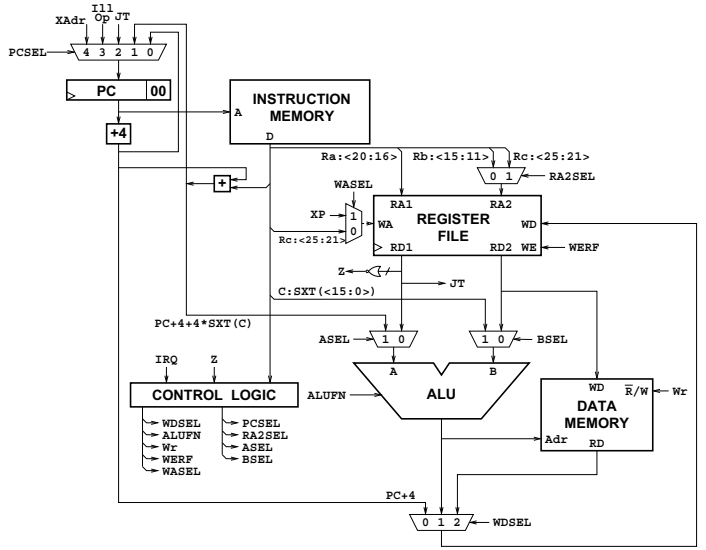
La technique du “pipeline” et la machine β

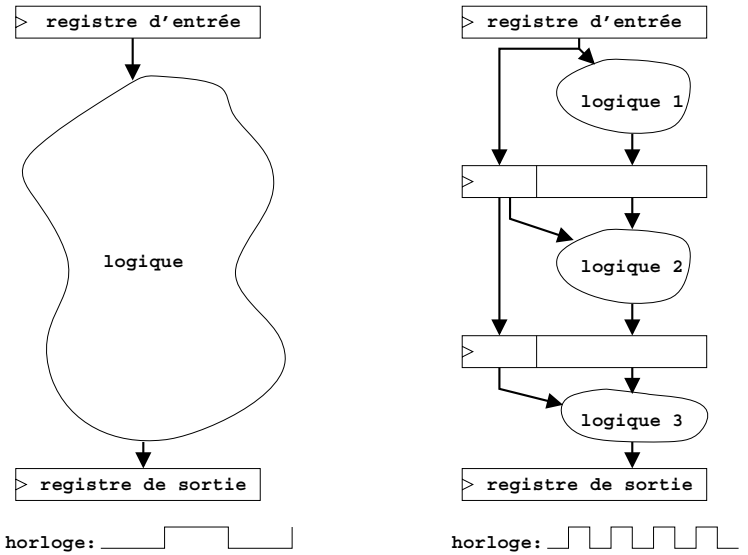
Rappel : une implémentation “un cycle” de β



Principe du pipeline

- Un pipeline est utile pour augmenter la cadence à laquelle un circuit combinatoire profond peut traiter des données.
- Un pipeline consiste à diviser un circuit combinatoire en “couches” ou *stades* (*stages*) et à séparer ces couches par des registres.
- Les données sont sauvées dans des registres entre chaque couche ; plusieurs phases sont donc nécessaires pour traiter des données, mais chaque phase peut être nettement plus rapide.
- Le temps de transit des données à travers le circuit combinatoire est inchangé ou légèrement augmenté, mais la cadence de traitement des données est fortement augmentée.

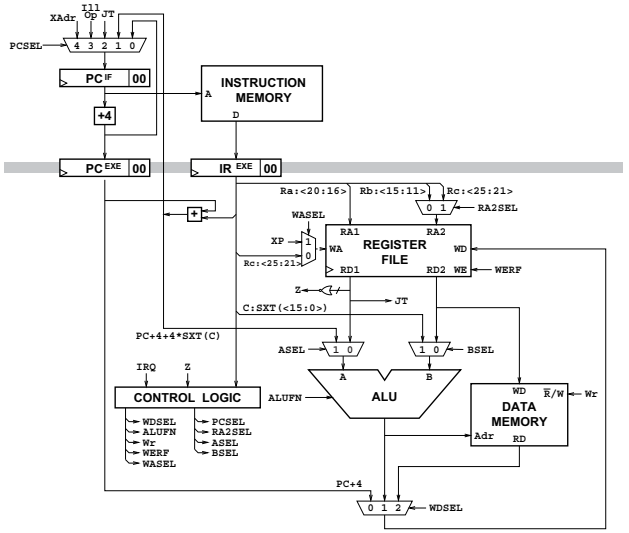
Principe du pipeline : illustration



Un pipeline pour β

- Dans l'implémentation considérée de la machine β, la logique combinatoire utilisée à chaque cycle comporte quatre *stades* (*stages*) bien identifiables.
  - La lecture de l'instruction dans la mémoire d'instruction : *Instruction Fetch* (IF) ;
  - La lecture des arguments de l'instruction en cours dans les registres : *Register Fetch* (RF)
  - Le calcul de l'ALU : ALU ;
  - La lecture de données en mémoire (instruction `LD`) : *Memory* (MEM) ;
  - et se termine par l'écriture des résultats dans les registres ·*Write Back* (WB)
- Un pipeline à quatre stades semble donc tout indiqué. Pour simplifier, nous commencerons par un pipeline à 2 stades : IF et EXE (*execute*).

Une machine β avec pipeline à 2 stades



Le fonctionnement du pipeline à 2 stades

A tout moment, il y a deux instructions en exécution : une au stade IF et une au stade EXE.

Considérons par exemple le programme suivant :

```
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
```

L'état du pipeline évolue alors comme représenté ci-dessous.

temps →

	i	i+1	i+2	i+3	i+4	i+5	i+6
stade IF	**ADDC**	**SUBC**	**XOR**	**MUL**	···		
stade EXE		**ADDC**	**SUBC**	**XOR**	**MUL**	···	

Le fonctionnement du pipeline à 2 stades : Le problème des sauts

Lorsqu'une instruction de saut (`JMP` ou `BXX`) apparaît, un problème se pose car l'instruction suivante a déjà été lue.

Considérons par exemple :

```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      XOR(r3, -1, r3)
```

Si le saut doit être effectué, le pipeline est dans un état incohérent.

temps →

	i	i+1	i+2	i+3	i+4	i+5	i+6
stade IF	**ADD**	**CMP**	**BT**	**XOR**	···		
stade EXE		**ADD**	**CMP**	**BT**	???	···	

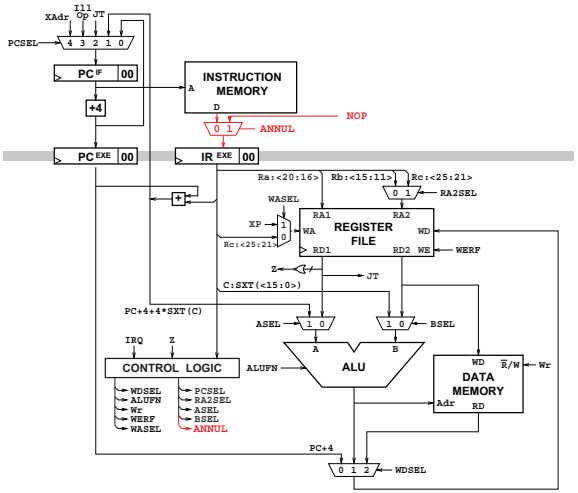
Le fonctionnement du pipeline à 2 stades : Solutions au problème des sauts

Il y a deux types de solutions possibles à ce problème.

- **Les solutions hardware** : le hardware "annule" l'instruction qui suit les sauts qui sont pris.

- **Les solutions software** : un saut est toujours suivi d'une instruction sans effet (`NOP`), ou d'une instruction "utile" non problématique.

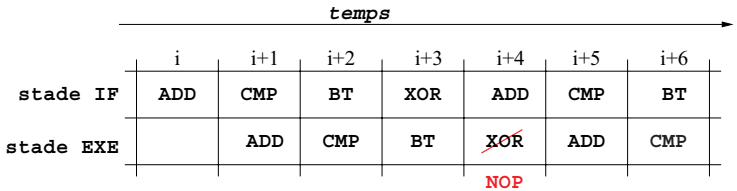
Une machine β avec pipeline à 2 stades : La solution hardware au problème des sauts



Une machine β avec pipeline à 2 stades : La solution hardware au problème des sauts – exemple

Pour le programme suivant
```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      XOR(r3, -1, r3)
```

Si le saut est effectué, le pipeline se comporte comme suit.

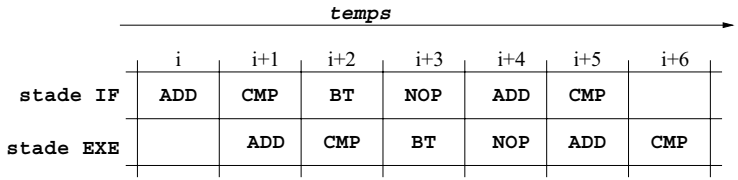


Une machine β avec pipeline à 2 stades : première solution software – exemple

Une instruction `NOP` est ajoutée après le saut.

```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      NOP()
      XOR(r3, -1, r3)
```

L'instruction `NOP` est systématiquement exécutée après le saut, mais est sans effet.



Une machine β avec pipeline à 2 stades : deuxième solution software – exemple

Le programme précédent est réécrit comme suit.

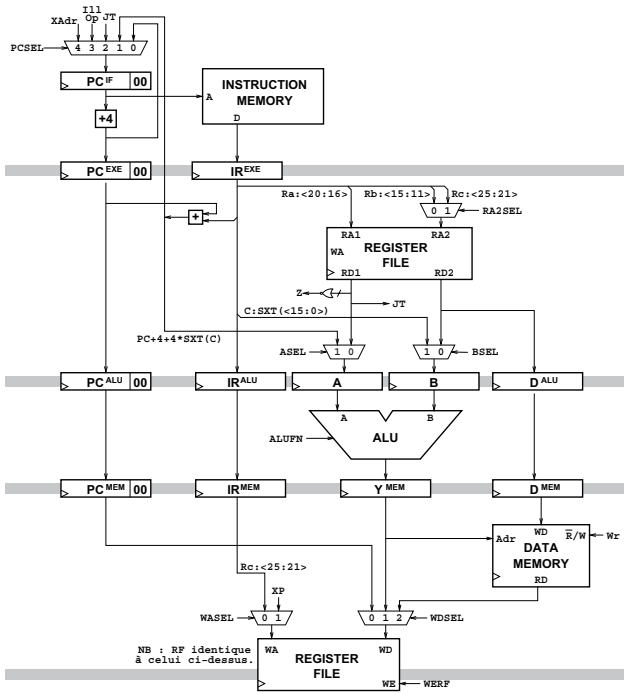
```
LOOP:  ADD(r1, r3, r3)
LOOPx: CMPLEC(r3, 100, r0)
       BT(r0, LOOPx)
       ADD(r1, r3, r3)
       SUB(r3, r1, r3)
       XOR(r3, -1, r3)
```

On insère donc après `BT` une instruction de toute façon exécutée lorsque l'on fait le saut (la majorité des cas) et l'on corrige l'effet de cette instruction lorsque l'on ne fait pas le saut.

Il faut noter que le comportement de ce programme n'est pas correct sur une machine sans pipeline.

Une machine β avec pipeline à 4 stades

- La figure de l'écran suivant montre une ébauche de réalisation de la machine β avec un pipeline à quatre stades.
- Les stades sont : IF, RF, ALU et MEM/WB.
- Les registres apparaissent comme un circuit combinatoire en lecture et comme un circuit cadencé par l'horloge en écriture.
- Les registres se trouvant en bas de la figure sont identiques à ceux se trouvant en haut de figure.



Le fonctionnement du pipeline à 4 stades

A tout moment, il y quatre instructions en exécution.

Considérons par exemple le programme suivant :

```
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
```

L'état du pipeline évolue alors comme représenté ci-dessous.

temps

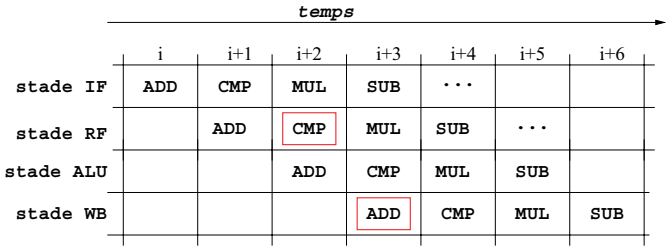
	i	i+1	i+2	i+3	i+4	i+5	i+6
stade IF	ADDC	SUBC	XOR	MUL	···		
stade RF		ADDC	SUBC	XOR	MUL	···	
stade ALU			ADDC	SUBC	XOR	MUL	
stade WB				ADDC	SUBC	XOR	MUL

Le fonctionnement du pipeline à 4 stades : le problème des conflits de données

Considérons par exemple le programme suivant :

```
ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
```

l'instruction `CMPLEC` a besoin au temps `i+2` du contenu du registre `r3` qui ne sera écrit qu'à la fin du cycle `i+3`.



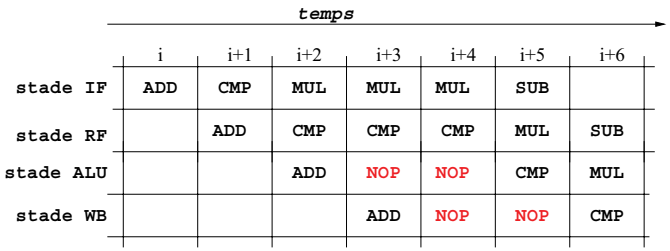
Le fonctionnement du pipeline à 4 stades : le problème des conflits de données – solution software

Une première solution consiste à modifier le programme pour qu'il n'existe plus de conflits de données.

```
ADD(r1, r2, r3)          devient     ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)                  MULC(r1, 100, r4)
MULC(r1, 100, r4)                    SUB(r1, r2, r5)
SUB(r1, r2, r5)                      CMPLEC(r3, 100, r0)
```


Le fonctionnement du pipeline à 4 stades : le problème des conflits de données – solution hardware 1

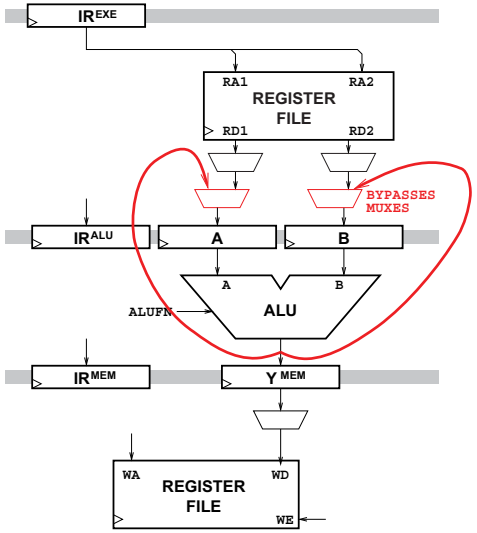
Une solution hardware consiste à bloquer le pipeline (et à insérer des `NOP`) lorsqu'il y a un conflit de données.



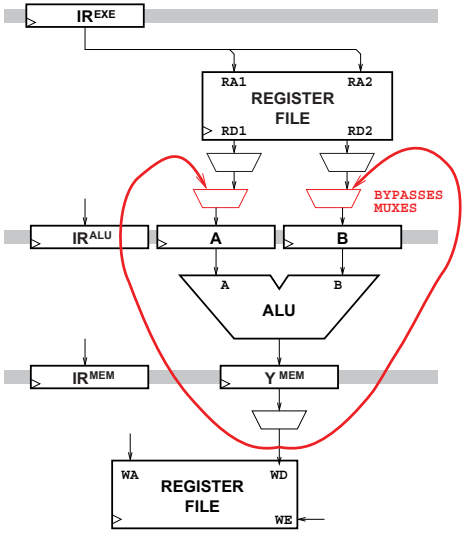
Le fonctionnement du pipeline à 4 stades : le problème des conflits de données – solution hardware 2

- Une deuxième solution hardware consiste à introduire un *contournement* (*bypass*) qui permet de récupérer la sortie de l'ALU à la place de ce qui vient en lecture des registres.
- Il est aussi nécessaire de pouvoir récupérer la sortie du stade WB lorsque les instructions en conflit sont séparées par une autre instruction.

Les chemins de “bypass” 1



Les chemins de “bypass” 2



Autres problèmes à considérer dans les pipeline

- Une instruction `LD` en conflit avec une instruction qui la suit pose un problème qui n’est pas soluble par un bypass.
- La lecture en mémoire peut être lente, ce qui allonge le temps de cycle. Une solution est d’ajouter un stade au pipeline et de laisser deux cycles pour la lecture en mémoire.
- Il faut correctement sauver `LP` en cas de saut et `XP` en cas d’exception et assurer la cohérence entre l’adresse de retour et ce qui a déjà été exécuté.