1.3 Test aléas de donnée

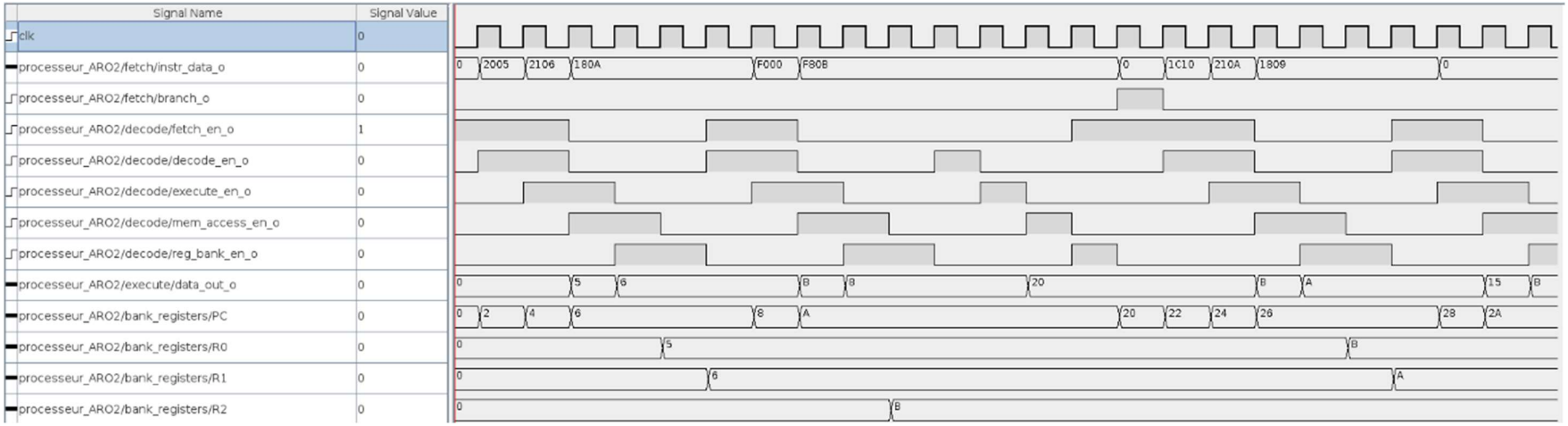
1.4 Programme et fonctionnement théorique

```
@ Programme de test des aléas de données

mov r0, #5
mov r1, #6
add r2, r1, r0  @RAW I2(R1), I1(R0)
bl test_1

.org 0x20
test_1:
mov r0, r2      @RAW I3(R2), I1(R0)
mov r1, #10
add r1, r0      @RAW I5(R0), I6(R1)
```

			Dep RAW	1	2	3	4	5	6	7	8	9	10	11	12
		Arret Pipeline / Forwarding													
1	MOV	r0, #5		F	D	E	M	W							
2	MOV	r1, #6			F	D	E	M	W						
3	ADD	r2, r1, r0	I2(R1), I1(r0)			F	D	E	M	W					
4	BL	0x20													
0x20															
5	MOV	r0, r2	I3(R2)						F	D	E	M	W		
6	MOV	r1, #10								F	D	E	M	W	
7	ADD	r1, r0	I5(R0)								F	D	E	M	W

En principe, avec un pipeline forwardé, il nous faudrait au minimum 12 cycles pour terminer notre programme, peut-être un peu plus car l'instruction BL ne se comporte pas comme une instruction de contrôle standard.

1.4.1 Chronogramme



Nous voyons dans ce chronogramme Que notre processeur ne fonctionne pas comme en théorie avec le forwarding. En effet, lors d'aléas (instructions 180A et 1809), le processeur met à 0 les enable de chaque étage du pipeline : le fetch, le décode, l'execute. Le memory access et le writeBack (RegBank). Or en théorie, ces aléas devraient ne pas stopper le pipeline, pas prendre de coups d'horloge supplémentaires grâce au forwarding.

A cause de cela, notre programme prend 22 coups de clock pour s'exécuter. A noter que l'instruction BL s'exécute en 8 coups de clock. Tout vient à penser que ce circuit de forwarding ne fonctionne pas correctement.