Cours 2 : Chemin de données du processeur RISC-V - opérateurs arithmétiques

Chemin de données du processeur RISC-V

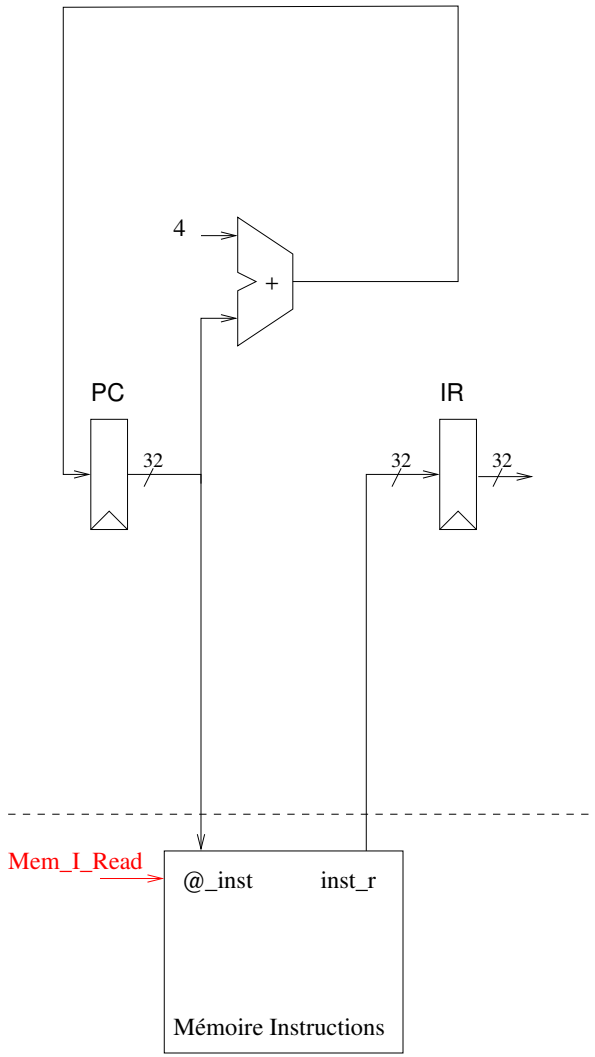
cf. l'ouvrage "Computer Organization and Design", Hennesy-Patterson, 2014, chapitre 4, "The Processor".

- Instructions 32 bits - opérandes (données et adresses) en registres
- Chemin de données pour 3 types d'instructions (+ de détails ds le cours 3)
  - opération ALU : (arithmétique ou logique)
    - ALUOP Rdest, Rsrc1, Rsrc2 : Rdest <- Rsc1 op Rsc2
    - ALUOP Rdest, Rsrc1, Imm : Rdes <- Rsc1 op (extension Imm)
  - lecture / écriture en mémoire :
    - LW Rdest, Imm(Raddr) : lit en mémoire, à l'adresse Raddr + (extension Imm) et le place dans Rdest : Rdest <- MEM(Raddr+Imm_{32})
    - SW Rsrc, Imm(Raddr) : écrit en mémoire (data) le contenu de Rsrc, à l'adresse Raddr+(extension Imm) : MEM(Raddr+Imm_{32}) <- Rsrc
  - branchement conditionnel (branch if equal) : BEQ Rsrc1, Rsrc2, dpclt

Élements constitutifs - chargement d'instruction

- place dans IR l'instruction dont l'adresse est rangée dans PC
- PC <- PC+4 (adresse de l'instruction suivante, si ce n'est pas un saut)

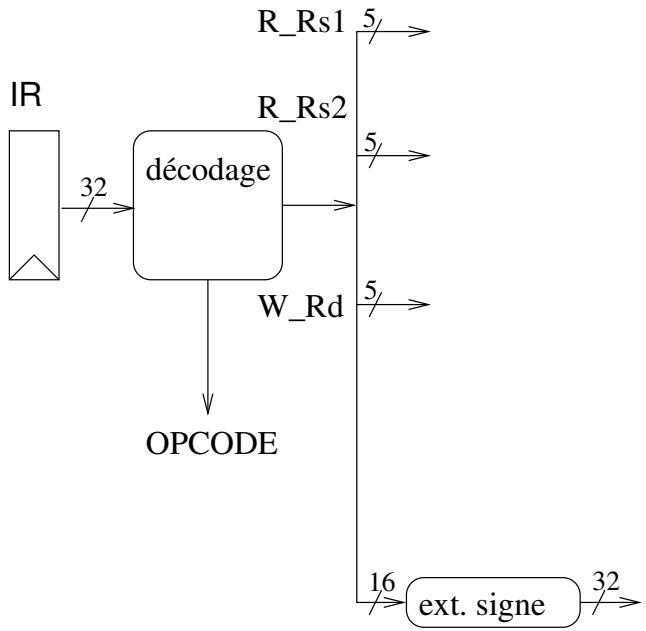
Remarque : la *mémoire Instruction* est extérieure au processeur ; elle doit répondre rapidement (idéalement en 1 cycle) -> cache L1 instructions.



Élements constitutifs - décodage d'instruction

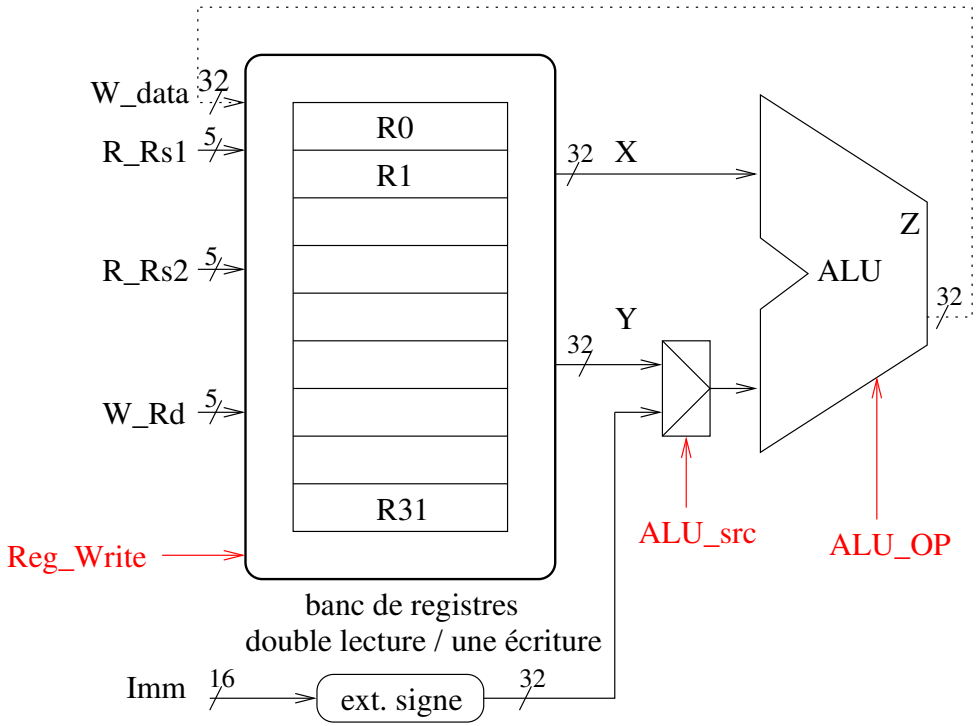
Décode l'instruction placée dans IR : identifie l'opération à réaliser, et les opérandes.

- `ALUOP Rd, Rsc1, Rsc2`
- `ALUOP Rdest, Rsc1, Imm`
- `SW Rsrc, Imm(Raddr)`
- `LW Rdest, Imm(Raddr)`
- `BEQ Rsrc1, Rsrc2, dpclt`



Élements constitutifs - exécution dans l'ALU

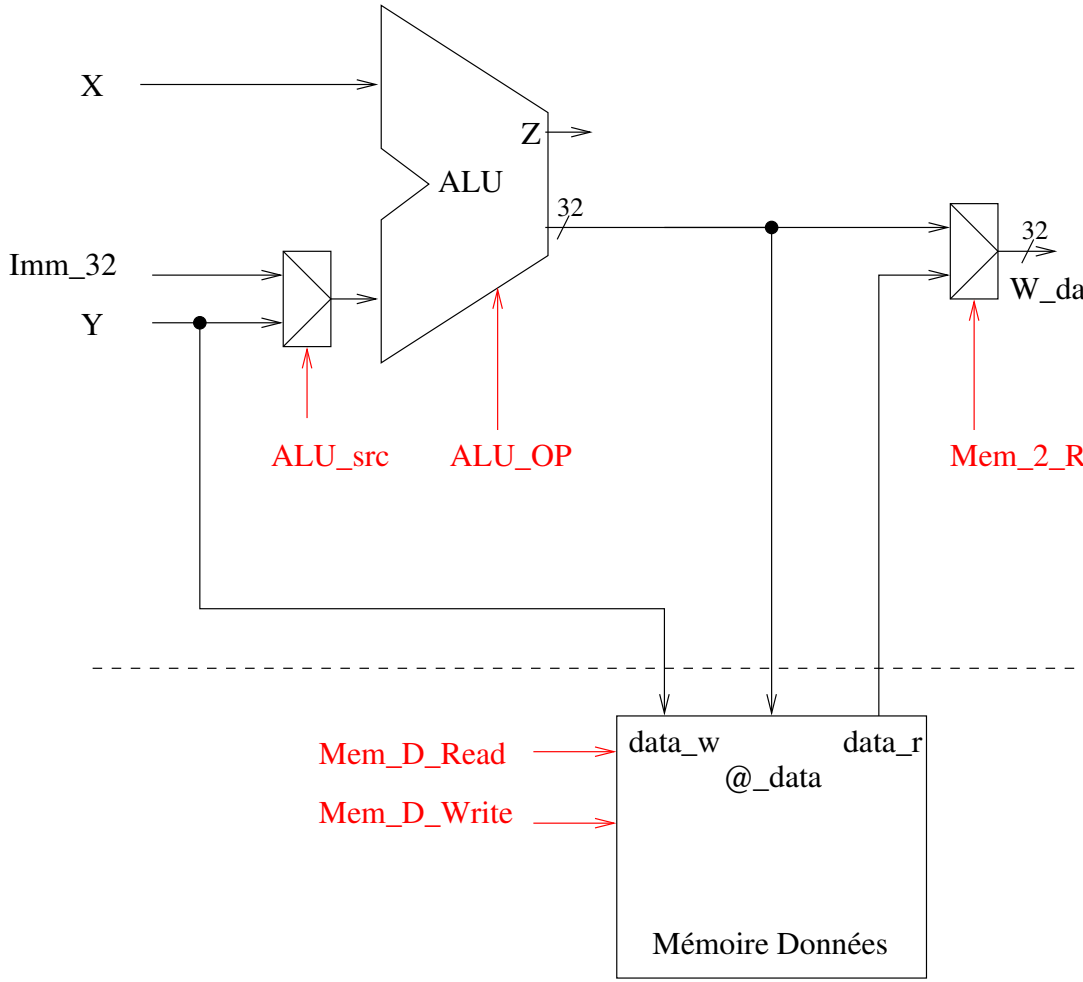
- Les données temporaires sont stockées dans le banc de registre : 2 accès en lecture (front montant de CK) et 1 accès en écriture (front descendant de CK, si Reg_Writ eest positionné)
- R_Rs1 et R_Rs2 identifient les registres à lire et placer sur la sortie du banc (X et Y)
- W_Rd identifie le registre recevant une écriture
- W_data : données à écrire
- Reg_Write, ALU_src et ALU_OP : commande



Élements constitutifs - lecture / écriture de données en mémoire

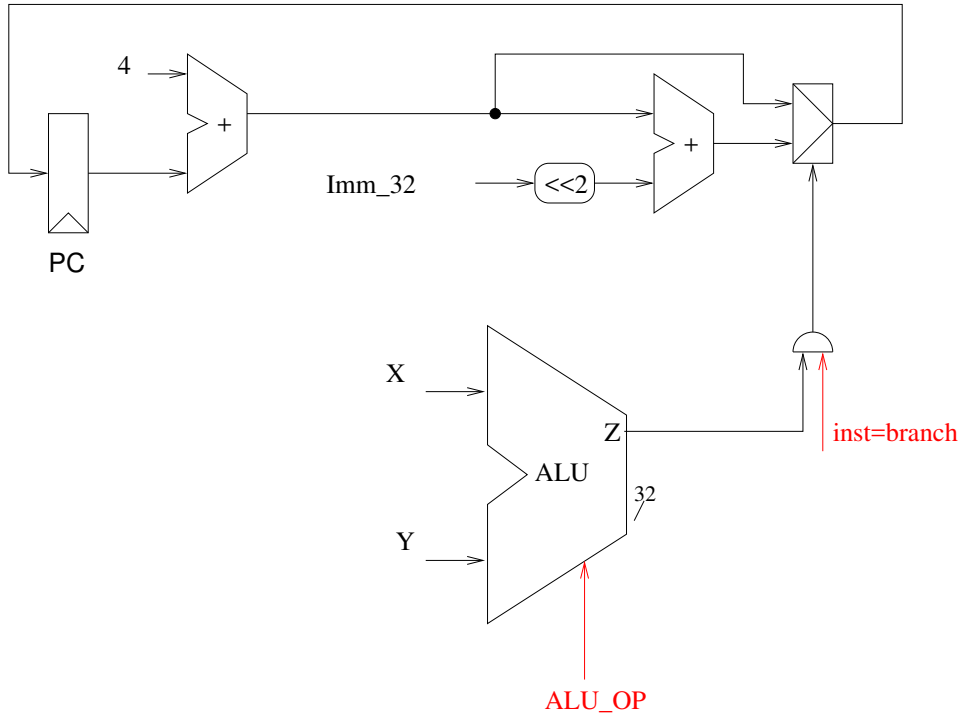
- la sortie X du banc de registres contient l'adresse à sommer à Imm_32
- la sortie de l'ALU contient l'adresse du mot mémoire à tranférer
- la sortie Y du banc de registres contient la donnée à écrire en cas de sw
- la valeur lue en mémoire est transmise sur W_Data (entrée du banc de registres) en cas de lw

Remarque : La *mémoire de Données* est externe au processeur et doit répondre rapidement (idéalement en 1 cycle) -> cache L1 données.

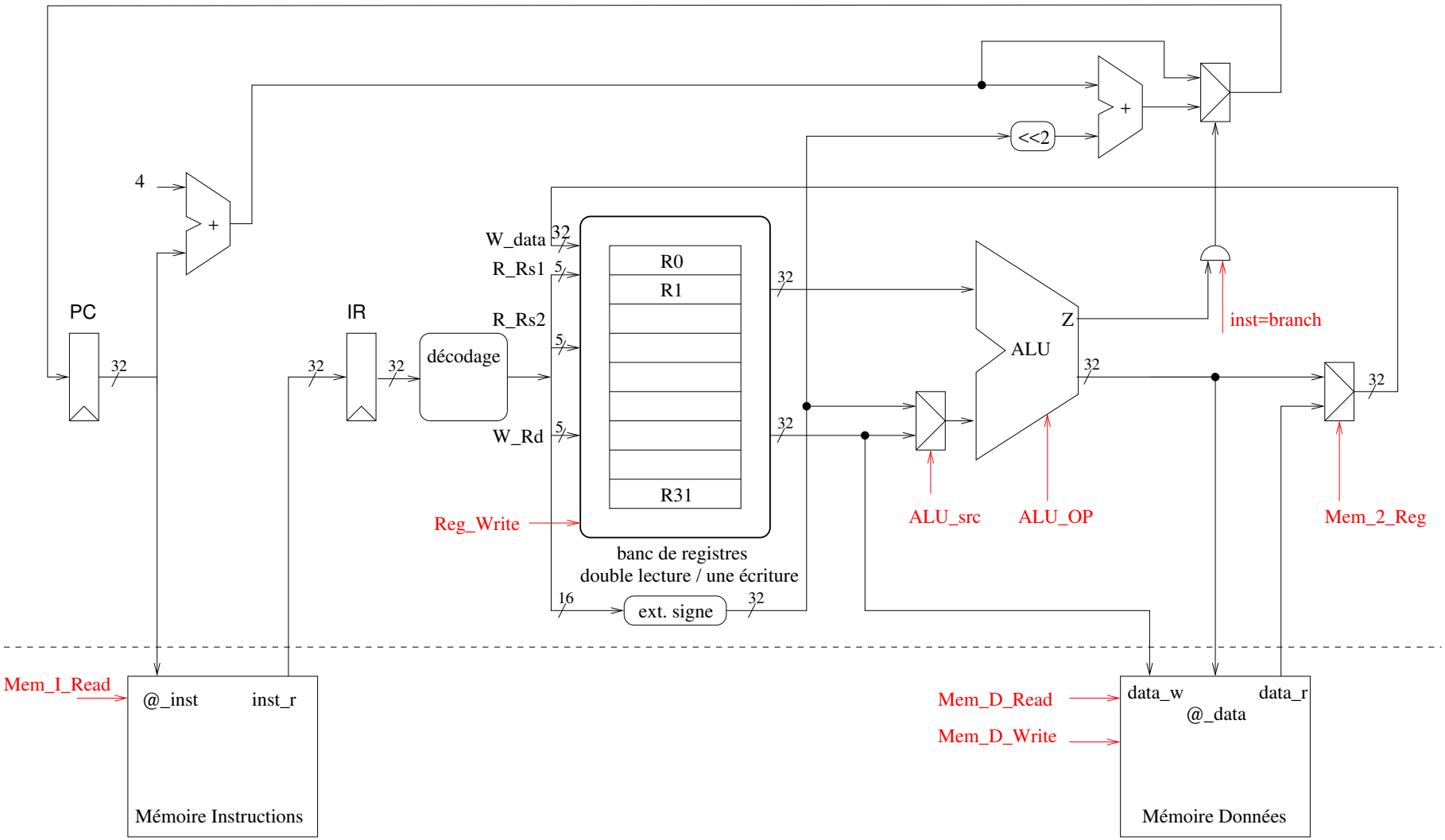


Élements constitutifs - calcul adresse de saut

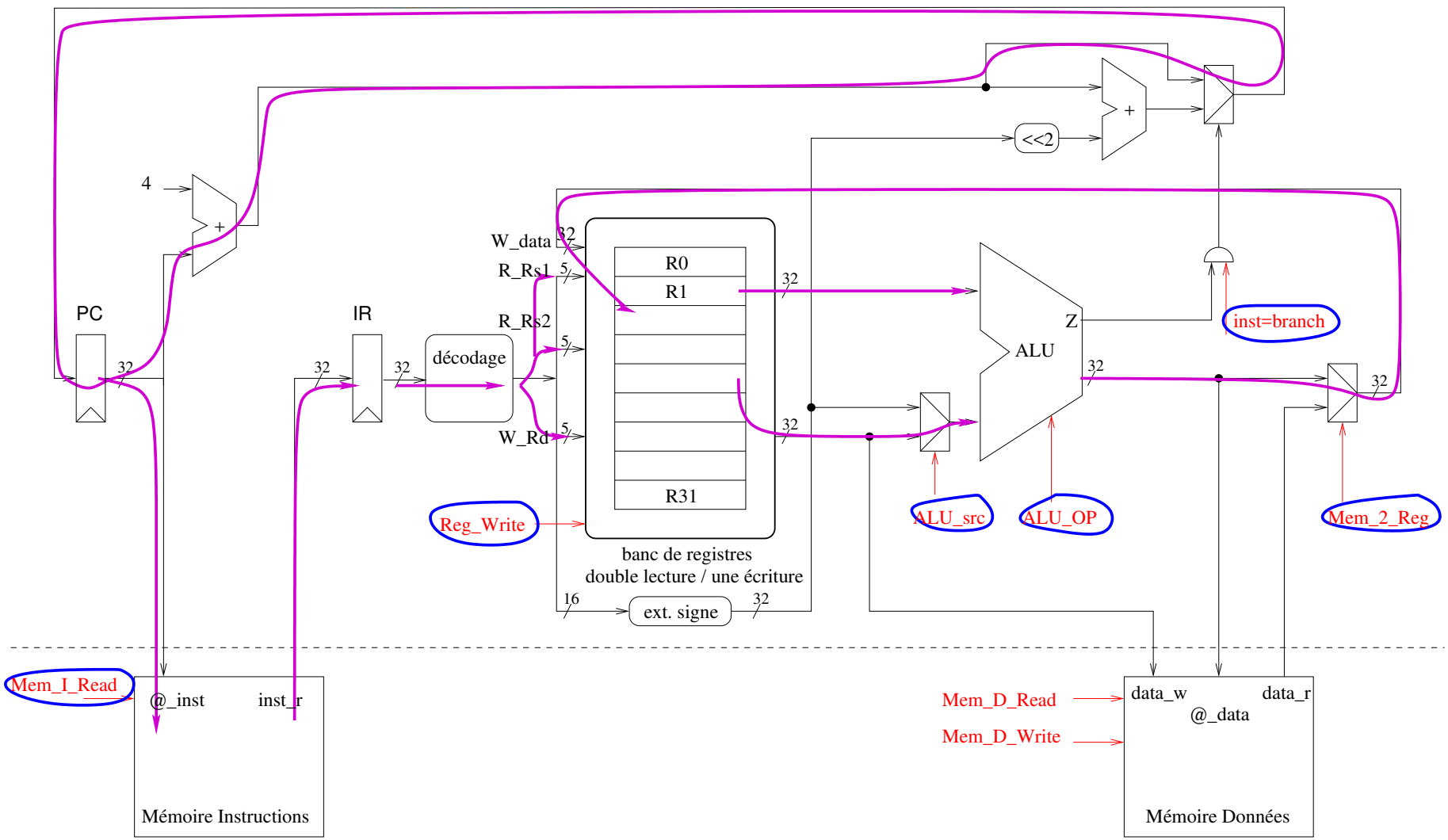
- BEQ Rsrc1, rsrc2, dpclt
- le test d'égalité des registres est réalisé par une comparaison dans l'ALU (soustraction), et si Z = 1, les 2 registres contiennent des valeurs égales.
- si Rsrc1 == Rsrc2 : PC <- PC + (dpclt«2)
- si Rsrc1 != Rsrc2 : PC <- PC + 4



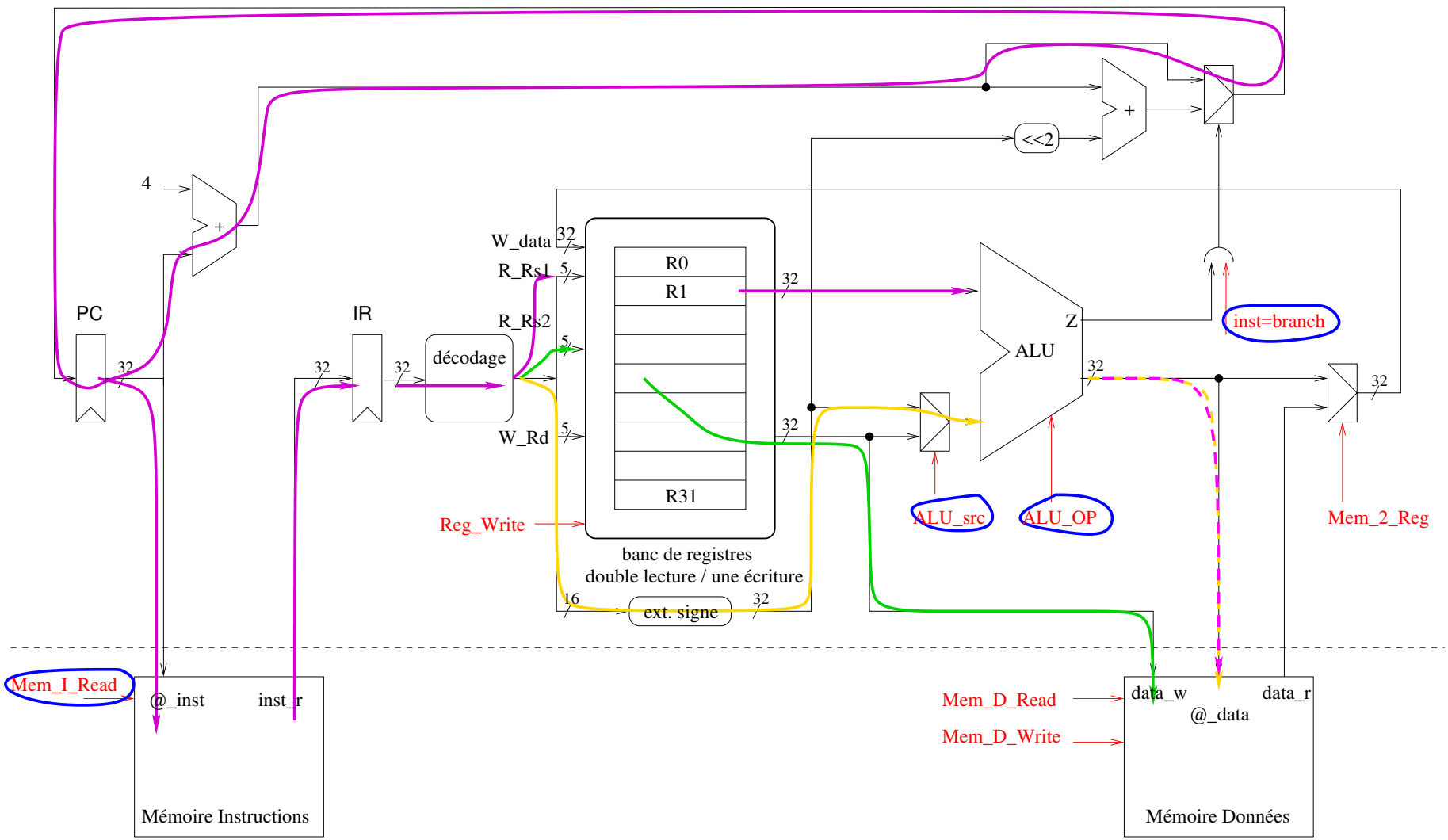
Chemin complet (non pipeliné)



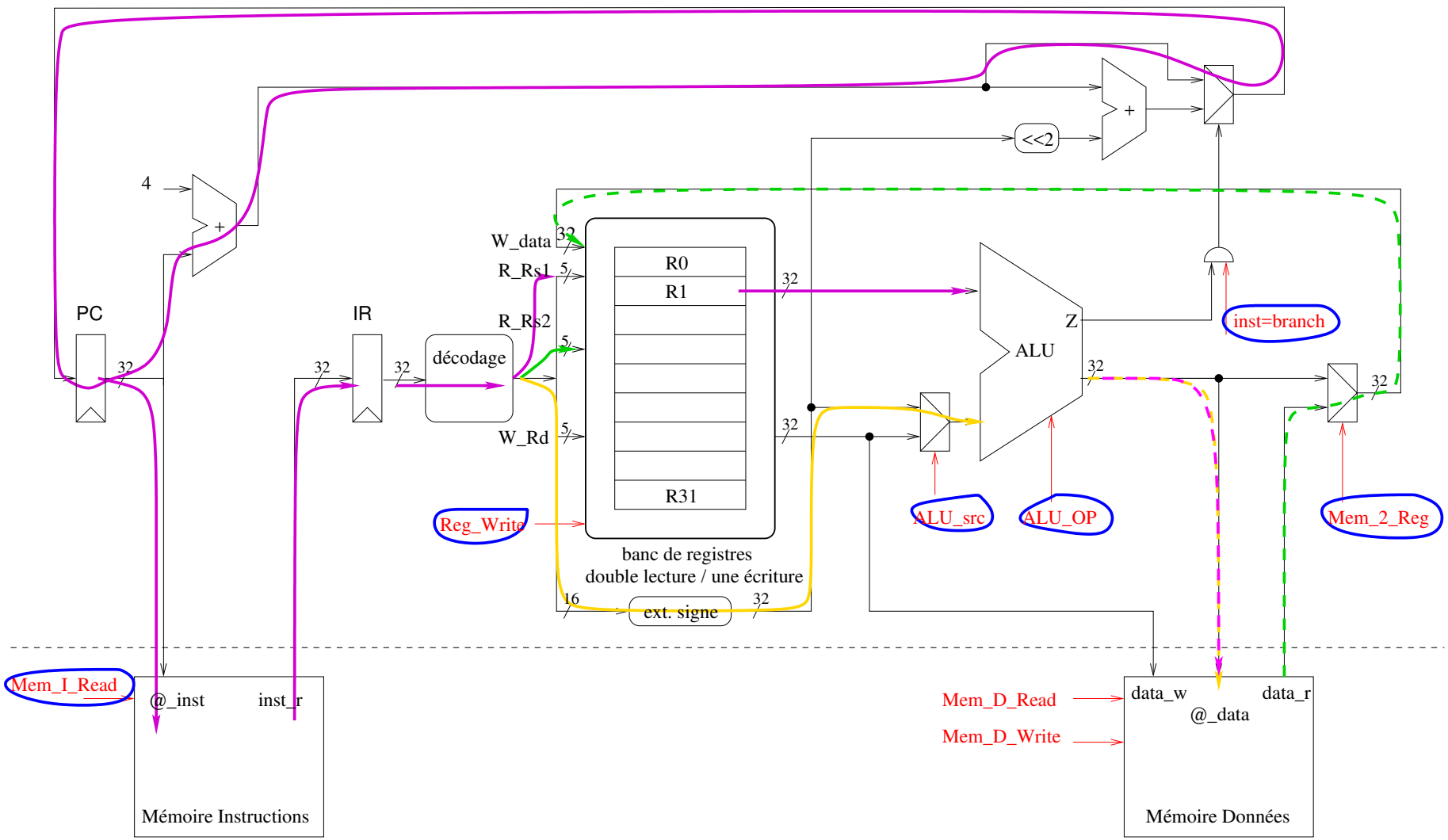
Chemin complet : exécution d'une ALUOP



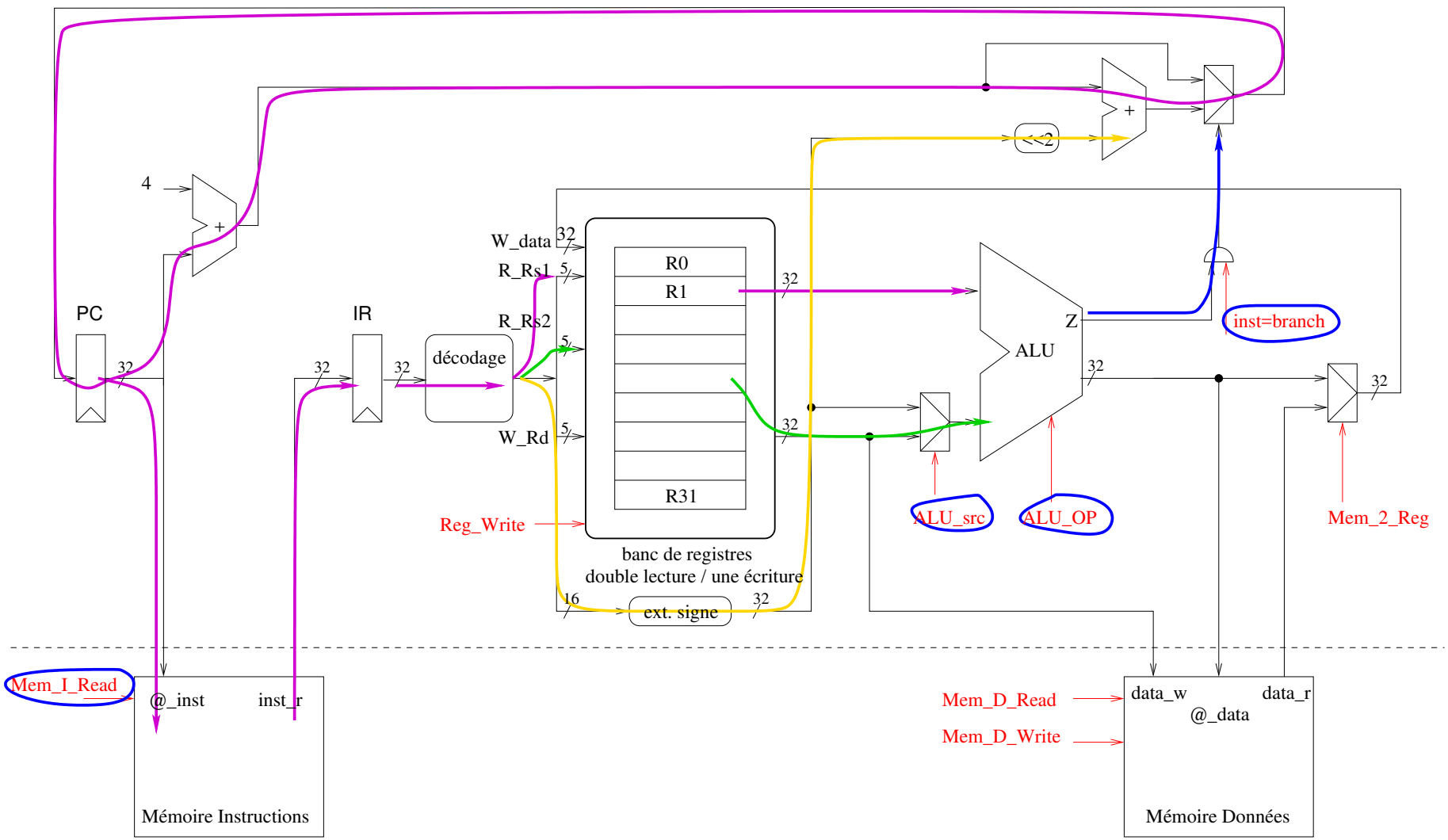
Chemin complet : exécution d'une SW



Chemin complet : exécution d'une LW



Chemin complet : exécution d'une BEQ



Chemin complet : limitations

- Temps d'exécution d'une instruction : passage par toutes les unités fonctionnelles nécessaires
- A un instant donné, l'intégralité du chemin de données est utilisée pour une seule instruction en cours d'exécution, même si toutes les unités fonctionnelles ne sont pas nécessaires (notamment lors des LW, SW, BEQ)
- passer à l'instruction suivante -> attendre la fin de l'instruction en cours
- Conséquences : temps par instruction est long et pas de recouvrement d'instruction possible

Principe du pipeline

cf. “Computer Architecture, A Quantitative Approach”, Henney-Patterson 2018, annexe 3.

but

- Améliorer le temps d’exécution d’un programme en permettant que plusieurs instructions utilisent simultanément le chemin de données (mais des parties fonctionnelles différentes au même instant),
- Réduire la période d’horloge en minimisant les chaines combinatoires entre registres,
- Obtenir une performance CPI proche de 1 (CPI : nbr de cycles exec programme / nbr instructions exécutées).

Moyens

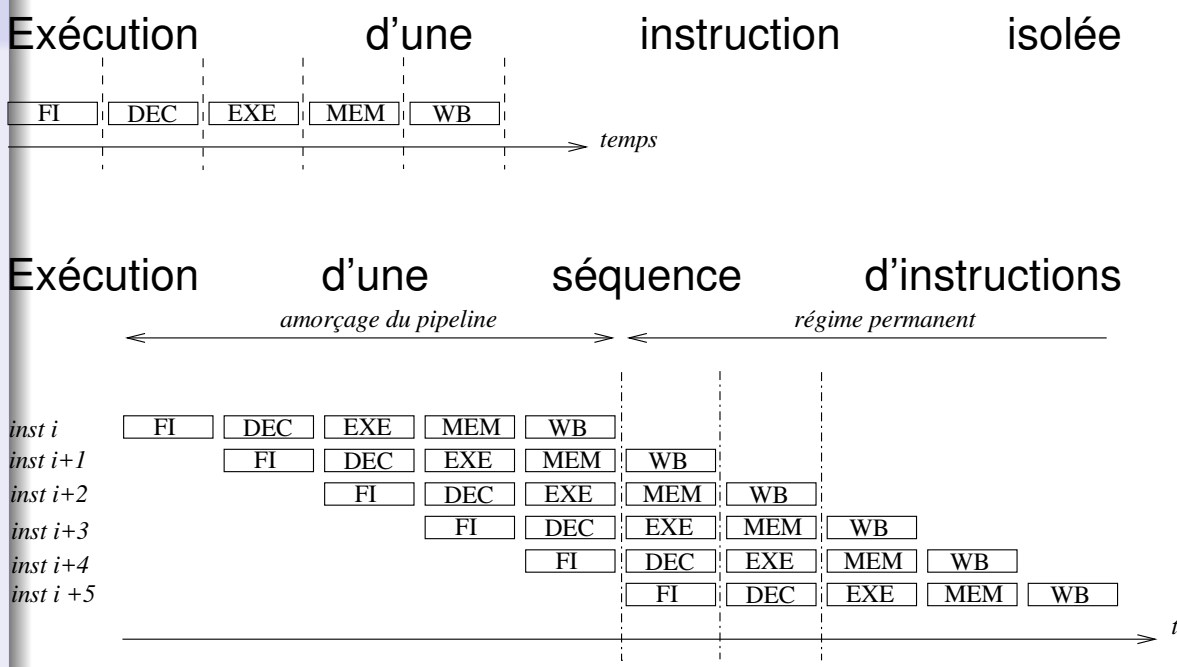
- Fractionnement du chemin de données en zones distinctes (étages) séparées par des barrières de registres (pipeline).
- Chaque instruction utilise à un cycle donné les unités fonctionnelles d’un étage du pipeline.
- Chaque instruction doit passer par toutes les étapes du pipeline pour conclure son exécution.
- Á un cycle donné, les différents étages du pipeline sont utilisés par différentes instructions.

Adaptation du chemin de données pour fonctionnement pipeline

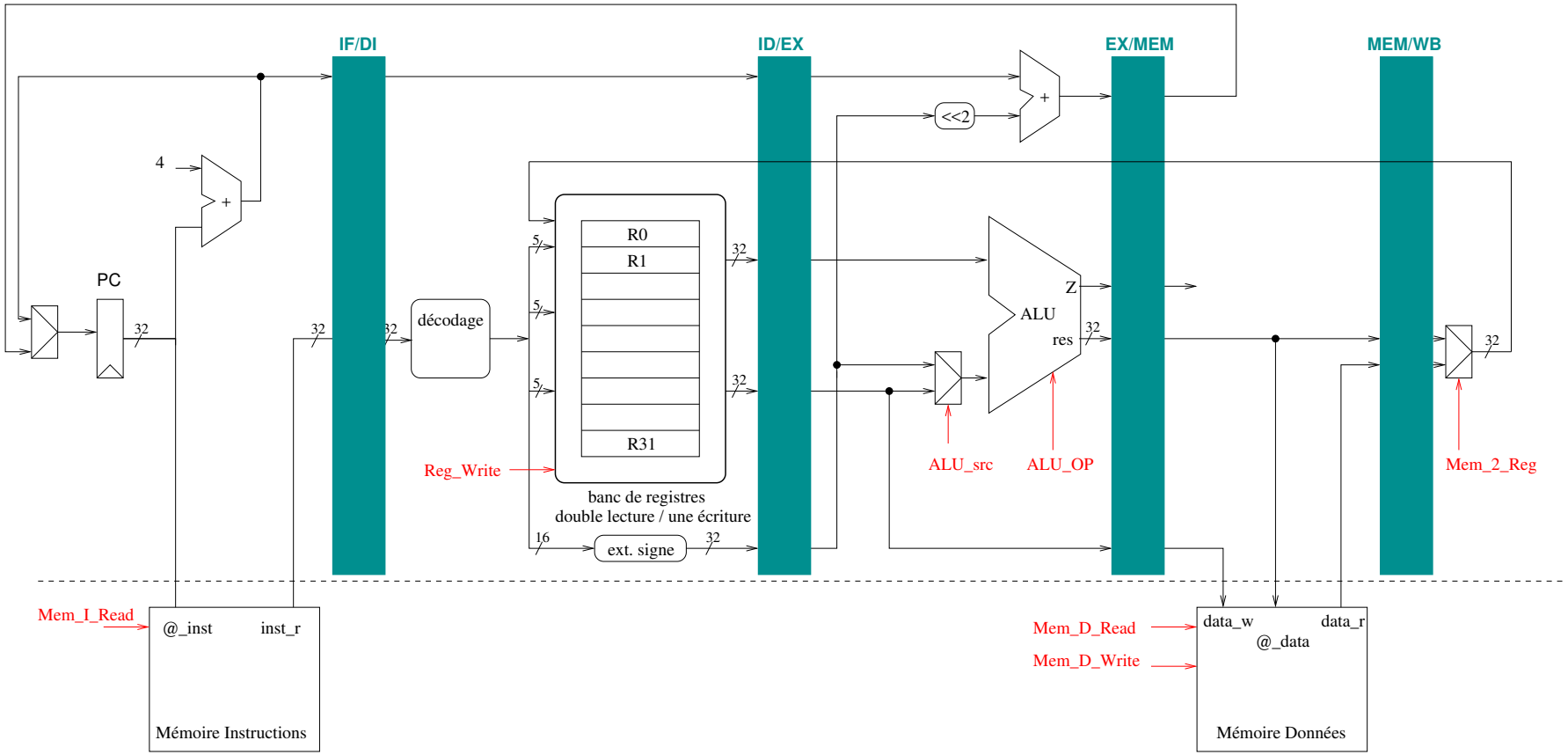
Pipeline à 5 étages

L'exécution d'une instruction est divisée en 5 étapes séquentielles :

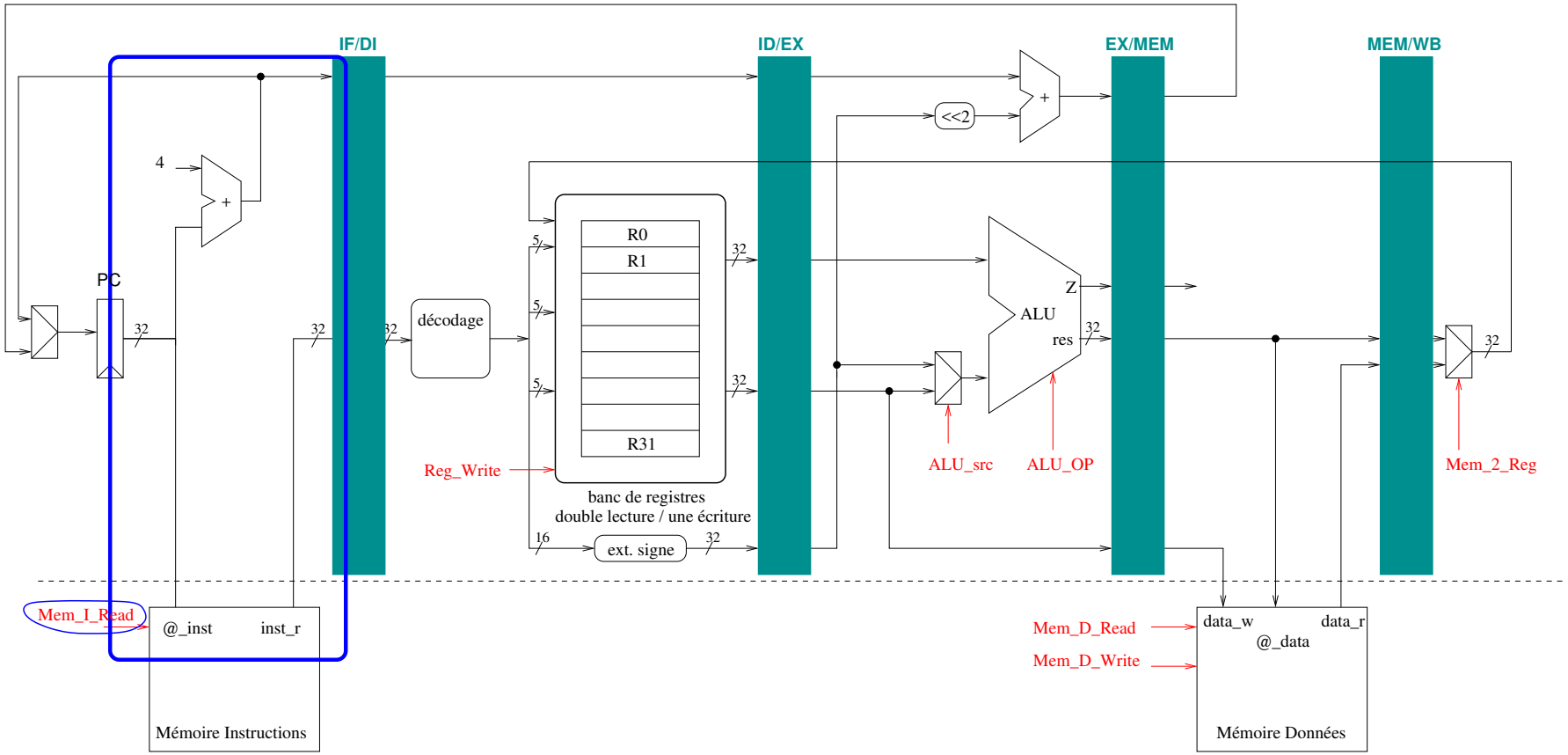
1. Fetch Instruction (FI)
2. Decode Instruction (DEC)
3. Execute (EXE) - passage par l'ALU
4. Accès à la mémoire de données (MEM) (pour LW et SW)
5. WriteBack (WR) écriture dans le banc de registres (pour SW et ALUOP)



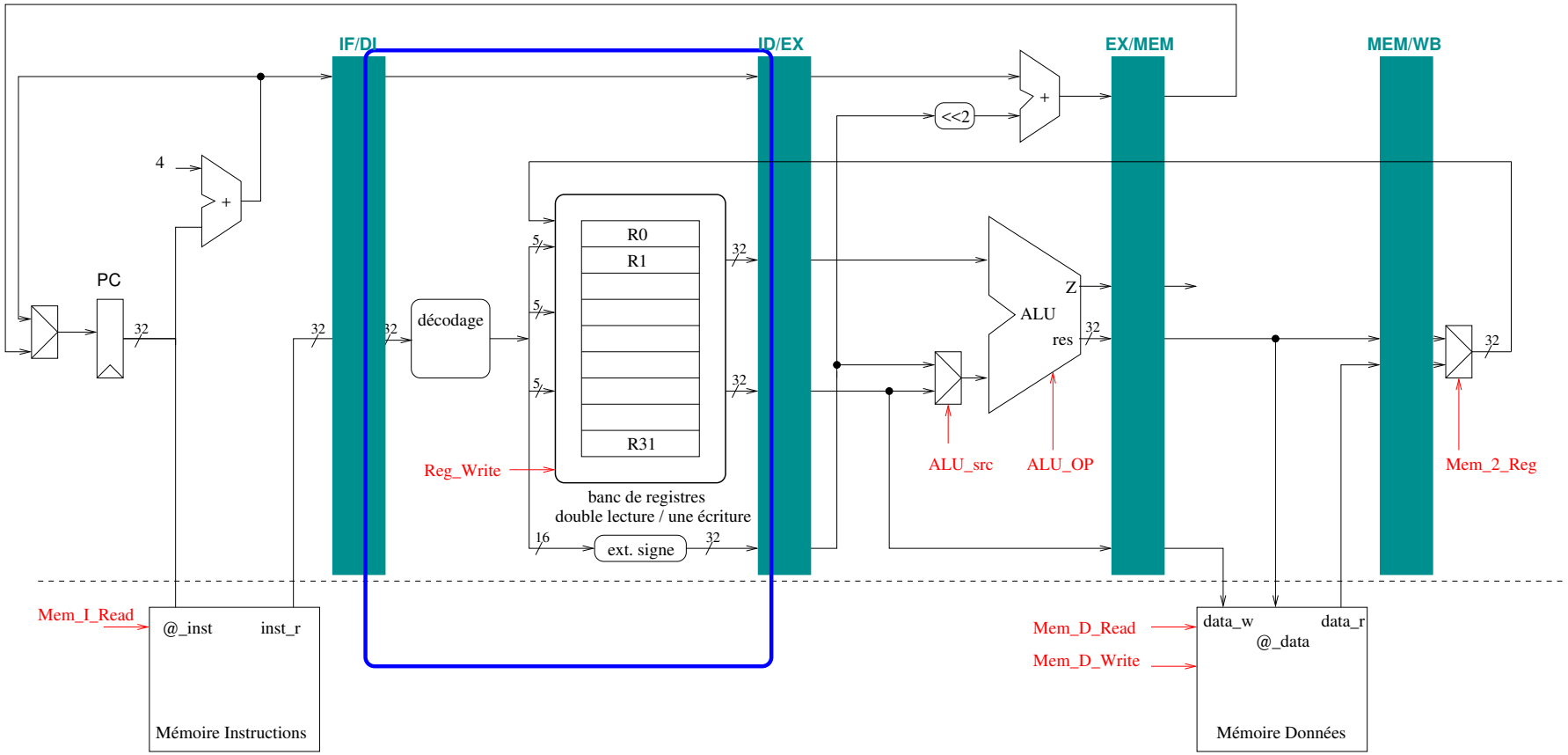
Chemin complet (pipeliné)



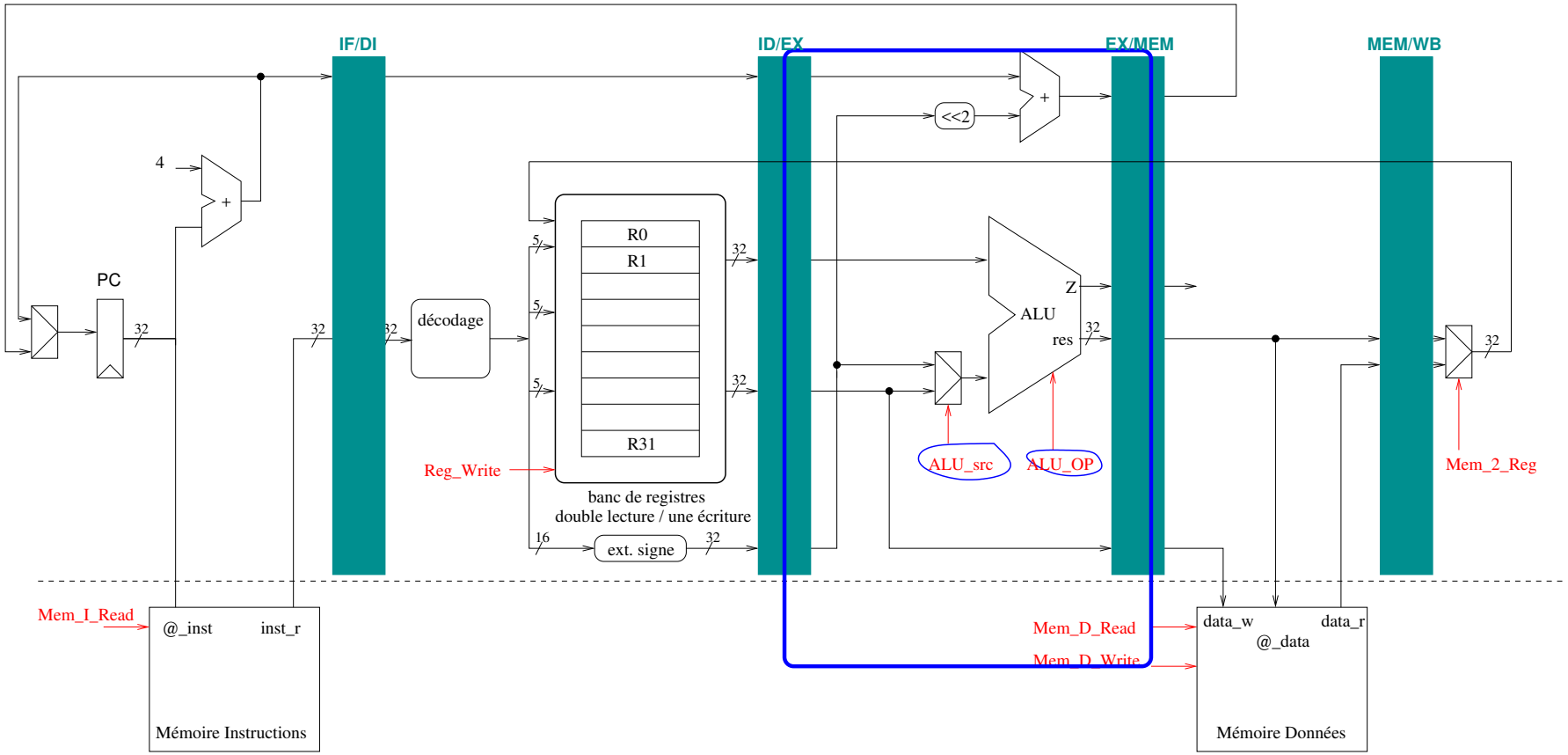
Chemin complet pipeliné - FI



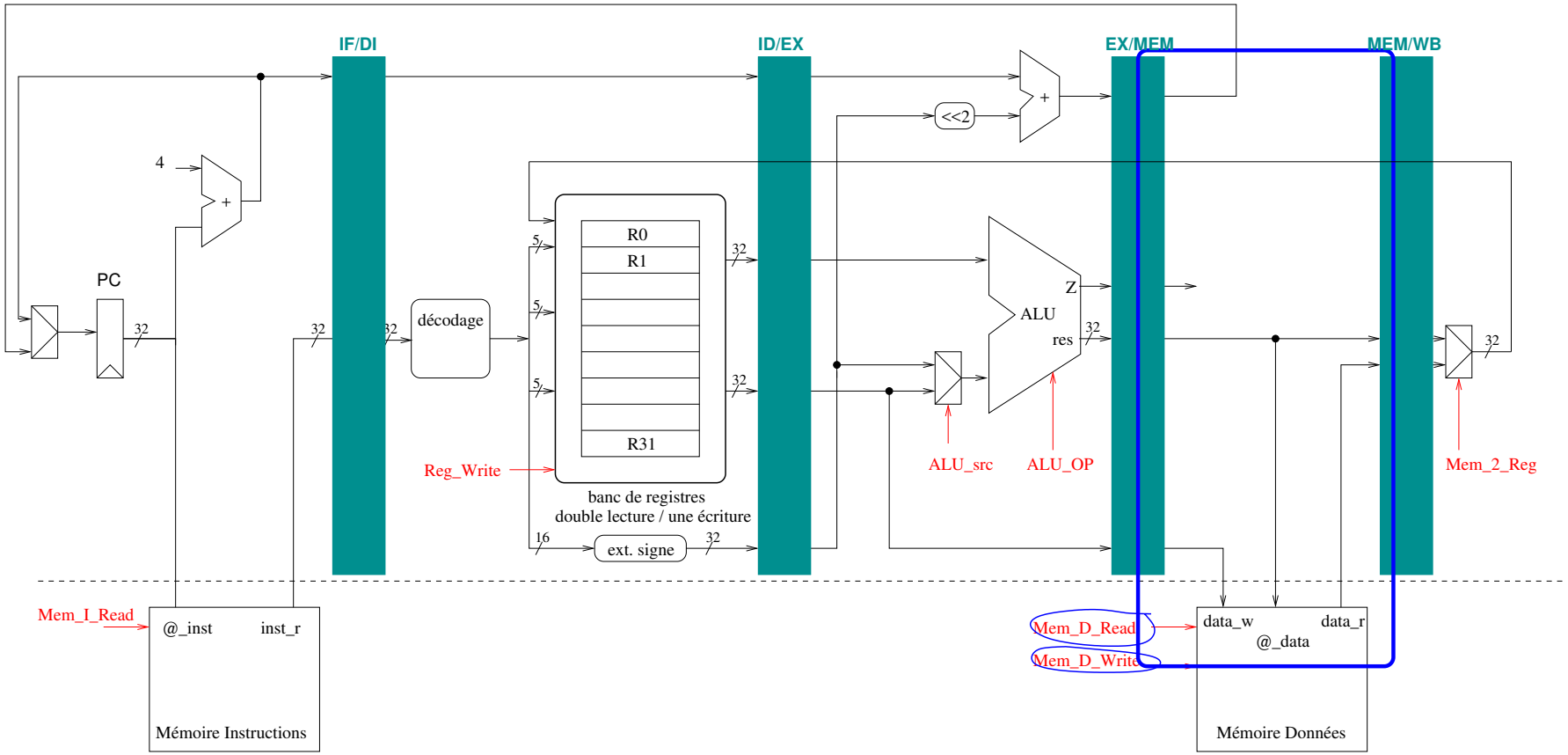
Chemin complet pipeliné - DEC



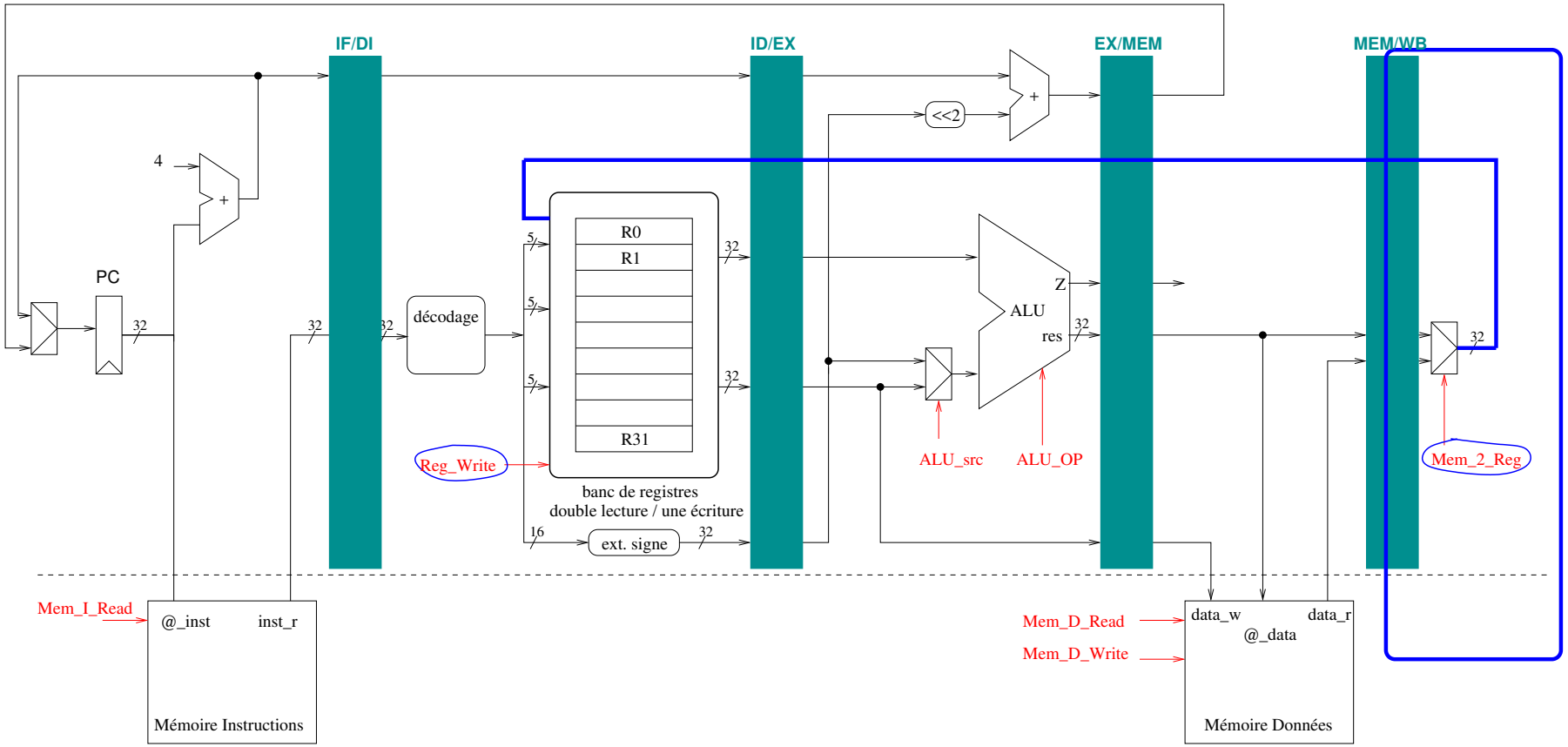
Chemin complet pipeliné - EXE



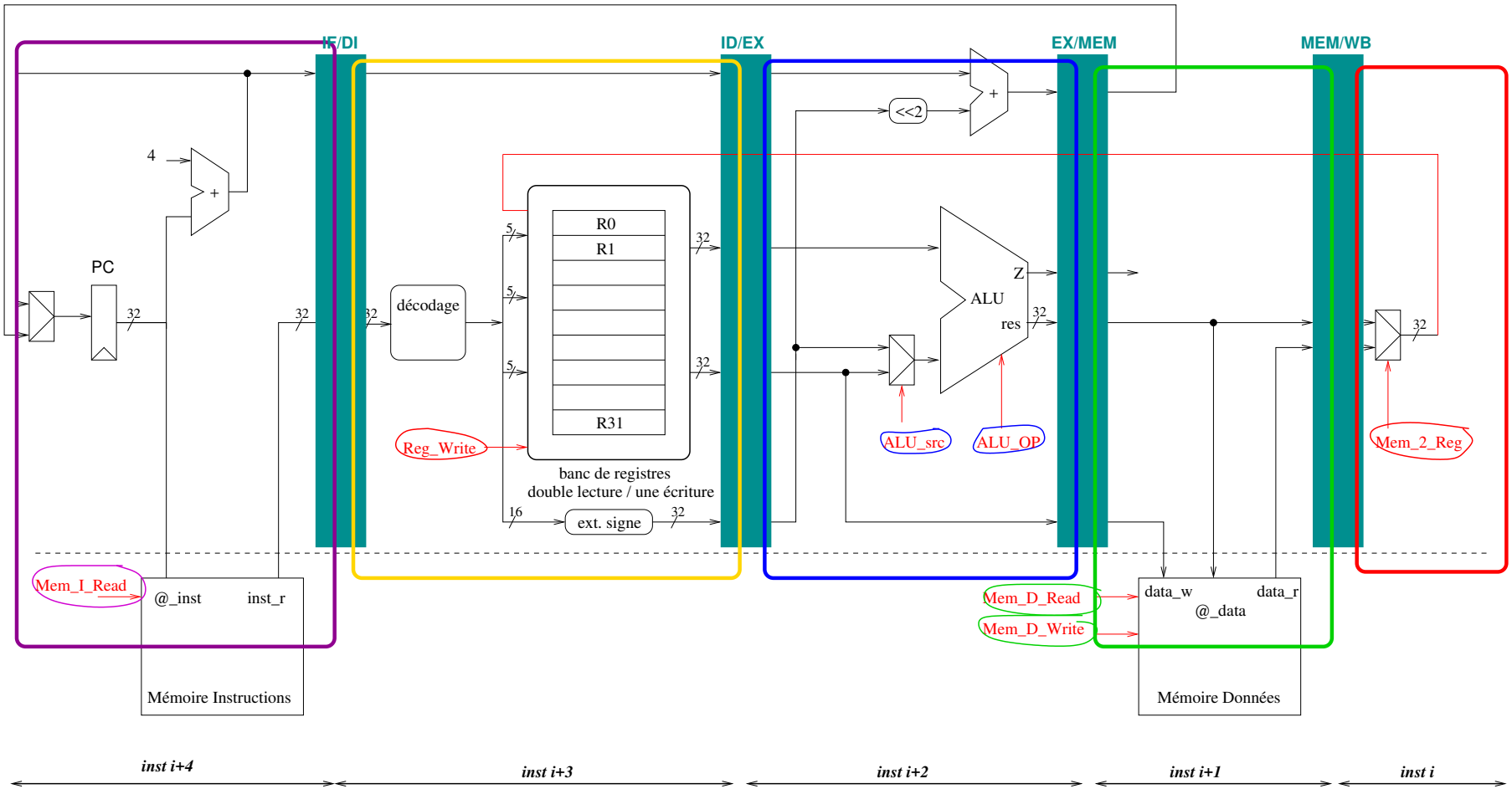
Chemin complet pipeliné - MEM



Chemin complet pipeliné WB



Chemin complet pipeline - Fonctionnement nominal



Irrégularités dans un pipeline - Causes

- Dépendances entre des instructions consécutives
- Sauts et branchement (rupture de séquence)
- Interruptions / exceptions
- Unités fonctionnelles prenant plus d'un cycle (multiplications et divisions entières, opérations sur les flottants)

Irrégularités dans un pipeline - Remédiations

- Matériel :
  - Forwarding : ajout de chemins entre différents étages du pipeline
  - Gels : empêche l'avancement des étages
  - Anticipation de saut (branch prediction)
  - Exécution desordonnée, ...
- Compilation : réordonnancement d'instructions, ...

Optimisation de l'addition entière

L'addition est une opération essentielle. Le temps de réalisation de l'addition est un facteur important de performance. Plusieurs schémas d'additionneurs ont été proposés, ils sont combinables pour répondre au mieux à chaque problème (addition entière, calcul d'adresse, somme de produits partiels dans la multiplication, étape intermédiaire dans les calculs flottants, ...) (cf. annexe J de l'ouvrage "Computer Architecture, a Quantitative Approach", Hennessy-Patterson)

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

Additionneur à sélection de retenue (carry select adder)

double la surface du matériel

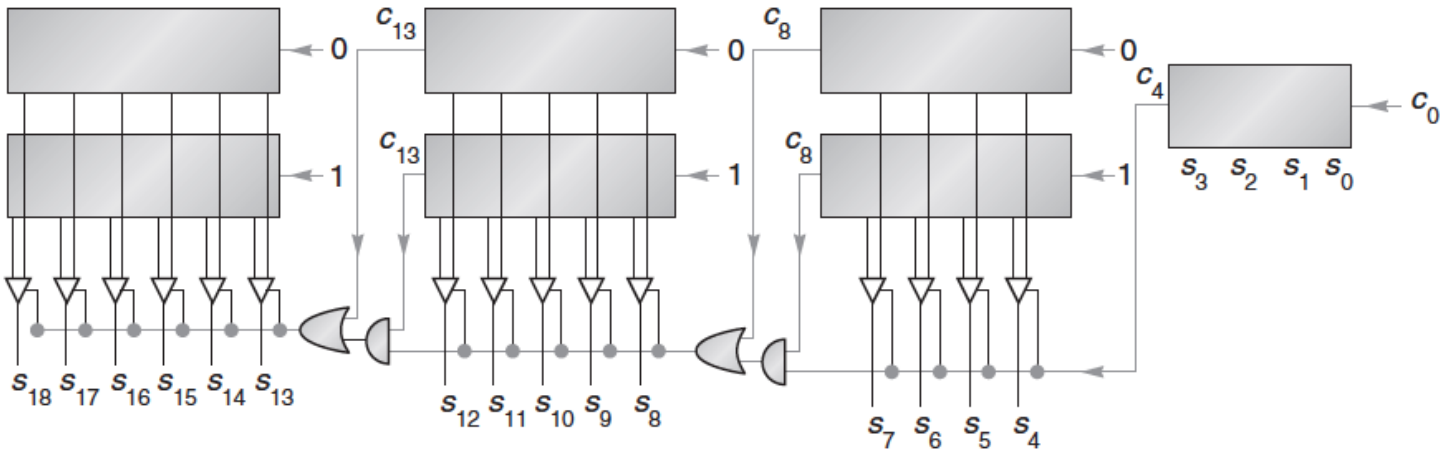


Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Additionneur à saut de retenue (carry skip adder)

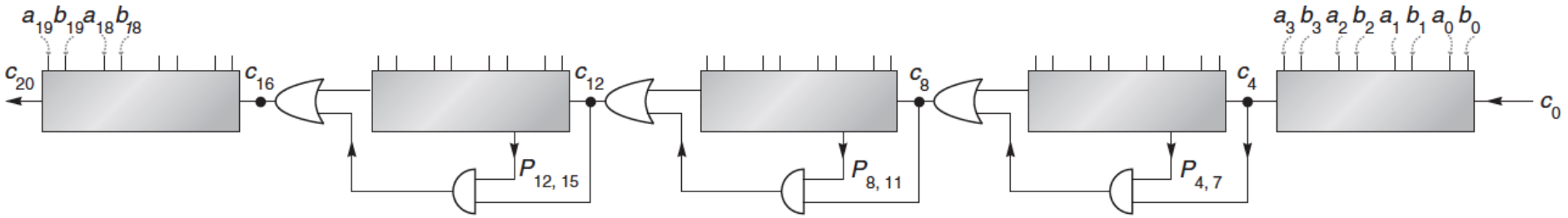
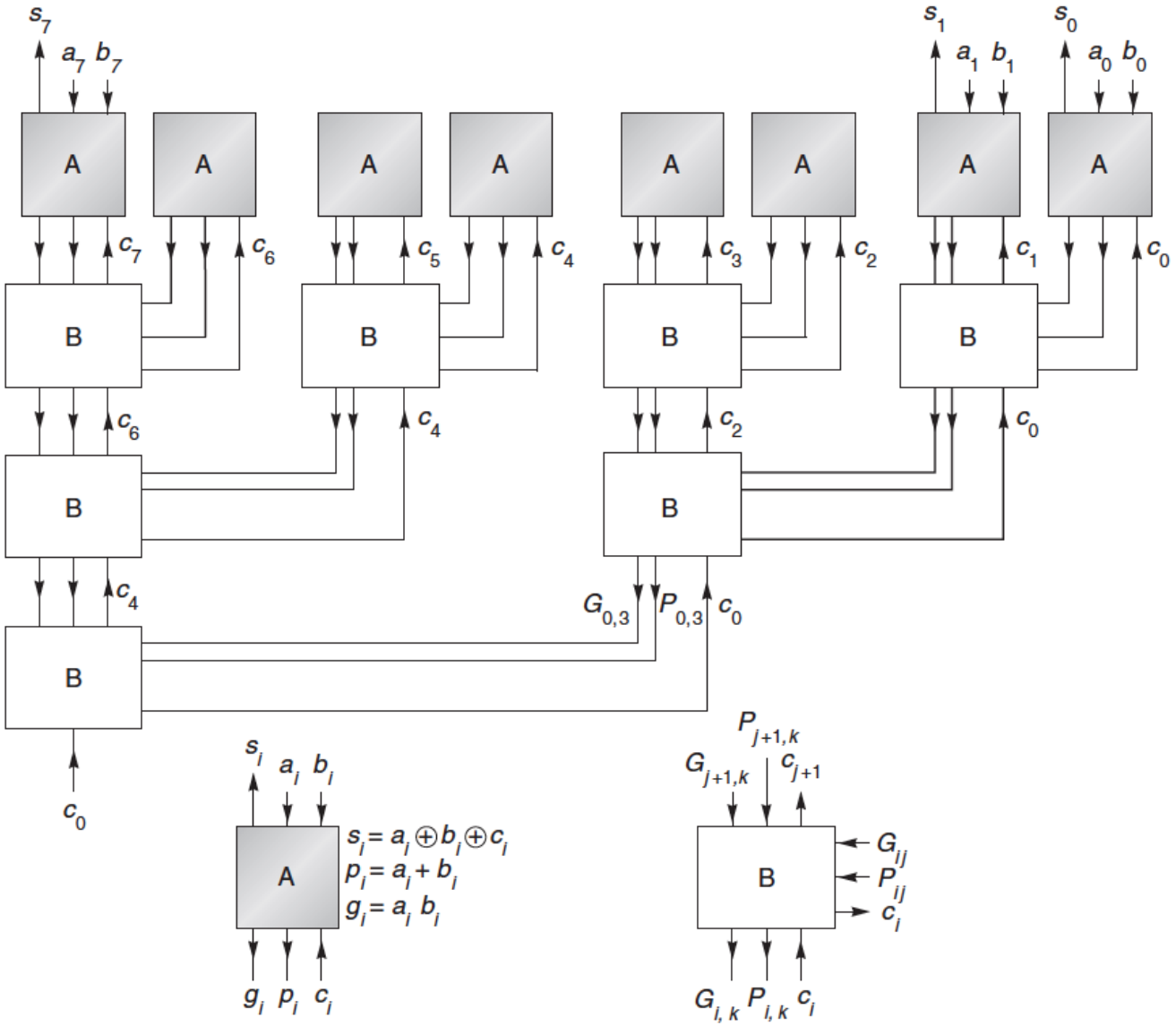
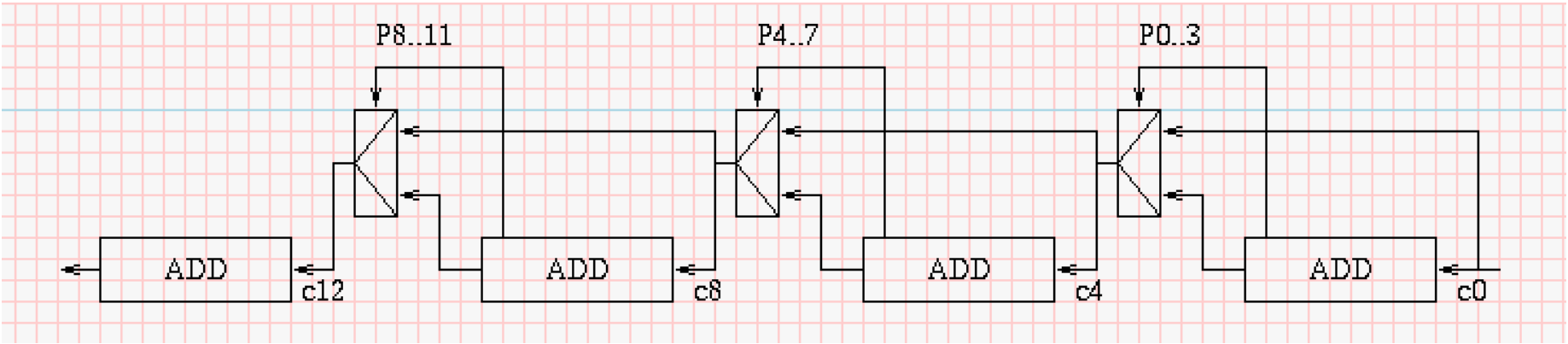


Figure J.18 Carry-skip adder. This is a 20-bit carry-skip adder ($n = 20$) with each block 4 bits wide ($k = 4$).

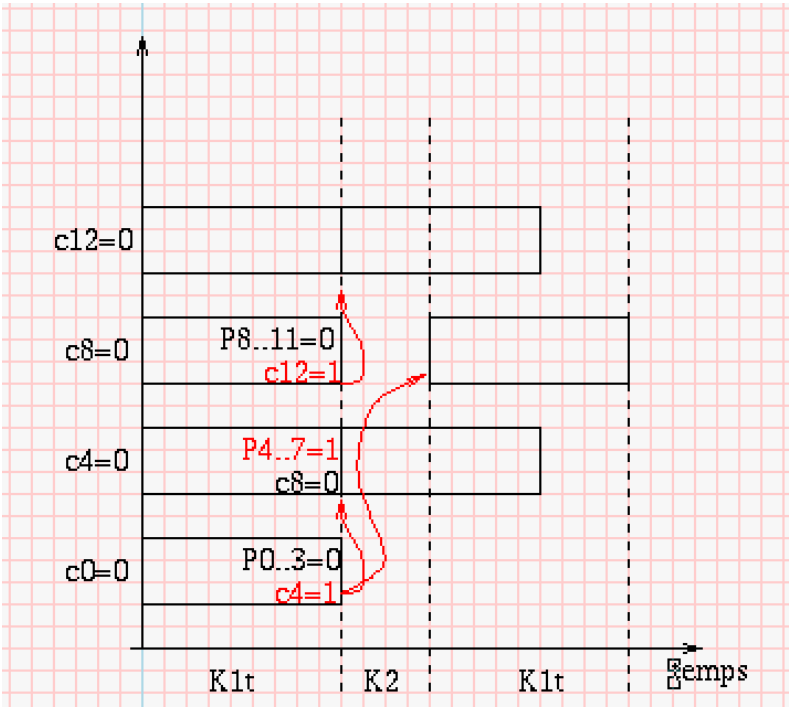
Additionneur à retenue anticipée (carry lookahead adder (CLA))



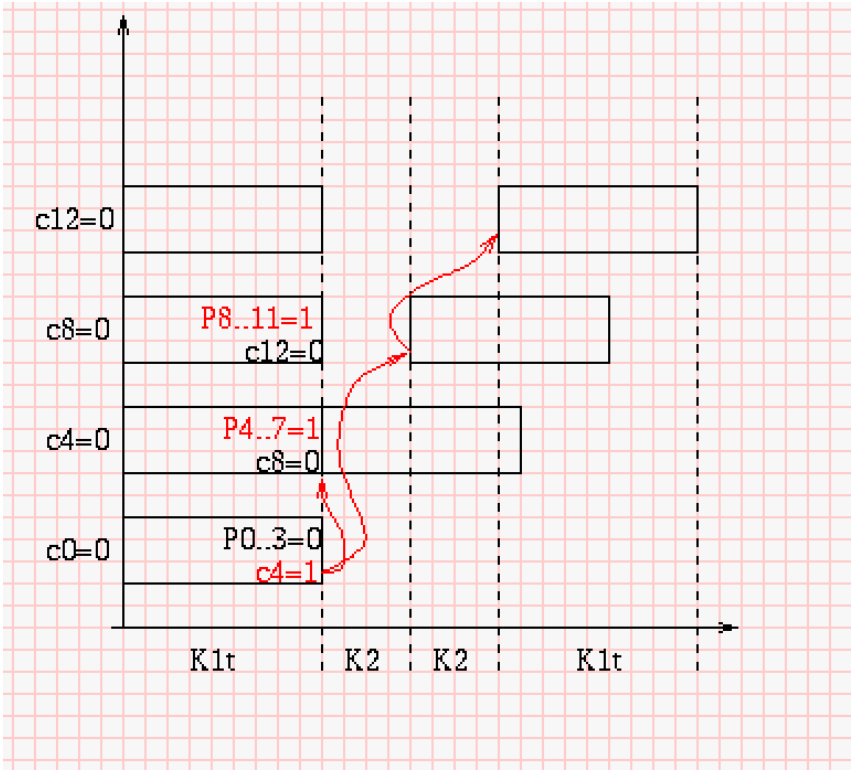
Additionneur à saut de retenue (carry skip adder)



exemples d'exécution



ex :
addition de 0000 1101 1110 1011
et 0001 1011 0001 0110



ex :
addition de 0000 0101 1110 1011
et 0001 1010 0001 0110