Processeur

PRÉSENTATION

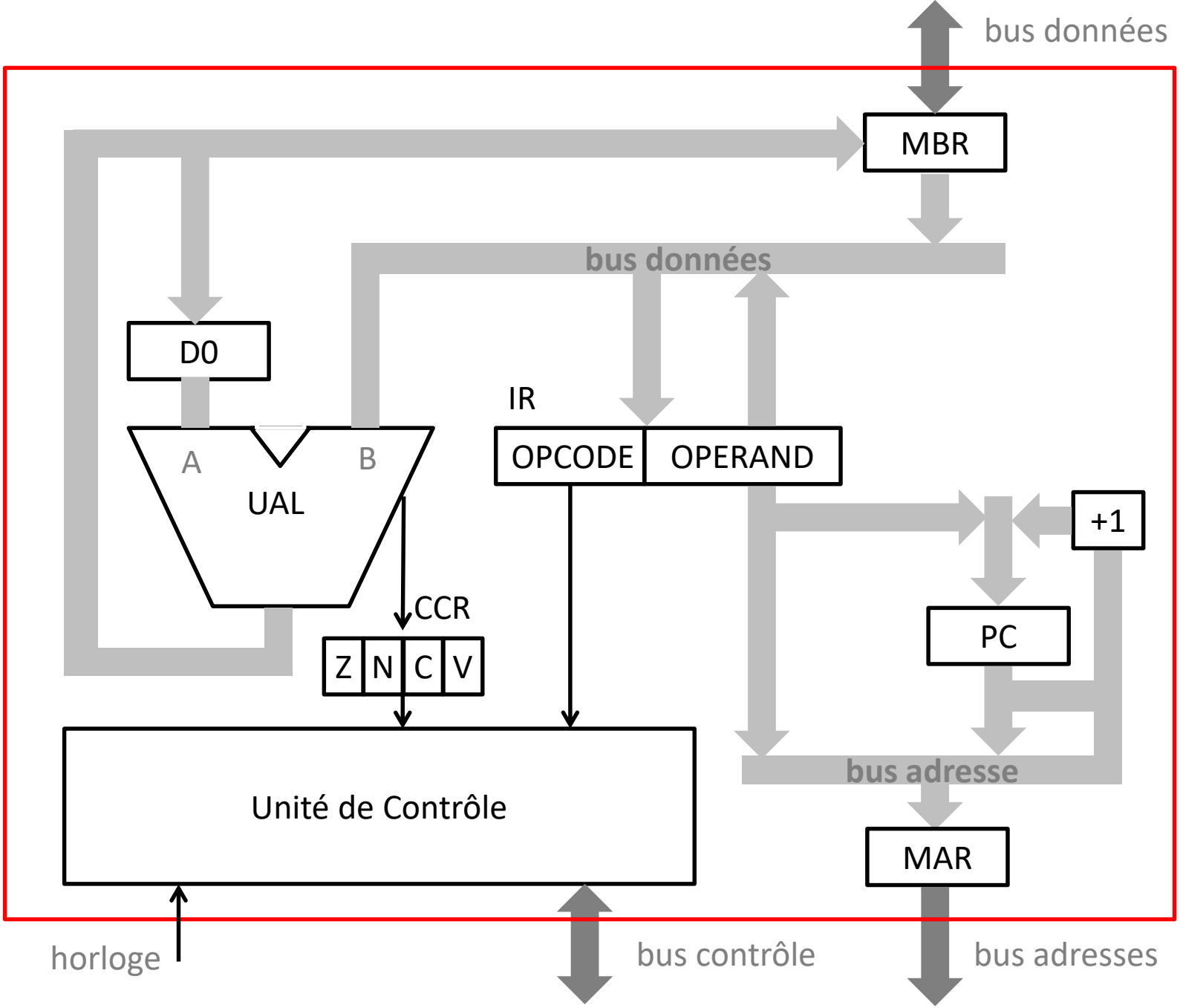
Rôle du processeur

- Le processeur est la partie intelligente et active d'un ordinateur
- Il exécute les instructions des différents programmes chargés en mémoire
- Il s'agit d'instructions machine, i.e. de bas niveau et directement compréhensibles par le processeur
  - résultent de la compilation d'un programme en langage de haut niveau
  - très simples : lecture valeur, addition autre valeur, écriture résultat

- La gamme des processeurs existants est vaste. Ils diffèrent sur :
  - leur jeu d'instructions
  - leur architecture interne
  - et donc leurs performances

Composants internes

- Registres
  - éléments de mémorisation (par ex. avec des verrous D)
  - en général spécialisés : adresse, opérande, instruction, etc.
- Unité Arithmétique et Logique (UAL)
  - réalise effectivement les calculs sur les données
  - simple circuit combinatoire
- Unité de contrôle
  - chef d'orchestre du processeur, pilote tous les autres composants
  - automate matériel : entrées, état courant, sorties, synchronisé sur l'horloge
- Le tout interconnecté par
  - un bus interne : circulation des données, des adresses, etc.
  - des signaux de contrôle : activation UAL, écriture dans registre, etc.


bus données
MBR
bus données
D0
IR
A
B
UAL
OPCODE
OPERAND
+1
CCR
Z
N
C
V
PC
bus adresse
Unité de Contrôle
MAR
horloge
bus contrôle
bus adresses

Registres

- MAR : Memory Address Register
  - stocke l'adresse mémoire à laquelle accède le processeur
  - elle doit être stable pendant tout le cycle de lecture ou écriture
- MBR : Memory Buffer Register
  - stocke le mot mémoire lu ou à écrire en mémoire
  - même remarque que ci-dessus : stabilité nécessaire
- PC : Program Counter
  - stocke l'adresse de la prochaine instruction à exécuter
  - initialisé à 0x00000000 au démarrage du processeur
- IR : Instruction Register
  - stocke l'instruction en cours d'exécution
  - divisé en deux champs distincts : code opération, opérande

Registres (2)

- D0
  - stocke l'opérande et le résultats d'un calcul
  - registre accumulateur
- Condition Code Register (CCR)
  - stocke les bits résumant le dernier calcul fait par l'UAL
    - **Z**ero : le résultat est 0
    - **N**egative : le résultat est négatif (strictement)
    - **C**arry : le résultat a généré une retenue sortante
    - o**V**erflow : le résultat a généré un débordement arithmétique
  - ces bits sont :
    - positionnés par les instructions arithmétiques et logiques
    - exploités par les instructions de branchement conditionnel (voir plus loin)

Unité Arithmétique et Logique

- Rôle :
  - effectue un calcul sur les opérandes présents sur ses entrées
  - produit un résultat en sortie et positionne les bits du CCR
- Calculs possibles :
  - arithmétiques : addition, soustraction, multiplication, division
  - logiques : et, ou, ou exclusif, non
  - **rien** : l'UAL retransmet en sortie un de ses opérandes
- L'UAL est un composant passif, elle ne décide pas :
  - de l'opération à effectuer
  - de l'origine des opérandes, de la destination du résultat
  - tout cela est entièrement piloté par l'unité de contrôle, au travers de signaux de contrôle

Unité de contrôle

- Rôle :
  - pilote entièrement le processeur
  - gère l'exécution de chaque instruction, étape par étape, et le passage à l’instruction suivante
- Entrées :
  - Instruction Register, partie code opération
  - le signal horloge, cadençant l'enchainement des opérations
  - signaux de contrôle du bus externe : fin de cycle mémoire, etc.
- Sorties : signaux de contrôles :
  - opération de l'UAL
  - écriture dans un registre interne
  - ouverture des différents segments du bus interne
  - pilotage du bus externe : demande d'un cycle lect. ou ecr., etc.

JEU D'INSTRUCTIONS

Définition

- Le jeu d'instruction d'un processeur est l'ensemble des instructions qu'il sait exécuter

- Il est propre au processeur, et fonction de son architecture interne, de ses registres, etc.

- Dans notre processeur très simple :
  - un seul registre de données : D0
  - à la fois opérande d'un calcul et destination du résultat
  - c'est ce qu'on appelle un registre **accumulateur**
  - la majorité des instructions s'y réfèrent **implicitement**

Format d'une instruction

- Les instructions sont nombreuses mais suivent un petit nombre de formats prédéfinis

- Dans notre processeur très simple, il y a deux formats :
  - code-opération opérande
    - suivi par presque toutes les instructions
  - code-opération
    - instruction STOP

- Code-opération :
  - indique ce que doit faire l'instruction : additionner, charger, stocker, etc.
- Opérande :
  - indique ce sur quoi doit agir l'instruction : valeur, adresse, etc.

Catégories d'instructions

- Instructions **de transfert** depuis / vers la mémoire
  - LDA opérande : charge D0 avec une valeur
  - STA opérande : range la valeur de D0 en mémoire

- Instructions **arithmétiques et logiques**
  - ADD opérande : additionne une valeur à D0 : D0 = D0 + valeur
  - SUB opérande : soustrait une valeur à D0 : D0 = D0 – valeur
  - MUL opérande : multiplie D0 par une valeur : D0 = D0 x valeur
  - DIV opérande : divise DO par une valeur : D0 = D0 / valeur

Catégories d'instructions (2)

- Instruction de **comparaison**
  - CMP opérande : compare D0 à une valeur : D0 – val
  - similaire à une soustraction SUB
    - le CCR est modifié
    - **mais le résultat n'est stocké nulle part**
- Instructions de **branchement**
  - BR**x** adresse (**x** = Z, N, C ou V) : branchement conditionnel
    - **si** x est à 1, poursuit l'exécution du programme à l'instruction située à *adresse* en mémoire
    - sinon, poursuit l'exécution du programme à l'instruction située après l'instruction de branchement
  - BRA adresse : branchement inconditionnel (Always)

    poursuit l'exécution du programme à l'instruction située à *adresse* en mémoire

Catégories d'instructions (3)

- Autres instructions (exemples)
  - STOP : met le processeur en attente (d'une interruption extérieure)
  - voir chapitres suivants :
    - gestion des langages de programmation de haut niveau
    - interface avec le système d'exploitation
    - pilotage des périphériques

Mode d'adressage

- La plupart des instructions ont un champ **opérande**
  - il contient une valeur N quelconque, rangée après le code opération

- La plupart des instructions ont deux façons d'interpréter N :
  - soit N est le **paramètre véritable** de l'opération à effectuer
    - exemple : ADD 123 : ajoute la valeur 123 à D0
  - soit N est l'**adresse mémoire du paramètre** de l'opération
    - exemple : ADD 123456 : ajoute à D0 la valeur située à l'adresse 123456

- Ces différentes interprétations du champ opérande s'appellent des **modes d'adressage**

Mode d'adressage (2)

- Comment distinguer les modes d'adressages
  - dans la notation des instructions en assembleur ?
  - dans leur codage en mémoire centrale ?

- Notation en assembleur :
  - ADD **#**123 : mode d'adressage **immédiat**
    - 123 est le valeur à ajouter à D0
  - ADD 123 : mode d'adressage **direct mémoire**
    - 123 est l'adresse mémoire où trouver la valeur à ajouter à D0

- Codage de l'instruction :
  - un champ de bits est réservé dans le **code opération** (pas l'opérande)
  - ici un bit suffirait, par exemple : 0 = immédiat, 1 = direct mémoire

TRADUCTION DE PROGRAMME

Traduction d'un programme en assembleur

- Traduire un programme écrit dans un langage de haut niveau (par exemple C) en assembleur nécessite :
  - de traduire les instructions du langage en instructions processeur
  - de placer les variables du programme en mémoire : dans la suite, on suppose pour simplifier que l'adresse de chaque variable est connue

- Pour effectuer des branchements, il faut connaître l'adresse des instructions processeur du programme. On fait deux hypothèses simplificatrices :
  - le programme assembleur est chargé en mémoire à l'adresse 0
  - toutes les instructions processeur ont une taille de 1 octet

Exemple de traduction – Construction if

Programme source :

```
int n;
// …
if (n < 1) {
        n -= 1;
}
```

Hypothèse : la variable n est placée à l'adresse 100

Traduction assembleur :

```
0  LDA   100   ; D0 = n
1  CMP   #1    ; D0 – 1 (i.e. n – 1)
2  BRN   4     ; aller à SUB
3  BRA   6     ; aller à la fin
4  SUB   #1    ; D0 = D0 - 1
5  STA   100   ; n = D0
```

Exemple de traduction – Construction for

Programme source :

```
int s = 0;
for (int i = 1; i <= 10; i++) {
        s += i;
}
```

Hypothèse : s à l'adresse 100, i à l'adresse 101

Traduction assembleur :

```
0   LDA   #0     ; D0 = 0
1   STA   100    ; s = D0
2   LDA   #1     ; D0 = 1
3   STA   101    ; i = D0
4   CMP   #11    ; D0 – 11
5   BRZ   11     ; si 0, aller à la fin
6   LDA   100    ; D0 = s
7   ADD   101    ; D0 = D0 + i
8   STA   100    ; s = D0
9   LDA   101    ; D0 = i
10  ADD   #1     ; D0 = D0 + 1
11  STA   101    ; i = D0
12  BRA   2      ; aller à CMP
13  STOP
```

Optimisation de code

- Le code peut être optimisé selons deux axes :
  - Optimisation de la taille du code : certaines instructions inutiles peuvent être éliminées, des séquences d'instruction en double peuvent être factorisées.

    Avantage : le programme prend moins de place en mémoire, ce qui permet d'y placer plus de programmes à un instant donné. Le programme prend aussi moins de place sur disque.
  - Optimisation du nombre d'instruction exécutées : certaines instructions inutiles peuvent être éliminées.

    Avantage : l'exécution du programme est plus rapide, pour une de ses branches (if), voire pour toutes ses branches.
  - Ces deux axes sont relativement indépendants : on peut par exemple optimiser la taille du code sans optimiser le nombre d'instruction exécutées (par ex. factorisation du code).

Version optimisée

Programme source :

```
int s = 0;
for (int i = 1; i <= 10; i++) {
        s += i;
}
```

Hypothèse : s à l'adresse 100, i à l'adresse 101

Traduction assembleur :

```
0   LDA   #0     ; D0 = 0
1   STA   100    ; s = D0
2   LDA   #1     ; D0 = 1
3   BRA   8
4   ADD   100    ; D0 = D0 (= i) + s
5   STA   100    ; s = D0
6   LDA   101    ; D0 = i
7   ADD   #1     ; D0 = D0 + 1
8   STA   101    ; i = D0
9   CMP   #11    ; D0 – 11
10  BRN   4
11  STOP
```

EXECUTION D'UNE INSTRUCTION

Exécution d'une instruction

- Elle se fait en 3 phases, sous le pilotage de l'unité de contrôle :
  1. FETCH : lecture en mémoire l'instruction
     - le processeur va lire l'instruction se trouvant à l'adresse PC
     - il la rapatrie dans le registre IR (Instruction Register)
     - il incrémente PC pour préparer l'instruction suivante
  2. DECODE : décodage de l'instruction lue
     - en fonction du code opération, l'unité de contrôle calcule :
     - les chemins que vont devoir emprunter les données à la phase suivante
     - les opérations qui leur seront (éventuellement) appliquées par l'UAL
  3. EXECUTE : exécution proprement dite de l'instruction
     - les données suivent les chemins (bus interne) calculés précédemment
     - l'UAL effectue l'opération demandée par l'unité de contrôle

  Puis le processeur reprend à la phase 1 pour l'instruction suivante

Accès mémoire durant une instruction

	# accès mémoire	type accès	information
FETCH	1	lecture	instruction
DECODE	0	-	-
EXECUTE	0 ou 1	lecture ou écriture	donnée

Constats :

- les instructions ne sont jamais écrites
- la phase DECODE ne génère aucun accès mémoire
- les seuls accès en phase EXECUTE concernent les données
- la seule écriture possible a lieu en phase EXECUTE
- pas d'accès en phase EXECUTE si mode d'adressage immédiat

Exécution d'une instruction (2)

- Pourquoi faire ce découpage ? Car ces phases :
  - ont généralement un temps d'exécution proche
  - concernent des composants différents du processeur
  - donc on va pouvoir les pipeliner, pour augmenter le débit des instructions

Chemin de données

- Lors des différentes phases d'exécution d'une instruction (à l'exception de DECODE), des données (ou adresses) vont circuler le long des bus internes des processeurs et dans son UAL.
- Le chemin dépend bien sûr de l'architecture interne du processeur : il faut avoir le schéma sous les yeux.
- Le cheminement est décrit au moyen de la notation Register Transfert Language (RTL)
  - chaque étape décrit le transfert d'une donnée d'un registre à un autre, éventuellement en passant par l'UAL
  - les accès mémoire (lecture ou écriture) sont également décrits, bien que ne correspondant pas à une circulation *interne* de données.

Phase FETCH

- Le processeur va chercher en mémoire la prochaine instruction à exécuter :
  1. MAR <- PC
  2. MBR <- Mem[MAR] || PC <- PC + 1
  3. IR <- MBR

- Cette séquence est la même pour toutes les instructions, à la taille de l'instruction près : pour les instructions longues, plusieurs accès mémoire peuvent être nécessaires, en répétant les étapes 2 et 3

Phase DECODE

- Par définition, il n'y a aucune circulation de données pendant cette phase
- Le séquenceur décode l'instruction et à la fin de la phase, les signaux de contrôle à envoyer sont prêts

Phase EXECUTE

- Contrairement à la phase FETCH, le chemin suivi par les données dépend de l'instruction exécutée.

- Exemples développés dans les slides suivants :
  - LDA #123, 123
  - STA 123
  - ADD #123, 123
  - BRx 123

LDA #123

- Rappel : charge le registre D0 avec la valeur 123
- A l'issue de la phase FETCH, le registre instruction contient :

- Il faut donc amener la valeur contenue dans IR<OPERAND> dans DO. On va passer par l'UAL, sans appliquer aucune opération (NOP) sur l'entrée B :

1. DO <- UAL [NOP B] <- IR<OPERAND>

LDA 123

- Rappel : charge le registre D0 avec la valeur contenue en mémoire à l'adresse 123
- A l'issue de la phase FETCH, le registre instruction contient :

- Le cheminement final est le même que précédemment. La différence est que la valeur vient de MBR, après un accès mémoire à l'adresse contenue dans IR<OPERAND> :
  1. MAR <- IR<OPERAND>
  2. MBR <- MEM[MAR] (lecture mémoire)
  3. DO <- UAL [NOP B] <- MBR

STA 123

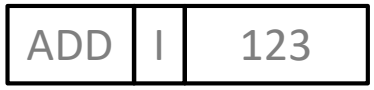
- Rappel : écrit en mémoire, à l'adresse 123, le contenu du registre D0
- A l'issue de la phase FETCH, le registre instruction contient :

- Il faut transférer le contenu de DO dans MBR, en passant par l'UAL, sans opération, puis l'écrire en mémoire à l'adresse contenue dans IR<OPERAND> :

1. MAR <- IR<OPERAND> || MBR <- UAL [NOP A] <- DO
2. MEM[MAR] <- MBR (écriture mémoire)

ADD #123

- Rappel : additionne au registre D0 la valeur 123
- A l'issue de la phase FETCH, le registre instruction contient :



- Il faut donc faire une addition avec les valeurs DO et IR<OPERAND> et stocker le résultat dans D0 :

1. DO <- UAL [A + B] <- DO, IR<OPERAND>

ADD 123

- Rappel : additionne au registre D0 la valeur contenue en mémoire à l'adresse 123
- A l'issue de la phase FETCH, le registre instruction contient :

- Le cheminement final est le même que précédemment. La différence est que le seconde opérande vient de MBR, après une lecture mémoire à l'adresse IR<OPERAND> :
  1. MAR <- IR<OPERAND>
  2. MBR <- MEM[MAR] (lecture mémoire)
  3. DO <- UAL [A + B] <- DO, MBR

BRx 123

- Rappel : poursuit l'exécution du programme à l'instruction située à l'adresse 123 si x égale 1, ou à l'instruction suivante sinon.
- A l'issue de la phase FETCH, le registre instruction contient :

- L'unité de contrôle teste le bit x du CCR :
  - si x == 0, il n'y a rien à faire (la prochaine instruction exécutée sera l'instruction suivante en mémoire centrale, puisqu'on a déjà incrémenté PC lors de la phase FETCH
  - si x == 1, alors il faut transférer l'adresse de branchement dans PC :

    PC <- IR<OPERAND>