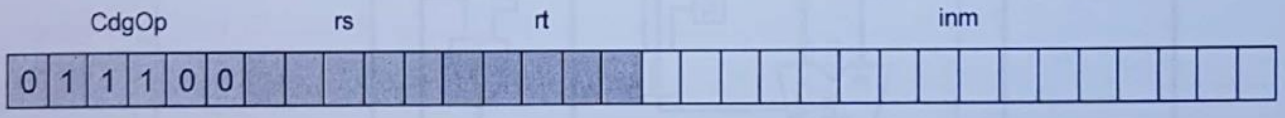
8

7.- La figura muestra una implementación MIPS monociclo que soporta el repertorio de instrucciones visto en las prácticas.

Modificad el diseño del procesador para que soporte las siguientes nuevas instrucciones. Dibujad sobre el esquema las modificaciones, elementos hardware y nuevos puntos de control, necesarios en la vía de datos. Especificad los valores de todos los puntos de control para cada instrucción. (inm* = extensión de signo del campo inm)

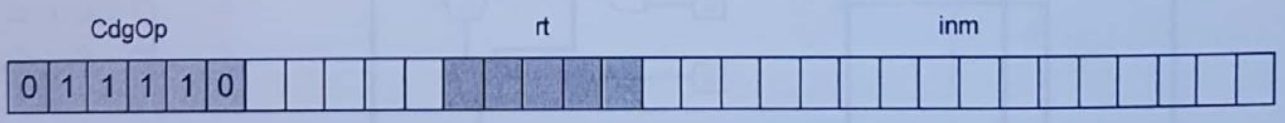
SET161 rs, rt, inm

If (MEM(rs) != 0) then PC ← rt
else PC ← PC+4+(inm*<<2)



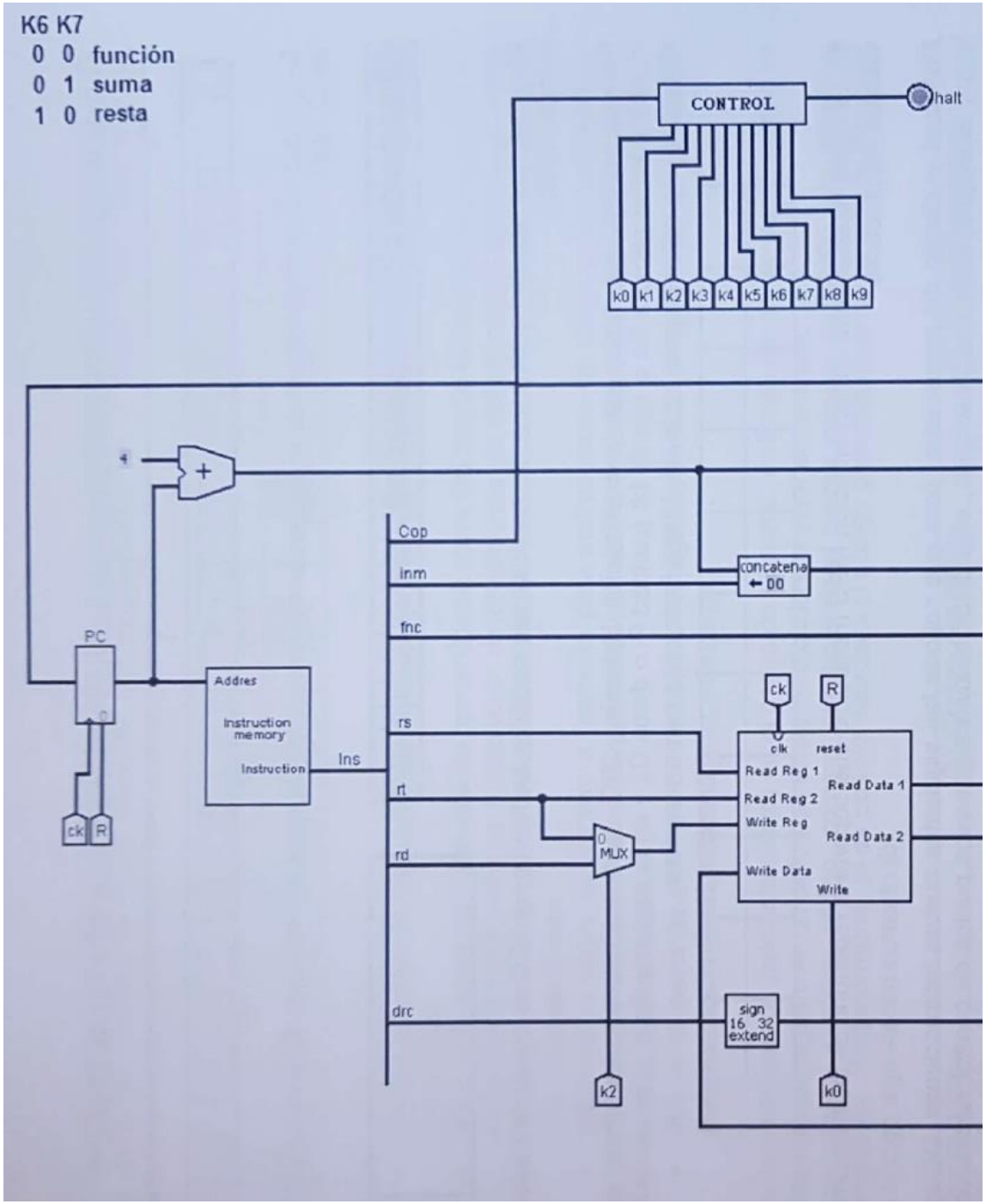
SET162 rt, inm

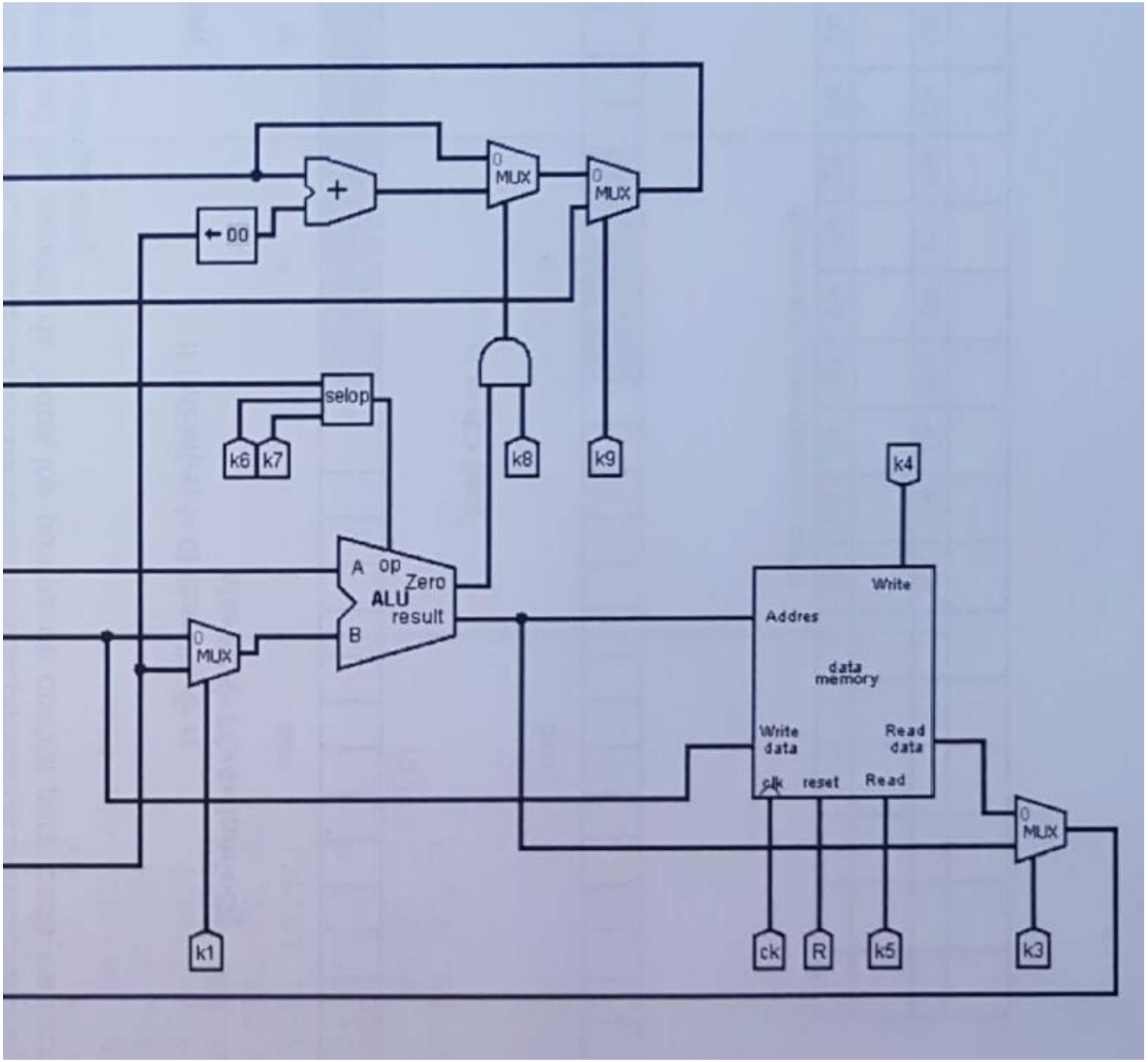
rt <- rt + inm*



Valores de los puntos de control

	K0	K1	K2	K3	K4	K5	K6	K7	K8	K9						
SET161																
	K0	K1	K2	K3	K4	K5	K6	K7	K8	K9						
SET162																


K6 K7
0 0 función
0 1 suma
1 0 resta
CONTROL
halt
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9
4
+
Cop
inm
concatena
← 00
fnc
PC
Addres
Instruction memory
Instruction
Ins
ck R
rs
rt
rd
drc
0
MUX
ck R
clk reset
Read Reg 1
Read Reg 2
Write Reg
Write Data
Read Data 1
Read Data 2
Write
sign
16 32
extend
k2
k0


+
00
MUX
MUX
selop
k6
k7
k8
k9
k4
A
op
Zero
ALU
result
B
MUX
Write
Addres
data
memory
Write
data
Read
data
clk
reset
Read
MUX
k1
ck
R
k5
k3

1. (H.P. 4.12) Supongamos que las etapas del camino de datos del procesador visto en clase tienen las siguientes latencias:

IF	ID	EX	MEM	WB
200ps	150ps	120ps	190ps	140ps

Se pide:

a) ¿Cuál es el ciclo de reloj si el procesador se decide implementar en una versión monociclo? ¿Y si fuera segmentado?

b) ¿Cuál es la latencia de una instrucción `lw` en un procesador segmentado y en uno monociclo?

c) Si se divide una etapa del camino de datos segmentado en dos nuevas etapas, cada una con una latencia mitad de la etapa original, ¿qué etapa se debería dividir y cuál sería el nuevo ciclo de reloj del procesador?

2. (H.P. 4.13) Sean las siguientes secuencias de instrucciones:

```
a.   lw  $1, 40($6)          b.   lw  $5, -16($5)
     add $6, $2, $2               sw  $5, -16($5)
     sw  $6, 50($1)               add $5, $5, $5
```

a) Indica las dependencias y su tipo.

b) Indica los riesgos y añade instrucciones `nop` para resoverlos.

3. (H.P. 4.15) Sean las nuevas instrucciones:

`bezi (rt), desp`	opec $\|r0\|rt\|desp$	If Mem[rt]=0 then PC ← PC+4+desp
`swi rd, rs(rt)`	opec $\|rs\|rt\|rd\|xxxx$	Mem[rs+rt] ← rd

a) ¿Qué cambios hay que hacer en el camino de datos para añadir estas instrucciones a la ISA del MIPS?

b) ¿Qué nuevas señales de control deben añadirse al diseño?

c) ¿Qué tipo de riesgos se pueden producir con estas instrucciones?

4. (H.P. 4.16) Para cada una de las siguientes instrucciones:

```
a.   lw  $1, 40($6)
b.   add $5, $5, $5
```

a) Identifica qué se almacena en cada uno de los registros situados entre dos etapas del pipeline.

b) ¿Qué registros necesitan leerse y cuáles se leen realmente?

c) ¿Qué hace la instrucción en las etapas EX y MEM?

5. Sea el programa `suma1` dado por el siguiente código MIPS:

```
        sub  $5, $0, $0
suma:   lw   $10, 1000($20)
        add  $5, $5, $10
        addi $20, $20, -4
        bne  $20, $0, suma
```

Se pide:

a) Describir brevemente la tarea realizada por `suma1`.

b) Detectar las dependencias que afectan a `suma1` en el MIPS segmentado en 5 etapas y clasificarlas en dependencias de datos y dependencias de control.

c) Gestión de dependencias por parte del compilador:

c.1 Suponer un MIPS segmentado sin ningún tipo de soporte hardware para solventar los problemas derivados de los riesgos de datos que conlleva la segmentación. Reescribir el código reordenándolo e insertando el el mínimo número de códigos de no-operación (nop) para producir una nueva versión, `suma2`, que pueda ejecutarse correctamente en este MIPS.

c.2 Evaluar la mejora obtenida hasta ahora con respecto al punto de partida. Para ello, comparar el tiempo de ejecución que ofrece `suma2` al ejecutarse sobre el MIPS anterior con respecto al de `suma1` sobre el MIPS sin segmentar de la implementación monociclo. Suponer un número `N` de iteraciones en ambos programas y que las latencias de las etapas del camino de datos del procesador son las vistas en clase:

IF	ID	EX	MEM	WB
40ps	20ps	40ps	40ps	20ps

6. Sea una arquitectura RISC con un conjunto de instrucciones similar al del procesador MIPS visto en clase. Su camino de datos presenta una segmentación en 3 etapas y una única memoria, común para instrucciones y datos. Esto hace que las instrucciones se ejecuten según se muestra a continuación:

INST	1	2	3	4	5
Inst1	IFD	REX	MEW		
Inst2		IFD	REX	MEW	
Inst3			IFD	REX	MEW

donde:

- `IFD` es la etapa de búsqueda y decodificación de instrucción.
- `REX` es la etapa de búsqueda de operandos (lectura del banco de registros), ejecución de operación ó resolución de condiciones de salto y cálculo de dirección para operando destino ó salto.
- `MEW` es la etapa de acceso a memoria para instrucciones de carga/almacenamiento, escritura de resultados en registros para instrucciones aritmético/lógicas y de carga y actualización del PC para instrucciones de salto.

Además no podrá leerse en un mismo ciclo un registro que va a ser escrito en dicho ciclo (es decir, no hay anticipación en el banco de registros).

Bajo estas condiciones, responder a las siguientes cuestiones:

a) ¿Qué tipo de riesgos pueden darse? Poner ejemplos.

b) ¿Qué ocurriría con las instrucciones de salto?

c) Representar en un diagrama de ciclos la evolución del cauce para el siguiente trozo de código e identifica los riesgos que puedan darse.

```
        sw  $5, 3000($7)
        or  $5, $4, $5
        sub $8, $4, $7
        lw  $4, 3000($8)
        bne $7, $0, etiq
        add $8, $8, $5
etiq:   sw  $5, 3000($8)
```

7. Sea el siguiente código MIPS, que a partir de ahora referenciaremos como `mi_prog`:

```
      ori  $2, $0, 1000h
loop: lw   $1, 2800h($2)
      sub  $4, $1, $0          rotar: add  $10, $4, $4
      jal  rotar                      muli $7, $10, 2
      sw   $7, 7800h($2)              jr $31
      sw   $1, C800h($2)
      subi $2, $2, 4
      bne  $2, $0, loop
```

Señalar las instrucciones que **pueden producir riesgos** en `mi_prog` para un MIPS segmentado con anticipación en el banco de registros. Responder en las siguientes tablas:

Riesgos de datos

Instrucción(es) (dar mnemotécnicos)	Registro(s) involucrado(s)

Riesgos de control

Instrucción(es) (dar mnemotécnicos)

Rel T3 29/5/19

1) a) $TC_{monociclo} = \sum etapas = 800\ ps.$

$TC_{segment} = max(etapas) = 200\ ps.$

b) Latencia monoc (LW) = 800 (1 cc)

" seg (LW) = $5 \times \underset{cc}{200} = 1000\ ps.$

IF1	IF2	EX	MEM	WB	
100	100	120	190	140	⇒ new CC = max(etapas) = 190 ps

2) a) 1. lw \$1, 40 (\$6)
2. add \$6, \$2, \$2
3. sw \$6, 50 (\$1)

pueden dar riesgos

Dependencias

Dep. datos {
Dep. verdadera → lw - sw (\$1), add - sw (\$6)
Antidep → lw - add (\$6)
Dep. salida →
}

Dep. control → X (pq no hay saltos)

podría riesgos de datos que no están controlados

	C1	C2 (\$6)	C3	C4	C5				
lw	IF	ID	EX	MEM	WB (\$1)				
add		IF	ID	EX	MEM	WB (\$6)			
~~sw~~			~~IF~~	~~ID (\$6 \$1)~~					
nop			IF	ID	EX	MEM	WB		
nop				IF	ID	EX (\$6 \$1)	MEM	WB	
sw					IF	ID	EX	MEM	WB

Esto sería correcto si tengo anticipación de registro, sino tengo habría que poner 3 nop's en lugar de 2.

b) 1. lw \$5, -16(\$5)
2. sw \$5, -16(\$5)
3. add \$5, \$5, \$5

Dep. datos {
Dep. verdadera → lw - sw (\$5), lw - add (\$5)
Antidep → lw - add (\$5), lw - sw (pero menor), sw - add (\$5)
Dep. salida → lw - add (\$5).
}

Dep. control → X

	C1	C2	C3	C4	C5	C6	C7
lw	IF	ID $5	EX	MEM	WB $5		
nop		IF	ID	EX	MEM	WB	
nop			IF	ID	EX $5	MEM	WB
sw				IF	ID $5	EX	MEM WB
add					IF	ID $5	EX MEM WB

suponiendo que tengo anticipación de registro.

5)
```
      sub $5, $0, $0
suma: lw  $10, 1000($20)
      add $5, $5, $10
      addi $20, $20, -4
      bne $20, $0, suma
```

a) Sumando el de un vector que está en la pos 1000($20), en adelante.

b) Dep verdaderas: sub - add ($5), lw - add ($10), add - addi ($5)
Dep. control: bne - lw. // addi - bne ($20), addi - addi ($20)
addi - lw ($20)

lee contenido registro.

escribe en fase WB registro d dato leído de memoria

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	...
sub	IF	ID	EX	MEM	WB $5						
lw		IF	ID $20	EX	MEM	WB					
nop			IF	ID	EX	MEM	WB				
nop				IF	ID	EX	MEM	WB			
nop					IF	ID	EX	MEM	WB		
add						IF	ID	EX	MEM	WB	
addi							IF	ID	EX	MEM	WB
nop								IF	ID	EX	MEM
nop									IF	ID	EX
nop										IF	ID
bne											IF
nop											
nop											
nop											

7 pc y siguiente instrucción.

Reordenando

sub
lw
addi
nop
nop
add
bne
nop
nop
nop

(dibujar diagrama para verlo mejor)

6) a) Riesgos de control ⟶ Hay instrucc. de salto, así que pueden existir.
Riesgos de datos → por. ej C3 se puede leer y escrib. en el mismo reg.
Riesgos estructurales

C1	C2	C3	
IFD	REX	MEW	
	IFD	REX	MEW
		IFD	REX, MEW

→ Si, por ej, en el C3 accedo a la misma memoria

b) Pues que me van a generar riesgo, entonces habría que introducir 2 nops

C1; C2 ...

c)

sw	IFD	REX	MEW				
or		IFD	REX	MEW ($5)			
sub			IFD	REX ($4, $8)	MEW ($8)		
lw				IFD	REX ($8)	MEW	
bne					IFD	REX ($1, $2)	MEW ($8, $5)
add						IFD	REX	MEW ($8, $5)
sw							IFD	REX	MEW

→ no hay riesgo
→ quiero leer y escribir al mismo tiempo en $8
→ Riesgo control.
→ Riesgo ($8)

(riesgo a nivel [illegible])

7) Riesgos control (saltos)

```
jal noter
jr $31
bne $2, $0, loop
```

Riesgos de datos

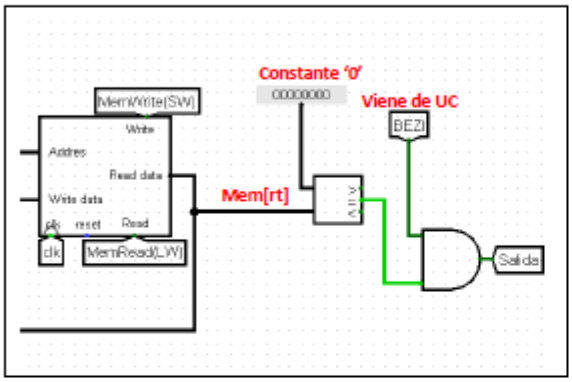
ori	IF	ID	EX	MEM	WB ($2)						
lw		IF	ID ($2)	EX	MEM	WB ($1)					
sub			IF	ID ($2, $0)	EX	MEM	WB ($4)				
jal				IF	ID	EX	MEM	WB			
add					IF	ID ($4)	EX	MEM	WB ($10)		
muli						IF	ID	EX	MEM	WB	
jr							IF	ID	EX	MEM	WB
sw											
subi											
bne											

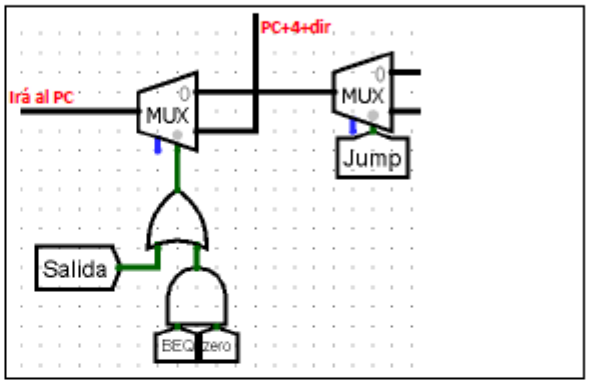
Riesgo datos

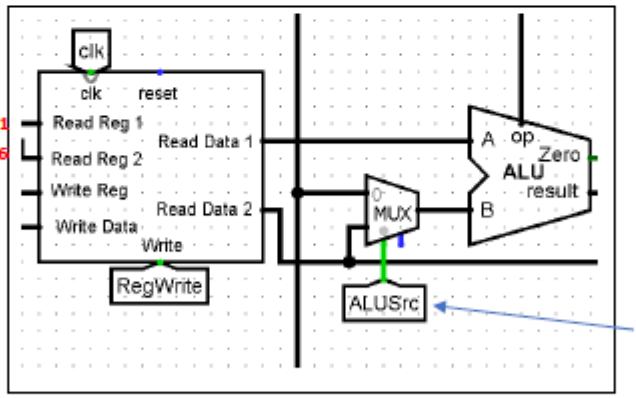
ori - lw ($2)
lw - sub ($1)
add - sub ($4) (jal)
add - muli ($10)
sw - muli ($7)
bne - subi ($2)
lw - subi ($2)

Ejercicio 3

A y B) Para la instrucción **bezi** tendríamos que diseñar la UC (nos dará una salida llamada "bezi"), luego añadir un **comparador** que compare una constante '0' con Mem[rt] y tras el comparador, utilizar una puerta lógica **AND** con la salida del comparador (salida '=') y con la salida "bezi" de la UC. Debemos tener en cuenta que el multiplexor cuya entrada tiene FPC0 tendrá que ser cambiada por una puerta **OR**. Por último, hay que recordar que hay que activar **ALUSrc** desde la UC

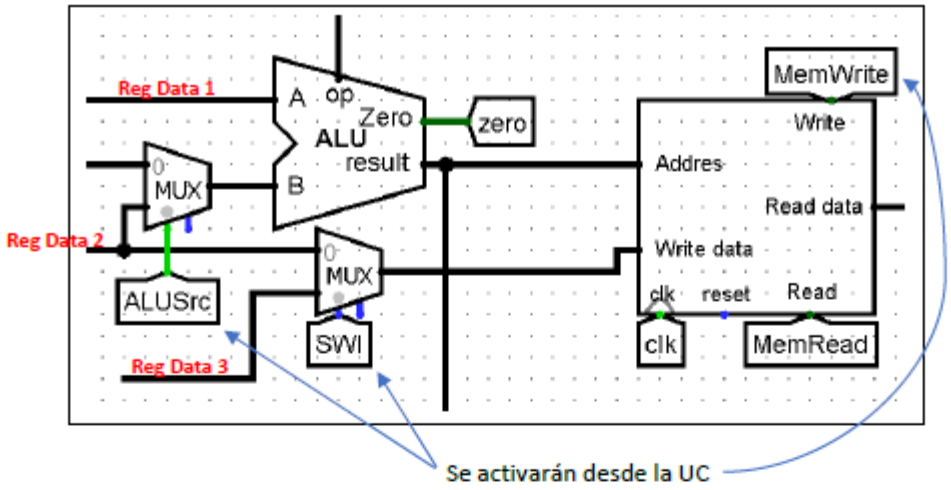






Nota: Los circuitos que aparecen solo sirven como esquema ya que en ellos no se encuentran los registros que hay entre cada etapa del pipeline

Para la instrucción **swi** habría que añadir una entrada de lectura más (Read reg 3 y Read Data 3) para rd y un multiplexor que controlará correctamente la entrada de rd a la memoria. Finalmente habría que activar tanto **ALUSrc** como **MemWrite.** Hay que recordar que hay que incluir dicha instrucción en la **UC**.



c) -Se puede producir riesgo de datos ya que swi escribirá dentro de una memoria
-Se puede producir riesgo de control ya con bezi, la condición de salto se conocerá en la etapa WB

Ejercicio 5

a) Básicamente lo que realiza el programa es sumar los datos que se encuentran desde la posición de memoria M[1000($20)] hasta M[1000] (cuando $20=0). Para ello guarda un valor temporal en el $10 de M[1000($20)] y luego realiza la suma con el $5(que es el registro donde estará el resultado de las sumas anteriores) y lo guarda en el $5. Finalmente, se le resta 4 al $20(la dirección del vector) y se compara $20 con el valor 0. Este proceso se repetirá mientas que $20 sea distinto de 0.

9 b)

```
        sub $5, $0, $0
suma:   lw $10, 1000($20)
        add $5, $5, $10
        addi $20, $20, -4
        bne $20, $0, suma
```

Dependencia de datos

-Dependencia verdadera: la instrucción **sub** produce un resultado que es usado por **add**

-Dependencia de verdadera: la instrucción **lw** produce un resultado que es usado por **add**

-Dependencia de verdadera: la instrucción **addi** produce un resultado que es usado por **bne**

-Dependencia de verdadera: la instrucción **addi** produce un resultado que es usado por **lw**

-Antidependencia: En **lw** lee el operando ($20) y en **addi** lo escribe

-Dependencia de salida: En **sub** y **add** escribimos en el mismo operando ($5)

Dependencia de control

-**bne:** Salto condicional

c)

C1.

```
        sub $5, $0, $0
suma:   lw $10, 1000($20)
        addi $20, $20, -4
        nop
        nop
        add $5, $5, $10
        bne $20, $0, suma
        nop
        nop
        nop
```

NOTA (para programar): Si debajo del bne se encuentran más instrucciones que no forman parte del programa es necesario poner nop's. En caso contrario no es necesario introducir nop's ya que en las siguientes posiciones (de la Memoria de instrucciones) después de la instrucción bne se encontrará el valor hexadecimal 00000000 (por defecto) que equivale a nop

C2.

```
        sub $5, $0, $0
suma:   lw $10, 1000($20)
        add $5, $5, $10
        addi $20, $20, -4
        bne $20, $0, suma
```

MIPS monociclo:

Latencia de una instrucción=40ps+20ps+40ps+40ps+20ps=160ps

1 instrucción

\+

4instrucciones + 4instrucciones*N interacciones del bucle

(5 instrucciones + 4instrucciones*N interacciones del bucle)) * 160ps

Resultado: 800ps+640ps*N interacciones del bucle

```
        sub $5, $0, $0
suma:   lw $10, 1000($20)
        addi $20, $20, -4
        nop
        nop
        add $5, $5, $10
        bne $20, $0, suma
        nop
        nop
        nop
```

MIPS segmentado:

Latencia de una instrucción=40ps*5ciclos=200ps

1 instrucción*200ps

\+

(9instrucciones + 9instrucciones*N interacciones del bucle) * 40ps

200ps+340ps+360ps*N interacciones del bucle

Resultado: 560ps+360ps*N interacciones del bucle

Ejercicio 6

10 a) Pueden darse **Riesgo de datos, Riesgo de control y Riesgo estructural:**
En el caso de **Riesgo de estructural y de datos** puede darse de la siguiente manera:

```
Sub $8, $4, $5
Add $9, $8, $8
```

INST	1	2	3
Sub	IFD	REX	MEW
Add		IFD	REX

Como se puede observar, en la instrucción sub escribe en el $8 mientras que en la 1º instrucción add lee el registro $8, generando un riesgo estructural (Acceso simultaneo a un registro) y de datos (RAW). Como no hay anticipación en el banco de registros, su solución será utilizar una instrucción nop entre sub y add.

En el caso de **Riego Estructural** puede darse de la siguiente manera:

```
sw $5,3000($7)
Or $5, $4, $5
Sub $8, $4, $7
```

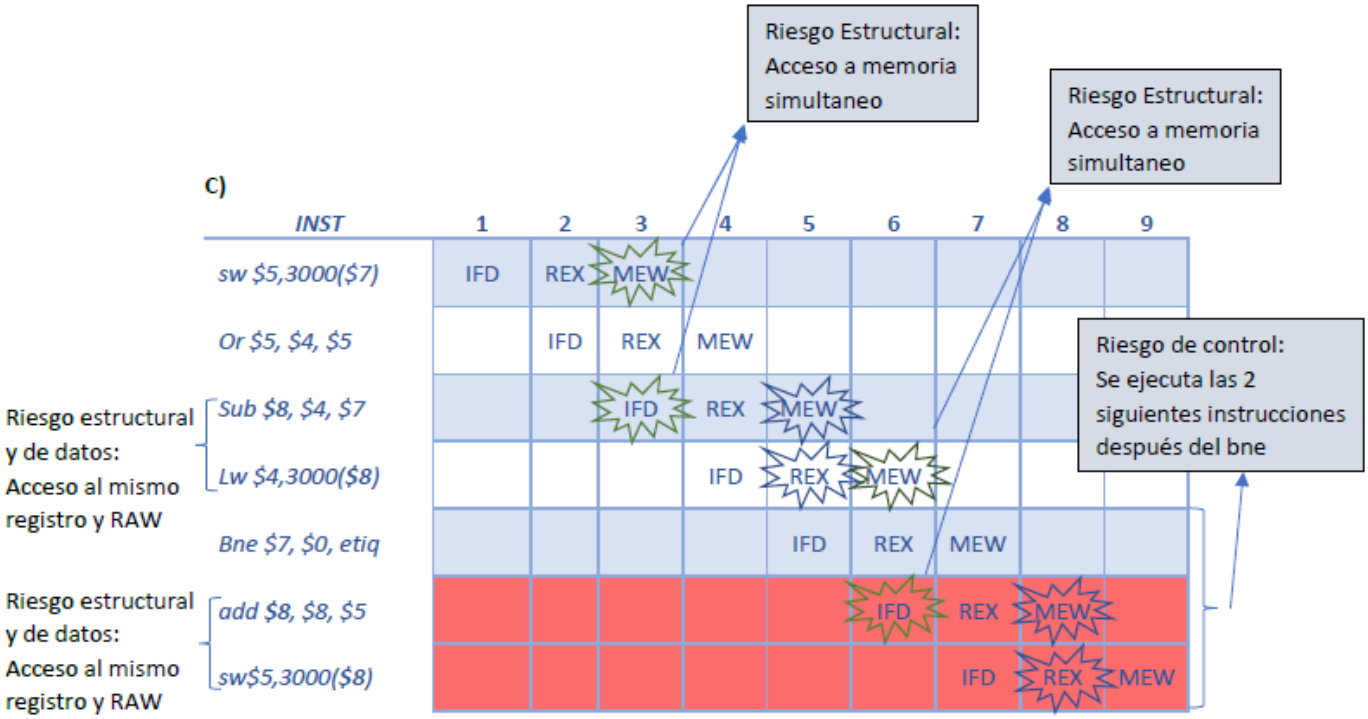
INST	1	2	3
sw	IFD	REX	MEW
or		IFD	REX
sub			IFD

Como se puede observar, en la instrucción sw se intenta guardar M[3000($7)]<-$5 mientras que a la vez se está leyendo una instrucción, provocando un riesgo estructural de acceso simultaneo en la memoria.

A y B) En el caso de **Riesgo de control**, puede darse de la siguiente manera:

```
Loop:   Sub $8, $4, $5
        J loop
        Add $9, $8, $8
        nop
        Add $9, $8, $8
```

Según el enunciado, las condiciones de salto se cargan y se actualizan en la etapa MEW, por lo que las siguientes 2 instrucciones después de un salto se ejecutarán. Su solución es utilizar 2 nop's(tanto para el **jump** como para el **beq**) después de una condición de salto.



Ejercicio 7

```
      ori  $2, $0, 1000h
loop: lw   $1, 2800h($2)
      sub  $4, $1, $0          rotar: add  $10, $4, $4
      jal  rotar                      muli $7, $10, 2
      sw   $7, 7800h($2)              jr $31
      sw   $1, C800h($2)
      subi $2, $2, 4
      bne  $2, $0, loop
```

Instrucción(es) (dar mnemotécnicos)	Registro(s) involucrado(s)
Ori-lw	$2
Lw-sub	$1
Sub-add	$4
Add-muli	$10
Muli-sw	$7
Subi-bne	$2
Subi-lw	$2

Instrucción(es) (dar mnemotécnicos)
jal
jr
bne

Ejercicio 1

IF	ID	EX	MEM	WB
200ps	150ps	120ps	190ps	140ps

a) Para la versión del monociclo, el ciclo del reloj será la suma de cada etapa:

T=200ps+150ps+120ps+190ps+140ps=800ps

Para el segmentado será el mayor de todos ellos:

T=200ps

b) Para el monociclo, la latencia será la suma de cada etapa:

Latencia=200ps+150ps+120ps+190ps+140ps=800ps

Para el segmentado, al ser 5 ciclos y el ciclo de reloj 200ps:

Latencia=5 ciclos*200ps=1000ps

c) Se debería dividir la etapa cuya latencia es la mayor de todas. En este caso, IF tiene la mayor latencia con 200ps. Si dividimos IF en dos etapas, "IF1" y "IF2" tendrían 100ps cada una y ahora la mayor latencia seria MEM con 190ps, por lo que el **nuevo ciclo de reloj sería de 190ps**

Ejercicio 2

a) **a.**

```
lw   $1, 40($6)
add  $6, $2, $2
sw   $6, 50($1)
```

Dependencia de datos

-Dependencia verdadera: la instrucción **lw** produce un resultado que es usado por **sw**

-Dependencia de verdadera: la instrucción **add** produce un resultado que es usado por **sw**

-Antidependencia: En **lw** lee el operando ($6) y en **add** lo escribe

b.

```
lw   $5, -16($5)
sw   $5, -16($5)
add  $5, $5, $5
```

Dependencia de datos

-Dependencia verdadera: la instrucción **lw** produce un resultado que es usado por **sw**

-Dependencia verdadera: la instrucción **lw** produce un resultado que es usado por **add**

-Antidependencia: En **lw** lee el operando ($5) y en **add** lo escribe

-Antidependencia: En **sw** lee el operando ($5) y en **add** lo escribe

-Dependencia de salida: En **lw** y **add** escribimos en el mismo operando ($5)

b) **a.**

-Hay riesgo de datos (RAW) entre lw y sw

-Hay riesgo de datos (RAW) entre add y sw

-Hay riesgos estructurales si usamos 2 nop(Acceso simultaneo al mismo registro). La solución será realizar la escritura en la primera mitad del ciclo y la lectura en la segunda mitad del ciclo.

```
lw   $1, 40($6)
add  $6, $2, $2
nop
nop
sw   $6, 50($1)
```

b.

-Hay riesgo de datos (RAW) entre lw y sw

-Hay riesgo de datos (RAW) entre lw y add

-Hay riesgos estructurales si usamos 2 nop (Acceso simultaneo al mismo registro). La solución será realizar la escritura en la primera mitad del ciclo y la lectura en la segunda mitad del ciclo.

```
lw   $5, -16($5)
nop
nop
sw   $5, -16($5)
add  $5, $5, $5
```

Ejercicio 4

a) **a.**

IF/ID: Obtenemos la instrucción de lw

ID/EX: Tenemos el valor del contenido de $6 y de $1(que no será usado). Además tenemos ExSig(40)

EX/MEM: Tenemos la suma del contenido de $6 y ExSig(40)

MEM/WB: Tenemos el contenido de la memoria de $6+ExSig(40)

b.

IF/ID: Obtenemos la instrucción de add

ID/EX: Tenemos dos veces el contenido de $5.

EX/MEM: Tenemos la suma del contenido de $5 + $5

MEM/WB: Seguimos teniendo la suma del contenido de $5+$5

b)

a. Se lee $1 y $6. Realmente solo necesita $6

b. Se leen los dos $5. Se necesitan ambos $5

c)

a.

-En EX lo que hacemos es sumar $6 con 40 (R[rs]+ExSig(desp)) para obtener la dirección de memoria que necesitamos leer

-En MEM leemos la posición de memoria cuya dirección la hemos calculado antes en EX

(M(R[rs]+ExSig(desp)))

b.

-En EX sumamos el contenido de los dos registros ($5+$5)

-En MEM no hacemos nada, el valor pasa directamente a la siguiente etapa (WB)

Segundo Parcial 2017

① Para un procesador MIPS monociclo las latencias asociadas a las distintas etapas del ciclo de instrucción son las que se muestran en la siguiente tabla:

IF	ID	EX	MEM	WB
200ps	150ps	125ps	250ps	125ps

a) ¿Cuál sería el tiempo de ciclo de reloj en una implementación monociclo del procesador?

$$TC_{mon} = (200ps + 150ps + 125ps + 250ps + 125ps) = 850ps.$$

b) ¿Cuál sería el tiempo de ciclo de reloj si se segmenta el cauce del procesador en esas 5 etapas?

$$TC_{seg} = Máx(200ps, 150ps, 125ps, 250ps, 125ps) = 250ps.$$

En la implementación segmentada del procesador, no se ha implementado anticipación en el banco de registros (adelantamiento). Además, el HW del procesador se ha modificado de tal manera que la terminación (= carga del PC) tanto del salto condicional como del incondicional se realiza en la etapa ID.

c) Para el siguiente código, que se ha numerado para vuestra conveniencia, marca las dependencias que dan lugar a riesgos en el cauce segmentado descrito con anterioridad, indicando de qué tipo son y el registro causante.

```
1        add $6, $0, $0
2        add $10, $0, $0
3   et : lw $12, 48($10)
4        lw $13, 20($10)
5        sub $14, $13, $12
6        add $6, $6, $14
7        addi $10, $10, 4
8        bne $10, $15, et
9        sw $8, 8($0)
```

Dependencias que dan lugar a riesgos:

2-3	7-3
2-4	7-4
3-5	7-8
4-5	bne
5-6	

d) Reescribe el código, primero introduciendo el mayor número de NOP's para evitar los riegos. Posteriormente, si es posible, reordena el código para reducir al mínimo el número de NOP's y que la ejecución del código siga siendo correcta. Especifica para cada código el tiempo de ejecución suponiendo que se realiza un número Niter de iteraciones. (el código original se ejecutaría en la implementación monociclo, mientras que los otros dos códigos se ejecutarían en la implementación segmentada).

Código Original	Código con NOP's	Código Reordenado
add $6, $0, $0	add $6, $0, $0	add $10, $0, $0
add $10, $0, $0	add $10, $0, $0	add $6, $0, $0
et: lw $12, 40($10)	nop	sw $8, 8($0)
lw $13, 20($10)	nop	nop
sub $14, $13, $12	nop	et: lw $12, 48($10)
add $6, $6, $14	et: lw $12, 48($10)	lw $13, 20($10)
addi $10, $10, 4	lw $13, 20($10)	addi $10, $10, 4
bne $10, $15, et	nop	nop
sw $8, 8($0)	nop	nop
	nop	sub $14, $13, $12
2 - 3	sub $14, $13, $12	nop
4 - 5	nop	nop
5 - 6	nop	bne $10, $15, et
7 - 8 - bue	add $6, $6, $14	add $6, $6, $14
8 - bue	addi $10, $10, 4	
8 - bue - 3	nop } se	
	nop } van	
	nop }	
	bne $10, $15, et	
	nop	
	sw $8, 8($0)	
Tiempo Monociclo	Tiempo ejecución	Tiempo ejecución
(2 + 6N + 1) * 850 =	(5 + 16N + 1 + 4) * 250 =	(4 + 9N + 1 - 4) * 250 =
= 2550 + 5100N	= 2500 + 4000N	= 2250 + 2250N

② Dado un número de 4 instrucciones para un procesador de tamaño de palabra de 8 bits, dos registros R0 y R1 y una memoria de 64 bytes

Instrucción	Acción	Formato de Instrucción
ADI rd, rs, cte	rd ← rs + ExtSig(cte)	OPCODE (0 1) | rd | rs | cte
ADM rd, rs	rd ← rd + M(rs[5:0])	opcode (1 0) | rd | rs | X X X X
STM rd, rs	M(rs[5:0]) ← rd	OPCODE (1 1) | rd | rs | X X X X
BBQ dir	If R1 < R0, PC ← dir	OPCODE (0 0) | dir

donde cte es un entero representado en complemento a 2

a) Diseña una unidad de datos para poder ejecutar esas cuatro instrucciones, en base a los elementos hardware proporcionados. Además puedes utilizar elementos hardware adicionales que creas oportunos (registros, multiplexores, sumadores, comparadores, etc..) Indica claramente las partes de control necesarios

En la página 5

b) Construye la tabla de verdad donde se indique claramente qué señales de control deben activarse (1) y cuales no (0) para cada instrucción.

Tabla de control:

	Señales de control							
Instrucción	FPC	FB	FA	WBR	ALUB	ALUA	MDR	MDW
ADI	1	0	0	1	0	0	0	0
ADM	1	0	0	1	1	1	1	0
STM	1	0	1	0	1	0	0	1
BBQ	$\overline{\text{MSB}(2)}$	1	1	0	X	X	0	0

Si inicialmente el contenido de los registros es 0 (incluido el contador de programa, PC):

c) Describir la evolución del contenido de los registros y de la memoria (traza de ejecución) si se ejecutan 6 instrucciones

Memoria	
Pos	Contenido
0	52_{hex}
1	65_{hex}
2	$d0_{hex}$
3	90_{hex}
4	02_{hex}
5	59_{hex}
6	..
7	..
...	...

Traza de ejecución

PC	Instrucción (ej: ADI R1, R0, 3)	Registro o posición de memoria a escribir	Valor a escribir
0	ADI R0, R1, 2	R0	2
4	ADI R1, R0, 5	R1	7
8	STM R0, R1	MEM(7)	2
12	ADM R0, R1	R0,	4
16	BBQ 2	PC	7
20	ADI R0, R1, -7	R0	0

- $52_{HEX} \Rightarrow 0101\ 0010$

  ADI R0, R1, 2
  ADI R0 ← 0 + 2
  ADI R0 ← 2

- $65_{HEX} \Rightarrow 0110\ 0101$

  ADI R1, R0, 5
  ADI R1 ← R0 + 5
  ADI R1 ← 2 + 5
  ADI R1 ← 7

- $d0_{HEX} \Rightarrow 1101\ 0000$

  STM R0, R1
  MEM(R1) ← R0
  MEM(7) ← R0
  MEM(7) ← 2

- $90_{HEX} \Rightarrow 1001\ 0000$

  ADM R0, R1
  ADM R0 ← R0 + MEM(R1)
  ADM R0 ← 2 + 2
  ADM R0 ← 4

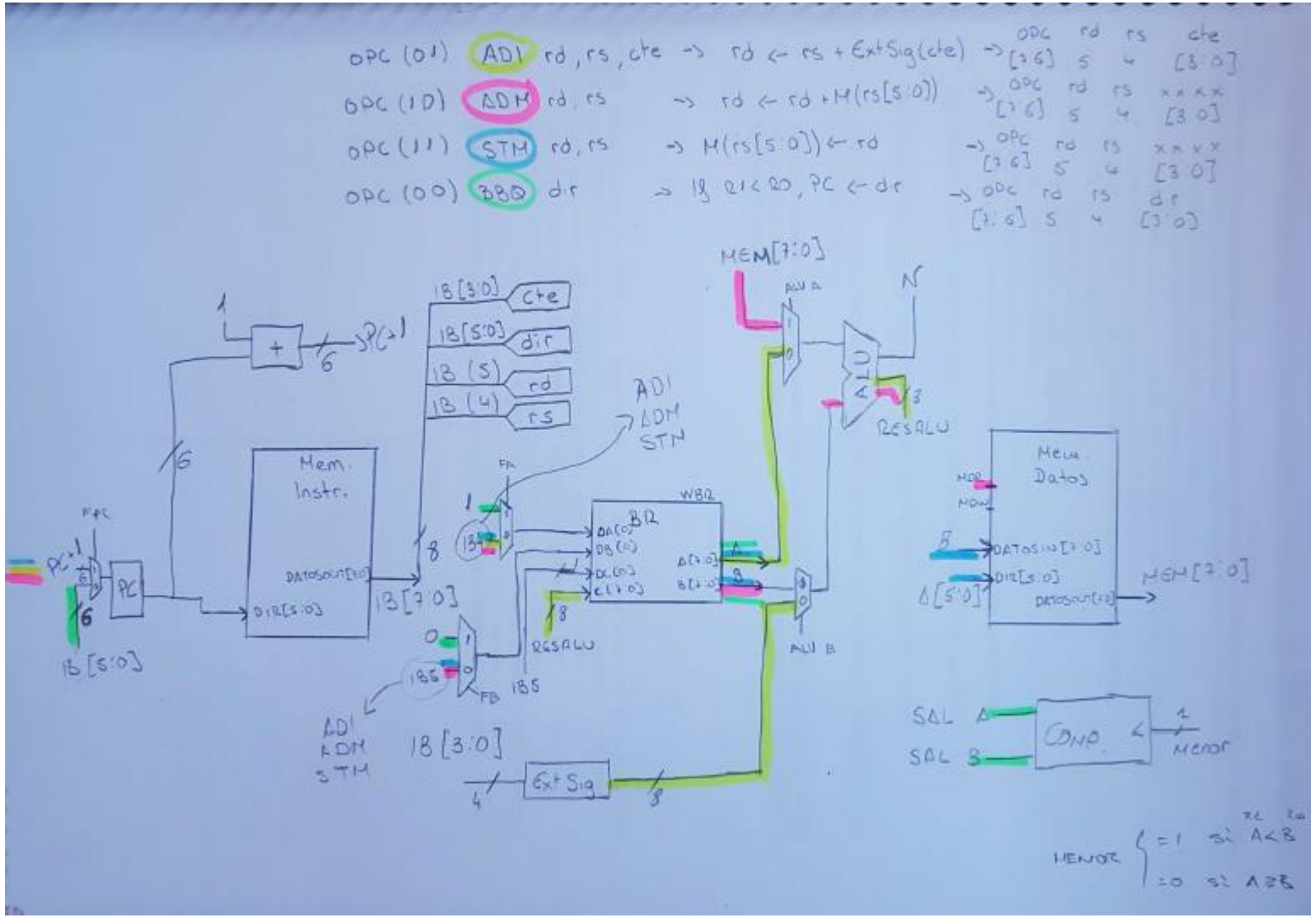
- $02_{HEX} \Rightarrow 0000\ 0010$

  BBQ 2
  If R1 < R0, PC ← dir
  7 ≮ 4

- $59_{HEX} \Rightarrow 0101\ 1001$

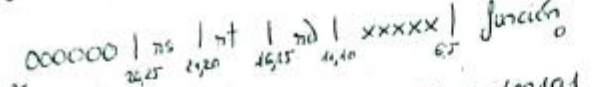
  ADI R0, R1, -7
  ADI R0 ← 7 - 7
  ADI R0 ← 0

  1001 → 0110 + 1 = 0111 → -7


OPC (01) ADI rd, rs, cte -> rd ← rs + ExtSig(cte)
OPC (10) ADM rd, rs -> rd ← rd + M(rs[5:0])
OPC (11) STM rd, rs -> M(rs[5:0]) ← rd
OPC (00) BBQ dir -> IB 21<20, PC ← dir
OPC rd rs cte
[7:6] 5 4 [3:0]
OPC rd rs xxxx
[7:6] 5 4 [3:0]
OPC rd rs xxxx
[7:6] 5 4 [3:0]
OPC rd rs dir
[7:6] 5 4 [3:0]
MEM[7:0]
ALU A
N
ALU
RESALU
IB[3:0] cte
IB[5:0] dir
IB (5) rd
IB (4) rs
ADI
ADM
STM
PC+1
PC
Mem. Instr.
DATOSOUT[7:0]
DIR[5:0]
IB[7:0]
IB[5:0]
FA
FB
IB5
RESALU
WBQ
BR
ALU B
Mem. Datos
MDR
MDW
DATOSIN[7:0]
DIR[5:0]
DATOSOUT[7:0]
MEM[7:0]
SAL A
SAL B
COMP
MENOR
MENOR =1 si A<B
=0 si A≥B

Ejercicios Computadores

1. Instr. ALU: OpAlu rd, rs, rt: R[rd] ← R[rs] (función) R[rt]



ADD → 100000 OR → 100101

SUB → 100010 SLT → 101010

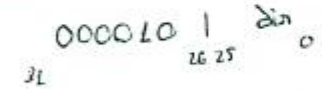
AND → 100100

Dada la siguiente instrucción Mips: Add $4, $8, $16

codifica en binario → 000000 01000 10000 00100 00000 100000

codificada en hexadecimal → 01102020

2. Instr. J: J dir: PC ← (PC+4)[31:28], dir, 00



Codifica en hexadecimal la siguiente instrucción Mips: J bucle

(sabemos que la instrucción J está en la dirección de memoria 2C, y que el valor asociado a la etiqueta bucle es 14. (la dirección efectiva de salto)

PC = 0000002C → 0000 0000 0000 0000 0000 0000 0010 1100

PC ← 0000 dir 00 = 14 → 00000014 = 0000 x6 0001 0100 <=>

<=> dir = 0000 x5 0001 01

Instrucción en binario → 0000 10 0000 0000 0000 0000 0000 0101

Instrucción en hexadecimal → 08000005

3. Instn. BEQ: Beq rs, rt, dir: if (R[rs] = R[rt]) PC ← (PC+4) + (ExSig(dir) << 2)

↓ mete dos 0's y se carga los 30 más significativos

000100	rs	rt	dir
31 26	25 21	20 16	15 0

Codifica en hexadecimal la siguiente instrucción Hirs: BEQ $8, $0, exit

(sabemos que la instrucción beq esta en la dirección de memoria 14, y que la etiqueta exit se corresponde con la dirección destino de salto 30.

PC = 14h

PC destino = 30h

PC ← (PC-4) + (Dir << 2) = (PC+4) + (Dir × 4)

30h ← (14h+4h) + Dir × 4 → DIR = (30h - 18h) / 4h = 18h / 4h = 6h

Instrucción en binario → 000100 01000 00000 0000 0000 0000 0110

Instrucción a hexadecimal → 11000006

4. Supongamos que las etapas del camino de datos del procesador visto en clase tienen las siguientes latencias:

IF	ID	EX	MEM	WB
200 ps	150 ps	120 ps	190 ps	140 ps

Se pide:

a) ¿Cuál es el ciclo de reloj si el procesador se decide implementar en una versión monociclo? ¿Y si fuera segmentado?

V. monociclo → 200+150+120+190+140 = 800 ps

V. segmentado → 200 ps

b) ¿Cuál es la latencia de una instrucción lw en un procesador segmentado y en uno monociclo?

P. segmentado → 200·5 = 1000 ps

P. monociclo → 800 ps

c) Si se divide una etapa del camino de datos segmentado en dos nuevas etapas, cada una con una latencia mitad de la etapa original, ¿qué etapa se debería dividir y cuál sería el nuevo ciclo de reloj del procesador?

Se debería dividir la etapa IF en dos de 100 y el nuevo ciclo de reloj sería 190 ps.

Sean las siguientes secuencias de instrucciones:

a)
```
lw  $1, 40($6)
add $6, $2, $2
sw  $6, 50($1)
```

b)
```
lw  $5, -16($5)
sw  $5, -16($5)
add $5, $5, $5
```

a) Indica las dependencias y su tipo.

a → { dependencias verdaderas: lw → sw, add → sw (escribo y leo)
Antidependencias: lw → add (leo y escribo)
dependencias de salida: ∅ (escribo y escribo) }

b → { dependencias verdaderas: lw → sw, lw → add
Antidependencias: lw → add, sw → add
dependencias de salida: lw → add }

b) Indica los riesgos y añade instrucciones nop para resolverlos.

a =>

cc1	cc2	cc3	cc4	cc5	cc6	cc7
IF	ID(RG)	EX	MEM	WB(W1)		
	IF	ID(R2)	EX	MEM	WB(W6)	
		IF	ID(R6,R1)	EX	MEM	WB

=>

=> Hay un riesgo en ID(R6,R1) ya que $1 y $6 se escriben más tarde, para solucionarlo añadimos dos "nops" después de add (ya que hay adelantamiento e el banco de registros, es decir, la escritura se realiza en la primera mitad del ciclo).

b =>

cc1	cc2	cc3	cc4	cc5	cc6	cc7
IF	ID (R5)	EX	MEM	WB (W5)		
	IF	ID (R5)	EX	MEM	WB	
		IF	ID (R5)	EX	MEM	WB (R5)

=>

=> Hay un riesgo en ID(R5)(cc3), para solucionarlo añadimos dos nops

6. Sea el programa suma1 dado por el siguiente código MIPS:

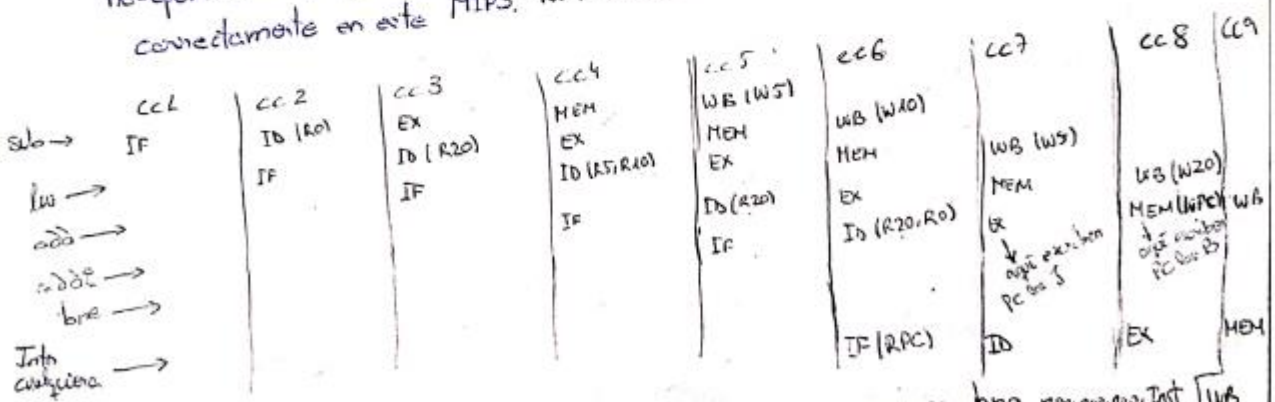
```
        sub $5, $0, $0
suma1:  lw  $10, 1000($20)
        add $5, $5, $10
        addi $20, $20, -4
        bne $20, $0, suma1
```

b) Detectar las dependencias que afectan a suma1 en el MIPS segmentado en 5 etapas y clasificarlas en dependencias de datos y dependencias de control.

Dependencias de datos:
- Dependencias verdaderas: sub → add, lw → add, addi → bne, addi → lw
- Antidependencias: lw → addi, add → lw, bne → addi
- Salida: sub → add

- Dependencias de control: bne → suma1

c) Suponer un MIPS segmentado sin ningún tipo de soporte hardware para solventar los problemas derivados de los riesgos de datos que conlleva la segmentación. Reescribir el código reordenándolo e insertando el mínimo número de códigos de no-operación (nop) para producir una nueva versión, suma2, que pueda ejecutarse correctamente en este MIPS. NO HAY ADELANTAMIENTO DE REGISTRO.



CORRECCIÓN INICIAL: sub, lw, nop, nop, nop, add, addi, nop, nop, nop, bne, nop, nop, nop, Inst

CORRECCIÓN DEFINITIVA: sub, lw, addi, nop, nop, add, bne, nop, nop, nop, Inst

obtenida hasta ahora con respecto al punto de partida. Para ello, ... ejecutarse sobre el MIPS

CORRECCIÓN INICIAL: sub, lw, [illegible]

CORRECCIÓN DEFINITIVA: sub, lw, addi, nop, nop, add, bne, nop, nop, nop, inst

d) Evaluar la mejora obtenida hasta ahora con respecto al punto de partida. Para ello, comparar el tiempo de ejecución que ofrece suma2 al ejecutarse sobre el MIPS anterior con respecto al de suma1 sobre el MIPS sin segmentar de la implementación monociclo. Suponer un número N de iteraciones en ambos programas y que las latencias de las etapas del camino de datos del procesador son las vistas en clase: IF (40 ps), ID (20 ps), EX (40 ps), MEM (40 ps), WB (20 ps).

Tiempo de ciclo para un monociclo → 160 ps = latencia monociclo.
Tiempo de ciclo segmentado → 40 ps. Latencia segmentado = 40·5 = 200 ps
Tiempo ejecución monociclo → (1 + 4·N)·160 ps
" " segmentada → (4 ciclos · 1 inst + 9 inst · N)·40 = (5+9·N)·40

una arquitectura RISC con un conjunto de instrucciones similar al del procesador MIPS visto en clase. Su camino de datos presenta una segmentación en 3 etapas y una única memoria, común para instrucciones y datos. Esto hace que las instrucciones se ejecuten según se muestra a continuación:

INST	1	2	3	4	5
Inst 1	IFD	REX	MEM		
Inst 2		IFD	REX	MEM	
Inst 3			IFD	REX	MEM

donde:

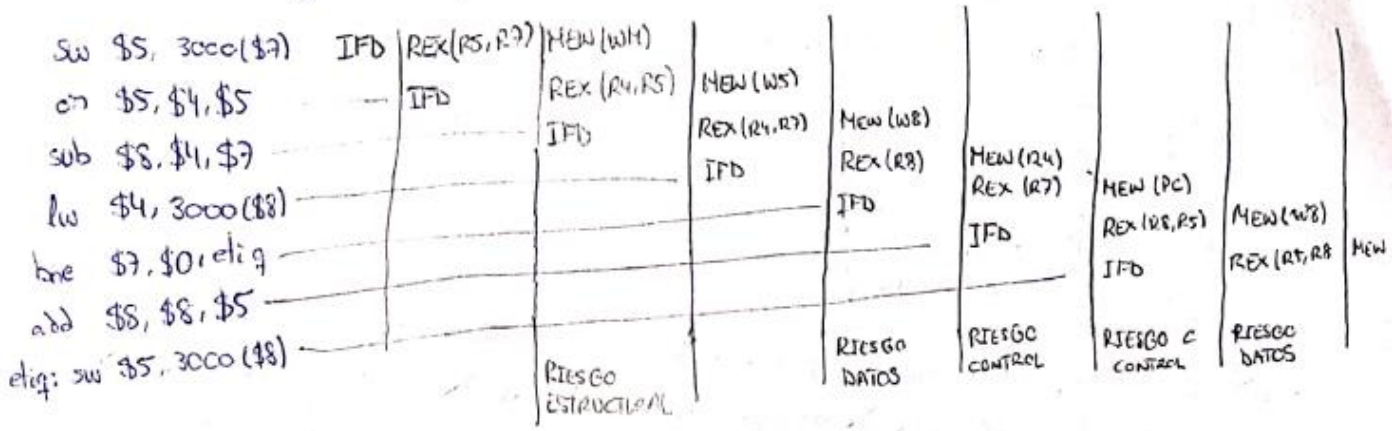
- IFD es la etapa de búsqueda y decodificación de instrucción
- REX es la etapa de búsqueda de operandos (lectura del banco de registros), ejecución de operación ó resolución de condiciones de salto y cálculo de dirección para operando destino ó salto.
- MEM es la etapa de acceso a memoria para instrucciones de carga/almacenamiento/escritura de resultados en registros para instrucciones aritméticas/lógicas y de carga y actualización del PC para instrucciones de salto.

Además no podrá leerse en un mismo ciclo un registro que a ser escrito en dicho ciclo (es decir, no hay anticipación en el banco de registros).

Bajo estas condiciones, responder a las siguientes cuestiones:

a) ¿Qué tipo de riesgos pueden darse? Poner ejemplos.

c) Representar en un diagrama de ciclos la evolución del cauce para el siguiente trozo de código e identificar los riesgos que pueden darse.



7. Sea el siguiente código MIPS, que a partir de ahora referenciaremos como mi_prog:

```
        ori  $2, $0, 1000h
bucle:  lw   $1, 2800h($2)
        sub  $4, $1, $0
        jal  rotar
        sw   $7, 7800h($2)
        sw   $1, C800h($2)
        subi $2, $2, 4
        bne  $2, $0, bucle

rotar:  add  $10, $4, $4
        muli $7, $10, 2
        jr   $31
```

Señalar las instrucciones que pueden producir riesgos en mi_prog para un MIPS segmentado con anticipación en el banco de registros. Responder en las siguientes tablas:

	Instrucción(es)	Registro(s) involucrado(s)
Riesgos de datos →	ORI → LW	R2
	LW → SUB	R1
	SUBI → BNE	R2
	ADD → MULI	R10
	MULI → SW	R7
	SUB → ADD	R4
	SUBI → LW	R2

	Instrucción(es)
Riesgos de control →	JAL → SW, SW
	BNE → I_4, I_7, I_3 (vienen dadas del BNE pero no las sabemos)
	JR → I_4, I_5