OPTIMITZACIÓ RISC-V

1. Després de fer diverses proves en un RISC-V single cycle, hem decidit dissenyar el nostre RISC-V amb 5 etapes de pipeline. Un cop dissenyat, programem un conjunt de patrons de test per a provar el nostre processador. Els següents casos fallen:

```
Suposarem que tots els registres s'inicialitzen a 0.

Test 1: Suposem que l'adreça d'una matriu amb tots els valors diferents
s'emmagatzema a s0 (emprem nomenclatura ripes)
      addi t0 x0 1
      slli t1 t0 2
      add t1 s0 t1
      lw t2 0(t1)
```

Cada cop que executem "Test 1" obtenim el mateix valor incorrecte a t2. Test 1 funciona perfectament en el RISC-V single cycle. Podeu explicar quin tipus de problema tenim? Com el solucionaríeu?

slli rd, rs1, shamt `x[rd] = x[rs1] << shamt`

Shift Left Logical Immediate. Tipo I, RV32I y RV64I.
Corre el registro x[*rs1*] a la izquierda por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado es escrito en x[*rd*]. Para RV32I, la instrucción solo es legal cuando *shamt*[5]=0.
Forma comprimida: **c.slli** rd, shamt

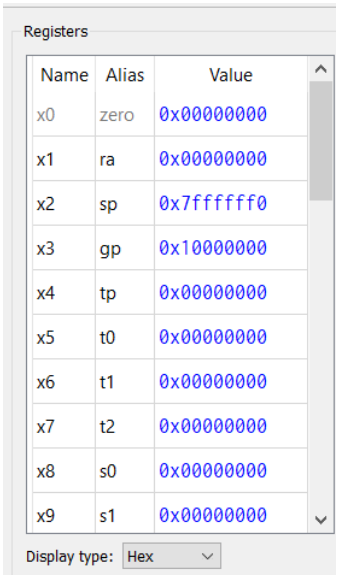
31 – 26	25 – 20	19 – 15	14 – 12	11 – 7	6 – 0
000000	shamt	rs1	001	rd	0010011

lw rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

Load Word. Tipo I, RV32I y RV64I.
Carga cuatro bytes de memoria en la dirección x[*rs1*] + *sign-extend*(*offset*) y los escribe en x[*rd*]. Para RV64I, el resultado es extendido en signo.
Formas comprimidas: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[11:0]	rs1	010	rd	0000011

		0	1	2	3	4	5	6	7	8
1	addi t0, x0, 1	F	D	C	M	W				
2	slli t1, t0, 2		F	D	C	M	W			
3	add t1, s0, t1			F	D	C	M	W		
4	lw t2, 0(t1)				F	D	C	M	W	



t0 → 1 0001 t0 → 1 0001
t1 → 1 0001 t1 → 4 0100
t1 → 1 0001 t1 → 4 0100
t2 → contenido que hay en la dirección de memoria 4 (en este ciclo aún no se ha actualizado el valor de t1

Como partimos de un Single Cycle, donde no se necesita control de Hazard ya que no hay riesgos de datos, al añadir un pipeline tendremos problemas.
Para solucionarlo podríamos usar NOOPs o añadir un Data Hazard para aplicar DFPs.

```
Test 2: (després de solucionar Test 1)

      addi s0 x0 4
      slli t1 s0 2
      bge s0 x0 greater
      xori t1 t1 -1
      addi t1 t1 1
      greater:
      mul t0 t1 s0
```

Cada cop que executem "Test 2" es finalitza amb t0 guardant el valor 0xFFFFFFC0, que no és el que hauria de tenir. Podeu explicar quin tipus de problema tenim ? Com el solucionaríeu ?
Nota: aquestes preguntes us apareixeran en forma de "preguntes amb diferents opcions de resposta" al qüestionari, és possible que hi hagi múltiples respostes.

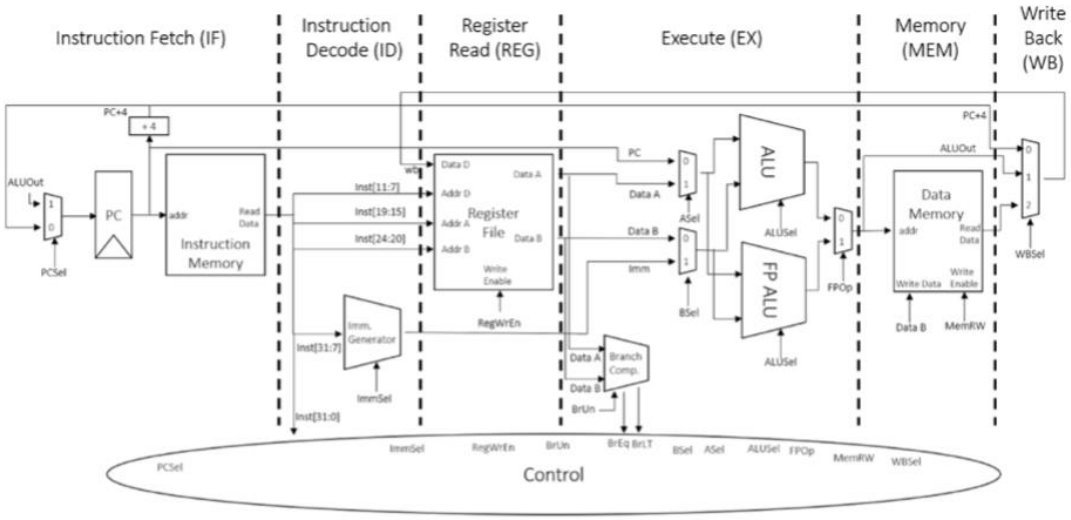
Para este test volvemos a tener el mismo problema.
Para tratar el salto podemos usar un Control Hazard de predicción de salto, que siempre que tengamos un salto condicional supondrá que la condición no se cumple y seguirá con la siguiente instrucción. Sabemos si el salto se va a producir o no después de la etapa M. En nuestro caso sí se produce, por lo que se hará un flush de las últimas instrucciones que han empezado después del salto.

		0	1	2	3	4	5	6	7	8	9	10
1	addi s0, x0, 4	F	D	C	M	W						
2	slli t1, s0, 2		F	D	C	M	W					
3	bge s0, x0, greater			F	D	C	M	W				
4	xori t1, t1, -1				F	D	C	M	W			
5	addi t1, t1, 1					F	D	C	M	W		
6	mul t0, t1, s0							F	D	C	M	W

s0 → 4 0100
t1 → 16 1 0000
salta a *greater*

2. En aquest problema, treballareu amb una CPU RISC-V modificada.

A diferència del RISC-V tradicional de 5 etapes, aquesta CPU alterada ha dividit la segona etapa (decode) en dues etapes diferenciades: decode i registre de lectura. A més, aquesta CPU té una ALU pels càlculs de coma flotant (FP ALU) junt amb una ALU tradicional (no s'afegeixen registres al registre file per guardar valors per a la FP ALU). El senyal de control afegit, FPOp, determina quina ALU s'ha d'utilitzar per a una instrucció determinada. Aquesta ALU de coma flotant funciona interpretant els seus operands de 32 bits en format de coma flotant IEEE 754. A diferència del RISC-V tradicional, suposem que les operacions de coma flotant utilitzen els mateixos registres que les operacions normals de coma no flotant. En la figura 1 mostrem un diagrama de la CPU modificada i les seves etapes corresponents. En la taula 1, proporcionem els retards de cada etapa:



Element	Register CLK-to-Q	Register Setup	MUX	ALU	FP ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup	Imm. Gen.
Delay (ps)	20	25	10	150	215	225	230	100	35	75

EC – Qüestionari 13 (Opcional) Problema Pipeline Extra

a) Penseu en la instrucció de coma flotant, faddi, que és similar a l'addi, tret que considera operands en format de coma flotant i executa una operació d'adició de coma flotant. Suposant que aquesta CPU NO té pipeline (és a dir, és una CPU single cycle), quin és el període de rellotge més curt possible per executar la instrucció faddi t0, s0, 2.71 correctament?

Calculamos el tiempo que tardará cada etapa:

Etapa IF: lee el registro PC y busca su contenido en memoria para saber el tipo de instrucción que debe ejecutar.

- MUX para saber el valor de PC: MUX (10ps)
- Usa el registro PC, por lo que consideramos el tiempo que tarda en salir el dato una vez llega al clock: RegClk-to-Q (20ps)
- Leemos en memoria el dato del PC: Mem Read (225ps) (*en paralelo* se ejecuta un MUX para saber el valor de PC: MUX (~~10ps~~))

255ps

Etapa ID:

- Imm. Generator (75ps)

75ps

Etapa REG:

- Reg File Read (100ps)

100ps

Etapa EX:

- Se ejecutan 2 MUX que podrían hacerse al mismo tiempo por lo que contamos solo 1 vez (10ps)
- Usamos la FP ALU (215ps)
- Usamos otro MUX (10ps)

235ps

Etapa MEM:

- En esta ocasión no tenemos que guardar nada en memoria.

Etapa WB:

- Se usa un MUX (10ps)
- Se guarda el resultado en un register file: RegFileSetup (35ps)

45ps

Como no tenemos pipeline, no estamos "segmentando" las etapas, ya que no necesitamos diferenciarlas porque en todo momento solo se ejecutará una instrucción. Si nos fijamos las etapas ID y REG pueden ejecutarse en paralelo, por lo que sólo contaremos el tiempo de la que más tarda.

$T_{TOTAL} = 255 + 100 + 235 + 45 = \mathbf{635\ ps}$

b) Quin és el període de rellotge més curt possible per executar instruccions en aquesta CPU si SÍ tenim pipeline?

En el caso de tener pipeline debemos separar las etapas: ID y REG ya no podrán ejecutarse paralelamente.
El tiempo de cada etapa será el de la que tarde más, ya que se deberá esperar a que todas las etapas hayan terminado para seguir el programa.

$T_{TOTAL} = 255 \times 6 = \mathbf{1530\ ps}$

c) Seguint amb la CPU amb pipeline modificat, executeu CORRECTAMENT el programa següent seguint les següent suposicions:

c1. No hi ha optimitzacions de pipeline (no forwarding, ni load delay slot, ni branch prediction, ni buidat de pipeline, etc)

c2. No podem llegir i escriure els registres en un mateix cicle de rellotge.

c3. Un integer (100) es guarda a la direcció de memòria 0x61C61C61, i que R[a0] = 0x61C61C61

c4. Un hazard entre dues instruccions s'ha de comptar com 1 sol hazard

```
Codi:
      lw t0, 0(a0) # R[a0] = 0x61C61C61
      srli s0, t0, 4
      faddi s1, t0, 1.7
      beq a0, s1, Label
      add a1, t2, t3
      Label ...
```

EC – Qüestionari 13 (Opcional) Problema Pipeline Extra

No importa el tipus de diagrama que feu servir per a completar l'execució, o si no en necessiteu cap. El resultat us ha de servir per contestar les següents preguntes:

i. Quants cicles de rellotge triga en executar-se el programa correctament? 18

ii. Quants stalls/noop heu hagut d'afegir per executar correctament el programa? 9

iii. Quants data hazards hi ha en el programa? 3

iv. Quants control hazard hi ha en el programa? 1

Haurem d'afegir els NOOPs necessaris per a que el codi s'executi correctament

srli rd, rs1, shamt `x[rd] = x[rs1] >>u shamt`

Shift Right Logical Immediate. Tipo I, RV32I y RV64I.

Corre el registro x[*rs1*] a la derecha por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado es escrito en x[*rd*]. Para RV32I, la instrucción solo es legal cuando *shamt*[5]=0.

Forma comprimida: **c.srli** rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0010011

beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

Branch if Equal. Tipo B, RV32I y RV64I.

Si el registro x[*rs1*] es igual al registro x[*rs2*], asignar al *pc* su valor actual más el *offset* sign-extended.

Forma comprimida: **c.beqz** rs1,offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	lw t0, 0(a0)	F	D1	D2	C	M	W														
2	srli s0, t0, 4		-	-	-	F	D1	D2	C	M	W										
3	faddi s1, t0, 1.7						F	D1	D2	C	M	W									
4	beq a0, s1, Label							-	-	-	F	D1	D2	C	M	W					
5	add a1, t2, t3											-	-	-	F	D1	D2	C	M	W	

d) Reordeneu el codi per aconseguir el mateix resultat minimitzant el nombre total de cicles d'execució:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	lw t0, 0(a0)	F	D1	D2	C	M	W														
2	faddi s1, t0, 1.7		-	-	-	F	D1	D2	C	M	W										
3	srli s0, t0, 4						F	D1	D2	C	M	W									
4	beq a0, s1, Label							-	-	-	F	D1	D2	C	M	W					
5	add a1, t2, t3											-	-	-	F	D1	D2	C	M	W	

i. Quants cicles de rellotge triga en executar-se el programa correctament?

ii. Quants stalls/noop heu hagut d'afegir per executar correctament el programa?

iii. Quants data hazards hi ha en el programa?

iv. Quants control hazard hi ha en el programa?